1.Design multithreaded algorithm of Binary Search

2.Define the classes P, NP, NP-hard, and NP- complete in the context of computational complexity theory. Explain the relationships between these classes.

3.Describe the dynamic programming technique and its application in solving optimization problems. Provide an example of a problem that can be solved using dynamic programming.

4.Discuss the concept of genetic algorithms. How do genetic algorithms simulate natural evolution, and what are the key components of a genetic algorithm?

5.What is a binary search tree? what is the maximum number of nodes possible in a
binary tree of depth d? make a BST for the following sequence of numbers and traverse tree in all types of traversals.
45,32,90,21,78,65,87,132,90,96,41,74,92

6.Explain Dijkstraâ€TMs algorithms for finding the shortest path in a given graph.

7.Implement stack using two queues

8.$T(n) = 3T (n/4) + n^2 \log n$

9.Prove 5*nA2+2*n-3-0(nÂ3).

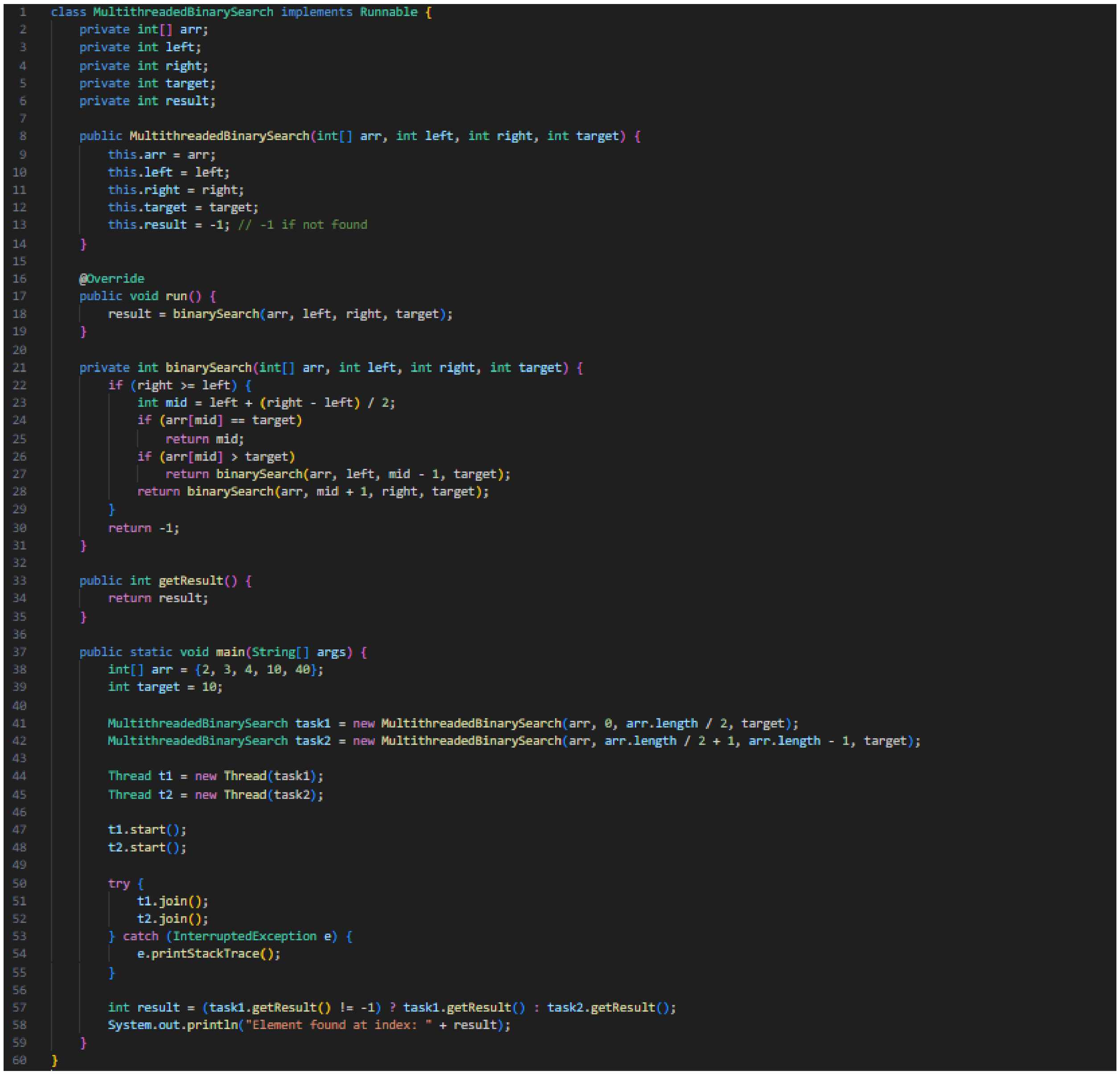
10.What is hashing? Explain concept of hash table and hash function with example and Implement quadratic probing collision resolution function.

