Today’s topics

- Minimum spanning tree

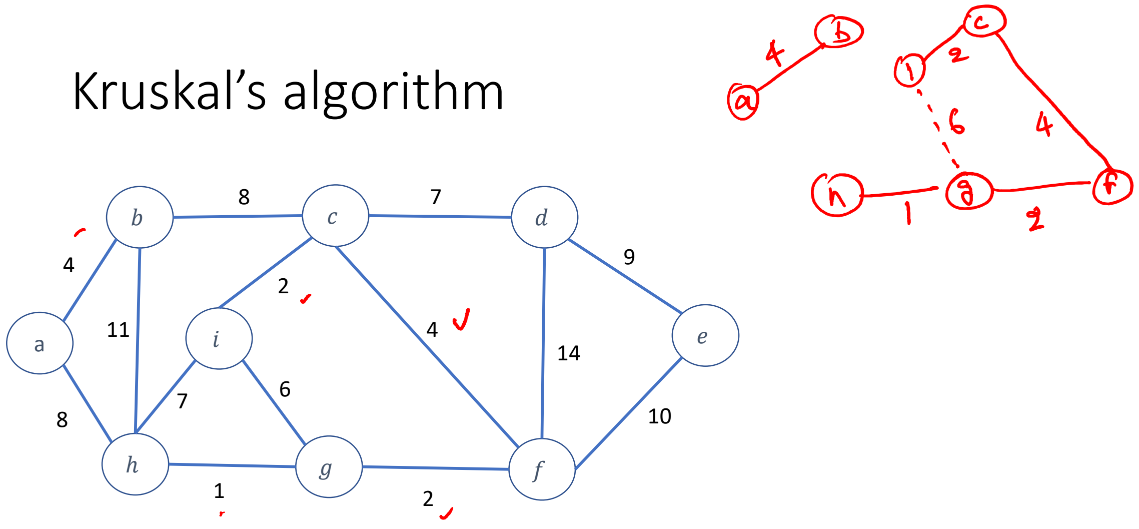
References

- Chapter-7.3 from Goodrich and Tamassia
- Chapter-4.2 from Goodrich and Tamassia
- Chapter-21 from CLRS
- Chapter-19 from CLRS

Minimum spanning tree

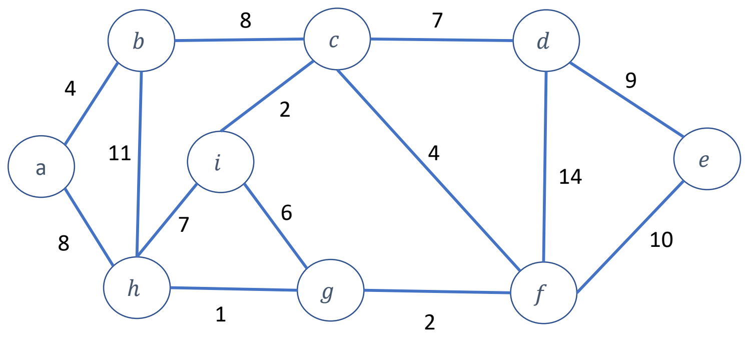
- Every connected undirected graph has a spanning tree
- The weight of a spanning tree is the sum of the weights of all the edges
- Out of all possible spanning trees for a connected undirected graph, the one with the minimum weight is the minimum spanning tree

Kruskal's algorithm



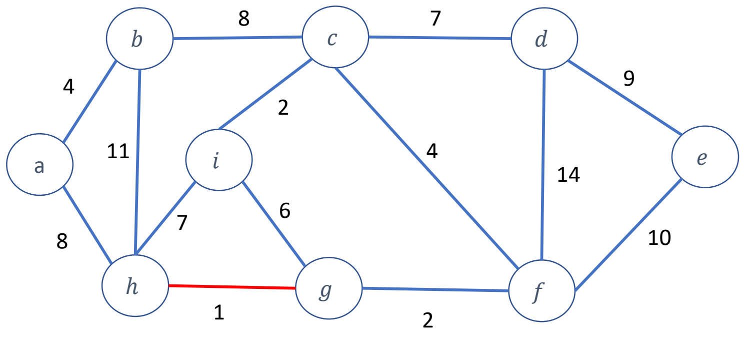
Kruskal's algorithm computes an MST T from graph G. Initially, T contains all the vertices of G without any edges. Afterward, we remove an edge with the minimum weight and add it to T if it doesn't create a cycle. We can ignore the edge if it creates a cycle. We keep removing and adding edges until |V|-1 edges are added to T. After the algorithm terminates, T is actually an MST. One challenging part of this algorithm is to find out whether adding an edge creates a cycle. We will discuss this next.

Kruskal’s algorithm

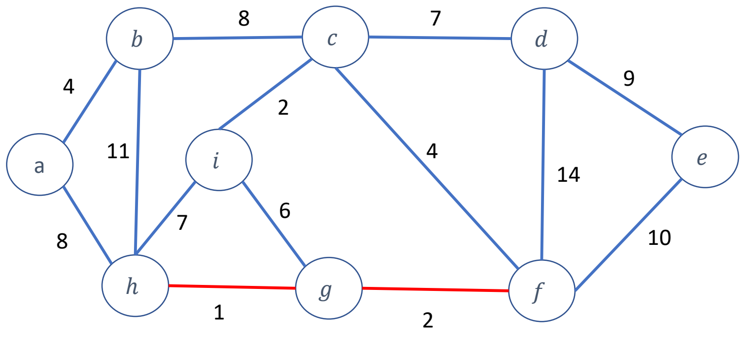


Kruskal’s algorithm

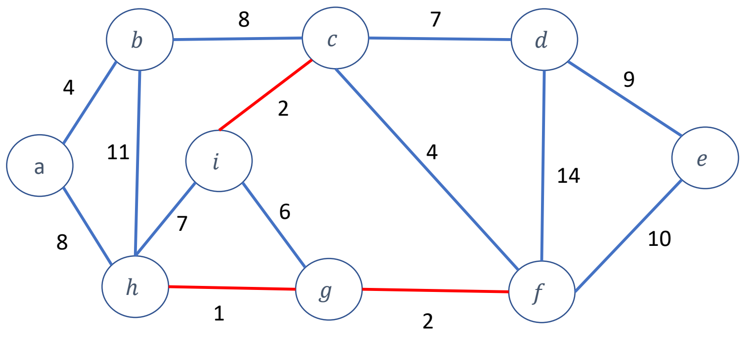
Red edges are part of a MST.



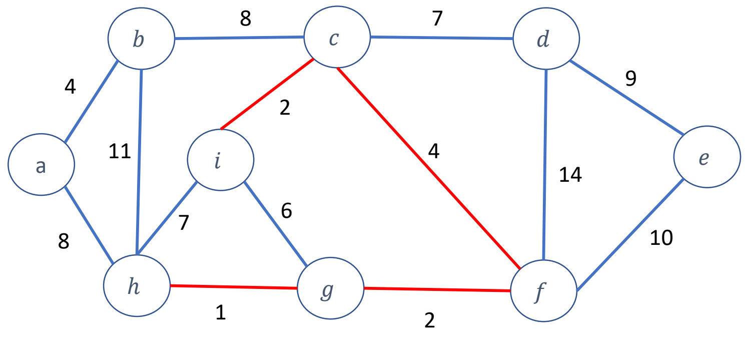
Kruskal’s algorithm



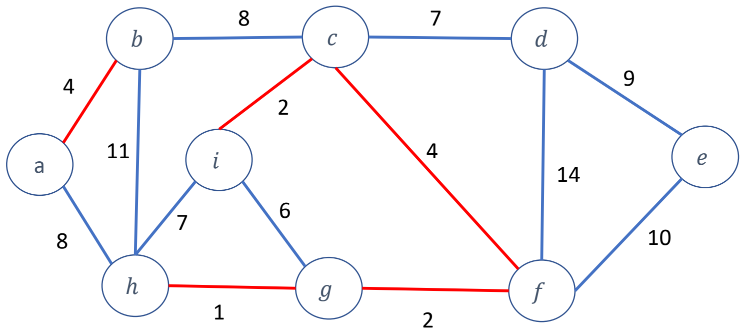
Kruskal’s algorithm



Kruskal’s algorithm

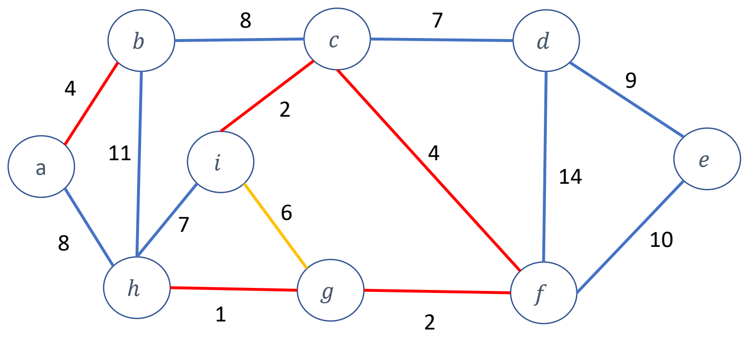


Kruskal’s algorithm

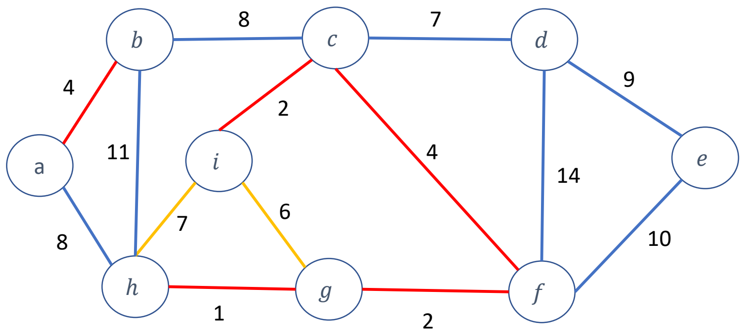


Kruskal's algorithm

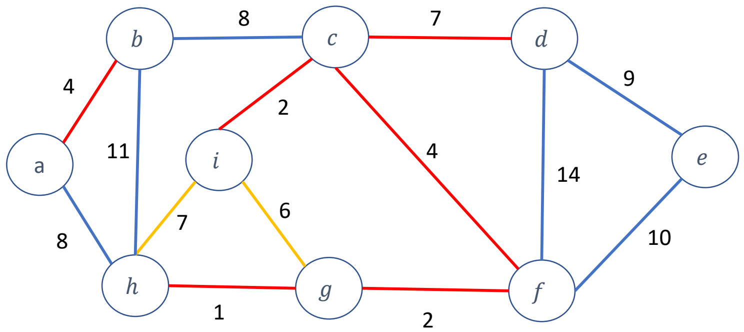
Yellow edges are ignored.



