POLYTECHNIQUE MONTRÉAL

Questionnaire
Contrôle périodique

26.5/50

INF2010

Sigle du cours

Identification de l'étudiant(e)
[redacted]

Sigle et titre du cours			
INF2010 Structures de données et algorithmes			
Professeur	**Téléphone**	**Groupe**	**Trimestre**
Samuel Kadoury	4262	Tous	2024-1
Jour	**Date**	**Durée**	**Heures**
Lundi	11 mars 2024	2h	12h45

Documentation	Calculatrice	Outils électroniques
☒ Aucune ☐ Toute ☒ Voir directives particulières	☐ Aucune ☐ Toutes ☒ Non programmable (AEP)	**Les appareils électroniques personnels sont interdits.**

Directives particulières

- Le professeur ne répondra à aucune question durant cet examen. Si vous estimez que vous ne pouvez pas répondre à une question pour diverses raisons, veuillez le justifier puis passer à la question suivante.
- Assurez-vous que le crayon que vous utilisez soit suffisamment lisible pour le correcteur.
- Vous pouvez utiliser le cahier brouillon, mais vous devez répondre sur le questionnaire.

Important

Cet examen contient 7 questions sur un total de 15 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

L'étudiant doit honorer l'engagement pris lors de la signature du code de conduite.

1 Pile et File (4pts)

La fonction *generateList* convertit une liste chaînée en une nouvelle liste en passant par *Structure*, une structure intermédiaire qui peut agir soit comme une pile soit comme une file. Voici le code:

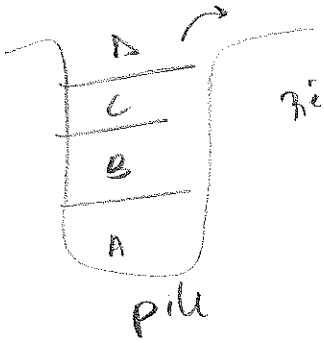
```
public static List generateList(List input) {
    Structure structure = new Structure();
    for (; input != null; input = input.next)
        structure.add(input.val);
    List result = null;
    while (!structure.isEmpty())
        result = new List(structure.remove(), result);
    return result;
}
```

La classe ***Structure*** est définie avec les deux méthodes principales suivantes:

- *add(val)*: Cette méthode ajoute un élément à la structure.
- *remove()*: Cette méthode retire un élément de la structure puis le retourne.

En considérant la liste d'entrée suivante, ***input***: $A \rightarrow B \rightarrow C \rightarrow D$

1.1 *(2pts)* Si *Structure* agit comme une **pile**, quel est la liste retournée après l'appel de la fonction *generateList* sur *input*?

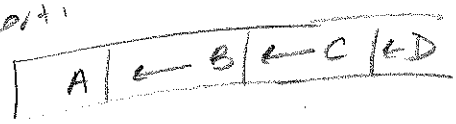


Rép: D → C → B → A

2

1.2 *(2pts)* Si *Structure* agit comme une **file**, quel est la liste retournée après l'appel de la fonction *generateList* sur *input*?

file = premier arrivé premier sorti



Rép: A → B → C → D

2

2 Tableau et Liste Chaînée (8pts)

Considérez le fragment de code suivant:

```
ArrayList<Integer> arrayList = new ArrayList<Integer>(10);
LinkedList<Integer> linkedList = new LinkedList<Integer>();
...
arrayList.add(10);
arrayList.add(i, 20);
linkedList.add(30);
```

list chaînée = pas de taille fixe

i

Indice: La méthode `add(int index, E element)` insère un élément à un index et décale les éléments suivants.

2.1 *(2pts)* Déterminez la complexité en pire cas de l'insertion de l'élément 10 dans `arrayList` avec un espace suffisant. Justifiez votre réponse. Est-ce que cette complexité change si l'espace est insuffisant? Expliquez le raisonnement.

décale les élément = itère à travers n elements avec un index = O(n) | à espace insuffisant: itèrera tjrs à travers l'index i itèrera tjrs à travers les n valeurs.

pire cas = 10 n'est pas là. je décale 10 elem. sans les trier. C'est bien O(n)

1

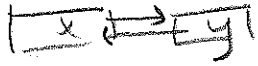
2.2 *(2pts)* Déterminez la complexité en pire cas de l'insertion de l'élément 20 à l'indice `i` dans `arrayList` avec un espace suffisant. Justifiez votre réponse. Cette complexité change-t-elle si l'espace est insuffisant? Fournissez le raisonnement.

