Stacks

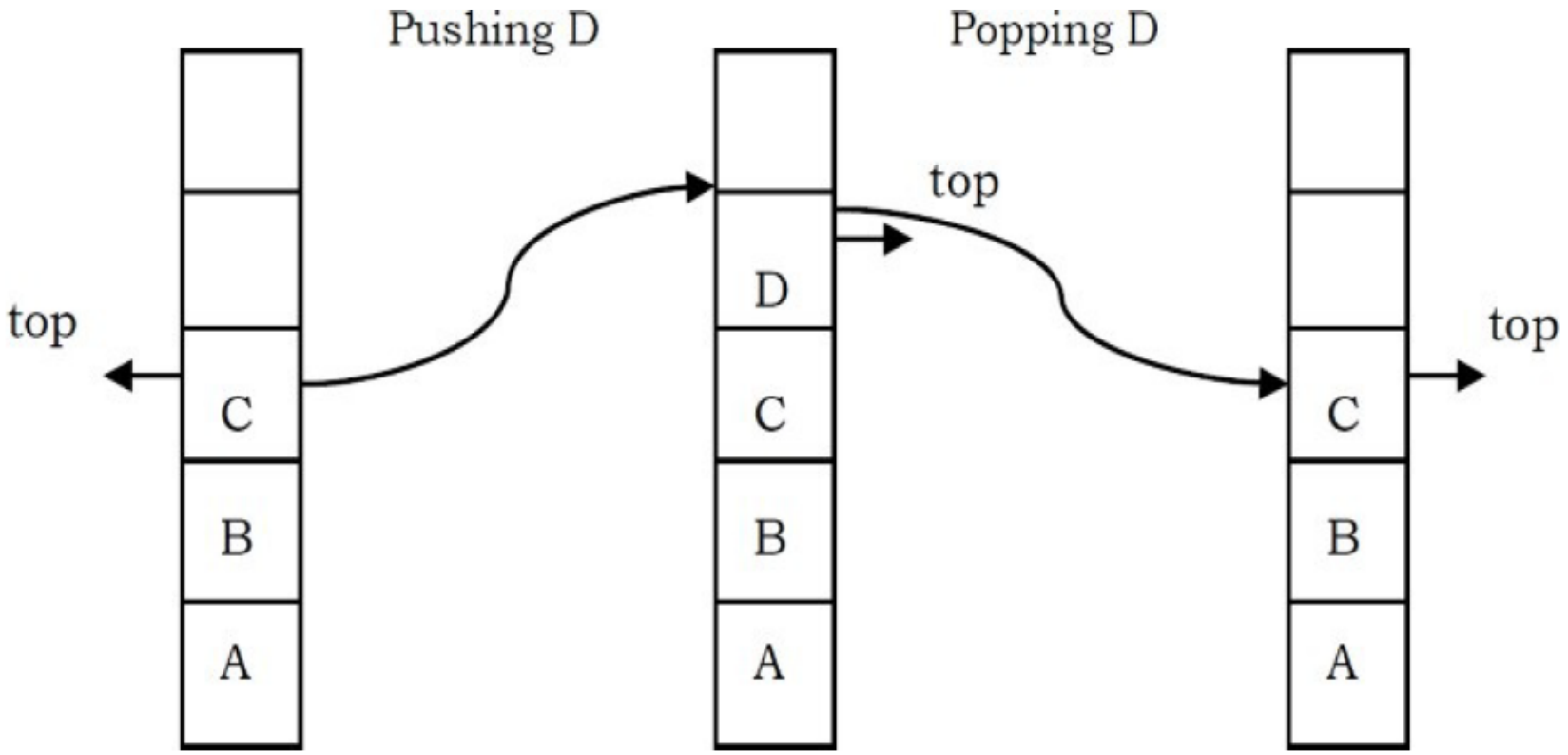
What is a Stack?

- A stack is a simple data structure used for storing data (similar to Linked Lists).
- In a stack, the order in which the data arrives is important.
- A **pile of plates** in a cafeteria is a **good example of a stack.**
  - The plates are added to the stack as they are cleaned and they are placed on the top. When a plate, is required it is taken from the top of the stack. The first plate placed on the stack is the last one to be used.

Definition

- A *stack* is an ordered list in which insertion and deletion are done at one end, called *top*. The last element inserted is the first one to be deleted. Hence, it is called the Last in First out (LIFO) or First in Last out (FILO) list.

- Special names are given to the two changes that can be made to a stack.
- When an element is inserted in a stack, the concept is called *push*, and when an element is removed from the stack, the concept is called *pop*.
- Trying to pop out an empty stack is called *underflow* and trying to push an element in a full stack is called *overflow*.
- Generally, we treat them as exceptions. As an example, consider the snapshots of the stack.


Pushing D
Popping D
top
top
top
D
C
C
B
A
B
A
C
B
A

How Stacks are used

- Consider a working day in the office. Let us assume a developer is working on a long-term project. The manager then gives the developer a new task which is more important. The developer puts the long-term project aside and begins work on the new task. The phone rings, and this is the highest priority as it must be answered immediately. The developer pushes the present task into the pending tray and answers the phone.

- When the call is complete the task that was abandoned to answer the phone is retrieved from the pending tray and work progresses. To take another call, it may have to be handled in the same manner, but eventually the new task will be finished, and the developer can draw the long-term project from the pending tray and continue with that.

Stack ADT

- The following operations make a stack an ADT. For simplicity, assume the data is an integer type.
- **Main stack operations**
  - Push (int data): Inserts *data* onto stack.
  - int Pop(): Removes and returns the last inserted element from the stack.

- **Auxiliary stack operations**
  - int Top(): Returns the last inserted element without removing it.
  - int Size(): Returns the number of elements stored in the stack.
  - int IsEmptyStack(): Indicates whether any elements are stored in the stack or not.
  - int IsFullStack(): Indicates whether the stack is full or not.

- **Exceptions**
- Attempting the execution of an operation may sometimes cause an error condition, called an exception. Exceptions are said to be “thrown” by an operation that cannot be executed. In the Stack ADT, operations pop and top cannot be performed if the stack is empty. Attempting the execution of pop (top) on an empty stack throws an exception. Trying to push an element in a full stack throws an exception.

