Programming Assignment 1: C++ Warm up with BSTs

Checkpoint deadline: Wednesday, Jan 18, 11:59pm
Final deadline: Friday, Jan 20, 11:59pm

Required solution files for checkpoint submission: BSTInt.cpp, BSTInt.h, BSTNodeInt.cpp, BSTNodeInt.h, testBST.cpp, questions.txt

Required solution files for final submission: BSTInt.cpp, BSTInt.h, BSTNodeInt.cpp, BSTNodeInt.h, testBST.cpp, questions.txt, BST.hpp, BSTNode.hpp, BSTIterator.hpp, main.cpp

Provided files: BSTInt.cpp, BSTInt.h, BSTNodeInt.cpp, BSTNodeInt.h, testBST.cpp, BST.hpp, BSTNode.hpp, BSTIterator.hpp, Makefile, actors_sorted.txt, actors.txt, actors100.txt

Submission Scripts:
We have also added the submission scripts for the checkpoint/final milestones in vocareum.

You can look at them by going into the resource dir in the vocareum terminal from your $work dir as follows.

```
cd ../resource
cat scripts/submit.sh
```

All the supporting files for running the submission scripts are present in

```
cd asnlib/
```

This might help you debug your issues with submissions for checkpoint/final. You are free to copy them and use them
Start early, and submit early. Our submission scripts will probably uncover errors in your code. You need to leave yourself time to debug after these errors are found!

Assignment Overview

In this assignment you will:

- Implement a "plain" and a template-based Binary Search Tree in C++ following the conventions of the C++ Standard Template Library
- Explore different correct and incorrect ways to write a Makefile
- Debug errors in C++ code
- Write test code to catch errors in a Binary Search Tree implementation

Grading Overview

This assignment is out of 26 points. There are 6 points for the Checkpoint, and 19 points for the final submission.

Checkpoint grading (6 points)

- Pre-survey completion: 1 point
- questions.txt: 2 points
- BSTInt correctness: 2 points
- Tester completeness/correctness: 1 point

Final Submission grading (19 points)

- questions.txt: 3 points
- BSTInt, and BST+Iterator correctness: 10 points
- Tester correctness/completeness: 2 points
- main program: 1 point
- Commenting and style (see the minimal style guide): 3 points

Checkpoint Instructions

The following steps must be completed by the checkpoint deadline.

0. Sign the Academic Integrity Agreement and Complete the pre-survey

Before you begin this assignment, please make sure you have filled out the academic integrity form here. **If you do not do so, you will not receive credit on this or any other CSE 100 assignment.**

Next, fill out a pre-survey, worth 1 point for filling it out.

- Here is the link to the survey

1. Get the starter code and get set up to code

Please follow the instructions provided in this PPT or video links below to get started with the assignment. It contains instructions for

- **associating your vocareum account with github -** video link
- **setting up the code for PA1 and submitting code for checkpoint and final milestone -** video link

In the first part of this assignment (the checkpoint) you will be constructing a class that implements a binary search tree for storing ints only. In the second part of this assignment, you will build a more generic version of your BST class by adding templates and modify the functionality of this class to better match the C++ STL standards.

2. Compile the code (and answer some questions)

In this course we will always use Makefiles to compile the code for our programming assignments. I assume you have written a Makefile before (in 15L if not elsewhere), but you might not exactly remember how it's done, so some of this assignment help you refresh your memory.

Compile the code using the command:

> make bst

There should not be any compile errors. Also, take a look at the provided Makefile. You'll notice that it's actually setup to compile the code from more than just this checkpoint. Then answer the following questions in a plain text file called "questions.txt". You will continue to add to this file throughout the assignment.

Once you create questions.txt you will need to make sure to add it to your repository, following the instructions in the book about adding files to git. This holds for any new required file you create for this assignment.

Make sure you label each answer with the question number, at least.

Q1. What are all the files that are created when I type 'make bst' for the first time?

Q2: In the file BSTInt.cpp, remove the name scope from the insert function. That is, change the function header from

bool BSTInt::insert(int item)

to

bool insert(int item)

Copy and paste the error that results when you now compile the code into your questions.txt file, and then explain what the error means and why it occurs.

As a side note, if you want to just try something out quick, there's no need to use a makefile (or separate .h/.cpp files). If it's a quick function just trying out something we do in class, feel free to put the function definition, together with a main function into a file called tryit.cpp (or whatever) and then compile on the command line using the simple line:

> g++ -std=c++11 -o test tryit.cpp

This will produce an executable called test which you can run. If you leave out the "-o test" option the executable file produced will be called a.out. I encourage you to use this quick method of trying things out. But for the PAs, please use Makefiles.

3. Run the code, write tests, and locate and fix the bug in the provided code

We've provided you with some starter code, but there's a bug in the provided code in either the find or the insert function. Run our basic tester by running the executable produced by the previous step:

> ./bst

Notice that the code actually passes all of the provided tests for both the find and the insert functions!

