CS201: Data Structures and Algorithms

Assignment 1

Rough Draft

Introduction

For your second programming assignment, you are to implement a heapsort algorithm using a binary tree structure. Since a heap can be efficiently stored in an array, there is never a need to implement a heap as a binary tree, even though heaps are often drawn that way. However, implementing a heap as a tree will give you needed practice in manipulating trees, stacks, and queues.

To make this task more challenging, tree nodes will be restricted to having the following fields: *value*, *left*, *right*, and *parent*, all pointers. All tree-based heap operations must be as efficient as those for a heap implemented as an array:

- reading in a value and adding it to the heap (constant time)
- building the heap (linear time)
- extracting the extreme value (logarithmic time)

You may not use an array anywhere in your implementation.

Input

You are required to use the standard *read* pattern for reading in your data:

```
read a value
while not at end of file
    {
    process the value that was read
    read a value
    }
```

You should use the scanner module to simplify the task of reading data elements:

```
wget troll.cs.ua.edu/ACP-C/scanner.c
wget troll.cs.ua.edu/ACP-C/scanner.h
```

Documentation on how to use the scanner can be found at the top of *scanner.c*.

Output

Your executable must be named:

```
heapsort
```

The executable will take a file name as a command-line argument and will produce, on *stdout*, the integers found in the given file in sorted order. The output must be all integers on a single line, each integer separated from the next by a single space. There should be no whitespace following the last integer, other than a newline. Here is an example invocation:

```
$ for i in {1..5}; do echo $RANDOM; done > data
$ heapsort data
4106 6986 9883 19357 21214 27330
$ heapsort -D data
27330 21214 19357 9883 6986 4106
$ heapsort -I -i data
4106 6986 9883 19357 21214 27330
$ heapsort -v
Written by Hailey Hackwell
My heapsort works in n log n time because...
$
```

where `$` is the system prompt. The file to be sorted is a free-format text file. That is, the integers within are separated by arbitrary amounts of whitespace and every line ends with a newline.

The executable must handle the following options:

option	*action*
`-v`	give author's name and explanation on how the implementationperforms the sorting in $\theta(n \log n)$ time – In particular explain h
`-i`	sort a file of integers (default)
`-r`	sort a file of real numbers
`-s`	sort a file of tokens or quoted strings
`-I`	sort in increasing order (default)
`-D`	sort in decreasing order

Here is a program that features some handy-dandy option-handling code that you may use verbatim without credit: `options.c` Options precede the data file and may come in any order.

Approach

Since heaps are structured as *complete* binary trees, you will need to build a balanced tree. Your will repurpose a binary search tree to accomplish this task. The specification for a binary search tree can be found here: `http://beastie.cs.ua.edu/cs201/assign-bst.ht` The specification for a heap can be found here: `http://beastie.cs.ua.edu/cs201/assign-heap.html`.

In order to build a complete tree, you will need to keep track of the nodes as they are added and as they are removed (via the heaps extract method). Use queues and stacks as needed for this task.

NOTE: Since you will not be using the bst modules as a search tree, your heapsort code can pass in NULL for the comparator and swapper.

System requirements

See the *previous* assignment.

Testing

I will be testing your heapsort implementation against my implementation. I will also be substituting my modules for your modules and vice versa. Therefore, it is important for you to adhere to the public interfaces found in the `.h` files.

Project compilation

You must implement your modules in C99. You must provide a file named *makefile*, which responds properly to the commands:

```
make
make test
make clean
make valgrind
```

The `make` command compiles your program, which should compile cleanly with no warnings or errors at the highest level of error checking (the `-Wall and -Wextra options`). The `make test` command should test your program, the `make clean` command should remove object files, and the `make valgrind` command should test an executable with *valgrind*.

The correctness and efficiency of your makefile will be tested. You must have the correct dependencies and your makefile should not perform any unnecessary compilations.

Documentation

All code you hand in must be attributed to its author. Comment sparingly but well. Do explain the purpose of your program. Do not explain obvious code. If code is not obvious, consider rewriting the code rather than explaining what is going on through comments.

Submission

You will hand in (electronically) your code for the preliminary assessment and for final testing with the commands:

```
submit cs201 lusth compile
submit cs201 lusth test1
submit cs201 lusth assign1
```