Student Number

210323041

Algorithms and Data Structures II

Midterm Project

Section 1. The essence of the solution.

My implementation for the midterm projects is composed of three main components and a helper function that runs the command prompt. When the expression is written to the console the said expression is evaluated by the **evaluatePostfix()** function, and based on what is in the expression it is either passed on to the **symbolTable** or the **Stack** for further processing.

Section 2. The explanation

evaluatePostfix() - this function reads the expression that is received by the console. Based on the type of the character (token) it decides whether it should be processed to **Stack, symbolTable** or it is a mathematical operator. If the character is a number it is pushed to the Stack, if it is a letter it is pushed to a **symbolTable** and stored as a variable or handled as an already defined variable.

Stack - basic implementations of a stack data structure with 3 functions in it **push(), pop(),** and **isEmpty().** Every time the **evaluatePostfix()** is called, a new Stack is created and all the numbers are pushed to the array until the mathematical operator is found in the expression. Once this is found the numbers are popped from the stack and the calculations are made.

symbolTable - it is an implementation of a hash table but works similarly to the counting sort. There are two arrays in a **symbolTable, keys,** and **variables.** When the expression ends in a "=" sign, the letter from the expression is added to the **keys** arrays, and the number is added to the **variables** array under the same index as the key. This way the variable index always corresponds to the key index. The **symbolTable** has 4 functions within it: **add(), get(), getIndex(),** and **isValidKey().** The first three functions make use of the **isValidKey()** function before they execute, this way I made sure that only 'A'-'Z' variables are allowed. **add()** function checks if the variable already exists, if yes, then it updates the value for this key, if not, then it creates a new key and adds the corresponding value to it. **get()** is responsible for getting the value based on the key, when the key is used in the expression. **getIndex()** is a helper function that checks the index of the key.

Section 3. Pseudocode

I didn’t write any pseudocode for this assignment. I just worked my way through the code.

Section 4. The data structures

The data structures I implemented in my solution are a stack and a hash table.
I used a stack as it provides a possibility to make the calculation one by one and will always follow the same sequence of operations. It is also fairly simple to implement and use.
I decided to implement a hash table in my solution as this allows for the correct and mistake-free way of storing the key-value pairs, which is essential for this project. I must admit though, it was very challenging to do.

Section 5. The source code presentation

Link to the video:
https://drive.google.com/file/d/1DbYDHvTdfJ9Q2biAqbQ96Uo6m2bZQLnq/view?usp=sharing
Link to the source code file:
https://drive.google.com/file/d/1rE8LzgTUvQkjkOMEyMt0dLIcKQBw5RV5/view?usp=sharing

Whole code is also pasted in this document at the end.

Section 6. The defects

1. When the unidentified variable is called in an expression, the console still returns the calculation of only one token. For example:
   >C 7 +
   Empty stack 7
   While it should give information that the variable is not specified and cannot make calculation on it.
2. In **startInteractiveSession()** function there is a piece of code that evaluates the expression. This piece of code should be in the **evaluatePostfix()** function.

Source Code:

```
// Define stack
class Stack {
    constructor() {
        this.items = [];
    }

    push(item) {
        this.items.push(item);
    }

    pop() {
        if (this.isEmpty()) return "Empty stack";
        return this.items.pop();
    }
    isEmpty() {
        return this.items.length === 0;
    }
}

// Define hash table
class symbolTable {
    constructor() {
        this.keys = []; // Using an array to store keys
        this.values = []; // Using an array to store values
    }

    // Method to add a key-value pair
    add(key, value) {
        //  check if key is valid
        if (!this.isValidKey(key)) {
            console.log("Invalid key! Please use keys from A-Z.");
            return;
        }
        //   check if key already exists, if not add key and value
        else if (this.getIndex(key) === false) {
            this.keys.push(key); // Add key to array
            const index = this.getIndex(key);
            this.values[index] = value; // Add value to array
        }
        //  if key exists, replace value
        else {
            const index = this.getIndex(key);
            this.values[index] = value; // Add value to array
        }
    };

    // Method to get a value by key
    get(key) {
        // check if key is valid
        if (!this.isValidKey(key)) {
            console.log("Invalid key! Please use keys from A-Z.");
            return;
        }
        const index = this.getIndex(key);
        //
        if (index !== -1) {
            return this.values[index]; // Return value if key exists
        }
        return // Return if key does not exist
    };
    // Helper method to get index of a key
    getIndex(key) {
        if (!this.isValidKey(key)) {
            console.log("Invalid key! Please use keys from A-Z.");
            return;
        }
        for (let i = 0; i < this.keys.length; i++) {
            if (this.keys[i] === key) {
                return i; // Return index if key exists
            }
        }
        return false; // Return false if key does not exist
    };

    //   check if key is between A and Z
    isValidKey(key) {
        return /^[A-Z]$/.test(key);
    };

}
// Helper function to check if a token is a capital letter
function isLetter(str) {
    return str.length === 1 && str.match(/[A-Z]/i);
}

dict = new symbolTable();

function evaluatePostfix(expression) {
    let stack = new Stack();

    for (let token of expression.split(' ')) {
        if (!isNaN(token)) {
            stack.push(Number(token)); // Push numbers to stack
        } else if (isLetter(token)) {
            if (dict.get(token)) {
                stack.push(dict.get(token));
            }
        } else {
            // Handle operators
            const operand2 = stack.pop();
            const operand1 = stack.pop();

            switch (token) {
                case '+':
                    stack.push(operand1 + operand2);
                    break;
                case '-':
                    stack.push(operand1 - operand2);
                    break;
                case '*':
                    stack.push(operand1 * operand2);
                    break;
                case '/':
                    stack.push(operand1 / operand2);
                    break;
                default:
                    return "Invalid operator";
            }
        }
    }
    return stack.pop(); // Result will be at the top of the stack
}

const readline = require('readline');

const rl = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

function startInteractiveSession() {
    rl.question('Enter a postfix expression (or type "exit" to quit): ',
(expression) => {
        if (expression === 'exit') {
            rl.close();
        } else {
            const tokens = expression.split(' ');
            // Check if the user is trying to add a variable
            if (tokens.length === 3 && tokens[2] === '=') {
                const variable = tokens[0];
                const value = Number(tokens[1]);
                dict.add(variable, value);
                console.log(`Added variable ${variable} with value
${value}`);
                startInteractiveSession();
            } else {    // Evaluate the expression
                const result = evaluatePostfix(expression);
                console.log(`Result: ${result}`);
                startInteractiveSession();
            }
        }
    });
}
```

```
startInteractiveSession();
```