Université d'Ottawa
Faculté de génie

École d'ingénierie et de
technologie de l'information

uOttawa

L'Université canadienne
Canada's university

University of Ottawa
Faculty of Engineering

School of Information
Technology and Engineering

Introduction à l'informatique II (CSI 1501)
Examen final

Professeur: Marcel Turcotte

Avril 2005, durée: 3 heures

Identification

Nom, prénom : _____ position désignée : _____

Signature : _____ numéro d'étudiant : _____

Consignes

1. Livres fermés ;
2. Sans calculatrice ou toute autre forme d'aide ;
3. Répondez sur ce questionnaire, utilisez le verso des pages si nécessaire, mais vous ne pouvez remettre aucune page additionnelle ;
4. Écrivez lisiblement, votre note en dépend ;
5. Commentez vos réponses ;
6. Ne retirez pas l'agrafe.

Barème

Question	Maximum	Résultat
1	20	
2	20	
3	10	
4	15	
5	15	
6	10	
7	5	
8	5	
Total	**100**	

Question 1 : Questions sans développement (20 points)

A. L'algorithme vu en classe pour l'évaluation d'expressions en notation suffixée (*postfix expressions*, aussi appelées expressions RPN, notation polonaise inversée) utilise une **pile** ou une **file** comme structure de données principale ? (2 points)

B. Complétez le tableau ci-bas. Pour chaque technique d'implémentation d'une liste chaînée, écrivez la lettre **C** dans la case du tableau si le nombre d'opérations effectuées par la méthode est constant, c'est-à-dire que le nombre d'opérations est indépendant du nombre d'éléments se trouvant présentement dans la structure de données au moment de l'appel, dans ce cas-ci la méthode est efficace. Sinon, inscrivez la lettre **V** si le nombre d'opérations varie en fonction du nombre d'éléments contenus dans la liste, dans ce cas-ci la méthode n'est pas efficace. Ne considérez que les cas généraux (et non pas les cas spécifiques). (4 points)

Implémentation	**addLast(Object o)**	**removeLast(Object o)**
Simplement chaînée sans pointeur arrière		
Simplement chaînée avec pointeur arrière		
Doublement chaînée sans pointeur arrière		
Doublement chaînée avec pointeur arrière		

En classe, le pointeur arrière a souvent été désigné par la variable **tail**.

C. Donnez la représentation binaire du nombre décimal sans signe suivant. (4 points)

$$(101.3125)_{10} = (\qquad\qquad\qquad\qquad)_2$$

D. Effectuez l'addition des deux nombres binaires sans signe suivants. (4 points)

$$\begin{array}{cccccccccc} & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 \\ + & 1 & 1 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ \hline \end{array}$$

E. Corrigez les trois erreurs (communes) dans l'implémentation de la méthode **add** ci-bas. Faites l'hypothèse que les paramètres sont valides, c'est-à-dire que $0 \leq$ **pos** $<$ **size** et **obj** $\neq$ **null**. (6 points)

```
public class LinkedList extends Object {

    private static class Elem {
        private Object value;
        private Node next;
        private Elem( Object value, Elem next ) {
            this.value = value;
            this.next = next;
        }
    }

    private Elem head;

    public void add( int pos, Object obj ) {

        if ( pos == 0 ) {

           head = new Elem( obj, null );

        } else {

           Elem p = head;

           for ( int i = 0; i < ( pos-1 ); i++ ) {
              p++;
           }

           p = new Elem( obj, p.next );
        }
   }
}
```

Question 2 : Queue (20 points)

Cette question porte sur le type abstrait de données (TAD) file (*queue*). Voici l'interface utilisée pour cette question.

```
public interface Queue {
    /** Retourne vrai si cette file est vide.
     *  @return true si la file est vide.
     */
    public abstract boolean isEmpty();

    /** Ajoute un élément à l'arrière de la file.
     *  @throws FullQueueException si cette file est pleine.
     */
    public abstract void enqueue( Object o ) throws FullQueueException;

    /** Retire et retourne l'élément avant de cette file.
     *  @return l'élément avant de cette file.
     *  @throws EmptyQueueException si cette file est vide.
     */
    public abstract Object dequeue() throws EmptyQueueException;

    /** Retire l'élément avant de cette file et l'insère à l'arrière.
     *  @throws EmptyQueueException si cette file est vide.
     */
    public abstract void requeue() throws EmptyQueueException;
}
```

Nous considérons deux implémentations de cette interface. La première implémentation est **LinkedQueue**. Elle utilise une liste simplement chaînée de noeuds afin de sauvegarder les éléments de la file. Afin d'être efficace, l'élément avant de la file est sauvegardé dans le premier noeud de la liste alors que l'élément arrière se trouve à la dernière position et est désigné par le pointeur arrière.

