Algorithms and Data Structures II

Midterm Report

Table of Contents

Section 1: Essence of solution to the challenge

My essence of solution

My solution is a Postfix++ interpreter created with JavaScript that can calculate postfix expression and manage variables using a hash table. The program incorporates a command-line interface (CLI) to interact with the users. The main function uses the CLI by presenting a menu of options to the user. Depending on the user's selection 1 to 5, it allows users to choose between different modes: calculating postfix expressions, searching for the value of variables, inserting values into variables, removing variables and displaying all variables with values. The Postfix++ interpreter implements an array-based stack data structure to calculate the expression using the Last In, First Out (LIFO) principle. The elements are added and removed from the top of the stack or from the left to right in an array. Additionally, variable management is implemented by a hash table using array to store the variables and their respective values. Error handling is also implemented to manage invalid inputs. When encountering errors, appropriate error message will be logged to the console to inform users. These implementation allows effective insertion, search, and deletion operations.

Section 2: Explanation of the original algorithms

STACK Class

A Stack class is created to provide stack operation such as push, pop, peek, isEmpty and print, which are required to evaluate postfix expressions. The Stack class is constructed with an empty array, which will then be used to store tokens.

The push method adds elements onto the stack. This method uses the unshift method of the array to add new elements to the left of the array. This ensures that the most recently added elements will also appear as the leftmost token in the array.

The pop method removes and return the element at the top of the stack. This method uses the shift method of array to remove the first element (the leftmost token) in the array. Since push method adds elements from the left into the array, while the pop method removes the elements from the left as well, it aligns with the Last In, First Out principle of a stack, where the last element added into the stack will be the first element to be removed.

The peek method returns the top element of the stack. This method returns the first element of the array, which is the top of the stack. It is similar to the pop method, but it returns the top of the stack without removing it.

The isEmpty method checks whether the stack is empty. This method checks whether the stack is empty by checking the length of the array. If the array length is zero, it returns true, indicating that the stack is empty. Otherwise, it returns false, indicating that the stack is not empty.

The print method returns the stack as an array. This method is helpful for debugging, it also allows the interpreter to print out the result after the calculations.

Postfix++ Interpreter using Stack

The Postfix++ interpreter main function is to calculate postfix expressions using a stack. The interpreter will first prompt for user input in the Postfix++ Interpreter section.

An empty stack from the Stack class, is then created to store operands during the calculation process. A for loop is then used to iterate through the array of tokens from left to right. When the interpreter encounters a number, it will then be converted from strings into a float point number using the JavaScript built-in parseFloat function, then pushed onto the stack. This ensures that all the numbers have the same data type to prevent incorrect results in calculations. If the token is an alphabetic character (a variable), it is then capitalized using built-in toUpperCase function and pushed onto the stack. This ensures that the variable names are consistent, so each alphabetic character can only hold a value.

For instance, the tokens “6”, “5”, “4” are numbers, so they are converted from strings into float point numbers and pushed onto the stack in that order, resulting the stack that looks like this, [4, 5, 6], the stack top is the leftmost token. This is due to the Last In, First Out (LIFO) principle of stacks, the last item being pushed onto a stack will be the first one to be removed. From example above, the leftmost token in the array, “6”, is first pushed onto the stack, followed by “5” then “4”, which results in “6” being at the bottom of the stack, “5” being at the middle and “4” at the top of the stack.

Nested switch case structure is implemented to handle different operators, which minimizes the code repetition for the parts where their code overlaps. When an operator is encountered, it checks if the operator matches “+” (addition), “-” (subtraction), “*” (multiplication), “/” (division), “%” (modulo), and “^” (exponentiation), where each will proceed with their respective operation if they match. Additionally, the implementation also includes advanced mathematical functionalities. It allows trigonometric calculations like sine ('sin'), cosine ('cos'), and tangent ('tan'), as well as mathematical operations such as ceiling ('ceil') and flooring ('floor'). Furthermore, absolute value ('abs'), rounding ('round'), and logarithmic ('log') operations are also supported to enhance the utility of the program.

When an operator is encountered, such as “%”, the interpreter then pops the top two operands from the stack, in this case, the “4” and “5” is removed from the stack, leaving the stack being [6]. The interpreter then performs the calculation depending on the operator on these operands. In this case, it performs 4 % 5, which means 4 mod 5, giving the result 4. The result will then be pushed back onto the stack, making the stack [4, 6].

The same algorithm repeats for the next operator. When the “^” operator is encountered, it again pops the top two operands from the stack, which are now “4” and “6”. The interpreter then performs the calculation 4 power 6, which returns 4096 and pushed back on the stack. At the end of the process, when the array is emptied, the stack will only contain a single element which is the result of the postfix expression, considering there are not invalid inputs. In this case, the stack being [4096] will be printed out in the console as the answer of the postfix expression.

Variables are also work similarly, their values are retrieved from the hash table and pushed onto the stack for operation. For example, if variable “A” has the value “2” stored in the hash table, having “A” in the postfix expression will have value “5” pushed onto the stack. This creates an interactive session where user can assign the value to a variable first then perform calculations with the variable.

