Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Stack

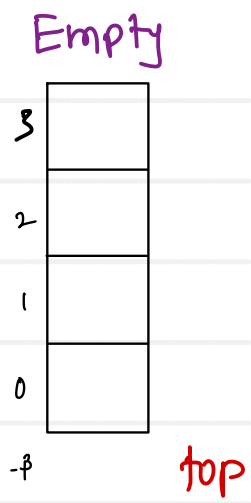
- Stack is a linear data structure which has only one end - top
- Data is inserted and removed from top end only.
- Stack works on principle of “Last In First Out” / “LIFO”

Capacity = 4

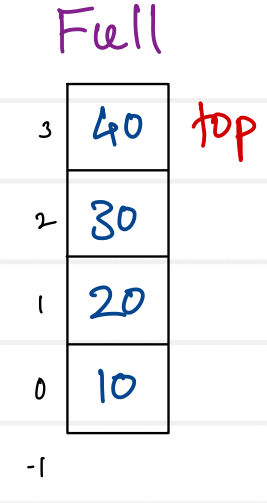
- top always points to last inserted data

operations:

1. Push:
   a. reposition top (inc)
   b. add data at top index
2. Pop:
   a. reposition top (dec)
3. Peek:
   a. read/return data of top index



top == -1



top == size-1

- all operations of stack are performed in O(1) time complexity.

Ascending stack

top = -1

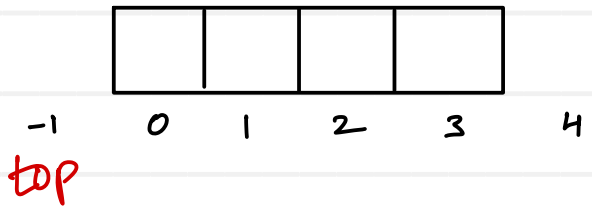
push : top++
arr[top] = value

pop : top--

peek : arr[top]

Empty : top == -1

Full : top == size-1



Descending stack

top = size

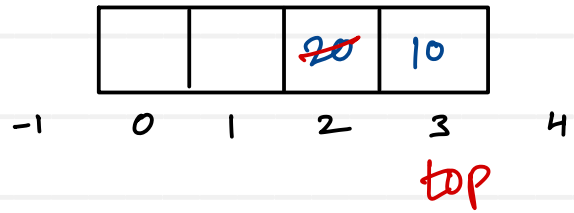
push : top--
arr[top] = value

pop : top++

peek : arr[top]

Empty : top == size

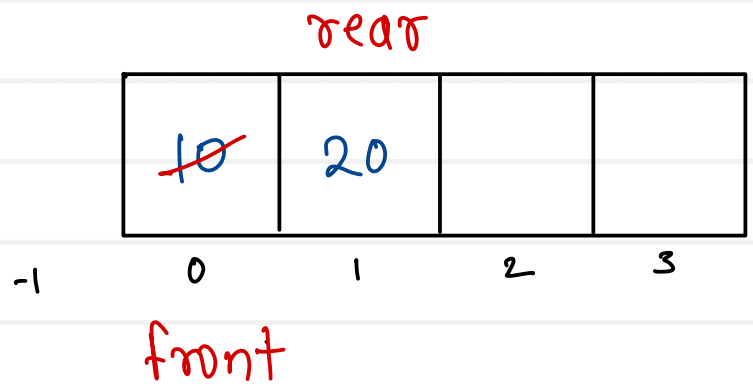
Full : top == 0



Linear queue

- linear data structure which has two ends - front and rear
- Data is inserted from rear end and removed from front end
- Queue works on the principle of “First In First Out” / “FIFO”