Applications

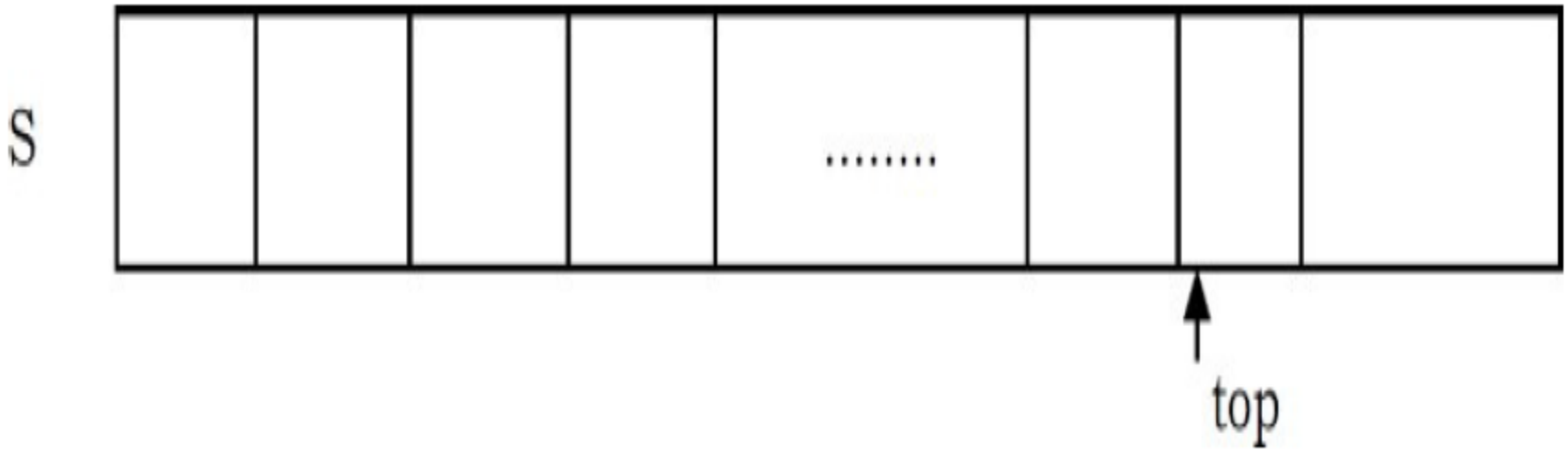
- Following are some of the applications in which stacks play an important role.
- **Direct applications**
  - Balancing of symbols
  - Infix-to-postfix conversion
  - Evaluation of postfix expression
  - Implementing function calls (including recursion)
  - Finding of spans (finding spans in stock markets, refer to *Problems* section)
  - Page-visited history in a Web browser [Back Buttons]
  - Undo sequence in a text editor
  - Matching Tags in HTML and XML

- **Indirect applications**
  - Auxiliary data structure for other algorithms (Example: Tree traversal algorithms)
  - Componen

Implementation

- There are many ways of implementing stack ADT; below are the commonly used methods.
  - Simple array based implementation
  - Dynamic array based implementation
  - Linked lists implementation

- **Simple Array Implementation**
- This implementation of stack ADT uses an array. In the array, we add elements from left to right and use a variable to keep track of the index of the top element.



- The array storing the stack elements may become full. A push operation will then throw a *full stack exception*. Similarly, if we try deleting an element from an empty stack it will throw *stack empty exception.*

```
#define MAXSIZE 10
struct ArrayStack {
        int top;
        int capacity;
        int *array;
};
struct ArrayStack *CreateStack() {
        struct ArrayStack *S = malloc(sizeof(struct ArrayStack));
        if(!S)
           return NULL;
        S→capacity = MAXSIZE;
        S→top = -1;
        S→array= malloc(S→capacity * sizeof(int));
        if(!S→array)
           return NULL;
        return S;
}
int IsEmptyStack(struct ArrayStack *S) {
        return (S→top == -1);     // if the condition is true then 1 is returned else 0 is returned
}
int IsFullStack(struct ArrayStack *S){
        //if the condition is true then 1 is returned else 0 is returned
        return (S→top == S→capacity - 1);
}
```

```
void Push(struct ArrayStack *S, int data){
        /* S→top == capacity -1 indicates that the stack is full*/
        if(IsFullStack(S))
            printf( "Stack Overflow");
        else        /*Increasing the 'top' by 1 and storing the value at 'top' position*/
            S→ array[++S→top]= data;
}
int Pop(struct ArrayStack *S){
      /* S→top == - 1 indicates empty stack*/
        if(IsEmptyStack(S)){
            printf("Stack is Empty");
            return INT_MIN;;
        }
        else /* Removing element from  'top' of the array and reducing  'top' by 1*/
            return (S→ array[S→top--]);
}
void DeleteStack(struct DynArrayStack *S){
      if(S) {
          if(S→array)
              free(S→array);
          free(S);
      }
}
```

Performance & Limitations

- **Performance**
- Let *n* be the number of elements in the stack. The complexities of stack operations with this representation can be given as:

Space Complexity (for n push operations)	O(*n*)
Time Complexity of Push()	O(1)
Time Complexity of Pop()	O(1)
Time Complexity of Size()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

- **Limitations**
- The maximum size of the stack must first be defined and it cannot be changed. Trying to push a new element into a full stack causes an implementation-specific exception.

Dynamic Array Implementation

- First, let's consider how we implemented a simple array based stack.
- We took one index variable *top* which points to the index of the most recently inserted element in the stack.
- To insert (or push) an element, we increment *top* index and then place the new element at that index.

- Similarly, to delete (or pop) an element we take the element at *top* index and then decrement the *top* index.
- We represent an empty queue with *top* value equal to –1. The issue that still needs to be resolved is what we do when all the slots in the fixed size array stack are occupied?

- **First try:** What if we increment the size of the array by 1 every time the stack is full?
  - Push(); increase size of S[] by 1
  - Pop(): decrease size of S[] by 1

- **Problems with this approach?**
- This way of incrementing the array size is too expensive. Let us see the reason for this. For example, at n = 1, to push an element create a new array of size 2 and copy all the old array elements to the new array, and at the end add the new element. At n = 2, to push an element create a new array of size 3 and copy all the old array elements to the new array, and at the end add the new element.

- Similarly, at $n = n - 1$, if we want to push an element create a new array of size n and copy all the old array elements to the new array and at the end add the new element.
- After n push operations the total time $T(n)$ ($n$umber of copy operations) is proportional to $1 + 2 + \ldots + n \approx O(n2)$.

Alternative Approach: Repeated Doubling

- Let us improve the complexity by using the array *doubling* technique. If the array is full, create a new array of twice the size, and copy the items. With this approach, pushing *n* items takes time proportional to *n* (not *n*2).

- For simplicity, let us assume that initially we started with n = 1 and moved up to n = 32. That means, we do the doubling at 1,2,4,8,16. The other way of analyzing the same approach is: at n = 1, if we want to add (push) an element, double the current size of the array and copy all the elements of the old array to the new array.

- At n = 1, we do 1 copy operation, at n = 2, we do 2 copy operations, and at n = 4, we do 4 copy operations and so on.
- By the time we reach n = 32, the total number of copy operations is 1+2 + 4 + 8+16 = 31 which is approximately equal to 2n value (32).
- If we observe carefully, we are doing the doubling operation *logn* times.
- Now, let us generalize the discussion.
- For n push operations we double the array size *logn* times.
- That means, we will have *logn* terms in the expression below.
- The total time $T(n)$ of a series of n push operations is proportional to

- $T(n)$ is $O(n)$ and the amortized time of a push operation is $O(1)$.

$$1+2+4+8\ldots+\frac{n}{4}+\frac{n}{2}+n = n+\frac{n}{2}+\frac{n}{4}+\frac{n}{8}\ldots+4+2+1$$

$$= n\left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}\ldots+\frac{4}{n}+\frac{2}{n}+\frac{1}{n}\right)$$

$$= n(2) \approx 2n = O(n)$$

```
struct DynArrayStack {
    int top;
    int capacity;
    int *array;
};
struct DynArrayStack *CreateStack(){
    struct DynArrayStack *S = malloc(sizeof(struct DynArrayStack));
    if(!S)
        return NULL;
    S→capacity = 1;
    S→top = -1;
    S→array = malloc(S→capacity * sizeof(int));  // allocate an array of size 1 initially
    if(!S→array)
        return NULL;
    return S;
}
```

```
int IsFullStack(struct DynArrayStack *S){
    return (S->top == S->capacity-1);
}
void DoubleStack(struct DynArrayStack *S){
    S->capacity *= 2;
    S->array = realloc(S->array, S->capacity * sizeof(int));
}
void Push(struct DynArrayStack *S, int x){
    // No overflow in this implementation
    if(IsFullStack(S))
        DoubleStack(S);
    S->array[++S->top] = x;
}
```

