Project 5: Generic Binary Search Tree

Prabhav Gupta, Michael Rodyushkin, Archita Hothur, Rohan Bafna, Matthew Free

Fall 2023

Contents

1 Overview

In this assignment, you will be implementing a generic binary search tree (BST). When we say generic we mean that the datatype of the data in each node is not specified in advance. Using the code that you create, we should be able to create integer binary search trees that stores integers, string binary search trees that store strings, etc.

2 Binary Search Tree Implementation

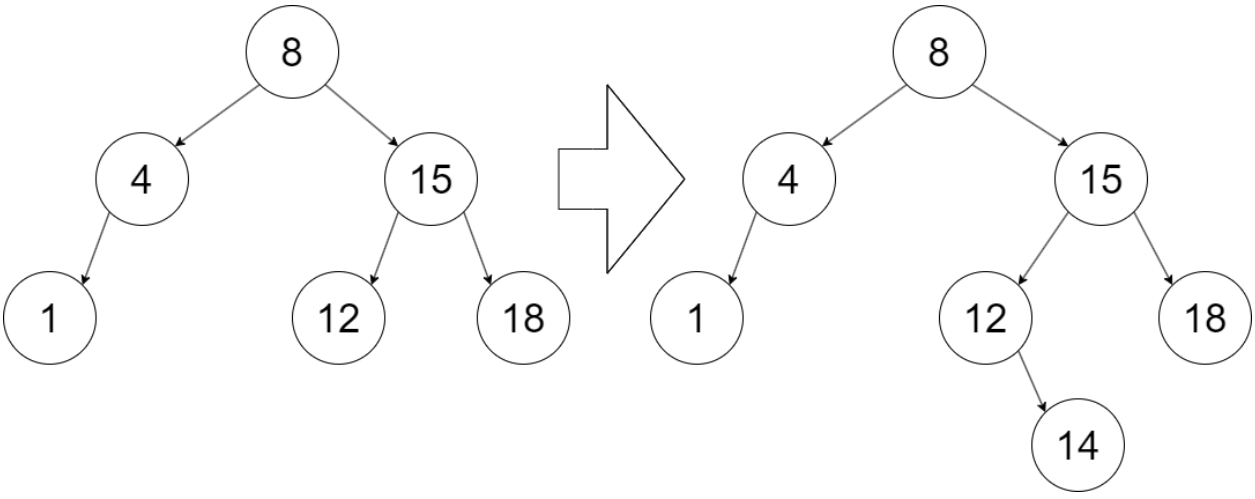
2.1 What is a binary tree?

A binary tree is a node-based data structure where each node stores a singular data value and points to two other nodes (referred to as the left and right "descendants"). Thus, because the top node of a tree directly or indirectly points to every other node in the structure (or we can say that the top node has every other node as a direct or indirect descendant of it), our tree data structure only needs to keep track of the top node (commonly referred to as the "root" of the tree) and, optionally, the number of nodes in the tree. With just this information, we should be able to reach any other node in the tree.

2.2 What is a binary search tree?

A binary search tree is a type of binary tree where data in the tree is organized in a specific way. In any given node, the left node must hold data that is less than the current node's data and the right node must hold data that is greater than the current node's data. All data must be unique, so neither the left nor the right node can be equal to the current node.
Thus, in order to add a new data value to the tree we would go through the tree starting from the root node. We move right if our new data is greater than the current node's data and left if our new data less than the current node's data. We continue this process until we reach an empty space and insert it there.

