Disjoint Set Data Structure
and
Huffman Coding

Disjoint Set or Union Find Data Structure

- Union-Find Algorithm: This is the main algorithm used in disjoint set data structure. It helps in creating and merging sets, and finding the parent or representative element of a set.

- Path Compression: This is an optimization technique used in disjoint set data structure that helps to reduce the time complexity of the find operation by compressing the path from a node to its parent.

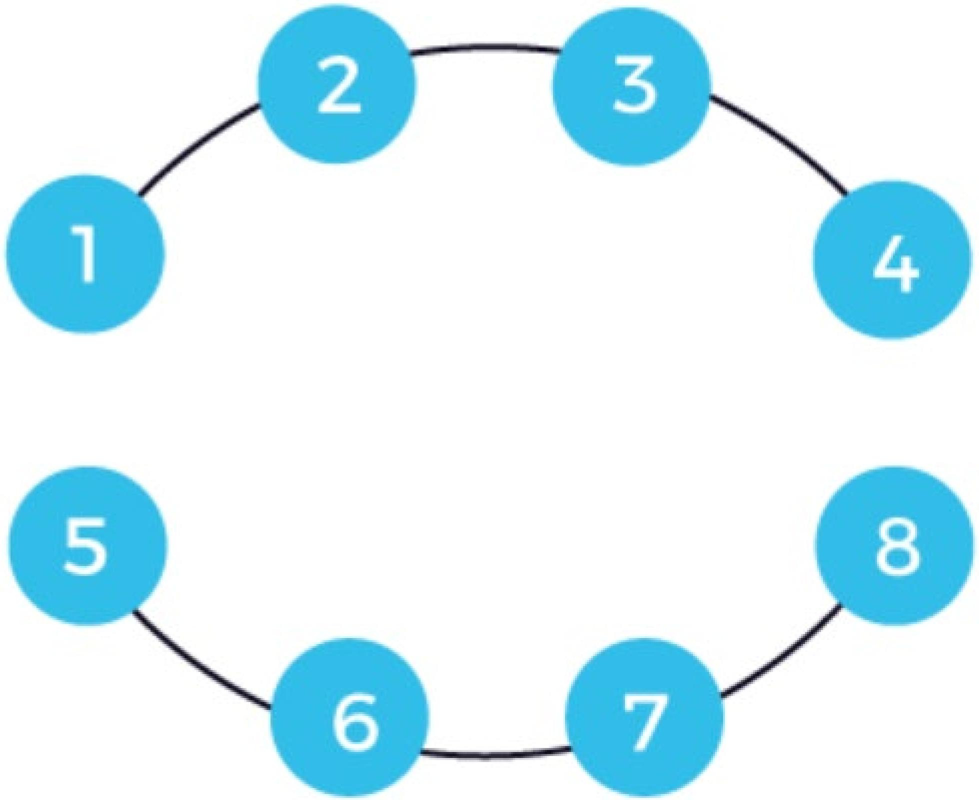
- Union by Rank: This is another optimization technique used in disjoint set data structure that helps to reduce the time complexity of the union operation by merging smaller sets into larger sets.

What is Disjoint Set?

- Disjoint set is a data structure that is used to maintain a collection of disjoint (non-overlapping) sets.

- The elements in each set are related to each other through a parent-child relationship, where the parent of an element is either itself or another element in the same set.

- Since there is no common element between these two sets, s1 and s2, we will not get anything if we consider the intersection between these two sets.



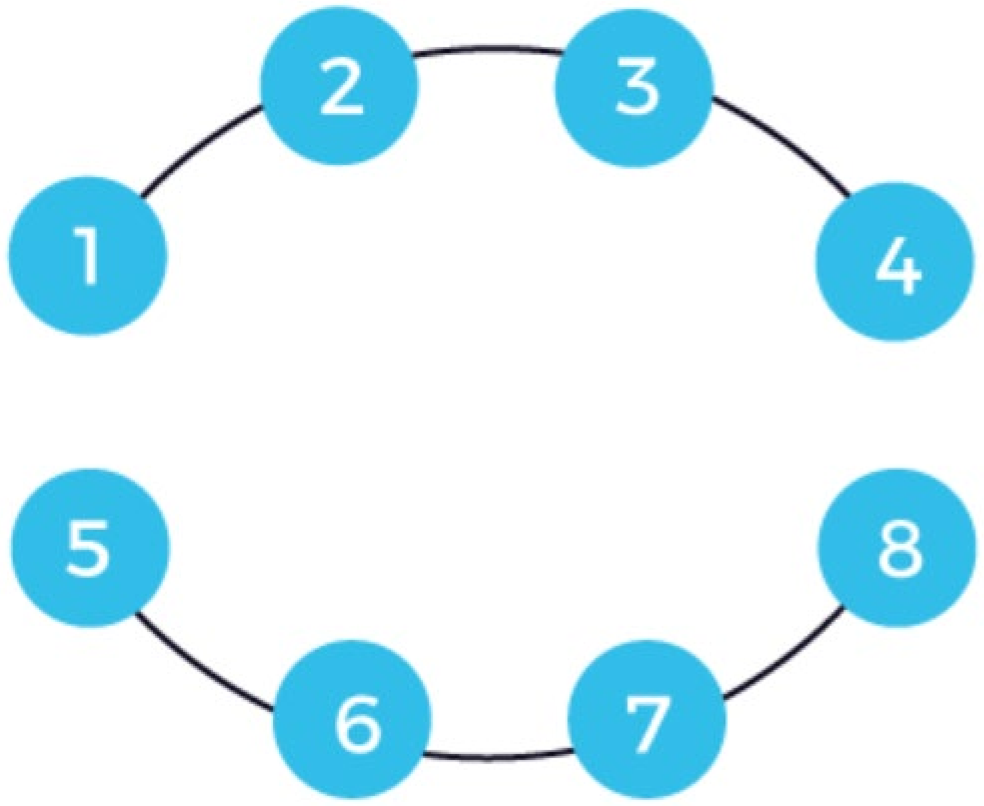
s1 = {1, 2, 3, 4}

s2 = {5, 6, 7, 8}

What is Disjoint Set? (cont.)

- The two main operations that can be performed on a disjoint set data structure are the Union and Find operations.

- It is also called a union–find data structure as it supports union and find operation on subsets.