Capacity = 4



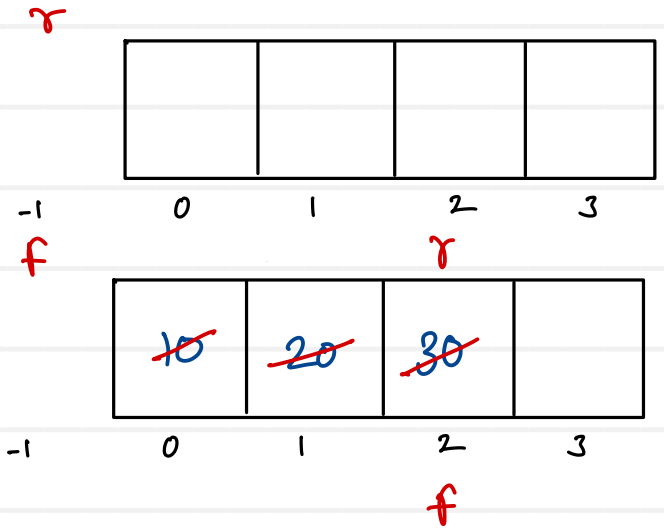
- all operations of queue are performed in O(1) time complexity.

operation:

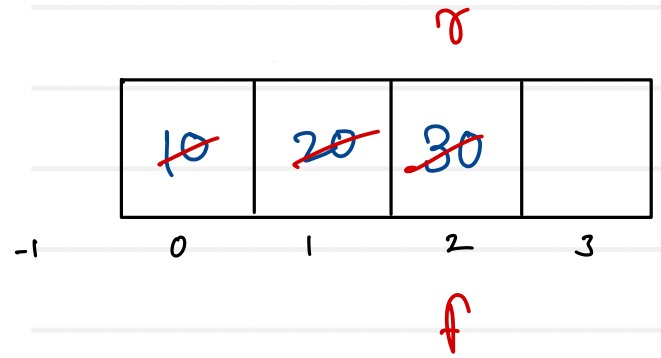
1. Enqueue
   a. reposition rear (inc)
   b. add data at rear index
2. Dequeue
   a. reposition front (inc)
3. Peek
   a. read/return data of front end
      (front+1) index

Linear queue - Conditions

Empty

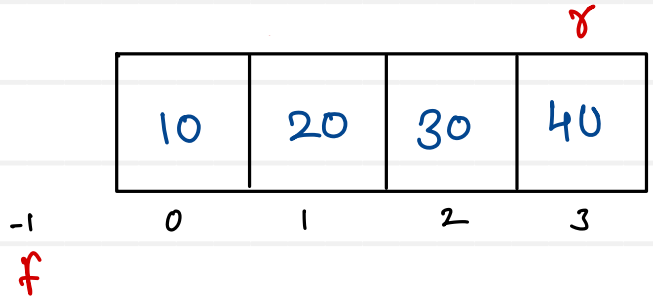


front == rear

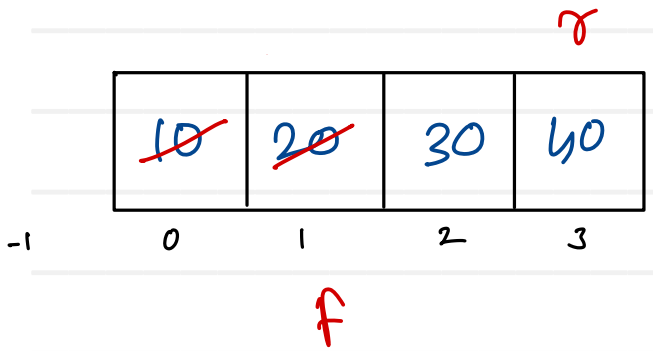


if (front == rear)
front = rear = -1

Full



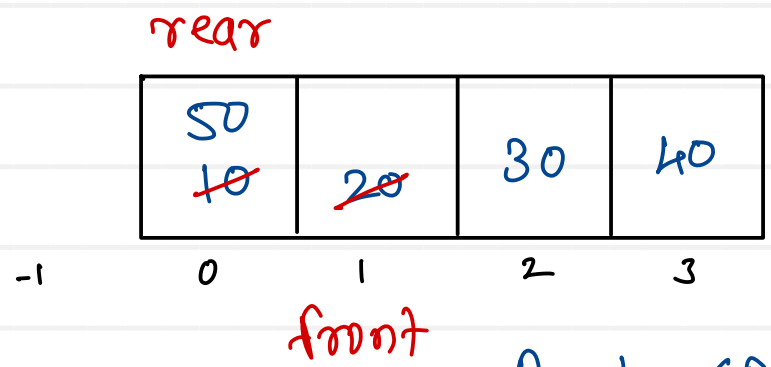
rear == size-1



if linear queue is partially empty (initial few locations are empty), it lead to poor memory utilization.

