CSE 428, Spring 2001. Pre-Test for preparation of Midterm 2

Exercise 1

Consider the following definitions of a class Queue in C++:

```
class Queue {

private:
   class Element {
   public:
      int info;
      Element* next;
      Element(int k, Element* n) { info = k; next = n; }
   };

   Element* front;
   Element* rear;

public:
   Queue() { front = rear = NULL; }
   ~Queue() { while (front!=NULL) pop(); }
   void insert(int k) {
      if (rear==NULL)
         rear = front = new Element(k,NULL);
      else
         rear = rear->next = new Element(k,NULL);
   }
   int pop() {
      if (front==NULL) { cout << "Empty queue!" ; return 0; }
      Element* temp = front;
      int n = front->info;
      front = front->next;
      if (front==NULL) rear = NULL;
      delete temp;
      return n;
   }
};
```

For each of the following declarations of `foo()`, say whether a call to `foo()` leaves memory leaks and/or dangling references. (Note: "a call to `foo()` leaves ml/dr" means "after a call to `foo()` returns, we have ml/dr".) For each dangling reference, specify whether it is to the stack or to the heap.

Assume that `p` is a global variable of type `Queue*` which before the call of `foo()` has value `NULL`.

1. `int foo(){ Queue q; q.insert(2); return q.pop(); }`

2. `int foo(){ Queue* q = new Queue(); q->insert(2); return q->pop(); }`

3. `void foo(){ Queue* q = new Queue(); q->insert(2); delete q; }`

4. `void foo(){ Queue* q = new Queue(); q->insert(2); p = q; }`

5. `int foo(){ Queue q; q.insert(2); p = &q; return p->pop(); }`

Exercise 2

Implement the class `Queue` of Exercise 1 in Java, in a way that supports multithread use (i.e. so that the same queue can be shared by more threads). Make sure that, when `pop()` is executed on a empty stack, the thread executing the `pop()` suspends, instead of printing an error message. The thread should be resumed when some other thread inserts an element in the queue.

Exercise 3

Suppose that the class Queue previously defined is enriched with the following methods, to insert and print an element, and to pop and print an element, respectively:

```
public synchronized void insertAndPrint(integer n){
   System.out.print("i" + n + " ");
   insert(n);
}

public synchronized void popAndPrint(){
   System.out.print("p" + pop() + " ");
}
```

Consider now the following Java program:

```
import Queue; // import the definition of the class Queue

class MyThread extends Thread{

   private Queue myQueue;
   private int myInt;

   public MyThread(Queue q, int n){ myQueue = q; myInt = n; }

   public void run(){
      sleep((long)(java.lang.Math.random()*1000)); // sleep for an arbitrary amount of time
      myQueue.insertAndPrint(myInt);
      sleep((long)(java.lang.Math.random()*1000)); // sleep for an arbitrary amount of time
      myQueue.popAndPrint();
   }
}

class Test {
   public static void main(String[] args){
      Queue q = new Queue();
      MyThread T1 = new MyThread(q,1);
      MyThread T2 = new MyThread(q,2);
      T1.start();
      T2.start();
   }
}
```

1. Show all the possible output sequences that this program can produce (in different executions).

2. Suppose that in the method `run()` the instruction `myQueue.popAndPrint();` is replaced by the instruction

   ```
   System.out.print("p" + myQueue.pop() + " ");
   ```

   Would this modification change the answer to previous question? Please explain.

Exercise 4

The problem of the sailing philosophers can be described as follows: there are a number of philosophers who try repeatedly to get a ride on a boat. Each philosopher has a certain weight, expressed by an integer number, and the boat has a certain capacity, also expressed by an integer number, representing the maximum weight that the boat can substain. If the boat has capacity c, and the philosophers already in the boat have total weight w, (where $c \geq w$), and there is another philosopher P who wants to get in the boat, and his weight is k, then P can get in the boat only if $k \leq c - w$, otherwise P will have to wait until some other philosopher gets out from the boat.

Define a class `Boat` which controls the access of the philosophers to the boat. In particular, `Boat` should have two methods `getIn(integer)` and `getOut(integer)` that are called by the philosophers when they want to get in the boat and get out from the boat, respectively. The integer parameter represents, in both cases, the weight of the philosopher who calls the method. The capacity should be a parameter of the constructor method of the class `Boat`.

You don't need to worry about Interrupted Exceptions.

Exercise 5

Consider the following ML declaration of a polimorphic datatype "bynary tree":

```
datatype 'a btree = empty | node of 'a btree * 'a * 'a btree;
```

Some examples of this structure are:

```
     t1              t2               t3                 t4

     1             [1,2,3]           true              (1,2)
    / \             /     \          /  \                / \
   2   3          [4]    [5,6]    false  true        (3,4) (5,6)
  /   / \         / \                /      \          /
 4   5   6      [7] [8]           true     false     (7,8)
```

Consider now the following extension of the function "reduce" to trees:

```
fun reduce f v empty = v
  | reduce f v (node(t1,a,t2)) = f(reduce f v t1, f(a, reduce f v t2));
```

Say what is the result of each of the following expressions, including error, where `t1`, `t2`, `t3` and `t4` are the trees represented in the figure above. Remember that `@` is the append on lists, `andalso` is the boolean conjunction, and the function `op` transforms an infix operator into the corresponding prefix operator.

1. `reduce (op +) 0 t1`

2. `reduce (op ::) [] t1`

3. `reduce (fn (x,y) => if x > y then x else y) 0 t1`

4. `reduce (op @) [] t2`

5. `reduce (op andalso) true t3`

6. `reduce (fn ((x1,y1),(x2,y2)) => (x1+x2, y1+y2)) (0,0) t4`

Exercise 6

Define in ML a function `positions: ('a -> true) -> 'a list -> int list` which, given property `p`, and a list of elements `L`, it returns the list of the positions in `L` of the elements which satisfy `p`. You can define auxiliary functions if you like.

For example, the function should return the following results:

```
- positions (fn x => x > 0) [0,1,~1,2];
val it = [2,4] : int list

- positions (fn x => x = "b") ["a", "b", "c", "d", "e", "f"];
val it = [2] : int list

- positions (fn x => length x = 2) [[1],[],[1,2,3]]; (* length x is the length of the list x *)
val it = [] : int list
```

Exercise 7

Define in ML a function `apply_repeatedly: int -> ('a -> 'a) -> 'a -> 'a` such that, given a non-negative integer `n`, a function `f`, and an element `x`, `apply_repeatedly n f x` gives as result `f(f(...f(x)...))`, where the application of `f` is repeated `n` times (if the value of `n` is 0, then the result is `x`).