CS 521 Lecture IV

DREXEL UNIVERSITY
DEPT. OF COMPUTER SCIENCE

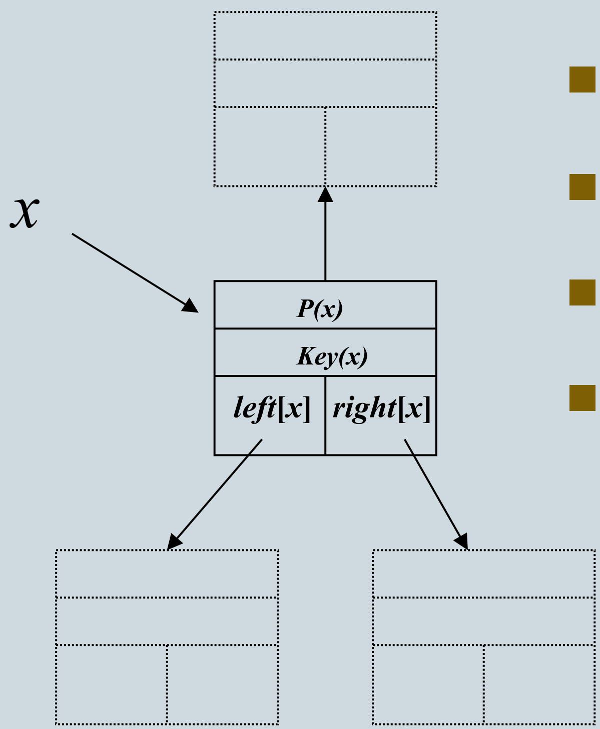
FALL 2011

Today's Lecture

- Binary Search Trees
- Balanced Binary Search Trees

The Structure

- Each node ***x*** in a binary search tree (BST) contains:



  - ***key***[*x*]- The value stored at ***x***.
  - ***left***[*x*]- Pointer to left child of ***x***.
  - ***right***[*x*]- Pointer to right child of ***x***.
  - ***p***[*x*]- Pointer to parent of ***x***.

BST- Property

- Keys in BST satisfy the following properties:
  - Let x be a node in a BST:
  - If y is in the left subtree of x then:

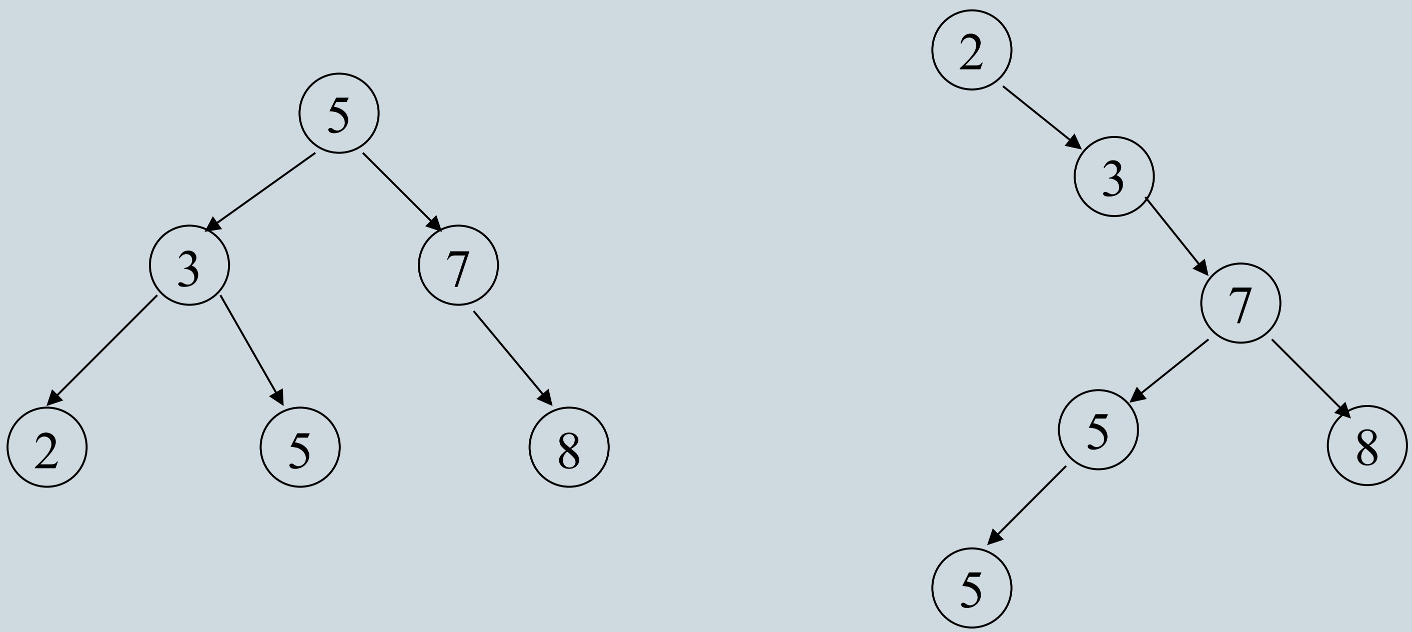
    $$key[y] <= key[x]$$

  - If y is in the right subtree of x then:

    $$key[y] > key[x]$$

Example:

- Two valid BST's for the keys: 2,3,5,5,7,8.



In-Order Tree walk

- Can print keys in BST with in-order tree walk.
- Key of each node printed between keys in left and those in right subtrees.
- Prints elements in monotonically increasing order.
- Running time?

In-Order Traversal

Inorder-Tree-Walk(x)

1: If $\boldsymbol{x}$!=$\boldsymbol{NIL}$ then

2: Inorder-Tree-Walk($\boldsymbol{left}[x]$)

3: Print($\boldsymbol{key}[x]$)

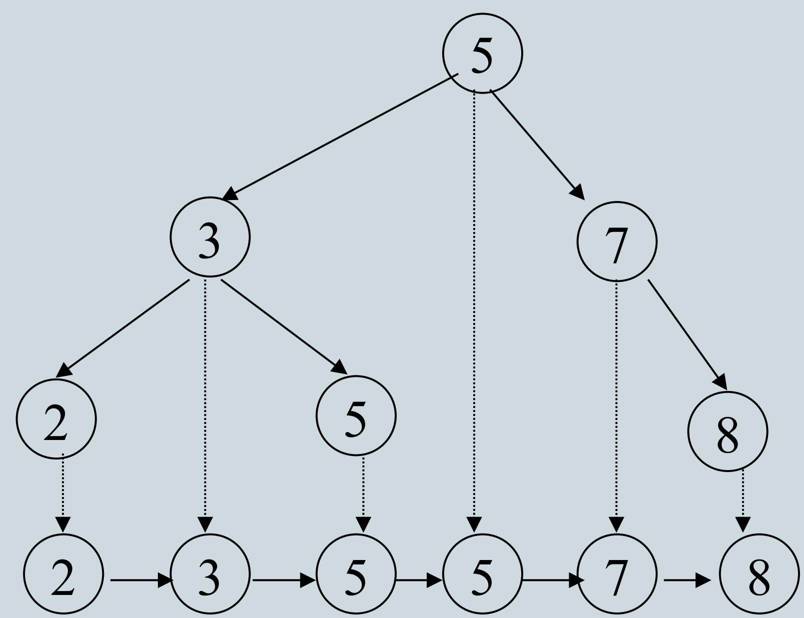
4: Inorder-Tree-Walk($\boldsymbol{right}[x]$)

What is the recurrence for $\boldsymbol{T}(\boldsymbol{n})$?

What is the running time?

In-Order Traversal

- In-Order traversal can be thought of as a projection of BST nodes on an interval.
- At most 2^d nodes at level d=0,1,2,...



Other Tree Walks

Preorder-Tree-Walk(x)

1: If $\boldsymbol{x}$!=$\boldsymbol{NIL}$ then

2: Print($\boldsymbol{key}$[$\boldsymbol{x}$])

3: Preorder-Tree-Walk($\boldsymbol{left}$[$\boldsymbol{x}$])

4: Preorder-Tree-Walk($\boldsymbol{right}$[$\boldsymbol{x}$])

Postorder-Tree-Walk(x)

1: If $\boldsymbol{x}$!=$\boldsymbol{NIL}$ then

2: Postorder-Tree-Walk($\boldsymbol{left}$[$\boldsymbol{x}$])

3: Postorder-Tree-Walk($\boldsymbol{right}$[$\boldsymbol{x}$])

4: Print($\boldsymbol{key}$[$\boldsymbol{x}$])

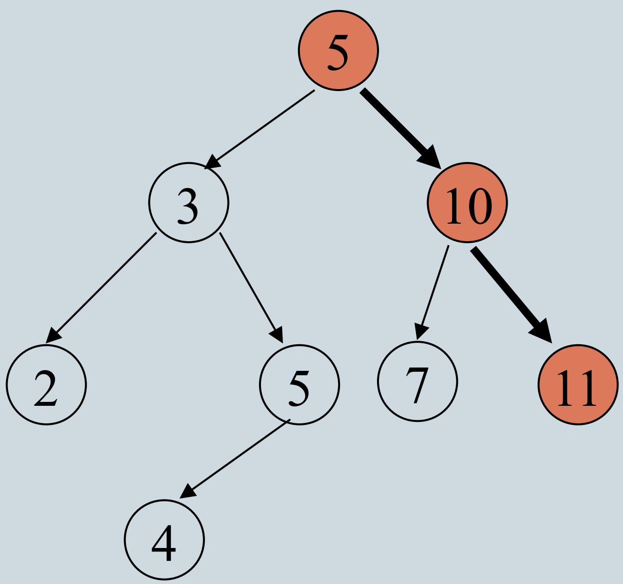
Searching in BST:

- To find element with key ***k*** in tree ***T***:
  - Compare ***k*** with
  - If ***k***<***key***[**root**[***T***]] search for k in
  - Otherwise, search for ***k*** in

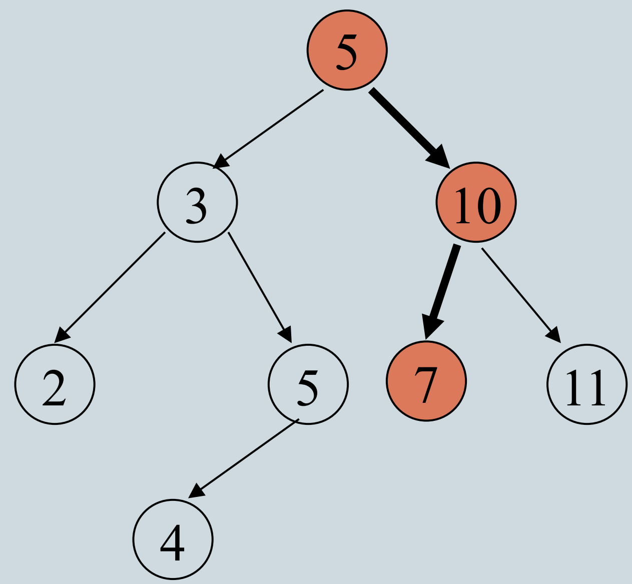
```
Search(T,k)
x=root[T]
If x=NIL then return("not found")
If k=key[x] then return("found the key")
If k< key[x] then Search(left[x],k)
else Search(right[x],k)
```

