Data and File Structures Laboratory

Disjoint Sets, Generic Implementations

http://www.isical.ac.in/~dfslab/2018/index.html

Outline

Practice problems

- Implement a stack whose items may be either integer, character or floating point data type. Your implementation should have usual methods such as *push*(), *pop*() and *display*() methods.
  [Hint: try using `Union`]

- Implement the *SkipList* data structure. Your implementation should have usual methods as linked list such as *insert*(), *delete*(), *update*() etc.

Disjoint Sets

Reference: (B4) Introduction to Algorithms, T. H. Cormen, et al.

A disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. Each set is identified by some representative, (usually) a member of the set.

Disjoint Sets

Reference: (B4) Introduction to Algorithms, T. H. Cormen, et al.

A disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets. Each set is identified by some representative, (usually) a member of the set.

In some applications there can be some pre-specified rule for choosing the representative, say the smallest member of the set; otherwise any member of the set can be used the representative.

Disjoint Sets: Operations

$MakeSet(x)$ creates a new set whose only member (and thus its representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

Disjoint Sets: Operations

$MakeSet(x)$ creates a new set whose only member (and thus its representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

$FindSet(x)$ returns a reference of the representative of the (unique) set containing x.

Disjoint Sets: Operations

$MakeSet(x)$ creates a new set whose only member (and thus its representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

$FindSet(x)$ returns a reference of the representative of the (unique) set containing x.

$Union(x, y)$ 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. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of $S_x \cup S_y$. Since we require the sets in the collection to be disjoint, conceptually we destroy sets S_x and S_y, removing them from the collection S. In practice, we often absorb the elements of one of the sets into the other set.

Disjoint Sets: An Array Based Implementation I

Assume there are all total of n elements indexed from 0 through $n-1$. We keep another array *sets* of size n such that $sets[i]$ is index of the representative of the set to which i-th element currently belongs to.

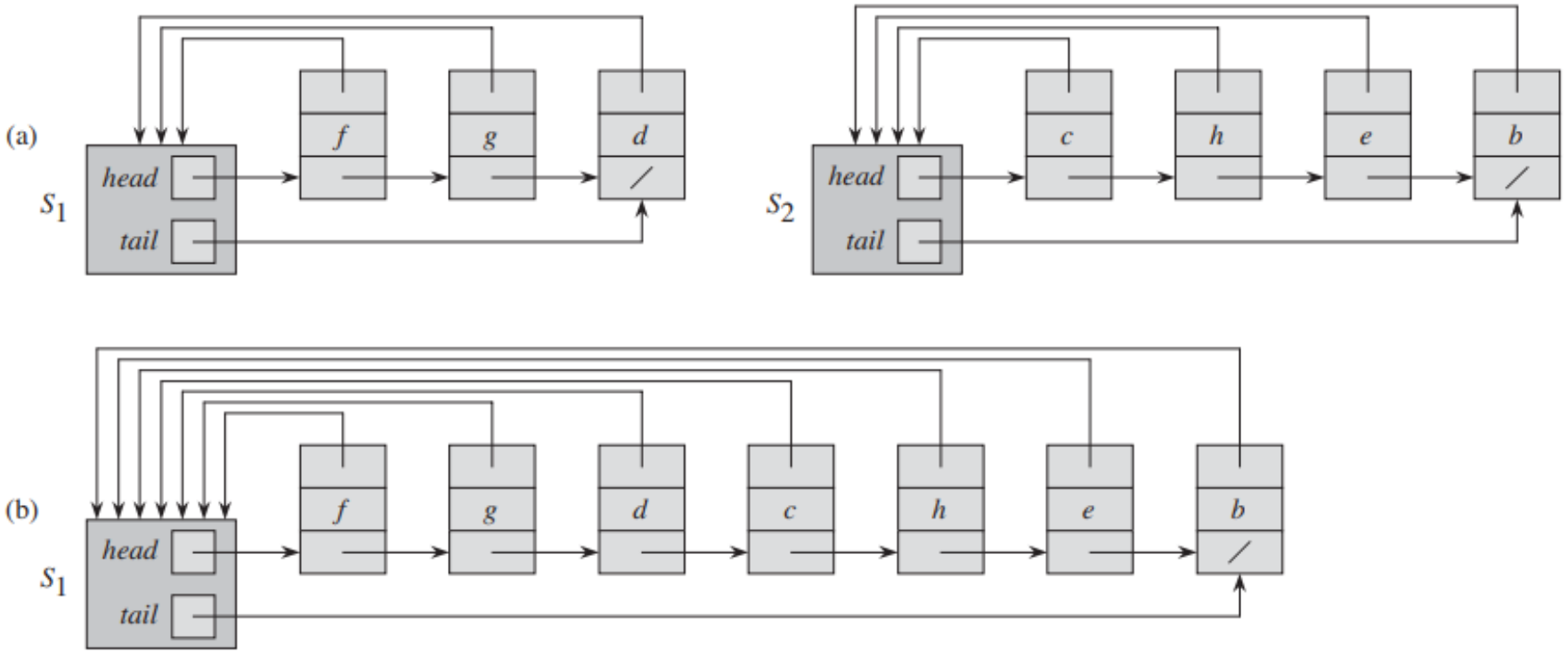
```
void initSet(int *sets, int n) {
    sets = MALLOC(n,int);
    for(i=0; i<n; i++){
        makeSet(sets, i);
    }
}

void makeSet(int *sets, int i) {
    sets[i] = i;
}
```

