CS8391
DATA STRUCTURES

Unit I – Linear Data Structures And List

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation – singly linked lists- circularly linked lists- doubly-linked lists – applications of lists – Polynomial Manipulation – All operations (Insertion, Deletion, Merge, Traversal).

Data Structures

Definition 1

- Data Structures is a way of Organizing,
- Storing and
- Retrieving Data
- And representing their relationship with each other.

Data Structures

Definition 2

- A *data structure* is basically a group of data elements
- That are put together under one name, and
- Which defines a particular way of storing and organizing data in a computer
- So that it can be used efficiently.

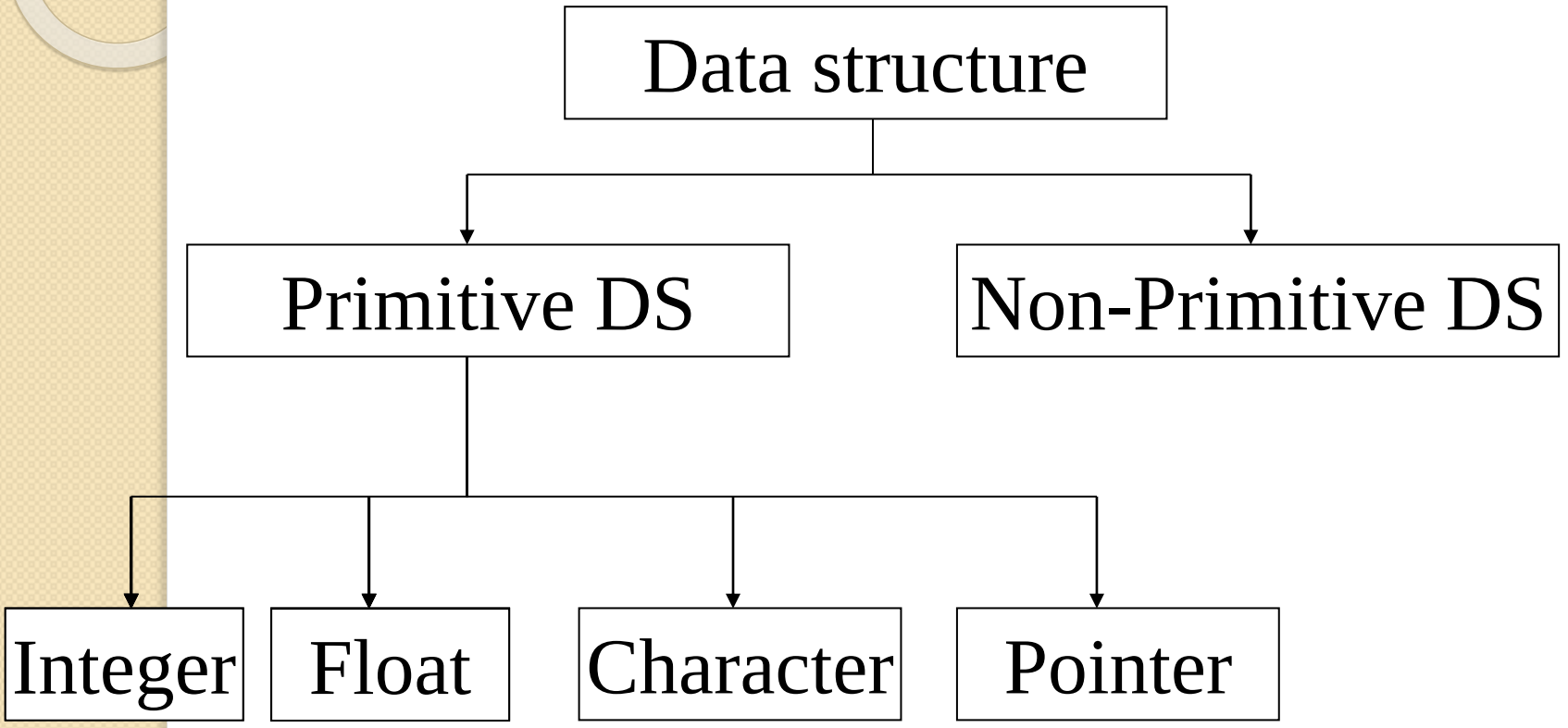
Data Structure

- Data structures are building blocks of a program.
- Data Structures = Related Data + Allowed Operations.
- Program = Algorithm + Data Structures.

