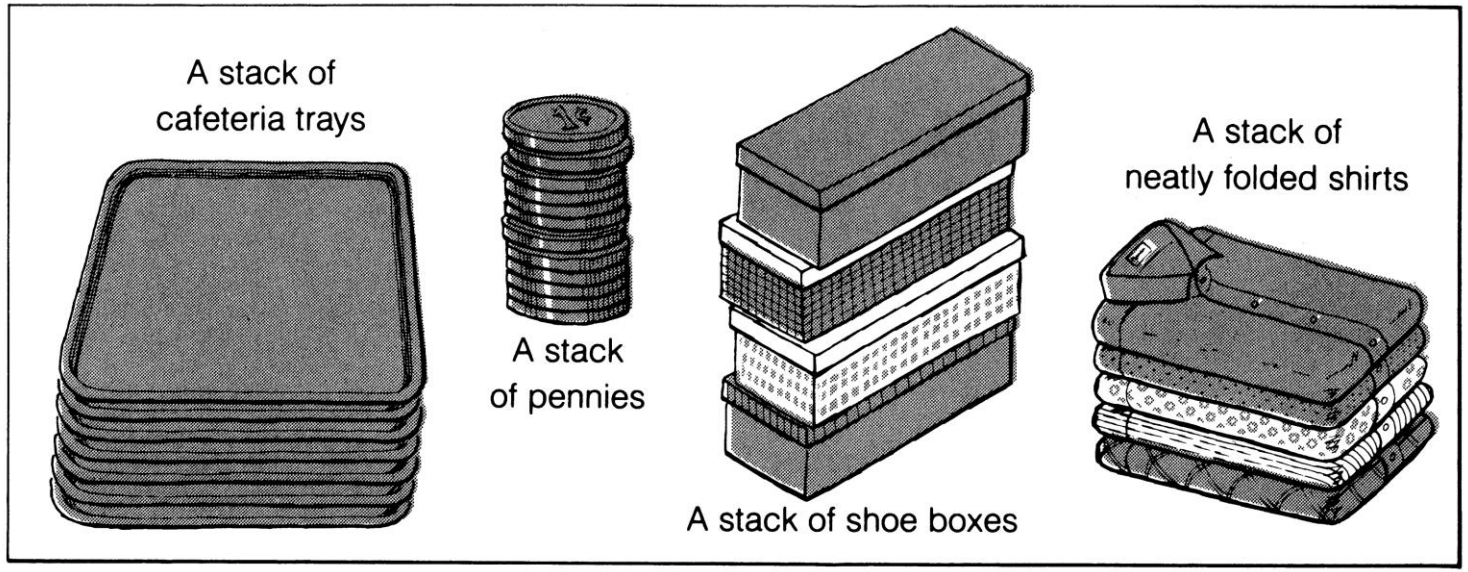
Stack Introduction

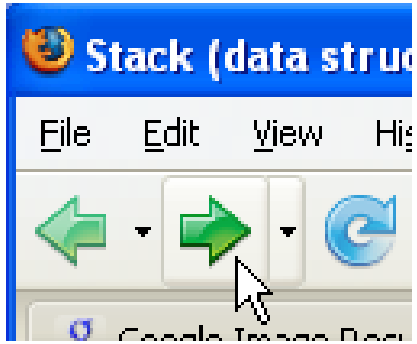
- It is a linear data structure
- It is an ordered group of homogeneous items of elements.
- It is a Last In First Out (LIFO) data structure
- i.e., recently pushed (inserted) data is popped (deleted) out from the stack.
- The last element to be added is the first to be removed (LIFO: Last In, First Out).

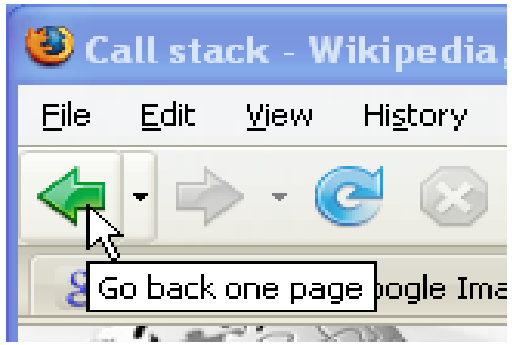
Stack Introduction

- Also called as First In Last Out (FILO)
- Elements are added to and removed from the top of the stack (the most recently added items are at the top of the stack).



Real time Uses

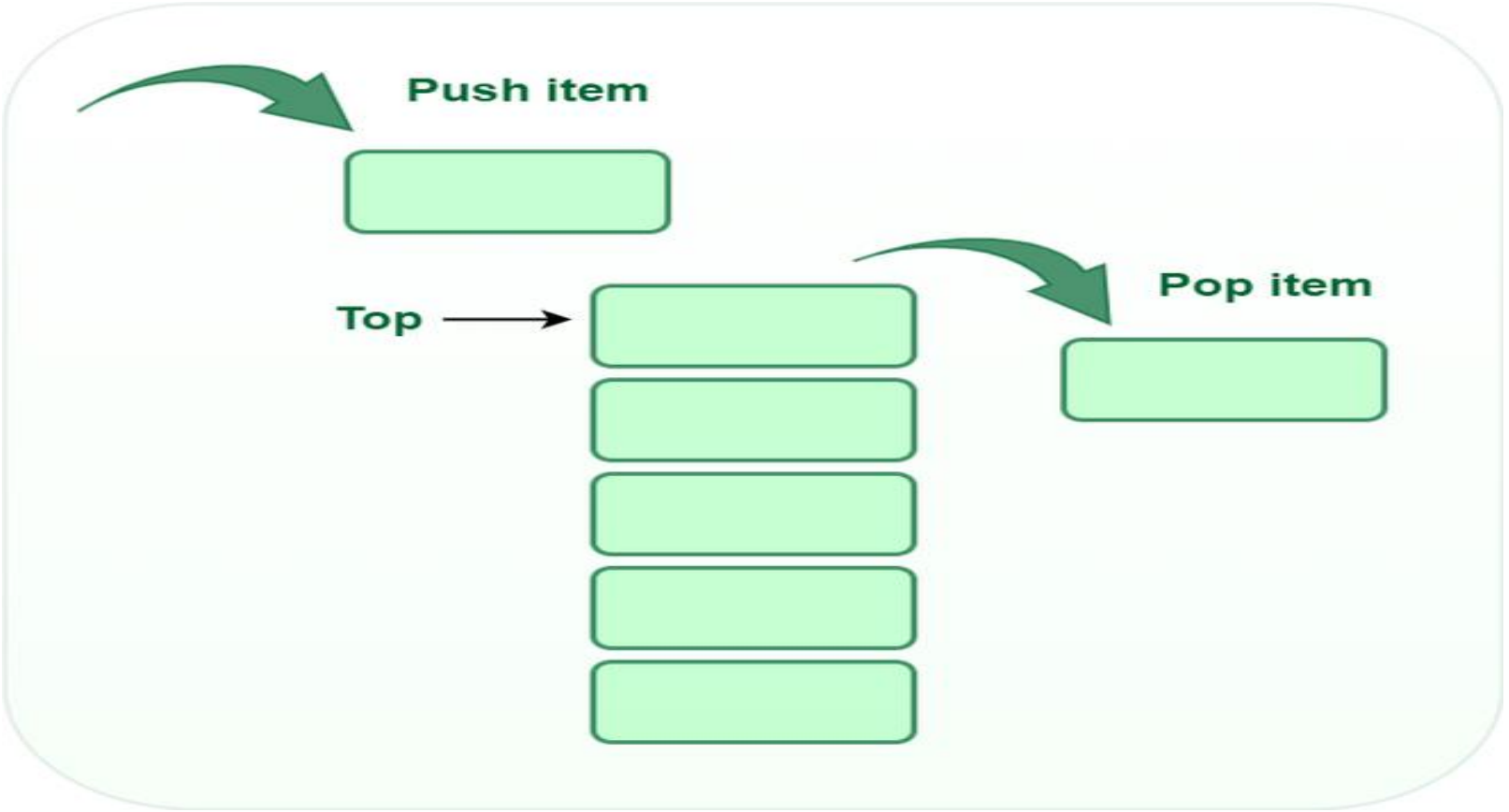




Operations on Stack

- Push (insertion)
- Pop (deletion)
- Peak or top (retrieval of top element)
- IsEmpty (check whether stack is empty)
- IsFull (check whether stack is full)

Push and PoP Operations on Stack



Push

- **Push**- Adds an item to the stack. If the stack is full, then it is said to be an Overflow condition.

```
public static void push(int value) {
    if(top>=n-1)
    {
        System.out.println("STACK is over flow");
    }
    else
    {
         top++;
        stack[top]=value;
    }
 }
```

Pop

- **Pop-** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an **Underflow condition.**

```
public static void pop() {
    if(top<=-1)
    {
        System.out.println(" Stack is under flow");
    }
    else
    {
        System.out.println(" The popped elements is "+stack[top]);
        top--;
    }
}
```

display

```
public static void display() {
    if(top>=0)
    {          System.out.println(" The elements in STACK ");
                for(int i=0; i<=top; i++)
                System.out.println(stack[i]);
          }
  else
{
System.out.println(" The STACK is empty");
    }
 }
```

Top

- Returns the top element of the stack.

