Lecture #7

LIST

→ It is a Data Structure.

→ It is ADT (Abstract Data Type)

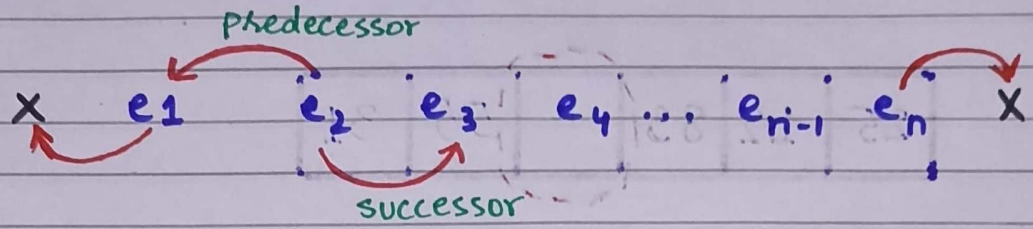
→ A set of values written down one below the other.

→ A number of connected items or names written or printed consecutively, typically one below the other.

→ List is a contagious memory allocation of homogeneous elements.

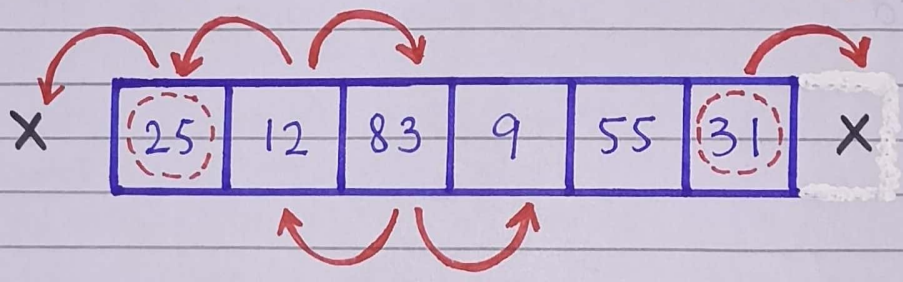
→ To Make A List (FOR A LIST)

* The elements of the list must have predecessor and successor relationship.



* The first element of the list has no predecessor therefore it is marked as the beginning of the list.

* The last element of the list has no successor therefore it is marked as the end of the list.



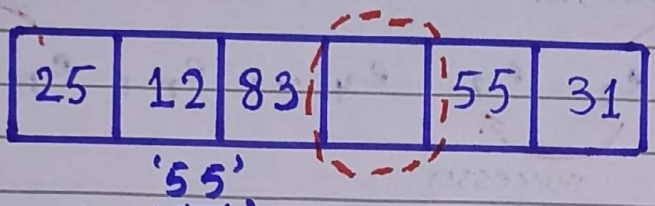
* Element '25' does not have any predecessor therefore indicates the beginning of the list.

* Element '31' does not have any successor therefore indicates the end of the List.

* All other elements have their predecessor and successor to maintain the list property.

→ INVALID LIST

* A list having vacant / empty space in the mid of the elements.



* Element '55' does not have any predecessor so it will be considered as the last element which is logically wrong, because there are two more elements exist after the empty cell.

* This list will be considered as "invalid".

TYPES OF LIST

(1) Unordered List

(2) Ordered List

(1) UNORDERED LIST :

* Elements are placed in the list in no particular order.
* Unordered list does not have in symentic or logical relation.
* It is only a list.
* Here, only relationship of predecessor and successor holds.
* We mostly maintains unordered list.
* In unordered list, the given list could be sorted as well as unsorted.

(2) ORDERED LIST:

* Elements maintain certain order either numerically or alphabetically.
* Elements are ordered according to some rule (relationship)
* Each element followed by one another.
* In ordered list, there have any relationship according to which data is arranged.
* In ordered list, we are always given sorted list.

NOTE:

* It is not necessary that ordered list will always be sorted and it is also not necessary that unordered list will always be unsorted.

→ UNSORTED LIST

RAYAN
SHABIR
ALI
SHAHMEER

→ SORTED LIST

* Names are sorted alphabetically.