O(n).

1

2.3 *(2pts)* En considérant que `linkedList` est une liste doublement chaînée, quelle est la complexité en pire cas pour l'ajout de l'élément 30? Justifiez votre réponse.

O(log n), une liste doublement chainée possède 2 pointeurs dans les 2 directions



0

2.4 *(2pts)* Si `linkedList` est une liste simplement chaînée, quelle est la complexité en pire cas pour l'ajout de l'élément 30? Justifiez votre réponse.

O(log n)

0

3 Analyse de Complexité (5pts)

Considérez les méthodes suivantes:

```
public void printA(int n) {
    for (int i = 0; i < n * n; i++) {
        if (i % 2 == 0) {
            for (int j = 0; j < i; j++) {
                System.out.println("...");
            }
        }
    }
}

public void printB(int n) {
    if (n > 100) {
        System.out.println("...");
        printB(n / 2);
    }
}
```

3.1 *(2.5pts)* Quelle est la complexité en pire cas de la méthode `printA(int n)`? Expliquez votre raisonnement.

$O(n^2)$ car il y a for loop imbriqués.

for (int i; ...) est $O(n)$
if ... $O(n)$
↳ for(int j; ...) $O(n) \times O(n)$

Rép: j'ai donc $O(n^2)$

0.5

3.2 *(2.5pts)* Quelle est la complexité en pire cas de la méthode `printB(int n)`? Expliquez votre raisonnement.

$O(1)$! c'est un if statement simple,
la méthode prend en param un entier (une constante),
et toute ses opérations sont simple, ne nécessitent
pas d'incrémentation, d'iteration à travers tableau... etc.

0

4 Tables de Dispersement (11pts)

Soit une table de dispersement avec sondage quadratique Hash(clé) = $(clé+i^2)\%N$ dont l'implémentation est fournie à l'Annexe 1 à titre de référence. Sachant que les clés suivantes ont été insérées dans cet ordre:

1, 18, 70, 27, 57, 44

4.1 *(6pts)* Donnez l'état de la table après les insertions initiales

N = 13 = taille hashmap

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Valeurs		1	27			18	70		44	57			

Donnez les détails de vos calculs

1 mod 13 = ((1 ÷ 13) − 0) × 13 = 1

6

18 mod 13 = ((18 ÷ 13) − 1) × 13 = 5

70 mod 13 = ((70 ÷ 13) − 5) × 13 = 5 = collision

↳ 71 mod 13 = ((71 ÷ 13) − 5) × 13 = 6

27 mod 13 = ((27 ÷ 13) − 2) × 13 = 1 = collision

↳ 28 mod 13 = ((28 ÷ 13) − 2) × 13 = 2

57 mod 13 = ((57 ÷ 13) − 4) × 13 = 5 = collision

↳ (58 mod 13) = ((58 ÷ 13) − 4) × 13 = 6 = collision

↳ $(57 + 2^2)$ mod 13 = ((61 ÷ 13) − 4) × 13 = 9

44 mod 13 = ((44 ÷ 13) − 3) × 13 = 5 = coll

↳ 45 mod 13 = ((45 ÷ 13) − 3) × 13 = 6 = coll

↳ (44 + 4) mod 13 = ((48 ÷ 13) − 3) × 13 = 9 = coll

↳ $(44 + 3^2)$ mod 13 = ((53 ÷ 13) − 4) × 13 = 1 = coll

↳ $(44 + 4^2)$ mod 13 = (60 ÷ 13) − 4) × 13 = 8 ✓

4.2 *(2pts)* On effectue un appel à `remove(30)` sur la table de dispersement quadratique suivante. Fournissez les étapes de cet appel en étant bref et précis. Référez-vous au code source fourni à l'Annexe 1.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Valeurs					4	17	6	19	30				

