21. Data Structures for Disjoint Sets

Some applications involve maintaining **a collection of disjoint sets**. The total number of objects in all sets is given (fixed). Two important operations are

- (i) finding to which set a given object x belongs
- (ii) merging two sets into a single set.

A disjoint set data structure maintains a collection $S = \{S_1, S_2, \ldots, S_k\}$ of disjoint sets of "objects", using the following operations:

- **Make_Set**(x): create a set whose only member is object x, where x is not in any of the sets yet.
- **Find_Set**(x): identify the set containing object x.
- **Union**(x, y): combine two sets containing objects x and y, say S_x and S_y, into a new set $S_x \cup S_y$, assuming that $S_x \neq S_y$ and $S_x \cap S_y = \emptyset$.

21.1 Disjoint Sets (continued)

To represent a set, a common technique is to choose one of objects in the set as "the representative of the set". As long as a set has not been modified, its representative does not change.

The value returned by **Find_Set**(x) is a pointer to the representative of the set that contains x.

We may also wish to implement additional operations such as:

- **insert** a new object to a specific set
- **delete** an object from the set it belongs to
- **split** a set into several smaller subsets according to some criterion
- **find** the size of a specific set

21.1 Connected Components

The operations in the disjoint sets can be used to solve the connected component problem in an undirected graph $G = (V, E)$ which is to find all connected components in G.

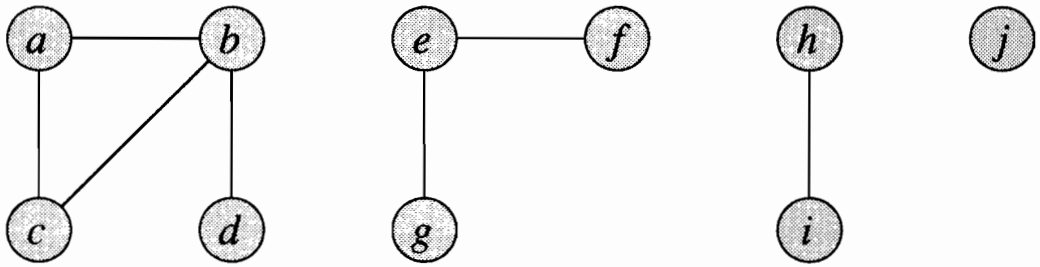
Connected_Components(G)

```
1   for each v ∈ V
2       Make_Set(v)
3   for every edge (u, v) ∈ E
4       if Find_Set(u) ≠ Find_Set(v)
5           then Union(u, v)
```

Same_Components(u, v)

```
1   if Find_Set(u) = Find_Set(v)
2       then return true
3       else return false
```

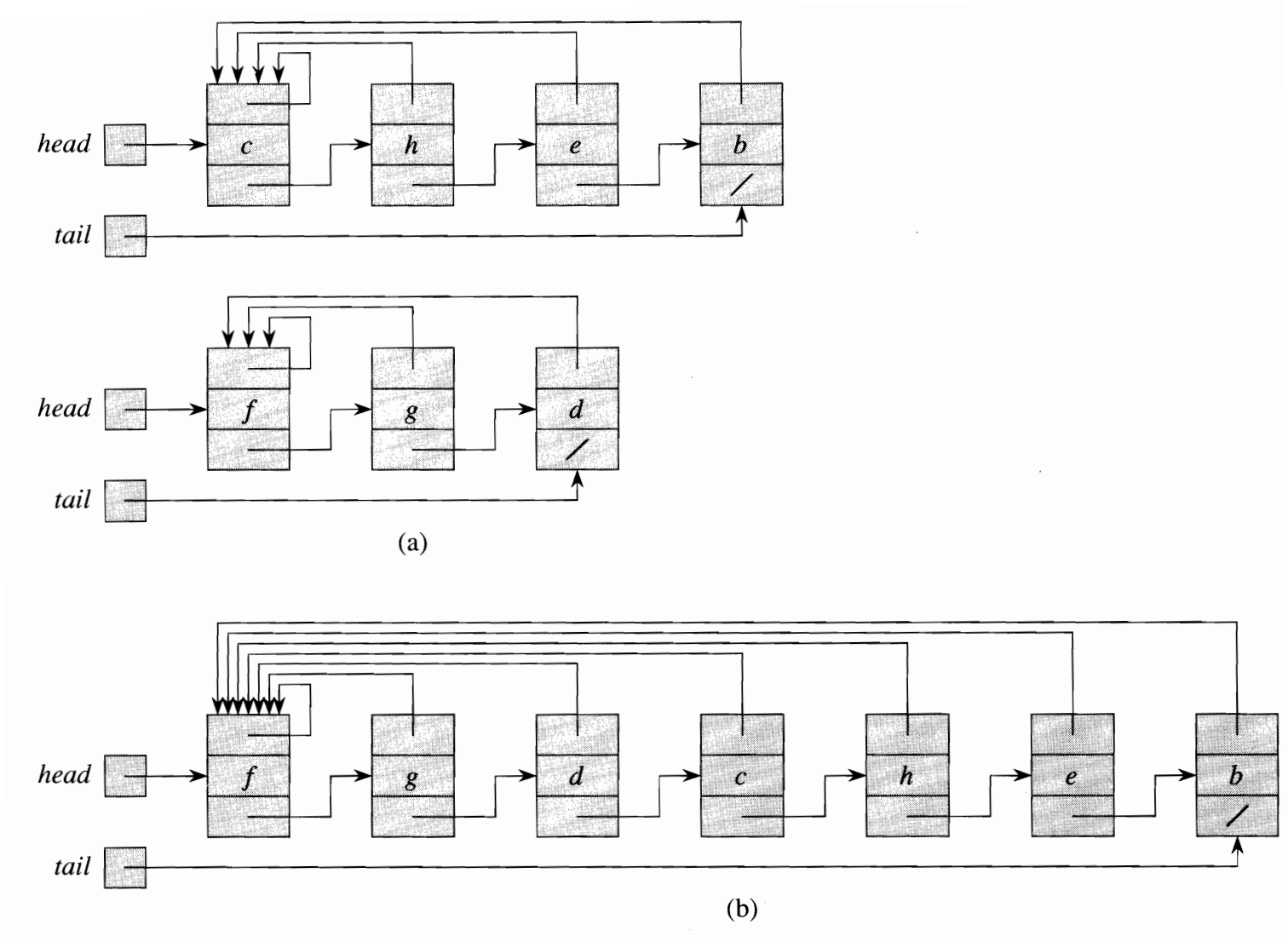
21.1 Example application: Connected Components of a Graph



