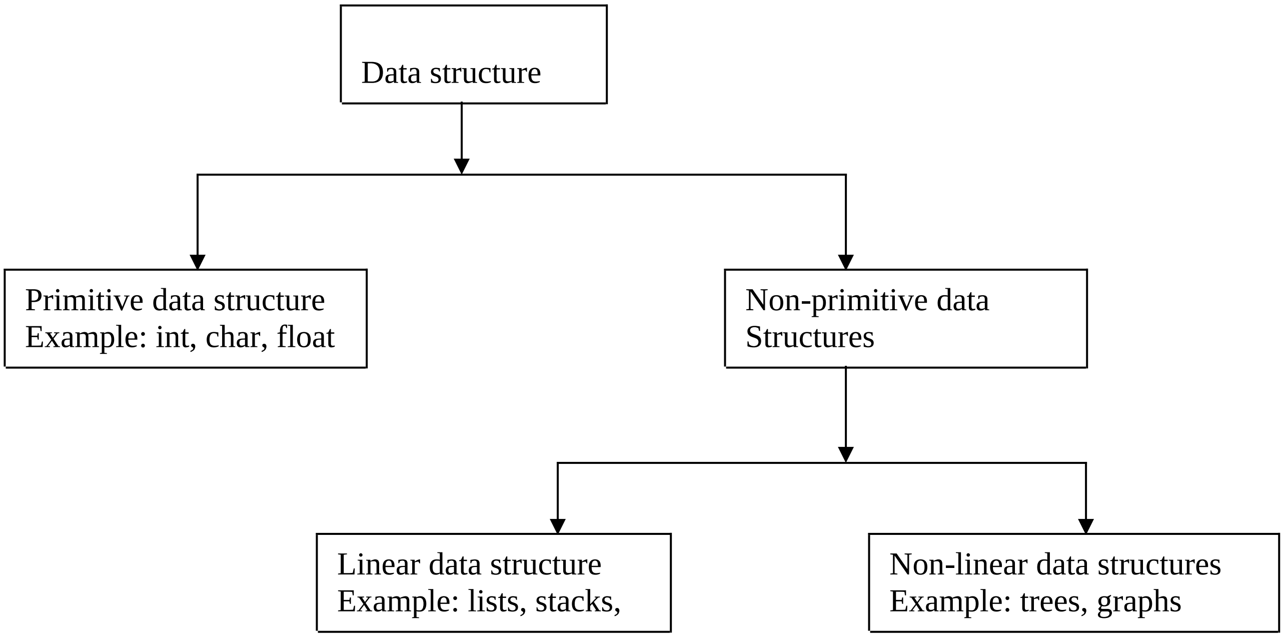
Unit II
LIST

Implementation of data structure can be done with help of programs. To write any program we need an algorithm. Algorithm is nothing but collection of instruction which ahs to be executed in step by step manner. And data structure tells us the way to organize data. To write any program we have to select proper algorithm and data structure. If we choose improper data structure, algorithm cannot work effectively. Thus there is a strong relationship between data structure and algorithm.

The following figure shows the classification of data structure.



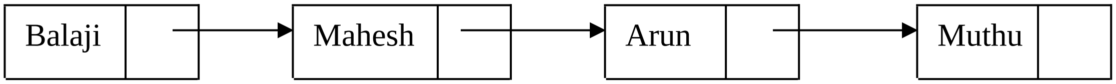
The list is the simplest data structure which represents the linear or sequential data structure. list, stacks and queues are also called as non primitive data structures. Data structures can also be categorized as primitive and non primitive data structures. Data structure which defines the non-primitive data structures are called primitive data structures. For example: integer, character, double, float. With the help of these data structures we can build list, stacks, queues.

Concept of lists

List means collection of elements in sequential order. In memory we can store the list in two ways, one way is we can store the element in sequential memory locations. This is known as arrays. And the other way is we can use pointers or links to associate the elements in sequentially. This is known as linked lists.

Balaji	Mahesh	Arun	Muthu	Senthil

List of sequentially stored elements-using arrays



List of elements associated pointers- using linked list

Arrays

An array is a collection of elements of similar data type. This collection is finite. And the elements are stored at adjacent memory locations. Thus array has to be finite in nature i.e. the size of the array should be specified.

For example: an array of 5 numbers-means the array size should not be less than 5 as well as the elements are numbers (either all are integer values are floating type values but not both)

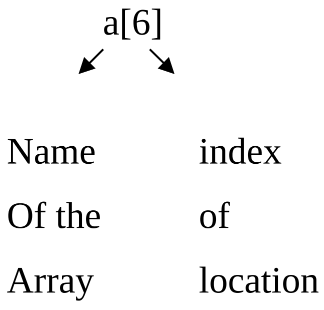
Thus we can say array of n number of elements. Remember usually array elements are starting from 0^{th} location; hence n number of elements can be counted from 0 to n-1. (No doubt, even we can store the elements from any location) the range of array is between a[0] to a[n-1]. (Here a is name of array). Range means total number of elements in the array. All these elements are always stored at contiguous memory locations. Any element of the array can be represented using index and name of the array. That means- a[0] represents the value stored at 0th location of the array, a[2] represents the value stored at 2nd location of the array and so on.

The syntax for array declaration is

Data type array_name[size_of_array];

For example: int a[6]

The array a of size 6, has all the elements which are of integer type.



a[6]

Name Of the Array

index of location

Let us understand such arrangement of array elements by following figure

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
10	20	30	40	50	60	70

Types of arrays

The arrays can be categorized as

1. One dimensional arrays
2. Multidimensional arrays.

One dimensional array

They are called one-dimensional because the data can be viewed entirely in one dimension. What that means is that you can think of the data in the array as being essentially one long chain. As we will see in the next topic, arrays can also extend to 2 dimensions, 3 dimensions, or any number of dimensions that you choose. The effect that the dimensions have on accessing the data is that one index number is needed for each dimension. In a one-dimensional array we can think of the single index as being an element's location in the list.

