STACKS

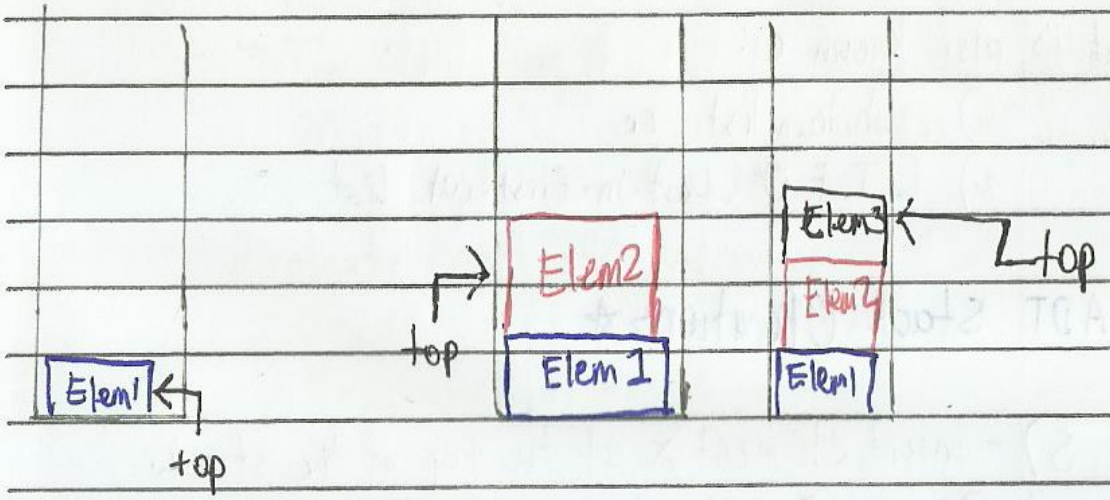
date 02/20/2023

"First IS Last"

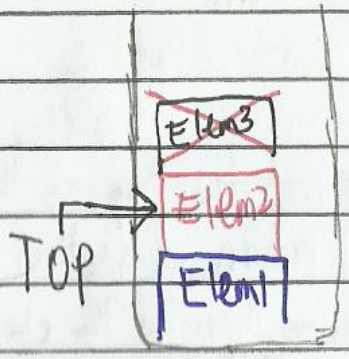
or

"Last Is First"

• Visual Representation of Inserting elements in Stack.



• Visual Representation of Deleting elements in Stack.



→ Stacks follow the Last In, First Out rule

→ Can also be First In, Last Out

date 2/20/2023

- What is a Stack?

→ It is a special kind of list in which all insertions and deletions take place at one end, called the top.

→ It is also known as:

1.) pushdown list or
2.) LIFO (Last-in-First-out) List

★ ADT Stack Operations ★

1.) PUSH (x, S) - inserts element x at the top of the stack.
≈ insertFirst()

2.) POP (S) - deletes the top element of the stack S
≈ deleteFirst()

3.) TOP (S) - returns the element at the top of the stack S
- if empty, it will still return a sentinel value

Note: Sentinel values are the same type as the element but not included in the set of elements.

4.) EMPTY (S) - returns true (nonzero value) if the stack S is empty; otherwise return false (zero).

5.) FULL (S) - returns true (nonzero value if the Stack S is full; otherwise return false (zero).

NOTE: only applicable for Array & CB Implementation

Strikemore

> Some Stack Applications

- Execution Stack
  - → In compiler design, a stack is used to organize the activation records of functions that are invoked or called.
  - → The activation record of the called function is found on top of the activation record of the calling function.

- Evaluate arithmetic expression
  - → Stack is used to convert infix expression to postfix Expression
  - → Evaluate postfix Expression

• Implementations of ADT Stack •

1.) Array Implementation
   - a.) Static Array
   - b.) Dynamic Array

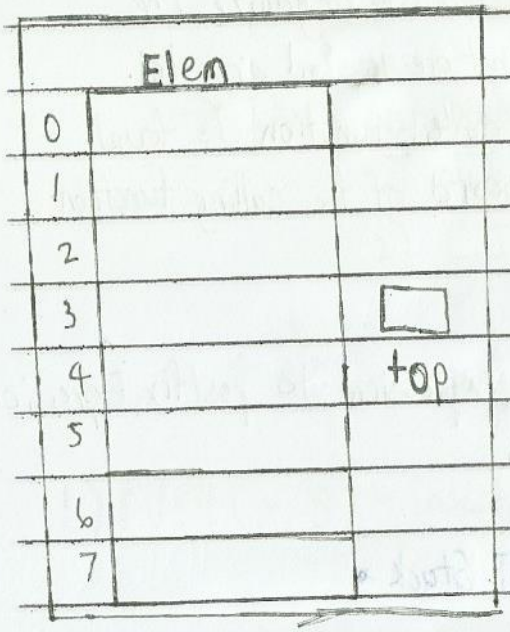
2.) Linked List (Using Pointers)
   - a.) Singly-Linked
   - b.) Doubly-Linked

3.) Cursor-Based Implementation

date 2/20/2023

* Array Implementation *

Illustration:



- Write an appropriate definition of datatype Stack.

