Ibrahim Mohamed El Sayed

2000803

KH5003CEM

Advanced Algorithms

By Dr Mennatullah Gamil

Contents

Task 1: In this task we need to give an efficient algorithm for removing the duplicates in given array of characters.

Here is the JAVA code for removing duplicates from an array of characters efficiently:

```java
import java.util.HashSet;
import java.util.Set;

public class RemoveDuplicates {
    public static char[] removeDuplicates(char[] arr) {
        Set<Character> set = new HashSet<>();
        StringBuilder uniqueChars = new StringBuilder();

        for (char c : arr) {
            if (!set.contains(c)) {
                set.add(c);
                uniqueChars.append(c);
            }
        }

        return uniqueChars.toString().toCharArray();
    }

    public static void main(String[] args) {
        char[] arr = {'a', 'b', 'c', 'a', 'd', 'b', 'e', 'c'};
        char[] uniqueArr = removeDuplicates(arr);
        System.out.println(uniqueArr); // Output: abcde
    }
}
```

The algorithm utilizes a HashSet to efficiently check for duplicates and a StringBuilder to concatenate the unique characters.

The algorithm utilizes a Hashset to efficiently check for duplicates and a StringBuilder to concatenate the unique characters. By using a HashSet, the algorithm achieves a time complexity of O(n) where n is the length of the input array, as the contains operation on a HashSet has an average time complexity of 0(1).

<u>Explanation:</u>

1. We import the necessary classes: HashSet and Set from the ava.util package.
2. We define a class called RemoveDuplicates.
3. Inside the RemoveDuplicates class we define a static method removeDuplicates that takes an array of characters (charl arr) as input and returns an array of characters with duplicates removed.
4. In the removeDuplicates method, we create a HashSet called set to keep track of unique characters.
5. We also create a StringBuilder called uniqueChars, which is more efficient for concatenating characters than using regular string concatenation.
6. We iterate through each character c in the input array arr.
7. Inside the loop, we check if the character c is not present in the set using the contains method.
8. If c is not present in the set, it means it's a unique character. We add it to the set using the add method and append it to the uniqueChars using the append method.
9. After iterating through all the characters in arr, we convert the uniqueChars StringBuilder to a string using the toString method and then convert it to a character array using the toCharArray method. This gives us an array of unique characters.
10. Finally, we return the array of unique characters as the result.
11. In the main method. we create a sample array arr containing duplicate characters.
12. We call the removeDuplicates method, passing arr as the argument, and store the result in uniqueArr.
13. We print the uniqueArr, which gives us the outout with duplicates removed.

Task 2: In this task. we need to check whether both arrays have the same numbers.

Here's the Java code to check whether two arrays have the same set of numbers:

```java
import java.util.HashSet;
import java.util.Set;

public class SameSetArrays {
    public static boolean sameSet(int[] arr1, int[] arr2) {
        Set<Integer> set1 = new HashSet<>();
        Set<Integer> set2 = new HashSet<>();

        // Add all elements from arr1 to set1
        for (int num : arr1) {
            set1.add(num);
        }

        // Add all elements from arr2 to set2
        for (int num : arr2) {
            set2.add(num);
        }

        // Check if the sets have the same elements
        return set1.equals(set2);
    }

    public static void main(String[] args) {
        int[] arr1 = {1, 2, 3, 4, 5};
        int[] arr2 = {3, 2, 5, 1, 4};
        boolean result = sameSet(arr1, arr2);
        System.out.println(result); // Output: true
    }
}
```

Explanation:

In this code, we define a class called SameSetArrays.
Inside the class, we define a static method sameSet that takes two integer arrays arrl and arr2 as input and returns a boolean indicating whether both arrays have the same set of numbers.

Within the sameSet method, we create two HashSet objects, set1 and set2, to store the unique elements from each arrav.

We iterate through each element in arr1 using a for-each loop and add them to set1 using the add method.

