Trees

Trees are used to organize data in many software applications, including:

- Databases
- Data compression
- Bitcoin
- Medical diagnosis

A tree is a special linked list-based data structure that has many uses in computer science:

- To organize hierarchical data
- To make information easily searchable
- To amplify the evaluation of mathematical expressions
- To make decisions

Basic Tree Facts

1. Trees are made of **nodes** (just like linked list nodes)
2. Every tree has a **"root"** pointer
3. The top node of a tree is called its **"root" node**.
4. Every node may have zero or more **"children"** nodes.
5. A node with **0** children is called a **"leaf"** node.
6. A tree with no nodes is called an **"empty tree"**.

```
struct node

{

    int value;

    node *left, *right; //instead of just one next pointer,

                        //a tree node can have two or more next pointers!

};

node *rootPtr;
```

Binary Trees
A **binary tree** is a special form of tree. In a binary tree, every node has at most **two children nodes**: a **left** child and a **right** child.

```
struct BTNODE //binary tree node

{

    string value;

    BTNODE *pLeft, *pRight;

}
```

We can pick any node in the tree, and then focus on its **"subtree"** - which includes it and all the nodes below it.

Operations on Binary Trees:

- Enumerating all the items
- Searching for an item
- Adding a new item at a certain position on the tree
- Deleting an item
- Deleting the entire tree (destruction)
- Removing a whole seciton of a tree (called **pruning**)
- Adding a whole section to a tree (called **grafting**)

A simple tree example

```
struct BTNODE //node
{
    int value; //data
    BTNODE *left, *right;
};
int main()
{
    BTNODE *temp, *pRoot;
    pRoot = new BTNODE;
    pRoot->value = 5;

    temp = new BTNODE;
    temp->value = 7;
    temp->left = NULL;
    temp->right = NULL;
    pRoot->left = temp;

    temp = new BTNODE;
    temp->value = -3;
    temp->left = NULL;
    temp->right = NULL;
    pRoot->right = temp;
}
```

A full binary tree is one in which every leaf node has the same depth and every non-leaf has exactly two children.

Traversing the binary tree
When we iterate through all the nodes in a tree, it's called a traversal. Any time we traverse through a tree, we always start with the root node.

1. Pre-order traversal
2. In-order traversal
3. Post-order traversal
4. Level-order traversal

The pre-order traversal

1. Process the current node
2. Process the nodes in the left sub-tree
3. Process the nodes in the right sub-tree

```
void PreOrder(Node *cur)

{

    if(cur == NULL)

        return;

    cout << cur->value;

    PreOrder(cur->left);

    PreOrder(cur->right);

}
```

The in-order traversal

1. Process the nodes in the left sub-tree
2. Process the current node
3. Process the nodes in the right sub-tree

```
void InOrder(Node *cur)

{

    if(cur == NULL)

    {

        return;

    }

    InOrder(cur->left);

    cout << cur->value;

    InOrder(cur->right);
```

```
}
```

The post-order traversal

1. Process the nodes in the left sub-tree
2. Process the nodes in the right sub-tree
3. Process the current node

```
void PostOrder(Node *cur)

{

    if(cur == NULL)

        return;

    PostOrder(cur->left);

    PostOrder(cur->right);

    cout << cur->value;

}
```

The level-order traversal
Visit each level's nodes, from left to right, before visiting nodes in the next level
Algorithm

1. Use a temp pointer variable and a queue of node pointers
2. Insert the root node pointer into the queue
3. While the queue is not empty:
    1. Dequeue the top node pointer and put it in temp
    2. Process the node
    3. Add the node's children to queue if they are not NULL

Since a traversal must process each node exactly once, and there are N nodes in a tree,

> The Big-O for any of the traversals is O(N)

Binary Search Trees

Binary search trees are a type of binary tree with specific properties that make them very efficient to search for a value in the tree. Like regular binary trees, we store and search for values in binary search trees.
For every node X in the tree, all nodes in X's left sub-tree must be less than X. All nodes in X's right subtree must be greater than X.
Here's what we can do to a binary search tree:

- Determine if the binary search tree is empty
- Search the binary tree for a value
- Insert an item in the binary search tree
- Delete an item from the binary search tree
- Find the height of the binary search tree

- Find the number of nodes and leaves in the binary search tree
- Traverse the binary search tree
- Free the memory used by the binary search tree

Searching a BST

Algorithm (value V to search for)

- Start at the root of the tree
- Keep going until we hit the NULL pointer
- If V is equal to current node's value, then found!
- If V is less than current node's value, go left
- If V is greater than current node's value, go right
- If we hit a NULL pointer, not found

```
bool search(int V, Node *ptr)
{
    if (ptr == NULL)
        return false;
    else if (V == ptr->value)
        return true;
    else if (V < ptr->value)
        return search(V, ptr->left);
    else if (V > ptr->value)
        return search(V, ptr->right);
}
```

```
bool search(int V, Node *ptr)
{
    while(ptr != NULL)
    {
        if(ptr->value == V)
            return true;
        else if (ptr->value < V)
            ptr = ptr->left;
        else if (ptr->value > V)
            ptr = ptr->right;
```

```
    }

    return false;

}
```

Big-O of BST search: **O(log2(N))**

Inserting a New Value into a BST
To insert a new node into our BST, we must place the new node so that the resulting tree is still a valid BST!
Algorithm:

