Data Structures

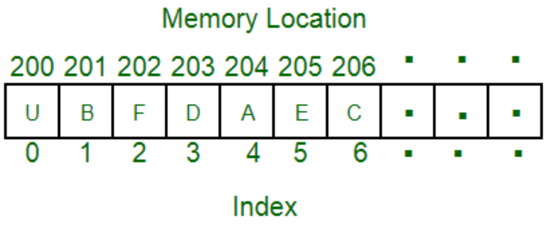
A data structure is a technique of storing and organizing the data in such a way that the data can be utilized in an efficient manner.
In order to make sure that our algorithm runs effectively we have to make sure that we don't just dump anything in the memory because it will be difficult to reach the needed data.
It is a lot more easier and efficient to reach data when memory or storage is organized and it even requires less time.
In order to make structures organized some kind of arrangement needs to be done for example making different index for different things.
In computers we have memory addresses which tells us where a particular files lies. For computer to pull out something it needs to read the address and go to that place to retrieve the data.
When the memory and storage is organized running algorithm efficiently is more easy and requires even less time
Pointers are used to point/indicate/address the file location.

Linear Data Structures:

A ***Linear data structure*** have data elements arranged in sequential manner or linearly and each member element is connected to its previous and next element. In linear data structure, single level is involved. This connection helps to traverse a linear data structure in a single level and in single run. Such data structures are easy to implement as computer memory is also sequential. Examples of linear data structures are List, Queue, Stack, Array etc.

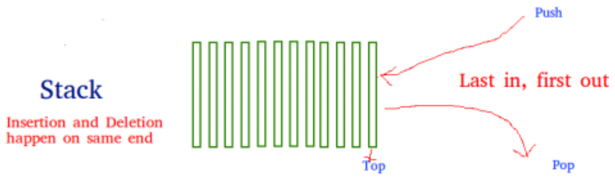
Array:

An array is a collection of items stored at contiguous memory locations.
An array is used to store a collection of data. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).
The lowest address corresponds to the first element and the highest address to the last element.
An array saves a lot of memory and reduces the length of the code.



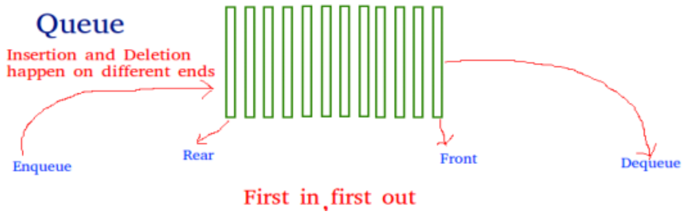
Stack:

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).
The example of stack is a website when you use the back arrow button
Stack is like piling up your books on the table
The thing you keep at the last has the pointer at the top and the thing you keep first has the pointer at the last.
The index follows from top to bottom and the thing you keep in last minute will be at the top.
The addition of data element in a stack is known as a push operation, and the deletion of data element form the list is known as pop operation.



Queue:

A Queue is a linear structure which follows a particular order in which the operations are performed.
The order is First In First Out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.
The difference between stack and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.
One that comes first is processed first



Stack and queue are like array but technically different from array.

Linked list:

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers.

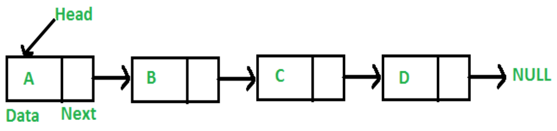
It is a collection of nodes that are made up of two parts, i.e., data element and reference to the next node in the sequence.

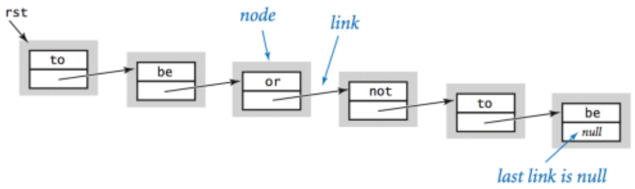
Sequence of elements where each element is linked to the next element Address of next linked data

5	---

23	----

All the elements of linked list do not lie together, it can lie anywhere in the memory.





