Data Structures

Infix to Postfix Converter Application

Steven Duckett

Contents

Summary & Introduction

The goal of this submission is to develop an application that takes a user input that is an infix expression, convert that expression to post fix form, and then output that conversion to the user.

Eg:

Infix Form (Input from user) $\xrightarrow{\text{Convert to}}$ Postfix Form (Output to User)

((P*Q)+(R/(S*(T^U)))) $\xrightarrow[\text{Convert to}]{}$ PQ*RSTU^*/+

This goal will be achieved by using and Abstract Data Type (ADT), an implementation of that ADT and other user-written Java classes. For this assignment there are two options of ADT to be chosen from; Stack or Queue. As well as two implementations to be chosen from; ArrayList or LinkedList.

On the following pages will be a discussion of the positives and negatives of each ADT and implementation, followed by a justification for whichever is chosen.

A class diagram for the application will also be included along with the full source code, and a separate JAR file so the developed application can be run from a terminal. Upon completion the system will be thoroughly tested with 6 varying expressions to test robustness and execution of the application that has been developed.

Abstract Data Type (ADT) Justification

ADT Type	Usages	Positives	Negatives
Stack	• Undo functions of a text editor • Forward/backward navigation buttons of a web browser • Parentheses matching • Evaluating expressions • Converting from infix to postfix form	• Specific usage of this ADT directly correlates to the goal of the application being developed, specifically using Dijkstra's Two-Stack Algorithm • The negative of the first item stored waiting the longest to be accessed is not an issue for the goal of this application	• Uses First In Last Out principle (FILO) so the first item added to the stack has to wait a long time to be accessed; or the entire stack has to be popped in order to access it which is not ideal
Queue	• Asynchronous data transfer • Customers waiting in a hold queue when calling a call centre • Attendees waiting in a queue to access a movie theatre	• Better response time and overall performance as data queues can free up jobs • Flexible and does not require any extra communications programming (Techwalla. 2015) • Ideal to use when the stored items need to be accessed in the order they were stored; First In First Out principle (FIFO)_	• When compared to Stack, this is a more complex ADT to implement

Choice: Stack

Justification

Whilst a Queue could be implemented successfully in the application, it's main use case is not to convert infix expressions to postfix expressions. This coupled with the fact it is a more complicated ADT to code make it seem like a redundant choice; with regards to writing code, best practices always state to keep code as simple as possible so that any developer working on it can pick it up and understand it easily so with this in mind Queue does not seem like the ideal choice.

Although there are positives and negatives to both ADT's as laid out above, the main point which justifies the Stack ADT is that it is specifically designed to convert expressions from infix form to postfix form, which is the goal of the application being developed.

Further to this, the negative discussed in the table above mainly refers to the scenario where users could be trying to download files; if multiple users start a download in a small timeframe, the first users request would be at the bottom of the Stack, and they would have to wait the longest to receive their download despite being the first request. However, in the application being developed, there will only be one user input at one time, so this issue is not a concern.

Dijkstra's Two Stack algorithm has also been selected to help achieve the goal of the application, because it can be specifically used to convert from Infix to Postfix form, as seen below using the example from the assignment brief:

Expression to be converted: ((P*Q)+(R/(S*(T^U))))

Step No.	Method	Stack 1 (S1)	Stack 2 (S2)
1	Meet left operand, ignore	Empty	Empty
2	Meet left operand, ignore	Empty	Empty
3	Meet P, push to S1	P,	Empty
4	Meet *, push to S2	P,	*,
5	Meet Q, push to S1	P, Q,	*,
6	Meet right operand, pop top from S2 and top 2 from S1 and concatenate	Empty	Empty
7	Push result of step 6 to S1	PQ*,	Empty
8	Meet +, push to S2	PQ*,	+,
9	Meet left operand, ignore	PQ*,	+,
10	Meet R, push to S1	PQ*, R,	+,
11	Meet /, push to S2	PQ*, R,	+, /,
12	Meet left operand, ignore	PQ*, R,	+, /,
13	Meet S, push to S1	PQ*, R, S,	+, /,
14	Meet *, push to S1	PQ*, R, S,	+, /, *,
15	Meet left operand, ignore	PQ*, R, S,	+, /, *,
16	Meet T, push to S1	PQ*, R, S, T,	+, /, *,
17	Meet ^, push to S2	PQ*, R, S, T,	+, /, *, ^,
18	Meet U, push to S1	PQ*, R, S, T, U,	+, /, *, ^,
19	Meet right operand, pop top from S2 and top 2 from S1 and concatenate	PQ*, R, S,	+, /, *,
20	Push result from step 19 to S1	PQ*, R, S, TU^,	+, /, *,
21	Meet right operand, pop top from S2 and top 2 from S1 and concatenate	PQ*, R,	+, /,
22	Push result from step 21 to S1	PQ*, R, STU^*	+, /,
23	Meet right operand, pop top from S2 and top 2 from S1 and concatenate	PQ*,	+,
24	Push result of step 23 to S1	PQ*, RSTU^*/	+,
25	Meet right operand, pop top from S2 and top 2 from S1 and concatenate	Empty	Empty
26	Push result of step 25 to S1	PQ*RSTU^*/	Empty
27	End of expression, pop top from S1 and display as outcome		

