CS 2413 – Data Structures – Spring 2022 – Project 3
Due 11:59 PM, May 4, 2022
Dr Katia Papakonstantinopoulou

DESCRIPTION OF THE PROJECT

In this project we build and query a two-dimensional binary search tree. Then we sort the preordered sequence of nodes using a couple of sorting algorithms with a bound on the number of allowed comparisons, and we compare the quality of the results.

A. BUILDING AND QUERYING A 2-D BINARY SEARCH TREE

Let $S = \{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ be a set of coordinate points in a plane. Since these points belong in a 2-D space, we need a 2-D data structure to store them. One simple idea is the matrix, but we prefer a structure in which storing or retrieving more than one points with the same x-value is simple and fast. We thus construct a 2-D binary search tree, in which the coordinates (x,y) are essentially implementing two connected binary search trees.

The x-values are stored in a binary search tree called the xTree. Each node of the xTree, apart from the fields required for a regular binary search tree node (as you already know it), has also a pointer to another binary search tree (called yTree) that stores the corresponding y-values. These trees are connected as described below. Assume that $(x_i, y_{i1}), (x_i, y_{i2}), \ldots, (x_i, y_{ik})$ is a set of coordinate points. There will be a node in the xTree that stores x_i, and the yTree (which is a binary search tree as well) that is connected to the x_i-node will store the values $y_{i1}, y_{i2}, \ldots, y_{ik}$. The purpose of this project is to practice in working with this two-dimensional binary search tree.

As part of the processing, you are required to build two classes xNode and yNode with the appropriate fields and methods. The xNode and yNode are obtained by appropriately templating the binary search tree class. The operations that will be performed on this structure are: **insert** (I), **remove** (R), **find** (F), **range search** (S), **balance y-tree** (Y), **balance x-tree** (X), and **print** (P).

In the input file you are given an unknown number of commands that correspond to the operations above. Let's see an example of how each one of these commands looks like and to which exact operation it corresponds:

a) Operation: insert (I) – Command: **I 3 7 52** – meaning: insert in the coordinate (3, 7) a node with identifier 52.
b) Operation: remove (R) – Command: **R 5 2** – meaning: remove the node stored in the coordinate (5,2) and print its identifier.
c) Operation: find (F) – Command: **F 4 3** – meaning: find the node stored in the coordinate (4, 3) and prints its identifier.

d) Operation: range search (S) – Command: **S 4 7 7 15** – meaning: print the identifiers of the nodes stored in coordinates whose x-value is between 4 and 7 (inclusive) and y-value is between 7 and 15 (inclusive).
e) Operation: balance y-tree (Y) – Command: **Y 3** – meaning: balance the yTree of the node that contains the x-value of 3 using the global rebalance method.
f) Operation: balance x-tree (X) – Command: **X** – meaning: balance the xTree using the global rebalance method.
g) Operation: print (P) – Command: **P** – meaning: print the xTree using preorder and inorder traversal, when you encounter each node print the yTree in preorder and inorder traversal respectively.

Regarding the *global rebalance* method: Assume that InA is an array that contains pointers to the binary search tree nodes and is stored after an inorder traversal of the binary search tree. The purpose of rebalancing a binary search tree is to convert it to an equivalent binary search tree, in which no node has subtrees whose heights differ by more than 1. The algorithm we use to perform Global Rebalance is the following:

```
Algorithm Global_Rebalance(InA,left,right) //left is initially 0 and right is n-1
        int mid  = 0;
        BST<Data>* temp = NULL;

        if (left <= right)
        {
                mid = (left  + right) / 2;
                temp = InA[mid];
                (*temp).left = Global_Rebalance(InA, left, mid – 1);
                (*temp).right = Global_Rebalance(InA, mid + 1, right);
        }
        return temp;
End
```

You are given the BinarySearchTree.h header file to use for this project. You have to add the lines of code that are missing, so that you implement the functionality described above. Remember that xNode and yNode are essentially Binary Search Tree nodes, so most of the implementation needed for them is already provided in the above header file. You just have to think about how the xNodes and yNodes are connected and use the given methods to implement the functionality of the 2-D binary search tree.

Input and Output: The input consists of an integer, that will be used later for sorting, followed by an unknown number of commands in the above format. As you will be processing each one of these input commands, you should be printing appropriate messages in the output that indicate how the corresponding operation was performed. The last command, command 'A', is used for executing the following part (approximate sorting and evaluation).