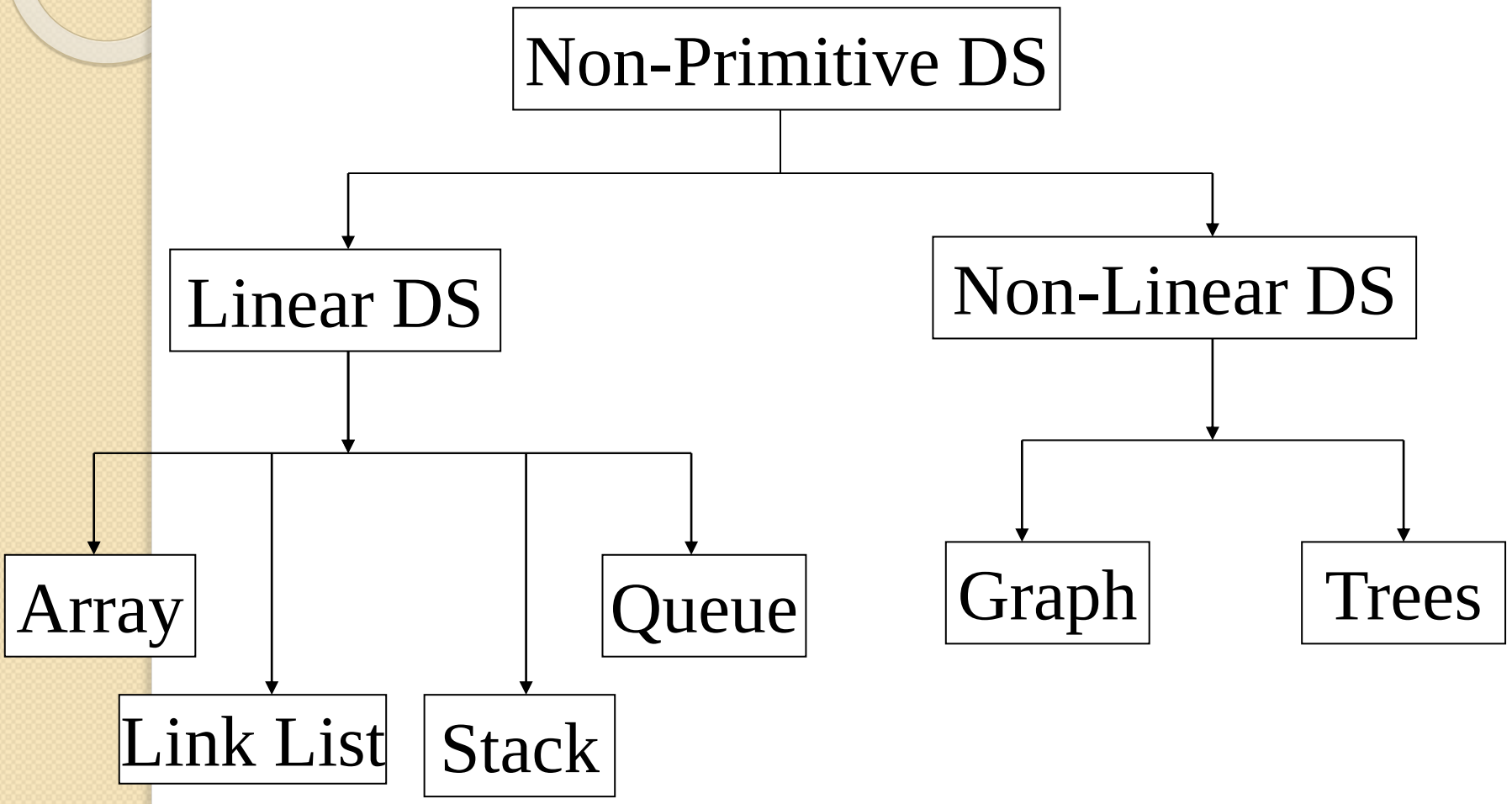
Operations on Data Structures

- Traversing
- Searching
- Inserting
- Deleting
- Sorting
- Merging

Classification of Data Structure



Classification of Data Structure



Primitive Data Structure

- There are basic structures and **directly operated** upon by the machine instructions.
- In general, there are different representation on different computers.
- Integer, Floating-point number, Character constants, string constants, pointers etc, fall in this category.

