COSC-160
Spring 2020
Project 2b Description

BST Efficiency

1. Summary

   Project 2b extends the work you started in 2a. Having already implemented a (non-balancing) **BST** structure, you will now design and implement a **Splay Tree** structure. As in Project 1a, you will then empirically compare the efficiency of the two structures (using the same two metrics as before: time and Traversal Count). 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 structures efficient is largely your charge.

2. Submission and Compilation Requirements **[SAME AS IN PROJECT 2a]**

   To standardize submissions, you will submit a makefile, which will contain the necessary compilation commands for your code. The target executable will be named **p2**.

   Thus, the graders should be able to compile your program by accessing it on the course server and writing the following command:

   make p2

   Check to make sure that it works on the class server. If the program does not compile (using the above make command) or the program does not run, the submission will not be accepted. 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.

3. Input Requirements

   After successful compilation, the following command will run your program:

   ./p2 [inputFilename]

   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). For both **BST** and **Splay Tree**, the program will first load and store a collection of integers into the tree structure. It will next process another sequence of integers and search for each integer in order (some integers may not exist in the tree). It will finally process a third sequence of integers and remove it if it exists in the tree.

   [BELOW IS THE DESCRIPTION OF THE INPUT FILE. IT IS THE SAME AS IN PROJECT 2a]

   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 tree structure. This line will be followed by a new line character. The second line—which serves to separate the initialization values from the search values—has just a single symbol, dollar sign ($), followed by a new line character. The third line contains the sequence of search values. For each of these integers, the program should search for the value from the tree (if the tree, in fact, contains the value). This line will be followed by a new line character. The fourth line—which serves to separate the search values from the removal values—has just a single symbol, dollar sign ($), followed by a new line character. The fifth line contains the sequence of search values. For each of these integers, the program should remove the value from the tree (if the tree, in fact, contains the value).

4. Output Requirements

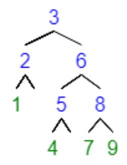
   Your program will evaluate the efficiency of the two tree structures based on the time needed for each structure and the its Traversal Count. For this project, you should increment the given tree's Traversal Count each time your program accesses a node during the additions, searches and removals. *For the **Splay Tree**, you should also increment the Traversal Count once for each splay operation* (e.g., if a given node starts at depth = 6, it will require 3 splay operations to get to the root level, so the Traversal Count should be incremented 3 times).

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

   A. Store the data items in **BST**, updating Traversal Count.
   B. Output the structure of the initialized tree.
   C. Complete the sequence of searches for **BST**, updating the Traversal Count
   D. Complete the sequence of removals for **BST**, updating the Traversal Count.
   E. Output the final structure of the tree (after all removals).
   F. Repeat Steps A – E for **Splay Tree** (with its own Traversal Count)
   G. Output a message that tells the user the Traversal Count and elapsed time (in nanoseconds) for each of the two tree structures. Declare which was more efficient with regard to Traversal Count, and which was more efficient with regard to elapsed time.

5. Structural and Operational Requirements

   A. **[SAME AS IN PROJECT 2A]** A **BST** (of integers) structure. The following is required:
   - Valid state of the tree is always maintained
   - Includes the following key operations: insert, remove, search and display (print to console)
     - To display the tree, you will need to provide a print method that can indicate 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

B. A **Splay Tree** (of integers) structure. This structure should contain the same operations as BST. Note that the insert, remove and search operations should call appropriate splay operations, as described in lecture and the text book.

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. Rubric

COSC-160 Rubric (see syllabus for more details)

Points	Explanation
3	Correct, efficient and elegant code
2	Correct, but lacking in efficiency/elegance
1	Incorrect
0	Project not submitted or missing components

List of Requirements
Makefile
Compiles/runs as specified
BST structure
Splay Tree structure
Correct output format
Traversal Counts in range
Time efficiency

Your program will be tested with files in the format specified. Your program will be evaluated based on the Traversal Count and on the elapsed time (as measured in nanoseconds) for each of the two structures. Before your final submission, you should test your program on the provided test input files. Below are the Traversal Count benchmarks for both **BST** and **Splay Tree**.

File Name	**BST**	**Splay Tree**
p2_test0.txt	190 - 200	~100
p2_test1.txt	7,300 – 7,350	~800 - 1000
p2_test2.txt	49,500 – 49,700	~3500 - 5000
p2_test3.txt	49,200 – 49,400	~3500 - 5000
p2_test4.txt	45,200 – 45,400	~3000 - 4000

The following sections are boilerplate for this course, but contain valuable information:

7. Programming Languages

   You should submit your projects using C++ (unless you have spoken with me and I have approved a different programming language for your project). Take note that there are many versions of C++; you must use the version that is currently running on the course server. 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. If you have any questions as to what structures are permitted (and which are not permitted), given your language of choice, please ask me.

8. 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.

9. 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.

10. 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, ...