Circular queue

Capacity=4



front = (front+1) % size

rear = (rear+1) % size

front = rear = -1

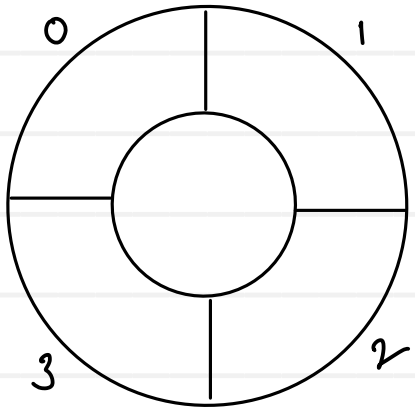
= (-1+1) % 4 = 0

= (0+1) % 4 = 1

= (1+1) % 4 = 2

= (2+1) % 4 = 3

= (3+1) % 4 = 0



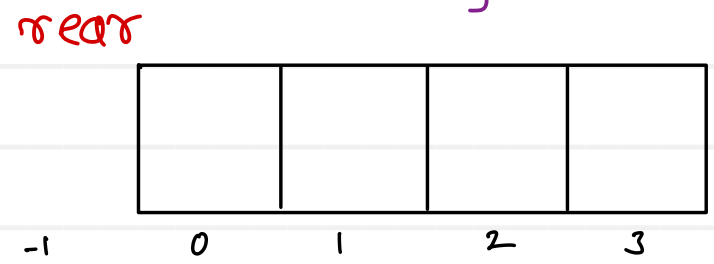
operation:

1. Enqueue
   a. reposition rear (inc)
   b. add data at rear index
2. Dequeue
   a. reposition front (inc)
3. Peek
   a. read/return data of front end
      (front+1) index

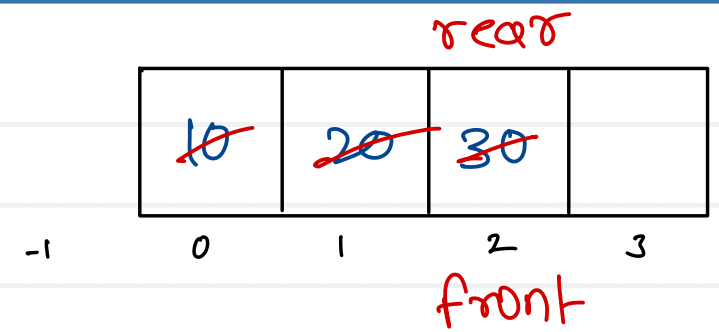
- all operations of queue are performed in O(1) time complexity.

Circular queue - Conditions

Empty

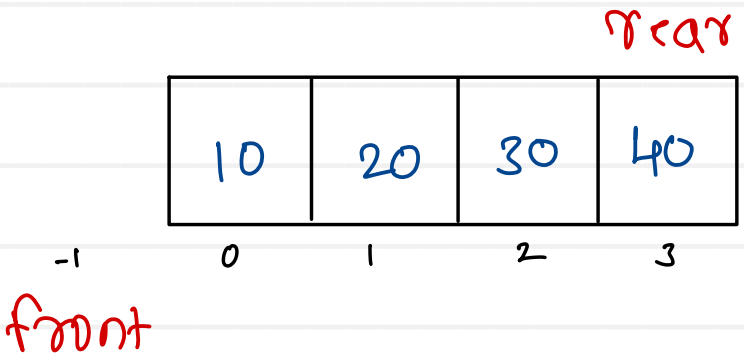


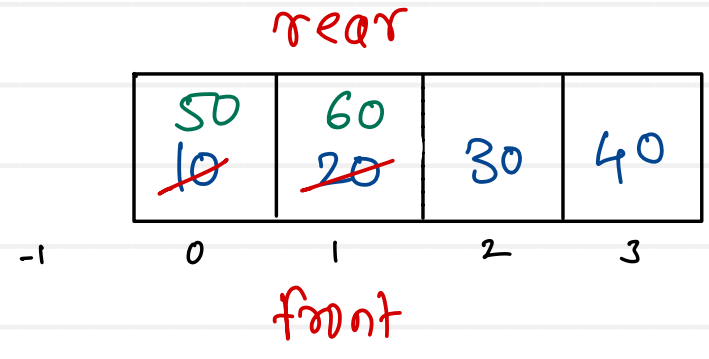
(front == rear && rear == -1)




```
if (front == rear)
    front = rear = -1;
```