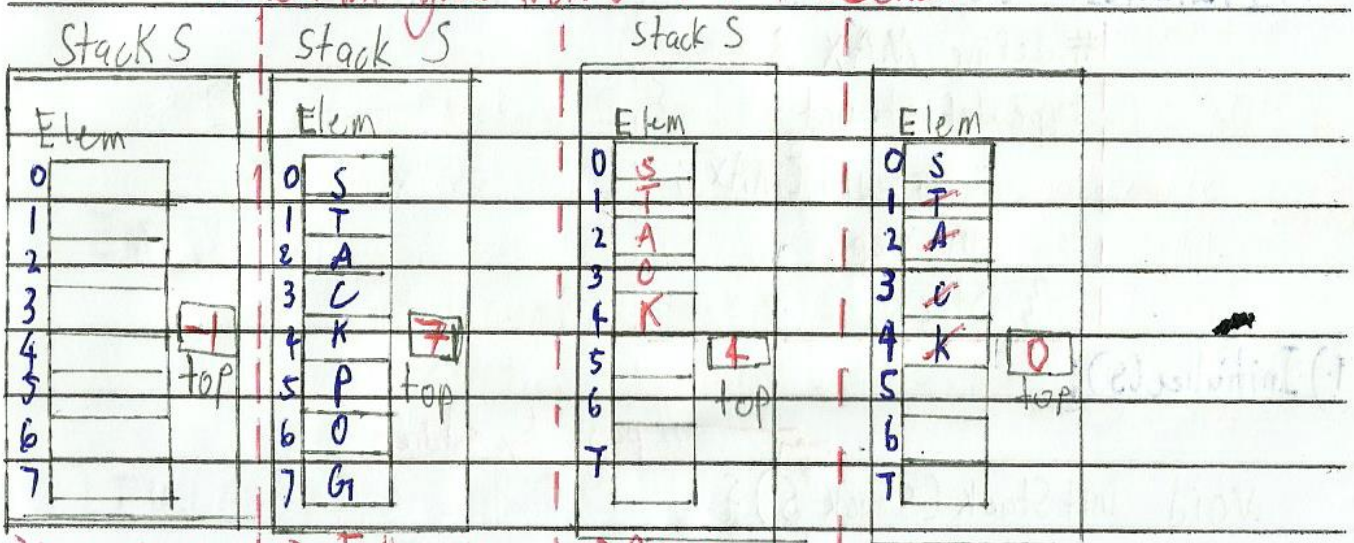
```
#define MAX 8
typedef struct {
  char Elem[MAX];
  int top;
}
```

- In Array Implementation, Stack has two views:
  1.) Stack will grow from max-1 to 0
  2.) Stack will grow from 0 to max-1 (the usual for Array Imp)

- View 1: Stack will grow from max-1 to 0 Scenarios

date 2/10/2023

- View 2: Stack will grow from 0 to max-1 Scenarios



1.) Empty Stack
↳ top is -1

2.) Full stack
↳ top is max-1

3.) PUSH(x,S)
↳ top increases

4.) POP(S)
↳ top decreases

- Exercise 1 - Write the code the ff. stack Operations using the declaration given and assuming that the stack will grow from max-1 to 0.

1.) INITIALIZE(S)
2.) PUSH(x, S)
3.) POP(S)
4.) TOP(S)
5.) EMPTY(S)
6.) FULL(S)

Exercise 1 - stack operations if stack grows from max-1 to 0

date 2/20/2023

Data Struct Def. (for Guide):

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

1.) Initialize(S);

```
// pass by address
void initStack(Stack *S) {
    S->top = MAX; // is equals to max since view 1 where an Empty set, top = MAX
}
```

2.) PUSH(x, S); ≈ recall: equivalent to insertFirst()

```
// pass by address since we're inserting
void push(char x, Stack *S) {
    if (S->top != 0) { // check if not full stack
        S->top--; // decrement top (or move)
        S->data[S->top] = X ; // assign the element to the the current position of top
    }
```

