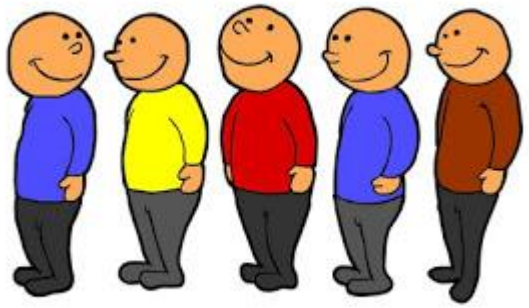
Week 4: Queue & Stack ADT

CS-250 Data Structure and Algorithms

DR. Mehwish Fatima | Assist. Professor
Department of AI & DS | SEECS, NUST

Queue

Queue ADT

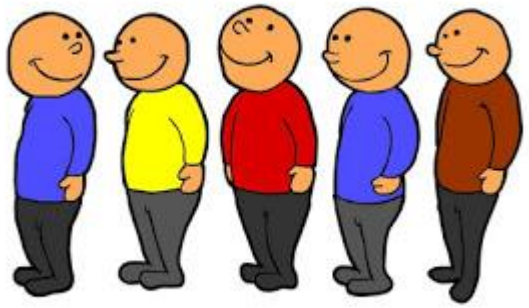


- Queue is a data structure that can be used to store data which can later be retrieved in the first in first out (FIFO) order which is explicit linear ordering.

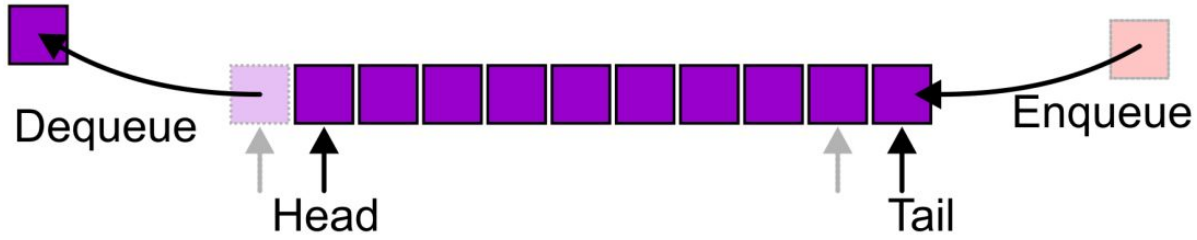
- Queue is an ordered-list in which all the insertions and deletions are made at two different ends to maintain the FIFO order.

- At the logical level, a queue is an ordered group of homogeneous items or elements.

Queue ADT

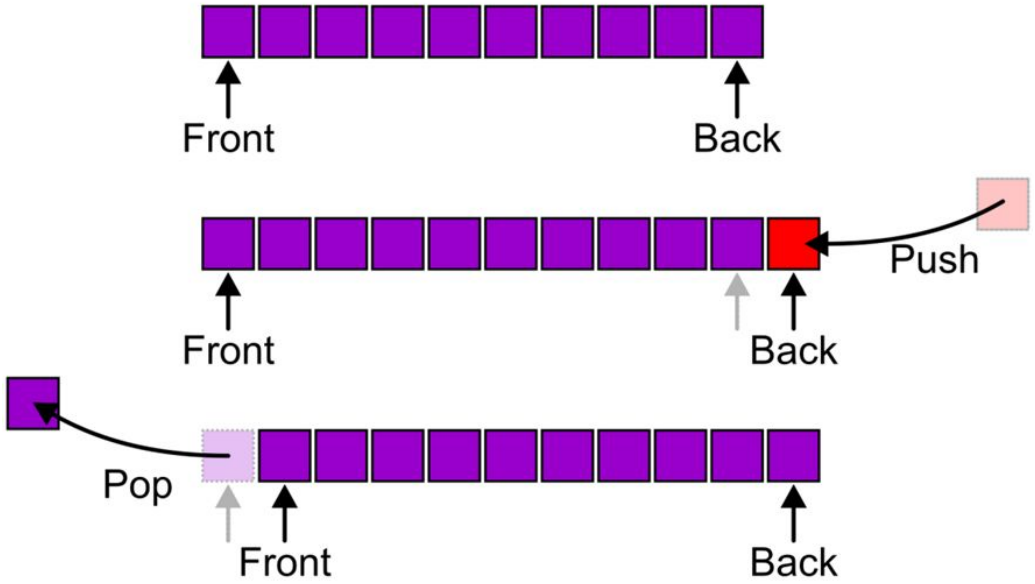


- At the logical level, a queue is an ordered group of homogeneous items or elements.
  - The removal of existing items can take place at one end only and the addition of new items can take place on other end.
  - Queue is RESTRICTED type of List, So Insertion and Deletion can take place only at opposite ends of list.

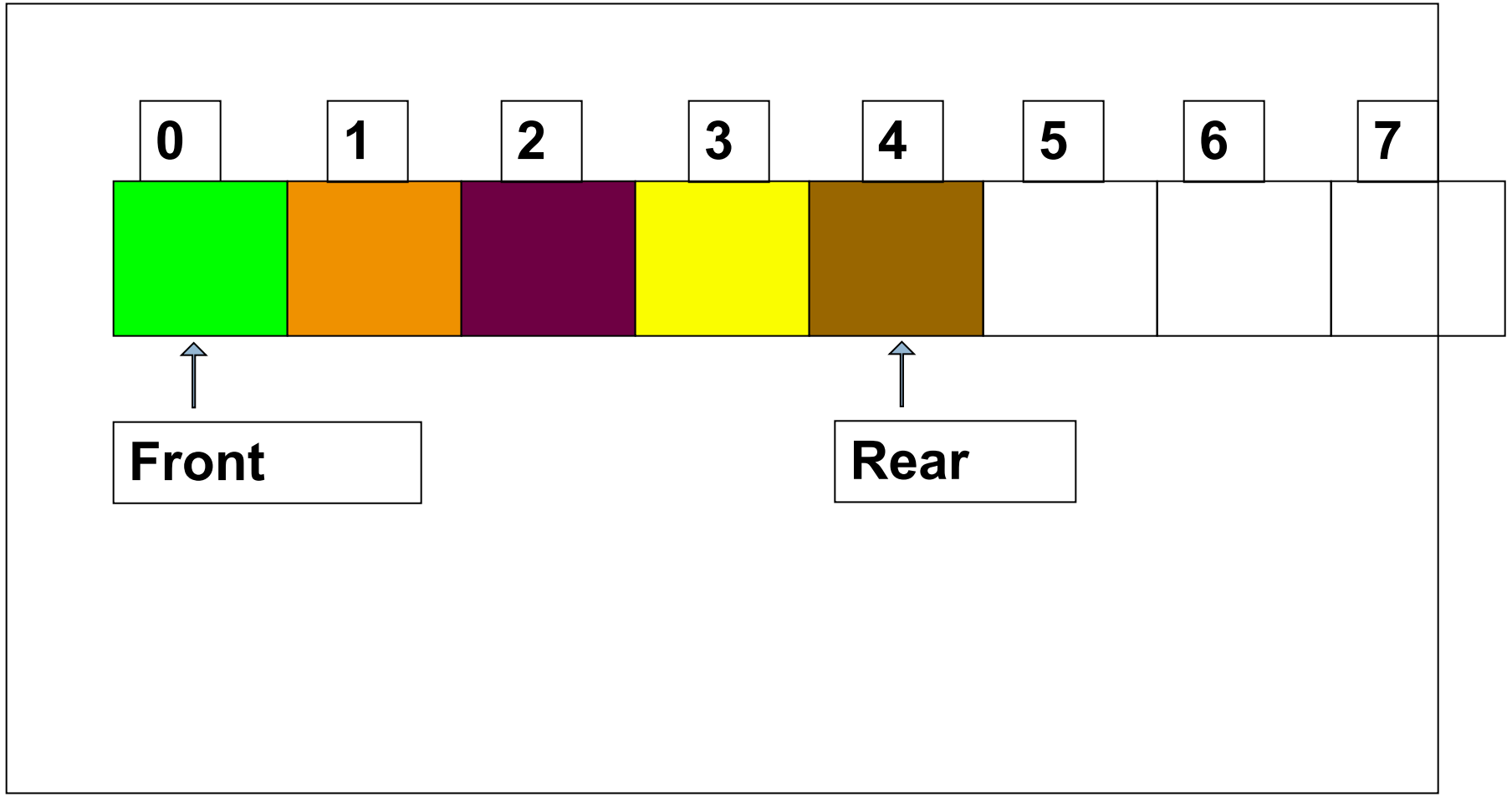


Operations on Queue

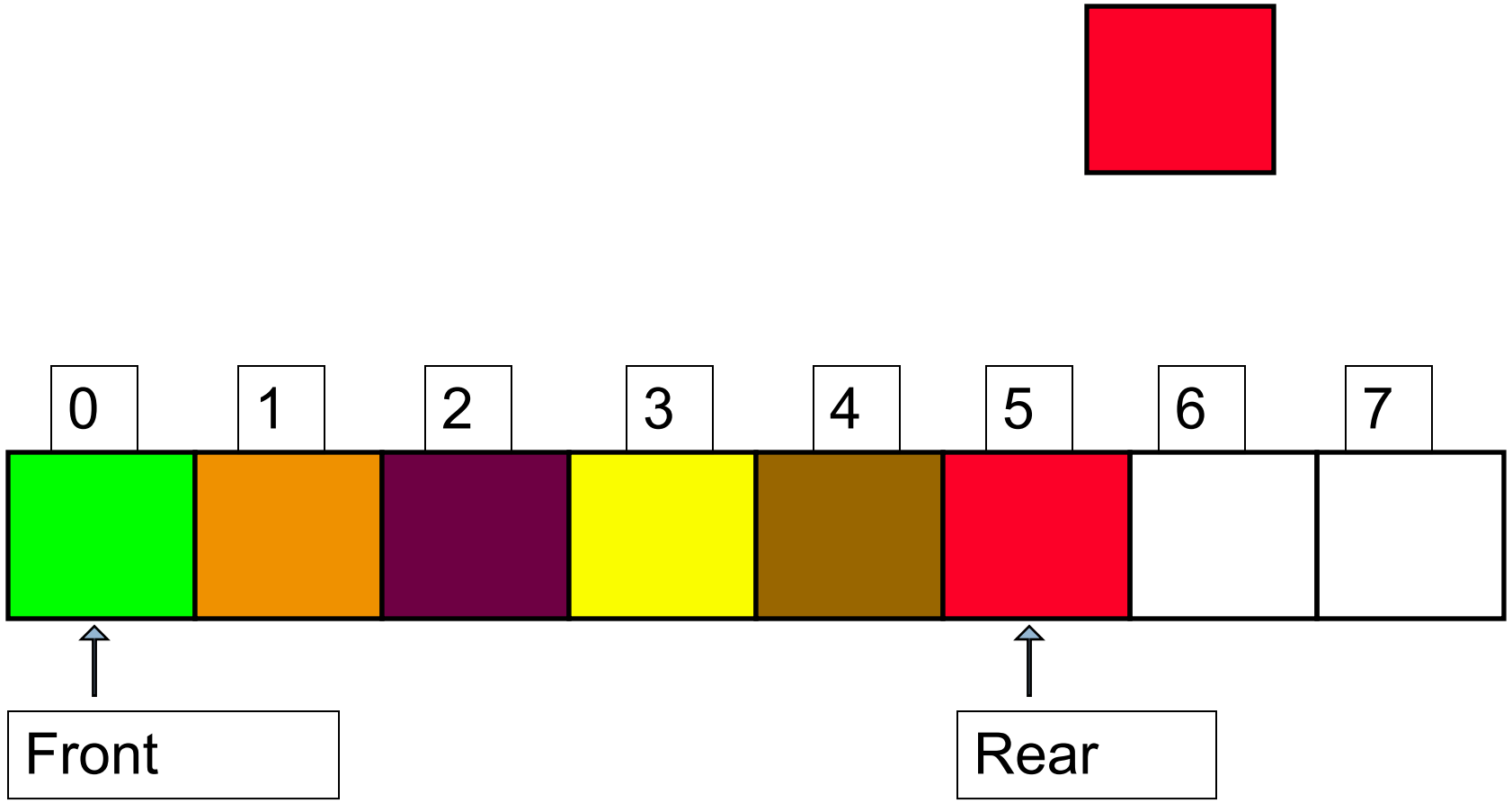
- The logical picture of the structure provides only half of the definition of an abstract data type.
- An ordered list where only the oldest item is accessible.
- Elements are added to the rear/back and removed from the front