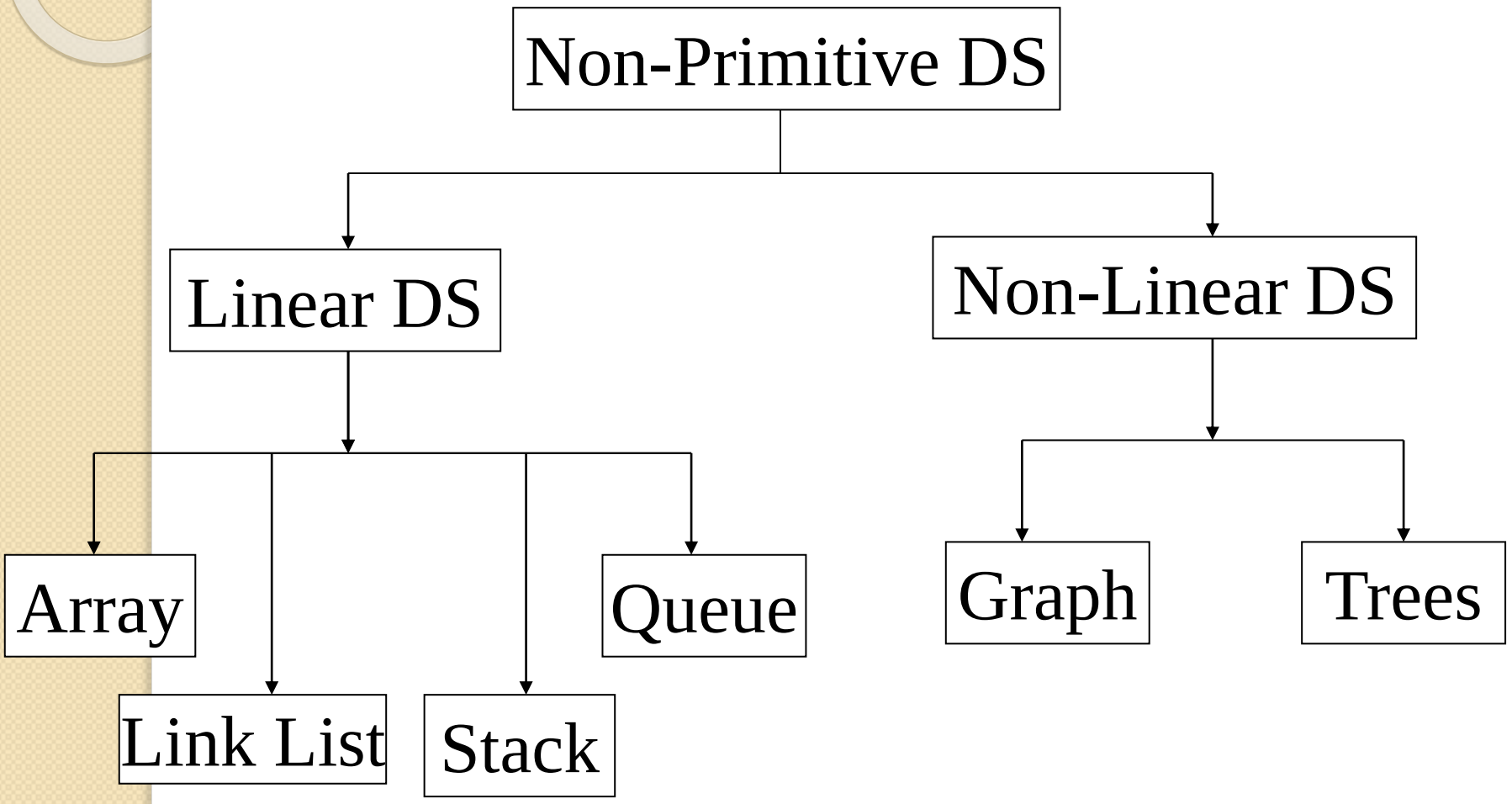
Non-Primitive Data Structure

- There are more sophisticated data structures.
- These are derived from the primitive data structures.
- The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.
- Lists, Stack, Queue, Tree, Graph are example of non-primitive data structures.
- The design of an efficient data structure must take operations to be performed on the data structure.