Outcome Expression: PQ*RSTU^*/+

Justification for ArrayList and LinkedList is on the following page

Implementation Type	Positives	Negatives
ArrayList	• Ideal to use when a fixed size is required • Ideal to use when searching is of the highest priority • Simple to implement and very efficient • Easy to retrieve elements/items from the ArrayList as each position has a specific index	• Has a fixed capacity – meaning the application will throw and IllegalStateException when the maximum capacity is reached, and an item is pushed onto the stack • When capacity is reached, the array list size must be doubled and copied regardless of how many new items need to be pushed onto the stack. So, if the ArrayList size is 500, but only 1 extra object needs to be stored there will be another 499 spaces that take up memory for no reason • Not an ideal choice when inserting or deleting is of highest priority • When implanted with ADT Queue, there is a need to remember that the size of the array must always be greater than the expression length; this could lead to bugs within the code
LinkedList	• Memory usage is proportional to the number of elements in the LinkedList • No capacity limit as the elements are added/stored dynamically • Ideal to use when inserting or deleting is of highest priority	• Not an ideal choice when searching is of highest priority as the entire LinkedList must be traversed

Choice: ArrayList

Justification:

The brief for the application to be developed specifically states that the length of the expression must contain at maximum of 20 characters, therefore because an ArrayList outperforms a LinkedList when the exact number of elements to be stored is known (stated above in the comparison table), there will not be any issue regarding the memory allocation as the developer knows that the size of the ArrayList must always be limited to 20 elements/items. Furthermore, because Dijkstra's two stack algorithm will be utilised to convert the expression, there will be two separate ArrayList's in use meaning that the number of elements stored will always be less than the length of the expression that is input by the user.

Another negative stated in the comparison table was that the ArrayList size has to be larger than the length of the expression, as this refers to the Queue ADT this is not an issue as the chosen ADT chosen is Stack and it does not apply.

A concern when choosing the ArrayList implementation of the Stack ADT is regarding inserting or deleting. This is because to work out where the element should be inserted/deleted, the index of that element must be checked against the index of each element/item already in the ArrayList using a for-loop. Regardless of this negative however, because the maximum size of the ArrayList is specified in the assignment brief to be 20, this is not a concern because performance will not be noticeably affected. Although if an application was being developed where the ArrayList size was required to be much larger, this would require further consideration.

