Trees

Introduction

A tree is a data structure that stores values in a hierarchical fashion

We often use **linked lists** to build trees

Basic Tree Facts

- Trees are made of **nodes**
- Every tree has a root pointer

```
struct node
{
    int value;
    node *left, *right;
};
node *rootPtr;
```

- The top node of a tree is the root node
- Every node may have 0 or more children nodes
- A node with 0 children is a leaf node
- A tree with no nodes is called an empty tree

A tree node can have ore than just 2 children

```
struct node
{
    int value;
    node *pChild1, *pChild2, *pChild3, ...;
}

struct node
{
    int value;
    node *pChildren[26];
}
```

Binary Trees

- Every node has at most two children nodes: **left and right**

```
struct BTNODE
{
    string value;
    BTNODE *pLeft, *pRight;
}
```

Operations on Binary Trees

Common Operations

- Enumerating all the tiems
- Searching for an item
- Adding a new item at a certain position on the tree
- Deleting an item
- Deleting the entire tree
- Remove a whole section of a tree (pruning)
- Adding a whole section to a tree (grafting)

Example

```
struct BTNODE
{
    int value;
    BTNODE *left, *right;
}

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;
    pRight->right = temp;
}
```

Binary Tree Traversals

Four common ways

- Pre-order
- In-order
- Post-order (^Recursion)
- Level-order (BFS)

Pre-order

```
void preorder(Node *p)
{
    if (p == nullptr)
        return;
    cout << p->value;
    preorder(p->left);
    preorder(p->right);

}
```

- handle base case: empty tree
- order
  - process current node's value
  - fully process left subtree of current node
  - fully process right subtree of current node

Inorder

```
void inorder(Node *p)
{
    if (p == nullptr)
        return;
    inorder(p->left);
    cout << p->value;
    inorder(p->right);
}
```

- process left first, root, then right

Postorder

```
void postorder(Node *p)
{
    if (p == nullptr)
        return;
    postorder(p->left);
    postorder(p->right);
    cout << p->value;
}
```

Level-order Traversal

We start at the root and visit each level's nodes, from left to right, before visiting nodes in the next level

Algorithm

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

---

Big-O of Traversals

Each of our traversals performs three operations per node:

- Process the value in the current node
- Initiate processing of its left subtree
- Initiate processing of its right subtree

For a tree with n nodes, that's 3*n operations: **O(n)**

Traversal Use Case: Expression Evaluation

```
(5+6)*(3-1)
```

Algorithm

1. If the current node is a number, return its value
2. Recursively evaluate the left subtree and get the result

3. Recursivley evaluate the right subtree and get the result

4. Apply the operator in the current node to the left and right results; return result

- A post order traversal

```
int eval(Node *p)
{
    if (p->type == VALUE)
        return p->value;
    int left = eval(p->left);
    int right = eval(p->right);
    return apply(p->operator, left, right);
}
```

Binary Search Trees

A binary search tree enables fast (log2N) searches by ordering its data in a special way

> For every node j in the tree, all children to j's left must be less than it, and all children to j's right must be greater than it

To see if a value V is in the tree:

1. Start at the root node

2. Compare V against the node, moving down left or right if V is less or greater

3. Repeat until you find V or hit a dead end

Operations on a Binary Search Tree

Searching a BST

Input: A value V to search for

Output: TRUE if found, FALSE otherwise

Algorithm

- 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

Two different search algorithms: iterative and recursive

```
bool Search(int v, Node *root)
{
    Node *ptr = root;
    while (ptr != nullptr)
    {
        if (v == ptr->value)
            return true;
        else if (v < ptr->value)
            ptr = ptr->left;
        else
            ptr = ptr->right;
    }
    return false; //nope
}
```

```
bool Search(int v, Node *root)
{
    if (root == nullptr)
        return false;
    else if (v == root->value)
        return true;
    else if (v < root->value)
        return Search(V, root->left);
    else
        return Search*V, root->right);
}
```

Big-O of BST Search

In the average BST with N values, how many steps are required to find our value?