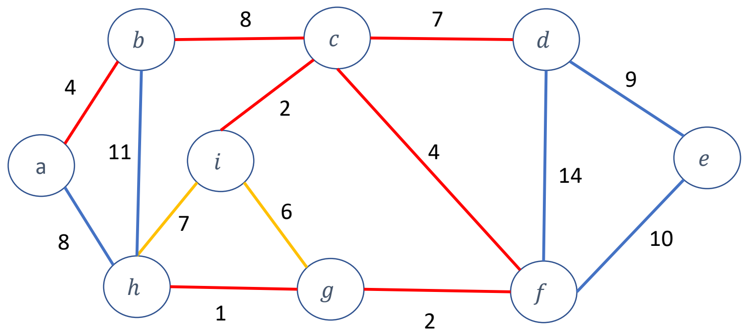
Kruskal's algorithm



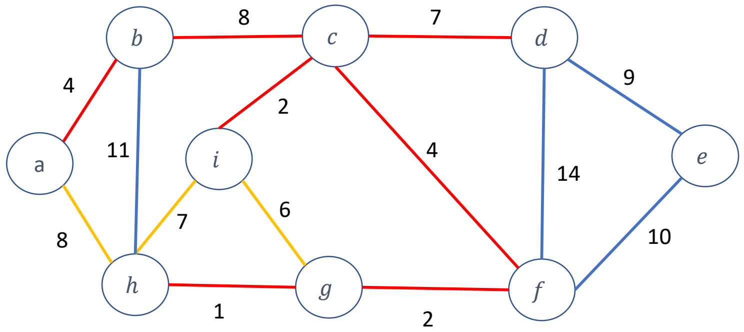
Kruskal's algorithm



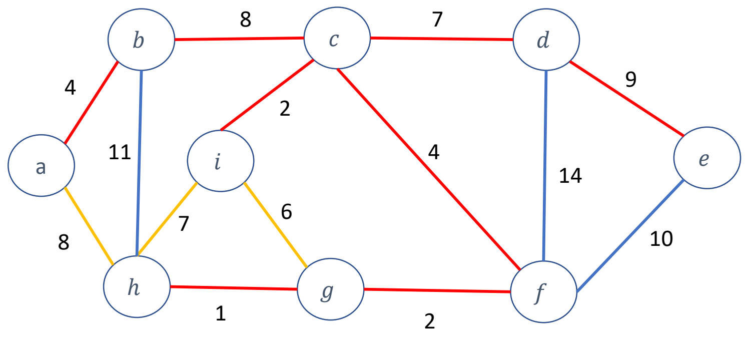
Kruskal’s algorithm



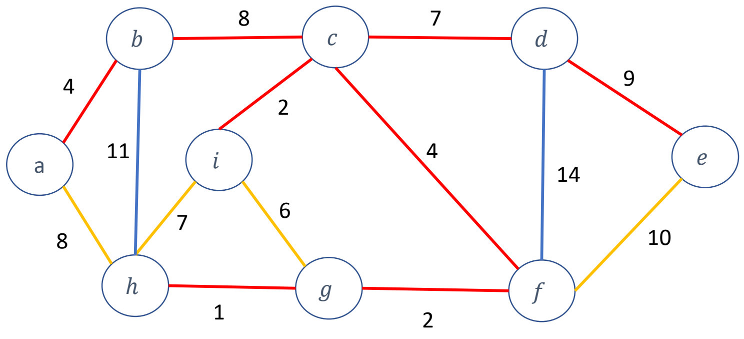
Kruskal’s algorithm



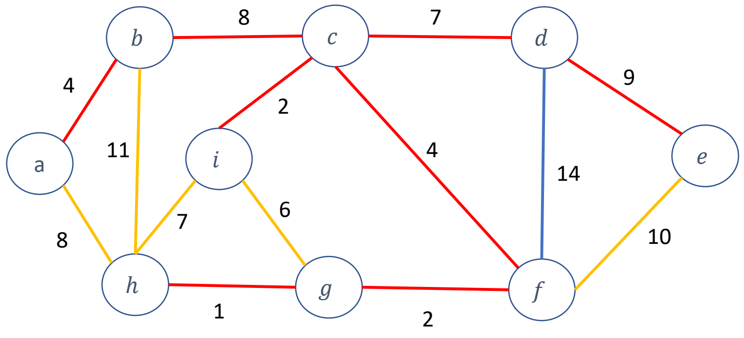
Kruskal’s algorithm



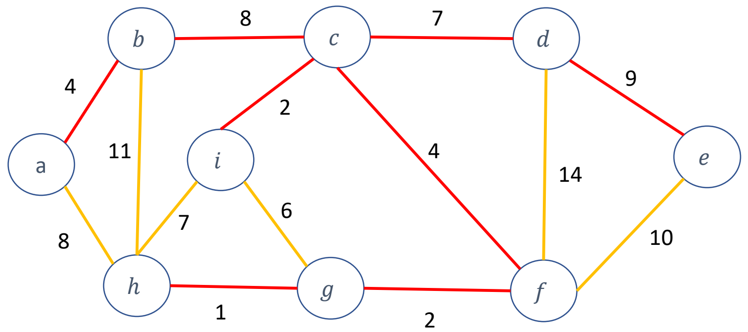
Kruskal’s algorithm



Kruskal’s algorithm



Kruskal’s algorithm



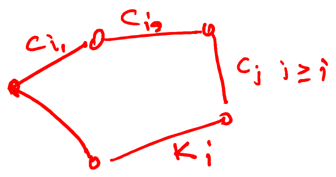
Kruskal’s algorithm

- How can we cheaply find the yellow edges during Kruskal’s algorithm?

Kruskal’s algorithm

- How can we cheaply find the yellow edges during Kruskal’s algorithm?
    - We will discuss some algorithms to do this in a while

Kruskal's algorithm



- Why does Kruskal's algorithm work?

$k_1 \; k_2 \; k_3 \ldots k_i \ldots k_{n-1}$

$\boxed{c_1 \; c_2 \; c_3 \ldots c_i \ldots c_{n-1}}$ $\quad k_i$

$c_i < k_i$

$c_i > k_i$

Kruskal’s algorithm

- Why does Kruskal’s algorithm work?

Let’s say all the edges are of distinct weights
Edges returned by Kruskal are
$k_1, k_2, k_3, \ldots, k_i, \ldots, k_{n-1}$, where for all 1 <= j <= n-2, $k_j < k_{j+1}$
Edges in the correct MST are
$c_1, c_2, c_3, \ldots, c_i, \ldots, c_{n-1}$, where for all 1 <= j <= n-2, $c_j < c_{j+1}$

Let’s say (k_i, c_i) is the first point where the edges differ
Is it possible that weight(c_i) < weight(k_i)?
Why didn’t Kruskal pick c_i instead of k_i?

Kruskal’s algorithm

- Why does Kruskal’s algorithm work?

Let’s say all the edges are of distinct weights
Edges returned by Kruskal are
$k_1, k_2, k_3, \ldots, k_i, \ldots, k_{n-1}$, where for all 1 <= j <= n-2, $k_j < k_{j+1}$
Edges in the correct MST are
$c_1, c_2, c_3, \ldots, c_i, \ldots, c_{n-1}$, where for all 1 <= j <= n-2, $c_j < c_{j+1}$

Let’s say (k_i, c_i) is the first point where the edges differ
Is it possible that weight(c_i) < weight(k_i)?
Why didn’t Kruskal pick c_i instead of k_i?

The only reason why Kruskal didn’t pick c_i is that c_i was part of a cycle containing a subset of edges from the set (k_1, k_2, …, k_{i-1}). Notice that all k-edges before k_i are the same as corresponding c-edges. Therefore, c_i is not part of an MST.

Kruskal’s algorithm

- Why does Kruskal’s algorithm work?

Let’s say all the edges are of distinct weights
Edges returned by Kruskal are
$k_1, k_2, k_3, \dots, k_i, \dots, k_{n-1}$ where for all 1 <= j <= n-2, $k_j < k_{j+1}$
Edges in the correct MST are
$c_1, c_2, c_3, \dots, c_i, \dots, c_{n-1}$, where for all 1 <= j <= n-2, $c_j < c_{j+1}$

Let’s say (k_i, c_i) is the first point where the edges differ
Is it possible that weight(c_i) > weight(k_i)?

Kruskal’s algorithm

- Why does Kruskal’s algorithm work?

Let’s say all the edges are of distinct weights
Edges returned by Kruskal are
$k_1, k_2, k_3, \dots, k_i, \dots, k_{n-1}$ where for all $1 <= j <= n-2$, $k_j < k_{j+1}$
Edges in the correct MST are
$c_1, c_2, c_3, \dots, c_i, \dots, c_{n-1}$, where for all $1 <= j <= n-2$, $c_j < c_{j+1}$

Let’s say (k_i, c_i) is the first point where the edges differ
Is it possible that $weight(c_i) > weight(k_i)$?

If this is the case, then k_i must be part of a cycle. Notice that edges from the set $(k_1, k_2, \dots, k_i)$ don’t create a cycle. Therefore, the cycle that contains k_i must have at least one of the edges from c_i to c_{n-1}. But because all edges from c_i to c_{n-1} have more weight than k_i; therefore, removing k_i from the cycle gives an incorrect MST.

