Stacks and Queues

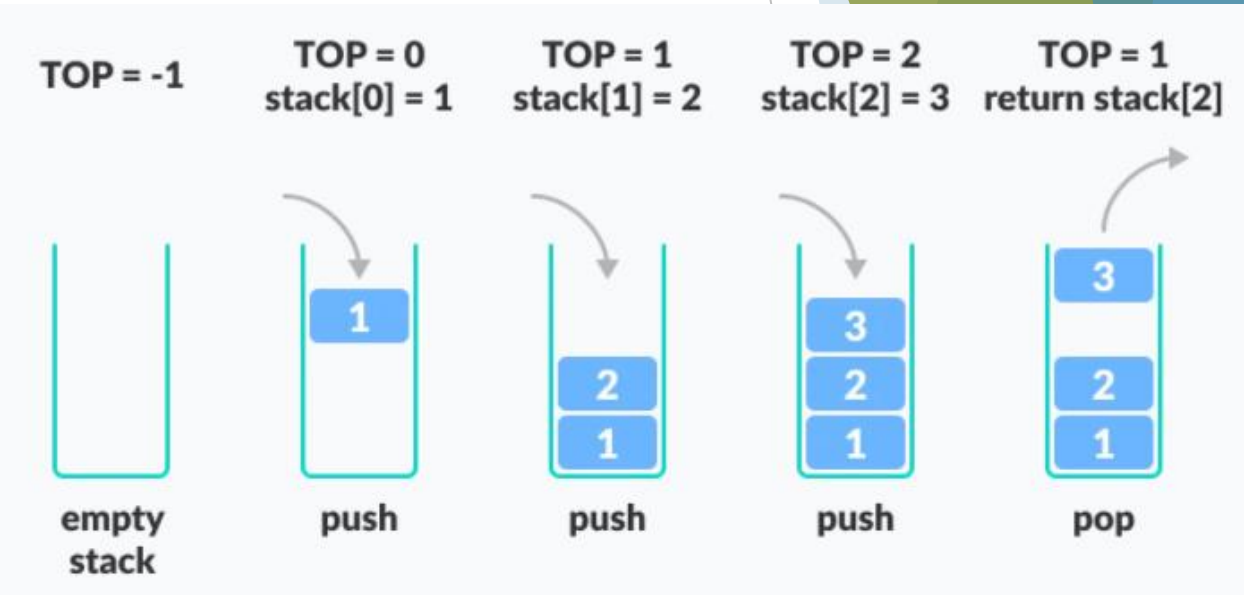
By Asst Prof Christopher Uz

Course: DSAA

Pillai College of Engineering

Stack (ADT)

- A Stack is a linear DS that follows the LIFO (Last-In-First-Out) principle
- It contains only one flag ‘top pointer’ pointing to the topmost element of the stack
- Insertion and Deletion of elements are done at only one end, which is known as the top of the stack
- Implemented using array/linked list



Basic Operations of Stack

- **push()** – Pushing (storing) an element on the stack
- **pop()** – Removing (accessing) an element from the stack
- **peek()** – Get the top data element of the stack, without removing it
- **isFull()** – Check if stack is full
- **isEmpty()** – Check if stack is empty
- **show()** – Display the stack

Array Implementation of Stack

IsFull Operation

```
bool isFull() {
   if(top == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

IsEmpty Operation

```
bool isEmpty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Array Implementation of Stack…

Push Operation

```
void push(int data) {
   if(!isFull()) {
      top++;
      stack[top] = data;
   }
   else {
       printf("Cannot insert data, Stack is full.\n");
   }
}
```

Pop Operation

```
int pop() {
   int data;
   if(!isEmpty()) {
      data = stack[top];
      top--;
      return data;
   }
   else {
         printf("Cannot retrieve data, Stack is empty.\n");
   }
}
```

