-STACK & QUEUE-

- Kinds of List

1.) Stack
2.) Queue
3.) Deque (double-ended queue)

* Stack

"Last in, First Out"

- special kind of list in which all insertions & deletions take place at one place: top
- pushdown list

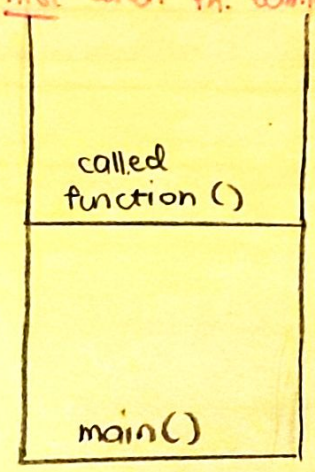
* Operations

1.) PUSH (x, S) - insert element x at the top of the stack
2.) POP (S) - delete top element of stack S
3.) TOP (S) - returns element at the top of stack S
4.) EMPTY (S) - returns TRUE if stack is empty } otherwise 0
5.) FULL (S) - returns TRUE if stack is full
6.) INITIALIZE (S) & MAKENULL(S)

* Implementations

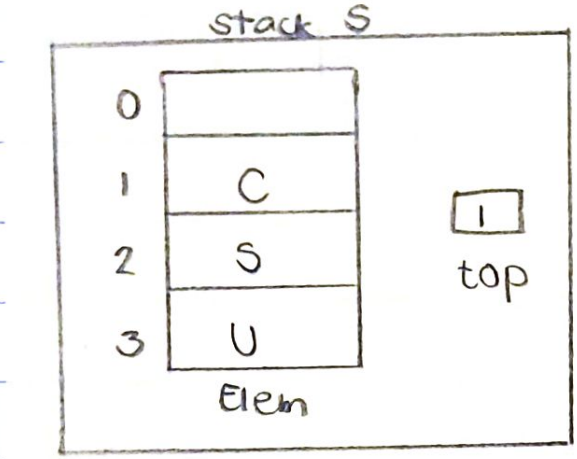
1.) Array
2.) Linked List
3.) CB List

* fn() A.R. is added last but will be removed first when fn. terminates.



Execution Stack

-ARRAY IMPLEMENTATION-
(STACK)

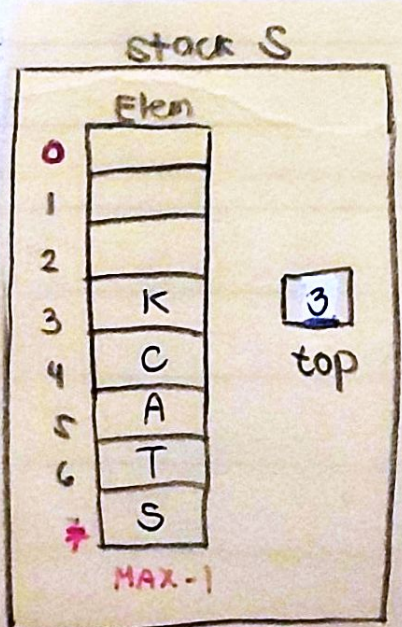


- **Definition**

```
#define MAX 4
typedef struct {
    char Elem [MAX];
    int top;
} Stack;
```

*** Two Views**

- **View 1** - Stack will grow from MAX-1 to 0.

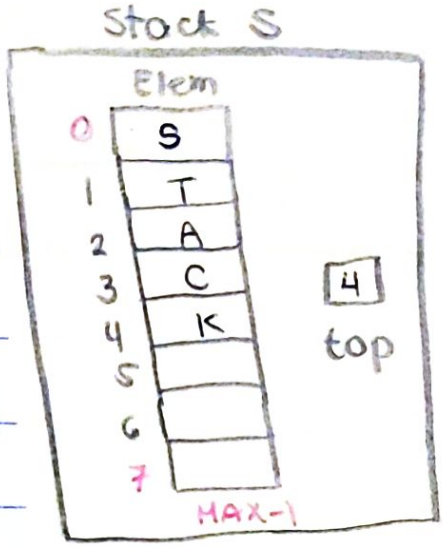


- scenario:
  1.) Empty stack → top is MAX
  2.) Full stack → top is 0
  3.) PUSH (x, S) → top decreases
  4.) POP (S) → top increases

[7] = MAX

* this new starts with top at MAX & will increase/decrease from there.

