Created by: Bryan Loo Chun Wai

Algorithms and Data Structure 2 Report

Table of contents

Section 1 - Design

Introduction

In the study of computer science, expression evaluation is a crucial concept with broad applications in areas such as compilers, calculators, and numerous other software systems. This project that I was tasked with focuses on the development of a postfix interpreter, an essential tool for retrieving user input and evaluating postfix expressions. Postfix is also known as reverse polish notation.

1.1) Difference between postfix and infix notations

Infix Notation

Infix notation example:

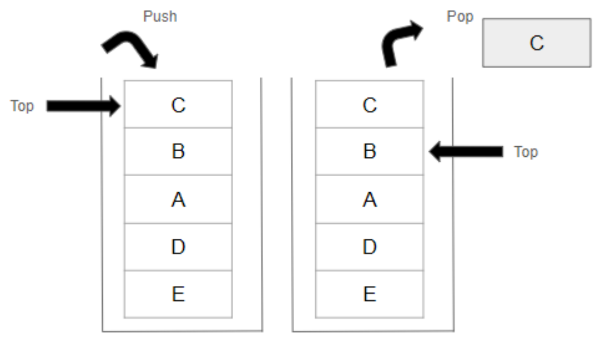
A = 1 B = 1 A + B

Explanation:

Unlike the regular notation known as infix notation, which is the most commonly used arithmetic notation that is used in everyday calculations and requires the knowledge of operator positioning and parenthesis, postfix notation presents a streamlined route to express and compute simple mathematical expressions.

Postfix Notation

Stack data structure visualisation:



Postfix notation example:

A 1 = B 1 = A B +

The primary objective of this project is to create an efficient postfix interpreter that has the ability of parsing and evaluating postfix expressions. The interpreter that has been created in this project uses a stack based approach, where the Last In, First Out principle is practised. In this principle, operands are pushed into a stack and operators subsequently pop operands for evaluation, thus returning the result back into the stack. This method of storing operands for a postfix interpreter is an efficient and structured method to ensure reliability and maintainability.

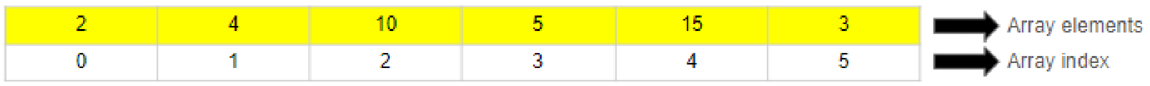
This report will detail the design and implementations of a postfix interpreter. Some areas that will be covered in this report will include algorithmic foundations, data structure considerations, and the usage of arithmetic operators. The report will also cover areas such as the potential challenges encountered during the development process and the solutions that could overcome these challenges. Through this project, I aim to demonstrate the practical application data structures and algorithms in building a postfix interpreter and apply the concepts that I have learnt in the classroom to ensure efficient and fast algorithm execution.

1.2) Data structures

What data structures are good for a Postfix interpreter?
For this implementation of the Postfix++ interpreter I will be discussing about 5 data structures that could be applied but of these 5 data structures, 2 data structures that may be more appropriate for this project. The 5 data structures include array, dynamic array, stack, queue, binary tree. These 5 data structures work efficiently in storing multiple elements of data in a structured manner but each of these data structures has its own limitations. I will provide a detailed explanation with diagram representations and justification why or why not each data structure might be a good fit for the interpreter program.

1.2.1) Array

Diagram 1.1



What does the diagram show?
In the above diagram, it shows a simple array data structure representation. In the first row it represents the array elements. The array elements in this array are 2, 4, 10, 5, 15, 3. Array elements are the values that are stored in the array. The second row represents the array index as shown using the arrows on the right. The index array is a numerical representation of where the element is in the array. For example, element 2 is at array index 0. This means that the value 2 is at the very first position in the array representation.

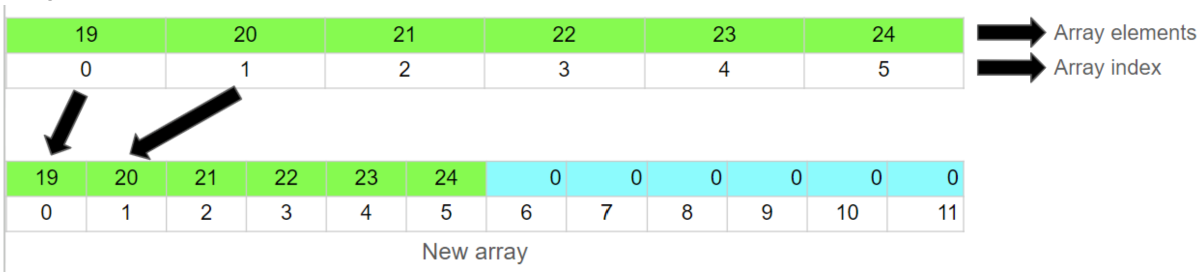
What can the array data structure offer?
The array data structure offers a simple and effective way to store a list of elements. This means that the array is able to store variables such as integers, doubles, strings. An example usage of an array can be if you need an efficient way to access data by an index or you want to know the size of the list of data.

Is an array a good fit for a postfix interpreter?
This data structure can be used in the implementation of the postfix program implementation as it provides an efficient index based access to elements, this means that it makes it easier to access any element in the array. Using this method the program is able to find the index in the array and can access the value in an instant.

Limitations
However this has limitations. This data structure is more suitable for fixed size data and used for storage of variables such as basic integers, doubles, and strings. This means that it will be inefficient to use memory if the array is too large or if there are too many values inside of the array the array might be too small. This data structure is also not useful if there are insertions and deletion operations as this involves moving elements around. Thus this will not be a good fit for the postfix program implementation.

1.2.2) Dynamic array

Diagram 1.2



What does the diagram show?
The above diagram shows a dynamic array. The first array at the top has 6 elements, 19, 20, 21, 22, 23, 24. Below the elements are the array indexes as labelled on the right. A dynamic array grows automatically when a value is inserted and there is no more space left for the new item. In most cases, the dynamic array will then grow double in size. As you can see in the diagram that a new array is created that is double the size of the old array of elements.

What can the dynamic array data structure offer?
As shown in the diagram, the blue coloured part of the new array is the dynamically added space from the original array. The dynamic array data structure is an efficient way to store a list of elements as it is able to grow and shrink dynamically. An example usage of this is when you require a data structure that is similar to an array with dynamic resizing.

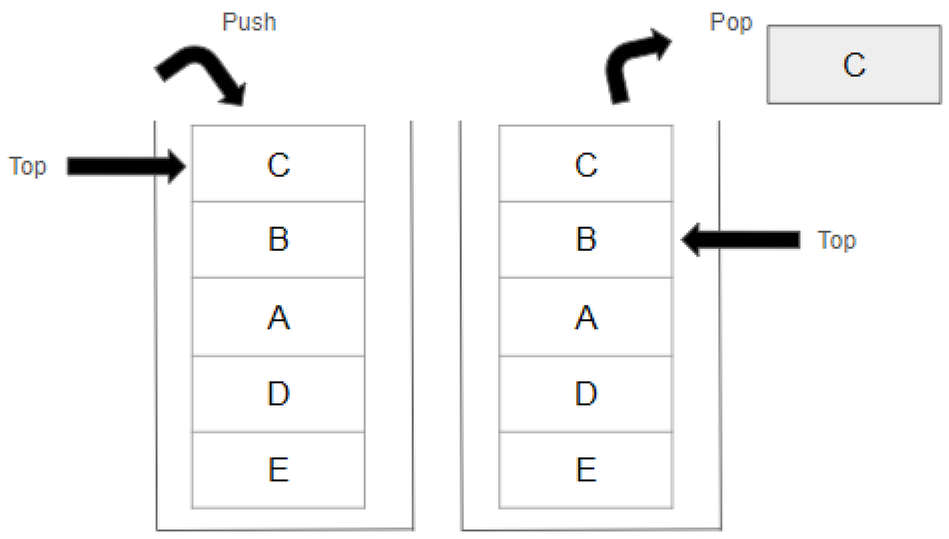
Is a dynamic array a good fit for a postfix interpreter?
This data structure can be considered as a suitable data structure to be used in the postfix program implementation as it provides a dynamic approach to store values. This data structure is very much similar to arrays but with an added feature of dynamic resizing. This data structure benefits the application by allowing for the size of the array to change based on the program execution.

Limitations
However this data structure has its limitations as resizing can drastically affect the time complexity of executing the algorithm. Even though it offers flexibility in size, in some cases it might lead to performance issues. For example, if the array is required to resize operations often.

With time complexity as a major consideration in the study of algorithms, this may not affect smaller programs such as a postfix evaluator but in larger production level programs this can surface as a huge issue especially when there are other users that are using your program.

1.2.3) Stack

Diagram 1.3



What does the diagram show?
In the diagram above it shows a stack data structure representation. On the left of the diagram, the stack contains 5 characters, C, B, A, D, E in order. Using the concept of a stack we now know that E was the first character pushed into the stack and C was the last character pushed into the stack.

In a stack it follows the last in, first out principle as seen in the diagram. On the left stack the letter C is pushed into the stack and becomes the top of the stack. So following the principle, if the stack were popped then C will be popped and the letter B will become the top of the stack. Using this principle, the element that was put in first will come out first.

Thus, on the right stack. After the C character has been popped, the new stack now contains the characters, B, A, D, E in order.

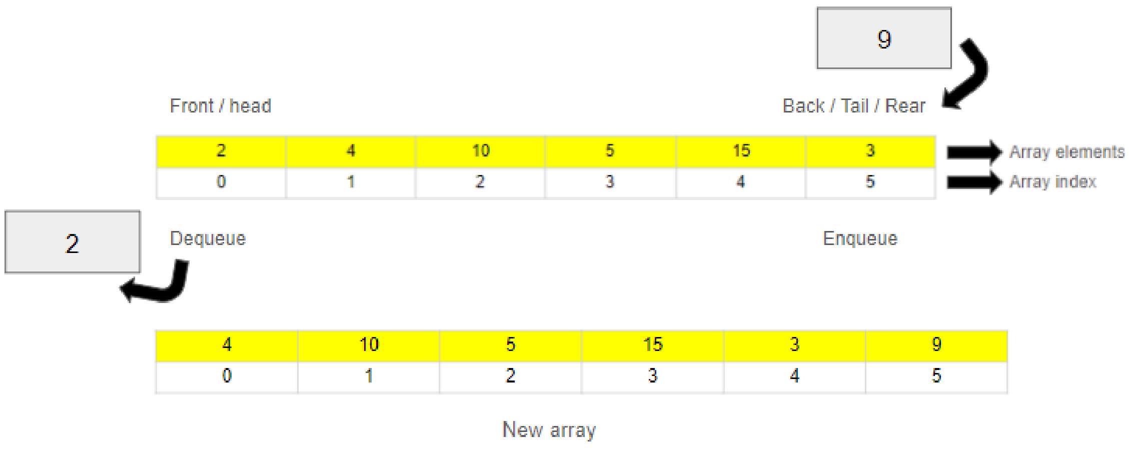
What can the stack data structure offer?
The stack data structure is an interesting data structure that follows the Last in, First out principle when it is required to store a collection of elements. An example usage of this data structure is to maintain a history of operations or if you require an order of operations in a reversed order.

Is a stack a good fit for a postfix interpreter?
This data structure is suitable for maintaining the order of operations such as in this case of a postfix program. This means that with this data structure, it provides a quick and efficient way of storing operators or variables and producing a result of the execution.

Limitations
However this data structure has its limitations. Stacks can only access the top element efficiently, which will affect the flexibility of information that is stored in this particular data type. Thus operations are limited to the top of the stack, and in the case that any request to access, insert or modify values in this data type will not be efficient as the last in, first out principle is applied to store values.

1.2.4) Queue

Diagram 1.4



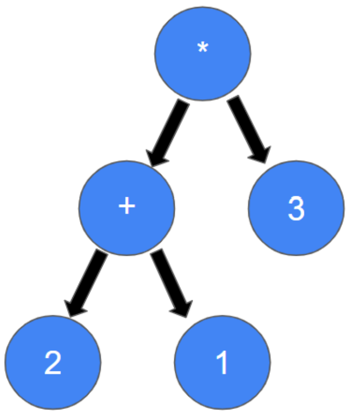
What does the diagram show?
In the diagram, it shows a queue data structure representation. In a queue it follows the first in first out principle. So using this diagram as a reference we can see that in the top array the value 9 is added and the bottom array the value 2 is taken out. Using this principle, we know that the number 2 was the first one that was added to the array. So the value 2 will be pushed out of the array and the value 9 is added at the top of the array. After these changes, the array indexes have also updated with the remaining values of the array to be matched with the new array indexes.

What can the stack data structure offer?
The queue data structure is a unique data structure that follows the First in, First out principle. This mean that the order that you entered is how the data is going to exit your queue. An example usage of this is if you want to manage a set of elements in the order that they were added.

Is a stack a good fit for a postfix interpreter?
This data structure might not be a good fit for a postfix program as it follows the first in first out principle will affect how the execution of the operations are done. This program calls for a last in first out principle where the top value is the result of the execution of the expression inserted by the user.

Limitations
However this data structure has its limitations. This data structure is very similar to stacks, in which both data structures are limited to the elements at both ends of the structure. It limits the access to edit the elements in the structure, in cases that the user wants to access, insert or modify the elements. Thus this concludes that this data structure is not a good fit for this program.

1.2.5) Binary Tree
Diagram 1.5



What does the binary tree diagram show?
In the diagram it shows a binary tree data structure. The root node is displayed as “ * ”, the multiplication symbol. The left child is “ + ”, the addition symbol. And the right child is “ 3 ”. Then from the left node, the left and right children are “ 2 ” and “ 1 ” respectively.

A binary tree diagram illustrates a hierarchical structure where each node has at most two children. These representations of children can also be referred to as the left child and the right child. At the top of the diagram is the root node. This root node provides the entry point of the binary tree. Each note in the tree contains a specific value, and the respective links to its children in the tree. Thus this forms a branching structure visualisation that extends in a downward direction.

What can the binary tree data structure offer?
The binary tree offers a wide range of value from hierarchical organisation structure. In a hierarchical organisation, it represents hierarchical relationships in a natural way, making it a suitable fit for expressions and decision processes. This means that the storage of data will be efficient and structured, thus in the case of storing expressions it will be an easy way to access information such as operands and operators in the binary tree.

Is a binary tree a good fit for a postfix interpreter?
Even with the efficiency in terms of storing and retrieving operands and operators in the binary tree, it may not be the best fit for a postfix interpreter as this data structure might grow to be too complex and may affect the runtime of algorithms within the program. Binary tree data structure might be a benefit in cases where there is a need to visualise expressions. When we want to convert a postfix expression into a binary tree it can help to visualise the hierarchical structure of the expression for educational purposes or for testing or debugging tasks.

Limitations
Binary tree data structures are powerful in some cases but it also has its limitations. Firstly, a binary tree can become complex in the process of constructing and managing a binary tree as compared to a stack for example. For simple linear evaluations a complex structure such as a binary tree can become a hindrance.
In the study of algorithms space complexity can also play a huge part in the execution time of an algorithm. In this case of using a binary tree, each node in the binary tree requires additional memory for pointers to its children. This can quickly become an inefficient usage for large datasets compared to more linear structures like stacks or arrays for example.

1.3) Justification and Summary

Justification

Through research and observation of these 5 data structures discussed in this report. I have found that the stack is the best fit for the postfix interpreter. This was my deduction of why the stack data structure is the best fit for a few reasons that includes, simplicity and direct mapping and efficiency.

1.3.1) Reasons for selected data structures

Stack

Firstly, simplicity and direct mapping. The postfix evaluation involves the nature of postfix expression also known as reverse polish notation that involves the last in first out principle of stacks. Operands are pushed into the stack and operators pop operands, performs the operations, and pushes the result back into the stack. This method is also known as direct mapping that makes use of the straightforward and direct implementation that makes the stack intuitive and efficient for evaluation.

Lastly, efficiency. The time complexity of the stack is O(1) in time complexity which is the best case that ensures efficient execution. The operations required for postfix evaluation processes which include push, pop all have this same best case time complexity of O(1).

Another important consideration in the context of algorithm efficiency is space complexity. The stack data structure efficiently manages its memory, as the size of the stack grows and shrinks in the context of the input expression provided by the user. This reduces the potential chance of managing more complex data structures.

Array

Another data structure that is a good fit for a simple postfix interpreter is the array. The static array has a fixed size of space for values to be inserted. This allows for simple evaluations to be done by the user. Even though the static array can be limiting it provides efficient access to all the elements in the array.

An important consideration in the context of algorithm efficiency is time and space complexity. Thus by using a data structure such as an array. The concern of time complexity reduces due to the fixed size of an array. However this data structure will not be beneficial in a case where there is a need to store a lot of data. But for a simple postfix interpreter it might still be good consideration as an efficient data structure.

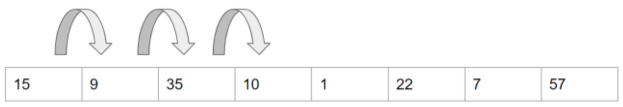
Summary

In summary, stacks and arrays are the best data structure for a postfix interpreter due to their simplicity properties, execution efficiency, and its direct alignment with the requirements of a postfix expression evaluation. While other data structures like dynamic arrays, queues, and binary trees have their own strengths for other cases, they do not offer the same level of efficiency for an interpreter. The stack's Last In, First Out nature, combined with its efficient and straightforward operations, makes it the best choice for the development of a reliable and effective postfix interpreter.