La seconde implémentation est **CircularQueue**. Elle utilise un tableau circulaire afin de sauvegarder les éléments de la file. La variable d'instance **front** désigne l'élément avant de la file alors que la variable d'instance **rear** désigne l'élément arrière. La file vide est représentée par l'assignation de la valeur -1 aux variables d'instance **front** et **rear**.

L'implémentation partielle de chacune des deux classes contient toute les méthodes de l'interface à l'exception de la méthode **requeue**. Vous devez donc écrire la méthode **requeue** pour les deux implémentations.

A. Pour l'implémentation partielle ci-bas de la classe **LinkedQueue**, ajoutez la méthode **requeue** (consultez l'interface **Queue** pour en connaître la définition). Bien que toutes les méthodes de l'interface existent, vous ne pouvez pas les utiliser. Vous ne pouvez pas simplement échanger les valeurs des deux noeuds. Il faut changer les liens qui unissent les noeuds afin de changer la structure de la liste de sorte que le premier noeud devienne le dernier. (10 points)

```
public class LinkedQueue implements Queue {
    private static class Elem { // les noeuds de la liste
        private Object value;
        private Elem next;
        private Elem( Object value, Elem next ) {
            this.value = value;
            this.next = next;
        }
    }
    private Elem front;
    private Elem rear;
    public void enqueue( Object obj ) { ... }
    public Object dequeue() throws EmptyQueueException { ... }
    public boolean isEmpty() { ... }

    public void requeue() throws EmptyQueueException {









    } // End of requeue()
} // End of LinkedQueue
```

B. Pour l'implémentation partielle de la classe **CircularQueue** ci-bas, vous devez ajouter la méthode **requeue** (consultez l'interface **Queue** afin d'en connaître la définition). Bien que toutes les méthodes de l'interface existent vous ne pouvez pas les utiliser, ainsi qu'aucune autre méthode de cette classe, entre autre, la méthode **nextIndex** n'a pas été définie. (10 points)

```
public class CircularQueue implements Queue {
    public static final int DEFAULT_CAPACITY = 100;
    private final int MAX_QUEUE_SIZE;
    private Object[] q; // sauvegarde les éléments de cette file
    private int front, rear;
    public CircularQueue() {
        this( DEFAULT_CAPACITY );
    }
    public CircularQueue( int capacity ) {
        MAX_QUEUE_SIZE = capacity;
        q = new Object[ MAX_QUEUE_SIZE ];
        front = -1; // Represents the empty queue
        rear = -1; // Represents the empty queue
    }
    public boolean isEmpty() { ... }
    public void enqueue( Object o ) throws FullQueueException { ... }
    public Object dequeue() throws EmptyQueueException { ... }

    public void requeue() throws EmptyQueueException {









    } // End of requeue()
} // End of CircularQueue
```

```
private static void test( Queue q ) {
    for ( int i=0; i<5; i++ ) {
        q.enqueue( new Integer( i ) );
    }
    for ( int i=0; i<=5; i++ ) {
        System.out.println( "before: " + q );
        q.requeue();
        System.out.println( "after:  " + q );
        System.out.println();
        q.dequeue();
    }
}
```

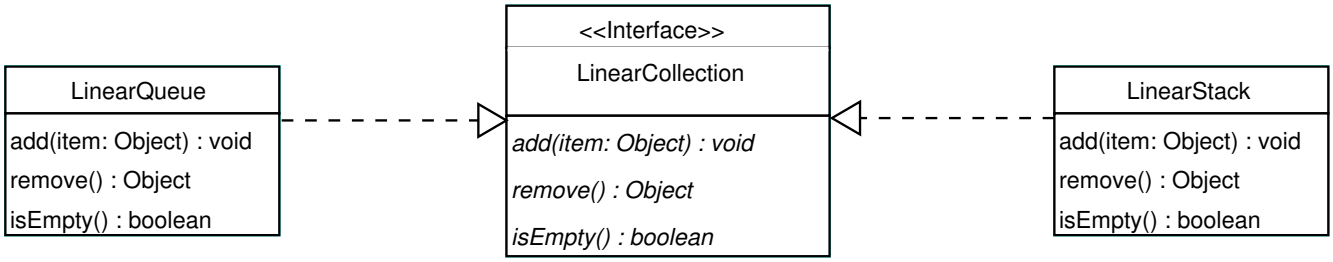
Les appels de méthode suivants : **test(new LinkedQueue())** et **test(new CircularQueue())**, doivent afficher le résultat suivant (vous n'avez pas à implémenter la méthode **toString** utilisée afin de produire cette sortie).

```
before: [0,1,2,3,4]
after:  [1,2,3,4,0]

before: [2,3,4,0]
after:  [3,4,0,2]

before: [4,0,2]
after:  [0,2,4]

before: [2,4]
after:  [4,2]

before: [2]
after:  [2]

before: []
Exception in thread "main" EmptyQueueException
```

