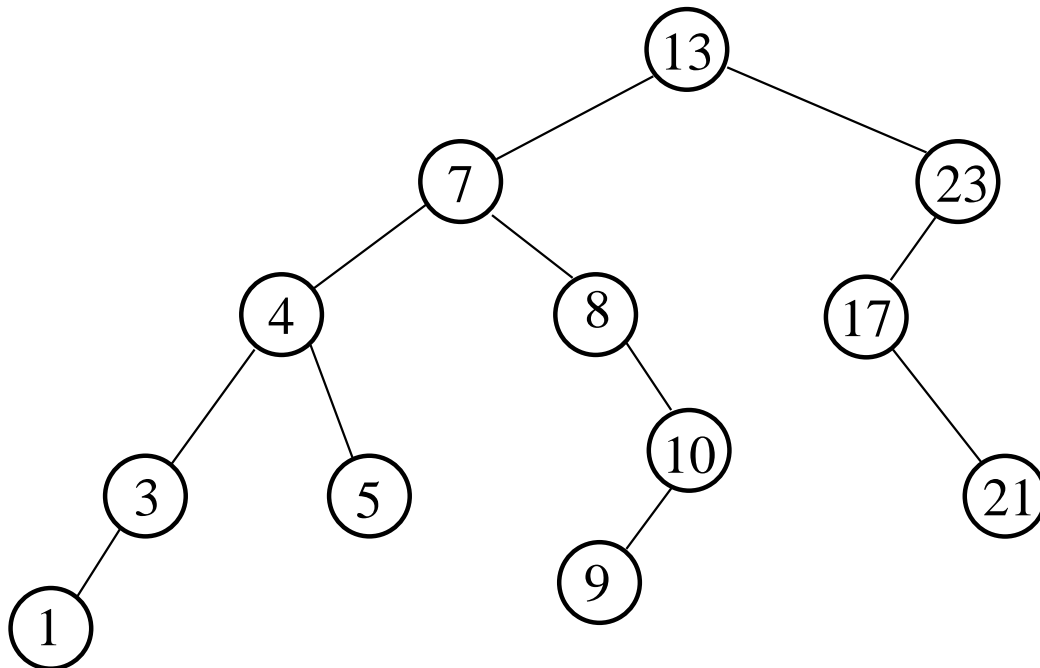


Binary Search Trees

```
class node {  
    int key;  
    node *left, *right, *parent;  
};
```

- A **Binary Search Tree** is a binary tree with the following properties: Given a node x in the tree
 - if y is a node in the left subtree of x , then $y \rightarrow \text{key} \leq x \rightarrow \text{key}$.
 - if y is a node in the right subtree of x , then $x \rightarrow \text{key} \leq y \rightarrow \text{key}$.



- We assume that all keys are distinct.

Binary Search Tree Operations

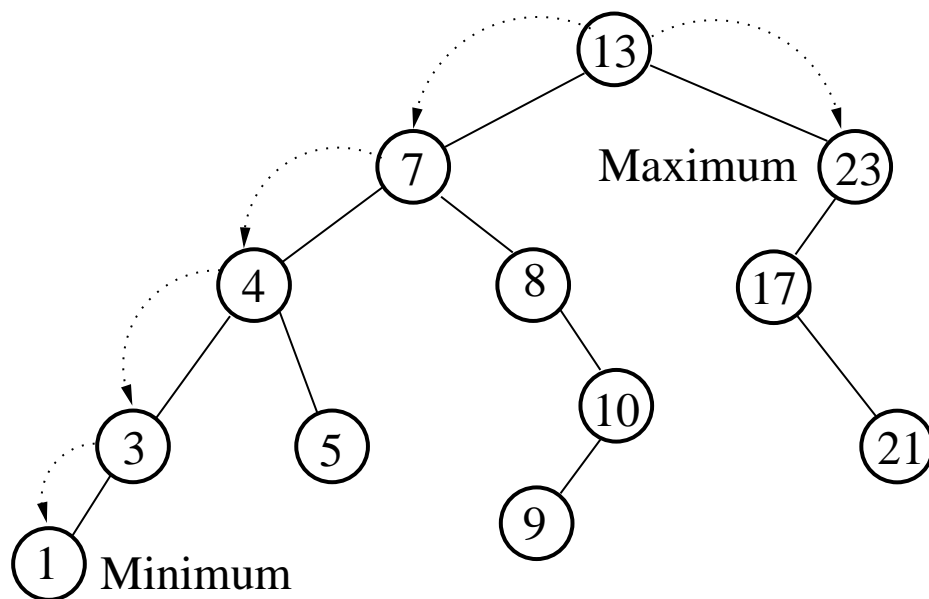
- Given a binary search tree, there are several operations we want to perform.
 - **Insert** an element
 - **Delete** an element
 - **Search** for an element
 - Find the **minimum/maximum** element
 - Find the **successor/predecessor** of a node.
- Once we see how these are done, it will be apparent that the complexity of each of these is $\Theta(h)$, where h is the height of the tree.
- The insert and delete operations are the hardest to implement.
- Finding the minimum/maximum and searching are the easiest, so we will start with these.