In testBST.cpp file, add more thorough tests for both the find and the insert functions. These tests must expose the bug in the implementation, but they should also expose any other bugs that might occur with find or insert. That is, if we ran your tester on a buggy version of either of these methods, it should fail. Then locate and fix the bug. You may use any method you like to debug the program, but we strongly encourage you to try the gdb debugger.

In your questions.txt file, write the answer to the following questions:

Q3: In a sentence or two, describe the bug in the code. What function was it is, and what was it?

Q4: In about a paragraph, describe the process you used to find an fix the bug. Include any tools you used and how you used them. If you were able to spot the bug by just looking at the code, describe your reasoning process in detail. We're not looking for the corrected code here (we already have that in your .cpp file). Rather we're looking for a description of your debugging process. Please be specific.

Have you committed to your repo and pushed to github recently? If not, do it now! Remember, commit and push early and often.

4. Implement and test the destructor

Run valgrind on the BST tester:

> valgrind ./bst

(there are lots of options you can use, but this is enough for now)

Notice under LEAK SUMMARY it reports that some memory was "definitely lost". (For details of what all the LEAK SUMMARY lines mean, see: http://valgrind.org/docs/manual/faq.html#faq.deflost) Thus, you have a memory leak. This is because you haven't written the destructor!
Write the destrutor now by writing the deleteAll method. You need to delete all of the memory your have created using new at any time in your program. But there is no need to set pointers to null (the memory

that holds them is going away anyway, so you won't have a dangling pointer problem). Ensure deleteAll method works correctly for trees of all sizes.

Test your program again using valgrind to ensure the memory leaks are gone.

5. Implement and test the remaining BSTInt methods

The final part of the checkpoint requirement is to complete the implementation of the BSTInt class by (1) filling in the missing method definitions in BSTInt.cpp, and (2) writing test cases for these methods in testBST.cpp. Note that we will grade you both on your implementation and on your test cases. That is, we will (1) run your BST code on our own tests and (2) run your tester on known buggy implementations of a BST. To receive full credit, your BST must pass all of our tests and your testBST.cpp must fail by returning -1 on buggy implementations and pass by returning 0 on correct implementations of a BST.

6. Submit your checkpoint

To submit the checkpoint milestone make sure you have all the required files added to your repository, all your changes committed, and the repository pushed to GitHub. Once you have your files to submit in the repo on GitHub, simply pull the code from github into vocareum by running,

git pull

on Vocareum. Then click on the submit button. Make sure you submit code for checkpoint milestone. For detailed instructions look at the ppt or video.

Final Submission Instructions

In the rest of this assignment you will build a template-based BST class, together with an iterator class, that closely matches the behavior of the BST built-in to the C++ STL (the C++ set class).

0. Fix any errors in the code you submitted for the checkpoint. You'll be graded again on it at the final submission.

1. Open and examine the provided code for the templated BST class: BST.hpp, BSTNode.hpp and BSTIterator.hpp

There are a few differences between the code you just wrote and what you will write for the templated BST class. The most important is that you must write all of the code for the templated BST class in one file, rather than separating it into a .cpp and a .h file. So we give these files the extension .hpp, to indicate that it's a combination header/implementation file. The reasons for this are a bit complicated, but have to do with the fact that C++ dynamically generates code for templated classes at link time.

Open the files BST.hpp, BSTNode.hpp and BSTIterator.hpp and take a look at the methods you'll be asked to implement. You'll notice that many, of these methods are very similar to the methods from the BSTInt class you just implemented.

Also, you'll recall from the checkpoint that the Makefile is already set up to compile the code for this part of the assignment.

Answer the following question in your questions.txt file:

Q5: Look at header for the insert method in the BST.hpp and compare it to the header of the insert method in BSTInt.h. For each class state whether the insert method passes its argument by value or by reference. Explain why each method probably uses the method that it uses (i.e. if pass by value, explain why not pass by reference, and if pass by reference, explain why not pass by value).

2. Implement the missing methods in BSTNode.hpp, BSTIterator.hpp, and BST.hpp, and add tests for these methods in testBST.cpp.

This is the main part of the assignment. You will complete all of the missing methods in your templated BST + iterator classes. These methods include:

In BST.hpp:

- The private `deleteAll` method, used to implement the destructor
- The `insert` method. Notice that it returns a pair instead of a bool. See the comments in the code for specifics.
- The `find` method. Notice that it returns an iterator instead of a bool. See the comments in the code for specifics.
- The `size` method.
- The `height` method.
- The `empty` method
- The private `first` method, which is used in the begin method

In BSTNode.hpp:

- The `successor` method. This method is used in the BSTIterator's ++ operators to advance the iterator.

In BSTIterator.hpp

- The overloaded == and != operators

Make sure you understand the iterator pattern and what an iterator is before you begin coding. If you have any doubts about how these classes work together, come ask an instructor, TA or tutor.

We have provided some elementary tests for the templated BST class and its iterator, which are commented out in testBST.cpp. You should uncomment these tests and **add your own.** Remember that part of your grade will be based on whether your tester code correctly catches errors in buggy BST implementations. The main method in your tester must return non-zero when tests fail.

