SP(GA) - Assessed Exercise #1

Dr. Yehia Elkhatib

Due: December 8^{th} 2024 at 2200hrs **This is an already extended deadline**

1 High-Level Specification

In this assessed exercise, you are asked to implement a binary search tree[1] in C. You are to:

- Define a new type, `node_t`, that represents a `struct` containing everything needed for the node of a binary search tree. Its members should be named exactly as follows: `value`, `left`, and `right`. If needed, any additional member can be named as you see fit.
- Implement a `node_t * search(node_t * node, int toFind)` function that returns a pointer to the node with the variable `toFind` if one exists in the tree, or `NULL` otherwise.
- Implement a `node_t * createTree(int firstElem)` function that allocates a new tree and returns a pointer to its root, or `NULL` if unsuccessful.
- Implement a `void destroyTree(node_t * node)` function that takes a tree's root and de-allocates the entire tree.
- Implement a `void insert(node_t * node, int elem)` function that inserts `elem` at the appropriate place in the tree. Duplicates should NOT be inserted!
- Implement a `void delete(node_t * node, int elem)` function that finds and deletes `elem` from the tree.
- Ensure that your implementation is memory-safe, i.e. containing no memory leaks and avoids segmentation faults.
- Write a short report about the state of your implementation.

The above specification is summarised in the following signatures which **your code will be marked against**. Testing your code in this manner ensures that you learn to write code that integrates reliably with larger systems, an essential software engineering principle called contract-based programming.

```c
typedef struct {
    // definition to go here
} node_t;

node_t * search(node_t * node, int toFind);
node_t * createTree(int firstElem);
void destroyTree(node_t * node);
void insert(node_t * node, int elem);
void delete(node_t * node, int elem);
```

2 Submission

You are to submit **one archive file** (zip, tar, etc.) containing **only 2 files**: `bst.c` and `report.pdf`. Any other files will be ignored. Submit through Moodle.

[1] Section 4 of this document will explain how a binary search tree works.

3 Marking Scheme

The coursework will be marked out of 100, with the following distribution by section.

Deliverable	*Maximum Marks*
Tree initialiser	15
Insert function	15
Delete function	20
Search function	15
Tree destroyer	15
Report	20

4 Binary Search Trees

4.1 Definition

Binary search trees are linked data structures that automatically sort data as they are input. They are effectively binary trees with rules imposed on them that dictate the overall structure of the tree.

First, a binary search tree is a tree. It is a directed graph in which each node has one parent and a unique set of children, similar to the graph illustrated in Figure 1. We store a value at each node.

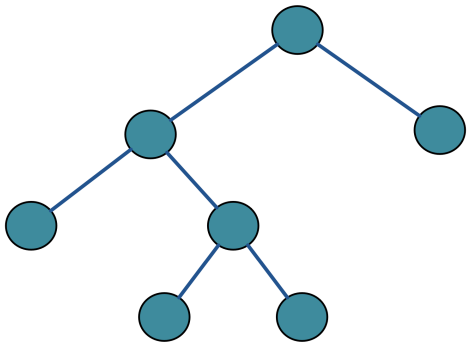

Figure 1: An example of the structure of a binary tree. (Image credit to Tmigler on wikipedia, CC BY-SA 4.0.)

A "binary tree" is a tree where every node has at most two children (e.g., Figure 1). This means that, when we navigate the tree, we can go "left" or "right", assuming a child exists in that direction. When a node is at the end of a tree, it has no children. We call such a node a "leaf". Each node in the binary search tree is itself the root of the nodes descending from it, making the node and its descendants a *subtree*.

A *binary search tree* is a binary tree with additional rules about *where* information is put when added / deleted from the tree. Specifically, when looking at any node, everything to that node's left is lower than it, and everything to the node's right is higher than it. For example, see Figure 2.