BST: Minimum/Maximum

- The minimum element is the left-most node, the maximum element is the right-most.
- The following functions find the min/max element in the BST rooted at x .

```
node *Find_Min(x) {  
  while(x->left!=NULL)  
    x=x->left;  
  return x;  
}
```

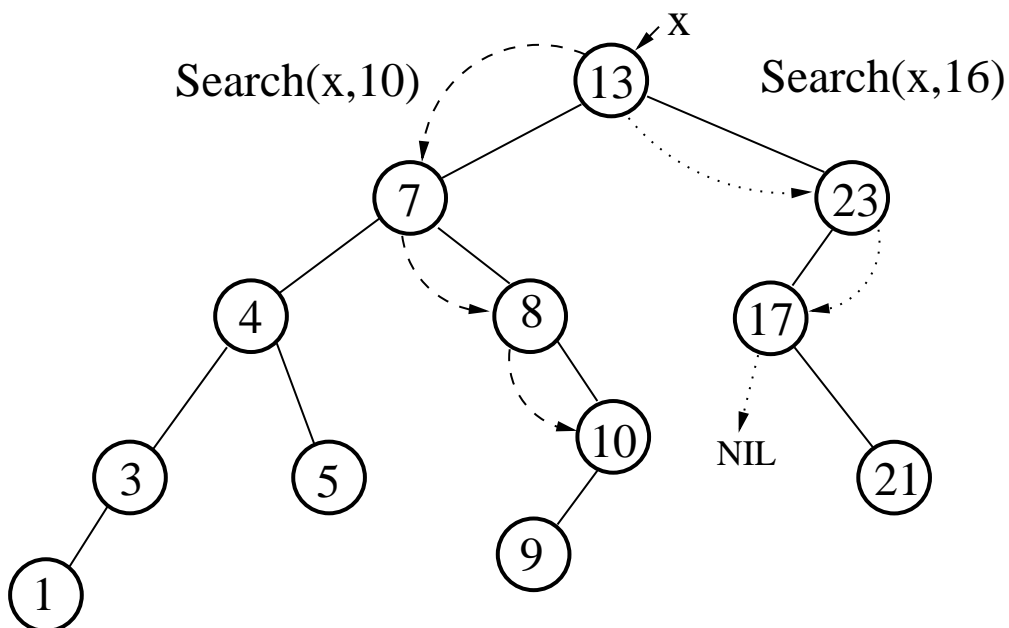
```
node *Find_Max(x) {  
  while(x->right!=NULL)  
    x=x->right;  
  return x;  
}
```



BST: Searching

- Searching for an element in a binary search tree is very straightforward:
- The following function searches for the value k in the tree rooted at x .

```
node *Search(node *x, int k) {  
    while(x != NULL && k != x->key) {  
        if(k < x->key)  
            x = x->left;  
        else  
            x = x->right;  
    }  
    return x;  
}
```



BST: Successor/Predecessor

- Finding the Successor/Predecessor of a node is a little harder.
- To find the successor y of a node x (if it exists)
 - If x has a nonempty right subtree, then y is the smallest element in the tree rooted at $x \rightarrow \text{right}$. Why?
 - If x has an empty right subtree, then y is the lowest ancestor of x whose left child is also an ancestor^a of x . (Of course!)

```
node *Successor(node *x) {  
    if(x->right != NULL)  
        return Find_Min(x->right);  
    y = x->parent;  
    while(y != NULL && x == y->right) {  
        x = y;  
        y = y->parent;  
    }  
    return y;  
}
```

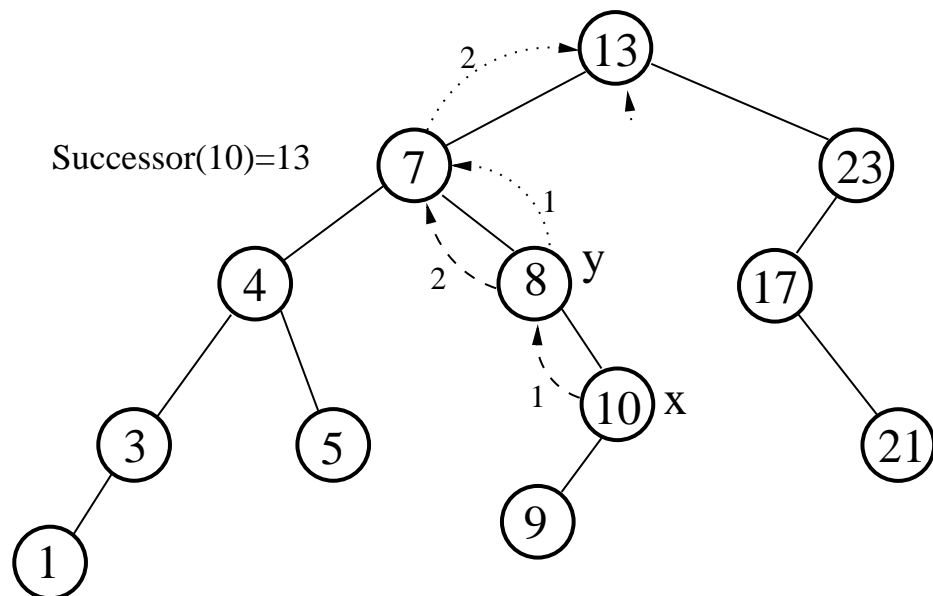
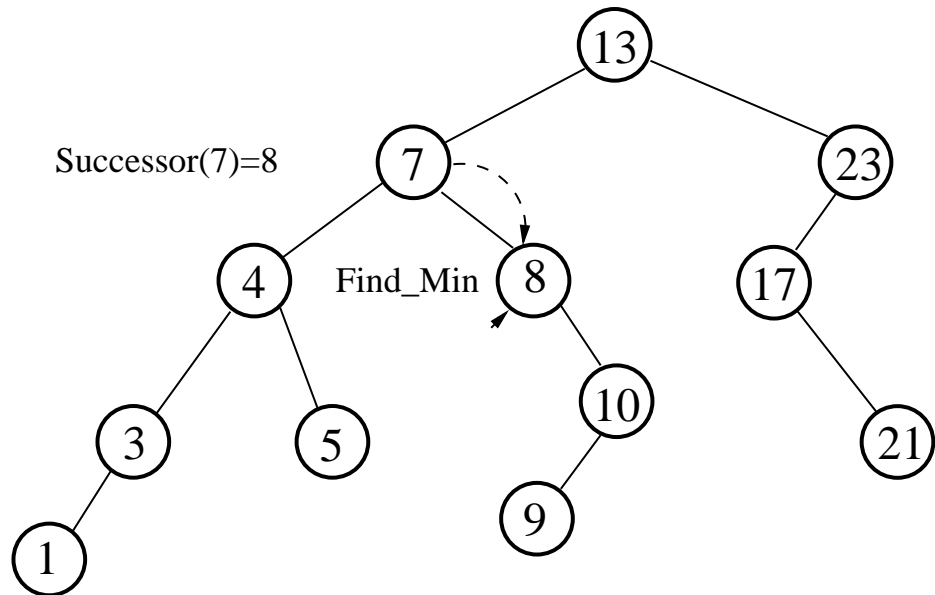
- The predecessor operation is symmetric to successor.