3. Implement a main method in main.cpp to explore different BST structures

Your program should do the following:

1. Open a formatted .txt file input from the command line. This has already been included in the skeleton code. When running the program, you need to type in **./main <input filename>** onto the command line.
2. Read in a formatted .txt file with various actor names, line by line and load them into a BST. Each line of the .txt file is a unique actor name.
3. Print the size of the BST.
4. Print the height of the BST.
5. Prompt the user to enter an actor name.
6. Print the result of the search (whether or not the actor name is found)
7. Continue to prompt the user to enter an actor name until the user elects to quit, by entering “n” and then the return key.

NOTE: main.cpp gives more details as to the specific formatting of the input, output, and prompts. If you deviate in any way from that output format, you will likely lose a lot of points for this PA.

Once your program works, use it to explore the height of your BST with different inputs. Notice that we have provided two files: actors.txt and actors_sorted.txt. One contains actors' names in random order and the other contains the same names in sorted order. You can create smaller files containing subsets of these files using the 'head' command and then redirecting the output to a new file (e.g. "head -10 actors.txt > actors10.txt"). Explore how the size and the height of the trees differ for the sorted vs the random files for different sizes of files, and then answer the following questions in your questions.txt file:

Q6: Which input file produces taller trees (sorted or random)?

Q7: Research on your own the height you would expect for the tree produced by the sorted input and for the tree produced by the random input. Give the approximate height you would expect for each tree based on the size of the tree, N. Then build the tree for different sizes of sorted and randomly ordered input, report your results, and comment on whether what you see matches your expectations. Be sure to cite your source for the heights you expect.

4. Submit your assignment

To submit the checkpoint milestone make sure you have all the required files added to your repository, committed, and pushed to GitHub. Then, simply pull the code from github into vocareum by running,

git pull

on Vocareum. Then click on the submit button. Make sure you submit code for final submission milestone. For detailed instructions look at the ppt or video.

Star Point (Completely Optional)

If you're gotten here and you still want more, we're providing this "star point" option. However, this is just a suggestion. Any significant extension or project will be considered for a star point. If you're interested, keep

reading.

First, what is a "star point"? Star points are challenging, open-ended extensions designed to engage those who really want to learn more and go beyond the basic requirements. However, these are not extra credit. If you do "enough" star point and are "close enough" to the boundary, you may be moved up to the higher grade, but do these star point extensions because you are intellectually curious and want a challenge. Not for the grade. The course staff will not answer questions like "If I do this, will I get a star point?". If you are doing the star point extension just to get the star point, then you're doing it for the wrong reason. Only do it if you would be happy whether or not you get the point in the end.

This assignment's suggested star point extension is to learn about and implement a decision tree, which is a tree-structure used in machine learning. Here are the steps to completing this extension.

1. Learn about Decision Trees, including how they work, how to use them for classification, and how to build them from data. You can use any resources you like, but we recommend the following series of videos which provide a pretty good and concise introduction: http://bit.ly/D-Tree You should initially watch the first three videos in this sequence, and then optionally watch more, depending on how far you want to take your extension.

2. Implement a Decision Tree Classifier that can learn from data. At a minimum you should implement the ID3 algorithm without using information gain to select the best attribute to split. That is, it's OK to simply select attributes randomly. Of course you are free to be as sophisticated as you like with your decision tree learning. Your classifier must support at least the following two operations:

- Learn from data: Given a data set, train the decision tree classifier to classify new examples with the same attributes of the training data.
- Classify new example: Given a new example with the same attributes as the tree was trained on, output a decision.

Note that it is fine to hard-code your decision tree for a single data set. You do not need to make it general purpose in terms of the attributes, but it must adapt to the data on which it is trained. Also, you can implement the above functionality however you choose, but make sure you explain your implementation sufficiently (see below).

3. Illustrate your Decision Tree in action. Train your decision tree on (a subset of) the data set of your choice and then use it to classify labeled examples from that data set on which it was NOT trained. Calculate the accuracy of its classifications. Here is one possible data set to use: http://archive.ics.uci.edu/ml/datasets/Mushroom, but you are free to use any other data set you want.

4. Write up your implementation and your results. This is the most important step, and without it you will not get any credit for this star point. Provide a writeup in a PDF or plain text file that describes the following:

- The classes and functions you implemented and how they implement the functionality described above. Include how you selected the attributes to split on (whether you used random selection, or

information gain) and if you included any pruning or other optimizations to improve the classifier.
- Where to obtain the data that your decision tree can be trained/tested on
- How to train your decision tree
- How to test your trained decision tree on additional examples
- The tests you ran and the accuracy of your decision tree on the test set

Again, the most important component of your star point is this writeup. We want to know that you really explored this topic and learned something. It's up to you to let us know that you did.

The starpoint will appear as a separate assignment on vocareum, Follow instructions shown in ppt to create your github repo for the star point.
Note that the github repo for starpoint will be empty. Add all your code and write up and push it to github. You can finally submit it using instructions as discussed earlier.