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 to Computer Science II (CSI 1101)
Final Examination

Instructor: Marcel Turcotte

April 2005, duration: 3 hours

Identification

Student name (last, first): _____ designated seat: _____

Signature: _____ student number: _____

Instructions

1. This is a closed book examination;
2. No calculators or other aids are permitted;
3. Write comments and assumptions to get partial marks;
4. Beware, poor hand writing can affect grades;
5. Do not remove the staple holding the examination pages together;
6. Write your answers in the space provided. Use the backs of pages if necessary. You may **not** hand in additional pages;

Marking scheme

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

Question 1: Short-Answer Questions (20 marks)

A. The algorithm seen in class for evaluating postfix (RPN) expressions is using a **stack** or a **queue** as its main data structure? (2 marks)

B. Complete the table below. For each implementation technique of a linked list, write the letter **C** in the table if the number of operations performed by the method is constant, i.e. independent of the number of elements currently stored in the data structure (in this case, the method is efficient), or write **V** if the number of operations varies depending on the number of elements that are currently stored in the data structure (in this case, the method is not efficient). Consider the general cases only (not the specific cases). (4 marks)

Implementation	**addLast(Object o)**	**removeLast(Object o)**
Singly linked without a tail pointer		
Singly linked with a tail pointer		
Doubly linked without a tail pointer		
Doubly linked with a tail pointer		

C. Convert the following unsigned decimal to binary. (4 marks)

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

D. Add the following two unsigned binary numbers. (4 marks)

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

E. Correct the three (commonly occurring) mistakes in the implementation of the method **add** below. Assume that **pos** and **obj** are valid arguments, i.e. $0 \leq$ **pos** $<$ **size** and **obj** $\neq$ **null**. (6 marks)

```
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 marks)

This question is about the queue abstract data type (ADT). Its interface is defined as follows.

```
public interface Queue {
    /** Returns true if this Queue has no elements.
     *  @return true if this Queue has no elements.
     */
    public abstract boolean isEmpty();

    /** Adds an element at the rear of this Queue.
     *  @throws FullQueueException if this queue is full.
     */
    public abstract void enqueue( Object o ) throws FullQueueException;

    /** Removes and returns the front element of this Queue.
     *  @return the front element of this Queue.
     *  @throws EmptyQueueException if this queue contains no elements.
     */
    public abstract Object dequeue() throws EmptyQueueException;

    /** Removes the front element of this Queue and adds it to its rear.
     *  @throws EmptyQueueException if this queue contains no elements.
     */
    public abstract void requeue() throws EmptyQueueException;
}
```

We consider two implementations of this interface. The first implementation is **LinkedQueue**. It uses a singly linked list structure to store the elements of this queue. For efficiency reasons, the front element is the first element of the list while the rear element is the last element of the list, the instance variable **rear** designates the last (tail) element of the list.

The second implementation is **CircularQueue**. It uses a circular array to store the elements of this queue. The instance variable **front** designates the front element while the instance variable **rear** designates the rear element of this queue. The empty queue is represented by assigning the value -1 to the variables **front** and **rear**.

The partial implementation of both classes contains all the methods of the interface except the method **requeue**. For this question, you must implement the method **requeue** for both implementations.

A. In the partial implementation of the class **LinkedQueue** below, add the method **requeue** (consult the interface for its definition). Although the methods of the interface exist, you cannot use them. You cannot simply exchange the **value**s of the nodes. Your implementation must change the links of the linked list; i.e. transform the structure of the list so that the first element becomes the last one. (10 marks)

```
public class LinkedQueue implements Queue {
    private static class Elem { // nodes of the list
        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. In the partial implementation of the class **CircularQueue** below, add the method **requeue** (consult the interface for its definition). Although the methods of the interface exist, you cannot use them. Do not assume the existence of other methods, in particular, the method **nextIndex** has not been defined. (10 marks)

```
public class CircularQueue implements Queue {
    public static final int DEFAULT_CAPACITY = 100;
    private final int MAX_QUEUE_SIZE;
    private Object[] q; // stores the elements of this queue
    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();
    }
}
```