Array Implementation of Stack…

Peek Operation

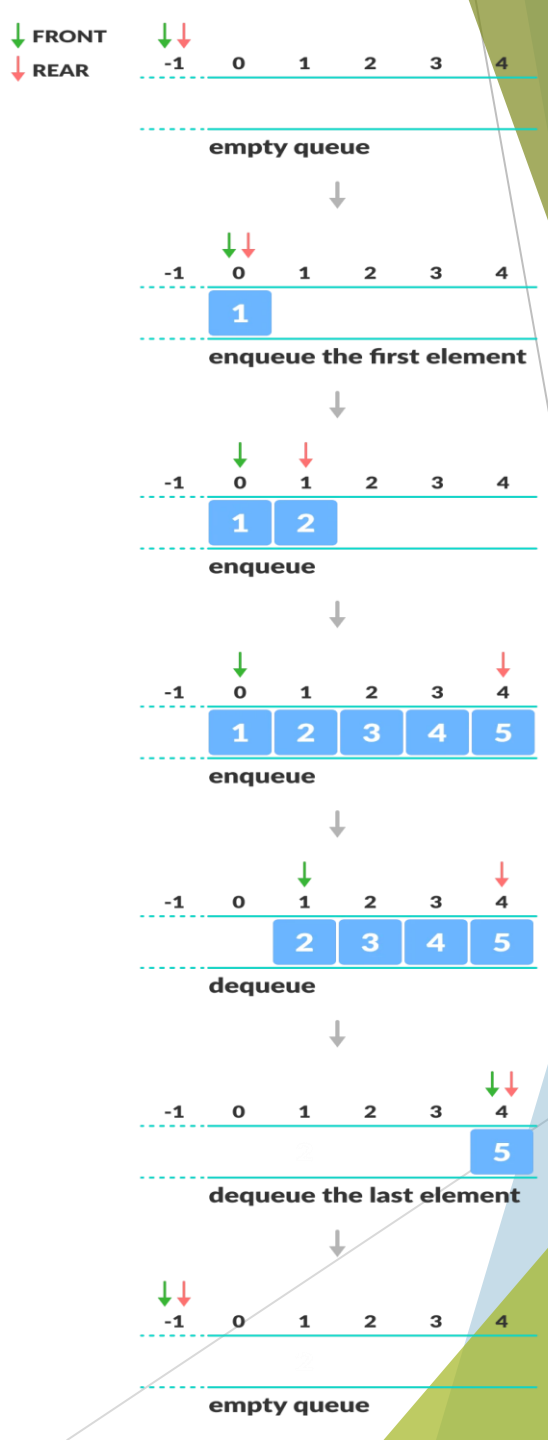
```
int peek() {
   return stack[top];
}
```

Show/Display Operation

```
void show() {
    for (i=top;i>=0;i--) {
        printf("%d\n",stack[i]);
    }
    if(isEmpty()) {
        printf("Stack is empty");
    }
}
```

Queue (ADT)

- A queue is a FIFO DS in which the element that is inserted first is the first one to be taken out
- Elements in a queue are added at one end called the **rear** and removed from the other end called the **front**
- Implemented using array/linked list



Basic Operations of Queue

- **enqueue()** – add (store) an item to the queue
- **dequeue()** – remove (access) an item from the queue
- **peek()** – Gets the element at the front of the queue without removing it
- **isfull()** – Checks if the queue is full
- **isempty()** – Checks if the queue is empty
- **show()** – Display the queue

Array Implementation of Linear Queue

Isfull Operation

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

Isempty Operation

```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Array Implementation of Linear Queue…

Enqueue Operation

```
void enqueue(int data) {
    if(isfull()) {
        printf("\nQueue is Full!!");
    }
    else {
    if (front == -1)
        front = 0;
    rear++;
    queue[rear] = data;
    printf("\nInserted -> %d", value);
}
```

Dequeue Operation