Remove(30) :
30 mod 13 = ((30 ÷ 13) - 2) x 13 = 4 → coll
31 mod 13 = 5 = coll
(30+4) mod 13 = 8 = coll
(30+3²) mod 13 = 0 ✓ ∅ de collision, BoolisActive is false

2

4.3 *(2pts)* On effectue un appel à `remove(20)` puis `insert(46)` sur la table de dispersement quadratique suivante. Fournissez les étapes de ces appels en étant bref et précis. Référez-vous au code source fourni à l'Annexe 1.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
Valeurs				46				7	33		10	20	

Remove 20 :
20 mod 13 = 7 = coll.
21 mod 13 = 8 = coll
(20+2²) mod 13 = 11 = coll
(20+3²) mod 13 = 3 ✓ ∅ de coll = BoolisActive is false

insert(46)
46 mod 13 = 7 = coll
47 mod 13 = 8 = coll
(46+2²) mod 13 = 11 = coll
(46+3²) mod 13 = 3 ✓

2

4.4 *(1pt)* Combien de nouvelles clés doivent être insérées dans la table de dispersement présentée à la question 4.3 pour déclencher l'appel à la fonction `rehash()`? Référez-vous au code source fourni à l'Annexe 1.

$f_c \geq 0,5$

$f_c = \frac{\#\text{d'element}}{\text{taille table}} = \frac{5}{13} = 0,38$

+1elem $= \frac{6}{13} = 0,46$

+1 $= \frac{7}{13} = 0,54$

Rép : 2 nouvelles clés
$= f_c \geq 0,5 \rightarrow$ on doit donc appeler à rehash()

0.5

5 QuickSort (12pts)

On souhaite utiliser l'algorithme QuickSort, dont l'implémentation est la même que celle vue en classe (voir Annexe 2), pour trier le tableau ci-dessous, en utilisant un CUTOFF de 5.

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Valeurs	34	7	23	32	5	62	78	4	9	12	15	21	2	30

pivot = 2

5.1 *(2pts)* Donnez l'état du tableau après la mise à l'écart du pivot de la première récursion de QuickSort:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Valeurs	7	23	32	5	62	78	2	4	9	12	15	21	30	34

0.5

tt ce qui est à gauche du pivot < 2

5.2 *(2pts)* Donnez l'état du tableau après l'exécution du partitionnement de la première récursion de QuickSort:

Index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Valeurs	2	7	23	32	5	62	78	4	9	12	15	21	30	34

0.5

5.3 *(2pts)* Au total, combien de fois la fonction récursive QuickSort aura été appelée pour exécuter le tri?
Signature de la fonction à considérer:

```
private static <AnyType extends Comparable<? super AnyType>> void
quicksort( AnyType [ ] a, int left, int right )
```

Votre réponse:

5

0

5.4 *(2pts)* Au total, combien de fois la fonction récursive QuickSort aurait été appelée si la valeur CUTOFF avait été 3 au lieu de 5?

Votre réponse:

7

1

5.5 *(2pts)* Expliquez la différence entre les complexités en pire cas de QuickSort et MergeSort de façon générale.

Quick sort ne divise pas le tableau en 2, n'a donc pas besoin de trouver une médiane. $T(\frac{N}{2}) + N$ → complexité pour 1 tableau en 2.