Disjoint Sets: An Array Based Implementation II

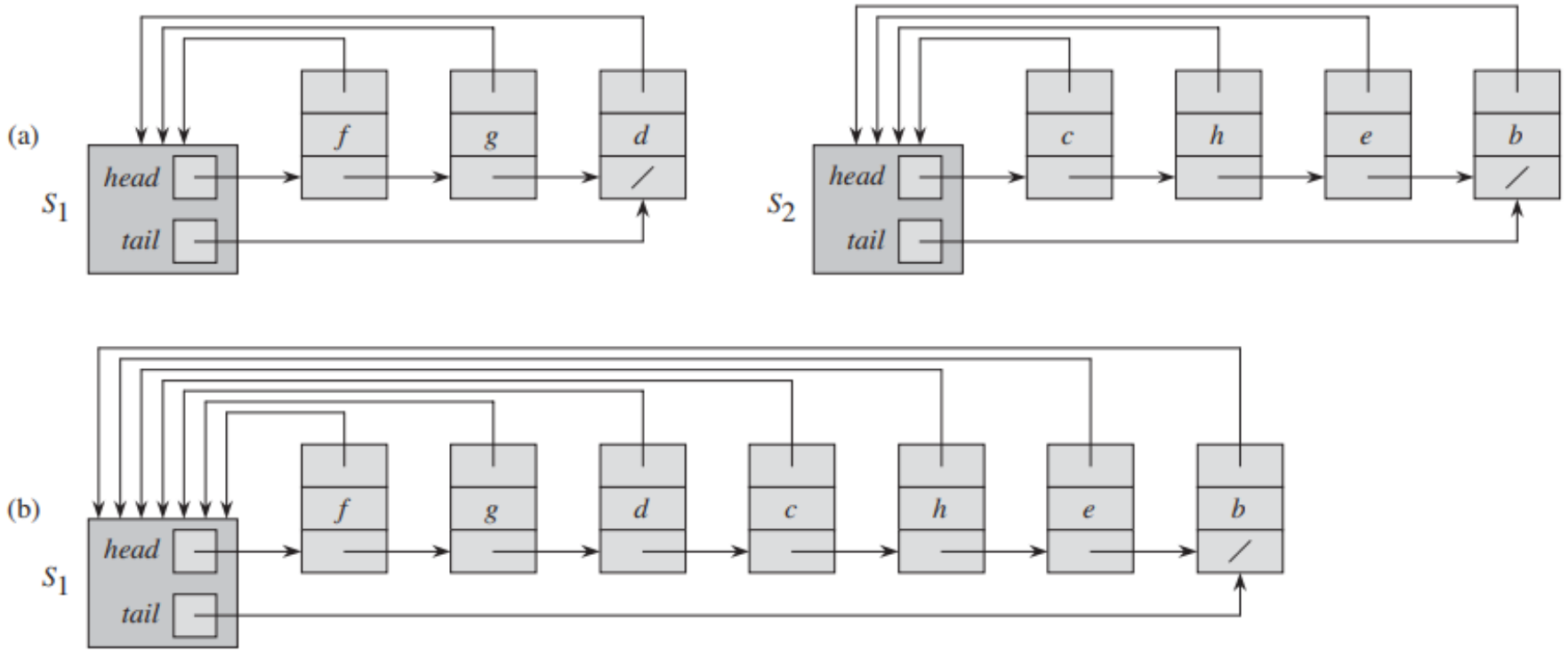
```
int findSet(int *sets, int i) {
    return sets[i];
}

void union(int *sets, int prevRepr, int newRepr, int n) {
    int i;
    for(i=0; i<n; i++){
        if(sets[i] == prevRepr) {
            sets[i] = newRepr;
        }
    }
}
```

