Exam II

15-122 Principles of Imperative Computation, Summer 2011
William Lovas

June 10, 2011

Name: **Sample Solution** Andrew ID: **wlovas**

Instructions

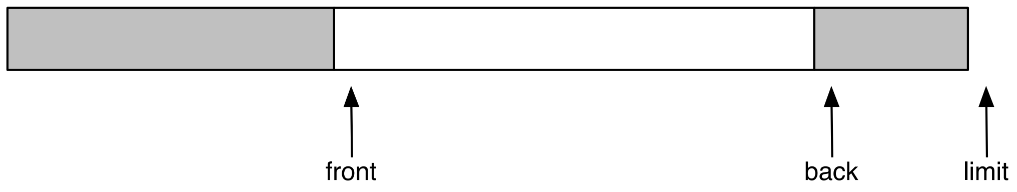
- This exam is closed-book with one sheet of notes permitted.
- You have 80 minutes to complete the exam.
- There are 4 problems.
- Read each problem carefully before attempting to solve it.
- Do not spend too much time on any one problem.
- Consider if you might want to skip a problem on a first pass and return to it later.
- Consider writing out programs or diagrams on scratch paper first.
- And most importantly,

DON'T PANIC!

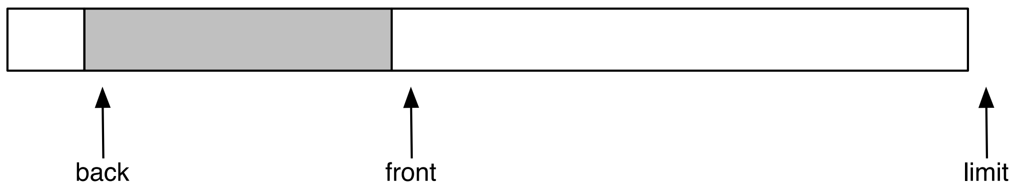
	UBA Queues	Heapsort	Ropes	BSTs and Rotations	
	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score	**40**	**40**	**40**	**30**	**150**
Max	40	40	40	30	150
Grader	??	??	??	??	

1 Queues as Unbounded Arrays (40 points)

We can use the idea behind an unbounded array to implement queues. We represent a queue as an array stored with its length and two indices, one to the *front* of the queue and one to the *back*:



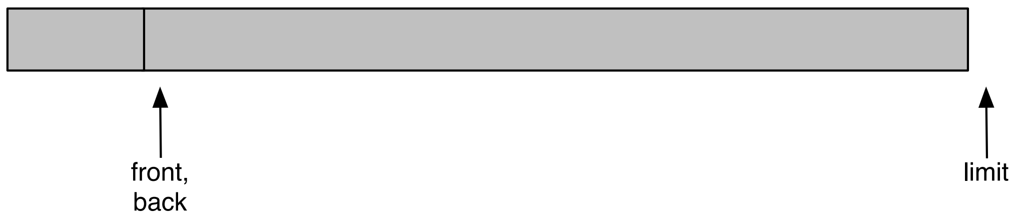
As usual, elements are inserted at the back and removed from the front. If either index ever passes the limit, it wraps around to the beginning of the array:



Due to this wrapping nature of the indices, the array can be visualized as a circle or a ring, so this representation is sometimes called a *circular buffer* or a *ring buffer*. If the array becomes full, we can double its size using a `queue_double` function, which doubles the array backing a queue, copying over all of its elements in their insertion order:

```
void queue_double(queue Q)
//@requires is_queue(Q);
//@ensures is_queue(Q);
//@ensures Q->limit == 2 * \old(Q->limit);
;
```

The *front* and *back* indices should only be equal if the queue is empty:



```
typedef struct queue* queue;
struct queue {
    int limit;  /* limit > 0 */
    elem[] A;   /* \length(A) == limit */
    int front;  /* 0 <= front && front < limit */
    int back;   /* 0 <= back && back < limit */
    /* front == back iff queue is empty */
};
```

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;
}
```

Task 1 (10 pts). Implement the `is_queue` function that checks the data structure invariants of a queue implemented as a ring buffer.

```
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;
```

```
}
```

(Continued)

Task 2 (10 pts). Implement the deq function that dequeues the element at the front of the queue.

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

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

```
}
```

Task 3 (10 pts). Implement the enq function that enqueues an element. Remember to double the size of the queue if necessary.

```
void enq(queue Q, elem 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)

For the following questions, express your answer in big-O notation in terms of the number of array reads and writes that an operation performs.

