Polytech Nice-Sophia **Année 2015-2016**
SI 3 **lundi 19 octobre**

Examen d'ADS 1

Durée : 2 heures. Aucun document autorisé.

*Vous devez répondre **uniquement** sur la feuille de réponse, dans les emplacements prévus à cet effet. Vous devez écrire toutes les méthodes demandées dans le **cadre strict** des classes données en annexe. En particulier, vous ne devez définir ou utiliser **aucune** autre méthode que celles demandées. Le barême est indicatif et suceptible d'être ajusté. Durant toute l'épreuve, il ne sera répondu à **aucune** question.*

1 Pile avec undo

On désire enrichir le type *pile* d'une nouvelle fonctionnalité, la fonction *undo*. Cette fonction, quand on l'appelle, annule l'effet de la dernière action *annulable* (*empiler* ou *dépiler*). Si on appelle la fonction *undo* n fois de suite, on annule les n dernières actions annulables effectuées (des *empiler* ou *dépiler*). Si on appelle la fonction *undo* alors qu'aucune fonction annulable n'a été effectuée, cet appel ne fait rien. Si la classe `UndoableStack` réalise un tel type de pile, le code suivant :

```
UndoableStack<Integer> s = new UndoableStack<Integer>();
s.push(10); s.push(20); // on empile 10 puis 20
System.out.println(s.peek()); // le sommet de la pile est 20
s.undo(); // on annule le dernier 'push'
System.out.println(s.peek()); // le sommet de la pile est 10
s.push(30); s.push(40); // on empile 30 puis 40
System.out.println(s.peek()); // le sommet de la pile est 40
s.pop(); s.pop(); // on dépile deux fois de suite
s.undo(); // on annule le dernier 'depiler'
System.out.println(s.peek()); // le sommet de la pile est 30
s.undo(); // on annule l'avant dernier 'depiler'
System.out.println(s.peek()); // le sommet de la pile est 40
```

affiche les valeurs suivantes :

```
20
10
40
30
40
```

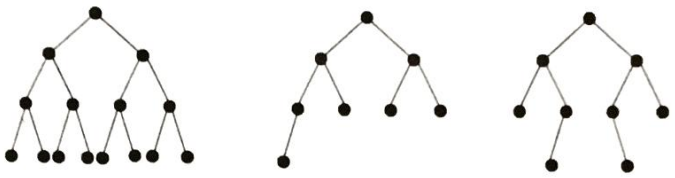
Remarquez que l'appel de la méthode `peek` n'a aucune incidence sur la méthode `undo` : cette méthode ne *changeant pas* l'état de la pile, elle ne peut pas être annulable. La classe `ArrayStack` proposée en annexe est une version simplifiée de celle étudiée en cours : on a retiré les traitements d'erreurs (les exceptions). On peut donc considérer dans la suite que les méthodes sont appelées *correctement* (par exemple, on ne fait *dépiler* que lorsque la pile n'est pas vide).

Question 1.1 (3 points) *En utilisant les attributs proposés dans la classe `UndoableStack` et sans en ajouter, écrivez le corps des méthodes `push`, `pop` et `undo`.*

Question 1.2 (1 point) *Expliquez concisément en français (pas de code) de quoi on aurait besoin si on voulait implémenter la fonction `redo` pour annuler l'effet du dernier `undo`.*

2 Taille d'un arbre binaire

On rappelle que la *taille* d'un arbre binaire est le nombre de ses noeuds. Par exemple, les arbres binaires suivants (de gauche à droite) :



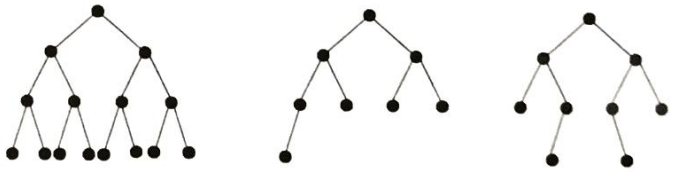
sont respectivement de taille 15, 8 et 9. On représente les arbres binaires par la classe `BinaryNode` fournie en annexe (la classe étudiée en cours).

Question 2.1 (1 point) *Complétez la méthode privée* `size` *de façon que la méthode publique de même nom calcule la taille d'un* `BinaryNode`.

