CODES DEVELOPPEMENT EFFICACE : DOCUMENTES

SIMPLE LIST

Cell.java

```java
public class Cell {

    public int value;
    public Cell next;

    public Cell(int value) {
        this.value = value;  // Assigne la valeur donnée à la cellule.
        next = null;         // La cellule ne pointe pas vers une autre cellule, elle est donc isolée (null).
    }
}
```

ListSimple.java

```java
public class ListSimple {

    public Cell head;   // Tête de la liste, qui pointe vers la première cellule.
    public int size;    // Taille de la liste, c'est-à-dire le nombre de cellules dans la liste.

    public ListSimple() {
        head = null;
        size = 0;
    }

    public Cell find(int value) {
        Cell c = head;  // On commence à la tête de la liste.
        // On parcourt la liste jusqu'à ce qu'on trouve la cellule contenant la valeur ou qu'on atteigne la fi
        while ((c != null) && (c.value != value)) {
            c = c.next;  // Avance à la cellule suivante.
        }
        return c;  // Retourne la cellule trouvée ou {@code null} si non trouvée.
    }

    public int findId(int value) {
        Cell c = head;  // On commence à la tête de la liste.
        // On parcourt la liste jusqu'à ce qu'on trouve la cellule contenant la valeur ou qu'on atteigne la fi
        int i = 0;
        while ((c != null) && (c.value != value)) {
            c = c.next;  // Avance à la cellule suivante.
            i++;
        }
        return (c == null) ? -1 : i;  // Retourne -1 si l'élément n'est pas trouvé.
    }

    public Cell get(int index) {
        if ((index < 0) || (index >= size)) return null;  // Vérifie que l'index est valide.
        Cell c = head;  // On commence à la tête de la liste.
        int i = 0;
        // On parcourt la liste jusqu'à atteindre l'index spécifié.
        while ((c != null) && (i < index)) {
            c = c.next;  // Avance à la cellule suivante.
            i += 1;      // Incrémente l'index.
        }
        return c;  // Retourne la cellule trouvée ou null si l'index est invalide.
    }
```

```java
public Cell append(int value) {
    Cell c = null;  // Variable temporaire pour la cellule existante.
    Cell newCell = new Cell(value);  // Crée la nouvelle cellule avec la valeur donnée.

    // Si la liste est vide (taille == 0), la nouvelle cellule devient la tête.
    if (size == 0) {
        head = newCell;
    } else {
        // Si la liste n'est pas vide, on obtient la dernière cellule (à l'index size-1).
        c = get(size - 1);
        c.next = newCell;  // La dernière cellule pointe maintenant vers la nouvelle cellule.
    }

    size++;  // Incrémente la taille de la liste.
    return newCell;  // Retourne la nouvelle cellule ajoutée.
}

public Cell insert(int value, int index) {
    Cell newCell = new Cell(value);  // Crée la nouvelle cellule avec la valeur donnée.

    // Si l'index est 0, la nouvelle cellule devient la première de la liste.
    if (index <= 0) {
        newCell.next = head;  // La nouvelle cellule pointe vers l'ancienne tête.
        head = newCell;       // La tête devient la nouvelle cellule.
    }
    // Si l'index est supérieur ou égal à la taille, insérer à la fin
    else if (index >= size) {
        return append(value);
    }
    // Si l'index est supérieur à 0, on trouve la cellule juste avant celle où on veut insérer.
    else {
        Cell c = get(index - 1);
        newCell.next = c.next;  // La nouvelle cellule pointe vers la cellule qui suit celle à l'index.
        c.next = newCell;       // La cellule précédente pointe vers la nouvelle cellule.
    }

    size++;  // Incrémente la taille de la liste.
    return newCell;  // Retourne la nouvelle cellule insérée.
}

public Cell replace(int value, int index) {
    if ((index < 0) || (index >= size)) return null;  // Vérifie que l'index est valide.

    Cell c = get(index);  // Récupère la cellule à l'index spécifié.
    c.value = value;      // Remplace la valeur de la cellule.

    return c;  // Retourne la cellule modifiée.
}

public Cell removeAt(int index) {
    if ((index < 0) || (index >= size)) return null;  // Vérifie que l'index est valide.

    // Si l'index est 0, on supprime la première cellule.
    if (index == 0) {
        Cell c = head;       // La cellule à supprimer.
        head = head.next;    // La tête de la liste devient la cellule suivante.
        size--;              // Réduit la taille de la liste.
        return c;            // Retourne la cellule supprimée.
    }
```

```
        // Si l'index est supérieur à 0, on trouve la cellule précédente.
        Cell c = get(index - 1);  // Récupère la cellule avant celle à supprimer.
        Cell delete = c.next;     // La cellule à supprimer est après la cellule précédente.
        c.next = delete.next;     // La cellule précédente va maintenant pointer sur la cellule après la suppr
        size--;                   // Réduit la taille de la liste.

        return delete;            // Retourne la cellule supprimée.
    }

    public Cell removeDeprecated(int value) {
        int index = findId(value);  // Recherche de l'index de la cellule avec la valeur donnée.
        // Vérifie si l'élément n'a pas été trouvé.
        if (index == -1) return null;   // Si la valeur n'est pas trouvée, on retourne null.
        else return removeAt(index);    // Si trouvé, on supprime la cellule à l'index trouvé.
    }

    public Cell remove(int value) {
        if (head == null) return null;

        Cell previous = null; // Cellule précédente.
        Cell current = head; // Cellule courante.

        // Recherche de la cellule contenant la valeur.
        while ((current != null) && (current.value != value)) {
            previous = current; // Previous devient current.
            current = current.next; // Avance dans la liste.
        }

        // Si la valeur a été trouvée (current n'est pas null).
        if (current != null) {
            // Si c'est la tête de la liste.
            if (current == head) {
                head = current.next; // La tête devient l'élément suivant.
                // Sinon, relie la cellule précédente à la suivante.
            } else {
                previous.next = current.next; // La cellule avant pointe après celle supprimée.
            }
            size--;
            return current;
        }

        return null;
    }

    public void print() {
        Cell c = head; // Commence à la tête de liste.

        if (head == null) {
            System.out.println("La liste est vide.");
            return;
        }
        while (c != null) {
            System.out.print(c.value + "->");
            c = c.next;
        }
        System.out.println("null");
    }
}
```