Data structures for disjoint sets

Disjoint-set

- A disjoint-set data structure maintains a collection, S = {S_1, S_2, …, S_k}, of disjoint dynamic sets
- Disjoint-set ADT
  - MAKE_SET(x): creates a new set with just one element x and return the set
  - UNION(A, B): compute and return set $R \leftarrow A \cup B$
  - FIND_SET(x): returns the set that contains element x

Disjoint-set

- Disjoint-set ADT
  - MAKE_SET(x): creates a new set with just one element x and return the set
  - UNION(A, B): compute and return set $R \leftarrow A \cup B$
  - FIND_SET(x): returns the set that contains element x

- Create a set (5, 11, 12, 10, 8) using the above APIs

S_1 = Make_set(5)
S_2 = Make_Set(11)
S_3 = Union(S_1, S_2)

S_4 = Make_Set(12)
S_5 = Union(S_3, S_4)
S_6 = Make_set(10)
S_7 = Union(S_5, S_6)

S_8 = Make_Set(8)
S_9 = Union(S_7, S_8)

Disjoint-set

- Disjoint-set ADT
  - MAKE_SET(x): creates a new set with just one element x and return the set
  - UNION(A, B): compute and return set $R \leftarrow A \cup B$
  - FIND_SET(x): returns the set that contains element x

- Create a set (5, 11, 12, 10, 8) using the above APIs
  - S1 = MAKE_SET(5)
  - S2 = MAKE_SET(11)
  - S3 = UNION(S1, S2)
  - S4 = UNION(S3, MAKE_SET(12))
  - S5 = UNION(S4, MAKE_SET(10))
  - S6 = UNION(S5, MAKE_SET(8))

Disjoint-set

(5, 11, 12, 10, 8)

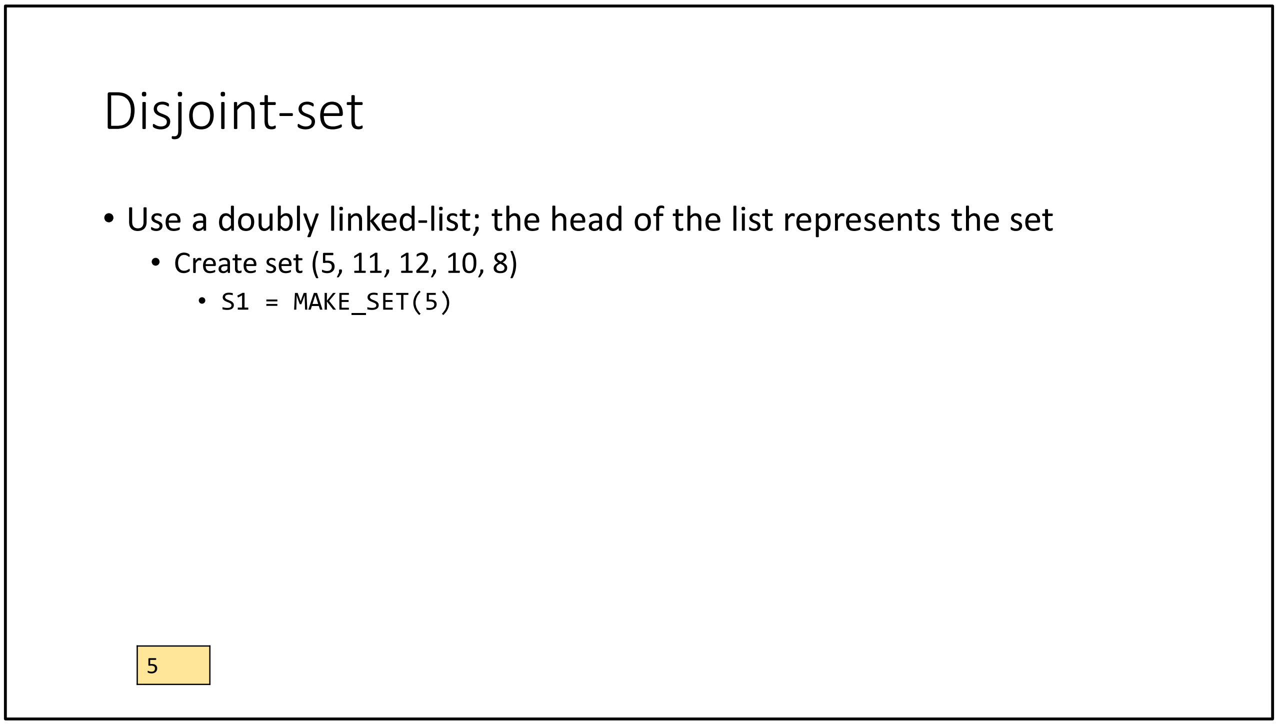
$$\begin{tikzcd} \boxed{5} \arrow[r, leftrightarrow] & \boxed{11} \arrow[r, leftrightarrow] & \boxed{10} \arrow[r, leftrightarrow] & \boxed{8} \end{tikzcd}$$

Disjoint-set

- Use a doubly linked-list; the head of the list represents the set
    - Create set (5, 11, 12, 10, 8)

Disjoint-set

- Use a doubly linked-list; the head of the list represents the set
  - Create set (5, 11, 12, 10, 8)



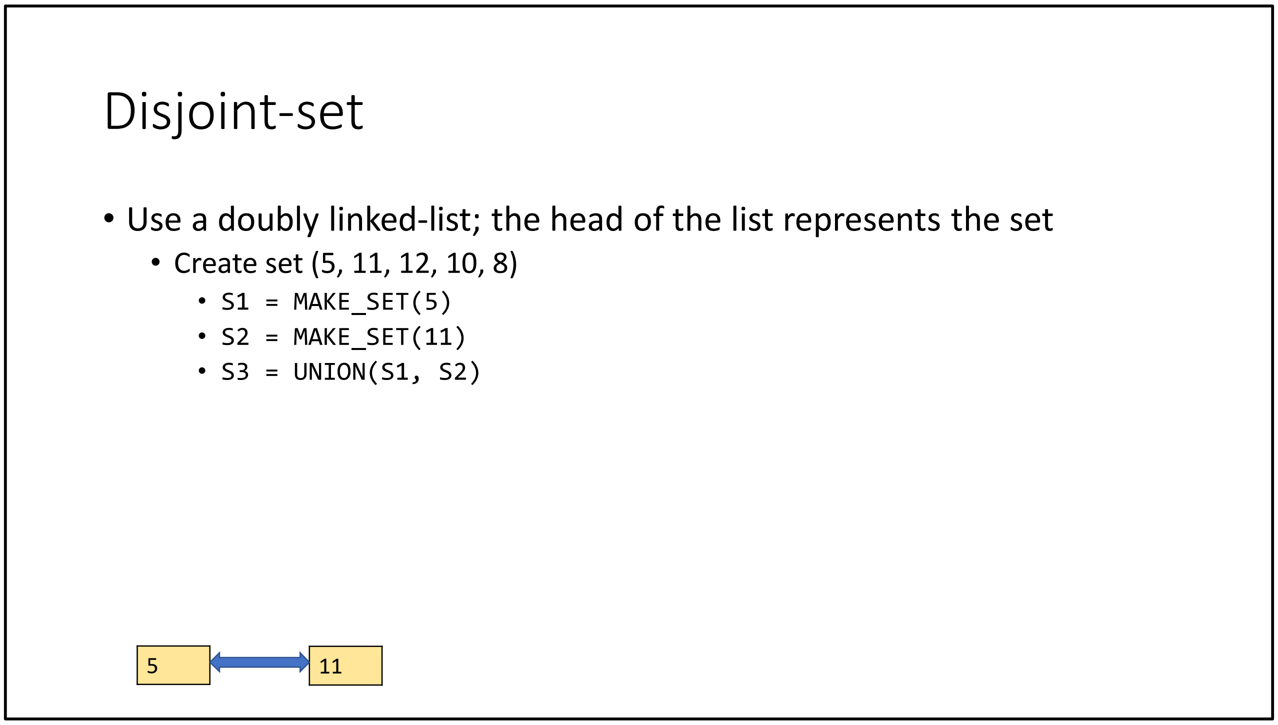
Disjoint-set

- Use a doubly linked-list; the head of the list represents the set
  - Create set (5, 11, 12, 10, 8)
    - `S1 = MAKE_SET(5)`

MAKE_SET creates a doubly linked list with just one node corresponding to the input element.

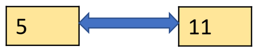
Disjoint-set

- Use a doubly linked-list; the head of the list represents the set
  - Create set (5, 11, 12, 10, 8)
    - `S1 = MAKE_SET(5)`
    - `S2 = MAKE_SET(11)`



Disjoint-set

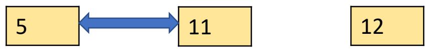
- Use a doubly linked-list; the head of the list represents the set
  - Create set (5, 11, 12, 10, 8)
    - `S1 = MAKE_SET(5)`
    - `S2 = MAKE_SET(11)`
    - `S3 = UNION(S1, S2)`



A set is identified using the head of the linked list. The union operation takes two linked lists representing two disjoint sets and appends the second linked list at the rear end of the first linked list.