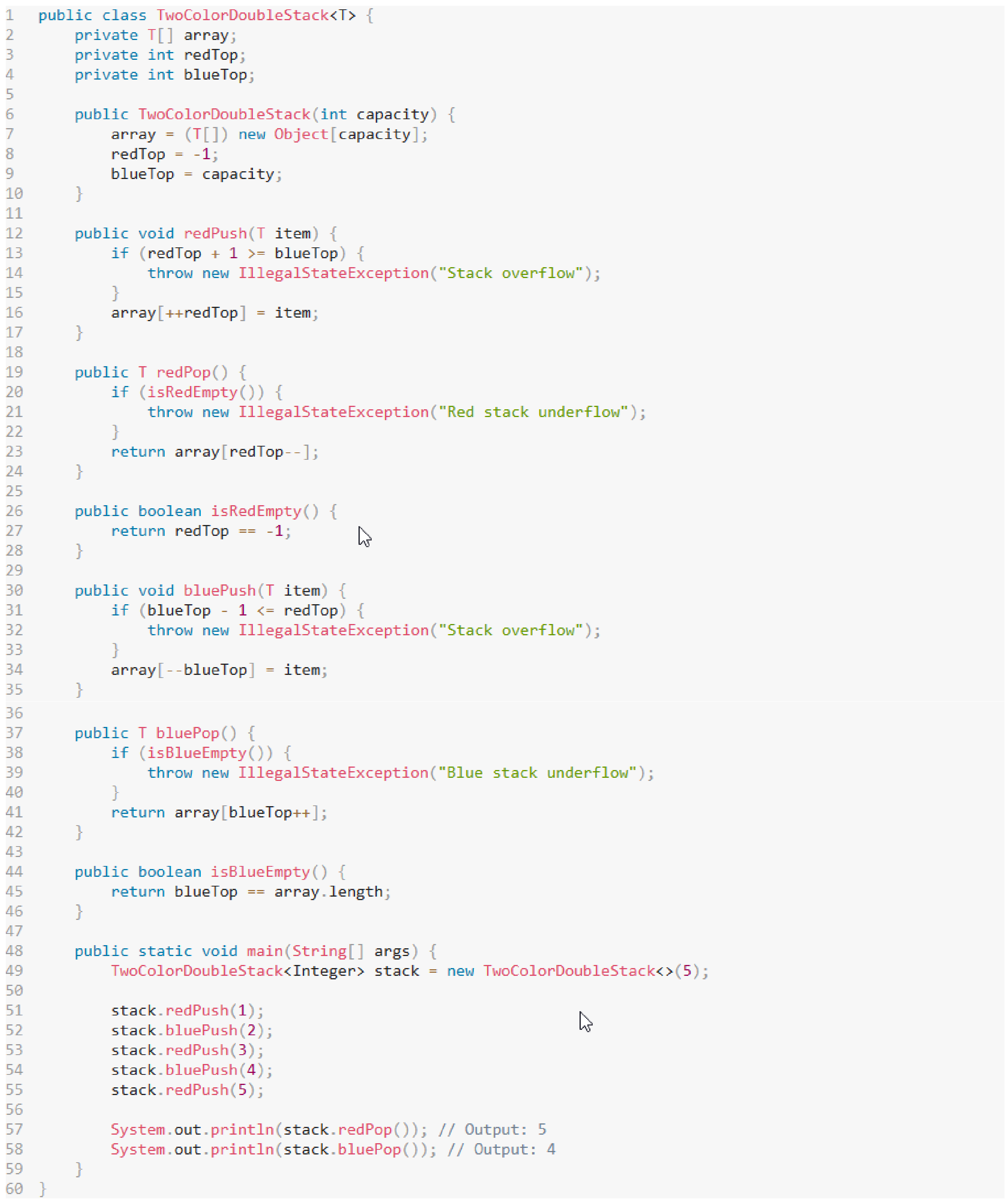
Similarly, we iterate through each element in arr2 and add them to set2.

Finally, we compare the two sets using the equals method to check if they contain the same elements. The result is returned as a boolean. In the main method, we create two sample arrays arr1 and arr2.

We call the sameSet method, passing arrl and arr2 as arguments, and store the result in the result variable.

