Data Structures:
A data structure is a process of how the data items are organis in the computer memory for processing and for subsequent usage.

Types of data structures.
Simple data structures: These data structures are generally built using ba data types (int, float, char)
e.g: Arrays, structures.

Compound data structures: These data structures are built from simple data structures and are more complex.
e.g: linkedlist, stacks, queues

Simple data structures
Arrays
Arrays are a collection of related variables which are referred to by common name and elements are placed adjacent to each other.
e.g: int array
1-D → e.g: int a[10];
2-D → eg: in a [10][10];
char array - e.g: char a[10];
(string)

Operations on arrays:
Traversal: It's a method to process each element individually and separately
Searching: It's a method which involves searching a particular element in to find its location or to check whether the element is present not
e.g: linear search, binary search.
Insertion: It involves adding a new element in the list of existing element at a specified location.
Deletion: It involves removing a particular element from the list by using its position

5) Sorting : It is a technique to arrange the elements of the list in a particular order (Ascending, Descending)

) Merging : It is a process to combine 2 or more lists into a single one may or may not be changed to a particular order.

Examples of 1-D arrays

) Linear search

```
{ int a[20], n, x, i, c = 0;
 cout << " Enter the size of the array";
 cin >> n;
 cout << " Enter the elements of the array";
 for(i=0, i <= n-1 ; i++)
 cin >> a[i];
 cout << " The elements of the array are";
 for(i=0; i <= n-1; i++)
 cout << a[i];
 cout << " Enter the element to be searched";
 cin >> x;
 for(i=0; i <= n-1; i++)
 if ( x == a[i])
{ cout << " Element present at " << i+1;
 c = 1;
 break;
}
 if (c == 0)
 cout << " Element absent";
}
```

Note: Linear search starts searching the element from the first element to last element. If no match is found it displays that no match in the array.

Binary Search

```
int a[20], n, x, c=0, l, h, m
cout<<"Enter the size of the array";
cin >> n;
cout << "Enter the elements of the array in ascending order";
for (i=0; i<=n-1, i++)
cin >> a[i];
cout << "The elements of the array are";
for (i=0; i<=n-1, i++)
cout << a[i];
for (l=0; l<=h; )
{ m = (l+h)/2 ;
 if (x = a[m])
{ c = 1;
cout << Element present at<<m+1;
break;

else if (x<a[m])
h = m-1;

if (c==0)
cout << "Element absent";
```

ote: Binary search repeatedly divides the array into 2 parts and focuses e
that half of the array that could contain the target value. It chec
the element to be searched with the middle element. If a match is
found it displays on the screen, else it checks whether the elements to be sear
is less or more than the middle element and accordingly concentrates only
on that half of the array. The process keeps dividing the array till
a match is found or not. Binary search is more efficient than line
search because each iteration reduces the number of comparison

imitation of binary search
Binary search works only on sorted arrays.

rite a C++ program to perform insertion and deletion of a number in a
ray depending on the user's choice.

```
int a[20], n, x, pos, c
clrscr();
cout << "Enter the size of the array";
cin >> n;
cout << "Enter the elements of the array";
for (int i = 0; i < n; i++)
cin >> a[i];

cout << "Enter your choice \n 1 - insertion \n 2 - deletion \n 3 - exit ";
cin >> c;
switch (c)
case 1 : cout << "Enter the element to be inserted and its position";
        cin >> x >> pos;
        pos = pos - 1;
   for(i=n , i >= pos, i--)
        a[i] = a[i-1];
        a[pos] = x;
        n = n+1;
ut << "The new array is";
r (i=0; i<n; i++)
ut << a[i];
eak;
se 2 : cout << "Enter the position to be deleted
        cin >> pos;

r (i = pos ; i < n; i++)
```

```
a[i-1] = a[i]
n=n-1;
cout<<"The new array is";
for (i=0;i<n;i++)
cout<<a[i];
break;
case 3: break;
default: break;
}}
while (choice != 3);
}
```

Sorting: It is a process of rearranging the elements of the array in a particular order (ascending or descending)

