ÉCOLE POLYTECHNIQUE MONTRÉAL

Le génie sans frontières

Questionnaire examen final

INF2010

Sigle du cours

Identification de l'étudiant(e)		
Nom :	**Prénom :**	
Signature :	**Matricule :**	**Groupe :**

Sigle et titre du cours	*Groupe*	*Trimestre*
INF2010 - Structures de données et algorithmes	**Tous**	**20073**
Professeur	***Local***	***Téléphone***
Coordonnateur : Ettore Merlo, Chargé : Chamseddine Talhi	**M-4021**	**5758 / 5193**

Jour	*Date*	*Durée*	*Heures*
Lundi	**17 décembre 2007**	**2h30**	**9h30**

Documentation	*Calculatrice*	
☐ Toute	☒ Aucune	**Les cellulaires, agendas électroniques ou téléavertisseurs sont interdits.**
☒ Aucune	☐ Programmable	
☐ Voir directives particulières	☐ Non programmable	

Directives particulières

Écrivez votre nom et votre matricule sur les pages réponses soit les pages 1, 2, 4, 5, 6 et 7.

Bonne chance à tous!

Important

Cet examen contient **5** questions sur un total de **14** pages **(excluant cette page)**

La pondération de cet examen est de **55** **%**

Vous devez répondre sur : ☐ le questionnaire ☐ le cahier ☒ les deux

Vous devez remettre le questionnaire : ☒ oui ☐ non

Le plagiat, la participation au plagiat, la tentative de plagiat entraînent automatiquement l'attribution de la note **F** dans tous les cours suivis par l'étudiant durant le trimestre. L'École est libre d'imposer toute autre sanction jugée opportune, y compris l'exclus

Note : Remplissez les tableaux au besoin et laissez les cases non pertinentes vides.

Question 1. Tables de dispersement (6 points)

Pour chaque question, donnez le numéro qui correspond à la bonne réponse.

1.1. Le facteur de compression dans une table de dispersement est le :
(a) nombre d'éléments/la taille de l'espace libre dans la table.
(b) nombre d'éléments/taille de la table.
(c) nombre de collisions/le nombre des éléments insérés dans la table.
(d) nombre de collisions/la taille de la table.
(e) nombre de collision/la taille de l'espace libre dans la table.

Votre réponse : (b)

1.2. Dans une table de dispersement, deux enregistrements synonymes sont :
(a) deux éléments qui ont la même clé.
(b) deux éléments dont les clés sont en collision.
(c) deux éléments identiques.

Votre réponse : (b)

1.3. Dans une table de dispersement de taille N, où les collisions sont gérées en utilisant les listes chaînées, la somme des tailles de toutes les listes utilisées est :
(a) inférieure à 2 N.
(b) plus grande que N.
(c) inférieure à N.
(d) sans limites.

Votre réponse : (c)

1.4. Il est TOUJOURS possible d'insérer un élément dans une table de dispersement à débordement progressif par dispersement quadratique si :
(a) la taille de la table correspond à un nombre pair et au moins le tiers (1/3) de l'espace de la table est vide.
(b) la taille de la table correspond à un nombre impair et au moins le quart (1/4) de l'espace de la table est vide.
(c) la taille de la table correspond à un nombre premier et au moins la moitié (1/2) de l'espace de la table est vide.
(d) la taille de la table correspond à un nombre impair et au moins la moitié (1/2) de l'espace de la table est vide.
(e) la taille de la table correspond à un nombre premier et au moins le tiers (1/3) de l'espace de la table est vide.

Votre réponse : (c)

Nom : _____ Matricule : _____

Question 2. Arbres (12points)

2.1. Pour chaque question, donnez le numéro qui correspond à la bonne réponse **(5 points).**

a. Un arbre binaire complet de degré N contient :
- **(a)** au moins $2^{(N-1)}$ nœuds.
- **(b)** au moins $2^{(N+1)}$ nœuds.
- **(c)** au plus $2^{(N+1)}$ nœuds.
- **(d)** au plus $2^{(N-1)}$ nœuds.

Votre réponse : (a)

b. Dans un arbre binaire de N nœuds, il y a :
- **(a)** au moins N + 1 fils NULL.
- **(b)** exactement N + 1 fils NULL.
- **(c)** exactement N - 1 fils NULL.
- **(d)** au plus N fils NULL.

