# Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Εισαγωγή

Ιστοσελίδα του μαθήματος: https://k08.chatzi.org/

Όλες οι πληροφορίες για βαθμολόγηση, εργαστήρια, εξετάσεις, βιβλιογραφία, συζητήσεις κλπ βρίσκονται εκεί.

## Γιατί είμαστε εδώ;

1. Για να μάθουμε να προγραμματίζουμε. Αλλά **γιατί;**

## Γιατί είμαστε εδώ;

1. Για να μάθουμε να προγραμματίζουμε. Αλλά **γιατί;**

- Γιατί είναι **χρήσιμο**
- Γιατί μαθαίνουμε να σκεφτόμαστε **αλγοριθμικά**
- Γιατί είναι **δημιουργικό**!

*Programming can be like composing poetry or music; [...] especially because it produces objects of beauty.*

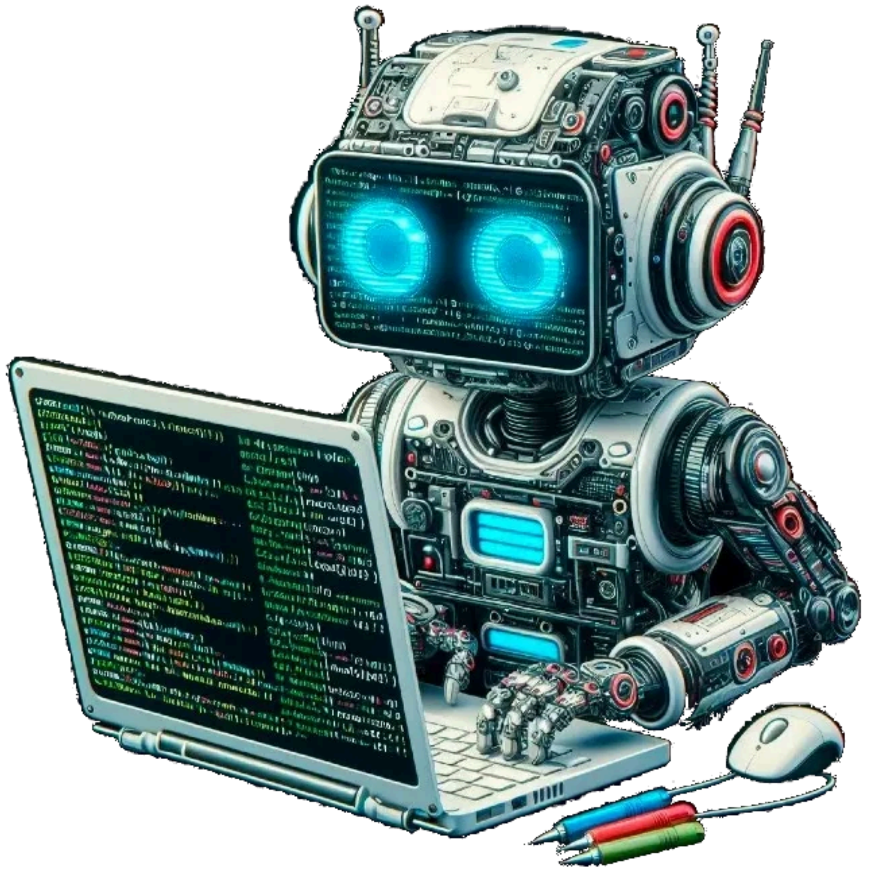
**Donald Knuth**, *CACM, 1974*

*The three virtues of a great programmer:*

- **Laziness***: The quality that makes you go to great effort to reduce overall energy expenditure.*
- **Impatience***: The anger you feel when the computer is being lazy.*
- **Hubris***: Makes you write programs that other people won't want to say bad things about.*

**Larry Wall**, *Programming Perl*

## Μα δε χρειαζόμαστε προγραμματιστές



## Γιατί σε C;

« Κανείς δε γράφει C πλέον, σε όλες τις δουλειές ζητάνε Javascript / Ruby / Python / R / Java / PHP / … »

« Η C είναι δύσκολη στην εκμάθηση »

## Γιατί είμαστε εδώ;

2. Για να μάθουμε να οργανώνουμε τα δεδομένα μας, χρησιμοποιώντας:
    - Αφηρημένους Τύπους Δεδομένων
    - Δομές Δεδομένων

Ας δούμε ένα παράδειγμα…

## Πώς θα αποθηκεύσουμε τα βιβλία μας...

**Πρόβλημα:**

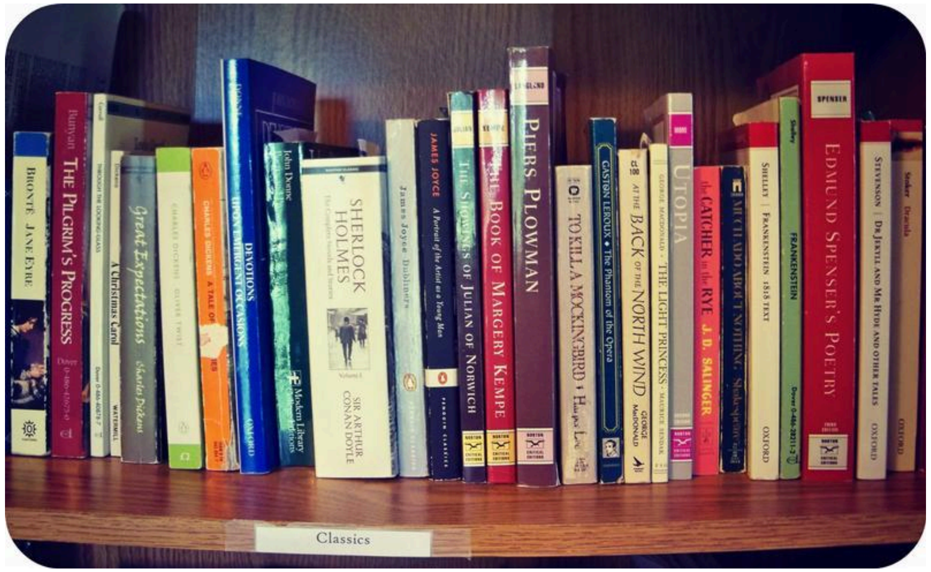
Ποιος είναι ο καλύτερος τρόπος να οργανώσουμε τα βιβλία μας;

## Πώς θα αποθηκεύσουμε τα βιβλία μας…

**Λύση 1** : Χάος

Απλή και εύκολη! (αρκεί να μην μας ενδιαφέρει το διάβασμα)

## Πώς θα αποθηκεύσουμε τα βιβλία μας…



**Λύση 2** : Array

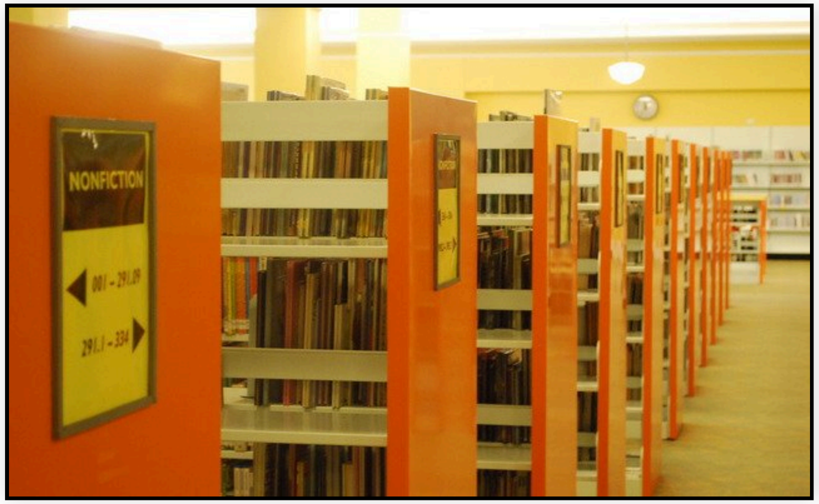
Πλεονεκτήματα; Προβλήματα;

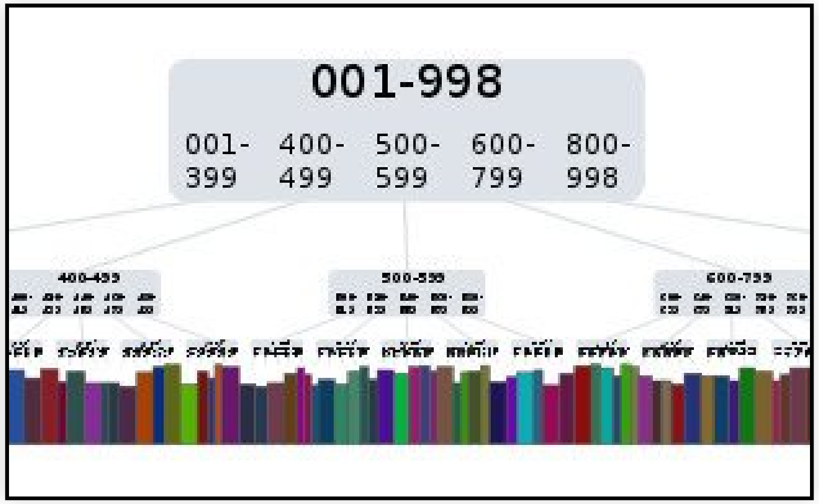
## Πώς θα αποθηκεύσουμε τα βιβλία μας...

**Λύση 3** : Sorted array

Πολύ καλύτερα! Τέλεια λύση; Προβλήματα;

## Πώς θα αποθηκεύσουμε τα βιβλία μας…





**Λύση 4 :** B-tree

Πλεονεκτήματα;

## Αφηρημένοι Τύποι Δεδομένων

### ADTBookStore

- `insert(title)`
  Πρόσθεσε το βιβλίο με τίτλο `title`

- `remove(title)`
  Αφαίρεσε το βιβλίο με τίτλο `title`

- `find(title)`
  Ψάξε το βιβλίο με τίτλο `title`

Ως χρήστες, δανειζόμαστε βιβλία χρησιμοποιώντας αυτή τη **διεπαφή**.
Δε μας απασχολεί πώς είναι αποθηκευμένα.

## Δομές Δεδομένων

4 Δομές Δεδομένων υλοποιούν τον ίδιο **ADTBookStore**

- Chaos
- Array
- SortedArray
- BTree

Κάθε δομή έχει διαφορετικά πλεονεκτήματα & μειονεκτήματα.

Μπορούμε να αλλάξουμε υλοποίηση **χωρίς να επηρεαστούν οι χρήστες**.

# Περιεχόμενο του Μαθήματος

- **Εισαγωγή & τεχνικές αποδοτικού προγραμματισμού**
  - Modules, Makefiles, Editors, Git
  - Recap: memory allocation, pointers, structs, typedefs, void pointers
  - Code style, Tests, Debugging
- **Αφηρημένοι Τύποι Δεδομένων και εφαρμογές**
  - Vectors, Λίστες, Στοίβες, Ουρές
  - Ουρές προτεραιότητας, Maps, Σύνολα
- **Δομές δεδομένων**
  - Δυναμικοί Πίνακες, Συνδεδεμένες Λίστες, Δένδρα, Σωροί
  - Δυαδικά δένδρα αναζήτησης, AVL δένδρα, B-δένδρα
  - Κατακερματισμός, Γράφοι

# Modules, Makefiles, Editors, Git

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Creating large programs

- A large program might contain hundreds of thousands lines of code
- Having such a program in a single `.c` file is not practical
  - Hard to write
  - Hard to read and understand
  - Hard to maintain
  - Slow to compile
- We need to split it in semantically related units

## Modules

- A **module** (ενότητα) is a collection of related data and operations
- They allow achieving **abstraction** (αφαίρεση), a notion of fundamental importance in programming
- The **user** of the module only needs to know **what** the module does
- Only the **author** of the module needs to know **how** it is implemented
  - This is useful even when the author and the user are the same person
- They will be used to implement **Abstract Data Types** later in this course

## Information Hiding

- A notion closely related to abstraction
- Since the user does not need to know **how** the module is implemented, anything not necessary for using the module should be **hidden**
  - internal data, auxiliary functions, data types, etc
- This allows **modifying** parts of the program **independently**
  - a function visible only within the module cannot affect other parts of the program
  - think of changing a car’s tires, it should not affect its engine!

## Modules in C

- A module in C is represented by a **header** file `module.h`
  - we already know several modules: `stdio.h`, `string.h`, …
- It simply **declares** a list of functions
  - also **constants** and **typedefs**
- Describes **what** the module does
  - often with documentation for these functions

## Modules in C

- E.g. A `stats.h` module with two functions

```c
// stats.h - Απλά στατιστικά στοιχεία για πίνακες

#include <limits.h>     // INT_MIN, INT_MAX

// Επιστρέφει το μικρότερο στοιχείο του array (INT_MAX αν size == 0)

int stats_find_min(int array[], int size);

// Επιστρέφει το μεγαλύτερο στοιχείο του array (INT_MIN αν size == 0)

int stats_find_max(int array[], int size);
```

- Prefixing all functions with `stats_` is a good practice (why?)

## Using a C module

- `#include "module.h"`
- Use the provided functions
- As **users**, we don't need to know how the module is implemented!

```c
// minmax.c - Το βασικό αρχείο του προγράμματος

#include <stdio.h>
#include "stats.h"

int main() {
    int array[] = { 4, 35, -2, 1 };

    printf("min: %d\n", stats_find_min(array, 4));
    printf("max: %d\n", stats_find_max(array, 4));
}
```

## Implementing a C module

- The module's **implementation** is provided in a file `module.c`
- `module.c` contains the definitions of all functions declared in `module.h`

```c
// stats.c - Υλοποίηση του stats module

#include "stats.h"

int stats_find_min(int array[], int size) {
    int min = INT_MAX;              // "default" τιμή, μεγαλύτερη από όλε

    for(int i = 0; i < size; i++)
        if(array[i] < min)
            min = array[i];     // βρέθηκε νέο ελάχιστο

    return min;
}
```

## Compiling a program with modules

- Simply compiling `minmax.c` together with `module.c` works

```
gcc minmax.c stats.c -o minmax
```

- But this compiles both files every time
- What if we change a single file in a program with 1000 `.c` files?

## Separate compilation

- We can compile each `.c` file **separately** to create an `.o` file
- Then **link** all `.o` files together to create the executable

```
gcc -c minmax.c -o minmax.o
gcc -c stats.c -o stats.o

gcc minmax.o stats.o -o minmax
```

- If we change `minmax.c`, we only need to recompile **that** file and relink
  - `Makefile`s make this very easy

## Multiple implementations of a module

- The same `module.h` can be implemented in **different** ways

```c
// stats_alt.c - Εναλλακτική υλοποίηση του stats module

#include "stats.h"

// Επιστρέφει 1 αν value <= array[i] για κάθε i
int smaller_than_all(int value, int array[], int size) {
    for(int i = 0; i < size; i++)
        if(value > array[i])
            return 0;
    return 1;
}

int stats_find_min(int array[], int size) {
    for(int i = 0; i < size; i++)
        if(smaller_than_all(array[i], array, size))
            return array[i];

    return INT_MAX;     // εδώ φτάνουμε μόνο σε περίπτωση κενού array
}
```

## Compiling with multiple implementations

- `minmax.c` is compiled **without knowing** how `stats.h` is implemented
  - this is **abstraction**!
- We can then link with **any** implementation we want

```
gcc -c minmax.c -o minmax.o

# use the first implementation
gcc -c stats.c -o stats.o
gcc minmax.o stats.o -o minmax

# OR the second
gcc -c stats_alt.c -o stats_alt.o
gcc minmax.o stats_alt.o -o minmax
```

## Multiple implementations of a module

- All implementations should provide the same high-level behavior
    - So the program will work with any of them
- But one implementation might be **more efficient** than some other
    - This often depends on the specific application
- Which implementation of `stats.h` would you choose?

## Makefiles

- Good programmers are **lazy**
  - they want to spend their time programming, not compiling
- Nobody likes typing the same `gcc ...` commands 100 times
- We can **automate** compilation with a `Makefile`

## A simple Makefile

```
# Ένα απλό Makefile (με αρκετά προβλήματα)
# Προσοχή στα tabs!
minmax:
	gcc -c minmax.c -o minmax.o
	gcc -c stats.c -o stats.o
	gcc minmax.o stats.o -o minmax
```

- This means: to create the file `minmax` run these commands
- To compile we run `make minmax`
  - or simply `make` to compile the **first** target in the `Makefile`

## A simple Makefile - first problem

- We modify `minmax.c`, but make refuses to rebuild `minmax`

```
$ make minmax
make: 'minmax' is up to date.
```

- solution: dependencies

```
minmax: minmax.c stats.c
    gcc -c minmax.c -o minmax.o
    gcc -c stats.c -o stats.o
    gcc minmax.o stats.o -o minmax
```

- this means: `minmax` **depends** on `minmax.c`, `stats.c`
  - if any of these files is **newer** (last modification time) than `minmax` itself, the commands are run again!

## A simple Makefile - second problem

- We modify `minmax.c`, but `make` recompiles **everything**
- Solution: **separate rules** for each file we create

```
minmax.o: minmax.c
    gcc -c minmax.c -o minmax.o

stats.o: stats.c
    gcc -c stats.c -o stats.o

minmax: minmax.o stats.o
    gcc minmax.o stats.o -o minmax
```

- To build `minmax` we need to build `minmax.o`, `stats.o`
  - `minmax.o` depends on `minmax.c` which is newer, so `make` recompiles
  - `stats.o` depends on `stats.c` which is older, so no need to recompile

## Implicit rules

- `make` knows how to make `foo.o` if a file `foo.c` exists, by running

  ```
  gcc -c foo.c -o foo.o
  ```

- This is called an **implicit rule**
- So we don’t need rules for `.o` files!

```
minmax: minmax.o stats.o
	gcc minmax.o stats.o -o minmax
```

## Variables

- We can use **variables** to further simplify the `Makefile`
  - To create a variable: `VAR = ...`
  - To use a variable we write `$(VAR)` anywhere in the `Makefile`
- This allows to easily reuse the `Makefile`

```
# Αρχεία .o (αλλάζουμε απλά σε stats_alt.o για τη δεύτερη υλοποίηση!)
OBJS = minmax.o stats.o

# Το εκτελέσιμο πρόγραμμα
EXEC = minmax

$(EXEC): $(OBJS)
	gcc $(OBJS) -o $(EXEC)
```

## CFLAGS variable

- A **special variable**
- Passed as arguments to the compiler when compiling a `.o` file using an implicit rule
- E.g. enable all warnings, treat them as errors, and allow debugging

```
CFLAGS = -Wall -Werror -g
```

## Auxiliary rules

- Then don't really create files but run useful commands
- E.g. we can use `make clean` to delete all files the compiler built

```
clean:
    rm -f $(OBJS) $(EXEC)
```

- And `make run` to compile and execute the program with predefined arguments

```
ARGS = arg1 arg2 arg3

run: $(EXEC)
    ./$(EXEC) $(ARGS)
```

## Structuring a large project

- As projects grow, having all files in a single directory is not practical
- E.g. we want the same module to be used by many programs
- A simple structure:

| **Directory** | **Content** |
|---|---|
| `include` | shared modules, used by multiple programs |
| `modules` | module implementations |
| `programs` | executable programs |
| `tests` | unit tests (we'll talk about these later) |
| `lib` | libraries (we'll talk about these later) |

## Putting the pieces together

```
# paths
MODULES = ../../modules
INCLUDE = ../../include

# Compile options. To -I<dir> χρειάζεται για να βρει ο gcc τα αρχεία
CFLAGS = -Wall -Werror -g -I$(INCLUDE)

# Αρχεία .o, εκτελέσιμο πρόγραμμα και παράμετροι
OBJS = minmax.o $(MODULES)/stats.o
EXEC = minmax
ARGS =

$(EXEC): $(OBJS)
	gcc $(OBJS) -o $(EXEC)

clean:
	rm -f $(OBJS) $(EXEC)

run: $(EXEC)
	./$(EXEC) $(ARGS)
```

## Editor use in programming

- Programs are plain text files
- Any editor can be used
- But using an editor **efficiently** is important
- It can make the difference between boring and creative programming

## Editor types

- Old-school editors: **vim, emacs, …**
  - Fast, reliable, very configurable, available everywhere
  - Compiling/debugging is hard, needs tweaking
- IDEs: **Visual Studio**, **Eclipse**, **NetBeans**, **CLion**, …
  - Integrated compiler, debugger and many other tools
  - Too much “magic”, not ideal for learning
- Modern code-editors: **VS Code**, **Sublime Text**, **Atom**, …
  - Good balance between the two
  - Many options, a bit of tweaking is needed

## VS Code

- Modern, open-source code editor, available for all major systems
- Made by Microsoft, but it’s completely different than Visual Studio (an IDE)
- Will be used in lectures
  - lecture code is configured for use in VS Code
  - but you are free to use any other editor you want
- Installation instructions for all tools used in the class

## Configuring VS Code

- `.vscode` dir provided in the lecture code
    - you can copy this directory in any of your projects
- You only need to modify `.vscode/settings.json`

```
{
    "c_project": {
        // Directory στο οποίο βρίσκεται το πρόγραμμα
        "dir": "programs/minmax",

        // Όνομα του εκτελέσιμου προγράμματος
        "program": "minmax",

        // Ορίσματα του προγράμματος.
        "args": ""1 -2 3 52",
    },
}
```

## Compiling/Executing in VS Code

- Menu `Terminal / Run Task`
- `Make: compile` executes

  ```
  make <program>
  ```

  Errors are nicely displayed
- `Make: compile and run` executes

  ```
  make <program>
  ./<program> <arg1> <arg2> ...
  ```
- `Ctrl-Shift-B` executes the default task

## Debugging in VS Code

- Set breakpoints (`F9`)
- `F5` to start debugging
- We can examine/modify variables while execution is paused
- We can execute code step by step
- We can see where **segmentation faults** happen

## A few useful VS Code features

- `Ctrl-P`: quickly open file
- `Ctrl-Shift-O`: find function
- `Ctrl-/`: toggle comment
- `Ctrl-Shift-F`: search/replace in all files
- ``Ctrl-` ``: move between code and terminal
- `F8`: go to next compilation error
- `Alt-up`, `Alt-down`: move line(s)

## Git

- A system for tracking changes in source code
  - used by most major projects today
- Very useful when multiple developers collaborate in the same code
  - but also for single-developer projects
- We will use it for
  - lecture code
  - labs
  - projects
- We will store repositories in `github.com`, a popular Git hosting site

## Git, main workflow

1. `clone` a repository, creating a local copy
2. Modify some files
3. `commit` changes to the local repository
4. `push` the changes to the remote repository

For multiple developers/machines:

5. `pull` changes from a different local repository copy

## Git, getting started

- Install Git following the instructions
- Configure Git

```
git config --global user.email "you@example.com"
git config --global user.name "Your Name"
```

- Create an account on `github.com`
- Create an empty (public or private) repository `test-repo` on `github.com`
  - Check *"Initialize this repository with a README"*
  - Its URL will be `https://github.com/<username>/test-repo`

## Git, cloning a repository

```
git clone https://github.com/<username>/test-repo
```

- This will create a directory `test-repo` containing a local repository copy
- Check that `README.md` is present
- Try running `git status` inside `test-repo`

## Git, committing changes

- Modify `README.md`
- Run `git status`
  - `README.md` appears as **modified**
- To commit the changes:

```
git commit -a -m "Change README"
```

`-a` : commit **all** modified files

`-m "..."` : assign a message to the commit

## Git, adding files

- Create a new file `foo.c`
- Run `git status`
  - `foo.c` appears as **untracked**
- To add it

```
git add foo.c
git commit -m "Add foo.c"
```

- Run `git status` again

```
Your branch is ahead of 'origin/master' by 2 commits.
```

## Git, pushing commits

- Visit (or clone) `https://github.com/<username>/test-repo`
  - the local changes do not appear
- To push your local commits to the remote repository

```
git push
```

## Git, pulling commits

- From a different local repository copy (e.g. a different machine)

```
git pull
```

- The remote changes are copied to the local repository
- Local changes should be committed before running this
  - They will be **merged** with the remote ones

## .gitignore

- Files listed in the `.gitignore` special file are ignored by Git (blacklist)
- The inverse is often useful
  - save nothing except files in `.gitignore` (whitelist)

```
# Αγνοούμε όλα τα αρχεία (όχι τα directories)
*
!*/

# Εκτός από τα παρακάτω
!*.c
!*.h
!*.mk
!Makefile
!.gitignore
!README.md
!.vscode/*.json
```

## Readings

- T. A. Standish. Data Structures, Algorithms and Software Principles in C, Chapter 4
- Robert Sedgewick. Αλγόριθμοι σε C, Κεφ. 4
- `make` manual, Chapter 2
- A beginner’s guide to Git
- VS Code introductory videos

# Recap : Memory Allocation, Pointers, Structs, Typedefs

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Computer memory

- Computers have memory, a device that allows storing and retrieving data
- Each **byte** of memory has an **address**
  - unique number associated with this byte
- Programs need to **allocate**, **deallocate**, **read** and **write** on memory
- In C the programmer has **direct access** to the memory
  - the only complicated part, in an otherwise very simplistic language

## Allocating memory

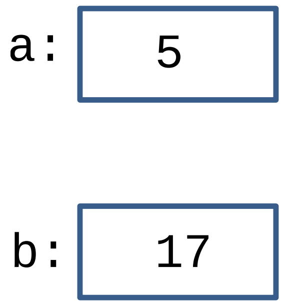
C programs allocate memory in 2 ways:

- **Automatic**
  - by declaring variables
- **Manual**
  - by calling `malloc`

# Allocating memory via variables

- Space for two `int`s is allocated the moment `foo` is **called**
- The values `5`, `17` are copied in the allocated memory

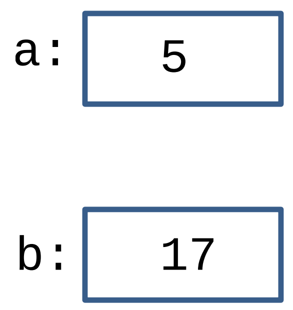
```
void foo() {
    int a = 5;
    int b = 17;
}

int main() {
    foo();
}
```



# Parameters

- Parameters are essentially just local variables
- Only difference: the **argument** provided by the caller is copied

```
void foo(int a) {
    int b = 17;
}

int main() {
    foo(5);
}
```



## Address-of operator `&`

- To see where a variable is stored, use the **address-of** operator `&`
- We can print it in hex

```
printf("Memory address of a in hex: %p \n", &a);
```

```
void foo(int a) {
    int b = 17;

    printf("foo &a: %p\n", &a);
    printf("foo &b: %p\n", &b);
}
```

## Deallocating a variable’s memory

- A variable’s memory is **deallocated** when the function call **returns**
- Deallocation simply means that such memory can be given to some **other variable**

```
void foo() {
    int a = 5;
    printf("foo &a: %p\n", &a);
}

void bar() {
    int a = 17;
    printf("bar &a: %p\n", &a);
}

int main() {
    foo();
    bar();
}
```

## Deallocating a variable’s memory

- Here, `foo` has **not returned** yet when `bar` is called
- Will we get the same result?

```
void bar() {
    int a = 17;
    printf("bar &a: %p\n", &a);
}

void foo() {
    int a = 5;
    printf("foo &a: %p\n", &a);
    bar();
}

int main() {
    foo();
}
```

## Global variables

They remain allocated until the program finishes

```c
int global = 5;

void foo() {
    printf("foo  &global: %p\n", &global);
}

int main() {
    printf("main &global: %p\n", &global);
    foo();
    printf("main &global: %p\n", &global);
}
```

## Pointers

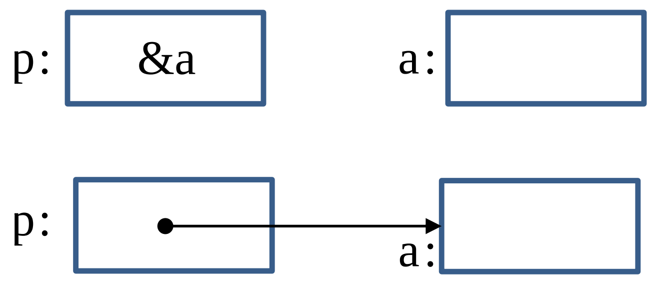
- Pointers are just variables, nothing special
- They are allocated/deallocated the same way as all variables are
  - Their **content** has **nothing** to do with allocation/deallocation

```
void foo() {
    int* p;
}
```

## Pointer content

- In a pointer we store **memory addresses**, e.g. the address of a variable
- Nothing special happens; we just store a **number** in a variable
  - we just think of `p` as “pointing to” `a`

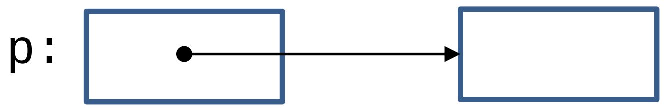
```
void foo() {
    int a;
    int* p = &a;

    printf("&a: %p\n", &a);
    printf("&p: %p\n", &p);
    printf(" p: %p\n",  p);
}
```



# Manual allocation

- Done by calling `malloc(size)` (actually easier to understand)
- Returns the **address** of the allocated memory
  - we need to **store** such address (in a pointer)

```c
int* p = malloc(sizeof(int));
```

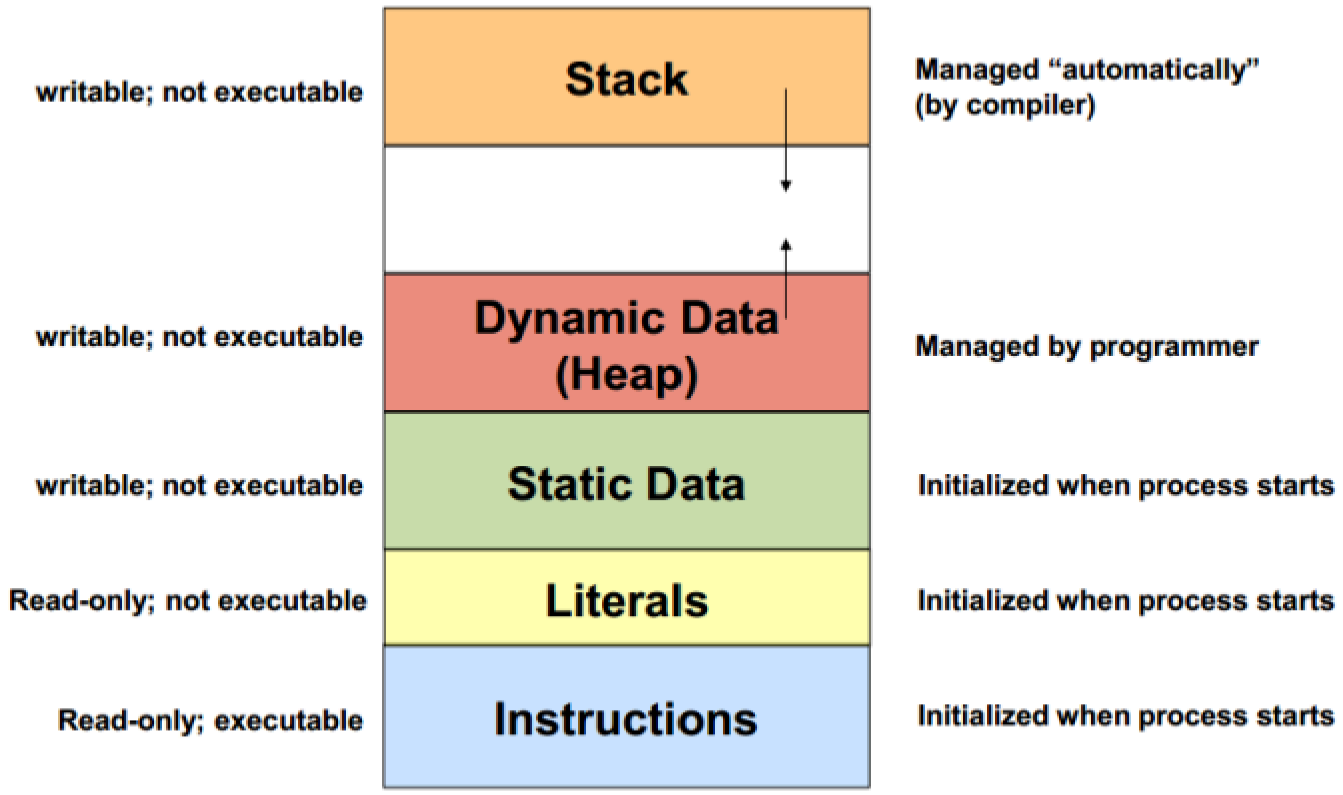


## Manual allocation

- The allocated memory is **not** the address of **any** variable
- In fact, the allocated memory is “far” from all variables
  - variables are allocated in the **stack**
  - `malloc` allocates memory in the **heap**
  - just fancy names for two different areas of memory

```
int* a = malloc(sizeof(int));

printf("&a: %p\n", &a);
printf(" a: %p\n",  a);
```

## Program Memory



## Manual deallocation

- Call `free(address)`, for some `address` previously returned by `malloc`
  - typically stored in a pointer
- `free`d memory can be reused (this is what “deallocated” means)

```
int* p1 = malloc(sizeof(int));
int* p2 = malloc(sizeof(int));

free(p2);
int* p3 = malloc(sizeof(int));

printf("p1: %p\n", p1);
printf("p2: %p\n", p2);
printf("p3: %p\n", p3);
```

## Remember

1. C never looks at the **content** of a variable when deallocating its memory

```c
void foo() {
    int* p = malloc(sizeof(int));
}
```

2. `free` does not modify the content of any variable

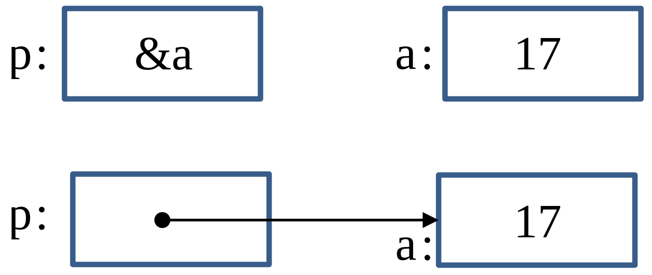
```c
int* p = malloc(sizeof(int));

printf("p: %p\n", p);
free(p);
printf("p: %p\n", p);

p = NULL;        // καλή πρακτική μετά το free
```

## Accessing memory via pointers, operator *

When reading or writing to a variable:

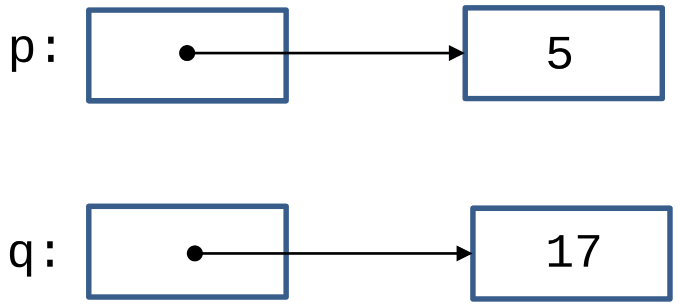
- a, p, read/write to the memory **allocated** for a, p
- *p reads/writes to the memory **stored** in p

```
int a;
int* p;

 p = &a;     // στη μνήμη που δεσμεύτηκε για το p, γράψε τον αριθμό &a
*p = 16;     // στη μνήμη που περιέχει το p (δηλαδή στην &a), γράψε το

 a = *p + 1;
```

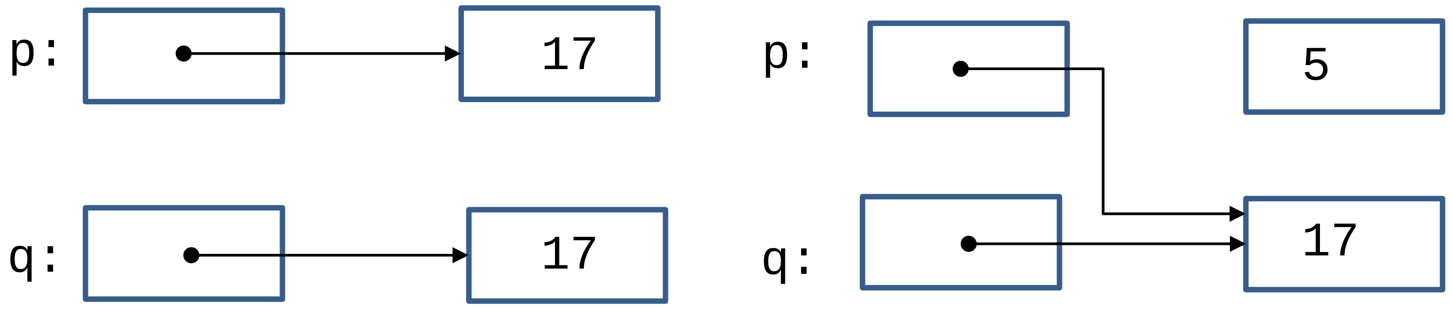


# Pointer quiz

- We have this situation:

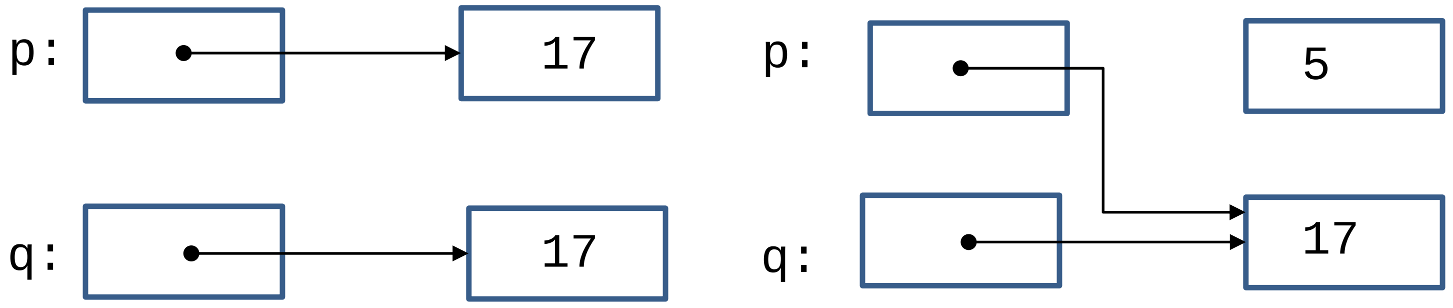


- Which commands produce each of the following?



## Pointer quiz

- Which commands produce each of the following?



```
*p = *q;    // αριστερό

 p =  q;    // δεξί
```

## Pointers as function arguments

- Nothing special happens at all
  - We just receive a **number** as an argument
- This is very useful for accessing memory outside the function

```c
void foo(int a, int* p) {
    *p = a;
}

int main() {
    int a = 1;
    foo(52, &a);
}
```

## Swap

Will this work?

```
void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}

int main() {
    int a = 1;
    int b = 5;
    swap(a, b);
}
```

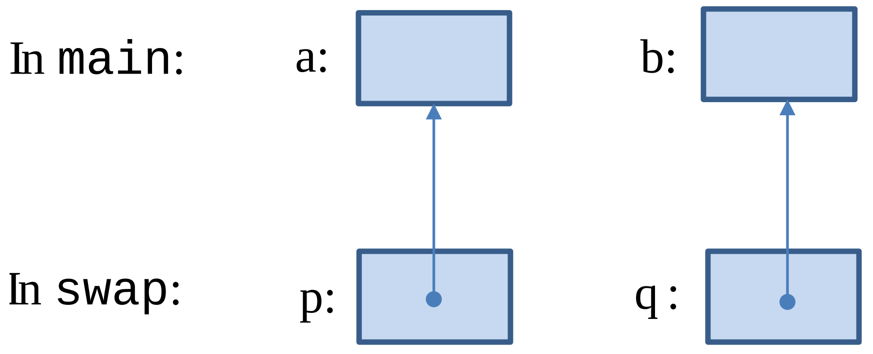
## Swap

Will this work?

```
void swap(int* p, int* q) {
    int* temp = p;
    p = q;
    q = temp;
}

int main() {
    int a = 1;
    int b = 5;
    swap(&a, &b);
}
```

## Swap, correct

```
void swap(int* p, int* q) {
    int temp = *p;
    *p = *q;
    *q = temp;
}

int main() {
    int a = 1;
    int b = 5;
    swap(&a, &b);
}
```



## Returning pointers

- Again, nothing special happens, we just **return a number**

```
int* foo() {
    int* p = malloc(sizeof(int));
    *p = 42;
    return p;
}

int main() {
    int* p = foo();
    printf("content of p: %d\n", *p);
    free(p);
}
```

## Dangling Pointers

- A pointer `p` containing **deallocated** memory is dangerous!
    - it’s not our memory anymore
    - using `*p` has **undefined** behaviour (typically it makes your PC explode)
- Think about deallocation rules **before returning a pointer**

```c
int* foo() {
    int a = 63;
    int* p = &a;
    return p;       // πού δείχνει ο p;
}

int* foo() {
    int* p = malloc(sizeof(int));
    *p = 42;
    free(p);
    return p;       // πού δείχνει ο p;
}
```

## Structs

- A simple way of storing several pieces of data together
- Useful for creating custom **types**
- A struct has **members**, each member has a **name**

```
struct point_2d {                   // ένα σημείο στον δισδιάστατο χώρο
    float x;
    float y;
};

int main() {
    struct point_2d point;          // μία μεταβλητή!
    point.x = 1.2;                  // έχει αρκετό χώρο
    point.y = 0.4;                  // για 2 floats
}
```

## Structs, allocation

- Nothing special, just like any other type

```c
void foo() {
    // θα αποδεσμευθεί στο τέλος της κλήσης της foo
    struct point_2d point;

    // θα αποδεσμευθεί όταν κάνουμε free
    struct point_2d* p = malloc(sizeof(struct point_2d));
}
```

## Structs, pointers

- When `p` is a **pointer to a struct**:
  - `p->member` is just a **synonym** for `(*p).member`

```c
void foo(struct point_2d* p) {
   (*p).x = -1.2;
   p->y = 0.4;

    // Μπορούμε να αντιγράφουμε και ολόκληρο το struct!
    struct point_2d point = *p;
    point.x = point.y * 2;
    *p = point;
}
```

## typedef

- Simply gives a **new name** to an existing type

```c
typedef int Intetzer;        // English style
typedef int Integker;        // Γκρικ στάιλ

int main() {
    Intetzer a = 1;
    Integker b = 2;
    a = b;                  // και τα δύο είναι απλά ints
}
```

## typedef, common uses

Simplify structs

```
struct point_2d {
    float x;
    float y;
};
typedef struct point_2d Point2d;

int main() {
    Point2d point;  // δε χρειάζεται το "struct point_2d"
}
```

Even simpler:

```
typedef struct {
    float x;
    float y;
} Point2d;
```

## typedef, common uses

“Hide” pointers

```c
// list.h
struct list {
    ...
};
typedef struct list* List;

List list_create();
void list_destroy(List list);
```

```c
// main.c
#include "list.h"

int main() {
    List list = list_create();        // ποιος "pointer";
    list_destroy(list);
}
```

## Function pointers

- Receive a **function** as argument
- A `typedef` is **highly** recommended

```c
// Για μια συνάρτηση σαν αυτή
int foo(int a) {
    ...
}

// Δηλώνουμε τον τύπο ως εξής (το foo αλλάζει σε (*TypeName))
typedef int (*MyFunc)(int a);

int main() {
    // Και μετά μπορούμε να αποθηκεύουμε το "foo" σε μια μεταβλητή f
    MyFunc f = foo;
    f(40);      // το ίδιο με foo(40)
}
```

## Function pointers

```c
typedef int (*MyFunc)(int a);

int foo1(int a) {
    return a + 1;
}
int foo2(int a) {
    return 2*a;
}

int bar(MyFunc f) {
    printf("f(0) = %d\n", f(0));
}

int main() {
    bar(foo1);
    bar(foo2);
}
```

## Void pointers

- All pointers are just numbers!
- A variable with type `void*` can store **any pointer**

```
int* int_p;
float* float_p;
Point2d* point_p;
MyFunc func_p;

void* p;

p = int_p;
p = float_p;
p = point_p;
p = func_p;

int_p = p;
float_p = p;
point_p = p;
func_p = p;
```

## Generic functions

- `void*` allows to define operations on data of **any type**

```c
// Ανταλλάσσει τα περιεχόμενα των p,q, μεγέθους size το καθένα
void swap(void* p, void* q, int size) {
    void* temp = malloc(size);   // allocate size bytes
    memcpy(temp, p, size);       // αντιγραφή size bytes από το p στο
    memcpy(p, q, size);
    memcpy(q, temp, size);
    free(temp);
}

int main() {
    int a = 1;
    int b = 5;
    swap(&a, &b, sizeof(int));

    float c = 4.3;
    float d = 1.2;
    swap(&c, &d, sizeof(float));
}
```

## Generic functions

Combine with **function pointers** for full power!

```
typedef void* Pointer;                      // απλούστερο

// Δείκτης σε συνάρτηση που συγκρίνει 2 στοιχεία a και b και επιστρέφ
// < 0  αν a <  b
//   0  αν a == b
// > 0  αν a >  b

typedef int (*CompareFunc)(Pointer a, Pointer b);

Pointer max(Pointer a, Pointer b, CompareFunc comp) {
    if(comp(a, b) > 0)
        return a;
    else
        return b;
}
```

## Generic functions

```c
#include <string.h>

int compare_ints(Pointer a, Pointer b) {
    int* ia = a;
    int* ib = b;
    return *ia - *ib;
}

int compare_strings(Pointer a, Pointer b) {
    return strcmp(a, b);
}

int main() {
    int a1 = 1;
    int a2 = 5;
    int* max_a = max(&a1, &a2, compare_ints);

    char* s1 = "zzz";
    char* s2 = "aaa";
    char* max_s = max(s1, s2, compare_strings);

    printf("max of a1,a2: %d\n", *max_a);
    printf("max of s1,s2: %s\n",  max_s);
}
```

## Readings

- Π. Σταματόπουλος, *Σημειώσεις Εισαγωγής στον Προγραμματισμό*.

# Code style, Tests, Debugging

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Code style

- In most programming languages **whitespace is ignored**
  - Leaves many options for styling
- The exact style is not important, no need to be “dogmatic” about it
- But it is very important to be **consistent**
  - Good style makes the code **readable**

## Naming conventions

- Similarly, how we **name** things is important
  - variables
  - modules
  - functions
  - types
  - etc.
- Consistent naming greatly improves code quality

## In this class

- The following slides present some style & naming choices
- The code used in the lectures follow this style
- You are **not** required to follow it
- But you **are** required to consistently follow a specific style

## Comments

```
// C++ style
// στα Ελληνικά

void foo() {
    int a = 1;      // μικρά comments στην ίδια γραμμή
}
```

- Makes it easy to toggle comments (`Ctrl-/` in VS Code)
- Don't over-use comments
  - they should not explain **what** the code does
  - but **how/why**
- **Don't** leave old garbage code in comments (Git keeps the history!)

## Brackets

```
// Στην ίδια γραμμή με την εντολή που τα ανοίγει

void foo() {
    for (unsigned int i = 0; i < ...; i++) {
        ...
    }

    // Για μικρές εντολές το παρακάτω είναι ok (χωρίς κατάχρηση)
    if (condition)
        do_something();
}
```

## Indentation

- One **tab** for each level
  - allows each developer to configure the tab size differently
- Alternative option: **4 spaces**
  - appears the same in all editors
- **Don’t** mix the two

## Pointer types

```
// Το * κολλητά με τον τύπο (όχι με το όνομα)

int* foo(char* param) {
    int* pointer = &var;
    ...
}
```

Conceptually, `int*` is a type.

## Variable declarations

```
// Μία δήλωση ανά γραμμή, επαναλαμβάνουμε τον τύπο
// Επίσης, δηλώνουμε μεταβλητές στο σημείο και το scope που χρειάζοντ

void foo() {
    int var1 = 1;
    int var2 = 3;

    ...

    int var3 = 3;                   // δε χρειάζεται πιο πάνω

    if (condition) {
        int var4 = 4;               // var4 ορατό μόνο μέσα στο if
        ...
    }

    for (unsigned int i = 0; i < N; i++) {   // i ορατό μόνο μέσα στο
        int var5 = 5;
        ...
    }
}
```

## Names

- **Functions, variables, parameters**: `lowercase_with_underscores`
- **Types**: `CamelCase`
- **Constants**: `UPPERCASE`
- Choose **readable** names (not `a`, `b`, `c`, …)
- In **modules**: prefix with name of module (or abbreviation)
    - avoids conflicts

## Names

```c
// ADTList.h

// Οι σταθερές αυτές συμβολίζουν εικονικούς κόμβους στην αρχή/τέλος.
#define LIST_BOF (ListNode)0
#define LIST_EOF (ListNode)0

// Λίστες και κόμβοι αναπαριστώνται από τους τύπους List και ListNode
typedef struct list* List;
typedef struct list_node* ListNode;

// Δημιουργεί και επιστρέφει μια νέα λίστα.

List list_create(DestroyFunc destroy_value);

// Προσθέτει έναν νέο κόμβο __μετά__ τον node με περιεχόμενο value.

void list_insert_next(List list, ListNode node, Pointer value);

// Αφαιρεί τον __επόμενο__ κόμβο από τον node.

void list_remove_next(List list, ListNode node);
```

## How to test our code

- For simple code, we typically test it in `main`
  - often with input from the user
- This does not work for larger programs
  - Time consuming
  - Easy to miss edge cases
  - No automation
  - We tend to assume that fixes remain forever

## Unit Tests

- A **test** is a piece of code that tests some other code
  - e.g. tests a **module**
- It calls some functions of the module, then checks the result
- Each test should be **independent**
- It should test some basic functionality
  - especially edge cases

## Unit Tests

Advantages

- Re-run on every change
- Detect regressions
- Test different implementations of the same module
- Run in automated scripts (e.g. on `git push`)
- Write specifications even before writing the actual code
  - test-driven development

## A simple test for stats.h

```c
#include "acutest.h"            // Απλή βιβλιοθήκη για unit testing

#include "stats.h"

void test_find_min(void) {
    int array[] = { 3, 1, -1, 50 };

    TEST_ASSERT(stats_find_min(array, 4) == -1);
    TEST_ASSERT(stats_find_min(array, 3) == -1);
    TEST_ASSERT(stats_find_min(array, 2) == 1);
    TEST_ASSERT(stats_find_min(array, 1) == 3);
    TEST_ASSERT(stats_find_min(array, 0) == INT_MAX);
}
```

## A simple test for stats.h

```c
void test_find_max(void) {
    int array[] = { 3, 1, -1, 50 };

    TEST_ASSERT(stats_find_max(array, 4) == 50);
    TEST_ASSERT(stats_find_max(array, 3) == 3);
    TEST_ASSERT(stats_find_max(array, 2) == 3);
    TEST_ASSERT(stats_find_max(array, 1) == 3);
    TEST_ASSERT(stats_find_max(array, 0) == INT_MIN);
}

// Λίστα με όλα τα tests προς εκτέλεση
TEST_LIST = {
    { "find_min", test_find_min },
    { "find_max", test_find_max },
    { NULL, NULL } // τερματίζουμε τη λίστα με NULL
};
```

## Test coverage

- How to know if the tests cover all functionalities of the code?
- Simple solution: check **which lines** are executed
- `lcov`: a test coverage tool for C
- Try the following in `sample-project`

```
cd tests
make coverage
firefox coverage/index.html
```

## Valgrind

- Tool to check memory access
- Finds memory **leaks**
- Also detects access of **deallocated** memory
- Simple use:

```
valgrind ./program
```

## Debugging

- Fixing bugs is an art that needs **experience**
- Often more difficult than writing the code
- But following some **concrete steps** can help

## Debugging

**Step 1**: **reproduce** the problem

- In the **simplest** possible way
  - Simplest code, smallest input, number of steps, etc
- Ideally: with a simple **automated test**
- Sometimes this is the hardest part!
  - Keep in mind when reporting bugs to others

## Debugging

**Step 2**: **isolate** the bug

- Which parts of the code are affected by the bug?
- Use the **debugger** (sometimes)
  - Pause at **segmentation faults**
  - Set **breakpoints** and conditional breakpoints
- Add useful logs (`printf`) and `assert`s
- **Comment out** parts of the code to see if behaviour changes

## Debugging

**Step 3**: find the **root cause**

- Often the code that breaks is not the real root
- Follow the code flow **backwards**
  - Examine the debugger’s **call stack**
  - Add logs
- Compare the flow in buggy and non-buggy executions

## Debugging

**Step 4**: **understand** and **fix** the bug

- Don’t do random changes
- A fix that you don’t **understand** is usually not a correct fix!
- Add **documentation**
  - Usually code that fixes a tricky bug needs explanation

## Debugging

**Step 5**: **test** the fix

- Add **tests** if you don’t have them already
- Important so that the bug does not **reappear** in the future
- Ideally the new tests should **only pass when the fix is applied**

# Introduction to Abstract Data Types

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Abstract Data Type (ADT)

- A collection of **data** and **operations** that
  - have precisely described behaviour (we know **what** they do)
  - but no precise implementation (we don’t know **how** they do it)
- **ADTBookStore** (from the first lecture)
  - `insert(title)`
  - `remove(title)`
  - `find(title)`
- Do we know any such type?

# Native types

- How is `int` implemented?
- What does `int a = -2;` store in memory?
  - `10...10` (sign-magnitude)
  - `1111101` (1-complement)
  - `1111110` (2-complement)
  - bit order? (little vs big endian)
  - size? (16, 32, 64 bits)
  - at least $3 \cdot 2 \cdot 3 = 18$ possibilities! The choice dependes on the CPU.
- How is `a++` implemented?

## Native types

- Even simple native types and operations are in reality **abstract**
- We know **what** they do but not **how**
- `int a = 1` stores **some representation** of 1 in `a`
- `a++` stores the representation of $a + 1$ in `a`
  - where $a$ is the number represented in `a`
- `printf("%d", a)` prints the number $a$ represented in `a`

## Why?

1. We can write programs without **thinking** (or even knowing) about how these operations are implemented
   - use complicated algorithms easily
2. We can **change the implementation** of `int` (eg change the CPU) without changing the code
   - easy maintenance

It would be impossible to write complex programs without these features!

## Writing our own ADTs

- ADTFoo will be represented by the **module** `ADTFoo.h`
  - Declare a list of functions, constants, typedefs, etc
  - Describe **what** the module does, with documentation!
- To **use** ADTFoo
  - `#include "ADTFoo.h"`
  - Call its methods, eg `foo_create()`
  - Link with `foo.o` (or some library containing it)
- To **implement** ADTFoo
  - Create `foo.c`, implementing all functions
  - The implementation should match the advertised behaviour

## Containers

- The ADTs we learn in this class are **containers**
  - They allow to **insert** data (stored in the container)
  - Then **retrieve** it in different ways
  - And **remove** it
- Store values of **any type**: `void*`
- They have similar interfaces
  - Differ in the **way** data is inserted/removed/retrieved

## ADT Overview

| ADT | Description |
|---|---|
| **ADTVector** | An abstract growable “array” |
| **ADTList** | Insert at any position, no “random access” |
| **ADTQueue** | First-in, First-out |
| **ADTStack** | Last-in, First-out |
| **ADTPriorityQueue** | Fast-access of the maximum element |
| **ADTMap** | Associate key => value (array with any type of index) |
| **ADTSet** | Ordered collection of unique items |

## Naming

- We use **different** names for **ADTs** and **Data Structures**
    - eg. **ADTVector** implemented by a **Dynamic Array**
- Loosely following the naming of the C++ standard library
- Be careful: each ADT/DS is known under many different names
    - also: the same name is often used for ADTs and DSs
- Remember the substance, not just the names!

## A typical container ADTFoo

```c
// ADTFoo.h

// Ένα foo αναπαριστάται από τον τύπο Foo. Ο χρήστης δε χρειάζεται να
// γνωρίζει το περιεχόμενο του τύπου αυτού, απλά χρησιμοποιεί τις συν
// foo_* που δέχονται και επιστρέφουν Foo.

typedef struct foo* Foo;
```

- We use an **incomplete struct** to hide the implementation
- The user cannot create `struct foo` variables or access their content
- We can only store **pointers** to `struct foo` created by the module
  - called **handles**
  - using the `Foo` typedef we forget that they are pointers!
- And pass them to other methods

# A typical container ADTFoo

```c
// Δημιουργεί και επιστρέφει ένα νεό foo

Foo foo_create();

// Επιστρέφει τον αριθμό στοιχείων που περιέχει το foo

int foo_size(Foo foo);

// Προσθέτει την τιμή value στο foo

void foo_insert(Foo foo, Pointer value, ...);

// Αφαιρεί και επιστρέφει μια τιμή από το foo

Pointer foo_remove(Foo foo, ...);

// Βρίσκει και επιστρέφει ένα στοιχείο από το foo

Pointer foo_find(Foo foo, ...);

// Ελευθερώνει όλη τη μνήμη που δεσμεύει το foo

void foo_destroy(Foo foo);
```

## A typical use of ADTFoo

```c
#include "ADTFoo.h"

int main() {
    Foo foo = foo_create();

    // Προσθήκη στοιχείων στον ADT
    foo_insert(foo, int_pointer1);
    foo_insert(foo, int_pointer2);

    // Εύρεση στοιχείου
    int* value = foo_find(foo, ...);
    printf("found: %d", *value);

    // Αφαίρεση στοιχείου
    foo_remove(foo, ...);

    // Εκκαθάριση μνήμης
    foo_destroy(foo);
}
```

## Many containers allow iterating

Using the concept of **node**.

```
Foo foo = foo_create();
// ...insert...

// Διάσχιση όλων των στοιχείων (η σειρά εξαρτάται από τον ADT)

for(FooNode node = foo_first(foo);               // ξενικάμε από τον πρώτο
    node != FOO_EOF;                             // μέχρι να φτάσουμε στο
    node = foo_next(foo, node)) {                // μετάβαση στον επόμενο

    int* value = foo_node_value(foo, node); // η τιμή του συγκεκριμέν
    printf("value: %d\n", *value);
}
```

# Vectors, Lists, Stacks, Queues

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## ADTVector

- A Vector can be seen as an abstract **resizable** “array”
  - It is **not** an array (remember, it’s abstract)
  - but it behaves like one
- We can access existing elements based on their **position** (random access)
- We can insert/remove elements **at the end** of the vector (dynamic size)
- We can **search** for elements (but this is usually inefficient)
- We can **iterate** over elements (also possible using random access)

a.k.a. Dynamic/Growable/Resizable/Mutable Array, Array List, List, …

## Create, destroy

```c
// Ένα vector αναπαριστάται από τον τύπο Vector. Ο χρήστης δε χρειάζε
// γνωρίζει το περιεχόμενο του τύπου αυτού, απλά χρησιμοποιεί τις συν
// vector_<foo> που δέχονται και επιστρέφουν Vector.
//
// Ο τύπος Vector ορίζεται ως pointer στο "struct vector" του οποίου
// περιεχόμενο είναι άγνωστο (incomplete struct), και εξαρτάται από τ
// υλοποίηση του ADT Vector.

typedef struct vector* Vector;

// Δημιουργεί και επιστρέφει ένα νεό vector μεγέθους size, με στοιχεί
// αρχικοποιημένα σε NULL. Αν δεν υπάρχει διαθέσιμη μνήμη επιστρέφει
// VECTOR_FAIL.

Vector vector_create(int size, DestroyFunc destroy_value);

// Ελευθερώνει όλη τη μνήμη που δεσμεύει το vector vec.

void vector_destroy(Vector vec);
```

An initial size is given at creation (ignore `destroy_value` for now).

## Random access

```
// Επιστρέφει την τιμή στη θέση pos του vector vec (μη ορισμένο αποτέ
// pos < 0 ή pos >= size)

Pointer vector_get_at(Vector vec, int pos);

// Αλλάζει την τιμή στη θέση pos του Vector vec σε value. ΔΕΝ μεταβάλ
// μέγεθος του vector, αν pos >= size το αποτέλεσμα δεν είναι ορισμέν

void vector_set_at(Vector vec, int pos, Pointer value);
```

- Example `[4, 6, 2, 1]`
  - Get at `1`: `6`
  - Set `8` at `0`: `[8, 6, 2, 1]`

## Insert and delete at the end

```
// Προσθέτει την τιμή value στο _τέλος_ του vector vec. Το μέγεθος το
// μεγαλώνει κατά 1. Αν δεν υπάρχει διαθέσιμη μνήμη το vector παραμέν
// ήταν (αυτό μπορεί να ελεγχθεί με τη vector_size)

void vector_insert_last(Vector vec, Pointer value);

// Επιστρέφει  την τιμή της τελευταίας θέσης του vector.
// Το μέγεθος του vector μικραίνει κατά 1.

void vector_remove_last(Vector vec);
```

- The size of the vector is modified (in contrast to C arrays)!
- Example `[4, 6, 2, 1]`
  - Insert `3`: `[4, 6, 2, 1, 3]`
  - Remove: `[4, 6, 2, 1]`

## Search

```
// Βρίσκει και επιστρέφει το πρώτο στοιχείο στο vector που να είναι ί
// (με βάση τη συνάρτηση compare), ή NULL αν δεν βρεθεί κανένα στοιχε

Pointer vector_find(Vector vec, Pointer value, CompareFunc compare);
```

- Usually sequential search (remember, the implementation is not fixed!)
- Reduntant, could be implemented by iterating

## Iteration

```c
// Μέσω random access

int size = vector_size(vec);
for (int i = 0; i < size; i++) {
    int* value = vector_get_at(vec, i);
    printf("%d\n", *value);
}

// Μέσω κόμβων

for(VectorNode node = vector_first(vec);        // ξενικάμε από τον π
    node != VECTOR_EOF;                         // μέχρι να φτάσουμε
    node = vector_next(vec, node)) {            // μετάβαση στον επόμ

    int* value = vector_node_value(vec, node);  // η τιμή του συγκεκρ
    printf("value: %d\n", *value);
}
```

## Memory management

- The memory reserved for the **vector itself** is managed by the module
- We are responsible for the **contents** (`Pointer`s)
- Simple memory management:
  - `destroy_value` function to be called when a value is **removed**

```
Vector vec = vector_create(0, free);

vector_insert_last(vec, strdup("foo"));
vector_insert_last(vec, strdup("bar"));

vector_remove_last(vec);    // free bar

vector_destroy(vec);        // free foo (και destroy το ίδιο το vecto
```

## When to use Vectors

- General purpose containers
- When we need random access
- When we don’t need to insert at random positions
- When we don’t need efficient search

## ADTList

- We sacrifice random access for **insert/delete flexibility**
- Only **sequential** access
- We can insert and remove elements **anywhere**
- We can **search** for elements (but this is usually inefficient)
- We can **iterate** over elements in the order of insertion

a.k.a. Forward list (also, “List” sometimes means something else)

## Insert and delete anywhere

```
// Προσθέτει έναν νέο κόμβο __μετά__ τον node, ή στην αρχή αν node ==
// με περιεχόμενο value.

void list_insert_next(List list, ListNode node, Pointer value);

// Αφαιρεί τον __επόμενο__ κόμβο από τον node, ή τον πρώτο κόμβο αν n

void list_remove_next(List list, ListNode node);
```

- Positions represented by **nodes**
- Insert/remove happens **after** the given node
- Example `(4, 6, 2, 1)`
  - Insert `3` after `6`: `(4, 6, 3, 2, 1)`
  - Remove after `4`: `(4, 3, 2, 1)`

## Iteration

```
// Μόνο μέσω κόμβων

for(ListNode node = list_first(list);          // ξενικάμε από τον πρ
    node != LIST_EOF;                          // μέχρι να φτάσουμε σ
    node = list_next(list, node)) {            // μετάβαση στον επόμε

    int* value = list_node_value(list, node);  // η τιμή του συγκεκρι
    printf("value: %d\n", *value);
}
```

## Other functions

Same as for Vectors.

```
List list_create(DestroyFunc destroy_value);

// Επιστρέφει τον αριθμό στοιχείων που περιέχει η λίστα.

int list_size(List list);

// Επιστρέφει την πρώτη τιμή που είναι ισοδύναμη με value
// (με βάση τη συνάρτηση compare), ή NULL αν δεν υπάρχει

Pointer list_find(List list, Pointer value, CompareFunc compare);

// Ελευθερώνει όλη τη μνήμη που δεσμεύει η λίστα list.
// Οποιαδήποτε λειτουργία πάνω στη λίστα μετά το destroy είναι μη ορι

void list_destroy(List list);
```

## When to use Lists

- General purpose containers
- When sequential access is enough
- When we need to insert/delete at random positions
- When we don’t need efficient search

## Stacks

- Very limited functionality
  - but useful in practice
  - allows for efficient implementations
- Insert and delete at the **top**
  - Last-in, first-out (LIFO)
- Acceess only the **top** element
  - No random access
  - No iteration

## Examples of Stacks in Real Life

## LIFO access

```
// Επιστρέφει το στοιχείο στην κορυφή της στοίβας (μη ορισμένο αποτέλ
// στοίβα είναι κενή)

Pointer stack_top(Stack stack);

// Προσθέτει την τιμή value στην κορυφή της στοίβας stack.

void stack_insert_top(Stack stack, Pointer value);

// Αφαιρεί την τιμή στην κορυφή της στοίβας (μη ορισμένο
// αποτέλεσμα αν η στοίβα είναι κενή)

void stack_remove_top(Stack stack);
```

- Example `[4, 6, 2, 1]`
  - Insert `3`: `[4, 6, 2, 1, 3]`
  - Remove: `[4, 6, 2, 1]`
- Commonly called **push** and **pop**

## When to use Stacks

- When LIFO access is enough
- Many applications
  - Storing information of active function calls
  - Parsing algorithms
  - Expression evaluation algorithms
  - Backtracking algorithms
  - …

## Using a Stack to check for balanced parentheses

- Determine whether parentheses/brackets balance properly in algebraic expressions.

- Example:

$$\{a^2 - [(b + c)^2 - (d + e)^2] * [\sin(x - y)]\} - \cos(x + y)$$

- This expression contains parentheses, square brackets, and braces in balanced pairs according to the pattern

$$\{[()()][()]\}()$$

## The Algorithm

- Start with an **empty stack**
- Scan the algebraic expression from left to right
  - On `(, [, {` we **insert** it to the stack.
  - On `), ], }` we **remove** the top item and check that its type matches
- The expression is balanced **iff**
  - all pairs match, and
  - at the end the stack is **empty**

## Postfix Expressions

- Expressions are usually written in **infix** notation $L\ op\ R$
  - The operator appears **between** the operands
  - eg. $(a + b) * 2 - c$
  - Parentheses are used to denote the order
- **Postfix**: write the operator **after** the operands $L\ R\ op$
  - eg. $ab + 2 * c-$
  - Advantage: **no need for parentheses**!

## Examples

| Infix | Postfix |
|---|---|
| (a + b) | a b + |
| (x - y - z) | x y - z - |
| (x - y - z) / (u + v) | x y - z - u v + / |
| (a^2 + b^2) * (m - n) | a 2 ^ b 2 ^ + m n - * |

## Using a Stack to evaluate postfix expressions

- Scan from **left to right**
- When we find an **operand** $X$, **insert** it int the stack
- When we find an **operator** $op$
  - **remove** the top operand into a variable $R$ (right operand)
  - **remove** another topmost operand into a variable $L$ (left operand)
  - Perform the operation $L\ op\ R$
  - **Insert** the value back into the stack
- End of expression: its value is the (only) item remaining in the stack

## Translating Infix expressions to Postfix

- We can also use a Stack to translate **fully parenthesized** infix arithmetic expressions to postfix.
- **Algorithm** to convert $(L\ op\ R)$ to the postfix form $L\ R\ op$
  - ignore the left parenthesis
  - convert $L$ to postfix
  - save $op$ on the stack
  - convert $R$ to postfix
  - then, on $)$, pop the stack and output the $op$

## Example

- We want to translate the infix expression `((5*(9+8))+7)` into postfix.
- The result will be `5 9 8 + * 7 +`

| Input | Output | Stack |
|---|---|---|
| ( | | |
| ( | | |
| 5 | 5 | |
| * | | * |
| ( | | * |
| 9 | 9 | * |
| + | | * + |
| 8 | 8 | * + |
| ) | + | * |
| ) | * | |
| + | | + |
| 7 | 7 | + |
| ) | + | |

## Queues

- Very limited functionality (similarly to stacks)
  - but useful in practice
  - allows for efficient implementations
- Insert in the **back**, remove from the **front**
  - First-in, first-out (FIFO)
- Acceess only the **front** and **back** elements
  - No random access
  - No iteration

## FIFO access

```
// Επιστρέφει το στοιχείο στο μπροστινό μέρος της ουράς

Pointer queue_front(Queue queue);

// Επιστρέφει το στοιχείο στο πίσω μέρος της ουράς

Pointer queue_back(Queue queue);

// Προσθέτει την τιμή value στο πίσω μέρος της ουράς queue.

void queue_insert_back(Queue queue, Pointer value);

// Αφαιρεί την τιμή στο μπροστά μέρος της ουράς

void queue_remove_front(Queue queue);
```

- Example `[4, 6, 2, 1]`
  - Insert `3`: `[4, 6, 2, 1, 3]`
  - Remove: `[6, 2, 1, 3]`
- Commonly called **push**/**pop** (or enqueue/dequeue)

## When to use Queues

- When FIFO access is enough
- Many applications
    - Sheduling of processes
    - Access to resources (CPU, printers, etc)
    - Breadth First Search in trees and graphs
    - …

## Example, experimental simulation

- A new job arrives to the CPU each second with pb $p$
- Each job takes between 1 and 4 seconds to execute (random)
- What is the **average** waiting time?
- We can write a program the **simulates** the system using a queue
  - time represented by an integer `t`
  - at each second, insert a job randomly (using `rand`)
  - select `duration` also randomly
  - remove job after `duration` seconds, compute its waiting time

## Readings

- T. A. Standish. Data Structures, Algorithms and Software Principles in C. Chapter 7.
- R. Sedgewick. Αλγόριθμοι σε C., Κεφ. 4.

# Priority Queues, Maps, Sets

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## ADTs for efficient search

- So far we've seen ADTs in which elements have a **predefined order**
  - Depending only on the insertion order, not on the **values**
- And search is inefficient
- We will now discuss ADTs that take the **order** of elements into account
  - **Any** order can be used (many options, even for the same data)

## The compare function

- A `compare` function determines the order of elements
    - `compare(a, b) < 0` iff `a` is “smaller” than `b`
    - `compare(a, b) > 0` iff `a` is “greater” than `b`
    - `compare(a, b) == 0` iff `a` and `b` are **equivalent** (**not** equal!)
- Examples for integers:
    - `return *(int*)a - *(int*)b`
    - `return abs(*(int*)a) - abs(*(int*)b)`
    - `return a - b`
    - Which elements are equivalent?
- Examples for strings, structs, Vectors, etc?

## Priority Queue

- Similar to a Queue or Stack
  - No iteration, only a specific item can be accessed/removed
- **BUT**: the item that can be accessed/removed is the **maximum** one
  - with respect to a given order (`compare`)
- We can insert elements in any order
- When we remove, we always get the maximum element!

## Create, insert

```
// Δημιουργεί και επιστρέφει μια νέα ουρά προτεραιότητας, της οποίας
// Αν destroy_value != NULL, τότε καλείται destroy_value(value) κάθε
// Αν values != NULL, τότε η ουρά αρχικοποιείται με τα στοιχεία του v

PriorityQueue pqueue_create(CompareFunc compare, DestroyFunc destroy_

// Προσθέτει την τιμή value στην ουρά pqueue.

void pqueue_insert(PriorityQueue pqueue, Pointer value);
```

- The `compare` function must be given to `pqueue_create`
  - Cannot be modified.
- We can initialize the queue with a Vector of `values`
  - might be **faster** than insterting them one by one!

## Accessing the maximum element

```
// Επιστρέφει το μεγαλύτερο στοιχείο της ουράς

Pointer pqueue_max(PriorityQueue pqueue);

// Αφαιρεί την μεγαλύτερη τιμή της ουράς

void pqueue_remove_max(PriorityQueue pqueue);
```

- Guaranteed to work independently from the insertion order
- Efficient, even after removing elements!

## When to use Priority Queues

- When we need to quickly find the maximum value
  - on data that is updated
- Many applications
  - Shortest Path in a graph (Dijkstra)
  - Minimum Spanning Tree (Prim)
  - Search in Artificial Intelligence (A*)
  - Sorting elements (Heapsort)
  - Load balancing algorithms
  - Compression (Huffman codes)
  - …

## ADTMap

- Associates a `key` to a `value`
- Allows to search for `value` given `key` (or a key **equivalent** to it)
- Can be thought as an “array” with non-integer indexes!

```
// Αυτό είναι συνηθισμένο
names[19] = "Bob";
...
printf("The student with id 19 is %s\n", names[19]);

// Ωραία θα ήταν να μπορούσαμε να κάνουμε το αντίστροφο (αλλά δε γίνε
ids["Bob"] = 19;
...
printf("The id of Bob is %d\n", ids["Bob"]);
```

- We can use a Map `ids` that associates `"Bob"` to `19`

a.k.a. Symbol table, Dictionary, Associative array, Unordered map

## Create

```
// Δημιουργεί και επιστρέφει ένα map, στο οποίο τα στοιχεία συγκρίνον
// τη συνάρτηση compare.
// Αν destroy_key ή/και destroy_value != NULL, τότε καλείται destroy_
// ή/και destroy_value(value) κάθε φορά που αφαιρείται ένα στοιχείο.

Map map_create(CompareFunc compare, DestroyFunc destroy_key, DestroyF
```

- We need to pass a `compare` function
  - the Map needs to know which keys are equivalent to `"Bob"`
- Separate destroy functions for keys and values

## Insert, find

```
// Προσθέτει το κλειδί key με τιμή value. Αν υπάρχει κλειδί ισοδύναμο
// παλιά key & value αντικαθίσταται από τα νέα.

void map_insert(Map map, Pointer key, Pointer value);

// Επιστρέφει την τιμή που έχει αντιστοιχιστεί στο συγκεκριμένο key,
// το key δεν υπάρχει στο map.

Pointer map_find(Map map, Pointer key);
```

## Example

```c
// name => id
ADTMap ids = map_create(compare_strings, NULL, NULL);

// ids["Bob"] = 19
map_insert(ids, "Bob", create_int(19));

...

// Find ids["Bob"]
printf("The id of Bob is %d\n", *(int*)map_find(ids, "Bob"));
```

## Remove

```
// Αφαιρεί το κλειδί που είναι ισοδύναμο με key από το map, αν υπάρχε
// Επιστρέφει true αν βρέθηκε τέτοιο κλειδί, διαφορετικά false.

bool map_remove(Map map, Pointer key);
```

Example:

```
map_remove(ids, "Bob");

...

if(map_find(ids, "Bob") == NULL) {
    printf("No student named Bob exists");
}
```

# Iteration

```
// Μέσω κόμβων

for(MapNode node = map_first(ids);                // ξενικάμε από τον πρώτο
    node != MAP_EOF;                              // μέχρι να φτάσουμε στο
    node = map_next(ids, node)) {                 // μετάβαση στον επόμενο

    char* name = map_node_key(ids, node);         // το κλειδί του συγκεκρι
    int* id = map_node_value(ids, node);          // η τιμή του συγκεκριμέν
    printf("%s's id is %d\n", name, *id);
}
```

## When to use Map

- When we need equality search
  - One of the most commonly used operations
  - And the data is updated frequently
- Check if some value is already seen
  - eg. memoization
- Provided natively by many programming languages
  - but we need to understand how it works!

## ADTSet

- An **ordered** set of **unique** values
- Allows to search for a `value` efficiently
- Allows to iterate the set in the correct **order**

## Create

```
// Δημιουργεί και επιστρέφει ένα σύνολο, στο οποίο τα στοιχεία συγκρί
// βάση τη συνάρτηση compare.
// Αν destroy_value != NULL, τότε καλείται destroy_value(value) κάθε
// αφαιρείται ένα στοιχείο.

Set set_create(CompareFunc compare, DestroyFunc destroy_value);
```

We need to pass a `compare` function.

The Set maintains the **order** of the values.

## Insert, find

```
// Προσθέτει την τιμή value στο σύνολο, αντικαθιστώντας τυχόν προηγού
// ισοδύναμη της value.

void set_insert(Set set, Pointer value);

// Επιστρέφει την μοναδική τιμή του set που είναι ισοδύναμη με value,
// δεν υπάρχει

Pointer set_find(Set set, Pointer value);
```

## Example

```c
ADTSet names = set_create(compare_strings, NULL);

set_insert(names, "Alice");
set_insert(names, "Bob");

...

printf("Is Bob present? %d\n", set_find(names, "Bob") != NULL);
```

# Remove

```c
// Αφαιρεί τη μοναδική τιμή ισοδύναμη της value από το σύνολο, αν υπά
// Επιστρέφει true αν βρέθηκε η τιμή αυτή, false διαφορετικά.

bool set_remove(Set set, Pointer value);
```

Example:

```c
set_remove(names, "Bob");

...

if(set_find(students, "Bob") == NULL) {
    printf("No student named Bob exists");
}
```

## Iteration

```
// Μέσω κόμβων, στη σειρά διάταξης!

for(SetNode node = set_first(ids);          // ξενικάμε από τον πρώτο
    node != SET_EOF;                        // μέχρι να φτάσουμε στο
    node = set_next(ids, node)) {           // μετάβαση στον επόμενο

    char* name = set_node_value(ids, node); // Η τιμή του συγκεκριμέν
    printf("%s\n", name, *id);
}
```

## When to use Set

- When we need to access values in a certain **order**
  - And the data is updated frequently
- When we need **range** search
  - One of the most commonly used operations
- But also equality search

# Analysis of Algorithms, Complexity

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Outline

- How can we measure and compare algorithms meaningfully?
  - an algorithm will run at different speeds on different computers
- $O$ notation.
- Complexity types.
  - Worst-case vs average-case
  - Real-time vs amortized-time

## Selection sort algorithm

```
// Ταξινομεί τον πίνακα array μεγέθους size

void selection_sort(int array[], int size) {
  // Βρίσκουμε το μικρότερο στοιχείο του πίνακα, το τοποθετούμε στη θ
  // και συνεχίζουμε με τον ίδιο τρόπο στον υπόλοιπο πίνακα.

  for (int i = 0; i < size; i++) {
      // βρίσκουμε το μικρότερο στοιχείο από αυτά σε θέσεις >= i
      int min_position = i;
      for (int j = i; j < size; j++)
          if (array[j] < array[min_position])
              min_position = j;

      // swap των στοιχείων i και min_position
      int temp = array[i];
      array[i] = array[min_position];
      a[min_position] = temp;
  }
}
```

## Running Time

- Array of 2000 integers
- Computers A, B, …, E are progressively faster.
    - The algorithm runs faster on faster computers.

| Computer | Time (secs) |
|---|---:|
| Computer A | 51.915 |
| Computer B | 11.508 |
| Computer C | 2.382 |
| Computer D | 0.431 |
| Computer E | 0.087 |

## More Measurements

- What about **different programming languages**?
- Or **different compilers**?
- Can we say whether algorithm A is better than B?

## A more meaningful criterion

- Algorithms **consume resources**: e.g. time and space
- In some fashion that depends on the **size of the problem** solved
  - the bigger the size, the more resources an algorithm consumes
- We usually use $n$ to denote the size of the problem
  - the **length of a list** that is searched
  - the **number of items** in an array that is sorted
  - etc

## `selection_sort` running time

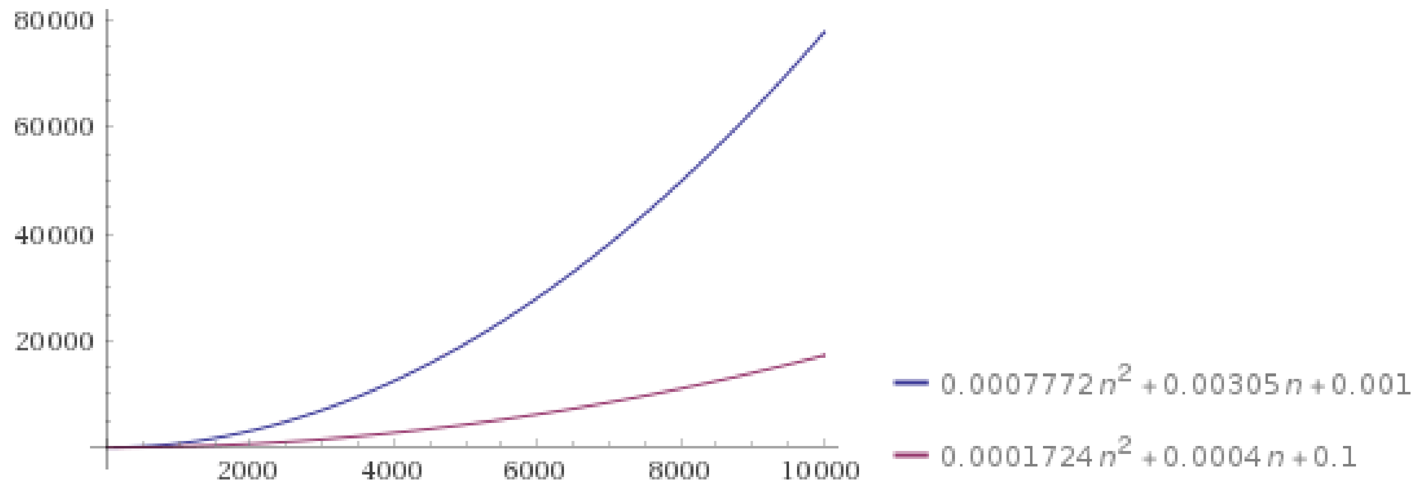
In msecs, on two types of computers

| Array Size | Home Computer | Desktop Computer |
|---|---|---|
| 125 | 12.5 | 2.8 |
| 250 | 49.3 | 11.0 |
| 500 | 195.8 | 43.4 |
| 1000 | 780.3 | 172.9 |
| 2000 | 3114.9 | 690.5 |

## Curves of the running times

If we plot these numbers, they lie on the following two curves:

- $f_1(n) = 0.0007772n^2 + 0.00305n + 0.001$
- $f_2(n) = 0.0001724n^2 + 0.00040n + 0.100$



## Discussion

- The curves have the **quadratic** form $f(n) = an^2 + bn + c$
  - difference: they have **different constants** $a, b, c$
- Different computer / programming language / compiler:
  - the curve that we get will be of the same form!
- The exact numbers change, but **the shape of the curve** stays the same.

## Complexity classes, $O$-notation

- We say that an algorithm belongs to a **complexity class**
- A class is denoted by $O(g(n))$
  - $g(n)$ gives the running time as a function of the size $n$
  - it describes the **shape** of the running time curve
- For `selection_sort` the time complexity is $O(n^2)$
  - take the **dominant term** of the expression $an^2 + bn + c$
  - throw away the constant coefficient $a$

## Why only the dominant term?

$$f(n) = an^2 + bn + c$$

with $a = 0.0001724, b = 0.0004$ and $c = 0.1$.

| $n$ | $f(n)$ | $an^2$ | $n^2$ **term as % of total** |
|---|---|---|---|
| 125 | 2.8 | 2.7 | 94.7 |
| 250 | 11.0 | 10.8 | 98.2 |
| 500 | 43.4 | 43.1 | 99.3 |
| 1000 | 172.9 | 172.4 | 99.7 |
| 2000 | 690.5 | 689.6 | 99.9 |

## Why only the dominant term?

- The lesser term $bn + c$ **contributes very little**
  - even though $b, c$ are much larger than $a$
  - Thus we can **ignore this lesser term**
- Also: we **ignore the constant** $a$ in $an^2$
  - It can be thought of as the “time of a single step”
  - It depends on the computer / compiler / etc
  - We are only interested in the shape of the curve

## Common complexity classes

| $O$-notation | Adjective Name |
|---|---|
| $O(1)$ | Constant |
| $O(\log n)$ | Logarithmic |
| $O(n)$ | Linear |
| $O(n \log n)$ | Quasi-linear |
| $O(n^2)$ | Quadratic |
| $O(n^3)$ | Cubic |
| $O(2^n)$ | Exponential |
| $O(10^n)$ | Exponential |
| $O(2^{2^n})$ | Doubly exponential |

## Sample running times for each class

Assume 1 step = 1 μsec.

| $g(n)$ | $n = 2$ | $n = 16$ | $n = 256$ | $n = 1024$ |
|---|---|---|---|---|
| 1 | 1 μsec | 1 μsec | 1 μsec | 1 μsec |
| $\log n$ | 1 μsec | 4 μsec | 8 μsec | 10 μsec |
| $n$ | 2 μsec | 16 μsec | 256 μsec | 1.02 ms |
| $n \log n$ | 2 μsec | 64 μsec | 2.05 ms | 10.2 ms |
| $n^2$ | 4 μsec | 25.6 μsec | 65.5 ms | 1.05 |
| $n^3$ | 8 μsec | 4.1 ms | 16.8 ms | 17.9 min |
| $2^n$ | 4 μsec | 65.5 ms | $10^{63}$ years | $10^{297}$ years |

## The largest problem we can solve in time T

Assume 1 step = 1 μsec.

| $g(n)$ | **T = 1 min** | **T = 1hr** |
|---|---|---|
| $n$ | $6 \times 10^7$ | $3.6 \times 10^9$ |
| $n \log n$ | $2.8 \times 10^6$ | $1.3 \times 10^8$ |
| $n^2$ | $7.75 \times 10^3$ | $6.0 \times 10^4$ |
| $n^3$ | $3.91 \times 10^2$ | $1.53 \times 10^3$ |
| $2^n$ | $25$ | $31$ |
| $10^n$ | $7$ | $9$ |

## Complexity of well-known algorithms

| | |
|---|---|
| Sequential searching of an array | $O(n)$ |
| Binary searching of a sorted array | $O(\log n)$ |
| Hashing (under certain conditions) | $O(1)$ |
| Searching using binary search trees | $O(\log n)$ |
| Selection sort, Insertion sort | $O(n^2)$ |
| Quick sort, Heap sort, Merge sort | $O(n \log n)$ |
| Multiplying two square x matrices | $O(n^3)$ |
| Traveling salesman, graph coloring | $O(2^n)$ |

## Formal definition of $O$-notation

$f(n)$ is the function giving the **actual time** of the algorithm.

We say that $f(n)$ is $O(g(n))$ iff

- there exist two positive constants $K$ and $n_0$
- such that $|f(n)| \leq K|g(n)| \quad \forall n \geq n_0$.

We will **not focus** on the formal definition in this course.

## Intuition

- An algorithm runs in time $O(g(n))$ iff it finishes in **at most** $g(n)$ **steps**.
- A “step” is anything that takes **constant time**
  - a basic operation, eg `a = b + 3`
  - a comparison, eg `if(a == 4)`
  - etc
- Typical way to compute this
  - find an expression $f(n)$ giving the exact number of steps (or an upper bound)
  - find $g(n)$ by removing the **lesser terms** and **coefficients** (justified by the formal definition)

## Example

- An algorithm takes $f(n)$ number of steps, where
  - $f(n) = 3 + 6 + 9 + \cdots + 3n$
- We will show that the algorithm runs in $O(n^2)$ steps.
- First find a closed form for $f(n)$:
  - $f(n) = 3(1 + 2 + \cdots + n) = 3\frac{n(n+1)}{2} = \frac{3}{2}n^2 + \frac{3}{2}n$
- Throw away
  - the lesser term $\frac{3}{2}n$
  - and the coefficient $\frac{3}{2}$
- We get $O(n^2)$

## Scale of strength for $O$-notation

To determine the dominant term and the lesser terms:

$$O(1) < O(\log n) < O(n) < O(n^2) < O(n^3) < O(2^n) < O(10^n)$$

Example:

- $O(6n^3 - 15n^2 + 3n \log n) = O(6n^3) = O(n^3)$

## Ignoring bases of logarithms

- When we use $O$-notation, we can **ignore the bases of logarithms**
  - assume that all logarithms are in base 2.
- Changing base involves multiplying by a **constant coefficient**
  - ignored by then $O$-notation
- For example, $\log_{10} n = \frac{\log n}{\log 10}$ . Notice now that $\frac{1}{\log 10}$ is a constant.

## $O(1)$

- It is easy to see why the $O(1)$ notation is the right one for constant time
- Constant time means that the algorithm finishes in $k$ steps
- $O(k)$ is the same as $O(1)$, constants are ignored

## Caveat 1

- $O$-complexity talks about the behaviour for **large values** of $n$
  - this is why we ignore lesser terms!
- For small sizes a “bad” algorithm might be faster than a “good” one
- We can test the algorithms **experimentally** to choose the best one

## Caveat 2

- $O(g(n))$ complexity is an **upper bound**
    - the algorithm finishes in **at most** $g(n)$ steps
- Comparing algorithms can be misleading!
    - item A costs **at most 10** euros
    - item B costs **at most 5000** euros
    - which one is cheaper?
- Programmers often say $O(g(n))$ but mean $\Theta(g(n))$
    - finishes in **"exactly"** $g(n)$ steps
    - we won't use $\Theta$ but keep this in mind

## Types of complexities

- Depending on the **data**
  - Worst-case vs Average-case
- Depending on the **number of executions**
  - Real-time vs amortized-time

## Worst-case vs Average-case

- Say we want to sort an array, **which values** are stored in the array?
- **Worst-case**: take the worst possible values
- **Average-case**: average wrt to all possible values
- Eg. quicksort
  - worst-case: $O(n^2)$ (when data are already sorted)
  - average-case: $O(n \log n)$

## Real-time vs amortized-time

- **How many times** do we run the algorithm?
- **Real-time**: just once
  - $n$ is the size of the problem
- **Armortized-time**: multiple times
  - take the average wrt all execution (**not** wrt the **values**!)
  - $n$ is the number of executions
- Example: Dynamic array! (we will see it soon)

## Some algorithms and their complexity

We will analyze the following algorithms

- Sequential search
- Selection sort
- Recursive selection sort

## Sequential search

```c
// Αναζητά τον ακέραιο target στον πίνακα target. Επιστρέφει τη θέση
// του στοιχείου αν βρεθεί, διαφορετικά -1

int sequential_search(int target, int array[], int size) {
    for (int i = 0; i < size; i++)
        if (array[i] == target)
            return i;

    return -1;
}
```

- The steps to locate `target` **depends on its position** in `array`
    - if `target` is in `array[0]`, then we need only one step
    - if `target` is in `array[i-1]`, then we need $i$ steps

## Complexity analysis

**Worst case**

- This is when `target` is in `array[size-1]`
- The algorithm needs $n$ steps
- So its complexity is $O(n)$

## Complexity analysis

**Average case**

- Assume that we always search for a `target` that **exists** in `array`
- If `target == array[i-1]` then we need $i$ steps
- Average wrt all possible positions $i$ (all are equally likely)

$$\text{Average} = \frac{1+\ldots+n}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n}{2} + \frac{1}{2}$$

- Therefore the average is $O(n)$
  - Same if we consider `target`s that don't exist in `array`

## Selection sort algorithm

```
// Ταξινομεί τον πίνακα array μεγέθους size

void selection_sort(int array[], int size) {
  // Βρίσκουμε το μικρότερο στοιχείο του πίνακα, το τοποθετούμε στη θ
  // και συνεχίζουμε με τον ίδιο τρόπο στον υπόλοιπο πίνακα.

  for (int i = 0; i < size; i++) {
      // βρίσκουμε το μικρότερο στοιχείο από αυτά σε θέσεις >= i
      int min_position = i;
      for (int j = i; j < size; j++)
          if (array[j] < array[min_position])
              min_position = j;

      // swap των στοιχείων i και min_position
      int temp = array[i];
      array[i] = array[min_position];
      a[min_position] = temp;
  }
}
```

# Complexity analysis of selection_sort

- Inner `for`
  - its body is constant: 1 step
  - $n - i$ repetitions ($n$ = `size`, $i =$current value of `i`)
  - so the whole loop takes $n - i$ steps
- Outer `for`:
  - its body takes $n - i$ steps
    - +1 for the constant swapping part (ignored compared to $n - i$)
  - first execution: $n$ steps, second: $n - 1$ steps, etc
  - Total: $n + \ldots + 1 = \frac{n(n+1)}{2}$ steps
- So the time complexity of the algorithm is $O(n^2)$

# Recursive selection_sort

Auxiliary functions

```
// Βρίσκει τη θέση του ελάχιστου στοιχείου στον πίνακα array

int find_min_position(int array[], int size) {
    int min_position = 0;

    for (int i = 1; i < size; i++)
        if (array[i] < array[min_position])
            min_position = i;

    return min_position
}

// Ανταλλάσει τα στοιχεία a,b του πίνακα array

void swap (int array[], int a, int b) {
    int temp = array[a];
    array[a] = array[b];
    array[b] = temp;
}
```

## Recursive selection_sort

Elegant recursive version of the algorithm

```
// Ταξινομεί τον πίνακα array μεγέθους size

void selection_sort(int array[], int size) {
    // Με λιγότερα από 2 στοιχεία δεν έχουμε τίποτα να κάνουμε
    if (size < 2)
        return;

    // Τοποθετούμε το ελάχιστο στοιχείο στην αρχή
    swap(array, 0, find_min_position(array, size));

    // Ταξινομούμε τον υπόλοιπο πίνακα
    selection_sort(&array[1], size - 1);
}
```

## Analysis of recursive selection_sort

- How many steps does `selection_sort` take?
  - Let $g(n)$ denote that number
- $g(0) = g(1) = 1$ (nothing to do)
- For $n > 1$ `selection_sort` calls:
  - `find_min_position`: $n$ steps
  - `swap`: 1 step (ignored compared to $n$)
  - `selection_sort`: $g(n-1)$ steps

So $g(n) = \begin{cases} n + g(n-1) & n > 1 \\ 1 & n \leq 1 \end{cases}$

## Analysis of recursive selection_sort

This is a **recurrence relation**, we can solve it by **unrolling**:

$$\begin{aligned}
g(n) &= n + g(n-1) \\
&= n + (n-1) + g(n-2) \\
&= n + (n-1) + (n-2) + g(n-3) \\
&\dots \\
&= n + \ldots + 1 \\
&= \frac{n(n+1)}{2}
\end{aligned}$$

So again we get complexity $O(n^2)$

## ADTList using Linked Lists

What is the worst case complexity of each operation?

- `list_insert_next`
- `list_remove_next`
- `list_next`
- `list_last`
- `list_find`

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*, Chapter 6.
- Robert Sedgewick. Αλγόριθμοι σε C, Κεφ. 2.

# Dynamic Arrays

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## How can we implement ADTVector?

- A Vector can be seen as an abstract resizable “array”
- So it makes sense to **implement** it using a **real array**
  - store Vector’s elements in the array
  - `vector_get_at`, `vector_set_at` are trivial
- But what about `vector_insert_last`?
  - Arrays in C have fixed size

## Dynamic arrays

- Main idea: **resize** the array
  - such arrays are called “dynamic” or “growable”
- **Problem**: we need to **copy** the previous values
- A possible algorithm for `vector_insert_last`
  - Allocate memory for `size+1` elements
  - Copy the `size` previous elements
  - Set the new element as last
  - Increase `size`
- What is the complexity of this?

## Dynamic arrays

- Main idea: **resize** the array
  - such arrays are called “dynamic” or “growable”
- **Problem**: we need to **copy** the previous values
- A possible algorithm for `vector_insert_last`
  - Allocate memory for `size+1` elements
  - Copy the `size` previous elements
  - Set the new element as last
  - Increase `size`
- What is the complexity of this?
  - $O(n)$, because of the copy!
  - Can we do better?

# Improving the complexity of insert

- **Idea**: allocate **more memory** than we need!
  - eg. allocate memory for 100 “empty” elements
    - **capacity**: total allocated memory
    - **size**: number of inserted elements
  - Insert is $O(1)$ if we have free space (just copy the new value)
- Does this change the complexity?
  - in the **worst-case**?
  - in the **average-case**?

## Improving the complexity of insert

- **Idea**: allocate **more memory** than we need!
  - eg. allocate memory for 100 “empty” elements
    - **capacity**: total allocated memory
    - **size**: number of inserted elements
  - Insert is $O(1)$ if we have free space (just copy the new value)
- Does this change the complexity?
  - in the **worst-case**?
  - in the **average-case**?
- **No**, for some values of $n$ the operation is still slow!
  - For **any values**, “average-case” makes no differece

## Amortized-time complexity

- We see here the value of **amortized-time** complexity
  - A single execution **can** be slow
  - But “most” are fast
  - In many application we only care about the **average** wrt all **executions**
- Assume we reserve 100 more elements each time
  - How many steps each insert takes on average?

## Amortized-time complexity

- We see here the value of **amortized-time** complexity
  - A single execution **can** be slow
  - But “most” are fast
  - In many application we only care about the **average** wrt all **executions**
- Assume we reserve 100 more elements each time
  - How many steps each insert takes on average?
- Intuitively: $\frac{n}{100}$. So **still** $O(n)$, same complexity!
  - Same for any **constant** number of empty elements $k$
  - Remember, complexity cares about large $n$! Think $n \gg k$
  - Can we do better?

## How to improve the complexity

- **Idea**: the number of empty elements must **depend on** $n$
  - Use more empty elements as the Vector grows!
- Standard approach: reserve $a \cdot n$ extra elements
  - for some constant $a > 1$, called the **growth factor**
- Common values
  - $a = 2$
  - $a = 1.5$
- In this class we will use $a = 2$
  - we always **double** the capacity

## A property to remember

- Consider the **geometric progression** with ratio 2

$$1, 2^1, 2^2, \ldots, 2^n$$

- Summing $n$ terms, we get the **next one minus 1**

$$1 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$$

- So each term is **larger** than **all the previous** together!
    - This is important since sereral quantities **double** in data structures

## From linear to constant time

- We always **double** the capacity
  - What is the amortized-time complexity of insert?
- We do $n$ insertions starting from an empty Vector
  - Assume the last one was “slow” (the most “unlucky” case)
- How many **steps** did we peform **in total**?
  - $n$ steps just for placing each element
  - $n$ steps for the **last resize**
  - How many for **all the prevous resizes together**?

$$\frac{n}{2} + \frac{n}{4} + \ldots + 1 = n - 1$$

- So less than $3n$ in total!
  - On average: $\frac{3n}{n} = O(1)$

- Key point: previous inserts are insignificant compared to the last one

## Removing elements

- What about `vector_remove_last`?
- Simplest strategy: just consider the removed space as “empty”
  - `vector_remove_last` is clearly worst-case $O(1)$
  - Insert is not affected (we never reduce the amount of free space)
- Commonly used in practice
  - eg. `std::vector` in C++
- **Problem**: wasted space

# Recovering wasted space

- **Idea**: if **half** of the array becomes empty, resize
  - the opposite of the doubling growing strategy
  - Is this ok?

# Recovering wasted space

- **Idea**: if **half** of the array becomes empty, resize
  - the opposite of the doubling growing strategy
  - Is this ok?

- Careful
  - this is ok if we only remove
  - but a combination of remove+insert might become slow!