Question 3 : Labyrinth (10 points)

Pour cette question, nous définissons une version simplifiée de la classe **Dungeon** du devoir 4. En particulier, il n'y a pas de créatures, d'armures ou d'armes. Nous nommons cette implémentation **Labyrinth** afin de la distinguer de celle du devoir 4.

De plus, comme nous l'avons vu en classe, nous pouvons utiliser une pile ou une file afin de sauvegarder les solutions partielles dans la méthode **solve**. Évidement, le comportement de l'algorithme change selon la structure de données utilisée. Pour cette question, vous devez identifier le bon comportement.



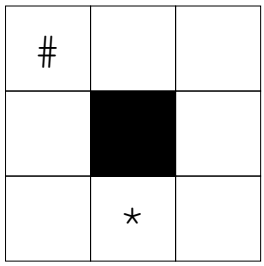
Nous définissons l'interface **LinearCollection** ayant trois méthodes : **void add(Object item)**, **Object remove()** et **boolean isEmpty()**, et nous considérons deux implémentations de cette interface : **LinearQueue** et **LinearStack**.

La classe **LinearQueue**, comme tout autre implémentation d'une file, est telle que la méthode **add** ajoute l'**item** à l'arrière de la file, la méthode **remove** retire l'élément avant de la file, et la méthode **isEmpty** retourne vrai si la file est vide.

La classe **LinearStack**, comme toute autre implémentation d'une pile, est telle que la méthode **add** ajoute l'**item** sur le dessus de la pile, la méthode **remove** retire l'élément du dessus de la pile, et la méthode **isEmpty** retourne vrai si cette pile est vide.

Comme vous le voyez sur la page qui suit, pour cette question, la méthode **solve** a été modifiée afin que l'on puisse passer en paramètre la structure de données qui sera utilisée afin de sauvegarder les solutions partielles.

Le labyrinthe suivant a été assigné à la variable d'instance **grid**. Le symbole `#`, en position $(0, 0)$, indique le point de départ alors que le symbole `*`, en position $(2, 1)$, indique le but (destination, le trésor).



L'appendice A présente la version complète de la classe **Labyrinth**.

```
public class Labyrinth {

    public static final char[] MOVES = { DOWN, UP, RIGHT, LEFT };

    private char[][] grid = { { '#', ' ', ' ' },
                              { ' ', '-', ' ' },
                              { ' ', '*', ' ' } };

    // (...)

    public String solve( LinearCollection solutions ) {

        String candidate = "";
        boolean solved = false;
        solutions.add( candidate );

        while ( ! solved && ! solutions.isEmpty() ) {

            String path = (String) solutions.remove();

            for ( int m=0; m < MOVES.length && ! solved; m++ ) {

                candidate = path + MOVES[ m ];

                if ( checkPath( candidate ) ) {

                    System.out.println( candidate );

                    if ( reachesGoal( candidate ) )
                        solved = true;
                    else
                        solutions.add( candidate );

                }
            }
        }
        if ( ! solved )
            return null;
        return candidate;
    }
}
```

A. Laquelle des sorties suivantes correspond à l'exécution des énoncés suivants.

```
Labyrinth labyrinth = new Labyrinth();
String solution = labyrinth.solve( new LinearStack() );
if ( solution == null ) {
    System.out.println( "This labyrinth has no solution!" );
} else {
    System.out.println( "Solution = " + solution );
}
```

