Structures de données avancées en Java

Les structures de données sont la base des algorithmes efficaces. Explorons quatre structures puissantes : Skip List, Union-Find, AVL Tree et Binary Indexed Tree. Elles sont largement utilisées dans des scénarios nécessitant des recherches rapides, des unions, un équilibrage ou des requêtes de plage.

1. Skip List : Recherche Probabiliste

Une skip list est une liste liée en couches qui permet des recherches, insertions et suppressions rapides avec une complexité moyenne en temps de O(log n), offrant une alternative aux arbres équilibrés.

Implémentation en Java

```
import java.util.Random;

public class SkipList {
    static class Node {
        int value;
        Node[] next;
        Node(int value, int level) {
            this.value = value;
            this.next = new Node[level + 1];
        }
    }

    private Node head;
    private int maxLevel;
    private Random rand;
    private int level;

    SkipList() {
        maxLevel = 16;
        head = new Node(-1, maxLevel);
        rand = new Random();
        level = 0;
    }

    private int randomLevel() {
        int lvl = 0;
        while (rand.nextBoolean() && lvl < maxLevel) lvl++;
```

```
    return lvl;
}

void insert(int value) {
    Node[] update = new Node[maxLevel + 1];
    Node current = head;
    for (int i = level; i >= 0; i--) {
        while (current.next[i] != null && current.next[i].value < value) current = current.next[i];
        update[i] = current;
    }
    current = current.next[0];
    int newLevel = randomLevel();
    if (newLevel > level) {
        for (int i = level + 1; i <= newLevel; i++) update[i] = head;
        level = newLevel;
    }
    Node newNode = new Node(value, newLevel);
    for (int i = 0; i <= newLevel; i++) {
        newNode.next[i] = update[i].next[i];
        update[i].next[i] = newNode;
    }
}

boolean search(int value) {
    Node current = head;
    for (int i = level; i >= 0; i--) {
        while (current.next[i] != null && current.next[i].value < value) current = current.next[i];
    }
    current = current.next[0];
    return current != null && current.value == value;
}

public static void main(String[] args) {
    SkipList sl = new SkipList();
    sl.insert(3);
    sl.insert(6);
    sl.insert(7);
    System.out.println("Recherche 6: " + sl.search(6));
    System.out.println("Recherche 5: " + sl.search(5));
}
```

```
}
```

Sortie :

```
Recherche 6: true
Recherche 5: false
```

2. Union-Find (Ensemble Disjoint) : Suivi de Connectivité

Union-Find gère efficacement les ensembles disjoints, prenant en charge les opérations d'union et de recherche en temps amorti presque O(1) avec la compression de chemin et les heuristiques de rang.

Implémentation en Java

```java
public class UnionFind {
    private int[] parent, rank;

    UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) parent[i] = i;
    }

    int find(int x) {
        if (parent[x] != x) parent[x] = find(parent[x]);
        return parent[x];
    }

    void union(int x, int y) {
        int rootX = find(x), rootY = find(y);
        if (rootX != rootY) {
            if (rank[rootX] < rank[rootY]) parent[rootX] = rootY;
            else if (rank[rootX] > rank[rootY]) parent[rootY] = rootX;
            else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
```

```
    public static void main(String[] args) {
        UnionFind uf = new UnionFind(5);
        uf.union(0, 1);
        uf.union(2, 3);
        uf.union(1, 4);
        System.out.println("0 et 4 connectés: " + (uf.find(0) == uf.find(4)));
        System.out.println("2 et 4 connectés: " + (uf.find(2) == uf.find(4)));
    }
}
```

Sortie :

```
0 et 4 connectés: true
2 et 4 connectés: false
```

3. AVL Tree : Arbre BST Auto-équilibré

Un arbre AVL est un arbre binaire de recherche auto-équilibré où la différence de hauteur entre les sous-arbres (facteur d'équilibre) est au plus 1, garantissant des opérations en O(log n).