^a x is one of its own ancestors

BST: Successor Argument

- So, why is it that if x has an empty right subtree, then y is the lowest ancestor of x whose left child is also an ancestor of x ?
- Let's look at it the other way.
- Let y be the lowest ancestor of x whose left child is also an ancestor of x .
- What is the predecessor of y ?
- Since y has a left child, it must be the largest element in the tree rooted at $y \rightarrow \text{left}$
- If x is not the largest element in the subtree rooted at $y \rightarrow \text{left}$, then some ancestor of x (in the subtree) is the left child of its parent.
- But y , which is not in this subtree, is the lowest such node.
- Thus x is the predecessor of y , and y is the successor of x .

BST: Successor Examples

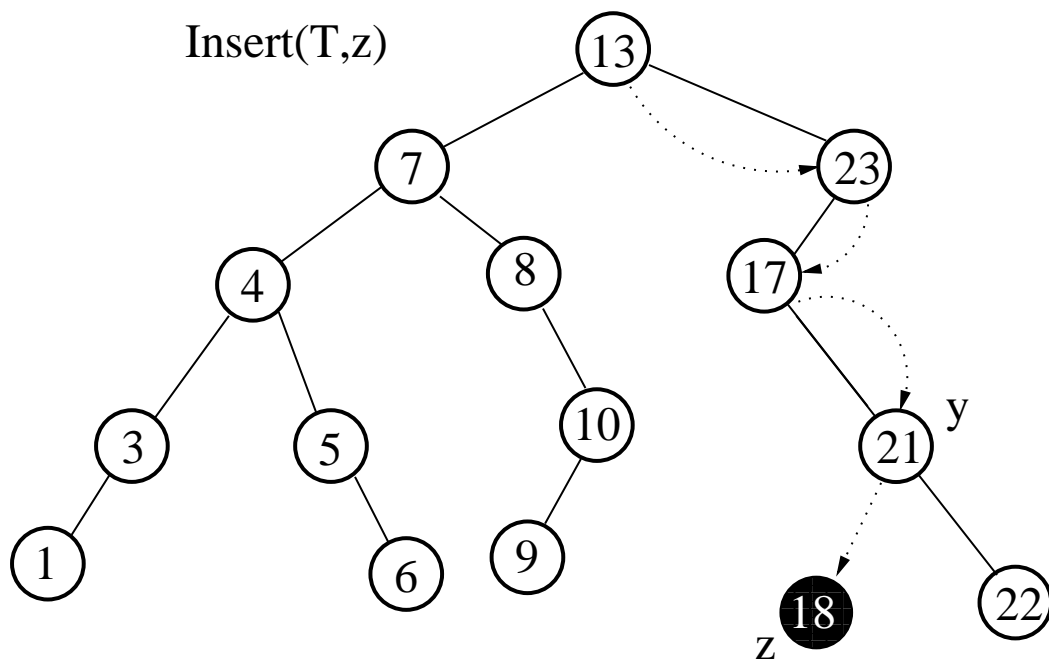
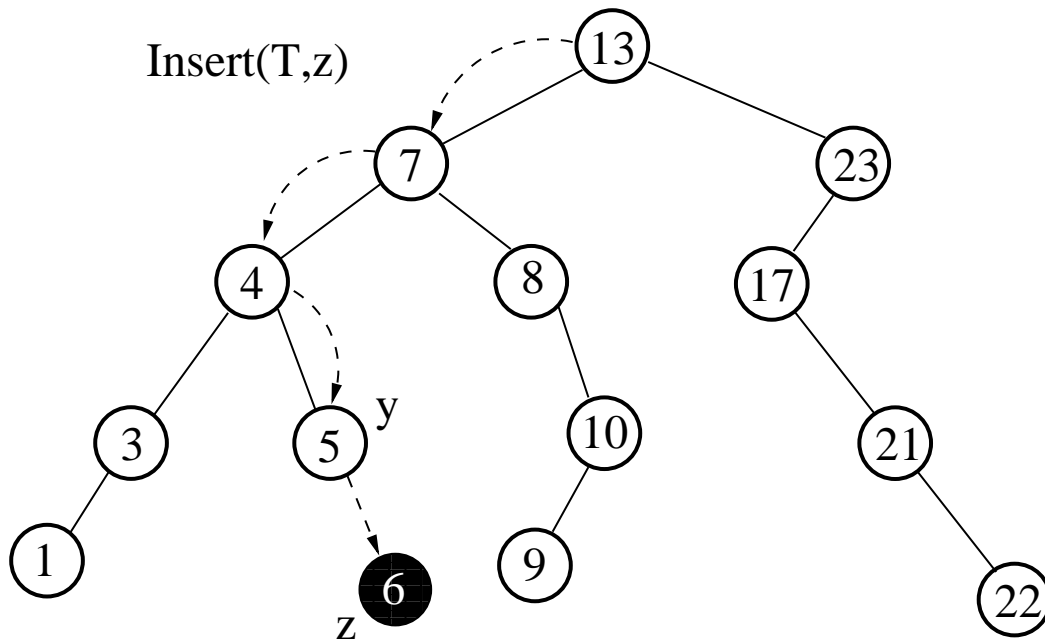


BST: Insertion

- To insert a node into a binary tree, we search the tree until we find a node whose appropriate child is NULL. We insert the new node there.
- T is the tree, and z the node we wish to insert.

```
Insert(T,z) {
    node *y=NULL;
    node *x=T.root;
    while(x != NULL) {
        y = x;
        if(z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }
    z->parent = y;
    if(y == NULL)
        T.root = z;
    else
        if(z->key < y->key)
            y->left = z;
        else
            y->right = z;
}
```


BST: Insertion Example



BST: Deletion

- Deleting a node z is by far the most difficult operation.
- There are 3 cases to consider:
 - If z has no children, just delete it.
 - If z has one child, splice out z . That is, link z 's parent and child.
 - If z has two children, splice out z 's successor y , and replace the contents of z with the contents of y .
- The last case works because if z has 2 children, then its successor has no left child. Why?
- Deletion is made worse by the fact that we have to worry about boundary conditions
- To simplify things, we will first define a function called **SpliceOut**.

