# Structures de Données

## 1. Tableaux (Arrays)

### Description :

Les tableaux sont des collections d'éléments de taille fixe, où chaque élément peut être accédé via un index. Ils permettent un accès rapide aux données, mais leur taille ne peut pas être modifiée après la création.

### Exemples d'utilisation :

- Stocker des listes d'éléments (par exemple, des scores de jeu).
- Manipuler des données dans des algorithmes (comme le tri).

### Exemples de Code :

**JavaScript :**

```javascript
const numbers = [1, 2, 3, 4, 5]; // Déclaration d'un tableau
console.log(numbers[0]); // Accès au premier élément : 1
numbers.push(6); // Ajout d'un élément à la fin
console.log(numbers.length); // Affiche la longueur du tableau : 6
```

**Java :**

```java
public class Main {
    public static void main(String[] args) {
        int[] numbers = {1, 2, 3, 4, 5}; // Déclaration d'un tableau
        System.out.println(numbers[0]); // Accès au premier élément : 1
        numbers[5] = 6; // Ajout d'un élément (erreur car la taille est fixe)
    }
}
```

---

## 2. Listes Chaînées (Linked Lists)

### Description :

Une liste chaînée est une collection d'éléments, où chaque élément (ou nœud) contient une valeur et un pointeur vers le nœud suivant. Contrairement aux tableaux, les listes chaînées peuvent changer de taille dynamiquement.

### Exemples d'utilisation :

- Manipuler des éléments lorsque la taille est inconnue à l'avance.
- Implémentation de structures plus complexes (comme des files ou des piles).

## Exemples de Code :

### JavaScript :

```
class Node {
    constructor(value) {
        this.value = value; // Valeur du nœud
        this.next = null; // Pointeur vers le prochain nœud
    }
}

class LinkedList {
    constructor() {
        this.head = null; // Tête de la liste
    }

    append(value) {
        const newNode = new Node(value);
        if (!this.head) {
            this.head = newNode; // Si la liste est vide, le nouveau nœud devient la
tête
        } else {
            let current = this.head;
            while (current.next) {
                current = current.next; // Parcours jusqu'à la fin de la liste
            }
            current.next = newNode; // Ajout du nouveau nœud à la fin
        }
    }

    printList() {
        let current = this.head;
        while (current) {
            console.log(current.value); // Affichage de la valeur
            current = current.next; // Passage au nœud suivant
        }
    }
}

// Utilisation
const list = new LinkedList();
list.append(1);
list.append(2);
list.printList(); // Affiche : 1, 2
```

### Java :

```java
class Node {
    int value;
    Node next; // Pointeur vers le prochain nœud

    Node(int value) {
        this.value = value;
        this.next = null;
    }
}

class LinkedList {
    Node head; // Tête de la liste

    // Ajoute un nœud à la fin de la liste
    public void append(int value) {
        Node newNode = new Node(value);
        if (head == null) {
            head = newNode; // Si la liste est vide, le nouveau nœud devient la tête
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next; // Parcours jusqu'à la fin de la liste
            }
            current.next = newNode; // Ajout du nouveau nœud à la fin
        }
    }

    // Affiche tous les nœuds de la liste
    public void printList() {
        Node current = head;
        while (current != null) {
            System.out.print(current.value + " "); // Affichage de la valeur
            current = current.next; // Passage au nœud suivant
        }
    }
}

// Utilisation
public class Main {
    public static void main(String[] args) {
        LinkedList list = new LinkedList();
        list.append(1);
        list.append(2);
        list.printList(); // Affiche : 1 2
    }
}
```

---

## 3. Piles (Stacks)

## Description :

Une pile est une collection d'éléments où l'ajout et la suppression se font selon le principe LIFO (Last In, First Out). Le dernier élément ajouté est le premier à être retiré.

## Exemples d'utilisation :

- Gestion des appels de fonctions (pile d'appels).
- Annuler des actions dans les applications.

## Exemples de Code :

**JavaScript :**

```
class Stack {
    constructor() {
        this.items = []; // Stocke les éléments de la pile
    }

    push(element) {
        this.items.push(element); // Ajoute un élément au sommet de la pile
    }

    pop() {
        return this.items.pop(); // Retire et renvoie le sommet de la pile
    }

    peek() {
        return this.items[this.items.length - 1]; // Montre le sommet sans le
retirer
    }

    isEmpty() {
        return this.items.length === 0; // Vérifie si la pile est vide
    }
}

// Utilisation
const stack = new Stack();
stack.push(1);
stack.push(2);
console.log(stack.peek()); // Affiche : 2
stack.pop();
console.log(stack.peek()); // Affiche : 1
```

**Java :**

```java
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        Stack<Integer> stack = new Stack<>(); // Crée une pile

        stack.push(1); // Ajoute un élément
        stack.push(2);
        System.out.println(stack.peek()); // Affiche : 2

        stack.pop(); // Retire le sommet de la pile
        System.out.println(stack.peek()); // Affiche : 1
    }
}
```

---

## 4. Files (Queues)

### Description :

Une file est une collection d'éléments où l'ajout et la suppression se font selon le principe FIFO (First In, First Out). Le premier élément ajouté est le premier à être retiré.