Error handling is also implemented to manage the invalid inputs and print out the error message into the console to notify users. When users input an invalid operator such as “#”, the interpreter print out the error message into the console, notifying users about their invalid input. This error handling makes sure that the program does not terminate upon encountering an invalid operator, instead, it ignores the invalid operator and proceeds with the remaining tokens. For example, when the user input “3 4 5 # +”, with “#” being an invalid operator, the interpreter would print out an error for “#”, ignore that operator, and continue with the calculation. The input will now be look through by the interpreter as “3 4 5 +”, following the algorithm above, “4 5 +” would result in “9” and pushed onto the stack, resulting in the result, [9, 3]. The result contains more than one value because the “#” operations could not be performed due to the missing valid operators. This implementation ensures that user is informed of the error without crashing the program.

The interpreter also includes error handling for scenarios where users input more operator than operands. For example, “2 3 + * %”. The interpreter will perform calculation on the “2 3 +” but encounters an extra “*” and “%”. The interpreter will log and error message showing the extra operator and proceed with the calculation by ignoring the extra operator. In this case, the result, “5”, the extra operators “*”, “%”, as well as the list of supported operators will be printed out in the console. This implementation let users understand which operator is invalid.

Another error handling mechanism implemented involves managing input with extra spaces between tokens. The interpreter is designed to skip the extra spaces when checking the tokens. For example, if user inputs “2 3 +”, the extra spaces will just be skipped until it reaches another operator or operand. This implementation ensures that typing error would not affect the output from the interpreter.

Managing Variables using HASH TABLE

A HashTable class has been implemented for the interpreter to manage variables and their values. This class provides operation to modify the hash table such as hash, insert, search, and remove. The hash table is constructed as an array with 26 empty arrays (buckets) inside, considering there are 26 alphabetic characters, treating uppercase and lowercase letters as the same character. This is a separate chaining mechanism to prevent collision, which ensures multi key-value pairs can be stored at the same index if necessary.

The hash method computes an index for an entry in a hash table based on the key. This method first converts the input key (variable) into an uppercase letter to ensure that both uppercase and lowercase version of the same letter are treated as the same key. The built-in JavaScript method charCodeAt is used to get the ASCII value of the key. For example, the key “A” returns a value of 65, while “Z” returns “90”. By subtracting 65 from the ASCII value, an index is created that starts at 0. Since the valid keys are only from A to Z, the indices will range from “0” to “25”. This hashing algorithm guarantees that each key uniquely maps to an index within the hash table. The hash table created will be as follow:

The insert method adds or update a key and its respective value into the hash table. It starts off by converting the input key into uppercase, ensuring both uppercase and lowercase of the same letter are treated as the same key. Next, it calculates the index where the key and value will be stored using the hash method in the HashTable class mentioned previously. For instance, they key “C” has an ASCII value of 67. Subtracting 65 from this value returns 2, this means the key “C” and its value will be stored in the bucket with the index 2. The key “C” will be stored as the first element (index 0) in that bucket, while the value will be stored as the second element (index 1). If the key already exists in the bucket, such as inserting (“A”, 10) into a bucket that already has [“A”, 3] stored, this method updates the existing value of the key “A” into 10. If the key does not exist in the bucket, this method inserts the new key and value into the bucket. This ensures that each value of the key can be updated and used in the postfix expression interpreter.

The search method searches for the value of the input key from the hash table. It starts by converting the key into uppercase, then compute the index where the bucket should

be using the hash method in the HashTable class. Once identified, it checks the corresponding bucket in the hash table. Next, a for loop is created to iterate through the bucket to check if the first element (index 0) matches the input key. If a match is found, the method returns the value stored as the second element (index 1) of that bucket. If the key is not found, it returns undefined, indicating that the key has not been previously stored in the hash table. For example, hashing the key "C" returns the index 2, this search method checks bucket index 2. Suppose the bucket contains ["C", 5], since the key "C" matches the input key "C" at index 0 of the bucket, it then returns the value, 5.

The remove method deletes a key and the respective value from the hash table based on the input key. It starts of by converting the key into uppercase, then calculates the index where the pair is located using the hash method. Once identified, it checks the corresponding bucket in the hash table. Next, a for loop is used to iterate through the bucket and searches the first element (index 0) for the key that matches the input key. If a match is found, built-in splice method is used to remove them from the bucket, leaving it as an empty array. It then returns true, indicating that the key and value has been removed successfully. If no match is found after iterating through the bucket, it returns false. For example, if the key "C" is hashes and found to be at index 2, the remove method will check the bucket at index 2. Suppose it contains the entry ["C", 5], the method will remove this entry using splice and return true. If the remove method is used again with the same input key "C", it would return false because they have already been removed successfully.

When any of the insert, search, or remove main functions are called, they will use the hash table class methods above to complete their respective commands. Each function incorporates error handling mechanism to validate input key before proceeding. Both search and remove functions uses the same input validation. They use regular expression to ensure that the input key consists of alphabetic characters from A to Z only, so no other input such as numbers or symbols can be inputted. After executing the functions, they also print the result in the console, informing the user whether the search or remove function succeeded. The insert function requires 2 tokens as input. After the input is retrieved through the readline module, it splits the input into array using the split function. Then it uses regular expression to ensure that the first token is alphabetic character between A to Z, and isNaN method is then used to ensure the second token is a number. When the function takes in key as the input, it also uses the charAt function to extract only the first character of the key. If user entered "AAAA", it would only read it as process the first character "A" as the key.

