**Πανεπιστήμιο Κρήτης**
**Τμήμα Επιστήμης Υπολογιστών**

**ΗΥ-252 – Οντοκεντρικός Προγραμματισμός**
**Βασίλης Χριστοφίδης**

**Τελική Εξέταση (3 ώρες)**
**Ημερομηνία: 25 Ιουνίου 2004**

**Ονοματεπώνυμο:**
**Αριθμός Μητρώου:**

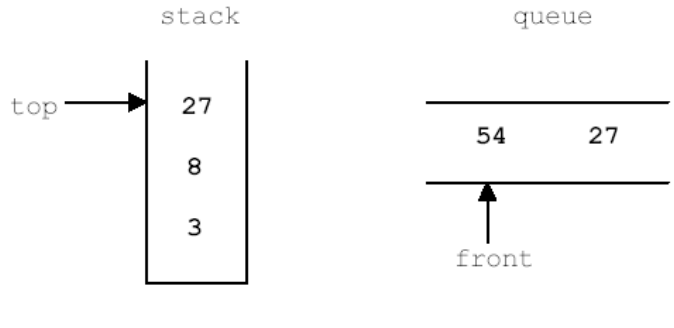
## Άσκηση 1 (50 μονάδες)

Υποθέστε ότι έχουμε μια στοίβα (`Stack`) και μια ουρά (`Queue`) από ακέραιους αριθμούς. Για την άσκηση αυτή χρησιμοποιήστε τα συμβόλαια (δηλ. τις μεθόδους) των ΑΤΔ (ADT) στοίβα και ουρά που σας έχουν δοθεί στις διαλέξεις του μαθήματος.

**1.1. (5 μονάδες)** Σχεδιάστε το περιεχόμενο της στοίβας και της ουράς με την εκτέλεση της παρακάτω ακολουθίας εντολών. Σημειώστε στα διαγράμματά σας την οροφή (`top`) της στοίβας και την κεφαλή ουράς (`front`).

```
Stack stack = createStack();
Queue queue = createQueue();
stack.push(3);
queue.enqueue(8);
stack.push(queue.peek());
stack.push(27);
stack.push(54);
queue.dequeue();
queue.enqueue(stack.pop());
queue.enqueue(stack.peek());
```

**Λύση:**




1.2. **(5 μονάδες)** Υλοποιήστε σε ψευτοκώδικα μία μη-αναδρομική μέθοδο `reverse(Queue queue)` η οποία αντιστρέφει τα στοιχεία της ουράς που δίνεται σαν παράμετρος χρησιμοποιώντας αποκλειστικά μια στοίβα (δηλ. χωρίς να χρησιμοποιείτε κάποια άλλη δομή όπως πίνακες).

**Λύση:**

```
public static reverse(Queue queue) {
   Stack stack = createStack();
   while (!queue.isEmpty())
          stack.push(queue.dequeue());
   while (!stack.isEmpty())
          queue.enqueue(stack.pop());
   return queue;
}
```

**1.3. (10 μονάδες)** Δώστε σε μορφή ψευτοκώδικα μια μη-αναδρομική μέθοδο `isInL(String str)` που χρησιμοποιεί αποκλειστικά μία στοίβα (δηλ. χωρίς να απαριθμεί τους χαρακτήρες) για να αναγνωρίζει συμβολοσειρές που ανήκουν στην γλώσσα $L = \{A^{2^n}B^n : n >= 0\} = \{\epsilon, AAB, AAAABB, AAAAAABBB, \ldots\}$. Η μέθοδος επιστρέφει αληθές (ή ψευδές) αν η συμβολοσειρά που δίνεται σαν παράμετρος ανήκει (ή όχι) στην L.

**Λύση:**

```
// Returns true if str is in L, false otherwise
public static boolean isInL(String str) {
  if (str is an empty string)
    return true;
  else {
    stack.createStack();
    i = 1
    ch = i-th character of str
    while (ch is the character 'A' and
           i < length of str ) {
           stack.push(ch)
           i = i + 1;
           ch = i-th character of str;
         }
    while (i <= lengthof str) {
          ch = i-th character of str;
          if (ch is the character 'B') {
             if (!stack.isEmpty()) {
                 stack.pop();
                 if (!stack.isEmpty())
                    stack.pop();
                 else
                   return false; //only one balanced A
                }
             else
                 return false; //no balanced A's
             }
          else
             return false; //ch is not B
         }
    if (stack.isEmpty())
        return true;
    else
        return false;
  }
}
```