Algorithm for Top:

```
begin
        return stack[top]
End procedure
```

isEmpty:

- Returns true if the stack is empty, else false.

Algorithm for isEmpty:

```
begin
        if top < 1
        return true
Else
        return false
end procedure
```

Complexity Analysis

- Time complexity

Operation	Complexity
Push()	O(1)
Pop()	O(1)
isEmpty()	O(1)
size()	O(1)

Types of Stacks:

- **Fixed Size Stack:** As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.

- **Dynamic Size Stack:** A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Implementation of Stack

- A stack can be implemented using an array or a linked list. In an array-based implementation, the push operation is implemented by incrementing the index of the top element and storing the new element at that index. The pop operation is implemented by decrementing the index of the top element and returning the value stored at that index.
- Stacks are commonly used in computer science for a variety of applications, including the evaluation of expressions, function calls, and memory management.
- In the evaluation of expressions, a stack can be used to store operands and operators as they are processed.
- In function calls, a stack can be used to keep track of the order in which functions are called and to return control to the correct function when a function returns.
- In memory management, a stack can be used to store the values of the program counter and the values of the registers in a computer program, allowing the program to return to the previous state when a function returns.

Infix to Postfix conversion

- Rules:

1. Priorities of Operators – Highest Priority - ^

Next Priority- *, /

Least priority - +, -

2. No Two Operators of same priority can stay together in the stack

3. Lowest priority operator cannot be placed before highest priority operator.

4. If any operator placed in between open and close brackets pop that operator and place it in postfix expression.

Infix to Postfix conversion

Convert the infix expression (A + B / C * (D + E)- F) to Postfix

Symbol	Stack	Postfix expression
(	(	
A	(	A
+	(+	A
B	(+	AB
/	(+/	AB
C	(+/	ABC
*	(+*	ABC/
(	(+*(	ABC/
D	(+*(	ABC/D
+	(+*(+	ABC/D
E	(+*(+	ABC/DE

Infix to Postfix conversion

Convert the infix expression (A + B / C * (D + E)- F) to Postfix

Symbol	Stack	Postfix expression
E	(+*(+	ABC/DE
)	(+*	ABC/DE+
-	(-	ABC/DE+*+
F	(-	ABC/DE+*+F
)	(-)	ABC/DE+*+F-

The postfix expression for the given infix expression (A + B / C * (D + E)- F) is ABC/DE+*+F-

Infix to Postfix conversion

Convert the infix expression A + B * C / D – F + A ^ E to Postfix

Symbol	Stack	Postfix expression
A		A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
/	+/	ABC*
D	+/	ABC*D
-	-	ABC*D/+
F	-	ABC*D/+F
+	+	ABC*D/+F-
A	+	ABC*D/+F- A

Infix to Postfix conversion

Convert the infix expression A + B * C / D – F + A ^ E to Postfix

Symbol	Stack	Postfix expression
A	+	ABC*D/+F- A
^	+^	ABC*D/+F-A
E	+^	ABC*D/+F-AE
		ABC*D/+F-AE^+

The postfix expression for the given infix expression A + B * C / D –F+A^E is ABC*D/+F-AE^+

Infix to Postfix conversion

Convert the infix expression A + B * C +D to Postfix

Symbol	Stack	Postfix expression
A		A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
+	+	ABC*+
D	+	ABC*+D
Empty		ABC*+D+

The postfix expression for the given infix expression A + B * C + D is ABC*+D+

Infix to Postfix conversion

Convert the infix expression ((a + b)-c *(d/e))+f to Postfix

Symbol	Stack	Postfix expression
(	(	
(	((	
a	((	a
+	((+	a
b	((+	ab
)	(	ab+
-	(-	ab+
C	(-	Ab+c
*	(-*	Ab+c
(	(-*(	Ab+c
D	(-*(	Ab+cd

Infix to Postfix conversion

Convert the infix expression ((a + b)-c *(d/e))+f to Postfix

Symbol	Stack	Postfix expression
D	(-*(	Ab+cd
/	(-*(/	Ab+cd
E	(-*(/	Ab+cdE
)	(-*	Ab+cde/
)	pop star and minus	Ab+cde/*-
+	+	Ab+cde/*-
F	+	Ab+cde/*-F+

Postfix to Infix conversion

Iterate the given expression from left to right, one character at a time.

Step 1 : If a character is operand, push it to stack.

Step 2: If a character is an operator,

> if there are fewer than 2 values on the stack
>
> give error "insufficient values in expression" goto Step 4
>
> else
>
> pop 2 operands from stack
>
> create a new string and by putting the operator between operands.
>
> push this string into stack
>
> Repeat Steps 1 and 2

Step 3: At last there will be only one value or one string in the stack which will be our infix expression

Step 4: Exit

Postfix to Infix conversion Example

Convert Postfix expression AB-DE+F*/ to infix expression

Token	Stack	Action
A	A	Push A into stack
B	A,B	Push B into stack
-	A-B	Pop A and B from stack put – symbol in between A, B
D	A-B, D	Push D into stack
E	A-B, D,E	Push E into stack
+	A-B,D+E	Pop D and From stack put + symbol in between D,E
F	A-B,D+E, F	Push F into stack
*	A-B, D+E*F	Pop D+E, F From stack and put * symbol in between D+E, F
/	(A-B)/(D+E*F)	Pop (A-B), (D+E*F) from stack and put / symbol in between (A-B), (D+E*F)

Postfix to Infix conversion Example

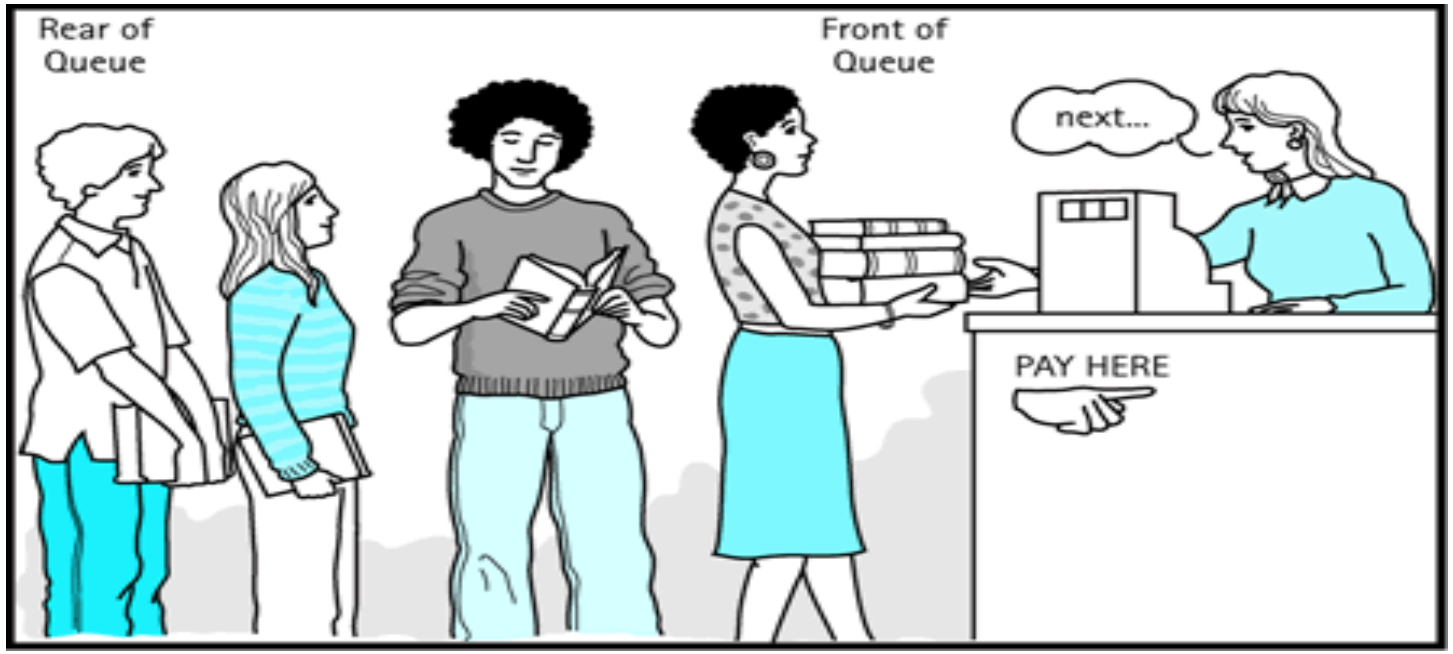
Convert Postfix expression ABC-+DE-+to infix expression