• VIEW 2 - stack will grow from 0 to MAX-1 ↓



• scenario:

1.) Empty Stack → top is -1

2.) Full Stack → top is MAX-1

3.) PUSH (X, S) → top increases

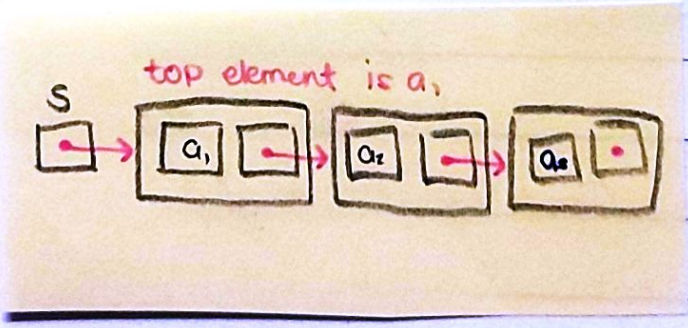
4.) POP (S) → top decreases

* top starts at 0 & increases/decreases from there.

* the whole point of stack is to AVOID SHIFTING!

- LINKED LIST IMPLEMENTATION - (STACK)

* Top element is stored in the cell pointed to by Stack pointer S.



- This orientation is O(1) because it does not require traversing to access the top element. So, all insertion & deletion (PUSH, POP), happens at the START of the list.

- STACK OPERATIONS -

(MAX-1 to 0)

(ARRAY LIST)

// initialize (S)

```
void initialize (stack *S){
    S→ top = MAX;
}
```

Definition:

```
typedef struct {
    char Elem [MAX];
    int top;
} Stack;
```

// PUSH (x, S)

```
void PUSH (stack *S, char elem) {
    if (isFull(*S) == 0) {
        S→ top--;
        S→ Elem [S→ top] = elem;
    }
}
```

* assuming all these can call other stack operation fns!

* increase top & insert element

// POP (S)

```
void POP (Stack *S) {
    if( isEmpty (*S) ==0){
        S→ top ++;
    }
}
```

* POP the stack by decrementing top.
no need to return anything unless specifically required.

// TOP (S)

```
char TOP (stack *S) {
    return (isEmpty(*S)==0)? S→ Elem [S→ top] : '\0';
}
```

* if stack is empty, return a sentinel char value!

NO:

DATE:

//isEmpty()

*should be BEFORE all other fns. since you use it later

```
int isEmpty (Stack S){
    return (S.top == MAX)? 1 : 0;
}
```

* return 0 if stack IS NOT full.

//isFull()

```
int isFull (Stack S) {
    return (S.top == -1)? 1 : 0;
}
```

* return 1 if stack IS FULL. since stack is from MAX-1 to 0, going past 0 means stack is full.

//displayStack()

```
void displayStack (Stack S){
    Stack temp;
    initialize (&temp);

    if(isEmpty(S) == 0){
        while (isEmpty(S) == 0){
            printf ("%c", top(S));
            push (&temp, top(S));
            pop(&S);
        }
    }
}
```

*since its just pass by copy, the actual OG stack is left untouched after dumping its contents in temp!

(1.) Declare & initialize temp variable.

- this will hold the values of stackS after we "dump" the contents since we CAN'T traverse.

(2) Check if stack is empty.

(3) while its not empty, print the top element, one by one.

- store it inside temp after printing.
- Decrement S (top) to move to next element.

push (&temp, top(s));

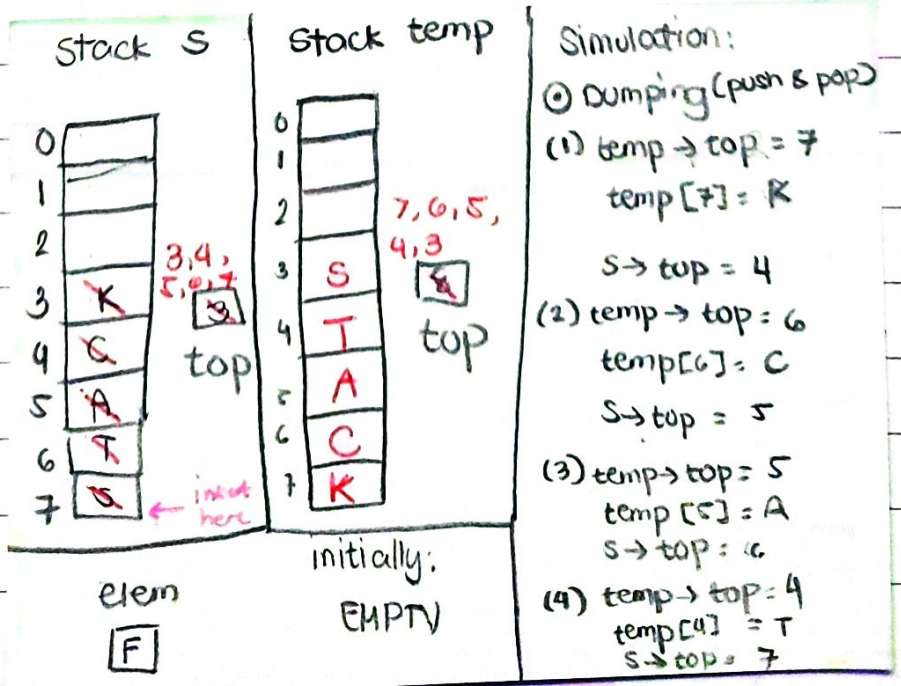
DATE:

// insertBottom () - insert new element at bottom of stack

```
void insertBottom (Stack *S, char elem) {
    Stack temp;
    initialize (&temp);

    if (isFull (*S) == 0) {
        while (isEmpty (*S) == 0) {
            push (&temp, top (*S));      } DUMPING
            pop (S);
        }
        push (S, elem);                  → INSERT

        while (isEmpty (temp) == 0) {
            push (S, top( temp));        } RETURN VALUES
            pop (&temp);
        }
    }
}
```

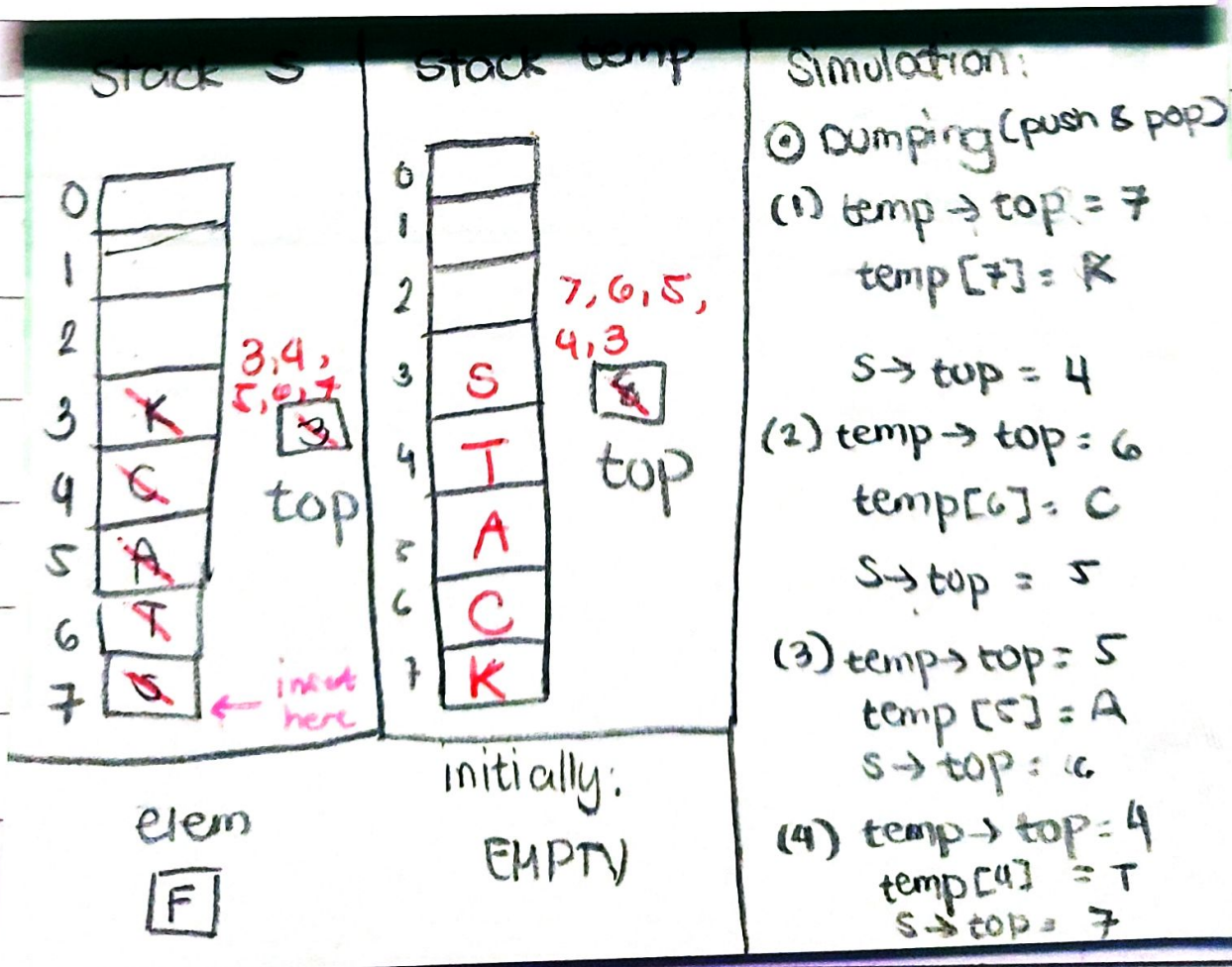


(1.) Declare & initialize temp variable to hold 'dumped' values from S.