TestSimple.java

```java
public class TestSimple {

    public static void main(String[] args) {

        // Création d'une nouvelle liste chaînée simple
        ListSimple liste = new ListSimple();

        // Test de l'ajout en fin de liste
        liste.append(20);  // Ajoute la valeur 20 (la liste est vide, donc insertion en tête)

        // Test de l'insertion en tête (index négatif)
        liste.insert(10, -5);  // Insère la valeur 10 à l'index -5, équivalent à une insertion en tête

        // Test de l'insertion en fin (index supérieur à la taille)
        liste.insert(30, 7);  // Insère la valeur 30 à l'index 7 (en dehors des limites), donc ajout à la fin

        // Test de l'ajout en fin
        liste.append(50);  // Ajoute la valeur 50 à la fin de la liste

        // Test de l'insertion au milieu de la liste
        liste.insert(40, 3);  // Insère la valeur 40 à l'index 3

        // Affichage de la liste après les ajouts et insertions
        liste.print();  // Doit afficher : 10 -> 20 -> 30 -> 40 -> 50 -> null

        // Test de suppression d'une valeur inexistante
        liste.remove(12);  // Ne doit rien faire car la valeur 12 n'est pas présente

        // Test de suppression de la tête
        liste.remove(10);  // Enlève la valeur 10 (qui est la tête de la liste)

        // Test de suppression à un index invalide (négatif)
        liste.removeAt(-2);  // Ne doit rien faire car l'index est négatif

        // Test de suppression à un index invalide (hors des limites)
        liste.removeAt(22);  // Ne doit rien faire car l'index est supérieur à la taille

        // Test de suppression d'un élément à un index valide
        liste.removeAt(2);  // Enlève la valeur à l'index 2 (i.e., la valeur 40)

        // Affichage de la liste après les suppressions
        liste.print();  // Doit afficher : 20 -> 30 -> 50 -> null

        // Test d'accès à une cellule avec un index invalide (négatif)
        Cell c = liste.get(-1);  // Devrait renvoyer null car l'index est invalide
        if (c != null) {
            System.out.println("Problème avec la méthode get() pour un index négatif");
        }

        // Test d'accès à une cellule avec un index invalide (hors des limites)
        c = liste.get(99);  // Devrait renvoyer null car l'index est hors des limites
        if (c != null) {
            System.out.println("Problème avec la méthode get() pour un index hors des limites");
        }

        // Test de recherche d'une valeur inexistante
        c = liste.find(99);  // Devrait renvoyer null car la valeur 99 n'est pas présente dans la liste
        if (c != null) {
```

```
            System.out.println("Problème avec la méthode find() pour une valeur inexistante");
        }

        // Test de recherche d'une valeur existante (20)
        c = liste.find(20);
        if (c == null || c.value != 20) {
            System.out.println("Problème avec la méthode find() pour la valeur 20");
        }

        // Test de recherche d'une valeur existante (50)
        c = liste.find(50);
        if (c == null || c.value != 50) {
            System.out.println("Problème avec la méthode find() pour la valeur 50");
        }
    }
}

// ----------------------------
// EXECUTION
// ----------------------------
//PS C:\Users\Gamer\Desktop\devEfficace\SimpleList\src> javac *.java
//PS C:\Users\Gamer\Desktop\devEfficace\SimpleList\src> java TestSimple
//10->20->30->40->50->null
//        20->30->50->null
```

LISTE DOUBLEMENT CHAINEE CIRCULAIRE

CellDouble.java

```
public class CellDouble {

    public int value;

    public CellDouble prev;

    public CellDouble next;

    public CellDouble(int value) {
        this.value = value;
        this.prev = null;
        this.next = null;
    }
}
```