Finalisation

ADT chosen: Stack

Implementation chosen: ArrayList

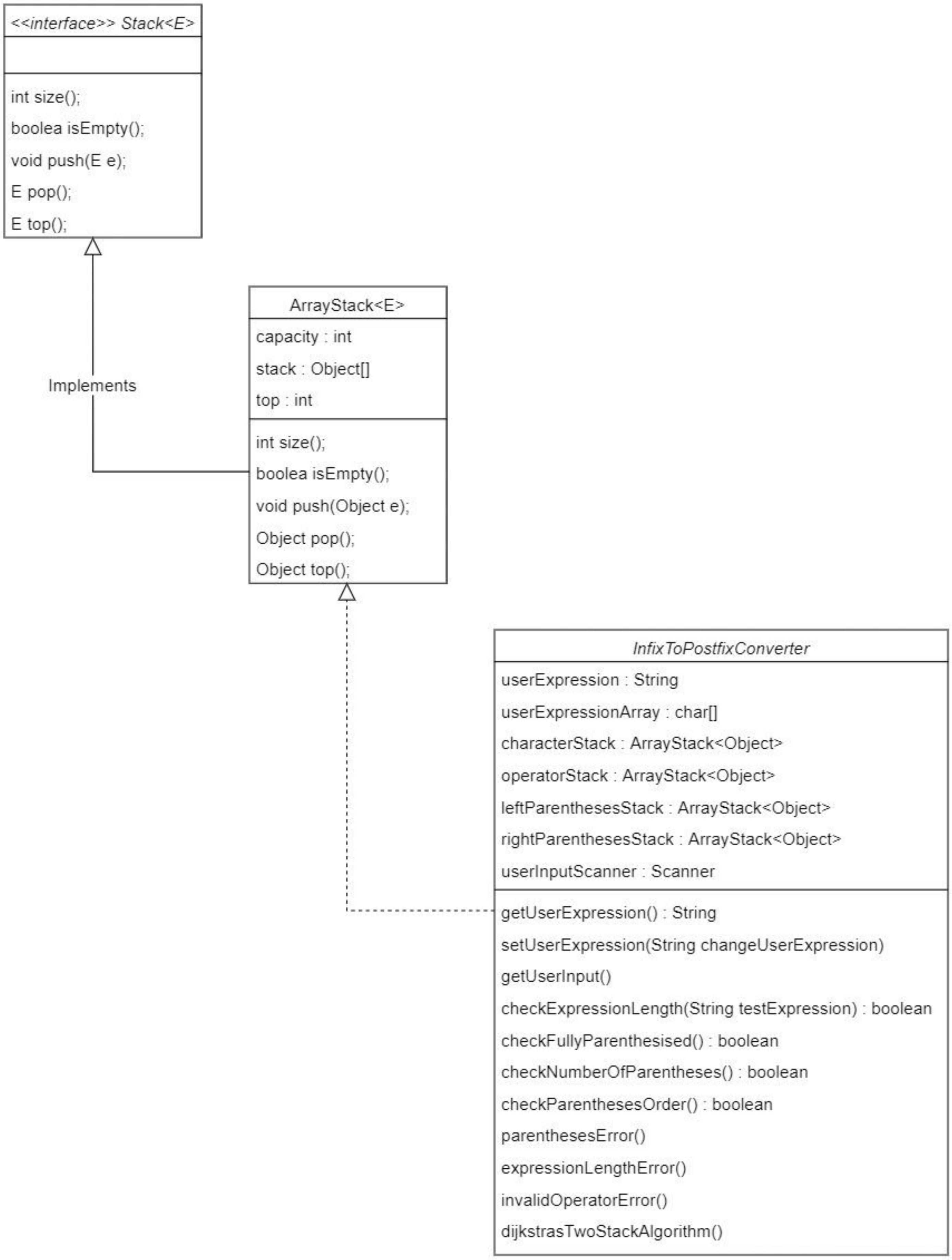
Acknowledgements

Inspiration for the source code for the Abstract Data Type: Stack class was taken from (Goodrich, M. T., & Tamassia, R. 2015), however the creator edited the method comments to language that they would understand and allow them to gain a deeper understanding of the methods before implanting them in their own application.

Inspiration for the source code for the ArrayStack class was taken from (Gaber, T. 2021), again the creator of this application edited the method comments and parameter values to ones that were more appropriate to them, allowing them to gain a deeper understanding and appreciation before utilising them in their own application.

During the justification stage, further research was conducted into the positives and negatives of each ADT (Stack and Queue) using the Data structures and algorithms in Java textbook written by Goodrich, M. T., & Tamassia, R.

Class Diagram



<<interface>> Stack<E>
int size();
boolea isEmpty();
void push(E e);
E pop();
E top();
Implements
ArrayStack<E>
capacity : int
stack : Object[]
top : int
int size();
boolea isEmpty();
void push(Object e);
Object pop();
Object top();
InfixToPostfixConverter
userExpression : String
userExpressionArray : char[]
characterStack : ArrayStack<Object>
operatorStack : ArrayStack<Object>
leftParenthesesStack : ArrayStack<Object>
rightParenthesesStack : ArrayStack<Object>
userInputScanner : Scanner
getUserExpression() : String
setUserExpression(String changeUserExpression)
getUserInput()
checkExpressionLength(String testExpression) : boolean
checkFullyParenthesised() : boolean
checkNumberOfParentheses() : boolean
checkParenthesesOrder() : boolean
parenthesesError()
expressionLengthError()
invalidOperatorError()
dijkstrasTwoStackAlgorithm()