Bubble sort:

e.g: If n = 5

pos		
0	44	Iteration I: 33 22 10 44 50
1	33	Iteration II: 22 10 33 44 50
2	22	Iteration III: 10 22 33 44 50
3	10	Iteration IV 10 22 33 44 50
4	50	

```
#include <iostream.h>
#include <conio.h>
{int a[10], n, i, j
cout<<"Enter the size of the array";
cin>>n;
cout<<"Enter the elements of the array";
for (i=0;i<n;i++)
cin>>a[i];
```

```
for(j=1; j<=n-1; j++)
{for(i=0; i<=n-j-1; i++)
if(a[i]>a[i+1])
{t=a[i];
a[i]=a[i+1];
a[i+1]=t;
}
cout<<"After iteration"<<j;
for(int i=0; i<=n-1; i++)
cout<<a[i];
cout<<"\n";
}}
```

Selection sort

e.g. If n=5

Pos	Elements	I	II	III	IV
0	5	1	1	1	1
1	1	5	2	2	2
2	6	6	6	5	3
3	2	2	5	6	6
4	3	3	3	3	5

```
for(j=1; j<=n-1; j++)
{small=a[j];
pos=j;
for(i=j+1; i<=n-1; i++) if(small>a[i])
{small=a[i];
pos=i;
}
if(a[j-1]>a[pos])
{t=a[j-1];
a[j-1]=a[pos];
```

→ Write a C++ program to perform insertion of a number in a sorted array.

e.g. If n = 5

I/p → 5 6 8 11 15

x → (element to be inserted) → 9

o/p → 5 6 8 9 11 15

```
for(i=0; i<n; i++)      // n (length of the array)
{ if(x < a[i])          // x (number to be inserted)
{ pos = i;
  break;
}}
for(i=n; i > pos; i--)
{ a[i] = a[i-1]; }
a[pos] = x; n = n+1;
}
cout << "The new array is";
for(i=0; i<n; i++)
cout << a[i];
```

n = 5
i = 0
1
pos = 3

Using array write a program to convert from decimal to binary.

2	10
2	5 — 0
2	2 — 1
	1 — 0

```
void main()
{ int n, a[20], i, d, k, h;
  cout << "Enter the decimal number to be converted";
  cin >> n;
  while (n>0)
```

Linked List

A linked list consists of nodes containing two parts, data and a pointer. The pointer points to the address of the next node.

Difference between an array and a linked list.

1. An array is a static data structure, i.e. its size is predefined and fixed and does not change during run-time. A linked list is a dynamic data structure and hence its size is created during run-time.
2. Array stores data of similar types. Linked list stores data and a linked part in every node where one link points to the next node.
3. Insertion of an element in an array requires shifting of elements. Insertion of an element in a linked list does not require shifting of elements and a node can be inserted at any location by just changing the direction of pointers.
4. Lot of memory space is wasted and the arrangement for any element has to be done before taking the data. No memory wastage in linked list since memory is utilised during run-time and the data for every element (node) can be stored during run-time.

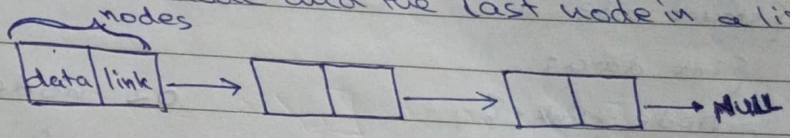
Note: Linked list is generally implemented using structures, since structures can hold data of different types.

Difference between an array and a structure.

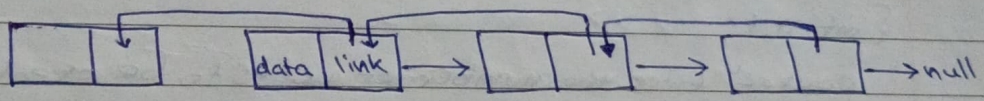
1. Array holds elements of the same type. Structures can hold data of multiple data types.
2. An array is a derived data type (derived from built-in data types, int, char, etc.). Structure is a programmer defined data type
3. Array behaves like a built-in data type. All we need to do is to declare an array variable and use it. For structures, first we have to design and declare data structures before the variables of that type are declared and used.