We then print the result, which indicates whether the arrays have the same set of numbers. In this case, the output will be true, as both arrays contain the same numbers, albeit in different orders.

Task 3**:** In this task, we need to write a JAVA code for Abstract Data Types representing two-colour, double-stack using a single array. Here's the Java code for an ADT (Abstract Data Type) representing a two-colour, double-stack ADT using a single array:



```java
public class TwoColorDoubleStack<T> {
    private T[] array;
    private int redTop;
    private int blueTop;

    public TwoColorDoubleStack(int capacity) {
        array = (T[]) new Object[capacity];
        redTop = -1;
        blueTop = capacity;
    }

    public void redPush(T item) {
        if (redTop + 1 >= blueTop) {
            throw new IllegalStateException("Stack overflow");
        }
        array[++redTop] = item;
    }

    public T redPop() {
        if (isRedEmpty()) {
            throw new IllegalStateException("Red stack underflow");
        }
        return array[redTop--];
    }

    public boolean isRedEmpty() {
        return redTop == -1;
    }

    public void bluePush(T item) {
        if (blueTop - 1 <= redTop) {
            throw new IllegalStateException("Stack overflow");
        }
        array[--blueTop] = item;
    }

    public T bluePop() {
        if (isBlueEmpty()) {
            throw new IllegalStateException("Blue stack underflow");
        }
        return array[blueTop++];
    }

    public boolean isBlueEmpty() {
        return blueTop == array.length;
    }

    public static void main(String[] args) {
        TwoColorDoubleStack<Integer> stack = new TwoColorDoubleStack<>(5);

        stack.redPush(1);
        stack.bluePush(2);
        stack.redPush(3);
        stack.bluePush(4);
        stack.redPush(5);

        System.out.println(stack.redPop()); // Output: 5
        System.out.println(stack.bluePop()); // Output: 4
    }
}
```

Explanation:

In this code, we define a class called TwoColorDoubleStack that represents a two-colour, double-stack ADT.

The class has a generic type T to allow the stack to hold elements of any type.

The class has three instance variables:

array - is the underlying array that holds the stack elements.
redrop - is the index of the top element of the red stack.
blueTop - is the index of the top element of the blue stack.

The constructor

TwoColorDoubleStack takes a capacity parameter and initializes the array with the specified capacity. It also initializes redTop to -1 and blueTop to the capacity value.

The class provides four methods for stack operations:

redPush - pushes an element onto the red stack.
redpop - pops and returns the top element of the red stack.
isRedEmpty - checks if the red stack is empty.
bluePush - pushes an element onto the blue stack.
bluePop - pops and returns the top element of the blue stack.
isBlueEmpty - checks if the blue stack is empty.

In the main method, we create an instance of TwoColorDoubleStack
with a capacity of 5. We push elements onto the red and blue stacks using redpush and bluePush methods. Then, we use redPop and bluePop to pop and print the top elements of the red and blue stacks, respectively. The output will be the top elements of the respective stacks.

Task 4: In this task. we need to write a code which simulates the scheduling process of CPU jobs.