Full





(front == -1 && rear == size-1) || (front == rear && rear != -1)

Stack

1. Add first / Delete first
2. Add last / Delete last

Empty : head == null
Full : never be full

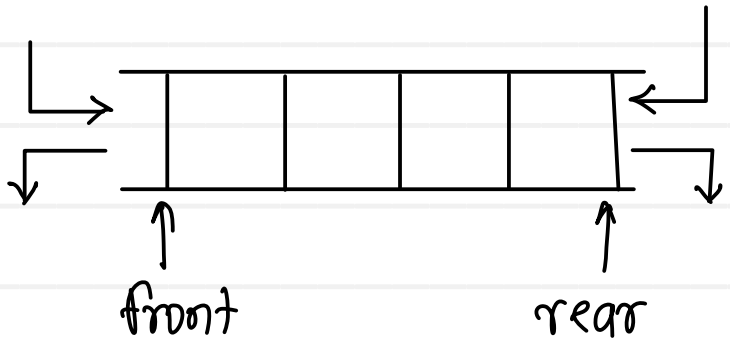
Queue

1. Add first / Delete last
2. Add last / Delete first

Empty : head == null
Full : never be full

Double Ended Queue - Deque

- data can be inserted or removed in a queue from both ends



- Deque do not follow "FIFO" principle

- to implement double ended queue, we can use circular array or linked list

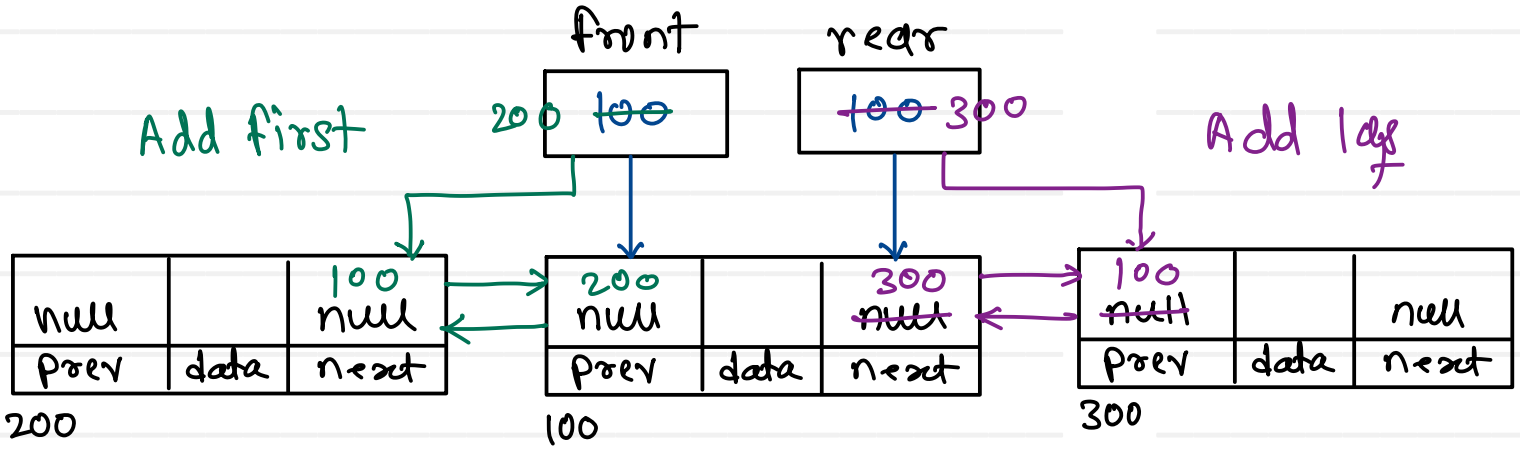
Operations :

