**CptS 223 – Advanced Data Structures in C++**

**Fall 2020**

***Take-Home Exam 1***

Friday, October 16, 2020

**Your Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**ID Number: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**TA’s Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

READ THE FOLLOWING INSTRUCTIONS:

This exam is take-home. You must work **individually** on this exam. You may use your book, notes, and online resources if necessary. Please either handwrite or type your answers into this document. Late exam solutions will **not** be accepted! **Please show all work!!!**

You must submit the exam through Git by Monday, October 19, midnight. On your local file system, and inside of your git repo for the class, create a new branch called Exam 1. Available to the branch create a .pdf called Exam1.pdf. Add, commit, and push Exam1.pdf using the technique discussed in class. Be sure to answer each question precisely. Do not provide superfluous details in your answers. *NOTE: you do not need to comment your code solutions.*

|  |  |  |
| --- | --- | --- |
| **Part** | **Points Possible** | **Points Earned** |
| I | 30 |  |
| II | 30 |  |
| III | 40 |  |
| BONUS | 5 |  |
| **Total** | **100** |  |

***Part I: Conceptual Questions (30 pts) – Short answer, Fill-in-the-blank, and Multiple-choice***

1. **(3 pts)** In your own words, describe the purpose behind performing algorithm analysis.

The idea behind algorithm analysis is to see how an algorithm works under different inputs to see its performance. It can reveal if the algorithm is better suited for a task or not if its time complexity is large or small and can show if memory issues can show up if the space complexity is large or small. So for instance, if you want to store data using a tree, an AVL tree would be better suited if you aren’t storing data that often since storing data and rotating nodes around can take a long time, but searching and finding the nodes is fast since the tree is balanced, so the furthest the search algorithm has to go is roughly log(base2)(n) or the height of the tree at worst. If you are storing data more than reading data, an RB tree might be more suited since there are fewer balancing restrictions so an insert algorithm takes less time, but reading can take a little longer as a consequence. So which algorithm or algorithms you want to use can depend on its purpose or the issue at hand which algorithm analysis helps with by showing which algorithms are faster, slower, take up more memory, or take up less memory in solving the problem.

1. **(3 pts)** What is the *Rule of the Big Five* in C++? Explain.

The Big Rule of 5 for C++ is that if you have 1 of these 5 things in a class, you should define all 5. Those being destructor, copy constructor, move constructor, copy assignment operator, and move assignment operator. This allows users not only to define how they want these to ask, but prevents the program to define them in their absence. The destructor frees up memory the object was taking up previously, copy constructor makes a new object based on an existing object, move constructor makes the object being made into the inputted object, including its address, while deleting the old one, the copy assignment operator copies data from one object to the other, and the move assignment operator, moves all data, including its pointer, from one object to the other and then deletes the object the data was moved from.

1. **(3 pts)** What are *Makefiles* and why do we use them for programs in this class? Explain.

Makefiles are files that can be ran in order to compile the entire project and run it. It lets us simply run the Makefile after a change in code rather than having to recompile the entire project and run it again and makes the code a little easier to understand.

1. **(4 pts)** What do the following Linux commands do?
   1. **(1 pt)** man

Displays the manual of a command

* 1. **(1 pt)** mv

Moves a file or directory from it’s current location to different location

* 1. **(1 pt)** mkdir

Makes a new code directory

* 1. **(1 pt)** rm

Removes a file or directory from the system

1. **(3 pts)** What is the general principle behind *software testing*? Explain.

Software testing lets us test how an algorithm or how code would work in different scenarios without implementing it yet. The idea is that through testing, we can check all possible scenarios of a certain piece of code without running the risk of damaging something and see if the code really works for all cases. And if there is an error, we can go in and fix it before we implement it, which can also give us a better idea of how the code works.

1. **(3 pts)** Given the following function, what is the worst-case Big-O time complexity?