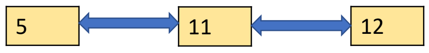
Disjoint-set

- Use a doubly linked-list; the head of the list represents the set
  - Create set (5, 11, 12, 10, 8)
    - `S1 = MAKE_SET(5)`
    - `S2 = MAKE_SET(11)`
    - `S3 = UNION(S1, S2)`
    - `S4 = MAKE_SET(12)`



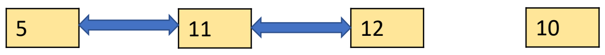
Disjoint-set

- Use a doubly linked-list; the head of the list represents the set
  - Create set (5, 11, 12, 10, 8)
    - `S1 = MAKE_SET(5)`
    - `S2 = MAKE_SET(11)`
    - `S3 = UNION(S1, S2)`
    - `S4 = MAKE_SET(12)`
    - `S5 = UNION(S3, S4)`



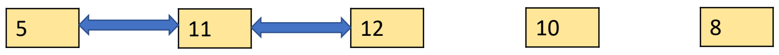
Disjoint-set

- Use a doubly linked-list; the head of the list represents the set
  - Create set (5, 11, 12, 10, 8)
    - `S1 = MAKE_SET(5)`
    - `S2 = MAKE_SET(11)`
    - `S3 = UNION(S1, S2)`
    - `S4 = MAKE_SET(12)`
    - `S5 = UNION(S3, S4)`
    - `S6 = MAKE_SET(10)`



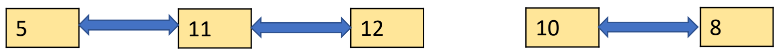
Disjoint-set

- Use a doubly linked-list; the head of the list represents the set
  - Create set (5, 11, 12, 10, 8)
    - `S1 = MAKE_SET(5)`
    - `S2 = MAKE_SET(11)`
    - `S3 = UNION(S1, S2)`
    - `S4 = MAKE_SET(12)`
    - `S5 = UNION(S3, S4)`
    - `S6 = MAKE_SET(10)`
    - `S7 = MAKE_SET(8)`



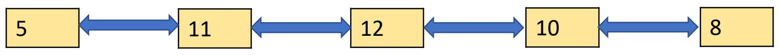
Disjoint-set

- Use a doubly linked-list; the head of the list represents the set
  - Create set (5, 11, 12, 10, 8)
    - `S1 = MAKE_SET(5)`
    - `S2 = MAKE_SET(11)`
    - `S3 = UNION(S1, S2)`
    - `S4 = MAKE_SET(12)`
    - `S5 = UNION(S3, S4)`
    - `S6 = MAKE_SET(10)`
    - `S7 = MAKE_SET(8)`
    - `S8 = UNION(S6, S7)`



Disjoint-set

- Use a doubly linked-list; the head of the list represents the set
  - Create set (5, 11, 12, 10, 8)
    - `S1 = MAKE_SET(5)`
    - `S2 = MAKE_SET(11)`
    - `S3 = UNION(S1, S2)`
    - `S4 = MAKE_SET(12)`
    - `S5 = UNION(S3, S4)`
    - `S6 = MAKE_SET(10)`
    - `S7 = MAKE_SET(8)`
    - `S8 = UNION(S6, S7)`
    - `S9 = UNION(S7, S8)`



Disjoint-set

- Storing an identifier to locate the corresponding set in each element of the set
  - Time complexity:
    - `MAKE_SET(x)` $O(1)$
    - `UNION(A, B)` $O(1)$
    - `FIND_SET(x)` $O(n)$

The find operation takes the address of an element in the linked list and returns the corresponding set by finding the head of the linked list. The time complexity of the FIND_SET is thus O(n).

Disjoint-set

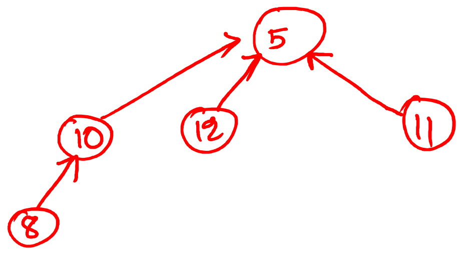
- How can we improve the time complexity of find?

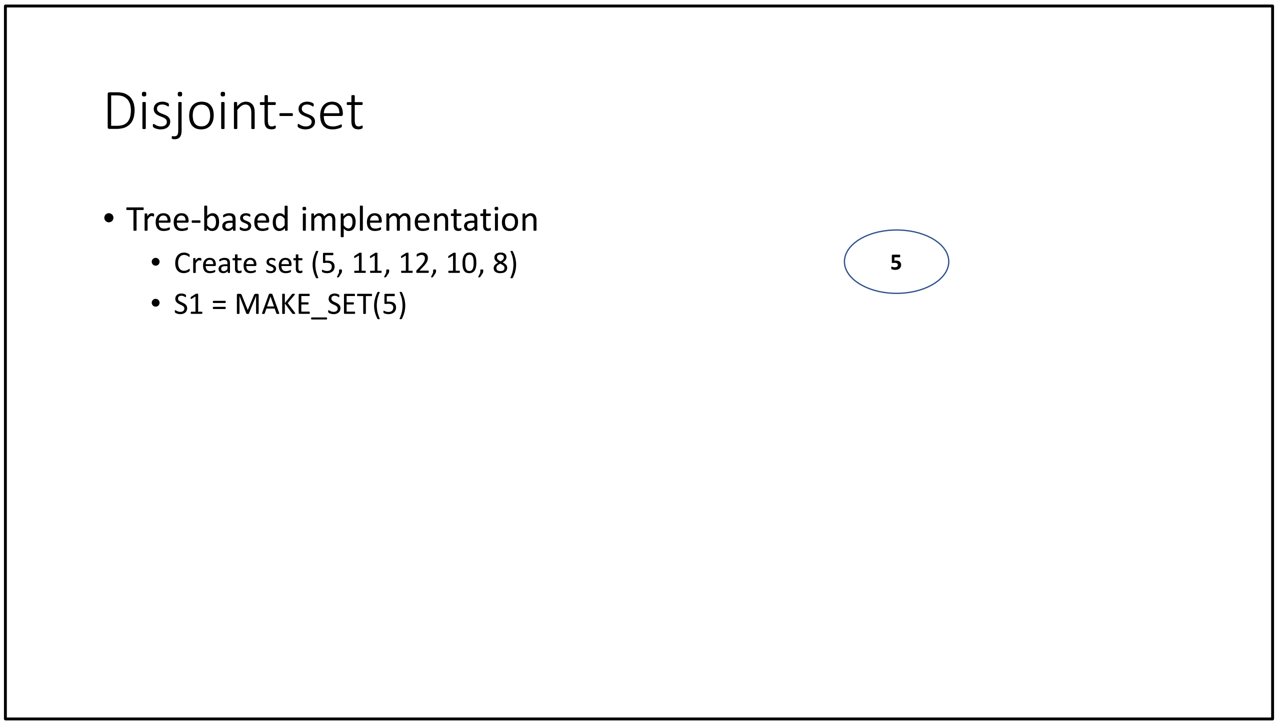
Disjoint-set

- A tree-based implementation in which the root of the tree is the identifier of the set, and the tree nodes are the elements in the set
- There is one tree for every set

Disjoint-set

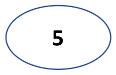
- Tree-based implementation
  - Create set (5, 11, 12, 10, 8)


$$\begin{tikzcd}
 & & \boxed{5} & \\
 & \boxed{10} \arrow[ur] & \boxed{12} \arrow[u] & \boxed{11} \arrow[ul] \\
\boxed{8} \arrow[ur] & & &
\end{tikzcd}$$



Disjoint-set

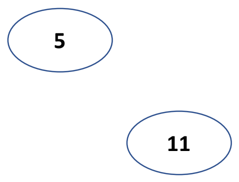
- Tree-based implementation
  - Create set (5, 11, 12, 10, 8)
  - S1 = MAKE_SET(5)



MAKE_SET simply creates a new tree with just one node corresponding to the input element.

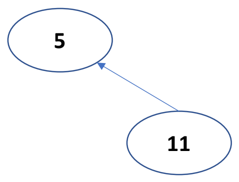
Disjoint-set

- Tree-based implementation
    - Create set (5, 11, 12, 10, 8)
    - S1 = MAKE_SET(5)
    - S2 = MAKE_SET(11)



Disjoint-set

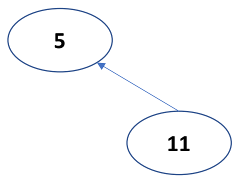
- Tree-based implementation
  - Create set (5, 11, 12, 10, 8)
  - S1 = MAKE_SET(5)
  - S2 = MAKE_SET(11)
  - S3 = UNION(S1, S2)



To unify two trees, make the root of one tree the parent of the root of the other tree.

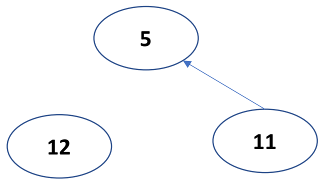
Disjoint-set

- Tree-based implementation
  - Create set (5, 11, 12, 10, 8)
  - S1 = MAKE_SET(5)
  - S2 = MAKE_SET(11)
  - S3 = UNION(S1, S2)
  - S4 = MAKE_SET(12)



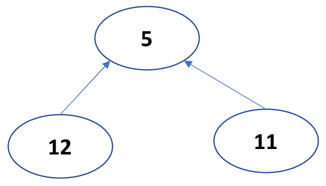
Disjoint-set

- Tree-based implementation
    - Create set (5, 11, 12, 10, 8)
    - S1 = MAKE_SET(5)
    - S2 = MAKE_SET(11)
    - S3 = UNION(S1, S2)
    - S4 = MAKE_SET(12)



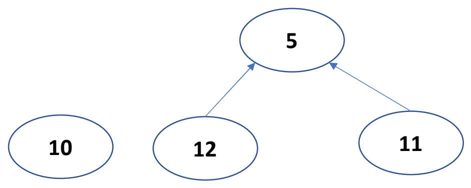
Disjoint-set

- Tree-based implementation
  - Create set (5, 11, 12, 10, 8)
  - S1 = MAKE_SET(5)
  - S2 = MAKE_SET(11)
  - S3 = UNION(S1, S2)
  - S4 = MAKE_SET(12)
  - S5 = UNION(S3, S4)



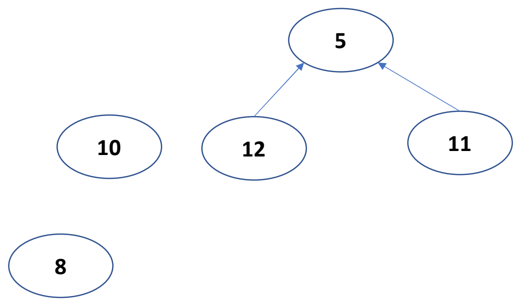
Disjoint-set

- Tree-based implementation
  - Create set (5, 11, 12, 10, 8)
  - S1 = MAKE_SET(5)
  - S2 = MAKE_SET(11)
  - S3 = UNION(S1, S2)
  - S4 = MAKE_SET(12)
  - S5 = UNION(S3, S4)
  - S6 = MAKE_SET(10)