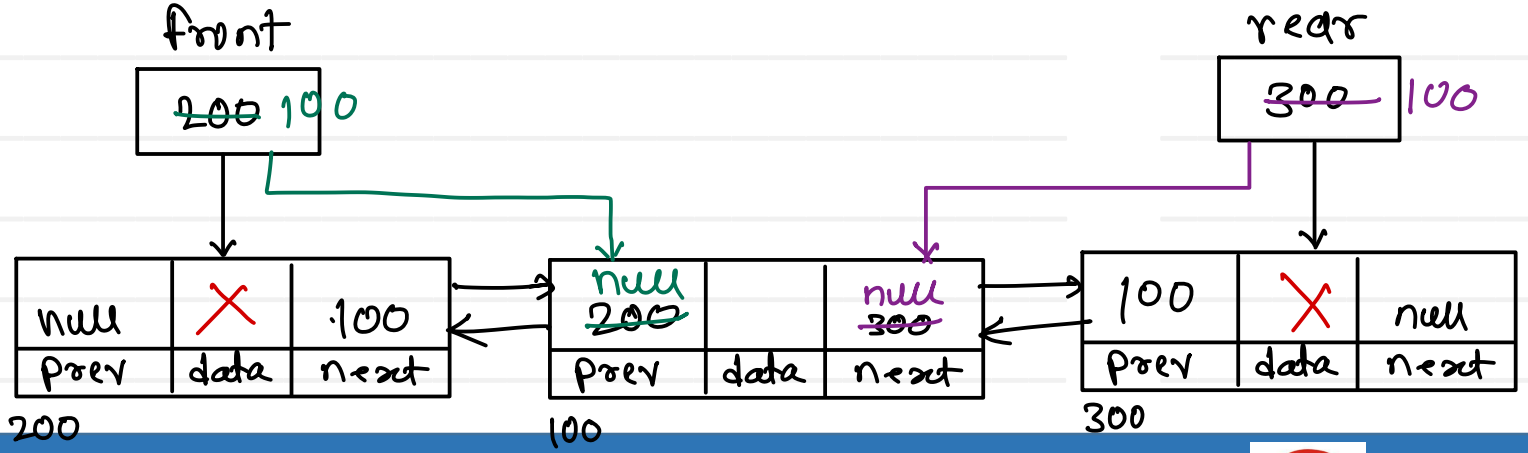
1. pushFront()
2. popFront()
3. peekFront()
4. pushRear()
5. popRear()
6. peekRear()

7. isEmpty()
8. isFull()