(a)
```
R
D
DD
DDR
Solution = DDR
```

(b)
```
D
R
DD
RR
DDR
Solution = DDR
```

(c)
```
R
D
RR
DD
RRD
DDR
Solution = DDR
```

(d)
```
D
R
RR
RRD
RRDD
RRDDL
Solution = RRDDL
```

(e)
```
D
R
RD
RR
RRD
RRDD
RRDL
RRDLD
Solution = RRDLD
```

(f)
```
This labyrinth has no solution!
```

B. Laquelle des sorties suivantes correspond à l'exécution des énoncés suivants.

```
Labyrinth labyrinth = new Labyrinth();
String solution = labyrinth.solve( new LinearQueue() );
if ( solution == null ) {
    System.out.println( "This labyrinth has no solution!" );
} else {
    System.out.println( "Solution = " + solution );
}
```

(a)
```
R
D
DD
DDR
Solution = DDR
```

(b)
```
D
R
DD
RR
DDR
Solution = DDR
```

(c)
```
R
D
RR
DD
RRD
DDR
Solution = DDR
```

(d)
```
D
R
RR
RRD
RRDD
RRDDL
Solution = RRDDL
```

(e)
```
D
R
DD
DR
RD
RR
DDR
Solution = DDR
```

(f)
```
This labyrinth has no solution!
```

Question 4 : Polynomial (15 points)

Pour cette question, nous représentons un polynôme à l'aide d'une liste ordonnée de termes. Un terme est constitué d'un coefficient et d'un exposant. Les termes sont ordonnés en fonction de leur exposant. L'appendice B présente une définition complète de la classe **Term**, l'appendice C présente l'interface **OrderedList**, finalement, l'appendice D présente l'interface **Iterator**. Faites l'hypothèse qu'il existe une classe nommée **OrderedLinkedList** qui réalise l'interface **OrderedList**. La méthode **iterator** retourne un itérateur valide pour cette liste. L'itérateur est implémenté à l'aide de la technique "fail-fast" vue en classe. Écrivez la classe **Polynomial** en suivant les instructions suivantes.

A. Un polynôme est une liste ordonnée de termes ;

B. Écrivez la méthode **void add(Term t)** ; celle-ci ajoute le terme **t** au polynôme. S'il y a un terme ayant le même exposant alors le coefficient de **t** est ajouté au coefficient de ce terme ;

C. Écrivez la méthode **double evaluate(double x)** ; elle évalue le polynôme pour la valeur **x**.

Les énoncés ci-bas ont pour but de créer le polynôme $2x^3 + 4x + 5$, qui est ensuite évalué pour la valeur 5. Vous n'avez pas à implémenter la méthode **toString** utilisée afin de produire cette sortie.

```
Polynomial p = new Polynomial();

p.add( new Term( -3, 3 ) );
p.add( new Term( 5, 3 ) );
p.add( new Term( 4, 1 ) );
p.add( new Term( 5, 0 ) );

System.out.println( "p( x ) = " + p );
System.out.println( "p( 5.0 ) = " + p.evaluate( 5.0 ) );

p( x ) = 2x^3 + 4x + 5
p( 5.0 ) = 275.0
```

Suggestion : Utilisez la méthode **Math.pow(a, b)** afin de calculer a^b.

Question 5 : Méthode récursive foldl (15 points)

Pour la classe **LinkedList** se trouvant à la page suivante, vous devez créer la méthode d'instance **foldl**. La méthode **foldl** applique l'opérateur passé en paramètre aux deux premiers éléments de la liste afin de produire un premier résultat intermédiaire, elle applique ensuite l'opérateur au résultat intermédiaire et au troisième élément afin de produire un nouveau résultat intermédiaire, et ainsi de suite. L'opérateur est toujours appliqué à un résultat intermédiaire et l'élément suivant de la liste afin de produire un nouveau résultat intermédiaire qui servira à l'étape suivante du calcul.

La plus petite liste valide contient deux éléments. Ici, l'opérateur **Plus** est appliqué à la liste contenant les entiers 1 et 2 (dans cet ordre). Le résultat final sera l'entier 3. Consultez l'appendice E afin de connaître la définition des opérateurs.

```
LinkedList xs = new LinkedList();
xs.addLast( new Integer( 1 ) );
xs.addLast( new Integer( 2 ) );
Integer result = (Integer) xs.foldl( new Plus() );
```

Dans le prochain exemple, l'opérateur **Maximum** est appliqué à une liste contenant les entiers 10, 0, 15 et 5 (dans cet ordre).

```
LinkedList xs = new LinkedList();
xs.addLast( new Integer( 10 ) );
xs.addLast( new Integer( 0 ) );
xs.addLast( new Integer( 15 ) );
xs.addLast( new Integer( 5 ) );
Integer result = (Integer) xs.foldl( new Maximum() );
```

