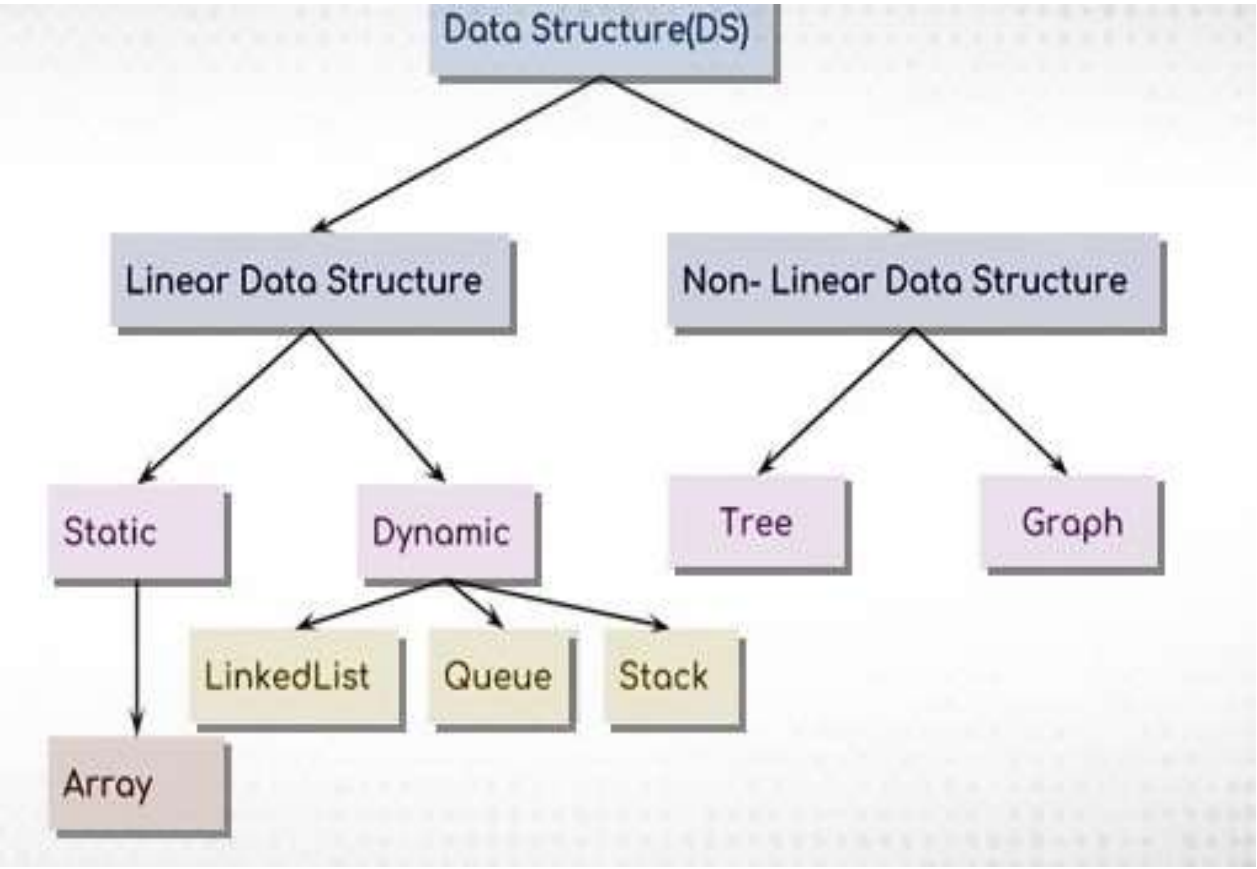
DS-Lecture-2

Summary of the first lecture:



When we think of data structures, there are generally four forms:

1. **Linear**: arrays, lists
2. **Tree**: binary, heaps, space partitioning etc.
3. **Hash**: distributed hash table, hash tree etc.
4. **Graphs**: decision, directed, acyclic etc.
5. ---

Differences **between array VS List**

The main difference between **these two data types is the operation you can perform on them. ...**

Also lists **are containers for elements having differing data types but** arrays **are used as containers for elements of the same data type**

Similarities between Lists and Arrays

- Both are used for storing data
- Both are mutable
- Both can be indexed and iterated through
- Both can be sliced

Differences

- The main difference between these two data types is the operation you can perform on them.

