高度なデータ構造の Java

データ構造は効率的なアルゴリズムの基盤です。ここでは、スキップリスト、ユニオンファインド、AVL 木、およびバイナリインデックスツリーの 4 つの強力なデータ構造を探索します。これらは、高速な検索、ユニオン、バランス、または範囲クエリが必要なシナリオで広く使用されます。

1. スキップリスト：確率的検索

スキップリストは、O(log n) の平均時間複雑度で高速な検索、挿入、削除を可能にする層化リンクリストです。バランスツリーの代替手段を提供します。

Java 実装

```
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));
}
```

```
}
```

出力:

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

2. ユニオンファインド（不連結集合）：接続性の追跡

ユニオンファインドは、パス圧縮とランクヒューリスティックを使用して、ユニオンとファインド操作をほぼ O(1) の平均時間で効率的に管理します。

Java 実装

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

出力:

```
0 and 4 connected: true
2 and 4 connected: false
```

3. AVL 木：自己バランス二分探索木

AVL 木は、サブツリーの高さ差（バランスファクター）が最大 1 である自己バランス二分探索木で、O(log n) の操作を保証します。

Java 実装

```
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);
    }
}
```

出力: `Preorder: 30 20 10 25 40 50`

4. バイナリインデックスツリー（フェニウィックツリー）：範囲クエリ

バイナリインデックスツリー（BIT）は、O(log n) の時間で範囲和クエリと更新を効率的に処理し、競プログラミングでよく使用されます。

Java 実装

```
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("Sum from 0 to 5: " + bit.getSum(5));
        System.out.println("Range sum 2 to 5: " + bit.rangeSum(2, 5));
        bit.update(3, 6); // Add 6 to index 3
        System.out.println("New range sum 2 to 5: " + bit.rangeSum(2, 5));
    }
}
```

出力:

```
Sum from 0 to 5: 12
Range sum 2 to 5: 9
New range sum 2 to 5: 15
```

ブログ 7: Java での検索とシミュレーションアルゴリズム

検索とシミュレーションアルゴリズムは、パス探索と確率的問題を解決します。ここでは、A* 検索とモンテカルロシミュレーションを探索します。

1. A* 検索：ヒューリスティックパス探索

A* は、ヒューリスティックを使用してグラフ内の最短パスを探索する情報検索アルゴリズムで、Dijkstra と貪欲検索の強みを組み合わせています。ゲームやナビゲーションで広く使用されます。

Java 実装

```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); // マンハッタン距離
    }

    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 = {};          // 方向配列を定義
        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("No path found!");
    }

    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("Path:");
        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);
    }
```

```
}
```

出力:

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

2. モンテカルロシミュレーション：確率的推定

モンテカルロ法は、ランダムサンプリングを使用して結果を推定し、例えば、正方形と円内の点をシミュレートしてπを近似します。

Java 実装

```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++; // 円内
        }
        return 4.0 * insideCircle / points; // 比率 * 4 で  を近似
    }

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

出力（ランダム性により変動）：

```
Estimated  with 1000000 points: 3.1418
Actual  : 3.141592653589793
```