(a)

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

21.2 Linked List Representation of Disjoint Sets



21.2 Linked List Representation (continued)

Assume that the location of each object in the linked list is given, then

Find_Set(x) has an obvious $O(1)$ implementation: just follow the back pointer.

Union(x, y) is more expensive, since many back pointers of one of the two merged sets need to be updated to (**the new representative**).

Example: consider n objects $x_1, x_2, \ldots, x_n$. There are n MAKE-SET(x_i) and m UNION(x_{i+1}, x_i) operations, i.e.,

- n: the number of **Make_Set** operations
- m: the total number of **Make_Set**, **Union**, and **Find_Set** operations

Using linked lists in an unfavorable way may require $\Theta(n^2)$ time.

21.2 Linked List Representation of Disjoint Sets

Weighted-union heuristic: When two lists are merged into one, always link the shorter list to the end of the longer list.

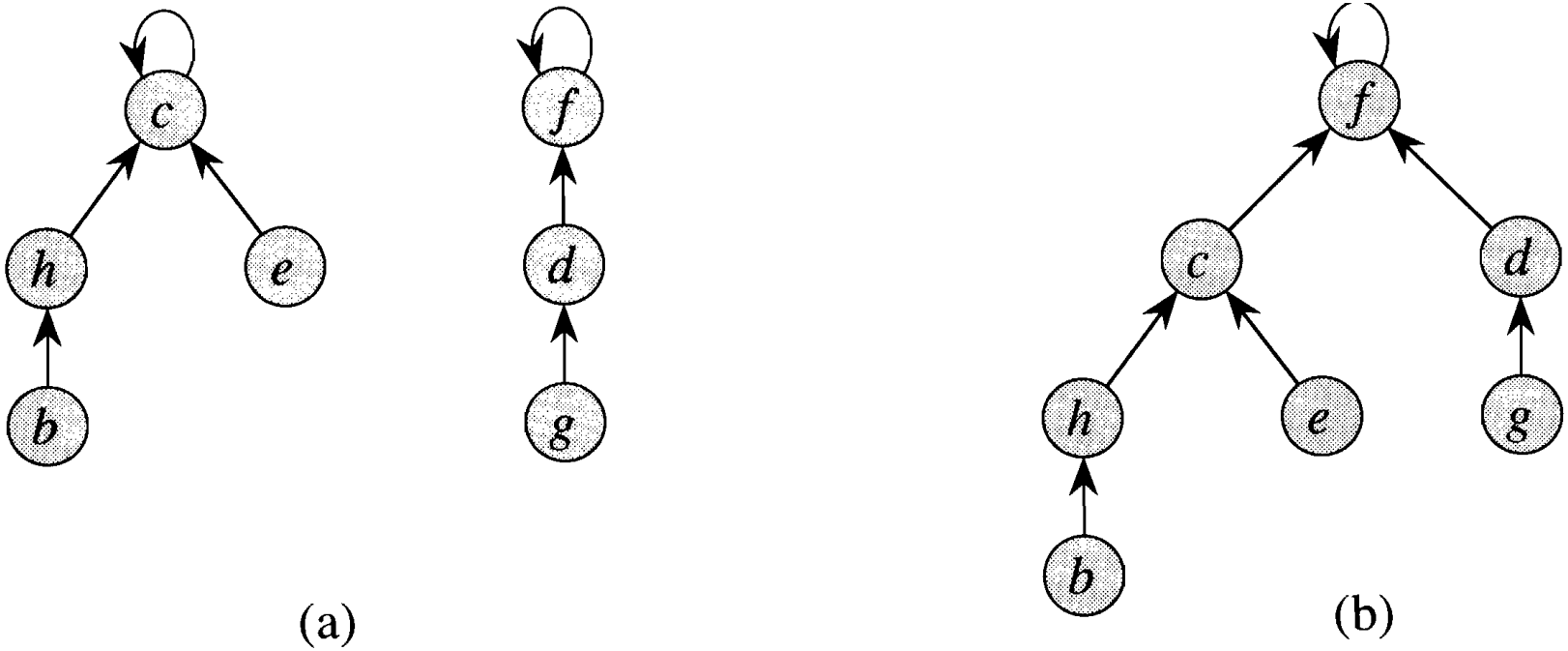
To implement the weighted-union heuristic, we need to maintain the length of each list. This is easily done by keeping the length with the representative (first element) of the set.

