Binary Trees

Data Structures Lab-9

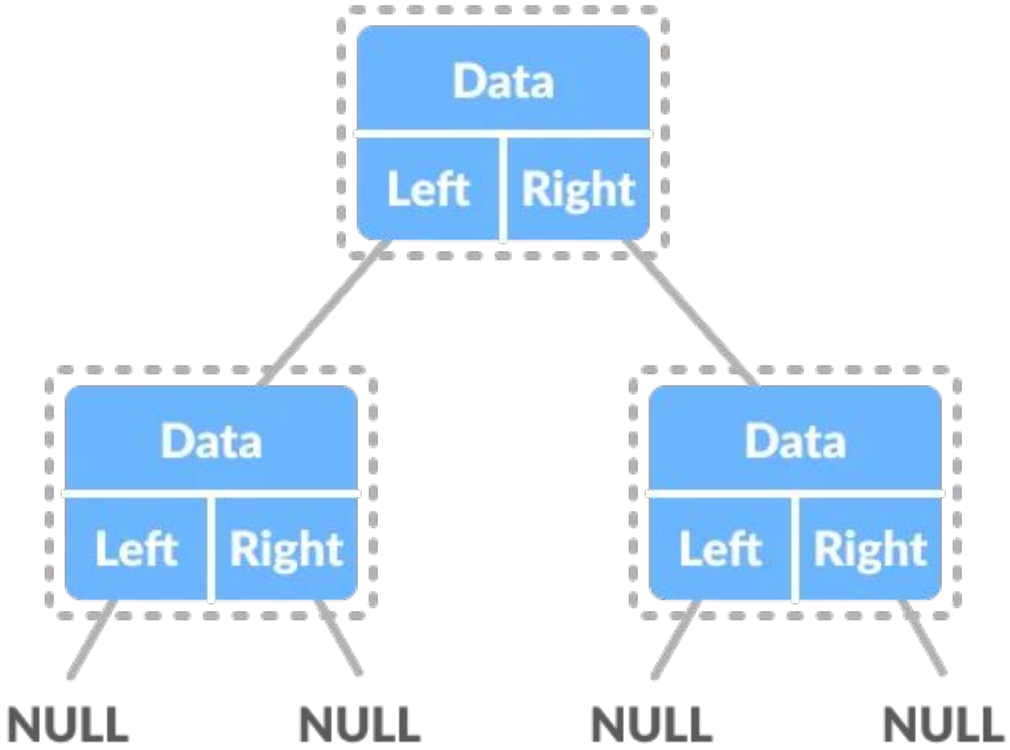
Today we have Quiz-2 & Binary Trees as a new Topic.

Today’s Content

- What is Binary Tree
- Important Concepts
- Tree Constraints
- Binary Search Trees & Operations on BST
- Binary Search Tree Traversing
- Lab Task

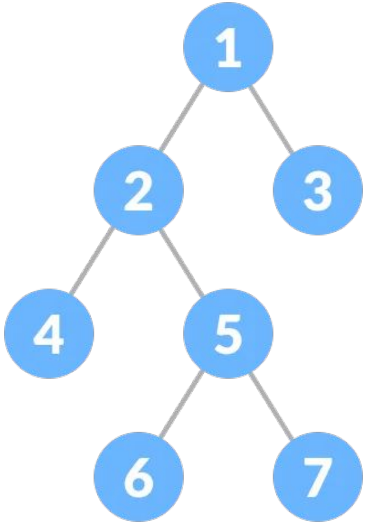
What is Binary Tree

```
template<class T>
class Node {
	T value;
	Node * left;
	Node * right;
};
```