Outside the hash table class, there is another function, showAllVar, that uses the hash table class methods as well. Its main purpose is to display all the variable with assigned values stored in the hash table. Initially, an isEmpty variable to set to true, assuming all the buckets in the hash table are empty. Next, it checks whether the hash table is empty

by iterating through the 26 buckets. If a non-empty bucket is found, the isEmpty will be set to false, indicating the hash table is not empty. It then breaks out of the for loop without having to continue looping through all the buckets because once a match is found, the hash table is guaranteed to not be empty. If all the buckets are empty, it prints a message to inform user that there are no variables that are currently set. If the hash table is not empty, and since each key and value pairs as inserted as array, a for loop is created to loop through each array in each of the buckets, then check the elements inside the array. When it encounters an empty bucket, the length of the bucket will be 0, so it skips over empty buckets. If a bucket is not empty, the length of the bucket will be larger than 0, which it will then print out the key and value of all the existing variables.

Section 3: Pseudocode for each original algorithm

Pseudocode

```
class STACK
        constructor
                stack <- []
        end constructor

        method PUSH(element)
                UNSHIFT(stack, element)
        end method

        method POP()
                return SHIFT(stack)
        end method

        method PEEK()
                return stack[0]
        end method

        method ISEMPTY()
                if LENGTH[stack] = 0 then
                        return true
                else
                        return false
                end if

                method PRINT()
                        return stack
                end method
```

```
end class

class HASHTABLE
        constructor(size)
                hash_table <- []
                for 0 <= i < size do
                        hash_table[i] <- []
                end for
        end constructor

        method HASH(key)
                key <- UPPERCASE[key]
                key <- CHARCODEAT[key] - 65
        end method

        method INSERT(key, value)
                index <- HASH(key)
                bucket <- hash_table[index]
                for 0 <= i < LENGTH[bucket] do
                        if bucket[i][0] = key then
                                bucket[i][1] <- value
                                return
                        end if
                end for
                PUSH([key, value], bucket)
        end method

        method SEARCH(key)
                index <- HASH(key)
                bucket <- hash_table[index]
                for 0 <= i < LENGTH[bucket] do
                        if bucket[i][0] = key then
                                return bucket[i][1]
                        end if
                end for
                return undefined
        end method

        method REMOVE(key)
                index <- HASH(key)
                bucket <- hash_table[index]
                for 0 <= i < LENGTH[bucket] do
                        if bucket[i][0] = key then
                                SPLICE[i, 1] from bucket
                                return true
                        end if
                end for
```

```
                return false
            end method
end class

function MAIN()
  PRINT question
  input <- READLINE
  input <- UPPERCASE[input]
  input <- TRIM[input]
  if input = "1" then
    POSTFIXCALC()
  else if input = "2" then
    SEARCHVAR()
  else if input = "3" then
    INSERTVAR()
  else if input = "4" then
    REMOVEVAR()
  else if input = "5" then
    SHOWALLVAR()
  else if input = "RETURN" then
    PRINT "Already in the main menu"
  else
    PRINT "Invalid input"
  end if
  MAIN()
end function

function POSTFIXCALC()
  PRINT "Enter PostFix++ Expression"
  input <- READLINE
  input_array <- SPLIT input BY " "
  PRINT "Steps breakdown:"
  result <- POSTFIX(input_array)
  PRINT "Final Output:", result
  POSTFIXCALC()
end function

function SEARCHVAR(input)
  PRINT "Enter variable name (A-Z)"
  input <- READLINE
  input <- SPLIT input BY " "
  if input = "RETURN" then
    MAIN()
  end if

  if input[0] MATCHES a-z OR A-Z then
    variable <- UPPERCASE[input[0]]
```

```
      value <- SEARCH(variable) from hashTable
      if value != undefined then
        PRINT "Variable '", variable, "' = ", value
      else
        PRINT "Variable '", variable, "' not found"
      end if
    else
      PRINT "Invalid input"
    end if
    SEARCHVAR()
end function

function INSERTVAR()
    PRINT "Enter variable name (A-Z) and the value"
    input <- READLINE
    if input = "RETURN" then
      MAIN()
    end if

    input <- SPLIT input BY " "
    if input[0] MATCHES a-z OR A-Z AND NOTANUMBER[input[1]] then
      variable <- UPPERCASE[input[0]]
      variable <- CHARAT[input[0], 0]
      value <- PARSEFLOAT[input[1]]
      INSERT(variable, value) from hashTable
      PRINT "Variable '", variable, "' set to ", value
    else if !NOTANUMBER[input[0]] then
      PRINT "Invalid variable name"

    else if NOTANUMBER[input[1]] then
        PRINT "Invalid value"
    end if
    INSERTVAR()
end function

function REMOVEVAR()
    PRINT "Enter variable name (A-Z)"
    input <- READLINE
    if input = "RETURN" then
      MAIN()
    end if

    input <- SPLIT input BY " "
    if input[0] MATCHES a-z OR A-Z then
      variable <- UPPERCASE[input[0]]
      variable <- CHARAT[input[0], 0]
      removed <- REMOVE(variable) from hashTable
```

```
      if removed then
        PRINT "Variable '", variable, "' removed"
      else
        PRINT "Variable '", variable, "' not found"
      end if
    else
      PRINT "Invalid input"
    end if
    REMOVEVAR()
end function

function SHOWALLVAR()
    isEmpty <- true
    for 0 <= i < LENGTH[hash_table] do
      if LENGTH[hash_table[i]] > 0 then
        isEmpty <- false
        break
      end if
    end for

    if isEmpty then
      PRINT "No set variables currently"
    else
      for 0 <= i < LENGTH[hash_table] do
        for 0 <= j < LENGTH[hash_table[i]] do
          if hash_table[i][j][0] != "RETURN" then
            PRINT "Variable '", hash_table[i][j][0], "' = ", hash_table[i][j][1]
          end if
        end for
      end for
    end if
    PRINT "Enter 'Return' to go back to main menu"
end function

function POSTFIX(array)
    stack <- new STACK from CLASS
    for 0 <= i < LENGTH[array] do
      element <- array[i]
      if !NOTANUMBER[element] and !EMPTY then
        PUSH(stack, PARSEFLOAT[element])
      else if MATCH[element] with a-z OR A-Z then
        element <- UPPERCASE[element]
        PUSH(stack, element)
      else
        element <- UPPERCASE[element]
        if element = ""
          break
```

```
        else if element = "+" OR "-" OR "*" OR "/" OR "%" OR "^" then

            if LENGTH[stack] < 2 then
               PRINT "Not enough operands for operation"
               break
            end if
            x <- POP(stack)
            y <- POP(stack)

            x <- GETVARVALUE(x)
            y <- GETVARVALUE(y)

            if element = "+" then
               result <-  x + y
            else if element = "-" then
               result <-  x - y
            else if element = "*" then
               result <-  x * y
            else if element = "/" then
               if y = 0 then
                  PRINT "Cannot divide by zero"
                  break
               end if
               result <-  x / y
            else if element = "%" then
               result <-  x % y
            else if element = "^" then
               result <-  x ^ y
            end if

        else if element = "SIN" OR "COS" OR "TAN" OR "SQRT" OR "CEIL" OR "FLOOR"
OR "ABS" OR "ROUND" OR "LOG" then
            if LENGTH[stack] < 1 then
               PRINT "Not enough operands for operation"
               break
            end if

            x <- POP(stack)
            x <- GETVARVALUE(x)

            if element = "SIN" then
               result <-  MATH.SIN[x]
               PRINT result
            else if element = "COS" then
               result <-  MATH.COS[x]
               PRINT result
            else if element = "TAN" then
```

```
      result <-  MATH.TAN[x]
      PRINT result
    else if element = "SQRT" then
      result <-  MATH.SQRT[x]
      PRINT result
    else if element = "CEIL" then
      result <-  MATH.CEIL[x]
      PRINT result
    else if element = "FLOOR" then
      result <-  MATH.FLOOR[x]
      PRINT result
    else if element = "ABS" then
      result <-  MATH.ABS[x]
      PRINT result
    else if element = "ROUND" then
      result <-  MATH.ROUND[x]
      PRINT result
    else if element = "LOG" then
      result <-  MATH.LOG[x]
      PRINT result
    end if

    PUSH(stack, result)


else if element = "=" then
    x <- POP(stack)
    y <- POP(stack)
    if !NOTANUMBER[x] AND !NOTANUMBER[y] then
      PRINT "Cannot assign a value to another number"
    end if

    if NOTANUMBER[x] then
      INSERT(y, SEARCH(x) from hashTable) into hashTable
      PRINT "Variable [", y, "] set to", SEARCH(x) from hashTable
    else
      INSERT(y, PARSEFLOAT[x]) into hashTable
      PRINT "Variable [", y, "] set to", x
    end if

else if element = "RETURN"
    main();
    break;

else
    PRINT "Invalid operator:", element
end if
```

```
    end if
  end for
  return PRINT(stack)
end function

function GETVARVALUE(element)
  if NOTANUMBER[element] then
    return SEARCH(element) from hashTable
  else
    return PARSEFLOAT[element]
  end if
end function
```