Les 2 demeurent $O(n \log n)$ toutefois

1

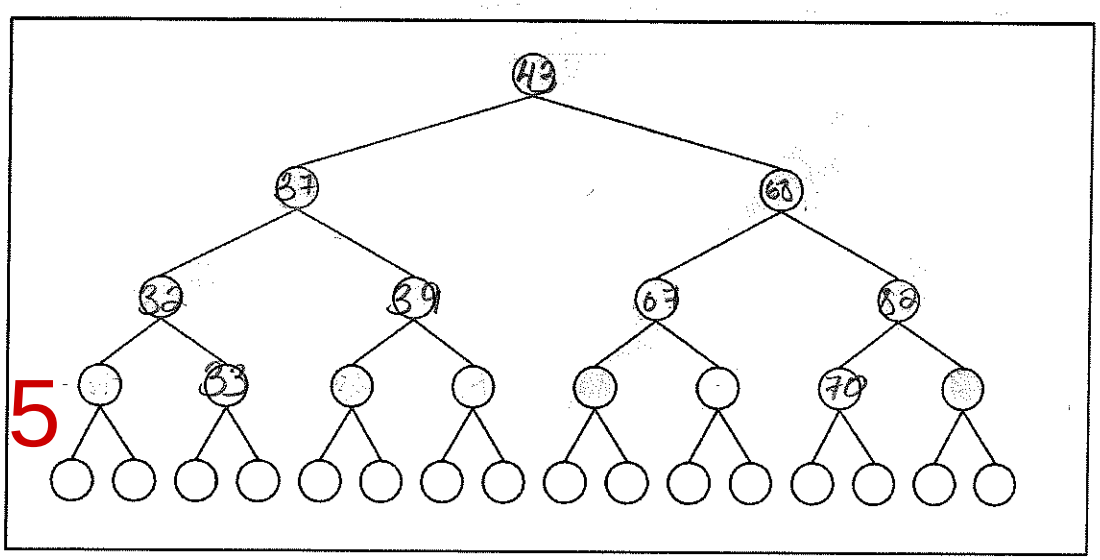
5.6 *(2pts)* Pourquoi privilégier QuickSort plutôt que MergeSort en pratique? Quel(s) avantage(s) justifie(nt) son utilisation?

Merge sort nécessite plus d'appels à sa fonction récursive pour trier 1 tableau de taille égale, dans le pire comme dans le mauvais cas, Quick sort fait le tableau avec - d'appels à sa fonction

1.5

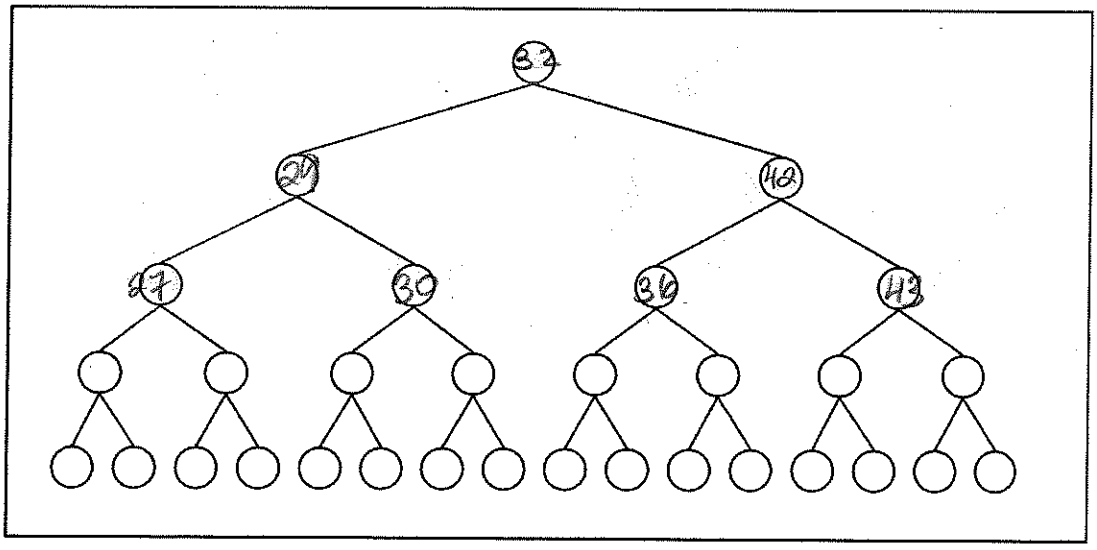
6 Arbre Binaire de Recherche (4pts)

6.1 *(2pts)* À partir de la séquence suivante, reconstruisez l'arbre binaire de recherche (BST):
Séquence en post-ordre : 32, 39, 37, 33, 68, 67, 82, 70, 43



0.5

6.2 *(2pts)* À partir de la séquence suivante, reconstruisez l'arbre binaire de recherche (BST):
Séquence en pré-ordre : 36, 27, 30, 29, 32, 43, 42

