SCJ2013 Data Structure & Algorithms

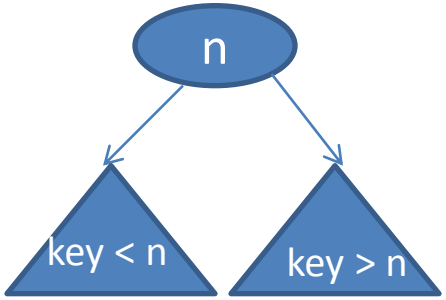
Binary Search Tree

Nor Bahiah Hj Ahmad

Binary Search Tree

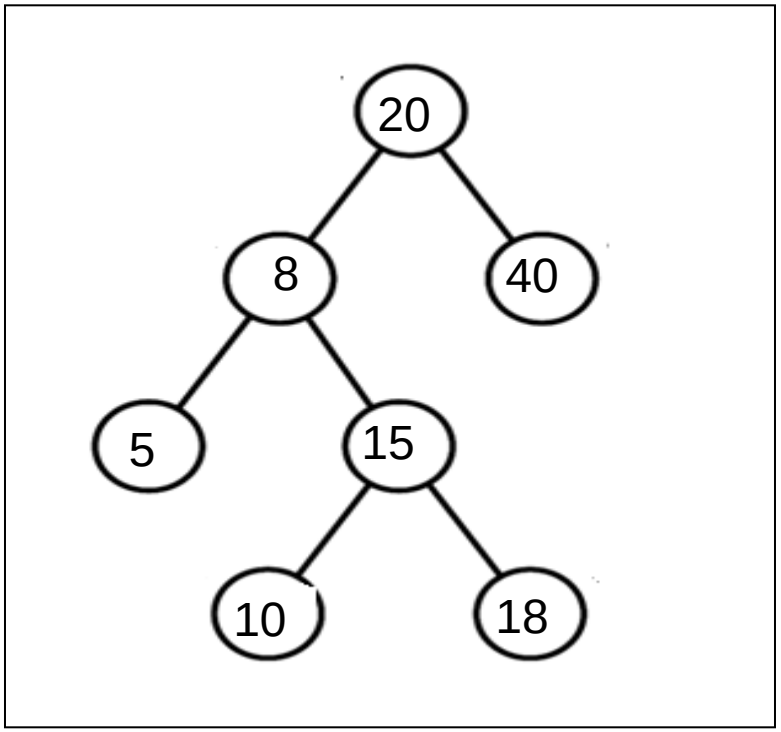
- A binary search tree has the following properties:
  - For every node n in the tree
    - Value of n is greater than all values in n's left subtree.
    - Value of n is less than all values in n's right subtree.
    - Both left subtree and right subtree are also binary search trees

Binary Search Trees

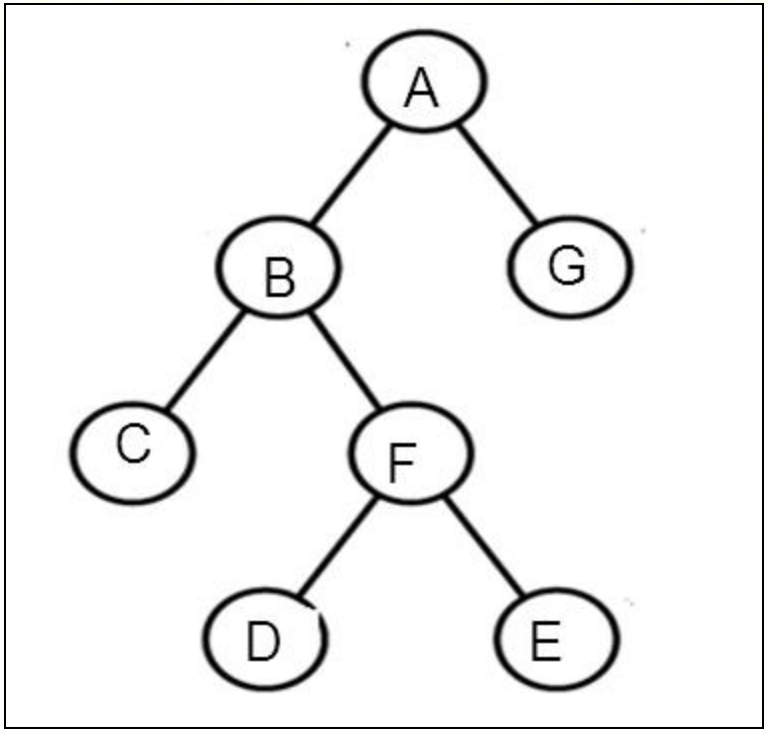


All key in the left sub-tree is less than n

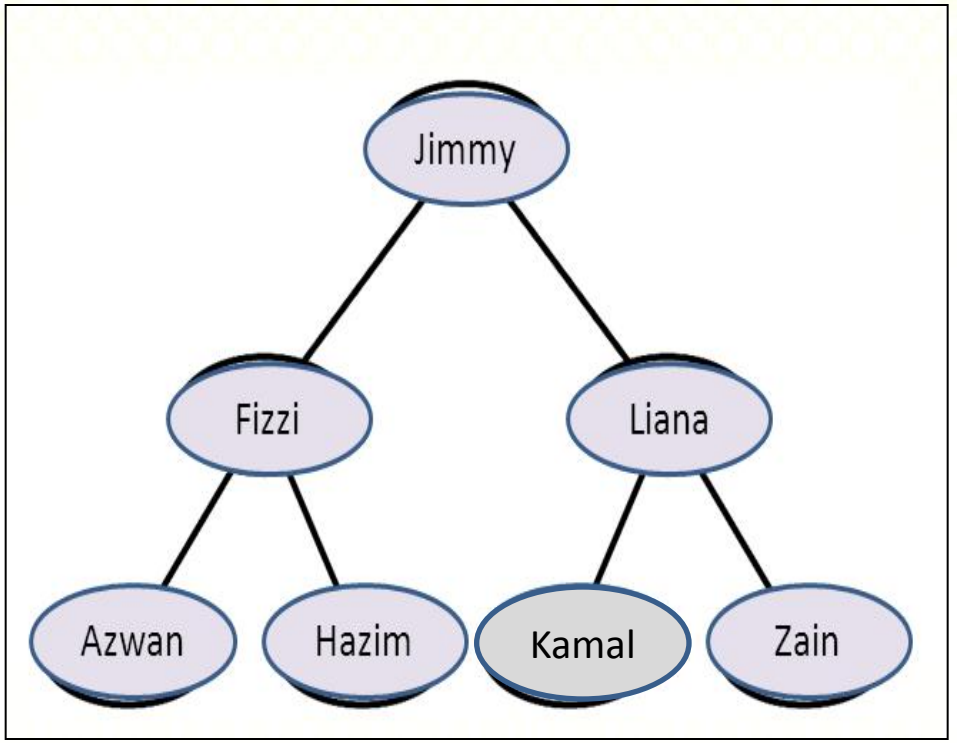
All key in the right sub-tree are greater than n



Binary Search Tree



Not a Binary Search Tree



Binary Search Tree

Advantages of Binary Search Tree

One of the most fundamental algorithms in Computer Science and the method of choice in many applications.