Testing the Application

The requirements for the application were stated as follows in the assignment brief:

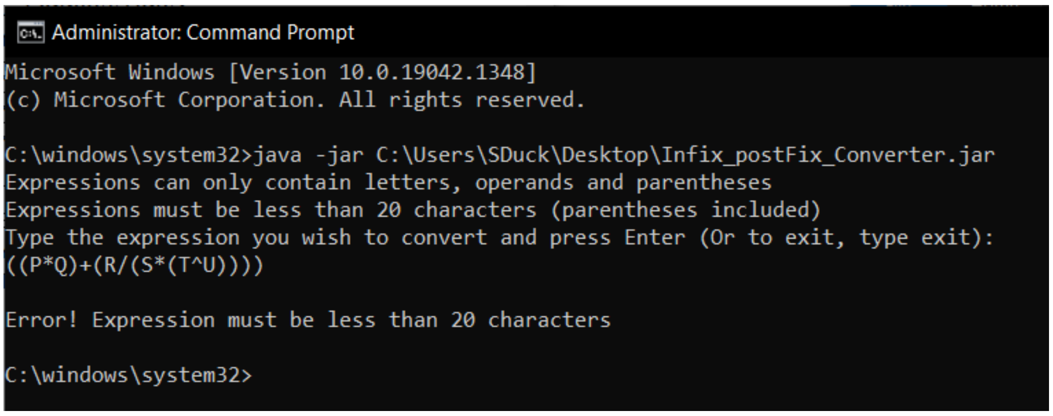
- The expression must be a legal infix expression that contains at most 20 characters
- It has only three operators: addition(+), multiplication(*) and division (/)
- It contains delimiter parentheses "(" and ")"
- It has variables taken from the set {a, b, …, z}
- It contains no blank spaces

The application has been designed in such a way that it runs in the following manner

1. It prompts the user for an input and informs them it must be in a specific format. If the user enters "exit" the program stops running, if not then assign their input to a variable (with all spaces removed via the .replace() method for Strings in Java) and proceed to step 2.
2. Check that the expression is no longer than 20 characters, including parentheses. If it is, stop the program and inform the user, if not proceed to step 3.
3. Now check that the user input is fully parenthesised, because Dijkstra's Two Stack Algorithm can only execute correctly with a fully parenthesised expression, by doing the following:
    a. To be fully parenthesised, each operator in the expression must have a pair of parentheses, so the number of parentheses entered must be double the number of operators entered. If not, stop the program and inform the user.
    b. If 3a is successful, now check that each pair of parentheses is opened and closed by its corresponding type, so a "(" is followed by a ")", if not stop the program and inform the user. If it is, proceed to step 4.
4. At this stage the expression has been confirmed to be of correct length and is fully parenthesised, so now it checks if the expression contains a "-" or a "^" operator, if so these are not allowed as per the requirements so stop the program and inform the user. If neither is present continue to step 5.
5. At this stage the expression has been confirmed to be of correct length, is fully parenthesised and only contains allowed operators so run Dijkstra's Two Stack Algorithm to convert the infix expression to postfix form.