- Think of the following scenario
  - Insert $n$ elements with $n = 2^k$
  - The vector is now full
  - Perform a series of: insert, remove, insert, remove, …

## Recovering wasted space

- **Better strategy**
  - when only $\frac{1}{4}$ of the array is full
  - resize to $\frac{1}{2}$ of the capacity!
  - So we still have “room” to both insert and remove
- We can show that even a combination of insert+remove is $O(1)$ amortized-time

## Implementation

Types

```
// Ένα VectorNode είναι pointer σε αυτό το struct.

struct vector_node {
    Pointer value;                  // Η τιμή του κόμβου.
};

// Ενα Vector είναι pointer σε αυτό το struct

struct vector {
    VectorNode array;               // Τα δεδομένα, πίνακας από struct ve
    int size;                       // Πόσα στοιχεία έχουμε προσθέσει
    int capacity;                   // Πόσο χώρο έχουμε δεσμεύσει
    DestroyFunc destroy_value;      // Συνάρτηση που καταστρέφει ένα στοι
};
```

## Implementation

```
Vector vector_create(int size, DestroyFunc destroy_value) {
    // Αρχικά το vector περιέχει size μη-αρχικοποιημένα στοιχεία, αλλ
    // δεσμεύουμε χώρο για τουλάχιστον VECTOR_MIN_CAPACITY για να απο
    // πολλαπλά resizes
    int capacity = size < VECTOR_MIN_CAPACITY ? VECTOR_MIN_CAPACITY :

    // Δέσμευση μνήμης, για το struct και το array.
    Vector vec = malloc(sizeof(*vec));
    VectorNode array = calloc(capacity, sizeof(*array));  // αρχικοπο

    vec->size = size;
    vec->capacity = capacity;
    vec->array = array;
    vec->destroy_value = destroy_value;

    return vec;
}
```

## Implementation

Random access is simple, since we have a real array.

```c
Pointer vector_get_at(Vector vec, int pos) {
    return vec->array[pos].value;
}

void vector_set_at(Vector vec, int pos, Pointer value) {
    // Αν υπάρχει συνάρτηση destroy_value, την καλούμε για
    // το στοιχείο που αντικαθίσταται
    if (value != vec->array[pos].value && vec->destroy_value != NULL)
        vec->destroy_value(vec->array[pos].value);

    vec->array[pos].value = value;
}
```

## Implementation

Insert, we just need to deal with resizes.

```
void vector_insert_last(Vector vec, Pointer value) {
    // Μεγαλώνουμε τον πίνακα (αν χρειαστεί), ώστε να χωράει τουλάχισ
    // στοιχεία. Διπλασιάζουμε κάθε φορά το capacity (σημαντικό για τ
    // πολυπλοκότητα!)
    if (vec->capacity == vec->size) {
        vec->capacity *= 2;
        vec->array = realloc(vec->array, vec->capacity * sizeof(*new_
    }

    // Μεγαλώνουμε τον πίνακα και προσθέτουμε το στοιχείο
    vec->array[vec->size].value = value;
    vec->size++;
}
```

# Takeaways

- **Dynamic arrays** are the standard way to implement ADTVector
- Insert is $O(1)$
  - but **amortized-time**!
  - would you use a dynamic array in the software controlling an Airbus?
- Remove is also $O(1)$
  - also amortized, if we care about recovering wasted space
- Random access (get/set) is always worst-case $O(1)$

# Binary Trees, Heaps

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

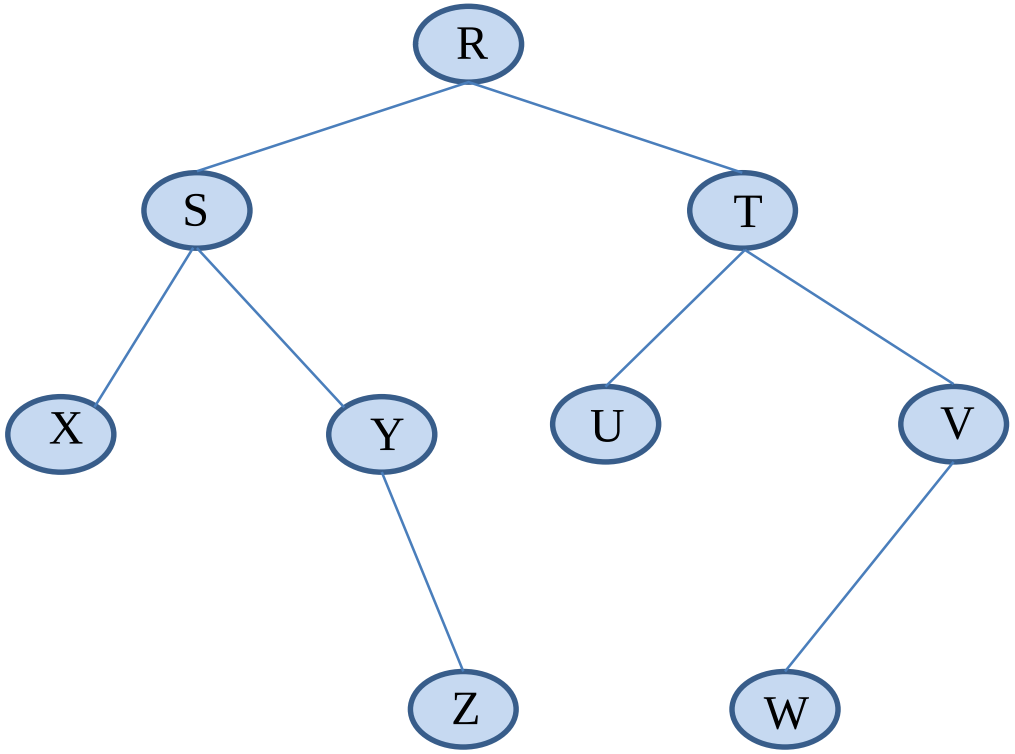
Κώστας Χατζηκοκολάκης

## Binary trees

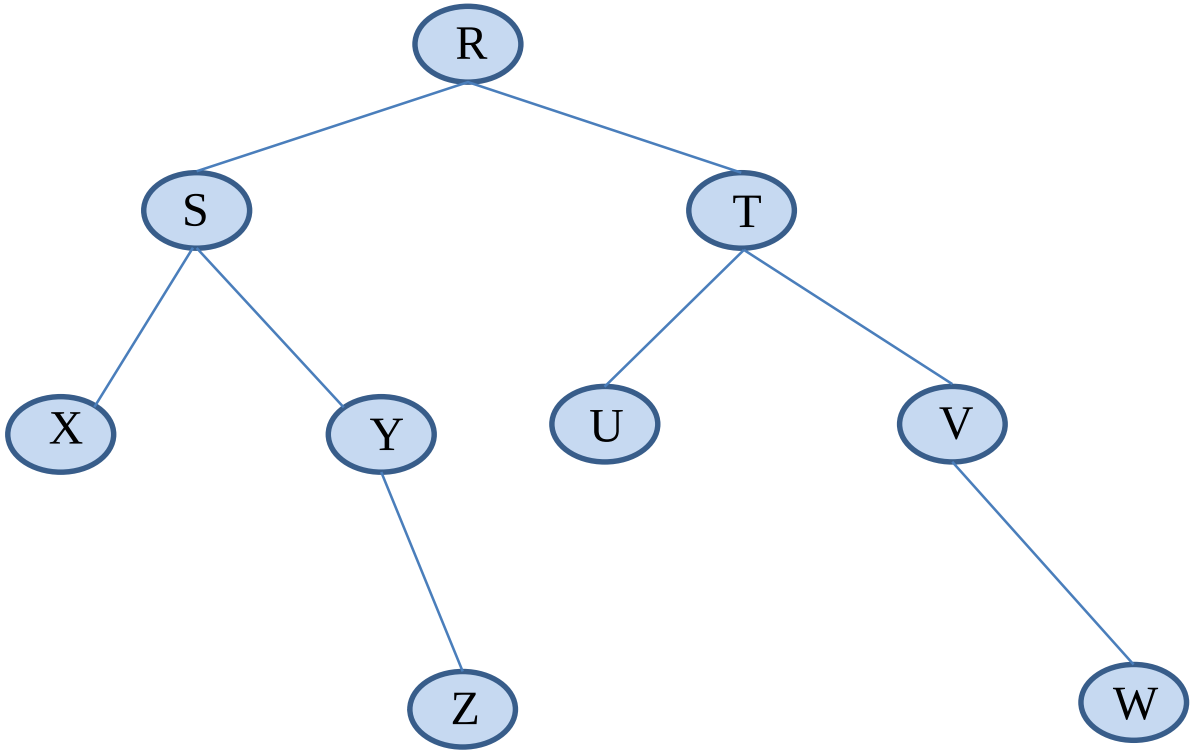
A **binary tree (δυαδικό δέντρο)** is a set of nodes such that:

- Exactly one node is called the **root**
- All nodes except the root have **exactly one parent**
- Each node has **at most two children**
    - and the are **ordered**: called **left** and **right**

## Example: a binary tree



## Example: a different binary tree



Whether a child is left or right matters.

## Terminology

- **path**: sequence of nodes traversing from parent to child (or vice-versa)
- **length** of a path: number of nodes -1 (= number of “moves” it contains)
- **siblings**: children of the same parent
- **descendants**: nodes reached by travelling downwards along any path
- **ancestors**: nodes reached by travelling upwards towards the root
- **leaf / external node**: a node without children
- **internal node**: a node with children

## Terminology

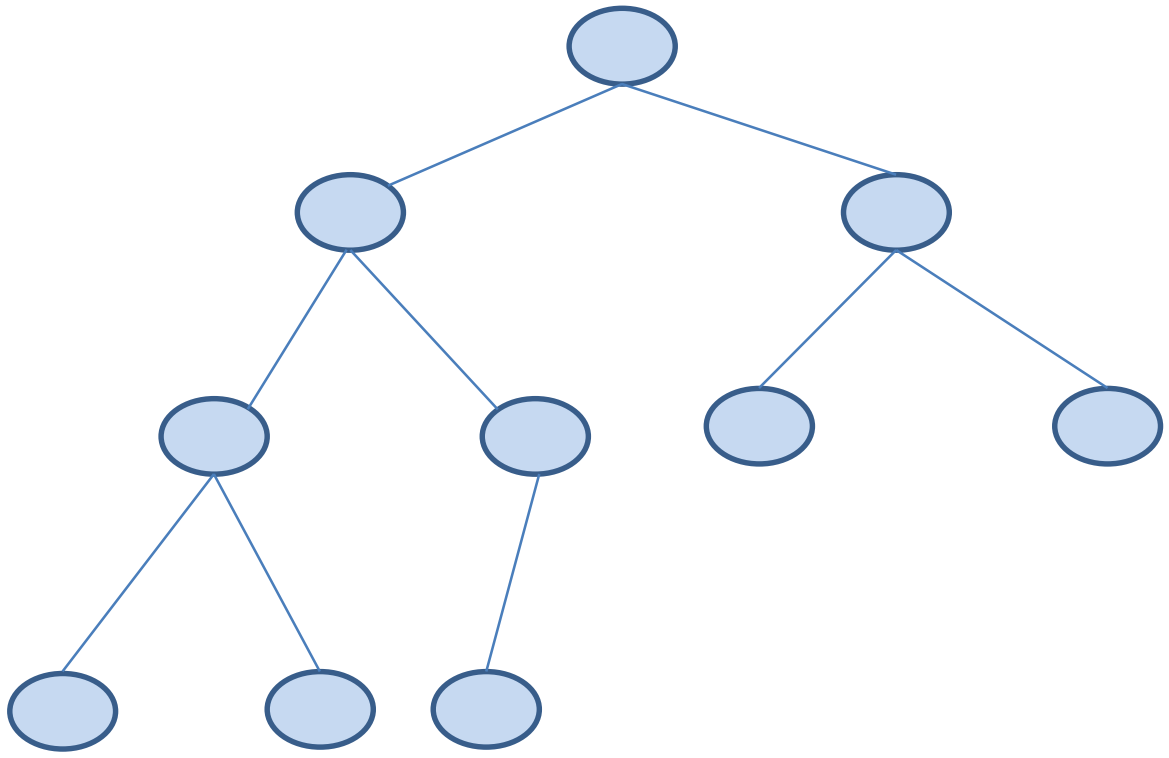
- Nodes tree can be arranged in **levels / depths**:
  - The root is at **level 0**
  - Its children are at **level 1**, their children are at **level 2**, etc.
- Note: node level = length of the (unique) path from the root to that node
- **height** of the tree: the largest depth of any node
- **subtree** rooted at a node: the tree consisting of that node and its descendants

## Complete binary trees

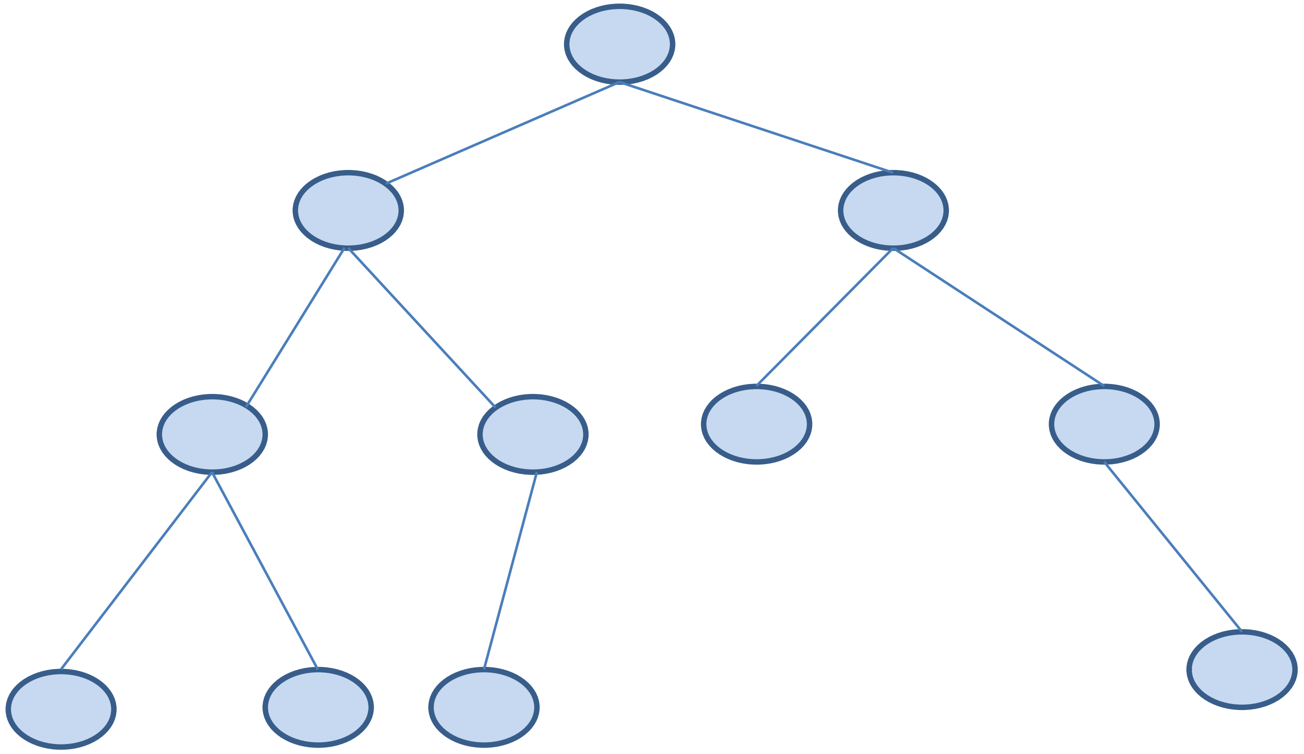
A binary tree is called **complete** (πλήρες) if

- All levels except the last are **“full”** (have the maximum number of nodes)
- The nodes at the last level fill the level “from left to right”

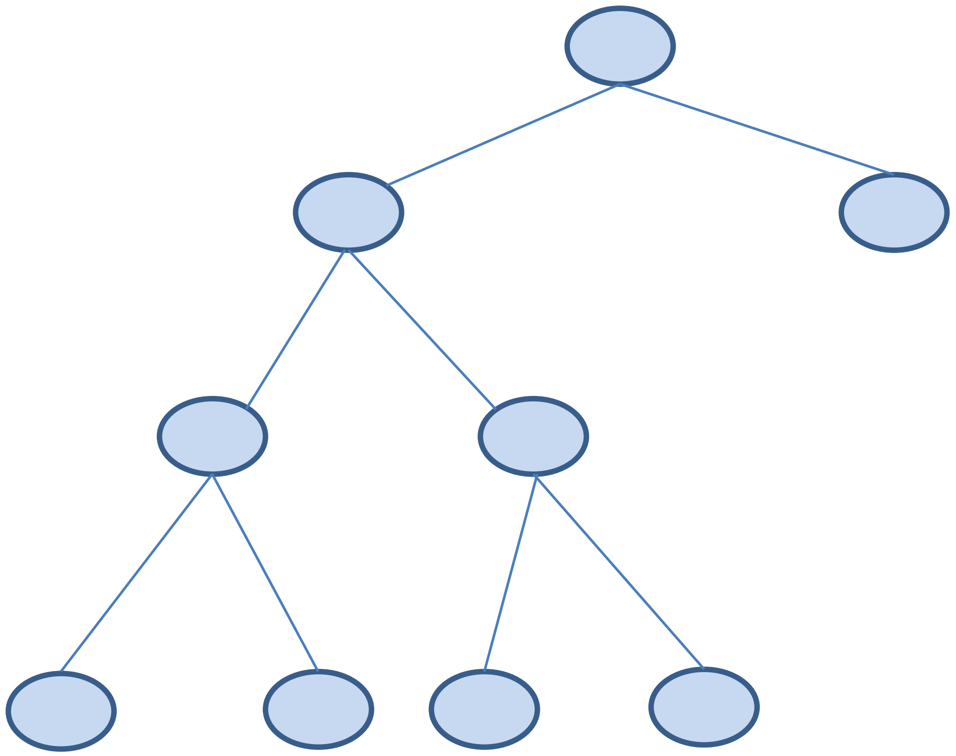
## Example: complete binary tree



## Example: not complete binary tree

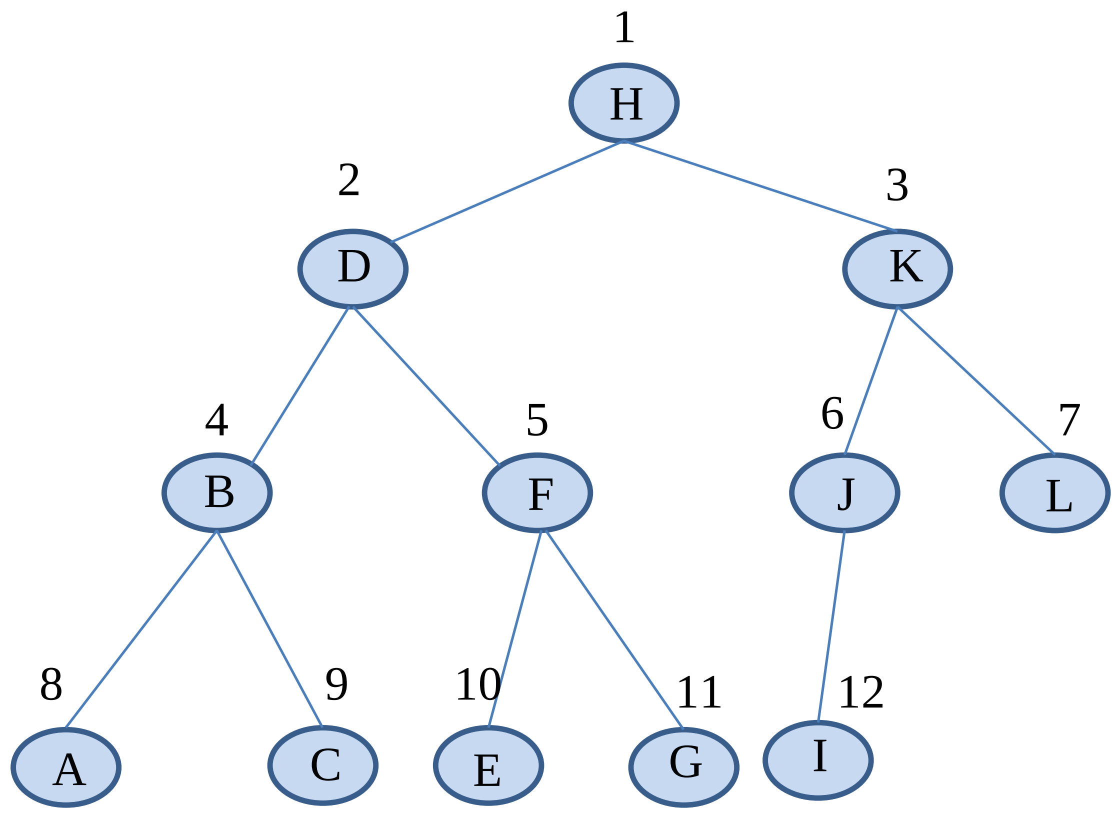


## Example: not complete binary tree



## Level order

Ordering the nodes of a tree **level-by-level** (and left-to-right in each level).



## Nodes of a complete binary tree

- How many nodes does a complete binary tree have at each level?
- At most
  - $1$ at level $0$.
  - $2$ at level $1$.
  - $4$ at level $2$.
  - $\ldots$
  - $2^k$ at level $k$ .

## Properties of binary trees

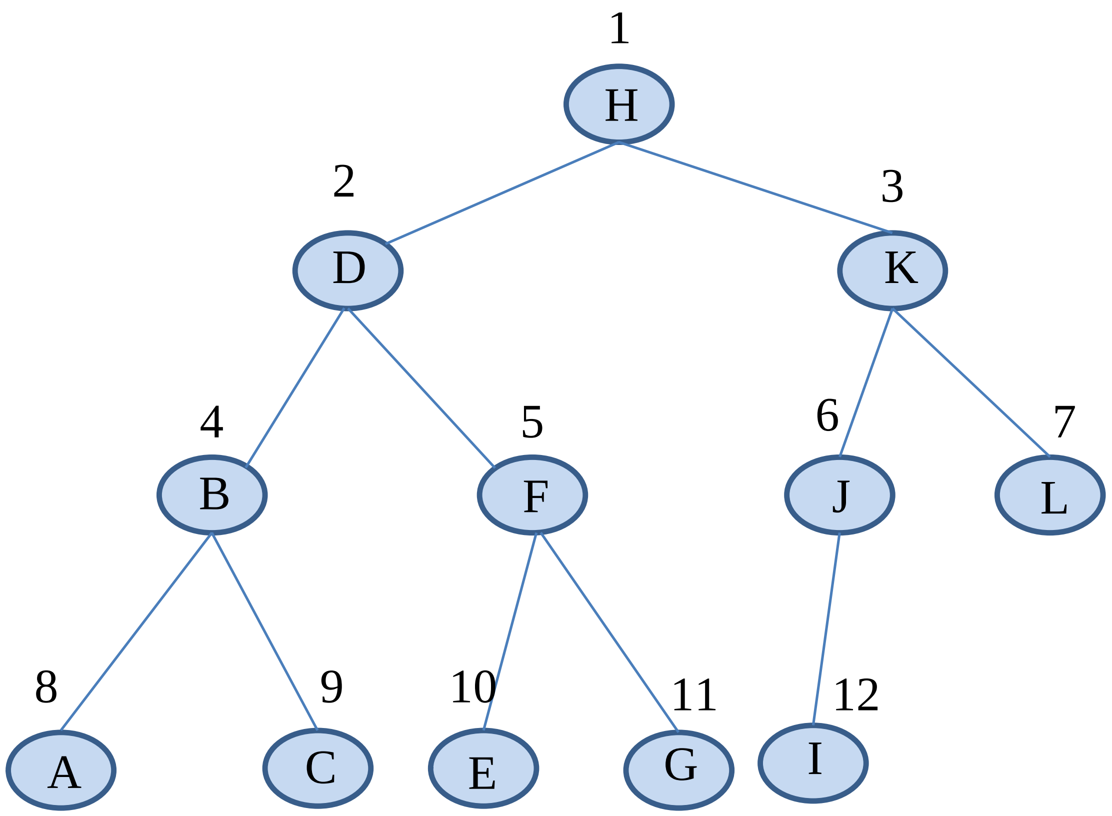
- The following hold:
  - $h + 1 \leq n \leq 2^{h+1} - 1$
  - $1 \leq n_E \leq 2^h$
  - $h \leq n_I \leq 2^h - 1$
  - $\log(n+1) - 1 \leq h \leq n - 1$
- Where
  - $n$: number of all nodes
  - $n_I$: number of internal nodes
  - $n_E$: number of external nodes (leaves)
  - $h$: height

## Properties of complete binary trees

$h \leq \log n$

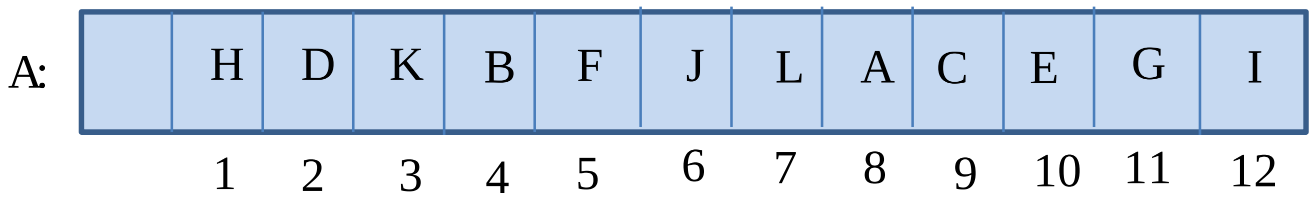
- Very important property, the tree cannot be too “tall”!
- Why?
  - Any level $l < h$ contains exactly $2^l$ nodes
  - Level $h$ contains at least one node
  - So $1 + 2 + \ldots + 2^{h-1} + 1 = 2^h \leq n$
  - And take logarithms on both sides

# How do we represent a binary tree?



# Sequential representation

Store the entries in an **array** at **level order**.



- Common for **complete trees**
- A lot of **space** is wasted for non-complete trees
  - missing nodes will have empty slots in the array

## How to find nodes

| **To Find:** | **Use** | **Provided** |
|---|---|---|
| The left child of $A[i]$ | $A[2i]$ | $2i \leq n$ |
| The right child of $A[i]$ | $A[2i+1]$ | $2i+1 \leq n$ |
| The parent of $A[i]$ | $A[i/2]$ | $i > 1$ |
| The root | $A[1]$ | $A$ is nonempty |
| Whether $A[i]$ is a leaf | | $2i > n$ |

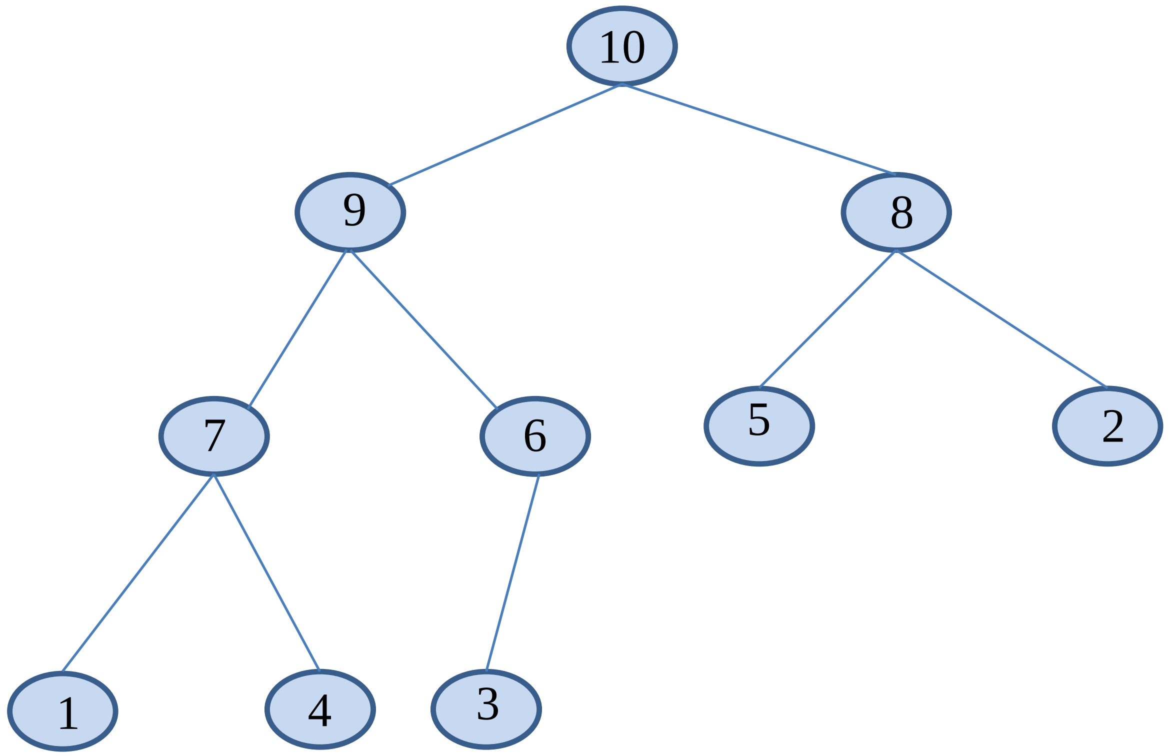
## Heaps

A binary tree is called a **heap** (σωρός) if

- It is **complete**, and
- each node is **greater or equal than its children**

(Sometimes this is called a **max-heap**, we can similarly define a min-heap)

## Example



## Heaps and priority queues

- Heaps are a common data structure for implementing **Priority Queues**
- The following operations are needed
  - find max
  - insert
  - remove max
  - create with data
- We need to **preserve the heap property** in each operation!

# Find max

- Trivial, the max is always **at the root**
  - remember: we always preserve the heap property
- Complexity?

## Inserting a new element

- The new element can only be inserted at the **end**
  - because a heap must be a **complete** tree
- Now all nodes **except the last** satisfy the heap property
  - to restore it: apply the `bubble_up` algorithm on the last node

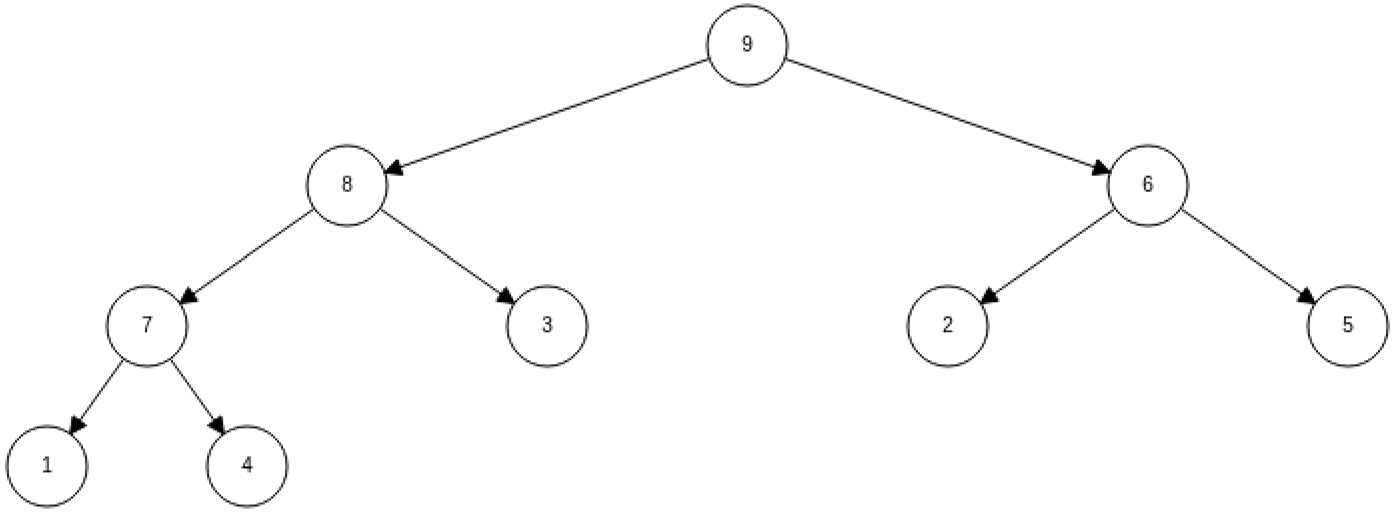
## Inserting a new element

`bubble_up(node)`

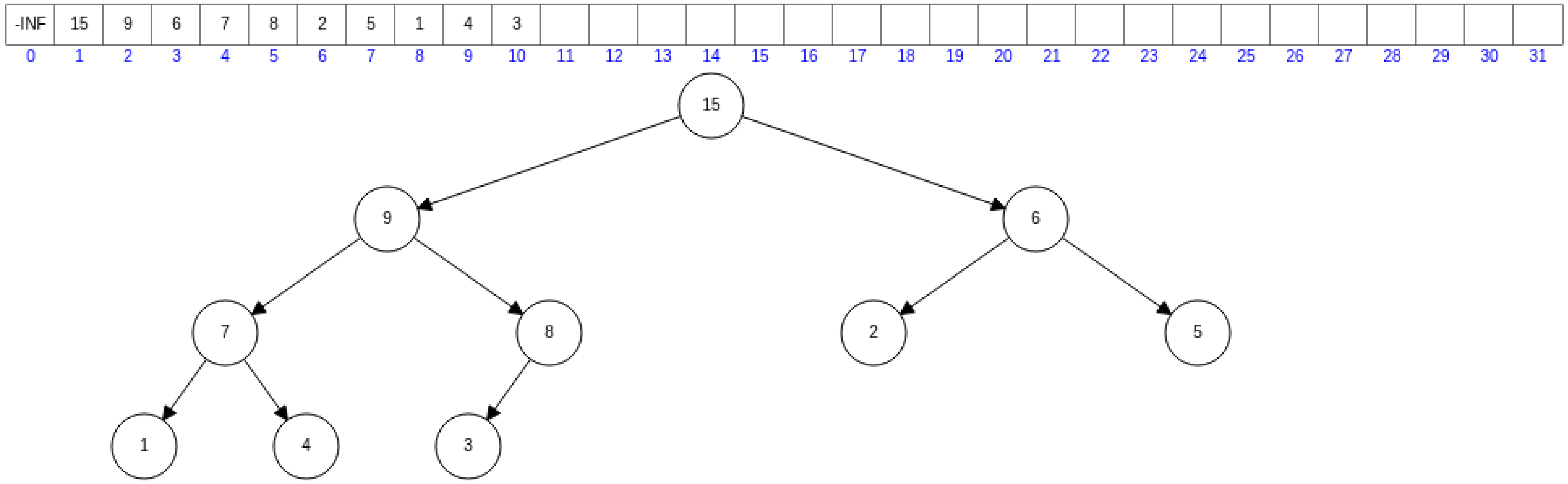
- **Before**
  - `node` might be **larger** than its parent
  - all other nodes satisfy the heap property
- **After**
  - all nodes satisfy the heap property
- **Algorithm**
  - if `node > parent`
    - **swap them** and call `bubble_up(parent)`

# Example insertion

| -INF | 9 | 8 | 6 | 7 | 3 | 2 | 5 | 1 | 4 | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |



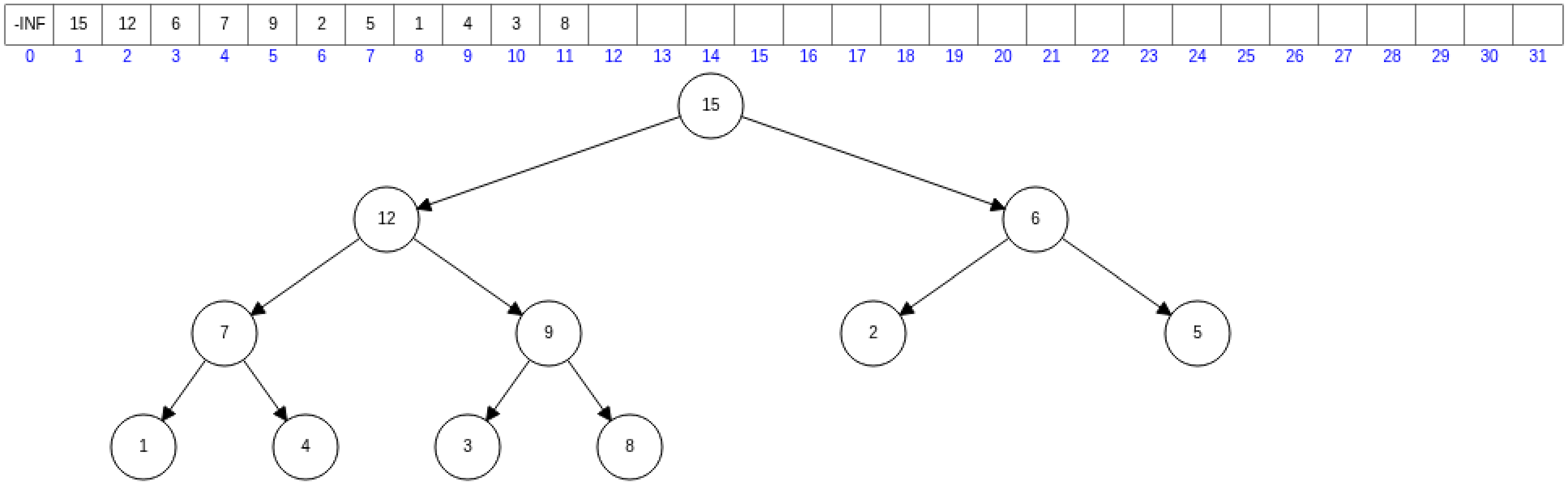
# Example insertion



| -INF | 15 | 9 | 6 | 7 | 8 | 2 | 5 | 1 | 4 | 3 | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

Inserting 15 and running bubble_up

# Example insertion



| -INF | 15 | 12 | 6 | 7 | 9 | 2 | 5 | 1 | 4 | 3 | 8 | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

Inserting 12 and running bubble_up

## Complexity of insertion

- We travel the tree from the last node to the root
  - on each node: 1 step (constant time)
- So we need at most $O(h)$ steps
  - $h$ is the height of the tree
  - but $h \leq \log n$ on a **complete tree**
- So $O(\log n)$
  - the “complete” property is crucial!

## Removing the max element

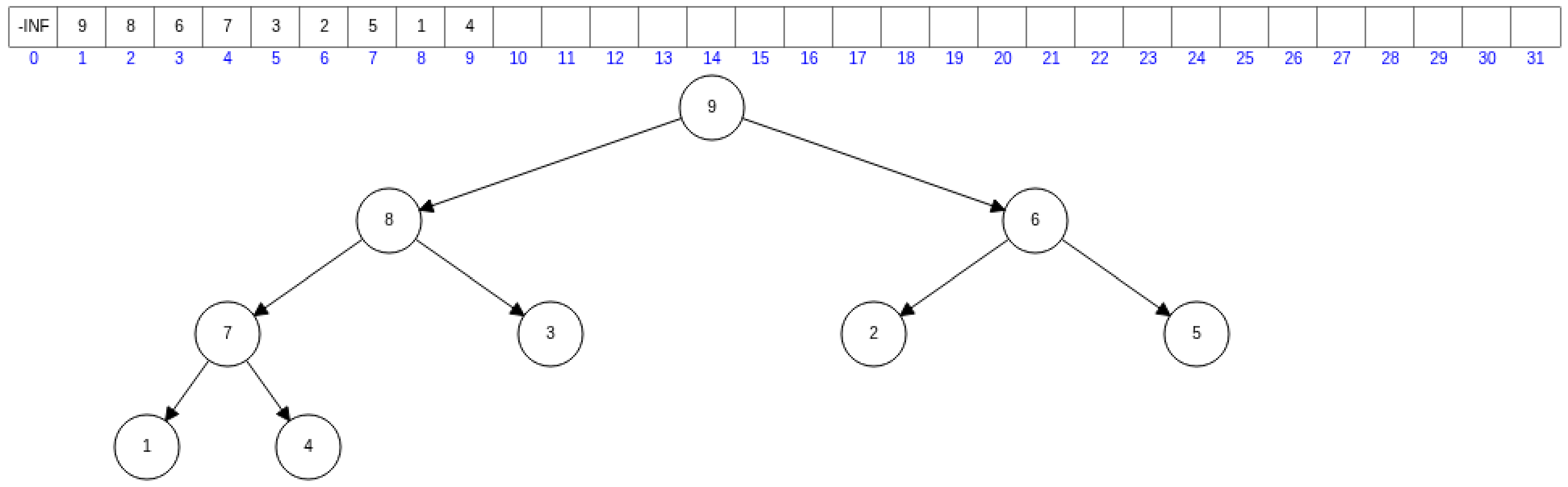
- We want to remove the root
  - but the heap must be a **complete** tree
- So **swap** the root with the **last** element
  - then remove the last element
- Now all nodes **except the root** satisfy the heap property
  - to restore it: apply the `bubble_down` algorithm on the root

## Removing the max element

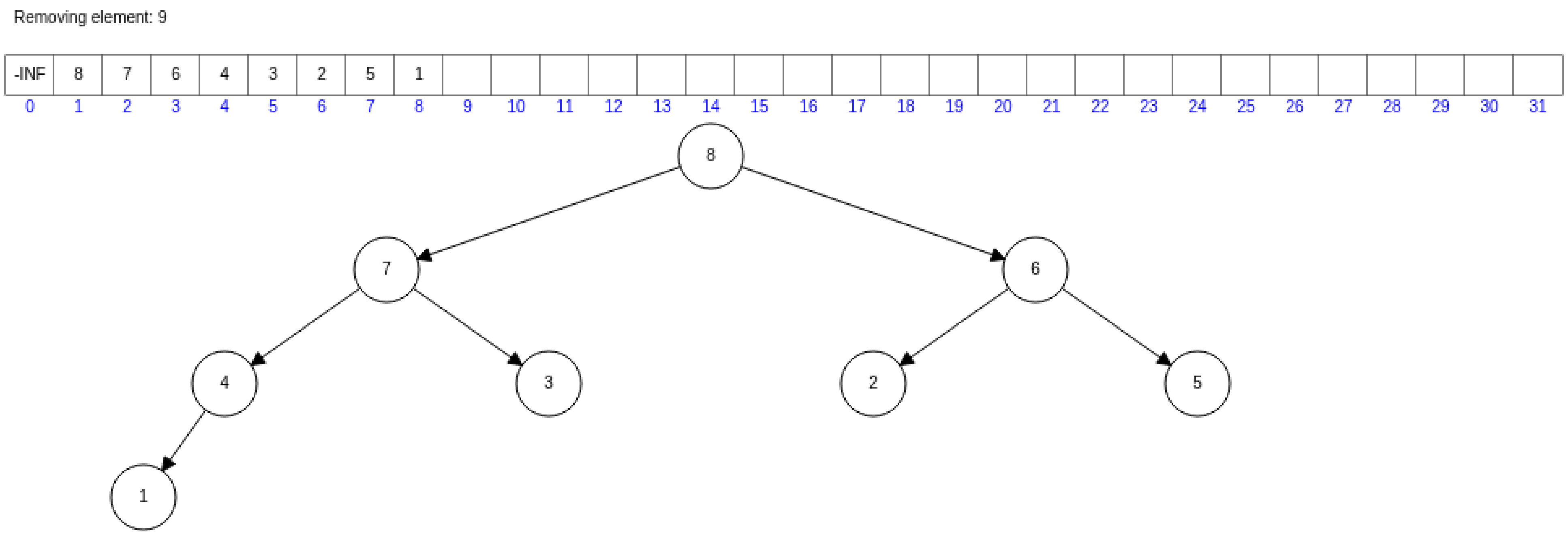
`bubble_down(node)`

- **Before**
  - `node` might be **smaller** than any of its children
  - all other nodes satisfy the heap property
- **After**
  - all nodes satisfy the heap property
- **Algorithm**
  - `max_child` = the **largest child** of `node`
  - If `node < max_child`
    - **swap them** and call `bubble_down(max_child)`

## Example removal



# Example removal



Removing 9 and restoring the heap property

## Complexity of removal

- We travel a single path from the root to a leaf
- So we need at most $O(h)$ steps
  - $h$ is the height of the tree
- Again $O(\log n)$
  - again, having a complete tree is crucial

# Building a heap from initial data

- What if we want to create a heap that contains some **initial values**?
    - we call this operation **heapify**
- “Naive” implementation:
    - Create an empty heap and **insert elements one by one**
- What is the complexity of this implementation?
    - We do $n$ inserts
    - Each insert is $O(\log n)$ (because of `bubble_up`)
    - So $O(n \log n)$ total
- Worst-case example?
    - sorted elements: each value with have to fully `bubble_up` to the root

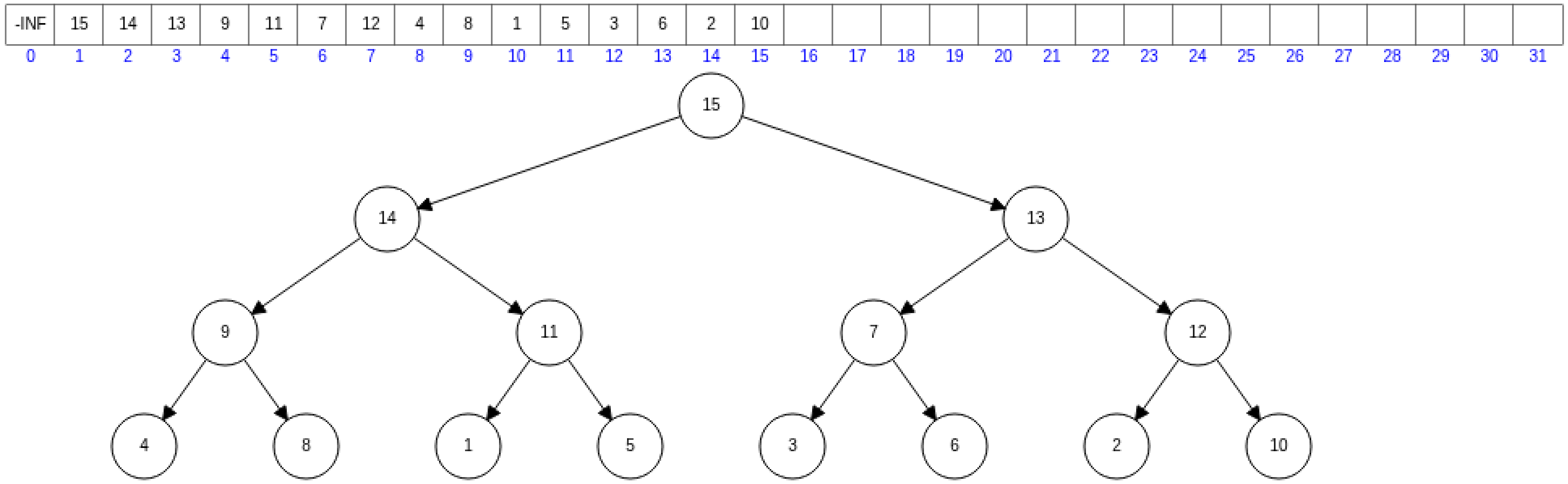
# Efficient heapify

- Better algorithm:
  - Visit all **internal nodes** in **reverse level order**
    - last internal node: $\frac{n}{2}$ (parent of the last leaf $n$)
    - first internal node: 1 (root)
  - Call `bubble_down` on each visited `node`
- Why does this work?
  - when we visit `node`, its **subtree is already a heap**
    - except from `node` **itself** (the precondition of `bubble_down`)
  - So `bubble_down` restores the heap property **in the subtree**
  - After processing the root, the whole tree is a heap

# Heapify example

| | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| -INF | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

# Heapify example



Visit internal nodes in inverse level order, call bubble_down.

## Complexity of heapify

- We call `bubble_down` $\frac{n}{2}$ times
    - So $O(n \log n)$?
- But this is only an **upper-bound**
    - `bubble_down` is faster **closer to the leaves**
    - and **most nodes** live there!
    - we might be over-approximating the number of steps

## Complexity of heapify

- More careful calculation of the number of steps:
  - If `node` is at level $l$, `bubble_down` takes at most $h - l$ steps
  - At most $2^l$ nodes at this level, so $(h - l)2^l$ steps for level $l$
  - For the whole tree: $\sum_{l=0}^{h-1} (h - l)2^l$
  - This can be shown to be less than $2n$ (exercise if you're curious)
- So we get worst-case $O(n)$ complexity

# Efficient vs naive heapify

- For `naive_heapify` we found $O(n \log n)$
  - maybe we are also over-approximating?
- No: in the worst-case (sorted elements) we really need $n \log n$ steps
  - try to compute the exact number of steps
- The difference:
  - `bubble_up` is faster closer to the **root**, but **few** nodes live there
  - `bubble_down` is faster closer to the **leaves**, and **most** nodes live there
- Note: in the **average-case**, the naive version is also $O(n)$

# Implementing ADTPriorityQueue

Types

```
// Ενα PriorityQueue είναι pointer σε αυτό το struct

struct priority_queue {
    Vector vector;              // Τα δεδομένα, σε Vector για μεταβλη
    CompareFunc compare;        // Η διάταξη
    DestroyFunc destroy_value;  // Συνάρτηση που καταστρέφει ένα στοι
};
```

## ADTPriorityQueue implementation

Types.

```
// Ενα PriorityQueue είναι pointer σε αυτό το struct

struct priority_queue {
    Vector vector;                 // Τα δεδομένα, σε Vector για μεταβλη
    CompareFunc compare;           // Η διάταξη
    DestroyFunc destroy_value;     // Συνάρτηση που καταστρέφει ένα στοι
};
```

## ADTPriorityQueue implementation

Finding the max is trivial.

```
Pointer pqueue_max(PriorityQueue pqueue) {
    return node_value(pqueue, 1);       // root
}
```

# ADTPriorityQueue implementation

For `pqueue_insert`, the non-trivial part is `bubble_up`.

```
// Αποκαθιστά την ιδιότητα του σωρού.
// Πριν: όλοι οι κόμβοι ικανοποιούν την ιδιότητα του σωρού, εκτός από
//       τον node που μπορεί να είναι _μεγαλύτερος_ από τον πατέρα το
// Μετά: όλοι οι κόμβοι ικανοποιούν την ιδιότητα του σωρού.

static void bubble_up(PriorityQueue pqueue, int node) {
    // Αν φτάσαμε στη ρίζα, σταματάμε
    if (node == 1)
        return;

    int parent = node / 2;      // Ο πατέρας του κόμβου. Τα node ids

    // Αν ο πατέρας έχει μικρότερη τιμή από τον κόμβο, swap και συνεχ
    if (pqueue->compare(node_value(pqueue, parent), node_value(pqueue
        node_swap(pqueue, parent, node);
        bubble_up(pqueue, parent);
    }
}
```

## ADTPriorityQueue implementation

```
// Πριν: όλοι οι κόμβοι ικανοποιούν την ιδιότητα του σωρού, εκτός από
//       node που μπορεί να είναι _μικρότερος_ από κάποιο από τα παιδ
// Μετά: όλοι οι κόμβοι ικανοποιούν την ιδιότητα του σωρού.

static void bubble_down(PriorityQueue pqueue, int node) {
    // βρίσκουμε τα παιδιά του κόμβου (αν δεν υπάρχουν σταματάμε)
    int left_child = 2 * node;
    int right_child = left_child + 1;
    int size = pqueue_size(pqueue);
    if (left_child > size)
        return;

    // βρίσκουμε το μέγιστο από τα 2 παιδιά
    int max_child = left_child;
    if (right_child <= size && pqueue->compare(node_value(pqueue, lef
            max_child = right_child;

    // Αν ο κόμβος είναι μικρότερος από το μέγιστο παιδί, swap και συ
    if (pqueue->compare(node_value(pqueue, node), node_value(pqueue,
        node_swap(pqueue, node, max_child);
        bubble_down(pqueue, max_child);
    }
}
```

# Other possible representations

| Operation | Heap | Sorted List | Unsorted Vector |
|---|---|---|---|
| `pqueue_create` (with data) | $O(n)$ | $O(n \log n)$ | $O(1)$ |
| `pqueue_remove` | $O(\log n)$ | $O(1)$ | $O(n)$ |
| `pqueue_insert` | $O(\log n)$ | $O(n)$ | $O(1)$ |

All of them have **some** advantage

- Heaps provide a great compromise between insertions and removals

## Using ADTPriorityQueue for sorting

- We can easily sort data using ADTPriorityQueue
  - create a priority queue with the data
  - remove elements in sorted order
- When ADTPriorityQueue is implemented by a **heap**
  - this algorithm is called **heapsort**
  - and runs in time $O(n \log n)$

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*. Chapter 9. Sections 9.1 to 9.6.

- R. Sedgewick. *Αλγόριθμοι σε C*. Κεφ. 5 και 9.

Proofs of given statements can be found in the following book:

- M. T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++*. 2nd edition. John Wiley and Sons, 2011.

# Binary Search Trees

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Search

- Searching for a specific value within a large collection is fundamental
- We want this to be efficient even if we have billions of values!
- So far we have seens two basic search strategies:
  - **sequential** search: slow
  - **binary** search: fast
    - but only for **sorted** data

## Sequential search

```c
// Αναζητά τον ακέραιο target στον πίνακα target. Επιστρέφει
// τη θέση του στοιχείου αν βρεθεί, διαφορετικά -1.

int sequential_search(int target, int array[], int size) {
  for (int i = 0; i < size; i++)
      if (array[i] == target)
          return i;

  return -1;
}
```

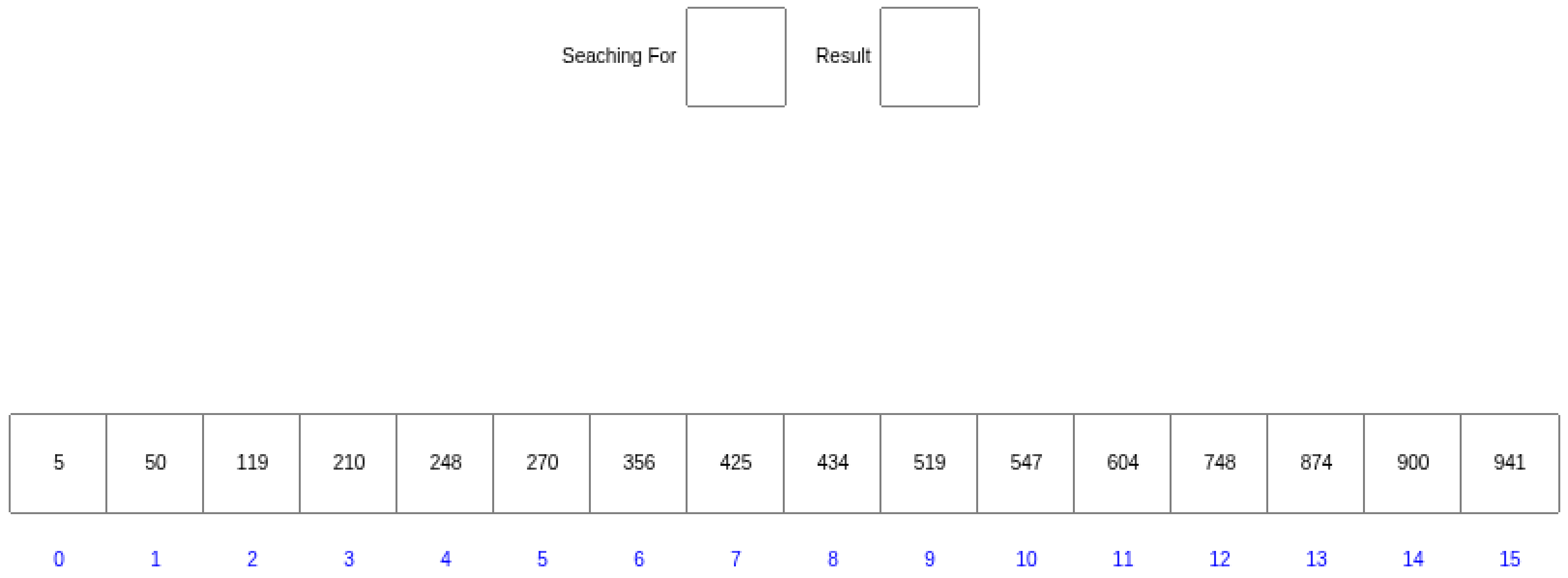
We already saw that the complexity is $O(n)$.

## Binary search

```c
// Αναζητά τον ακέραιο target στον __ταξινομημένο__ πίνακα target.
// Επιστρέφει τη θέση του στοιχείου αν βρεθεί, διαφορετικά -1.

int binary_search(int target, int array[], int size) {
    int low = 0;
    int high = size - 1;

    while (low <= high) {
        int middle = (low + high) / 2;

        if (target == array[middle])
            return middle;              // βρέθηκε
        else if (target > array[middle])
            low = middle + 1;           // συνεχίζουμε στο πάνω μισο
        else
            high = middle - 1;          // συνεχίζουμε στο κάτω μισό
    }

    return -1;
}
```

**Important**: the array needs to be **sorted**

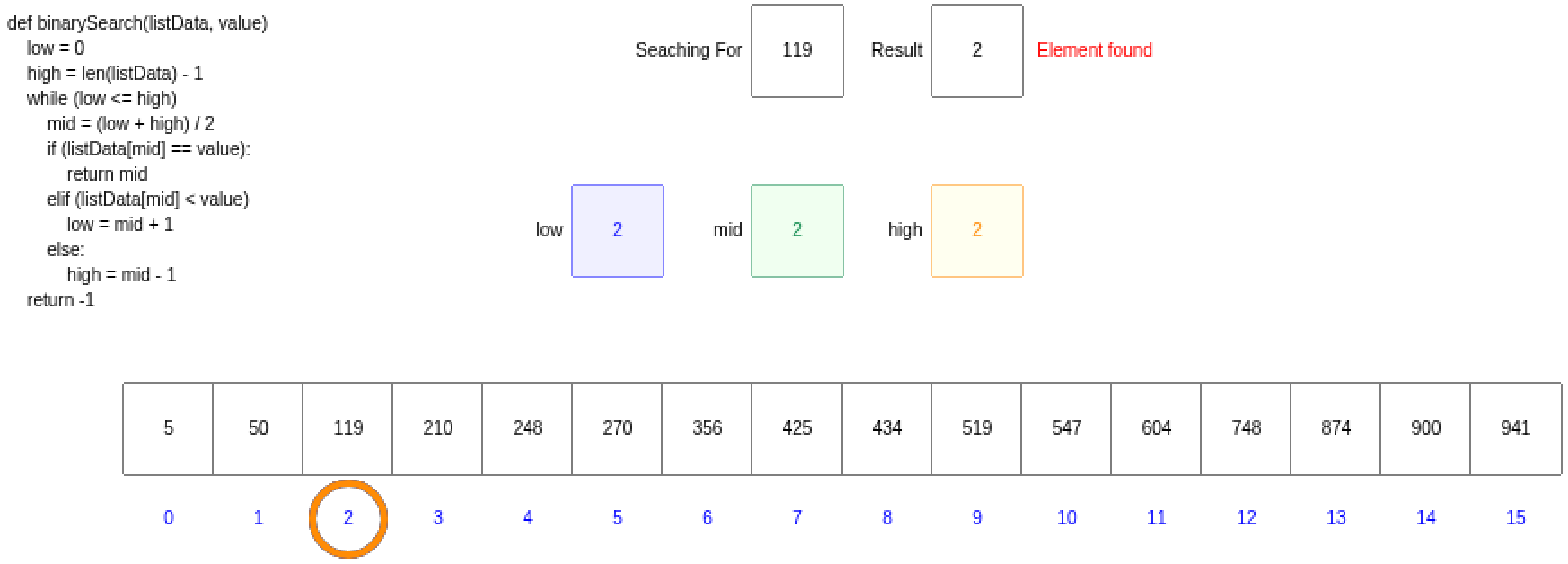
## Binary search example



| 5 | 50 | 119 | 210 | 248 | 270 | 356 | 425 | 434 | 519 | 547 | 604 | 748 | 874 | 900 | 941 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

At each step the search space is cut in half.

# Binary search example



```
def binarySearch(listData, value)
  low = 0
  high = len(listData) - 1
  while (low <= high)
    mid = (low + high) / 2
    if (listData[mid] == value):
      return mid
    elif (listData[mid] < value)
      low = mid + 1
    else:
      high = mid - 1
  return -1
```

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 5 | 50 | 119 | 210 | 248 | 270 | 356 | 425 | 434 | 519 | 547 | 604 | 748 | 874 | 900 | 941 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |

At each step the search space is cut in half.

## Complexity of binary search

- **Search space**: the elements remaining to search
  - those between `low` and `right`
- The size of the search space is **cut in half** at each step
  - After step $i$ there are $\frac{n}{2^i}$ elements remaining
- We **stop** when $\frac{n}{2^i} < 1$
  - in other words when $n < 2^i$
  - or equivalently when $\log n < i$
- So we will do at most $\log n$ steps
  - complexity $O(\log n)$
  - **30 steps** for one **billion** elements

## Conclusions

- Binary search is fundamental for efficient search
- But we need **sorted data**
- Maintaining a sorted array **after an insert** is hard
  - complexity?
- How can we keep data sorted **and simultaneously** allow efficient inserts?

## Binary Search Trees (BST)

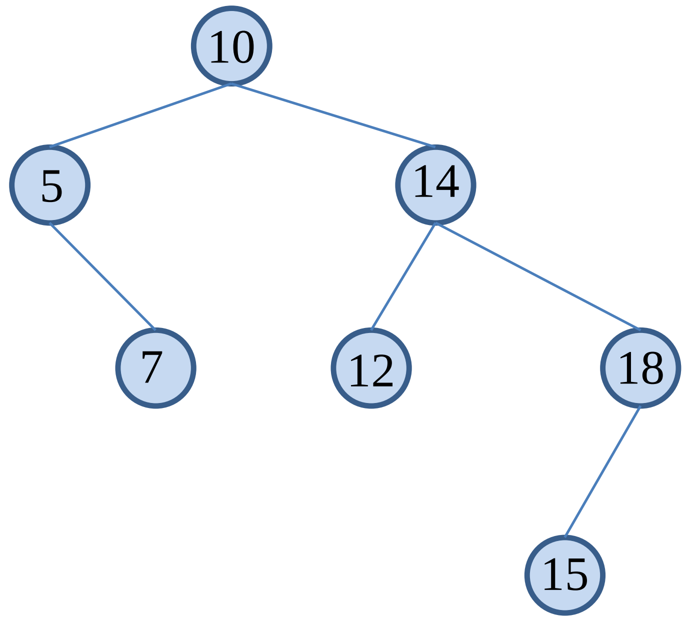
A **binary search tree** (δυαδικό δέντρο αναζήτησης) is a binary tree such that:

- every node is **larger** than all nodes on its **left subtree**
- every node is **smaller** than all nodes on its **right subtree**

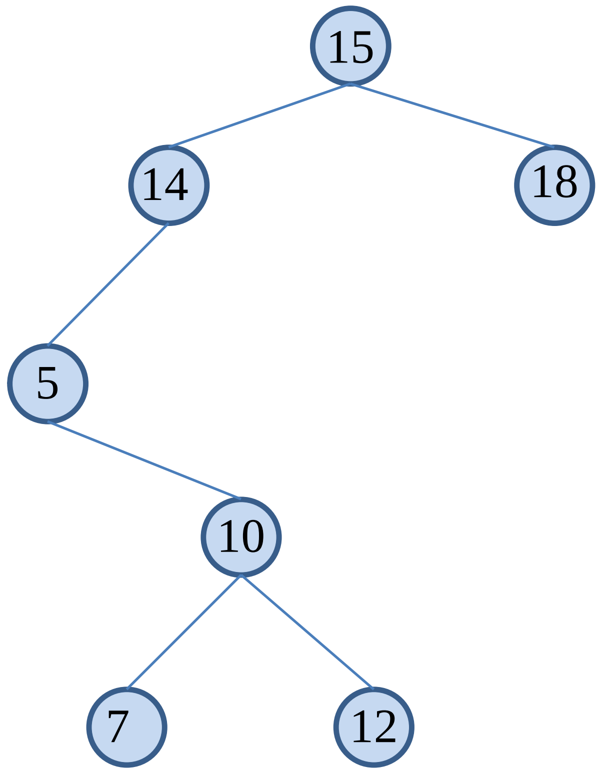
Note

- No value can appear **twice**
  (it would violate the definition)
- **Any** `compare` function can be used for ordering.
  (with some mathematical constraints, see the piazza post)

## Example

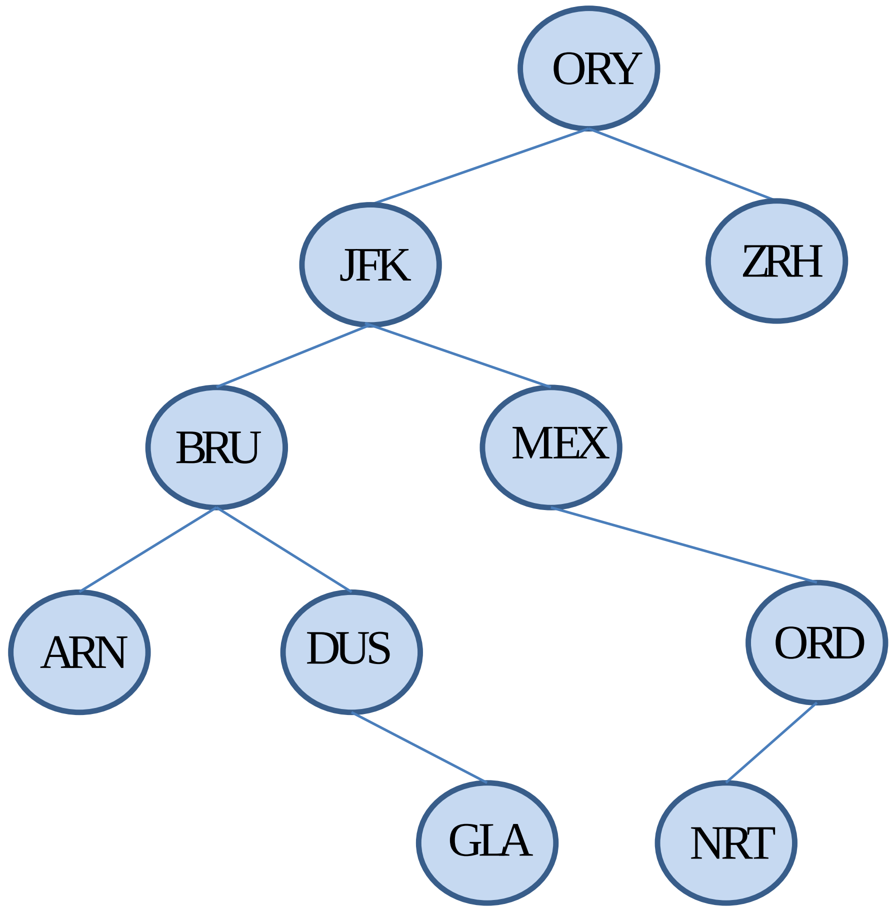


## Example



A different tree with the **same values**!

## Example



## BST operations

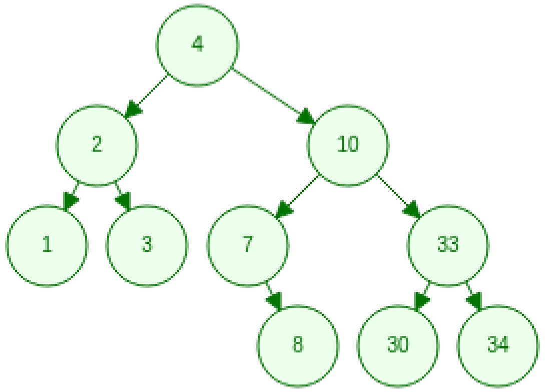
- Container operations
  - **Insert** / **Remove**
- **Search** for a given value
- **Ordered** traversal
  - Find **first** / **last**
  - Find **next** / **previous**
- So we can use BSTs to implement
  - **ADTMap** (we need search)
  - **ADTSet** (we need search and ordered traversal)

## Search

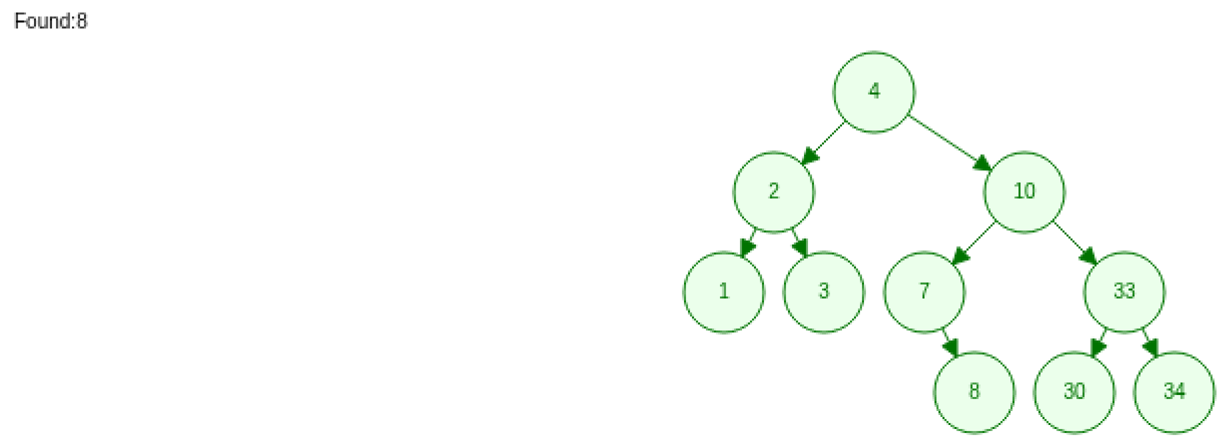
We perform the following procedure **starting at the root**

- If the tree is empty
  - `target` does not exist in the tree
- If `target` = `current_node`
  - Found!
- If `target` < `current_node`
  - continue in the **left subtree**
- If `target` > `current_node`
  - continue in the **right subtree**

## Search example



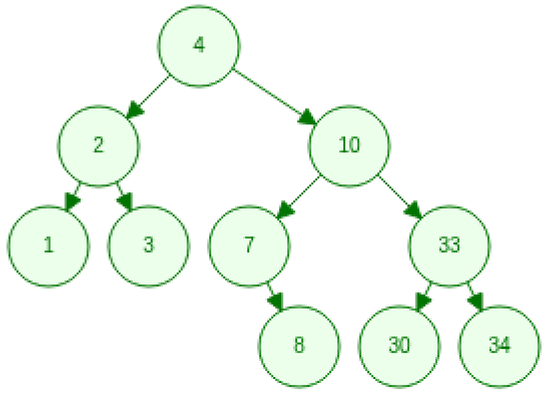
# Search example



Searching for 8

## Search example

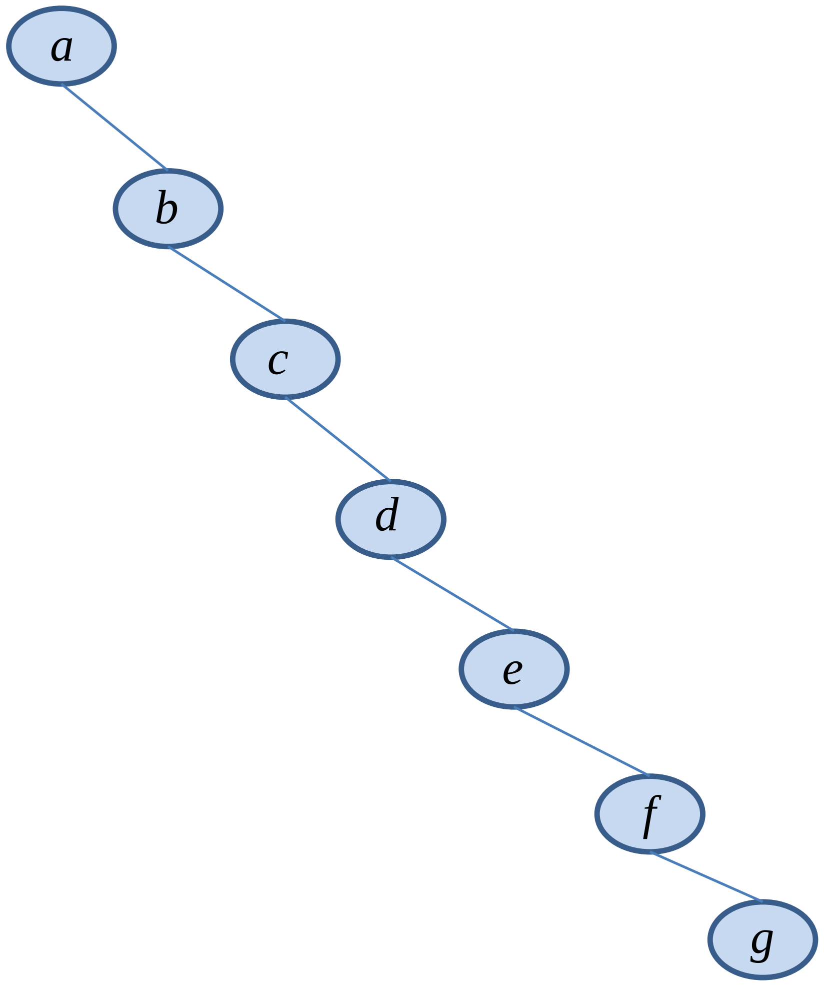
Found:8



## Complexity of search

- How many steps will we make in the worst case?
  - We will traverse a path from the root to the tree
  - $h$ steps max (the **height** of the tree)
- But how does $h$ relate to $n$?
  - h = $O(n)$ in the worst case!
  - when the tree is essentially a degenerate "list"

## Searching in this tree is slow



## Complexity of search

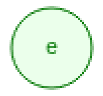
- This is a very common pattern in trees
  - Many operations are $O(h)$
  - Which means worst-case $O(n)$
- Unless we manage to **keep the tree short**!
  - We already saw this in **complete** trees, in which $h \le \log n$
- Unfortunately maintaining a complete BST is not easy (why?)
  - But there are other methods to achieve the same result
    - AVL, B-Trees, etc
  - We will talk about them later

## Inserting a new value

- Inserting a `value` is **very similar to search**
- We follow the same algorithm as if we were searching for `value`
  - If `value` is found we stop (no duplicates!)
  - If we reach an **empty subtree** insert `value` **there**

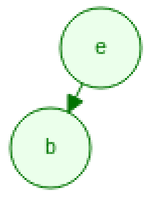
## Insert example

## Insert example



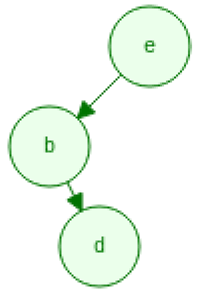
Inserting e

## Insert example



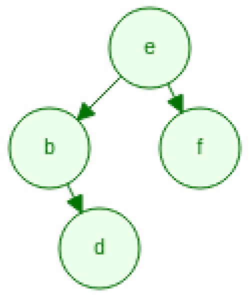
Inserting b

## Insert example



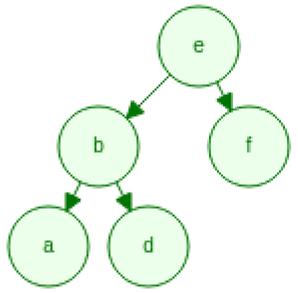
Inserting d

## Insert example



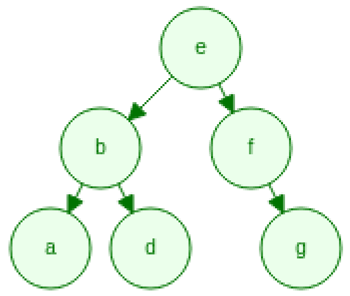
Inserting f

## Insert example



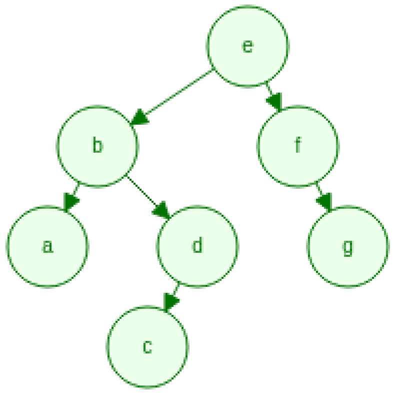
Inserting a

## Insert example



Inserting g

## Insert example



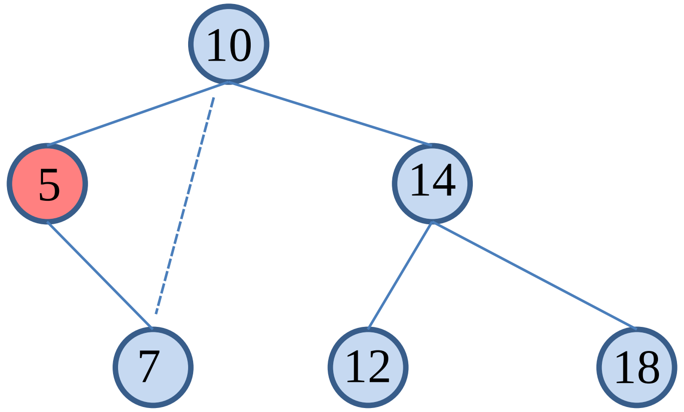
Inserting c

## Complexity of insert

- Same as **search**
- $O(h)$
  - So $O(n)$ unless the tree is short

## Deleting a value

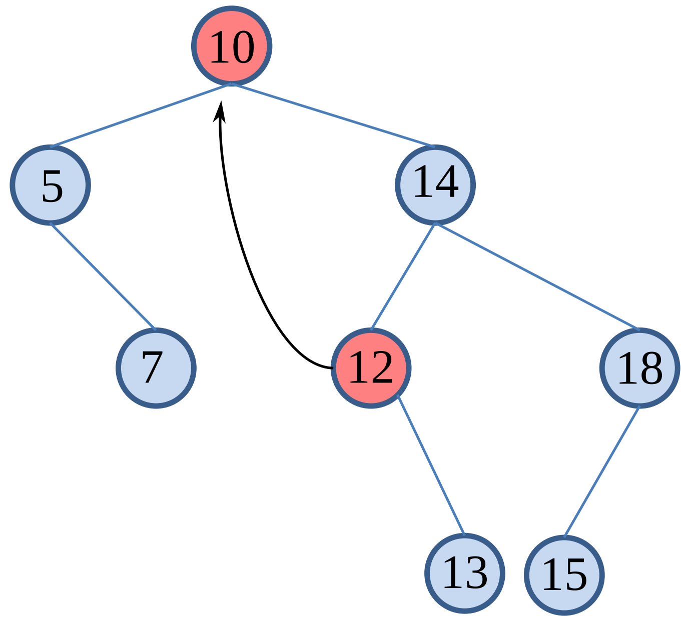
- We might want to delete **any node** in a BST
- Easy case: `node` has **as most 1 child**
- Connect the child directly to `node`'s **parent**
- BST property is preserved (why?)



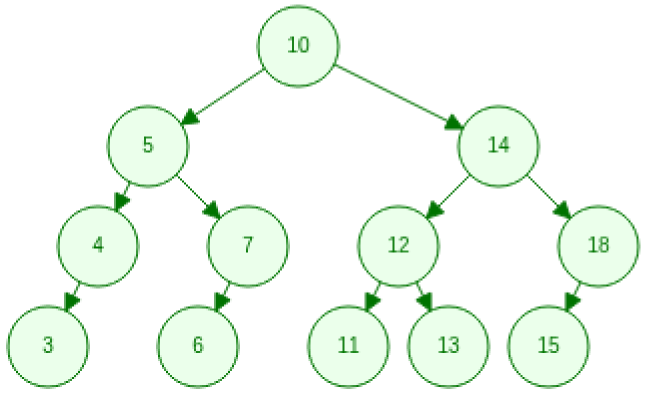
## Deleting a value

- Hard case: `node` has **two children** (eg. 10)
- Find the `next` node in the order (eg. 12)
  - **left-most** node in the right sub-tree!

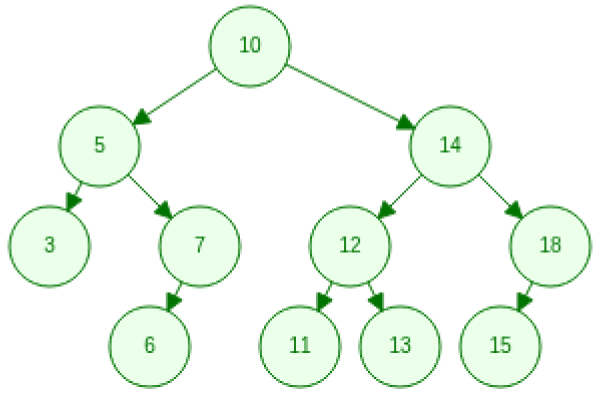
  (or equivalently the `previous` node)
- We can replace `node`'s value with `next`'s
  - this preserves the BST property (why?)
- And then delete `next`
  - This has to be an **easy** case (why?)



## Delete example

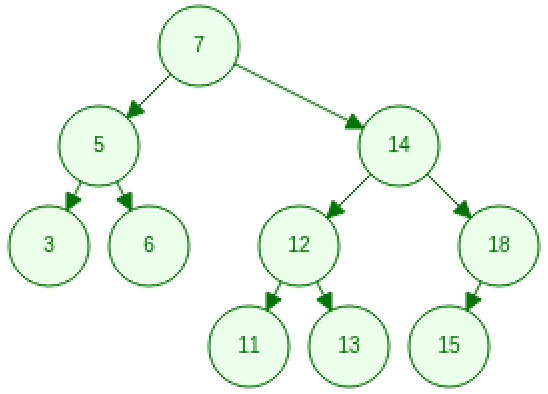


## Delete example



Delete 4 (easy).

## Delete example


$$\begin{tikzcd}[column sep=tiny]
 & & & 7 \arrow[dll] \arrow[drr] & & & \\
 & 5 \arrow[dl] \arrow[dr] & & & & 14 \arrow[dl] \arrow[dr] & \\
3 & & 6 & & 12 \arrow[dl] \arrow[dr] & & 18 \arrow[dl] \\
 & & & 11 & & 13 \quad 15 &
\end{tikzcd}$$

Delete 10 (hard). Replace with 7 and it becomes easy.

## Complexity of delete

- Finding the node to delete is $O(h)$
- Finding the `next` / `previous` is also $O(h)$

## Ordered traversal: first/last

- How to find the **first** node?
  - simply follow left children
  - $O(h)$
  - same for **last**

## Ordered traversal: next

- How to find the **next** of a given `node`?
- Easy case: the node has a right child
    - find the left-most node of the right subtree
    - we used this for **delete**!
- Hard case: no right-child, we need to go up!

# Ordered traversal: next

General algorithm for any node.
Perform the following procedure **starting at the root**

```
// Ψευδοκώδικας, current_node είναι η ρίζα του τρέχοντος υποδέντρου,
// node είναι ο κόμβος του οποίου τον επόμενο ψάχνουμε.

find_next(current_node, node) {
    if (node == current_node) {
        // Ο target είναι η ρίζα του υποδέντρου, ο επόμενος είναι ο μ
        // του δεξιού υποδέντρου (αν είναι κενό τότε δεν υπάρχει επόμ
        return node_find_min(right_child);      // NULL αν δεν υπάρχε

    } else if (node > current_node)) {
        // Ο target είναι στο αριστερό υποδέντρο,
        // οπότε και ο προηγούμενός του είναι εκεί.
        return node_find_next(node->right, compare, target);

    } else {
        // Ο target είναι στο αριστερό υποδέντρο, ο επόμενός του μπορ
        // επίσης εκεί, αν όχι ο επόμενός του είναι ο ίδιος ο node.
        res = node_find_next(node->left, compare, target);
        return res != NULL ? res : node;
    }
}
```

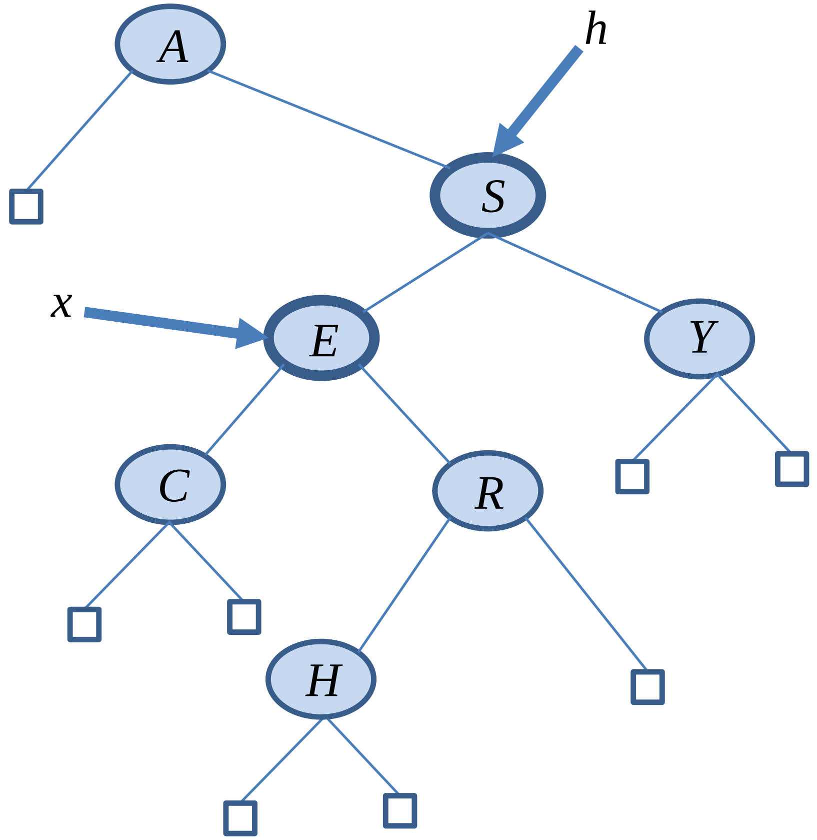
## Complexity of next

- Similar to search, traversing the tree from the root to the leaves
  - so $O(h)$
- We can do it faster by keeping more structure
- We can keep a bidirectional list of all nodes in order
  - $O(1)$ to find next, no extra complexity to update
- More advanced: keep a **link to the parent**
  - Find the next by going **up** when needed
  - Can you find the algorithm?
  - Real-time complexity is still $O(h)$ if we traverse to the root
  - But what about amortized-time?

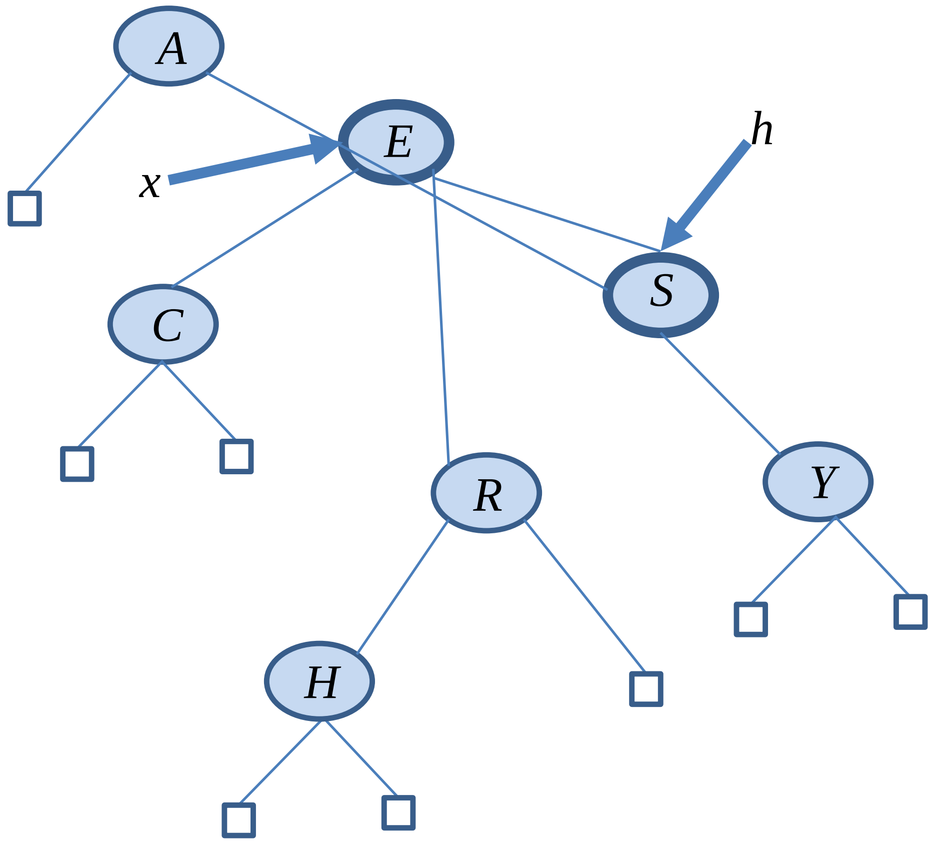
## Rotations

- **Rotation (περιστροφή)** is a fundamental operation in BSTs
  - swaps the role of a **node and one of its children**
  - while still **preserving the BST property**
- **Right rotation**
  - swap a node $h$ and its **left child** $x$
  - $x$ becomes the root of the subtree
  - the **right** child of $x$ becomes **left** child of $h$
  - $h$ becomes a **right** child of $x$
- **Left rotation**
  - symmetric operation with **right** child

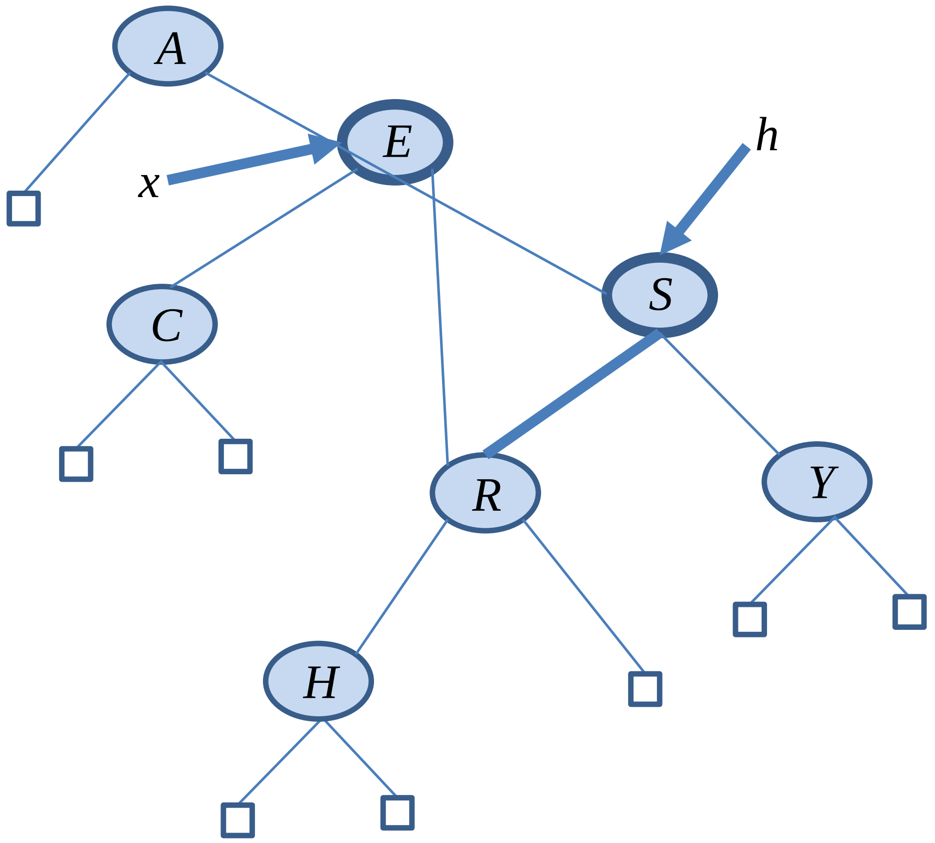
## Example: right rotation



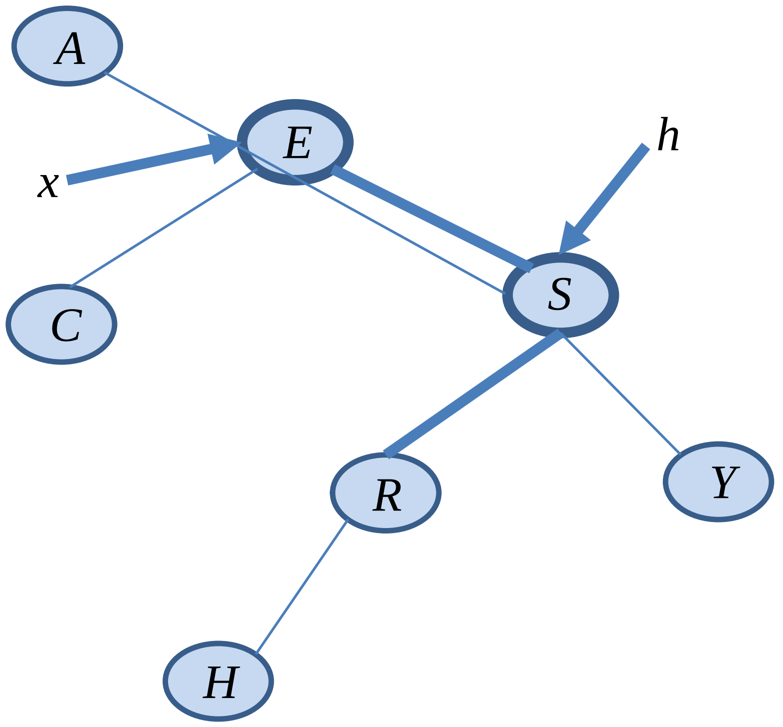
## Example: right rotation



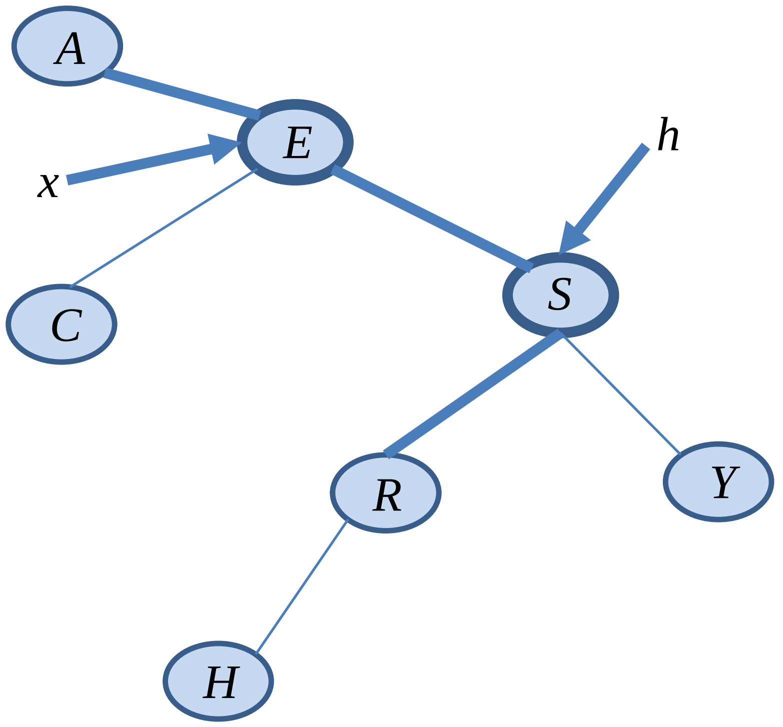
## Example: right rotation



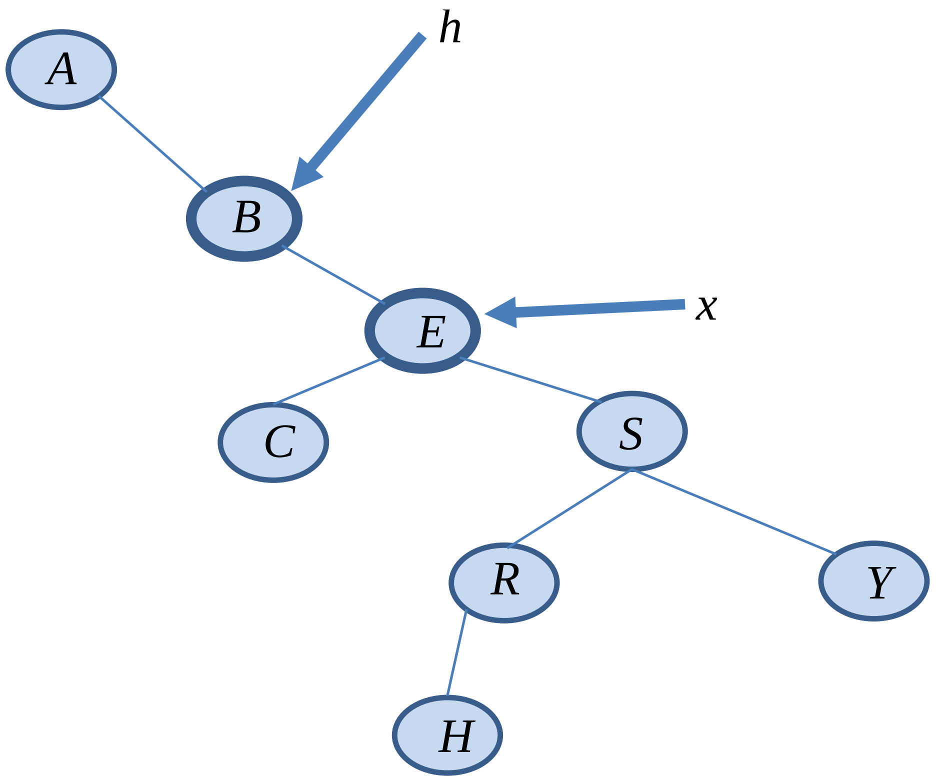
## Example: right rotation



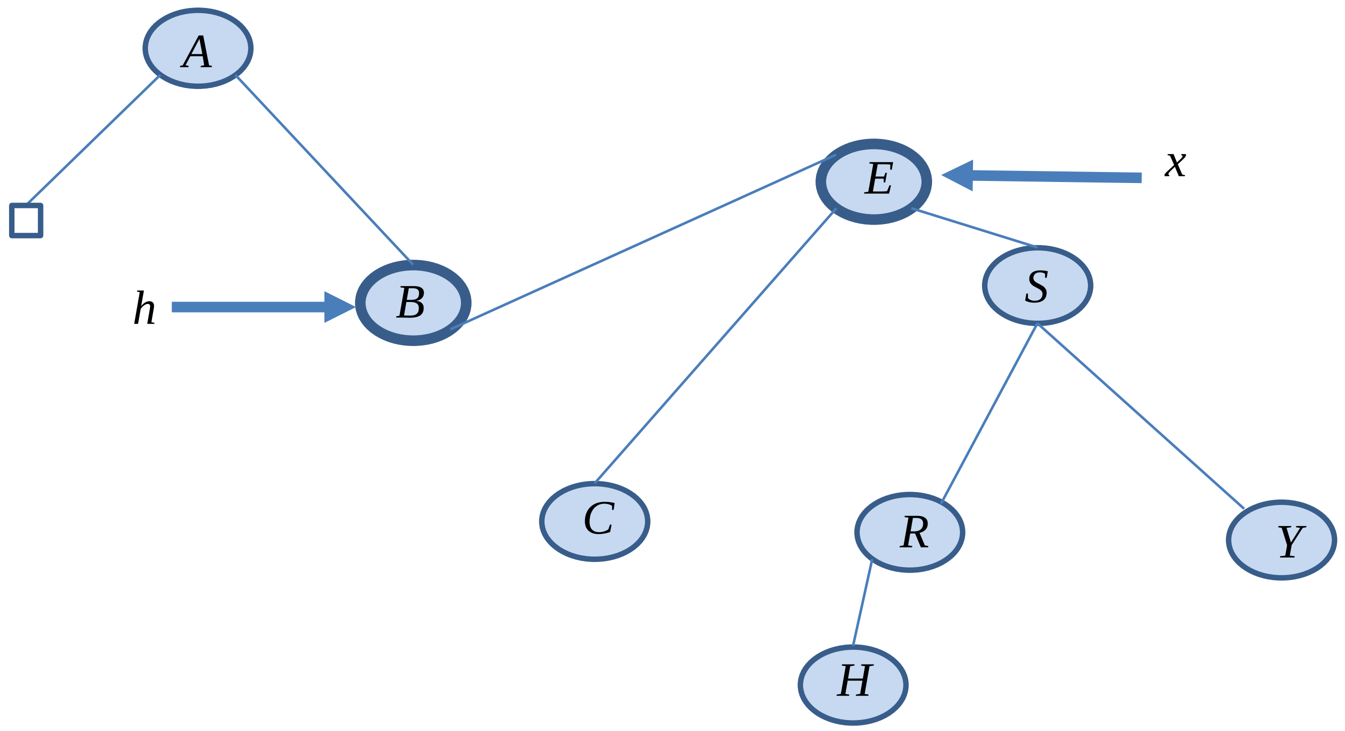
# Example: right rotation



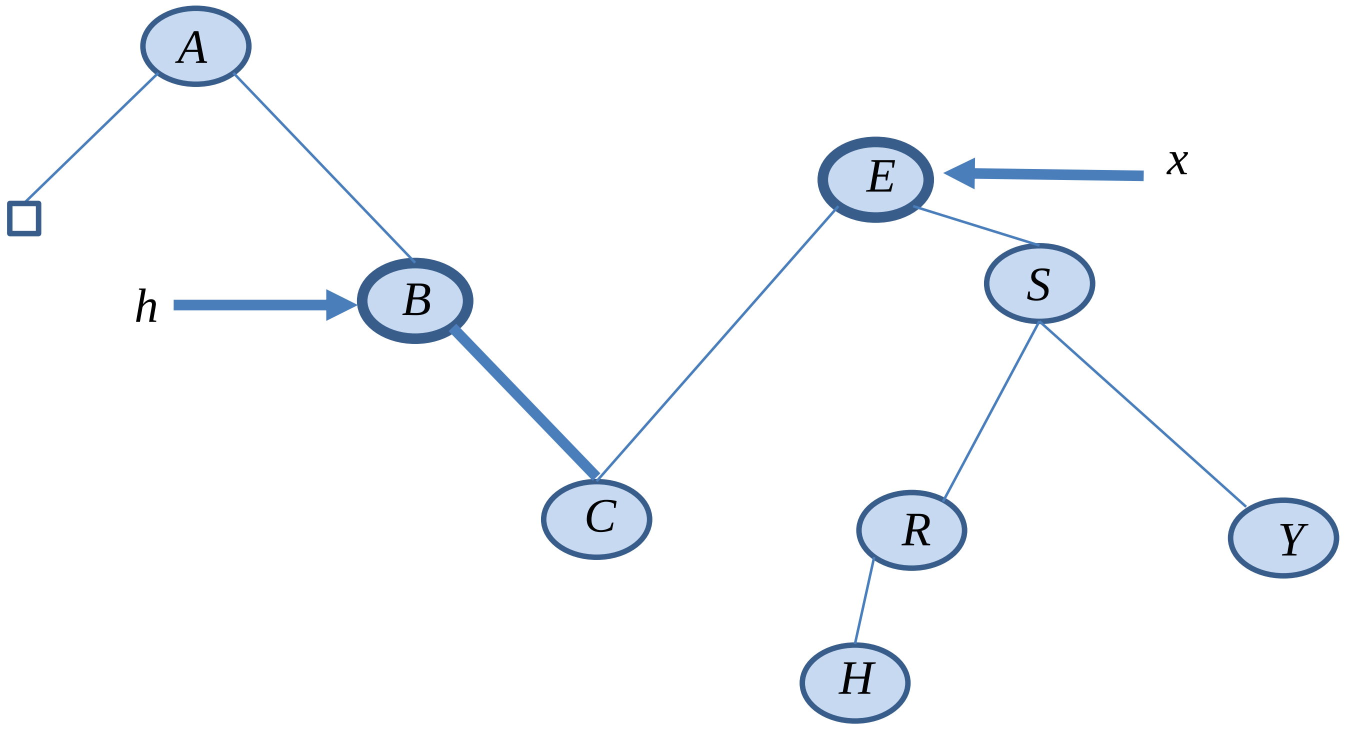
## Example: left rotation



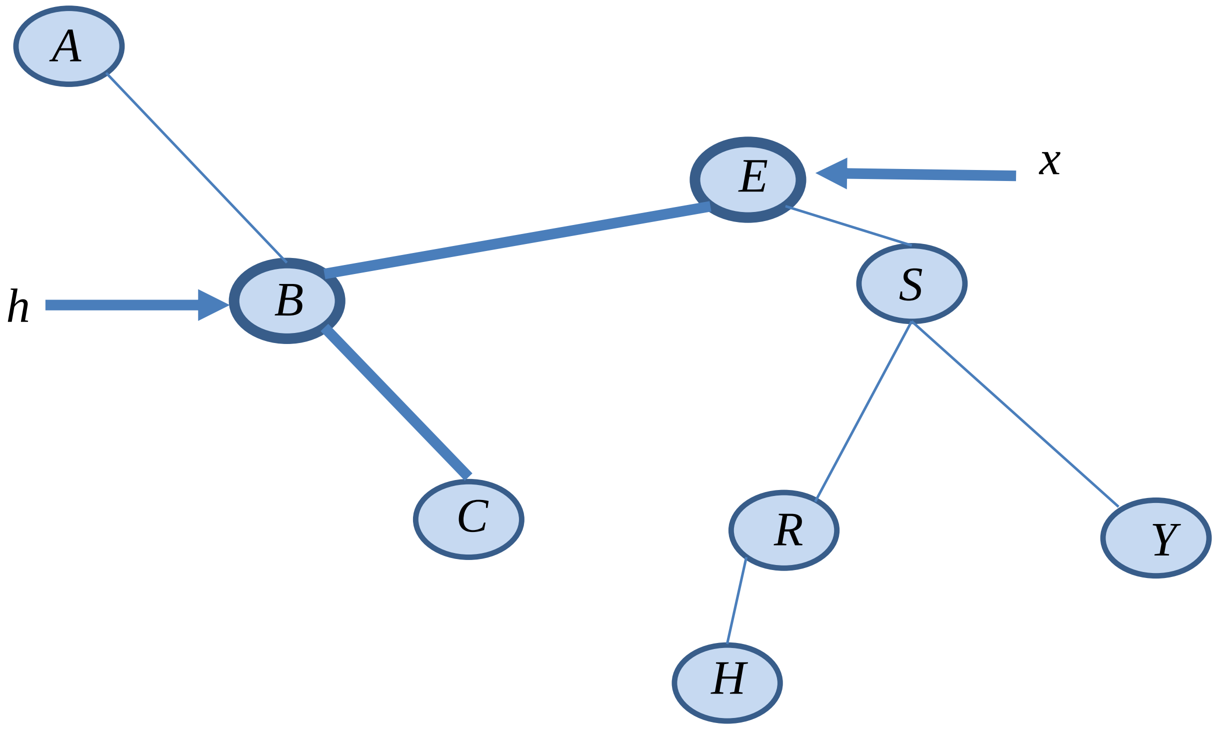
# Example: left rotation



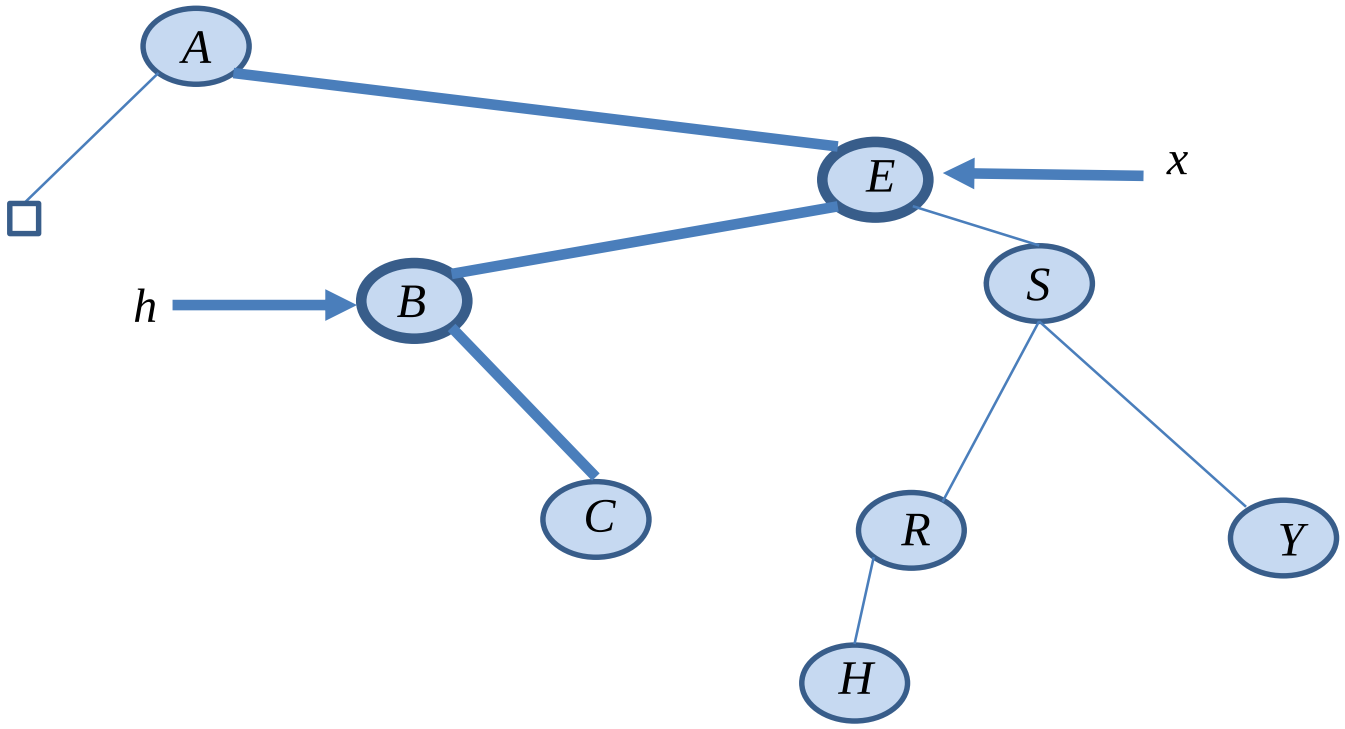
## Example: left rotation



## Example: left rotation



## Example: left rotation



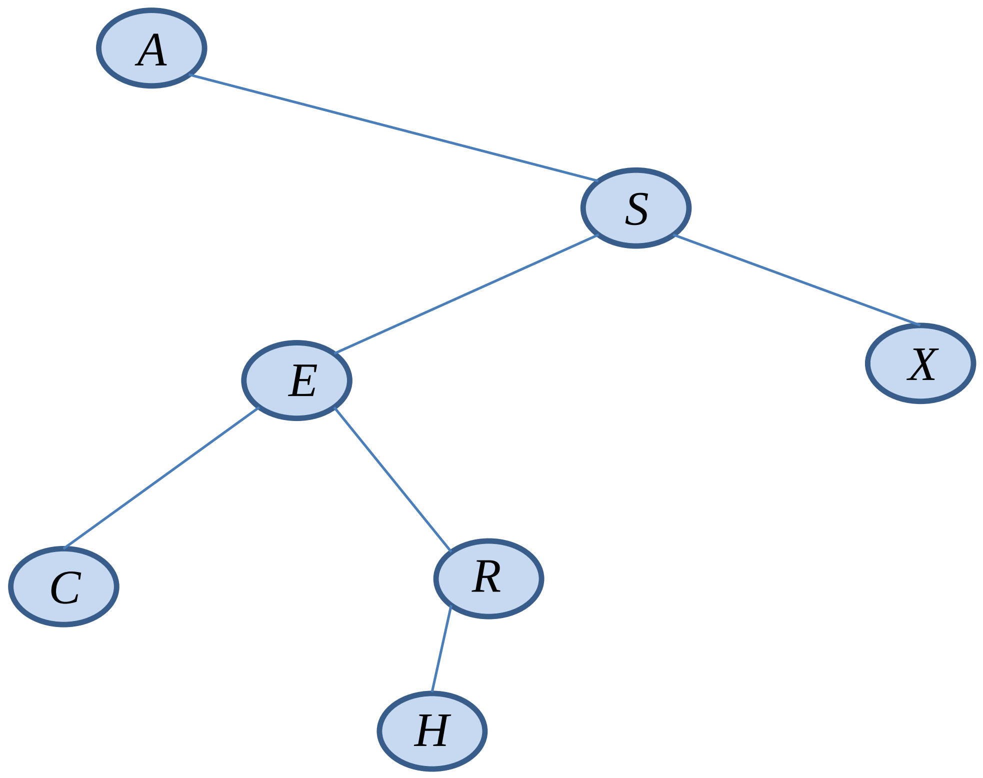
## Complexity of rotation

- Only changing a few pointers
- No traversal of the tree!
- So $O(1)$

## Root insertion

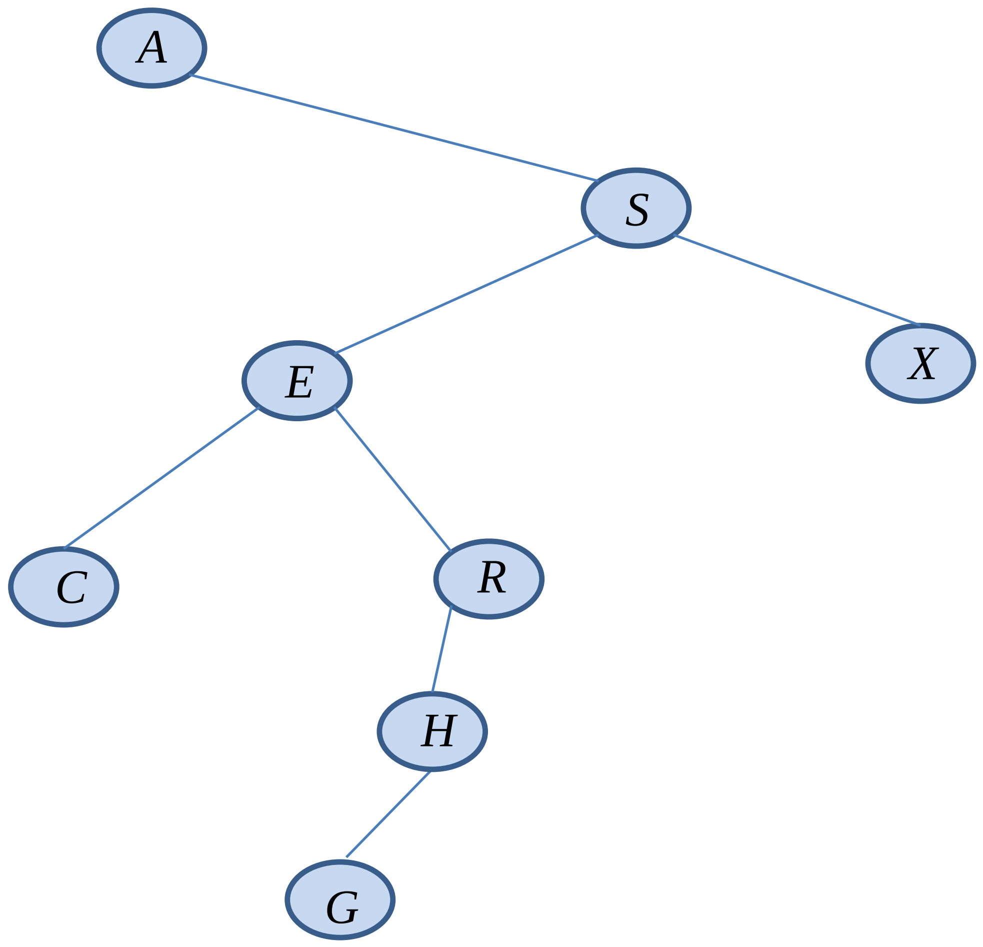
- Goal
  - insert a new element
  - place it **at the root** of the tree
- Simple **recursive** algorithm using **rotations**
  1. If empty: trivial
  2. **Recursively** insert in the **left/right subtree**
     - depending on whether the value is smaller than the root or not
     - after the recursive call finishes we have a **proper BST**
     - with the value as the **root** of the left/right **subtree**
  3. **Rotate** left or right
     - the value comes at the root!

