Queue

Definition

(a) A Queue is an ordered List.

(b) In Queue, Insertions are made at rear end, Deletions are made at front end.

(c) Queue is FIFO manner.

(d) FIFO Means First-In-First out.

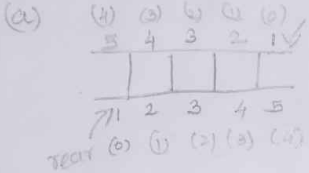
Operations:

We Can Perform Two Operations in Queue

(i) Insertion (ii) Deletion.

Example:

(a)

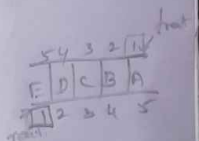


front = 0
rear = 0
n = size
n = 5.

Procedure ADDQ (Q, n, front, rear, item)

rear ← rear+1 mod n.

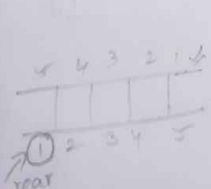
if front = rear then call QUEUEFULL endif



Q(rear) ← item

end ADDQ.

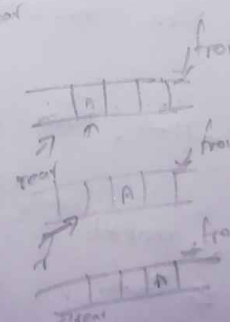
(b)

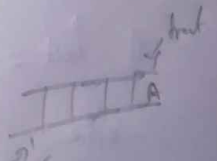


rear ← rear+1

if front = rear

(c)





Procedure DELETEQ(Q, n, front, rear, item)

if front = rear then Call QUEUE EMPTY endif

front ← front + 1 mod n

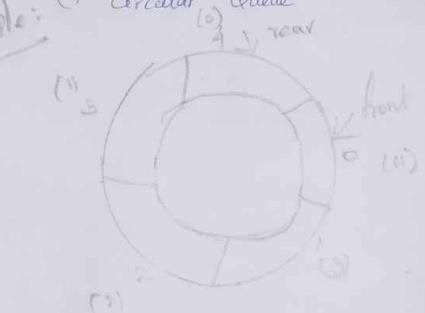
item ← Q(front)

end DELETEQ

Circular Queue:

- Circular Queue is an Ordered List.
- Last Position to be Connected back to the front Position to make a Circular. "This is Called "Ring Buffer".
- Circular Queue is also FIFO Manner.

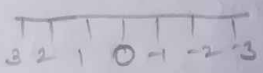
Example:



Circular Queue:

We Can Perform Two Operations

(I) Insertion (II) Deletion



Procedure ADDQ(Q, n, front, rear, item).

if front = 0 and rear = n-1 (or) front = rear + 1, then Call CIRCULAR QUEUE IS FULL endif

if front = -1, then front = 0 and rear = 0.

else

if rear = n-1, then, rear = 0.

else

rear ← rear + 1

Q(rear) ← item

Delete:

Procedure DELETEQ (Q, n, front, rear, item)

if front = -1 then call CIRCULAR QUEUE IS EMPTY end if.

else

item ← Q(front)

if front = rear then front and rear will be -1.

else

if front = n-1, then front = 0

else

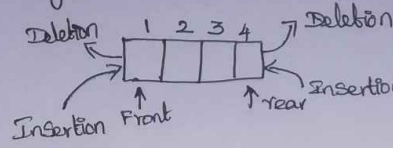
front ← front+1

end DELETEQ.

Double - Ended Queue

* Double Ended Queue means Insertions and Deletions are Performed at both ends, which means you can Insert an element at front end and rear end, Similarly you can delete an element at front end and rear end.

Eg:



n = size front = 0
n = 4 rear = 0

Algorithm for Insertion at rear end:

Procedure INSERTREAR (Q, n, Front_rear, element)
  if rear = n then Call "Queue is Full" end if
  else
    rear ← rear + 1
    Q(rear) ← element
  if rear = 0 then rear = 1
  if Front = 0 then Front = 1
  return
end INSERTREAR

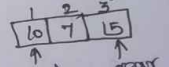
Eg:

Elements are: 10, 7, 15, 20.

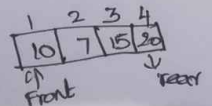
① Front = 1, rear = 1

② rear = 1, front = 1
rear ← rear + 1
rear ← 1 + 1
(2)

③ rear = 2, Front = 1
rear ← rear + 1
rear ← 2 + 1
(3)



④ rear = 3, front = 1
rear ← rear + 1
rear ← 3 + 1
← 4



⑤ rear = 4, front = 1.
if rear = n then
4 = 4
end if

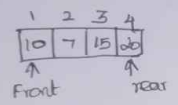
Queue Full Condition is true

So Procedure End.

Algorithm for Deletion at rear end

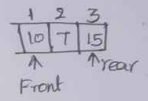
Procedure DELETEREAR (Q, n, front, rear, element)
  if rear = 0 then Call "Queue is Empty" end if
  else
  element ← Q(rear)
  if front = rear then
    front = 0;
    rear = 0;
  else
  rear ← rear - 1
  return
end DELETEREAR.