**1.4. (20 μονάδες)** Υλοποιήστε σε ψευτοκώδικα μία μη-αναδρομική μέθοδο `infix2postfix(String expression)` η οποία μετατρέπει την αριθμητική έκφραση που δίνεται σαν παράμετρος από την μορφή του ενθέματος (infix) στην μορφή του επιθέματος (postfix) χρησιμοποιώντας αποκλειστικά μια στοίβα. Για παράδειγμα, η έκφραση ενθέματος a*(b/c*d)+e θα πρέπει να μετατρέπεται σύμφωνα με την μέθοδο της άσκησης στην postfix έκφραση abc/d**e+ (οι κενοί χαρακτήρες μπορούν να αγνοηθούν). Για κάθε χαρακτήρα της έκφρασης ενθέματος που διαβάζετε στην είσοδο (π.χ. a*(b/c*d)+e), το περιεχόμενο της στοίβας που χρησιμοποιεί η μέθοδός σας καθώς και η έκφραση επιθέματος που τυπώνεται στην έξοδο, σας δίνονται ακολούθως για το παράδειγμα της άσκησης:

| Character of the Infix Expression | stack (bottom to top) | Postfix Expression |
|---|---|---|
| a | | a |
| * | * | a |
| ( | *( | a |
| b | *( | ab |
| / | *(/ | ab |
| c | *(/ | abc |
| * | *(<br>*(* | abc/ |
| d | *(* | abc/d |
| ) | *(<br>* | abc/d* |
| + | <br>+ | abc/d** |
| e | + | abc/d**e abc/d**e+ |

**Λύση:**

```
String infix2postfix(String expression) {
      Stack stack = new Stack();
      // the postfix expression to be returned
      String postfix = "";
      // tokenize the string
      StringTokenizer st = new StringTokenizer(expression);
      while (st.hasMoreTokens()) {
            String token = st.nextToken();
            if (token.isOperand()) { // is operand (1..9, a..z)
                  // move operands to the postfix expression
                  postfix = postfix + token;
            } else if (token.isLeftParenthesis()) {
            // is left parenthesis (
                  stack.push(token);
            } else if (token.isRigthParenthesis()) {
            // is right parenthesis )
                  // shift operators before the right parenthesis
                  // from the stack to the postfix expression
                  // until the corresponding left parenthesis is
                  // found
                  do (op = stack.pop()) {
                        if (!op.isLeftParenthesis())
                              postfix = postfix + op;
                  } while (!op.isLeftParenthesis())
            } else { //is an operator but not a parenthesis
                  // shift operators with higher or equal
                  // priority from the stack to the postfix
                  // expression
                  while (!stack.isEmpty() &&
```

```
                        stack.peek().priorityHigherOrEqual(token)) {
                           op = stack.pop();
                           postfix = postfix + op;
                     }
               }
         }
         // shift any remaining operators
         // from the stack to the postfix expression
         while (!stack.isEmpty()) {
               op = stack.pop();
               postfix = postfix + op;
         }
}
```

**1.5. (5 μονάδες)** Μοιάζει η συμπεριφορά μιας λίστας γεγονότων στον γεγονοστροφή (event-driven) προγραμματισμό με την συμπεριφορά του ΑΤΔ ουρά; Δικαιολογήστε σύντομα την απάντησή σας.

**Λύση:**
No, because an event list does not exhibit the first-in first-out behavior. Insertion into a queue must be performed at the back of the queue, but insertion into an event list could be done, for example, at the front of the event list, if the event to be inserted has the earliest time compared to all events already in the event list.

**1.6. (5 μονάδες)** Ποιο είναι το σημαντικότερο μειονέκτημα στην υλοποίηση του ΑΔΤ Ουρά χρησιμοποιώντας τον ΑΤΔ Λίστα; Δικαιολογήστε σύντομα την απάντησή σας.

**Λύση:**
The main disadvantage of implementing the ADT Queue using the ADT List is that the enqueue operation would be inefficient in sense that it runs in linear time, not constant time. This is because we have to traverse the entire linked list, starting from the beginning of the list, to reach the back of the list (assuming that the ADT List uses the reference-based implementation).

## Άσκηση 2 (45 μονάδες)

**2.1. (5 μονάδες)** Δώστε την μέθοδο **`intersection(TreeSet u, TreeSet v)`** η οποία υπολογίζει την τομή δύο συνόλων υλοποιημένων σύμφωνα με την κλάση Java **`TreeSet`** (μην χρησιμοποιήσετε την μέθοδο **`retainAll (Collection c)`**):

```
// Post: Returns a java.util.TreeSet which is an
// intersection (the mathematical set intersection) of u
// and v. u and v should remain intact.
public static TreeSet intersection (TreeSet u, TreeSet v) {
// your code here
}
```

**Λύση:**

```
Set w = new TreeSet();
Iterator itr = u.iterator();
while (itr.hasNext()) {
      Object o = itr.next();
```

```
      if (v.contains(o)) {
          w.add(o);
      }
    }
return w;
```

**2.2. (5 μονάδες)** Ποια είναι η πολυπλοκότητα (complexity) του χρόνου εκτέλεσης (στην χειρότερη περίπτωση) της μεθόδου **intersection** που υλοποιήσατε στο προηγούμενο ερώτημα; Δικαιολογήστε σύντομα την απάντησή σας.

