SUBJECT : DSA

Linked list Data Structure

A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node. For example

The address of the first node a special name called HEAD. Also, the last node in the linked list can be identified because its next portion points to NULL.

Representation of Linked List

Each node consists:

A data item

An address of another node

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

Why linked list data structure needed?

- **Dynamic Data structure:** The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.

- **Ease of Insertion/Deletion:** The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
- **Efficient Memory Utilization:** As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
- **Implementation:** Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

Difference between Arrays and Linked list

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

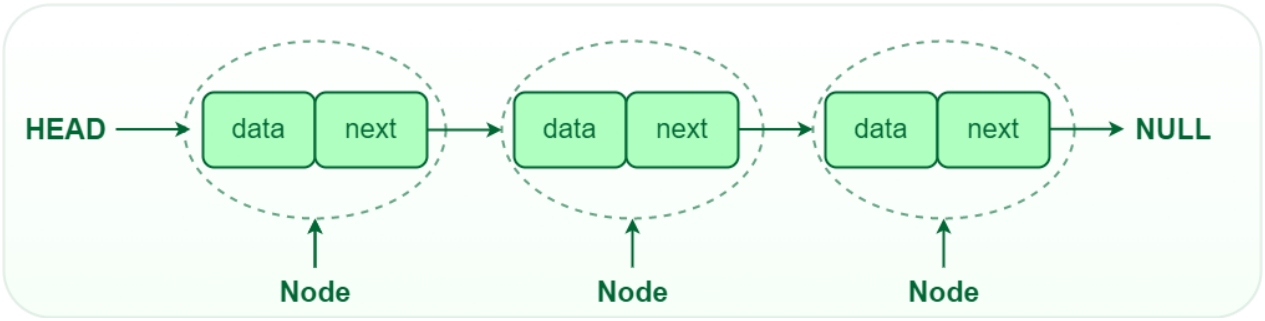
Types of linked lists:

There are mainly three types of linked lists:

1. Single-linked list
2. Double linked list
3. Circular linked list

Single-linked list:

In a singly linked list, each node contains a reference to the next node in the sequence. Traversing a singly linked list is done in a forward direction.

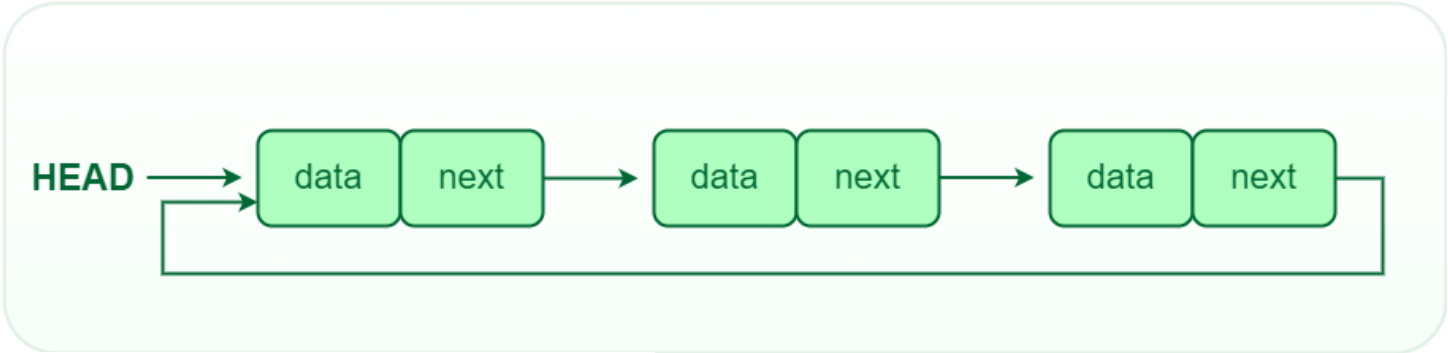


. [Double-linked list]():

In a doubly linked list, each node contains references to both the next and previous nodes. This allows for traversal in both forward and backward directions, but it requires additional memory for the backward reference.

[Circular linked list]():

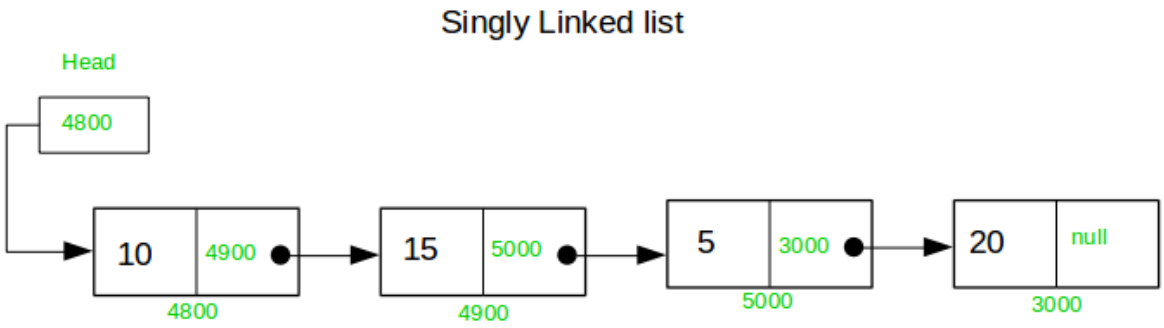
In a circular linked list, the last node points back to the head node, creating a circular structure. It can be either singly or doubly linked.



Operations on Linked Lists

1. [Insertion](): Adding a new node to a linked list involves adjusting the pointers of the existing nodes to maintain the proper sequence. Insertion can be performed at the beginning, end, or any position within the list