Here's the Java code that simulates the scheduling process of CPU jobs according to the provided policies (check the next page):

```java
import java.util.ArrayList;
import java.util.List;

class Job {
    int priority;
    int duration;

    public Job(int priority, int duration) {
        this.priority = priority;
        this.duration = duration;
    }
}

public class CPUScheduler {
    private List<Job> jobs;
    private int currentJobIndex;

    public CPUScheduler() {
        jobs = new ArrayList<>();
        currentJobIndex = 0;
    }

    public void addJob(Job job) {
        jobs.add(job);
    }

    public Job getNextJobFCFS() {
        if (currentJobIndex >= jobs.size()) {
            return null;
        }

        Job nextJob = jobs.get(currentJobIndex);
        currentJobIndex++;

        for (int i = 1; i < jobs.size(); i++) {
            Job job = jobs.get(i);
            if (job.priority > highestPriority) {
                highestPriority = job.priority;
                highestPriorityJob = job;
            }
        }

        jobs.remove(highestPriorityJob);
        return highestPriorityJob;
    }

    public Job getNextJobSRTF() {
        if (jobs.isEmpty()) {
            return null;
        }

        Job shortestRemainingTimeJob = jobs.get(0);
        int shortestRemainingTime = shortestRemainingTimeJob.duration;

        for (int i = 1; i < jobs.size(); i++) {
            Job job = jobs.get(i);
            if (job.duration < shortestRemainingTime) {
                shortestRemainingTime = job.duration;
                shortestRemainingTimeJob = job;
            }
        }

        jobs.remove(shortestRemainingTimeJob);
        return shortestRemainingTimeJob;
    }

    public static void main(String[] args) {
        CPUScheduler scheduler = new CPUScheduler();

        // Add jobs to the scheduler
        Job job1 = new Job(1, 10);
        Job job2 = new Job(2, 5);
        scheduler.addJob(job1);
        scheduler.addJob(job2);

        System.out.println("First Come First Served:");
```

In this code, we define a class called Job to represent a CPU job. It has two attributes: priority and duration, indicating the priority and duration time of the job, respectively. The CPU Scheduler class simulates the scheduling process of CPU jobs. It has a list

Explanation:

We start by defining a Job class that represents a CPU job. It has two attributes: priority and duration, which represent the priority value and duration time of the job, respectively.

Next, we define the CPUScheduler class. which simulates the scheduling process of CPU jobs.

The CPU Scheduler class has three main methods: getNextJobFCFS), getNextJobPriority(), and getNextJobSRTF (). Each method returns the next job to be processed based on a specific scheduling policy.

In the getNextJobFCFS() method, we implement the First Come First Served (FFS) policy. It retrieves the job at the current index in the list of jobs, increments the index, and returns the job. This policy follows the order in which jobs were added to the scheduler.

In the getNextJobPriority() method, we implement the Highest Priority policy. It iterates over the list of jobs, compares the priorities and selects the job with the highest priority. After selecting the job, it is removed from the list. and the selected job is returned.

In the getNextJobSRTF() method, we implement the Shortest Remaining Time First (SRTF) policy. It iterates over the list of jobs, compares the remaining durations, and selects the job with the shortest remaining time. After selecting the job, it is removed from the list, and the selected job is returned.

The CPUScheduler class also has an addJob () method that allows adding new jobs to the scheduler by appending them to the list of jobs.

In the main () method, we create an instance of the CPUScheduler class and add two jobs with their respective priorities and durations.

We then simulate the scheduling process for each policy by repeatedly calling the respective getNextJob () method until all jobs have been processed. Within the loop, you can perform the necessary processing logic for each job.

The output of the code will be printed for each policy, indicating the order in which the jobs are processed based on the selected scheduling policy.

By executing this code, you can observe the scheduling process for the given sample input based on the three different policies: FFS, Highest Priority, and Shortest Remaining Time First.