Task 4 (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 5 (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 $O(k)$.

2 Heapsort (40 points)

We can use the invariant behind heaps in order to implement an in-place sorting algorithm for arrays called *heapsort*. For simplicity, we use *max* heaps, which satisfy:

> **Max Heap Ordering Invariant:** Each node except for the root must be *less or equal* to its parent.

This guarantees that a *maximal* element is at the root of the heap, rather than a minimal one as we did in lecture.

The algorithm proceeds in two phases. In phase one we build up a heap spanning the whole array, and in phase two we successively delete the maximum element from the heap and move it the end.

Here is our implementation, written compactly, with only pre- and post-conditions, but no loop invariants or assertions. Note that we only sort the range $A[1, n)$, ignoring $A[0]$.

```
void heapsort(int[] A, int n)
//@requires 1 <= n && n <= \length(A);
//@ensures is_sorted(A, 1, n);
{
    int i;
    for (i = 2; i < n; i++) {       /* phase one */
      sift_up(A, i, i+1);
    }
    for (i = n-1; 2 <= i; i--) {    /* phase two */
      swap(A, 1, i);
      sift_down(A, 1, i);
    }
}
```

The functions `sift_up` and `sift_down` are like the functions we wrote in lecture, except that they take an array as a first argument and what we called `H->next` (the index right after the last element currently in the heap) as the third argument. The contracts for both functions are given below.

Your main task will be to enrich this code with invariants and assertions. You should assume the following functions:

```
bool is_heap(int[] A, int n);
bool is_heap_except_up(int[] A, int i, int n);
bool is_heap_except_down(int[] A, int i, int n);
bool is_sorted(int[] A, int lower, int upper);
```

with the following interpretation:

`is_heap(A, n)` means that the range $A[1, n)$ satisfies the heap invariant.

`is_heap_except_up(A, i, n)` means that the range $A[1, n)$ satisfies the heap invariant except that $A[i]$ (which must be in the heap) may be greater than its parent.

`is_heap_except_down(A, i, n)` means that the range $A[1, n)$ satisfies the heap invariant except that $A[i]$ (which must be in the heap) may be less than one or both of its children.

`is_sorted(A, lower, upper)` means that the range $A[lower, upper)$ is sorted in increasing order.

Here are the contracts for the `sift_up` and `sift_down` functions that are called from `heapsort`.

```
void sift_up(int[] A, int i, int n)
//@requires is_heap_except_up(A, i, n);
//@ensures is_heap(A, n);
  ;

void sift_down(int[] A, int i, int n)
//@requires is_heap_except_down(A, i, n);
//@ensures is_heap(A, n);
  ;
```

Task 1 (25 pts). The following is a correct implementation of `heapsort`, which sorts the range $A[1, n)$ in place. Fill in the strongest correct annotations in the given places, using only the functions `is_heap`, `is_heap_except_up`, `is_heap_except_down` and `is_sorted`. To give you a head start we have included loop index invariants already. (**Hint:** you may find it helpful to draw a picture visualizing the intermediate stages of heapsort.)

```
void heapsort(int[] A, int n)
//@requires 1 <= n && n <= \length(A);
//@ensures is_sorted(A, 1, n);
{   int i;
    for (i = 2; i < n; i++)
    //@loop_invariant 2 <= i && i <= n;
    //@loop_invariant ___ is_heap(A, i) ____________________________________ ;
    {
        //@assert ___ is_heap_except_up(A, i, i+1) _________________________ ;
        sift_up(A, i, i+1);
    }
    //@assert ___ is_heap(A, n) ____________________________________________ ;
    for (i = n-1; 2 <= i; i--)
    //@loop_invariant 1 <= i && i <= n-1;
    //@loop_invariant ___ is_heap(A, i+1) && is_sorted(A, i+1, n) __________ ;
    {
        swap(A, 1, i);
        //@assert ___ is_heap_except_down(A, 1, i) && is_sorted(A, i, n) ___ ;
        sift_down(A, 1, i);
    }
}
```

Task 2 (15 pts). Analyze the asymptotic complexity of our version of heapsort.

During phase one, each of the $n - 2$ call to sift-up is bounded by $O(\log(n))$, so phase one is bounded by $O(n * \log(n))$. Similarly, each of the $n - 2$ calls to sift-down is bounded by $O(\log(n))$, so phase two is also bounded by $O(n * \log(n))$, leading to a total of $O(2 * n * \log(n)) = O(n * \log(n))$.