Testing Screenshots

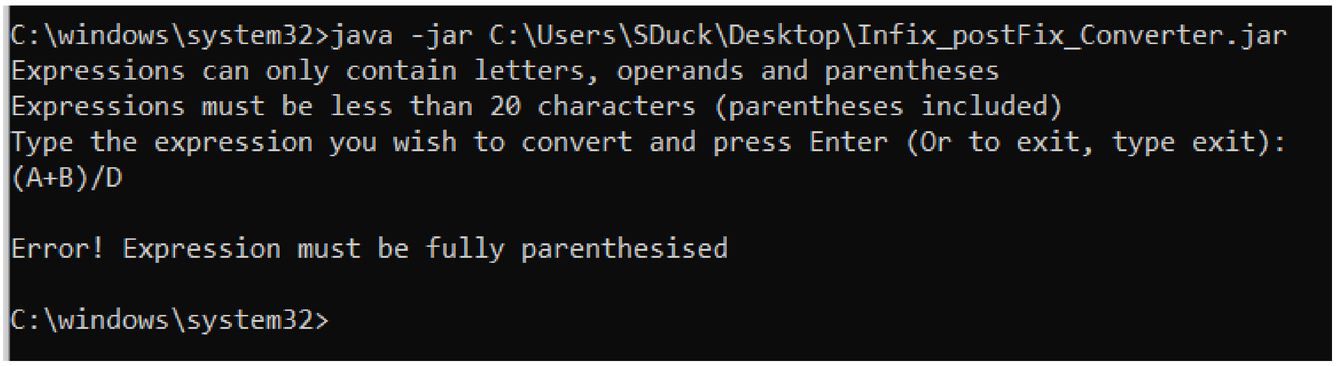
All testing for the application/program has been conducted using the .JAR file running in the Windows command prompt. Each of the following Figures will be screenshots that relate to testing a specific step in the process described above. There will also be 6 figures that relate to step 5 of the process to robustly test that the program can convert from infix to postfix using Dijkstra's Two Stack Algorithm.



```
Microsoft Windows [Version 10.0.19042.1348]
(c) Microsoft Corporation. All rights reserved.

C:\windows\system32>java -jar C:\Users\SDuck\Desktop\Infix_postFix_Converter.jar
Expressions can only contain letters, operands and parentheses
Expressions must be less than 20 characters (parentheses included)
Type the expression you wish to convert and press Enter (Or to exit, type exit):
((P*Q)+(R/(S*(T^U))))

Error! Expression must be less than 20 characters

C:\windows\system32>
```

Figure 1 - Testing Step 2 of the process using the expression given in the assignment brief

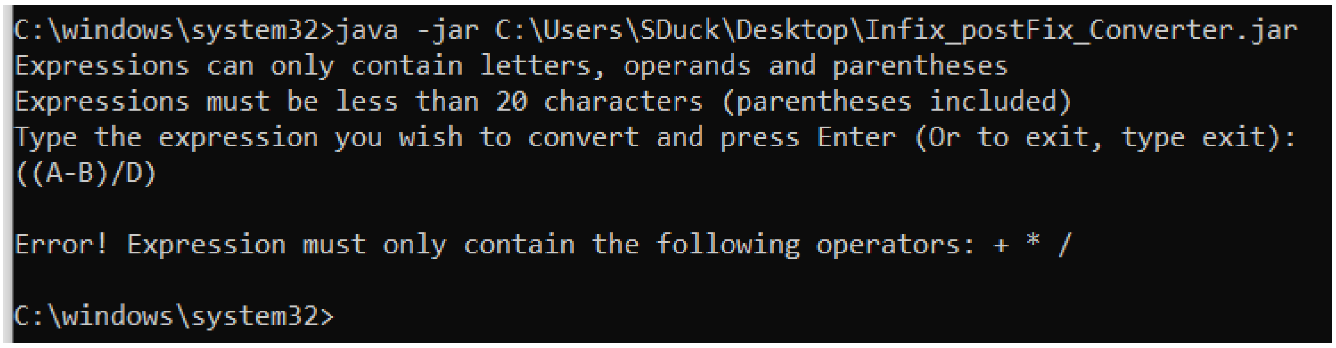
As can be seen in Figure 1, the expression given in the assignment brief cannot be used for testing as when it is made to be fully parenthesised it is more than 20 characters long which violates one of the requirements in the assignment brief.



```
C:\windows\system32>java -jar C:\Users\SDuck\Desktop\Infix_postFix_Converter.jar
Expressions can only contain letters, operands and parentheses
Expressions must be less than 20 characters (parentheses included)
Type the expression you wish to convert and press Enter (Or to exit, type exit):
(A+B)/D

Error! Expression must be fully parenthesised

C:\windows\system32>
```

Figure 2 - Testing Step 3 of the process and showing the application informs the user of an error when input is not fully parenthesised