3.) POP(S); ≈ recall: equivalent to deleteFirst()

```
void pop(Stack *S) {
    if (S->top != MAX) { // check if not empty Stack
        S->top++; // increase top
    }
```

Strathmore

Exercise 1: Stack Operations if Stack grows from max-1 to 0

date 2/20/2023

4.) TOP(S);

pass by copy since we just return what value is at top

```
char top(Stack S){
    return S.top != MAX ? S.data[S.top] : '\0';
}
```

Check if not empty

returns sentinel value if empty

5.) EMPTY(S);

Pass by copy

```
int isEmpty(Stack S){
    return S.top == MAX ? 1 : 0; // in this view 1, Stack is Empty if top == MAX
}
```

6.) FULL(S);

Pass by copy

```
int isFull(Stack S){
    return S.top == 0 ? 1 : 0; // in view 1, stack is full if top == 0.
}
```

Practice: Stack operations but stack grows from 0 to max-1

1.) Initialize(S);

```
void initStack(Stack *S){
    S->top = -1; // set stack as Empty,
}                // Stack is empty if top=-1 in View/ver-2
```

2.) PUSH(x, S);

```
void push(char x, Stack *S){
    if(S->top != MAX-1){ // Check if not Full Stack
        S->top++; // move to vacant space (?)
        S->data[S->top] = x; // assign the element to said space
    }
```

Practicing — if Stack grows from 0 to max-1

date 2/20/23

3.) POP(S);

Pass by address

```
void pop (Stack *S){
    if (S->top != -1){ // check is Stack is not empty
        S->top--; // decrement the top
    }
}
```

4.) TOP(S);

// pass by copy

```
void top(Stack S){
return S.top != -1 ? S.data[S.top] : '\0';
}
```

check if not empty

returns sentinel value if empty

5.) EMPTY(S);

```
Void isEmpty (Stack S){
return S.top == -1 ? 1 : 0
}
```

6.) FULL(S);

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

Strickmore

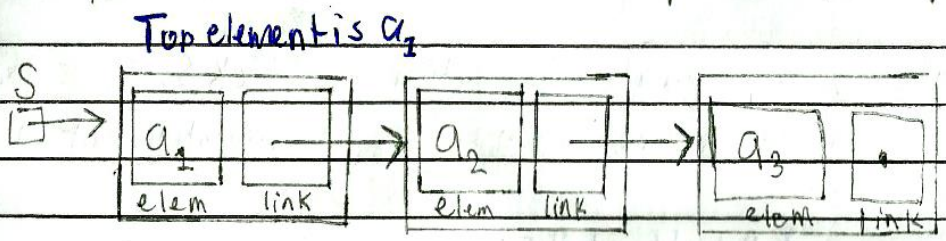
Singly-Linked Implementation

date 2/20/23

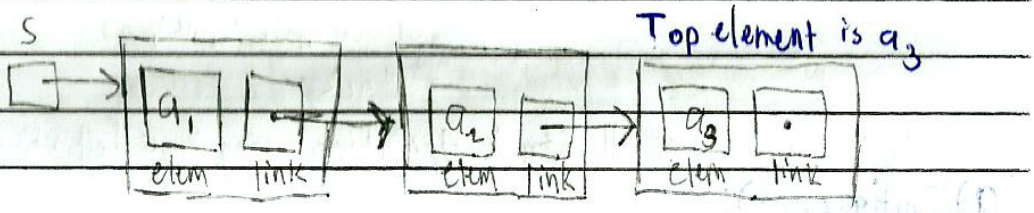
Illustration

→ Two orientations of top element:

1.) Top element is stored in the cell pointed to by stack pointer S.



2.) Top element is stored in the cell whose next field is NULL



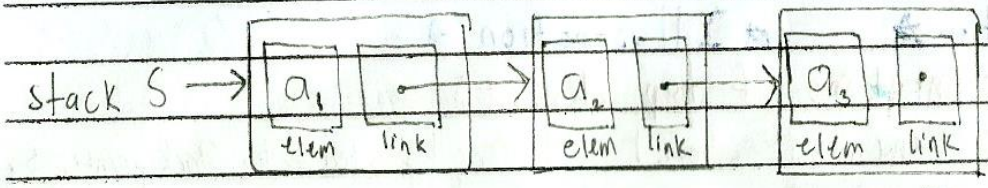
• Which Orientation is better? •

→ Orientation 1 is better than 2 since we would only need to access the top element. Orientation 2 requires traversal and is considered to have a running time of O(N), which is not the best. Orientation 1 requires NO traversal & has a running time of O(1)

Strathmore

Exercise 2 - Singly Linked-List Implementation of Stack

date 2/20/2023



1.) Write an appropriate definition of Stack S. NOTE that "top" in this implementation is only an orientation, not a variable.

