Stack, Queue

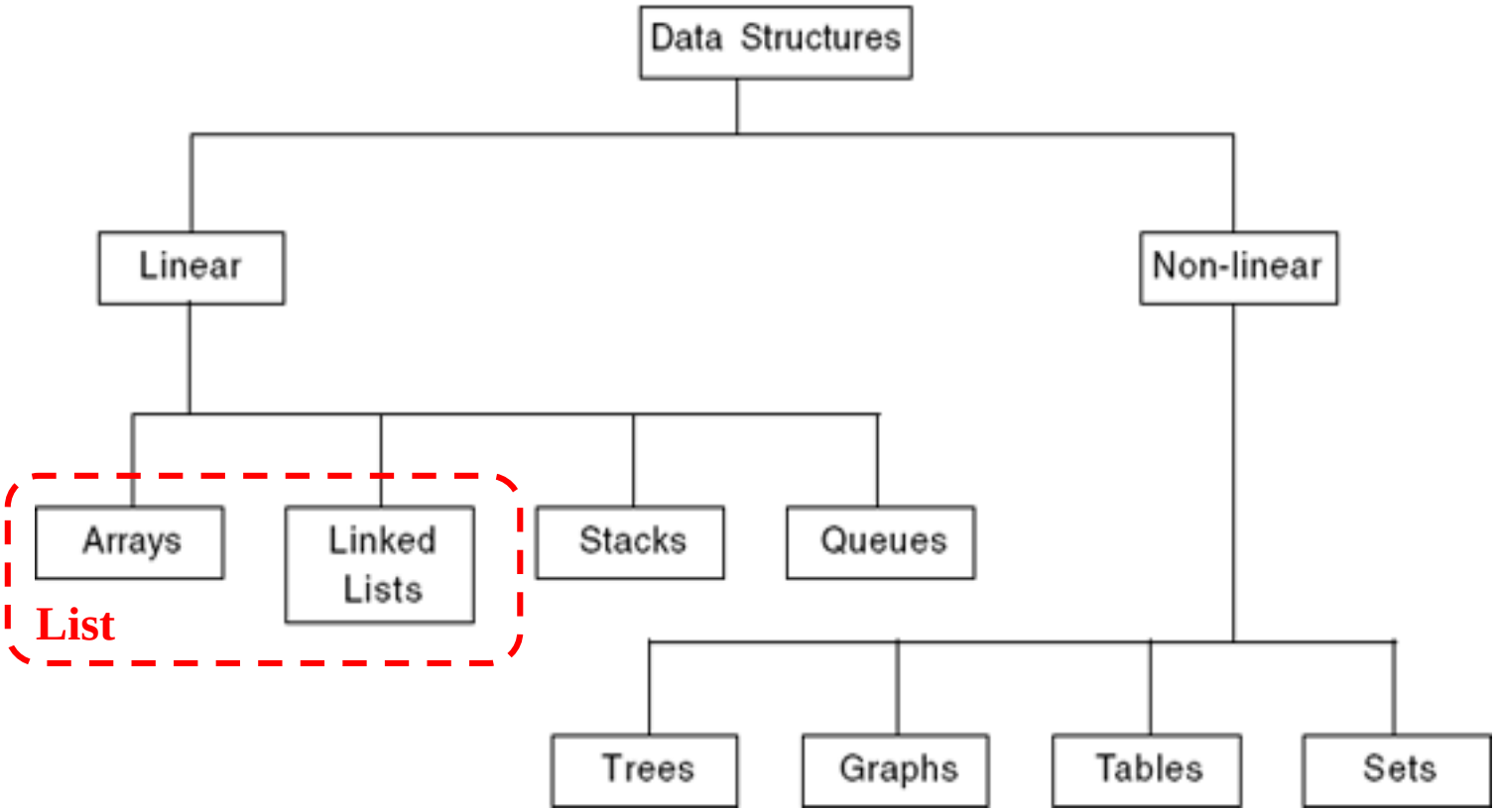
CSC-114 Data Structure and Algorithms

Slides credit: Ms. Saba Anwar

Outline

- Stack
  - How it Works?
  - Implementation
  - Applications
    - Arithmetic expression evaluation
- Queue
  - How it Works?
  - Implementation
  - Variations
    - Circular
    - Priority
    - Deque

Data Structures

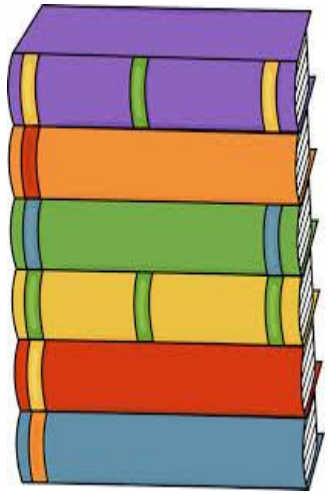


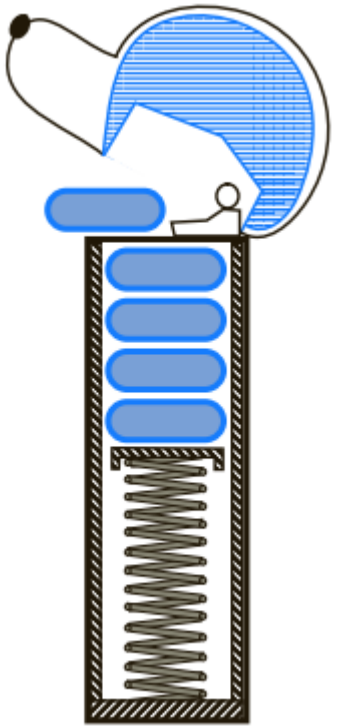
Stack

A ***stack*** is a linear data structure in which objects are inserted and removed according to the ***last-in first-out*** (***LIFO***) principle. Objects can be inserted into a stack at any time, but only the most recently inserted (that is, "last") object can be removed at any time.

Examples:

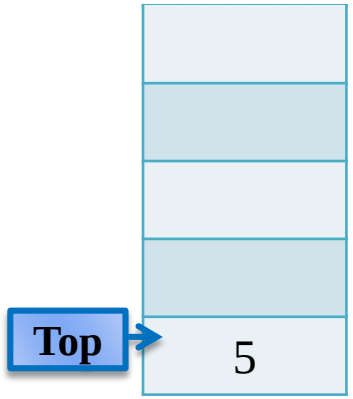
- Pile of books
- Goods in cargo
- Do/undo list in text editor
- Web page history



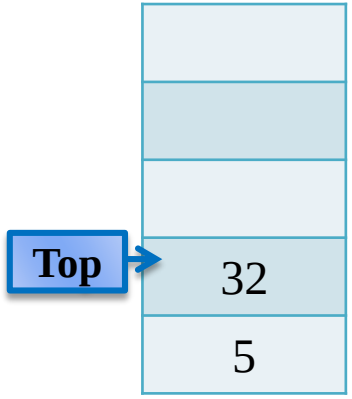


How Stack Works?

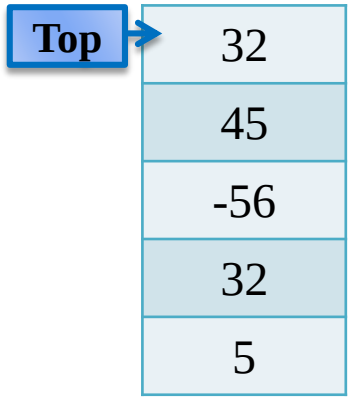
Push (5)



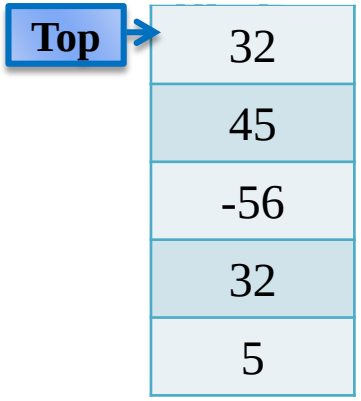
Push(32)



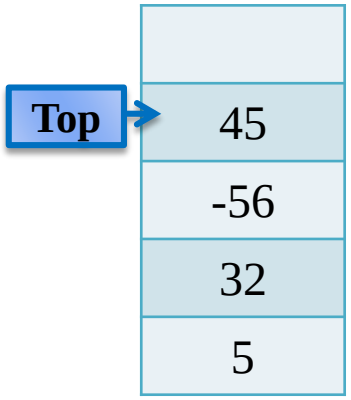
Push(-56), Push(45), Push(32)



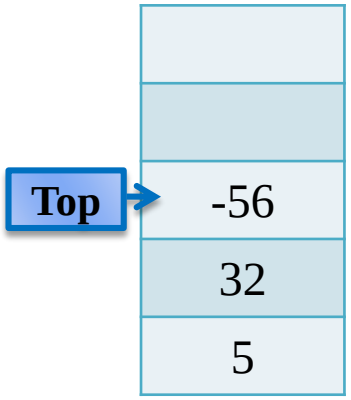
Push(11) → **Stack is full**



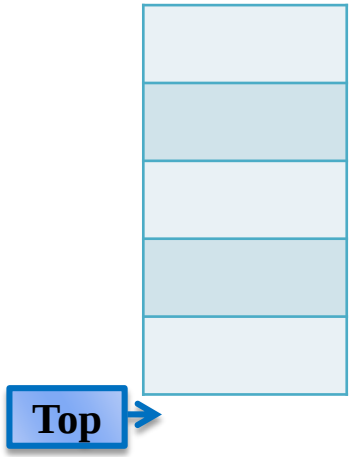
Pop()



