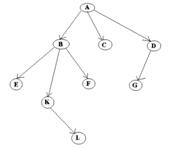
|  |
| --- |
| **NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES**  **CS 201–DATA STRUCTURES LAB**  **Lab Session 13** |
| **Instructors:** Mr. Irfan Ayub, Ms. Safia |

**Trees :**

A tree is a collection of nodes connected by directed (or undirected) edges. A tree is a nonlinear data structure, compared to arrays, linked lists, stacks and queues which are linear data structures. A tree can be empty with no nodes or a tree is a structure consisting of one node called the **root** and zero or one or more sub trees. A tree has following general properties:

* One node is distinguished as a root;
* Every node (exclude a root) is connected by a directed edge from exactly one other node; A direction is: parent -> children



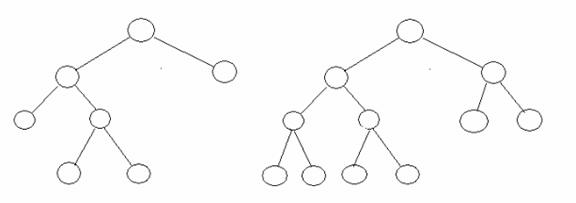
A is a parent of B, C, D,   
B is called a child of A.   
on the other hand, B is a parent of E, F, K

* In the above picture, the root has 3 sub-trees.
* Each node can have arbitrary number of children.
* Nodes with no children are called leaves, or external nodes.
* In the above picture, C, E, F, L, G are leaves.
* Nodes, which are not leaves, are called internal nodes. Internal nodes have at least one child.
* Nodes with the same parent are called siblings.
* In the picture, B, C, D is called siblings.
* The depth of a node is the number of edges from the root to the node.
* The depth of K is 2.
* The height of a node is the number of edges from the node to the deepest leaf.
* The height of B is 2.
* The height of a tree is a height of a root.

**Binary Trees**

A binary tree in which each node has exactly zero or two children is called a full binary tree. In a full tree, there are no nodes with exactly one child.

A complete binary tree is a tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right. A complete binary tree of the height h has between 2h and 2(h+1)-1 nodes. Here are some examples:



**Binary Search Trees**

Binary search tree (BST) or a lexicographic tree is a binary tree data structure which 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 holds true for every node in the tree.

If a BST allows duplicate values, then it represents a multi-set. This kind of tree uses non-strict inequalities (<=, >=). Everything in the left sub-tree of a node is strictly less than the value of the node, but everything in the right sub-tree is either greater than or equal to the value of the node.

If a BST doesn't allow duplicate values, then the tree represents a set with unique values, like the mathematical set. Trees without duplicate values use strict inequalities, meaning that the left sub-tree of a node only contains nodes with values that are less than the value of the node, and the right sub-tree only contains values that are greater.

The choice of storing equal values in the right sub-tree only is arbitrary; the left would work just as well. One can also permit non-strict equality in both sides. This allows a tree containing many duplicate values to be balanced better, but it makes searching more complex.

**Traversals**

Stepping through the items of a tree, by means of the connections between parents and children, is called walking the tree and the action is a walk of the tree (traverse). Often, an operation might be performed when a pointer arrives at a particular node (visiting the node – for example, printing the value/s that the node contains).

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.

**Insertion**

Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right sub-trees as before. Eventually, we will reach a leaf and add the value as its right or left child, depending on the node's value.

**Step 1:**

**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;

}

};

Step 2:

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

Node newNode = new Node(key, name);

// If there is no root this becomes root

if (root == NULL) {

root = newNode;

} else {

// Set root as the Node we will start

// with as we traverse the tree

Node focusNode = root;

// Future parent for our new Node

Node parent;

while (true) {

// root is the top parent so we start

// there

parent = focusNode;

// Check if the new node should go on

// the left side of the parent node

if (key < focusNode.key) {

// Switch focus to the left child

focusNode = focusNode.leftChild;

// If the left child has no children

if (focusNode == NULL) {

// then place the new node on the left of it

parent.leftChild = newNode;

return; // All Done

}

} else { // If we get here put the node on the right

focusNode = focusNode.rightChild;