```
void dequeue() {
    if(isempty()) {
        printf("\nQueue is Empty!!");
    }
    else {
     printf("\nDeleted : %d", queue[front]);
     front++;
     if (front > rear)
        front = rear = -1;
     }
}
```

Array Implementation of Linear Queue…

Peek Operation

```
int getFront() {
    return (isEmpty())
            ? INT_MIN
            : queue[front];
}
int getRear() {
    return (isEmpty())
            ? INT_MIN
            : queue[rear];
}
```

Show/Display Operation

```
void show() {
  if (rear == -1)
    printf("\nQueue is Empty!!!");
  else {
    int i;
    printf("\nQueue elements are:\n");
    for (i = front; i <= rear; i++)
      printf("%d\t", queue[i]);
  }
}
```

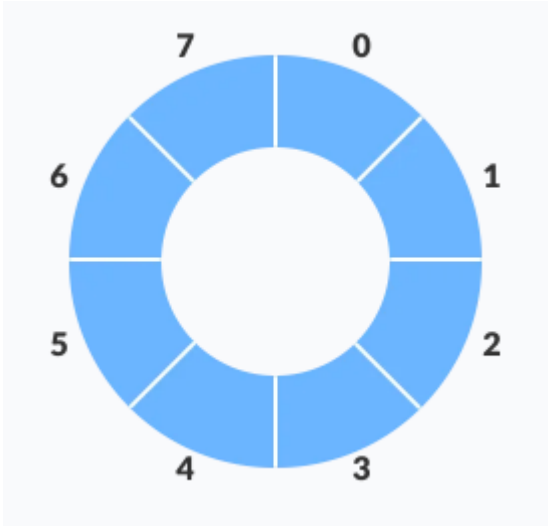
Types of Queues

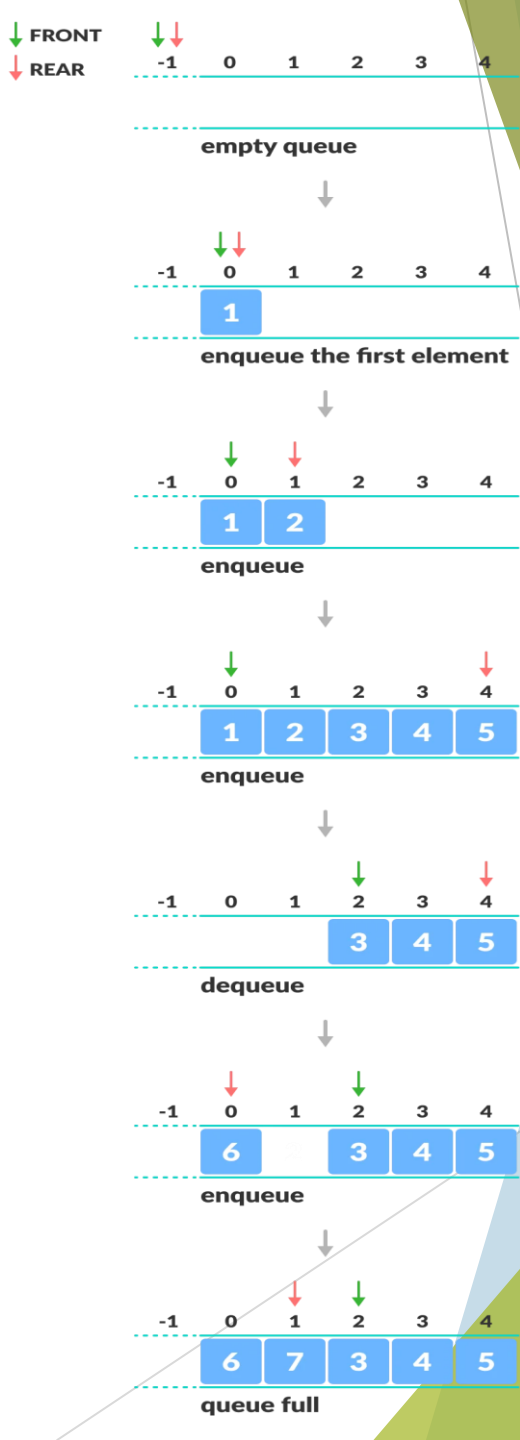
- Circular Queue
- Priority Queue
- Double Ended Queue

Circular Queue (Ring Buffer)

- It is also based on FIFO concept except that the *last position is connected to the first position*

- Circular queue solves the major limitation of the linear queue and has better memory utilization





Array Implementation of Circular Queue