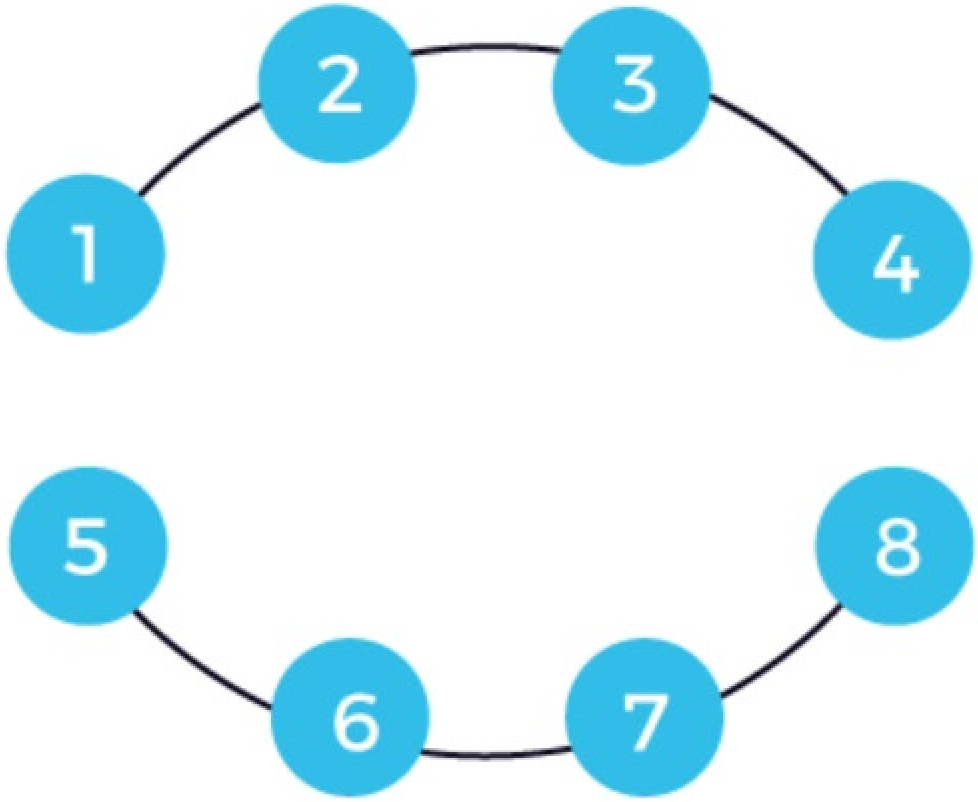
s1 = {1, 2, 3, 4}

s2 = {5, 6, 7, 8}

What is Disjoint Set? (cont.)

- The two main operations that can be performed on a disjoint set data structure are the Union and Find operations.

- It is also called a union–find data structure as it supports union and find operation on subsets.



s1 = {1, 2, 3, 4}

s2 = {5, 6, 7, 8}

Union-Find algorithm

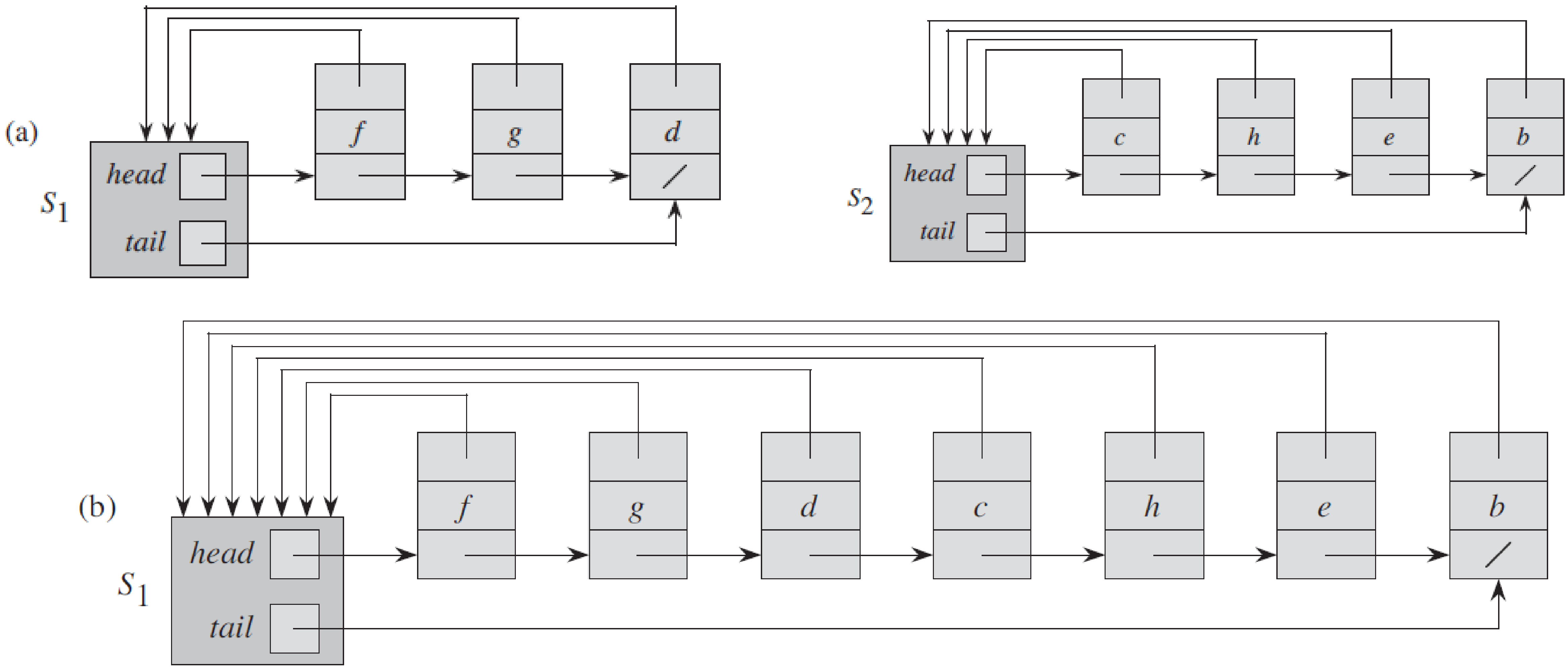
Find

To find the subset a particular element 'k' belongs to. It is generally used to check if two elements belong to the same subset or not.

Union

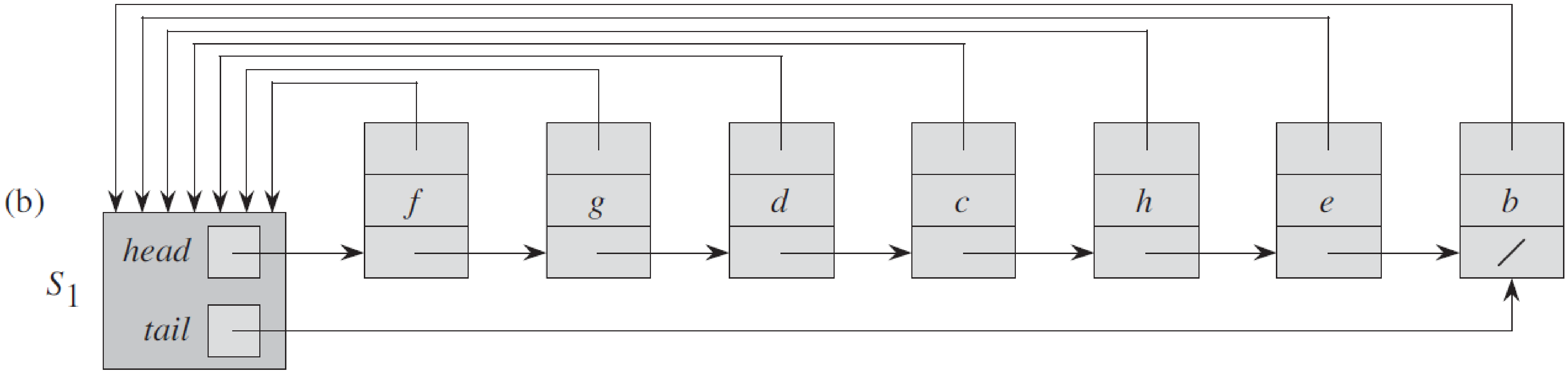
It is used to combine two subsets into one. A union query, say Union(x, y) combines the set containing element x and set containing element y.