La méthode **foldl** appliquera d'abord l'opérateur **Maximum** aux entiers 10 et 0 afin de produire le résultat intermédiaire 10. Ensuite, l'opérateur **Maximum** sera appliqué au résultat intermédiaire 10 et à l'élément suivant, l'entier 15, le résultat intermédiaire de cette étape sera 15. Finalement, l'opérateur **Maximum** est appliqué au résultat intermédiaire 15 et au prochain élément, l'entier 5, afin de produire le résultat final, 15. Le résultat de l'exécution de cette méthode est l'entier 15.

```
LinkedList xs = new LinkedList();
xs.addLast( "a" );
xs.addLast( "b" );
xs.addLast( "c" );
xs.addLast( "d" );
xs.addLast( "e" );
xs.addLast( "f" );
xs.addLast( "g" );
xs.addLast( "h" );
System.out.println( xs.foldl( new Concat() ) );
```

Finalement, l'exécution des énoncés ci-haut produira la sortie suivante : "`a,b,c,d,e,f,g,h`".

Vous devez écrire une implémentation **récursive** de la méthode d'instance **foldl** décrite ci-haut. Vous devez utiliser la technique présentée en classe où l'on utilise une méthode publique afin d'amorcer le premier appel à la méthode privée récursive.

```
public class LinkedList {
    private static class Elem {
        private Object value;
        private Elem next;
        private Elem( Object value, Elem next ) {
            this.value = value;
            this.next  = next;
        }
    }
    private Elem first;

    // Les autres méthodes de cette classe seraient ici, mais vous ne
    // devez pas les utiliser.

    public Object foldl( Operator op ) {









    } // End of foldl

    private          foldl(                                          ) {









    } // End of foldl
} // End of LinkedList
```

Question 6 : Architecture TC-1101 (10 points)

Sur cette page et celle qui suit, vous trouverez une implémentation partielle du simulateur TC-1101, `Sim.java`. En particulier, les énoncés simulant l'exécution des instructions ont été remplacés par un commentaire comme suit "`/* code simulant ... */`".

Effectuez tous les changements nécessaires afin d'ajouter l'instruction **DECR** au jeu d'instructions du TC-1101. L'instruction **DECR** décrémente la valeur de l'accumulateur de un (c'est-à-dire soustraire un de l'accumulateur) **et** met à jour les registres de statut **Z** et **N**. Vous devez aussi choisir l'**opCode** de cette instruction.

```
class Sim {

    public static final int MAX_ADDRESS = 9999; // les addresses ont 2 "octets"



    public static final int LDA = 91; // charge (load) une valeur de la mémoire dans l'accumulateur
    public static final int STA = 39; // sauve (store) le contenu de l'accumulateur en mémoire
    public static final int CLA =  8; // mise-à-zero (clear) de l'accumulateur
    public static final int INC = 10; // incrémente (ajoute 1) à l'accumulateur
    public static final int ADD = 99; // ajoute (add) à l'accumulateur
    public static final int SUB = 61; // soustrait de l'accumulateur
    public static final int JMP = 15; // branchement inconditionnel (jump, go to)
    public static final int JZ  = 17; // branchement si le bit de statut Zero est vrai
    public static final int JN  = 19; // branchement si le bit de statut Negatif est vrai
    public static final int DSP =  1; // affiche a l'écran (display)
    public static final int HLT = 64; // arrêt (halt)

    // la mémoire
    private static final int[] memory = new int[MAX_ADDRESS + 1];

    // les registres
    private static int pc;            // compteur de programme
    private static int a;             // accumulateur
    private static int opCode;        // l'opCode de l'instruction courante
    private static int opAddr;        // l'adresse de l'opérande de l'instruction courante
    private static boolean z;         // bit de statut "Zero"
    private static boolean n;         // bit de statut "Negatif"
    private static boolean h;         // bit de statut "Halt"

    private static int mar;           // registre adresse mémoire (Memory Address Register)
    private static int mdr;           // registre donnée mémoire (Memory Data Register)
    private static boolean rw;        // bit lecture/écriture (Read/Write)
                                      // lecture=true/écriture=false

    private static void accessMemory() {
        if (rw) {
            // rw=true signifie lecture, copie de la mémoire au processeur
            mdr = memory[mar];
        } else {
            // rw=false signifie écriture, copie du processeur à la mémoire
            memory[mar] = mdr;
        }
    }
```

```
    public static void run() {
        pc = 0;       // l'exécution débute toujours à l'adresse 0
        h  = false;   // remise à faux du bit de statut Halt

        while (h == false) {

            // 1. transfert de l'opCode
            mar = pc;
            pc  = pc + 1; // notez que pc est incrémenté dès maintenant
            rw  = true;
            accessMemory() ;
            opCode = mdr ;

            // 2. transfert de l'adresse de l'opérande,
            //    déterminé en fonction de l'opCode
            if ((opCode % 2) == 1) {
                mar = pc;
                pc  = pc + 1;    // incrémente la valeur du pc
                rw  = true;
                accessMemory();  // la partie haute d'opAddr
                opAddr = mdr;
                mar = pc;
                pc  = pc + 1;    // incrémente la valeur du pc
                rw  = true;
                accessMemory();             // la partie basse d'opAddr
                opAddr = 100*opAddr + mdr;  // joindre les parties haute et basse
            }

            // 3. exécution de l'instruction
            switch (opCode) {

            case LDA: { /* code simulant LDA */ break; }
            case STA: { /* code simulant STA */ break; }
            case CLA: { /* code simulant CLA */ break; }
            case INC: { /* code simulant INC */ break; }
            case ADD: { /* code simulant ADD */ break; }
            case SUB: { /* code simulant SUB */ break; }
            case JMP: { /* code simulant JMP */ break; }
            case JZ : { /* code simulant JZ  */ break; }
            case JN : { /* code simulant JN  */ break; }
            case HLT: { /* code simulant HLT */ break; }
            case DSP: { /* code simulant DSP */ break; }

            default: System.out.println("Erreur - opcode inconnu: " + opCode) ;

            }
        }
    }
}
```

