Data Structures for Disjoint Sets

Κ08 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)

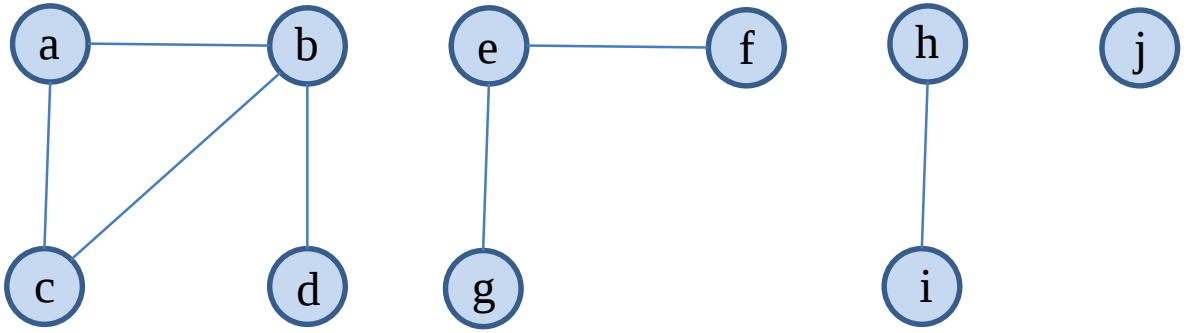
The disjoint-set data structure supports the following operations:

- $Make - Set(x)$: 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.

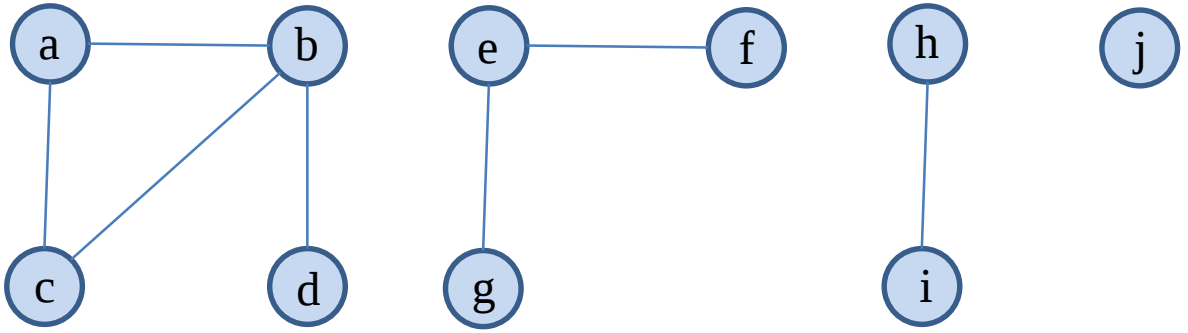
```
Connected-Components(G)
  for each vertex v in  V[G]
    do Make-Set(v)
  for each edge(u,v) in E(G)
      do if not(Find-Set(u) = Find-Set(v))
        then Union(u,v)
```

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 $a \neq b$ 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:

  $$1 \equiv 2, 5 \equiv 6, 3 \equiv 4, 1 \equiv 4, 1 \equiv 3$$

- Note that the relation $1 \equiv 3$ 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:

  $1 \equiv 2$ {1,2} {3} {4} {5} {6} {7}

  $5 \equiv 6$ {1,2} {3} {4} {5,6} {7}

  $3 \equiv 4$ {1,2} {3,4} {5,6} {7}

  $1 \equiv 4$ {1,2,3,4} {5,6} {7}

  $1 \equiv 3$ follows from the other statements and is discarded

Example (cont'd)