Examples:

- Search(T,11)

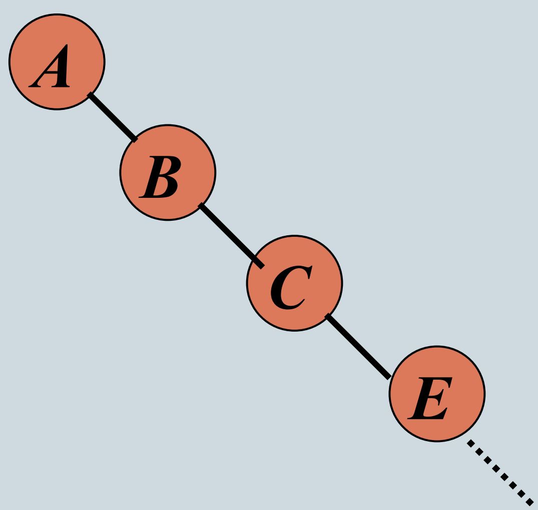


- Search(T,6)



Analysis of Search

- Running time of height ***h*** is
- After insertion of ***n*** keys, worst case running time of search is



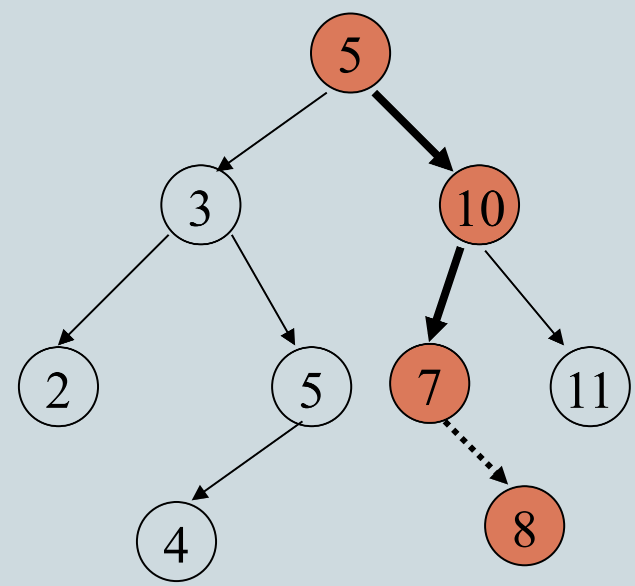
BST Insertion

- Basic idea: similar to search.
- BST-Insert:
  - Take an element ***z*** (whose right and left children are NIL) and insert it into ***T***.
  - Find a place where ***z*** belongs, using code similar to that of Search.
  - Add ***z*** there.

Insert Key


```
BST-Insert(T,z)
y=NIL
x=root[T]
While x!=NIL do
    y=x;
    if key[z]<key[x] then
        x=left[x]
    else x=right[x]
p[z]=y
if y=NIL the root[T]=z
else if key[z]<key[y] then left[y]=z
else right[y]=z
```

Insert(*T*,8)



Locating the Minimum

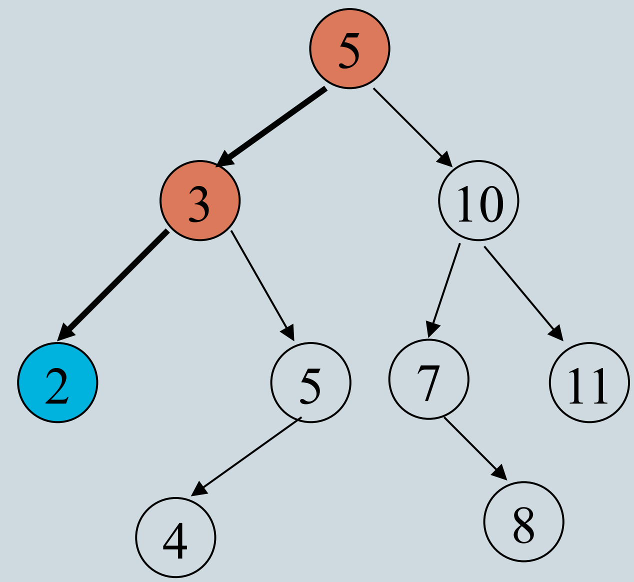
BST-Minimum(***T***)

1: ***x***=root[***T***]

2: While ***left***[***x***]!=***NIL*** do

3: ***x***=***left***[***x***]

4: return ***x***



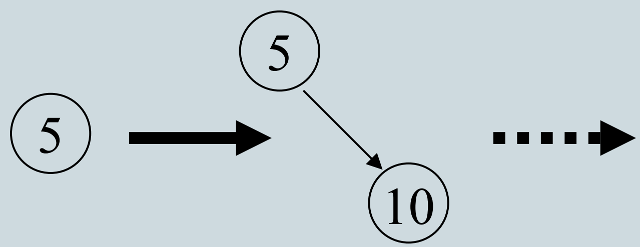
Application: Sorting

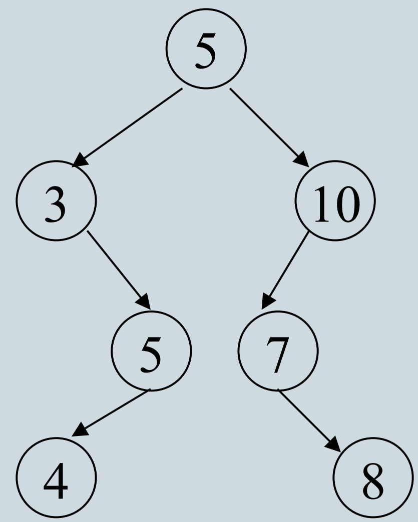
- Can use BST-Insert and Inorder-Tree-Walk to sort list of ***n*** numbers

BST-Sort

```
root[T]=NIL
for i=1to n do
    BST-Insert(T,A[i])
Inorder-Tree-Walk(T)
```

Sort Input: 5, 10, 3, 5, 7, 5, 4, 8





Inorder Walk: 3, 4, 5, 5, 7, 8, 10

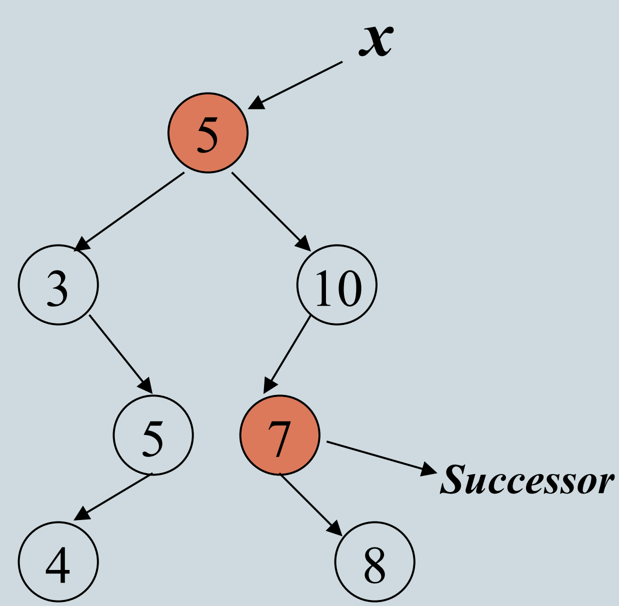
Analysis:

- The running time depends on the height of the tree (the Insert time).
- The average case analysis is like quick sort (which element will sit in the root).
- Therefore the expected running time is $\boldsymbol{O(n \log n)}$.
- Average BST height is $\boldsymbol{O(\log n)}$.

Successor

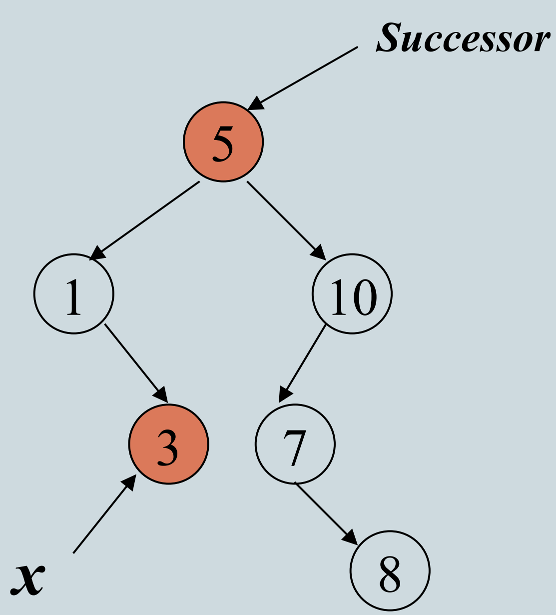
Given ***x***, find node with smallest key greater than ***key***[***x***]. Here are two cases depending on right subtree of ***x***.

- Successor Case 1:
  - The right subtree of ***x*** is not empty. Successor is leftmost node in right subtree. That is, we must return BST-Minimum(***right***[***x***])



Successor

- Successor Case 2: The right subtree of ***x*** is empty. Successor is lowest ancestor of ***x***. Observe that, "Successor" is defined as the element encountered by ***inorder*** traversal.