**Λύση:**
$O(n \log_2 n)$ where n is maximum of the sizes of u and v. For each element in u (that is where the first n comes from), it checks to see if it is contained in v and it adds it to w if it is. v being a `TreeSet` object, the `contains` method is guaranteed to work in $O(\log_2 m)$ worst case time where m is the size of v. Similarly for `add`.

**2.3. (5 μονάδες)** Δώστε μια εναλλακτική υλοποίηση της μεθόδου **intersection(HashSet u, HashSet v)** για σύνολα υλοποιημένα σύμφωνα με την κλάση Java **HashSet** (μην χρησιμοποιήσετε την μέθοδο **retainAll (Collection c)**):

```
// Post: Returns a java.util.HashSet which is an
// intersection (the mathematical set intersection) of u
// and v. u and v should remain intact.
public static HashSet intersection (HashSet u, HashSet v) {
// your code here
}
```

**Λύση:**
```
Set w = new HashSet();
Iterator itr = u.iterator();
while (itr.hasNext()) {
      Object o = itr.next();
      if (v.contains(o)) {
          w.add(o);
      }
    }
return w;
```

**2.4. (5 μονάδες)** Ποια είναι η πολυπλοκότητα (complexity) του χρόνου εκτέλεσης (στην μέση περίπτωση) της μεθόδου **intersection** που υλοποιήσατε στο προηγούμενο ερώτημα; Δικαιολογήστε σύντομα την απάντησή σας.

**Λύση:**
$O(n)$ where n is the size of u. For each element in u (that is where the n comes from), it checks to see if it is contained in v and it adds it to w if it is. v being a `HashSet` object, the `contains` method is guaranteed to work in $O(1)$ average time. Similarly for `add`.

**2.5. (25 μονάδες)** Υποθέστε ότι έχουμε μία κλάση Σύνολο αντικειμένων (`Set`) η οποία υλοποιεί την διεπαφή `Comparable`. Ποια από τις παρακάτω υλοποιήσεις της στατικής μεθόδου **`max`**, η οποία επιστρέφει το μέγιστο στοιχείο του συνόλου, προσφέρει τις καλύτερες επιδόσεις εκτέλεσης για μεγάλα σύνολα; Δικαιολογήστε σύντομα την απάντησή σας, δίνοντας για κάθε υλοποίηση την πολυπλοκότητα (complexity) του χρόνου εκτέλεσής της (στην χειρότερη περίπτωση).

_____ A.
```
public static Object max (Set s) {
  List lst = new ArrayList (s);
  Collections.sort (lst);
  return lst.get (lst.size() - 1);
}
```

**Λύση:**
// Version A does a sort, which requires `O(n lg n)` time.

_____ B.
```
public static Object max (Set s) {
  Iterator it = new TreeSet (s).iterator();
  Object result = it.next();
  while (it.hasNext())
        result = it.next();
  return result;
}
```

**Λύση:**
// Version B creates a `TreeSet`, which requires `O(n lg n)` time.

_____ C.
```
public static Object max (Set s) {
  return new TreeSet (s).last();
}
```

**Λύση:**
// Version C creates a `TreeSet`, which requires `O(n lg n)` time.

_____ D.
```
public static Object max (Set s) {
  List lst = new ArrayList (s);
  Collections.sort (lst);
  Collections.reverse (lst);
  return lst.get (0);
}
```

**Λύση:**
// Version D does a sort, which requires **`O(n lg n)`** time.

XXXXX E.
```
public static Object max (Set s) {
   Iterator it = s.iterator();
   Object result = it.next();
   while (it.hasNext()) {
         Comparable x = (Comparable) it.next();
         if (x.compareTo (result) > 0)
            result = x;
      }
   return result;
}
```

**Λύση:**
// Version E visits every element of the set, which requires `O(n)` time, but does
// nothing else. This should be the fastest for large n.