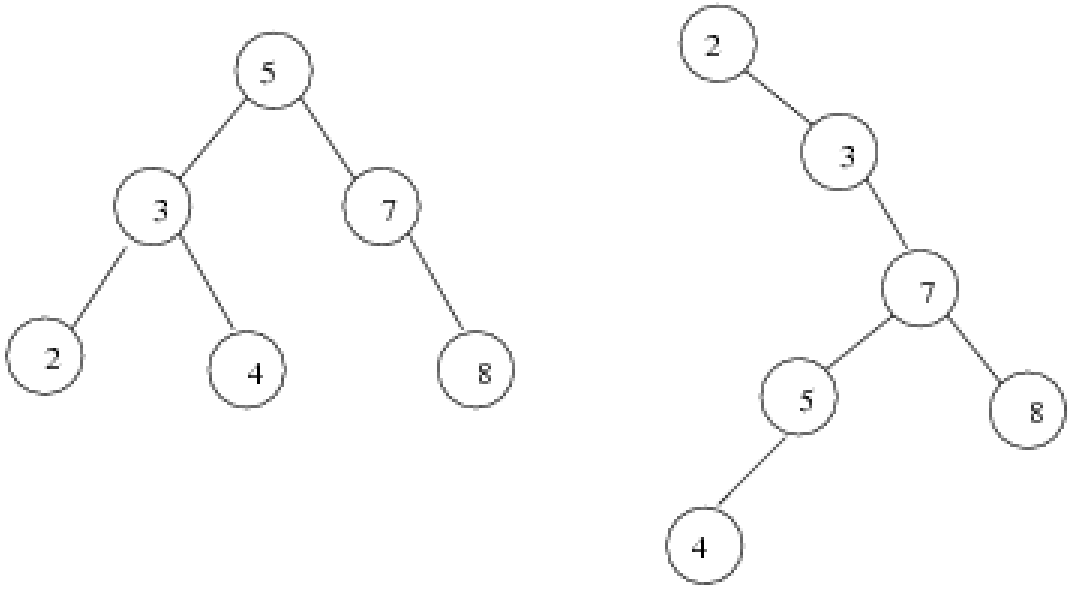
- Stores keys in the nodes in a way that searching, insertion and deletion can be done efficiently.
- Simple Implementation
- Nodes in tree are dynamic

Disadvantages of Binary Search Tree

- The shape of the tree depends on the order of insertions, and it can be degenerated.
- When inserting or searching for an element, the key of each visited node has to be compared with the key of the element to be inserted/found.
- Keys in the tree may be long and the run time may increase.

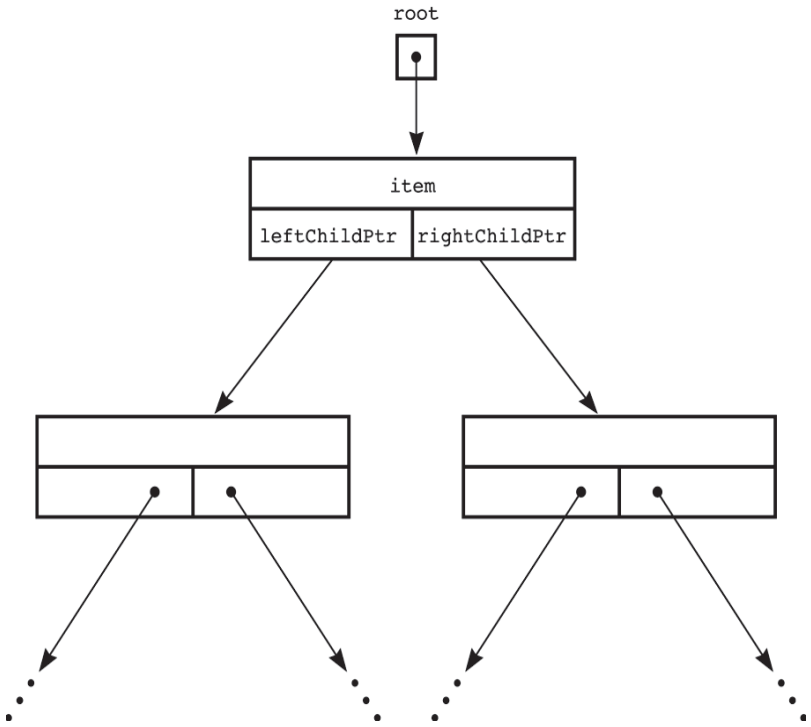
Binary search tree

Two binary search trees representing the same set:



– Average depth of a node is O(log n);
– Maximum depth of a node is O(n)

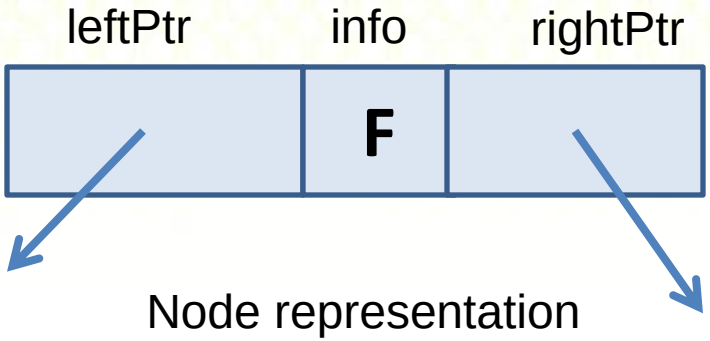
Pointer-based ADT Binary Tree



A pointer-based implementation of a binary tree

- Elements in a binary tree is represented by using nodes.
- Nodes store the information in a tree.
- Each node in the tree must contain at least 3 fields containing:
  - item
  - Pointer to left subtree
  - Pointer to right subtree
- Need 2 declaration in a tree implementation:
  1. Node declaration
  2. Tree declaration

