Midterm II Exam

15-122 Principles of Imperative Computation, Spring 2012
André Platzer   Ananda Gunawardena

April 5, 2012

Name: **Sample Solution**   Andrew ID: **aplatzer**

Instructions

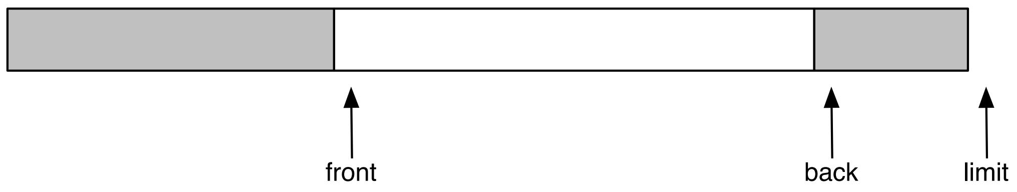
- This exam is closed-book with one sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 3 problems.
- Read each problem carefully before attempting to solve it.
- Do not spend too much time on any one problem.
- Consider if you might want to skip a problem on a first pass and return to it later.
- Consider writing out programs or diagrams on scratch paper first.
- And most importantly,

DON'T PANIC!

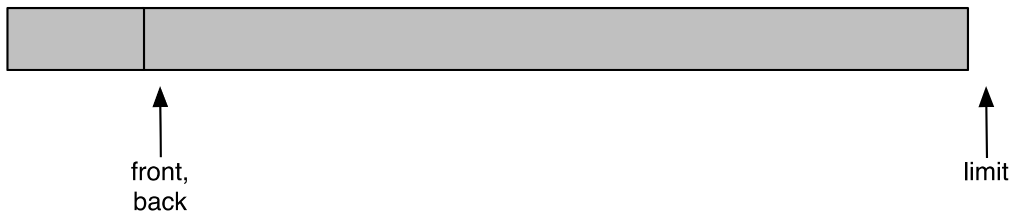
	Message Buffer	Virus Scan	BSTs	
	Prob 1	Prob 2	Prob 3	Total
Score	**45**	**25**	**30**	**100**
Max	45	25	30	100
Grader	??	??	??	

1 Message Buffer (45 points)

You are implementing a network driver that stores the messages that it received in a queue. We can implement that queue in an array, remembering its length and two indices, one to the *front* of the queue and one to the *back*:



As usual, messages are inserted at the back and removed from the front. If either index ever passes the limit, it wraps around to the beginning of the array. You may assume that the limit is greater than zero. Due to this wrapping nature of the indices, the array can be visualized as a ring, which is why it is called a *ring buffer*. The *front* and *back* indices should only be equal if the queue is empty:



```
typedef struct queue* queue;
struct queue {
    int limit;
    message[] A;
    int front;
    int back;
};
```

As an example, here is the function that checks whether a queue is empty.

```
bool queue_empty(queue Q)
//@requires is_queue(Q);
{
    return Q->front == Q->back;
}
```

(Continued)

Task 1 (10 pts). Identify all relevant data structure invariants for the (ring buffer) `queue` by implementing the `is_queue` function to check them.

```
bool is_queue(queue Q) {
```

```
    if (Q == NULL) return false;
    if (!(Q->limit > 0)) return false;
    //@assert \length(Q->A) == Q->limit;
    if (!(0 <= Q->front && Q->front < Q->limit)) return false;
    if (!(0 <= Q->back && Q->back < Q->limit)) return false;
    return true;
```

```
}
```

Task 2 (5 pts). Implement the deq function that dequeues the message at the front of the queue.

```
message deq(queue Q)
//@requires is_queue(Q);
//@requires !queue_empty(Q);
//@ensures is_queue(Q);
{
```

```
    assert(!queue_empty(Q));
    message e = Q->A[Q->front];
    Q->front = (Q->front + 1) % Q->limit;
    return e;
```

```
}
```

Task 3 (5 pts). How can you use the idea behind how unbounded arrays double arrays to make sure the queue is never full, i.e., that enq always works successfully (without losing messages), no matter how big the array of the queue used to be before calling enq? You do not need to give code.