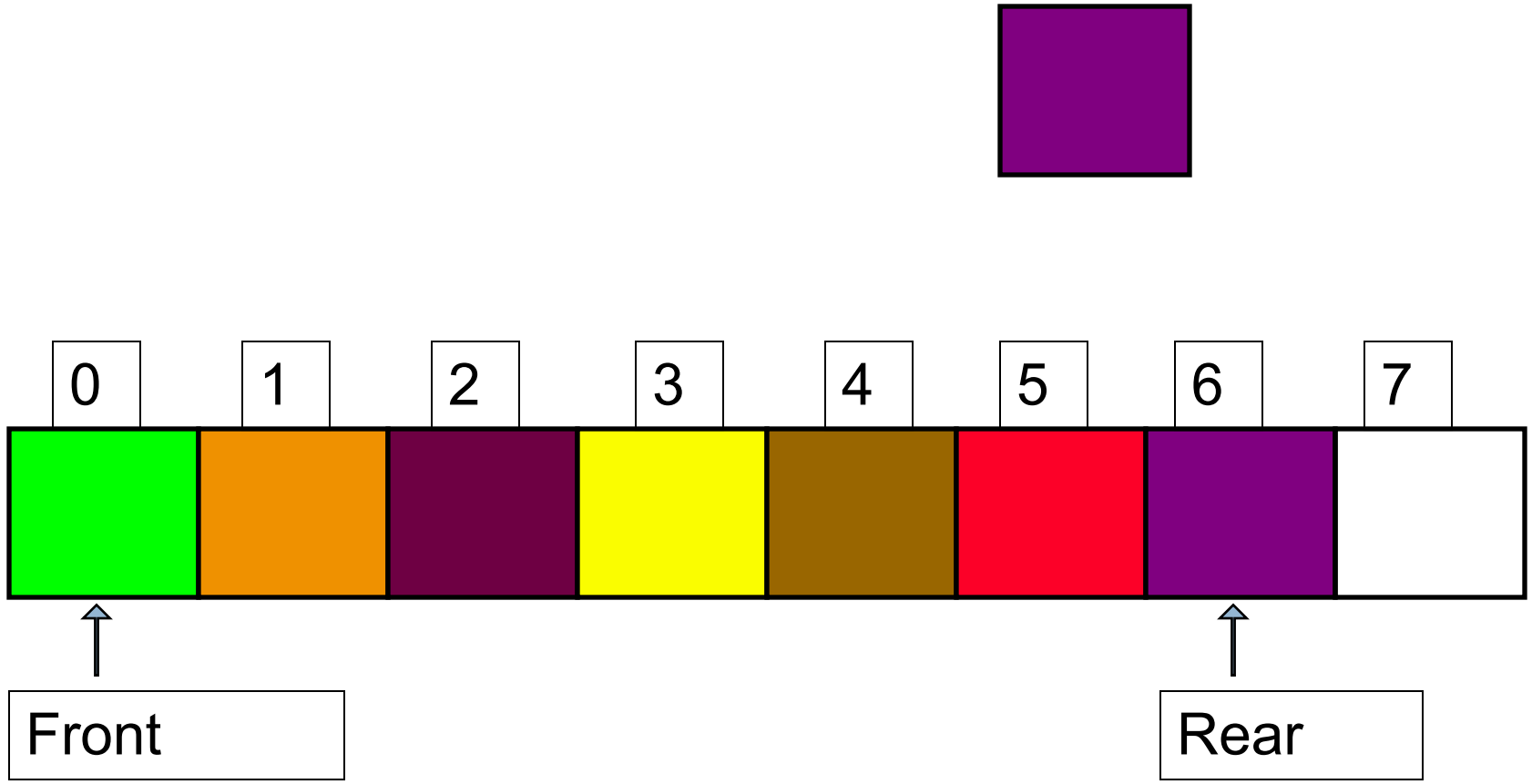
Example



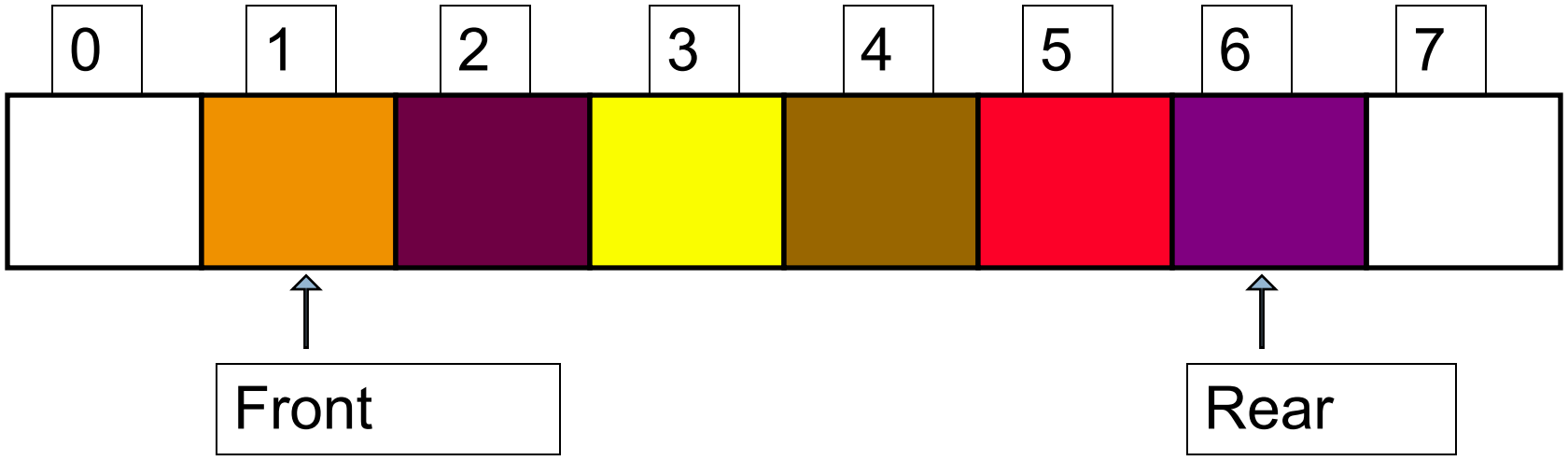
Insert operation

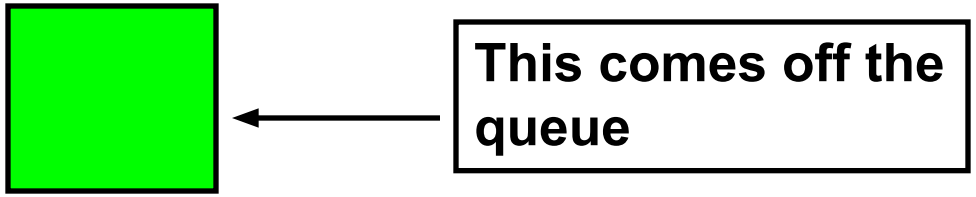


Insert operation

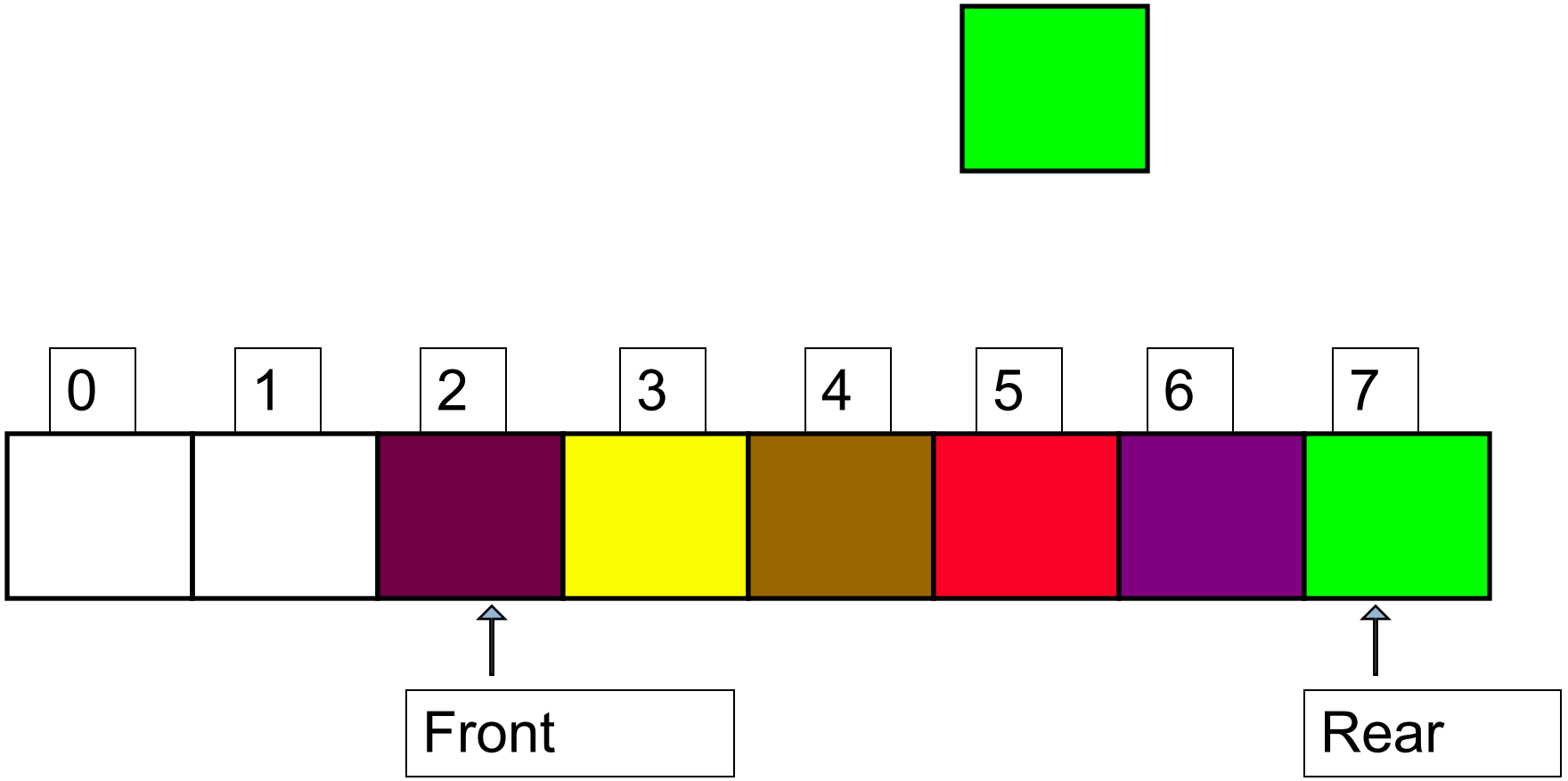


Delete





Insert



Operations on Queue

Main Queue operations:

- **FRONT():** Retrieve the element from the front.
- **ENQUEUE():** Insert new element at Back of the queue.
- **DEQUEUE():** Remove element at Front of the queue.
- **ISEMPTY():** Return true if it is an empty queue; return false otherwise.
- **ISFULL():** Return true if it is full queue; return false otherwise.
- **SIZE():** Return the total number of elements present in queue.

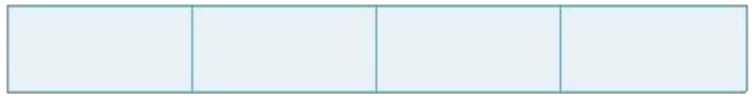
Applications

- The most common application is in client-server models
  - Multiple clients may be requesting services from one or more servers
  - Some clients may have to wait while the servers are busy
  - Those clients are placed in a queue and serviced in the order of arrival
- Grocery stores, banks, music player list and airport security use queues
- The SSH Secure Shell and SFTP are clients
- Most shared computer services are servers:
  - Web, file, ftp, database, mail, printers, etc.

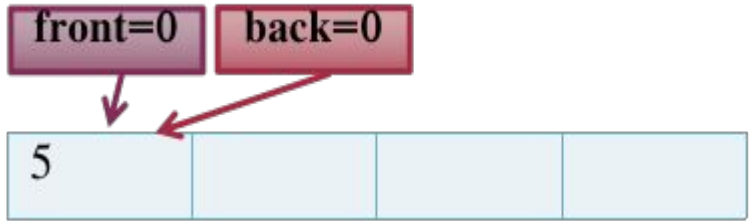
Array-based Queue

Enqueue

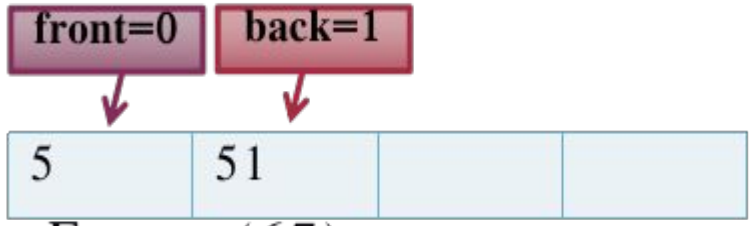
At start: front =-1 , back = -1



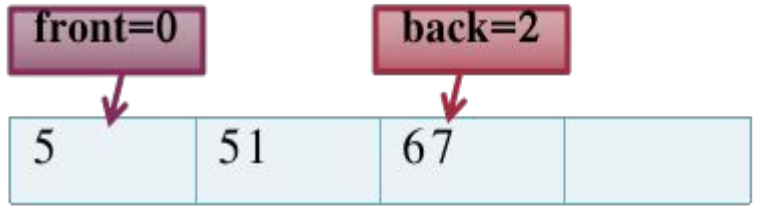
Enqueue(5)



Enqueue(51)

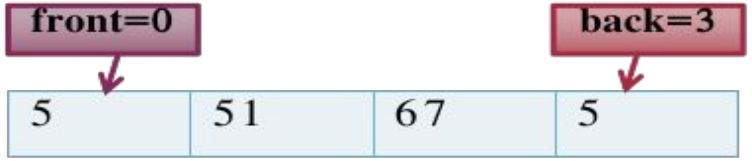


Enqueue(67)



Enqueue(5)

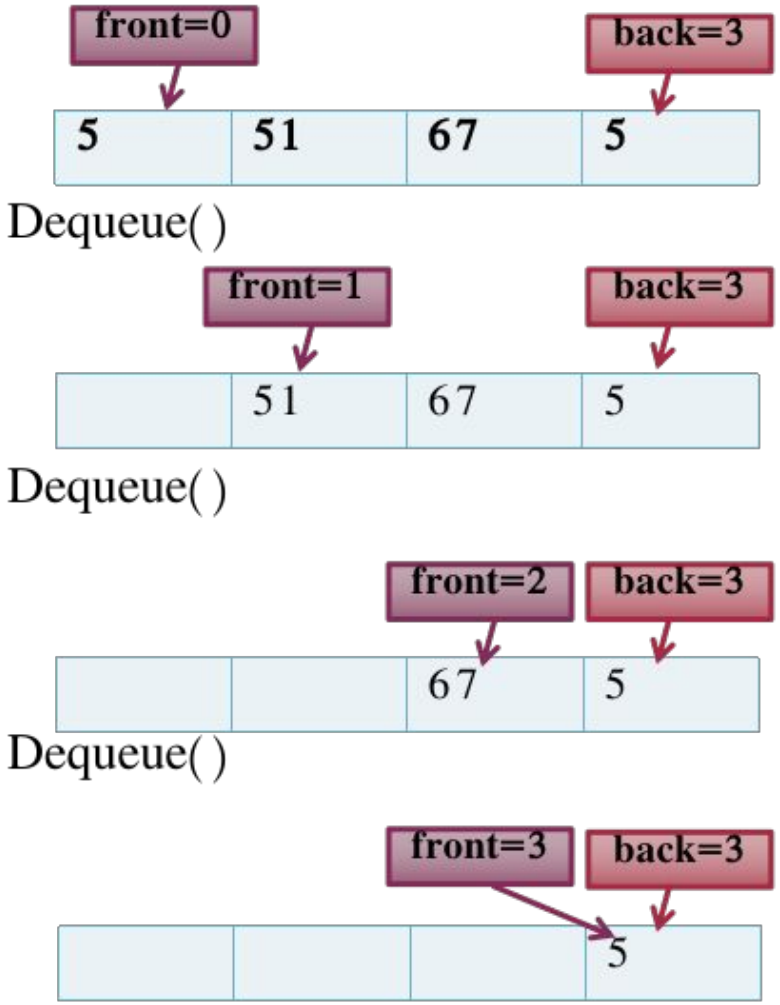
Eequeue(91) Error



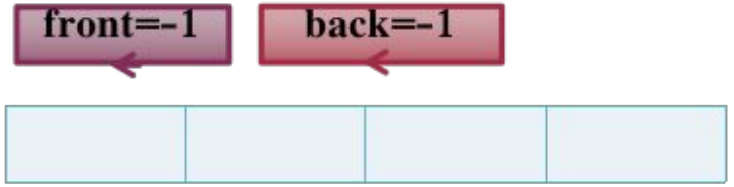
No more elements can be inserted if queue is full.

How to know if queue is full ?

Dequeue



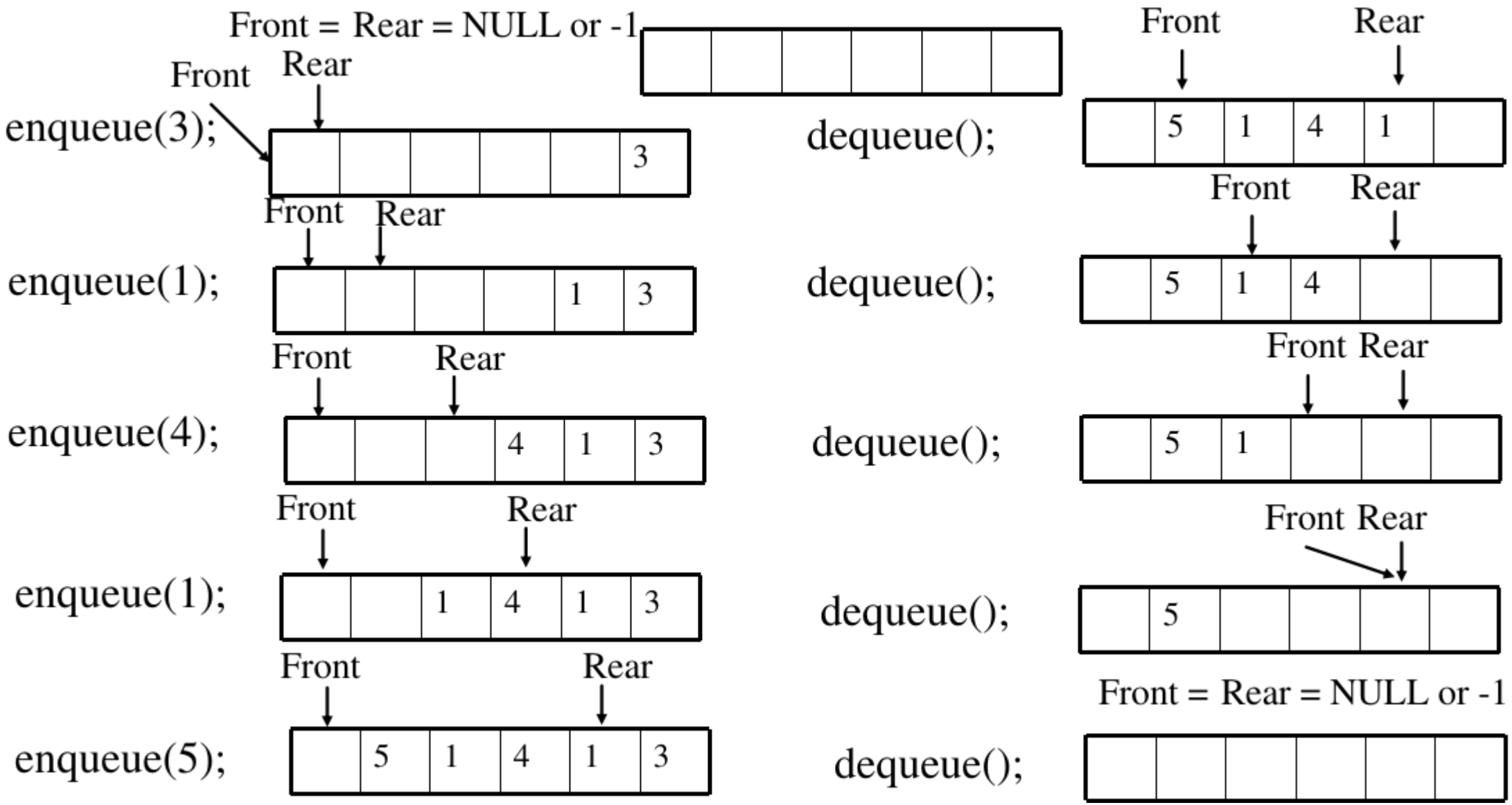
Dequeue( )



Dequeue( ) Error

No more elements can be removed if queue is empty.

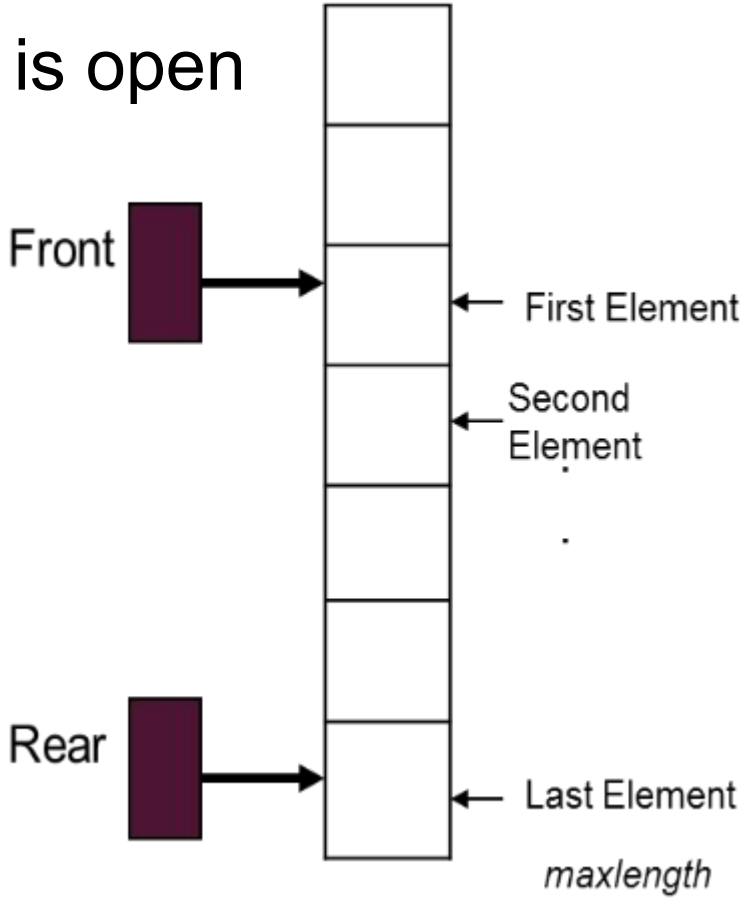
How to know if queue is empty ?


Front = Rear = NULL or -1
Front
Rear
enqueue(3);
3
Front
Rear
enqueue(1);
1
3
Front
Rear
enqueue(4);
4
1
3
Front
Rear
enqueue(1);
1
4
1
3
Front
Rear
enqueue(5);
5
1
4
1
3
dequeue();
Front
Rear
5
1
4
1
dequeue();
Front
Rear
5
1
4
dequeue();
Front
Rear
5
1
dequeue();
Front
Rear
5
dequeue();
Front = Rear = NULL or -1

An Array Implementation of Queue

- Consider an array based LIST in vertical cell position.

- One end of list is open for insertion only and other is open for deletion only so we have 2 possibilities:
  - Either Pick insertion at start and deletion at end.
  - Or Pick insertion at end and deletion at start.
  - Insertion & Deletion At middle is not applicable.

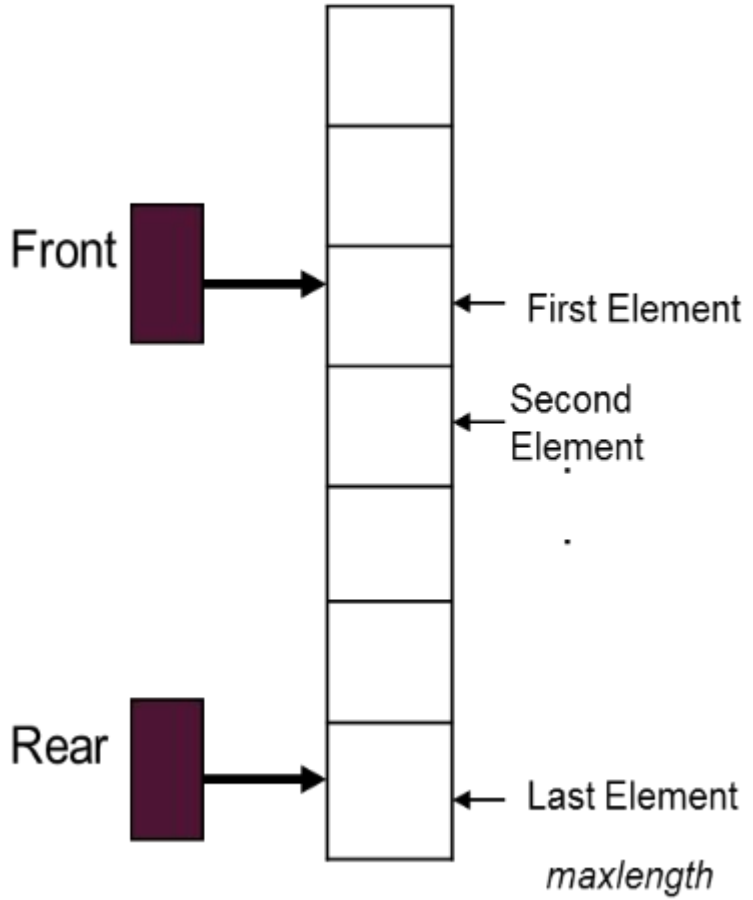


An Array Implementation of Queue

- For maintaining open ends, we have two reference points called
  - **Front** (for Deletion) and
  - **Back/Rear** (for insertion).

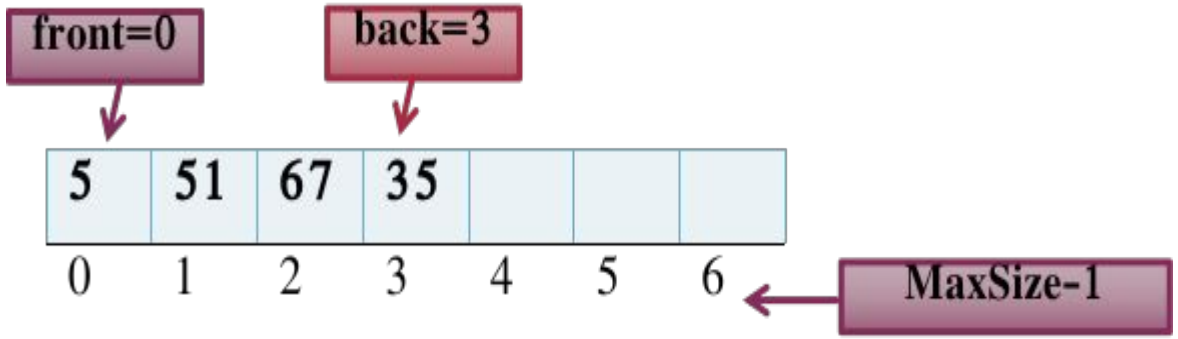
- When there is only one value in the queue, both rear/back and front have same index.

- When queue is empty, front and back will be -1.



Implementation and Performance

- Queue can be implemented with simple array.