Linked List Representation of Disjoint Sets



Each set object has pointers *head* and *tail* to the first and last objects

Linked List Representation of Disjoint Sets

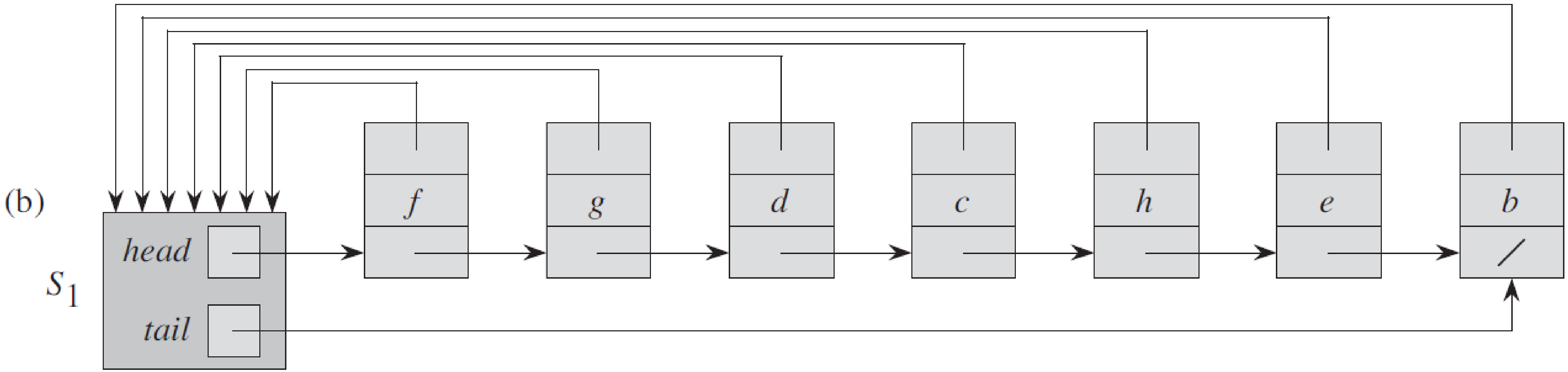


we perform UNION(x, y) by appending y's list onto the end of x's list.

We use the *tail* pointer for x's list to quickly find where to append y's list.

- 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.

A weighted-union heuristic



we perform UNION(x, y) by appending y's list onto the end of x's list.

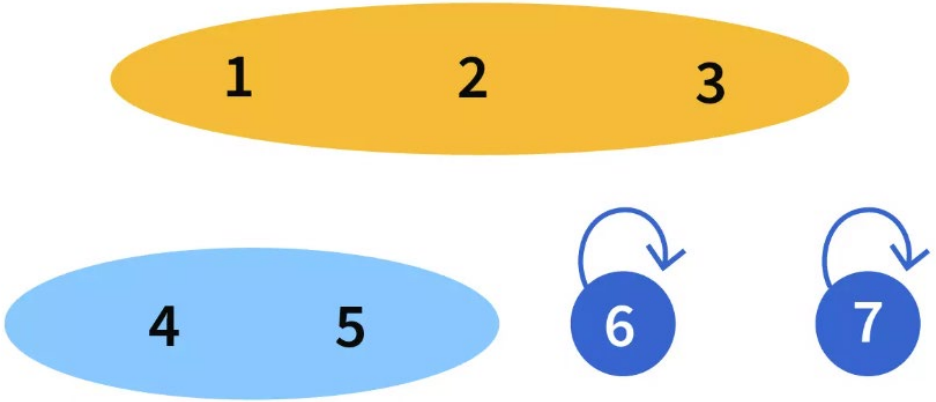
- we must update the pointer to the set object for each object originally on y's list,
  - which takes time linear in the length of y's list.
- In the worst case, the above implementation of the UNION procedure requires an average of time per call because we may be appending a longer list onto a shorter list
- We always append the shorter list onto the longer, breaking ties arbitrarily. Called weighted-union heuristic

Disjoint Set Forest

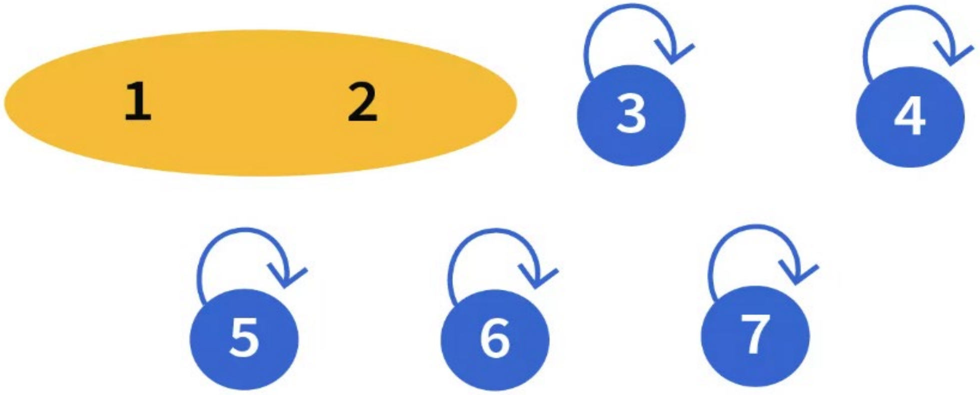
- we represent sets by rooted trees, with each node containing one member and each tree representing one set.
- Straight forward representation of this are no faster than the linked list version. We can use heuristic!!
- Each tree corresponds to one set and the root of the tree will be the parent/leader/representative of the set.
- All the seven nodes are parents of themselves. we have seven different trees cor