BST: Splice Out

- Any node with ≤ 1 child can be “spliced out”.
- Splicing out a node involves linking the parent and child of a node.
- The algorithm:

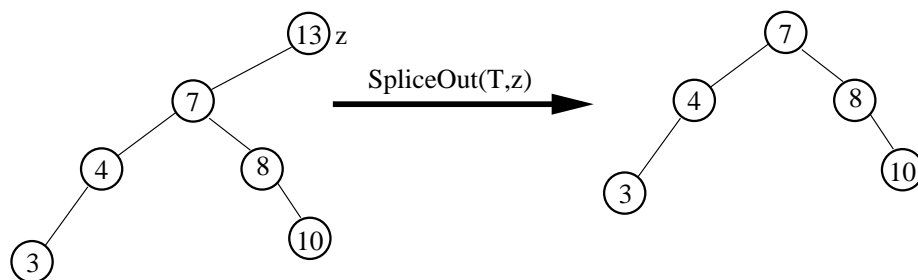
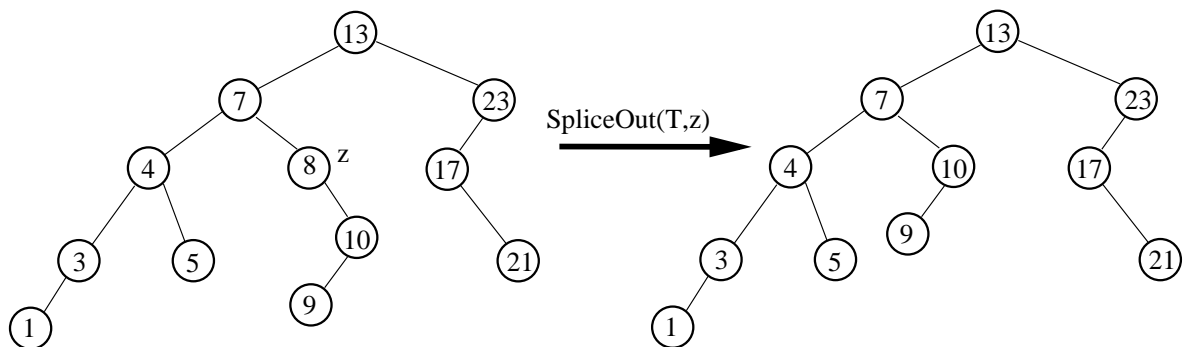
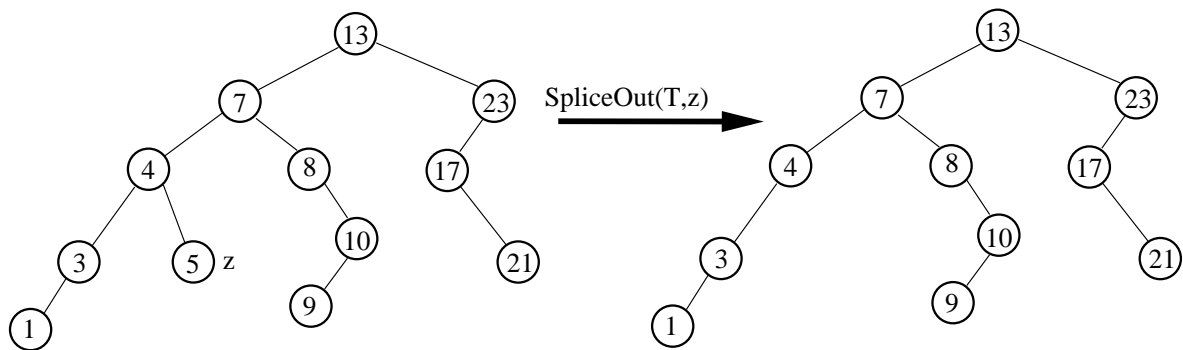
```
SpliceOut(tree T, node *y) {
    if(y->left != NULL && y->right != NULL)
        return;    //Two children; can't splice out

    if(y->left != NULL)        //Locate child of y
        x = y->left;
    else if (y->right != NULL)
        x = y->right;
    else
        x = NULL;

    if(x != NULL)    //If y has child, set parent
        x->parent = y->parent;

    //Set y's parent's child to y's child
    if(y->parent == NULL)
        T.root = x;
    else {
        if(y == y->parent->left)
            y->parent->left = x;
        else
            y->parent->right = x;
    }
}
```