- Performance
  - The space used is O(n)
  - Both operations will take **O(1)**
- Limitations
  - The maximum size of the queue must be defined a priori , and cannot be changed.
  - Trying to enqueue a new element into a full queue causes an implementation-specific exception.

Linked-based Queue

A Linked Implementation of Queue

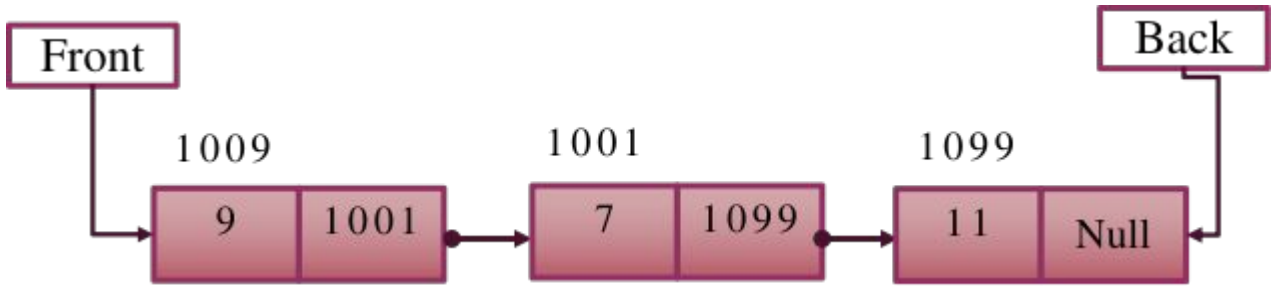
- Consider a dynamic list which one end of list is open for insertion only and other is open for delete only. so we have 2 possibilities:
  - Either Pick insertion at start and deletion at end.
  - Or Pick insertion at end and deletion at start.
    - Insertion & Deletion At middle is not applicable.

A Linked Implementation of Queue

- For maintaining open ends, we have two reference points called
  - Front (for Deletion) and
  - Back/Rear (for insertion).

- When there is only one value in the queue, both rear/back and front point the same node.

- When queue is empty, front and back will be null.



Circular Queue

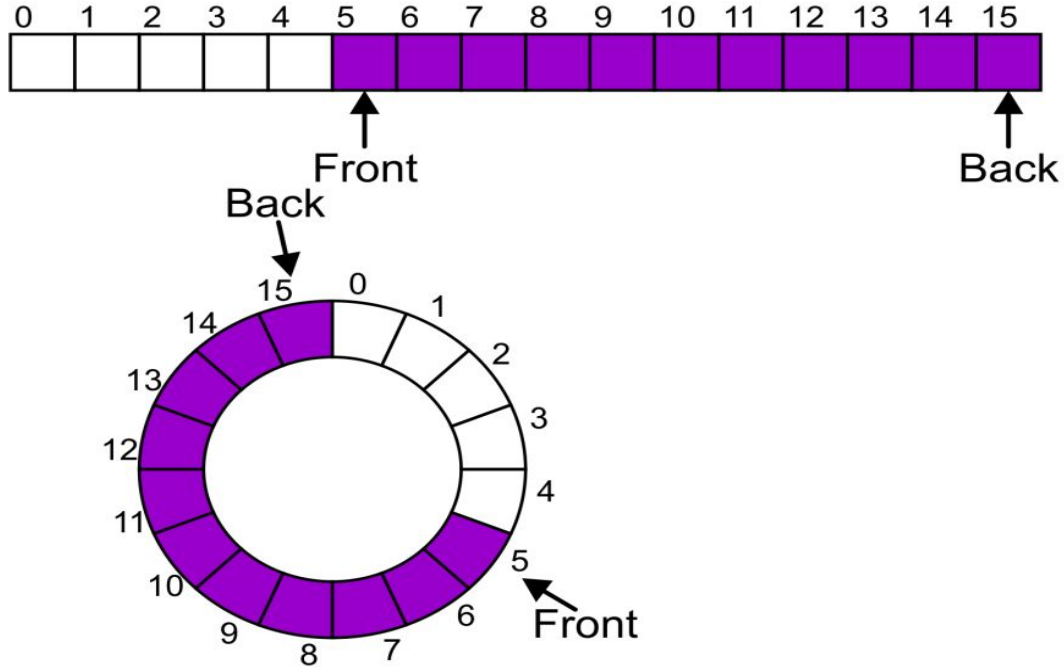
Circular Queue

- This simple array implementation of queue has a problem
  - It is wasting space
  - Because front is moving towards back and leaving empty space behind
  - Elements are stored towards end of array and front may be empty due to dequeue
  - To resolve this problem array can be used as a circular array.
    - When we reached at end of array, start from beginning

- For linked based implementation, use doubly list for **Circular Queues**.

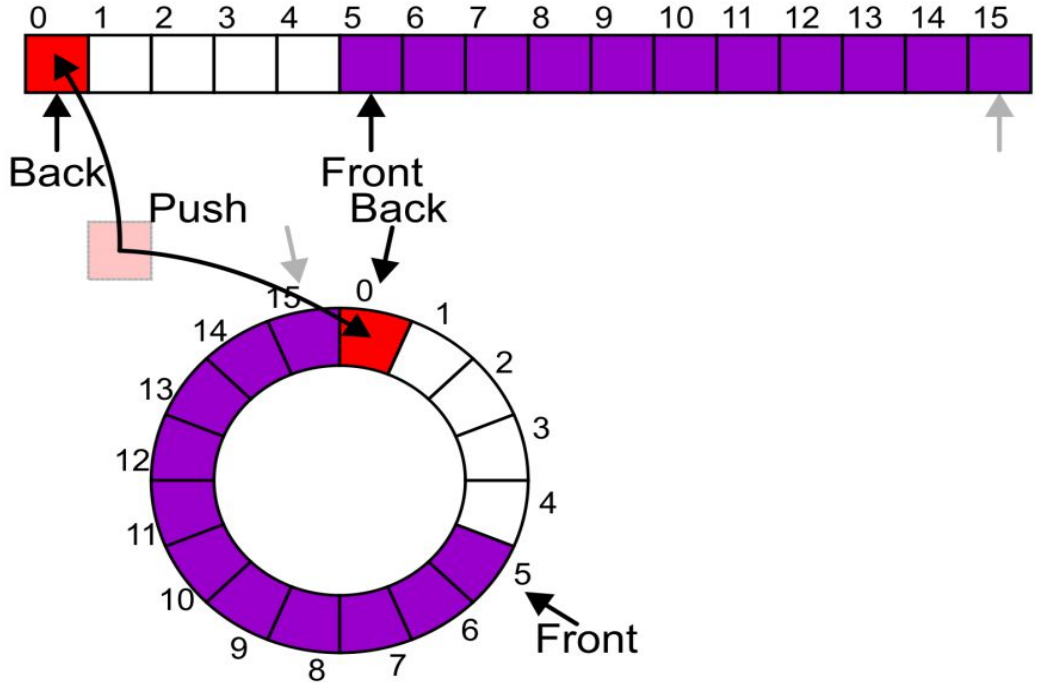
Simple vs Circular Queue

- In circular queue the last node is connected back to the first node to make a circle.
- Instead of viewing the array on the range 0, …, 15, consider the indices being cyclic:
  0, 1, …, 15, 0, 1, …

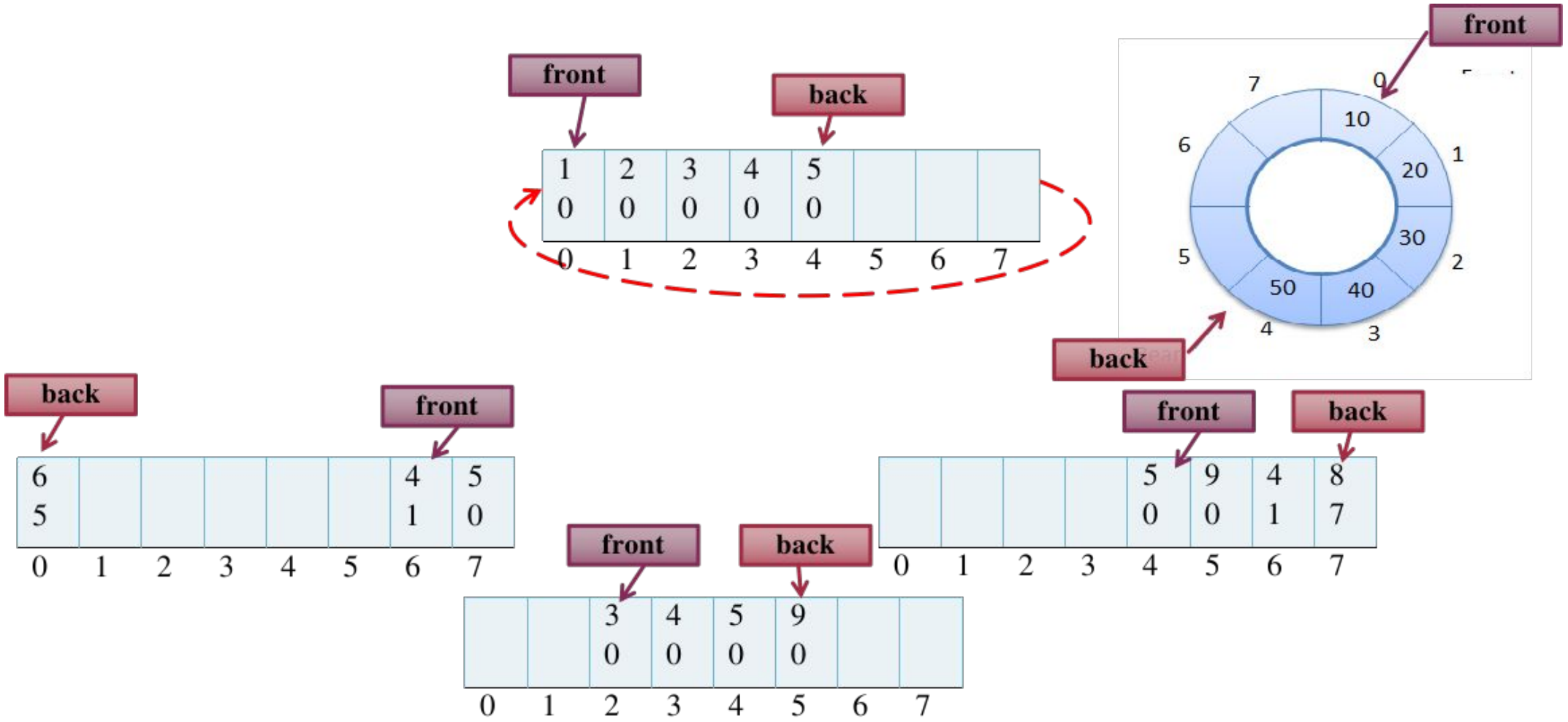


Simple vs Circular Queue

- In circular queue the last node is connected back to the first node to make a circle.
- Instead of viewing the array on the range 0, …, 15, consider the indices being cyclic:
  0, 1, …, 15, 0, 1, …



Circular Queue



Circular Queue

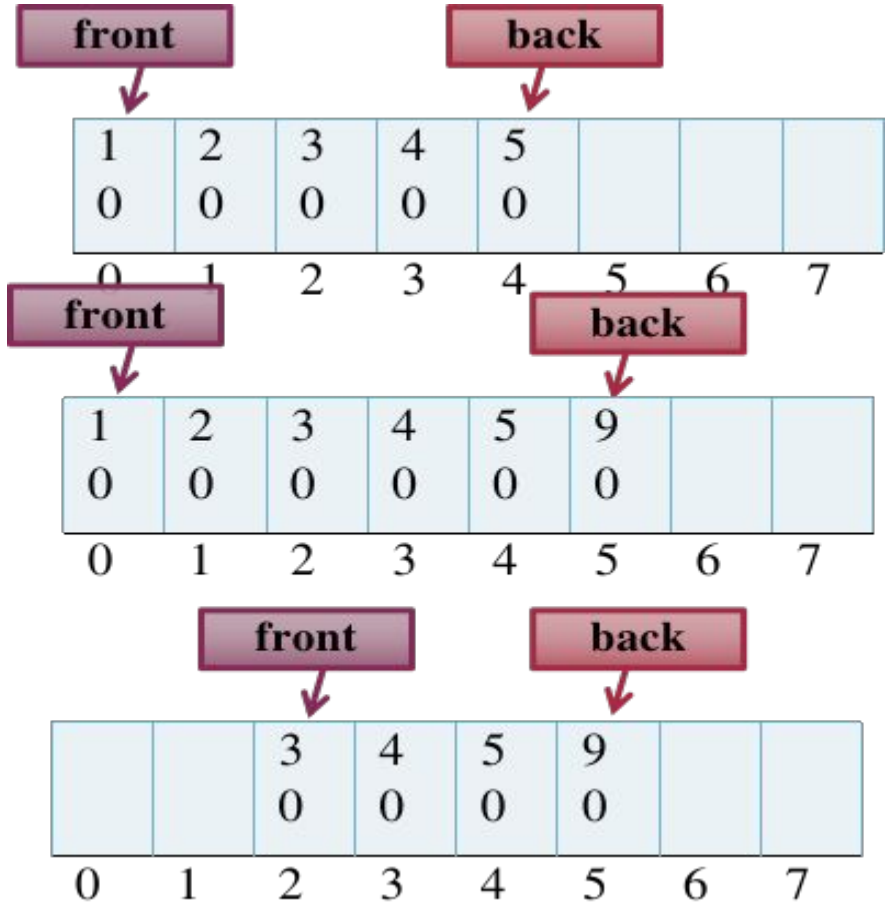
- Circular queue is a simple queue, But logically it says that queue[0] comes after queue[MaxSize-1]

- Allow rear to wrap around the array.
  - use module arithmetic
  - rear = (rear + 1) % queueSize;

- Elements can be anywhere in queue.

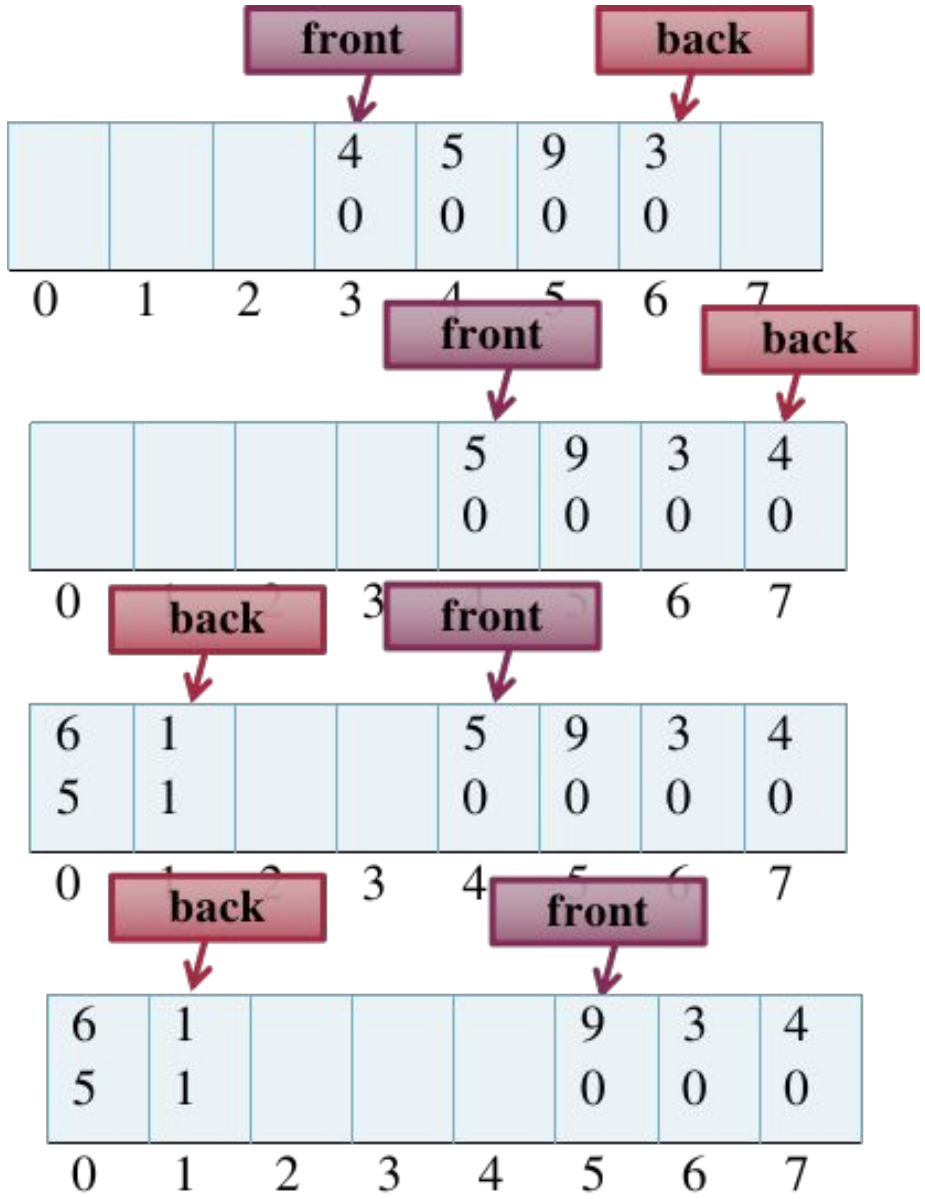
How it Works

- In circular queue, both front and back move in clockwise direction
  - Let say after 5 Enqueue(item) state of queue is:
  - Now perform following operations:
  - Enqueue(90)
  - Dequeue() Dequeue()

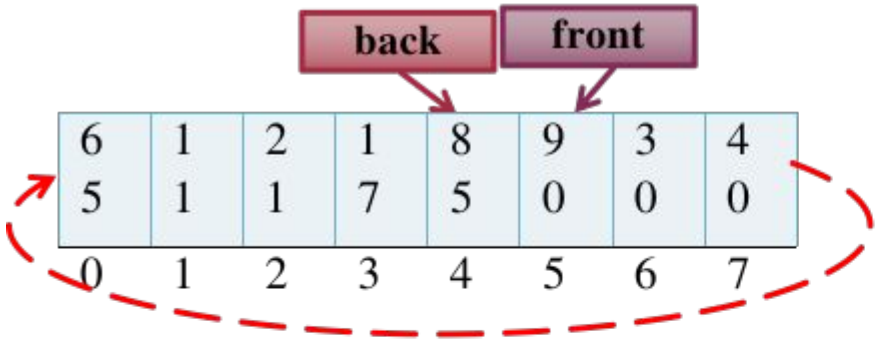


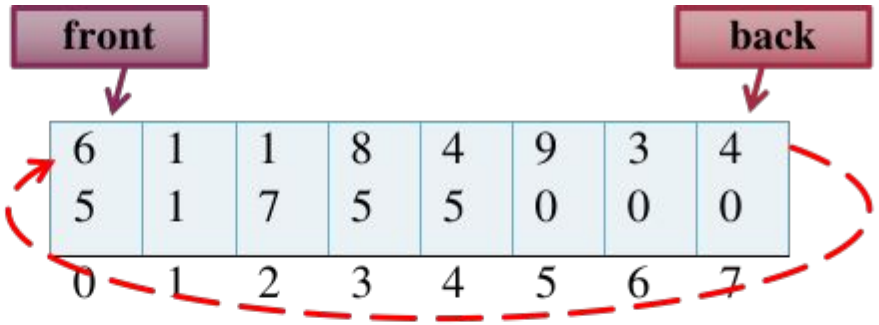
How it Works

- Enqueue(Dequeue())
  - Remove element from front and enqueue it
- Enqueue(Dequeue())
- Enqueue(65) Enqueue(11)
- Dequeue()

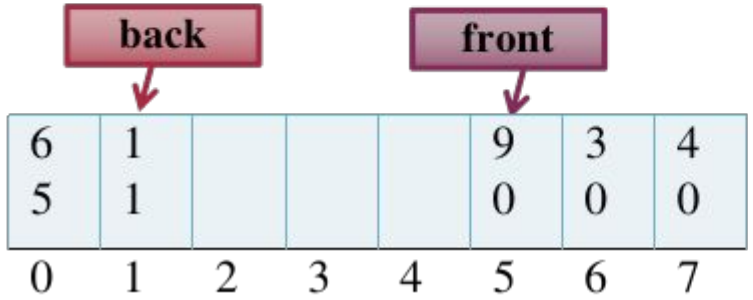


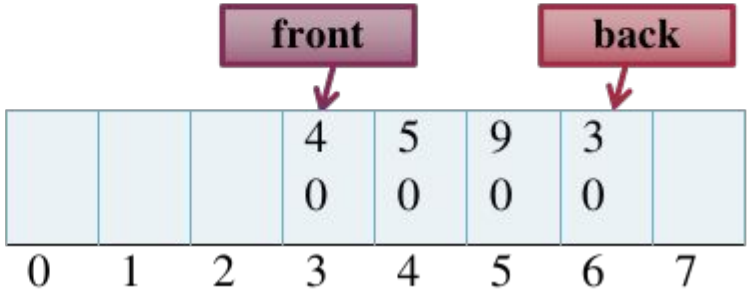
- How to find if Circular Queue is Empty or Full?
  - Front== -1 or Back== -1 (Empty) Front== (Back+1)%MAX_SIZE
- Full?