## Example: root insertion



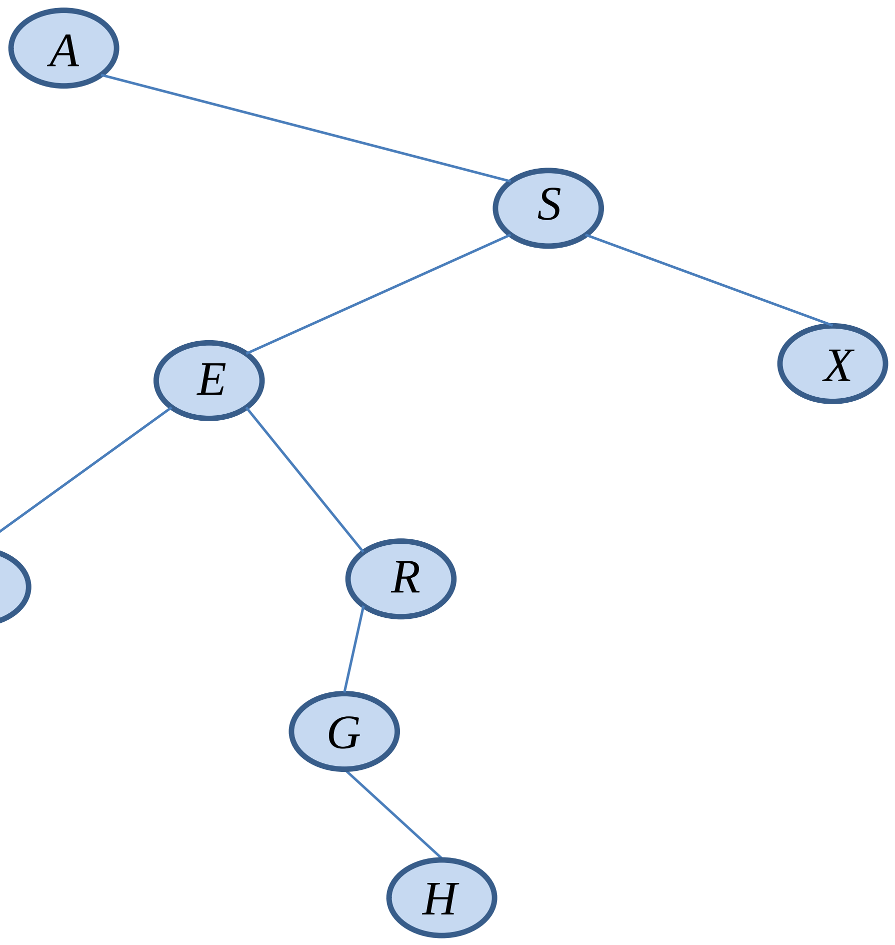
We are inserting G. The recursive algorithm is first called on the root A, then it makes **recursive calls** on the right subtree S, then on E, R, H, and finally a recursive call is made on the empty left subtree of H.

## Example: root insertion



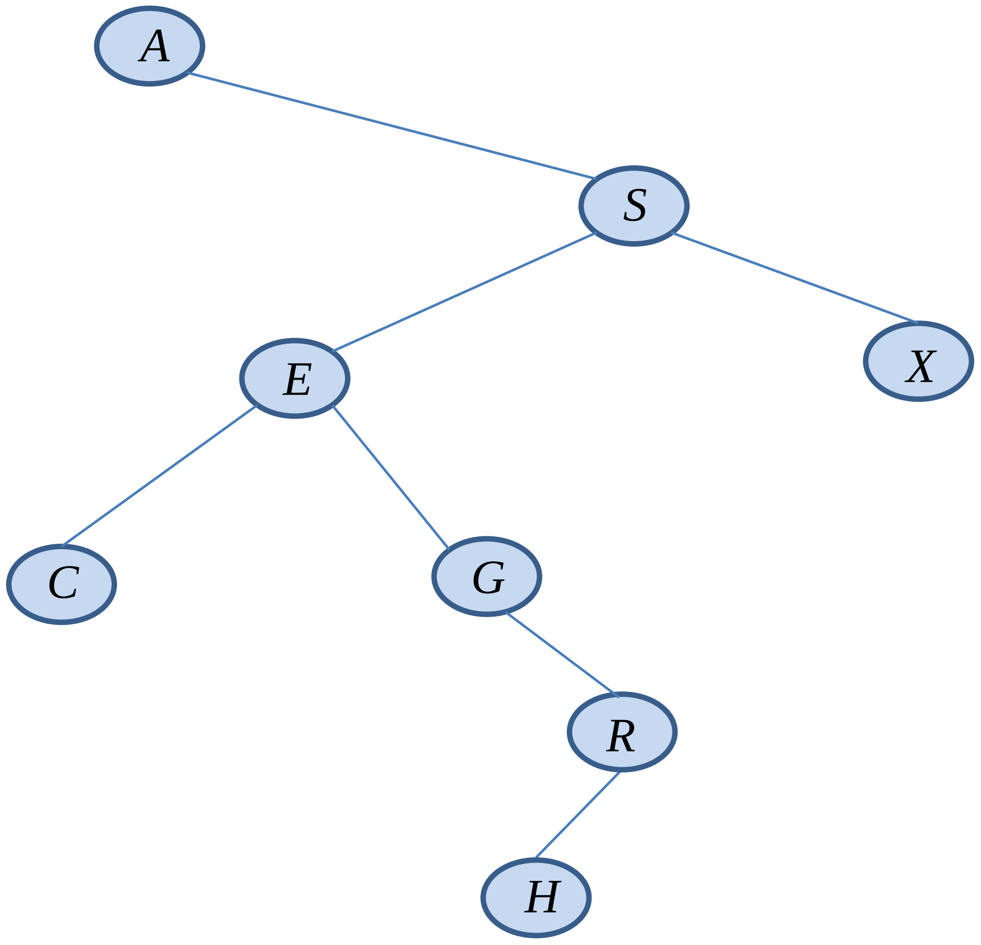
G is inserted in the empty left subtree of H.

## Example: root insertion



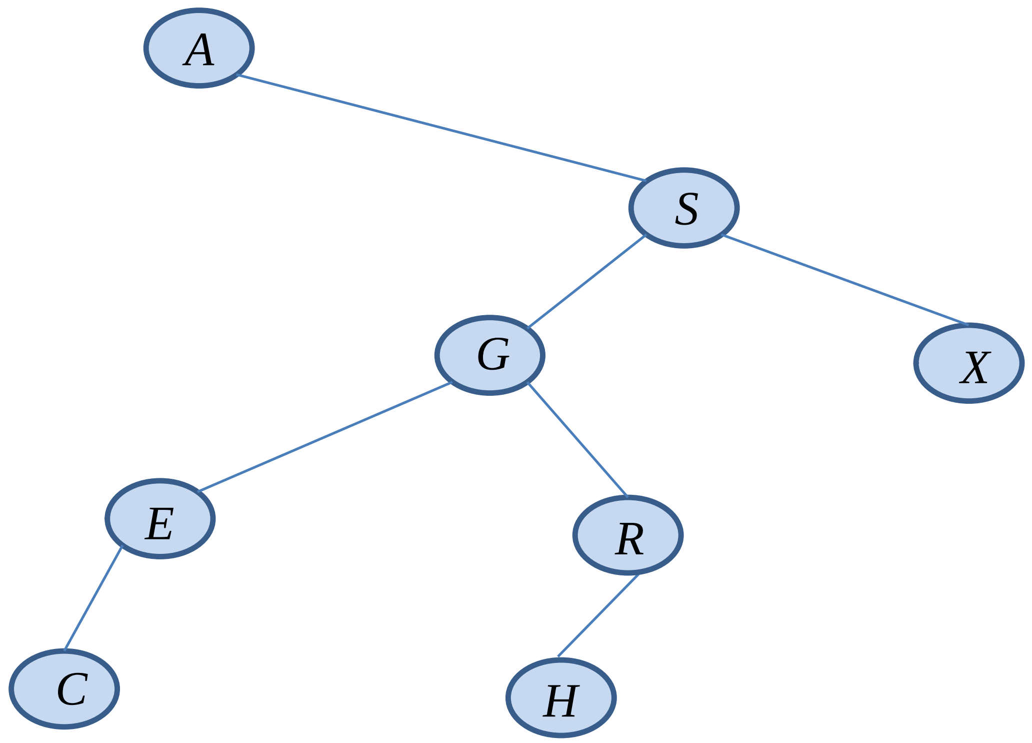
The call on H does a right rotation, G moves up.

## Example: root insertion



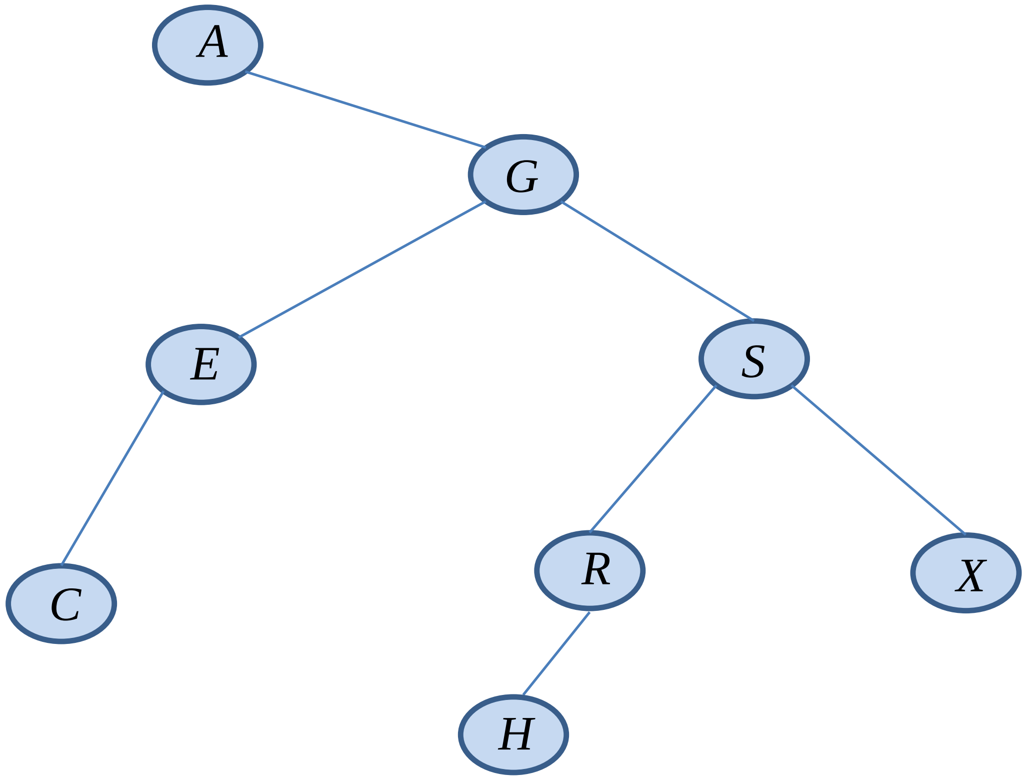
The call on R does a right rotation, G moves up.

## Example: root insertion



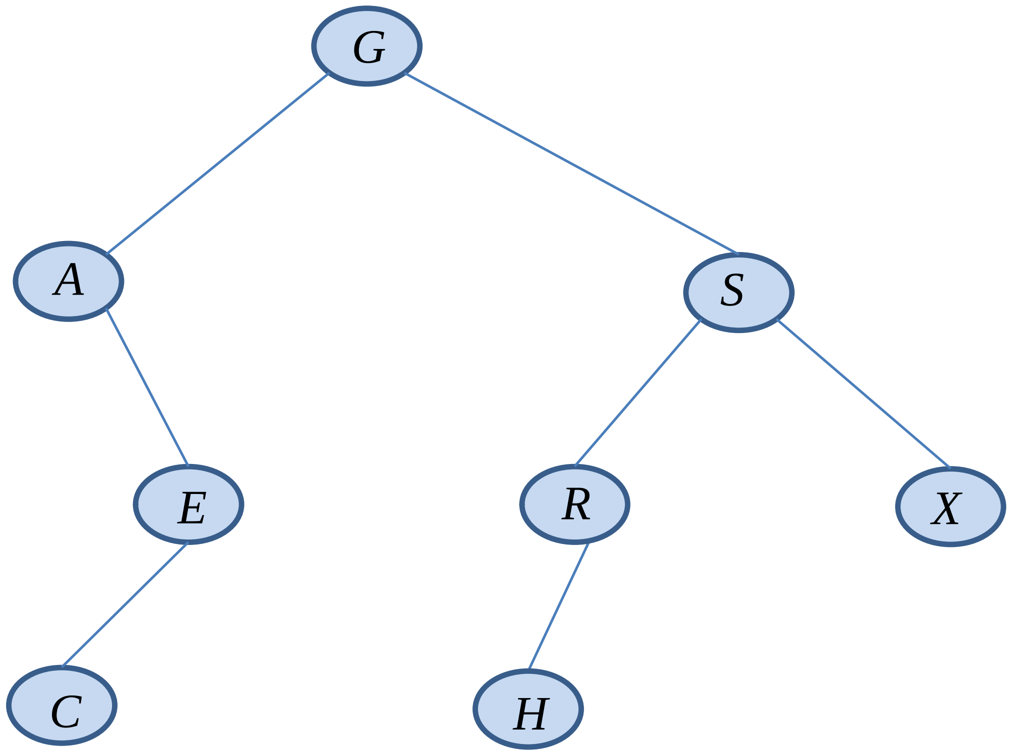
The call on E does a left rotation, G moves up.

## Example: root insertion



The call on R does a right rotation, G moves up.

## Example: root insertion



The call on A does a left rotation, G arrives at the root.

## Complexity of root insertion

- The algorithm is similar to a normal insert
  - traversing the tree towards the leaves: $O(h)$
- With an **extra rotation** at every step
  - which is $O(1)$
- So still $O(h)$

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C.*
  - Chapter 5. Sections 5.6 and 6.5.
  - Chapter 9. Section 9.7.
- R. Sedgewick. Αλγόριθμοι σε C.
  - Κεφ. 12.
- M.T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++*. 2nd edition.
  - Section 9.3 and 10.1

# AVL Trees

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Balanced trees

- We saw that most of the algorithms in BSTs are $O(h)$
    - But $h = O(n)$ in the worst-case
- So it makes sense to keep trees **"balanced"**
    - Many different ways to define what "balanced" means
    - In all of them: $h = O(\log n)$
- Eg. **complete** are one type of balanced tree (see Heaps)
    - But it's hard to maintain both BST and complete properties together
- **AVL**: a different type of balanced trees

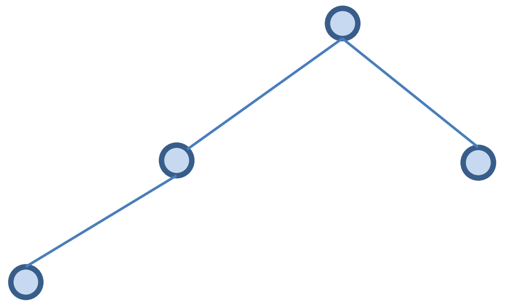
# AVL Trees

- An AVL tree is a BST with an extra property:

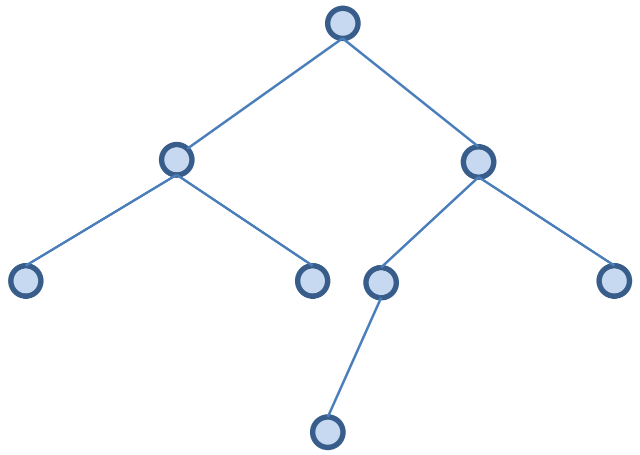
  For **all nodes**: $|\text{height}(\text{left-subtree}) - \text{height}(\text{right-subtree})| \leq 1$

- In other words, no subtree can be much shorter/taller than the other
- Recall: **height** is the longest path from the root to some leaf
  - tree with only a root: height 0
  - empty tree: height -1
- Named after Russian mathematicians Adelson-Velskii and Landis

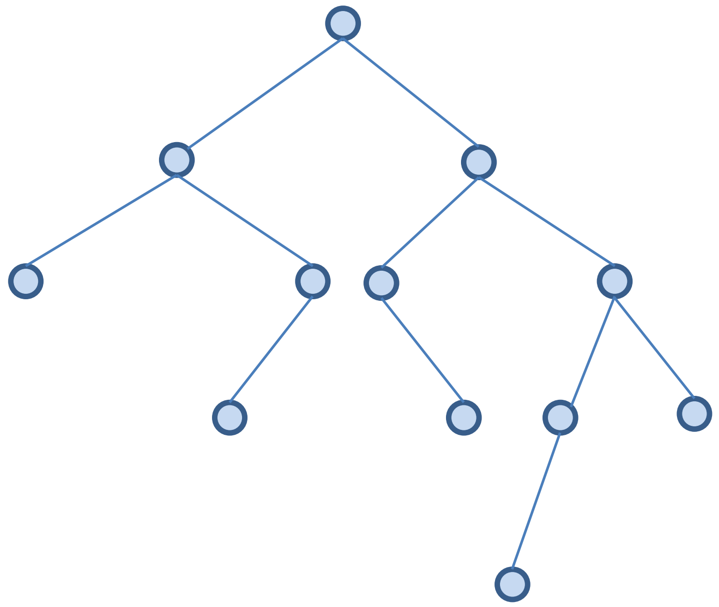
# Example – AVL tree



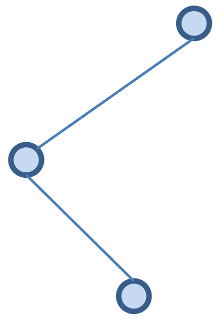
# Example – AVL tree



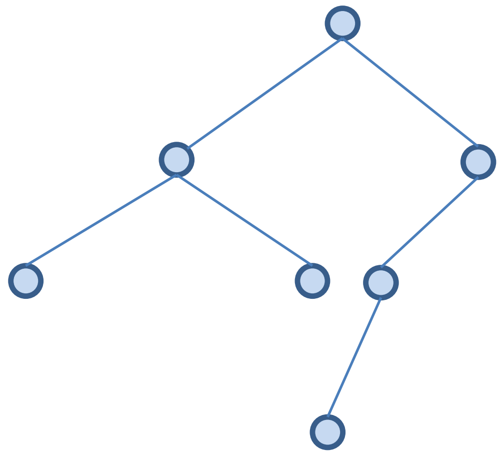
# Example – AVL tree



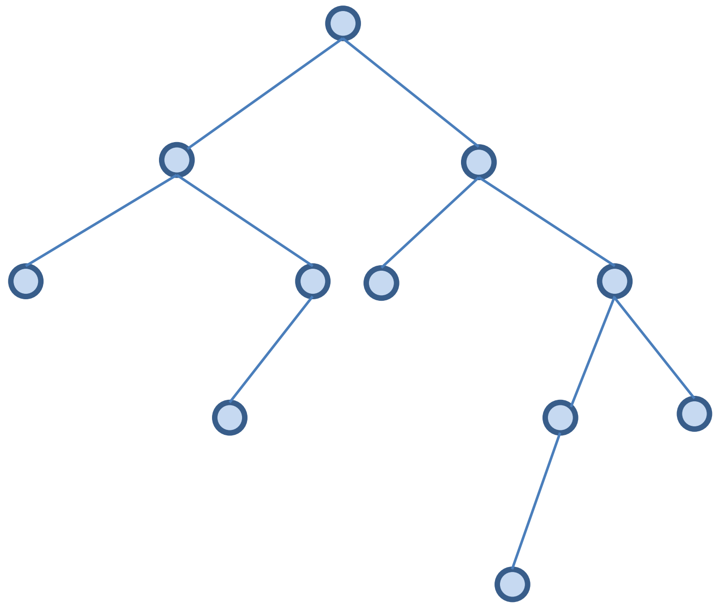
## Example – Non-AVL tree



## Example – Non-AVL tree



## Example – Non AVL tree



## The desired property

- In an AVL tree: $h = O(\log n)$
    - Proving this is not hard
- $n(h)$: **minimum number of nodes** of an AVL tree with height $h$
- We show that $h \leq 2 \log n(h)$
    - by **induction on** $h$
    - induction works very well on recursive structures!
- The base cases hold trivially (why?)
    - $n(0) = 1$
    - $n(1) = 2$

## The desired property

- Inductive step
  - Assume $\frac{h}{2} \leq \log n(h)$ for all $h < k$
  - Show that it holds for an AVL tree of height $h = k$
- **Both subtrees** of the root have height at least $h - 2$
  - because of the AVL property!
  - So $n(k) \geq 2n(k-2) \qquad\qquad (1)$
- Induction hypothesis for $h = k - 2$
  - $\frac{k-2}{2} \leq \log n(k-2)$
- From $(1)$ we take $\log$ on both sides and apply the ind. hypothesis
  - $\log n(k) \geq 1 + \log n(k-2) \geq 1 + \frac{k-2}{2} = \frac{k}{2}$

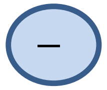
# Balance factor

A node can have one of the following “balance factors”

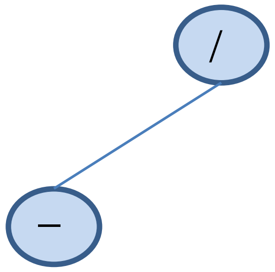
| Balance factor | Meaning |
|---|---|
| `-` | Sub-trees have equal heights |
| `/` | Left sub-tree is $1$ higher |
| `//` | Left sub-tree is $> 1$ higher |
| `\` | Right sub-tree is $1$ higher |
| `\\` | Right sub-tree is $> 1$ higher |

Nodes `-`, `/`, `\` are AVL.
Nodes `//`, `\\` are not AVL.

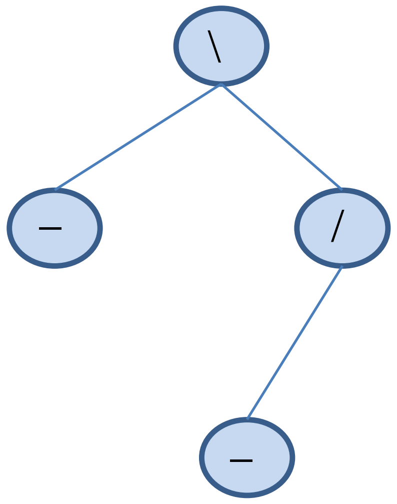
## Example AVL Tree



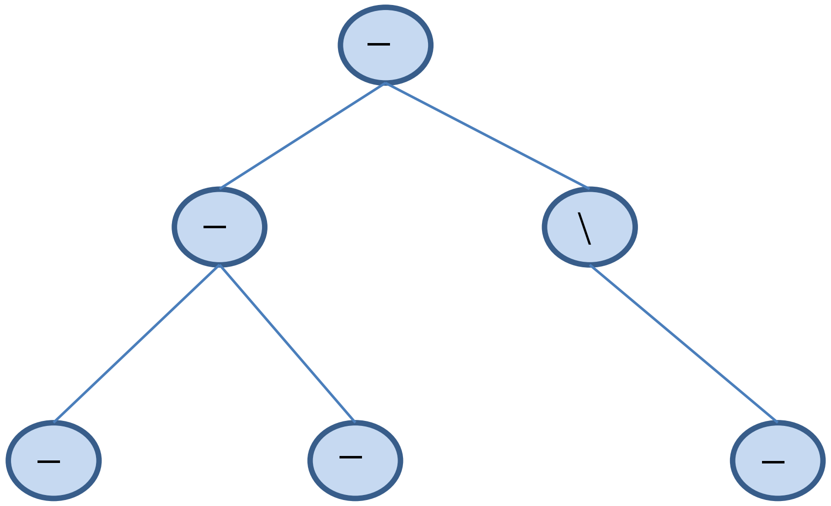
## Example AVL Tree



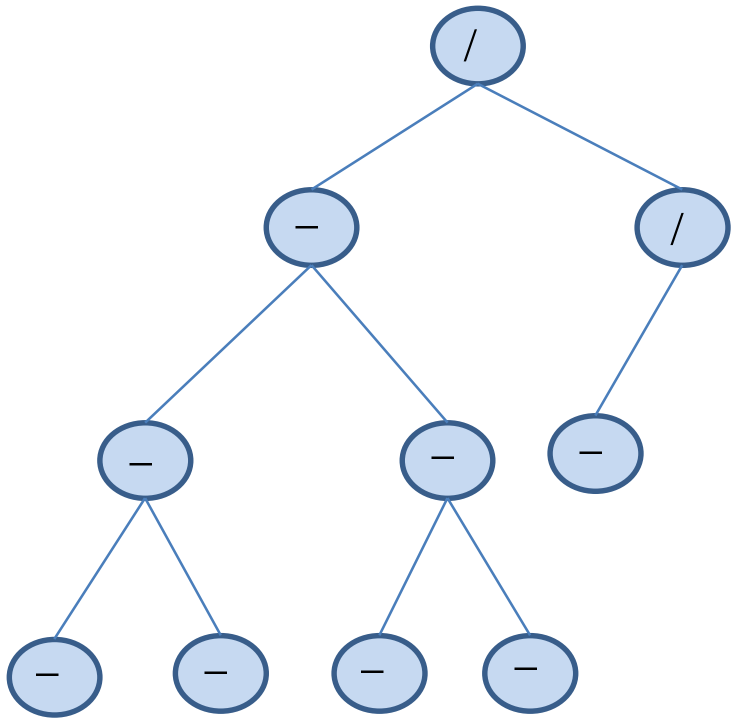
# Example AVL Tree



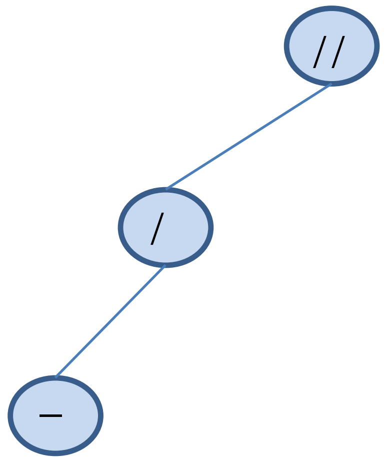
# Example AVL Tree



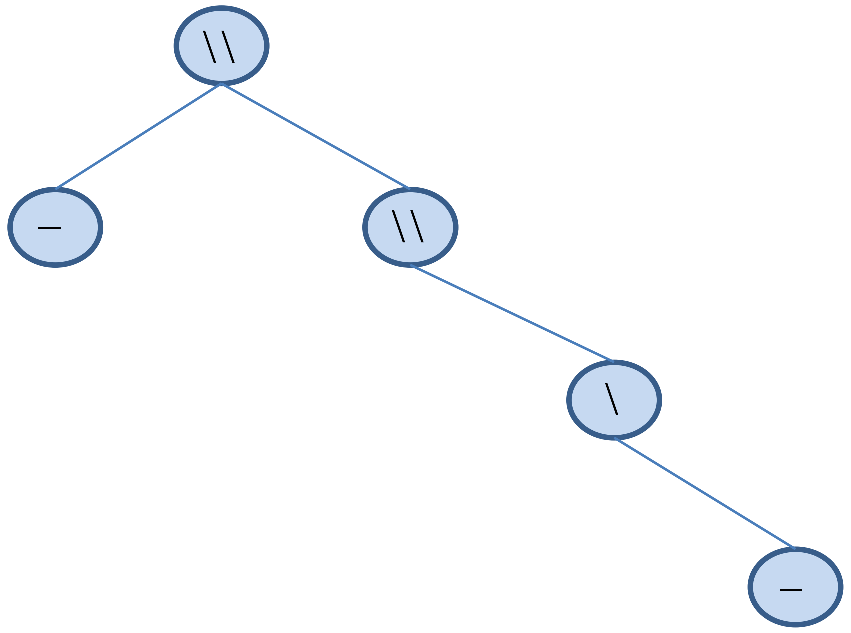
## Example AVL Tree



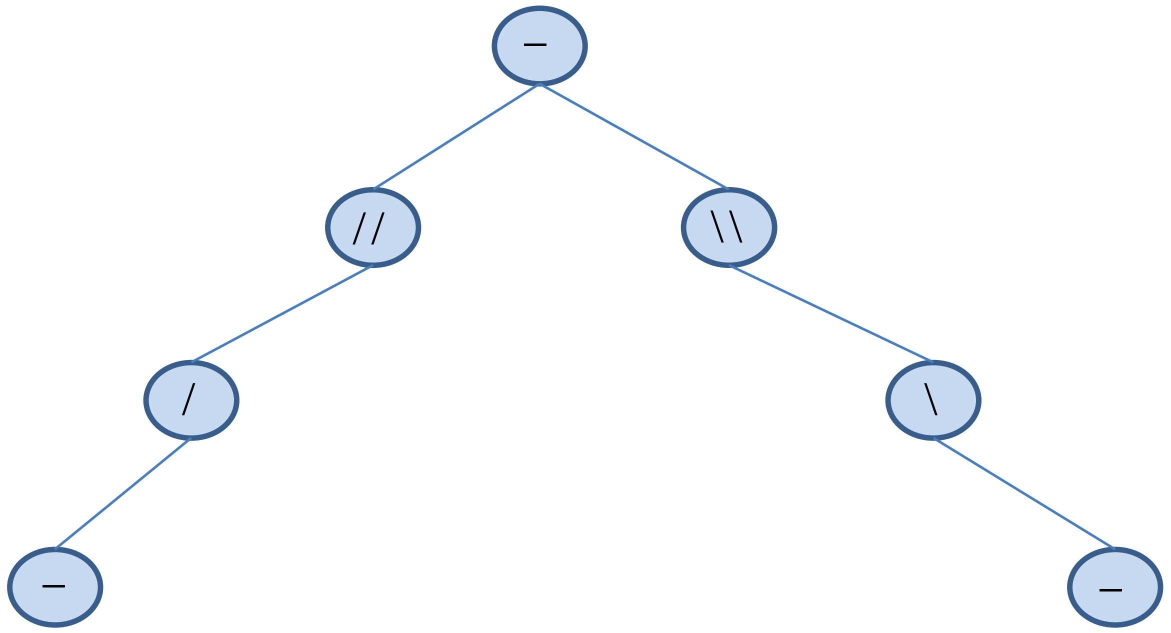
## Example non-AVL Tree



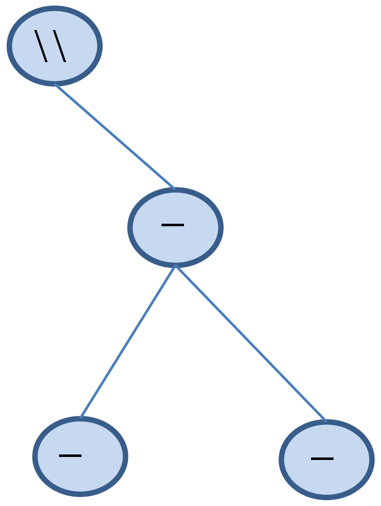
## Example non-AVL Tree



## Example non-AVL Tree



## Example non-AVL Tree



## Operations in an AVL Tree

- Same as those of a BST
- Except that we need to **restore** the AVL property
  - after **inserting** a node
  - or **deleting** a node
- We do this using **rotations**

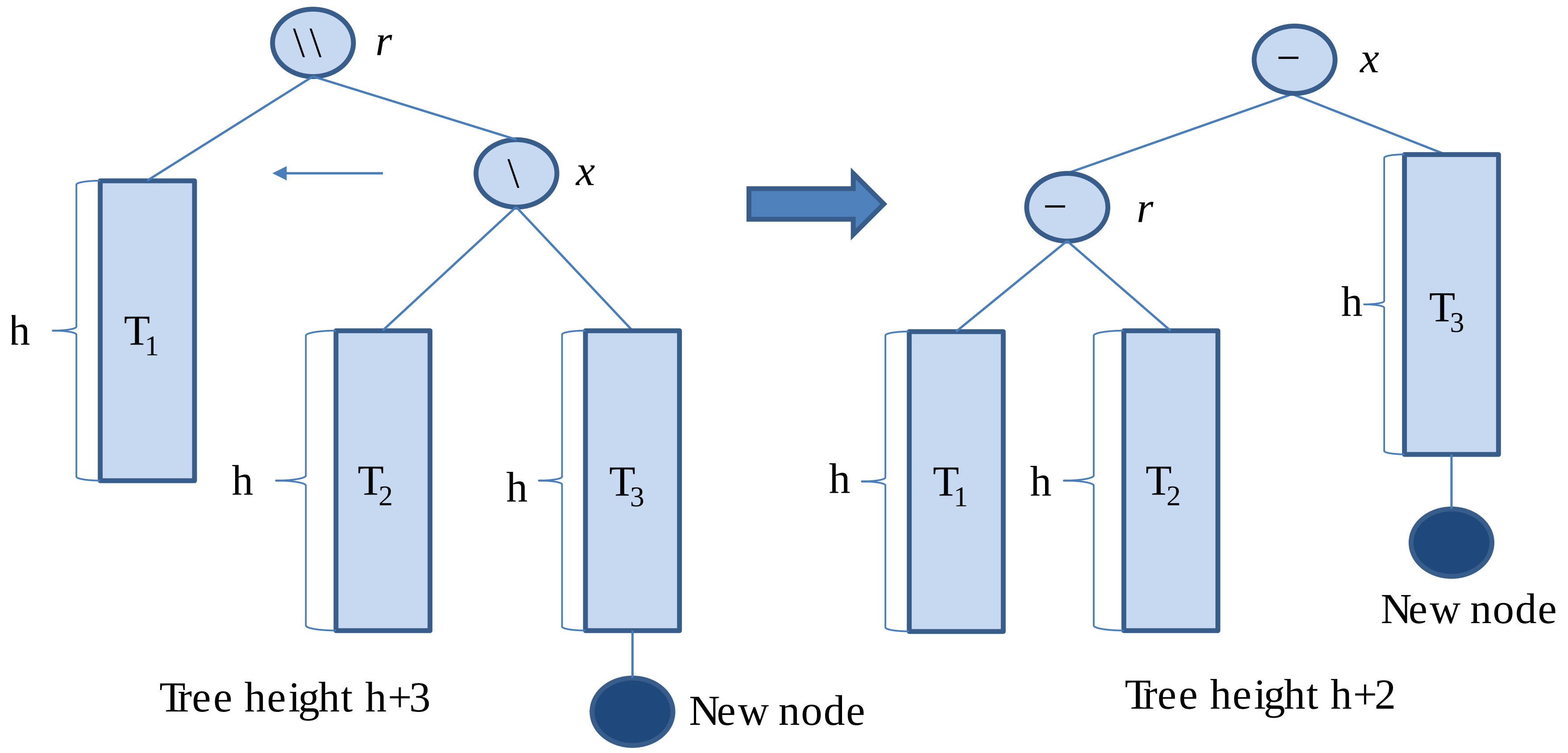
## Recursive AVL restore

- Restoring the AVL property is a **recursive** operation
- It happens during an insert or delete
  - Which are both recursive
  - When their recursive calls are **unwinding** towards the root
- So when we restore a node $r$, its **children** are already restored **AVL trees**

## AVL restore after insert

- Assume $r$ became `\\` after an insert (the case `//` is symmetric)
- Let $x$ be the **root** of the **right subtree**
  - The new value was inserted under $x$ (since $r$ is `\\`)
- What can be the **balance factor** of $x$?
  - `\\` and `//` are not possible since the child $x$ is **already restored**
- Case 1: $x$ is `\`
  - A **left-rotation** on $r$ restores the property!
  - Both $r$ and $x$ become `-` (easily seen in a drawing)

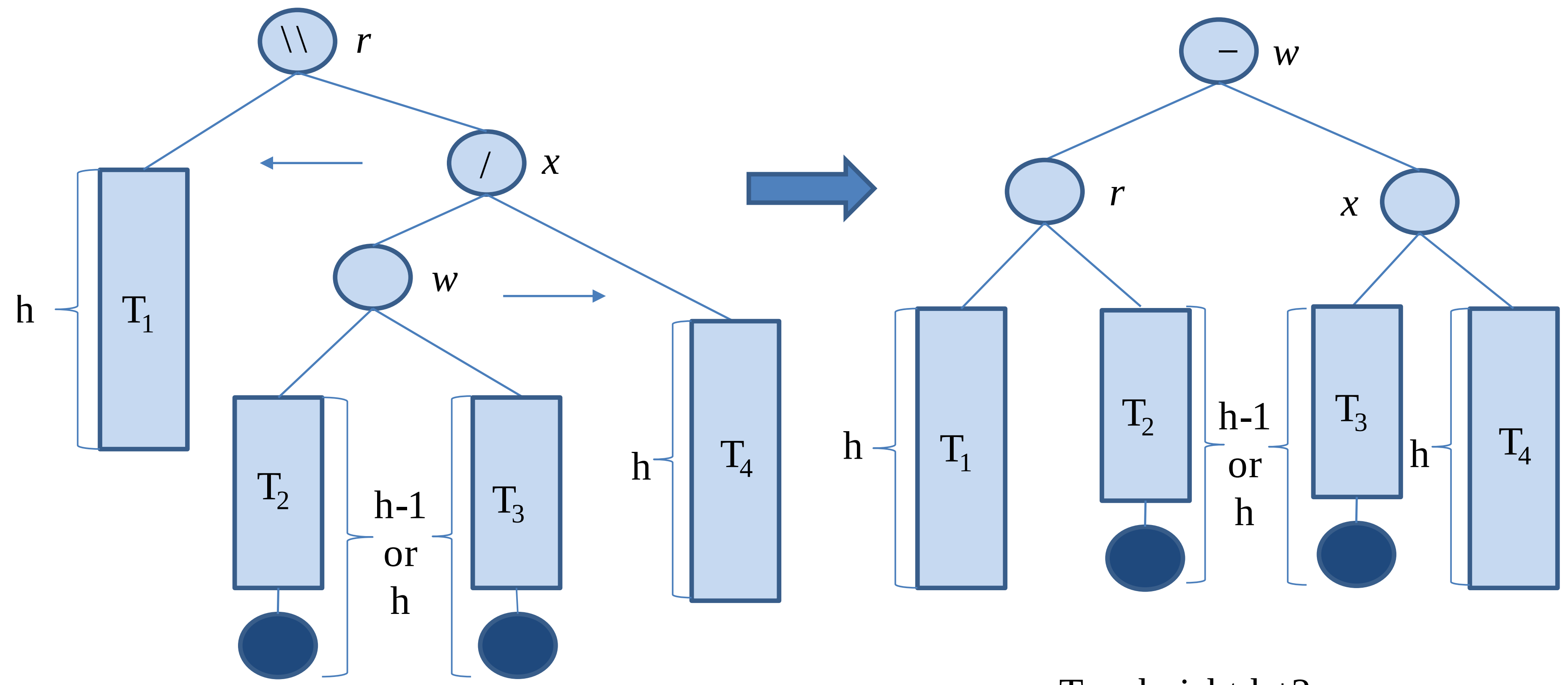
## Insert: single left rotation at r



## AVL restore after insert

- Case 2: $x$ is `/`
  - This is more tricky
  - A left-rotation on $r$ (as before) might cause $x$ to become `//`
- We need to do a **double** right-left rotation
  - First **right-rotation** on $x$
  - Then **left-rotation** on $r$
- The left-child $w$ of $x$ becomes the new root
  - $w$ becomes `-`
  - $r$ becomes `-` or `/`
  - $x$ becomes `-` or `\`

# Insert: double right-left rotation at x and r



One of $T_2$ or $T_3$ has the new node and height h
Tree height h+3

Tree height h+2

## AVL restore after insert

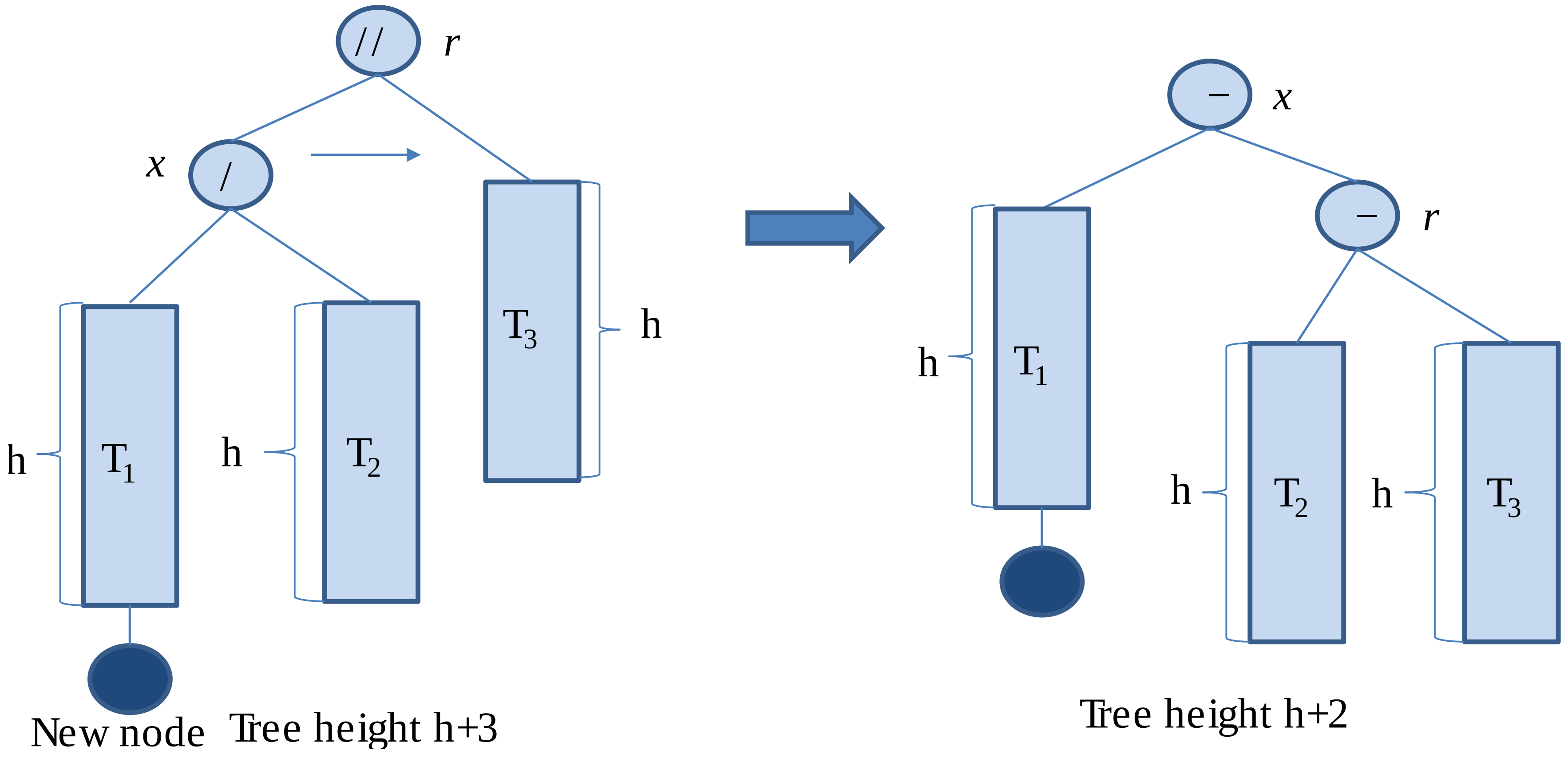
- Case 3: $x$ is `-`

- This in fact **cannot happen**!
  - Assume both subtrees of $x$ have height $h$
  - Then the left subtree of $r$ also must have height ($h$)
  - Otherwise AVL would be violated **before** the insert (see the drawings)

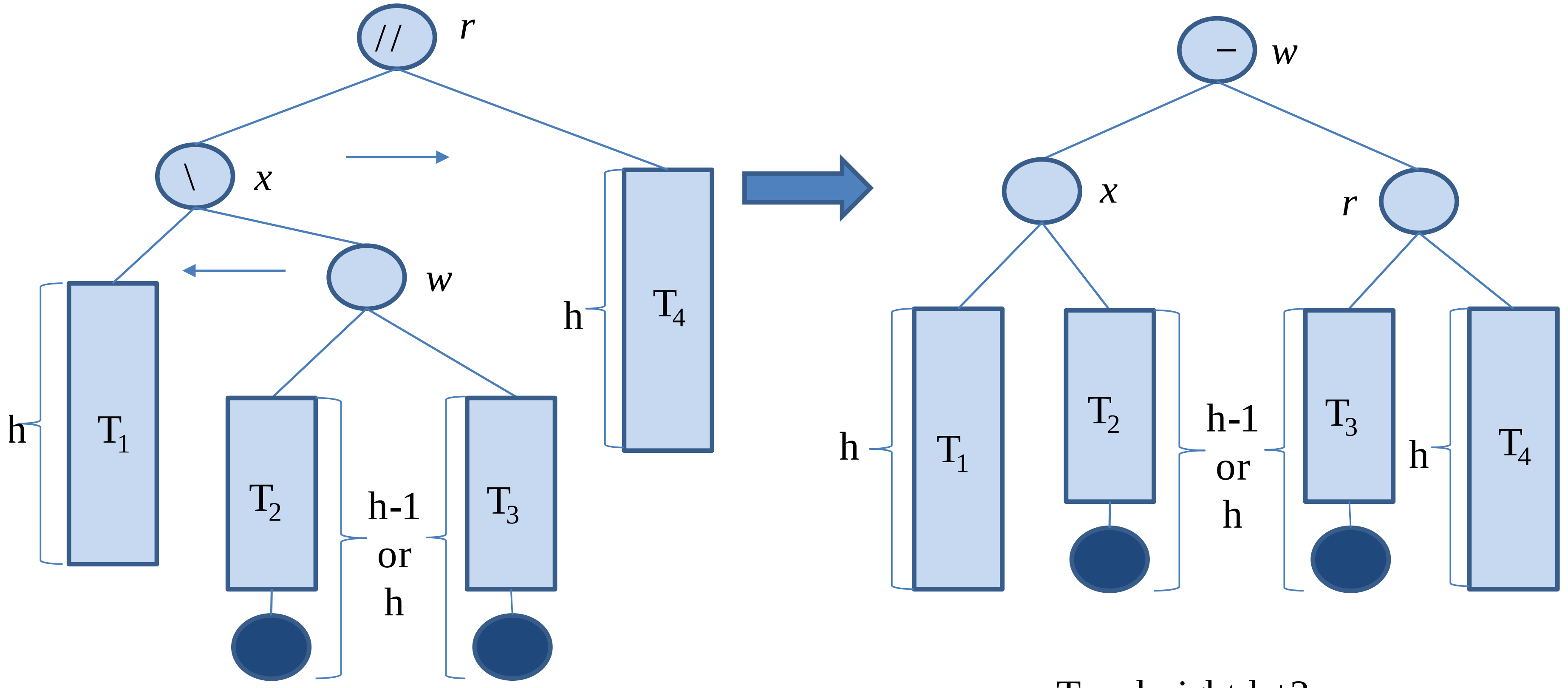
# Symmetric case

- The case when $x$ becomes `//` is **symmetric**
- We need to consider the BF of its **left-child** $x$
  - $x$ is `/` : we do a **single right** rotation at $r$
  - $x$ is `\` : we do a **double left-right** rotation at $x$ and $r$
  - $x$ is `-` : **impossible**

## Insert: single right rotation at r



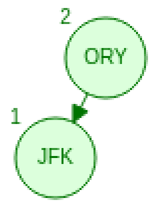
## Insert: double left-right rotation at x and r



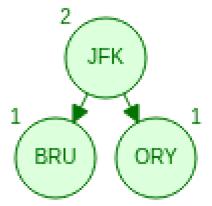
One of $T_2$ or $T_3$ has the new node and height h
Tree height h+3

Tree height h+2

## Insert example

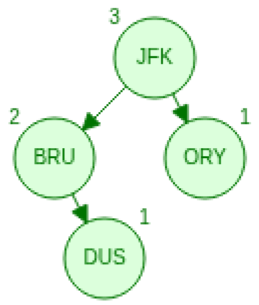


## Insert example



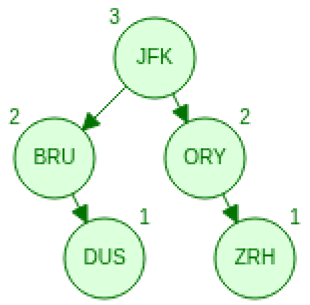
Inserting BRU, causes single right-rotate at ORY

# Insert example



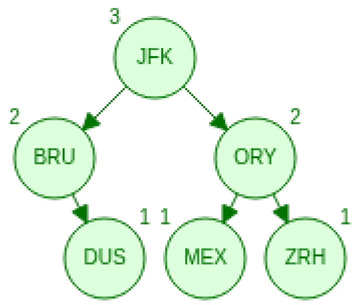
Inserting DUS

## Insert example



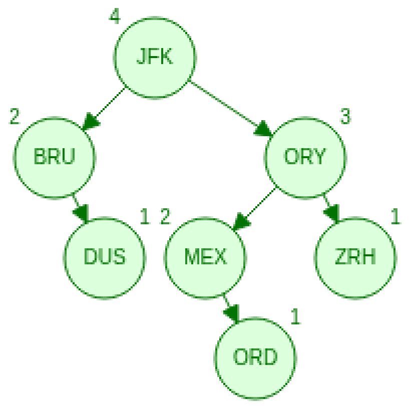
Inserting ZRH

## Insert example



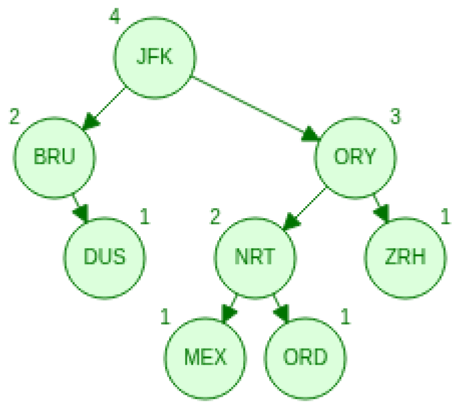
Inserting MEX

## Insert example



Inserting ORD

## Insert example

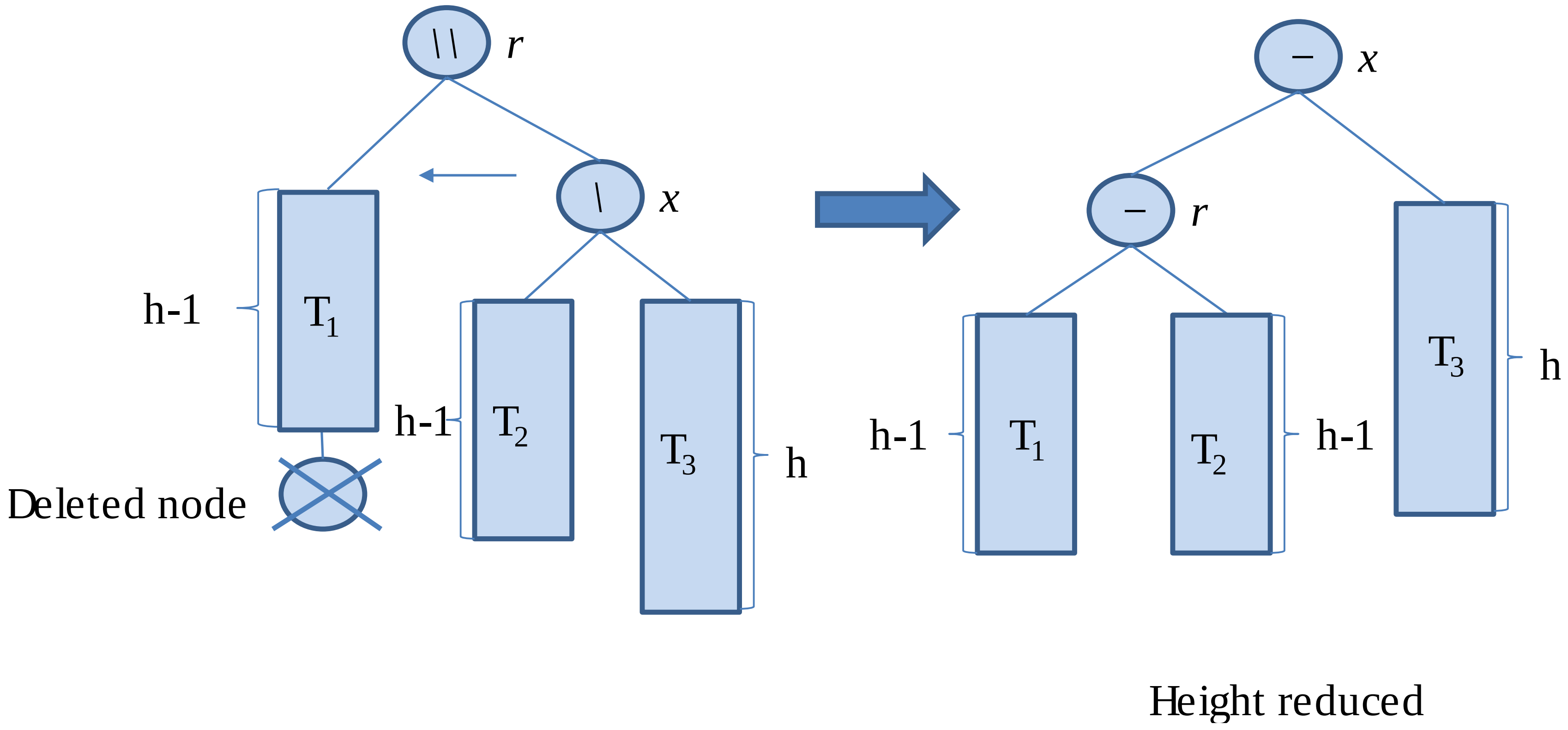


Inserting NRT, causes double right-left rotation at ORD and MEX

## AVL restore after delete

- Assume $r$ became `\\` after delete (the case `//` is symmetric)
- Let $x$ be the **root** of the **right-subtree**
  - The value was deleted from the left sub-tree (since $r$ is `\\`)
- What can be the **balance factor** of $x$?
  - `\\` and `//` are not possible since the child $x$ is **already restored**
- Case 1: $x$ is `\`
  - A **left-rotation** on $r$ restores the property!
  - Both $r$ and $x$ become `-` (easily seen in a drawing)

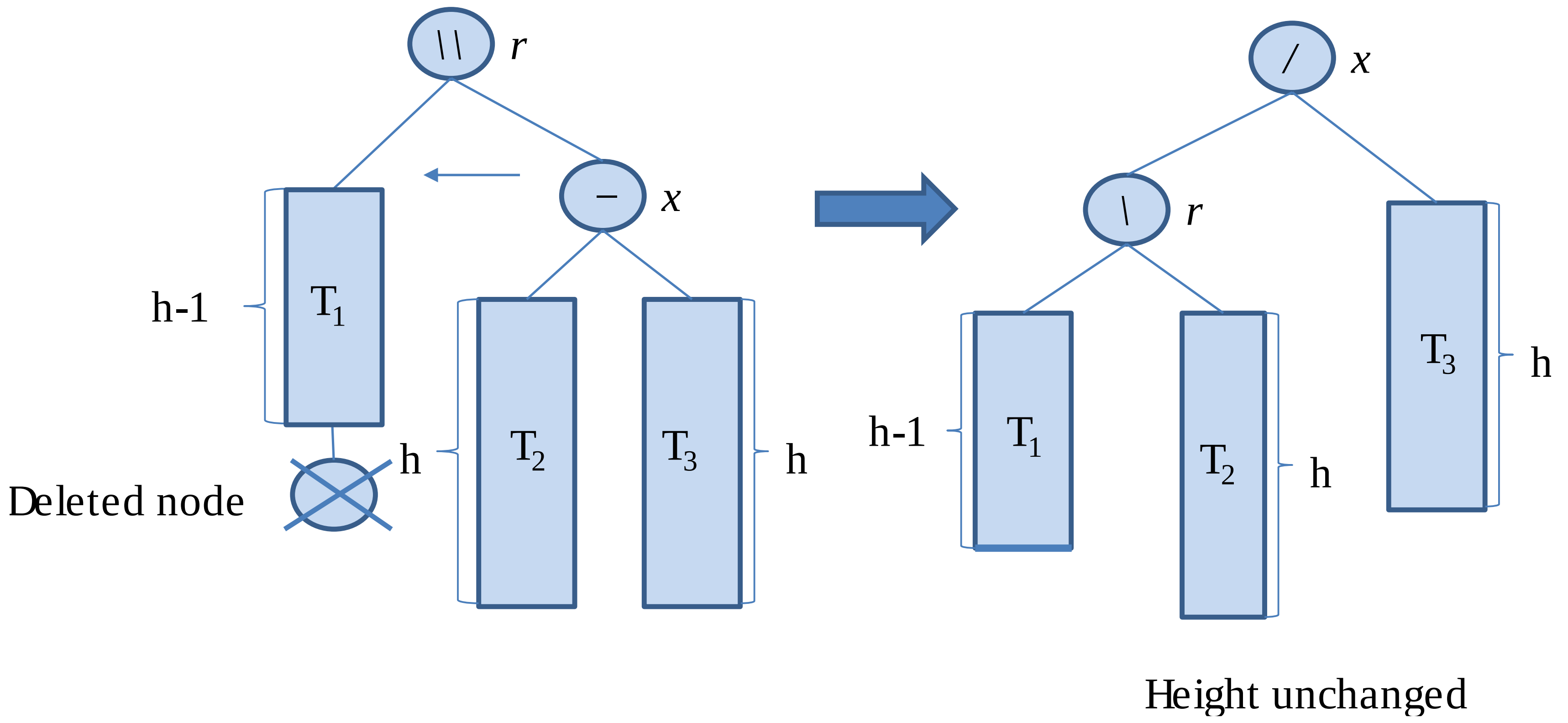
# Delete: single left-rotation at r



## AVL restore after delete

- Case 2: $x$ is `-`
  - After a **delete** this is possible!
  - A **left-rotation** on $r$ again restores the property
  - $r$ becomes `\`, $x$ becomes `/`

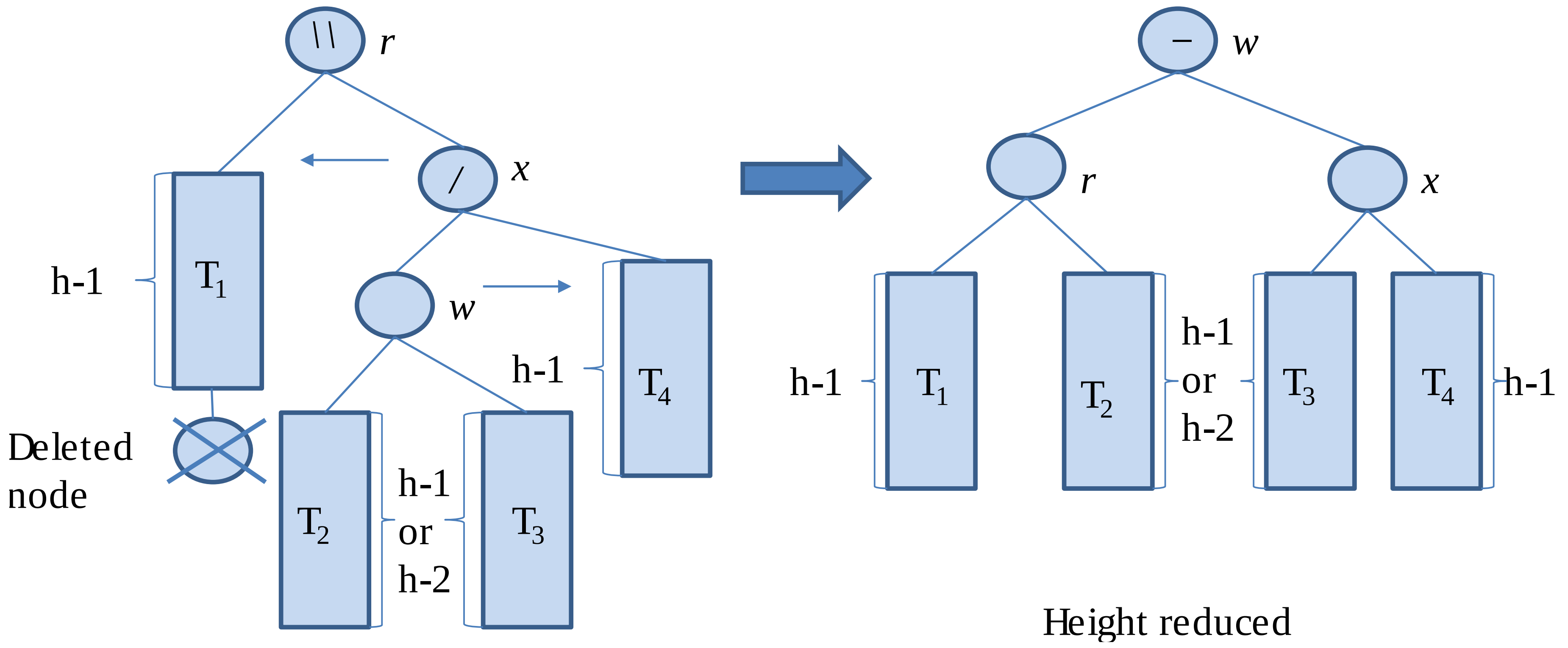
# Delete: single left-rotation at r



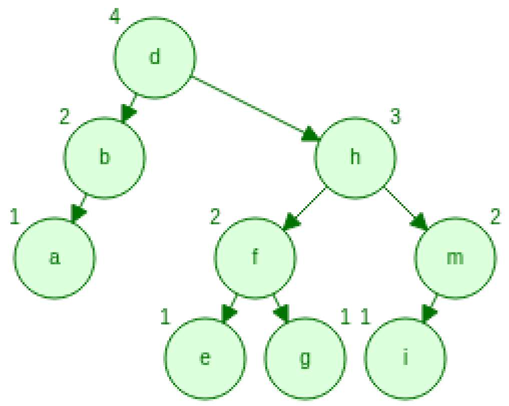
## AVL restore after delete

- Case 3: $x$ is `/`
  - This is more tricky
  - A left-rotation on $r$ (as before) might cause $x$ to become `//`
- We need to do a **double** right-left rotation
  - First **right-rotation** on $x$
  - Then **left-rotation** on $r$
- The left-child $w$ of $x$ becomes the new root
  - $w$ becomes `-`
  - $r$ becomes `-` or `/`
  - $x$ becomes `-` or `\`

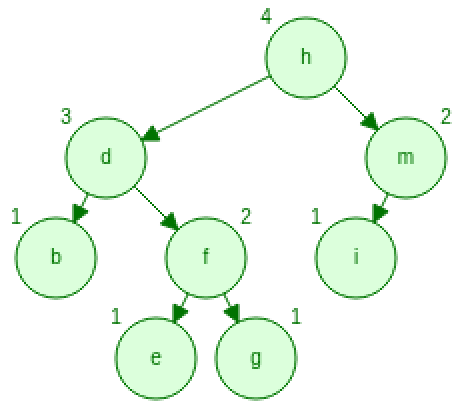
## Delete: double right-left rotation at r



## Delete example

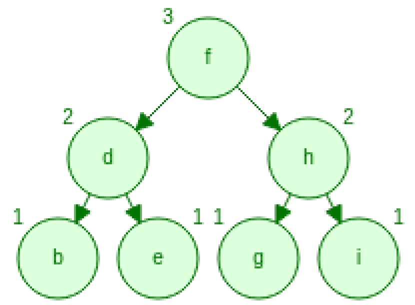


## Delete example



Deleting a, causes single left-rotate at d

## Delete example



Deleting m, causes double left-right rotation at d and h

# Complexity of operations on AVL trees

- Search on BST is $O(h)$
  - So $O(\log n)$ for AVL, since $h \leq 2\log n$
- Insert/delete on BST is $O(h)$
  - We add at most on rotation at each step, each rotation is $O(1)$
  - So also $O(\log n)$
- Interesting fact
  - During insert **at most one rotation** will be performed!
  - Because both rotations we saw **decrease** the height of the sub-tree

## Implementation details

- We need to keep the **height** of each subtree
  - to compute the balance factors
  - If we need to save memory we can store **only** the balance factors
- Restoring after both insert and delete are similar
  - We can treat them together

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*. Chapter 9. Section 9.8.
- R. Kruse, C.L. Tondo and B.Leung. *Data Structures and Program Design in C*. Chapter 9. Section 9.4.
- M.T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++*. 2nd edition. Section 10.2

# Multi-Way Search Trees

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

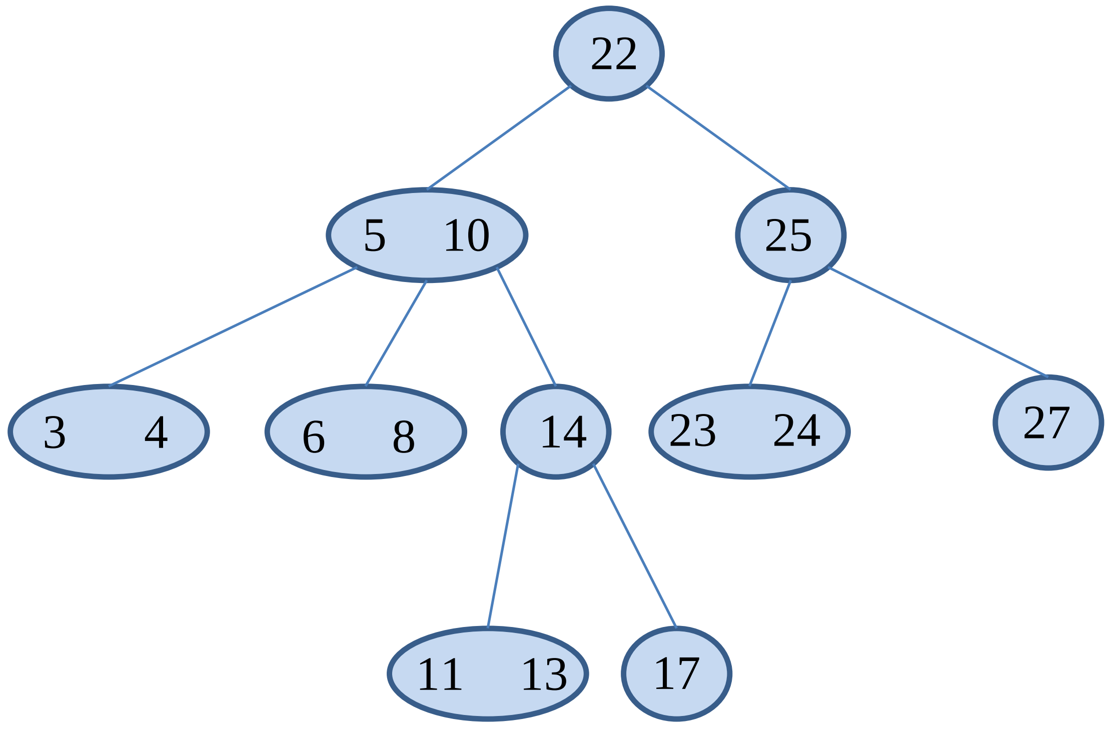
## Motivation

- We keep the **ordering** idea of BSTs
    - **Fast search**, by excluding whole subtrees
- And add **more than two children** for each node
    - Gives more flexibility in restructuring the tree
    - And news ways to **keep it balanced**

## Multi-way search trees

- $d$-node: a node with $d$ children
- Each **internal** $d$-node stores $d-1$ **ordered** values $k_1 < \ldots < k_{d-1}$
  - **No duplicate** values in the whole tree
- All values in a **subtree** lie **in-between** the corresponding node values
  - For all values $l$ in the $i$-th subtree: $k_{i-1} < l < k_i$
  - Convention: $k_0 = -\infty, k_d = +\infty$
- $m$-way search tree: all nodes have **at most** $m$ children
  - A BST is a 2-way search tree

## Example multi-way search tree

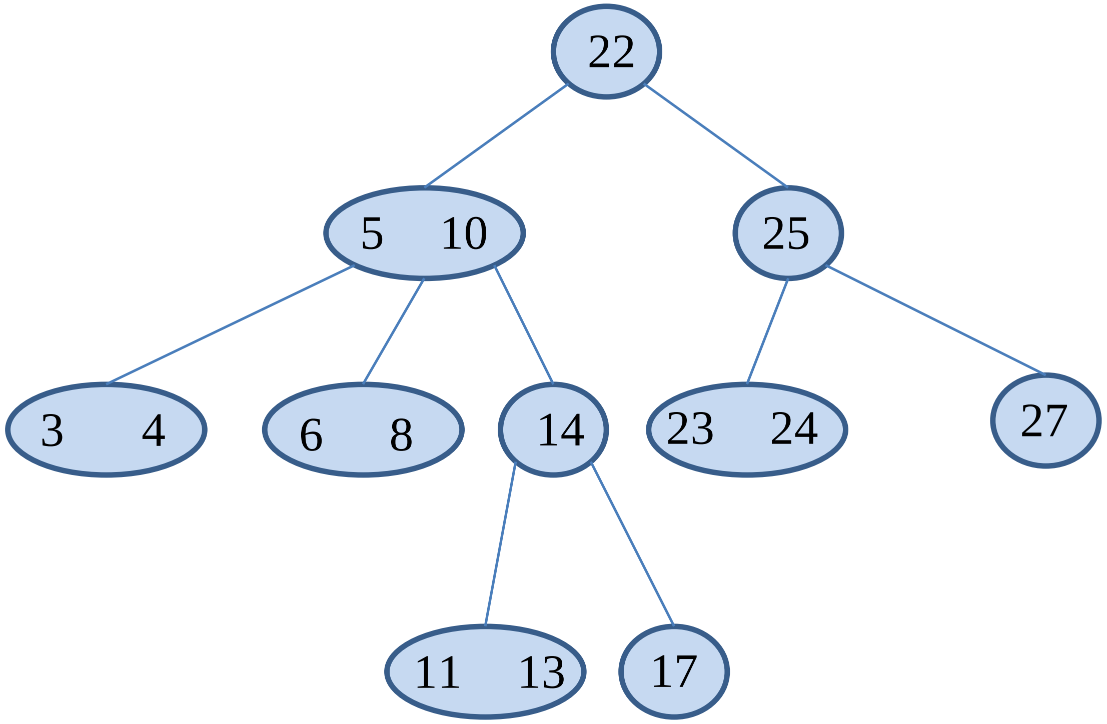


$m = 3$

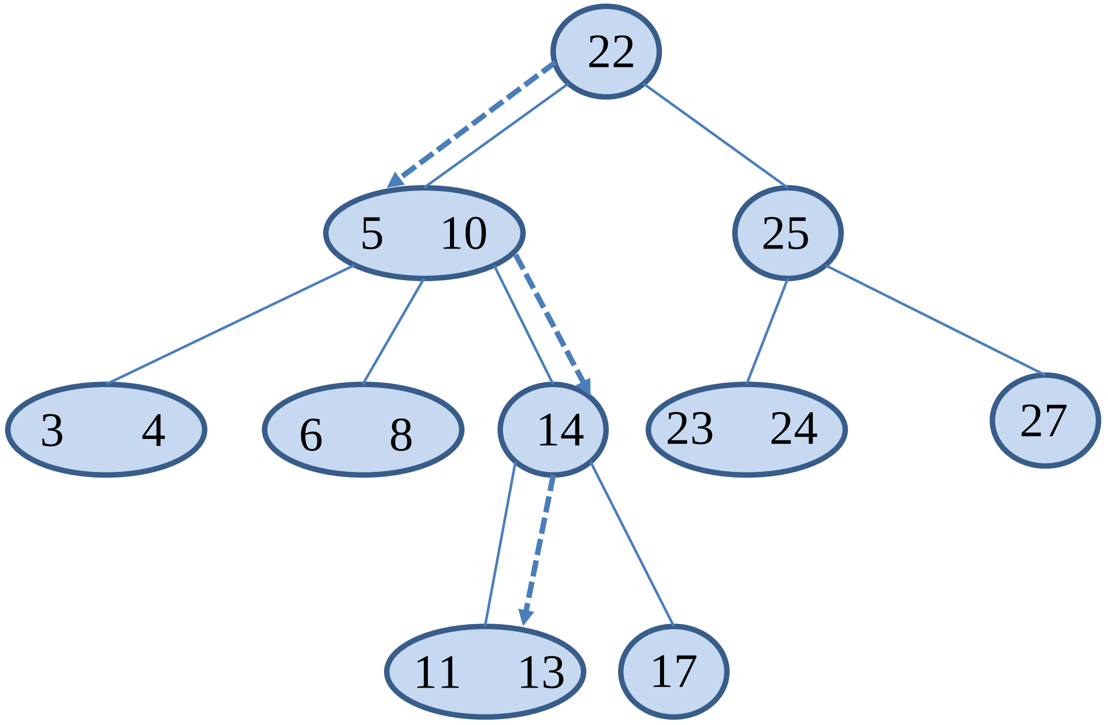
## Searching in a multi-way search tree

- Simple adaptation of the algorithm for BSTs
- Start from the root, traverse towards the leaves
- In each node, there is **a single subtree** that can possibly contain a value $l$
  - The subtree $i$ such that $k_{i-1} < l < k_i$
  - Continue in that subtree

## Example multi-way search tree

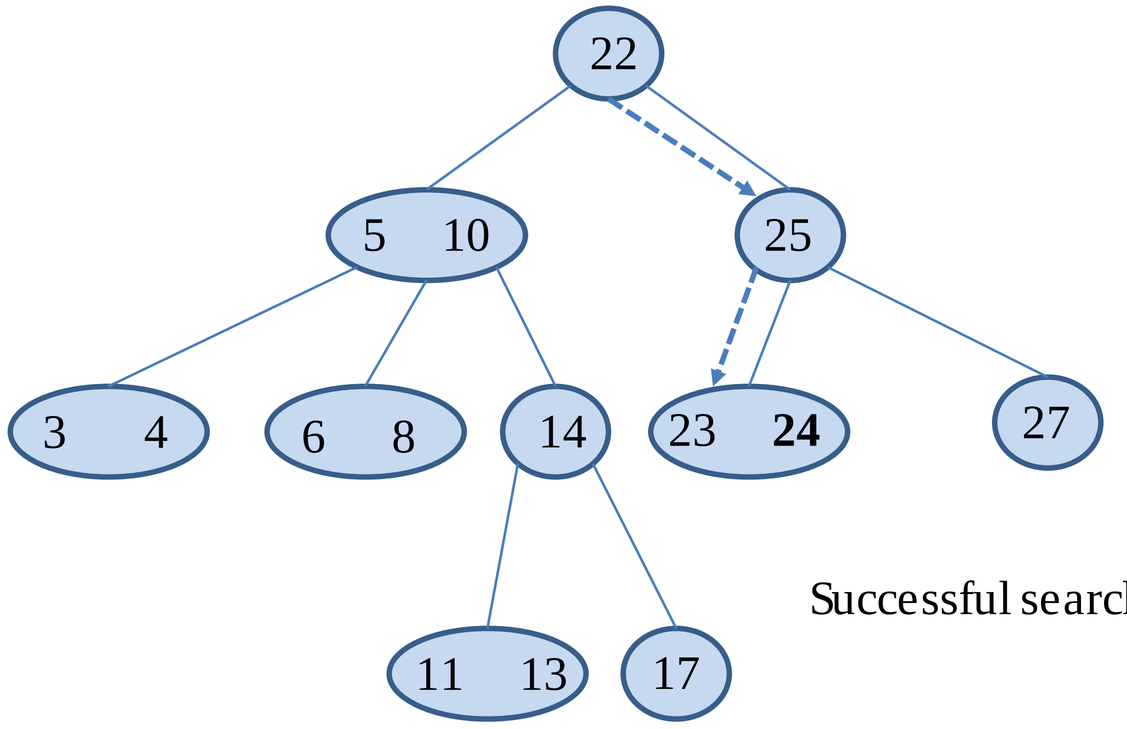


# Search for value 12



Unsuccessful search

## Search for value 24

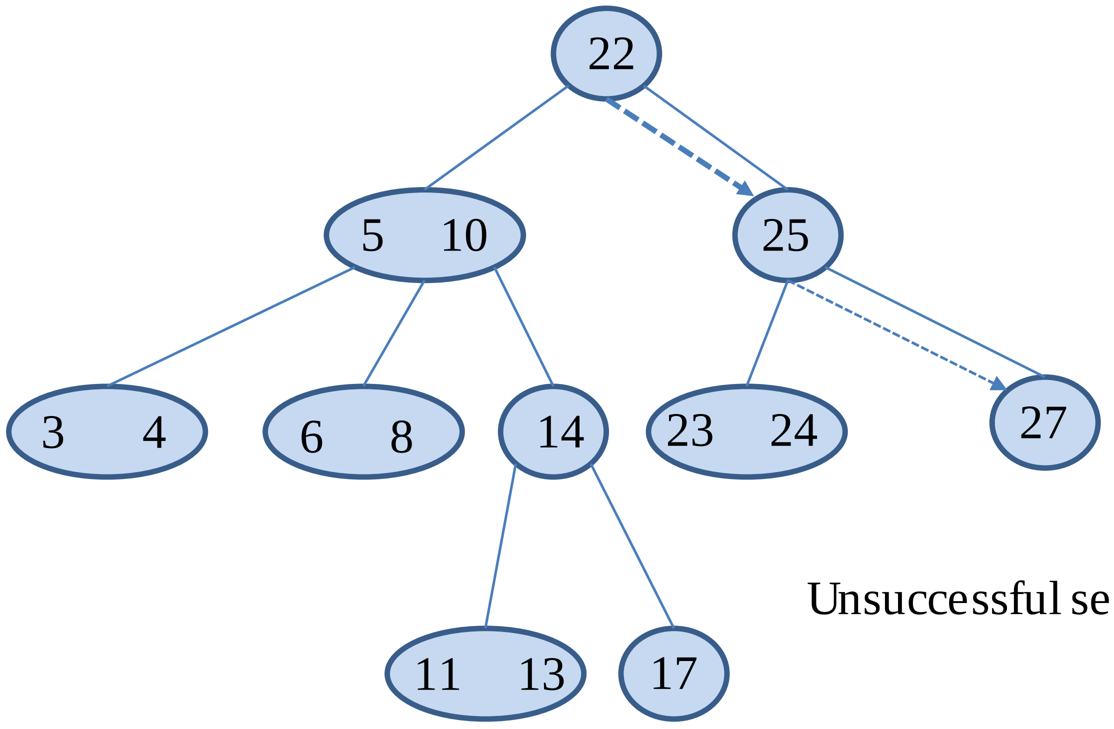


Successful search

## Insertion in a multi-way search tree

- Again, simple adaptation of BSTs
    - **But**: we don't always need to create a new node
    - We can insert in an existing one if there is space
- Start with a search for the value $l$ we want to insert
- If found, stop (no duplicates)
- If not found, insert at the **leaf** we reached
    - If full, create an $i$-th child, such that $k_{i-1} < l < k_i$

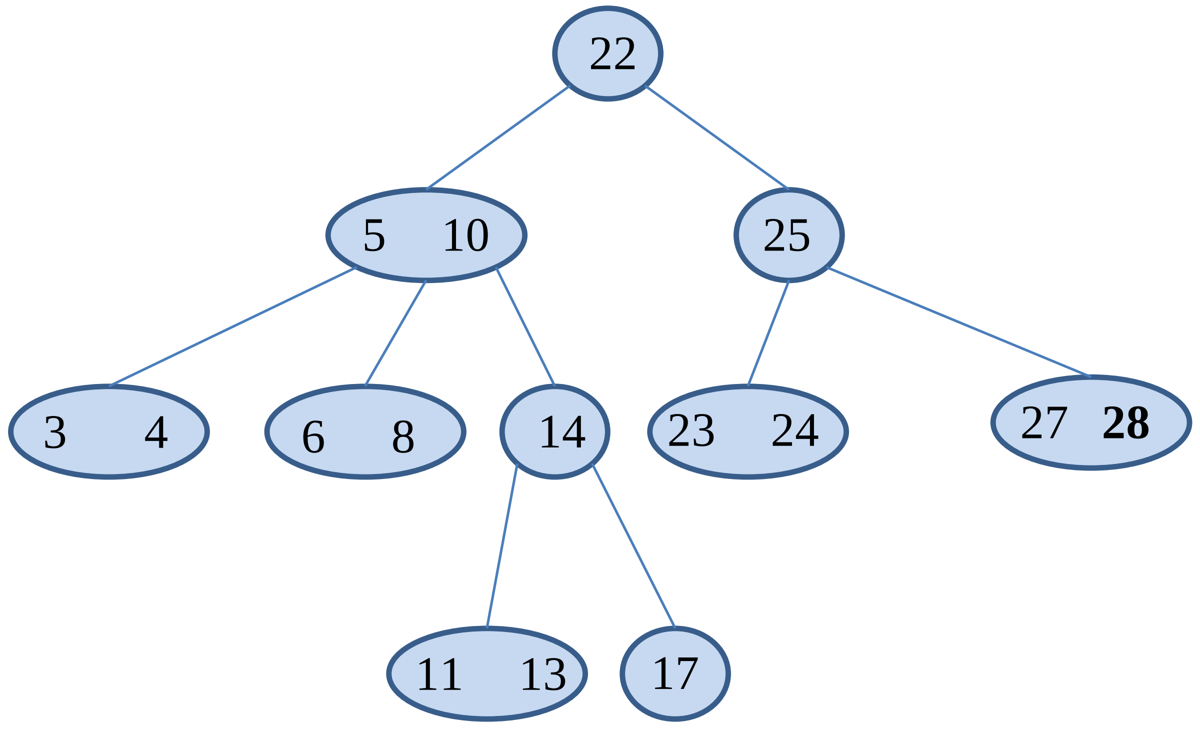
## Insert value 28



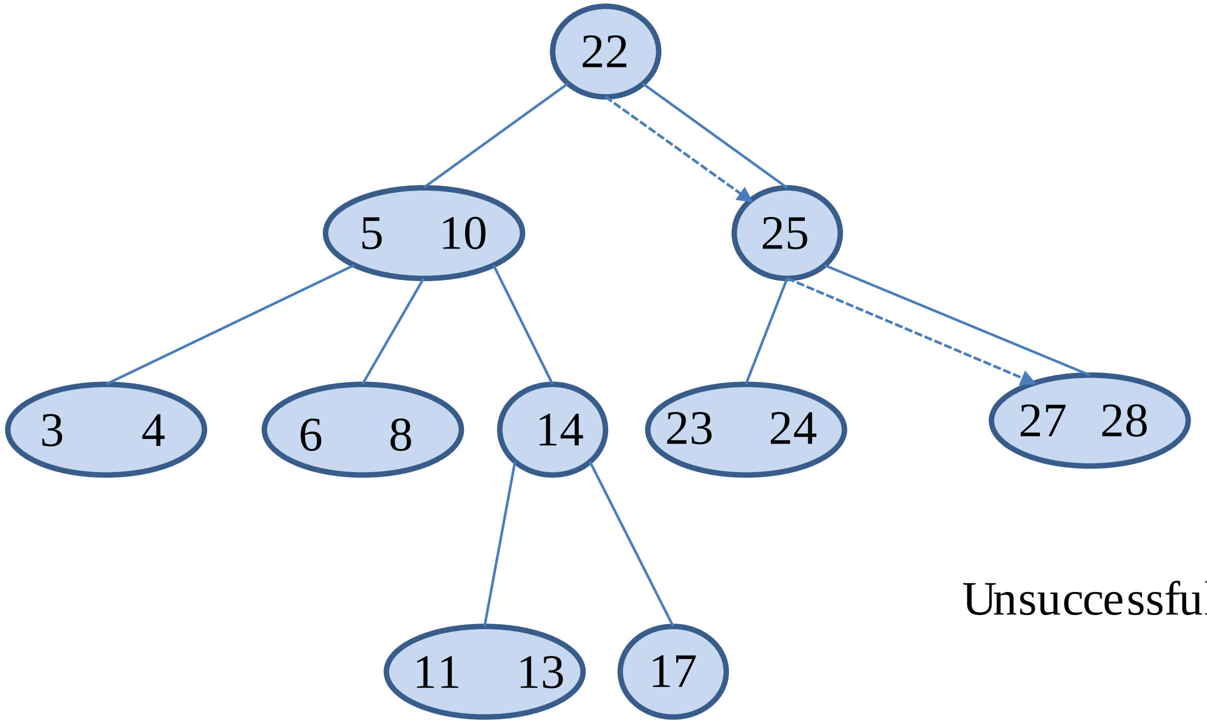
Unsuccessful search

$m = 3$

## Value 28 inserted

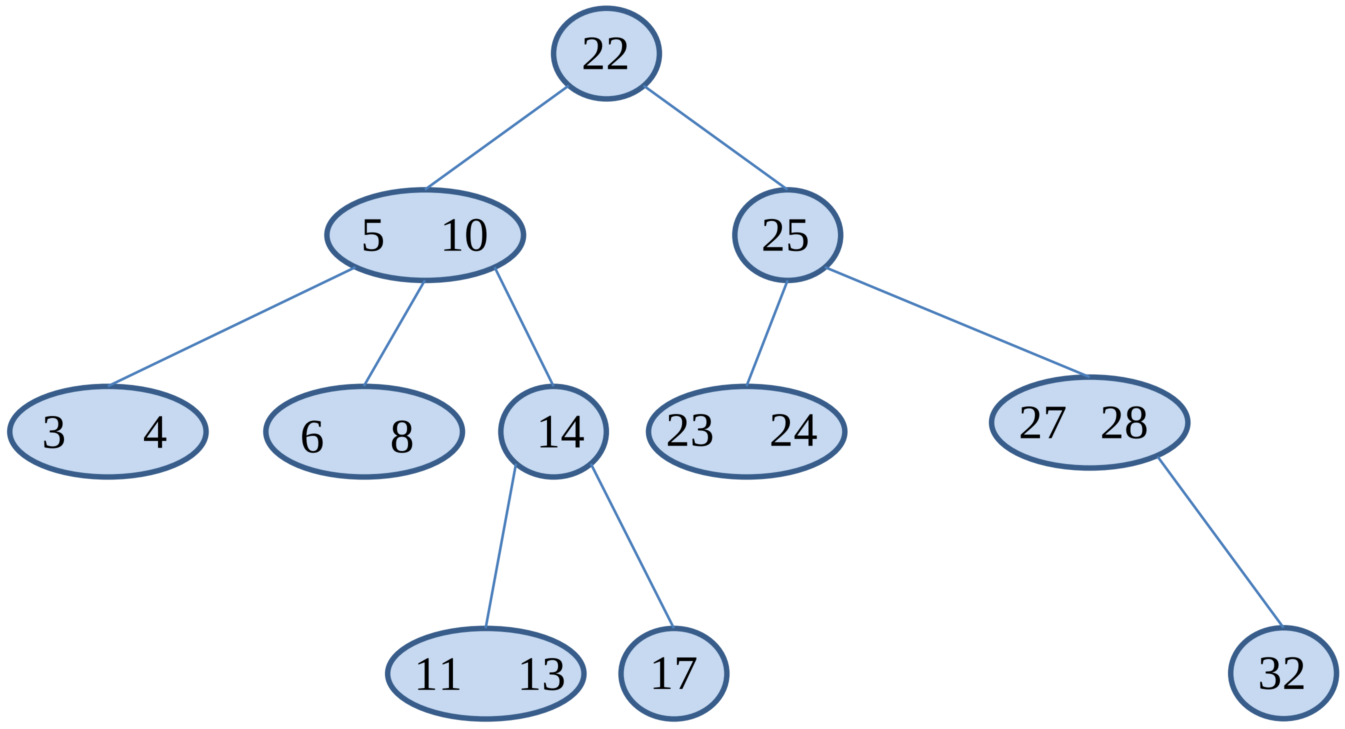


## Insert value 32

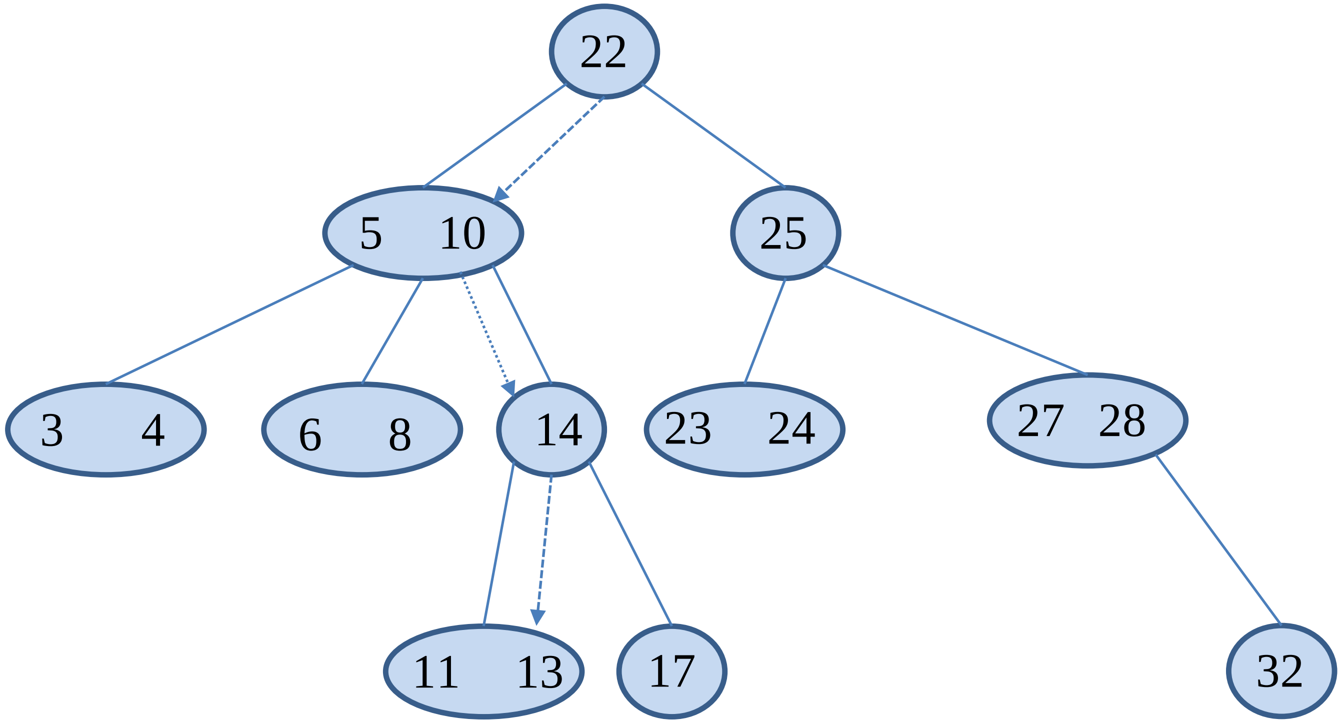


Unsuccessful search

## Value 32 inserted

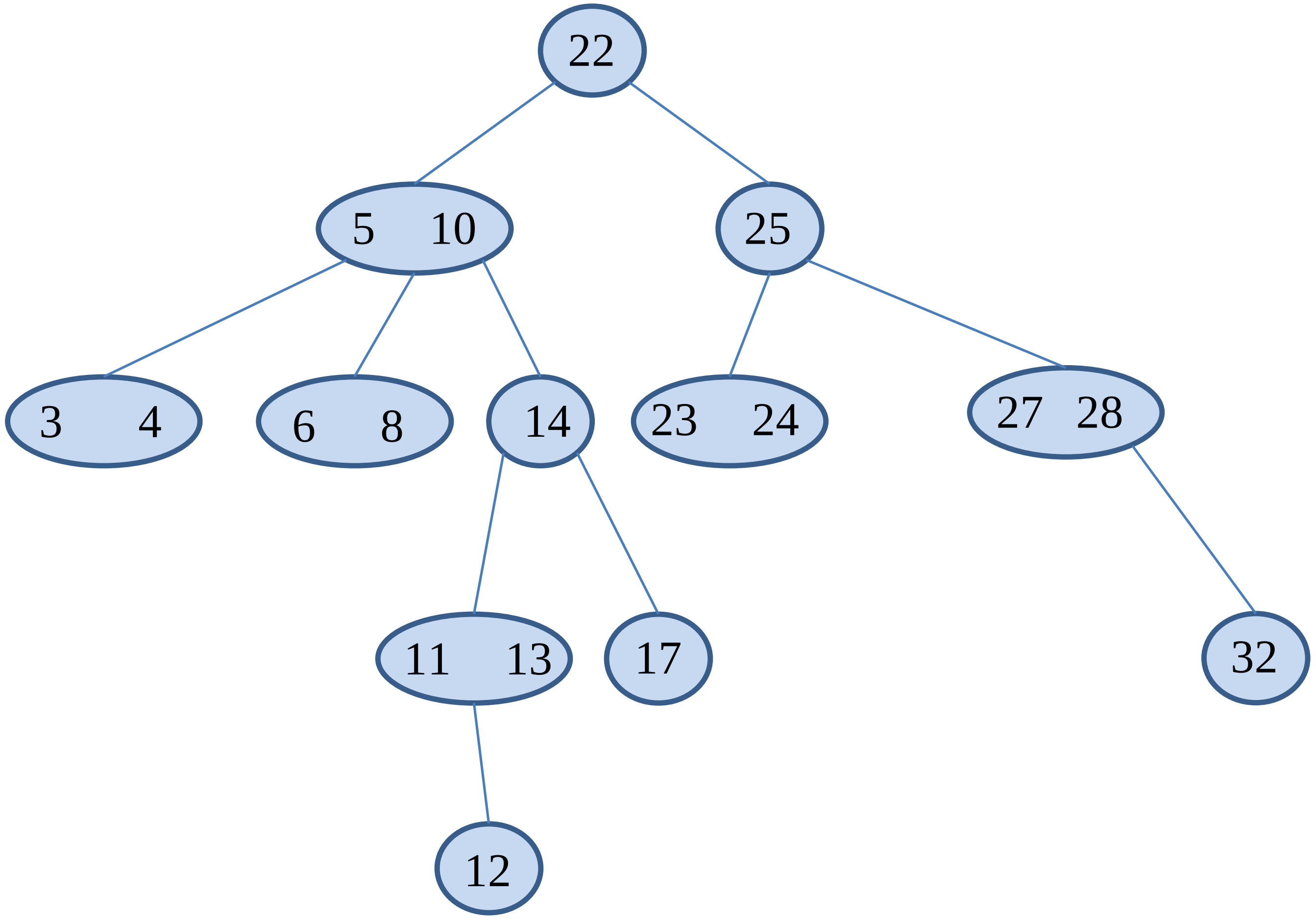


# Insert value 12



Unsuccessful Search

## Value 12 inserted



## Deletion from a multi-way search tree

Left as an exercise.