ALI
RAYAN
SHABIR
SHAHMEER

PROPERTIES / TERMINOLOGIES OF A LIST:

→ To maintain a list of elements, following things should be considered

• Max Size

★ Every list must have some "Max Size" → total available sort (memory)

↳ represent total capacity (which it maximum holds data)

• Current Size

★ Every list must also have "cur Size" → sorts (memory) which we use

↳ current number of elements a list holds.

★ Initial curSize is '0'.

NOTE:

★ We mostly deals with "curSize".

If maxSize is '10' and curSize is '5', then traverse / processing will be done upto curSize (i.e. 5th location).

GENERAL OPERATIONS PERFORMED ON A LIST:

(1) **INSERTION**: Adds an element in the list.

(2) **DISPLAY**: Display the complete data of list

(3) **SEARCH**: Searches an element using the given key.

(4) **UPGRADE**: Upgrades an element with new one.

(5) **DELETION**: Deletes an element using the given key.

(6) **MERGE**: It merges two lists.

(7) **SPLIT**: Dividing list into two parts.

(8) **COMPARISON**: compares a list with another list.

(1) INSERTION IN UNORDERED LIST:

* It inserts / adds an element in the list.

* To perform insertion in the list, we first have to check if either the list is full or not:

curSize == MaxSize?

Is curSize equal to MaxSize?

* If at start curSize = 0 and user wants to insert element '5' then this new element will be inserted at the position **curSize + 1** (i-e. 1st location)

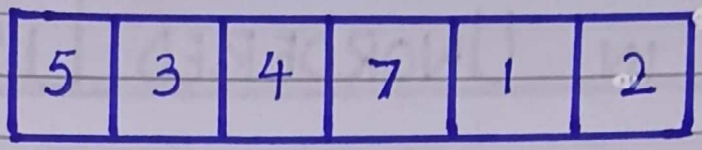
curSize ++
→ curSize = 1

* Then if user again want to insert element '3' then again check **curSize == maxSize**

Then this new element will be inserted at the position **curSize + 1**

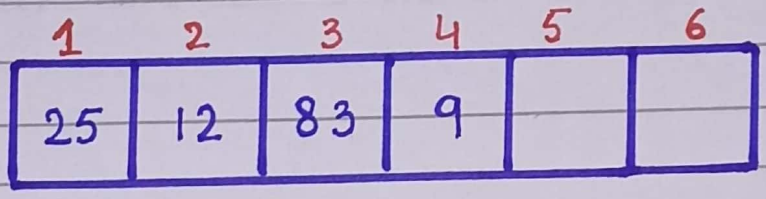
cursize ++
→ cursize = 2

Then again check repeating by same above process.



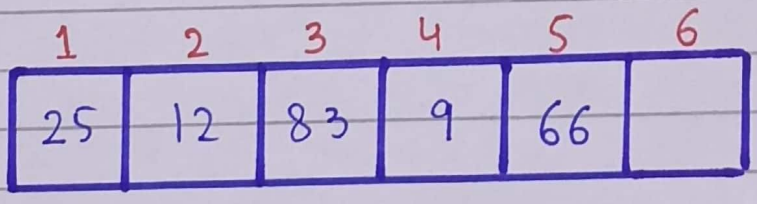
curSize = 6

* Whenever we insert an element in an unordered list, we insert it at the last (curSize + 1) index, because it is an unordered list, so we don't need to write data in order, we always in insert the new element at the end.



→ The max size of this is '6' and current size is '4'.

→ A new element '66' is inserted in the list on position **current size + 1** i.e **5** in this case.



→ The current size raises to '5'.

→ Whenever we do insertion, we will increment the curSize by '1'.

→ Insertion takes constant time to evaluate. So, its time complexity is **O(1)**.

2) DISPLAY IN UNORDERED LIST:

* Whenever we print data of a list, we print the data upto **curSize**.
* We don't print upto **maxSize** because we want to print the data which is present on the locations we are using.
* So, that is the reason, we execute our loop from **0** to less than curSize **(< curSize)** that data is in use.

