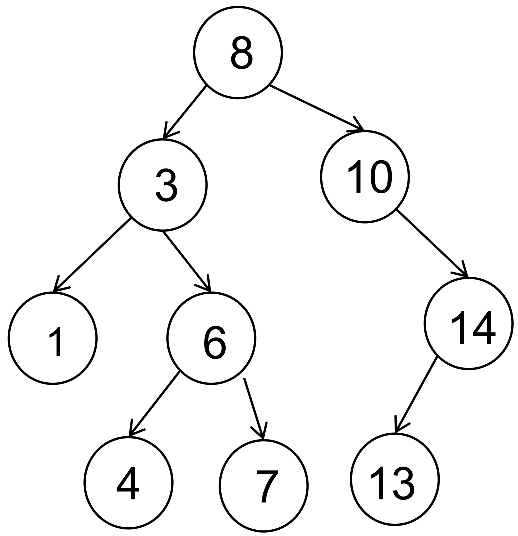
DS:
Binary Search Tree

Liwei

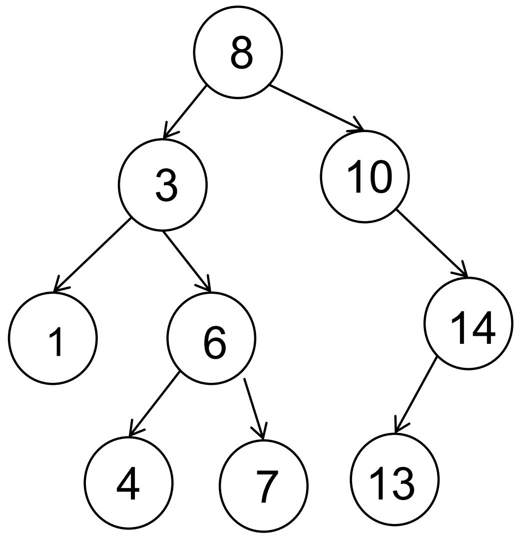
What is a binary search tree?

- Binary Search Tree is a Binary Tree in which all the nodes follow the bellow-mentioned properties
  - The left sub-tree of a node has a key less than or equal to its parent node's key
  - The right sub-tree of a node has a key greater than its parent node's key



What is a binary search tree?

- ***Binary-search property***
- Let x be a node in a binary search tree.
- If y is a node in the left subtree of x, then key[y] $\leq$ key[x].
- If y is a node in the right subtree of x, then key[x] $\leq$ key[y].



Why should we learn tree

	Array	Linked List	Tree
Creation	O(1)	O(1)	?
Insertion	O(n)	O(n)	
Deletion	O(n)	O(n)	
Searching	O(n)	O(n)	
Traversing	O(n)	O(n)	
Deleting entire	O(1)	O(1)	
Space Efficienct?	No	Yes	
Implementation	Easy	Moderate	

Common operations of BST

- Creation of BST
- Search for a value
- Traverse all nodes
- Insertion of a node
- Deletion of a node
- Deletion of BST

Searching a node in BST

BST_Search(root, value)	
if (root is null)	O(1)
return null	O(1)
else if (root == value)	O(1)
return root	O(1)
else if (value < root)	O(1)
BST_Search (root.left, value)	T(n/2)
else if (value > root)	O(1)
BST_Search (root.right, value)	T(n/2)

Time complexity: O(logn)

TREE_SEARCH(*x*,*k*)

TREE_SEARCH(*x*,*k*)

if $x = nil$ **or** $k = key[x]$

then return *x*

if $k < key[x]$

then return TREE_SEARCH(left[*x*],*k*)

else return TREE_SEARCH(right[*x*],*k*)

Time complexity: O(logn)

ITERATIVE_SEARCH(x,k)

ITERATIVE_SEARCH(x,k)

1 **While** $x \neq nil$ **or** $k \neq key[x]$

2 **do if** $k < key[x]$

3 **then** $x \leftarrow left[x]$

4 **else** $x \leftarrow right[x]$

5 **return** x

MAXIMUM and MINIMUM

TREE_MINIMUM(x)

```
while left[x] ≠ NIL
    do x ← left[x]
return x
```

TREE_MAXIMUM(x)

```
while right[x] ≠ NIL
    do x ← right[x]
return x
```

InOrder Traversal of BST

```
inOrderTraversal(root)
	if (root equals null)
		return error message
	else
		inOrderTraversal(root.left)
		print root
		inOrderTraversal(root.right)
```

Time & Space Complexity

inOrderTraversal(root)	
if (root equals null)	$O(1)$
return error message	$O(1)$
else	$O(1)$
inOrderTraversal(root.left)	$T(n/2)$
print root	$O(1)$
inOrderTraversal(root.right)	$T(n/2)$