head → front : add first / delete first

tail → rear : add last / delete last

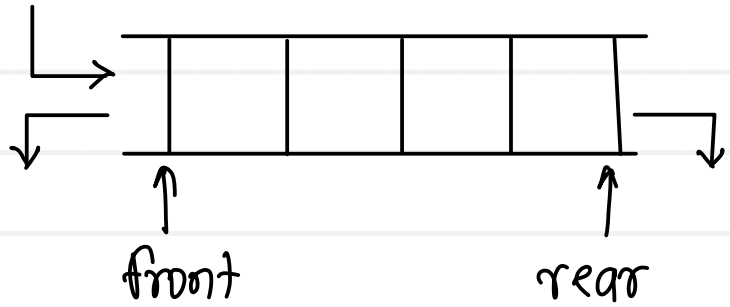




Input Restricted Deque

remove from both ends

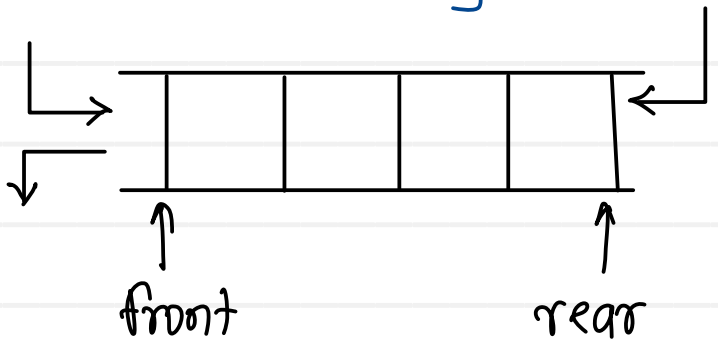
insert from only one end

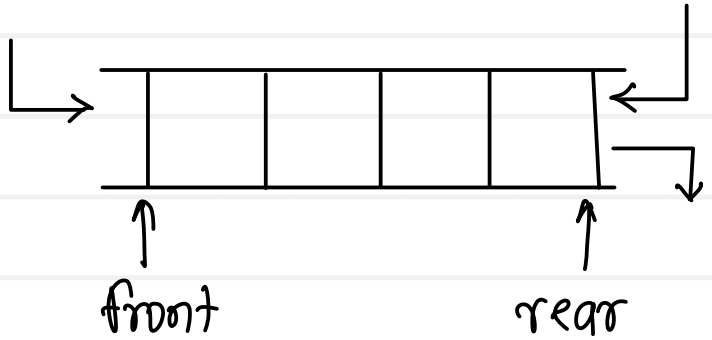


Output Restricted Deque

insert from both ends

remove from only one end





Linked list - Applications

- linked list is a dynamic data structure because it can grow or shrink at runtime.
- Due to this dynamic nature, linked list is used to implement other data structures like
  1. Stack
  2. Queue
  3. Hash table
  4. Graph

Stack

(LIFO)

1. Add First
   delete First
2. Add Last
   delete Last

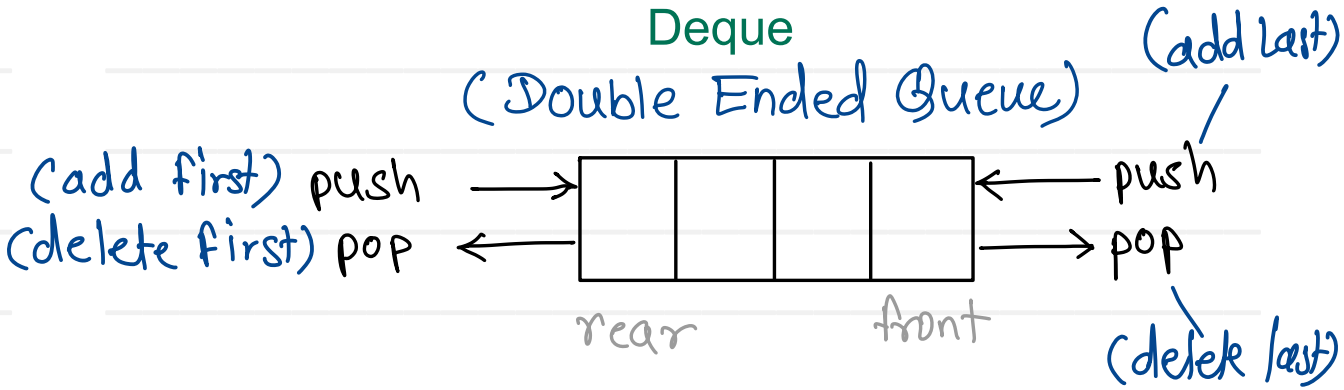
Queue

(FIFO)

1. Add First
   Delete Last
2. Add Last
   Delete First

Deque

(Double Ended Queue)



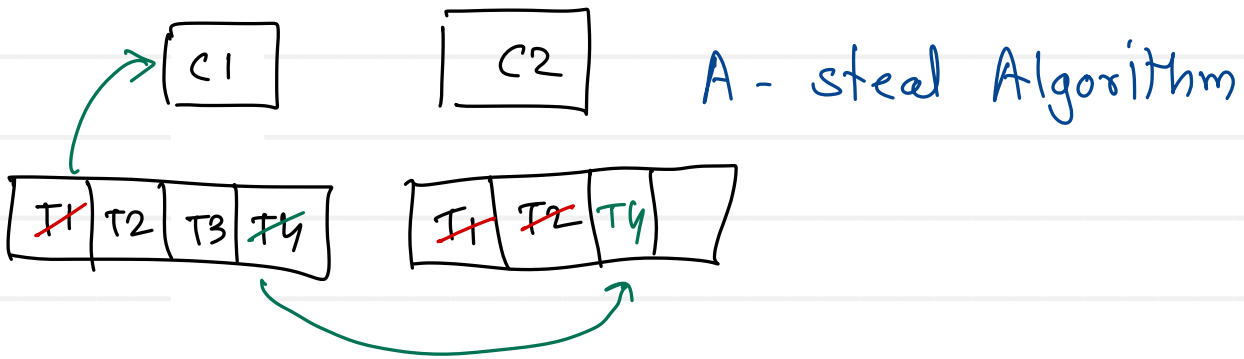
1. Input restricted deque
   - one input (push) is closed
2. Output restricted deque
   - one output (pop) is closed