(2.) Check if S is full (check if there is space available for insertion).

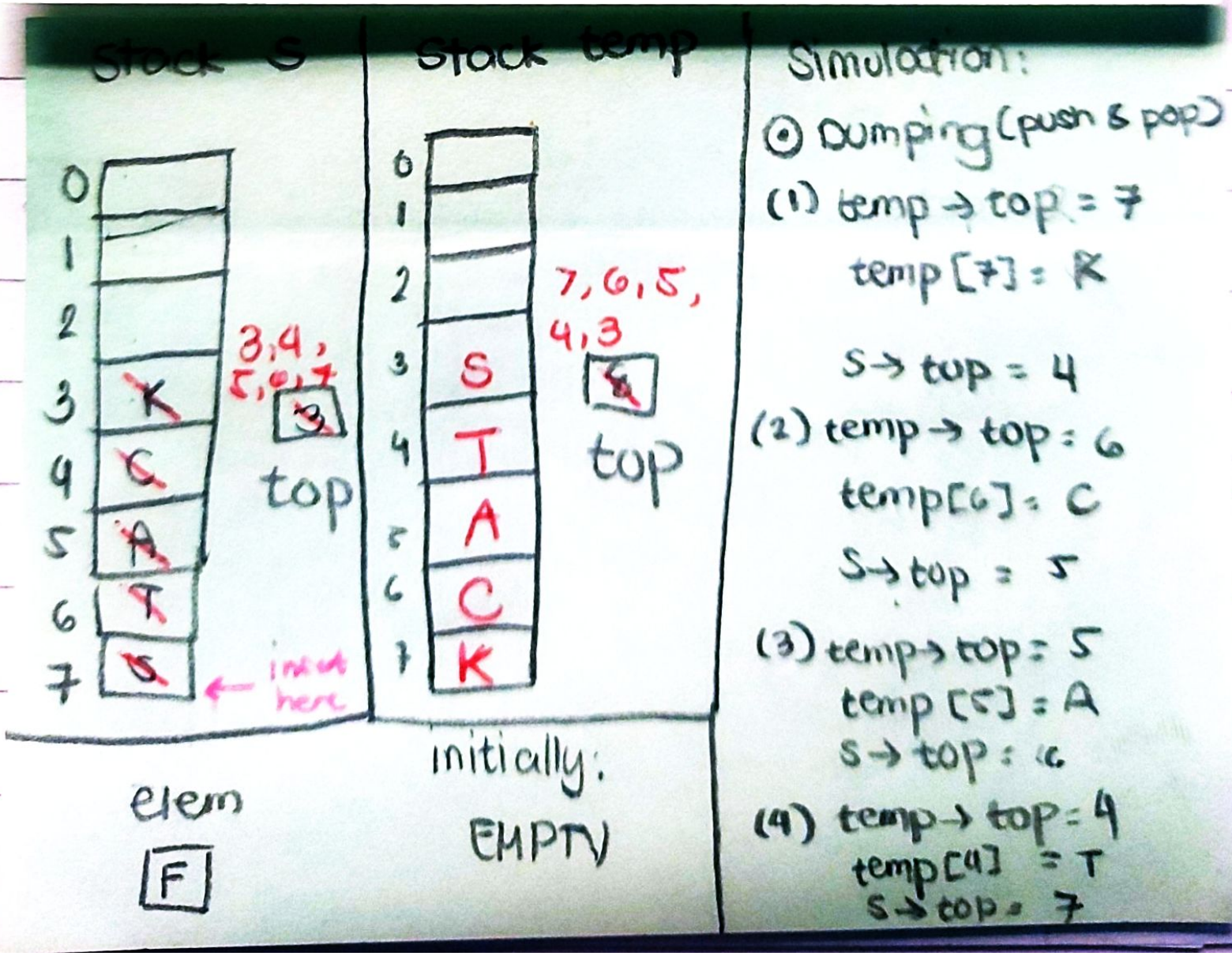
(3.) while S is NOT empty, dump each element into temp list/variable. and decrement S as you go.

