COSC-160-01
Fall 2018
Project 2 Description

BST Efficiency

1. Summary

   Earlier this semester, Project 1 involved two different linear data structures that are designed for efficient data retrieval: orderedList and MTFlist. We have now turned our attention to non-linear data structures—specifically trees—but we maintain our focus on efficiency. In Project 2, you will design and implement two tree data structures: **AVLtree** and **splayTree**. As in Project 1, you will empirically compare the efficiency of the two structures. The details of implementation are largely at your discretion. Major factors to consider for the design should be theoretical efficiency (both time and space), practical efficiency, and appropriate memory management. Although some related concepts and tips are provided below (and have been provided in class), the determination of how to make the structure efficient is largely your charge.

2. Programming Languages

   I encourage you to submit your projects using C++. Take note that there are many versions of C++; you must use the version that is currently running on the course server. With prior approval from me, you may choose a different programming language (I have already approved Java for students). Keep in mind that one of the main goals of our class projects is for you to learn how to construct various data structures from the most elemental programming constructs. Thus you will not receive credit when using any pre-existing structures from programming libraries (or code that has been created or designed by others). For example in C++ you *cannot* use the pre-existing vectors, stacks, lists, etc. For some programming languages, complex data structures (non-elemental constructs) are "built-into" the language. You cannot use any built-in structure. For example, in Python, both list and "array" structures are fairly complex and not elemental programming constructs, e.g., they can change size dynamically. For this reason and others, I am unlikely to approve the use of Python for projects. If you hypothetically did use Python, however, you would not receive credit when using those built-in structures. If you have any questions as to what structures are permitted (and which are not permitted), given your language of choice, please ask me.

3. Planning and Design

   Before implementation, you should plan and design your project using standard approaches, e.g. UML class diagrams, flow diagrams, etc. If you have questions pertaining to your project, I will first ask to see your designs. I will not look at your code without first viewing your design documents. You will be faced with many design decisions during this project. It is best to spend the requisite time during the design stages to assure an appropriate and efficient implementation is built. Consider your options, perform a theoretical complexity analysis of the different options, and base your decision on the results of your analysis.

4. Input Requirements

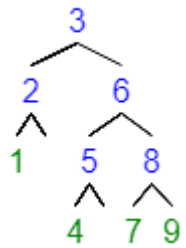
   The program will take one command line argument, which will be a text file (assumed to be located in the same directory as the executable). The program will load and store a collection of integers into the tree structures. It will then perform a sequence of removals from the tree.

   The first line of the input file will contain all the integer numbers to store, separated by spaces. Using this input, the program should initialize the given tree structure. This line will be followed by a new line character. The second line—which serves to separate the initialization values from the removal values—has just a single symbol, dollar sign ($), followed by a new line character. The third line contains the sequence of removal values. For each of these integers, the program should remove the value from the tree (if the tree, in fact, contains the value).

5. Structural and Operational Requirements
   A. A BST (of integers) structure
   B. An AVLtree (of integers) structure

   The following is required for both structures:

   - Valid state of the tree is always maintained
   - Includes the following key operations: insert, remove and display (print to console)
     - To display the tree, you will need to provide a print method or overload the "<<" operator. Your program needs to print a display that indicates the structure of the current tree. It can then be rendered using the tool found at the Syntax Tree Generator website (http://mshang.ca/syntree/). To construct such a print statement, perform a Depth First Traversal and 1.) print an open bracket when traversing down 1 depth to a child node, 2.) print the key for that node (or nothing, if null) and 3.) print a close bracket for each depth retreated. For leaf nodes, the program should not print its null children (it should only print a null node if it's a null child of a parent with one non-null child.
       - Example: [3 [2 [1][]] [6 [5 [4][]] [8 [7] [9]]]]



   - Appropriately allocate/deallocate memory

   Notes: To earn full marks it is expected that you will implement the structures very efficiently (in space and time). If you are faced with a space / time tradeoff, you will opt to improve time complexity (if the cost of space is relatively minor).

6. Output Requirements

   The following steps should be executed (efficiently) by the main method:

   A. Store the data items in BST, maintaining a running count of the total number of comparisons.
   B. Output the structure of the initialized tree, along with the total number of comparisons so far.
   C. Complete the sequence of removals for BST, outputting the new structure after each removal and updating the running count of the total number of comparisons for BST.
   D. Store the data items in AVLtree (i.e., the same values it used to initialize the BST), maintaining a new running count of the total number of comparisons for this new tree structure.
   E. Output the structure of the initialized tree, along with its total number of comparisons so far.
   F. Complete the sequence of removals for AVLtree, outputting the structure after each removal and updating the running count of the total number of comparisons for AVLtree.
   G. Print a final output that includes the respective total number of comparisons for each tree structure, and declare which is more efficient.

7. Submission and Compilation Requirements

   As with all project submissions for COSC-160, you must fill out a cover letter. Your cover letter for Project 2 should explain what your program does and describe the various components. It should also include a brief explanation for how you optimized the efficiency. Finally, it should include a brief report that you produce after running your program on test inputs. It should show the number of comparisons each tree structure executed when processing the same input data. Budget your time well. Include significant time for design / planning and testing / debugging. Please submit early and often (version control)! Your last submission will be graded.

   You must include directions on how to compile and run your program (either in your cover letter or as a separate README file). For this project, *we will recommend but not require* that you submit a makefile. If the program does not compile (based on whatever directions you supply) or the program does not run, the submission will not be graded.

8. Testing and Debugging (Not Submitted)

   You may wish to construct an interactive interface to test the functionality of your structure at intermediate stages of development. This would likely be most efficient with an interactive interface that allowed you to interactively test various functionalities of your structure given different inputs. If you do implement a testing interface, please be sure to comment it (so that it does NOT execute) before submission. I also strongly encourage you to construct and test many input files to test the functionality of your implementation on varying inputs.

9. Version Control (Not submitted, but encouraged)

   I strongly recommend that you back-up your work periodically throughout the development process. This can mitigate a disaster scenario where you might accidentally delete your program files. I also recommend employing a version control strategy which records your development at different stages (versions). If you have time, I encourage you to investigate GitHub to facilitate version control. Otherwise you can make use of a more simplistic naming scheme: each time you save a file, change the filename to indicate a version: filename v1.cpp, filename v2.cpp, ...

10. Rubric

List of Requirements	Percentage
BST Insert	0.15
BST Remove	0.15
BST Display	0.10
AVLtree Insert	0.15
AVLtree Remove	0.15
AVLtree Display	0.10
Efficient Main	0.10
Cover Letter	0.10
TOTAL	**1.00**