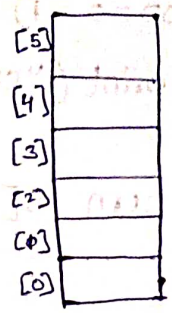
14/10/20

Data Structures

Stack Operations



Push:

→ Let N be size of array S

→ Let Top is a pointer pointing to topmost element of the stack

→ Initially when stack is ~~em~~ empty, value of top is -1.

→ To push an element into stack, first increament top and store element into position pointed by top

```
void PUSH (S, N, Top, a)
                    ↳ element to be pushed
{
    if (Top == N-1) // Assuming array indices start from 1.
    {
        printf("stack overflow");
        return;
    }
    else
    {
        Top = Top + 1;
        S[TOP] = a;
    }
}
```

POP:

(i) Delek the element pointed by Top and decrement Top by 1.

```
x = S[Top];
Top = Top - 1;
return x;
```

(ii) If Top = -1, then we say stack underflow has occured and we cannot perform deletion in this case.

```
int Pop(S, Top)
{
    if(Top == -1)
        print("Stack Underflow");
    else
        return S[Top--];
}
```

Note: Popping doesn't remove value stored. It just decreases value of top.

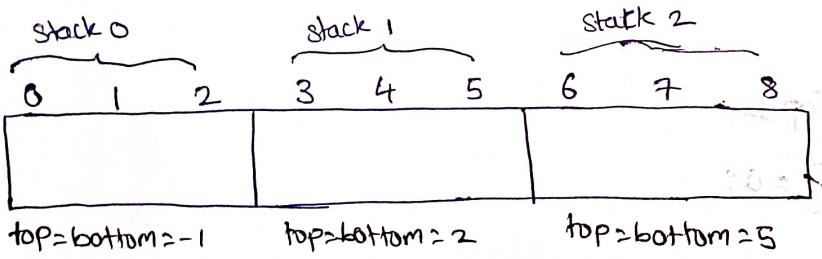
Multiple Stack Implementation:

Let us consider an array of size m with 'n' stacks.

For example if

$m = 9,\ n = 3$

size of each stack $= \frac{m}{n} = \frac{9}{3} = 3$



Let top[i] & bottom[i] denote top & bottom values of each stack.

bottom values are fixed and top values change.

For $0 \le i < n$

$$top[i] = bottom[i] = i * \left\lfloor \frac{m}{n} \right\rfloor - 1$$

i	top[i]	bottom[i]
0	-1	-1
1	2	2
2	5	5

Also we store bottom[3] = m-1 → used to detect stack overflow of last stack.

↳ Stack number

```
void PUSH(A, i, a)
{
    if(top[i] == bottom[i+1])
        printf("Stack overflow");
    else
        A[top[i]] = a;
}
int Pop (A, i)
{
    if (top[i] == bottom[i])
        print ("Stack underflow);
    else
        return A[top[i]--];
}
```

Queue:

→ Queue is a linear DS.

→ Each queue is associated with two pointers front & rear

Deletion is done at front end

Insertion is done at rare end.

→ Initially, front = -1
rear = -1

~~Enqueue (Insertion)~~ :

~~rear = rear ++;~~

~~Q[rear] = x;~~

~~if (front == -1) // front = -1 indicates Queue is empty.~~

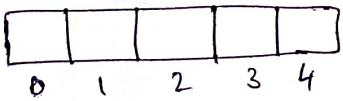
~~front = 0~~

front ↑ rear ↑ → size of array

```
ENQUEUE (Q, F, R, x, MAX)
{
    if (R==MAX-1)
    {
        printf("Q overflow");
        return;
    }
    else
    {
        R=R+1;
        Q[R]=x;
        if (F==-1)
            F=0;
    }
}
```

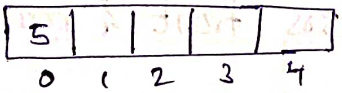
Consider below example

(i) front = -1, rear = -1



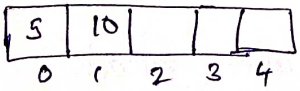
(ii) Insert 5

front = 0; rear = 0



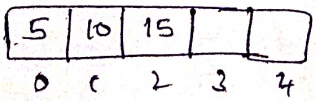
(iii) Insert 10

front = 0; rear = 1



(iv) Insert 15

front = 0 rear = 2

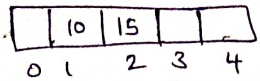


(v) ~~Delete~~

```
int Dequeue(Q, F, R, MAX)
{
    if (F==R+1 || F==-1)
        Print("Queue Underflow");
    else
        return Q[F++];
}
```

