Infix, prefix, postfix

$a + b * c + d * e + f$

$a + b * c * d + e / f - g$

Prefix Expression

- Given the above observations, we can write it as + 2 * 3 6.
- Another example: 3 + 4 + 2 * 6. The prefix is + 3 + 4 * 2 6.
- But can we write prefix expressions? We are used to writing infix expressions.
- Our next steps are as follows
  1. Given an infix expression, convert it into a prefix expressions.
  2. Evaluate a prefix expression.

Our Next Steps

- We have two problems. Of these let us consider the second problem first.
- The problem is to evaluate a given prefix expression.
- Our solution closely resembles how we do a manual calculation.

Evaluating a Prefix Expression

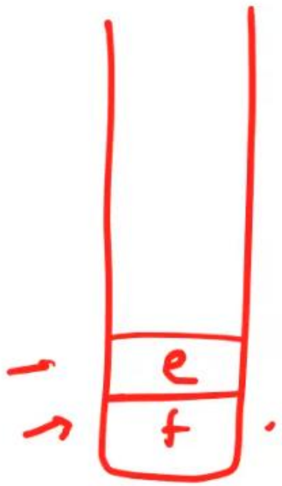
- Some observation(s)
  - The operator precedes the operands.
  - Therefore, the operands are usually pushed to the right of the prefix expression.
  - This suggests that we should evaluate from right to left.
- This helps us in devising an algorithm.
- Imagine that the prefix expression is stored in an array.
  - one operator/operand at an index.

Evaluating a Prefix Expression

- The above suggests the following approach.
- Start from the right side.
- For every operand, push it onto the stack.
- For every operator, evaluate the operator by taking the top two elements of the stack.
  - place the result on top of the stack.

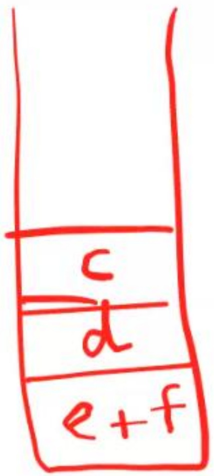
Example to Evaluate a Prefix Expression

- Consider the expression + * + a b + c d + e f.
- Show the contents of the stack and the output at every step.



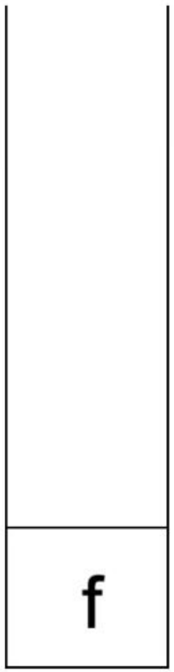
stack

e+f



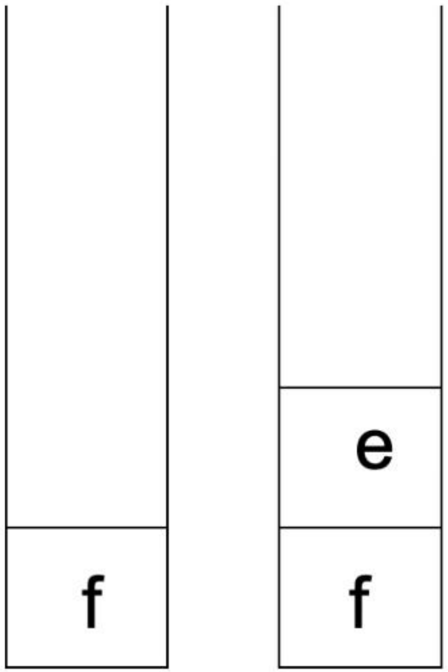
Example

+ * + a b + c d + e f.



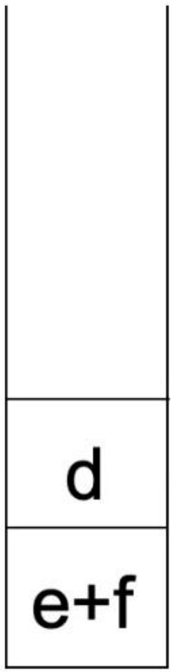
Example

\+ * + a b + c d + e f.



