le cnam
Grand Est

# Algorithmes avancés

## Exercice 1

Soient A, B, C trois nombre que vous devez faire saisir par l'utilisateur :

- Calculez la factorielle de A
- Calculez la factorielle de B
- Calculez la factorielle de C
- Résolvez l'équation $Ax + B = 0$
- Résolvez l'équation $Bx + C = 0$
- Résolvez l'équation $Cx + A = 0$
- Résolvez l'équation $Ax^2 + Bx + C = 0$
- Résolvez l'équation $Bx^2 + Cx + A = 0$
- Résolvez l'équation $Cx^2 + Ax + B = 0$

## Exercice 2 – les listes

### 2.1 La classe List

Créez la classe List<T>. Elle devra contenir les fonctions abstraites suivantes :

- public int size() : envoit la taille de la liste
- public T get(i) : revoit le ième élément de la liste
- public void add(T e) : ajoute l'élément e à la fin de la liste
- public void add(T e, int index) : ajoute l'élément e à la position spécifiée dans la liste
- public boolean contains(T e) : renvoie vraie si la liste contient l'élément spécifié en paramètre
- public boolean remove(T e) : supprime l'élément e de la liste. Renvoit vrai si l'élément a été trouvé.
- public void remove(int index) : supprime l'élément N° index de la liste

### 2.2 La classe Array

Créez la classe Array<T> (Tableau). Elle devra hériter de la classe List et devra remplir toutes les fonctions abstraites de la classe List. Par ailleurs, le tableau devra réserver un nombre de cases supplémentaires pour d'éventuelles utilisations ultérieures. Pour vous aider, la classe Array devra posséder les fonctions suivantes :

- private void agrandirSiNecessaire(int taille)
- private void retrecirSiNecessaire()

### 2.3 La classe LinkedList

Créez la classe ChainedList<T> (List chaînée). Elle devra hériter de la classe List et devra remplir toutes les fonctions abstraites de la classe List. Pour vous aider, vous devrez créer une classe Cell<T>, qui devra contenir

- la valeur de la cellule,

**Jason Mahdjoub**
*jason.mahdjoub@distri-mind.fr*

- la référence vers la cellule suivante
- la référence vers la cellule précédente

## 2.4 Le tri

Dans la classe List, codez une fonction qui permet de trier la liste grâce au tri par sélection. Vous pouvez vous aider de http://lwh.free.fr internet pour visualiser le fonctionnement du tri. Voici l'entête de la fonction à créer :

```
public void sort(Comparator<T> comparator)
```

Reportez-vous à la documentation de l'interface Comparator pour comprendre son fonctionnement : https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

En option, vous pouvez développer la même fonction, qui surcharge la fonction « sort » dans la classe Array, pour implémenter l'algorithme de tri rapide.

## 2.5 La recherche dichotomique

La recherche dichotomique consiste à rechercher une valeur dans un tableau, mais en divisant le problème par 2, à chaque cycle de recherche. Cet algorithme ne fonctionne qu'avec des tableaux déjà triés. Le principe est de sélectionner la case du milieu. Si la valeur recherchée est plus petite que celle de la case du milieu, on la recherchera dans la première moitié du tableau. Sinon on la recherchera dans la seconde moitié. On divise ainsi chaque portion du tableau 2, à chaque cycle de recherche, jusqu'à trouver, ou non, la valeur recherchée.

Dans la classe Array, implémentez la recherche dichotomique à travers la fonction suivante :

public boolean containsDicho(T element)

## 2.6 La recherche dichotomique (récursive)

Écrivez la version récursive de la recherche dichotomique

## 2.7 La classe Stack

Créez la classe Stack<T> (Pile). Elle devra hériter de la classe ChainedList et devra remplir les fonctions suivantes :

