

Programming Project 1

Checkpoint Deadline: 11:59pm on Thursday, 10/11 (not eligible for slip day)

Final Deadline: 11:59pm on Thursday, 10/18 (slip day eligible)

Overview

In this project, you will be asked to complete two major tasks. The first is to implement a number of BST methods. This part is essentially a warmup on C++ programming in the STL. The second part is to implement a few core methods for a new data structure, K-D Trees. You may not rely on other libraries' implementations in your solutions. **Your first step is to read these instructions carefully.** We've given you a lot of advice here on how to succeed, please take it. And please don't post a question on piazza asking something which is already answered here.

[Pair Programming Partner Instructions](#)

[Retrieving Starter Code](#)

[Part 1: Implementing a Binary Search Tree in C++](#)

[Part 2: KD Tree](#)

[Testing Your Code](#)

[Submitting Your Checkpoint and Final PA](#)

[Getting Help](#)

Pair Programming Partner Instructions

If you wish to work with a partner, please be sure to read the guidelines for Pair Programming in the syllabus carefully. Once you are both sure you can work together as a pair, please fill out the form below:

<https://goo.gl/forms/y2E1IBtAq4noGjF03>

Should a relationship dissolve, you can fill out the following form to request a break-up. But by doing so, you are agreeing to delete all the work you and your partner have completed. Any code in common between dissolved partnerships is a violation of the Academic Integrity Agreement.

<https://goo.gl/forms/9HjopYtZImYdQR2t2>

Retrieving Starter Code

For PA1, you are given a fair amount of initial code and additional files to help you get started. To retrieve these files, you need to login to your CSE 100 account (**cs100f<???**>, or your **<username>@ucsd.edu**) on a linux machine in the basement or ssh into the machine using the command “**ssh cs100f<???**>**@ieng6.ucsd.edu**”. When logging in, you will be prompted for a password. Once you are logged in, open up a linux terminal and run the command “**getprogram1 <directory>**” where **<directory>** is the name of a folder that you want to be created to hold the starter code. After this command is run, you should see that the directory you specified has been created if it previously did not exist, and the directory should contain the starting files. The following files *require* implementations: **BST.hpp**, **BSTNode.hpp**, **BSTIterator.hpp**, **KDT.hpp**, and **main2.cpp**. **KDT.hpp** and **main2.cpp** are required only for the final submission, so they *do not* need to be completed by the checkpoint. The files you retrieve should be the following:

```
├── Makefile
├── create-submission-zip.sh
├── data
│   ├── actors.txt
│   ├── actors_sorted.txt
│   ├── points.txt
│   ├── queryPoints.txt
│   ├── solutionPoints.txt
│   └── test_points.txt
├── src
│   ├── BST.hpp
│   ├── BSTIterator.hpp
│   ├── BSTNode.hpp
│   ├── KDT.hpp
│   ├── main.cpp
│   └── main2.cpp
└── test
    ├── test_BST.cpp
    └── test_KDT.cpp
```

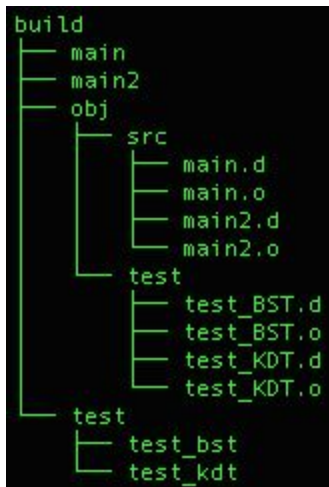
To complete the assignment, you will modify the files in the starter code to provide full definitions for functions marked with `// TODO`. To see all of the `TODO` tags, you can run the command `<<grep -r TODO .>>` in your PA1 directory.

(We personally recommend using `grep` like this:

```
> grep -r -A 5 -B 5 TODO *pp
```

as you can manipulate the arguments after “A” and “B” to get a good snapshot of what is required.)

In order to compile your code and create the main files, simply run `make`. After running `make` you will see that a new directory called `build` will have appeared.



To run `main2` file type `./build/main2` from your project root dir, to run `test_bst` run `./build/test/test_bst`, etc.

The object files under the `obj/` directory are needed to generate the executables, you cannot run them.

You can also choose to compile specific executables. By running "`make`" without any arguments like we just did we ran the "default" build target in the Makefile under:

```
# === Build targets ===
```

To run a different build target like "`test_bst`", you can run "`make test_bst`".