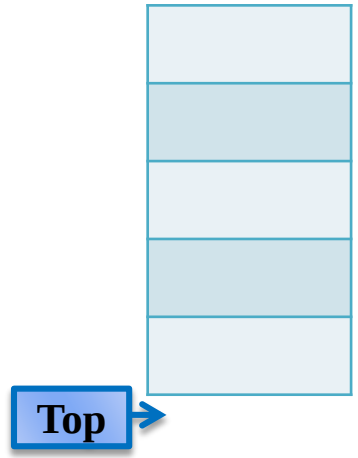
Pop()



after 3 more pop()



Pop()→**Stack is empty**



Stack as ADT

Structure

New elements is added on top of existing elements

Only most recently added element is accessible, also called top element

An implicit cursor (**top)** is maintained, that points to most recently added item

Operations

push(E): insert the new element on top of stack

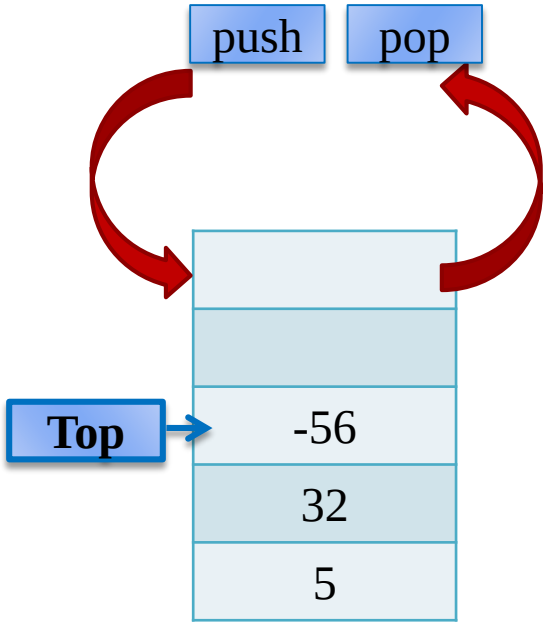
pop(): remove the top element of stack

- Helper methods:

peek() or top(): returns the top element of stack, no removal

size(), isEmpty()

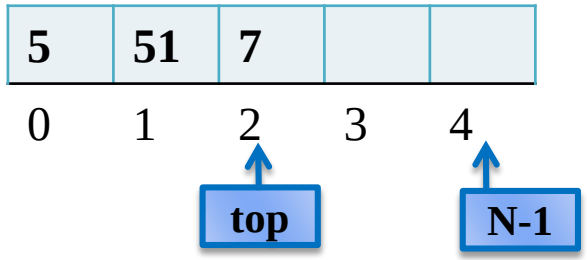
isFull(): if stack is fixed size

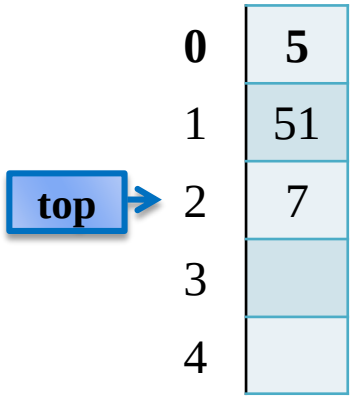


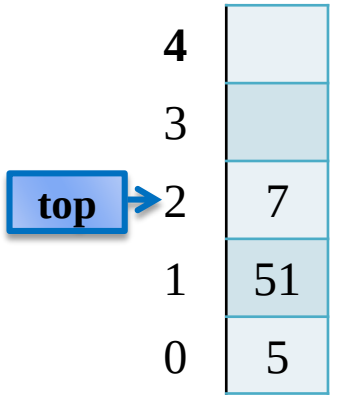
Implementation

Using array

Add and delete at end → O(1)



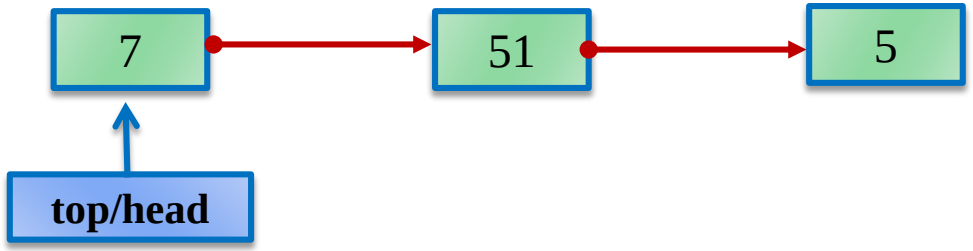




Top to bottom & bottom to top

Using Linked List

Add and delete at start → O(1)



Implementation

Algorithms:

algorithm depends upon individual implementation

In array

a variable top is maintained to denote most recent element's index

top =-1 →stack is empty

Size = top+1

top =N-1 →stack is full

In Linked list

Head node is top

Head=null→ stack is empty

- No need of tail

Algorithms(Array based)

Algorithm: Push(Stack, E)

Input: a Stack, a data element E

Output: updated stack with E inserted

Steps:

1. If(Stack is Full)
2. Print “Stack overflow”
3. Else
4. top=top+1
5. Stack[top]= E
6. End If

Algorithm: Pop(Stack)

Input: a stack

Output: updated stack, with top element removed

Steps:

1. Let E = null
2. If(Stack is Empty)
3. Print “Stack underflow”
4. Else
5. E=Stack[top]
6. top=top-1
7. End If
8. return E

Application of Stacks

Methods calls within a program, functions are called in order and data is stored in activation stack

Back button on browser store page history

Undo/redo in editors

Used by compilers to check program syntax

Arithmetic Expression Evaluation

To reverse contents of something like string, array

Decimal to binary conversion

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

Arithmetic Expression Evaluation

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

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".

Arithmetic Expression Evaluation

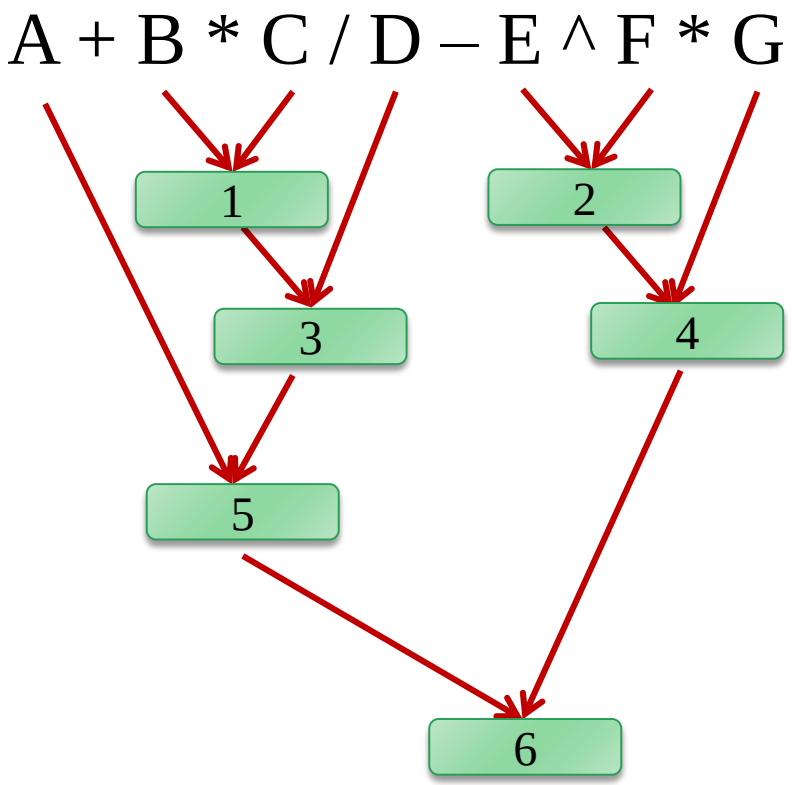
Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

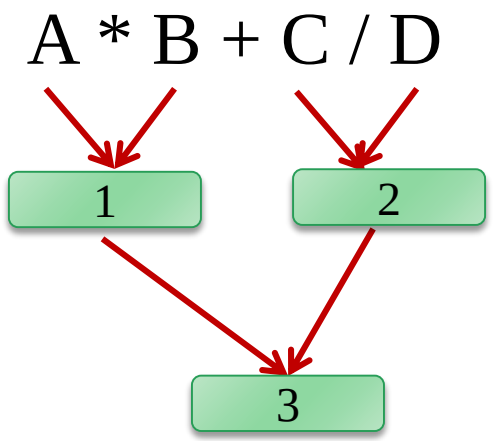
http://www.cs.man.ac.uk/~pjj/cs212/fix.html

Infix Evaluation

Order of evaluation according to precedence and associativity

Easy to understand





Postfix Evaluation

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

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

Postfix Evaluation

We can evaluate a postfix expression using a stack.