Example program for one dimensional array

```
/*searching a number*/
#include<stdio.h>
#include<conio.h>
void main()
{
int a[20],i,count,n,x,b;
clrscr();
printf("\n eneter the number of elements:");
scanf("%d",&n);
printf("\n enter the array elements:");
for(i=0;i<n;i++)/*accepting the values in the array using
                                          for loop*/
{
scanf("%d",&a[i]);
}
printf("\n enter the element u want to search:");
scanf("%d",&x);
for(i=0;i<n;i++)  /* searching the number using for loop*/
{
if(x==a[i])
{
count=1;
b=i;
}
}
if(count==1)
printf("match found and the element is stored in %d position:",b);
else
printf("match not found:");
getch();
}
```

Output1

```
eneter the number of elements:3

 enter the array elements:10
20
30
```

```
 enter the element u want to search:10
match found and the element is stored in 0 position:
```

Output2

```
eneter the number of elements:3
 enter the array elements:10
20
30
 enter the element u want to search:40
match not found:
```

Two dimensional arrays

In two dimensional array the elements are stored in rows and columns. Thus to denote rows and columns we required two indices for the array. With the help of the indices we can decide at which position the element is stored in the array. The following figure shows how elements can be stored in two dimensional arrays

	Col0	Col1	Col2
Row0	10 (0,0)	20 (0,1)	30 (0,2)
Row1	40 (1,0)	50 (1,1)	60 (1,2)
Row2	70 (2,0)	80 (2,1)	90 (2,2)

The above array is a[3][3] i.e. of total size of row is 3 and total size of columns is 3. if we want to access the element at position a[2][1] then that will be = 80

The two dimensional array is also called matrix because they represent the structure of matrix.

Example program for two dimensional array

```
/* addition of two matrices*/
#include<stdio.h>
#include<conio.h>
#include<math.h>
void main()
{
int a[20][20],b[20][20],c[20][20],i,j,n;
clrscr();
printf("enter the order of the matrix:");
scanf("%d",&n);
for(i=0;i<n;i++) /*getting the values of a matrix*/
for(j=0;j<n;j++)
{
printf("enter the element for a position %d%d :",i,j);
scanf("%d",&a[i][j]);
}
for(i=0;i<n;i++) /*getting the values of b matric*/
for(j=0;j<n;j++)
{
printf("\n enter the element for position %d%d :",i,j);
scanf("%d",&b[i][j]);
}
for(i=0;i<n;i++) /*adding a mtrix with b*/
for(j=0;j<n;j++)
{
c[i][j]=0;
c[i][j]=a[i][j]+b[i][j];
}
printf("\n the addition of two matrix a and b: is");
for(i=0;i<n;i++) /* printing the resultant matrix*/
{
printf("\n");
for(j=0;j<n;j++)
{
printf("\t%d",c[i][j]);
}
}
getch();
}
```

Output

```
enter the order of the matrix:2
 enter the element for a matrix position 00 :1
 enter the element for a matrix position 01 :1
 enter the element for a matrix position 10 :1
 enter the element for a matrix position 11 :1
 enter the element for b matrix position 00 :1
 enter the element for b matrix position 01 :1
 enter the element for b matrix position 10 :1
 enter the element for b matrix position 11 :1
```

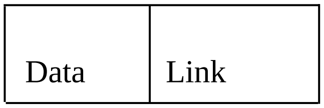
the addition of two matrix a and b is:

```
    2     2
    2     2
```

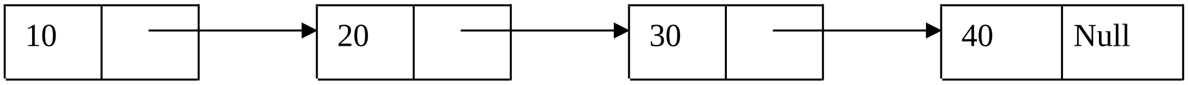
Linked list:

Array is a static representation of list and linked list is a dynamic representation of list. Here static means that in arrays the number of elements is limited to the size of the array. If, total number of elements those are to be stored in the array are very few then the space in the array gets wasted. And dynamic means that as per memory requirement we can allocate the space or we can de-allocate the memory.

In the array the elements are stored in the adjacent memory locations but this is not the condition in the case of linked list. We can define linked list as a collection of similar data items, which are stored in the nodes. Here is a linked node.



And these nodes form a linked list as follows.



```
/*implementation of linked list*/
#include<stdio.h>
#include<conio.h>
typedef struct NODE
{
int data;
struct NODE *next;
}node;
node n1,n2,n3,n4;
node *one,*temp;
void main()
{
clrscr();
n1.data=10;
n1.next=&n2;
n2.data=20;
n2.next=&n3;
n3.data=30;
n3.next=&n4;
n4.data=40;
n4.next=NULL;
one=&n1;
temp=one;
while(temp!=NULL)
{
printf("\n%d",temp->data);
temp=temp->next;
}
getch();
}
```

output:

```
10
20
30
```

40

Advantages Of Linked List Over Arrays:

As we know the basic draw back of static memory and array is management of memory. In both the cases either we create unnecessary extra memory or we get lack of memory. For example if array size we have declared is 50 and we have utilized only 10 locations then rest of the 40 locations get wasted or even reverse can be the situation that means there may be a case that we have to store 100 elements then we have to change the array size from 50 to at least 100. Of course this is to be poor space utilization. So there is a concept called dynamic memory management which came into picture for proper utilization of memory.

In computer world the two words ‘static’ and ‘dynamic’ have great importance. The static refers to an activity which is carried out at the time of compilation of a program and before execution of the program whereas dynamic means the activity carried out while the program is executed.

The static memory management means allocating/deallocating of memory at compilation time while the word refers to allocation/deallocation of memory while program is running. The advantage of dynamic memory management in handling the linked list is that we can create as many nodes as we desire and if some nodes are not required we can deallocate them. Such a dealliocated memory can be reallocated for some other nodes. Thus the scheme results in 100% memory utilization.