What are the execution time of these methods?

Disjoint Sets: Linked List Representation

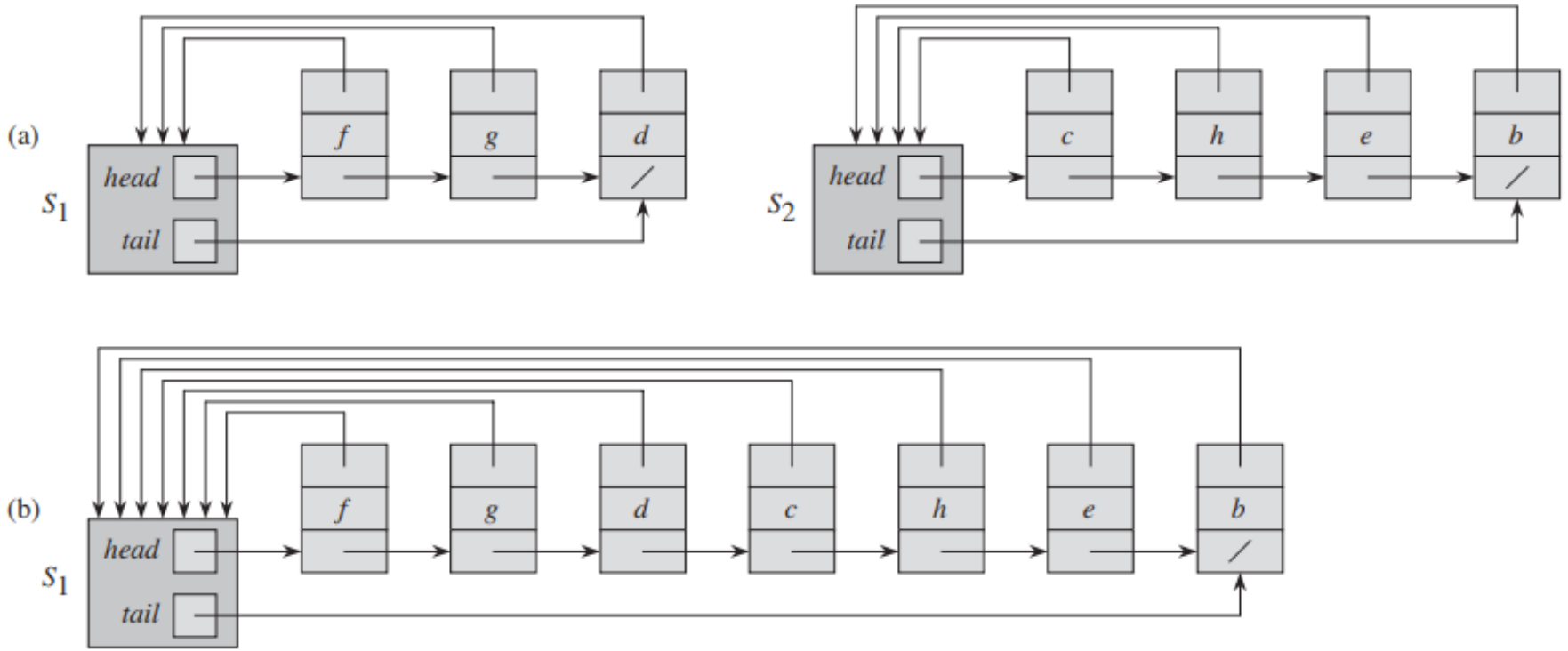


Disjoint Sets: Linked List Representation



What about the execution time?