Implémentation en Java

```
public class AVLTree {
    static class Node {
        int key, height;
        Node left, right;
        Node(int key) {
            this.key = key;
            this.height = 1;
        }
    }

    private Node root;

    int height(Node node) { return node == null ? 0 : node.height; }
    int balanceFactor(Node node) { return node == null ? 0 : height(node.left) - height(node.right); }

    Node rightRotate(Node y) {
        Node x = y.left, T2 = x.right;
        x.right = y;
```

```
    y.left = T2;
    y.height = Math.max(height(y.left), height(y.right)) + 1;
    x.height = Math.max(height(x.left), height(x.right)) + 1;
    return x;
}

Node leftRotate(Node x) {
    Node y = x.right, T2 = y.left;
    y.left = x;
    x.right = T2;
    x.height = Math.max(height(x.left), height(x.right)) + 1;
    y.height = Math.max(height(y.left), height(y.right)) + 1;
    return y;
}

Node insert(Node node, int key) {
    if (node == null) return new Node(key);
    if (key < node.key) node.left = insert(node.left, key);
    else if (key > node.key) node.right = insert(node.right, key);
    else return node;

    node.height = Math.max(height(node.left), height(node.right)) + 1;
    int balance = balanceFactor(node);

    if (balance > 1 && key < node.left.key) return rightRotate(node);
    if (balance < -1 && key > node.right.key) return leftRotate(node);
    if (balance > 1 && key > node.left.key) {
        node.left = leftRotate(node.left);
        return rightRotate(node);
    }
    if (balance < -1 && key < node.right.key) {
        node.right = rightRotate(node.right);
        return leftRotate(node);
    }
    return node;
}

void insert(int key) { root = insert(root, key); }

void preOrder(Node node) {
```

```
        if (node != null) {
            System.out.print(node.key + " ");
            preOrder(node.left);
            preOrder(node.right);
        }
    }

    public static void main(String[] args) {
        AVLTree tree = new AVLTree();
        tree.insert(10);
        tree.insert(20);
        tree.insert(30);
        tree.insert(40);
        tree.insert(50);
        tree.insert(25);
        System.out.print("Préfixe: ");
        tree.preOrder(tree.root);
    }
}
```

Sortie : `Préfixe: 30 20 10 25 40 50`

4. Binary Indexed Tree (Fenwick Tree) : Requêtes de Plage

Un Binary Indexed Tree (BIT) gère efficacement les requêtes de somme de plage et les mises à jour en O(log n), souvent utilisé en programmation compétitive.

Implémentation en Java

```
public class BinaryIndexedTree {
    private int[] bit;
    private int n;

    BinaryIndexedTree(int[] arr) {
        n = arr.length;
        bit = new int[n + 1];
        for (int i = 0; i < n; i++) update(i, arr[i]);
    }

    void update(int index, int val) {
```

```
        index++;
        while (index <= n) {
            bit[index] += val;
            index += index & (-index);
        }
    }

    int getSum(int index) {
        int sum = 0;
        index++;
        while (index > 0) {
            sum += bit[index];
            index -= index & (-index);
        }
        return sum;
    }

    int rangeSum(int l, int r) { return getSum(r) - getSum(l - 1); }

    public static void main(String[] args) {
        int[] arr = {2, 1, 1, 3, 2, 3, 4, 5};
        BinaryIndexedTree bit = new BinaryIndexedTree(arr);
        System.out.println("Somme de 0 à 5: " + bit.getSum(5));
        System.out.println("Somme de plage 2 à 5: " + bit.rangeSum(2, 5));
        bit.update(3, 6); // Ajouter 6 à l'index 3
        System.out.println("Nouvelle somme de plage 2 à 5: " + bit.rangeSum(2, 5));
    }
}
```

Sortie :

```
Somme de 0 à 5: 12
Somme de plage 2 à 5: 9
Nouvelle somme de plage 2 à 5: 15
```

Blog 7 : Algorithmes de Recherche et de Simulation en Java

Les algorithmes de recherche et de simulation traitent les problèmes de recherche de chemin et probabilistes. Explorons la recherche A* et la simulation de Monte Carlo.

