**Introduction:**

I created a Binary Search Tree that uses numbers. Therefore, if the number is bigger than the first number, it goes down the left branch, if it is smaller, it goes down the right branch. If there is another number, it does another comparison to figure out which branch to follow until it reaches an empty spot. The add method adds a number to the tree, the find method finds the number in the tree (is used to delete said number), findMax finds the item with the largest key, deleteing uses find to locate the number and the deletes the number differently depending on if it has none, one, or two children, and display shows the tree and all of its legs.

**Programmer’s Guide:**

**// Node structure - Used to make each node. Automatically makes a root node.**

struct Node

{

   int key;

   int number;

   struct Node \*left;

   struct Node \*right;

}\*root;

**// Create the class - Include all the methods needed**

class binaryTree

{

public:

   void insert(Node \*, Node \*, int);

   void find(int, Node \*\*, Node \*\*);

   void findMax();

   void deleting(int);

   void noChildren(Node \*, Node \*);

   void oneChild(Node \*, Node \*);

   void twoChildren(Node \*, Node \*);

   void display(Node \*, int);

   binaryTree() // Initiallizer

   {

      root = NULL;

   }

};

**// Insert numbers into the tree**

void binaryTree::insert(Node \*tree, Node \*newnode, int key)

{

**// Tree is empty == create the root**

   if (root == NULL)

   {

      root = new Node;

      root->number = newnode->number;

      root->key = key;

      root->left = NULL;

      root->right = NULL;

      cout << "Root Node has been added" << endl;

      return;

   }

**// If it's already in the tree, don’t add it, again**

   if (tree->number == newnode->number)

   {

      cout << "That number is already in the tree" << endl;

      return;

   }

**// If it's a bigger number, add it to the left leg**

   if (tree->number > newnode->number)

   {

      if (tree->left != NULL)

      {

          insert(tree->left, newnode, key);

      }

      else

      {

          tree->left = newnode;

          (tree->left)->key = key;

          (tree->left)->left = NULL;

          (tree->left)->right = NULL;

          cout << "Node added to the left leg." << endl;

          return;

      }

   }

**// Otherwise add it to the right leg**

   else

   {

      if (tree->right != NULL)

      {

          insert(tree->right, newnode, key);

      }

      else

      {

          tree->right = newnode;

          (tree->right)->key = key;

          (tree->right)->left = NULL;

          (tree->right)->right = NULL;

          cout << "Node Added To Right" << endl;

          return;

      }

   }

}

**// Find a number**

void binaryTree::find(int item, Node \*\*parent, Node \*\*location)

{

   Node \*legToFollow, \*legSave;

**// If tree is empty, there's nothing to find**

   if (root == NULL)

   {

      \*location = NULL;

      \*parent = NULL;

      return;

   }

**// If the root is the number**

   if (item == root->number)

   {

      \*location = root;

      \*parent = NULL;

      return;

   }

**// Follow the left legs**

   if (item < root->number)

   {

      legToFollow = root->left;

   }

**// Follow the right**

   else

   {

      legToFollow = root->right;

   }

   legSave = root;

**// Keep following til found or the end**

   while (legToFollow != NULL)

   {

      /**/ If found**

      if (item == legToFollow->number)

      {

          \*location = legToFollow;

          \*parent = legSave;

          return;

      }

      legSave = legToFollow;

      if (item < legToFollow->number)

      {

          legToFollow = legToFollow->left;

      }

      else

      {

          legToFollow = legToFollow->right;

      }

   }

   \*location = NULL;

   \*parent = legSave;

}

**// Find the number with the maximum key**

void binaryTree::findMax()

{

   Node \*legToFollow, \*legSave;

**// If tree is empty, there's nothing to find**

   if (root == NULL)

   {

      cout << "The tree is empty" << endl;

      return;

   }

**// If the root is the max key**

   if (root->left == NULL && root->right == NULL)

   {

      cout << "The item with the maximum key is " << root->number << endl;

      return;

   }

**// Follow the left legs if the key is bigger**

   if (root->left->key > root->right->key)

   {

      legToFollow = root->left;