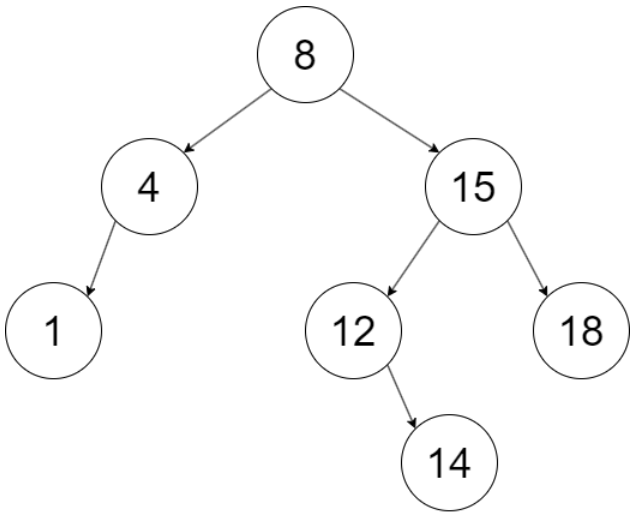
Here is a visual representation of a binary search tree and how it adds the data value 14 to it:



We traverse the tree to insert the data value 14 by starting at the root 8. First, we move to the node 15 as $14 > 8$. Second, we move to the node 12 as $14 < 15$. Fourth, we move to the right spot of 12 as $14 > 12$. Finally, because that right spot is empty, we insert the data value 14 there.

2.3 What is a preorder traversal?

There are many different ways of "traversing" or moving through and recording the nodes of a tree. For this project, you will be required to traverse through the tree in a preorder manner. What this means is you will record the current node, repeat the process on the left subtree (the tree formed by having the left node be the root node), and then repeat the process on the right subtree (the tree formed by having the right node be the root node). If a node does not exist you do not mark anything. The example below demostrates this preorder traversal.



Preorder Traversal Order: { 8, 4, 1, 15, 12, 14, 18 }

The traversal starts on the root node 8. We record 8 and move to the left subtree with 4 as the root. We record 4 and move to the left subtree formed with 1 as the root. We record 1 and because 1 has no left or right subtree, we move back to 4. 4 has no right subtree so we move back to 8. We move to 8's right subtree with 15 as the root. We record 15 and move to the left subtree with 12 as the root. We record 12 and (because there is no left subtree) move to the right subtree with 14 as the root. We record 14 and because it has no left or right subtree we move back to 12 as our root. We traversed both left and right for 12 so we move back to 15 as our root. We move to 15's right subtree with 18 as our root. We record 18 and because 18 has no left or right subtree we move back to 15. We traversed 15's left and right subtree so we move back to 8 as our root. We traversed 8's left and right subtree so we finish.

2.4 How will we implement a BST?

The visualization below demonstrates how our binary search tree is structured. Each node in the tree (of type `Node`) contains data as well as pointers to its left and right descendants, which are also of type `Node`. If a node does not have a left or a right descendant, then the corresponding pointer should be `NULL`. The tree itself is represented by a `binary_search_tree` struct, which contains a pointer to the root node, a field `num_nodes` containing the number of nodes in the tree, and three function pointers (which are explained below). You will be responsible for adding functions to perform operations on this `binary_search_tree` structure.

Our binary search tree will be generic. That means that the functions you implement will create a BST that can hold any data type; in fact, the BST should know nothing about the kind of data that it is storing. We accomplish this by dynamically allocating our data and storing pointers to the data inside each Node, instead of the data themselves. The pointer is stored in the field `data`, which has type `void *`, which indicates a pointer to an unknown type. This allows us to store data of different types in the same struct.

However, if the BST does not know the types of the data, then how is it supposed to interpret the data?

