15-122: Principles of Imperative Computation, Summer 2011
Assignment 3: Sorting, Queues, Stacks, and Tab Completion

Zachary Sparks (zsparks@andrew)

Out: Friday, May 27, 2011
Due: Wednesday, June 1, 2011
(Written part: before lecture,
Programming part: 11:59 pm)

1 Written: (25 points)

The written portion of this week's homework will give you some practice working with sorting algorithms, stacks and queue data structures and performing some asymptotic analysis tasks. You can either type up your solutions or write them *neatly* by hand, and you should submit your work in class on the due date just before lecture begins. Please remember to *staple* your written homework before submission.

1.1 Sorting

Exercise 1 (11 pts)**.** Consider the following function that sorts an array of integers.

```
void sort(int[] A, int n)
{
    int i = 0;
    while (i < n)
    {
        int j = findmin(A, i, n);
        swap(A, i, j);
        i = i + 1;
    }
}
```

The code makes use of a `findmin(A, lower, upper)` function to find the index of the minimum element in $A[lower .. upper)$. In other words, the `findmin` function obeys the following contract:

```
int findmin(int[] A, int lower, int upper)
  //@requires 0 <= lower && lower <= upper && upper <= \length(A);
  //@ensures lower <= \result && \result < upper;
  //@ensures leq(A[\result], A, lower, upper);
  ;
```

(a) Annotate the code with all contracts (pre-conditions, assertions, loop invariants, and post-conditions) required to justify the correctness of this code. You may use functions such as `is_sorted`, `lt`, `gt`, `leq`, `geq` and other functions given in lecture or recitation code. (You are not required to prove the code correct, but your annotations should be sufficient to allow a proof.)

(b) Assume that `findmin` requires $i - 1$ "instructions" to find the minimum of an unsorted array segment of size i. Let $T(n)$ be the total number of "instructions" executed by `findmin` calls on an array A of size n in the function `sort`. Express $T(n)$ as a function of n in closed form (that is, do not include "$\ldots$" in your final answer). Show all work.

(c) We say that $T(n)$ is $O(f(n))$ if there exist a constant $c > 0$ and $n_0 \geq 0$ such that $T(n) \leq cf(n)$ for all $n \geq n_0$. Using the results obtained in previous part, show that $T(n)$ is $O(n^2)$ by finding appropriate values of c and n_0.

1.2 Run-time Complexity

Exercise 2 (4 pts)**.** The following run times were obtained when using two different algorithms on a data set of size n. Based on this timing data, guess at the asymptotic complexity of each algorithm as a function of n. Use big-O notation in its simplest form and briefly explain how you reached your conclusion.

(a)
```
n           Execution Time (in seconds)
1000        0.743
2000        3.021
4000        12.184
8000        50.320
```

(b)
```
n           Execution Time (in microseconds or millionths of a second)
1000        0.01
1000000     20
1000000000  30000
```

1.3 Stacks and Queues

A `queue` and a `stack` that contains elements of of type `elem` are implemented with the following interfaces:

```
queue queue_new();
bool queue_empty(queue Q);
int queue_size(queue Q);
void enq(queue Q, elem e);
elem deq(queue Q);

stack stack_new();
bool stack_empty(stack S);
int stack_size(stack S);
void push(stack S, elem e);
elem pop(stack S);
```

We do not know how these data structures are implemented. That is, we don't know if the programmer used arrays or linked lists, or something else—just that they somehow implemented the functions shown above, and that they have the same observable behavior as the ones we implemented in class.

Exercise 3 (10 pts)**.** Consider the interfaces for `queue` and `stack` as given above.

(a) Write a function `reverse` for reversing the order of a queue using a stack. Include proper annotations in your solution. What is the worst case run-time complexity of this function if there are n elements in the queue? Explain.

(b) Write a function `stack_equal` that takes two stacks of the same size containing `elem`s and returns true if the stacks contain the same elements in the same order. You may assume a function `bool elem_equal(elem e1, elem e2)` for comparing elements. After the operations, both stacks must remain in the original order; you are allowed to use additional stacks or queues as needed—but no other data structures. What is the worst case run-time complexity of your code if each stack contains n elements? Explain.

Helpful suggestion: Try to write these functions by hand at first, using tools like contracts, pictures, and small, on-paper examples to reason about your code. Once you believe your functions are correct, you may find it useful to test them by implementing and running them on a computer. But working through the functions "offline" first will help you to hone critical reasoning skills and programming instincts that will serve you immeasurably in the future.

2 Programming: Tab Completion (25 points)

Most modern shells (including `csh`, which most of you are probably using) have a feature called *tab completion*, which allows you to only type part of a command, followed by the tab key. It then autocompletes to that entire command. For example,

```
$ vim strin<TAB>
$ vim stringsearch.c0
```

However, this is not always unambiguous—it is possible to type a prefix which has multiple possible completions. In this case, instead of automatically completing to one thing which may or may not be correct, what many shells opt to do is list out all of the options. As another example:

```
$ gcc<TAB>
gcc         gcc34       gcc44       gccbug      gccmakedep
```

Another way to treat an ambiguous tab-completion is by cycling through the possible completions one-by-one. This makes the programmer have to type less, but is inconvenient when there are a large number of possible completions. To make things more useful for the user, we can make this cycle through the possible completions in an order based on some heuristic value that we keep track of.