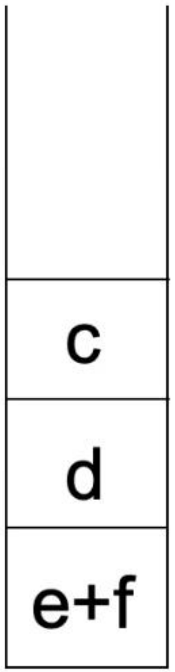
Example

\+ * + a b + c d + e f.



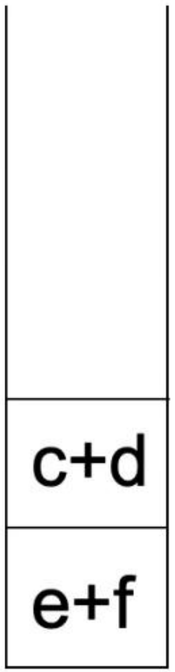
Example

+ * + a b + c d + e f.



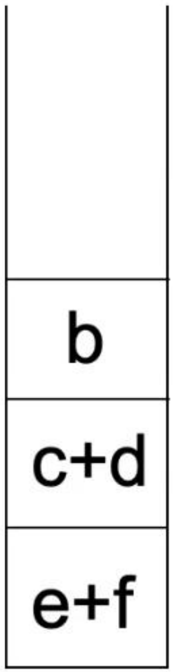
Example

\+ * + a b + c d + e f.



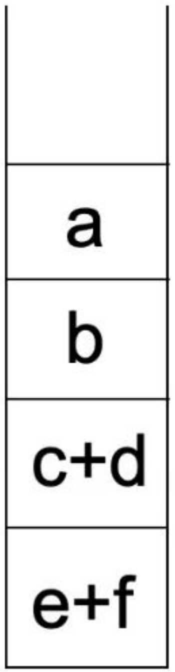
Example

+ * + a b + c d + e f.



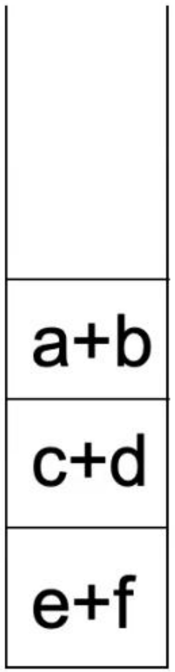
Example

\+ * + a b + c d + e f.



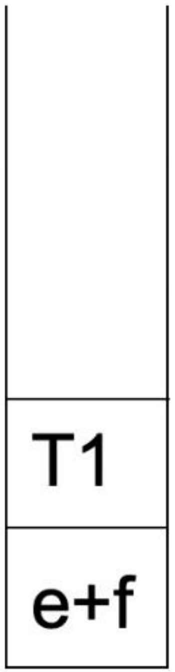
Example

\+ * + a b + c d + e f.



Example

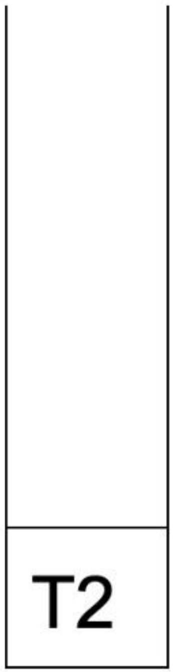
+ * + a b + c d + e f.



T1 = (a+b) * (c+d)

Example

+ * + a b + c d + e f.



T1 = (a+b) * (c+d)

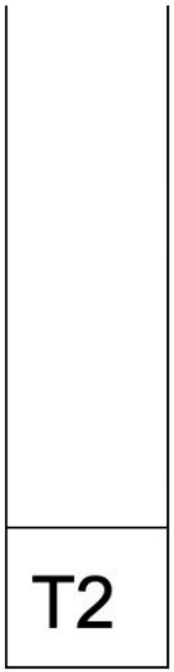
T2 = (T1) + (e+f)

Example

+ * + a b + c d + e f.

