# Laboratory 8

Title of the Laboratory Exercise: Binary trees

1. Introduction and Purpose of Experiment

Linear organization used on arrays, vectors, stacks and queues become inefficient in some applications. Then we choose the structures which provide non-linear organization. Binary tree is a non-linear data structure used in many applications. This experiment introduces binary search trees and its applications.

1. Aim and Objectives

Aim

* To develop Binary search tree ADT

Objectives

At the end of this lab, the student will be able to

* Design binary tree ADT
* Use binary tree ADT to illustrate the binary tree operations
* Use binary tree ADT and illustrate binary tree traversals

1. Experimental Procedure
   * 1. Analyse the problem statement
     2. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
     3. Implement the algorithm in C language
     4. Compile the C program
     5. Test the implemented program
     6. Document the Results
     7. Analyse and discuss the outcomes of your experiment
2. Calculations/Computations/Algorithms
   * + 1. Insertion Algorithm

Step 1 : Start

Step 2 : check, whether value in current node and a new value are equal. If so, duplicate is found. Otherwise,

Step 3 : if a new value is less, than the node's value:

Step 3.1 : if a current node has no left child, place for insertion has been found;

Step 3.2 : otherwise, handle the left child with the same algorithm.

Step 4 : if a new value is greater, than the node's value:

Step 4.1 : if a current node has no right child, place for insertion has been found;

Step 4.2 : otherwise, handle the right child with the same algorithm.

Step 5: Stop

* + - 1. Deletion Algorithm

Step 1 : Start

Step 2 : Node to be deleted is leaf: Simply remove from the tree.

Step 3 : Node to be deleted has only one child: Copy the child to the node and delete the child.

Step 4 : 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. The important thing to note is, inorder successor is needed only when right child is not empty. In this particular case, inorder successor can be obtained by finding the minimum value in right child of the node.

Step 5: Stop

Implementation:

#include<stdio.h>

#include<stdlib.h>

struct node

{

int key;

struct node \*left, \*right;

};

// A utility function to create a new BST node

struct node \*newNode(int item)

{

struct node \*temp = (struct node \*)malloc(sizeof(struct node));

temp->key = item;

temp->left = temp->right = NULL;

return temp;

}

// A utility function to do inorder traversal of BST

void inorder(struct node \*root)

{

if (root != NULL)

{

inorder(root->left);

printf("%d ", root->key);

inorder(root->right);

}

}

/\* A utility function to insert a new node with given key in BST \*/

struct node\* insert(struct node\* node, int key)

{

/\* If the tree is empty, return a new node \*/

if (node == NULL) return newNode(key);

/\* Otherwise, recur down the tree \*/

if (key < node->key)

node->left = insert(node->left, key);

else

node->right = insert(node->right, key);

/\* return the (unchanged) node pointer \*/

return node;

}

/\* Given a non-empty binary search tree, return the node with minimum

key value found in that tree. Note that the entire tree does not

need to be searched. \*/

struct node \* minValueNode(struct node\* node)

{

struct node\* current = node;

/\* loop down to find the leftmost leaf \*/

while (current->left != NULL)

current = current->left;

return current;

}

/\* Given a binary search tree and a key, this function deletes the key

and returns the new root \*/

struct node\* deleteNode(struct node\* root, int key)

{

// base case

if (root == NULL) return root;

// If the key to be deleted is smaller than the root's key,

// then it lies in left subtree

if (key < root->key)

root->left = deleteNode(root->left, key);

// If the key to be deleted is greater than the root's key,

// then it lies in right subtree

else if (key > root->key)

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

// if key is same as root's key, then This is the node

// to be deleted

else

{

// node with only one child or no child

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;

}

// node with two children: Get the inorder successor (smallest

// in the right subtree)

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

// Copy the inorder successor's content to this node

root->key = temp->key;

// Delete the inorder successor

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

}

return root;

}

// Driver Program to test above functions

int main()

{

/\* Let us create following BST

50

/ \

30 70

/ \ / \

20 40 60 80 \*/

struct node \*root = NULL;

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

printf("Inorder traversal of the given tree \n");

inorder(root);

printf("\nDelete 20\n");

root = deleteNode(root, 20);

printf("Inorder traversal of the modified tree \n");

inorder(root);

printf("\nDelete 30\n");

root = deleteNode(root, 30);

printf("Inorder traversal of the modified tree \n");

inorder(root);

printf("\nDelete 50\n");

root = deleteNode(root, 50);

printf("Inorder traversal of the modified tree \n");

inorder(root);

return 0;

}

1. Presentation of Results

Inorder traversal of the given tree

20 30 40 50 60 70 80

Delete 20

Inorder traversal of the modified tree

30 40 50 60 70 80

Delete 30

Inorder traversal of the modified tree

40 50 60 70 80

Delete 50

Inorder traversal of the modified tree

40 60 70 80

1. Analysis and Discussions

Time Complexity: The worst case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).

Time Complexity: The worst case time complexity of delete operation is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of delete operation may become O(n)

1. Conclusions

A naïve algorithm for inserting a node into a BST is that, we start from the root node, if the node to insert is less than the root, we go to left child, and otherwise we go to the right child of the root. We continue this process (each node is a root for some sub tree) until we find a null pointer (or leaf node) where we cannot go any further. We then insert the node as a left or right child of the leaf node based on node is less or greater than the leaf node. We note that a new node is always inserted as a leaf node. A recursive algorithm for inserting a node into a BST is as follows. Assume we insert a node N to tree T. if the tree is empty, the we return new node N as the tree. Otherwise, the problem of inserting is reduced to inserting the node N to left of right sub trees of T, depending on N is less or greater than T. A BST is a connected structure. That is, all nodes in a tree are connected to some other node. For example, each node has a parent, unless node is the root. Therefore deleting a node could affect all sub trees of that node.