Copy the array over into a new array that is twice as big whenever the old array is full. For future reference in Task 4, we call this function

```
void queue_double(queue Q);
```

Task 4 (5 pts). Implement the enq function that enqueues a message. Make sure it works as expected no matter how big the array of the queue used to be. You can use helper functions that you have explained in your answer to Task 3.

```
void enq(queue Q, message e)
//@requires is_queue(Q);
//@ensures is_queue(Q);
//@ensures !queue_empty(Q);
{
```

```
     Q->A[Q->back] = e;
     Q->back = (Q->back + 1) % Q->limit;
     if (Q->back == Q->front) queue_double(Q);
```

```
}
```

(Continued)

The following questions refer to the `enq` function that you just implemented. Express your answer in big-O notation in terms of the number of array reads and writes that an operation performs.

Task 5 (5 pts). What is the worst-case asymptotic complexity of a single queue operation on a queue of size n? Justify your answer in a sentence or two.

The worst case is an `enq` operation that causes the queue to become full: the array will have to be doubled in size, and since `queue_double` copies every element, the operation will require $O(n)$ array reads and writes.

Task 6 (5 pts). What is the worst-case asymptotic complexity of a sequence of k queue operations starting from an empty queue? Justify your answer in a sentence or two.

Starting from an empty queue, we know we'll only have to double every time we enqueue enough elements to exceed our limit. As with unbounded arrays, if we budget 3 tokens per `enq` operation, then we will always have enough tokens to pay for a doubling, making the total cost of k operations less than $3k$, or $O(k)$.

(Continued)

Task 7 (10 pts). How can you change your implementation so that it gives priority to high-priority messages (flagged with an extra input `important==true` on delivery). The order of the messages upon delivery via deq should still be in order, except that high-priority messages (`important==true`) should always be delivered faster than low-priority messages (`important==false`).
No code required. A *precise* explanation of your approach is sufficient. If you are unsure about whether your explanation is precise, you should write code.

```
void enqp(queue Q, message e, bool important)
//@requires is_queue(Q);
//@ensures is_queue(Q);
//@ensures !queue_empty(Q);
;
```

The priority queue can be formed out of two queues (e.g., the array-based ones we have implemented here), one for the important messages one for the unimportant ones. When enqueuing, messages get `enq`'d into the respective queue according to priority. When dequeuing, dequeue from the queue for important messages first, and use the unimportant queue only if the important one is empty. This requires simple changes to the struct, enqp, deq, and new.

2 Virus Scan (25 points)

You have a database of m bit sequences (called *signatures*), each of length at most k. Each signature is a bit sequence that identifies a computer virus. Your task is to write a function `infected` that checks whether one of the virus signatures is contained in a file of size n bits. We assume that the file has already been read into an array F of bits (represented as `bool`) of length n.

Task 1 (5 pts). A naive implementation of `infected` looks at each position i of F and checks whether any of the signatures start at that position by comparing the signature with F starting at i. What is the asymptotic number of comparisons of bits (from F), as a function of m, n and k, of this naive implementation of the function `infected`? Justify your answer (1 or 2 sentences).

$O(n * m * k)$, because, starting from each of the n positions, each of the m signatures needs to be checked for up to k bits

(Continued)

Task 2 (15 pts). In order to improve the efficiency of your virus scanner, you decide to store the virus signatures in a binary trie. Recall that binary tries decompose bit sequence representations bit by bit in order to access data stored in the trie.

```
typedef struct btrie* btrie;
struct btrie {
  bool virus;   /* true if identifies a virus */
  btrie btrue;  /* trie for bit true */
  btrie bfalse; /* trie for bit false */
};
```

Reimplement `infected` in a more efficient way by using the virus signatures stored in a btrie. Your implementation does not have to achieve the best possible performance. It is enough if the code is correct and asymptotic complexity strictly better than that of the naive implementation.
Hint: If you use recursion, then you may find it useful to implement an auxiliary function that handles one index.

```
bool infected_from(btrie signatures, bit[] F, int i, int n)
//@requires 0 <= i && i <= n && n == \length(F);
{
  if (signatures == NULL) return false;
  if (signatures->virus) return true;
  if (i == n) return false;
  if (F[i]) return infected(signatures->btrue, F, i+1, n);
  else return infected(signatures->bfalse, F, i+1, n);
}
bool infected(btrie signatures, bit[] F, int n)
//@requires 0 <= n && n == \length(F);
{
  for (int i = 0; i < n; i++)
    if (infected_from(signatures, F, i, n)) return true;
  return false;
}
```

Task 3 (5 pts). What is the asymptotic number of comparisons of bits (from F), as a function of m, n and k, of your improved implementation of the function `infected` from Task 2? Justify your answer in a sentence or two.

