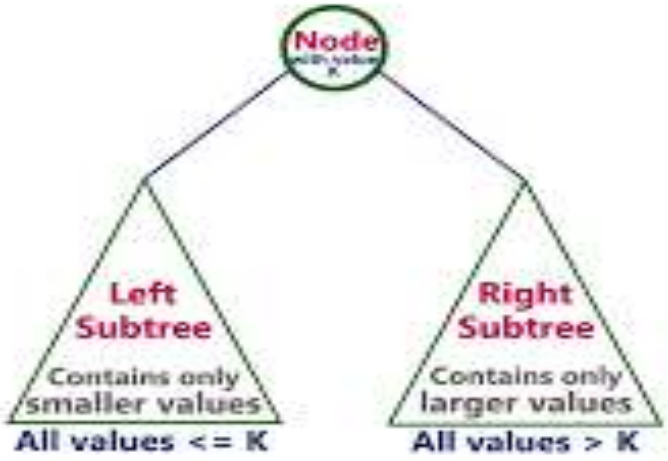
Lab 08: Implementation of Tree Traversal and Binary Search Tree

Binary Search Tree (BST)

A binary search tree is a binary tree with additional properties, in binary search tree each node has a comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left sub tree and smaller than the keys in all nodes in that node's right sub tree. A graphical representation of binary search tree is given below



Basic implementation of BST

Each node of binary tree contains the following information:

- A value (user's data)
- A link to the left child
- A link to the right child

In some implementations, node may store a link to the parent, but it depends on algorithm, programmer want to apply to BST. For basic operations, like addition, removal and search a link to the parent is not necessary. It is needed in order to implement iterators.

Operations on a BST

Following are the basic operations which can be applied to a binary search tree

- Add/ Insert a new value
- Search for a value
- Remove/Delete a value

Search Operation in a BST

Searching in a BST always starts at the root. We compare a data store at the root with the key we are searching for. If the node does not contain the key we proceed either to the left or right child depending upon comparison. If the result of comparison is negative we go to the left child, otherwise to the right child. Here are two implementations of the dynamic set operation search

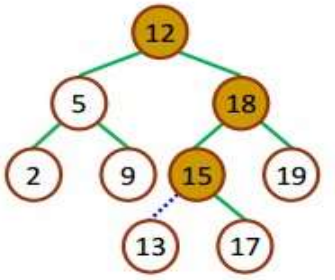
```
public BNode Search(int e)
{
    return Search(root, e);
}

private BNode Search(BNode node, int e)
{
    if (node == null)
        return null;

    if (node.data == e)
        return node;

    if (e < node.data)
        return Search(node.left, e);
    else
        return Search(node.right, e);
}
```

Insert/Add operation in a BST

To insert any element in a BST requires the concept of search operation. In a search operation when the element is not present then its output is null. Insert operation will replace null with a new node. Like search operation, insertion can also be recursive and iterative.

Algorithm for Insertion

```
TREE-INSERT(T, z)
1.   y ← NIL; x ← root[T]
2.   while x ≠ NIL
3.       do y ← x
4.          if key[z] < key[x]
5.            then x ← left[x]
6.            else x ← right[x]
7.   p[z] ← y
8.   if y = NIL
9.      then root[T] ← z      /* Tree T was empty */
10.     else if key[z] < key[y]
11.          then left[y] ← z
12.          else right[y] ← z
```



Inserting an item with key 13

```
public void Add(int e)
{
    root = Add(root, e);
}
private BNode Add(BNode node, int e)
{
    if (node == null)
    {
        node = new BNode(e);
        return node;
    }

    if (e < node.data)
        node.left = Add(node.left, e);
    else
        node.right = Add(node.right, e);

    return node;
}
```

Minimum and Maximum operations in a BST

The binary search tree property guarantees that

- The minimum key of a BST is located at the leftmost node, and
- The maximum key of a BST is located at the rightmost node

Traverse the appropriate pointers (left or right) until NIL is reached.