* Data Struct Def.:

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

2.) Write the code of all Stack Operations using your definition.

a.) Initialize(S);

```
void initStack (Stack *S) {
    *S = NULL;
}
```

b.) push(x, S);

```
void push ( char x, Stack *S) {
    Stack temp = (Stack) malloc (sizeof (struct node));
    if (temp != NULL) {
        temp->data = elem;
        temp->link = *S
        *S = temp;
```

*NOTE: This is equivalent to insertFirst()

Exercise 2

date 2/20/2023

3.) POP(S);

```
void pop(stack *S){
    Stack temp;
    if(*S != NULL){
        temp = *S
        *S = temp -> link;
        free(temp);
    }
}
```

// Note: this is equivalent to deleteFirst()

4.) TOP(S);

```
char top(stack S){
    return S!=NULL ? S->data : '\0';
```

5.) EMPTY(S);

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

6.) FULL(S);

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

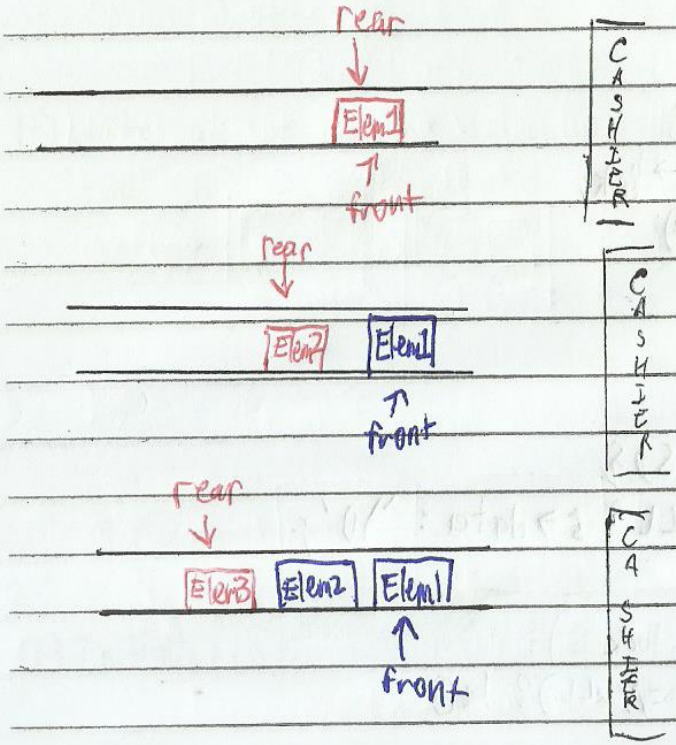
// NOTE: This function is only applicable in Array & Cursor based implementation, NOT LL implementation

QUEUES

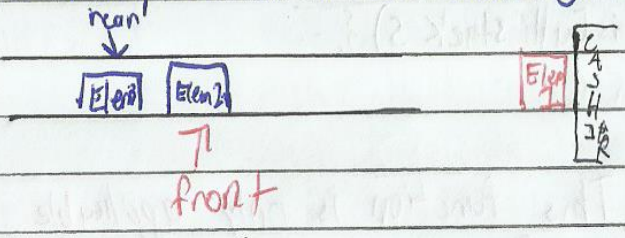
date 2/20/2023

"First is First" or "Last is Last"

* Visual Representation of inserting elements in the Queue



* Visual Representation of deleting elements in the Queue



- What is a queue?
  - → It is a special kind of list in which insertion is done on one side called rear, while deletion is done on the other side called front.
  - → is called FIFO (First In First Out) List.

QUEUES

date 2/20/2023

• ADT Queue Operations •

1.) ENQUEUE (X, Q) - insert element X at the rear of the Queue Q.

2.) DEQUEUE (Q) - deletes the front element of the Queue Q.

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

4.) EMPTY(Q) – returns true (nonzero value) if the Queue Q is empty; Otherwise returns false (zero)

5.) FULL (Q) - returns true (nonzero value) if the Queue Q is full; otherwise return false (zero).
- NOTE: applicable for both Array & C.B. implementations only

★ Some Queue Applications ★

1.) Scheduling Algorithms
- → process scheduling in a time-shared computing system.
- → resource scheduling as printers, disks, etc.