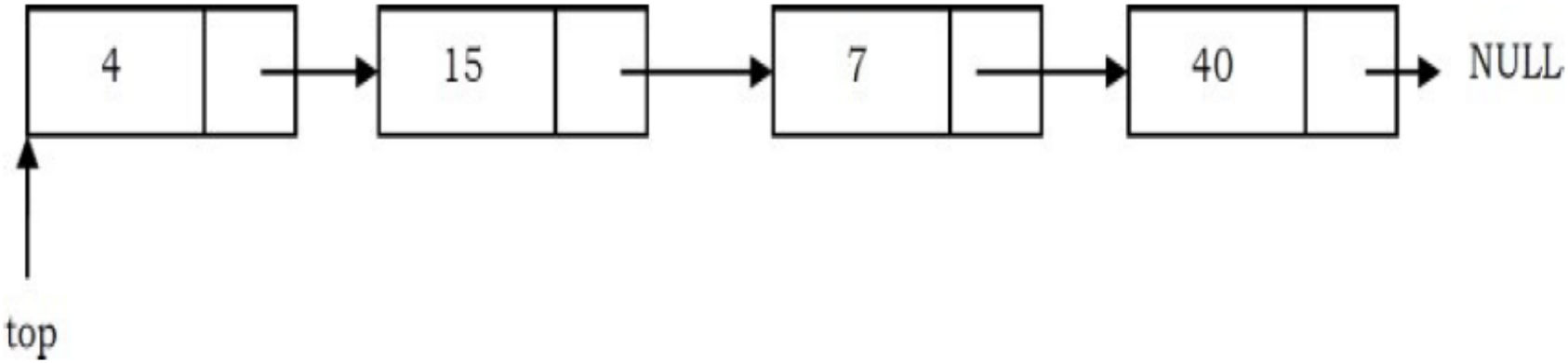
```
int IsEmptyStack(struct DynArrayStack *S){
    return S→top == -1;
}
int Top(struct DynArrayStack *S){
    if(IsEmptyStack(S))
        return INT_MIN;
    return S→array[S→top];
}
```

```
int Pop(struct DynArrayStack *S){
    if(IsEmptyStack(S))
        return INT_MIN;
    return S→array[S→top--];
}
void DeleteStack(struct DynArrayStack *S){
    if(S) {
        if(S→array)
            free(S→array);
        free(S);
    }
}
```

- **Performance**
- Let *n* be the number of elements in the stack. The complexities for operations with this representation can be given as:
- **Note:** Too many doublings may cause memory

Space Complexity (for *n* push operations)	O(*n*)
Time Complexity of CreateStack()	O(1)
Time Complexity of PushQ	O(1) (Average)
Time Complexity of PopQ	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmpryStackf)	O(1))
Time Complexity of IsFullStackf)	O(1)
Time Complexity of DeleteStackQ	O(1)

Linked List Implementation



- The other way of implementing stacks is by using Linked lists. Push operation is implemented by inserting element at the beginning of the list. Pop operation is implemented by deleting the node from the beginning (the header/top node).

```
struct ListNode{
    int data;
    struct ListNode *next;
};
struct Stack *CreateStack(){
    return NULL;
}
void Push(struct Stack **top, int data){
    struct Stack *temp;
    temp = malloc(sizeof(struct Stack));
    if(!temp)
        return NULL;
    temp→data = data;
    temp→next = *top;
    *top = temp;
}
```

```
int IsEmptyStack(struct Stack *top){
    return top == NULL;
}
int Pop(struct Stack **top){
    int data;
    struct Stack *temp;
    if(IsEmptyStack(top))
        return INT_MIN;
    temp = *top;
    *top = *top→next;
    data = temp→data;
    free(temp);
    return data;
}
```

```
int Top(struct Stack * top){
    if(IsEmptyStack(top))
        return INT_MIN;
    return top→next→data;
}
void DeleteStack(struct Stack **top){
    struct Stack *temp, *p;
    p = *top;
    while( p→next) {
        temp = p→next;
        p→next = temp→next;
        free(temp);
    }
    free(p);
}
```

- **Performance**
- Let *n* be the number of elements in the stack. The complexities for operations with this representation can be given as:

Space Complexity (for *n* push operations)	O(*n*)
Time Complexity of CreateStack()	O(1)
Time Complexity of Push()	O(1) (Average)
Time Complexity of Pop()	O(1)
Time Complexity of Top()	O(1)
Time Complexity of IsEmptyStack()	O(1)
Time Complexity of DeleteStack()	O(*n*)

Comparison of Implementations

- **Comparing Incremental Strategy and Doubling Strategy**
- We compare the incremental strategy and doubling strategy by analyzing the total time $T(n)$ needed to perform a series of n push operations. We start with an empty stack represented by an array of size 1.
- We call *amortized* time of a push operation is the average time taken by a push over the series of operations, that is, $T(n)/n$.
- **Incremental Strategy**
- The amortized time (average time per operation) of a push operation is $O(n)$ [$O(n2)/n$].
- **Doubling Strategy**
- In this method, the amortized time of a push operation is O(1) [$O(n)/n$].

- **Comparing Array Implementation and Linked List Implementation**
- **Array Implementation**
  - Operations take constant time.
  - Expensive doubling operation every once in a while.
  - Any sequence of n operations (starting from empty stack) – *"amortized"* bound takes time proportional to n.
- **Linked List Implementation**
  - Grows and shrinks gracefully.
  - Every operation takes constant time O(1).
  - Every operation uses extra space and time to deal with references.

