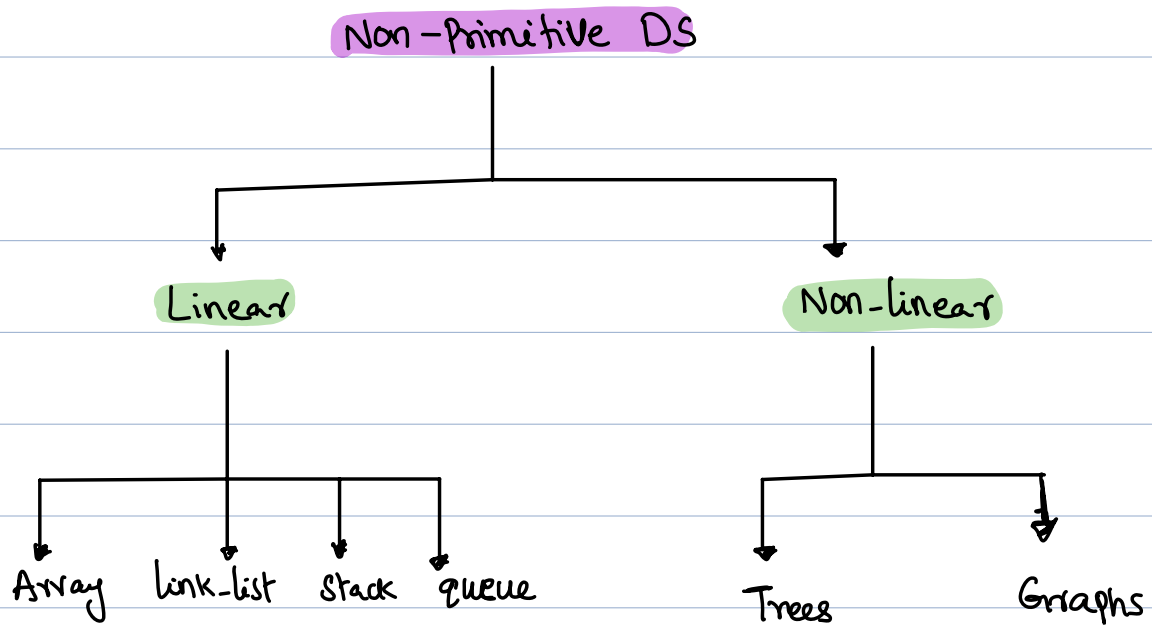
Datastructures are divided into categories:
Primitive and Non-Primitive.

Primitive data structures are basic data types built into a Programming language from which all other data types are constructed.

Examples: integers, float, char, Boolean etc

Non-Primitive data structures are more complex and they are created using Primitive data Structures.

Example: Arrays, link-list, Stacks etc.



- In linear data structure the elements are stored in a linear or sequential order.

Operations on Datastructures

Traversing: systematically visiting every element of datastructure

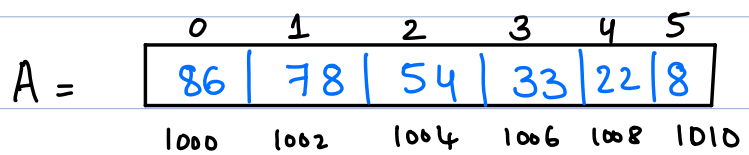
Searching: used to find the location of one or more data items that satisfy the given constraint

Inserting: used to add new data items to the existing list of data items.

Deleting: used to delete a particular data item from the existing list of data items.

Arrays

An array is a collection of similar data elements. These data elements have the same data type. The elements of the array are stored in consecutive memory locations and are referenced by an index.



Operations on Arrays

- **Traversing an array**

  Accessing each element of the array for a specific task.

```
1   Traverse(A, n)
2       i = 0
3       While i < n
4           Print A[i]
5           i ← i+1
```

Inserting an element in an Array

let A be an array of n integers.

task: Insert an element k at position P in A

Pseudo code

```
1   Insert (A, n, P, k)
2       i = n
3       while i >= P
4           A[i+1] = A[i]
5           i = i-1
6       n = n+1
7       A[P] = k
```

Running time = $\Theta(n)$

Deleting an element from an Array

let A be an array of n integers.

task: Delete the element present at position P in A

Pseudo code

```
Delete (A, n, P)
1       i = P
2       while i < n
3           A[i] = A[i+1]
4           i = i+1
5       n = n-1
```

Running time = $\Theta(n)$

Merging Two Arrays

Given two sorted arrays A and B, the task is to merge them to form a new sorted array C.