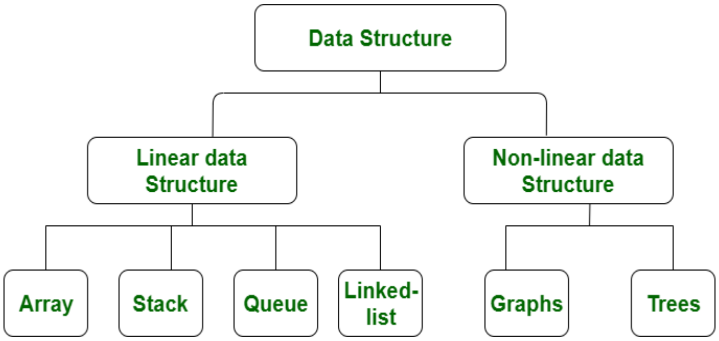
- The size of the link list needs not be declared at the start, when the program runs, run time memory allocation takes place, its size can both grow and link.
- However for an array the size needs to be declared first hence the space is declared first and reserved
- Array elements can be directly accessed by directly addressing the index number for element in the array. Array can be searched quickly
- Linked list is always sequentially addressed. You can just pick up data randomly from the middle.
- For a linked list complexity and time required to search the needed data increases as the linked list grows in size.
- Linked list will require more memory since in it the data and pointer representing the data both are being saved.
- Memory storage of linked list is more efficient because the memory expands as per its own requirements but in case of array size in memory
- needs to

  be declared at the start we declare an array. In this way if our requirement isn't much and we are not able to store anything there, it is then the waste of memory.

Non Linear Data Structures:

Data structures where data elements are not arranged sequentially or linearly are called *non-linear data structures*. In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only. Non-linear data structures are not easy to implement in comparison to linear data structure.

It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are trees and graphs.

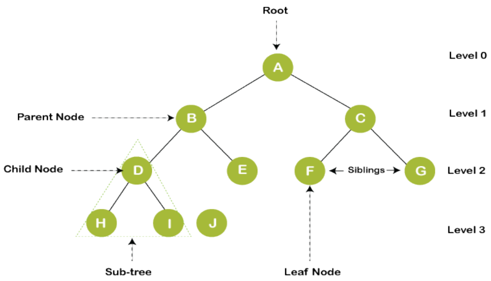


Difference between Linear And Nonlinear

	LINEAR DATA STRUCTURE	NON-LINEAR DATA STRUCTURE
1.	In a linear data structure, data elements are arranged in a linear order where each and every elements are attached to its previous and next adjacent.	In a non-linear data structure, data elements are attached in hierarchically manner.
2.	In linear data structure, single level is involved.	Whereas in non-linear data structure, multiple levels are involved.
3.	Its implementation is easy in comparison to non-linear data structure.	While its implementation is complex in comparison to linear data structure.
4.	In linear data structure, data elements can be traversed in a single run only.	While in non-linear data structure, data elements can't be traversed in a single run only.
5.	In a linear data structure, memory is not utilized in an efficient way.	While in a non-linear data structure, memory is utilized in an efficient way.
6.	Its examples are: array, stack, queue, linked list, etc.	While its examples are: trees and graphs.
7.	Applications of linear data structures are mainly in application software development.	Applications of non-linear data structures are in Artificial Intelligence and image processing.

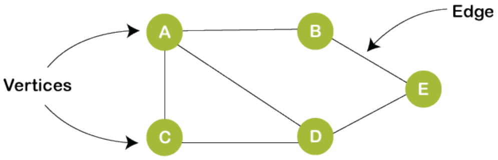
Tree:

It is a non-linear data structure that consists of various linked nodes. It has a hierarchical tree structure that forms a parent-child relationship.



Graph:

A graph is a non-linear data structure that has a finite number of vertices and edges, and these edges are used to connect the vertices. The vertices are used to store the data elements, while the edges represent the relationship between the vertices. A graph is used in various real-world problems like telephone networks, circuit networks, social networks like LinkedIn, Facebook. In the case of Facebook, a single user can be considered as a node, and the connection of a user with others is known as edges.



Differences between Array and Linked list

We cannot say which data structure is better, i.e., array or linked list. There can be a possibility that one data structure is better for one kind of requirement, while the other data structure is better for another kind of requirement. There are various factors like what are the frequent operations performed on the data structure or the size of the data, and other factors also on which basis the data structure is selected.

1. Cost of accessing an element

In case of an array, irrespective of the size of an array, an array takes a constant time for accessing an element. In an array, the elements are stored in a contiguous manner, so if we know the base address of the element, then we can easily get the address of any element in an array. We need to perform a simple calculation to obtain the address of any element in an array. So, accessing the element in an array is **O(1)** in terms of time complexity.

In the linked list, the elements are not stored in a contiguous manner. It consists of multiple blocks, and each block is represented as a node. Each node has two fields, i.e., one is for the data field, and another one stores the address of the next node. To find any node in the linked list, we first need to determine the first node known as the head node. If we have to find the second node in the list, then we need to traverse from the first node, and in the worst case, to find the last node, we will be traversing all the nodes. The average case for accessing the element is O(n). We conclude that the cost of accessing an element in array is less than the linked list. Therefore, if we have any requirement for accessing the elements, then array is a better choice.

