CO3105/CO7105 Resit Coursework (July 2020)

Released July 20, 2020 **Deadline August 20, 2020 5:00 pm**

You should only submit the coursework if you have been offered the resit by the relevant Board of Examiners. All students who have been offered this resit should have received an email to that effect. If you have not been offered the resit and submit this resit coursework, it will not be marked.

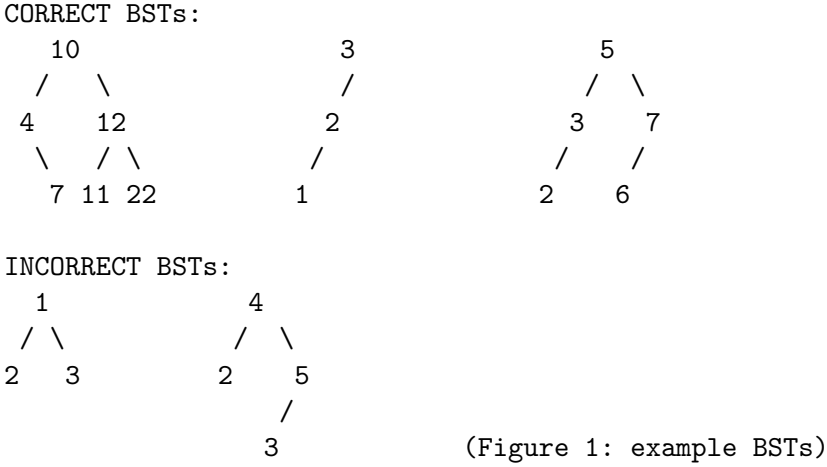
This coursework mainly assesses your knowledge on pointers and memory management, recursion, operator overloading, templates, and file I/O, in addition to general programming skills and program design.

You are asked to implement a templated data structure, and demonstrate its use in two different applications. These are detailed in the three parts below.

Part 1: The BST data structure

You are asked to complete the class `Bst`, which is an implementation of a tree-based data structure called BST. It has a template parameter `T`, which are the types of objects stored in this data structure. The type `T` can be a simple data type like an `int`, or any class, as long as it implements the `operator==` and `operator>` functions. For simplicity, in the following illustrations we assume the type `T` is integer.

The BST data structure conceptually consists of a number of nodes arranged as a tree. Each node has up to two children. Each node carries a `val` variable, which is the object (of type `T`) stored in that node. It also have two pointers, which point to the left and right child of that node, which are (recursively) also BSTs. The BST has the property that, for any node v, all nodes in its left subtree have values smaller than that at v, and all nodes in its right subtree have values larger than that at v. No two nodes of the tree can have the same value. Here the comparisons of values are according to the operators `==` and `>` defined by the type `T`. The following figure shows some examples of correct and incorrect BSTs.



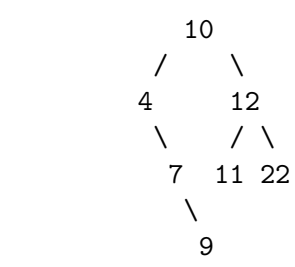
(Figure 1: example BSTs)

The `Bst` class should provide the following functions:

- Constructor with one parameter (that constructs a BST with one node) and destructor;
- Copy constructor and copy assignment operator;
- `bool insert(const T& x)`

  Inserts the object `x` into the tree. If another object of the same value (as given by the `==` operator) as `x` is already in the tree, do nothing and return false. Returns true for successful insertion.

  There is only one correct way of insertion that satisfies the left/right ordering specified above, without modifying other parts of the tree. This can be done by starting at the root, comparing its value with `x`, go to either the left or right subtree depending on whether it is larger or smaller than `x`, and repeat. For example, inserting 9 into the first example in Figure 1 results in the following:



(Figure 2: insertion)

- `bool find(T& x)`

  `bool find(const T& x)`

  Search the tree for an object of value `x`. Again, this should start at the root and (if not equal to `x`) recursively go to either the left or right subtree.

- `int height(const T& x)`

  Return the height of the object `x` in the tree. The height is defined as the length of the path from the root to that node. For example, in Figure 2, 10 has a height of 0; 4 and 12 have height 1; 7, 11, 22 have height 2; and 9 has height 3. If `x` is not in the tree, return -1.

- `std::vector<T> toVector()`

  Returns all the entries of the tree, as an STL vector of objects of type `T`. The entries should be sorted according to `operator>` with the 'smallest' object first.

The precise specifications of the above functions, which you must follow, are given in the header file `Bst.h`. You will have to implement all the functions in that file.