BST: SpliceOut Examples

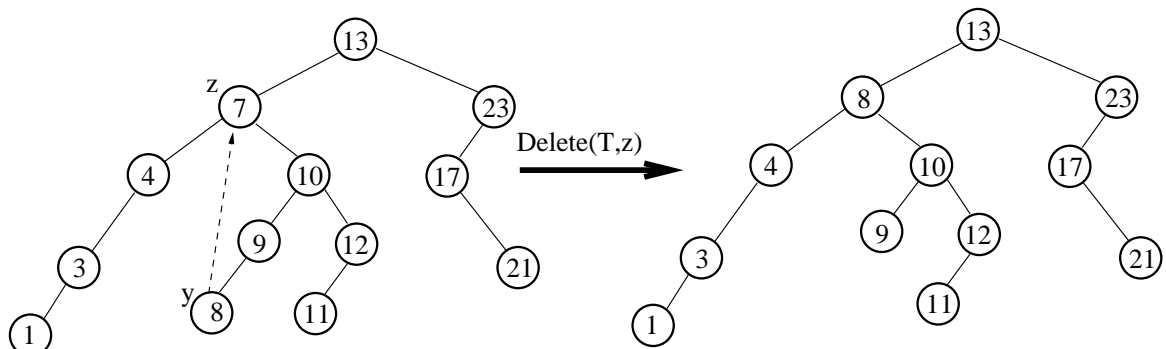
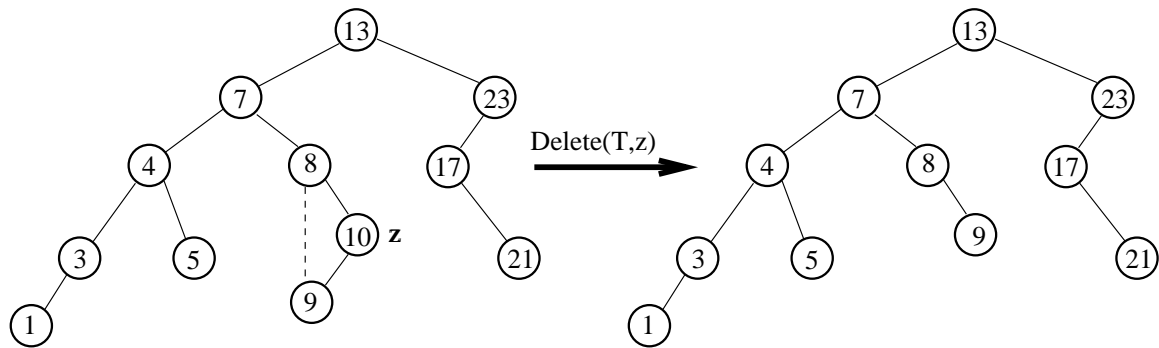
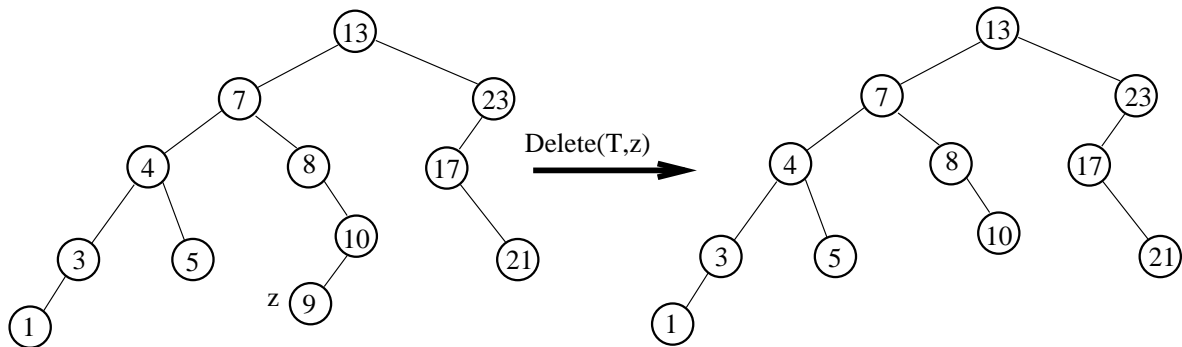


BST: Deletion Algorithm

- Once we have defined the function SpliceOut, deletion looks simple.
- Here is the algorithm to delete z from tree T .

```
Delete(tree T, node *z) {  
    if(z->left == NULL || z->right == NULL)  
        SpliceOut(T,z);  
    else {  
        y = Successor(z);  
        z->key = y->key;  
        SpliceOut(T,y);  
    }  
}
```

BST: Deletion Examples

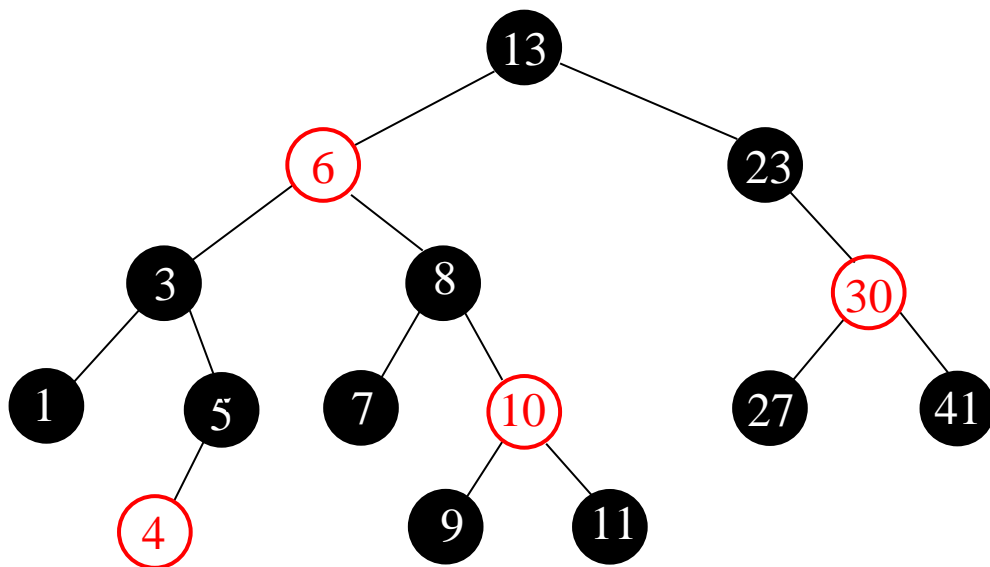


BST: Time Complexity

- We stated earlier, and have now seen, that all of the BST operations have time complexity $\Theta(h)$, where h is the height of the tree.
- However, in the worst case, the height of a BST is $\Omega(n)$, where n is the number of nodes.
- In this case, the BST has gained us nothing.
- To prevent this worst-case behavior, we need to develop a method which ensures that the height of a BST is kept to a minimum.
- **Red-Black Trees** are binary search trees which have height $\Theta(\log n)$.

