Midterm I Exam

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

September 30, 2010

Name: **Sample Solution** Andrew ID: **fp**

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	**40**	**50**	**30**	**30**	**150**
Max	40	50	30	30	150
Grader	fp	tc	wjl	tc/fp	

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.

```
18  if (lower < n && A[lower] == x)
19    return lower;
20  else
21    return -1;
```

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 $A[i-1] < x$.

Since $0 \leq i < n$, we know the condition in line 18 must be true, so $lower < n$ and $A[lower] = x$. Also, $lower$ is returned, so $i = lower$.

By loop invariant (11), $lower = 0$ or $A[lower - 1] < x$. This is exactly what we have to show since the return value $i = lower$ and $lower$ is not modified after the loop.

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.

Since the result $i = -1$ and $lower \geq 0$, the test on line (18) must be false. So either $lower = n$ or $A[lower] \neq x$. Since $lower = upper$ by the assertion on line (17), we are allowed to ignore the first case, by the problem statement. Therefore $A[upper] = A[lower] \neq x$.

By loop invariant on line (12), $A[upper] \geq x$ and, since also $A[upper] \neq x$, we have $A[upper] > x$. Also, $A[lower - 1] < x$ by invariant on line (11) and assumption $lower \neq 0$.

Since the array is sorted and $lower = upper$, x cannot be in array: $A[lower - 1] < x$ and $x < A[upper] = A[lower]$.

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.

The size of the temporary array is the size of the result of the merge. During mergesort of a list of length n, we have $O(\log(n))$ levels of recursion, and at each level we perform a merge of n elements (either one merge of size n, or two merges each of size $n/2$, etc.). Therefore we allocate $O(n * \log(n))$ temporary space.

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);
{

   while (!s_empty(S))
     push(pop(S),D);

}
```

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) {

   push(x, Q->in);

}
```

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) {

   assert(!s_empty(Q->in) || !s_empty(Q->out)
          "cannot dequeue from empty queue");
   if (s_empty(Q->out)) { rev(Q->in, Q->out); }
   return pop(Q->out);

}
```

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.

$O(1)$ for `enq` and $O(m)$ for `deq`.

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.

The complexity is $O(n)$, that is, the *amortized* complexity of a single `enq` or `deq` is $O(1)$. We prove this by setting aside 2 tokens for every `enq`, and spending 2 tokens when we move an element from the input queue to the output queue.

Claim: We always have exactly $2 * i$ tokens, if i is the number of elements in the input stack.

It is true initially, since $i = 0$ and we have no tokens.

On every `enq` we add one element and two tokens, so the invariant remains true.

On every `deq` with a non-empty output stack, neither the tokens nor the input stack are affected, so the invariant remains true.

On a `deq` where the output stack is empty, we have $2 * i$ tokens for the i elements in the input stack. During the reverse, we perform one `pop` and one `push` for each of the i elements, spending all $2 * i$ tokens. We end up with 0 tokens and 0 elements on in the input stack, preserving our invariant.

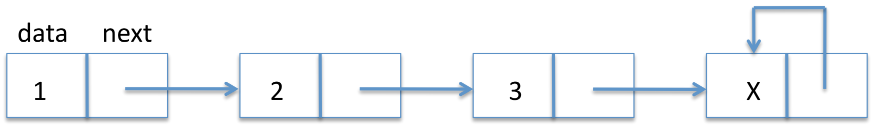
So during n operations, the total number of stack operations is bounded by the number e of `enq`'s, plus the number d of `deq`'s, plus the number of tokens which is $2 * e$. In total, this gives us $O(3 * e + d) = O(n)$, since $n = e + d$.

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) {

  while (p != NULL) {
    if (p == p->next) return true;
    p = p->next;
  }
  return false;

}
```

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;

}
```

1. The function should return `void`. Alternatively, it could return either p or q.
2. The allocation should be `list q = alloc(struct list);`
3. We need to create the self-loop. Add after the last line in the function body: `q->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.

We start the tortoise and the hare both at the beginning of the list, the tortoise stepping by ones and hare by twos. If the hare hits `NULL`, it is not a valid sloop and we return `false`. If the hare catches the tortoise, they must be in a cycle. At that point, the tortoise moves forward by one and checks if it is still at the same place as the hare. If so we have a valid sloop and return `true`. Otherwise we return `false` since we have a longer cycle.

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) {
```

```
  return (n >> k) & ~(((1<<31)>>k)<<1)
```

```
}
```

```
// or, as a loop:
int iushr(int n, int k) {
  int i;
  k = k & 0x1F;
  for (i = 0; i < k; i++)
    n = (n>>1) & ~(1<<31);
  return n;
}
```

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) {
```

```
  int result = x+y;
  if (x > 0 && y > 0) assert(result > 0, "overflow");
  if (x < 0 && y < 0) assert(result < 0, "overflow");
  return result;
```

```
}
```

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 $S, \mathrm{iushr}(x, y)$, where iushr is the function defined in Task 1. You may use the `iushr` function.

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

    int y = pop(S);
    int x = pop(S);
    push(iushr(x,y),S);
    pc += 1;

}
```

Task 4 (5 pts). Implement the hypothetical instruction `oadd` (`0xBA`) which transforms the stack S, x, y to $S, \mathrm{oadd}(x, y)$, 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) {

    push(oadd(pop(S),pop(S)),S);
    pc += 1;

}
```

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) {

    int i = P[pc+1];
    int c = P[pc+2];
    if (c > 127) c = c-256;
    V[i] += c;
    pc += 3;

}
```

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

```
int binsearch_smallest(int x, int[] A, int n)
//@requires 0 <= n && n <= \length(A);
//@requires is_sorted(A,n);
/*@ensures (\result == -1 && !is_in(x, A, n))
        || (A[\result] == x && (\result == 0 || A[\result-1] < x));
 @*/
{ int lower = 0;
  int upper = n;
  while (lower < upper)
    //@loop_invariant 0 <= lower && lower <= upper && upper <= n;
    //@loop_invariant lower == 0 || A[lower-1] < x;
    //@loop_invariant upper == n || A[upper] >= x;
    { int mid = lower + (upper-lower)/2;
      if (A[mid] < x) lower = mid+1;
      else /*@assert(A[mid] >= x);@*/ upper = mid;
    }
  //@assert lower == upper;

  if (                                 )

    return lower;
  else
    return -1;
}
```