Token	Stack	Action
A	A	Push A into stack
B	A,B	Push B into Stack
C	A,B,C	Push C into Stack
-	A,B-C	Pop B and C from stack and put – symbol in between B and C
+	A+B-C	Pop A, B-C from stack and put + symbol in between A, B-C
D	A+B-C, D	Push D into stack
E	A+B-C, D, E	Push E into Stack
-	A+B-C,D-E	Pop D and E from stack and Put- symbol in between D and E
+	A+B-C+D-E	Pop A+B-C, D-E from stack and put + symbol in between A+B-C, D+E

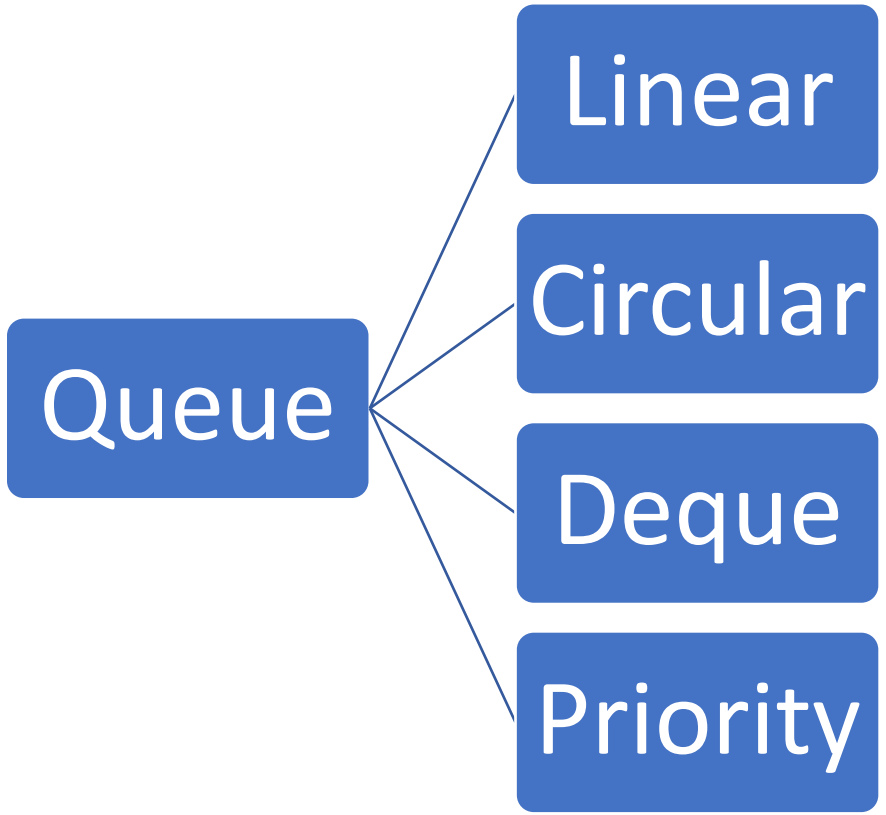
Queue Introduction

- It is also a linear data structure like Stack
- It is an ordered group of homogeneous items of elements.
- It is a First In First Out (FIFO) or Last In Last Out (LILO) data structure
- A list structure with two access points called the front and rear.
- All insertions (enqueue) occur at the rear and deletions (dequeue) occur at the front.

Queue Introduction



Queue Types



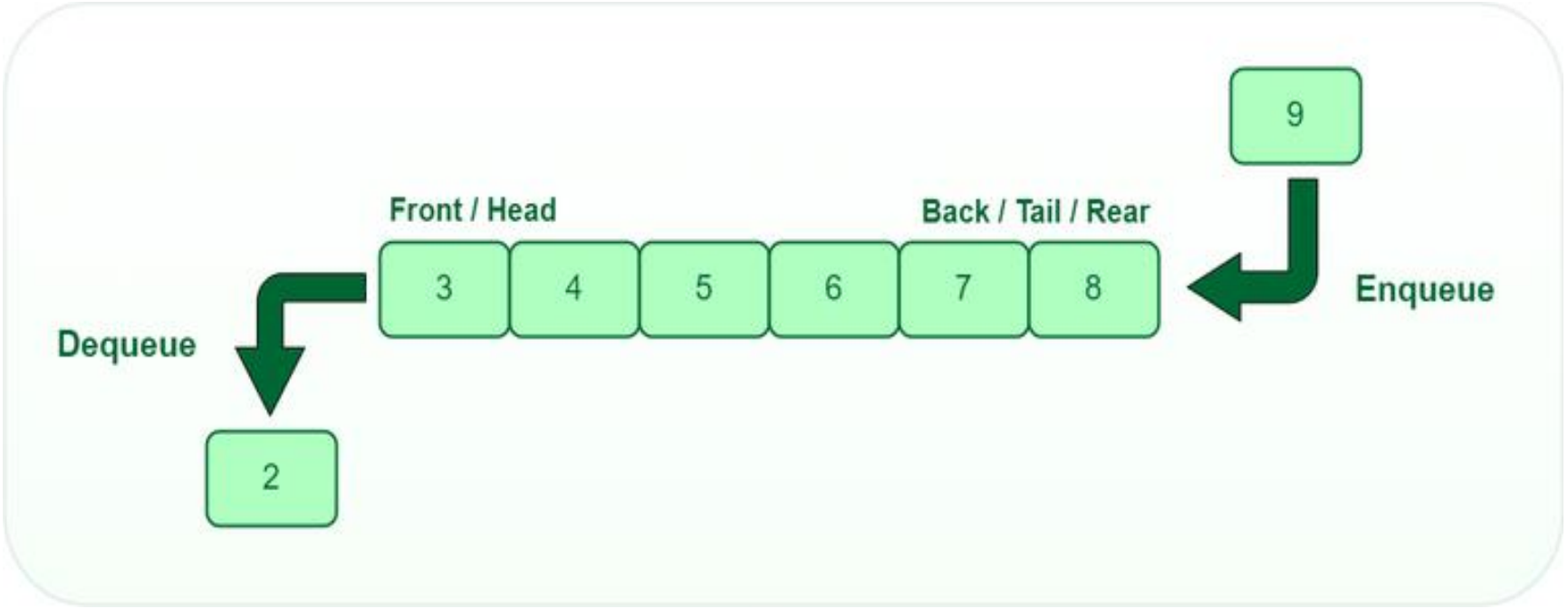
Operations on Queue

- Enqueue (insertion)
  - enqueue - adds an element to the rear of a queue
- Dequeue (deletion)
  - dequeue - removes and returns the front element of the queue
- IsEmpty (check whether queue is empty)
- IsFull (check whether queue is full)

Operations on Queue

- **peek() or front()-** Acquires the data element available at the front node of the queue without deleting it.
- **rear() –** This operation returns the element at the rear end without removing it.
- **size():** This operation returns the size of the queue i.e. the total number of elements it contains.

Queue Data Structure



Queue Operations

1. Add 1st element

rear = front = -1 

In the above case front = rear = 0

2. Queue is full when rear = length-1

In the above case front = 0, rear = 3

3. 1 element left. In the below case if front = rear = 2 and if we delete the below element then front will surpass rear then we need to reset front and rear to -1.

Queue Operations

4. Queue Empty

Rear = -1, front = -1

Queue Data Structure

```
Class queue{
int arr[];
int front, rear;
queue(int len)
{
arr = new int[len];
front = rear = -1;
}
}
```

Queue Addition(enqueue) Operation

```
void addrear(int element)
{
if( rear == arr.length-1)
{
//Queue is full
return;
}
if(front == -1){
// Queue is Empty
front = 0; rear =0;
arr[0] = element;
return;
}
rear++;
arr[rear]=element;
}
```

Queue Deletion(dequeue) Operation

```
int deletefront()
{
if(front == -1)
{
//Queue is empty
return -1;
}
if (front == rear){
int temp1 = arr[rear];
front = -1; rear = -1;
return temp1;
}
int temp = arr[front];
front++;
return temp;
}
```

Introduction to Dequeue

Deque or Double Ended Queue is a type of queue in which insertion and removal of elements can either be performed from the front or the rear. Thus, it does not follow FIFO rule (First In First Out).