1.4) Algorithms to search the symbol table

For this implementation I have used a hash table for the search function. However, I will be discussing the other searching algorithms that exist and why I have chosen a hash table algorithm to do the search. The search algorithms that I will cover are linear search, binary search, and hashing.

1.4.1) Linear search



Firstly, linear search. This searching algorithm provides a straightforward method where each element in the dataset is checked in a sequential way until the specific element is found or the list has ended. The linear search is beneficial in areas where there are smaller datasets. And the linear search algorithm is easier to implement. However, this can become inefficient very quickly in the case that there is a large set of data to search with.

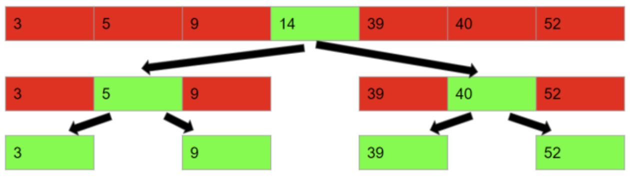
Time complexity

In a linear search algorithm, To consider the time complexity we need to see the best case, average case and worst case. Firstly, the best case time complexity is O(1). The best case occurs when the element that is searched is at the first position. Secondly, the average case. The average case time complexity is O(n). This might not be the fastest time execution that the program is able to provide to the user. And lastly, the worst case time complexity O(n). This case occurs when the element is at the last position in the list or it is not found.

Space complexity

In a linear search algorithm, the space complexity is O(1). This is in the case that the algorithm is structured as an iterative implementation.

1.4.2) Binary search



Green marked as mid where division of array takes place

Secondly, binary search. This searching algorithm requires a difficult prerequisite, which is that it will only work with a sorted dataset. How it works is that it repeatedly divides the search interval in half, reducing the search space by half each time. The binary search algorithm can be beneficial in the case that there is a large dataset. However it is limited to a sorted dataset, which may not work in the case of my application. As my application inserts the key and value based on the insertion from the user. This means that the list of symbols are stored based on when the user first inserted the value into the table. Another drawback of having a binary search algorithm is that it is more complex to implement than linear search. Having an implementation that is easier to implement allows for future enhancements and readability.

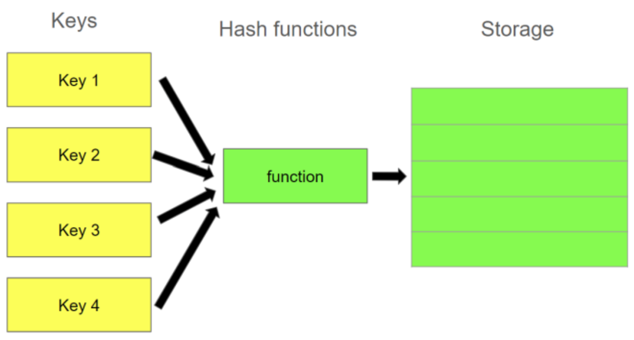
Time complexity

In a binary search algorithm, to consider the time complexity we need to see the best case, average case and the worst case. Firstly, the best case time complexity is O(1). This case happens when the element is at the middle of the list. Secondly, the average case. The average case time complexity is O(LogN). And lastly, the worst case time complexity is O(LogN).

Space complexity

In binary search, the space complexity for the iterative implementation structure is O(1) and in the case of a recursive structure is O(LogN) in the case due to a call stack.

1.4.3) Hashing



Lastly, the hashing algorithm. This algorithm uses a hash function to map keys to their associated values. This provides efficient execution for both search and inserting operations. The hashing algorithm can be beneficial in cases where there are search, insert, deletion operations involved. I feel that this algorithm will be a good fit for my code implementations. As I require all these functions for my application. However, the hashing algorithm is limited to some potential collisions and in addition it requires a good hash function.

Time complexity
In the hashing algorithm, to consider the time complexity we need to cover the best case, average case and the worst case. In the best case, the time complexity is O(1), where there are no collisions and the algorithm uses a hash function to map keys to their associated values. Secondly, the average case. The time complexity is O(1). This is a situation where the hash table is used for searching, inserting new values and deletion operations. And lastly, in the worst case. The time complexity is O(n) This is the case when collisions occur when doing the search, inserting and deletion operations.

Space complexity
In a hashing algorithm, the space complexity is O(n). This case occurs when values are stored in the hash table. This may be considered a disadvantage of using a hashing algorithm for storing values in my code implementation due to the restriction that the device has limited storage.

1.4.4) Algorithm Summary

For my code implementation for the symbol table, I will need to perform, search, inserting and deletion operations. And from this list of algorithms in consideration, the hashing algorithm fits the requirements the best.

For this project, the symbol table is used to store key to value pairs and perform these appending operations mentioned before. Hence this would most probably be the best choice as an algorithm for the symbol table.

Why may the hashing algorithm be useful?
There are 3 areas where the hashing algorithm may contribute to the successful execution of my code implementation. These 3 areas include efficiency, simplicity and scalability.

Efficiency
Firstly, efficiency. Hash tables provide average case constant time complexity of O(1) for search, inserting and deletion operations, which makes this algorithm a suitable algorithm for these requirements. This is because the hash function maps keys to specific indices in an array, allowing direct access to its values.

Another aspect of efficiency can be seen in the quick access with O(1) complexity, where the hash table is significantly faster as compared to other data structures like linked lists, which have O(n) complexity for these operations in the worst case.

Simplicity
Secondly, simplicity. Implementing a hash table is straightforward, and it effectively handles dynamic data with minimal performance implications. This means that the basic concepts behind hash tables are easy to understand and to implement and will allow a working hash table to be developed with basic operations much quickly.

Another aspect of simplicity is the minimal code required. The code required to implement a hash table is relatively concise. This helps in the maintenance and code debugging. This is due to the fewer lines to manage as compared to more complex data structures.

Scalability
Lastly, scalability. Hash tables perform well with large datasets, maintaining efficient operations even as the dataset grows. In other words, this means that hash tables can dynamically resize when the load factor increases. This ensures that the performance is consistent even in the case that more elements are being added.

Another aspect of scalability is space efficiency. While hash tables require additional space for the array and possible linked list for collision handling, they are generally space efficient for larger datasets. This is especially when compared to tree based structures which require more memory for pointers.
Time and space complexity considerations
The time complexity for the hashing algorithm in the best case and average case is O(1) which makes it an efficient algorithm to ensure the quick execution of appending the symbol table. But in the worst case, the time complexity is O(n). This happens when there are collisions when doing these operations. In terms of the space complexity, this can hinder the storage of elements as the space complexity is O(n). This can raise concerns for storage on the device given in this project. Given all of these detailed considerations, I have concluded that the hashing algorithm is the best fit for the requirements and tasks that are involved with the symbol table.

Section 2 - Implementation

2.1) Language choice

2.1.1) Why javascript?
For this project, I chose to use javascript to develop this postfix interpreter due to 3 advantages that javascript provides. These 3 advantages include real time interactivity, widely known language, and a learning curve.

Firstly, real time interactivity. Javascript provides real time interaction for users. This means that you can instantly evaluate postfix expressions and also allow instant feedback to the user. Execution time is crucial in the efficiency of algorithms. By having instant feedback and faster response time, allows for a better user experience and interaction with applications.

Secondly, a widely known language. Javascript is one of the most widely used programming languages. Due to its popularity, there is a huge amount of resources online that provide tutorials, and as well as collaboration with other developers that can support your learning journey as a developer. This allows learning easier and also when there are other new developers that want to further enhance your code implementations, it makes it easier for development and improvements.

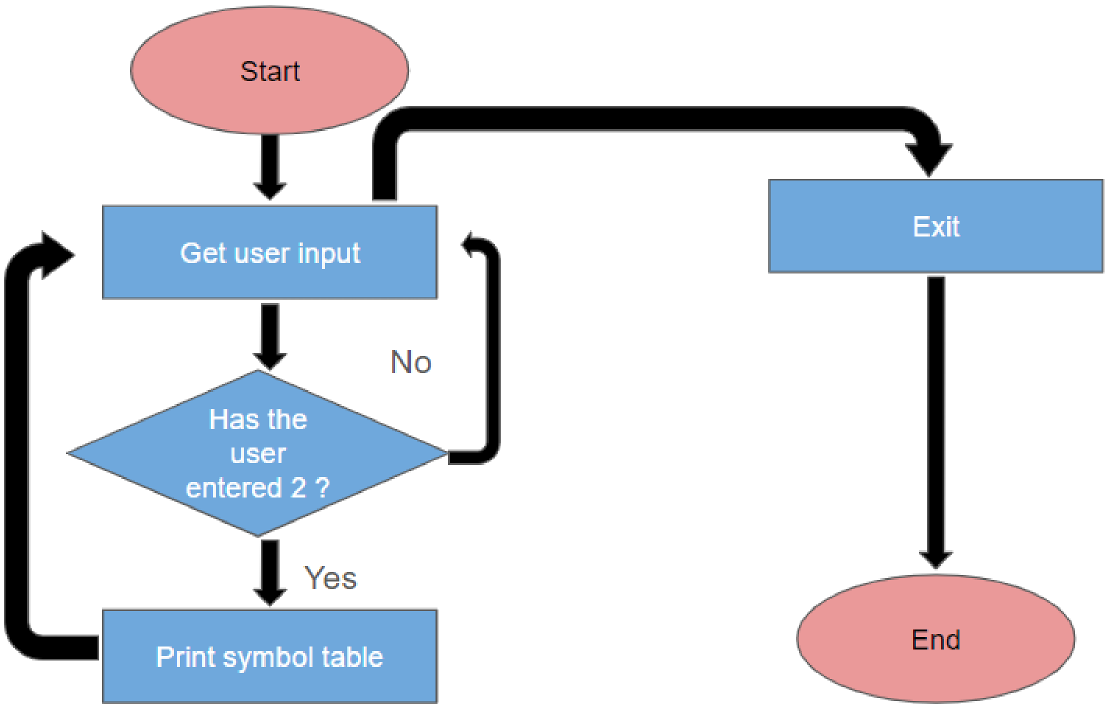
Lastly, a learning curve. Javascript is a language that is relatively easier to learn as compared to other languages. For many developers, this language has a gentle learning curve, which will lead to an increase in development time as well as reduce the learning period for new members in the team. For example, in the setting of a company or team project.

In summary, Javascript provides easy integration and fast response times which is essential for this project to do well. By combining both the gentle learning curve and instant feedback to users of the product. This language in my opinion is the best fit for the project.

2.2) Code implementation explanation

For this section I will be discussing the 4 main functions to manipulate the symbols in the symbol table. The 4 main functions include, view symbol, search symbol, update symbol, delete symbol.

2.2.1) View symbol table



Purpose
The purpose of this view symbol algorithm is to allow a user to view the current symbol table. This allows the user to be informed of all the available or existing symbols in the symbol table. This is the case that the user does not know what the existing symbols are.

Flow of algorithm
Step 1
In this view symbol algorithm, if the user inserts the value 2 into the terminal, then the symbol table with all the contents is being printed.

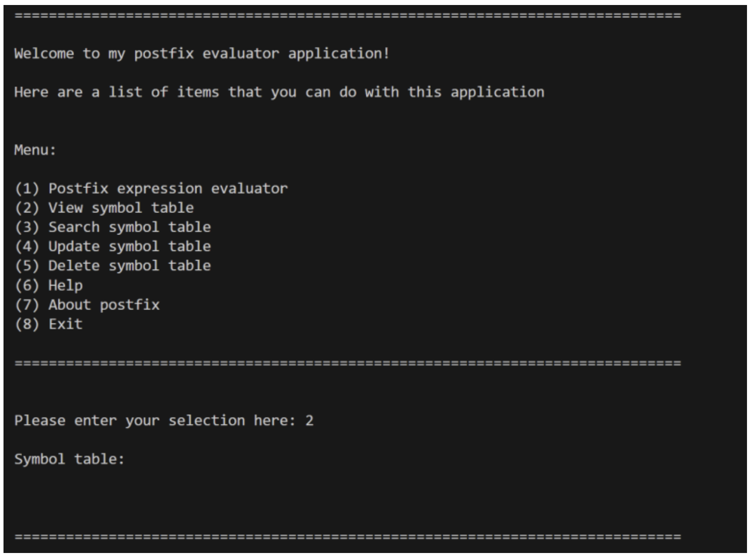
Step 2
The algorithm will only print the symbol table in the case that the user has inserted the value 2. If the user has not inserted the value 2 it will not display the symbol table and will just show the menu.

Step 3
From the menu selection, the user will also be able to exit the program. If the user has decided to exit the program, the program will end. That will be the end of the session.

View symbol algorithm design

This design is created such that it provides a pleasant menu driven interface, an aim of providing an efficient user interactive experience.

Graceful menu driven interface



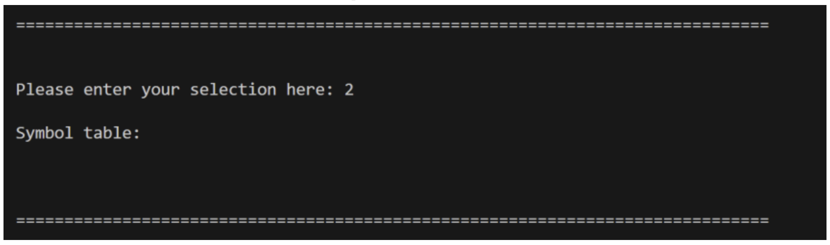
```
=======================================================================

Welcome to my postfix evaluator application!

Here are a list of items that you can do with this application


Menu:

(1) Postfix expression evaluator
(2) View symbol table
(3) Search symbol table
(4) Update symbol table
(5) Delete symbol table
(6) Help
(7) About postfix
(8) Exit

=======================================================================


Please enter your selection here: 2

Symbol table:



=======================================================================
```

Diagram 2.2.1.1

In diagram 2.2.1.1, a menu is shown and a user has inserted the value 2 into the terminal. After the user has inserted the value 2 the contents of the symbol table are returned. This provides an example of a graceful menu driven interface.

Firstly, a graceful menu driven interface. By providing a menu that is able to handle situations in a structured and systematic manner reduces the chances for errors and unexpected implications that might arise. In a hypothetical situation the user might enter a invalid value. By specifying specific values for the user to enter. This increases precision and execution time of the algorithm. This is because by eliminating the chance the user might insert another value, in the case that the user inserts another value the symbol table will not be printed.

Efficient user interactive experience

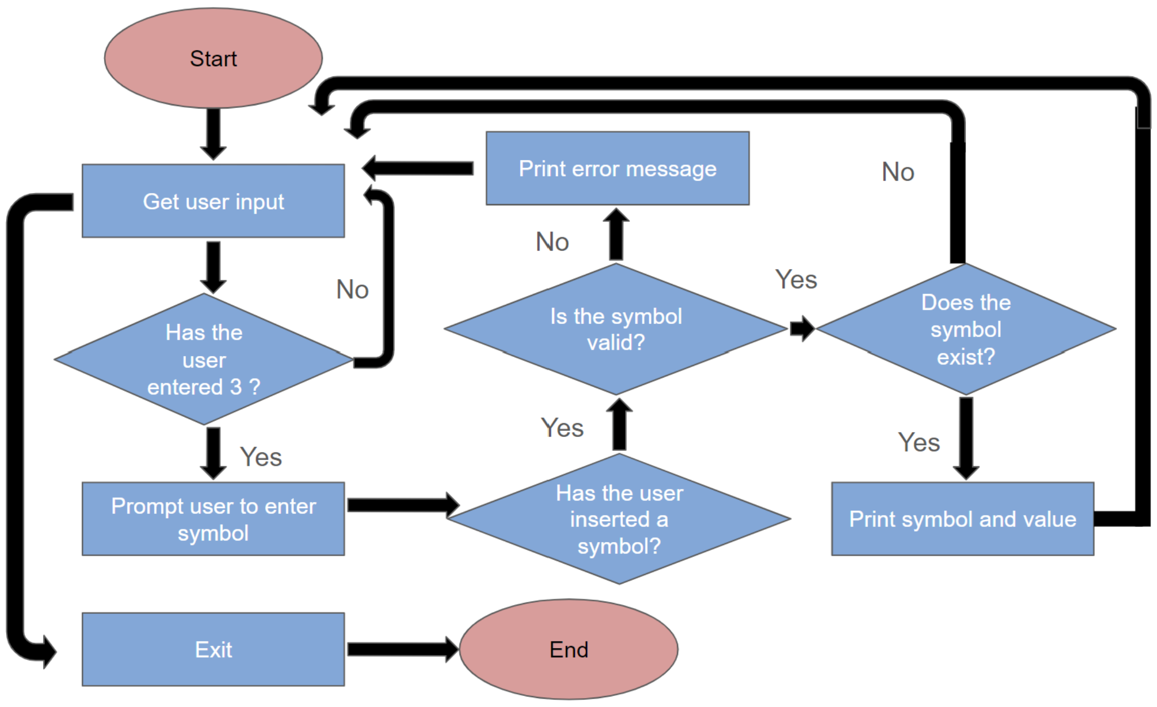


```
=======================================================================


Please enter your selection here: 2

Symbol table:



=======================================================================
```

Diagram 2.2.1.2

In diagram 2.2.1.2, a user has inserted the value 2 and the symbol table is returned back to the user. This is an example of an efficient user interface as it provides the user to select an option from the menu.
Secondly, providing an efficient user interactive experience. The algorithm uses user input to determine the choice from the user. This provides a pleasant experience as this provides the user with multiple choices. In this case the user selects the value 2, which can simulate an interaction like pressing a button on a physical device. Providing a interactive and friendly interface increases the effectiveness of the user experience. This helps the user to have a better time when using the application.