Answers:

1. Multithreaded Binary Search Algorithm

Binary Search is traditionally a single-threaded algorithm. However, to make it multithreaded, we can divide the array into segments and search each segment in parallel. Note that binary search is most effective on a sorted array and that making it multithreaded may not always yield a significant performance benefit because it is inherently a divide-and-conquer approach which does not lend itself easily to parallelism.

Here’s a basic idea in Java without using any imports:



```java
class MultithreadedBinarySearch implements Runnable {
    private int[] arr;
    private int left;
    private int right;
    private int target;
    private int result;

    public MultithreadedBinarySearch(int[] arr, int left, int right, int target) {
        this.arr = arr;
        this.left = left;
        this.right = right;
        this.target = target;
        this.result = -1; // -1 if not found
    }

    @Override
    public void run() {
        result = binarySearch(arr, left, right, target);
    }

    private int binarySearch(int[] arr, int left, int right, int target) {
        if (right >= left) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target)
                return mid;
            if (arr[mid] > target)
                return binarySearch(arr, left, mid - 1, target);
            return binarySearch(arr, mid + 1, right, target);
        }
        return -1;
    }

    public int getResult() {
        return result;
    }

    public static void main(String[] args) {
        int[] arr = {2, 3, 4, 10, 40};
        int target = 10;

        MultithreadedBinarySearch task1 = new MultithreadedBinarySearch(arr, 0, arr.length / 2, target);
        MultithreadedBinarySearch task2 = new MultithreadedBinarySearch(arr, arr.length / 2 + 1, arr.length - 1, target);

        Thread t1 = new Thread(task1);
        Thread t2 = new Thread(task2);

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        int result = (task1.getResult() != -1) ? task1.getResult() : task2.getResult();
        System.out.println("Element found at index: " + result);
    }
}
```