Time complexity: $O(n)$

Inorder tree walk

INORDER_TREE_WALK(x)

1 **if** $x \neq nil$

2 **then** INORDER_TREE_WALK(left[x])

3 print key[x]

4 INORDER_TREE_WALK(right[x])

PreOrder Traversal of BST

```
preOrderTraversal(root)
    if (root equals null)
        return error message
    else
        print root
        preOrderTraversal(root.left)
        preOrderTraversal(root.right)
```

Time & Space Complexity

```
preOrderTraversal(root)
    if (root equals null)                    O(1)
        return error message                 O(1)
    else                                     O(1)
        print root                           O(1)
        preOrderTraversal(root.left)         T(n/2)
        preOrderTraversal(root.right)        T(n/2)
```

Time complexity: O(n)

Preorder tree walk (VLR)

PREORDER_TREE_WALK(x)

1 **if** $x \neq nil$

2 print key[x]

3 **then** PREORDER_TREE_WALK(left[x])

4 PREORDER_TREE_WALK(right[x])

PostOrder Traversal of BST

```
postOrderTraversal(root)
        if (root equals null)
                return error message
        else
                postOrderTraversal(root.left)
                postOrderTraversal(root.right)
                print root
```

Time & Space Complexity

```
postOrderTraversal(root)
    if (root equals null)                      O(1)
        return error message                   O(1)
    else                                       O(1)
        postOrderTraversal(root.left)          T(n/2)
        postOrderTraversal(root.right)         T(n/2)
        print root                             O(1)
```

Time complexity: O(n)

Postorder tree walk (LRV)

POSTORDER_TREE_WALK(*x*)

if $x \neq nil$

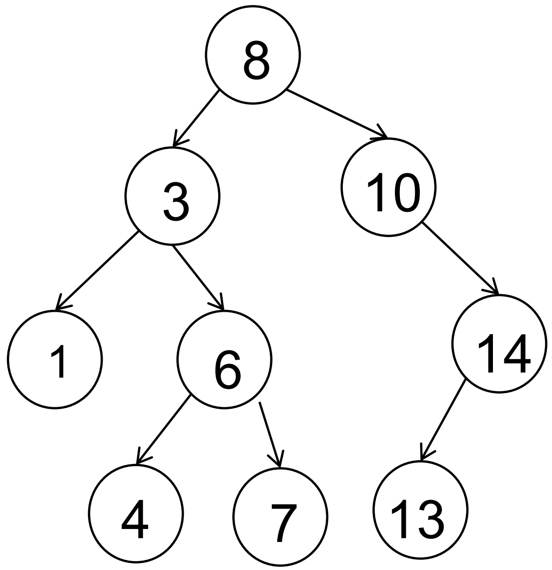
then POSTORDER_TREE_WALK(*left*[*x*])

POSTORDER_TREE_WALK(*right*[*x*])

print *key*[*x*]

Inserting a node in BST

- Case 1: BST is blank
- Case 2: BST is non-blank



Inserting a node in BST

BST_insert (root, value2Insert)	
if(root is null)	O(1)
initiate root with 'value2Insert'	O(1)
else if (value2Insert <= root's value)	O(1)
root.left = BST_Insert(root.left, value2Insert)	T(n/2)
else	O(1)
root.right = BST_Insert(root.right, value2Insert)	T(n/2)
return root	O(1)

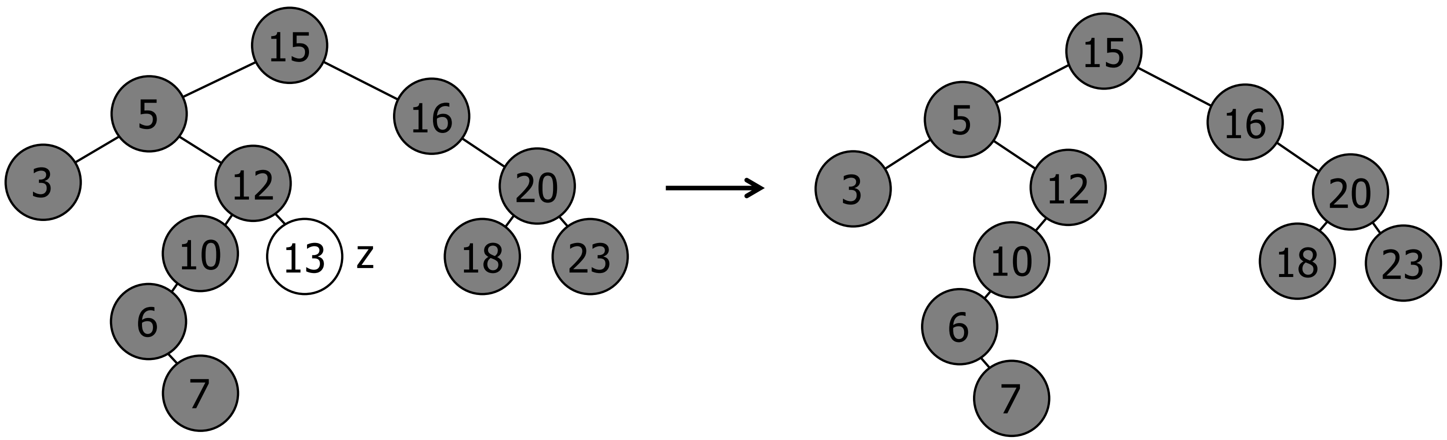
Call: BST_insert (root, 8)