Disjoint Sets: Linked List Representation

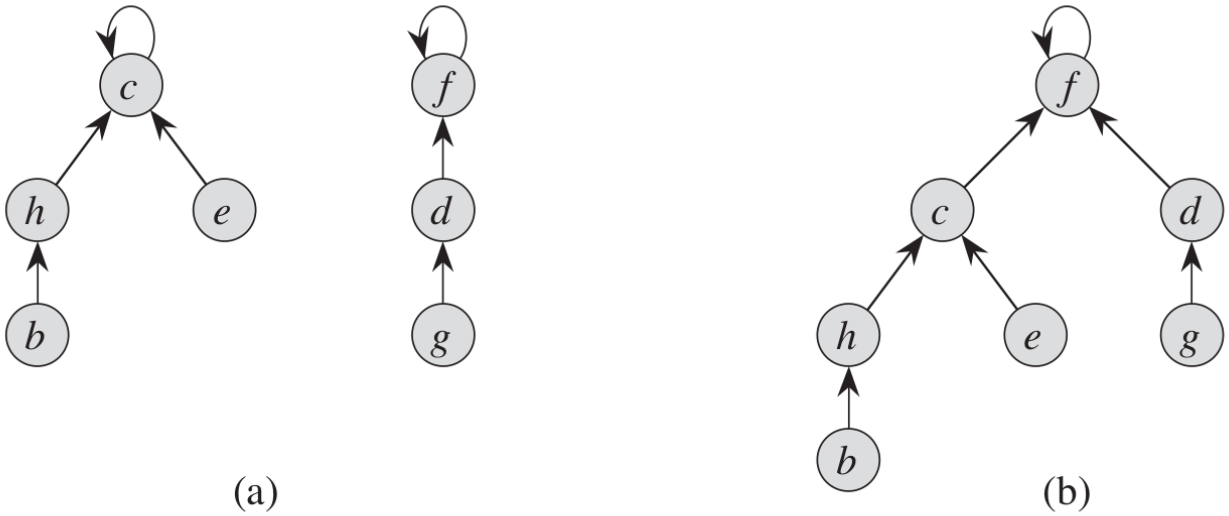


What about the execution time?

Can we improve?

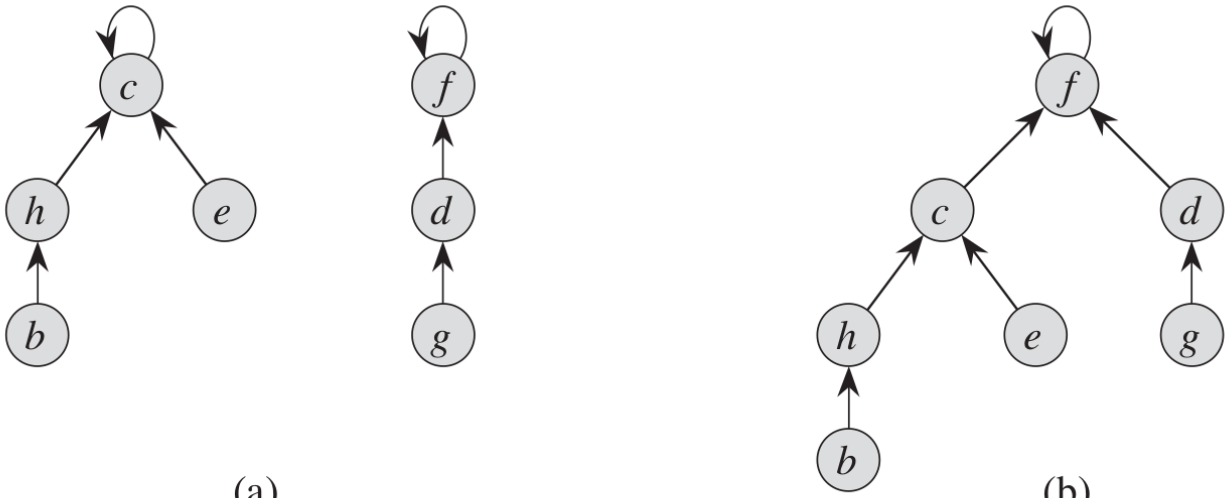
Disjoint Sets: Forest

In a faster implementation of disjoint sets, we represent sets by rooted (inverted) trees, with each node containing one member and each tree representing one set.
In a disjoint-set forest, each member points only to its parent. The root of each tree contains the representative and is its own parent.