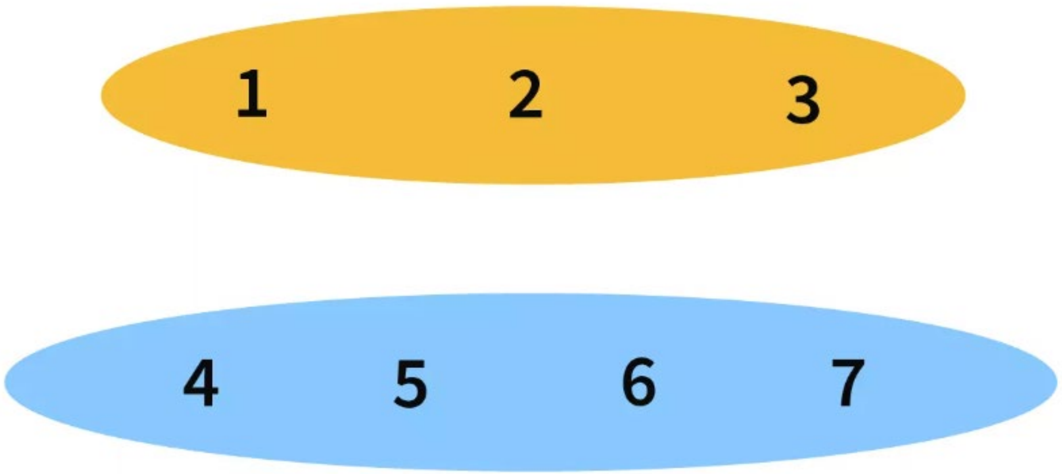
Union

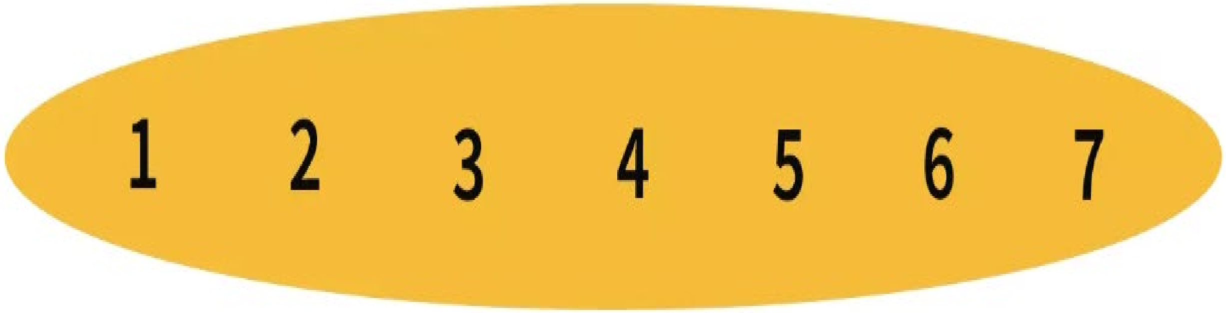
In *Union(2, 3),* need to join the sets which contain elements 2 and 3.
After performing the query we can see that, 1, 2, and 3 are clubbed into one set so our disjoint set will look like:



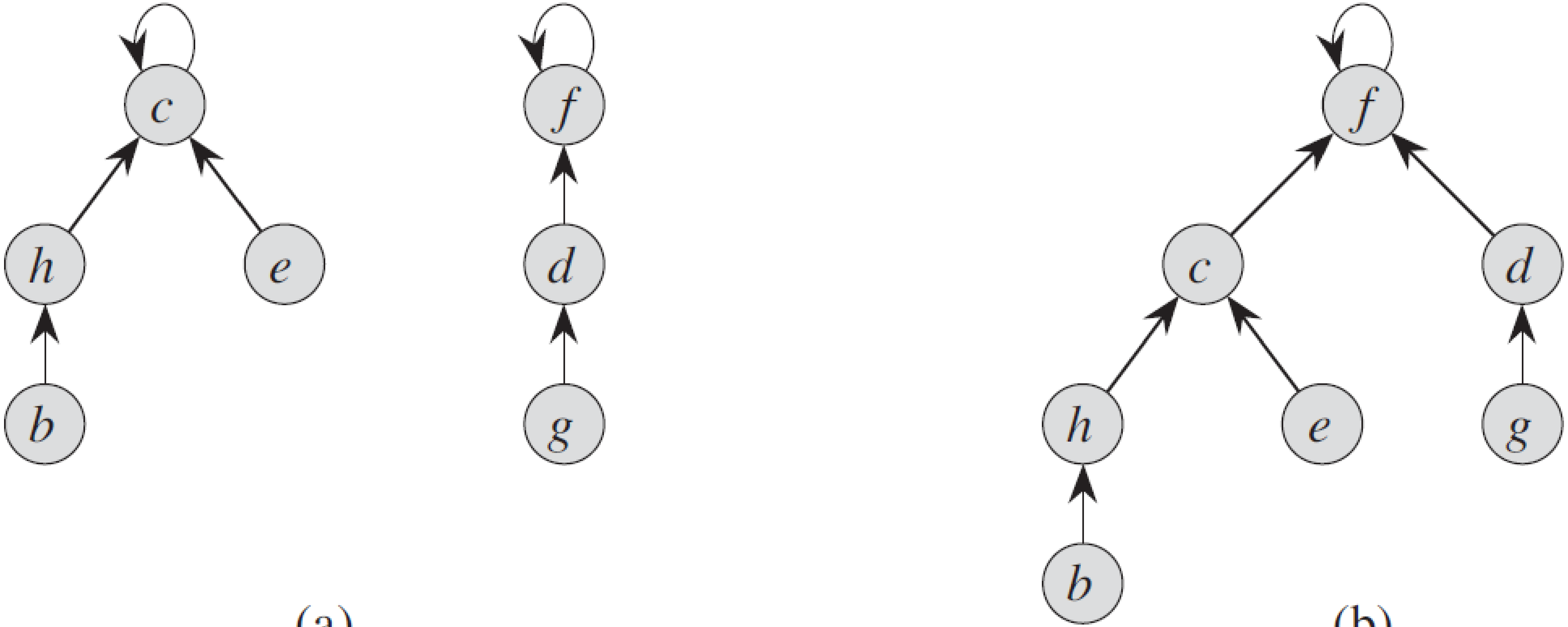
- Union(1, 2)
- Union(2, 3)
- Union(4, 5)
- Union(6, 7)
- Union(5, 6)
- Union(2, 6)







Example



(a) (b)

Naïve Implementation of Disjoint Set

Complexity Analysis:

- Find - Time complexity of Find operation is $O(n)$ in the worst case (consider the case when all the elements are in the same set and we need to find the parent of a given element then we may need to make n recursive calls).
- Union - For union query (say *Union(u, v)*) we need to find the parents of u and v making its time complexity to be $O(n)$.

