15 Data Structure: Stack and Queue

Programming in C

Teacher: Po-Wen Chi
neokent@gapps.ntnu.edu.tw

January 18, 2019

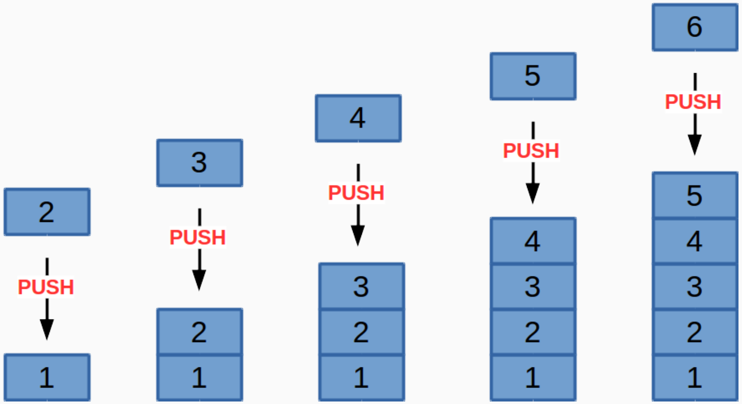
Department of Computer Science and Information Engineering,
National Taiwan Normal University

Stack

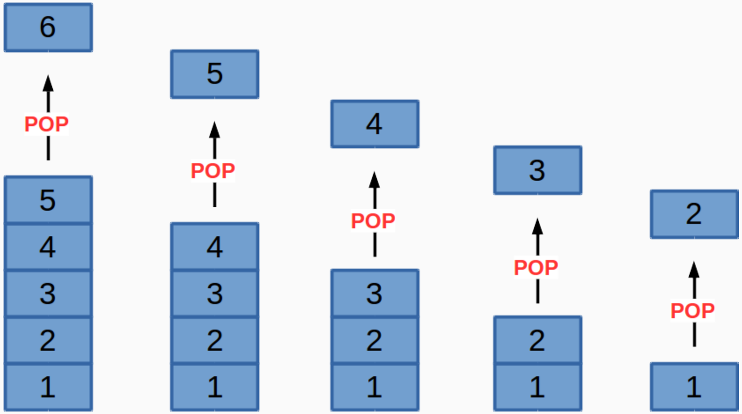
Stack

- In computer science, a stack is a data structure (**abstract data type**) that serves as a collection of elements, with **two principal operations**:
    1. **push**: adds an element to the collection.
    2. **pop**: removes the most recently added element that was not yet removed.

Push



Pop



LIFO: Last in, First out.

Where is Stack in My Computer??

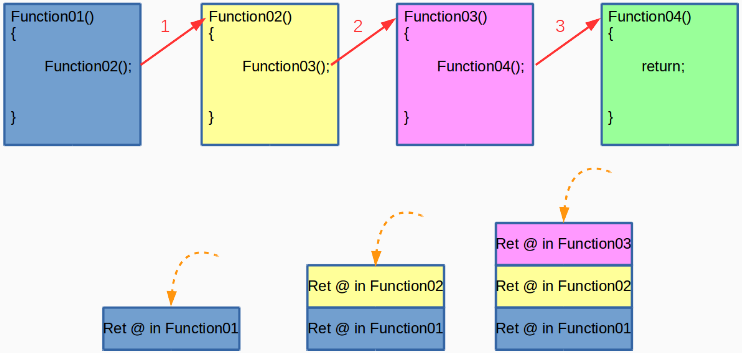
- Compiler.
    - Parentheses matching.
- Function Call.
    - Stacks entered the computer science literature in 1946, when Alan M. Turing used the terms bury and unbury as a means of calling and returning from subroutines.

Parentheses Matching

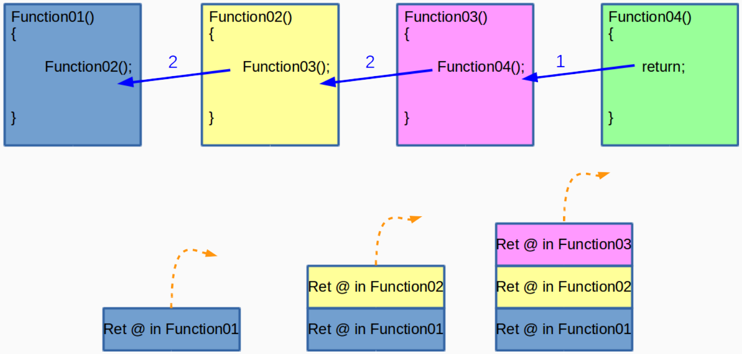
Are the following lines valid or invalid?