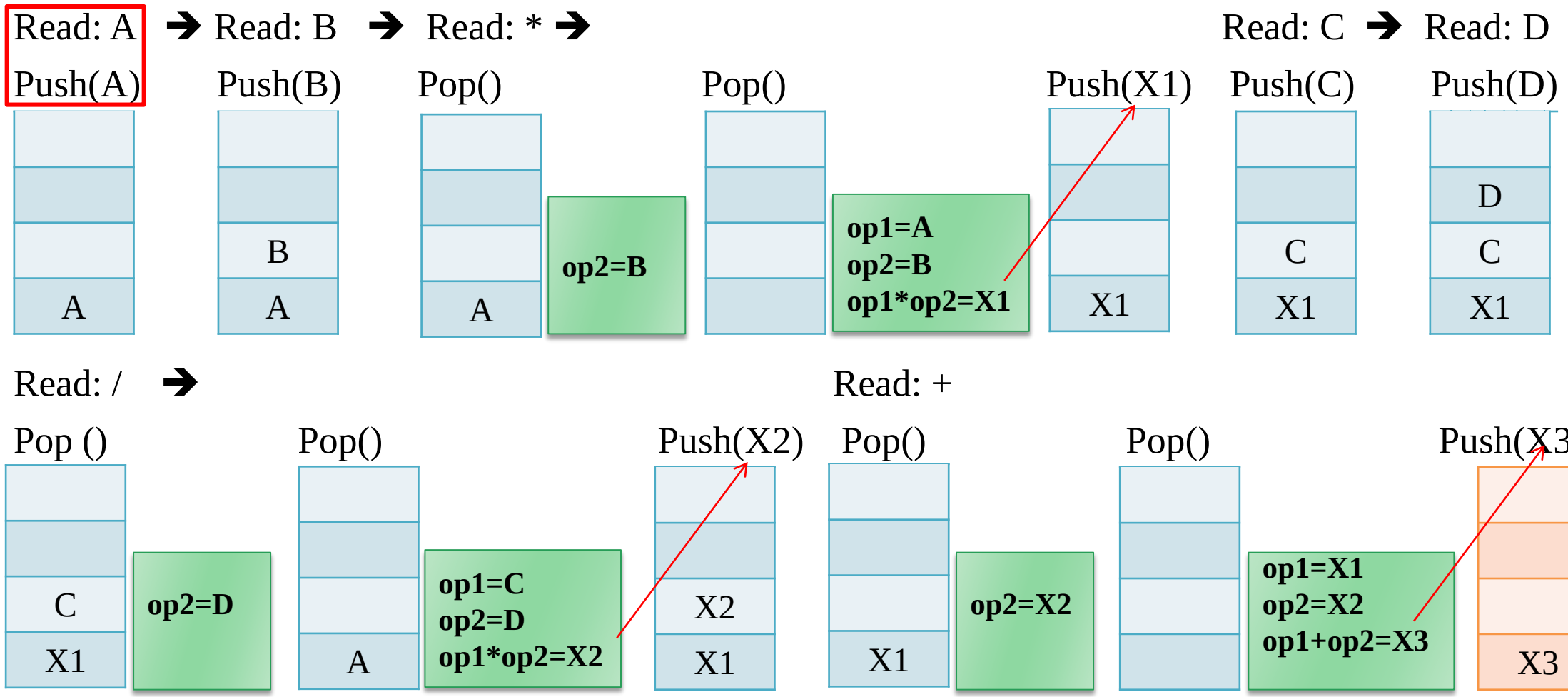
- Each operator in a postfix expression corresponds to the previous two operands .
- Each time we read an ***operand*** we push it onto a stack.
- Each time we read an ***operator,*** its associated operands (the top two elements on the stack) are popped out.
- Operator is performed on them and result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

  A B * C D / +

  6 5 2 3 + 8 * + 3 + *

Postfix Evaluation – With Stack (Example-1)

A B * C D / +

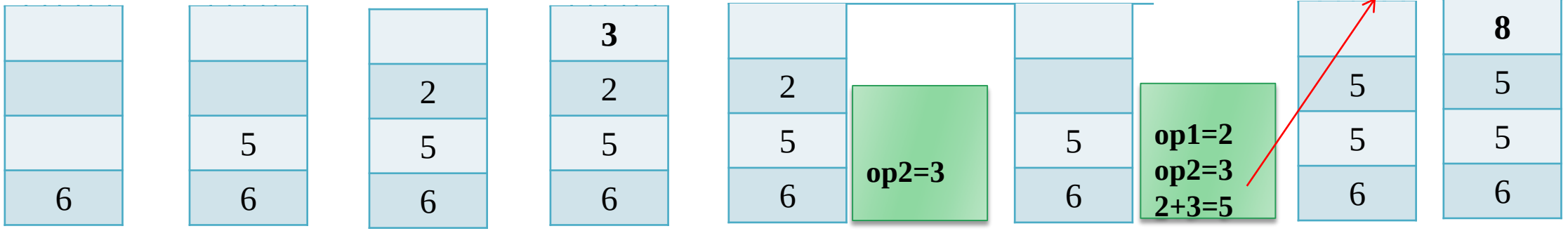


Postfix Evaluation – With Stack (Example-2)

6 5 2 3 + 8 * + 3 + *

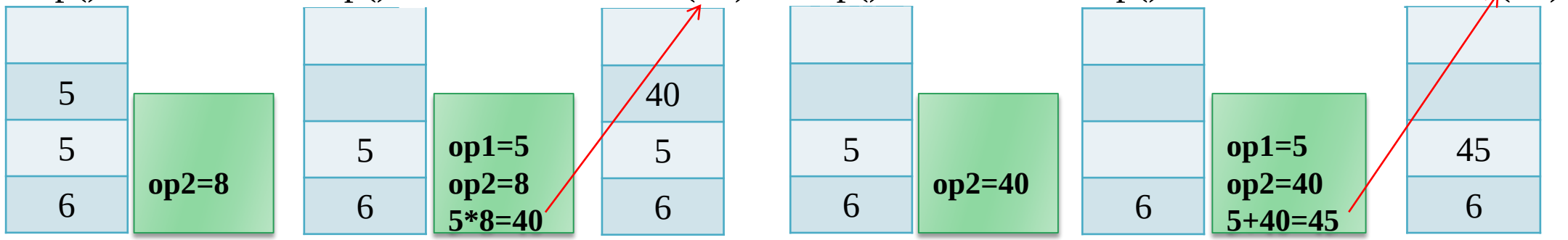
Read: 6 ➔ Read: 5 ➔Read: 2 ➔ Read: 3 ➔Read: + ➔ Read: 8

Push(6) Push(5) Push(2) Push(3) Pop() Pop() Push(5) Push(8)



Read: * ➔ Read: + ➔ Read: +

Pop() Pop() Push(40) Pop() Pop() Push(45)

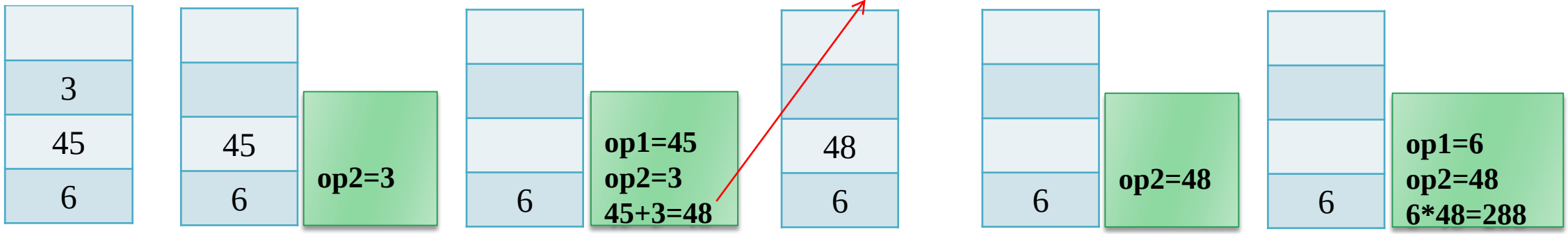


Postfix Evaluation – With Stack (Example-2)

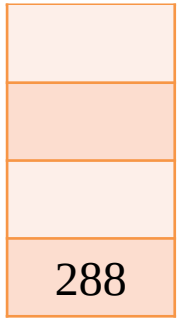
6 5 2 3 + 8 * + 3 + *

Read: 3 ➔ Read: + ➔ Read: *

Push(3) Pop() Pop() Push(48) Pop() Pop()



Push(288)



Postfix Evaluation: Algorithm

Input: a post fix expression string

Output: answer of expression

Steps:

1. Let say S is a character stack
2. While(input expression has more characters)
3. Read the next character
4. If character is an operand
5. push it to S
6. else **//character is operator**
7. op2 = pop S
8. op1 = pop S
9. Apply the operator to op1 and op2
10. Push the result to stack
11. End While
12. Pop the stack for final result

Postfix Evaluation

Most programming languages use postfix notation to solve arithmetic expressions, because of its left to right easy evaluation order. But expressions are written in infix notation.

