

# Questionnaire examen intra

**INF2010**

Sigle du cours

<i>Identification de l'étudiant(e)</i>		
<b>Nom :</b>	<b>Prénom :</b>	
<b>Signature :</b>	<b>Matricule :</b>	<b>Groupe :</b>

<i>Sigle et titre du cours</i>		<i>Groupe</i>	<i>Trimestre</i>
<b>INF2010 – Struct. de donn. et algorith.</b>		<b>Tous</b>	<b>20073</b>
<i>Professeur</i>		<i>Local</i>	<i>Téléphone</i>
<b>Ettore Merlo, coordonnateur – Chamseddine Talhi, chargé</b>		<b>M-4105</b>	<b>5758</b>
<i>Jour</i>	<i>Date</i>	<i>Durée</i>	<i>Heures</i>
<b>Lundi</b>	<b>22 octobre 2007</b>	<b>2h00</b>	<b>18h00</b>
<i>Documentation</i>		<i>Calculatrice</i>	
<input type="checkbox"/> Toute <input type="checkbox"/> Aucune <input type="checkbox"/> Voir directives particulières		<input type="checkbox"/> Aucune <input type="checkbox"/> Programmable <input type="checkbox"/> Non programmable <div style="text-align: center; padding-top: 10px;"><b>Les cellulaires, agendas électroniques ou téléavertisseurs sont interdits.</b></div>	
<i>Directives particulières</i>			
<i>Bonne chance à tous!</i>			

<b>Important</b>	<p>Cet examen contient <b>4</b> questions et <b>3</b> annexes sur un total de <b>13</b> pages (excluant cette page)</p> <p>La pondération de cet examen est de <b>30 %</b></p> <p>Vous devez répondre sur : <input type="checkbox"/> le questionnaire <input type="checkbox"/> le cahier <input type="checkbox"/> les deux</p> <p>Vous devez remettre le questionnaire : <input type="checkbox"/> oui <input type="checkbox"/> non</p>
------------------	--

**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'exclusion.



**Question 1 - Listes et tables de dispersement Java****(8 points)**

Considérez le code en Annexe I qui implante une liste doublement chaînées.

On veut ajouter un index (table de dispersement) à la liste afin d'améliorer les performances des opérations

```
public AnyType get( int idx )  
public AnyType set( int idx, AnyType newVal )
```

à l'intérieur de la liste chaînée pour les rendre à complexité  $O(1)$  lorsque la liste est utilisée à partir de l'état courant, sans insertions ni modifications par la suite.

- 1.1 Écrivez le code Java pour ajouter une table de dispersement de type `HashMap<K, V>` (Annexe II) à `MyLinkedList` afin d'indexer les éléments de la liste chaînée selon leur position. Chaque élément de la table contient une position et la référence à l'objet de la liste qui correspond à la position. **(1 point)**

- 1.2 Écrivez le code de la routine

```
public void indexer()
```

qui construit et remplit un index (table de dispersement) approprié à partir de l'état courant de la liste. **(5 points)**

- 1.3 Écrivez le code des nouvelles opérations

```
public AnyType getIndexed( int idx )  
public AnyType setIndexed( int idx, AnyType newVal )
```

qui correspondent aux opérations "get" et "set" précédentes, mais qui doivent montrer une complexité  $O(1)$ . **(2 points)**

**Question 2 - Tables de dispersement****(8 points)**

Considérez :

- la fonction de dispersement  $h(x) = x \% 11$
- les différentes tables de dispersement de dimension 11
- les clefs : 92, 50, 17, 24, 39, 42

- 2.1 Considérez une table de dispersement avec listes chaînées pour la résolution des collisions.

2.1.1 Quelle est la valeur  $h(x)$  utilisée pour l'insertion de chaque clef indiquée ? **(1.5 point)**

2.1.2 Dessinez l'état de la table après l'insertion de toutes les clefs indiquées. **(1 point)**

- 2.2 Considérez une table de dispersement par débordement progressif avec "sondage" linéaire ( $f(i) = i$ ).

2.2.1 Quelles sont les valeurs  $i$  et  $h_i(x)$  utilisées pour l'insertion de chaque clef indiquée ? **(1.5 point)**

2.2.2 Dessinez l'état de la table après l'insertion de toutes les clefs indiquées. **(1 point)**

2.3 Considérez une table de dispersement par débordement progressif avec "sondage" à double dispersement ( $f(i) = i * h_2(x)$ ), avec une fonction de dispersement secondaire ( $h_2(x) = 7 - (x \% 7)$ ).