Eg: ①



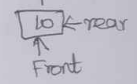
Front = 1
rear = 4

② element ← Q(rear)
20 ← Q(4)
if front = rear
1 = 4
else
rear ← rear - 1
rear ← 4 - 1
rear ← 3



③ Front = 1, rear = 3
element ← Q(rear)
15 ← Q(3)
if front = rear
1 = 3
else
rear ← rear - 1
rear ← 3 - 1
rear ← 2

④ Front = 1, rear = 2
element ← Q(rear)
7 ← Q(2)
if front = rear
1 = 2
else
rear ← rear - 1
rear ← 2 - 1
rear ← 1



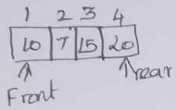
⑤ Front = 1 rear = 1,
element ← Q(rear)
element ← Q(1)
if front = rear then
1 = 1
Front = 0;
rear = 0;

⑥ Front = 0, rear = 0
if rear = 0 then Call Queue Empty end if
0 = 0.
Condition is True
So Procedure End

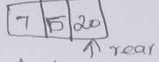
Algorithm for Deletion at front end:

```
Procedure DELETEFRONT (Q, n, front, rear, element(s)
    If front = 0 then Call "Queue is Empty" end If
    else
    element ← Q(front)
    if front = rear then
        front = 0
        rear = 0
    else
    front ← front + 1
    return
end DELETEFRONT
```

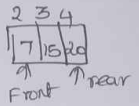
Eg: ①



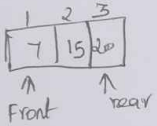
front = 1 element Q(front)
rear = 4 10 ← Q(1)



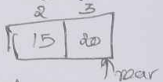
front ← front + 1
front ← 1 + 1
(2)



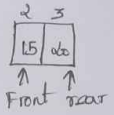
②



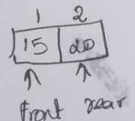
front = 1 element Q(front)
rear = 3 7 ← Q(1)



front ← front + 1
front ← 1 + 1
(2)

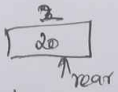


③

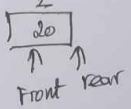


Front = 1
rear = 2

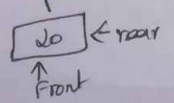
element ← Q(front)
15 ← Q(1)



front ← front + 1
front ← 1 + 1
front ← (2)



④



front = 1 element Q(front)
rear = 1 20 ← Q(1)
if front = rear then
1 = 1

front = 0
rear = 0

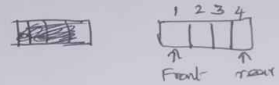
⑤ if front = 0
then
Queue Empty
0 = 0
Condition is True end

Algorithm for Insertion at front end:

Procedure INSERTFRONT (Q, n, front, rear, element)
if front = 1 then call "Queue is FULL" and if
else

front ← front − 1
Q(front) ← element
return
end INSERTFRONT

Eg①:

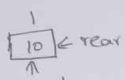


front = 0
rear = 0

So, before an element is Inserted into Q we cannot find front

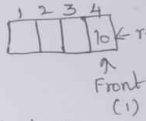
So first element to be Inserted at rear end.

②



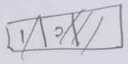
front = 1
rear = 1

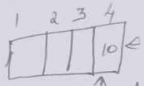
Consider if rear = 1 means that is the 4th position from front.



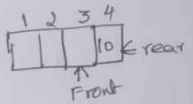
So, Front = 4.

③

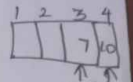




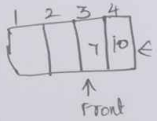
Front = 4.
front ← front − 1
front ← 4 − 1
front ← 3



Q(front) ← element
Q(3) ← 7

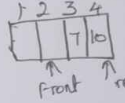


④

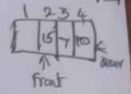


front = 3
rear = 4

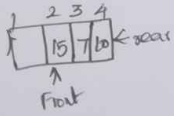
front ← front − 1
front ← 3 − 1
front ← 2



Q(front) ← element
Q(2) ← 15

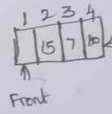


⑤

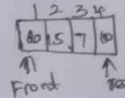


Front = 2
rear = 4

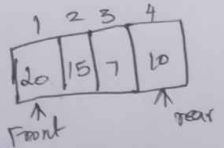
front ← front − 1
front ← 2 − 1
front ← 1



Q(front) ← element
Q(1) ← 20



⑥



Front = 1
rear = 4

If front = 1 Then Call "Queue FULL"
1 = 1 Condition True
So Procedure End.

Linked List

(x) A Linked List is a Sequence of data structures, which are connected together via Links.

(x) Linked List is a Sequence of links which Contains items. Each link Contains a Connection link to another Link.

Important Terms to understand Linked List:

Link ⇒ Each link of a Linked list Can store a data called an "Element".

Next ⇒ Each link of a Linked List Contains a link to the next link called "Next".

Linked List ⇒ A Linked List Contains a Connection link to the first link called "First / Head".

Linked List Representation:

Linked List Can be visualized as a chain of nodes, where every node Points to the next node.

(x) Linked List Contains a link element called First / Head.

(x) Each link carries a data fields and a link field called "next".

(x) Each link is linked with its next link by using its "next".

(x) Last link carries a link as NULL to Indicate the end of the List.

Types of Linked List:

(i) Simple / single Linked List:
Elements Can be Navigated only forward approach.

(ii) Doubly Linked List:
(x) Elements Can be Navigated Forward and backward approach.

(iii) Circular Linked List:
(x) Last element Contains a link of the First element as next, and the first element has a link to the last element as Previous.

Operations:

(i) Insertion (ii) Deletion (iii) Display (iv) Search (v) Delete

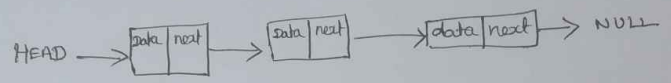
(i) Single [simple Linked List:

(a) Each link of a single Linked List can store a data element at data field, and Contains a link to another link in the next field.

(x) Contains a Connection link (first /head) to the first link of a single linked list.

