CIS 120 Midterm II November 10, 2017

SOLUTIONS

1. **OCaml Mutable Queues and the ASM** (28 points)

In this problem we consider a variant of the *doubly-linked queue* data structures that you saw in homework. A *circular queue* (whose type we will write as `'a ceque`) is another possible implementation of the queue abstraction. The nodes of a `ceque`, like those of a `deque`, contain both `next` and `prev` references that form a doubly-linked sequence. Appendix A gives the code for defining `ceque`s and shows two valid `int ceque` structures, as drawn using the conventions of the OCaml abstract stack machine.

As shown in those pictures, the difference between a `ceque` and a `deque` is that in a `ceque` the `next` pointer at the tail node of the queue, rather than being `None`, instead points back to the head node, and, symmetrically, the `prev` link at the head of the queue points back to the tail. Thanks to this circularity, a `ceque` doesn't need both `head` and `tail` references, it needs only a `head` reference—the `tail` node can be reached by following the `prev` pointer from the head.

(6 points) The *invariants* for a `ceque` are similar to those of `deque`s, except that they take into account the "wrap-around" from the tail of the queue to the head of the queue:

CEQUE INVARIANTS: A value `q : 'a ceque` satisfies the ceque invariants if either...

(1) The ceque is empty and `q.head` is `None`, or

(2) The ceque is non-empty, and `q.head = Some n1`, and

(i) `n1` is reachable from `n1` by following one or more 'next' pointers

(ii) `n1` is reachable from `n1` by following one or more 'prev' pointers

and, moreover, for every node `n` in the cdeque:

(iii) if `n.next = Some m` then `m.prev = Some mp` and `n == mp`

(iv) if `n.prev = Some m` then `m.next = Some mn` and `n == mn`

a. True ☐ False ☒ Comparing two non-empty `ceque` values `q1` and `q2` by using structural equality `q1 = q2` will always cause the program to go into an infinite loop (which might exhaust the stack).

b. True ☐ False ☒ If `q` satisfies the ceque invariants and `qn` is a `cqnode` reachable from `q`, it is possible that `qn.next` is `None`.

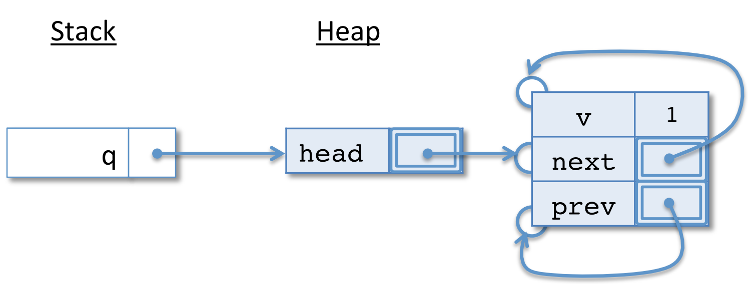
c. True ☒ False ☐ If `q` satisfies the ceque invariants and `q.head` is `Some n`, then it is possible that `n.next == n.prev` evaluates to **`true`**.

d. (8 points) The `insert` operation for ceques is supposed to add a new node to the tail of the queue (i.e. immediately `prev` to the `head` node), as shown by the examples in the Appendix. You wrote the following code for `insert`, but it seems to be buggy.

```
let broken_insert (x: 'a) (q: 'a ceque) : unit =
  let new_node = {v = x; next = None; prev = None} in
  begin match q.head with
  | None ->
     new_node.next <- Some new_node;
     new_node.prev <- Some new_node;
     q.head <- Some new_node

  | Some n ->
     new_node.next <- Some n;
     n.prev <- Some new_node;
     new_node.prev <- n.prev;
     begin match n.prev with
      | None -> failwith "invariant broken"
      | Some m -> m.next <- Some new_node
     end
  end
```

