From CS261

Main: ProgrammingAssignment4

Binary Search Trees

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) (See worksheet 29). You will then modify it and add functionality for storing arbitrary structures by implementing a `compare` and `print_type` function.

We are providing you with the following files:

- **bst.c** is the file in which you'll finish implementing the unfinished BSTree.
- **bst.h** should not be changed.
- **structs.h** you can change to test your code with different data types
- **compare.c** put your compare and print function in here
- **main.c** you should modify to thoroughly exercise your BST. This is just a starter.
- **makefile** Remember to 'save as' makefile (without the .txt extension)

Worksheet 29 will get you started on these functions, 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 embed it in the data structure, we will declare it and assume that the user has provided an implementation of compare in a file called compare.c. That way, the user can substitute an appropriate compare 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 data {
double num;
}
```

and then define your compare to simply compare based on num . However, a user of your data structure could also do the following:

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

and define a compare that compares pizzas based on name or cost or numToppings .

Your `TYPE` variable in this assignment is of type void*, giving you the freedom of type casting it into any desired type. For example, you can type cast it into struct data* (find the definition in structs.h) --notice that it is a pointer. `compare()` is necessary because you do not want to compare pointers with the >, <, or == comparisons. These would be useless since a pointer is just a memory address. Instead, you want to compare some field of the type (e.g. val->number < otherVal->number). `compare()` will make this cleaner for you. Finally, we strongly recommend that you build your own `main` function to exercise your binary search tree by adding, removing, and testing for elements. We have provided an example starter `main.c`.

Questions

In addition to the programming portion you will also be answering some questions about binary search trees. All the questions will require you to write your answers on the **empty_graph.pdf**. We will assume that empty node boxes are non-existent nodes. Print those out, fill in the tree with the answer, and WRITE THE PROBLEM NUMBER... Your name would also be helpful.

Question 1

Show the binary search tree built by adding numbers in this specific order, the graph is empty to start with: 9, 4, 8, 12, 15, 13.

Question 2

The trouble with binary search trees is they can become unbalanced depending on the order you insert values in. Give an order for inserting the values 1 through 7 such that the tree is completely balanced. Balanced means that the depth of all the branches is the smallest they could be for a tree that fits seven nodes. This problem does not require you to fill in a tree, just write down the order in which you would insert the values.

Question 3

- Part A: Given the following tree, **question3.pdf**, show the tree after removing the value 50.
- Part B: Using the tree produced by Part A, show the tree after removing the value 42.

Question 4

A user is given the following decision tree, **question4.pdf**. They have an animal in mind and will answer the questions shown in the tree. Each of their responses is used to determine the next question to ask. For example, if they are thinking of a sea turtle, they 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 they would answer No.

Show the decision tree 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 eat space marines, do not live in the water, do not climb trees, and are not mammals.

Extra Credit 1

In question 4 above, we explore the idea of using decision trees to store a database of questions and animals for playing the "Guess the Animal" game. The game proceeds by asking users a list of YES/NO questions to determine the animal users are thinking about. A sample session of the game is shown below:

```
Beginning animal game...
Please think of an animal!
> Does it live in the water?
Yes
> Is it a mammal?
No
> Is it a sea turtle?
Yes
That was great! You were thinking of a sea turtle.
```

For this extra credit question, you are asked to implement your own "Guess the Animal" program. Note that we intentionally leave out several details to give you the freedom of creatively building your program. Still, following are some hints for you to begin with.

- Your program should make use of the decision tree structure as in question 4 above. How do you represent your decision tree? What kind of node would be a question? What kind of node would be an animal?
- Use a database file to store your list of questions and animals. Read your database into the decision tree every time your program begins.
- What would you do if your program does not know the animal users think of? Should it be able to learn new animals? If yes, then you should be able to update your database file.
- You can go to the website **http://www.animalgame.com/** to try out an implementation of this game.

Submit an `animal.zip` file containing all the files in your project.

Extra Credit 2

For extra credit, you can compare your recursive implementation of the BST to an iterative version. In particular, we're interested in seeing how they compare in terms of speed. First, create an iterative implementation, compile, and test it to verify that it works properly. You are welcome to use a version that either you write yourself or that you find (and understand) online.

Now, write a program to empirically compare the time required to insert and remove a large number of elements with your BST to the iterative BST.

The first tool we will need is the ability to discover how much execution time (wall clock time) is being spent in a segment of code. The function we will use for this purpose is named clock. To use this function you need to include <time.h>. Try the following sample program to illustrate how this works.

```
#include <stdio.h>
#include <time.h>

double getMilliseconds() {
   return 1000.0 * clock() / CLOCKS_PER_SEC;
}

   int main() {
      int i;
      double j;
```

```
        double elapsed;

        elapsed = -getMilliseconds();
        j = 0;
        for(i = 0; i < 100000000; ++i)
            j = j + i / 3.0; /* nonsense computation that just takes time */

        elapsed += getMilliseconds();

        printf("Elapsed Milliseconds = %lf\n", elapsed);

        return 0;
    }
```

The helper function getMilliseconds returns the number of milliseconds since the program began executing. It uses the clock function to get the number of clocks and a constant to convert clocks to actual time. In the main function, the time it takes to run the loop is: elapsedTime = endTime - beginTime. Note how the variable elapsed is used twice to record the elapsedTime. You can play around with the upper limit on the loop shown here to see the effect on execution time.

Using the millisecond timer, you will print the time required to run your BST test on a large data set and compare the performance to that of the iterative BST.

To produce a random value you can use the function rand(), which is found in the library given by the header file stdlib.h. rand() will produce the same sequence over and over. To produce truly random values you can reset the random number generator using the time function, as in

srand(time());

However, to compare two approaches fairly, you should use the same seed and thus generate the same sequence of random numbers

Answer the following questions and submit all the answers in a text file named `iterativeBST.txt`

1. In a couple of paragraphs, explain your strategy for comparing the timing of the two approaches. How many elements did you insert? How many times did you perform the simulation?
2. Which BST is faster and why do you think this is the case?
3. Which BST takes up less memory? Explain why.

Grading Rubric

- Compile/Style = 20
- struct Node *_addNode(struct Node *cur, TYPE val) = 15
- int containsBSTree(struct BSTree *tree, TYPE val) = 10
- TYPE _leftMost(struct Node *cur) = 10
- struct Node *_removeLeftMost(struct Node *cur) = 10
- struct Node *_removeNode(struct Node *cur, TYPE val) = 15

Questions

- Ques 1 = 5
- Ques 2 = 5
- Ques 3 = 5
- Ques 4 = 5

Extra Credit 1

- Guess the Animal program = 15

Extra Credit 2

- Ques 1 = 2
- Ques 2 = 2
- Ques 3 = 2

What to turn in via TEACH

- bst.c
- answers.txt (or .pdf) (for the four questions. Please scan in your answers to turn them in electronically and also turn in a hardcopy during recitation this week)
- animal.zip (if you did the extra credit 1)
- iterativeBST.txt (if you did the extra credit 2)