Votre réponse : (b)

c. En manipulant un arbre binaire implanté par des pointeurs, où chaque nœud pointe vers ses deux fils :
- **(a)** il est facile de parcourir l'arbre par niveau.
- **(b)** il est difficile de parcourir l'arbre en postordre.
- **(c)** il est difficile de parcourir l'arbre par niveau.

Votre réponse : (c)

d. Dans un parcours préordre d'un arbre binaire :
- **(a)** on commence par le fils de gauche, puis le fils de droite et enfin le nœud lui même.
- **(b)** on commence par le nœud puis son fils de gauche et enfin son fils de droite.
- **(c)** on commence par le fils de gauche, puis le nœud et enfin le fils de droite.

Votre réponse : (b)

e. Après une insertion dans un arbre AVL :
- **(a)** on doit faire une rotation double si l'insertion a été effectuée dans le sous-arbre de gauche du fils gauche ou dans le sous-arbre de droite du fils de droite.
- **(b)** on doit faire une rotation simple si l'insertion a été effectuée dans le sous-arbre de droite du fils gauche ou dans le sous-arbre de gauche du fils de droite.
- **(c)** on doit faire une rotation double si l'insertion a été effectuée dans le sous-arbre de droite du fils gauche ou dans le sous-arbre de gauche du fils de droite.

Votre réponse : (c)

Nom : _____ Matricule : _____

2.2. Complétez le code des fonctions du tableau 2.1 de la page 5. Les deux fonctions sont récursives. Chaque fonction reçoit comme paramètre une référence vers la racine d'un arbre binaire. La première fonction retourne le nombre de nœuds dans l'arbre binaire et la deuxième fonction retourne le nombre de feuilles dans l'arbre binaire. Quelle est la complexité en temps d'exécution de chaque fonction? Le nombre d'éléments dans l'arbre est N. **(7 points)**

Question 3. Monceaux (12 points)

Considérez le code en annexe 3.1 aux pages 8 à 11.

a) On veut ajouter une fonction **void heapsort( )** à la classe BinaryHeap. **heapsort( )** trie en ordre descendant les éléments du monceau (dans le tableau *array* du code de l'annexe 3.1) sans l'utilisation d'un tableau intermédiaire. Donnez le code de la fonction **void heapsort( ) (Répondez sur le cahier). (5 points)**

Réponse :

```
void heapsort( )
  {
    for( int i = currentSize; i > 0; i-- )
    {
       int x = array[0];
       array[0] = array[i];
       array[i] = x;
       currentSize--;
       percDown( 0);
    }
  }
```

b) On veut ajouter une fonction **int findMax( )** à la classe BinaryHeap. Donnez le code de la fonction **int findMax( )** en assurant un temps d'exécution dans le pire cas qui ne doit pas dépasser currentSize/2. **(Répondez sur le cahier) (5 points)**

Note : afin de pouvoir répondre à la question c), donnez votre algorithme même si son temps d'exécution en pire cas dépasse currentSize/2.

Réponse :

```
int findMax( )
   {
      int x = 0;
      for( int i = currentSize/2; i <= currentSize; i++ )
      {
         if (array[i] < x) x = array[i];
      }
         return x ;
   }
```

c) Quel est le temps d'exécution de votre fonction **int findMax( )** dans le pire cas ? Expliquez. **(Répondez sur le cahier) (2points)**

Réponse :

currentSize/2
Le nombre d'itérations de la boucle while ne dépasse pas currentSize/2 !