Operations on Dequeue

Here in the above case front =1, rear = 2

1. AddRear(5)

Front = 1, rear = 3, rear++

2. DeleteRear()

Front =1, rear = 2, rear--

Operations on Dequeue

3. AddFront(5) then front = 0, rear =2, front--

4. DeleteFront() then front = 1, rear =2, front++

5. DeleteFront() then front = 2, rear =2, front++

6. DeleteRear() then front = 2, rear =1, rear--. Here in this case front surpasses rear when it is happened reset front and rear

Double Ended Queue Operations

```
void addRear(int element)
{
if( rear == arr.length-1)
{
//rear full
return;
}
if(front == -1)
{
front =0; rear =0;
arr[0] = element;
return;
}
rear++;
arr[rear] = element;
 }
```

```
void addFront(int element)
{
if(front == 0){
// front full
return;
}
if(front == -1)
{ front =0; rear =0;
arr[0] = element;
return;
}
front--;
arr[front]= element;
}
```

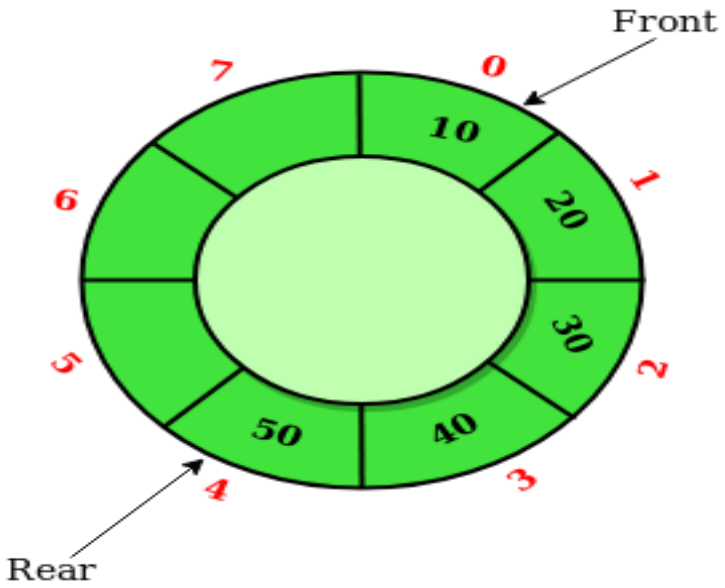
Double Ended Queue Operations

```
int deleteFront()
{
if(front == -1)
{
//Queue Empty
return -1;
}
if( front == rear)
{
// 1 element is left
int temp1 = arr[front];
front= rear =-1;
return temp1;
}
int temp = arr[front];
front++;
return temp;
}
```

```
Int deleteRear()
{
if(rear == -1){
// Queue Empty
return -1;
}
if (front == rear)
{// 1 element left.
int temp1 = arr[rear];
front = rear = -1;
 return temp1;
}
int temp =  arr[rear];
rear--;
return temp;
}
```

Circular Queues

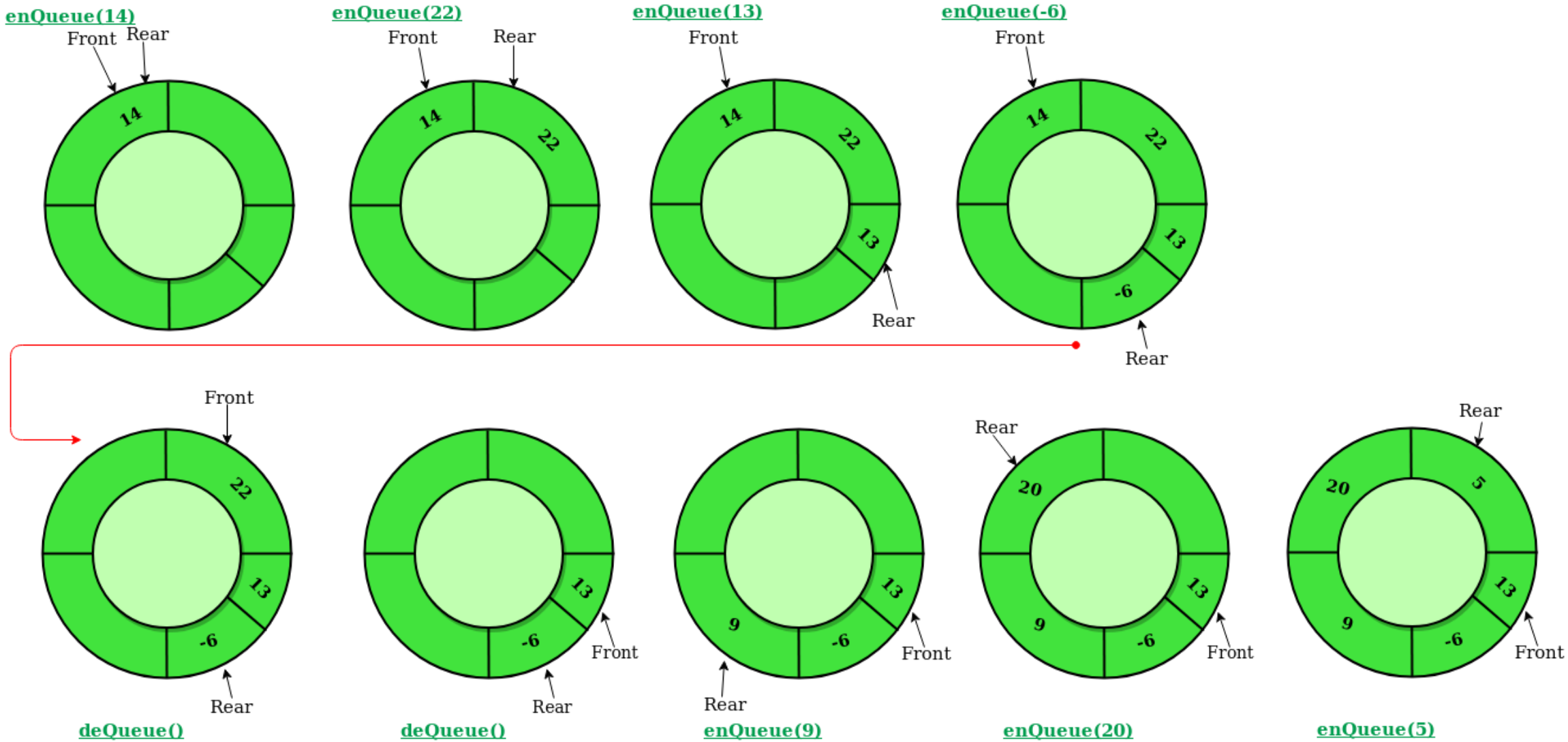
- A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.
- The operations are performed based on FIFO (First In First Out) principle. It is also called ‘Ring Buffer’.



Circular Queues

- In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

Circular Queues



Circular Queues

Consider the below Queue Where front =1, rear = 3

Add rear(5), for the above queue then front = 1, rear = 0 as per circular queue

Rear ++ = 4
Rear = Rear ++ % length = 4%4 =0

delete front(), for the below queue assume currently rear = 1, front = 3 as per circular queue

front ++ = 4
front = front ++ % length = 4%4 =0

Circular Queue Operations

```
 void addrear(int element)
{
if ((r-f == arr.length-1)||(f-r ==1))
{
// queue is full
return;
}
if(front == -1){
Front =0, rear =0;
arr[0] = element;
return;}
rear = rear++ % arr.length;
arr[rear] = element;
}
```

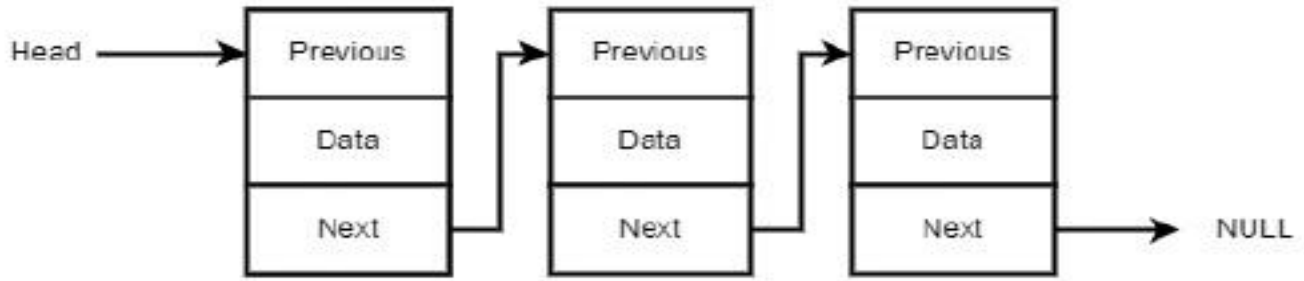
```
int deletefront()
{
if(front == -1){// queue empty
return -1;
}
if(rear == front) { // last element of queue
int temp1 = arr[rear];
front = rear =-1;
return temp1;
}
int temp = arr[front];
front = front++ % arr.length;
return temp;}
```

Intro to Linked list

- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.
- In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.
- **Singly Linked List** − The nodes only point to the address of the next node in the list.
- **Doubly Linked List** − The nodes point to the addresses of both previous and next nodes.
- **Circular Linked List** − The last node in the list will point to the first node in the list. It can either be singly linked or doubly linked.