Node Implementation



Node representation

```
typedef char ItemType;
struct TreeNode
{
    itemType info;
    TreeNode *left;
    TreeNode *right;
}
```

info, the node store char value.

left, pointer to left subtree

Right, pointer to right subtree

Tree Implementation

```
class TreeType {
public:
  TreeType();
  ~TreeType();
  bool IsEmpty()const;
  int NumberOfNodes()const;
  void RetrieveItem(ItemType&,bool& found);
  void InsertItem(ItemType);
  void DeleteItem(ItemType);
  void PrintTree() const;
private:
  TreeNode * root;
};
```

The tree class declaration above, declare the binary serach tree using class TreeType.

The tree can be accessed using root, which is a pointer to root of the tree.

Tree Operations

Among the tree operations in the class TreeType:

1. Initialize tree , using constructor.
2. Destroy tree, destructor.
3. Check for empty tree, IsEmpty().
4. Count number of nodes in the tree.
5. Search item in the tree
6. Insert item into a tree.
7. Delete item from tree.
8. Print all item in the tree (Inorder traversal)

Tree Constructor

```
TreeType::TreeType()
{     root = NULL;    }
```

The constructor create an empty tree by initializing root to NULL value.

Tree Destructor

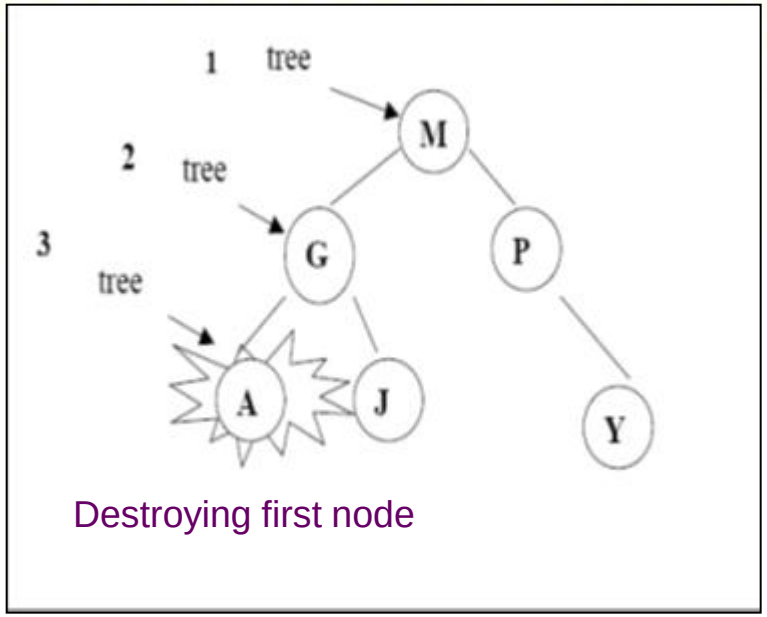
```
TreeType::~TreeType()
{   Destroy(root);   }
void Destroy(TreeNode *& tree)
{   if (tree!=NULL)
    {   Destroy(tree->left);
        Destroy(tree->right);
        delete(tree);
     }
}
```

Destructor will destroy all the nodes in the tree

Function `Destroy()` is implemented recursively whereby the function will destroy all nodes in the left subtree first, followed by destroying nodes in the right subtree. Lastly, the root node will be destroyed.