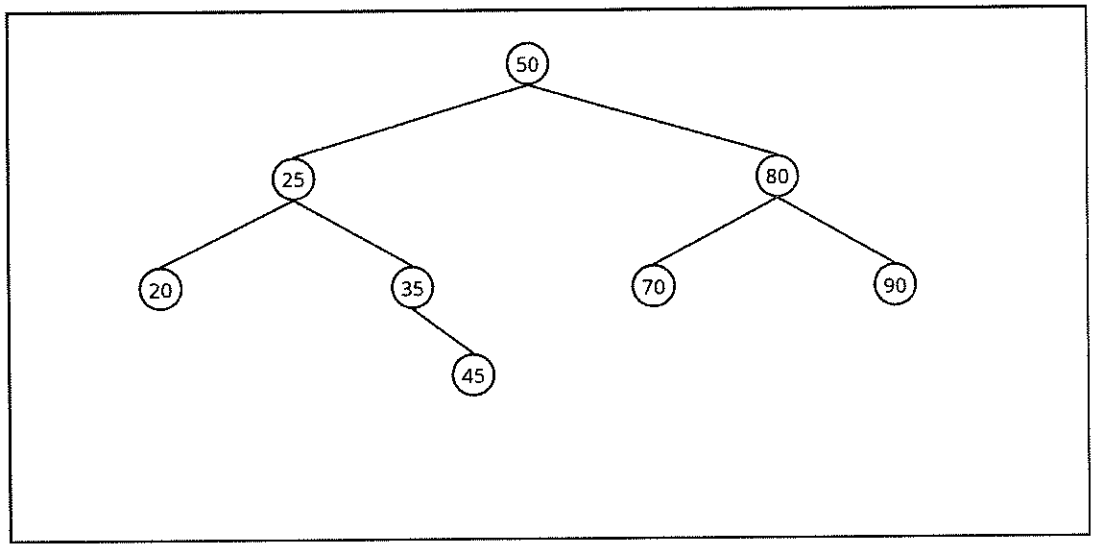


0

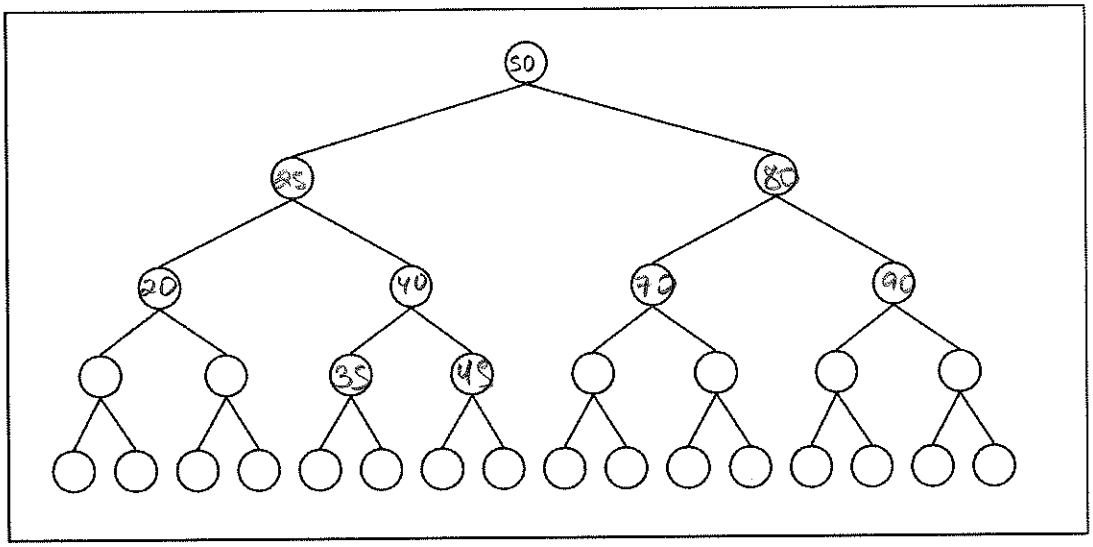
7 Arbre AVL (6pts)

Pour chaque opération listée ci-dessous, dessinez l'arbre AVL résultant.