- Its size?





MaxSize=8

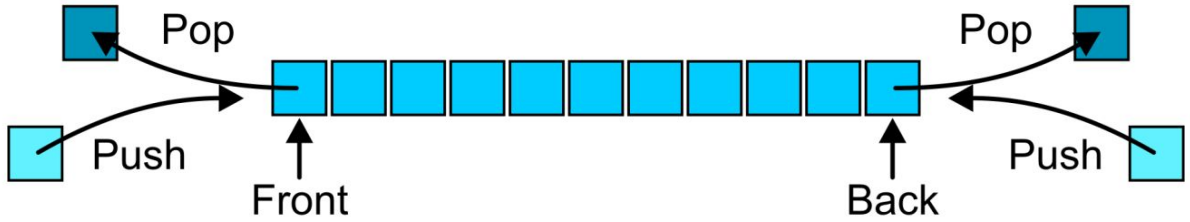
Double Ended Queue

Double Ended Queue

- A variation of linear queue, where insertions and deletions take place at both ends.
  - Uses a explicit linear ordering

- The operations will be called

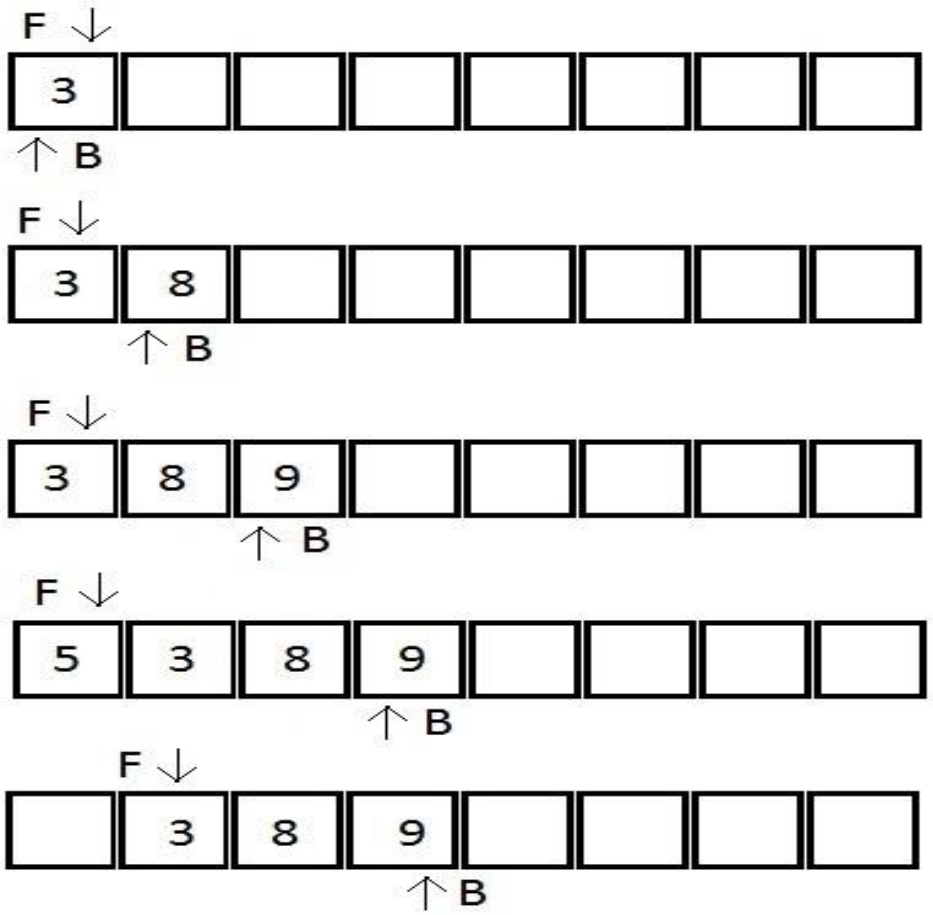
Front	Back
FrontInsert	BackInsert
RemoveFront	RemoveBack

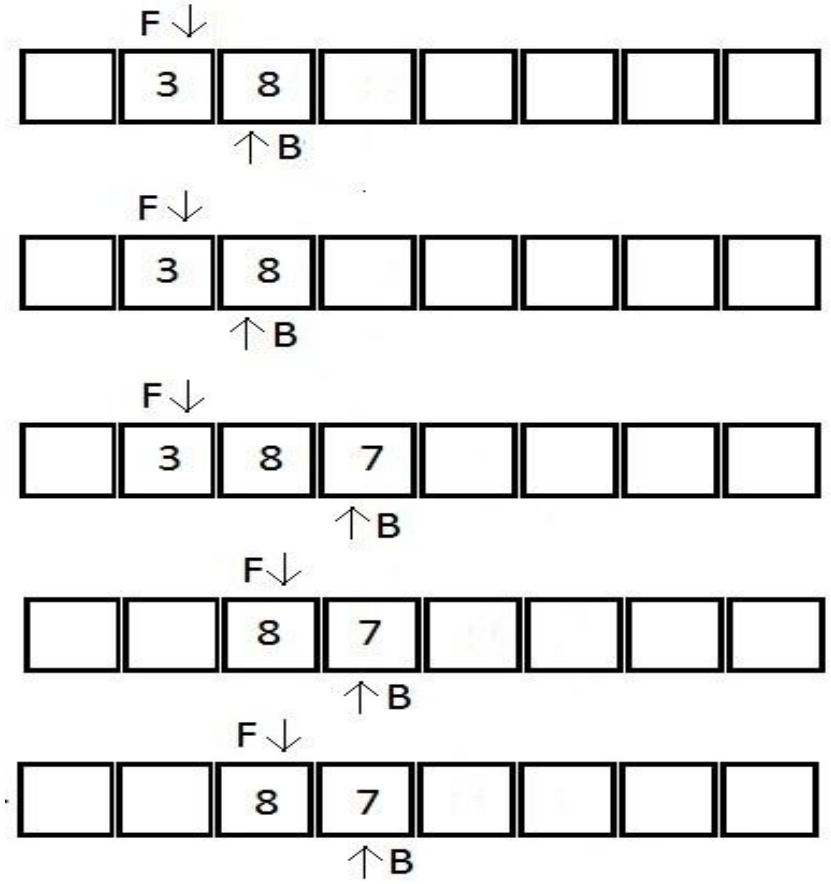


- It is an undefined operation to access or remove from an empty deque

Double Ended Queue

insertFront(3), insertBack(), insertBack(9), insertFront(5), removeFront(), eraseBack(), insertBack(7), removeFront()





Priority Queue

Priority Queue

- A priority queue is a collection of zero or more items, associated with each item is a priority.
  - In a normal queue the enqueue operation add an item at the back of the queue, and the dequeue operation removes an item at the front of the queue.
  - In priority queue, enqueue and dequeue operations consider the priorities of items to insert and remove them.
  - Priority queue does not follow "first in first out" order in general.
    - The highest priority can be either the most minimum value or most maximum
    - we will assume the highest priority is the minimum.

Priority Queue

- Priority queue does not follow "first in first out" order in general.
  - The highest priority can be either the most minimum value or most maximum
  - we will assume the highest priority is the minimum.

- Examples
  - Process scheduling
    - Few processes have more priority
  - Job scheduling
    - N Jobs with limited resources to complete

Priority Queue ADT

- A priority queue has following operations
  - Enqueue(item)
  - Dequeue(): remove the item with the highest priority
  - Find() return the item with the highest priority

Priority Queue ADT

- A priority queue has following operations
  - Enqueue(item)
  - Dequeue(): remove the item with the highest priority
  - Find() return the item with the highest priority

- Way1:
  - Enqueue operation add item at the back of the queue;

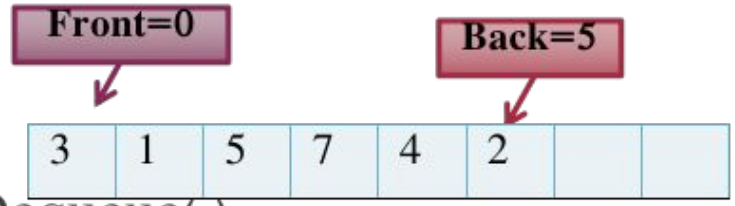
  - Dequeue operation removes item with highest priority.
    - If the deleted item is not the front most element of queue, then shift all elements to right side to fill the vacant position

Way 1

Enqueue(3) Enqueue(1)

Enqueue(5) Enqueue(7)

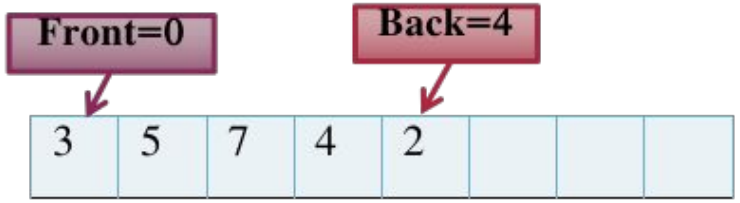
Enqueue(4) Enqueue(2)



Dequeue( )

Find highest priority 1

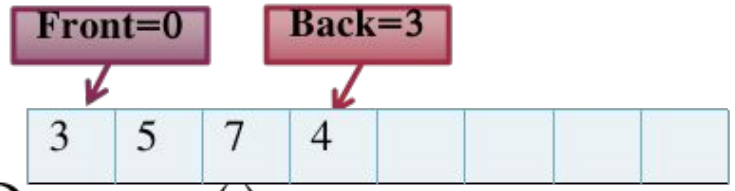
Remove it



Dequeue( )

Find highest priority 2

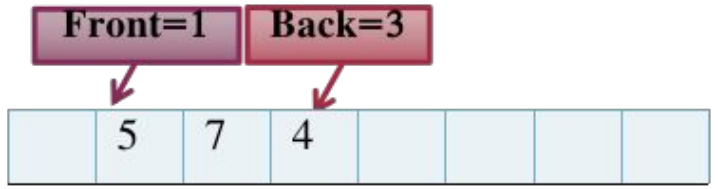
Remove it



Dequeue( )

Find highest priority 3

Remove it



Priority Queue ADT

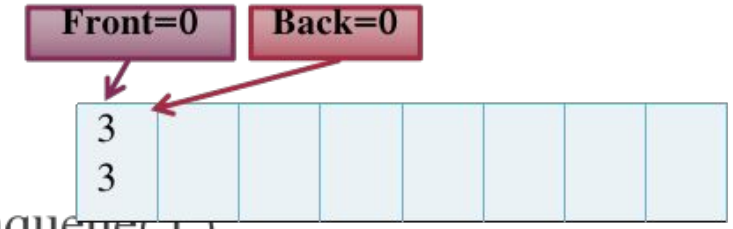
- A priority queue has following operations
  - Enqueue(item)
  - Dequeue(): remove the item with the highest priority
  - Find() return the item with the highest priority

- Way 2:
  - Dequeue operation removes an item from front of the queue;
  - Enqueue operation insert items according to their priorities.
    - A higher priority item is always enqueued before a lower priority element, so item with highest priority will always be inserted at front.
    - If the new element has the same priority as one or more elements already in the queue, then it is enqueued exactly after all those elements .

Way 2

Enqueue(3)

Queue is empty, so insert at front

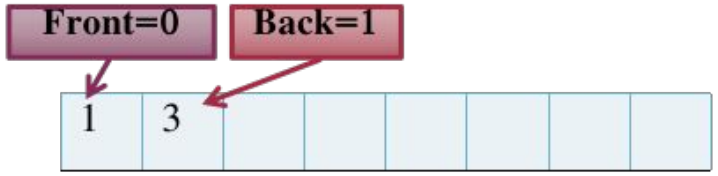


Enqueue(1)

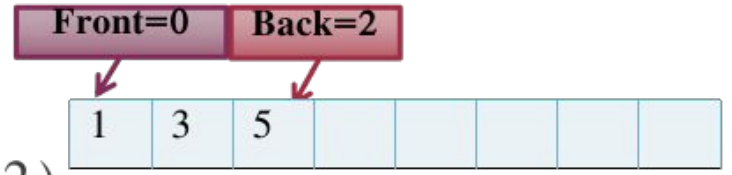
Queue is not Empty, So find appropriate location to insert according to priority

if is free, insert.

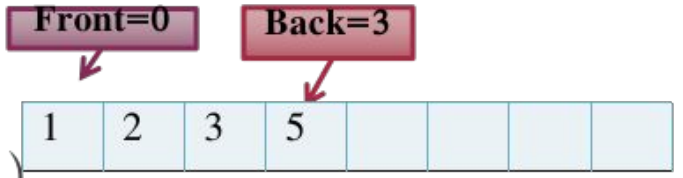
If it is not free, shift elements to right and insert



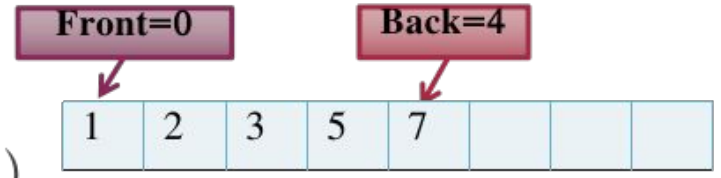
Enqueue(5)



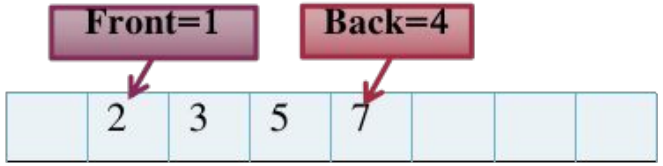
Enqueue(2)



Enqueue(7)

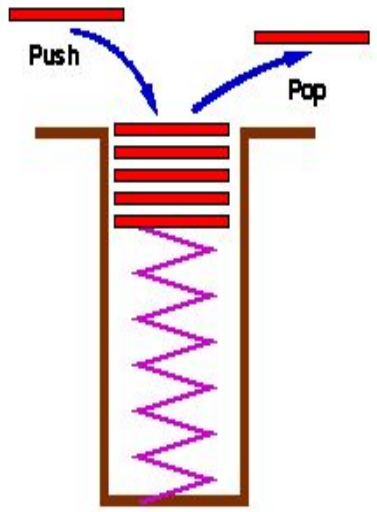


Dequeue( )



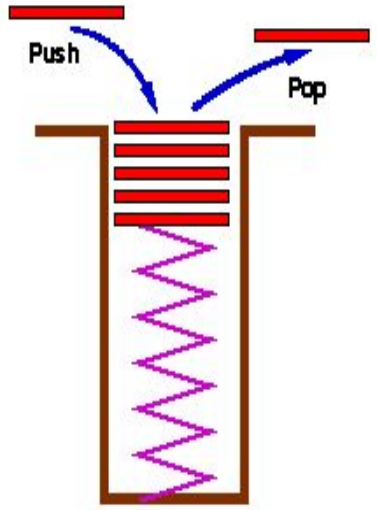
Stack ADT

Stack ADT



- A data structure to store data in which the elements are added and removed from one end only
  - a Last In First Out (LIFO) data structure.
- Also called push down list.
  - Think of a spring-loaded plate dispenser.
- At the logical level, a stack is an ordered group of homogeneous items or elements.
  - The removal of existing items and the addition of new items can take place only at the top of the stack.

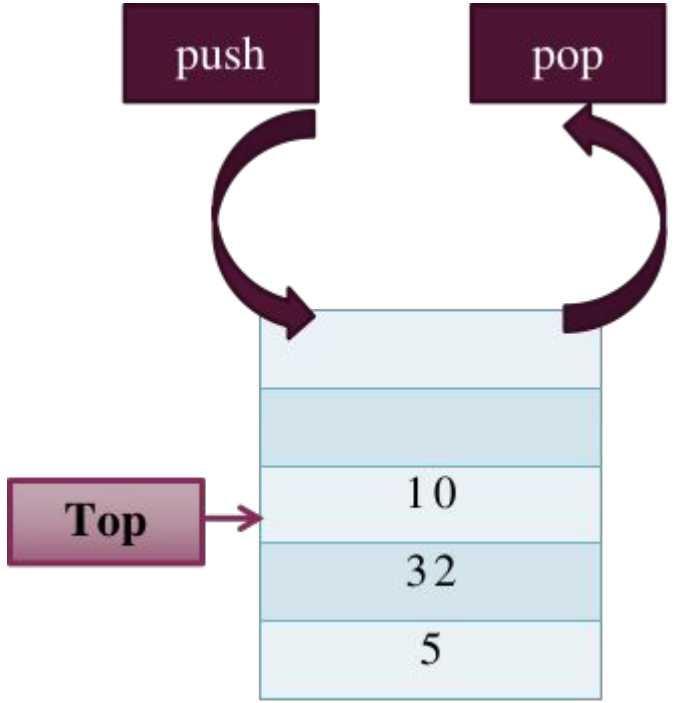
Stack ADT



- Stack is RESTRICTED type of List
  - So Insertion and Deletion can take place only at one end while other end of list is closed.
- Ordered
  - At any time, given any two elements in a stack, one is higher than the other and it is explicit linear ordering.
- Oldest is at the bottom and newest is at the top

Operations on Stacks

- Main stack operations:
  - **TOP():** Retrieve the element at the top of stack.
  - **POP():** Remove the top element of the stack.
  - **PUSH():** Insert the new element at the top of the stack.
  - **ISEMPTY():** Return true if it is an empty stack; return false otherwise.
  - **ISFULL():** Return true if it is full stack; return false otherwise.
  - **SIZE():** Return the total number of elements present in stack.