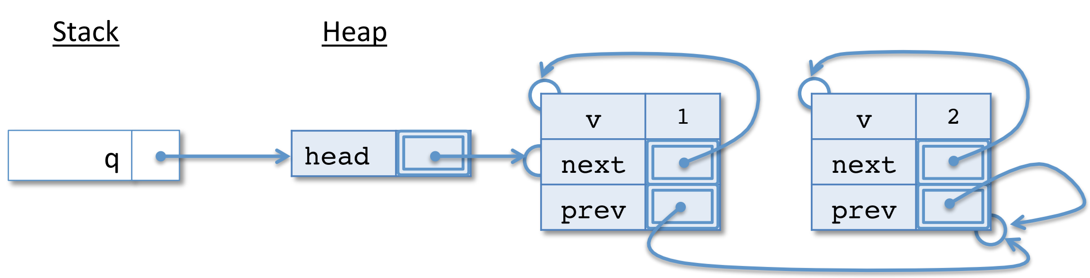
Your testing has revealed that creating a ceque of length 1 works fine, and that the code below correctly creates the ASM state as shown

```
let q = broken_insert 1 (create ())
```



Inserting a second value does not work. Complete the following ASM diagram so that it shows the results of running the following command starting from the state above:

```
;; broken_insert 2 q
```



e. (14 points) We can re-implement all of the queue operations for this `ceque` representation type, but we have to be a bit more careful with iteration because of the inherently cyclic structure of a valid `ceque`. Fill in the blanks below to implement the `length` operation, which should calculuate the number of `cqnode`s in a `ceque`.

- You may assume that the input `q` satisfies the ceque invariants. *If a case is impossible because of the invariants use* `failwith "invariants violated"`.
- The `loop` should be *tail recursive*.

```
let length (q: 'a ceque) : int =
  begin match q.head with
  | None -> 0
  | Some n ->
    let rec loop (qno: 'a cqnode option) (acc: int) : int =
      begin match qno with
      | None -> failwith "broken invariant"
      | Some m ->
        if m == n then acc else loop m.next (1+acc)
      end
    in
    loop n.next 1
  end
```

2. OCaml: Objects and Encapsulation (12 points)

Consider the following OCaml code, which uses `counter` object type. (We include the definition of `'a ref` as a reminder.)

```
type 'a ref = { mutable contents : 'a }

type counter = {
    reset : unit -> unit;
    incr : unit -> int;
  }

let mk_counters () : counter * counter =
  let cnt : int ref = {contents = 0} in
  let ctr : counter = {
      reset = (fun () -> cnt.contents <- 0);
      incr = (fun () -> cnt.contents <- cnt.contents + 1;
                   cnt.contents)
    }
  in (ctr, ctr)


let (ctr1, ctr2) = mk_counters ()
;; print_int (ctr1.incr ())
;; print_int (ctr1.incr ())
;; ctr2.reset ()
;; print_int (ctr1.incr ())
```

What will be printed to the terminal when the above program is run?

1 2 1

3. Java: Subtyping, Interfaces, and Inheritance (20 points)

Consider the following Java code that creates new classes and interfaces:

```
public class Animal {...}

public interface Cartoon {
   public void saySomething(String say);
   ...
}

public class WildAnimal extends Animal {...}

public class Bear extends WildAnimal {
   public void eatSomething(){...}
   ...
}

public class Lion extends WildAnimal {...}

public class CartoonLion extends Lion implements Cartoon {...}
```