```
int items[SIZE];
int front = -1, rear = -1;

// Check if the queue is full
int isFull() {
    if ((front == rear + 1) ||
    (front == 0 && rear == SIZE - 1))
      return 1;
    return 0;
}
```

```
// Check if the queue is empty
int isEmpty() {
      if (front == -1)
       return 1;
      return 0;
}
```

Array Implementation of Circular Queue…

```
// Adding an element
void enQueue(int element) {
      if (isFull())
       printf("\n Queue is full!! \n");
      else {
       if (front == -1)
            front = 0;
       rear = (rear + 1) % SIZE;
       items[rear] = element;
       printf("\n Inserted -> %d", element);
       }
 }
```

Array Implementation of Circular Queue…

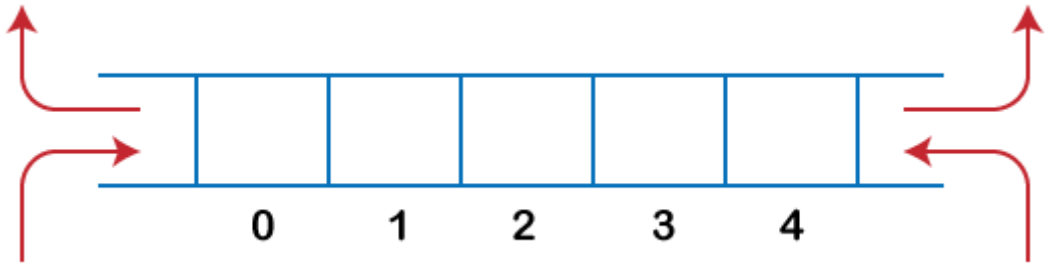
```
// Removing an element
void deQueue() {
    int element;
    if (isEmpty()) {
        printf("\n Queue is empty !! \n");
        return (-1);
    }
    else {
        element = items[front];
        if (front == rear) {
         front = -1;
         rear = -1;
        }
        else
         front = (front + 1) % SIZE;
        printf("\n Deleted element -> %d \n",
element);
  }
}
```

Array Implementation of Circular Queue…

```
// Display the queue
void show() {
      int i;
      if (isEmpty())
       printf(" \n Empty Queue\n");
      else {
       printf("\n Elements -> ");
       for (i = front; i != rear; i = (i + 1) % SIZE) {
            printf("%d ", items[i]);
       }
       printf("%d ", items[i]);
      }
}
```

Double Ended Queue (Deque)

- In a deque(pronounced *deck*), insertion and removal of elements can be performed from either front or rear
- This hybrid linear structure provides all the capabilities of **stacks** and **queues** in a single data structure
- Implemented using *circular array* and *doubly linked list*



Types of Deque

- Input Restricted Deque
  - In this deque, input is restricted at a single end but allows deletion at both the ends

- Output Restricted Deque:
  - In this deque, output is restricted at a single end but allows insertion at both the ends

Array Implementation of Deque [Extra]

```
// Insert the value from the front
void enqueue_front(int x) {
    if((f==0 && r==size-1) || (f==r+1)) {
        printf("Deque is full");
    }
    else if((f==-1) && (r==-1)) {
        f=r=0;
        deque[f]=x;
    }
    else if(f==0) {
        f=size-1;
        deque[f]=x;
    }
    else  {
        f=f-1;
        deque[f]=x;
    }
}
```

Array Implementation of Deque… [Extra]