Types of Linked List

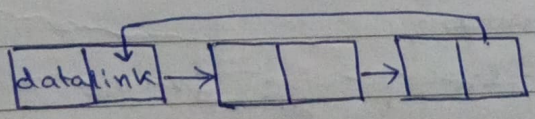
1. **Singly Linked List** – consists of nodes containing two parts. that is data and a pointer and the pointer points to the address of the next node and the last node in a list points to Null.



2. **Doubly Linked List** – consists of nodes containing two parts, i.e. data and a pointer and the pointer points to the address of the previous node and the next node



3. **Circular Linked List** – consists of nodes having data and link parts and the link of the last node points to the first node in the list.



Operations performed on a linked list.

1. Creation of a linked list (n nodes)
2. Traversing through linked lists
3. Insertion of a node in a linked list.

4. Deletion of a node from a linked list.
5. Displaying the final linked list.
6. Searching whether a particular element (node) is present in a linked list.
7. Counting the total number of nodes in a linked list.

Data structures implemented using linked lists are stacks and queues.

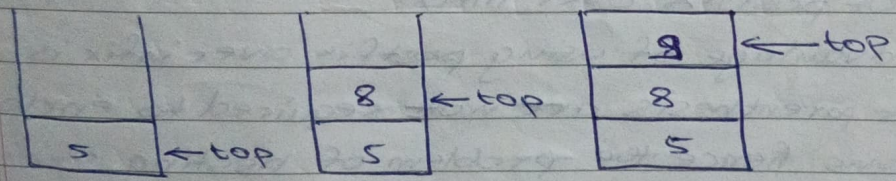
Operator new and delete allows the user to allocate memory and deallocate memory respectively dynamically (during execution of the program).

Stacks

A last in first out data structure (LIFO) as the last element inserted on the stack is the first to be removed from the stack. In a stack, insertion or deletion can take place at only one end called the top of the stack. Insertion of an element on the stack is called pushing onto the stack and deletion is called popping from the stack.

Operations on stacks.

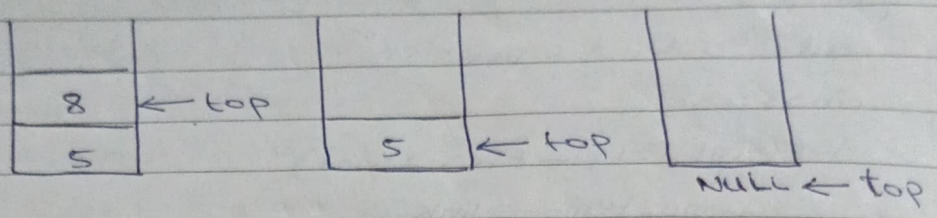
1. Push - Insert element 5, 8 and then 9 onto the stack. Display the stack after every insertion. Also show the pointer at every insertion.



When a stack is full and no more elements can be pushed onto

the stack, an overflow takes place.

2. Pop -



If a stack is empty and it is not possible to pop from the stack then underflow takes place.

Applications of stacks

Since stacks are LIFO structures, it is a useful data structure for applications in which information must be stored and later retrieved in reverse order.

Conversion from infix to postfix.

1. Types of expressions.

Infix expressions- An expression where the operator is in between the two operands. (A+B)

Postfix expressions- An expression where the operator is after the two operants. (AB+)

Prefix expressions- An expression where the operator is before the two operants. (+AB)

Advantages of postfix over infix

- the main advantage of using postfix over infix is
- brackets or parenthesis are not required to enclose the operations hence the problem of nesting of expressio is solved. Every operator in a postfix expression is placed according to its precedance or priority.