ListDoubleCirc.java

```
public class ListDoubleCirc {

    public CellDouble head;

    public int size;

    public ListDoubleCirc() {
        head = null;
        size = 0;
    }

    public CellDouble find(int value) {
        // Si la liste est vide, retourner null.
        if (head == null) return null;
```

```
        // Commencer à la tête de la liste.
        CellDouble current = head;

        // Parcourir la liste circulairement.
        do {
            // Si la valeur correspond, retourner la cellule.
            if (current.value == value) {
                return current;
            }
            current = current.next; // Passer à la cellule suivante.
        } while (current != head); // Revenir à la tête si fin de liste.

        // Retourner null si non trouvé.
        return null;
    }

    public CellDouble get(int index) {
        // Vérifie si l'index est valide
        if ((index < 0) || (index >= size)) return null;

        CellDouble current;
        int i;

        // Si l'index est dans la première moitié
        if (index < size / 2) {
            current = head;  // Commence au début
            i = 0;
            // Traverse jusqu'à l'index
            while (i < index) {
                current = current.next;  // Avance d'une cellule
                i++;
            }
        } else {
            // Si l'index est dans la deuxième moitié
            current = head.prev;  // Commence à la fin
            i = size - 1;
            // Traverse en sens inverse jusqu'à l'index
            while (i > index) {
                current = current.prev;  // Recule d'une cellule
                i--;
            }
        }
        return current;  // Retourne la cellule trouvée
    }

    public CellDouble append(int value) {
        if (head == null) {
            head = new CellDouble(value);
            head.next = head;  // La tête pointe vers elle-même.
            head.prev = head;  // La tête pointe vers elle-même.
            size++;
            return head;
        }

        // Si la liste n'est pas vide, on ajoute un élément à la fin.
        CellDouble append = new CellDouble(value);
        CellDouble last = head.prev;    // Dernière cellule de la liste.

//        last <-> append
```

```
        last.next = append;            // La cellule précédente pointe vers le nouvel élément.
        append.prev = last;            // Le nouvel élément pointe vers la dernière cellule.
        append.next = head;            // Le nouvel élément pointe vers la tête (circularité).
        head.prev = append;            // La tête pointe vers le nouvel élément comme dernier élément.

        size++;
        return append;
    }

    public CellDouble prepend(int value) {
        if (head == null) {
            head = new CellDouble(value);
            head.next = head;  // La tête pointe vers elle-même.
            head.prev = head;  // La tête pointe vers elle-même.
            size++;
            return head;
        }

        // Ajouter un élément avant la tête existante.
        CellDouble prepend = new CellDouble(value);
        CellDouble first = head;
        CellDouble last = first.prev;

//        prev       head       next
//        last <-> prepend <-> first

        // Mise à jour des pointeurs de la nouvelle cellule.
        prepend.prev = last;    // La nouvelle cellule pointe vers l'ancienne dernière cellule.
        prepend.next = first;   // La nouvelle cellule pointe vers la tête.

        // Mise à jour des pointeurs de l'ancienne dernière cellule et de l'ancienne tête.
        last.next = prepend;    // L'ancienne dernière cellule pointe vers la nouvelle cellule.
        first.prev = prepend;   // L'ancienne tête pointe maintenant vers la nouvelle cellule comme précédent.

        // La tête devient la nouvelle cellule.
        head = prepend;         // La nouvelle cellule devient la tête de la liste.

        size++;
        return prepend;
    }

    public CellDouble insert(int value, int index) {
        // Si l'index est négatif, insère au début de la liste.
        if (index < 0) return prepend(value);

        // Si l'index est supérieur ou égal à la taille, insère à la fin de la liste.
        if (index >= size) return append(value);

        // Si la liste est vide, insère le premier élément.
        if (head == null) {
            head = new CellDouble(value);
            head.next = head;  // La tête pointe vers elle-même (circularité).
            head.prev = head;  // La tête pointe vers elle-même (circularité).
            size++;
            return head;
        }

        // Parcours pour insérer au bon emplacement dans une liste non vide.
        CellDouble current = get(index);
```

```
        // Création de la nouvelle cellule à insérer.
        CellDouble newCell = new CellDouble(value);

//      (prev) previousCell (next) <-> (prev) newCell (next) <-> (prev) current (next)

        // Mise à jour des pointeurs de la nouvelle cellule.
        newCell.prev = current.prev;  // La nouvelle cellule pointe vers la cellule précédente.
        newCell.next = current;       // La nouvelle cellule pointe vers la cellule actuelle.

        // Mise à jour des pointeurs des cellules voisines.
        CellDouble previousCell = current.prev; // La cellule précédente de l'élément actuel
        previousCell.next = newCell;  // La cellule précédente pointe maintenant vers la nouvelle cellule
        current.prev = newCell;       // La cellule actuelle pointe vers la nouvelle cellule comme précédente.

        // Si l'insertion est à l'index 0, mettre à jour la tête.
        if (index == 0) {
            head = newCell;  // La nouvelle cellule devient la tête.
        }

        size++;

        // Retourne la nouvelle cellule insérée.
        return newCell;
    }

    public CellDouble replace(int value, int index) {
        // Vérifie si l'index est valide. Si l'index est hors limites, retourne null.
        if (index < 0 || index > size) return null;

        // Récupère la cellule à l'index spécifié.
        CellDouble replace = get(index);

        // Remplace la valeur de la cellule.
        replace.value = value;

        // Retourne la cellule modifiée.
        return replace;
    }

    public CellDouble removeAt(int index) {
        // Vérifie si l'index est valide. Si l'index est en dehors des limites, retourne null.
        if (index < 0 || index >= size) return null;

        // Récupère la cellule à supprimer à l'index spécifié.
        CellDouble toRemove = get(index);
        CellDouble previous = toRemove.prev;  // Cellule avant celle à supprimer.
        CellDouble next = toRemove.next;      // Cellule après celle à supprimer.

//        previous (next) <-> (prev) toRemove (next) <-> (prev) next

        // Mise à jour des pointeurs pour les cellules voisines :
        previous.next = next;   // La cellule précédente pointe vers la cellule suivante.
        next.prev = previous;   // La cellule suivante pointe vers la cellule précédente.

        // Si la cellule à supprimer est la tête, il faut mettre à jour la tête.
        if (toRemove == head) {
            head = next;  // La tête devient la cellule suivante.
        }

        // La taille de la liste est réduite après la suppression de l'élément.
```

```
        size--;

        // Retourne la cellule supprimée pour éventuellement utiliser sa valeur ou d'autres informations.
        return toRemove;
    }

    public CellDouble remove(int value) {
        // Recherche de la cellule contenant la valeur à supprimer.
        CellDouble toRemove = find(value);

        // Si la cellule n'est pas trouvée, on retourne null.
        if (toRemove == null) return null;

        // On appelle removeAt() pour supprimer cette cellule, mais on doit d'abord connaître son index.
        int index = 0;
        CellDouble current = head;

        // Parcours de la liste pour trouver l'index de la cellule à supprimer.
        while (current != toRemove) {
            current = current.next;
            index++;  // Incrémentation de l'index à chaque itération.
        }

        // Appel de la méthode removeAt pour effectuer la suppression à l'index trouvé.
        return removeAt(index);
    }

    public void print() {
        if (head == null) {
            System.out.println("La liste est vide");
            return;
        }

        CellDouble current = head;
        do {
            System.out.print(current.value + " ");
            current = current.next;
        } while (current != head);
        System.out.println();
    }

}
```