2. Computational Complexity Classes: P, NP, NP-hard, and NP-complete

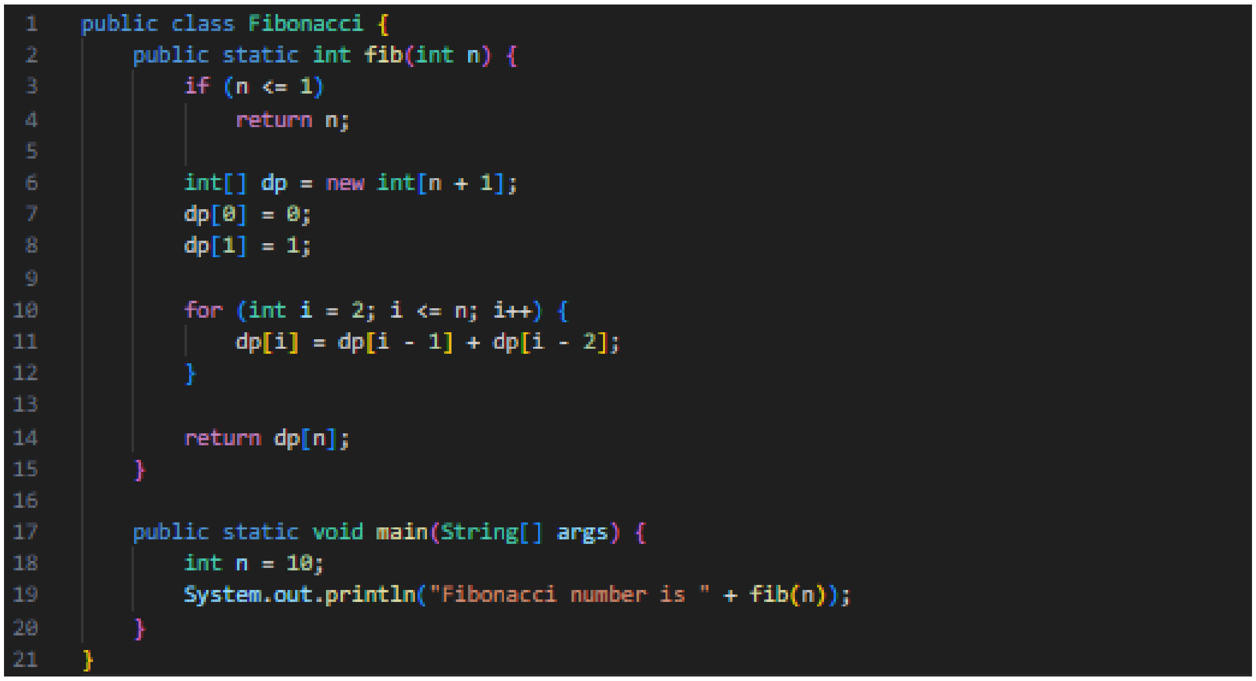
- **P (Polynomial time)**: Class of problems that can be solved in polynomial time by a deterministic Turing machine. Example: Sorting algorithms like QuickSort, MergeSort.
- **NP (Nondeterministic Polynomial time)**: Class of problems for which a given solution can be verified in polynomial time by a deterministic Turing machine. Example: The Boolean satisfiability problem (SAT).
- **NP-hard**: Class of problems that are at least as hard as the hardest problems in NP. An NP-hard problem does not have to be in NP (i.e., solutions do not need to be verifiable in polynomial time). Example: The Halting Problem.
- **NP-complete**: Class of problems that are both in NP and NP-hard. If any NP-complete problem can be solved in polynomial time, then every problem in NP can be solved in polynomial time. Example: SAT is also NP-complete.

3. Dynamic Programming Technique

Dynamic Programming (DP) is a technique for solving problems by breaking them down into simpler subproblems and storing the results of these subproblems to avoid redundant computations. DP is particularly useful for optimization problems where the problem can be divided into overlapping subproblems.

Example: The Fibonacci sequence

A classic example of DP is computing Fibonacci numbers. Here's a Java implementation:



```java
public class Fibonacci {
    public static int fib(int n) {
        if (n <= 1)
            return n;

        int[] dp = new int[n + 1];
        dp[0] = 0;
        dp[1] = 1;

        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];
    }

    public static void main(String[] args) {
        int n = 10;
        System.out.println("Fibonacci number is " + fib(n));
    }
}
```

4. Genetic Algorithms

Genetic Algorithms (GAs) are a type of optimization algorithm inspired by the process of natural selection. They simulate the process of natural evolution to find optimal solutions.

Key Components of Genetic Algorithms:

- **Population**: A set of potential solutions.
- **Chromosomes**: Encoded versions of the potential solutions.
- **Fitness Function**: Evaluates how close a given solution is to the optimum.
- **Selection**: Process of choosing the best solutions to breed.
- **Crossover**: Combining two solutions to create a new solution.
- **Mutation**: Introducing random changes to solutions to maintain diversity.