// If the right child has no children

if (focusNode == NULL) {

// then place the new node on the right of it

parent.rightChild = newNode;

return; // All Done

}

}

}

}

}

void in\_orderTraverseTree(Node focusNode)//Recursive {

}

void preorderTraverseTree(Node focusNode) //Recursive{

if (focusNode != NULL) {

cout<<focusNode<<” ”;

preorderTraverseTree(focusNode.leftChild);

preorderTraverseTree(focusNode.rightChild);

}

void post\_orderTraverseTree(Node focusNode) //Recursive{

}

void in\_orderTraverseTreeNR(Node focusNode)//Non Recursive {

}

void preorderTraverseTreeNR(Node focusNode)// Non Recursive

{

}

void post\_orderTraverseTreeNR(Node focusNode) // Non Recursive

{

}

Create class having main method

int main() {

BinaryTree theTree = new BinaryTree();

theTree.addNode(50, "Boss");

theTree.addNode(25, "Vice President");

theTree.addNode(15, "Office Manager");

theTree.addNode(30, "Secretary");

theTree.addNode(75, "Sales Manager");

theTree.addNode(85, "Salesman 1");

// Different ways to traverse binary trees

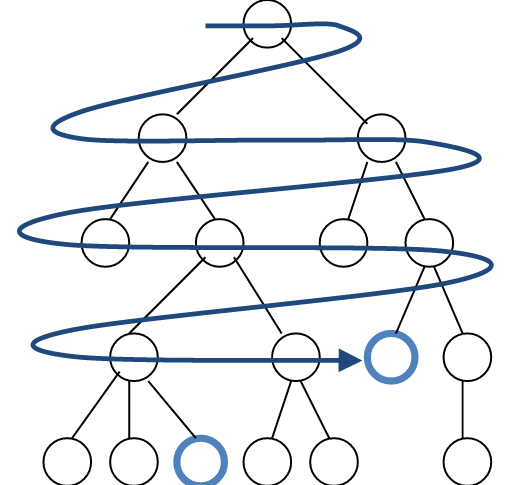
theTree.in\_orderTraverseTree(theTree.root);

theTree.preorderTraverseTree(theTree.root);

theTree.post\_orderTraverseTree(theTree.root);

}

**Breadth-first searching:**

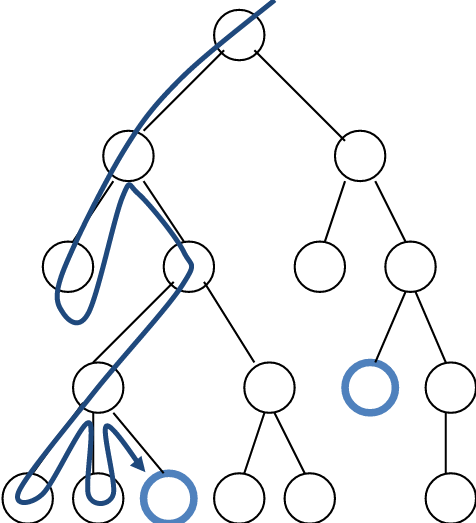


* A breadth-first search (BFS) explores nodes nearest the root before exploring nodes further away
* For example, after searching A, then B, then C, the search proceeds with D, E, F, G
* Node are explored in the order A B C D E F G H I J K L M N O P Q
* J will be found before N

**How to do breadth-first searching:**

Put the root node on a queue;  
while (queue is not empty) {  
 remove a node from the queue;  
 if (node is a goal node) return success;  
 put all children of node onto the queue;  
}  
return failure;

**Depth-first searching**



* A depth-first search (DFS) explores a path all the way to a leaf before backtracking and exploring another path
* For example, after searching A, then B, then D, the search backtracks and tries another path from B
* Node are explored in the order A B D E H L M N I O P C F G J K Q
* N will be found before J

**How to do depth-first searching:**

Put the root node on a stack;  
while (stack is not empty) {  
 remove a node from the stack;  
 if (node is a goal node) return success;  
 put all children of node onto the stack;  
}  
return failure;

**Lab Task:**

* Implement all pre-order, post-order, in-order traversal functions (recursive and non recursive both).
* Implement Breadth first and Depth first searching.