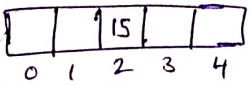
(v) Delete

front = 1 rear = 2



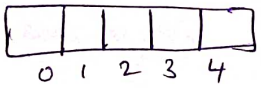
(vi) Delete

front = 2 rear 2



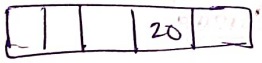
(vii) Delete

front = 3 rear = 2



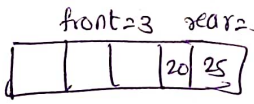
(viii) Insert 20

front = 3 rear = 3



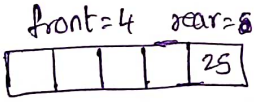
(ix) Insert 25

front = 3 rear = 4



(x) Delete

front = 4 rear = 4



(xi) Delete

front = 5 rear = 4

Note :

→ At any point, no of elements in queue is given by

rear - front + 1

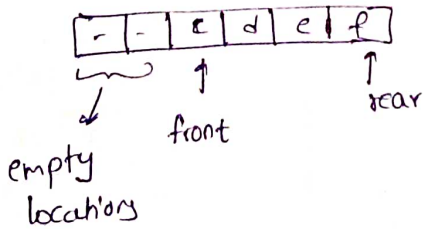
→

```
bool isEmpty()
{
    if ( front == -1 || front == rear+1)
        return true;
    else
        return false;
}

bool isFull()
{
    if ( rear == MAX-1) // MAX is size of array
        return true;
    else
        return false;
}
```

Drawback in array implementation of Queue:

→ Consider the situation where rear is at last position and front is not at 0th position.

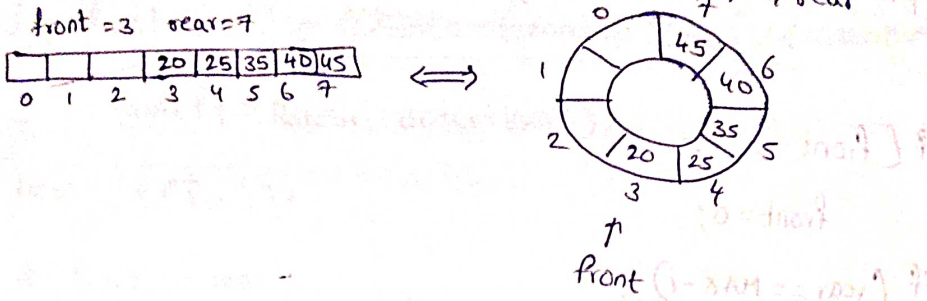


Even though there are empty locations, we cannot perform insertion.

This drawback is overcome using circular queue.

Circular Queue:

→ we logically treat array as circular array.



→ when front or rear is MAX-1, then rather that increamenting we set them to '0'.

→ If Queue is full then

front = (rear+1) %. MAX

→ If Queue is empty then

(front = -1) or front = (rear+1) %. MAX

Common Condition

Note:

→ At any point, if front = rear then there is exactly one element in the queue.

```
int isEmpty()
{
    if (front == -1)
        return 1;
    else
        return 0;
}
```

```
int isFull
{
    if ((front == 0 && rear == MAX-1) || (front = rear+1))
        return 1;
    else
        return 0;
}
```

```
void Enqueue (int item)
{
    if (isFull())
        print ("Queue Overflow");
    else
    {
        if (front == -1)
            front = 0;
        if (rear == MAX-1)
            rear = 0;
        else
            rear = rear+1;
        Q[rear] = item;
    }
}
```

Method 2:

```
void Enqueue2 (int item)
{
    if (front == (rear+1) % MAX)
        printf("Queue Overflow");
    else
    {
        if (front == -1)
            front = 0
        rear = (rear+1) % MAX
        Q[rear] = item;
    }
}
```

```
int dequeue()
{
    int item;
    if ((front==-1))
        printf("Queue underflow");
    item = q[front];
    if(front==rear)
    {
        front = -1;
        rear = -1;
    }
    else
        front= (front+1)%MAX
    return item;
}
```