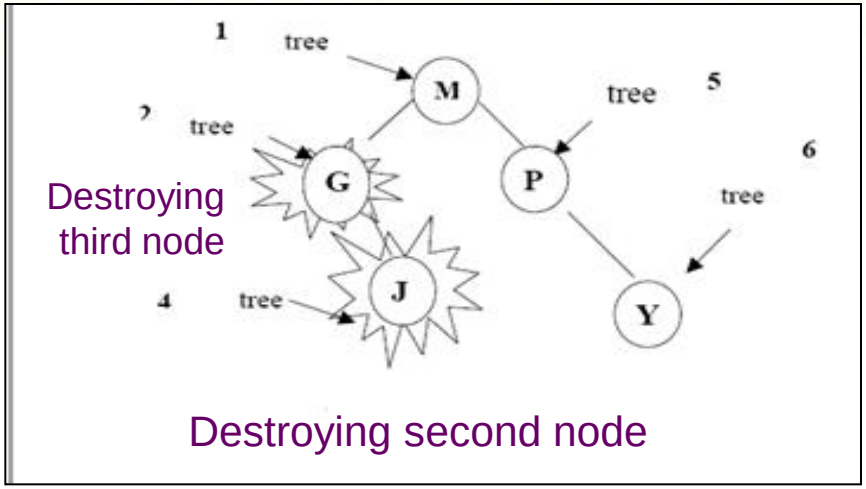
Destroying a tree using destructor

Step 1



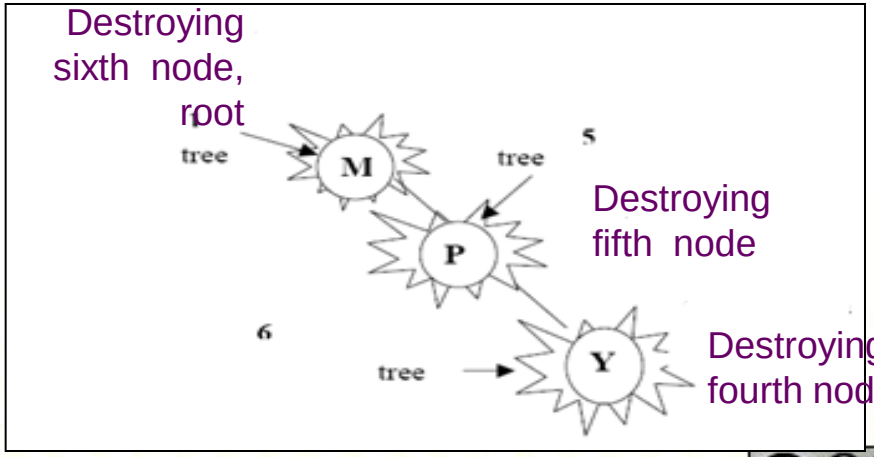
Destroying first node

Step 2



Destroying second node

Destroying sixth node, root



Destroying fifth node

Destroying fourth node

Step 3

`IsEmpty()` Function

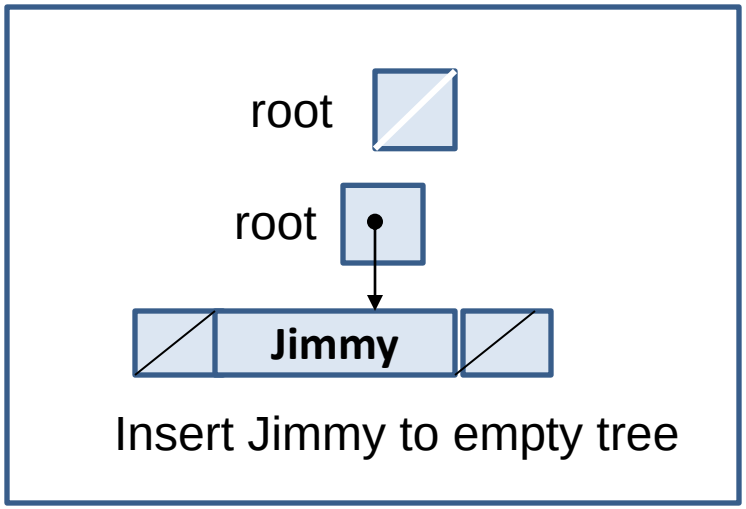
- Binary Search Tree is Empty when the root has NULL value.
- Function IsEmpty() will return True if root is NULL and will return False if the root is not NULL.

```
bool IsEmpty() const
{ if (root == NULL)
    return True; // tree is empty
  else
    return False; // tree is not empty
}
```

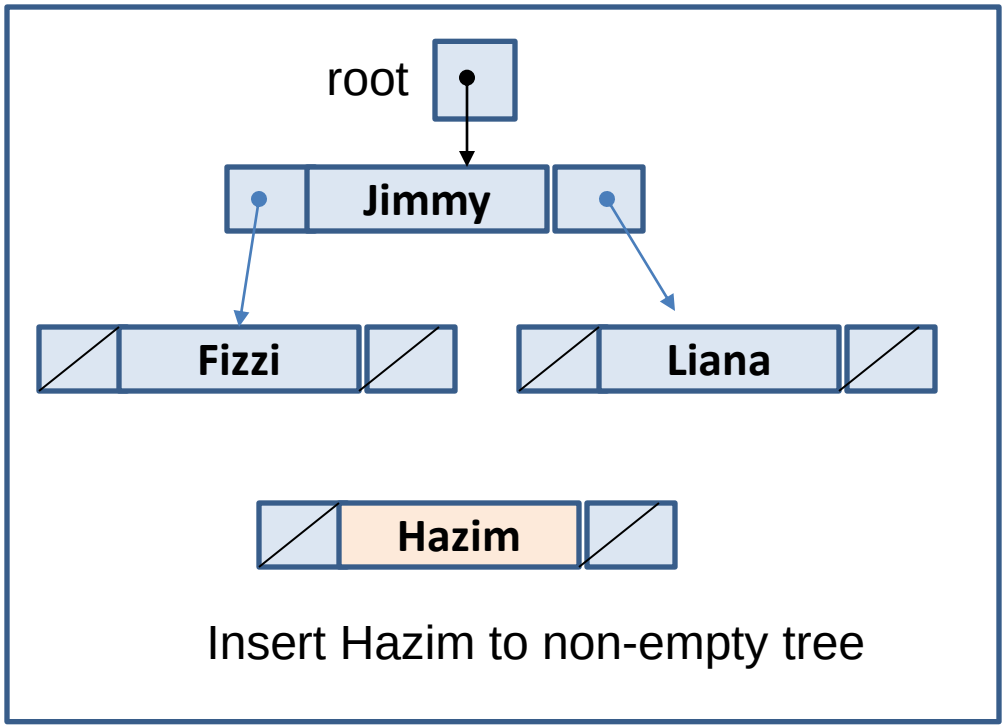
Insert Node Into a Binary Search Tree

- The insert operation will insert a node to a tree and the new node will become leaf node.
- Before the node can be inserted into a BST, the position of the new node must be determined. This is to ensure that after the insertion, the BST characteristics is still maintained.
- Steps to insert a new node in BST
  1. Find the position of the new node in the tree.
  2. Allocate new memory for the new node.
  3. Set NULL value to left and right pointer.
  4. Assign the value to be stored in the tree.