Dynamic memory management in C:

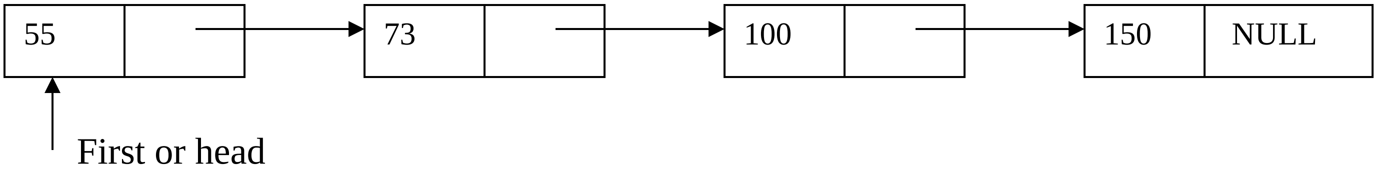
In C language for allocating the memory dynamically ‘malloc’ function is used we should include alloc.h file in our program to support ‘malloc’. Similarly for deallocating the memory ‘free’ function is used.

Types of linked list:

There are various types of linked list such as

1. Singly linear linked list
2. Singly circular linked list
3. Doubly linear linked list
4. Doubly circular linked list

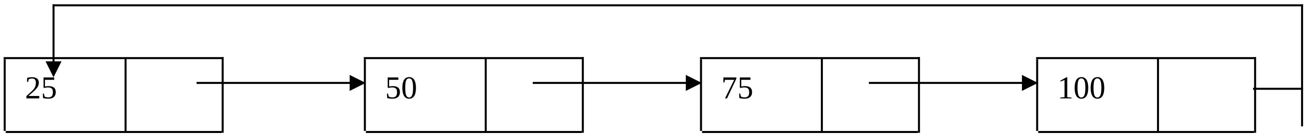
Singly linear linked list:



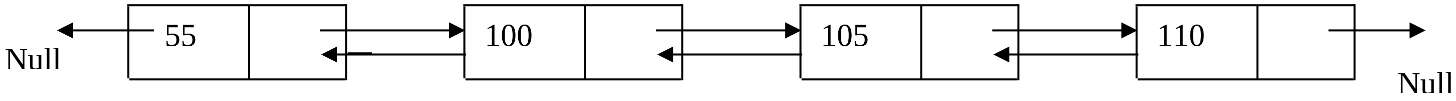
It is called singly because this list consists of only one link, to point to next node or element. This is also called linear list because the last element points to nothing it is linear in nature. The last field of last node is NULL which means that there is no further list. The very first node is called head or first.

Singly circular linked list:

In this type of linked list only one link is used to point to next element and this list is circular means that the last node's link field points to be the first or head node. That means according to example after 100 the next number will be 25.So the list is circular in nature.

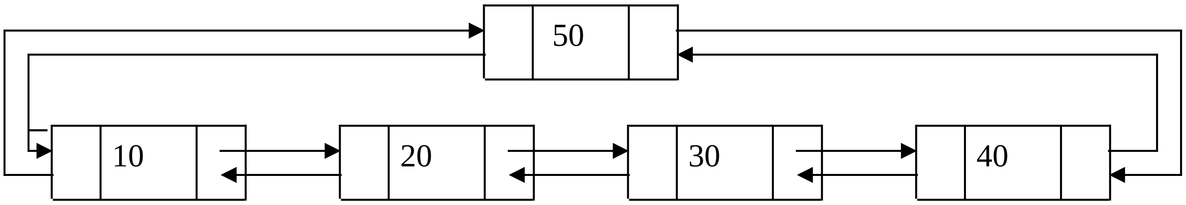


Doubly linear linked list:



The list called doubly because each node has two pointers previous end and next pointers. The previous pointer points to previous node and next pointer points to next node. Only in the case of head node the previous pointer is obviously NULL and last node's next pointer points to NULL. This list is a linear one.

Doubly circular linked list:

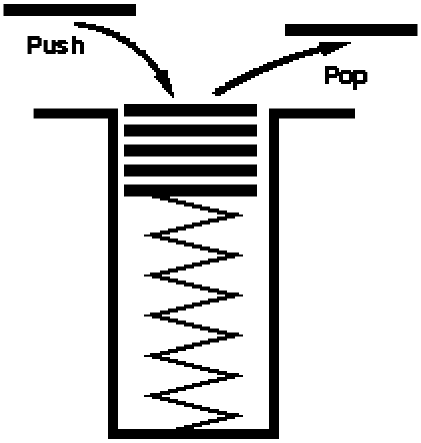


In circular doubly linked list the previous pointer of first node and the next pointer of last node is pointed to head node. Head node is a special node which may have any dummy data or it may have some useful information such as total number of nodes in the list which may be used to simplify the algorithms carrying various operations on the list.

Stack

The stack is an ordered list but the insertions and deletions are restricted by certain rules. For example, in a stack the operations are carried out in a way such that the last element which is inserted will be the first one to come out. Such a order is called last in first out or LIFO. The ordered list does not have such specific ordering and the elements can be inserted and deleted in any random order. Thus the stack may be treated as special case of ordered list.

Now the most important question which arises is that, what is the notation of such specific ordering such as LIFO? Consider a book shelf which is closed form all the sides except the top. How do we keep books onto such a shelf? The only way to arrange one book over other. Now how can we remove a book from it? The only way is to remove all the books, one at a time from the top till the book which we want is removed. So if we want to remove the most recently stacked book, it will be at the top most position and can be removed immediately. On the other hand, if we want to remove the book which we have stacked at the very first time, then we will have to remove all the books which are stacked on top of it, one book at a time. clearly this is an example of last in first out LIFO structure. In computer science, many of the problems can be solved by using a LIFO ordering, i.e. by using stacks. Stacks are used for expression conversion during compilation.



The above figure shows the block diagram of stack.

Representation of stack:

A stack is a special case of an ordered list, i.e. it is a ordered list with some restrictions on the way in which we perform various operations on a list. It is therefore quite natural to use sequential representation for implementing a stack.. to do this, we need to define an array of some maximum size. Moreover, we need an integer variable top which will keep track of the top of the stack as more and more elements are inserted into and deleted from the stack. The declarations in C are as follows

Declaration 1:

```
#define size 100
int stack[size],top=-1;
```

In the above declaration we assume that the elements in the stack are integers and so we will refer to such a stack as integer stack. This stack is capable of storing size number of elements. As we have given # define size 100. the very first element will be at position stack[0], the next at stack[1] and so on. The last element is at position stack[size-1].

