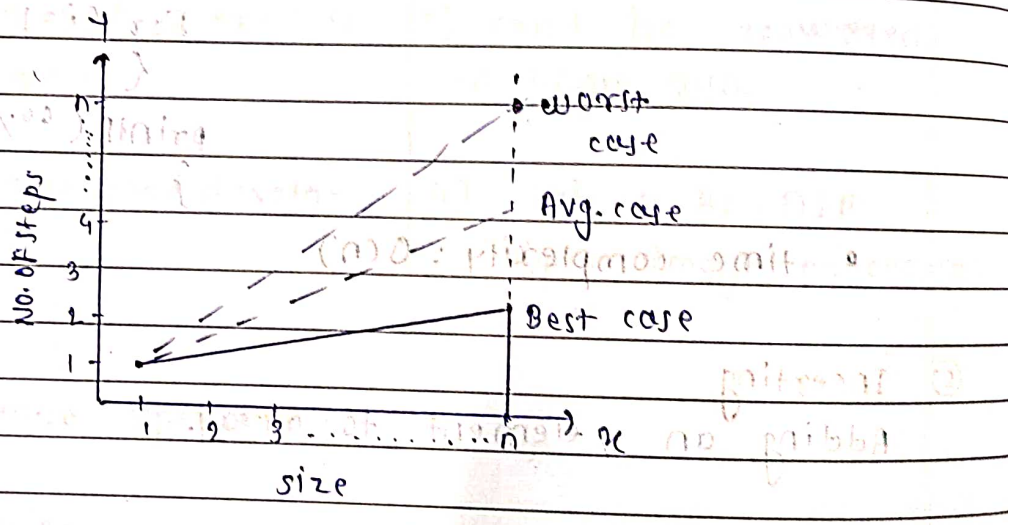
A[N+1] = A[N] O(n) : time complexity
k++; ↑ worst case

③ Insert in between

10	20	11	30	40	50		

O(n)

[Efficient Algorithm : less time + less memory]



Best case

$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < \ldots\ldots n^n$ [case]

O(0)

▷ Algorithm of Insertion

No. of array

Insert (A, N, K, item)

location (item) ↑ to be insert

Algorithm:

Step 1 : set J = N
2 : Repeat step 3 & 4 while J ≥ K
3 : set A[J+1] = A[J]
4 : J = J-1
5 : set A[K] = item
6 : set N = N+1
7 : exit.

for J = 5 A[5] = 50
J ≥ 3
J = 6 A[6] = 50
J-1

N = 5 , item = 11 , K = 3

J = 5 , 4, 3, 2

2 ≥ 3

10	20	30	40	50			→	10	20	11	30	40	50	
1	2	3	4	5	6	7		1	2	3	4	5	6	7

Address of element ⇒ A[i] = b + w (i - lb)

A(1, ..., 100)

b = 1000

size of element = 4 byte ← size of element

loc A[50] = 1000 + (4)(50-1) ← First index value / Index value of item found

= 1000 + 196

= 1196

base address

```
void insert (int [], int);
void main ()
{
int a[10], len, ch, i;
printf("Enter no. of elements you want in array");
scanf("%d", &len);
printf("Enter elements in the array");
for(i=0; i<len; i++)
  {
   scanf("%d", &a[i]);
  }
do
{
printf("select operation you want to perform");
printf("1. insertion");
printf("2. deletion");
printf("3. display");
printf("4. exit");
scanf("%d", &ch);
switch(ch)
 {
 case1 : Insert (a, len);
         break;
 case 2 : Delete ();
         break;
 case 3 : display ();
         break;
 case4 : printf("exit");
         break;
 default : printf("Invalid choice");
         break;
 }
 while (ch != 4);

void insertion (int a[], int len)
{
 int i, pos, num;
 printf("Which element you want to insert");
 scanf("%d", &num);
 printf("Element position");
 scanf("%d", &pos);
 for(i=len-1; i>=pos; i--)
 {
   a[i+1] = a[i]
 }
 a[pos] = num;
 len++;

printf("Array after insertion");  // To print
 for(i=0; i<len; i++)
  {
   printf("%d\t", a[i]);
  }
}

void delete (int a[], int len)   // Deletion
{
 int pos, i;
 printf("Enter position of an element you want
        to delete");
 scanf("%d", &pos);
```

```
for(i=pos; i<len-1; i++)
    a[i] = a[i+1];
}
len = len-1;
for (i=0; i<len; i++)
printf ("%d\t", a[i]);
}
}
```