Disjoint Sets: Forest

In a faster implementation of disjoint sets, we represent sets by rooted (inverted) trees, with each node containing one member and each tree representing one set.
In a disjoint-set forest, each member points only to its parent. The root of each tree contains the representative and is its own parent.



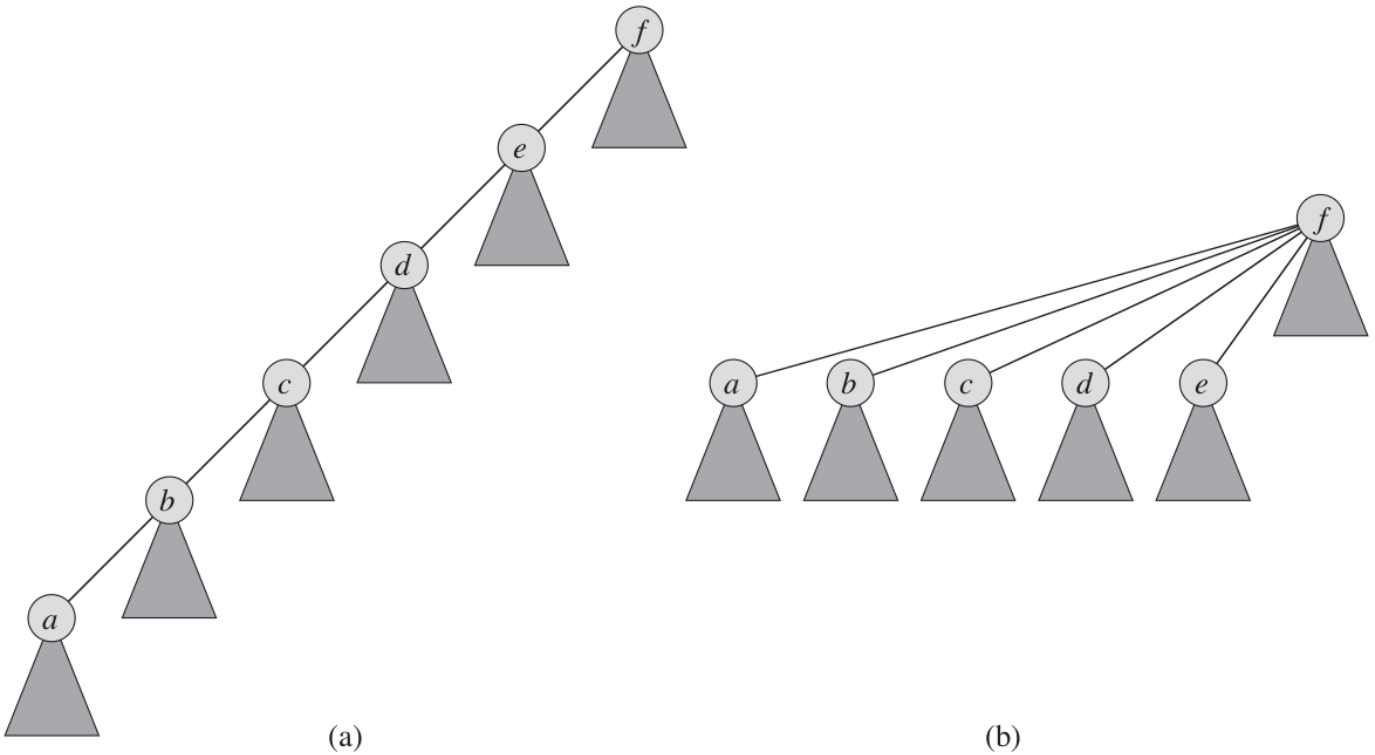
What about the execution time? Can we improve?