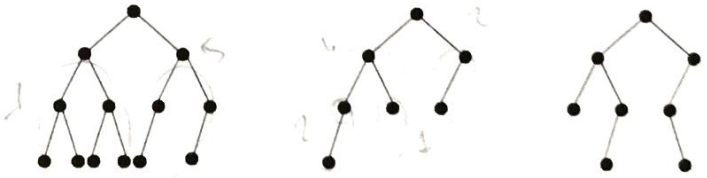
Question 2.2 (1 point) *Quelle est la complexité de cette méthode quand on l'applique sur un* `BinaryNode` *de taille n ? Justifiez votre réponse.*

3 Arbre binaire proportionné

On dit qu'un arbre binaire est *proportionné* s'il est vide, ou bien si ses deux sous-arbres gauche et droit sont *proportionnés* et la différence de taille entre ces deux sous-arbres n'excède pas 1 en valeur absolue. Par exemple, les arbres binaires suivants sont *proportionnés* :

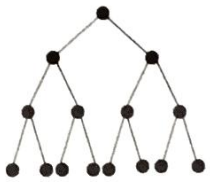


alors que les arbres binaires suivants ne sont **pas** *proportionnés* :



Question 3.1 (1 point) *Complétez la méthode privée* `proportionate1` *de façon que la méthode publique de même nom teste si un* `BinaryNode` *est proportionné. Cette méthode privée* **doit** *utiliser la méthode* `size` *de la question précédente.*

Question 3.2 (2 points) *Calculez la complexité de cette méthode quand on l'applique à un* `BinaryNode` ***parfait*** *de taille n. On rappelle qu'un arbre binaire parfait est un arbre binaire dont tous les sous-arbres (et l'arbre lui-même) sont tels que les sous-arbres gauche et droit ont exactement la même taille. Par exemple, l'arbre suivant est un arbre parfait de taille 15 :*



La méthode récursive `proportionate1` utilise la méthode `size`, qui est donc appelée un grand nombre de fois sur les mêmes sous-arbres, ce qui conduit à la complexité calculée à la question précédente. Afin d'éviter ce problème, on écrit maintenant la méthode `proportionate2` qui teste si un `BinaryNode` est proportionné et qui **en même temps** calcule sa taille.

Question 3.3 (3 points) *Complétez la méthode privée* `proportionate2` *de façon que la méthode publique de même nom teste si un* `BinaryNode` *est proportionné. Cette méthode privé* ***ne doit pas*** *utiliser la méthode* `size` *de la question 1. Cette méthode, lorsqu'on l'applique à un* `BinaryNode`*, retourne sa taille s'il est proportionné, ou bien une valeur spéciale pour indiquer qu'il n'est pas proportionné.*

Question 3.4 (1 point) *Quelle est la complexité de cette méthode dans le pire des cas quand on l'applique sur un* `BinaryNode` *de taille n ? Justifiez votre réponse.*

4 Arbres binaires de recherche

On considère la classe `BinarySearchTree` qui implémente les arbres binaires de recherche, classe étudiée en cours. On considère les méthodes publique et privée `whatisit` définies dans cette classe.

Question 4.1 (1 point) *Décrivez précisément ce que retourne la méthode publique* `whatisit` *quand on l'applique à un* `BinarySearchTree`*. Donnez un petit exemple pour illustrer votre explication.*

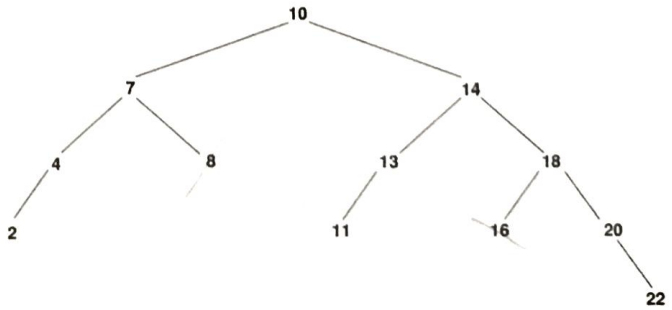
Question 4.2 (1 point) *Quelle est la complexité de cette méthode quand on l'applique à un* `BinarySearchTree` *de taille n ? Justifiez votre réponse.*

5 Arbres AVL

On rappelle qu'un arbre AVL est un arbre binaire de recherche tel que pour tout sous-arbre (y compris l'arbre lui-même) la différence de hauteur entre les sous-arbres gauche et droit est au plus égale à 1 en valeur absolue. Par ailleurs, la méthode de *suppression* dont il est question plus bas est celle qui, en cas de suppression d'un élément porté par un noeud d'arité 2, remplace cet élément par le **maximum du sous-arbre gauche**.