Queues

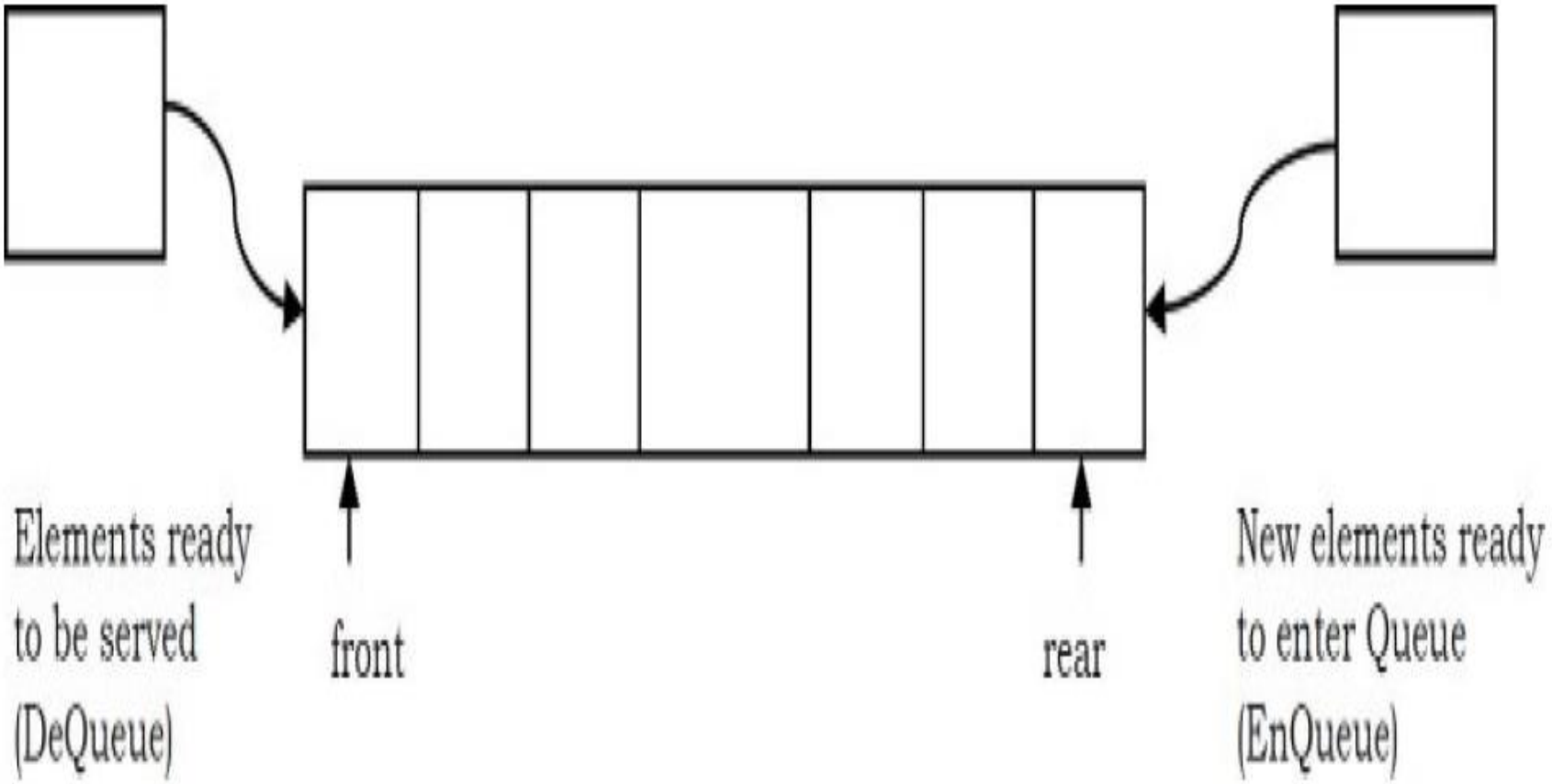
What is a Queue?

- A queue is a data structure used for storing data.
- In queue, the **order** in which data arrives **is important**.
- A queue is a line of people or things waiting to be served in sequential order starting at the beginning of the line or sequence.

- **Definition:** A *queue* is an ordered list in which **insertions** are done at one end (***rear***) and **deletions** are done at other end (***front***).
- The **first element** to be **inserted** is the **first** one to be **deleted**.
- Hence, it is called **First in First out (FIFO)** or **Last in Last out (LILO)** list.

- When an element is **inserted** in a queue, the concept is called ***EnQueue***,
- When an element is **removed** from the queue, the concept is called ***DeQueue***.

- ***DeQueueing*** **an empty queue** is called ***underflow***
- ***EnQueuing*** an element in a full queue is called ***overflow***.
- Generally, we treat them as exceptions. As an example, consider the snapshot of the queue.


Elements ready
to be served
(DeQueue)
front
rear
New elements ready
to enter Queue
(EnQueue)

How are Queues Used?

- The concept of a queue can be explained by observing a line at a reservation counter.
- When we enter the line we stand at the end of the line and the person who is at the front of the line is the one who will be served next. He will exit the queue and be served.

- As this happens, the next person will come at the head of the line, will exit the queue and will be served.
- As each person at the head of the line keeps exiting the queue, we move towards the head of the line.
- Finally we will **reach the head** of the line and we will **exit the queue** and be served.
- This behavior is very useful in cases where there is a need to maintain the order of arrival.

Queue ADT

- The following operations make a queue an ADT.
- Insertions and deletions in the queue must follow the FIFO scheme.
- For simplicity we assume the elements are integers.
- **Main Queue Operations**
  - **EnQueue(int data): Inserts** an element at the **end of the queue**
  - **int DeQueue(): Removes** and returns the element at the **front** of the queue

- **Auxiliary Queue**
  - **int Front()**: **Returns** the **element at the front** without removing it
  - **int QueueSize()**: **Returns** the **number of elements** stored in the queue
  - **int IsEmptyQueueQ: Indicates** whether **no elements are stored** in the queue or not

Exceptions

- Similar to other ADTs, executing *DeQueue* on an empty queue throws an *“Empty Queue Exception”* and executing *EnQueue* on a full queue throws *“Full Queue Exception”.*

Applications

- Following are some of the applications that use queues.
- **Direct Applications**
  - **Operating systems schedule jobs** (with equal priority) in the order of arrival (e.g., a **print queue**).
  - Simulation of real-world queues such as **lines at a ticket counter** or any other first-come first-served scenario requires a queue.
  - **Multiprogramming**.
  - **Asynchronous data transfer** (file IO, pipes, sockets).
  - **Waiting times** of customers at call center.
  - **Determining number of cashiers** to have at a supermarket.
- **Indirect Applications**
  - Auxiliary data structure for algorithms
  - Component of other data structures

Linear Queue using Array

- Input: An element ITEM that has to be inserted.
- Output: The ITEM is at the REAR of the queue.
- Data Structure: Q is the array representation of queue structure; two pointers FRONT and REAR of the queue Q are known.

Algorithm: ENQUEUE(ITEM)

- Steps:
- If(REAR == N) then

```
        Printf("Queue is full");
        Exit;
  Else
  if ( REAR == 0) and (FRONT=0)
        FRONT =1
  Endif
        REAR = REAR + 1
        Q(REAR) = ITEM
  EndIf
  Stop
```

Algorithm: DEQUEUE

- Input: A Queue with elements, FRONT and REAR are the pointers of the queue Q.
- Output: the deleted element is stored in ITEM
- Data structures : Q is the array representation of queue structure.

Steps:

1. If (FRONT = 0) then
    1. Print "Queue is empty"
    2. Exit
2. Else
    1. ITEM = *Q*[FRONT] //Get the element
    2. If (FRONT = REAR) //When queue contains single element
        1. REAR = 0 //The queue becomes empty
        2. FRONT = 0
    3. Else
        1. FRONT = FRONT + 1
    4. EndIf
3. EndIf
4. Stop

- Input: An element ITEM to be inserted into the circular queue.
- Output: Circular queue with the ITEM at FRONT. If the queue is not full
- Data structure: CQ be the array to represent the circular queue. Two pointers FRONT and REAR are known.

Circular Queue EnQueue

Steps:

```
1. If (FRONT = 0) then                          //When queue is empty
    1. FRONT = 1
    2. REAR = 1
    3. CQ[FRONT] = ITEM
2. Else                                         //Queue is not empty
    1. next = (REAR MOD LENGTH) + 1
    2. If (next ≠ FRONT) then                   //If queue is not full
        1. REAR = next
        2. CQ[REAR] = ITEM
    3. ELSE
        1. Print "Queue is full"
    4. EndIf
3. EndIf
4. Stop
```

Circular Queue Dequeue

Algorithm DECQUEUE( )
Input: A queue CQ with elements. Two pointers FRONT and REAR are known.
Output: The deleted element is ITEM if the queue is not empty.
Data structures: CQ is the array representation of circular queue.