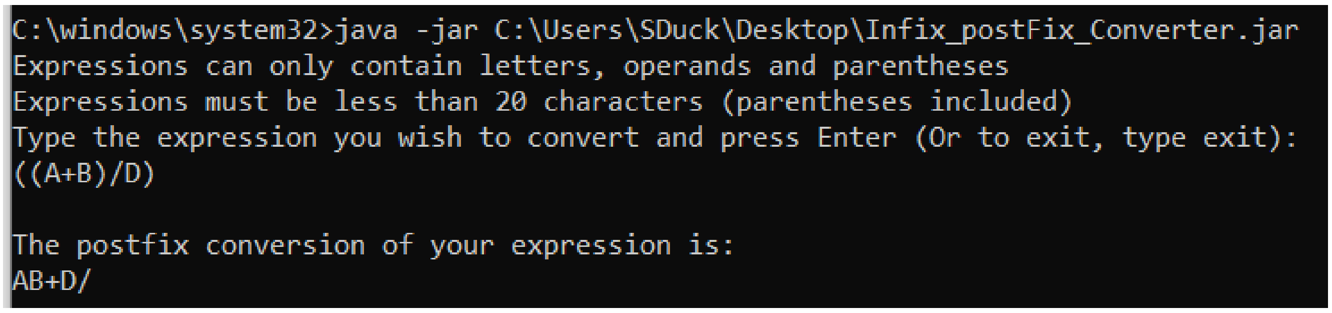
Figure 2 shows that if an expression entered by the user is not fully parenthesised then the program informs them and stops running. The program has been designed this way as Dijkstra's Two Stack Algorithm only accepts fully parenthesised expressions so without full parentheses it will not convert from infix to postfix form.



```
C:\windows\system32>java -jar C:\Users\SDuck\Desktop\Infix_postFix_Converter.jar
Expressions can only contain letters, operands and parentheses
Expressions must be less than 20 characters (parentheses included)
Type the expression you wish to convert and press Enter (Or to exit, type exit):
((A-B)/D)

Error! Expression must only contain the following operators: + * /

C:\windows\system32>
```

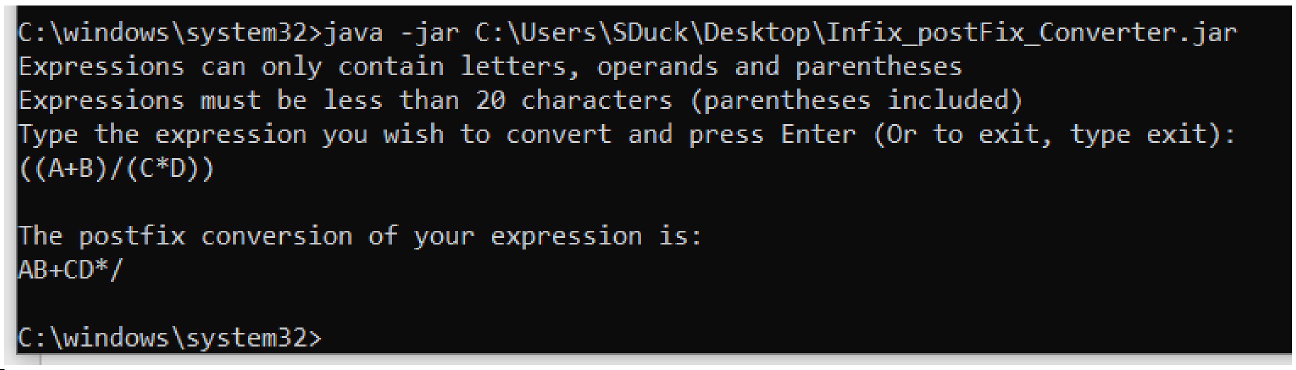
Figure 3 - Testing Step 4 of the process and throwing an error if an illegal operator is given in the expression

Figure 4 below shows that the previous steps in the process (1, 2, 3a, and 3b) were all successful, however even though the user entered a fully parenthesised expression of correct length, because it contains an operator that is not allowed (per the requirements) the program stops running and informs the user of the error.



