Assignment 5: Heap and to-do list

There are two parts to this assignment. In the first part, you will complete the implementation of **heap-based Priority Queue** and implement the **Heapsort** algorithm. In the second part, you will use the Priority Queue to implement a **to-do list application**.

Heap implementation

We will approach the heap implementation by adding an interface, in the form of a set of functions, to the dynamic array. You are required to implement all functions in `dynamicArray.c` that begin with the `// FIXME: implement` comment.

The dynamic array uses the `void*` type for elements, so some of the heap functions will also take a function pointer as a parameter to use for element comparisons. This compare function has the following signature and will be implemented for use with the `struct Task` type in the to-do list part of the assignment.

```
#define TYPE void*
...
/**
 * Returns
 * -1 if left < right,
 *  1 if left > right,
 *  0 if left == right.
 */
int compare(TYPE left, TYPE right);
```

All heap interface function names begin with `dyHeap`. Refer to worksheets 33 and 34 for more information about them. Remember that all heap operations must maintain the heap property.

Tests

A test suite, using the CuTest library, is provided in `tests.c`. It covers all functions you are required to implement for this part of the assignment. If your implementation looks reasonable and passes all these tests, it is probably correct and will earn full points for this part of the

assignment.

To build the test suite, enter `make tests` on the command line. This will create a binary named `tests`. When you run it with `./tests`, it will print a report detailing which tests failed with the line of the assertion that caused the failure.

Each test function in `tests.c` has a name prefixed with `test` and suffixed with the name of the function being tested, and takes a `CuTest*` as a parameter. Feel free to add your own tests–there is more on how to do that in the `tests.c` comments.

> Side note: any failed test may elicit a memory leak report in Valgrind. This is because the test halted on an assertion and did not reach the following cleanup code. If a memory leak is reported for a call stack containing a failing test, you may ignore it until that test passes.

To-do list implementation

In this part of the assignment you will implement a to-do list application with a heap-based priority queue. This interactive application allows the user to manage a prioritized to-do list by adding new tasks, viewing the highest priority task, removing the highest priority task, saving the list to a file, and loading the list from a file.

When ran, the to-do list must prompt 7 options to the user:

- `l` to load the list from a file (function implemented for you).
- `s` to save the list to a file (function implemented for you).
- `a` to add a new task to the list.
- `g` to get the first task from the list.
- `r` to remove the first task from the list.
- `p` to print the list (function implemented for you).
- `e` to exit the program.

Once the user picks an option, the program should carry out the operation and then present the user with the 7 options again. This should continue until the user selects `e` to exit the program.

You must implement the option-handling function with the `// FIXME: implement` comment in `toDo.c` to complete the application logic. You must also implement some functions in `task.c`, including a compare function, to complete the `Task` type interface.

In addition to the skeleton code and tests, you are provided with the following files.

- `todo.txt` – An example file storing a saved to-do list. Your save function should generate a to-do list file with the same format.
- `programDemo.txt` – A log demonstrating the approximate behavior your application should emulate.

> Note: If you wish to complete the to-do list application before completing the heap implementation, you can simulate the heap with the following function call replacements:
>
> - `dyOrderedAdd` instead of `dyHeapAdd`.
> - `dyGet(heap, 0)` instead of `dyHeapGetMin`.
> - `dyRemoveAt(heap, 0)` instead of `dyHeapRemoveMin`.
>
> These replacements can only be used temporarily. You **must** replace them with the proper heap functions before submitting.

Tests

This program will be tested interactively, but there are also tests in `tests.c` for two Task related functions you must implement in `task.c`.

Extra credit

The following extension of the heap implementation is worth a few points:

Imagine adding 10 tasks with the same priority to the priority queue. Simulate this by hand. In what order are they removed when you remove the min element 10 times? Is this what you would expect? If not, modify your implementation to produce a more appropriate result when duplicate tasks are added and subsequently removed.

Zip a complete program demonstrating the effect of the changes you made in a separate zip file.

Grading

All code must compile and run on flip to be graded.

Each unit test that passes is worth the following points. Ideally, these tests are independent, so it is possible for some tests to pass and others not to.

- `testAdjustHeap` – 15
- `testBuildHeap` – 5
- `testDyHeapAdd` – 10
- `testDyHeapRemoveMin` – 5
- `testDyHeapGetMin` – 5
- `testDyHeapSort` – 5
- `testTaskNew` – 5
- `testTaskCompare` – 10

To-do list:

- To-do list application – 25
- Compile/style – 15

Submission

Sumbit the following files to TEACH and Canvas.

- `dynamicArray.c`
- `task.c`
- `toDo.c`
- `extraCredit.zip` (optional)