Introduction

- Disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.

Introduction

- Disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.

- We identify each set by a representative, which is some member of the set.

Introduction

- Disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.

- We identify each set by a representative, which is some member of the set.

- In some applications, it doesn't matter which member is used as the representative; we care only that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times.

Introduction

- Disjoint-set data structure maintains a collection $\mathcal{S} = \{S_1, S_2, \ldots, S_k\}$ of disjoint dynamic sets.

- We identify each set by a representative, which is some member of the set.

- In some applications, it doesn't matter which member is used as the representative; we care only that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times.

- Other applications may require a prespecified rule for choosing the representative, such as choosing the smallest member in the set (assuming, of course, that the elements can be ordered).

Introduction

- Each element of a set is represented by an object.

Introduction

- Each element of a set is represented by an object.
- Letting x denote an object, we want to support the following operations:

Introduction

- Each element of a set is represented by an object.
- Letting x denote an object, we want to support the following operations:
  - MAKE-SET(x)
    - creates a new set whose only member (and thus representative) is x.

Introduction

- Each element of a set is represented by an object.
- Letting x denote an object, we want to support the following operations:
  - MAKE-SET(x)
    - creates a new set whose only member (and thus representative) is x.
    - Since the sets are disjoint, we require that x not already be in some other set.

Introduction

- Each element of a set is represented by an object.
- Letting x denote an object, we want to support the following operations:
  - MAKE-SET(x)
    - creates a new set whose only member (and thus representative) is x.
    - Since the sets are disjoint, we require that x not already be in some other set.
  - UNION(x, y)
    - Unites the dynamic sets that contain x and y, say Sx and Sy, into a new set that is the union of these two sets.

Introduction

- Each element of a set is represented by an object.
- Letting x denote an object, we want to support the following operations:
  - MAKE-SET(x)
    - creates a new set whose only member (and thus representative) is x.
    - Since the sets are disjoint, we require that x not already be in some other set.
  - UNION(x, y)
    - Unites the dynamic sets that contain x and y, say Sx and Sy, into a new set that is the union of these two sets.
    - We assume that the two sets are disjoint prior to the operation.

Introduction

- Each element of a set is represented by an object.
- Letting x denote an object, we want to support the following operations:
  - MAKE-SET(x)
    - creates a new set whose only member (and thus representative) is x.
    - Since the sets are disjoint, we require that x not already be in some other set.
  - UNION(x, y)
    - Unites the dynamic sets that contain x and y, say Sx and Sy, 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 Sx U Sy, although many implementations of UNION specifically choose the representative of either Sx or Sy as the new representative.

Introduction

- Each element of a set is represented by an object.
- Letting x denote an object, we want to support the following operations:
  - UNION(x, y)
    - Since we require the sets in the collection to be disjoint, conceptually we destroy sets Sx and Sy, removing them from the collection S.

Introduction

- Each element of a set is represented by an object.
- Letting x denote an object, we want to support the following operations:
  - UNION(x, y)
    - Since we require the sets in the collection to be disjoint, conceptually we destroy sets Sx and Sy, removing them from the collection S.
    - In practice, we often absorb the elements of one of the sets into the other set.

Introduction

- Letting x denote an object, we want to support the following operations:
  - FIND-SET(x)
    - returns a pointer to the representative of the (unique) set containing x.

Introduction

- Throughout the discussions, we shall analyse the running times of disjoint-set data structures in terms of two parameters

Introduction

- Throughout the discussions, we shall analyse the running times of disjoint-set data structures in terms of two parameters
  - m – total number of MAKE-SET, UNION & FIND-SET operations,

Introduction

- Throughout the discussions, we shall analyse the running times of disjoint-set data structures in terms of two parameters
  - m – total number of MAKE-SET, UNION & FIND-SET operations,
  - n – total number of MAKE-SET operations

Introduction

- Throughout the discussions, we shall analyse the running times of disjoint-set data structures in terms of two parameters
  - m – total number of MAKE-SET, UNION & FIND-SET operations,
  - n – total number of MAKE-SET operations
  - Max number of times UNION can be called
    - n – 1 (Since the sets are disjoint, each UNION operation reduces the number of sets by one. After n - 1 UNION operations, therefore, only one set remains.)