(3) SEARCHING IN UNORDERED LIST:

* Searching of an element in the list simply be done by comparing the required **key** element **one by one** with the elements of the list.
* Checks each element with a **"key"** element.
* But here we first have to check "if the list is empty or not"

curSize = 0? (if list is empty)

if curSize > 0 (if list is not empty)

→ If it is return true that List is not empty, then traverse (loop will be evaluated to process data upto ursSize)

→ So, here, all elements (i.e **n** elements) are being checked one by one. So, its time complexity will be **O(n)** or **Θ(n)**.

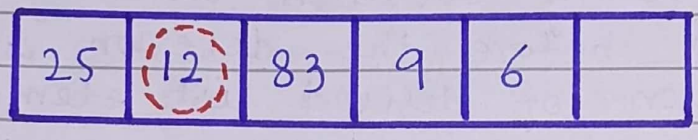
★ The Searching operation usually performed before the deletion and/or upgradation of desired list element.

(4) DELETION IN UNORDERED LIST:

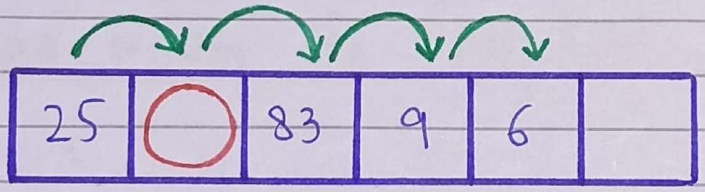
★ "Deletion" operation cannot perform without "Search" operation.

★ Whenever we want to delete in an unordered list, first we have to search.

If we have a list

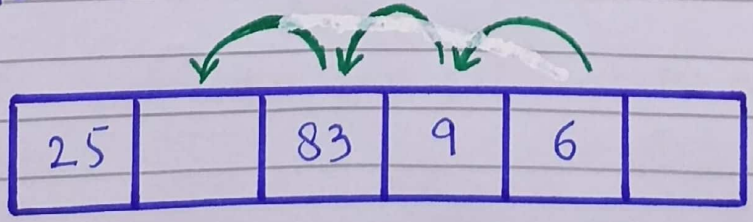


→ And we want to delete '12'.



→ After removal an empty (vacant) space is left, so here rule of predecessor and successor violates. It becomes invalid list.

★ → To maintain the property of the list (predecessor and successor), after removing/deleting an element from unordered list and still makes it a list.



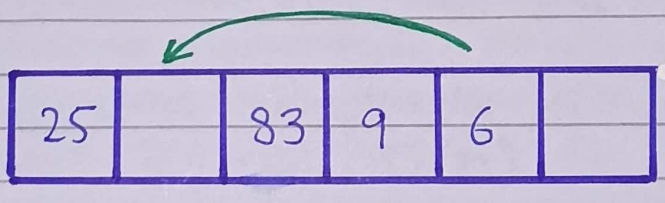
* Here we shift an element towards the empty cell and decrement the curSize by one.
  → But the shifting here is not valid because it requires **'n-1' (or 'n')** shiftings.

* Shifting element like this will take **O(n)**.

  → As list in unordered;
  So, best way to remain it as a list **(maintain property of the list)**

* To make it efficient, simply replace the deleted element with the last element of list and decrement the **curSize** by one.

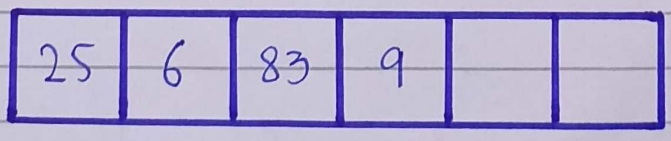
Like:



also

```
curSize --;
```

So, list will be:



The time complexity for deletion for unordered list is also **'n' → O(n)**.

* It is because deletion also first search the element, for which its time complexity will become 'n' (O(n))

* Then, after searching, the element will be deleted and last element will be replaced to empty cell, for this, its time complexity will be O(1).

* So, orignally the timecomplexity for deleting will be O(1) and for searching will be O(n). The overall time complexity for deletion operation will be 'n+1' i-e O(n).

(5) UPGRADE IN UNORDERED LIST:

* First check if list is not empty;

curSize = 0 ?