Theorem: Using the weighted-union heuristic, any sequence of m **Make_Set**, **Union**, and **Find_Set** operations, n of which are **Make_Set**, takes $O(m + n \log n)$ time.

Proof sketch: For any object x, whenever the back pointer of x is adjusted during **Union**, the size of the set containing x is at least doubled. Therefore, the back pointer of x is adjusted at most $\log n$ times, as there are at most n objects in the collection of all disjoint sets.

21.3 Directed Forest Representation of Disjoint Sets

A faster implementation of disjoint sets uses directed rooted trees, (some referres it to as an inverted tree). Each set is represented by a tree, rather than **a linked list**. Every node in a tree has only **a parent pointer**, and the tree root's pointer points itself.



21.3 Directed Forest Representation (continued)

- The **Make_Set** operation simply creates a tree with a single node.
- The **Find_Set** operation is performed by following parent pointers until the root of the tree is found. The object in the tree root is used to represent the set.
- The **Union** operation finds the roots of the two trees, then changes the parent pointer of one of the roots to point to the other.

The non-constant contribution to the running time of **Find_Set** or **Union** operations consists of following the path from a specified node to the root of its tree. To reduce the time spent in following these paths, two heuristics are used:

- Union by Rank
- Path Compression

21.3 The Union by Rank Heuristic

This is very similar to the weighted-union heuristic in linked lists:

When merging two trees, make the root pointer of the smaller "rank" tree point to the root of the larger rank tree.

Instead of maintaining the size of each tree, we maintain a value called the **rank** for each node, which is defined as follows.