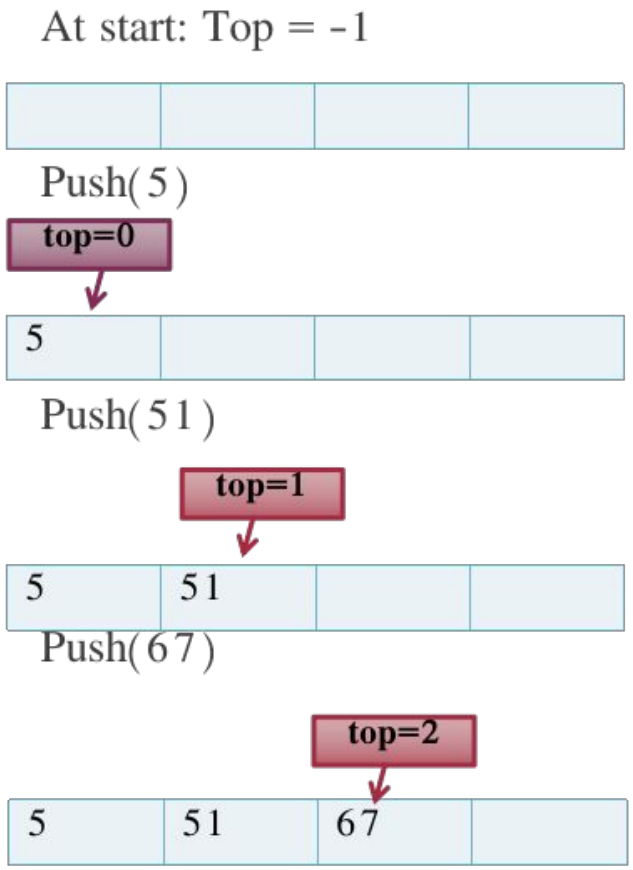


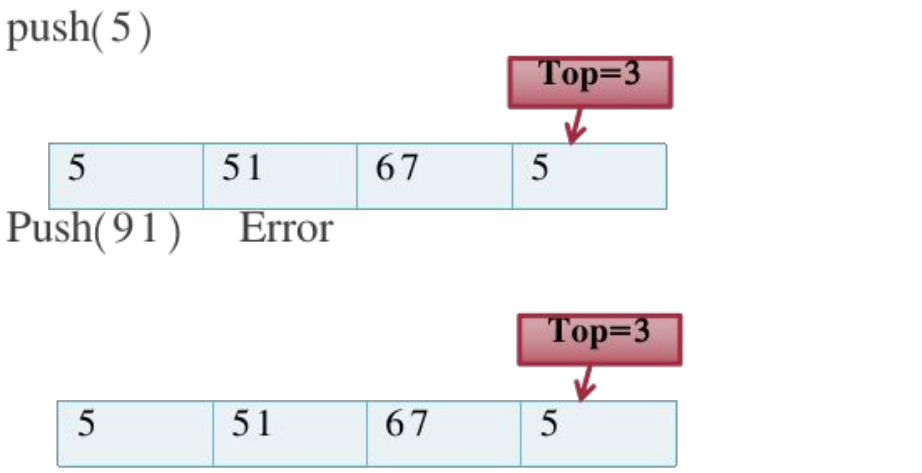
Applications

- Page-visited history in a Web browser
- Postponed Decision
- Parsing code
  - Matching parenthesis
- Tracking function calls
- Dealing with undo/redo operations
- Reverse-Polish calculators (Mathematical Expressions)

Array-based Stack

Push

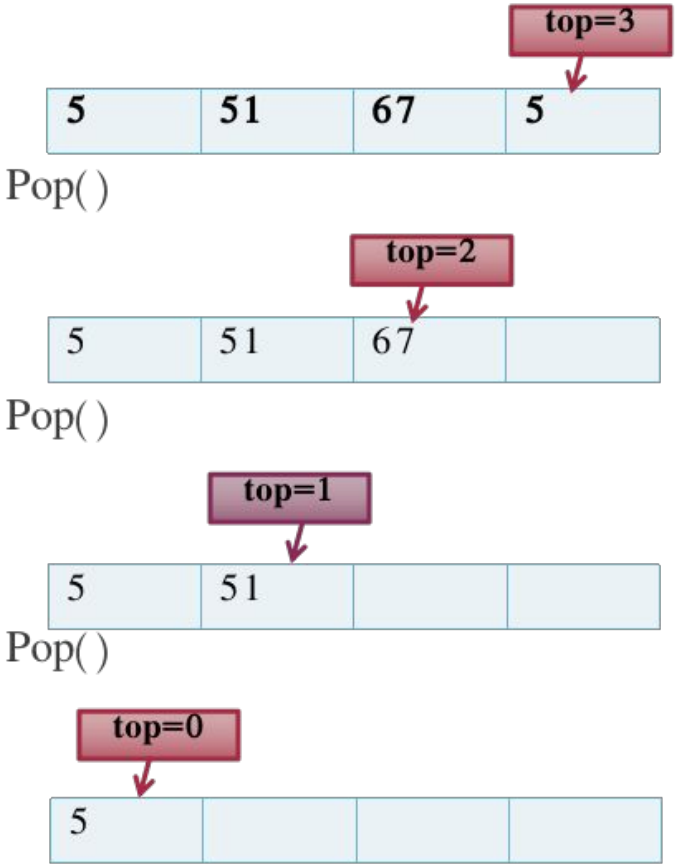




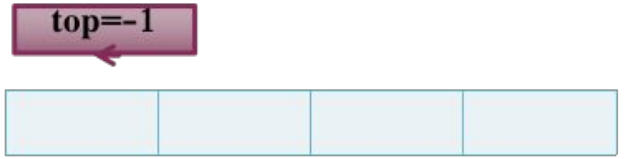
No more elements can be inserted if Stack is full.

How to know if Stack is full?

Pop



Pop( )



Top( ) Error

No more elements can be removed if Stack is empty.

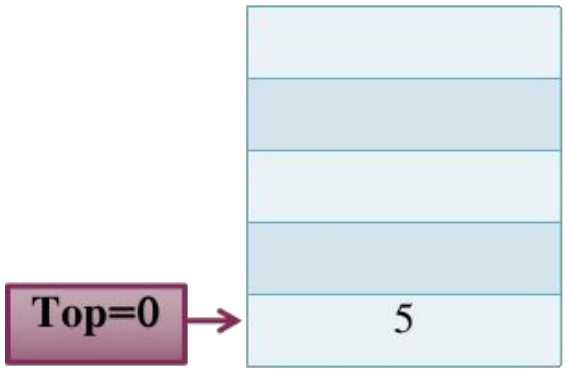
How to know if Stack is empty ?

How it works

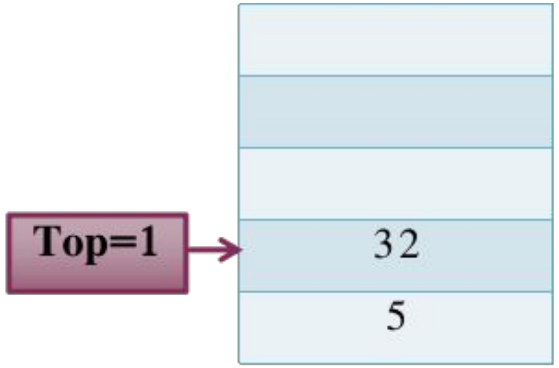
- At start top=-1
  - Push (5)
  - Push(32)
  - Push(-56),
  - Push(45),
  - Push(32)
  - Push(11)
  - Pop()
  - Pop()

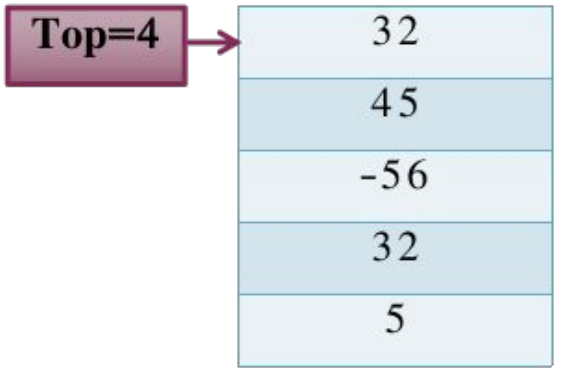
How it works

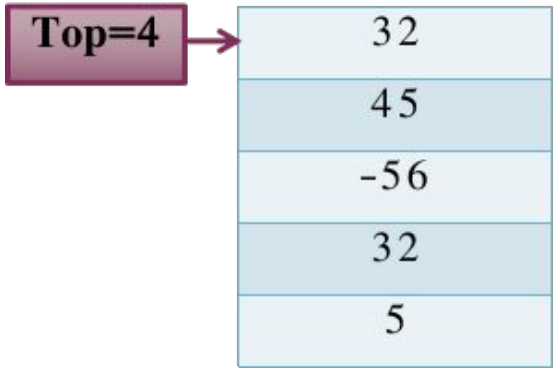
- At start top=-1
  - Push (5)
  - Push(32)
  - Push(-56),
  - Push(45),
  - Push(32)
  - Push(11)
  - Pop()
  - Pop()

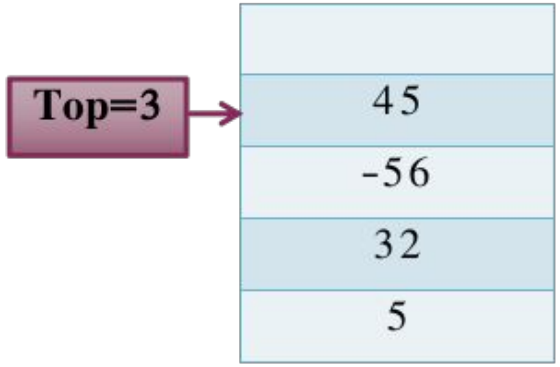


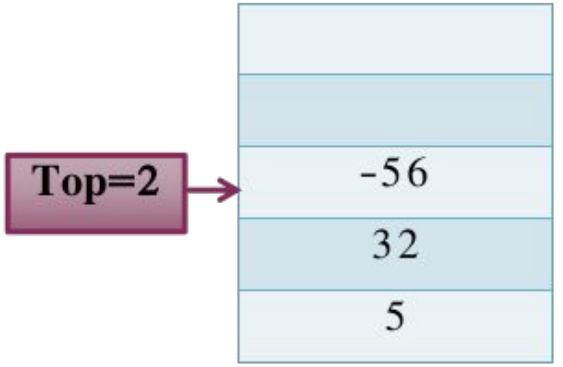
Error Stack is full





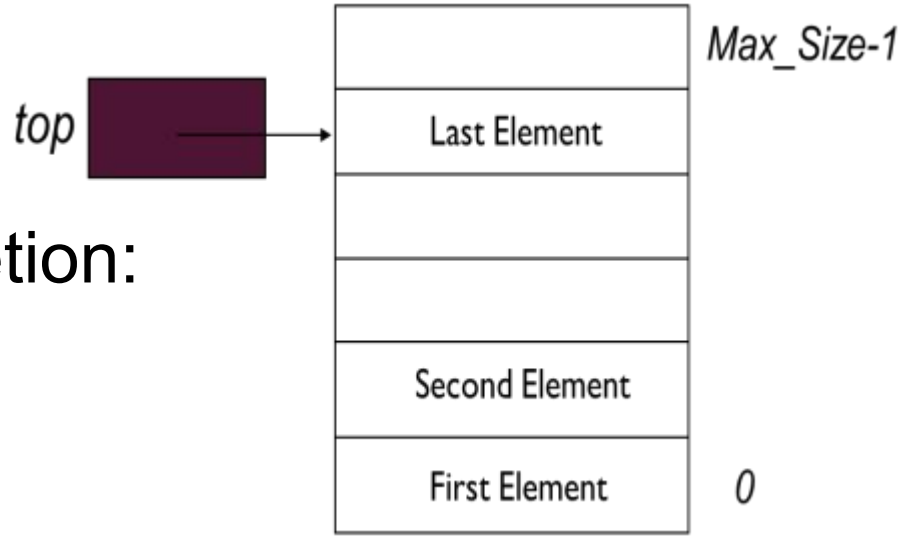






An Array Implementation of Stack

- Consider an array based LIST in vertical cell position.
- One end of list is closed and other is open
- We have 2 possibilities for Insertion and Deletion:
  - Either Pick insertion at end and deletion at end.
  - Or Pick insertion at start and deletion at start.
  - Insertion & Deletion At middle is not applicable.



An Array Implementation of Stack

- For maintaining open end, we have a reference point called TOP.
- Elements can be inserted/deleted to the top of the list.
- The following two conventions can be used for array based stack:
  - A stack can grow upwards: from index 0 to the maximum index.
  - Or it can grow downwards: from the maximum index to index 0.

Implementation and Performance

- A simple way to implement stack is to use fixed size array
  - Here Stack is simple array with top and MaxSize known
    - When top =-1 it means stack is empty
    - Stack Size = top+1
    - When top =MaxSize-1 means stack is full
- Performance
  - The space used is O(n)
  - Each operation runs in time **O(1)** If we take “At End” cases

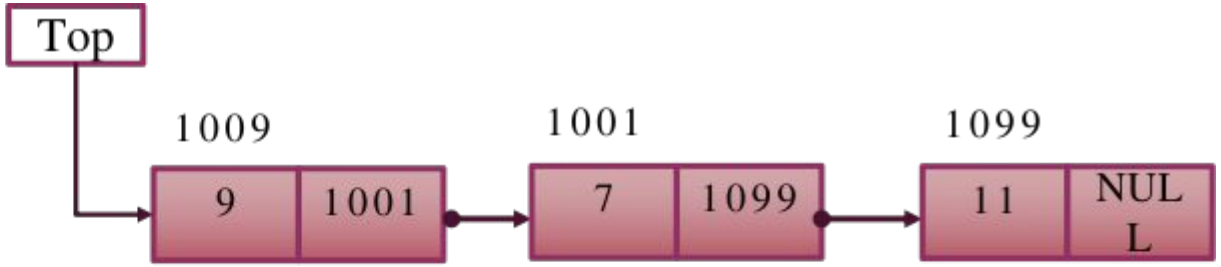
Implementation and Performance

- Limitations
  - The maximum size of the stack must be defined a priori , and cannot be changed
  - Trying to push a new element into a full stack causes an implementation-specific exception

Linked-based Stack

A Linked Implementation of Stack

- Consider a dynamic list which one end of list is closed and other is open, we have to possibilities for Insertion and Deletion:
  - Either Pick insertion at end and deletion at end.
  - Or Pick insertion at start and deletion at start.
  - Insertion & Deletion At middle is not applicable.
- For maintaining open end, we have a reference point called TOP.
- Elements can be inserted/deleted to the top of the list.



Here Stack is simple list with top known:
When top = NULL
it means stack is empty

Stack Size = Total Number of nodes

Applications

Arithmetic Expression Evaluation

A + B * C / D – E ^ F * G

- Arithmetic expression contains:
  - Operands
  - Operators: Every operator has a **precedence** and **associativity**

Operators	Precedence	Associativity
--, ++, NOT (!)	6	Left To Right
^	6	Right to Left
***, /**	5	Left To Right
+, _	4	Left To Right
<, <=, >, >=	3	Left To Right
AND	2	Left To Right
OR, XOR	1	Left To Right

Mathematical Expressions

- **Infix Notation** (Standard Notation)
  - Operator is between operands: A+B
  - Example: (((1 + 2) * 3) + 6) / (2 + 3)
  - Add 1 and 2, multiply with 3, add 6, add 2 and 3, divide
- **Prefix** Notation (Polish Notation)
  - Operator is before operands: +AB
  - Example: (/ (+ (* (+ 1 2) 3) 6) (+ 2 3))
  - Add 1 and 2, multiply with 3, add 6, add 2 and 3, divide

Mathematical Expressions

- **Postfix Notation** (Reverse Polish Notation)
  - Operator is after operands: AB+
  - Example: 1 2 + 3 * 6 + 2 3 + /
  - This means "take 1 and 2, add them, take 3 and multiply, take 6 and add, take 2 and 3, add them, and divide".
  - The order of evaluation of operators is always left-to-right, and brackets cannot be used to change this order.
  - Because the "+" is to the left of the "*" in the example above, the addition must be performed before the multiplication.

Example:Mathematical Expressions

- Infix Expression **A * B + C / D**
- Scan from left to right and choose higher precedence operator first.
  - Show transition with brackets
    - **[AB*] + C /D**
    - **[AB*] + [CD/]**
  - Let's assume X = AB* and Y = CD/, so our new expression will be XY+
  - **XY+ = AB* CD/ +**

- When transition is finished, no brackets are used

Example:Mathematical Expressions

- Infix Expression **A * B + C / D**
- Scan from left to right and choose higher precedence operator first.
  - Show transition with brackets
    - **[AB*] + C /D**
    - **[AB*] + [CD/]**
  - Let's assume X = AB* and Y = CD/, so our new expression will be XY+
  - **XY+ = AB* CD/ +**

- When transition is finished, no brackets are used

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

Example:Mathematical Expressions

Infix	Postfix
a+b	ab+
a+b*c	abc*+
a*b+c	ab*c+
(a+b)*c	ab+c*
(a-b)*(c+d)	ab-cd+*
(a+b)*(c-d/e)+f	ab+cde/-*f+

Example:Mathematical Expressions

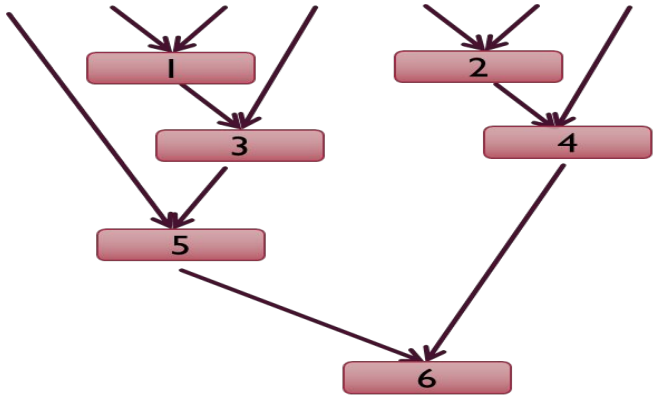
Infix	Postfix
a+b	ab+
a+b*c	abc*+
a*b+c	ab*c+
(a+b)*c	ab+c*
(a-b)*(c+d)	ab-cd+*
(a+b)*(c-d/e)+f	ab+cde/-*f+

Infix	Postfix
a+b*c	abc*+
a*b+c*d	ab*cd*+
(a+b)*(c+d)/e-f	ab+cd+*e/f-
a/b-c+d*e-a*c	ab/c-de*+ac*-
a+b/c*(e+g)+h-f*i	abc/eg+*+h+fi*-

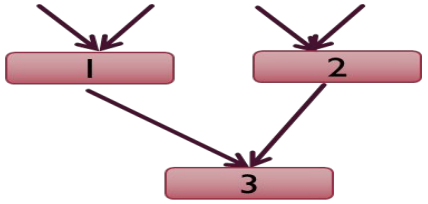
Infix vs. Post Evaluation

Order of evaluation according to precedence and associativity

A + B * C / D – E ^ F * G



A * B + C / D



- In postfix expression order of evaluation is strictly from left to right.
- No parentheses are used to change this order.
- Because operators come after operands, that's why **operator is applied to two immediate left operands**.

Postfix Evaluation

A B * C D / +

1. Read A
2. Read B
3. Read * A*B =X1
4. Read C
5. Read D
6. Read / C/D= X2
7. Read + X1+X2
8. Print Result

2 5 8 3 10 * + / -

1. Read 2
2. Read 5
3. Read 8
4. Read 3
5. Read 10
6. Read * 10/3= 3
7. Read + 3+8= 11
8. Read / 11/5= 2
9. Read - 2-2 = 0
10. Print 0

3 4 5 * 6 / +

1. Read 3
2. Read 4
3. Read 5
4. Read * 5*4= 20
5. Read 6
6. Read / 20/6= 3
7. Read + 3+3 = 6
8. Print 6

Evaluation of Mathematical Expressions

- The normal way of writing expressions is by placing a **binary operator** in-between its two operands, is called the **infix notation**.

- It is not easy to evaluate arithmetic and logic expressions written in infix notation since they must be evaluated according to **operator precedence** rules.
  - E.g., a+b*c must be evaluated as (a+(b*c)) and not ((a+b)*c).

Evaluation of Mathematical Expressions

- The **postfix or Reverse Polish Notation (RPN)** is used by the compilers for expression evaluation.

- In RPN, each **operator appears after the operands** on which it is applied.
  - This is a parenthesis-free notation.

- **Stacks** can be used to convert an expression from its infix form to RPN and then evaluate the expression.

Evaluation of Expressions

- Evaluation of expression like **a+b/c*(e-g)+h-f*l** was a challenging task for compiler writers.

- A fully parenthesized expression can be evaluated with the help of a stack.

- Conversion to Prefix Expression
  - The precedence rules for converting an expression from **infix to prefix** are identical.
  - The only change from postfix is that the operator is placed before the operands rather than after them.
  - However, the order of operands or operator is not affected.

Evaluation of Expressions

- Evaluation of expression like **a+b/c*(e-g)+h-f*l** was a challenging task for compiler writers.

- A fully parenthesized expression can be evaluated with the help of a stack.

- Evaluating a Postfix Expression
  - Each operator in a postfix string refers to the previous two operands in the string.