**AVL TREE:**

**Introduction**  
  
Binary search trees are an excellent data structure to implement array and sets. The main difficulty is that they are efficient only when they are balanced. Straightforward sequences of insertions can lead to highly unbalanced trees with poor asymptotic complexity and unacceptable practical efficiency. For  
  
example, if we insert n elements with keys that are in strictly increasing or decreasing order, the complexity will be O(n2). On the other hand, if we can keep the height to O(log(n)), as it is for a perfectly balanced tree, then the complexity is bounded by O(n \_ log(n)). The solution is to dynamically rebalance the search tree during insert or search operations. We have to be careful not to destroy the ordering invariant of the tree while we rebalance. Because of the importance of binary search trees, researchers have developed many different algorithms for keeping trees in balance, such as AVL trees, red/black trees, splay trees, or randomized binary search trees. They differ in the invariants they maintain (in addition to the ordering invariant), and when and how the rebalancing is done.

**The Height Invariant**

Recall the ordering invariant for binary search trees. Ordering invariant. At any node with key k in a binary search tree, all keys of the elements in the left sub-tree are strictly less than k, while all keys of the elements in the right sub-tree are strictly greater than k.  
  
  
To describe AVL trees we need the concept of tree height, which we define as the maximal length of a path from the root to a leaf. So the empty tree has height 0, the tree with one node has height 1, a balanced tree with three nodes has height 2. If we add one more node to this last tree is will have height 3. Alternatively, we can define it recursively by saying that the empty tree has height 0, and the height of any node is one greater than the maximal height of its two children. AVL trees maintain a height invariant (also sometimes called a balance invariant).  
  
**Height Invariant**: At any node in the tree, the heights of the left and right sub-trees differ by at most 1.  
  
As an example, consider the following binary search tree of height 3.  
If we insert a new element with a key of 14, the insertion algorithm for binary search trees without rebalancing will put it to the right of 13.  
  
Now the tree has height 4, and one path is longer than the others. However, it is easy to check that at each node, the height of the left and right sub-trees still differ only by one. For example, at the node with key 16, the left sub-tree has height 2 and the right sub-tree has height 1, which still obeys our height invariant.  
  
Now consider another insertion, this time of an element with key 15. This is inserted to the right of the node with key 14.  
  
All is well at the node labeled 14: the left sub-tree has height 0 while the right sub-tree has height 1. However, at the node labeled 13, the left sub-tree has height 0, while the right sub-tree has height 2, violating our invariant. Moreover, at the node with key 16, the left sub-tree has height 3 while the right sub-tree has height 1, also a difference of 2 and therefore an invariant violation. We therefore have to take steps to rebalance tree. We can see without too much trouble, that we can restore the height invariant if we move the node labeled 14 up and push node 13 down and to the right, resulting in the following tree.  
  
The question is how to do this in general. In order to understand this we  
  
need a fundamental operation called a rotation, which comes in two forms,  
left rotation and right rotation.  
**The AVL Tree Rotations**   
  
1. Rotations: How they work  
  
**Left Rotation (LL)**  
  
Imagine we have this situation:  
Figure 1-1

|  |
| --- |
| a \  b  \  c |

To fix this, we must perform a left rotation, rooted at A. This is done in the following steps:  
b becomes the new root.  
a takes ownership of b's left child as its right child, or in this case, null.  
b takes ownership of a as its left child.  
The tree now looks like this:  
  
Figure 1-2

|  |
| --- |
| b / \ a c |

**Right Rotation (RR)**  
A right rotation is a mirror of the left rotation operation described above. Imagine we have this situation:  
Figure 1-3

|  |
| --- |
| c  /  b / a |

To fix this, we will perform a single right rotation, rooted at C. This is done in the following steps:  
b becomes the new root.  
c takes ownership of b's right child, as its left child. In this case, that value is null.  
b takes ownership of c, as it's right child.  
  