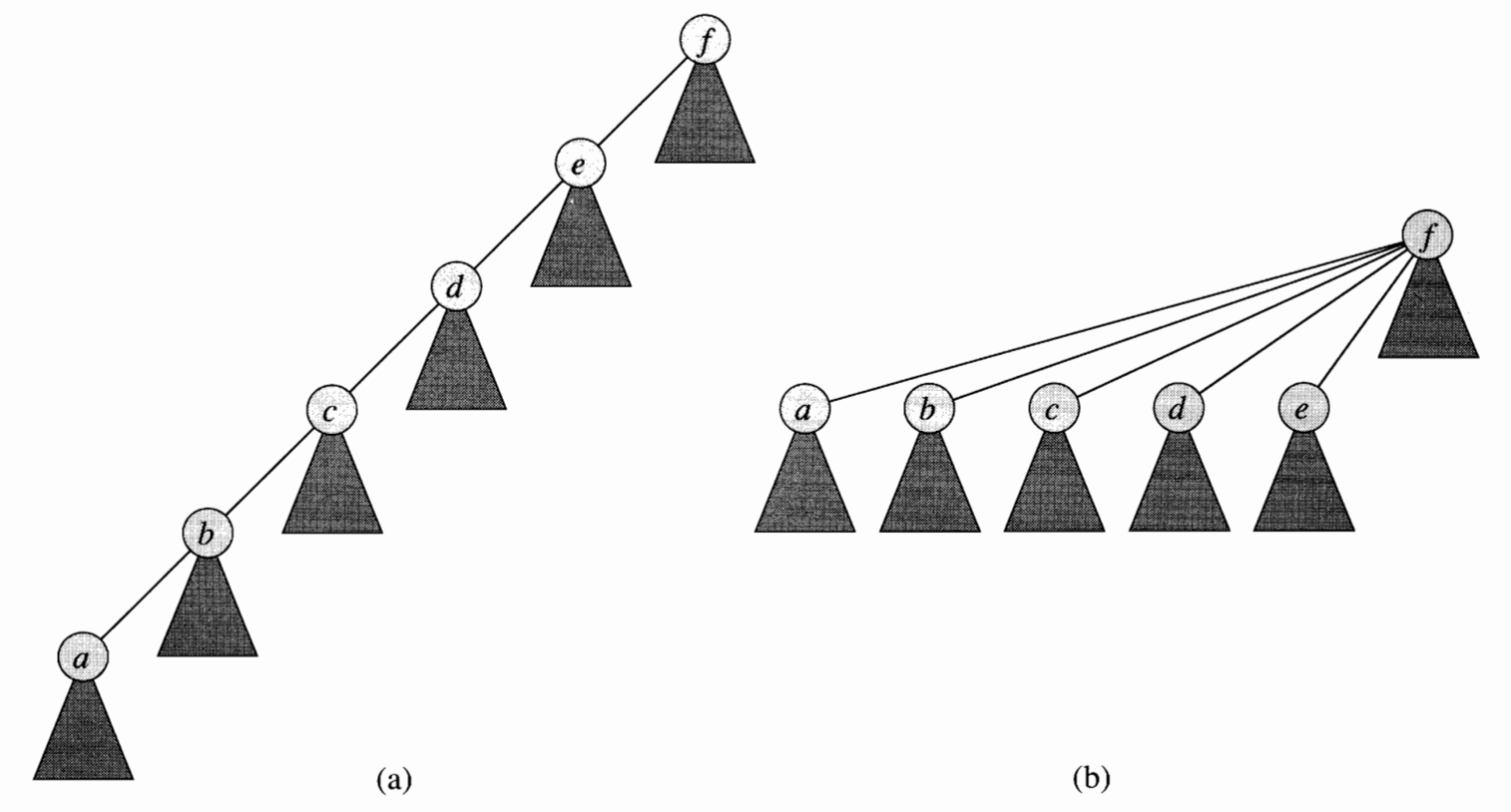
- A node which is the only node in a tree has rank 0.
- If the roots of two trees have the same rank k, the new root of their merge is given rank k+1. Otherwise, the rank of the new root is the larger rank between the two ranks.

Theorem: Any tree with root rank k has

- Height at most k.
- At least 2^k nodes.

21.3 The Path Compression Heuristic

When we identify the root of a tree containing some node a (which happens during **Find_Set** and **Union** operations), we set the parent pointers of all the nodes on the path from a to the root directly. (The rank of the tree does not changed.)



21.3 Disjoint Sets Using the Two Heuristics

For each node x, $p[x]$ is the parent and $rank[x]$ is its rank.

The condition $x = p[x]$ identifies the roots of trees.

1. Make_Set(x)

```
1    p[x] ← x
2    rank[x] ← 0
```

2. Find_Set(x)

```
1    if  x ≠ p[x]
2        then  p[x] ← Find_Set(p[x])
3    return p[x]
```

21.3 Disjoint Sets using the two heuristics (continued)

3. Union(x, y)

```
1    xroot ← Find_Set(x);
2    yroot ← Find_Set(y);
3    if rank[xroot] > rank[yroot]
4        then p[yroot] ← xroot
5        else p[xroot] ← yroot;
6    if rank[xroot] = rank[yroot]
7        then rank[yroot] ← rank[yroot] + 1
```

Theorem: If there are m operations altogether, including n **Make_Set** operations, the total running time is $O(m\,\alpha(n))$, where $\alpha(n)$ is an ***extremely*** slowly growing function (the inverse of the Ackermann function).

In fact, $\alpha(n) \leq 4$ for $n \leq 10^{600}$.

21.4 The Ackermann function

The Ackermann function is a very quickly growing function

$$A_k(j) = \begin{cases} j+1 & : k = 0 \\ A_{k-1}^{(j+1)}(j) & : k > 1, \end{cases}$$

where $A_{k-1}^{(0)}(j) = j$

$A_{k-1}^{(i)}(j) = A_{k-1}(A_{k-1}^{(i-1)}(j))$ for $i \geq 1$.

$A_2(j) = 2^{j+1}(j+1) - 1 \geq 2^j$

$A_3(j) \geq 2^{2^{2^{\cdots 2^j}}}$

The function $\alpha(n)$ is the inverse of $A_n(1)$ and grows very slowly.