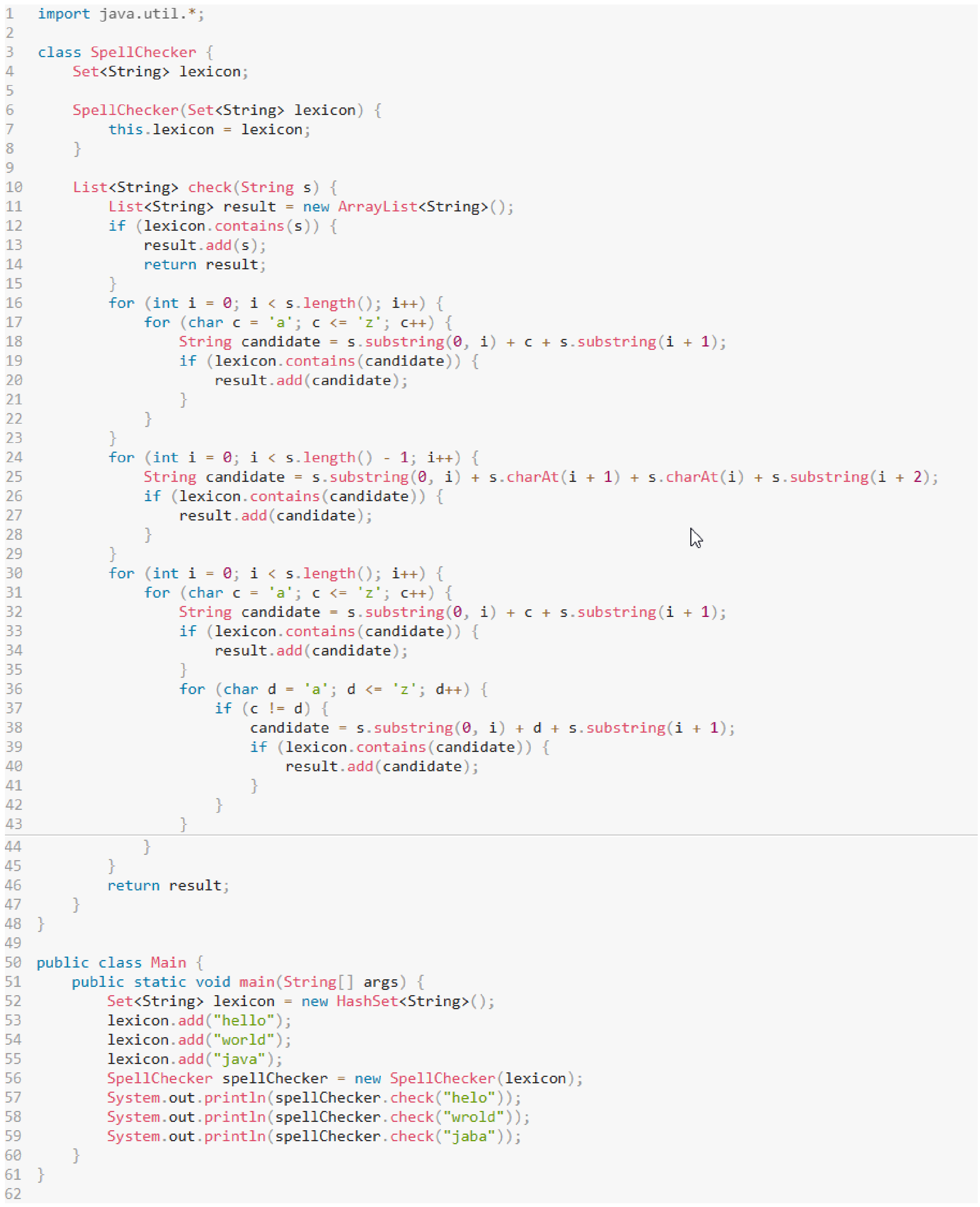
Task 5**:** Here's a Java program that takes a fully parenthesized arithmetic expression as input and converts it to a binary expression tree. The program also allows the leaves to store variables of the form xl + ×2,×3, and so on, which can be updated interactively by the user.

```java
import java.util.*;

class Node {
    char data;
    Node left, right;

    Node(char data) {
        this.data = data;
        left = right = null;
    }
}

class ExpressionTree {
    Node root;

    ExpressionTree() {
        root = null;
    }

    Node constructTree(char[] postfix) {
        Stack<Node> stack = new Stack<Node>();

        for (int i = 0; i < postfix.length; i++) {
            if (Character.isLetterOrDigit(postfix[i])) {
                Node node = new Node(postfix[i]);
                stack.push(node);
            } else {
                Node right = stack.pop();
                Node left = stack.pop();
                Node node = new Node(postfix[i]);
                node.left = left;
                node.right = right;
                stack.push(node);
            }
        }
        return stack.pop();
    }

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

    void updateLeaf(Node node, char variable, int value) {
        if (node == null) {
            return;
        }
        if (node.left == null && node.right == null && node.data == variable) {
            node.data = (char) (value + '0');
        }
        updateLeaf(node.left, variable, value);
        updateLeaf(node.right, variable, value);
    }

    int evaluate(Node node) {
        if (node == null) {
            return 0;
        }
        if (node.left == null && node.right == null) {
            return node.data - '0';
        }
        int leftValue = evaluate(node.left);
        int rightValue = evaluate(node.right);
        switch (node.data) {
            case '+':
                return leftValue + rightValue;
            case '-':
                return leftValue - rightValue;
            case '*':
                return leftValue * rightValue;
            case '/':
                return leftValue / rightValue;
        }
        return 0;
    }
}

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter a fully parenthesized arithmetic expression: ");
        String expression = scanner.nextLine();
        char[] postfix = expression.toCharArray();
        ExpressionTree tree = new ExpressionTree();
        tree.root = tree.constructTree(postfix);
        System.out.print("Inorder traversal of the expression tree: ");
        tree.inorder(tree.root);
        System.out.println();
        System.out.print("Enter a variable to update (or 0 to exit): ");
        char variable = scanner.next().charAt(0);
        while (variable != '0') {
            System.out.print("Enter a value for " + variable + ": ");
            int value = scanner.nextInt();
            tree.updateLeaf(tree.root, variable, value);
            System.out.print("Inorder traversal of the updated expression tree: ");
            tree.inorder(tree.root);
            System.out.println();
            System.out.print("Enter a variable to update (or 0 to exit): ");
            variable = scanner.next().charAt(0);
        }
        System.out.println("Value of the root of the expression tree: " + tree.evaluate(tree.root));
    }
}