- Arrays are specially optimized for arithmetic computations so if you're going to perform similar operations you should consider using an array instead of a list.

- Also lists are containers for elements having differing data types but arrays are used as containers for elements of the same data type.

- The dividing an array by a certain number and doing the same for a list.

- When we try the same operation (example: division) on a list, we get a TypeError because builtin python lists do not support the `__div__` protocol.

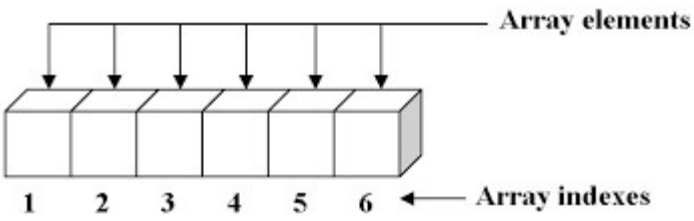
- It takes an extra step to perform this calculation on a list because then you'd have to loop over each item one after the other and save to another list.

- Making use of more robust array data types isn't also without it's cost implications.

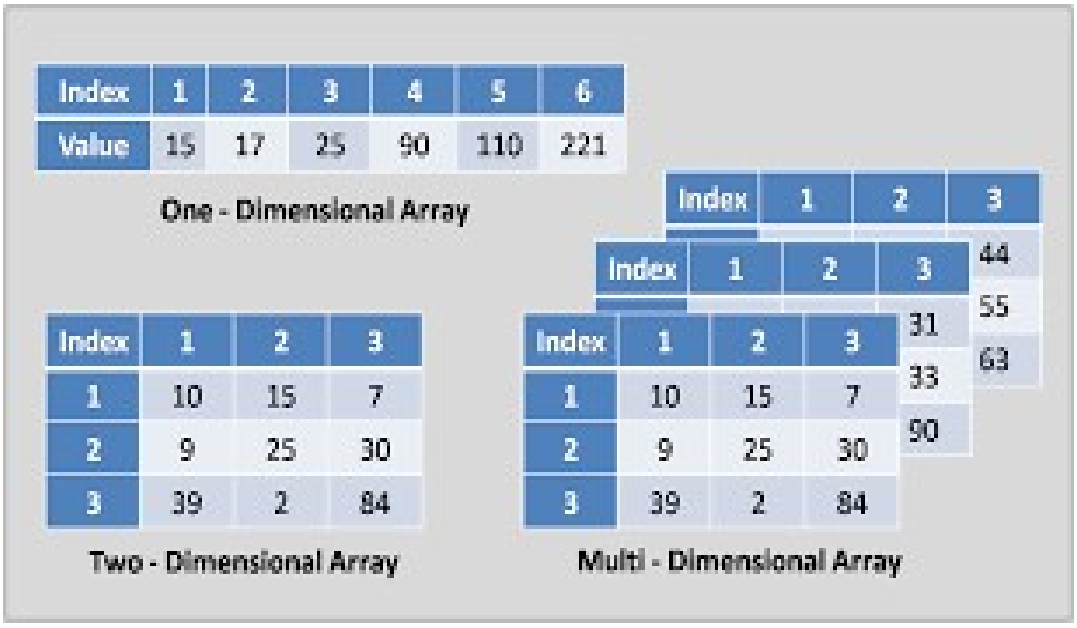
For example, to use `numpy` arrays you need to introduce a dependency in your project on the `numpy` library. You'd have to install the `numpy` package, import it and declare it while a list can be created on the fly.