After all, our BST implementation needs to be able to

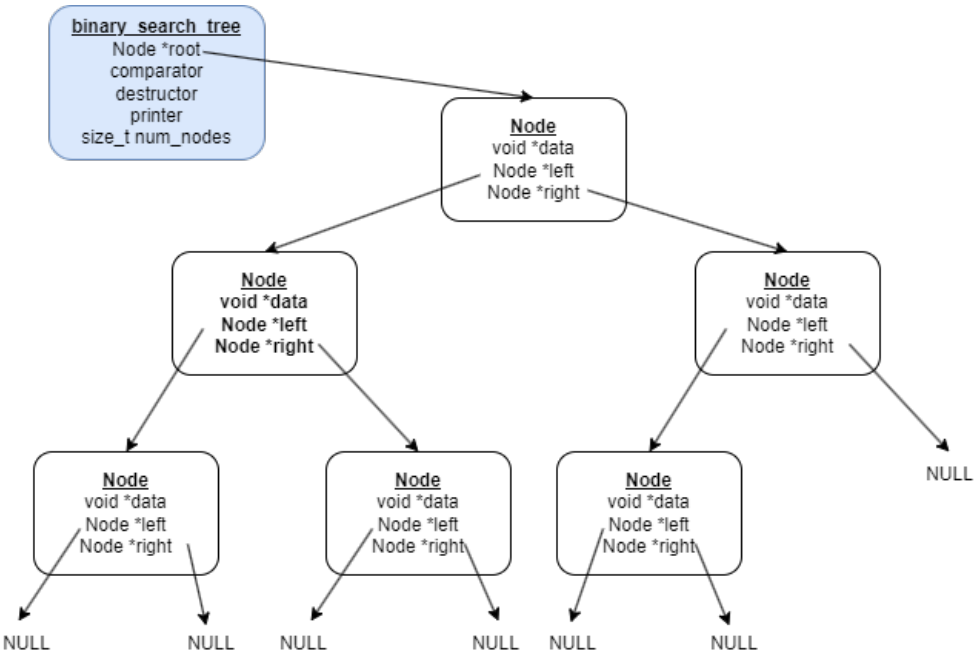
- compare two pieces of data and decide which one is less than the other, so that we can look up an item in the tree,
- free the data once the tree itself needs to be destroyed,
- and print out the data so that the tree can be inspected.

These three actions will be different for different data types. In order to make our BST generic, we need to tell the BST how to perform these actions.

We do this by creating a function for each action, and storing a pointer to that function inside the BST. When the BST needs to perform one of these actions, it can simply call the pointer to the appropriate function. There are three such function pointers in the `binary_search_tree` struct:

- `int (*comparator)(void *a, void *b)`, which compares the data `a` to `b` and returns an integer similarly to `strcmp`,
- `void (*destructor)(void *d)`, which destroys the data pointed to by `d`,
- `void (*printer)(FILE *f, void *d)`, which prints out `d` to the file `f`. (You don't need to worry too much about this function or what a `FILE *` is, as we have provided the code to print out a binary search tree for you already.)

NOTE: Any one tree will ONLY store data of a single type. We will not store data of different types in the same tree.



3 Instructions

3.1 General instructions

You have been given 4 C files: `bst.h`, `bst.c`, `data_lib.h`, `data_lib.c`.

The data_lib files contain some functions that can help you understand the assignment better. In addition, those functions can be very helpful when you are testing your code.

You should look at the `bst.h` file first. It contains the structs that you will use throughout the assignment as well as the functions that you are required to implement.

Finally, you have the `bst.c` file. This is the only file that you have to modify and submit. Here, you will implement the BST functions. Each function has a block comment that describes exactly what it should do. A synopsis is included below. **NOTE: You cannot assume data passed to a function is always valid. Our Unit Tests will test invalid arguments.**

You can use any function included in stdlib.h, string.h and stdio.h, and they are already included for you. **Including any other library is forbidden.**

You should not be leaking memory in any of the functions. For any functions utilizing malloc, if malloc returns null, return either 0 or null (based on the function header) and ensure that no memory is leaked. The autograder catches memory leaks with Valgrind. See Section 4.1 for details on Valgrind.

Be sure not to modify any other files besides bst.c. Doing so may result in you failing Gradescope test cases.

Some of the functions have pseudocode provided in **BST_Pseudocode.pdf**.