Representation of Single linked list:

Forward approach →



Backward approach ←

Node is represented as:

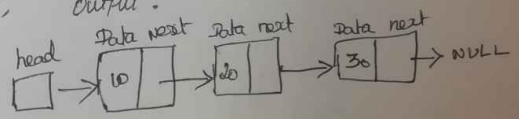
```
struct node {
    int data;
    struct node * next;
};
```

A three member Single linked list can be Created as.

```
/* Initialize nodes */
struct node * head;
Struct node * one = NULL;
struct node * two = NULL;
struct node * three = NULL;

/* Allocate Memory */
one = malloc (size of (struct node));
two = malloc size of (struct node));
three = malloc (size of C struct node));

/* Assign data values */
one -> data = 10;
two -> data = 20;
three -> data = 30;
```

```
/* Connect nodes */
one -> next = two;
two -> next = three;
three -> next = NULL;

/* Save the address of first node in the head */
head = one.
```

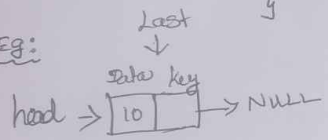
output:



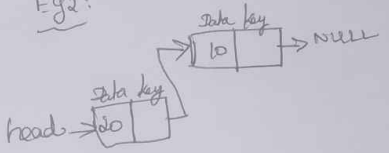
Insertion Algorithm for Singly linked List

```
Procedure INSERT ( link, data, key)
    Struct node *link = (struct node*) malloc (sizeof (struct node));
        link -> data = data;
        link -> key  = key;
    if (ISEMPTY()) {
        last = link;
        link -> next = NULL;
        head = link;
        }
    else
        {
        link -> next = head;
        link = head
        }
```

Eg:



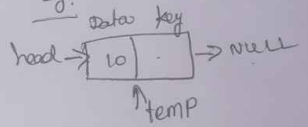
Eg2:



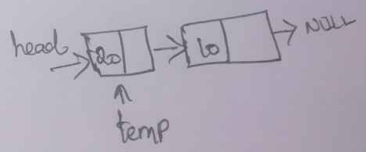
Deletion Operation:

```
Procedure DELETE ( link, data, key)
    Struct node *temp = head;
    if (head -> next == NULL)
        {
        Last = NULL;
        }
    else
        {
        head -> next = head;

        }
    return temp;

}
```

Eg:



Eg 2:



Doubly linked list

A node / link contains three parts

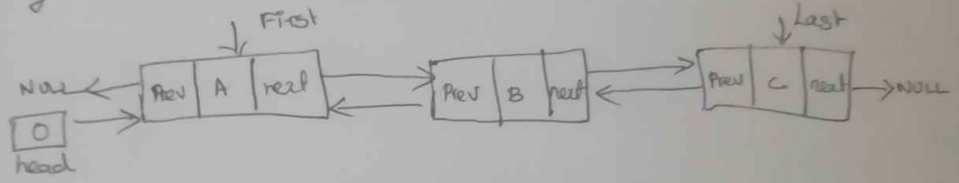
(a) Each link of a linked list can store an element at Data field.

(b) Each link of a linked list contains a connection link to the next link by using next field.

(c) Each link of a linked list contains a connection link to the Previous link by using Prev field.

(d) A Linked list contains the connection link to the first link called First / Head, and contains the connection link to the last link called Last.

Doubly linked list Representation:



Insertion algorithm at First:

```
Procedure INSERTFIRST (link, Prev, next, data)

    // Create a link
    struct node *link = (struct node*) malloc (size of (struct node));
        link->data = data;
        link->key = key;

    if (isEmpty())
    {
        last = link;
    }
    else
        {  head -> Prev = link;
        }  link -> next = head;
```

Deletion at Front / first :

Procedure DELETEFIRST (link, Prev, next, data)

```
Struct node * deleteFirst ()
{
 struct node *temp = head ;
 if (head -> next == NULL)
 {
  last = null ;
 }
 else {
     head -> next -> Prev = NULL ;
    }
  head = head -> next ;
  return temp ;
}
```

Insertion algorithm at last :

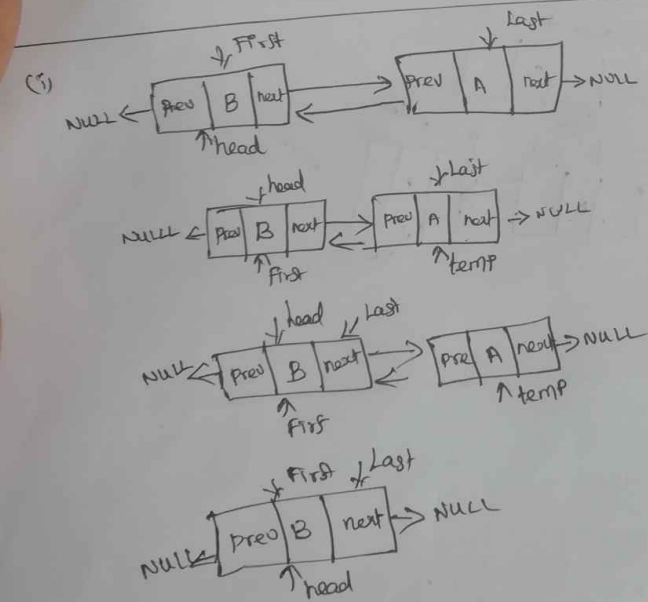
Procedure INSERTLAST (link, Prev, next, data)

```
// Create a link
Struct node * link = (struct node *) malloc (Size of (Struct node));
    link -> data = data ;
    link -> key = key ;
 if (isEmpty ())
  {
    last = link ;
  }
 else
  { last ->next = link ;
  }
    link -> Prev = last
    last = link ;
}
```