(+ab) x (+cd) + (+ef)

$\boxed{+ \times + a b + c d + e f}$



T1 = (a+b) * (c+d)

T2 = (T1) + (e+f)

Ans = (a+b) * (c+d) + (e+f)

Example

ef+

(e+f)

+ef

f o_1

e o_2

+ * + a b + c d + e f.

(+ab) × (+cd) + (+ef)

$\boxed{+ \times + ab + cd + ef}$

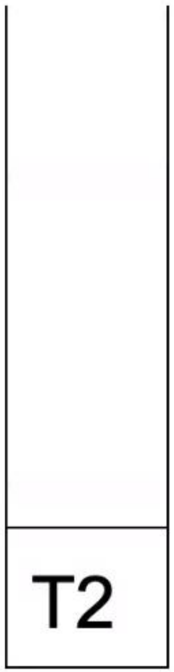
T1 = (a+b) * (c+d)

o_1 (*) o_2

e o_1

f o_2

T2 = (T1) + (e+f)



... (c+d) + (e+f)

e+f   +ef

Algorithm for Evaluating a Prefix Expression

```
Algorithm EvaluatePrefix(E)
begin
        Stack S;
        for i = n down to 1 do
        begin
        if E[i] is an operator, say o then
                operand1 = S.pop();
                operand2 = S.pop();
                value = operand1 o operand2;
                S.push(value);
        else
                S.push(E[i]);
        end-for
end-algorithm
```

- Here, n refers to the number of operators + the number of operands.
- The time taken for the above algorithm is linear in n.
  - There is only one for loop which looks at each element, either operand or operator, once.
- We will see an example next.

Reading Exercise

- We omitted a few details in our description.
- Some of them are:
  - How to handle unary operators?
  - How can this be extended to ternary operators?

- Another possibility is to use postfix expressions.
  - Also called as Reverse Polish Notation.
- They can be evaluated left to right with a stack.
- Try to arrive at the details.

Back to The First Question

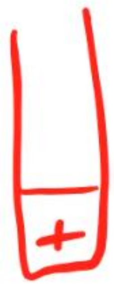
- Let us now consider how to convert a given infix expression to its prefix/postfix equivalent.
- The issues
  - Operands not easily known
  - There may be parentheses also in the expression.
  - Operators have precedence.

$a + b * c$

$a + (b * c)$

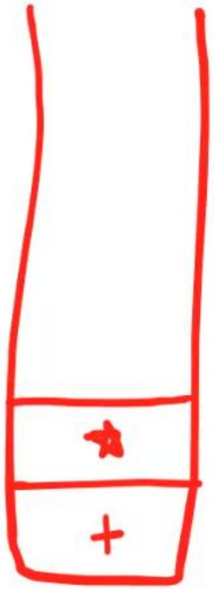
$ab+$

$abc*+$



Lets look at postfix first

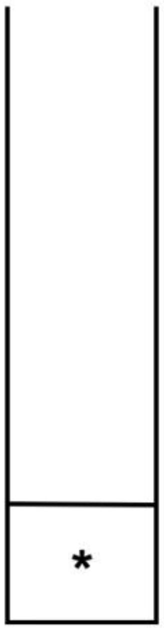
$A+B*C-D*E \longrightarrow ABC*+DE*-$



$ABC*+DE*-$

infix-prefix

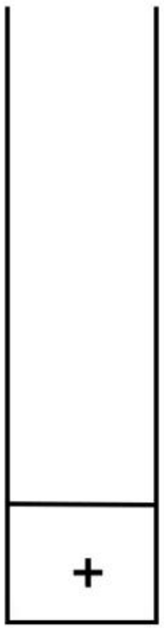
- Let us consider an expression of the form a + b + c * d + e * f.