Section 4: A list of the data structures in the program

Stack

I have chosen a stack data structure with an array-based approach to interpret postfix notations, due to its Last In, First Out (LIFO) principle. In postfix notation, the two rightmost operands beside an operator are used first in the interpretation. For example, in the expression “1 2 3 + -”, the operand “2” and “3” will be used to perform calculation with the leftmost operator “+” which returns “5”. The result is then used with the operand “1” and the “-” operator. This aligns with the LIFO behaviour of stacks, making stack my first choice for the interpreter.

To adapt the stack to the requirement, I have modified the stack operations to ensure the array-based stack will have the leftmost element in the array representing the top of the stack. Instead of using the default push and pop operations, which add and remove elements from the last element of the array, I used shift and unshift operations to add and remove elements from the beginning of the array. Additionally, I also modified the peek function to return the first element of the array, as this represents the top of the stack in this structure.

A stack ensures that the rightmost operand is always at the top of the stack due it being the last one to be pushed. When the two operands from the top if the stack are removed, their values will be stored in variables to be use in the calculations with the valid operator. Once all operands are pushed onto the stack and the array is left with operators, the interpreter will check if the operator is valid. When an operator is recognised, the top two elements of the stack are removed, have their values stored in variables, and the calculation is performed then pushed back onto the stack. This shows how the stack nature aligns perfectly with the requirements of a postfix interpreter.