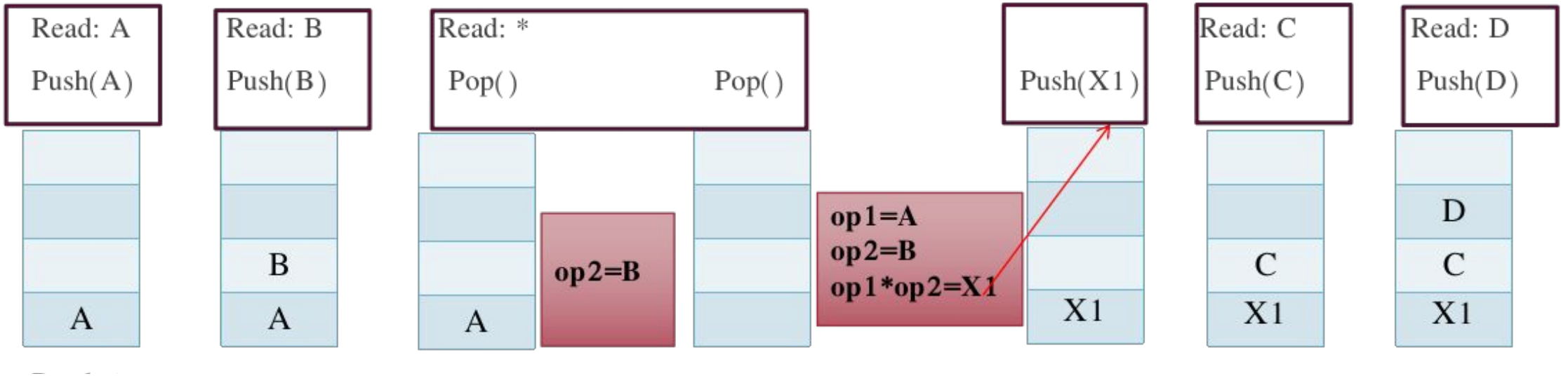
Postfix Evaluation

- We can evaluate a postfix expression using a stack.
  - Each operator in a postfix expression corresponds to the previous two operands
  - Each time we read an operand we push it onto a stack.
  - Each time we read an operator, its associated operands (the top two elements on the stack) are popped out.
  - Operation is performed on those operands and result is pushed onto the stack
  - Example

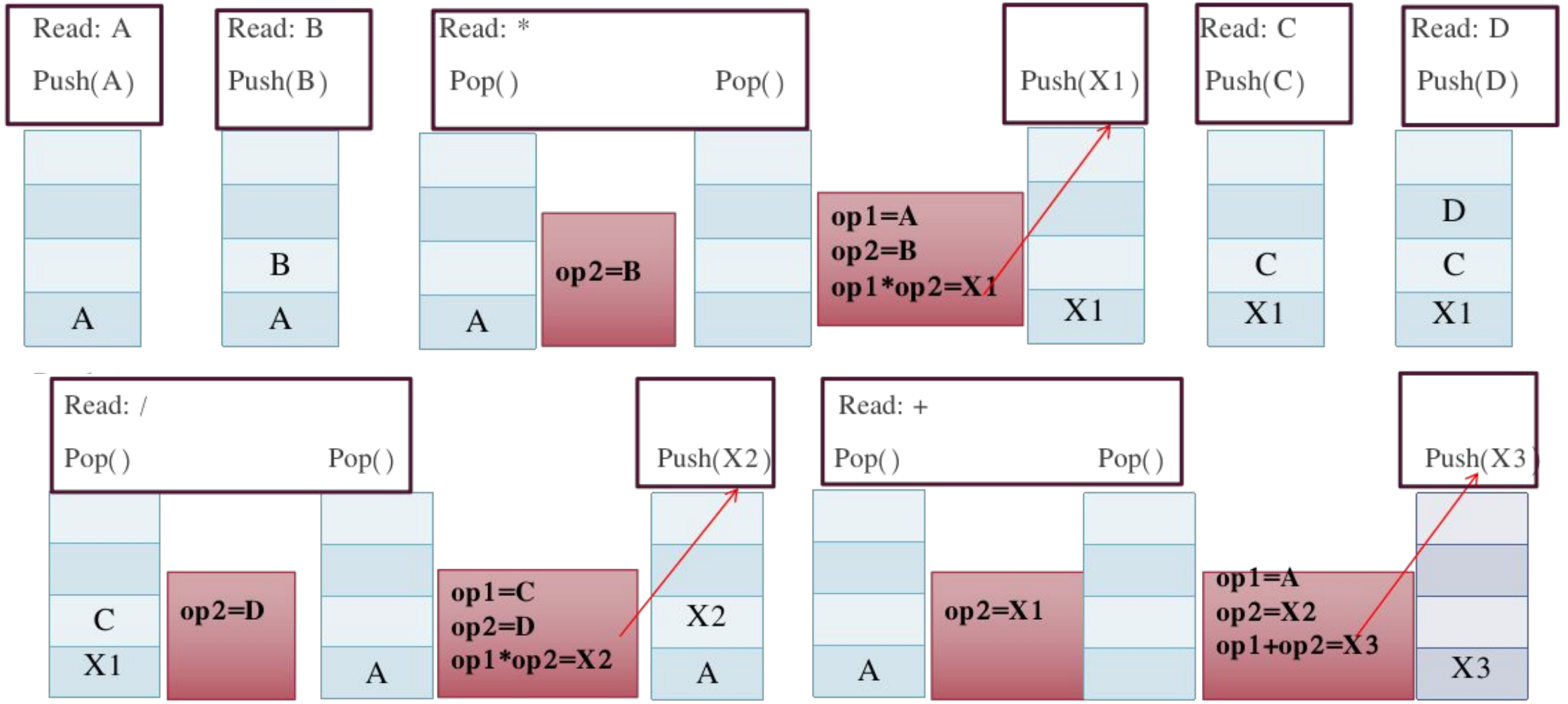
    A B * C D / +

    6 5 2 3 + 8 * + 3 + *

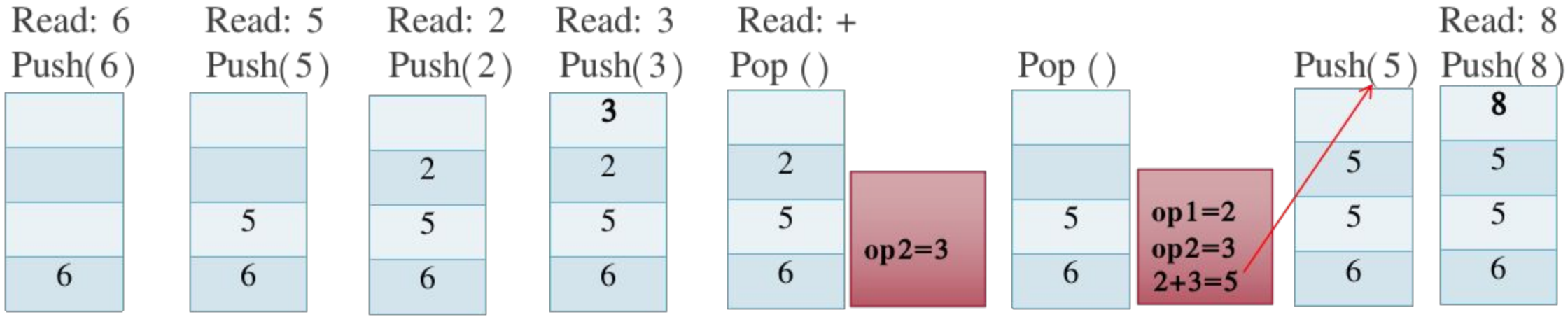
Postfix Evaluation: A B * C D / +



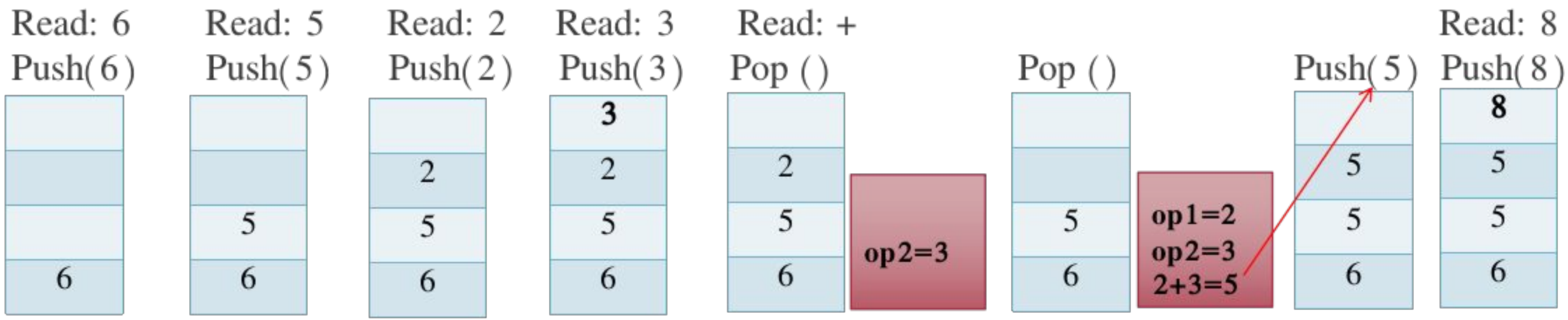
Postfix Evaluation: A B * C D / +



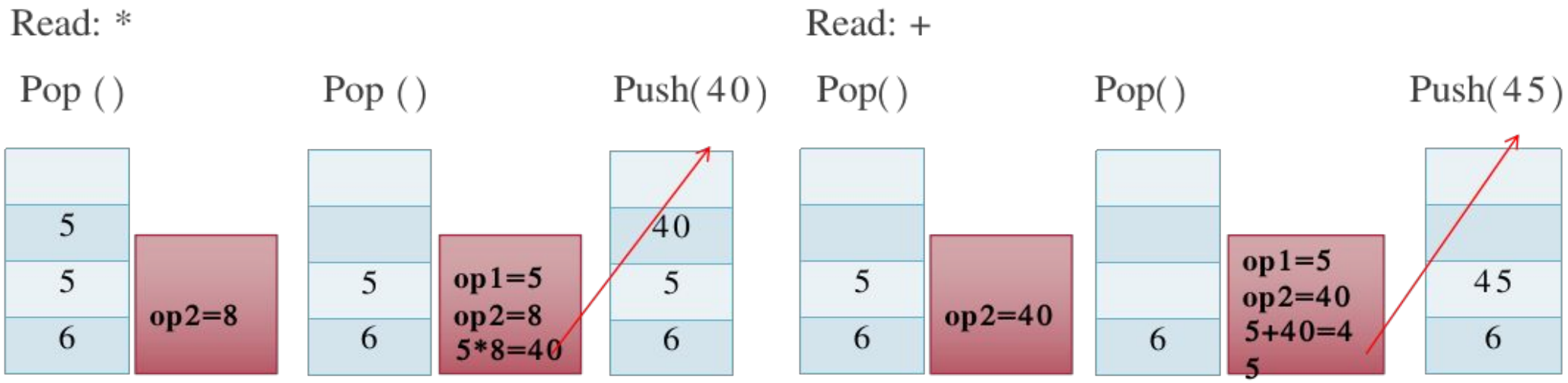
Postfix Evaluation: 6 5 2 3 + 8 * + 3 + *



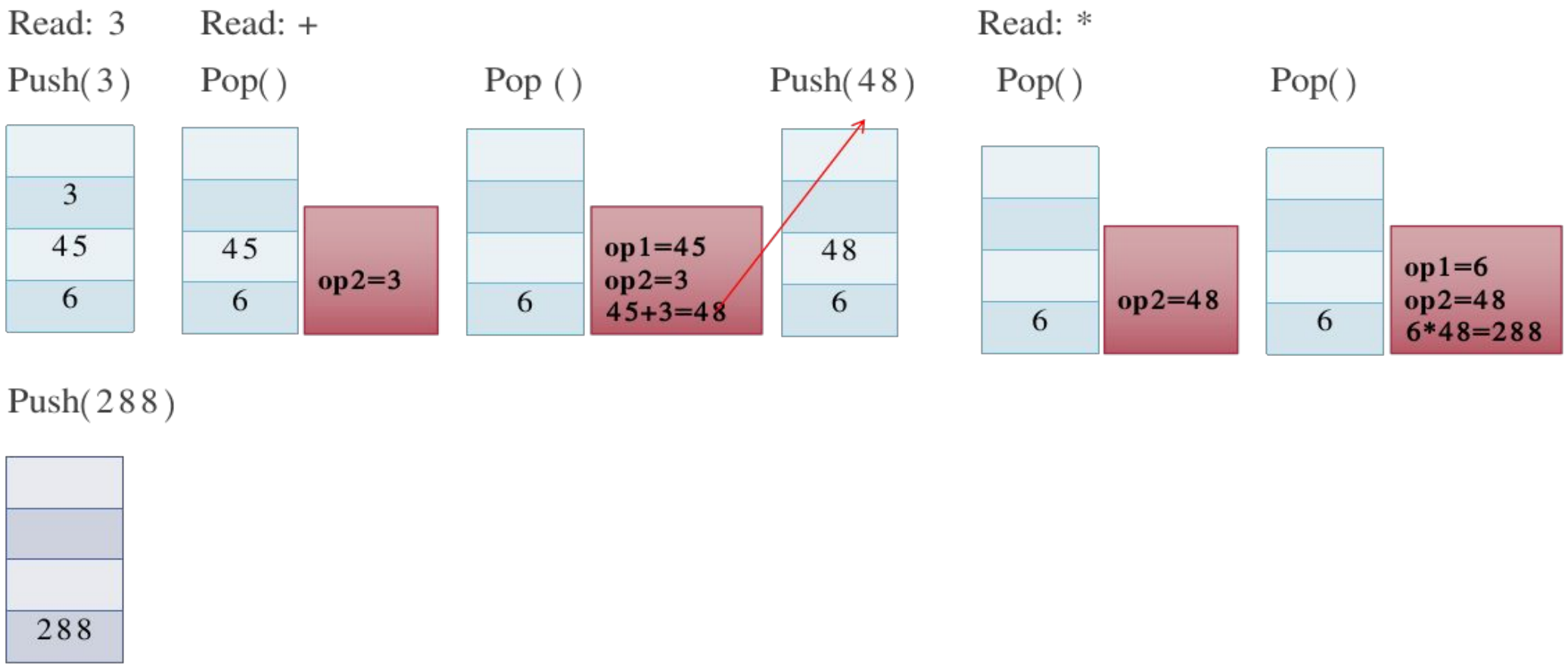
Postfix Evaluation: 6 5 2 3 + 8 * + 3 + *



Postfix Evaluation: 6 5 2 3 + 8 * + 3 + *



Postfix Evaluation: 6 5 2 3 + 8 * + 3 + *



Postfix Evaluation(P)

This algorithm finds the VALUE of P written in postfix notation.
1. Add a Dollar Sign “$” at the end of P. [This acts as sentinel.]
2. Scan P from left to right and repeat Steps 3 and 4 for each element of P until the sentinel “$” is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator © is encountered, then:
    a. Remove the two top elements of STACK, where A is the top element and B is the next-to-top-element.
    b. Evaluate B © A.
    c. Place the result of (b) back on STACK.
    [End of If structure.]
[End of Step 2 loop.]
5. Set VALUE equal to the top element on STACK.
6. Exit.

Infix to Postfix Conversion

Infix Expression: a + b * c

Precedence of * is higher than +, So convert the multiplication

a + [b c *]

Convert the addition a b c * +

Postfix Expression: a b c * +

Infix Expression: (a +b) * c

Convert addition

[a b +]* c

Convert the multiplication a b + c *

Post Expression: a b + c*

Infix Expression: a + ((b * c) / d)

Convert multiplication a + ([b c *] / d)

Convert division a + [b c * d/]

Convert the addition a b c * d/

Postfix Expression: a b c * d/+

Keep in Mind:

1. Relative order of variables is not changed
2. No parenthesis in postfix/prefix
3. Operators are arranged according to precedence

Infix_to_Postfix (Q, P)

Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push “(“ onto STACK, and add “)” to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator © is encountered, then:
    a. Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same or higher precedence/priority than ©
    b. Add © to STACK. [End of If structure.]
6. If a right parenthesis is encountered, then:
    a. Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
    b. Remove the left parenthesis. [Do not add the left parenthesis to P.] [End of If structure.]

[End of Step 2 loop.]

7. Exit.

Infix_to_Postfix (Q, P)

Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push "(" onto STACK, and add ")" to the end of Q.
2. Scan Q from left to right and repeat Steps 3 to 6 for each element of Q until the STACK is empty:
3. If an operand is encountered, add it to P.
4. If a left parenthesis is encountered, push it onto STACK.
5. If an operator © is encountered, then:
   a. Repeatedly pop from STACK and add to P each operator (on the top of STACK) which has the same or higher precedence/priority than ©
   b. Add © to STACK. [End of If structure.]
6. If a right parenthesis is encountered, then:
   a. Repeatedly pop from STACK and add to P each operator (on the top of STACK) until a left parenthesis is encountered.
   b. Remove the left parenthesis. [Do not add the left parenthesis to P.] [End of If structure.]

[End of Step 2 loop.]

7. Exit.

Keep in Mind:

1. Relative order of variables is not changed
2. No parenthesis in postfix
3. Operators are arranged according to precedence

Infix to Postfix Conversion

3+4*5/6

Output:

3 4 5 * 6 / +

(((1 + 2) * 3) + 6) / (2 + 3)

Output:

1 2 + 3 * + 2 3 + /

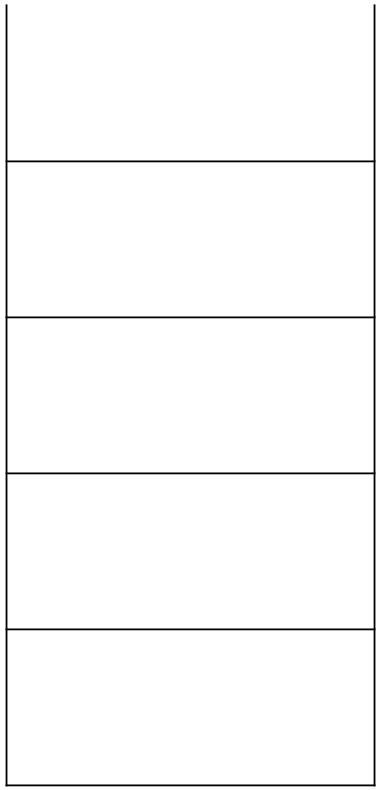
Evaluate Expressions in RPN

(a+b)*(c+d) ab+cd+* Assuming a=2, b=6, c=3, d=-1

Input Symbol	Stack	Remarks
a	a	Push
b	a b	Push
+	8	Pop a and b from the stack, add, and push the result back
c	8 c	Push
d	8 c d	Push
+	8 2	Pop c and d from the stack, add, and push the result back
*	16	Pop 8 and 2 from the stack, multiply, and push the result back. Since this is end of the expression, hence it is the final result.