f e

infix-prefix

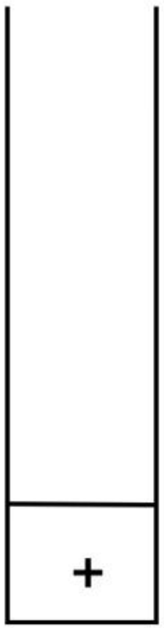
- Let us consider an expression of the form a + b + c * d + e * f.



f e *

infix-prefix

- Let us consider an expression of the form a + b + c * d + e * f.



f e * d

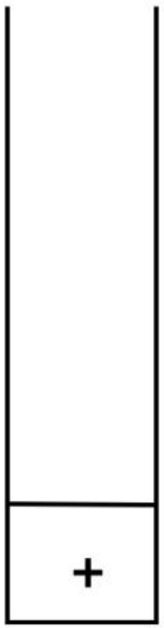
infix-prefix

- Let us consider an expression of the form a + b + c * d + e * f.

f e * d c

infix-prefix

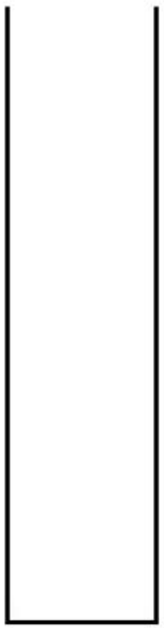
- Let us consider an expression of the form a + b + c * d + e * f.



f e * d c *

infix-prefix

- Let us consider an expression of the form a + b + c * d + e * f.



f e * d c * + b + a +

Invert as:

+ a + b + * c d * e f

Reading Exercise

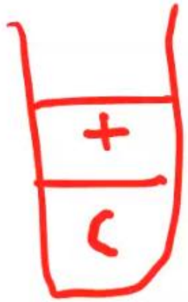
- Read or devise ways to handle parentheses.
  - Open parentheses indicates the start of a subexpression, closing parentheses indicates the end of the subexpression.
  - Important to keep track of these.

- Similarly, how to handle unary operators?

$a + b * c$

$\rightarrow a + (b * c)$

$\rightarrow (a + b) * c$



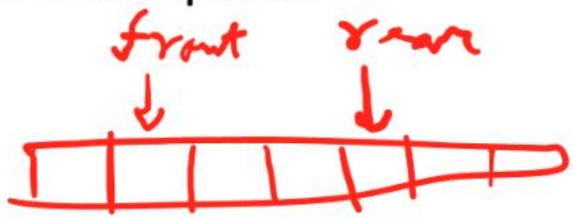
ab

Lets move to Queue

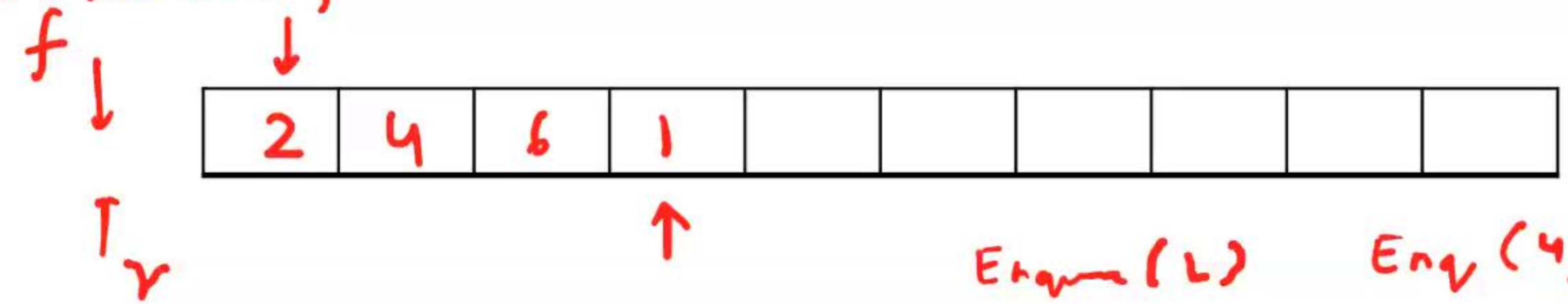
- Consider a different setting.
- Think of booking a ticket at a train reservation office.
  - When do you get your chance?
- Think of a traffic junction.
  - On a green light, which vehicle(s) go(es) first.?
- Think of airplanes waiting to take off.
  - Which one takes off first?