2.2.2) Search symbol table



Purpose
This search symbol algorithm allows the user to search for an existing symbol within the symbol table. Provided that the symbol already exists in the table, then the symbol and value are returned back to the user. However, if the user has inserted a symbol that does not exist, the menu is returned back to the user. This provides the user with a simple and quick response from the application.

Flow of algorithm
Step 1
The algorithm starts with receiving the user input. If the user has inserted the value 3, the algorithm will prompt the user to enter a symbol. If the user has not inserted the value 3 from the menu then the program will not do anything while the menu is displayed to the user.

Step 2
Next, the user will insert a symbol to search. If the symbol has been inserted then the algorithm will check if the symbol is in the correct format. If the user has not inserted any value when prompted to enter the symbol nothing happens.

Step 3
Next, if the symbol is valid, the algorithm will check if the symbol exists in the symbol table. If the symbol inserted is not valid an error message is returned and the menu is displayed back to the user.

Step 4
Next, if the symbol exists in the table then the symbol and its associated value is returned back to the user. However, in the case that the symbol does not exist in the table, then the menu is displayed to the user.

Search symbol algorithm design

The design of this algorithm provides a modular design and efficient error handling. This develops a more robust algorithm which is able to handles lots of different cases by verifying the accuracy of the input from the user.

Modular design

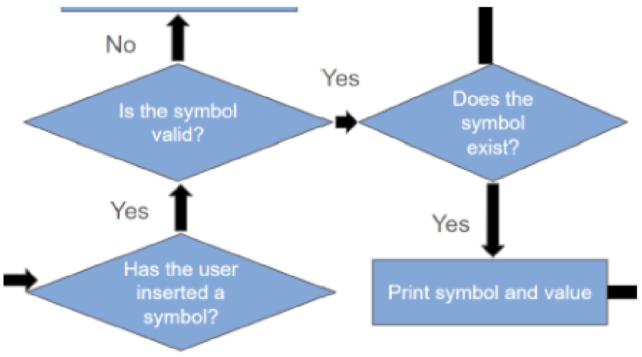
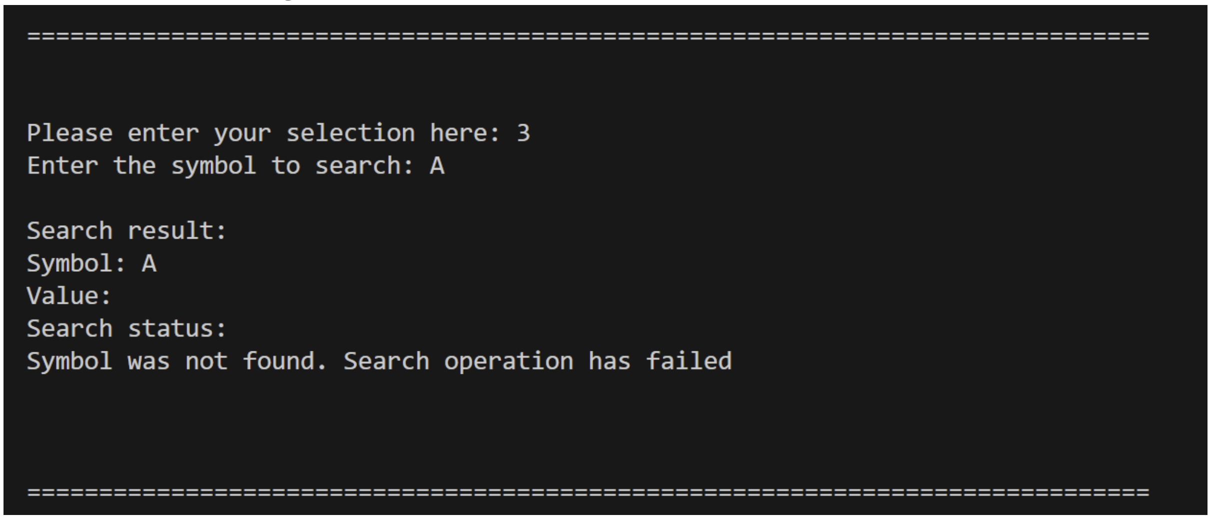


Diagram 2.2.2.1

In diagram 2.2.2.1, it shows a decision making process of what happens in the algorithm. This separates the concerns of the value inserted by the user. From symbol validation to checking for the existence of the symbol in the table.

Firstly, modular design. The implementation uses separate functions, such as one for asking a question and another function for searching the symbol hash. This separates user interaction and the symbol table search logic. This provides a higher level of reusability and maintainability.

Efficient error handling



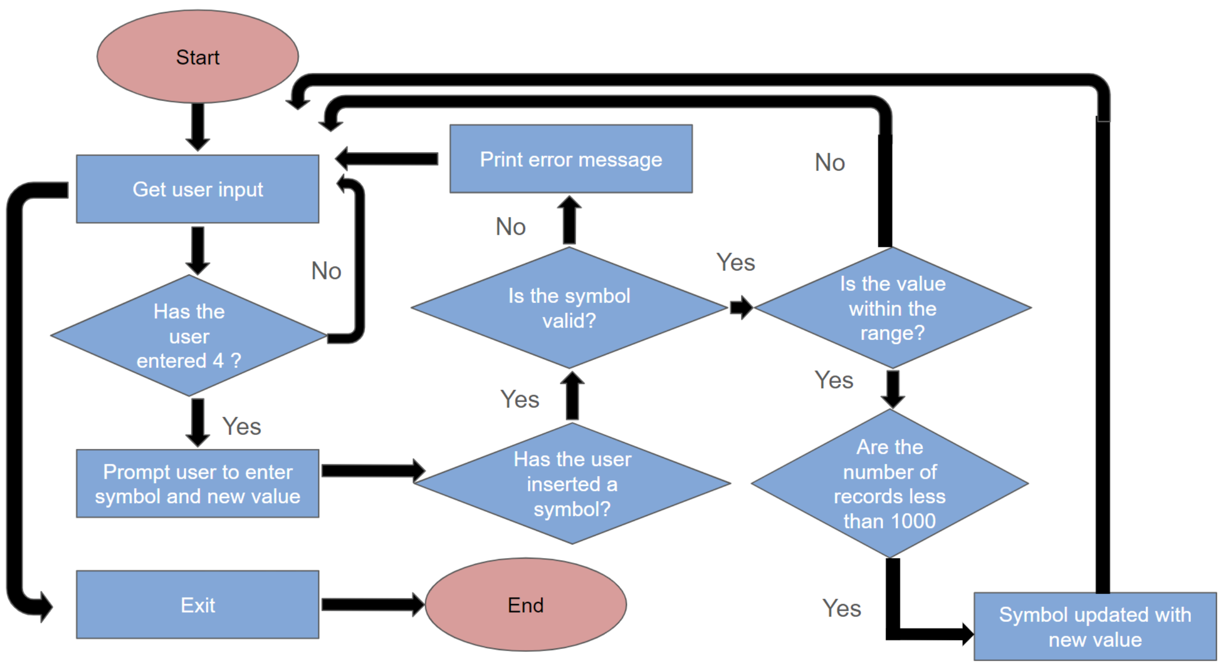
```
==============================================================================


Please enter your selection here: 3
Enter the symbol to search: A

Search result:
Symbol: A
Value:
Search status:
Symbol was not found. Search operation has failed




==============================================================================
```

Diagram 2.2.2.2

In diagram 2.2.2.2, it shows a user inserted the value 3 and provided the symbol A to be deleted. This is an example of how the user will delete a specific symbol from the table

Secondly, efficient error handling. By having different set cases for the algorithm to check, it enables the algorithm to react and respond in a quick way. For example, if the case where the symbol is not found and an error message is returned. This not only allows quick response time but also provides an informative message to provide information of what went wrong, and what could be possible remedies to the problem.

2.2.3) Update symbol table


Start
Get user input
Has the user entered 4 ?
No
Yes
Prompt user to enter symbol and new value
Exit
End
Print error message
No
Is the symbol valid?
Yes
Has the user inserted a symbol?
Yes
No
Is the value within the range?
Yes
Are the number of records less than 1000
Yes
Symbol updated with new value

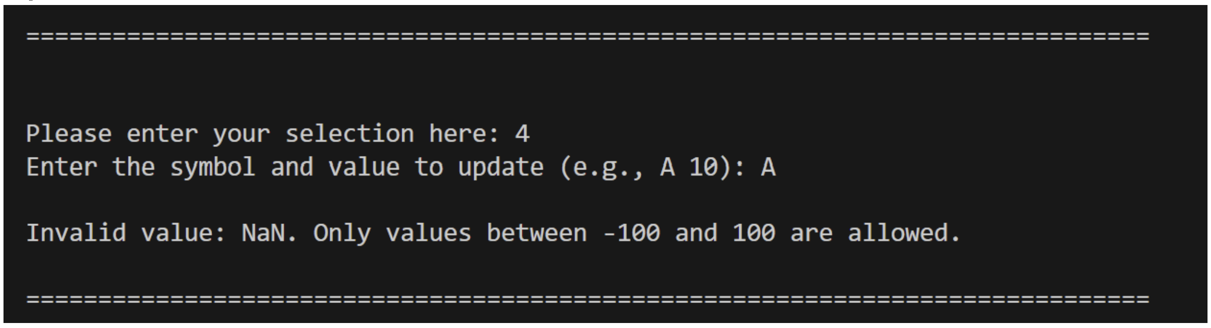
Purpose
The purpose of this update symbol algorithm is to allow a user to update the existing symbol in the symbol table with a new value. Through a few verification processes, it ensures that the value provided by the user is within the set range and restrictions implemented in the algorithm. With a clear set of restrictions, the updated value is ensured to be in the best condition to be stored. With proper checks it helps with other features of the application such as searching and viewing.

Flow of algorithm
Step 1
In this algorithm, if the user has inserted the value 4 into the terminal. This indicates that the user is requesting to update an existing symbol from the symbol table. If the user has not inserted the value 4, the application will continue to display the menu back to the user.
Step 2
Next, the algorithm prompts the user to enter the symbol and the new value. If the user has not entered anything the algorithm nothing will happen.
Step 3
Next, if the user has inserted a symbol, the algorithm will check if the symbol is or not. In the case that the symbol is not valid an error message is returned. In the case that the symbol is valid then the algorithm will check if the value inserted is within the set range.
Step 4
The set range for the value is -100 to 100. If the value inserted is not within the range then an error message is returned. In the case that the value inserted is within the range, then the algorithm checks if the number of records within the symbol table is below 1000.
Step 5
If the number of records is not less than 1000 an error message is returned notifying the user that the table is full. If the number of records is less than 1000 then the symbol is updated with the new value and the menu is displayed back to the user.

Update symbol algorithm design

The design of this algorithm provides input validation and error handling. This design provides a robust algorithm that is able to handle different cases of user input. Using efficient input validation and error handling procedures, enhances user interface experience.

Input validation



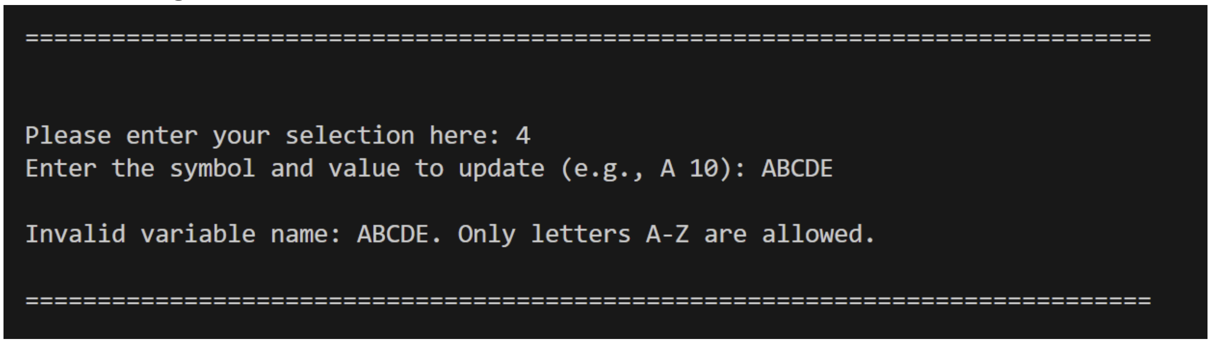
```
==============================================================================


Please enter your selection here: 4
Enter the symbol and value to update (e.g., A 10): A

Invalid value: NaN. Only values between -100 and 100 are allowed.

==============================================================================
```

Diagram 2.2.3.1

In diagram 2.2.3.1, it shows a user insert a value 4 into the terminal and provided an invalid value but it is in the range. This displays an effective user input validation by ensuring that the symbol provided by the user is in the correct format and notation.

Firstly, input validation. The algorithm validates both the symbol format and the value range to ensure the integrity of data in the symbol table. The set range for the symbol format is A to Z and the value range set is -100 to 100. By having clearly set restrictions, the new value is ensured to be of the best condition to be accessed and used in the future. For example, searching for the symbols and associated values. This is also in the case of the value range. By setting a limit on the value it prevents the user from inserting an extremely huge number or low number.

Error handling



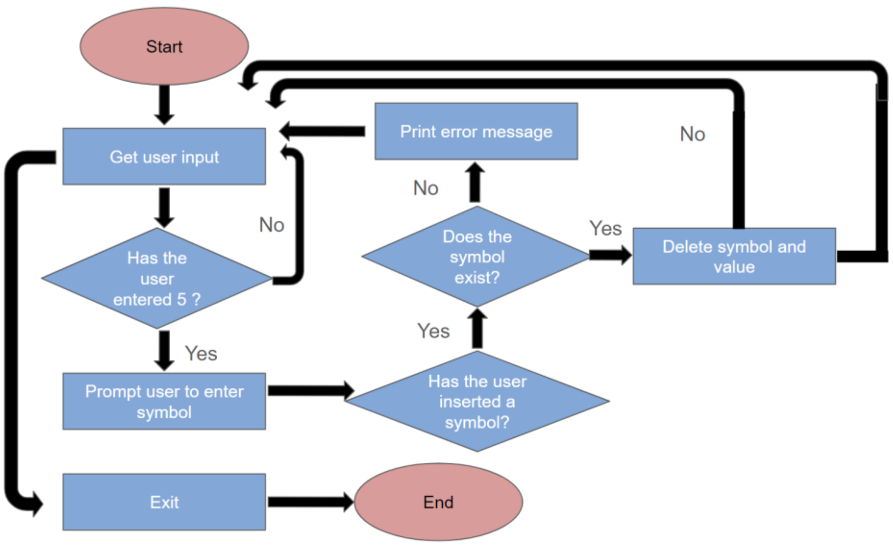
```
==============================================================================


Please enter your selection here: 4
Enter the symbol and value to update (e.g., A 10): ABCDE

Invalid variable name: ABCDE. Only letters A-Z are allowed.

==============================================================================
```

Diagram 2.2.3.2

In diagram 2.2.3.2, it shows a user inserted a value 4 into the terminal and inserted an invalid symbol that is not within the range of A to Z. This is a demonstration of how the algorithm is able to interpret the symbol inserted by the user.

Secondly, error handling. The algorithm is able to handle various error scenarios. This includes invalid symbols and values or even cases of exceeding the storage space of the symbol table. By having efficient error handling, it allows the user to improve on the way that values are inserted into the terminal. By restricting values inserted, it allows the algorithm to respond quickly and handle the situation by printing an error message to the user to change the value that was previously inserted by the user.

2.2.4) Delete symbol table



Purpose
The purpose of this algorithm is to allow a user to delete an existing symbol from the symbol table. Through checks such as symbol existence, ensure that only the specific symbol is selected and deleted. This ensures the integrity of data as well as the accuracy of the algorithm. If the symbol was deleted incorrectly this might cause difficulties and an inconvenience to the user.

Flow of algorithm
Step 1
In this algorithm, if the user has inserted the value 5 into the terminal, this indicates that the user is requesting to delete an existing symbol and value. If the user has not inserted any value then the algorithm will not do anything, and the menu will be displayed to the user as a selection has not been made.

Step 2
Next, if the user has inserted the value 5 then the algorithm prompts the user to insert a symbol. If the symbol was not inserted nothing happens. But in the case that a symbol is inserted, the algorithm will check whether the symbol exists in the symbol table.

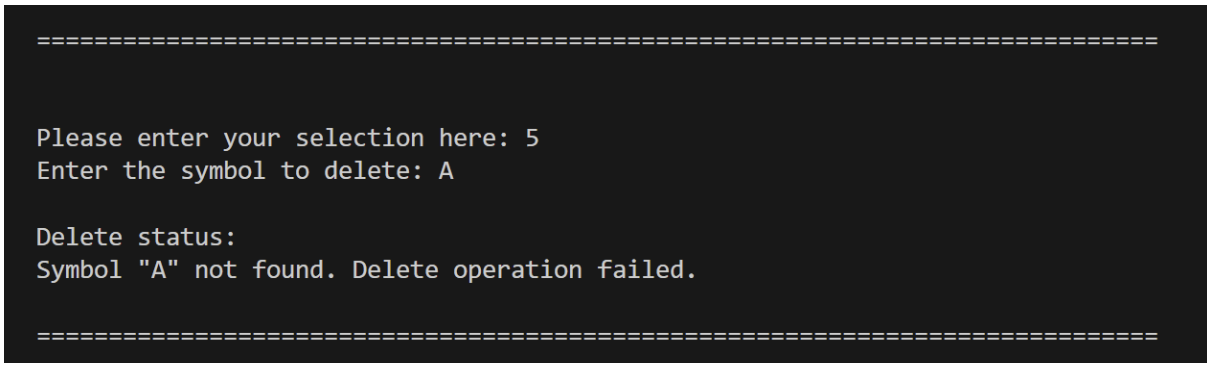
Step 3
If the symbol does not exist then then an error message is printed back to the user. However, in the case that the symbol is found then the symbol and its associated value is deleted. And at the end of this the menu is displayed back to the user.

