STACKS & QUEUES

STACKS

→ a kind of list where insertions & deletions take place at the top
→ think of a stack of pancakes or an execution stack
→ First in, last out / last in, first out
→ insertFirst() & deleteFirst()

The challenge of stack is finding where the "top" of our list is located in our datatype.

STACK IMPLEMENTATIONS
① ARRAY IMPLEMENTATION
② LINKED LIST
③ CURSOR BASED

STACK OPERATIONS

1) PUSH (x, S) — inserts element at the top of the Stack (insertFirst)

2) POP (S) — deletes the element at the top of the stack (deleteFirst)

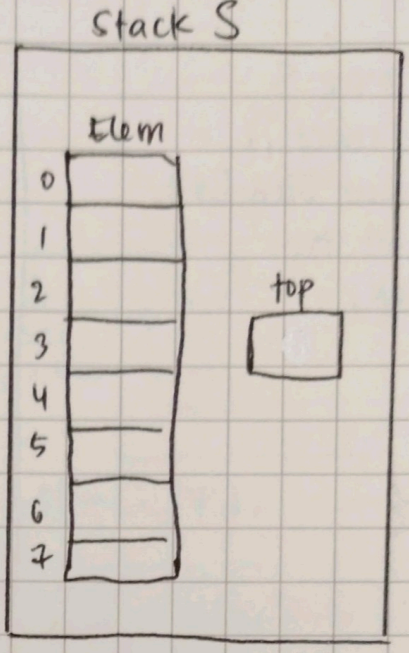
3) TOP (S) — returns the element at the top of the stack

4) EMPTY (S) — returns non-zero value if empty
FULL (S) — returns non-zero value if full
↳ only applicable for array / cursor based implementation

5) INITIALIZE (S) — initializes the Stack to be NULL so it can be used for the first time

STACKS - ARRAY IMPLEMENTATION

```
#define MAX 8
typedef struct {
    char Elem[8];
    int top;
} Stack;
```

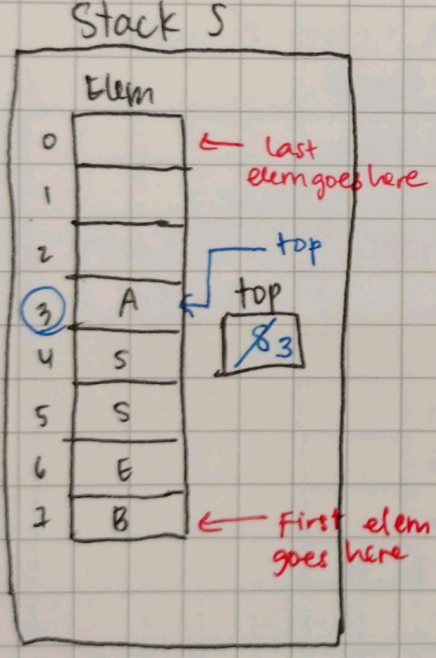


★ TWO VIEWS

① STACK GROWS FROM MAX-1 to 0

② STACK GROWS FROM 0 to MAX-1

VIEW 1 - STACK GROWS FROM MAX-1 to 0.



Empty stack → top = MAX

Full stack → top = 0

Push (x,S) → top decrements
insertFirst(x,S)

Pop (x,S) → top increments.