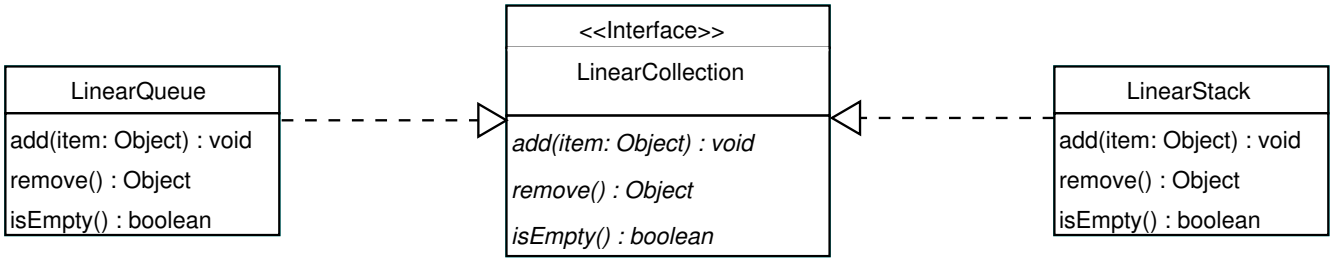
The method calls **test(new LinkedQueue())** and **test(new CircularQueue())** should both display the following output (you do not have to implement the method **toString** that was used to produce this output).

```
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 marks)

For this question, there is a simplified implementation of the class **Dungeon** from the assignment 4. In particular, there are no creatures, no armors and no weapons. Let's call this implementation **Labyrinth** to distinguish it from the one of the assignment 4.

Furthermore, as seen in class, a queue or stack can be used for saving the partial solutions in the method **solve**. Obviously, the behaviour of the algorithm changes depending on which data structure is used. For this question, you must identify the correct behaviour.



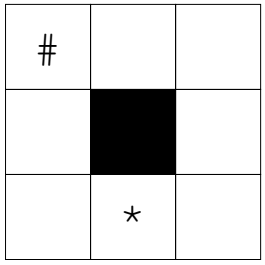
We define the interface **LinearCollection** to contain three methods: **void add(Object item)**, **Object remove()** and **boolean isEmpty()**. There are two implementations of this interface: **LinearQueue** and **LinearStack**.

LinearQueue, as all the other implementations of a queue, is such that the method **add** enqueues the **item** at the rear of the queue, the method **remove** dequeues the front element, and the method **isEmpty** returns **true** if this queue contains no elements.

LinearStack, as all the other implementations of a stack, is such that the method **add** pushes the **item** onto the top of the stack, the method **remove** pops (removes and returns) the top element, and the method **isEmpty** returns **true** if this stack contains no elements.

As can be seen on the next page, for this question, the method **solve** has been modified so that the data structure that is used for saving the partial solutions is given as an argument.

The following labyrinth has been assigned to the instance variable **grid**. The symbol `#`, in position $(0, 0)$, indicates the starting position, the symbol `*`, in position $(2, 1)$, indicates the goal (gold).



See Appendix A for the complete implementation of the class **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. Which of the following printouts corresponds to the execution of the following statements.

```
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. Which of the following printouts corresponds to the execution of the following statements.

```
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 marks)

For this question, a polynomial will be represented by an ordered list of terms. Each term consists of a coefficient and an exponent. The terms are ordered by their exponents. See Appendix B for a complete definition of the class **Term**, Appendix C for the definition of the interface **OrderedList** and Appendix D for a definition of the interface **Iterator**. Assume the existence of the class called **OrderedLinkedList** that implements the interface **OrderedList**. Its method **iterator** returns a valid **Iterator** for this list. It is implemented using the "fail-fast" technique seen in class. Write the implementation of the class **Polynomial** according to the following instructions.

A. A polynomial is an **OrderedList** of **Term**s;

B. Write the method **void add(Term t)**; adds the term **t** to this polynomial. If a term with the same exponent already exists in this polynomial then the coefficient of **t** is added to the coefficient of this term;

C. Write the method **double evaluate(double x)**; evaluates the polynomial at the given value **x**.

The statements below are creating the polynomial $2x^3+4x+5$, which is then evaluated at the value 5. You do not have to implement the method **toString**, which was used to produce the output below.

```
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
```

Hint: Use **Math.pow(a, b)** to calculate a^b.

Question 5: Recursive method foldl (15 marks)

For the class **LinkedList** on the next page, write the recursive instance method **foldl**. The method **foldl** applies the given operator to the first two elements of the list to produce a first intermediate result, it then applies the operator to the intermediate result and the third element to produce a new intermediate result, and so on. The operator is always applied to the intermediate result and the next element of the list to produce a new intermediate result for the next step in the calculation.

The smallest valid list is a list of two elements. Here the operator **Plus** is applied to a list that contains the integers 1 and 2 (in that order). The final result is the integer 3. Consult the Appendix E for the definitions of the operators.

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

For the next example, the operator **Maximum** is applied to a list that contains the integers 10, 0, 15 and 5 (in that order).

```
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() );
```

The method **foldl** will first apply the operator **Maximum** to the integers 10 and 0 to produce the intermediate result 10. Next, the operator **Maximum** is applied to the intermediate result 10 and the next element, which is the integer 15, the intermediate result for this step will be 15. Finally, the operator **Maximum** is applied to the intermediate result 15 and the next element, the integer 5, to produce the final result, 15. At the end of the execution of the method **foldl** the local variable **result** will contain the **Integer** 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() ) );
```

Finally, the above statements are producing the following output: "`a,b,c,d,e,f,g,h`".