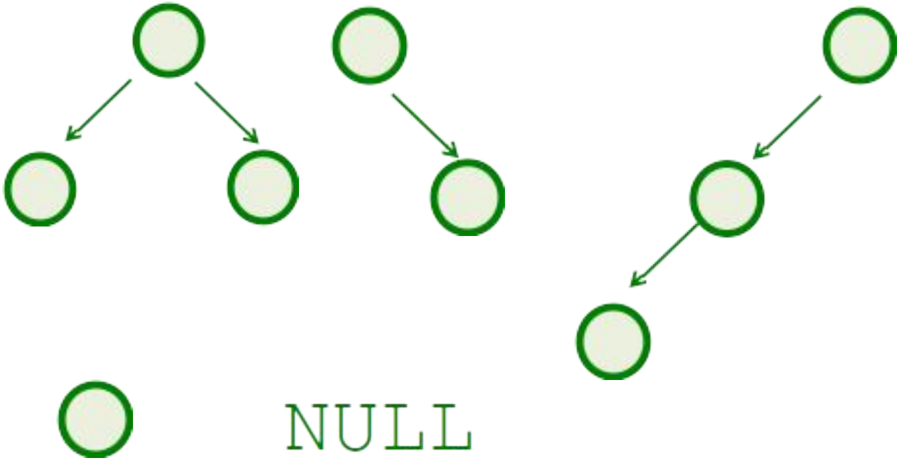
Important Concepts with BT

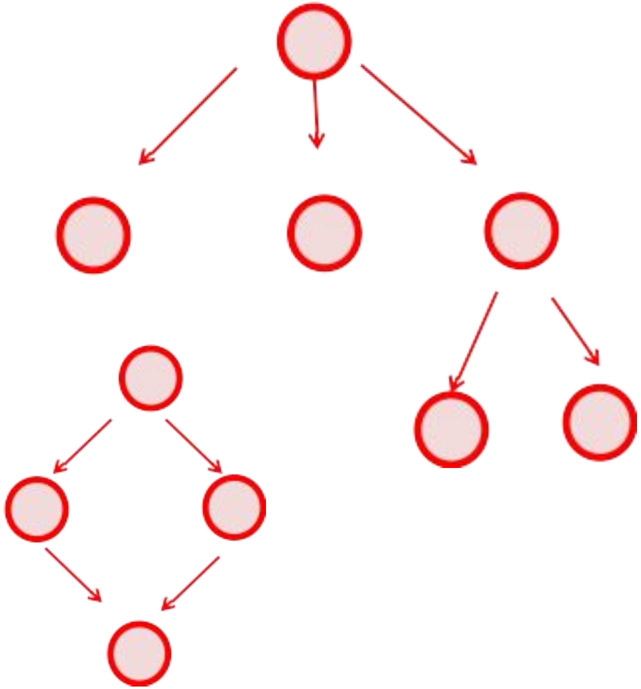
- A single element is a node or tree.
- First top node is the **parent** [i.e. 1].
- The nodes that we point to are **children** [i.e. 2, 3].
- Nodes that share the same parent are **siblings** [i.e. 4, 5].
- Depth/Level is the # of links from the root. [Depth of 5 = 2, Depth of 7 = 3].
- I.e. 6 and 7 are **leaves.**
- Nodes that have children are **inner nodes** [i.e. 2, 5].
- A node without any parents is the **root** [i.e. 1].
- Nodes are organized in levels.
- Height(h) = # levels (or #levels -1).
- Full tree, every node has two children and leaves are on the same level.
- Tree operations are being measured by h.



Important Constraints

- Nodes MUST have only one parent.
- No cycles in the tree.
- So we have unique path from root to every node.