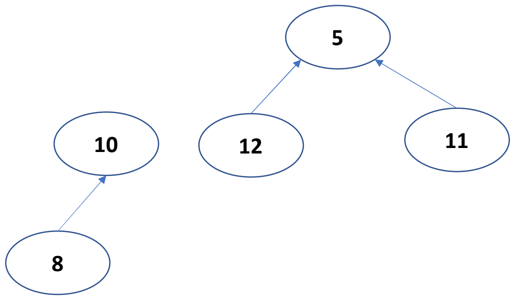
Disjoint-set

- Tree-based implementation
  - Create set (5, 11, 12, 10, 8)
  - S1 = MAKE_SET(5)
  - S2 = MAKE_SET(11)
  - S3 = UNION(S1, S2)
  - S4 = MAKE_SET(12)
  - S5 = UNION(S3, S4)
  - S6 = MAKE_SET(10)
  - S7 = MAKE_SET(8)



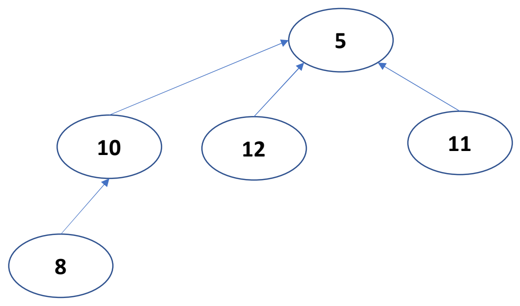
Disjoint-set

- Tree-based implementation
  - Create set (5, 11, 12, 10, 8)
  - S1 = MAKE_SET(5)
  - S2 = MAKE_SET(11)
  - S3 = UNION(S1, S2)
  - S4 = MAKE_SET(12)
  - S5 = UNION(S3, S4)
  - S6 = MAKE_SET(10)
  - S7 = MAKE_SET(8)
  - S8 = UNION(S6, S7)



Disjoint-set

- Tree-based implementation
  - Create set (5, 11, 12, 10, 8)
  - S1 = MAKE_SET(5)
  - S2 = MAKE_SET(11)
  - S3 = UNION(S1, S2)
  - S4 = MAKE_SET(12)
  - S5 = UNION(S3, S4)
  - S6 = MAKE_SET(10)
  - S7 = MAKE_SET(8)
  - S8 = UNION(S6, S7)
  - S9 = UNION(S5, S8)

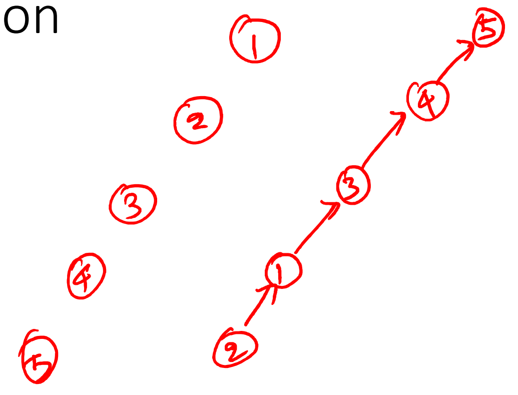


Disjoint-set

- Tree-based implementation
  - Time complexity
    - MAKE_SET(x) $O(1)$
    - UNION(A, B) $O(1)$
    - FIND_SET(x)

Tree-based implementation

- Worst-case
  - S1 = MAKE_SET(1)
  - S2 = MAKE_SET(2)
  - S3 = MAKE_SET(3)
  - S4 = MAKE_SET(4)
  - S5 = MAKE_SET(5)
  - S6 = UNION(S1, S2)
  - S7 = UNION(S3, S6)
  - S8 = UNION(S4, S7)
  - S9 = UNION(S5, S8)



Disjoint-set

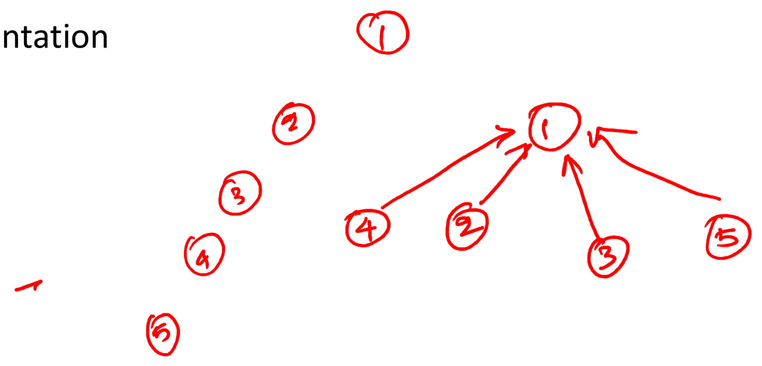
- Tree-based implementation
  - Optimizations

Disjoint-set

- Tree-based implementation
  - Optimizations
    - During union, always make the root of the tree with the larger height the parent of the root of the tree with the smaller height

Tree-based implementation

- Optimized implementation
  - S1 = MAKE_SET(1)
  - S2 = MAKE_SET(2)
  - S3 = MAKE_SET(3)
  - S4 = MAKE_SET(4)
  - S5 = MAKE_SET(5)
  - S6 = UNION(S1, S2)
  - S7 = UNION(S3, S6)
  - S8 = UNION(S4, S7)
  - S9 = UNION(S5, S8)

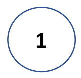


Tree-based implementation

- Optimized implementation

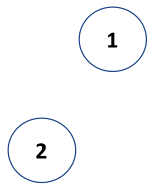
Tree-based implementation

- Optimized implementation
  - S1 = MAKE_SET(1)



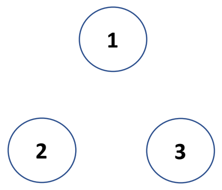
Tree-based implementation

- Optimized implementation
  - S1 = MAKE_SET(1)
  - S2 = MAKE_SET(2)



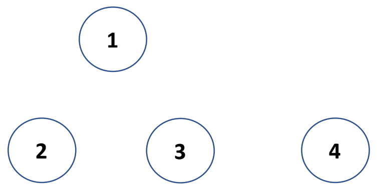
Tree-based implementation

- Optimized implementation
  - S1 = MAKE_SET(1)
  - S2 = MAKE_SET(2)
  - S3 = MAKE_SET(3)



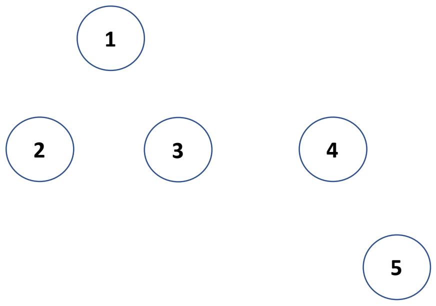
Tree-based implementation

- Optimized implementation
  - S1 = MAKE_SET(1)
  - S2 = MAKE_SET(2)
  - S3 = MAKE_SET(3)
  - S4 = MAKE_SET(4)



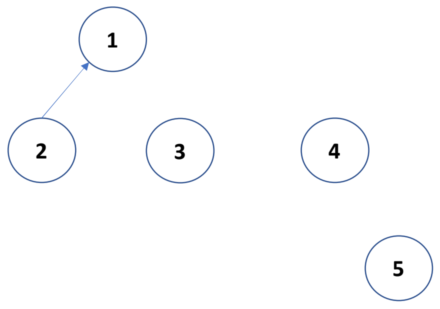
Tree-based implementation

- Optimized implementation
  - S1 = MAKE_SET(1)
  - S2 = MAKE_SET(2)
  - S3 = MAKE_SET(3)
  - S4 = MAKE_SET(4)
  - S5 = MAKE_SET(5)



Tree-based implementation

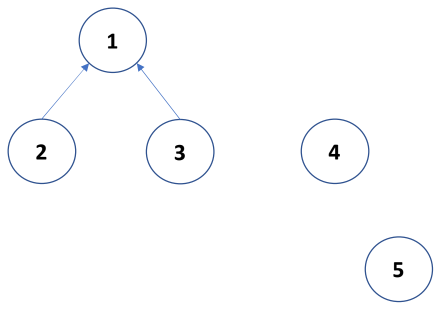
- Optimized implementation
    - S1 = MAKE_SET(1)
    - S2 = MAKE_SET(2)
    - S3 = MAKE_SET(3)
    - S4 = MAKE_SET(4)
    - S5 = MAKE_SET(5)
    - S6 = UNION(S1, S2)



Tree-based implementation

- Optimized implementation
  - S1 = MAKE_SET(1)
  - S2 = MAKE_SET(2)
  - S3 = MAKE_SET(3)
  - S4 = MAKE_SET(4)
  - S5 = MAKE_SET(5)
  - S6 = UNION(S1, S2)
  - S7 = UNION(S3, S6)

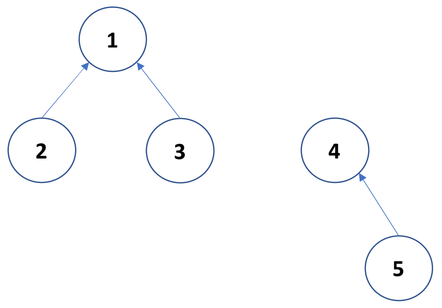
$$\begin{tikzcd} & 1 & & \\ 2 \arrow[ur] & & 3 \arrow[ul] & 4 \\ & & & 5 \end{tikzcd}$$

