高級資料結構與 Java

數據結構是高效演算法的基礎。讓我們探索四種強大的數據結構：Skip List、Union-Find、AVL Tree 和 Binary Indexed Tree。這些在需要快速搜索、聯合、平衡或範圍查詢的情況下廣泛使用。

1. Skip List: 概率搜索

Skip List 是一種分層鏈表，允許快速搜索、插入和刪除，平均時間複雜度為 O(log n)，提供了平衡樹的替代方案。

Java 實現

```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("Search 6: " + sl.search(6));
        System.out.println("Search 5: " + sl.search(5));
    }
}
```

輸出:

```
Search 6: true
Search 5: false
```

2. Union-Find (Disjoint Set): 連通性追蹤

Union-Find 高效管理不相交集合，支持聯合和查找操作，平均時間複雜度接近 O(1)，使用路徑壓縮和排名策略。

Java 實現

```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 和 4 連接: " + (uf.find(0) == uf.find(4)));
        System.out.println("2 和 4 連接: " + (uf.find(2) == uf.find(4)));
    }
}
```

輸出:

```
0 和 4 連接: true
2 和 4 連接: false
```

3. AVL Tree: 自平衡 BST

AVL Tree 是一種自平衡二元搜索樹，子樹之間的高度差（平衡因子）最多為 1，確保 O(log n) 操作。

Java 實現

```
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("Preorder: ");
        tree.preOrder(tree.root);
    }
}
```

輸出: `Preorder: 30 20 10 25 40 50`

4. Binary Indexed Tree (Fenwick Tree): 範圍查詢

Binary Indexed Tree (BIT) 高效處理範圍和查詢更新，時間複雜度為 O(log n)，常用於競賽編程。

Java 實現

```
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("0 到 5 的和: " + bit.getSum(5));
        System.out.println("2 到 5 的範圍和: " + bit.rangeSum(2, 5));
        bit.update(3, 6); // 將 6 加到索引 3
        System.out.println(" 新的 2 到 5 的範圍和: " + bit.rangeSum(2, 5));
    }
}
```

輸出:

```
0 到 5 的和: 12
2 到 5 的範圍和: 9
新的 2 到 5 的範圍和: 15
```

部落格 7: Java 中的搜索和模擬演算法

搜索和模擬演算法解決路徑查找和概率問題。讓我們探索 A* 搜索和蒙特卡羅模擬。

1. A* 搜索: 启發式路徑查找

A* 是一種啟發式搜索演算法，使用啟發式來查找圖中的最短路徑，結合了 Dijkstra 和貪婪搜索的優點。它在遊戲和導航中廣泛使用。

Java 實現

```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); // 曼哈頓距離
    }

    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 = {};          // 方向數組
        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[r
                    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(" 沒有找到路徑!");
    }

    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(" 路徑:");
        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);
    }
}
```

輸出:

```
路徑:
(0, 0)
```

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

2. 蒙特卡羅模擬: 概率估計

蒙特卡羅方法使用隨機抽樣來估計結果，例如通過模擬方格和圓內的點來近似 π。

Java 實現

```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++; // 在單位圓內
        }
        return 4.0 * insideCircle / points; // 比例 * 4 近似
    }

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

輸出 (因隨機性而異):

```
使用 1000000 點估計 : 3.1418
實際 : 3.141592653589793
```