Data Structures Lab
Session 9

Course: Data Structures (CS2001) **Semester:** Fall 2022
Instructor: Mafaza Mohi **T.A:** N/A

Note:
- Lab manual cover following topics
  {Tree, BST, Design and implement classes for binary tree nodes and nodes for general tree, Traverse the tree with the three common orders, Operation such as searches, insertions, and removals on a binary search tree and its applications}
- Maintain discipline during the lab.
- Just raise your hand if you have any problem.
- Completing all tasks of each lab is compulsory.
- Get your lab checked at the end of the session

Binary Tree

A typical graphical representation of a binary tree is essentially that of an upside down tree. It begins with a root node, which contains the original key value. The root node has two child nodes; each child node might have its own child nodes. Ideally, the tree would be structured so that it is a perfectly balanced tree, with each node having the same number of child nodes to its left and to its right. A perfectly balanced tree allows for the fastest average insertion of data or retrieval of data. The worst case scenario is a tree in which each node only has one child node, so it becomes as if it were a linked list in terms of speed. The typical representation of a binary tree looks like the following:

```
      10
     /  \
    6    14
   / \   / \
  5   8 11  18
```

```
Class node
{
  int key_value;
  node *left;
  node *right;
};
class btree
{
    public:
        btree(){
  root=NULL;}

        ~btree();

        void insert(int key);
        node *search(int key);
        void destroy_tree();

    private:
        void destroy_tree(node *leaf);
        void insert(int key, node *leaf);
        node *search(int key, node *leaf);

        node *root;
}; void btree::destroy_tree(node *leaf)
{
  if(leaf!=NULL)
  {
    destroy_tree(leaf->left);
    destroy_tree(leaf->right);
    delete leaf;
  }
}

void btree::insert(int key, node *leaf)
{
  if(key< leaf->key_value)
  {
    if(leaf->left!=NULL)
     insert(key, leaf->left);
    else
    {
      leaf->left=new node;
      leaf->left->key_value=key;
      leaf->left->left=NULL;    //Sets the left child of the child node to null
      leaf->left->right=NULL;   //Sets the right child of the child node to
null
    }
  }
  else if(key>=leaf->key_value)
  {
    if(leaf->right!=NULL)
      insert(key, leaf->right);
    else
    {
      leaf->right=new node;
      leaf->right->key_value=key;
      leaf->right->left=NULL;  //Sets the left child of the child node to null
      leaf->right->right=NULL; //Sets the right child of the child node to null
    }
  }
}
```

<u>Binary Search Tree</u>

KeyPoint: A Binary Search Tree (BST) is a binary tree with the following properties:

- The left subtree of a particular node will always contain nodes whose keys are less than that node’s key.

- The right subtree of a particular node will always contain nodes with keys greater than that node's key. The left and right subtree of a particular node will also, in turn, be binary search trees

Task-1:

Build a functionality named DispatcherTest which will check a queue/list of processes to process using its two way dispatcher. The dispatcher will only be able to process those list of process that form an actual BST(Hint: left child with smaller children and right with larger children) . If any list passes the autodispatcher test, It can then be sent to Dispatcher with a message.
Message: Allowed/Rejected

BST Insertion

Sample Code of class Nodes

```
Create class Nodes
class Node {    private:
 int key;
 string name;

 Node leftChild;
  Node rightChild; public:
 Node(int key, string name) {

      this.key = key;
      this.name = name;

 }
string toString() {

      return cout<<name<< " has the key " <<key<<endl;
        } };
```

Task-2: Complete the following Code:

A. Create class BinaryTree and create a function which add nodes in BST

```
class BinaryTree {
  private:   Node root;
 public:
 void addNode(int key, string name) {

      -----------------------// Create a new Node and initialize it


      // If there is no root this becomes root

      if (root == NULL) {

             ----------------------------
```

```
} else {

      // Set root as the Node we will start with as we traverse the tree

      -----------------------------
      // Future parent for new Node

      Node parent;

      while (true) {

            // root is the top set the parent node to the root node

            -------------------------

            // Check if the new node should go on
            // the left side of the parent node
            Key is compared with that of root. If the key is less than root,
            it is compared with root’s left child key. If greater, it is
            compared with the root's right child. Continue this process until
            the new node is compared with a leaf node and added either on the
            right or left child depending on its key.

            }
```