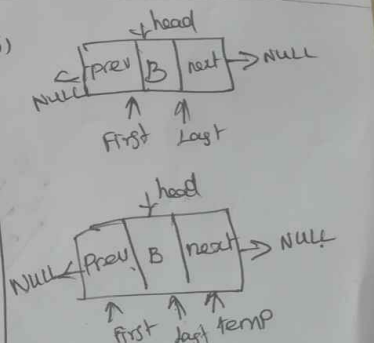
Deletion at Last:

```
Procedure DELETELAST ( link, Prev, next, data)

  Struct node* deleteLast()
    {
      Struct node *temp = Last ;
    if ( Last -> next == NULL)
    {
       Last -> Prev = Last ;
      return temp;
    }
  }
```

(i)



(ii)



Last -> Prev = Last

Last -> Prev = NULL

Last = NULL

Circular Linked List

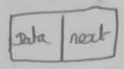
(x) Circular Linked List is different from Singly Linked List and Doubly Linked List.

(x) In Circular Linked List, First link to be Connected to Last link to made Circular, Similarly Last link to be Connected to First link to made Circular.

(x) Both Singly Linked List and Doubly Linked List can be made into a circular linked List.

Singly Linked List:

In Singly Linked List a link Contains Two fields.

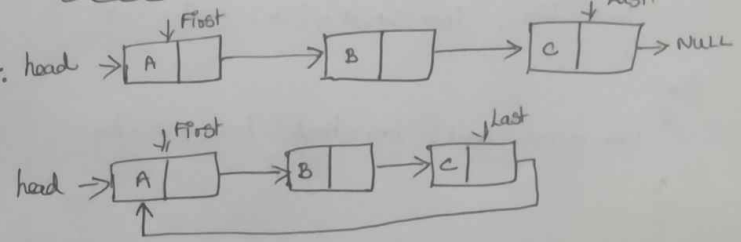


(x) Data Can be Stored at Data field.

(x) Next field can be used to Connection link to next link

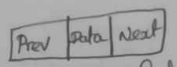
Representation of Singly Linked List to made Circular

Singly Linked List : head → A → B → C → NULL



Doubly Linked List:

In Doubly Linked List a link Contain Three fields.

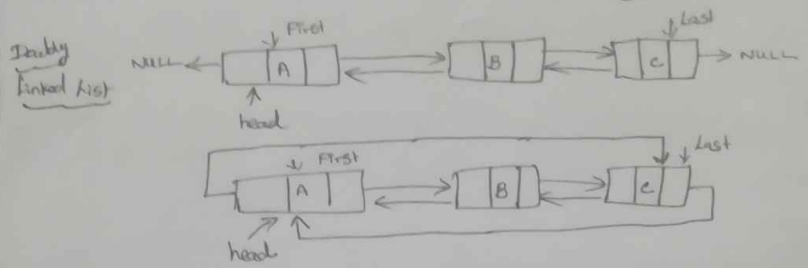


(x) Data Can be Stored at Data field.

(x) Next field can be Used to Connection link to next link.

(x) Prev field Can be Used to Connection link to its Previous link

Representation of Doubly Linked List into Circular:



Insertion Algorithm for Singly linked list to made Circular Linked List:

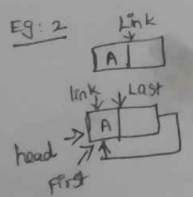
(→)

```
Procedure INSERT ( link )
  // Create a link
  struct node *link = (struct node *) malloc (sizeof(struct node));
        link -> data = data ;
        link -> next = next ;

  if (IsEmpty() )
      {
       Last = link ;  link ->next = link ;  head = link ;
      }
  else
      {  link ->next = head ;  link = head ;  Last ->next = head ;
      }
```

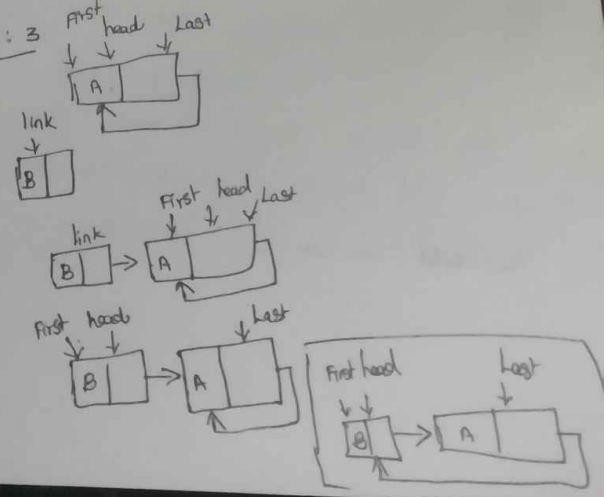
Eg: 1

empty

Eg: 2

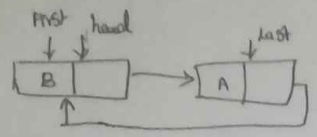


Eg: 3



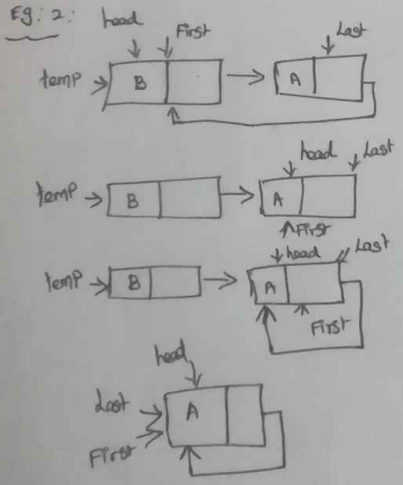
Deletion Algorithm for Singly linked list to make Circular.

