Midterm I Exam

15-122 Principles of Imperative Computation
Frank Pfenning, Tom Cortina, William Lovas

September 30, 2010

Name: Andrew ID:

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.
- Consider writing out programs on scratch paper first.

	Searching & sorting	Stacks & queues	Linked lists	Modular arith. & JVM	
	Prob 1	Prob 2	Prob 3	Prob 4	Total
Score					
Max	40	50	30	30	150
Grader					

1 Searching and Sorting (40 pts)

Shown here is the binary search program from Homework Assignment 2 that has been repaired, except that the condition before the `return` statements at the end of the function has been omitted. A copy of this code is provided on the last sheet, which you may tear off and use for reference while working on Tasks 2 and 3.

```
 1  int binsearch_smallest(int x, int[] A, int n)
 2  //@requires 0 <= n && n <= \length(A);
 3  //@requires is_sorted(A,n);
 4  /*@ensures (\result == -1 && !is_in(x, A, n))
 5          || (A[\result] == x && (\result == 0 || A[\result-1] < x));
 6   @*/
 7  { int lower = 0;
 8    int upper = n;
 9    while (lower < upper)
10      //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
11      //@loop_invariant lower == 0 || A[lower-1] < x;
12      //@loop_invariant upper == n || A[upper] >= x;
13      { int mid = lower + (upper-lower)/2;
14        if (A[mid] < x) lower = mid+1;
15        else /*@assert(A[mid] >= x);@*/ upper = mid;
16      }
17    //@assert lower == upper;

18    if (                                  )

19      return lower;
20    else
21      return -1;
22  }
```

Task 1 (10 pts). Fill in the missing condition at the end of the function so that the proper value is returned.

The next two questions ask you to show that the postcondition of the function is satisfied, in two parts. For each part, reason from the function's precondition, loop invariants, the explicit assertion, and the condition you inserted. You can refer to annotations by the line they appear in.

Task 2 (10 pts). If the function returns some result i for $0 \leq i < n$, show that either $i = 0$ or $A[i-1] < x$.

Task 3 (10 pts). If the function returns -1, show that x cannot be in the array. To simplify the reasoning, you may assume that `lower != 0` and `upper != n` at line 17.

Task 4 (10 pts). The `merge` function employed by `mergesort` as discussed in lecture allocates some fresh temporary space each time it is called. If we call `mergesort` with an array of size n, how much temporary space does `mergesort` allocate, overall? Give your answer in big-O notation and briefly explain your reasoning.

2 Stacks and Queues (50 pts)

Consider the following interface to stacks, as introduced in class.

```
typedef struct stack* stack;
stack s_new();                  /* O(1); create new, empty stack */
bool s_empty(stack S);          /* O(1); check if stack is empty */
void push(int x, stack S);      /* O(1); push element onto stack */
int pop(stack S);               /* O(1); pop element from stack  */
```

In these problem you do not need to write annotations, but you are free to do so if you wish. You may assume that all function arguments of type `stack` are non-`NULL`.

Task 1 (10 pts). Write a function `rev(stack S, stack D)`. We require that D is originally empty. When `rev` returns, D should contain the elements of S in reverse order, and S should be empty.

```
void rev(stack S, stack D)
//@requires s_empty(D);
//@ensures s_empty(S);
{









}
```

Now we design a new representation of queues. A queue will be a pair of two stacks, `in` and `out`. We always add elements to `in` and always remove them from `out`. When necessary, we can reverse the `in` queue to obtain `out` by calling the function you wrote above.

```
struct queue {
  stack in;
  stack out;
};
typedef struct queue* queue;
```

Task 2 (10 pts). Write the `enq` function.

```
void enq(queue Q, int x) {









}
```

Task 3 (10 pts). Write the `deq` function. Make sure to abort the computation using an appropriate `assert(_,_)` statement if `deq` is called incorrectly.

```
int deq(queue Q) {









}
```

Now we analyze the complexity of this data structure. We are counting the total number of `push` and `pop` operations on the underlying stack.

Task 4 (10 pts). What is the worst-case complexity of a single `enq`? What is the worst-case complexity of a single `deq`? Phrase your answer in terms of big-O of m, where m is the total number of elements already in the queue.

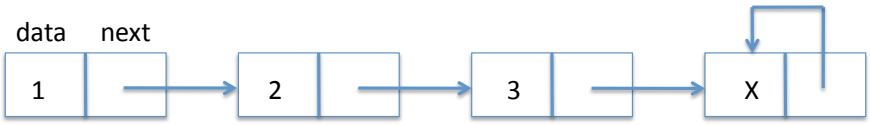
Task 5 (10 pts). What is the worst-case complexity of a sequence of n operations, each of which could be `enq` or `deq`? Justify your answer using amortized analysis, if appropriate.