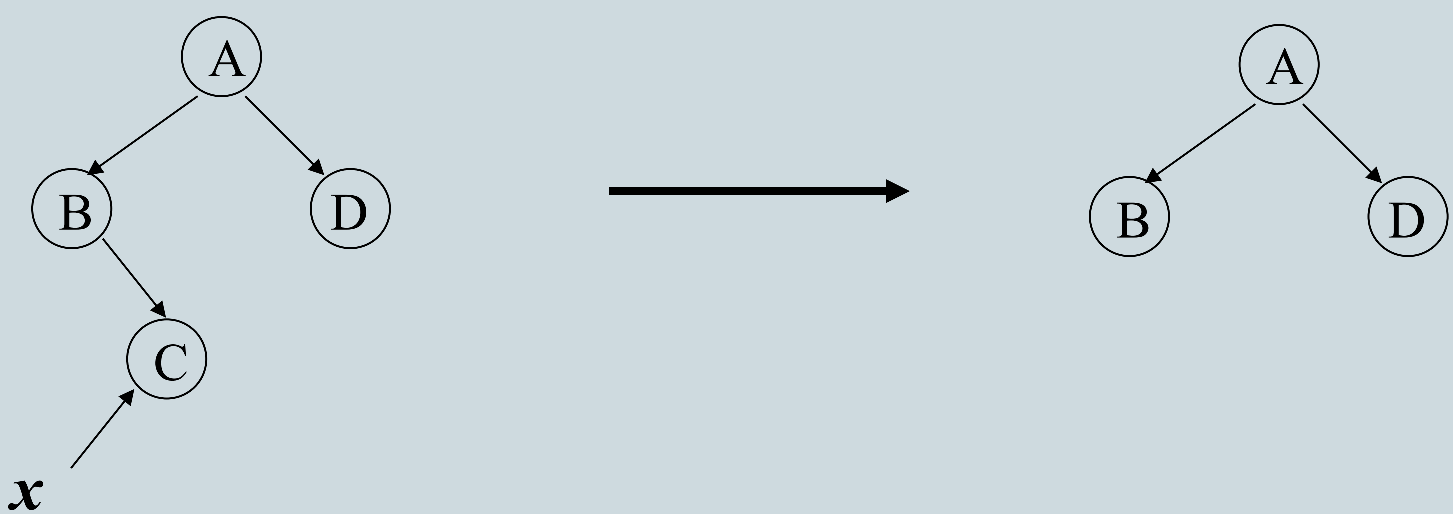
BST-Successor(*x*)

```
If right[x]!=NIL then
    return BST-Minimum(right[x])
y=p[x]
While (y!=NIL) and (x=right[y])
    x=y
    y=p[y]
return y
```

Running time?

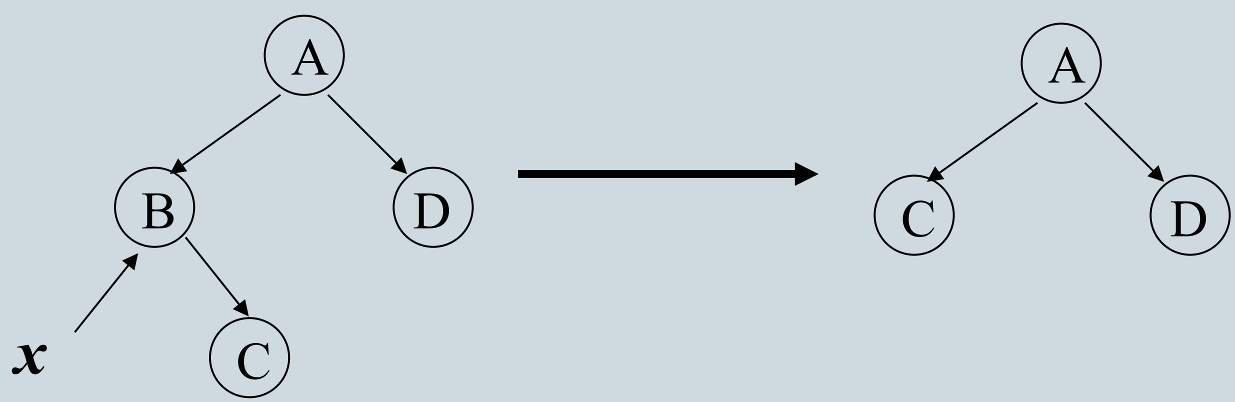
Deletion

- Delete a node ***x*** from tree ***T***:
  - Case 1: ***x*** has no children.



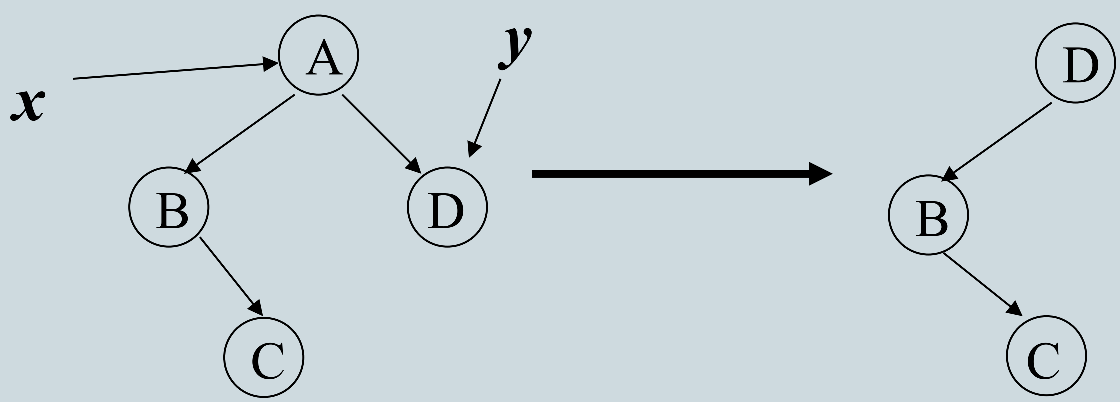
Deletion:

- Case 2: $\boldsymbol{x}$ has one child (call it $\boldsymbol{y}$). Make $\boldsymbol{p}[\boldsymbol{x}]$ to replace $\boldsymbol{y}$ instead of $\boldsymbol{x}$ as its child, and make $\boldsymbol{p}[\boldsymbol{x}]$ to be $\boldsymbol{p}[\boldsymbol{y}]$.



Deletion:

- Case 3: ***x*** has two children:
  - Find its successor (or predecessor) ***y***.
  - Remove ***y***. (Note ***y*** has at most one child, why?)
  - Replace ***x*** by ***y***.



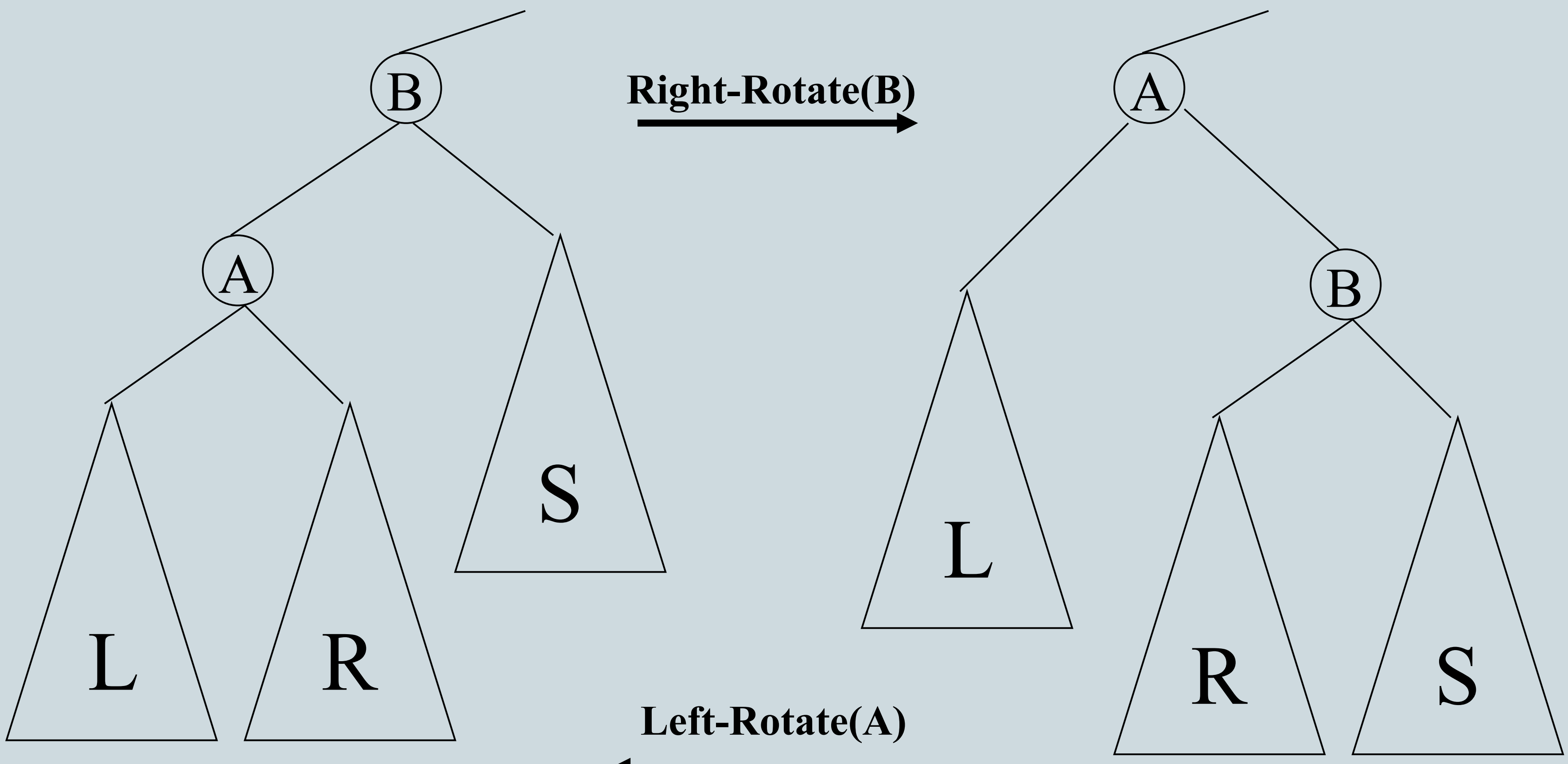
Delete Procedure


```
BSFT-Delete(T, z)
If (left[z]=NIL) or (right[z]=NIL) then
     y=z
else y=BST-Successor(z)
If left[y]!=NIL then
      x=left[y]
else x=right[y]
If x!=NIL then p[x]=p[y]
If p[y]=NIL then root[T]=x
else if y=left[p[y]] then left[p[y]]=x
else right[p[y]]=x
if y!=z then key[z]=key[y]
return y
```

Red-black Trees

- They are *balanced search trees,* which means their height is $O(\log n)$.
- Most of the search and update operations on these trees take $O(\log n)$ time.
- The structure is well balanced, i.e. each subtree itself is a balanced search tree.