2.Cost of inserting an element

There can be three scenarios in the insertion:

- ***Inserting the element at the beginning:***

To insert the new element at the beginning, we first need to shift the element towards the right to create a space in the first position. So, the time complexity will be proportional to the size of the list. **If n is the size of the array, the time complexity would be O(n).**

In the case of a linked list, to insert an element at the starting of the linked list, we will create a new node, and the address of the first node is added to the new node. In this way, the new node becomes the first node. So, the time complexity is not proportional to the size of the list. The time complexity would be constant, i.e., O(1).

- ***Inserting an element at the end:***

If the array is not full, then we can directly add the new element through the index. In this case, the time complexity would be constant, i.e., O(1). If the array is full, we first need to copy the array into another array and add a new element. In this case, the time complexity would be O(n).

To insert an element at the end of the linked list, we have to traverse the whole list. If the linked list consists of n elements, then the time complexity would be O(n).

- ***Inserting an element at the mid***

Suppose we want to insert the element at the i^{th} position of the array; we need to shift the n/2 elements towards the right. Therefore, the time complexity is proportional to the number of the elements. The time complexity would be O(n) for the average case.

In the case of linked list, we have to traverse to that position where we have to insert the new element. Even though, we do not have to perform any kind of shifting, but we have to traverse to n/2 position. The time taken is proportional to the n number of elements, and the time complexity for the average case would be O(n)

Ease of use:

The implementation of an array is easy as compared to the linked list. While creating a program using a linked list, the program is more prone to errors like segmentation fault or memory leak. So, lots of care need to be taken while creating a program in the linked list.

Dynamic in size:

The linked list is dynamic in size whereas the array is static. Here, static doesn't mean that we cannot decide the size at the run time, but we cannot change it once the size is decided.

3.Memory requirements

As the elements in an array store in one contiguous block of memory, so array is of fixed size. Suppose we have an array of size 7, and the array consists of 4 elements then the rest of the space is unused. The memory occupied by the 7 elements:

Memory space = 7*4 = 28 bytes

Where 7 is the number of elements in an array and 4 is the number of bytes of an integer type.

In case of linked list, there is no unused memory but the extra memory is occupied by the pointer variables. If the data is of integer type, then total memory occupied by one node is 8 bytes, i.e., 4 bytes for data and 4 bytes for pointer variable. If the linked list consists of 4 elements, then the memory space occupied by the linked list would be:

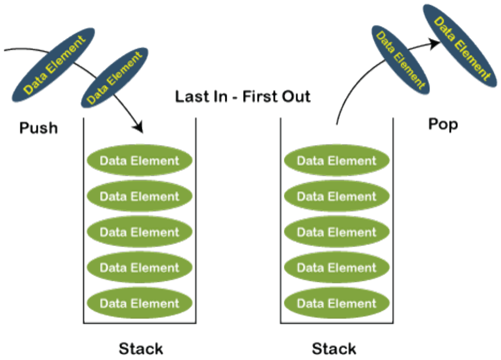
Memory space = 8*4 = 32 bytes

The linked list would be a better choice if the data part is larger in size. Suppose the data is of 16 bytes. The memory space occupied by the array would be 16*7=112 bytes while the linked list occupies 20*4=80, here we have specified 20 bytes as 16 bytes for the size of the data plus 4 bytes for the pointer variable. If we are choosing the larger size of data, then the linked list would consume a less memory; otherwise, it depends on the factors that we are adopting to determine the size.

Array	Linked list
An array is a collection of elements of a similar data type.	A linked list is a collection of objects known as a node where node consists of two parts, i.e., data and address.
Array elements store in a contiguous memory location.	Linked list elements can be stored anywhere in the memory or randomly stored.
Array works with a static memory. Here static memory means that the memory size is fixed and cannot be changed at the run time.	The Linked list works with dynamic memory. Here, dynamic memory means that the memory size can be changed at the run time according to our requirements.
Array elements are independent of each other.	Linked list elements are dependent on each other. As each node contains the address of the next node so to access the next node, we need to access its previous node.
Array takes more time while performing any operation like insertion, deletion, etc.	Linked list takes less time while performing any operation like insertion, deletion, etc.
Accessing any element in an array is faster as the element in an array can be directly accessed through the index.	Accessing an element in a linked list is slower as it starts traversing from the first element of the linked list.
In the case of an array, memory is allocated at compile-time.	In the case of a linked list, memory is allocated at run time.
Memory utilization is inefficient in the array. For example, if the size of the array is 6, and array consists of 3 elements only then the rest of the space will be unused.	Memory utilization is efficient in the case of a linked list as the memory can be allocated or deallocated at the run time according to our requirement.