- ()()()()
- (((())))
- (()))()

Function Call



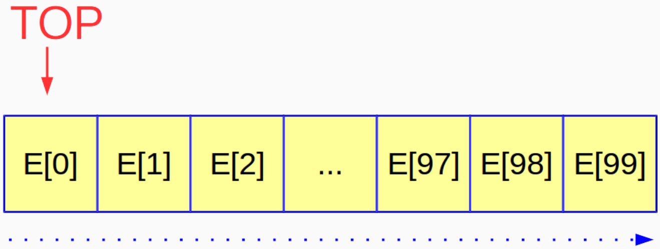
Function Call



Array-based Implementation

```c
#define MAX_SIZE    (100)

typedef struct _sStack
{
    size_t  top;
    int32_t elements[MAX_SIZE];
} sStack;
```



Push and Pop

```
int32_t push_stack( sStack *pS, int32_t val )
{
    pS -> elements[ pS -> top ] = val;
    pS -> top += 1;
    return 0;
}

int32_t pop_stack( sStack *pS, int32_t *pVal )
{
    pS -> top -= 1;
    *pVal = pS -> elements[ pS -> top ];
    return 0;
}
```

Implementation

Please see stack.array.

Demo

Parentheses Balance

A string of this type is said to be **Parentheses Balanced** if

1. if it is the empty string.
2. if A and B are correct, AB is correct.
3. if A is correct, (A), [A], $\{A\}$ is correct.

Write a function to decide if a given string is Parentheses Balanced.

How to do this?

Idea

- Undoubtedly, left one and right one should be **symmetric**.
- When we see a right parenthesis, the last parenthesis must be left and we can cancel this parenthesis pair.
- If we can remove all parentheses, the input must be Parentheses Balanced.

Examples

- Balanced: $((()())())$

Examples

- Balanced: ((()())())
  1. ((()())())

Examples

- Balanced: ((()())())
  1. ((()())())
  2. ((())())

Examples

- Balanced: $((()())())$
  1. $((()())())$
  2. $(())())$
  3. $(()())$

Examples

- Balanced: ((()())())
    1. ((()())())
    2. ((())())
    3. (()())
    4. (())

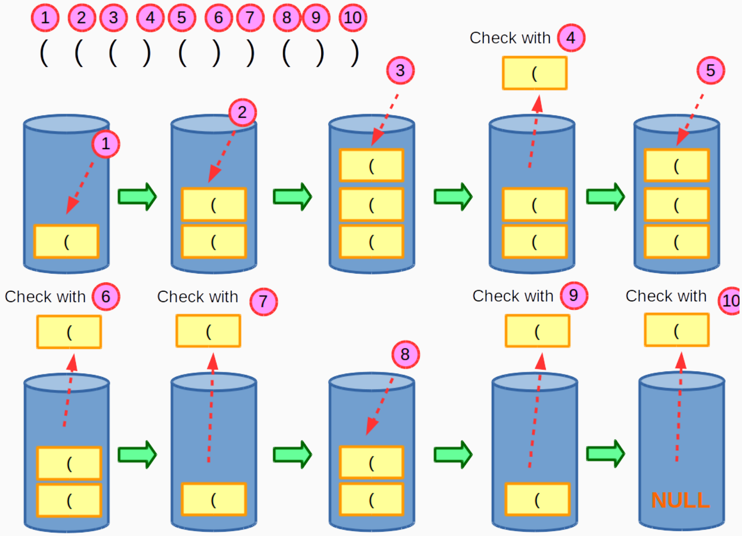
Examples

- Balanced: ((()())())
  1. ((()())())
  2. ((())())
  3. (()())
  4. (())
  5. ()

Examples

- Balanced: ((()())())
  1. ((()())())
  2. ((())())
  3. (()())
  4. (())
  5. ()
  6. NULL

Solved with a Stack



Implementation

Please see balance.c.

Practice: Latex Begin End Balance

You may know that all my slides are made from latex. You do not know latex syntax and that is fine. I will teach you a little bit.