Steps:

```
1. If (FRONT = 0) then
   1. Print "Queue is empty"
   2. Exit
2. Else
   1. ITEM = CQ[FRONT]
   2. If (FRONT = REAR) then                    //If the queue contains single element
      1. FRONT = 0
      2. REAR = 0
   3. Else
      1. FRONT = (FRONT MOD LENGTH) + 1
   4. EndIf
3. EndIf
4. Stop
```

Linked List Enqueue

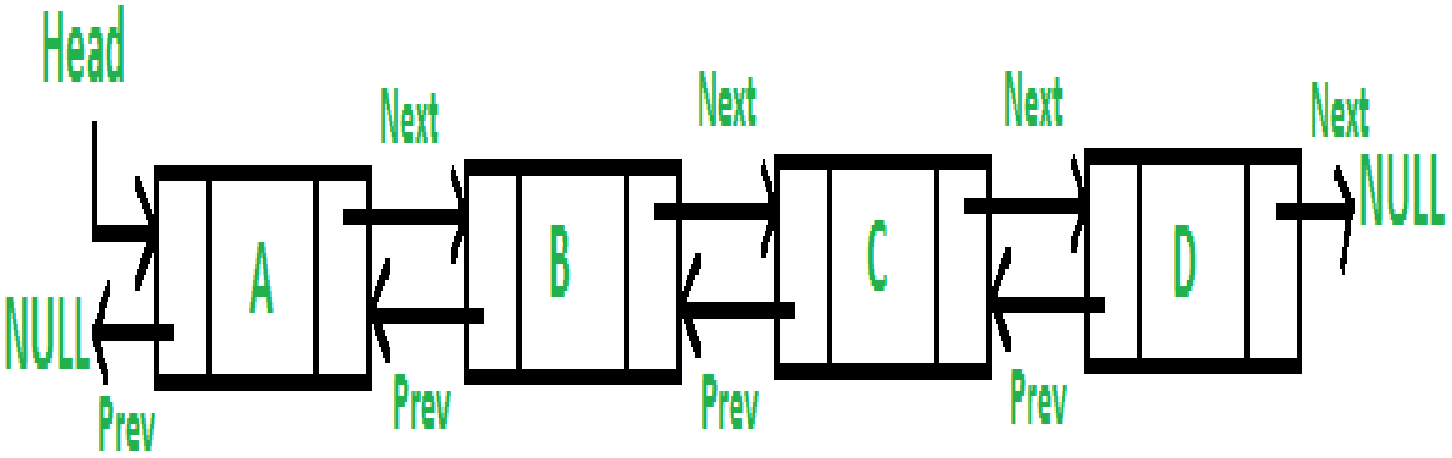
- Algorithm INSERT_DL_END(X)
- Input: X the data content of the node to be inserted.
- Output: A Double Linked List enriched with a node containing data as X at the end of the list.
- Data Structure: Double Linked list Structure whose pointer to the header node is HEADER.

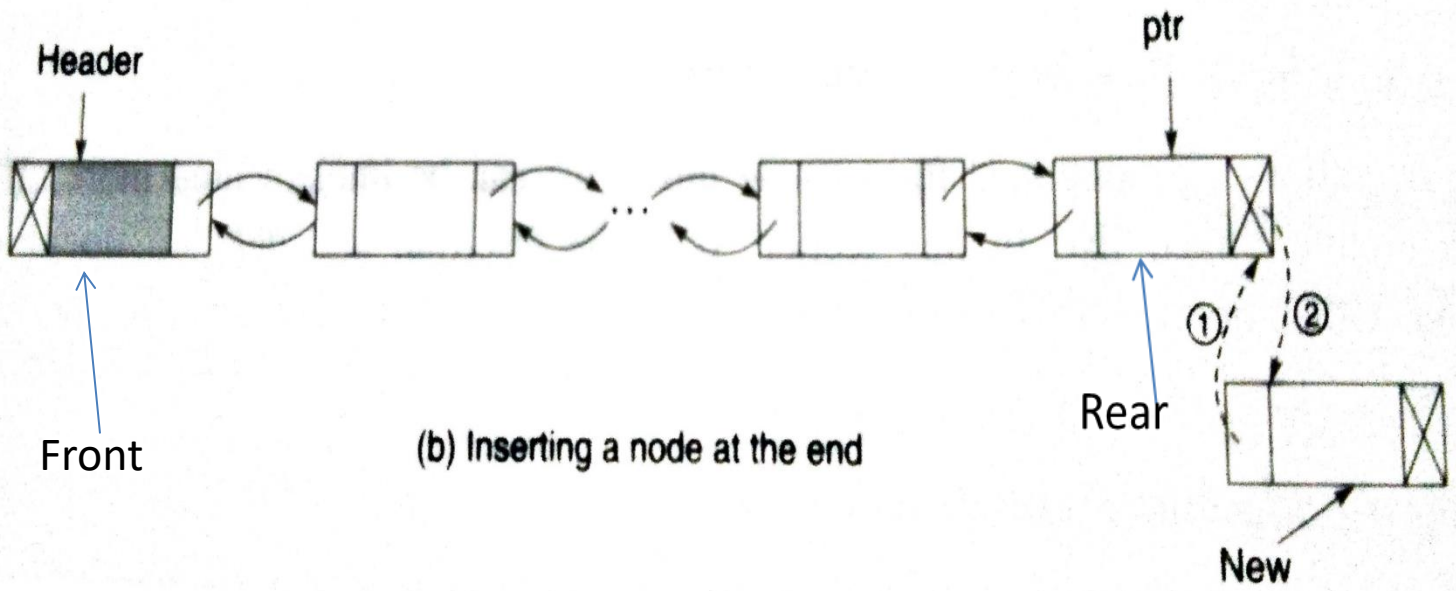
Queue using Linked List

- Disadvantage of using Array:
  - 1) Inadequate service of insertion representation with array is the rigidness of its length.
  - 2) length of queue may be predicated before and it varies abruptly.
- So, to overcome these two problems, we use linked list.
  - Hence, we use Doubly Linked List which allows to move on both sides.

- Pointers FRONT and REAR point the first node and the last node in the list.
- Two states of the queue namely, empty or it contains some element can be judged by following tests.