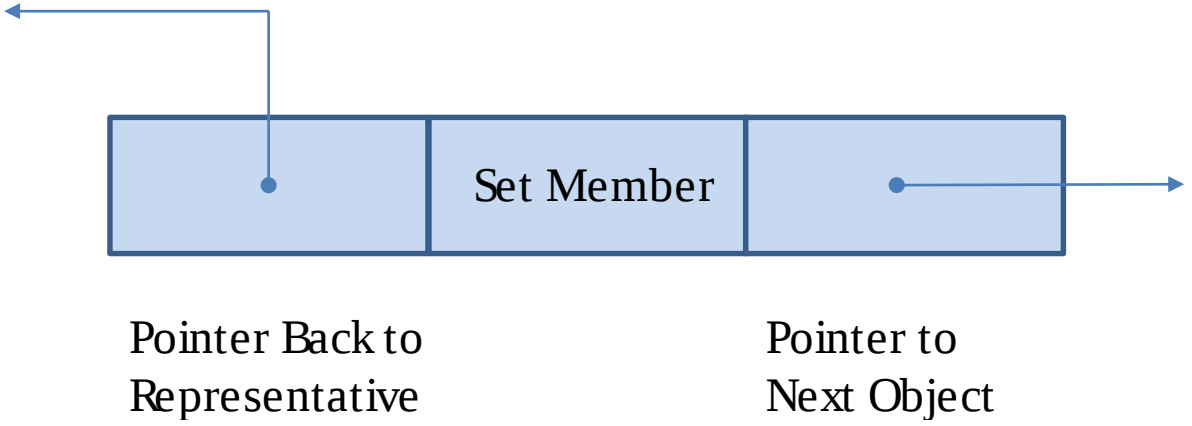
- Therefore, the equivalent classes of S are the subsets

  {1,2,3,4}, {5,6} and {7}.

Linked-List Representation of Disjoint Sets

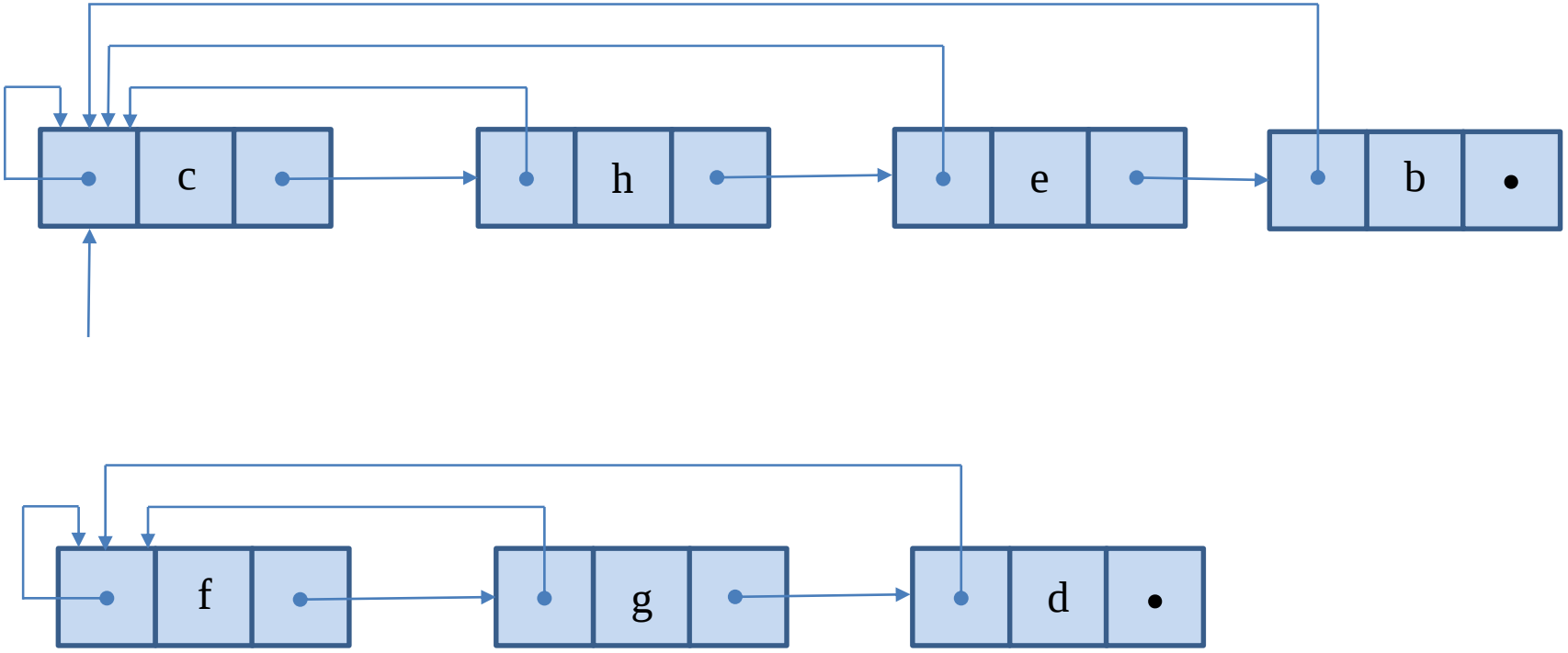
- A simple way to implement a disjoint-set data structure is to represent each set by a **linked list.**
- The **first object** in each linked list serves as its set's **representative**. The remaining objects can appear in the list in any order.
- Each object in the linked list contains a set member, 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
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_1, x_2)	1
UNION(x_2, x_3)	2
UNION(x_3, x_4)	3
⋮	⋮
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 $O(m + n \log_2(n))$.