There are lots of blocks in a latex file. One block starts with a keyword begin and ends with a keyword end. The example is as follows:

```
\begin{displaymath}
\end{displaymath}
```

There may be lots of keywords, like displaymath in the example. Please write a program to check if a given latex file is balanced with begin and end.

Array-based Stack Implementation

- Pros:
    - Easy to understand.
    - Simple implementation.
- Cons:
    - Size limitation.

Array-based Stack Implementation

- Pros:
  - Easy to understand.
  - Simple implementation.
- Cons:
  - Size limitation.

Can we use **linked list** to implement Stack?

Array-based Stack Implementation

- Pros:
    - Easy to understand.
    - Simple implementation.
- Cons:
    - Size limitation.

Can we use **linked list** to implement Stack?

Sure! Actually, Linus's implementation can be treated as a stack.

Practice

Please implement a stack with list.h used in the previous section.

Queue

Queue

- What is a **Queue**:
  - A queue is an ordered list where all insertions take place at the rear while deletions occur at the front.
  - Also known as **First-In-First-Out (FIFO)**.
- A real world example:

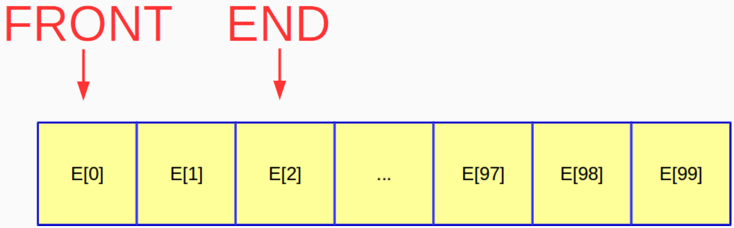
Operations

- Two Main Operations:
  1. **enqueue**: insert an element at the end.
  2. **dequeue**: remove the element at the front.
- Other Useful Operations:
  - **size**
  - **isEmpty**

Array-based Implementation

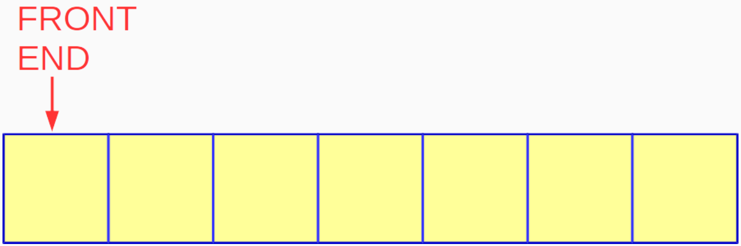
```
#define MAX_SIZE    (100)

typedef struct _sQueue
{
    size_t  front;
    size_t  end;
    int32_t elements[MAX_SIZE];
} sQueue;
```

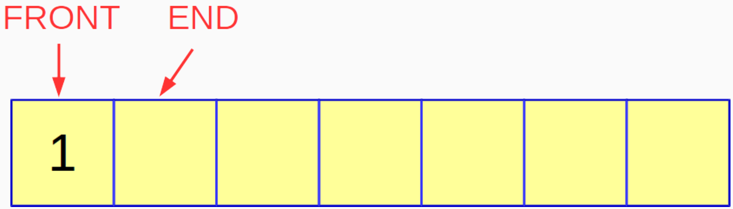


Enqueue and Dequeue

- Initialize



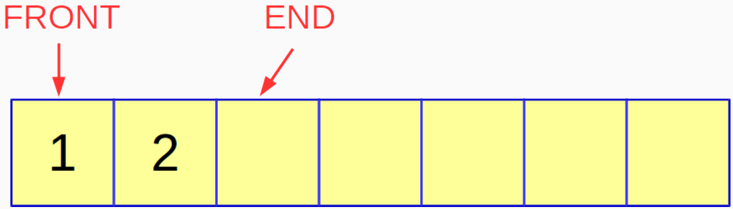
- Enqueue 1



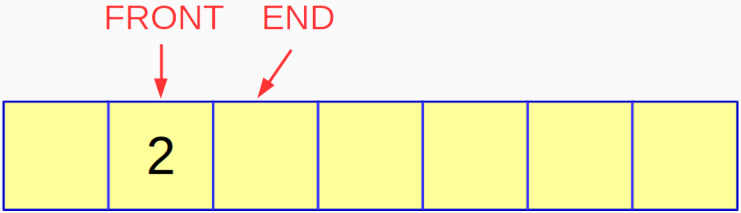
- Enqueue 2

Enqueue and Dequeue

- Enqueue 2



- Dequeue



Array-based Implementation

```
int32_t enqueue( sQueue *pQ, int32_t val )
{
    pQ -> elements[ pQ -> end ] = val;
    pQ -> end++;
    return 0;
}

int32_t dequeue( sQueue *pQ, int32_t *pVal )
{
    *pVal = pQ -> elements[ pQ -> front ];
    pQ -> front++;
    return 0;
}
```