Postfix Expression Calculator

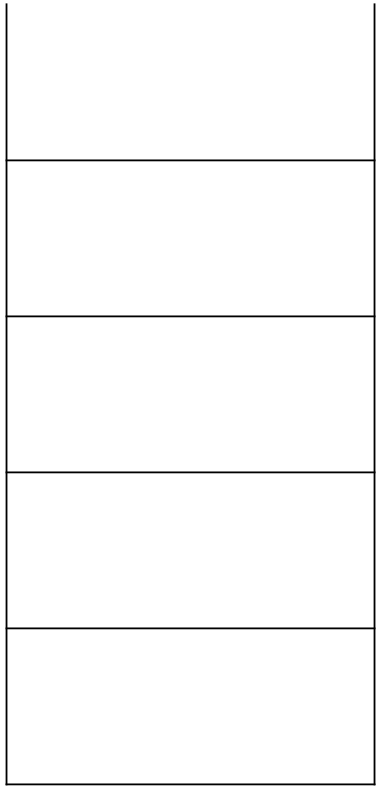
Expression: 6 3 + 2 * =



Postfix Expression Calculator

Expression: 6 3 + 2 * =

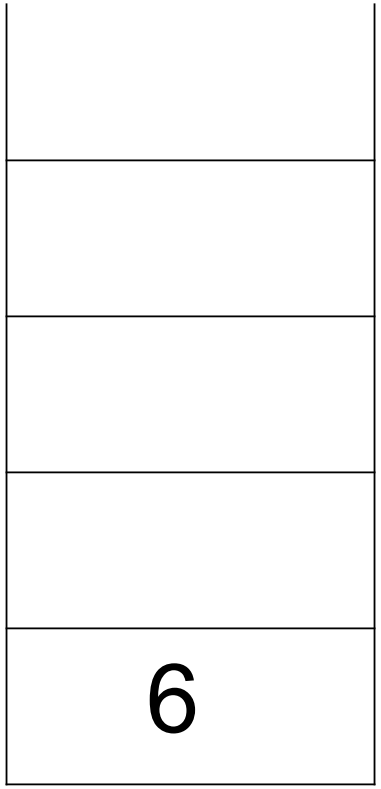
1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display



Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

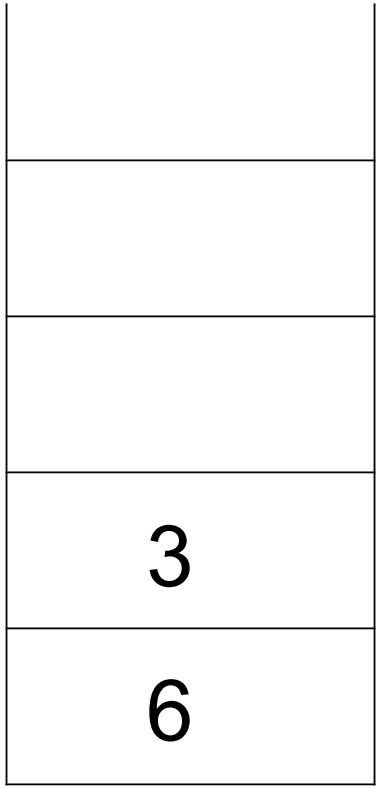


1. Push 6

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

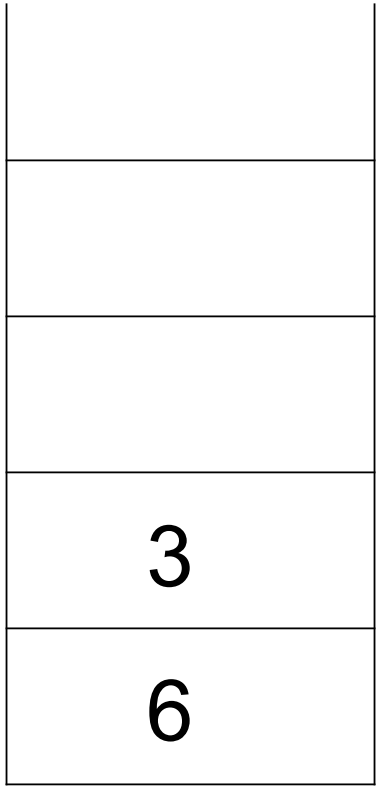


1. Push 6
2. Push 3

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

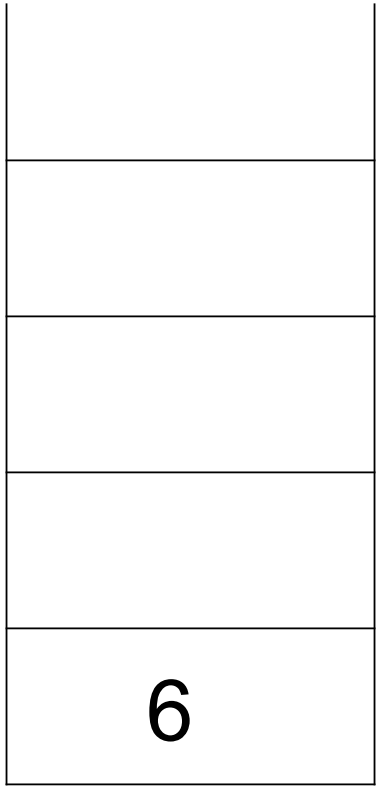


1. Push 6
2. Push 3
3. Symbol is + so pop 2 times

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display



1. Push 6
2. Push 3
3. Symbol is + so pop 2 times

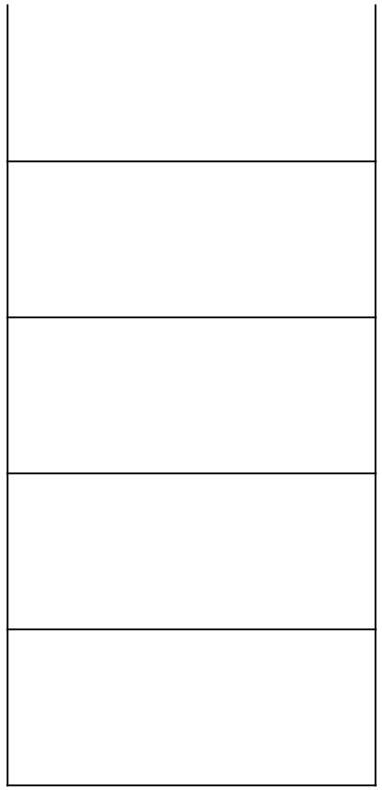
+ 3

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

6 + 3



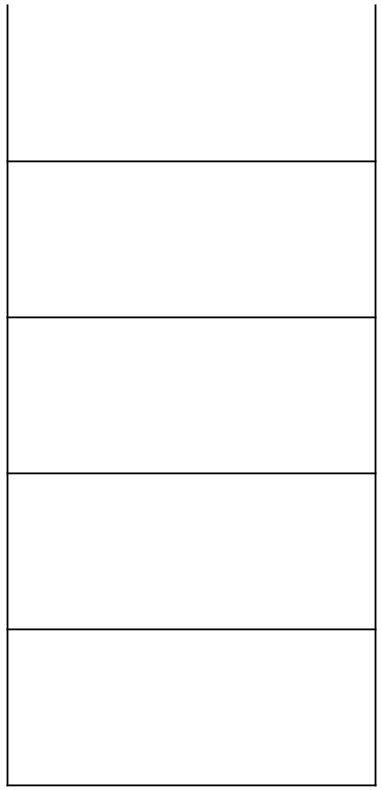
1. Push 6
2. Push 3
3. Symbol is + so pop 2 times

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

6 + 3 = 9

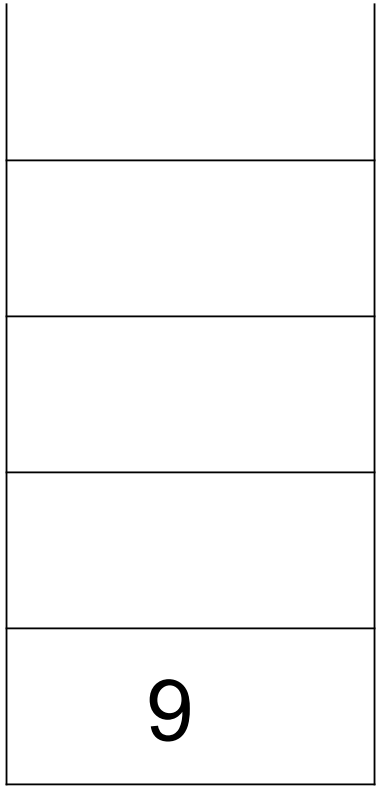


1. Push 6
2. Push 3
3. Symbol is + so pop 2 times

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

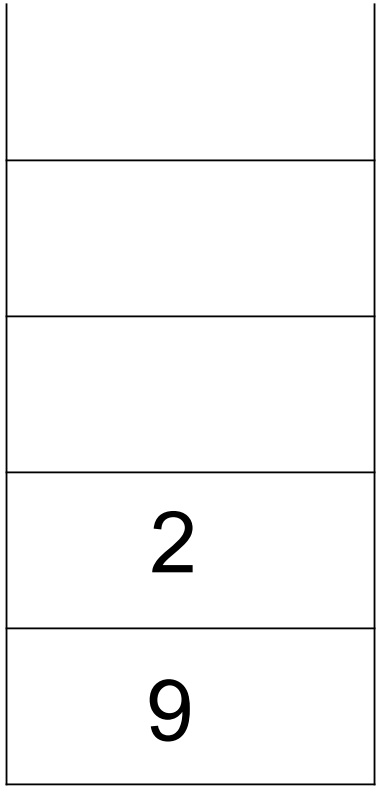


1. Push 6
2. Push 3
3. Symbol is + so pop 2 times
4. Push result = 9

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

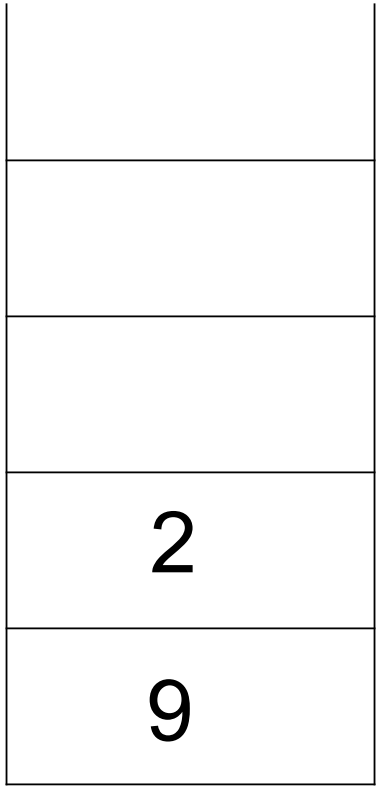


1. Push 6
2. Push 3
3. Symbol is + so pop 2 times
4. Push result = 9
5. Push 2

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

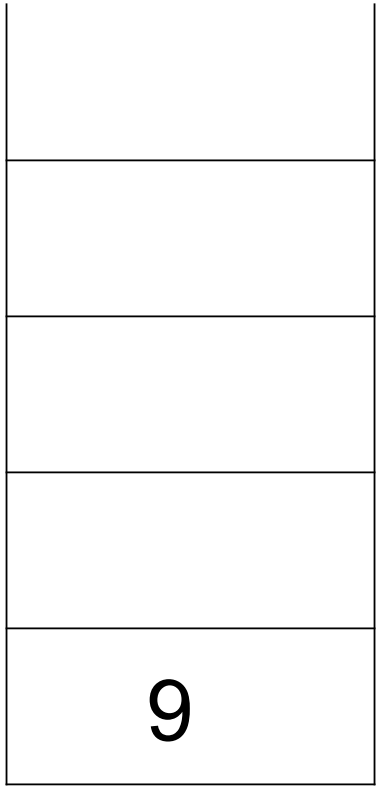


1. Push 6
2. Push 3
3. Symbol is + so pop 2 times
4. Push result = 9
5. Push 2
6. Symbol is * so pop 2 times

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display



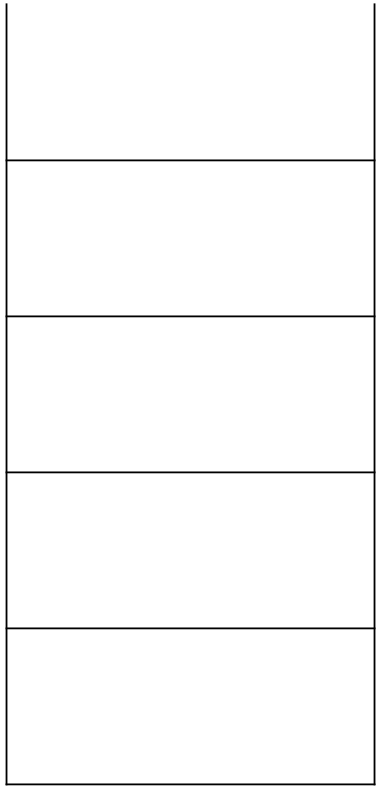
1. Push 6
2. Push 3
3. Symbol is + so pop 2 times
4. Push result = 9
5. Push 2
6. Symbol is * so pop 2 times

*** 2**

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display



1. Push 6
2. Push 3
3. Symbol is + so pop 2 times
4. Push result = 9
5. Push 2
6. Symbol is * so pop 2 times

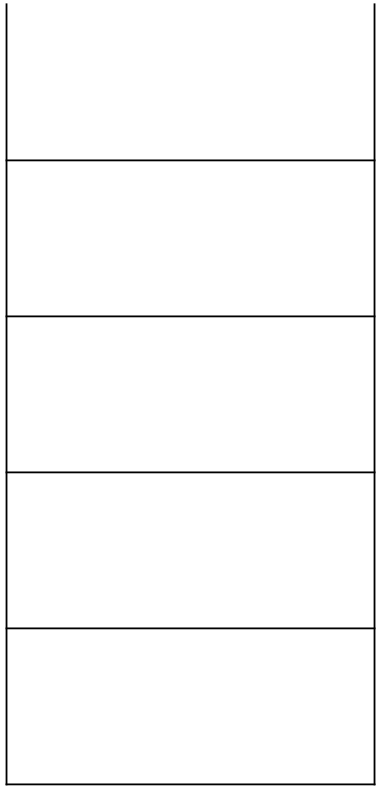
9 * 2

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

9 * 2 = 18

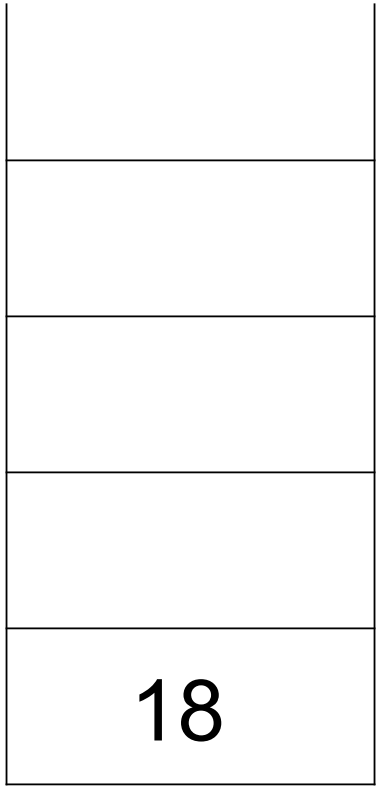


1. Push 6
2. Push 3
3. Symbol is + so pop 2 times
4. Push result = 9
5. Push 2
6. Symbol is * so pop 2 times
7. Push result = 18

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

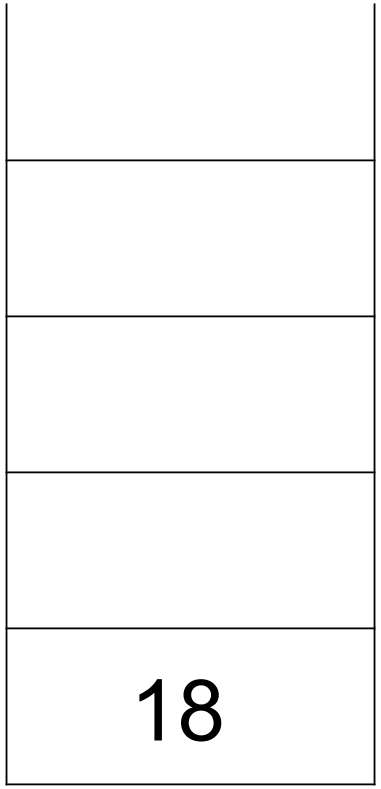


1. Push 6
2. Push 3
3. Symbol is + so pop 2 times
4. Push result = 9
5. Push 2
6. Symbol is * so pop 2 times
7. Push result = 18

Postfix Expression Calculator

Expression: 6 3 + 2 * =

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display

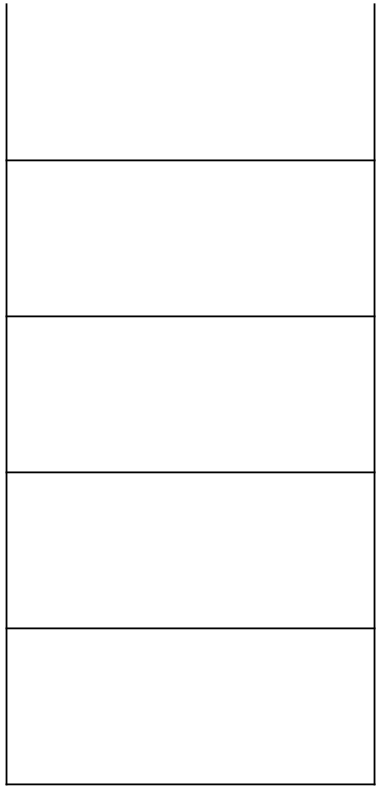


1. Push 6
2. Push 3
3. Symbol is + so pop 2 times
4. Push result = 9
5. Push 2
6. Symbol is * so pop 2 times
7. Push result = 18
8. Symbol is = so pop & display

Postfix Expression Calculator

Expression: 6 3 + 2 * = 18

1. If symbol is operand then push it in the stack
2. Else if symbol is operator then pop 2 operands and perform action & push the result in the stack again
3. else if symbol is = then the expression ends. Pop the result & display



1. Push 6
2. Push 3
3. Symbol is + so pop 2 times
4. Push result = 9
5. Push 2
6. Symbol is * so pop 2 times
7. Push result = 18
8. Symbol is = so pop & display

Implementation of Subprograms

- In a typical program, the subprogram calls are **nested.**
- Some of these subprogram calls may be **recursive**.
- Address of the next instruction in the calling program must be saved in order to resume the execution from the point of subprogram call.
- Since the subprogram calls are nested to an arbitrary depth, use of stack is a natural choice to preserve the return address.

Activation Record/ Stack Frame

- An activation record is a data structure which keeps important information about a sub program.
  - In modern languages, whenever a subprogram is called, a **new activation record** corresponding to the subprogram call is created, and pushed into the stack.

- The information stored in an activation record includes
  - the address of the next instruction to be executed, and
  - current value of all the local variables and parameters. i.e.
    - the context of a subprogram is stored in the activation record.

Activation Record/ Stack Frame

- When the subprogram finishes its execution and returns back to the calling function,
  - its activation record is popped from the stack and destroyed-restoring the context of the calling function.