There are multiple ways to implement this functionality; one way is firmly within our grasp in C_0, using only arrays and some basic data structures!

2.1 Linked Lists and Queues

This section offers a brief introduction to lists and queues; for a more detailed discussion, please see the Lecture Notes on Queues and the Lecture Notes on Stacks.

Eventually, we will return a data structure called a *queue* that contains the `word_freqs` matching a given prefix, sorted by frequency, but first we need to talk about the data structure used to back our implementation of queues: namely, *linked lists*.[1]

Linked lists are a powerful but simple data structure. Each node of a linked list contains two things: the data it stores and a pointer to the next node in the linked list. This structure is defined as follows:

```
struct list_node {
    elem data;
    struct list_node *next;
};

typedef struct list_node* list;
```

[1] `http://cmubash.org/?1609`

We will consider a linked list `L` to be valid if it is `NULL` or if `L->data` is a valid element and `L->next` is a valid linked list.

Task 1 (5 pts)**.** In order to specify that a linked list is sorted and well-formed, we need to write some specification functions. In the file `list.c0`, implement functions matching the following prototypes:

```
bool all_valid (list L);

bool is_sorted_list (list L)
    //@requires all_valid (L);
    ;
```

The function `all_valid` should make sure every piece of data in `L` is valid (according to the `is_elem` function), and `is_sorted_list` should make sure its list is sorted (according to the `elem_leq` function).

Now that we have linked lists, we can write our implementation of queues. A queue is a FIFO (that is, "first in, first out") data structure—elements are removed from it in the order that they are inserted. Importantly, we will not be using the queue insert and remove (cleverly called "enqueue" and "dequeue") to construct the queue that our tab completion function returns, but it is still important to understand these functions in order to understand queues.

In C_0, our representation of queues is as follows:

```
struct queue {
    list front;
    list back;
};

typedef struct queue* queue;
```

A valid queue `Q` cannot be `NULL`, and we maintain a special invariant about queues: instead of being like linked lists where the null pointer represents the empty list, we create a special "blank" node that `back` always points to. Once we reach this node, we have reached the end of the queue—the data stored in it is irrelevant. If both the front and back pointers point to the same (blank) list node, then the queue is empty.

Recall that when we are tab-completing a prefix, we want to cycle through all possible tab completions in order of frequency (that is, in the order in which they have been placed in the queue that `tab_complete` will have returned). The dequeue operation does not support this, though, since it removes an element from the queue—when we reach the last possible tab completion, we want to cycle around to the first.

Task 2 (5 pts)**.** In the file `queue.c0`, implement a function matching the following prototype:

```
queue make_queue (list L)
    //@requires all_valid (L);
    //@ensures \result != NULL;
    //@ensures queue_all_valid (\result);
    //@ensures is_queue(\result);
    ;
```

Your function should make sure the linked list backing the returned queue obeys the queue invariants as described above (and in the lecture notes). You may modify and/or reuse the input list, though you are not required to—there are several ways to solve this problem!

Task 3 (5 pts)**.** In the file `queue.c0`, implement a function matching the following prototype:

```
elem cycle (queue Q)
    //@requires is_queue(Q);
    //@ensures is_queue(Q);
    ;
```

that cycles through the elements of `Q`. It should behave similarly to `dequeue`, but it should never (permanently) remove an element from the queue. Your function should *only* use the queue operations `enqueue` and `dequeue`.

If `Q` has only one element, calling `cycle (Q)` should always return that element. If `Q` has exactly two elements, repeatedly calling `cycle (Q)` should first return the first element, then the second element, then the first element again, and so on.

2.2 Freqshow

As mentioned above, we want to cycle through possible completions in some heuristic order based on some metadata. This kind of analysis can get quite sophisticated, but for now, we will just keep track of an integer in addition to strings. This string represents the *frequency* with which that specific command is used.

In order to do this, we need a data structure containing both a `string` and an `int`. Based on its intended use, we call this `word_freq`; it is defined as follows:

```
struct word_freq {
    string word;
    int freq;
};

typedef struct word_freq* word_freq;
```

So, if we have a variable `wf != NULL` of type `word_freq`, then we can access its string with `wf->word` and its frequency with `wf->freq`.

2.3 Putting It All Together

With linked lists and queues out of the way, we can now write the actual tab-completion code.

Task 4 (10 points). In the file `prefix.c0`, implement a function matching the following prototype:

```
queue tab_complete (word_freq [] A, int n, string prefix)
    //@requires 0 <= n && n <= \length(A);
    //@requires freq_is_alpha_sorted (A, n);
    //@ensures is_queue (\result);
    //@ensures queue_all_valid (\result);
    ;
```

that returns a queue containing the elements of `A`, sorted according to their frequencies. Recall that `A` is sorted alphabetically, *not* according to frequency.

There are two functions you may find useful in implementing `tab_complete`:

1. `struct range* find_matching_range(word_freq[] A, int n, string prefix)`, found in `array.c0`, returns the `struct range *` describing the subarray of `A` containing words which match the given `prefix`.

   (Recall that a range is represented by a struct containing two `int`s:

   ```
   struct range {
       int start;
       int end;
   };
   ```

   A `struct range* r` represents the half-open interval `[r->start .. r->end)`.)

2. `list insert_in_order(elem data, list L)`, found in `list.c0`, inserts `data` into the sorted-by-frequency list `L` in its appropriate place, returning the result.