Eg 1:

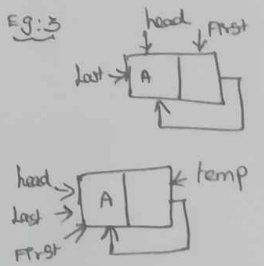



```
Procedure DELETE (link)
    struct node* temp = head.
    if (head -> next == NULL)
    {
      last = NULL;
    }
    else
    {
      head -> next = head;
    }
    last -> next = head;
    return temp;
}
```

Eg: 2:



Eg: 3



last = NULL;

return temp;

NO link in linked list.

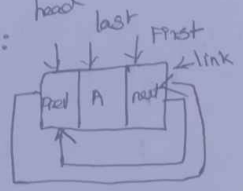
Insertion algorithm for Doubly linked list at first into to made circular Linked List :

```
Procedure INSERTFIRST (link)
// create a link
struct node *link = (struct node *) malloc (sizeof (struct node));
    link -> data = data;
    link -> Prev = Prev;
    link -> next = next;
If (ISEMPTY())
  {
    last = link;
    link -> next = first;
    link -> Prev = last;
  }
else {
      head -> Prev = link; link -> next = head;
      link -> Prev = last; last -> next = first;
     }
    head = link;
}
```

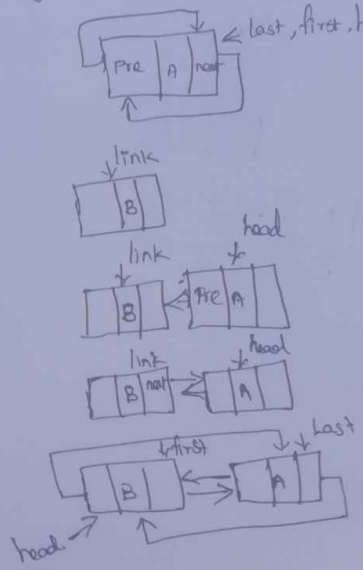
Eg:1

Empty

Eg 2:



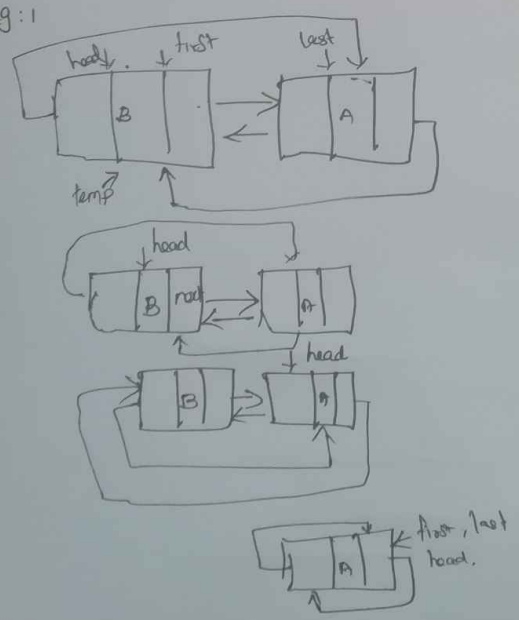
Eg 3:



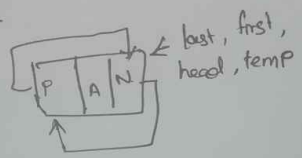
Deletion Algorithm for Doubly linked list at first into to made a Circular Linked list:

```
Procedure DELETEFIRST ( link)

Struct node * deleteFirst ()
{
  struct node *temp= head;
  ~~else (head->~~
  ~~head->~~ ~~next=head~~;
if ( head -> next == head)
  {
    last=NULL;
  }
else
  {
    head ->next= head;
  }
return temp;
}
end DELETEFIRST.
```

Eg : 1



Eg : 2



Arithmetic Expressions:

- Arithmetic Expressions is also known as "Notation".
- Arithmetic Expressions can be written in three different ways but all are Produce same result.

There are three types of Notations.

(i) Infix Notation
(ii) Prefix Notation
(iii) Postfix Notation.

Infix Notation:

< Operand > < Operator > < Operand >

eg: A + B → Operand.
↓ → Operator
Operand

When an Operator is Placed between the two Operands is called Infix Notation.

Prefix Notation:

< Operator > < Operand >

eg: + A

When an Operator is Placed before an Operand is called Prefix Notation.

Postfix Notation:

<Operand> <Operator>

eg:

A+

When an Operator is Placed after the Operand is called Postfix Notation.

Operator Precedence:

When an Operand is Placed between two different Operator means. which Operator can take the Operand for Processing first is decided by the Operator Precedence Rule.

Eg:

A+B*C

Consider B Operand, it placed between two different Operator. + and *, which Operator can take B Operand as first for Processing is decided by Operator Precedence Rule.

Symbol of Operator	Name of Operator	Precedence
( )	Parenthesis	Highest Operator Precedence
↑, ^	Exponent	Second highest
*, /	Multiplication, Divison	Third
+, -	Addition, Subtraction	Last

Associativity:

When an Operand is placed between two different Operator but both the Operator have a same Operator Precedence value means, which Operator can take the Operand first is decided by Associativity Rule.

Eg:

A+B-C

here Operator + and - have same Operator Precedence value.

Operator Symbol	Associativity Rule
( )	-
↑, ^	Right to left
*, /	Left to Right
+, -	Left to Right

Ex:

Convert Infix Expression to Prefix Expression

step 1: Reverse Infix Expression and also reverse '(' becomes ')' and ')' will becomes '('.

step 2: Obtain the Postfix Expression of the modified expression.

step 3: Reverse the Postfix Expression.

we can get Prefix Expression.