Example: If A = | 7 | 1 | 8 | 9 | and B = | 4 | 5 |

then C = | 1 | 4 | 5 | 7 | 8 | 9 |

let n_1 and n_2 denote the number elements of A and B respectively

Pseudo code

```
1   Merge (A, B, n1, n2)
2     n = n1 + n2
3     i = 0 , j = 0 , k = 0
4     A[n1] = ∞ ,   B[n2] = ∞
5     while k < n
6       if A[i] < B[j]
7         C[K] = A[i]
8         i = i + 1
9       else
10        C[K] = B[j]
11        j = j + 1
12      K = K + 1
13.   return C
```

Running time = $\Theta(n)$

[Details are skipped.
You need to proper analysis]

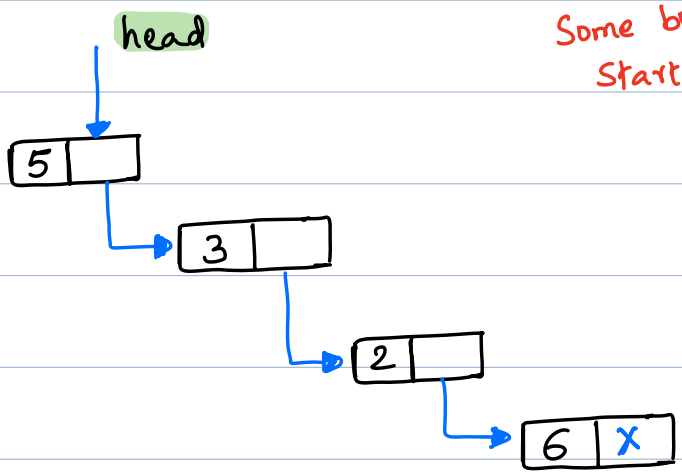
Linked Lists

- A link list, is a linear collection of data elements. These data elements are called nodes.

- Link lists are also used to implement other data structures such stacks and queues.

- Every node in linked list contains two parts: data and a link to the next node.
[Pointer]

Example



Some books write start as head

- Link lists contain a variable head that contains link of the first node.

Arrays Vs Linked Lists

- Unlike an array, a linked list does not store its nodes in consecutive memory locations.

- Nodes in a linked list can be accessed only in a sequential manner.

- We can add any number of elements in the list. This was not possible in array (static)

Operations on Linked Lists

- **Traversing an Linked Lists**

next will store the address of the next node in sequence.

```
Traverse (head)
    Ptr = head
    While Ptr != NULL
        Ptr = Ptr.next
```

Running time = $\Theta(n)$

- Searching for a value in a linked list

```
1   Search (head , k)
2       Ptr = head
3       While ptr != NULL
4           if K == Ptr.data
5               Return found
6           Ptr = Ptr.next
```

Inserting a new node in a linked list

- At the beginning
- At the end
- After a given node
- Before a given node

- Insert Front (head, key)

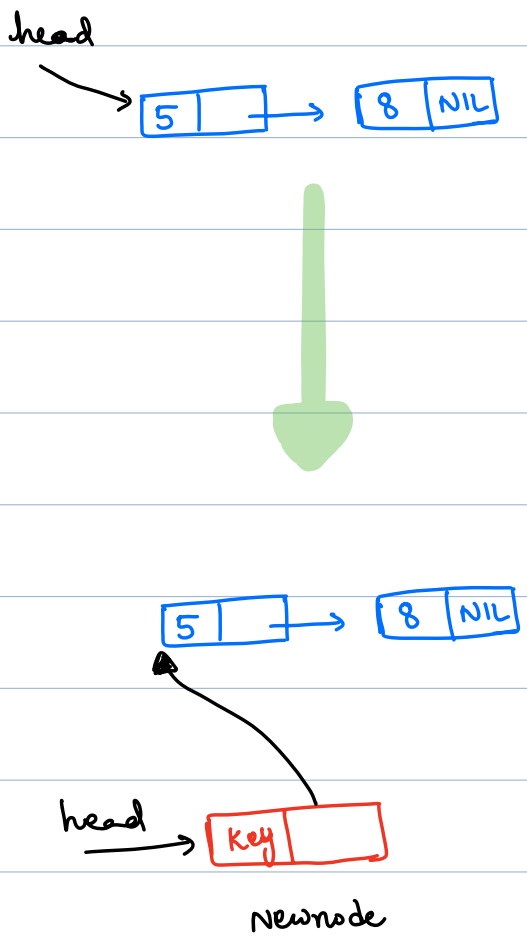
  Create a newnode

  newnode.key = key

  newnode.next = head

  head = newnode

  Return head



- Insert Back (head, key)

```
create a newnode
newnode.key = key
Ptr = head
While Ptr.next != NULL
    Ptr = Ptr.next
Ptr.next = newnode
```

inserting a node after a given node

Here we want to insert a new node after a node that has value num.

```
Insert-middle (head, num, Val)
    Create a new node
    new node.key = val
    Ptr = head,
    Ptr1 = Ptr
    While Ptr1.key != num
        Ptr1 = Ptr
        Ptr = Ptr.next
    Ptr1.next = newnode
    new node = Ptr
    return start
```

Similarly,

insert a node before a given node.

