Wesleyan University, Fall 2021, COMP 211
Homework 7: More arrays: an editor buffer
Due by 11:59pm on November 9, 2021

---

1. Written problems (5 points)

Problem 1. *In a priority queue, each element has a priority assigned to it. For example, if your queue comprised integer values, each value stored might represent the priority of the entry. Now, in a priority queue, rather than always dequeueing the first element of the queue, the highest priority element is dequeued. If multiple elements have the same priority, than the element closest to the front of the queue is dequeued. You now also have some flexibility in choosing how elements are enqueued: rather than just enqueuing elements at the back of the queue, you might choose to keep your queue sorted based on priority, or you might choose to keep your queue unsorted.*

Explain how you would use an array to implement a priority queue. In particular, write out C code for the enqueue and dequeue functions. Assume that each element is an integer, and its priority is its value, so you will work with an array of integers. You can choose to use a circular array if you wish, but it is not necessary. You should assume fixed array capacity, but your enqueue function should assume there is space remaining in the array to enqueue another element. Your dequeue function should assume there is at least one element in the queue. Analyze the cost of your functions in terms of the length of the queue.

Hint: you can implement and test your C code to check that your functions work correctly; if you choose to do this, please include your C code for the enqueue and dequeue functions only in your write-up.

Solution: See Program 1 for a C implementation. The body of each `for` loop has constant cost, and each `for` loop is iterated at most n times, where n is the number of elements in the queue. Thus each `for` loop has cost $O(n)$, and hence both `enqueue` and `dequeue` have cost $O(n)$.

You will learn in COMP 212 that there are more efficient implementations of the priority queue abstract type, ones where the various operations have $O(\lg n)$ cost.

2. Programming problems (15 points)

For this assignment, you will use structures and arrays to implement the buffer of a (very simple) text editor. Think of the data upon which a text editor operates as a sequence of characters, along with an "insertion point" that indicates where the next change is to take place. The *buffer* is the data structure that represents the text being edited, the location of the insertion point and any other information that may be needed. The buffer supports operation such as insertion of a character, deletion of a character next to the insertion point, moving the insertion point left or right, etc.

The code for managing the buffer is typically independent of the code for managing the *display* of the buffer contents (i.e., what the user actually sees). More accurately, an editor typically consists of three components: the buffer, a *view* of the buffer (some sort of display), and a *controller* that accepts user input. At a high level, the controller is really an infinite loop that executes the following steps, where the buffer is initially empty:

```c
typedef struct queue_header {
    int data[MAX_QUEUE_LEN];
    int front; // will always be 0
    int back;
} queue;

void enqueue(queue *q, int val) {
    int insert = q->back;
    for (int i=q->front; i < q->back; i++) {
        if (q->data[i] < val) {
            insert = i;
            break;
        }
    }

    for (int i =q->back+1; i > insert; i--) {
        q->data[i] = q->data[i-1];
    }
    q->data[insert] = val;
}

char dequeue(queue *q) {
    int val = q->data[q->front];
    for (int i=q->front; i < q->back; i++) {
        q->data[i] = q->data[i+1];
    }
    q->back--;
    return val;
}
```

Program 1. Enqueue and dequeue functions for a priority queue

(1) Update the view, which will display the text in the buffer, along with an indication of the location of the insertion point (often by a vertical bar between two of the characters in the text).

(2) Wait for an *event* that indicates that the user wishes to change the buffer. Typical events are a key press of an ordinary character (indicating that the user wishes to insert a character into the buffer), a key press of the backspace or delete key (indicating that the user wishes to delete a character from the buffer), or a key press of an arrow key (indicating that the user wishes to change the position at which the next event will affect the buffer).

(3) Modify the buffer according to the event.

The driver program provided in the code distribution implements a text-based version of a controller and view by presenting the user with a menu of options (like insert a character, delete the character to the left of the insertion point, etc.) and displaying the contents of the buffer as a string. Provided you implement an appropriate set of functions for modifying and querying the buffer, neither the controller nor the view need to know anything about how the text and insertion point are actually represented. Your job is to implement the buffer structure and that appropriate set of functions. In a bit more detail, you will implement the following:

- A buffer structure that has at least one field: an array of `char` values. You will almost certainly have other fields (see below).
- An insertion function that inserts a new character into the buffer at the current insertion point.
- Two deletion functions that delete the character to the left or right of the current insertion point.
- Two functions to move the insertion point to the left and the right.
- A function to set the insertion point to a specific position.
- A function that takes a buffer and two `char` arrays as parameters, and fills one of the arrays with the contents to the left of the insertion mark and the other with the contents to the right of the insertion point. The view uses this function to display the contents of the buffer and the insertion point.