This ensures that whever Queue become empty front & rear will be -1 but not front = rear+1

```
int isEmpty()
{
    if(front==-1)
        return 1;
    else
        return 0;
}
```

```
int isFull()
{
    if (front==(rear+1)%MAX)
        return 1;
    else
        return 0;
}
```

Deque (Double ended queue) (not needed for gate)

~~(i) Ins~~

Here insertion & deletion can be done at both the ends

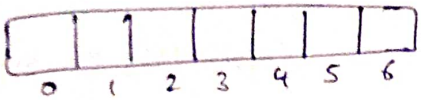
(i) insertion at rear end → Inc rear & insert
(ii) insertion at front end → Dec front & insert
(iii) Deletion at rear end → delete & dec rear
(iv) Deletion at front end → delete & inc front

Also here we treat array as circular array

Eg :

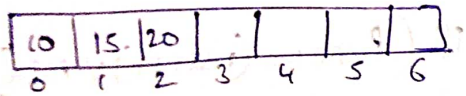
(i) Empty queue

front = -1 rear = -1



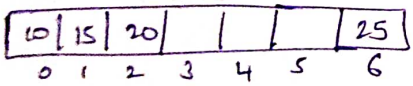
(ii) insert 10, 15, 20 at rear end

front = 0 rear = 2



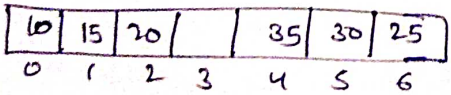
(iii) Insert 25 at rear end → so dec front & insert.