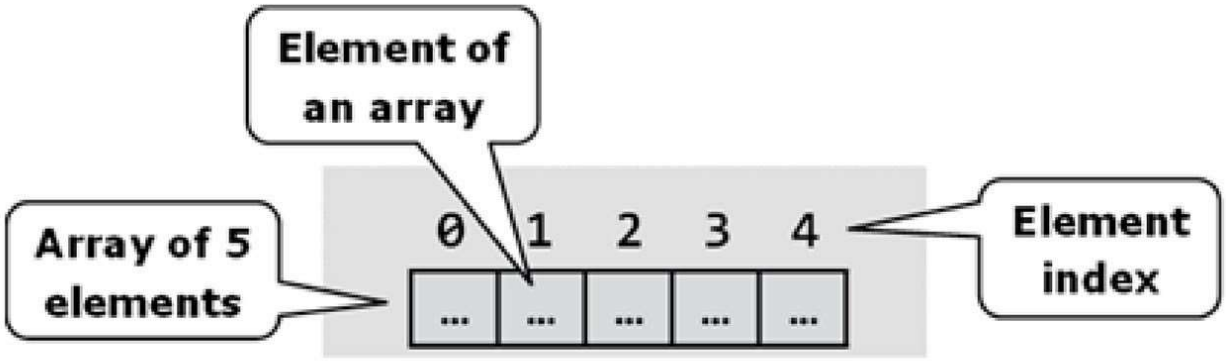
Array

An array is a finite group of data, which is allocated contiguous (i.e. sharing a common border) memory locations, and each element within the array is accessed via an index key (typically numerical, and zero based).



One-dimensional array with six elements





ARRAY IN DATA STRUCTURE

- The name assigned to an array is typically a pointer to the first item in the array.

- Meaning that given an array identifier of arr which was assigned the value ["a", "b", "c"], in order to access the "b" element you would use the index 1 to lookup the value: arr[1].

- Arrays are traditionally 'finite' in size, meaning you define their length/size (i.e. memory capacity) up front, but there is a concept known as 'dynamic arrays' (and of which you're likely more familiar with when dealing with certain high-level programmings languages) which supports the ***growing*** (or resizing) of an array to allow for more elements to be added to it.

- In order to resize an array you first need to allocate a new slot of memory (in order to copy the original array element values over to), and because this type of operation is quite 'expensive' (in terms of computation and performance) you need to be sure you increase the memory capacity just the right amount (typically double the original size) to allow for more elements to be added at a later time without causing the CPU to have to resize the array over and over again unnecessarily.

- One consideration that needs to be given is that you don't want the resized memory space to be ***too*** large, otherwise finding an appropriate slot of memory becomes more tricky.

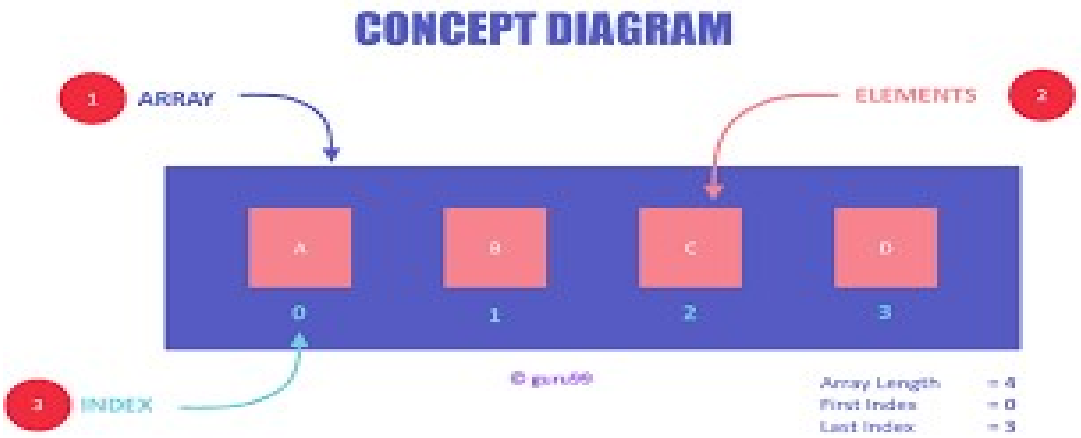
- When dealing with modifying arrays you also need to be careful because this requires significant overhead due to the way arrays are allocated memory slots.