Write a **recursive** implementation of the instance method **foldl** described above. You must use the technique presented in class where a public method initiates the first call to the recursive private method.

```
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;

    // The other methods of the class would be here but they cannot be used.

    public Object foldl( Operator op ) {









    } // End of foldl

    private          foldl(                                          ) {









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

Question 6: TC-1101 Architecture (10 marks)

On this and the next page, you will find a partial implementation of the simulator for the TC-1101 computer, `Sim.java`. In particular, the statements that simulate the execution of the instructions have been replaced by a comment of the following form "`/* code simulating ... */`".

Make all the necessary changes to the program in order to add the instruction **DECR** to the TC-1101. This instruction decrements the accumulator by one (i.e subtracts one from the accumulator) **and** updates the status registers **Z** and **N** accordingly. You also have to select an **opCode** for this instruction.

```
public class Sim {

  public static final int  MAX_ADDRESS = 9999; // addresses are 2 "bytes"



  public static final int  LDA = 91; // Load Accumulator from memory
  public static final int  STA = 39; // Store Accumulator into memory
  public static final int  CLA =  8; // Clear (set to zero)  the Accumulator
  public static final int  INC = 10; // Increment (add 1 to) the Accumulator
  public static final int  ADD = 99; // Add to Accumulator
  public static final int  SUB = 61; // Subtract from Accumulator
  public static final int  JMP = 15; // Jump ("go to")
  public static final int  JZ  = 17; // Jump if the Zero status bit is TRUE
  public static final int  JN  = 19; // Jump if the Negative status bit is TRUE
  public static final int  DSP =  1; // Display (write on the screen)
  public static final int  HLT = 64; // Halt

  private static final int[] memory = new int[MAX_ADDRESS + 1];
  private static int pc;       // program counter
  private static int a;        // accumulator
  private static int opCode;   // the opcode of the current instruction
  private static int opAddr;   // the ADDRESS of the operand of
                               // the current instruction
  private static boolean z;    // "Zero" status bit
  private static boolean n;    // "Negative" status bit
  private static boolean h;    // "Halt" status bit

  private static int mar;      // Memory Address register
  private static int mdr;      // Memory Data register
  private static boolean rw;   // Read/Write bit.  Read = True; Write = False

  private static void accessMemory() {
     if (rw) {   // rw = True means "read"
                 //    = copy a value from memory into the CPU
         mdr = memory[mar];
     } else {    // rw = False means "write"
                 //    = copy a value into memory from the CPU
         memory[mar] = mdr;
     }
  }
```

```
public static void run() {
    pc = 0;       // always start execution at location 00 00
    h  = false;   // reset the Halt status bit

    while ( h == false ) {

        // 1. FETCH OPCODE
        mar = pc;
        pc  = pc + 1;  // NOTE that pc is incremented immediately
        rw  = true;
        accessMemory();
        opCode = mdr;

        // 2. FETCH THE ADDRESS OF THE OPERAND
        if ( ( opCode % 2 ) == 1 ) {
            mar = pc;
            pc  = pc + 1;   // pc is incremented immediately
            rw  = true;
            accessMemory();
            opAddr = mdr;   // this is just the HIGH byte of the opAddr
            mar = pc;
            pc  = pc + 1;   // pc is incremented immediately
            rw  = true;
            accessMemory();            // this gets the LOW byte
            opAddr = 100*opAddr + mdr; // put the two bytes together
        }

        // 3. EXECUTE THE OPERATION

        switch (opCode) {

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

        default: System.out.println("Error - unknown opcode: " + opCode) ;

        } // End of the case statement
    } // End of fetch-execute loop
} // End of "run" method
```

Write your answer in the space below. Using arrows or labels, indicate the exact location where the statements should be added.

Question 7: TC-1101 Assembly Language (5 marks)

Convert the following Java program into the assembly language for the toy computer, TC-1101, discussed in class. The mnemonics available are: CLA, INC, HLT, LDA, STA, ADD, SUB, DSP, JMP, JZ and JN. Make sure to give the instructions to reserve the storage for the variable(s) and constant(s).

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

Question 8: Assembly process (5 marks)

Convert the following assembly program into machine code. Write down the result into the table below. Each byte must be written at the memory location (address) where it would be stored by the TC-1101 computer. Consult the Appendix F to find out the opCodes for this assembly language.

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

Address	Content
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 {

    /** Returns the size of the list.
     *  @return the size of the list.
     */
    public abstract int size();

    /** Adds an element in increasing order, as defined by
     *  the method compareTo of the Comparable object.
     *  @param obj the item to be added.
     */
    public abstract void add( Comparable obj );

    /** Removes the element at the specified position.
     *  @param the position of the element to be removed.
     *  The first element is found at position 0.
     */
    public abstract void remove( int pos );

    /** Returns an iterator onto this list.
     *  @return an iterator for this list.
     */
    public abstract Iterator iterator();
}
```