4.2 Construction

We begin making a binary search tree by producing a root node that contains any element. We then insert future nodes into the tree by comparing the element we want to insert with the root node. If the element we want to insert is higher than the root element, we move to the root node's right child; otherwise we move to the left. We continue to move down the tree by making this comparison for every future node until we descend to a leaf node,

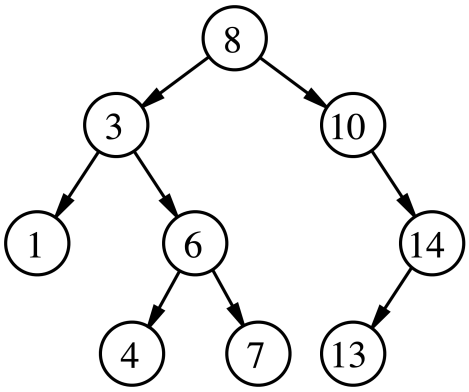

Figure 2: A binary search tree. Note that if you pick any node, everything descending from the left child is less than that node, and those on the right are higher. (Image credit to Dcoetzee on wikipedia.)

at which point a new node can be inserted as a left or right child as appropriate. Following this procedure means that we always end up with a binary tree that obeys the properties we laid out earlier.

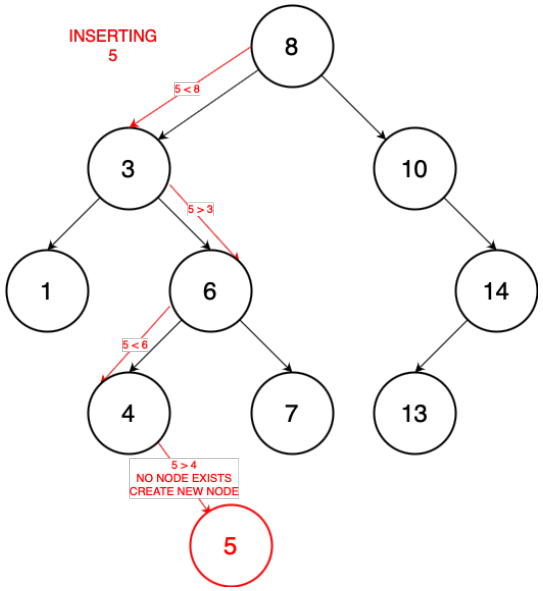

Figure 3: To insert 5, the comparisons we make are highlighted in red. They are, in order: $5 < 8$, $5 > 3$, $5 < 6$, $5 > 4$.

There are similar rules for deletion from the tree, with an additional caveat that you need to reconnect the remaining subtrees to form a valid BST. There are 3 cases:

1. If the node to be deleted is a leaf, there is nothing else to do.
2. If the node to be deleted has exactly one child, that child will move up to replace its parent.
3. If the node to be deleted has two children, you need to find the *successor*, i.e. the value that would replace the one to be deleted. This is typically either the largest value in the left subtree or the smallest value in the right subtree, depending on which fits without reshuffling the entire tree structure.

5 Implementation Dos and Don'ts

► You must submit your code as **a single C file** named `bst.c` [Penalty: 10%]
► Your code must follow the API given in Section 1; i.e., your functions need to have the specified signatures (name, parameter types and numbers, and return type). [Penalty: 10% per violation]
► Your implementation must compile and run on clang v12 or later. If you use a specific C standard when compiling your code, **make sure you mention this**. [Penalty: up to 50%]
► Your file should **not** include a `main()` function. Naturally, you might use one while testing your code. If so, remove it once you are ready to submit your code. [Penalty: 5%]
► You must remove all print-to-screen statements (e.g., `printf`) before submitting. [Penalty: 5% per statement]

6 Report

You are to submit a short report that includes your name and GUID. You should submit it as **PDF**. [Penalty: 5%]

The report will summarise how your code works and the rationale for your approach. It should also *honestly* state whether your attempt works for each expected functionality. In particular, you should include a sentence or two for each of the following topics with clear headings:

- **Search** – An explanation of how your **search** function works.
- **Generation** – An explanation of how you **create** and **destroy** the tree.
- **Maintenance** – An explanation of how your **insertion** and **deletion** functions work.
- **Memory Management** – An explanation of how your functions are memory-safe, and what issues pertaining to manual memory management do and do not require mitigation in the functions you implemented.

If you do not manage to accomplish any particular section, please state this and describe instead how you *would* approach the problem were you to make an attempt, or what attempts you made, and where they failed.

The length of your report should not exceed 1 page with reasonable font size ($\geq 10pt$) and margins ($\geq 2cm$).

7 FAQs

- *Does the tree need to be balanced?*
  No.
- *Can I add helper methods?*
  Yes, as long as you do not change the function signatures laid out in Section 1.
- *Can my code be iterative? / Does it need to be recursive?*
  Both approaches are acceptable.
- *Will my code be marked on X? (e.g. X=time complexity, comments, ...etc.)*
  Please refer to the marking scheme in Section 3.

Good luck!