Introduction

- Throughout the discussions, we shall analyse the running times of disjoint-set data structures in terms of two parameters
  - m – total number of MAKE-SET, UNION & FIND-SET operations,
  - n – total number of MAKE-SET operations
  - Max number of times UNION can be called
    - n – 1 (Since the sets are disjoint, each UNION operation reduces the number of sets by one. After n - 1 UNION operations, therefore, only one set remains.)
  - m >= n (since the MAKE-SET operations are included in the total number of operations m)

Introduction

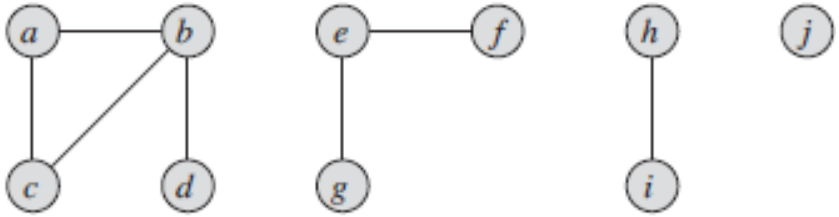
- Throughout the discussions, we shall analyse the running times of disjoint-set data structures in terms of two parameters
  - m – total number of MAKE-SET, UNION & FIND-SET operations,
  - n – total number of MAKE-SET operations
  - Max number of times UNION can be called
    - n – 1 (Since the sets are disjoint, each UNION operation reduces the number of sets by one. After n - 1 UNION operations, therefore, only one set remains.)
  - m >= n (since the MAKE-SET operations are included in the total number of operations m)
  - It is assumed that n MAKE-SET operations are the first n operations performed.

Data Structures for Disjoint Set

- An Application of Disjoint-Set Data Structures & its Operations
  - Determining the connected components (a connected component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.) of undirected graph

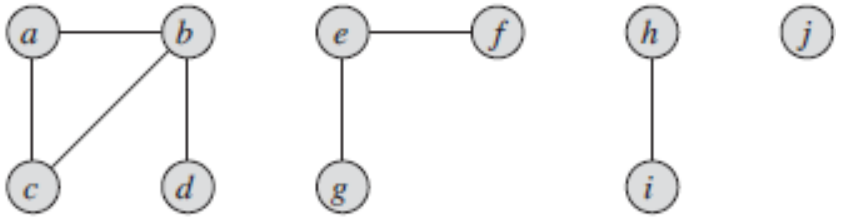
Data Structures for Disjoint Set

- An Application of Disjoint-Set Data Structures & its Operations
  - Determining the connected components (a connected component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.) of undirected graph
    - Following figure shows a graph with four connected components.



Data Structures for Disjoint Set

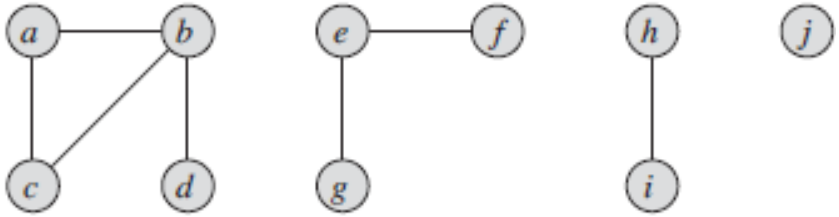
- An Application of Disjoint-Set Data Structures & its Operations
  - Determining the connected components (a connected component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.) of undirected graph
    - Following figure shows a graph with four connected components.



    - The procedure CONNECTED-COMPONENTS uses the disjoint-set operations to compute the connected components of a graph. (See Table)

Data Structures for Disjoint Set

- An Application of Disjoint-Set Data Structures & its Operations
  - Determining the connected components (a connected component of an undirected graph is a subgraph in which each pair of nodes is connected with each other via a path.) of undirected graph
    - Following figure shows a graph with four connected components.



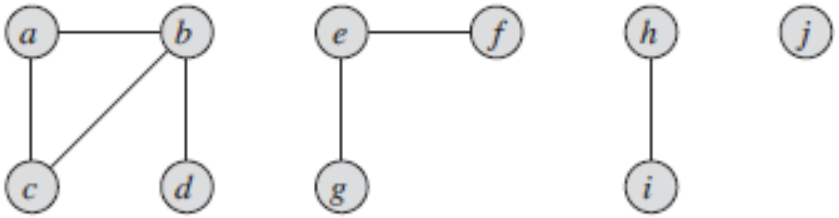
    - The procedure CONNECTED-COMPONENTS uses the disjoint-set operations to compute the connected components of a graph. (See Table)
    - Once CONNECTED-COMPONENTS has pre-processed the graph, the procedure SAME-COMPONENT answers queries about whether two vertices are in the same connected component.

