# CSE 100 PA1: Implementing BST operations in C++

Checkpoint deadline: **Friday April 13, 11:59pm** (No late days allowed for checkpoint)

Final deadline: **Friday, April 20, 11:59pm**

Starter code:
https://drive.google.com/a/eng.ucsd.edu/file/d/1_goJwpMMb5jOyR4MSPVREx-H9TFF1-lD/view?usp=sharing

## Read first

- **Start early, and submit early (and often).** Our submission script will probably uncover errors in your code. Leave yourself time to debug!
- **Make sure to click "Submit" button on vocareum. Only uploading your work without clicking "Submit" button is NOT considered as a valid submission.**
- **Note that taking code from your classmates/any outside resources is strictly prohibited** and will be considered as a violation of academic integrity. We will run a sophisticated third-party code plagiarism detector on every submission, therefore **DO NOT CHEAT**. You all have done the source code plagiarism tutorial. When in doubt, refer to the tutorial or ask your instructors.

## Assignment Overview

In this assignment you will:

- Implement 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

## Required files for checkpoint submission

BSTInt.cpp, BSTInt.h, BSTNodeInt.cpp, BSTNodeInt.h, testBST.cpp

## Required files for final submission

BSTInt.cpp, BSTInt.h, BSTNodeInt.cpp, BSTNodeInt.h, testBST.cpp,
BST.hpp, BSTNode.hpp, BSTIterator.hpp, main.cpp

## Starter 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

### Grading Overview

This assignment is out of **100 points**. There are **15 points** for the checkpoint and **85 points** for the final submission. There are also **10 points** for extra credit. **\o/**

***Checkpoint (15 points)***

- BSTNode correctness: **10 points**
- Tester completeness/correctness: **5 points**

***Final submission (85 points)***

- BSTNode and BST + Iterator correctness: **60 points**
- Tester completeness/correctness: **10 points**
- main program: **15 points**

***Extra credit (10 points - details in EC section)***

***Style is NOT graded.*** *As a CSE 100 student, you should have developed a solid understanding of good code styles.*

Before you begin this assignment, please make sure you are done with PA0 and completed the academic integrity form and code plagiarism tutorial mentioned in PA0.
**If not completed, you will not receive credit on this or any other CSE 100 assignment.**

# Checkpoint Instructions

We highlighted the most important instructions like this :^) but please do read through everything!

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

In 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). See the first page of this document for the starter code link.

### 2. Examine and compile the code

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.

*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, 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 test cases for testing **`find`** and **`insert`** functions from BSTInt class. 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.

### 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". Thus, you have a memory leak. This is because you haven't written the destructor!

*For details of what all the LEAK SUMMARY lines mean, see:*
*http://valgrind.org/docs/manual/faq.html#faq.deflost*

Write the destructor 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 **make sure you submit code for checkpoint milestone** on vocareum (PA1 has two parts on vocareum: checkpoint and final submission).

## 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.

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

NOTE: While going through the code, please note that **we only allow you to use "<" operator to compare "Data" elements**. Please assume that the "Data" **has and only has** an overloaded "<" operators and the rest of comparison operators are not available for use.
(That is also the reasoning behind only using "<": developers only need to overload one operator for the program to work.)

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. **Also, read through method headers before you implement any methods.**

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. Explore different BST structures (main.cpp)

Implement a main method in main.cpp. 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.**

### *Optional (very beneficial) section

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 think about the following questions:

Q1: Which input file produces taller trees (sorted or random)?

Q2: 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.

**Answers to these questions are NOT needed for PA1 submissions and will NOT be graded.** Practice them for your own knowledge.

### 4. Submit your assignment

To submit the checkpoint **make sure you submit code for final submission** on vocareum.

# Extra credit Instructions

## Files required for extra credit submission:

**Makefile, BSTExtraCredit.hpp, testBSTExtraCredit.cpp**
(Submit them along with your other final submission files on vocareum. There is no separate milestone for extra credit on vocareum. Deadline for EC is the same as the final submission deadline.)

For the extra credit, you will research **(3 pts)** and implement **(7 pts)** a self-balancing binary search tree. You have the full flexibility on how you design and implement your self-balancing BST. However, there are some requirements for your submission:

(1) You must implement all code related to the extra credit part in a separate file named **`BSTExtraCredit.hpp`.** This is to ensure that your implementation does not affect your regular submission.
(2) You must describe your design/algorithm clearly and thoroughly in comments **IN YOUR OWN WORDS**.
(3) Your self-balancing BST must have the following methods just as in **`BST.hpp`** (keep the method signatures the same)**:**
   - The **`insert`** method.
   - The **`size`** method.
   - The **`height`** method. Note that in a self-balancing BST, the difference between nodes with the highest height and the lowest height is no more than 1.
   - The **`empty`** method

   You may create any helper methods for any methods mentioned above.
(4) You must provide us with a few test cases in a file named **`testBSTExtraCredit.cpp`** showing that your self-balancing BST works.
   (eg. make a for loop to insert 100 **sorted** elements into your self-balancing BST and print out the height of the tree. )
(5) You must add a target rule in **`Makefile`** to create an executable called **`bstec`** that runs your test cases from **`testBSTExtraCredit.cpp`**. An example can be:

```
.....
all: bst main

bstec: testBSTExtraCredit.o $(BSTHPP) BSTExtraCredit.hpp
	$(CC) $(FLAGS) -o bstec testBSTExtraCredit.o

bst: testBST.o $(BSTOBJ) $(BSTHPP)
	$(CC) $(FLAGS) -o bst testBST.o $(BSTOBJ)
.....
```

*(Feel free to modify it!)*

Some resources that you can look into:

1. Splay Tree
2. AVL Tree
3. Treap

...and more online! :^) **If you have any questions and/or want any clarifications, ask on piazza!**

## Extra Credit Grading (10 pts)

- Clear & Thorough Design/Algorithm 3 pts
    - If you are running out of time and/or will not implement your self-balancing BST, you can still attempt some extra credit! Do your research and comment your design/algorithm in your **`BSTExtraCredit.hpp`.**
    - **Do write everything IN YOUR OWN WORDS.**
- Implementation 7 pts
    - Remember that you have the full flexibility on this part, as long as you follow the requirement (1)-(5) listed above.