Rotation



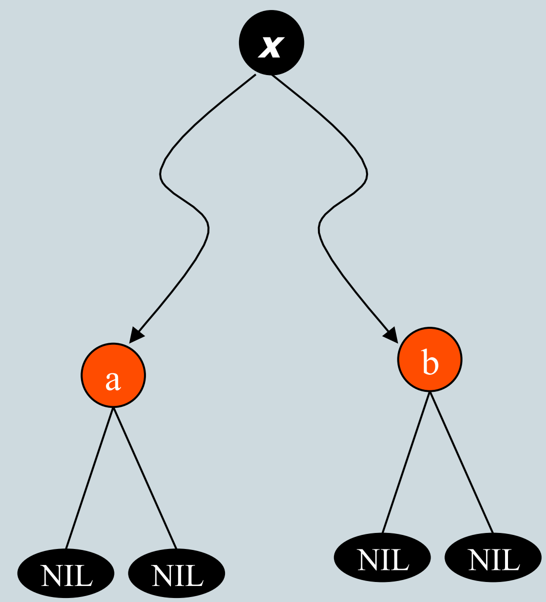
Rotation

- Is the basic operation for maintaining balanced trees.
- Maintains inorder key ordering:
  - For all ***a*** in **L**, ***b*** in **R**, ***c*** in **S** we have ***a<=b<=c***.
- Depth(**L**) decreases by 1.
- Depth(**R**) stays the same.
- Depth(**S**) increases by 1.
- Takes ***O***(1)

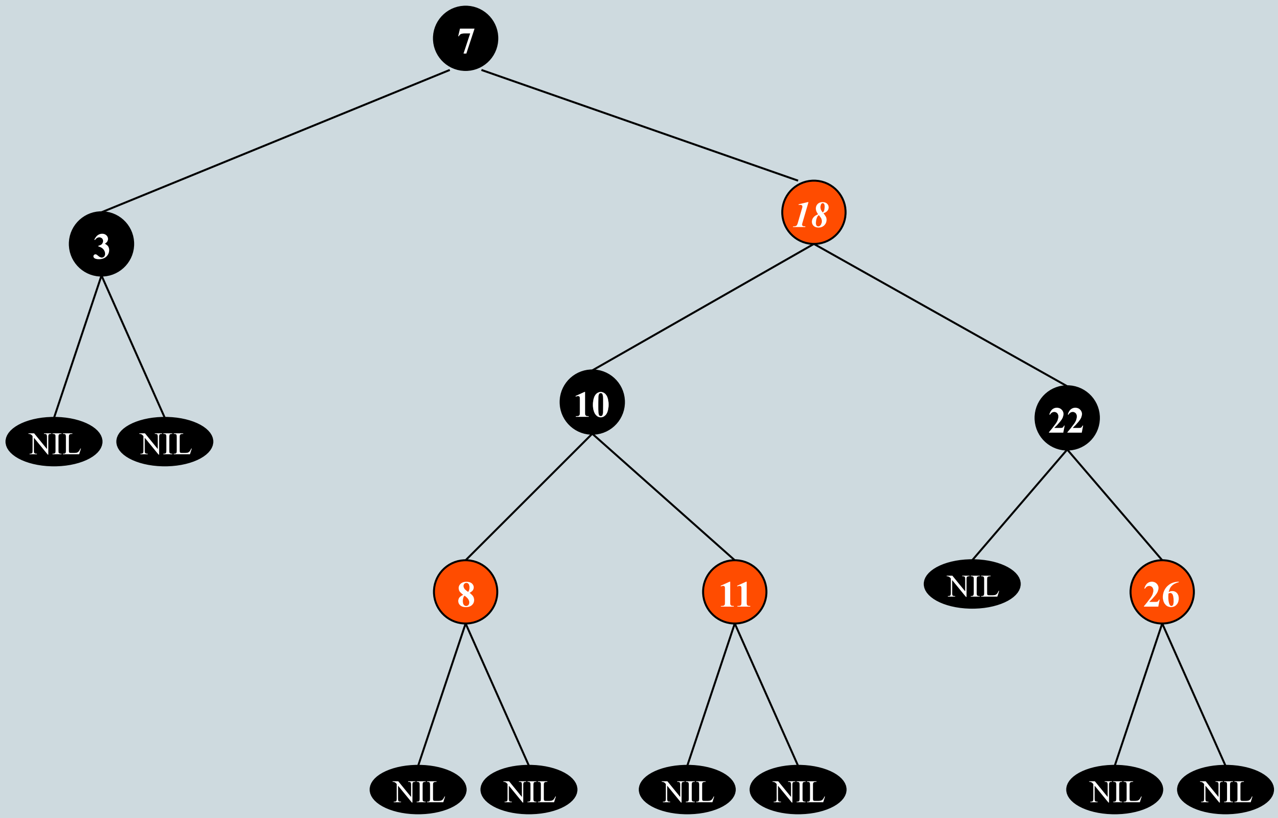
```
Left-Rotate(T, x)
y=right[x]
right[x]=left[y]
If left[y]!=NIL then p[left[y]]=x
p[y]=p[x]
If p[x]=NIL then root[T]=y
else if x=right[p[x]] then
        left[p[x]]=y
else right[p[x]]=y
left[y]=x
p[x]=y
```

Red-Black Trees

- Every node is either ***red*** or ***black***.
- Root and Leaves (NIL) are black.
- If a node is red, then both its children are black.
- All paths from a node ***x*** to a leaf have same number of black nodes (Black-Height(***x***))

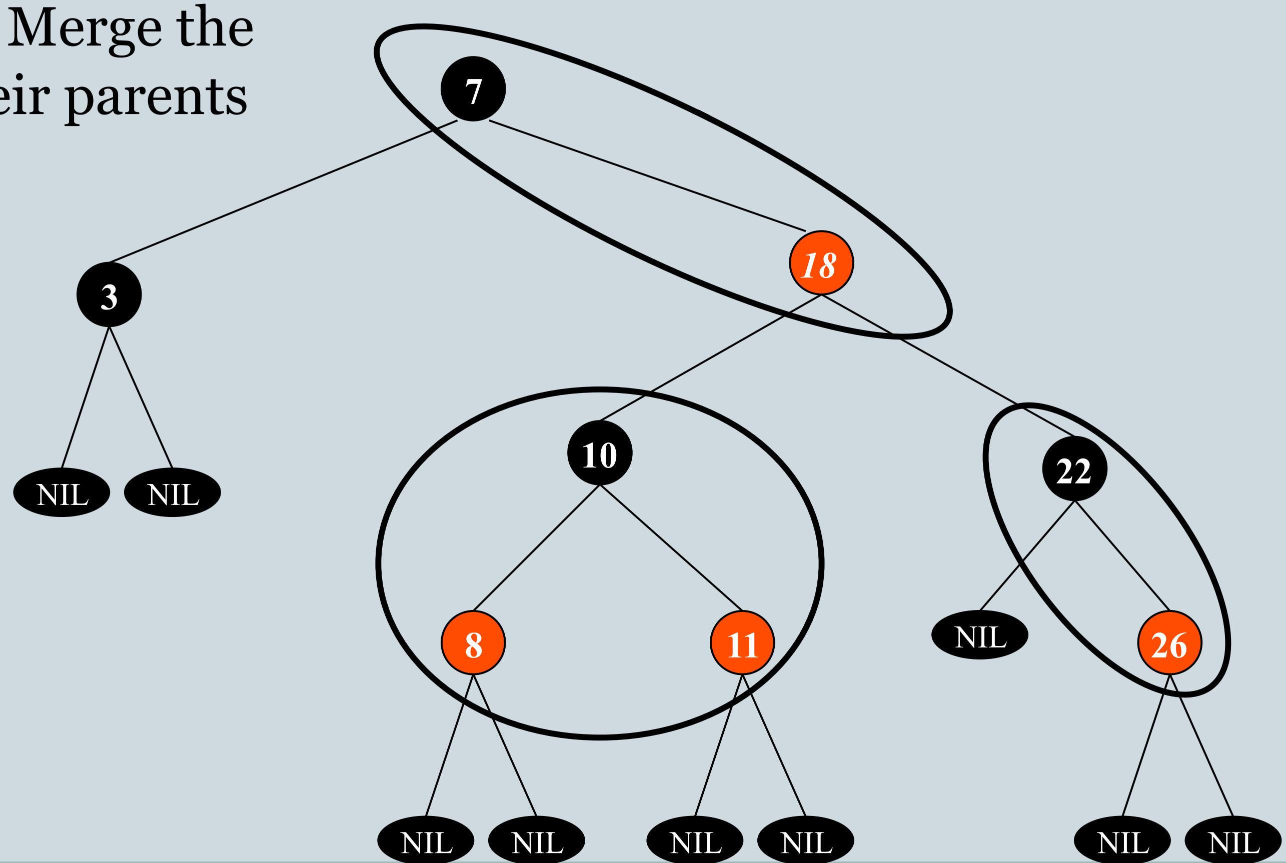


Example



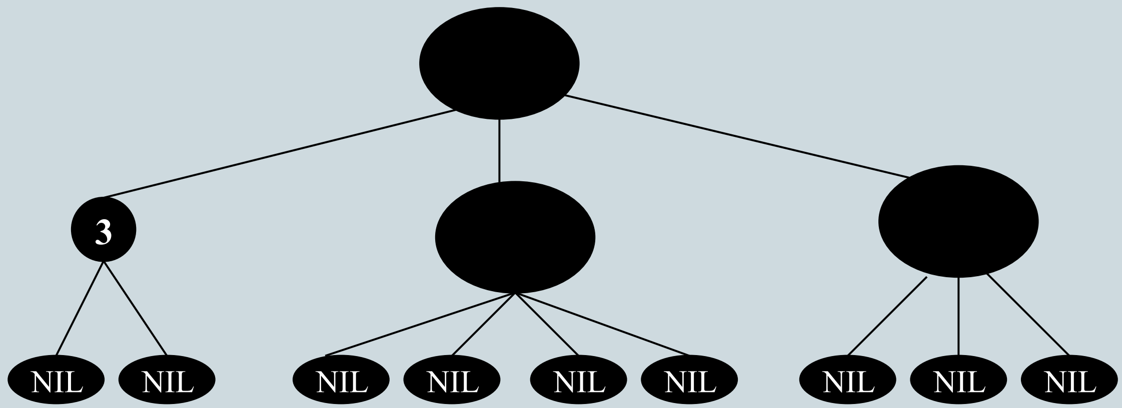
Height

- A red-black tree with n keys has height $\leq 2\log(n+1)$.
- Proof (Intuition): Merge the red nodes into their parents



Proof:

- Produces a tree with nodes having 2,3, or 4 nodes



- Height ***h'*** of new tree is black height of original tree:
  - $h' >= h/2$
  - n+1 leaves implies $n+1>=2^{h'}$
  - $\log(n+1)>=h'>=h/2$

Red-Black Insertion

- Insert ***x*** into tree
- Color ***x*** red.
- Red-black property 1 still holds (as long as x isn't the root).
- Red-Black property 2 still holds (inserted node has NIL's for children).
- Red-black property 4 still holds (***x*** replaces a black NIL and has NIL children).

