ÉCOLE POLYTECHNIQUE MONTRÉAL

Le génie sans frontières

Questionnaire examen intra

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	20063
Professeur	***Local***	***Téléphone***
Ettore Merlo	M-4105	5758

Jour	*Date*	*Durée*	*Heures*
Vendredi	20 octobre 2006	2h00	12h45

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

Directives particulières

Bonne chance à tous!

Important

Cet examen contient 5 questions sur un total de 7 pages (excluant cette page)

La pondération de cet examen est de 30 %

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

Vous devez remettre le questionnaire : ☒ oui ☐ non

Question 1 (20 points)

Écrivez une procédure "*intersect*" avec complexité linéaire (T(N) = O(N)) et avec la signature suivante :

LinkedList intersect(LinkedList L1, LinkedList L2)

qui calcule L3 = L1 ∩ L2 (où L1 et L2 sont des listes triées) en utilisant les opérations de base des listes.

Note : Si au moins une des listes L1 ou L2 n'est pas triée, intersect retourne la liste vide.

Question 2 (20 points)

Considérez la classe "*Stack*" suivante qui implante une pile en Java :

public class Stack <T>
extends java.lang.Object

Stack() constructor

a) Écrivez la routine "*eval*" suivante :

int eval(char[] expr)

pour évaluer les expressions post-fixes en utilisant une pile de type "*Stack*".

Note : "*Stack*" possède les méthodes de base suivantes :

boolean empty()
T void pop()
T void push(T x)
T peek()

Note : Limitez les expressions à l'opérateur binaire '+' et aux opérandes constitués par les chiffres de '0' à '9'.

Note : Considérez la méthode :

static int digit(char ch, int radix)

dans la classe "*Character*" en Java qui retourne la valeur entière d'un caractère en base "*radix*".

Exemple :

```
char c = '4';
i = Character.digit(c, 10);
```

b) Donnez le résultat de l'évaluation de l'expression suivante :

```
char[] expr = {'3', '1', '4', '+', '+', '0', '5', '+', '+'};
```

Question 3 (30 points)

Considérez :

- la fonction de dispersement h(x) = x % 7
- différentes tables de dispersement de dimension 7
- les clefs : 15, 29, 16, 22.

a) Considérez une table de dispersement avec listes chaînées pour la résolution des collisions.

a.1) Quelle-est la valeur h(x) utilisée pour l'insertion de chaque clef indiquée ?

a.2) Dessinez l'état de la table après l'insertion de toutes les clefs indiquées.

b) Considérez une table de dispersement par débordement progressif avec "*sondage*" linéaire (f(i) = i)

b.1) Quelles sont les valeurs i et $h_i(x)$ utilisées pour l'insertion de chaque clef indiquée ?

b.2) Dessinez l'état de la table après l'insertion de toutes les clefs indiquées.

c) Considérez une table de dispersement par débordement progressif avec "*sondage*" à double dispersement ($f(i) = i * h_2(x)$), avec fonction de dispersement primaire $h_1(x) = h(x)$ et avec fonctions de dispersement secondaire ($h_2(x) = 5 - (x \% 5)$).

c.1) Quelles sont les valeurs i et $h_i(x)$ utilisées pour l'insertion de chaque clef indiquée ?

c.2) Dessinez l'état de la table après l'insertion de toutes les clefs indiquées.

Question 4 (20 points)

Considérez le code en annexe 1 à la page 4.

Considérez le vecteur "a" qui suit :

```
int[] a = {5, 6, 0, 4, 7, 8, 0, 5, 1, 7, 9, 3};
```

Remplissez l'annexe 2 à la page 7 avec les résultats des impressions exécutées par le programme en annexe 1.

Notez : Si une sortie n'est pas pertinente, laissez la case correspondante blanche.

Question 5 (10 points)

a) Quelle est la formule récursive du temps d'exécution de "*quicksort*" ?

T(N) = ...

b) Quelle est l'expression de la complexité de "*quicksort*" dans le meilleur cas ?