front = 6 rear = 2



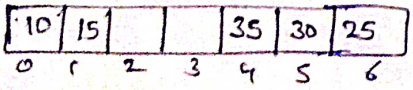
(iv) Insert 30, 35 at front end

front = 4 rear = 2



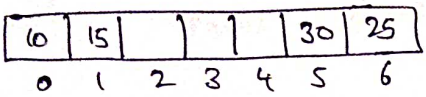
(v) Delete from rear end

front = 4 rear = 1

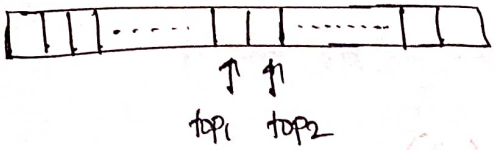


(vi) Delete from front end

front = 5 rear = 1



P/3

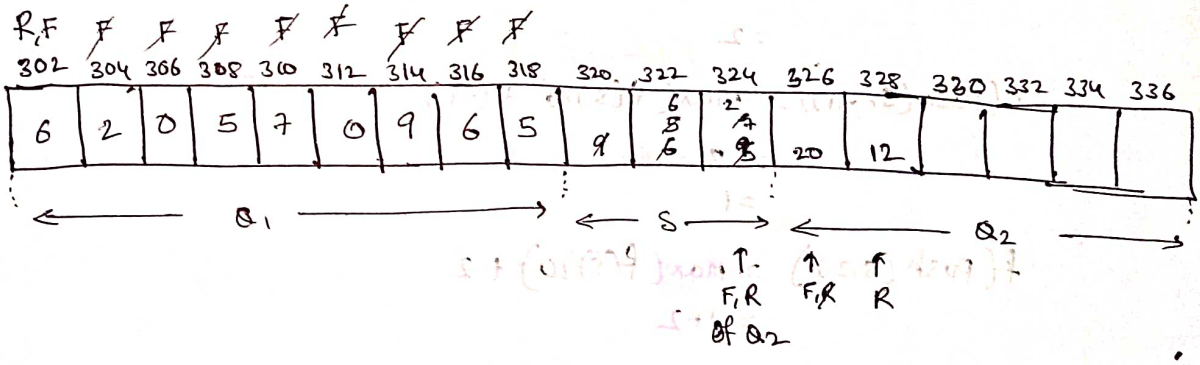


Here stack is full

$\therefore top1 = top2 - 1$

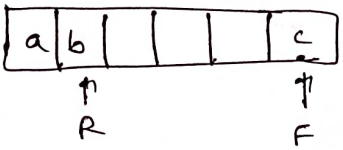
15/10/20

P/18

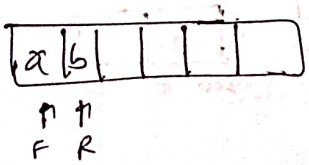


$\therefore loc(6) = 322 \quad loc(20) = 326$

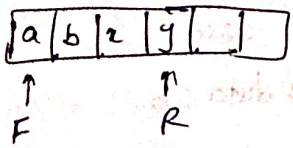
P/19



Delete



Insert x 4 y

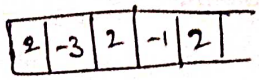


$\therefore 144$

P14 $f(\phi) = 0$

↓

empty stack



$f(Push(S,2)) = max(f(S),0) + 2$

$= max(0,0) + 2$

$= 2$

$f(Push(S,-3)) = max(f(S),0) + (-3)$

$= max(2,0) - 3$

$= -1$

$f(Push(S,2)) = max(f(S),0) + 2$

$= 0 + 2$

$= 2$

$f(Push(S,-1)) = max(f(S),0) + (-1)$

$= 2 - 1$

$= 1$

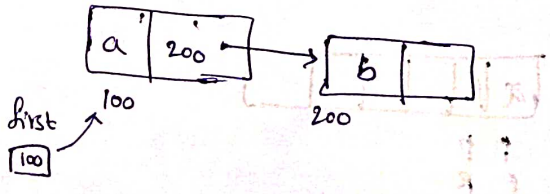
$f(Push(S,2)) = max(f(S),0) + 2$

$= 1 + 2$

$= 3$

Linked List:

```
Struct node
{
    int data;
    struct node *link;
};
struct node *first;
struct node s1,s2;
first = &s1;
s1.link = &s2;
```

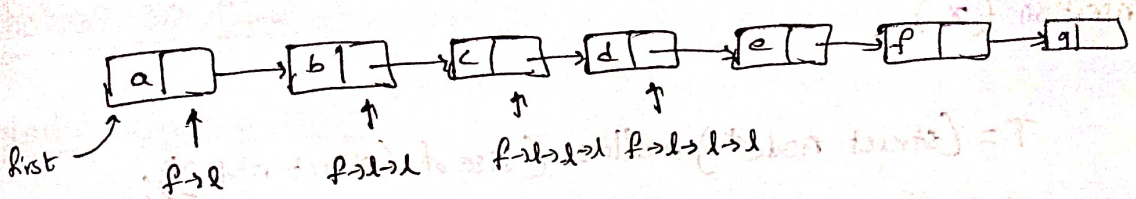


first → data : a

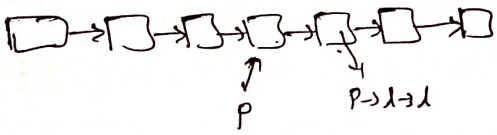
frist → link : 200

first → link → data : b

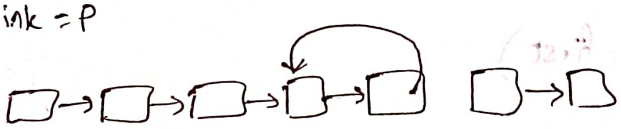
P/1



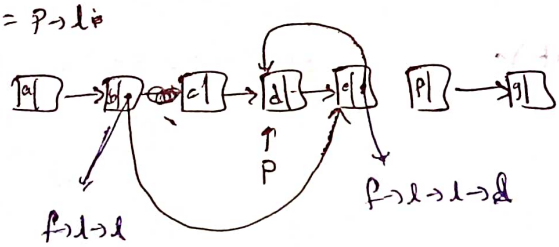
∴ (i) P = f→l→l→l→l;