## Complexity of operations

- We need to traverse the tree from the root to a leaf
- The time spent at each node is constant
  - Eg. find $i$ such that $k_{i-1} < l < k_i$
  - Assuming $m$ is **fixed**!
- So as usual all complexities are $O(h)$
  - $O(n)$ in the worst-case

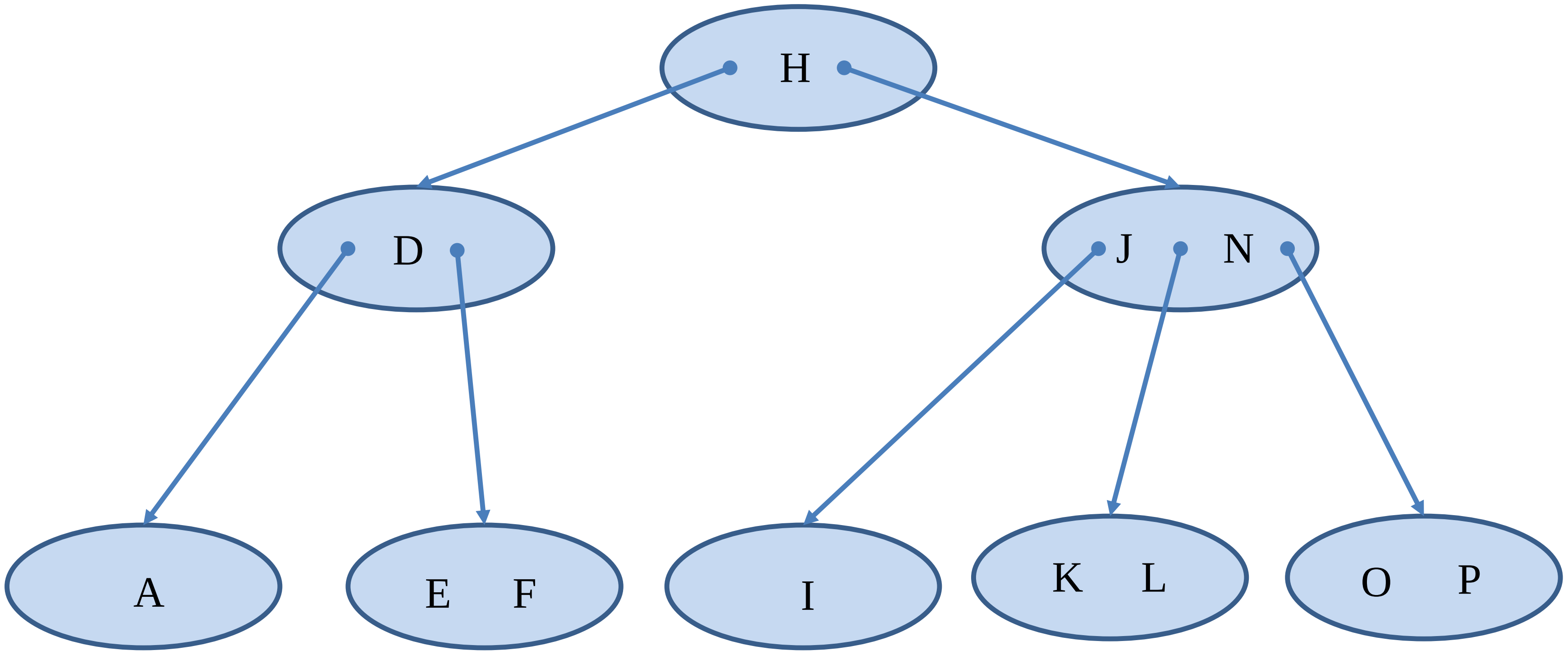
# Balanced multi-way search trees

- Similarly to BSTs we need to keep the tree **balanced**
  - So that $h = O(\log n)$
- AVL where a kind of balanced BSTs
- We will study two kinds of **balanced multi-way** search trees:
  - **2-3 trees**
  - **2-3-4 trees** (also known as 2-4 trees)

## 2-3 trees

- A **2-3 tree** is a 3-way search tree which has the following properties
- **Size property**
  - Each node contains **1 or 2 values**
  - **Internal** nodes with $n$ values have exactly $n+1$ **children**
- **Depth property**
  - All **leaves** have the **same depth** (lie on the same level)

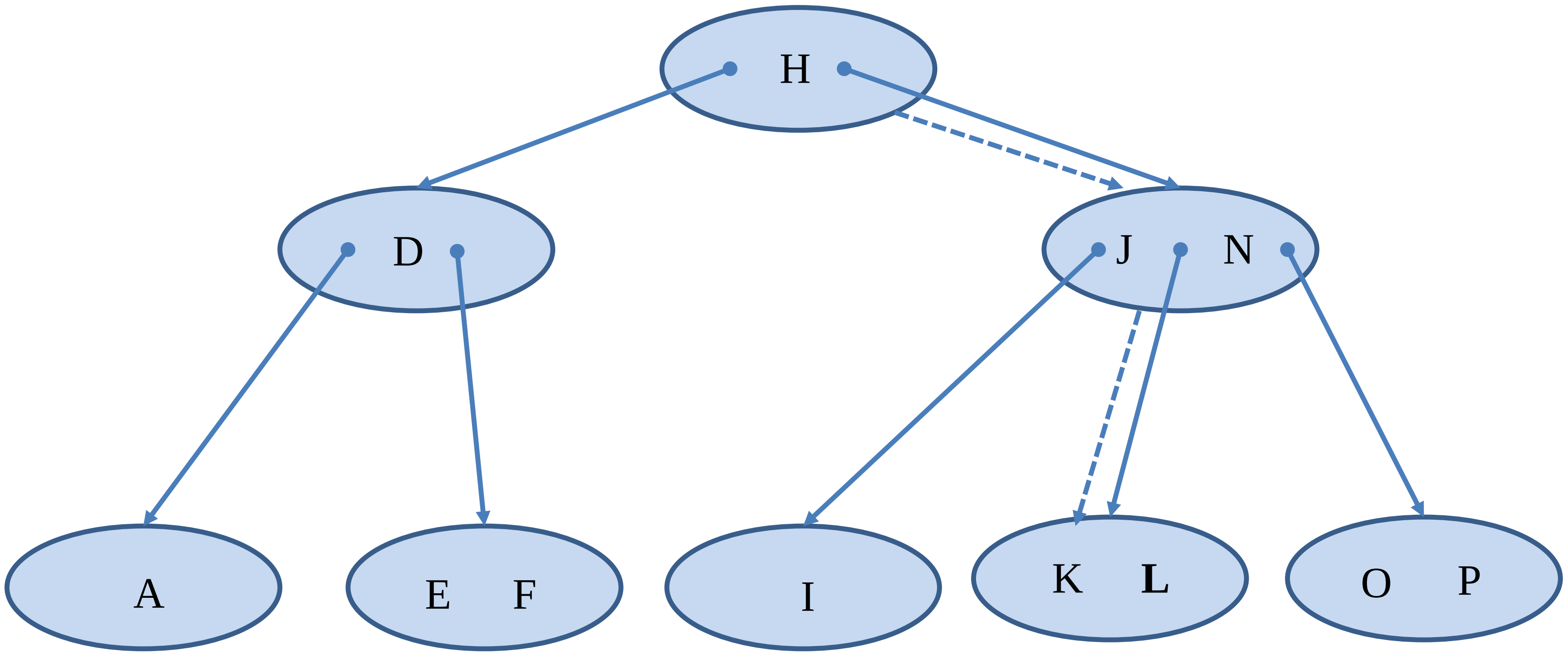
## Example of 2-3 tree



## Height of 2-3 trees

- **All nodes** at **all levels** except the last one are **internal**
  - And each internal node has at least 2 children
  - So at level $i$ we have at least $2^i$ nodes
- Hence $n \geq 2^h$, in other words $h = O(\log n)$
- So we can search for an element in time $O(\log n)$
  - Using the standard algorithm for $m$-way trees

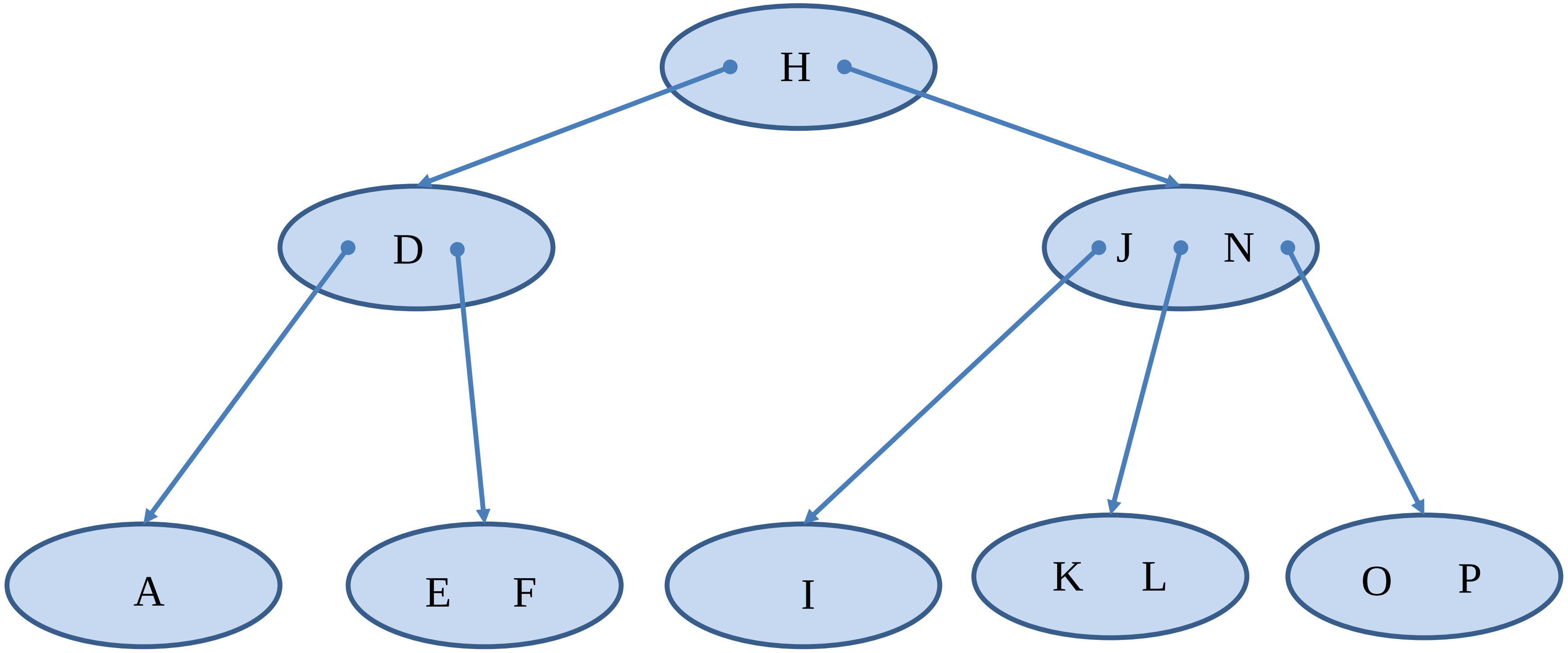
## Search for L



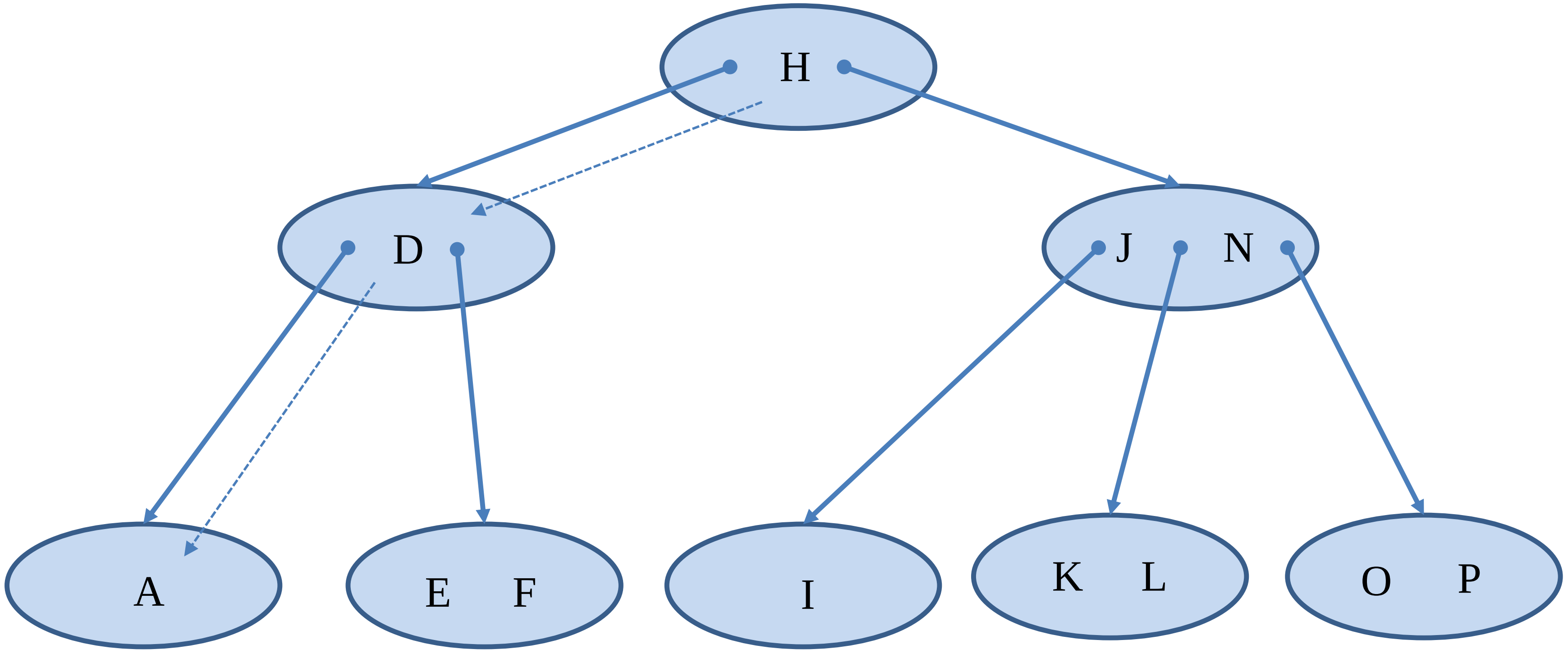
## Insertion in 2-3-trees

- We can start by following the generic algorithm for $m$-way trees
- Search for the value $l$ we want to insert
- If found, stop (no duplicates)
- If not found, insert at the **leaf** we reached

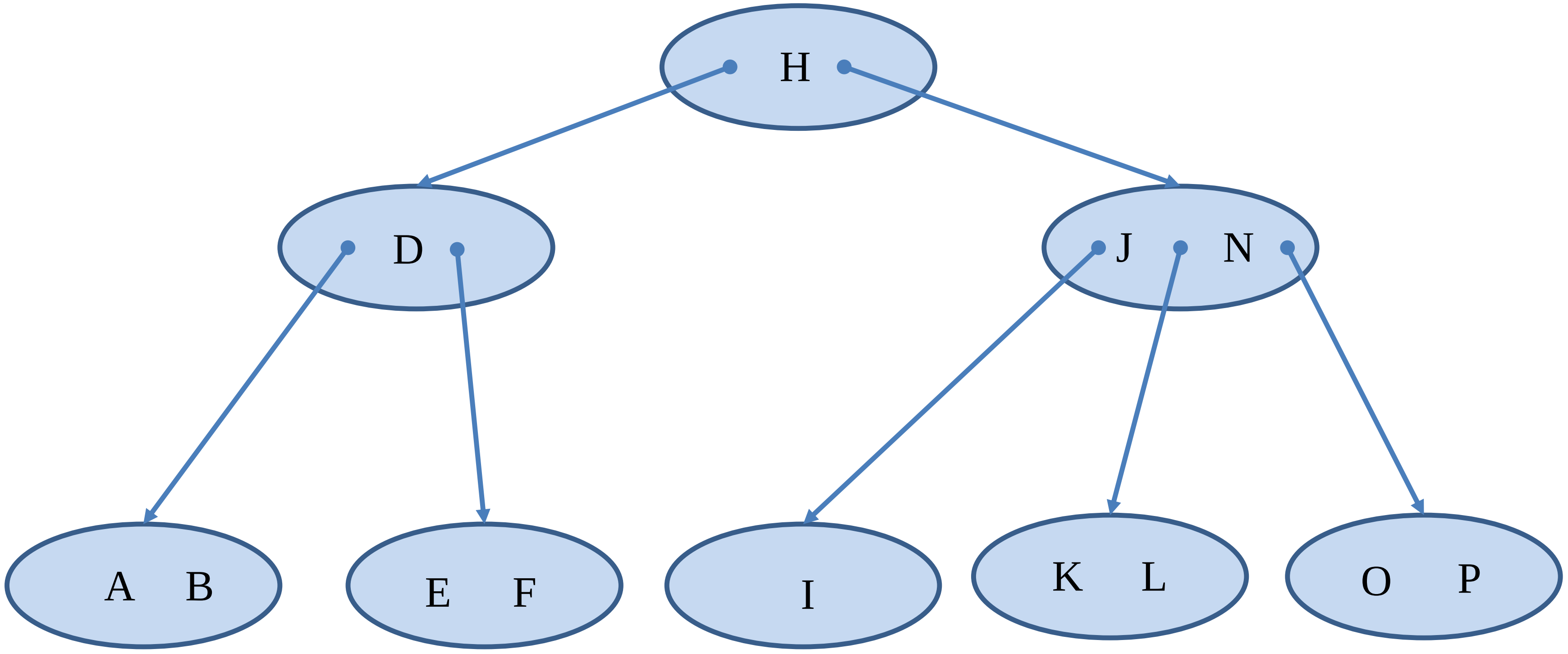
## Example: insert B



## Example: insert B



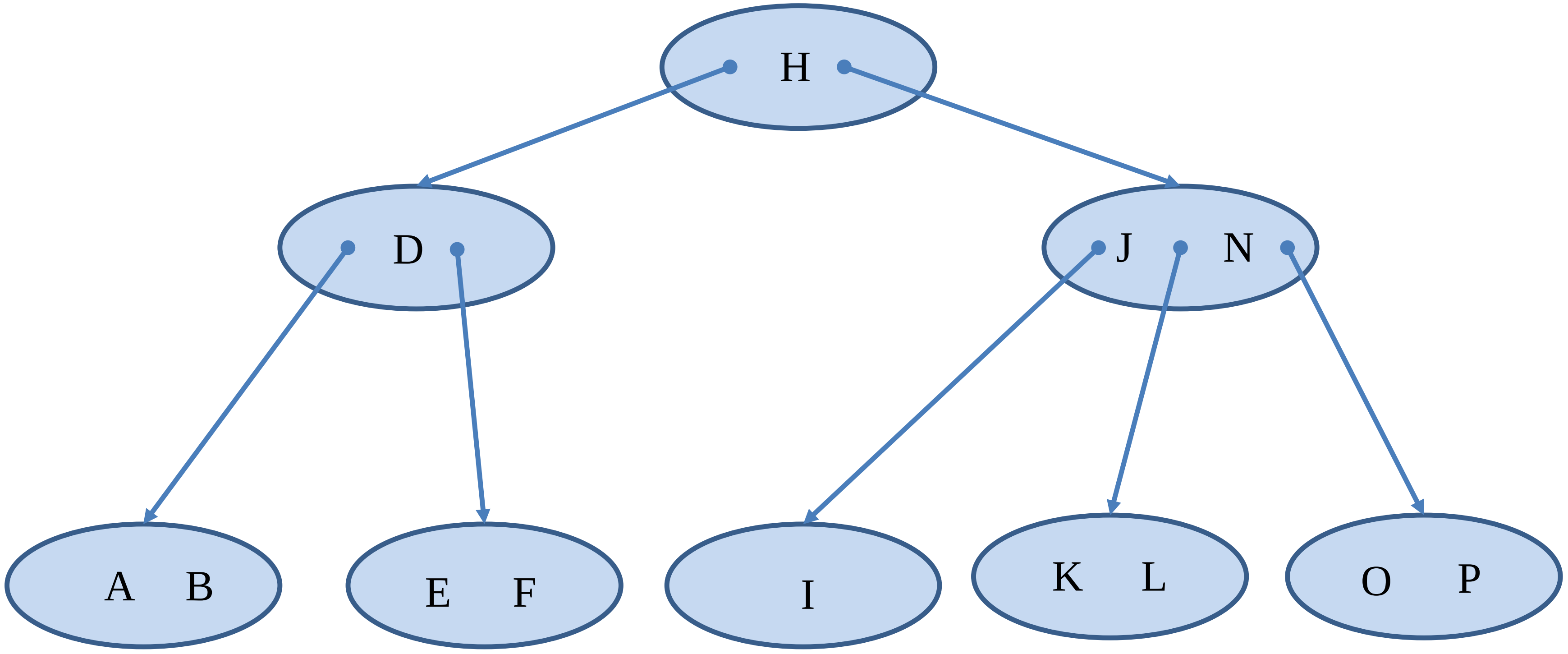
## Example: result



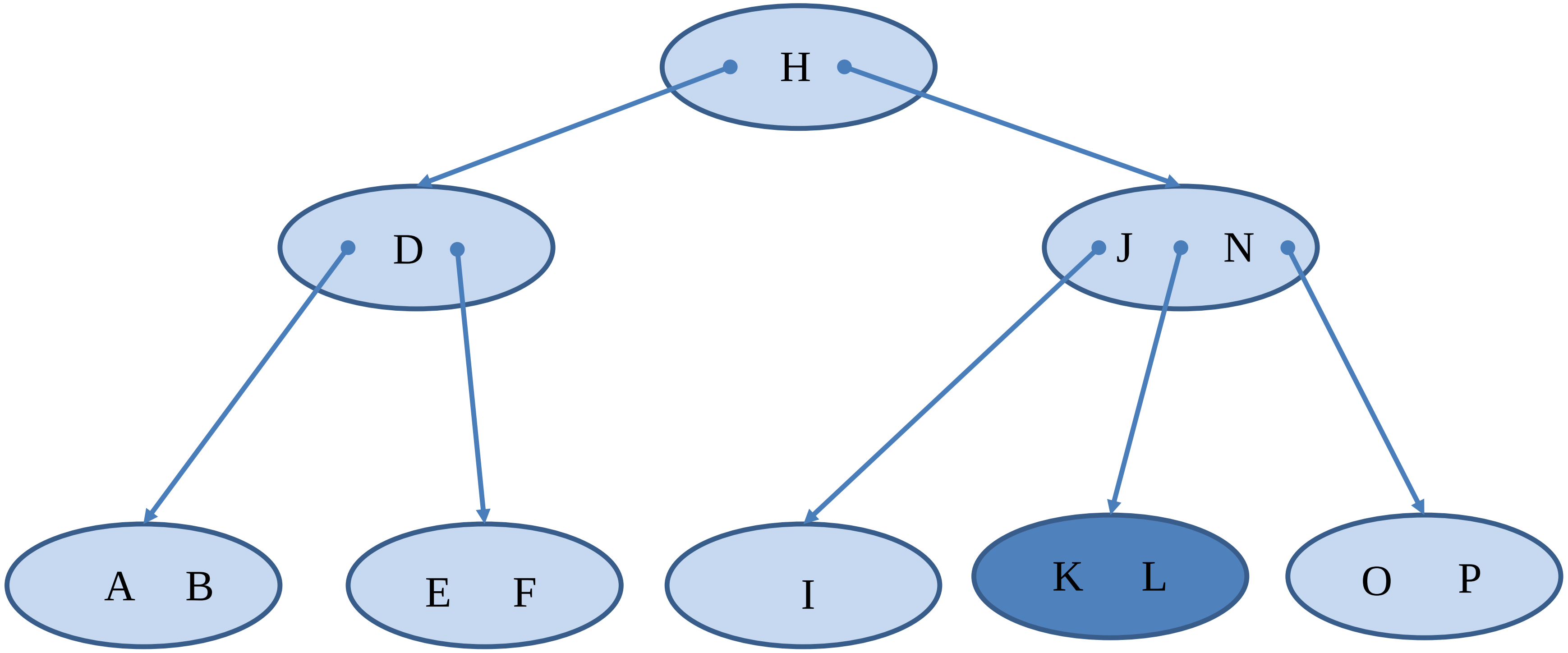
## Insertion in 2-3-trees

- But what if there is **no space at the leaf** (overflow)?
- The standard algorithm will insert a child at the leaf
  - But this **violates the depth property**!
  - The new leaf is not at the same level
- Different strategy
  - **split** the overflowed node into two nodes
  - pass the **middle value** to the parent (**separator** of the two nodes)
- The middle value might **overflow the parent**
  - Same procedure: split and send the middle value up

## Example: insert M

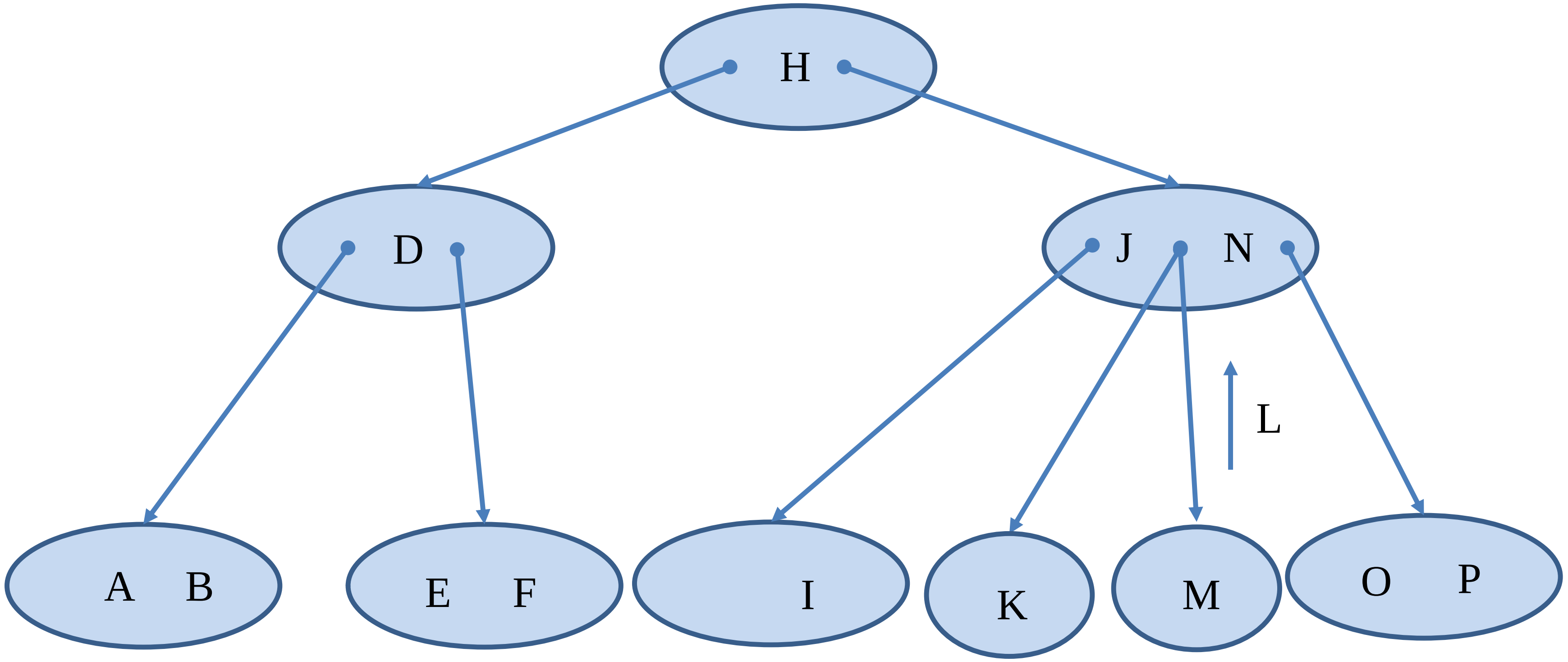


## Example: insert M



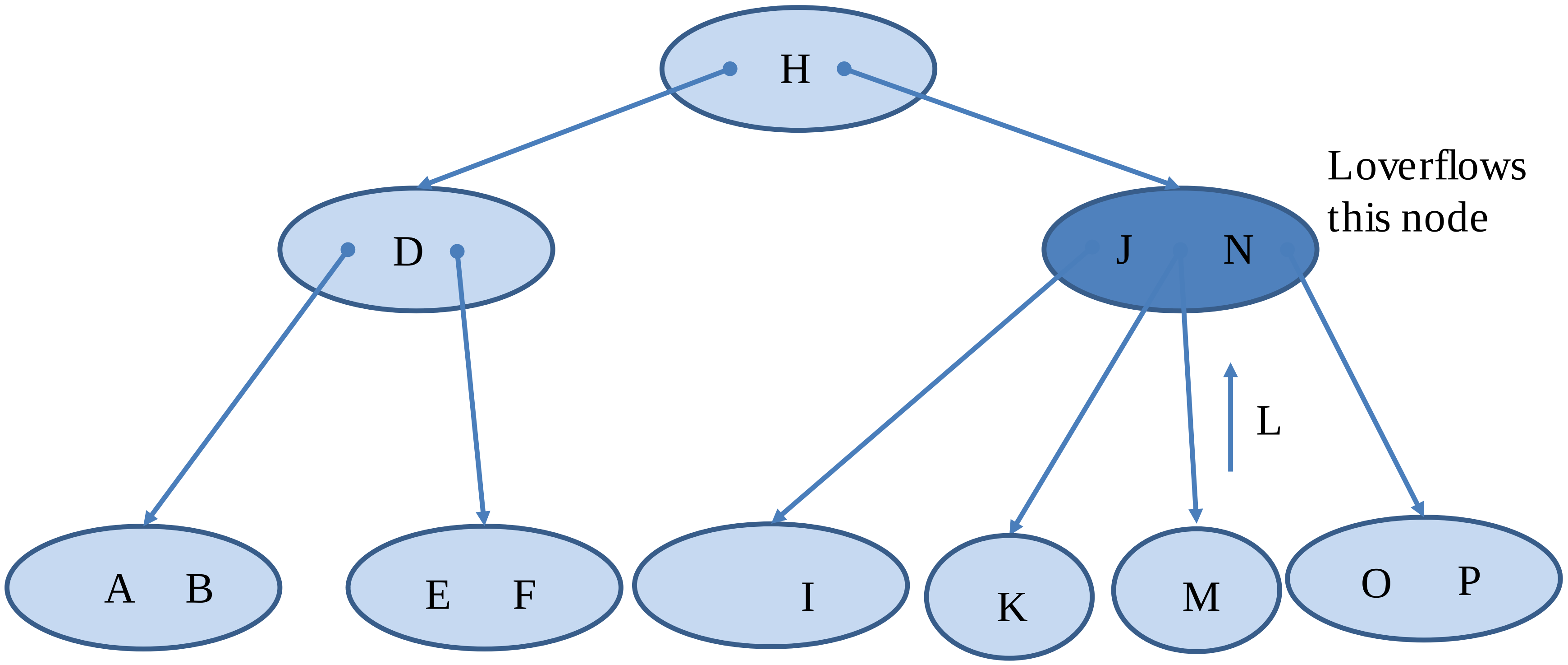
M overflows this node.

## Example: insert M

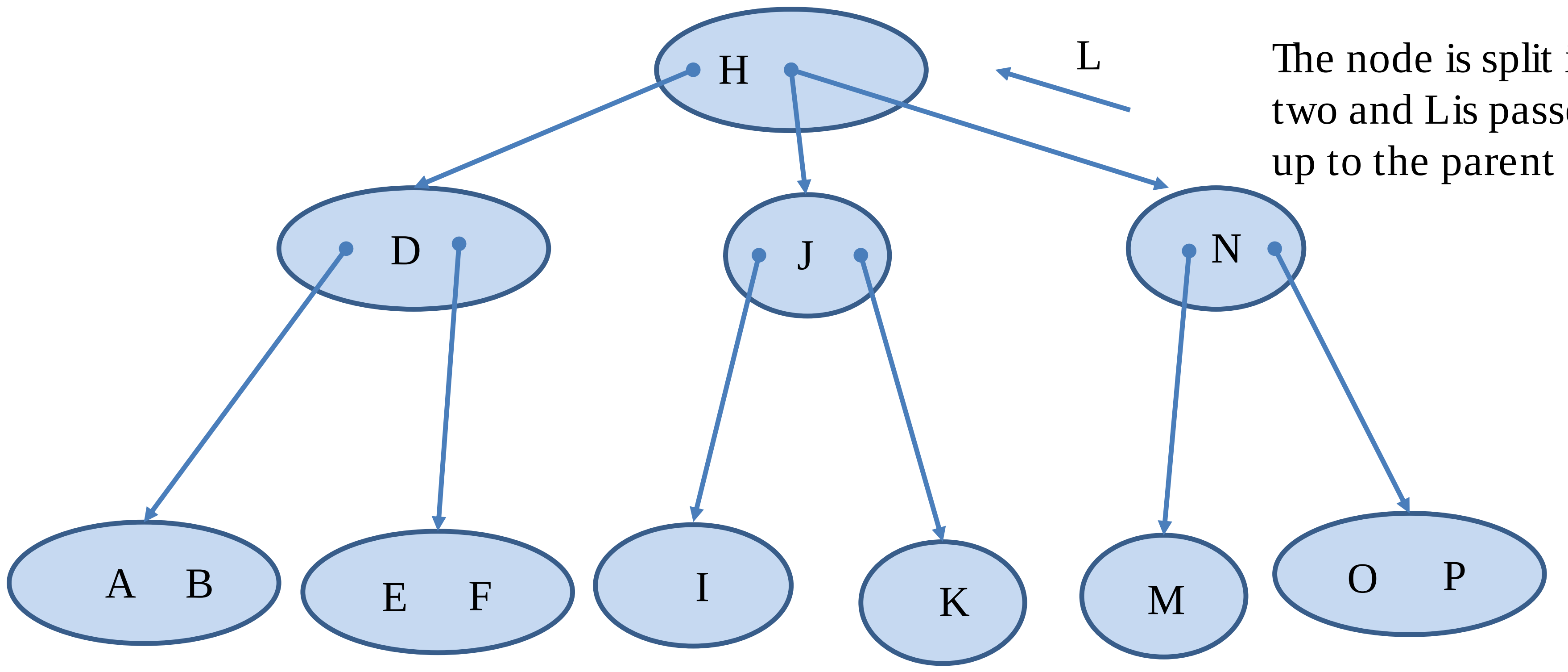


The node is split in two and L is passed to the parent node

## Example: insert M

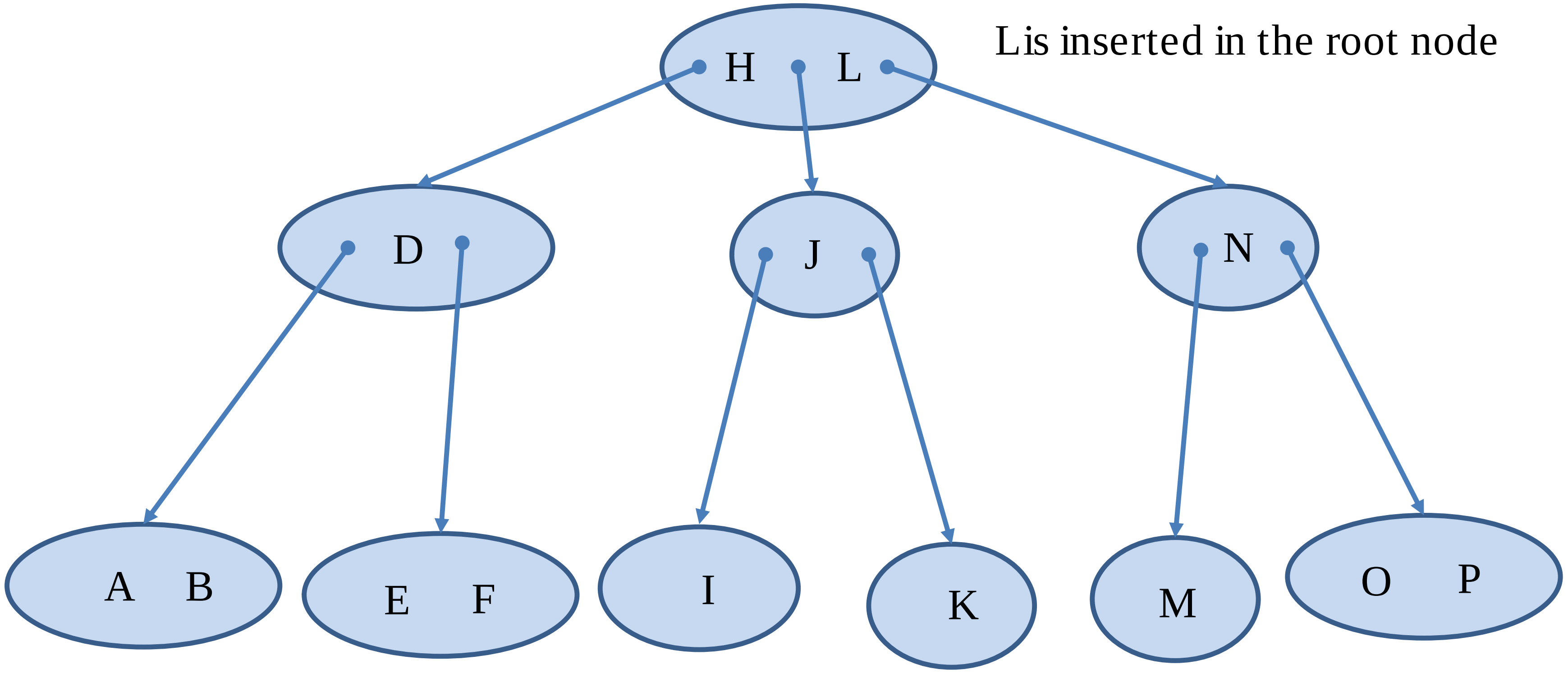


# Example: insert M



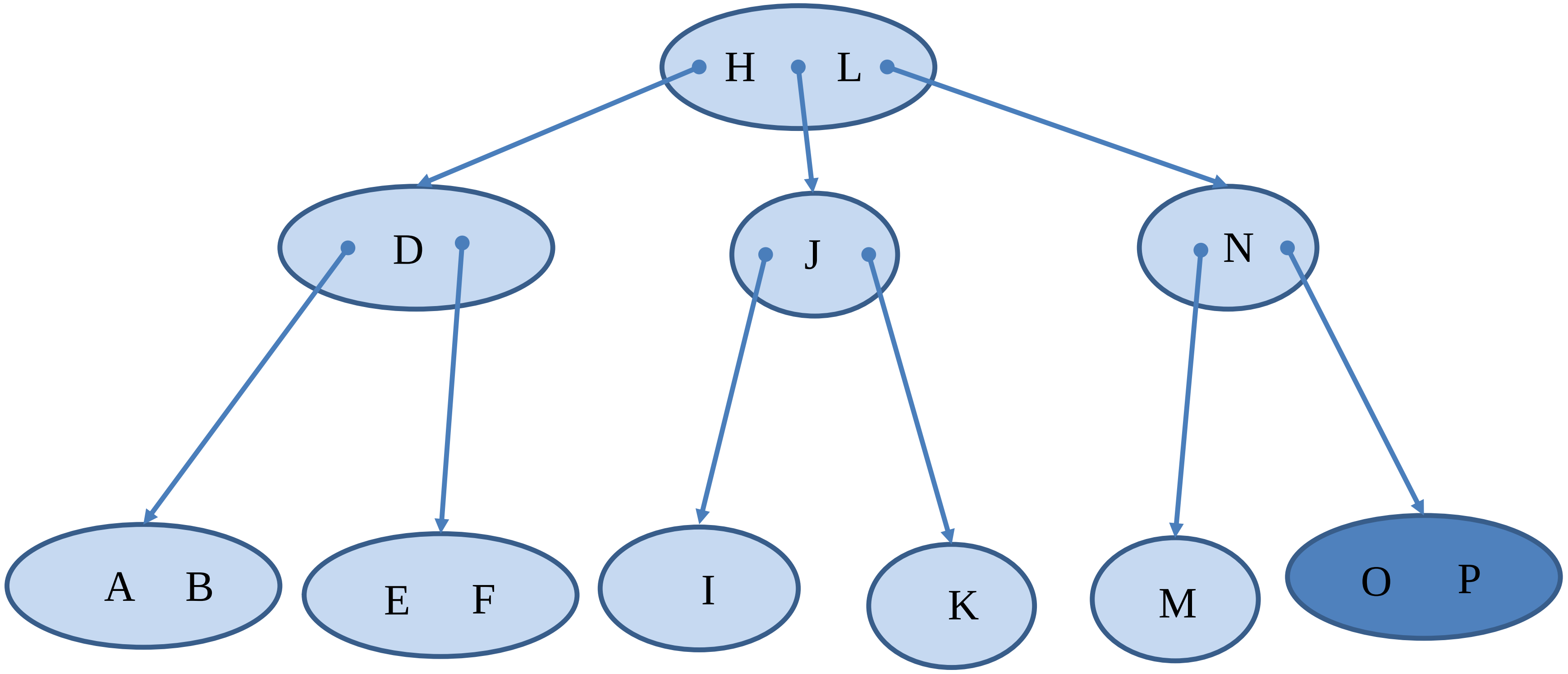
The node is split in two and L is passed up to the parent

## Example: result



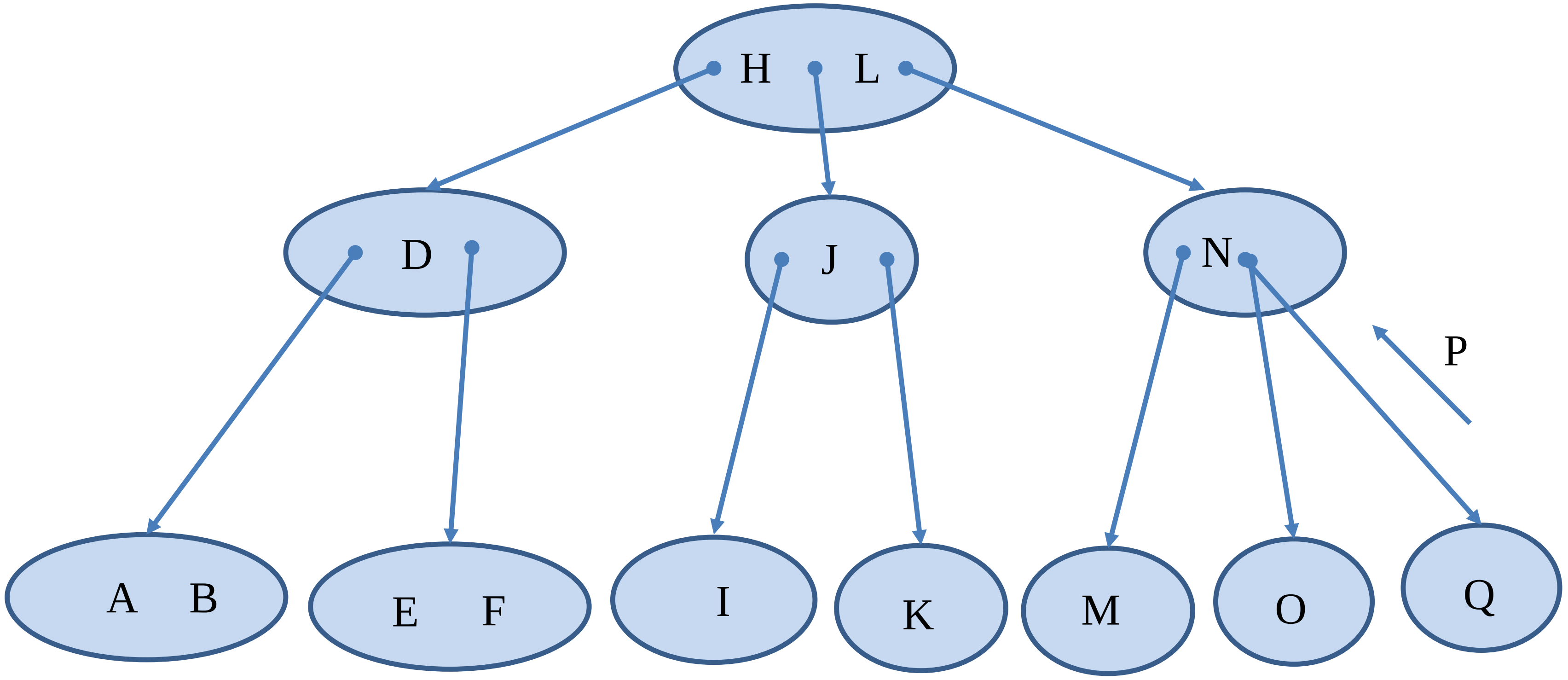
L is inserted in the root node

## Example: insert Q



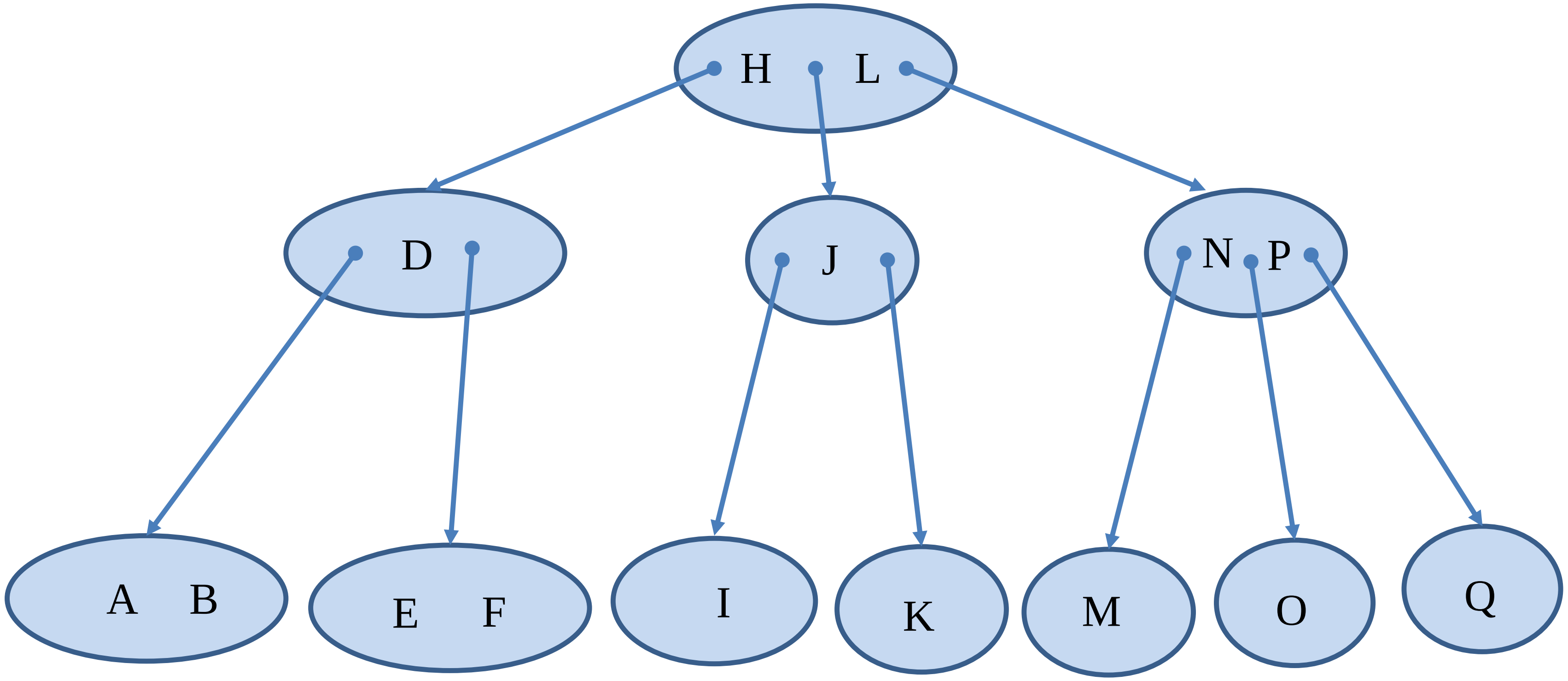
Q overflows
this node

## Example: insert Q

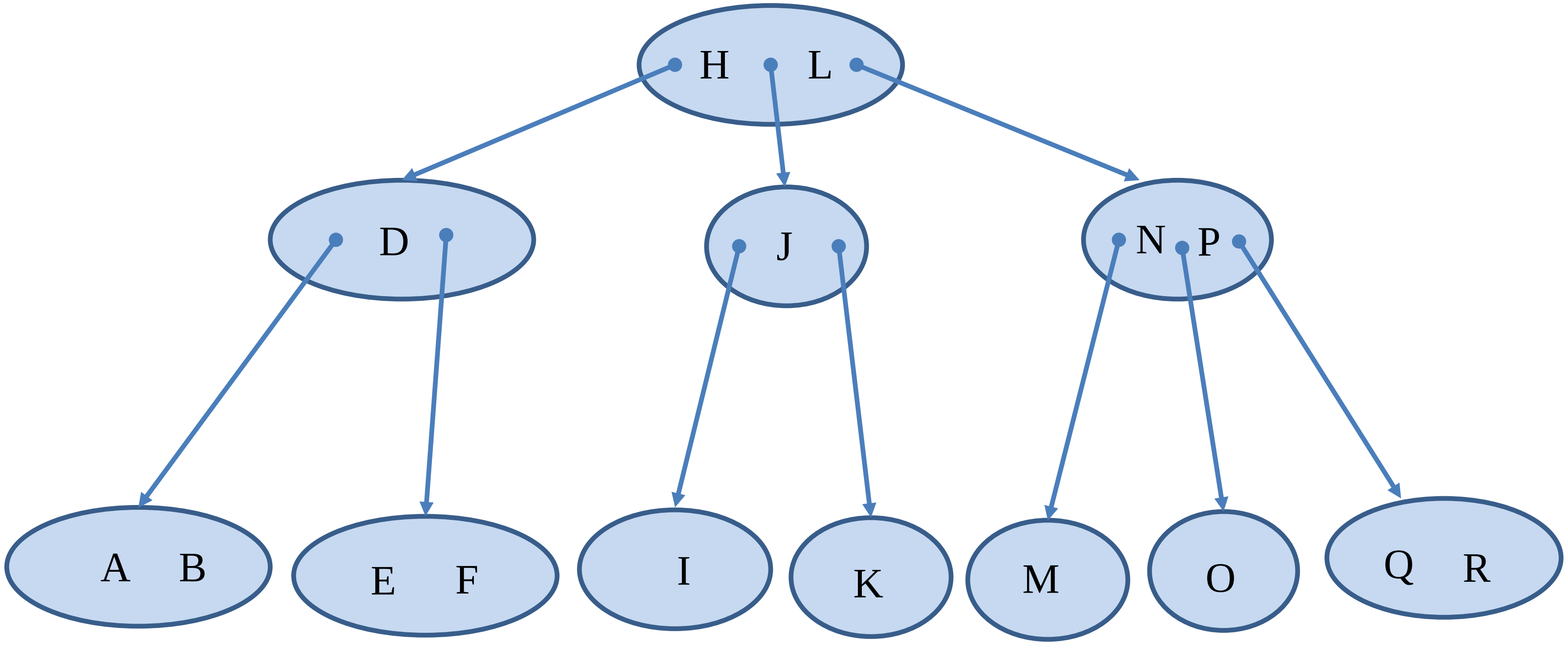


This node is split up
and P is passed up

## Example: result



## Example: insert R

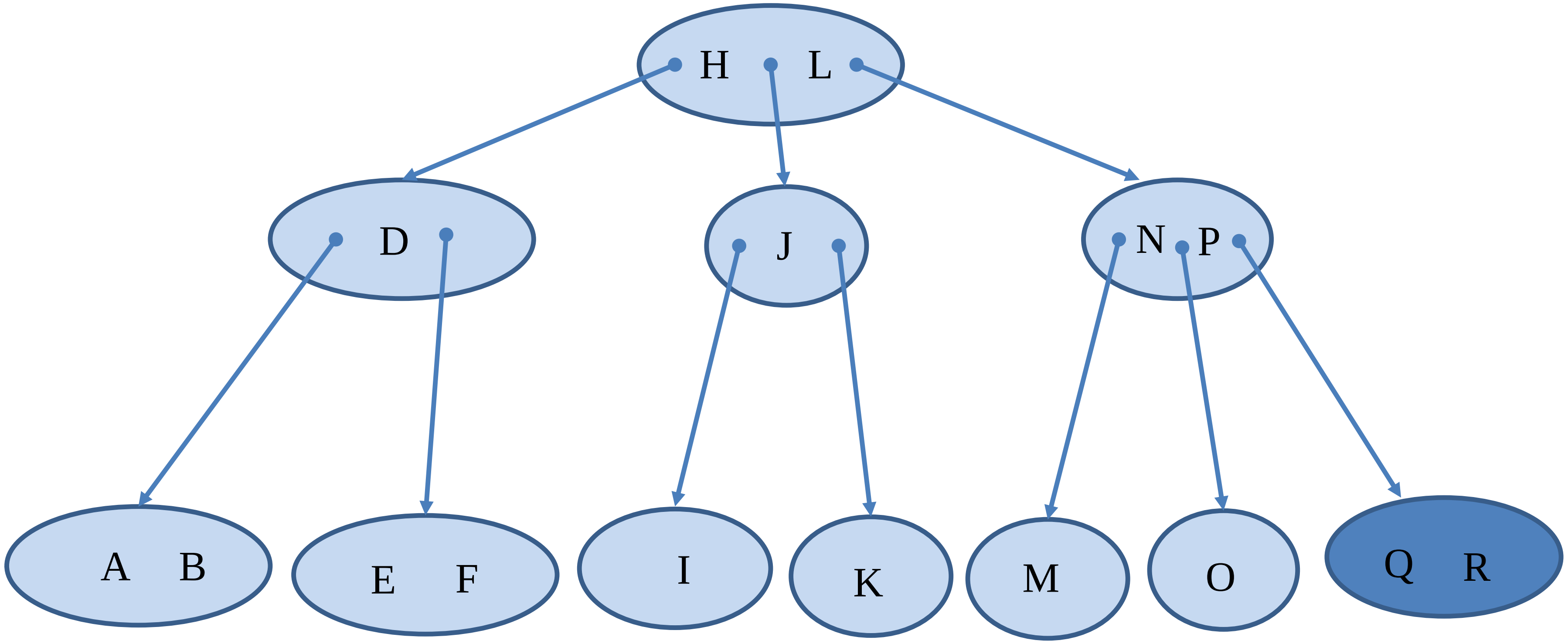


R is inserted in the node with Q where there is space.

## Insertion in 2-3-trees

- The **root** might also **overflow**
- Same procedure
  - Split it
  - The middle value moves up, creating a **new root**
- This is the **only** operation that **increases** the tree’s **height**
  - It increases the depth of **all nodes** simultaneously
  - 2-3-trees grow at the root, not at the leaves!

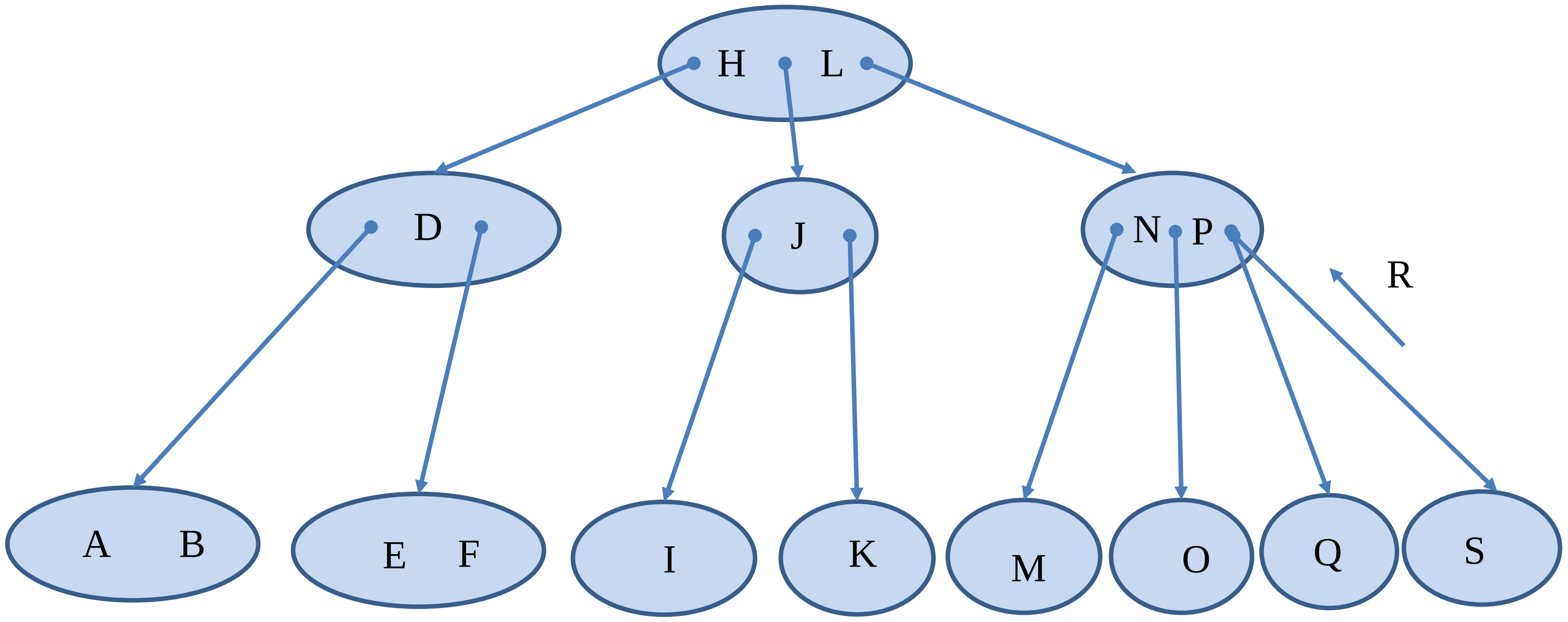
## Example: insert S



S overflows this node

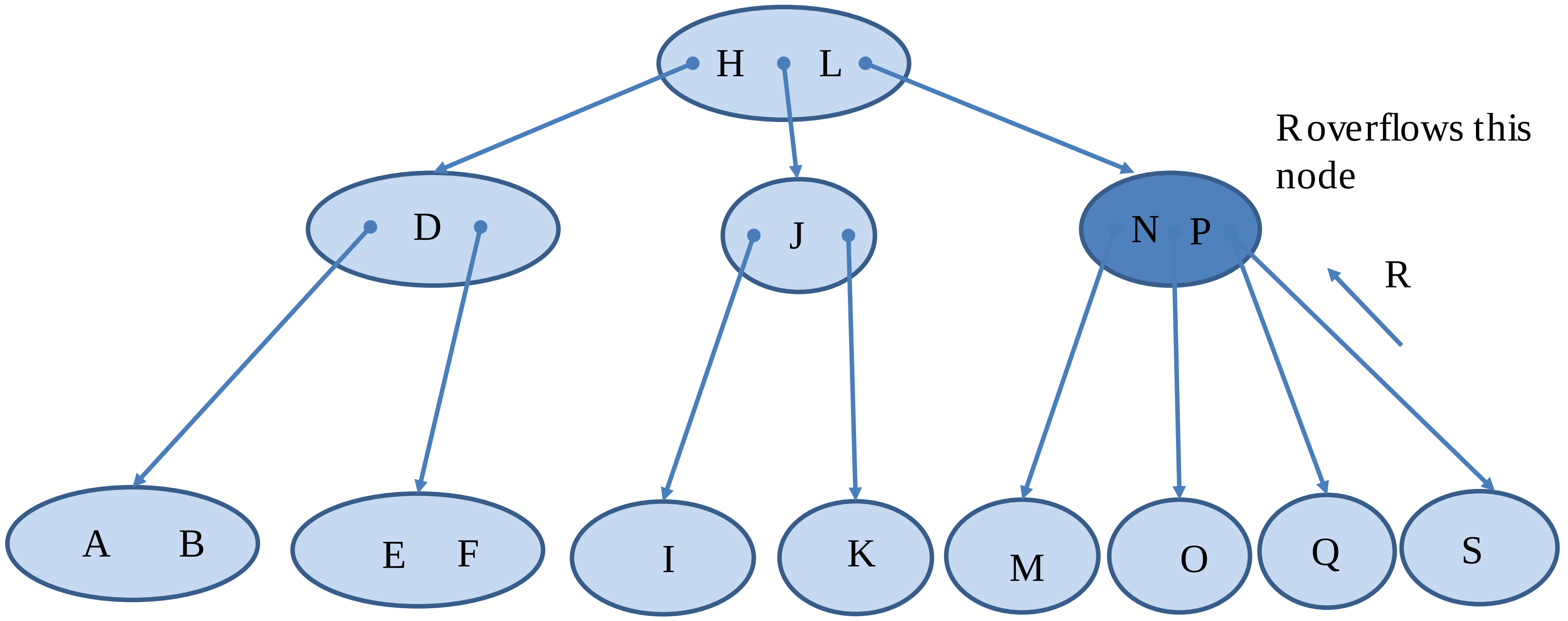
S overflows this node

# Example: insert S

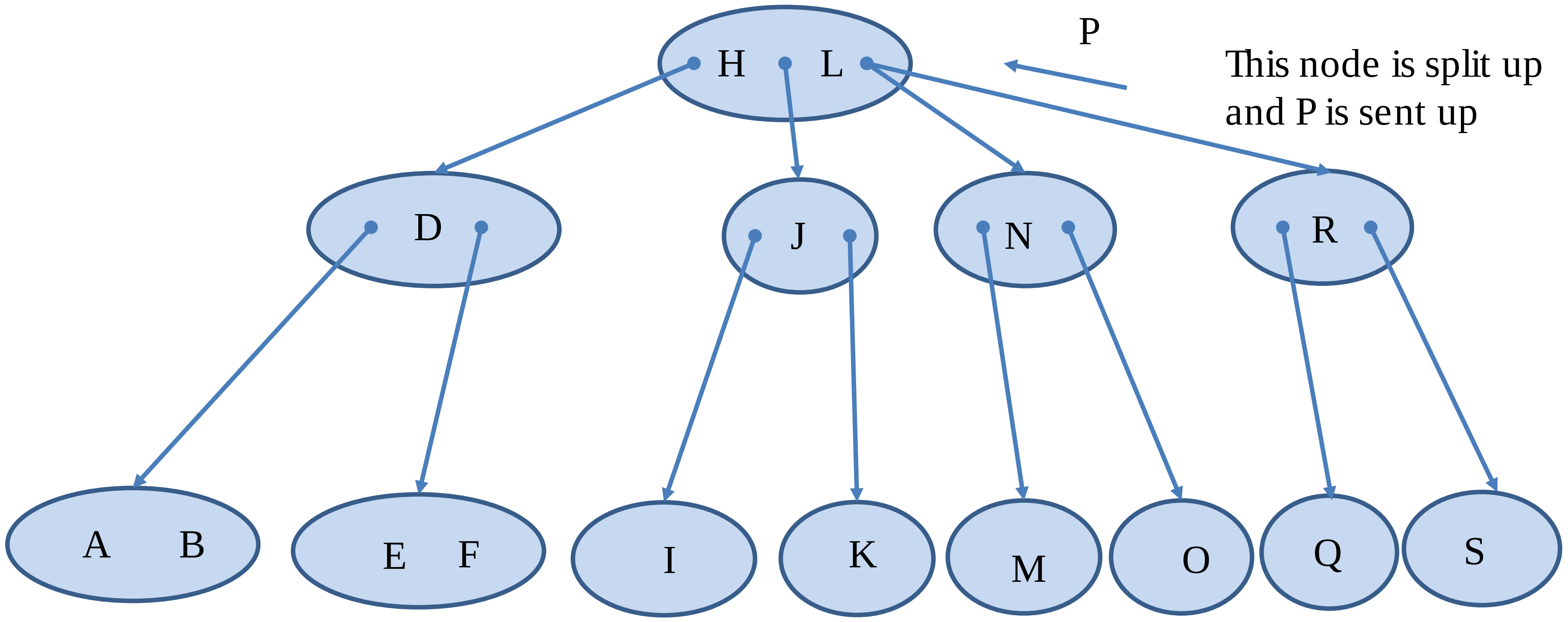


This node is split
and R is sent up

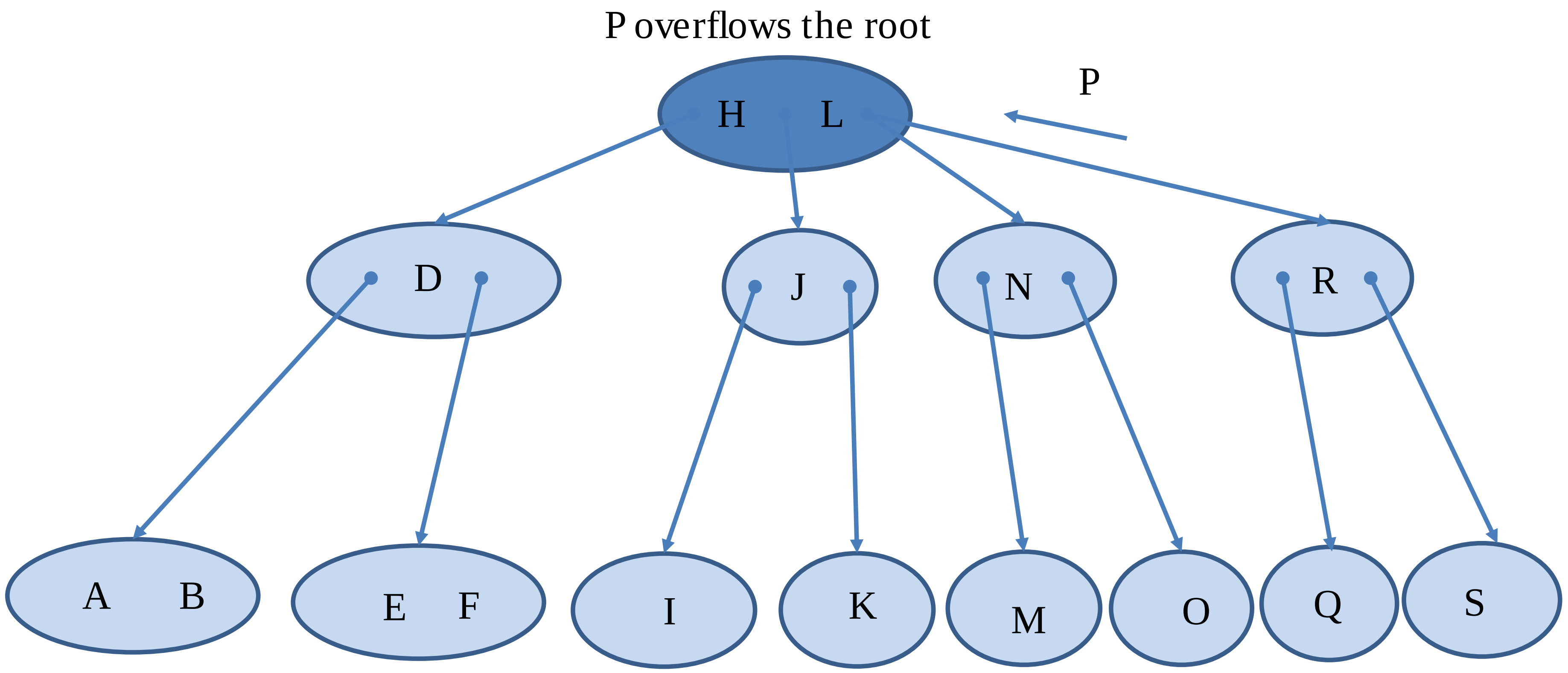
# Example: insert S



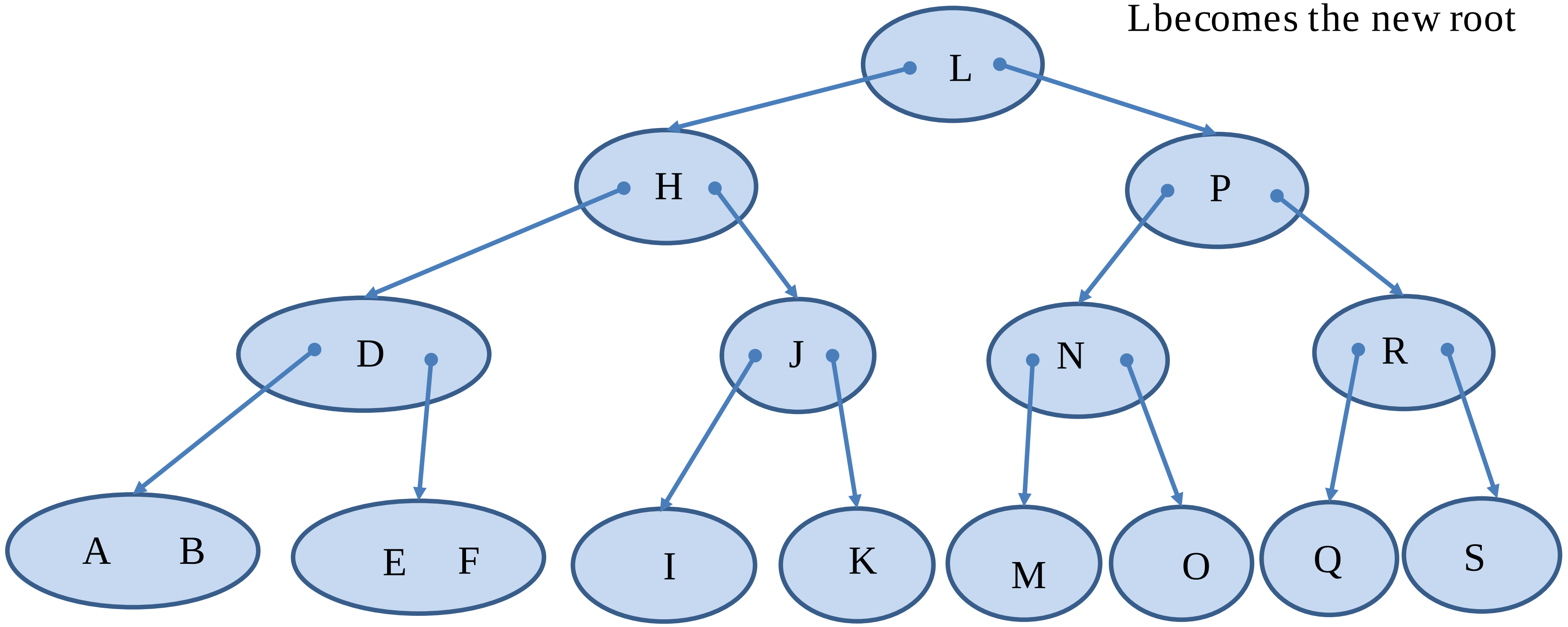
# Example: insert S



## Example: insert S



# Example: result



The root splits and
Lbecomes the new root

## Complexity of insertion

- We traverse the tree
  - From the root to a leaf when searching
  - From the leaf back to the root while splitting
- Each split takes constant time
  - We do at most $h + 1$ of them
- So in total $O(h) = O(\log n)$ steps
  - Recall, the tree is balanced

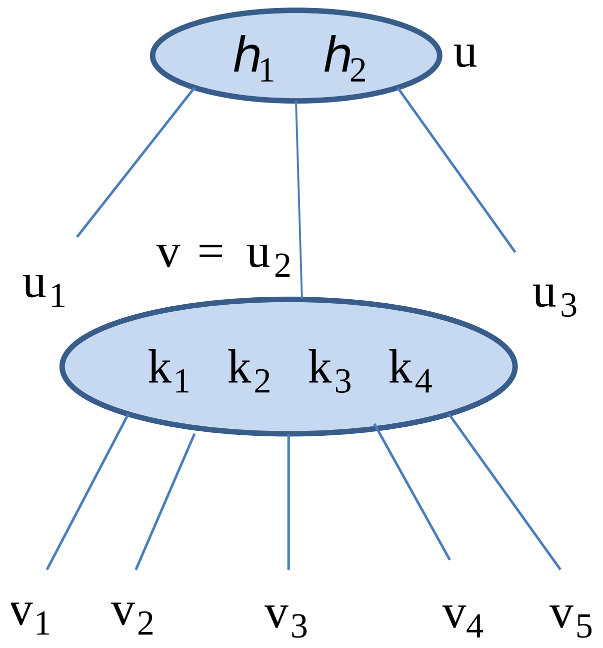
## 2-4 trees

- A **2-4 tree** (or 2-3-4 tree) is a 4-way search tree with 2 extra properties
- **Size property**
  - Each node contains between **1 and 3 values**
  - **Internal** nodes with $n$ values have exactly $n + 1$ **children**
- **Depth property**
  - All **leaves** have the **same depth** (lie on the same level)
- Such trees are **balanced**
  - $h = O(\log n)$
  - Proof: exercise

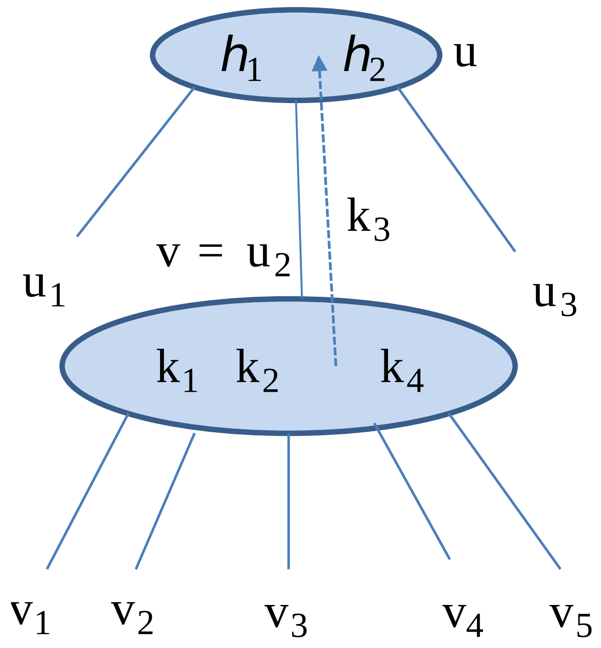
## Insertion in 2-4 trees

- Same as for 2-3-trees
  - Search for the value
  - Insert at a leaf
- In case of an overflow (5-node)
  - Split it into a 3-node and a 2-node
  - Move the separator value $k_3$ to the parent

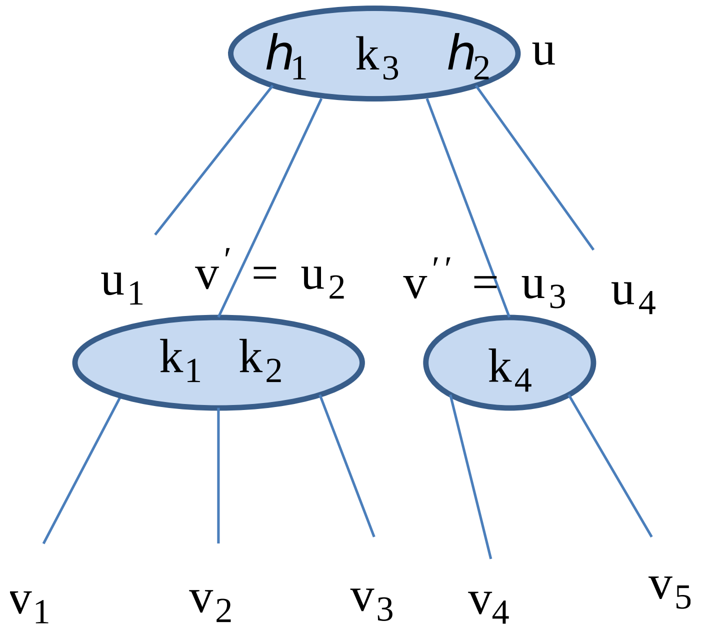
## Overflow at a 5-node



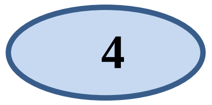
## The separating value is sent to the parent node



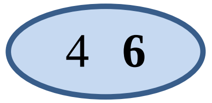
# Node replaced with a 3-node and a 2-node



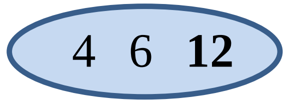
## Example: insert 4



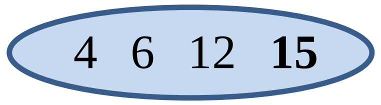
## Example: insert 6



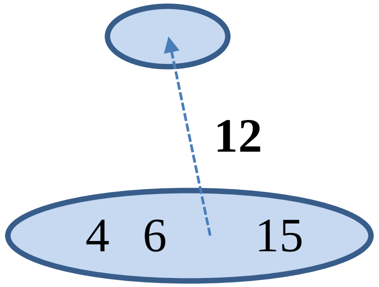
## Example: insert 12



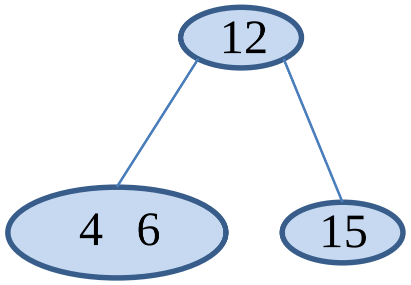
## Example: insert 15 - overflow



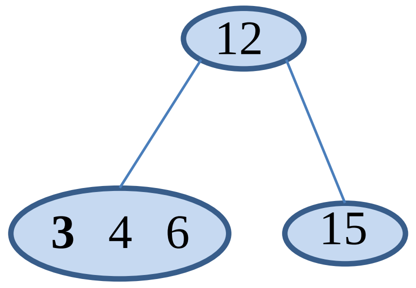
## Creation of new root node



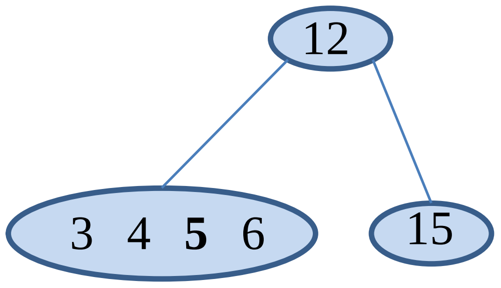
## Split



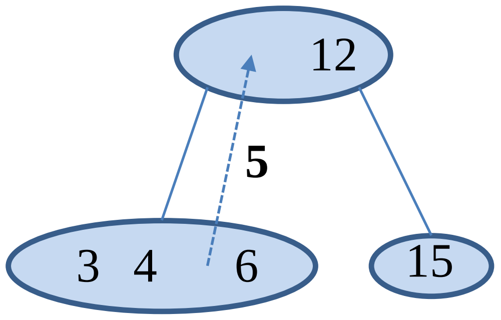
## Example: insert 3



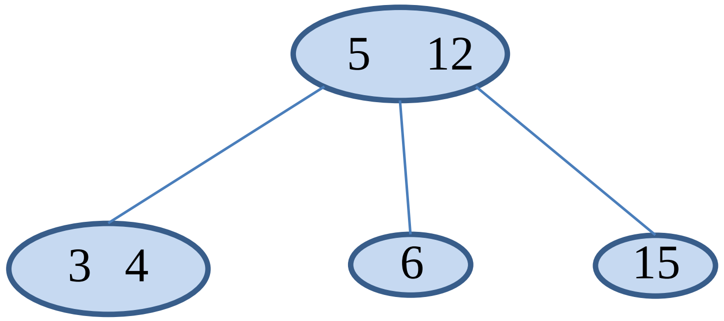
## Example: insert 5 - overflow



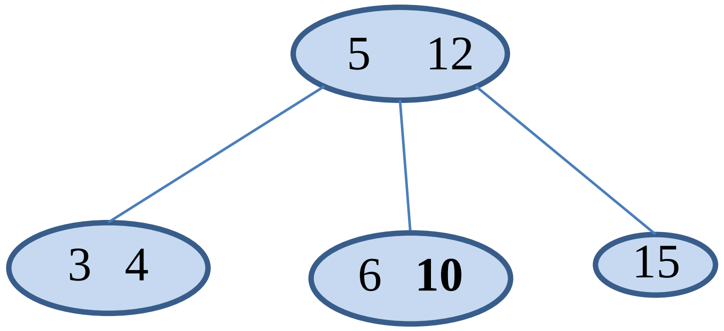
## 5 is sent to the parent node



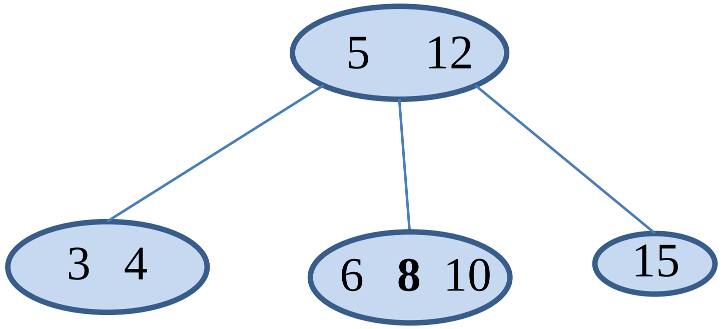
## Split



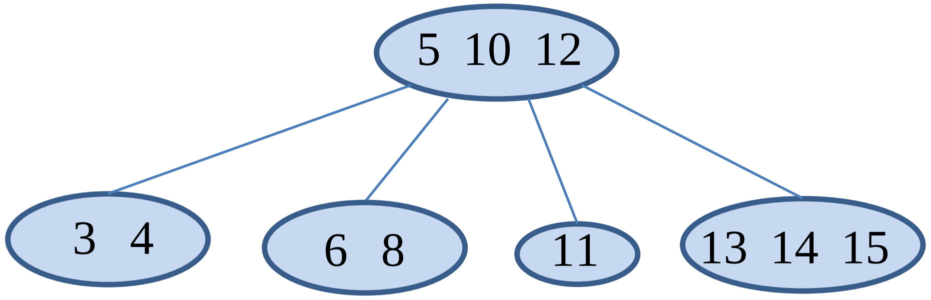
## Example: insert 10



## Example: insert 8

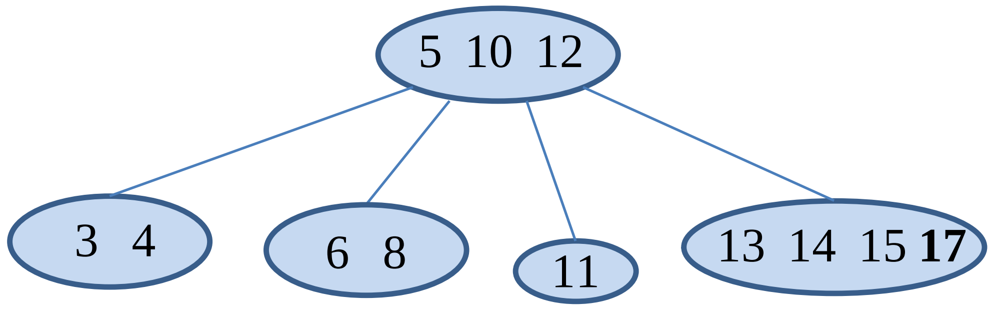


## Example

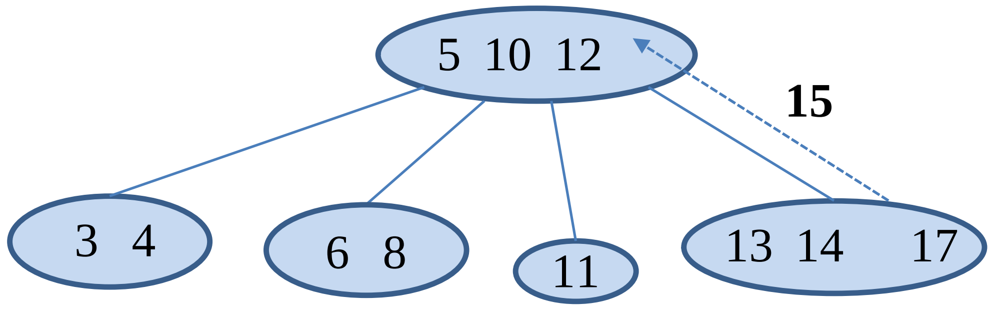


Inserted 11, 13 and 14.