```
// Insert the value from the rear
void enqueue_rear(int x) {
    if((f==0 && r==size-1) || (f==r+1)) {
        printf("Deque is full");
    }
    else if((f==-1) && (r==-1)) {
        f=r=0;
        deque[r]=x;
    }
    else if(r==size-1) {
        r=0;
        deque[r]=x;
    }
    else {
        r++;
        deque[r]=x;
    }
}
```

Array Implementation of Deque… [Extra]

```
// Delete element from the front
void dequeue_front() {
    if((f == -1) && (r == -1)) {
            printf("Deque is empty");
    }
    printf("\nThe deleted element is %d", deque[f]);
    else if(f == r) {
            r = f = -1;
    }
    else if(f == (size-1)) {
      f = 0;
    }
    else {
      f = f+1;
    }
}
```

Array Implementation of Deque… [Extra]

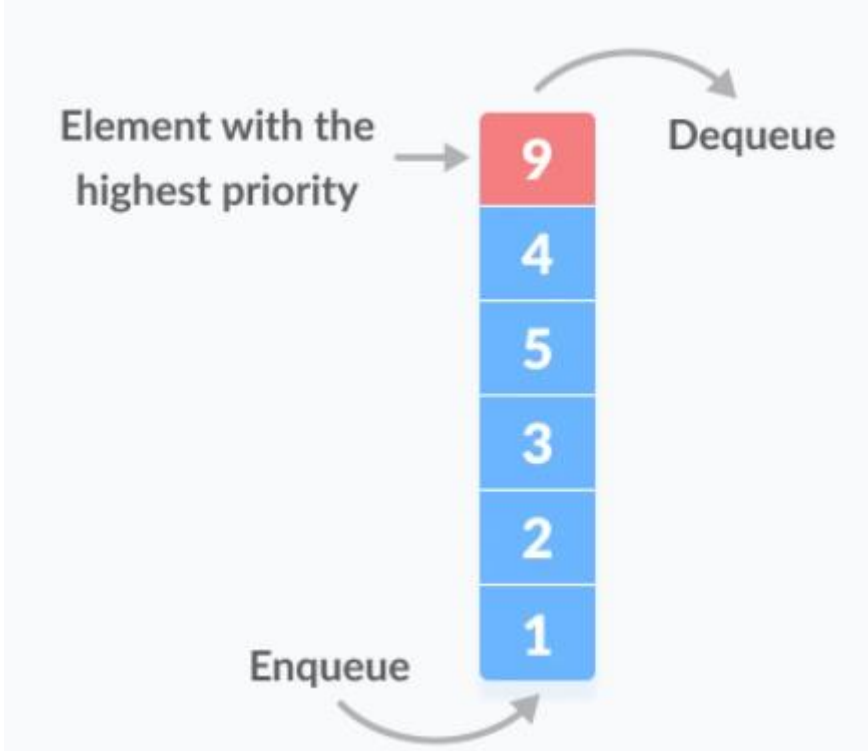
```
// Delete element from the rear
void dequeue_rear() {
    if((f == -1) && (r == -1)) {
            printf("Deque is empty");
    }
    printf("\nThe deleted element is %d", deque[r]);
    else if(f == r) {
            f = r = -1;
    }
    else if(r == 0) {
        r = size-1;
    }
    else {
        r = r-1;
    }
}
```

Priority Queue

- Special type of queue in which each element is associated with a priority value and is served according to its priority
- Same priority elements are served based on their order in the queue
- Insertion occurs based on the arrival of the values and removal occurs based on priority
- Implemented using *arrays*, *linked list* and *binary heap*

Array Implementation of Priority Queue

- Can be implemented by having separate array for storing priority and data
- Another way of implementing would be by adding new data in ascending or descending order by priority



Any Questions?