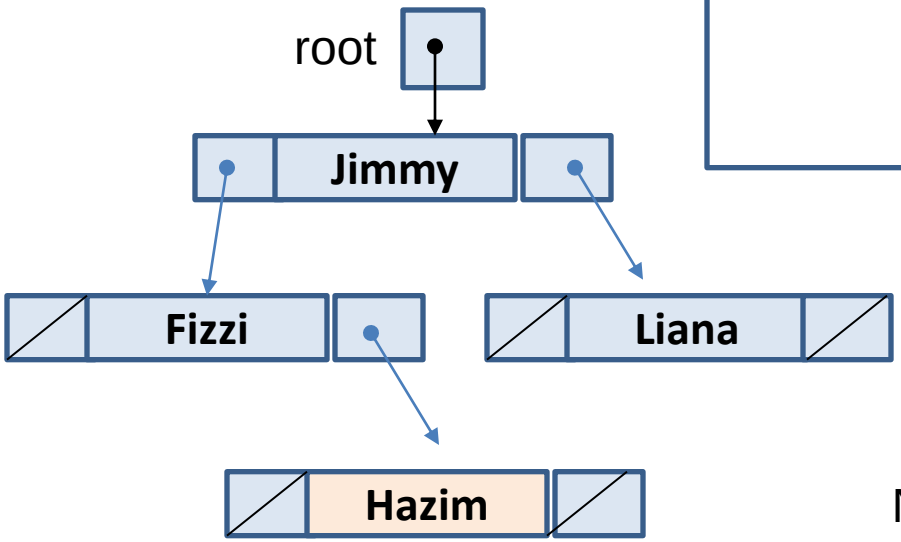
ADT Binary Search Tree: Insertion



Insert Jimmy to empty tree



Insert Hazim to non-empty tree



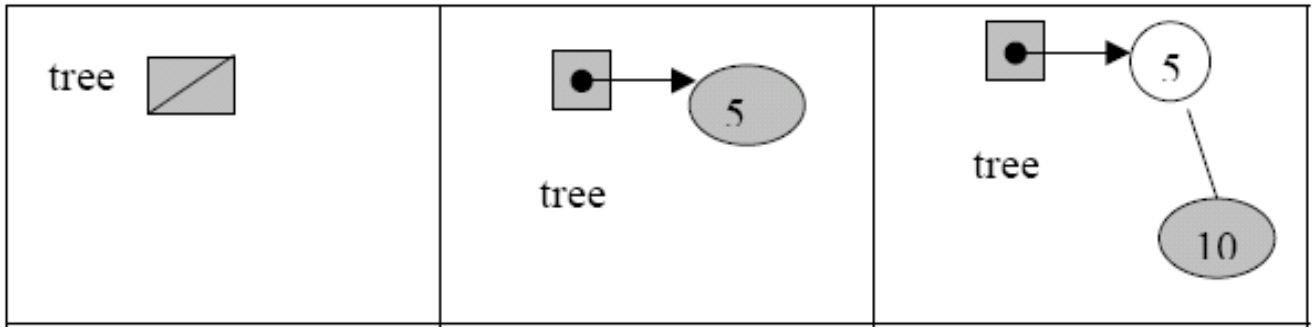
Node with value Hazim becomes a leaf.

Insert new node Implementation

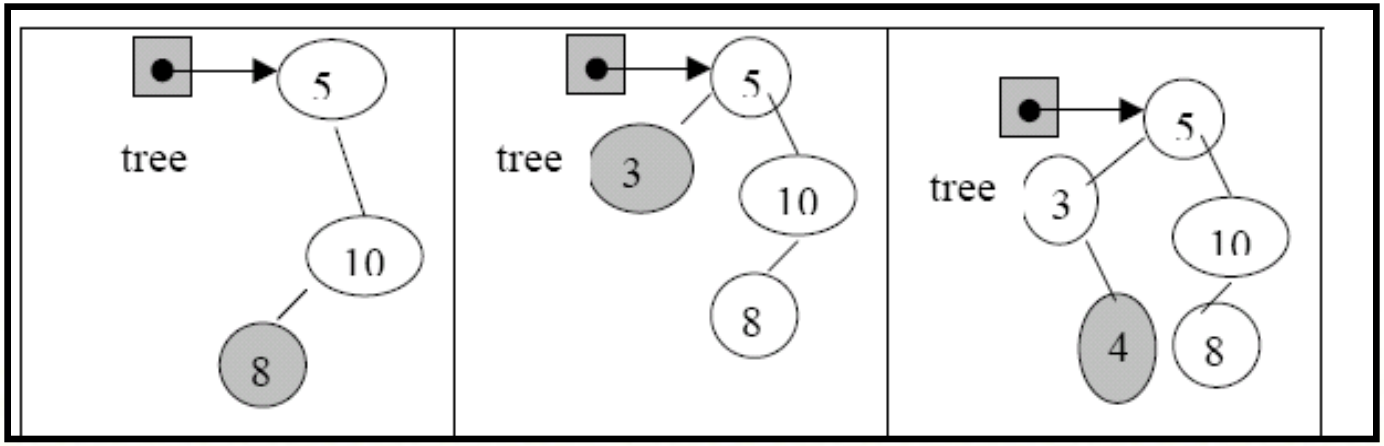
```
void TreeType::InsertItem(ItemType item)
{ Insert(root, item);}

void Insert(TreeNode*& tree, ItemType item)
{   if (tree == NULL) { // base case
          tree = new TreeNode;
          tree->right = NULL;
          tree->left = NULL;
          tree->info = item;
}
else if (item < tree->info)
         Insert(tree->left, item);
else
         Insert(tree->right, item);
}
```