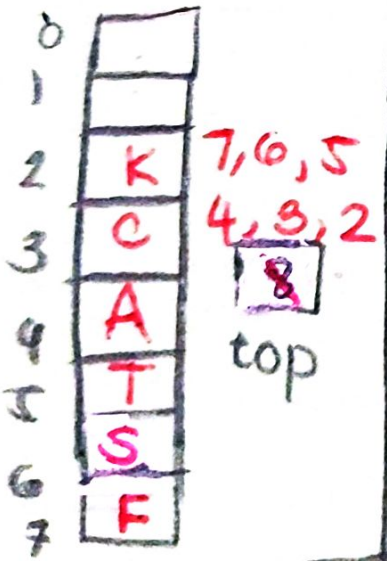
(4.) When list is empty, insert new elem. This is now at the bottom.

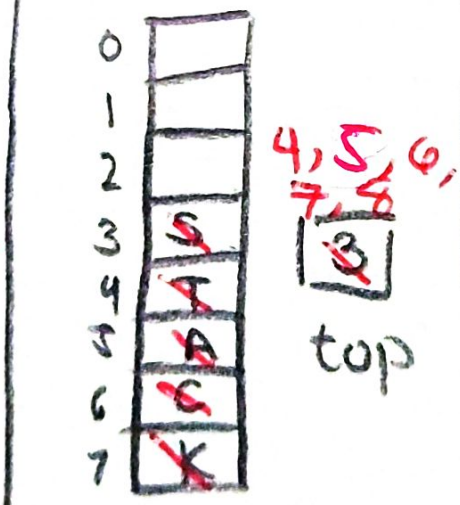


(5) Reinsert values back into S from temp.

(6) DONE.







initially EMPTY

(5) temp→top = 3
temp[5] = S
S→top = 8

(6) S→top = MAX (8)
so STOP! List is empty.

⊙ Insert Elem
S→top = 7
S[7] = 'F'

⊙ Reinsert values

final: S [2] temp [8]

NO.

DATE:

- STACK OPERATIONS -

(0 to MAX-1)

(LINKED LIST)

Definition:

```
typedef struct node{
    char data;
    struct node *link;
} *stack;
```

//initialize (s)

```
void initialize (stack *S){
    *S = NULL;
}
```

// PUSH (s, x)

```
void PUSH (stack *S, char elem){
    Stack temp;
    temp = (stack) malloc (sizeof (struct node));
```

* basically insertFirst()

```
    if (temp != NULL){
        temp -> data = elem;
        temp -> link = S;
        *S = temp;
    }
}
```

(1.) Declare & malloc temp node.
- Holds new data.

(2.) Check if malloc is successful & no need to check for available space in list since its linked list.

(3.) Store data in temp.

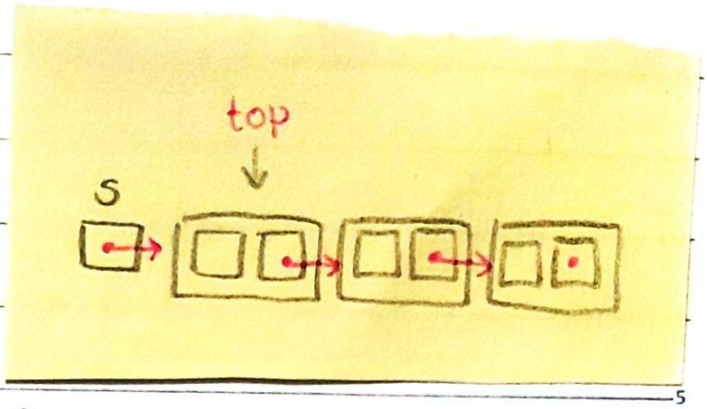
(4.) Update temp's link to → to node pointed to by S.

(5.) Update S.

NO:

DATE:

// POP (S)




```
void POP(stack *S) {
    Stack temp;

    if(isEmpty(*S) == 0) {
        temp = *S;
        *S = temp -> link;
        free(temp);
    }
}
```

*basically deleteFirst()!

// TOP (S)

```
char TOP(stack S) {
    return (isEmpty(S) == 0)? S -> data : '\0';
}
```

// isEmpty (S)

```
int isEmpty(stack S) {
    return (S == NULL)? 1 : 0;
}
```

// isFull (S)

*list will never be full, so just return 0.

```
int isFull(stack S) {
    return 0;
}
```

- QUEUE -

DATE:

* Queue "First In, First Out"

