

Fortgeschrittene Datenstrukturen in Java

Datenstrukturen sind die Grundlage effizienter Algorithmen. Lassen Sie uns vier leistungsstarke Datenstrukturen erkunden: Skip List, Union-Find, AVL-Baum und Binärer Indexierter Baum. Diese werden weit verbreitet in Szenarien verwendet, die schnelle Suchen, Vereinigungen, Ausgleich oder Bereichsabfragen erfordern.

1. Skip List: Probabilistische Suche

Eine Skip List ist eine geschichtete verknüpfte Liste, die schnelle Suche, Einfügen und Löschen mit einer durchschnittlichen Zeitkomplexität von $O(\log n)$ ermöglicht und eine Alternative zu ausgewogenen Bäumen bietet.

Java-Implementierung

```
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("Suche 6: " + sl.search(6));
}

```

```

        System.out.println("Suche 5: " + s1.search(5));
    }
}

```

Ausgabe:

Suche 6: true
Suche 5: false

2. Union-Find (Disjunkte Menge): Verbindungsverfolgung

Union-Find verwaltet disjunkte Mengen effizient und unterstützt Union- und Find-Operationen in nahezu $O(1)$ amortisierter Zeit mit Pfadkompression und Rangheuristik.

Java-Implementierung

```

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 und 4 verbunden: " + (uf.find(0) == uf.find(4)));
        System.out.println("2 und 4 verbunden: " + (uf.find(2) == uf.find(4)));
    }
}

```

Ausgabe:

```

0 und 4 verbunden: true
2 und 4 verbunden: false

```

3. AVL-Baum: Selbstausgleichender BST

Ein AVL-Baum ist ein selbstausgleichender binärer Suchbaum, bei dem die Höhenunterschied zwischen Unterbäumen (Balancefaktor) höchstens 1 beträgt, was $O(\log n)$ Operationen gewährleistet.

Java-Implementierung

```

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("Vorordnung: ");
    tree.preOrder(tree.root);
}
}

```

Ausgabe: Vorordnung: 30 20 10 25 40 50

4. Binärer Indexierter Baum (Fenwick-Baum): Bereichsabfragen

Ein Binärer Indexierter Baum (BIT) verarbeitet Bereichssummenabfragen und -aktualisierungen in $O(\log n)$ Zeit, oft in Wettbewerbsprogrammierung verwendet.

Java-Implementierung

```

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("Summe von 0 bis 5: " + bit.getSum(5));
    System.out.println("Bereichssumme 2 bis 5: " + bit.rangeSum(2, 5));
    bit.update(3, 6); // 6 zu Index 3 hinzufügen
    System.out.println("Neue Bereichssumme 2 bis 5: " + bit.rangeSum(2, 5));
}
}

```

Ausgabe:

Summe von 0 bis 5: 12

Bereichssumme 2 bis 5: 9

Neue Bereichssumme 2 bis 5: 15

Blog 7: Such- und Simulationsalgorithmen in Java

Such- und Simulationsalgorithmen lösen Pfadfindungs- und probabilistische Probleme. Lassen Sie uns A*-Suche und Monte-Carlo-Simulation erkunden.

1. A*-Suche: Heuristische Pfadfindung

A* ist ein informierter Suchalgorithmus, der eine Heuristik verwendet, um den kürzesten Pfad in einem Graphen zu finden, indem er die Stärken von Dijkstras und gierigem Suchen kombiniert. Er wird weit verbreitet in Spielen und Navigation verwendet.

Java-Implementierung

```
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); // Manhattan-Distanz
    }

    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 = {};          // Richtungen hinzufügen
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[newX][newY]) {
            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("Kein Pfad gefunden!");
}

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("Pfad:");
    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);
}
}

```

Ausgabe:

Pfad:

```

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

```

2. Monte-Carlo-Simulation: Probabilistische Schätzung

Monte-Carlo-Methoden verwenden zufällige Stichproben, um Ergebnisse zu schätzen, wie z.B. π durch Simulation von Punkten in einem Quadrat und Kreis zu approximieren.

Java-Implementierung

```

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++; // Innerhalb des Einheitskreises
        }
        return 4.0 * insideCircle / points; // Verhältnis * 4 approximiert
    }

    public static void main(String[] args) {

```

```
int points = 1000000;
double pi = estimatePi(points);
System.out.println("Geschätzte mit " + points + " Punkten: " + pi);
System.out.println("Tatsächliche : " + Math.PI);
}
}
```

Ausgabe (variiert aufgrund der Zufälligkeit):

Geschätzte mit 1000000 Punkten: 3.1418

Tatsächliche : 3.141592653589793