2.3.1 Quelles sont les valeurs  $i$  et  $h_i(x)$  utilisées pour l'insertion de chaque clef indiquée ? **(2 points)**

2.3.2 Dessinez l'état de la table après l'insertion de toutes les clefs indiquées. **(1 point)**

### Question 3 - Algorithmes de tri

**(7 points)**

3.1 Soit la variante suivante de l'algorithme de tri par insertion :

```
public static <AnyType extends Comparable<? super AnyType>> void insertionSort( AnyType [ ] a )
{
    int j;
    for( int p = 1; p < a.length; p++ )
    {
        AnyType tmp = a[ p ];
        for( j = p; j > 0 && tmp.compareTo( a[ j-1 ] ) <= 0; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

3.1.1 Quel est le temps d'exécution de cet algorithme lorsque tous les éléments du tableau sont égaux ? Argumentez votre réponse. **(2 points)**

3.1.2 Est-ce que le temps d'exécution que vous avez obtenu correspond à celui attendu du tri par insertion pour ce cas particulier de tableau (tous les éléments sont égaux) ? **(1.5 points)**

3.1.3 Si vous avez répondu par « Non » à la question précédente, montrez la partie du code qui a fait la différence par rapport à la version originale de l'algorithme. Argumentez votre réponse. **(1.5 points)**

3.2 Le tri par insertion assure une meilleure performance dans le meilleur cas (tableau déjà trié). Plus précisément, son temps d'exécution est de  $O(N)$  alors que le temps d'exécution des deux autres algorithmes (sélection et en bulle) est de  $O(N^2)$ .

Expliquez pourquoi le tri par insertion est plus rapide lorsque le tableau est déjà trié en le comparant avec le tri par sélection **(2 points)**. Les deux algorithmes sont les suivants :

### **Tri par sélection**

```
public static <AnyType extends Comparable<? super AnyType>> void SelectionSort( AnyType [ ] a)
{
    int i, j, min;
    AnyType tmp;
    for ( i=0 ; i<a.length-1 ; i++ ) {
        //Identification de l'index du plus petit élément
        min = i;
        for ( j=i+1 ; j< a.length ; j++ ) {
            if ( a[j]<a[min] )
                min = j;
        }
        //Permutation des elements
        tmp = a[i];
        a[i] = a[min];
        a[min] = tmp;
    }
}
```

### **Tri par insertion**

```
public static <AnyType extends Comparable<? super AnyType>> void insertionSort( AnyType [ ] a )
{
    int j;
    for( int p = 1; p < a.length; p++ )
    {
        AnyType tmp = a[ p ];
        for( j = p; j > 0 && tmp.compareTo( a[ j - 1 ] ) < 0; j-- )
            a[ j ] = a[ j - 1 ];
        a[ j ] = tmp;
    }
}
```

**Question 4 – Arbres****(7 points)**

- 4.1 Soit un arbre binaire de recherche vide T utilisé pour la sauvegarde d'entiers positifs. Dessinez l'arbre T après chacune des opérations suivantes : **(2 points)**

Insertion de l'élément 10  
Insertion de l'élément 14  
Insertion de l'élément 7  
Insertion de l'élément 8  
Insertion de l'élément 5  
Insertion de l'élément 17  
Insertion de l'élément 12  
Insertion de l'élément 15  
Suppression de l'élément 14  
Insertion de l'élément 13  
Suppression de l'élément 10

- 4.2 La classe `BinaryNode<AnyType>` et les principales fonctions utilisées pour la manipulation d'un arbre binaire de recherche sont données dans l'Annexe III.

En supposant que l'arbre T, vide au départ, a été mis à jour en utilisant les fonctions d'insertion et de suppression présentées dans l'Annexe III :

- 4.2.1 Dessinez l'état de l'arbre avant et après chaque opération de suppression présentée dans la question précédente. **(2 points)**
- 4.2.2 Comparez les états de l'arbre T obtenus dans 4.2.1 avec ceux obtenus dans 4.1. **(1 point)**
- 4.2.3 Expliquez pourquoi les résultats obtenus dans 4.2.1 sont différents de ceux obtenus dans 4.1. **(1 point)**
- 4.2.4 Quels sont les changements appropriés à apporter au code de l'Annexe III afin d'obtenir les bons résultats ? **(1 point)**

## Annexe I

```
1
2  /**
3   * LinkedList class implements a doubly-linked list.
4   */
5  public class MyLinkedList<AnyType> implements Iterable<AnyType>
6  {
7      /**
8       * Construct an empty LinkedList.
9       */
10     public MyLinkedList()
11     {
12         clear();
13     }
14
15     /**
16      * Change the size of this collection to zero.
17      */
18     public void clear()
19     {
20         beginMarker = new Node<AnyType>( null, null, null );
21         endMarker = new Node<AnyType>( null, beginMarker, null );
22         beginMarker.next = endMarker;
23
24         theSize = 0;
25     }
26
27     /**
28      * Returns the number of items in this collection.
29      * @return the number of items in this collection.
30      */
31     public int size()
32     {
33         return theSize;
34     }
35
36     public boolean isEmpty()
37     {
38         return size() == 0;
39     }
40
```