2.) Searching Algorithms
- → Non-linear search such as Depth First Search (?)

★ Implementations of ADT Queue ★

1.) Linked List (using pointers) - Singly-Linked, Doubly-Linked
2.) Array Implementation - Static Array, Dynamic Array
3.) Cursor-based Implementation.

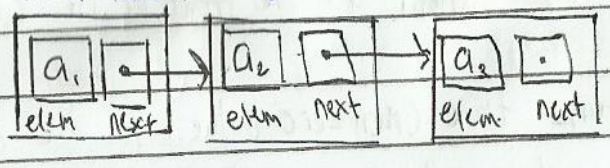
Queue Linked-List (pointer) Implementation

date 2/20/2023

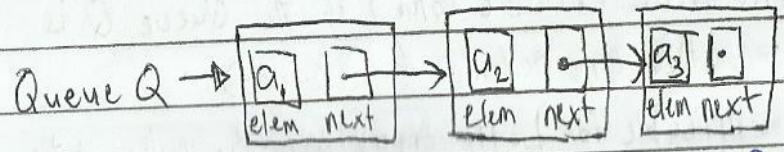
- Singly-Linked Version 1 - Queue is a pointer to a self-reference structure

- Two Orientations of FRONT element

1.) Front element is stored in the cell pointed to by Queue Q.



2.) Front element is stored in the cell whose next field is NULL.



Which Orientation is better?

For orientation #1, ENQUEUE is $O(N)$ & DEQUEUE is $O(1)$, constant time.

For orientation #2, ENQUEUE is $O(1)$ & DEQUEUE is $O(N)$.

They are BOTH the SAME

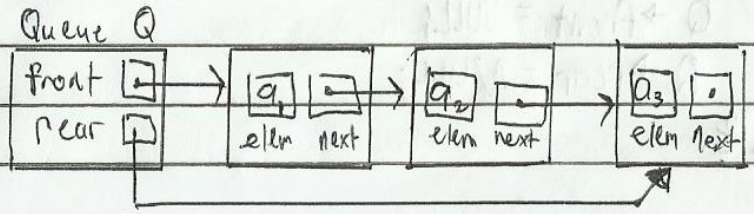
Queue Linked List (Pointer) Implementation

date 2/20/2023

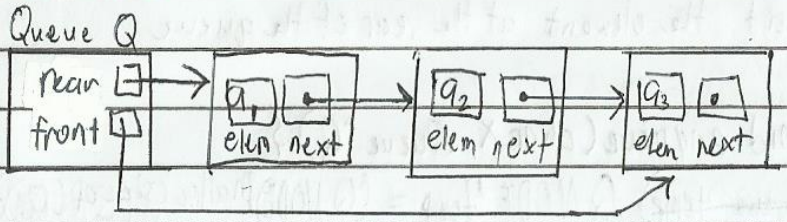
- Singly-Linked Version 2 - Queue is a structure containing FRONT & REAR pointers.

- Two versions of Queue Q.

Version A:



Version B:



Which version is better? - A is the better version

- Exercise - Write an appropriate declaration of Queue Q

> Code:

```
typedef struct node {
    char elem;
    struct node *next;
} QNODE;

typedef struct {  // Queue is a structure in this version
    QNODE *front, *rear;
} Queue;
```

Queues - LL imple practice

date 2/20/2023

- Write the code of the ff. functions:

a.) initQueue() - the function will initialize it to be empty.

- Code:

```
void initQueue(Queue *Q){
    Q->front = NULL;
    Q->rear = NULL;
}
```

set to empty to initialize

b.) enqueue() - Given the queue and an element, the function will insert the element at the rear of the queue

~ InsertLast in LL

- Code:

```
void enqueue(char X, Queue *Q){
    QNODE *temp = (QNODE*) malloc(sizeof(QNODE));
    if(temp != NULL){ //checking
        temp->elem = X;
        temp->next = NULL;

        if(Q->front == NULL && Q->rear == NULL){
            Q->front = temp; //set the front to new node (temp)
        }else{
            Q->rear->next = temp; //add the new node in rear->next
        }
        Q->rear = temp; //make the new node as the new rear
    }
}
```

pointer to pointer concept

to check if Queue is empty & if it is the first node to be inserted

NOTE: This can still be improved, im just too sleepy r.n.