3.2 Functions you need to implement

- `binary_search_tree *init_tree(comparator, destructor, printer)`: Allocates and initializes the BST with the passed in comparator, destructor, and printer functions.
- `Node *create_node(void *data)`: Allocates and initializes a new Node with the passed in data, and return a pointer to it.
- `int insert_node(binary_search_tree *tree, void *data)`: Creates a new node given the passed in data and correctly inserts it into the passed in BST.
- `int contains(binary_search_tree *tree, void *data);`: Checks to see if a Node with the given data is in the BST.
- `void** preorder_traversal(binary_search_tree *tree)`: Allocate and initialize an array of pointers that point to the data from the given tree in preorder traversal order.
- `binary_search_tree *duplicate_without(binary_search_tree *tree, void *data_removed)`: Create a new tree that is a duplicate of the passed in tree excluding the node that has data that matches the data passed in.
- `void destroy_node(void (*destructor)(void *), Node *node, int destroy_data)`: Free the passed in node and free all nodes that are direct and indirect descendants of the node. If destroy_data is 1, invoke the destructor on every element of the tree as well.
- `void destroy_tree(binary_search_tree *tree, int destroy_data)`: Free the entire tree including all of the nodes.

3.3 Functions provided

Do not modify any function listed in this section.

- `int_comparator(const void *a, const void *b)`: Comparator for integers
- `double_comparator(const void *a, const void *b)`: Comparator for doubles
- `str_comparator(const void *a, const void *b)`: Comparator for strings
- `int_destructor(void *data)`: Destructor for integers
- `double_destructor(void *data)`: Destructor for doubles
- `str_destructor(void *data)`: Destructor for strings
- `int_printer(void *data)`: Printer for integers
- `double_printer(void *data)`: Printer for doubles
- `str_printer(void *data)`: Printer for strings
- `void print_tree(FILE *f, binary_search_tree *tree)`: Prints a given tree in a formatted fashion to the file `f`.

  If you want to print to the screen, pass in `stdout` as the first parameter to `print_tree`. `stdout` is a variable of type `FILE *` declared in `stdio.h` and represents the computer screen.
- `void print_helper(FILE *f, binary_search_tree *tree, Node *node, int level)`: used to help print_tree print the tree.

4 Building & Testing

All of the commands below should be executed in your Docker container terminal, in the same directory as your project files.

4.1 Helpful Info

1. **For issues with Docker** please refer to the 2110 Docker Guide for C
2. **Use Docker's "Interactive Terminal"** Run the following command in your terminal to access Docker's terminal much easier: `./cs2110docker-c.sh` on Linux/Mac and `./cs2110docker-c.bat` on Windows.
3. From within the "Interactive Terminal" you should notice the "host/" directory when you type `ls`. Navigate to your Project directory and run the unit tests using the commands mentioned later.
4. To exit Docker's "Interactive Terminal" simply type: **exit**.
5. **make** is a program to help you build your project (in other words, it helps you compile).
6. **GDB** is a very useful debugger however, it may be a bit confusing at first. Please refer to the following GDB Cheatsheet or ask a TA for help!

4.2 Unit Tests

These are the same tests that will run on Gradescope

To run the autograder locally (without GDB):

1. **make clean** - Clean your working directory
2. **make tests** - Compile all the required files
3. **./tests** - Run the unit tests
4. **make** - Compile all the required files and Run the unit tests (previous two steps with one command)

Executing **./tests** will run all the test cases and print out a percentage, along with details of the **failed test cases**.
Other available commands (after running make tests):

- To run a specific test case (to avoid all printing output/debug messages for all test cases):

  ```
  make run-case TEST=testCaseName

  Example: make run-case TESTS=test_compare
  ```

- To run a test case with gdb:

  ```
  make run-gdb TEST=testCaseName (or no testCase to run all in gdb)
  ```

4.3 Write Your Own Tests

In your Project 5 `main.c` file there is a `main()` function that can be executed with the following command.