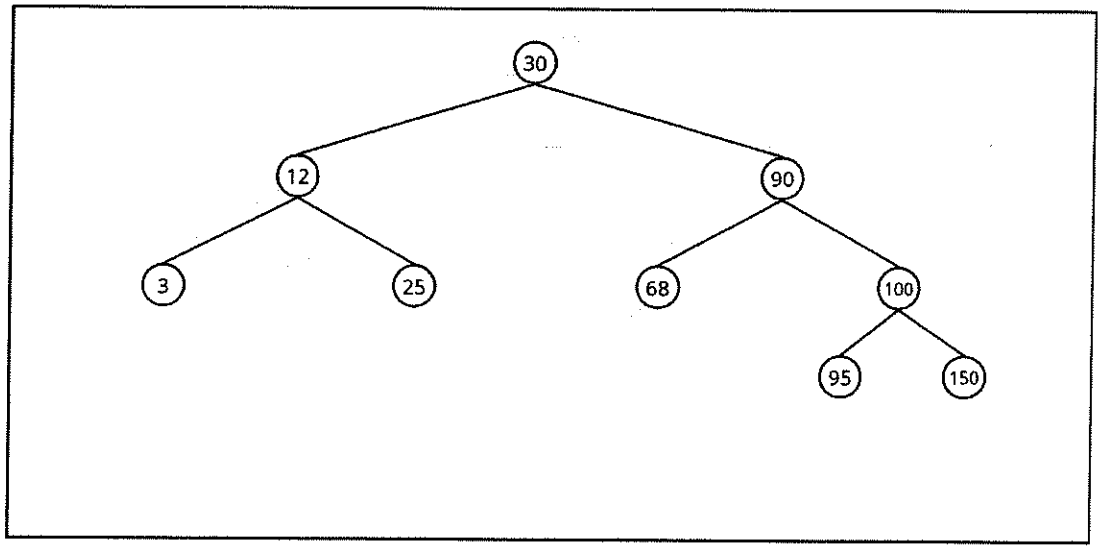
7.1 *(2pts)* Insérez 40.