Comparing the stack with other data structures, such as queue, it shows that stack is more suitable for this task. In contrast with the stack LIFO principle, queue operates on First In, First Out (FIFO) principle, which could make the calculations more complicated than it should be. The postfix interpreter calculates the two rightmost operands with the leftmost operator. However, due to the FIFO principle of a queue, the first element added to the queue will be the first one to be removed. For example, in the expression "1 2 3 + *", the first two element in the queue "1" and "2" would be used with the "+" operator first. This does not align with how a postfix interpreter works, as "2" and "3" should be the calculated first. So, I have chosen to use stack over queue with its LIFO principle to calculate postfix expression.

Hash Table

I have chosen a hash table data structure to store and retrieve variables and their values due to its efficiency. The hash table in this case have the time complexity of O(1) for insertions, search, and deletion, due to a good hash function being implemented that ensures no collision will appear.

To implement the hash table, I create an array-based structure with a fixed size, where each index points to a bucket that holds the key and value as a pair. The hash function is designed to compute an index for each input key, ensuring each key point towards a unique bucket. I used the ASCII value of the variable name to compute the index, which ensures each unique key will only have a bucket, preventing collision.

When insertion, search or deletion operations is called, the hash function will compute the index that points specifically to the bucket they are supposed to be stored in. During insertion, the key and value will be added into the bucket. If a pair with the same key already exists, its value will be overwritten. When the search or remove function called, hash function again finds the target bucket and the bucket. The bucket is then checked to either return the value or remove the key and value from the bucket.

For example, in the expression "A 5 =", the hash function converts "A" to the index "0" by taking its ASCII value and subtracting 65 from it. Then the key and value are stored in a bucket with that index. If the users search for the value of "A", the hash function computes the index for "A" again, and the value "5" is returned.

Comparing hash table with other data structures, such as linked list, it clearly shows that hash table is more suitable for the task. When using a hash table, the input key is hashed to find an index that directly points to the bucket where the key and value are stored. Unlike linked list that must check each element sequentially to find a matching key, hash table allows, insertion, search, or delete operations to efficiently access the correct bucket without having to iterate through the entire table. For instance, when

searching for the key “Z” in a hash table, the hash function computes the exact index where “Z” is stored, allowing us to access its value immediately. In contrast, if “Z” element was the last element in a linked list, it would require checking through every element before itself. This cause a greater time consumption, therefore, hash table is the better choice when storing and managing variables.