Algorithm for Minimum and Maximum

```
TREE-MINIMUM(x)
1.  while left[x] ≠ NIL
2.      do x ← left[x]
3.  return x
```

```
TREE-MAXIMUM(x)
1.  while right[x] ≠ NIL
2.      do x ← right[x]
3.  return x
```

```
public BNode Minimum(BNode node)
{
    if (node == null)
        return null;

    if (node.left == null)
        return node;
    else
        return Minimum(node.left);
}
```

```
public BNode Maximum(BNode node)
{
    if (node == null)
        return null;

    if (node.right == null)
        return node;
    else
        return Maximum(node.right);
}
```

Height of a BST

The height of the tree is determined by the number of nodes in the longest branch of the tree, either in left side or in right side. For a tree with just one node, the root node, the height is defined to be 0, if there are 2 levels of nodes the height is 1 and so on. A null tree (no nodes except the null node) is defined to have a height of -1.

```
public int Height()
{
    return Height(root);
}

public int Height(BNode node)
{
    if (node == null)
        return -1;

    int l = Height(node.left);
    int r = Height(node.right);

    if (l > r)
        return l + 1;
    else
        return r + 1;
}
```

Size of a BST

The **size** of the binary tree is the total number of nodes it contains. An empty binary tree has size 0.

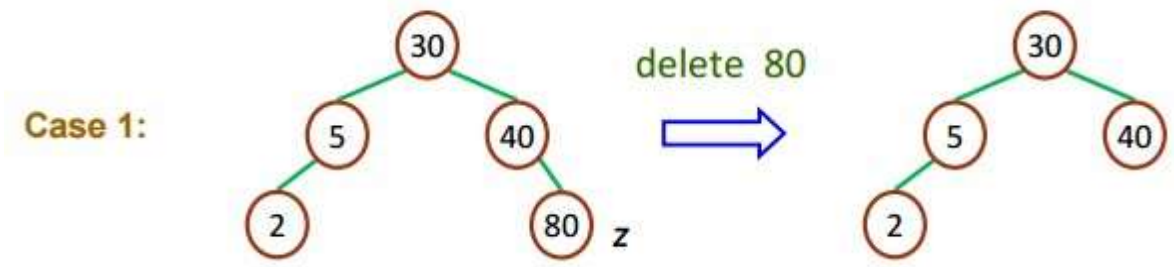
```
public int Size()
{
    return Size(root);
}

public int Size(BNode node)
{
    if (node == null)
        return 0;

    return 1 + Size(node.left) + Size(node.right);
}
```