- So if you imagine you have an array and you want to remove an element from the middle of the array, try to think about that in terms of memory allocation.

- An array needs its indexes to be contiguous, and so we have to re-allocate a new chunk of memory and copy over the elements that were placed ***around*** the deleted element.

- These types of operations, when done at scale, are the foundation behind why it's important to have an understanding of how data structures are implemented.

- The reason being, when you're writing an algorithm you will hopefully be able to recognize when you're about to do something (let's say modify an array many times within a loop construct) that could ultimately end up being quite a memory intensive set of operations.



Linked List

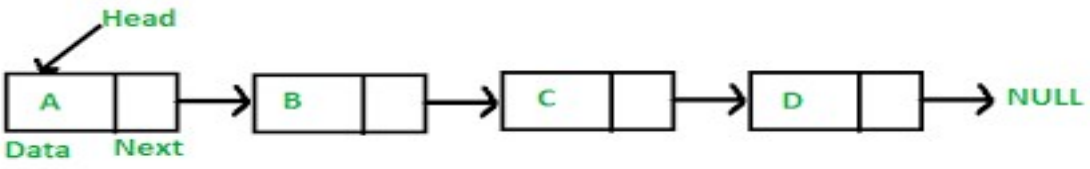
A linked list is different to an array in that the order of the elements within the list are not determined by a contiguous memory allocation. Instead the elements of the linked list can be sporadically (means irregularly) placed in memory due to its design. The each element of the list (also referred to as a 'node') consists of two parts:

1. the data
2. a pointer

Linked Lists

- List: A set of items organized sequentially
- Array can be used for it.Sequential
- organization is provided by the index.
- Index is used for accessing and manipulating elements.
- Array implementation has no. of drawbacks

1



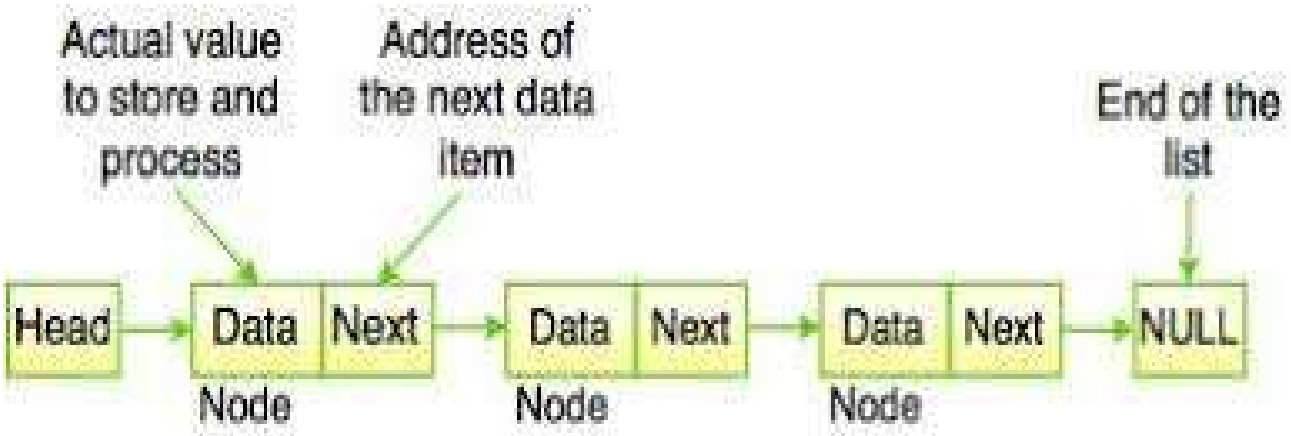
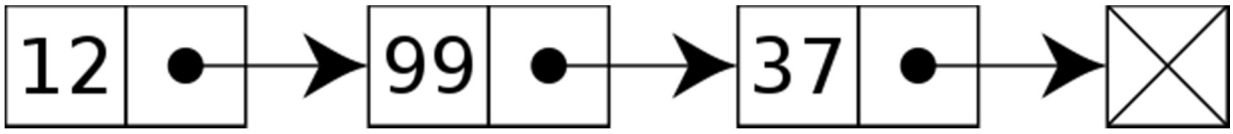


Fig. Linked List

The ***data is*** what you've assigned to that ***element/node***, whereas the ***pointer is a memory address reference to the next node in the list***.

