Sunbeam Institute of Information Technology
Pune and Karad

Algorithms and Data structures

Trainer - Devendra Dhande

Email – devendra.dhande@sunbeaminfo.com

Data Structure

- organising data inside memory for efficient processing along with operations like add, delete, search, etc which can be performed on data.
- eg stack - push/pop/peek

- data structures are used to achieve
  - Abstraction
    Abstract Data Types (ADT)
  - Reusability
  - Efficiency
    time
    space

Types of data structures

Linear data structures
(Basic)

- data is organised sequentially/ linearly

- data can be accessed sequentially

e.g. Array, structure/class, stack, queue, lined list

Non linear data structures
(Advanced)

- data is organised in multiple levels (hierarchy)

- data can not be accessed sequentially

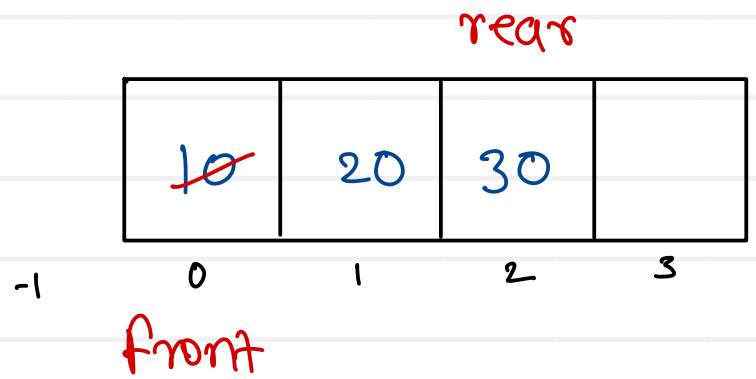
e.g. Tree, Graph

Hash table

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

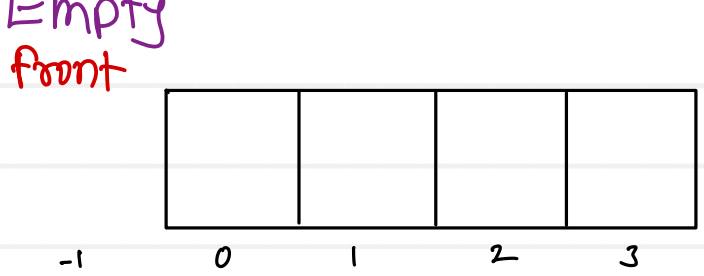


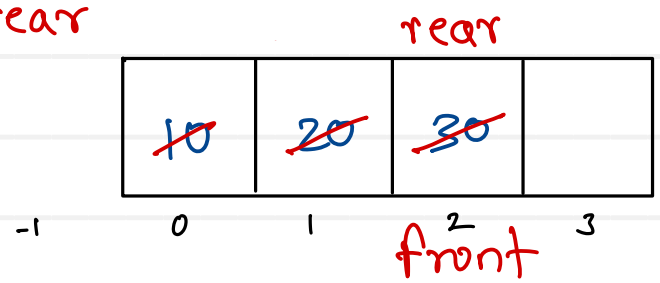
Operations :

1. Push / enqueue / add / insert :
   a. reposition rear (inc)
   b. add value at rear index
2. Pop / dequeue / delete / remove :
   a. reposition front (inc)
3. Peek :
   a. read / return data of front+1 index