Deleting a Node from a linked-list

- Delete Front (head)

```
head = head.next
return head
```

Deletes 1st Node

- Delete end (head)

```
Ptr = head , Ptr1 = head
while( Ptr.next != NULL)
    Ptr1 = Ptr
    Ptr = Ptr.next
Ptr1.next = NULL
return Start
```

Deletes last node

Remark:

Given a sorted linked list LIST. We want to search for an item in LIST.

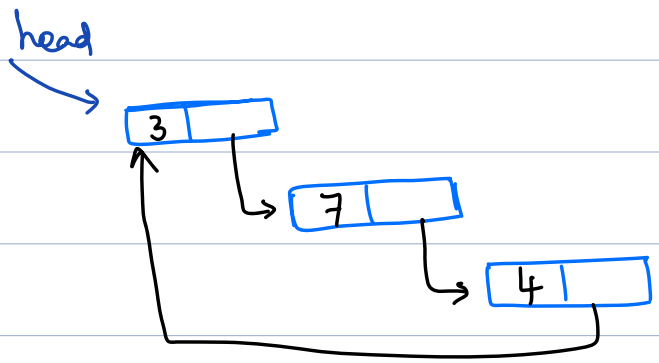
This can be done by comparing item with value present at each node, one by one of LIST. The complexity of this procedure is $O(n)$, where n = # of elements in LIST.

Recall that with a sorted array we can apply a binary search whose running time is proportional to $\log_2^n$. But binary search algorithm cannot be applied to a sorted linked list, since there is no way of indexing the middle element in the list.

This is one of the drawback in using linked list as a datastructure.

Circular linked lists

A circular linked list is a linked list, in which the last node contains a pointer to the first node of the list.



Inserting a New Node in a Circular Linked List

- at the beginning/front
- at the end
- After a given node

Inserting at the begining

```
Insert-Circular-front ( head, value)
    Create a newnode
    newnode.key = value
    Ptr = head
    While  Ptr.next != head
            Ptr = Ptr.next
    Ptr.next = newnode
    newnode.next = head
    head = newnode
    return head.
```

Similary we can insert a newnode at the end

Deleting a Node in a Circular Linked List

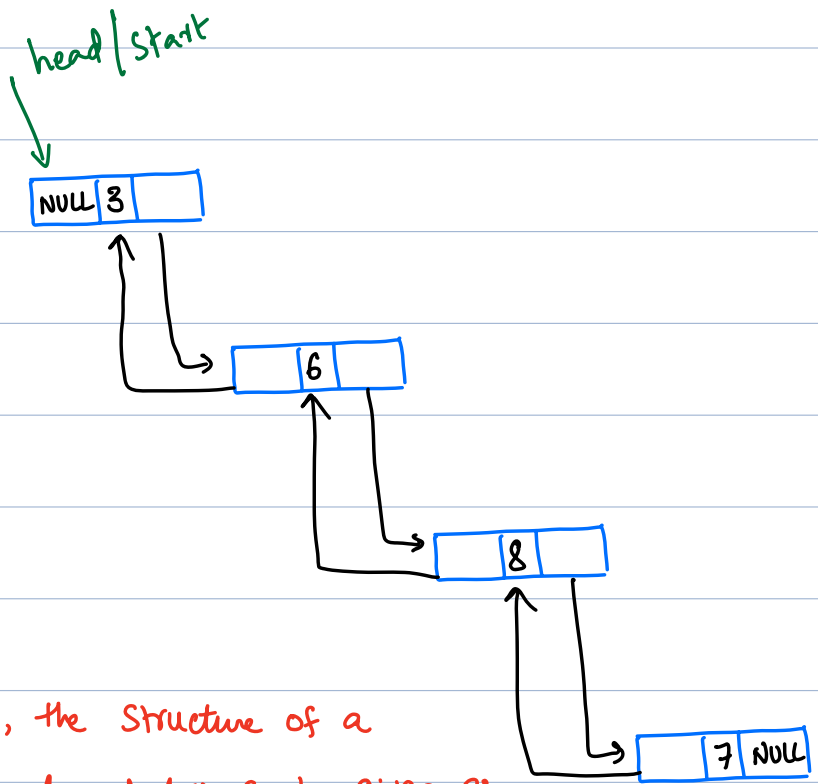
- The first node
- The last node

```
Delete-circular-front (head)
    Ptr = head
    While Ptr.next != head
        Ptr = Ptr.next
    Ptr.next = head.next
    return head
```

Similarly we can delete the last node

Doubly linked lists

A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.



In C, the structure of a doubly linked list can be given as

```
Struct node
    {
      Struct node  * prev ;
      Struct node  * next ;
       int  data ;
    }
```

Doubk-Insert Front (head, Val)

Create a newnode

newnode.key = Val

head.Prev = newnode

newnode.next = Start

newnode.Prev = NULL

head = newnode

Return head.

Similarly to the single linked lists we can to other operations on double linked lists.

Practice Questions

① Write a Program to delete k^{th} node from a linked list

② Write a Program to reverse a linked list using
ⓐ iterative method ⓑ recursion

③