- log2N steps

In the worst case BST with N values, how mnay steps are required to find our value?

- N steps

Inserting a New Value Into A BST

To insert a new node in our BST, we must place the new node so that the resulting tree is still a valid BST

Input: A value V to insert

Algorithm:

- If the tree is empty
  - Allocate a new node and put V into it

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

Code

```
struct Node
{
    Node(const std::string &myVal)
    {
        value = myVal;
        left = right = nullptr;
    }
    std::string value;
    Node *left, *right;
}
```

```
class BST
{
    public:
        BST()
        {
            m_root = nullptr;
        }

        void insert(const std::string &value)
        {
            ...
        }
    private:
        Node *m_root;
};
```

- our BST class has a single member variable, the root pointer to the tree
- our constructos initializes that root pointer to nullptr when we create a new tree (indicate tree is empty)

```
void insert(const std::string &value)
{
    if (m_root == nullptr)
        { m_root = new Node(value); return; }
    Node *cur = m_root;
    for (;;)
    {
        if (value == cur->value) return;
        if (value < cur->value)
        {
            if (cur->left != nullptr)
                cur = cur->left;
            else
            {
                cur->left = new Node(value);
                return;
            }
        }
        else if (value > cur->value)
        {
            if (cur->right != nullptr)
                cur = cur->right;
            else
            {
                cur->right = new Node(value);
                return;
            }
        }
    }
}
```

- If our tree is empty, allocate a new node and point the root pointer to it
- Start traversing down from the root of the tree
    - `for(;;)` is the same as an infinite loop
- If our value is already in the tree, then we're done, return
- If the value to insert is less than the current node's value, go left
- If there is a node to our left, advance to that node and continue
    - Otherwise, we've found the proper spot for our new value
    - Add our value as the left child of the current node
- If the value we want to isnert is greater than the current node's value, then traverse/insert to the right

> As with BST Search, there is a **recursive version** of the Insertion algorithm too!

Big-O of BST Insertion

O(log2n)

- We have to first use a binary serach to find where to insert out node and binary search is O(log2n)
- Once we found the right spot, we can insert our new node in O(1) time

Finding Min & Max of a BST

How do we find the minimum and maximum values in a BST?

> The minimum value is located at the left-most node

> The maximum value is located at the right-most node

```
int GetMin(node *pRoot)
{
    if (pRoot == nullptr)
        return -1; //empty

    while (pRoot->left != nullptr)
        pRoot = pRoot->left;

    return pRoot->value;
}
```

```
int GetMax(Node *pRoot)
{
    if (pRoot = nullptr)
        return -1;

    while (pRoot->right != nullptr)
        pRoot = pRoot->right;

    return pRoot->value;
}
```

Big-O

O(log2(n))

- We just go to the bottom of the tree

Recursion

```
int GetMin(node *pRoot)
{
    if (pRoot == nullptr)
        return -1; //empty

    if (pRoot->left == nullptr)
        return pRoot->value;

    return GetMin(pRoot->left);
}
```

```
int GetMax(Node *pRoot)
{
    if (pRoot = nullptr)
        return -1;

    if (pRoot->right == nullptr)
        return pRoot->value;

    return GetMax(pRoot->right);
}
```

Printing a BST In Alphabetical Order

In order traversal

```
void InOrder(Node* cur)
{
    if (cur == nullptr)
        return;

    InOrder(cur->left);
    cout << cur->value;
    InOrder(cur->right);
}
```

Big-O: O(n)

- Since we have to visit and print all n items

Freeing The Whole Tree

It's another traversal s

```
void FreeTree(Node* cur)
{
```

```
    if (cur == nulltpr)
        return;
    FreeTree(cur->left); //delete nodes in left sub-tree
    FreeTree(cur->right); //delete nodes in right sub-tree
    delete cur; //free current node
}
```

Big-O: O(n)

Deleting a Node from a Binary Search Tree

If we remove a node in the middle of our tree, we need to reorder many nodes to have a valid binary search tree