Step 4
The user at this point can also exit at any time during this session.

Design

The design of this algorithm provides integrity of stored data and simple user experience.

Integrity of stored data



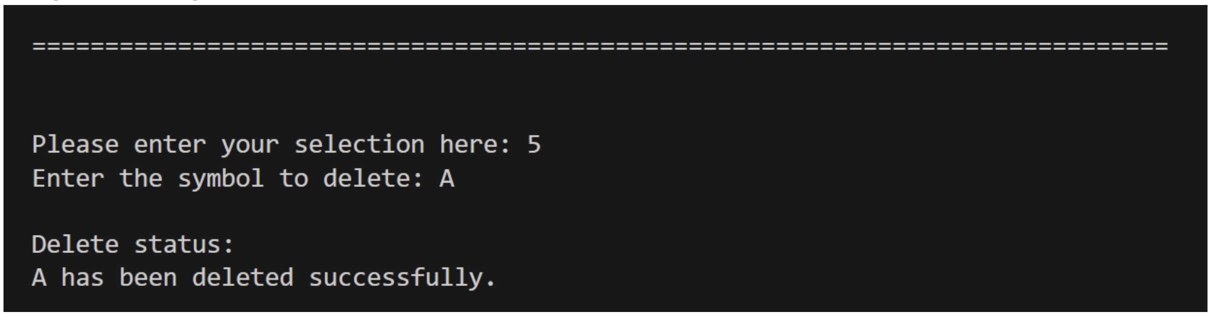
```
===============================================================================


Please enter your selection here: 5
Enter the symbol to delete: A

Delete status:
Symbol "A" not found. Delete operation failed.

===============================================================================
```

Diagram 2.2.4.1

In diagram 2.2.4.1, it shows a user inserting a value 5 into the terminal. The user then provides a symbol that does not exist in the table. An error message is then returned back to the user. The error message indicates that the symbol was not found from the table. This is a demonstration of how specific symbols are selected by the user to be deleted from the table to ensure data integrity.

Firstly, integrity of stored data. By ensuring the integrity of stored data, it prevents any unwanted deletions of existing symbols that the user might still want to keep. By allowing the user to enter the specific symbol, this will specifically select that symbol and that value and delete it. With this restriction set in place, it provides a reliable and effective algorithm that accurately carries out deletion operations.

Simple user experience



```
===============================================================================


Please enter your selection here: 5
Enter the symbol to delete: A

Delete status:
A has been deleted successfully.
```

Diagram 2.2.4.2

In diagram 2.2.4.2, It shows a user insert the value 5 into the terminal. Then the user provides the symbol A. In this case the symbol existed in the table to the delete status indicates that the symbol was successfully deleted from the table. This is a demonstration that this is a simple user interface experience as all the user needs to do is to insert the symbol A to delete the symbol from the table.

Secondly, simple user experience. By providing the user with a simple insertion of a symbol, it provides the user with a more hands off approach to the algorithm. All the user needs to do is to insert the value into the terminal and the deletion operation is done. This is provided that the user has inserted a symbol that exists in the table

Section 3 - Report

3.1) What does my program do?

My solution to this challenge for creating a postfix expression evaluator is by creating a program that allows the user to insert a postfix expression into the terminal. From this expression there can be two cases, firstly the expression contains arithmetic operators and is evaluated which then returns the result at the top of the stack or the expression contains an “ = ” operator, which means that this is a request to append the symbol table. This solution utilises data structures such as arrays and stacks to store and alter symbols effectively by storing values in a structured way. In this way the user will be able to access or append existing values. This program provides a list of features such as postfix expression evaluation, symbol table updating through selecting the specific symbol in the table, and also includes a user-friendly interface for efficient user interaction. By having an efficient user interface it allows a non-technical user to easily use this program to evaluate expressions, and to view, edit, update, delete symbols in the symbol table.

Overview

In this section I will explain the original algorithms in my solution to create a postfix interpreter in a non technical language. I will cover the workings of my program design and management of the symbol table. The program allows users to perform various basic arithmetic operations through a sime ple user interface.

3.2) Overview of key components

Postfix expression evaluator
The purpose of this evaluator is to calculate the result of mathematical expressions written in the postfix notation. Postfix notation are written in the format where the arithmetic operator such as +, -, *, / come after their operands such as variables or numbers. An example of a postfix notation can be written in the format “ 3 4 + ”. The regular notation that is normally written in technologies such as calculators are written in the infix notation as “ 3 + 4 ”.

Symbol table management
The purpose of the symbol table is to store and manage variables and their assigned values. This is to achieve an efficient way for the user to access stored variables to do postfix evaluation using existing values that were assigned before. The symbol table is a collection where each variable is represented by a letter from A to Z and is then associated with a numerical value that is stored into the variable that is also known as a symbol.

3.3) Detailed explanation of original algorithms

Evaluating postfix expressions
In postfix expression evaluation there are 2 parts, which include input processing, handling tokens.

Input processing
Firstly, Input processing. The expression is split into individual tokens such as numbers, variables, or operators. This input is inserted by the user and the postfix evaluator takes these elements and understands what the purpose of each token is. For example for a number, the evaluator understands that it is has the nature of a numerical value.

Handling tokens
Secondly, handling tokens. After the evaluator has split the tokens into numbers, variables and operators. It evaluates each token and utilises each of them based on its understanding of what each token provides. So with the understanding that there are three parts, numbers, variables and operators. I will cover how each of the parts are being understood by the evaluator.

Numbers

Numbers, a token that is understood as having the nature of a numerical value. So for my program, from the understanding that the storage of elements will be for a small device, I have limited the range of numbers to start from -100 and 100. The program first identifies if the value is in this range. In this decision there are two cases, the first case is that the value inserted is not in the range and the program returns an error message and the second case is that the value is in the range and the value is pushed into the stack directly.

Variables
Variables, a token that is understood as having the nature of an alphabetical value. So for my program, from the understanding that the storage of elements will be for a small device, I have limited the range of variables to be from A to Z. The program first identifies if the variable is in the list of A to Z characters. In this decision there are two cases, the first case is that the variable is not in the list of A to Z and the program returns an error message and the second case is that the value is in the range and the value is pushed into the stack directly. And to elaborate more on this second case, a verification is done to check if the variable has been assigned a value or not. If the variable was not being assigned a value then a value of “ 0 ” is assigned to the inserted variable.

Operators
Operators, a token that is understood as having the nature of an operation value. So for my program, it handles 6 different operators “ +, -, *, /, %, ^ “. The program will pop two numbers from the stack, perform the operation and push the result of the operation back onto the stack. The program will first identify whether the operator entered is one of the 6 operators implemented in the solution. Then it carries out the pop and push actions required using the stack that stores all the elements extracted from the user input expression.

Result
Lastly, after handling the tokens the final result is returned. The result of the operations is stored at the top of the stack. The stack follows the Last In, First Out principle. So taking reference to this principle the result will pop from the stack and is printed in the terminal.

Updating the symbol table
Symbol table updating can be split into two parts, input validation and storage. Input validation is known as a verification process of whether the value inserted has met a certain condition before the program proceeds with a specified action. And the other part is storage, for storage in this case the values are stored into the symbol table. The structured storage into a symbol table provides an efficient way for the program to easily access any element in the symbol table and use it for postfix evaluation. For example, if the user wants to use a previously inserted variable to do some calculations.

For input validation, the program verifies that the variable name is a character between A to Z and also checks if the value inserted is in the allowed range of -100 to 100. If both of these condition are met then the new variables and numbers are stored and updated in the symbol table.

Deleting a symbol

Symbol deletion in this program can be split into two parts, which includes existence verification and the actual deletion of the symbol from the symbol table.

Existence verification, essentially means to verify if the symbol that has been requested to be deleted exists in the symbol table. For this verification, there can be two cases. The first case is that the symbol does not exist and an error message is returned notifying the user that the symbol does not exist in the symbol table. And the other case is the symbol exists and the deletion operation can be executed.

Searching for a symbol
Symbol searching can be split into two parts, existence verification and returning the result. Firstly, existence check. Existence verification involves the process of checking if the variable exists in the symbol table. In this case there can be two outcomes. The first outcome is that the program was not able to find the symbol in the symbol table. If the symbol was not found in the table an error message is returned. The second outcome is that the symbol is found and the symbol and value is returned and printed onto the terminal.

User interface

The program offers a wide range of features that can be accessed through a friendly user interface that has the following options, which include, postfix expression evaluator, view symbol table, search symbol table, update symbol table, delete symbol from symbol table, help page, history of postfix, exit. I will be explaining what is the purpose of each feature and how it provides value to the user.

Postfix expression evaluator
Purpose
The purpose of the postfix expression evaluator is to evaluate a postfix expression inserted by the user through the terminal. The postfix evaluator only understands an expression that is in the postfix notation.

What does it provide?
This provides the user with the freedom to enter any postfix expression without the worry that the expression was inserted incorrectly. This is due to the effective verification checks built in to the program which allows for not only a robust verification layer but also ensures that the values stored in the symbol table or stack is in the best condition for the purposes of execution and evaluation of expressions.

View symbol table
Purpose
The purpose of a view feature for the symbol table is to allow the user to view all the records of all the symbols that were inserted previously.

What does the symbol table view provide?
This provides the user an easy access to all the records so that the user will have the ability to use for future postfix operations. For example, if the user wants to find out all the previously stored symbols and does not know what symbols were already inserted into the program. The view feature will be able to provide that information to the user.

Search symbol table

Purpose
The purpose of a search feature for a symbol table is to allow the user with a quick way to search up any previously stored symbols that the user already knows of. The user only wants to search up the symbol to know what value was associated with the particular symbol.

What does the search symbol table provide?
This provides the user with an efficient and quick way to first insert the selected symbol in the terminal and retrieve the value associated with that symbol. This provides the user with a simple way to get informed of the symbols and values in the symbol table.

Update symbol table
Purpose
The purpose of an update feature for a symbol table is to provide the user with a quick and simple way to update existing symbols already in the symbol table with a new value. The user will first have to enter the symbol name and right after that the new value.

What does the update symbol table provide?
This provides the user with an efficient and user-friendly way to update existing symbols without the need to worry whether the symbol exists in the table or not. If the symbol does not exist an error message will be returned and if the symbol exists in the symbol table the symbol will be updated directly in the symbol table.

Delete symbol
Purpose
The purpose of a delete feature for a symbol table is to provide the user with a simple way to delete existing symbols already in the symbol table. The user will first have to enter the value to be deleted. Then through a series of verification processes, the symbol will then be deleted from the symbol table.

What does the delete symbol provide?
This provides the user with a user-friendly way to delete existing symbols without the need to worry whether the symbol exists in the symbol table or not. This is because the program is able to check whether the symbol exists in the symbol table. If the symbol exists in the symbol table then the symbol will be deleted from the symbol table, if not an error message will be returned.

Help page
Purpose
The purpose of a help page is to provide a help guide for any users who do not know how to enter a postfix expression into the terminal. The program will ask the user whether the user knows how to insert a postfix expression. If the user does not know then the user can attain this knowledge by referring to the help page

What does the help page provide?
The help page provides the user with the necessary order of elements needed in a postfix expression. This is for users who do not have the prior knowledge of how a postfix expression is inserted. This provides the opportunity for any user whether the user already has the knowledge of postfix expressions or not to have the ability to use the program to evaluate postfix expressions.

About postfix

Purpose
The purpose of having a about postfix page is to allow the user to have the knowledge of what postfix is and also a comparison to infix notation expressions. Usually expressions are inserted as infix notation in technologies such as calculators. But in this case, this program evaluates postfix expressions. So this page provides a general description of what postfix is and how the expression is structured.

What does the about postfix page provide?
The about postfix page provides the user with a general description of what postfix is and how the expression is structured. It provides the user with only the general description and not the history of postfix and how it came about. This is because this might be too long and complex for the user to understand. For the aim of this project I wanted to create a simple and user friendly interface for the user.

Exit
Purpose
The purpose of having an exit for this program is to allow the user to exit the program after all the calculations and operations. This is a totally optional choice for the user and provides a whole complete cycle for the program from expression evaluations all the way to ending or existing the program.

What does exit provide?
Exit provides the user with an option to end the program and to make sure that the session ends. This is a totally optional choice for the user and just provides a complete cycle for the program from start to finish.

3.4) Pseudocode

Algorithm 1: evaluatePostfixExp

```
function evaluatePostfixExp(inputExp)
    PRINT("Expression being evaluated: " + inputExp)
    expTokens ← SPLIT(inputExp, ' ')

    hasOperator ← FALSE

    FOR i ← 1 TO LENGTH(expTokens) DO
        token ← expTokens[i - 1]
        PRINT("Token: " + token)

        IF token MATCHES /^[A-Z]$/ THEN
            value ← collection[token]

            IF value = NIL THEN
                PRINT("Symbol " + token + " not found, pushing 0")
                PUSH(stack, 0)
            ELSE
                PRINT("Symbol " + token + " found, pushing " + value)
                PUSH(stack, value)
            ENDIF

        ELIF token ∈ ['+', '-', '*', '/', '%', '^'] THEN
            operand1 ← POP(stack)
            operand2 ← POP(stack)

            IF operand1 = NIL OR operand2 = NIL THEN
            ERROR("Insufficient operands for operator: " + token)
         ENDIF

         PRINT("Popped operands: " + operand2 + ", " + operand1)
         hasOperator ← TRUE

         SWITCH token DO
            CASE '+':
               PUSH(stack, operand2 + operand1)
               PRINT("Performed addition: " + operand2 + " + " + operand1 + " = " + (operand2 + operand1))
            CASE '-':
               PUSH(stack, operand2 - operand1)
               PRINT("Performed subtraction: " + operand2 + " - " + operand1 + " = " + (operand2 - operand1))
            CASE '*':
               PUSH(stack, operand2 * operand1)
               PRINT("Performed multiplication: " + operand2 + " * " + operand1 + " = " + (operand2 * operand1))
            CASE '/':
               IF operand1 = 0 THEN
                  ERROR("Division by zero error")
               ENDIF
               PUSH(stack, operand2 / operand1)
               PRINT("Performed division: " + operand2 + " / " + operand1 + " = " + (operand2 / operand1))
            CASE '%':
               PUSH(stack, operand2 % operand1)
               PRINT("Performed modulus: " + operand2 + " % " + operand1 + " = " + (operand2 % operand1))
            CASE '^':
               PUSH(stack, POW(operand2, operand1))
               PRINT("Performed exponentiation: " + operand2 + " ^ " + operand1 + " = " + POW(operand2, operand1))
            DEFAULT:
               ERROR("Invalid operator: " + token)
         END SWITCH

      ELIF token = '=' THEN
         updateVariable ← expTokens[i - 2]
         value ← POP(stack)
         UPDATE-SYMBOL-TABLE(updateVariable, value)
         PRINT(updateVariable + " has been updated successfully!")

      ELSE
         num ← PARSE-FLOAT(token)
         IF NOT IS-NAN(num) AND IS-VALID-NUMBER(num) THEN
            PUSH(stack, num)
            PRINT("Pushed number: " + num)
         ELSE
            PRINT("Invalid number: " + token + ". Only values between -100 and 100 are allowed.")
         ENDIF
      ENDIF

      PRINT("Current stack: " + JSON.STRINGIFY(stack))
   END FOR

   result ← POP(stack)
   IF hasOperator THEN
      PRINT("Final result: " + result)
   ENDIF

   PRINT("Final stack: " + JSON.STRINGIFY(stack))

   RETURN result
END function
```

Purpose
The purpose of this algorithm is to evaluate postfix expressions provided by the user through user input. It processes each token in this expression, which can include a symbol, operator, assignment token, or a numerical value. Additionally, it provides an update functionality for the symbol table. With symbols corresponding to values extracted from this expression.

Explanation of implementations
To explain the implementation of this algorithm I will split it into 2 sections, iteration of variables and token handling.

Firstly, iteration of variables. The algorithm splits the expression into tokens which are stored into a stack that is initialised to store operands and returns a result.

Secondly, token handling. There are different token meant for different purposes and with this implementation the program can make use of these token through verifications and checking of operators and operands. After calculations the algorithm pops the result from the top of the stack.

Algorithm 2: updateSymbolTable

```
function updateSymbolTable(symbol, value)
   IF symbol MATCHES /^[A-Z]$/ THEN
      IF IS-VALID-NUMBER(value) THEN
         IF symbolCounter < 1000 THEN
            collection[symbol] ← value
            symbolCounter ← symbolCounter + 1
            PRINT("Update status: Assigned: " + symbol + " = " + value + " successfully.")
            PRINT("Current size of symbol table is " + symbolCounter)
         ELSE
            PRINT("Update status: Update failed.")
            PRINT("The symbol table is full, the current number of symbols in the table are " +
symbolCounter)
         ENDIF
      ELSE
         PRINT("Invalid value: " + value + ". Only values between -100 and 100 are allowed.")
      ENDIF
   ELSE
      PRINT("Invalid variable name: " + symbol + ". Only letters A-Z are allowed.")
   ENDIF
END function
```

Purpose
The purpose of this algorithm is to update the symbol table with a new value associated with a symbol provided by the user using user input. It ensures that symbols are valid and values fall within the specified range of -100 to 100. In addition, it manages the size limit of the symbol table to prevent overflow. The maximum number of records in the symbol table is 1000 records.

Explanation of implementations
For this explanation I will split it into 3 parts, symbol validation, value range validation and symbol table size management.