- insertion is done at the rear
- deletion is done at the front

* Operations

1.) ENQUEUE (x, Q) - insert element x at rear of queue

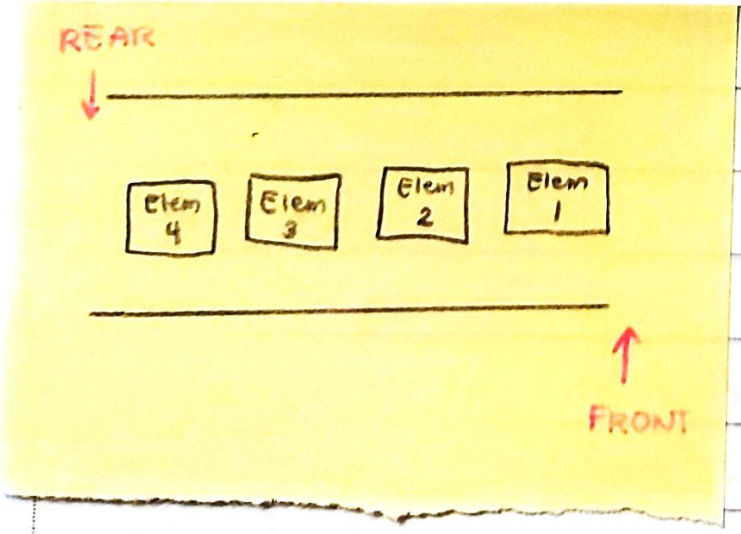
2.) DEQUEUE (Q) - delete element at front of queue

3.) FRONT (Q) - returns element at front of queue

4.) EMPTY (Q) - returns 0 if queue IS NOT empty
TRUE (nonzero value) IF EMPTY

5.) FULL (Q) - returns 0 if queue IS NOT full
TRUE(nonzero value) IF FULL

6.) INITIALIZE (Q) & MAKENULL (Q)



* Implementations

1.) Linked List

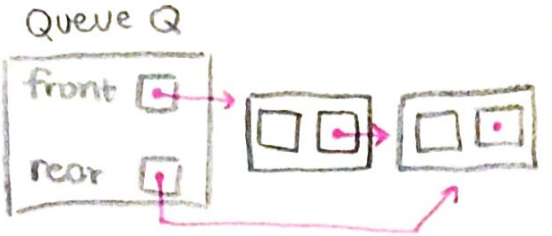
2.) Array

3.) CB List

NO:

DATE:

- LINKED LIST IMPLEMENTATION (QUEUE)

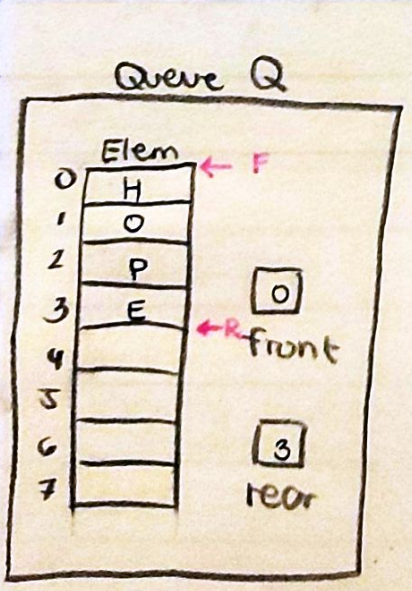


Enqueue = rear

Dequeue = front

- By having TWO POINTERS (front/rear) we can minimize the need for traversing. Since our operations take place at the FRONT and END of the list, it makes these operations O(1).

- ARRAY IMPLEMENTATION - (QUEUE)



• ENQUEUE (steps)

1.) Check if there is available space.

2.) If rear is MAX-1 & front != 0 then SHIFT.

3.) Adjust front & rear.

4.) Increment rear.

5.) Insert element.

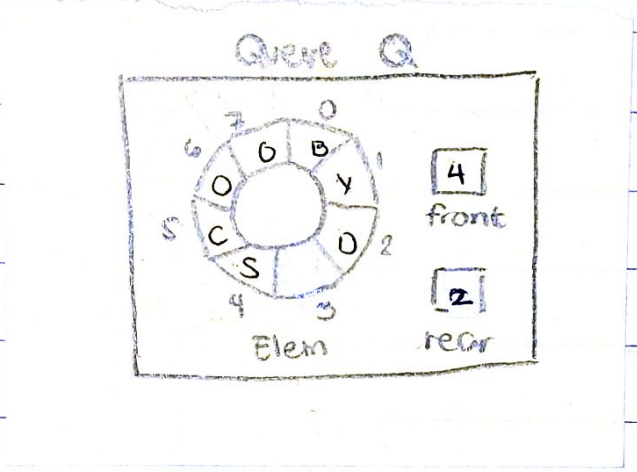
* BUT this implementation still involves SHIFTING! which is a problem and makes it O(N).

NO:

DATE:

* Circular Array Implementation

- solution to the problem of shifting
- declared as any array in C
- circular: orientation / manner in which array is manipulated
- no beginning / no end
- CLOCKWISE or COUNTERCLOCKWISE



* use MODULO (%) operator to move F & R.