Different between them

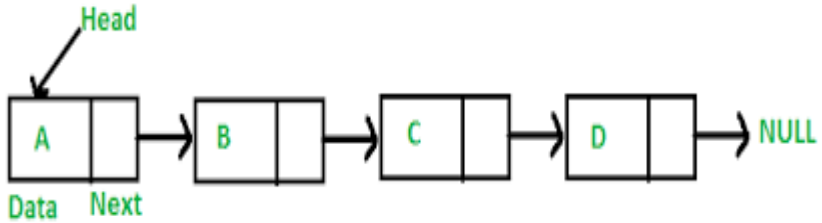
- A primitive data structure is generally a basic structure that is usually built into the language, such as an integer, a float.
- A non-primitive data structure is built out of primitive data structures linked together in meaningful ways, such as a or a linked-list, binary search tree, AVL Tree, graph etc.

Classification of Data Structure



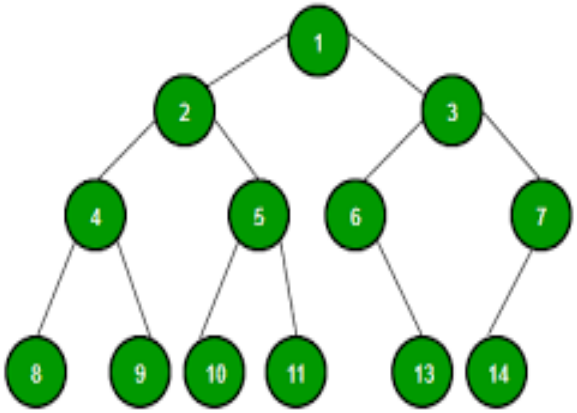
Linear Data Structures

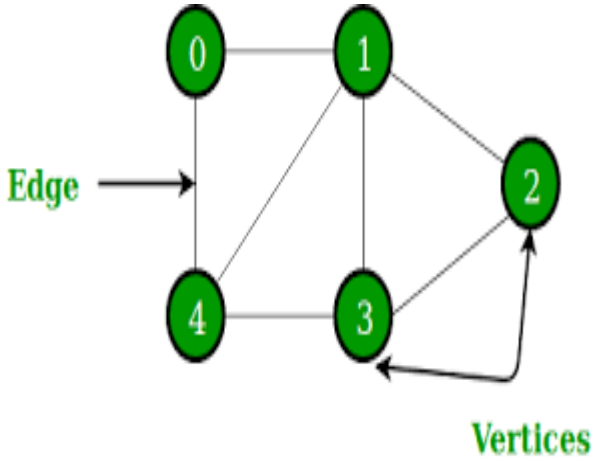
- If the elements of a DS are stored in a linear or sequential order then it is called as Linear DS.
- Linear DS can be represented in memory in two different ways.
- One way is to have to a linear relationship between elements by means of sequential memory locations.
- The other way is to have a linear relationship between elements by means of links.
- Ex : List, Linked List, Stack, Queue.



Non Linear Data Structures

- If the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure.
- A DS which represents a hierarchical arrangement is called Non Linear Data Structures.
- Ex : Tree, Graph.





Applications of Data Structures

- Compiler Design
- Operating System
- Statistical Analysis
- Numerical Analysis
- Artificial Intelligence
- Database Management System
- Graphics

CS8391 DATA STRUCTURES

UNIT I LINEAR DATA STRUCTURES – LIST

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation —singly linked lists- circularly linked lists- doubly-linked lists – applications of lists –Polynomial Manipulation – All operations (Insertion, Deletion, Merge, Traversal).

UNIT II LINEAR DATA STRUCTURES – STACKS, QUEUES

Stack ADT – Operations - Applications - Evaluating arithmetic expressions- Conversion of Infix to postfix expression - Queue ADT – Operations - Circular Queue – Priority Queue - deQueue – applications of queues.

UNIT III NON LINEAR DATA STRUCTURES – TREES

Tree ADT – tree traversals - Binary Tree ADT – expression trees – applications of trees – binary search tree ADT –Threaded Binary Trees- AVL Trees – B-Tree - B+ Tree - Heap – Applications of heap.

UNIT IV NON LINEAR DATA STRUCTURES - GRAPHS

Definition – Representation of Graph – Types of graph - Breadth-first traversal - Depth-first traversal – Topological Sort – Bi-connectivity – Cut vertex – Euler circuits – Applications of graphs.

UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES

Searching- Linear Search - Binary Search. Sorting - Bubble sort - Selection sort - Insertion sort - Shell sort – Radix sort. Hashing- Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

Abstract Data Type

- An ADT is the way we look at a DS *focusing what it does*, ignoring how it does the job.
- An ADT is a set of elements together with the set of well defined operations.
- ADT is an extension of Modularity design.
- ADT is a model used to understand the design of DS.
- Abstract : Means not considering detailed specifications or implementation.
- Data Type : Data type of a variable is the set of values that the variable can take.

Abstract Data Type

- Advantages
  - ADT make simple the modification to a program.
  - It separates the use of a DS from the details of its implementation.
  - If the element at position i is Ai then its successor is A_{i+1} is and its predecessor is A_{i-1}
- List ADT
  - List is an ordered set of elements.
  - Collection of Elements is called as a List

    $A_1, A_2, A_3, \ldots\ldots\ldots\ldots A_n$
  - Where $A_1 \rightarrow$ First Element of the list

    $A_n \rightarrow$ Last Element of the list

    $n \rightarrow$ Size of the List

Operations on List ADT

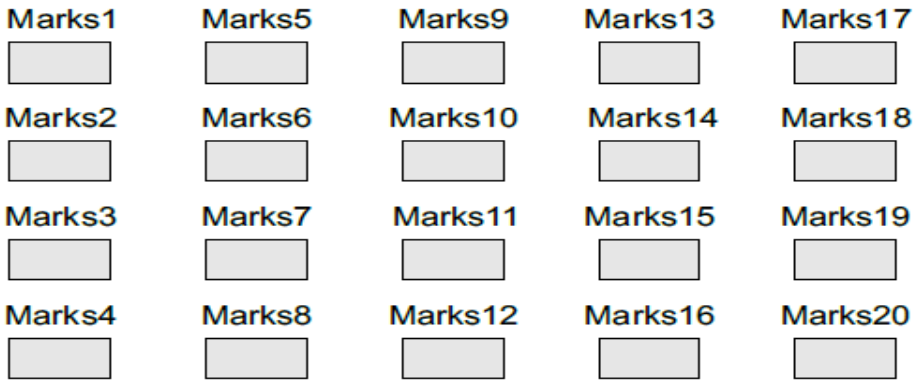
- Insert(x,6) - Insert the element x after the position 9.
- Delete(x) – The element x will be deleted.
- Find(x) - Returns the position of x.
- Previous(i) – Returns the position of its predecessor(i-1).
- Next(i) - Returns the position of its successor(i+1).
- PrintList() - Contents of the list is displayed.
- MakeEmpty() – Makes the list empty.

Implementation of List ADT

- Array Implementation
- Linked List Implementation
- Cursor Implementation

Array Implementation

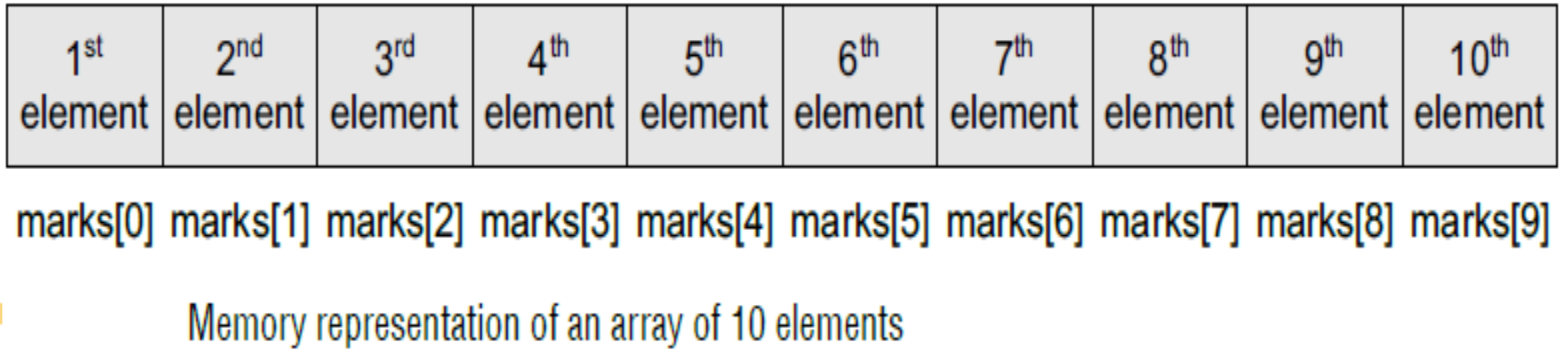
- An array is a collection of similar data elements.
- These data elements have the same data type.
- The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the *subscript*).
- The subscript is an ordinal number which is used to identify an element of the array.