Symbol validation
Firstly, symbol validation. Symbol validation provides a check for single uppercase letter format. This means that only letters A to Z are allowed as symbols for this program. If the letter inserted is not in single uppercase letter format, an error message is returned notifying the user that the symbol format is incorrect.

Range validation
Secondly, value range validation. For the symbol table it stores the symbol and the value that is associated with it. Thus, by using value range validation. It ensures that the value inserted is numeric and that it falls in the range of -100 to 100. If the value is not in the range, it returns an error message notifying the user that the value inserted is not allowed.

Symbol table size management
Lastly, symbol table size management. Taking into consideration that the device that is storing all of this data has a small memory, I created a limit to the number of records that can be stored into the symbol table. The maximum number of records in a symbol table is 1000 records. In the implementation, a counter for the number of records is used to track the current number of records existing in the table. The program checks if the number of records is less than 1000 records before allowing a new record to be inserted. If there are more than 1000 records then an error message is returned. However, if there is less than 1000 records then a success message is returned. This success message confirms that the assignment and the updates the current size of the symbol table. This symbol table size is then notified to the user so that the user knows how many records more the user can store into the symbol table.
Algorithm 3: deleteSymbol

```
function deleteSymbol(symbol)

  IF collection.HAS-OWN-PROPERTY(symbol) THEN
    DELETE collection[symbol]
    symbolCounter ← symbolCounter - 1
    PRINT("Delete status: " + symbol + " has been deleted successfully.")
    PRINT("The current number of symbols in the table are " + symbolCounter)

  ELSE
    PRINT("Delete status: Symbol "'" + symbol + "'" not found. Delete operation failed.")

  ENDIF
END function
```

Purpose
The purpose of this algorithm is to remove a specific symbol from the symbol table based on the input from the user. It verifies if the symbol exists in the table and, if the symbol is found, it deletes it while updating the symbol counter. If the symbol does not exist, it returns an error message notifying the user that the deletion operation has failed.

Explanation of implementations
For the explanation of this implementation I will be splitting it into 2 parts, which include function initialization and deletion.

Function initialization
Firstly, function initialization. The algorithm starts by accepting a single parameter known as a symbol. This symbol represents the symbol to be deleted by the program from the symbol table. When the user inserts a symbol into this parameter the algorithm will take this symbol and compare it with the symbol table to extract the relevant symbol in the table and directly delete it if the symbol has been found.

Deletion
Secondly, deletion. The algorithm will check for the existence of the symbol. If the symbol has been found then the user will be notified that the operation was successful and the counter for the number of records will decrease by 1. This means that one record has been successfully deleted from the symbol table.

Importance of checking for the existence of a symbol
Checking for the existence of a symbol is important as it provides 3 benefits, which include preventing errors, ensuring data integrity and error handling.

Firstly, preventing errors. Before performing operations such as updates or deletions, it is essential to verify that the symbol exists in the table. The attempt to modify or delete a non-existent symbol can lead to runtime error or non desirable behaviours in the program.

Secondly, ensuring data integrity. Symbol tables often store critical data, such as variable values or identifiers. Checking for the existence of a symbol ensures that only valid and recognized symbols are changed, this helps to maintain integrity and consistency of data storage.

Lastly, error handling. By checking if a symbol exists before attempting an operation, the program can then handle scenarios where an invalid or non existing symbol is referenced.
Algorithm 4: searchSymbolHash

```
function searchSymbolHash(symbol)
   IF collection.HAS-OWN-PROPERTY(symbol) THEN
      PRINT("Search status: Symbol has been found successfully.")
      RETURN collection[symbol]
   ELSE
      RETURN "Search status: Symbol was not found."
   ENDIF
END function
```

Purpose
The purpose of this algorithm is to search for a specific symbol in the symbol table. It effectively checks if the symbol exists in the symbol table and returns either the symbol value if found. If the symbol has been found then it returns a success message and the value of the symbol that was inserted. However if the symbol was not found then an error message will be returned to notify the user that the symbol inserted was not found in the table.

Explanation of implementation
For this explanation I will split into 2 parts, function initialisation and search.

Firstly, function initialisation. The algorithm begins by accepting a single parameter known as a symbol. This symbol represents the symbol that has been requested to search from the symbol table. As mentioned, if the symbol has not been found an error message is returned.

Secondly, search. The algorithm checks for the existence of the symbol. By searching for the symbol it uses a built in function from the symbol table to check for the existence of the symbol. This can be seen in the implementation of the search function. If the search has been successful then the symbol inserted will be returned with the value from the symbol table known as collection.

Time Complexity
The time complexity of this algorithm can be split into 3 parts. These 3 parts include checking if the symbol exists, retrieving the value, and conditional statements. All these parts combined returns a time complexity of O(1). This is very efficient as an algorithm as this is the fastest execution time. This will greatly improve the efficiency and reliability of this algorithm.

Space Complexity
The space complexity of this algorithm can be discussed in 2 parts. These 2 parts include function execution and return value. Both these parts combined return a space complexity of O(1). This is because it does not use extra space that depends on the size of the input.

Algorithm 5: isValidNumber

```
function isValidNumber(num)
   RETURN (num ≥ -100) AND (num ≤ 100)
END function
```

Purpose
The purpose of this algorithm is to validate whether a given number falls within the range of -100 to 100. It provides a simple check to ensure that the user input for numerical values is verified with the specified criteria

Explanation of implementations
For this explanation I will split into 2 parts function initialisation and validation.

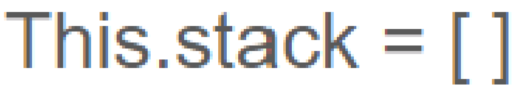
Firstly, function initialisation.The algorithm begins by accepting a single parameter known as num. This represents the number that is to be validated. This validation is done so that only the values -100 to 100 are accepted into the symbol table. Upon successful verification the relevant operations could potentially be carried out, which include updating existing values or creating new symbols in the symbol table.

Secondly, validation. The algorithm checks if the value is between -100 and 100 and if that is verified then it will return true. This means that the number is valid according to the range specified. However, if the value inserted is not within the range then the algorithm will return false. This indicates that the number is outside of the accepted range and an error message is returned back to the user.

Importance of checking for valid values
By checking valid values, it provides a robust and efficient symbol table that allows the user to have quick access to all the available symbols in the symbol table. In the case where the device has a small memory. Taking into account the values inserted also plays an important part in memory space. By limiting the values to a range of -100 and 100, the management of memory and execution efficiency will be significantly improved. This increases reliability and trust from the user who uses this program for expression evaluations.

3.5) Explanation of data structures

Overview
In this section I will be covering the 3 data structures that I have used to develop my solution for this project. The data structures that I have chosen include array and stack.
Array



```
This.stack = [ ]
```

What is an Array?
An array is an ordered collection of elements. Each element is identified by an index, starting from 0. Array can store elements of any type, including other arrays and objects. In the context of this project, the array can come in handy in situations where there is a need for a data structure that provides an index based storage structure.

In the above diagram the array is used in the stack to store the tokens from the postfix expression from the user input.

Why is the array a good fit for the task?
The array offers 3 main benefits that are good for storing elements such as tokens for a postfix expression evaluator. The benefits include, ordered collections, index based access, array methods.

Firstly, ordered collections. Arrays are used to store ordered collections of items. Arrays are flexible for the use of storing values and provide an ordered collection of elements. Such that it allows for a visual representation of all the elements that are stored in the array.

Secondly, Index based access. Elements can be accessed using their index number. Using an index based access this allows for easy management and accessibility of any element stored in the array. For the case of storing the tokens, this provides an effective and structured method of storing the tokens such that it allows the program to be able to access each element instantly and reduces the execution time of specific algorithms in the program.

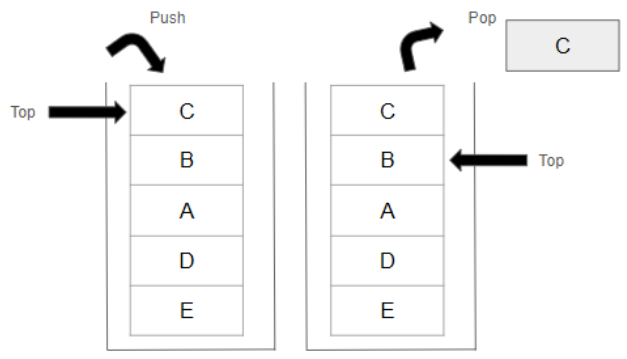
Lastly, array methods. Javascript provides many built-in methods for manipulating arrays. Some methods can include, push, pop. Which is essential for the usage to store tokens from the postfix expression after each token has been split from the terminal using the input from the user.

Usage in code implementation:

```
1  //import libraries
2  const readline = require('readline');
3
4  //postfix interpreter class
5  class PostFixInterpreter {
6    constructor() {
7
8        //include the collection and stack updateVariable
9        this.collection = [];
10       this.stack = [];
11       this.symbolCounter = 0;
12   }
```

Explanation:
At line 10 of the diagram the stack variable is initialised as an array. This stack is used to store all the tokens that are extracted from the postfix expression provided by the user. From this stack the program is able to manipulate the tokens in the array using built in array methods such as push and pop.

Stack



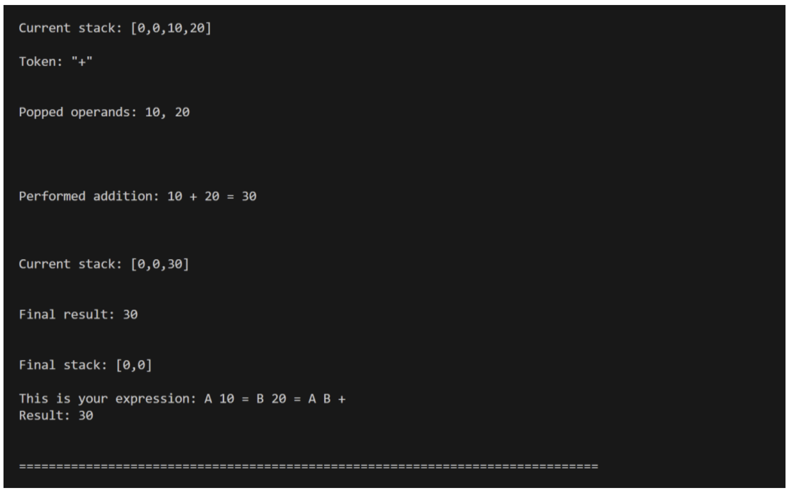
What is a stack?
A stack is a collection of elements with two principal operations: push and pop. Stacks follow the Last In, First Out principle. Pushing is the action of adding an element to the top of the stack. And pop is the action of removing the element at the top of the stack. Following the LIFO principle as mentioned earlier.

Why is the stack good fit for the task?
One of the benefits of using a stack for the task of storing the tokens after expressions have been split into separate tokens is that stacks are good for parsing expressions.

This can come in useful in compilers and calculators for evaluating expressions. This means that the stack data structures are the best fit for this task of taking in the expressions and making sense of the data.

In the context of a postfix expression evaluator this will definitely benefit the execution time of parsing of expressions and returning a result back to the user. In the study of algorithms, time complexity and space complexity plays a huge part in the efficiency of algorithms. The better the time and space complexity the more reliable the algorithm is for calculations such as for a postfix evaluator.

Usage of stack in code implementation:

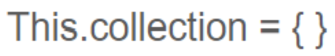


```
Current stack: [0,0,10,20]

Token: "+"


Popped operands: 10, 20



Performed addition: 10 + 20 = 30


Current stack: [0,0,30]


Final result: 30


Final stack: [0,0]

This is your expression: A 10 = B 20 = A B +
Result: 30

==========================================================================
```

Explanation:
This diagram shows an example usage of the stack for the postfix evaluator. As you can see the top of the stack is 20 and after the operator was identified, addition was executed and resulted to be the value 30. Once the result was evaluated the result was returned and printed onto the terminal.

Other data structures that can be used

Object



```
This.collection = { }
```

What is an object?
An object data structure is a collection of properties, where each property is an association between a key and a value. The key is also known as a name or can also be called as an identifier. Values can be of any data type, this also includes other objects and functions. But for this project the object data structure is used to store the symbols as keys and their corresponding values.

Why is the object suitable for its task?
The object in this project is initialised as the variable collection. Objects have 2 main properties, which include storing related data and dynamic properties.

Firstly, storing related data. Objects are used to store related data and functionalities together. This can benefit the storage of symbols and values, as both elements are related to each other and have the same function which is to be stored in the object until the program requests for the particular symbol and returns the assigned value.

Secondly, dynamic properties. For a symbol table an object can be advantageous, as properties in the object can be added, modified, or deleted. The user will have the ability to append the symbol table at any time when the program is running. This makes the object a good fit for this task as it allows the user to have the flexibility to edit the symbol table efficiently without much complications.

Usage in code implementation:

```
1  //import libraries
2  const readline = require('readline');
3
4  //postfix interpreter class
5  class PostFixInterpreter {
6    constructor() {
7
8        //include the collection and stack updateVariable
9        this.collection = {};
10       this.stack = [];
11       this.symbolCounter = 0;
12   }
```

Explanation

At line 9 the collection variable was initialised as an object data structure. When the collection variable is called in the program, its role is to efficiently store the symbols and values in a structured and easily accessible manner

3.6) Code implementation

3.6.1) Implementation

Code snippet 3.6.1.1

```
1  //import libraries
2  const readline = require('readline');
3
4  //postfix interpreter class
5  class PostFixInterpreter {
6    constructor() {
7
8        //include the collection and stack updateVariable
9        this.collection = [];
10       this.stack = [];
11       this.symbolCounter = 0;
12   }
```

Purpose:

The purpose of this code snippet is to set up a basic framework for a postfix interpreter in Javascript. It imports its necessary libraries, drones the postfixIterpreter class, and initialises essential properties such as collection for storing symbols, a stack for expression evaluations, and a symbol counter.

Explanation:

In code snippet **3.6.1.1**, line 2. The readline library is imported and initialised as a readline variable. Readline offers a useful and simple way to interact with the user. At line 5 the postfixInterpreter class is created. The posftfixInterpreter class will contain all the implemented functions such as symbol table search, update and delete. Then lastly, from line 9 to 11, the collection variable is initialised as an object so store the symbols inserted by the user. Then the stack is initialised as an array. The stack is used to store all the tokens that are extracted from the expression inserted by the user. Then the symbol counter is created and assigned a value of -1. This counter is used to count the number of records that are allowed in the symbol table. Given that the memory of the device that this program is stored on has a small memory. This counter is made to ensure that the records are limited.

Code snippet 3.6.1.2

```
//include the evaluatePostfixExp function that receives one parameter
evaluatePostfixExp(inputExp) {
    console.log(`Expression being evaluated: "${inputExp}"`);
    const expTokens = inputExp.split(' ');

    let hasOperator = false;

    for (let i = 0; i < expTokens.length; i++) {
        const token = expTokens[i];
        console.log(`Token: "${token}"`);

        //checks if the token already exists
        if (token.match(/^[A-Z]$/)) {
            const value = this.collection[token];

            if (value === undefined) {
                console.log(`\nSymbol "${token}" not found, pushing 0\n`);
                this.stack.push(0);

            } else {
                //if the symbol has already been found then push that value into the stack
                console.log(`\nSymbol "${token}" found, pushing ${value}\n`);
                this.stack.push(value);
            }

            //checks if there are any arithmetic operators
        } else if (['+', '-', '*', '/', '%', '^'].includes(token)) {
            const operand1 = this.stack.pop();
            const operand2 = this.stack.pop();

            //checks if there are enough operators to evaluate the expression
            if (operand1 === undefined || operand2 === undefined) {
                throw new Error(`\n\nInsufficient operands for operator: ${token}\n\n`);
            }

            console.log(`\n\nPopped operands: ${operand2}, ${operand1}\n\n`);
            hasOperator = true;

            //when the token has been detected then verify what kind of operator it is
            switch (token) {

                //for this case if the addition operator has been inserted then addition will be
                  executed
                case '+':
                    this.stack.push(operand2 + operand1);
                    console.log(`\n\nPerformed addition: ${operand2} + ${operand1} = ${operand2 +
                    operand1}\n\n`);
                    break;

                case '-':
                    this.stack.push(operand2 - operand1);
                    console.log(`\n\nPerformed subtraction: ${operand2} - ${operand1} = ${operand2
                    - operand1}\n\n`);
                    break;

                case '*':
                    this.stack.push(operand2 * operand1);
                    console.log(`\n\nPerformed multiplication: ${operand2} * ${operand1} = ${operand2 * operand1}\n\n`);
                    break;

                case '/':
                    if (operand1 === 0) {
                      throw new Error(`\n\nDivision by zero error\n\n`);
                    }

                    this.stack.push(operand2 / operand1);
                    console.log(`\n\nPerformed division: ${operand2} / ${operand1} = ${operand2 /
                    operand1}\n\n`);
                    break;

                case '%':
                    this.stack.push(operand2 % operand1);
                    console.log(`\n\nPerformed modulus: ${operand2} % ${operand1} = ${operand2
                    % operand1}\n\n`);
                    break;

                case '^':
                    this.stack.push(Math.pow(operand2, operand1));
                    console.log(`\n\nPerformed exponentiation: ${operand2} ^ ${operand1} = ${Math.pow(operand2, operand1)}\n\n`);
                    break;

                default:
                    throw new Error(`\n\nInvalid operator: ${token}\n\n`);
            }

        //this checks if the user only requests to update a value
        } else if (token === '=') {

            const updateVariable = expTokens[i - 2];
            const value = this.stack.pop();
            this.updateSymbolTable(updateVariable, value);

            //if the variable that was selected has been updated successfully
            console.log(`${updateVariable} has been updated successfully!`);
        } else {

            const num = parseFloat(token);
            if (!isNaN(num) && this.isValidNumber(num)) {
                this.stack.push(num);
                console.log(`\nPushed number: ${num}\n`);

            } else {

                  //this verifies if the number that has been inserted is between -100 and 100 value range
                  console.log(`\nInvalid number: ${token}. Only values between -100 and 100 are allowed.\n`);
            }
        }

        //returns the current values in the stack
        console.log(`\nCurrent stack: ${JSON.stringify(this.stack)}\n`);
      }

```

```
116       const result = this.stack.pop();
117
118      //if the operator is present then return the final result of the expression
119      if (hasOperator) {
120          console.log(`\nFinal result: ${result}\n`);
121      }
122      console.log(`\nFinal stack: ${JSON.stringify(this.stack)}\n`);
123
124      return result;
125   }
```

Purpose:
The purpose of this code snippet is to define the evaluatePostfixExp function within the PostFixInterpreter class. This function evaluates a given postfix expression, handles variables, performs arithmetic operations, and updates a symbol table if necessary.