Priority	Operators
1	( )
2	*, /, %, ^/↑
3	+, -

Algorithm for conversion from infix to postfix expression

1. When an operant is encountered, it is immediately placed onto the output.
2. When an operator is encountered, it is placed onto the stack
3. If we see right parenthesis, then we pop the stack writing symbols until we encounter a corresponding left parenthesis.
4. If we see another symbol, then we pop the entries from the stack until we find an entry of lower priority. When the poping is done, push the operator onto the stack.
5. Finally we read the end of input, pop the stack until it is empty, writing symbols onto the output. The expression obtained, is the final postfix expression.

① A * B + (C - D / F)

I/p	stack status.	postfix
A	$	A
*	*	A
B	*	AB
+	+	AB *
(	+(	AB *
C	+(	AB * C
-	+(-	AB * C
D	+(-	AB * C D
/	+(-/	AB * CD
F	+(-/	AB * CDF
)	$	⇒ AB * CDF/-+

② A + B * C + D / E - F

I/P	stack status	O/P
A	$	A
+	+	A
B	+	AB
*	+*	AB
C	+*	ABC
+	+	ABC*+
D	+	ABC*D+
/	+/	ABC*D+
E	+/	ABC*D+E
-	-	ABC*D+E/+
F	-	ABC*D+E/+F
$	$	⇒ ABC*D+E/+F-

③ (A+B) * C+D/E-F

I/P	stack status	O/P
(	(	$
A	(	A
+	(+	A
B	(+	AB
)	$	AB+
*	*	AB+
C	*	AB+C
+	+	AB+C*
D	+	AB+C*D
/	+/	AB+C*D
E	+/	AB+C*DE
-	-	AB+C*DE/+
F	-	A·B+C*DE/+
$	$	⇒ AB+C*DE/+

④ A+B*(C-D)/E

I/P	Stack status	O/P
A		A
+	+	A
B	+	AB
*	+*	AB
(	+*(	AB
C	+*(	ABC
-	+*(-	ABC
D	+*(-	ABCD
)	+*	ABCD-
/	+/	ABCD-*
E	+/	ABCD-*E
$	$	⇒ ABCD-*E/+

⑤ A-B+C*D%E*G/H ⇒ AB-CD*E%G*H/+

⑥ A+(B*C-(D/E%F)*G)*H

I/P	Stack status	O/P
A	$	A
+	+	A
(	+(	A
B	+(	AB
*	+(*	AB
C	+(*	ABC
-	+(-	ABC*
(	+(-(	ABC*
D	+(-(	ABC*D
/	+(-(/	ABC*D
E	+(-(/	ABC*DE
%	+(-(%	ABC*DE/
F	+(-(%	ABC*DE/F
)	+(-	ABC*DE/F%

*	+(-A	ABC*DE/F%0
G	*(-*	ABC*DE/F%0G
)	+	ABC*DE/F%0G*-
*	+*	ABC*DE/F%G*-
H	+*	ABC*DE/F%0G*-H
$	$	⇒ ABC*DE/F%0G*-H*+

⑦ A-B+C+D%0E*G/H ⇒ AB-C+DE%0G*H/+

⑧ A+(B*C-(D/E↑F)*G)*H ⇒ ABC*DE/F↑G*-H*+

⑨ A-B+C*D↑E*G/H ⇒ AB-CD*E↑G*H/+

⑩ x-y/(z+u)*v ⇒ xyzu+/v*-

⑪ (A+B)*C+D/E-F ⇒ AB+C*DE/+F-

⑫ A+B*(C-D)/E+F-G ⇒ ABCD-*E/+F+G-

20*(8/4) 2+3 2*5-10

Evaluate the postfix expressions.