Section 5: Commented source code and link to video

Link to demo video

https://youtu.be/ZIsGx4A7cm8

Commented source code

```
class Stack {
 // Create a stack with an empty array
 constructor() {
  this.stack = [];
 }

 // A method to push an element to the stack using unshift to add it to the front of the
array
 push(element) {
  this.stack.unshift(element);
 }

 // A method to pop an element from the stack using shift to remove the first element of
the array
 pop() {
  return this.stack.shift();
 }

 // A method to get the top element of the stack
 peek() {
  return this.stack[0];
 }

 // A method to check if the stack is empty
 isEmpty() {
  if (this.stack.length == 0) {
   return true;
  } else {
   return false;
```

```
  }
 }

 // A method to print the stack
 print() {
  return this.stack;
 }
}

class HashTable {
 constructor(size) {
  this.size = size;
  // Create a hash table with a size of 26 for the 26 alphabets and fill it with empty arrays
  this.hash_table = [];
  for (let i = 0; i < size; i++) {
   this.hash_table[i] = [];
  }
 }

 // Hash function to get the index of the key
 // ASCII value of A is 65, so we -65 to get the index of A starting from 0
 hash(key){
  return key.toUpperCase().charCodeAt(0) - 65;
 }

 // A function to set the value of the key in the hash table
 insert(key, value){
  // Get the index of the key
  const index = this.hash(key);
  // Get the bucket of the index
  const bucket = this.hash_table[index];

  // Loop through the bucket
  for(var i = 0; i < bucket.length; i++) {
   // If a matching key is the same as the input key
   if(bucket[i][0] == key) {
    // Set the value to the input value
    bucket[i][1] = value;
    return;
   }
  }
  // Push the key and value to the bucket
  bucket.push([key, value]);
 }

 // A function to get the value of the key in the hash table
 search(key){
```

```
    // Get the index of the key
    const index = this.hash(key);
    // Get the bucket of the index
    const bucket = this.hash_table[index];

    // Loop through the bucket
    for(var i = 0; i < bucket.length; i++) {
      // If a matching key is the same as the input key
      if(bucket[i][0] == key) {
        // Return the value of the key
        return bucket[i][1];
      }
    }
    // Return undefined if the key is not in the hash table
    return undefined;
  }

  // A function to delete the key in the hash table
  remove(key){
    // Get the index of the key
    const index = this.hash(key);
    // Get the bucket of the index
    const bucket = this.hash_table[index];

    // Loop through the bucket
    for(var i = 0; i < bucket.length; i++) {
      // If a matching key is the same as the input key
      if(bucket[i][0] == key) {
        // Remove the key and value from the bucket
        bucket.splice(i, 1);
        // Return true if the key is removed
        return true;
      }
    }
    // Return false if the key is not found
    return false;
  }
}

// Requiring Readline Module to retrieve user input
// https://nodejs.org/en/learn/command-line/accept-input-from-the-command-line-in-nodejs
const readline = require("node:readline");

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
```

```
});

// Main function to run the program
function main() {
  // Prompt for user input
  rl.question("--------------\n[ Algorithm and Data Structure II ]\n--------------\nPlease enter one of the following: \n1 - PostFix++ Calculator\n2 - Search for the value of a variable.\n3 - Insert value to a variable.\n4 - Remove a variable and its value.\n5 - Show all variables and their value\nReturn - Return to main menu\n--------------\nYour Input: ", (input) => {

    // Convert the user input to uppercase if it is a string
    input = input.toUpperCase();
    input = input.trim();

    // Check the user input and run the respective function
    switch (input){
      case "1":
        postFixCalc();
        break;

      case "2":
        searchVar();
        break;

      case "3":
        insertVar();
        break;

      case "4":
        removeVar();
        break;

      case "5":
        showAllVar();
        break;

      // If the user input is RETURN
      case "RETURN":
        console.log(">> You are already in the main menu.\n");
        break;

      // If the user input is not in the list
      default:
        console.log(">> Error: Invalid input. Please enter a valid input.\n");
        break;
    }
```

```
    // Run the main function again to allow for more user input
    main();
  });
}

// Run the main function to start
main();

// Creating a new hash table with a size of 26 for the 26 alphabets
const hashTable = new HashTable(26);

function postFixCalc() {

  rl.question("\n--------------\n[ PostFix++ Calculator ]\n--------------\nPlease Enter a PostFix++ Expression.\nFormat: <operand> <operator>\nExample: 3 4 5 + *\nSupported operations: +, -, *, /, %, ^, =, SIN, COS, TAN, SQRT, CEIL, FLOOR, ABS, ROUND, LOG.\nReturn - Return to main menu\n\nYour Input: ", (input_array) => {

    // Convert the user input to an array when there is a space
    input_array = input_array.split(" ");
    console.log("Steps breakdown: ");
    // Show the output of the PostFix function in the console using the user input
    console.log("Final Output:", postFix(input_array));

    // Run the postFixCalc function again to allow for more user input
    postFixCalc();
  });
}

// A function to search for the value of a variable
function searchVar() {

  rl.question("\n--------------\n[ Search for the value of a variable. ]\n--------------\nPlease enter a variable name (A - Z).\nFormat: <variable>\nExample: a\nReturn - Return to main menu\n\nYour Input: ", (input) => {

    // Return to the main menu if the user input is "RETURN"
    returnToMain(input);

    input = input.split(" ");

    if(input[0].match(/[a-zA-Z]/)) {
      // Convert the first element of the user input array into uppercase
      const variable = input[0].toUpperCase().charAt(0);
      // Get the value of the variable from the hash table
      const value = hashTable.search(variable);
```

```
    // Log the value of the variable to the console if it is found
    if (value !== undefined) {
      console.log(">> Variable '" + variable + "' =", value + ".");
    } else {
      console.log(">> Variable '" + variable + "' not found.");
    }
  } else {
    console.log(">> Error: Invalid input. Please enter a valid variable name (A - Z).");
  }

  // Run the searchVar function again to allow for more user input
  searchVar();
 });
}

// A function to set a value to a variable
function insertVar() {
 rl.question("\n--------------\n[ Insert value to a variable. ]\n--------------\nPlease enter a
variable name (A - Z) and the value.\nFormat: <variable> <value>\nExample: a
5\nReturn - Return to main menu\n\nYour Input: ", (input) => {

  // Return to the main menu if the user input is "RETURN"
  returnToMain(input);

  // Convert the user input to an array when there is a space
  input = input.split(" ");

  if (input[0].match(/^[a-zA-Z]$/) && !isNaN(input[1])) {

    // Convert the first element of the user input array into uppercase as it is a variable
    const variable = input[0].toUpperCase().charAt(0);
    // Get the value of the variable from the user input array
    const value = parseFloat(input[1]);

    // Set the value of the variable to the value
    hashTable.insert(variable, value);
    console.log(">> Variable '" + variable + "' has been set to", value);
  } else if (!(input[0].match(/^[a-zA-Z]$/))) {
    console.log(">> Error: Invalid input. Please enter a valid variable name (A - Z).");
  } else if (isNaN(input[1])) {
    console.log(">> Error: Invalid input. Please enter a valid float value.");
  }

  // Run the insertVar function again to allow for more user input
  insertVar();
 });
}
```

```
// A function to remove a variable and its value
function removeVar() {
  rl.question("\n--------------\n[ Remove a variable and its value. ]\n--------------\nPlease enter a variable name (A - Z).\nFormat: <variable>\nExample: a\nReturn - Return to main menu\n\nYour Input: ", (input) => {

    // Return to the main menu if the user input is "RETURN"
    returnToMain(input);