Heuristics to improve the running time

❑ Path Compression

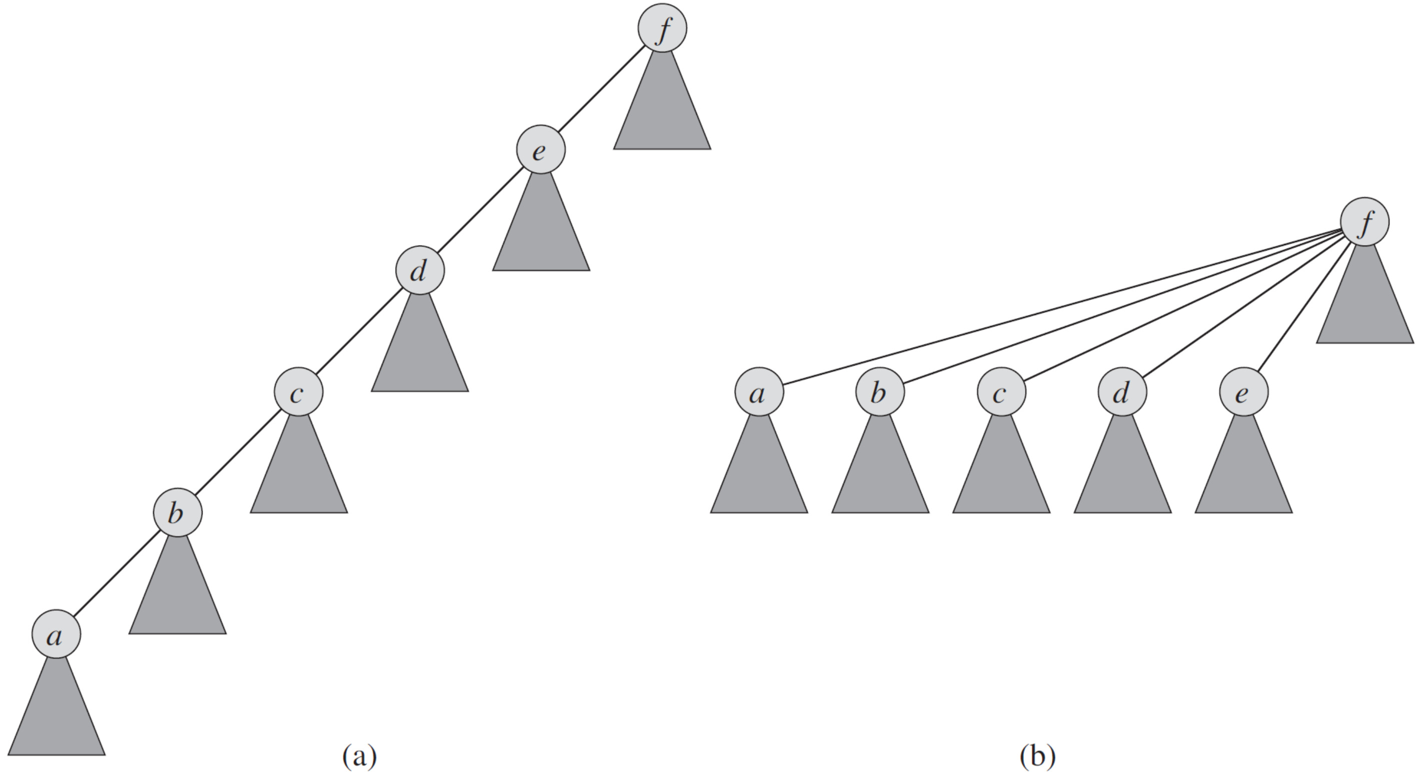
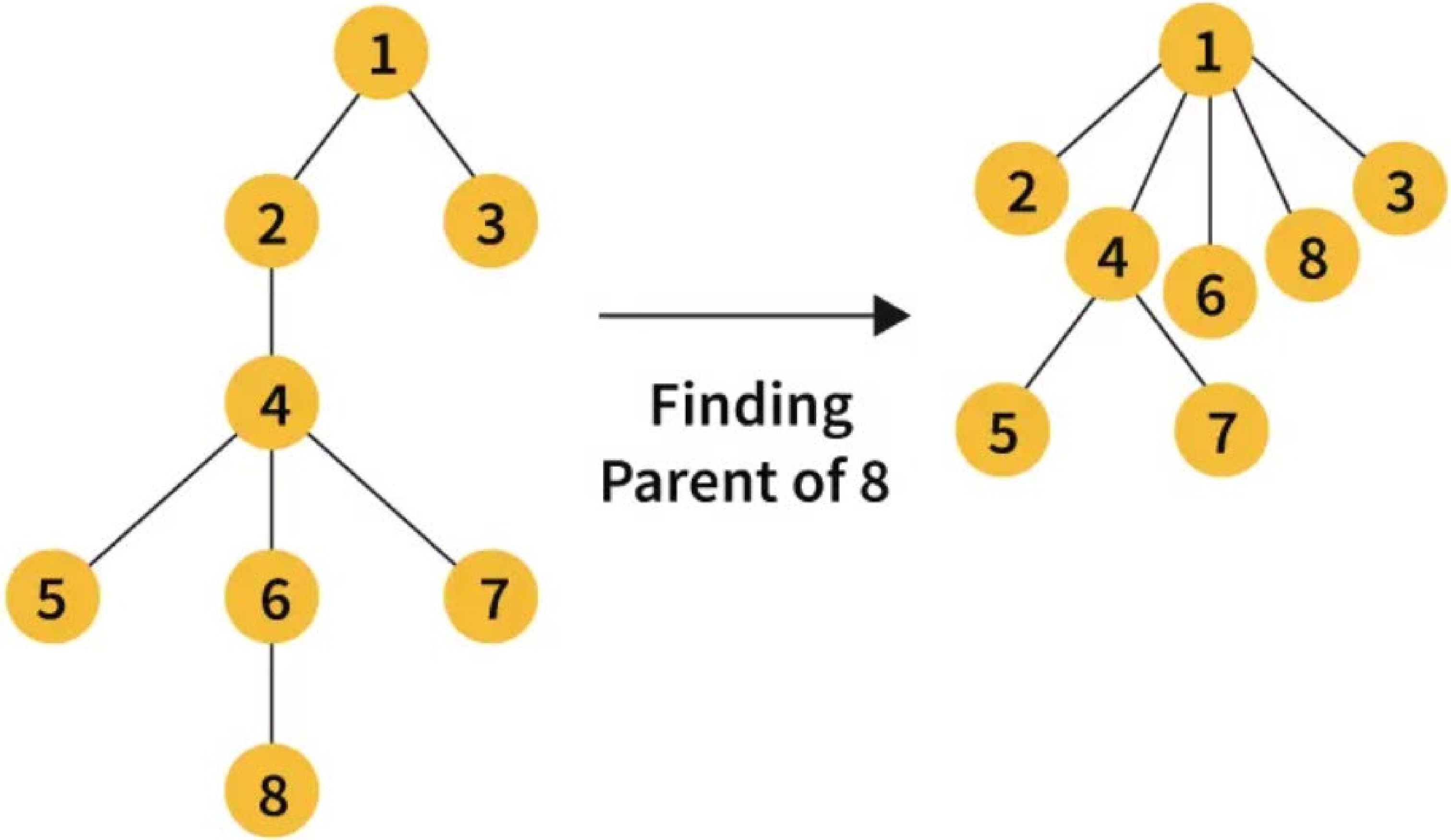


Figure 21.5 Path compression during the operation FIND-SET. Arrows and self-loops at roots are omitted. **(a)** A tree representing a set prior to executing FIND-SET(a). Triangles represent subtrees whose roots are the nodes shown. Each node has a pointer to its parent. **(b)** The same set after executing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