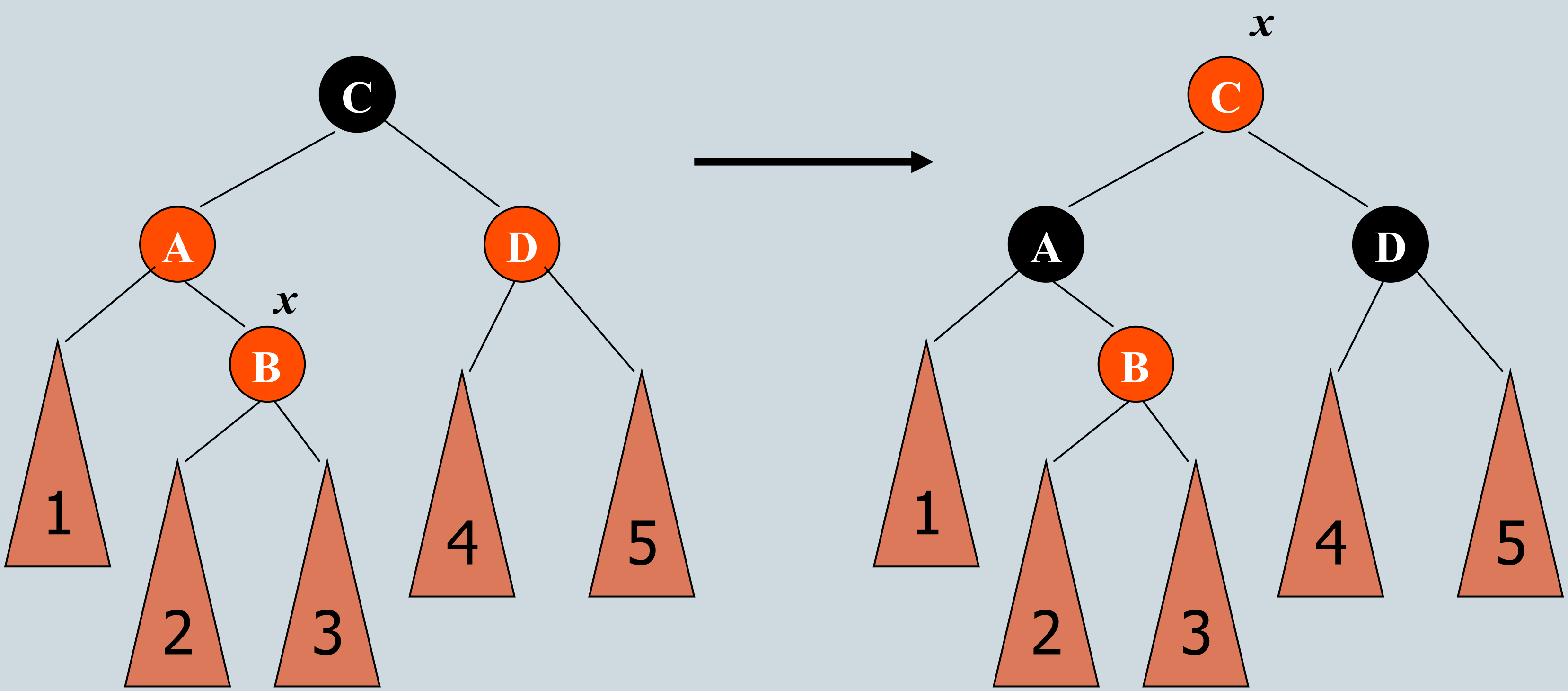
Red-Black Insertion

- If $p[x]$ is red, then property 3 is violated.
- To correct, we move violation up in tree until it can be fixed.
- No new violations will be introduced during this process.
- For each iteration, there are six possible cases.

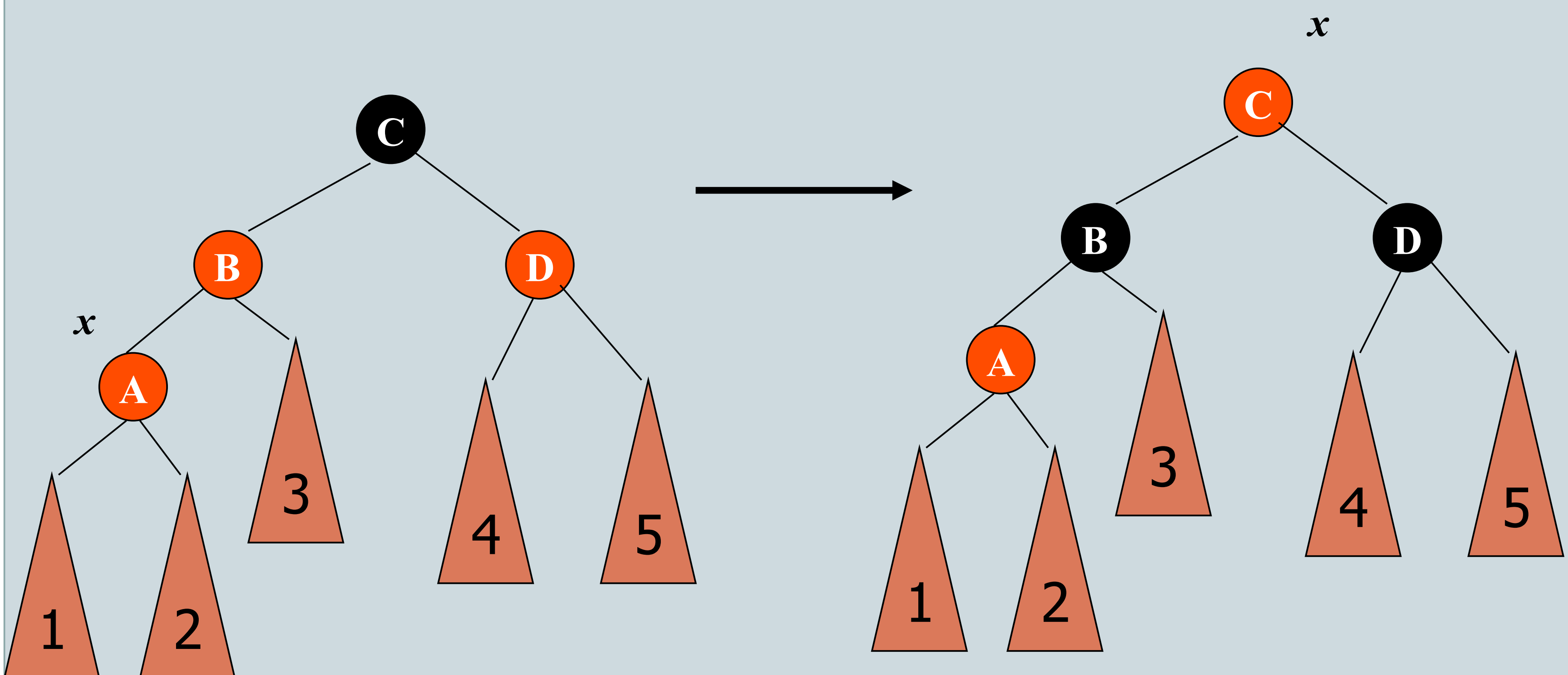
Insertion Cases

- ***x***'s parent is the left child of ***x***'s grandparent.
- ***x***'s parent's sibling (***x***'s uncle) is red.
- Then
  - $\boldsymbol{Color}[\boldsymbol{p}[\boldsymbol{x}]]$= Black
  - $\boldsymbol{Color}[\boldsymbol{right}[\boldsymbol{p}[\boldsymbol{p}[\boldsymbol{x}]]]]$= Black
  - $\boldsymbol{Color}[\boldsymbol{p}[\boldsymbol{p}[\boldsymbol{x}]]]$= Red
  - $\boldsymbol{x}=\boldsymbol{p}[\boldsymbol{p}[\boldsymbol{x}]]$