Red-Black Trees

- A **red-black tree** is a binary search tree with the following properties:
 - Each node is colored either **red** or **black**.
 - Every leaf (NULL) is black.
 - If a node is red, both its children are black.
 - Every simple path from a node to a descendent leaf has the same number of black nodes.



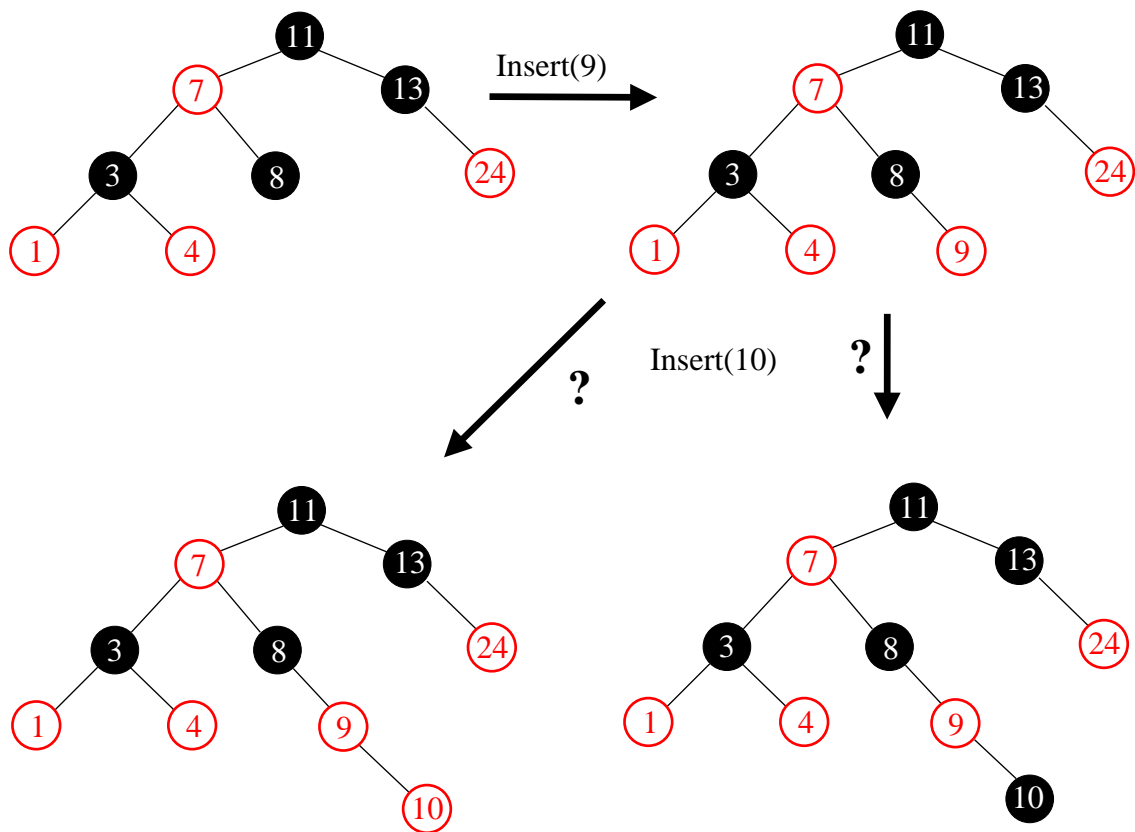
- The leaf nodes are all empty and black, so we will omit them in the figures.
- When we talk about the nodes in a red-black tree, we will mean the internal nodes.

Red-Black Trees Fact and Terms

- The **black-height** of a node x is the number of black nodes, not including x , on a path to any leaf.
- A red-black tree with n nodes has height at most $2 \log(n + 1)$.
- Since **red-black trees** are binary search trees, all of the operations that can be performed on binary search trees can be performed on them.
- Furthermore, the time complexity will be the same— $O(h)$ —where h is the height.
- Unfortunately, insertion and deletion as defined for regular binary search trees will not work for red-black trees. Why not?
- Fortunately, insertion and deletion can both be modified so that they work, and still have time complexity $O(h)$.

Insert and Delete in RB Trees

- Here are a few examples of why inserting a node into a red-black tree is not trivial.



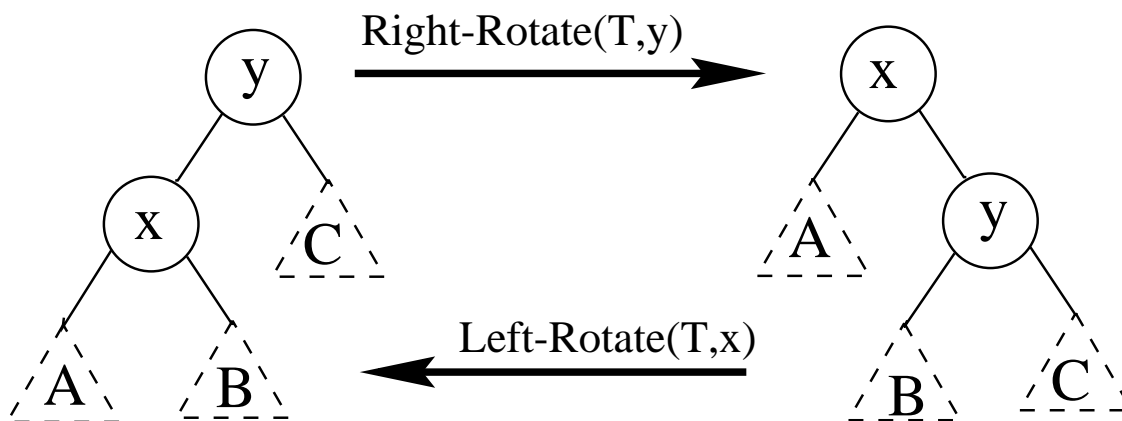
- Similar things happen when we try to delete nodes.
- We will not discuss in depth these operations.
- We will discuss some of the concepts, however.