⚠ ATTENTION Our tests compile your code with the `make` command, so you will need to *make sure* that the `make` command works before you submit your code.

Part 1: Implementing a Binary Search Tree in C++

In this part of the assignment, you will implement the fundamental BST operations using concepts from the C++ Standard Template Library. Your BST *does not* need to handle duplicates. You will be able to test your code using an interactive program that will prompt the user to enter a query and check if the query exists in the BST until the user elects to quit.

Your first task is to understand the code. Do not attempt to start writing code until you have a firm grasp of what the provided code does and what you are supposed to do with it. **Nearly all the required functionality is specified in the code and the comments, so you'll need to read everything carefully.** Make sure you can

understand the provided test file (`test_BST.cpp`), which partially tests some aspects of the BST class, and then add your own tests *before* you start implementing any code of your own.

Note that, when you implement these functions, any comparisons you make between two elements **must be done using the < operator**. The reason for this is that some data types in this project do not have the other operators (>, <=, >=, etc.) implemented, so for safety reasons, it's best to only use < to compare. Remember: given an element, you need to check if it's less than, greater than, or equal to elements currently in the BST. How can you do that with just the < operator? If you cannot figure it out, ask a tutor to give you a push in the right direction.

Additionally, you will also have for your convenience a main function in `main.cpp` that does the following:

1. Opens a formatted `.txt` file input from the command line. This has already been included in the skeleton code. Usage is as follows:

```
> ./main <input_filename>
```
2. Reads in the 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. Outputs the size of the BST (i.e., the number of elements it contains)
4. Outputs the height of the BST (i.e., the number of edges in the longest path from the root to any leaf where **a leaf node is height 1**)
5. Prompts the user to enter an actor name
6. Outputs the result of the search (i.e., whether or not the actor name is found)
7. Continues to prompt the user to enter an actor name until the user exits by entering "n" (no quotes)

The `Makefile` is provided for your convenience. Running `make` in the directory will compile your code and create the executable files. You are free to modify the `Makefile` as you see fit: as long as we can enter your repository's folder, run the `make` command, and get two executable files `test_bst` and `main` whose usage are exactly how we have specified, then your `Makefile` is good to go. Realize that running `make test_bst` will give you a number of warnings until you start making progress in implementing the missing functions. (We recommend just using `make test_bst` while working on part 1. This will be faster and hide the warnings about unimplemented features of the KD-tree for part 2.)

If you have any doubts at all about the provided code, feel free to visit a tutor in B260 or post on piazza (see "Getting Help below"). Start early, start often!

Part 2: KD Tree

(No need to read this section until you have completed the first checkpoint. Continue reading at “Testing your code”)

Binary search trees are useful data structures and their fundamental structure is used in other data structures. As such, they have been modified and adapted to help solve a variety of real-world problems. A few of these problems are graphics-related, and we will explore this in part 2 of PA 1.

KD Trees are used to keep track of a set of points and to find the point in that set that is closest to another arbitrary point. They can also help you find all points within a particular region (but we won't implement that here). KD Trees can be used on points of any arbitrary dimension size, but we will stick to using 2D points. (In fact, you are implementing a 2D-Tree, though we'll call it a KD Tree.)

KD Trees enforce a sorted ordering on points by alternating the dimension that is being searched through at each layer of the tree (e.g., first use the x coordinate, then use the y, ...). If you have all the relevant points at the time you build the tree, each node will split the tree in half by nodes that have a dimensional value less than or equal to the given node, and nodes that have a dimensional value greater than or equal to the given node. Again, which dimension is being considered rotates at each layer down the tree. These alternating dimensions need to be taken into account for both the “build” and “findNearestNeighbor” functionality. The nearest neighbor algorithm also takes into account the square distance between two points which you will need to implement as well.

Note that we are inheriting from BSTs as there are many commonalities between the two Trees. But, there are some methods inherited from BST which will not work properly. The right solution is to implement them, but we felt this made the assignment too large. So we've simply had those methods fail if you try to run them. You could imagine your work as an in-between point on the project with someone else writing those new, fixed, methods.

As with part 1, you will want to search the KDT.hpp file for “TODO” statements to find what parts you are required to implement.