Twenty variables for 20 students

Declaration of Arrays

- Declaring an array means specifying the following:
- *Data type*—the kind of values it can store, for example, int, char, float, double.
- *Name*—to identify the array.
- *Size*—the maximum number of values that the array can hold.
- Syntax: type name[size]; Ex: int marks[10];



Memory representation of an array of 10 elements

Operations on Arrays

- Traversing an array
- Inserting an element in an array
- Searching an element in an array
- Deleting an element from an array
- Merging two arrays
- Sorting an array in ascending or descending order

Traversing an Array

- Traversing the data elements of an array A can include printing every element, counting the total number of elements, or performing any process on these elements.
- Array is a linear data structure (because all its elements form a sequence), traversing its elements is very simple and straightforward

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:      Apply Process to A[I]
Step 4:      SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

Algorithm for array traversal

Inserting an Element in an Array

- If an element has to be inserted at the end of an existing array, the
- n the task of insertion is quite simple.

```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3: EXIT
```

Algorithm to append a new element to an existing array

Insert an Element in the Middle of an Array

- The algorithm INSERT will be declared as INSERT (A, N, POS, VAL). The arguments are
- (a) A, the array in which the element has to be inserted
- (b) N, the number of elements in the array
- (c) POS, the position at which the element has to be inserted
- (d) VAL, the value that has to be inserted

```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:          SET A[I + 1] = A[I]
Step 4:          SET I = I - 1
        [END OF LOOP]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```

Algorithm to insert an element in the middle of an array.

Example

Initial `Data[]` is given as below.

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

Calling `INSERT(Data, 6, 3, 100)` will lead to the following processing in the array:

45	23	34	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	100	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

Deleting an Element from an Array

- Deleting an element from an array means removing a data element from an already existing array.
- If the element has to be deleted from the end of the existing array, then the task of deletion is quite simple.
- We just have to subtract 1 from the upper bound.

```
Step 1: SET upper_bound = upper_bound - 1
Step 2: EXIT
```

Algorithm to delete the last element of an array

Delete an element from the middle of an array

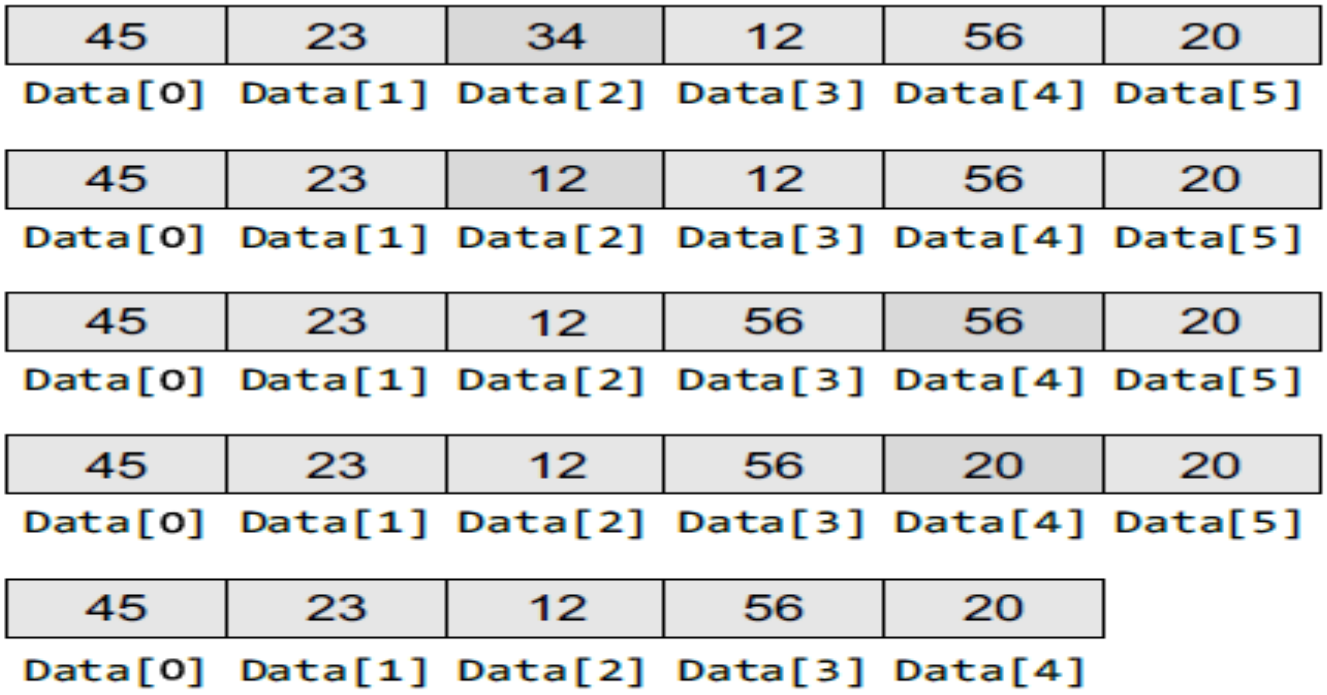
- The algorithm DELETE will be declared as DELETE(A, N, POS). The arguments are:
- (a) A, the array from which the element has to be
- deleted
- (b) N, the number of elements in the array
- (c) POS, the position from which the element has to
- be deleted.