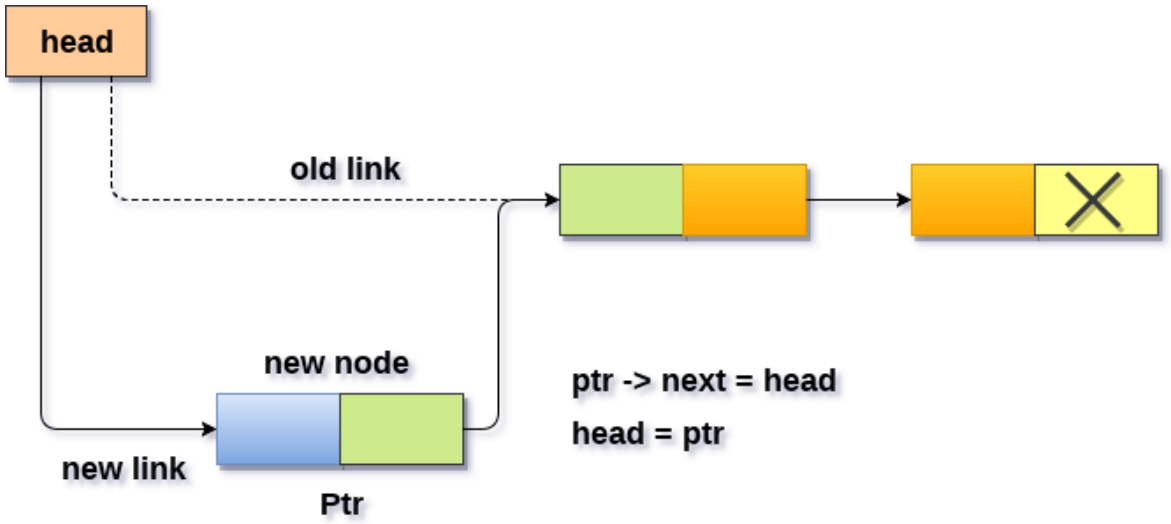
2. Deletion: Removing a node from a linked list requires adjusting the pointers of the neighboring nodes to bridge the gap left by the deleted node. Deletion can be performed at the beginning, end, or any position within the list.
3. Searching: Searching for a specific value in a linked list involves traversing the list from the head node until the value is found or the end of the list is reached.



Operations on Singly Linked List

```
struct node
{
   int data;
   struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

Insertion in singly linked list at beginning



Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links. There are the following steps which need **to** be followed in order **to insert a new node in the list at beginning.**

- Allocate the space for the new node and store data into the data part of the node. This will be done by the following statements.

1. **ptr = (struct node *) malloc(sizeof(struct node *));**
2. **ptr → data = item**

ALGORITHM

- **Step 1:** IF PTR = NULL

  Write OVERFLOW
  Go to Step 7
  [END OF IF]

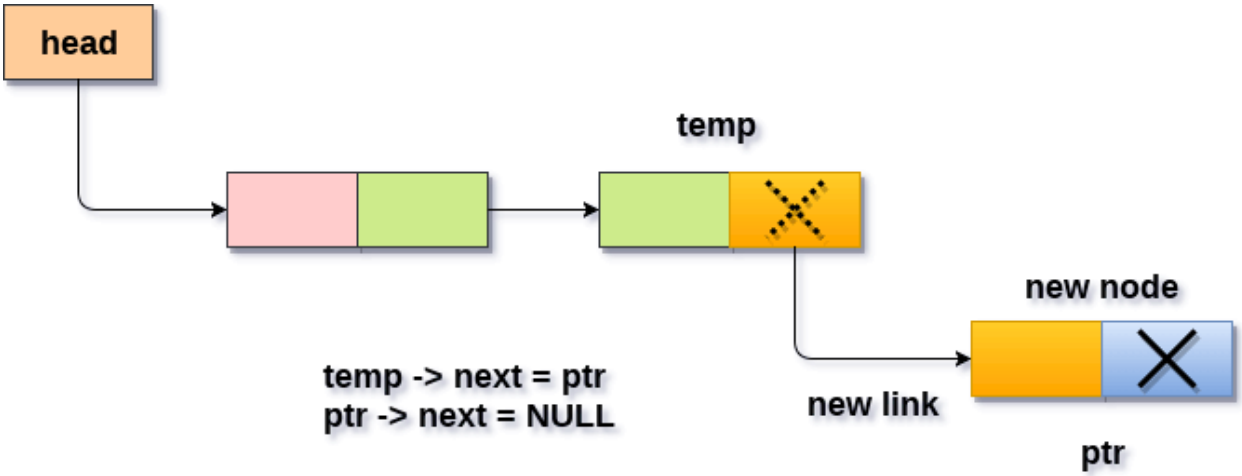
- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR → NEXT
- **Step 4:** SET NEW_NODE → DATA = VAL
- **Step 5:** SET NEW_NODE → NEXT = HEAD
- **Step 6:** SET HEAD = NEW_NODE
- **Step 7:** EXIT

Program

