**University of Ontario Institute of Technology**

**INFR 2820U: Algorithms & Data Structures**

**Assignment 3**

**Date Handed out: October 23, 2013**

**Due Date: Nov. 7, 2013 via Blackboard.**

**Instructions:** This is a group assignment. Submit a single file with your answers via Blackboard submission system. No other mean of submission (e.g., email, printouts, etc.) is accepted. Acceptable formats include Word and pdf. If you need to use ELC’s, you will need an ELC from each group member to get you a 24 hours extension.

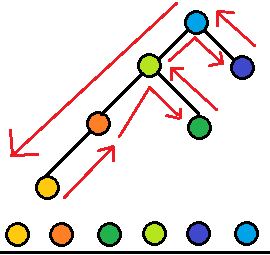
**The learning Outcome:** the objective of this assignment is to get some hands-on experience with Binary Search Trees, as well as with their run-time complexity.

1. In the lecture notes (Lecture\_07\_Trees.ppt) you were given the code for binary tree (binaryTreeType ) and binary search tree(bSearchTreeType) . Assuming that you are storing integers values in the tree nodes, and extend the binary search tree code to do the following:
2. Include the member function *display*(root) that displays the content of a BST.
3. Include the member function *singleParentCount*(root) that returns the number of nodes in the tree that have only one child.
4. Include the member function *lessThanValueCount(root, value)*  that returns the number of nodes in the tree that contains value less than the parameter *value.*
5. Use your code to find and plot the running time that it takes code to insert a large list of random items in to the BST. Try it for different list sizes and plot the results in graph. Repeat the experiment with SORTED lists now, and plot the results on the same graph. Comment on the performance of your algorithm with sorted unsorted lists.

**What should you submit?**

1. Only the source code. No Executable.
2. A page or two that include instructions on how to run your code and the answer to part B.

# Function Explanation

Display Function

***int display(nodeType<elemType> \*p)***

***{***

***if (root == NULL) {***

***cout << "TREE IS NOT INITIALIZED"<<endl;***

***return 0;***

***}***

***if (p->llink != NULL)***

***display(p->llink);***

***if (p->rlink != NULL)***

***display(p->rlink);***

***cout << p->info<<" ";***

***return 0;***

***}***

The display function first verifies that the list has been initialized (contains at least 1 element, the root). It then starts off by checking if the root contains a left element. The display function recursively calls itself, and performs the same operation. Eventually the function arrives at a node without a left element. At which point it verifies if the node as a right element, and continues until it has reach a leaf node. At which point it prints its value and (due to the recursion) returns to the previous value. It then checks if the node has a right element, continues to the right elements leaf node and prints its value and moves up until all elements to the left of the initial root are printed. The function then performs the same operation on the right elements of the root node. In essence, the function prints all leaf nodes to the left, then the right, then the root.

SingleParentCount Function

***int singleParentCount(nodeType<elemType> \*p)***

***{***

***int counter = 0;***

***if (root == NULL) {***

***cout << "TREE IS NOT INITIALIZED"<<endl;***

***return 0;***

***}***

***if (p->llink != NULL && p->rlink != NULL) {***

***counter += singleParentCount(p->llink);***

***counter += singleParentCount(p->rlink);***

***return counter;***

***}***

***if (p->llink != NULL)***

***counter += singleParentCount(p->llink);***

***if (p->rlink != NULL)***

***counter += singleParentCount(p->rlink);***

***if (p->rlink == NULL && p->llink == NULL)***

***return 0;***

***counter++;***

***return counter;***

***}***

The SingleParentCount function works by first ensuring the tree is initialized. It then works similarly to the display function, in that is checks all elements to the left of the root, then the elements to the right. First the node is checked to see if it possess links to a left and right element, if so, check the left side first then the right (reclusively), the returned values are added to the counter. After which, the value of the counter is returned and no further action is taken as a node with two elements cannot be a single parent.

The next steps are only executed for nodes which have only one element, IE single parents. The function is called recursively on for the single child of the parent to ensure all nodes are checked, the counter is then incremented and returned up the recursive stack.

lessThanValueCount function

***int lessThanValueCount(nodeType<elemType> \*p, int value) {***

***if (root == NULL) {***

***cout << "TREE IS NOT INITIALIZED"<<endl;***

***return 0;***

***}***

***int counter = 0;***

***if (p->llink != NULL)***

***counter += lessThanValueCount(p->llink, value);***

***if (p->rlink != NULL)***

***counter += lessThanValueCount(p->rlink, value);***

***if (p->info < value)***

***counter++;***

***return counter;***

***}***

The lessThanvalueCount function is simply a combination of the display function and the counter system of the previous function. The function starts off by checking to see if the tree has been initialized. Afterwards it is called recursively first to the left nodes, then to the right until it reaches a leaf node. At which point the value is compared to the supplied value, if node value is lower, the counter is incremented. Afterwards the counter value is returned recursively up the stack, and incrementing each time a lower value is found.