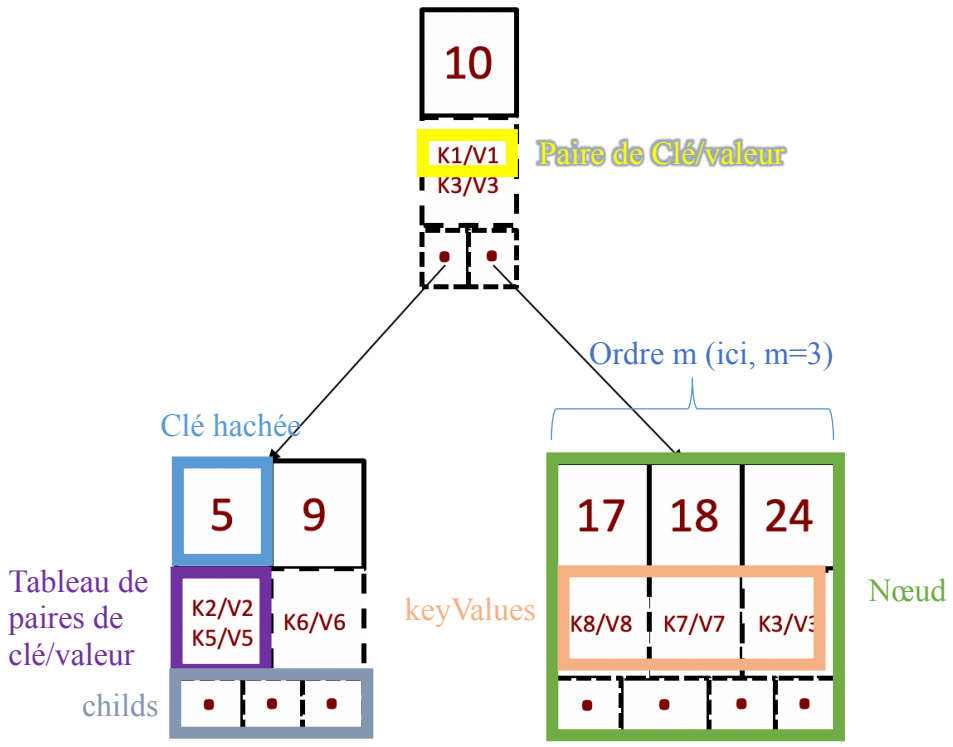
- public void push(T value) : empile la nouvelle valeur
- public T pop() : dépile une valeur et la retourne

## 2.8 La classe HashMap

Sur la base :

- du cours que vous avez vu,
- du lien suivant : https://cis.stvincent.edu/html/tutorials/swd/btree/btree.html
- et du schéma ci-dessous

Implémentez un tableau associatif (dictionnaire).




### 2.8.1 Classe KeyValue

Créez la classe suivante :

```
class KeyValue {
	private final Object key;
	private Object value;

	KeyValue(Object key, Object value)
	{
		this.key=key;
		this.value=value;
	}

	Object getValue() {
		return value;
	}

	void setValue(Object value) {
		this.value = value;
	}

	Object getKey() {
		return key;
	}

}
```

### 2.8.2 Classe HashMap

Créez la classe suivante :

```
public class HashMap {
	private final Node root;
	private int size;
```

```
	public HashMap(int order)
	{
		root=new Node(order, null);
		this.size=0;
	}

	public Object put(Object key, Object value)
	{
		Object res=root.put(new KeyValue(key, value));
		if (res==null)
			++size;
		return res;
	}

	public Object get(Object key)
	{
		return root.get(key);
	}

	public Object remove(Object key)
	{
		Object res=root.remove(key);
		if (res!=null)
			--size;
		return res;
	}

	public int size()
	{
		return size;
	}

}
```