So, compiler needs to do two things:

- Conversion of infix expression into its equivalent postfix expression.
- Evaluation of the postfix expression

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 *) +

Remove parenthesis

Postfix Expression: a b c * +

Infix Expression: (a +b) * c

Convert addition

(a b +)* c

Convert the multiplication

(a b + c *)

Remove parenthesis

Postfix Expression: a b + c*

Keep in Mind:

1. Relative order of variables is not changed
2. No parenthesis in postfix/prefix
3. Operators are arranged according to precedence
4. If same precedence operators, then evaluate left to right

Infix to Postfix Conversion

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/))+

Remove parenthesis

Postfix Expression: a b c * d/+

Infix Expression: ((a + b) * c – (d – e))/(f + g)

Convert operators in parenthesis

((a b+)*c-(d e-))/(f g+)

Convert multiplication

((a b+)c* -(d e-))/(f g+)

Convert the division

((a b+)c* -(d e-))(f g+)/

Convert subtraction

((a b+)c* (d e-)-)(f g+)/

Remove parenthesis

Postfix Expression: a b + c * d e - - f g + /

More Examples

Infix	Postfix
(a + b) * (c – d)	a b + c d - *
a – b / (c + d * e)	a b c d e * + / -
((a + b) * c – (d – e))/(f + g)	a b + c * d e - - f g + /
(300+23)*(43-21)/(84+7)	300 23 + 43 21 -* 84 7 + /

Can we use **stack** to convert an infix expression into post fix expression

Infix to Postfix: Algorithm

We have learned how to convert infix into postfix expression

But that was not a systematic way to do the task.

We want to write an algorithm which can convert a given infix expression into postfix expression.

- Let say we have an infix expression as a string,
- What to do next?

A stack can be very helpful for this conversion

- Let see how?

Infix to Postfix: Algorithm

Input: a infix expression string

Output: a postfix expression string of given input

Steps:

1. Let P is string, and S is character stack
2. While(input expression has more characters)
3. Read the next character
4. If character is an operand
5. append it to P
6. Else If character is an operator
7. pop S, until top of the S has an element of lower precedence
8. append popped character to P
9. Then push the character to S
10. Else If character is "("
11. Push it to S
12. Else If character is ')'
13. pop S until we find the matching '('
14. append popped character to P
15. pop "(" // '**(**' has the lowest precedence when in the stack but has the highest precedence when in the input
16. End If
17. End While
18. Pop until the stack is empty and append popped character to P

Keep in Mind:

1. Relative order of variables is not changed
2. No parenthesis in postfix
3. Operators are arranged according to precedence
4. If same precedence operators, then evaluate left to right

Examples are given in infix-to-postfix slides

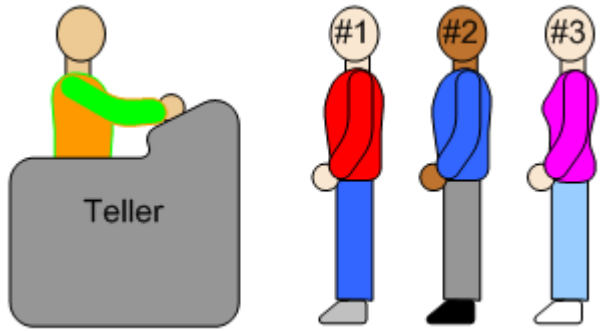
Queue

A queue is a container in which elements are inserted and removed according to the ***first-in first-out*** (***FIFO***) principle.

Elements are inserted at **rear/back** end and removed from **front** end.

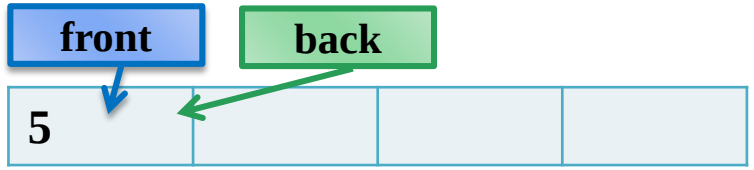
Example:

- A queue of people at bank
- A queue of passengers at airport for boarding
- A line of vehicle at toll booth
- Process scheduling in operating system
- Printing requests to printer
- Client request to server

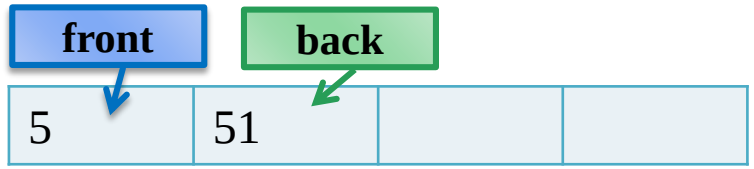


How Queue Works?

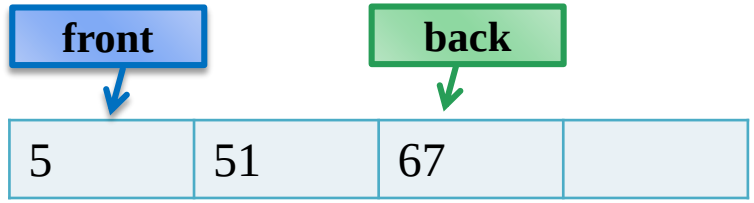
Enqueue(5)



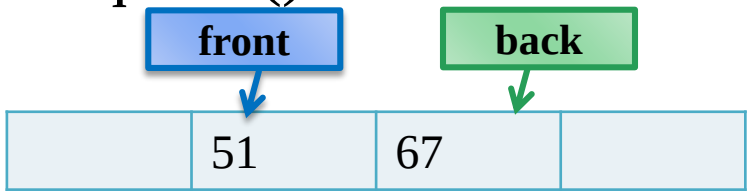
Enqueue(51)



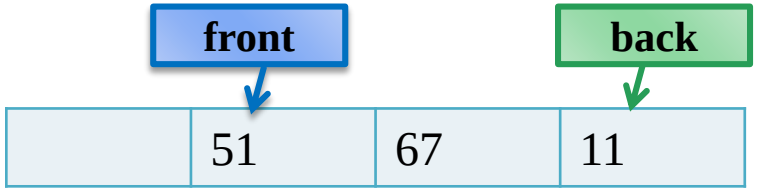
Enqueue(67)



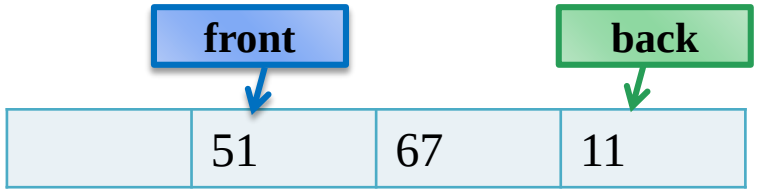
Dequeue()



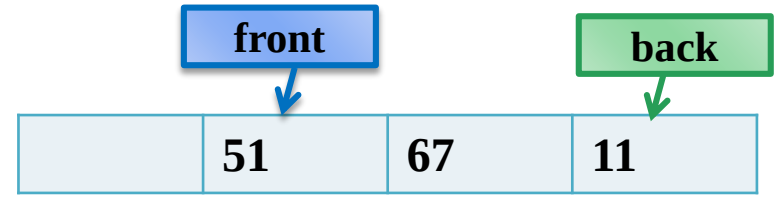
Enqueue(11)



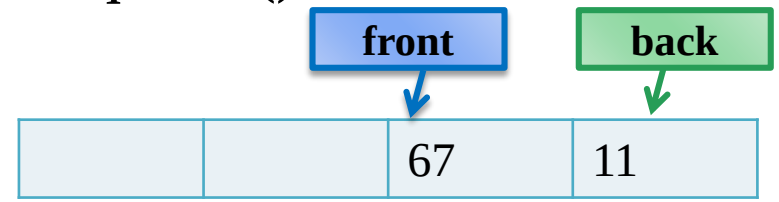
Enqueue(3)→ **full**



How Queue Works?

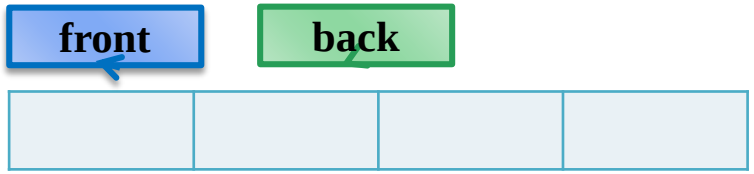


Dequeue()

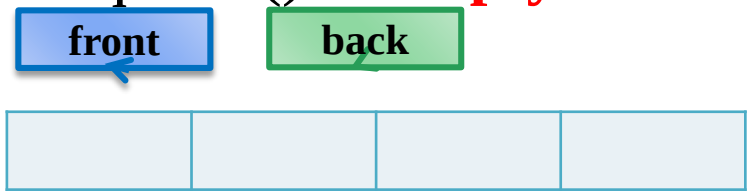


Dequeue()

Dequeue()



Dequeue()→ **empty**



No elements cab be added if queue is full

No more elements can be removed if queue is empty.