Deleting a node from a Binary Search Tree

The deletion operation in BST is bit complex, it is divided in to three cases

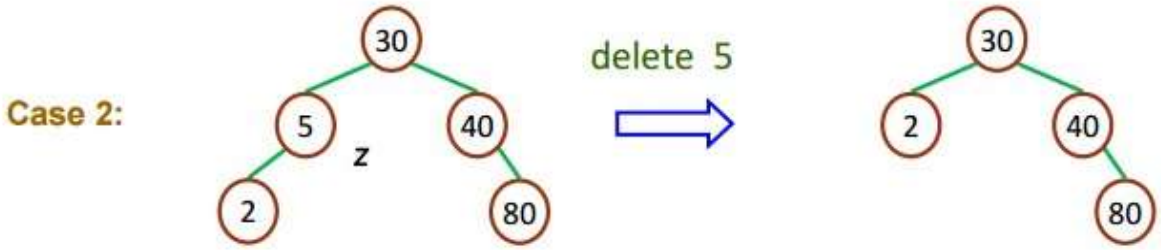
Case 1: if z has no children

Delete z by making the parent of z point to NIL, instead of to z



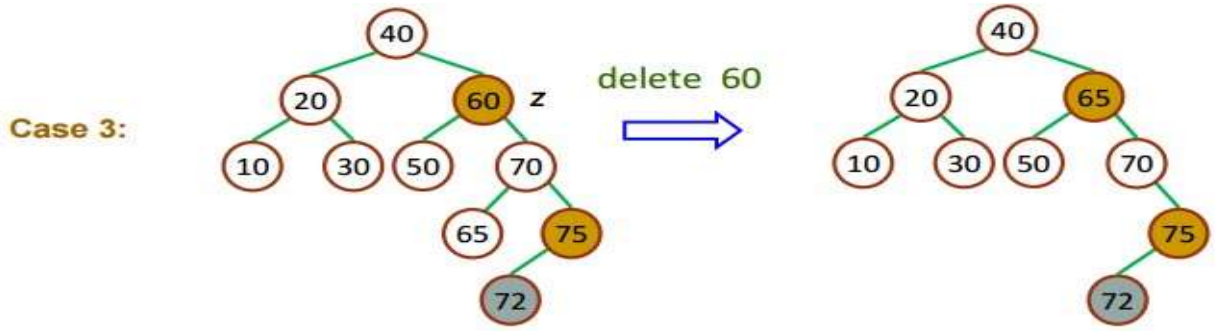
Case 2: If z has one child

Delete z by making the parent of z point to z's child, instead of to z



Case 3: if z has two children

- z's successor y has either no children or one child.
- (y is the minimum node with no left child in z's right sub tree)
- Delete y from the tree (using case1 or 2)
- Replace z's key with the deleted item y.

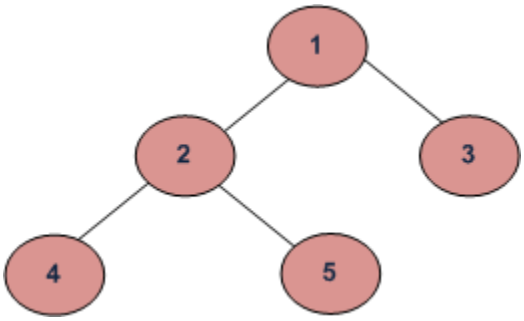


```
public BNode Delete(BNode node, int e)
{
    if (node == null)
        return null;

    if (node.data == e)
    {
        // case 1
        if (node.left == null && node.right == null)
            return null;
        // case 2 - left child is not empty - connect parent with left child
        if (node.left != null && node.right == null)
            return node.left;
        // case 2 - right child is not empty - connect parent with right child
        if (node.right != null && node.left == null)
            return node.right;
        // nothing like above then
        // both children are not empty
        if (node.right != null)
        {
            BNode succ = Successor(node);
            node.data = succ.data;
            node.right = Delete(node.right, succ.data);
        }
        else
        {
            BNode pred = Predecessor(node);
            node.data = pred.data;
            node.left = Delete(node.left, pred.data);
        }
```

```
        return node;
    }

    if (e < node.data)
        node.left = Delete(node.left, e);
    else
        node.right = Delete(node.right, e);

    return node;
}
```

Binary Search Tree Traversal:

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc.) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.



Example Tree

Breadth First or Level Order Traversal: 1 2 3 4 5

Depth First Traversals:
(a) Inorder (Left, Root, Right) : 4 2 5 1 3
(b) Preorder (Root, Left, Right) : 1 2 4 5 3
(c) Postorder (Left, Right, Root) : 4 5 2 3 1

Inorder Traversal:

An inorder traversal visits all the nodes in a BST in ascending order of the node key values.

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

```
public void InOrder()
{
    InOrder(root);
}
private void InOrder(BNode node)
{
    if (node == null)
        return;
    InOrder(node.left);
    Console.WriteLine(node.data);
    InOrder(node.right);
}
```

Uses of Inorder
In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversals reversed can be used.
Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal:

A preorder traversal visits the root node first, followed by the nodes in the subtrees under the left child of the root, followed by the nodes in the subtrees under the right child of the root.

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

```
public void PreOrder()
{
    PreOder(root);
}
private void PreOder(BNode node)
{
    if (node == null)
        return;

    Console.WriteLine(node.data);
    PreOder(node.left);
    PreOder(node.right);
}
```

Uses of Preorder
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.
Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal:

A postorder traversal, the method first recurses over the left subtrees and then over the right subtrees.

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

```
public void PostOrder()
{
    PostOder(root);
}
private void PostOder(BNode node)
{
    if (node == null)
        return;

    PostOder(node.left);
    PostOder(node.right);
    Console.WriteLine(node.data);
}
```

Uses of Postorder

Postorder traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression of an expression tree.

Example: Postorder traversal for the above given figure is 4 5 2 3 1.