* "Upgrade" operation also cannot perform without "Search" operation.

* To Upgrade an unordered list, first we have to search for the element which we want to update with new element

* But here neither **curSize ++** nor **curSize --**, because here value is only updating; neither adding nor decreasing.

So, time complexity of unordered list is also **O(n).**

(6) MERGE:

* We merge/join two lists.
* The resultant will be a combination of the elements of both lists.
* If we want to merge lists

l_1 = **n** elements
l_2 = **m** elements

then resultant list will be:

resultant = **n+m** elements

(7) SPLITTING:

* We divide a list into two parts.

(8) COMPARISON:

* We compare two lists with each other.

GENERAL OPERATIONS FOR ORDERED LIST:

* Here list is sorted.

(1) INSERTION IN SORTED LIST:

* Here every element couldn't be placed directly at last position, we first have to check where to place the element according to some relationship.

* If we want to insert

element 1 : e1 = 1
element 2 : e2 = 4

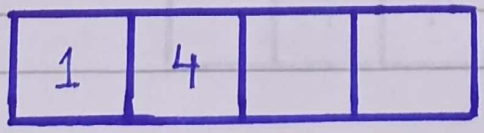
* Here each element will not be placed one after another.
First, we have to check

if $e_1 < e_2$? (CHECK)

$e_1 < e_2 < e_3 < e_4 < e_5 \ldots < e_n$

and then insert.

→ So, after checking



→ then if we want to again insert

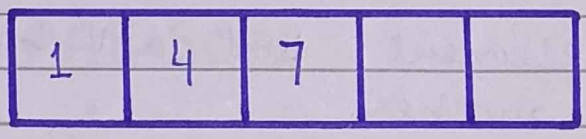
$e_3 = 7$

★ Then again, we will compare element. But remember we always compare with last element i.e **cursize**.

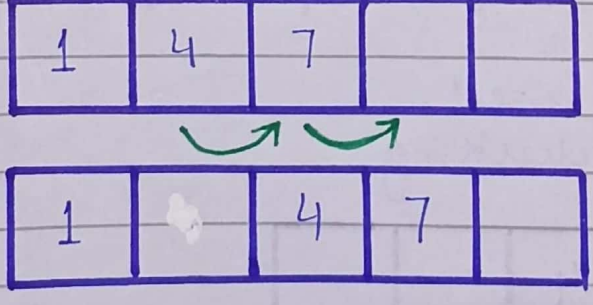
★ And if the element (which is being compared) comes according to relationship i-e **"less than cursize"** (< cursize), then we keep comparing backward until we place the element at right position.

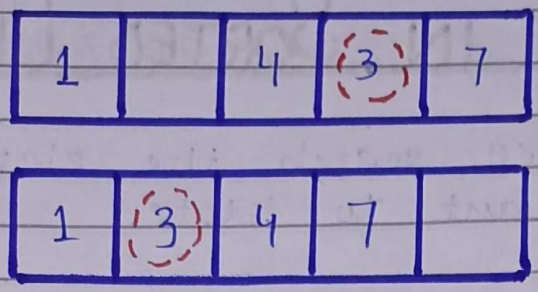
★ So, here better loop choice is **"While"** because iterations here are not known.

★ List became:



→ Then, suppose we want to insert a new element '3' in a sorted list, we have to shift all elements one step forward to place '3' in right place.





* In ordered list (sorted list), we must have to shift elements, to maintain order of list.

* Thus, time complexity of insertion in sorted list is **O(n)**.

NOTE:

So, Best way to make a list, first maintain unordered list and then maintain its order by using some sorting algorithm.

(2) SEARCHING IN SORTED LIST:

* In sorted list, we have to traverse each element with one another.

So, its time complexity is also **'n'** → **O(n)**.

(3) DELETION IN SORTED LIST:

* First, we will search the element which we want to delete.
* Here we cannot place the last element to the deleted position (like we did in unordered list) because its order will not maintain.
* So, here, only way to delete element is by shifting, so that order maintains.