\_\_\_\_\_\_\_O( n^2 )\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

// Prints all subarrays in arr[0..n-1]

void subArray(int arr[], int n)

{

    // Pick starting point

    for (int i=0; i <n; i++)

    {

        // Pick ending point

        for (int j=i; j<n; j++)

        {

            // Print subarray between current starting

            // and ending points

            for (int k=i; k<=j; k++)

{

                cout << arr[k] << " ";

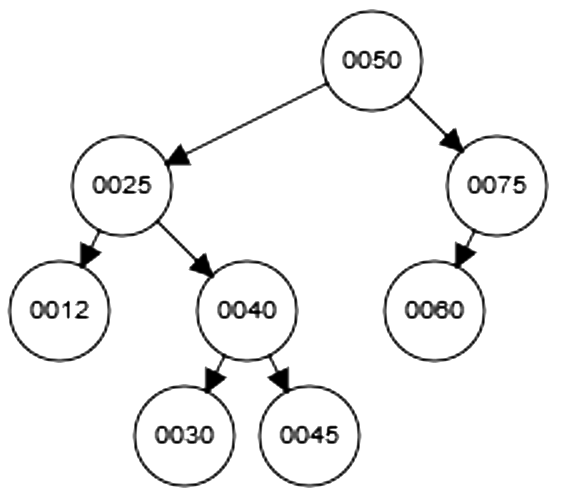
}

            cout << endl;

        }

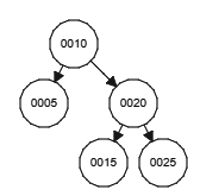
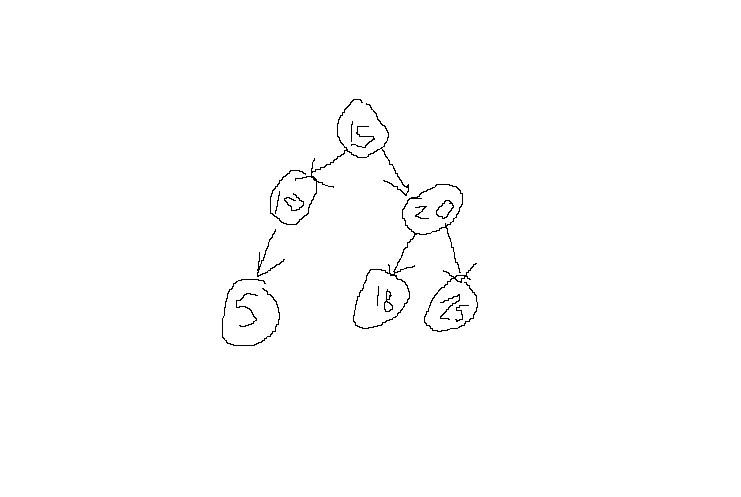
    }

1. **(5 pts)** Fill-in-the-blank - Using the following BST, answer the questions below:

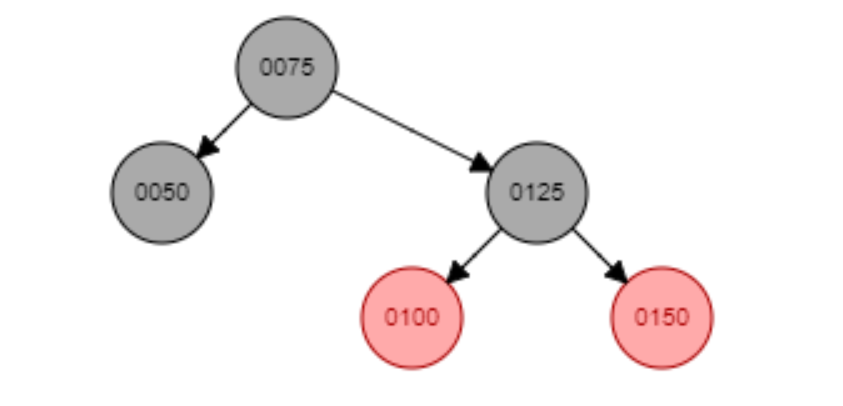
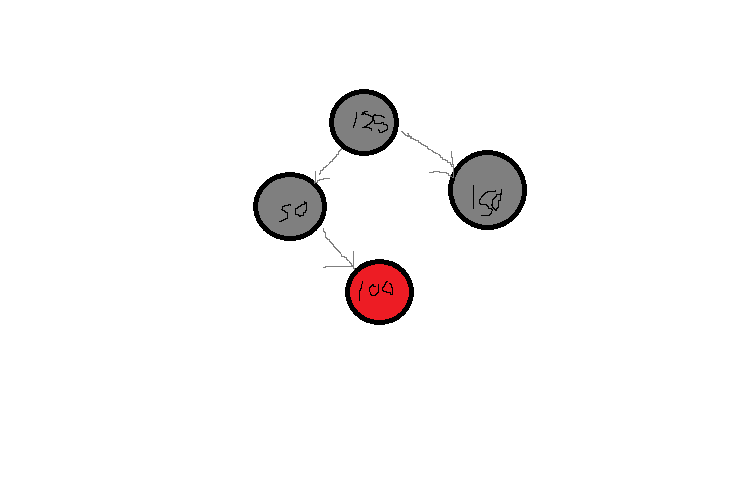