Applications of Queue

- job queue, ready queue or waiting queue in OS
- jobs submitted to printer [spooler directory]
- call to customer care
- token systems
- in networks to modify common files
- advanced data structures for traversal (BFS - queue)

Applications of Deque

- stack & queue can be efficiently implemented using deque
- can be used to check palindrome
- undo & redo operations in softwares
- Browser history
- per CPU seperate deque is maintained to keep processes



A - steal Algorithm

Stack Applications

- Lexical analysis in compilers
  - parenthesis balancing
- function activation records
  ↳ control the flow program
- traversal in advanced data structures
  DFS - stack
  preorder/inorder/postorder - stack
- Expression conversion & evaluation

Expression : combination of operands & operators

Types :

1. Infix : a + b → human
2. Prefix : + a b
3. Postfix : a b +

→ machine → CPU → ALU

Applications – Stack and Queue

Stack

- Parenthesis balancing
- Expression conversion and evaluation
- Function calls
- Used in advanced data structures for traversing

- **Expression conversion and evaluation:**
  - Infix to postfix
  - Infix to prefix
  - Postfix evaluation
  - Prefix evaluation

Queue

- Jobs submitted to printer
- In Network setups – file access of file server machine is given to First come First serve basis
- Calls are placed on a queue when all operators are busy
- Used in advanced data structures to give efficiency.
- Process waiting queues in OS

Postfix Evaluation

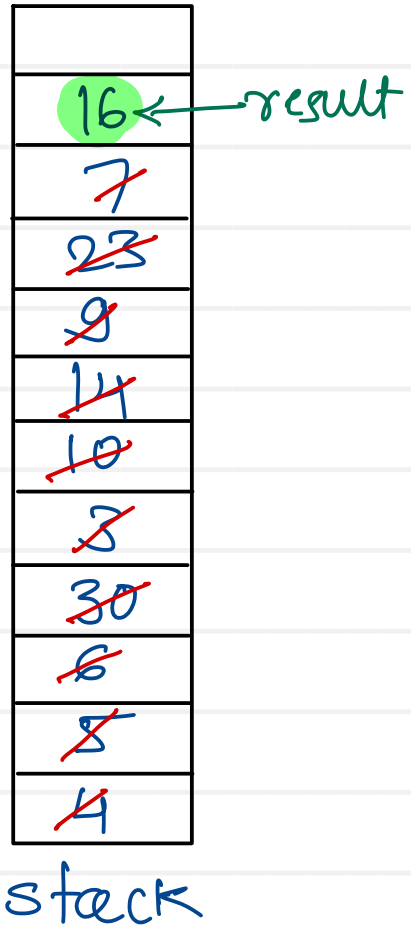
- Process each element of postfix expression from left to right
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. 4 5 6 * 3 / + 9 + 7 -

Postfix evaluation

Postfix expression : 4 5 6 * 3 / + 9 + 7 -

l ⟶ r

5. 23 − 7 = 16
4. 14 + 9 = 23
3. 4 + 10 = 14
2. 30 / 3 = 10
1. 5 * 6 = 30



Prefix Evaluation

- Process each element of prefix expression from right to left
- If element is operand
  - Push it on a stack
- If element is operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Perform current element (Operator) operation between Op1 and Op2
  - Again push back result onto the stack
- When single value will remain on stack, it is final result
- e.g. - + + 4 / * 5 6 3 9 7