The Queue

- The fundamental operations for such a data structure are:
  - Create : create an empty queue
  - Enqueue : Insert an item into the queue
  - Dequeue : Delete an item from the queue.
  - size : return the number of elements in the queue.



front = rear = -1



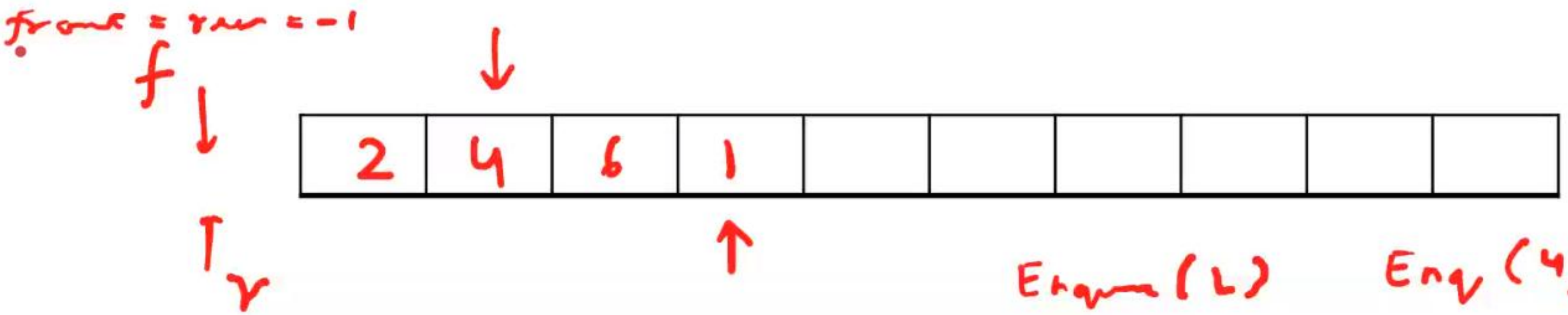
Enqueue(2) Enq(4) Enq(6)

Enq(1)

```
IsEmpty()
begin
if front==-1 && rear == -1
    return true;
else
    return false
end
```

```
Enqueue(x)
begin
if rear == MAXSIZE then
    return;
else if IsEmpty()
    front ← rear ← 0;
else
    rear = rear+1;
Queue[rear] = x;
end
```

```
Dequeue(x)
begin
if IsEmpty()
    return;
else if front==rear
    front ← rear ← -1;
else
    front = front +1;
end
```



Enqueue(2) Enq(4) Enq(6)

Enq(1)

Deq

```
IsEmpty()
begin
if front==-1 && rear == -1
    return true;
else
    return false
end
```

```
Enqueue(x)
begin
if rear == MAXSIZE then
    return;
else if IsEmpty()
    front ← rear ← 0;
else
    rear = rear+1;
Queue[rear] = x;
end
```

```
Dequeue(x)
begin
if IsEmpty()
    return;
else if front==rear
    front ← rear ← -1;
else
    front = front +1;
end
```