Insert 5, 10, 8, 3 , 4 and 15 in a BST

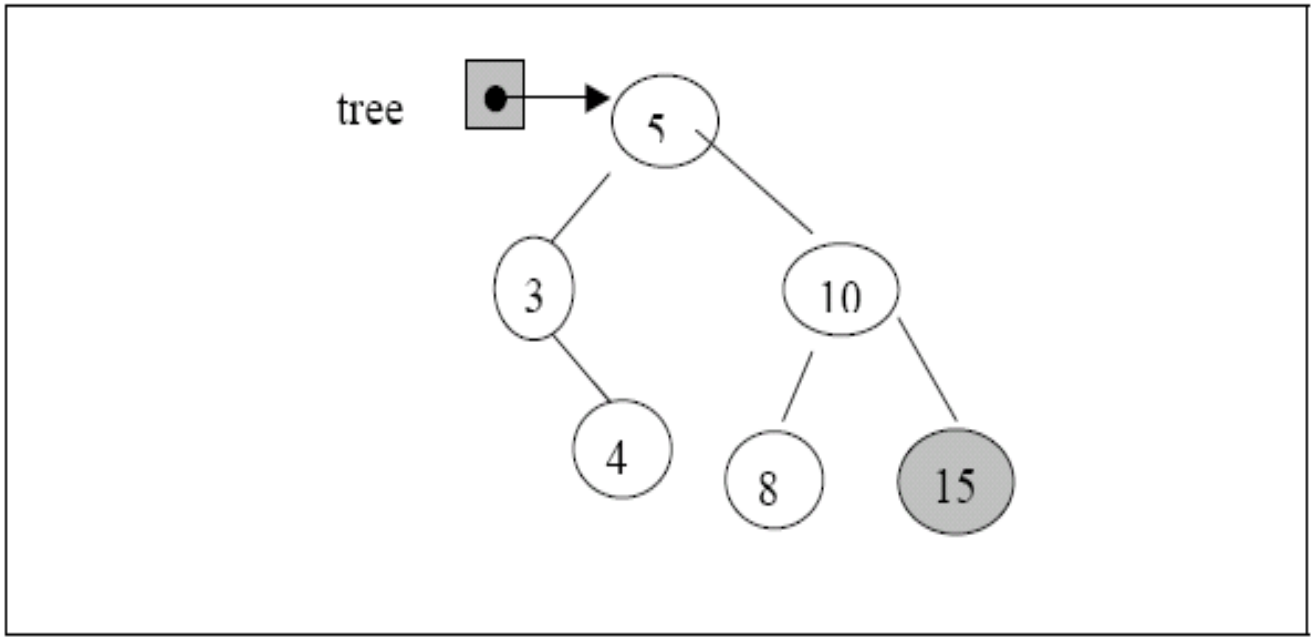


Empty tree Insert 5 Insert 10



Insert 8 Insert 3 Insert 4

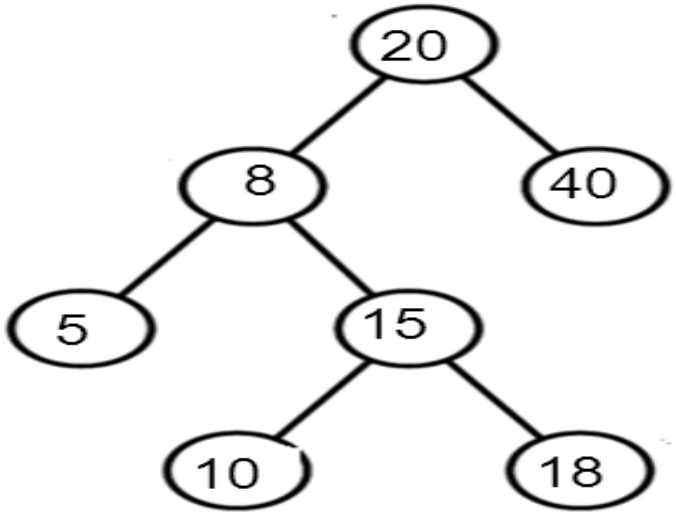
Insert 5, 10, 8, 3 , 4, 15 to a tree



Finally, Insert the last node; 15.

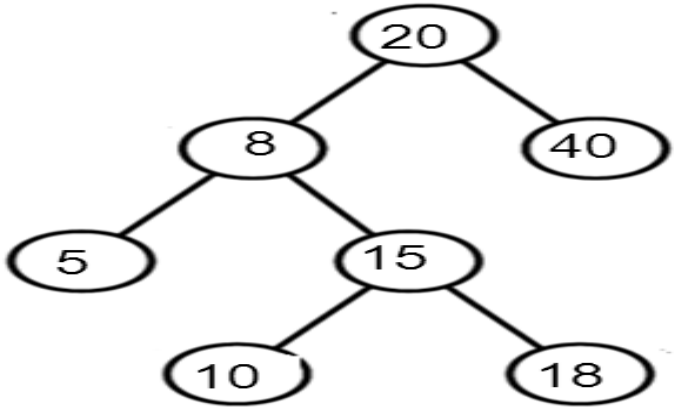
Time complexity = O(height of the tree)

Searching from BST



- From the figure, if we search for value 20, ***then we are done at the root.***
- If we search for a value < 20, then searching will be at the left subtree.
- If we are searching for a value > 20, then searching will be at the right subtree.

Searching from BST



Search value 10 from the tree

1. Compare 10 with value 20 (at root), go to left subtree.
2. Compare 10 with 8, go to right subtree.
3. Compare 10 with 15, go to left subtree.
4. Compare 10 with 10, value is found.

Time complexity = O(height of the tree)

= O(4)

Searching: RetrieveItem() function

```
void TreeType:: RetrieveItem
                (ItemType& item,bool& found)
{ Retrieve(root, item, found);}

void Retrieve(TreeNode* tree,
                 ItemType& item,bool& found)
{ if (tree == NULL) // base case 2
    found = false;
 else if(item < tree->info)
    Retrieve(tree->left, item, found);
 else if(item > tree->info)
    Retrieve(tree->right, item, found);
 else {  // base case 1
    found = true;
 }
```

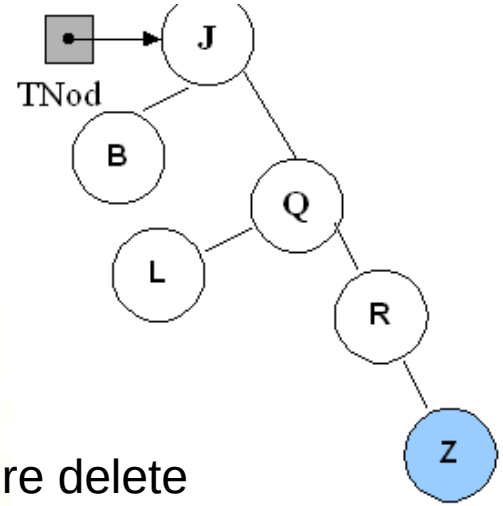
Delete a Node from a Tree

- When a node is deleted, the children of the deleted node must be taken care of to ensure that the property of the search tree is maintained.
- There are 3 possible cases to delete a node in a tree:
  1. Delete a leaf
  2. Delete a node with one child
  3. Delete a node that has two children

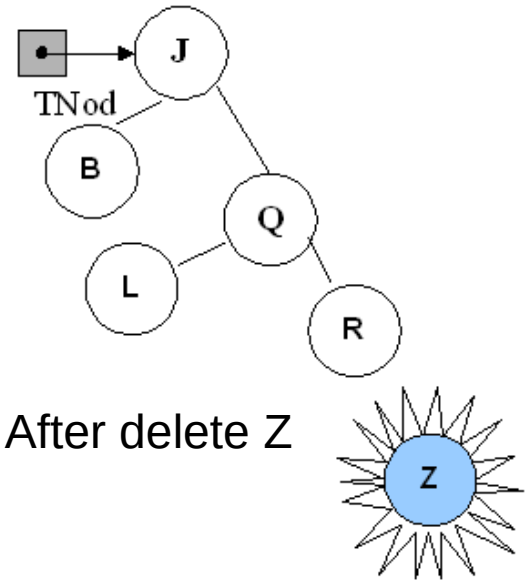
Delete a Leaf Node

The node to be deleted is a leaf

- Set the pointer in N's parent to NULL and delete it immediately
- Example : Delete leaf Node: Z