Declaration 2:

```
#define size 10
struct stack {
int s[size];
int top;
}st;
```

Now compare declaration 1 and 2. both are for stack declaration only. But the second declaration will be always preferred on why? Because in the second declaration we have used a structure for stack elements and top. By this we are binding or co-relating top variable with stack elements. Thus top and stack are associated with each other by putting them together in a structure. The stack can be passed to the function by simply passing the structure variable. we will make use of the second method of representing the stack in our program.

Stack empty operation:

Initially stack is empty. And that time the top should be initialized to –1 or 0. if we set top to –1 initially then the stack will contain the elements from 0th position and if we set top to 0 initially, the elements will be stored from 1st position, in the stack.. elements may be pushed onto the stack and there may be a case that all the elements are removed from the stack. Then the stack becomes empty. Thus whenever top reaches to –1 we can say the stack is empty.

```
Int stempty()
{
if(st.top==-1)
return 1;
else
return 0;
}
```

stack full operation:

when we create a stack using sequential representation, we need to fix the maximum size of it. We then can go on inserting elements into stack up to this maximum limit. So every time, to insert an element into the stack, we must check whether the stack is full or not. if it is full then we can not insert an element. So we need a mechanism to decide whether the stack is full or not and it is as shown below.

```
Int stfull()
{
```

```
if(st.top>=size-1)
return 1;
else
return 0;
}
```

The ‘stfull’ is a boolean function, which returns 1 if there is no more room to accommodate a new element and returns 0 otherwise. Note that the original value of top is not modified. The top of the stack will be compared with the maximum size –1, since the array starts placing the elements from 0th position.

The ‘push’ and ‘pop’ function:

We will now discuss the two important functions which are carried out on a stack.. push is a function which inserts new element at the top of the stack. The function is as follows.

```
Void push(int item)
{
st.top++;
st.s[st.tp]=item;
}
```

note that the push function takes the parameter item which is actually the element which we want to insert into the stack means we are pushing the element onto the stack. In the function we have checked wheather the stack is full or not, if the stack is not full then the insertion of the elements can be achieved by means of push operation

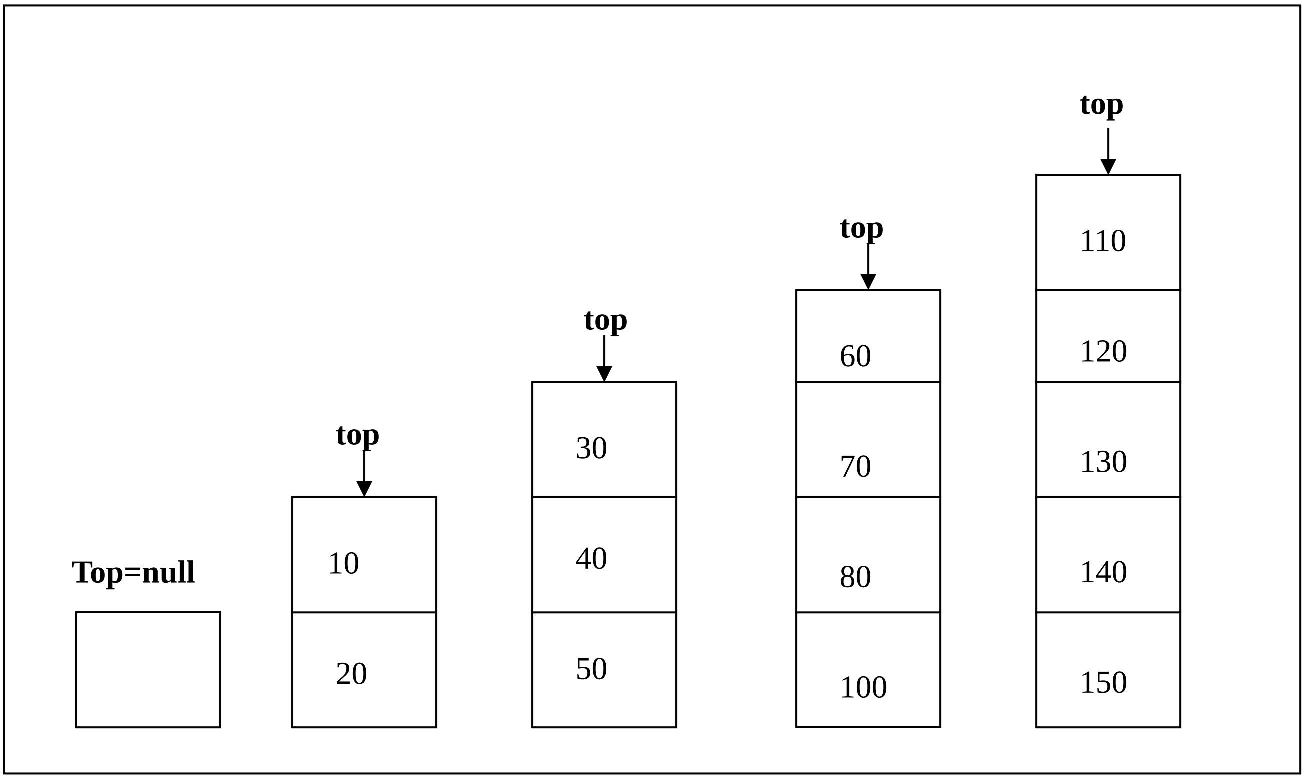
Now let us discuss the operation pop, which deletes the element at the top of the stack. The function pop is given as follows.

```
Int pop()
```