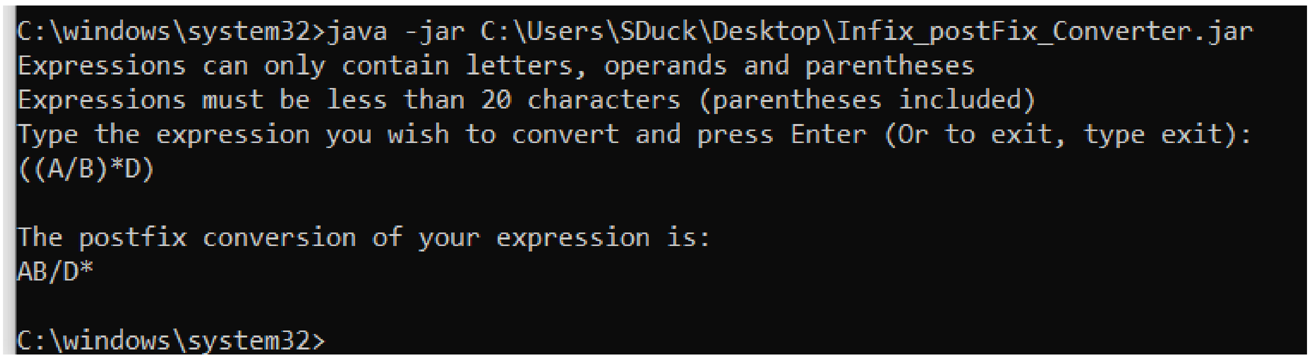
```
C:\windows\system32>java -jar C:\Users\SDuck\Desktop\Infix_postFix_Converter.jar
Expressions can only contain letters, operands and parentheses
Expressions must be less than 20 characters (parentheses included)
Type the expression you wish to convert and press Enter (Or to exit, type exit):
((A+B)/D)

The postfix conversion of your expression is:
AB+D/
```

Figure 4 - A successful conversion from infix to postfix form



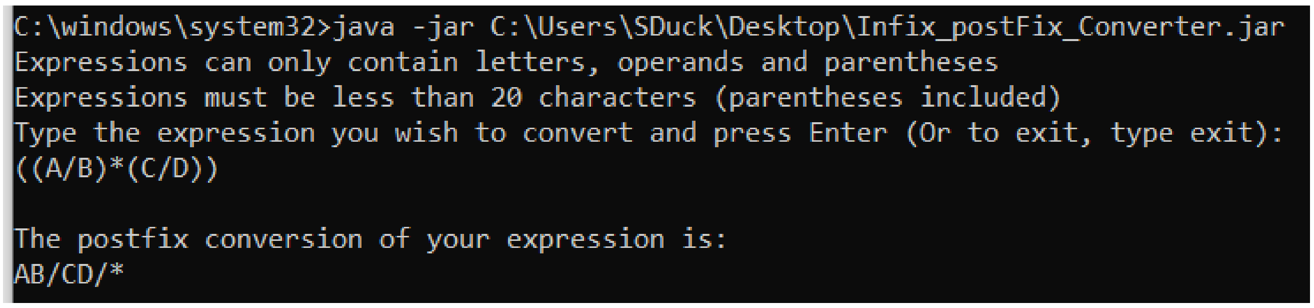
```
C:\windows\system32>java -jar C:\Users\SDuck\Desktop\Infix_postFix_Converter.jar
Expressions can only contain letters, operands and parentheses
Expressions must be less than 20 characters (parentheses included)
Type the expression you wish to convert and press Enter (Or to exit, type exit):
((A+B)/(C*D))

The postfix conversion of your expression is:
AB+CD*/

C:\windows\system32>
```

Figure 5 - A successful conversion from infix to postfix form



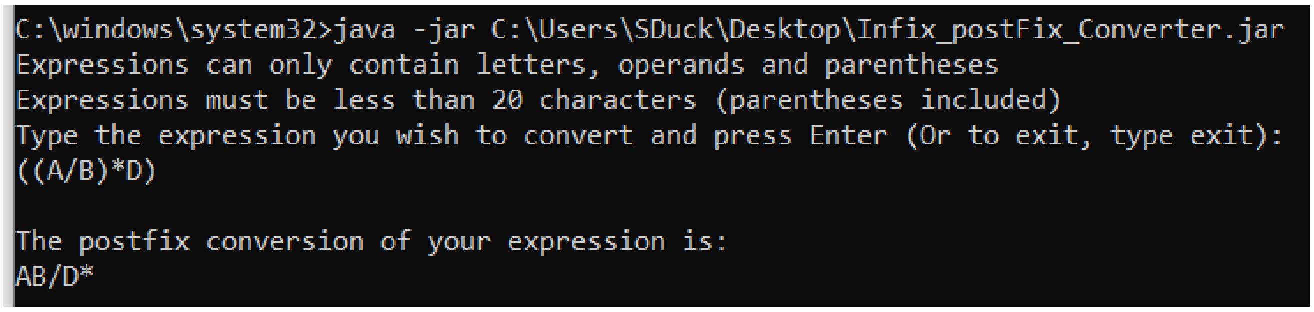
```
C:\windows\system32>java -jar C:\Users\SDuck\Desktop\Infix_postFix_Converter.jar
Expressions can only contain letters, operands and parentheses
Expressions must be less than 20 characters (parentheses included)
Type the expression you wish to convert and press Enter (Or to exit, type exit):
((A/B)*D)

The postfix conversion of your expression is:
AB/D*

C:\windows\system32>
```