You are already given private member variables to the class, which you should not change, except if you wish to use smart pointers you can change the pointer member variables accordingly. If using raw pointers, you must take care of all memory management issues.

You must not change the existing public interface of the classes, but you are allowed to add private or public member functions.

The `main1.cpp` file shows some examples of how this class is used. This is not part of the submission.

Hint: you may find recursion useful.

Part 2: A telephone directory

In this part you are asked to implement a small program that uses `Bst` in Part 1 to answer telephone directory queries of a user. Specifically it should do the following:

1. It reads a file `phonebook.txt` in the current directory. Each line in the file contains three comma-separated fields, which represent the first name, last name and telephone number of a person. All three fields should be treated as strings. Note that there may be spaces, hyphens etc in these strings, which should all be kept.

   For example a line may look like this

   `John,von Neumann,0116-234-5789`

   You can assume the file conforms to the above format and have no errors. You can also assume there are no two people in the file with identical firstname and lastname; however there can be people with same lastname but different firstnames (or vice versa).

   The program should create an appropriate BST to store the read contents.

2. The program repeatedly asks the user to type in the last name, followed by firstname. It should then search the data structure and display the phone number of this person (or that such a person is not found). This repeats until the user types "exit" or "quit" as the lastname. Here is an example execution:

```
Welcome to the telephone directory system

Enter lastname of the person to search for:
Smith
Enter firstname of the person to search for:
John
The phone number of John Smith is
0123 456 7890

Enter lastname of the person to search for:
Smith
Enter firstname of the person to search for:
Mary
No records found

Enter lastname of the person to search for:
quit
```

You should implement this application in the file `main2.cpp`.

Hint: The class (type) `T` of BST can have multiple member variables (fields), and the comparison operators `==` and `>` can be defined based on only some of these fields, with the rest of the member variables being data to be carried.

Part 3: Optimising insertion orders of BSTs

In this part, you should write an application using the BST data structure in Part 1. BSTs containing the same set of objects can be of different 'shape', depending on the order the objects

were inserted. For example, if the numbers 1, 2, 3 are inserted into an initially empty BST, the result is the left one below, but if it is inserted in the order 2, 1, 3, the right one is the result.

You are asked to implement a program that uses `Bst`, which will experimentally find the insertion sequence that minimises the total height of the nodes in the tree. More specifically it should do the following.

1. It prompts the user to enter two positive integers, denoted N and T below, which represents the number of integers and the number of tries.
2. The program then repeatedly do the following, for a total of T times:
   - 2.1 Arrange the integers $0, 1, 2, \ldots N-1$ in a random order.
   - 2.2 Insert these integers, in this random order, into a BST.
   - 2.3 Calculate the total height of all nodes in the final BST. The height is as defined in Part 1.
3. The program should find, among the T tries, the one with the minimum total height. It should then display this height and the ordering of those N integers.

An example output may look like this:

```
Enter a positive integer N:
7
Enter a positive integer T:
100

The best order is
3 1 0 2 5 4 6
and the total height is 10.
```

Note that as this program simply tries to find the minimum by going through many possibilities, it may not find the actual minimum solution. Also there are many equally optimal (minimum) solutions.

You should implement this application in the file `main3.cpp`.

Marking criteria

You are expected to submit code that constitutes a correct implementation of the required class template and functions, and to provide comments sufficiently explaining the logic of your code. 10% of the mark will be given for submitting well-commented code and for following good C++ coding practices. 90% of the mark is based on the correctness of your implementation, to be assessed by testing and manual inspection. Parts 1, 2 and 3 constitutes 50%, 20% and 20% of this mark respectively.

The correctness part uses a project-style marking criteria; there will not be separate marks for passing test cases, and there are no runnable test suites as in previous assignments/tasks. Any testing will only be done to assist code inspection. However, your programs will be tested on the departmental linux system. Please make sure your programs are compilable and runnable on the departmental linux system.

Further details of marking criteria can be found on the module website.

Submission instructions

You are provided with the following files:

- `Bst.h`: contains the incomplete definition of the class template `Bst` and the detailed descriptions of its functions.
- `main1.cpp`: A sample of how `Bst.h` can be used.
- `main2.cpp`: This is where you should implement Part 2.
- `main3.cpp`: This is where you should implement Part 3.
- A makefile.

You should submit only the files `Bst.h`, `main2.cpp` and `main3.cpp`, by email to the convenor. All your code should be contained in these three files. You do not need to submit `main1.cpp` or the makefile.

Anonymous marking is achieved by having only the userid in the submission process. Please do not write down your name or other identifiable information in your submission.