## Άσκηση 3 (15 μονάδες)

Τι εκτυπώνεται στην στάνταρ έξοδο μετά την εκτέλεση του παρακάτω προγράμματος;

```
import java.util.*;
import java.io.*;
/* an object for storing a fraction */
public class Ratio {
  protected int numerator; // numerator of ratio
  protected int denominator; // denominator of ratio
  public Ratio(int top, int bottom) throws NumberFormatException{
  /* pre: bottom != 0
     post: constructs a ratio equivalent to top/bottom */
     if (bottom == 0) throw new NumberFormatException("1");
     this.numerator = top;
     this.denominator = bottom;
   }
  public int getNumerator() {
  /* post: return the numerator of the fraction */
     return this.numerator;
   }
  public int getDenominator() {
  /* post: return the denominator of the fraction */
     return this.denominator;
   }
  public double value() {
  /* post: returns the real value equivalent to ratio */
     return (double)this.numerator/(double)this.denominator;
   }
  public Ratio add(Ratio other) throws Exception{
  /* pre: other is non-null
     post: return new fraction - the sum of this and other */
     if (other == null) throw new Exception("Other is null");
         return new Ratio(this.numerator*other.denominator+
                          this.denominator*other.numerator,
                          this.denominator*other.denominator);
   }
  public static void main(String[] args) {
     Ratio r1=null;
     try{
        Ratio r = new Ratio(1,1);
        System.out.println(r.value());
        r = new Ratio(1,0);
        System.out.println(r.value());
        r.add(r1);
        System.out.println(r.value());
        r = r.add(new Ratio(1,0));
        System.out.println(r.value());
        }
     catch(NumberFormatException Message) {
        System.out.println("PRINTING FROM NUMBER FORMAT CATCH ");
        System.out.println("I Caught a Number Exception: " +
                                                    Message);
        Ratio r = new Ratio(1,1);
        System.out.println(r.value());
      }
```

```
     catch(Exception Message) {
        System.out.println("PRINTING FROM EXCEPTION CATCH ");
        System.out.println("Exception "+ Message);
        Ratio r = new Ratio(1,2);
        System.out.println(r.value());
       }
    finally {
        System.out.println("PRINTING FROM FINALLY ");
        Ratio r = new Ratio(1,1);
        System.out.println(r.value());
        System.out.println("Finally the exam is over ");
       }
    Ratio r = new Ratio(1,2);
    System.out.println(r.value());
  }
}
```

**Λύση:**

OUTPUT: (in the order in which things are printed, possibly more than 10)

```
Print#1: 1.0
Print#2: PRINTING FROM NUMBER FORMAT CATCH
Print#3: I Caught a Number Exception: 1
Print#4: 1.0
Print#5: PRINTING FROM FINALLY
Print#6: 1.0
Print#7: Finally the exam is over
Print#8: 0.5
Print#9:
Print#10:
```

NOTE: No marks if try block continued to be executed after first exception was caught.

## Άσκηση 4 (20 μονάδες)

Σας δίνεται η παρακάτω μέθοδος **compareTo**, η οποία χρησιμοποιείται για την λεξικογραφική ταξινόμηση συμβολοσειρών, π.χ. {aa,b,a,ac} διατάσσεται σε {a,aa,ac,b}. Τροποποιήστε την μέθοδο **compareTo** έτσι ώστε να χρησιμοποιείται για την ταξινόμηση συμβολοσειρών κατά αύξουσα σειρά του μήκους τους ενώ παράλληλα να διατηρείται η λεξικογραφική διάταξη των συμβολοσειρών ίδιου μήκους π.χ. {aa,b,a,ac} διατάσσεται σε {a,b,aa,ac}.

```
public int compareTo(String string) {
   int len1 = this.count;
   int len2 = string.count;
   int n = Math.min(len1, len2);
   char v1[] = this.value;
   char v2[] = string.value;
   int i = this.offset;
   int j = string.offset;
   while (n-- != 0) {
       char c1 = v1[i++];
       char c2 = v2[j++];
       if (c1 != c2) {
```

```
            return c1 - c2;
        }
      }
    return len1 - len2;
}
```

**Λύση:**

```
public int compareTo(String string) {
  int len1 = this.count;
  int len2 = string.count;
  ------INSERT HERE -------
  if (len1 != len2) return(len1 - len2);
  ---DELETE THIS STATEMENT int n= Math.min(len1,len2);
  char v1[] = this.value;
  char v2[] = string.value;
  int i = this.offset;
  int j = string.offset;
  ---CHANGE while (n-- != 0) { -- TO:
  while(len1-- != 0) {
       char c1 = v1[i++];
       char c2 = v2[j++];
       if (c1 != c2) {
           return c1 - c2;
       }
     }
  ---CHANGE return len1 - len2; -- TO:
  return (0);
}
```