Prefix evaluation

Prefix expression : - + + 4 / * 5 6 3 9 7

l ← r

23 − 7 = 16

14 + 9 = 23

4 + 10 = 14

30 / 3 = 10

5 * 6 = 30

Infix to Postfix Conversion

- Process each element of infix expression from left to right
- If element is Operand
  - Append it to the postfix expression
- If element is Operator
  - If priority of topmost element (Operator) of stack is greater or equal to current element (Operator), pop topmost element from stack and append it to postfix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the postfix expression
- e.g. a * b / c * d + e – f * h + i

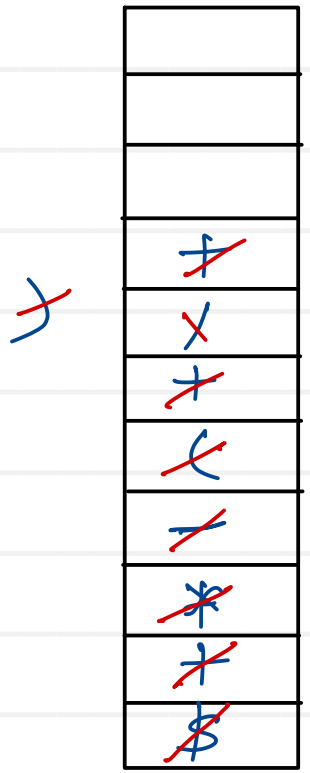
( )
$
* / %
+ –

Infix to Postfix conversion

Infix expression : 1 $ 9 + 3 * 4 - (6 + 8 / 2) + 7

l ⟶ r

Postfix expression : 19$34*+682/+-7+



Infix to Prefix Conversion

- Process each element of infix expression from right to left
- If element is Operand
  - Append it to the prefix expression
- If element is Operator
  - If priority of topmost element of stack is greater than current element (Operator), pop topmost element from stack and append it to prefix expression
  - Repeat above step if required
  - Push element on stack
- Pop all remaining elements (Operators) from stack one by one and append them into the prefix expression
- Reverse prefix expression
- e.g. a * b / c * d + e – f * h + i

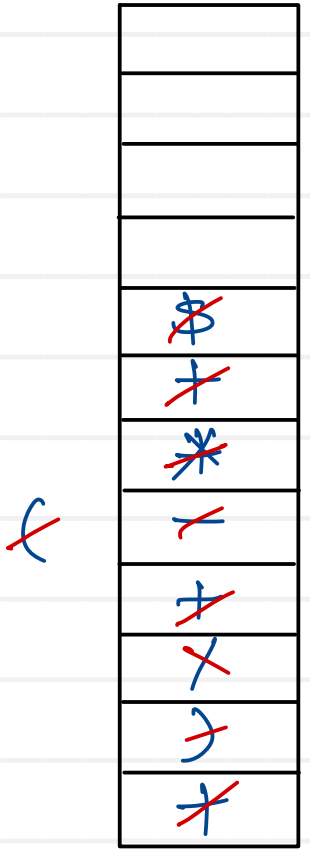
Infix to Prefix conversion

Infix expression : 1 $ 9 + 3 * 4 - (6 + 8 / 2) + 7

l ← r

Expression : 7 2 8 / 6 + 4 3 * 9 1 $ + − +

Prefix expression : + − + $ 1 9 * 3 4 + 6 / 8 2 7



Prefix to Postfix

- Process each element of prefix expression from right to left
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op1 – first popped element
    - Op2 – second popped element
  - Form a string by concatenating Op1, Op2 and Opr (element)
  - String = “Op1+Op2+Opr”, push back on to the stack
- Repeat above two steps until end of prefix expression.
- Last remaining on the stack is postfix expression
- e.g. * + a b – c d

Postfix to Infix

- Process each element of postfix expression from left to right
- If element is an Operand
  - Push it on to the stack
- If element is an Operator
  - Pop two elements (Operands) from stack, in such a way that
    - Op2 – first popped element
    - Op1 – second popped element
  - Form a string by concatenating Op1, Opr (element) and Op2
  - String = “Op1+Opr+Op2”, push back on to the stack
- Repeat above two steps until end of postfix expression.
- Last remaining on the stack is infix expression
- E.g. a b c - + d e – f g – h + / *

Thank you!!!

Devendra Dhande

devendra.dhande@sunbeaminfo.com