Figure 6 - A successful conversion from infix to postfix form



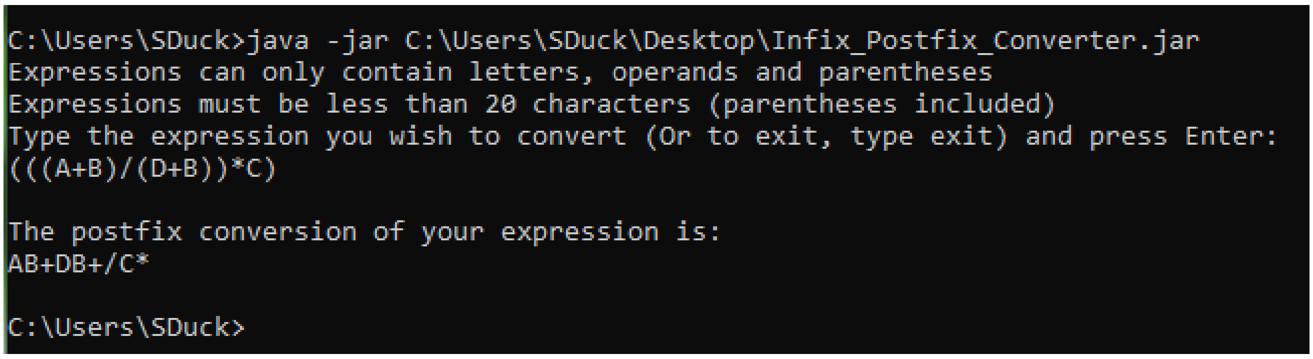
```
C:\windows\system32>java -jar C:\Users\SDuck\Desktop\Infix_postFix_Converter.jar
Expressions can only contain letters, operands and parentheses
Expressions must be less than 20 characters (parentheses included)
Type the expression you wish to convert and press Enter (Or to exit, type exit):
((A/B)*(C/D))

The postfix conversion of your expression is:
AB/CD/*
```

Figure 7 - A successful conversion from infix to postfix form



```
C:\windows\system32>java -jar C:\Users\SDuck\Desktop\Infix_postFix_Converter.jar
Expressions can only contain letters, operands and parentheses
Expressions must be less than 20 characters (parentheses included)
Type the expression you wish to convert and press Enter (Or to exit, type exit):
((A/B)*D)

The postfix conversion of your expression is:
AB/D*
```

Figure 8 - A successful conversion from infix to postfix form



```
C:\Users\SDuck>java -jar C:\Users\SDuck\Desktop\Infix_Postfix_Converter.jar
Expressions can only contain letters, operands and parentheses
Expressions must be less than 20 characters (parentheses included)
Type the expression you wish to convert (Or to exit, type exit) and press Enter:
(((A+B)/(D+B))*C)

The postfix conversion of your expression is:
AB+DB+/C*

C:\Users\SDuck>
```

Figure 9 - A successful conversion from infix to postfix form

Figures 4, 5, 6, 7, 8 and 9 all show successful conversions from infix to postfix form utilising Dijkstra’s Two Stack Algorithm. They are successful because when the process reaches step 5 (where Dijkstra’s Two Stack Algorithm is implemented) it has already checked and validated that the expression inputted by the user passes all the necessary requirements specified in the brief.

References

Gaber, T. (2021). *Data Structure & Algorithm Section 5 Abstract Data Type: Stack*. Lecture, Virtual Lecture.

Goodrich, M. T., & Tamassia, R. (2015). *Data structures and algorithms in Java.* John Wiley.

The Advantages of a Queue in Data Structure | Techwalla. (2015). Techwalla. https://www.techwalla.com/articles/the-advantages-of-a-queue-in-data-structure