The resulting tree:  
Figure 1-4

|  |
| --- |
| b / \ a c |

**Left-Right Rotation (LR) or "Double Rotation left"**  
Sometimes a single left rotation is not sufficient to balance an unbalanced tree. Take this situation:  
Figure 1-5

|  |
| --- |
| a \  c |

Perfect. It's balanced. Let's insert 'b'.  
Figure 1-6

|  |
| --- |
| a \  c /  b |

Our initial reaction here is to do a single left rotation. Let's try that.  
Figure 1-7

|  |
| --- |
| c / a \  b |

Our left rotation has completed, and we're stuck in the same situation. If we were to do a single right rotation in this situation, we would be right back where we started. What's causing this? The answer is that this is a result of the right sub-tree having a negative balance. In other words, because the right sub-tree was left heavy, our rotation was not sufficient. What can we do? The answer is to perform a right rotation on the right sub-tree. Read that again. We will perform a right rotation on the right sub-tree. We are not rotating on our current root. We are rotating on our right child. Think of our right sub-tree, isolated from our main tree, and perform a right rotation on it:  
**Before:**  
Figure 1-8

|  |
| --- |
| c / b |

**After:**Figure 1-9

|  |
| --- |
| b \  c |

After performing a rotation on our right sub-tree, we have prepared our root to be rotated left. Here is our tree now:  
  
  
Figure 1-10

|  |
| --- |
| a \  b  \  c |

Let's do the left rotation  
Figure 1-11

|  |
| --- |
| b / \ a c |

Problem solved by double rotation  
**Right-Left Rotiation (RL) or "Double rotation right"**

A double right rotation, or right-left rotation, or simply RL, is a rotation that must be performed when attempting to balance a tree which has a left sub-tree, that is right heavy. This is a mirror operation of what was illustrated in the section on Left-Right Rotations, or double left rotations. Let's look at an example of a situation where we need to perform a Right-Left rotation.  
Figure 1-12

|  |
| --- |
| c / a \  b |

In this situation single right rotation will not solve our problem. Let's try it:  
Figure 1-13

|  |
| --- |
| a \  c / b |

Looks like that didn't work. Now we have a tree that has a balance of 2. It would appear that we did not accomplish much. That is true. What do we do? Well, let's go back to the original tree, before we did our pointless right rotation:

Figure 1-14

|  |
| --- |
| c / a \  b |

The reason our right rotation did not work, is because the left sub-tree, or 'a', has a positive balance factor, and is thus right heavy. Performing a right rotation on a tree that has a left sub-tree that is right heavy will result in the problem we just witnessed. What do we do? The answer is to make our left sub-tree left-heavy. We do this by performing a left rotation our left sub-tree. Doing so leaves us with this situation:  
Figure 1-15

|  |
| --- |
| c  /  b / a |

This is a tree which can now be balanced using a single right rotation. We can now perform our right rotation rooted at C. The result:  
Figure 1-16

|  |
| --- |
| b / \ a c |

**A Balanced Tree:**2. Rotations, When to Use Them and Why?  
A tree rotation is necessary when you have inserted or deleted a node which leaves the tree in an unbalanced state.   
This is a balanced tree:  
Figure 2-1

|  |
| --- |
| 1 / \ 2 3 |

This is an unbalanced tree:  
  
Figure 2-2

|  |
| --- |
| 1 \  2  \  3 |

In figure 2-2, we see that the tree is considered "right heavy". We can correct this by performing "left rotation". How we determine which rotation to use follows a few basic rules. See pseudo code:

|  |  |
| --- | --- |
| IF tree is right heavy  {   IF tree's right sub-tree is left heavy   {   Perform Double Left rotation    }   ELSE   {   Perform Single Left rotation  }  }  ELSE IF tree is left heavy | {   IF tree's left sub-tree is right heavy   {   Perform Double Right rotation   }   ELSE   {   Perform Single Right rotation   }  } |

As you can see, there is a situation where we need to perform a "double rotation". A single rotation in the situations described in the pseudo code leave the tree in an unbalanced state. Follow these rules, and you should be able to balance an AVL tree following an insert or delete every time.  
  