Inscrivez votre réponse dans l'espace ci-bas et indiquez à l'aide de flèches ou d'étiquettes l'endroit exact où les énoncés doivent être insérés.

Question 7 : Langage d'assemblage (5 points)

Traduisez ces énoncés Java en langage d'assemblage pour le TC-1101, présenté en classe. Les mnémoniques disponibles sont : CLA, INC, HLT, LDA, STA, ADD, SUB, DSP, JMP, JZ et JN. Assurez-vous d'inclure les instructions nécessaires afin de réserver les espaces mémoire pour les variables et constantes.

```
i = 5;
while ( i > 0 ) {
  System.out.println( i );
  i = i-1;
}
```

Question 8 : Processus d'assemblage (5 points)

Traduisez le programme assembleur suivant en langage machine pour le TC-1101. Inscrivez le résultat dans le tableau ci-bas. Chaque "octet" doit être inscrit dans la case (adresse) mémoire où il serait chargé par le TC-1101. Consultez l'appendice F pour connaître les opCodes de cet assembleur.

```
       LDA X
[Loop] SUB Z
       JN  [End]
       STA X
       JMP [Loop]
[End]  HLT
X      BYTE 01
Y      BYTE 08
```

Adresse	Contenu
00 00	
00 01	
00 02	
00 03	
00 04	
00 05	
00 06	
00 07	
00 08	
00 09	
00 10	
00 11	
00 12	
00 13	
00 14	
00 15	
00 16	
00 17	
00 18	
00 19	
00 20	
00 21	
00 22	
00 23	
00 24	
00 25	
00 26	
00 27	