Insertion case 1



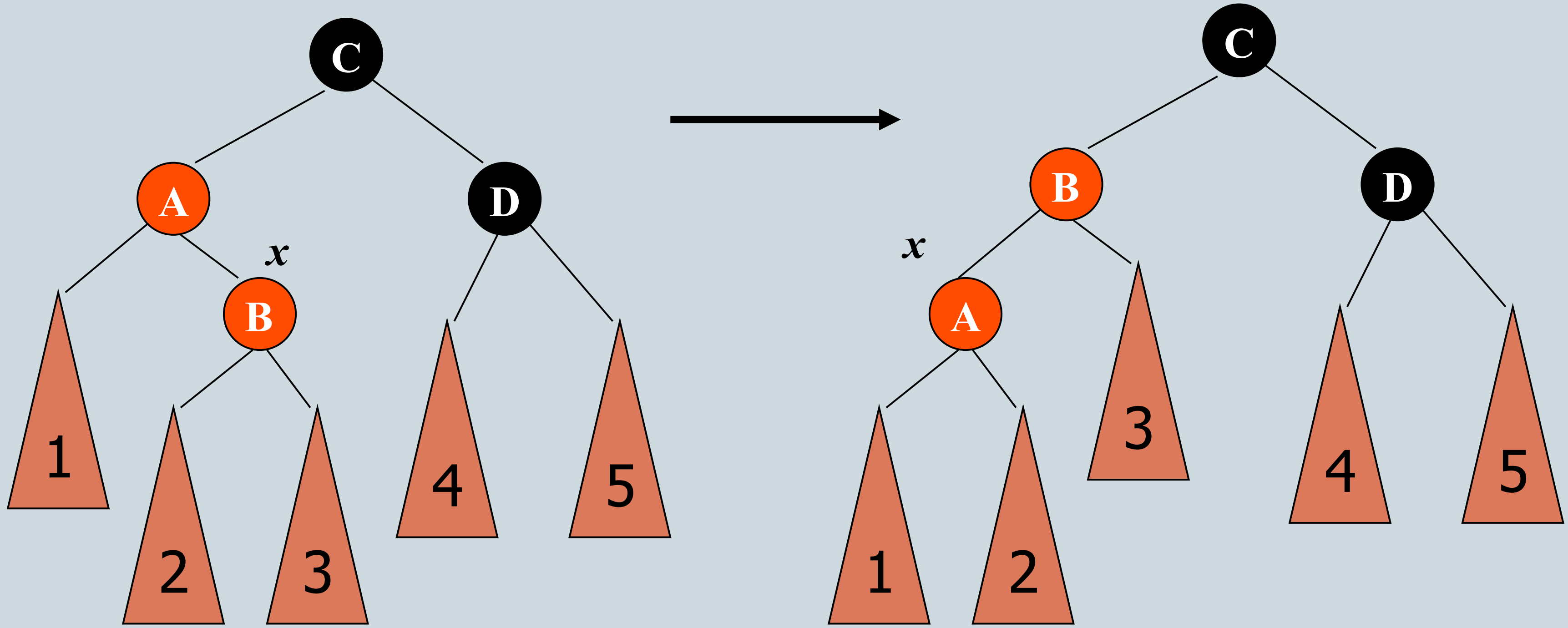
Insertion case 1



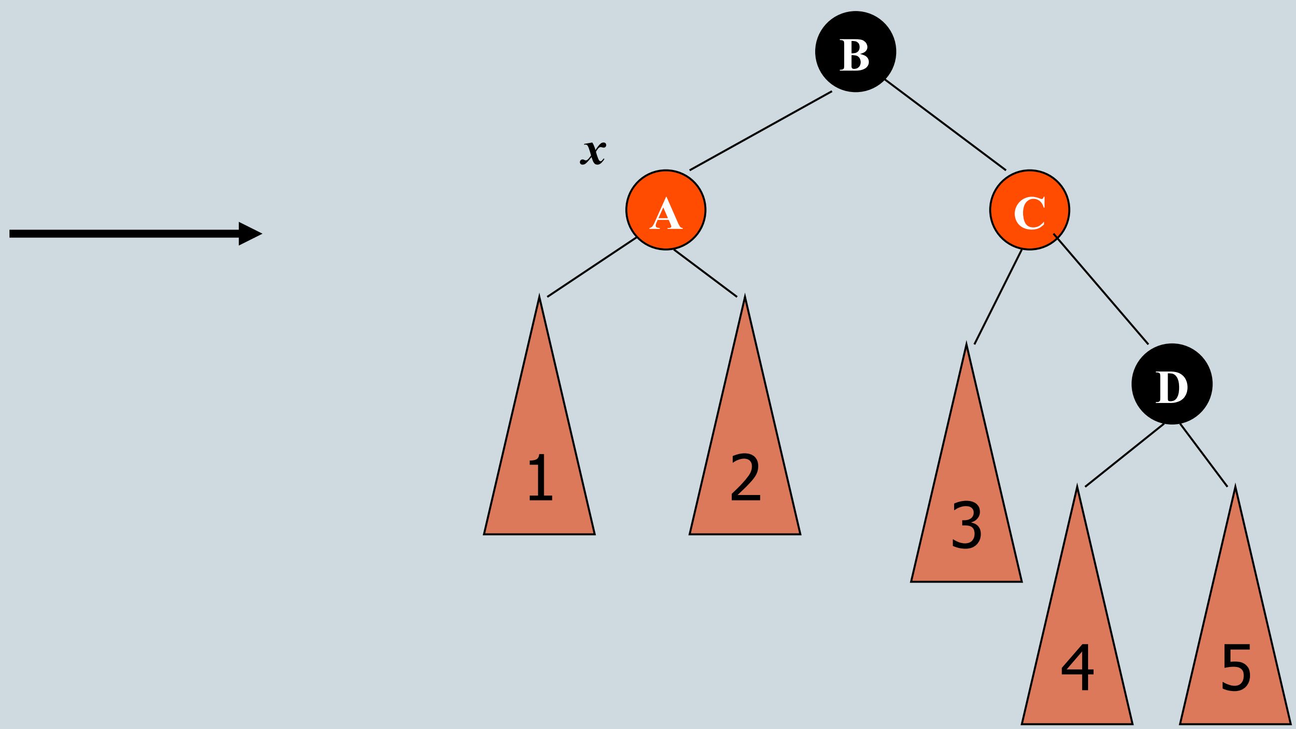
Insertion Cases

- $\boldsymbol{x}$'s parent is the left child of $\boldsymbol{x}$'s grandparent.
- $\boldsymbol{x}$'s uncle is Black.
- $\boldsymbol{x}$ is right child of $\boldsymbol{p[x]}$.
- Then
  - $x = p[x]$
  - **Left-Rotate(*T*,*x*)**
  - ***Color***$[p[x]]$= Black
  - ***Color***$[p[p[x]]]$= Red
  - **Right-Rotate(*T*,*p*[*p*[*x*]])**

Insertion case 2



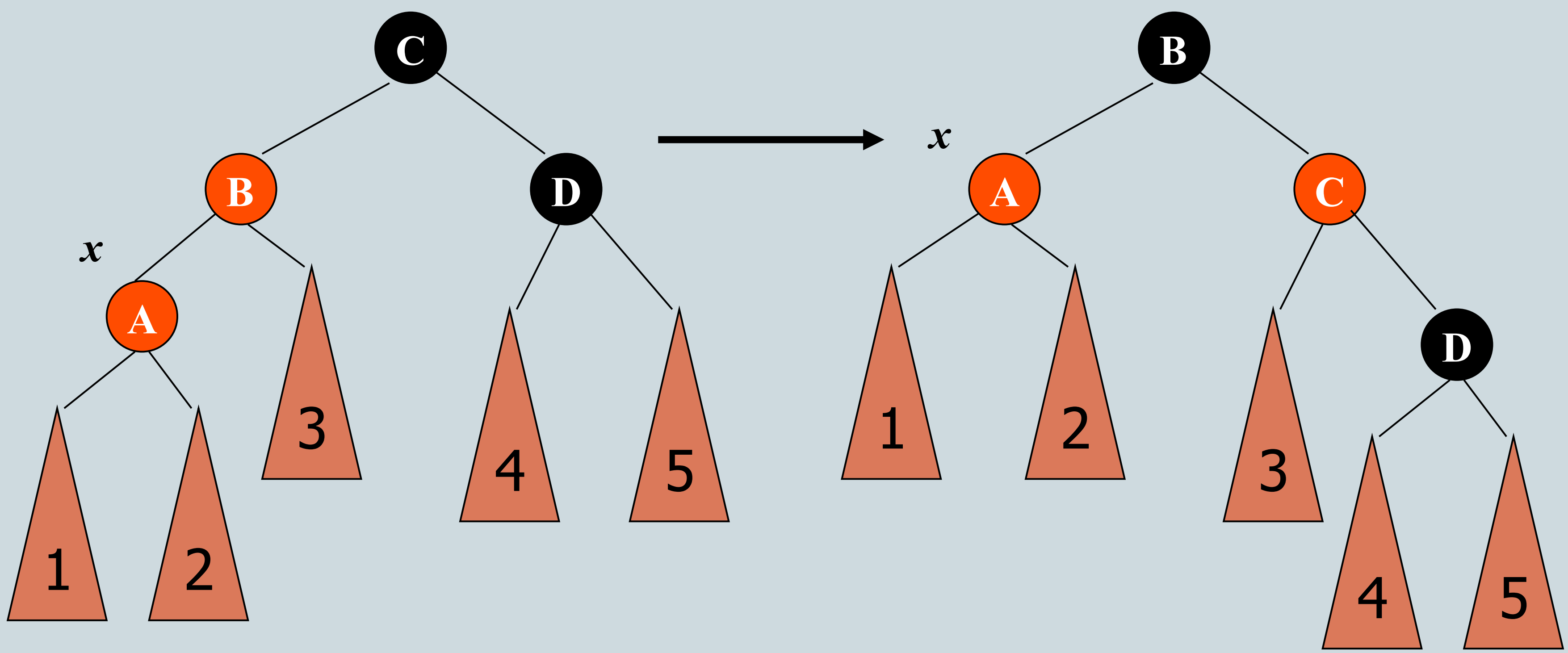
Insertion case 2



Insertion Cases

- $\boldsymbol{x}$'s parent is the left child of $\boldsymbol{x}$'s grandparent.
- $\boldsymbol{x}$'s uncle is Black.
- $\boldsymbol{x}$ is the left child of $\boldsymbol{p}[\boldsymbol{x}]$
- Then
  - ***Color***[$\boldsymbol{p}[\boldsymbol{x}]$]= Black
  - ***Color***[$\boldsymbol{p}[\boldsymbol{p}[\boldsymbol{x}]]$]= Red
  - **Right-Rotate(*T*,*p*[*p*[*x*]])**

Insertion case 3



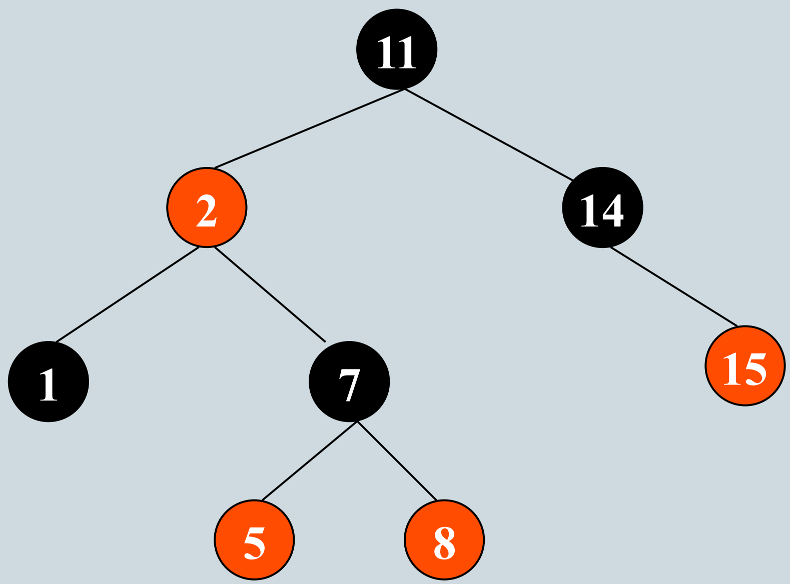
Red-Black Insert

- Cases 4, 5 ,6 are symmetric to 1, 2, 3 (x's parent is the right child of x's grandparent).
- After case 2 or 3, no further correction is needed.