ADT (Abstract Data Type)

Stack:
① push()
② pop()
③ peek()

Time complexity

	Best case	worst case	Avg. case
push()	O(1)	O(n)	O(1)
pop()	O(1)	O(1)	O(1)
peek()	O(1)	O(1)	O(1)

ADT:
Abstract collection of data element and their accessing function where we are not concerned about how the accessing function will be implemented is referred as ADT.

Also we are not concerned with space and time complexity at abstract level.

eg. Array
the result of operations like insertion, deletion, search, we will get but how it process behind that we will not conclude.

ADT Stack:

Stack is an ordered list of similar type of elements in which an element may be inserted or deleted only at one end. called as the 'top'.

- It follows the principle
  LIFO or FILO
  Last in First out

eg: stack of chairs; stack of plates, etc.

- push() and pull() function are used in stack.
- Top always point to the last element of the stack.

Algorithm

① push()

step 1 : check whether stack is full or not
2 : otherwise increment the top by 1
3 : Insert the element
stack[top] = element

② pop() → if(top == -1)

step 1 : check whether the stack is empty or not.
2 : otherwise decrement the top by 1
3 : Delete the element.

Stack :

Push()

▷ Application of Stack :

① conversion on Mathmetic expression is

a + b ← operand

↑

operator

i) infix expression
ii) prefix expression (polish Notation)
iii) postfix expression (Reverse polish Notation)

① ( ) Bracket
② ^ Exponential
③ * / multiplication and Division
④ + - Addition & subtraction

① For infix :

operand1 , operator , operand 2

eg. (i) a + b
(ii) b(7-4) - (p+q)
A - B
⇒ -AB ⇒ -(-74)(+pq) ← B

② For prefix : Operator Operand1 Operand2

+ a b

③ For postfix : Operant1 Operand 2, operator

ab+
AB-
xy- pq+ -

Imp
① Infix to postfix

Algorithm for converting infix to postfix using stack.

1) push '(' onto the stack & ')'
2) scan exp left to right to end of exp and Repeat step 3 to 6.
3) If an operand encountered p. add to post expression.
4) If left parenthesis encoundered push to stack.
5) If operator encountered then
   ⓐ Repeatedly pop from stack
   ⓑ add to post exp. for each operator same or highest procedure.
   ⓒ add operator to stack.
6) If right parenthesis encountered then,
   a) pop from stack and add to post exp until left parenthesis.
   b) Remove left parenthesis.
7) exit.

eg: (A * B) + (C * D)

p q

pq +

AB* CD* +

eg. 2)

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

Symbol	Stack	Post exp
A	C	
D	C	A
*	C*	A
C	C* C	A
B	C* C	AB
+	C* C +	AB
D	C* C +	ABD*
)	C *	ABD+
/	C * /	ABD+
E	C * /	ABD+ E
-	C -	ABD+ E/*
F	C -	ABD+ E/*F
*	C - *	ABD+ E/*F
C	C - * C	ABD+ E/*F
G	C - * C	ABD+ E/*FG
+	C - * C +	ABD+ E/*FG
H	C - * C +	ABD+ E/*FGH
/	C - * C + /	ABD+ E/*FGH
K	C - * C + /	ABD+ E/*FGHK

$ has highest priority

) C - * ABD+ E/*FGHK/+

) - ABD+ E/*FGHK/+*-

(3) a + (b + c * d + e) + f/g

Symbol	Stack	exp.