Insertion into a queue using doubly linked list





(b) Inserting a node at the end

- **Queue is empty**

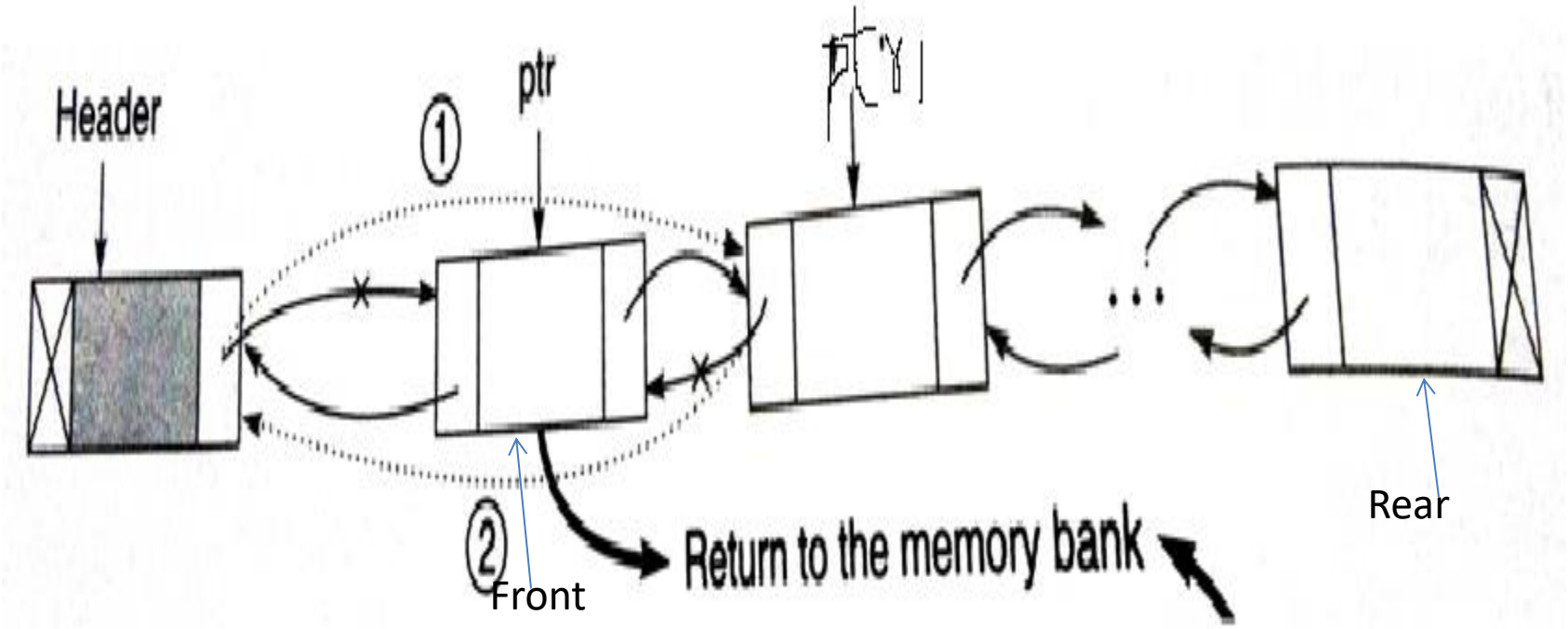
  FRONT = REAR = HEADER

  HEADER. RLINK(next) = NULL
- **Queue contains at least one element**

  HEADER.RLINK(next) = NULL

- 1.ptr = HEADER
  2.While (ptr.RLINK(next) = NULL) do //Move to the
  // Last Node
  i.ptr=ptr.RLINK(next)
  3. EndWhile
  4.new= GETNODE(NODE) //Avail a new node
  5.if(new != NULL) then //if the node is available
  i. new.LLINK(prev) = ptr //Change the pointer
  ii. ptr.RLINK(next) = NEW //Change the pointer
  iii. new.RLINK(next) = NULL //Make the new node as the last node
  iv. new.Data=X(ITEM)(ELEMENT)
  6. Else
  i. print(“Unable to Allocate Memory. Insertion is not possible”)
  7. Endif
  8. Stop

Deletion of node from Queue using Doubly Linked List



Linked List Dequeue

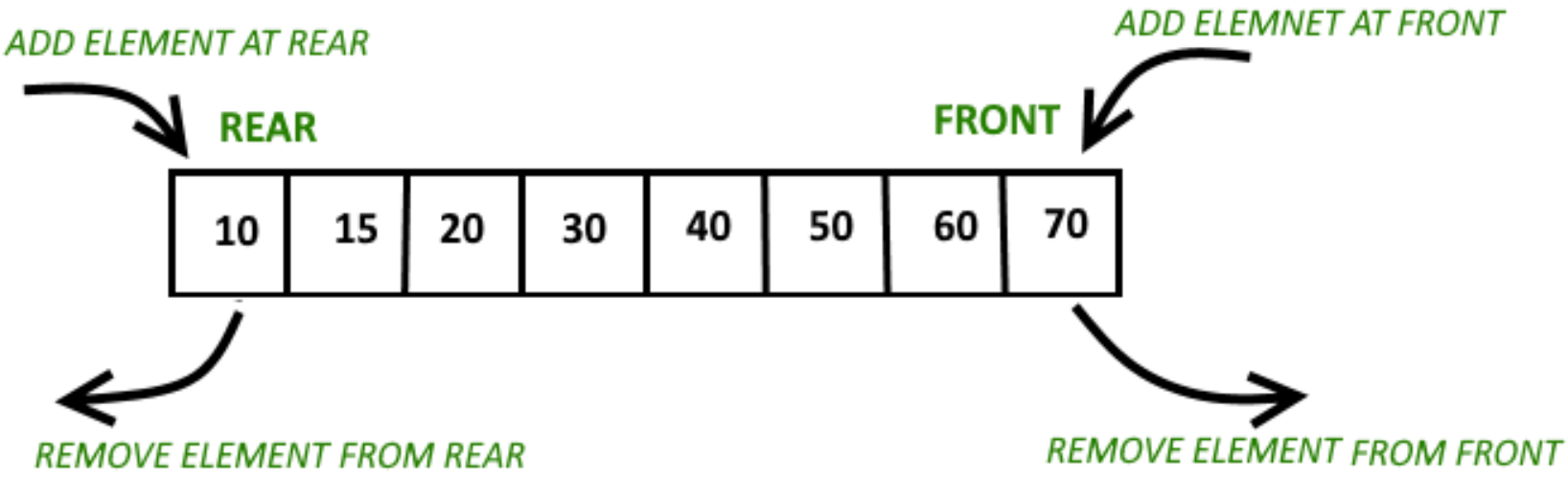
Algorithm DELETE_DL_FRONT( )
Input: A double linked list with data.
Output: A reduced double linked list.
Data structure: Double linked list structure whose pointer to the header node is HEADER.

Steps:

1. ptr = HEADER.RLINK //Pointer to the first node
2. If (ptr = NULL) then //If the list is empty
   1. Print "List is empty: No deletion is made"
   2. Exit
3. Else
   1. ptr1 = ptr.RLINK //Pointer to the second node
   2. HEADER.RLINK = ptr1 //Change the pointer
   3. If (ptr1 ≠ NULL) //If the list contains a node after the first node
      //of deletion
      1. ptr1.LLINK = HEADER //Change the pointer

Deque

- In Deque insertion and deletion can be made at either end of the structure.
- Deque means Double Ended QUEue.



- DEQUE can be used both as stack and queue.
- We can represent deque by using Double Linked List and Circular Array.
- Following four operations are possible on a deque which consists of a list of items.
  - 1. PUSHDQ(ITEM): To insert ITEM at the FRONT end of deque
  - 2.POPDQ(): To remove the FRONT item from deque.
  - 3. INJECT(ITEM): To insert ITEM at the REAR END OF DEQUE.
  - 4. EJECT(): To remove the REAR ITEM from deque.

Algorithm PUSHDQ(ITEM)

- Algorithm: PUSHDQ(ITEM)
- Input: ITEM to be inserted at the FRONT
- Output: Deque with newly inserted element ITEM if it is not full already.
- Data Structures: DQ being the circular array representation of deque.

- **Steps:**
- **1. if (FRONT=1) THEN** // If FRONT is at extreme left
  i. ahead = LENGTH
  2. ELSE // If FRONT is at extreme right or deque is
  // empty
  i. if(FRONT = LENGTH) Or(FRONT =0) Then
  a. ahead = 1 //FRONT is at an intermediate position
  else
  ahead= FRONT -1
  Endif
  3. If(ahead = REAR) then
  i. print “Deque is full”
  ii. Exit
  4. Else
  i. FRONT = ahead // push the ITEM
  ii.DQ[FRONT] = ITEM
  5. Stop

Deque Eject()

- Input: A dequeu with elements in it
- Output: the item is deleted from the REAR end.
- Data Structures: DQ being the circular array representation of deque.