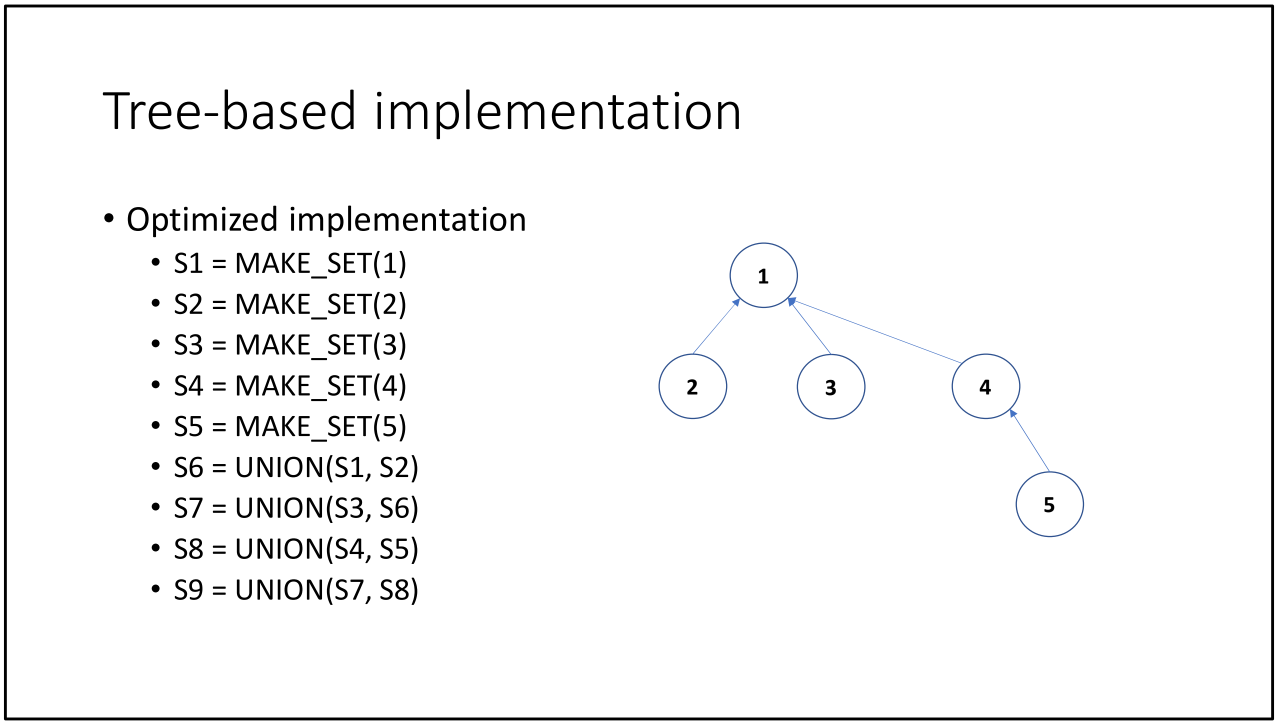


Make the root of the tree of the larger height the parent of the root of the node of the smaller height.

Tree-based implementation

- Optimized implementation
  - S1 = MAKE_SET(1)
  - S2 = MAKE_SET(2)
  - S3 = MAKE_SET(3)
  - S4 = MAKE_SET(4)
  - S5 = MAKE_SET(5)
  - S6 = UNION(S1, S2)
  - S7 = UNION(S3, S6)
  - S8 = UNION(S4, S5)





Tree-based implementation

- Optimized implementation
  - S1 = MAKE_SET(1)
  - S2 = MAKE_SET(2)
  - S3 = MAKE_SET(3)
  - S4 = MAKE_SET(4)
  - S5 = MAKE_SET(5)
  - S6 = UNION(S1, S2)
  - S7 = UNION(S3, S6)
  - S8 = UNION(S4, S5)
  - S9 = UNION(S7, S8)

If the heights of both trees are the same, any tree can be the parent of the other tree. The height of the new tree is one more than the height of the trees before the union.

Disjoint-set

- Tree-based implementation
  - Optimizations
    - The height of a tree is also called the rank of a tree
    - During the union operation, the root of the tree with the larger rank becomes the parent of the root of the smaller tree
    - This is also called union-by-rank

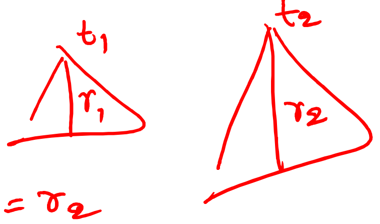
Disjoint-set

- If $n(t)$ is the number of nodes in a tree t of rank r, then $n(t) \geq 2^r$

For tree with rank 0 [illegible] nodes

$n(t) = n(t_1) + n(t_2)$

$n(t_1) \geq 2^{r_1}$ $\quad n(t_2) \geq 2^{r_2}$



$r_1 > r_2$

$r = r_1$

$n(t) \geq 2^{r_1} + 2^{r_2} > 2^r$

$r_1 = r_2$

$r = r_1 + 1$

$n(t) \geq 2^{r_1} + 2^{r_2}$

$\geq 2 \times 2^{r_1}$

$\geq 2^{r_1+1}$

Disjoint-set

- If $n(t)$ is the number of nodes in a tree t of rank r, then $n(t) \geq 2^r$

Disjoint-set

- If $n(t)$ is the number of nodes in a tree t of rank r, then $n(t) \geq 2^r$

The number of elements in a tree of rank 0 is 1; therefore, the condition is true for rank 0

Let's say the condition holds for trees t1 and t2 of rank r1 and r2.

We'll prove that the condition holds for tree t, such that t = UNION(t1, t2)

Disjoint-set

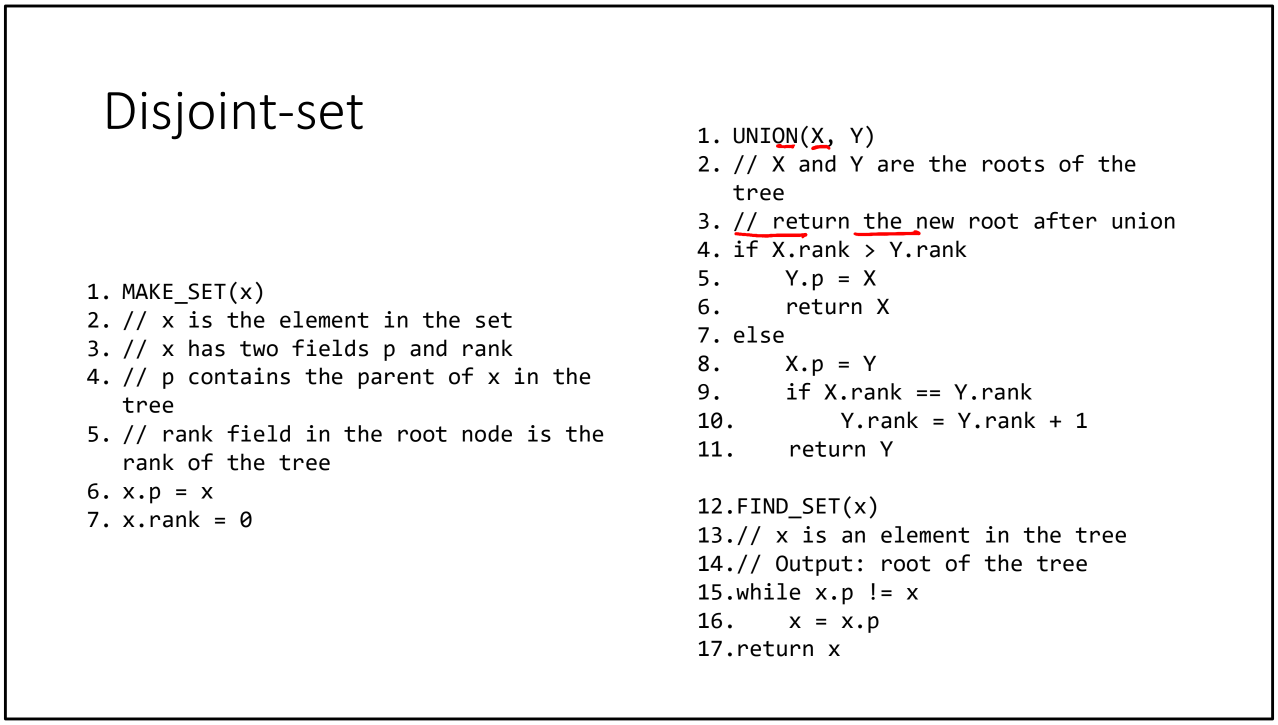
Let’s say r is the rank of t
If r1 > r2, then r = r1
The number of nodes in the new tree is:
n(t) = $n(t1) + n(t2) \geq 2^{r1} + 2^{r2} > 2^{r1} > 2^{r}$
If r1 == r2, then r = r1+1
The number of nodes in the new tree is:
n(t) = $n(t1) + n(t2) \geq 2^{r1} + 2^{r2}$
$\geq 2 * 2^{r1}$ // because r1 = r2
$\geq 2^{r1+1}$
$\geq 2^{r}$
Therefore the condition holds for t.

Disjoint-set

- If n is the number of nodes in a tree representing a disjoint-set, what is the upper bound of the height of the tree

log n

Time coplexity of find = O(log n)



Disjoint-set

```
1. MAKE_SET(x)
2. // x is the element in the set
3. // x has two fields p and rank
4. // p contains the parent of x in the
   tree
5. // rank field in the root node is the
   rank of the tree
6. x.p = x
7. x.rank = 0
```

```
1. UNION(X, Y)
2. // X and Y are the roots of the
   tree
3. // return the new root after union
4. if X.rank > Y.rank
5.     Y.p = X
6.     return X
7. else
8.     X.p = Y
9.     if X.rank == Y.rank
10.        Y.rank = Y.rank + 1
11.    return Y

12.FIND_SET(x)
13.// x is an element in the tree
14.// Output: root of the tree
15.while x.p != x
16.    x = x.p
17.return x
```

If the rank of X is greater than Y, then we make X the parent of Y at line-5. Otherwise, Y becomes the parent of X at line-8. If the ranks of both the tree are the same, then the rank (or height) of the new tree is incremented by one at line-10. In the FIND_SET algorithm, we keep iterating the parent nodes until we find a node that is the parent of itself. This node is the root of the tree and is returned to the caller. The MAKE_SET procedure creates a tree with a single node, also the root. The rank of the root is set to zero at line-7, and the parent of the root is set to itself at line-6.