Data Structures for Disjoint Set

- An Application of Disjoint-Set Data Structures & its Operations
  - Determining the connected components of undirected graph

```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ G.V
2      MAKE-SET(v)
3  for each edge (u, v) ∈ G.E
4      if FIND-SET(u) ≠ FIND-SET(v)
5          UNION(u, v)
```

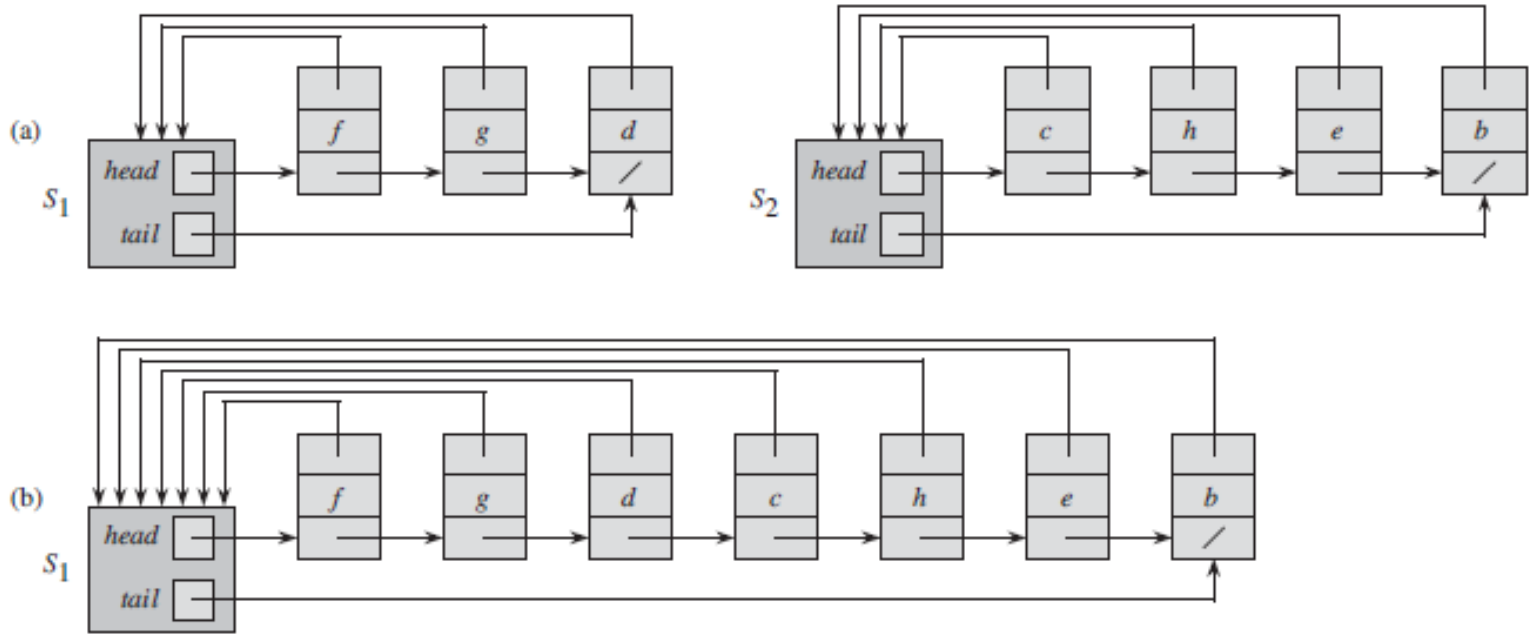
```
SAME-COMPONENT(u, v)
1  if FIND-SET(u) == FIND-SET(v)
2      return TRUE
3  else return FALSE
```