Linked List Representation

Single Linked List



Double Linked List

Linked List Representation

- Linked List contains a link element called first (head).
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

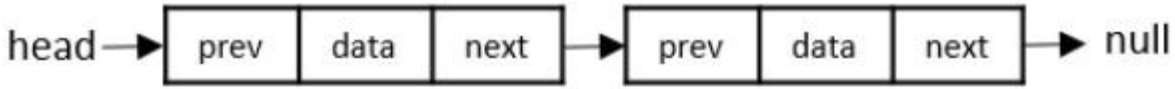
Types of Linked Lists

Singly Linked Lists

Singly linked lists contain two “buckets” in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.

Types of Linked Lists

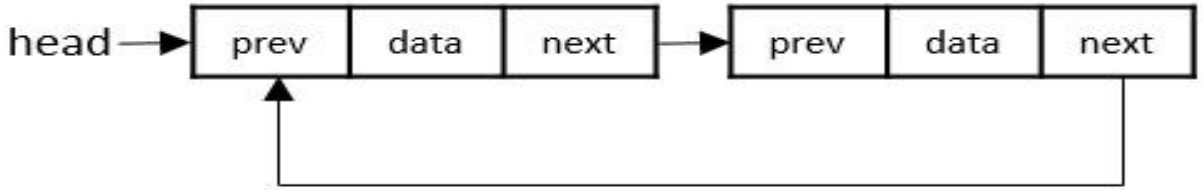
Doubly Linked Lists



Doubly Linked Lists contain three “buckets” in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.

Types of Linked Lists

Circular Linked Lists



Circular linked lists can exist in both singly linked list and doubly linked list. Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.

Basic Operations on Linked Lists

- The basic operations in the linked lists are insertion, deletion, searching, display, and deleting an element at a given key.
- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

How to create a Node in Java

Create a java class for node

```
class Node {
    int value;
    Node next;
}
```

Create an object for node class

```
Node newnode=new Node();
newnode.value=element;
newnode.next=null;
```

And initially linked list is

```
static Node head=null;
```

Insertion at Beginning- Pseudocode

```
public static void insertBegin(int element)    {
        Node newnode=new Node();
        newnode.value=element;
        if(head==null)  {
                head=newnode;
        }
        else  {
                newnode.next=head;
                head=newnode;
        }
   }
```

- Assume we have an initially empty linked list, represented by the head variable:

```bash
(head)
```

1.We call insertBegin(5) to insert the value 5 at the beginning of the list.
2.Inside the method, a new node newnode is created with the value 5:

```scss
newnode (5)
```

3.We check if the head is null, which indicates that the list is empty. In this case, the head is null, so we execute the if block.
4.Inside the if block, we set the head to point to the newnode:

```scss
(head) --> newnode (5)
```

Now, the linked list contains only one node, which is the newnode with the value 5, and it becomes the new head of the list.

If we call insertBegin(3) to insert 3 at the beginning of the list again:

A new node newnode is created with the value 3:

```scss
newnode (3)
```

We check if the head is null, but it's not null this time because the list already has a head node.

Inside the else block, we set the next reference of the newnode to point to the current head:

```scss
(newnode) --> (head) --> newnode (5)
```

Finally, we update the head to point to the newnode.

```scss
(head) --> newnode (3) --> (old head) --> newnode (5)
```

Insertion at Middle- Pseudocode

```
public static void insertMiddle(int position, int element)    {
        int current=0;
        Node newnode=new Node();
        newnode.value=element;
        Node temp=head;
        while(temp.next!=null)  {
                        current++;
                        if(current==position)
                        break;
                        temp=temp.next;
        }
        newnode.next=temp.next;
        temp.next=newnode;        }
```

- Imagine we have a linked list initially containing the elements: 1 -> 2 -> 3 -> 4
- We want to insert the value 5 at position 2. Here's how it works step by step:
- Initial Linked List:

```rust
1 -> 2 -> 3 -> 4
```

- We call insertMiddle(2, 5) to insert 5 at position 2.
- current is initialized to 0, and a new node newnode is created with the value 5.
- temp is initialized to head, which is the first node (1).
- Now, we enter a loop to find the insertion position:
- current is 0, and temp is pointing to 1.
- We increment current to 1 (since we've checked the first node).
- current is not equal to 2 (the desired position), so we continue.
- We move temp to the next node, so it now points to 2.
- At this point, temp is at the node just before the desired insertion position.

Linked List State:

```rust
1 -> [temp] 2 -> 3 -> 4
```

- Now, we break out of the loop since current is equal to 2 (the desired position).
- We proceed to insert newnode:
- newnode.next is set to temp.next, which is the node currently at position 2. So, newnode.next points to 2.
- temp.next is updated to point to newnode. This effectively inserts 5 at the desired position.

Linked List After Insertion:

```rust
1 -> 2 -> [newnode] 5 -> [temp] 3 -> 4
```

The code successfully inserted the value 5 at the specified position (2), and the existing nodes in the list were adjusted accordingly.

Insertion at end- Psuedocode

```
public static void insertEnd(int element)   {
        Node newnode=new Node();
        newnode.value=element;
        newnode.next=null;
        Node temp=head;
        while(temp.next!=null)
                temp=temp.next;
        temp.next=newnode;
    }
```

Assume we have a linked list with the following elements:

```rust
1 -> 2 -> 3
```

- We call insertEnd(4) to insert the value 4 at the end of the list.
- Inside the method, a new node newnode is created with the value 4:

```scss
newnode (4)
```

The next reference of newnode is set to null because it will be the last node in the list:

```csharp
newnode (4) -> null
```

- A temporary node temp is created and initialized to the head of the list:

```rust
(temp) -> 1 -> 2 -> 3
```

- We enter a loop that iterates through the linked list until we find the last node.
- In this case, we move temp through the list until temp.next becomes null:

```rust
(temp) -> 1 -> 2 -> 3 -> (null)
```

- Once we reach the last node (where temp.next is null), we exit the loop.
- Now, we set the next reference of the last node (previously temp) to point to the newnode:

```rust
(temp) -> 1 -> 2 -> 3 -> (newnode) -> 4 -> (null)
```

This action effectively inserts the new node newnode with the value 4 at the end of the linked list, and the list is updated accordingly.
The final linked list is: 1 -> 2 -> 3 -> 4

Deletion at Beginning- Pseudocode

```
public static void deleteBegin()    {
            head=head.next;
    }
```

Assume we have a linked list with the following elements:

1.We call the deleteBegin method to remove the first element (the head) from the list.
2.Inside the method, the head pointer is updated to point to the second element in the list (i.e., head.next):

rust

```
(head) -> 1 -> 2 -> 3 -> 4
```

rust

```
(head) -> 2 -> 3 -> 4
```

Delete at middle

```
public static void deleteMiddle(int position)    {
        int current=0;
        Node temp=head;
                while(temp.next!=null) {
                current++;
                if(current == position)
                        break;
                temp=temp.next;
        }
        temp.next = temp.next.next;
}
```

Assume we have a linked list with the following elements:

```rust
(head) -> 1 -> 2 -> 3 -> 4
```

- We call the deleteMiddle(2) method to remove the element at position 2. Here, position is the position of the element we want to delete.
- Inside the method, a loop is used to traverse the linked list until we reach the desired position (position). A current variable keeps track of the current position as we move through the list.
- Initially, current is 0, and temp is initialized to head:

```rust
(temp) -> 1 -> 2 -> 3 -> 4
```

We increment current to 1 and move temp to the next node (2):

```rust
1 -> (temp) -> 2 -> 3 -> 4
```

- current is now 1, and we continue until current reaches the desired position (2).
- Once we reach the desired position (2 in this case), we break out of the loop.
- To delete the node at the specified position, we update the next reference of the previous node (the node just before the one to be deleted) to skip over the node we want to delete. In this case, we set temp.next to temp.next.next:

```rust
1 -> (temp) -> 2 -> 3 -> 4
```

```rust
1 -> (temp) -> 3 -> 4
```

- Now, the node with the value 2 has been effectively removed from the linked list.
- The final linked list, after calling deleteMiddle(2), is: 1 -> 3 -> 4

Delete at End

```
public static void deleteEnd()   {
        Node temp=head;
        Node prev=head;
        while(temp.next!=null) {
                prev=temp;
                temp=temp.next;
        }
        prev.next=null;
 }
```

Assume we have a linked list with the following elements:

```rust
(head) -> 1 -> 2 -> 3 -> 4
```

1.We call the deleteEnd method to remove the last element from the list.
2.Inside the method, two pointers, temp and prev, are used to traverse the linked list. Both temp and prev are initially set to head:

```rust
(prev) -> (temp) -> 1 -> 2 -> 3 -> 4
```

A loop is used to traverse the linked list until we reach the last element. The loop continues until temp.next becomes null, which indicates the last element:
•In the first iteration, prev and temp both point to the first element (1).
•In subsequent iterations, prev is one step behind temp.