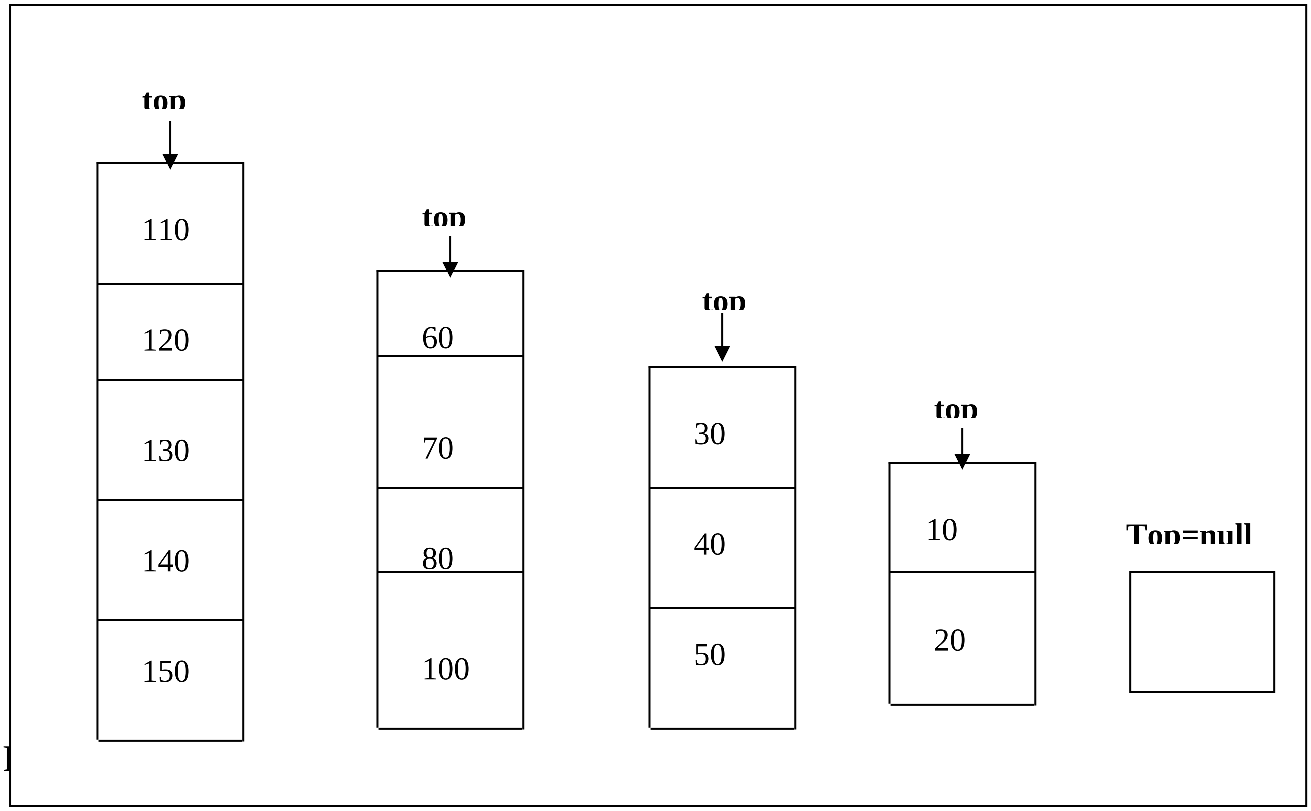
```
{
int item;
item = st.s[st.top];
st.top--;
return(item);
}
```

in the choice of pop it invokes the function ‘stempty’ to determine whether the stack is empty or not. If it is empty, then the function generates an error as stack underflow ! If not, then pop function returns the element which is at the top of the stack. The vale at the top is stored in some variable as item and it then decrements the value of the top, which now points to the element which is just under the element being retrieved from the stack..

Finally it returns the value of the element stored in the variable item. Note that this is called as logical deletion and not a physical deltion, i.e. even when we decrement the top, the element is just retrieved from the stack remains there itself, but it no longer belongs to the stack. Any subsequent push will overwrite this element.



Pictorial representation of stack after inserting elements



I

Stack Implementation:

/*stack implmentation*/

```
#include<stdio.h>
#include<conio.h>
#define size 10
struct stack
{
int s[size];
int top;
}st;
int stfull(void)
{
if(st.top==size-1)
return(1);
else
return(0);
}
void push(int item)
{
st.top++;
st.s[st.top]=item;
}
int stempty(void)
{
if(st.top==-1)
return(1);
else
return(0);
}
```

```
int pop()
{
int item;
item=st.s[st.top];
st.top--;
return(item);
}
void display()
{
int i;
if(stempty())
printf("stack is empty:");
else
for(i=st.top;i>=0;i--)
{
printf("%d",st.s[i]);
}
}
void main(void)
{
int choice,item;
char ans;
st.top=-1;
clrscr();
do
{
printf("\n 1.push \n 2.pop\n 3.display\n 4.exit");
printf("\n enter your choice:");
scanf("%d",&choice);
switch(choice)
{
case 1:
printf("enter the item u want to push:");
scanf("%d",&item);
if(stfull())
printf("stack is full:");
else
push(item);
break;
case 2:
if(stempty())
printf("stack is empty:");
else
{
item=pop();
printf("the poped element is%d",item);
}
break;
case 3:
display();
break;
case 4:
exit(0);
}
```

```
printf("\n do u want to continue?");
ans=getche();
}
while(ans=='y');
getch();
}
```

output:

```

 1.push
 2.pop
 3.display
 4.exit
 enter your choice:1
enter the item u want to push:10

 do u want to continue?y
 1.push
 2.pop
 3.display
 4.exit
 enter your choice:1
enter the item u want to push:20

 do u want to continue?y
 1.push
 2.pop
 3.display
 4.exit
 enter your choice:1
enter the item u want to push:30

 do u want to continue?y
 1.push
 2.pop
 3.display
 4.exit
 enter your choice:3
30
20
10

 do u want to continue?y
 1.push
 2.pop
 3.display
 4.exit
 enter your choice:2
the poped element is30
 do u want to continue?y
 1.push
 2.pop
 3.display
 4.exit
 enter your choice:2
```

```
the poped element is20
 do u want to continue?y
 1.push
 2.pop
 3.display
 4.exit
 enter your choice:2
the poped element is10
 do u want to continue?y
 1.push
 2.pop
 3.display
 4.exit
 enter your choice:2
stack is empty:
 do u want to continue? N
```

Applications of stack:

The place where stacks are frequently used is in expression conversion. An arithmetic expression consist of operands and operators. The operands can be numeric values or numeric variables. The operators used in an arithmetic expression represent the operations like addition, subtraction, multiplication, division and exponentiation.

