CSCI 1120 (Low-Level Computing), Fall 2019

Homework 9

Credit: 20 points.

1 Reading

Be sure you have read, or at least skimmed, the assigned readings for classes through 11/13.

2 Programming Problems

Do the following programming problems. You will end up with at least one code file per problem. Submit your program source (and any other needed files) by sending mail to `bmassing@cs.trinity.edu` with each file as an attachment. Please use a subject line that mentions the course and the assignment (e.g., "csci 1120 hw 9" or "LL hw 9"). You can develop your programs on any system that provides the needed functionality, but I will test them on one of the department's Linux machines, so you should probably make sure they work in that environment before turning them in.

1. (20 points) Your mission for this assignment is to complete a partial implementation in C of a binary search tree (a.k.a. sorted binary tree) of `int`s. (I'm hoping that all of you know about this data structure from CS2 or another course. If not, the Wikipedia article is a reasonable description (but I recommend that you not read the example code until you try to write your own).

   This partial implementation consists of a number of files:

   - Function declarations for tree: int-bst.h.
   - Starter file for function definitions: int-bst.c.
   - Test program and supporting files: test-int-bst.c, test-helper.c, test-helper.h.
   - Makefile for compiling (comments in the file tell you how to use it): Makefile.

   I've made a ZIP file containing all of these files, so it will probably be simplest just to download that and unzip it (command `unzip` on our machines). If you prefer to download individual files, *NOTE* that you should use your browser's "download" or "save" function to obtain the Makefile rather than copying and pasting text. This is because copy-and-paste will likely replace the tab characters in the file with spaces, with bad consequences (since tabs are semantically significant in makefiles.)

   Your job is to modify the file `int-bst.c` so it includes function definitions for all the functions declared in `int-bst.h`. The test program is self-contained and contains code to call the functions you will write, so you don't need to write any input/output code, aside from implementing two print functions. You compile the test program by typing `make test-int-bst` and run it by typing `./test-int-bst`.

Note that the function that removes a single element of the tree (`int_bst_remove`) is optional; you can provide an "implementation" that just prints an error message, or for extra credit you can actually implement this operation.

You should not modify any other files, unless you want to add additional tests to `test-int-bst.c`.

Sample output of the test program:

```
inserting 40 into tree [ ]
result [ 40 ]
inserting 30 into tree [ 40 ]
result [ 30 40 ]
inserting 50 into tree [ 30 40 ]
result [ 30 40 50 ]
inserting 20 into tree [ 30 40 50 ]
result [ 20 30 40 50 ]
inserting 60 into tree [ 20 30 40 50 ]
result [ 20 30 40 50 60 ]
inserting 16 into tree [ 20 30 40 50 60 ]
result [ 16 20 30 40 50 60 ]
inserting 14 into tree [ 16 20 30 40 50 60 ]
result [ 14 16 20 30 40 50 60 ]
inserting 18 into tree [ 14 16 20 30 40 50 60 ]
result [ 14 16 18 20 30 40 50 60 ]
inserting 24 into tree [ 14 16 18 20 30 40 50 60 ]
result [ 14 16 18 20 24 30 40 50 60 ]
inserting 56 into tree [ 14 16 18 20 24 30 40 50 60 ]
result [ 14 16 18 20 24 30 40 50 56 60 ]
inserting 64 into tree [ 14 16 18 20 24 30 40 50 56 60 ]
result [ 14 16 18 20 24 30 40 50 56 60 64 ]
inserting 30 into tree [ 14 16 18 20 24 30 40 50 56 60 64 ]
result [ 14 16 18 20 24 30 40 50 56 60 64 ]
inserting 50 into tree [ 14 16 18 20 24 30 40 50 56 60 64 ]
result [ 14 16 18 20 24 30 40 50 56 60 64 ]
test data in order [ 14 16 18 20 24 30 30 40 50 50 56 60 64 ]
40
  30
    20
      16
        14
          .
          .
        18
          .
          .
      24
        .
        .
    .
  50
```

```
    .
    60
      56
        .
        .
      64
        .
        .
finding 0 in tree [ 14 16 18 20 24 30 40 50 56 60 64 ]
result false
finding 100 in tree [ 14 16 18 20 24 30 40 50 56 60 64 ]
result false
finding 10 in tree [ 14 16 18 20 24 30 40 50 56 60 64 ]
result false
finding 40 in tree [ 14 16 18 20 24 30 40 50 56 60 64 ]
result true
finding 14 in tree [ 14 16 18 20 24 30 40 50 56 60 64 ]
result true
finding 64 in tree [ 14 16 18 20 24 30 40 50 56 60 64 ]
result true
removing 0 from tree [ 14 16 18 20 24 30 40 50 56 60 64 ]
result [ 14 16 18 20 24 30 40 50 56 60 64 ]
removing 16 from tree [ 14 16 18 20 24 30 40 50 56 60 64 ]
result [ 14 18 20 24 30 40 50 56 60 64 ]
removing 60 from tree [ 14 18 20 24 30 40 50 56 60 64 ]
result [ 14 18 20 24 30 40 50 56 64 ]
removing 30 from tree [ 14 18 20 24 30 40 50 56 64 ]
result [ 14 18 20 24 40 50 56 64 ]
removing 50 from tree [ 14 18 20 24 40 50 56 64 ]
result [ 14 18 20 24 40 56 64 ]
40
  20
    14
      .
      18
        .
        .
    24
      .
      .
  64
    56
      .
      .
    .
inserting 0 into tree [ 14 18 20 24 40 56 64 ]
result [ 0 14 18 20 24 40 56 64 ]
inserting 100 into tree [ 0 14 18 20 24 40 56 64 ]
```

```
result [ 0 14 18 20 24 40 56 64 100 ]
inserting 0 into tree [ 0 14 18 20 24 40 56 64 100 ]
result [ 0 14 18 20 24 40 56 64 100 ]
inserting 100 into tree [ 0 14 18 20 24 40 56 64 100 ]
result [ 0 14 18 20 24 40 56 64 100 ]
after removing all elements [ ]
```