Before delete

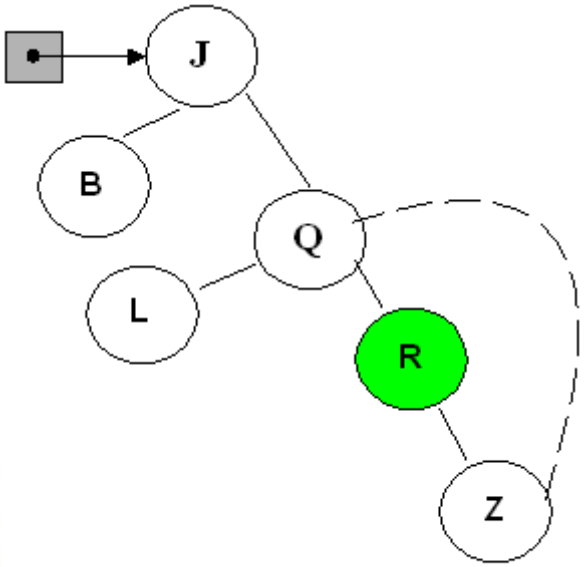


After delete Z

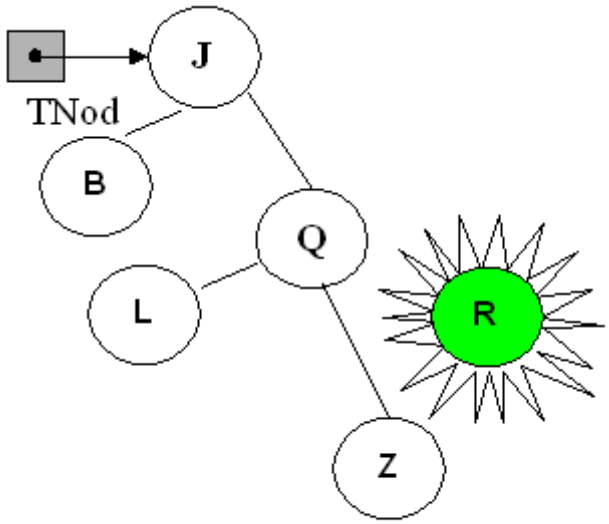
Delete a Node With One Child

Delete node R.

Adjust a pointer from the parent to bypass that node



Before delete R



After delete R

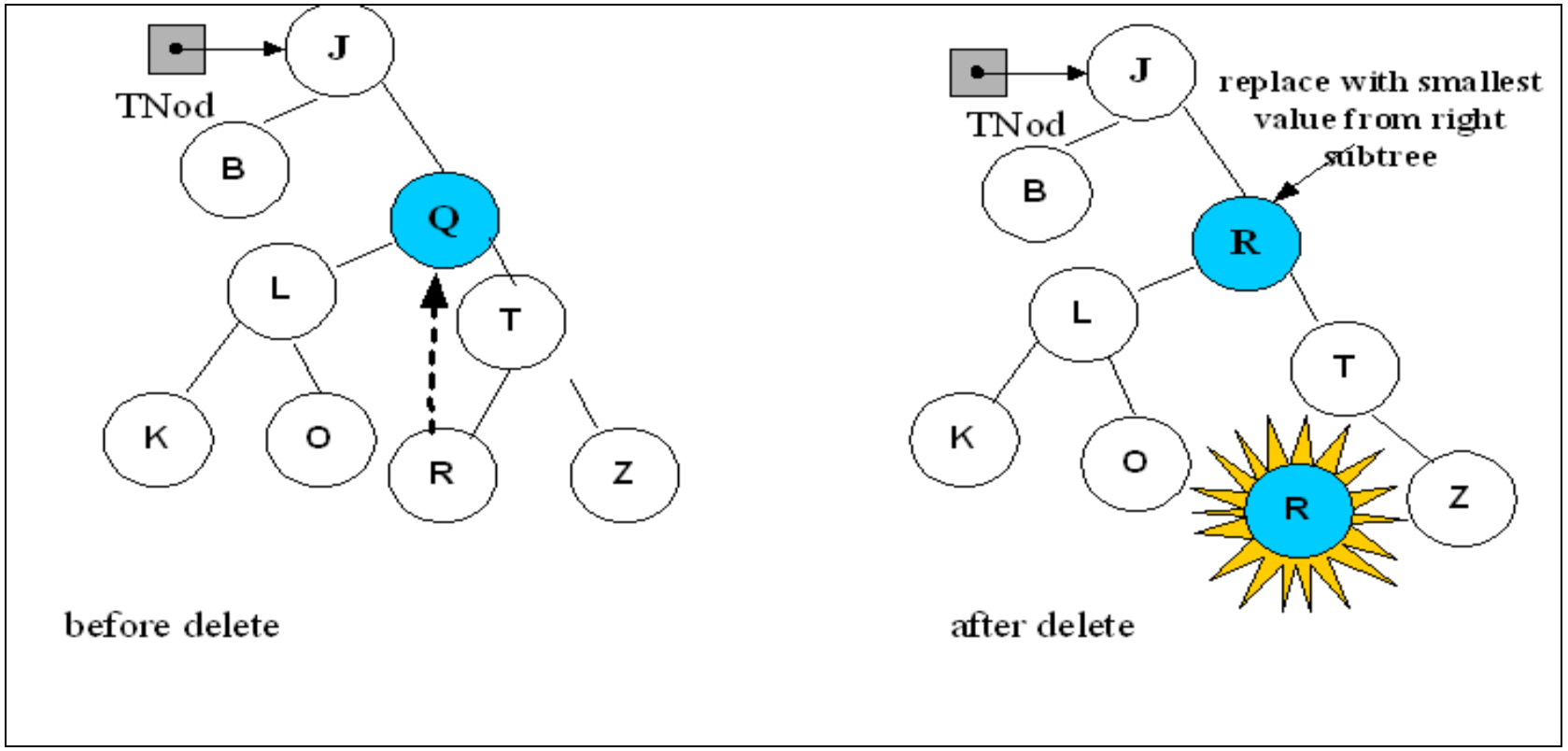
Delete a Node With Two Children

To delete a node N that has two children.