Deletion in Deque(Double Ended Queue)

Steps:

```
1. If (FRONT = 0) then
   1. Print "Deque is empty"
   2. Exit
2. Else
   1. If (FRONT = REAR) then                  //The deque contains single element
      1. ITEM = DQ[REAR]
      2. FRONT = REAR = 0                     //Deque becomes empty
   2. Else
      1. If (REAR = 1) then                   //REAR is at extreme left
         1. ITEM = DQ[REAR]
         2. REAR = LENGTH
      2. Else
         1. If (REAR = LENGTH) then           //REAR is at extreme right
            1. ITEM = DQ[REAR]
            2. REAR = 1
         2. Else                              //REAR is at an intermediate position
            1. ITEM = DQ[REAR]
            2. REAR = REAR - 1
         3. EndIf
      3. EndIf
   3. EndIf
3. EndIf
4. Stop
```

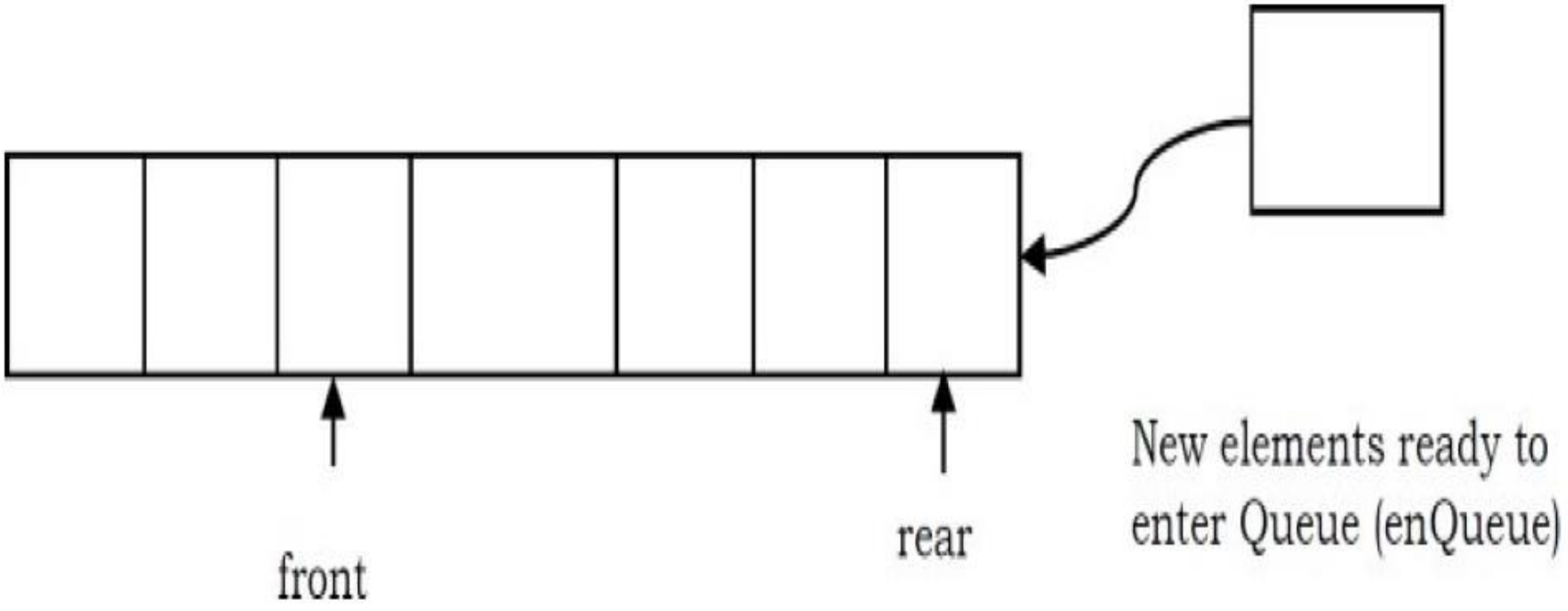
Implementation

- Implementing queue operations and some of the commonly used methods are listed below.
  - Simple circular array based implementation
  - Dynamic circular array based implementation
  - Linked list implementation

Why Circular Arrays?

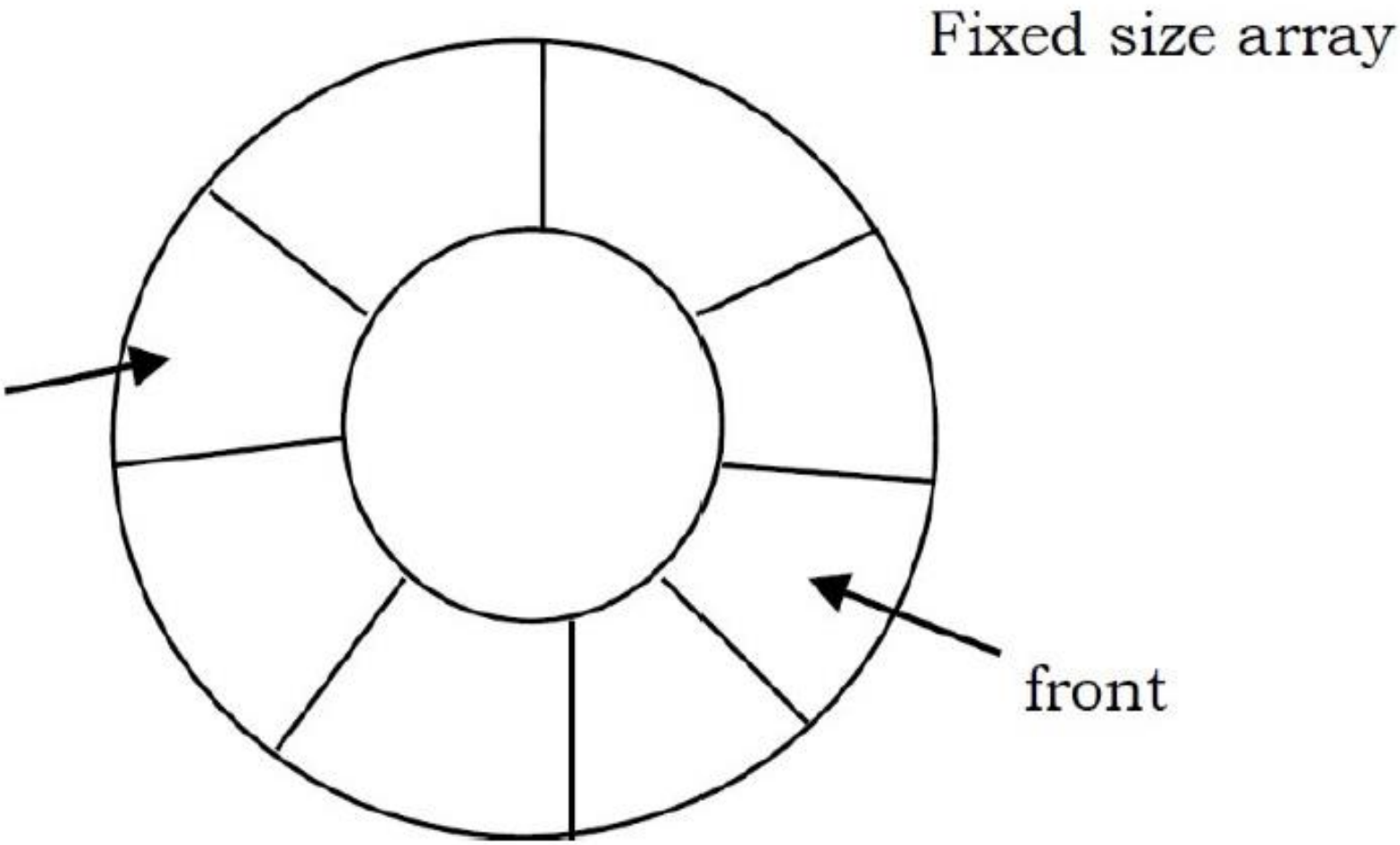
- We know that, in queues, the **insertions** are performed at **one end(rear)** and **deletions** are performed at the **other end(front)**.