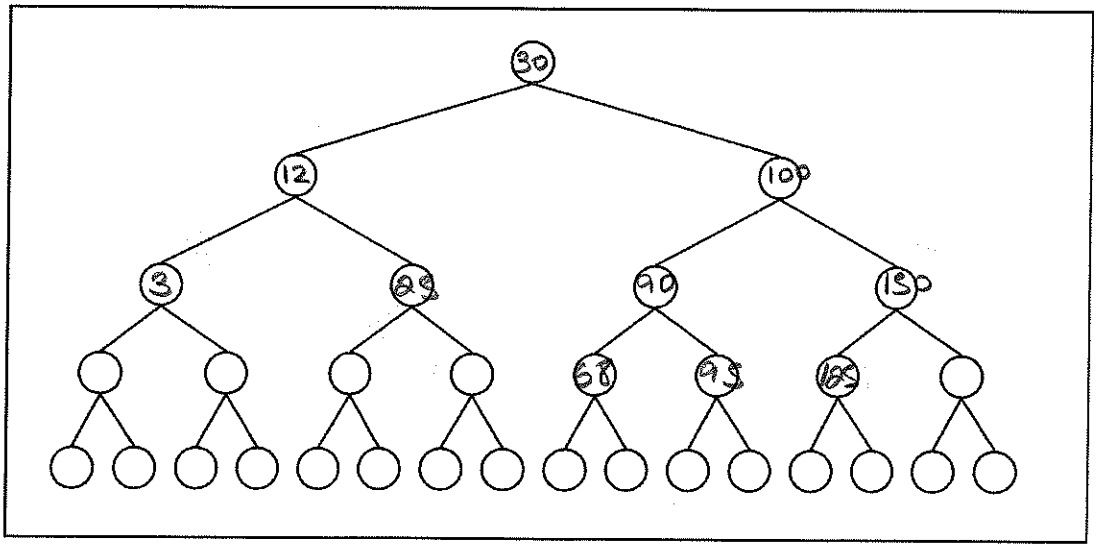
2



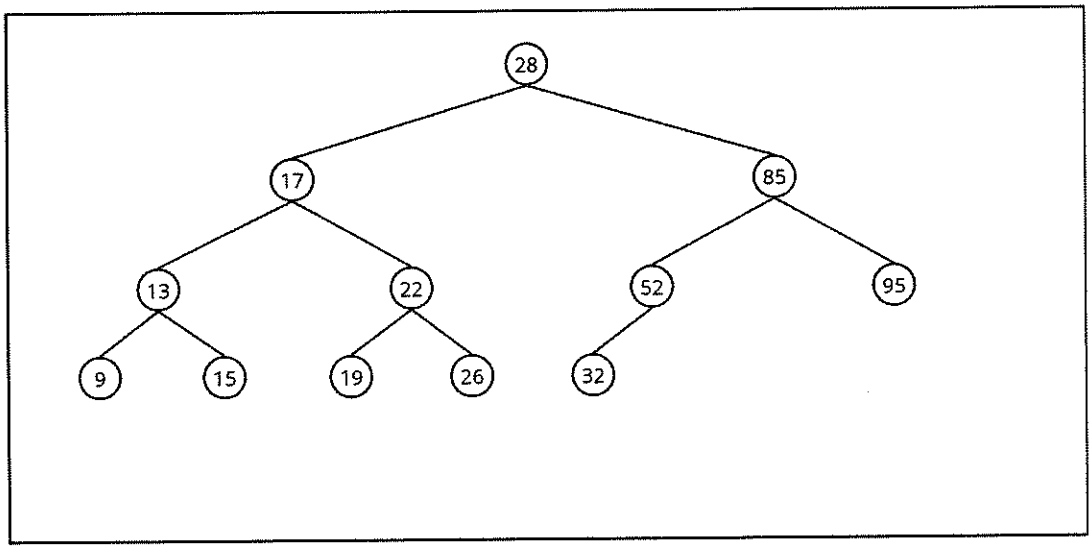
7.2 *(2pts)* Insérez 125.

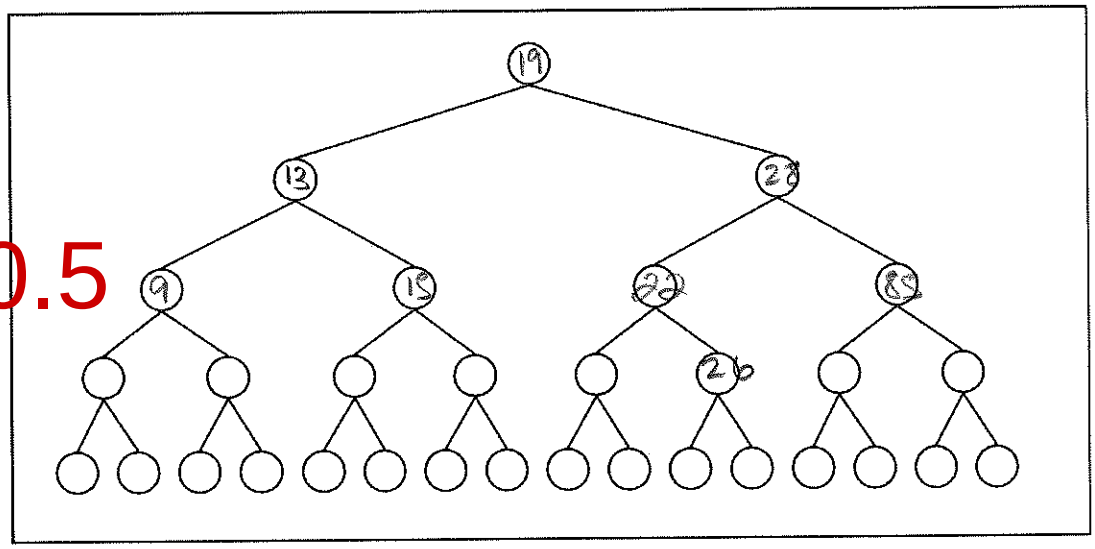


2



7.3 *(2pts)* Retirez 17.