High-level algorithm

Given a value V to delete from the tree

1. Find the value V in the tree, with a slightly-modified BST search
    - Use two pointers: a `cur` pointer & a `parent` pointer
2. If the node was found, delete it from the tree, making sure to preserve its ordering
    - Three cases

Algorithm

Step 1: Searching for value V

1. `parent = nullptr`
2. `cur = root`
3. `while (cur != nullptr)`

    a. `If (V == cur->value)` then we're done

    b. `If (V < cur->value)`

    `parent = cur;`

    `cur = cur->left;`

    c. `Else if (V > cur->value)`

    `parent = cur;`

    `cur = cur->right;`

- Similar to traditional BST search, but **has a parent pointer**
- When we are done with our loop below, we want the **parent pointer** to point to the **node just above the target node** we want to delete

Step 2: Once we've found our target node, we have to delete it

1. Case 1: Our node is a leaf
    - Subcase 1: 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`
            - If target node is parent node's right child, set `parent->right = nullptr`
        2. Delete the target node `cur`
    - Subcase 2: The target node IS the root node
        1. Set the root **pointer** to `NULL`
        2. Then delete the target node `cur`
2. Case 2: Our node has one child
    - Subcase 1: The target node is NOT the root node
        1. Relink the parent node to the target node's only child
        2. Then delete the target node `cur`
    - Subcase 2: The target node IS the root node
        1. Relink the root pointer to the target node's only child
        2. Delete the target node `cur`
3. Case 3: Our node has two children
    - We need to find a replacement for our target node that still leaves the BST consistent, we can't just pick some arbitary node and move it up into the vacated slot
    - We **don't actually delete the node itself**, we **replace its value** with one from **another node**
    - Replace `cur` with either:
        1. `cur`'s left subtree's largest-valued child
        2. `cur`'s right subtree's smallest-valued child
        - Both of these are either a leaf or have just one child
            - We found the left subtree's max value by going all the way to the right
                - By definition it can't have a right child
                - Either has a left child or no children at all
            - For smallest value in right subtree, by def cannot have a left child

        - Pick one, copy its value up, then delete that node
    - Delete this node with above 1 or 2 methods

Appications of BST

STL map and set

Map and set use a type of special balanced BST to store the items

Huffman Encoding

Huffman Encoding is a data compression technique that can be used to compress and decompress files

ASCII

Each character stored as a number, characters are stored in the computer's memory as numbers

High level Algorithm

1. Compute the frequency of each character in the file.dat
2. Build a Huffman tree (a binary tree) based on these frequencies
    - Create a binary tree lead node for each entry in our table, but don't insert any of these into a tree!
    - Build a binary tree from the leaves
        - While we have more than one node left
            1. Find the two nodes with lowest frequencies
            2. Create a new parent node
            3. Link the parent to each of the children
            4. Set the parent's total frequency equal to the sum of its children's frequencies
            5. Place the new parent node in our grouping
    - Now label each left edge with a "0" and each right edge with a "1"
        - Now we can determine the new bit-encoding for each character
        - The bit encoding for a character is the path of 0's and 1's that you take from the root of the tree to the character of interest
3. Use this binary tree to convert the original file's contents to a more compressed form
    - Find the seqeucne of bits for each char in the message
4. Save the converted (compressed) data to a file

- The data + some meta data to tell machine the specifications of the encoding

Decoding

1. Extract the encoding scheme from the compressed file
2. Build a Huffman tree based on the encodings
3. Use this binary tree to convert the compressed file's content back to the original characters
4. Save the converted (uncompressed) data to a file

Balanced Search Trees

Ensure all insertions, searches, and deletions would be O(logn)

Everytime you add/delete a value, they automatically shift the nodes around so the tree is balanced

AVL Tree

Tracks the height of ALL subtrees in the BST

After an insertion/deletion, if the height of the subtrees under any node is different by more than one level (e.g. right subtree has height 5, left subtree has height 3)

- Then the AVL algorithm shifts the nodes around to maintain balance

> Balanced BSTs are always O(log2N) for insertion and deletion