B. Implement main.cpp for the code provided such that a given array is passed to form a BST{ 15, 10, 20, 8, 12, 16, 25 }

Task 3:

You need to create a structurally similar tree such that given two integer arrays, X and Y, representing a set of BST keys, check if they represent the same BSTs or not. Assume that the keys are inserted into the BST in the same order as they appear in the array.

Sample Input
X[] = { 15, 25, 20, 22, 30, 18, 10, 8, 9, 12, 6 }
Y[] = { 15, 10, 12, 8, 25, 30, 6, 20, 18, 9, 22 }

<u>Tree Traversals: Inorder, PreOrder, PostOrder</u> .

Task-4:

A. Write recursive algorithms that perform preorder and inorder tree walks.

Preorder Traversal approach.

1. Visit Node.

2. Traverse Node’s left sub-tree.
3. Traverse Node’s right sub-tree

B. Write the iterative code for preorder traversal.

BST Deletion

BST Deletion

1) ***Node to be deleted is the leaf:*** Simply remove from the tree.

```
       50                            50
    /     \         delete(20)      /   \
   30      70       --------->    30     70
  /  \    /  \                     \    /  \
20   40  60   80                   40  60   80
```

2) ***Node to be deleted has only one child:*** Copy the child to the node and delete the child

```
       50                            50
    /     \         delete(30)      /   \
   30      70       --------->    40     70
     \    /  \                          /  \
     40  60   80                       60   80
```

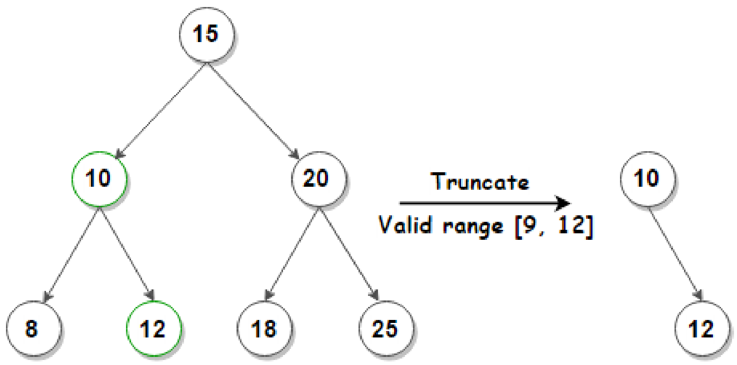
3) ***Node to be deleted has two children:*** Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor. Note that inorder predecessor can also be used.

```
       50                            60
    /     \         delete(50)      /   \
   40      70       --------->    40     70
          /  \                             \
         60   80                            80
```

The important thing to note is, inorder successor is needed only when the right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in the right child of the node.

Task 5:

Given a BST and a range of keys(values), remove nodes from BST that have keys outside the given range



Searching

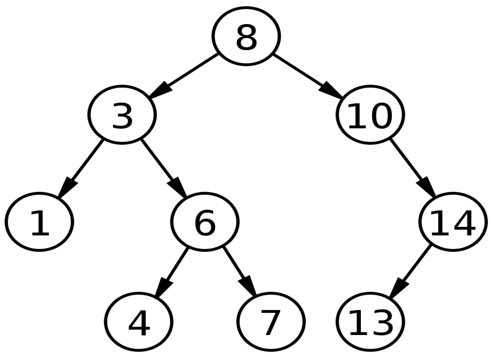
We use the following procedure to search for a node with a given key in a binary search tree. Given a pointer to the root of the tree and a key *k*, TREE-SEARCH returns a pointer to a node with key *k* if one exists; otherwise, it returns NIL.

```
TREE-SEARCH (x, k)
1 if x = NIL or k = key[x]
2     then return x
3  if k < key[x]
4     then return TREE-SEARCH (left[x], k)
5     else return TREE-SEARCH (right[x], k)
```