Which of the following lines is legal Java code that will not cause any compile-time (i.e. type checking) errors? If it is legal code, just check the "Legal Code" box. If it is not legal, check "Not Legal" and explain why. (2 points each)

a. `Animal scar = new Animal();`

☒ Legal Code ☐ Not Legal

Reason for not legal: ____________________

b. `Cartoon pumbaa = new Cartoon();`

☐ Legal Code ☒ Not Legal

Reason for not legal: Cannot instantiate Interfaces

c. `Bear baloo = new WildAnimal();`

☐ Legal Code ☒ Not Legal

Reason for not legal: `WildAnimal` is a supertype of `Bear`

d. `WildAnimal baloo = new Bear();`

☒ Legal Code ☐ Not Legal

Reason for not legal: ____________________

e. `Lion simba = new CartoonLion();`

☒ Legal Code ☐ Not Legal

Reason for not legal: ____________________

For the following lines, recall that `Bear` has a method with the signature `public void eatSomething()` and that `Cartoon` had a method with the signature `public void saySomething(String say)`.

f. `Bear.eatSomething();`

☐ Legal Code ☒ Not Legal

Reason for not legal: `eatSomething` is not a static method

g.
```
Bear baloo = new Bear();
baloo.eatSomething();
```

☒ Legal Code ☐ Not Legal

Reason for not legal: ______________________________

h.
```
Animal baloo = new Bear();
baloo.eatSomething();
```

☐ Legal Code ☒ Not Legal

Reason for not legal: `eatSomething` is not a defined for `Animal`

i.
```
CartoonLion simba = new CartoonLion();
simba.saySomething("Hey, Uncle Scar, guess what?");
```

☒ Legal Code ☐ Not Legal

Reason for not legal: ______________________________

j.
```
CartoonLion simba = new CartoonLion();
simba.eatSomething();
```

☐ Legal Code ☒ Not Legal

Reason for not legal: `eatSomething` is not a defined for `CartoonLion`

4. Understanding Array Code (15 points)

Consider the following snippet of code in Java:

```java
1  int[] values = { 5, 4, 3, 1, 2 };
2  for (int i = 0; i < values.length; i++) {
3     int a = values[i];
4     int b = i;
5
6     for (int j = i; j < values.length; j++) {
7        int val = values[j];
8        if (val < a) {
9           a = val;
10          b = j;
11       }
12    }
13    int temp = values[i];
14    values[i] = a;
15    values[b] = temp;
16
17    // values?
18 }
```

a. (10 points) Consider the for loops. At the end of each iteration of the outer for loop (i.e., at line 19), what are the contents of the values array? Fill out the table below:

Iteration	Contents of the `values` array
1	`{ 1, 4, 3, 5, 2 }`
2	`{ 1, 2, 3, 5, 4 }`
3	`{ 1, 2, 3, 5, 4 }`
4	`{ 1, 2, 3, 4, 5 }`
5	`{ 1, 2, 3, 4, 5 }`

b. (3 points) Semantically, what does this snippet of code do?

It sorts the array in an ascending order.

c. (2 points) If the outer for loop went from `0` to `values.length - 1`, would the code still work correctly? Explain why.

Yes, it will still work correctly. The inner for loop does the swapping of values. Due to this, the outer for loop doesn't need to look at the last element in the array since it will be moved to the correct index by the inner for loop.

5. Writing and Testing Array Code (25 points)

For this part, you'll write Java code and JUnit tests that implements and test a `sub` method. This method returns a new array that is a subarray of the input array. The subarray begins at the specified `beginIndex` and extends to the element at index `endIndex - 1`. Thus the length of the returned subarray is `endIndex-beginIndex`. For invalid inputs (such as `beginIndex` being negative), the code should throw `IllegalArgumentException()`.

a. (10 points) Complete two JUnit tests for this code, being sure to give them descriptive names. We have given you two initial tests that should pass for your implementation. The first one you provide should test exceptional circumstances.

```
@Test
public void testNormalRange() {
   int[] array = { 1, 2, 3, 4, 5 };
   int[] ret = sub(array, 1, 3);
   int[] expected = {2, 3};
   assertArrayEquals(expected, ret);
}
@Test(expected=IllegalArgumentException.class)
 public void testNegativeIndex () {
   int[] array = { 1, 2, 3, 4, 5 };
   int[] ret = sub(array, -4, 3);
}

@Test(expected=IllegalArgumentException.class)
public void testNegativeRange() {

   int[] array = { 1, 2, 3, 4, 5 };
   int[] ret = sub(array, 4, 3);

   assertArrayEquals(array, ret);
}

@Test
public void testFullRange() {

   int[] array = { 1, 2, 3, 4, 5 };
   int[] ret = sub(array, 0, 5);

   assertArrayEquals(array, ret);
}
```