Disjoint Sets Forest: Heuristics

- **Union by Rank**: maintain a *rank* value for each node, where rank is an upper bound on the height of that node

Disjoint Sets Forest: Heuristics

- **Union by Rank**: maintain a *rank* value for each node, where rank is an upper bound on the height of that node
- **Path Compression**: shorten the tree height during *FindSet*



Disjoint Sets Forest: Implementation I

Assume there are all total of n elements indexed from 0 through $n-1$. We keep two additional arrays *parent* and *rank* each of size n.

```
void makeSet(int x) {
    parent[x] = x;
    rank[x] = 0;
}

int findSet(int x){
    if(x != parent[x]) {
        parent[x] = findSet(parent[x]);
    }
    return parent[i];
}
```

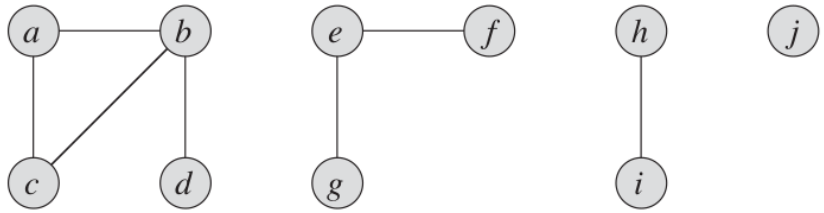
Disjoint Sets Forest: Implementation II

```
void union (int x, int y) {
    link(findSet(x), findSet(y));
}

void link (int x, int y) {
    if(rank[x] > rank[y]) {
        parent[y] = x;
    } else {
        parent[x] = y;
        if(rank[x] == rank[y]) {
            rank[y]++;
        }
    }
}
```

Running time:$\Theta(m\ \alpha(n))$, where $\alpha(n)$ is a very slowly growing function

Disjoint Sets: An Application I

Problem: finding connected components of an undirected graph

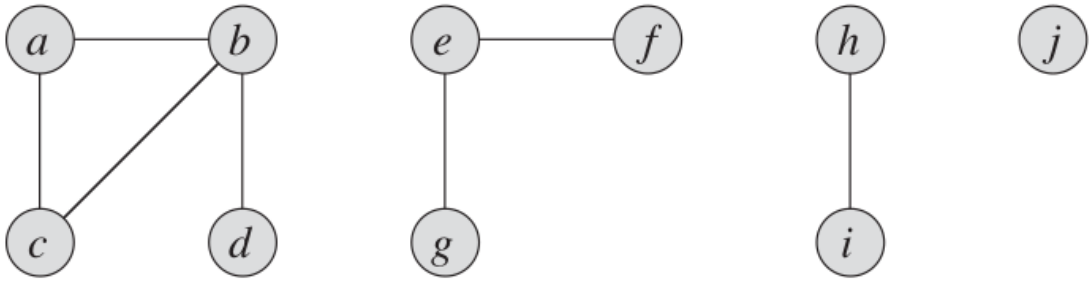



```
1: procedure ConnectedComponents(G = (V, E))
2:     for each vertex v ∈ V do
3:         MakeSet(v)
4:     end for
5:     for each edge (u, v) ∈ E do
6:         if FindSet(u) ≠ FindSet(v) then
7:             Union(u, v)
8:         end if
9:     end for
10: end procedure
```

Disjoint Sets: An Application II

```
1: procedure IsSameComponents(u, v))
2:     if FindSet(u) ≠ FindSet(v) then
3:         return TRUE
4:     else
5:         return FALSE
6:     end if
7: end procedure
```

Disjoint Sets: An Application III



(a)

Edge processed	Collection of disjoint se					
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$
(e,g)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,g\}$	$\{f\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,g\}$	$\{f\}$
(a,b)	$\{a,b,c,d\}$				$\{e,g\}$	$\{f\}$
(e,f)	$\{a,b,c,d\}$				$\{e,f,g\}$	
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$	

(b)

Generic Implementations

Motivation

Writing code (once) in generic way such that the same code can be used (as it is) for any datatypes, specially user defined ones

Generic Implementations

Motivation

Writing code (once) in generic way such that the same code can be used (as it is) for any datatypes, specially user defined ones

An Example

Consider having a sorting routine which can sort a collection of data of any type