* 1. **(1 pt)** How many *children* does the node containing the number 0040 have? \_\_\_\_2\_\_\_
  2. **(1 pt)** What is the *height* of the tree? \_\_\_\_3\_\_\_\_
  3. **(1 pt)** What is the *depth* of the node with value 0040? \_\_\_\_2\_\_\_\_
  4. **(1 pt)** At what *level* is the root node? \_\_\_\_1\_\_\_\_
  5. **(1 pt)** What is the *height* of the node with value 0040? \_\_\_\_1\_\_\_\_

1. **(3 pts)** Insert the value 0018 into the following *AVL* tree; draw the resulting tree:

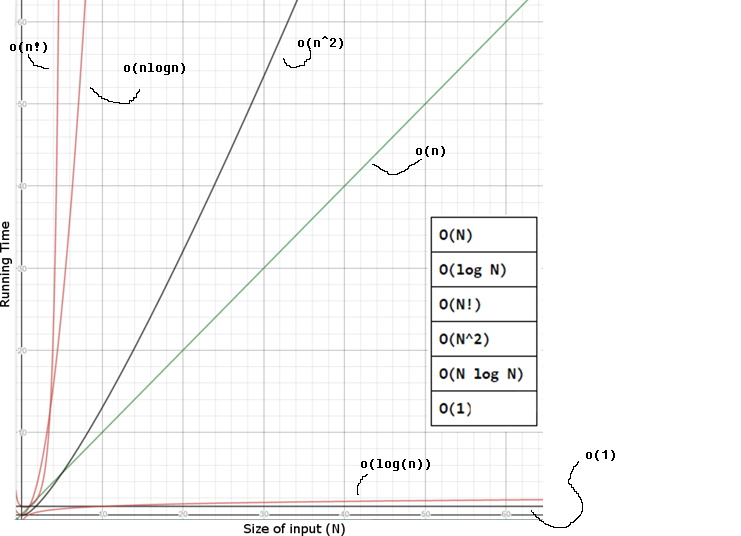


1. **(3 pts)** Delete the value 0075 from the following *red-black* tree; draw the resulting tree:

******

***Part II: Algorithm Analysis and Big-O Notation (30 pts)***

1. **(4 pts)** The graph below shows different function growth complexities (Big-O orders). Label each line with the appropriate Big-O growth function from the table on the chart.

****

11. **(6 pts)** Given the following function answer the questions below.

|  |
| --- |
| **void someFunction( vector<int> & list ) {**  **int clean\_pass = false;  for( int i = 1; !clean\_pass && i < list.size(); i++ ) {  clean\_pass = true;  for(int j = 0; j < (list.size() - i); j++)  if( list[j] > list[j+1] ) {   int temp = list[j];  list[j] = list[j+1];  list[j+1] = temp;  clean\_pass = false;  }  }** |

* 1. **(2 pts)** What does the function do?

This function looks at each element in the list and the element after it and if the element after the current is larger than the current, it swaps it. It continues swapping until either the algorithm has traversed the whole list, or until the algorithm makes a clear pass through the list without having to swap items.

* 1. **(2 pts)** What is the worst-case Big-O *time* complexity for algorithm?

The worst case for time complexity is when the algorithm has to swap n elements to be in order. So, something like 5 4 3 2 1. It has to look at an element at most n times, and then n-1, n-2, n-3, and so on for swapping which is equivalent to n^2, so I think the worst case is O(n^2).

* 1. **(2 pts)** What is the Big-O *space* complexity for algorithm?

Space doesn’t change as not new memory is allocated dynamically. The only memory allocated is the instance of the function and its variables. So, space complexity is O(1).

12**. (4 pts)** Given a *binary* tree, NOT *search* tree, what is the worst-case Big-O *time* complexity for finding the *max* value in the tree? List any assumptions you make.

The worst case for a “find max” binary tree is that it’s skewed all to the right so that the algorithm has to see n elements. In that case, to find the max, the algorithm has to look at root->right recursively to get to the max so the worst-case would-be O(n).