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}

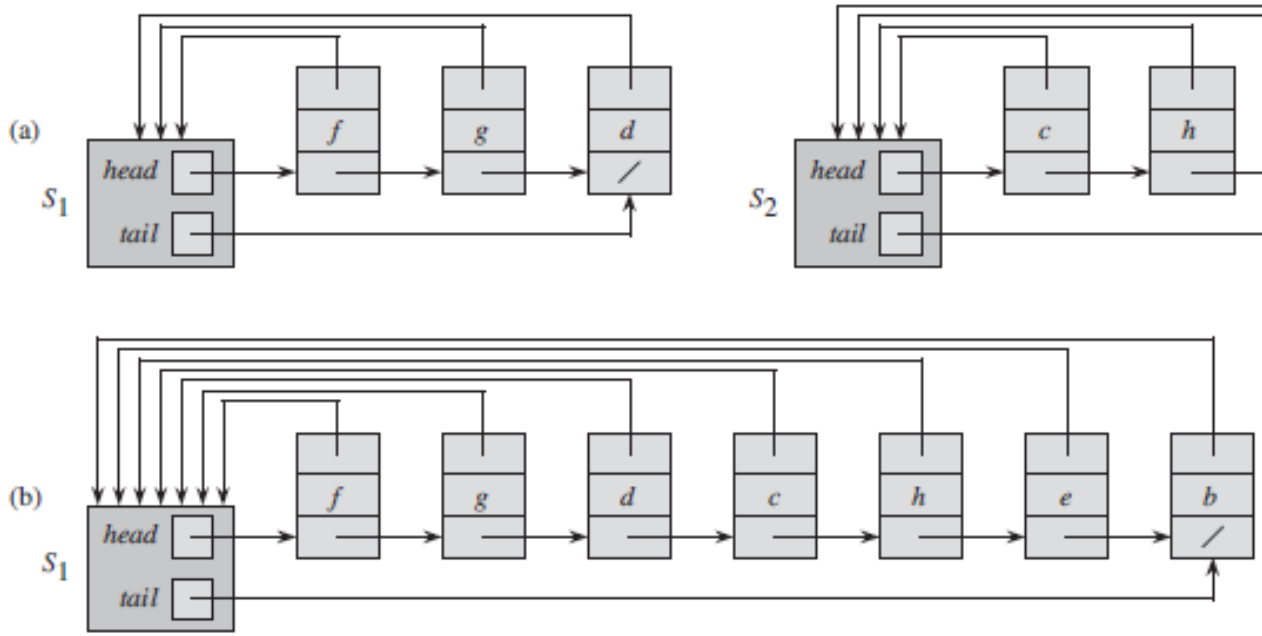
Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - Figure shows a simple way to implement a disjoint-set data structure: **each set is represented by its own linked list**.
  - The object for each set has attributes head, pointing to the first object in the list, and tail, pointing to the last object.



Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object.
  - Within each linked list, the objects may appear in any order.
  - The representative is the set member in the first object in the list.



Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets (Look at Union Later)