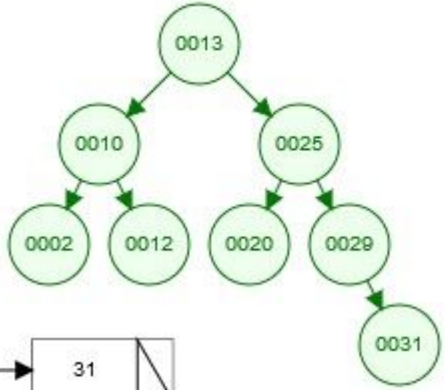


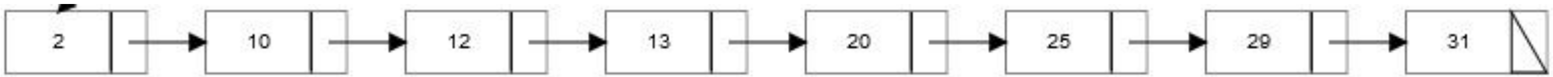
BST and its Operations

- In BST for any node, the left subtree is **less than** that node, the right subtree is **greater than** that node.
- Less than and Greater than depends on the type of the data.
- All nodes in the left subtree are less than the value in the root node.
- All nodes in the right subtree are greater than the value in the root node.
- The smallest element is the leftmost element.
- The largest element is the rightmost element.
- BST Operations: [Insert, Delete, Search] **Time Complexity = O(h) = O(Log n).**
- BST Operations are recursive calls per child.

Search for [31] in both the sorted linked list and the binary search tree.

Compare the performance!





Insertion in BST

```
void insert(int data) {
   struct node *tempNode = (struct node*) malloc(sizeof(struct node));
   struct node *current;
   struct node *parent;

   tempNode->data = data;
   tempNode->leftChild = NULL;
   tempNode->rightChild = NULL;

   //if tree is empty
   if(root == NULL) {
      root = tempNode;
   } else {
      current = root;
      parent = NULL;

      while(1) {
         parent = current;

         //go to left of the tree
         if(data < parent->data) {
            current = current->leftChild;
            //insert to the left

            if(current == NULL) {
               parent->leftChild = tempNode;
               return;
            }
         }//go to right of the tree
         else {
            current = current->rightChild;

            //insert to the right
            if(current == NULL) {
               parent->rightChild = tempNode;
               return;
            }
         }
      }
   }
}
```

Search in BST

```
struct node* search(int data){
   struct node *current = root;
   printf("Visiting elements: ");

   while(current->data != data){

      if(current != NULL) {
         printf("%d ",current->data);

         //go to left tree
         if(current->data > data){
            current = current->leftChild;
         }//else go to right tree
         else {
            current = current->rightChild;
         }

         //not found
         if(current == NULL){
            return NULL;
         }
      }
   }
   return current;
}
```

Deletion in BST

```
void deleteNode(struct node* root, int data){

    if (root == NULL) root=tempnode;

    if (data < root->key)
        root->left = deleteNode(root->left, key);


    else if (key > root->key)
        root->right = deleteNode(root->right, key);

    else
    {
        if (root->left == NULL)
        {
            struct node *temp = root->right;
            free(root);
            return temp;
        }
        else if (root->right == NULL)
        {
            struct node *temp = root->left;
            free(root);
            return temp;
        }

        struct node* temp = minValueNode(root->right);

        root->key = temp->key;

        root->right = deleteNode(root->right, temp->key);
    }
    return root;

}
```

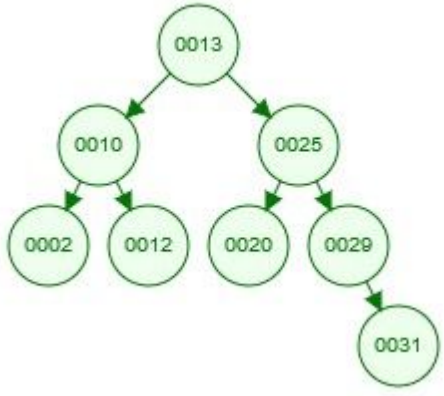
BST Traversing

Traversing can be done in two ways:

- Breadth-first (Visiting nodes level by level from left to right)
- Depth-first
    - In-order (Left, Head, Right)
    - Pre-order (Head, Left, Right)
    - Post-order (Left, Right, Head)

Breadth-First Traversing

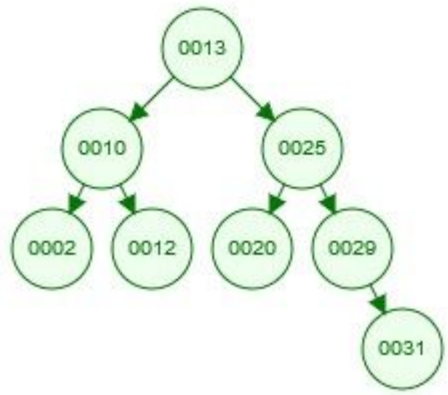
```
q.enqueue(root)
while(q.size() != 0)
{
  node = q.dequeue()
  print(node->value)
  q.enqueue(node->left)
  q.enqueue(node->right)
}
```