```
41  /**
42   * Adds an item to this collection, at the end.
43   * @param x any object.
44   * @return true.
45   */
46  public boolean add( AnyType x )
47  {
48      add( size( ), x );
49      return true;
50  }
51
52  /**
53   * Adds an item to this collection, at specified position.
54   * Items at or after that position are slid one position higher.
55   * @param x any object.
56   * @param idx position to add at.
57   * @throws IndexOutOfBoundsException if idx is not between 0 and size(), inclusive.
58   */
59  public void add( int idx, AnyType x )
60  {
61      addBefore( getNode( idx, 0, size( ) ), x );
62  }
63
64  /**
65   * Adds an item to this collection, at specified position p.
66   * Items at or after that position are slid one position higher.
67   * @param p Node to add before.
68   * @param x any object.
69   * @throws IndexOutOfBoundsException if idx is not between 0 and size(), inclusive.
70   */
71  private void addBefore( Node<AnyType> p, AnyType x )
72  {
73      Node<AnyType> newNode = new Node<AnyType>( x, p.prev, p );
74      newNode.prev.next = newNode;
75      p.prev = newNode;
76      theSize++;
77  }
78
79
80  /**
81   * Returns the item at position idx.
82   * @param idx the index to search in.
83   * @throws IndexOutOfBoundsException if index is out of range.
84   */
85  public AnyType get( int idx )
86  {
87      return getNode( idx ).data;
88  }
89
90  /**
91   * Changes the item at position idx.
```



```
92     * @param idx the index to change.
93     * @param newVal the new value.
94     * @return the old value.
95     * @throws IndexOutOfBoundsException if index is out of range.
96     */
97     public AnyType set( int idx, AnyType newVal )
98     {
99         Node<AnyType> p = getNode( idx );
100        AnyType oldVal = p.data;
101
102        p.data = newVal;
103        return oldVal;
104    }
105
106    /**
107     * Gets the Node at position idx, which must range from 0 to size( ) - 1.
108     * @param idx index to search at.
109     * @return internal node corresponding to idx.
110     * @throws IndexOutOfBoundsException if idx is not between 0 and size( ) - 1, inclusive.
111     */
112    private Node<AnyType> getNode( int idx )
113    {
114        return getNode( idx, 0, size( ) - 1 );
115    }
116
117    /**
118     * Gets the Node at position idx, which must range from lower to upper.
119     * @param idx index to search at.
120     * @param lower lowest valid index.
121     * @param upper highest valid index.
122     * @return internal node corresponding to idx.
123     * @throws IndexOutOfBoundsException if idx is not between lower and upper, inclusive.
124     */
125    private Node<AnyType> getNode( int idx, int lower, int upper )
126    {
127        Node<AnyType> p;
128
129        if( idx < lower || idx > upper )
130            throw new IndexOutOfBoundsException( "getNode index: " + idx + "; size: " + size( ) );
131
132        if( idx < size( ) / 2 )
133        {
134            p = beginMarker.next;
135            for( int i = 0; i < idx; i++ )
136                p = p.next;
137        }
```

```
138     else
139     {
140         p = endMarker;
141         for( int i = size( ); i > idx; i-- )
142             p = p.prev;
143     }
144
145     return p;
146 }
147
148 /**
149  * Removes an item from this collection.
150  * @param idx the index of the object.
151  * @return the item was removed from the collection.
152  */
153 public AnyType remove( int idx )
154 {
155     return remove( getNode( idx ) );
156 }
157
158 /**
159  * Removes the object contained in Node p.
160  * @param p the Node containing the object.
161  * @return the item was removed from the collection.
162  */
163 private AnyType remove( Node<AnyType> p )
164 {
165     p.next.prev = p.prev;
166     p.prev.next = p.next;
167     theSize--;
168
169     return p.data;
170 }
171
172 /**
173  * Returns a String representation of this collection.
174  */
175 public String toString( )
176 {
177     StringBuilder sb = new StringBuilder( "[ " );
178
179     for( AnyType x : this )
180         sb.append( x + " " );
181     sb.append( "]" );
182
183     return new String( sb );
184 }
185
```