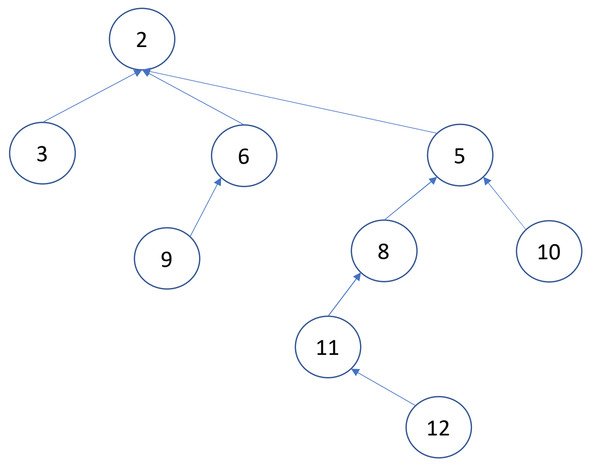
Disjoint-set

- Further optimizations
    - After a find operation, can we make subsequent find operations on the same node cheaper

Disjoint-set

- Further optimizations
  - After a find operation, can we make subsequent find operations on the same node cheaper
- Path compression: update the parent pointer in each node visited during a find operation to point to the root node

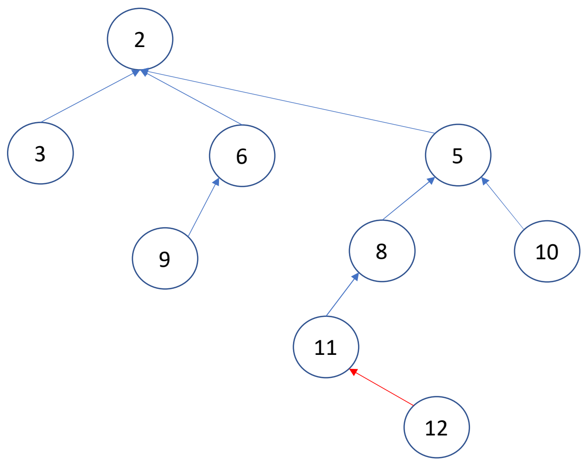
Disjoint set



FIND_SET(12)

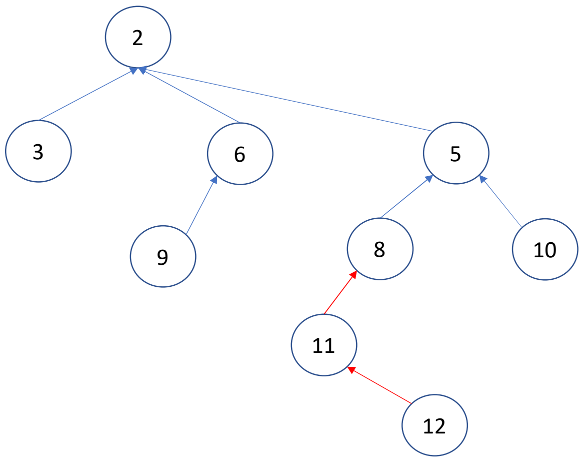
During the FIND_SET operation, we walk from the element to the root using the parent field in the tree nodes. The parent of the root node is set to itself.

Disjoint set



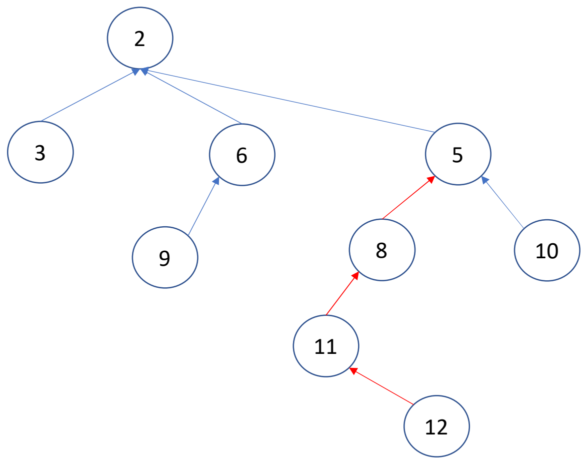
FIND_SET(12)

Disjoint set

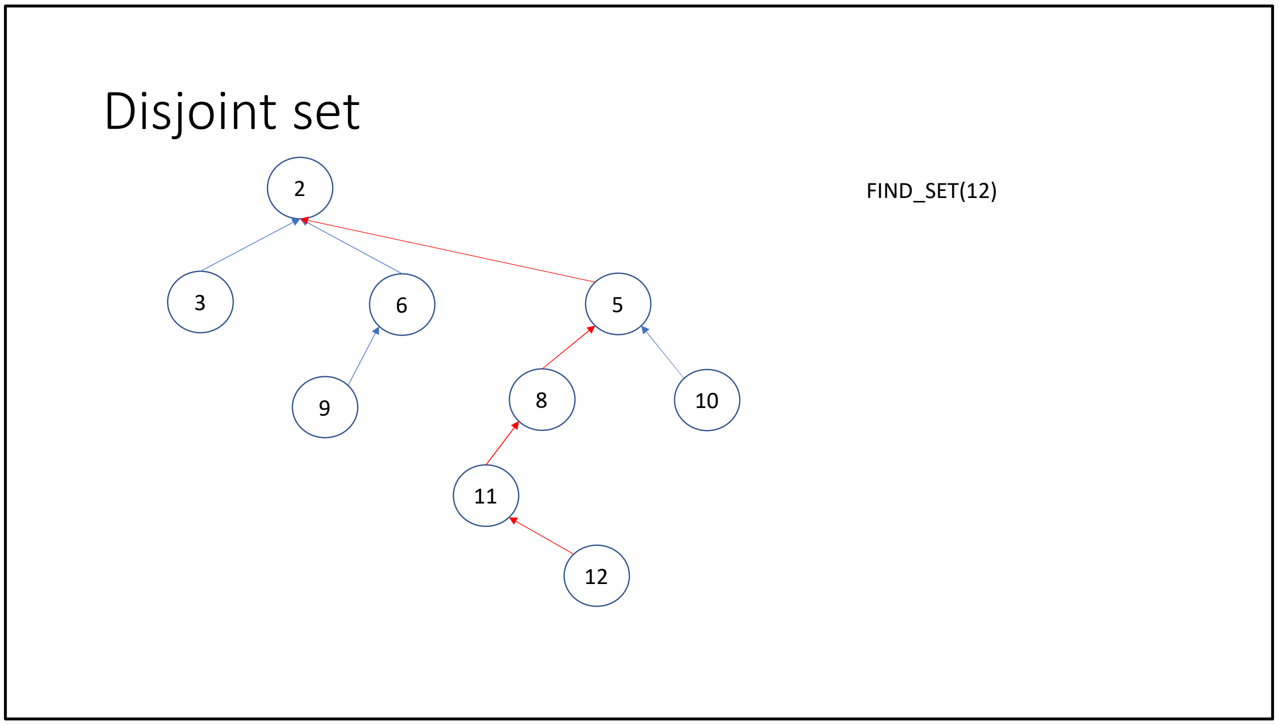


FIND_SET(12)

Disjoint set

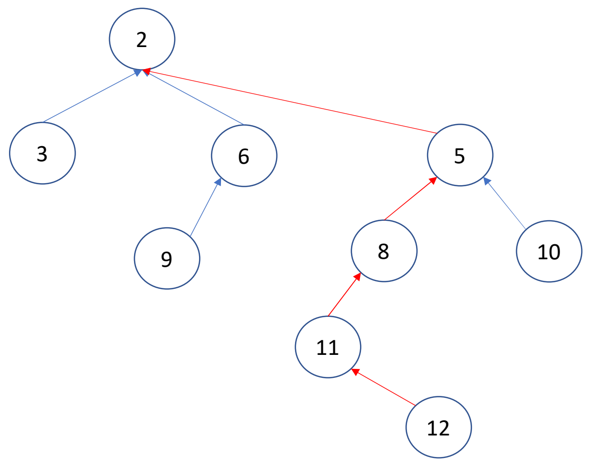


FIND_SET(12)