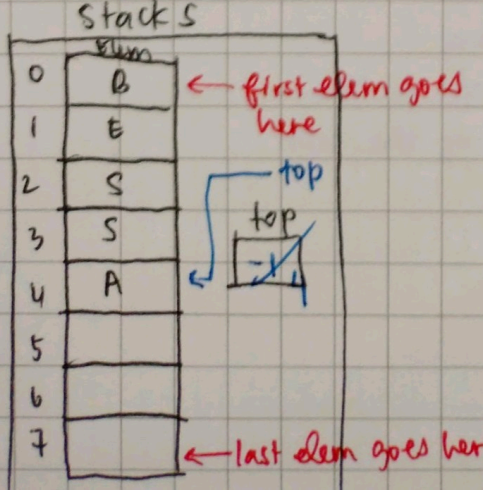
STACK GROWS FROM 0 to MAX-1 - VIEW 2

Empty stack → top = -1

Full stack → top = MAX-1

push (x,S) → top increases

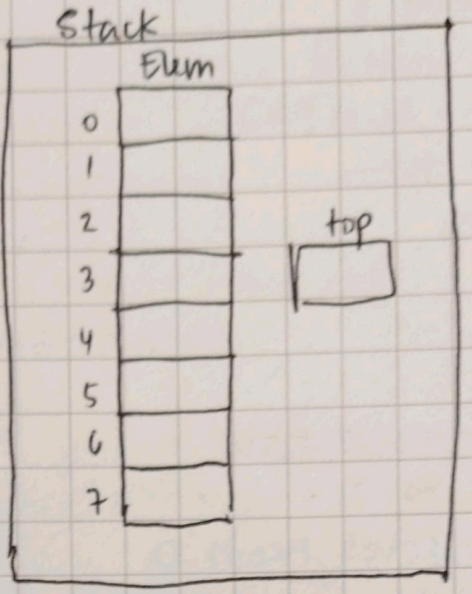
Pop (x,S) → top decreases



PRACTICE PROBLEM

Q. WRITE THE CODE OF THE FOLLOWING STACK OPERATIONS ASSUMING STACK WILL GROW FROM MAX-1 TO 0.

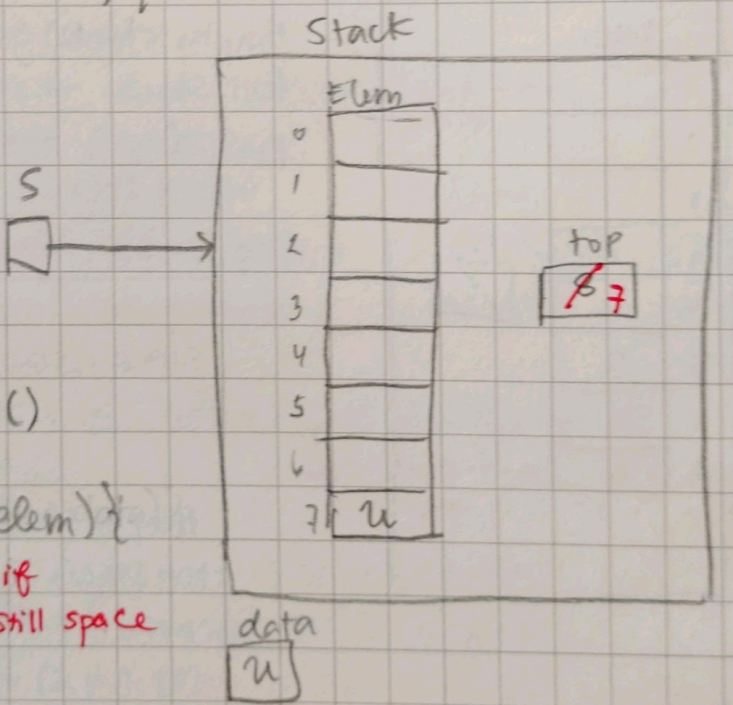
```
#define MAX 8
typedef struct {
        char Elem[8];
        int top;
} Stack;
```



ANS

A] INITIALIZE

```
void initializeStack(Stack *S){
    S → top = MAX;
}
```



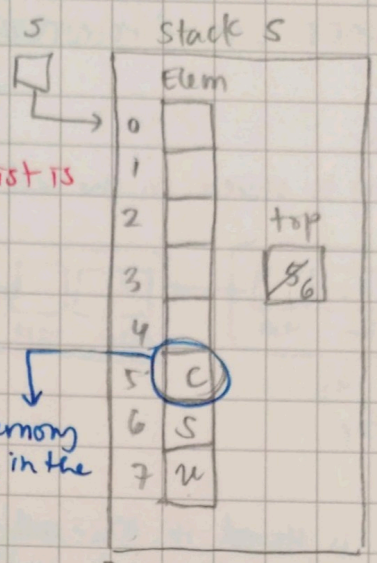
B] PUSH (x,S) ≈ insertFirst()

```
void push(Stack *S, char elem){
    if (S→top != 0){// checks if
                    there is still space
        S→top--;
        S→Elem[S→top] = elem; // assigning
                                 the element to the
                                 top index of the list
    }
```

→ when element is inserted, the top of the list moves up

C] POP (S) ≈ deleteFirst ()

```
void pop(stack *s){
    if(s->top != MAX){ // checks if list is empty
        s->top++;
    }
}
```



'C' will still be in memory but it is not logically in the queue.