* Enqueueing: spell check ang

- 1 element - Relative position of F & R: EQUAL
- Full Queue - Relative position of F & R: F ahead of R by 2 cells
- Empty Queue - Relative position of F & R: F ahead of R by 1 cell

* Queue is FULL if there is MAX-1 elements, since we sacrificed 1 cell to distinguish a Full Queue & Empty Queue.

DATE:

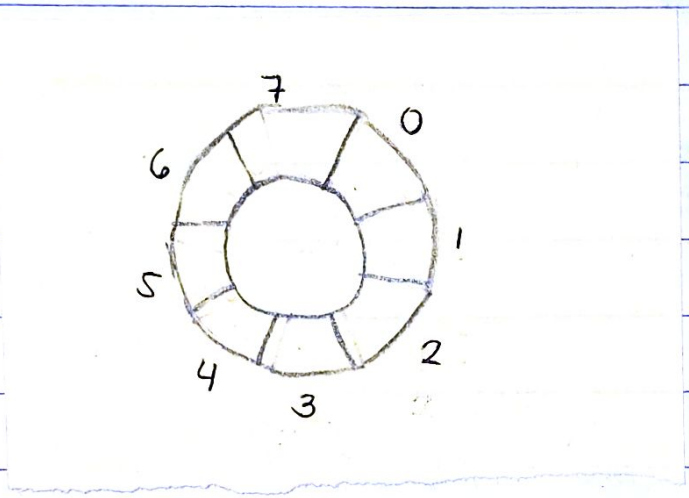
★ Checking FULL or EMPTY

• EMPTY

if((rear+1) % MAX == front)

Front is ahead by 1 cell

rear	front
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	0



• FULL

if(rear+2) % MAX == front)

Front is ahead by 2 cells

rear	front
0	2
1	3
2	4
3	5
4	6
5	7
6	0
7	1

★ to move Front/Rear by 1 cell:

front/rear = (front+1) % MAX

NO:

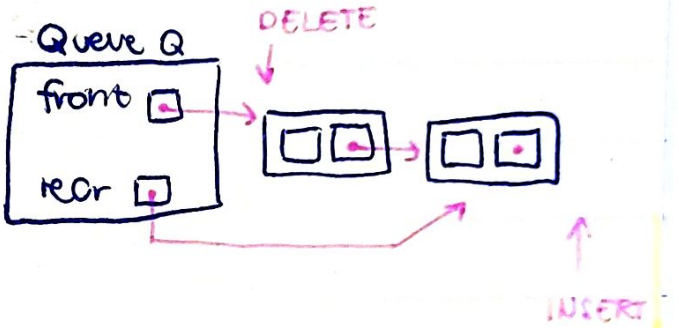
DATE:

- QUEUE OPERATIONS -
(LINKED LIST)

Definition

```
typedef struct node{
    char data;
    struct node *link;
} Nodetype;

typedef struct {
    Nodetype *front,
             *rear;
} Queue;
```



// initialize (Q)

```
void initialize (Queue *Q) {
    Q→ front = NULL;
    Q→ rear = NULL;
}
```

// ENQUEUE (Q, x)

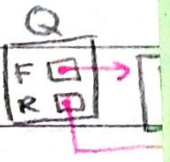
```
void Enqueue( Queue *Q, char elem) {

    Nodetype *temp;
    temp = (Nodetype*) malloc (sizeof (struct node));

    if (temp != NULL) {
        temp → data = elem;
        temp → link = NULL;
```

*no need to check if list is FULL bec its linked list

```
        Q → rear → link = temp;
    }
}
```



STEPS

IDE

(1) Allocate & declare temp node.

- assign new elem
- set link to NULL since this will be the LAST node in the list

(2) Assign rear's link field to point to temp node to connect it back to list.

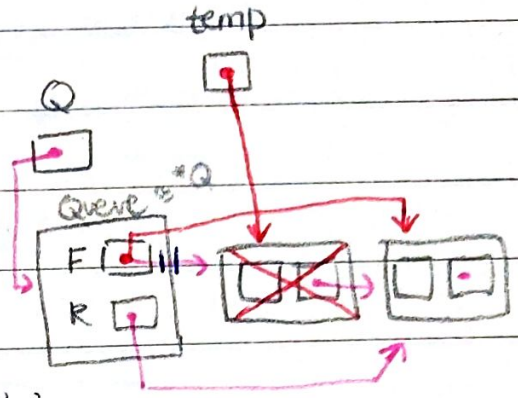
NO:

DATE:

// DEQUEUE (Q)