Array-based Implementation

```
int32_t enqueue( sQueue *pQ, int32_t val )
{
    pQ -> elements[ pQ -> end ] = val;
    pQ -> end++;
    return 0;
}

int32_t dequeue( sQueue *pQ, int32_t *pVal )
{
    *pVal = pQ -> elements[ pQ -> front ];
    pQ -> front++;
    return 0;
}
```

What is the problem?

Array-based Implementation

```
int32_t enqueue( sQueue *pQ, int32_t val )
{
    pQ -> elements[ pQ -> end ] = val;
    pQ -> end = ( pQ -> end + 1 ) % MAX_SIZE;
    return 0;
}

int32_t dequeue( sQueue *pQ, int32_t *pVal )
{
    *pVal = pQ -> elements[ pQ -> front ];
    pQ -> front = ( pQ -> front + 1 ) % MAX_SIZE;
    return 0;
}
```

Implementation

Please see queue.array.

Practice

Please modify the code so that when a new element is enqueued, the oldest element will be removed.

Practice

Please implement a stack with list.h used in the previous section.

Classic Problems

We have learned some data structures.

Now we will use them to solve some classic problems.

Classic Problems

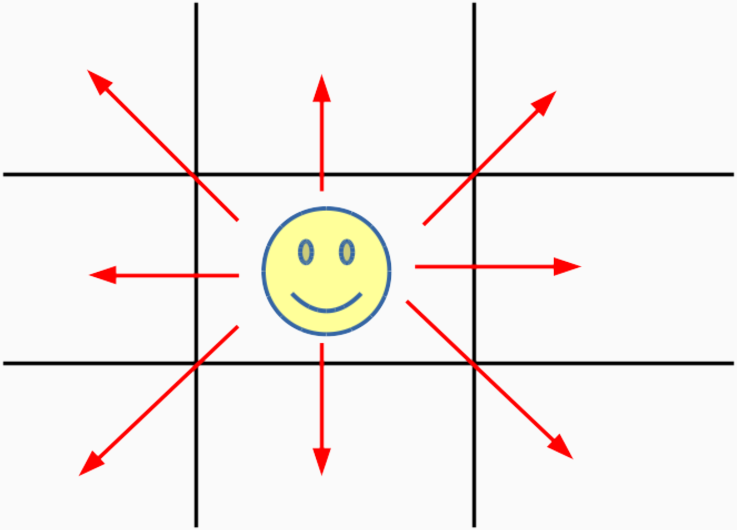
- Maze Problems.
- Postfix from Infix.
- Service Simulation.

Please Find a Path

0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

0: you can walk; 1: blocked.

Movement



Of Course, We can Do it By Hand

0	1	0	0	0	1	1	0	0	0	1	1	1	1	1
1	0	0	0	1	1	0	1	1	1	0	0	1	1	1
0	1	1	0	0	0	0	1	1	1	1	0	0	1	1
1	1	0	1	1	1	1	0	1	1	0	1	1	0	0
1	1	0	1	0	0	1	0	1	1	1	1	1	1	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	0	1	1	0	1	1	1	0	1	0	0	1	0	1
0	1	1	1	1	0	0	1	1	1	1	1	1	1	1
0	0	1	1	0	1	1	0	1	1	1	1	1	0	1
1	1	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	1	1	1	1	1	0	0	0	1	1	1	1	0
0	1	0	0	1	1	1	1	1	0	1	1	1	1	0

0: you can walk; 1: blocked.

Idea

- Push all possible movements into the stack.
- Pop the first element and then push all possible movements into the stack.
- Termination condition:
    - **Reach endpoint**.
    - **Empty stack**: No possible movements.

Demo

Please see maze.

Practice

Please modify the code for the following two features:

1. Print the path.
2. Find the shortest path.

Practice

Please modify the code for the following two features:

1. Print the path. maze2
2. Find the shortest path.

Potential Problem

There is a serious potential problem in maze and maze2. Can you find it?