### Exemples d'utilisation :

- Gestion des tâches dans un système d'exploitation.
- Traitement des requêtes dans les serveurs web.

### Exemples de Code :

**JavaScript :**

```javascript
class Queue {
    constructor() {
        this.items = []; // Stocke les éléments de la file
    }

    enqueue(element) {
        this.items.push(element); // Ajoute un élément à la fin de la file
    }

    dequeue() {
        return this.items.shift(); // Retire le premier élément ajouté
    }

    peek() {
```

```
        return this.items[0]; // Montre le premier élément sans le retirer
    }

    isEmpty() {
        return this.items.length === 0; // Vérifie si la file est vide
    }
}

// Utilisation
const queue = new Queue();
queue.enqueue(1);
queue.enqueue(2);
console.log(queue.peek()); // Affiche : 1
queue.dequeue();
console.log(queue.peek()); // Affiche : 2
```

**Java :**

```
import java.util.LinkedList;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<>(); // Crée une file

        queue.offer(1); // Ajoute un élément
        queue.offer(2);
        System.out.println(queue.peek()); // Affiche : 1

        queue.poll(); // Retire le premier élément ajouté
        System.out.println(queue.peek()); // Affiche : 2
    }
}
```

---

## 5. Arbres (Trees)

### Description :

Un arbre est une structure de données hiérarchique composée de nœuds, où chaque nœud a une valeur et peut avoir plusieurs nœuds enfants. Un arbre binaire est un arbre où chaque nœud a au maximum deux enfants.

### Exemples d'utilisation :

- Modélisation de données hiérarchiques (comme les fichiers sur un disque).
- Algorithmes de recherche et de tri.

## Exemples de Code :

### JavaScript :

```
class TreeNode {
    constructor(value) {
        this.value = value;
        this.left = null; // Enfant gauche
        this.right = null; // Enfant droit
    }
}

class BinaryTree {
    constructor() {
        this.root = null; // Racine de l'arbre
    }

    // Ajoute un nœud à l'arbre
    insert(value) {
        const newNode = new TreeNode(value);
        if (!this.root) {
            this.root = newNode;
            return;
        }
        this.insertNode(this.root, newNode);
    }

    insertNode(node, newNode) {
        if (newNode.value < node.value) {
            if (!node.left) {
                node.left = newNode;
            } else {
                this.insertNode(node.left, newNode);
            }


 } else {
            if (!node.right) {
                node.right = newNode;
            } else {
                this.insertNode(node.right, newNode);
            }
        }
    }

    // Parcours en ordre (in-order)
    inOrder(node = this.root) {
        if (node) {
            this.inOrder(node.left); // Parcours à gauche
            console.log(node.value); // Affiche la valeur
            this.inOrder(node.right); // Parcours à droite
```

```
        }
    }
}

// Utilisation
const tree = new BinaryTree();
tree.insert(5);
tree.insert(3);
tree.insert(7);
tree.inOrder(); // Affiche : 3, 5, 7
```

**Java :**

```
class TreeNode {
    int value;
    TreeNode left, right; // Enfants gauche et droit

    TreeNode(int value) {
        this.value = value;
        this.left = null;
        this.right = null;
    }
}

class BinaryTree {
    TreeNode root; // Racine de l'arbre

    // Ajoute un nœud à l'arbre
    public void insert(int value) {
        TreeNode newNode = new TreeNode(value);
        if (root == null) {
            root = newNode;
            return;
        }
        insertNode(root, newNode);
    }

    private void insertNode(TreeNode node, TreeNode newNode) {
        if (newNode.value < node.value) {
            if (node.left == null) {
                node.left = newNode;
            } else {
                insertNode(node.left, newNode);
            }
        } else {
            if (node.right == null) {
                node.right = newNode;
            } else {
                insertNode(node.right, newNode);
            }
```

```
        }
    }

    // Parcours en ordre (in-order)
    public void inOrder(TreeNode node) {
        if (node != null) {
            inOrder(node.left); // Parcours à gauche
            System.out.print(node.value + " "); // Affiche la valeur
            inOrder(node.right); // Parcours à droite
        }
    }
}

// Utilisation
public class Main {
    public static void main(String[] args) {
        BinaryTree tree = new BinaryTree();
        tree.insert(5);
        tree.insert(3);
        tree.insert(7);
        tree.inOrder(tree.root); // Affiche : 3 5 7
    }
}
```

---

## Concepts Clés

- **Complexité Spatiale et Temporelle :** Chaque structure de données a une complexité en termes de temps (comment la performance évolue avec la taille des données) et d'espace (la mémoire utilisée). Par exemple, les tableaux offrent un accès rapide aux éléments par index, mais leur taille est fixe, tandis que les listes chaînées offrent une taille dynamique mais un accès plus lent.
- **Choix de la Structure :** Le choix de la structure de données dépend de l'utilisation prévue. Par exemple, si tu as besoin d'un accès rapide aux éléments, un tableau peut être préférable. Si tu prévois beaucoup d'ajouts et de suppressions, une liste chaînée peut être plus efficace.

---

Cette fiche te donne un bon aperçu des structures de données essentielles que tu dois connaître.