- If the tree is empty...
  - Allocate a new node and put V into it
  - Point the root pointer to our new node. DONE!
- Start at the root of the tree
- While we're not done...
  - If V is equal to current node's value, DONE!
  - If V is less than current node's value
    - If there is a left child, then go left
    - Else, allocate a new node and put V into it and set current node's left pointer to new node
  - If V is greater than current node's value
    - If there is a right child, then go right
    - Else, allocate a new node and put V into it, set current node's right pointer to new node. DONE!

```
struct Node

{

    Node(const std::string &myVal)

    {

        value = myVal;

        left = right = NULL;

    }

    std::string value;

    Node *left, *right;

};

void insert(const std::string &value)

{

    if (m_root == NULL)

    {

        m_root = new Node(value);

        return;

    }
```

```
    Node *cur  = m_root;
    for(;;)
    {
        if (value == cur->value)
            return;
        if(value < cur->value)
        {
            if(cur->left != NULL)
                cur = cur->left;
            else
            {
                cur->left = new Node(value);
                return;
            }
        }
        else if (value > cur->value)
        {
            if (cur->right != NULL)
            {
                cur = cur->right;
            }
            else
            {
                cur->right = new Node(value);
                return;
            }
        }
    }
}
```

Big-O of Inserting a New Value into a BST: **O(log2(N))**
Recursive approach

```
void insert(const std::string &value, Node *cur)
{
    if(m_cur == NULL)
    {
        return;
    }
    if(value == m_cur->value)
    {
        return;
    }
    if(value < m_cur->value)
    {
        if(cur->left == NULL)
            cur->left = new Node(value);
        else
            insert(value, cur->left);
    }
    else
    {
        if(cur->right == NULL)
            cur->right = new Node(value);
        else
            insert(value, cur->right);
    }
}
```

Finding the Min and Max of a Binary Search Tree

```
int GetMin(node *pRoot)
{
    if(pRoot == NULL)
        return -1;
```

```
    while(pRoot->left != NULL)
        pRoot = pRoot->left;
    return pRoot->value;
}
int GetMax(node *pRoot)
{
    if(pRoot == NULL)
        return -1;
    while(pRoot->right != NULL)
        pRoot = pRoot->right;
    return pRoot->value;
}
```

Recursive versions

```
int GetMin(node *pRoot)
{
    if(pRoot == NULL)
        return -1;
    if(pRoot->left == NULL)
        return pRoot->value;
    return GetMin(pRoot->left);
}
int GetMax(node *pRoot)
{
    if(pRoot == NULL)
        return -1;
    if(pRoot->right == NULL)
        return pRoot->value;
    return GetMax(pRoot->right);
}
```

Printing a Binary Search Tree in Alphabetical Order
==> Use the in-order traversal
Big-O of printing all the items in the tree is **O(N)**

```
void print(Node *root)
{
    if(root == NULL)
        return;
    print(root->left);
    cout << root->value;
    print(root->right);
}
```

Freeing the Whole Tree
==> Use the post-order traversal
Big-O is still **O(N)**

```
void FreeTree(Node *cur)
{
    if(cur == NULL)
        return;
    FreeTree(cur->left);
    FreeTree(cur->right);
    delete cur;
}
```

Binary Search Tree **Node Deletion**
Algorithm (given a value V to delete from the tree):

- Find the value V in the tree, with a slightly-modified BST search
  - Use two pointers: a cur pointer and a parent pointer
- If the node was found, delete it from the tree, making sure to preserve its ordering!
- There are three cases

BST Deletion Step #1
cur should point at the node we want to delete, and **parent** points to the node above it!

```
parent = NULL;
cur = root;
```

```
while (cur != NULL)
{
    if (cur->value == V)
        return;
    if (V < cur->value)
    {
        parent = cur;
        cur = cur->left;
    }
    else if (V > cur->value)
    {
        parent = cur;
        cur = cur->right;
    }
}
```

Step #2
Once we've found our target node, we have to delete. There are 3 cases:

- 1: Our node is a leaf. (Two sub-cases)
  - 1a) The target node is NOT the root node
  - 1b) The target node is the root node
- 2: Our node has one child.
  - 1a) The target node is NOT the root node
  - 1b) The target node is the root node
- 3: Our node has two children.

Case 1
Sub-case a: The target node is NOT the root node

1. Unlink the parent node from the target node (**cur**) by setting the parent's appropriate link to NULL
2. Delete the target (**cur**) node.

```
parent->right = NULL;
//OR
parent->left = NULL;
delete cur;
```

Sub-case b: The target node is the root node

1. Set the root pointer to NULL
2. Then delete the target (**cur**) node

```
root = NULL;

delete cur;
```

Case 2
Sub-case a: The target node is NOT the root node

1. Re-link the parent node to the target (**cur**) node's only child.
2. Then delete the target (**cur**) node.

```
parent->left = cur->left;

//OR

parent->right = cur->left;

//OR

parent->left = cur->right;

//OR

parent->right = cur->right;

delete cur;
```

Sub-case b: The target node is the root node.

1. Re-link the root pointer to the target's only child.
2. Then delete the target node.

```
root = cur->left;

//OR

root = cur->right;

delete cur;
```

Case 3
We don't actually delete the node itself. Instead, we replace its value with one from another node!
Let's say we want to delete a node **k** that has two children.
We will replace **k** with either

- k's left subtree's largest valued child
- k's right subtree's smallest valued child

So we pick one up, copy its value up, then delete that node! They'll either be a leaf or have just one child.
This ensures we can use one of our simpler deletion algorithms for the replacement!

Balanced BSTs are always **O(log n)** for insertion and deletion. If ever asked a BST question in a job/internship interview, make sure to ask the interviewer if you may assume the BST is balanced!