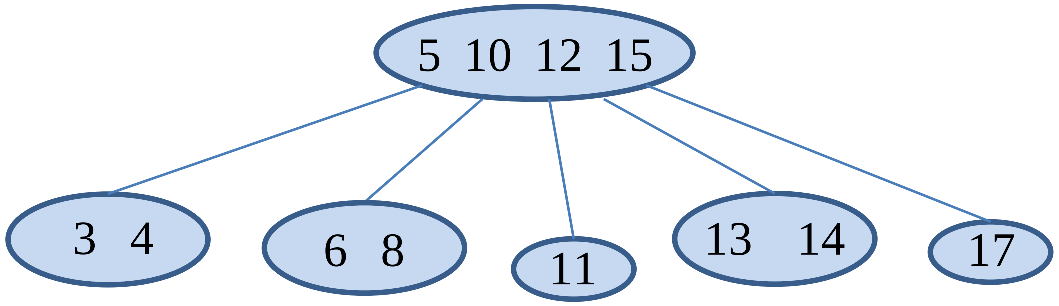
# Example: insert 17 - overflow



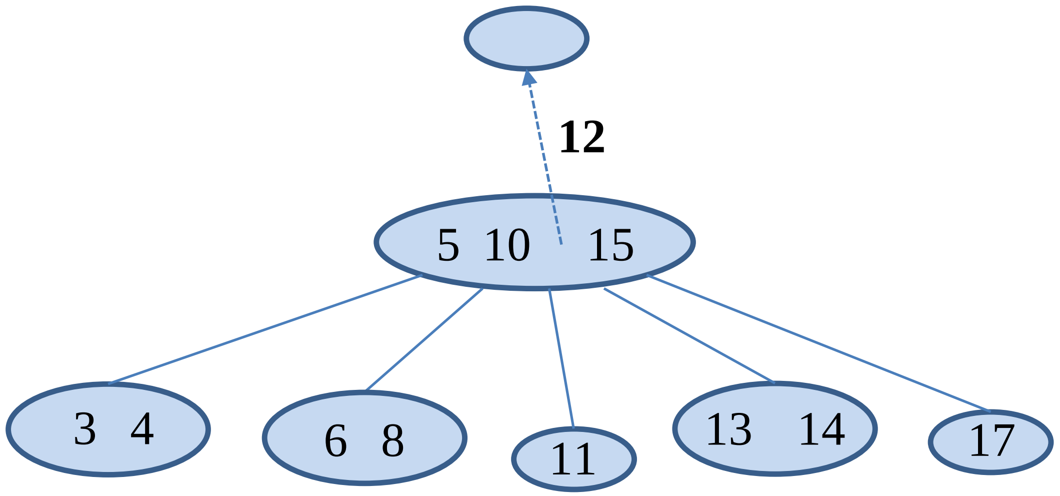
# Split and send 15 to the parent node



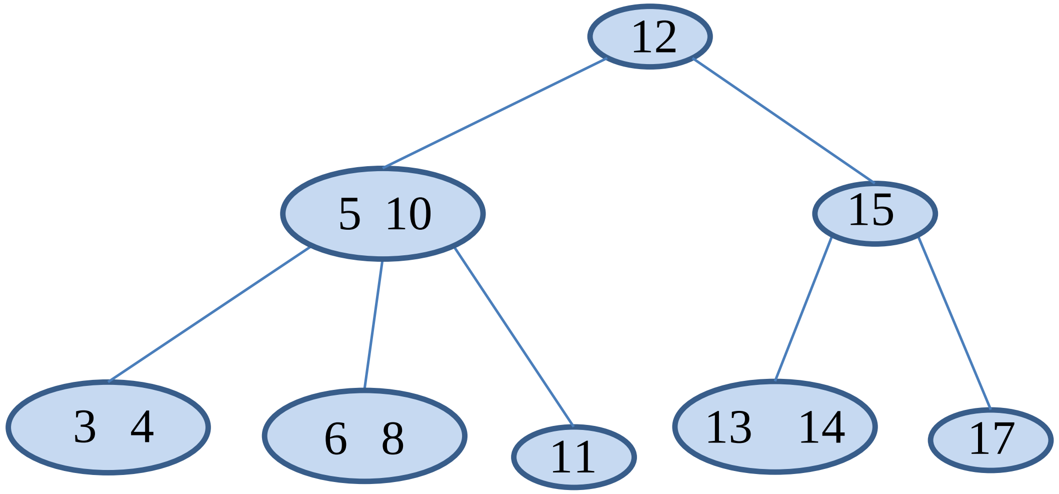
## The root overflows



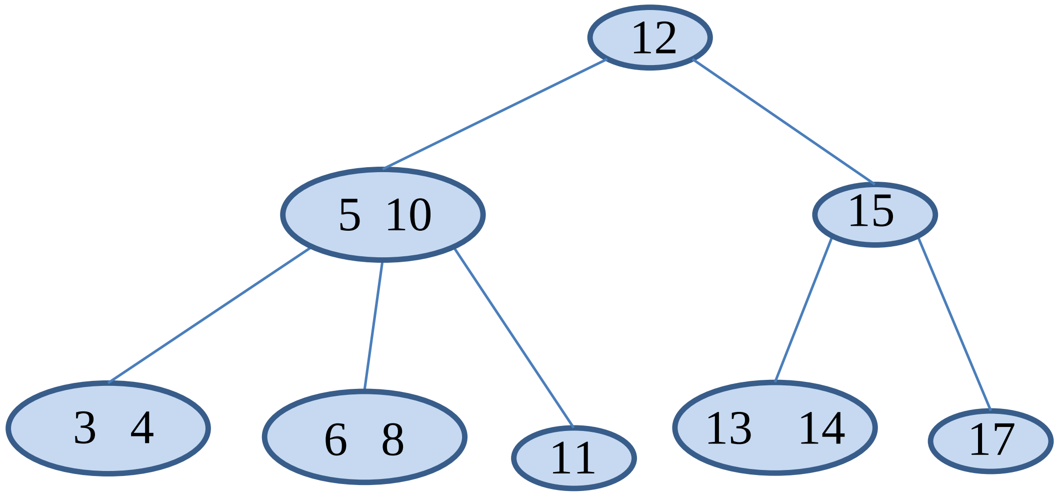
## Creation of new root



## Split



## Final tree



## Complexity

- Same as for 2-3-trees
  - At most $h$ splits
  - Each split is constant time
- $O(\log n)$
  - Because the tree is balanced

## Removal in 2-4 trees

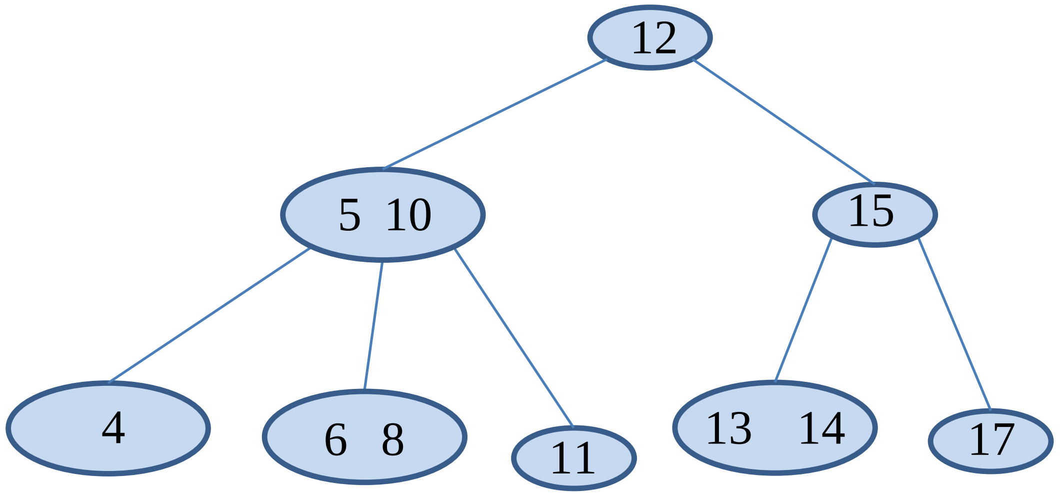
- To remove a value $k_i$ from an **internal** node
  - Replace with its **predecessor** (or its **successor**)
  - Right-most value in the $i$-th subtree
  - Similar to the BST case of nodes with two children
- To remove a value from a **leaf**
  - We simply remove it
  - But it might viotalate the **size** property (**underflow**)

## Fixing underflows

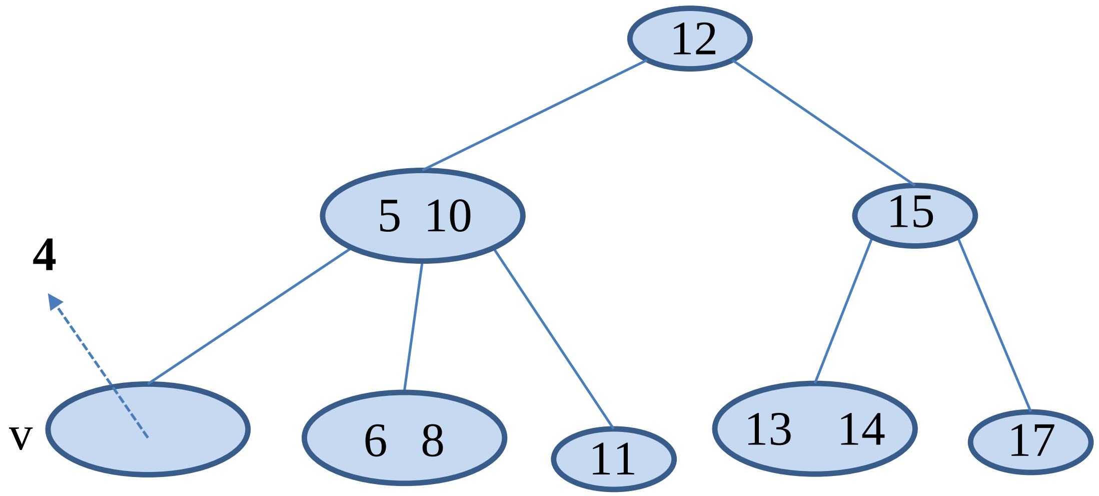
Two strategies for fixing an underlow at $\nu$

- Is there an **immediate sibling** $w$ with a “spare” value? (2 or 3 values)
- If so, we do a **transfer** operation
  - Move a value of $w$ to its parent $u$
  - Move a value of the parent $u$ to $\nu$
- If not, we do a **fusion** operation
  - Merge $\nu$ and $w$, creating a new node $\nu'$
  - Move a value from the parent $u$ to $\nu'$
  - This might **underflow the parent**, continue the same procedure there

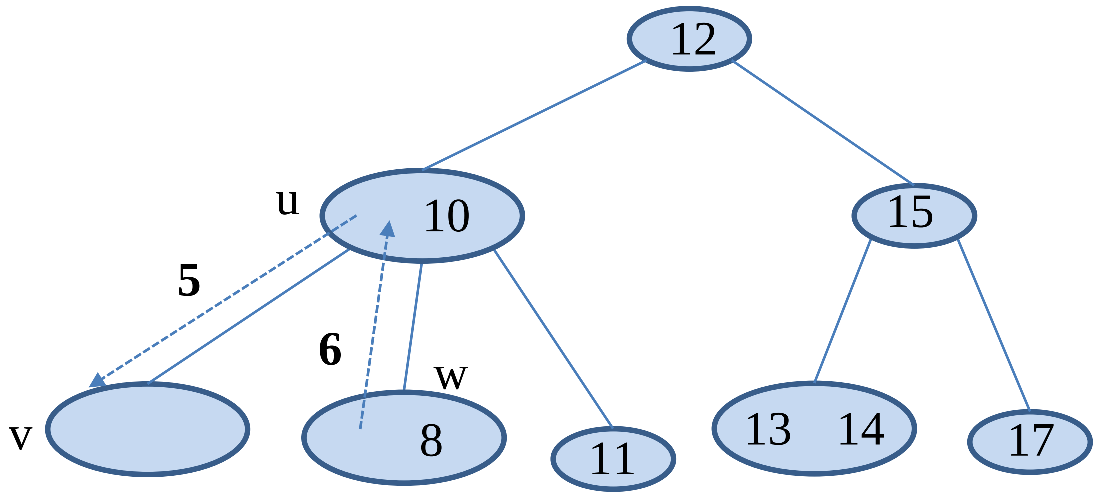
## Initial tree



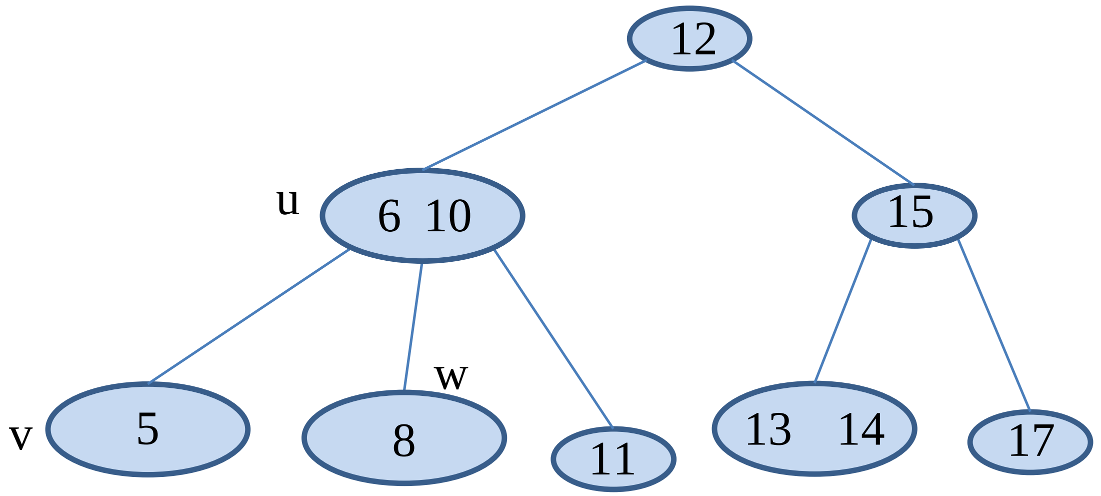
## Remove 4



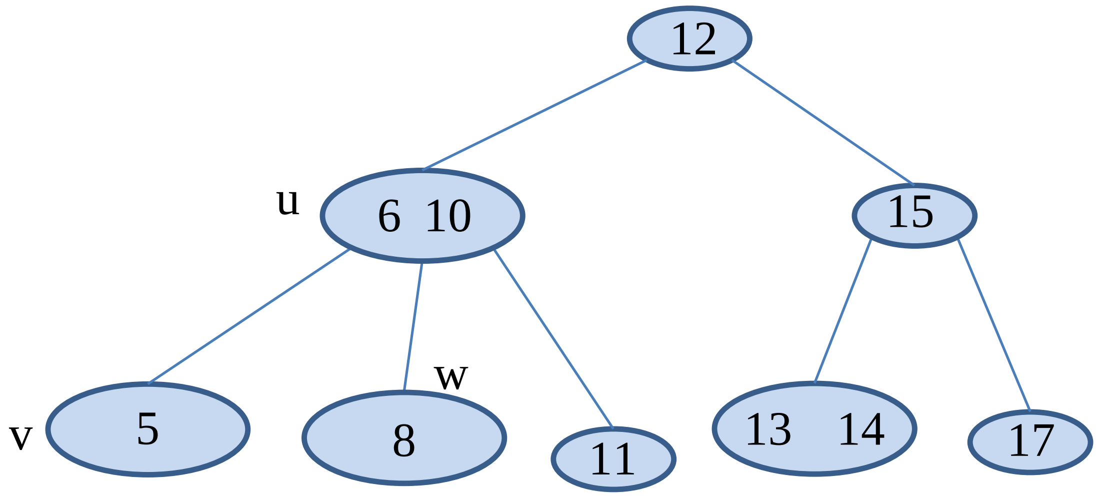
## Transfer



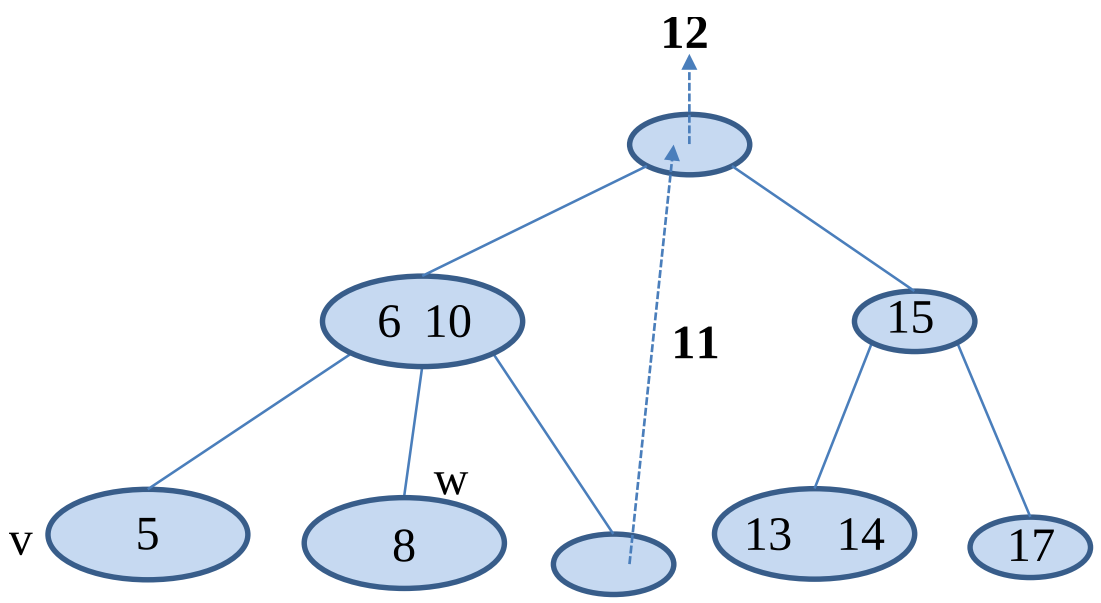
## After the transfer



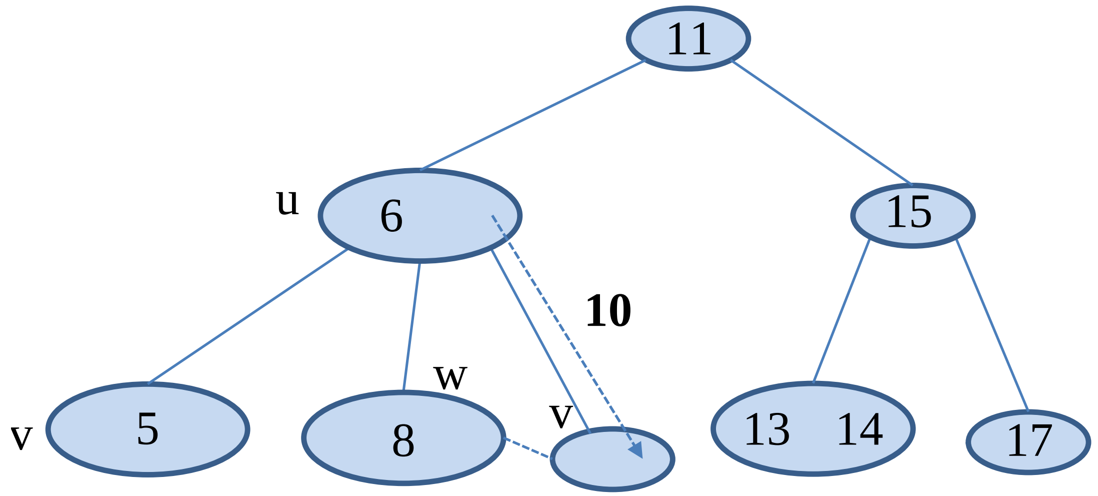
## Remove 12



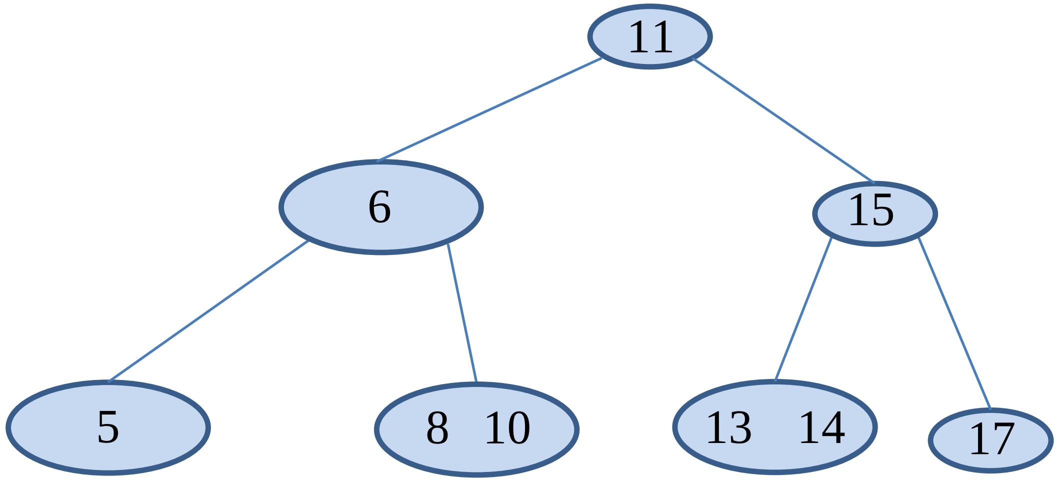
## Remove 12



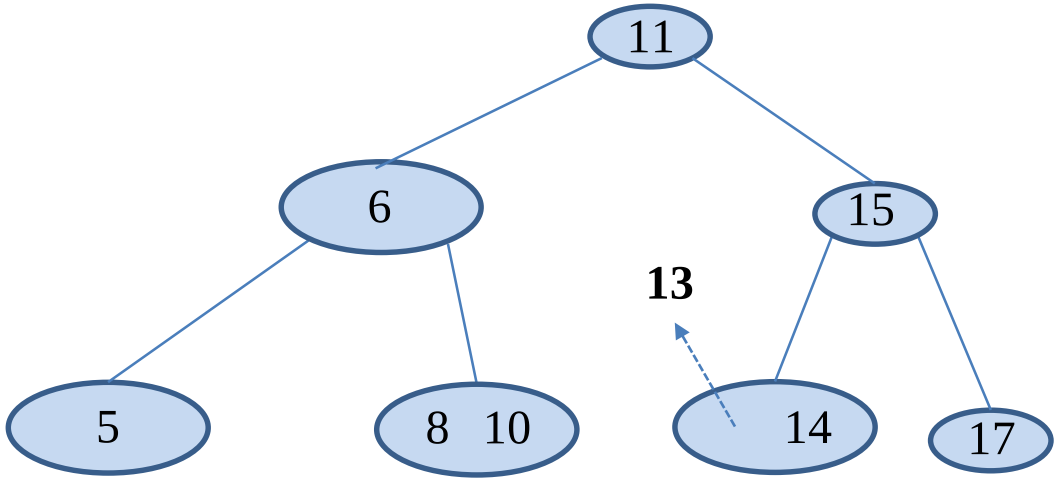
# Fusion of and



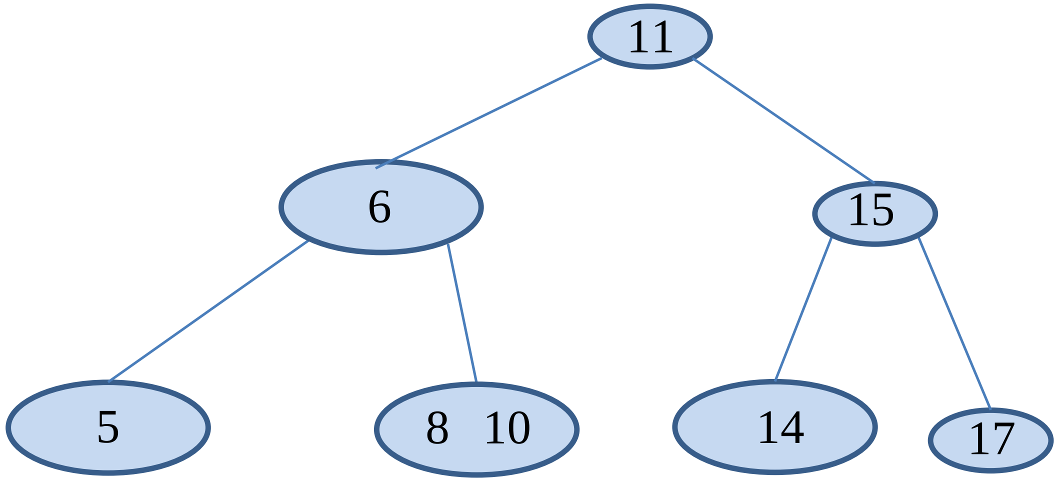
## After the fusion



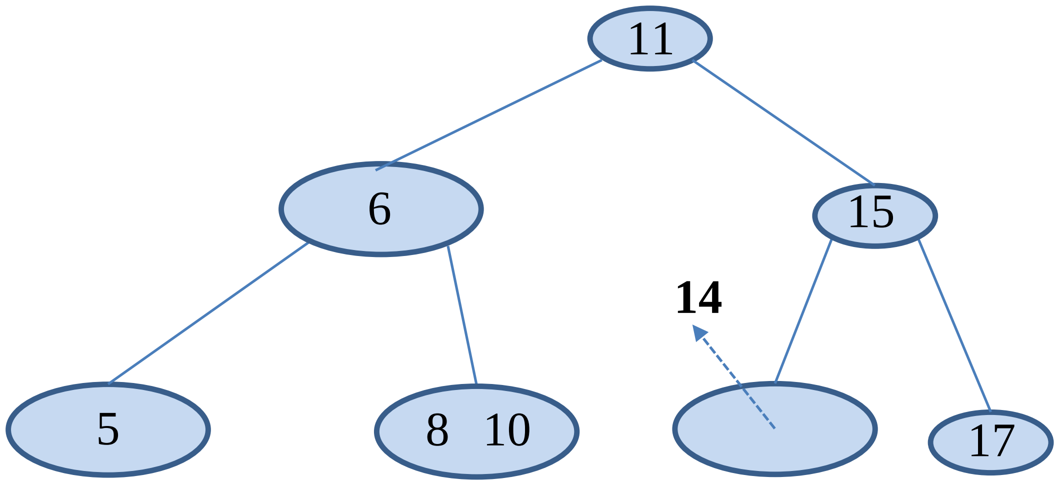
## Remove 13



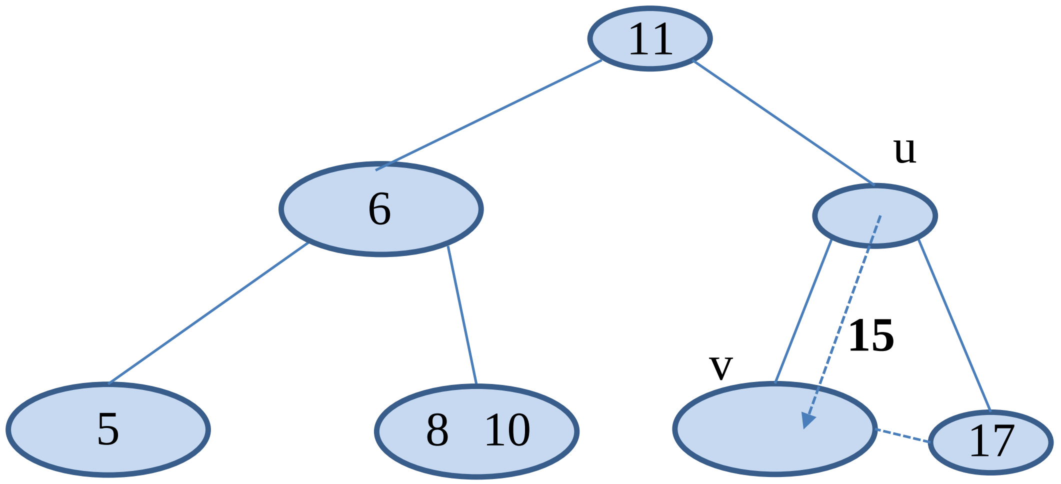
## After the removal of 13



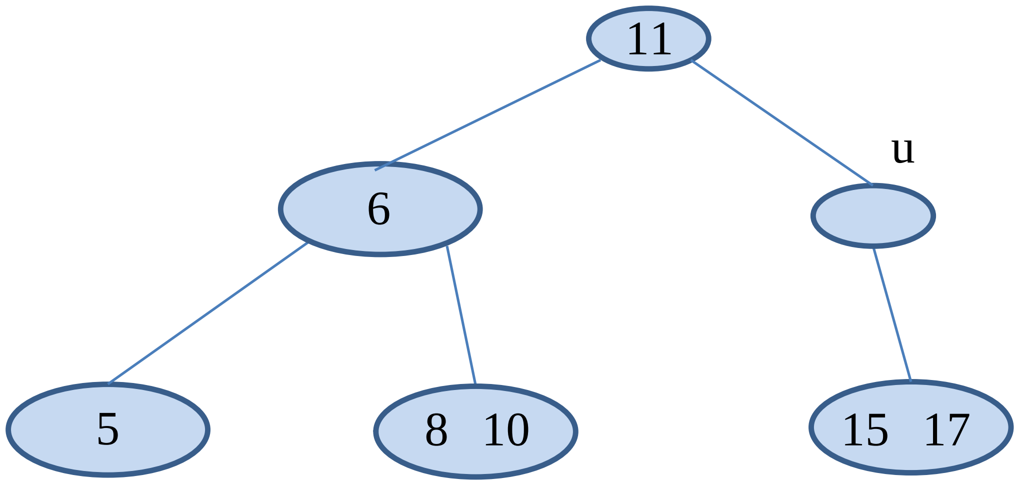
## Remove 14 - underflow



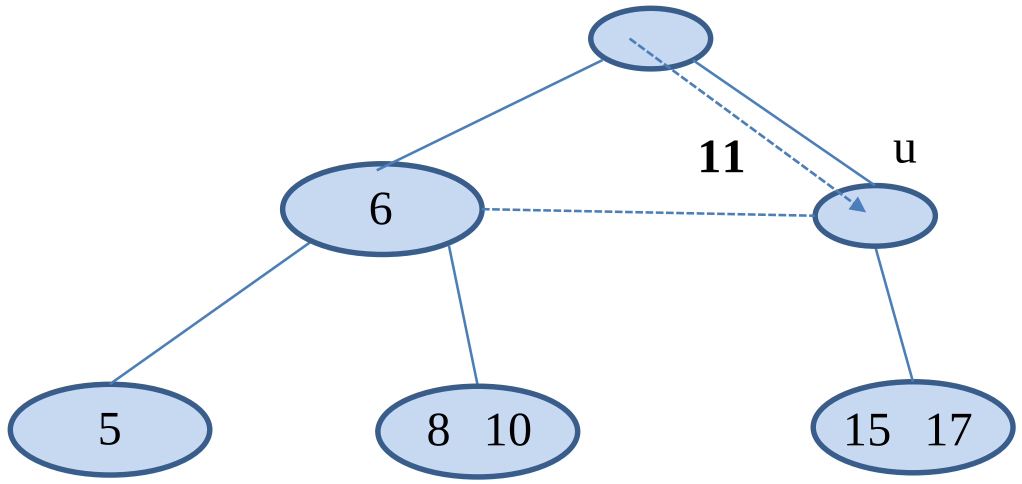
## Fusion



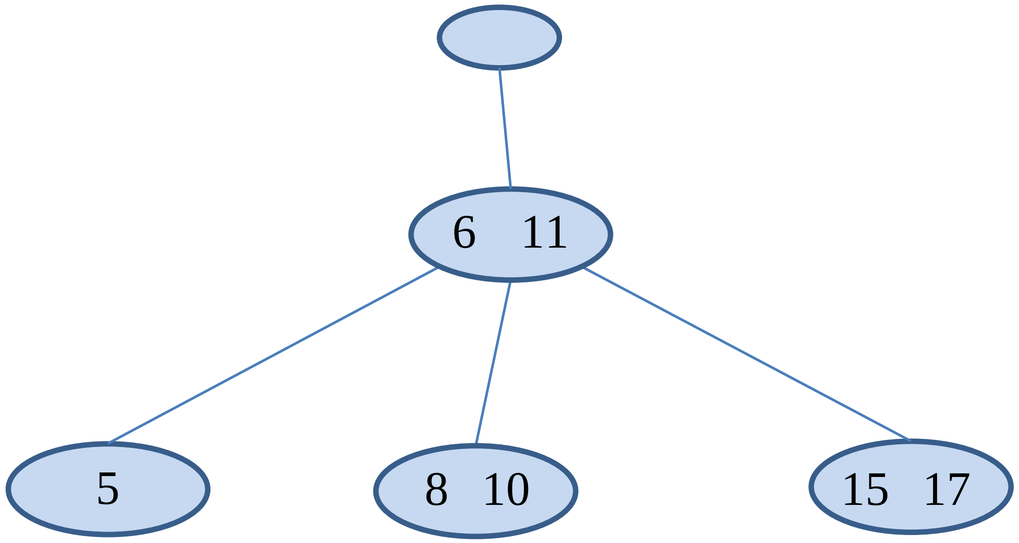
## Underflow at



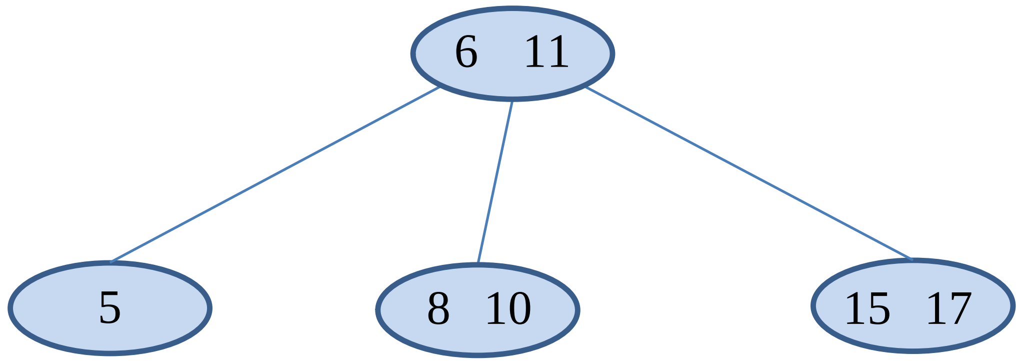
# Fusion



## Remove the root



## Final tree



## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*. Section 9.9
- M. T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++.* Section 10.4
- R. Sedgewick. *Αλγόριθμοι σε C*. 3η Αμερικανική Έκδοση. Εκδόσεις Κλειδάριθμος. Section 13.3

# B-Trees

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Searching on disks

- So far we have assumed that our data are stored in memory
- What about storing data on a **disk**?
  - **Example**: databases
- **Disk access** can be at least 100,000 to 1,000,000 slower
  - Goal: **minimize disk accesses**
- Also: data is read in **blocks**
  - Eg 512 or 1024 bytes
  - Reading 1 byte is the same as **reading a whole block**

## B-Trees

We can easily generalize 2-3 / 2-4-trees:

- A **B-tree** of order $m$ is an $m$-way search tree with 2 extra properties
- **Size property**
  - Any node contains between $\lceil \frac{m}{2} \rceil - 1$ **and** $m - 1$ **values**
  - **Internal** nodes with $n$ values have exactly $n + 1$ **children**
  - The **root is excluded** from this rule
- **Depth property**
  - All **leaves** have the **same depth** (lie on the same level)

## B-Trees

- We select $m$ so that the **whole B-tree node** fits in a block
  - We read “multiple nodes” for the “price” of one
  - Fewer disk accesses than reading multiple nodes of a BST / AVL / …
- Such trees are **balanced**
  - $h = O(\log n)$
  - Proof: exercise
- 2-3 and 2-4-trees are B-trees of order 3 and 4
- Generalize even more: require between $a$ and $b$ children
  - for some $2 \leq a \leq \frac{(b+1)}{2}$

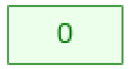
## ADT Set using BTree, types

```c
// Κόμβος του set, περιέχει μία μόνο τιμή. Κάθε btree_node έχει πολλά
struct set_node {
    Pointer value;          // Η τιμή του κόμβου.
    BTreeNode owner;        // Ο btree_node στον οποίο ανήκει αυτό το
};

// Το struct btree_node είναι ο κόμβος ενός Β-Δέντρου.
struct btree_node {
    int count;                              // Αριθμός στοιχείων
    BTreeNode parent;                       // Πατέρας
    BTreeNode children[MAX_CHILDREN + 1];   // Παιδιά
    SetNode set_nodes[MAX_VALUES + 1];      // Δεδομένα (μέσα σε set
};

// Υλοποιούμε τον ADT Set μέσω B-Tree, οπότε το struct set είναι ένα
struct set {
    BTreeNode root;             // Η ρίζα του δέντρου
    int size;                   // Μέγεθος, για αποδοτικό set_size
    CompareFunc compare;        // Διάταξη
    DestroyFunc destroy_value;  // Συνάρτηση που καταστρέφει ένα στοι
};
```

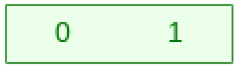
# Insertion in a B-tree

- Same as for 2-3 and 2-3-4-trees
  - Search for the value
  - Insert at a leaf
- In case of an overflow ($m + 1$ children)
  - Split it into two nodes of $m/2$ children each
  - Move the separator value (median) to the parent

## Insert example, $m = 5$

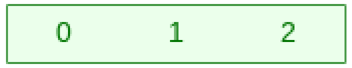


## Insert example, $m = 5$



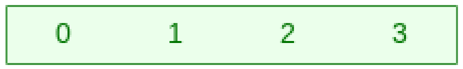
Inserting 1

## Insert example, $m = 5$



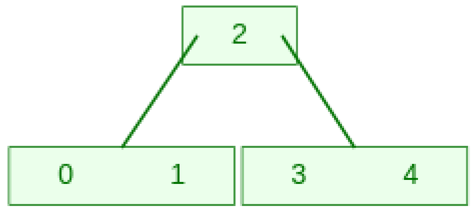
Inserting 2

## Insert example, $m = 5$



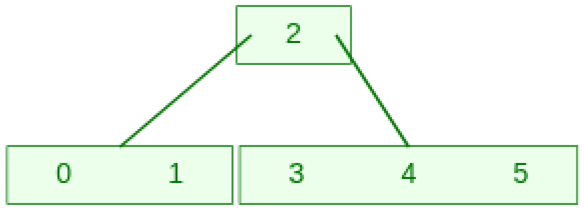
Inserting 3

# Insert example, $m = 5$



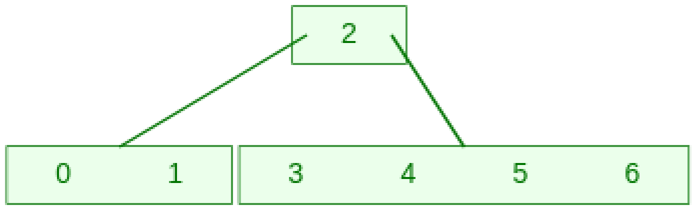
Inserting 4: overflow, 2 moves to a new root

## Insert example, $m = 5$



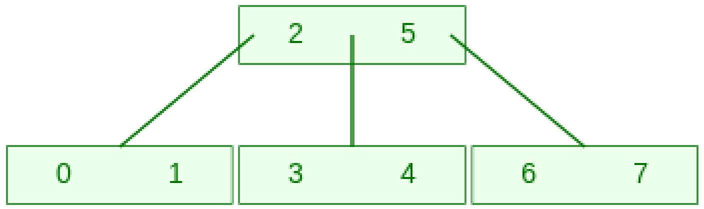
Inserting 5

## Insert example, $m = 5$



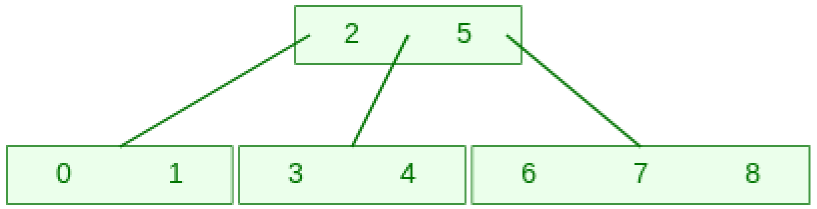
Inserting 6

## Insert example, $m = 5$



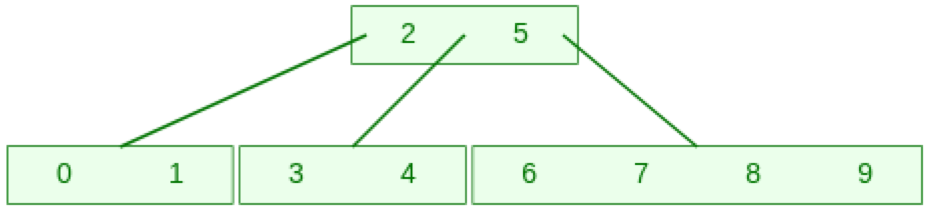
Inserting 7: overflow, 5 moves up

## Insert example, $m = 5$



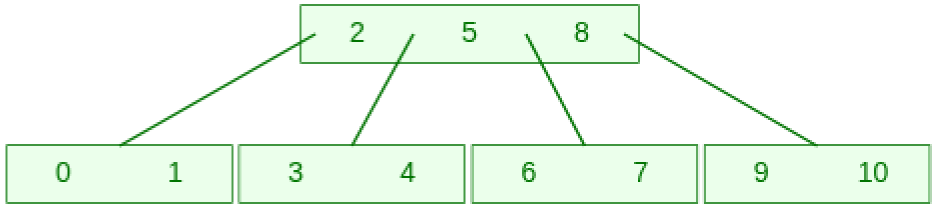
Inserting 8

## Insert example, $m = 5$



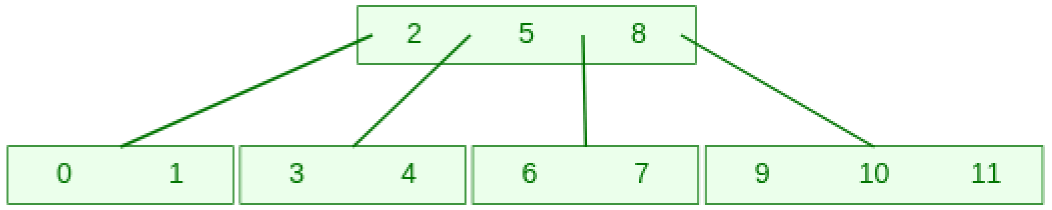
Inserting 9

## Insert example, $m = 5$



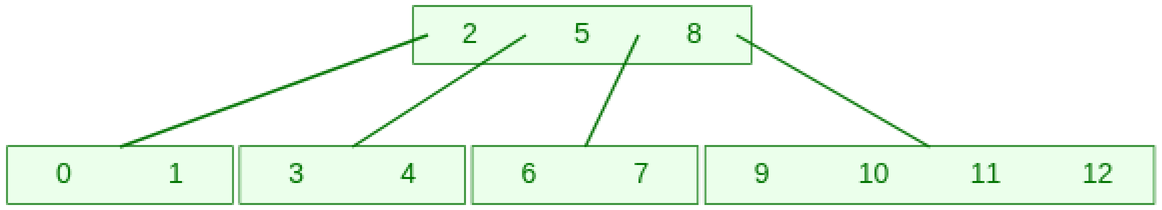
Inserting 10: overflow, 8 moves up

## Insert example, $m = 5$



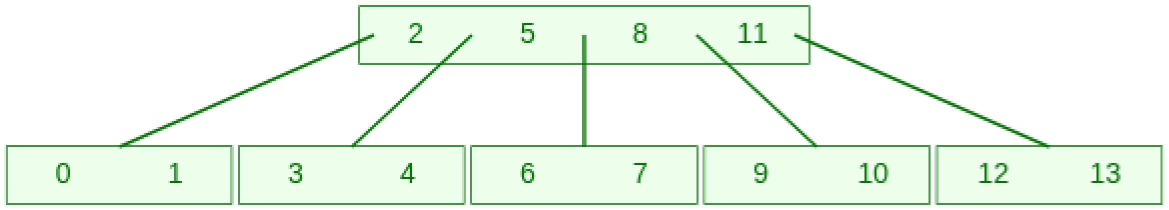
Inserting 11

## Insert example, $m = 5$



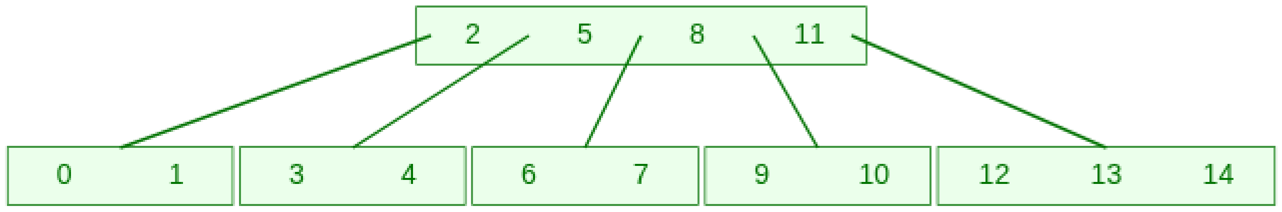
Inserting 12

## Insert example, $m = 5$



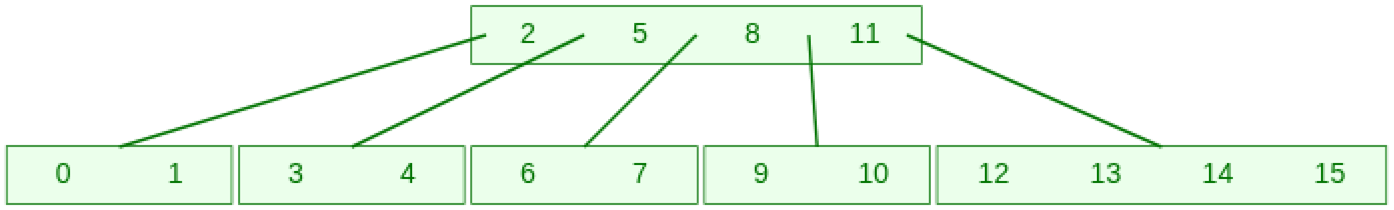
Inserting 13: overflow, 11 moves up

# Insert example, $m = 5$



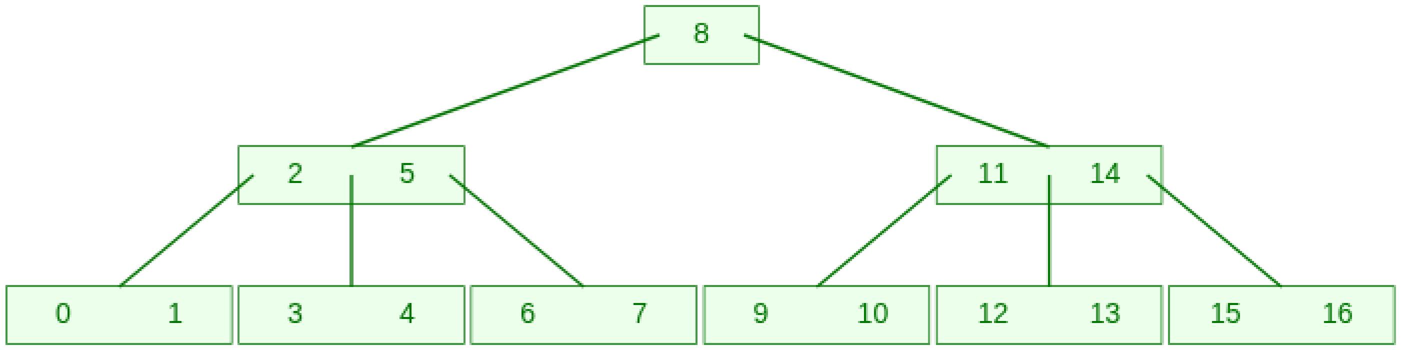
Inserting 14

## Insert example, $m = 5$



Inserting 15

## Insert example, $m = 5$



Inserting 16: overflow, 14 moves up, creating a new overflow

## Code

- As always, the code is in `lecture-code`
  - `modules/UsingBTree/ADTSet.c`
- We only highlight some parts here

## Code, `node_insert`

```
BTreeNode node_insert(BTreeNode root, CompareFunc compare, Pointer va
    // [απλός κώδικας για την περίπτωση κενού δέντρου]

    // Εύρεση του κόμβου στον οποίο πρέπει να γίνει insert
    int index;
    BTreeNode node = node_find(root, compare, value, &index);
    if (index != -1) {          // Υπάρχει ήδη η τιμή
        node->set_nodes[index]->value = value;
        return root;
    }

    // Εύρεση της θέσης που πρέπει να μπει το value
    for (index = 0;
         index < node->count && compare(value, node->set_nodes[index]
         index++)
        ;

    node_add_value(node, set_node_create(value), index);

    if (node->count > MAX_VALUES) // overflow
        split(node, compare);

    // Επιστρέφουμε τη ρίζα, μπορεί να έχει δημιουργηθεί νέα
    return root->parent != NULL ? root->parent : root;
}
```

## Code, `split`

```
// Καλείται όταν ο κόμβος node έχει υπερχειλήσει, τον χωρίζει σε 2 κό
// Στέλνει τη μεσαία από τις τιμές του κόμβου node στον πατέρα του.

static void split(BTreeNode node, CompareFunc compare) {
    // Χωρίζουμε τον κόμβο node σε 2 κόμβους
    BTreeNode right = node_create();
    right->parent = node->parent;     // Οι 2 κόμβοι έχουν τον ίδιο π

    // Μετακίνησε τις μισές τιμές και παιδιά από τον αριστερό κόμβο σ
    int half = node->count/2;
    if (!is_leaf(node))
        for (int i = 0; i <= half; i++)
            node_add_child(right, node->children[i + half + 1], i);

    for (int i = 0; i < half; i++) {
        node_add_value(right, node->set_nodes[i + half + 1], i);
        node->count--;
    }

    // Αφαίρεση μεσαίας τιμής
    SetNode median = node->set_nodes[node->count-1];
    node->count--;

    ...
```

## Code, split

```
    // Προσθέρουμε το median στον πατέρα του κόμβου node.
    BTreeNode parent = node->parent;
    if (parent == NULL) {                           // Ο node είναι η ρίζ
        BTreeNode new_root = node_create();         // Δημιούργησε καινού

        node_add_value(new_root, median, 0);

        right->parent = node->parent = new_root;
        new_root->children[0] = node;
        new_root->children[1] = right;

    } else {
        int index;  // Βρες τη θέση εισαγωγής της τιμής στον πατέρα.
        for (index = 0; index < parent->count; index++)
            if (compare(median->value, parent->set_nodes[index]->valu
                break;

        // Πρόσθεσε τον right ως δεξιό παιδί της (νέας) διαχωριστικής
        node_add_child(parent, right, index+1);
        node_add_value(parent, median, index);

        if (parent->count > MAX_VALUES)  // parent overflows
            split(parent, compare);
    }
}
```

## Removal from B-trees

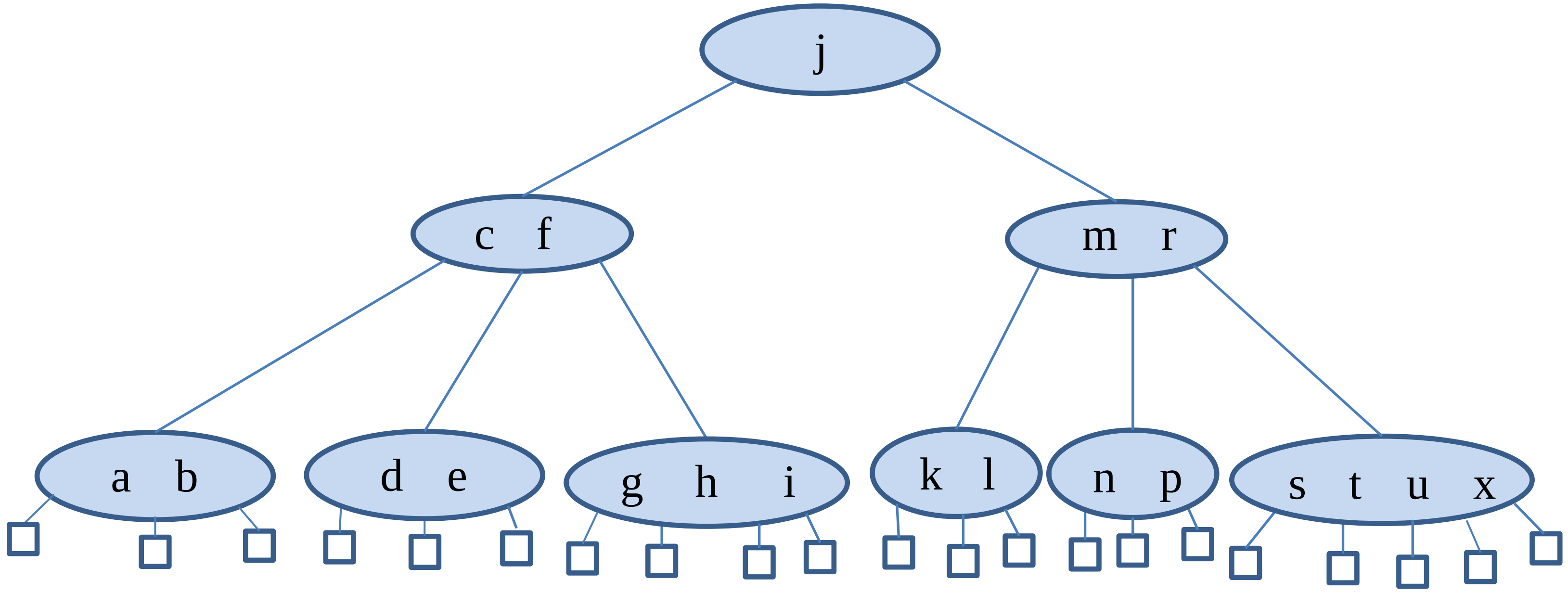
- Same as for 2-3 and 2-3-4-trees
- To remove a value $k_i$ from an **internal** node
  - Replace with its **predecessor** (or its **successor**)
  - Right-most value in the $i$-th subtree
- To remove a value from a **leaf**
  - We simply remove it
  - But it might viotalate the **size** property (**underflow**)

# Fixing underflows

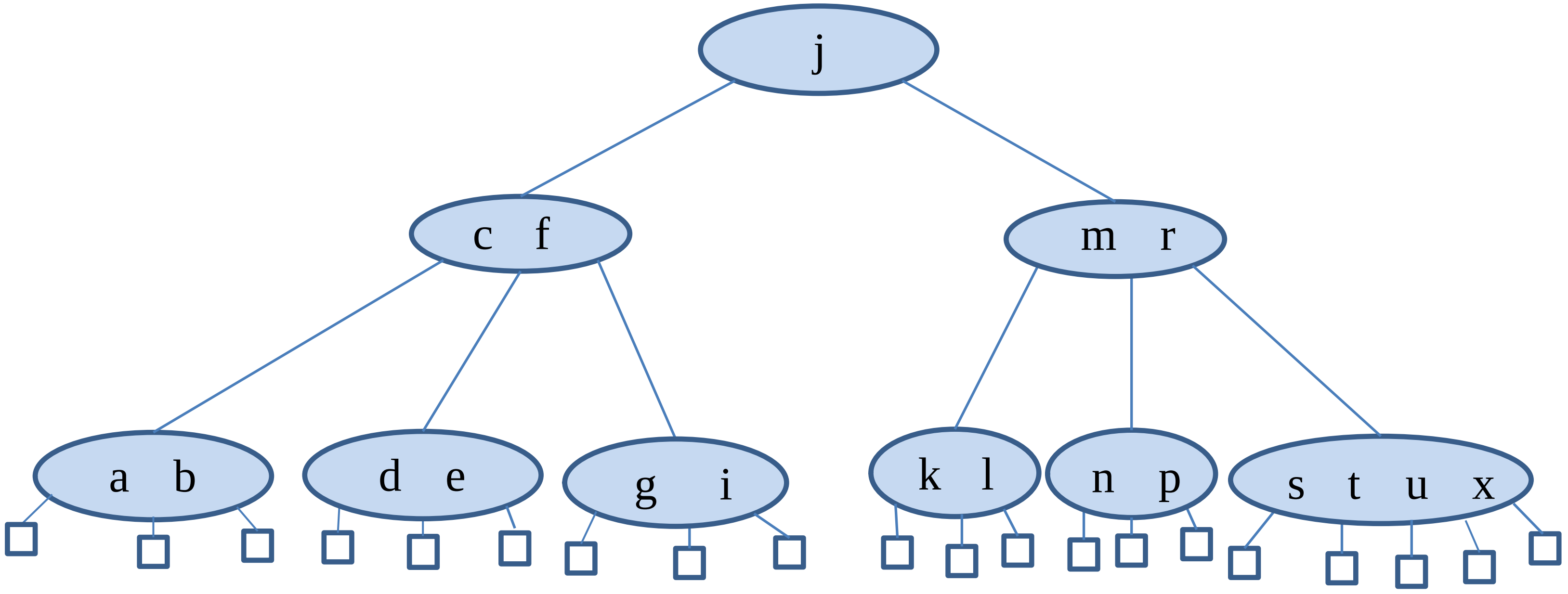
Two strategies for fixing an underlow at $\nu$

- Is there an **immediate sibling** $w$ with a “spare” value?
- If so, we do a **transfer** operation
  - Move a value of $w$ to its parent $u$
  - Move a value of the parent $u$ to $\nu$
- If not, we do a **fusion** operation
  - Merge $\nu$ and $w$, creating a new node $\nu'$
  - Move a value from the parent $u$ to $\nu'$
  - This might **underflow the parent**, continue the same procedure there

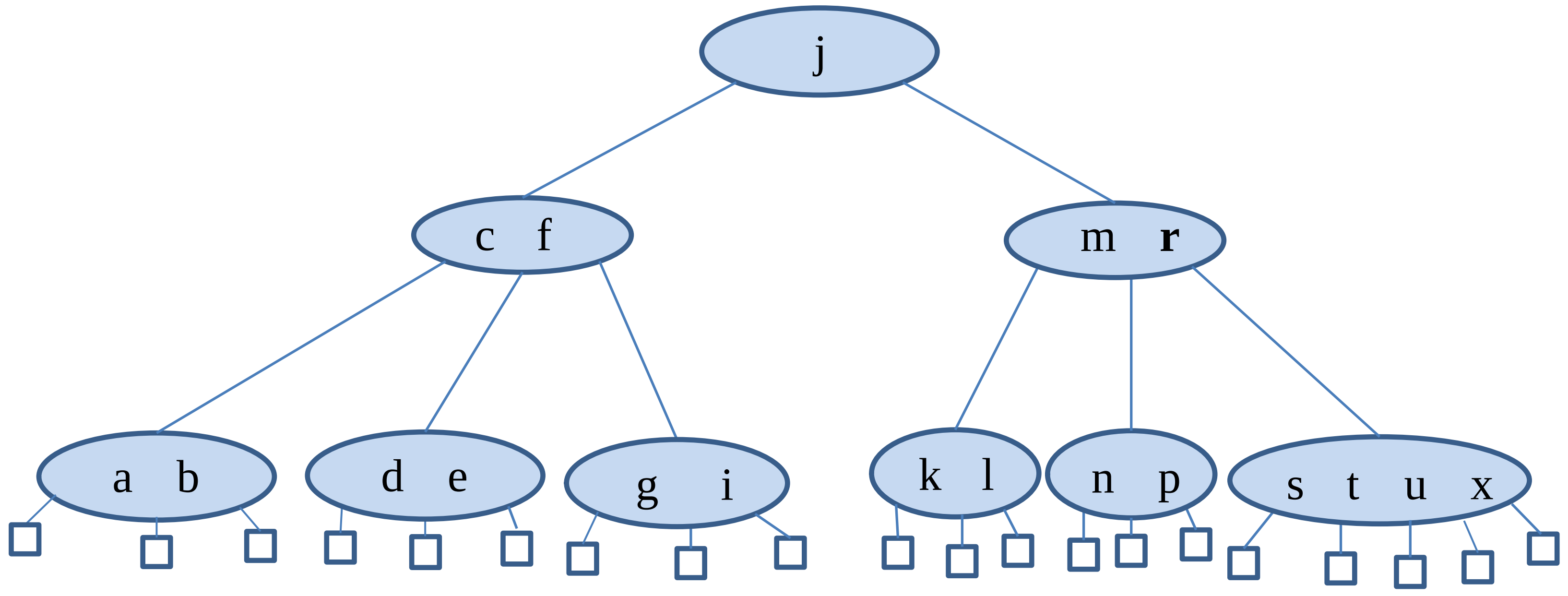
## Example



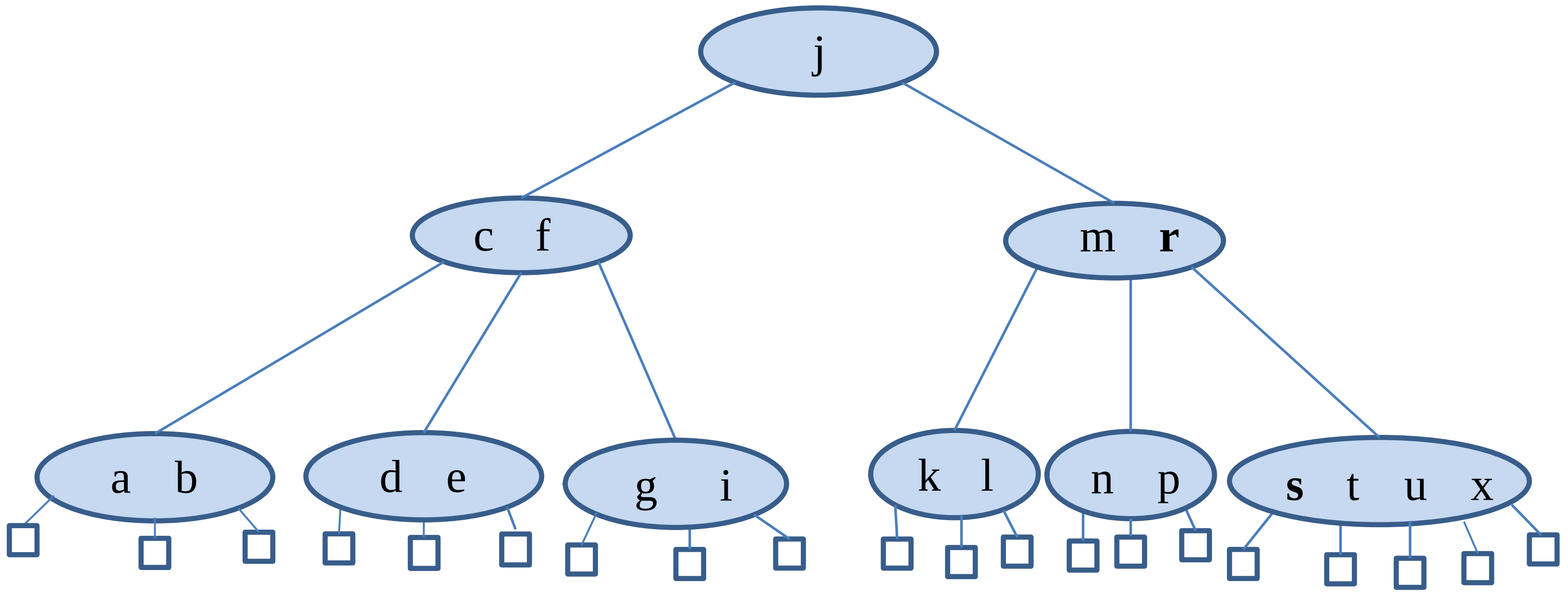
## Delete h



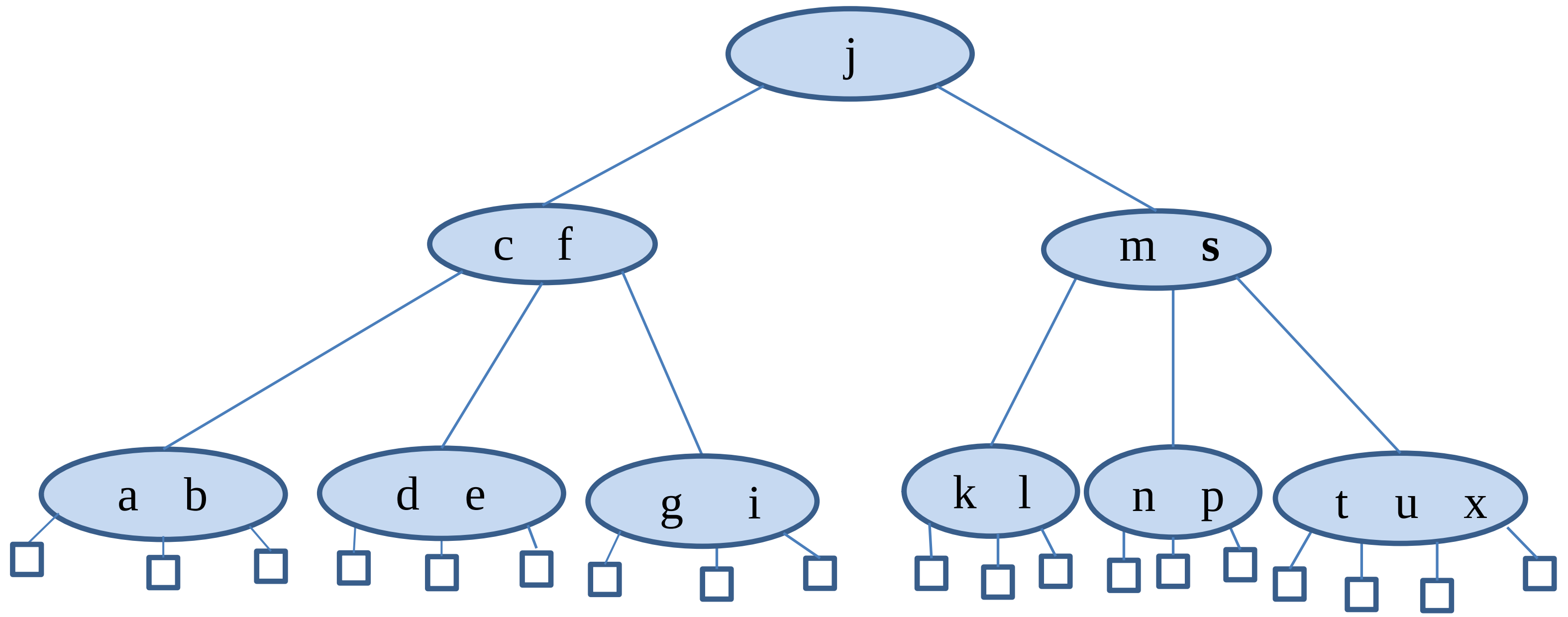
## Delete r



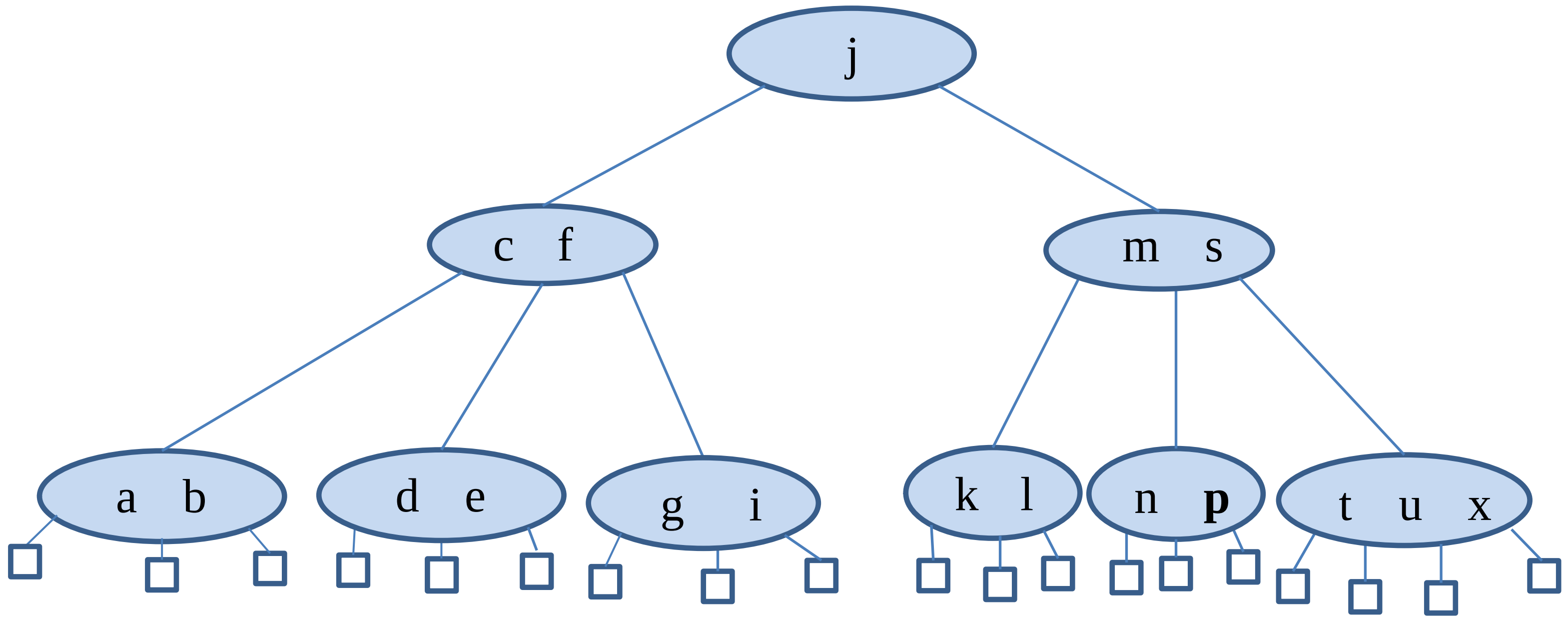
## Find the Successor of r



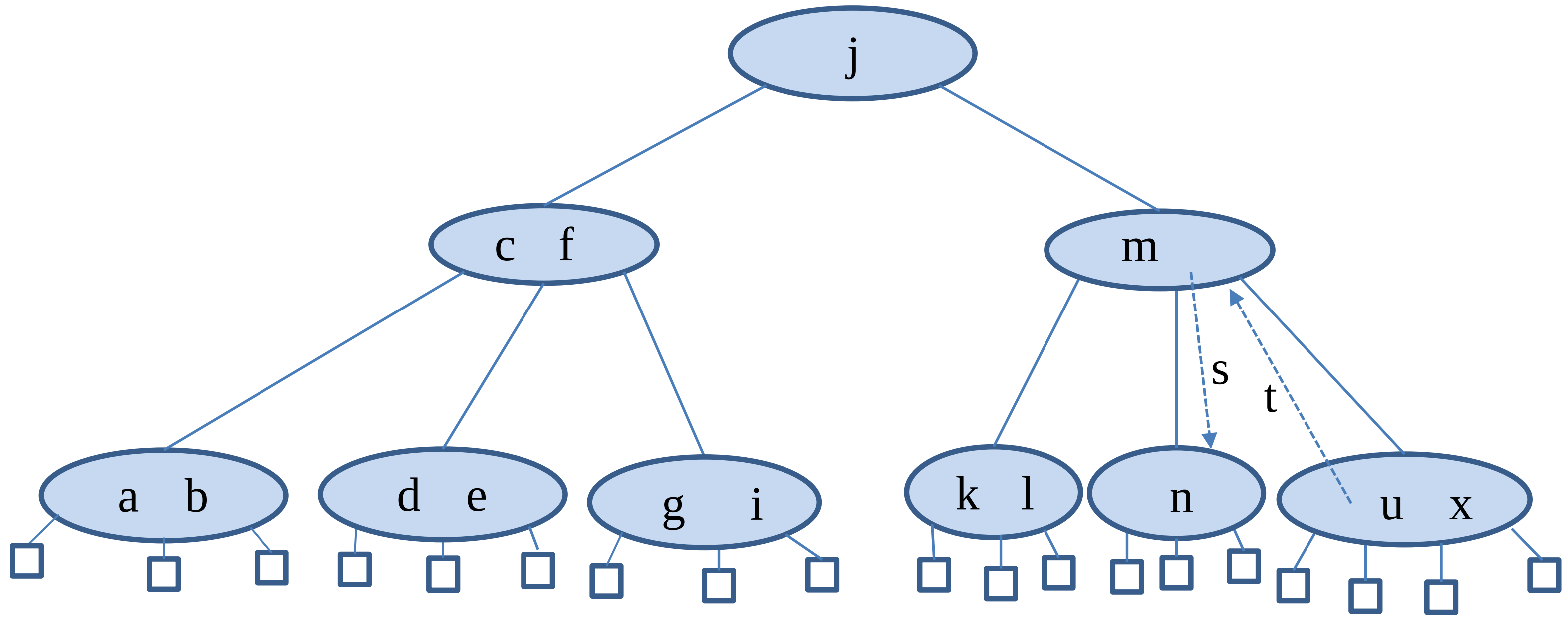
## Promote the Successor of r – Delete the Successor from its Place



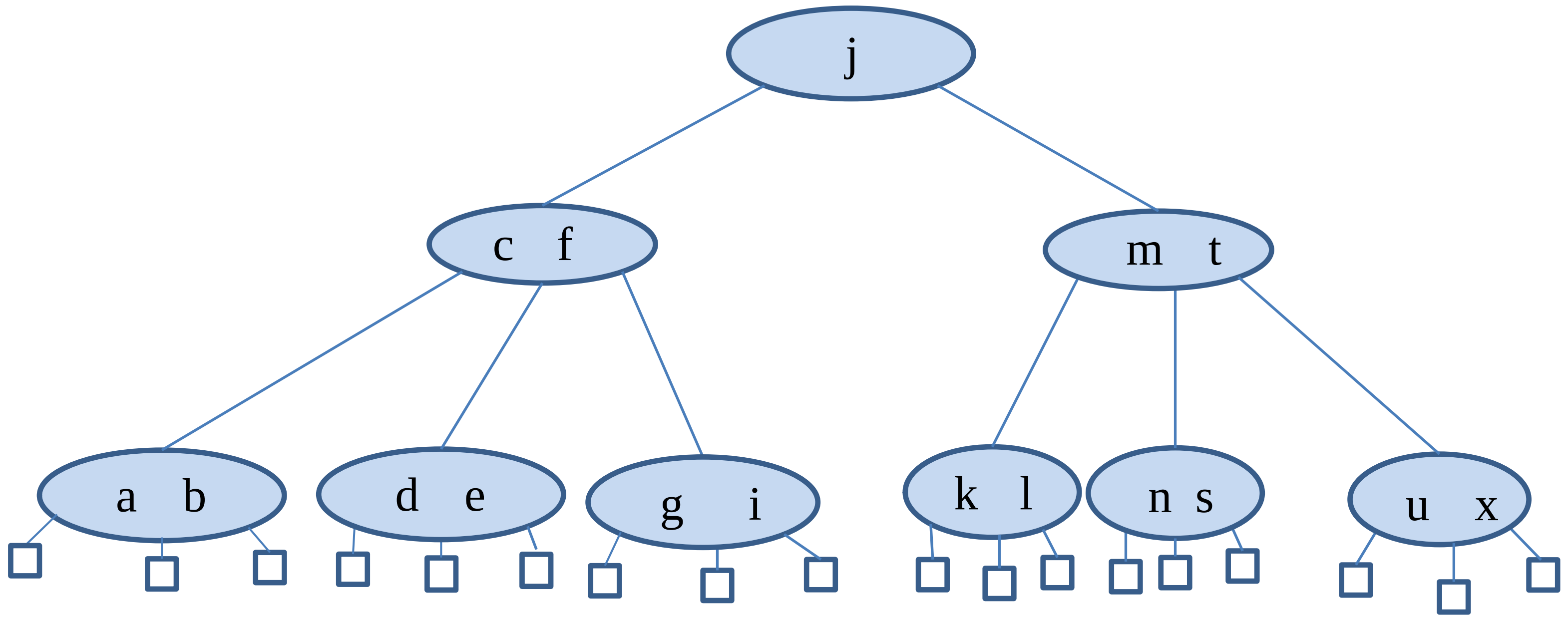
## Delete p



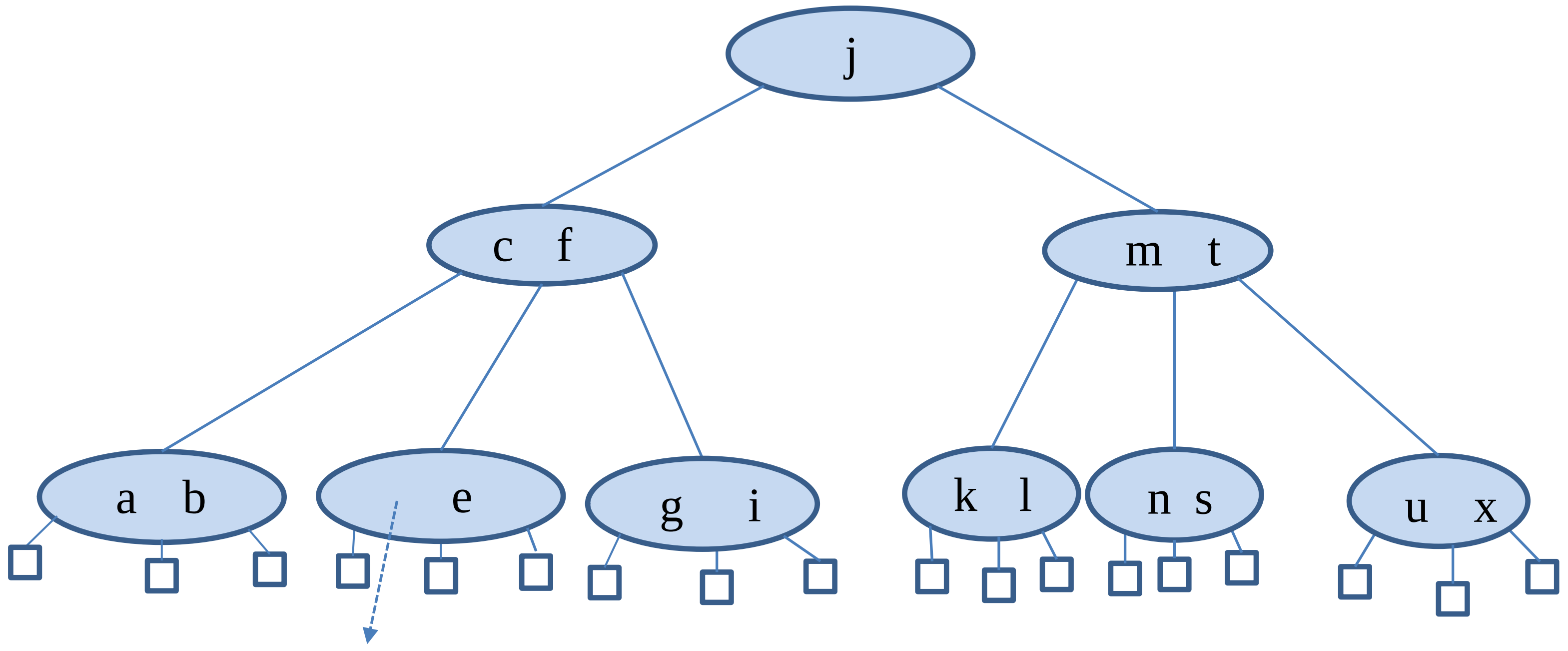
## Transfer



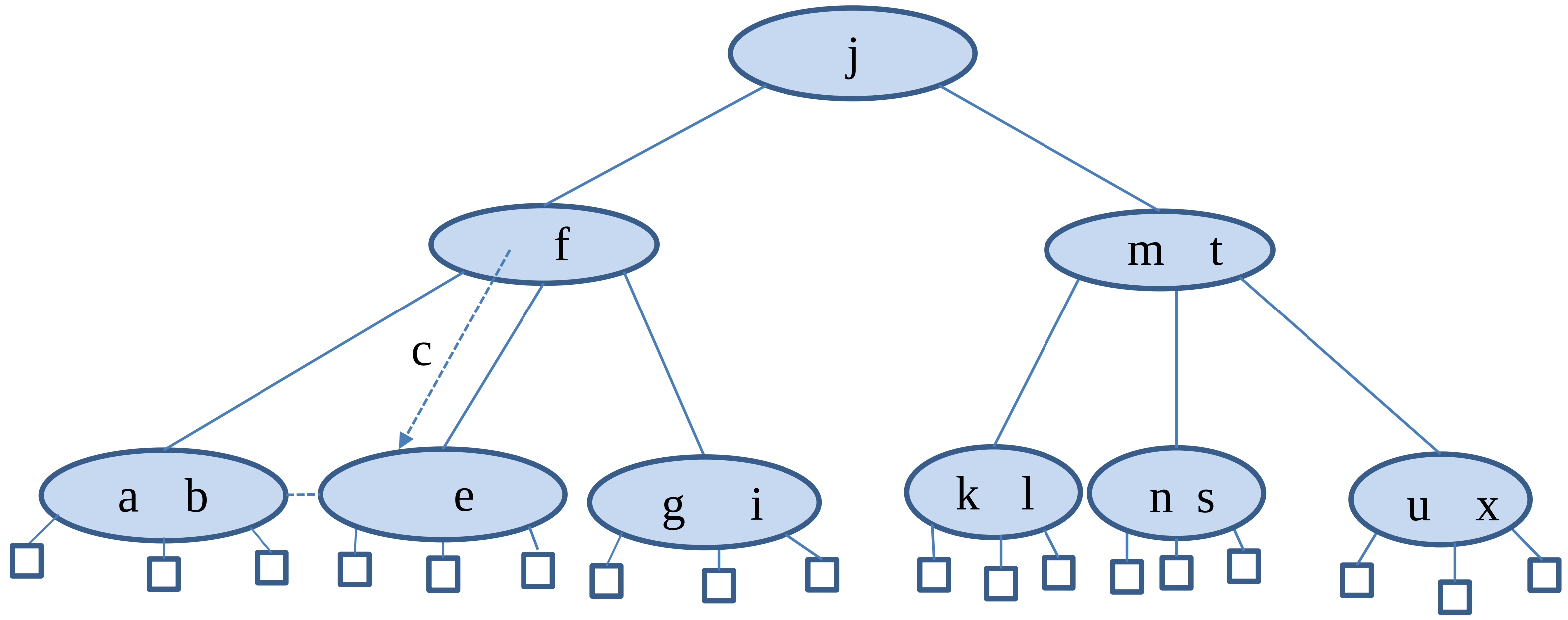
## After the Transfer



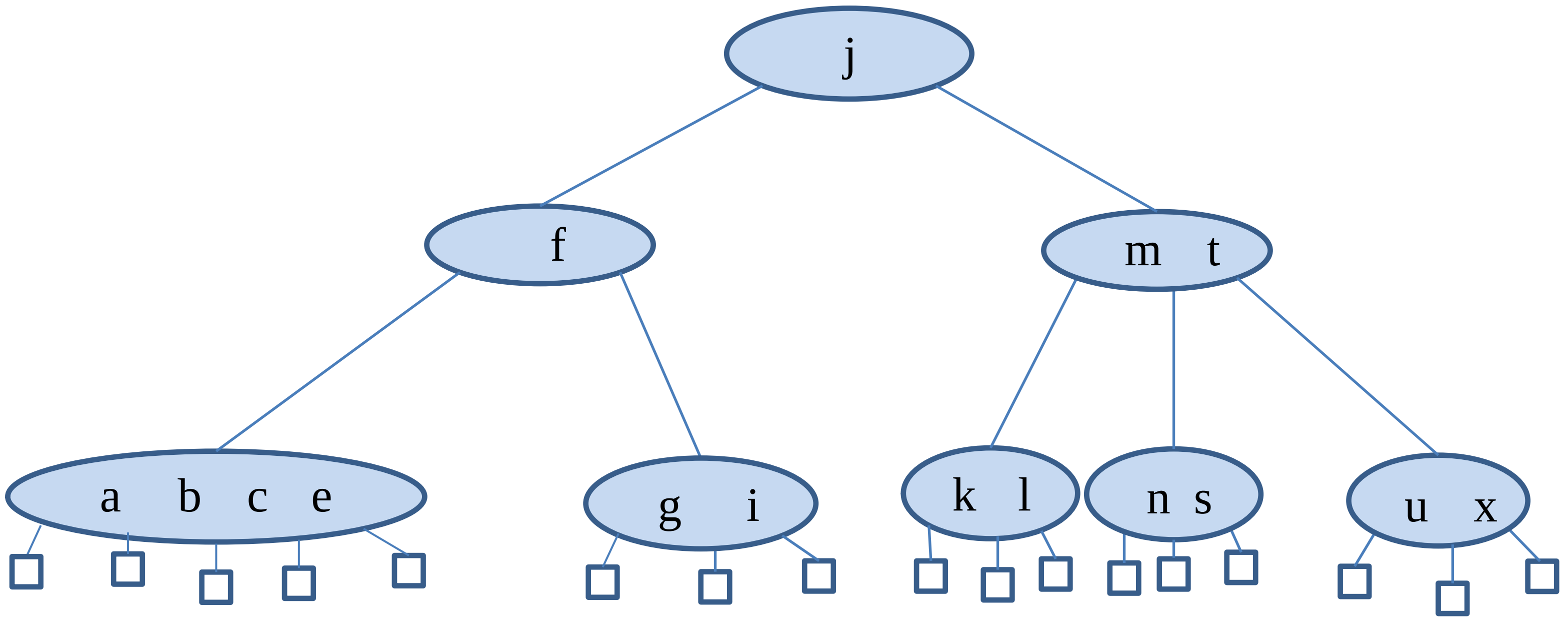
## Delete d



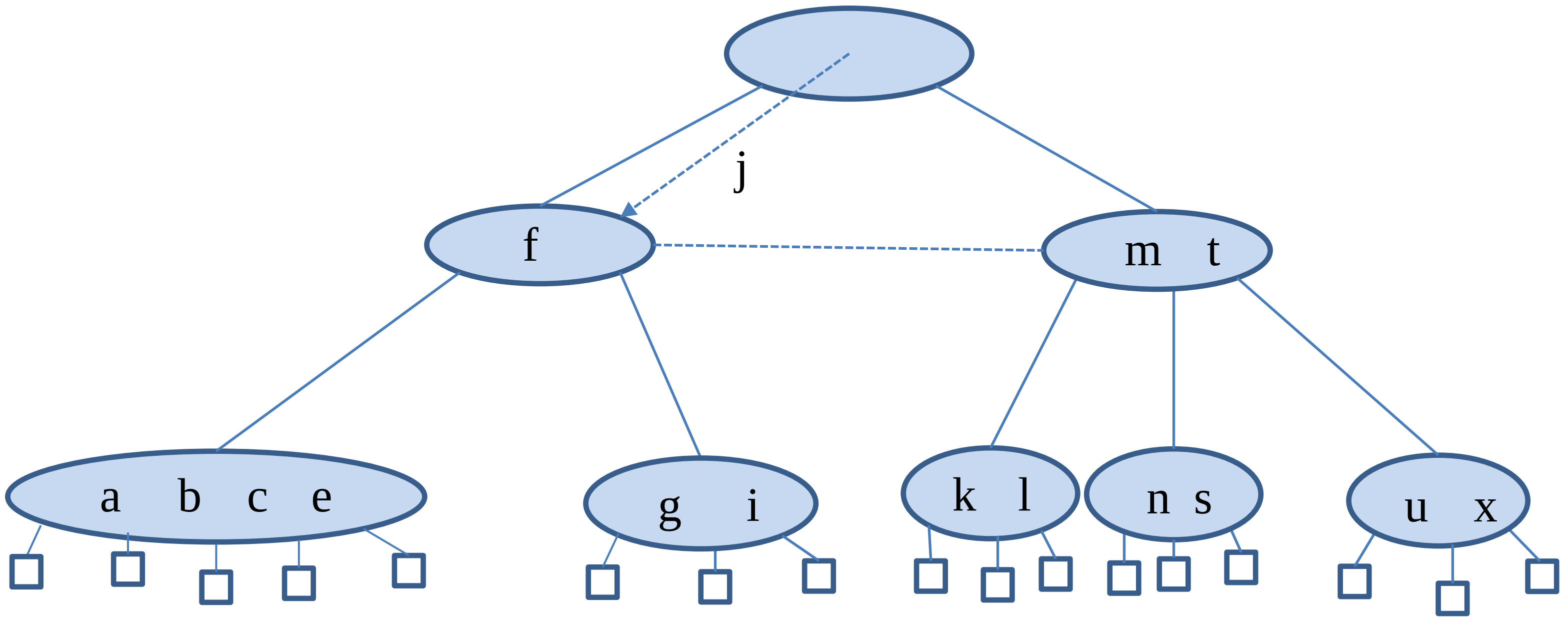
## Fusion



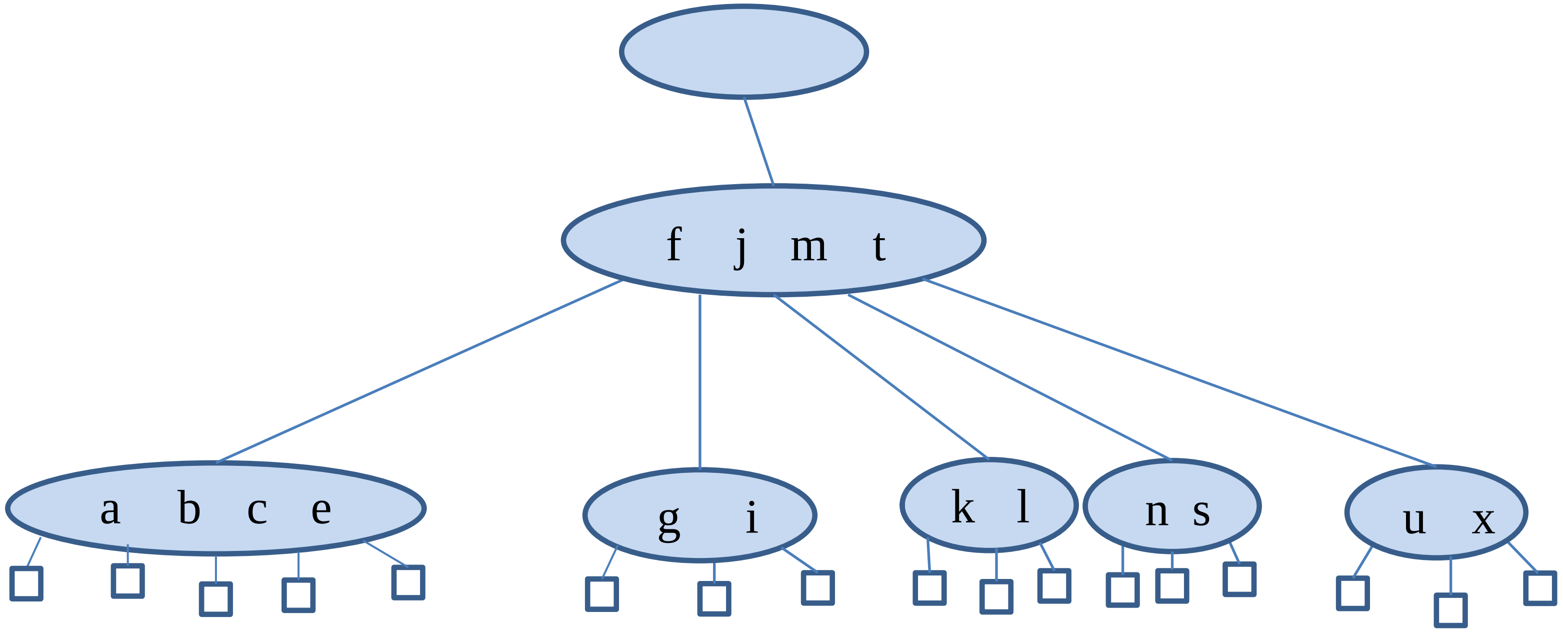
## After the Fusion – Underflow at f



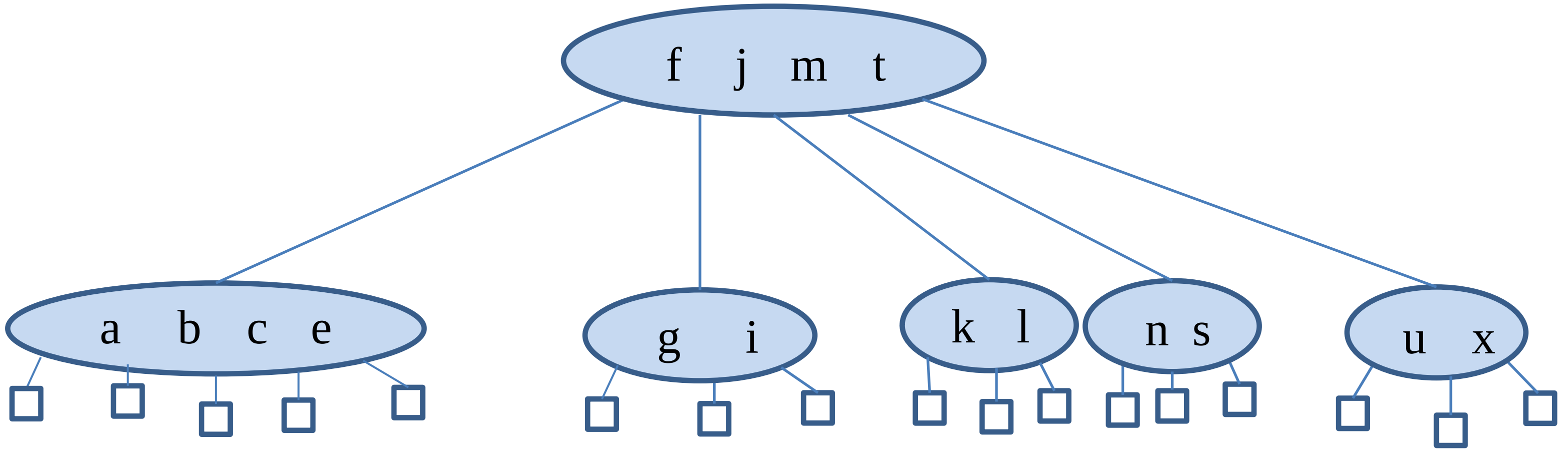
## Fusion



## After the Fusion – Delete Root



## Final Tree



## Code, `node_remove`

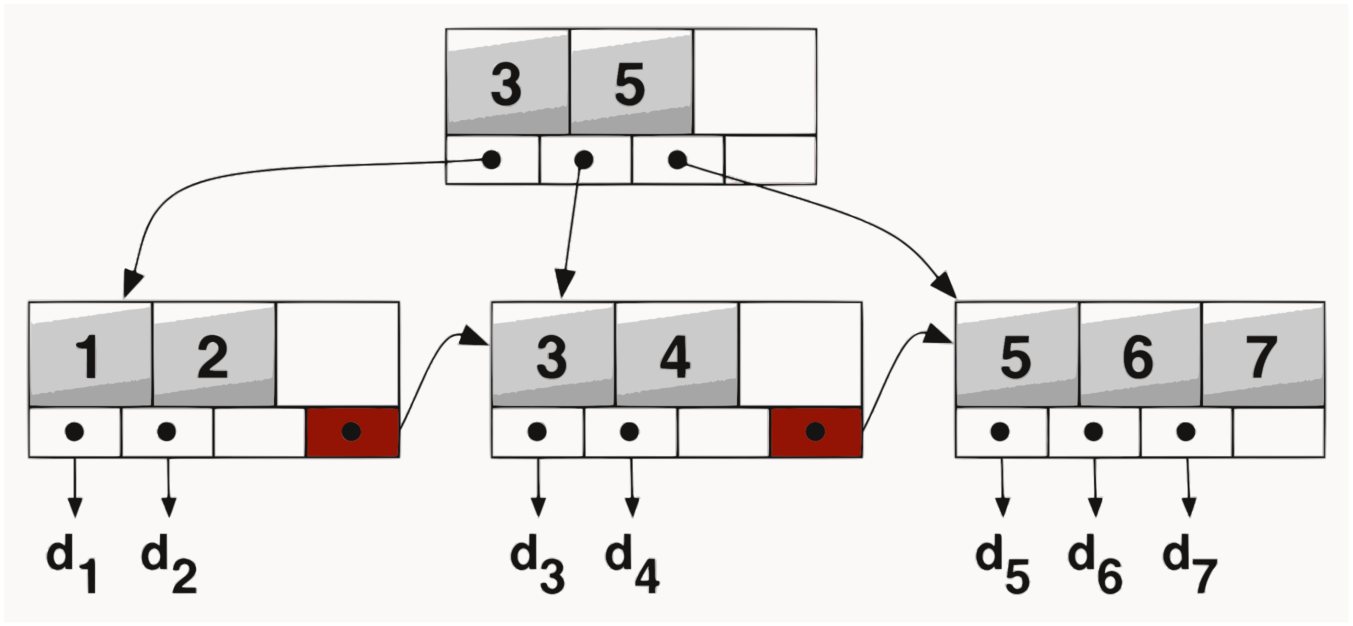
```
BTreeNode node_remove(BTreeNode root, CompareFunc compare, Pointer va
    // Βρες τον κόμβο που περιέχει την τιμή.
    int index;
    BTreeNode node = node_find(root, compare, value, &index);

    if (index == -1)   // Η τιμή δεν υπάρχει στο δέντρο.
        return root;

    // Βρέθηκε ισοδύναμη τιμή στον node, οπότε τον διαγράφουμε
    // Το πώς θα γίνει αυτό εξαρτάται από το αν έχει παιδιά.

    if (is_leaf(node)) {
        // Φύλλο: διάγραψε την τιμή, αναδιάταξε τα δεδομένα, repair
        // Ολίσθησε όλα τα δεδομένα 1 θέση αριστερά.
        for (int i = index; i < node->count-1; i++)
            node->set_nodes[i] = node->set_nodes[i + 1];

        node->count--;           // Αφαίρεσε το δεδομένο.

        repair_underflow(node);  // Αναδιαμόρφωσε το δένδρο.
```

## Code, node_remove

```
    } else {
        // Άν είναι εσωτερικός κόμβος αντικατάσταση με την next τιμε
        // και remove της τιμής αυτής
        SetNode max = node_find_max(node->children[index]);

        BTreeNode max_node = max->owner;
        max_node->count--;    // Αφαίρεσε το δεδομένο.

        node->set_nodes[index] = max;
        max->owner = node;

        repair_underflow(max_node);   // Αναδιαμόρφωσε το δέντρο.
    }

    // Αν η ρίζα αδειάσει, ρίζα γίνεται το (μοναδικό, αν έχει) παιδί
    if (root->count == 0) {
        BTreeNode first_child = root->children[0];
        if (first_child != NULL)
            first_child->parent = NULL;
        root = first_child;
    }
    return root;
}
```

## B+-trees

A variant of B-trees, important in today's **file systems** and **databases**.



## Readings

- M. T. Goodrich, R. Tamassia and D. Mount. Data Structures and Algorithms in C++. 2nd edition. John Wiley.
- Sartaj Sahni. Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++. Εκδόσεις Τζιόλα.
- R. Sedgewick. Αλγόριθμοι σε C. Κεφ. 16.3

# Hashing (Κατακερματισμός)

Κ08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Efficient implementation of ADT Map

- We need fast **equality search**
- Balanced trees
  - AVL / B-trees / Red-black / …
  - Store `(key, value)` in each node
- Or any efficient implementation of **ADT Set**
  - Store `(key, value)` as elements in the set
- The above provide search in $O(\log n)$
  - But also ordered traversal, which is **not needed**!
- Can we do better?
  - Yes, using hashing!

## Hashing

- We need to store a **`(key, value)`** pair
- Idea: use the **key** as an **index** in an array
- This is easy if `key` is a **small integer**
  - Insert: simply store `value` in `array[key]`
  - Find: read `array[key]`
- **Problem:** does not work when `key` is large (or not an integer)
  - Solution: apply a **hash function** that transforms **keys** to **indexes**

## Example

- Keys: integers, eg $1,\ 3,\ 18$
- Store data in an array of size $M = 7$
  - called a **hash table**
- Use a simple **hash function**

$$h(k) = k \mod 7$$

- A pair `(key, value)` is stored at index $h($`key`$)$

## Table T after Inserting keys $2, 10, 14, 19$

| Table T | |
|---|---|
| 0 | $14$ |
| 1 | |
| 2 | $2$ |
| 3 | $10$ |
| 4 | |
| 5 | $19$ |
| 6 | |

- Keys are stored in their **hash addresses**
- The cells of the table are often called **buckets (κάδοι)**

## Insert $24$

| Table T | |
|---|---|
| 0 | $14$ |
| 1 | |
| 2 | $2$ |
| 3 | $10$ |
| 4 | |
| 5 | $19$ |
| 6 | |

- **Collision**, $h(24) = 3$ is already taken
- **Resolution policy**
  - look at lower locations of the table to find a place for the key

## Insert $24$

| Table T | | |
|---|---|---|
| 0 | $14$ | |
| 1 | $\mathbf{24}$ | $\leftarrow$ 3rd probe |
| 2 | $\mathbf{2}$ | $\leftarrow$ 2nd probe |
| 3 | $10$ | $\leftarrow$ 1st probe |
| 4 | | |
| 5 | $19$ | |
| 6 | | |

$h(24) = 3$

## Insert $23$

| Table T | | |
|---|---|---|
| 0 | $14$ | $\leftarrow$ 3rd probe |
| 1 | $24$ | $\leftarrow$ 2nd probe |
| 2 | $2$ | $\leftarrow$ 1st probe |
| 3 | $10$ | |
| 4 | | |
| 5 | $19$ | |
| 6 | $23$ | $\leftarrow$ 4th probe |

$h(23) = 2$

## Open Addressing

- **Open addressing**
  - The method of inserting colliding keys into empty locations
- **Probe**
  - The inspection of each location
  - The locations we examined are called a **probe sequence**
- **Linear probing**
  - Examine consecutive addresses

## Double Hashing

- **Double hashing** uses non-linear probing by computing different probe decrements for different keys using a second hash function $p(k)$.
- Let us define the following probe decrement function:

$$p(k) = \max(1, \frac{k}{7})$$

## Insert $24$

| Table T | | |
|---|---|---|
| 0 | $14$ | $\leftarrow$ 2nd probe |
| 1 | | |
| 2 | $2$ | |
| 3 | $10$ | $\leftarrow$ 1st probe |
| 4 | $24$ | $\leftarrow$ 3rd probe |
| 5 | $19$ | |
| 6 | | |

$h(24) = 3$

We use a probe decrement of $p(24) = 3$

## Insert $23$

| Table T | | |
|---|---|---|
| 0 | $14$ | |
| 1 | | |
| 2 | $2$ | $\leftarrow$ 1st probe |
| 3 | $10$ | |
| 4 | $24$ | |
| 5 | $19$ | |
| 6 | $23$ | $\leftarrow$ 2th probe |

$h(23) = 2$

We use a probe decrement of $p(23) = 3$

## Collision Resolution by Separate Chaining

- The method of **collision resolution by separate chaining (χωριστή αλυσίδωση)** uses a linked list to store keys at each table entry.
- This method should not be chosen if space is at a premium, for example, if we are implementing a hash table for a mobile device.

## Example

| Table T | | |
|---|---|---|
| 0 | $14$ | |
| 1 | | |
| 2 | $2$ | $\rightarrow 23$ |
| 3 | $10$ | $\rightarrow 24$ |
| 4 | | |
| 5 | $19$ | |
| 6 | | |

## Good Hash Functions

- Suppose $T$ is a hash table having entries whose addresses lie in the range 0 to $M - 1$.
- An **ideal hashing function** $h(k)$ maps keys onto table addresses in a **uniform and random** fashion.
- In other words, for any arbitrarily chosen key, any of the possible table addresses is equally likely to be chosen.
- Also, the computation of a hash function should be **very fast.**

## Collisions

- A **collision** between two keys $k$ and $k'$ happens if, when we try to store both keys in a hash table $T$ both keys have the same hash address $h(k) = h(k')$.
- Collisions are relatively frequent even in sparsely occupied hash tables.
- A good hash function should **minimize collisions**.
- The **von Mises paradox**: if there are more than 23 people in a room, there is a greater than 50% chance that two of them will have the same birthday $(M = 365)$.

## Primary clustering

- Linear probing suffers from what we call **primary clustering (πρωταρχική συσταδοποίηση)**.
- A **cluster** (**συστάδα**) is a sequence of adjacent occupied entries in a hash table.
- In open addressing with linear probing such clusters are formed and then grow bigger and bigger. This happens because all keys colliding in the same initial location trace out identical search paths when looking for an empty table entry.
- Double hashing does not suffer from primary clustering because initially colliding keys search for empty locations along **separate** probe sequence paths.

## Ensuring that Probe Sequences Cover the Table

- In order for the open addressing hash insertion and hash searching algorithms to work properly, we have to guarantee that every probe sequence used can probe **all locations** of the hash table.
- This is obvious for linear probing.
- Is it true for double hashing?

## Choosing Table Sizes and Probe Decrements

- If we choose the table size to be a **prime number (πρώτος αριθμός)** $M$ and probe decrements to be positive integers in the range $1 \leq p(k) \leq M$ then we can ensure that the probe sequences cover all table addresses in the range 0 to $M - 1$ exactly once.

## Good Double Hashing Choices

- Choose the table size $M$ to be a **prime number**, and choose probe decrements, any integer in the range 1 to $M - 1$.
- Choose the table size $M$ to be a **power of 2** and choose as probe decrements any **odd integer** in the range 1 to $M - 1$.
- In other words, it is good to **choose probe decrements to be relatively prime with** $M$

## Deletion

- The function for deletion from a hash table is left as an exercise.
- But notice that **deletion poses some problems**.
- If we delete an entry and leave a table entry with an empty key in its place then we destroy the validity of subsequent search operations because a search terminates when an empty key is encountered.
- As a solution, we can leave the deleted entry in its place and mark it as deleted (or substitute it by a special entry “available”). Then search algorithms can treat these entries as not deleted while insert algorithms can treat them as deleted and insert other entries in their place.
- However, in this case, if we have many deletions, the hash table can easily become clogged with entries marked as deleted.

## Load Factor

The **load factor (συντελεστής πλήρωσης)** $\alpha$ of a hash table of size $M$ with $N$ occupied entries is defined by

$$\alpha = \frac{N}{M}$$

- The load factor is an important parameter in characterizing the performance of hashing techniques.

## Performance Formulas

- Hash table of size $M$ with exactly $N$ occupied entries
  - load factor $\alpha = \frac{N}{M}$
- $C_N$ : average number of probes during a **successful search**
- $C'_N$ : average number of probes during an **unsuccessful search**
  - or insertion

## Efficiency of Linear Probing

- For **open addressing with linear probing**, we have the following performance formulas:

$$C_N = \frac{1}{2}(1 + \frac{1}{1-\alpha})$$

$$C_N’ = \frac{1}{2}(1 + (\frac{1}{1-\alpha})^2)$$

- The formulas are known to apply when the table $T$ is up to 70% full (i.e., when $a \leq 0.7$).

## Efficiency of Double Hashing

- For **open addressing with double hashing**, we have the following performance formulas:

$$C_N = \frac{1}{a} \ln \frac{1}{1-\alpha}$$

$$C\text{'}_N = \frac{1}{1-\alpha}$$

## Efficiency of Separate Chaining

For **separate chaining**, we have the following performance formulas:

$$C_N = 1 + \frac{1}{2}\alpha$$

$$C'_N = \alpha$$

## Important

Important consequence of these formulas:

- The performance depends **only on the load factor** $\alpha$
- **Not** on the **number of keys** or the size of the table

## Theoretical Results: Apply the Formulas

- Let us now compare the performance of the techniques we have seen for different load factors using the formulas we presented.
- Experimental results are similar.

## Successful Search

### Load Factors

| | 0.10 | 0.25 | 0.50 | 0.75 | 0.90 | 0.99 |
|---|---|---|---|---|---|---|
| Separate chaining | 1.05 | 1.12 | 1.25 | 1.37 | 1.45 | 1.49 |
| Open/linear probing | 1.06 | 1.17 | 1.50 | 2.50 | 5.50 | 50.5 |
| Open/double hashing | 1.05 | 1.15 | 1.39 | 1.85 | 2.56 | 4.65 |

## Unsuccessful Search

### Load Factors

| | **0.10** | **0.25** | **0.50** | **0.75** | **0.90** | **0.99** |
|---|---|---|---|---|---|---|
| Separate chaining | 0.10 | 0.25 | 0.50 | 0.75 | 0.90 | 0.99 |
| Open/linear probing | 1.12 | 1.39 | 2.50 | 8.50 | 50.5 | 5000 |
| Open/double hashing | 1.11 | 1.33 | 2.50 | 4.00 | 10.0 | 100.0 |

## Complexity of Hashing

- Use a hash table that is never more than **half-full** ($\alpha \leq 0.50$)
- If the table becomes more than half-full, we can expand the table by choosing a new table twice as big and by **rehashing** the entries in the new table.
- Suppose also that we use one of the hashing methods we presented.
- Then the previous tables show that successful search can never take more than 1.50 key comparisons and unsuccessful search can never take more than 2.50 key comparisons.
- So the behaviour of hash tables is independent of the size of the table or the number of keys, hence the complexity of searching is $O(1)$

## Complexity of Hashing

- Insertion takes the same number of comparisons as an unsuccessful search, so it has complexity $O(1)$ as well.
- Retrieving and updating also take $O(1)$ time.
- For **ordered traversal** we must sort the keys ($O(n \log n)$), so hash tables are not good candidates for ADT Set

## Important observations

1. It **can** happen that all entries **hash to the same index**
    - So the **worst-case** complexity of search/insert is $O(n)$
    - But the **average-case** remains $O(1)$
        - Under the assumption of a good hash function
2. **Rehashing** takes $O(n)$ time
    - So the **real-time** complexity of insert is $O(n)$
    - But it happens rarely
        - So the **amortized-time** complexity is $O(1)$
        - Similarly to a dynamic array

# Complexity, summary

| Search | Worst-case | Average-case |
|---|---|---|
| Real-time | $O(n)$ | $O(1)$ |
| Amortized-time | $O(n)$ | $O(1)$ |

| Insert | Worst-case | Average-case |
|---|---|---|
| Real-time | $O(n)$ | $O(n)$ |
| Amortized-time | $O(n)$ | $O(1)$ |

## Load Factors and Rehashing

- Experiments and average case analysis suggest that **we should maintain**
  - $\alpha < 0.5$ for open addressing schemes
  - $\alpha < 0.9$ for separate chaining
- With open addressing, as the load factor grows beyond 0.5 and starts approaching 1, clusters of items in the table start to grow as well.
- **At the limit, when a is close to 1, all table operations have linear expected running times** since, in this case, we expect to encounter a linear number of occupied cells before finding one of the few remaining empty cells.

## Load Factors and Rehashing

- If the load factor of a hash table goes significantly above a specified threshold, then it is common to require the table to be resized to regain the specified load factor. This process is called **rehashing (ανακατακερματισμός)** or **dynamic hashing (δυναμικός κατακερματισμός).**
- When rehashing to a new table, a good requirement is having the new array's size be **at least double** the previous size.

## Summary: Open Addressing or Separate Chaining?

- Open addressing schemes save space but they are not faster.
- As you can see in the above theoretical results (and corresponding experimental results), the separate chaining method is either competitive or faster than the other methods depending on the load factor of the table.
- So, **if memory is not a major issue, the collision handling method of choice is separate chaining**.

# Comparing ADT Map implementations

| | **Search** | **Insert** | **Delete** | **Ordered traversal** |
|---|---|---|---|---|
| Sorted Array | $O(\log n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| AVL | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | $O(n)$ |
| Hashing | $O(1)$ | $O(1)$ | $O(1)$ | $O(n \log n)$ |

## Choosing a Good Hash Function

- Ideally, a hash function will map keys **uniformly and randomly** onto the entire range of the hash table locations with each location being equally likely to be the target of the function for a randomly chosen key.

## Example of a Bad Choice

- **Keys**
  - Strings of 3 ASCII characters
  - 24-bit integer containing the 3 8-bit bytes
- Use open addressing with double hashing.
- Select a table size $M = 2^8 = 256$
- Define our hashing function as $h(k) = k \mod 256$

## Example

- This hash function is a poor one because it **selects the low-order character of the three-character key** as the value of $h(k)$
- If the key is $321$ , when considered as a 24-bit integer, it has the numerical value $3 \times 256^2 + 2 \times 256^1 + 1 \times 256^0$
- Thus when we do the modulo 256 operation, we get the value $1$

## Example

- “Similar” keys create collisions

$$h(AAA) = h(ABA) = h(ACA) = h(BAA) = \ldots$$

- Thus this hash function will create and preserve clusters instead of **spreading** them as a good hash function will do.
- Hash functions should take into account **all the bits of a key**, not just some of them.

## Hash Functions

Hash function $h(k)$ as consisting of two actions:

- **Hash code**
  - Map the key $k$to an integer
- **Compression function**
  - Map the hash code to the range of indices 0 to $M - 1$

## Hash Codes

- The first action that a hash function performs is to take an arbitrary key and map it into an integer value.
- This integer need not be in the range 0 to $M - 1$ and may even be negative, but we want the set of hash codes to **avoid collisions**.
- If the hash codes of our keys cause collisions, then there is no hope for the compression function to avoid them.

## Hash Codes in C

- The hash codes described below are based on the assumption that the number of bits of each data type is known.

## Converting to an Integer

- For any data type that is D represented using at most as many bits as our integer hash codes, we can simply **take an integer interpretation of the bits as a hash code** for elements of D.
- Thus, for the C basic types `char`, `short int` and `int`, we can achieve a good hash code simply by **casting** this type to `int`.

## Converting to an Integer

- On many machines, the type `long int` has a bit representation that is twice as long as type `int`.
- One possible hash code for a long element is to simply cast it down to an int.
- But notice that this hash code **ignores half of the information** present in the original value. So if many of the keys differ only in these bits, they will **collide** using this simple hash code.
- A **better hash code**, which takes all the original bits into consideration, **sums** an integer representation of the high-order bits with an integer representation of the low-order bits.

## Converting to an Integer

- In general, if we have **an object $x$ whose binary representation can be viewed as a k-tuple** of integers $(x_0, x_1, \ldots, x_{k-1})$, we can form a hash code for as $\sum_{i=0}^{k-1} x_i$
- **Example**: Given any floating-point number, we can sum its mantissa and exponent as long integers and then apply a hash code for long integers to the result.

## Summation Hash Codes

- The summation hash code, described above, is **not a good choice** for character strings or other variable-length objects that can be viewed as tuples of the form $(x_0, x_1, \ldots, x_{k-1})$ where the order of the $x_i$'s is significant.
- **Example**: Consider a hash code for a string s that sums the ASCII values of the characters in s. This hash code produces lots of unwanted collisions for common groups of strings e.g., `temp01` and `temp10`.
- A better hash code should take the order of the $x_i$'s into account.

## Polynomial Hash Codes

- Let be an integer constant such that $a \neq 1$
- We can use the polynomial

$$x_0 a^{k-1} + x_1 a^{k-2} + \cdots + x_{k-2} a + x_{k-1}$$

as a hash code for $(x_0, x_1, \ldots, x_{k-1})$.

- This is called a **polynomial hash code**.
- To evaluate the polynomial we should use the efficient **Horner’s method**:

$$x_{k-1} + a(x_{k-2} + a(x_{k-3} + \cdots + a(x_1 + a x_o)) \ldots ))$$

## Polynomial Hash Codes

- Experiments show that in a list of over 50,000 English words, if we choose $a = 33,\ 37,\ 39,\ 41$ we produce **less than seven collisions** in each case.
- For the sake of speed, we can apply the hash code to only a fraction of the characters in a long string.

## Polynomial Hash Codes

```
// dbj2 hash function

uint hash_string(Pointer value) {
  uint hash = 5381;

  for (char* s = value; *s != '\0'; s++)
        hash = (hash * 33)+ *s;

  return hash;
}
```

## Polynomial Hash Codes

- In theory, we first compute a polynomial hash code and then apply the compression function *modulo M*
- The previous hash function takes the modulo M **at each step.**
- The two approaches are the same because the following equality holds for all $a, b, x, M$ that are nonnegative integers:

$$(((ax) \mod M) + b) \mod M = (ax + b) \mod M$$

- The approach of the previous function is preferable because, otherwise, **we get errors with long strings** when the polynomial computation produces overflows (try it!).

## Hashing Floating Point Quantities

- We can achieve a better hashing function for floating point numbers than casting them down to `int` as follows.
- Assuming that a `char` is stored as an 8-bit byte, we could **interpret a 32-bit float as a four-element character array** and use the hashing functions we discussed for strings.