Explanation:
At code snippet **3.6.1.2**, line 4 the expression entered by the user is split by spaces and stored into the variable expTokens. At line 13 to 24 the token is verified whether it is within the range of uppercase format. If the token is found then it pushes its value together with the letter. If no value is found the a 0 value is pushed. At line 27 to 86, the evaluation of operands and operators happens and returns a result at the end by popping it from the stack. At line 89 to 96. It checks if the user just wants to update a value or assign a value. If that's what the user wants to do then an evaluation result will not appear, only a success message notifying the user that the update or assignment has been done.

Code snippet 3.6.1.3

```
1  //this function updates existing variables inserted by the user
2     updateSymbolTable(symbol, value) {
3
4         //verifies that the variable that the user has inserted is from A to Z
5         if (/^[A-Z]$/.test(symbol)) {
6
7             //this verifies if the value that was inserted is between -100 and 100 by calling the
isValidNumber function
9             if (this.isValidNumber(value)) {
10
11                 if(this.symbolCounter < 1000){
12
13                     this.collection[symbol] = value;
14                     this.symbolCounter++;
15
16                     //returns a print to display that the new value has been inserted into the table
17                     console.log(`\nUpdate status: \nAssigned: ${symbol} = ${value} successfully.`);
18                     console.log(`\nCurrent size of symbol table is " ${this.symbolCounter} "`);
19                 }else{
20                     console.log("\nUpdate status: Update failed.");
21                     console.log(`The symbol table is full, the current number of symbols in the table are
" ${this.symbolCounter} "`);
23                 }
24             }
25             else {
```

```
26              console.log(`\nInvalid value: ${value}. Only values between -100 and 100 are
allowed.\n`);
28            }
29        } else {
30
31            //checks that the variable that was entered is A to Z
32            console.log(`\nInvalid variable name: ${symbol}. Only letters A-Z are allowed.\n`);
33        }
34    }
```

Purpose:
The purpose of this code snippet is to define the updateSymbolTable function within the postfixInterpreter class. This function updates the value of a variable in the symbol table, ensuring the variable name and value adhere to the specific constraints.

Explanation:
In snippet **3.6.1.3**, line 5. It checks if the symbol is within A to Z. At line 9, it checks if the value is within -100 and 100. At line 11. There is a check to see if the records are below 1000. If all the checks succeed and there are no issue then the counter will increase by 1 and the new record is updated.

Code snippet 3.6.1.4

```
1  //this function deletes existing variables that the user has entered previously
2     deleteSymbol(symbol) {
3        if (this.collection.hasOwnProperty(symbol)) {
4           delete this.collection[symbol];
5           this.symbolCounter--;
6           console.log(`\nDelete status: \n${symbol} has been deleted successfully.\n`);
7           console.log(`The current number of symbols in the table are " ${this.symbolCounter} "`);
8        } else {
9           console.log(`\nDelete status: \nSymbol "${symbol}" not found. Delete operation failed.\n`);
10       }
11    }
12
```

Purpose:
The purpose of this code snippet is to define the deleteSymbol function within the PostFixInterpreter class. This function deletes a variable from the symbol table if it exists, updating the symbol counter accordingly.

Explanation:
In code snippet **3.6.1.4**, line 3. The function checks if the symbol exists in the symbol table. In line 4, the symbol is deleted from the table. In line 5 the symbolCounter is decreased by one indicating that the deletion was successful.

Code snippet 3.6.1.5

```
1  //this function allows the user to search for existing symbols in the table
2     searchSymbolHash(symbol) {
3        if (this.collection.hasOwnProperty(symbol)) {
4           console.log("\nSearch status: \nSymbol has been found successfully.");
5           return this.collection[symbol];
6
7        } else {
8           return "\nSearch status: \nSymbol was not found. Search operation has failed\n";
9        }
10    }
```

Purpose:

The purpose of this code snippet is to define the searchSymbolHash function within the PostFixInterpreter class. This function allows the users to search for exiting symbols in the symbol table and returns the corresponding value if found.

Explanation:

In code snippet **3.6.1.5**, line 3. The function checks if the symbol inserted exists in the symbol table. In line 5. The value and the symbol is returned and it is printed onto the terminal.

Code snippet 3.6.1.6

```
1  //this function verifies whether the input from the user for variable values are between -100 and
100
3     isValidNumber(num) {
4         return num >= -100 && num <= 100;
5     }
6  }
```

Purpose:

The purpose of this code snippet is to check if the value inserted is within the range of -100 to 100.

Explanation:

In code snippet **3.6.1.6**, line 4. The verification is done to ensure that the value is within the range.

Code snippet 3.6.1.7

```
1  //this assigns the postfixinterpreter class to a variable
2  const interpreter = new PostFixInterpreter();
3
4  //this assigns the userInput variable to be able to take in user input in the console
5  const userInput = readline.createInterface({
6     input: process.stdin,
7     output: process.stdout
8  });
9
```

Purpose:

The purpose of this code snippet is to create an instance of the PostFixInterpreter class and set up a mechanism for taking user input from the console.

Explanation:

In code snippet **3.6.1.7**, line 2. The interpreter variable is initialised as a new PostFixInterpreter class. At line 5 to 8. The interface is created to receive user input and also to print out text onto the console.

Code snippet 3.6.1.8

```
1  //this function retrieves the user input
2  function getUserInput() {
3
4     //this prints is created to have a user friendly GUI for easy and efficient access to the postfix
5program features
6
7console.log("==========================================================
8=============");
9     console.log("\nWelcome to my postfix evaluator application! \n\nHere are a list of items that
10you can do with this application \n");
11    console.log("\nMenu: \n\n(1) Postfix expression evaluator \n(2) View symbol table \n(3) Search
12symbol table \n(4) Update symbol table \n(5) Delete symbol table \n(6) Help \n(7) About postfix
13\n(8) Exit \n");
14
15console.log("==========================================================
16==============\n\n");
```

Purpose:

The purpose of this code snippet is to define the getUserInput function, which provides a user-friendly interface for accessing the features of the postfix evaluator application by displaying a menu with options.

Explanation:
In code snippet **3.6.1.8**, line 2. The getUserInput function is created. This function provides the user a friendly user interface to interact with the program without much issues.

Code snippet 3.6.1.9

```
//from the menu provided to the user a selection is inserted into the command line
  userInput.question('Please enter your selection here: ', (selection) => {

    //depending on what the user has selected the program will respond accordingly
    switch (selection) {
      case '1':

        //some users do not know how to enter a postfix expression so the program will ask if the user knows how to enter
        userInput.question('Do you know how to enter a postfix expression?\n\nPlease enter (yes / no) : ', (postfixBool) => {
          if (postfixBool === "yes") {
            console.log("\nThe available operators are ' + , - , * , / , % , ^ '\n");
            userInput.question('Enter a postfix expression (or "exit" to quit): ', (inputExp) => {
              if (inputExp !== "exit") {
                try {
                  const result = interpreter.evaluatePostfixExp(inputExp);
                  if (inputExp.includes('+') || inputExp.includes('-') || inputExp.includes('*') || inputExp.includes('/') || inputExp.includes('%') || inputExp.includes('^')) {
                    console.log(`This is your expression: ${inputExp}`);
                    console.log(`Result: ${result} \n\n`);
                  }
                } catch (error) {
                  console.log(error.message);
                }
              }
              getUserInput(); // Return to the main menu
            });
          } else if (postfixBool === "no") {
            console.log("\nPlease refer to the help page from the menu selection.");
            getUserInput(); // Display the menu
          } else {
            console.log("\nPlease enter a valid value! Please try again.");
            getUserInput();
          }
        });
        break;
```

Purpose:
The purpose of this code snippet is to handle user input based on their menu selection, guiding them through entering a postfix expression and evaluating it or providing relevant help if needed.
Explanation:

In code snippet **3.6.1.9**, line 5. The switch case is created to receive user input. In line 6, if the user select 1 then the program will go to the next stage. In line 8 to 28, the program will use the expression inserted and do the necessary evaluation of the postfix notation. In line 29 to 34. If the user does not know how to insert a postfix expression the user can refer to the help page. If any other value is inserted then an error message is returned.

Code snippet 3.6.1.10

```
1 //view existing symbol table
2           case '2':
3              console.log("\nSymbol table: ");
4              for(const [symbol, value] of Object.entries(interpreter.collection)){
5                 console.log(`{"${symbol} = ${value}"}`);
6              }
7              console.log("\n\n")
8              getUserInput();
9              break;
```

Purpose: The purpose of this code snippet is to print out all the available symbols in the symbol table.

Explanation: In code snippet **3.6.1.10**, line 2 to 7. If the user entered 2, the symbol table and menu is printed

Code snippet 3.6.1.11

```
1  //search for existing symbols
2           case '3':
3              userInput.question('Enter the symbol to search: ', (symbol) => {
4                 const result = interpreter.searchSymbolHash(symbol);
5                 console.log(`\nSearch result: \nSymbol: ${symbol} \nValue: ${result} \n\n`);
6                 getUserInput();
7              });
8              break;
9
```

Purpose

The purpose of this code snippet is to allow the user to search for a symbol in the symbol table and display the result. It handles the interaction with the user, performs the search, and displays the search result.

Explanation

In code snippet **3.6.1.11**, line 2. It defines a case where the user inserts a 3 into the terminal. In line 3, The user is asked to enter the symbol that the user wants to search from the symbol table. In line 4, the program uses the symbol that the user has inserted and calls the search function in the interpreter class to search for the symbol. In line 5, the search result is printed. The search result contains both the symbol inserted and also the value that the symbol is associated with. And lastly, in line 6. The menu is printed back to the user.

Code snippet 3.6.1.12

```
1   //update existing symbols in the table
2           case '4':
3              userInput.question('Enter the symbol and value to update (e.g., A 10): ', (input) => {
4                 const [symbol, value] = input.split(' ');
5                 interpreter.updateSymbolTable(symbol, parseFloat(value));
6                 getUserInput();
7              });
8              break;
```

Purpose
The purpose of this code is to allow the user to update an existing symbol in the symbol table with a new value. It handles the interaction with the user, performs the update, and ensures the program continues to prompt for further actions.

Explanation
In code snippet **3.6.1.12**, line 2. It defines the case where the user inserts a 4 into the terminal. In line 3, the user is prompted to insert the symbol and new value to be updated. The format for the update has to be in the correct format in order for the symbol to be updated with the new value. In Line 5, the update function is called from the interpreter class. It takes in two parameters, the symbol and the new value. And lastly, in line 6. The menu is printed back to the user.

Code snippet 3.6.1.13

```
1  //delete existing symbols in the table
2              case '5':
3                  userInput.question('Enter the symbol to delete: ', (symbol) => {
4                      interpreter.deleteSymbol(symbol);
5                      getUserInput();
6                  });
7                  break;
8
```

Purpose
The purpose of this code snippet is to allow the user to delete a symbol from the symbol table. It handles the interaction with the user, performs the deletion, and ensures that the program continues to prompt for further actions.

Explanation
In code snippet **3.6.1.13**, line 2. It defines the case where the user has inserted the value 5 into the terminal. In line 3, the user is prompted to enter a symbol that the user wants to remove from the symbol table. In line 4, the delete function is called from the interpreter class. It takes in one parameter. This parameter that the function takes in is the symbol that the user wants to remove from the table. In line 5, the menu is printed back to the user.

Code snippet 3.6.1.14

```
1  //help page
2              case '6':
3                  console.log("\nHelp page\n\nPostfix expression can be entered like this: \n" +
4                              "' A 10 = B 20 = A B + '\n\n" +
5                              "If you wanted to update your value in the symbol table you can insert as such:
6  \n" +
7                              "' A 20 '\n\n");
8                  getUserInput();
9                  break;
10
```

Purpose

The purpose of this code is to display a help page to the user, providing guidance on how to enter postfix expressions and update values in the symbol table. It ensures the user understand how to use the system.

Explanation
In code snippet **3.6.1.14**, line 2. It defines a case where the user inserts the value 6 into the terminal. From line 3 to 7. It provides useful information for the user to enter the postfix expression into the terminal. For this there are two cases. Firstly, the user can insert a postfix expression for evaluation with arithmetic operators for calculation. And the second case is that the user just wants to assign the symbol with a value and no calculations are carried out. This reduces the run time as the program does not have to do additional calculations on a expression that does not involve any calculations like in this example of just assigning a symbol with a value.

Code snippet 3.6.1.15

```
1  //allow the user know about postfix program
2           case '7':
3               console.log("\nWhat is Postfix?\n\n"+
4               "Postfix is a Reverse Polish notation (RPN), or some might refer to it as Polish postfix 5
notation.\n"+
6               "It is a mathematical notation in which operators follow their operands, as compared to
7  prefix or Polish notation\n"+
8               "In which operators precede their operands. The notation does not need any
9parentheses for as long as each operator\n"+
10              "has a fixed number of operands. The term postfix notation desribes the general
11scheme in computer sciences.\n\n\n"+
12              "Explanation :\n\n"+
13              "In reverse polish notation, the operators follow their operands. For example, if you
14wanted to add 3 to 4, the expression is :\n\n"+
15              "'3 4 +'\n\n"+
16              "As compared to the infix notation '3 + 4'. The conventional notation expression '3 - 4 +
17  5' is inserted as '3 4 - 5 +' in\n "+
18              "reverse polish notation. 4 is first subtracted from 3. then 5 is added to it.\n\n");
19              getUserInput();
20              break;
21
```

Purpose
The purpose of this code is to educate the user about postfix notation and explain how it works through examples. It ensures the user understands the concept of postfix notation and how to use it in application.

Explanation
In code snippet **3.6.1.15**, line 2. It defines a case where the user inserts the value 7 into the terminal. From line 3 to 18. It is meant to educate the user on the concepts of postfix notation and expressions. It provides a detailed documentation of what goes on with postfix, and also includes examples of postfix expressions so as to ensure that the user knows how to insert a postfix expression. In line 19. The menu is printed back to the user.

Code snippet 3.6.1.16

```
1  //exit and end the program session
2            case '8':
3               console.log("\nThank you for using my application, have a nice day! \n\n");
4               userInput.close();
5               break;
6
```

Purpose

The purpose of this code is to gracefully exit and end the program session, providing a polite message to the user before terminating the interaction.

Explanation

In code snippet **3.6.1.16**, line 2. It defines a case where the user inserts the value 8 into the terminal. In line 3, an exit message is printed to the user to notify the user that the user has exited from the program successfully. In line 4, the program exits and the session ends. In this case the menu will not be printed back to the user as the session has ended and the user has exited from the program.

Code snippet 3.6.1.17

```
1  //if the user has inserted a value that is not between 1 to 8
2            default:
3               console.log("\nInvalid choice. Please try again. \n\n");
4               getUserInput();
5               break;
6
```

Purpose

The purpose of this code is to handle invalid user inputs that do not fall within the specified range of 1 to 8 in the menu provided to the user. If the user inserts a value that is not within this range a error message is returned to the user and is prompted to try again.

Explanation

In code snippet **3.6.1.17**, line 2. It defines the default case for the switch case implementation for the user menu. In line 3, the user is prompted with an error message. Notifying the user that the value inserted is an invalid choice. In line 4, the menu is printed back to the user. This is so that the user is able to enter another choice. This helps the program to continue to run even if there are incorrect values inserted into the terminal.

Code snippet 3.6.1.18

```
1  getUserInput();
```

Purpose

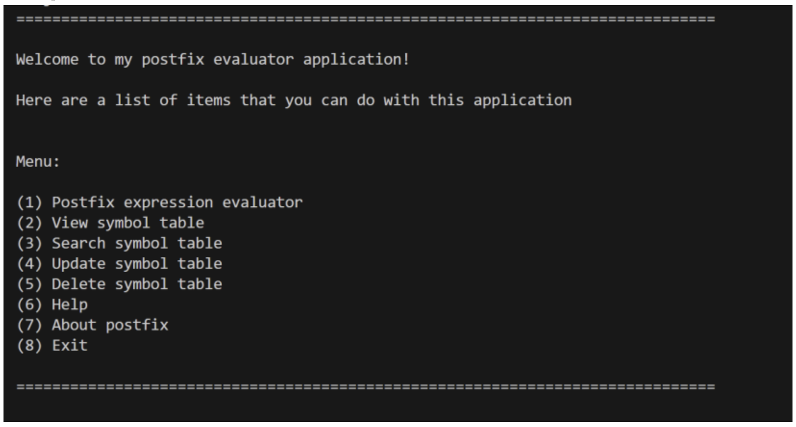
The purpose of this code is to call the user input function. This function will receive user input based on the menu provided to the user. Based on the user input the program will react accordingly.