Question 4. Graphes (13 points)

a) En se basant sur les classes en annexe 4.1 aux pages 12, 13 et 14, et en admettant l'existence d'une classe **Queue** offrant les deux fonctions **enqueue** et **dequeue**, complétez le code de la méthode **unweighted()** (tableau 4.1 page 6) qui fera partie de la classe **graph**. La méthode **unweighted()** doit calculer pour chaque nœud du graphe, la longueur du plus court chemin menant de la racine du graphe (le premier nœud de nodeArr) vers ce nœud. La distance doit être sauvegardée dans la variable **dist**. En plus, cette méthode doit mettre à jour le lien **pred** de chaque nœud du graphe afin de pointer vers le nœud qui le précède dans le plus court chemin qui mène de la racine du graphe vers ce nœud. L'algorithme doit suivre la démarche améliorée, suggérée par M. A. Weiss que vous avez vue en cours et qui utilise une liste de travail (Worklist). **(7 points)**
Quelle est la complexité de cette méthode ? Expliquez! Considérez N et M comme nombre de nœuds et nombre d'arêtes du graphe respectivement. **(3 points)**

La complexité de cette fonction est O(N + M)

Explication :

Le nombre maximal de fils visités est égale au nombre d'arêtes.
Le nombre maximal de nœuds insérés puis retirés de la file est égale au nombre de nœuds dans le graphe

b) En se basant sur les classes en annexe 4.1 aux pages 12, 13 et 14, écrivez la méthode **PrintPath(graphNode node)** qui imprime pour un nœud du graphe, le plus court chemin menant de la racine du graphe vers ce nœud. On suppose que la fonction **unweighted()** a déjà été exécutée et que la variable **pred** de chaque nœud pointe vers le nœud qui le précède dans le plus court chemin qui mène de la racine du graphe vers ce nœud. Respectez l'ordre des nœuds dans le chemin dans l'impression du chemin. (**Répondez sur le cahier) (3 points)**

Réponse :

```
PrintPath(graphNode node) {
        if (node !=null){
        PrintPath(node.pred);
        System.out.println("to");
        }
node.print();
}
```

Question 5. Chaînes (12 points)

Considérez la chaîne c = "bababb" et le texte t = "abababbababb".

a) Construisez l'automate à états finis qui effectue la recherche de la chaîne c dans un texte arbitraire. **(10 points)**

REMPLISSEZ le tableau 5.1 de la page 7.

b) Identifiez le(s) décalage(s) (*shift*(s)) de concordance du texte t avec la chaîne c. **(2 points)**

REMPLISSEZ le tableau 5.2 de la page 7 avec le(s) décalage(s) identifié(s).

Nom : _____ Matricule : _____

Tableau 2.1

a)

```
static int countNodes(BinaryNode t ){                                   //   (2points)
        if (t == null) return  0                ;

        retturn  countNodes(t.left) +  countNodes(t.right) + 1          ;

        }
```

La complexité de cette fonction en temps d'exécution est : O(N) **(1 point)**

b)

```
static int countLeaves(BinaryNode t ){                                  //   (3points)
        if (t == null) { return  0  }

        else if (  t.left == null && t.right ==null                     )

                {return 1}

        else retturn countLeaves(t.left) +  countLeaves(t.right)        ;
```

La complexité de cette fonction en temps d'exécution est : O(N) **(1point)**

- Le code de la classe BinaryNode est le suivant :

```
private static class BinaryNode<AnyType>
   {
       BinaryNode( AnyType theElement )      // Constructors
        { this( theElement, null, null ); }

      BinaryNode( AnyType theElement, BinaryNode<AnyType> lt, BinaryNode<AnyType> rt )
       { element  = theElement; left = lt; right = rt; }

      AnyType element;           // The data in the node
      BinaryNode<AnyType> left;   // Left child
      BinaryNode<AnyType> right;  // Right child
   }
```

Tableau 4.1 **(7 points)**

```
void unweighted( )  // Le code demandé est une instruction ou
                    //  une partie d'une instruction
  {
    Queue< graphNode > q = new Queue<graphNode >( );
   for( int i = 0; i < size(); i++ )
    {  //Initialisation de dist et pred.
       getNode(i).init();
    }
    // attribuer la valeur 0 à dist de la racine
    getNode(0). UpdateDist(0);      /* Complétez */
    //insérer le noeud racine dans la file
    q.enqueue(  getNode(0)  ) ; /* Complétez  */

    while( !q.isEmpty( ) )
    {
       //retirer un nœud de la file
       graphNode v =  q.dequeue( )  ; /* Complétez  */

      //récupérer les fils (adjacents) du noeud v
       adj kids =  v.getKids()  ; /* Complétez */

       // lire le premier noeud adjacent
       kids.first();
       graphNode w = kids.getCurrent();
       // vérifier tous les fils
       while (w !=null)
         {
          if( w.getDist() == UNDEF_VAL )
           {
             // mettre à jour la distance de ce fils
            w. UpdateDist(v.getDist + 1);  ; /* Complétez */
              // mettre à jour le prédécesseur  de ce fils
            w. updatePred (v);  ; /* Complétez */
             q.enqueue( w );
           }
         // lire le prochain fils
         kids.next();
          w = kids.getCurrent();  ; /* Complétez */
         }
    }
  }
```