The procedure begins its search at the root and traces a path downward in the tree, as shown in Figure 13.2. For each node *x* it encounters, it compares the key *k* with *key*[*x*]. If the two keys are equal, the search terminates. If *k* is smaller than *key*[*x*]*,* the search continues in the left subtree of *x,* since the binary-search-tree property implies that *k* could not be stored in the right subtree. Symmetrically, if *k* is larger than *key*[*k*]*,* the search continues in the right subtree. The nodes encountered during the recursion form a path downward from the root of the tree, and thus the running time of TREE-SEARCH is *O*(*h*)*,* where *h* is the height of the tree.

Task:6

Using the following BST, find the inorder successor of nodes 10, 6, and 13.

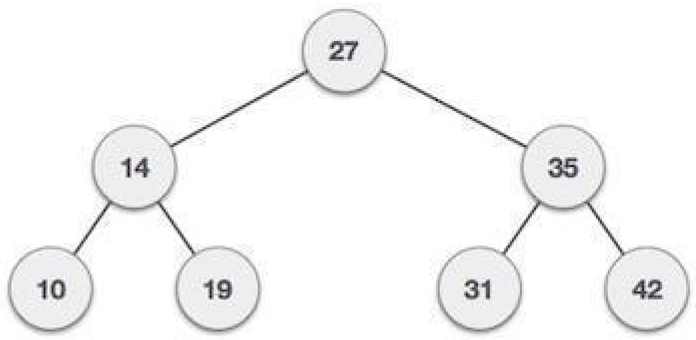


Summary Discussion

Binary search tree (BST)

Binary search tree (BST) or a lexicographic tree is a binary tree data structure that has the following binary search tree properties:

- Each node has a value.
- The key value of the left child of a node is less than to the parent's key value.
- The key value of the right child of a node is greater than (or equal) to the parent's key value.
- And these properties hold true for every node in the tree.



- **Subtree**: any node in a tree and its descendants.
- **Depth of a node**: the number of steps to hop from the current node to the root node of the tree.
- **Depth of a tree**: the maximum depth of any of its leaves.
- **Height of a node**: the length of the longest downward path to a leaf from that node.
- **Full binary tree**: every leaf has the same depth and every non-leaf has two children.
- **Complete binary tree**: every level except for the deepest level must contain as many nodes as possible; and at the deepest level, all the nodes are as far left as possible.
- **Traversal**: an organized way to visit every member in the structure.

Traversals

The binary search tree property allows us to obtain all the keys in a binary search tree in a sorted order by a simple traversing algorithm, called an in order tree walk, that traverses the left sub tree of the root in in order traverse, then accessing the root node itself, then traversing in in-order the right sub tree of the root node.

The tree may also be traversed in preorder or post order traversals. By first accessing the root, and then the left and the right sub-tree or the right and then the left sub-tree to be traversed in preorder. And the opposite for the post order.
The algorithms are described below, with Node initialized to the tree's root.

• **Preorder Traversal**

1. Visit Node.
2. Traverse Node's left sub-tree.
3. Traverse Node's right sub-tree.

• **In-order Traversal**

1. Traverse Node's left sub-tree.
2. Visit Node.
3. Traverse Node's right sub-tree

• **Post-order Traversal**

1. Traverse Node's left sub-tree.
2. Traverse Node's right sub-tree.
3. Visit Node

Reference:

http://staff.ustc.edu.cn/~csli/graduate/algorithms/book6/chap13.htm

Lab9: Binary Search Tree		
Std Name:		**Std ID:**
Lab1-Tasks	**Completed**	**Checked**
Task #1		
Task #2		
Task #3		
Task# 4		
Task# 5		
Task# 6		