Potential Problem

There is a serious potential problem in maze and maze2. Can you find it?

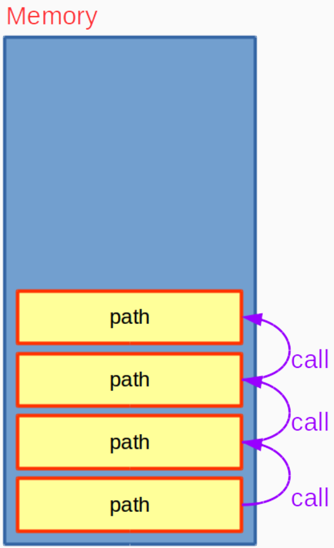
Memory leak.

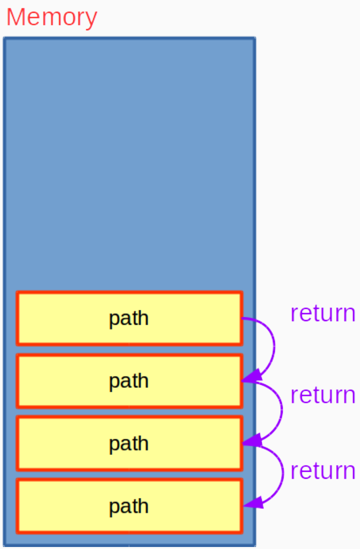
I Do Not Like Stack

I can do this with **recursive way**.

Please see maze3.

Actually, This is still a Stack





Infix, Postfix and Prefix

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions.

- **Infix**: $a + b$
  - Operators are written in-between their operands.
- **Postfix**: $ab+$
  - Operators are written after their operands.
- **Prefix**: $+ab$
  - Operators are written before their operands.

Infix, Postfix and Prefix

Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions.

- **Infix**: $a + b$
  - Operators are written in-between their operands.
- **Postfix**: $ab+$
  - Operators are written after their operands.
- **Prefix**: $+ab$
  - Operators are written before their operands.

Are you mad? Why do we need this??

Because of Precedence

- Infix notation needs extra information to make the order of evaluation of the operators clear.
  - Operator precedence.
  - Brackets () to allow users to override these rules.
- Other two notations have no precedence issue and no brackets are allowed.
  - Always **left-to-right**.
  - So it will become very easy to calculate the result.

Example

- Infix:

$$A/B - C + D \times E - A \times C$$

- Postfix:

$$AB/C - DE \times +AC \times -$$

Infix to Postfix

1. Fully parenthesize the expression.
2. Move all operators so that they replace their corresponding right parentheses.
3. Delete all parentheses.

Example

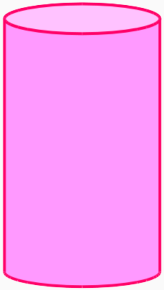
1. Input: $A/B - C + D \times E - A \times C$.
2. $((((A/B) - C) + (D \times E)) - (A \times C))$.
3. $((((A/B) - C) + (D \times E)) - (A \times C))$.
4. $AB/C - DE \times + AC \times -$

How to Implement?

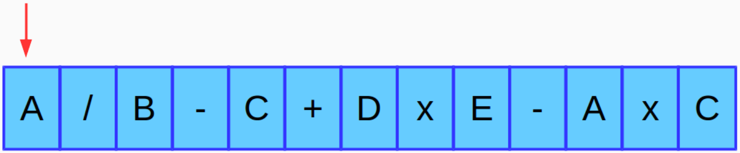
- Actually, parenthesis implies precedence.
- That is,
  - If the precedence of the input. operator is greater than the precedence of the operator in the stack, push it.
  - Else, Pop all the operators from the stack which are greater than or equal to in precedence than that of the scanned operator. After doing that Push the input operator to the stack.