Queue as ADT

Structure

An ordered list where only the oldest item is accessible. Elements are added to the rear/back and removed from the front; a 'first in, first out' (FIFO) structure."

Operations:

enqueue(E): add the element at back or end of existing data

dequeue(): remove the element from front end

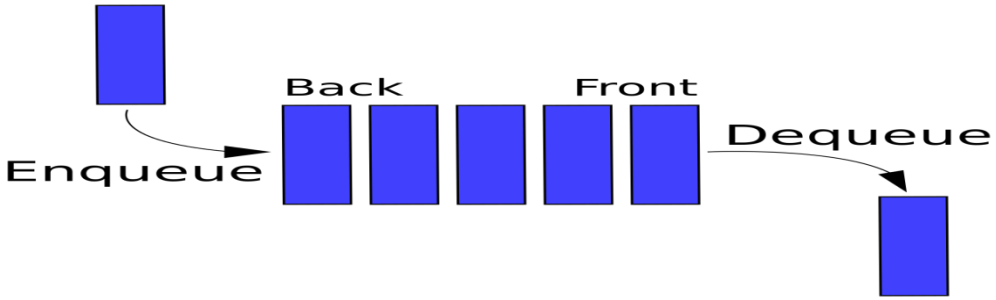
- **Helper methods**

front(): return the front element (no removal)

size(): tells number of elements in queue

isEmpty(): tells if queue is empty or not

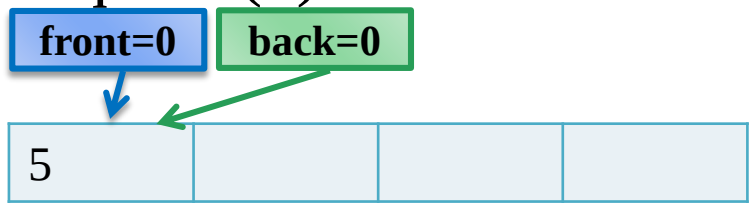
isFull(): if queue is full or not



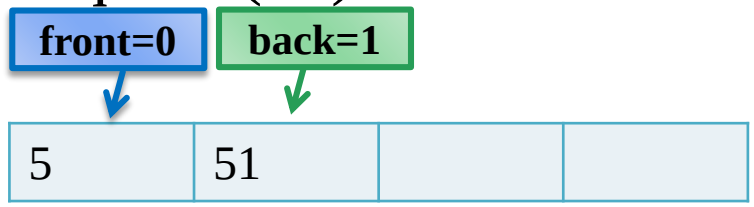
Enqueue

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

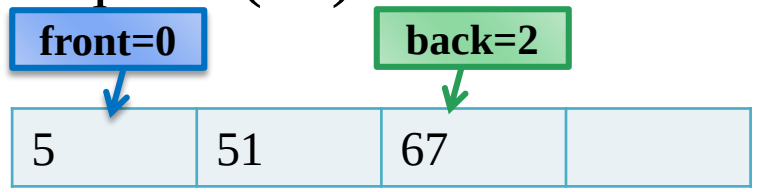
Enqueue(5)



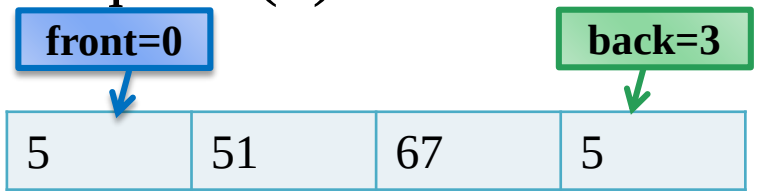
Enqueue(51)



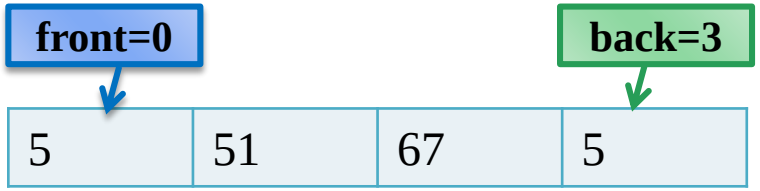
Enqueue(67)



Enqueue(5)



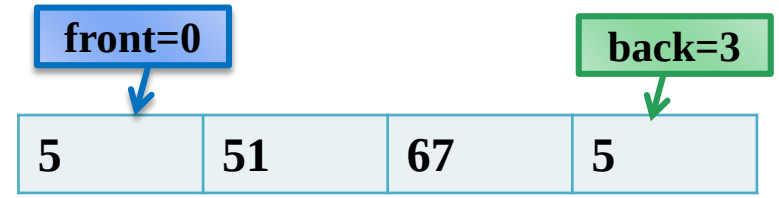
Eequeue(91)➔ Error



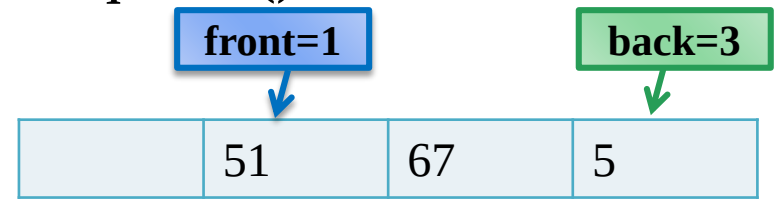
No more elements can be inserted if queue is full.

How to know if queue is full?

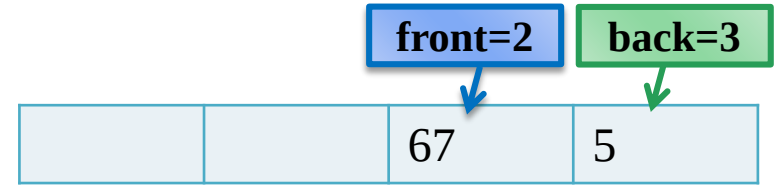
Dequeue



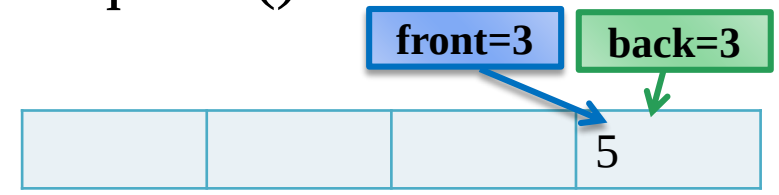
Dequeue()



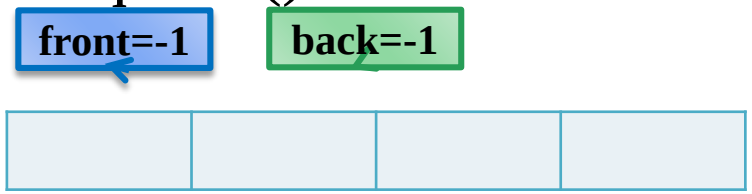
Dequeue()



Dequeue()



Dequeue()



Dequeue()➔ Error

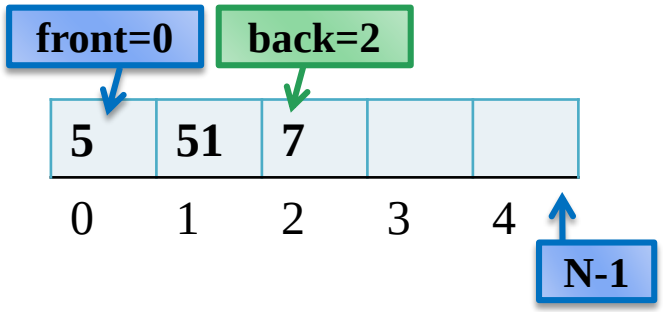
No more elements can be removed if queue is empty.

How to know if queue is empty?

Implementation

Using array

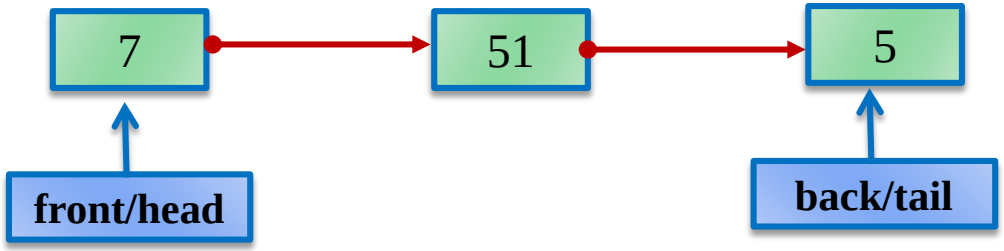
Add at end, delete from start → O(1)



Helper Methods	Array	Linked List
isEmpty()	front, back = -1	Front/head=null
isFull()	back+1= N	N/A
Size()	back-front + 1	Count Nodes
Front()	Return element at front index	Return front/head node

Using Linked List

Add at end, and delete at start → O(1)



Algorithms (Array based)

Algorithm: **ENQUEUE** (Queue, E)

Input: a queue, an element to be added

Output: queue with E inserted

Steps:

1. If(Queue isFull)
2. Print "Queue overflow"
3. Else
   Back=Back+1
1. Queue[Back]=E
   If(Front==-1) //this is first item
   Front=0
   End If
1. End If