Heuristics to improve the running time

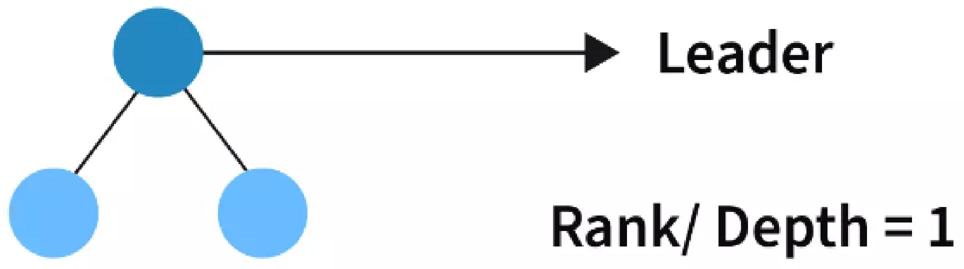
- Path Compression

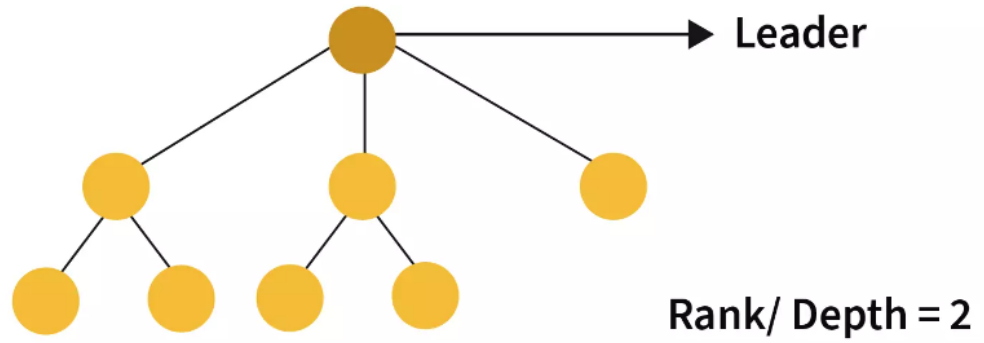


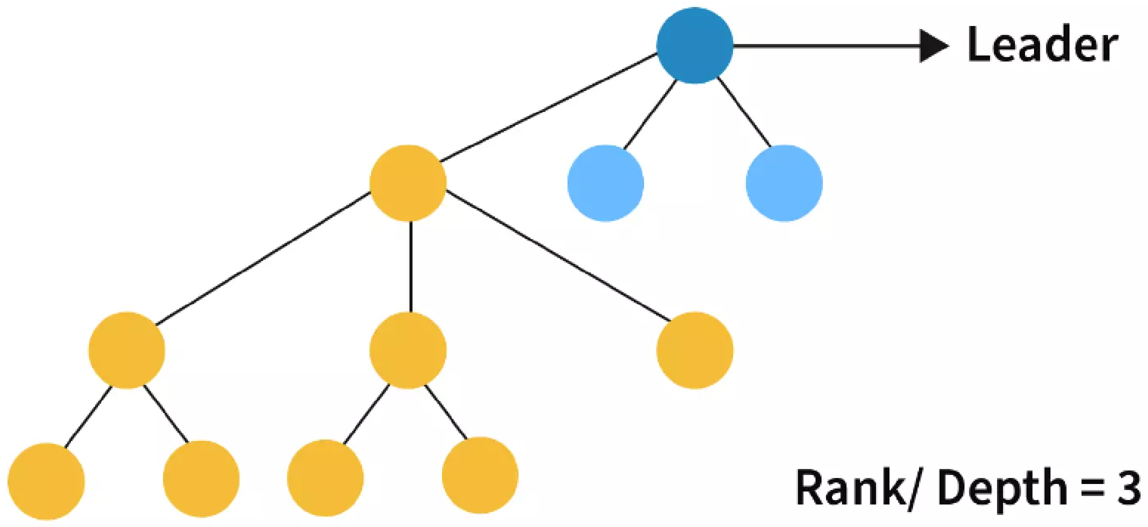
we reduced the time complexity from $O(n)$ to $O(log(n))$

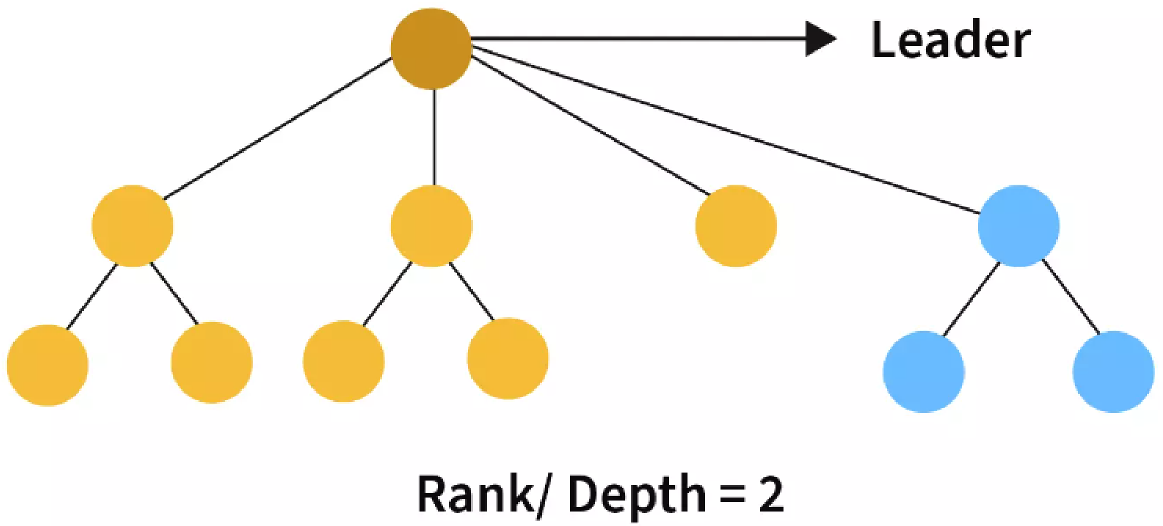
Heuristic: Union by Rank, and combination of the two heuristics?

- 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.









Use Union Find Algorithm to detect cycle?

- Initially create a **parent[]** array to keep track of the subsets.
- Traverse through all the edges:
  - Check to which subset each of the nodes belong to by finding the parent[] array till the node and the parent are the same.
  - If the two nodes belong to the same subset then they belong to a cycle.
  - Otherwise, perform union operation on those two subsets.
- If no cycle is found, return false.

Use Union Find Algorithm to detect cycle?

***parent[] = {0, 1, 2}**. Also when the value of the node and its parent are same, that is the root of that subset of nodes.*

Edge 0-1:

=> Find the subsets in which vertices 0 and 1 are.

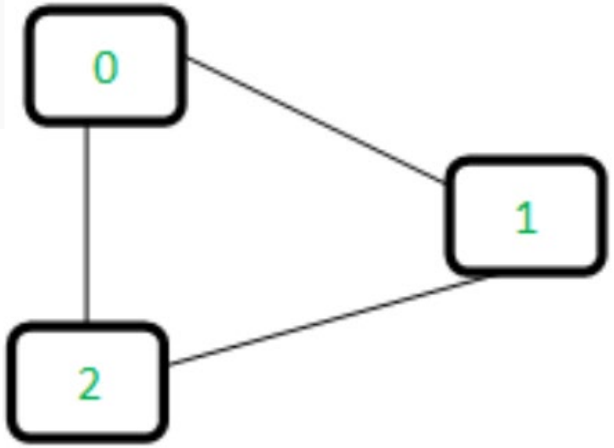
=> 0 and 1 belongs to subset 0 and 1.

=> Since they are in different subsets, take the union of them.

=> For taking the union, either make node 0 as parent of node 1 or vice-versa.