```
186  /**
187   * Obtains an Iterator object used to traverse the collection.
188   * @return an iterator positioned prior to the first element.
189   */
190  public java.util.Iterator<AnyType> iterator( )
191  {
192      return new LinkedListIterator( );
193  }
194
195  /**
196   * This is the implementation of the LinkedListIterator.
197   * It maintains a notion of a current position and of
198   * course the implicit reference to the MyLinkedList.
199   */
200  private class LinkedListIterator implements java.util.Iterator<AnyType>
201  {
202      private Node<AnyType> current = beginMarker.next;
203      private boolean okToRemove = false;
204
205      public boolean hasNext( )
206      {
207          return current != endMarker;
208      }
209
210      public AnyType next( )
211      {
212          if( !hasNext( ) )
213              throw new java.util.NoSuchElementException( );
214
215          AnyType nextItem = current.data;
216          current = current.next;
217          okToRemove = true;
218          return nextItem;
219      }
220
221      public void remove( )
222      {
223          if( !okToRemove )
224              throw new IllegalStateException( );
225
226          MyLinkedList.this.remove( current.prev );
227          okToRemove = false;
228      }
229  }
230
```

```
231  /**
232   * This is the doubly-linked list node.
233   */
234  private static class Node<AnyType>
235  {
236      public Node( AnyType d, Node<AnyType> p, Node<AnyType> n )
237      {
238          data = d; prev = p; next = n;
239      }
240
241      public AnyType data;
242      public Node<AnyType> prev;
243      public Node<AnyType> next;
244  }
245
246  private int theSize;
247  private Node<AnyType> beginMarker;
248  private Node<AnyType> endMarker;
249 }
```

## Annexe II

### java.util Class HashMap<K,V>

#### Constructor Summary

<a href="#">HashMap()</a>	Constructs an empty HashMap with the default initial capacity (16) and the default load factor (0.75).
<a href="#">HashMap(int initialCapacity)</a>	Constructs an empty HashMap with the specified initial capacity and the default load factor (0.75).
<a href="#">HashMap(int initialCapacity, float loadFactor)</a>	Constructs an empty HashMap with the specified initial capacity and load factor.
<a href="#">HashMap(Map&lt;? extends K,? extends V&gt; m)</a>	Constructs a new HashMap with the same mappings as the specified Map.

#### Method Summary

void	<a href="#">clear()</a> Removes all mappings from this map.
<a href="#">Object</a>	<a href="#">clone()</a> Returns a shallow copy of this HashMap instance: the keys and values themselves are not cloned.
boolean	<a href="#">containsKey(Object key)</a> Returns true if this map contains a mapping for the specified key.
boolean	<a href="#">containsValue(Object value)</a> Returns true if this map maps one or more keys to the specified value.
<a href="#">Set&lt;Map.Entry&lt;K,V&gt;&gt;</a>	<a href="#">entrySet()</a> Returns a collection view of the mappings contained in this map.
<a href="#">V</a>	<a href="#">get(Object key)</a> Returns the value to which the specified key is mapped in this identity hash map, or null if the map contains no mapping for this key.
boolean	<a href="#">isEmpty()</a> Returns true if this map contains no key-value mappings.
<a href="#">Set&lt;K&gt;</a>	<a href="#">keySet()</a> Returns a set view of the keys contained in this map.
<a href="#">V</a>	<a href="#">put(K key, V value)</a> Associates the specified value with the specified key in this map.
void	<a href="#">putAll(Map&lt;? extends K,? extends V&gt; m)</a> Copies all of the mappings from the specified map to this map. These mappings will replace any mappings that this map had for any of the keys currently in the specified map.
<a href="#">V</a>	<a href="#">remove(Object key)</a> Removes the mapping for this key from this map if present.
int	<a href="#">size()</a> Returns the number of key-value mappings in this map.
<a href="#">Collection&lt;V&gt;</a>	<a href="#">values()</a> Returns a collection view of the values contained in this map.

#### Methods inherited from class java.util.[AbstractMap](#)

[equals](#), [hashCode](#), [toString](#)

#### Methods inherited from class java.lang.[Object](#)

[finalize](#), [getClass](#), [notify](#), [notifyAll](#), [wait](#), [wait](#), [wait](#)

#### Methods inherited from interface java.util.[Map](#)

[equals](#), [hashCode](#)

## Annexe III

```
private static class BinaryNode<AnyType>
{
    // Constructors
    BinaryNode( AnyType theElement )
    { 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
}
```

```
private BinaryNode<AnyType> findMin( BinaryNode<AnyType> t )
{
    if( t != null )
        while( t.left != null )
            t = t.left;
    return t;
}
```

```
private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        return new BinaryNode<AnyType>( x, null, null );

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = insert( x, t.left );
    else if( compareResult > 0 )
        t.right = insert( x, t.right );
    else
        ; // Duplicate; do nothing
    return t;
}
```

```
private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t )
{
    if( t == null )
        return t; // Item not found; do nothing

    int compareResult = x.compareTo( t.element );

    if( compareResult < 0 )
        t.left = remove( x, t.left );
    else if( compareResult > 0 )
        t.right = remove( x, t.right );
    else if( t.left != null && t.right != null ) // Two children
    {
        t.element = findMin( t.right ).element;
    }
    else
        t = ( t.left != null ) ? t.left : t.right;
    return t;
}
```