Example as a numeric value:



Example as a string value:

Also unlike an array, there is no index access.

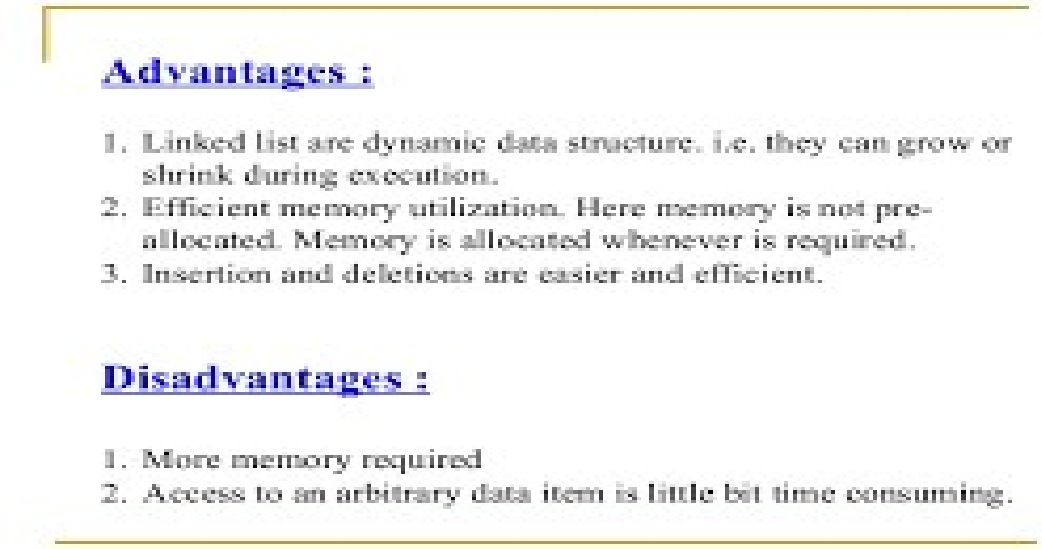
Remark- *So in order to locate a specific piece of data you'll need to traverse the entire list until you find the data you're looking for.*

- This is one of the key performance characteristics of a linked list, and is why (for most implementations of this data structure) you're not able to ***append*** data to the list (because if you think about the performance of such an operation it would require you to traverse the entire list to find the end/last node).

- Instead linked lists generally will only allow ***prepending*** to a list as it's much quicker.

- The newly added node will then have its pointer set to the original 'head' of the list.

Advantages and Disadvantages of Linked List



Advantages :

1. Linked list are dynamic data structure. i.e. they can grow or shrink during execution.
2. Efficient memory utilization. Here memory is not pre-allocated. Memory is allocated whenever is required.
3. Insertion and deletions are easier and efficient.

Disadvantages :

1. More memory required
2. Access to an arbitrary data item is little bit time consuming.

Difference between Array VS Linked List

ARRAY	LINKED LIST
Size of an array is fixed	Size of a list is not fixed
Memory is allocated from stack	Memory is allocated from heap
It is necessary to specify the number of elements during declaration (i.e., during compile time).	It is not necessary to specify the number of elements during declaration (i.e., memory is allocated during run time).
It occupies less memory than a linked list for the same number of elements.	It occupies more memory.
Inserting new elements at the front is potentially expensive because existing elements need to be shifted over to make room.	Inserting a new element at any position can be carried out easily.
Deleting an element from an array is not possible.	Deleting an element is possible.