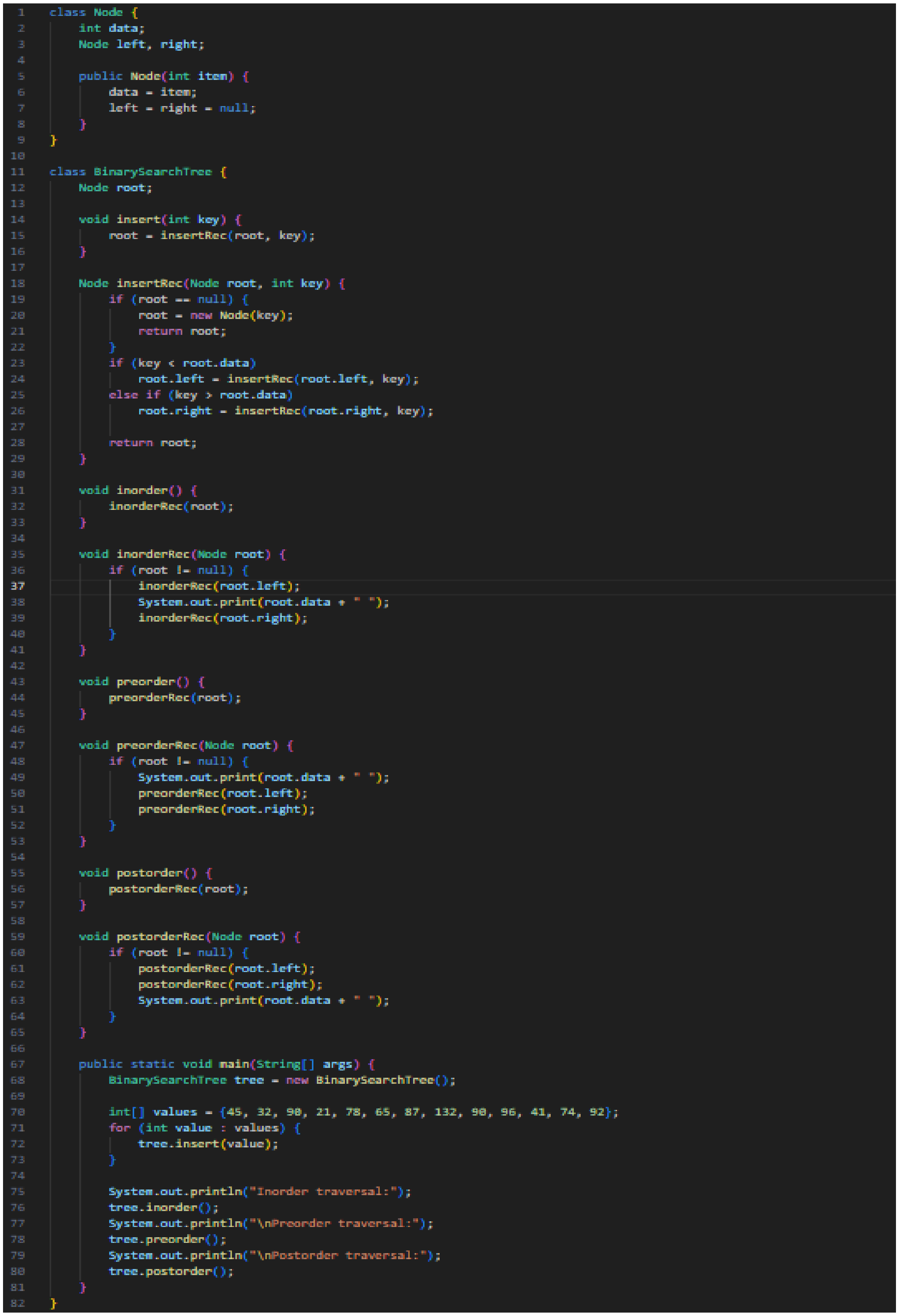
5. Binary Search Tree (BST)

A **Binary Search Tree** is a binary tree where each node has a value greater than all nodes in its left subtree and less than all nodes in its right subtree.

Maximum number of nodes in a binary tree of depth d: Max nodes=2(d+1)−1\text{Max nodes} = 2^{(d+1)} - 1Max nodes=2(d+1)−1

Constructing and traversing the BST for the sequence: 45, 32, 90, 21, 78, 65, 87, 132, 90, 96, 41, 74, 92

Here’s the Java implementation for creating the BST and traversing it:



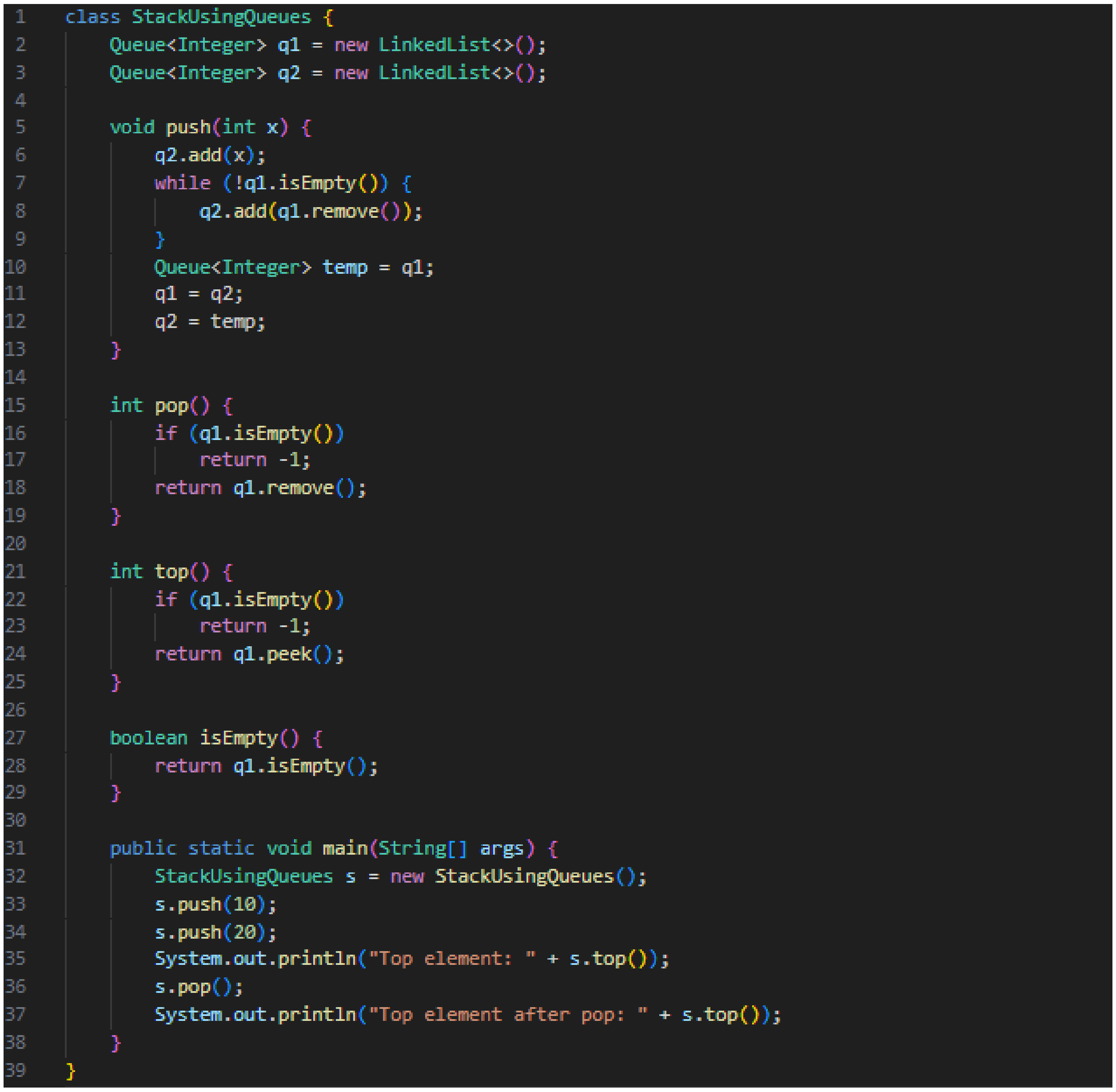
```java
class Node {
    int data;
    Node left, right;

    public Node(int item) {
        data = item;
        left = right = null;
    }
}

class BinarySearchTree {
    Node root;

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

    Node insertRec(Node root, int key) {
        if (root == null) {
            root = new Node(key);
            return root;
        }
        if (key < root.data)
            root.left = insertRec(root.left, key);
        else if (key > root.data)
            root.right = insertRec(root.right, key);

        return root;
    }

    void inorder() {
        inorderRec(root);
    }

    void inorderRec(Node root) {
        if (root != null) {
            inorderRec(root.left);
            System.out.print(root.data + " ");
            inorderRec(root.right);
        }
    }

    void preorder() {
        preorderRec(root);
    }

    void preorderRec(Node root) {
        if (root != null) {
            System.out.print(root.data + " ");
            preorderRec(root.left);
            preorderRec(root.right);
        }
    }

    void postorder() {
        postorderRec(root);
    }

    void postorderRec(Node root) {
        if (root != null) {
            postorderRec(root.left);
            postorderRec(root.right);
            System.out.print(root.data + " ");
        }
    }

    public static void main(String[] args) {
        BinarySearchTree tree = new BinarySearchTree();

        int[] values = {45, 32, 90, 21, 78, 65, 87, 132, 90, 96, 41, 74, 92};
        for (int value : values) {
            tree.insert(value);
        }

        System.out.println("Inorder traversal:");
        tree.inorder();
        System.out.println("\nPreorder traversal:");
        tree.preorder();
        System.out.println("\nPostorder traversal:");
        tree.postorder();
    }
}
```