Additionally, you will also have to implement the much of the main function in `main2.cpp` that does the following (feel free to use our `main.cpp` as a template):

1. Opens a formatted `.txt` file input from the command line. This has already been included in the skeleton code. Usage is as follows: `./build/main2 <input_filename>`
2. Reads in the formatted `.txt` file with a line by line list of points where the x and y coordinate are separated by a space, and uses this file to build a KD Tree
3. Outputs the size of the resulting KD Tree (i.e., the number of elements it contains)
4. Outputs the height of the KD Tree (i.e., the number of edges in the longest path from the root to any leaf)
5. Prompts the user to enter a point as an x and y double value
6. Outputs the nearest neighboring point from the points text file to the input point.
7. Continues to prompt the user to enter an Point until the user exits by entering "n" (no quotes)

NOTE: `main2.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.

Testing Your Code

You will want to be sure to test your code at every step. As a suggestion, you should write your own method which prints out all the details of the tree - that includes node contents along with left, right, and parent pointers. The time spent writing this code will more than offset the time you will save in debugging.

In C++, using the `cout` keyword is used to output values to `stdout`, but using `cerr` instead outputs to `stderr`. These can be convenient for easily filtering out program output in the terminal. Examples of how both of these are used can be found throughout the starter code. For added convenience, we also overrode the `<<` operator for the `BSTNode` class so that outputting a `BSTNode` object will show you not only the node's data but also its memory address and the memory address of its children and parent.

As a second suggestion, we recommend using debug print statements. These are statements which, if a `#define` flag is turned on, will print what the method is doing and possibly the state of the tree before and after. If the `#define` flag is turned off, all comments go away. It's really helpful to get that debugging output from every method when you get stuck (and is significantly better than commenting in/out print statements). There are lots of ways to do this, but a nice suggestion from [stackoverflow](https://stackoverflow.com) is:

```
#define DEBUG 1
#if DEBUG
#define D(x) x
#else
#define D(x)
#endif
```

You can then do debug print statements like this: `D(cout << ... ;)`. But feel free to do what works best for you.

Most importantly, we encourage you to write your own BST Tester (feel free to use and extend `test_BST.cpp` and `test_KDT.cpp` as an example) which tests method by method as you add them to the code base. Be sure to test every method, no matter how simple. You don't want to spend two hours debugging your insert method only to realize you messed up an equality test (trust us, we've done it...). (The `test_*.cpp` files we've provided for you stop execution at the first failure, so if you're extending those files, add your tests above the ones already there.)

You should wait with submitting your code until you are fairly confident all your code works properly as our grading scripts, for the most part, expect a fully functional BST code and may fail if there are unimplemented methods (as the code fails, it may produce unreasonable grading output). Also, beware, our testing scripts use randomization, so if you've run it once with full credit, there is no guarantee that with new random values you'll get the same score when run on gradescope. This is all to say: *Testing your code is ultimately your responsibility.*

Lastly, our grading scripts use various methods to ensure your code is as efficient as it should be. For example, in part 2 of the assignment, an inefficient `findNearestNeighbor` (one which fails to properly prune the search space) may lose significant points. **Be sure to remove any debugging output from your BST before running the grading script** as I/O is expensive and could make an efficient method seem inefficient.

Academic Integrity and Honest Implementations

We will hand inspect, randomly, a percentage of all submissions and will use automated tools to look for plagiarism or deception. **Attempting to solve a problem by other than your own means will be treated as an Academic Integrity Violation.** This includes all the issues discussed in the Academic Integrity Agreement, but in addition, it covers deceptive implementations. For example, if you use a library (create a library object and just reroute calls through that library object) rather than write your own code, that's seriously not okay and will be treated as dishonest work.

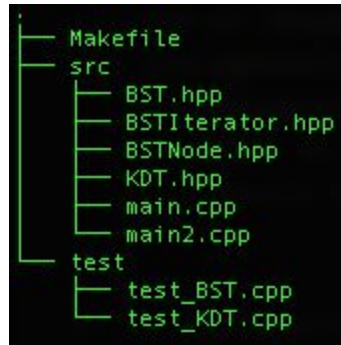
Submitting Your Checkpoint and Final PA

When you have completed all of the requirements for the final submission, you should follow these instructions to submit your code:

1. There will be 3 assignments on gradescope:
 - PA1Checkpoint, for part 1 at the checkpoint deadline
 - PA1FinalBST, for part 1 at the final deadline
 - PA1FinalKDT, for part 2 at the final deadline

2. For each of the 3 assignments on GradeScope you need to submit a zip file.

- Run “./create-submission-zip.sh”. Which will create such a zip file for you.
- Make sure it includes at least the following files.



```
├── Makefile
├── src
│   ├── BST.hpp
│   ├── BSTIterator.hpp
│   ├── BSTNode.hpp
│   ├── KDT.hpp
│   ├── main.cpp
│   └── main2.cpp
└── test
    ├── test_BST.cpp
    └── test_KDT.cpp
```

- If you added any other files to src or test make sure those are included too.
- **Make sure that the “make” command works for the zip you are handing in.**
- Please note that you do not need to have your name or PID in the turn in files. In fact, by adding your name/PID to these files, they will be exposed to a 3rd party server (so feel free to remove them).