Various applications of stack are

1. Expression conversion
2. Expression evaluation
3. Parsing well formed parenthesis
4. Decimal to binary conversion

5. Reversing a string

One of the application of stack is conversion of expression. First of all let us see various types of expressions. There are three types of expressions

1. Infix expression:

in this type of expression the operator is in between the operands. which means that if we take + operator which is a binary operator then in infix type it will be operand operator operand i.e.

infix expression: a + b where a and b are the operands and + is an operator.

2. Prefix expression:

In this type of expression the operator come first and then two operands can be placed. The way to write the prefix expression can be

Prefix expression=operator operand operand

For example the infix expression a + b can be written as +ab

3. Postfix expression

In the postfix expression the operator should be written as last i.e. postfix expression = operand operand operator

For example the infix expression a+b can be written as ab+

Let us see few more examples

Infix	**Postfix**	**Prefix**
A + B * C	A B C * +	+ A * B C
(A + B) * C	A B + C *	* + A B C
A - B - C	A B - C -	- - A B C
A ^ B ^ C	A B C ^ ^	^ A B ^ C

Note that the prefix and postfix forms do not make use of parenthesis as these forms are unambiguous. The order of applying operators is unique. In the first example of postfix expression, the operators ‘*’ and ‘+’ appear after the operands. The expression will be evaluated by first applying the operator ‘*’ on B and C and then apply ‘+’ with a and the result. In case of prefix form, similar order can be extracted.

Converting an expression from infix into postfix:

Algorithm for converting infix into postfix expression

1. Scan the expression from left to right.
2. If any operands comes print it simply
3. If any operator comes compare the incoming operator with stack operator. If the incoming operator priority is higher than stack operator priority push the incoming operator.
4. If the incoming operator has less priority than the operator inside the stack then go on popping the operator from top of the stack and print them till this condition is true and then push the incoming operator on top of the stack..
5. If both incoming and stack operator priority are equal then pop the stack operator till this condition is true.
6. If the operator is ‘)’ then go on popping the operators from top of the stack and print them till a matching ‘(‘ operator is found. Delete ‘(‘ from top of the stack..

Let us take the example

Example 1: A * B + C $

Input	**stack**	**Output**
none	empty	none
A	empty	A
*	*	A
B	*	AB

+	Pop * and Push +	AB*
C	+	AB*C
$	empty	AB*C+

Example 2: A * (B + D) / E – F * (G + H / K)

Input	**Stack**	**Output**
A	none	A
*	*	A
(	(*	A
B	(*	AB
+	+(*	AB
D	+(*	ABD
)	* Pop + and delete (	ABD+
/	Pop * and push /	ABD+*
E	/	ABD+*E
-	Pop / and push -	ABD+E/
F	-	ABD+E/F
*	* -	ABD+E/F
(	(*-	ABD+E/F
G	(*-	ABD+E/FG
+	+(*-	ABD+E/FG
H	+(*-	ABD+E/FGH
/	/+(*-	ABD+E/FGH
K	/+(*-	ABD+E/FGHK
)	*- Pop /+ delete (	ABD+E/FGHK/+
None	Pop *-	ABD+E/FGHK/+*-

Converting an expression from infix into prefix:

Algorithm for converting infix into prefix expression

1. Reverse the given expression
2. Scan the expression from left to right.
3. If any operands comes print it simply
4. If any operator comes compare the incoming operator with stack operator. If the incoming operator priority is higher than stack operator priority push the incoming operator.

5. If the incoming operator has less priority than the operator inside the stack then go on popping the operator from top of the stack and print them till this condition is true and then push the incoming operator on top of the stack.
6. If both incoming and stack operator priority are equal then push the incoming operator.
7. If the operator is ‘)’ then push the operator. Repeat the steps 3,4,5,6 for upcoming operators till a matching ‘(‘ operator is found. then go on pop the operators from top of the stack and delete the ‘)’ operator.

Let us take the example

Example 1: A * B + C $

Input	**Stack**	**Output**
A	None	A
*	*	A
B	*	AB
+	Pop * and push +	AB*
C	+	AB*C
$	$ +	AB*C
None	Pop $ +	AB*C$+

Queue:

What is queue?

The queue can be formally defined as ordered collection of elements that has two ends named as front and rear. From the front one can delete the elements and from the rear end one can insert the elements.

For example:

The typical example can be a queue of people who are waiting for a city bus at the bus stop. Any new person is joining at one end of the queue, you can call it as the rear end. When the bus arrives the person at the other end first enters in the bus. You can call it as the front end of the queue.

Following figure-represents the queue of few elements.

94	98	34	58	77	12	32	44

Front Rear

Sequential representation of queue:

Queue is nothing but the collection of items. Both the ends of the queue are having their own functionality. The queue is also called as FIFO i.e. First In First Out data structure. All the elements in the queue are stored sequentially. The various operations on the queue are

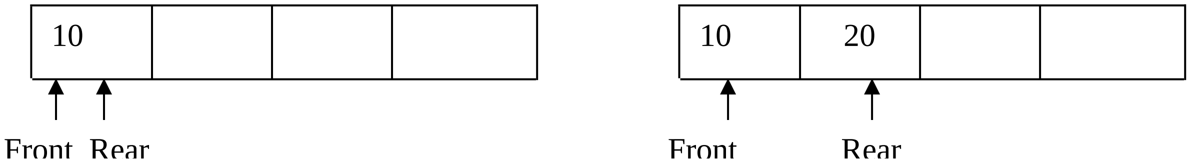
1. Queue Overflow
2. Insertion of the element into the queue.
3. Queue Underflow
4. Deletion of the element from the queue.
5. Display of the Queue.