Example: $A/B - C + D \times E - A \times C$

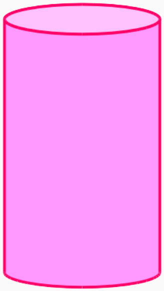
A	/	B	-	C	+	D	x	E	-	A	x	C



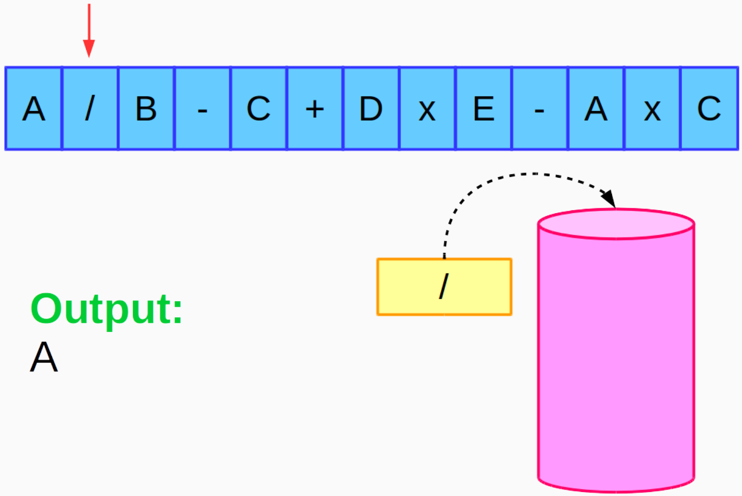
Example: $A/B - C + D \times E - A \times C$



Output:
A



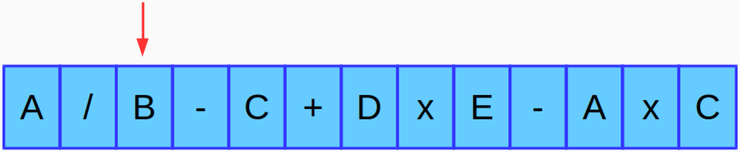
Example: $A/B - C + D \times E - A \times C$



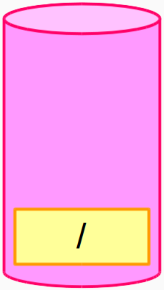
Output:

A

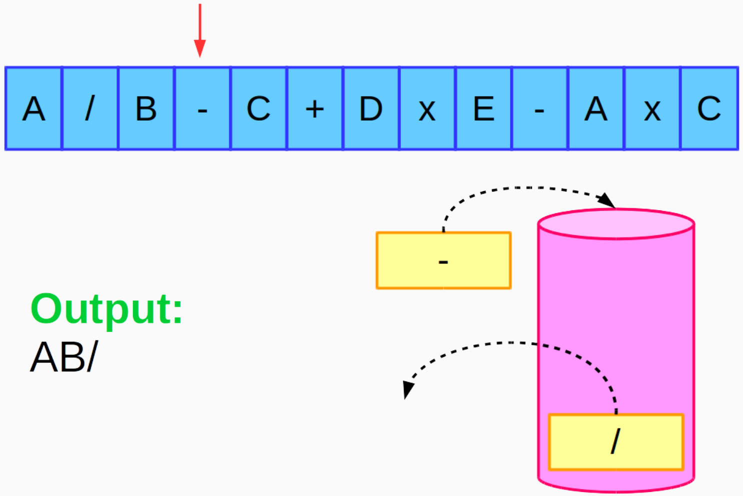
Example: $A/B - C + D \times E - A \times C$



Output:
AB

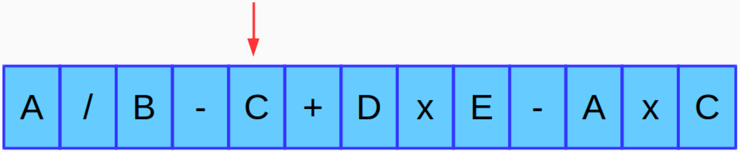


Example: $A/B - C + D \times E - A \times C$

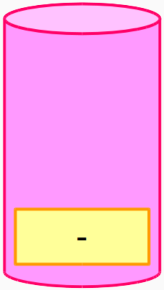


Output:
AB/

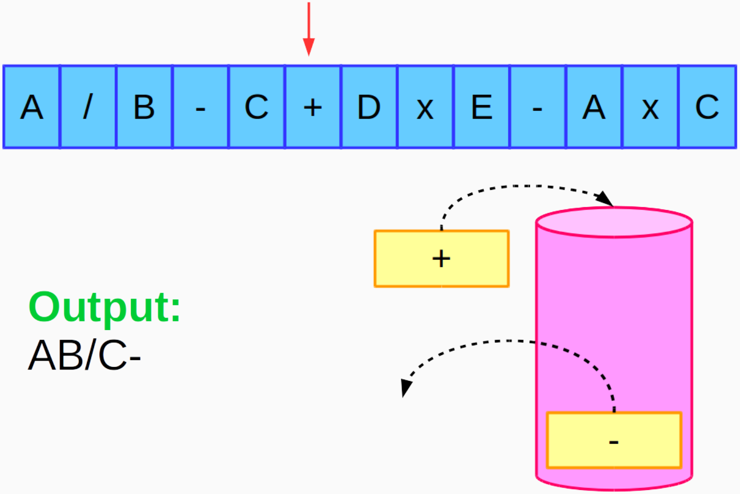
Example: $A/B - C + D \times E - A \times C$