(ii) P→link→link = P

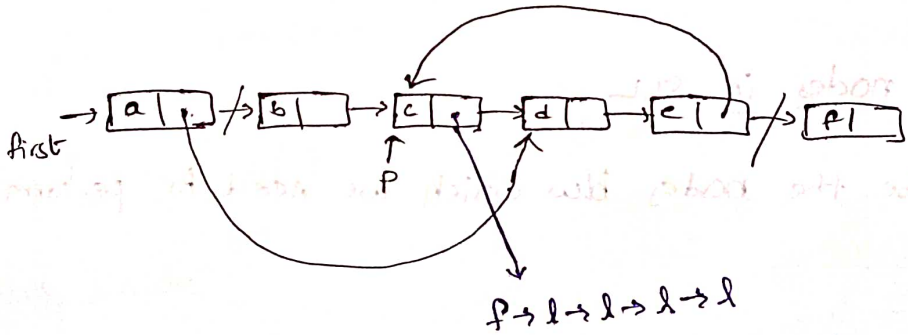


(iii) f→l→l = P→l



∴ f→l→l→l→d = d

P/2



Insertion

at beginning of SLL:

Step 1: Create node 'T'

Step 2: Store data in 'T'

Step 3: Update link field of T i.e., T→link = first.

Step 4: Update start pointer. ie, ~~first~~ first = T

Insertion (x)

```
{
    T = (struct node*) malloc (sizeof (struct node));
    T->data = x;
    T->link = first;
    first = T;
}
```

at the end in SLL:

```
insert_at_end (x, first)
{
    P = first;
    while (P->next);
        P = P->next;
    T = getNode();
    T->data = x;
    T->next = NULL;
    P->next = T;
}
```

inserting blw nodes in SLL

Let P & Q be the nodes blw which we need to perform the insertion.

```
insert_between (x, P, Q, first)
{
    temp = first;
    while (temp != P)
        temp = temp->next;
    T = getNode();
    T->data = x
    T->link = P->link;
    P->link = T;
}
```

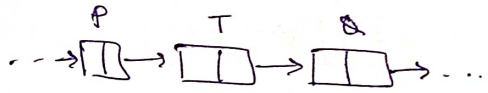
Deletion on SLL

Deleting starting node:

~~temp = first;~~

```
temp = first → next;
free (first);
first = temp;
```

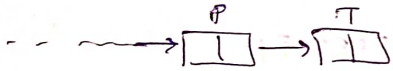
Deletion in blw



~~Consit~~ Consider deleting T

```
P→link = T→link;
free(T);
```

Deletion at end:




```
P→link = NULL;
free(T);
```

Doubly Linked List:

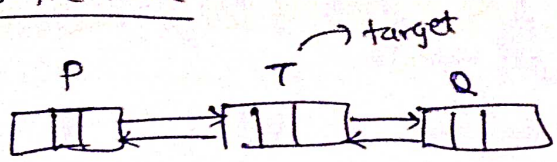
```
struct Dnode
{
    struct Dnode *lptr;
    int data;
    struct Dnode *rptr;
};
```

Insertion:

at beginning:

```
T = getNode();
T→data = x;
T→rptr = start; T→lptr = NULL;
start→lptr = T;
~~start~~ start = T;
```

Deletion btw the node

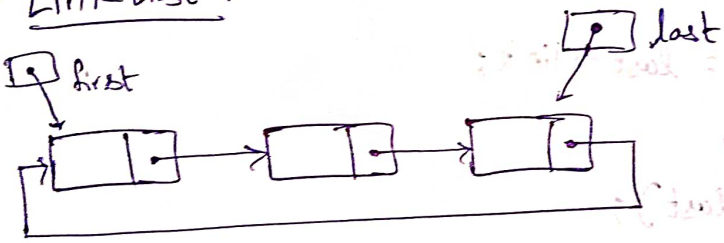



```
P→rptr = T→rptr;
Q→lptr = T→lptr;
free(T);
```

Deletion in the end


```
P→rptr = NULL;
free(T);
```

Circular Link list:



Insertion:

at beginning:

```
T→ data = x;
T→ link = first;
last→link = T;
first = T;
```

at ending:

```
T→data = x;
T→ link = first;
last→link = T;
last = T;
```

Insertion blw nodes:

Same as SLL

Deletion:

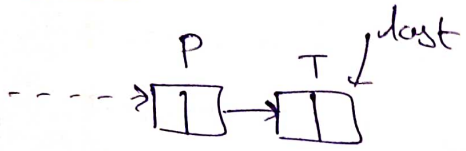
at beginning:

~~T = first;~~
~~first->link = T~~
~~first = first->link;~~
~~free(T);~~
~~last = first;~~

(or)

last->link = first->link
free(first)
first = last->link;

at ending:



P->link = last->link;
~~last = P~~
free(last);
last = P;