TestDouble.java

```
public class TestDouble {

    public static void main(String[] args) {

        ListDoubleCirc liste = new ListDoubleCirc();

        // Tests d'insertion
        liste.append(20);      // Insertion en fin (liste vide -> insertion en tête)
        liste.insert(10, -5);  // Insertion en tête (index négatif)
        liste.insert(30, 7);   // Insertion en fin (index hors limites)
        liste.append(50);      // Ajout en fin
        liste.insert(40, 3);   // Insertion à l'index 3
        liste.print();         // Affiche la liste : 10 20 30 40 50

        // Tests de suppression
```

```
        liste.remove(12);       // Valeur non existante, ne fait rien
        liste.remove(10);       // Suppression de la valeur 10 (tête de liste)
        liste.removeAt(-2);     // Index négatif, ne fait rien
        liste.removeAt(22);     // Index hors limites, ne fait rien
        liste.removeAt(2);      // Suppression à l'index 2 (i.e. valeur 40)
        liste.print();          // Affiche la liste : 20 30 50

        // Tests d'accès par index
        CellDouble c = liste.get(-1); // Accès hors limites (index négatif) -> renvoie null
        if (c != null) {
            System.out.println("Problème avec get(-1)");
        }

        c = liste.get(99); // Accès hors limites (index trop grand) -> renvoie null
        if (c != null) {
            System.out.println("Problème avec get(99)");
        }

        c = liste.get(1); // Accès à l'index 1 (i.e. valeur 30)
        if (c == null || c.value != 30) {
            System.out.println("Problème avec get(1)");
        }

        // Tests de recherche par valeur
        c = liste.find(99); // Valeur inexistante -> renvoie null
        if (c != null) {
            System.out.println("Problème avec find(99)");
        }

        c = liste.find(20); // Recherche de la valeur 20
        if (c == null || c.value != 20) {
            System.out.println("Problème avec find(20)");
        }

        c = liste.find(50); // Recherche de la valeur 50
        if (c == null || c.value != 50) {
            System.out.println("Problème avec find(50)");
        }
    }
}

// ----------------------------
// EXECUTION
// ----------------------------
// PS C:\Users\Gamer\Desktop\devEfficace\LinkedList\src> java TestDouble
// 10 20 30 40 50
// 20 30 50
```