   }

   /**/ Follow the right leg if the key is bigger on the right**

   else

   {

      legToFollow = root->right;

   }

   legSave = root;

**// Keep following til found or the end**

   while (legToFollow != NULL)

   {

      /**/ If found**

      if (legToFollow->left == NULL && legToFollow->right == NULL)

      {

          cout << "The item with the maximum key is " << legToFollow->number << endl;

          return;

      }

      legSave = legToFollow;

      if (legToFollow->left->key > legToFollow->right->key)

      {

          legToFollow = legToFollow->left;

      }

      else

      {

          legToFollow = legToFollow->right;

      }

   }

}

**// Deleting numbers from the tree**

void binaryTree::deleting(int item)

{

   Node \*parent, \*location;

**// If the tree is empty**

   if (root == NULL)

   {

      cout << “The tree is empty" << endl;

      return;

   }

**// Find the number to delete**

   find(item, &parent, &location);

**// If the number is not found**

   if (location == NULL)

   {

      cout << "That number could not be found" << endl;

      return;

   }

**// If the node has No Children**

   if (location->left == NULL && location->right == NULL)

   {

      noChildren(parent, location);

   }

**// If the node has One Child**

   if (location->left != NULL && location->right == NULL || location->left == NULL && location->right != NULL)

   {

      oneChild(parent, location);

   }

**// If the node has Two Children**

   if (location->left != NULL && location->right != NULL)

   {

      twoChildren(parent, location);

   }

   free(location);

}

**// Deleting a node with no children**

void binaryTree::noChildren(Node \*parent, Node \*location)

{

**// delete the root if it has no parent (If it is the root)**

   if (parent == NULL)

   {

      root = NULL;

   }

**// Delete the number**

   else

   {

      if (location == parent->left)

      {

          parent->left = NULL;

      }

      else

      {

          parent->right = NULL;

      }

   }

}

**// Deleting a node with one child**

void binaryTree::oneChild(Node \*parent, Node \*location)

{

   Node \*child;

**// Move the location of the child to the left or right node (where parent was)**

   if (location->left != NULL)

   {

      child = location->left;

   }

   else

   {

      child = location->right;

   }

**// Make the root the child**

   if (parent == NULL)

   {

      root = child;

   }

**// Move the parent of the child to the left or right node (where the parent's parent was)**

   else

   {

      if (location == parent->left)

      {

          parent->left = child;

      }

      else

      {

          parent->right = child;

      }

   }

}

**// Deleting a node with two children**

void binaryTree::twoChildren(Node \*parent, Node \*location)

{

   Node \*following, \*saveLoc, \*suc, \*parsuc;

   saveLoc = location;

   following = location->right;

**// While there's something to follow**

   while (following->left != NULL)

   {

      saveLoc = following;

      following = following->left;

   }

   suc = following; **// save follow spot**

   parsuc = saveLoc; **// save the location**

**// If both legs are null do no children deletion call deletion with no children**

   if (suc->left == NULL && suc->right == NULL)

   {

      noChildren(parsuc, suc);

   }

   else // One child deletion

   {

      oneChild(parsuc, suc);

   }

**// No parent means it is the root**

   if (parent == NULL)

   {

      root = suc;

   }

**// Move the children**

   else

   {

      if (location == parent->left)

      {

          parent->left = suc;

      }

      else

      {

          parent->right = suc;

      }

   }

   suc->left = location->left;

   suc->right = location->right;

}

**// Display the tree and all the legs**

void binaryTree::display(Node \*ptr, int level)

{

   int i;

**// Until everything has been displayed, show it as either left or right legs.**

   if (ptr != NULL)

   {

      display(ptr->right, level + 1);

      cout << endl;

      if (ptr == root)

      {

          cout << "Root->:  ";

      }

      else

      {

          for (i = 0; i < level; i++)

          {

              cout << "       ";

          }

      }

      cout << ptr->number;

      display(ptr->left, level + 1);

   }

}

**// Main Program**

int main()