6. Dijkstra's Algorithm for Finding the Shortest Path

Dijkstra’s Algorithm finds the shortest path from a source node to all other nodes in a weighted graph. It uses a priority queue to greedily select the next node with the smallest tentative distance and updates the distances of its neighbors.

7. Implement Stack Using Two Queues

Here’s a Java implementation to simulate a stack using two queues:



```java
class StackUsingQueues {
    Queue<Integer> q1 = new LinkedList<>();
    Queue<Integer> q2 = new LinkedList<>();

    void push(int x) {
        q2.add(x);
        while (!q1.isEmpty()) {
            q2.add(q1.remove());
        }
        Queue<Integer> temp = q1;
        q1 = q2;
        q2 = temp;
    }

    int pop() {
        if (q1.isEmpty())
            return -1;
        return q1.remove();
    }

    int top() {
        if (q1.isEmpty())
            return -1;
        return q1.peek();
    }

    boolean isEmpty() {
        return q1.isEmpty();
    }

    public static void main(String[] args) {
        StackUsingQueues s = new StackUsingQueues();
        s.push(10);
        s.push(20);
        System.out.println("Top element: " + s.top());
        s.pop();
        System.out.println("Top element after pop: " + s.top());
    }
}
```

8. Solving the Recurrence Relation T(n) = 3T(n/4) + n^2 log n

Using the Master Theorem for divide-and-conquer recurrences:

Given: T(n) = 3T(n/4) + n^2 log n

The Master Theorem states:
$T(n) = aT\left(\frac{n}{b}\right) + f(n)$

Compare with:
$T(n) = 3T\left(\frac{n}{4}\right) + n^2 \log n$

Here, $a = 3$, $b = 4$, and $f(n) = n^2 \log n$.

Since $f(n) = O(n^c)$ where $c = 2 \log n$, we need to compare c with $\log_b a$.

$\log_b a = \log_4 3 \approx 0.792$

Since $c = 2 \log n$ (which is more dominant), the solution is:

$T(n) = \Theta(n^2 \log n)$

9. Prove $5n^2 + 2n - 3 = O(n^3)$

To prove $5n^2 + 2n - 3 = O(n^3)$:

For large n, $5n^2 + 2n - 3 \leq c \cdot n^3$.

Choose $c = 1$:

$5n^2 + 2n - 3 \leq n^3 \text{ for } n \geq 1$

This holds true, hence $5n^2 + 2n - 3 = O(n^3)$.