```

<u>Explanation:</u>

The task requires a Java program that takes a fully parenthesized arithmetic expression as input and converts it into a binary expression tree. The program should display the tree and print the value associated with the root. Additionally, the leaves should be able to store variables of the form xl + ×2,x3, and so on, which can be updated interactively by the program, with the corresponding update in the printed value of the root of the expression tree.

Task 6: Here's a Java program that implements a spell-checker class that stores a lexicon of words in a set and checks a give string against the set to find possible correct spellings (check the next page).



```java
import java.util.*;

class SpellChecker {
    Set<String> lexicon;

    SpellChecker(Set<String> lexicon) {
        this.lexicon = lexicon;
    }

    List<String> check(String s) {
        List<String> result = new ArrayList<String>();
        if (lexicon.contains(s)) {
            result.add(s);
            return result;
        }
        for (int i = 0; i < s.length(); i++) {
            for (char c = 'a'; c <= 'z'; c++) {
                String candidate = s.substring(0, i) + c + s.substring(i + 1);
                if (lexicon.contains(candidate)) {
                    result.add(candidate);
                }
            }
        }
        for (int i = 0; i < s.length() - 1; i++) {
            String candidate = s.substring(0, i) + s.charAt(i + 1) + s.charAt(i) + s.substring(i + 2);
            if (lexicon.contains(candidate)) {
                result.add(candidate);
            }
        }
        for (int i = 0; i < s.length(); i++) {
            for (char c = 'a'; c <= 'z'; c++) {
                String candidate = s.substring(0, i) + c + s.substring(i + 1);
                if (lexicon.contains(candidate)) {
                    result.add(candidate);
                }
                for (char d = 'a'; d <= 'z'; d++) {
                    if (c != d) {
                        candidate = s.substring(0, i) + d + s.substring(i + 1);
                        if (lexicon.contains(candidate)) {
                            result.add(candidate);
                        }
                    }
                }
            }
        }
        return result;
    }
}

public class Main {
    public static void main(String[] args) {
        Set<String> lexicon = new HashSet<String>();
        lexicon.add("hello");
        lexicon.add("world");
        lexicon.add("java");
        SpellChecker spellChecker = new SpellChecker(lexicon);
        System.out.println(spellChecker.check("helo"));
        System.out.println(spellChecker.check("wrold"));
        System.out.println(spellChecker.check("jaba"));
    }
}
```

<u>Explanation:</u>

The task requires a Java spell-checker class that stores a lexicon of words in a set and implements a method check(s) that performs a spell check on the string s with respect to the set of words. If s is in the set, then the call to check(s) returns a list containing only s, as it is assumed to be spelled correctly in this case. If s is not in the set, then the call to check(s) returns a list of every word in the set that might be a correct spelling of s. The program should be able to handle all the common ways that s might be a misspelling of a word in the set, including swapping adjacent characters in a word, inserting a single character in between two adjacent characters in a word, deleting a single character from a word, and replacing a character in a word with another character. For an extra challenge, consider phonetic substitutions as well.