front = rear = -1

f

r

2	4	6	1	7	17	4	1	3	5

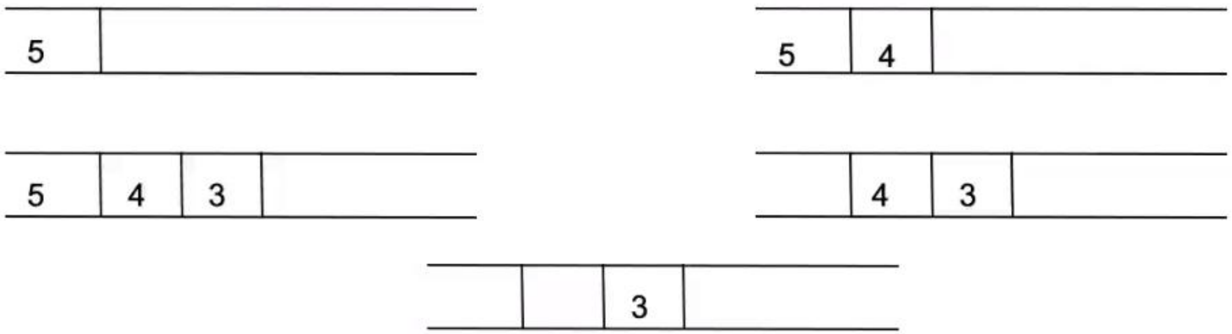
Enqueue(2) Enq(4) Enq(6) Enq(1) Deq() Deq() Deq() Enq(7) Enq(17) Enq(5) Enq(1) Enq(1) Enq(4)

```
IsEmpty()
begin
if front==-1 && rear == -1
    return true;
else
    return false
end
```

```
Enqueue(x)
begin
if rear == MAXSIZE then
    return;
else if IsEmpty()
    front ← rear ← 0;
else
    rear = rear+1;
Queue[rear] = x;
end
```

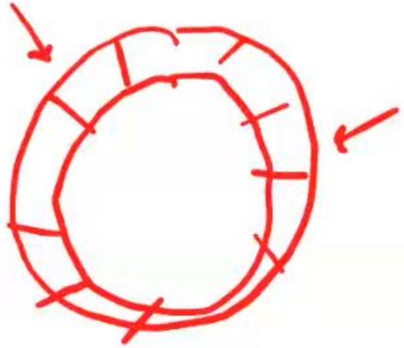
```
Dequeue(x)
begin
if IsEmpty()
    return;
else if front==rear
    front ← rear ← -1;
else
    front = front +1;
end
```

Queue Example



- Starting from an empty queue, consider the following operations.
  - Enqueue(5), Enqueue(4), Enqueue(3), Dequeue(), Dequeue()
- The result is shown in the figure above.

Other Variations of the Queue



- To save space, a circular queue is also proposed.
- Operations that update front and rear have to be based on modulo arithmetic.
- The circular queue is declared full when (rear+1)%N == front

A Sample Application with Stack and Queue

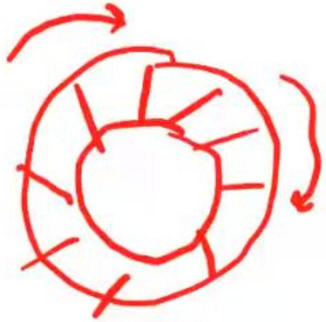
- A palindrome is a string that reads the same forwards and backwards, ignoring non-alphabetic characters.
- Examples:
  - Malayalam
  - Wonton? not now
  - Madam, i'm Adam
- Problem: Given a string, determine if it is a palindrome.
  - May not know the length of the string apriori.

r r f = 3

7	8	7	3	4	1	7	19	12	2

r = 2

Enq (7)



```
IsEmpty()
begin
if front==-1 && rear == -1
    return true;
else
    return false
end
```

```
Enqueue(x)
begin
if (rear+1)%N+1 == front then
    return;
else if IsEmpty()
    front ← rear ← 0;
else
    rear = (rear+1)%N;
Queue[rear] = x;
end
```

```
Dequeue(x)
begin
if IsEmpty()
    return;
else if front==rear
    front ← rear ← -1;
else
    front = (front +1)%N;
end
```

A Sample Application with Stack and Queue

- Need to compare the first character with the last character.
- So, store the characters in a stack and a queue also.
- Once notified of the end of the string, compare the top of the stack with the front of the queue.
  - Continue until both the stack and the queue are empty.