Linear queue - Conditions

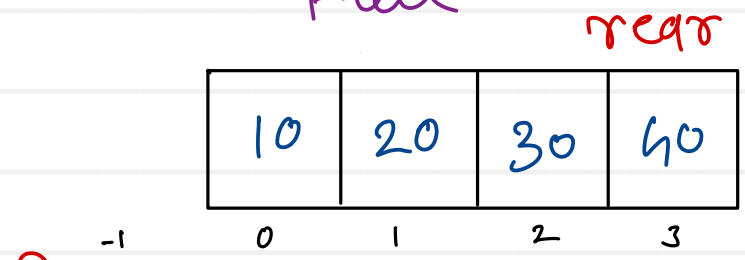
Empty



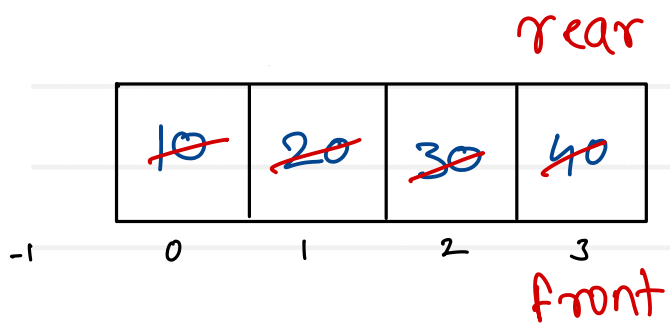


front == rear

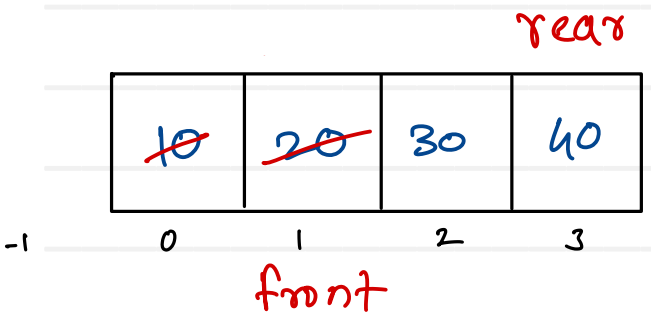
Full



rear == size-1

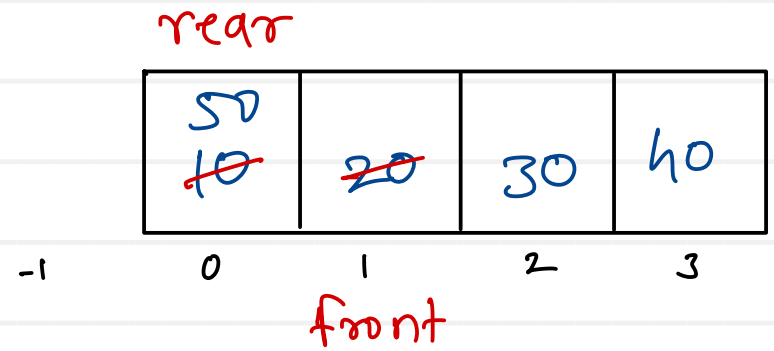



```
pop():
  if(front==rear)
    front=rear=-1;
```

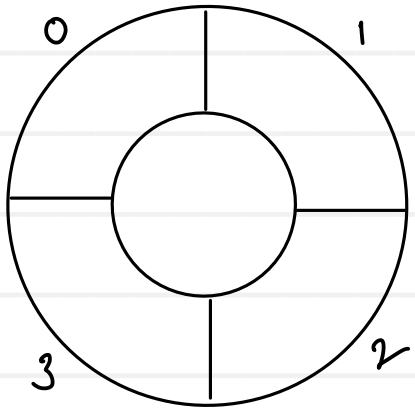


if few initial locations of queue are vacant, we can not reuse them till queue is not empty, this leads to poor memory utilization

Circular queue



size = 4



front = (front+1) % size
rear = (rear+1) % size

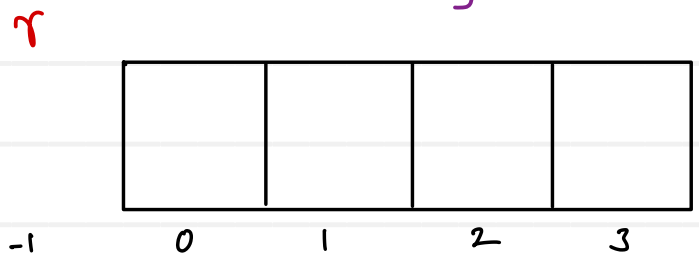
front = rear = -1
= (-1+1) % 4 = 0
= (0+1) % 4 = 1
= (1+1) % 4 = 2
= (2+1) % 4 = 3
= (3+1) % 4 = 0

Operations :

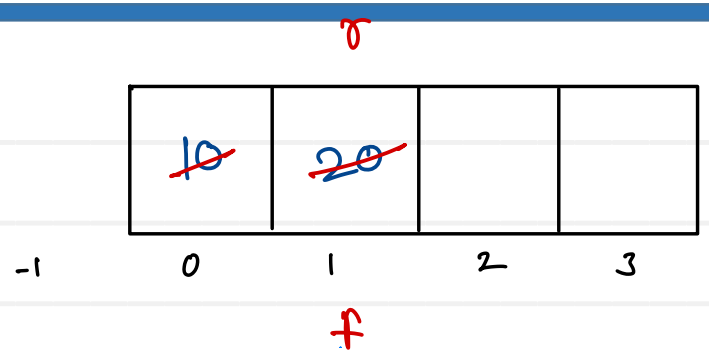
1. Push / enqueue / add / insert :
   a. reposition rear (inc)
   b. add value at rear index
2. Pop / dequeue / delete / remove :
   a. reposition front (inc)
3. Peek :
   a. read / return data of front+1 index