ARBRES

Node.java

```
import java.util.*;

class Node {

    public List<Node> children;

    public int value;

    public Node(int value) {
```

```
        this.value = value; // Attribue la valeur spécifiée au nœud.
        children = new ArrayList<Node>(); // Initialise une liste vide pour les enfants.
    }

    public Node addChild(int value) {
        Node newNode = new Node(value);
        children.add(newNode); // Ajoute le nouvel enfant à la liste des enfants.
        return newNode;
    }

    public void addChild(Node n) {
        if (n != null) {
            this.children.add(n); // Ajoute le nœud spécifié à la liste des enfants.
        }
    }

    public Node getChild(int index) {
        if (index < 0 || index >= this.children.size()) {
            return null; // Retourne null si l'index est hors limites.
        }
        return children.get(index); // Retourne l'enfant à l'index spécifié.
    }
}
```

Tree.java

```
import java.util.*;

class Tree {

    public Node root;

    public int nbNodes;

    public Tree() {
        root = null;
        nbNodes = 0;
    }

    public Node addNode(int value, Node parent) {
        // si parent est null.
        if (parent == null) {
            Node newNode = new Node(value); // noeud parent.
            // si root existe déja.
            if (root == null) {
                newNode.addChild(root); // ancien root devient fils de newNode.
            }
            root = newNode; // newNode devient le nœud root.
            nbNodes++;
            return newNode;
        } else if (contains(parent, root) == null) {
            Node newNode = parent.addChild(value);
            nbNodes++;
            return newNode;
        }
        return null; // parent introuvable.
    }

    public Node contains(Node toSearch, Node parent) {
        if (parent == null) {
```

```
        return null;
    }
    if (parent == toSearch) {
        return parent; // condition d'arrêt.
    }

    for (Node child : parent.children) {
        Node found = contains(toSearch, child); // appel récursif.
        if (found != null) {
            return found; // renvoie résultat.
        }
    }

    return null; // valeur non trouvée.
}

public Node searchValueByLevel(int value, Node parent) {
    if (parent == null) {
        return null;
    }

    Queue<Node> queue = new LinkedList<>();
    queue.add(parent);

    while (!queue.isEmpty()) {
        Node current = queue.poll(); // supprime et renvoie le nœud en tête de liste.
        if (current.value == value) {
            return current;
        }
        queue.addAll(current.children); // recherche depuis le fils.
    }

    return null; // valeur non trouvée.
}

public Node searchValueByDepth(int value, Node parent) {
    if (parent == null) return null;

    if (parent.value == value) {
        return parent; // condition d'arrêt.
    }

    for (Node child : parent.children) {
        Node found = searchValueByDepth(value, child); // appel récursif.
        if (found != null) {
            return found; // renvoie résultat.
        }
    }

    return null; // valeur non trouvée.
}

public Node searchValue(int value, int type) {
    Node n = null;
    if (type == 1) n = searchValueByDepth(value, root);
    else if (type == 2) n = searchValueByLevel(value, root);
    return n;
}

public void print() {
```

```
        if (root != null) {
            printNode(root,0);
        }
    }

    private void printNode(Node n,int level) {
        for (int i = 0; i < 2 * level; i++) {
            System.out.print(" ");
        }
        System.out.println(n.value);
        for (Node child : n.children) {
            printNode(child, level + 1);
        }
    }

    public void printLevel() {
        if (root == null) {
            System.out.println("L'arbre est vide.");
            return;
        }

        Queue<Node> queue = new LinkedList<>();
        Map<Node, Integer> levels = new HashMap<>(); // Associe chaque nœud à son niveau.
        queue.add(root);
        levels.put(root, 0);

        while (!queue.isEmpty()) {
            Node current = queue.poll();
            int level = levels.get(current);

            // Indentation pour refléter le niveau actuel.
            for (int i = 0; i < 2 * level; i++) {
                System.out.print(" ");
            }
            System.out.println(current.value);

            // Ajout des enfants à la queue avec leur niveau.
            for (Node child : current.children) {
                queue.add(child);
                levels.put(child, level + 1);
            }
        }
    }
}
```

TestTree.java

```
public class TestTree {

    public static void main(String[] args) {

        Tree tree = new Tree();

        Node root = tree.addNode(20, null); // création racine
        root = tree.addNode(10, null); // test remplacement racine
        Node n1 = tree.addNode(21, root);
        n1.addChild(30);
        Node n2 = n1.addChild(31);
        Node n3 = n1.addChild(32);
        n2.addChild(40);
```

```
        n2 = n2.addChild(41);
        n3.addChild(45);
        n3.addChild(46);
        Node n4 = n3.addChild(47);
        n4.addChild(50);
        n4 = n4.addChild(51);

        tree.print();
        tree.printLevel();

        Node s = tree.contains(n3, root);
        if (s != n3) {
            System.out.println("échec recherche du noeud contenant 32");
        }
        s = tree.contains(n2, root);
        if (s != n2) {
            System.out.println("échec recherche du noeud contenant 41");
        }
        s = tree.contains(n4, root);
        if (s != n4) {
            System.out.println("échec recherche du noeud contenant 51");
        }

        s = tree.searchValue(45, 1); // recherche en profondeur
        if (s.value != 45) {
            System.out.println("échec recherche valeur 45");
        }
        s = tree.searchValue(50, 2); // recherche en largeur
        if (s.value != 50) {
            System.out.println("échec recherche valeur 50");
        }
        s = tree.searchValue(60, 1); // recherche en profondeur
        if (s != null) { // si on trouve quelque chose = pas normal
            System.out.println("échec recherche valeur 60");
        }
    }
}

// ----------------------------
// EXECUTION
// ----------------------------
// PS C:\Users\Gamer\Desktop\devEfficace\Trees\src> javac *.java
// PS C:\Users\Gamer\Desktop\devEfficace\Trees\src> java TestTree
// 10
//   21
//     30
//     31
//       40
//       41
//     32
//       45
//       46
//       47
//         50
//         51

// 10
//   21
//     30
//     31
```

```
//     32
//       40
//       41
//       45
//       46
//       47
//         50
//         51
```

