DHA Suffa University
Department of Computer Science
CS 2001L – Data Structures and Algorithms Lab
Spring 2021

Lab 10 – Binary Search Tree

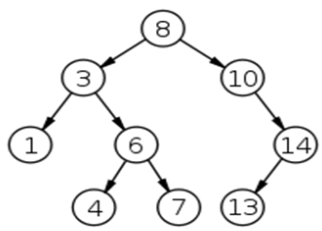
Objective:

To learn about

- Binary Search Tree

Binary Search Tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:



- The left subtree of a node contains only nodes with data less than the node's data.
- The right subtree of a node contains only nodes with data greater than the node's data.
- The left and right subtree each must also be a binary search tree

Main applications of trees include:

- Manipulate hierarchical data.
- Make information easy to search (see tree traversal).
- Manipulate sorted lists of data.
- As a workflow for compositing digital images for visual effects.
- Router algorithms

Algorithm for Insertion:

```
Algorithm: insert(value)
    Pre: value has passed custom type checks for type T
    Post: value has been placed in the correct location in the tree
            if root = null
               root ← temp;
               print Root created
               return root;
          end if
          else
              temp ← insertNode(root, value);
              return temp;
        end else
end insert(value)
```

```
Algorithm: insertNode(root, value)
      Pre: root is the node to start from
            value has passed custom type checks for type T
      Post: value has been placed in the correct location in the tree

    if value = root.data
            print Value already exist

    if value < root.data
     //if value is less than root than insertion at left
        if root->left = null
            root.left ← temp
            print left node created
        end if

       else
        insertNode(root.left, value)
       end else

    end if

    else
        //if value is greater than root than insertion at right
        if root.right = null
            root.right ← temp
            print right node created
        end if

        else
            insertNode(root.right, value)
       end else
 return root
end insertNode(root, value)
```

Algorithm for Searching:

```
Algorithm: Search(root, value)
    Pre: root is the root node of the tree, value is what we would like to locate
    Post: value is either located or not

      if root = null
         print Value not found
     end if
    else
        if root.data = value)
            print Value Found
        end if
        else if value < root.data)
            search(root.left, value)
        end else if
        else
            search(root.right, value)
       end else
   end else
end Search(root, value)
```

Algorithm for Deletion:

Algorithm findNode(*root*, *value*)
Pre: *value* is the value of the node we want to find the parent of
root is the root node of the BST
Post: a reference to the node of *value* if found; otherwise

```
  if root = null
      return 0
  if root.data = value
      return root
   else if value < root.data
      return findNode (root.left, value)
   else
      return findNode (root.right, value)
```

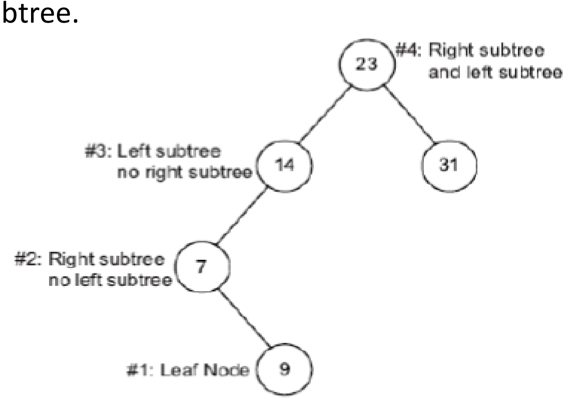
end findNode(*root*, *value*)

Algorithm FindParent(value, root)
Pre: value is the value of the node we want to find the parent of
root is the root node of the BST and is $\neq$ null
Post: a reference to the parent node of value if found; otherwise null

```
  if value = root.data
          return 0
      if value < root.data
          if root.left = null
              return 0
          else if root.left.data = value
              return root
          else
              return findParent(root.left, value)
      end if
      else
          if root.right = null
              return 0
         else if root.right.data = value
             return root
         else
             return findParent(root.right, value)
```

Removing a node from a BST is fairly straightforward, with four cases to consider:

1. The value to remove is a leaf node; or
2. The value to remove has a right subtree, but no left subtree; or
3. The value to remove has a left subtree, but no right subtree; or
4. The value to remove has both a left and right subtree in which case we promote the largest value in the left subtree.



Algorithm: Delete(root, value)

Pre: value is the value of the node to remove, root is the root node of the BST
Count is the number of items in the BST
Post: node with value is removed if found in which case yields true, otherwise false

```
nodeToRemove ←FindNode(root, value)
if nodeToRemove = null
      return false // value not in BST
end if

parent ← FindParent(root, value)

 if Count = 1 //Additional statement for check
       root ← null // we are removing the only node in the BST

  // case #1
 else if nodeToRemove.left = null AND nodeToRemove.right = null
        if nodeToRemove.data  <  parent.data
                  parent.left ← null
        end if
        else

                     parent.right ← null
         end else

    // case # 2
  else if nodeToRemove.left = null AND nodeToRemove.right ≠ null
           if nodeToRemove.data < parent.data
                    parent.left ← nodeToRemove.right
           end if
           else
                    parent.right ← nodeToRemove.right
           end else

   // case #3
 else if nodeToRemove.left ≠ null AND nodeToRemove.right = null
           if nodeToRemove.data < parent.data
                    parent.left ← nodeToRemove.left
            end if
            else
                    parent.right  ← nodeToRemove.left
            end else
            else
      // case #4
 largestValue ←  nodeToRemove.left
 while largestValue.right ≠ null
      //find the largest value in the left subtree of nodeToRemove
      largestValue ← largestValue.right
 end while
     // set the parents' Right pointer of largestValue to null
     FindParent(root, largestValue.data).right ← null
     nodeToRemove.data ← largestValue.data

        Count ← Count - 1
        return true
end delete(value)
```

Tree Traversals (Inorder, Preorder and Postorder)

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.

Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order.

Algorithm: Inorder(root)

Pre: root is the root node of the tree, value is what we would like to locate

Post: value is either located or not

```
inorderPrint( TreeNode *root )
      if ( root != NULL )
           inorderPrint( root->left )
           Print 'root->item'
           inorderPrint( root->right )
```

endAlgorithm Inorder(root)

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

Algorithm: Preorder(root)

Pre: root is the root node of the tree, value is what we would like to locate

Post: value is either located or not

```
preorderPrint( TreeNode *root )

        if ( root != NULL )

           Print 'root->item'
           preorderPrint( root->left )
           preorderPrint( root->right )
        end if
```

endAlgorithmPreorder(root)

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.

Algorithm: Postorder(root)

Pre: root is the root node of the tree, value is what we would like to locate

Post: value is either located or not

```
postorderPrint( TreeNode *root )

   if ( root != NULL )
postorderPrint( root->left )
postorderPrint( root->right )
Print 'root->item'
```

endAlgorithmPostorder(root)

LAB ASSIGNMENT

1. Create a BST with the following values 21, 16, 2, 25, 30, 14, 2, 60, 8, 15, 35, 40, 100, 55.
   a) Write a function to find the height of the tree.
   b) Write a function that will print the Right Sub Tree of the BST only in preorder, postorder and inorder.

2. Create a similarly structured tree as created in Q2. Write a function that will take both trees as argument and create a third final tree having nodes values equal to the sum value of nodes of the previous two trees.

SUBMISSION GUIDELINES

- Take a screenshot of each task (code and its output), labeled properly.
- Place all the screenshots and the code files (properly labeled) in a single folder labeled with Roll No and Lab No. e.g. **'cs191xxx_Lab01'**.
- Submit the folder at [LMS]
- **-100%** policies for plagiarism.