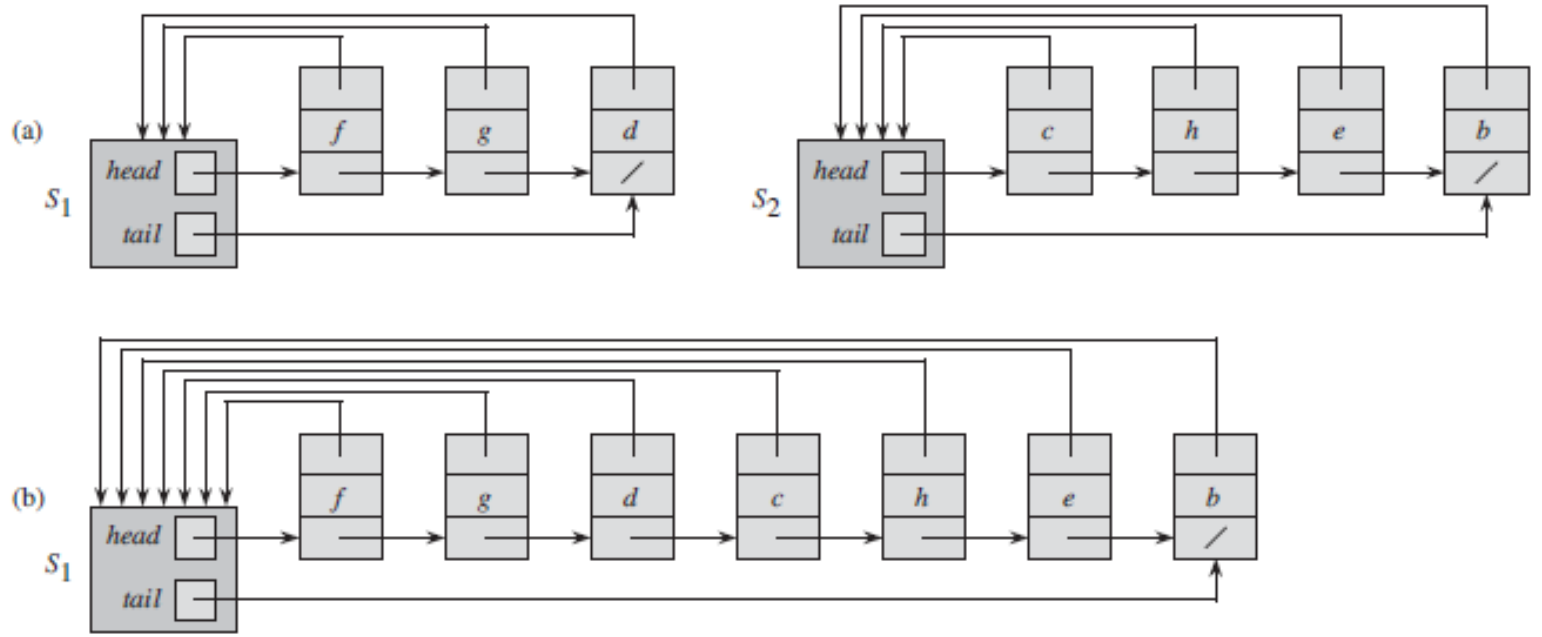


Figure 21.2 (a) Linked-list representations of two sets. Set S_1 contains members d, f, and g, with representative f, and set S_2 contains members b, c, e, and h, with representative c. Each object in the list contains a set member, a pointer to the next object in the list, and a pointer back to the set object. Each set object has pointers *head* and *tail* to the first and last objects, respectively. (b) The result of UNION(g, e), which appends the linked list containing e to the linked list containing g. The representative of the resulting set is f. The set object for e's list, S_2, is destroyed.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - With this linked-list representation, both MAKE-SET and FIND-SET are easy, requiring O(1) time.

  - To carry out MAKE-SET(x), we create a new linked list whose only object is x. (Here x is a value of the object for which a node is to be created. This procedure creates a set object and a node with x as a value and appropriate pointers. It also takes care of head & tail pointer)

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - With this linked-list representation, both MAKE-SET and FIND-SET are easy, requiring O(1) time.
  - To carry out MAKE-SET(x), we create a new linked list whose only object is x. (Here x is a value of the object for which a node is to be created. This procedure creates a set object and a node with x as a value and appropriate pointers. It also takes care of head & tail pointer)
  - For FIND-SET(x), we just follow the pointer from x back to its set object and then return the member in the object that head points to. (Here, x is a pointer referring to the node containing x)

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - With this linked-list representation, both MAKE-SET and FIND-SET are easy, requiring O(1) time.
  - To carry out MAKE-SET(x), we create a new linked list whose only object is x. (Here x is a value of the object for which a node is to be created. This procedure creates a set object and a node with x as a value and appropriate pointers. It also takes care of head & tail pointer)
  - For FIND-SET(x), we just follow the pointer from x back to its set object and then return the member in the object that head points to. (Here, x is a pointer referring to the node containing x)
  - For example, in Figure on last slide, the call FIND-SET(g) would return f.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET.
    - As earlier figure shows, we perform UNION(x, y) by appending y's list onto the end of x's list.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET.
    - As earlier figure shows, we perform UNION(x, y) by appending y's list onto the end of x's list.
    - The representative of x's list becomes the representative of the resulting set.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET.
    - As earlier figure shows, we perform UNION(x, y) by appending y's list onto the end of x's list.
    - The representative of x's list becomes the representative of the resulting set.
    - We use the tail pointer for x's list to quickly find where to append y's list.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET.
    - As earlier figure shows, we perform UNION(x, y) by appending y's list onto the end of x's list.
    - The representative of x's list becomes the representative of the resulting set.
    - We use the tail pointer for x's list to quickly find where to append y's list.
    - Because all members of y's list join x's list, we can destroy the set object for y's list. **(Note: Just set object is destroyed, other nodes will remain in their initial memory location)**

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - The simplest implementation of the UNION operation using the linked-list set representation takes significantly more time than MAKE-SET or FIND-SET.
    - As earlier figure shows, we perform UNION(x, y) by appending y's list onto the end of x's list.
    - The representative of x's list becomes the representative of the resulting set.
    - We use the tail pointer for x's list to quickly find where to append y's list.
    - Because all members of y's list join x's list, we can destroy the set object for y's list. **(Note: Just set object is destroyed, other nodes will remain in their initial memory location)**
    - Unfortunately, we must update the pointer to the set object for each object originally on y's list, which takes time linear in the length of y's list.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - In Figure 21.2, for example, the operation UNION(g, e) causes pointers to be updated in the objects for b, c, e, and h.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - Suppose that we have objects x_1, x_2, …, x_n.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - Suppose that we have objects $x_1, x_2, \ldots, x_n$.
    - Assume that we have executed the sequence of n MAKE-SET operations to create these objects.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - Suppose that we have objects $x_1, x_2, \ldots, x_n$.
    - Assume that we have executed the sequence of n MAKE-SET operations to create these objects.
    - Now let us assume that these MAKE-SET operations are followed by n - 1 UNION operations as shown in following figure, so that m = 2n - 1.

Operation	Number of objects updated
MAKE-SET(x_1)	1
MAKE-SET(x_2)	1
⋮	⋮
MAKE-SET(x_n)	1
UNION(x_2, x_1)	1
UNION(x_3, x_2)	2
UNION(x_4, x_3)	3
⋮	⋮
UNION(x_n, x_{n-1})	$n-1$

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - We spend O(n) time performing the n MAKE-SET operations.