Output:
AB/C

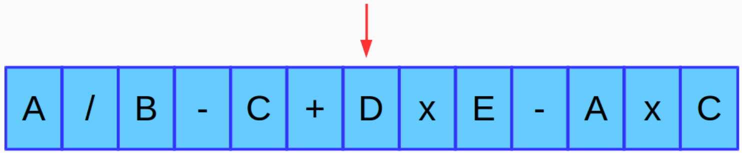


Example: $A/B - C + D \times E - A \times C$

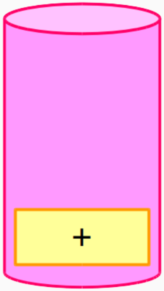


Output:
AB/C-

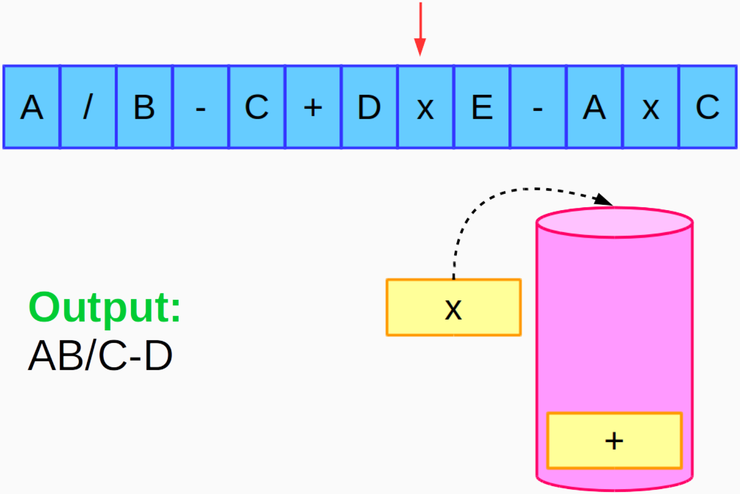
Example: $A/B - C + D \times E - A \times C$



Output:
AB/C-D

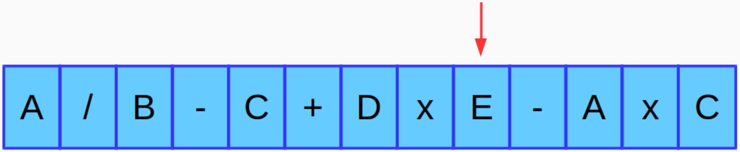


Example: $A/B - C + D \times E - A \times C$

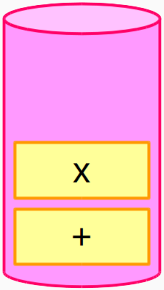


Output:
AB/C-D

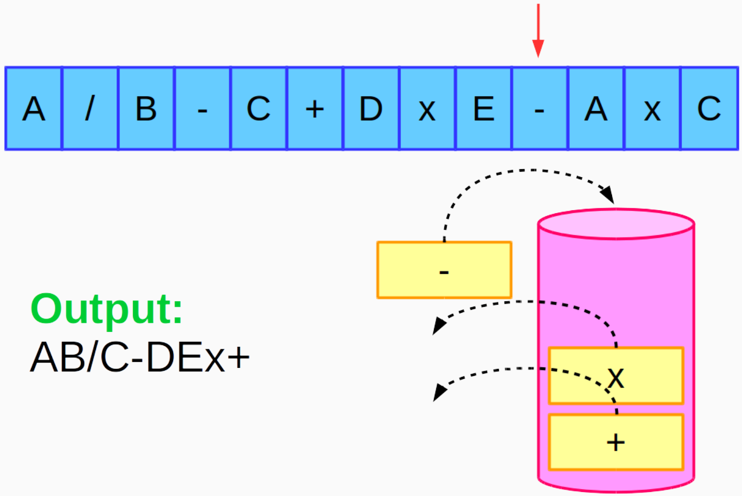
Example: $A/B - C + D \times E - A \times C$