$O(n * k)$, because `infected` calls `infected_from` on each of the n positions, which in turn searches for matches for up to k recursive steps.

3 Binary Search Trees (30 points)

In this problem we consider plain binary search trees with of `int` keys, without rebalancing.

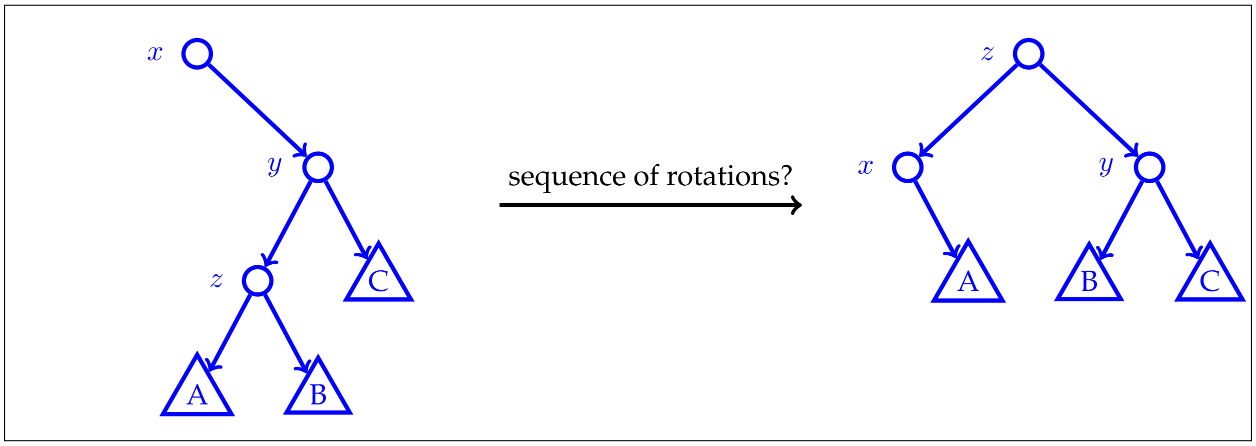
Task 1 (10 pts). Give a binary search tree with 3 elements (`int` keys) for which there is only one node at which a new `int` key can be inserted. That is, any key inserted into the binary search tree will end up in exactly the same node (assuming that it is not a duplicate of a key already in the tree). Of course, more than one key could be inserted into the binary search tree, but they all need to end up in the same node of the binary search tree.

```
         MININT+1                                 MAXINT-1
        /         \            or               /          \
MININT             MININT+2              MAXINT-2            MAXINT
```

Task 2 (5 pts). Assume that we have an array sorted *in reverse* from which we want to construct a binary search tree. Which order of insertion should we follow to obtain a tree that is as balanced as possible?

To insert a range $A[lower..upper)$ of an array, start with the middle element $A[mid]$ for $mid = lower + (upper - lower)/2$ and then recursively insert the ranges $A[lower..mid)$ and $A[mid + 1..upper)$, in either order.

Task 3 (10 pts). The binary search tree on the left has been rotated by a sequence of single left or right rotations to the binary search tree on the right. Unlike the nodes x, y, z, which represent `int` keys, A, B, and C represent subtrees of unknown size. Complete the shape of the tree on the right by putting the subtrees A, B, C where they belong.



Task 4 (5 pts). Give a sequence of single left or right rotations, and the two nodes being rotated, to transform the binary search tree on the left to the binary search tree on the right that you identified in Task 3. For full credit, find the shortest sequence (which requires less than 5 steps). Indicate in the rightmost column what we know about whether the tree *after* the rotation satisfies the balance invariant of the AVL trees (i.e., the heights of the left and right children of any node differ by at most one) by indicating yes/no/maybe.

Step	**Left or Right?**	**at nodes**	**AVL?**
1	right	y, z	no
2	left	x, z	maybe
3			
4			
5			