COLLECTIONS : WORDS

AppComp.java

```
import java.util.ArrayList;
public class AppComp {

    public static void main(String[] args) throws Exception {

        // Nombre d'éléments à sélectionner aléatoirement dans le fichier de mots
        int nbElements = 10000;
        // Crée une instance de la classe ListOfWords pour récupérer une liste de mots
        ListOfWords lWords = new ListOfWords();
        // Sélectionne un certain nombre de mots aléatoires dans la liste de mots
        ArrayList<String> l = lWords.randomSelect(nbElements);

        // Affiche les mots sélectionnés
//          for(String s : l){
//          System.out.println(s);
//          }

        // Démarre la mesure du temps d'exécution pour la recherche dans la liste
        long start = System.currentTimeMillis();
        ArrayList<String> lfound = lWords.find(l);
        long end = System.currentTimeMillis();
        long timeElapsed = end - start;

        // Affiche les résultats de la recherche
//          for(String s : lfound){
//              System.out.println(s);
//          }

        // Affiche le temps d'exécution pour la recherche dans la liste
        System.out.println("time with List " + timeElapsed + "ms");

        // Boucle qui effectue des tests sur trois types de Map : HashMap, TreeMap, LinkedHashMap
        for (int i = 0; i < 3; i++) {

            // Crée une instance de HashOfWordsComp en fonction du type de Map sélectionné
            HashOfWordsComp hWords = new HashOfWordsComp(i);

            // Affiche le type de Map en cours de test
            switch(i) {
                case 0:
                    System.out.println("HashMap");
                    break;
                case 1:
                    System.out.println("TreeMap");
                    break;
                case 2:
                    System.out.println("LinkedHashMap");
                    break;
```

```
            }

            // Mesure le temps d'exécution pour rechercher des mots dans les valeurs de la Map en utilisant un
            start = System.currentTimeMillis();
            lfound = hWords.findValuesList(l);
            end = System.currentTimeMillis();
            timeElapsed = end - start;

            // Affichage des résultats pour containsValue (désactivé ici)
            // for (String s : lfound) {
            //     System.out.println(s);
            // }
            System.out.println("time with HashMap values List " + timeElapsed + "ms");

            // Mesure le temps d'exécution pour rechercher des mots dans les valeurs converties en HashSet
            start = System.currentTimeMillis();
            lfound = hWords.findValuesToSet(l);
            end = System.currentTimeMillis();
            timeElapsed = end - start;

            // Affichage des résultats pour contains (désactivé ici)
            // for (String s : lfound) {
            //     System.out.println(s);
            // }
            System.out.println("time with HashMap values converted to Set " + timeElapsed + "ms");

            // Mesure le temps d'exécution pour rechercher des mots dans les clés de la Map
            start = System.currentTimeMillis();
            lfound = hWords.findKeys(l);
            end = System.currentTimeMillis();
            timeElapsed = end - start;

            // Affichage des résultats pour containsKey (désactivé ici)
            // for (String s : lfound) {
            //     System.out.println(s);
            // }

            System.out.println("time with HashMap keys " + timeElapsed + "ms");
        }
    }
}


// EXECUTION HashOfWords ITERATOR
// ------------------------------
// 100.000 mots
// ------------------------------
//time with List 5066ms
//HashMap
//time with HashMap values List 20364ms
//time with HashMap values converted to Set 42ms
//time with HashMap keys 27ms (second)
//        TreeMap
//time with HashMap values List 10384ms
//time with HashMap values converted to Set 15ms (first)
//time with HashMap keys 44ms
//        LinkedHashMap
//time with HashMap values List 4843ms
//time with HashMap values converted to Set 33ms (third)
//time with HashMap keys 50ms
```

```
// ----------------------------
// RESULTATS
// ----------------------------
//Les tests montrent que les Map sont plus performantes que l'ArrayList pour les grandes collections.
//
//HashMap : Très rapide, grâce à sa gestion optimisée des hachages.
//LinkedHashMap : Très performant, mais un léger surcoût dû à l'ordre d'insertion.
//TreeMap : Très performant, malgré son tri automatique des éléments.
//ArrayList : Moins efficace, surtout pour de grandes collections.
```