Red-Black Tree Insertion: Method

- To insert a node x into a red-black tree, we do the following:
 - Insert x with the standard BST *Insert*.
 - Color x red.
 - If x 's parent is red, fix the tree.
- Notice that x 's children, *NULL*, are black.
- Since we colored x red, we have not changed the black height.
- The only problem we have is (possibly) having a red node with a red child.
- Fixing the tree involves re-coloring some of the nodes and performing rotations.

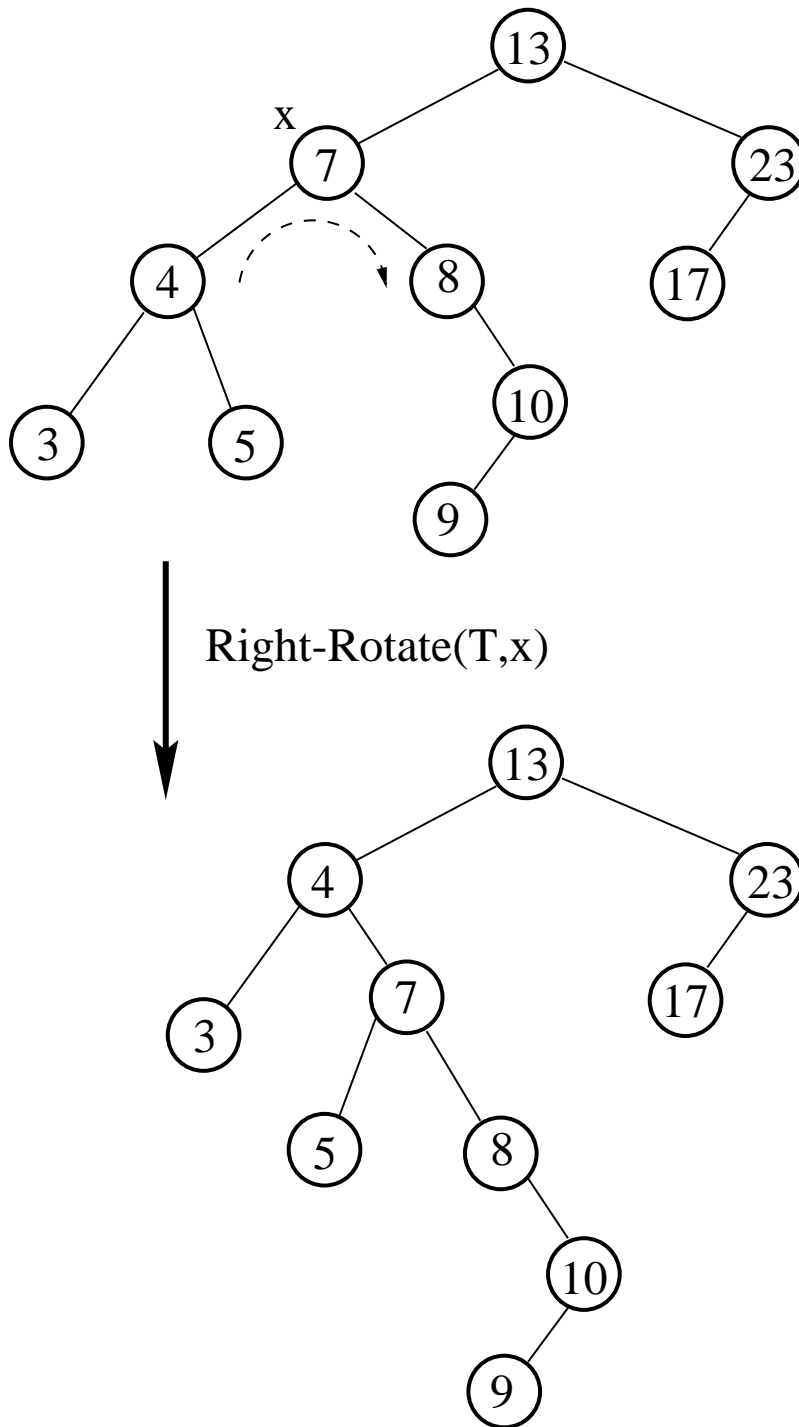
Left- and Right-Rotations

- Rotations are best defined by an illustration:

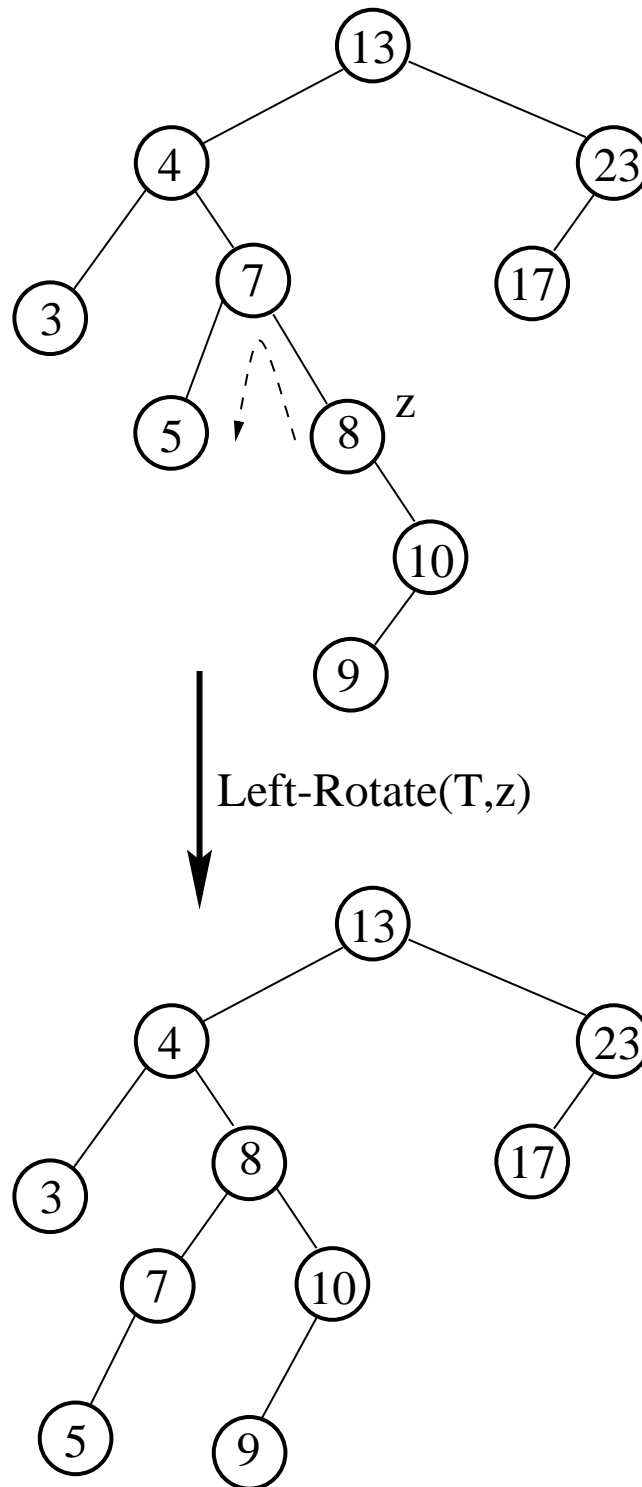


- Here, the letters A , B , and C represent arbitrary subtrees. They could even be empty.
- It is not too hard to see that the binary search tree property will still hold after a rotation.

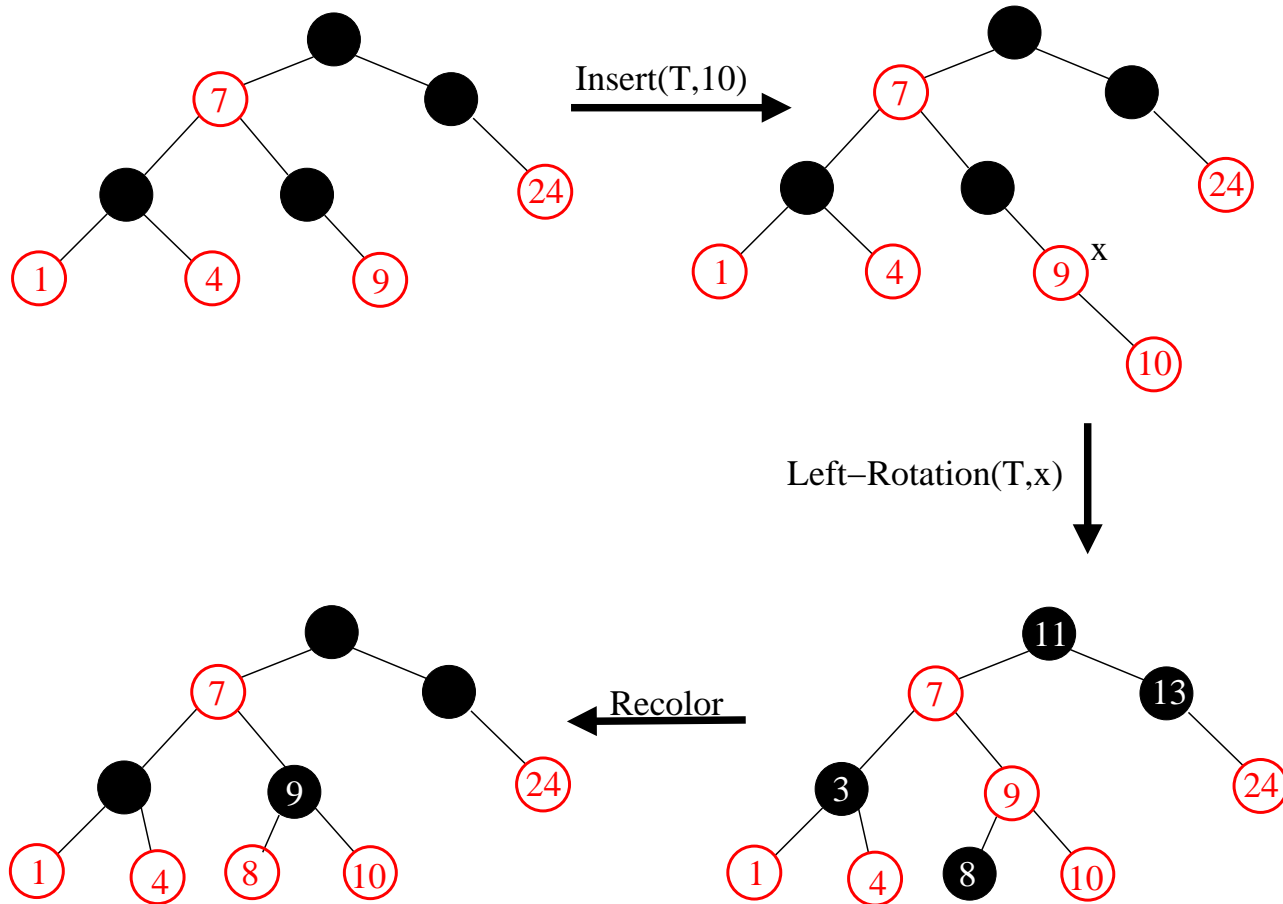
Rotation Example



Rotation Example



Insertion Example



Red Black Tree Summary

- Red-black trees are binary search trees which have height $\Theta(\log n)$ guaranteed.
- The basic operations can all be implemented in time $O(\log n)$.
- Although inserting and deleting nodes only requires time $O(\log n)$, they are nonetheless not trivial to implement.
- A regular binary search tree does not guarantee time complexity of $O(\log n)$, only $O(h)$, where h can be as large as n .
- Thus red-black trees are useful if one wants to guarantee that the basic operations will take $O(\log n)$ time.