Output of the "print as tree" function is a bit obscure. Described recursively, it works as follows:

- For an empty tree, print ".".
- For a non-empty tree, first print its root and then print its two subtrees, each indented two spaces.

Partially annotated output:

```
40 <-- root
  30 <-- left child of 40
    20 <-- left child of 30
      16
        14
          .
          .
        18
          .
          .
      24
        .
        .
    . <-- right child of 30 (empty)
  50 <-- right child of 40
    . <-- left child of 50 (empty)
    60 <-- right child of 60
      56
        .
        .
      64
        .
        .
```

It may be worth noting that my code for "remove" does something a bit tricky that I don't really expect yours to do: For nodes with two children, one can replace the node with either the largest element to its left or the smallest to the right; my code alternates between the two.

Hints:

- There are several functions you need to write — the ones declared in `int-bst.h`. You might start by just writing stub versions that return something (anything) if they need to and otherwise do nothing; then you can compile and try the test program. It won't

do anything very meaningful, but at least you can check that you know how to compile and run it. Then start filling in the functions one at a time, checking that each works or at least compiles before going on to the next.

- You may want to add additional "helper" functions, but if so they should go only in `int-bst.c`. (This might be particularly useful for the "print as tree" function.)
- You may find it helpful to look more closely at the sorted-list example shown in the video lectures and available on the course "sample programs" page; it's meant to be a model for one way to implement a linked data structure in C, and the functions you need to write code for are meant to be tree versions of functions in `sorted-int-list.c`. It's up to you whether to use recursion or iteration or both, but I advise that recursion will probably be much easier for the two functions that print the tree and is effective for the others as well.
- I recommend that at some point you run the completed test program with `valgrind` to check that you don't have memory leaks.

What to turn in: Just send me your `int-bst.c` file, unless you added more tests to `test-int-bst.c`, in which case send that too (but *be sure* your code works with the provided version as well).

3 Honor Code Statement

Include the Honor Code pledge or just the word "pledged", plus *at least one of the following* about collaboration and help (as many as apply).[1] Text *in italics* is explanatory or something for you to fill in. For programming assignments, this should go in the body of the e-mail or in a plain-text file `honor-code.txt` (no word-processor files please).

- This assignment is entirely my own work. *(Here, "entirely my own work" means that it's your own work except for anything you got from the assignment itself — some programming assignments include "starter code", for example — or from the course Web site. In particular, for programming assignments you can copy freely from anything on the "sample programs page".)*
- I worked with *names of other students* on this assignment.
- I got help with this assignment from *source of help — ACM tutoring, another student in the course, the instructor, etc. (Here, "help" means significant help, beyond a little assistance with tools or compiler errors.)*
- I got help from *outside source — a book other than the textbook (give title and author), a Web site (give its URL), etc.. (Here too, you only need to mention significant help — you don't need to tell me that you looked up an error message on the Web, but if you found an algorithm or a code sketch, tell me about that.)*
- I provided help to *names of students* on this assignment. *(And here too, you only need to tell me about significant help.)*

[1] Credit where credit is due: I based the wording of this list on a posting to a SIGCSE mailing list. SIGCSE is the ACM's Special Interest Group on CS Education.

4 Essay

Include a brief essay (a sentence or two is fine, though you can write as much as you like) telling me what about the assignment you found interesting, difficult, or otherwise noteworthy. For programming assignments, it should go in the body of the e-mail or in a plain-text file `essay.txt` (no word-processor files please).