Nom : _____ Matricule : _____

Tableau 5.1 **(3 points)**

Q	0,1,2,3,4,5,6
Q_o	0
F	6

Fonction de transition δ **(7 points)**

Symbole de l'alphabet

État

	a	**b**		
0	0	1		
1	2	1		
2	0	3		
3	4	1		
4	0	5		
5	4	6		
6	2	1		
7				
8				

Tableau 4.2 **(2 points)**

0	1	2	3	4	5	6	7	8
	X					X		

Décalages identifiées

Annexe 3.1

```
import java.io.*;
import java.util.*;
import java.lang.*;

// BinaryHeap class
//
// CONSTRUCTION: with optional capacity (that defaults to 100)
//               or an array containing initial items
//
// ******************PUBLIC OPERATIONS*********************
// void insert( x )       --> Insert x
// Comparable deleteMin( )--> Return and remove smallest item
// Comparable findMin( )  --> Return smallest item
// boolean isEmpty( )     --> Return true if empty; else false
// void makeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Throws UnderflowException as appropriate

/**
 * Implements a binary heap.
 * Note that all "matching" is based on the compareTo method.
 * @author Mark Allen Weiss
 */
public class BinaryHeap<AnyType extends Comparable<? super AnyType>>
{
    /**
     * Construct the binary heap.
     */
    public BinaryHeap( )
    {
        this( DEFAULT_CAPACITY );
    }

    /**
     * Construct the binary heap.
     * @param capacity the capacity of the binary heap.
     */
    public BinaryHeap( int capacity )
    {
        currentSize = 0;
        array = (AnyType[]) new Comparable[ capacity + 1 ];
    }




/**
     * Construct the binary heap given an array of items.
     */
    public BinaryHeap( AnyType [ ] items )
```

```
{
    currentSize = items.length;
    array = (AnyType[]) new Comparable[ ( currentSize + 2 ) * 11 / 10 ];

    int i = 1;
    for( AnyType item : items )
        array[ i++ ] = item;
    buildHeap( );
}

/**
 * Insert into the priority queue, maintaining heap order.
 * Duplicates are allowed.
 * @param x the item to insert.
 */
public void insert( AnyType x )
{
    if( currentSize == array.length - 1 )
        enlargeArray( array.length * 2 + 1 );

        // Percolate up
    int hole = ++currentSize;
    for( ; hole > 1 && x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
        array[ hole ] = array[ hole / 2 ];
    array[ hole ] = x;
}

private void enlargeArray( int newSize )
{
    AnyType [] old = array;
    array = (AnyType []) new Comparable[ newSize ];
    for( int i = 0; i < old.length; i++ )
        array[ i ] = old[ i ];
}

/**
 * Find the smallest item in the priority queue.
 * @return the smallest item, or throw an UnderflowException if empty.
 */
public AnyType findMin( )
{
    if( isEmpty( ) )
        //throw new UnderflowException( );
        System.exit(1);
    return array[ 1 ];
}

/**
 * Remove the smallest item from the priority queue.
 * @return the smallest item, or throw an UnderflowException if empty.
 */
public AnyType deleteMin( )
```

```
{
    if( isEmpty( ) )
        //throw new UnderflowException( );
        System.exit(1);

    AnyType minItem = findMin( );
    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );

    return minItem;
}

/**
 * Establish heap order property from an arbitrary
 * arrangement of items. Runs in linear time.
 */
private void buildHeap( )
{
    for( int i = currentSize / 2; i > 0; i-- )
        percolateDown( i );
}

/**
 * Test if the priority queue is logically empty.
 * @return true if empty, false otherwise.
 */
public boolean isEmpty( )
{
    return currentSize == 0;
}

/**
 * Make the priority queue logically empty.
 */
public void makeEmpty( )
{
    currentSize = 0;
}

private static final int DEFAULT_CAPACITY = 10;

private int currentSize;      // Number of elements in heap
private AnyType [ ] array; // The heap array

/**
 * Internal method to percolate down in the heap.
 * @param hole the index at which the percolate begins.
 */
private void percolateDown( int hole )
```

```
    {
        int child;
        AnyType tmp = array[ hole ];

        for( ; hole * 2 <= currentSize; hole = child )
        {
            child = hole * 2;
            if( child != currentSize &&
                    array[ child + 1 ].compareTo( array[ child ] ) < 0 )
                child++;
            if( array[ child ].compareTo( tmp ) < 0 )
                array[ hole ] = array[ child ];
            else
                break;
        }
        array[ hole ] = tmp;
    }

    public void print() {

        int i = 0;

        for (i=1; i <= currentSize; i++) {
            if (array[i] != null)
               System.out.println("POS: " + i + " VAL: " + array[i]);
        }
        System.out.println();
    }

}
```