Eg 1:

Convert Infix Expression into ~~Pos~~ Prefix Expression

A + B * C

Step 1: Reverse the Infix Expression

C * B + A

You can use stack for this Conversion

Expression	stack	output	Description
C	–	C	Print
*	*	C	Push
B	*	CB	Print
+	+	CB*	Pop * and Push +
A	+	CB*A	Print
End	–	CB*A+	Pop +

Step 2: Obtained modified Postfix Expression is

CB*A+

Step 3: Reverse the Postfix Expression to get Prefix Expression

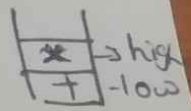
+A*BC

Eg 2:

Infix to Postfix

A + B * C

Expression	stack	Output	Description
A	-	A	Print
+	+	A	Push
B	+	AB	Print
*	+, *	AB	Push
C	+, *	ABC	Print
End	+	ABC*	Pop *
End	Empty	ABC*+	Pop +



ABC*+ is the Postfix Expression.

Eg 3: Prefix to Infix Conversion

+A*Bc

Step 1 : Reverse the Prefix Expression

C B * A +

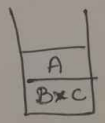
Expression	Stack	Description
C		Push
B		Push
*		Push

Pop B and C.

Concatenate:

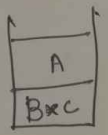
B * C

A



Push

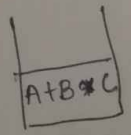
+



Pop A and B * C

Concatenate

A + B * C



End

Pop A+B*C from stack

A+B*C

(i) If the character is an operand Push into stack

(ii) If the character is an operator, Pop the Two operand from stack.

Concatenate :

1st top value + operator + 2nd top value.

(iii) Now Push this result back into Stack.

Eg 1: Prefix to Postfix Conversion:

+A*BC.

Step 1: Reverse Prefix Expression

C B * A +

(i) If Character is an Operand Push into Stack.

(ii) If Character is an Operator Pop the top two operands from Stack.

Concatenate:

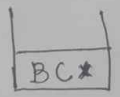
<Operand 1st top> <Operand 2nd top> <Operator>

(iii) Push the new result into stack.

Expression	Stack	Description
C	C	Push
B	B C	Push
*		Pop the top two operands and Concatenate <1st op> <2nd op> <operator>

B C *

(iii) Push the result BC*



A	A BC*	Push, After Pushing A Consider BC* A
*		Pop the top two operand and Concatenate <1st operand> <2nd operand> <operator> A BC *

Push the result ABC* into stack

+	ABC*+	Push.

Result in the stack is ABC*+

Postfix

Eg5 : Postfix to Infix

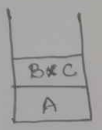
ABC*+

(i) If character is Operand Push into stack.

(ii) If character is operator, Pop the top two operands from stack, Concatenate

<2nd top operand> <operator> <1st top operand>

(iii) Push the result back into stack.

Expression	Stack	Description
A		Push
B		Push
C		Push
*		Pop the top two operands from Stack. C <1st top operand> B <2nd top operand> Concatenate: <2nd top> <op> <1st top> B * C
		(ii) Result B * C Push back to the Stack Consider Stack: A B*C 
+		Pop the top two operands from stack B*C <1st top operand> A <2nd top operand> Concatenate: <2nd top operand> <operator> <1st top operand> A + B * C
		(iii) Result A+B*C Push Back into stack.

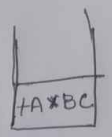
End.

Finally Result in the stack A+B*C

is Infix

Eg 6 : Postfix to Prefix

ABC * +

Expression	Stack	Description
A		Push
B		Push
C		Push
*		Pop the top two Operands from stack C <1'st top Operand> B <2nd top Operand> (ii) Push the result * BC into stack
+		Consider stack :  Pop the top two Operands from stack * * BC <1st top Operand> A <2nd top Operand> Concatenate : <Operator> <2nd top> <1st top> + A * BC (ii) Push the Result + A * BC ~~from~~ back to Posh stack
End		So, Result + A * BC in the stack are prefix Expression.

(i) If character is Operand
Push into stack.

(ii) If Character is an Operator
Pop the top two Operands
from stack and
Concatenate
<Operator> <2nd top Operand>
<1'st top Operand>

(iii) Push the new result back into stack.

Concatenate :
<Operator> <2nd top> <1st top>
* B C

Stack applications :-

Stack :

Stack is an abstract data type and a data structure that follows LIFO (Last In First Out) strategy. It means the elements added last will be removed first.

Applications :

1. Stack data structure is used for evaluating the expression.

Ex : $5 * (6+2) - 12/4$

→ $(6+2) = 8$

→ $5 * 8 = 40$

→ $12/4 = 3$

→ $40 - 3 = 37$.

2. Stacks can be used to check parenthesis matching in an expression.

Ex: $(a+b) * (c+d)$ → Valid expression

$(a+b) * [c+d)$ → Not valid expression.

3. Stacks are used for conversion from one form of expression to another.

Ex:

Infix to prefix, postfix.

Prefix to Infix, postfix.

Postfix to Infix, prefix.

Infix	prefix	postfix
a+b	+ba	ab+
(a+b) * (c+d)	*+dc+ba	ab+cd+*
b*b-4*a*c	-*c*a4*bb	bb*4a*c*

4. Syntax parsing :- Many compilers use a stack for parsing the syntax of expressions, program blocks etc. before translating into low level code.

5. Back tracking : Suppose we are finding a path for solving maze problem. We choose a path and after following it we realize that it is wrong. Now we need to go back to the beginning of the path to start with new path. This can be done with the help of stack.