Time complexity: O(logn)

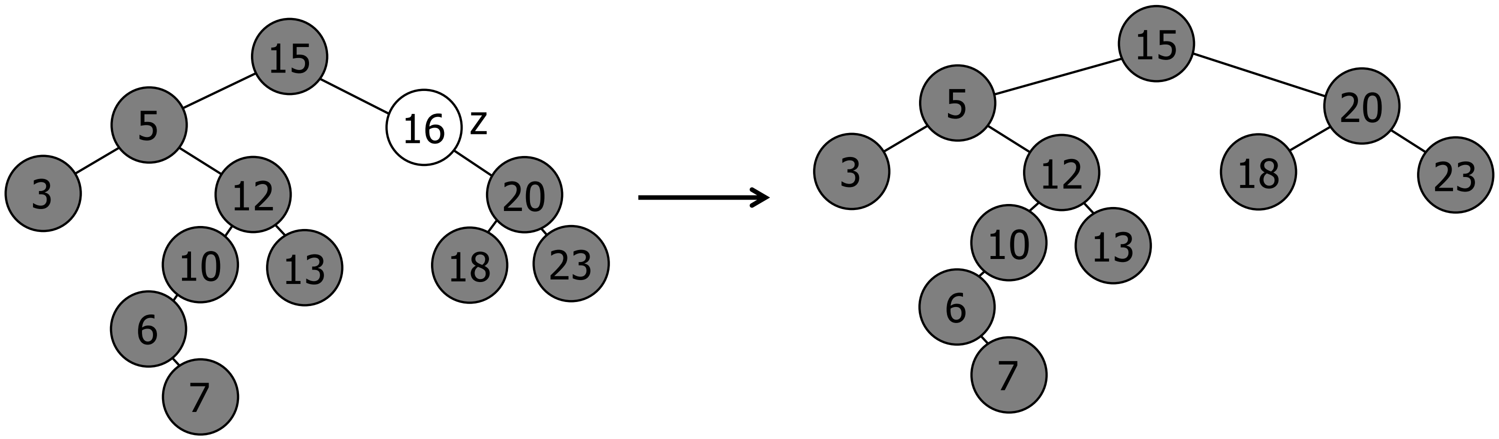
Deleting a node from BST

- Case 1: Node to be deleted is leaf node
- Case 2: Node to be deleted is having 1 child
- Case 3: Node to be deleted is having 2 children