D] TOP (S)

```
char top(stack s){
    return (s.top== MAX?)? '\0' : s.Elem[s.top];
}
```

↳ returns a char sentinel value if stack is empty

E] EMPTY (S)

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

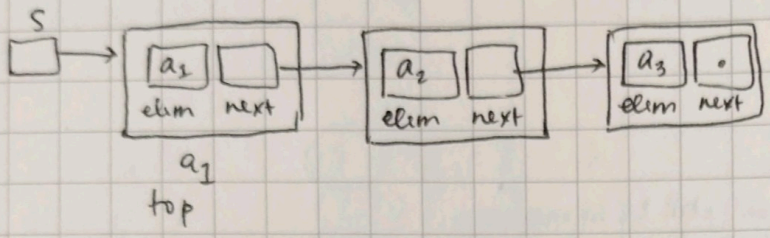
F] FULL (S) * only applicable to array & cursor based

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

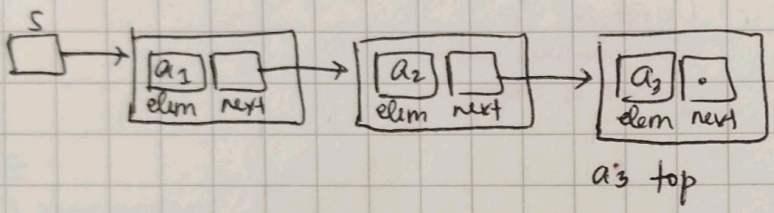
STACK - LINKED LIST (POINTER IMPLEMENTATION)

★ TWO VIEWS:

① Top elem is stored in the cell pointed to by stack pointer S



② Top elem is stored in the cell whose field is NULL

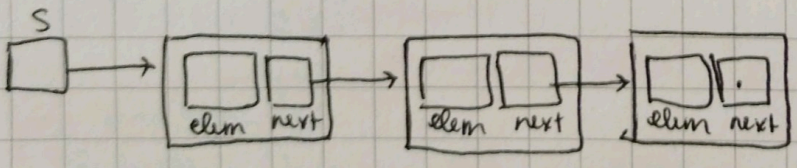


The goal is to have O(1) running time. O(N) is worst case scenario.

view ①: only need to insert & delete First, no traversal.
Running time = O(1) ✓

view ②: needs traversal first in order to insertFirst & deleteFirst().
Running time = O(n) X

PRACTICE EXERCISE



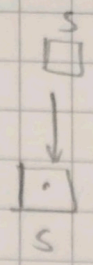
Q. Write the datatype of Stack S & write the code of the Stack operations.

ANS.
```
typedef struct node {
    char elem;
    struct node * link;
} * Stack;
```

Stack operations:

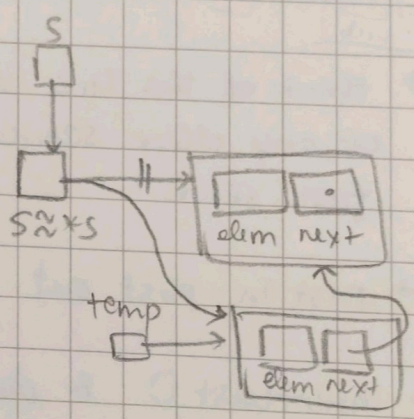
A] INITIALIZE

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



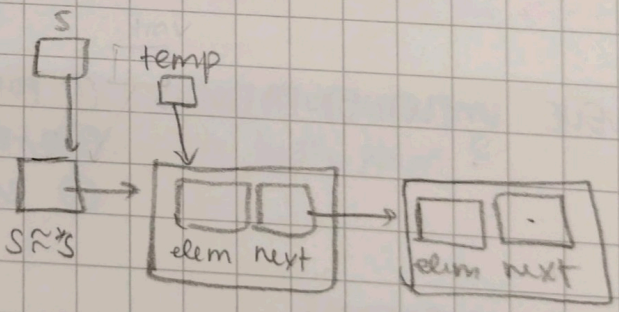
B] PUSH (x, S) ≈ insertFirst()

```
void push(stack *s, char data){
    Stack temp;
    temp= (stack) malloc(sizeof(struct node)); temp
    if (temp != NULL){
        temp → elem = data;
        temp → link = *s;
        *s = temp;
    }
}
```



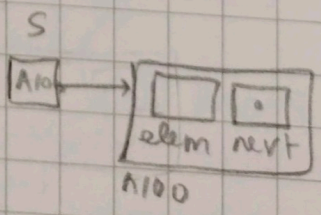
C] POP (S) ≈ deleteFirst()

```
void pop( stack *s){
    Stack temp;
    if (*s != NULL){
        temp = *s;
        *s = temp → next;
        free (temp);
    }
}
```



D] TOP (S)

```
char top(Stack s){
    return (s!=NULL)? s→elem : '\0';
}
```



E] Empty (s)

```
int isempty (s) {
    return ((*s) == NULL)? 1:0);
}
```