T(N) = O(...)

c) Dérivez mathématiquement la réponse 3b) de la réponse 3a) et JUSTIFIEZ les passages mathématiques.

Annexe 1

```
/**
 * Quicksort algorithm.
 * @param a an array of Comparable items.
 */
public static void quicksort( Comparable [ ] a )
{
    quicksort( a, 0, a.length - 1 );
}

private static final int CUTOFF = 10;

/**
 * Method to swap to elements in an array.
 * @param a an array of objects.
 * @param index1 the index of the first object.
 * @param index2 the index of the second object.
 */
public static final void swapReferences( Object [ ] a, int index1, int index2
)
{
    Object tmp = a[ index1 ];
    a[ index1 ] = a[ index2 ];
    a[ index2 ] = tmp;
}

/**
 * Return median of left, center, and right.
 * Order these and hide the pivot.
 */
private static Comparable median3( Comparable [ ] a, int left, int right )
{
    int center = ( left + right ) / 2;
    if( a[ center ].compareTo( a[ left ] ) < 0 )
        swapReferences( a, left, center );
    if( a[ right ].compareTo( a[ left ] ) < 0 )
        swapReferences( a, left, right );
    if( a[ right ].compareTo( a[ center ] ) < 0 )
        swapReferences( a, center, right );

        // Place pivot at position right - 1
    swapReferences( a, center, right - 1 );
    return a[ right - 1 ];
}

/**
 * Internal quicksort method that makes recursive calls.
 * Uses median-of-three partitioning and a cutoff of 10.
 * @param a an array of Comparable items.
 * @param left the left-most index of the subarray.
 * @param right the right-most index of the subarray.
 */
private static void quicksort( Comparable [ ] a, int left, int right )
{
```

```
    //
    // indexe d'impression
    //

    int index = 0;

    if( left + CUTOFF <= right )
    {
        Comparable pivot = median3( a, left, right );

                    // Begin partitioning
        int i = left, j = right - 1;
        for( ; ; )
        {
            while( a[ ++i ].compareTo( pivot ) < 0 ) { }
            while( a[ --j ].compareTo( pivot ) > 0 ) { }
            if( i < j )
                swapReferences( a, i, j );
            else
                break;
        }

        swapReferences( a, i, right - 1 );   // Restore pivot

        //
        // impression
        //

      System.out.println("LEFT: " + a[left]);
      System.out.println("PIVOT: " + pivot);
      System.out.println("RIGHT: " + a[right]);

      for (index = left; index <= i - 1; index++) {
          System.out.println("INDEX: " +
                         index +
                         " ELEMENT: " +
                         a[index]);
      }

      for (index = i + 1; index <= right; index++) {
          System.out.println("INDEX: " +
                         index +
                         " ELEMENT: " +
                         a[index]);
      }

        quicksort( a, left, i - 1 );    // Sort small elements
        quicksort( a, i + 1, right );   // Sort large elements
    }
    else  // Do an insertion sort on the subarray
        insertionSort( a, left, right );
}
```

```
/**
 * Internal insertion sort routine for subarrays
 * that is used by quicksort.
 * @param a an array of Comparable items.
 * @param left the left-most index of the subarray.
 * @param right the right-most index of the subarray.
 */
private static void insertionSort( Comparable [ ] a, int left, int right )
{
    for( int p = left + 1; p <= right; p++ )
    {
        Comparable tmp = a[ p ];
        int j;

        for( j = p; j > left && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

Annexe 2

Impression 1 :

Left	Pivot	Right	a											
			0	1	2	3	4	5	6	7	8	9	10	11

Impression 2 :

Left	Pivot	Right	a											
			0	1	2	3	4	5	6	7	8	9	10	11

Impression 3 :

Left	Pivot	Right	a											
			0	1	2	3	4	5	6	7	8	9	10	11

Impression 4 :

Left	Pivot	Right	a											
			0	1	2	3	4	5	6	7	8	9	10	11

Impression 5 :

Left	Pivot	Right	a											
			0	1	2	3	4	5	6	7	8	9	10	11

Nom de l'étudiant : _____ Matricule : _____