```
1.  #include<stdio.h>
```

```
2.  #include<stdlib.h>
3.  void beginsert(int);
4.  struct node
5.  {
6.      int data;
7.      struct node *next;
8.  };
9.  struct node *head;
10. void main ()
11. {
12.     int choice,item;
13.     do
14.     {
15.         printf("\nEnter the item which you want to insert?\n");
16.         scanf("%d",&item);
17.         beginsert(item);
18.         printf("\nPress 0 to insert more ?\n");
19.         scanf("%d",&choice);
20.     }while(choice == 0);
21. }
22. void beginsert(int item)
23.     {
24.         struct node *ptr = (struct node *)malloc(sizeof(struct node *));
25.         if(ptr == NULL)
26.         {
27.             printf("\nOVERFLOW\n");
28.         }
29.         else
30.         {
31.             ptr->data = item;
32.             ptr->next = head;
33.             head = ptr;
34.             printf("\nNode inserted\n");
35.         }
36.
37.     }
```

Insertion in singly linked list at the end



Inserting node at the last into a non-empty list

In order to insert a node at the last, there are two following scenarios which need to be mentioned.

1. The node is being added to an empty list
2. The node is being added to the end of the linked list

In the first case,

- The condition (head == NULL) gets satisfied. Hence, we just need to allocate the space for the node by using malloc statement in C. Data and the link part of the node are set up by using the following statements.

1. ptr->data = item;
2. ptr -> next = NULL;

- Since, **ptr** is the only node that will be inserted in the list hence, we need to make this node pointed by the head pointer of the list. This will be done by using the following Statements.

1. Head = ptr

In the second case,

- The condition **Head = NULL** would fail, since Head is not null. Now, we need to declare a

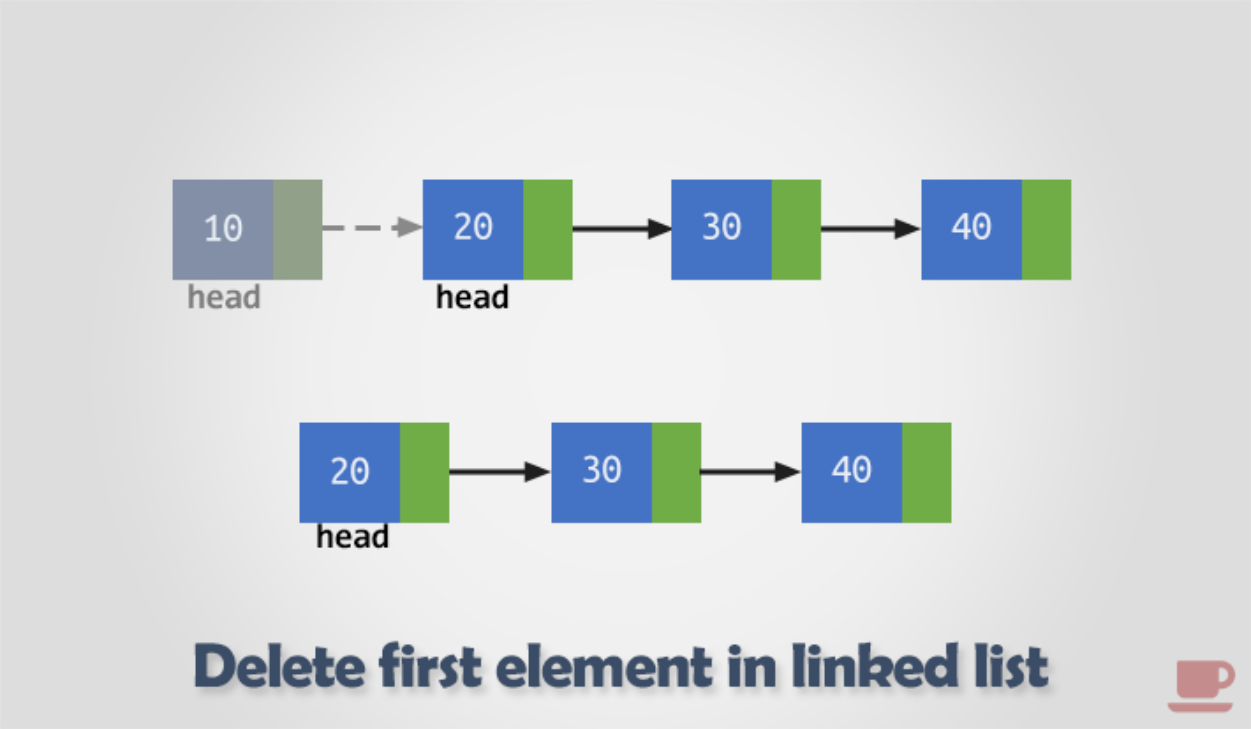
temporary pointer temp in order to traverse through the list. **temp** is made to point the first node of the list.

1. Temp = head
    - Then, traverse through the entire linked list using the statements:

Algorithm

- **Step 1:** IF PTR = NULL Write OVERFLOW
  Go to Step 1
  [END OF IF]
- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR - > NEXT
- **Step 4:** SET NEW_NODE - > DATA = VAL
- **Step 5:** SET NEW_NODE - > NEXT = NULL
- **Step 6:** SET PTR = HEAD
- **Step 7:** Repeat Step 8 while PTR - > NEXT != NULL
- **Step 8:** SET PTR = PTR - > NEXT
  [END OF LOOP]
- **Step 9:** SET PTR - > NEXT = NEW_NODE
- **Step 10:** EXIT

C program to delete first node of Singly Linked List

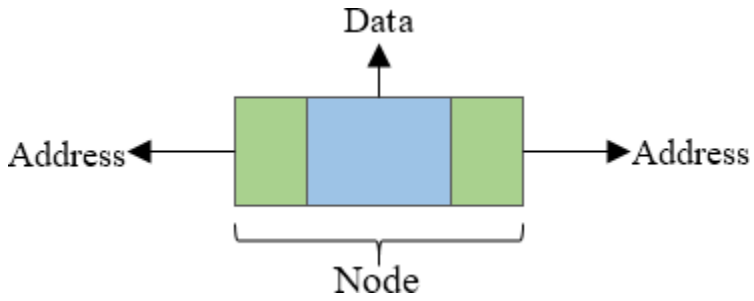


Algorithm to delete first node from Singly Linked List

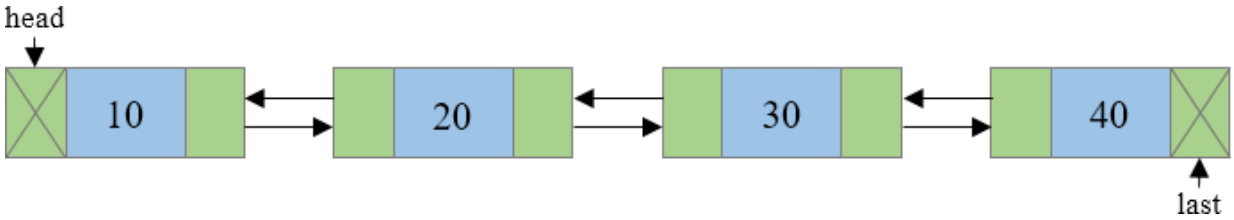
```
Algorithm to delete first node of Singly Linked List

%%Input:  head of the linked list

Begin:

    If (head != NULL) then
        toDelete ← head head

        ← head.next unalloc

        (toDelete)

    End if
```

Data Structure : Doubly Linked List

Doubly linked list is a collection of nodes linked together in a sequential way. Each node of the list contains two parts (as in singly linked list) **data part** and the **reference or address part**. The basic structure of node is shown in the below image



Doubly linked list is almost similar to singly linked list except it contains two address or reference fields, where one of the address field contains reference of **the next node and other contains reference of the previous node.** First and last node of a linked list contains a terminator generally a `NULL` value, that determines the start and end of the list. Doubly linked list is sometimes also referred as **bi-directional linked list** since it allows traversal of nodes in both direction. The basic structure of a doub



ly linked list is represented as:

Basic structure of a doubly linked list