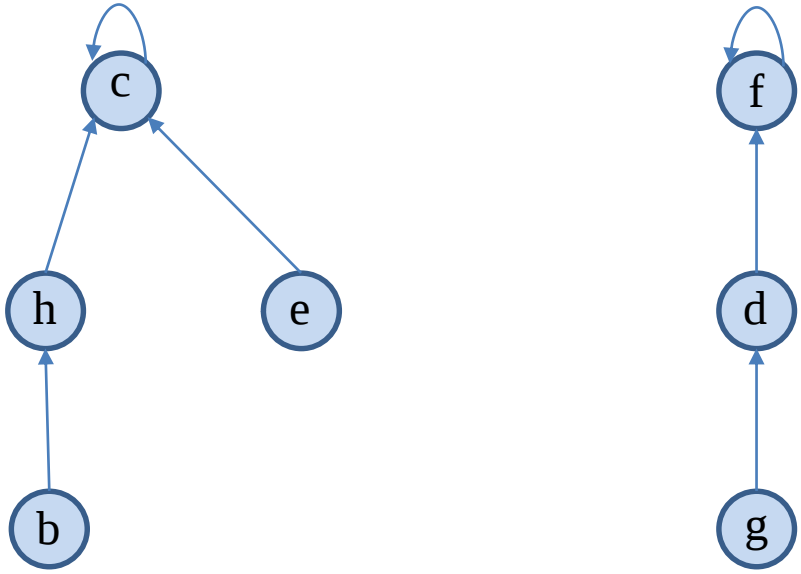
Complexity (cont'd)

- 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}

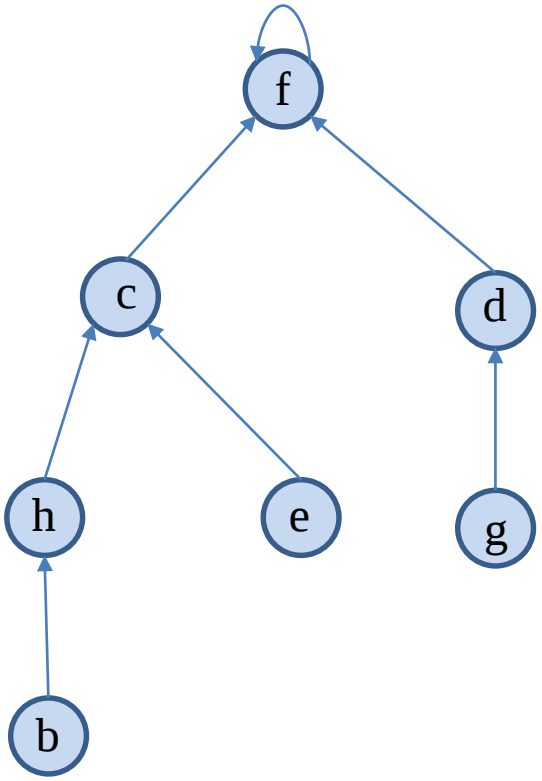


The **representatives** of the two sets are c and f.

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.