- The state of each function is characterized by an activation record.
  - It is the private pool of information for a function.
  - It has the following information:
    - Parameters and local variables: of the function
    - Return Address: Caller function's return address

Runtime Stack

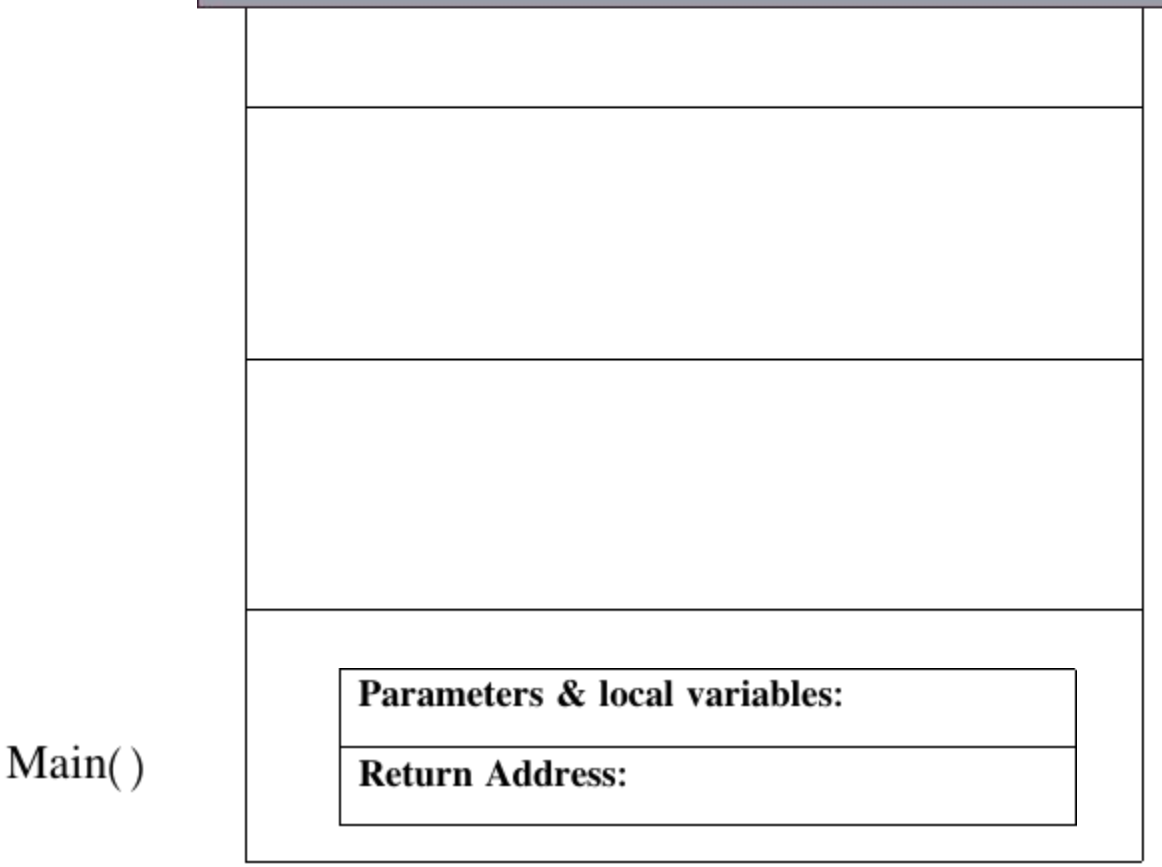
```
int main()
{    int x,y;
     statement1;
     A();
     statement2;
     statement3;
     B();
     statement4;
}
void A()
{    C();
     statement 5;
}
```

Runtime stack/System Stack is a stack whose each element is the activation records of the different functions being called during the running of the program

Runtime Stack

```
int main()
{    int x,y;
     statement1;
     A();
     statement2;
     statement3;
     B();
     statement4;
}
void A()
{    C();
     statement 5;
}
```

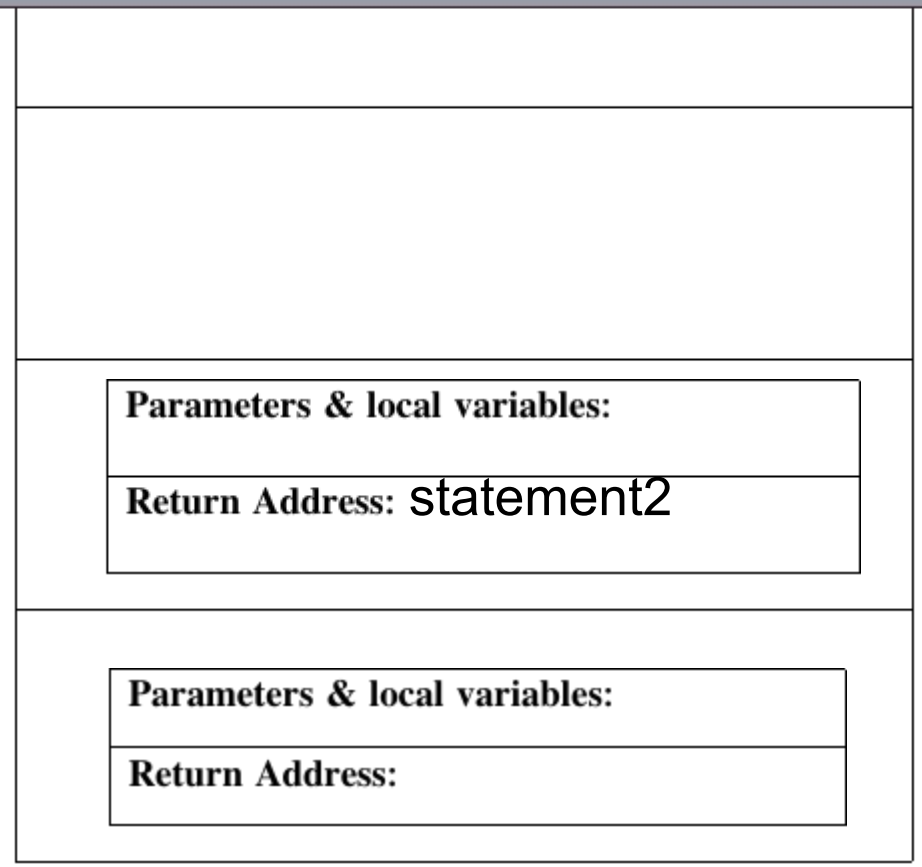
Runtime stack/System Stack is a stack whose each element is the activation records of the different functions being called during the running of the program



Runtime Stack

```
int main()
{   int x,y;
    statement1;
    A();
    statement2;
    statement3;
    B();
    statement4;
}
void A()
{   C();
    statement 5;
}
```

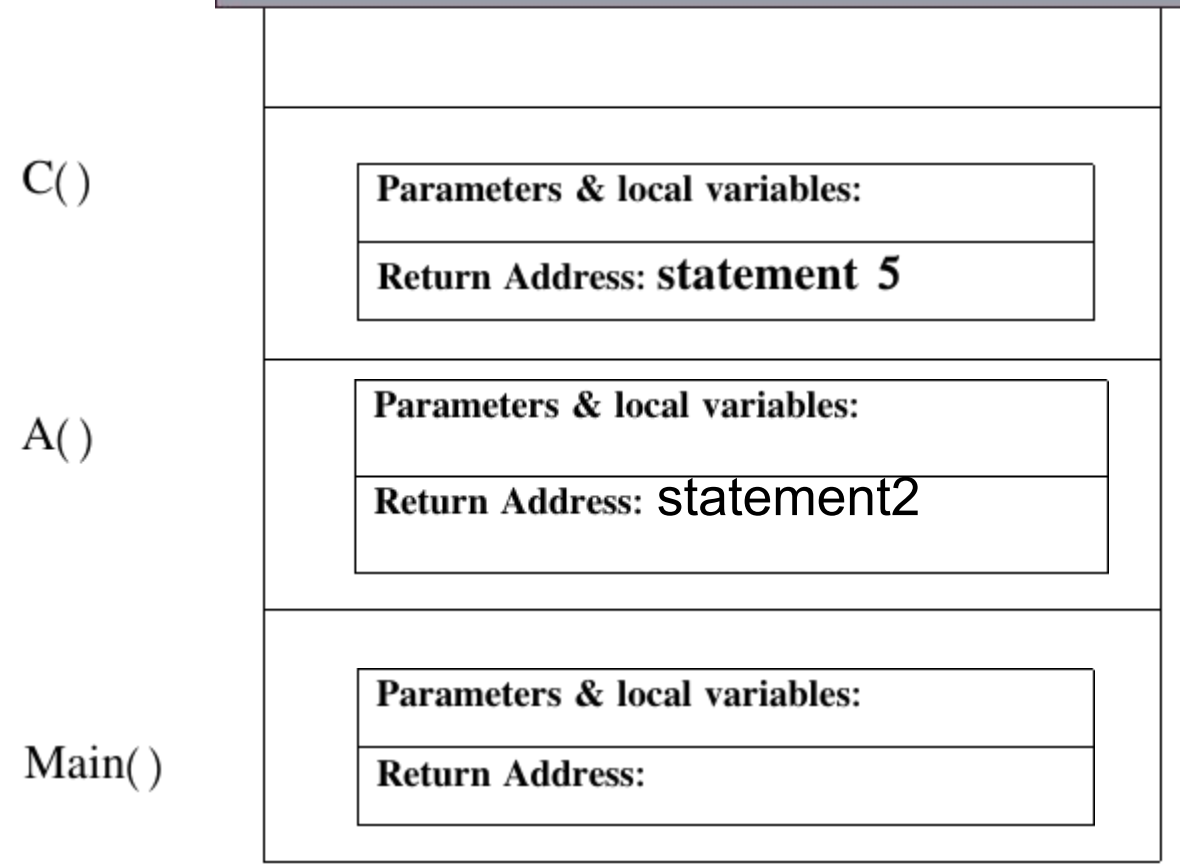
Runtime stack/System Stack is a stack whose each element is the activation records of the different functions being called during the running of the program



Runtime Stack

```
int main()
{    int x,y;
     statement1;
     A();
     statement2;
     statement3;
     B();
     statement4;
}
void A()
{    C();
     statement 5;
}
```

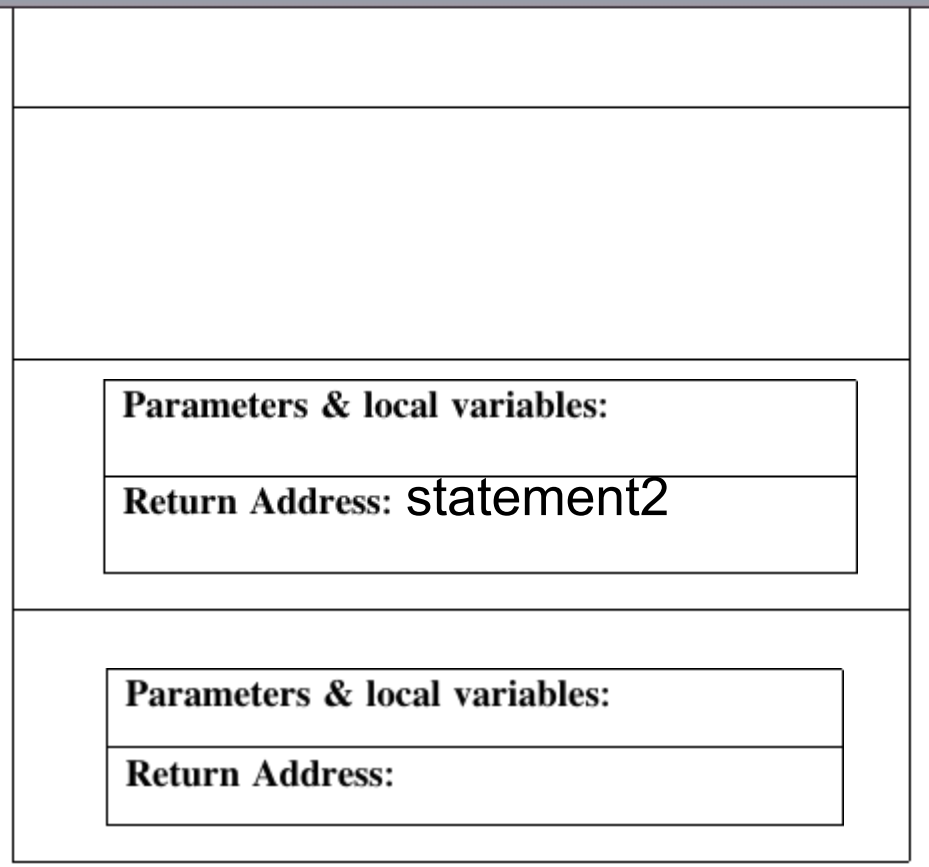
Runtime stack/System Stack is a stack whose each element is the activation records of the different functions being called during the running of the program



Runtime Stack

```
int main()
{    int x,y;
     statement1;
     A();
     statement2;
     statement3;
     B();
     statement4;
}
void A()
{    C();
     statement 5;
}
```

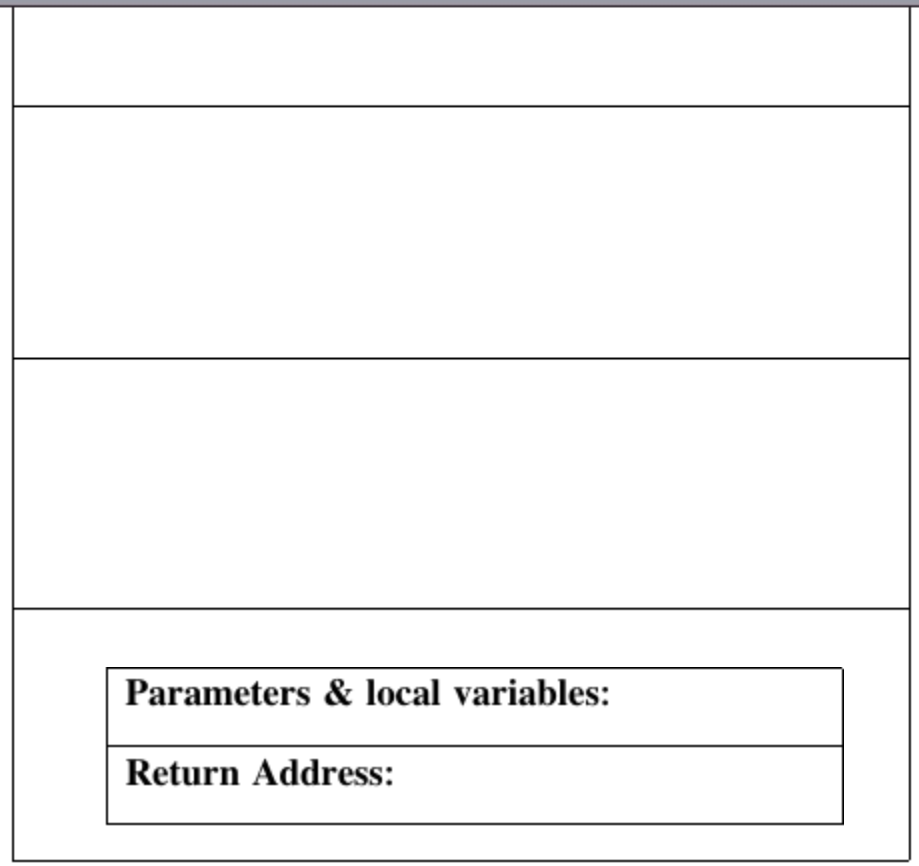
Runtime stack/System Stack is a stack whose each element is the activation records of the different functions being called during the running of the program



Runtime Stack

```
int main()
{   int x,y;
    statement1;
    A();
    statement2;
    statement3;
    B();
    statement4;
}
void A()
{   C();
    statement 5;
}
```

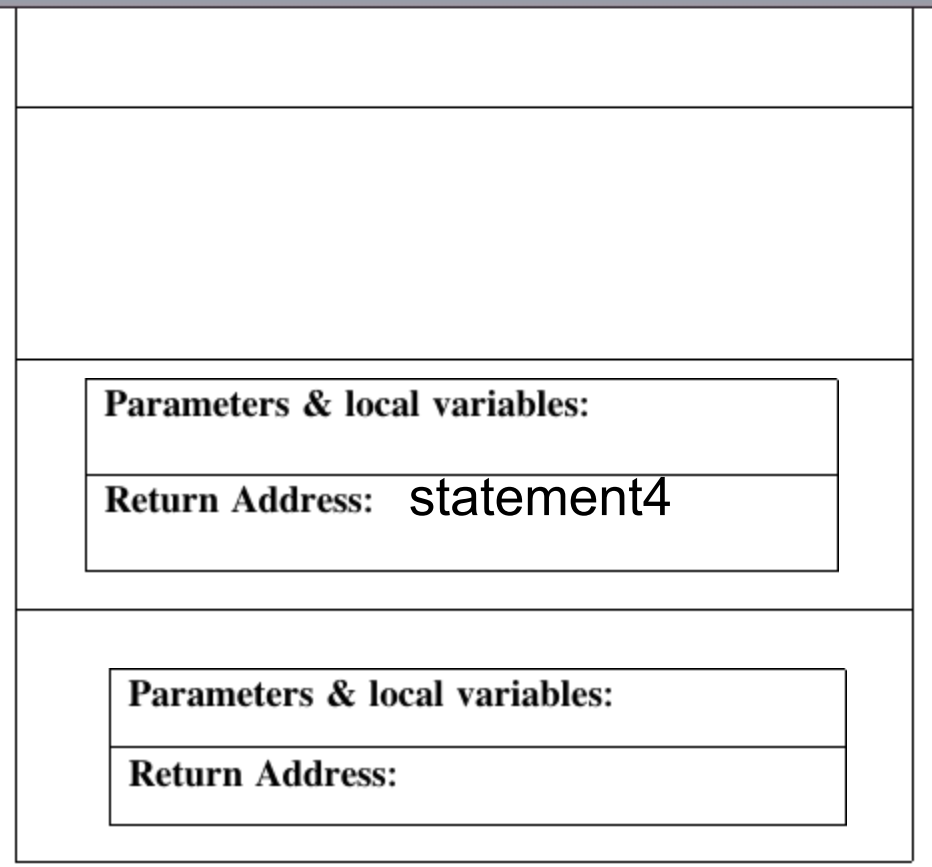
Runtime stack/System Stack is a stack whose each element is the activation records of the different functions being called during the running of the program



Runtime Stack

```
int main()
{   int x,y;
    statement1;
    A();
    statement2;
    statement3;
    B();
    statement4;
}
void A()
{   C();
    statement 5;
}
```

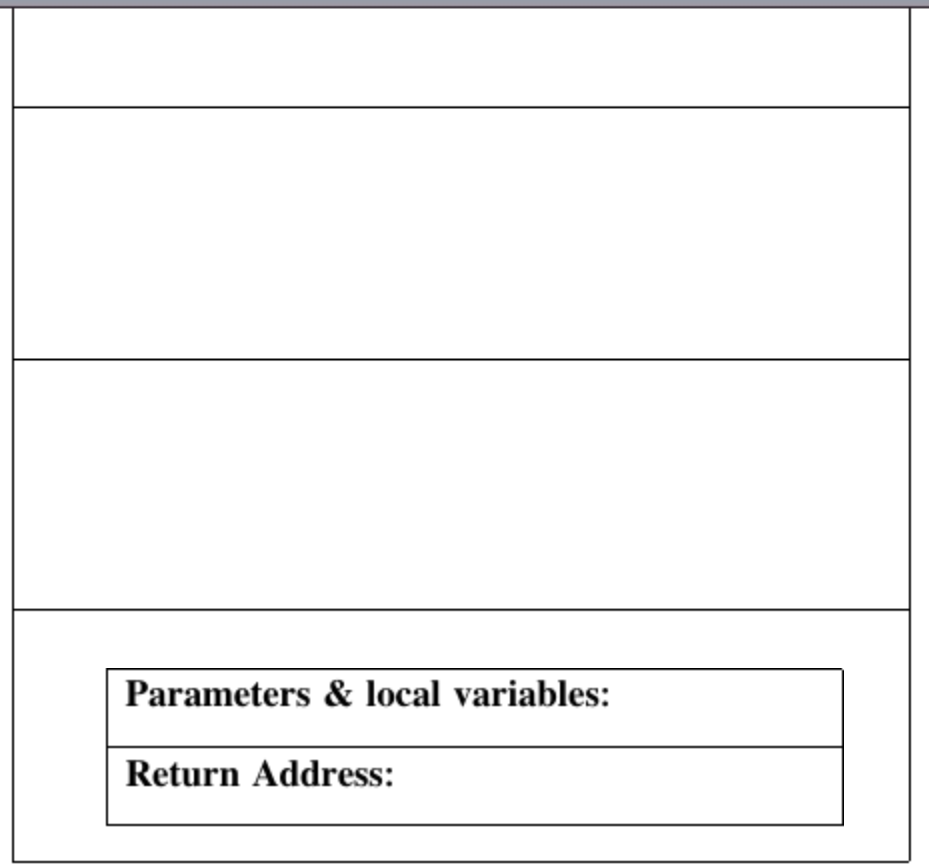
Runtime stack/System Stack is a stack whose each element is the activation records of the different functions being called during the running of the program



Runtime Stack

```
int main()
{   int x,y;
    statement1;
    A();
    statement2;
    statement3;
    B();
    statement4;
}
void A()
{   C();
    statement 5;
}
```

Runtime stack/System Stack is a stack whose each element is the activation records of the different functions being called during the running of the program



Runtime Stack

```
int main()
{   int x,y;
    statement1;
    A();
    statement2;
    statement3;
    B();
    statement4;
}
void A()
{   C();
    statement 5;
}
```

Runtime stack/System Stack is a stack whose each element is the activation records of the different functions being called during the running of the program

Runtime Stack

- The C++ run-time system keeps track of the chain of active functions with a stack
- When a function is called, the run-time system pushes on the stack a frame containing
  - Local variables and return value
  - Program counter, keeping track of the statement being executed
- When a function returns, its frame is popped from the stack and control is passed to the method on top of the stack

```
main() {
  int i = 5;
  foo(i);
  }

foo(int j) {
  int k;
  k = j+1;
  bar(k);
  }

bar(int m) {
  …
  }
```