So,
its time complexity will be

$$\underbrace{n}_{\text{Search}} + \underbrace{n}_{\text{Shift}} = 2n$$

(4) UPGRADE IN SORTED LIST:

* First, we will have to search the element which we want to upgrade.
* Then, we will check the value with which we want to upgrade the element.
We will check if the element is on its right position or not by comparing / shifting elements
* So, here time complexity will be

$$\underbrace{n}_{\text{Search}} + \underbrace{n}_{\text{Shift}} = 2n$$

(5) MERGE:

* To merge two sorted lists, we cannot directly merge them.
* But we have to use some reliable algorithm to perform merge because we have to maintain order of resultant list.

(6) SPLITTING:

* Splitting is easy in a sorted list; we will simply divide the list into two parts and each part will be in sorted form.

(7) COMPARISON:

* We compare two sorted lists with each other.

NOTE:

* Generally, it is a good idea to maintain unsorted lists at first instead of maintaining order of elements.
* List can be sorted easily with help of some efficient sorting algorithms later or when needed.
* In DSA, we avoid to use "Continue", "Break" and "GOTO" statements.

Lecture #8

ARRAY:

* "Memory Representation of List is done by Array."
* It is a data structure.
* Array data structure enables us to store a group of data together in one place.
* Array is a finite collection of homogeneous data elements
* All the elements of array are stored in contigious locations of memory.

ARRAY TERMINOLOGY:

→ **Size:**
The number of elements in an array.

→ **Type:**
The kind of data an array represents e.g. integers, character string, etc.

→ **Base:**
memory address of the first element of array.

→ **Word:**

Denotes the size of an element.

→ **Index:**

A subscript integer value used to refer elements of the array.

→ **Range of Indices:**

Indices of the array elements may change from lower bound (LB) to an upper bound (UB), and these bounds are also called the **"Boundaries of an Array"**.

ARRAY MEMORY ALLOCATION:

* Array can be declared in one of two ways in C++.

(1) IN STACK:

```
int ar[50];
```

→ Declares an array (ar) of 50 integers on stack memory.
→ Automatically deallocated when goes out of scope.

(2) IN HEAP:

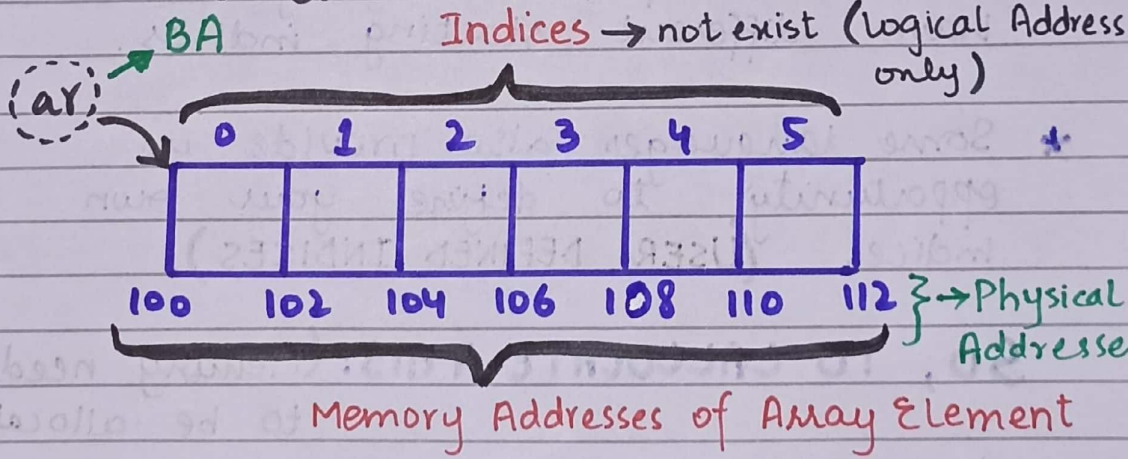
```
int *ar = new int[50];
```

→ Declares an array (ar) of 50 integers on Heap memory.
→ Needs to be deallocated by programmer with the help of `delete []ar;` statement.