- In the example shown below, it can be seen clearly that the initial slots of the array are getting wasted.
- So, simple array implementation for queue is not efficient. To solve this problem we assume the arrays as circular arrays. That means, we treat the last element and the first array elements as contiguous. With this representation, if there are any free slots at the beginning, the rear pointer can easily go to its next free slot.


front
rear
New elements ready to
enter Queue (enQueue)

- **Note:** The simple circular array and dynamic circular array implementations are very similar to stack array implementations.
- Refer to *Stacks* chapter for analysis of these implementations.

Simple Circular Array Implementation



- This simple implementation of Queue ADT uses an array. In the array, we add elements circularly and use two variables to keep track of the start element and end element.
- Generally, *front* is used to indicate the start element and *rear* is used to indicate the end element in the queue.
- The array storing the queue elements may become full.
- An *EnQueue* operation will then throw a *full queue exception*. Similarly, if we try deleting an element from an empty queue it will throw *empty queue exception.*
- **Note:** Initially, both front and rear points to -1 which indicates that the queue is empty.

```
struct ArrayQueue {
    int front, rear;
    int capacity;
    int *array;
};
```

```
struct ArrayQueue *Queue(int size) {
    struct ArrayQueue *Q = malloc(sizeof(struct ArrayQueue));
    if(!Q)
        return NULL;
    Q→capacity = size;
    Q→front = Q→rear = -1;
    Q→array= malloc(Q→capacity * sizeof(int));
    if(!Q→array)
        return NULL;
    return Q;
}
```

```
int IsEmptyQueue(struct ArrayQueue *Q) {
    // if the condition is true then 1 is returned else 0 is returned
    return (Q→front == -1);
}
```

```
int IsFullQueue(struct ArrayQueue *Q) {
    //if the condition is true then 1 is returned else 0 is returned
    return ((Q→rear +1) % Q→capacity == Q→front);
}
```

```
int QueueSize() {
    return (Q→capacity - Q→front + Q→rear + 1)% Q→capacity;
}
```

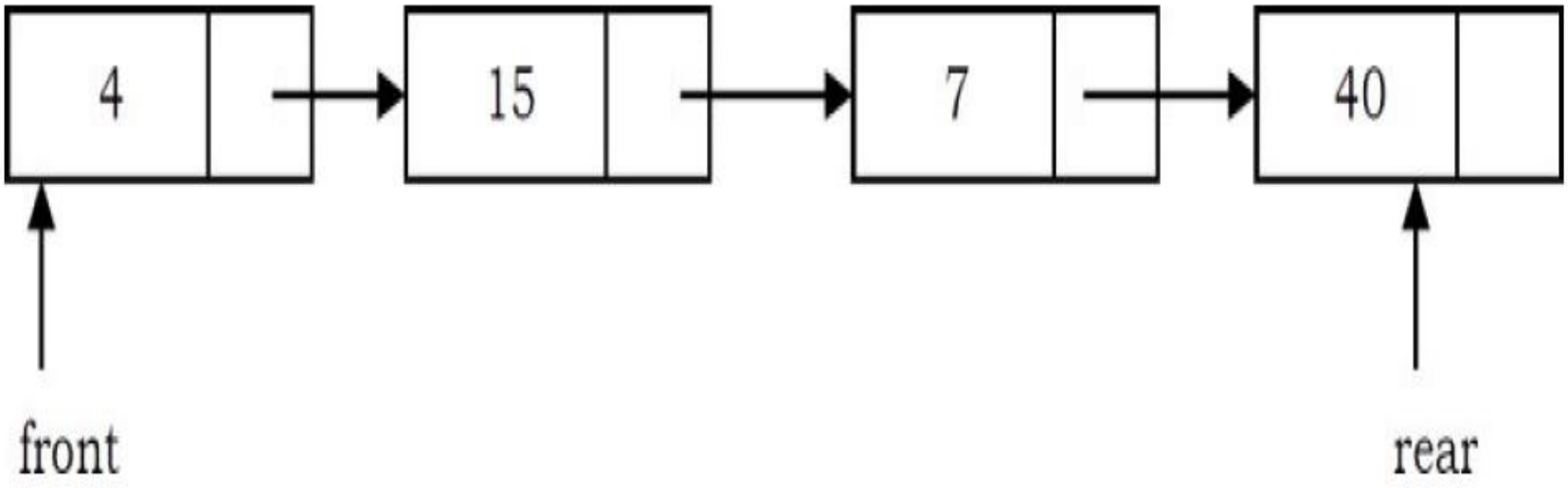
```
void EnQueue(struct ArrayQueue *Q, int data) {
    if(IsFullQueue(Q))
            printf("Queue Overflow");
    else {
            Q→rear = (Q→rear+1) % Q→capacity;
            Q→ array[Q→rear]= data;
            if(Q→front == -1)
                    Q→front = Q→rear;
    }
}
```

```
int DeQueue(struct ArrayQueue *Q) {
    int data = 0;//or element which does not exist in Queue
    if(IsEmptyQueue(Q)) {
            printf("Queue is Empty");
            return 0;
    }
    else {
        data = Q→array[Q→front];
        if(Q→front == Q→rear)
            Q→front = Q→rear = -1;
        else  Q→front = (Q→front+1) % Q→capacity;
    }
    return data;
}
```

```
void DeleteQueue(struct ArrayQueue *Q) {
    if(Q) {
        if(Q→array)
            free(Q→array);
        free(Q);
    }
}
```

Linked List Implementation

- Another way of implementing queues is by using Linked lists. *EnQueue* operation is implemented by inserting an element at the end of the list.
- *DeQueue* operation is implemented by deleting an element from the beginning of the list.


4
15
7
40
front
rear

```
struct ListNode {
    int data;
    struct ListNode *next;
};
```

```
struct Queue {
    struct ListNode *front;
    struct ListNode *rear;
};
```

```
struct Queue *CreateQueue() {
    struct Queue *Q;
    struct ListNode *temp;
    Q = malloc(sizeof(struct Queue));
    if(!Q)
        return NULL;
    temp = malloc(sizeof(struct ListNode));
    Q→front = Q→rear = NULL;
    return Q;
}
```

```
int IsEmptyQueue(struct Queue *Q) {
    // if the condition is true then 1 is returned else 0 is returned
    return (Q→front == NULL);
}
void EnQueue(struct Queue *Q, int data) {
    struct ListNode *newNode;
    newNode = malloc(sizeof(struct ListNode));
    if(!newNode)
        return NULL;
    newNode→data = data;
    newNode→next = NULL;
    if(Q→rear) Q→rear→next = newNode;
    Q→rear = newNode;

    if(Q→front == NULL)
        Q→front = Q→rear;
}
```

```
int DeQueue(struct Queue *Q) {
    int data = 0;      //or element which does not exist in Queue
    struct ListNode *temp;
    if(IsEmptyQueue(Q)) {
        printf("Queue is empty");
        return 0;
    }
    else {
        temp = Q→front;
        data = Q→front→data;
        Q→front== Q→front→next;
        free(temp);
    }
    return data;
}
```

```
void DeleteQueue(struct Queue *Q) {
    struct ListNode *temp;
    while(Q) {
        temp = Q;
        Q = Q→next;
        free(temp);
    }
    free(Q);
}
```