Algorithm: **DEQUEUE** (Queue)

Input: a queue

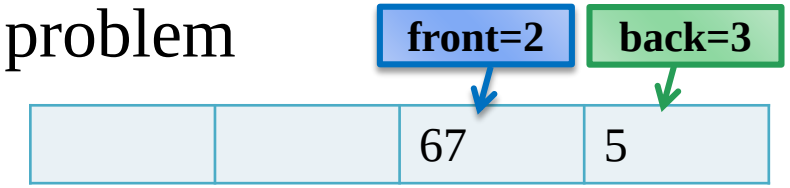
Output: queue with front element removed

Steps:

1. Let E = null
2. If(Queue isEmpty)
3. Print "Queue underflow"
4. Else
5. E=Queue[front]
6. If(Front==Back) //this is last item
7. Front=-1, Back=-1
8. Else //there are more items
9. Front=Front+1
10. End If
11. End IF
12. return E

Circular Queue

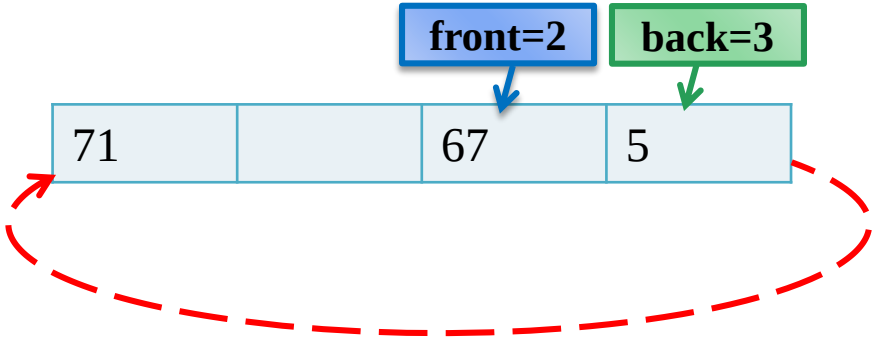
The simple array implementation of queue has a problem



- It is wasting space
- Because front is moving towards back and leaving empty space behind.
- Elements are stored towards end of array and front cells 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

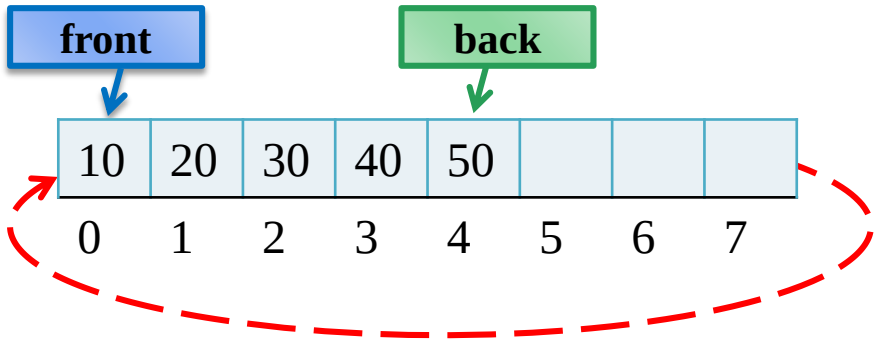
Examples:

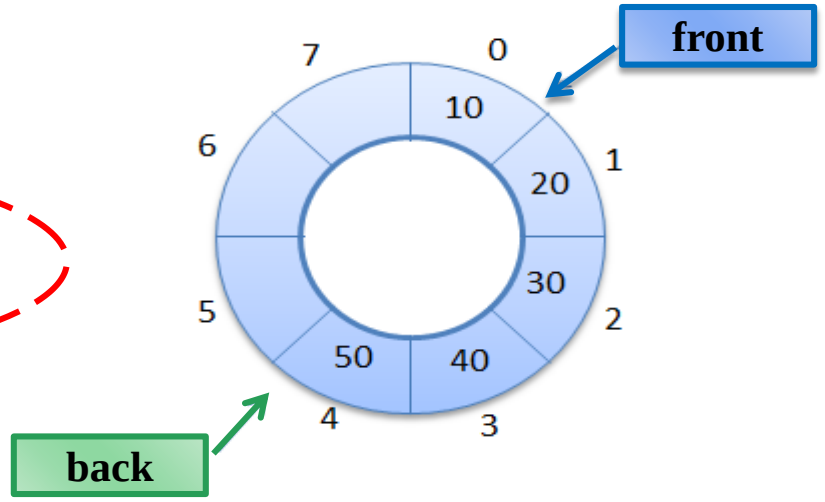
- Round robin scheduling in operating system
- Games where players take turns
- Musical chair game played in schools



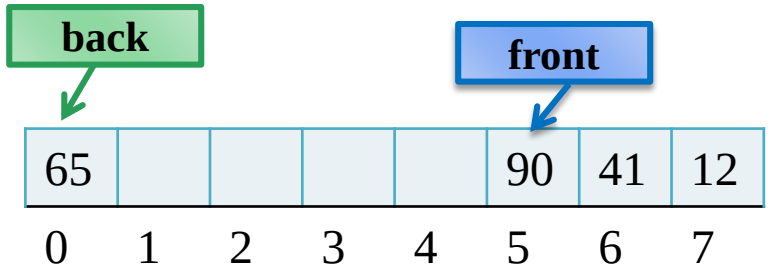
Circular Queue

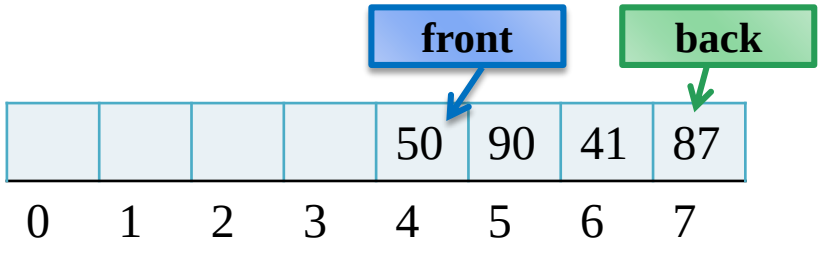
Circular queue is a simple queue, But logically it says that queue[0] comes after queue[N-1]

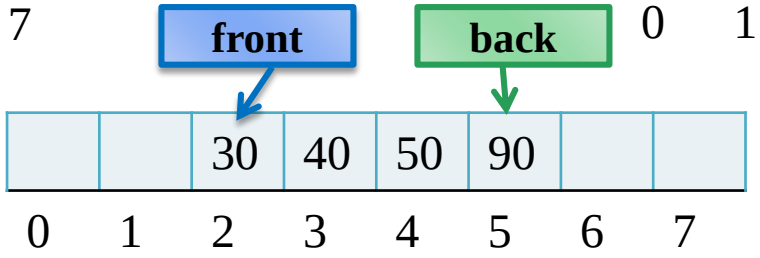




Elements can be anywhere in queue.



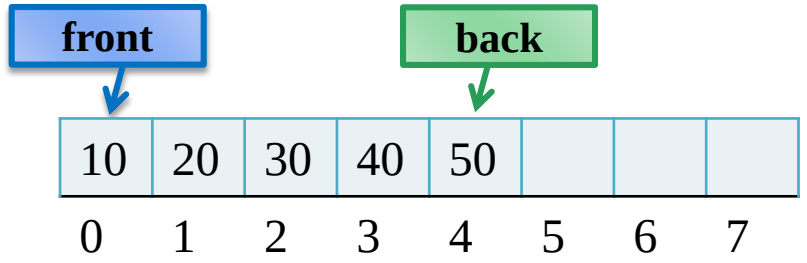




How it Works?

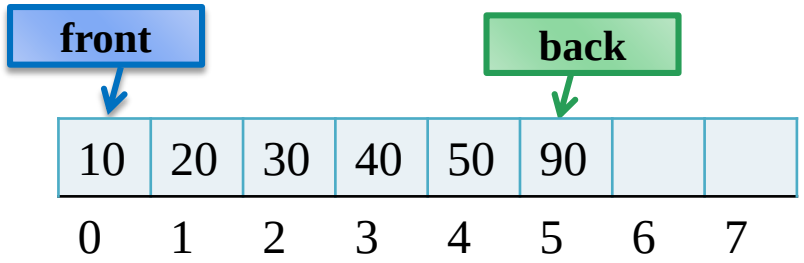
In circular queue, both front and back move in clockwise direction

Let say after 5 Enqueue(item) state of queue is:

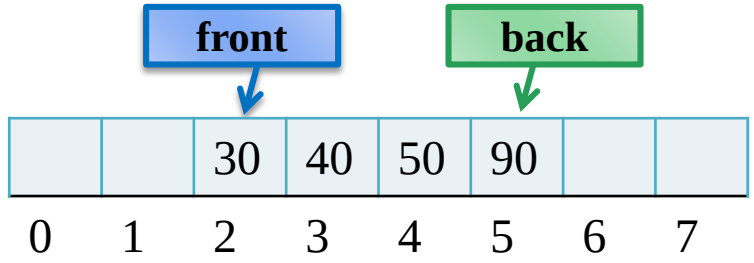


Now perform following operations:

Enqueue(90)



Dequeue() → Dequeue()



How it Works?