    input = input.split(" ");

    if(input[0].match(/[a-zA-Z]/)) {
      // Convert the first element of the user input array into uppercase as it is a variable
      const variable = input[0].toUpperCase().charAt(0);
      // Remove the variable from the hash table
      const remove = hashTable.remove(variable);

      // Return a message to the console if the variable is removed or not found
      if (remove) {
        console.log(">> Variable: '" + variable, "' has been removed.");
      } else {
        console.log(">> Variable '" + variable + "'not found.");
      }
    } else {
      console.log(">> Error: Invalid input. Please enter a valid variable name (A - Z).");
    }
    // Run the removeVar function again to allow for more user input
    removeVar();
  });
}

// A function to show all variables and their values
function showAllVar() {
  // Check if the hash table is empty
  let isEmpty = true;
  // Loop through the hash table to check if all the buckets are empty
  for (var i = 0; i < hashTable.hash_table.length; i++) {
    // If the length of the bucket is more than 0
    if (hashTable.hash_table[i].length > 0) {
      // Set isEmpty to false
      isEmpty = false;
      // Break out of the loop when a non-empty bucket is found
      break;
    }
  }
```

```
  // Log a message if the hash table is empty
  if (isEmpty) {
    console.log(">> There are currently no set variables.\n");
  } else {
    // Loop through the hash table to display all the variables and their values
    for (var i = 0; i < hashTable.hash_table.length; i++) {
      for (var j = 0; j < hashTable.hash_table[i].length; j++) {
        // Skip the 'RETURN' variable
        if (hashTable.hash_table[i][j][0] != "RETURN") {
          // Log the variable and its value to the console
          console.log(">> Variable '" + hashTable.hash_table[i][j][0] + "' = ",
hashTable.hash_table[i][j][1]);
        }
      }
    }
    console.log("\n");
  }
}

// A function to return users back to the main menu when they type "RETURN"
function returnToMain(input) {
  // Convert the user input to uppercase
  input = input.toUpperCase();
  // Check if the user input is "RETURN"
  if(input == "RETURN") {
    console.log(">> Returning to main menu.");
    // Run the main function to call the main menu
    main();
  }
}

// A function to calculate the PostFix expression
function postFix(array) {
  // Create a new stack
  var stack = new Stack();

  for (var i = 0; i < array.length; i++) {
    // Get the element at the current index
    let element = array[i];
    let x, y;

    // Check if the element is a number
    if (!isNaN(element) && element !== "") {
      // Push the element to the stack if it is a number
      stack.push(parseFloat(element));

    // Check if the element is an alphabet
```

```
} else if (element.match(/^[a-zA-Z]$/)) {
   // Convert the alphabet variable to uppercase
   element = element.toUpperCase();

   // Push the alphabet variable to the stack
   stack.push(element);
} else {
  // Convert the element to uppercase
  element = element.toUpperCase();

  // Check if the element is an operator
  switch (element) {
    case "":
     break;

   // If the element is one of these operators +, -, *, /, %, ^
   case "+":
   case "-":
   case "*":
   case "/":
   case "%":
   case "^":
    // Log an error if there are not enough operands for the operator
    if (stack.print().length < 2) {
      console.log(">> Error: Not enough operands for the operator '" + element + "'");
      break;
    }

    // Pop top 2 operand from the top of the stack
    x = stack.pop();
    y = stack.pop();

    // Get the value if it is a variable
    x = getVarValue(x);
    y = getVarValue(y);

    // Perform operation based on the operator
    switch (element) {
      // Perform addition if the operator is +
      case "+":
        result = x + y;
        break;

      // Perform subtraction if the operator is -
      case "-":
        result = x - y;
        break;
```

```
  // Perform multiplication if the operator is *
  case "*":
    result = x * y;
    break;

  // Perform division if the operator is /
  case "/":
    if(y == 0) {
      console.log("\n>> Error: Cannot divide by 0.\n");
      return;
    }
    result = x / y;
    break;

  // Perform modulo if the operator is %
  case "%":
    result = x % y;
    break;

  // Perform exponentiation if the operator is ^
  case "^":
    result = x ** y;
    break;
 }

 // Log the result
 console.log(x, element, y, "=", result);
 // Push the result back to the stack
 stack.push(result);
 break;

// Unary operators: SIN, COS, TAN
case "SIN":
case "COS":
case "TAN":
case "SQRT":
case "CEIL":
case "FLOOR":
case "ABS":
case "ROUND":
case "LOG":
 // Log an error if there are not enough operands for the operator
 if (stack.print().length < 1) {
   console.log(">> Error: Not enough operands for the operator '" + element + "'");
   break;
 }
```

```
// Pop operand from the stack
x = stack.pop();

// Get the value if it is a variable
x = getVarValue(x);

switch (element) {
 // Perform sine if the operator is SIN
 case "SIN":
  result = Math.sin(x);
  console.log("sin(" + x + "radians) =", result);
  break;

 // Perform cosine if the operator is COS
 case "COS":
  result = Math.cos(x);
  console.log("cos(" + x + "radians) =", result);
  break;

 // Perform tangent if the operator is TAN
 case "TAN":
  result = Math.tan(x);
  console.log("tan(" + x + "radians) =", result);
  break;

 // Perform square root if the operator is SQRT
 case "SQRT":
  result = Math.sqrt(x);
  console.log("sqrt(" + x + ") =", result);
  break;

 // Return ceiling if the operator is CEIL
 case "CEIL":
  result = Math.ceil(x);
  console.log("ceil(" + x + ") =", result);
  break;

 // Return floor if the operator is FLOOR
 case "FLOOR":
  result = Math.floor(x);
  console.log("floor(" + x + ") =", result);
  break;

 // Return absolute value if the operator is ABS
 case "ABS":
  result = Math.abs(x);
```

```
          console.log("abs(" + x + ") =", result);
          break;

        // Return rounded value if the operator is ROUND
        case "ROUND":
          result = Math.round(x);
          console.log("round(" + x + ") =", result);
          break;

        case "LOG":
          result = Math.log(x);
          console.log("ln(" + x + ") =", result);
          break;
      }

      // Push the result back to the stack
      stack.push(result);
      break;

    // If the element is =
    case "=":
      // Pop two element from the stack
      x = stack.pop();
      y = stack.pop();

      if(!isNaN(x) && !isNaN(y)) {
        console.log(">> Error: Cannot assign value to a number. Please assign a value to a
variable.");
        break;
      }

      // Check if the element is a variable
      if (isNaN(x)) {
        // Set the value of the variable to the value of the other variable
        hashTable.insert(y, hashTable.search(x));
        console.log(">> Variable [", y.toUpperCase(),"] has been set to",
hashTable.search(x) + ".");
      } else {
        // Set the value of the variable to the number as a float
        hashTable.insert(y, parseFloat(x));
        console.log(">> Variable [", y.toUpperCase(),"] has been set to", x + ".");
      }
      break;

    case "RETURN":
      main();
      break;
```

```
      // If the element is not a operator mentioned above
      default:
        // Log an error message to the console
        console.log(">> Error: The input '" + element + "' is invalid and will be ignored.
Please only use the following operators: +, -, *, /, %, ^, =, SIN, COS, TAN, SQRT, CEIL,
FLOOR, ABS, ROUND, LOG.");
        // Break out of the switch statement
        break;
    }
   }
  }
  // Return the top element of the stack
  return stack.print();
}

// A function to check if the element is a variable or a number
function getVarValue(element) {
  // Check if the element is a number
  if (isNaN(element)) {
    // Return the value of the variable if it is a variable
    // return variables[element];
    return hashTable.search(element);
  } else {
    // Convert the element to a float if it is a number and return it
    return parseFloat(element);
  }
}
```

