CS2710 - Programming and Data Structures Lab

Lab 11, Homework Assignment (graded)

Out: Oct 18, 2024 (**Due: 21st Oct, 2024, 23:59 hrs**)

Instructions

- You are expected to solve ALL the problems as a **homework** assignment, using a system (computer, C++ language, and g++ compiler) that is compatible with the DCF lab system (since we will be using the lab system to evaluate your code). (Bonus problems can fetch 5% capped bonus, and Optional problems are just for fun and won't be graded.)
- You should submit your code to the course **moodle** on time (due date and time mentioned above; moodle submission link to be made available soon).
- You must strictly adhere to the following naming convention for your .cpp files and the single .zip file submission. For example, for a Lab Session 11 consisting of 2 questions, a student with roll number CS23B000 should
  - Name their .cpp files as **CS23B000_LAB11_Q1.cpp** and **CS23B000_LAB11_Q2.cpp**
  - Put both these .cpp files in a directory named **CS23B000_LAB11**
  - Zip this directory into a file named **CS23B000_LAB11.zip**
  - Submit only this single .zip file to moodle.
- If you need assistance, ask your TA, not your classmate.
- You have to write code on your own – you can refer to codes provided in course moodle and refer to slides or Weiss book; but otherwise, you cannot use any other sources such as the internet, or take help of your friends or fellow students. Plagiarism checks will be strictly implemented both among all students and any source codes available in the internet.
- We will make available in course Moodle the public test cases and the evaluation script, which you can use to test your programs.
- An AVL Tree is the primary data structure you will implement for the questions in this assignment. You may use string and vector data types for bookkeeping or as helper tools. **As the questions make it clear, recursion is NOT allowed in this assignment. You have to implement all questions in an iterative fashion.** Please clarify with the TAs if you have any questions about which data structures to use or avoid.

---

1. [AVL INSERT - HANDS TIED! (I.E., WITHOUT RECURSION)] You are tasked with implementing an AVL Tree (a height-balanced BST) as a C++ class using iterative (non-recursive) member functions. Your task is to maintain the AVL balance at each node during insertion.

   The Operation of insertion is as follows:
   Insert(x): Insert a key x into the AVL.

   If it helps, you can start with an iterative implementation of BST insert, and then modify it to get an implementation of the AVL tree. If you need to store any additional information in a Tree Node (beyond data/key, left pointer, right pointer, and height fields), you are allowed to do so.

   Input Format:

   - Each operation will be an insertion **Insert X** where **X** is an integer value.
   - End line will be **PostOrder**
   - There might be insertion of duplicate elements into the AVL; these operations should be ignored without any error messages.

   Output Format:

   - Print one line representing **POST ORDER** traversal of the AVL. Post order traversal should also be implemented iteratively (using std::stack if needed).

   Constraints:

   - Number of operations $\leq 10^5$.
   - Value of nodes $\leq 10^5$.

   Example:
   Input1:
   Insert 5
   Insert 7
   Insert 4
   Insert 10
   Insert 20
   PostOrder

   Output1:
   4 7 20 10 5

2. [WATCH YOUR CLOCKS!] Your task is to empirically compare the running times of the following operations on the AVL tree implementations (recursive implementation in WeissBook provided in class, as well as your iterative implementation for Q1 above), and Binary Search Tree (BST) implementations (specifically implementation in WeissBook provided in class):

   - **Insert(x):** Insert a key x into both the AVL tree and the BST.
   - **Search(x):** Search for a node with key x in both the AVL tree and the BST. Returns a pointer to the Node containing x or nullptr if key not found.

   Execute each operation type (insertion and search) on the three implementations. Measure and calculate the average time taken (in nanoseconds) for each operation type, and use plots to summarize your results. Code snippets from WeissBook in this URL (`https://users.cs.fiu.edu/~weiss/dsaa_c++4/code/RemoveEveryOtherItem.cpp`) may be used to measure time.

   Input Format:

   - First line contain two integer n and m, where n is total number of insert operations and m is total number of search operations.
   - Above line is followed by n lines of **Insert X** and m lines of **Search X** operations.
   - There might be insertion of duplicate elements in the AVL; this operation should be ignored without any error messages.

   Output Format:

   - **First** Line as the average time of AVL Tree (your iterative implementation), AVL Tree (WeissBook recursive implementation), and BST (WeissBook implementation) in insertion of all values.
   - **Second** line as the average time of AVL Tree (your iterative implementation), AVL Tree (WeissBook recursive implementation), and BST (WeissBook implementation) in searching all values.

   In addition, use the above outputs with any software/language to generate plots (e.g., barplots, scatterplots, or whatever is relevant) to show running time differences between (i) AVL tree recursive vs. iterative implementations, and (ii) AVL tree vs. BST implementations (to show tradeoff between insertion and search times in AVL tree vs. BST). When doing these comparisons, consider different types of insertion order (e.g., sorted, reverse sorted, and random insertion order of keys).

   Constraints:

   - $1 \leq m, n \leq 10^5$.
   - value of nodes $\leq 10^5$.

   Input:
   7 3
   Insert 5
   Insert 7
   Insert 4
   Insert 10
   Insert 20
   Insert 3
   Insert 25
   Search 4
   Search 20
   Search 25

Sample Output:
9076 8010 5789
3890 4800 8967

(In addition, use above outputs across several test cases to generate your plots summarizing running time comparisons as described above. These plots may be generated using other software, but running time measurements are done in C++. All your plots should be assembled into a single pdf file and uploaded to moodle under the name **[rollno]_LAB11_Q2.pdf**.)

3. [OPTIONAL: EXTENSION TO AVL DELETE - HANDS TIED AGAIN!] To question 1 above, add a non-recursive (iterative) implementation of deletion operation in the AVL Tree. So your extended code should support both these operations:

   Insert(x): Insert a key x into the AVL.
   Delete(x): Delete the node with key x from the AVL tree. Use the same logic as seen in class, including use of in-order successor to replace the deleted element if the node containing x has two (non-null) children.

   Input Format:

   - Each operation will either be an insertion **Insert X** or a deletion **Delete X**, where **X** is an integer value.
   - End line will be **PostOrder**
   - There might be insertion of duplicate elements to or removal of non-existent elements from the AVL; these operations should be ignored without any error messages.

   Output Format:

   - Print one line representing **POST ORDER** traversal of the AVL. Post order traversal should also be implemented iteratively (using std::stack if needed).

   Constraints:

   - Number of operations $\leq 10^5$.
   - Value of nodes $\leq 10^5$.

   Example:
   Input1:
   Insert 5
   Insert 7
   Insert 4
   Insert 10
   Insert 20
   Delete 10
   Insert 8
   Delete 7
   PostOrder

   Output1:
   4 20 8 5