A Labyrinth

```
public class Labyrinth {

    // Constants
    public static final char WALL  = '-';
    public static final char IN    = '#';
    public static final char GOLD  = '*';
    public static final char EMPTY = ' ';
    public static final char LEFT  = 'L';
    public static final char RIGHT = 'R';
    public static final char UP    = 'U';
    public static final char DOWN  = 'D';
    public static final char[] MOVES = { DOWN, UP, RIGHT, LEFT };

    // Instance variables
    private int rows = 3;
    private int columns = 3;
    private int startRow = 0;
    private int startCol = 0;
    private char[][] grid = { { '#', ' ', ' ' },
                              { ' ', '-', ' ' },
                              { ' ', '*', ' ' } };

    public String solve( LinearCollection solutions ) {

        String candidate = "";
        boolean solved = false;
        solutions.add( candidate );

        while ( ! solved && ! solutions.isEmpty() ) {
            String path = (String) solutions.remove();
            for ( int m=0; m < MOVES.length && ! solved; m++ ) {
                candidate = path + MOVES[ m ];
                if ( checkPath( candidate ) ) {
                    System.out.println( candidate );
                    if ( reachesGoal( candidate ) )
                        solved = true;
                    else
                        solutions.add( candidate );
                }
            }
        }
        if ( ! solved )
            return null;
        return candidate;
    }
```

```
    private boolean checkPath( String path ) {

        boolean[][] visited = new boolean[ rows ][ columns ];

        int row = startRow;
        int col = startCol;

        visited[ row ][ col ] = true;

        boolean valid = true;
        int pos=0;

        while ( valid && pos < path.length() ) {
            char direction = path.charAt( pos++ );
            switch ( direction ) {
            case LEFT:
                col--;
                break;
            case RIGHT:
                col++;
                break;
            case UP:
                row--;
                break;
            case DOWN:
                row++;
                break;
            default:
                throw new IllegalArgumentException( "not a valid move: " + direction );
            }

            if ( row < 0 || row >= rows || col < 0 || col >= columns ) {
                valid = false;
            } else if ( visited[ row ][ col ] ) {
                valid = false;
            } else if ( isWall( row, col ) ) {
                valid = false;
            } else {
                visited[ row ][ col ] = true;
            }
        }

        return valid;
    }
}
```

```
    public boolean isGold( int x, int y ) {
        return grid[ x ][ y ] == GOLD;
    }
    public boolean isWall( int x, int y ) {
        return grid[ x ][ y ] == WALL;
    }
    private boolean reachesGoal( String path ) {

        int row = startRow;
        int col = startCol;

        for ( int pos=0; pos < path.length(); pos++ ) {
            char direction = path.charAt( pos );
            switch ( direction ) {
            case LEFT:  col--; break;
            case RIGHT: col++; break;
            case UP:    row--; break;
            case DOWN:  row++;break;
            }
        }

        return isGold( row, col );
    }
}
```

B Term

```
public class Term implements Comparable {
    // instance variables
    private int coefficient;
    private int exponent;
    public Term( int coefficient, int exponent ) {
        this.coefficient = coefficient;
        this.exponent = exponent;
    }

    public int getCoefficient() { return coefficient; }

    public int getExponent() { return exponent; }

    public void plus( Term other ) {

        if ( other == null )
            throw new IllegalArgumentException( "null" );
        if ( exponent != other.getExponent() )
            throw new IllegalArgumentException( "exponents are different" );

        coefficient + other.coefficient;
    }
    public int compareTo( Object obj ) {
        if ( ! ( obj instanceof Comparable ) )
            throw new IllegalArgumentException();
        Term other = (Term) obj;
        if ( exponent < other.exponent )
            return -1;
        else if ( exponent == other.exponent )
            return 0;
        else
            return 1;
    }
    public String toString() {
      switch ( exponent ) {
        case 0:
            return Integer.toString( coefficient );
        case 1:
            return coefficient + "x";
        default:
            return coefficient + "x^" + exponent;
        }
    }
}
```

C OrderedList

```
public interface OrderedList {

    /** Retourne la taille de la liste.
     *  @return la taille de la liste.
     */
    public abstract int size();

    /** Ajout un élément en ordre croissant selon l'ordre
     *  établi par la méthode compareTo des objets Comparable.
     *  @param l'objet à ajouter.
     */
    public abstract void add( Comparable obj );

    /** Retire l'élément se trouvant à la position spécifiée.
     *  @param la position de l'élément à retirer.
     *  Le premier élément se trouve à la position 0.
     */
    public abstract void remove( int pos );

    /** Retourne un itérateur pour cette liste.
     *  @return un itérateur pour cette liste.
     */
    public abstract Iterator iterator();
}
```