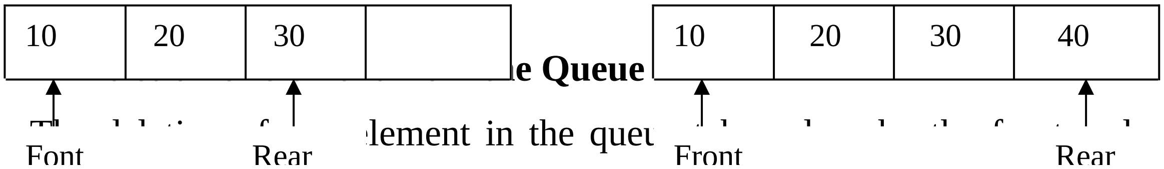
Let us see each operation one by one

1. **Insertion of element into the queue**

The insertion of any element in the queue will always take place from the rear end. Before performing insert operation you must check whether the queue is full or not. If the rear pointer is going beyond the maximum size of the queue then the queue is overflow occurs

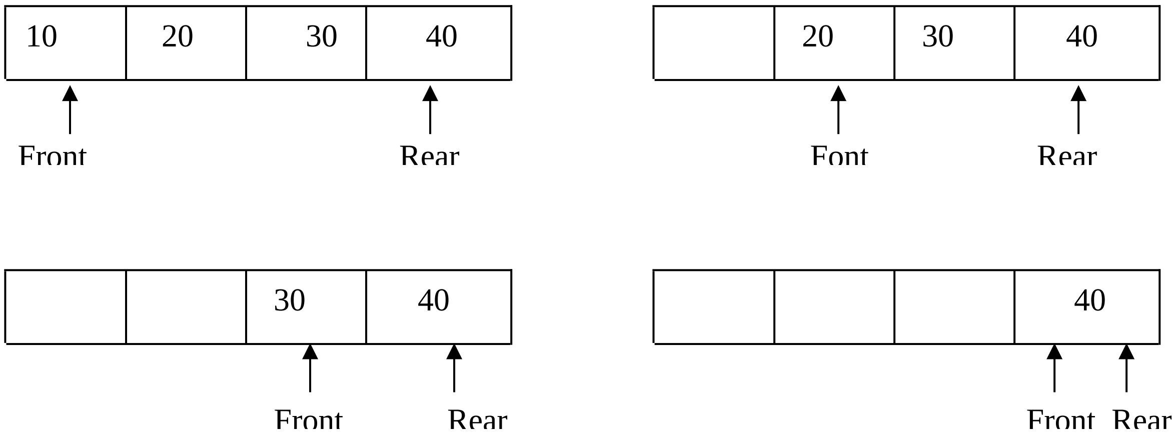
.





e Queue

lement in the queu always. Before performing any delete operation one must check whether the queue is empty or not. If the queue is empty, you can not perform the deletion. The result of illegal attempt to delete an element from the empty queue is called the queue underflow condition.



Queue Implementation:

```
/*Queue Implementation*/
#include<stdio.h>
#include<conio.h>
#define size 10
struct queue
{
int que[size];
int front,rear;
}q;
int full()
{
if(q.rear>=size-1)
return(1);
else
return(0);
}
int insert(int item)
{
if(q.front==-1)
q.front++;
q.que[++q.rear]=item;
return q.rear;
}
int empty()
{
if((q.front==-1)||(q.front>q.rear))
```

```
return(1);
else
return(0);
}
int delete()
{
int item;
item=q.que[q.front];
q.front++;
printf("\n the deleted itemis%d",item);
return q.front;
}
void display()
{
int i;
for(i=q.front;i<=q.rear;i++)
printf("%d\n",q.que[i]);
}
void main(void)
{
int choice,item;
char ans;
clrscr();
q.front=-1;
q.rear=-1;
do
{
```

```
printf("\nmenu");
printf("\n 1.insert \n 2.delete \n 3.display");
printf("\n enter ur choice");
scanf("%d",&choice);
switch(choice)
{
case 1:
if(full())
printf("\n cannot insert element:");
else
{
printf("enter the item u wnat to insert:");
scanf("%d",&item);
insert(item);
}
break;
case 2:
if(empty())
printf("\n queue is under flow:");
else
delete();
break;
case 3:
if(empty())
printf("queue is empty");
else;
display();
break;
}
printf("\n do u wnat to continue?");
ans=getche();
}
while(ans=='y');
}
```

Output:

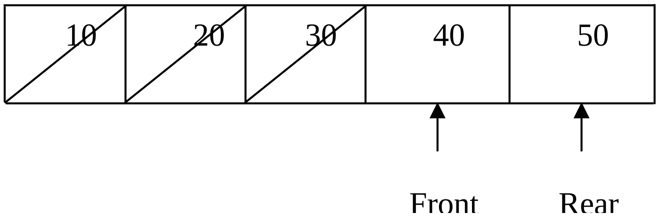
```
menu
 1.insert
 2.delete
 3.display
 enter ur choice1
enter the item u wnat to insert:10
 do u wnat to continue?y
menu
 1.insert
 2.delete
 3.display
 enter ur choice1
enter the item u wnat to insert:20
 do u wnat to continue?y
```

```
menu
 1.insert
 2.delete
 3.display
 enter ur choice
enter the item u wnat to insert:30
 do u wnat to continue?y
menu
 1.insert
 2.delete
 3.display
 enter ur choice3
10
20
30
 do u wnat to continue?y
menu
 1.insert
 2.delete
 3.display
 enter ur choice2
the deleted itemis10
 do u wnat to continue?y
menu
 1.insert
 2.delete
 3.display
 enter ur choice2
 the deleted itemis20
 do u wnat to continue?y
menu
 1.insert
 2.delete
 3.display
 enter ur choice2
 the deleted itemis30
 do u wnat to continue?y
menu
 1.insert
 2.delete
 3.display
 enter ur choice2
 queue is under flow:
 do u wnat to continue?n
```