B. SORTING IN LIMITED TIME

Now we are going to compare two sorting algorithms: Bubble sort and Shell sort. Each of these algorithms have time complexity in the order of $O(n^2)$ in the worst case. Moreover, each one takes a certain number of comparisons of elements to compute the result (i.e., the completely sorted array). But what if we don't have enough time to make all those comparisons? One idea would be sacrifice the accuracy of the solution in order to come up with a solution in the available time. As such a solution would be suboptimal, we have to estimate how close it is to the optimal one.

We model this case by considering an upper bound on the number of comparisons that we are allowed to make in each one of the sorting algorithms. These two algorithms are implemented in the template file that is provided in canvas. As you can see, there is a check before every comparison to see whether after doing this comparison, we exceed the given number of comparisons or not.

If the bound on the number of comparisons we are allowed to do is greater than what is necessary for the specific algorithm, then the result will be a completely sorted array (correct result). Otherwise, we may end up with an ***approximately sorted array***. Such a solution is obviously not correct, and its quality depends on its distance from the sorted array.

The quality of the approximately sorted array is assessed by some quality measure. In this project we use the **number of inversions** as our quality measure. An inversion is defined as follows.

> *Consider a given random set of numbers σ, and let i and j be index values, and σ(i) and σ(j) the numbers at those positions. An inversion in in σ is a pair* ***(σ(i),σ(j))*** *such that* ***i < j*** *and* ***σ(i) > σ(j)****.*

Example:

Consider as example the array:

A = [1, 5, 2, 4, 3]

In order to compute the number of inversions, we work as follows. For each element A[i] in the array, we count the number of elements that are less than A[i], and are to the right of the element A[i] (i.e., we check the indices from i+1 to n-1). In particular:

- For the element A[0] it is 0 because there are no elements on its right side that are less than 1.
- For the element A[1] it is 3 because there are 3 elements on its right side that are less than 5.
- For the element A[2] it is 0 because there are no elements on its right side that are less than 2.
- For the element A[3] it is 1 because there is 1 element on its right side that is less than 4.
- For the element A[4] it is 0 because there are no elements on its right side that are less than 3.

Now we add the numbers we got: 0 + 3 + 0 + 1 + 0 = 4. Therefore, the ***number of inversions*** necessary to go from this given array example to a completely sorted array is **4**.

Input and Output: The first line of the input file, as mentioned before, contains the maximum number of comparisons that are allowed for each sorting algorithm. Starting with the sequence of identifiers that is produced when printing the preordered traversal of the above binary search tree, apply the two sorting algorithms and for each one of them, print the approximately ordered sequence that it produces, along with the number of inversions that are needed to derive the sorted sequence.

We have given you part of the code for the class above (in canvas), and you are asked to **fix** any errors and write the **code** that is missing. YOU ARE NOT ALLOWED TO USE ANY LIBRARIES OTHER THAN #include <iostream>. You are also given the main function. You can modify it, but it should follow the guidelines and produce output with the correct format. You must ensure that your program outputs the correct results.

INSTRUCTIONS AND GUIDELINES

Programming Objectives:

1. All code must be written in **standard C++** (so that it can be compiled by a g++ compiler).
2. **You have to submit**: A file named **HW3_CS2413.cpp** that contains all the code of this project.
3. **You are provided**: A file named template3.cpp including the template your submission should follow, as well as sample input and output files of your program. These files are available in "Project 3" section in Canvas.
4. You will create the classes described above.
5. The only header you will use is #include <iostream> and using namespace std.
6. All input will be read via **redirected input**. That is, you should not open a file inside the program.
7. Please refer to the sample input and output files given, and make sure to follow these formats exactly.
8. The **classes structure** should be as shown in the provided template file (you are responsible for filling-in the code that is missing, as well as for fixing any syntax errors!). We have provided code for a few methods, and you need to make sure that they work correctly. You will also write code for other methods needed.
9. The structure of your **main** program is also provided, and you should use that in your project.
10. The submission will be done through GradeScope's autograder.

Redirected Input:

Redirected input provides you a way to send a file to the standard input of a program without typing it using the keyboard. Please check out the **Visual Studio Installation and Setup guidelines**

for C++.doc provided to you on canvas. Section 3 in this document has instructions on setting up the redirected input to Visual Studio.

Remarks:

1. None of the projects is a group project. Consulting with other members of this class or seeking coding solutions from other sources including the web on programming projects is strictly not allowed and plagiarism charges will be imposed on students who do not follow this.
2. You can post questions about the projects in the respective thread in the discussion board in Canvas. Please do not include code there.
3. In case you have some specific coding question, please upload it to codePost and we will review it there.

This file will be being updated, if necessary, to reflect any further clarifications that may be given to the students by the instructor or the TAs.