Circular queue - Conditions

Empty

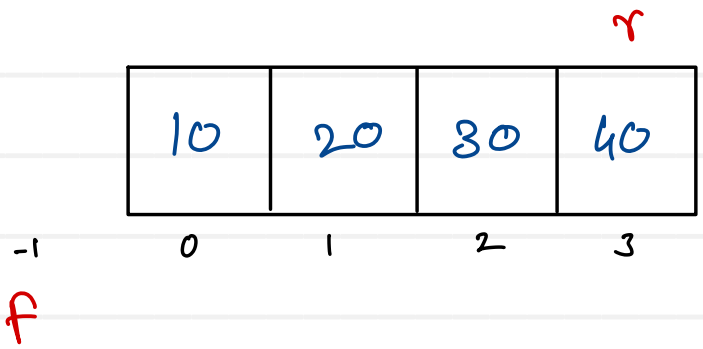


front == rear && rear == -1

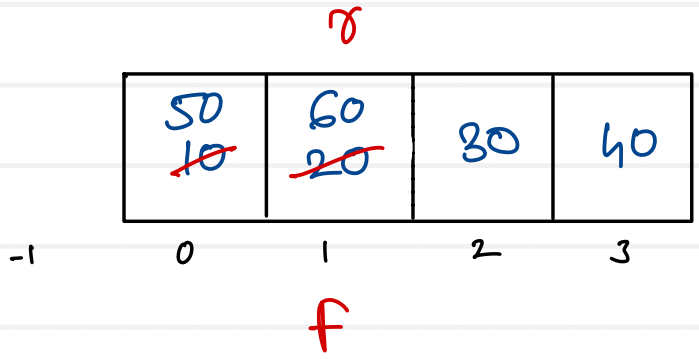


pop() : if(front == rear)
front = rear = -1

Full



front == -1 && rear == size-1



front == rear && rear != -1

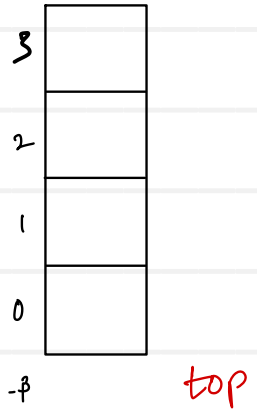
Stack

- 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”

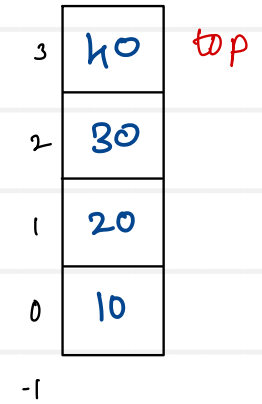
capacity = 4

Operations :

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



top == -1



top == size-1

Ascending stack

top = -1

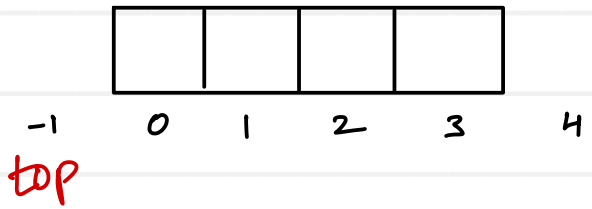
push : top++
arr[top] = value

pop : top--

peek : arr[top]

Empty : top == -1

Full : top == size-1



Descending stack

top = size

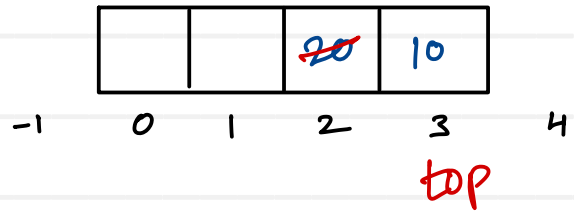
push : top--
arr[top] = value

pop : top++

peek : arr[top]

Empty : top == size

Full : top == 0



Applications – Stack and Queue

Stack

- Parenthesis balancing — lexical analysis
- 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 [spooler directory]
- 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