{

   binaryTree tree; **// Regular tree**

   binaryTree tree100; **// 100 nodes tree**

   binaryTree tree1000; **// 1000 nodes tree**

   binaryTree tree10000; **// 10000 nodes tree**

   binaryTree tree100000; **// 100000 nodes tree**

   int numbers[] = { 25, 7, 17, 91, 8, 7, 42, 15, 83, 76, 100 }; **// Array of unsorted numbers**

   int key = 0;

   cout << "Deleting a number from an empty tree." << endl;

   tree.deleting(7); **// Deletion while tree is empty**

   system("pause");

   cout << endl;

**// Adding the array to the tree**

   for (int i = 0; i < 11; ++i)

   {

      Node \*temp = new Node;

      temp->number = numbers[i];

      tree.insert(root, temp, key);

      key++;

   }

   cout << endl << "All numbers in the array have been added." << endl;

   system("pause");

**// Display the tree with the array added**

   cout << endl << "The tree once all numbers have been added." << endl;

   tree.display(root, 1);

   cout << endl << endl << endl;

   system("pause");

**// Deleting root (two children)**

   cout << endl << "The tree once the root (25) has been deleted." << endl;

   tree.deleting(25);

   tree.display(root, 1);

   cout << endl << endl << endl;

   system("pause");

**// Deleting 8 (One child)**

   cout << endl << "The tree once 8 has been deleted." << endl;

   tree.deleting(8);

   tree.display(root, 1);

   cout << endl << endl << endl;

   system("pause");

**// Deleting 76 (No Children)**

   cout << endl << "The tree once 76 has been deleted." << endl;

   tree.deleting(76);

   tree.display(root, 1);

   cout << endl << endl << endl;

   system("pause");

**// Finding the number with the max key.**

   cout << endl;

   tree.findMax();

   cout << endl;

**// Adding 100 nodes to the tree**

   for (int i = 0; i < 101; ++i)

   {

      Node \*temp = new Node;

      temp->number = i;

      tree100.insert(root, temp, key);

      key++;

   }

   cout << endl << "100 nodes added." << endl;

   system("pause");

**// Deleting 100 nodes from the tree**

   for (int i = 0; i < 101; ++i)

   {

      tree.deleting(i);

      key--;

   }

   cout << endl << "100 nodes deleted." << endl;

   system("pause");

**// Adding 1000 nodes to the tree**

   for (int i = 0; i < 1001; ++i)

   {

      Node \*temp = new Node;

      temp->number = i;

      tree1000.insert(root, temp, key);

      key++;

   }

   cout << endl << "1000 nodes added." << endl;

   system("pause");

**// Deleting 1000 nodes from the tree**

   for (int i = 0; i < 1001; ++i)

   {

      tree.deleting(i);

      key--;

   }

   cout << endl << "1000 nodes deleted." << endl;

   system("pause");

**// Adding 10000 nodes to the tree**

   for (int i = 0; i < 10001; ++i)

   {

      Node \*temp = new Node;

      temp->number = i;

      tree10000.insert(root, temp, key);

      key++;

   }

   cout << endl << "10000 nodes added." << endl;

   system("pause");

**// Deleting 10000 nodes from the tree**

   for (int i = 0; i < 10001; ++i)

   {

      tree.deleting(i);

      key--;

   }

   cout << endl << "10000 nodes deleted." << endl;

   system("pause");

**// Adding 100000 nodes to the tree**

   for (int i = 0; i < 100001; ++i)

   {

      Node \*temp = new Node;

      temp->number = i;

      tree100000.insert(root, temp, key);

      key++;

   }

   cout << endl << "100000 nodes added." << endl;

   system("pause");

**// Deleting 100000 nodes from the tree**

   for (int i = 0; i < 100001; ++i)

   {

      tree.deleting(i);

      key--;

   }

   cout << endl << "100000 nodes deleted." << endl;

   system("pause");

   return 0;

}

**Analysis:**

From what I can tell, deleting nodes doesn’t seem to take very long, but adding nodes just goes up the more you add. Because of this, keeping your trees at 100 or less nodes would be ideal and possibly divide larger trees into smaller ones when possible.