The LessThenValueCount function simply increments a counter each time the value of the node is less than the value supplied, instead of displaying the node value like the display function.

# Run time analysis

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Number or elements | 10 | 20 | 40 | 80 | 160 | 320 | 640 | 1280 | 2560 | 5120 | 10240 | 20480 | 40960 | 81920 |
| Unsorted runtime | 0ms | 0ms | 0ms | 0ms | 0ms | 0ms | 0ms | 1ms | 3ms | 7ms | 11ms | 26ms | 50ms | 98ms |
| Sorted runtime | 0ms | 0ms | 0ms | 0ms | 0ms | 0ms | 2ms | 5ms | 19ms | 74ms | 330ms | 1612ms | 6708ms | 30504ms |

Sorted list appear to increase the runtime at an exponential rate taking a fairly lengthy amount of time to populate a tree of 81920 elements when compared to adding unsorted elements. We are unsure as to the reasoning behind this, as we notice no change when increase the maximum random value to try and minimize any duplicates. This may be due to the fact that a sorted list will create a “linked list” structure instead of a binary tree. As every element added is greater than the previous.

CODE

Source.cpp

***# include "bSearchTreeType.h"***

***# include <sys/timeb.h>***

***using namespace std;***

***int getMilliCount(){***

***timeb tb;***

***ftime(&tb);***

***int nCount = tb.millitm + (tb.time & 0xfffff) \* 1000;***

***return nCount;***

***}***

***int getMilliSpan(int nTimeStart){***

***int nSpan = getMilliCount() - nTimeStart;***

***if (nSpan < 0)***

***nSpan += 0x100000 \* 1000;***

***return nSpan;***

***}***

***int random(int max, int min) { return rand() % max + min; }***

***int main()***

***{***

***bSearchTreeType<int> \*tree = new bSearchTreeType<int>;***

***int time = 0;***

***int value[14] = { 10, 20, 40, 80, 160, 320, 640, 1280, 2560, 5120, 10240, 20480, 40960, 81920 };***

***int j = 0;***

***cout << "Random numbers" << endl;***

***cout << "Value \t|\tTime" << endl***

***<< "========+===========" << endl;***

***for (j = 0; j < 14; j++) {***

***time = getMilliCount();***

***//random number in a sorted list***

***for (int i = 0; i < value[j]; i++)***

***tree->insert(random(32767, 0));***

***cout << value[j] << "\t|\t" << getMilliSpan(time)<< endl;***

***tree = new bSearchTreeType<int>;***

***}***

***cout <<endl<<endl<<endl<< "Sorted List of Numbers" << endl;***

***cout << "Value \t|\tTime" << endl***

***<< "========+===========" << endl;***

***for (j = 0; j < 14; j++) {***

***time = getMilliCount();***

***//random number in a sorted list***

***for (int i = 0; i < value[j]; i++)***

***tree->insert(i);***

***cout << value[j] << "\t|\t" << getMilliSpan(time) << endl;***

***tree = new bSearchTreeType<int>;***

***}***

***tree = new bSearchTreeType<int>;***

***cout << "Testing Tree" << endl;***

***cout << "tree->display result:";***

***tree->display(tree->root);***

***cout << endl << "Initilizing the tree" << endl;***

***tree->insert(60);***

***tree->insert(50);***

***tree->insert(70);***

***tree->insert(53);***

***tree->insert(30);***

***tree->insert(20);***

***tree->insert(1);***

***tree->insert(0);***

***cout << "tree->display result:";***

***tree->display(tree->root);***

***cout << endl<<"tree->singleParentCount(tree->root):"<< tree->singleParentCount(tree->root);***

***cout << endl << "tree->lessThenValueCount(tree->root,2):" << tree->lessThanValueCount(tree->root, 2) << endl;***

***system("pause");***

***return 0;***

***}***

***binaryTreeType.h***

***# include <iostream>***

***using namespace std;***

***template <class elemType>***

***struct nodeType***

***{***

***elemType info;***

***nodeType<elemType> \*llink;***

***nodeType<elemType> \*rlink;***

***};***

***template <class elemType>***

***class BinaryTreeType***

***{***

***public:***

***bool isEmpty() { return (root == NULL);}***

***int height(nodeType<elemType> \*p) {***

***if (p == NULL)***

***return 0;***

***else***

***return 1 + max(height(p->llink), height(p->rlink));***

***}***

***int nodeCount(nodeType<elemType> \*p) {***

***if (p == NULL)***

***return 0;***