④ A$B*C-D+E/F/(G+H)) (Infix to postfix)

Symbol	Stack	Postfix expression
	(	
A	(	A
$	(+$	A
B	(+$	AB
*	(+*	AB$
C	(+*	AB$C
-	(+-	AB$C*
D	(+-	AB$C*D
+	(+	AB$C*D-
E	(+	AB$C*D-E
/	(+/	AB$C*D-E
F	(+/	AB$C*D-EF
/	(+//	AB$C*D-EF
C	(+//C	AB$C*D-EF
G	(+//C	AB$C*D-EFG
+	(+//C+	AB$C*D-EFG
H	(+//C+	AB$C*D-EFGH
)	(+//	AB$C*D-EFGH+
)		AB$C*D-EFGH+//+-

conversion of infix to prefix

Algorithm:-

① Reverse the infix expression

eg. (a+b)*c ⟶ c*(b+a)

② Apply infix to postfix algorithm. to obtain postfix expression

③ Reverse that postfix expression to obtain prefix expression

eg.

(a+b)*c ⟶ c*(b+a))

Symbol	Stack	Postfix exp.
	(	
c	(	c
*	(*	c
(	(*(	c
b	(*(	cb
+	(*(+	cb
a	(*(+	cba
)	(*	cba+
)		cba+*

cba+* ⟶ *+abc

(a+b)*c

P

P*c

*pc

*+abc

$(d-c) * (b-a) \longrightarrow (a-b) * (c-d)$

$\underbrace{(d-c)}_{x} * \underbrace{(b-a)}_{y}$

$*xy$

$*-dc-ba$

Symbol	Stack	exp.
(	(	
a	(	a
-	(-	a
b	(-	ab
)		ab-
*	*	ab-
(	*(	ab-
c	*(	ab-c
-	*(-	ab-c
d	*(-	ab-cd
)	*	ab-cd-
)		ab-cd-*

ab-cd-* $\longrightarrow$ *-dc-ba

② Evaluation of postfix expression

Algorithm

① Add round bracket '(' (closing) at the end of the expression.
② Scan the expression from left to right until round bracket encountered ')'
③ If an operand encountered push it to stack.
④ If an operator encountered push it to then
  1) pop top two operand from stack
  First pop operand is denoted by op1, second pop operand is denoted by op2
  2) Evaluate operand 2 to and operand 1.
  op2 ④ op1
  3) put that answer to stack.
⑤ Top of the stack will be the final answer.
⑥ Exit.

5 6 2 + * 12 4 / -)

Stack Symbol	Stack	
5	5	
6	5 6	op2 op1
2	5 6 2	⑥ + ② → 8
+	5 8	op2 op1
*	40	5 8 → 40
12	40 12	
4	40 12 4	op2 op1
/	40 3	12 / 4 → 3
-	37	op2 op1
)		40 - 3 → 37

4 5 4 2 ^ + * 2 2 ^ 9 3 / * -

Symbol	Stack
4	4
5	4 5
4	4 5 4
2	4 5 4 2
^	4 5 16 — 4^2 = 16
+	4 21
*	84
2	84 2
2	84 2 2
^	84 4
9	84 4 9
3	84 4 9 3
/	84 4 3
*	84 12
-	72

100 200 + 2 / 5 * 7 +)

Symbol	Stack
100	100
200	100 200
+	300
2	300 2
/	150
5	150 5
*	750
7	750 7
+	757
)	