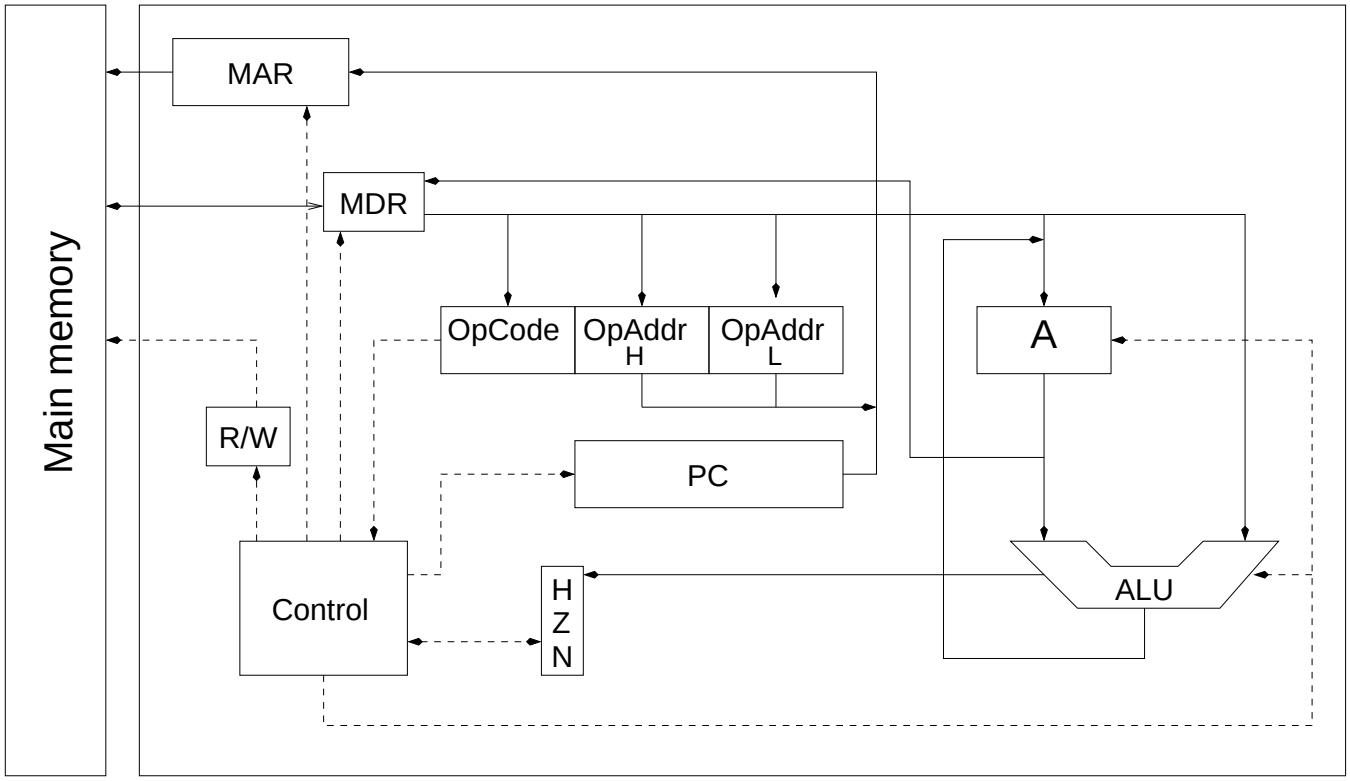
D Iterator

```
public interface Iterator {

    /** Retourne vrai si l'itération n'a plus d'éléments.
     *  @return vrai si l'itération n'a plus d'éléments et
     *  faux sinon.
     */
    public abstract boolean hasNext();

    /** Retourne le prochain élément de la liste.
     *  @return le prochain élément de la liste.
     */
    public abstract Comparable next();
}
```

E Operator

```
public interface Operator {
    public abstract Object apply( Object a, Object b );
}

public class Plus implements Operator {

    public Object apply( Object first, Object second ) {

        Integer a = (Integer) first;
        Integer b = (Integer) second;

        return new Integer( a.intValue() + b.intValue() );
    }
}

public class Maximum implements Operator {

    public Object apply( Object first, Object second ) {

        Integer a = (Integer) first;
        Integer b = (Integer) second;

        if ( a.compareTo( b ) > 0 )
            return a;
        else
            return b;
    }
}

public class Concat implements Operator {

    public Object apply( Object first, Object second ) {

        String a = (String) first;
        String b = (String) second;

        return a + "," + b;
    }
}
```

F Architecture et jeu d'instructions du TC-1101



Mnémonique	opCode	Description
LDA	**91**	charge x dans l'accumulateur
STA	**39**	sauvegarde de l'accumulateur à l'adresse x
CLA	**08**	remise-à-zéro (a=0, z=vrai, n=faux)
INC	**10**	incrémente l'accumulateur (modifie z et n)
ADD	**99**	ajoute x à l'accumulateur (modifie z et n)
ADDi	**83**	ajoute une valeur à l'accumulateur (modifie z et n)
SUB	**61**	retranche x de l'accumulateur (modifie z et n)
JMP	**15**	branchement inconditionnel vers x
JZ	**17**	branchement sur x si z==vrai
JN	**19**	branchement sur x si n==vrai
DSP	**01**	affiche la valeur se trouvant à l'adresse x
HLT	**64**	fin

où x est une adresse mémoire.

(page blanche)

Il y a 10 types de personnes, celles qui comprennent le binaire et les autres. (Anonyme)