HashOfWordsComp.java

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.*;

public class HashOfWordsComp {
    private Map<Integer, String> map;

    public HashOfWordsComp(int mapType) throws IOException {
        // Initialisation de la Map selon le type choisi
        switch (mapType) {
            case 0:
                map = new HashMap<>(); // Organise en fonction d'insertion {key, value} (efficace avec une clé
                break;
            case 1:
                map = new TreeMap<>(); // Trier dans l'ordre croissant {key, value} (recherche)
                break;
            case 2:
                map = new LinkedHashMap<>(); // Organise en fonction d'insertion {key, value} (insertions)
                break;
            default:
                throw new IllegalArgumentException("Type de map invalide: " + mapType);
        }
        loadWords();
    }

    private void loadWords() throws IOException {
        try (BufferedReader br = new BufferedReader(new FileReader("french_words.txt"))) {
            String line;
            while ((line = br.readLine()) != null) {
                map.put(line.hashCode(), line);
            }
        }
    }

    public ArrayList<String> findValuesList(ArrayList<String> words) {
        // Liste retournée
        ArrayList<String> result = new ArrayList<>();

        // Initialiser l'iterator pour les mots à rechercher
        Iterator<String> wordIterator = words.iterator();

        // Lecture des mots un à un
        while (wordIterator.hasNext()) {
            // Récupère le premier mot et pointe maintenant sur le prochain mot
```

```
            String word = wordIterator.next();

            boolean found = map.containsValue(word);

            result.add(word + (found ? " YES" : " NO"));
        }

        // OU

//        for (String word :words) {
//            if(map.containsValue(word))
//                result.add(word+" YES");
//            else
//                result.add(word+" NO");
//        }

        return result;
    }

    public ArrayList<String> findValuesToSet(ArrayList<String> words) {
        // Liste retournée
        ArrayList<String> result = new ArrayList<>();
        // Initialiser la liste HashSet des mots à trouver
        HashSet<String> valueSet = new HashSet<>();

        // Utiliser un iterator pour ajouter les valeurs du HashMap dans le HashSet
        Iterator<Map.Entry<Integer, String>> iterator = map.entrySet().iterator();
        while (iterator.hasNext()) {
            // Récupérer la prochaine entrée (clé-valeur)
            Map.Entry<Integer, String> entry = iterator.next();
            // Ajouter la valeur de chaque entrée
            valueSet.add(entry.getValue());
        }

        // Initialiser l'iterator pour les mots à rechercher
        Iterator<String> wordIterator = words.iterator();
        // Parcours de la liste des mots à rechercher
        while (wordIterator.hasNext()) {
            String word = wordIterator.next();
            boolean found = valueSet.contains(word);
            result.add(word + (found ? " YES" : " NO"));
        }

        // OU

//        HashSet<String> valueSet = new HashSet<String>(map.values());
//
//        for (String word: words) {
//            if (valueSet.contains(word)) {
//                result.add(word+" YES");
//            } else {
//                result.add(word+" NO");
//            }
//        }

        return result;
    }

    public ArrayList<String> findKeys(ArrayList<String> words) {
        // Liste retournée
```

```
        ArrayList<String> result = new ArrayList<>();

        // Initialiser un iterator pour la liste des mots
        Iterator<String> wordIterator = words.iterator();

        // Parcours des mots un à un
        while (wordIterator.hasNext()) {
            String word = wordIterator.next();

            // Calculer la valeur hachée du mot
            int hashCode = word.hashCode();
            boolean found = map.containsKey(hashCode);

            result.add(word + (found ? " YES" : " NO"));
        }

        // OU

//        for (String word:words) {
//            if(map.containsKey(word.hashCode()))
//                result.add(word+" YES");
//            else
//                result.add(word+" NO");
//
//        }

        return result;
    }
}
```

ListOfWord.java

```
import java.io.*;
import java.util.*;

public class ListOfWords {
    private ArrayList<String> wordsList;

    public ListOfWords() {
        // Liste qui stock les mots de french_words.txt
        wordsList = new ArrayList<>();
        try {
            // Lecture du fichier contenant les mots français
            BufferedReader br = new BufferedReader(new FileReader("french_words.txt"));
            String line;
            while ((line = br.readLine()) != null) {
                wordsList.add(line.trim()); // Ajoute chaque mot dans la liste sans espace/saut de ligne/tab
            }
            br.close();
        } catch (IOException e) {
            System.err.println("Erreur lors de la lecture du fichier : " + e.getMessage());
        }
    }

    public ArrayList<String> randomSelect(int nbElements) {
        // Liste avec les mots a trouver dans la liste de french_words.txt
        ArrayList<String> selectedWords = new ArrayList<>();
        Random random = new Random();

        for (int i = 0; i < nbElements; i++) {
```

```
            // Sélectionne un index aléatoire parmis la liste de mots francais
            int randomIndex = random.nextInt(wordsList.size());
            // Ajout dans selectedWords un mot avec son index
            selectedWords.add(wordsList.get(randomIndex));
        }
        return selectedWords;
    }

    public ArrayList<String> find(ArrayList<String> inputWords) {
        ArrayList<String> foundWords = new ArrayList<>();
        // Each word in inputWords à found
        for (String word : inputWords) {
            // Utilisation de contains pour chercher au sein de la liste
            if (wordsList.contains(word)) {
                foundWords.add(word + " YES");
            } else {
                foundWords.add(word + " NO");
            }
        }
        return foundWords;
    }
}
```

ABRES BINAIRES : PARCOURS DE LISTES

BinaryNode