```
struct node {
    int data;           // Data field
    struct node * prev; // Address of previous node
    struct node * next; // Address of next node
};
```

Advantages of Doubly linked list

- As like singly linked list it is the easiest data structures to implement.
- Allows traversal of nodes in both direction which is not possible in singly linked list.
- Deletion of nodes is easy when compared to singly linked list, as in singly linked list deletion requires a pointer to the node and previous node to be deleted. Which is not in case of doubly linked list we only need the pointer which is to be deleted.
- Reversing the list is simple and straightforward.
- Can allocate or de-allocate memory easily when required during its execution.
- It is one of most efficient data structure to implement when traversing in both direction is required.

Disadvantages of Doubly linked list

Not many but doubly linked list has few disadvantages also which can be listed below:

- It uses extra memory when compared to array and singly linked list.
- Since elements in memory are stored randomly, hence elements are accessed sequentially no direct access is allowed.

Applications/Uses of doubly linked list in real life

- Doubly linked list can be used in navigation systems where both front and back navigation is required.
- It is used by browsers to implement backward and forward navigation of visited web pages i.e. **back** and **forward** button.
- It is also used by various application to implement **Undo** and **Redo** functionality.
- It can also be used to represent deck of cards in games.
- It is also used to represent various states of a game.

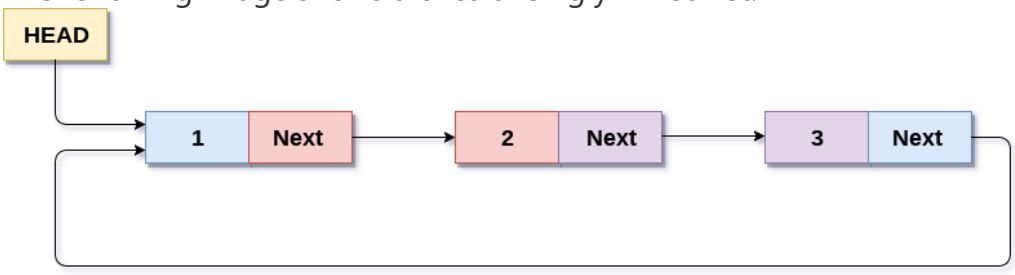
S. No.	Singly linked list	Doubly linked list
1	In case of singly linked lists, the complexityof insertion and deletion is O(n)	In case of doubly linked lists, the complexity of insertion and deletion is O(1)
2	The Singly linked list has two segments: dataand link.	The doubly linked list has three segments.First is data and second, third are the pointers.
3	It permits traversal components only in oneway.	It permits two way traversal.
4	We mostly prefer a singly linked list for theexecution of stacks.	We can use a doubly linked list to executebinary trees, heaps and stacks.
5	When we want to save memory and do notneed to perform searching, we prefer a singly linked list.	In case of better implementation, while searching, we prefer a doubly linked list.
6	A singly linked list consumes less memoryas compared to the doubly linked list.	The doubly linked list consumes more memory as compared to the singly linkedlist.

Circular Singly Linked List

In a circular Singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.

The following image shows a circular singly linked list.



Circular Singly Linked List

Circular linked list are mostly used in task maintenance in operating systems. There are many examples where circular linked list are being used in computer science including browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button

A circular linked list is a variation of a simple linked list. A circular linked list could be:

1. Singly circular linked list

2. Doubly circular linked list

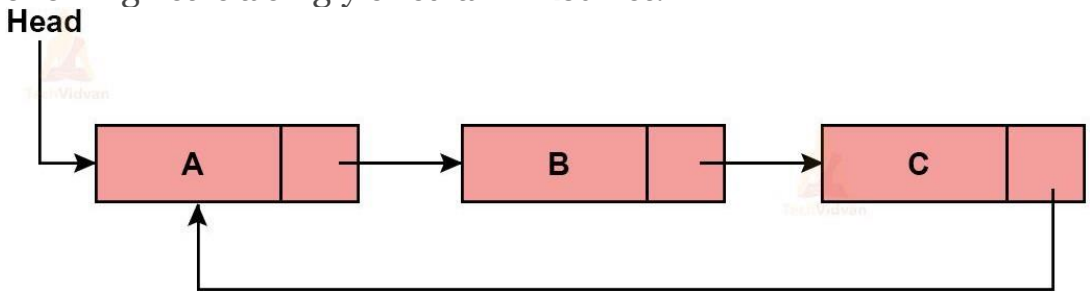
What is a Circular Linked list?

A circular list is a list that does not contain any pointer pointing to NULL. In a circular linked list, all the nodes are inter-connected in a cyclic manner. Both singly and doubly linked lists can be circular.

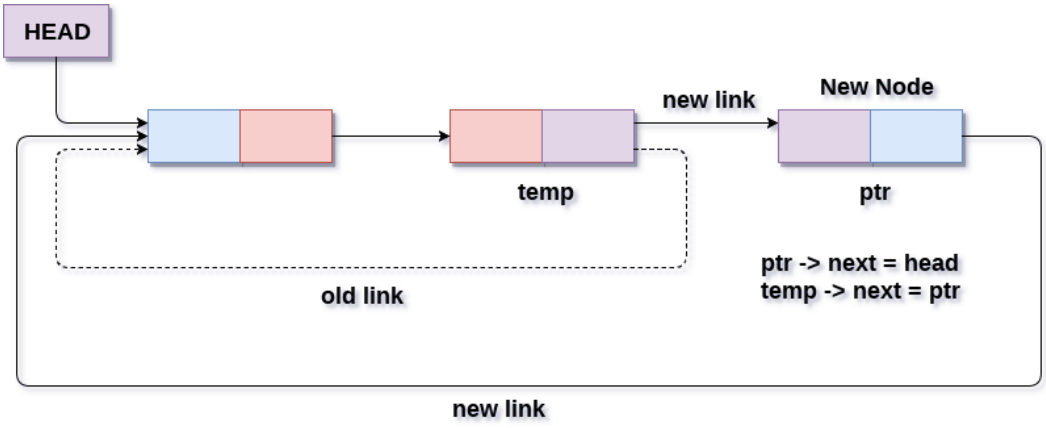
1. Singly Circular Linked List

There is only one pointer in a singly linked list that points to NULL, i.e. the pointer from the last node of the list. If we want to make the singly linked list circular, we need to make the 'Next' pointer of the last node point to the first node.

The following list is a singly circular linked list:



Insertion into circular singly linked list at the end



Insertion into circular singly linked list at end

Algorithm

- **Step 1:** IF PTR = NULL