The functions are specified in `hw7.h`, and you will implement them in `hw7.c`. The header file `hw7.h` has an empty definition of a buffer structure; it is up to you to fill in the details. Do not modify any other part of `hw7.h`.

3. Additional notes

(1) As mentioned, your `buffer` structure must have at least a character array, which you will use to store the characters in the buffer. However, you must also somehow represent the location of the insertion mark. One way to do this is to use one `NULL` character to stand for the insertion point, and a second `NULL` character to stand for the end of the text (the `char` array is of fixed size, so in general will be bigger than the number of characters in the text). Another way is to use another field that is the index of the character just to the right or left of the insertion point. The former might be slightly simpler for this assignment, but I recommend trying the latter. The reason is that we will revisit this problem in a later assignment using a linked list instead of an array, and you will need to use a version of the latter approach then, so you might as well get comfortable with it now.

(2) Insertion and deletion of characters will be inefficient, as you will have to "shift" characters in the array. When we revisit this problem using a linked list, insertion and deletion will be constant-time operations.

(3) Not every value of type `struct buffer` will represent a valid editor buffer. For example, if you use a field for the index of the character next to the insertion point, there will be some sort of bounds on the value of that field. In other words, you must have some sort of invariant on your `buffer` values. You must code up this invariant and `assert` that every function that mutates a `buffer` preserves it. For example, your implementation of `move_left` would look something like the following

```
void move_left(struct buffer* b) {
        ....
        assert(buf_ok(b));
        return;
}
```

where your `buf_ok` function has signature

```
bool buf_ok(struct buffer* b);
```

and returns `true` if `b` satisfies your invariant.

(4) The code distribution comes with a driver program for you to test your code. Figure 1 shows a sample session using the driver. Notice that you must create an empty buffer before performing any operations on it!

Writing automated tests for this sort of structure is a bit challenging. An appropriate test suite will test your buffer on many sequences of arbitrary operations. It is worth thinking about how you might implement such a suite of tests.

4. Code distribution

This assignment comes with a *code distribution*, comprising files that you will need to complete the programming problems for this assignment:

- `hw7.h`: header file for the code you will write. This file declares the functions that you must implement. The only code you should add to this file is to add fields to the `buffer` struct.
- `hw7.c`: function stubs matching hw7.h have been implemented for you to get you started.
- `driver.c`: a small driver program. To compile the driver program, use the command

  ```
  gcc --std=c99 -o driver driver.c hw7.c
  ```

- `Makefile`: a Makefile for this assignment. Instead of using the above command for compiling your code, you can use the command `make driver`.

5. Submission

Submit your written work as `hw6.pdf` and your programming problem as `hw7.h` and `hw7.c` to the Google Drive directory I have created for you named `comp211-f21-USERNAME/hw7/`. You should replace `USERNAME` with your Wesleyan username.

Do not forget that your written work must be submitted as a PDF! And make sure that at the top of each file you have put your name! Do not, however, change the names of the files.

```
$ ./driver
(0) Exit
(1) Create empty buffer
(2) Insert character
(3) Delete left
(4) Delete right
(5) Move insert mark left
(6) Move insert mark right
(7) Set insert mark position
(8) Print buffer
Enter choice: 1
Buffer contents:
|
(0) Exit
...
Enter choice: 2
Enter character: a
Buffer contents:
a|
(0) Exit
...
Enter choice: 2
Enter character: b
Buffer contents:
ab|
(0) Exit
...
Enter choice: 2
Enter character: c
Buffer contents:
abc|
(0) Exit
...
(8) Print buffer
Enter choice: 2
Enter character: d
Buffer contents:
abcd|
```

```
(0) Exit
...
Enter choice: 7
Enter position: 3
Buffer contents:
abc|d
(0) Exit
...
Enter choice: 5
Buffer contents:
ab|cd
(0) Exit
...
Enter choice: 3
Buffer contents:
a|cd
(0) Exit
...
Enter choice: 4
Buffer contents:
a|d
(0) Exit
...
Enter choice: 4
Buffer contents:
a|
(0) Exit
...
Enter choice: 4
Buffer contents:
a|
(0) Exit
...
Enter choice: 3
Buffer contents:
|
(0) Exit
...
Enter choice: 3
Buffer contents:
|
```

Figure 1. A sample session using the editor buffer driver program. The menu has been elided to save space.