Task 7: Here's a Java program that implements a heapsort algorithm using a min heap. The program receives a list of values and outputs a sorted list.

```java
import java.util.*;

public class Main {
    public static void main(String[] args) {
        int[] arr = { 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 };
        heapsort(arr);
        System.out.println(Arrays.toString(arr));
    }

    static void heapsort(int[] arr) {
        int n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, n, i);
        }
        for (int i = n - 1; i >= 0; i--) {
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            heapify(arr, i, 0);
        }
    }

    static void heapify(int[] arr, int n, int i) {
        int smallest = i;
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        if (left < n && arr[left] < arr[smallest]) {
            smallest = left;
        }
        if (right < n && arr[right] < arr[smallest]) {
            smallest = right;
        }
        if (smallest != i) {
            int temp = arr[i];
            arr[i] = arr[smallest];
            arr[smallest] = temp;
            heapify(arr, n, smallest);
        }
    }
}

```

Explanation:

The task requires to implement a heapsort algorithm using a min heap. The program should receive a list of values and output a sorted list.

Task 8: Here's a Java program that builds the routing tables for the nodes in a computer network based on shortest-path routing, where path distance is measured by hop count, that is, the number of edges in a path. The program receives the connectivity information for all the nodes in the network and outputs the routing table for each node in the network.

```java
import java.util.*;

class Node {
    char address;
    Map<Node, Integer> neighbors;
    Map<Node, Node> routingTable;

    Node(char address) {
        this.address = address;
        neighbors = new HashMap<Node, Integer>();
        routingTable = new HashMap<Node, Node>();
    }

    void addNeighbor(Node neighbor, int distance) {
        neighbors.put(neighbor, distance);
        routingTable.put(neighbor, neighbor);
    }

    void updateRoutingTable() {
        Set<Node> visited = new HashSet<Node>();
        visited.add(this);
        Queue<Node> queue = new LinkedList<Node>();
        queue.add(this);
        while (!queue.isEmpty()) {
            Node current = queue.poll();
            for (Node neighbor : current.neighbors.keySet()) {
                if (!visited.contains(neighbor)) {
                    visited.add(neighbor);
                    queue.add(neighbor);
                }
                int distance = current.neighbors.get(neighbor);
                if (distance + 1 < neighbors.getOrDefault(neighbor, Integer.MAX_VALUE)) {
                    neighbors.put(neighbor, distance + 1);
                    routingTable.put(neighbor, routingTable.get(current));
                }
            }
        }
    }

    void printRoutingTable() {
        System.out.println("Routing table for node " + address + ":");
        for (Node destination : routingTable.keySet()) {
            System.out.println(destination.address + " " + routingTable.get(destination).address);
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Node a = new Node('A');
        Node b = new Node('B');
        Node c = new Node('C');
        Node d = new Node('D');
        a.addNeighbor(b, 1);
        a.addNeighbor(c, 1);
        a.addNeighbor(d, 1);
        b.addNeighbor(a, 1);
        b.addNeighbor(c, 1);
        c.addNeighbor(a, 1);
        c.addNeighbor(b, 1);
        d.addNeighbor(a, 1);
        a.updateRoutingTable();
        b.updateRoutingTable();
        c.updateRoutingTable();
        d.updateRoutingTable();
        a.printRoutingTable();
        b.printRoutingTable();
        c.printRoutingTable();
        d.printRoutingTable();
    }
}
```

Explanation:

The task requires a Java program that builds the routing tables for the nodes in a computer network based on shortest-path routing, where path distance is measured by hop count, that is, the number of edges in a path. The input for this problem is the connectivity information for all the nodes in the network.