① 20,8,4,/,2,3,+,*,-

	Action/stack	Result
20	20	$
8	20,8	$
4	20,8,4	$
/	pop4, pop8, divide 8 by 4 — 20,2	$
2	20,2,2	$
3	20,2,2,3	$
+	perform 2+3 — 20,2,5	$
*	20,10	

classmate
Date
Page

10 10

(2) 10, 40, +, 8, 2, +, *, 10, − ⇒ 490

(3) 5, 3, 2, 4, +, 5, *, +, 6, +, − ⇒ −34

(4) 50, 40, +, 18, 14, −, 4, *, + ⇒ 106

(5) 500, 20, 30, +, 10, *, + ⇒ 1000

(6) 5, 11, −, 6, 8, +, 12, *, / ⇒ −1/28

(7) 100, 40, 8, +, 20, 10, −, +, * ⇒ 5800

(8) 5, 6, 9, +, 80, 5, *, −, / ⇒ −1/77

(9) ABCD − * E/+ ⇒ A + B*(C−D)/E

(10) AB*CDF/−+ ⇒ A*B+C−D/F

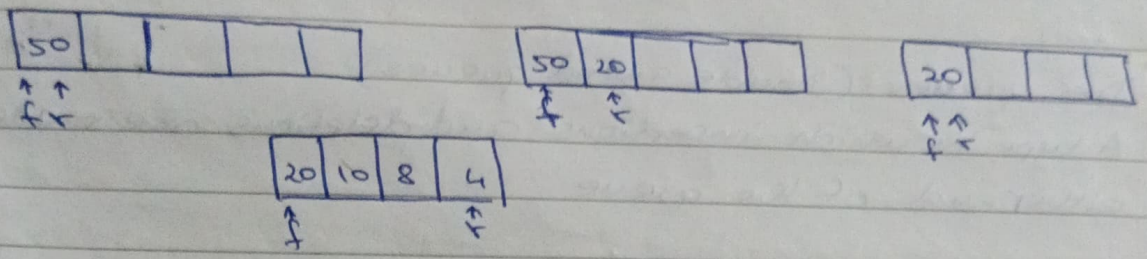
Queues - A first in first out data structure and insertion in a queue is done at the rear end and deletion is done from the front.

Types of queues

1. Linear queue - A first in, first out data structure consisting of two pointers, front and rear. An element is inserted at the rear end and deleted from the front end.

Consider a queue of size 5 performing the following operations and also show the position of front and rear pointers after each operation.

① Insert 50
② Insert 20
③ Delete 50
④ Insert 10
⑤ Insert 8
⑥ Insert 4



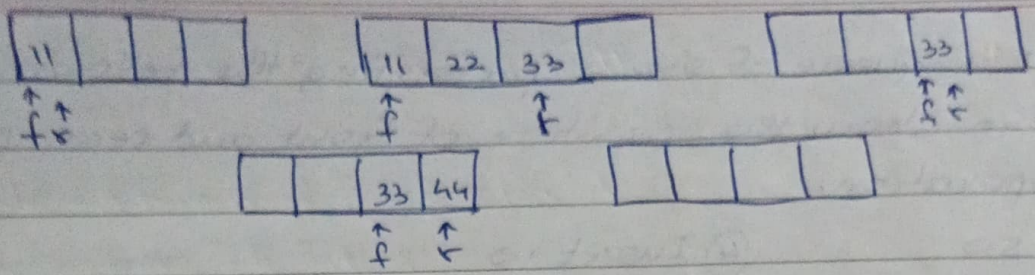
If the user tries to insert above queue, overflow takes place and an error message could be displayed

2. Circular queues – the first element of the queue is made to be present immidiately after the last element of the queue, i.e. if the last position of the queue has an element, a new element can be inserted next to it. At the first position, the front and rear pointer is together and as the insertion or deletion takes place, the pointers move to store the current front and rear resp. in the queue

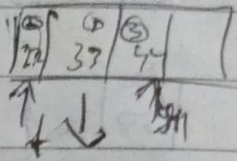
Operations.

consider a queue of size 4 and perform the following operation and also display the position of front and rear pointers after each operation

① Insert 11
② Insert 22
③ Insert 33
④ Delete 11
⑤ Delete 22
⑥ Insert 44
⑦ Insert 55
⑧ Delete 33
⑨ Delete 44
⑩ Delete 55



Hence circular queues are better than linear queues as it allows reusing memory locations.



3. DE queues (Double ended queues)
A type in which insertions and deletions are made from either end of the queue

Priority queues – A queue in which insertion or deletion of elements at any position happens depending on some priority