Assignment 4:
Binary Search Trees

Part 1: Programming

For this assignment, you will implement a binary search tree that can store any arbitrary struct in its nodes. You will start by completing the *recursive implementation* of the binary search tree (BST) in Worksheet 29. You will then modify it to so it can store arbitrary structures at each node, provided you have a an implementation of a compare and print_type function for that structure.

We are providing you with the following files:

- bst.c - This is the file in which you'll finish implementing the unfinished BSTree implementation. There is a main function in this file that you should modify to exercise your BST. The file contains several test cases such as **testAddNode**, **testContainsBSTree**, **testLeftMost**, **testRemoveLeftMost**, and **testRemoveNode**. Your implementation must pass all these test cases, and you are strongly encouraged to add your own tests as well.
- bst.h - This file should not be changed.
- structs.h - This file can be extended to test your code with different data types.
- compare.c- Put your compare and print functions in here.
- makefile.txt - Remember to rename this file to **makefile** (without the .txt extension).

Worksheet 29 will get you started on your implementation. However, there is one function not mentioned in the worksheet. You will be using the **compare** function to test two values of a node to determine if one is less than, greater than, or equal to the other. This function is similar to the **compareTo** function in the **Comparable** interface in Java. Rather than embedding it into the data structure, as you would do in Java, we will declare it and assume that the user has provided an implementation of **compare** in the file **compare.c**. That way, the user can substitute an appropriate compare function for any data type that they plan to store in the tree.

For example, if you want to store doubles in your tree, you might define the following struct to store at each node:

```
struct data {
      double num;
}
```

And then define your **compare** function to simply compare the two structs based on the **num** field. However, a user of your data structure could also do the following:

```
struct pizza {
      double cost;
      int numToppings;
      char *name;
}
```

And define a **compare** function that compares pizzas based on their name, cost, or number of toppings.

In this assignment, the **TYPE** macro is set to **void***. This means that the type of value stored in a node is a *void pointer*, which means it can be a "pointer to anything". Whenever you dereference a void pointer, you *must* cast it to a specific type. For example, you can cast a void pointer to **struct data*** (see the definition in **structs.h**), then dereference it to get the **struct data** value the pointer points to. It is the programmer's responsibility to ensure that they cast the void pointer to the same type of value that was actually stored at that location. You should read up on void pointers in your C reference or on the internet.

The **compare** function is needed because we need some way to compare the values stored in the tree nodes. Note that we can't just compare the pointers with the >, <, or == operations since this would just compare the memory addresses the pointers point to. Instead, we want to compare some field of the struct that the pointer points to (e.g. **val->number < otherVal->number**). The **compare** function will make changing this function for different structs much easier.

Finally, I strongly recommend that you add to the **main** function to exercise your binary search tree by adding, removing, and testing for elements.

Scoring (80 pts)

- struct Node *_addNode(struct Node *cur, TYPE val) (20 pts)
- int containsBSTree(struct BSTree *tree, TYPE val) (15 pts)
- TYPE _leftMost(struct Node *cur) (10 pts)
- struct Node *_removeLeftMost(struct Node *cur) (10 pts)
- struct Node *_removeNode(struct Node *cur, TYPE val) 20 pts)
- compare.c (5 pts)

Part 2: Written Questions

In addition to the programming portion of the assignment you will also be answering some questions about binary search trees. Some of the questions will require you to write your answers on the tree in empty_graph.pdf. We will assume that any empty node boxes are non-existent nodes. For each of these problems, print out a copy of the blank tree, fill in the answers for the question, and please make sure to **WRITE THE QUESTION NUMBER** and your name on the sheet. If you know how to annotate PDF files directly, you can also do that instead of printing off the trees (this may be easier since you will have to submit a digital version of your solutions).

Question 1
Show the binary search tree built by adding numbers in this specific order, assuming the graph is empty to start with: 50, 16, 90, 14, 32, 71, 42, 5 (You may need to add more boxes to the diagram).

Question 2
The trouble with binary search trees is that they can become unbalanced depending on the order that you insert values. Give an order for inserting the numbers 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13 such that the resulting tree is a complete binary search tree. Please make that your intermediate trees are also complete binary search trees as well. (Hint: it might be helpful to first draw the full/complete tree to figure out how the values must be arranged, then you can determine the order to add them.)

Question 3

- Part A: Given the following tree, question3.pdf, show the tree after removing the value 40.
- Part B: Using the tree produced by Part A, show the tree after removing the value 16.

Question 4
The computer has built the following decision tree for the Guess the Animal Game, question4.pdf. The player has an animal in mind and will answer the questions shown in the tree. Each of the player's responses is used to determine the next question to ask. For example, if the player is thinking of a sea turtle, she would answer Yes to the first (top) question, "does it live in the water?", which leads to the second question "is it a mammal?", to which she would answer No.

Show the decision tree that the computer should build after adding a Zergling and a question to differentiate it, "Does it eat space marines?", to the tree. The question and the animal should be added below existing questions in the tree. Note that Zerglings *do* eat space marines, *do not* live in the water, *do not* climb trees, and *are not* mammals (just in case you didn't know :-))

Scoring (20 pts)

- Question 1 (5 pts)
- Question 2 (5 pts)
- Question 3 (5 pts)
- Question 4 (5 pts)

What to Turn In

Turn in the following files to both TEACH and Canvas:

1. bst.c
2. compare.c
3. structs.h (if you have changed it)
4. answers.pdf (if you printed off the trees, please scan your answers)