Application of array

1) Linear search

Program:

```
#include <stdio.h>
int main()
{
int a[20], n, item, pos=-1, i, found=0;
printf("Enter number of elements and item to be search\n");
scanf("%d %d", &n, &item);
printf("Enter elements\n");
for(i=0; i<n; i++)
{
scanf("%d", &a[i]);
}
for(i=0; i<n; i++)
{
if(item==a[i])
{
pos=i;
printf("%d item found at %d position\n", item, pos+1);
found=1;
break;
}
}
if(found==0)
{
printf("search unsuccessful");
}
}
```

2) Binary search

Program:

```
#include <stdio.h>
int main()
{
int a[20], n, i, beg, end, mid, item, found=0;
printf("Enter value of n and item to be searched\n");
scanf("%d %d", &n, &item);
beg=0;
end=n-1;
mid=(beg+end)/2;
printf("Input elements\n");
for(i=0; i<n; i++)
{
scanf("%d", &a[i]);
}
for(i=0; i<n; i++)
{
if(item==a[mid])
{found=1;
printf("%d item found at %d position\n", item, mid+1);
break;
}
}
if(item<a[mid])
{
end=mid-1;
if(item>a[mid])
beg=mid+1;
mid=(beg+end)/2;
}
```

```
if (found == 0)
{
printf("search unsuccessful");
}
```

Algorithm for deletion

1. If (front == -1 || front > rear)
   print "queue is empty"
   else
   item = queue[front]
2. Increment front by 1

```
#include <stdio.h>
#define MAXSIZE 5
void enqueue();
void dequeue();
void peek();
void display();
int queue[MAXSIZE];
int front = 0, rear = -1;
void main()
{
 int ch;
 do
 {
printf("1. enqueue Insertion\n");
printf("2. Deletion\n");
printf("3. Displaying top item\n");
printf("4. Display");
printf("5. exit");
printf("Enter your choice");
scanf("%d", &ch);
```

```
switch(ch)
{
case 1: enqueue();
break;
```

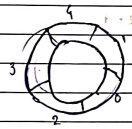
Circular Queue

In normal queue we can insert the element till queue becomes full but once queue becomes full we cannot insert the next element even if there is space in queue. Front of the queue. and thus is the drawback of simple queue(linear queue).

So wastage of memory occurs in simple queue, to utilize this memory a new concept occurs which is circular queue / ring buffer / circular buffer.

In circular queue last position is connected to the first position to make the queue circular.

The main advantage of circular queue is utilization of memory.



Algorithm :

N = max size

F = front

1) Insertion

Step1 : If (Rear+1)

(R+1) % N = F

display "overflow"

2) // If ~~front, R=-1 set F=R=0~~

else

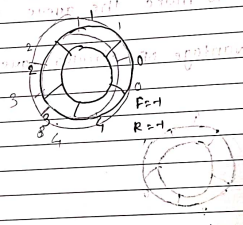
set R=(R+1) % N

Step 3: Q set Q[R] = item

Step 4: exit.

2) Deletion

1. if F = -1
   display "underflow"
2. set item = Q[F]
3. IF F = R set F = R-1
   else
   F = (F+1) % N
4. exit.



complexity

simple queue =

time complexity = O(1) ← deque, enque, peek

display → O(n)

circular queue =

complexity → enqueue → O(1)
dequeue → O(1)
display → O(n)

```
# include <stdio.h>
# define MAXSIZE 5

void enqueue();
void dequeue();
void peek();
void display();
int cqueue[MAXSIZE];
int front = -1, rear = -1;
void main()
{
int ch;
do
{
printf(" 1. Insertion\n");
printf (" 2. Deletion \n");
printf (" 3.
```

Linked list

Ways to maintain a list in memory

↳ Two ways

① Arrays DS

1	2	3	4	5

↓

Arrays have some limitations and disadvantages

↳ linked list overcome these disadvantage

② Linked list

data	addr	→	data	addr

- Types of linked list
  - ① Single linked list
    - Navigation is formed only.
    - we cant go backward
  - ② Doubly linked list
    - Forward and backward navigation is possible.
  - ③ Circular linked list
    - last element is linked to the first element.