Output:
AB/C-DE

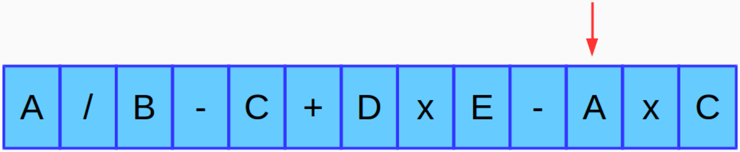


Example: $A/B - C + D \times E - A \times C$

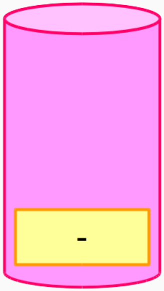


Output:
AB/C-DEx+

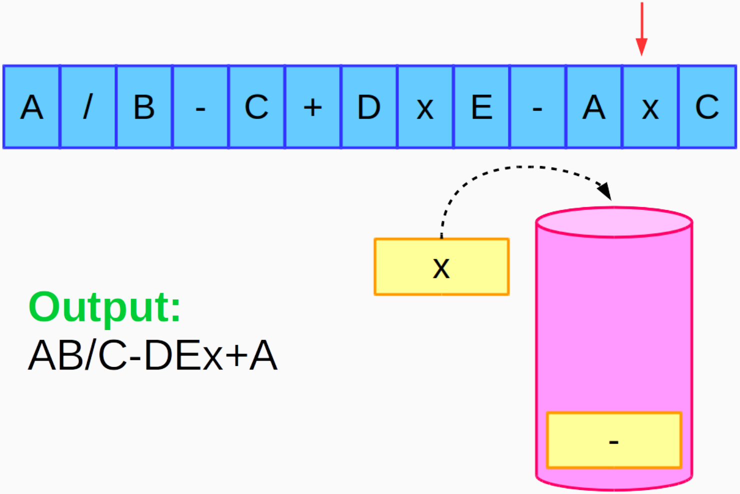
Example: $A/B - C + D \times E - A \times C$



Output:
AB/C-DEx+A

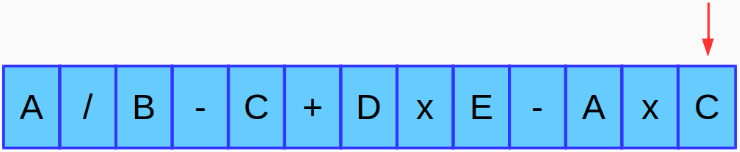


Example: $A/B - C + D \times E - A \times C$

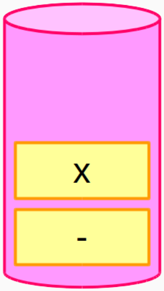


Output:
AB/C-DEx+A

Example: $A/B - C + D \times E - A \times C$



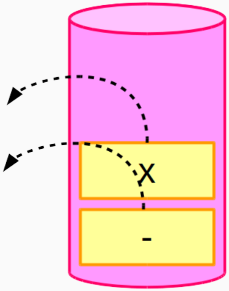
Output:
AB/C-DEx+AC



Example: $A/B - C + D \times E - A \times C$

A	/	B	-	C	+	D	x	E	-	A	x	C

Output:
AB/C-DEx+ACx-



Demo

Please see postfix.

Practice

The example code does not consider (), please implement this feature.

Besides, please implement the calculator again.

Service Simulation

Service simulation:

- The supermarket opens 12-hour a day.
- The customer's service time is a random number from 1 to 4.
- The customer's arrival time is a random number from 1 to 4.

Please write a program to simulate the scenario and answer some questions.

Questions

1. What is the maximum number of customers in the queue?
2. What is the longest/average wait any one customer experience?
3. What happens if the arrival interval is changed from 1-4 minutes to 1-3 minutes?

Demo

Please see service.simulation.

Demo

Please see service.simulation.

Please modify this code to simulate the above questions.

Demo

Please see service.simulation.

Please modify this code to simulate the above questions.

Actually, there is a class called Queuing Theorem.

The End of Programming

Final Reminder

- We are in the CSIE department. Programming is undoubtedly a necessary skill for everyone.
- The only way to improve your skill is to practice more and think more.
- Learning is a never-ending thing and no one can help you.

Congratulations and Good Luck!