**Implementation of AVL:**

|  |  |
| --- | --- |
| #include<iostream>  using namespace std;  class BST  {  struct node  {  int data;  node\* left;  node\* right;  int height;  };  node\* root;  void makeEmpty(node\* t)  {  if(t == NULL)  return;  makeEmpty(t->left);  makeEmpty(t->right);  delete t;  }  node\* insert(int x, node\* t)  {  if(t == NULL)  {  t = new node;  t->data = x;  t->height = 0;  t->left = t->right = NULL;  }  else if(x < t->data)  {  t->left = insert(x, t->left);  if(height(t->left) - height(t->right) == 2)  {  if(x < t->left->data)  t = singleRightRotate(t);  else  t = doubleRightRotate(t);  }  }  else if(x > t->data)  {  t->right = insert(x, t->right);  if(height(t->right) - height(t->left) == 2)  {  if(x > t->right->data)  t = singleLeftRotate(t);  else  t = doubleLeftRotate(t);  }  }  t->height = max(height(t->left), height(t->right))+1;  return t;  }  node\* singleRightRotate(node\* t)  {  node\* u = t->left;  t->left = u->right;  u->right = t;  t->height = max(height(t->left), height(t->right))+1;  u->height = max(height(u->left), t->height)+1;  return u;  }  node\* singleLeftRotate(node\* t)  {  node\* u = t->right;  t->right = u->left;  u->left = t;  t->height = max(height(t->left), height(t->right))+1;  u->height = max(height(t->right), t->height)+1 ;  return u;  }  node\* doubleLeftRotate(node\* t)  {  t->right = singleRightRotate(t->right);  return singleLeftRotate(t);  }  node\* doubleRightRotate(node\* t)  {  t->left = singleLeftRotate(t->left);  return singleRightRotate(t);  }  node\* findMin(node\* t)  {  if(t == NULL)  return NULL;  else if(t->left == NULL)  return t;  else  return findMin(t->left);  }  node\* findMax(node\* t)  {  if(t == NULL)  return NULL;  else if(t->right == NULL)  return t;  else  return findMax(t->right);  }  node\* remove(int x, node\* t)  {  node\* temp;  // Element not found  if(t == NULL)  return NULL;  // Searching for element  else if(x < t->data)  t->left = remove(x, t->left);  else if(x > t->data)  t->right = remove(x, t->right); | // Element found  // With 2 children  else if(t->left && t->right)  {  temp = findMin(t->right);  t->data = temp->data;  t->right = remove(t->data, t->right);  }  // With one or zero child  else  {  temp = t;  if(t->left == NULL)  t = t->right;  else if(t->right == NULL)  t = t->left;  delete temp;  }  if(t == NULL)  return t;  t->height = max(height(t->left), height(t->right))+1;  // If node is unbalanced  // If left node is deleted, right case  if(height(t->left) - height(t->right) == 2)  {  // right right case  if(height(t->left->left) - height(t->left->right) == 1)  return singleLeftRotate(t);  // right left case  else  return doubleLeftRotate(t);  }  // If right node is deleted, left case  else if(height(t->right) - height(t->left) == 2)  {  // left left case  if(height(t->right->right) - height(t->right->left) == 1)  return singleRightRotate(t);  // left right case  else  return doubleRightRotate(t);  }  return t;  }  int height(node\* t)  {  return (t == NULL ? -1 : t->height);  }  int getBalance(node\* t)  {  if(t == NULL)  return 0;  else  return height(t->left) - height(t->right);  }  void in\_order(node\* t)  {  if(t == NULL)  return;  in\_order(t->left);  cout << t->data << " ";  in\_order(t->right);  }  public:  BST()  {  root = NULL;  }  void insert(int x)  {  root = insert(x, root);  }  void remove(int x)  {  root = remove(x, root);  }  void display()  {  in\_order(root);  cout << endl;  }  };  int main()  {  BST t;  t.insert(67);  t.insert(43);  t.insert(21);  t.insert(10);  t.insert(89);  t.insert(38);  t.insert(69);    t.display();  t.remove(89);  t.remove(43);  t.remove(88);  t.remove(20);  t.display();  } |