Trees 3 🌴

Everything on the left should be smaller than the right within a binary search tree. In other words the smallest element must be on the left.

- The complexity must be $O(n)$, due to all nodes needing to be traversed
- Assumes nodes are organized in a totally ordered binary tree

Consequences

- The smallest element in a binary search tree is the "leftmost" node.
- The largest element is the rightmost node
- Inorder traversal of the BST encounters nodes in increasing order

Search In BST

- Compare the search value to the current node, and decide whether to go left or right (depending if less (left) or more (right).
- Runtime ≤ descending path length ≤ depth of tree or height of tree

```C++
/* How It Works */

// Note: This is my personal code idea not concrete implementation, check documentation for better code example

void TreeAddNode(Node* currNode, Node* nodeAdd){
        if(currNode != nullptr){
                if(currNode->val > nodeAdd->val){
                        TreeAddNode(currNode->left, nodeAdd);
                }
                else if(currNode->val < nodeAdd->val){
                        TreeAddNode(currNode->right, nodeAdd);
                }
        } else {
                if(currNode->val > nodeAdd->val){
                        currNode->left = nodeAdd;
                }
                else if(currNode->val < nodeAdd->val){
                        currNode->right = nodeAdd;
                }
        } return;
}
```

Delete Node From Tree

- Must restructure the tree
- Find the similar branches to restructure
- Pick largest node in the subtree to be a new root

Finding the Minimum

- We can do this recursively, by going all the way to the left, aka following all of the left nodes
- If we have no left notes then we are done with following the tree (we have found the minimum value)

Tail Recursion

- Recursion is in the last line of the program
- Can be replaced with a loop (some compilers do this by default), very efficient

Finding the Maximum

- Non recursive

Implementation :

```C++
Node* findMax(){
        if(t!= nullptr){
                while(t->right != nullptr){
                        t=t->right;
                }
                return t;
        }
}
```

Deletion Example

```C++
#include <iostream>
using namespace std;

void deleteNode(const Comparable& x, BinaryNode* &t){
    if(t == nullptr){
        return;
    }
    if(x <t>element){
        remove (x, t->left);
    }
    else if(t->left != nullptr && t->right != nullptr){
        t->element = findMin(t->right)->element;
        remove(t->element, t->right);
    } else{
        BinaryNode * oldNode = t;
        t = (t->left != nullptr) ? t->left: t->right;
        delete oldNode;
    }
```

Destructor

- Left tree first
- Right tree second
- loop, Post order traversal, can use any traversal

Course Code Examples

IO Examples :

```C++
// IO Examples

Template <typename Comparable>

Class BinarySearchTree {

   public:
```

```
BinarySearchTree();

BinarySearchTree(const BinarySearchTree & rhs); // copy

BinarySearchTree(BinarySearchTree &&rhs); // move

~BinarySearchTree();


const Comparable & findMin() const;

const Comparable & findMax() const;

bool contains(const Comparable &x) const;

bool isEmpty() const;

void printTree(ostream & out = std::cout) const;


void makeEmpty();

void insert(const Comparable &x);

void insert(Comparable &&x); // move

void remove(const Comparable &x);


BinarySearchTree & operator=(const BinarySearchTree &rhs);

BinarySearchTree & operator=(BinarySearchTree && rhs); // move
```

Finding Smallest Element :

```C++
BinaryNode * findMin( BinaryNode *t ) const

    {

        if( t == nullptr )

            return nullptr;

        if( t->left == nullptr )

            return t;

        return findMin( t->left );

    }
```

Finding the Largest Element :

```cpp
BinaryNode * findMax( BinaryNode *t ) const
    {
        if( t != nullptr )
            while( t->right != nullptr )
                t = t->right;
        return t;
}
```

Deletion :

```cpp
void remove( const Comparable & x, BinaryNode * & t ) {

        if( t == nullptr )

            return;   // Item not found; do nothing

        if( x < t->element )
            remove( x, t->left );
        else if( t->element < x )
            remove( x, t->right );
        else if( t->left != nullptr && t->right != nullptr )  { // two children
            t->element = findMin( t->right )->element;
            remove( t->element, t->right );
        }
        else {
            BinaryNode *oldNode = t;
            t = ( t->left != nullptr ) ? t->left : t->right;
            delete oldNode;
        }
    }
```

Destructor :

```cpp
    ~BinarySearchTree( )
    {
        makeEmpty( );
    }

    /**
     * Internal method to make subtree empty.
     */
    void makeEmpty( BinaryNode * & t )
    {
        if( t != nullptr )
        {
            makeEmpty( t->left );
            makeEmpty( t->right );
            delete t;
        }
        t = nullptr;
    }
```

Inorder Traversal :

```cpp
    // Print the tree contents in sorted order.
    void printTree( ostrem & out ) const
    {
```

```
        if( isEmpty( ) )
            cout << "Empty tree" << endl;
        else
            printTree( root, out);
    }

    /**
     * Internal method to print a subtree rooted at t in sorted order.
     */
    void printTree( BinaryNode *t, ostream & out ) const
    {
        if( t != nullptr )
        {
            printTree( t->left );
            out << t->element << endl;
            printTree( t->right );
        }
    }
```