- Locate another node *M* that is easier to delete
  - M is the leftmost node in N's right subtree
  - M will have no more than one child
  - M's search key is called the inorder successor of N's search key
- Copy the item that is in M to N
- Remove the node M from the tree

Delete a Node with 2 Children

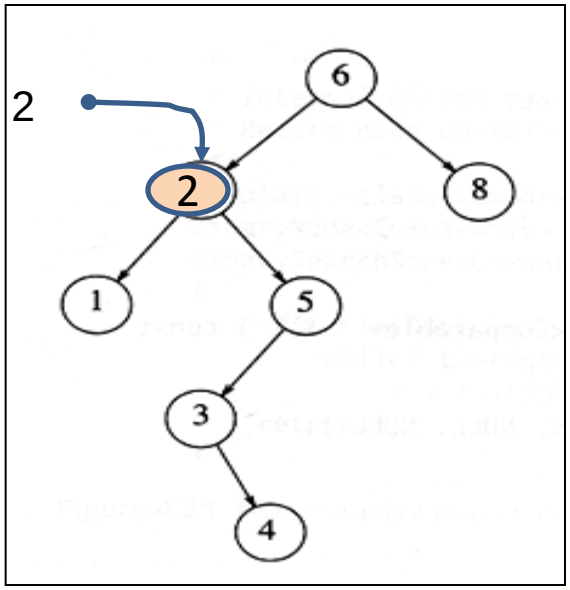
Delete Q that has 2 children

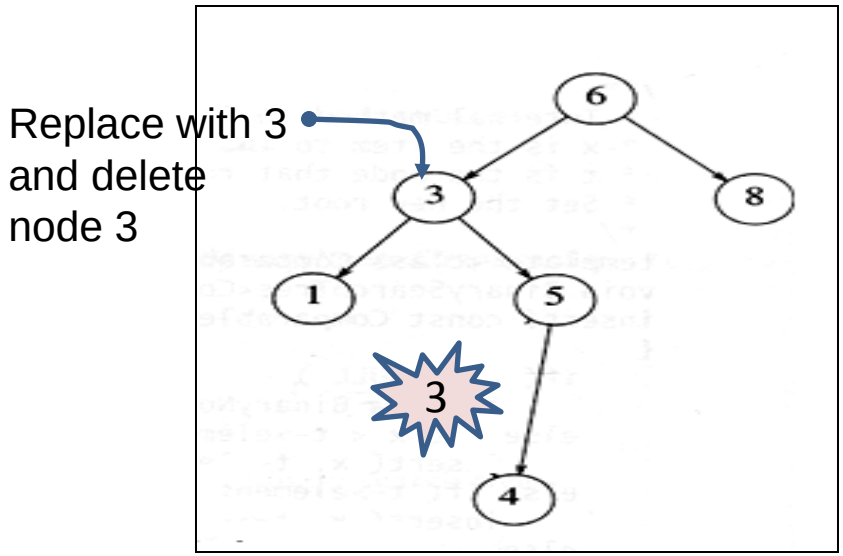


Delete a Node With 2 Children

Delete 2 with 2 children.

- Replace the key of that node with the minimum element at the right subtree.
- Delete the minimum element that has either no child or only right child because if it has a left child, that left child would be smaller and would have been chosen.





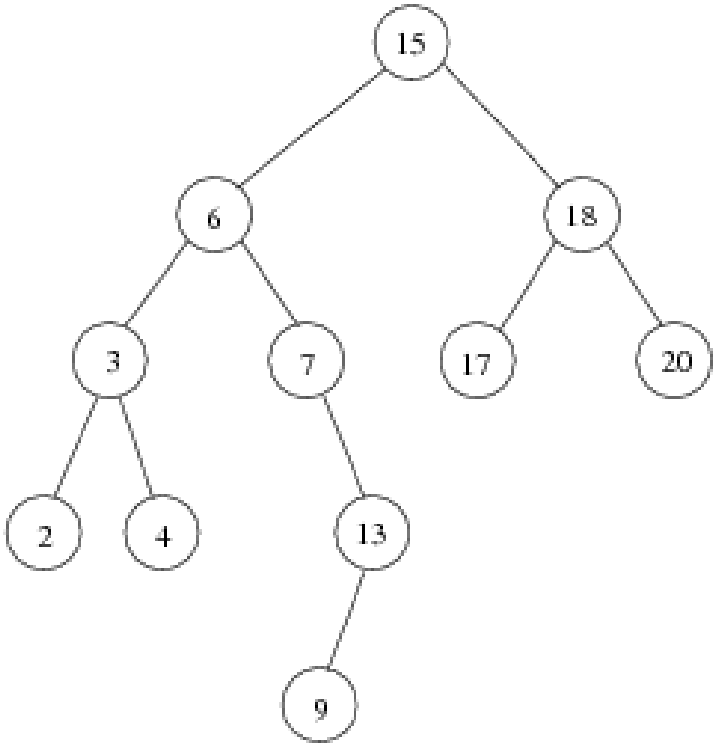
Print Values in BST

```
void TreeType::PrintTree()
{
 Print(root);
}
void Print(TreeNode* tree)
{
 if(tree != NULL) {
   Print(tree->left);
   cout << tree->info;
   Print(tree->right);
 }
```

Function PrintTree() print all values in BST using inorder traversal. Print() function will be called recursively, starting from left subtree, root and right subtree.

Inorder traversal of BST

Print out all the keys in sorted order



Inorder: 2, 3, 4, 6, 7, 9, 13, 15, 17, 18, 20

The Efficiency of Binary Search Tree Operations

- The maximum number of comparisons required by any binary search tree operation is the number of nodes along the longest path from root to a leaf, which is the tree's height.
- The order in which insertion and deletion operations are performed on a binary search tree affects its height.
- Insertion in random order produces a binary search tree that has near-minimum height.
- Insertion in sequential order produces a binary search tree that is unbalanced and has height = O(n).

The Efficiency of Binary Search Tree Operations

Operation	Average case	Worst case
Retrieval	O(log *n*)	O(*n*)
Insertion	O(log *n*)	O(*n*)
Deletion	O(log *n*)	O(*n*)
Traversal	O(*n*)	O(*n*)

Summary and Conclusion

- Binary search trees come in many shapes. The shape of the tree determines the efficiency of its operations
- The height of a binary search tree with *n* nodes can range from a minimum of O$\log_2(n + 1)$ to a maximum of *n.*
- The efficiency of binary search tree operations:

Operation	Average case	Worst case
Retrieval	O(log *n*)	O(*n*)
Insertion	O(log *n*)	O(*n*)
Deletion	O(log *n*)	O(*n*)
Traversal	O(*n*)	O(*n*)

References

- Nor Bahiah et al. *"Struktur data & algoritma menggunakan C++". Penerbit UTM. 2005.*
- Frank M. Carano, Janet J Prichard. "*Data Abstraction and problem solving with C++" Walls and Mirrors*. 5th edition (2007). Addision Wesley.