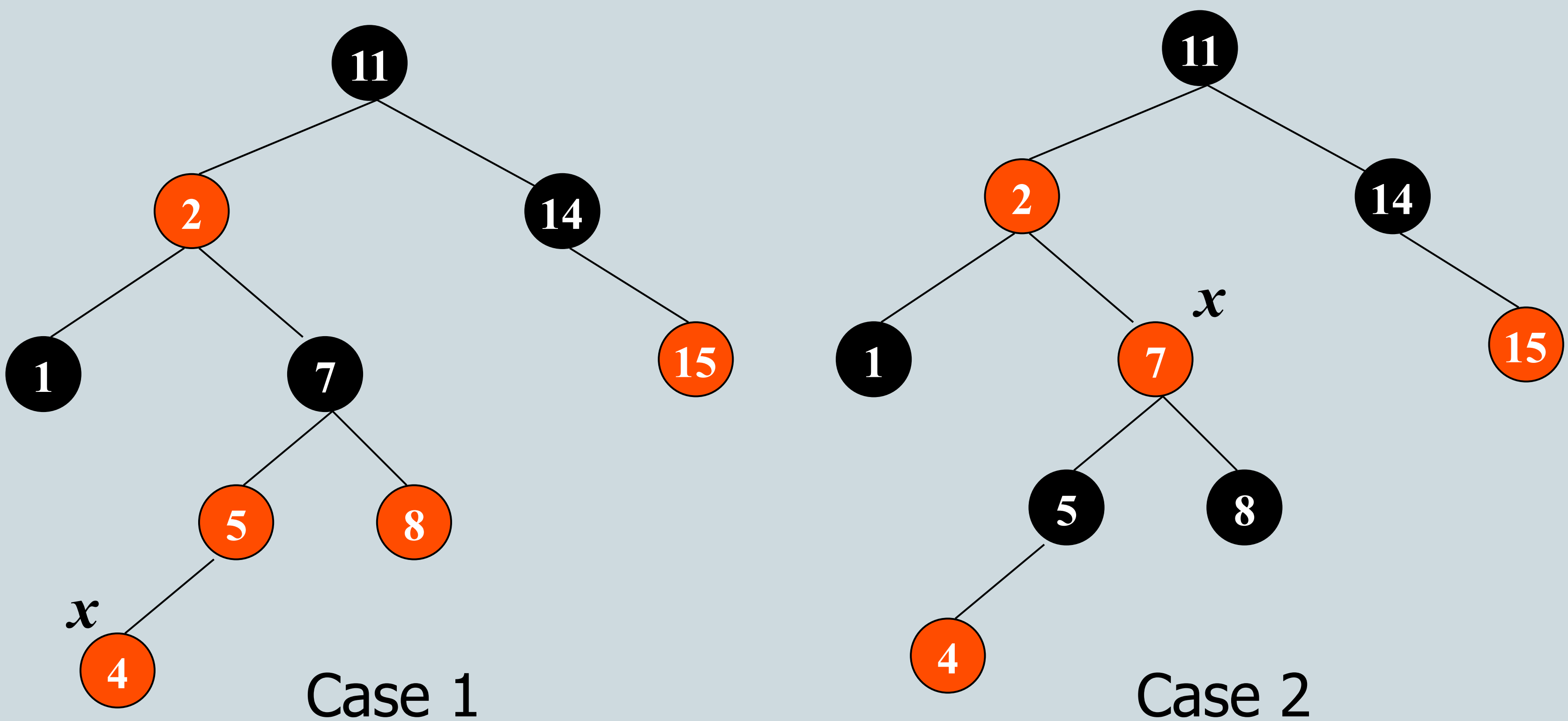
```
RB-Insert(T, x)
1:  Tree-Insert(T,x)
2:  color[x]=Red
3:  While x!=root[T] and color[p[x]]=Red
4:     If  p[x]=left[p[p[x]]] then
5:        y=right[p[p[x]]]
6:        If color[y]=Red then
7:           color[p[x]]=Black
8:           color[y]=Black
9:           color[p[p[x]]]=Red
10:          x = p[p[x]]
11:       else
             if x=right[p[x]] then
12:              x=p[x]
13:              Left-Rotate(T,x)
14:          color[p[x]]=Black
15:          color[p[p[x]]]=Red
16:           Right-Rotate(T, p[p[x]])
17:    else{same as the clause [line 5] with "Right" and "Left" exchanged.}
18:  color[root[T]]=Black
```

Example

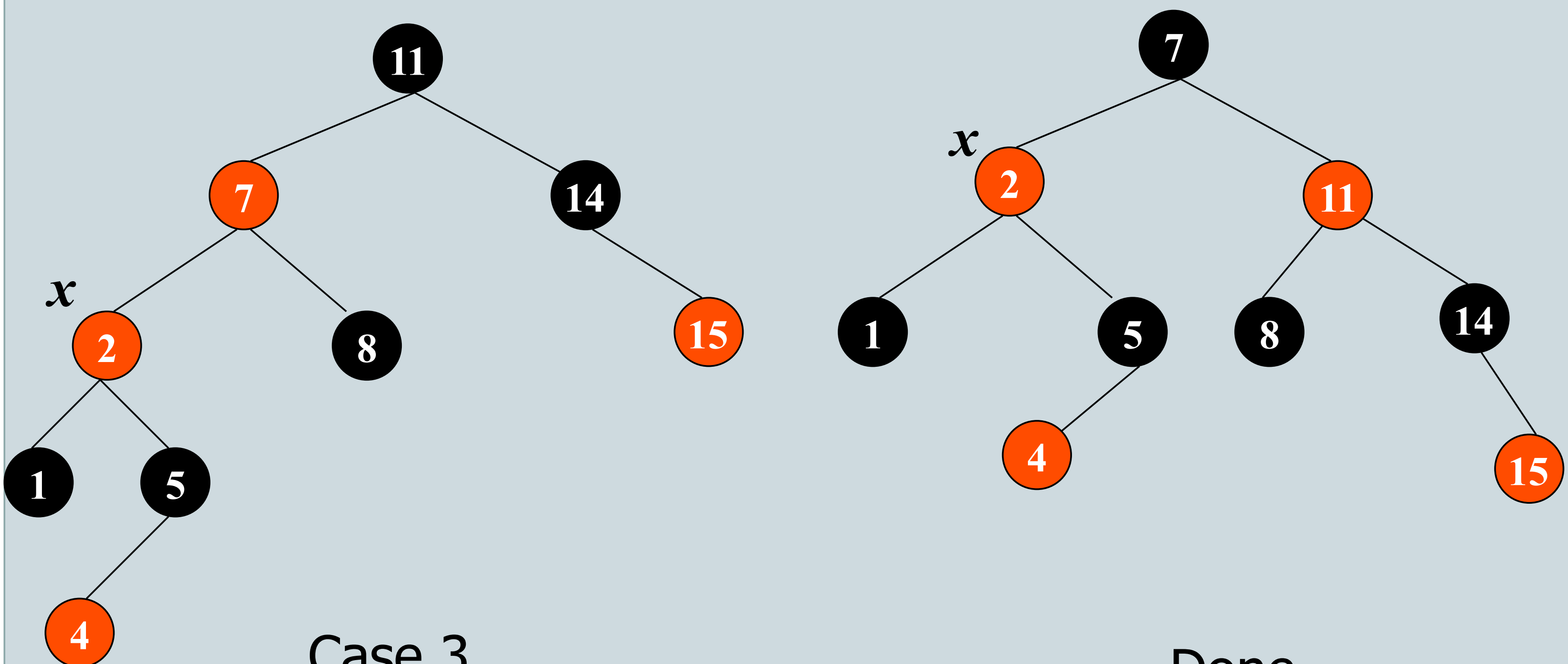
- Use R-B Insert to insert element with key 4.



Example



Example

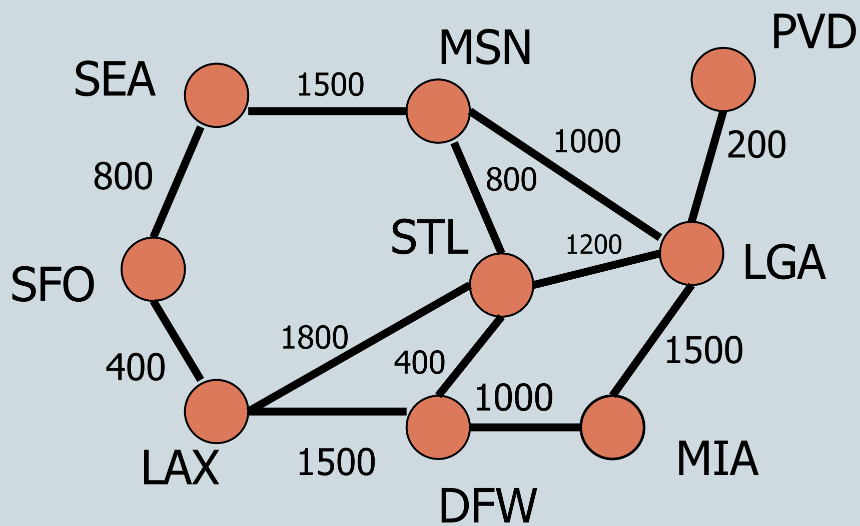


Case 3

Done

Introduction to Graph Theory

- A graph $G = (V,E)$ is pair of sets:
  - V: vertex set.
  - E: edge set.
- A graph may be weighted and its edges might be directed.

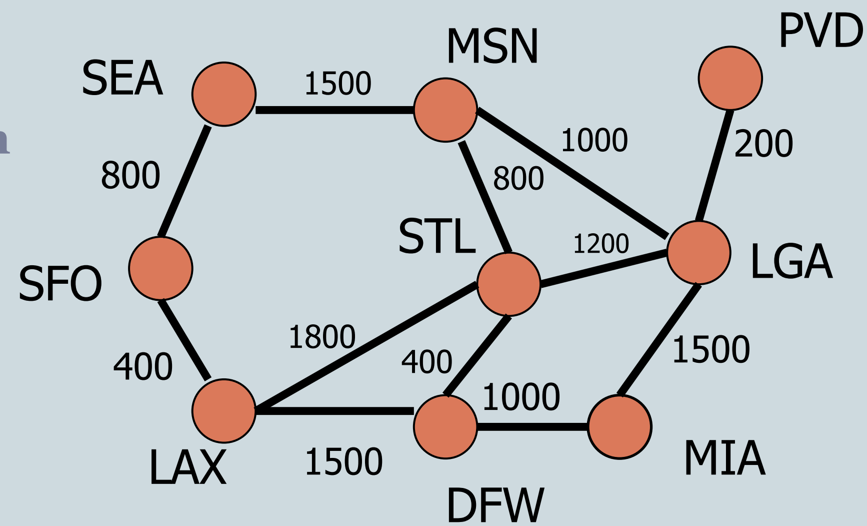


V={Sea, Sfo, Lax, Msn, Stl, Dfw, Mia, Lga, Pvd}

E={(Sea, Sfo), (Sfo,Lax), (Sea,Msn),…, (Lga, Pvd)}

Preliminaries

- Things to know:
    - Path
    - Cycle
    - Sub-graph
    - Degree of a Node
    - Maximum and Minimum Degree
    - Maximum Number of Edges in an Undirected Graph
    - Connected Components of a Graph
    - Shortest Path in a Weighted Graph
    - Tree (rooted tree)
    - Spanning Tree of a Graph:
    - Acyclic Graph
    - Bipartite Graph

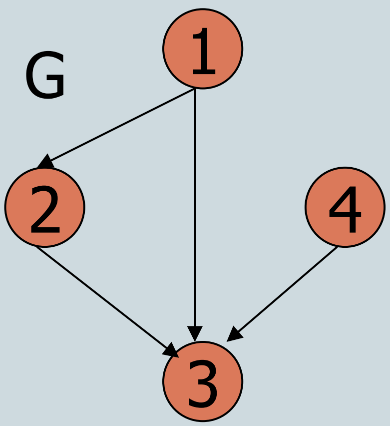


Notations:

- Given A graph $G=(V,E)$, where
  - V is its vertex set, $|V|=n$,
  - E is its edge set, with $|E|=m=O(n^2)$.
- If G is connected then for every pair of vertices u,v in G there is path connecting them.
- In an undirected graph an edge $(u,v)=(v,u)$.
- In directed graph (u,v) is different from (v,u).
- In a weighted graph there are weights associated with edges and/or vertices.
- Running time of graph algorithms are usually expressed in terms of n or m.