***else***

***return 1 + nodeCount(p->llink) + nodeCount(p->rlink);***

***}***

***int leavesCount(nodeType<elemType> \*p) {***

***if (p == NULL)***

***return 0;***

***else if ((p->llink == NULL) && (p->rlink == NULL))***

***return 1;***

***else***

***return (leavesCount(p->llink) + leavesCount(p->rlink));***

***return 0;***

***}***

***void preorder(nodeType<elemType> \*p) {***

***if (p != NULL)***

***{***

***cout << p->info << " ";***

***preorder(p->llink);***

***preorder(p->rlink);***

***}***

***}***

***void inorder(nodeType<elemType> \*p) {***

***if (p != NULL)***

***{***

***inorder(p->llink);***

***cout << p->info << " ";***

***inorder(p->rlink);***

***}***

***}***

***void postorder(nodeType<elemType> \*p) {***

***if (p != NULL)***

***{***

***postorder(p->llink);***

***postorder(p->rlink);***

***cout << p->info << " ";***

***}***

***}***

***nodeType<elemType> \*root;***

***};***

***bSearchTreeType.h***

***# include "binaryTreeType.h"***

***# include <assert.h>***

***using namespace std;***

***template <class elemType>***

***class bSearchTreeType : public BinaryTreeType<elemType>***

***{***

***public:***

***bSearchTreeType() { root = NULL;}***

***bool search(const elemType& searchItem)***

***{***

***nodeType<elemType> \*current;***

***bool found = false;***

***if (root == NULL)***

***cerr << "Cannot search the empty tree." << endl;***

***else***

***{***

***current = root;***

***while (current != NULL && !found)***

***{***

***if (current->info == searchItem)***

***found = true;***

***else***

***if (current->info > searchItem)***

***current = current->llink;***

***else***

***current = current->rlink;***

***}***

***}***

***return found;***

***}***

***void insert(const elemType& insertItem) {***

***nodeType<elemType> \*current;***

***nodeType<elemType> \*trailCurrent = NULL;***

***nodeType<elemType> \*newNode;***

***newNode = new nodeType<elemType>;***

***assert(newNode != NULL);***

***newNode->info = insertItem;***

***newNode->llink = NULL;***

***newNode->rlink = NULL;***

***if (root == NULL)***

***root = newNode;***

***else***

***{***

***current = root;***

***while (current != NULL)***

***{***

***trailCurrent = current;***

***if (current->info == insertItem)***

***{***

***//cerr << "The item is already in the tree ";***

***//cerr << "duplicates are not allowed." << endl;***

***return;***

***}***

***else if (current->info > insertItem)***

***current = current->llink;***

***else***

***current = current->rlink;***

***}***

***if (trailCurrent->info > insertItem)***

***trailCurrent->llink = newNode;***

***else***

***trailCurrent->rlink = newNode;***

***}***

***}***

***int display(nodeType<elemType> \*p) {***

***if (root == NULL) {***

***cout << "TREE IS NOT INITIALIZED"<<endl;***

***return 0;***

***}***

***if (p->llink != NULL)***

***display(p->llink);***

***if (p->rlink != NULL)***

***display(p->rlink);***

***cout << p->info<<" ";***

***return 0;***

***}***

***int singleParentCount(nodeType<elemType> \*p) {***

***int counter = 0;***

***if (root == NULL) {***

***cout << "TREE IS NOT INITIALIZED"<<endl;***

***return 0;***

***}***

***if (p->llink != NULL && p->rlink != NULL) {***

***counter += singleParentCount(p->llink);***

***counter += singleParentCount(p->rlink);***

***return counter;***

***}***

***if (p->llink != NULL)***

***counter += singleParentCount(p->llink);***

***if (p->rlink != NULL)***

***counter += singleParentCount(p->rlink);***

***if (p->rlink == NULL && p->llink == NULL)***

***return 0;***

***counter++;***

***return counter;***

***}***

***int lessThanValueCount(nodeType<elemType> \*p, int value) {***

***if (root == NULL) {***

***cout << "TREE IS NOT INITIALIZED"<<endl;***

***return 0;***

***}***

***int counter = 0;***

***if (p->llink != NULL)***

***counter += lessThanValueCount(p->llink, value);***

***if (p->rlink != NULL)***

***counter += lessThanValueCount(p->rlink, value);***

***if (p->info < value)***

***counter++;***

***return counter;***

***}***

***};***