13. **(4 pts)** Given a red-black tree, what is the worst-case Big-O *time* complexity for finding the *min* value in the tree? List any assumptions you make

I think the worst case for the min value in a RB tree would be O(log(base2)(n)). With an RB tree, the black height always has to be balanced, so the min value will roughly take log(base2)(n) nodes to traverse to get to where n is the total number of black nodes in the tree. The min node in worst case would be the black height away from the root + 1 since the furthest away a node can be is if it’s red.

It would also roughly be times log(base2)(n) where n is roughly the number of red nodes in the tree. So the total would be 2log(base2)(n) + 1 so O(log(base2)(n)) for the time complexity.

14. **(4 pts)** Given an *AVL* tree, what is the worst-case Big-O *time* complexity for *inserting* an item in to the tree? List any assumptions you make.

Because an AVL tree is strictly balanced, the insert algorithm has to look at most log(base2)(n) nodes where n is the total number of nodes in the tree to find where it needs to insert. The insertions after that involve at worst rotating nodes in a predetermined way depending on the structure of the tree, which in worst case involves rotating each element the tree looked at, but not all of them, so roughly log(base2)(n) rotation functions, a little more since some might have to be rotated twice. So at worst, the algorithm has O(log(base2)(n)).

15. **(4 pts)** Given a *BST*, not necessarily an AVL tree, what is the worst-case Big-O *time* complexity for determiningthe *height* of a node in the tree? List any assumptions you make.

The worst case for 1 is looking at the height of the root since that will have the longest path to a leaf in the tree. The second is assuming that the tree is skewed entirely 1 way. Then this is the worst case and the tree has to look at n elements. So the algorithm for finding the height of the tree in worst case is O(n).

16. **(4 pts)** We need to *eliminate* all *duplicates* from a string. What is the worst-case Big-O *time* complexity of your algorithm? List any assumptions you make. Also, justify your answer with a concise explanation.

I think the worst case for an eliminate duplicate algorithm for a string would be O(n). In my algorithm, the worst case would be all elements in the string were different. To see if there were duplicates in the string, the algorithm would have to compare each element to each other element in the string after it. So, for something like “sdfg”, s would be compared to dfg, d to fg, f to g, and g wouldn’t be compared. This results in the first element being compared to n-1 elements, the second to n-2 elements, and so on until no comparisons are made. So we have n-1 + n-2 + n-3 + n-4 +… + n-n comparisons made which when simplified is equal to n-1 + … + 1 so n times a constant. This means that the time complexity is O(cn) where c is a constant, or O(n).

***Part III: C++ Programming Questions (40 pts)***

17. **(15 pts)** You are given the following class definition for a BSTNode:

class BSTNode {

private:

int data, height;

BSTNode \* left, \* right;

public:

BSTNode() : left(nullptr), right(nullptr) { }

};

Write a function isBalancedBST() that accepts a pointer to a BSTNode and determines if a BST is balanced. **After you write your function, state its *time* complexity using Big-O notation.** Make sure to include a function header and to determine an appropriate return type for your function. You may assume appropriate getters and setters are defined for the BSTNode class. List any other assumptions that youmake.

//The worst-case scenario is if it checks each if the tree is unbalanced, but the right and left nodes are defined (not NULL). Assuming the heights are accurate to each node, the worst case would be O(1) since there are a constant number of checks depending on the situation that doesn’t depend on how many nodes are in the tree besides the at most 2 it’s looking at.

//Function returns true if it is balanced and false if it isn’t since the function only wants to see if it is balanced or not, not the balance factor.

bool isBalancedBST(BSTNode\* root){

if(root == NULL){

return true;

}

else if(root->get\_left() == NULL && root->get\_right() == NULL){

return true;

}

else if(root->get\_right() == NULL){

if(get\_height(root->get\_left()) > 1 || get\_height(root->get\_left()) < -1){

return false;

}

else{

return true;

}

}

else if(root->get\_left() == NULL){

if(get\_height(root->get\_right()) > 1 || get\_height(root->get\_right()) < -1){

return false;

}

else{

return true;

}

}

//Scenario where right and left aren’t null

int left\_height = get\_height(root->get\_left());

int right\_height = get\_height(root->get\_right());

int balance = left\_height – right\_height;

if(balance <= 1 && balance >= -1){

return true;

}

else{

return false;