## Some Applications of Hash Tables

- Databases
- Symbol tables in compilers
- Browser caches
- Peer-to-peer systems and torrents (distributed hash tables)

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*. Chapter 11
- M.T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++*. 2nd edition. Chapter 9
- R. Sedgewick. *Αλγόριθμοι σε C*. 3η Αμερικανική Έκδοση. Εκδόσεις Κλειδάριθμος. Κεφάλαιο 14

# Graphs (Γράφοι)

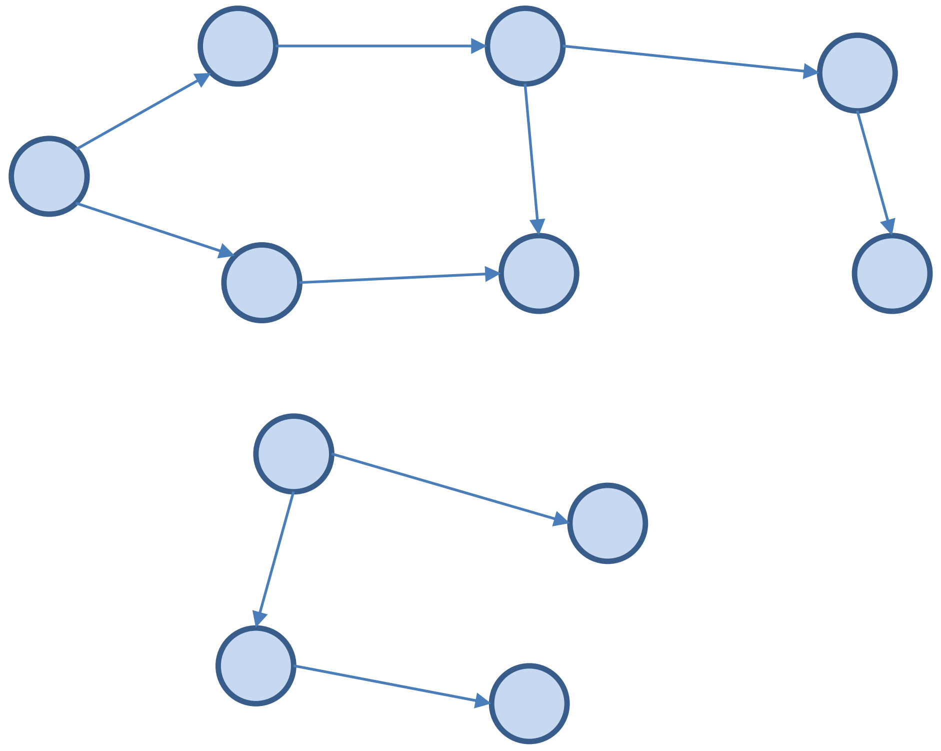
K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

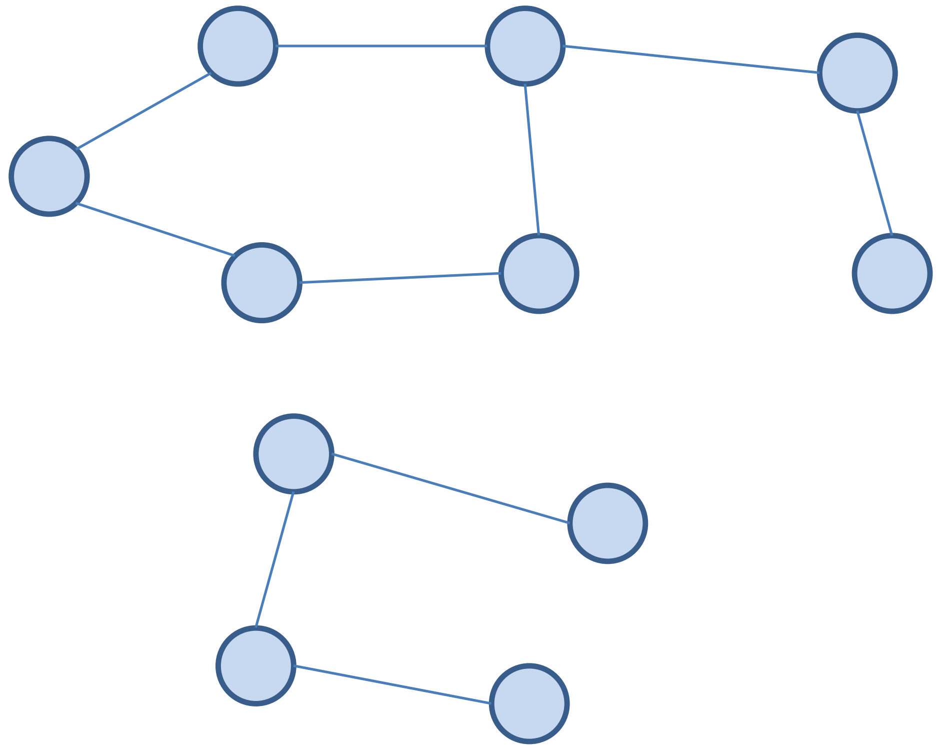
## Graphs

- **Graphs** are collections of nodes in which various pairs are connected by line segments. The nodes are usually called **vertices** (**κορυφές**) and the line segments **edges (ακμές).**
- Graphs are **more general than trees**. Graphs are allowed to have cycles and can have more than one connected component.
- Some authors use the terms **nodes (κόμβοι)** and **arcs (τόξα)** instead of vertices and edges.

## Example of Graphs (Directed)



# Example of Graphs (Undirected)



## Examples of Graphs

- Transportation networks
- **Interesting problem**: What is the path with one or more stops of shortest overall distance connecting a starting city and a destination city?

## Examples

- A network of oil pipelines
- **Interesting problem**: What is the maximum possible overall flow of oil from the source to the destination?

## Examples

- The Internet
- **Interesting problem**: Deliver an e-mail from user A to user B

## Examples

- The Web
- **Interesting problem**: What is the PageRank of a Web site?

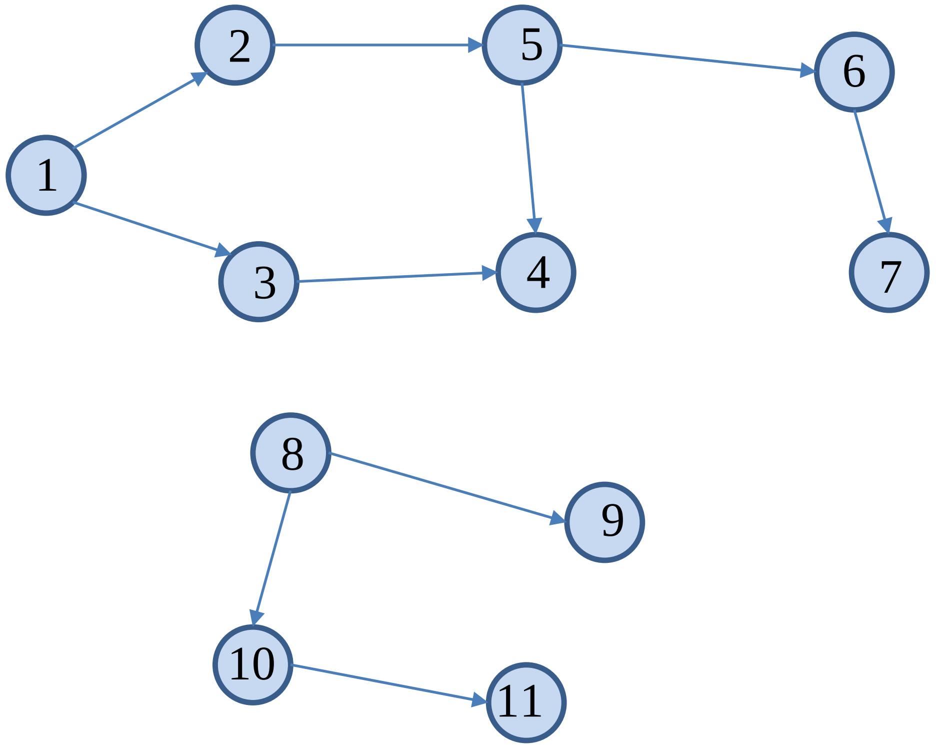
## Examples

- The Facebook social network
- **Interesting problem**: Are John and Mary connected? What interesting clusters exist?

# Formal Definitions

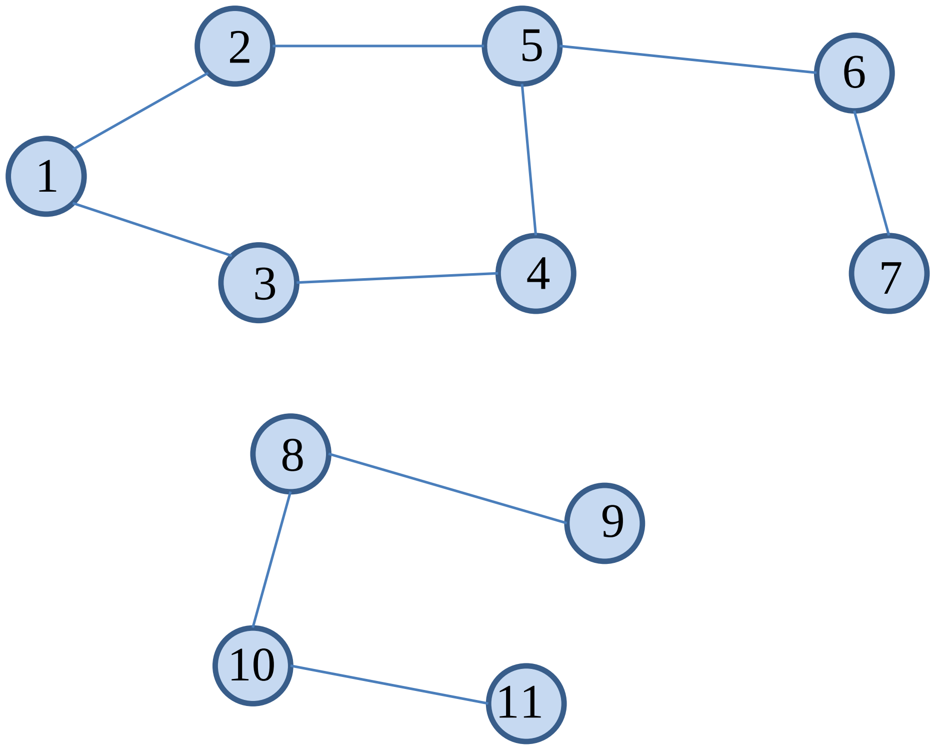
- A **graph** $G = (V, E)$ consists of a set of **vertices** $V$ and a set of **edges** $E$, where the edges in $E$ are formed from pairs of **distinct** vertices in $V$.
- If the edges have directions then we have a **directed graph (κατευθυνόμενο γράφο)** or **digraph**. In this case edges are ordered pairs of vertices e.g., $(u, v)$ and are called **directed**. If $(u, v)$ is a directed edge then $u$ is called its **origin** and $v$ is called its **destination**.
- If the edges do not have directions then we have an **undirected graph (μη-κατευθυνόμενος γράφο)**. In this case edges are unordered pairs of vertices e.g., $\{u, v\}$ and are called **undirected**.
- For simplicity, we will use the directed pair notation noting that in the undirected case $(u, v)$ is the same as $(v, u)$ .
- When we say simply graph, we will mean an undirected graph.

## Example of a Directed Graph



$$G = (V, E)$$
$$V = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11$$
$$E \;=\; (1,2), (1,3), (2,5), (3,4), (5,4), (5,6), (6,70, (8,9), (8,10), (10,11$$

## Example of an Undirected Graph



$G = (V, E)$

$V = 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11$

$E\ = \ (1, 2), (1, 3), (2, 5), (3, 4), (5, 4), (5, 6), (6, 70, (8, 9), (8, 10), (10, 11$

# More Definitions

- Two different vertices $v_i, v_j$ in a graph $G = (V, E)$ are said to be **adjacent (γειτονικές)** if there exists an edge $(v_i, v_j) \in E$.
- An edge is said to be **incident (προσπίπτουσα)** on a vertex if the vertex is one of the edge's endpoints.
- A **path (μονοπάτι)** $p$ in a graph $G = (V, E)$, is a sequence of vertices of $V$ of the form $p = v_1 v_2 \ldots v_n, (n \geq 2)$ in which each vertex $v_i$, is adjacent to the next one $v_{i+1}$ (for $1 \leq i \leq n - 1$ ).
- The **length** of a path is the number of edges in it.
- A path is **simple** if each vertex in the path is distinct.
- A **cycle** is a path $p = v_1 v_2 \ldots v_n$ of length greater than one that begins and ends at the same vertex (i.e., $v_1 = v_n$ ).

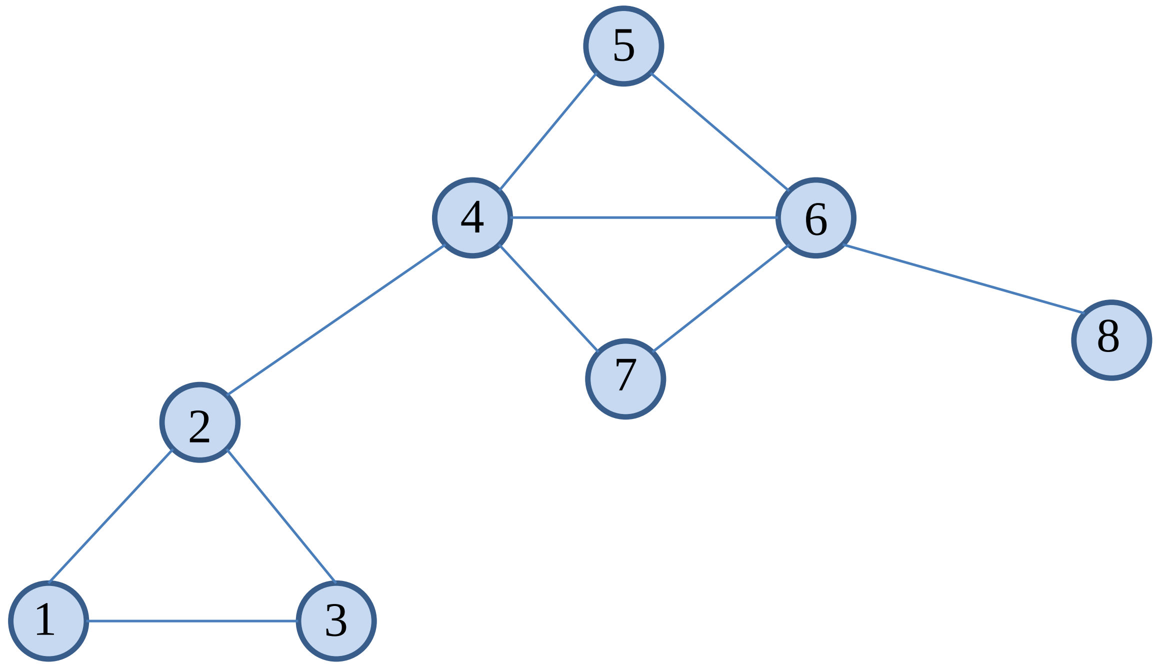
## Definitions

- A **directed path** is a path such that all edges are directed and are traversed along their direction.
- A **directed cycle** is similarly defined.

## Definitions

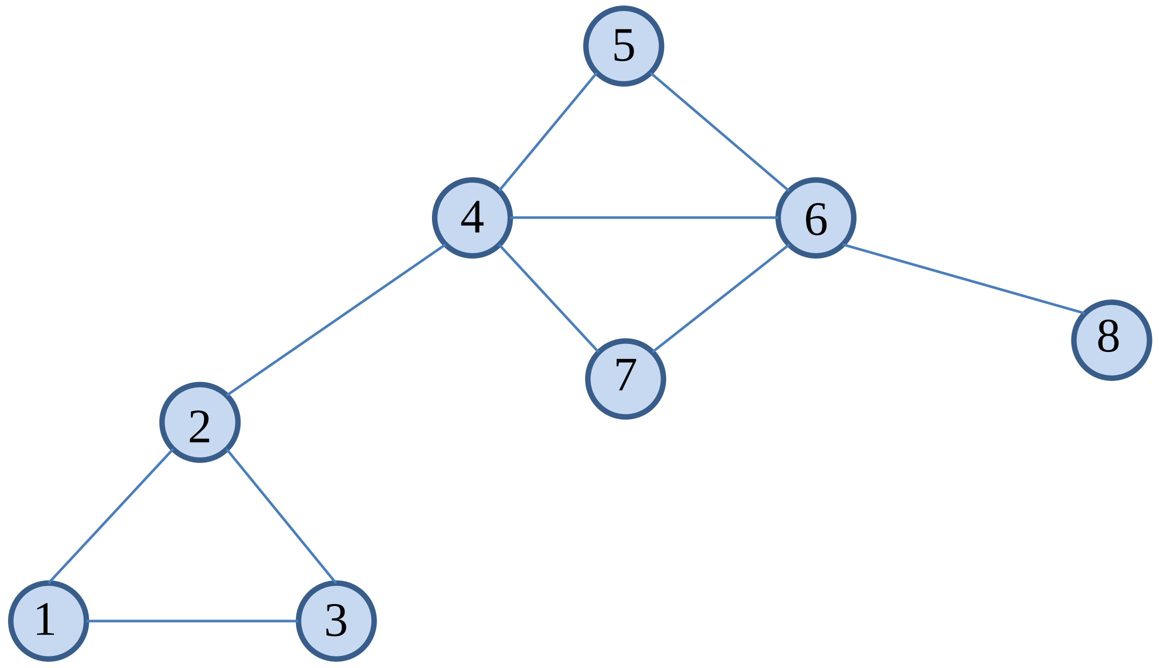
- A **simple cycle** is a path that travels through three or more **distinct** vertices and connects them into a loop.

## Example



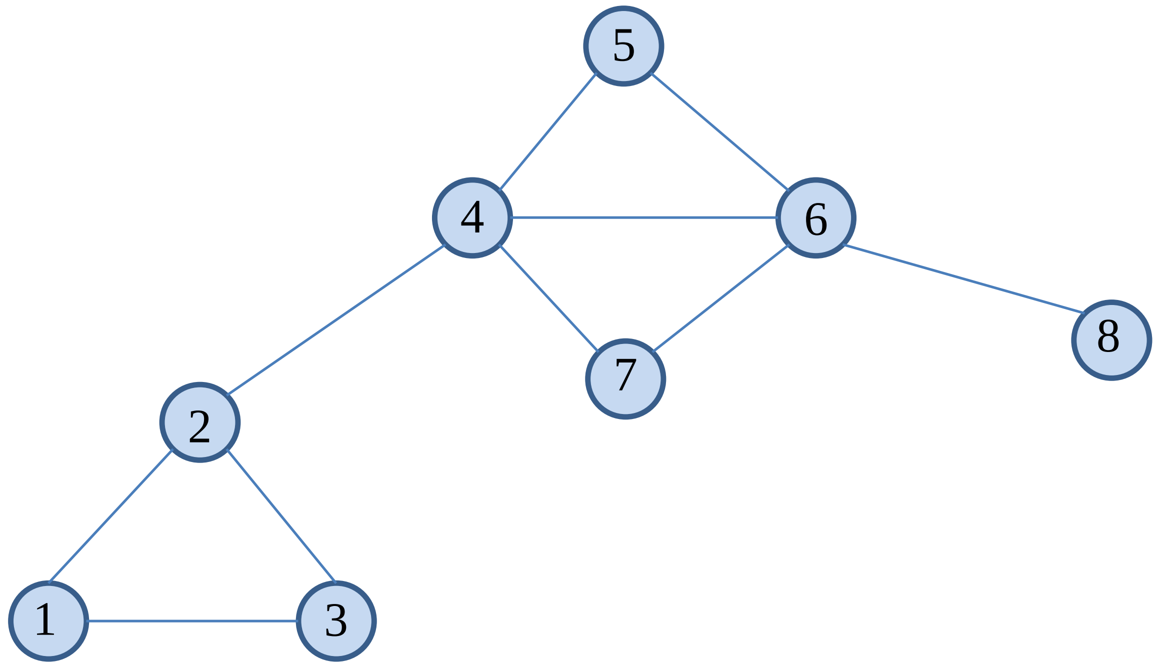
**Four simple cycles:** (1,2,3,1) (4,5,6,7,4) (4,5,6,4) (4,6,7,4)

## Example



**Two non-simple cycles:** (1,2,1) (4,5,6,4,7,6,4)

## Example



**A path that is not a cycle:** (1,2,4,6,8)

## Connectivity and Components

- Two vertices in a graph $G = (V, E)$ are said to be **connected (συνδεδεμένες)** if there is a path from the first to the second in $G$
- Formally, if $x \in V \text{ and } y \in V, \text{ where } x \neq y$ , then $x$ and $y$ are **connected** if there exists a path $p = v_1 v_2 \ldots v_n \in G$ in such that $x = v_1$ and $y = v_n$

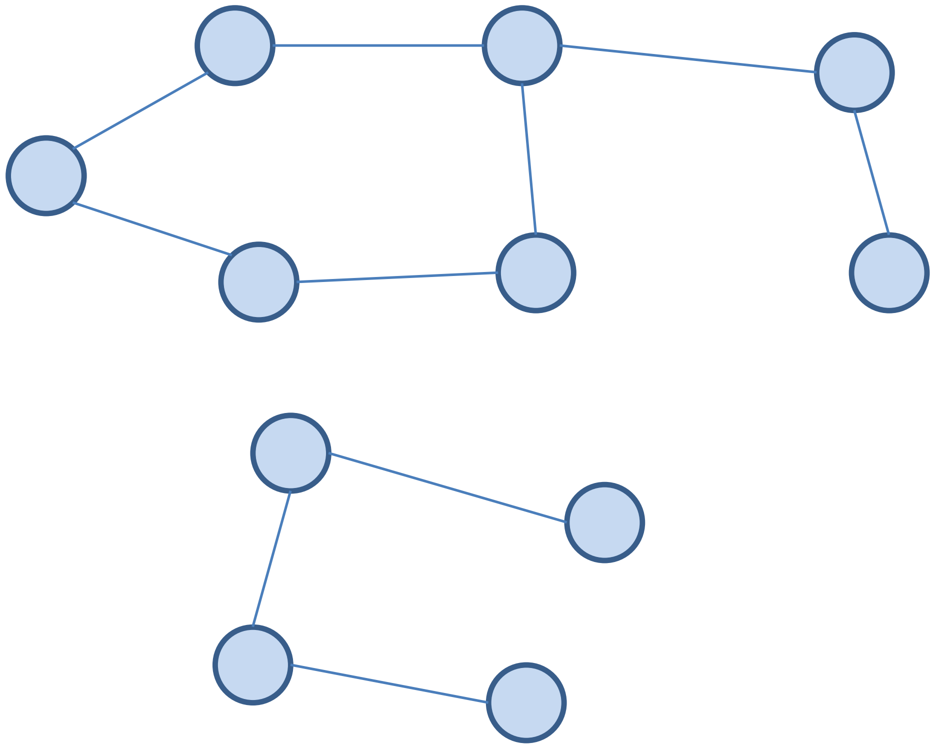
## Connectivity and Components

- In the graph $G = (V, E)$ , a **connected component (συνεκτική συνιστώσα)** is a subset $S$ of the vertices $V$ that are all connected to one another.

- A connected component $S$ of $G$ is a **maximal connected component (μέγιστη συνεκτική συνιστώσα)** provided there is no bigger subset $T$ of vertices in $V$ such that $T$ properly contains $S$ and such that $T$ itself is a connected component of $G$.

- An undirected graph $G$ can always be separated into maximal connected components $S_1, S_2, \ldots ; , S_n$ such that $S_i \cap S_j = \varnothing$ whenever $i \neq j$.

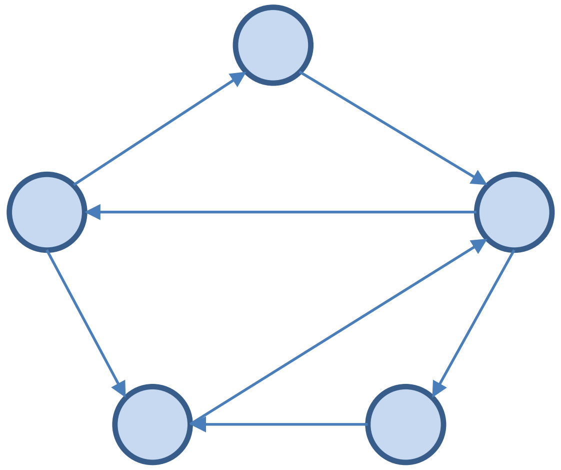
## Example of Undirected Graph and its Separation into Two Maximal Connected Components



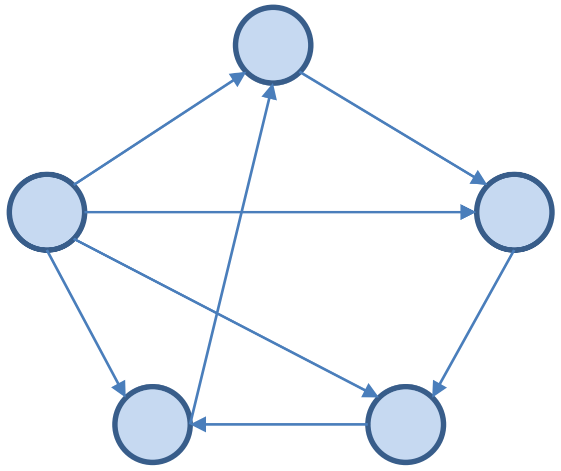
## Connectivity and Components in Directed Graphs

- A subset $S$ of vertices in a **directed** graph $G$ is **strongly connected (ισχυρά συνεκτικό)** if for each pair of distinct vertices $(v_i, v_j)$ in $S$, $v_i$ is connected to $v_j$ **and** $v_j$ is connected to $v_i$.
- A subset $S$ of vertices in a **directed** graph $G$ is **weakly connected (ασθενώς συνεκτικό)** if for each pair of distinct vertices $(v_i, v_j)$in $S$, $v_i$ is connected to $v_j$ **or** $v_j$ is connected to $v_i$.

## Example: A Strongly Connected Digraph



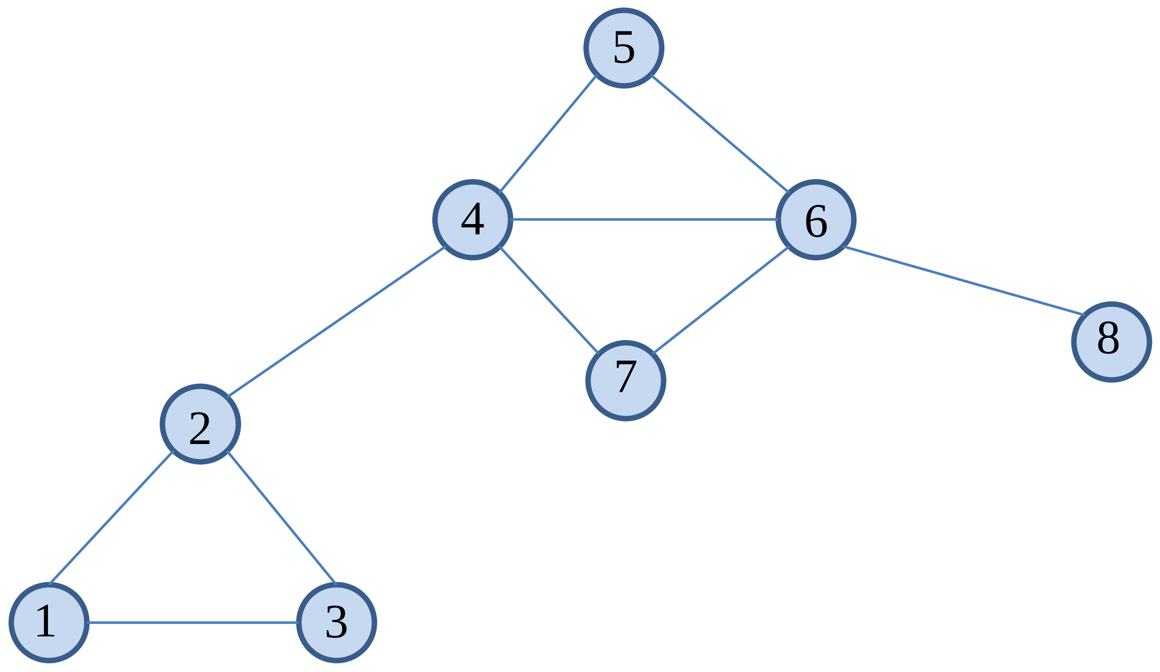
## Example: A Weakly Connected Digraph



## Degree in Undirected Graphs

- In an undirected graph $G$ the **degree** (**βαθμός**) of vertex $x$ is the number of edges $e$ in which $x$ is one of the endpoints of $e$.
- The degree of a vertex $x$ is denoted by $\deg(x)$.

## Example



The degree of node 1 is 2.
The degree of node 4 is 4.
The degree of node 8 is 1.

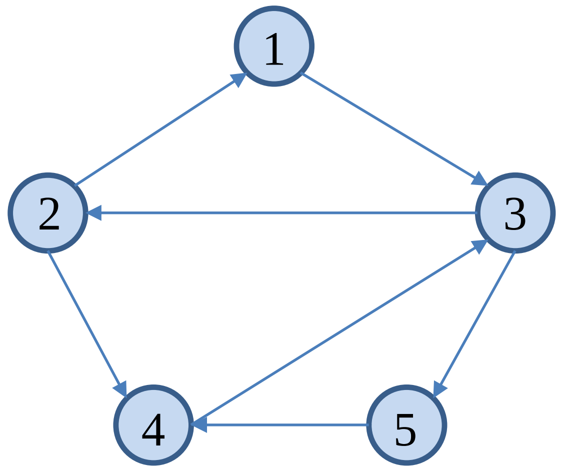
## Predecessors and Successors in Directed Graphs

- If $x$ is a vertex in a **directed** graph $G = (V, E)$ then the set of **predecessors (προηγούμενων)** of $x$ denoted by $\mathrm{Pred}(x)$ is the set of all vertices $y \in V$ such that $(y, x) \in E$.
- Similarly the set of **successors (επόμενων)** of $x$ denoted by $\mathrm{Succ}(x)$ is the set of all vertices $y \in V$ such that $(x, y) \in E$.

## In-Degree and Out-Degree in Directed Graphs

- The **in-degree** of a vertex $x$ is the number of predecessors of $x$
- The **out-degree** of a vertex $x$ is the number of successors of $x$
- We can also define the in-degree and the out-degree by referring to the **incoming** and **outgoing** edges of a vertex.
- The in-degree and out-degree of a vertex $x$ are denoted by $\mathbf{indeg}(x)$ and $\mathrm{outdeg}(x)$ respectively.

## Example



The in-degree of node 4 is 2. The out-degree of node 4 is 1.

## Proposition

- If $G$ is an undirected graph with $m$ edges, then

$$\sum_{v \in G} \deg(v) = 2m$$

  .

- Proof?
  - Each edge is counted twice

## Proposition

- If is a directed graph with edges, then

$$\sum_{v \in G} \mathbf{indeg}(v) = \sum_{v \in G} \mathbf{outdeg}(v) = m$$

- Proof?
    - Each edge is counted once

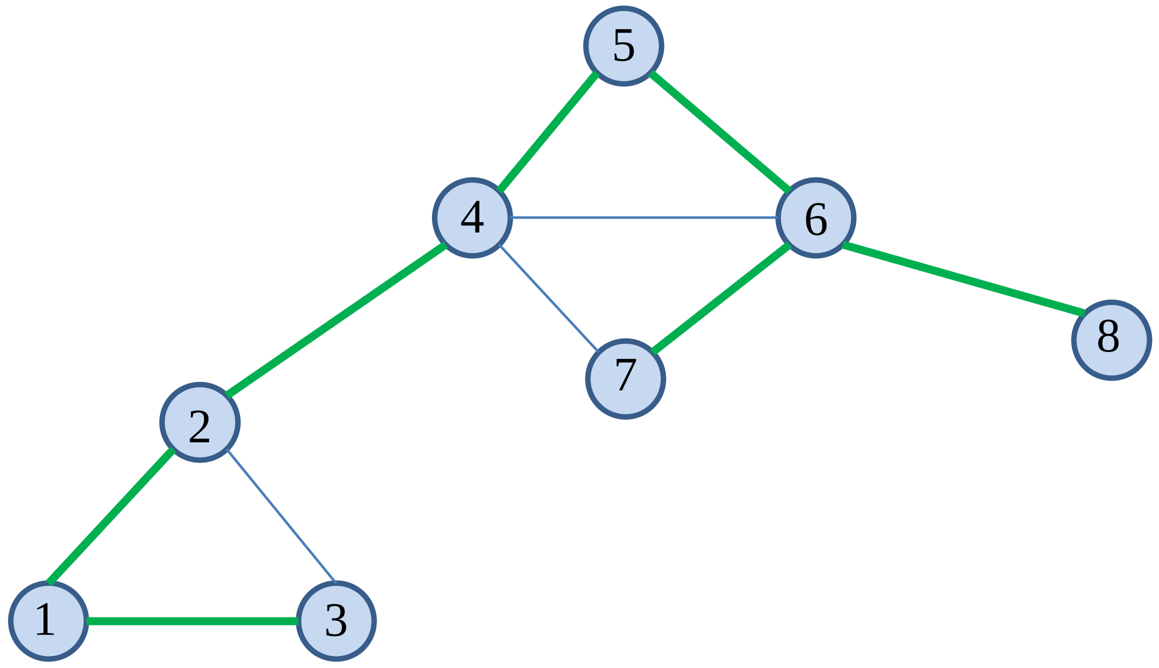
## Proposition

- Let $G$ be a graph with $n$ vertices and $m$ edges. If $G$ is undirected, then $m \leq \frac{n(n-1)}{2}$ and if $G$ is directed, then $m \leq n(n-1)$.
- Proof?
  - If $G$ is undirected then the maximum degree of a vertex is $n-1$. Therefore, from the previous proposition about the sum of the degrees, we have $2m \leq n(n-1)$.
  - If $G$ is directed then the maximum in-degree of a vertex is $n-1$. Therefore, from the previous proposition about the sum of the in-degrees, we have $m \leq n(n-1)$.

# More definitions

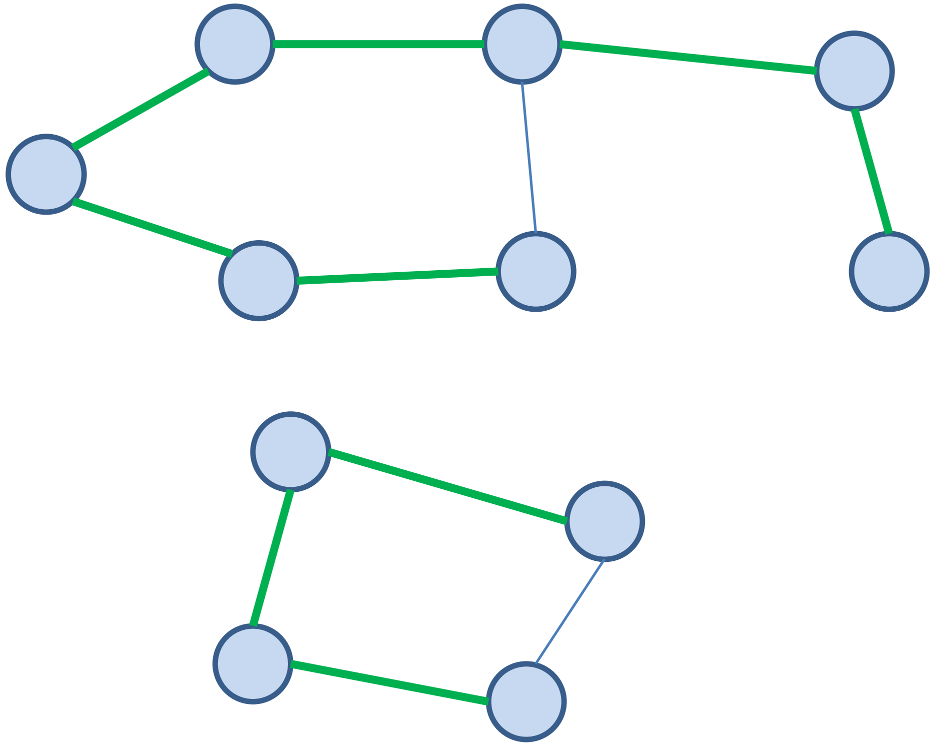
- A **subgraph (υπογράφος)** of a graph $G$ is a graph $H$ whose vertices and edges are subsets of the vertices and edges of $G$ respectively.
- A **spanning subgraph (υπογράφος επικάλυψης)** of $G$ is a subgraph of $G$ that contains all the vertices of $G$.
- A **forest (δάσος)** is a graph without cycles.
- A **free tree (ελεύθερο δένδρο)** is a connected forest i.e., a connected graph without cycles. The trees that we studied in earlier lectures are **rooted trees (δένδρα με ρίζα)** and they are different than free trees.
- A **spanning tree (δένδρο επικάλυψης)** of a graph is a spanning subgraph that is a free tree.

## Example



The thick green lines define a spanning tree of the graph.

## Example

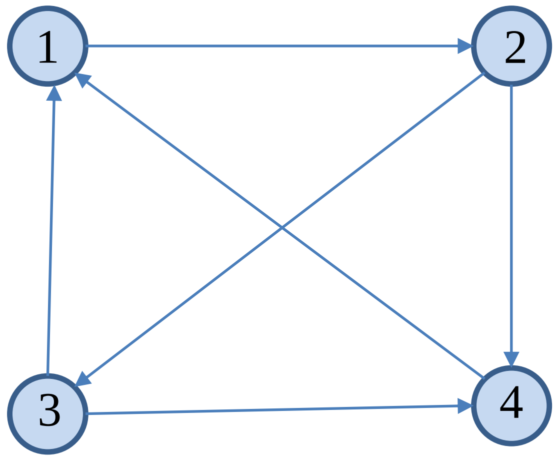


The thick green lines define a forest which consists of two free trees.

## Graph Representations: Adjacency Matrices

- Let $G = (V, E)$ be a graph. Suppose we number the vertices in $V$ as $v_1, v_2 \ldots v_n$
- The **adjacency matrix (πίνακας γειτνίασης)** corresponding to $G$ is an $n \times n$ matrix such that $T[i, j] = 1$ if there is an edge $(v_i, v_j) \in E$, and $T[i, j] = 0$ if there is no such edge in $E$.

## Example



A graph G

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 0 & 0 \\ 2 & 0 & 0 & 1 & 1 \\ 3 & 1 & 0 & 0 & 1 \\ 4 & 1 & 0 & 0 & 0 \end{array}$$

The adjacency matrix for graph G

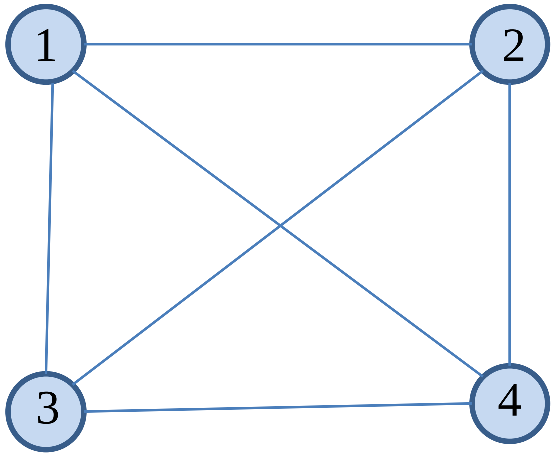
## Adjacency Matrices

- The adjacency matrix of an **undirected graph** $G$ is a **symmetric matrix** i.e., $T[i, j] = T[j, i]$ for all and in the range $1 \leq i, j \leq n$
- The adjacency matrix for a **directed graph** need not be symmetric.

## Adjacency Matrices

- The **diagonal entries** in an adjacency matrix (of a directed or undirected graph) **are zero**, since graphs as we have defined them are not permitted to have looping self-referential edges that connect a vertex to itself.

## Example



An undirected graph G

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 1 & 1 & 1 \\ 2 & 1 & 0 & 1 & 1 \\ 3 & 1 & 1 & 0 & 1 \\ 4 & 1 & 1 & 1 & 0 \end{array}$$

The adjacency matrix for graph G

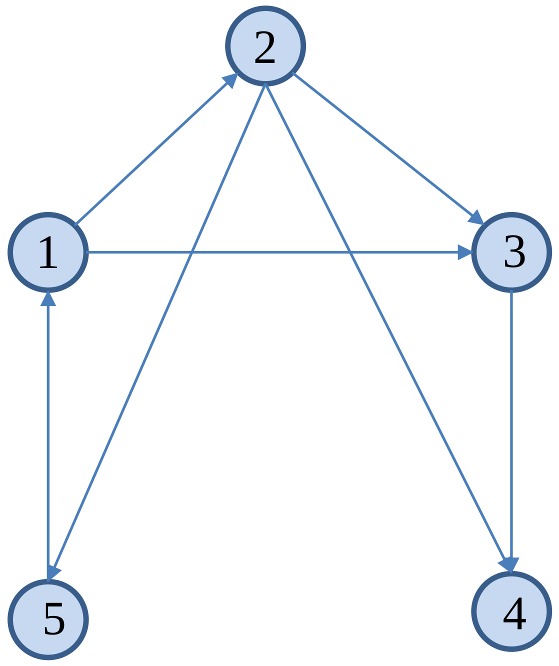
## Adjacency Sets

- Another way to define a graph $G = (V, E)$ is to specify **adjacency sets (σύνολα γειτνίασης)** for each vertex in $V$.
- Let $V_x$ stand for the set of all vertices **adjacent** to $x$ in an undirected graph $G$ or the set of all vertices that are **successors** of $x$ in a directed graph $G$.
- If we give both the vertex set $v$ and the collection $A = \{V_x | x \in V\}$ of adjacency sets for each vertex in then we have given enough information to define the graph $G$.

## Graph Representations: Adjacency Lists

- Another family of representations for a graph uses **adjacency lists (λίστες γειτνίασης)** to represent the adjacency set $V_x$ for each vertex $x$ in the graph.

## Example Directed Graph

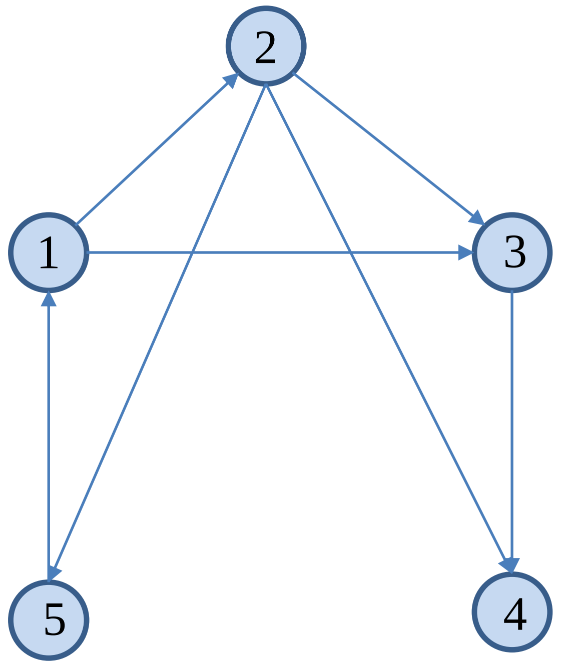


A directed graph G

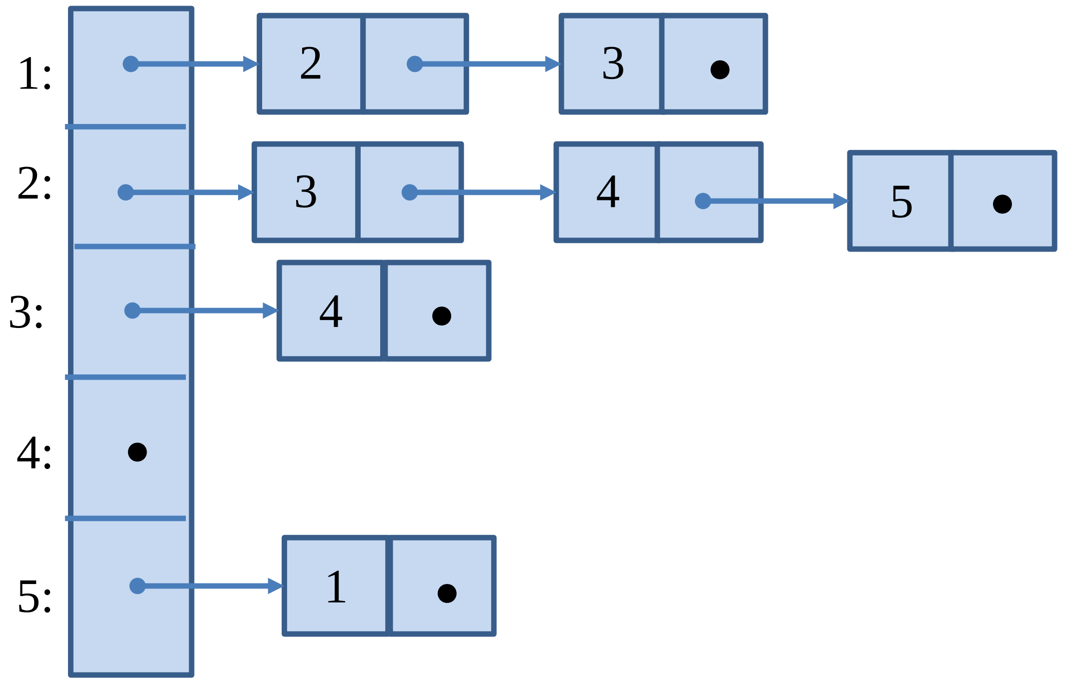
| Vertex Number | Out Degree | Adjacency list |
|---|---|---|
| 1 | 2 | 2 3 |
| 2 | 3 | 3 4 5 |
| 3 | 1 | 4 |
| 4 | 0 | |
| 5 | 1 | 1 |

The **sequential** adjacency lists for graph G. Notice that vertices are listed in their **natural order**.

## Example Directed Graph

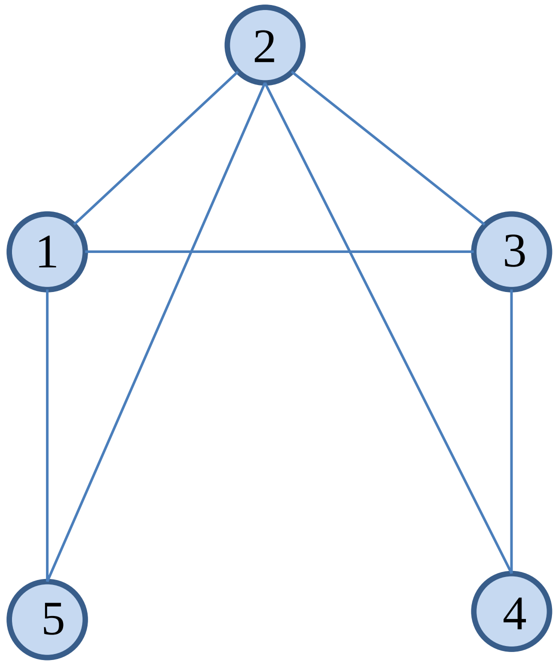


A directed graph G



The **linked** adjacency lists for graph G. Notice that vertices in a list are organized according to their **natural order.**

# Example Undirected Graph



An undirected graph G

| Vertex Number | Degree | Adjacency list |
|---|---|---|
| 1 | 3 | 2 3 5 |
| 2 | 4 | 1 3 4 5 |
| 3 | 3 | 1 2 4 |
| 4 | 2 | 2 4 |
| 5 | 2 | 1 2 |

The sequential adjacency lists for graph G

## Graph Searching

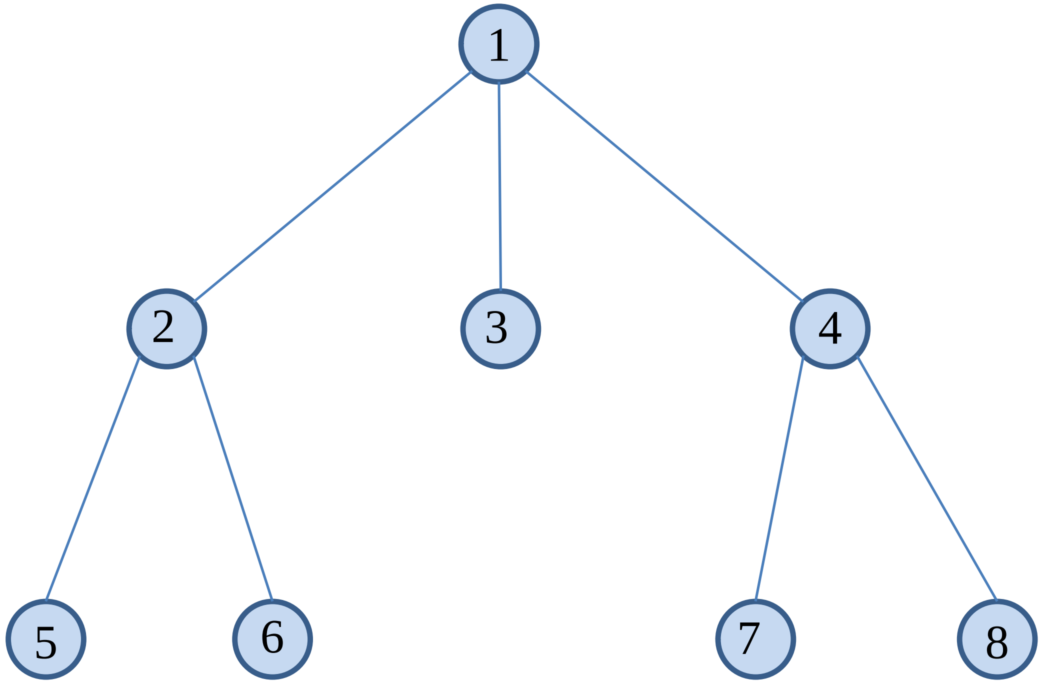
- To search a graph $G$, we need to visit all vertices of $G$ in some systematic order.
- Each vertex $v$ can be a structure with a `bool` valued member $v$. Visited which is initially `false` for all vertices of $G$. When we visit $v$, we will set it to `true`.

## An Algorithm for Graph Searching

```
// Ψευδοκώδικας, επίσκεψη όλων των κόμβων του γράφου

void graph_search(G) {
    Let G = (V,E) be a graph
    Let C be an empty container

    for (each vertex x in V) {
        x.visited = false;
    }
    Insert v into C;

    while (C is non-empty) {
        Remove a vertex x from container C;
        if (!x.visited) {
            visit(x);
            x.visited = true;
            for (each vertex w adjacent to x) {
                if (!w.visited))
                   Insert w into C;
            }
        }
    }
}
```

## Graph Searching

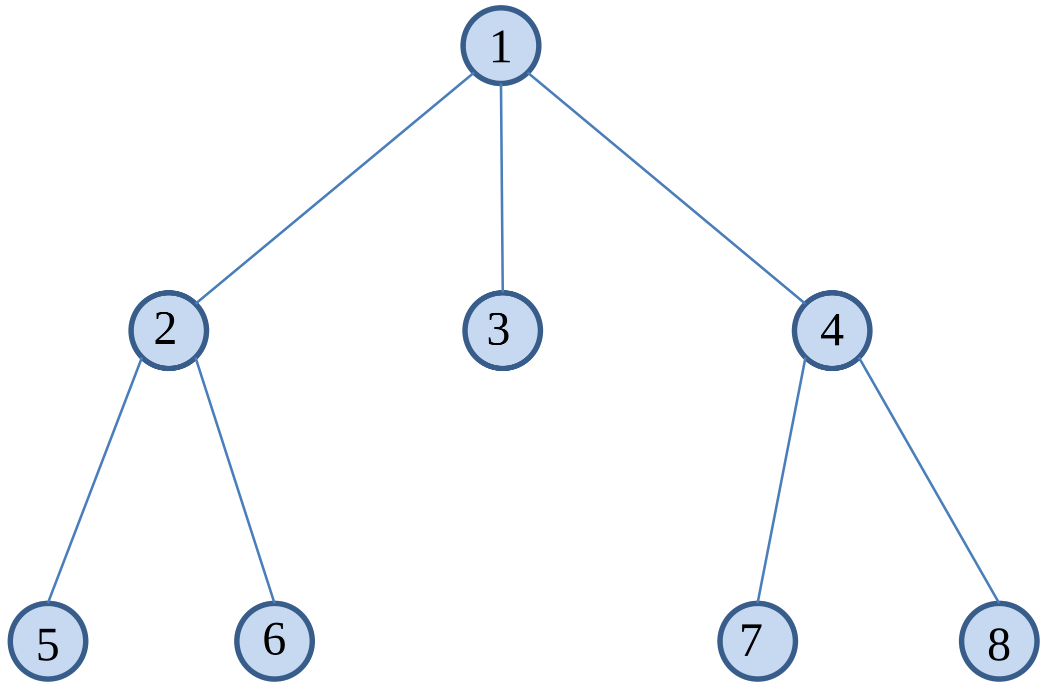
Interesting case: the container $C$ is a stack.



In what order vertices are visited?

# Graph Searching

Eg. the container $C$ is a stack.



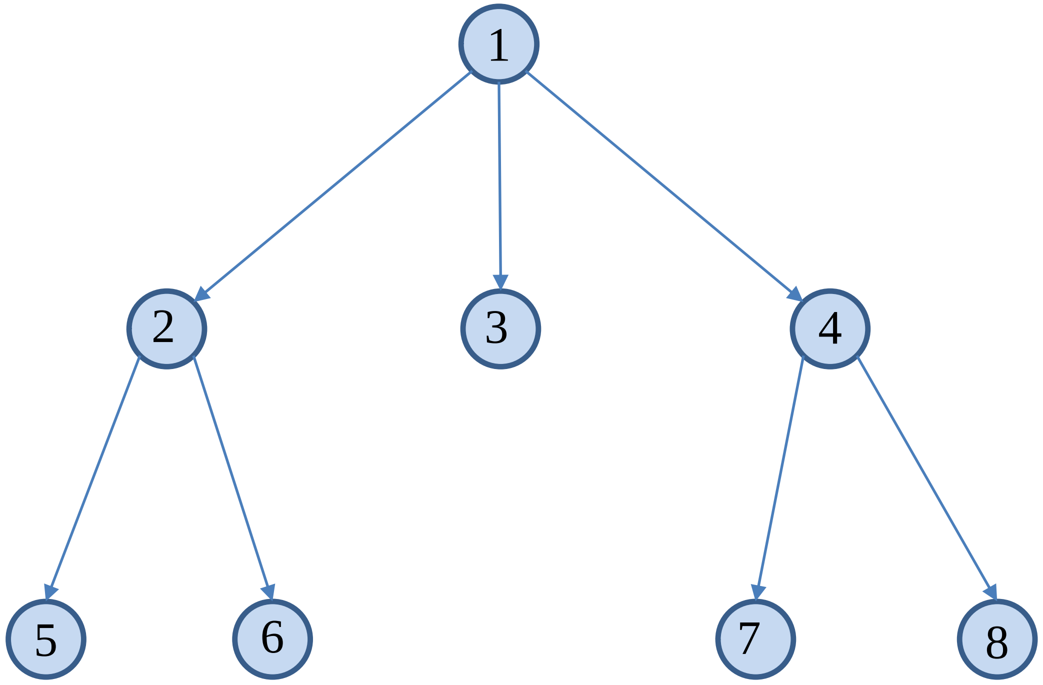
The vertices are visited in the order 1, 4, 8, 7, 3, 2, 6, 5.

## Depth-First Search (DFS)

- When $C$ is a **stack**, the tree in the previous example is searched in **depth-first order**.
- **Depth-first search (αναζήτηση πρώτα κατά βάθος)** at a vertex always goes down (by visiting unvisited children) before going across (by visiting unvisited brothers and sisters).
- Depth-first search of a graph is analogous to a **pre-order traversal** of an ordered tree.

## Graph Searching

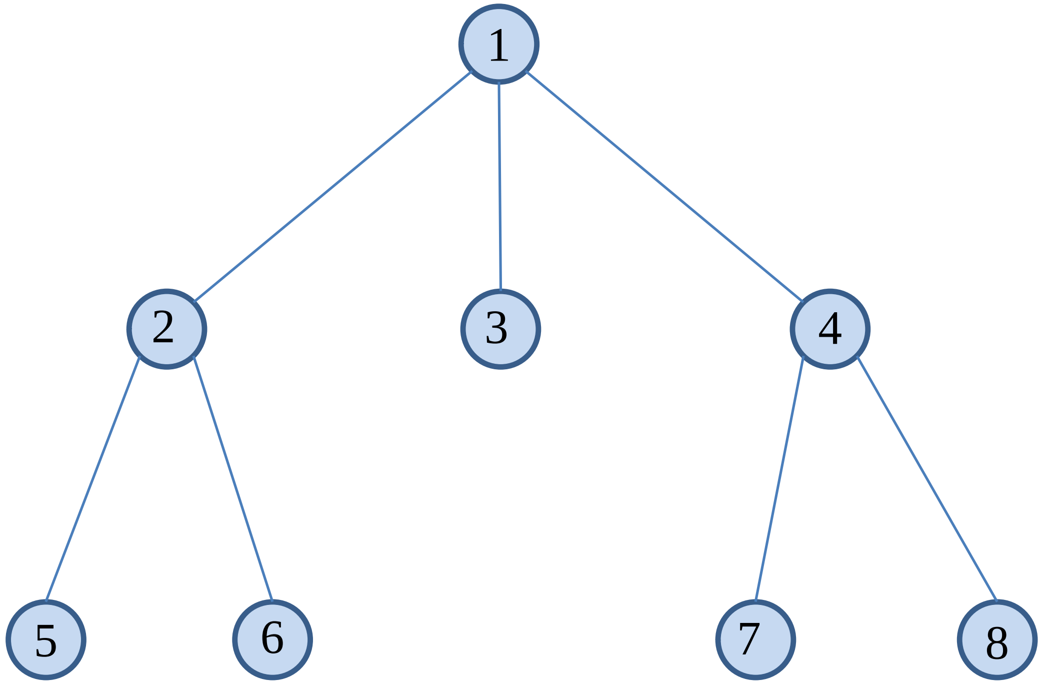
Another interestg case: the container $C$ is a queue.



What is the order vertices are visited?

## Graph Searching

Another interestg case: the container $C$ is a queue.

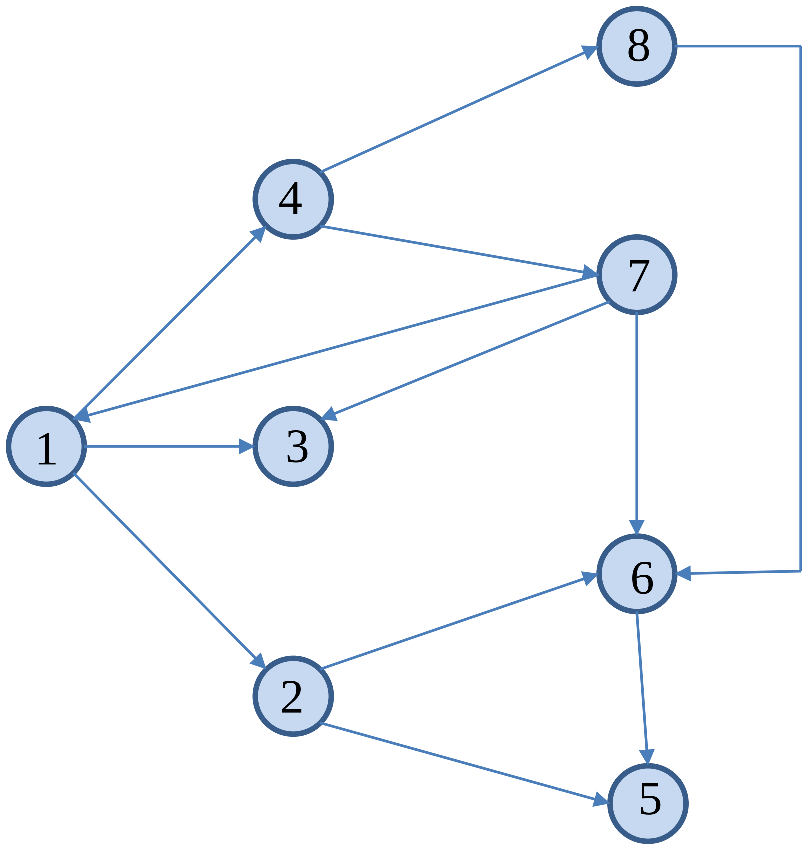


The vertices are visited in the order 1, 2, 3, 4, 5, 6, 7 and 8.

## Breadth-First Search (BFS)

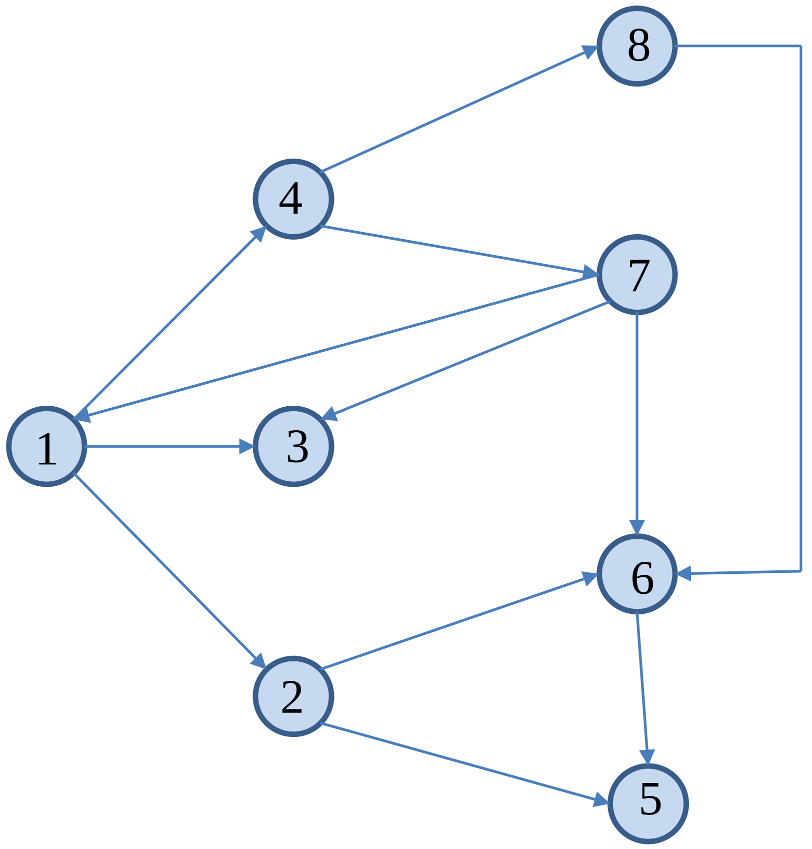
- When $C$ is a **queue**, the tree in the previous example is searched in **breadth-first order**.
- **Breadth-first search (αναζήτηση πρώτα κατά πλάτος)** at a vertex always goes broad before going deep.
- Breadth-first traversal of a graph is analogous to a traversal of an ordered tree that visits the nodes of the tree in **level-order**.
- BFS subdivides the vertices of a graph in **levels**. The starting vertex is at level 0, then we have the vertices adjacent to the starting vertex at level 1, then the vertices adjacent to these vertices at level 2 etc.

## Example



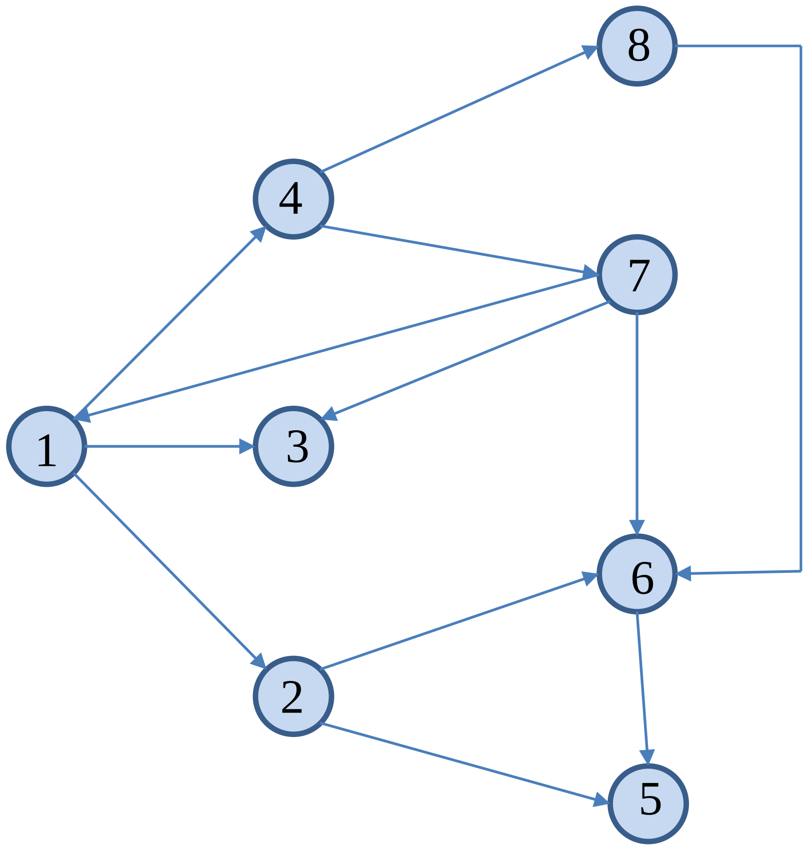
What is the order of visiting vertices for DFS?

# Example



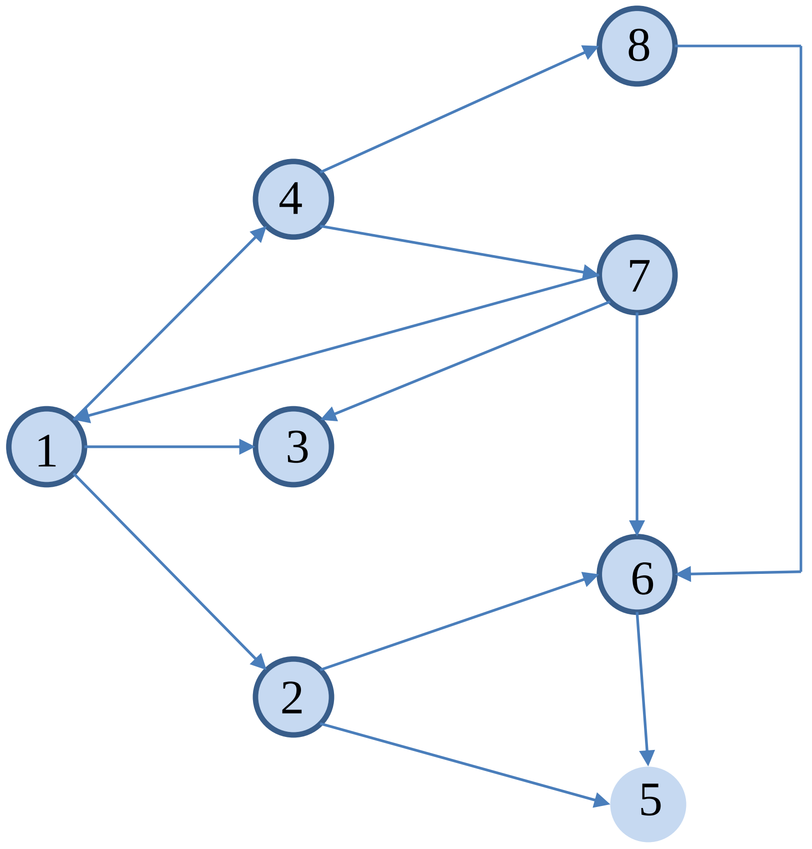
Depth-first search visits the vertices in the order 1, 4, 8, 6, 5, 7, 3 and 2

## Example



What is the order of visit for BFS?

## Example



Breadth-first search visits the vertices in the order 1, 2, 3, 4, 5, 6, 7 and 8.

## Exhaustive Search

- Either the stack version or the queue version of the algorithm `GraphSearch` will visit every vertex in a graph $G$ provided that $G$ consists of a single strongly connected component.
- If this is not the case, then we can enumerate all the vertices of $G$ and run `GraphSearch` starting from each one of them in order to visit all the vertices of $G$.

## Exhaustive Search

```
void graph_exhaustive_search(G) {
    Let G = (V,E) be a graph.
    for (each vertex v in G) {
        graph_search(G, v)
    }
}
```

## Recursive DFS

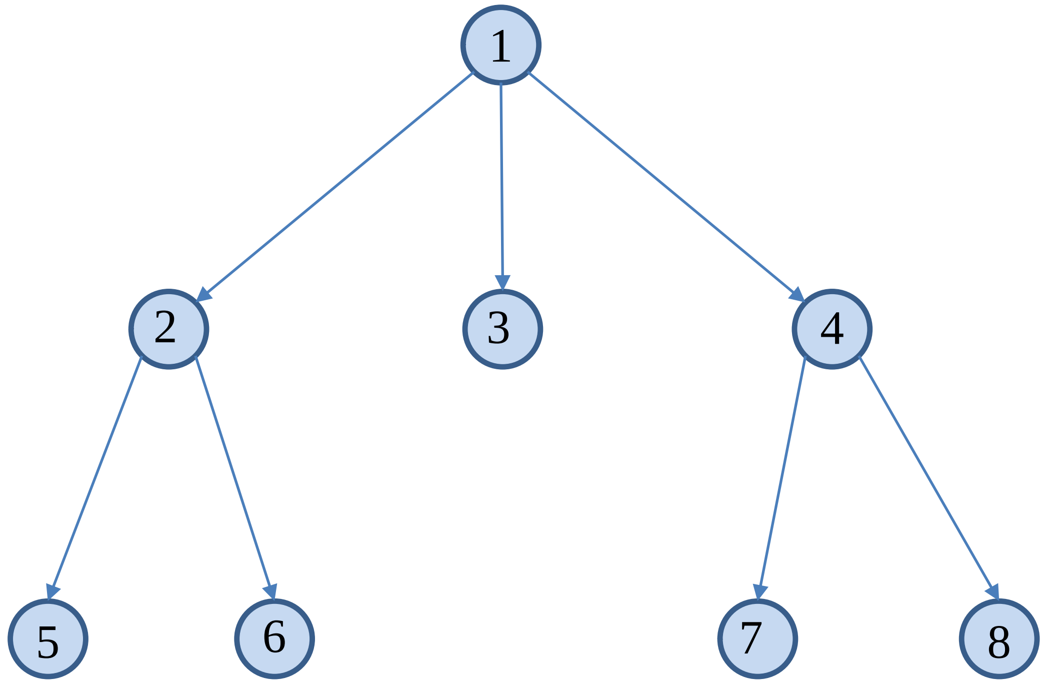
- DFS can be also written recursively
- The stack is essentially replaced by the **function call stack**

## Recursive DFS

```
// Ψευδοκώδικας, επίσκεψη όλων των κόμβων του γράφου

void graph_dfs(G) {
    for (each vertex x in V) {
        x.visited = false;
    }
    for (each vertex x in V) {
        if (!x.visited))
            traverse(G, x);
    }
}

void traverse(G, x) {
    visit(x);
    x.visited = true;

    for (each vertex w adjacent to v) {
        if (!w.visited))
            traverse(G, w);
    }
}
```

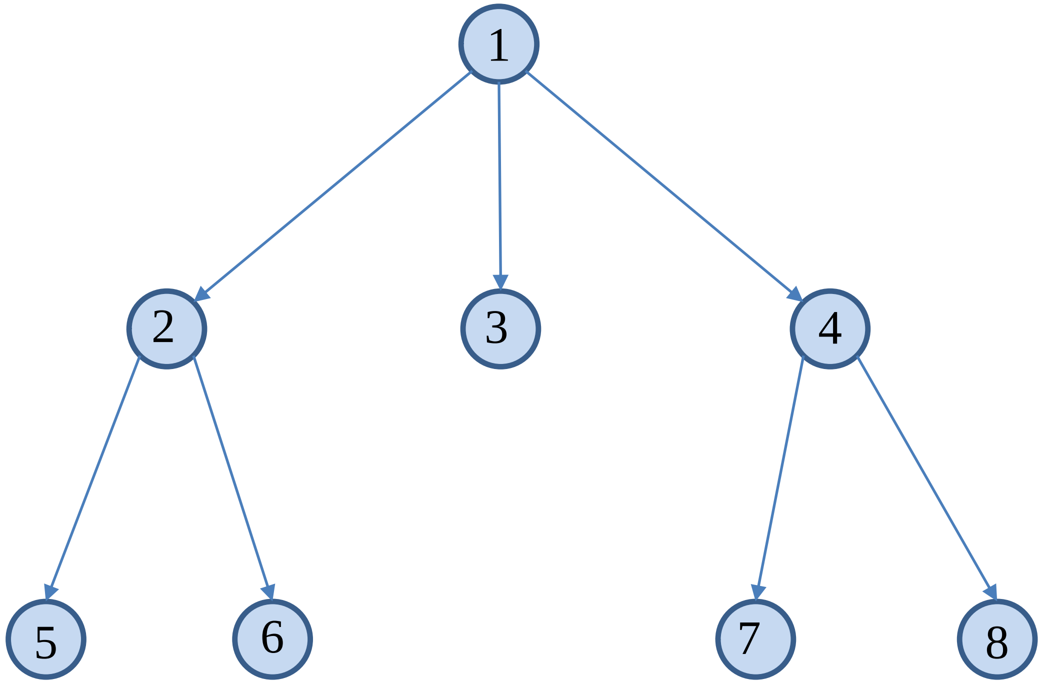
# Example of Recursive DFS

What is the order vertices are visited?



## Example

The vertices are visited in the order 1, 2, 5, 6, 3, 4, 7 and 8. This is different than the order we got when using a stack!



## Complexity of DFS

- DFS as implemented above (with adjacency lists) on a graph with $e$ edges and $n$ vertices has complexity $O(n + e)$.
- To see why observe that on no vertex is `traverse` called more than once, because as soon as we call `traverse` with parameter $x$ , we mark $x$ visited and we never call `traverse` on a vertex that has previously been marked as visited.
- Thus, the total time spent going down the adjacency lists is proportional to the lengths of those lists, that is $O(e)$
- The initialization steps in `graph_dfs` have complexity $O(n)$
- Thus, the total complexity is $O(n + e)$

# Complexity of DFS

- If DFS is implemented using an adjacency matrix, then its complexity will be $O(n^2)$.
- If the graph is **dense (πυκνός)**, that is, it has close to $O(n^2)$ edges the difference of the two implementations is minor as they would both run in $O(n^2)$ time.
- If the graph is **sparse (αραιός)**, that is, it has close to $O(n)$ edges, then the adjacency matrix approach would be much slower than the adjacency list approach.

## Complexity of BFS

- BFS with adjacency lists has the same complexity as DFS i.e., $O(n + e)$.

## Readings

- T. A. Standish. *Data Structures , Algorithms and Software Principles in C.* Chapter 10
- R. Kruse and C.L. Tondo and B. Leung. *Data Structures and Program Design in C*. 2nd edition. Chapter 11
- A. V. Aho, J. E. Hopcroft and J. D. Ullman. *Data Structures and Algorithms.* Chapters 6 and 7
- M. T. Goodrich, R. Tamassia and D. Mount. *Data Structures and Algorithms in C++.* 2nd edition. Chapter 13

# Weighted graphs

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Κώστας Χατζηκοκολάκης

## Weighted graphs

- Graphs with numbers, called **weights**, attached to each edge
  - Often restricted to **non-negative**
- Directed or undirected
- Examples
  - **Distance** between cities
  - **Cost** of flight between airports
  - **Time** to send a message between routers

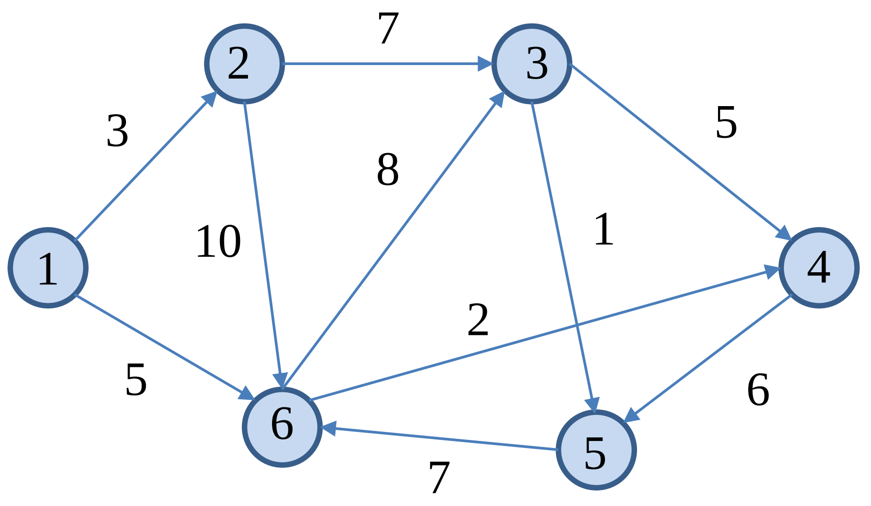
## Weighted graphs

- Adjacency matrix representation

$$T[i,j] = \begin{cases} w_{i,j} & \text{if } i,j \text{ are connected} \\ \infty & \text{if } i \neq j \text{ are not connected} \\ 0 & \text{if } i = j \end{cases}$$

- Similarly for adjacency lists

## Example weighted graph



## Example weighted graph

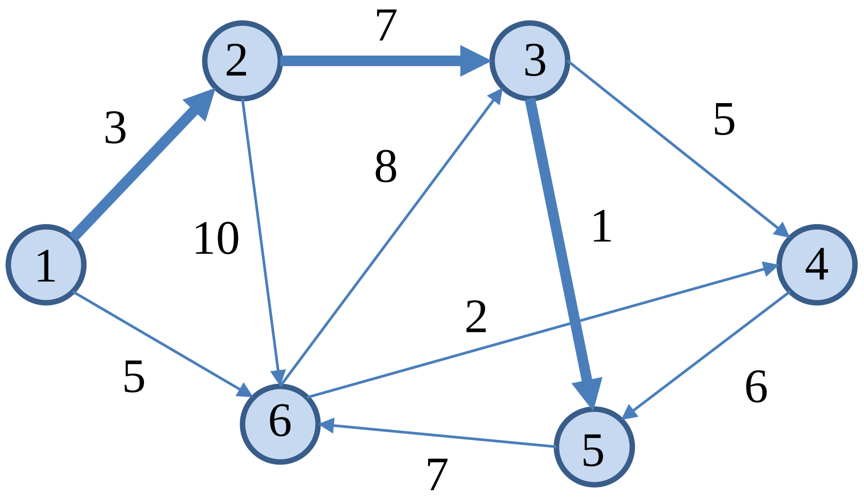
|  | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 3 | ∞ | ∞ | ∞ | 5 |
| 2 | ∞ | 0 | 7 | ∞ | ∞ | 10 |
| 3 | ∞ | ∞ | 0 | 5 | 1 | ∞ |
| 4 | ∞ | ∞ | ∞ | 0 | 6 | ∞ |
| 5 | ∞ | ∞ | ∞ | ∞ | 0 | 7 |
| 6 | ∞ | ∞ | 8 | 2 | ∞ | 0 |

Adjacency matrix

## Shortest paths

- The **length** of a path is the **sum of the weights** of its edges
- Very common problem
  - find the **shortest path** from $s$ to $d$
- Examples
  - Shortest route between cities
  - Cheapest connecting flight
  - Fastest network route
  - …

# Shortest path from vertex 1 to vertex 5



## Shortest path problem

Two main variants:

- **Single source $s$**
  - Find the shortest path from $s$ to each node
  - **Dijkstra's** algorithm
    - Only for **non-negative** weights (important!)
- **All-pairs**
  - Find the shortest path between all pairs $s, d$
  - **Floyd-Warshall** algorithm
    - Any weights

## Dijkstra's algorithm

Main ideas:

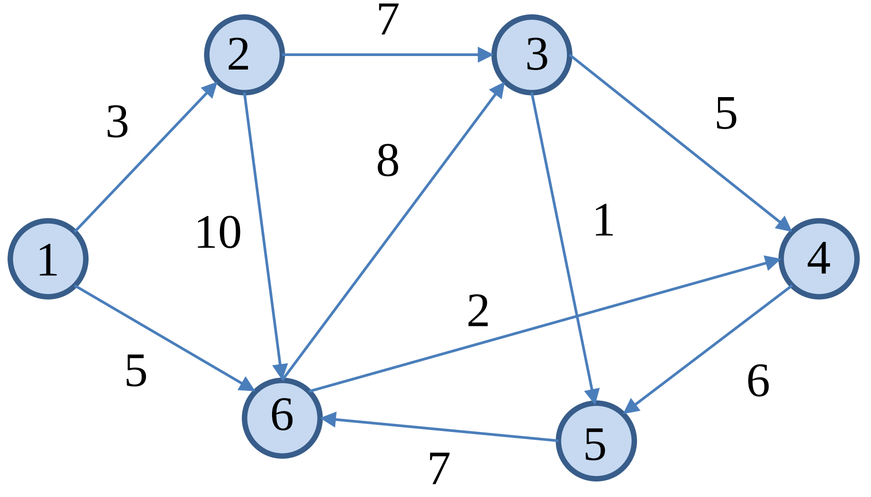
- Keep a set $W$ of **visited** nodes
  - Start with $W = \{s\}$ (or $W = \{\}$)
- Keep a matrix $\Delta[u]$
  - Minimum distance from $s$ to $u$ **passing only through** $W$
  - Start with $\Delta[u] = T[s, u]$ (or $\Delta[s] = 0, \Delta[u] = \infty$)
- At each step we **enlarge** $W$ by adding a **new vertex** $w \notin W$
  - $w$ is the one with **minumum** $\Delta[w]$

## Dijkstra's algorithm

Main ideas:

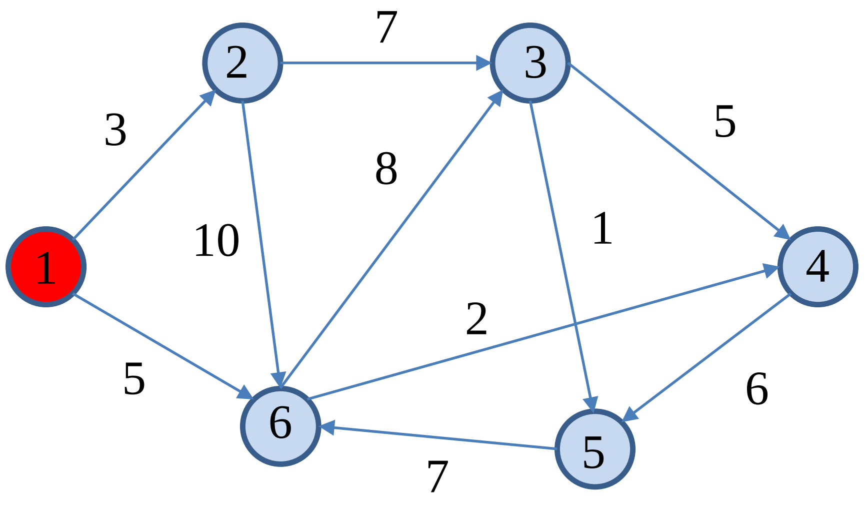
- Adding $w$ to $W$ might affect $\Delta[u]$
    - For some **neighbour** $u$ of $w$
- We might now have a **shorter** path to $u$ **passing through** $w$
    - Of the form $s \to \ldots \to w \to u$
    - If $\Delta[u] > \Delta[w] + T[w,u]$
- In this case we update $\Delta$
    - $\Delta[u] = \Delta[w] + T[w,u]$

## Example graph



# Expanding the vertex set w in stages

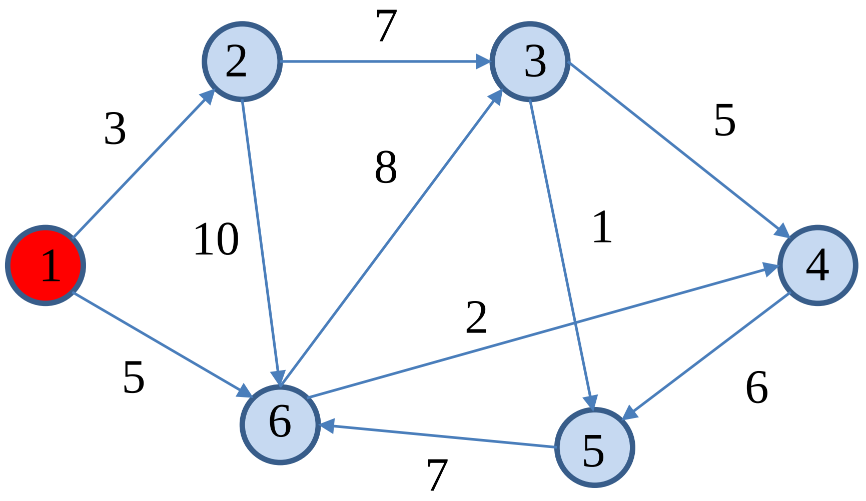
| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |



# Expanding the vertex set w in stages

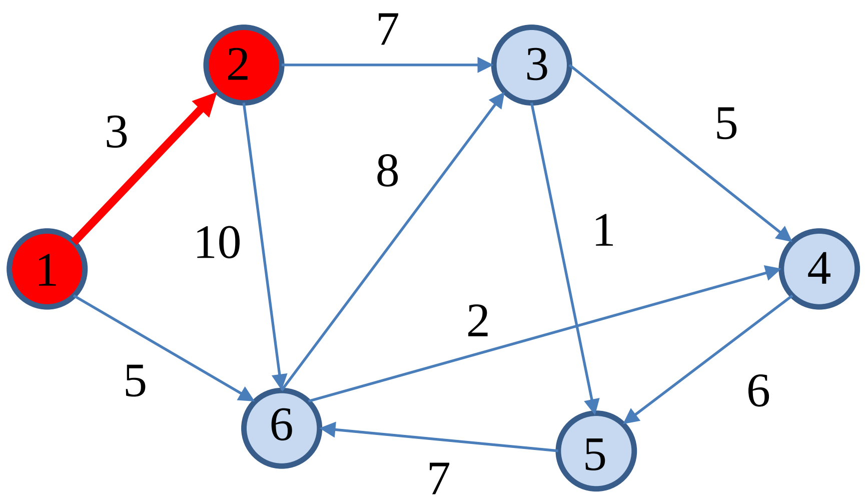
**W=2** is chosen for the second stage.

| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |



# Expanding the vertex set w in stages

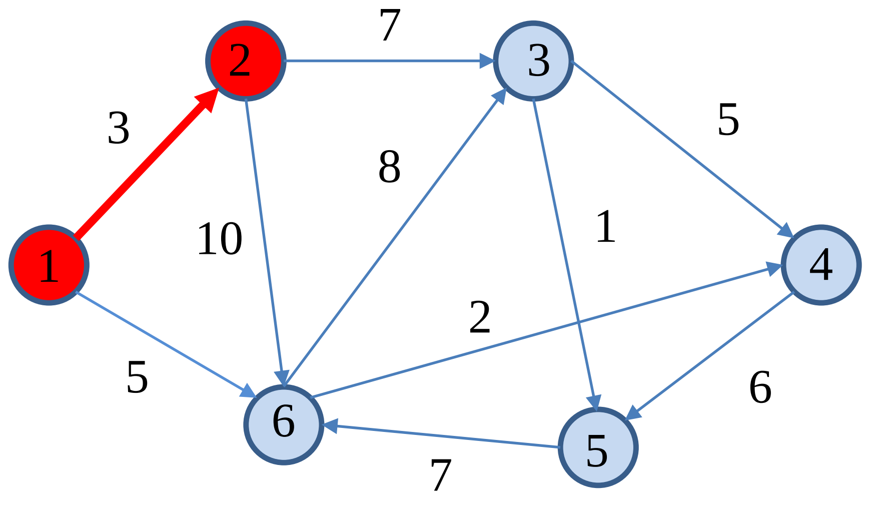
| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |
| 2 | {1,2} | {3,4,5,6} | 2 | 3 | 0 | 3 | 10 | ∞ | ∞ | 5 |



# Expanding the vertex set w in stages

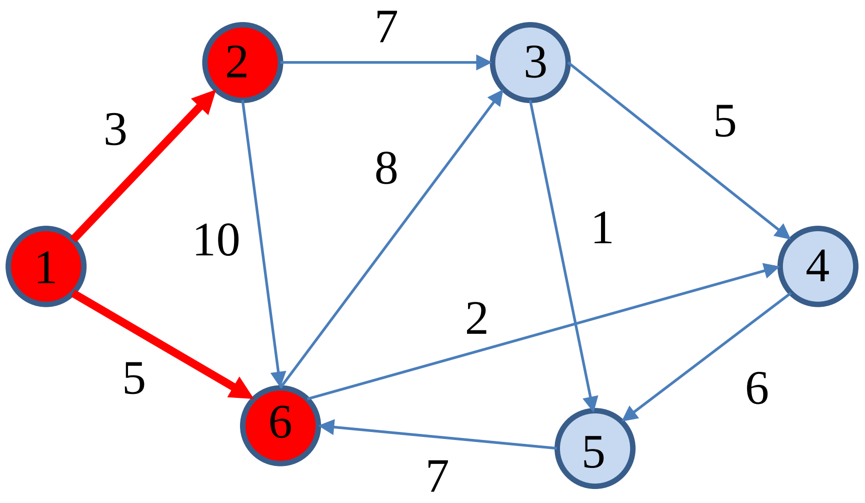
**W=6** is chosen for the third stage.

| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |
| 2 | {1,2} | {3,4,5,6} | 2 | 3 | 0 | 3 | 10 | ∞ | ∞ | 5 |



# Expanding the vertex set w in stages

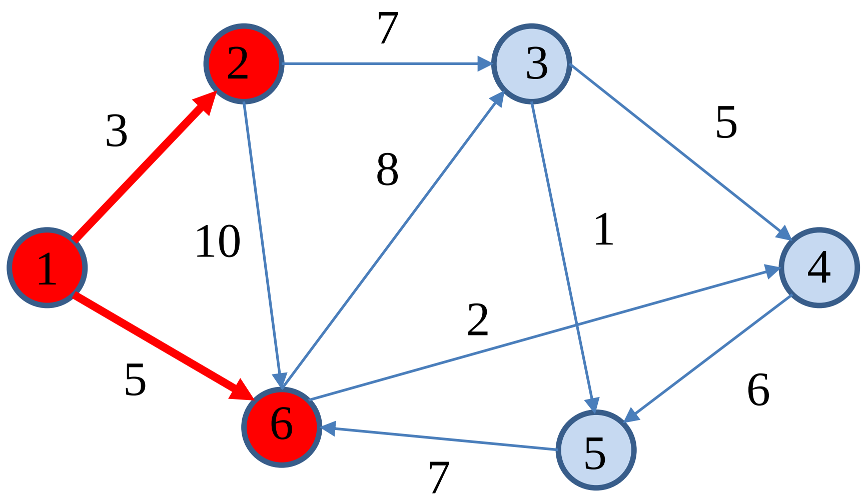
| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |
| 2 | {1,2} | {3,4,5,6} | 2 | 3 | 0 | 3 | 10 | ∞ | ∞ | 5 |
| 3 | {1,2,6} | {3,4,5} | 6 | 5 | 0 | 3 | 10 | 7 | ∞ | 5 |



# Expanding the vertex set w in stages

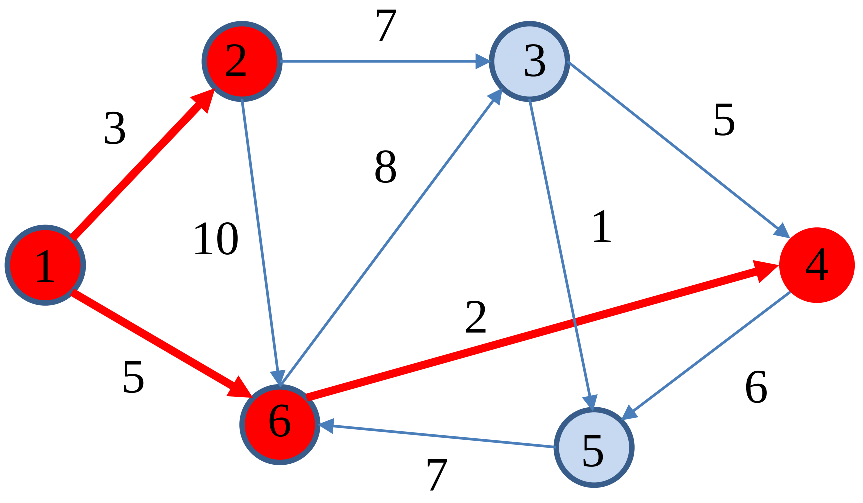
**W=4** is chosen for the fourth stage.

| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |
| 2 | {1,2} | {3,4,5,6} | 2 | 3 | 0 | 3 | 10 | ∞ | ∞ | 5 |
| 3 | {1,2,6} | {3,4,5} | 6 | 5 | 0 | 3 | 10 | 7 | ∞ | 5 |



# Expanding the vertex set w in stages

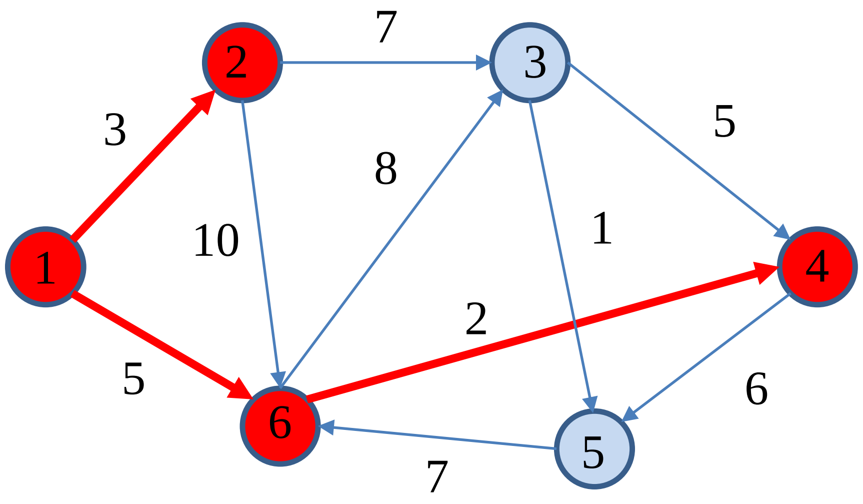
| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |
| 2 | {1,2} | {3,4,5,6} | 2 | 3 | 0 | 3 | 10 | ∞ | ∞ | 5 |
| 3 | {1,2,6} | {3,4,5} | 6 | 5 | 0 | 3 | 10 | 7 | ∞ | 5 |
| 4 | {1,2,6,4} | {3,5} | 4 | 7 | 0 | 3 | 10 | 7 | 13 | 5 |



# Expanding the vertex set w in stages

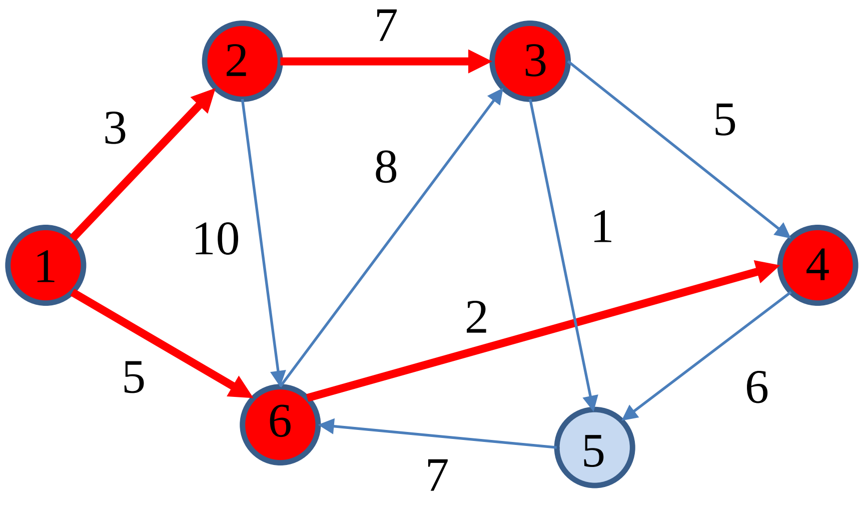
**W=3** is chosen for the fifth stage.

| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |
| 2 | {1,2} | {3,4,5,6} | 2 | 3 | 0 | 3 | 10 | ∞ | ∞ | 5 |
| 3 | {1,2,6} | {3,4,5} | 6 | 5 | 0 | 3 | 10 | 7 | ∞ | 5 |
| 4 | {1,2,6,4} | {3,5} | 4 | 7 | 0 | 3 | 10 | 7 | 13 | 5 |



# Expanding the vertex set w in stages

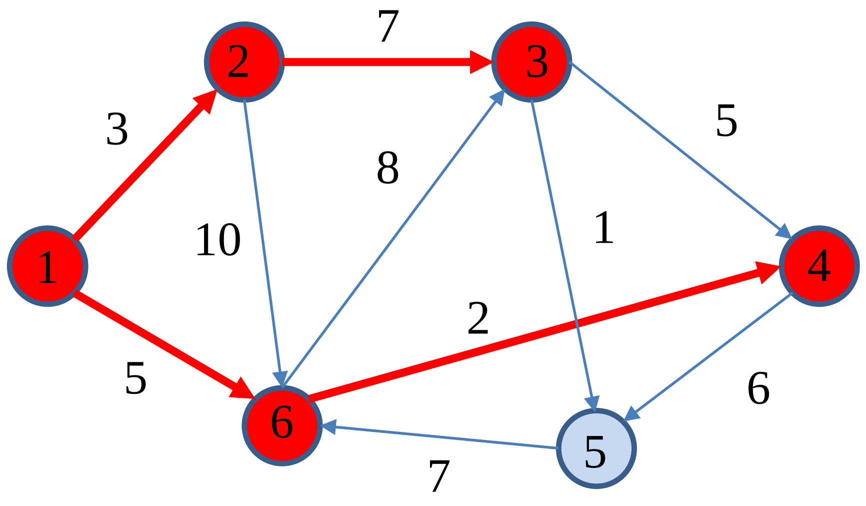
| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |
| 2 | {1,2} | {3,4,5,6} | 2 | 3 | 0 | 3 | 10 | ∞ | ∞ | 5 |
| 3 | {1,2,6} | {3,4,5} | 6 | 5 | 0 | 3 | 10 | 7 | ∞ | 5 |
| 4 | {1,2,6,4} | {3,5} | 4 | 7 | 0 | 3 | 10 | 7 | 13 | 5 |
| 5 | {1,2,6,4,3} | {5} | 3 | 10 | 0 | 3 | 10 | 7 | 11 | 5 |



# Expanding the vertex set w in stages

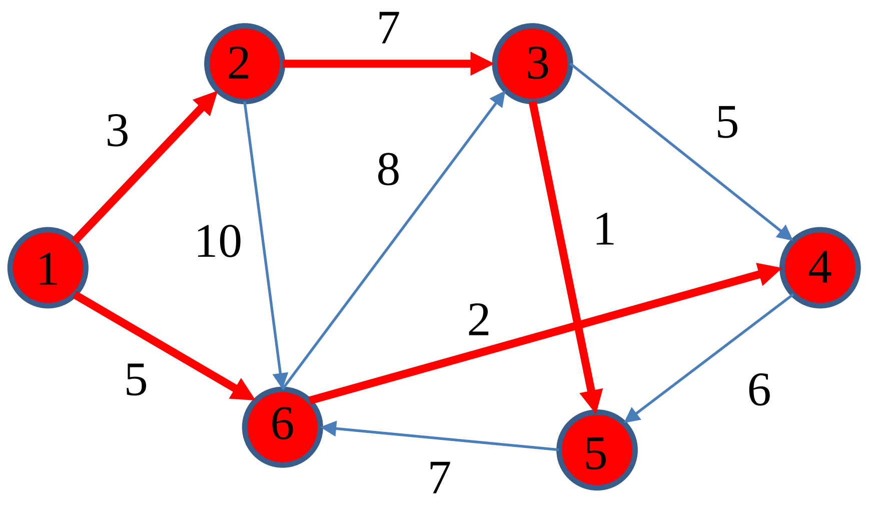
**W=5** is chosen for the sixth stage.

| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |
| 2 | {1,2} | {3,4,5,6} | 2 | 3 | 0 | 3 | 10 | ∞ | ∞ | 5 |
| 3 | {1,2,6} | {3,4,5} | 6 | 5 | 0 | 3 | 10 | 7 | ∞ | 5 |
| 4 | {1,2,6,4} | {3,5} | 4 | 7 | 0 | 3 | 10 | 7 | 13 | 5 |
| 5 | {1,2,6,4,3} | {5} | 3 | 10 | 0 | 3 | 10 | 7 | 11 | 5 |



# Expanding the vertex set w in stages

| Stage | W | V-W | w | Δ(w) | Δ(1) | Δ(2) | Δ(3) | Δ(4) | Δ(5) | Δ(6) |
|---|---|---|---|---|---|---|---|---|---|---|
| Start | {1} | {2,3,4,5,6} | - | - | 0 | 3 | ∞ | ∞ | ∞ | 5 |
| 2 | {1,2} | {3,4,5,6} | 2 | 3 | 0 | 3 | 10 | ∞ | ∞ | 5 |
| 3 | {1,2,6} | {3,4,5} | 6 | 5 | 0 | 3 | 10 | 7 | ∞ | 5 |
| 4 | {1,2,6,4} | {3,5} | 4 | 7 | 0 | 3 | 10 | 7 | 13 | 5 |
| 5 | {1,2,6,4,3} | {5} | 3 | 10 | 0 | 3 | 10 | 7 | 11 | 5 |
| 6 | {1,2,6,4,3,5} | {} | 5 | 11 | 0 | 3 | 10 | 7 | 11 | 5 |



## Dijkstra’s algorithm in pseudocode

```
// Δεδομένα
src  : αρχικός κόμβος
dest : τελικός κόμβος

// Πληροφορίες που κρατάμε για κάθε κόμβο v
W[u]     : 1 αν ο u είναι στο σύνολο W, 0 διαφορετικά
dist[u]  : ο πίνακας Δ
prev[u]  : ο προηγούμενος του v στο βέλτιστο μονοπάτι

// Αρχικοποίηση: W={} (εναλλακτικά μπορούμε και W={src})
for each vertex u in Graph
  dist[u] = INT_MAX    // infinity
  prev[u] = NULL
  W[u] = 0

dist[src] = 0
```

## Dijkstra’s algorithm in pseudocode

```
// Κυρίως αλγόριθμος
while true
    w = vertex with minimum dist[w], among those with W[w] = 0

    W[w] = 1
    if w == dest
        stop
        // optimal cost = dist[dest]
        // optimal path = dest <- prev[dest] <- ... <- src (inverse)

    for each neighbor u of w
        if W[u] == 1
            continue
        alt = dist[w] + weight(w,u)     // κόστος του src -> ... -> w
        if alt < dist[u]                // καλύτερο από πριν, update
            dist[u] = alt
            prev[u] = w
```

# Using a priority queue

- Finding the $w \notin W$ with **minumum** $\Delta[w]$ is slow
  - $O(n)$ time
- But we can use a **priority queue** for this!
  - We only keep vertices $w \notin W$ in the queue
  - They are compared based on their $\Delta[w]$
    (each $w$ has “priority” $\Delta[w]$)
- Careful when $\Delta[w]$ is modified!
  - Either use a priority queue that allows **updates**
  - Or insert multiple copies of each $w$ with **different priorities**
    - the queue might contain **already visited** vertices: ignore them

## Dijkstra’s algorithm with priority queue

```
// Δεδομένα
src  : αρχικός κόμβος
dest : τελικός κόμβος

// Πληροφορίες που κρατάμε για κάθε κόμβο u
W[u]     : 1 αν ο v είναι στο σύνολο W, 0 διαφορετικά
dist[u]  : ο πίνακας Δ
prev[u]  : ο προηγούμενος του v στο βέλτιστο μονοπάτι
pq       : Priority queue, εισάγουμε tuples {u,dist[u]}
           συγκρίνονται με βάση το dist[u]

// Αρχικοποίηση: W={} (εναλλακτικά μπορούμε και W={src})
prev[src] = NULL
dist[src] = 0
pqueue_insert(pq, {src,0})  // dist[src] = 0
```

# Dijkstra’s algorithm with priority queue

```
// Κυρίως αλγόριθμος
while pq is not empty
    w = pqueue_max(pq)  // w with minimal dist[u]
    pqueue_remove_max(pq)

    if exists(W[w])     // το w μπορεί να υπάρχει πολλές φορές στην ο
        continue        // δεν κάνουμε replace), και να είναι ήδη vis
    W[w] = 1
    if w == dest
        stop            // optimal cost/path same as before

    for each neighbor u of w
        if exists(W[u])
            continue
        alt = dist[w] + weight(w,u)     // cost of src->...->w->u
        if !exists(dist[u]) OR alt < dist[u]
            dist[u] = alt
            prev[u] = w
            pqueue_insert(pq, {u,alt})  // προαιρετικά: replace αν υπ

stop // pq άδειασε πριν βρούμε το dest => δεν υπάρχει μονοπάτι
```

## Notation

- $s \to d$
  - Direct step step from $s$ to $d$
- $s \xrightarrow{W} d$
  - Multiple steps $s \to \ldots \to d$
  - All intermediate steps belong to the set $W \subseteq V$
- $s \overset{W}{\Longrightarrow} d$
  - Shortest path among all $s \xrightarrow{W} d$
  - So $s \overset{V}{\Longrightarrow} d$ is the overall shortest one

## Proof of correctness

- We need to prove that $\Delta[u]$ is the **minimum distance to** $u$
  - **after** the algorithm finishes
- But it's much easier to reason **step by step**
  - we need a property that holds **at every step**
  - this is called an **invariant** (property that never changes)

# Proof of correctness

**Invariant of Dijkstra's algorithm**

- $\Delta[u]$ is the cost of the shortest path **passing only through** $W$
- And the shortest **overall** when $u \in W$

Formally:

1. For all $u \in V$ the path $s \overset{W}{\Longrightarrow} u$ has cost $\Delta[u]$
2. For all $u \in W$ the path $s \overset{V}{\Longrightarrow} u$ has cost $\Delta[u]$

Proof: **induction** on the **size of** $W$, for both (1), (2) together

# Proof of correctness

Base case $W = \{s\}$

- Trivial, the only path $s \xrightarrow{W} u$ is the direct one $s \to u$
- For (1): its cost is exactly $T[s,u] = \Delta[u]$
  - initial value of $\Delta[u]$
- For (2): the only $u \in W$ is $s$ itself

# Proof of correctness

Inductive case

- Assume $|W| = k$ and (1),(2) hold
- The algorithm
    - Updates $W$, adding a new vertex $w \notin W$
    - Updates $\Delta[u]$ for all neighbours $u$ of $w$
- Let $W$', $\Delta'$ be the values **after** the update
- Show that (1),(2) still hold for $W$', $\Delta'$

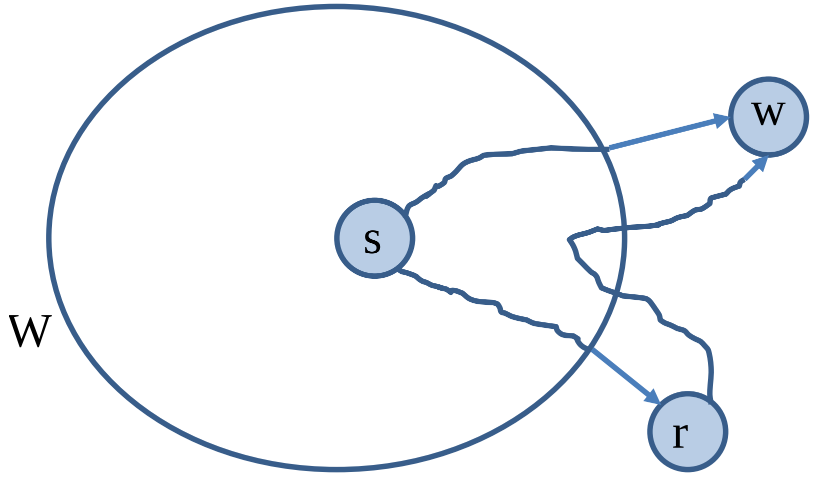
# Proof of correctness

We start showing that (2) still holds for $W$', $\Delta'$

- The interesting vertex is the $w$ we just added
  - Vertices $u \neq w$ are trivial from the induction hypothesis
- Show: $s \overset{V}{\Longrightarrow} w$ has cost $\Delta$'$[w]$
  - Note: $\Delta$'$[w] = \Delta[w]$ (we do not update $\Delta[w]$)
  - We already know that $s \overset{W}{\Longrightarrow} w$ has cost $\Delta[w]$ (ind. hyp)
  - So we just need to prove that there is **no better** path **outside** $W$

## Proof of correctness

- Assuming such path exists, let $r$ be its **first** vertex outside $W$
  - So the path $s \overset{W}{\Longrightarrow} r \overset{V}{\Longrightarrow} w$ has cost $c < \Delta[w]$
  - So the path $s \overset{W}{\Longrightarrow} r$ has cost at most $c < \Delta[w]$ (no negative weights!)
  - So $\Delta[r] < \Delta[w]$
- **Impossible!** We chose $w$ to be the one with min $\Delta[w]$



## Proof of correctness

It remains to show (1) for $W', \Delta'$

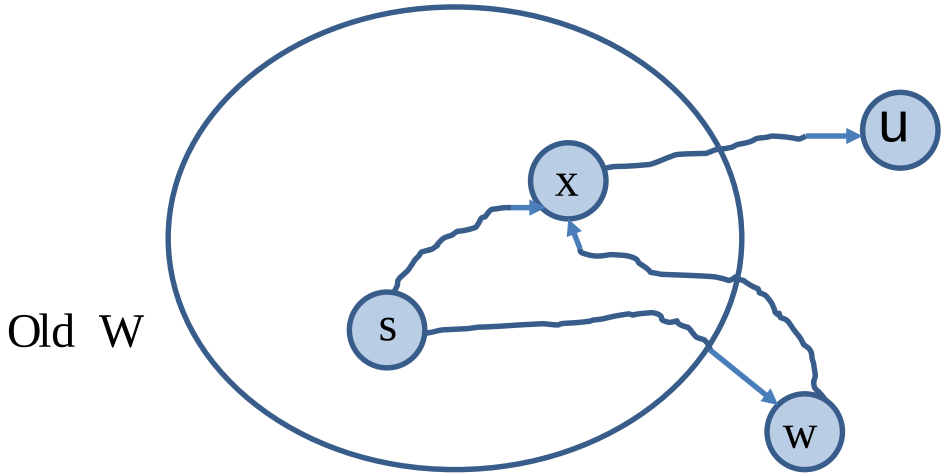
- Take some arbitrary $u$
  - Let $c$ be the cost of $s \overset{W'}{\Longrightarrow} u$
  - Show: $c = \Delta'[u]$
- Three cases for the optimal path $s \overset{W'}{\Longrightarrow} u$
- Case 1: the path does not pass through $w$
  - So it is of the form $s \overset{W}{\Longrightarrow} u$
  - This path has cost $\Delta[u]$ (induction hypothesis)
  - No update: $\Delta'[u] = \Delta[u] = c$

# Proof of correctness

- Case 2: $w$ is right before $u$
  - So the path is of the form $s \overset{W}{\Longrightarrow} w \to u$
  - The cost of $s \overset{W}{\Longrightarrow} w$ is $\Delta[w]$ (induction hypothesis)
  - So $c = \Delta[w] + T[w, u]$
  - So the algorithm will set $\Delta'[u] = \Delta[w] + T[w, u]$ when updating the neighbours of $w$
  - So $c = \Delta'[u]$

## Proof of correctness

- Case 3: some other $x$ appears after $w$ in the path
  - So the path is of the form $s \overset{W}{\Longrightarrow} w \to x \overset{W}{\Longrightarrow} u$
  - But the path $s \overset{W}{\Longrightarrow} w \to x$ is no shorter than $s \overset{W}{\Longrightarrow} x$
    - From the induction hypothesis for $x \in W$
  - So $s \overset{W}{\Longrightarrow} x \to u$ is also optimal, reducing to case 1!