Single linked list :

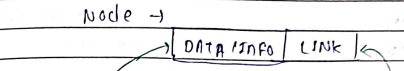
A single linked list is a list made up of nodes that consists of two parts :

① data

② link

Node = data + link
↳ information

Node →



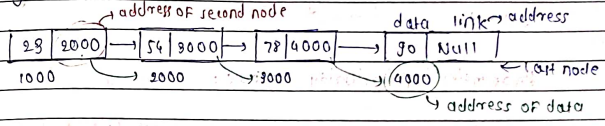
contains the actual data

contains the address of the next node of the list

* linked list is made up of node.

Representation of single list

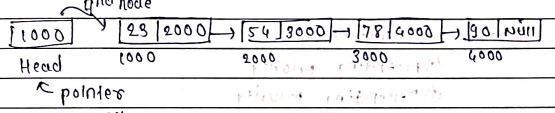
eg. Suppose, we want to store a list of numbers: 23, 54, 78, 90



The data in linked list need not to be in sequential order as it is not an array it is a linked list.

Q. How to access the first node of the linked list?

with the help of pointer we can access the first node of linked list.



3) Priority Queue

Implementation
① Using array
② Using multiple queue
③ Using Linked list
④ Using Heap — max / min

Each element in the queue has associated priority and for deque higher priority element is get deleted.

Properties of priority queue:

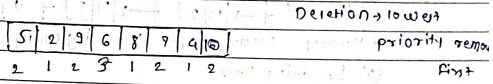
- Based on first in first out approach.
- Every item has priority associated with it.
- An element with higher priority is deque before an element with lower priority
- If two elements have the same priority they are served according to their order in the queue.

There are two type
① Ascending priority
② Descending priority

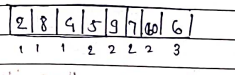
① Ascending
Insertion : It can be normal O(1) or special O(n) insertion.
Deletion : Highest priority (lowest number)
↳ special deletion O(n)
↳ normal deletion O(1).

② Descending
elements with highest priority get removed first.

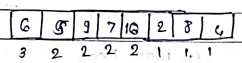
Implementation of priority
① By using Array



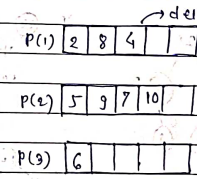
Ascending priority



Descending priority



② By using multiple queues



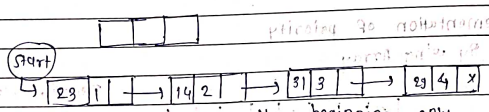
3) By using linked list

Every node is get divided into three parts

① Information

② priority

③ Address of next node

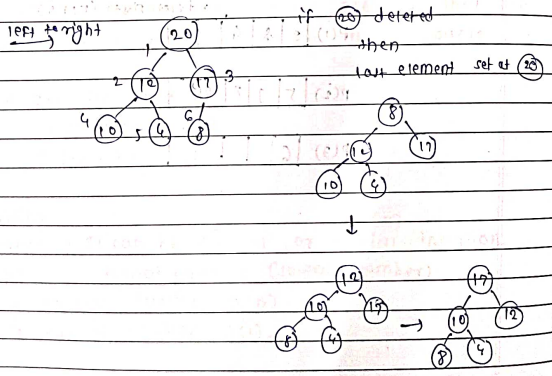


* Deletion occurs from the beginning only.

④ By using Heap

① max Heap

Information in parent heap is greater than its children heap.



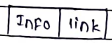
Linked list

It is also known as one way list. It is linear collection of data element called node, where linear order is given by pointer.

Each node is divided into two parts :

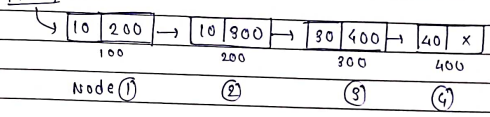
① Info

② Link



Info : Information / data of element

link : Address of next node.