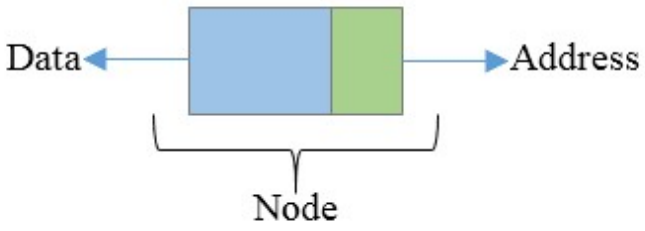
There is also a modified version of this data structure referred to as a 'doubly linked list' which is essentially the same concept but with the exception of a third attribute for each node: a pointer to the ***previous*** node (whereas a normal linked list would only have a pointer to the ***following*** node).

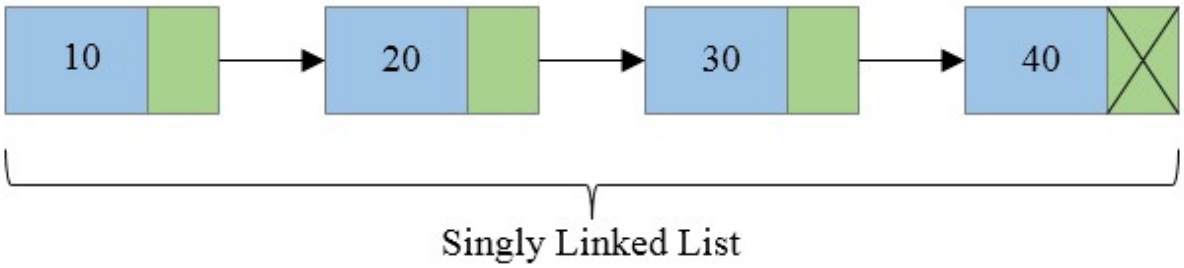
Types of Linked List

Linked lists

- Singly linked lists
- Doubly linked lists
- Circular lists
- Self-organizing lists
- `LinkedList` and `ArrayList`