Write OVERFLOW
Go to Step 1
[END OF IF]

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET NEW_NODE -> NEXT = HEAD
- **Step 6:** SET TEMP = HEAD
- **Step 7:** Repeat Step 8 while TEMP -> NEXT != HEAD
- **Step 8:** SET TEMP = TEMP -> NEXT

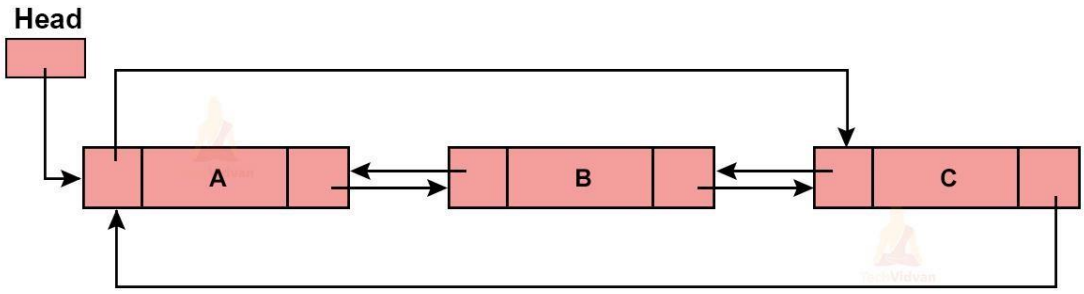
  [END OF LOOP]

- **Step 9:** SET TEMP -> NEXT = NEW_NODE
- **Step 10:** EXIT

. *Doubly Circular Linked List*

A doubly linked list points to not only the next node but to the previous node as well. A circular doubly linked list contains 2 NULL pointers. The 'Next' of the last node points to the first node in a doubly-linked list. The 'Prev' of the first node points to the last node.

A doubly circular linked list looks as follows:



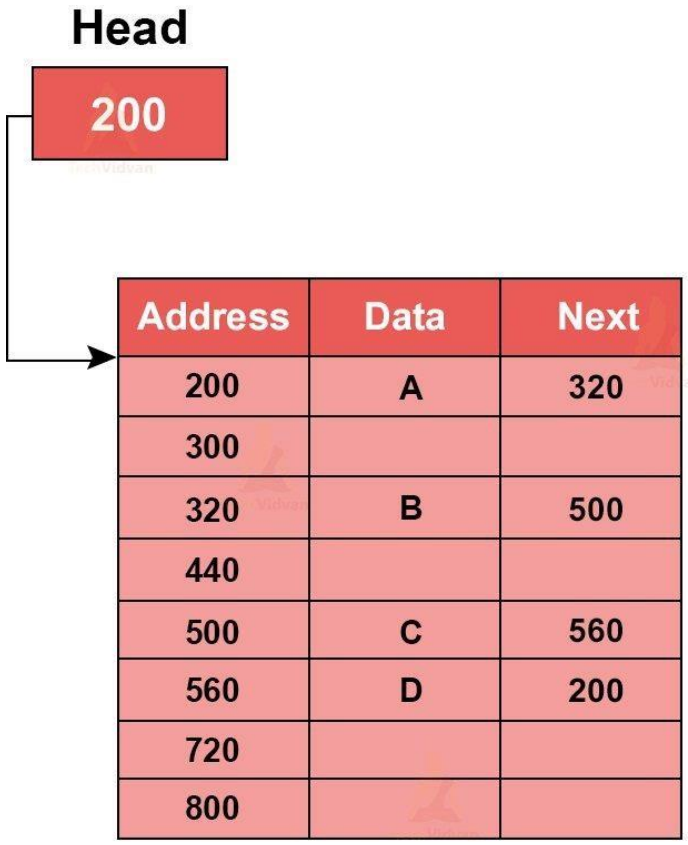
Memory Representation of Circular Linked List

Memory representation defines how a linked list is stored in the main memory.

1. Singly Circular Linked List

A singly circular list consists of only one addition pointer named ‘Next’. The ‘Next’ of each node, except the last node will contain the address of the upcoming node. The last node’s ‘Next’ will store the address of the first node.

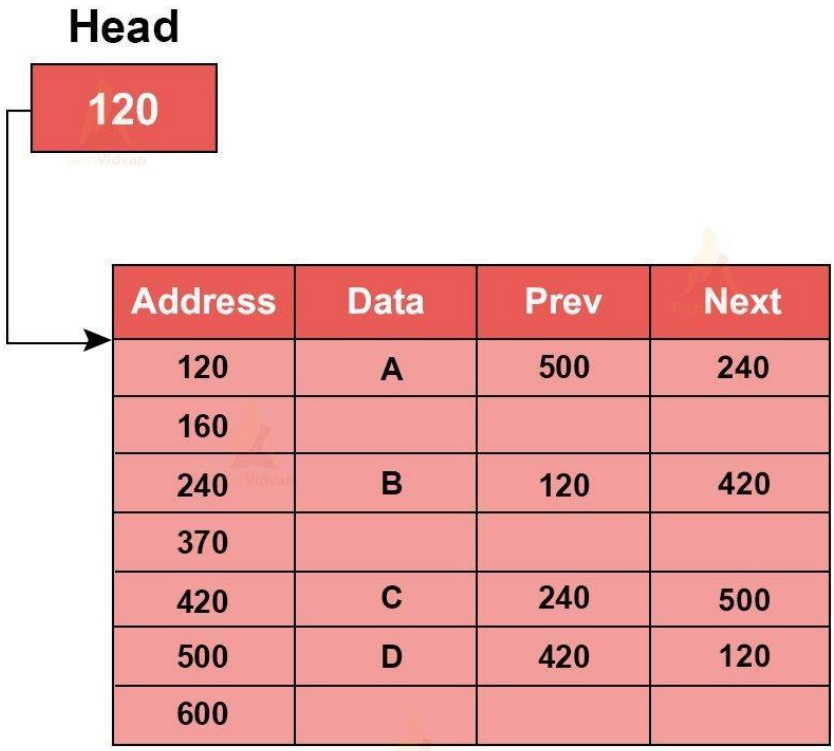
For a singly linked list, the memory will be represented as follows:



Doubly Circular Linked List

In a doubly-linked list, there are two additional pointers: 'Next' and 'Prev'. The 'Next' of each node links to the address of the upcoming node's 'Prev'. The 'Next' of the last node links to the first node. The 'prev' of each node has the address of the previous node. The 'Prev' of the first node links to the last node.

The memory representation diagram for the doubly linked list is:



Address	Data	Prev	Next
120	A	500	240
160			
240	B	120	420
370			
420	C	240	500
500	D	420	120
600			

Applications of Circular Linked List

A circular linked list has some real-life applications such as:

1. **In multi-player games**, all the players are kept/arranged in a circular linked list form. This gives a fixed turn number to every player.

2. Circular linked lists are implied in our computer's operating system for scheduling various processes. In many cases, a lot of processes are aligned and they need to use the same resource. In such cases, they are put into a circular linked list and **given a fixed time slot for execution**.

3. We can also implement a circular queue using a circular linked list. This will **help to reduce the number of pointers from 2 to 1 because a circular linked list will require only one pointer.**

Basic operations on a Circular Linked List:

Traversal Operation:

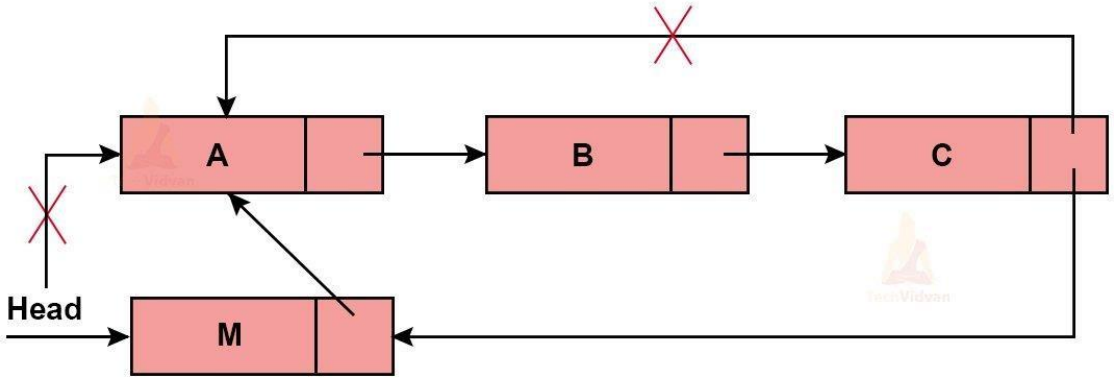
Display Operation:

Insertion Operation:

Insertion at the beginning:

Since a circular list could be both a singly circular list and a doubly circular linked list, let us check the insertion operation for both of them.

i. Singly circular linked list:
We will insert a node at the front of the list in the following way:



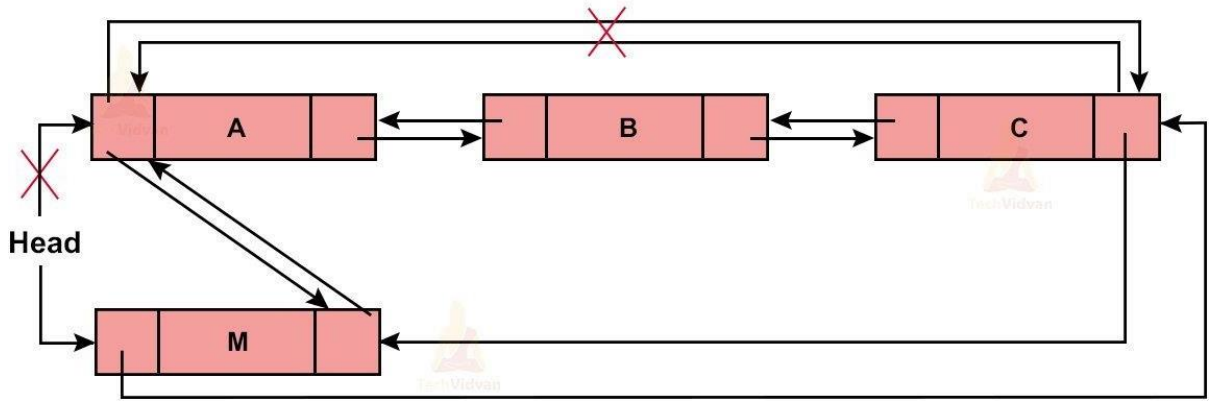
ii. Doubly circular linked list:

Insertion operation will lead to the shifting of various pointers in the list.

2. We insert a node at the beginning such that the next pointer points to the node which was at first before.

3. Let us take an example:
Initially, let the list be A→B→C→D.

Let us try inerting M into the list as shown:



Deletion in doubly linked list at the end

Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.

In order to delete the last node of the list, we need to follow the following steps.

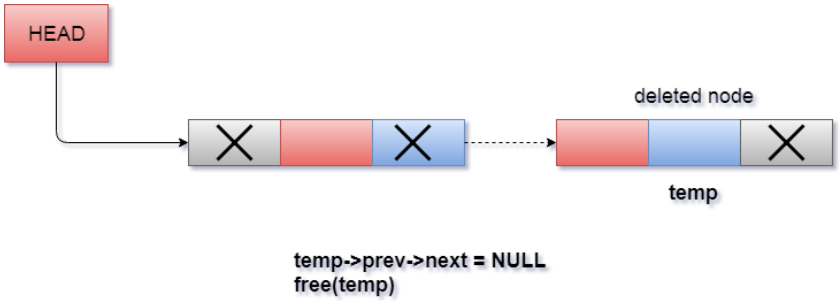
- If the list is already empty then the condition head == NULL will become true and therefore the operation can not be carried on.
- If there is only one node in the list then the condition head → next == NULL become true. In this case, we just need to assign the head of the list to NULL and free head in order to completely delete the list.
- Otherwise, just traverse the list to reach the last node of the list. This will be done by using the following statements.