Advantages :

1) Dynamic size

2) Ease of insertion and deletion.

Disadvantage :

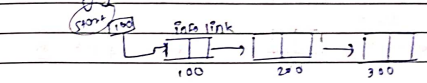
① Random access is not allowed.

② Extra memory used at every node.

Types of Linked list

① Singly (one way list)
② Doubly
③ Circular

① Singly linked list




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

operations of singly linked list

① Traversing

Algorithm:

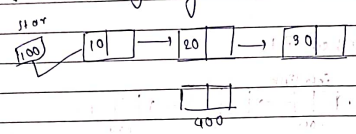
1) set, temp = start ← struct node *temp;
2) Repeat the step ③ and ④ until (temp != null)
   ↳ pointer variable
   ↳ provides address of node
3) Display
   temp → info;
4) temp = temp → next;

For empty condition, (temp == null)

time complexity = O(n)

② Insertion

☆ ① At the beginning



① Allocate memeory to the new node.

```
temp = malloc (sizeof (struct node));
```

② store data

```
temp → info = (40);
```

↳ input element from user

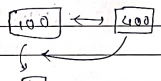
③ Change,
next of new node to point to head.

```
temp → next = start;
```

④ change 'start' with new node

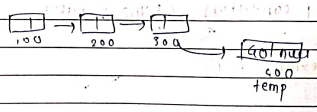
```
start = temp;
```

↳ it will give address (pointer variable)



time complexity = O(1);

☆ ② At the end

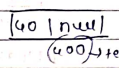


① allocate the memory to new node

```
temp = malloc (sizeof (struct node));
```

② store data.

```
temp → info = 40;
```



③ temp → next = null;

④ traverse the list to the last

```
while (temp → next != NULL)
{
    temp = temp → next;
}
```

⑤ change, next of last node to recently created node.

```
temp → next = newnode
```

time complexity = O(n)

☆ ③ At the middle

① allocate the memory to new node

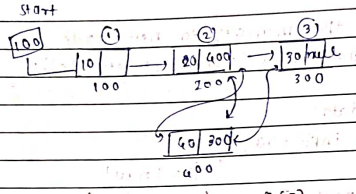
```
temp = malloc (sizeof (struct node));
```

② store data

```
temp → info = 40;
```

③ Traverse the node just before the position.

④ change the pointer according to new node.



time complexity = O(n)

Code (Insertion)

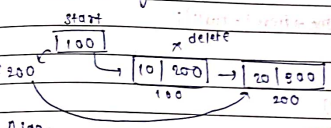
```
void insert()                    newnode → declare
{
 struct node *temp;
 int info, pos, i;
 temp = malloc (sizeof(struct node));
 temp → next = null;   newnode = malloc (sizeof(struct node));
 printf("Enter the element you want to insert");
 scanf("%d", &temp → info);
 if (start == null)
 {
   start = temp;
 }
 else
 {
 printf("At which position you want to insert:");
 scanf("%d", &pos);
  Start code
  for(i=2; i<pos; i++)
  {
  if (temp → next != null)
  {
```

If we have to insert element at 1st position

```
   temp = temp → next;
  }
 }
 newnode → next = temp → next;
 temp → next = newnode;
}
```

② Deletion: complexity O(1)

① From begin



Algo.

① point start start → next

② Delete from end complexity O(n)

Algo.

① traverse the list to the second last node

② change its next point into null.

③ Delete from middle (specified position)

complexity O(n)

p → next = temp → next

```
Struct node *p, *temp;
while (temp! = null)
 {
   p temp
   temp = start
   p = temp
```

Searching

```
do{
if (temp->info == inf)
{
    printf("\nItem found");
}
else
{
    temp = temp->next;
}
}while (temp->next != null);
```

```
do {
}while(temp->next!=null);
```

complexity

① Best case → O(1)

② Worst case → O(n)