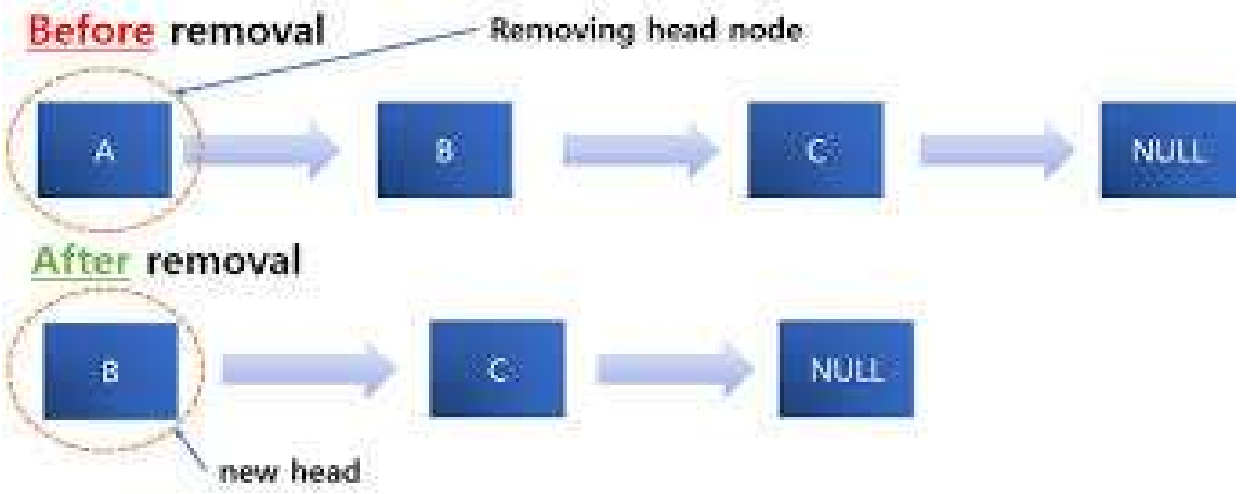
Singly linked List



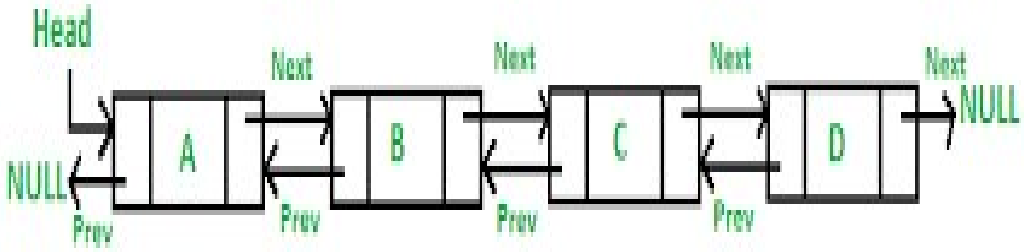


Operation on LL

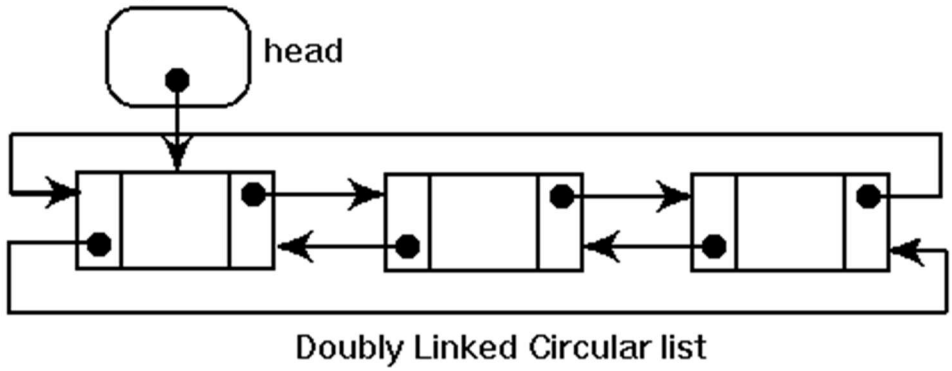
For example removal of elements:



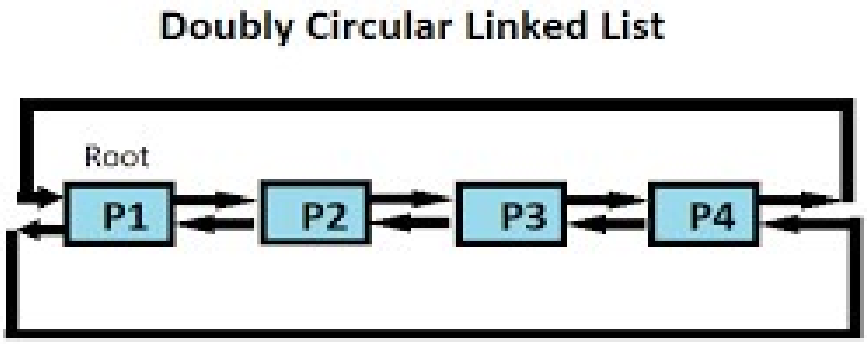
Doubly Linked List



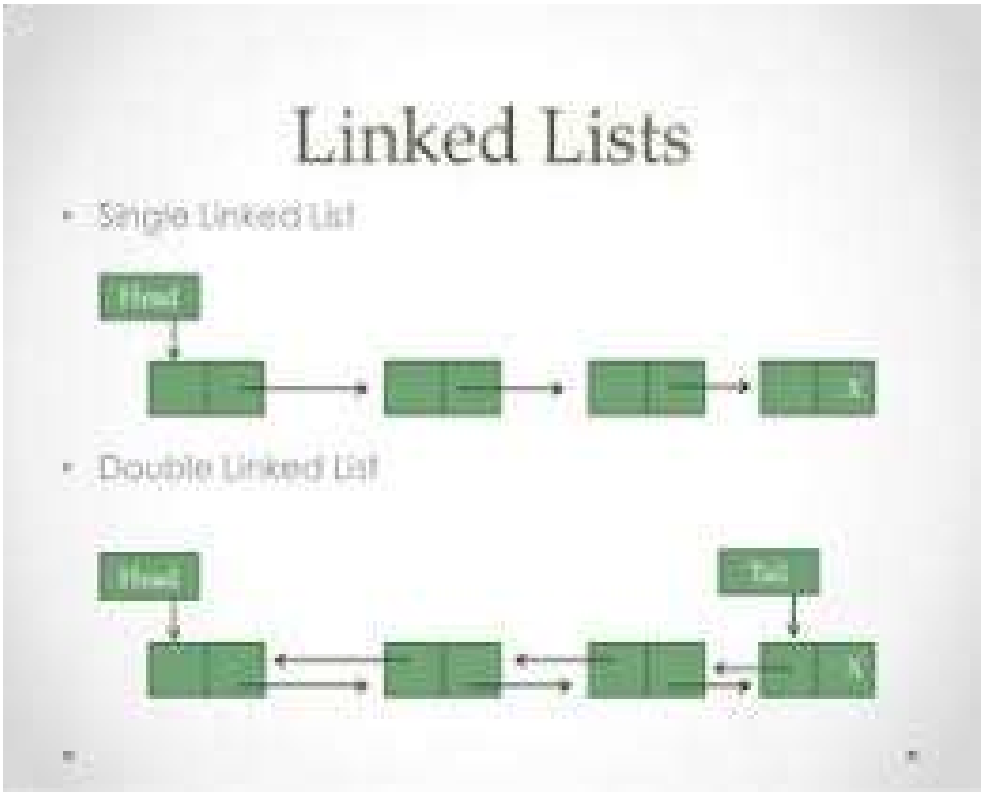
Circularly Linked List



Doubly Linked Circular list



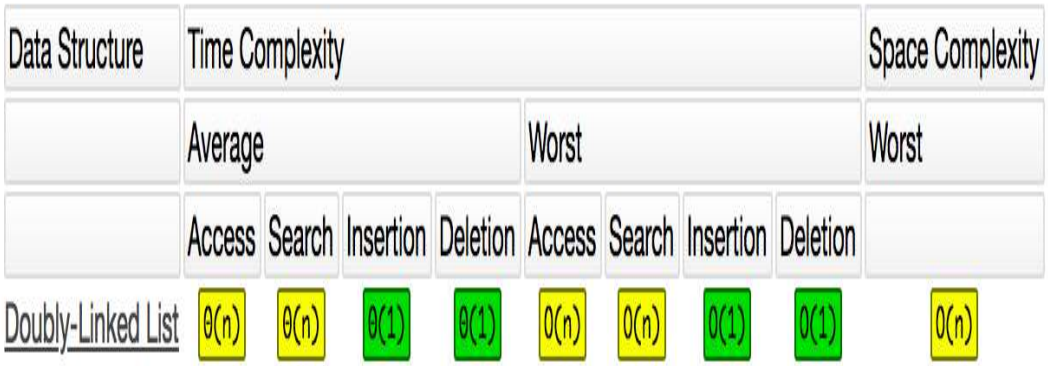
Comparison of SLL VS DLS



Note: again, performance considerations need to be given for the types of operations being made with a doubly linked list, such as the addition or removal of nodes in the list, because you now have not only the pointers to

the following node that need to be updated, but also the pointers back to a previous node that now also need to be updated.

Order of Complexity



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Doubly-Linked List	θ(n)	θ(n)	θ(1)	θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)

Singly Linked Lists and Arrays

Singly linked list	Array
Elements are stored in linear order, accessible with links.	Elements are stored in linear order, accessible with an index.
Do not have a fixed size.	Have a fixed size.
Cannot access the previous element directly.	Can access the previous element easily.
No binary search.	Binary search.