Annexe 4.1

```
import java.io.*;
import java.util.*;

class adj {
    static final int UNDEF_VAL = -9999;

    ArrayList<graphNode> adjList = new ArrayList<graphNode>();
    //ArrayIterator<graphNode> listIt = null;
    //graphNode curItem = null;
    int curIndex = UNDEF_VAL;

    void first() {
        curIndex = 0;
    };

    boolean currentIsValid() {
        return((curIndex >= 0) && (curIndex < adjList.size()));
    };

    graphNode getCurrent() {
        if ((curIndex >= 0) && (curIndex < adjList.size()))
            return(adjList.get(curIndex));
        else
            return(null);
    };

    void next() {
        if ((curIndex >= 0) && (curIndex < (adjList.size() - 1)))
            curIndex++;
        else
            curIndex = UNDEF_VAL;
    };

    void add(graphNode node) {
        adjList.add(node);
    }
};
```

```
import java.io.*;
import java.util.*;

class graph {

    static final int UNDEF_VAL = -9999;

    ArrayList<graphNode> nodeArr = new ArrayList<graphNode>();
    int curIndex = UNDEF_VAL;

    void initNodes() {
        int i = UNDEF_VAL;

        for (i = 0; i < nodeArr.size(); i++) {
            nodeArr.get(i).init();
        }
    }

    void first() {
        curIndex = 0;
    };

    boolean currentIsValid() {
        return((curIndex >= 0) && (curIndex < nodeArr.size()));
    };

    graphNode getCurrent() {
        if ((curIndex >= 0) && (curIndex < nodeArr.size()))
            return(nodeArr.get(curIndex));
        else
            return(null);
    };

    graphNode getNode(int pos) { // get(0) retourne la racine du graphe
        if ((pos >= 0) && (pos < nodeArr.size()))
            return(nodeArr.get(pos));
        else
            return(null);
    };

    void next() {
        if (curIndex < (nodeArr.size() - 1))
            curIndex++;
        else
            curIndex = UNDEF_VAL;
    };


    int size() {
        return(nodeArr.size());
    }
}
```

```
class graphNode {

    static final int UNDEF_VAL = -9999;

    int nodeId;
    int dist; //distance du plus court chemin de la racine vers ce noeud
    graphNode pred;    // le prédécesseur dans le plus cours chemin
    adj kids = new adj();      //Les noeuds reliés avec des arcs sortants
    adj parents = new adj();   //Les noeuds reliés avec des arcs entrants

    graphNode() {
        nodeId = UNDEF_VAL;
        dist = UNDEF_VAL;
        pred = null;
    }

    void init() {
        dist = UNDEF_VAL;
        pred = null;
    }

    int getDist(){
      return dist;
     }


    void updateDist(int newDist){
    dist = newDist;
    }

    void updatePred(graphNode newPred){
    pred = newPred;
    }


     adj getKids(){
      return kids;
     }

     adj getParents(){
      return parents;
     }

    void addKid (graphNode node) {
        kids.add(node);
    }

    void addParent (graphNode node) {
        parents.add(node);
    }
void print() {
        System.out.println("NODE: " + nodeId);
        }
};
```