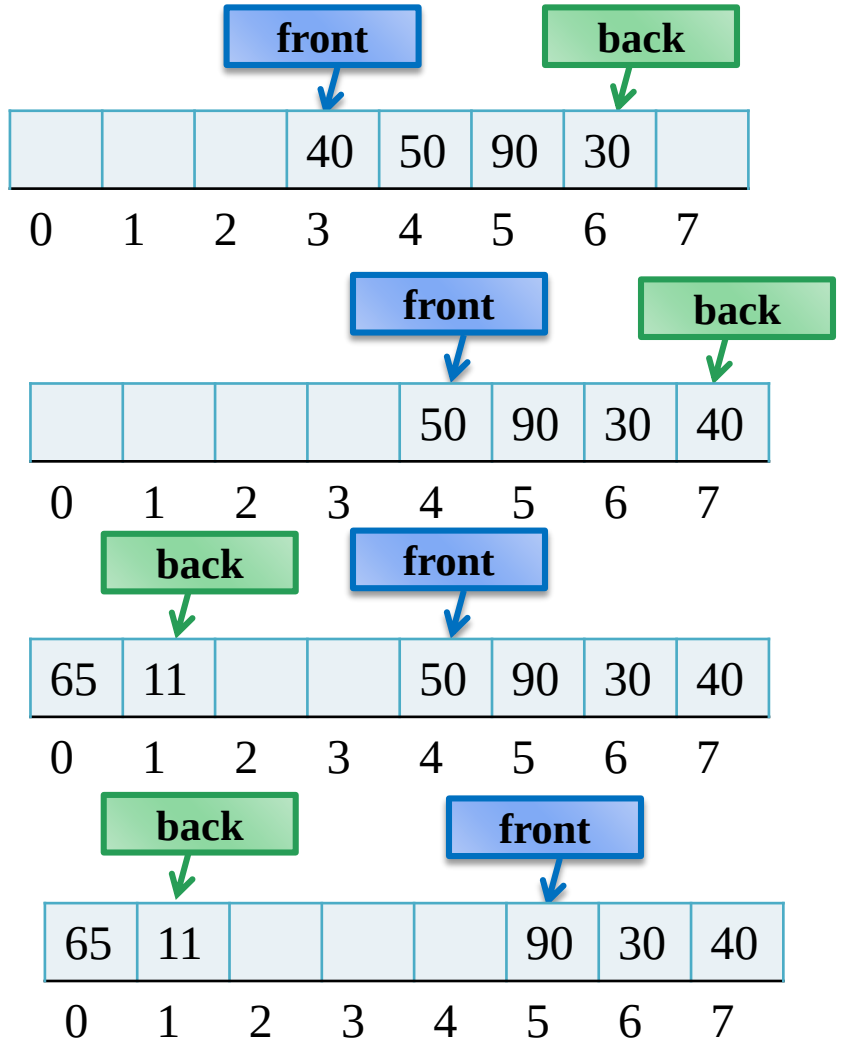
Enqueue(Dequeue())

Remove element from front and enqueue it

Enqueue(Dequeue())

Enqueue(65)→ Enqueue(11)

Dequeue()



How it Works?

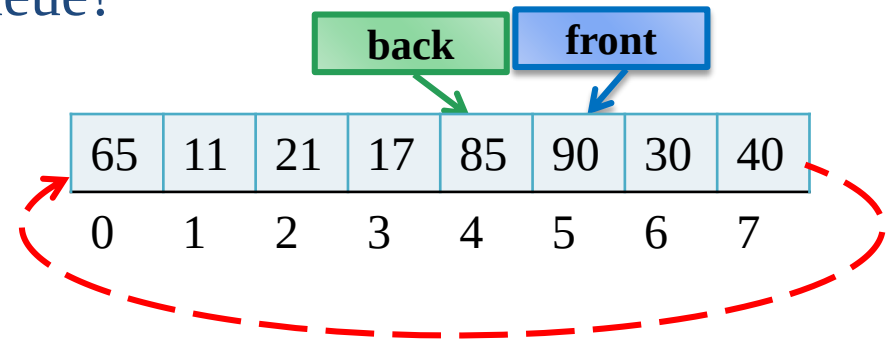
How to find if Circular Queue is Empty?

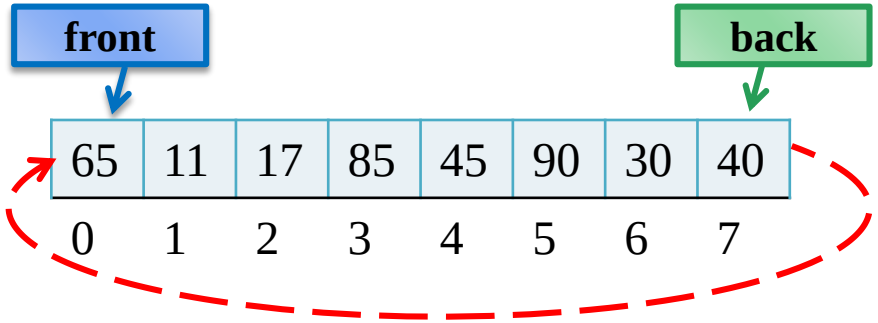
Front=Back= -1

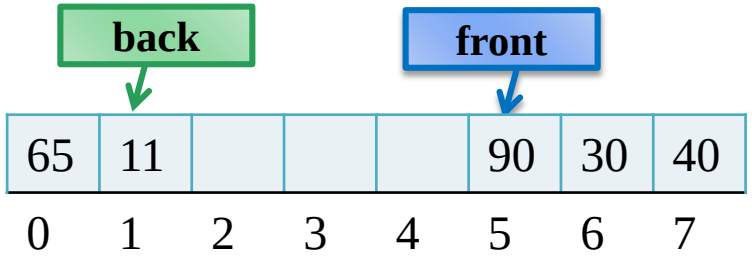
- Full?

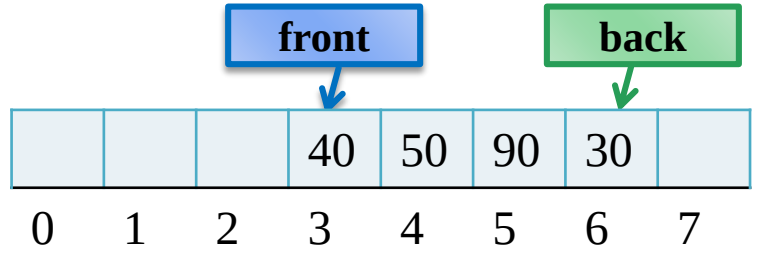
Where to Enqueue?

Its size?









N=8

Algorithms

Algorithm:CIRCULAR_ENQUEUE(Queue, E)

Input: a queue, an element E

Output: updated queue with E inserted

Steps:

1. If(Queue is Full)
2. Print “Queue overflow”
3. Else
4. Back=(Back+1)%N
5. Queue[Back]=E
6. IF(Front==-1)
7. Front=0
8. End If
9. End If

Algorithm: CIRCULAR_DEQUEUE(Queue)

Input: a queue

Output: updated queue, with front element removed

Steps:

1. Let E = null
2. IF(Queue is Empty)
3. Print “Queue underflow”
4. Else
5. E=Queue[Front]
6. Front=(Front+1)%N
7. IF(Front==Back)
8. Front=Back=-1
9. End If
10. End If
11. return E

Helper Operations

Helper Methods	Queue	Circular Queue
isEmpty()	front, back = -1	same
isFull()	back+1= N	(back+1)%N= F
Size()	back-front + 1	(N-front+back)%N + 1 If front<back back-front + 1 Else (N-front) +(back+1) Back

Priority Queue

Imagine a ticket window at Daewoo Terminal, the person who arrives first will get the ticket before the person who arrives later.

Now imagine process queue for CPU, ideally it should execute the process whose request was arrived first, but some time situation arise when a component/process raise request that is of highest priority and needs to be executed first regardless of the order of arriving in process queue, like computer shut down

That is the case where we need a queue which can handle priority.

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 as ADT

Operations:

Enqueue(E)

Dequeue(): remove the item with the highest priority

Find() return the item with the highest priority

Examples

Process scheduling

Few processes have more priority

- Job scheduling

  N Jobs with limited resources to complete

- Patients treatment in emergency

Priority Queue as ADT

First Approach:

- enqueue operation add item at the back of the queue →O(1)
- dequeue operation removes item with highest priority → O(N)
  - Find highest priority element, remove it by shifting elements to left

Second Approach:

- enqueue operation insert items according to their priorities. → O(N)
  - A higher priority item is always enqueued before a lower priority element, so item with highest priority will always be at start.
- dequeue operation removes an item from front end → O(1)

Key Note:

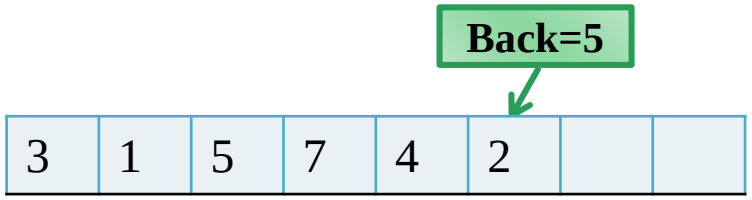
- If two or more elements have same priority, then they follow FIFO. Element that came first, will be removed first.