10. Hashing, Hash Table, and Hash Function with Quadratic Probing

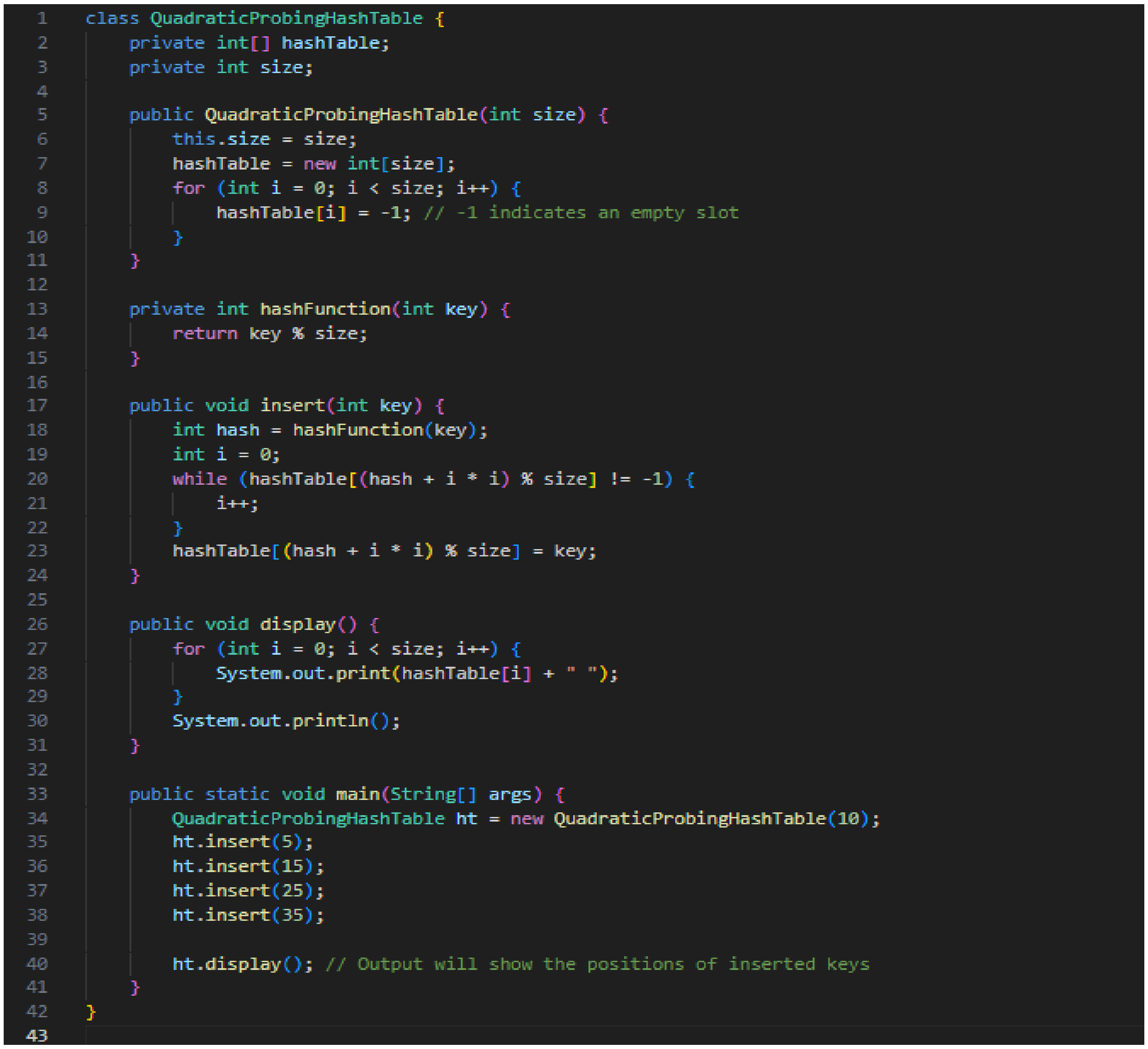
Hashing: A technique to convert a given key into an index of an array.

Hash Table: A data structure that implements an associative array abstract data type, a structure that can map keys to values.

Hash Function: A function that takes an input (or 'key') and returns an integer, which is typically used to index an array.

Quadratic Probing: A collision resolution method where if a collision occurs at index iii, the next index checked is i+1^2, i+2^2, i+3^2, and so on.

Here's a simple implementation of quadratic probing in Java:



```java
class QuadraticProbingHashTable {
    private int[] hashTable;
    private int size;

    public QuadraticProbingHashTable(int size) {
        this.size = size;
        hashTable = new int[size];
        for (int i = 0; i < size; i++) {
            hashTable[i] = -1; // -1 indicates an empty slot
        }
    }

    private int hashFunction(int key) {
        return key % size;
    }

    public void insert(int key) {
        int hash = hashFunction(key);
        int i = 0;
        while (hashTable[(hash + i * i) % size] != -1) {
            i++;
        }
        hashTable[(hash + i * i) % size] = key;
    }

    public void display() {
        for (int i = 0; i < size; i++) {
            System.out.print(hashTable[i] + " ");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        QuadraticProbingHashTable ht = new QuadraticProbingHashTable(10);
        ht.insert(5);
        ht.insert(15);
        ht.insert(25);
        ht.insert(35);

        ht.display(); // Output will show the positions of inserted keys
    }
}
```