3.When the loop exits, temp is pointing to the last element (4), and prev is pointing to the element just before it (3).
4.To delete the last element, we update the next reference of the previous node (pointed to by prev) to null, effectively breaking the link to the last element:

```rust
(prev) -> 1 -> 2 -> 3 -> (temp) -> 4
```

```rust
(prev) -> 1 -> 2 -> 3 -> (temp) -> null
```

Now, the last node with the value 4 has been effectively removed from the linked list.
The final linked list, after calling deleteEnd, is: 1 -> 2 -> 3
So, the deleteEnd method successfully removes the last element from the linked list by updating the next reference of the previous node to null.

Display

```
public static void display()   {
        Node temp=null;
        temp=head;
        while(temp!=null)  {
                System.out.print(temp.value+" ");
                temp=temp.next;
        }
 }
```

Doubly Linked List- Insert at begin

```
public static void insertBegin(int element)    {
            DLLNode newnode=new DLLNode();
            newnode.value= element;
if(head==null)  {
       head=tail=newnode;
}
else   {
            newnode.next=head;
            head.prev=newnode;
            head=newnode;
}
   }
```

1.We create a new DLLNode called newnode with the value 7.
2.We check if the head is null, which means the list is empty. In this case, head and tail are both set to the new node (7): head = tail = newnode;
3.If the list is not empty (i.e., head is not null), we perform the following operations:

1. We update the next reference of the new node (7) to point to the current head (which is the node 3): newnode.next = head;
2. We update the prev reference of the current head (node 3) to point back to the new node (7): head.prev = newnode;
3. We update the head reference to point to the new node (7): head = newnode;

Before the insertion:

rust Copy code

```
3 <-> 5
```

After the insertion:

rust Copy code

```
7 <-> 3 <-> 5
```

Doubly Linked List- Insert at Middle

```
public static void insertMiddle(int position, int element)    {
        int current=0;
        DLLNode newnode=new DLLNode();
        newnode.value=element;
        DLLNode temp=head;
        while(temp.next!=null)  {
                current++;
                if(current==position)
                break;
                temp=temp.next;
        }
        newnode.next=temp.next;
        temp.next.prev=newnode;
        temp.next=newnode;
        newnode.prev=temp;
}
```

Example of inserting the value 7 at position 2 in the following doubly linked list:

```rust
1 <-> 3 <-> 5
```

1.current is initialized to 0, and temp is a reference to the head of the list.
2.The code enters a loop that iterates through the list as long as temp.next is not null. This loop allows us to traverse the list until we reach the desired position.
3.Inside the loop:

1. current is incremented by 1 to keep track of our position in the list.
2. We check if current is equal to the target position (in this case, 2).
3. If current becomes equal to the position, we break out of the loop.

4.After breaking out of the loop, temp is now pointing to the node just before the position where we want to insert the new node (1 in this case).
5.We create a new DLLNode called newnode with the value 7.
6.We update the next reference of newnode to point to the node that comes after the position where we want to insert it (3 in this case): newnode.next = temp.next;
7.We update the prev reference of the node that comes after the position (3) to point back to the new node (7): temp.next.prev = newnode;
8.We update the next reference of temp to point to the new node (7): temp.next = newnode;
9.Finally, we update the prev reference of the new node (7) to point back to temp: newnode.prev = temp;

7.We update the prev reference of the node that comes after the position (3) to point back to the new node (7): temp.next.prev = newnode;
8.We update the next reference of temp to point to the new node (7): temp.next = newnode;
9. Finally, we update the prev reference of the new node (7) to point back to temp: newnode.prev = temp;

Before the insertion:

rust Copy code

```
1 <-> (temp) <-> 3 <-> 5
```

After the insertion:

rust Copy code

```
1 <-> (temp) <-> 7 <-> 3 <-> 5
```

Doubly Linked List- Insert at End

```
public static void insertEnd(int element)   {
            DLLNode newnode=new DLLNode();
            newnode.value=element;
            newnode.next=null;
            tail.next=newnode;
            newnode.prev=tail;
            tail=newnode;
   }
```

```rust
3 <-> 5
```

1.We create a new DLLNode called newnode with the value 7.
2.We set the next reference of newnode to null. This prepares newnode to be the new tail of the list.
3.We update the next reference of the current tail (5) to point to the new node (7): tail.next = newnode;
4.We update the prev reference of the new node (7) to point back to the current tail (5): newnode.prev = tail;
5.Finally, we update the tail reference to point to the new node (7): tail = newnode;

Before the insertion:

```rust
3 <-> 5
```

After the insertion:

```rust
3 <-> 5 <-> 7
```

Doubly Linked List- Delete at Begin

```
public static void deleteBegin()    {
        head=head.next;
        head.prev=null;
    }
```

Before the deletion:

```rust
3 <-> 5 <-> 7
```

1.We update the head reference to point to the node that comes after the current head (in this case, 5): head = head.next;
2.We update the prev reference of the new head (node 5) to null because it is now the first node: head.prev = null;

Before the deletion:

```rust
3 <-> 5 <-> 7
```

After the deletion:

```rust
5 <-> 7
```

Doubly Linked List- Delete at middle

```
public static void deleteMiddle(int position)    {
        int current=0;
        DLLNode temp=head;
        while(temp.next!=null) {
                current++;
                if(current ==position)
                        break;
                temp=temp.next;
        }
        temp.next.next.prev=temp;
        temp.next=temp.next.next;
}
```

Before the deletion:

```rust
3 <-> 5 <-> 7 <-> 9
```

1.We initialize a variable current to 0 to keep track of the current position in the list.
2.We start with the temp node pointing to the head of the list (3).
3.We use a while loop to traverse the list until we reach the node at the specified position (position).
    1. In this example, we want to delete the node at position 2, so we will traverse to the node with the value 7.
4.Once we reach the node before the one to be deleted (which is 5 in this case), we perform the deletion:
    1. We update the prev reference of the node after the one to be deleted (7) to point back to the node before the one to be deleted (5): temp.next.next.prev = temp;
    2. We update the next reference of the node before the one to be deleted (5) to point to the node after the one to be deleted (9): temp.next = temp.next.next;

After the deletion (node `7` at position `2` is deleted):

```rust
3 <-> 5 <-> 9
```

Doubly Linked List- Delete at End

```
public static void deleteEnd()   {
        tail=tail.prev;
                tail.next=null;
 }
```

Before the deletion:

```rust
3 <-> 5 <-> 7
```

1.We update the tail reference to point to the node before the current tail. In other words, we make tail point to its prev node: tail = tail.prev;
2.We set the next reference of the new tail (which was the previous tail) to null because it is now the last node: tail.next = null;

After the deletion:

```rust
3 <-> 5
```

Doubly Linked List- Display

```
public static void display()   {
        Node temp=null;
        temp=head;
        while(temp!=null)  {
                System.out.print(temp.value+" ");
                temp=temp.next;
        }
 }
```

Doubly Linked List- reverse

```
public static void reverse()   {
        DLLNode temp=tail;
        while(temp!=null)      {
                System.out.print(temp.value+" ");
        temp=temp.prev;
 }
```

Before the reversal:

```rust
3 <-> 5 <-> 7 <-> 9
```

1.We initialize a temporary node temp and set it to the current tail of the doubly linked list. This means temp initially points to the last node (9).
2.We use a while loop to traverse the list from the tail (last node) to the head (first node).
    1. In each iteration of the loop, we print the value of the current node pointed to by temp.
    2. We then move temp to the previous node using temp = temp.prev;.
3.The loop continues until temp becomes null, indicating that we have traversed the entire list in reverse order.