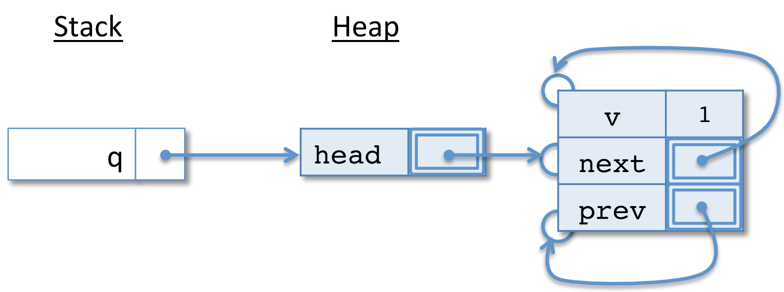
b. (15 points) Now implement the `sub` method based on the specifications and test(s) above. As a reminder, for invalid input (such as `beginIndex` being negative), the code should `throw new IllegalArgumentException()`.

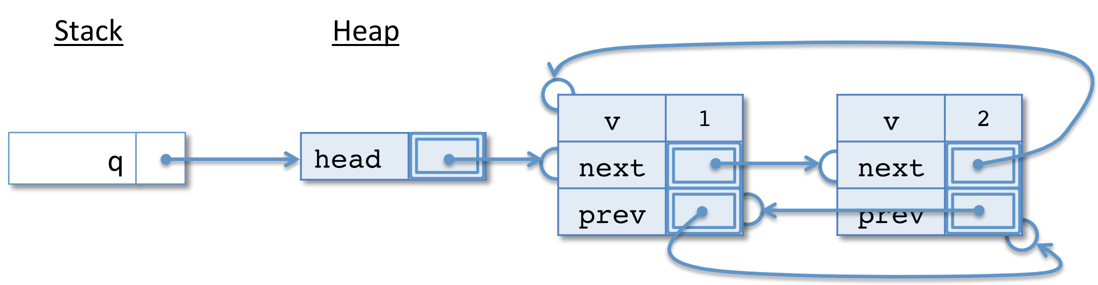
```
/**
 * Returns a new array that is a subarray of the input array
 * @param array the input array
 * @param beginIndex the start index, inclusive
 * @param endIndex the end index, exclusive
 * @return the specified subarray
 * @throws IllegalArgumentException if
 *    - array is null, or
 *    - beginIndex or endIndex are not valid for the given array
 */
public static int[] sub(int[] array, int beginIndex, int endIndex) {

   if (array == null || beginIndex < 0 ||
            endIndex > array.length || beginIndex > endIndex) {
      throw new IllegalArgumentException();
   }

   int[] ret = new int[endIndex - beginIndex];

   for (int i = 0; i < ret.length; i++) {
      ret[i] = array[i + beginIndex];
   }

   return ret;
}
```

1 Appendix: OCaml Ceque Implementation

```
type 'a cqnode = {
  v: 'a;
  mutable next: 'a cqnode option;
  mutable prev: 'a cqnode option;
}

type 'a ceque = {
  mutable head: 'a cqnode option;
}

let create () : 'a ceque =
  { head = None }

let is_empty (q: 'a ceque) : bool =
  q.head = None

let insert (x: 'a) (q: 'a ceque) : unit = (* omitted *)
```

Example valid ceque ASM states






```
let q = insert 1 (create ())
(* FIRST CEQUE ASM STATE *)
;; insert 2 q
(* SECOND CEQUE ASM STATE *)
```