1. Recherche A* : Recherche de Chemin Heuristique

A* est un algorithme de recherche informé qui utilise une heuristique pour trouver le chemin le plus court dans un graphe, combinant les forces de Dijkstra et de la recherche gloutonne. Il est largement utilisé dans les jeux et la navigation.

Implémentation en Java

```java
import java.util.*;

public class AStar {
    static class Node implements Comparable<Node> {
        int x, y, g, h, f;
        Node parent;
        Node(int x, int y) {
            this.x = x;
            this.y = y;
            this.g = 0;
            this.h = 0;
            this.f = 0;
        }
        public int compareTo(Node other) { return this.f - other.f; }
    }

    static int heuristic(int x1, int y1, int x2, int y2) {
        return Math.abs(x1 - x2) + Math.abs(y1 - y2); // Distance de Manhattan
    }

    static void aStarSearch(int[][] grid, int[] start, int[] goal) {
        int rows = grid.length, cols = grid[0].length;
        PriorityQueue<Node> open = new PriorityQueue<>();
        boolean[][] closed = new boolean[rows][cols];
        Node startNode = new Node(start[0], start[1]);
        Node goalNode = new Node(goal[0], goal[1]);
        startNode.h = heuristic(start[0], start[1], goal[0], goal[1]);
        startNode.f = startNode.h;
        open.add(startNode);

        int[][] dirs = {};          // Directions de mouvement
        while (!open.isEmpty()) {
```

```
            Node current = open.poll();
            if (current.x == goal[0] && current.y == goal[1]) {
                printPath(current);
                return;
            }
            closed[current.x][current.y] = true;
            for (int[] dir : dirs) {
                int newX = current.x + dir[0], newY = current.y + dir[1];
                if (newX >= 0 && newX < rows && newY >= 0 && newY < cols && grid[newX][newY] != 1 && !closed[n
                    Node neighbor = new Node(newX, newY);
                    neighbor.g = current.g + 1;
                    neighbor.h = heuristic(newX, newY, goal[0], goal[1]);
                    neighbor.f = neighbor.g + neighbor.h;
                    neighbor.parent = current;
                    open.add(neighbor);
                }
            }
        }
        System.out.println("Aucun chemin trouvé!");
    }

    static void printPath(Node node) {
        List<int[]> path = new ArrayList<>();
        while (node != null) {
            path.add(new int[]{node.x, node.y});
            node = node.parent;
        }
        Collections.reverse(path);
        System.out.println("Chemin:");
        for (int[] p : path) System.out.println("(" + p[0] + ", " + p[1] + ")");
    }

    public static void main(String[] args) {
        int[][] grid = {
            {0, 0, 0, 0},
            {0, 1, 1, 0},
            {0, 0, 0, 0}
        };
        int[] start = {0, 0}, goal = {2, 3};
        aStarSearch(grid, start, goal);
```

```
    }
}
```

Sortie :

```
Chemin:
(0, 0)
(1, 0)
(2, 0)
(2, 1)
(2, 2)
(2, 3)
```

2. Simulation de Monte Carlo : Estimation Probabiliste

Les méthodes de Monte Carlo utilisent l'échantillonnage aléatoire pour estimer les résultats, comme l'approximation de π en simulant des points dans un carré et un cercle.

Implémentation en Java

```java
import java.util.Random;

public class MonteCarlo {
    static double estimatePi(int points) {
        Random rand = new Random();
        int insideCircle = 0;
        for (int i = 0; i < points; i++) {
            double x = rand.nextDouble();
            double y = rand.nextDouble();
            if (x * x + y * y <= 1) insideCircle++; // À l'intérieur du cercle unité
        }
        return 4.0 * insideCircle / points; // Ratio * 4 approximates
    }

    public static void main(String[] args) {
        int points = 1000000;
        double pi = estimatePi(points);
        System.out.println("  estimé avec " + points + " points: " + pi);
        System.out.println("  réel: " + Math.PI);
    }
}
```

Sortie (varie en raison de l’aléatoire) :

```
estimé avec 1000000 points: 3.1418
réel: 3.141592653589793
```