First Approach

Enqueue(3)→Enqueue(1)

Enqueue(5)→ Enqueue(7)

Enqueue(4)→ Enqueue(2)

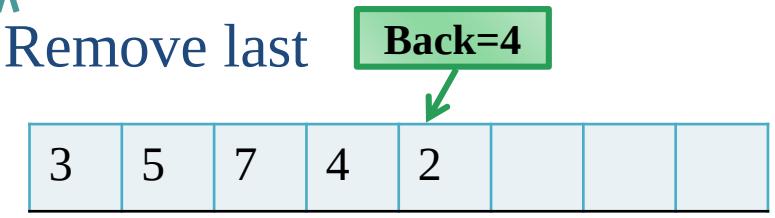


Dequeue()

Find highest priority→1

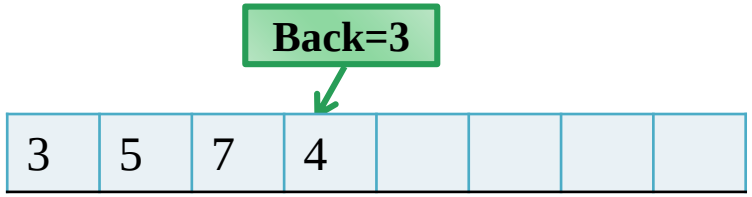
~~Swap with last~~

Remove last



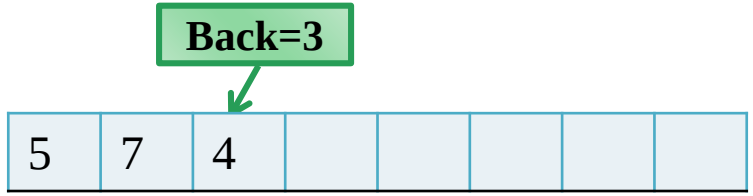
Dequeue()

highest priority→2



Dequeue()

highest priority→3



Algorithms

Algorithm: **ENQUEUE** (PQ, E)

Input: priority queue, an element E

Output: updated queue, with E inserted

Steps:

1. If (PQ isFull)
2. Print "Queue overflow"
3. Else
4. Back=Back+1
5. PQ[Back]=E
6. End If

Algorithm: **DEQUEUE** (PQ)

Input: priority queue,

Output: updated queue, with highest priority element removed

Steps:

1. Let E = null
2. If (PQ is Empty)
3. Print "Queue underflow"
4. Else
5. index=**FIND_INDEX_OF_MIN(PQ)** // function to find index of min
6. E=PQ[index]
7. For i=index;i<back;i++ //shift back
8. PQ[i]=PQ[i+1]
9. End for
10. back--
11. End if
12. return E

Algorithms

Algorithm: FIND_INDEX_OF_MIN()

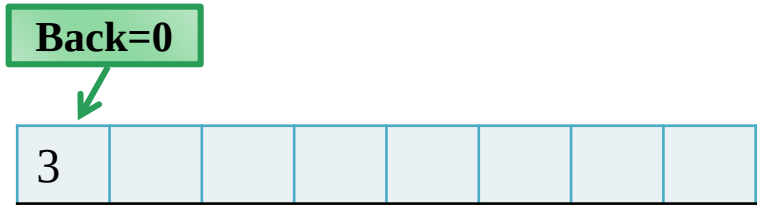
1. Let m=0
2. **For** i=1; i< =Back; i++
3. If PQ[i]'s priority <PQ[m]'s priority // it depends what is stored in array, priority as a number itself, or an object which contains data and priority as attributes
4. m=i
5. End if
6. Return m

	Priority Queue simulation
1	Use single array of integers which only store priority
2	Use one array for data, other for priority
3	Use an object's array, where object contains both its data and priority as well

Second Approach

Enqueue(3)

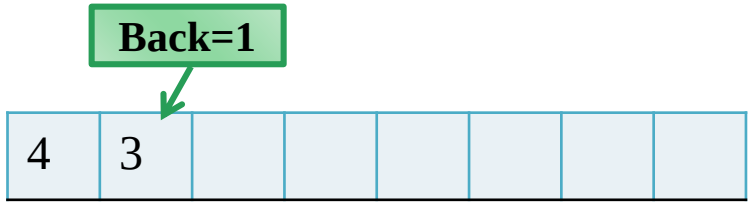
Queue is empty, so insert at front



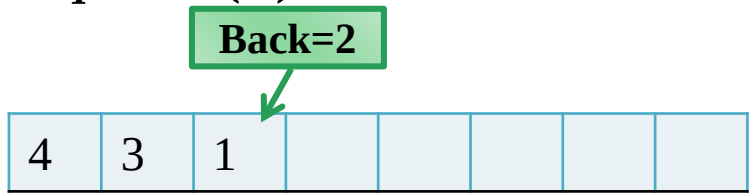
Enqueue(4)

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

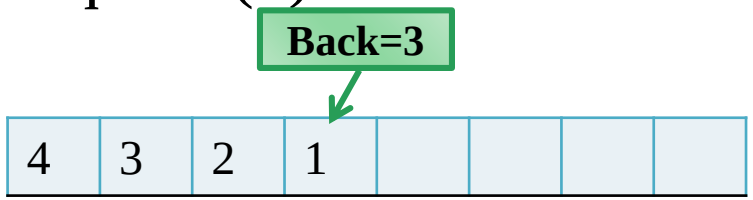
shift elements to right and insert, if needed



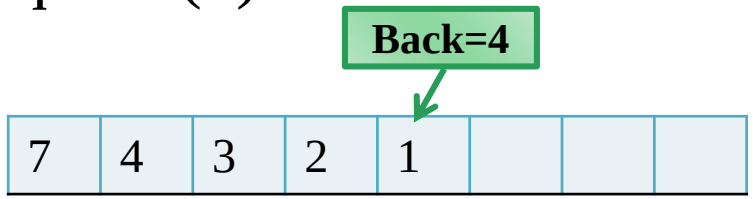
Enqueue(1)→



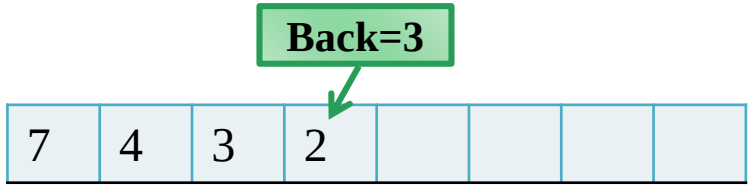
Enqueue(2)



Enqueue(7)



Dequeue()



Algorithms

Algorithm: **ENQUEUE** (PQ, E)

Input: priority queue, an element E

Output: updated queue, with E inserted

Steps:

1. If(PQ is Full)
2. Print "Queue overflow"
3. Else
4. If(Back==-1)
5. PQ[++Back]=E
6. Else
7. Back=Back+1
8. **SORTED_INSERT()**
9.
10. End If
11. End If

SORTED_INSERT()

1. Set i=0
2. //find correct location
3. While (E's priority > PQ[i]'s priority and i <Back)
4. i=i+1
5. End While
6. // shift to right
7. For j = Back ; j > i; j--
8. PQ[j]=PQ[j-1]
9. End For
10. PQ[i]=E

Algorithms

Algorithm:DEQUEUE()

Input: priority queue,

Output: updated queue, with highest priority element removed

Steps:

1. Let E = null
2. If(PQ is Empty)
   Print “Queue underflow”
3. Else
4. E=PQ[Back]
5. Back= Back-1
6. End if
7. Return E

Implementation

Priority queues hold pairs of (data, priority). So we can implements them:

1. Using array of Objects

   Object holds data and priority

2. Using single Array

   Only store priority, assume data value as priority itself

   Here Type can be an Integer, or some user defined class

3. A third approach can be of two arrays, use one for data, other for priority in PriorityQueue class.

Deque (Double Ended Queue)

A variation of linear queue, where insertions and deletions take place at both ends.

Operations:

- enqueue_front(E)
- enqueue_back(E)
- dequeue_front()
- dequeue_back()

Regarding Priority Queue: Home Task: Read Round Robin Algorithm (Topic 5.5.2 and 5.5.3 in D. Samanta Book)

How it works

Sample operations Seqence	Queue Contents
enqueue_front(4) enqueue_front(1) enqueue_front(5)	{4} {1,4} {5,1,4}
enqueue_back(7) enqueue_back(6)	{5,1,4,7} {5,1,4,7,6}
enqueue_front(11) enqueue_back(3)	{11,5,1,4,7,6} {11,5,1,4,7,6,3}
dequeue_front()	{5,1,4,7,6,3}
dequeue_back()	{5,1,4,7,6}
dequeue_front() dequeue_front()	{1,4,7,6} {4,7,6}
dequeue_back()	{4,7}
enqueue_back(dequeue_front())	{7,4}