Output: 13, 10, 25, 2, 12, 20, 29, 31

Depth-First Traversing (*In-Order*)

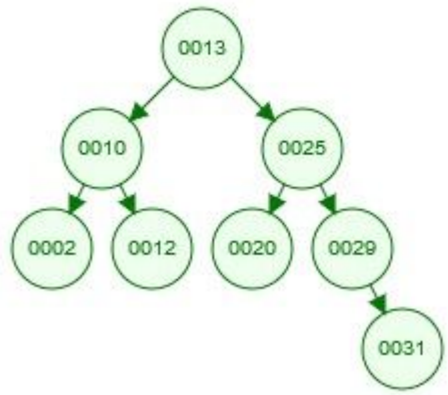
```
inorder(node)
{
  if (node == 0) return;
  inorder(node->left);
  print(node->value);
  inorder(node->right);
}
```



Output: 2, 10, 12, 13, 20, 25, 29, 31

Depth-First Traversing (*Pre-Order*)

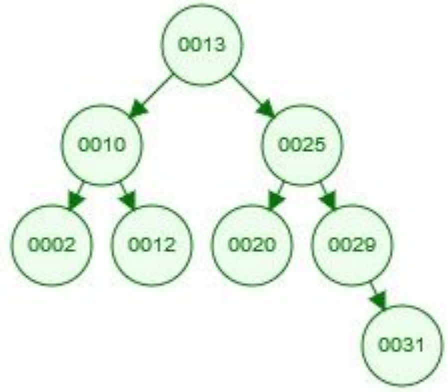
```
preorder(node)
{
  if (node == 0) return;
  print(node->value);
  preorder(node->left);
  preorder(node->right);
}
```



Output: 13, 10, 2, 12, 25, 20, 29, 31

Depth-First Traversing (*Post-Order*)

```
postorder(node)
{
  if (node == 0) return;
  postorder(node->left);
  postorder(node->right);
  print(node->value);
}
```



Output: 2, 12, 10, 20, 31, 29, 25, 13

Worst Cases in BST

Insert 2, 7, 8, 10, 21 into an empty tree.

The solution is to keep the BST as much balanced as possible.

- Self-Balancing Trees:
  - Red-Black Trees
  - AVL Trees
  - Splay Trees

Lab Task

Task is to be delivered on Moodle

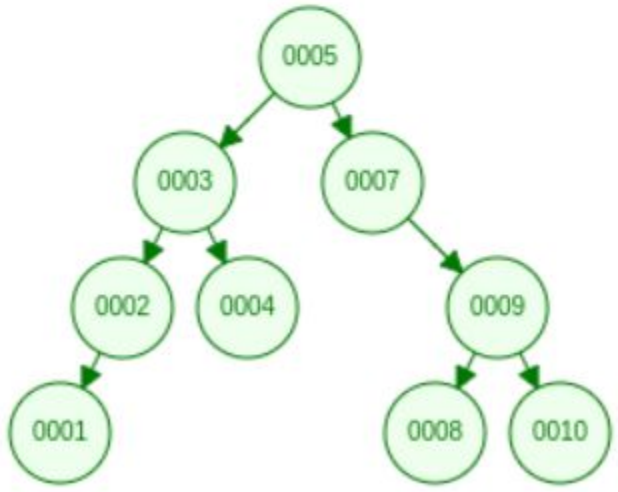
Using the **In-Order** technique, print the nodes within a specific range.

TEST CASE:

printBSTRange(3, 6) => [3, 4, 5]

printBSTRange(8, 15) => [9, 10]

printBSTRange(6, 6) = > []



Submit ***ONLY*** ID.cpp file, other ***IS NOT ACCEPTED***

Submission is 24 hours after lab time.

Thank you!