Explanation

In code snippet **3.6.1.18**, line 1. The getUserInput function is called. This function allows the user to insert a value into the terminal from a given set of choices from the menu. Based on the selection from the user the program will react accordingly. For example, if the user were to select the help selection, then the program will provide the user with the necessary information to insert a postfix expression into the terminal.

3.6.2) Output

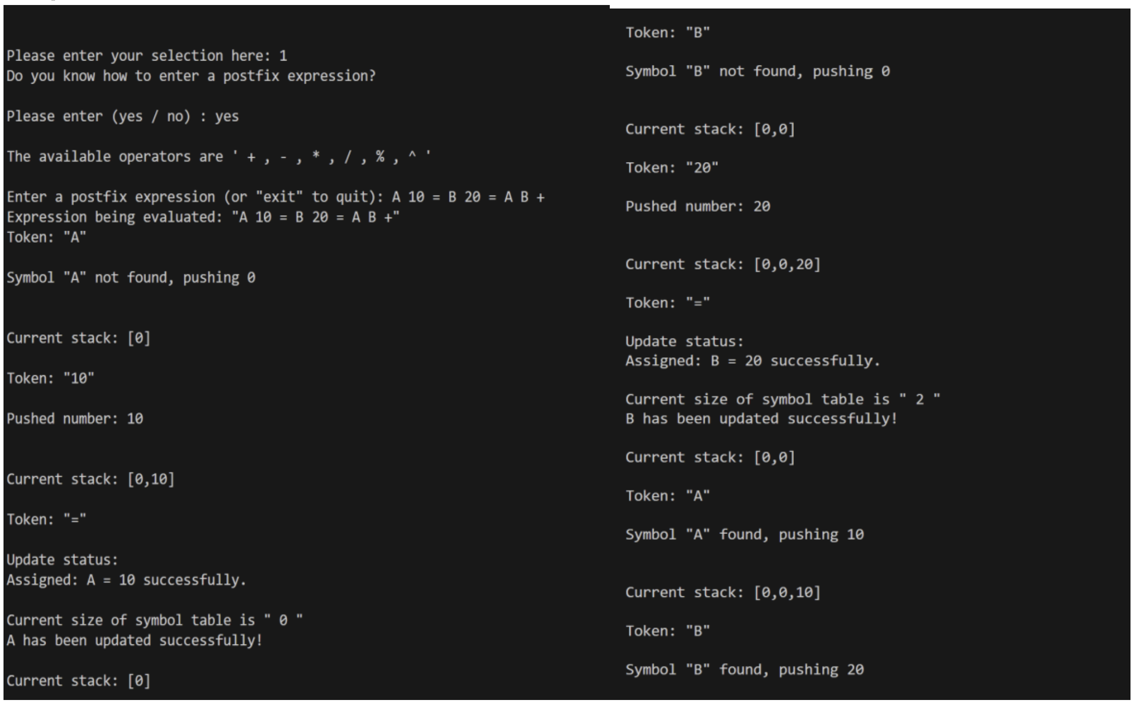
Output 3.6.2.1



```
===============================================================================

Welcome to my postfix evaluator application!

Here are a list of items that you can do with this application


Menu:

(1) Postfix expression evaluator
(2) View symbol table
(3) Search symbol table
(4) Update symbol table
(5) Delete symbol table
(6) Help
(7) About postfix
(8) Exit

===============================================================================
```

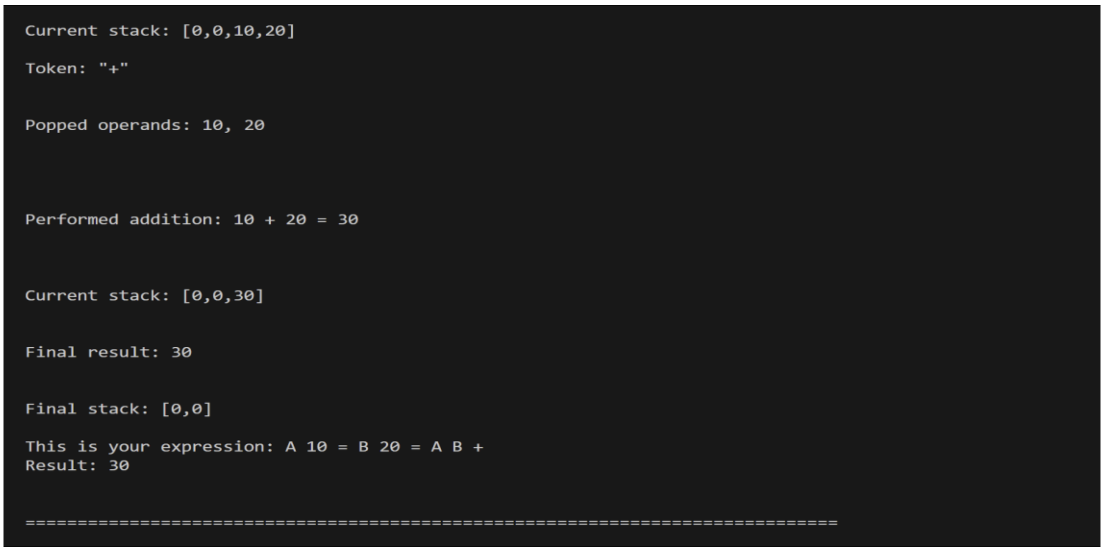
Purpose
The purpose of this output is to provide the user with a user friendly menu to select from.

Explanation
In output **3.6.2.1**, it shows a friendly user menu that first welcomes the user to use the application. Then a menu is provided to the user. The menu has 8 selections. These 8 selections include, postfix expression evaluator, view symbol table, search symbol table, update symbol table, delete symbol table, help, about postfix and exit.
Output 3.6.2.2



```

Please enter your selection here: 1
Do you know how to enter a postfix expression?

Please enter (yes / no) : yes

The available operators are ' + , - , * , / , % , ^ '

Enter a postfix expression (or "exit" to quit): A 10 = B 20 = A B +
Expression being evaluated: "A 10 = B 20 = A B +"
Token: "A"

Symbol "A" not found, pushing 0


Current stack: [0]

Token: "10"

Pushed number: 10


Current stack: [0,10]

Token: "="

Update status:
Assigned: A = 10 successfully.

Current size of symbol table is " 0 "
A has been updated successfully!

Current stack: [0]
```

```
Token: "B"

Symbol "B" not found, pushing 0


Current stack: [0,0]

Token: "20"

Pushed number: 20


Current stack: [0,0,20]

Token: "="

Update status:
Assigned: B = 20 successfully.

Current size of symbol table is " 2 "
B has been updated successfully!

Current stack: [0,0]

Token: "A"

Symbol "A" found, pushing 10


Current stack: [0,0,10]

Token: "B"

Symbol "B" found, pushing 20
```

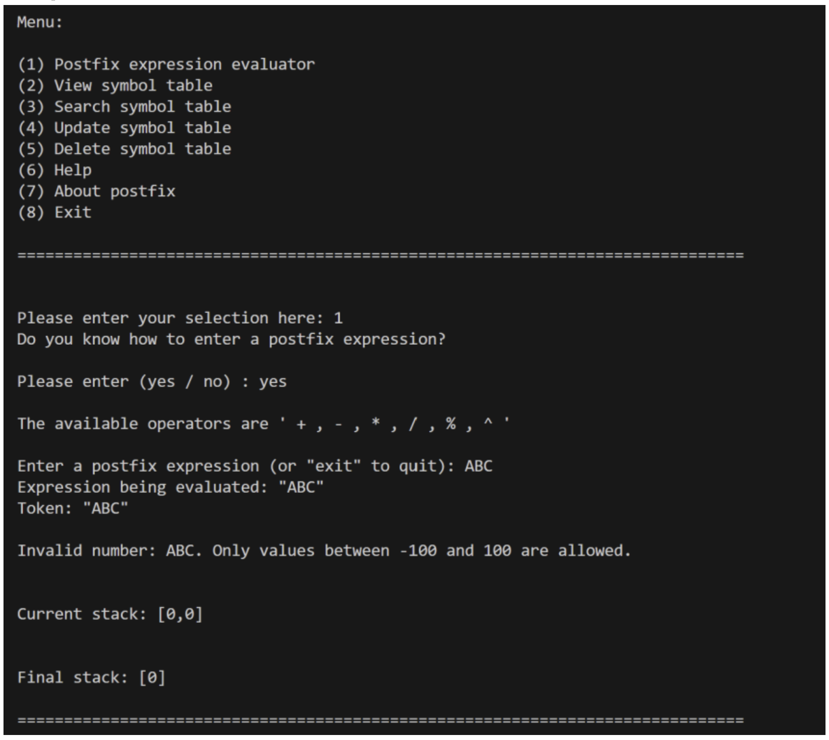


```
Current stack: [0,0,10,20]

Token: "+"


Popped operands: 10, 20



Performed addition: 10 + 20 = 30


Current stack: [0,0,30]


Final result: 30


Final stack: [0,0]

This is your expression: A 10 = B 20 = A B +
Result: 30


==============================================================================
```

Purpose
The purpose of this output is to display the step by step process of how the tokens are extracted from the postfix expression. This output provides a detailed documentation of how each symbol is assigned with its associated value. And after assignment, it evaluates both values with the arithmetic operator extracted from the expression.

Explanation
In output **3.6.2.2**, the symbol A is found. Initially the symbol does not have a value assigned to it, so a value 0 is assigned to it. The program continues to read the expression. It finds that 10 is assigned to symbol A. This can be understood by the program as the “ = ” sign is read. This means that the value 10 is understood to be associated with the symbol A. This is done with the symbol B as well. After each value has been assigned, the symbols are pushed into the stack in the form of its values. Now with the updated stack. The operator is identified at the end of the expression and the two values are added to each other in this case. After the addition operation is done, it returns the final result to the stack. Since the calculation is done, the final result is printed back to the terminal. This results with the value 30 from the addition of 10 and 20.

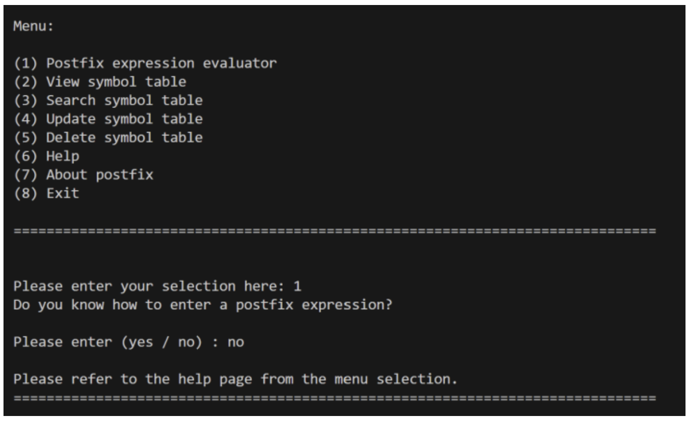
Output 3.6.2.3



```
Menu:

(1) Postfix expression evaluator
(2) View symbol table
(3) Search symbol table
(4) Update symbol table
(5) Delete symbol table
(6) Help
(7) About postfix
(8) Exit

==============================================================================


Please enter your selection here: 1
Do you know how to enter a postfix expression?

Please enter (yes / no) : yes

The available operators are ' + , - , * , / , % , ^ '

Enter a postfix expression (or "exit" to quit): ABC
Expression being evaluated: "ABC"
Token: "ABC"

Invalid number: ABC. Only values between -100 and 100 are allowed.


Current stack: [0,0]


Final stack: [0]

==============================================================================
```

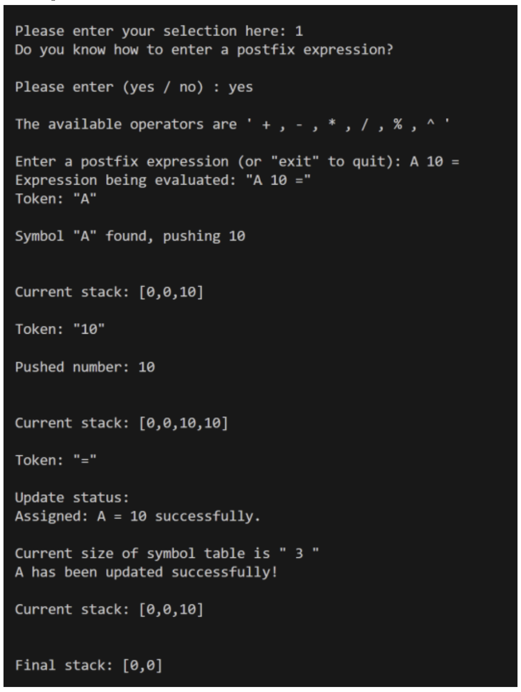
Purpose
The purpose of this output is in the case that the user has entered a “yes” value into the terminal, declaring that the user knows how to enter a postfix expression but enters an invalid value. In this case the program will return an error message to the user notifying the user that an invalid value has been entered. And for the purpose of the stack display, is to document that there are no values pushed into the stack due to the lack of a valid value.
Explanation
In output **3.6.2.3**, the user inserts the value 1 into the terminal. This indicates that the user wants to evaluate a postfix expression. The user is then prompted to enter either a yes or no to indicate whether the user knows how to insert a postfix expression. If the user inserts a “yes” value then the program will prompt the user to enter the postfix expression and also provides the user with all the available operators for evaluation. If the user does not know how to insert a postfix expression the program recommends the user to refer to the help page. And prints the menu back to the user. In the case where the user has entered an invalid value such as “ABC” into the terminal then the program will return an error message back to the user. Notifying the user that the value is invalid and does not match the specified range of values. The range includes symbols in A to Z and -100 to 100 for values. At the end of this output, the empty stack is printed to document that no values was pushed into the stack.

Output 3.6.2.4



```
Menu:

(1) Postfix expression evaluator
(2) View symbol table
(3) Search symbol table
(4) Update symbol table
(5) Delete symbol table
(6) Help
(7) About postfix
(8) Exit

=============================================================================


Please enter your selection here: 1
Do you know how to enter a postfix expression?

Please enter (yes / no) : no

Please refer to the help page from the menu selection.
=============================================================================
```

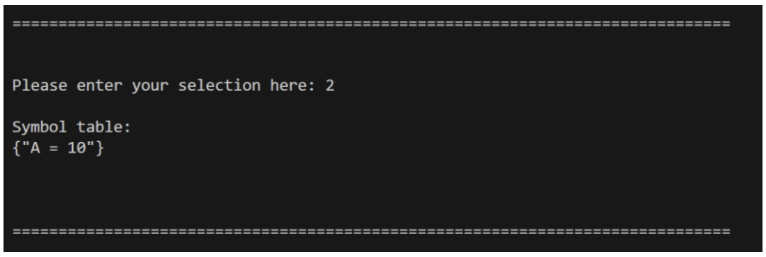
Purpose
The purpose of this output is to allow the user to refer to the help page to insert a postfix expression.
Explanation
In output **3.6.2.4**. The user inserts the value 1 into the terminal. This indicates that the user is requesting to insert a postfix expression for evaluation. Then upon selecting the value 1, the user is prompted to answer either yes or no to indicate whether the user knows how to insert a postfix expression. In this case the user has entered no into the terminal. The user is then notified to refer to the help page for further instructions about postfix notation.
Output 3.6.2.5



```
Please enter your selection here: 1
Do you know how to enter a postfix expression?

Please enter (yes / no) : yes

The available operators are ' + , - , * , / , % , ^ '

Enter a postfix expression (or "exit" to quit): A 10 =
Expression being evaluated: "A 10 ="
Token: "A"

Symbol "A" found, pushing 10


Current stack: [0,0,10]

Token: "10"

Pushed number: 10


Current stack: [0,0,10,10]

Token: "="

Update status:
Assigned: A = 10 successfully.

Current size of symbol table is " 3 "
A has been updated successfully!

Current stack: [0,0,10]


Final stack: [0,0]
```

Purpose
The purpose of this output is to show a step by step process of how an expression to assign a value is done in this program.
Explanation
In output **3.6.2.5**. The user requests to enter a postfix expression. But the user only wants to assign a value and not to insert an expression for calculation. In this case, the user will enter the symbol followed by the value and lastly the "=" sign. Initially the symbol is identified but does not have a value, so a 0 value is assigned to it first. Then the program identifies that there is a number associated with the symbol. And to confirm this assignment, the program will read the "=" sign. This indicates that the user wants to assign the value to this particular symbol.

Output 3.6.2.6

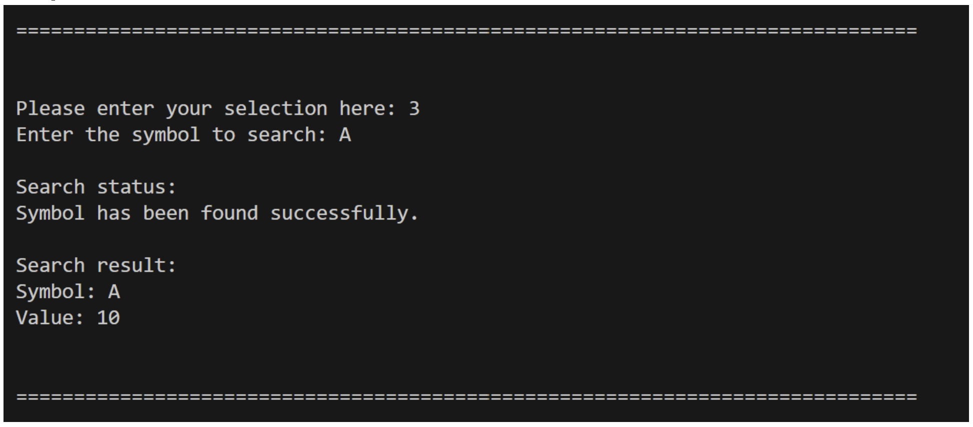


```
==========================================================================


Please enter your selection here: 2

Symbol table:
{"A = 10"}



==========================================================================
```

Purpose
The purpose of this output is to show the user all the available symbols together with its associated value in the symbol table.