## Complexity

Without a priority queue:

- Initialization stage: loop over vectices: $O(n)$
- The while-loop adds one vertex every time: $n$ iterations
- Finding the new vertex loops over vertices: $O(n)$
  - same for updating the neighbours
- So total $O(n^2)$ time

## Complexity

With a priority queue:

- Initialization stage: loop over vectices, so $O(n)$
- Count the number of **updates** (steps in the **inner** loop)
  - Once for every neighbour of every node: $e$ total
  - Each update is $O(\log n)$ (at most $n$ elements in the queue)
- Total $O(e \log n)$
  - Assuming a connected graph ($e \geq n$)
  - And an implementation using adjacency lists
- Only good for **sparse** graphs!
  - But $O(n \log n)$ can be hugely better than $O(n^2)$

## The all-pairs shortest path problem

- Find the shortest path between all pairs $s, d$
- **Floyd-Warshall** algorithm
- Any weights
  - Even negative
  - But no **negative loops** (why?)

## The all-pairs shortest path problem

Main idea

- At each step we compute the shortest path through a **subset of vertices**
  - Similarly to $W$ in Dijkstra
  - But now the set at step $k$ is $W_k = \{1, \ldots, k\}$
    - Vectices are numbered in any order
- Step $k$: the cost of $i \overset{W_k}{\Longrightarrow} j$ is $A_k[i, j]$
  - Similar to $\Delta$ in Dijstra (but for all **pairs** $i, j$ of nodes)

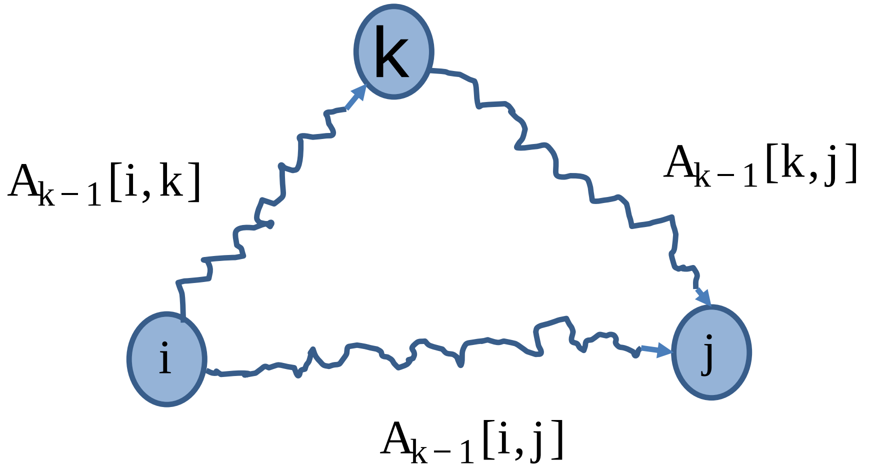
## Floyd-Warshall algorithm

- The algorithm at each step computes $A_k$ from $A_{k-1}$
- Initial step $k = 0$
  - Start with $A_0[i, j] = T[i, j]$
  - Only direct paths are allowed

## Floyd-Warshall algorithm

$k$-th iteration: the optimal $i \xRightarrow{W_k} j$ either **passes thorugh** $k$ or not.

$$A_k[i,j] = \min \begin{cases} A_{k-1}[i,j] \\ A_{k-1}[i,k] + A_{k-1}[k,j] \end{cases}$$



## Floyd-Warshall algorithm in pseudocode

```
void floyd_warshall() {

    for (int i = 0; i <= size-1; i++)
        for (int j = 0; j <= size-1; j++)
            A[i][j] = weight(i, j)

    for (int i = 0; i <= size-1; i++)
        A[i][i] = 0;

    for (int k = 0; k <= size-1; k++)
        // Compute A_k from A_{k-1}
        for (int i = 0; i <= size-1; i++)
            for (int j = 0; j <= size-1; j++)
                if (A[i][k] + A[k][j] < A[i][j])
                    A[i][j] = A[i][k] + A[k][j]
}
```

`A` is the **current** $A_k$ at every step $k$.

## Complexity

- Three simple loops of $n$ steps
- So $O(n^3)$
- **Not** better than $n$ executions of Dijkstra in **complexity**
  - But much simpler
  - Often faster in practice
  - And works for **negative** weights

## Readings

- T. A. Standish. *Data Structures , Algorithms and Software Principles in C.* Chapter 10
- A. V. Aho, J. E. Hopcroft and J. D. Ullman. *Data Structures and Algorithms.* Chapters 6 and 7

# Linked Data Representations

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Under construction

## Linked Data Representations

- Linked data representations such as **lists, stacks, queues, sets** and **trees** are very useful in Computer Science and applications. E.g., in Databases, Artificial Intelligence, Graphics, Web, Hardware etc.
- We will cover all of these data structures in this course.
- Linked data representations are useful when it is difficult to predict the size and shape of the data structures needed.

## Levels of Data Abstraction

## Pointers

- The best way to realize linked data representations is using pointers.
- A **pointer (δείκτης)** is a variable that references a unit of storage.
- Graphical notation (α is a pointer to β):

## Pointers in C

```c
typedef int *IntegerPointer;
IntegerPointer A, B;
/* the declaration int *A, *B has the same effect */

A = (IntegerPointer) malloc(sizeof(int));
B = (int*) malloc(sizeof(int));
```

The above code results in the following situation:

## `typedef`

- C provides a facility called `typedef` for creating new data type names.
- `typedef`s are useful because:
  - They help to organize our data type definitions nicely.
  - They provide better documentation for our program.
  - They make our program portable.

## Pointers in C (cont’d)

- The previous statements first define a new data type name `IntegerPointer` which consists of a pointer to an integer.
- Then they define two variables `A` and `B` of type `IntegerPointer`.
- Then they allocate two blocks of storage for two integers and place two pointers to them in `A` and `B`.
- The `void` pointer returned by `malloc` is casted into a pointer to a block of storage holding an integer. You can omit this casting and your program will still work correctly.

## `malloc`

- `void *malloc(size_tsize)` is a function of the standard library `stdlib`.
- `malloc` returns a pointer to space for an object of size `size`, or `NULL` if the request cannot be satisfied. The space is obtained from the **heap** and is uninitialized.
- This is called **dynamic storage allocation (δυναμική δέσμευση μνήμης)**.
- `size_t` is the unsigned integer type returned by the `sizeof` operator.

## Program Memory

## The Operator *

```
*A = 5;
*B = 17;
```

- The unary operator `*` (**τελεστής αναφοράς**) on the left side of the assignment designates the storage location to which the pointer `A` refers. We call this **pointer dereferencing**.
- The unary operator `&` (**τελεστής διεύθυνσης**) gives the address of some object (in the above diagram the address of variable `X`).

## The Operator &

```
int X = 3;
```

The unary operator `&` (τελεστής διεύθυνσης) gives the address of some object (in the above diagram the address of variable `X`).

## Pointers in C (cont'd)

Consider again the following statements:

```
int *A, *B;
*A = 5;
*B = 17;
```

**Question:** What happens if we now execute `B = 20`?

## Pointers in C (cont’d)

```c
int *A, *B;
*A = 5;
*B = 17;
```

**Answer:** We have a **type mismatch error** since `20` is an integer but `B` holds a pointer to integers. The compiler `gcc` will give a **warning** : “assignment makes pointer from an integer without a cast.”

## Pointers in C (cont’d)

Suppose we start with the diagram below:

# Pointers in C (cont'd)

**Question:** If we execute `A = B;` which one of the following two diagrams results?

## Pointers in C (cont’d)

```
A = B;
```

**Answer:** The right diagram. Now `A` and `B` are called **aliases** because they name the same storage location. Note that the storage block containing 5 is now **inaccessible**. (Some languages such as Lisp have a **garbage collection** facility for such storage.)

## Recycling Used Storage

We can reclaim the storage space to whichApoints by using the **reclamation function** free:

```
free(A);
A = B;
```

# Dangling Pointers

- Let us now consider the following situation:
- **Question:**
    - Suppose now we call `free(B)`.
    - What is the value of `*A + 3` then?

## Dangling Pointers (cont’d)

- **Answer:** We do not know. Storage location `A` now contains a **dangling pointer** and should not be used.
- It is reasonable to consider this to be a **programming error** even though the compiler or the runtime system will not catch it.

## NULL

- There is a special address denoted by the constant `NULL` which is not the address of any node. The situation that results after we execute `A = NULL` is shown graphically below:
- Now we cannot access the storage location to which `A` pointed to earlier. So something like `*A = 5` will give us “segmentation fault”.
- `NULL` is automatically considered to be a value of any pointer type that can be defined in C. `NULL` is defined in the standard input/output library stdio.h and has the value 0.

## Pointers and Function Arguments

- Let us suppose that we have a sorting routine that works by exchanging two out-of-order elements using a function `Swap`.
- **Question:** Can we call `Swap(A,B)` where the `Swap` function is defined as follows?

```
void Swap(int X, int Y) {
    int Temp;
    Temp = X;
    X = Y;
    Y = Temp;
}
```

## Pointers and Function Arguments (cont’d)

- **Answer:** No. C passes arguments to functions **by value** (**κατ’ αξία**) therefore `Swap` can’t affect the arguments `A` and `B` in the routine that called it. `Swap` only swaps **copies** of `A` and `B`.
- The way to have the desired effect is for the calling program to pass **pointers** to the values to be changed:

```
Swap(&A, &B);
```

## The Correct Function Swap

```
void Swap(int *P, int *Q) {
    intTemp;
    Temp = *P;
    *P = *Q;
    *Q = Temp;
}
```

## In Pictures

## Linked Lists

- A **linear linked list** (or **linked list** ) is a sequence of nodes in which each node, except the last, links to a successor node.
- We usually have a pointer variableLcontaining a pointer to the first node on the list.
- The link field of the last node containsNULL.
- **Example:** a list representing a flight

## Diagrammatic Notation for Linked Lists

## Declaring Data Types for Linked Lists

The following statements declare appropriate data types for our linked list:

```c
typedef char AirportCode[4];

typedef structNodeTag {
    AirportCode Airport;
    struct NodeTag *Link;
} NodeType;

typedef NodeType* NodePointer;
```

We can now define variables of these data types:

```c
NodePointer L;
// or equivalently
NodeType *L;
```

## Structures in C

- A **structure (δομή)** is a collection of one or more variables possibly of different types, grouped together under a single name.
- The variables named in a structure are called **members (μέλη)**.
- In the previous structure definition, the nameNodeTagis called a **structure tag** and can be used subsequently as a shorthand for the part of the declaration in braces.

## Example

Given the previous `typedef`s, what would be the output of the following piece of code:

```
AirportCodeC;
NodePointerL;

strcpy(C, "BRU");
printf("%s\n", C);

L = (NodePointer) malloc(sizeof(NodeType));
strcpy(L->Airport, C);
printf("%s\n", L->Airport);
```

## The Function strcpy

- The function `strcpy(s, ct)` copies string `ct` to string `s`, including `\0`. It returns `s`.
- The function is defined in header file string.h.

## Accessing Members of a Structure

- To access a member of a structure, we use the **dot notation** as follows:

```
var.member
```

- To access a member of a structure pointed to by a pointer `P`, we can use the notation

```
(*P).member
```

  or the equivalent **arrow notation**

```
P->member
```

## Question

Why didn’t I write `C = "BRU"` and `L->Airport = "BRU"` in the previous piece of code?

## Answer

The assignment `C = "BRU"` assigns to variable `C` a **pointer to the character array** `"BRU"`. This would result in an error (type mismatch) because `C` is of type `AirportCode`.

Similarly for the second assignment.

## Example

Given the previous `typedef`s, what does the following piece of code do?

```
NodePointer L, M;

L = (NodePointer) malloc(sizeof(NodeType));
strcpy(L->Airport, "DUS");

M = (NodePointer) malloc(sizeof(NodeType));
strcpy(M->Airport, "ORD");

L->Link = M;
M->Link = NULL;
```

## Answer

The piece of code on the previous slide constructs the following linked list of two elements:

## Inserting a New Second Node on a List

**Example:** adding one more airport to our list representing a flight

## Inserting a New Second Node on a List

```
voidInsertNewSecondNode(void) {
    NodeType*N;
    N = (NodeType*)malloc(sizeof(NodeType));
    strcpy(N->Airport,"BRU");
    N->Link = L->Link;
    L->Link = N;
}
```

## Inserting a New Second Node on a List (cont'd)

Let us execute the previous function step by step:

```
N = (NodeType*) malloc(sizeof(NodeType));
```

```
strcpy(N->Airport,"BRU");
```

## Inserting a New Second Node on a List (cont’d)

```
N->Link = L->Link;
```

## Inserting a New Second Node on a List (cont’d)

```
L->Link = N;
```

## Comments

In the function `InsertNewSecondNode`, variable `N` is **local** . Therefore it vanishes after the end of the function execution. However, **the dynamically allocated node remains in existence** after the function has terminated.

## Searching for an Item on a List

Let us now define a function which takes as input an airport code `A` and a pointer to a list `L` and returns a pointer to the first node of `L` which has that code. If the code cannot be found, then the function returns `NULL`.

## Searching for an Item on a List

```
NodeType*ListSearch(char *A,NodeType*L) {
    NodeType *N;
    N = L;
    while (N != NULL){
        if (strcmp(N->Airport, A)==0){
            return N;
        } else {
            N = N->Link;
        }
    }
    return N;
}
```

## Comments

The function `strcmp(cs, ct)` compares string `cs` to string `ct` and returns

- a negative integer if `cs` precedes `ct` alphabetically,
- 0 if `cs` and `ct` are equal (same contents, not necessarily same address), and
- a positive integer if `cs` follows `ct` alphabetically

(using the ASCII codes of the characters of the strings)

## Comments (cont’d)

Let us assume that we have the list below and we are searching for item “ORD”. When the initialization statement `N = L` is executed, we have the following situation:

## Comments (cont'd)

Later on, inside the `while` loop, the statement `N = N->Link` is executed and we have the following situation:

## Comments (cont'd)

Then, the `if` inside the `while` loop is executed and the value of `N` is returned. Assuming that we did not find “ORD” here, the statement `N = N->Link` is again executed and we have the following situation:

## Comments (cont’d)

Then, the `while` loop is executed one more time and the statement `N = N->Link` results in the following situation:

## Comments (cont’d)

Then, we exit from the `while` loop and the statement `return N` returns `NULL`:

## Deleting the Last Node of a List

Let us now write a function to delete the last node of a list `L`.

- If `L` is **empty**, there is nothing to do.
- If `L` has **one node**, then we need to dispose of the node's storage and then set `L` to be the empty list.
- If `L` has **two or more nodes** then we can use a pair of pointers to implement the required functionality as shown on the next slides.

## Deleting the Last Node of a List (cont'd)

- Note that we need to pass the **address** of `L` as an actual parameter in the form of `&L` enabling us to change the contents of `L` inside the function.
- Therefore the corresponding formal parameter of the function will be a **pointer to a pointer** to `NodeType`.

## Deleting the Last Node of a List

```
void DeleteLastNode(NodeType**L) {
    NodeType *PreviousNode, *CurrentNode;

    if (*L != NULL) {
        if ((*L)->Link == NULL) {
            free(*L);
            *L = NULL;

        } else {
            PreviousNode= *L;
            CurrentNode = (*L)->Link;

            while (CurrentNode->Link != NULL) {
                PreviousNode = CurrentNode;
                CurrentNode = CurrentNode->Link;
            }
            PreviousNode->Link = NULL;
            free(CurrentNode);
        }
    }
}
```

## Comments

When we advance the pointer pair to the next pair of nodes, the situation is as follows:

## Why **?

- This is for the case that `L` has one node only.
- Then, the value of pointer `L` must be set to `NULL` in the function `DeleteLastNode`.
- This can only be done by passing `&L` in the call of the function.

## Inserting a New Last Node on a List

```c
void InsertNewLastNode(char *A, NodeType**L) {
    NodeType *N, *P;

    N = (NodeType*) malloc(sizeof(NodeType));
    strcpy(N->Airport, A);

    N->Link = NULL;
    if (*L == NULL) {
        *L = N;

    } else {
        P = *L;
        while (P->Link != NULL)
            P = P->Link;
        P->Link = N;
    }
}
```

## Why **?

- This is for the case that `L` is empty.
- Then, the value of pointer `L` must be set topoint to the new node in the function `DeleteLastNode`.
- This can only be done by passing `&L` in the call of the function.

## Question

- Assume now that we have a pointer `Tail` pointing to the last element of a linked list.
- How would the operations of deleting the last node of a list or inserting a new last node on a list change to exploit the pointer `Tail`?

## Printing a List

```
void PrintList(NodeType*L) {
    NodeType *N;

    printf("(");

    N = L;
    while (N != NULL) {
        printf("%s", N->Airport);

        N = N->Link;
        if (N != NULL)
            printf(",");
    }

    printf(")\n");
}
```

## The Main Program

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

typedef charAirportCode[4];
typedef struct NodeTag{
    AirportCodeAirport;
    struct NodeTag *Link;
} NodeType;
typedef NodeType *NodePointer;

/* function prototypes */
void InsertNewLastNode(char *, NodeType**);
void DeleteLastNode(NodeType**);
NodeType *ListSearch(char*, NodeType*);
void PrintList(NodeType*);
```

## The Main Program (cont’d)

```
int main(void) {
    NodeType *L;

    L = NULL;
    PrintList(L);

    InsertNewLastNode("DUS", &L);
    InsertNewLastNode("ORD", &L);
    InsertNewLastNode("SAN", &L);
    PrintList(L);

    DeleteLastNode(&L);
    PrintList(L);

    if (ListSearch("DUS", L) != NULL) {
        printf("DUS is an element of the list\n");
    }
}

// Code for functions InsertNewLastNode, PrintList,
// ListSearch and DeleteLastNodegoes here.
```

## Linked Lists vs. Arrays

- Compare the data structure linked list that we defined in these slides with arrays.
- What are the pros and cons of each data structure?

## Linked Lists vs. Arrays

- The **simplicity of inserting and deleting a node** is what characterizes linked lists. This operation is more involved in an array because all the elements of the array that follow the affected element need to be moved.
- Linked lists are **not appropriate for finding the $i$-th element of a list** because we have to follow $i$ pointers. In an array, the same functionality is implemented with one operation.
- Such discussion is important when we want to choose a data structure for solving a practical problem.

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C.* Chapter 2.
- (προαιρετικά) R.Sedgewick. *Αλγόριθμοι σε C* .Κεφάλαιο 3.

# Recursion

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Under construction

## Recursion

- Recursion is a **fundamental concept** of Computer Science.
- It usually help us to write simple and elegant solutions to programming problems.
- You will learn to program recursively by working with many examples to develop your skills.

## Recursive Programs

A **recursive program** is one that calls itself in order to obtain a solution to a problem.

The reason that it calls itself is to compute a solution to a **subproblem** that has the following properties:

- The subproblem is **smaller** than the problem to be solved.
- The subproblem can be solved **directly (as a base case)** or **recursively by making a recursive call**.
- The subproblem's solution can be **combined** with solutions to other subproblems to obtain a solution to the overall problem.

## Example

- Let us consider a simple program to add up all the squares of integers from m to n.
- An **iterative function** to do this is the following:

```
int SumSquares(int m, int n) {
  int i, sum;
  sum=0;
  for (i = m; i <= n; i++) sum += i*i;
  return sum;
}
```

## Recursive Sum of Squares

```
int SumSquares(int m, int n) {
  if (m < n) {
    return m*m + SumSquares(m+1, n);
  }
  else {
    return m*m;
  }
}
```

## Comments

- In the case that the range m:n contains more than one number, the solution to the problem can be found by adding (a) the solution to the smaller subproblem of summing the squares in the range m+1:n and (b) the solution to the subproblem of finding the square of m. (a) is then solved in the same way (recursion).
- We stop when we reach the **base case** that occurs when the range m:n contains just one number, in which case m==n.
- This recursive solution can be called **“going-up” recursion** since the successive ranges are m+1:n, m+2:n etc.

## Going-Down Recursion

```
int SumSquares(int m, int n) {
  if (m < n) {
    return SumSquares(m, n-1) + n*n;
  }
  else {
    return  n*n;
  }
}
```

## Recursion Combining Two Half-Solutions

```
int SumSquares(int m, int n) {
  int middle;
  if (m == n) {
    return m*m;
  }
  else {
    middle = (m + n) / 2;
  return;
  SumSquares(m,middle) + SumSquares(middle + 1,n);
  }
}
```

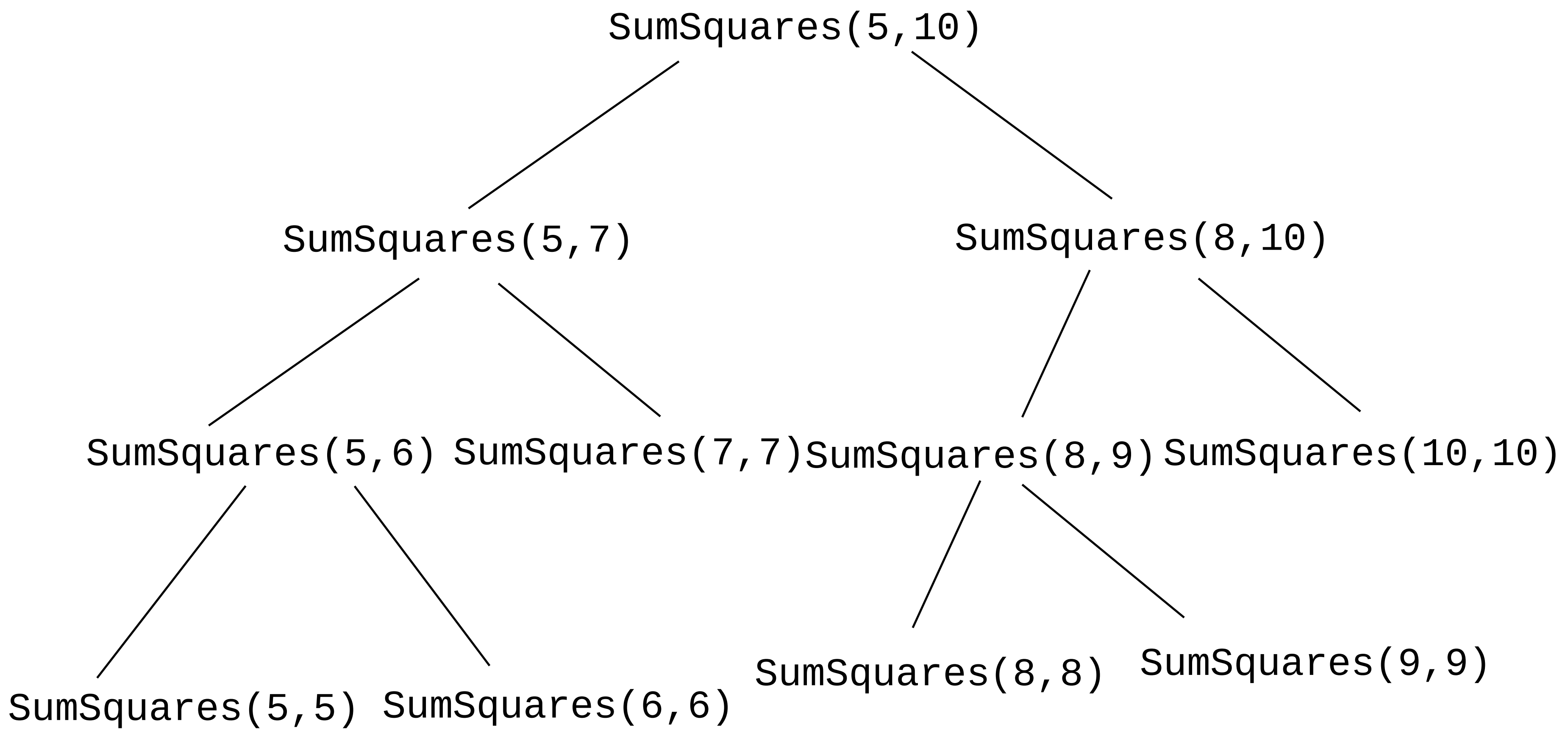
## Comments

- The **recursion** here says that the sum of the squares of the integers in the range m:n can be obtained by adding the sum of the squares of the left half range, m:middle, to the sum of the squares of the right half range, middle+1:n.
- We stop when we reach the **base case** that occurs when the range contains just one number, in which case m==n.
- The middle is computed by using **integer division** (operator /) which keeps the quotient and throws away the remainder.

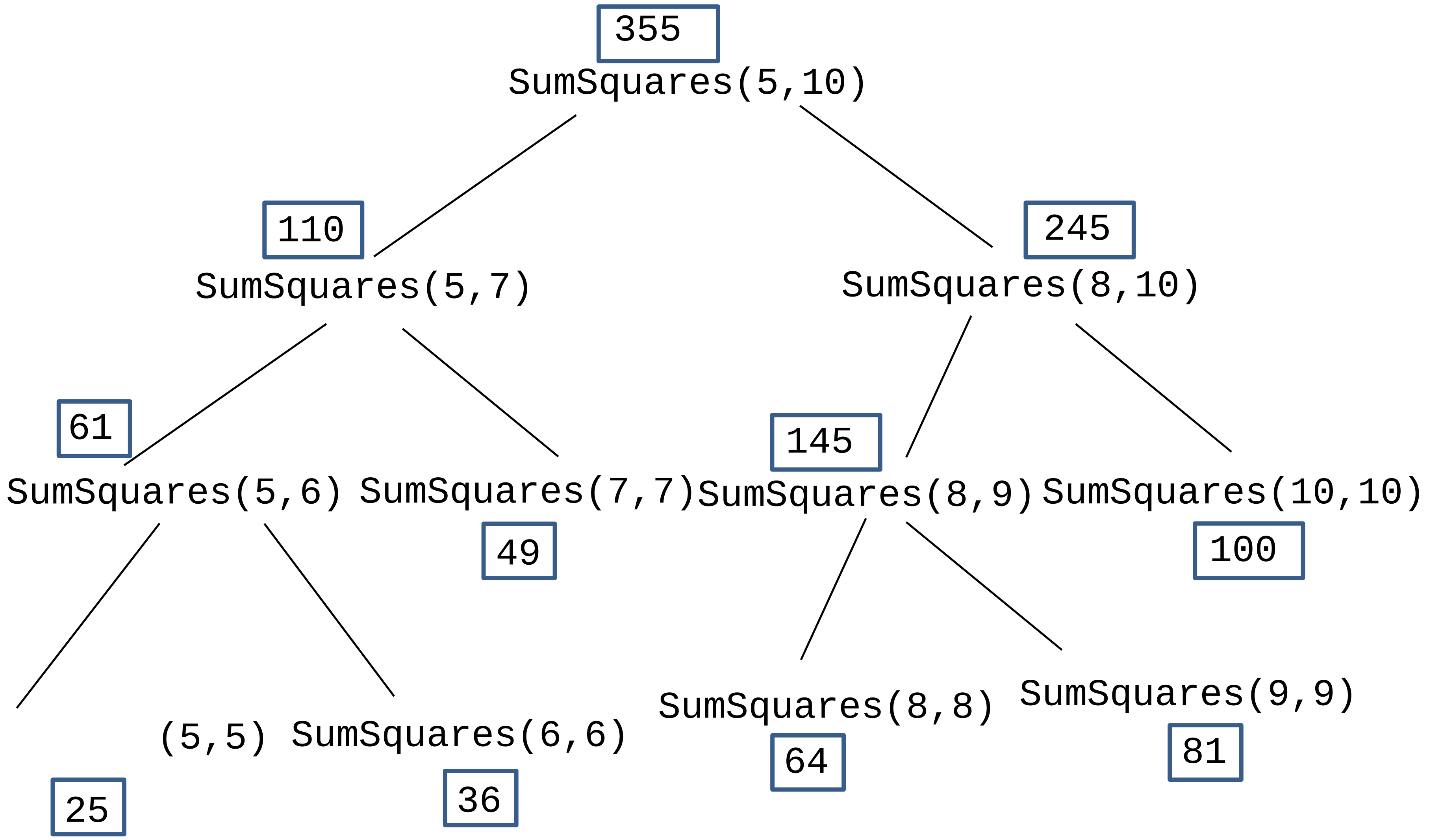
## Call Trees and Traces

- We can depict graphically the behaviour of recursive programs by drawing **call trees** or **traces**.

## Call Trees


SumSquares(5,10)
SumSquares(5,7)
SumSquares(8,10)
SumSquares(5,6)
SumSquares(7,7)
SumSquares(8,9)
SumSquares(10,10)
SumSquares(5,5)
SumSquares(6,6)
SumSquares(8,8)
SumSquares(9,9)

## Annotated Call Trees



## Traces

```
SumSquares(5,10) = SumSquares(5,7) + SumSquares(8,10)
= SumSquares(5,6) + SumSquares(7,7) + SumSquares(8,9) + SumSquares(10
= SumSquares(5,5) + SumSquares(6,6) + SumSquares(7,7) +
 + SumSquares(8,8) + SumSquares(9,9) + SumSquares(10,10)
= ((25 + 36) + 49)+((64 + 81) + 100)
= (61 + 49) + (145 + 100)
= (110 + 245)
= 355
```

## Computing the Factorial

- Let us consider a simple program to compute the factorial n! of n.
- An **iterative function** to do this is the following:

```
int Factorial(int n) {
  int i, f;
  f = 1;
  for (i = 2; i <= n; i++)
    f = f * i;
return f;
}
```

## Recursive Factorial

```
int Factorial(int n) {
  if (n == 1) {
    return 1;
  }
  else {
  return n * Factorial(n-1);
  }
}
```

## Computing the Factorial (cont’d)

- The previous program is a “going-down” recursion.
- Can you write a “going-up” recursion for factorial?
- Can you write a recursion combining two half-solutions?
- The above tasks do not appear to be easy.

## Computing the Factorial (cont’d)

- It is easier to first write a function Product(m,n) which **multiplies** together the numbers in the range m:n.
- Then $Factorial(n) = Product(1, n)$.

## Multiplying m:n Together Using Half-Ranges

```
int Product(int m, int n) {
  int middle;
  if (m == n) {
  return m;
  }
  else {
  middle = (m + n) / 2;
  return Product(m,middle) * Product(middle + 1,n);
  }
}
```

## Reversing Linked Lists

- Let us now consider the problem of reversing a linked list L.
- The type NodeType has been defined in the previous lecture as follows:

```
typedef char AirportCode[4];
typedef struct NodeTag {
  AirportCode Airport;
  struct NodeTag * Link;
} NodeType;
```

## Reversing a List Iteratively

- An **iterative function for reversing a list** is the following:

```
void Reverse(NodeType ** L){
  NodeType * R, * N, * L1;
  L1 = L1 * L;
  R = NULL;
  while (L1 != NULL) {
    N = L1;
    L1 = L1->Link;
    N->Link = R;
    R = N;
  }
  * L = R;
}
```

## Question

- If in our main program we have a list with a pointer A to its first node, how do we call the previous function?

## Answer

- We should make the following call:

```
Reverse(&A)
```

## Reversing Linked Lists (cont'd)

- A recursive solution to the problem of reversing a list L is found by partitioning the list into its **head** $Head(L)$ and **tail** $Tail(L)$ and then concatenating the reverse of $Tail(L)$ with $Head(L)$.

## Head and Tail of a List

- Let L be a list. $Head(L)$ is a list containing the first node of L. $Tail(L)$ is a list consisting of L's second and succeeding nodes.
- If $L == NULL$ then $Head(L)$ and $Tail(L)$ are not defined.
- If L consists of a single node then $Head(L)$ is the list that contains that node and $Tail(L)$ is $NULL$.

## Example

- Let $L = (SAN, ORD, BRU, DUS)$. Then
- $Head(L) = (SAN)$ and
- $Tail(L) = (ORD, BRU, DUS)$.

## Reversing Linked Lists (cont’d)

```
NodeType * Reverse(NodeType * L){
  NodeType * Head, * Tail;
  if (L == NULL) {
    return NULL;
  }
  else {
    Partition(L, &Head, &Tail);
    return Concat(Reverse(Tail), Head);
  }
}
```

## Reversing Linked Lists (cont’d)

```
void Partition(NodeType * L, NodeType ** Head, NodeType ** Tail) {
  if (L != NULL) {
    * Tail = L->Link;
    * Head = L;
    (* Head)->Link = NULL;
  }
}
```

## Reversing Linked Lists (cont’d)

```
NodeType *Concat(NodeType *L1, NodeType *L2) {
  NodeType * N;
  if (L1 == NULL) {
    return L2;
  }
  else {
    N = L1;
    while (N->Link != NULL)
      N = N->Link;
    N->Link = L2;
    return L1;
  }
}
```

## Infinite Regress

Let us consider again the recursive factorial function:

```
int Factorial(int n){
  if (n == 1) {
    return 1;
  }
  else {
    return n * Factorial(n-1);
  }
}
```

What happens if we call $Factorial(0)$?

## Infinite Regress (cont'd)

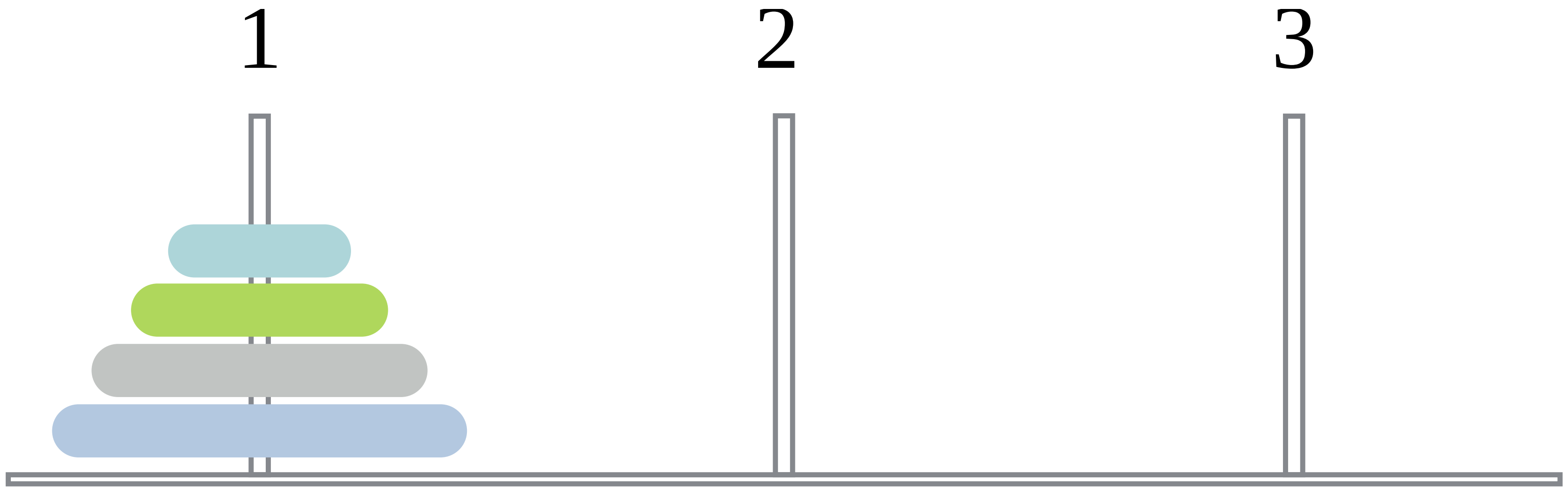
Factorial(0)= 0 * Factorial(-1)

= 0 * (-1) * Factorial(-2)

= 0 * (-1) * Factorial(-3)

- and so on, in an infinite regress.
- When we execute this function call, we get "Segmentation fault (core dumped)".

## The Towers of Hanoi

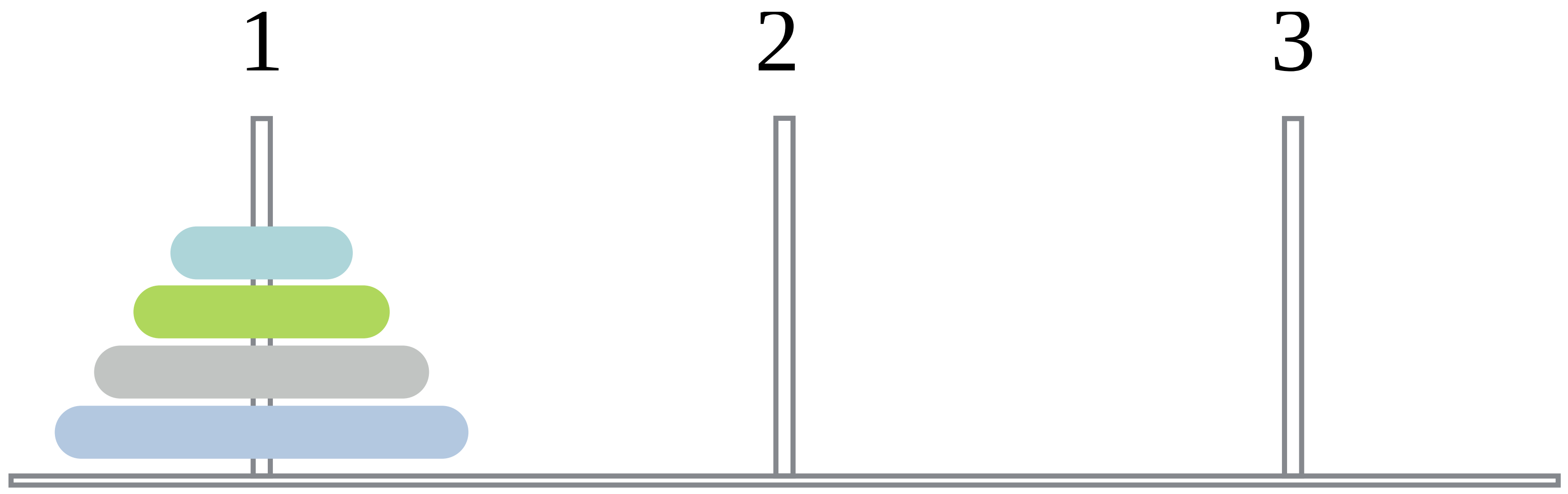


## The Towers of Hanoi (cont'd)

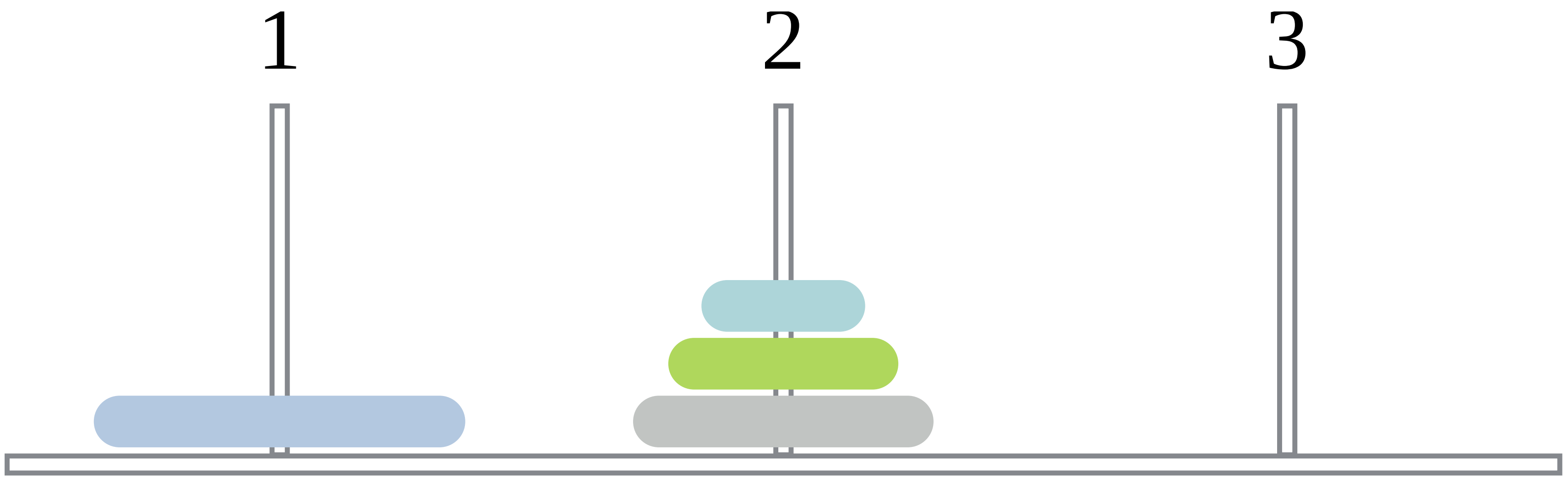
To Move 4 disks from Peg 1 to Peg 3:

- Move 3 disks from Peg 1 to Peg 2
- Move 1 disk from Peg 1 to Peg 3
- Move 3 disks from Peg 2 to Peg 3

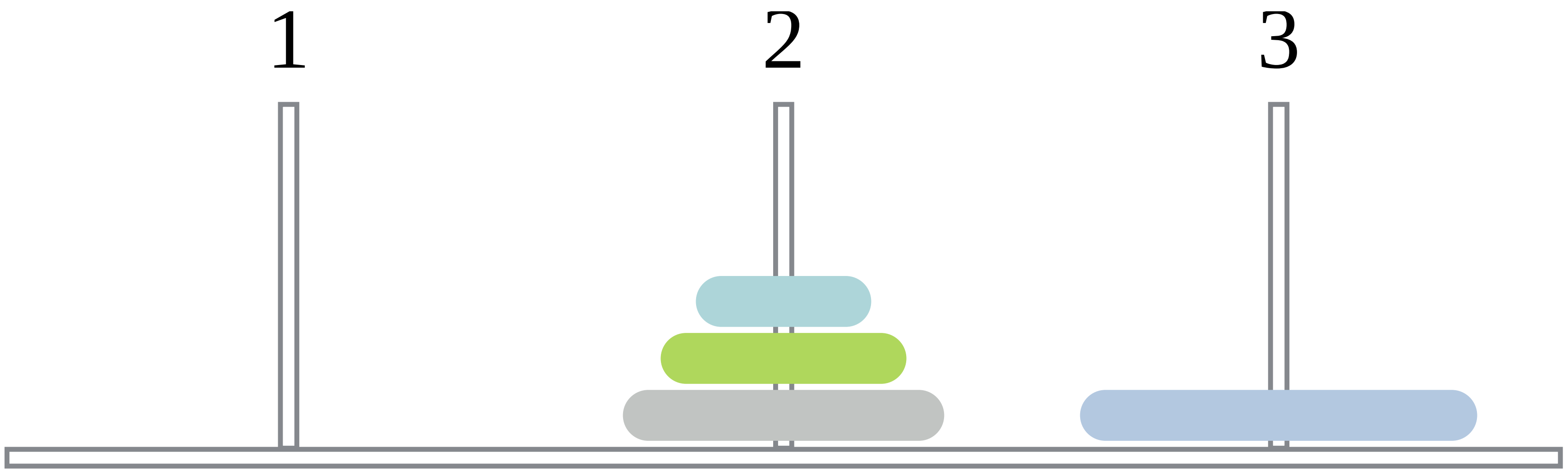
## Move 3 Disks from Peg 1 to Peg 2



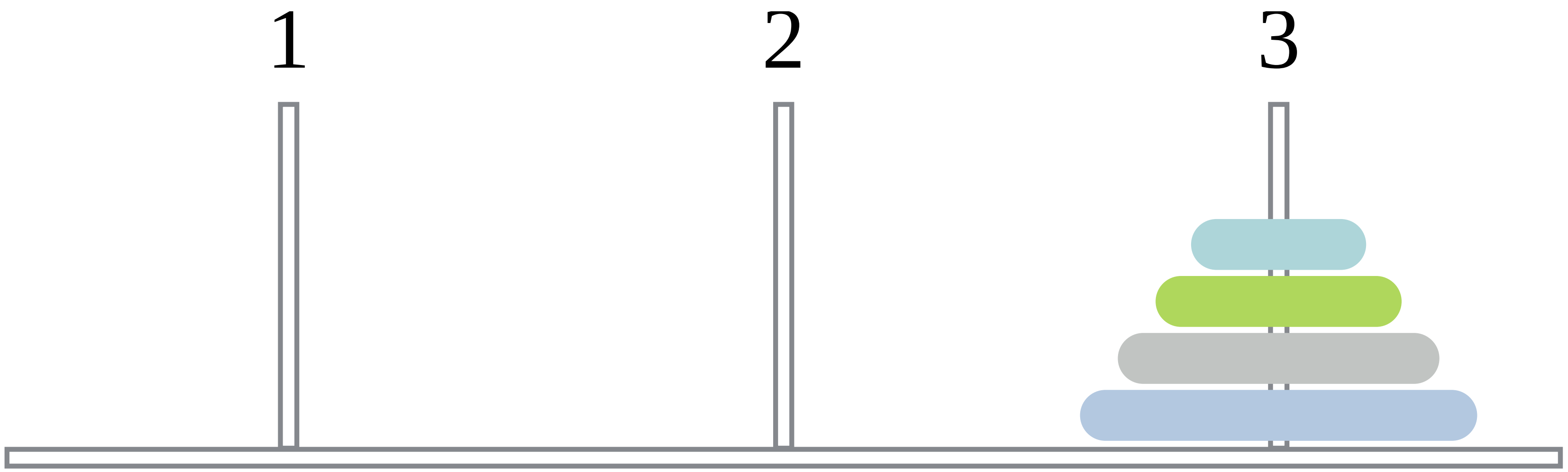
## Move 1 Disk from Peg 1 to Peg 3



## Move 3 Disks from Peg 2 to Peg 3



## Done!



## A Recursive Solution

```c
void MoveTowers(int n, int start, int finish, int spare) {
  if (n == 1) {
    printf("Move a disk from peg %1d to peg %1d\n", start, finish);
  }
  else {
    MoveTowers(n-1, start, spare, finish);
    printf("Move a disk from peg %1d to peg %1d\n", start, finish);
    MoveTowers(n-1, spare, finish, start);
  }
}
```

## Analysis

Let us now compute the **number of moves** L(n) that we need as a function of the number of disks n:

$$L(1) = 1\, L(n) = L(n-1) + 1 + L(n-1) = 2 * L(n-1) + 1, n > 1$$

The above are called **recurrence relations**. They can be solved to give:

$$L(n) = 2n - 1$$

## Analysis (cont’d)

- Techniques for solving recurrence relations are taught in the Algorithms and Complexity course.
- The running time of algorithm MoveTowers is **exponential** in the size of the input.

## Readings

- T. A. Standish. *Data structures, algorithms and software principles in C*.
- Chapter 3.
- (προαιρετικά) R. Sedgewick. *Αλγόριθμοι σε C*. Κεφ. 5.1 και 5.2.

# Modularity and Data Abstraction

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Under construction

## Procedural Abstraction

- When programs get large, certain **disciplines of structuring** need to be followed rigorously. Otherwise, the programs become complex, confusing and hard to debug.
- In your first programming course you learned the benefits of **procedural abstraction (διαδικαστική αφαίρεση).** When we organize a sequence of instructions into a function F(x1, …, xn), we have a named unit of action.
- When we later on use this function F, we only need to know **what** the function does, not **how** it does it.

## Procedural Abstraction (cont’d)

**Separating the what from the how** is an act of **abstraction (αφαίρεση)**. It provides two benefits:

- Ease of use
- Ease of modification

## Information Hiding

- In your first programming course, you have also learned the benefits of having **locally defined variables**.
- This is an instance of **information hiding (απόκρυψη πληροφορίας)**.
- It has the advantage that local variables do not interfere with identically named variables outside the function.
- Abstraction and information hiding in a programming language are greatly enhanced with the concept of **module (ενότητα).**

## Modules and Abstract Datatypes

- A **module** is a unit of organization of a software system that packages together a collection of entities (such as **data** and **operations**) and that carefully controls what external users of the module can see and use.
- Modules have ways of hiding things inside their boundaries to prevent external users from accessing them. This is called **information hiding**.
- **Abstract data types (αφαιρετικοί τύποι δεδομένων, ADTs)** are collections of objects and operations that present well defined **interfaces (διεπαφές)** to their users, meanwhile hiding the way they are represented in terms of lower-level representations.
- Modules can be used to **implement abstract data types**.

## Modules (cont’d)

Many modern programming languages offer **modules** that have the following important features:

- They provide a way of grouping together related data and operations.
- They provide clean, well-defined interfaces to users of their services.
- They hide internal details of operation to prevent interference.
- They can be separately compiled.

## Modules (cont’d)

- Modules are an important tool for **“dividing and conquering”** a large software task by combining separate components that interact cleanly.
- They ease **software maintenance (συντήρηση λογισμικού)** by allowing changes to be made locally.

## Encapsulation

- When we have features like modules in programming languages, we use the term **encapsulation** (**ενθυλάκωση**, the hidden local entities are **encapsulated** and a module is a **capsule**).

## Modules in C

- By means of careful use of **header files,** we can arrange for separately compiled C program files to have the above four properties of modules.
- In this way C modules are similar to **packages** or **modules** in other languages such as Modula-2 and Ada.

## Modules in C (cont'd)

- A C module M consists of two files MInterface.h and MImplementation.c that are organized as follows.
- The file Minterface.h:

```
/*------<the text for the file MInterface.h starts here>---------- */
(declarations of entities visible to external users of the module)
/*--------------<end of file MInterface.h>-------------------------*/
```

## Modules in C (cont'd)

- The file MImplementation.c:

```
-   /*-------<the text for the file Mimplementation.c starts here>---
#include <stdio.h>
#include "MInterface.h"
  (declarations of entities private to the module plus the)
  (complete declarations of functions exposed by the module)
/*---------------<end of file MImplementation.c>--------------------*
```

## The Interface file

- MInterface.h is the **interface** file.
- It declares all the entities in the module that are **visible** to (and therefore usable by) the external users of the module.
- Such visible entities include **constants, typedefs, variables** and **functions**. Only the prototype of each visible function is given (and only the argument types, not the argument names).
- The book by Standish recommends that declarations of functions in the interface file are **“extern”** declarations. This is not necessary so we will not follow it.

## The Implementation File

- MImplementation.c is the **implementation** file.
- It contains all the **private entities** in the module, that are not visible to the outside.
- It contains the **full declarations and implementations** of functions whose prototypes have been given in the interface file.
- It **includes** (via #include) the user interface file.

## The Main Program

- A **main program (client program)** that uses two modules A and B is organized as follows:

```
#include <stdio.h>
#include "ModuleAInterface.h"
#include "ModuleBInterface.h"
  (declarations of entities used by the main program)
int main(void) {
  (statements to execute in the main program)
}
```

## Separate Compilation

We can compile the module and the client program **separately:**

```
gcc -c MImplementation.c -o M.o

gcc -c ClientProgram.c -o ClientProgram.o

gcc M.o ClientProgram.o --o ClientProgram.exe
```

- With the first two commands, we compile the C files to produce **object files**. Then, the object files are **linked** to produce the final executable.

## Priority Queues – An Abstract Data Type

- A **priority queue** is a container that holds some prioritized items. For example, a list of jobs with a deadline for processing each one of them.
- When we remove an item from a priority queue, we always get the item with highest priority.

## Defining the ADT Priority Queue

A **priority queue** is a finite collection of items for which the following operations are defined:

- **Initialize** the priority queue, *PQ*, to the empty priority queue.
- Determine whether or not the priority queue, *PQ*, is **empty**.
- Determine whether or not the priority queue, *PQ*, is **full**.
- **Insert** a new item, *X*, into the priority queue, *PQ*.
- If *PQ* is non-empty, **remove** from *PQ* an item *X* of highest priority in *PQ*.

## A Priority Queue Interface File

```
/* this is the file PQInterface.h */
#include "PQTypes.h"
/* defines types PQItem and PriorityQueue */
void Initialize (PriorityQueue *);
int Empty (PriorityQueue *);
int Full (PriorityQueue *);
void Insert (PQItem, PriorityQueue *);
PQItem Remove (PriorityQueue *);
```

## Sorting Using a Priority Queue

Let us now define an array A to hold ten items of type PQItem, where PQItems have been defined to be integer values, such that bigger integers have greater priority than smaller ones:

```
typedef int PQItem;
typedef PQItem SortingArray[10];
SortingArray A;
```

We can now use a priority queue to sort the array A.

We can successfully use the ADT priority queue whose interface was given earlier **without having to know any details of its implementation.**

## Sorting Using a Priority Queue (cont’d)

```
/* this is the main program */
#include <stdio.h>
#include "PQInterface.h"
typedef PQItem SortingArray[MAXCOUNT];
/* Note: MAXCOUNT is 10 */
void PriorityQueueSort(SortingArray A) {
    int i;
    PriorityQueue PQ;
    Initialize(&PQ);
    for (i = 0; i < MAXCOUNT; i++) Insert(A[i], &PQ);
    for (i = MAXCOUNT - 1; i >= 0; i--) A[i] = Remove(&PQ);
}
```

## Sorting Using a Priority Queue (cont’d)

```
int SquareOf(int x) {
    return x * x;
}
int main(void) {
    int i; SortingArray A;
    for (i = 0; i < 10; i++){
        A[i] = SquareOf(3 * i - 13);
        printf("%d ",A[i]);
    }
    printf("\n");
    PriorityQueueSort(A);
    for (i = 0; i < 10; i++) {
        printf("%d ",A[i]);
    }
    printf("\n");
    return 0;
}
```

## Implementations of Priority Queues

We will present two implementations of a priority queue:

- Using sorted linked lists
- Using unsorted arrays

## The Priority Queue Data Types

- In the **sorted linked list case**, the file PQTypes.h can be defined as follows:

```
#define MAXCOUNT 10
typedef int PQItem;
typedef struct PQNodeTag {
    PQItem NodeItem;
    struct PQNodeTag * Link;
} PQListNode;

typedef struct {
    int Count;
    PQListNode * ItemList;
} PriorityQueue;
```

## Implementing Priority Queues Using Sorted Linked Lists

```
/* This is the file PQImplementation.c */
#include <stdio.h>
#include <stdlib.h>
#include "PQInterface.h"
/* Now we give all the details of the functions */
/* declared in the interface file together with */
/* local private functions. */

void Initialize(PriorityQueue *PQ) {
    PQ->Count = 0;
    PQ->ItemList = NULL;
}
```

## Implementing Priority Queues Using Sorted Linked Lists (cont'd)

```c
int Empty(PriorityQueue *PQ) {
    return(PQ->Count == 0);
}

int Full(PriorityQueue *PQ) {
    return(PQ->Count == MAXCOUNT);
}
```

## Implementing Priority Queues Using Sorted Linked Lists (cont'd)

```
PQListNode *SortedInsert(PQItem Item, PQListNode *P) {
    PQListNode * N;
    if ((P == NULL) || (Item >= P->NodeItem)){
        N = malloc(sizeof(PQListNode));
        N->NodeItem = Item;
        N->Link = P;
        return(N);
    }
    else {
        P->Link = SortedInsert(Item, P->Link);
        return(P);
    }
}
```

## Implementing Priority Queues Using Sorted Linked Lists (cont’d)

```c
void Insert(PQItem Item, PriorityQueue *PQ) {
    if (! Full(PQ)) {
        PQ->Count++;
        PQ->ItemList = SortedInsert(Item, PQ->ItemList);
    }
}
```

## Functions Insert and SortedInsert

The function Insert keeps the elements of the list in **decreasing order** (the first item has the highest priority).

The function Insert calls SortedInsert for doing the actual insertion.

SortedInsert has three cases to consider:

- If the ItemList of PQ is empty.
- If the new item has priority greater than or equal the priority of the first item on ItemList.
- If the new item has priority less than that of the first item on ItemList. In this case the function is called recursively on the tail of the list.

## Implementing Priority Queues Using Sorted Linked Lists (cont'd)

```
PQItem Remove(PriorityQueue *PQ) {
    PQItem temp;
    if (!Empty(PQ)) {
        temp = PQ->ItemList->NodeItem;
        PQ->ItemList = PQ->ItemList->Link;
        PQ->Count--;
        return(temp);
    }
}
```

## Function Remove

The function Remove simply deletes the item in the first node of the linked list representing PQ (this is the item with highest priority) and returns the value of its field NodeItem.

## The Priority Queue Data Types

- In the **unsorted array case**, the file PQTypes.h can be defined as follows:

```c
#define MAXCOUNT 10
typedef int PQItem;
typedef PQItem PQArray[MAXCOUNT];
typedef struct {
    int Count;
    PQArray ItemArray;
} PriorityQueue;
```

## Implementing Priority Queues Using Unsorted Arrays

```
/* This is the file PQImplementation.c */
#include <stdio.h>
#include "PQInterface.h"
/* Now we give all the details of the functions */
/* declared in the interface file together with */
/* local private functions. */
void Initialize(PriorityQueue *PQ) {
    PQ->Count = 0;
}
```

## Implementing Priority Queues Using Unsorted Arrays (cont’d)

```
int Empty(PriorityQueue *PQ) {
    return(PQ->Count == 0);
}

int Full(PriorityQueue *PQ) {
    return(PQ->Count == MAXCOUNT);
}
```

## Implementing Priority Queues Using Unsorted Arrays (cont’d)

```
void Insert(PQItem Item, PriorityQueue * PQ) {
    if (!Full(PQ)) {
        PQ->ItemArray[PQ->Count] = Item;
        PQ->Count++;
    }
}
```

## Function Insert

The function Insert simply appends the new item to the end of array ItemArray of PQ.

## Implementing Priority Queues Using Unsorted Arrays (cont’d)

```
PQItem Remove(PriorityQueue *PQ) {
    int i;
    int MaxIndex;
    PQItem MaxItem;
    if (!Empty(PQ)) {
        MaxItem = PQ->ItemArray[0];
        MaxIndex = 0;
        for (i = 1; i < PQ->Count; i++) {
            if (PQ->ItemArray[i] > MaxItem) {
              MaxItem = PQ->ItemArray[i];
              MaxIndex = i;
            }
        }
        PQ->Count--;
        PQ->ItemArray[MaxIndex] = PQ->ItemArray[PQ->Count];
        return(MaxItem);
    }
}
```

## Function Remove

In the function Remove, we first find the item with highest priority. Then, we save it in a temporary variable (MaxItem), we delete it from the array ItemArray and move the last item of the array to its position. Then, we return the item of the highest priority.

## Interface Header Files

Note that the module interface header file PQInterface.h is included in two important but distinct places:

- At the beginning of the **implementation files** that define the hidden representation of the externally accessed module services.

- At the beginning of **programs** that need to gain access to the external module services defined in the interface file.

## Separate Compilation

We can compile the module and the client program **separately:**

```
gcc -c PQImplementation.c -o PQ.o
gcc -c sorting.c -o sorting.o
gcc PQ.o sorting.o --o program.exe
```

- With the first two commands, we compile the C files to produce **object files**. Then, the object files are **linked** to produce the final executable.

## Information Hiding Revisited

- Let us revisit the sorting program we wrote earlier and consider the new printf statement.

```c
#include <stdio.h>
#include "PQInterface.h"
typedef PQItem SortingArray[MAXCOUNT];
/* Note: MAXCOUNT is 10 */
void PriorityQueueSort(SortingArray A) {
    int i;
    PriorityQueue PQ;
    Initialize(&PQ);
    for (i = 0; i < MAXCOUNT; i++)
        Insert(A[i], &PQ);
    printf("The queue contains %d elements\n",PQ.Count);
    for (i = MAXCOUNT - 1 ; i >= 0; i--)
        A[i] = Remove(&PQ);
}
```

## Information Hiding Revisited (cont'd)

- This printf statement accesses the Count field of the priority queue PQ. Therefore, the previous module organization **has not achieved information hiding** as nicely as we would want it.
- We can live with that deficiency or try to address it. How?

## Another Example: Complex Number Arithmetic

- A **complex number** is an expression $a + bi$ where $a$ and $b$ are reals.
- $a$ is called the **real part** and $b$ the **imaginary part.**
- $i = \sqrt{-1}$ is the **imaginary unit**. It follows that $i^2 = -1$.
- To multiply complex numbers, we follow the usual algebraic rules.

## Examples

- $(a+bi)(c+di) = ac + bci + adi + bdi^2 = (ac - bd) + (ad + bc)i$
- $(1-i)(1-i) = 1 - i - i + 1^2 = -2i$
- $(1+i)^4 = 4i^4 = -4$
- $(1+i)^8 = 16$
- Dividing the two parts of the above equation by $16 = \sqrt{-1}^8$, we find that $(1/\sqrt{2} + i/\sqrt{2})^8 = 1$

## Complex Roots of Unity

- In general, there are many complex numbers that evaluate to 1 when raised to a power. These are the **complex roots of unity**.
- For each N, there are exactly N complex numbers z such that $z^n = 1$
- The numbers $\cos(2pk/N)$ for $k = 0, 1, \ldots, N-1$ can be easily shown to have this property.
- Let us now write a program that computes and outputs these numbers for a given $N$

## An ADT for Complex Numbers: the Interface

```
/* This is the file COMPLEX.h */
typedef struct complex * Complex;
Complex COMPLEXinit(float, float);
float Re(Complex);
float Im(Complex);
Complex COMPLEXmult(Complex, Complex);
```

## Notes

- The interface on the previous slide provides clients with **handles** to complex number objects but does not give any information about the representation.
- The representation is a struct that is not specified except for its tag name.

## Handles

- We use the term **handle** to describe a reference to an abstract object.
- Our goal is to give client programs handles to abstract objects that can be used in assignment statements and as arguments and return values of functions in the same way as built-in data types, while hiding the representation of objects from the client program.

# Complex Numbers ADT Implementation

```
/* This is the file CImplementation.c */
#include <stdlib.h>
#include "COMPLEX.h"
struct complex {
    float Re;
    float Im;
};
Complex COMPLEXinit(float Re, float Im) {
    Complex t = malloc(sizeof * t);
    t->Re = Re; t->Im = Im;
    return t;
}
float Re(Complex z) {
    return z->Re;
}
float Im(Complex z) {
    return z->Im;
}
Complex COMPLEXmult(Complex a, Complex b) {
    return COMPLEXinit(Re(a) * Re(b) - Im(a) * Im(b), Re(a) * Im(b) +
}
```

## Notes

- The implementation of the interface in the previous program includes **the definition of structure** complex (which is hidden from the clients) as well as **the implementation of the functions** provided by the interface.
- Objects are pointers to structures, so we dereference the pointer to refer to the fields.

## Client Program

```c
/* Computes the N complex roots of unity for given N */
/* This is file roots-of-unity.c */
#include <stdio.h>
#include <math.h>
#include "COMPLEX.h"
#define PI 3.141592625
int main(int argc, char *argv[]) {
    int i, j, N = atoi(argv[1]);
    Complex t, x;
    printf("%dth complex roots of unity\n", N);
    for (i = 0; i < N; i++) {
        float r = 2.0 * PI * i/N;
        t = COMPLEXinit(cos(r), sin(r));
        printf("%2d %6.3f %6.3f ", i, Re(t), Im(t));
        for (x = t, j = 0; j < N-1; j++)
            x = COMPLEXmult(t, x);
        printf("%6.3f %6.3f\n", Re(x), Im(x));
    }
}
```

## Notes

- The client program outputs the powers of unity one by one, together with a verification that they are indeed such powers. To verify this, raising to a power is implemented by multiplication.

## Notes

- In this case, we can see that the exact representation of a complex number is hidden from the client program.
- The client program can refer to the real and the imaginary part of a number **only by using the functions** Re and Im provided by the interface.

## Command Line Arguments

```
argc (argument count) // is the number of command line arguments.
argv (argument vector) // is pointer to an array of character strings
```

- By convention, argv[0] is the name by which the program was invoked so argc is at least 1.
- In the previous program argv[1] contains the value of N.

## Separate Compilation

We compile the module and the client program separately__:__

```
gcc -c CImplementation.c -o CI.o
gcc -c roots-of-unity.c -o roots-of-unity.o
gcc CI.o roots-of-unity.o --o program.exe -lm
```

- With the first two commands we compile the C files to produce object files. Then the object files are linked to produce the final executable. Notice that we have to use the option –lm to link the math library.

## Exercise

- Revisit the ADT priority queue and define a better interface and its implementation so that we have information hiding.

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C.*
- Chapter 4.
- Robert Sedgewick. Αλγόριθμοι σε C.
- Κεφ. 4

# Stacks (Στοίβες)

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Under construction

## Stacks and Queues

- **Linear data structures** are collections of components arranged in a straight line.
- If we restrict the growth of a linear data structure so that new components can be added and removed only at one end, we have a **stack.**
- If new components can be added at one end but removal of components must take place at the opposite end, we have a **queue (ουρά αναμονής).**

## Examples of Stacks in Real Life

## Stacks in Computer Science

Stacks are used in many areas of Computer Science:

- Parsing algorithms
- Pushdown automata
- Expression evaluation algorithms
- Backtracking algorithms
- Activation records in run-time stack.

## Stacks

- Stacks are sometimes called **LIFO lists** where LIFO stands for “last-in, first-out”.
- When we add a new object to the top of a stack, this is called “**pushing**”.
- When we remove an object from the top of a stack, this is called “**popping**”.
- Pushing and popping are inverse operations.

## Sequences

- A finite-length sequence $S = (s_1, s_2, \ldots, sn)$ is just an ordered arrangement of finitely many components $s_1, s_2, \ldots, s_n$.
- The **length** of a sequence is the number of its components.
- There is a special sequence with length *0* called the **empty sequence**.

## An Abstract Data Type for Stacks

A **stack** $S$ of items of type $T$ is a sequence of items of type $T$ on which the following operations can be defined:

- Initialize the stack $S$ to be the **empty stack**.
- Determine whether or not the stack $S$ is **empty**.
- Determine whether or not the stack $S$ is **full**.
- **Push** a new item onto the top of stack $S$.
- If $S$ is nonempty, **pop** an item from the top of stack $S$.

## An Interface for Stacks

- Using **separately compiled C files**, we can define **C modules** that specify the underlying representation for stacks and implement the abstract stack operations.

## The Stack ADT Interface

```c
/* This is the file StackInterface.h */
#include "StackTypes.h"
void InitializeStack(Stack *S);
int Empty(Stack *S);
int Full(Stack *S);
void Push(ItemType X, Stack *S);
void Pop(Stack *S, ItemType *X);
```

## Using the Stack ADT to Check for Balanced Parentheses

- The first application of the Stack ADT that we will study involves determining whether parentheses and brackets balance properly in algebraic expressions.
- Example: $\{a^2 - [(b+c)^2 - (d+e)^2] * [\sin(x-y)]\} - \cos(x+y)$
- This expression contains parentheses, square brackets, and braces in balanced pairs according to the pattern { [ ( ) ( ) ] [ ( ) ] } ( )

## The Algorithm

- We can start with an **empty stack** and scan a string representing the algebraic expression from left to right.
- Whenever we encounter a left parenthesis (, a left bracket [ or a left brace {, we **push** it onto the stack.
- Whenever we encounter a right parenthesis ), a right bracket ] or a right brace }, we **pop** the top item off the stack and check to see that its type matches the type of right parenthesis, bracket or brace encountered.
- If the stack is **empty** by the time we get to the end of the expression string and if all pairs of matched parentheses were of the same type, the expression has properly balanced parentheses. Otherwise, the parentheses are not balanced properly.

## The Program

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char *InputExpression;

int Match(char c, char d) {
    switch (c) {
        case '(' : return d==')';
            break;
        case '[' : return d==']';
            break;
        case '{' : return d=='}';
            break;
        default : return(0);
            break;
    }
}
```

## The Program (cont'd)

```
void ParenMatch(void) {
    int n, i=0;  char c, d;
    Stack ParenStack;
    InitializeStack(&ParenStack);
    n = strlen(InputExpression);

    while (i < n) {
        d = InputExpression[i];
        if ( d=='(' || d=='[' || d=='{' ) {
            Push(d, &ParenStack);
        } else if ( d==')' || d==']' || d=='}' ) {
            if (Empty(&ParenStack)) {
                printf("More right parentheses than left parentheses\
                return;
```

## The Program (cont’d)

```
            } else {
                Pop(&ParenStack,&c);
                if( ! Match(c,d)) {
                    printf("Mismatched Parentheses: %c and %c\n",c,d
                    return;
                }
            }
        }
        i++;
    }
    if (Empty(&ParenStack)) {
        printf("Parentheses are balanced properly\n");
    } else {
        printf("More left parentheses than right parentheses\n");
    }
}
```

## The Program (cont’d)

```
int main(void) {
    InputExpression = malloc(100 * sizeof(char));
    printf("Give Input Expression without blanks:");
    scanf("%s", InputExpression);
    ParenMatch();

    return 0;
}
```

## Using the Stack ADT to Evaluate Postfix Expressions

- Expressions are usually written in **infix** notation **(ενθεματικό συμβολισμό)** e.g., `(a+b)*2-c`. Parentheses are used to denote the order of operation.

- **Postfix (μεταθεματικές)** expressions are used to specify algebraic operations using a parentheses free notation. For example, `ab+2*c-`.

- The postfix notation `L R op` corresponds to the infix notation `L op R`.

## Examples

| Infix | Postfix |
|---|---|
| (a + b) | a b + |
| (x - y - z) | x y - z - |
| (x - y - z) / (u + v) | x y - z - u v + / |
| (a^2 + b^2) * (m - n) | a 2 ^ b 2 ^ + m n - * |

## Prefix Notation

- There is also **prefix (or Polish) notation (ενθεματικός συμβολισμός)** in which the operator precedes the operands.
- **Example**: + 3 * 2 5 is the prefix form of (2 * 5) + 3
- Prefix and postfix notations **do not need parentheses** for denoting the order of operations.

## The Algorithm

- To evaluate a postfix expression *P*, you **scan from left to right**.
- When you encounter an operand *X*, you **push** it onto an evaluation stack *S*.
- When you encounter an operator *op*, you **pop** the topmost operand stacked on *S* into a variable *R* (which denotes the right operand), then you **pop** another topmost operand stacked on *S* onto a variable *L* (which denotes the left operand).
- Finally, you perform the operation *op* on *L* and *R*, getting the value of the expression *L op R*, and you **push** the value back onto the stack *S*.
- When you finish scanning *P*, the value of *P* is the only item remaining on the stack *S*.

## The Program

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <ctype.h>
#include <string.h>
#include "StackInterface.h"
Stack EvalStack;
char PostfixString[20];
void InterpretPostfix(void) {
    float LeftOperand, RightOperand, Result;
    int i;
    char c;
    char s[] = 'x';
    InitializeStack(&EvalStack);
```

## The Program (cont'd)

```
    for (i = 0; i < strlen(PostfixString); i++) {

        s[0] = c = PostfixString[i];

        if (isdigit(c)) {
            Push( (float) (atof(s)), &EvalStack);
        } else if (c=='+' || c=='-' || c==' * ' || c=='/' || c=='^')
            Pop(&EvalStack, &RightOperand);
            Pop(&EvalStack, &LeftOperand);
```

## The Program (cont'd)

```
        switch (c) {
            case '+': Push(LeftOperand+RightOperand, &EvalStack);
                      break;
            case '-': Push(LeftOperand-RightOperand, &EvalStack);
                      break;
            case '* ': Push(LeftOperand*RightOperand, &EvalStack);
                      break;
            case '/': Push(LeftOperand/RightOperand, &EvalStack);
                      break;
            case '^': Push(pow(LeftOperand, RightOperand), &EvalStack
                      break;
            default: break;
            }
        }
    }
    Pop(&EvalStack, &Result);
    printf("Value of postfix expression = %f\n", Result);
}
```

## Notes

- We need the string $s[]$ (which is terminated by the character '\ 0') so that we can apply the function atof(char $*$ ) to it and get back a real number. We **cannot** apply this function to the character variable c.

## The Program (cont’d)

```c
int main(void) {
    printf("Give input postfix string without blanks:");
    scanf("%s", PostfixString);
    InterpretPostfix();
    return 0;
}
```

## Implementing the Stack ADT

We will present two implementations of the stack ADT based on:

- arrays (sequential representation)
- linked lists (linked representation)

Both implementations can be used to realize the two applications we presented earlier.

## The Implementation Based on Arrays

```
/* This is the file StackTypes.h */
#define MAXSTACKSIZE 100
typedef char ItemType;
/* char is the type for our first application */
/* float is the type for our second application */
typedef struct{
    int Count;
    ItemType Items[MAXSTACKSIZE];
} Stack;
```

## The Implementation Based on Arrays (cont’d)

```
/* This is the file StackImplementation.c */
#include <stdio.h>
#include <stdlib.h>
#include "StackInterface.h"
void InitializeStack(Stack *S) {
    S->Count = 0;
}
int Empty(Stack *S) {
    return (S->Count == 0);
}
```

## The Implementation Based on Arrays (cont’d)

```
int Full(Stack * S) {
    return(S->Count == MAXSTACKSIZE);
}

void Pop(Stack * S, ItemType * X) {
    if (S->Count ==0) {
        printf("attempt to pop the empty stack");
    } else {
        --(S->Count);
        * X = S->Items[S->Count];
    }
}
```

## The Implementation Based on Arrays (cont’d)

```c
void Push(ItemType X, Stack *S) {
    if (S->Count == MAXSTACKSIZE){
        printf("attempt to push new item on a full stack");
    } else {
        S->Items[S->Count] = X;
        ++(S->Count);
    }
}
```

## The Implementation Based on Linked Lists

```
/* This is the file StackTypes.h */
typedef char ItemType;
/* char is the type for our first application */
/* float is the type for our second application */
typedef struct StackNodeTag {
    ItemType Item;
    struct StackNodeTag * Link;
} StackNode;

typedef struct {
    StackNode * ItemList;
} Stack;
```

## The Implementation Based on Linked Lists (cont’d)

```c
/* This is the file StackImplementation.c */
#include <stdio.h>
#include <stdlib.h>
#include "StackInterface.h"
void InitializeStack(Stack *S) {
    S->ItemList = NULL;
}

int Empty(Stack *S) {
    return (S->ItemList==NULL);
}

int Full(Stack *S) {
    return 0;
}
/* We assume an already constructed stack is not full since it can
otentially */
/* grow as a linked structure */
```

## The Implementation Based on Linked Lists (cont’d)

```
void Push(ItemType X, Stack *S) {
    StackNode * Temp;
    Temp = malloc(sizeof(StackNode));
    if (Temp == NULL) {
        printf("system storage is exhausted");
    } else {
        Temp->Link = S->ItemList;
        Temp->Item = X;
        S->ItemList = Temp;
    }
}
```

## The Implementation Based on Linked Lists (cont’d)

```
void Pop(Stack *S, ItemType *X) {
    StackNode * Temp;
    if (S->ItemList == NULL){
        printf("attempt to pop the empty stack");
    } else {
        Temp = S->ItemList;
        * X = Temp->Item;
        S->ItemList = Temp->Link;
        free(Temp);
    }
}
```

## Information Hiding Revisited

- The two previous specifications of the ADT stack **do not hide the details of the representation** of the stack since a client program can access the array or the list data structure because it includes StackInterface.h and therefore StackTypes.h.
- We will now present another specification which does a better job in hiding the representation of the stack.

## The Interface File STACK.h

```
void STACKinit(int);
int STACKempty();
void STACKpush(Item);
Item STACKpop();
```

The type Item will be defined in a header file Item.h which will be included in the implementation of the interface and the client programs.

## The Implementation of the Interface

As previously, we will consider an array implementation and a linked list implementation of the ADT stack.

## The Array Implementation

```c
#include <stdlib.h>
#include "Item.h"
#include "STACK.h"
static Item *s;
static int N;
void STACKinit(int maxN) {
    s = malloc(maxN * sizeof(Item));
    N = 0;
}
int STACKempty() {
    return N == 0;
}
void STACKpush(Item item) {
    s[N++] = item;
}
Item STACKpop() {
    return s[--N];
}
```

## Notes

- The variable s is a pointer to an item (equivalently, the name of an array of items defined by Item s[]).
- When there are N items in the stack, the implementation keeps them in array elements s[0], …, s[N-1].
- The variable N shows the top of the stack (where the next item to be pushed will go).
- N is defined as a **static** variable i.e., it **retains its value throughout calls** of the various functions that access it.
- The client program passes the maximum number of items expected on the stack as an argument to STACKinit.
- The previous code does not check for errors such as pushing onto a full stack or popping an empty one.

## The Linked List Implementation

```
#include <stdlib.h>
#include "Item.h"

typedef struct STACKnode* link;
struct STACKnode {
    Item item;
    link next;
};

static link head;

link NEW(Item item, link next) {
    link x = malloc(sizeof (* x));
    x->item = item;
    x->next = next;
    return x;
}
```

## The Linked List Implementation(cont’d)

```
void STACKinit(int maxN) {
    head = NULL;
}

int STACKempty() {
    return head == NULL;
}

STACKpush(Item item) {
  head = NEW(item, head);
}

Item STACKpop() {
    Item item = head->item;
    link t = head->next;
    free(head);
    head = t;
    return item;
}
```

## Notes

- This implementation uses an auxiliary function NEW to allocate memory for a node, set its fields from the function arguments, and return a link to the node.
- In this implementation, we keep the stack in the reverse order of the array implementation; from most recently inserted elements to least recently inserting elements.
- **Information hiding:** For both implementations (with arrays or linked lists), the data structure for the representation of the stack (array or linked list) is defined **only** in the implementation file thus it is not accessible to client programs.

## Translating Infix Expressions to Postfix

- Let us now use the latest implementation of the stack ADT to implement a translator of **fully parenthesized** infix arithmetic expressions to postfix.
- The **algorithm** for doing this is as follows. To convert (A+B) to the postfix form AB+, we ignore the left parenthesis, convert A to postfix, save the + on the stack, convert B to postfix, then, on encountering the right parenthesis, pop the stack and output the +.

## Example

- We want to translate the infix expression ((5*(9+8))+7) into postfix.
- The result will be 5 9 8 + * 7 +.

| Input | Output | Stack |
|---|---|---|
| ( | | |
| ( | | |
| 5 | 5 | |
| * | | * |
| ( | | * |
| 9 | 9 | * |
| + | | * + |
| 8 | 8 | * + |
| ) | + | * |
| ) | * | |
| + | | + |
| 7 | 7 | + |
| ) | + | |

## The Client Program

```
#include <stdio.h>
#include <string.h>
#include "Item.h"
#include "STACK.h"
int main(int argc, char *argv[]) {
    char * a = argv[1];
    int i, N = strlen(a);
    STACKinit(N);
    for (i = 0; i < N; i++) {
    if (a[i] == ')')
        printf("%c ", STACKpop());
    if ((a[i] == '+') || (a[i] == '*'))
        STACKpush(a[i]);
    if ((a[i] >= '0') && (a[i] <= '9'))
        printf("%c ", a[i]);
    }
    printf("\n");
    return 0;
}
```

## The File Item.h

- The file Item.h can only contain a typedef which defines the type of items in the stack.
- For the previous program, this can be:

```
typedef char Item;
```

## A Weakness of the 2nd Solution

- The 2nd solution for defining and implementing a stack ADT is weaker than the 1st one since it allows the construction and operation of a **single stack** by a client program.
- Conversely, the 1st solution allows us to define **many stacks** in the client program.

## Exercise

- Modify the 1st solution so that it does better information hiding without losing the capability to be able to define many stacks in the client program.

## Question

- Which implementation of a stack ADT should we prefer?

## Answer

- It depends on the application.
- In the linked list implementation, push and pop take more time to allocate and de-allocate memory.
- If we need to do these operations a huge number of times then we might prefer the array implementation.
- On the other hand, the array implementation uses the amount of space necessary to hold the maximum number of items expected. This can be wasteful if the stack is not kept close to full.
- The list implementation uses space proportional to the number of items but always uses extra space for a link per item.
- Note also that the **running time** of push and pop in each implementation is **constant**.

## How C Implements Recursive Function Calls Using Stacks

- When calling an instance of a function *F(a1,a2,…,an)* with actual parameters *a1*, *a2,…,an*, C uses a **run-time stack**.
- A collection of information called a **stack frame** or **call frame** or **activation record** is prepared to correspond to the call and it is placed on top of other previously generated stack frames on the run-time stack.

## Stack Frames

The information in a stack frame consists of:

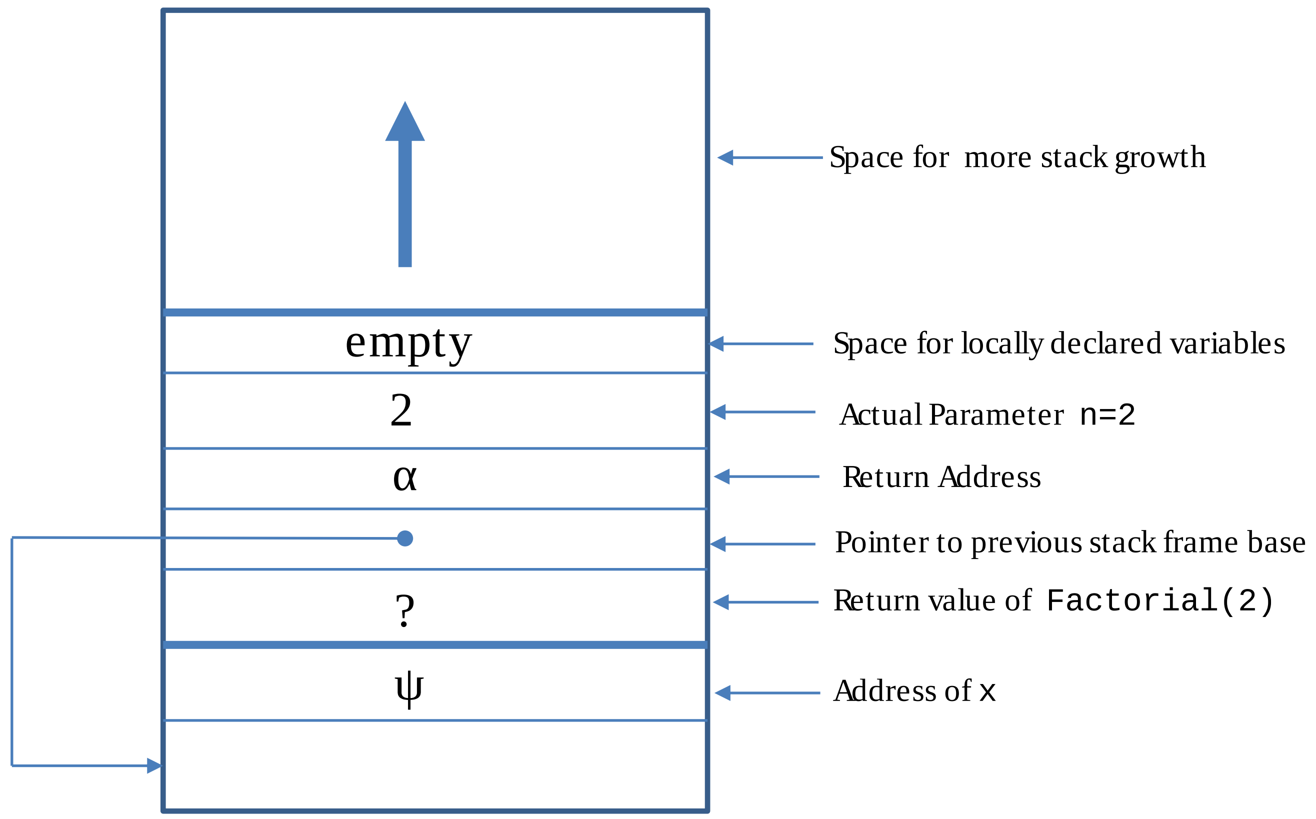
- Space to hold the **value returned by the function**.
- A **pointer to the base of the previous stack frame** in the stack.
- A **return address**, which is the address of an instruction to execute in order to resume the execution of the caller of the function when the call has terminated.
- Parameter storage sufficient to hold **the actual parameter values** used in the call.
- A set of storage locations sufficient to hold the values of the **variables declared locally** in the function.

## Example - Factorial

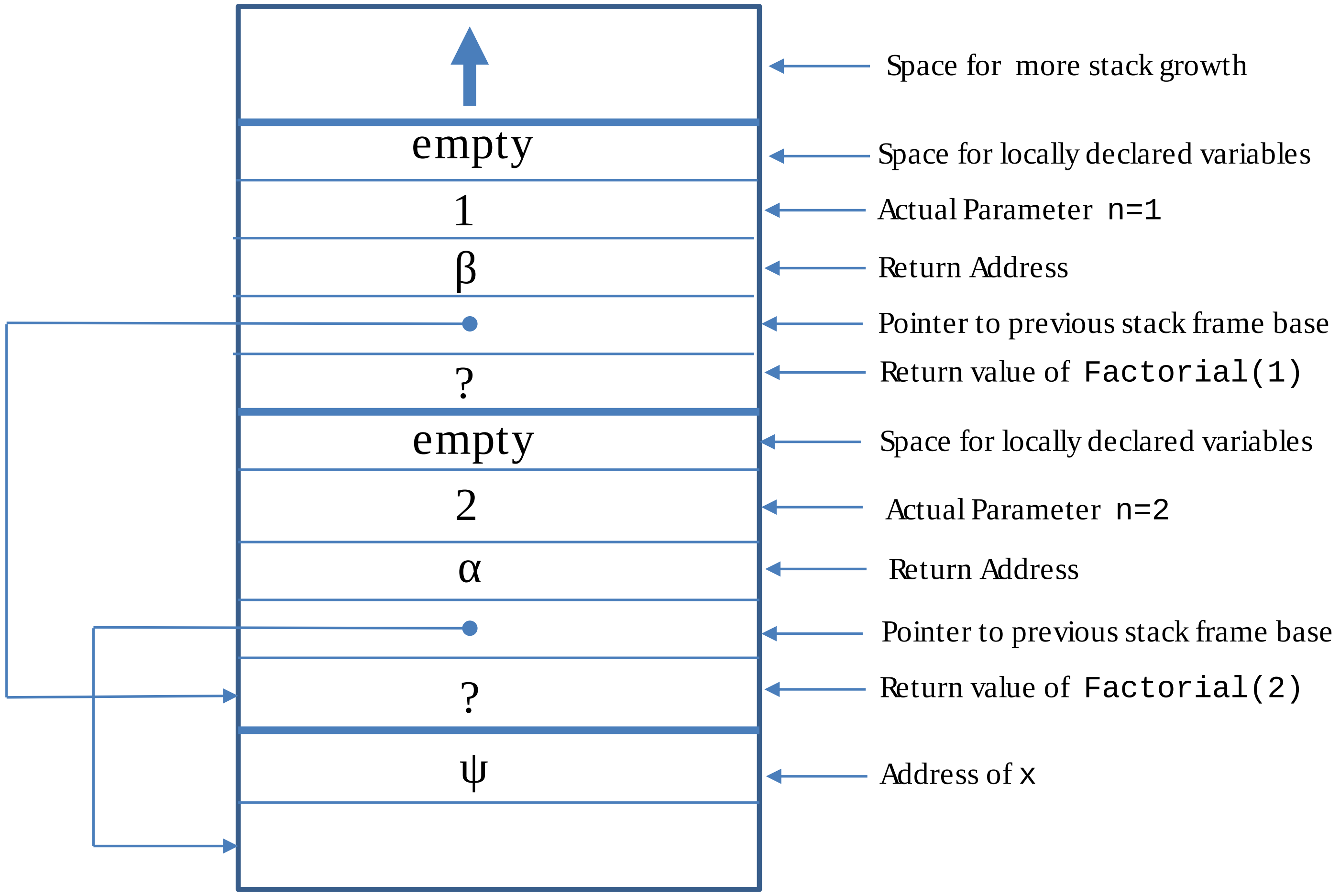
```
int Factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * Factorial(n-1);
    }
}
```

- Let us consider the call x=Factorial(2).

# Stack Frame for Factorial(2)

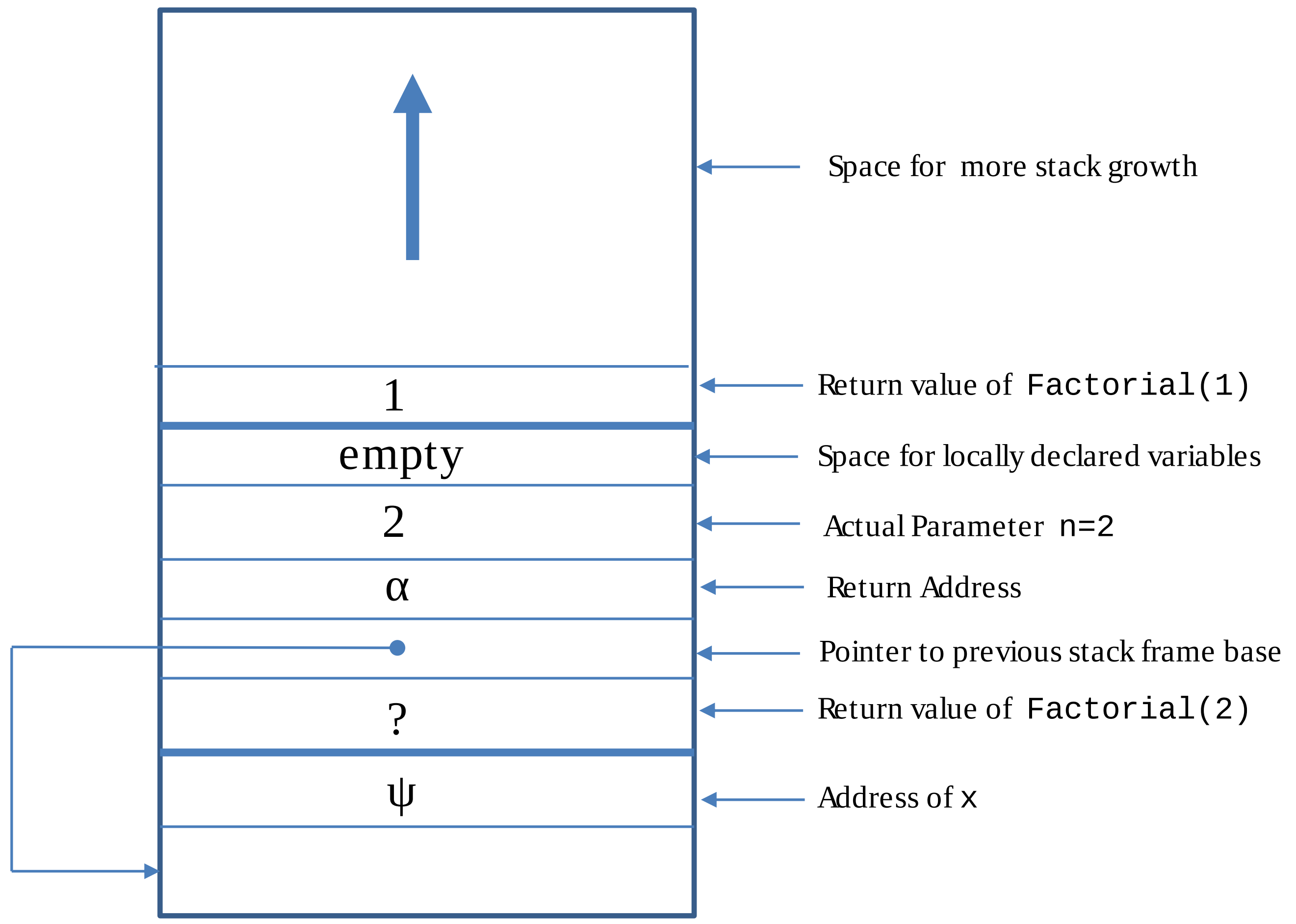

empty
2
α
?
ψ
Space for more stack growth
Space for locally declared variables
Actual Parameter n=2
Return Address
Pointer to previous stack frame base
Return value of Factorial(2)
Address of x

# Stack Frame for Factorial(2)and Factorial(1)



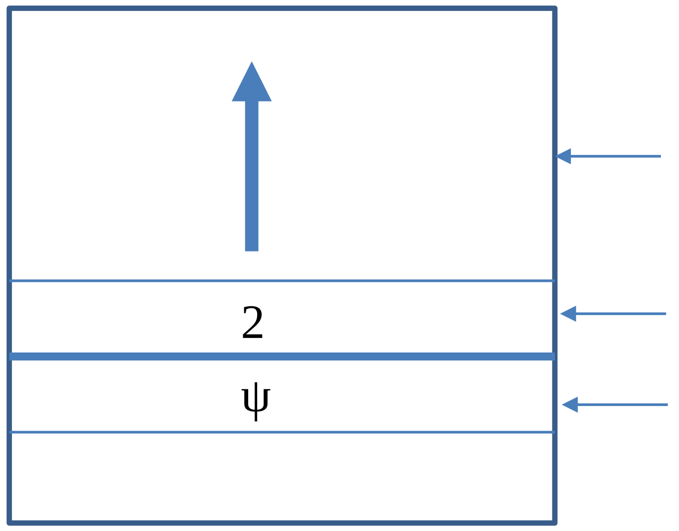
empty
1
β
?
empty
2
α
?
ψ
Space for more stack growth
Space for locally declared variables
Actual Parameter n=1
Return Address
Pointer to previous stack frame base
Return value of Factorial(1)
Space for locally declared variables
Actual Parameter n=2
Return Address
Pointer to previous stack frame base
Return value of Factorial(2)
Address of x

## Stack After Return from Factorial(1)



1
empty
2
α
?
ψ
Space for  more stack growth
Return value of `Factorial(1)`
Space for locally declared variables
Actual Parameter `n=2`
Return Address
Pointer to previous stack frame base
Return value of `Factorial(2)`
Address of `x`

## Stack After Return from Factorial(2)



## More Details

- **Stack + Iteration** can implement **Recursion**.
- Run-time stacks are discussed in more details in a Compilers course.

## Using Stacks

- Generally speaking, stacks can be used to implement any kind of **nested structure**.
- When processing nested structures, we can start processing the outermost level of the structure, and if we encounter a nested substructure, we can interrupt the processing of the outer layer to begin processing an inner layer by putting a record of the interrupted status of the outer layer's processing on top of a stack.
- In this way the stack contains **postponed obligations** that we should resume and complete.

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C.*
- Chapter 7.
- R. Sedgewick. Αλγόριθμοι σε C.
- Κεφ. 4

# Queues

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Under construction

## The ADT Queue

A **queue** $Q$ of items of type $T$ is a sequence of items of type $T$ on which the following operations are defined:

- Initialize the queue to the **empty queue**.
- Determine whether or not the queue is **empty**.
- Determine whether or not the queue is **full**.
- Provided $Q$ is not full, **insert** a new item onto the rear of the queue.
- Provided $Q$ is nonempty, **remove** an item from the front of $Q$.

## The ADT Queue (cont'd)

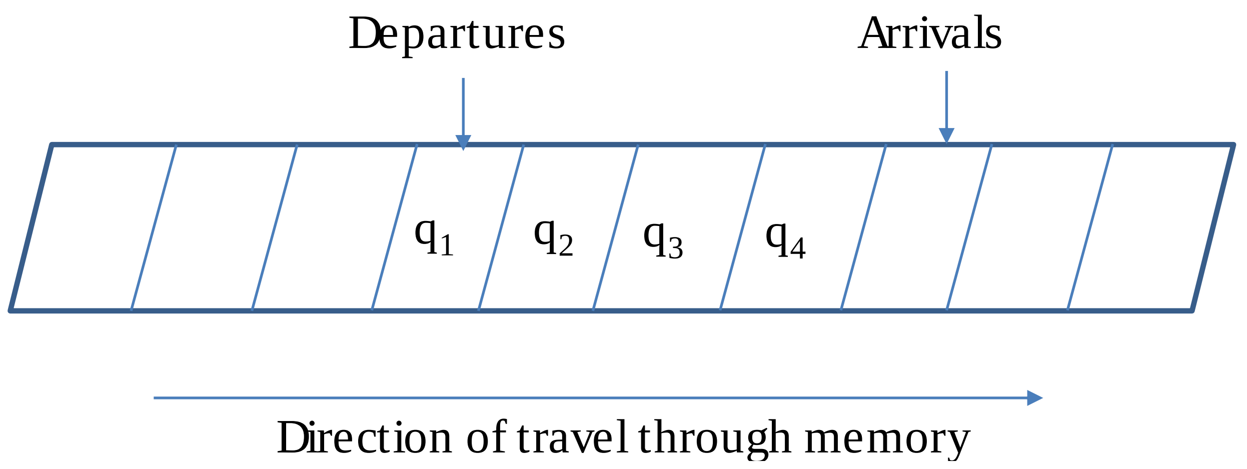
Queues are also known as **FIFO lists** (first-in first-out).

## Queue Representations

The ADT queue can be implemented using either **sequential** or **linked** representations.

## Sequential Queue Representations

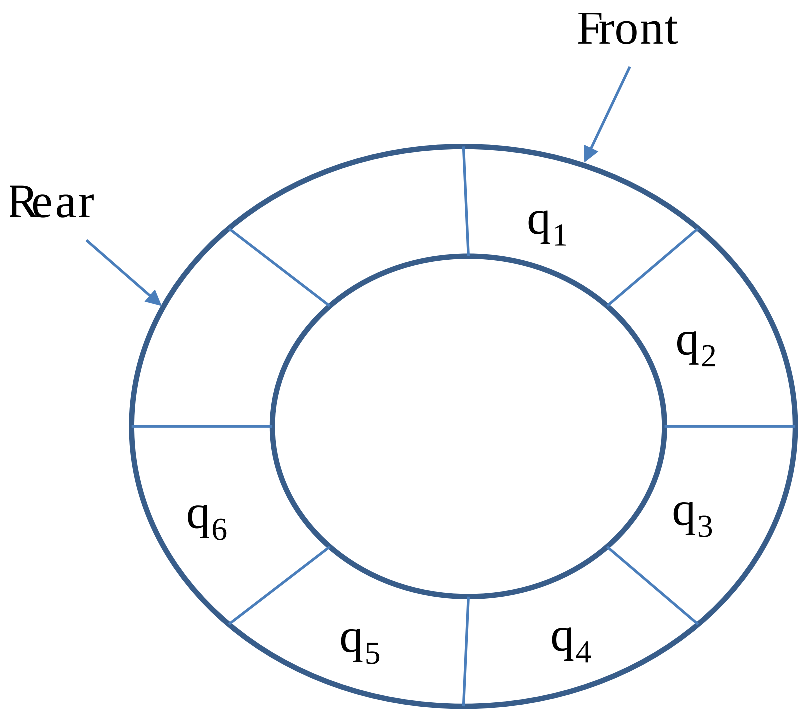
We can use an **array** as follows:



## Sequential Queue Representations (cont’d)

- This representation is **not very handy**.
- The positions of the array to the right will be filled until there is space to do so, while the positions to the left of the array will be freed but we will not be able to use that free space.
- The bounded space representation proposed next is a better one.

# Circular Queue Representation



## Circular Queue Representation (cont’d)

- If we have an array Items[0:N-1] and two pointers Front and Rear as in the previous figure, then we can use the following assignment statements to increment the pointers so that they always wrap around after falling off the high end of the array.
- Front=(Front+1)%N
- Rear=(Rear+1)%N
- The operator % computes the **remainder** of the division by N so the values of Front and Rear are always in the range 0 to N-1.

## Defining the Queue Data Type