Question 5.1 (1 point) *Dessinez l'arbre AVL qu'on obtient en ajoutant successivement les éléments suivants dans un arbre AVL initialement vide : 2, 11, 10, 6, 4, 1, 3, 5, 8, 9, 7*

Question 5.2 (1 point) *Dessinez l'arbre AVL qu'on obtient en supprimant successivement les éléments 8, 16, 14, 13 de l'arbre AVL suivant :*

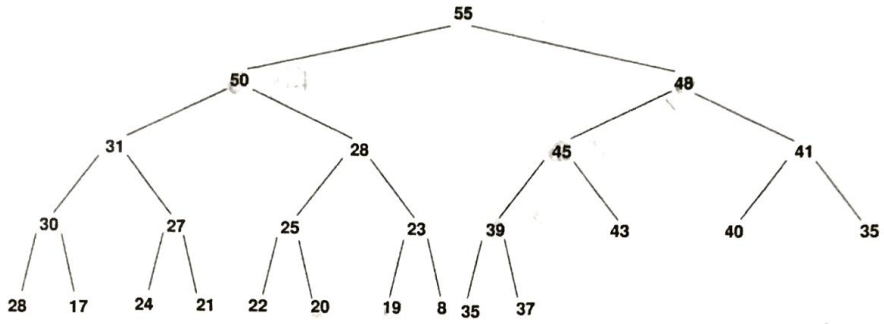


Question 5.3 (1 point) *Dessinez l'arbre AVL* ***le plus haut possible*** *(i.e. qui a la hauteur la plus grande) contenant exactement et seulement les éléments suivants : 56, 87, 29, 34, 7, 203, 65, 11, 78, 101, 93, 42*

Question 5.4 (1 point) *Dessinez l'arbre AVL* ***le moins haut possible*** *(i.e. qui a la hauteur la plus petite) contenant exactement et seulement les éléments suivants : 56, 87, 356, 29, 34, 7, 203, 65, 11, 472, 78, 101, 93, 597, 42*

6 Tas binaires

On considère le tas *maximum* suivant (l'élément maximal est à la racine) :



Question 6.1 (1 point) *Dessinez le tas obtenu après avoir appelé successivement* ***cinq fois*** *la fonction* `deleteExtreme` *(suppression de l'élément maximum) sur le tas précédent.*

Question 6.2 (1 point) *Dessinez le tas obtenu après avoir* ***ajouté*** *les valeurs 41, 49, 60, 52 et 51 au tas de la figure précédente (et* ***pas*** *au tas obtenu à la question précédente !).*

7 Tas et arbres binaires de recherche

Dans cette partie, on s'intéresse aux tas vus comme des arbres binaires et on peut donc les comparer à des arbres binaires de recherche, bien que leur *implémentation* soit totalement différente (tableau et structure arborescente). On considère ici des tas et des arbres binaires de recherche contenant des **entiers**. On rappelle par ailleurs que les valeurs contenues dans un arbre binaire de recherche sont toutes **distinctes**.

Question 7.1 (1 point) *Dessinez l'arbre binaire* ***le plus haut possible*** *contenant des entiers (choisissez les valeurs) qui soit* ***en même temps*** *un* ***arbre binaire de recherche*** *et un* ***tas maximum*** *(i.e. un tas avec la valeur maximum à la racine)*

Question 7.2 (1 point) *Dessinez l'arbre binaire* ***le plus haut possible*** *contenant des entiers (choisissez les valeurs) qui soit* ***en même temps*** *un* ***arbre binaire de recherche*** *et un* ***tas minimum*** *(i.e. un tas avec la valeur minimum à la racine)*

```java
/**
 * An array-based stack class
 */
public class ArrayStack<AnyType> {

        private static final int DEFAULT_CAPACITY = 128;

        private AnyType[] array;
        private int size;

        /**
         * Build an empty stack
         * Complexity: THETA(1)
         */
        public ArrayStack() {
                array = (AnyType[]) new Object[DEFAULT_CAPACITY];
                size = 0;
        }

        /**
         * Check if the stack is empty
         * Complexity: THETA(1)
         */
        public boolean isEmpty() {
                return size == 0;
        }

        /**
         * Return the next value to be popped from the stack
         * Complexity: THETA(1)
         */
        public AnyType peek() {
                if ( isEmpty() )
                        return null;
                return array[size-1];
        }

        /**
         * Push the value x onto the stack.
         * Complexity: THETA(1)
         */
        public void push(AnyType x) {
                if ( size < array.length )
                        array[size++] = x;
        }

        /**
         * Pop the stack and return the value popped
         * Complexity: THETA(1)
         */
        public AnyType pop() {
                if ( isEmpty() )
                        return null;
                return array[--size];
        }
}
```



```java
/**
 * A class for undoable stacks
 */
public class UndoableStack<AnyType> {

        private static final String pushAction = "push";
        private static final String popAction  = "pop";

        private ArrayStack<AnyType> stack;
        private ArrayStack<AnyType> popStack;
        private ArrayStack<String> actionStack;

        /**
         * Build an undoable stack
         * Complexity: THETA(1)
         */
        public UndoableStack() {
                stack = new ArrayStack<AnyType>();
                popStack = new ArrayStack<AnyType>();
                actionStack = new ArrayStack<String>();
        }

        /**
         * Check if the stack is empty
         * Complexity: THETA(1)
         */
        public boolean isEmpty() {
                return stack.isEmpty();
        }

        /**
         * Return the next value to be popped from the stack
         * Complexity: THETA(1)
         */
        public AnyType peek() {
                return stack.peek();
        }

        /**
         * Push the value x onto the stack.
         * Complexity: THETA(1)
         */
        public void push(AnyType x) {
            <TO BE COMPLETED>
        }

        /**
         * Pop the stack and return the value popped
         * Complexity: THETA(1)
         */
        public AnyType pop() {
            <TO BE COMPLETED>
        }

        /**
         * Undo the last effective 'push' or 'pop'
         * Complexity: THETA(1)
         */
        public void undo() {
            <TO BE COMPLETED>
        }
}
```