Strandmore

Queues - LL Imple pract.

date

c.) dequeue() - given the queue, the function will delete the front element if it exists.

delete first
NULL(Q)

• Code:

```
Void dequeue(Queue *Q){
    QNODE *temp;
    if(Q->front != NULL){
        temp = Q->front; // let temp hold the current front
        Q->front = Q->front->next; // set front to the next value
        free(temp); // free temp
    }
    if(Q->front == NULL){ } if front is NULL,
        Q->rear = NULL; } set rear to NULL
    }
```

d.) front() - Given the queue, the function will return the front element if it exists.

```
Char front(Queue Q){
    return Q.front != NULL ? Q.front->elem : '\0';
```

e.) Empty() -

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

Strathmore

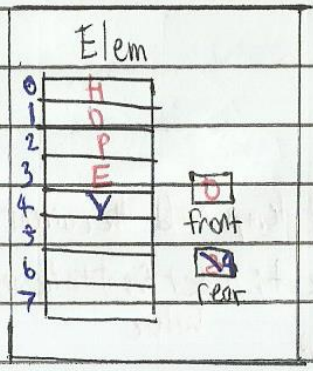
Queues Array Implementation

date 2/20/2023

- Variation 1

•ENQUEUE•

Queue Q



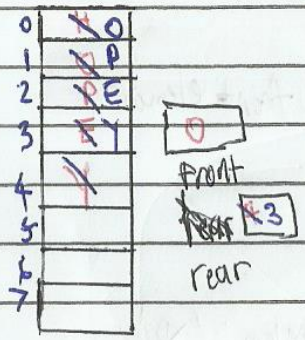
Enqueue element 'Y':

Steps:

- Check if there is available space
- increment rear
- insert the element.

• DE QUEUE •

Queue Q



Dequeue an element:

Steps:

- Check if queue is not empty
- Shift elements
- decrement rear

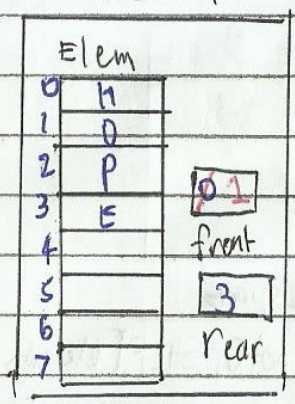
NOTE: Physically element Y at index 4 is still there but logically it is not part of the queue anymore.

Strathmore

Queues Array Implementation

• Variation 2 date 2/20/2023

Queue Q



• DEQUEUE •

Dequeue an element

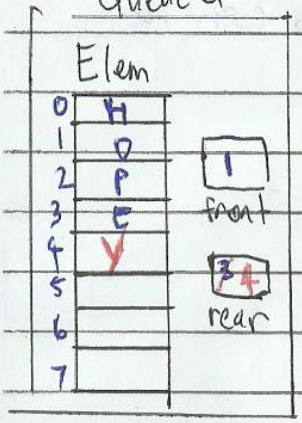
steps:

- check if queue is not empty
- Increment front

NOTE: Physically H element is still there but logically it is not part of the queue anymore.

• ENQUEUE •

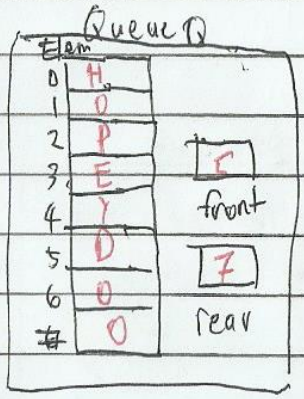
Queue Q



Steps:

- Check if there is available space
- increment rear
- insert the element

• What if Queue was like this"

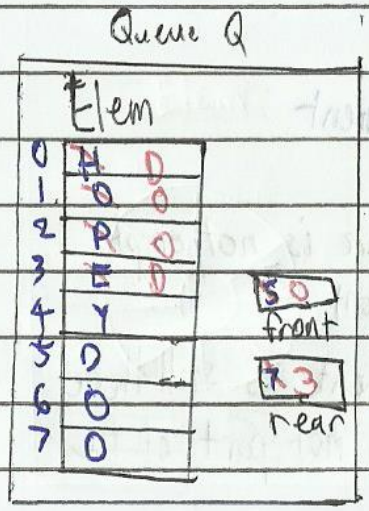


- How many elements are there in the Queue Q? - 3
- Is Enqueue possible & How
  - Yes, shift existing elements to make room for new elements