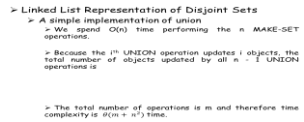
Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - 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 by all n - 1 UNION operations is

$$\sum_{i=1}^{n-1} i = \Theta(n^2)$$

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A simple implementation of union
    - 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 by all n - 1 UNION operations is



    - The total number of operations is m and therefore time complexity is $\theta(m + n^2)$ time.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - Through this heuristic, we will try to achieve a tighter upper bound on a sequence of m MAKE-SET, UNION and FIND-SET operations.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - Through this heuristic, we will try to achieve a tighter upper bound on a sequence of m MAKE-SET, UNION and FIND-SET operations.
    - In this heuristic, it is assumed that each list also includes the length of the list and that we always append the shorter list onto the longer, breaking ties arbitrarily.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - Through this heuristic, we will try to achieve a tighter upper bound on a sequence of m MAKE-SET, UNION and FIND-SET operations.
    - In this heuristic, it is assumed that each list also includes the length of the list and that we always append the shorter list onto the longer, breaking ties arbitrarily.
    - Consider a particular object x

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - Through this heuristic, we will try to achieve a tighter upper bound on a sequence of m MAKE-SET, UNION and FIND-SET operations.
    - In this heuristic, it is assumed that each list also includes the length of the list and that we always append the shorter list onto the longer, breaking ties arbitrarily.
    - Consider a particular object x
    - We know that each time x's pointer was updated, x must have started in the smaller set.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - Through this heuristic, we will try to achieve a tighter upper bound on a sequence of m MAKE-SET, UNION and FIND-SET operations.
    - In this heuristic, it is assumed that each list also includes the length of the list and that we always append the shorter list onto the longer, breaking ties arbitrarily.
    - Consider a particular object x
    - We know that each time x's pointer was updated, x must have started in the smaller set.
    - The first time x's pointer was updated, therefore, the resulting set must have had at least 2 members.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - Similarly, the next time x's pointer was updated, the resulting set must have had at least 4 members.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - Similarly, the next time x's 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 pointer has been updated $\lceil \lg k \rceil$ times, the resulting set must have at least k members.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - Similarly, the next time x's 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 pointer has been updated $\lceil \lg k \rceil$ times, the resulting set must have at least k members.
    - Since the largest resulting set can have at most n members, **each** object's pointer is updated at most $\lceil \lg n \rceil$ times over all the UNION operations.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - Similarly, the next time x's 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 pointer has been updated $\lceil \lg k \rceil$ times, the resulting set must have at least k members.
    - Since the largest resulting set can have at most n members, **each** object's pointer is updated at most $\lceil \lg n \rceil$ times over all the UNION operations.
    - Thus the total time spent updating object pointers over all UNION operations is O(n lg n) (as there are n objects).

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - We must also account for updating the tail pointers and the list lengths, which take only $\theta(1)$ time per UNION operation.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - We must also account for updating the tail pointers and the list lengths, which take only $\theta(1)$ time per UNION operation.
    - The total time spent in all UNION operations is thus O(n lg n).

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - We must also account for updating the tail pointers and the list lengths, which take only $\theta(1)$ time per UNION operation.
    - The total time spent in all UNION operations is thus O(n lg n).
    - The time for the entire sequence of m operations follows easily. Each MAKESET and FIND-SET operation takes O(1) time, and there are O(m) of them.

Data Structures for Disjoint Set

- Linked List Representation of Disjoint Sets
  - A weighted-union heuristic
    - We must also account for updating the tail pointers and the list lengths, which take only $\theta(1)$ time per UNION operation.
    - The total time spent in all UNION operations is thus $O(n \lg n)$.
    - The time for the entire sequence of m operations follows easily. Each MAKESET and FIND-SET operation takes $O(1)$ time, and there are $O(m)$ of them.
    - The total time for the entire sequence is thus $O(m + n \lg n)$

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - In a faster implementation of disjoint sets, we represent sets by rooted trees, with each node containing one member and each tree representing one set.
  - In a disjoint-set forest, illustrated in Figure 21.4(a), each member points only to its parent.