SI 3 Page 3/6

```
/**
 * A class for simple binary nodes
 */
public class BinaryNode<AnyType> {

        private AnyType data;
        private BinaryNode<AnyType> left, right;

        ////////////////// constructors

        /**
         * Build a binary node which is
         * a leaf holding the value 'data'
         */
        public BinaryNode(AnyType data) {
                this.data = data;
                this.left = null;
                this.right = null;
        }

        ////////////////// accessors

        public AnyType data() {
                return data;
        }

        public BinaryNode<AnyType> left() {
                return left;
        }

        public BinaryNode<AnyType> right() {
                return right;
        }

        ////////// size

        /**
         * Return the size of the BinaryNode
         */
         public int size() {
                return size(this);
         }

        /**
         * Return the size of the BinaryNode t
         */
        private int size(BinaryNode<AnyType> t) {
            <TO BE COMPLETED>
        }
```

SI 3 Page 4/6

```
        ////////// proportionate, version 1

        /**
         * Check if the BinaryNode is proportionate
         */
        public boolean proportionate1() {
                return proportionate1(this);
        }

        /**
         * Check if the BinaryNode t is proportionate
         */
        private boolean proportionate1(BinaryNode<AnyType> t) {
            <TO BE COMPLETED>
        }


        ////////// proportionate, version 2

        private static final int NOT_PROPORTIONATE = -1;
        // or whatever value < 0

        /**
         * Check if the BinaryNode is proportionate
         */
        public boolean proportionate2() {
                int s = proportionate2(this);
                return s != NOT_PROPORTIONATE;
        }

        /**
         * Check if the BinaryNode t is proportionate.
         * If t is proportionate, return the size of t
         * else return NOT_PROPORTIONATE
         */
        private int proportionate2(BinaryNode<AnyType> t) {
            <TO BE COMPLETED>
        }
}
```

```
/**
 * A class for Binary Search Trees
 */
public class BinarySearchTree<AnyType extends Comparable<? super AnyType>> {

    private BinaryNode<AnyType> root;

    /**
     * Construct an empty BST
     */
    public BinarySearchTree( ) {
        root = null;
    }

    /**
     * Check if the BST is empty
     */
    public boolean isEmpty( ) {
        return root == null;
    }

    /**
     * Find an item in the BST
     */
    public boolean contains( AnyType x ) {
        return contains( x, root );
    }

    /**
     * Internal method to find an item in a BST
     */
    private boolean contains( AnyType x, BinaryNode<AnyType> t ) {
        ....
    }

    /**
     * Insert into the BST. Duplicates are ignored.
     */
    public void insert( AnyType x ) {
        root = insert( x, root );
    }

    /**
     * Internal method to insert into a BST
     */
    private BinaryNode<AnyType> insert( AnyType x, BinaryNode<AnyType> t ) {
        ....
    }

    /**
     * Remove from the BST. Nothing is done if x is not found.
     */
    public void remove( AnyType x ) {
        root = remove( x, root );
    }

    /**
     * Internal method to remove from a BST
     */
    private BinaryNode<AnyType> remove( AnyType x, BinaryNode<AnyType> t ) {
        ....
    }
```



```
    /////////////// whatisit


    public ArrayStack<AnyType> whatisit() {
        ArrayStack<AnyType> stack = new ArrayStack<AnyType>();
        whatisit(root,stack);
        return stack;
    }


    private void whatisit(BinaryNode<AnyType> t, ArrayStack<AnyType> stack) {
        if ( t != null ) {
                whatisit(t.right, stack);
                stack.push(t.element);
                whatisit(t.left, stack);
        }
    }



    ////////////////////////////////////////////////////
    // Inner class BinaryNode<AnyType>
    ////////////////////////////////////////////////////

    // Basic node stored in unbalanced binary search trees
    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
    }
}
```