3 Ropes (40 points)

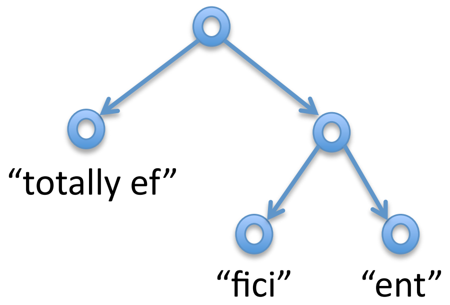
In C0 and C, strings are typically represented as arrays of characters. This allows constant-time access of a character at an arbitrary position, but it also has some disadvantages. In particular, concatenating two strings (function `string_join`) is an expensive operation since we have to create a new character array and copy the two given strings into the new array character by character.

Task 1 (10 pts). What is the asymptotic complexity of the following loop as a function of n? Assume that `string_fromint` is a constant-time operation.

```
string string_fromarray(int[] A, int n)
//@requires 0 <= n && n < \length(A);
{
    string s = "["; int i;
    for (i = 0; i < n; i++)
        //@loop_invariant 0 <= i && i <= n;
        s = string_join(s, string_fromint(A[i]));
    return string_join(s, "]");
}
```

The length of the result of `string_fromint` is bounded by a constant c, and the complexity of `string_join` is $O(k)$, where k is the the number of characters in the result. Therefore the complexity is $O(1 * c + 2 * c + 3 * c + \cdots + n * c) = O(n^2 * c) = O(n^2)$.

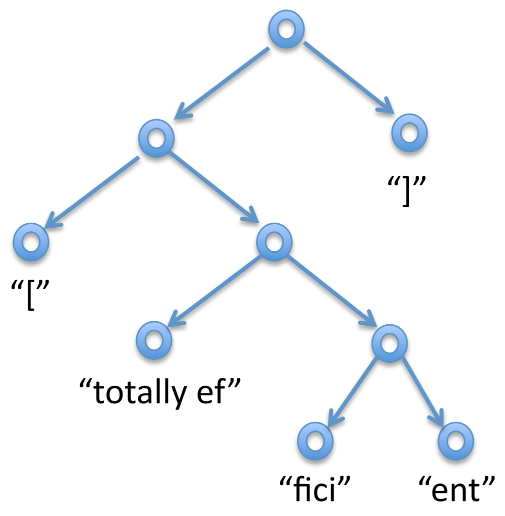
The data structure of *ropes* attempts to improve efficiency of concatenation by representing strings as binary trees, where the leaves contain ordinary strings and the intermediate nodes represent concatenations. For example, the string `"totally efficient"` might look as follows (among many other possibilities):



Note that ordinary strings are only stored at the leaves. Assuming no rebalancing, concatenation of ropes is a constant-time operation.

(Continued)

Task 2 (10 pts). Assuming no rebalancing, show the final result of concatenating the rope above first on the left with `"["` then on the right with `"]"` to form a rope for the string `"[totally efficient]"`. (Remember that concatenation should be a constant-time operation!)



Note: Leaf insertion, as we have done for ordinary binary search trees, would require $O(\log(n))$ steps even for balanced trees, rather than $O(1)$ as explained.

(Continued)

Task 3 (10 pts). Describe what additional information you might store in the nodes so that accessing the ith character in a string of length n represented by a rope is $O(\log(n))$ if the rope is balanced.

Store the total length of the string represented by the tree at every node. Looking up an index i goes to the left if the index is strictly less than the length l stored in the left subtree. If $i \geq l$ we go to the right subtree and look up index $i - l$. An index is out of bounds at the root if it is negative or greater or equal to the stored length.

Note: Several variations are possible, but storing the absolute character range for the substring stored in a rope would require an unacceptable $O(n)$ traversal of the right tree after each concatenation to restore this invariant.

Task 4 (10 pts). Carefully describe the invariant for your data structure. You do not need to write code to check it, but your description should be precise and detailed enough that it would be clear how to write the code.

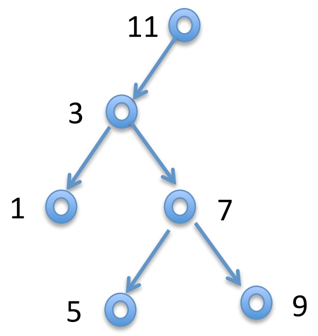
We check that ropes are not empty, that non-empty strings are only stored at leaf nodes (with no children), and that interior nodes have exactly two children.