What is a Stack?

A Stack is a linear data structure. In case of an array, random access is possible, i.e., any element of an array can be accessed at any time, whereas in a stack, the sequential access is only possible. It is a container that follows the insertion and deletion rule. It follows the principle **LIFO (Last In First Out)** in which the insertion and deletion take place from one side known as a **top**. In stack, we can insert the elements of a similar data type, i.e., the different data type elements cannot be inserted in the same stack. The two operations are performed in LIFO, i.e., **push** and **pop** operation.



The following are the operations that can be performed on the stack:

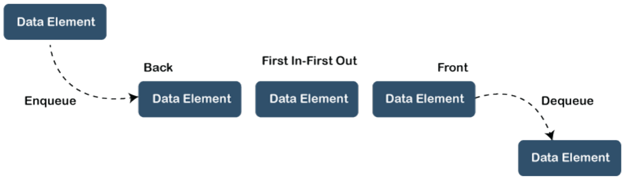
- **push(x):** It is an operation in which the elements are inserted at the top of the stack. In the **push** function, we need to pass an element which we want to insert in a stack.
- **pop():** It is an operation in which the elements are deleted from the top of the stack. In the **pop()** function, we do not have to pass any argument.
- **peek()/top():** This function returns the value of the topmost element available in the stack. Like pop(), it returns the value of the topmost element
- but does not remove that element from the stack.
- **isEmpty():** If the stack is empty, then this function will return a true value or else it will return a false value.
- **isFull():** If the stack is full, then this function will return a true value or else it will return a false value.

In stack, the **top** is a pointer which is used to keep track of the last inserted element. To implement the stack, we should know the size of the stack. We need to allocate the memory to get the size of the stack. There are two ways to implement the stack:

- **Static:** The static implementation of the stack can be done with the help of arrays.
- **Dynamic:** The dynamic implementation of the stack can be done with the help of a linked list.

What is the Queue?

A Queue is a linear data structure. It is an ordered list that follows the principle FIFO (First In -First Out). A Queue is a structure that follows some restrictions on insertion and deletion. In the case of Queue, insertion is performed from one end, and that end is known as a rear end. The deletion is performed from another end, and that end is known as a front end. In Queue, the technical words for insertion and deletion are **enqueue()** and **dequeue(),** respectively whereas, in the case of the stack, the technical words for insertion and deletion are push() and pop(), respectively. Its structure contains two pointers **front pointer** and **rear pointer**, where the front pointer is a pointer that points to the element that was first added in the queue and the rear pointer that points to the element inserted last in the queue.



Similarities between stack and queue.

There are two similarities between the stack and queue:

- ***Linear data structure***
  Both the stack and queue are the linear data structure, which means that the elements are stored sequentially and accessed in a single run.
- ***Flexible in size***
  Both the stack and queue are flexible in size, which means they can grow and shrink according to the requirements at the run -time.

Differences between stack and queue

Basis for comparison	Stack	Queue
Principle	It follows the principle LIFO (Last In- First Out), which implies that the element which is inserted last would be the first one to be deleted.	It follows the principle FIFO (First In -First Out), which implies that the element which is added first would be the first element to be removed from the list.
Structure	It has only one end from which both the insertion and deletion take place, and that end is known as a top.	It has two ends, i.e., front and rear end. The front end is used for the deletion while the rear end is used for the insertion.
Number of pointers used	It contains only one pointer known as a top pointer. The top pointer holds the address of the last inserted or the topmost element of the stack.	It contains two pointers front and rear pointer. The front pointer holds the address of the first element, whereas the rear pointer holds the address of the last element in a queue.
Operations performed	It performs two operations, push and pop. The push operation inserts the element in a list while the pop operation removes the element from the list.	It performs mainly two operations, enqueue and dequeue. The enqueue operation performs the insertion of the elements in a queue while the dequeue operation performs the deletion of the elements from the queue.
Examination of the empty condition	If top==-1, which means that the stack is empty.	If front== -1 or front = rear+1, which means that the queue is empty.
Examination of full condition	If top== max-1, this condition implies that the stack is full.	If rear==max-1, this condition implies that the stack is full.
Variants	It does not have any types.	It is of three types like priority queue, circular queue and double ended queue.
Implementation	It has a simpler implementation.	It has a comparatively complex implementation than a stack.
Visualization	A Stack is visualized as a vertical collection.	A Queue is visualized as a horizontal collection.