After the reversal (printing the values in reverse order):

```
9 7 5 3
```

Infix to Prefix conversion

Convert the infix expression A*B+C/D to Prefix

- Given Expression is A*B+C/D

  (A*B)+(C/D)

  (*AB)+(/CD) Assume x1= (*AB), x2= (/CD)

  x1+ x2

  (+x1x2)

  +*AB/CD

Infix to Prefix conversion

Convert the infix expression A*B-C/D+E to Prefix

- Given Expression is A*B-C/D+E

  (A*B)-(C/D)+E

  (*AB)-(/CD) Assume x1= (*AB), x2= (/CD)

  (x1- x2) +E

  -x1x2+E Assume x3 = -x1x2

  x3+E

  +x3E

  +-x1x2E

  +-*AB/CDE

Infix to Prefix conversion

Convert the infix expression (A+B)/C*D-E to Prefix

- Given Expression is **(A+B)/C*D-E**

  (+AB)/C*D-E Assume x1= +AB

  x1/C*D-E

  (x1/C)*D-E

  (/x1C)*D-E Assume x2 = /x1C

  (x2*D)-E Assume x3 = *x2D

  x3-E

  -x3E

  -*x2DE = -*/x1CDE= -*/+ABCDE

Infix to Prefix conversion

Convert the infix expression A/B*C-D+E/F/(G+H) to Prefix

- Given Expression is **A/B*C-D+E/F/(G+H)**

  (A/B)*C-D+E/F/(G+H)

  (/AB)*C-D+E/F/(+GH) Assume x1= /AB, x2= +GH

  x1*C -D+E/F/x2

  *x1C – D+E/F/x2 Assume x3= *x1C

  x3-D+(//EFx2) Assume x4 = //EFx2

  x3-D+x4

  -x3D+x4 Assume x5 = -x3D

  x5+x4

  +x5x4

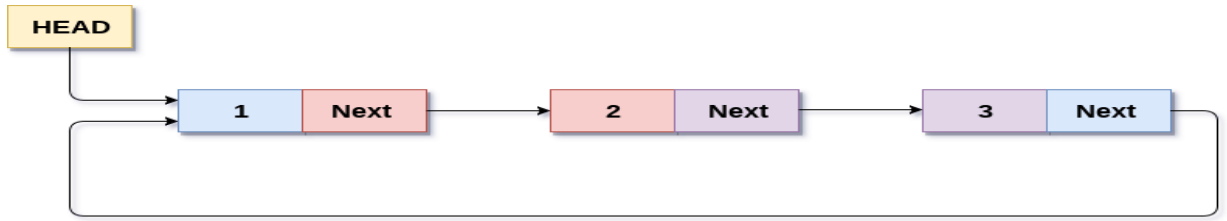
  +-x3D//EFx2

  +-*x1CD//EF+GH

  +-*/ABCD//EF+GH

Circular singly Linked List

- In a circular Singly linked list, the last node of the list contains a reference to the first node of the list.
- We traverse a circular singly linked list until we reach the same node where we started.
- The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes. The following image shows a circular singly linked list.



Circular Singly Linked List

How to create a Node in Java

Create a java class for node

```
class Node {
    int value;
    Node next;
}
```

Create an object for node class

```
Node newnode=new Node();
newnode.value=element;
```

Insert at begin

```
public static void insertBegin(int element) {
        CSLNode newnode = new CSLNode();
        newnode.value = element;
        if (head == null) {
                head = newnode;
                newnode.next = head;
        }
        else {
                CSLNode temp = head;
                while (temp.next != head)
                        temp = temp.next;
                newnode.next = head;
                head = newnode;
                temp.next = head;
                }
}
```

Insert at middle

```
public static void insertMiddle(int position, int element) {
        int current = 0;
        CSLNode newnode = new CSLNode();
        newnode.value = element;
        CSLNode temp = head;
        while (temp.next != head) {
                current++;
                if (current == position)
                break;
                temp = temp.next;
        }
        newnode.next = temp.next;
        temp.next = newnode;  }
```

Before the insertion:

1.We create a new node newnode with the value 12.
2.We initialize a variable current to keep track of the current position while traversing the CSLL. It's initially set to 0.
3.We initialize a temporary node temp and set it to the head of the CSLL to start the traversal.
4.We use a while loop to traverse the CSLL. The loop continues until temp.next is equal to head, which means we have completed one full cycle of the CSLL.

1. In each iteration, we increment the current position.
2. We check if the current position is equal to the desired position where we want to insert the new node (in this case, position is 2).
3. If current matches the desired position, we break out of the loop.
4. If not, we update temp to point to the next node in the CSLL, continuing the traversal.

5.Once we break out of the loop, we have reached the node at the desired position - 1 (i.e., one node before the desired position). This is where we want to insert the new node.
6.We update the newnode.next to point to the node that was originally at the desired position, which is temp.next. This effectively inserts newnode after temp.
7.Finally, we update temp.next to point to newnode, completing the insertion.

After the insertion of `12` at position 2:

```markdown
 3 <-> 5 <-> 12 <-> 7
 ^                  |
 |__________________|
```

Insert at end

```
public static void insertEnd(int element) {
        CSLNode newnode = new CSLNode();
        newnode.value = element;
        CSLNode temp = head;
        while (temp.next != head)
                temp = temp.next;
        temp.next = newnode;
        newnode.next = head;
}
```

1.We create a new node newnode with the value 12.
2.We initialize a temporary node temp and set it to the head of the CSLL to start the traversal.
3.We use a while loop to traverse the CSLL. The loop continues until temp.next is equal to head, which means we have completed one full cycle of the CSLL.

1. In each iteration, we check if temp.next is not equal to head. If it's not, it means we haven't reached the end of the CSLL yet, so we update temp to point to the next node and continue the traversal.
2. When we reach the end of the CSLL (i.e., when temp.next is head), we exit the loop.

4.Once we exit the loop, temp points to the last node in the CSLL.
5.We update temp.next to point to the newnode. This effectively inserts newnode after the last node.
6.Finally, we update newnode.next to point back to the head, completing the circular structure of the CSLL.

Before the insertion:

```markdown
3 <-> 5 <-> 7
^           |
|___________|
```

After the insertion of `12` at the end:

```markdown
3 <-> 5 <-> 7 <-> 12
^                 |
|_________________|
```

Delete at begin

```
public static void deleteBegin() {
        CSLNode temp = head;
        while (temp.next != head)
                temp = temp.next;
        head = head.next;
        temp.next = head;
}
```

Before the deletion:

markdown

```
3 <-> 5 <-> 7 <-> 12
^                  |
|__________________|
```

1.We initialize a temporary node temp and set it to the head of the CSLL to start the traversal.
2.We use a while loop to traverse the CSLL. The loop continues until temp.next is equal to head, which means we have completed one full cycle of the CSLL.
    1. In each iteration, we check if temp.next is not equal to head. If it's not, it means we haven't reached the end of the CSLL yet, so we update temp to point to the next node and continue the traversal.
    2. When we reach the end of the CSLL (i.e., when temp.next is head), we exit the loop.
3.Once we exit the loop, temp points to the last node in the CSLL.
4.We update the head of the CSLL to point to the next node, effectively removing the first node.
5.We update temp.next to point back to the new head, which completes the circular structure of the CSLL with the first node removed.

After the deletion of the first node:

markdown

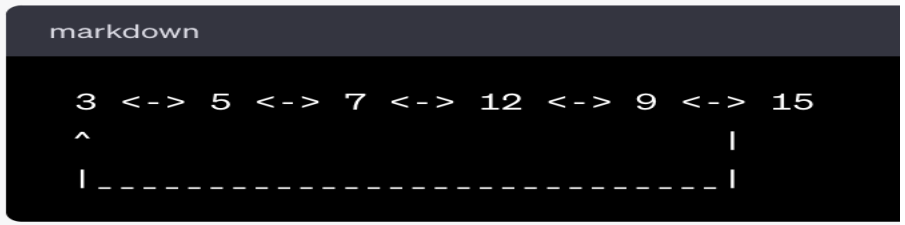
```
5 <-> 7 <-> 12
^             |
|_____________|
```

Delete at middle

```
public static void deleteMiddle(int position) {
        int current = 0;
        CSLNode temp = head;
        while (temp.next != head) {
                current++;
                if (current == position)
                        break;
                temp = temp.next;
        }
        temp.next = temp.next.next;
}
```

Before the deletion:

markdown



1.We initialize a variable current to keep track of the current position, and a temporary node temp, which we set to the head of the CSLL to start the traversal.
2.We use a while loop to traverse the CSLL. The loop continues until temp.next is equal to head, indicating that we have completed one full cycle of the CSLL.