6. Parenthesis checking :
Stack is used to check the proper opening and closing of parenthesis.

7. String Reversal :
Stack is used to reverse a string. We push the characters of string one by one into stack and then pop character from stack

8. Function call :
Stack is used to keep information about the active functions or subroutines.

9. Stack is used to Redo - undo features at many places like editors, photoshop.

10. It is used to do forward and back - ward features in web browsers.

11. In memory management any modern computer uses stack as the primary-management for a running purpose. Each program that is running in a computer system has its own memory allocations.

12. In Graph algorithms like Topological Sorting and Strongly connected components are using Stack data structure.

13. In Language processing, space for parameters and local variables is created internally using a stack.

14. The stack pattern is also used to keep track of the 'most recently used' feature.

15. In real life stack is using many places. like, a stack of plates, Pile of books in the cupboard, Wearing and removing bangles.

Polynomial Manipulation

(x) Polynomials is an Expression of Variables, that Contains Coefficients and non-negative integer Exponents.

(x) Polynomials can be Used to Perform addition, Subtraction, Multiplication Operations.

Ex:

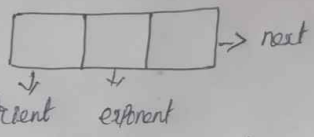
$$15x^{10} + 3x^{8} + 10$$

x = Variable Coefficients are 15, 3, 10
exponents are 10, 8

There are two ways to represent Polynomial
(i) Array (ii) Linked List

Polynomial Representation by Using Linked List

A Link Contains three fields



Each link in a linked list to store coefficient Value at coefficient field, exponent value to store at exponent field and next field is used to Connect another link in the Linked List.

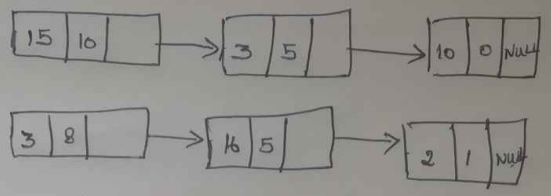
(x) Last link in the Linked List also have a Connection Link to be marked as "NULL".

Eg:

$$\text{Poly 1 } (P) = 15x^{10} + 3x^5 + 10$$

$$\text{Poly 2 } (q) = 3x^8 + 16x^5 + 2x$$

To be represented as follows.



Procedure for Addition:

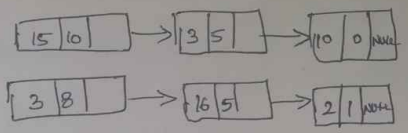
```
Procedure ADDITION ( Poly , Poly1 , Poly2)
Poly1 = P ;  Poly2 = q ;  r = newnode() ;  Poly = r ;
while (P -> next and q -> next)
{
  if (P -> exp > q -> exp)
  {
    r -> coef = P -> coef ;
    r -> exp = P -> exp ;
    P -> next = P ;
  }
  else if (P -> exp < q -> exp)
  {
    r -> coef = q -> coef ;
    r -> exp = q -> exp ;
    q -> next = q ;
  }
  else
  {
    r -> coef = P -> coef + q -> coef ;
    r -> exp = P -> exp ;
    P -> next = P ;
    q -> next = q ;
  }
```

```
if (P->next == null)
{
  r->coef = P->coef;
  r->exp = P->exp;
}
else
{
  r->coef = q->coef;
  r->exp = q->exp;
}
```

Ex:

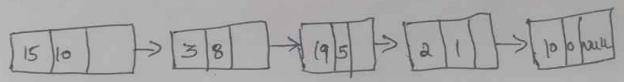
Poly 1 (P) = $15x^{10} + 3x^5 + 10$

Poly 2 (q) = $3x^8 + 16x^5 + 2x$

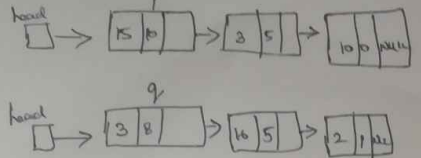


Addition:

Poly (r) (P+q) = $15x^{10} + 3x^8 + 19x^5 + 2x + 10$



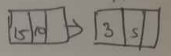
Eg(1)



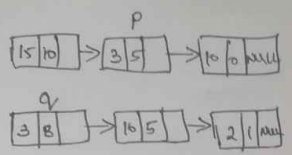
if(P->exp > q->exp) true

r = 15 10

P->next = P;



Eg(2):

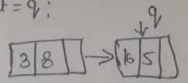


if (P->exp > q->exp) x false

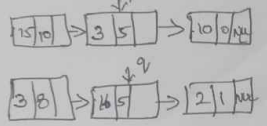
elseif (P->exp < q->exp) true

r = 15 10 → 3 8

q->next = q;



Eg(3):



if (P->exp > q->exp) not Consider true or false

elseif (P->exp < q->exp) not Consider true or false

else

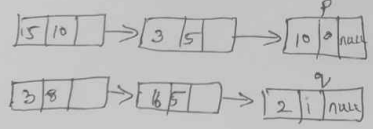
r->coef = P->coef + q->coef

r->exp = P->exp

P->next = P; q->next = q;

r = 15 10 → 3 8 → 19 5

Eg(4):



if (P->exp > q->exp) false

elseif (P->exp < q->exp) true

r->coef = q->coef;

r->exp = q->exp;

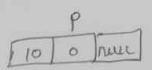
q->next = q;

r = 15 10 → 3 8 → 19 5 → 2 1

q->next = NULL

q = NULL;

Eg(5)



q = NULL

P->next = NULL

if (P->next == NULL) is true

{

r->coef = P->coef;

r->ext = P->exp;

}

r = 15 10 → 3 8 → 19 5 → 2 1 → 10 0 null

Expt. No. :
Date :

Page No. :

Applications of Queue :

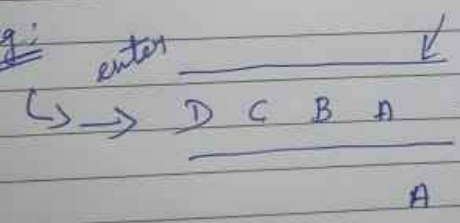
Queue is used when things don't have to be processed immediately, but have to be processed in "First in First out" order like 'Breath first search".