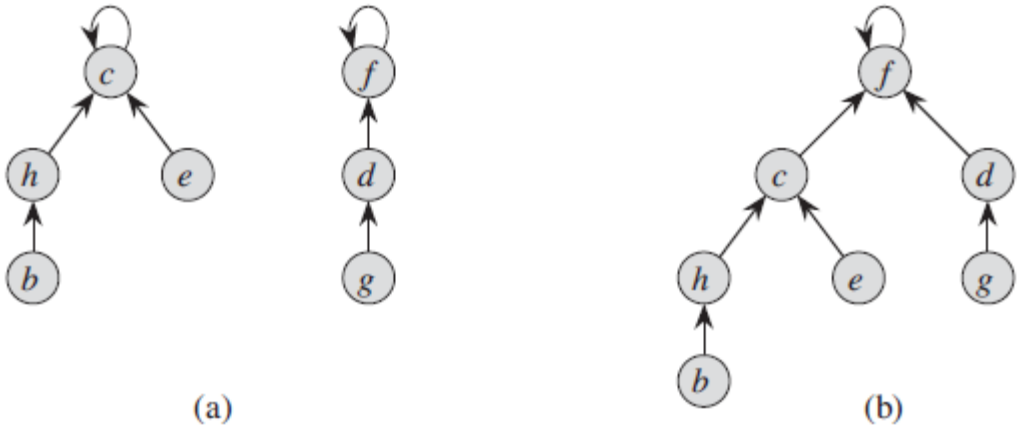


Figure 21.4 A disjoint-set forest. **(a)** Two trees representing the two sets of Figure 21.2. The tree on the left represents the set $\{b, c, e, h\}$, with c as the representative, and the tree on the right represents the set $\{d, f, g\}$, with f as the representative. **(b)** The result of UNION(e, g).

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - The root of each tree contains the representative and is its own parent.
  - As we shall see, although the straightforward algorithms that use this representation are no faster than ones that use the linked-list representation, by introducing two heuristics—"union by rank" and "path compression"—we can achieve an asymptotically optimal disjoint-set data structure.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - We perform the three disjoint-set operations as follows.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - We perform the three disjoint-set operations as follows.
  - A MAKE-SET operation simply creates a tree with just one node. (constant time)

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - We perform the three disjoint-set operations as follows.
  - A MAKE-SET operation simply creates a tree with just one node. (constant time)
  - We perform a FIND-SET operation by following parent pointers until we find the root of the tree. (depends on the height of the tree or the rank of the root)

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - We perform the three disjoint-set operations as follows.
  - A MAKE-SET operation simply creates a tree with just one node. (constant time)
  - We perform a FIND-SET operation by following parent pointers until we find the root of the tree. (depends on the height of the tree or the rank of the root)
  - The nodes visited on this simple path toward the root constitute the find path.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - We perform the three disjoint-set operations as follows.
  - A MAKE-SET operation simply creates a tree with just one node. (constant time)
  - We perform a FIND-SET operation by following parent pointers until we find the root of the tree. (depends on the height of the tree or the rank of the root)
  - The nodes visited on this simple path toward the root constitute the find path.
  - A UNION operation, shown in Figure 21.4(b), causes the root of one tree to point to the root of the other. (depends on find set)

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - Heuristics to improve the running time
    - So far, we have not improved on the linked-list implementation.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - Heuristics to improve the running time
    - So far, we have not improved on the linked-list implementation.
    - Union by Rank
      - It is similar to the weighted-union heuristic we used with the linked-list representation.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - Heuristics to improve the running time
    - So far, we have not improved on the linked-list implementation.
    - Union by Rank
      - It is similar to the weighted-union heuristic we used with the linked-list representation.
      - Idea is to make the root of the tree with less height point to the root of the tree with more height.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - Heuristics to improve the running time
    - So far, we have not improved on the linked-list implementation.
    - Union by Rank
      - It is similar to the weighted-union heuristic we used with the linked-list representation.
      - Idea is to make the root of the tree with less height point to the root of the tree with more height.
      - For each node, we maintain a rank, which is an upper bound on the height of the node.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - Heuristics to improve the running time
    - So far, we have not improved on the linked-list implementation.
    - Union by Rank
      - It is similar to the weighted-union heuristic we used with the linked-list representation.
      - Idea is to make the root of the tree with less height point to the root of the tree with more height.
      - For each node, we maintain a rank, which is an upper bound on the height of the node.
      - In union by rank, we make the root with smaller rank point to the root with larger rank during a UNION operation.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - Heuristics to improve the running time
    - Path Compression
      - We use it during FIND-SET operations to make each node on the find path point directly to the root. We do not change any ranks.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - Path Compression