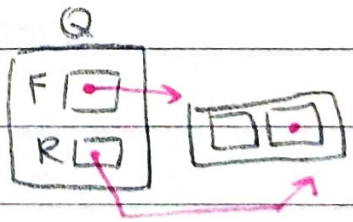
```
void dequeue (Queue *Q) {
    Nodetype *temp;

    if (isEmpty (Q) == 0) {
        temp = (*Q) -> front;
        (*Q) -> front  = temp -> link;
        free (temp);
    }
}
```



// FRONT (Q)

```
char front (Queue Q) {
  return (isEmpty (Q) == 0)? '\0' : (Q.front) -> Elem ;
}
```



// EMPTY

```
int isEmpty (Queue Q) {
  return (Q.front == NULL)? 1 : 0;
```

- QUEUE OPERATIONS -
(CIRCULAR ARRAY)

NO:
DATE:

//initialize (Q)

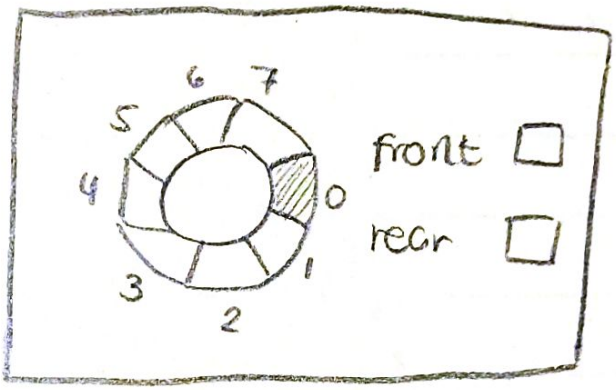
```
void initialize (Queue *Q){
    Q → front = 1;
    Q → rear = 0;
}
```

* remember that in an EMPTY QUEUE, front is ahead by 1 cell!

Definition:

```
typedef struct {
    char Elem [MAX];
    int front, rear;
} Queue;
```

*declared like any array!



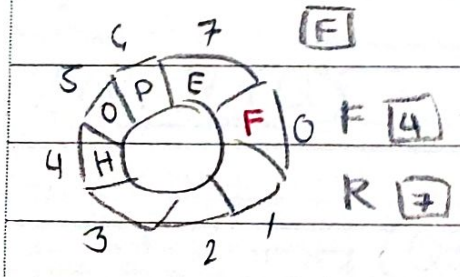
Queue Q

//ENQUEUE(Q, elem)

```
void enqueue (Queue *Q, char
if (isFull(*Q) == 0){
    Q → rear = (Q → rear +1) % MAX;
    Q → Elem[Q → rear] = elem;
}
}
```

- REMEMBER -
* insert at REAR
* delete at FRONT

15



① Simulation

Q→rear = ~~X~~ 0

Elem[0] = 'F'

① Check if Queue has space for new element.

② Increment rear by 1;

③ With new rear index, insert new element.

Front remains stationary!

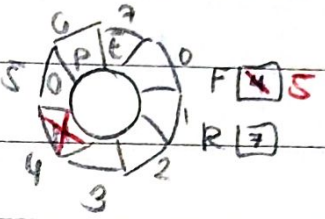
NO:

DATE:

// DEQUEUE (Q)

```
void dequeue (Queue *Q) {
    if (isEmpty == 0) {
        Q->front = (Q->front + 1) % MAX;
    }
}
```

*just move front by 1 to 'delete' element.



// FRONT (Q)

```
char front (Queue Q) {
    return (isEmpty(Q) != 0)? Q->Elem[Q->front]; '\0';
}
```

// FULL (Q)

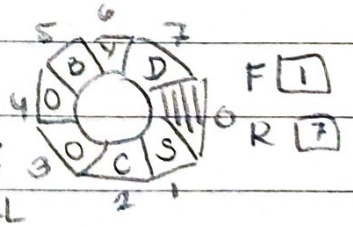
```
int isFull (Que Q) {
    return ((Q->rear + 2) % MAX == Q->front)? 1 : 0;
}
```

(7+2) % 8 = 1

9 % 8 = 1

1 = 1? YES

ITS FULL

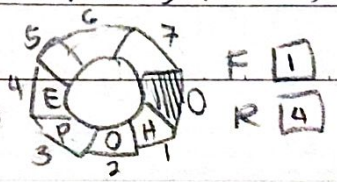


// EMPTY (Q)

```
int isEmpty (Queue Q) {
    return ((Q->rear+1) % MAX == Q->front)? 1: 0;
}
```

(4+1) % 8 = 1

NOT EMPTY 3 = 1? NO



IDEAS COME FROM GUESSING