make student

For example, if you want to write a test for your `init_tree` and `insert_node` functions, you could write the following function in `main.c`, and then run `make student` inside Docker.

```
int main(void) {
    // Add your own test cases here to test your functions :)

    // Initialize integer binary search tree
    binary_search_tree *tree = init_tree(int_comparator, int_destructor, int_printer);
    printf(
        "Initialize BST - Expected num_nodes value: %d, Actual num_nodes value: %d\n",
        0, (int) tree->num_nodes);

    //Insert new node
    int* data = (int*) malloc(sizeof(int));
    *data = 5;
    int result = insert_node(tree, data);
    // Verify the node was added
    printf("Inserting node in BST - Expected Return Value: 1, Actual: %d\n",
```

```
            result);
    if (result == 0) {
        printf("Node not inserted after insert. Exiting test.\n");
        return 1;
    }
    // Verify the newly added node is the head of the BST without any descendents
    printf("Inserting node in BST - Expected head data value: %d, Actual: %d\n",
    5, *(int*) tree->root->data);
    printf("Inserting node in BST - Expected left pointer: %d, Actual: %d\n",
    0, tree->root->left != NULL);
    printf("Inserting node in BST - Expected right pointer: %d, Actual: %d\n",
    0, tree->root->right != NULL);

    printf("\nThe BST after insertion:\n");
    // Print the tree to the screen
    print_tree(stdout, tree);

    destroy_tree(tree, 1);
    return 0;
}
```

Then, assuming your functions are correct, you should receive an output similar to this:

```
Initialize BST - Expected num_nodes value: 0, Actual num_nodes value: 0
Inserting node in BST - Expected Return Value: 1, Actual: 1
Inserting node in BST - Expected head data value: 5, Actual: 5
Inserting node in BST - Expected left pointer: 0, Actual: 0
Inserting node in BST - Expected right pointer: 0, Actual: 0

The current BST:
Root: 5
```

5 Deliverables

Turn in the file `bst.c` to Gradescope by the due date.

Please do not wait until the last minute to run/test your project, history has proved that last minute turn-ins will result in long queue times for grading on Gradescope. You have been warned!!!

6 Rules and Regulations

6.1 General Rules

1. Although you may ask TAs for clarification, you are ultimately responsible for what you submit.
2. Please read the assignment in its entirety before asking questions.
3. Please start assignments early, and ask for help early. Do not email us a few hours before the assignment is due with questions.
4. If you find any problems with the assignment, it would be greatly appreciated if you reported them to the author (which can be found at the top of the assignment). Announcements will be posted if the assignment changes.

6.2 Submission Conventions

1. In order to submit your assignment, submit the files individually to the Gradescope assignment.
2. Do not submit links to files. The autograder does not understand it, and we will not manually grade assignments submitted this way as it is easy to change the files after the submission period ends.

6.3 Submission Guidelines

1. You are responsible for turning in assignments on time. This includes allowing for unforeseen circumstances. If you have an emergency let us know **IN ADVANCE** of the due time supplying documentation (i.e. note from the dean, doctor's note, etc). Extensions will only be granted to those who contact us in advance of the deadline and no extensions will be made after the due date.
2. You are also responsible for ensuring that what you turned in is what you meant to turn in. After submitting you should be sure to download your submission into a brand new folder and test if it works. No excuses if you submit the wrong files, what you turn in is what we grade. In addition, your assignment must be turned in via Gradescope. Under no circumstances whatsoever we will accept any email submission of an assignment. Note: if you were granted an extension you will still turn in the assignment over Gradescope.
3. Projects turned in late receive partial credit within the first 48 hours, as defined in the syllabus. Between 0 and 24 hours late, you can receive a maximum score of 70%. Between 24 and 48 hours late, you can receive a maximum score of 50%. We will not accept projects turned in over 48 hours late.
4. You are solely responsible for submitting your project before it is due. Last-minute technical difficulties (Canvas/Gradescope crashing, bad internet, etc.) are NOT valid excuses.