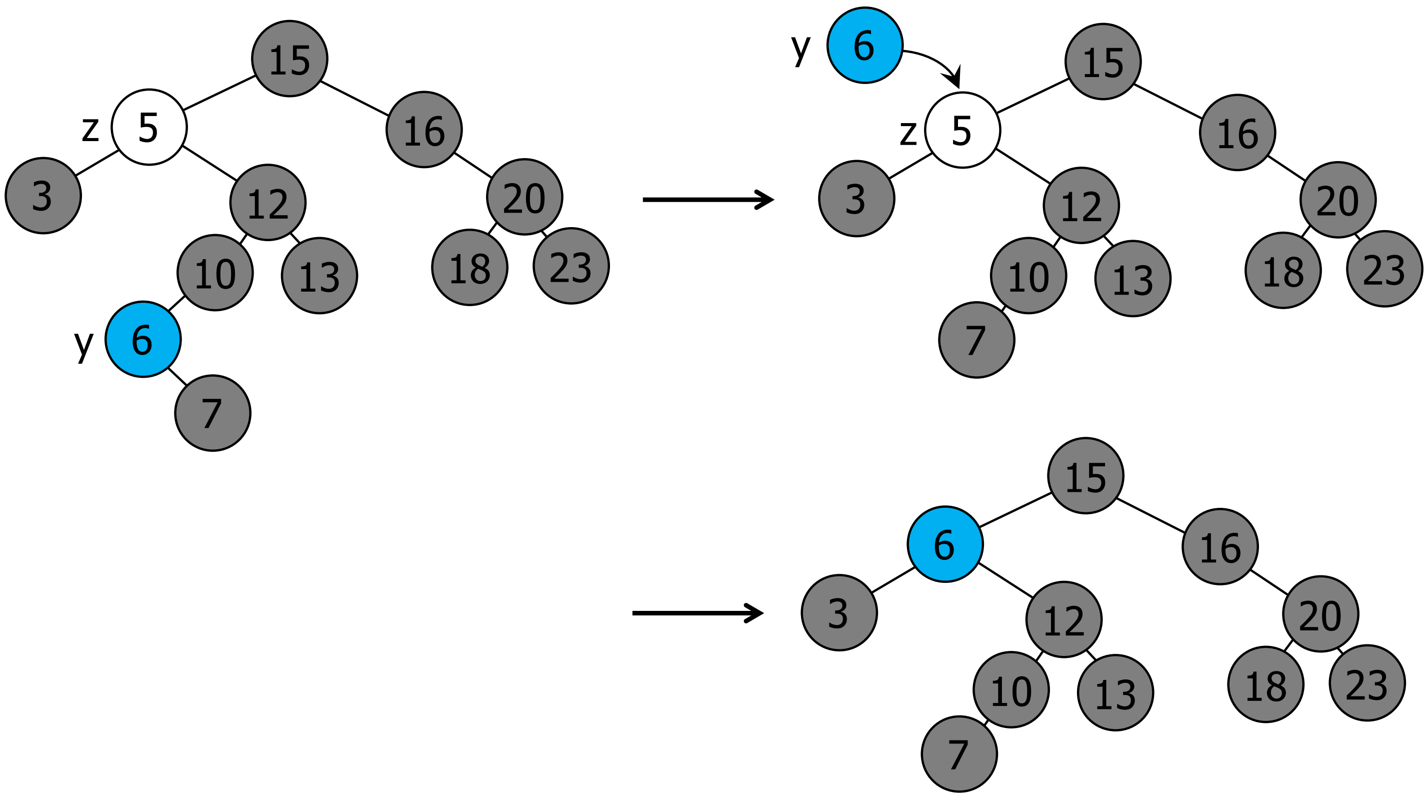
Deletion: z has no children



Deletion: z has only one child



Deletion: z has two children



```
deleteNodeOfBST (root, value2Delete)
    if(root==null) return null                                                        O(1)
    if(value2Delete < root.value)                                                     O(1)
        deleteNodeOfBST (root.left, value2Delete)                                     T(n/2)
    else if (valueToBeDeleted > root.value)                                           O(1)
        deleteNodeOfBST (root.right, value2Delete)                                    T(n/2)
    else // if currentNode is the node be deleted                                     O(1)
        if root have both children, find minimum element from right subtree (case #3) O(log n)
            replace current node (e.g., current root)  with minimum node from right subtree   O(1)
            delete minimum node from right subtree                                    O(1)
        else if root has only left (case #2)                                          O(1)
            root = root.Left()       // replace current node with the left node       O(1)
        else if root has only right (case #2)                                         O(1)
            root = root.Right()     // replace current node with the right node       O(1)
        else      // if root do not have child (Case #1)                              O(1)
            root = null;  // delete the current root                                  O(1)
```

Time complexity: O(logn)

Deletion

Tree-Delete(T,z)

```
if  left[z] = NIL or right[z] = NIL  ▶ one or no child
 then y ← z
      else  y ← Tree-Successor(z)          ▶ two children
if  left[y] ≠ NIL                        ▶ set x to be y's child
      then x ← left[y]
      else x ← right[y]
if x ≠ NIL                               ▶ if at least one child
 then p[x] ← p[y]          ▶ connect the child to its parent
if  p[y] = NIL                              ▶ y is root
 then root[T] ← x            ▶ y will be deleted, x becomes root
    else if y = left[p[y]]
            then left[p[y]] ← x                ▶ connect parent to child
            else  right[p[y]] ← x
if y ≠ z
    then key[z] ← key[y]
       copy y's satellite data into z
  return y
```

Time and Space Complexity of BST

	Time Complexity
Creation of Tree	O(1)
Searching for a value	O(log n)
Traversing Tree	O(n)
Insertion of value in Tree	O(log n)
Deletion of value from Tree	O(log n)
Deleting entire Tree	O(1)