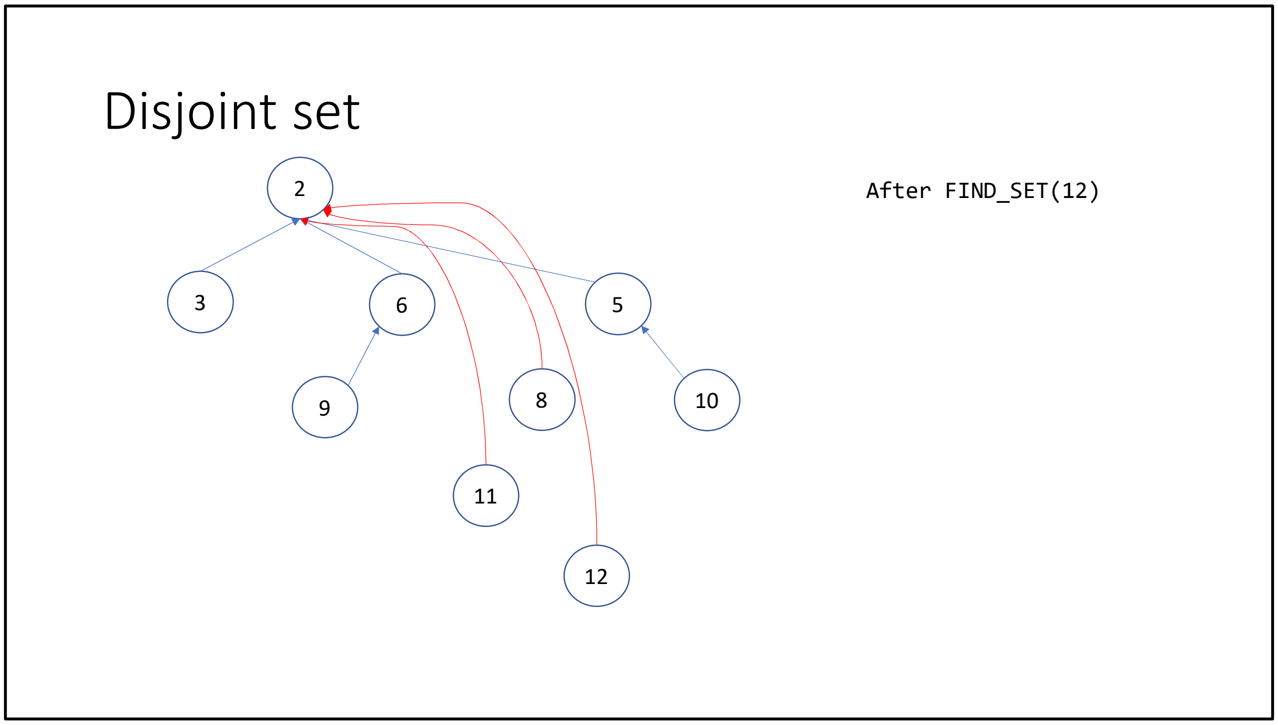


Disjoint set

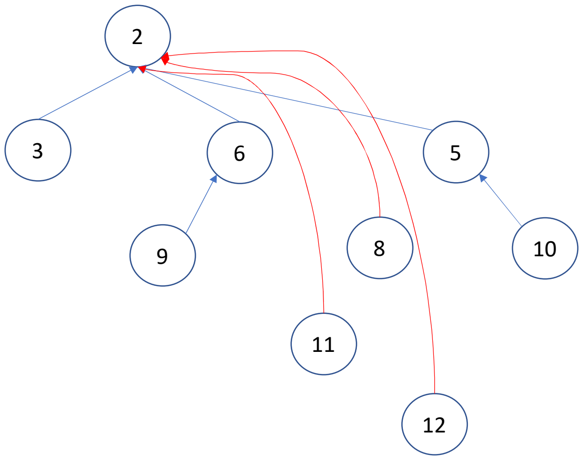
FIND_SET(12)



Nodes 12, 11, 8, 5, 2 lie on the path from 12 to 2. The original FIND_SET walk all these nodes and return the root. In the path-compression algorithm, the parents of nodes 12, 11, and 8 are set to node 2, as shown in the next slide.

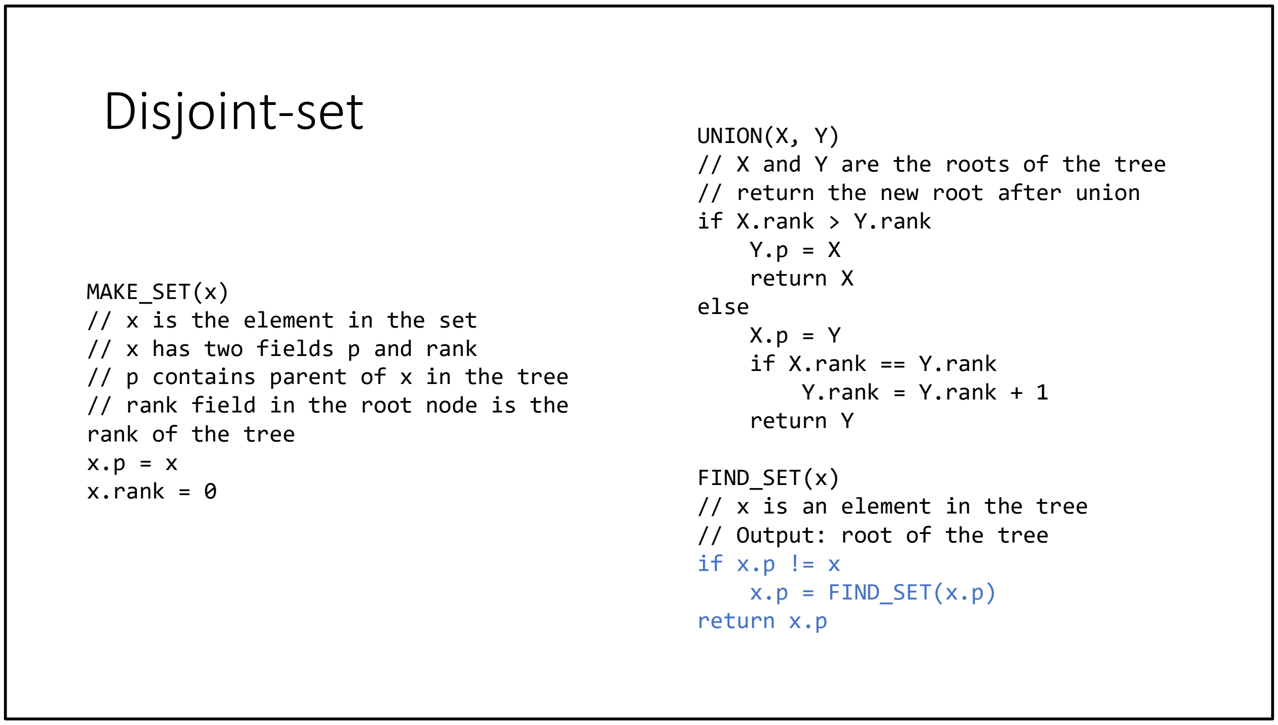


Disjoint set



`After FIND_SET(12)`

Before returning the root of the tree, the parents of all the nodes in the path from 12 to root are set to root.



Disjoint-set

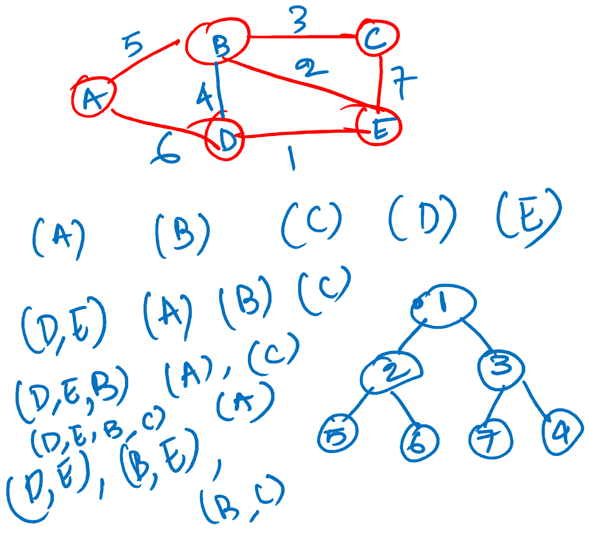
```
MAKE_SET(x)
// x is the element in the set
// x has two fields p and rank
// p contains parent of x in the tree
// rank field in the root node is the
rank of the tree
x.p = x
x.rank = 0
```

```
UNION(X, Y)
// X and Y are the roots of the tree
// return the new root after union
if X.rank > Y.rank
    Y.p = X
    return X
else
    X.p = Y
    if X.rank == Y.rank
        Y.rank = Y.rank + 1
    return Y

FIND_SET(x)
// x is an element in the tree
// Output: root of the tree
if x.p != x
    x.p = FIND_SET(x.p)
return x.p
```

Here, FIND_SET demonstrates the path compression algorithm. FIND_SET returns the root of the tree. While finding the set of an element, the parents of all the nodes from the element to the root are set to the root of the tree. Notice that the path compression algorithm may reduce the height, and if we don’t change the rank, the rank of the tree can be more than the height of the tree. Even though the rank of the tree is greater than or equal to the height of the tree, the find and union operations will work correctly, and the invariant n(t) >= 2^r still holds. Thus even in this case, the height of the tree is O(log (n)) because r is O(log(n)) and the height is less than or equal to r.

Kruskal’s algorithm



```
1. MST_KRUSKAL(G)
2. // G is the graph G = (V, E, w)
3. // Output: a set that contains all
   edges in the MST

4. A = $\phi$
5. Let H be a min heap
6. for each vertex v ∈ G.V
7.     MAKE_SET(v)
8. for each vertex u ∈ $G.V$
9.    for each v in G.AdjList[u]
10.       H.insert(pair(u, v))

11.while |A| != |V| - 1
12.    (u, v) = H.ExtractMin()
13.    set_u = FIND_SET(u)
14.    set_v = FIND_SET(v)
15.    if set_u != set_v
16.        $A = A \cup \{(u, v)\}$
17.        UNION(set_u, set_v)
18.return A
```

Initially, all the vertices are their own disjoint sets. All the edges are stored in a min-heap. An edge is only added to MST if its endpoints belong to different sets (see line-15). We an edge (u, v) is added to the MST; the algorithm unifies the sets corresponding to u and v at line-17. The unification ensures that all the vertices within a disjoint-set are connected, and therefore if an edge whose both endpoints belong to the same set always creates a cycle and thus is not added to the MST.

Kruskal’s algorithm

Time complexity

```
MST_KRUSKAL(G)
// G is the graph G = (V, E, w)
// Output: a set that contains all
edges in the MST

A = $\phi$
Let H be a min heap
for each vertex v ∈ G.V
    MAKE_SET(v)
for each vertex u ∈ $G.V$
   for each v in G.AdjList[u]
       H.insert(pair(u, v))

while |A| != |V| - 1
    (u, v) = H.ExtractMin()
    set_u = FIND_SET(u)
    set_v = FIND_SET(v)
    if set_u != set_v
        $A = A \cup \{(u, v)\}$
        UNION(set_u, set_v)
return A
```

Time complexity

- Line-7 makes O(|V|) operations
- Line-8,9,10 can be implemented using BUILD_MIN_HEAP, which requires O(|E|) operations
- The loop at line-11 may execute |E| times
- The total number of operations at line-12 are O(|E| * log(|E|))
- The total number of operations at Line-13 and Line-14 are O(|E|*log(|V|))
- The other operations in the loop take constant time during every iteration
- Therefore, the time complexity of Kruskal's algorithm is O(|E| * log (|V|))
  - Notice that in a connected graph |E| >= |V|-1
  - log(|E|) <= 2 * log(|V|)