}

}

18. **(25 pts) Parts a. and b. are related.**

a. **(10 pts)** You are given the following functions for an *AVL* tree, where pTree points to an AVLNode where an *imbalance* is found, and heightDiff() returns the difference between the heights of an AVLNode’s left and right children:

void AVLTree::leftRotate(AVLNode \*& pTree);

void AVLTree::rightRotate(AVLNode \*& pTree);

void AVLTree::leftRightRotate(AVLNode \*& pTree);

void AVLTree::rightLeftRotate(AVLNode \*& pTree);

int AVLNode::heightDiff();

First, write the function definition for the rightRotate() function given above. You are given the following definition for the AVLNode class:

class AVLNode {

private:

int data, height;

AVLNode \* left, \* right;

public:

AVLNode() : left(nullptr), right(nullptr) { }

};

You may assume appropriate getters and setters are defined. List any other assumptions that you make.

void AVLTree::rightRotate(AVLNode \*& pTree){

AVLNode\* temp = pTree->get\_left();

AVLNode\* temp2 = temp->get\_right();

//Makes sure data to the right isn’t lost by moving it to pTree

ptree->set\_left(temp2);

//Rotate

temp2 = pTree;

pTree = temp;

}

b. **(15 pts)** Next, write a recursive function to insert a node into an AVL tree. Your insert function MUST ensure the tree is balanced after performing the insertion. You may assume the data values inserted into each tree are unique. Also, please use any functions described in part a. List any other assumptions that you make. Use the following function header:

//Loosely based off of my memory of my own AVL tree insert, but without a return type

void AVLTree::insert(AVLNode \*& pTree, int newData){

AVLNode\* new\_node = new AVLNode();

new\_node->set\_data(newData);

AVLNode\* pTree\_left = nullptr;

AVLNode\* pTree\_right = nullptr;

int pTree\_data = 0;

int pTree\_left\_data = 0;

int pTree\_right\_data = 0;

bool left\_initialized = false;

bool right\_initialized = false;

//Makes sure there's no access violations

if (pTree != nullptr) {

pTree\_data = pTree->get\_data();

pTree\_left = pTree->get\_left();

pTree\_right = pTree->get\_right();

if (pTree\_left != nullptr) {

pTree\_left\_data = pTree\_left->get\_data();

left\_initialized = true;

}

if (pTree\_right != nullptr) {

pTree\_right\_data = pTree\_right->get\_data();

right\_initialized = true;

}

}

if (pTree == NULL) {

return;

}

int balance = 0;

//Do recursive insertion

if (pTree->get\_data() > newData) {

insert(pTree\_left, newData);

pTree->set\_left(pTree\_left);

}

else if(pTree->get\_data() <= newData){

insert(pTree\_right, newData)

pTree->set\_right(pTree\_right);

}

//Continue to next page

if (!left\_initialized && !right\_initialized) {

balance\_factor = 0;

}

else if (!left\_initialized) {

balance\_factor = pTree\_right->get\_height() \* -1;

}

else if (!right\_initialized) {

balance\_factor = pTree\_left->get\_height();

}

else {

balance\_factor = pTree\_left->get\_height()- pTree\_right->get\_height(); }

//Left then Left

if (balance\_factor > 1 && newData < pTree\_left\_data) {

rightRotate(pTree);

}

//Right then Right

if (balance\_factor < -1 && newData < pTree\_right\_data) {

leftRotate(pTree);

}

//Left then Right

if (balance\_factor > 1 && newData > pTree\_left\_data){

leftRotate(pTree\_left);

pTree->set\_left(pTree\_left));

rightRotate(pTree);

}

//Right Left

if (balance\_factor < -1 && newData > pTree\_right\_data){

rightRotate(pTree\_right);

pTree->set\_right(pTree\_right);

leftRotate(pTree);

}

}

**BONUS (5 pts):** Discuss tradeoffs between using the C++ STL *list* and *vector*.

One of the big differences for vectors and lists is that a list allocates memory in memory dynamically, but not necessarily together while vectors allocate memory next to each other, like an array. Vectors are faster than Lists for finding elements as if you know the index of an element in a vector, you can look at that place in memory for it, but for lists, you have to traverse through each node through their pointers until you find it. For inserting nodes into them, for lists you have to go to the end of the list to insert or to a point and move some pointers around to fit it in, but for vectors, inserting inside the vector, not at the end, means you have to adjust the elements after that to account for the new element. The same is true for both with the delete function. So vectors are much more useful for if you want a structure that doesn’t change its data a lot and you need fast search times, whereas lists are better for if you are inserting and deleting a lot in the data structure.