- The adjacency matrix of a graph G, denoted by A_G is an n by n defined as follows:

$$A_G[i,j] = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases}$$

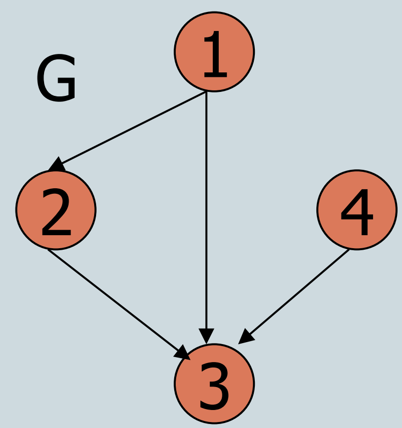


- If G is undirected then A_G is symmetric.

$$A_G = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

- In this method for each vertex v in V, a list $Adj[v]$ will represent those vertices adjacent to v. The size of this list is the degree of v.

$$Adj[1] = \{2,3\}$$
$$Adj[2] = \{3\}$$
$$Adj[3] = \{\}$$
$$Adj[4] = \{3\}$$



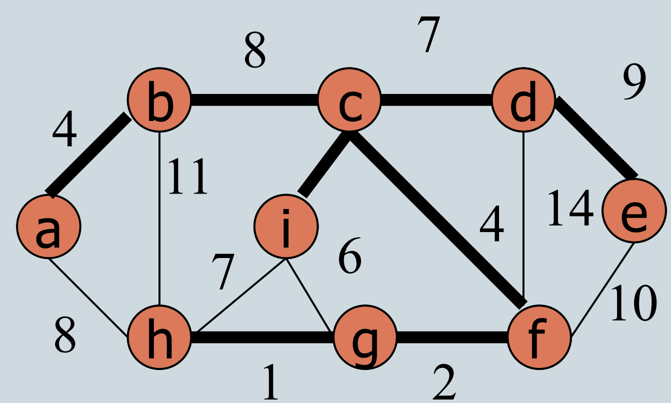
Note that:

- Number of 1's in A_G is m if G is directed; if its undirected, then number of 1's is $2m$.
- Degree of a vertex is the sum of entries in corresponding row of A_G
- If G is undirected then sum of all degree is $2m$.
- In a directed graph sum of the out degrees is equal to m.

Minimum Spanning Tree (MST) in a Weighted Graph

- Let $G=(V,E)$ be a graph on n vertices and m edges, and a weight function w on edges in E.
- A sub-graph T of G through all vertices which avoids any cycle is a spanning tree.
- The weight of T is defined as sum of the weights of all edges in T:

$$w(T) = \sum_{(u,v)\in T} w(u,v)$$



Greedy MST

- The greedy algorithm tries to solve the MST problem by making locally optimal choices:
  1. Sort the edges by weight.
  2. For each edge on sorted list, include that in the tree if it does not form a cycle with the edges already taken; Otherwise discard it
- The algorithm can be halted as soon as n-1 edges have been kept.
- Step 1. takes $O(m \log m)=O(m \log n)$.
- Today, we will see that Step 2 can be done in $O(n \log n)$ time, later we will present a linear time implementation from this step.

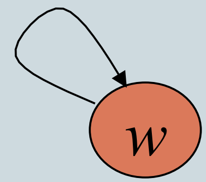
Set Operation

- In the proof of running time for our MST algorithm we will use the following set operations:
  - **Make-Set**(v): creates a set containing element v, $\{v\}$.
  - **Find-Set**(v): returns the set to which v belongs to.
  - **Union**(u,v): creates a set which is the union of the two sets, one containing v and one containing u.
- As an example, we can use a pointer to implement a set system: **Make-Set(*v*)** will create a single node containing element ***v***.

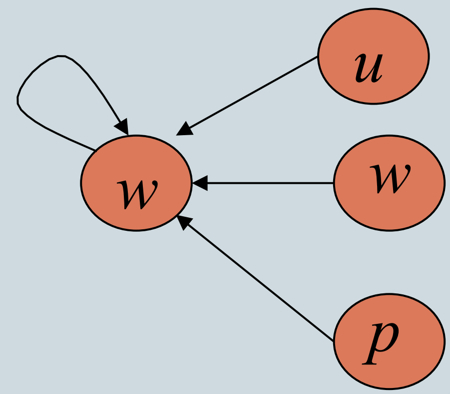
  Find-set(*u*) will return the name of the first element in the set that contains ***u***, and finally the **union(*u*,*v*)** will concatenate the sets containing ***u*** and ***v***.

Example of a set Operations

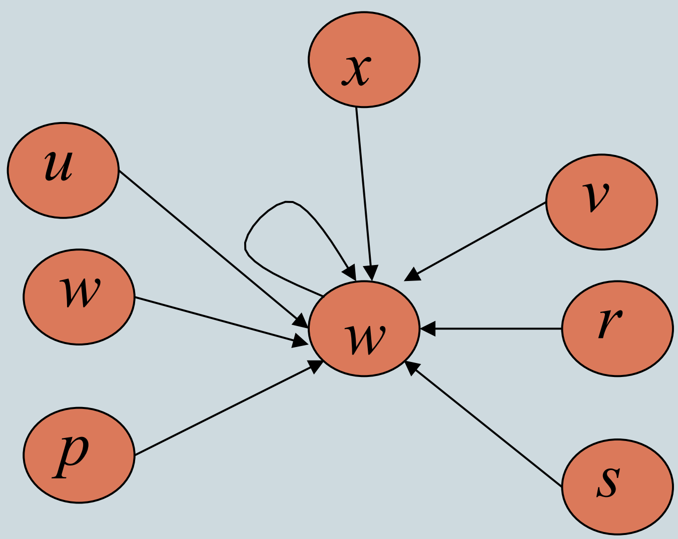
- Use linked list to show a set
- Make-Set(w):

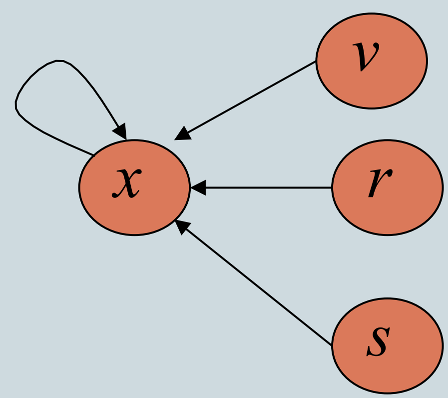


- Find-Set(u): (will return w)



- Union(u,v):





Running Time of Set Operations

- The Make-Set and Find-Set will run in $O(1)$-time.
- How fast can we compute the union.
- Let us ask a different question. Let $N=\{1,\ldots,n\}$ be a set of n integers, and let $P=\{(u,v) \mid u$ and v in $N\}$ be a subset of pairs from $n \times n$.
- For $u=1$ to n Make-Set(u);

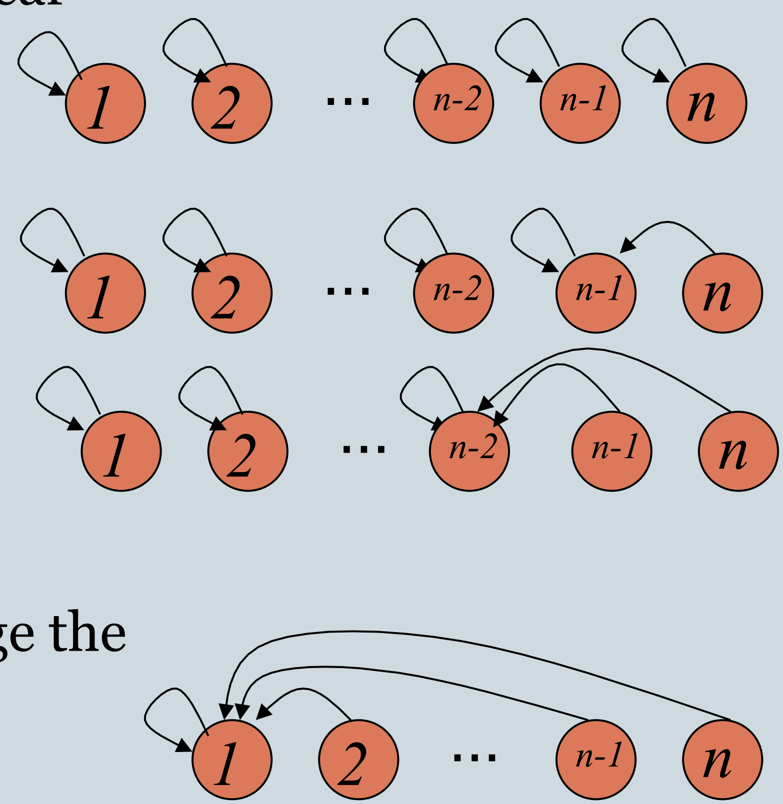
  For every pair (u,v) in P

  If Find-Set(u)!= Find-Set(v)

  Union(u,v)
- Question: How many times does the pointer for an element get redirected?

Union Operation

- Each merge of two sets might take linear number of pointer changes.
- We might have up $O(n^2)$ pointer changes.
- Let us keep a number associated with each set in its root, Rank(u), which tell how many elements a set has.
- When merging two lists, always change the pointers in the list with smaller rank.



Union Operation

- Now each time a pointer changes its corresponding set doubles in the size.
- During the whole process the maximum set can become of size at most n.
- For a specific pointer this happen at most log n times,

$$2^0, 2^1, 2^2, \ldots, 2^k = m, \text{ which means } k = \log n$$

- Over all n elements, this will result in an O(n log n) number of pointer updates.

Kruskal's MST Algorithm

- It is directly based on Generic MST.
- At each iteration, it finds a light edge, which is also safe, and adds it to an ever growing set, *A*, which will eventually become the MST.
- During the course of algorithm, the structure generated by algorithm is a forest.

1. $A \leftarrow \varnothing$
2. for each $v \in V_G$ do
3. Make-Set(v)
4. Sort Edges in E_G
5. for each $(u, v) \in E_G$

   (In order of increasing weights)
6. if Find-Set(u) $\neq$ Find-Set(v)
7. $A \leftarrow A \cup \{(u, v)\}$
8. Union(u, v)
9. Return A

Running time of Kruskal's Algorithm

- Step 1: $O(1)$
- Steps 2,3: $O(n)$
- Step 4: $O(m \log n)$
- Steps 5-8:
  $O(m + (n \log n))$

1. $A \leftarrow \varnothing$
2. for each $v \in V_G$ do
3. $\quad$ Make-Set(v)
4. Sort Edges in E_G
5. for each $(u, v) \in E_G$
   (In order of increasing weights)
6. $\quad$ if Find-Set$(u) \neq$ Find-Set(v)
7. $\quad\quad A \leftarrow A \cup \{(u, v)\}$
8. $\quad\quad$ Union(u, v)
9. Return A