=> 1 is made parent of 0 (1 is now representative of subset {0, 1})

=> parent[] = {1, 1, 2}

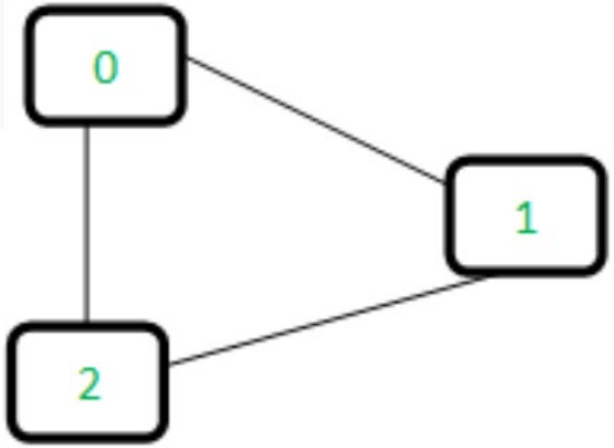


- Initially create a **parent[]** array to keep track of the subsets.
- Traverse through all the edges:
  - Check to which subset each of the nodes belong to by finding the parent[] array till the node and the parent are the same.
  - If the two nodes belong to the same subset then they belong to a cycle.
  - Otherwise, perform union operation on those two subsets.
- If no cycle is found, return false.

Use Union Find Algorithm to detect cycle?

***parent[] = {0, 1, 2}**. Also when the value of the node and its parent are same, that is the root of that subset of nodes.*

parent[] = {1, 1, 2}



Edge 1-2:

=> 1 is in subset 1 and 2 is in subset 2.

=> Since they are in different subsets, take union.

=> Make 2 as parent of 1. (2 is now representative of subset {0, 1, 2})

=> parent[] = {1, 2, 2}

- Initially create a **parent[]** array to keep track of the subsets.
- Traverse through all the edges:
  - Check to which subset each of the nodes belong to by finding the parent[] array till the node and the parent are the same.
  - If the two nodes belong to the same subset then they belong to a cycle.
  - Otherwise, perform union operation on those two subsets.
- If no cycle is found, return false.

Use Union Find Algorithm to detect cycle?

parent[] = {0, 1, 2}*. Also when the value of the node and its parent are same, that is the root of that subset of nodes.*

parent[] = {1, 2, 2}

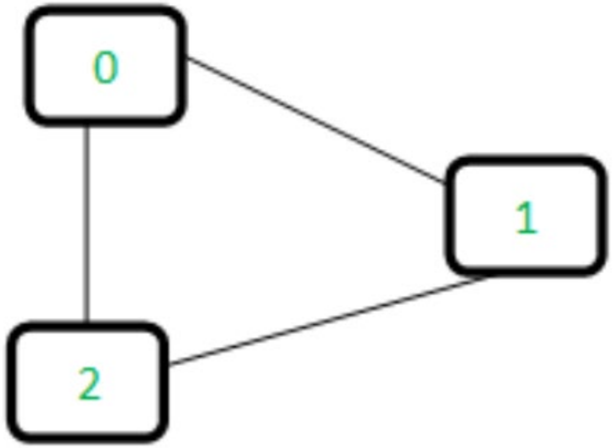
Edge 0-2:

=> 0 is in subset 2 and 2 is also in subset 2.

=> Because 1 is parent of 0 and 2 is parent of 1. So 0 also belongs to subset 2

=> Hence, including this edge forms a cycle.

Therefore, the above graph contains a cycle.



- Initially create a **parent[]** array to keep track of the subsets.
- Traverse through all the edges:
  - Check to which subset each of the nodes belong to by finding the parent[] array till the node and the parent are the same.
  - If the two nodes belong to the same subset then they belong to a cycle.
  - Otherwise, perform union operation on those two subsets.
- If no cycle is found, return false.

Huffman Coding

- Huffman codes compress data very effectively: savings of 20% to 90% are typical, depending on the characteristics of the data being compressed.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101

A character-coding problem

- A data file of 100,000 characters contains only the characters a–f, with the frequencies indicated.
- If we assign each character a 3-bit codeword, we can encode the file in 300,000 bits.

What is a variable length code

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

- A variable-length code can do considerably better than a fixed-length code, by giving frequent characters short codewords and infrequent characters long codewords.
- Here, the 1-bit string 0 represents a, and the 4-bit string 1100 represents f.

$$(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1{,}000 = 224{,}000 \text{ bits}$$