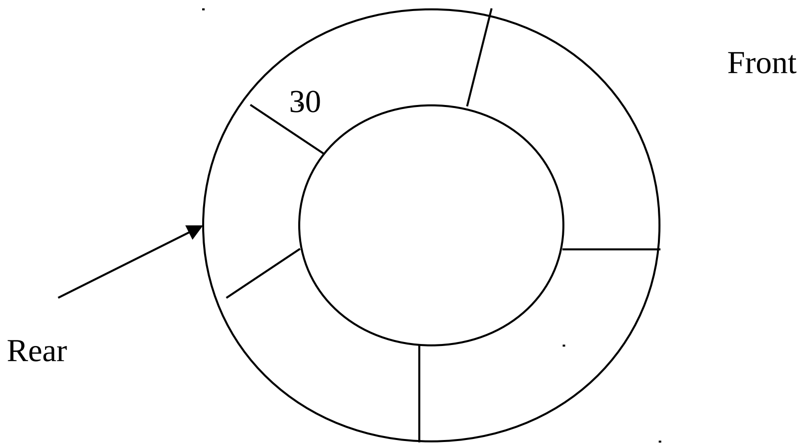
<u>Types of Queue:</u>

<u>Circular queue</u>

In case of linear queue the elements get deleted logically. Following fig can show this



We have deleted the elements 10,20 and 30 means simply the rear pointer is shifted ahead. We will consider a queue from front to rear always. And now if we try to insert any more elements then it won’t be possible as it is going to give “queue full” message. Although there is a space of elements 10,20 and 30 we cannot utilize them because queue is nothing but a linear array. Hence there is a concept called circular queue. The main advantage of circular queue is we can utilize the space of the queue fully. The circular queue is shown below.



There is a formula which has to be applied for setting the front and rear pointers for a circular queue.

Rear = (rear+1) % size

Front = (front+1) % size

For example

Rear= (rear+1) %size

= (4+1) % 5

Rear=0

So we can store the element 60 at 0th location similarly while deleting the element

Front = (front+1) size

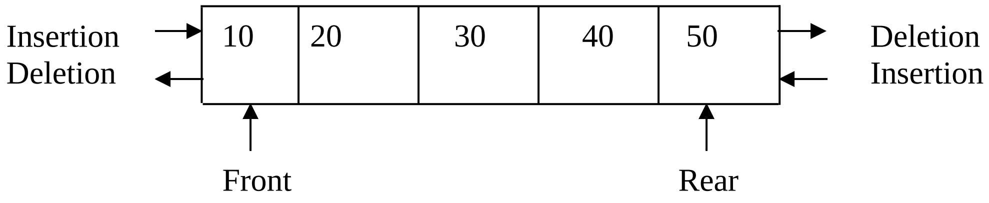
= (3+1) % 5

= 4

So delete the element at 4th location i.e. element 50.

Deque

The word Deque is a short form of double-ended queue and defines a data structure in which items can be added or deleted at either the front end or rear and, but no changes can be made elsewhere in the list. Thus a deque is a generalization of both a stack and a queue. Below figure show the representation of a deque.



As we know, normally we insert the elements by rear and delete the elements from front end. Now if we wish to insert the element from front end then first we have to shift all the elements to the right. If one wishes to perform the deletion operation in the rear end simply decrement the rear pointer by one.

Priority Queue:

A priority queue is a collection of elements where the elements are stored according to their priority levels. The order in which the elements should get added or removed is decided by the priority of the element. Following rules are applied to maintain a priority queue.

(a) The element with a higher priority is processed before any element of lower priority.

(b) If there are elements with the same priority, then the element added first in the queue would get processed first.

There are two types of priority queues

1. Ascending priority queue: it is a collection of item in which the items can be inserted arbitrarily but only the smallest element can be removed first.
2. Descending priority queue: it is a collection of item in which insertion of elements is an order but only the largest element can be removed first.

Applications of priority Queue:

Priority queues are used for implementing job scheduling by the operating system where jobs with higher priorities are to be processed first. Another application of priority queues is simulation systems where priority corresponds to event times.

Applications of Queue

There are various applications of queue such as

Josephus problem:

This is the application of the queue in which the circular queue is being used.

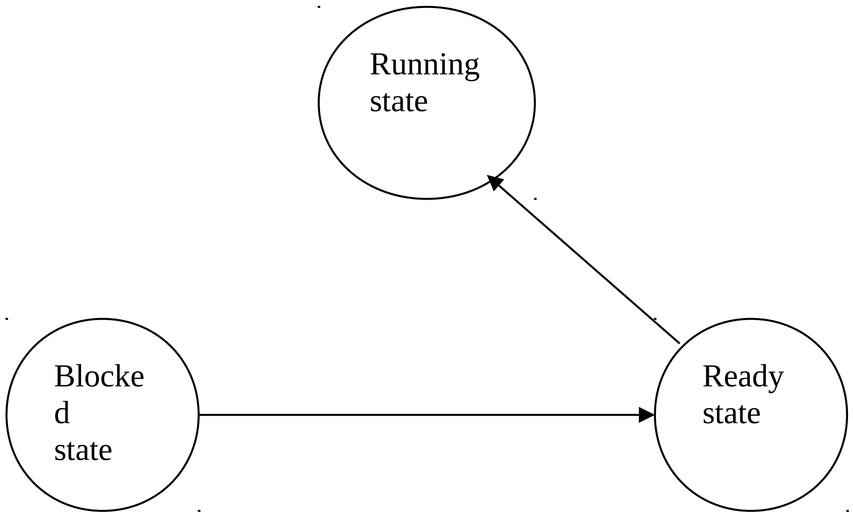
Job scheduling

In the operating system various programs are getting executed.

We cal these programs as jobs. In this process, some programs are in executing state. The state of these programs is called as ‘running’ state.

Some programs are not executing but they are in a position to get executed at any time such programs are in the 'ready' state. And there are certain programs which are neither in running state nor in ready state. Such programs are in a state called as blocked state.

The operating system maintains a queue of all such running state, ready state, blocked state programs. Thus use of queues help the operating system to schedule the jobs. The jobs which are in running state are removed after complex execution of each job, then the jobs which are in ready state change their state from ready to running and get entered in the queue for running state. Similarly the jobs which are in blocked state can change their state from blocked to ready state. These jobs then can be entered in the queue for ready state jobs. Thus every job changes its states and finally get executed.



Abstract Data Type:

In programming, a data type that is defined in terms of the information it can contain and the operations that can be performed with it. An abstract data type is more generalized than one constrained by the properties of the objects it contains – for example, the data type “pet” is more generalized than the data types “pet dog,” “pet bird,” and “pet fish.” The standard example used in illustrating an abstract data type is the stack, a small portion of memory used to store information, generally on a temporary basis.