```
/* This is the file QueueTypes.h */
#define MAXQUEUESIZE 100
typedef int ItemType;
/* the item type can be arbitrary */
typedef struct {
    int Count;
    int Front;
    int Rear;
    ItemType Items[MAXQUEUESIZE];
} Queue;
```

## The Interface File

```c
/* This is the file QueueInterface.h */
#include "QueueTypes.h"
void InitializeQueue(Queue *Q);
int Empty(Queue *Q);
int Full(Queue *Q);
void Insert(ItemType R, Queue *Q);
void Remove(Queue *Q, ItemType *F);
```

## The Implementation

```
/* This is the file QueueImplementation.c */
#include <stdio.h>
#include <stdlib.h>
#include "QueueInterface.h"
void InitializeQueue(Queue *Q) {
    Q->Count = 0;
    Q->Front = 0;
    Q->Rear = 0;
}
```

## The Implementation (cont’d)

```
int Empty(Queue * Q) {
    return(Q->Count == 0);
}
int Full(Queue * Q) {
    return(Q->Count == MAXQUEUESIZE);
}
```

## The Implementation (cont’d)

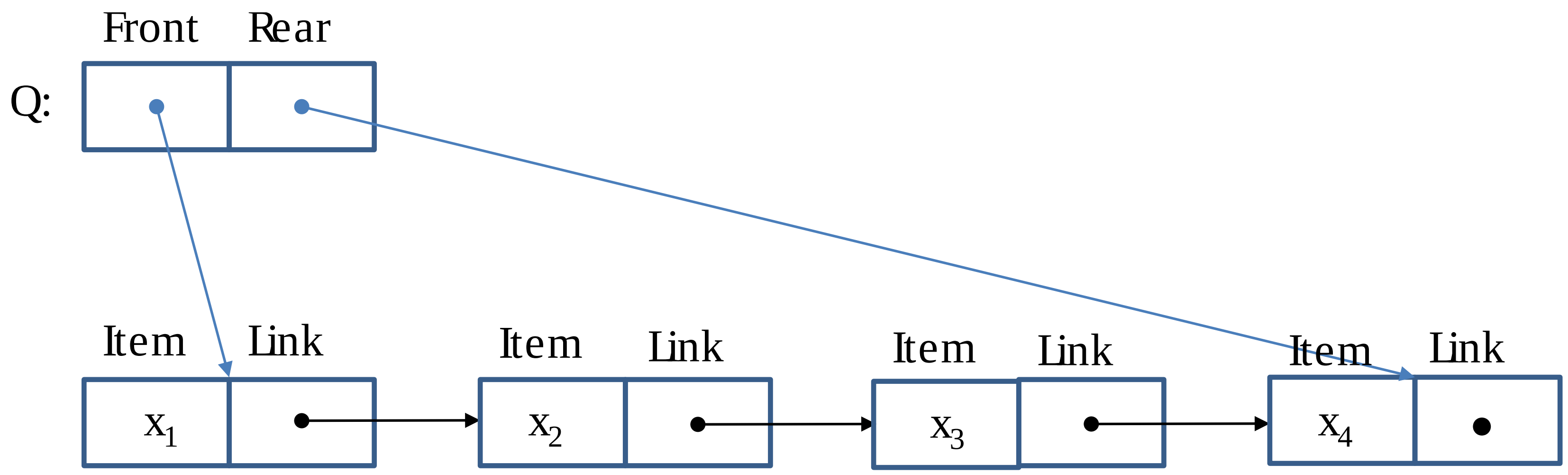
```
void Insert(ItemType R, Queue * Q) {
    if (Q->Count == MAXQUEUESIZE) {
        printf("attempt to insert item into a full queue");
    } else {
        Q->Items[Q->Rear] = R;
        Q->Rear = (Q->Rear + 1) % MAXQUEUESIZE;
        (Q->Count)++;
    }
}
```

## The Implementation (cont’d)

```
void Remove(Queue * Q, ItemType * F) {
    if (Q->Count == 0){
        printf("attempt to remove item from empty queue");
    } else {
        * F = Q->Items[Q->Front];
        Q->Front = (Q->Front + 1) % MAXQUEUESIZE;
        (Q->Count)--;
    }
}
```

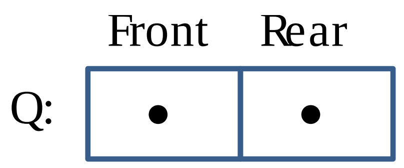
## Linked Queue Representation

In this implementation, we represent a queue by a struct containing pointers to the front and rear of a linked list of nodes.



## Linked Queue Representation (cont'd)

The **empty queue** is a special case and it is represented by a structure whose front and rear pointers are NULL.



## Defining the Queue Data Type

```c
/* This is the file QueueTypes.h */
typedef int ItemType;
/* the item type can be arbitrary */
typedef struct QueueNodeTag {
    ItemType Item;
    struct QueueNodeTag * Link;
} QueueNode;

typedef struct {
    QueueNode * Front;
    QueueNode * Rear;
} Queue;
```

## The Implementation

```
/* This is the file QueueImplementation.c */
#include <stdio.h>
#include <stdlib.h>
#include "QueueInterface.h"

void InitializeQueue(Queue *Q) {
    Q->Front = NULL;
    Q->Rear = NULL;
}
```

## The Implementation (cont’d)

```
int Empty(Queue *Q) {
    return(Q->Front == NULL);
}
int Full(Queue *Q) {
      return(0);
}
/* We assume an already constructed queue */
/* is not full since it can potentially grow */
/* as a linked structure. */
```

## The Implementation (cont'd)

```
void Insert(ItemType R, Queue *Q) {
    QueueNode * Temp;
    Temp = malloc(sizeOf(QueueNode));
    if (Temp == NULL) {
        printf("System storage is exhausted");
    } else {
        Temp->Item = R;
        Temp->Link = NULL;
        if (Q->Rear == NULL) {
            Q->Front = Temp;
            Q->Rear = Temp;
        } else {
            Q->Rear->Link = Temp;
            Q->Rear = Temp;
        }
    }
}
```

## The Implementation (cont'd)

```
void Remove(Queue *Q, ItemType *F) {
    QueueNode * Temp;
    if (Q->Front == NULL){
        printf("attempt to remove item from an empty queue");
    } else {
        * F = Q->Front->Item;
        Temp = Q->Front;
        Q->Front = Temp->Link;
        free(Temp);
        if (Q->Front==NULL)
            Q->Rear=NULL;
    }
}
```

## Example main program

```c
#include <stdio.h>
#include <stdlib.h>
#include "QueueInterface.h"
int main(void) {
    int i,j;
    Queue Q;
    InitializeQueue(&Q);
    for (i = 1; i < 10; i++) {
        Insert(i, &Q);
    }
    while ( ! Empty(&Q)) {
        Remove(&Q, &j);
        printf("Item %d has been removed.\n", j);
    }
    return 0;
}
```

## Comparing Linked and Sequential Queue Representations

- The **sequential queue representation** is appropriate when there is a bound on the number of queue elements at any time.
- The **linked representation** is appropriate when we do not know how large the queue will grow.

## Information Hiding Revisited

- The previous definitions and implementations of the ADT queue do not do good information hiding since client programs can get access to the queue representation because the file QueueTypes.h is included in the file QueueInterface.h.
- We will now give another way to define the ADT queue that does not have this weakness and also has all the nice features of the previous code such as the ability to define multiple queues in a client program.

## The Queue ADT Interface

```
typedef struct queue * QPointer;
QPointer QUEUEinit(int maxN);
int QUEUEempty(QPointer);
void QUEUEput(QPointer, Item);
Item QUEUEget(QPointer);
```

- In this interface the typedef statement defines the type QPointer which is a **handle** to a structure for which we only give the name queue. The details of this structure are given in the implementation file and, in this way, they are **hidden from client programs**.
- The functions of the interface take arguments of type QPointer.

## The Implementation of the Interface

- Let us now see how we can implement this interface using the linked list representation of a queue that we introduced earlier.
- The front and the rear of the queue are now accessed using pointer variables head and tail.

## The Implementation

```
#include <stdlib.h>
#include "Item.h"
#include "QUEUE.h"
typedef struct QUEUEnode* link;
struct QUEUEnode {
    Item item;
    link next;
};
struct queue {
    link head;
    link tail;
};
```

## The Implementation (cont'd)

```
link NEW(Item item, link next) {
    link x = malloc(sizeof (*x));
    x->item = item;
    x->next = next;
    return x;
}
QPointer QUEUEinit(int maxN) {
    QPointer q = malloc(sizeof * q);
    q->head = NULL;
    q->tail = NULL;
    return q;
}
```

## The Implementation (cont'd)

```
int QUEUEempty(QPointer q) {
    return q->head == NULL;
}

void QUEUEput(QPointer q, Item item) {
    if (q->head == NULL) {
        q->tail = NEW(item, q->head);
        q->head = q->tail;
        return;
    }
    q->tail->next = NEW(item, q->tail->next);
    q->tail = q->tail->next;
}

Item QUEUEget(QPointer q) {
    Item item = q->head->item;
    link t = q->head->next;
    free(q->head);
    q->head = t;
    return item;
}
```

## Queue Simulation

- Let us now use the previous queue interface and implementation in a client program.
- The following client program simulates an environment with M queues where clients (queue members) are assigned to one of these queues randomly.

## The Client Program

```c
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "QUEUE.h"
#define M 10
int main(int argc, char *argv[]) {
    int i, j, N = atoi(argv[1]);
    QPointer queues[M];
    for (i = 0; i < M; i++)
        queues[i] = QUEUEinit(N);
    for (i = 0; i < N; i++)
        QUEUEput(queues[rand() % M], i);
    for (i = 0; i < M; i++, printf("\n"))
        for (j = 0; !QUEUEempty(queues[i]); j++)
              printf("%3d ", QUEUEget(queues[i]));
    return 0;
}
```

## Information Hiding Revisited

- Notice that the previous client program cannot access the structure that represents the queue because this information is not revealed by the interface file QUEUE.h.
- The details are hidden in the implementation which is not accessible to the client.

## Using Queues

- **Queues of jobs** are used a lot in operating systems and networks (e.g., a printer queue).
- Queues are also used in **simulation**.
- **Queuing theory** is a branch of mathematics that studies the behaviour of systems with queues.

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C.*
- Chapter 7.
- R. Sedgewick. Αλγόριθμοι σε C.
- Κεφ. 4.

# Introduction to the Analysis of Algorithms

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Under construction

## Outline

- How can we measure and compare algorithms meaningfully?
- $O$ notation.
- Analysis of a few interesting algorithms.

## Introduction

- **How do we measure and compare algorithms meaningfully** given that the same algorithm will run at different speeds and will require different amounts of space when run on different computers or when implemented in different programming languages?
- **Example**: let us consider a sorting algorithm for sorting an array `A[0:n-1]`.

## Selection Sorting Algorithm

```
void SelectionSort(InputArray A) {
    int MinPosition, temp, i, j;
    for (i = n - 1; i > 0; --i) {
        MinPosition = i;
        for (j = 0; j < i; ++i){
            if (A[j] < A[MinPosition]) {
                MinPosition = j;
            }
        }
        temp = A[i];
        A[i] = A[MinPosition];
        A[MinPosition] = temp;
    }
}
```

## Running Times in Seconds to Sort an Array of 2000 Integers

- Computers A, B, etc. up to E are progressively faster.
- The algorithm runs faster on faster computers.

| Computer | Time |
|---|---:|
| Computer A | 51.915 |
| Computer B | 11.508 |
| Computer C | 2.382 |
| Computer D | 0.431 |
| Computer E | 0.087 |

## More Measurements

- In addition to trying different computers, we should try **different programming languages** and **different compilers**.
- Shall we take all these measurements to decide whether an algorithm is better than another one?

## A More Meaningful Criterion

- We can observe that algorithms usually **consume resources (e.g., time and space)** in some fashion that depends on the **size of the problem** solved.
- Usually, the bigger the size of a problem, the more resources an algorithm consumes.
- We usually use to denote the size of the problem.
- **Examples of sizes**: the length of a list that is searched, the number of items in an array that is sorted etc.

## SelectionSort Running Times in Milliseconds on Two Types of Computers

| Array Size | Home Computer | Desktop Computer |
|---|---|---|
| 125 | 12.5 | 2.8 |
| 250 | 49.3 | 11.0 |
| 500 | 195.8 | 43.4 |
| 1000 | 780.3 | 172.9 |
| 2000 | 3114.9 | 690.5 |

## Two Curves Fitting the Previous Data

If we plot these numbers on a graph and try to fit curves to them, we find that they lie on the following two curves:

$$f_1(n) = 0.0007772n^2 + 0.00305n + 0.001$$

$$f_2(n) = 0.0001724n^2 + 0.00040n + 0.100$$

## Discussion

- The curves on the previous slide have the **quadratic** form $f(n) = an^2 + bn + c$
- The difference between the two curves is that they have **different constants** $a, b$ and $c$
- Even if we implement `SelectionSort` on another computer using another programming language and another compiler, the curve that we will get will be of the same form.
- So, even though the particular measurements will change under different circumstances, **the shape of the curve** will remain the same.

## Complexity Classes

- The running times of various algorithms belong to different **complexity classes**.
- Each complexity class is characterized by a **different family of curves**.
- All of the curves in a given complexity class share **the same basic shape**. The shape is characterized by an equation that gives running times as a function of problem size.

## $O$-notation

- This notation is used in Computer Science for taking about the time complexity of an algorithm.
- For `SelectionSort`, the time complexity is $O(n^2)$
- We find this complexity by taking the **dominant term** of the expression $an^2 + bn + c$ and throwing away the constant coefficient $a$.

## $O$-notation (cont'd)

- Let us consider the equation with , and . Then we have the following table:

| $n$ | $f(n)$ | $an^2$ | $n^2$ **term as % of total** |
|---|---|---|---|
| 125 | 2.8 | 2.7 | 94.7 |
| 250 | 11.0 | 10.8 | 98.2 |
| 500 | 43.4 | 43.1 | 99.3 |
| 1000 | 172.9 | 172.4 | 99.7 |
| 2000 | 690.5 | 689.6 | 99.9 |

## $O$-notation (cont'd)

- We conclude that **the lesser term** $bn + c$ **contributes very little** to the value of even though $c$ is 250 times more than $b$ and $b$ is more than two times $a$. Thus we can **ignore this lesser term.**

- We will also **ignore the constant of proportionality** $a$ in $an^2$ since we want to concentrate in the general shape of the curve. $a$ **will differ for different implementations on different computers.**

## Some Common Complexity Classes

| $O$-notation | Adjective Name |
|---|---|
| $O(1)$ | Constant |
| $O(\log n)$ | Logarithmic |
| $O(n)$ | Linear |
| $O(n \log n)$ | Quasi-linear |
| $O(n^2)$ | Quadratic |
| $O(n^3)$ | Cubic |
| $O(2^n)$ | Exponential |
| $O(10^n)$ | Exponential |
| $O(2^{2^n})$ | Doubly exponential |

## Comparing Complexity Classes

- Let us assume that we have an algorithm A that runs on a computer that executes one step of this algorithm every microsecond.
- Let us assume that is the number of steps required by A to solve a problem of size $n$.
- Then we have the following table.

## Running Times for Algorithm A

| $f(n)$ | $n = 2$ | $n = 16$ | $n = 256$ | $n = 1024$ |
|---|---|---|---|---|
| 1 | 1 µsec | 1 µsec | 1 µsec | 1 µsec |
| $\log_2 n$ | 1 µsec | 4 µsec | 8 µsec | 10 µsec |
| $n$ | 2 µsec | 16 µsec | 256 µsec | 1.02 ms |
| $n \log_2$ | 2 µsec | 64 µsec | 2.05 ms | 10.2 ms |
| $n^2$ | 4 µsec | 25.6 µsec | 65.5 ms | 1.05 |
| $n^3$ | 8 µsec | 4.1 ms | 16.8 ms | 17.9 min |
| $2^n$ | 4 µsec | 65.5 ms | $10^{63}$ years | $10^{297}$ years |

## Size of Largest Problem Algorithm We Can Solve in

| Number of steps | T = 1 min | T = 1hr |
|---|---|---|
| $n$ | $6 \times 10^7$ | $3.6 \times 10^9$ |
| $n \log_2 n$ | $2.8 \times 10^6$ | $1.3 \times 10^8$ |
| $n^2$ | $7.75 \times 10^3$ | $6.0 \times 10^4$ |
| $n^3$ | $3.91 \times 10^2$ | $1.53 \times 10^3$ |
| $2^n$ | $25$ | $31$ |
| $10^n$ | $7$ | $9$ |

## Time Complexity Discussion

- $1$: This is the case when all the statements in a program are executed a **constant** number of times.
- $\log n$ When the time complexity of an algorithm is **logarithmic**, the algorithm runs a little bit slower when increases. This time complexity is found in algorithms that solve a problem by transforming it into a series of smaller problems, reducing in each step the size of the problem by a constant amount. Every time doubles, increases only by a constant.
- $n$ When the time complexity of an algorithm is **linear** what happens usually is that a small part of the processing takes place for each element of the input. When doubles, the running time of the algorithm doubles too. This time complexity is optimal for an algorithm that needs to process inputs or to output outputs.

## Time Complexity Discussion (cont'd)

- $n \log n$ This time complexity appears when an algorithm solves a problem by dividing it into smaller problems and combining the partial solutions. When doubles, the time complexity more than doubles (but it is not very far from the double).
- $n^2$ When the time complexity is **quadratic**, the algorithm is practically useful only for small problems. Quadratic running times usually appear in algorithms that process pairs of elements of a problem (e.g., with two nested loops). When doubles, the running time increases four times.
- $n^3$ Similarly, an algorithm that processes triples of elements of a problem (e.g., usually with three nested loops) has **cubic** running time. It is useful only for small problem sizes. When doubles, the running time increases eight times.
- $2^n$ When the time complexity of an algorithms is **exponential**, the algorithm can be used in practice only for very small problem sizes. This is

## Time Complexity

We have to consider the following cases:

- Worst case
- Best case
- Average case

## Well-Known Algorithms and Their Time Complexity

- Sequential searching of an array: $O(n)$
- Binary searching of a sorted array: $O(\log n)$
- Hashing (under certain conditions): $O(1)$
- Searching using binary search trees: $O(\log n)$
- SelectionSort, InsertionSort: $O(n^2)$
- QuickSort, HeapSort, MergeSort: $O(n \log n)$
- String pattern matching: $O(n)$
- Multiplying two square x matrices: $O(n^3)$
- Traveling salesman, graph coloring: $O(2^n)$

## Formal Definition of $O$-notation

We say that $f(n)$ **is** $O(g(n))$ if there exist two positive constants $K$ and $n_0$ such that $|f(n)| \leq K|g(n)|\ \forall\, n \geq n_0$.

## Three Ways of Saying it in Words

Let us assume that $f$ and $g$ are positive functions. Then:

- $f(n)$ is $O(g(n))$ provided the curve $K$ X $g(n)$ can be made to lie above the curve for $f(n)$ whenever we are to the right of some big enough value of $n_0$.

- $f(n)$ is $O(g(n))$ if there is some way to choose a constant of proportionality $K$ so that the curve $f(n)$ for is bounded above by the curve for $K$ X $g(n)$ whenever is big enough (i.e., when $n \geq n_0$).

- $f(n)$ is $O(g(n))$ if for all but finitely many small values of $n$, the curve for $f(n)$ lies below the curve for some suitably large constant multiple of $g(n)$

## Example of a Formal Proof

- Let us suppose that a sorting algorithm A sorts a sequence of numbers in ascending order with number of steps $f(n) = 3 + 6 + 9 + \cdots + 3n$
- We will show that the algorithm runs in $O(n^2)$ steps.
- **Proof:** We will first find a closed form for $f(n)$

## Proof (cont'd)

- Note that $f(n) = 3 + 6 + 9 + \cdots + 3n = 3(1 + 2 + \cdots + n) = 3\frac{n(n+1)}{2}$.
- Then, choosing $K = 3,\ n_0 = 1$ and $g(n) = n^2$, we can show that for all $n \geq 1$, the following inequality holds:

$$3\frac{n(n+1)}{2} \leq 3n^2$$

## Proof (cont'd)

- Multiplying both sides of the above inequality with $\frac{2}{3}$ gives $n^2 + n \leq 2n^2$ .
- Subtracting $n^2$ from both sides gives $n \leq n^2$.
- Dividing this inequality by $n$ gives $n \geq 1$.
- The proof is now complete.

## Practical Shortcuts for Manipulating $O$-notation

- In practice we can deal with $O$-notation in an easier way by separating the expression for $f(n)$ into a **dominant** term and **lesser** terms and throwing away the lesser terms.
- In other words: $O(f(n))$ = $O($ dominant term $\pm$ lesser terms $) =$ $O($ dominant term $)$

## Scale of Strength for $O$-notation

- We can rank the usual complexity functions on the following **scale of strength** so it is easy to determine the dominant term and the lesser terms:
  $O(1) < O(\log n) < O(n) < O(n^2) < O(n^3) < O(2^n) < O(10^n)$

## Example

- $O(6n^3 - 15n^2 + 3n \log n) = O(6n^3) = O(n^3)$
- Let us see why we are allowed to do the above. Notice that we have $6n^3 - 15n^2 + 3n \log n < 6n^3 + 3n \log n < 6n^3 + 3n^3 < 9n^3$
- This is the inequality that the definition of $O$-notation needs for $K = 9$ and $n \geq 1$

# Ignoring Bases of Logarithms

- When we use $O$-notation, we can **ignore the bases of logarithms** and assume that all logarithms are in base 2.
- Changing the bases of logarithms involves **multiplying by constants**, and constants of proportionality are ignored by $O$-notation.
- For example, $\log_{10} n = \frac{\log_2 n}{\log_2 10}$ . Notice now that $\frac{1}{\log_2 10}$ is a constant.

## $O(1)$

- It is easy to see why the $O(1)$ notation is the right one for constant time complexity.
- Suppose that we can prove that an algorithm A runs in a number of steps $f(n)$ that are always less than K steps for all $n$. Then $f(n) \leq K$ X $1$ for all $n \leq 1$ Therefore $f(n)$ is $O(1)$.

## Some Algorithms and Their Complexity

- Sequential searching
- Selection sort
- Recursive selection sort
- Towers of Hanoi

## Analysis of Sequential Searching

- Suppose we have an array `A[0:n-1]` that contains distinct keys $K_i$ $1 \leq i \leq n)$ and assume that $K_i$ is stored in position `A[i-1]`.
- **Problem**: we are given a key $K$ and we would like to determine its position in `A[0:n-1]`

## An Algorithm for Sequential Searching

```
#define n 100
typedef int Key;
typedef Key SearchArray[n];

int SequentialSearch(Key K, SearchArray A) {
    int i;
    for (i =0 ; i < n; ++i) {
       if (K == A[i]) return i;
    }
    return(-1);
}
```

## Complexity Analysis

- The amount of work done to locate key $K$ **depends on its position** in `A[0:n-1]`
- For example, if $K$ is in `A[0]`, then we need only one comparison.
- In general, if $K$ is in `A[i-1]`, then we need $i$ comparisons.

# Complexity Analysis (cont'd)

- **Best case**: This is when $K$ is in `A[0]`. The complexity is $O(1)$
- **Worst case**: This is when $K$ is in `A[n-1]`. The amount of work is $an + b$ where a and b are constants. Therefore the complexity is $O(n)$.
- **Average case**: Let us assume that each key is equally likely to be used in a search. The average can then be computed by taking the total of all the work done for finding all the different keys and dividing by $n$.

# Complexity Analysis (cont'd)

- The work needed to find the i-th key $K_i$ is of the form $ai + b$ for some constants and a and b. Therefore:

$$Total = \sum_{i=1}^{n}(ai + b) = a\sum_{i=1}^{n} i + \sum_{i=1}^{n} b = a\frac{n(n+1)}{2} + bn$$

- Now the average is:

$$Average = \frac{Total}{n} = a\frac{n+1}{2} + b = \frac{a}{2}n + (\frac{a}{2} + b)$$

- Therefore the average is $O(n)$

## Selection Sorting Algorithm

```
void SelectionSort(InputArray A) {

    int MinPosition, temp, i, j;

    for (i = n - 1; i > 0; --i) {
        MinPosition = i;
        for (j = 0; j < i; ++i) {
            if (A[j] < A[MinPosition]) {
                MinPosition = j;
            }
        }
        temp = A[i];
        A[i] = A[MinPosition];
        A[MinPosition] = temp;
    }
}
```

## Complexity Analysis of SelectionSort

- We start from the inner `for` statement. The `if` statement inside the `for` takes a constant amount of time $a$. Thus, the `for` statement takes $ia$ time units.
- Let us now consider the outer `for`. The statements inside this `for`, except the inner `for`, take a constant amount of time $b$. Thus all the statements inside the outer `for` take time $ai + b$

## Complexity Analysis (cont'd)

- The outer `for` takes time $\sum_{i=1}^{n}(ai+b) = a\sum_{i=1}^{n} i + \sum_{i=1}^{n} b = a\frac{(n-1)n}{2} + (n-1)b = \frac{a}{2}n^2 + (b - \frac{a}{2})n - b$
- Therefore the time complexity of the algorithm is $O(n^2)$

## Recursive SelectionSort

```
// FindMin is an auxiliary function used by the Selection sort bel
int FindMin(InputArray A, int n) {
    int i,j = n;
    for (i = 0; i < n; ++i)
       if (A[i]<A[j]) j=i;
    return j;
}

void SelectionSort(InputArray A, int n) {
    int MinPosition, temp;
    if (n > 0) {
        MinPosition = FindMin(A,n);
        temp = A[n]; A[n] = A[MinPosition]; A[MinPosition] = temp;
        SelectionSort(A, n-1);
    }
}
```

## Analysis of Recursive SelectionSort

- To use this recursive version of `SelectionSort` to perform selection sorting on the array `A[0:n-1]`, we make the function call `SelectionSort(A,n-1)`.
- The first thing we need to do is to analyze the running time of function `FindMin` which finds the position of the smallest element in the array `A[0:n]`.
- It is easy to see that the time for this function is $an + b_1$ for suitable constants $a$ and $b_1$.

## Analysis of Recursive SelectionSort (cont’d)

- We now analyze the running time of recursive function `SelectionSort`.
- Let $T(n)$ stand for the cost, in time units, of calling `SelectionSort` on `A[0:n]`.
- Then the costs in SelectionSort are as follows:

```
    if (n > 0) {
        Cost an+b1
        Cost b2
        Cost T(n-1)
    }
```

## Analysis of Recursive SelectionSort (cont’d)

- If $b = b_1 + b_2$ then the following **recurrence relation** holds for $n > 0$ : $T(n) = an + b + T(n-1)$.
- The base case of this recurrence relation is $T(0) = c$ where is the cost of executing `SelectionSort(A,0)`.
- To solve such recurrence relations, we can use a method called **unrolling**.

## Analysis of Recursive SelectionSort (cont'd)

$T(n) = an + b + T(n-1)$
$T(n) = an + b + a(n-1) + b + T(n-2)$
$T(n) = an + b + a(n-1) + b + a(n-2) + b + T(n-3)$
$\ldots$
$T(n) = an + b + a(n-1) + b + a(n-2) + b + \cdots + a \times 1 + b +$
$T(0)$
$T(n) = an + b + a(n-1) + b + a(n-2) + b + \cdots + a \times 1 + b + c$

## Analysis of Recursive SelectionSort (cont’d)

Rearranging some of the terms so that all those with coefficients and are collected together, we have:

$$T(n) = (an + a(n-1) + a(n-2) + \cdots + a) + nb + c$$

$$T(n) = \sum_{i=1}^{n}(ai) + nb + c = a\frac{n(n+1)}{2} + nb + c = \frac{a}{2}n^2 + (\frac{a}{2}b)n + c$$

## Analysis of Recursive SelectionSort (cont'd)

Therefore $T(n)$ but also $T(n-1)$ is $O(n^2)$.

## A Recursive Solution to the Towers of Hanoi

```
void MoveTowers(int n, int start, int finish, int spare) {
    if (n == 1) {
        printf("Move a disk from peg %1d to peg %1d\n", start, fin
    } else {
        MoveTowers(n-1, start, spare, finish);
        printf("Move a disk from peg %1d to peg %1d\n", start, fin
        MoveTowers(n-1, spare, finish, start);
    }
}
```

## Analysis of Towers of Hanoi

- Let $n$ be the number of towers to be moved. Then the running time $T(n)$ of the algorithm is given by the following recurrence relations: $T(1) = a\, T(n) = b + 2T(n-1)$

- We will solve these recurrence relations using the technique of **unrolling plus summation**.

## Analysis of Towers of Hanoi (cont’d)

$$T(n) = b + 2T(n-1)$$
$$T(n) = b + 2(b + 2T(n-2))$$
$$T(n) = b + 2b + 2^2 T(n-2)$$
$$T(n) = b + 2b + 2^2(b + 2T(n-3))$$
$$T(n) = b + 2b + 2^2 b + 2^3 T(n-3)$$
$$\ldots$$
$$T(n) = b + 2b + 2^2 b + 2^{(i-1)} b + 2^i T(n-i)$$

## Analysis of Towers of Hanoi (cont'd)

- When $i = n - 1$ we have: $T(n-i) = T(n-(n-1)) = T(n-n+1) = T(1) = a$
- Therefore, can be expressed as follows: $T(n) = 2^0 b + 2^1 b + 2^2 b + \cdots + 2^{(n-2)} b + 2(n-1)a = \sum_{i=0}^{n-2} 2^i b + 2(n-1)a = b \sum_{i=0}^{n-2} 2^i + 2(n-1)a$.

## Analysis of Towers of Hanoi (cont'd)

- Now we can see that the sum is a standard **geometric progression**. So we will use the fact that $\sum_{k=0}^{m} x^k = \frac{x^{m+1}-1}{x-1}$ to conclude the following: $\sum_{i=0}^{n-2} 2^i = \frac{2^{(n-1)}-1}{2-1} = 2^{(n-1)} - 1$

## Analysis of Towers of Hanoi (cont’d)

- Therefore: $T(n) = b(2^{(n-1)} - 1) + 2^{(n-1)}a = (a+b)2^{(n-1)} - b = \frac{a+b}{2}2^n - b$
- Finally, we can see that $T(n)$ is $O(2^n)$

## What $O$-notation Does Not Tell You

- $O$-notation **does not apply to small problem sizes** because in this case the constants might dominate the other terms.
- One can use **experimental testing** to select the best algorithm in this case.
- Experimental testing is also useful if we want to compare algorithms that are **in the same complexity class.**

## Space Complexity

- In a similar way, we can measure the **space complexity** of an algorithm.

## Other notations

- There also other complexity notations such as $o(n), \Theta(n), \Omega(n), \omega(n)$
- More details in the Algorithms course.

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C.*
- Chapter 6.
- Robert Sedgewick. Αλγόριθμοι σε C.
- Κεφ. 2.

# Lists and Strings

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Under construction

## A List ADT

A **list** *L* of items of type *T* is a sequence of items of type *T* on which the following operations are defined:

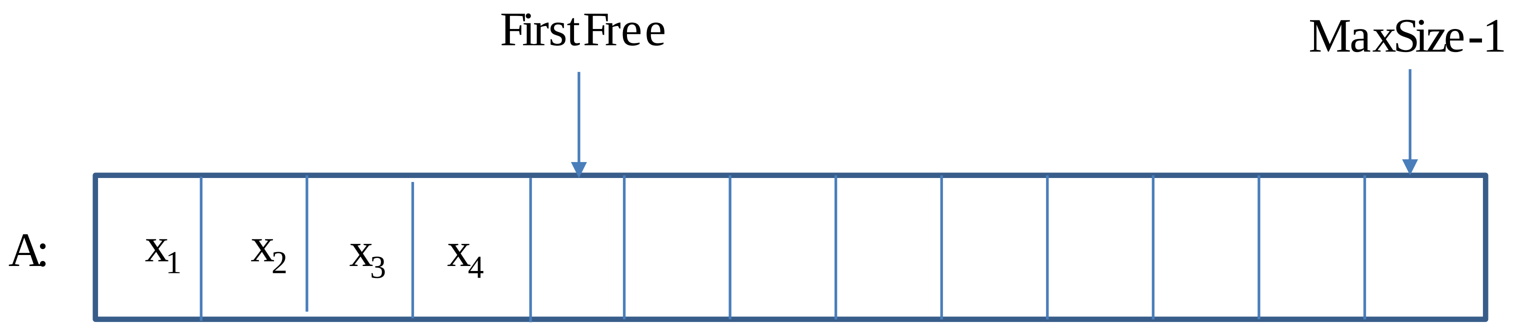
- **Initialize** the list *L* to be the empty list.
- Determine whether or not the list *L* is **empty**.
- Find the length of a list *L* (where the **length** of *L* is the number of items in *L* and the length of the empty list is 0).
- **Select** the -th item of a list *L*, where $1 \leq i \leq length(L)$.
- **Replace** the -th item *X* of a list *L* with a new item *Y* where $1 \leq i \leq length(L)$
- **Delete** any item *X* from a nonempty list *L*.
- **Insert** a new item *X* into a list *L* in any arbitrary position (such as before the first item of *L*, after the last item of *L* or between any two items of *L*).

## Lists

- Lists are more general kinds of containers than stacks and queues.
- Lists can be represented by **sequential representations** and **linked representations**.

## Sequential List Representations

- We can use an array `A[0:MaxSize-1]` as we show graphically (items are stored **contiguously**):



## Advantages and Disadvantages

Advantages:

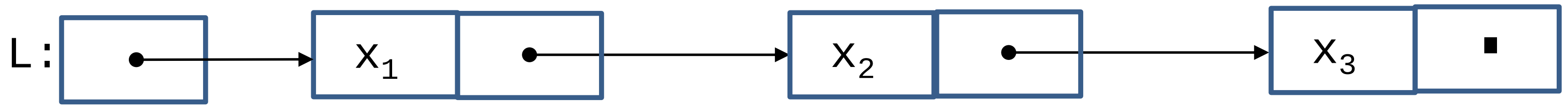
- Fast access to the i-th item of the list in *O(1)* time.

Disadvantages:

- Insertions and deletions may require shifting all items i.e., an *O(n)* cost on the average.
- The size of the array should be known in advance. So if we have small size, we run the risk of overflow and if we have large size, we will be wasting space.

# One-Way Linked Lists Representation

- We can use chains of linked nodes as shown below:



## Declaring Data Types for Linked Lists

- The following statements declare appropriate data types for our linked lists from earlier lectures:

```c
typedef char AirportCode[4];
typedef struct NodeTag {
    AirportCode Airport;
    struct NodeTag* Link;
} NodeType;
typedef NodeType* NodePointer;
```

- We can now define variables of these datatypes:

```c
NodePointer L;
```

- or equivalently

```c
NodeType* L;
```

## Accessing the *ith* Item

```
void PrintItem(int i, NodeType* L) {
    while ((i > 1) && (L != NULL)) {
        L = L->Link;
        i--;
    }
    if ((i == 1) && (L != NULL)) {
        printf("%s", L->Item);
    } else {
        printf("Error -- attempt to print an item that is not on the
    }
}
```

# Computational Complexity

- Suppose that list *L* has exactly n items. If it is equally likely that each of these items can be accessed, then the average number of n pointers followed to access the *ith* item is:
  Average = $\frac{(1+2+\cdots+n)}{n} = \frac{\frac{n(n+1)}{2}}{n} = \frac{n}{2} + \frac{1}{2}$
- Therefore, the average time to access the *ith* item is *O(n)*.
- The complexity bound is the same for inserting before or after the *ith* item or deleting it or replacing it.

## Comparing Sequential and Linked List Representations

| List Operation | Sequential | Linked list |
|---|---|---|
| Finding length of L | $O(1)$ | $O(n)$ |
| Inserting a new first item | $O(n)$ | $O(1)$ |
| Deleting the last item | $O(1)$ | $O(n)$ |
| Replacing the $ith$ item | $O(1)$ | $O(n)$ |
| Deleting the $ith$ item | $O(n)$ | $O(n)$ |

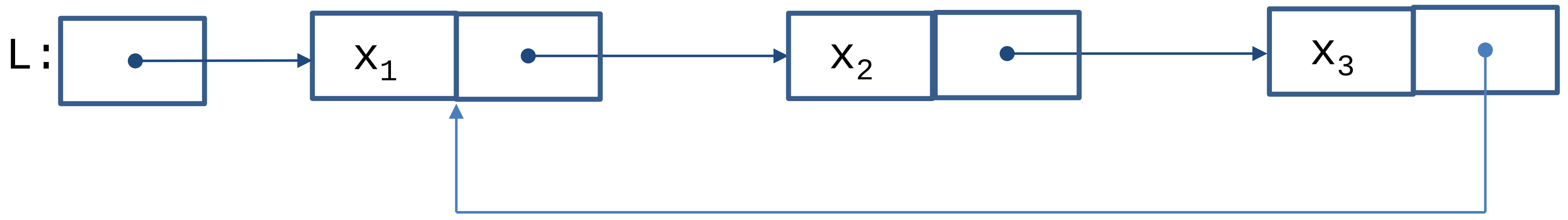
The above table gives **average running times**. But time is not the only resource that is of interest. **Space** can also be an important resource in some applications.

## Other Linked List Representations

- Circular linked lists
- Two-way linked lists
- Linked lists with header nodes

## Circular linked lists

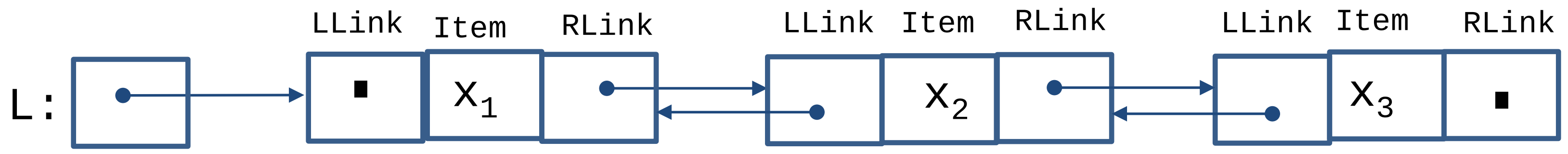
- A **circular linked list** is formed by having the link in the last node of a one-way linked list point back to the first node.



- The advantage of a circular linked list is that any node on it is accessible by any other node.

## Two-Way Linked Lists

- Two-way linked lists are formed from nodes that have pointers to both their right and left neighbors on the list.

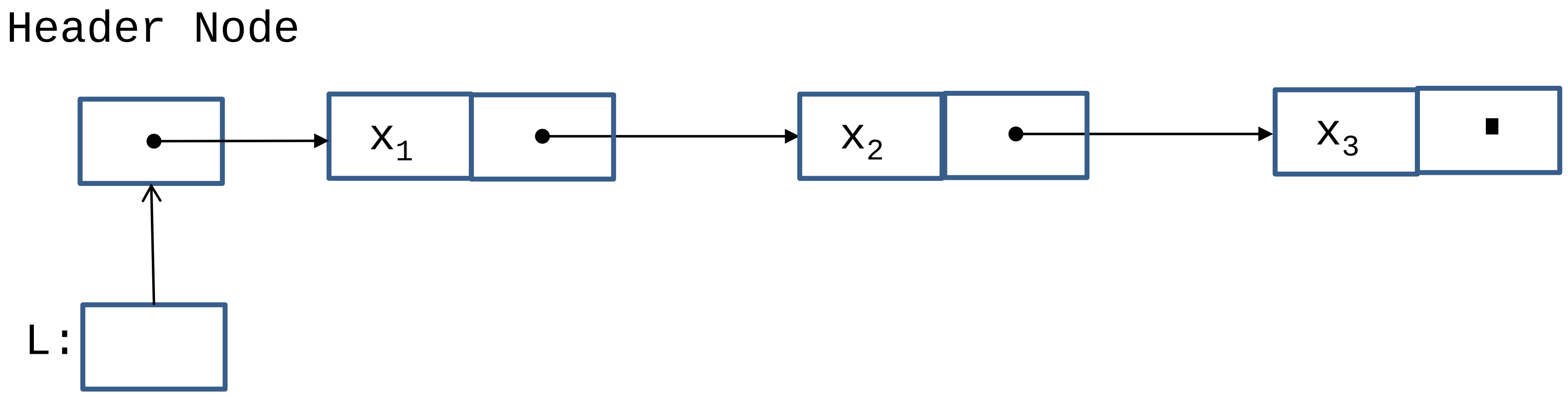


## Two-Way Linked Lists (cont’d)

- Given a pointer to a node *N* in a two-way linked list, we can follow links in either direction to access other nodes.
- We can insert a node *M* either before or after *N* starting only with the information given by the pointer to *N*.

# Linked Lists with Header Nodes

- Sometimes it is convenient to have a special **header node** that points to the first node in a linked list of item nodes.



## Linked Lists with Header Nodes (cont’d)

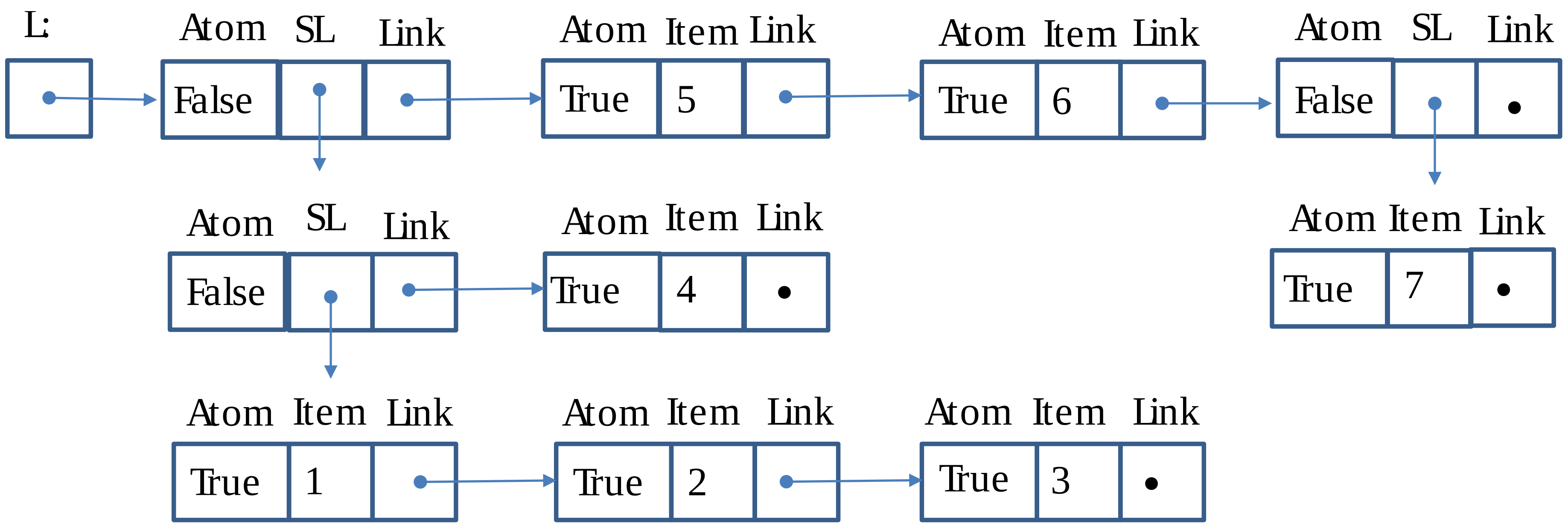
- Header nodes can be used to hold information such as the number of nodes in the list etc.

## Generalized Lists

- A **generalized list** is a list in which the individual list items are permitted to be sublists.
- **Example**: $(a_1, a_2, (b_1, (c_1, c_2), b_3), a_4, (d_1, d_2), a_6)$
- If a list item is not a sublist, it is said to be **atomic**.
- Generalized lists can be represented by sequential or linked representations.

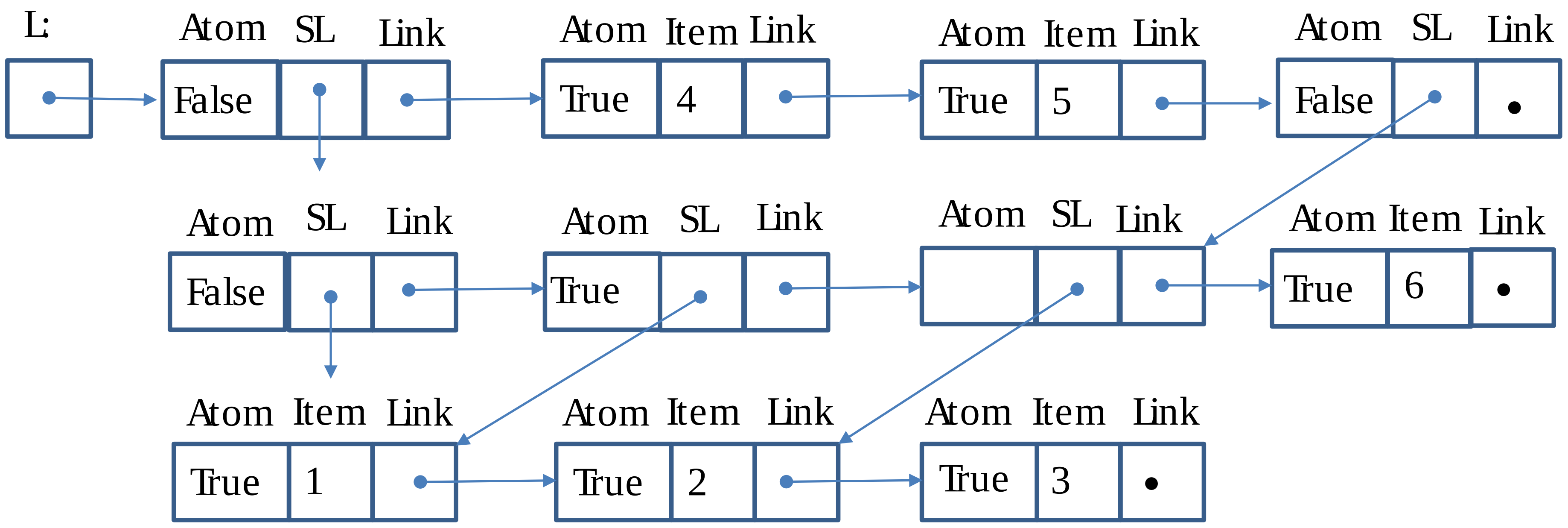
# Generalized Lists (cont'd)

- The generalized list $L = (((1, 2, 3), 4), 5, 6, (7))$ can be represented without shared sublists as follows:



# Generalized Lists (cont'd)

- The generalized list $L = (((1,2,3),(1,2,3),(2,3),6),4,5,((2,3),6))$ can be represented with shared sublists as follows:



## A Datatype for Generalized List Nodes

```
typedef struct GenListTag {
    GenListTag* Link;
    int Atom;
    union SubNodeTag {
        ItemType Item;
        struct GenListTag* Sublist;
    } SubNode;
} GenListNode;
```

## Printing Generalized Lists

```
void PrintList(GenListNode* L) {
    GenListNode* G;
    printf("(");
    G = L;
    while (G != NULL) {
        if (G->Atom) {
            printf("%d", G->SubNode.Item);
        } else {
            printList(G->SubNode.SubList);
        }
        if (G->Link != NULL) printf(",");
        G = G->Link;
    }
    printf(")");
}
```

## Applications of Generalized Lists

- Artificial Intelligence programming languages LISP and Prolog offer generalized lists as a language construct.
- Generalized lists are often used in Artificial Intelligence applications.
- More in the courses “Artificial Intelligence” and “Logic Programming”.

## Strings

**Strings** are sequences of characters. They have many applications:

- Word processors
- E-mail systems
- Databases
- . . .

## Strings in C

- A **string** in C is a sequence of characters terminated by the null character `\0`.
- **Example**: To represent a string `S=="canine"` in C, we allocate a block of memory B at least seven bytes long and place the characters “canine” in bytes `B[0:5]`. Then, in byte `B[6]`, we place the character `\0`.

# A String ADT

- In C’s **standard library** you can access a collection of useful string operations by including the header file `<string.h>` in your program.
- These functions define a **predefined string ADT.**

## Examples of String Operations

Let us assume that S and T are string variables (i.e., of type char* ). Then:

- `strlen(S)`: returns the number of characters in string S (not including the terminating character `\0`).
- `strstr(S,T)`: returns a pointer to the first occurrence of string S in string T (or `NULL` if there is no occurrence of string S in string T).
- `strcat(S,T)`: concatenate a copy of string T to the end of string S and return a pointer to the beginning of the enlarged string S.
- `strcpy(S,T)`: make a copy of the string T including a terminating last character `\0`, and store it starting at the location pointed to by the character pointer S.

## Concatenating Two Strings

```c
char* Concat(char* S, char* T) {
    char* P;
    char* temp;
    P = malloc(1 + strlen(S) + strlen(T));
    temp = P;
    while ((* P++ = * S++) != '\0');
    P--;
    while ((* P++ = * T++) != '\0');
    return(temp);
}
```

## Readings

- T. A. Standish. *Data Structures, Algorithms and Software Principles in C*.
- Chapter 8, Sections 8.1-8.5.
- Robert Sedgewick. Αλγόριθμοι σε C.
- Κεφ. 3.

# Makefiles

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Under construction

## The Utility make

- The make utility automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
- It is useful when we write large programs that are contained in more than one file.

## Preparing and Running make

- In order to use make, you should create a file named **Makefile**. This file describes the relationships among files in your program and provides commands for updating each file.
- In a program, typically, the **executable** file is updated from **object files**, which in turn are made by compiling **source files**.
- Once a suitable makefile exists, each time you change some source files, the simple shell command

```
make
```

- suffices to perform all necessary recompilations.
- The make program uses the **makefile** in the current directory and the **last-modification times** of the files to decide which of the files need to be updated. For each of those files, it issues the **recipes** recorded in the makefile.

## Rules

- A simple makefile consists of **rules** with the following syntax:

```
target ... : prerequisites ...
        recipe
        ...
        ...
```

- A **target or a goal** is usually the name of a file that is generated by a program. Examples of targets are executable or object files. A target can also be the name of an action to carry out, such as ‘clean’.
- A **prerequisite** is a file that is used as input to create the target. A target often depends on several files.
- A **recipe** is an action that make carries out. A recipe may have more than one command.
- Prerequisites are **optional**. For example, the rule containing the delete command associated with the target ‘clean’ does not have prerequisites.

- A **rule** explains how and when to remake certain files which are the targets of the particular rule. make carries out the recipe on the prerequisites to create or update the target.

## Example

```
pqsort : sorting.o PQImplementation.o
  gcc sorting.o PQImplementation.o -o pqsort

sorting.o :sorting.c PQInterface.h PQTypes.h
  gcc -c sorting.c

PQImplementation.o :PQImplementation.c PQInterface.h PQTypes.h
  gcc -c PQImplementation.c

clean:
  rm pqsort sorting.o PQImplementation.o
```

**Mind the tabs!!!**

## Comments

- In this example makefile, the **targets** include the executable file pqsort, and the object files sorting.o and PQImplementation.o.
- The **prerequisites** are files such as sorting.c and PQInterface.h and PQTypes.h
- **Recipes** include commands like gcc -c sorting.c and gcc -c PQImplementation.c
- A **recipe** may follow each line that contains a target and prerequisites. These recipes say how to update the target file. **Important**: A **tab character** must come at the beginning of every line that contains a recipe to distinguish recipes from other lines in the makefile.

## Comments (cont'd)

- The target clean is not a file, but merely the name of an action. Notice that clean is not a prerequisite of any other rule. Consequently, make never does anything with it unless you tell it specifically. Note also that the rule for clean does not have any prerequisites, so the only purpose of the rule is to run the specified recipe. Targets that do not refer to files but are just actions are called **phony targets**.

## How make is invoked

- For the previous example, after some of the source .c files have changed, and we would like to create a new executable, we just write make on the command line.

## How make Processes a Makefile

- When make is called, it reads the makefile in the current directory and starts processing the first rule of the makefile. In our case, this is the rule for the executable file pqsort.
- However, before make can process this rule, it should process the rules that update the files on which pqsort depends i.e., sorting.o and PQImplementation.o.
- These in turn depend on files such as sorting.c, PQInterface.h and PQTypes.h which are not the targets of any rule so the recursion stops here.

## Variables

- **Variables** allow a text string to be defined once and substituted in multiple places later.
- For example, it is standard practice for every makefile to have a variable named objects, which is defined to be a list of all object file names.
- We can define this variable by writing

```
    objects = sorting.o PQImplementation.o
```

- Then the variable can be used in the makefile using the notation $(variable).

## Example (cont'd)

```
objects = sorting.o PQImplementation.o

pqsort : $(objects)
  gcc $(objects) -o pqsort

sorting.o :sorting.c PQInterface.h PQTypes.h
  gcc -c sorting.c

PQImplementation.o :PQImplementation.c PQInterface.h PQTypes.h
  gcc -c PQImplementation.c

clean:
  rm pqsort $(objects)
```

## Letting make Deduce the Recipes

- It is not necessary to spell out the recipes for compiling the individual C source files, because make can figure them out.
- make has an **implicit rule** for updating a .o file from a correspondingly named .c file using a cc -c command (not gcc).
- So we can write our example as follows.

## Example (cont'd)

```
objects = sorting.o PQImplementation.o

pqsort : $(objects)
  gcc $(objects) -o pqsort

sorting.o : PQInterface.h PQTypes.h

PQImplementation.o : PQInterface.h PQTypes.h

clean:
  rm pqsort $(objects)
```

## Rules for Cleaning the Directory

- We can use makefiles to do other things except compiling programs. For example, we can have a recipe that deletes all the object files and executables so that the directory is clean.
- In our example, this is done by the following rule:

```
clean:
rm pqsort $(objects)
```

clean here is called a **phony target**.

- To avoid problems with files with the name clean in the same directory, you can write the above rule as follows:

```
.PHONY clean
clean:
rm pqsort $(objects)
```

## Rules for Cleaning the Directory (cont'd)

- You can execute the above rule by executing the shell command

```
make clean
```

## Readings

- These slides were created by copying (sometimes verbatim!) material from the manual http://www.gnu.org/software/make/manual/make.html
- Read this manual for more information (just reading Chapter 2 will suffice).

# Data Structures for Disjoint Sets

K08 Δομές Δεδομένων και Τεχνικές Προγραμματισμού

Under construction

## Dynamic Sets

- Sets are fundamental for mathematics but also for computer science.
- In computer science, we usually study **dynamic sets** i.e., sets that can grow, shrink or otherwise change over time.
- The data structures we have presented so far in this course offer us ways to represent **finite, dynamic sets** and manipulate them on a computer.

## Dynamic Sets and Symbol Tables

- Many of the data structures we have so far presented for symbol tables can be used to implement a dynamic set (e.g., a linked list, a hash table, a (2,4) tree etc.).

## Disjoint Sets

- Some applications involve grouping distinct elements into a collection of **disjoint sets (ξένα σύνολα)**.
- Important operations in this case are to construct a set, to find which set a given element belongs to, and to unite two sets.

## Definitions

- A **disjoint-set data structure** maintains a collection $S = S_1, S_2, \ldots, S_n$ of disjoint dynamic sets.
- Each set is identified by a **representative (αντιπρόσωπο)**, which is some member of the set.
- The disjoint sets might form a **partition (διαμέριση)** of a universe set

## Definitions (cont'd)

The disjoint-set data structure supports the following operations:

- $Make-Set(x)$: It creates a new set whose only member (and thus representative) is pointed to by $x$. Since the sets are disjoint, we require that $x$ not already be in any of the existing sets.

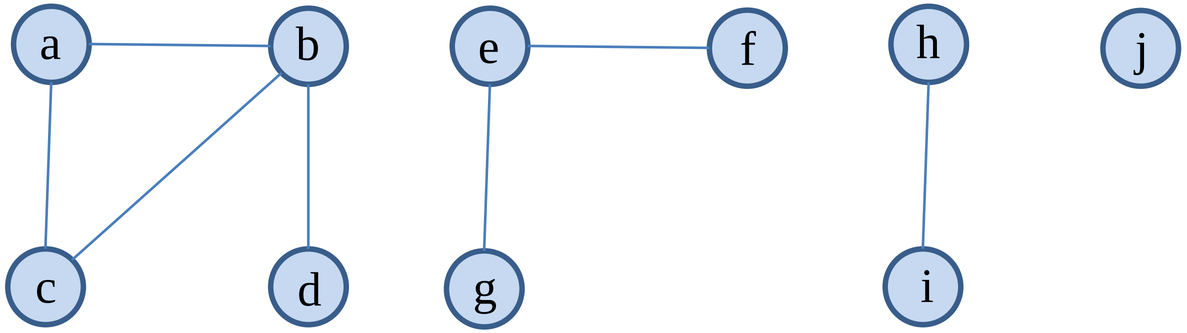
- $Union(x)$: It unites the dynamic sets that contain $x$ and $y$, say $S_x$ and $S_y$, into a new set that is the union of these two sets. One of the $S_x$ and $S_y$ and give its name to the new set and the other set is "destroyed" by removing it from the collection S. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is some member of $S_x \cup S_y$ (usually the representative of the set that gave its name to the union).

- $Find-Set(x)$ returns a pointer to the representative of the unique set containing x.

## Determining the Connected Components of an Undirected Graph

- One of the many applications of disjoint-set data structures is **determining the connected components (συνεκτικές συνιστώσες) of an undirected graph**.
- The implementation based on disjoint-sets that we will present here is appropriate when the edges of the graph are not static e.g., when **edges are added dynamically** and we need to maintain the connected components as each edge is added.

## Example Graph



## Computing the Connected Components of an Undirected Graph

- The following procedure Connected-Components uses the disjoint-set operations to compute the connected components of a graph.

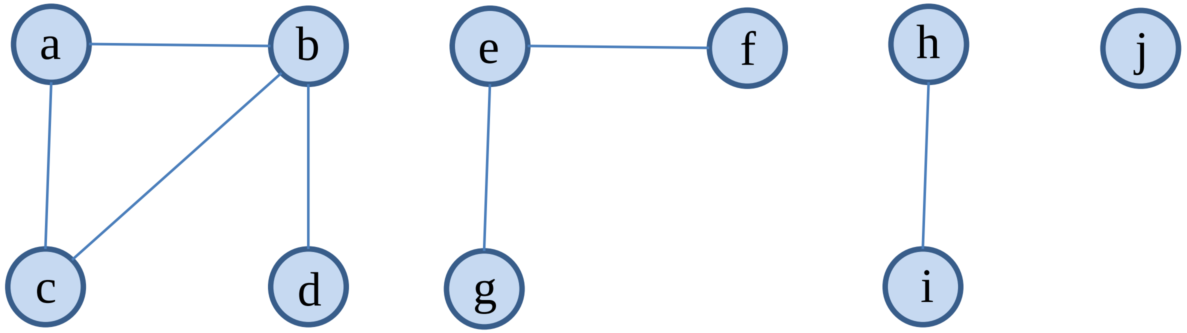
```
Connected-Components(G)
  for each vertex v in  V[G]
    do Make-Set(v)
  for each edge(u,v) in E(G)
      do if not(Find-Set(u) = Find-Set(v))
        then Union(u,v)
```

## Computing the Connected Components (cont’d)

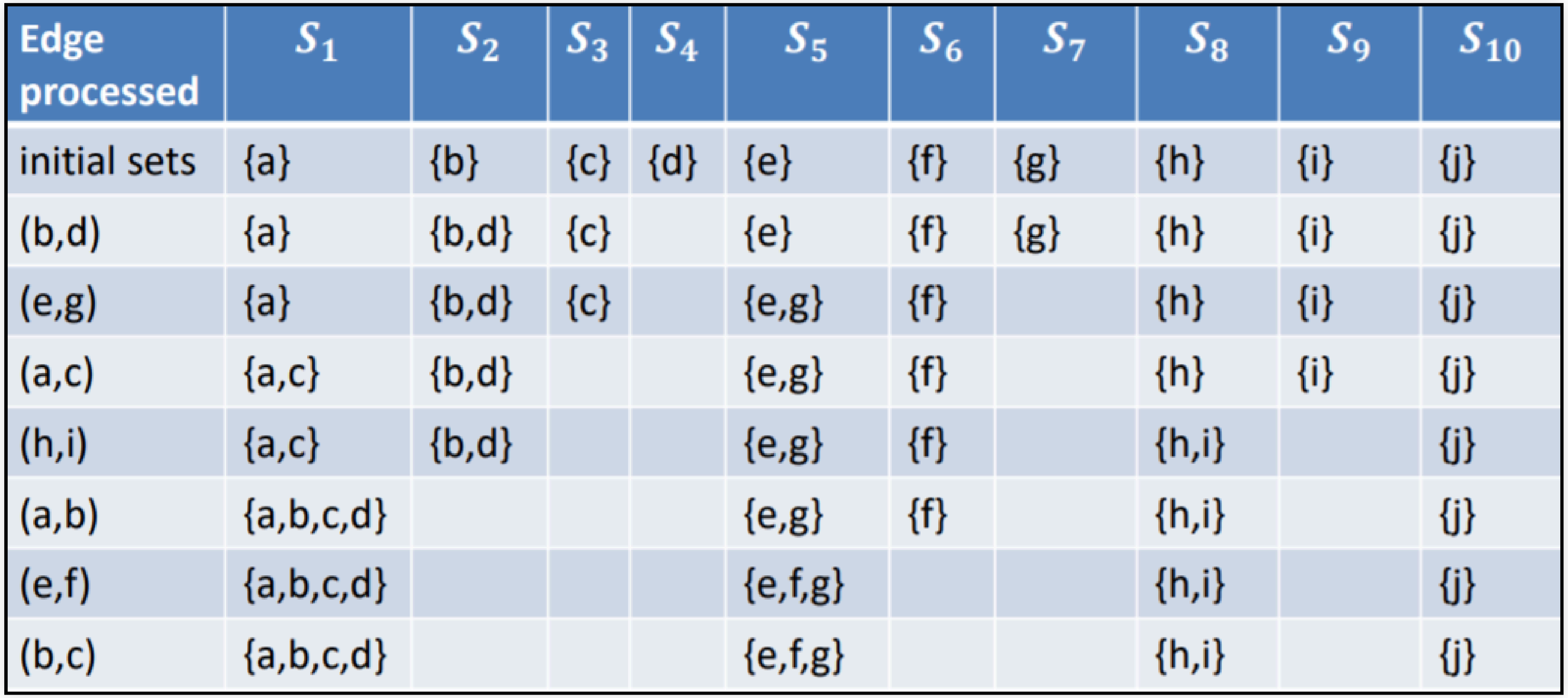
- Once Connected-Components has been run as a preprocessing step, the procedure Same-Component given below answers queries about whether two vertices are in the same connected component.

```
Same-Component(u,v)
  if Find-Set(u) = Find-Set(v)
    then return TRUE
    else return FALSE
```

## Example Graph



## The Collection of Disjoint Sets After Each Edge is Processed



| Edge processed | $S_1$ | $S_2$ | $S_3$ | $S_4$ | $S_5$ | $S_6$ | $S_7$ | $S_8$ | $S_9$ | $S_{10}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| initial sets | {a} | {b} | {c} | {d} | {e} | {f} | {g} | {h} | {i} | {j} |
| (b,d) | {a} | {b,d} | {c} |  | {e} | {f} | {g} | {h} | {i} | {j} |
| (e,g) | {a} | {b,d} | {c} |  | {e,g} | {f} |  | {h} | {i} | {j} |
| (a,c) | {a,c} | {b,d} |  |  | {e,g} | {f} |  | {h} | {i} | {j} |
| (h,i) | {a,c} | {b,d} |  |  | {e,g} | {f} |  | {h,i} |  | {j} |
| (a,b) | {a,b,c,d} |  |  |  | {e,g} | {f} |  | {h,i} |  | {j} |
| (e,f) | {a,b,c,d} |  |  |  | {e,f,g} |  |  | {h,i} |  | {j} |
| (b,c) | {a,b,c,d} |  |  |  | {e,f,g} |  |  | {h,i} |  | {j} |

## Minimum Spanning Trees

- Another application of the disjoint set operations that we will see is Kruskal’s algorithm for computing the **minimum spanning tree** of a graph.

# Maintaining Equivalence Relations

Another application of disjoint-set data structures is to maintain **equivalence relations.**

**Definition**. An **equivalence relation** on a set S is relation $\equiv$ with the following properties:

- **Reflexivity**: for all $a \in S$, we have $a \equiv a$.
- **Symmetry**: for all $a, b \in S$, if $a \equiv b$ then $b \equiv a$ .
- **Transitivity**: for all $a, b, c \in S$, if $a \equiv b$ and $b \equiv c$ then $a \equiv c$ .

## Examples of Equivalence Relations

- **Equality**
- **Equivalent type definitions in programming languages**. For example, consider the following type definitions in C:

```c
struct A {
    int a;
    int b;
};
typedef A B;
typedef A C;
typedef A D;
```

The types A, B, C and D are equivalent in the sense that variables of one type can be assigned to variables of the other types without requiring any casting.

## Equivalent Classes

- If a set S has an equivalence relation defined on it, then the set can be partitioned into disjoint subsets $S_1, S_2, \ldots, S_n$ called **equivalence classes** whose union is S
- Each subset $S_i$ consists of equivalent members of S. That is, $a \equiv b$ for all $a$ and $b$ in $S_i$ , and $a \not\equiv b$ if $a$ and $b$ are in different subsets.

## Example

- Let us consider the set S = {1,2, …, 7}.
- The equivalence relation on is defined by the following:

$$\mathbf{1 \equiv 2, 5 \equiv 6, 3 \equiv 4, 1 \equiv 4, 1 \equiv 3}$$

- Note that the relation $\mathbf{1 \equiv 3}$ follows from the others given the definition of an equivalence relation.

## The Equivalence Problem

- The **equivalence problem** can be formulated as follows.
- We are given a set S and a sequence of statements of the form $a \equiv b$.
- We are to process the statements in order in such a way that, at any time, we are able to determine in which equivalence class a given element of S belongs.

## The Equivalence Problem (cont'd)

- We can solve the equivalence problem by starting with each element in a named set.
- When we process a statement $a \equiv b$, we call $Find - Set(a)$ and $Find - Set(b)$.
- If these two calls return different sets then we call $Union$ to unite these sets. If they return the same set then this statement follows from the other statements and can be discarded.

## Example (cont'd)

- We start with each element of S in a set:

  {1} {2} {3} {4} {5} {6} {7}

- As the given equivalence relations are processed, these sets are modified as follows:

  $\mathbf{1} \equiv \mathbf{2}$ {1,2} {3} {4} {5} {6} {7}

  $\mathbf{5} \equiv \mathbf{6}$ {1,2} {3} {4} {5,6} {7}

  $\mathbf{3} \equiv \mathbf{4}$ {1,2} {3,4} {5,6} {7}

  $\mathbf{1} \equiv \mathbf{4}$ {1,2,3,4} {5,6} {7}

  $\mathbf{1} \equiv \mathbf{3}$ follows from the other statements and is discarded

## Example (cont'd)

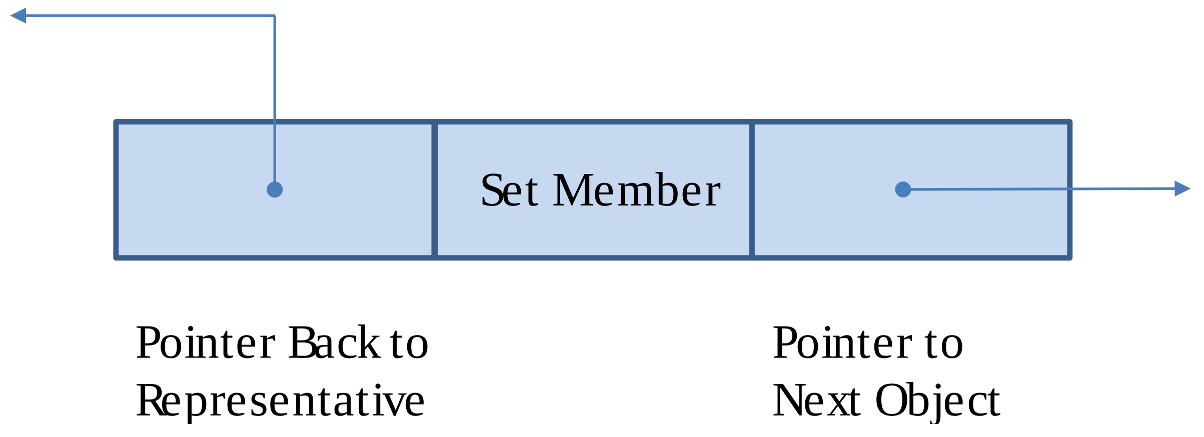
- Therefore, the equivalent classes of S are the subsets

  {1,2,3,4}, {5,6} and {7}.

## Linked-List Representation of Disjoint Sets

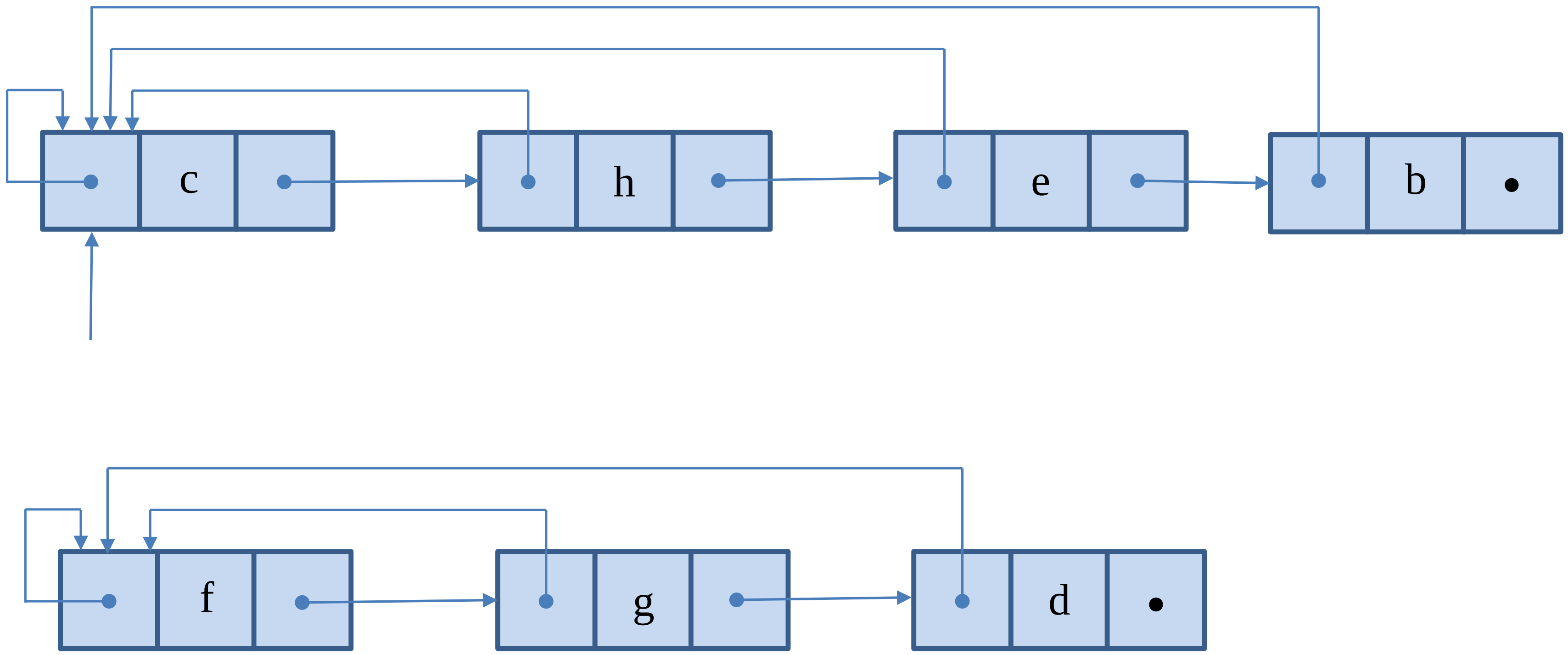
- A simple way to implement a disjoint-set data structure is to represent each set by a **linked list.**
- The **first object** in each linked list serves as its set's **representative**. The remaining objects can appear in the list in any order.
- Each object in the linked list contains a set member, a pointer to the object containing the next set member, and a pointer back to the representative.

## The Structure of Each List Object



## Example: the Sets {c, h, e, b} and {f, g, d}

The **representatives** of the two sets are c and f.



## Implementation of Make-Set and Find-Set

- With the linked-list representation, both Make-Set and Find-Set are easy.
- To carry out $Make - Set(x)$, we create a new linked list which has one object with set element x.
- To carry out, $Find - Set(x)$, we just return the pointer from x back to the representative.

## Implementation of Union

- To perform $Union(x)$, we can append x’s list onto the end of y’s list.
- The representative of the new set is the element that was originally the representative of the set containing y
- We should also update the pointer to the representative for each object originally in x’s list.

## Amortized Analysis

- In an **amortized analysis (επιμερισμένη ανάλυση)**, the time required to perform a sequence of data structure operations is averaged over all operations performed.
- Amortized analysis can be used to show that the average cost of an operation is small, if one averages over a sequence of operations, even though a single operation might be expensive.
- Amortized analysis differs from the average-case analysis in that probability is not involved; an amortized analysis guarantees the **average performance of each operation in the worst case**.

## Techniques for Amortized Analysis

- The **aggregate method (η μέθοδος της συνάθροισης).** With this method, we show that for all $m$, a sequence of $m$ operations takes time $T(m)$ in total, in the worst case. Therefore, in the worst case, the average cost, or **amortized cost**, per operation is $T(m)/m$
- The accounting method.
- The potential method.
- We will only use the aggregate method in this lecture. For the other methods, see any advanced algorithms book e.g., the one cited in the readings.

## Complexity Parameters for the Disjoint-Set Data Structures

We will analyze the running time of our data structures in terms of two parameters:

- n, the number of Make-Set operations, and
- m, the total number of Make-Set, Union and Find-Set operations.

Since the sets are disjoint, each union operation reduces the number of sets by one. Therefore, after $n - 1$ Union operations, only one set remains. The number of Union operations is thus at most $n - 1$.

Since the Make-Set operations are included in the total number of operations, we have $m \geq n$.

## Complexity of Operations for the Linked List Representation

- $Make - Set$ and $Find - Set$ take $O(1)$ time.
- $Union(x, y)$ takes time $O(|x| + |y|)$ where $|x|$ and $|y|$ denote the cardinalities of the sets that contain $x$ and $y$. We need $O(|y|)$ time to reach the last object in y's list to make it point to the first object in x's list. We also need $O(|x|)$ time to update all pointers to the representative in x's list.
- If we keep a pointer to the last object in the list in each representative then we do not need to scan y's list, and we only need $O(|x|)$ time to update all pointers to the representative in 's list.
- In both cases, the complexity of $Union$ is $O(n)$since the cardinality of each set can be at most $n$.

## Complexity (cont'd)

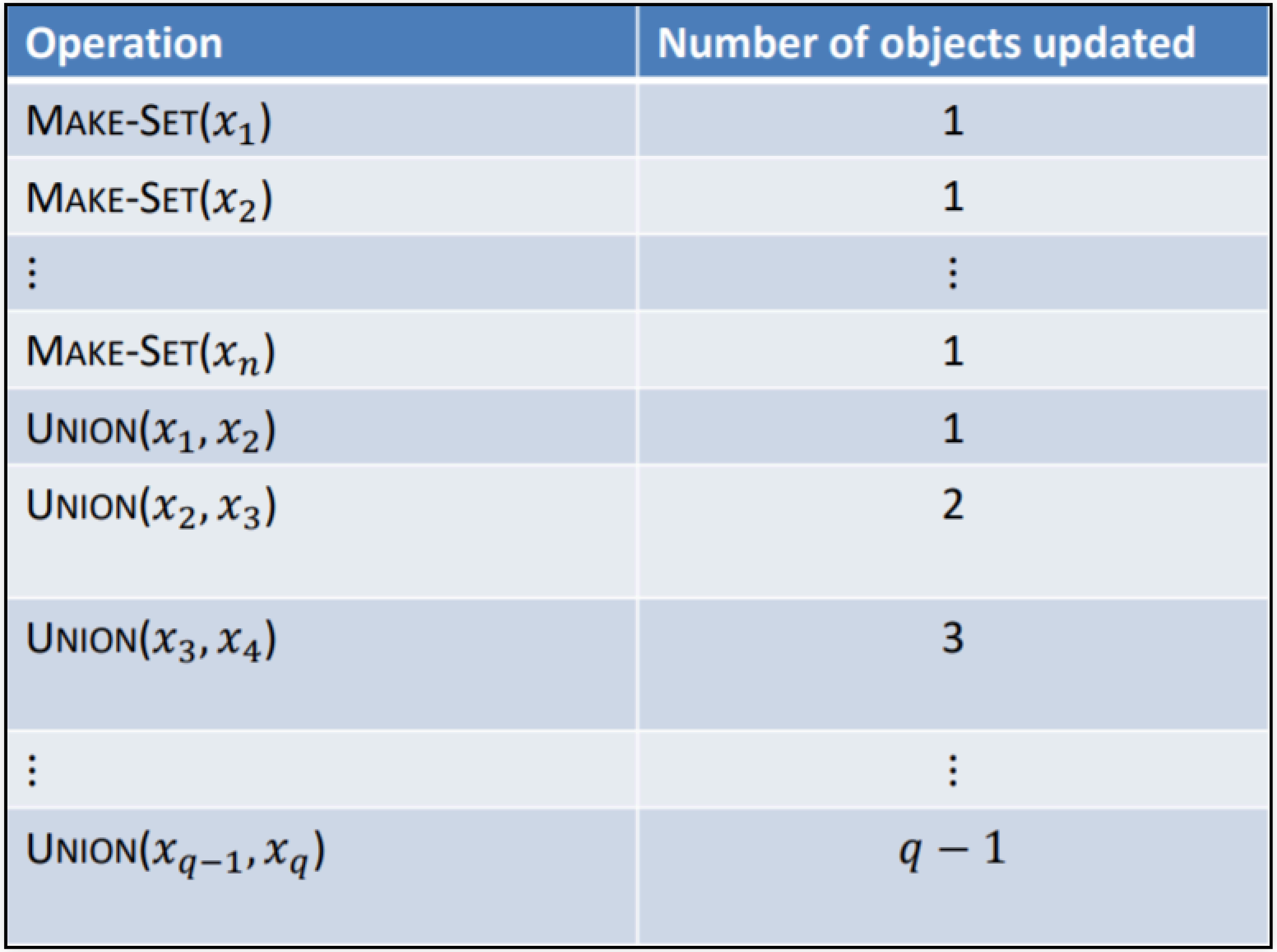
- We can prove that there is a sequence of Make-Set and Union operations that take $O(m^2)$ time. Therefore, the amortized time of an operation is $O(m)$.

Proof?

## Proof

- Let $n = ceil(m/2) + 1$ and $q = m - n = floor(m/2) - 1$
- Suppose that we have n objects $x_1, x_2, \ldots, x_n$
- We then execute the sequence of $m = n + q$ operations shown on the next slide.

## Operations



| Operation | Number of objects updated |
|---|---|
| MAKE-SET($x_1$) | 1 |
| MAKE-SET($x_2$) | 1 |
| $\vdots$ | $\vdots$ |
| MAKE-SET($x_n$) | 1 |
| UNION($x_1, x_2$) | 1 |
| UNION($x_2, x_3$) | 2 |
| UNION($x_3, x_4$) | 3 |
| $\vdots$ | $\vdots$ |
| UNION($x_{q-1}, x_q$) | $q - 1$ |

## Proof (cont'd)

- We spend $O(n)$ time performing the $n\ Make - Set$ operations.
- Because the i-th Union operation updates i objects, the total number of objects updated are $\sum_{i=1}^{q-1} i = \frac{q(q-1)}{2} = O(q^2)$ .
- The total time spent therefore is $O(n + q^2)$ which is $O(m)$ since $n = O(m)$ and $q = O(m)$.

## The Weighted Union Heuristic

- The above implementation of the Union operation requires an average $O(m)$ of time per operation because we may be appending a longer list onto a shorter list, and we must update the pointer to the representative of each member of the longer list.

- If **each representative also includes the length of the list** then we can always append the smaller list onto the longer, with ties broken arbitrarily. This is called the **weighted union heuristic.**

## Theorem

- Using the linked list representation of disjoint sets and the weighted union heuristic, a sequence of $mMake - Set, Union$ and $Find - Set$ operations,n of which are Make-Set operations, takes $O(m + n\log_2(n))$ time.
- Proof?

## Proof

- We start by computing, for each object in a set of size n, an upper bound on the number of times the object's pointer back to the representative has been updated.
- Consider a fixed object x. We know that each time x's representative pointer was updated, x must have started in the smaller set and ended up in a set (the union) at least twice the size of its own set.
- For example, the first time x's representative pointer was updated, the resulting set must have had at least 2 members. Similarly, the next time x's representative pointer was updated, the resulting set must have had at least 4 members.
- Continuing on, we observe that for any $k \leq n$, after x's representative pointer has been updated $\log 2(k)$ times, the resulting set must have at least k members.

## Proof (cont'd)

- Since the largest set has at most n members, each object's representative pointer has been updated at most $\log_2(n)$ times over all Union operations. The total time used in updating n objects is thus $O(n \log_2(n))$.
- The time for the entire sequence of operations follows easily.
- Each Make-Set and Find-Set operation takes $O(1)$ time, and there are $O(m)$ of them.
- The total time for the entire sequence is thus $O(m + n \log_2(n))$.

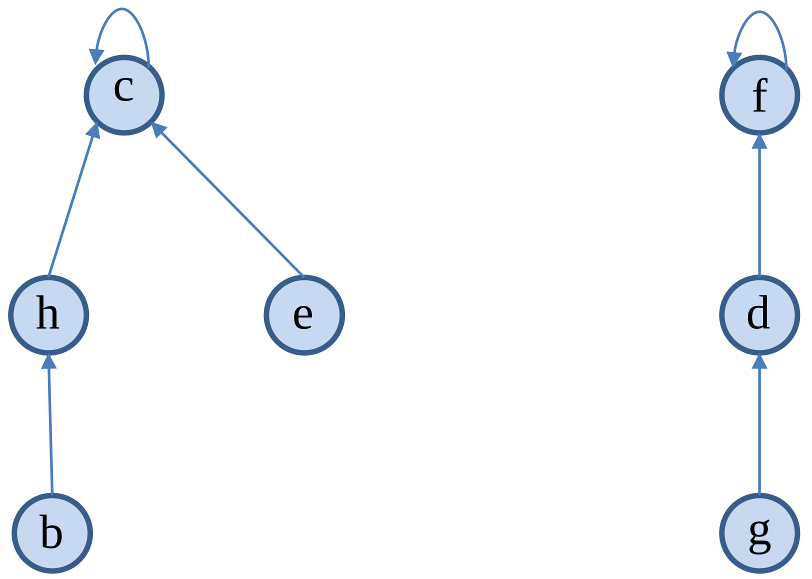
## Complexity (cont'd)

- The bound we have just shown can be seen to be $O(m \log_2(m))$ therefore the amortized time for each of the operations is $O(\log_2(m))$
- There is a faster implementation of disjoint sets which improves this amortized complexity.
- We will present this method now.

## Disjoint-Set Forests

- In the faster implementation of disjoint sets, we represent sets by **rooted trees**.
- Each node of a tree represents one set member and each tree represents a set.
- In a tree, each set member points only to its parent. **The root of each tree contains the representative** of the set and is its own parent.
- For many sets, we have a **disjoint-set forest.**

## Example: the Sets {b, c, e, h} and {d, f, g}

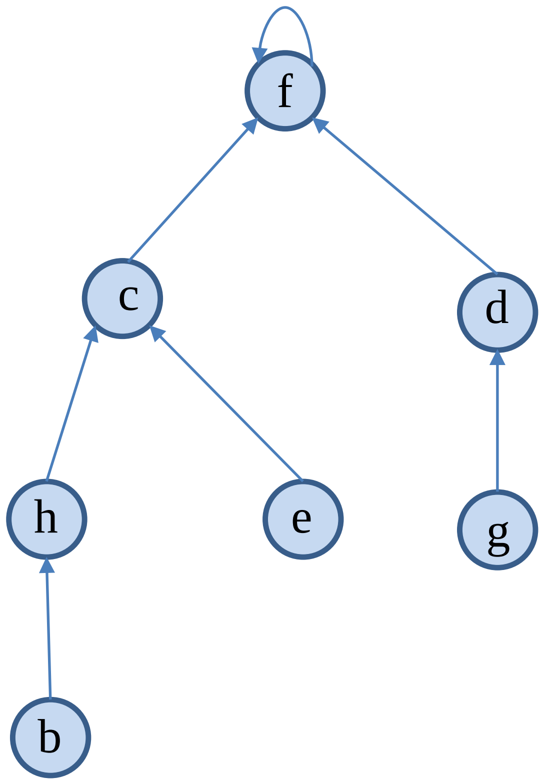


The **representatives** of the two sets are c and f.

## Implementing Make-Set, Find-Set and Union

- A Make-Set operation simply creates a tree with just one node.
- A Find-Set operation can be implemented by chasing parent pointers until we find the root of the tree. The nodes visited on this path towards the root constitute the **find-path**.
- A Union operation can be implemented by making the root of one tree to point to the root of the other.

## Example: the Union of Sets {b, c, e, h} and {d, f, g}



## Complexity

- With the previous data structure, we do not improve on the linked-list implementation.
- A sequence of $n - 1$ Union operations may create a tree that is just a **linear chain** of n nodes. Then, a Find-Set operation can take O(n) time. Similarly, for a Union operation.
- By using the following two heuristics, we can achieve a running time that is almost linear in the number of operations m.

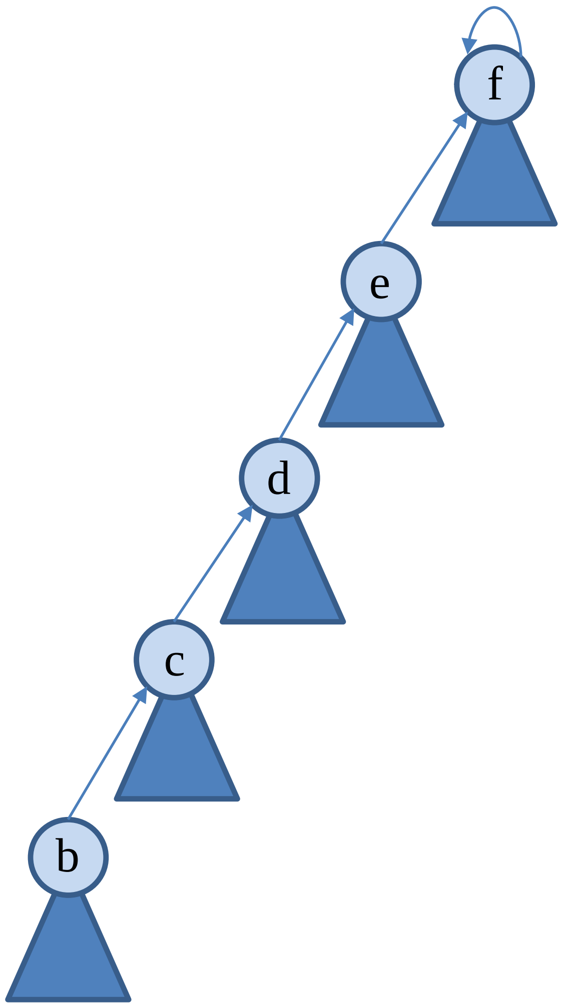
## The Union by Rank Heuristic

- The first heuristic, **union by rank**, is similar to the weighted union heuristic we used with the linked list representation.
- The idea is to make the root of the tree with fewer nodes to point to the root of the tree with more nodes.
- We will not explicitly keep track of the size of the subtree rooted at each node. Instead, for each node, we maintain a **rank** that approximates the logarithm of the size of the subtree rooted at the node and is also an upper bound on the height of the node (i.e., the number of edges in the longest path between x and a descendant leaf).
- In union by rank, the root with the smaller rank is made to point to the root with the larger rank during a Union operation.

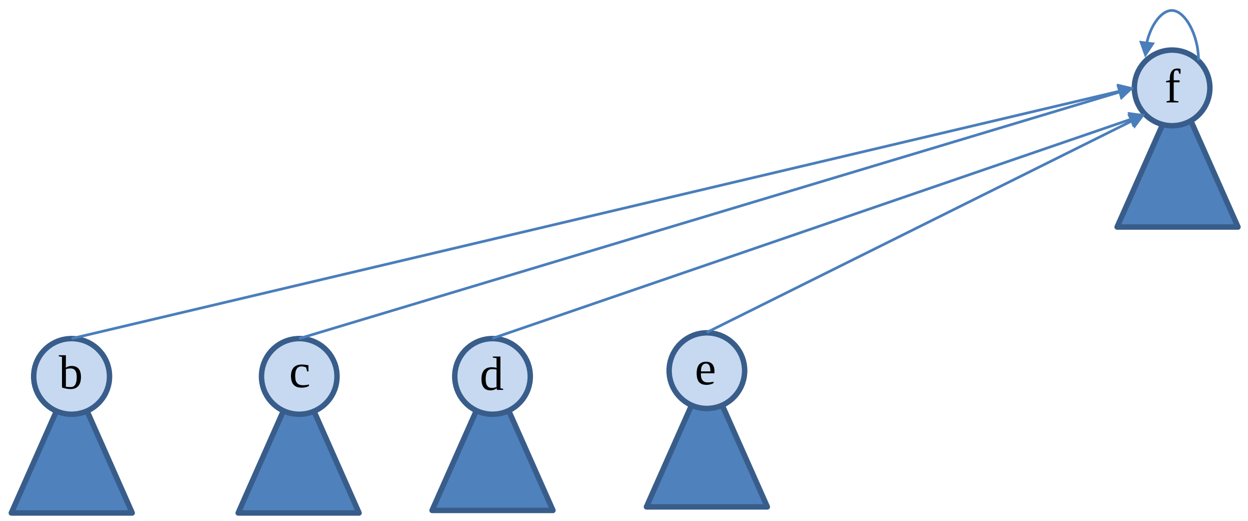
## The Path Compression Heuristic

- The second heuristic, **path compression**, is also simple and very effective.
- This heuristic is used during Find-Set operations to **make each node on the find path point directly to the root.**
- In this way, trees with **small height** are constructed.
- Path compression does not change any ranks.

## The Path Compression Heuristic Graphically



## The Path Compression Heuristic Graphically (cont'd)



## Implementing Disjoint-Set Forests

- With each node x, we maintain the integer value rank ✓ , which is an upper bound on the height of x.
- When a singleton set is created by Make-Set, the initial rank of the single node in the corresponding tree is 0.
- Each Find-Set operation leaves ranks unchanged.
- When applying Union to two trees, we make the root of higher rank the parent of the root of lower rank. In this case ranks remain the same. In case of a tie, we arbitrarily choose one of the roots as the parent and increment its rank.

## Pseudocode

We designate the **parent** of node x by p[x].

```
Make-Set(x)
    p[x]← x
    rank[x] ← 0

Union(x,y)
    Link(Find-Set(x), Find-Set(y))
```

## Pseudocode (cont’d)

```
Link(x)
    if rank[x] > rank[y]
        then p[y] ← x
        else p[x] ← y
            if rank[x] = rank[y]
            then rank[y] ← rank[y]+1
Find-Set(x)
    if x != p[x]
        then p[x] ← Find-Set(p[x])
    return p[x]
```

## The Find-Set Procedure

- Notice that the Find-Set procedure is a **two-pass method**: it makes one pass up the find path to find the root, and it makes a second pass back down the find path to update each node so it points directly to the root.
- The second pass is made as the recursive calls return.

## Complexity

- Let us consider a sequence of Make-Set, Union and Find-Set operations, n of which are Make-Set operations.
- When we use both union by rank and path compression, the worst case running time for the sequence of operations can be proven to be **O(m a(m,n))** where a(m,n) is the very slowly growing **inverse of Ackermann's function**.
- Ackermann's function is an exponential, very rapidly growing function. Its inverse, a, grows **slower than the logarithmic function**.
- In any conceivable application of a disjoint-union data structure, we have $a(m, n) \leq 4$
- Thus we can view the running time as **linear in m** in all practical situations.
- Therefore, the amortized complexity of each operation is **O(1)**

## Implementation in C

- Let us assume that the sets will have positive integers in the range 0 to N-1 as their members.
- The simplest way to implement in C the disjoint sets data structure is to use an array id[N] of integers that take values in the range 0 to N-1. This array will be used to keep track of the representative of each set but also the members of each set.
- Initially, we set id[i]=i, for each i between 0 and N-1. This is equivalent to N Make-Set operations that create the initial versions of the sets.
- To implement the Union operation for the sets that contain integers p and q, we scan the array id and change all the array elements that have the value p to have the value q.
- The implementation of the Find-Set(p) simply returns the value of id[p].

## Implementation in C (cont’d)

- The program on the next slide initializes the array id, and then reads pairs of integers (p,q) and performs the operation Union(p,q) if p and q are not in the same set yet.

- The program is an implementation of the equivalence problem defined earlier. Similar programs can be written for the other applications of disjoint sets presented.

## Implementation in C (cont'd)

```c
#include <stdio.h>
#define N 10000
int main(void) {
    int i, p, q, t, id[N];

    for (i = 0; i < N; i++)
        id[i] = i;

    while (scanf("%d %d", &p, &q) == 2) {
        if (id[p] == id[q]) continue;
        for (t = id[p], i = 0; i < N; i++)
            if (id[i] == t) id[i] = id[q];
        printf("%d %d\n", p, q);
    }
}
```

## Implementation in C (cont’d)

- The extension of this implementation to the case where sets are represented by linked lists is left as an exercise.

## Implementation in C (cont’d)

- The disjoint-forests data structure can easily be implemented by changing the meaning of the elements of array id. Now each id[i] represents an element of a set and points to another element of that set. The root element points to itself.
- The program on the next slide illustrates this functionality. Note that after we have found the roots of the two sets, the Union operation is simply implemented by the assignment statement id[i]=j.
- The implementation of the Find-Set operation is similar.

## Implementation in C (cont’d)

```c
#include <stdio.h>
#define N 10000
int main(void) {
    int i, j, p, q, t, id[N];
    for (i = 0; i < N; i++)
        id[i] = i;
    while (scanf("%d %d", &p, &q) == 2) {
        for (i = p; i != id[i]; i = id[i]) ;
        for (j = q; j != id[j]; j = id[j]) ;
        if (i == j) continue;
        id[i] = j;
        printf("%d %d\n", p, q);
    }
}
```

## Implementation in C (cont’d)

- We can implement a **weighted version** of the Union operation by keeping track of the size of the two trees and making the root of the smaller tree point to the root of the larger.
- The code on the next slide implements this functionality by making use of an array sz[N] (for size).

## Implementation in C (cont'd)

```c
#include <stdio.h>
#define N 10000
int main(void) {
    int i, j, p, q, id[N], sz[N];
    for (i = 0; i < N; i++) {
        id[i] = i; sz[i] = 1;
    }
    while (scanf("%d %d", &p, &q) == 2) {
        for (i = p; i != id[i]; i = id[i]) ;
        for (j = q; j != id[j]; j = id[j]) ;
        if (i == j) continue;
        if (sz[i] < sz[j]) {
            id[i] = j;
            sz[j] += sz[i];
        }
        else {
            id[j] = i;
            sz[i] += sz[j];
        }
        printf("%d %d\n", p, q);
    }
}
```

## Implementation in C (cont’d)

- In a similar way, we can implement the union by rank heuristic.
- This heuristic together with the path compression heuristic are left as exercises.

## Readings

T.H. Cormen, C. E. Leiserson and R. L. Rivest. *Introduction to Algorithms*. MIT Press.

- Chapter 22

Robert Sedgewick. *Αλγόριθμοι σε C*. 3η Αμερικανική Έκδοση. Εκδόσεις Κλειδάριθμος.

- Κεφάλαιο 1