1. In each iteration, we check if temp.next is not equal to head. If it's not, it means we haven't reached the end of the CSLL yet. We increment the current position and update temp to point to the next node, continuing the traversal.
2. When we reach the desired position (the position provided as an argument), we exit the loop.

3.Once we exit the loop, temp points to the node just before the one we want to delete.
4.We update temp.next to point directly to the node after the one we want to delete, effectively removing the desired node from the CSLL.

Before the deletion:

markdown Copy code

```
3 <-> 5 <-> 7 <-> 12 <-> 9 <-> 15
^                              |
|______________________________|
```

After the deletion of the node at `position` (assuming `position` is 2):

markdown Copy code

```
3 <-> 5 <-> 12 <-> 9 <-> 15
^                        |
|________________________|
```

Delete at end

```
public static void deleteEnd() {
    CSLNode temp = head;
    CSLNode prev = head;
    while (temp.next != head) {
        prev = temp;
        temp = temp.next;
    }
    prev.next = head;
}
```

Before the deletion:

```markdown
3 <-> 5 <-> 7 <-> 12 <-> 9 <-> 15
^                              |
|______________________________|
```

1.We initialize two pointers, temp and prev, to keep track of the current node and the node just before it. Both pointers are initially set to head.
2.We use a while loop to traverse the CSLL. The loop continues until temp.next is equal to head, indicating that we have completed one full cycle of the CSLL.

1. In each iteration, we check if temp.next is not equal to head. If it's not, it means we haven't reached the end of the CSLL yet. We update prev to point to the current temp, and then update temp to point to the next node, continuing the traversal.
2. When we reach the end of the CSLL (i.e., temp.next becomes head), we exit the loop. At this point, temp will be pointing to the last node in the CSLL.

3.Once we exit the loop, temp points to the last node, and prev points to the second-to-last node.
4.We update prev.next to point back to head, effectively removing the last node from the CSLL. This action "severs" the circular link.

Before the deletion:

markdown

```
3 <-> 5 <-> 7 <-> 12 <-> 9 <-> 15
^                               |
|_______________________________|
```

After the deletion of the last node:

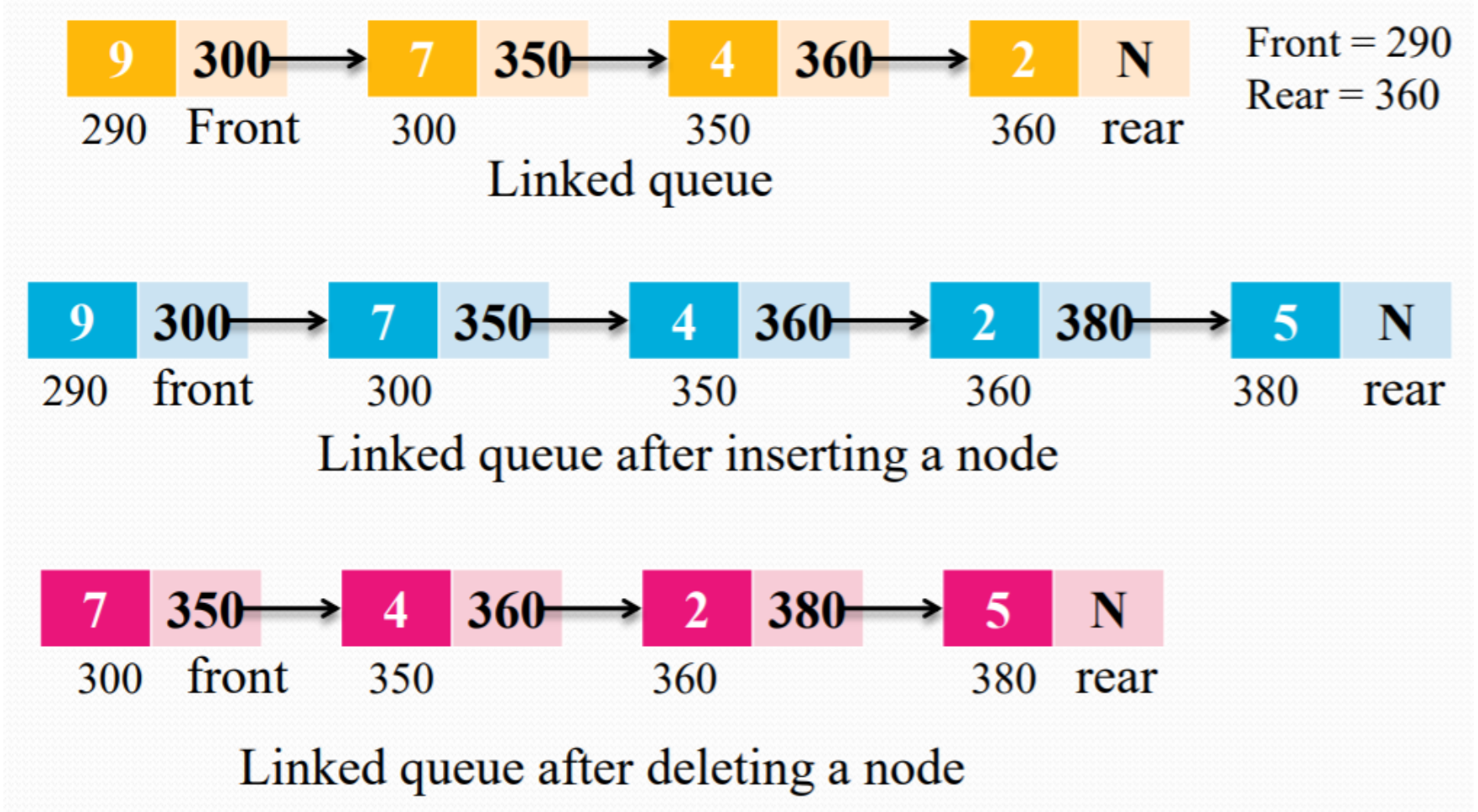
markdown

```
3 <-> 5 <-> 7 <-> 12 <-> 9
^                   |
|___________________|
```

Display

```
public static void display() {
    CSLNode temp = null;
    temp = head;
    while (temp.next != head) {
        System.out.print(temp.value + " ");
        temp = temp.next;
    }
    System.out.println(temp.value + "\n");
}
```

Linked Representation of Queues



enqeue() – List Implementation

```
public static void enqueue(int element)
 {
      Node newnode=new Node();
      newnode.value=element;
      newnode.next=null;
      Node temp=head;
      while(temp.next!=null)
                temp=temp.next;
      temp.next=newnode;
 }
```

Before the enqueue operation:

```css
Front [5] <- [8] <- [3] <- [12] Rear
```

1.We initialize a new node called newnode to represent the element we want to enqueue. The value of this new node is set to the element we want to enqueue.
2.We set the next of newnode to null because it will be the new last node in the queue.
3.We initialize a temporary node temp and set it to head, which initially points to the front of the queue.
4.We use a while loop to traverse the linked list until we find the last node in the queue. We check if temp.next is not equal to null. If it's not null, we update temp to point to the next node in the list. This loop continues until temp reaches the current rear node.
5.Once we exit the loop, temp is pointing to the current rear node (the last element in the queue).
6.We update the next of the current rear node (pointed to by temp) to point to the new node newnode. This effectively adds the new element to the back of the queue.

After the enqueue operation:

```css
Front [5] <- [8] <- [3] <- [12] <- [9] Rear
```

deqeue() – List Implementation

```
public static void dequeue()
{
    if(head==null) {
        System.out.println(" queue is empty");
    }
    else if(head.next==null)    {
         System.out.println(" The removed elements is "+head.value);
         head=null;
    }
    else    {
        head=head.next;
        System.out.println(" The removed elements is "+head.value);
    }
}
```

Before the dequeue operation:

```
css
Front [5] <- [8] <- [3] <- [12] Rear
```

1.The dequeue function is responsible for removing the front element from the queue.
2.It first checks if the head (front) of the queue is null. If it is null, it means the queue is empty, and it prints a message indicating that the queue is empty.
3.If the head is not null, it checks whether head.next is null. If head.next is null, it means there is only one element in the queue. In this case, it prints the value of the element that is being dequeued and sets head to null, effectively emptying the queue.
4.If neither of the above conditions is met, it means there are at least two elements in the queue. It updates head to point to the next element in the queue and then prints the value of the element that is being dequeued.

After the dequeue operation (if there was more than one element):

```
css
Front [8] <- [3] <- [12] Rear
```