3. Be sure to test your code on ieng6. We will be grading your code using the same versions of software that are running on ieng6 and there may be issues with compilers/etc. if you only tested your code on your personal machine.

4. We are testing your code for memory leaks so be sure to run valgrind.
5. Make sure you removed all cout statements that you added as our grading scripts will fail tests if they see unexpected output.
6. If you are submitting with a partner, make sure that you two have registered that you two are working together.

You can submit as many times as you like before the deadline. By default your last submission will be used, however GradeScope allows you to manually select a final submission too. If you submit the assignment within 24 hours after the deadline (even a minute after the deadline) you will be charged a slip day. If it is more than 24 but within 48 hours, you will be charged 2 slip days. If you submitted after the deadline, but didn't get a higher score than before the deadline you can save your slip day by selecting a submission before the deadline as your final submission in GradeScope.

⚠ If you're doing pair programming Only *one* of you should make a submission. **Do not** make separate individual submissions. Make sure to [add your PA partner](#) to your *final* submission. This selection **must** match the partner you indicated in the Pair Programming sign-up form. A mismatch between our records and your gradescope submission will constitute a violation of the Academic Integrity Agreement.

Grading Breakdown:

- **Code which does not compile will not be graded.**
- 22 points for part 1 based on your implementation of BSTNode.hpp, BST.hpp, and BSTIterator.hpp
- 23 points for part 2 based on your implementation of KDT.hpp and main2.cpp
- Grading is holistic
- GradeScope will begin auto grading your code as soon as you submit and will tell you exactly how many points you scored.
- If you miss points on the checkpoint, you can gain $\frac{1}{2}$ of them back if you fix the problem in the final submission. For example, if you get 16 points on Part 1 for the checkpoint but then 22 points for the final submission, your grade for Part 1 will be $(16+22)/2$. Be careful, you can also lose points this way if you break your BST as part of building the KD Tree.

Getting Help

Tutors in the labs are there to help you debug. TA and Professor OH are dedicated to homework and/or PA conceptual questions, but they will not help with debugging (to ensure fairness and also so students have a clear space to ask conceptual questions). Questions about the intent of starter code can be posted on piazza. Please do not post your code to piazza either publicly or privately - debugging support comes from the tutors in the labs.

Format of your debugging help requests

At various times in the labs, the queue to get help can become rather long (all the more reason to start early). To ensure everyone can get help, we have a 5 minute debugging rule for tutors in that they are not supposed to spend more than 5 minutes with you before moving onto a new group. Please respect them and this rule by not begging them to stay longer as you're essentially asking them to NOT help the next group in exchange for helping you more.

5 minutes?!

Yes, 5 minutes. The job of tutors is to help you figure out the *next step in the debugging process*, not to debug for you. So this means, if you hit a segfault and wait for help from a tutor, the tutor is going to say "run your code in gdb, then run bt to get a backtrace." Then the tutor will leave as they have gotten you to the next step.

This means you should use your time with tutors effectively. Before asking for help, you will want to already have tried running your code in gdb (or valgrind, depending on the error). You should know roughly which line is causing the error and/or have a clear idea of the symptoms. When the tutor comes over, you should be able to say:

What you are trying to do. For example, "I'm working on Part 1 and am trying to get the insert method in the BST to work correctly."

What's the error. For example, "the code compiles correctly, but when I insert a child in my right subtree, it seems to lose the child who were there before."

What you've done already. For example, "I added the method which prints the whole tree (pointers and all) and you can see here <point to screen of output before and after insert> that insert to the right subtree of the root just removes what was on the right subtree previously. But looking at my code for that method, it seems like it should traverse past that old child before doing the insert. What do you suggest I try next?"

Acknowledgements

Special Thanks to Dylan McNamara, Niema Moshiri, Christine Alvarado, and Paul Kube for creating the base on which this assignment is built.