Queues Array Implementation

date 2/20/2023

• ENQUEUE Revisted •



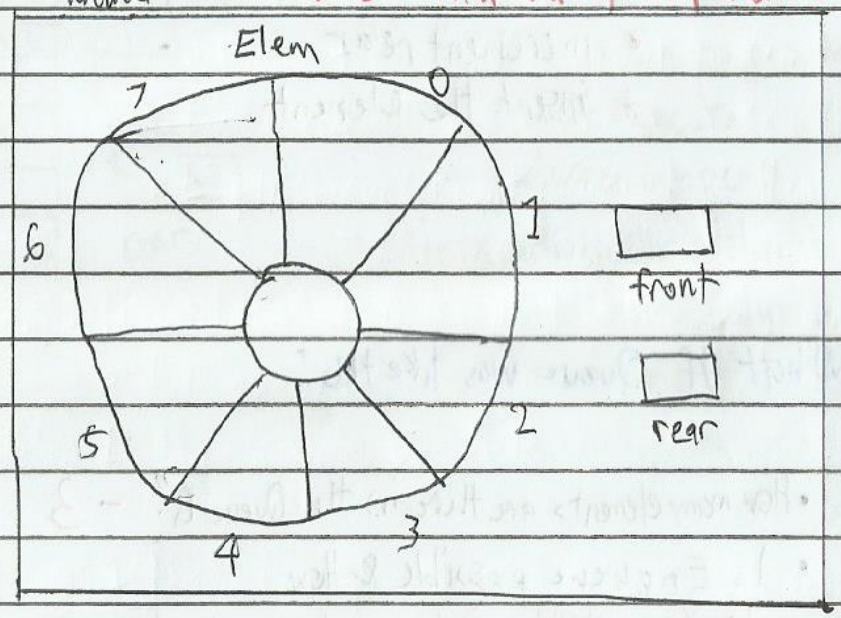
Enqueue element D:

Steps:

- check if there is available space
- if rear is max-1 & front not 0, SHIFT elements
- Adjust front & rear
- increment rear
- insert the element.

is there a solution where shifting of elements can be avoided? Yes:

* Circular Array Implementation



Queue – Circular - Array Implementation

date 4/20/2023

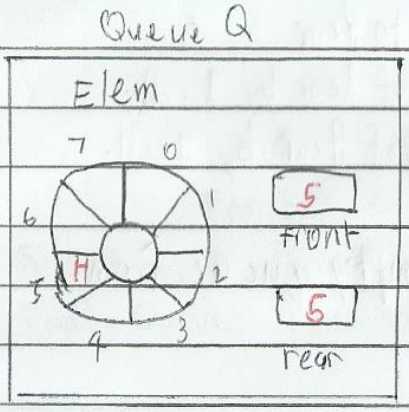
★ Characteristics of Circular Array ★

- Declared in the same manner as any array in C.
- Being circular is just an orientation, or a manner in which the array is manipulated.
- Like a circle, the array has no beginning and no end; hence the first element can be stored anywhere.

- Succeeding elements are inserted in CLOCKWISE or COUNTERCLOCKWISE direction; hence front & rear will also move clockwise or counterclockwise

- Note: CLOCKWISE MOVEMENT will be used in the ff. operations.

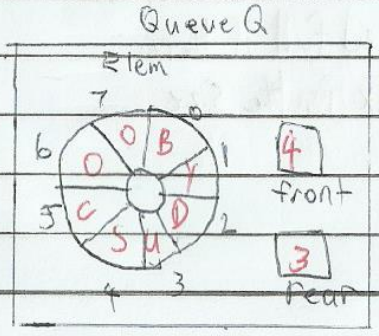
★ ENQUEUE 1 ELEMENT



Enqueue 1st element: H:

- How many different options are there?
  → 8
- Relative position of front & rear?
  → Front is equal to rear

★ FULL QUEUE ★



Enqueue the elements: SCOOBYDU

- What is the relative position of front & rear? - front is ahead by 1 cell

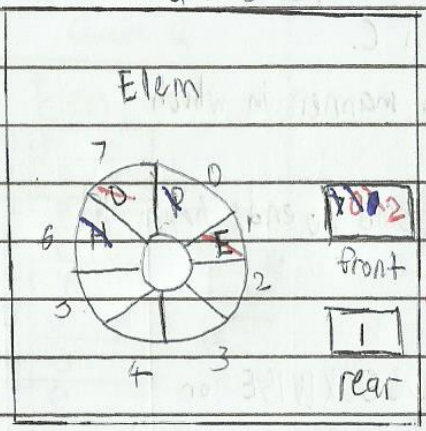
Strathmore

Queues - Circular Array Implementation

EMPTY QUEUE

date 2/20/2023

Queue Q



Dequeue	Remove	Front
1st	H	7
2nd	O	0
3rd	P	1
4th	E	2

EMPTY QUEUE!!