3 Linked Lists (30 pts)

Recall the definition of linked lists with integer data.

```
struct list {
  int data;
  struct list* next;
};
typedef struct list* list;
```

An alternative to terminating lists with `NULL` is to terminate them with a self-loop. We call such a list a *sloop*. For example, the following is a sloop of length 3.



Task 1 (10 pts). Write a function `is_sloop(list p)` to test if p is a sloop, that is, a linked list terminated by a self-loop. You should assume that there are no other cycles in the list.

```
bool is_sloop (list p) {









}
```

Task 2 (10 pts). The following program is supposed to add an element to the end of a sloop, but it contains three bugs. Fix the bugs by clearly modifying a given statement or adding new statements.

```
list addend (list p, int k)

//@requires is_sloop(p);

//@ensures is_sloop(p);

{ list q = alloc(list);

  while (p != p->next)
    //@loop_invariant is_sloop(p);
  {
    p = p->next;
  }

  p->data = k;

  p->next = q;

}
```

Task 3 (10 pts). Explain in detail how to use the idea behind the tortoise-and-the-hare algorithm to write a stronger `is_sloop` function than you wrote in Task 1. It should terminate with `false` on lists containing a cycle, unless the cycle has only one node. Your description should be concise and complete. If you wish, you can write code to support the explanation, but that is not required.

4 Modular Arithmetic and JVM (30 pts)

Task 1 (5 pts). Implement a function `iushr(n, k)` which is like `n >> k` except that it fills the highest bits with zeros instead of copying the sign bit. `iushr` stands for *integer unsigned shift right*.

```
int iushr(int n, int k) {









}
```

Task 2 (5 pts). Implement a function `oadd(x, y)` which is like `x + y` except that it aborts the computation with an appropriate `assert(_,_)` statement if we have an overflow. Here, *overflow* means that the result would be less than the minimal representable negative number or greater than the maximal representable positive number, assuming 32-bit two's complement arithmetic. Your code does not need to be particularly efficient.

```
int oadd(int x, int y) {









}
```

Now recall the inner loop of the JVM_{00} implementation we developed in class, reduced here to just two instructions to show the overall structure.

```
// P[pc], 0 <= pc < max_pc is the program code
// V[i], 0 <= i < max_local are the local variables
// S is the operand stack
while (true) {
  int inst = P[pc];
  if (inst == 0x60) { push(pop(S)+pop(S),S) ; pc+=1; } // iadd
  else if (inst == 0x15) { push(V[P[pc+1]],S); pc+=2; } // iload <i>
  ... your instructions should go here ...
  else assert(false, "unrecognized instruction");
}
```

Task 3 (5 pts). Implement the instruction `iushr` (`0x7C`). It should transform the operand stack S, x, y to $S, \text{iushr}(x, y)$, where iushr is the function defined in Task 1. You may use the `iushr` function.

```
else if (inst == 0x7C) {









}
```

Task 4 (5 pts). Implement the hypothetical instruction `oadd` (`0xBA`) which transforms the stack S, x, y to $S, \text{oadd}(x, y)$, where oadd is the function defined in Task 2. You may use the `oadd` function. The JVM should abort with an appropriate `assert(_,_)` statement if there is an overflow as defined in Task 2.

```
else if (inst == 0xBA) {









}
```

Task 5 (10 pts). Implement the instruction `iinc <i>,<c>` (`0x84`) which has two more bytes: the index i of a local variable and a byte-size constant c which is interpreted according to 8-bit two's complement representation. It increments $V[i]$ by c and does not affect the operand stack.

```
if (inst == 0x84) {









}
```

You may tear off this sheet and use it for reference while working on Problem 1.

```
 1  int binsearch_smallest(int x, int[] A, int n)
 2  //@requires 0 <= n && n <= \length(A);
 3  //@requires is_sorted(A,n);
 4  /*@ensures (\result == -1 && !is_in(x, A, n))
 5          || (A[\result] == x && (\result == 0 || A[\result-1] < x));
 6   @*/
 7  { int lower = 0;
 8    int upper = n;
 9    while (lower < upper)
10      //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
11      //@loop_invariant lower == 0 || A[lower-1] < x;
12      //@loop_invariant upper == n || A[upper] >= x;
13      { int mid = lower + (upper-lower)/2;
14        if (A[mid] < x) lower = mid+1;
15        else /*@assert(A[mid] >= x);@*/ upper = mid;
16      }
17    //@assert lower == upper;

18    if (                                  )

19      return lower;
20    else
21      return -1;
22  }
```