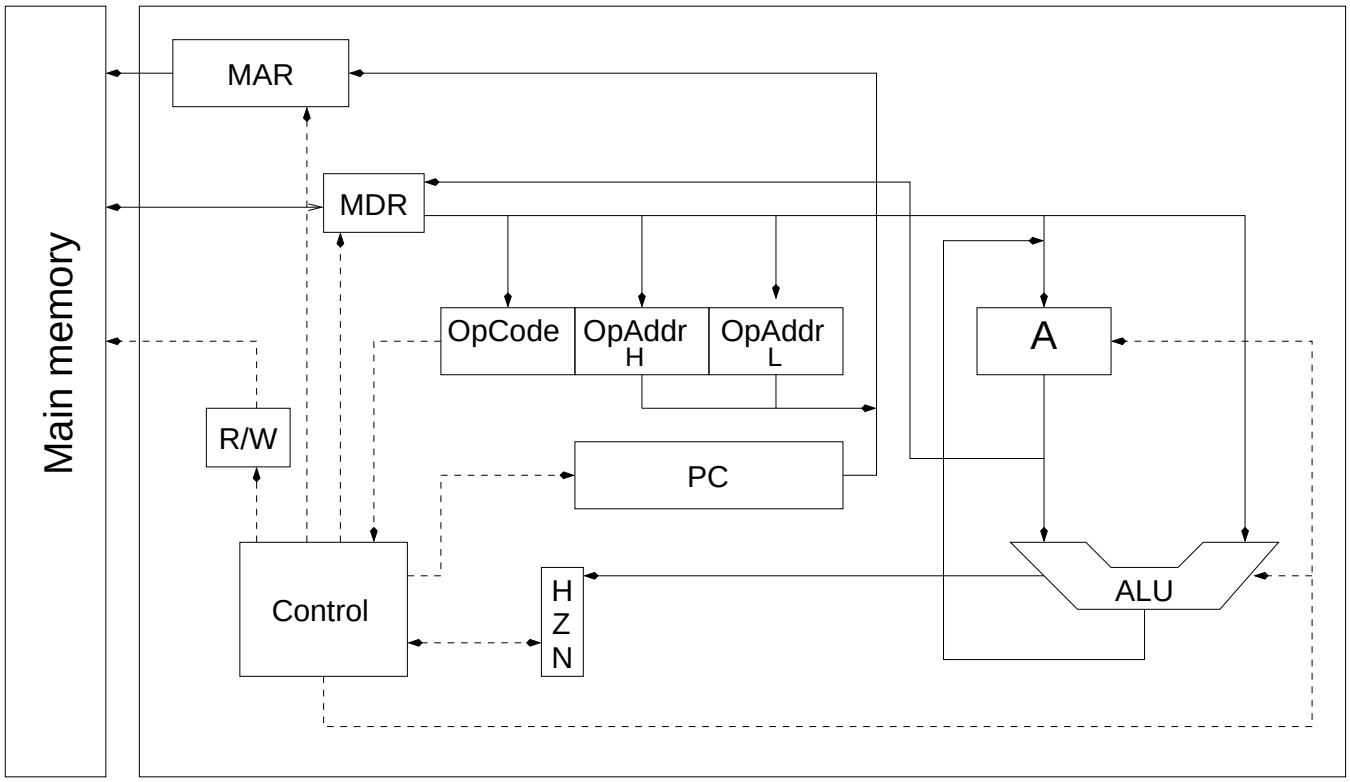
D Iterator

```
public interface Iterator {

    /** Returns true if the iteration has more elements.
     *  @return true if the iteration has more elements and
     *  false otherwise.
     */
    public abstract boolean hasNext();

    /** Returns the next element in the list.
     *  @return the next element in the list.
     */
    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 TC-1101 Architecture and Instruction Set



Mnemonic	opCode	Description
LDA	**91**	loads x
STA	**39**	stores x
CLA	**08**	clears the accumulator (a=0, z=true, n=false)
INC	**10**	increments accumulator (modifies z and n)
ADD	**99**	adds x to the accumulator (modifies z and n)
ADDi	**83**	adds a value to the accumulator (modifies z and n)
SUB	**61**	subtracts x from the accumulator (modifies z and n)
JMP	**15**	unconditional branch to x
JZ	**17**	go to x if z==true
JN	**19**	go to x if n==true
DSP	**01**	displays the content of the memory location x
HLT	**64**	halts the TC-1101

where x is a memory location.

(blank space)

There are 10 kinds of people, those who understand binary and those who don't. (Anonymous)