- ptr = head;
- if(ptr->next != NULL)
- {
- ptr = ptr -> next;
- }
- The ptr would point to the last node of the ist at the end of the for loop. Just make the next pointer of the previous node of **ptr** to **NULL**.

1. ptr → prev → next = NULL

   free the pointer as this the node which is to be deleted.

1. free(ptr)

ALGORITHM



Deletion in doubly linked list at the end

- **Step 1:** IF HEAD = NULL

  Write UNDERFLOW
  Go to Step 7
  [END OF IF]

- **Step 2:** SET TEMP = HEAD
- **Step 3:** REPEAT STEP 4 WHILE TEMP->NEXT != NULL
- **Step 4:** SET TEMP = TEMP->NEXT

  [END OF LOOP]

- **Step 5:** SET TEMP ->PREV-> NEXT = NULL
- **Step 6:** FREE TEMP
- **Step 7:** EXIT

Advantages of a Circular Linked List

1. Since the list is circular, any node could be the starting point while traversal or some other operations.

2. We can traverse to any node while standing at a particular node. We didn't **have this privilege in a linear list as we could not go back to the previous nodes.**

3. The circular linked list helps in the **implementation of the queue.** If we implement the queue using a circular linked list, we need not maintain two pointers for front and rear. Rather we can fulfill our purpose using a single pointer.

4. Circular doubly linked lists help in implementing advanced data structures **like a Fibonacci heap.**

5. Circular linked lists are used **in various CPU scheduling algorithms such as the Round Robin scheduling algorithm**. In this algorithm, **processes are linked up in a circular manner with fixed time slots.**

Disadvantages of a Circular Linked List

1. It is very difficult to reverse a circular linked list.

2. Finding the end of the list is very hard.

3. If not implemented rightly, there is a high possibility of an infinite loop.

Memory Allocation: Memory allocation is a process by which computer programs and services are assigned with physical or virtual memory space. The memory allocation is done either before or at the time of program execution. There are two types of memory allocations:

1. Compile-time or Static Memory Allocation
2. Run-time or Dynamic Memory Allocation

Static Memory Allocation: Static Memory is allocated for declared variables by the compiler. The address can be found using the *address of* operator and can be assigned to a pointer. The memory is allocated during compile time.
Dynamic Memory Allocation: Memory allocation done at the time of execution(run time) is known as **dynamic memory allocation**. Functions calloc() and malloc() support allocating dynamic memory. In the Dynamic allocation of memory space is allocated by using these functions when the value is returned by functions and assigned to pointer variables.

Difference between Static and Dynamic Memory Allocation

S.No	Static Memory Allocation	Dynamic Memory Allocation
1	When the allocation of memory performs at thecompile time, then it is known as static memory.	When the memory allocation is done at the execution or run time,then it is called dynamic memory allocation.
2	The memory is allocatedat the compile time.	The memory is allocated at theruntime.
3	In static memory allocation, while executing a program,	In dynamic memory allocation, while executing a program, the memory can be changed.
	the memory cannot bechanged.	

4	Static memory allocation is preferred inan array.	Dynamic memory allocation ispreferred in the linked list.
5	It saves running time asit is fast.	It is slower than static memoryallocation.
6	Static memory allocation allots memory from the stack.	Dynamic memory allocation allotsmemory from the heap.
7	Once the memory is allotted, it will remainfrom the beginning to end of the program.	Here, the memory can be allotedat any time in the program.
8	Static memory allocation is less efficient as compared toDynamic memory allocation.	Dynamic memory allocation is more efficient as compared to theStatic memory allocation.
9	This memory allocationis simple.	This memory allocation iscomplicated.

Example:

```
int i;
float j;
```

Example:

```
p = malloc(sizeof(int));
```

Array in Data Structure

In this article, we will discuss the array in data structure. Arrays are defined as the collection of similar types of data items stored at contiguous memory locations. It is one of the simplest data structures where each data element can be randomly accessed by using its index number.

In C programming, they are the derived data types that can store the primitive type of data such as int, char, double, float, etc. For example, if we want to store the marks of a student in 6 subjects, then we don't need to define a different variable for the marks in different subjects. Instead, we can define an array that can store the marks in each subject at the contiguous memory locations.

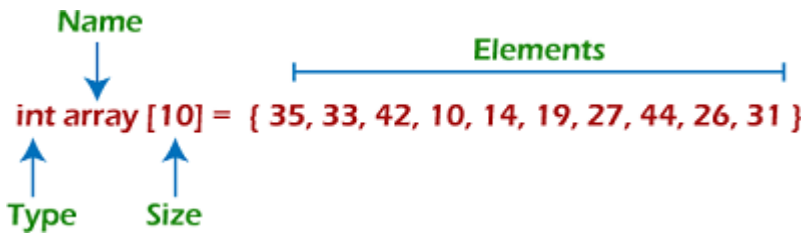
Properties of array

There are some of the properties of an array that are listed as follows -

- Each element in an array is of the same data type and carries the same size that is 4 bytes.
- Elements in the array are stored at contiguous memory locations from which the first element is stored at the smallest memory location.
- Elements of the array can be randomly accessed since we can calculate the address of each element of the array with the given base address and the size of the data element.

Representation of an array

We can represent an array in various ways in different programming languages. As an illustration, let's see the declaration of array in C language -



s per the above illustration, there are some of the following important points -

- Index starts with 0.
- The array's length is 10, which means we can store 10 elements.
- Each element in the array can be accessed via its index.

Why are arrays required?

Arrays are useful because -

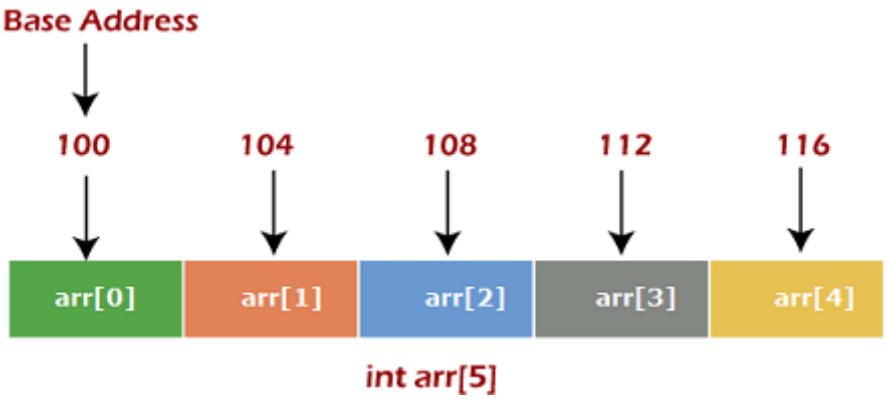
- Sorting and searching a value in an array is easier.
- Arrays are best to process multiple values quickly and easily.
- **Arrays are good for storing multiple values in a single variable -** In computer programming, most cases require storing a large number of data of a similar type. To store such an amount of data, we need to define a large number of variables. It would be very difficult to remember the names of all the variables while writing the programs. Instead of naming all the variables with a different name, it is better to define an array and store all the elements into it.

Memory allocation of an array

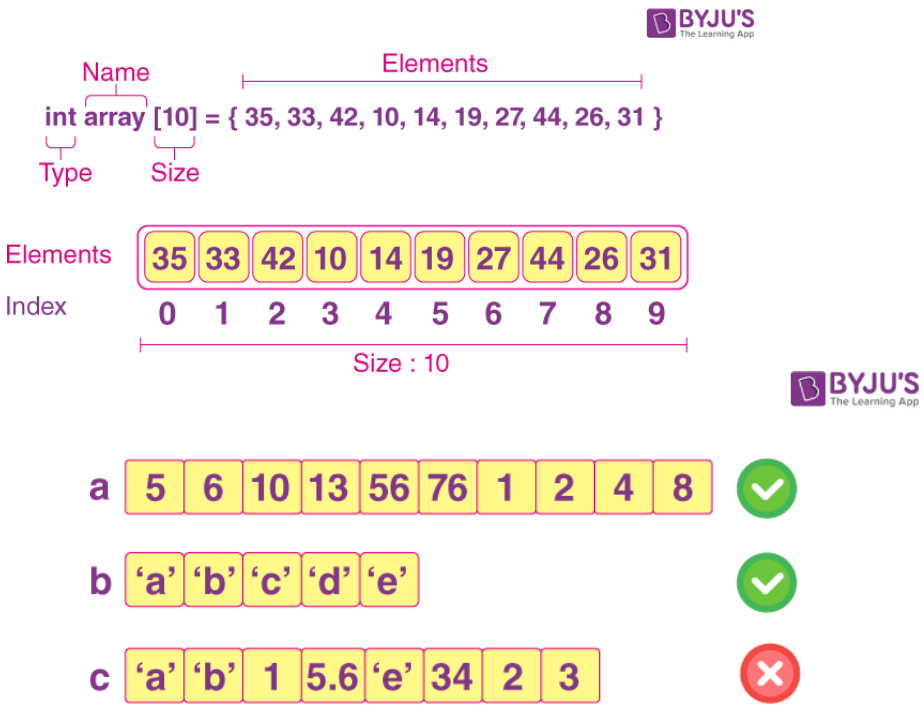
As stated above, all the data elements of an array are stored at contiguous locations in the main memory. The name of the array represents the base address or the address of the first element in the main memory. Each element of the array is represented by proper indexing.

We can define the indexing of an array in the below ways -

1. 0 (zero-based indexing): The first element of the array will be arr[0].
2. 1 (one-based indexing): The first element of the array will be arr[1].
3. n (n - based indexing): The first element of the array can reside at any random index number.



In the above image, we have shown the memory allocation of an array arr of size 5. The array follows a 0-based indexing approach. The base address of the array is 100 bytes. It is the address of arr[0]. Here, the size of the data type used is 4 bytes; therefore, each element will take 4 bytes in the memory.



- In the array a, we have stored all integral values (same type)
- In the array b, we have stored all char values (same type)
- In the array c, there is integral, float, char all types of values and this is not something an array can store so, option 3 is wrong because an array cannot store different types of values.

How to access an element from the array?

We required the information given below to access any random element from the array -

- Base Address of the array.
- Size of an element in bytes.
- Type of indexing, array follows.

The formula to calculate the address to access an array element -