```
class BinaryNode {

    BinaryNode left;

    BinaryNode right;

    int value;

    public BinaryNode(int value) {
        this.value = value;
        left = null;
        right = null;
    }
}
```

BinaryTree.java

```
class BinaryTree {

    public BinaryNode root;

    public int nbBinaryNodes;

    public BinaryTree() {
        root = null;
    }

    public void addBinaryNode(int value) {
        root = insertRecur(root, value); // Utilisation de la méthode de récursion pour ajouter le nœud
    }

    private BinaryNode insertRecur(BinaryNode n, int value) {
        // Si l'arbre est vide, on crée un nouveau nœud.
        if (n == null) {
```

```
            return new BinaryNode(value);
        }

        // Si la valeur est inférieure ou égale à celle du nœud courant, on insère à gauche.
        if (value <= n.value) {
            n.left = insertRecur(n.left, value);
        }
        // Si la valeur est plus grande que celle du nœud courant, on insère à droite.
        else {
            n.right = insertRecur(n.right, value);
        }
        return n;
    }

    public BinaryNode search(int value) {
        BinaryNode n = root;
        while ((n != null) && (n.value != value)) {
            if (value <= n.value) {
                n = n.left;
            } else {
                n = n.right;
            }
        }
        return n; // retourne le nœud trouvé ou null si la valeur n'existe pas
    }

    public void print() {
        if (root != null) {
            printBinaryNode(root);
        }
    }

    private void printBinaryNode(BinaryNode n) {
        int height = height(n);
        for (int i = 1; i <= height; i++) {
            printLevel(n, i, height);
            System.out.println();
        }
    }

    private int height(BinaryNode node) {
        if (node == null) {
            return 0;
        }
        return 1 + Math.max(height(node.left), height(node.right));
    }

    private void printLevel(BinaryNode node, int level, int height) {
        if (node == null) {
            printSpaces(height - level);
            return;
        }

        if (level == 1) {
            System.out.print(node.value);
        } else {
            printLevel(node.left, level - 1, height);
            printLevel(node.right, level - 1, height);
        }
        if (level < height) {
```

```
            System.out.print(" ");
        }
    }

    private void printSpaces(int spaces) {
        for (int i = 0; i < 1; i++) {
            System.out.print(" ");
        }
    }
}
```

TestBinaryTree.java

```
public class TestBinaryTree {

    public static void main(String[] args) {

        // Créer un arbre binaire
        BinaryTree tree = new BinaryTree();

        // Ajouter des nœuds
        tree.addBinaryNode(20); // Racine
        tree.addBinaryNode(10);
        tree.addBinaryNode(30);
        tree.addBinaryNode(5);
        tree.addBinaryNode(15);
        tree.addBinaryNode(25);
        tree.addBinaryNode(35);
        tree.addBinaryNode(36);
        tree.addBinaryNode(37);
        tree.addBinaryNode(38);
        tree.addBinaryNode(34);


        // Afficher l'arbre binaire
        System.out.println("Affichage de l'arbre binaire :");
        tree.print();

        // Recherche de nœuds
        BinaryNode foundNode = tree.search(15); // Recherche de la valeur 15
        if (foundNode != null && foundNode.value == 15) {
            System.out.println("Recherche réussie pour la valeur 15.");
        } else {
            System.out.println("Échec recherche de la valeur 15.");
        }

        foundNode = tree.search(5); // Recherche de la valeur 5
        if (foundNode != null && foundNode.value == 5) {
            System.out.println("Recherche réussie pour la valeur 5.");
        } else {
            System.out.println("Échec recherche de la valeur 5.");
        }

        foundNode = tree.search(25); // Recherche de la valeur 25
        if (foundNode != null && foundNode.value == 25) {
            System.out.println("Recherche réussie pour la valeur 25.");
        } else {
            System.out.println("Échec recherche de la valeur 25.");
        }
```

```
        // Recherche d'une valeur non existante
        foundNode = tree.search(40); // Recherche d'une valeur inexistante
        if (foundNode == null) {
            System.out.println("Recherche échouée pour la valeur 40, comme attendu.");
        } else {
            System.out.println("Échec recherche de la valeur 40.");
        }
    }
}

// ----------------------------
// EXECUTION
// ----------------------------
// PS C:\Users\Gamer\Desktop\devEfficace\BinaryTrees\src> javac *.java
// PS C:\Users\Gamer\Desktop\devEfficace\BinaryTrees\src> java TestBinaryTree
// Affichage de l'arbre binaire :
// 20
// 10 30
// 5 15  25 35
//           34 36
//               37
//                38
// Recherche réussie pour la valeur 15.
// Recherche réussie pour la valeur 5.
// Recherche réussie pour la valeur 25.
// Recherche a échoué pour la valeur 40, comme attendu.
```