Explanation
In output **3.6.2.6**. The user inserts a value 2 into the terminal. This indicates that the user wants to view the symbol table. After this value 2 has been inserted, the program returns all the available symbols back to the user by printing the text symbol table. And with this title, symbol table, the user will know that this is the symbol table. After this title has been printed the available symbols are printed onto the terminal. This provides the user with a quick way to be informed of the available symbols in the symbol table without much complications.

Output 3.6.2.7

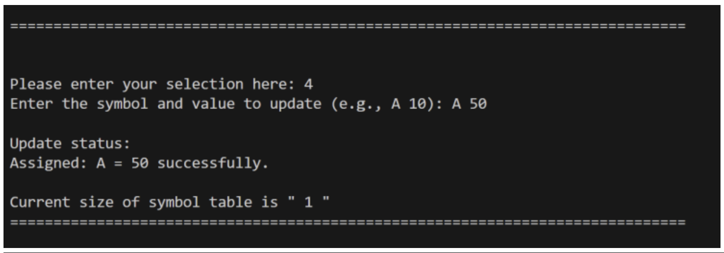


```
==============================================================================


Please enter your selection here: 3
Enter the symbol to search: A

Search status:
Symbol has been found successfully.

Search result:
Symbol: A
Value: 10


==============================================================================
```

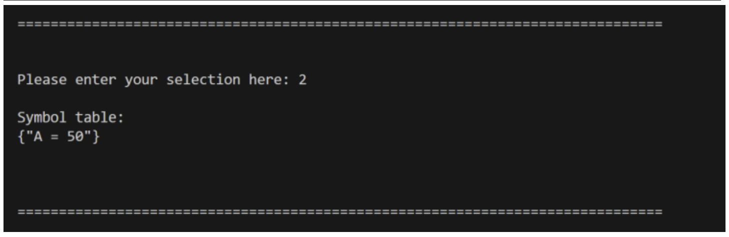
Purpose
The purpose of this output is to allow the user to insert an existing value into the terminal in order to retrieve its associated value.

Explanation
In output **3.6.2.7**. The user inserts the value 3. This indicates that the user is requesting to search for an existing symbol. The program will then take this inserted symbol and return a success message to notify the user that the symbol has been found. And lastly, the search result returns the symbol and its associated value. This provides the user with a seamless route to access existing symbol and its values.

Output 3.6.2.8



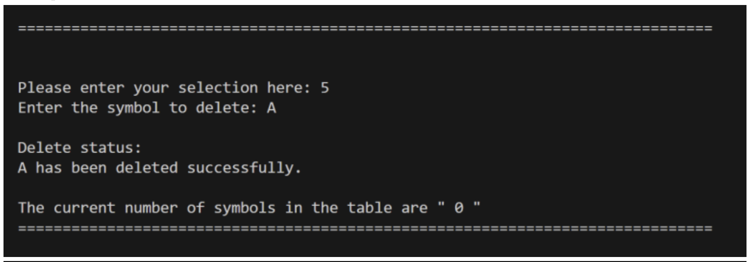
```
=============================================================================


Please enter your selection here: 4
Enter the symbol and value to update (e.g., A 10): A 50

Update status:
Assigned: A = 50 successfully.

Current size of symbol table is " 1 "
=============================================================================
```



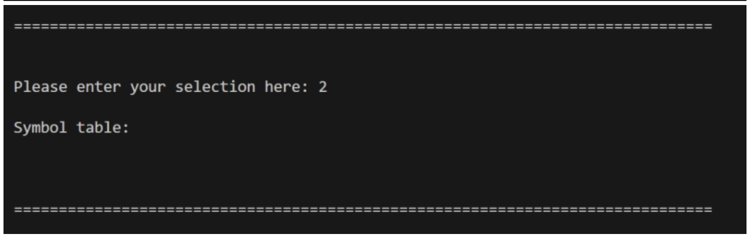
```
=========================================================================


Please enter your selection here: 2

Symbol table:
{"A = 50"}



=========================================================================
```

Purpose
The purpose of this output is to show how the user can update an existing value in the symbol table.
Explanation
In output **3.6.2.8**, the user inserts the value 4. This indicates that the user wants to update an existing value. The user is prompted to insert the existing symbol followed by the new value. The prompt also provides an example of how the format of expression should be inserted. Then the user will be prompted that the symbol has been updated successfully. Then the symbol table size is printed to show the user that the number of records in the table is 1. This is due to the limited size of the device that I have limited the number of records to be 1000. After this operation the user will be able to view the updated value by selecting value 2 from the user menu.

Output 3.6.2.9

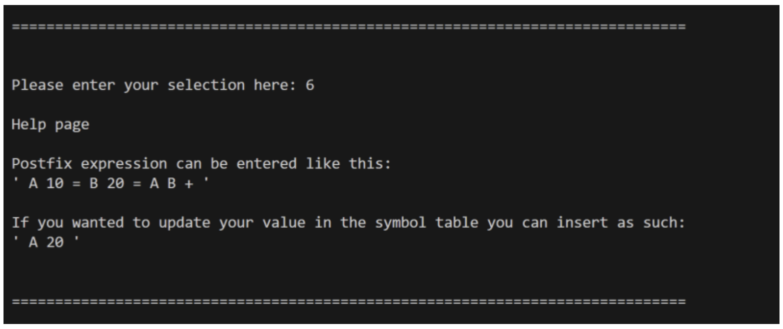


```
=============================================================================


Please enter your selection here: 5
Enter the symbol to delete: A

Delete status:
A has been deleted successfully.

The current number of symbols in the table are " 0 "
=============================================================================
```



```
=============================================================================


Please enter your selection here: 2

Symbol table:



=============================================================================
```

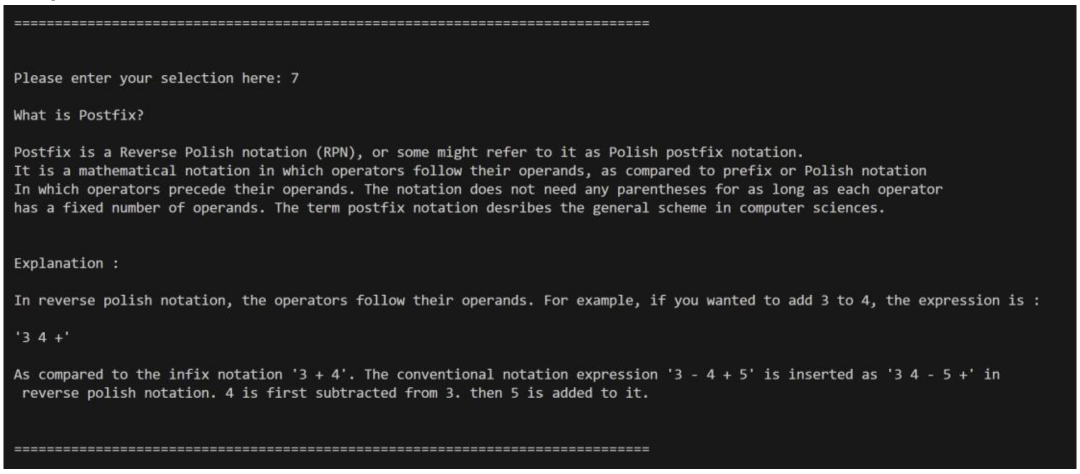
Purpose
The purpose of this output is to show how the user deletes an existing symbol from the symbol table.
Explanation
In output **3.6.2.9**, the user inserts the value 5 into the terminal. This indicates that the user is requesting to delete an existing symbol. Upon selecting 5, the user is prompted to enter the symbol that the user wants to delete. If the symbol has been found then the user is prompted that the symbol has been deleted. Then at the end of the operation the counter for the number of records is updated and printed back to the user to document how many records are left in the symbol table. The user will also be able to check the symbol table by selecting 2 from the user menu. As you can see after deletion the symbol table is empty.
Output 3.6.2.10



```
==============================================================================


Please enter your selection here: 6

Help page

Postfix expression can be entered like this:
' A 10 = B 20 = A B + '

If you wanted to update your value in the symbol table you can insert as such:
' A 20 '


==============================================================================
```

Purpose
The purpose of this output is to provide the user with some essential information on how to insert a postfix expression. Although this was already stated in the option 7 with the description of postfix, this provides a more focused example of postfix notation.
Explanation
In output **3.6.2.10**, the user inserts a value 6 into the terminal. This indicates that the user is requesting to view the help page. This means that the user requires some help to insert a postfix expression. The documentation provides 2 cases. The first case is if the user wants to insert a postfix expression for calculations. And the second case is where the user just wants to update the existing symbol in the table.

Output 3.6.2.11

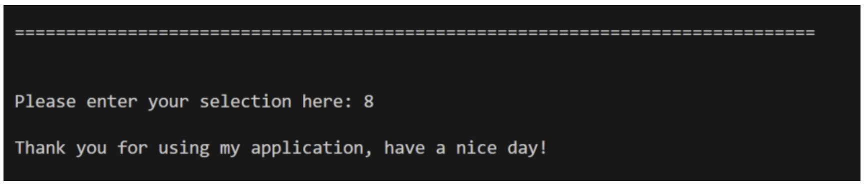


```
==============================================================================


Please enter your selection here: 7

What is Postfix?

Postfix is a Reverse Polish notation (RPN), or some might refer to it as Polish postfix notation.
It is a mathematical notation in which operators follow their operands, as compared to prefix or Polish notation
In which operators precede their operands. The notation does not need any parentheses for as long as each operator
has a fixed number of operands. The term postfix notation desribes the general scheme in computer sciences.


Explanation :

In reverse polish notation, the operators follow their operands. For example, if you wanted to add 3 to 4, the expression is :

'3 4 +'

As compared to the infix notation '3 + 4'. The conventional notation expression '3 - 4 + 5' is inserted as '3 4 - 5 +' in
 reverse polish notation. 4 is first subtracted from 3. then 5 is added to it.


==============================================================================
```

Purpose
The purpose of this output is to provide the user with some background of what postfix is and how the notation is structured. This provides an informative experience for the user. I felt that for this evaluator it would be a better experience for the user to know what is postfix. And after knowing about the background the user will have a better appreciation and understanding when inserting postfix expressions for calculation.

Explanation
In output **3.6.2.11**. The user inserts the value 7 into the terminal. This indicates that the user is requesting to know more about postfix. The user is then provided with a general documentation of what is postfix and also provided with some examples of postfix notations. A comparison to the widely used infix notation is also provided to the user to know the difference between the two notations.

Output 3.6.2.12



```
===============================================================================


Please enter your selection here: 8

Thank you for using my application, have a nice day!
```

Purpose
The purpose of this output is to allow the user to exit the program and end the session. This provides a well rounded program from start to finish. This provides the user with the flexibility to end the session at any time. This also provides the user with a friendly user interface.
Explanation
In output **3.6.2.12**, the user inserts a value 8 into the terminal. This indicates that the user wants to exit the program and end the session. Upon selecting the value 8, the user is then prompted with a message to notify the user that the program has ended and wishes the user a nice ending message.

Program Demonstration Video Link:

https://youtu.be/DlqlmJzjHyg

3.7) Defects and improvements in implementation

Error handling
The implementation that I used for error handling was too vague and did not provide sufficient information for the user. This can be a disadvantage, as this might affect the usability of my application by potential users. I could have provided more example postfix notations and better redirection instructions in the case that the user has inserted a invalid value into the terminal.

In my implementation I also used an if else statement for error handling. This is wrong to a small extent, as usually a try catch block is used for error handling. Although the if else statement did work for my application. The technically correct method to carry out error handling should be done using a try catch block.

Magic numbers
The implementation uses magic numbers. For example, 1000 records. Without explaining its significance. This limitation was considered for the limited space of the device. But this was not notified to the user.

I decided not to notify the user of this as I observed that this is a product specification and does not provide the user with much functionality. I decided that the user will be notified when the user has already reached the limit of storage.

This can be shown in examples such as symbols limited to A to Z characters and values limited to -100 to 100. With this limitation and lack of information, I feel that this may hinder the experience of some of the users that want to store a lot of values.

Code organisation

The code is not well-organised, with some functions performing multiple, unrelated tasks. This can become a disadvantage for other developers to append my implementations in the future. For projects to grow and for multiple people to work on the same project, the structure of code is important. This allows for other developers to be able to easily understand and read your implementation.

This can affect the handover time if this was a bigger project. As newly informed developers might not know the functionality behind the implementations and may require a longer time to understand the flow and outcome of the program.

Code organisation is important as it may also sometimes affect the run time of algorithms. The efficiency and effectiveness of an algorithm is important as this directly impacts the reliability and ability to do calculations and evaluations.

What can I do to improve on these defects on my implementations?
Error handling
Implement try-catch blocks to handle errors and exceptions instead of using if else statements. Use error codes or error messages to provide more informative error handling. This to inform the users about the nature of the error and some possible solutions to the issue. For example, if the user has inserted a wrong value, the program will be able to redirect the user to a section of the implementation to get this information.

In my implementation I did do a small redirection for the user. But, some users require more information and details to use the application. To achieve the aim of making this application user friendly, I need to take account of all the different types of users. With effective error messages and error handling, it helps to build a more robust program that provides informative and beneficial details of where the error occurred and some potential solutions to particular issues.

By having an informative error message developers are able to save time to solve an issue in the implementation. Or in the case of the user encountering an issue, the error messages are able to provide some informative messages such as postfix notations or suggestions to stick within the range of restrictions set for values and symbols.

Magic numbers
Define constants or enumerations to explain the significance of magic numbers. In other words, use named constants instead of magic numbers in the code. By using named constants, this makes for a more structured code structure. At times having a named constant for random values such as value restrictions can be beneficial for developers. After some time the developer might not remember why the restriction was placed there. By having named constants, it provides a more significant detail for the developer as it allows the developer to be able to make more sense with a named constant. For example for records, the value 1000 could have been assigned to a constant named maxRecords.

Another way that I can improve in this area is to notify the users of any limitations upfront to help manage their expectations and improve their experience. By providing the limitations upfront there is a less likely chance that the user will encounter errors while using the application.

Code organisation

In my implementation I felt that I could have improved by breaking down large functions into smaller, more focused functions. By breaking down large functions, this helps for a more readable code structure. This makes it easier to manage and understand the code implementation.
Use a modular design to separate concerns and improve code readability. By adopting a modular design, I will be able to separate concerns as well as enhance code organisation. By separating concerns, I will be able to visualise the vision and structure of the implementation. This allows me to enhance different parts of the code better in order to create a better user experience and better time and space complexity.

Summary

Improvements
In summary, there are a total of 3 areas of improvement for my code implementation. These improvements include but are not limited to, error handling, magic numbers and code organisation. To provide a general overview of what I missed out. Firstly, error handling. I should have used try catch blocks to handle errors, this to provide detailed error messages, and to offer examples and clear instructions. Secondly, magic numbers. I should have used constants and notify the user of any restrictions or limitations upfront to manage user expectation. And lastly, code organisation. Break down large functions, adopt a modular design, and ensure functions have a single responsibility. By understanding and implementing all these improvements, my application could be more user friendly, maintainable and robust.

Overall
This detailed breakdown of data structures and algorithms, provides the step by step process of how I developed the solution for this challenge. This challenge provided me with a good visualisation and application of what I learnt in the classroom. Some of the concepts that I was able to consider were the time and space complexities of past algorithms such as linear and binary search as well as creating my own hashing algorithm to store key to value pairs in a simple symbol table. I was also able to learn how to consider the drawbacks and implications of using different data structures for different situations. For example, for the case of storing postfix expressions, the stack data structure is the best fit for the situation instead of a static array. With all the content provided, it helps to build towards a well rounded and informative documentation of a postfix interpreter program.

3.8) Real World Applications

Compilers and interpreters
In a real world application of a postfix evaluator, compilers and interpreters can be useful applications. They can contribute in two areas, including expression evaluation and intermediate code generation.

Firstly, expression evaluation. In the topic of compilers, postfix notation makes the evaluations of mathematical expressions a lot simpler. Instead of using the regular infix expressions that are used in calculators, we used postfix. It will amenable compilers to have a much faster execution time to parse and execute operations in the correct order. This allows compilers to not have the need to manage operator order and other structures. This is an important process when code is executed to an intermediate form that can be transformed into machine codes.

Secondly, intermediate code generation. Postfix notation provides an efficient intermediate representation during the compilation process. This form allows efficient generation of machine code. This will increase the level of efficiency throughout the program. Be it through compilation or execution time.

Stack-based Virtual Machines
In the topic of stack based virtual machines, some examples of these machines include Java Virtual Machine and .NET Common Language Runtime. Also known as, JVM and .NET CLR. These machines depend on stack based structures a lot. In these environments, postfix notation aligns with the stack based model. This makes it easier to implement and run operations of these machines. This relates to the execution of bytecode. In this execution, operations are stored and taken out of the stack using the push and pop actions that postfix notation works efficiently with.

Summary
Despite the simplicity of this postfix interpreter, it embodies the basic operations of understanding postfix notations and expressions. From splitting the expression into tokens and storing it into a stack based structure, and executing push and pop operations.

In summary, this postfix interpreter is a scaled and adapted version of a more complex application in the real world. Applications that require performance critical tasks in compilers and virtual machines such as JVM and .NET CLR.