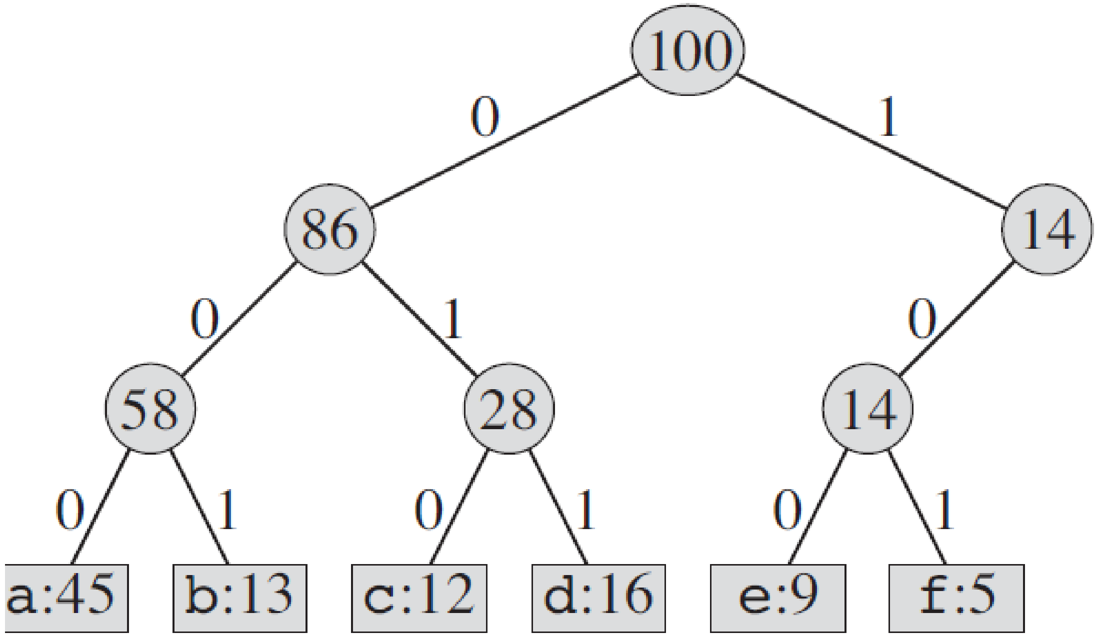
savings of approximately 25%

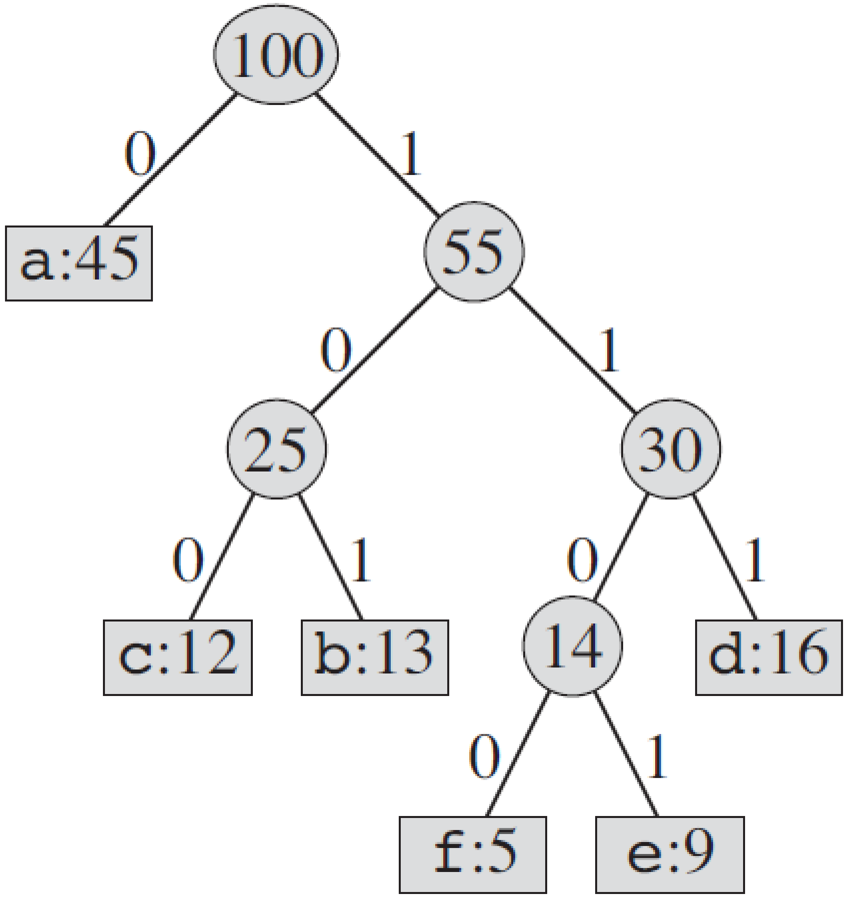
Prefix Code for variable length coding

- We consider here only codes in which no codeword is also a prefix of some other codeword, also called as Prefix Codes.
- Prefix codes are desirable because they simplify decoding.
- Since no codeword is a prefix of any other, the codeword that begins an encoded file is unambiguous.

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Trees corresponding to the coding schemes





	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Constructing a Huffman Code

- Huffman invented a greedy algorithm that constructs an optimal prefix code called a Huffman code.

The alphabet C contains 6 characters, n = 6
5 merge steps build the tree.

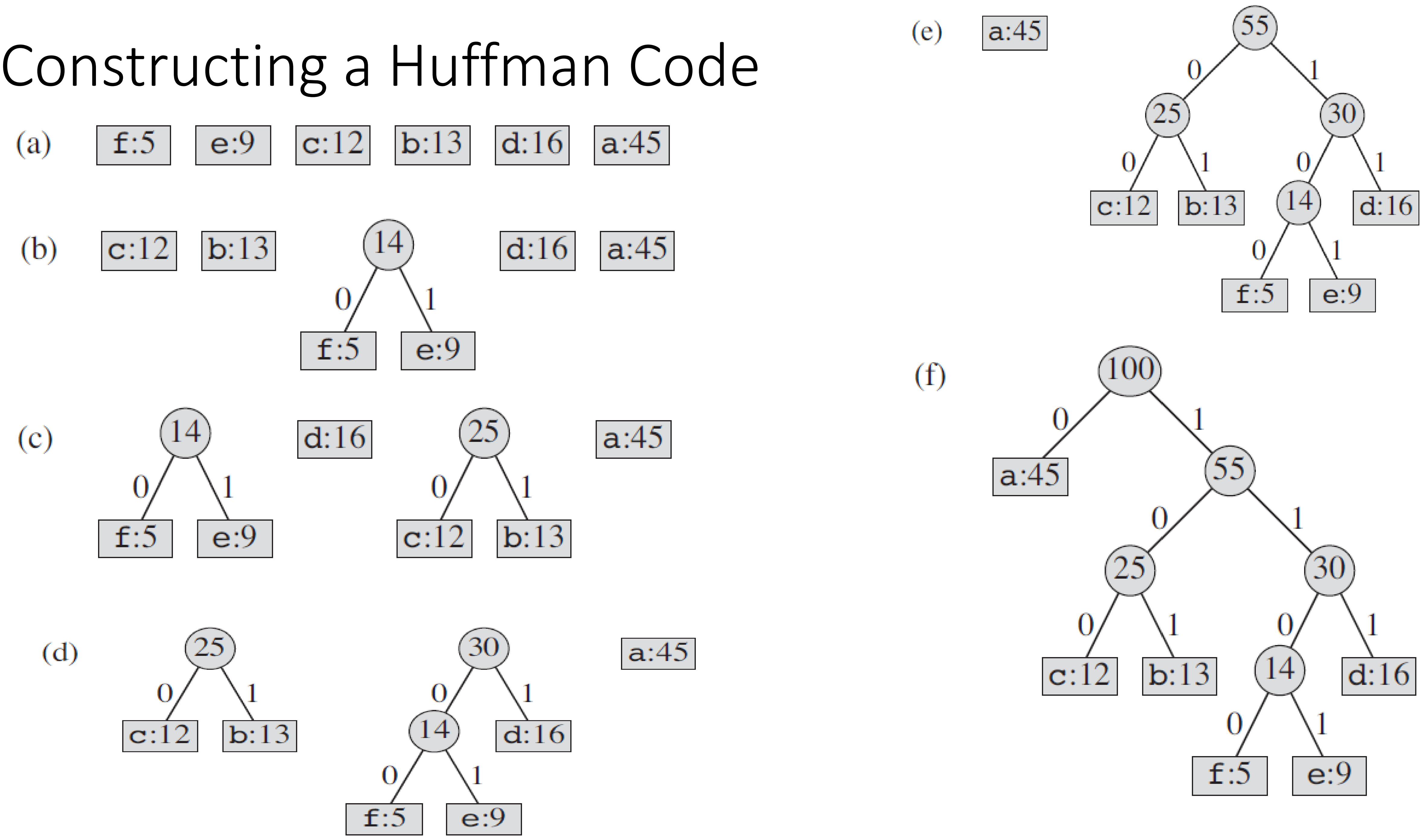
HUFFMAN(C)

1 $n = |C|$
2 $Q = C$
3 **for** $i = 1$ **to** $n - 1$
4   allocate a new node z
5     $z.left = x =$ EXTRACT-MIN(Q)
6     $z.right = y =$ EXTRACT-MIN(Q)
7     $z.freq = x.freq + y.freq$
8   INSERT(Q, z)
9 **return** EXTRACT-MIN(Q)   // return the root of the tree

Line 2 initializes the min-priority queue Q with the characters in C.

Extracts the two nodes x and y of lowest frequency from the queue, replacing them in the queue with a new node Z.

Constructing a Huffman Code



Complexity

- we assume that Q is implemented as a binary min-heap
- For a set C of n characters, we can initialize Q in line 2 in O(n) time using the BUILD-MIN-HEAP procedure

lines 3–8 executes exactly $n - 1$ times

each heap operation requires time $O(\lg n)$

the loop contributes $O(n \lg n)$

HUFFMAN(C)

1 $n = |C|$
2 $Q = C$
3 **for** $i = 1$ **to** $n - 1$
4   allocate a new node z
5     $z.left = x =$ EXTRACT-MIN(Q)
6     $z.right = y =$ EXTRACT-MIN(Q)
7     $z.freq = x.freq + y.freq$
8   INSERT(Q, z)
9 **return** EXTRACT-MIN(Q) // return