The length invariant is that each node contains exactly the total length of the string represented by its tree. We can check this at the leaves by comparing it with the length of the string stored there. At interior nodes we check that the stored length is equal to the lengths stored at the subtrees.

4 Binary Search Trees and Rotations (30 points)

In this problem we consider plain binary search trees, without rebalancing.

Task 1 (5 pts). Consider the tree below. Give a sequence of numbers that, when inserted in that order into an empty tree, would construct exactly this tree.



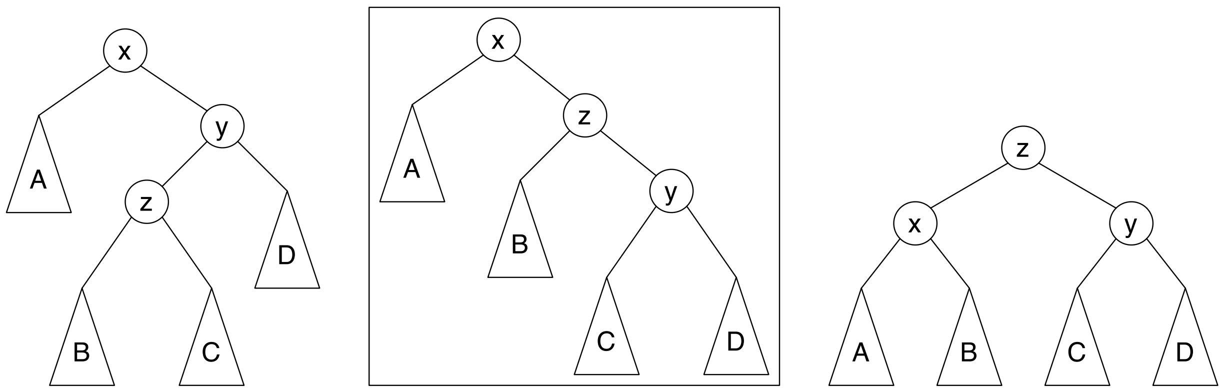
11, 3, 1, 7, 5, 9

Task 2 (5 pts). Assume that we have a sorted array 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?

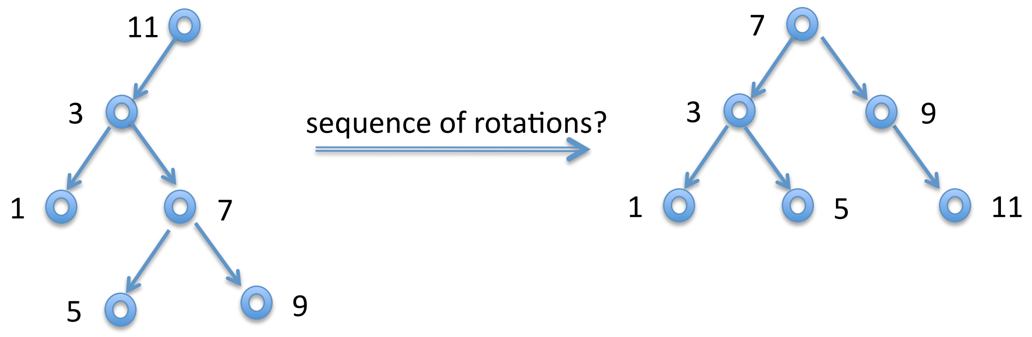
To insert a range $A[lower..upper)$ of an array, start with the middle element $A[mid]$ for $mid = lower + (upper - lower)/2$ and then recursively insert the ranges $A[lower..mid)$ and $A[mid + 1..upper)$, in either order.

The transformation below, from the tree on the left to the tree on the right, we called a *double rotation*. It can be used to rebalance binary search trees after insertion. We assume keys are integers and x, y, and z are the keys for the three nodes shown explicitly.

Task 3 (10 pts). Show that the name *double rotation* is justified by drawing an intermediate tree between the two, so that each step is a single rotation.



Task 4 (10 pts). Give a sequence of single left or right rotations, and the two nodes being rotated, to transform the tree on the left to the tree on the right. For full credit, find the shortest sequence (which requires less than 5 steps). Indicate in the rightmost column if 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).



Step	**Left or Right?**	**at nodes**	**AVL?**
1	left	3, 7	no
2	right	11, 7	yes
3	right	11, 9	yes
4			
5			