6.4 Is collaboration allowed?

From the syllabus:

- You must submit an assignment or project as your own work. **No collaboration on answers is permitted. Absolutely no code or answers may be copied from others. Such copying is Academic Misconduct.**
- Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "submission of material that is wholly or substantially identical to that created or published by another person or persons").
- Publishing your assignments on public repositories, github, etc, that is accessible to other students is unauthorized collaboration and thus Academic Misconduct.

6.5 Coding Guidelines C/C++ code

1. You must turn in ALL files specified in the "Deliverables" section of the assignment instructions. We reserve the right to impose a penalty on submissions that do not follow the given submission directions.
2. You must provide a Makefile that compiles and links your code by default. If you are given a Makefile with the project, we expect your code to compile with make. (don't worry, we'll explain what all this means later)
3. Your code must compile with gcc on Ubuntu 22.04 LTS. If your code does not compile, you will receive a 0 for the assignment.

4. You will be penalized if your code produces warnings when compiled with the given Makefile, or the following flags if no Makefile is provided: gcc -Wall -Wextra -Wstrict-prototypes -pedantic -O2

5. Code should be well commented and use a clean, consistent (readable) style (i.e., proper indenting, etc.). We reserve the right to impose style requirements, and deduct for non-conforming solutions. This is not the obfuscated C code competition!

6.6 Syllabus Excerpt on Academic Misconduct

The goal of all assignments in CS 2110 is for you to learn. Learning requires thought and hard work. Copying answers thus prevents learning. More importantly, it gives an unfair advantage over students who do the work and follow the rules.

1. **As a Georgia Tech student, you have read and agreed to the Georgia Tech Academic Honor Code.** The Academic Honor Code defines Academic Misconduct as "***any act that does or could improperly distort Student grades or other Student academic records.***"

2. You must submit an assignment or project as your own work. **No collaboration on answers is permitted. Absolutely no code or answers may be copied from others. Such copying is Academic Misconduct**.

3. Using code from GitHub, via Googling, from Stack Overflow, etc., is Academic Misconduct (Honor Code: Academic Misconduct includes "*submission of material that is wholly or substantially identical to that created or published by another person or persons*").

4. Publishing your assignments on public repositories, github, etc, that is accessible to other students is unauthorized collaboration and thus Academic Misconduct.

5. Suspected Academic Misconduct will be reported to the Division of Student Life Office of Student Integrity. It will be prosecuted to the full extent of Institute policies.

6. Students suspected of Academic Misconduct are informed **at the end of the semester**. Suspects receive an *Incomplete* final grade until the issue is resolved.

7. We use accepted forensic techniques to determine whether there is copying of a coding assignment.

8. Submitting an assignment with code or text from an AI assistant (e.g., ChatGPT) is academic misconduct.

9. **If you are not sure about any aspect of this policy, please ask your lecturer**.

Using AI Assistants

Anything you did not write in your assignment will be treated as an academic misconduct case. If you are unsure where the line is between collaborating with AI and copying AI, we recommend the following heuristics:

Heuristic 1: Never hit "Copy" within your conversation with an AI assistant. You can copy your own work into your own conversation, but do not copy anything from the conversation back into your assignment. Instead, use your interaction with the AI assistant as a learning experience, then let your assignment reflect your improved understanding.

Heuristic 2: Do not have your assignment and the AI agent open at the same time. Similar to the above, use your conversation with the AI as a learning experience, then close the interaction down, open

your assignment, and let your assignment reflect your revised knowledge. This heuristic includes avoiding using AI directly integrated into your composition environment: just as you should not let a classmate write content or code directly into your submission, so also you should avoid using tools that directly add content to your submission.

Deviating from these heuristics does not automatically qualify as academic misconduct; however, following these heuristics essentially guarantees your collaboration will not cross the line into misconduct.