0.5

Annexe 1

```
public class QuadraticProbingHashTable<AnyType> {
    private static final int DEFAULT_TABLE_SIZE = 13;
    private HashEntry<AnyType>[] array;
    private int currentSize;

    public QuadraticProbingHashTable() {
        this(DEFAULT_TABLE_SIZE);
    }

    public QuadraticProbingHashTable(int size) {
        allocateArray(size);
        makeEmpty();
    }

    public void insert(AnyType x) {
        int currentPos = findPos(x);
        if (isActive(currentPos)) return;
        array[currentPos] = new HashEntry<AnyType>(x, true);

        if (++currentSize > array.length / 2) rehash();
    }

    private void rehash() {
        HashEntry<AnyType>[] oldArray = array;

        allocateArray(2 * oldArray.length);
        currentSize = 0;

        for (int i = 0; i < oldArray.length; i++)
            if (oldArray[i] != null && oldArray[i].isActive)
                insert(oldArray[i].element);
    }

    private int findPos(AnyType x) {
        int offset = 1;
        int currentPos = myhash(x);

        while (array[currentPos] != null && !array[currentPos].element.equals(x)) {
            currentPos += offset;
            offset += 2;
            if (currentPos >= array.length)
                currentPos -= array.length;
        }

        return currentPos;
    }

    public void remove(AnyType x) {
        int currentPos = findPos(x);
        if (isActive(currentPos))
            array[currentPos].isActive = false;
        currentSize--;
```

```
    }

    public boolean contains(AnyType x) {
        int currentPos = findPos(x);
        return isActive(currentPos);
    }

    private boolean isActive(int currentPos) {
        return array[currentPos] != null && array[currentPos].isActive;
    }

    public void makeEmpty() {
        currentSize = 0;
        for (int i = 0; i < array.length; i++)
            array[i] = null;
    }

    private int myhash(AnyType x) {
        int hashVal = x.hashCode();
        hashVal %= array.length;
        if (hashVal < 0)
            hashVal += array.length;
        return hashVal;
    }

    private static class HashEntry<AnyType> {
        public AnyType element;
        public boolean isActive;

        public HashEntry(AnyType e, boolean i) {
            element = e;
            isActive = i;
        }
    }

    @SuppressWarnings("unchecked")
    private void allocateArray(int arraySize) {
        array = new HashEntry[nextPrime(arraySize)];
    }

    private static int nextPrime(int n) {
        if (n % 2 == 0) n++;
        for (; !isPrime(n); n += 2) ;
        return n;
    }

    private static boolean isPrime(int n) {
        if (n == 2 || n == 3) return true;
        if (n == 1 || n % 2 == 0) return false;
        for (int i = 3; i * i <= n; i += 2)
            if (n % i == 0) return false;
        return true;
    }
}
```

Annexe 2

```
public final class Sort {
    private static final int CUTOFF = 5;

    public static <AnyType extends Comparable<? super AnyType>>
    void quicksort(AnyType[] a) {
        quicksort(a, 0, a.length - 1);
    }

    public static <AnyType> void swapReferences(AnyType[] a, int index1, int index2) {
        AnyType tmp = a[index1];
        a[index1] = a[index2];
        a[index2] = tmp;
    }

    private static <AnyType extends Comparable<? super AnyType>>
    AnyType median3(AnyType[] 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];
    }

    private static <AnyType extends Comparable<? super AnyType>>
    void insertionSort(AnyType[] a, int left, int right) {
        System.out.println("DEBUG: call to insertion sort");
        for (int p = left + 1; p <= right; p++) {
            AnyType 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;
        }
    }

    private static <AnyType extends Comparable<? super AnyType>>
    void quicksort(AnyType[] a, int left, int right) {
        if (left + CUTOFF <= right) {
            AnyType pivot = median3(a, left, right);
            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

                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);
            }
        }

        public static <AnyType extends Comparable<? super AnyType>>
        void printArray(AnyType[] a) {
            for (AnyType item : a)
                System.out.print(item + " ");
            System.out.println();
        }
    }
```