Expression: combination of operands & operators

1. Infix : a+b (human)
2. Prefix : +ab
3. Postfix : ab+

} (computer/machine) → CPU → ALU

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 -

left ⟶ right

Result = 16

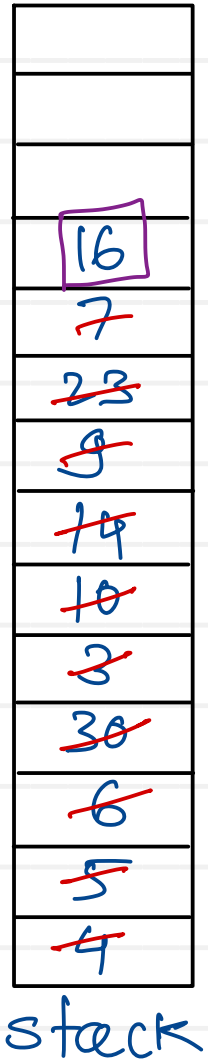
⑤ 23 − 7 = 16

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

left ← right

23 – 7 = 16

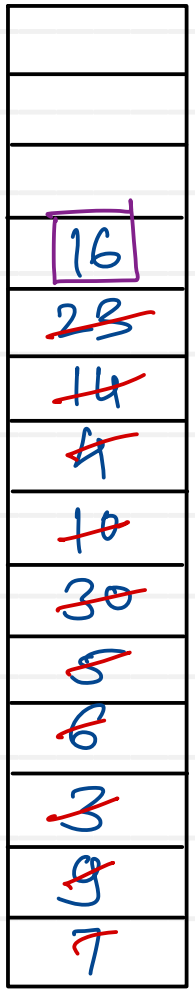
14 + 9 = 23

4 + 10 = 14

30 / 3 = 10

5 * 6 = 30

Result = 16



infix : 11 + 22

prefix : + 11 22

postfix : "11 22 +"

```
String arr[]
    = postfix.split();
```

arr[] = {"11", "22", "+"};

Infix to Postfix Conversion

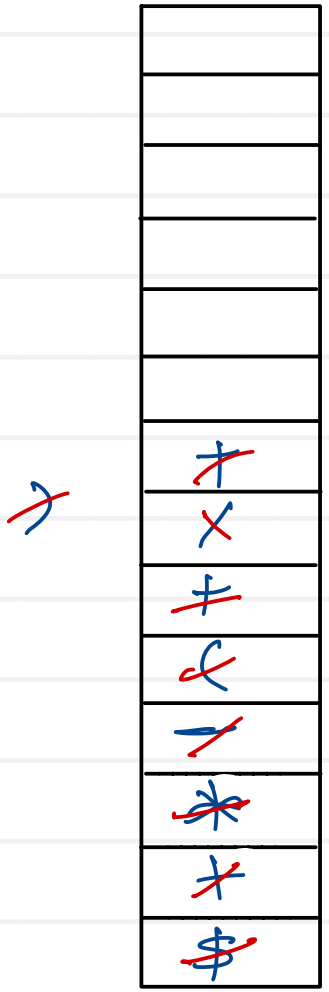
- Process each element of infix expression from left to right
- If element is Operand
  - Append it to the postfix expression
- If element is Operator
  - If priority of topmost element (Operator) of stack is greater or equal to current element (Operator), pop topmost element from stack and append it 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

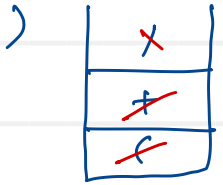
Infix to Postfix conversion

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

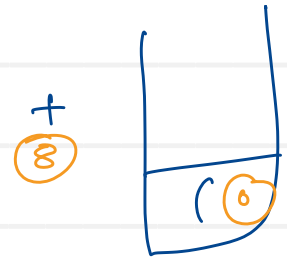
left → right

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





while(st.peek() != '(')
expr.append(st.pop());
st.pop();



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

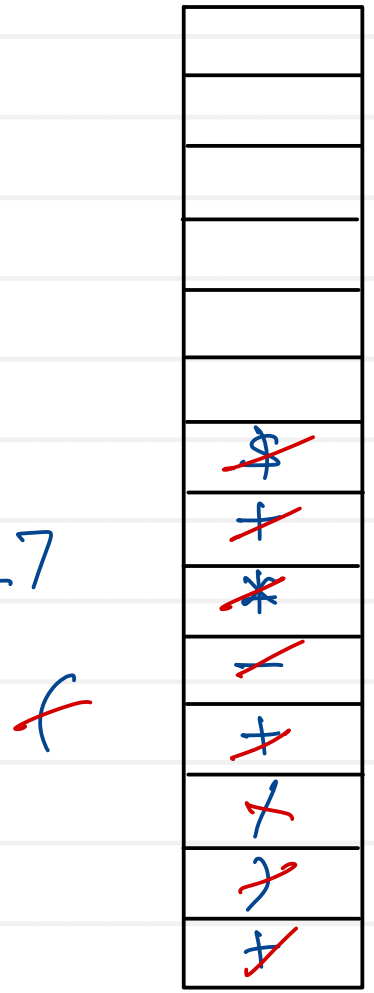
Infix to Prefix conversion

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

left ← right

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