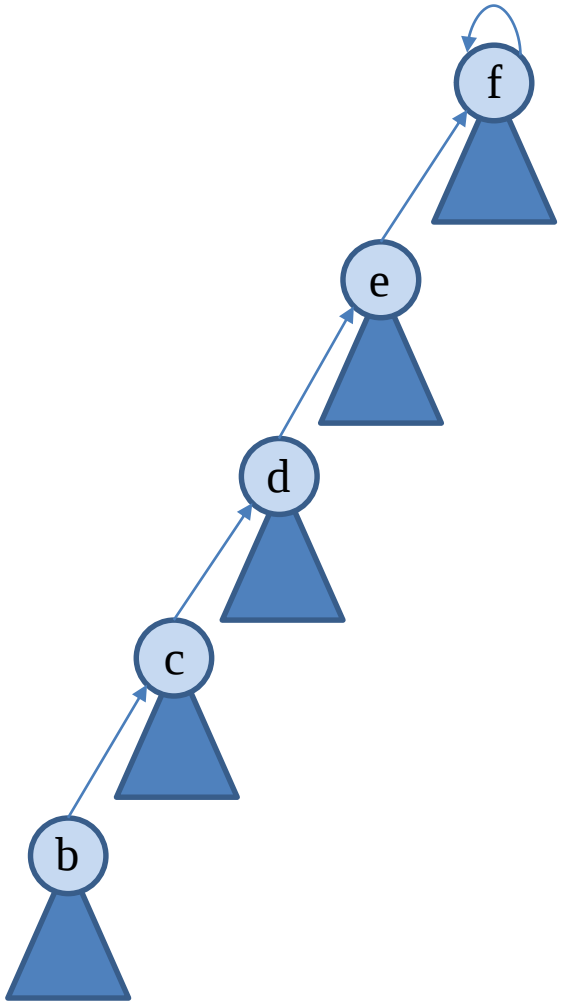
The Union by Rank Heuristic

- 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.

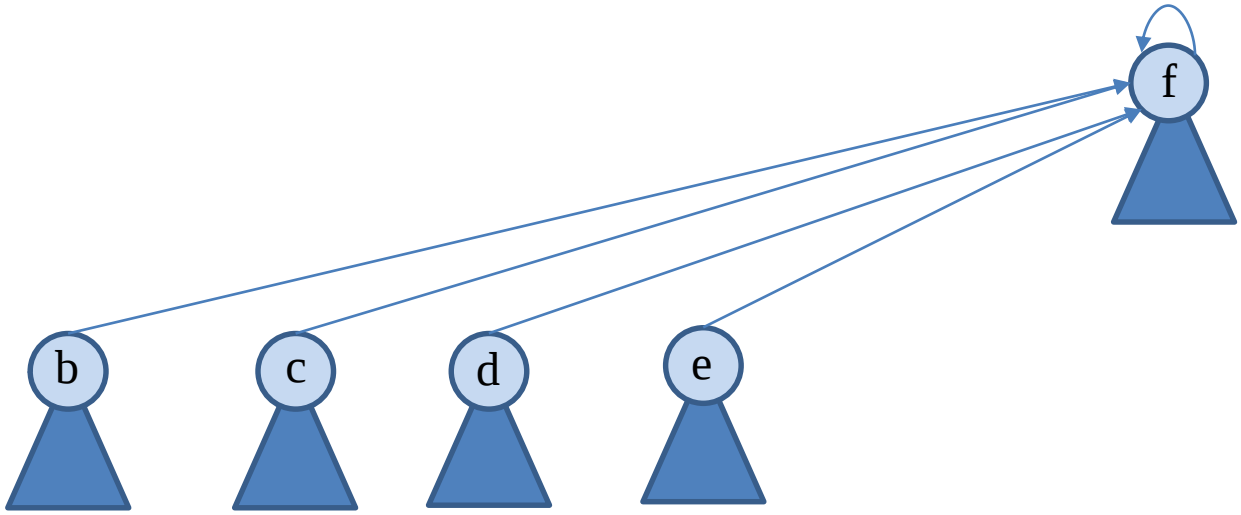
The Path Compression Heuristic

- 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