- What is the relative position of front and rear?
  - front is ahead of rear by 1 cell

✱ Summary: Circular Queue

- 1 Element: Front is equal to Rear
- Full Queue: Front is ahead of Rear by 1 cell.
- Empty Queue: Front is ahead of Rear by 1 cell.

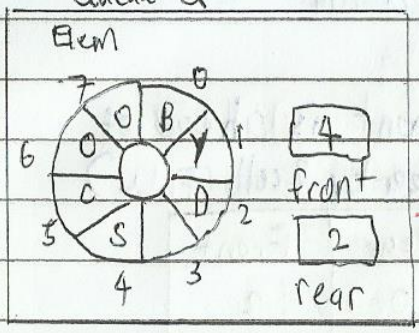
Problem: How do we distinguished an empty queue from a full queue?

- Solution 1: Add a counter variable
- Solution 2: Sacrifice 1 cell (Queue is considered full if there are (max - 1) elements, where max is the size of the array.

Queues - Circular Array Implementation

date 2/20/2023

Queue Q



Sol 2: FULL QUEUE

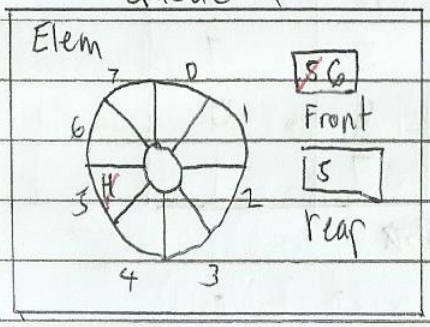
• Enqueue elements:
SCOOBYDU

→ 'U' is sacrificed

"What is the relative position of front & rear?
→ front is ahead of rear by 2 cells.

Solution 2 : Empty Queue

Queue Q



• Dequeue an element.
• Relative position of front & rear?
→ Front is ahead of rear by 1 cell

★ SUMMARY: Solution 2 ★

- 1 Element: Front is equal to Rear
- Full Queue: Front is ahead of Rear by 2 cells.
- Empty Queue: Front is ahead of Rear by 1 cell.

More illustration: →

Strathmore

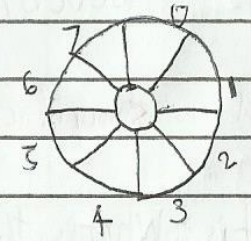
Queues: Circular Array Implementation

date 2/20/2023

- Checking if Empty or Full.

Front is ahead of Rear by 1 cell (Empty)

Rear	Front
0	1
1	2
2	3
3	4
4	5
5	6
6	7
7	0



Front is ahead of Rear by 2 cells (FULL)

Rear	Front
0	2
1	3
2	4
3	5
4	6
5	7
6	0
7	1

★ ADDITIONAL NOTES ★

- To move front:
  front = (front + 1) % MAX
- To move rear
  rear = (rear + 1) % MAX
- Empty: (rear + 1) % MAX == front.
- Full: (rear + 2) % MAX == front

Practice Exercise:

1.) Write an appropriate declaration of the circular array implementation of Queue Q using Solution 2.

2.) Using the declaration in #1, write the code of the Queue operations:

Queue Circular Array Imp.
Exercise

date

• Declaration of Queue Q: // writers note: not sure if correct :>

```
#define MAX 8
typedef Struct {
    char Elem[MAX];
    int front;
    int rear;
} Queue;
```

• Queue operations in Solution 2 circular Array

1) initialize:

```
void initQueue(Queue *Q) {
    Q->front = 1;   // front is ahead by 1 cell if empty
    Q->rear = 0;
}
```

2) int isEmpty():

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

3) int isFull():

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

4) enqueue():

```
void enqueue(Queue *Q, char x) {
    if(isFull(*Q) != 1) {
        Q->rear = (Q->rear + 1) % MAX;   // set rear to the next node(?)
        Q->Elem[Q->rear] = data;         // place the data in the new rear (?)
    }
}
```

Strathmore

→

Queue Circular Array Imple Exercise

date

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

6.) front():
```
char front(Queue Q){
    return (isEmpty(Q)!=1)? Q.Elem[Q.front]: '\0';
```

7.) display Queue:
```
void displayQueue(Queue Q){
    int index;
    if((Q.rear+1)%MAX != Q.front){
    for(index=Q.front; index != (Q.rear+1)%MAX;
                index = (index+1)%MAX){
        printf("%c ", Q.Elem[index]);
    }
    }else{
        printf("Queue is empty");
    }
}
```

Set index to front

If index has not reached rear proceed (?)