Section 6: Defects of the program and their remedies

Nested switch-case structure on time complexity

To enhance the program's readability, I used switch-case statements instead of the usual if-else statements to handle calculations in the PostFix function. A nested switch-case structure was implemented to avoid code repetition across the different mathematical operations. The outer switch case checks whether the operator is a basic or advanced mathematical function, and then the inner switch case handles their respective calculations. For example, when the input is an operator like “^”, the outer switch case first checks if it's a basic or advanced mathematical function, once determined, then the inner switch case performs the operation and returns the result. This keeps the code organized and reduces code repeatability. However, this nested switch-case structure comes with a higher time complexity. Each nested switch adds

an additional comparison, which can increase the overall time complexity of the function. While the increase in time complexity might not be significant for small input sizes, it might show the difference when it is a large input.

To remedy this defect, a possible remedy is to revert the nested switch-case structure back into a single switch-case structure. This can make the time complexity faster by reducing the number of comparisons performed. However, this would result in more code repetition, making the program harder to read.

Limited operation support

The current interpreter supports simple mathematical function like addition, subtraction, multiplication, division, modulus, and exponentiation. Additionally, advanced mathematical functions like sine, cosine, tangent, square root, ceiling, floor, absolute value, rounding, and logarithm are also included. But other operations such as factorial, permutation, and combination, are not supported. This restricts the users that need more comprehensive mathematical functionality.

To fix this defect, additional mathematical functions available in JavaScript's Math library could be integrated into the interpreter function, which would expand the calculator's utility to include complex mathematical functions such as “asinh" which returns the hyperbolic arcsine of a number. Additionally, static properties such as “Math.PI” could be added as an input, allowing users to use Pi value as an operand without typing out the value. This would make the calculator more user-friendly.

No operator precedence support

One of the main defects of this program is the lack of support for PEMDAS and BDOMAS operator precedence rules. In mathematical expressions, operations like multiplication and division have a higher precedence than addition and subtraction. This means expressions should be calculated according to their precedence order to ensure accuracy. This postfix interpreter only calculates the expression based on the input order, calculating each operand and operator based on the sequence they are entered. For example, the expression “3 4 5 + *” is being treated as “(5 + 4) * 3” due to its input order even though no parentheses were inputted in the expression. According to PEMDAS/BODMAS, multiplication should be calculated before addition if there are no parentheses in the expression. In this case, “5 + 4 * 3”, “4 * 3" will be calculated first before being added by 5.

To remedy this defect, I could enhance the interpreter to support parentheses as an input within expressions. This would allow users to group operations, overriding the

default precedence rules. The interpreter could group operations and rearrange them according to the PEMDAS/BODMAS rules before performing any calculations.

Ambiguity of operator expectations

Another defect of this program is the ambiguity regarding the interpretation of the operand and operators. In mathematics, expressions like “ab”, “2a” or "2sin60" are interpreted as “a * b” and “2 * sin60” respectively. However, this program requires the explicit operator “*” to indicate multiplication, which may not align with users' expectations and leads to inaccurate results being returned.

A potential remedy is to modify the postfix interpreter to automatically compute multiplication between operands when no other operators are provided. However, the risk of unintended multiplication also rises. Alternatively, it could just prompt users for extra input when facing this issue.