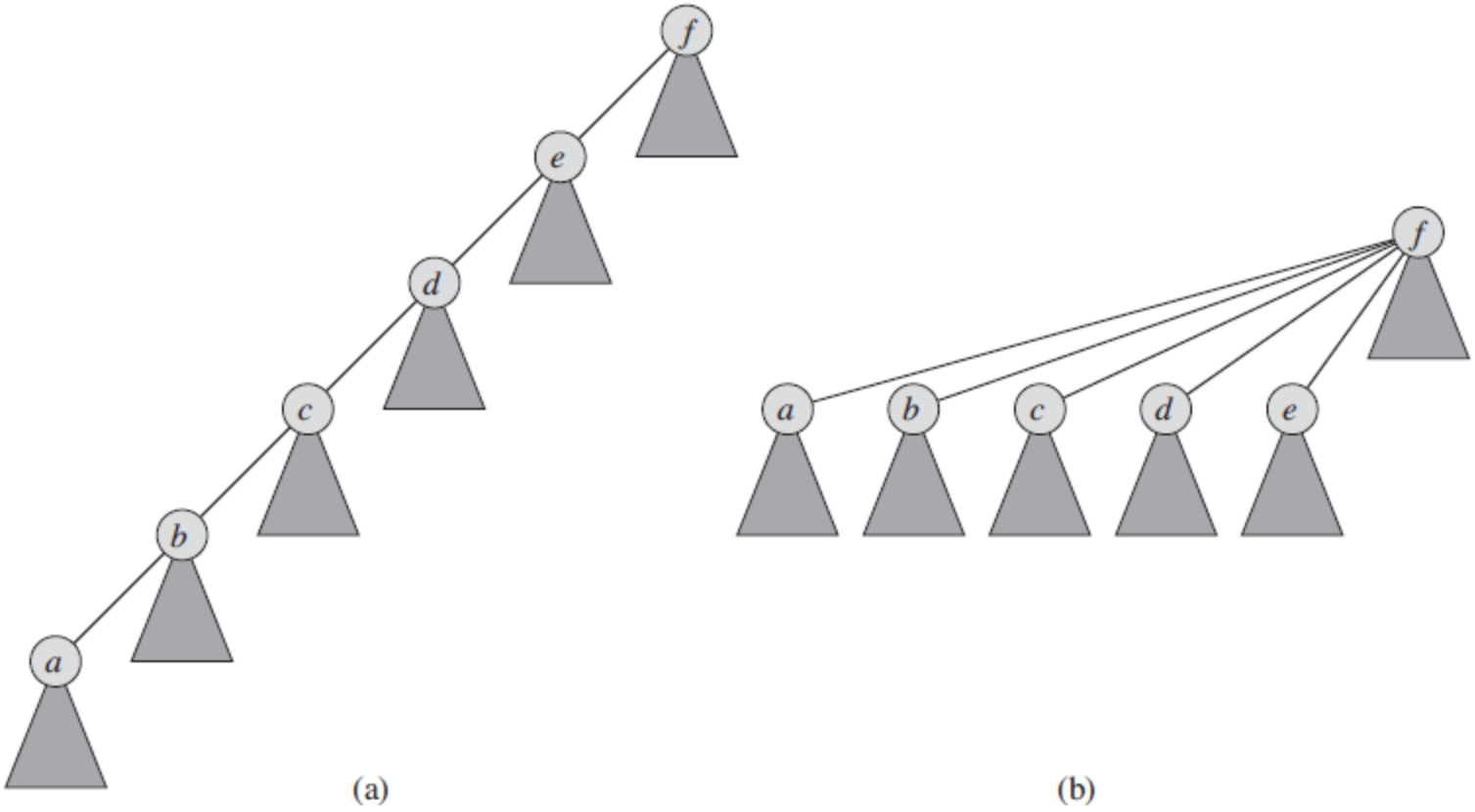


Figure 21.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing FIND-SET(a). Each node on the find path now points directly to the root.

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - Pseudocode

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0

UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))

LINK(x, y)
1  if x.rank > y.rank
2       y.p = x
3  else x.p = y
4       if x.rank == y.rank
5            y.rank = y.rank + 1
```

The FIND-SET procedure with path compression is quite simple:

```
FIND-SET(x)
1  if x ≠ x.p
2       x.p = FIND-SET(x.p)
3  return x.p
```

Data Structures for Disjoint Set

- Rooted Tree Representation of Disjoint Sets
  - Running Time Analysis
    - When we use both union by rank and path compression, the worst-case running time is $O(m\,\alpha(n))$, where $\alpha(n)$ is a *very* slowly growing function.

      In any conceivable application of a disjoint-set data structure, a(n) <= 4; thus, we can view the running time as linear in m in all practical situations.

Disclaimer

- Contents of this presentation are not original and they have been prepared from various sources just for the teaching purpose.