### 2.8.3 Classe Node

Créez et complétez la classe suivante :

```
public class Node {
	private final int order;
	private final Node parent;
	private final ArrayList<Integer> hachedKeys=new ArrayList<>();
	private final ArrayList<ArrayList<KeyValue>> keyValues=new ArrayList<>();
	private final ArrayList<Node> childs=new ArrayList<>();
	Node(int order, Node parent, KeyValue ...keyValues)
	{
		this.order=order;
		this.parent=parent;
		childs.add(null);
		for (KeyValue kv : keyValues)
			put(kv);
	}

	Object put(KeyValue keyValue)
	{
		Node n=findNearestNode(keyValue.getKey());
		return n.putKeyValueHere(keyValue);

	}

    Node findNearestNode(Object key)
	{
	int hachedKey=key.hashCode();
```

```
		/*TODO retourner le noeud le plus proche,
		 * correspondant à la clé hachée de keyValue
		 **/

		return null;
	}



	private Object putKeyValueHere(KeyValue keyValue)
	{
		int hachedKey=keyValue.getKey().hashCode();
		/*TODO Si nécéssaire, ajouter la clée hachée correspondant
		 * à la clé/valeur dans le tableau hachedKeys
		 *
		 * S'il y a plus de clé/valeur que le nombre contenu dans 'order',
		 * créer des noeuds fils en conséquence, et mettre à jour le noeud courant.
		 **/

		/*TODO ajouter la clé/valeur au tableau keyValues,
		 * dans le même ordre correspondant au tableau de clés hachées.
		 *
		 * Si la clé existe déjà, remplacer la valeur associée à cette clé
		 *
		 **/

		/*TODO
		 * Mettre à jour la liste de nœuds fils
		 **/


		/*
		 * TODO retourner la valeur précédente, associée à la même clée
		 * S'il n'y avait pas de valeurs associées, retourner null
		 */
		return null;
	}

	Object get(Object key)
	{
		Node n=findNearestNode(key);
		return n.getHere(key);

	}

	private Object getHere(Object key)
	{
		int hachedKey=key.hashCode();
		/*
		 * TODO retourner la pair de clé/valeur correspondant à la clé donnée.
		 * Rechercher dans les noeuds fils si nécéssaire.
		 */

		return null;
	}

	Object remove(Object key)
	{
		Node n=findNearestNode(key);
		return n.removeHere(key);
	}
	private Object removeHere(Object key)
	{
		int hachedKey=key.hashCode();
```

```
            /*
             * TODO supprimer la clé/valeur qui correspond à la clé donnée.
             *
             * Si la clé haschée n'est plus associée à aucune paire clé/valeur,
             * supprimer la clé hachée et mettre à jour le noeud
             *
             * Si le noeud ne contient plus aucune donnée,
             * supprimer le noeud dans le noeud parent s'il y a un noeud parent.
             *
             */

            /*TODO
             * Mettre à jour la liste de nœuds fils
             **/

            /*
             * TODO retourner la valeur précédente, associée à la même clée
             * S'il n'y avait pas de valeurs associées, retourner null
             */
            return null;
        }

}
```

### 2.8.4 Tests

Testez les fonctions put, get, et remove de la classe HashMap.

## Exercice 3 – L'indexation (utilisation des tableaux associatifs)

Soit la variable 'livre' (en Texte) contenant un gros volume de caractères. Nous souhaitons rechercher dans notre livre toutes les positions où le mot contenu dans la variable 'toFind' a été trouvé. Cependant, une recherche classique en parcourant tout le livre prendrait trop de temps. Pour optimiser notre recherche, nous allons indexer tous les mots de notre livre.

Les données :

Pour cela, il faut créer un tableau associatif indexé en Texte dont chaque case contient un tableau de positions. Le tableau se définit comme suit :

*Tableau index en Tableau en Entier simple, indexé en Texte*

Et en Java comme suit :

HashMap<String, ArrayList<Integer>> index=new HashMap<>()

Pour obtenir les positions liées à un mot, il faudra utiliser notre tableau associatif comme suit :

*index(« le mot à rechercher »)* → renvoit un tableau de positions codées par des entiers

L'objectif de cet exercice est de remplir ce tableau associatif 'index'. On dit alors que l'on souhaite indexer notre livre.

**L'algorithme d'indexation :**

pour chaque mot (séparés par le caractère espace) contenu dans le livre

1. récupérer le tableau de positions associé à ce mot : index(« le mot »). Le tableau est automatiquement créé si le mot concerné n'a pas encore été indexé
2. ajouter la position de la nouvelle occurrence du mot dans le tableau de positions récupéré

**Les fonctions :**

a) Écrivez la fonction qui permet d'indexer notre livre en fonction du patron suivant :

public HashMap<String, ArrayList<Integer>> generateIndex(String book)

b) Écrire ensuite la procédure qui affiche toutes les occurrences (positions) d'un mot présentes dans notre livre :

public void screenPositions(String toFind, HashMap<String, ArrayList<Integer>> index)

c) Écrivez ensuite la fonction qui permet de déterminer si un mot est présent au moins une fois dans notre livre

public boolean isPresent(String toFind, HashMap<String, ArrayList<Integer>> index)

## Exercice 4 – Optimisation par colonies de fourmis

Réalisez une application qui calcule le plus court chemin pour résoudre le problème du voyageur de commerce. Aidez-vous de cet article : http://khayyam.developpez.com/articles/algo/voyageur-de-commerce/colonies-de-fourmis/