```
Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N - 1
Step 3:          SET A[I] = A[I + 1]
Step 4:          SET I = I + 1
        [END OF LOOP]
Step 5: SET N = N - 1
Step 6: EXIT
```

Algorithm to delete an element from the middle of an array

Example

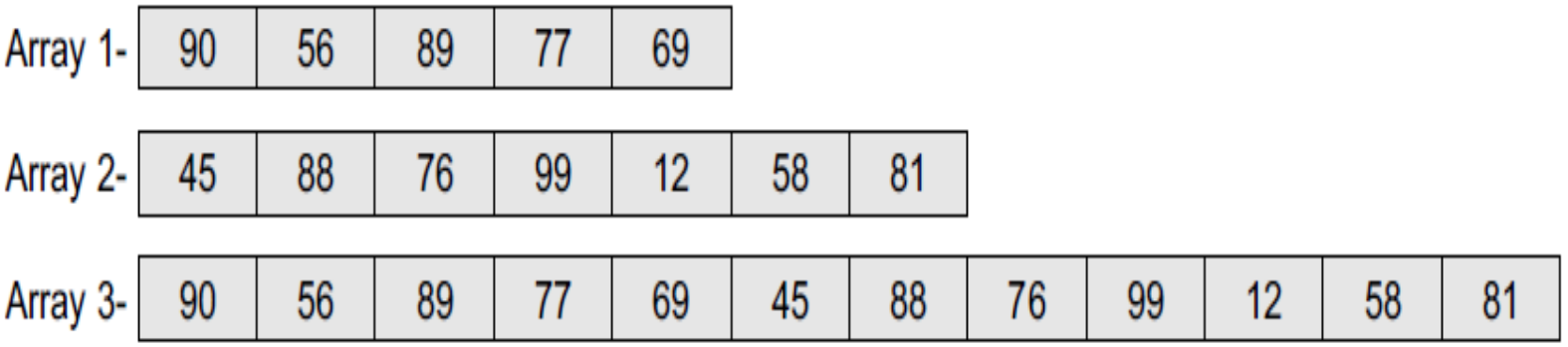
`DELETE(Data, 6, 2)` will lead to the following processing in the array.



Deleting elements from an array

Merging Two Arrays

- Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array.
- Hence, the merged array contains the contents of the first array follow ed by the contents of the second array.



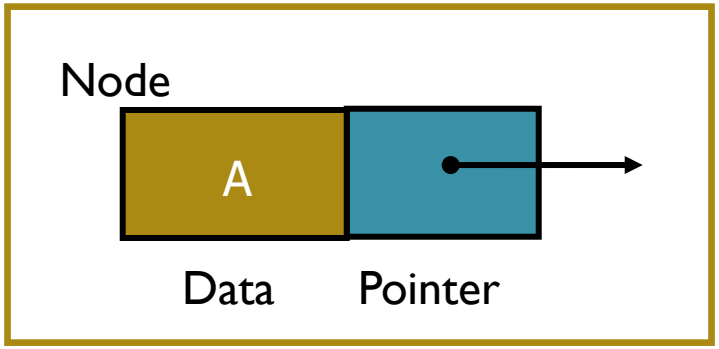
Merging of two unsorted arrays

Limitations of Arrays

- Arrays have a fixed dimension
- Once the size of the array decided it can not be increased or decreased during execution.
- Insertion and deletion operations are pretty tedious.
- To over come these limitations we can use LINKED LIST.

Linked List

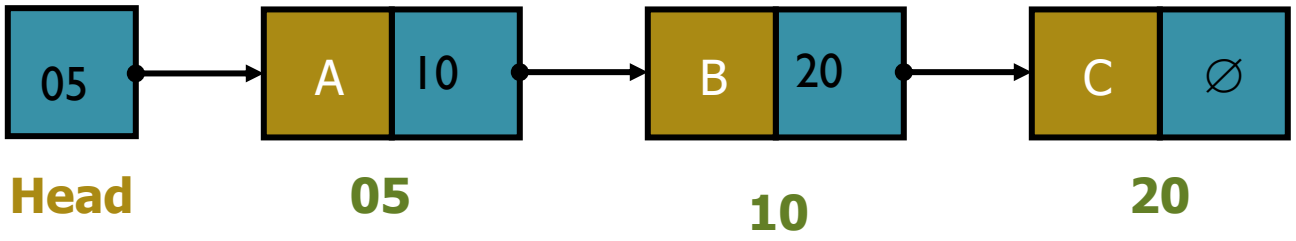
- A linked list is a series of connected nodes.
- Linked list is a collection of nodes.



- Each node contains two fields
  - Data Field(any type)
  - Address Field (Pointer to the next node in the list)

Linked List

- Head is a pointer to the first node.
- The last node of the Linked list points to NULL.



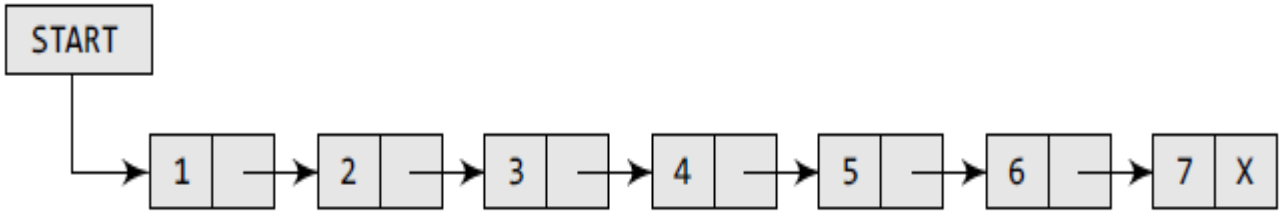
- Insertion and deletion operations are easily performed using Linked List.
- A Linked list can easily grow and shrink in size.(Dynamic)
- With a linked list, no need to move other nodes. Only need to reset some pointers.

Types of Linked List

- Singly Linked List