ARRAY DESCRIPTOR TABLE

* Whenever array creates, there is a descriptor table behind it.
* This table is known as: "ARRAY DESCRIPTOR TABLE".
* It holds **Name** of array.
* It holds **Base Address** (Starting Address) of array.
* It holds **Word Size**.
* It holds **Lower Bound** (LB) and **Upper Bound** (UB).

```
int ar[10];
```



Base Address = **100**, Element Size (Word Size) = **2 Bytes**, Lower Bound (index) = **0** and, Upper Bound (index) = **5**.

Name	Base Address	Word Size	Lower Bound	Upper Bound
ar	100	2Bytes	0	5

* In most of the arrays, Base Address is written as **"Alpha"** or **"Beta"**.

ARRAY MEMORY ALLOCATION:

* Most language start with index **'0'** like C, C++, etc.

* But most of the language also have different starting indices.

* Some languages also provide us opportunity to define your own indices (USER DEFINED INDICES).

SO, TO CALCULATE THIS: (Memory needs to be allocated)

First system FIND: (Array size count of Element)

Upper Bound (Index) = 5
Lower Bound (Index) = 0

} These can also be used for Bound checking.

Count = UB - LB + 1

Size (Array) = 5 - 0 + 1

Size (Array) = 6

→ Now, calculating Memory size of Array:

Base Address = 100

Element Size = 2 Bytes (Word size)

Size (Array) = 6

Memory Size (Array) = Size (Array) × Word size

= 6 × 2

= 12 Bytes

ARRAY MAPPING FORMULA:

→ So, to calculate the Memory Address:

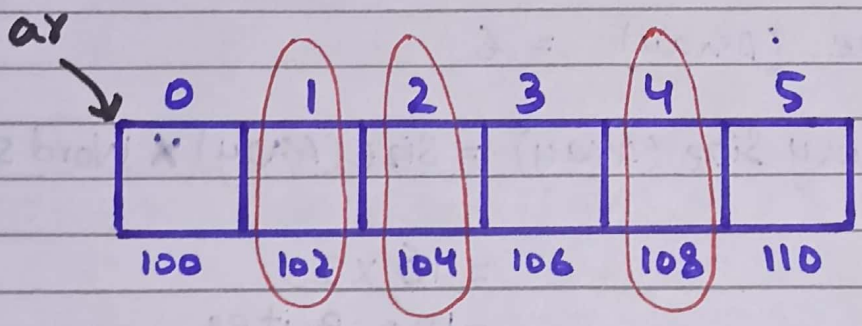
Base Address = 100

Element Size = 2 Bytes (Word size)

Address(A[index]) = Base Address + Offset

Address(A[i]) = Base Address + (i x word size)

Address(A[1]) = 100 + (1 x 2) = 100 + 2 = 102
Address(A[2]) = 100 + (2 x 2) = 100 + 4 = 104
Address(A[4]) = 100 + (4 x 2) = 100 + 8 = 108



GENERAL FORMULA (FOR FINDING ADDRESS WHICH WILL BE MAPPED ON MEMORY ACCORDING TO INDEX)

Address (Index) = BA + (Index * ES)

↳ The complexity of this algorithm is constant **O(1)**.

NOTE: IF

starting index = '1'

then,

Address (Index) = BA + (Index - 1 * ES)

↳ Bcz we then go at start location ('0' index)

ONE-DIMENSIONAL ARRAY (1-D)

* If only **one subscript / index** is required to reference all the elements in an array, then the array is termed as: **"ONE-DIMENSIONAL / 1-D ARRAY."**

* Array elements are logically referred with the help of index / subscript e.g.

  ar[i] refers to the **i^{th}** element of the array.

* But physically memory address is required to map the particular index to its actual location in the memory.

* 1-D array does not exist in memory, Basically, it exists as 'memory junks'.

ARRAY GENERAL OPERATIONS

→ **INSERTION:**
Adds an element in the array.

→ **TRAVERSING:**
Accessing all the array elements.

→ **SORTING:**
Arranging the elements in certain order.

→ **SEARCH:**
Searches an element using the given key.

→ **DELETE:**
Deletes an element using the given key.

→ **MERGING:**
Combining two or more arrays into one.

→ **SPLITTING:**
Dividing an array into two or more sub arrays.

→ COMPARING: Check similarity among the arrays.