1. Byte address of element A[i] = base address + size * (i - first index)
   Here, size represents the memory taken by the primitive data types. As an instance, **int** takes 2 bytes, **float** takes 4 bytes of memory space in C programming.

Basic operations

Now, let's discuss the basic operations supported in the array -

- Traversal - This operation is used to print the elements of the array.
- Insertion - It is used to add an element at a particular index.
- Deletion - It is used to delete an element from a particular index.
- Search - It is used to search an element using the given index or by the value.

- Update - It updates an element at a particular index

Types of arrays?

There are two types of array:

- Two-dimensional array.
- Multi-dimensional array

How to represent two-dimensional arrays?

Syntax: DataType ArrayName[row_size][column_size];
For Example int arr[5][5];

ypes of Arrays:

There are two types of arrays:

- One-Dimensional Arrays
- Multi-Dimensional Arrays

One -Dimensional Arrays

A one-dimensional array is a kind of linear array. It involves single sub-scripting. The [] (brackets) is used for the subscript of the array and to declare and access the elements from the array.

Syntax: DataType ArrayName [size];

For example: int a[10];

Multi-Dimensional Arrays

In multi-dimensional arrays, we have two categories:

- Two-Dimensional Arrays
- Three-Dimensional Arrays

1.

1. Two-Dimensional Arrays

An array involving two subscripts [] [] is known as a two-dimensional array. They are also known as the array of the array. Two-dimensional arrays are divided into rows and columns and are able to handle the data of the table.

Syntax: DataType ArrayName[row_size][column_size];

For Example: int arr[5][5];

2. Three-Dimensional Arrays

When we require to create two or more tables of the elements to declare the array elements, then in such a situation we use three-dimensional arrays.

Syntax: DataType ArrayName[size1][size2][size3];

For Example: int a[5][5][5];

Advantages of Array

- Array provides the single name for the group of variables of the same type. Therefore, it is easy to remember the name of all the elements of an array.
- Traversing an array is a very simple process; we just need to increment the base address of the array in order to visit each element one by one.
- Any element in the array can be directly accessed by using the index.

Disadvantages of Array

- Array is homogenous. It means that the elements with similar data type can be stored in it.
- In array, there is static memory allocation that is size of an array cannot be altered.
- There will be wastage of memory if we store less number of elements than the declared size.

Linear Search

assume that item is in an array in random order and we have to find an item. Then the only way to search for a target item is, to begin with, the first position and compare it to the target. If the item is at the same, we will return the position of the current item. Otherwise, we will move to the next position. If we arrive at the last position of an array and still can not find the target, we return -1. This is called the Linear search or Sequential search.

BINARY SEARCH

In a binary search, however, cut down your search to half as soon as you find the middle of a sorted list. The middle element is looked at to check if it is greater than or less than the value to be searched. Accordingly, a search is done to either half of the given list

Linear Search	Binary Search
In linear search input data need not to be in sorted.	In binary search input data need to be in sorted order.

It is also called sequential search.	It is also called half-interval search.
The time complexity of linear search **O(n)**.	The time complexity of binary search **O(log n)**.
Multidimensional array can be used.	Only single dimensional array is used.
Linear search performs equality comparisons	Binary search performs ordering comparisons
It is less complex.	It is more complex.
It is very slow process.	It is very fast process.