Real world examples :

Patients waiting outside the doctor's clinic :-

- The patient who comes first visits the doctor first, and the patients who comes last visits the doctor last.
- Therefore, it follows the FIFO manner.

eg:



- And Ticket counter also the example of queue.

- phone answering system : The person who calls ~~last~~ ~~gets~~ first gets a response first from the phone answering system. The person who calls last gets the response last.
- Bank line where ~~first~~ people who come first will done his transaction first.
- Key press sequence in keyboard.
- ATM both line.
- single one-way road, where the vehicle enters first, exits first.
- Queue of air planes waiting for landing instruction.
- Job scheduling.

Chapter 1

INTRODUCTION

1.1 WHAT IS AN ALGORITHM?

The word algorithm comes from the name of a Persian author, Abu Ja'far Mohammed ibn Musa al Khowarizmi (c. 825 A.D.) who wrote a textbook on mathematics. An examination of the latest edition of Webster's dictionary defines its meaning as "any special method of solving a certain kind of problem." But this word has taken on a special significance in computer science, where *algorithm* has come to refer to a precise method useable by a computer for the solution of a problem. This is what makes the notion of an algorithm different from words such as process, technique or method.

An algorithm is composed of a finite set of steps, each of which may require one or more operations. The possibility of a computer carrying out these operations necessitates that certain constraints be placed on the type of operations an algorithm can include. For example, each operation must be *definite*, meaning that it must be perfectly clear what should be done. Directions such as "compute 5/0" or "add 6 or 7 to x" are not permitted because it is not clear what the result is or which of the two possibilities should be done. Another important property each operation should have is that it be *effective*; each step must be such that it can, at least in principle, be done by a person using pencil and paper in a finite amount of time. Performing arithmetic on integers is an example of an effective operation, but arithmetic with real numbers is not, since some values may be expressible only by an infinitely long decimal expansion. Adding two such numbers would violate the effectiveness property. An algorithm produces one or more *outputs* and may have zero or more *inputs* which are externally supplied.

Another important criterion we will assume about algorithms in this book is that they *terminate* after a finite number of operations. There is another word for an algorithm which obeys all of the above properties ex-

Chapter 2

ELEMENTARY DATA STRUCTURES

Now that we have presented the fundamental methods we need to express and analyze algorithms you might feel all set to begin. But alas we need to make one last diversion to which we devote this chapter, and that is a discussion of data structures. One of the basic techniques for improving algorithms is to structure the data in such a way that the resulting operations can be efficiently carried out. Though we can't possibly survey here all of the techniques that are known, in this chapter we have selected several which we feel occur most frequently. Maybe you have already seen these techniques in a course on data structures (hopefully having used *Fundamentals of data structures*). If so, you may either skip this chapter or scan it briefly. If you haven't been exposed to the ideas of stack, queues, sets, trees, graphs, heaps, or hashing then lets begin our study of algorithms right now with some interesting problems from the field of data structures.

2.1 STACKS AND QUEUES

One of the most common forms of data organization in computer programs is the ordered or linear list, which is often written as $A = (a_1, a_2, \ldots a_n)$. The a_is are referred to as *atoms* and they are chosen from some set. The null or empty list has $n = 0$ elements. A *stack* is an ordered list in which all insertions and deletions are made at one end, called the *top*. A *queue* is an ordered list in which all insertions take place at one end, the *rear*, while all deletions take place at the other end, the *front*.

Figure 2.1 Example of a stack and a queue

The operations of a stack imply that if the elements A,B,C,D,E are inserted into a stack, in that order, then the first element to be removed/deleted must be E. Equivalently we say that the last element to be inserted into the stack will be the first to be removed. For this reason stacks are sometimes referred to as **L**ast **I**n **F**irst **O**ut (LIFO) lists. The operations of a queue require that the first element which is inserted into the queue will be the first one to be removed. Thus queues are known as **F**irst **I**n **F**irst **O**ut (FIFO) lists. See Figure 2.1 for an example of a stack and a queue each containing the same five elements inserted in the same order. Note that the data object queue as defined here need not necessarily correspond to the concept of queue which is studied in queuing theory.

The simplest way to represent a stack is by using a one-dimensional array, say STACK(1:n), where n in the maximum number of allowable entries. The first or bottom element in the stack will be stored at STACK(1), the second at STACK(2) and the ith at STACK(i). Associated with the array will be a variable, typically called *top*, which points to the top element in the stack. To test if the stack is empty we ask "**if** *top* = 0". If not, the topmost element is at STACK(*top*). Two more substantial operations are inserting and deleting elements. The corresponding procedures are given as algorithms 2.1(a) and (b).

```
procedure ADD(item, STACK, n, top)
  //insert item into the STACK of maximum size n; top is the//
  //number of elements currently in STACK//
    if top ≥ n then call STACKFULL endif
    top ← top + 1
    STACK(top) ← item
end ADD
```

(a) Insertion of an element

```
procedure DELETE(item, STACK, top)
  //remove the top element of STACK and store it//
  //in item unless STACK is empty//
    if top ≤ 0 then call STACKEMPTY endif
    item ← STACK(top)
    top ← top − 1
end DELETE
```

(b) Deletion of an element

Algorithm 2.1 Stacking operations