- Doubly Linked List
- Circular Linked List

Singly Linked List

- A SLL is a LL in which each node contains only one linked field pointing to the next node in the list.
- A singly linked list is the simplest type of linked list.
- A singly linked list allows traversal of data only in one way.



Singly linked list

Declaration for Linked List

```
Struct Node
{
        int data;
        position *next;
};
```

Operations on Singly Linked List

- Traversing a Linked List
- IsEmpty and IsLast
- Find
- Find Previous and Find Next
- Inserting a New Node in a Linked List
  - Case 1: The new node is inserted at the beginning.
  - Case 2: The new node is inserted at the end.
  - Case 3: The new node is inserted after a given node.
  - Case 4: The new node is inserted before a given node.
- Deleting a New Node from a Linked List
  - Case 1: The first node is deleted.
  - Case 2: The last node is deleted.
  - Case 3: The node after a given node is deleted.
- Merging two linked List

Traversing a Linked List

- Traversing a linked list means accessing the nodes of the list in order to perform some processing on them.
- Remember a linked list always contains a pointer variable START which stores the address of the first node of the list.
- End of the list is marked by storing NULL or –1 in the NEXT field of the last node.
- For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed.

Traversing a Linked List

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:             Apply Process to PTR->DATA
Step 4:             SET PTR = PTR->NEXT
        [END OF LOOP]
Step 5: EXIT
```

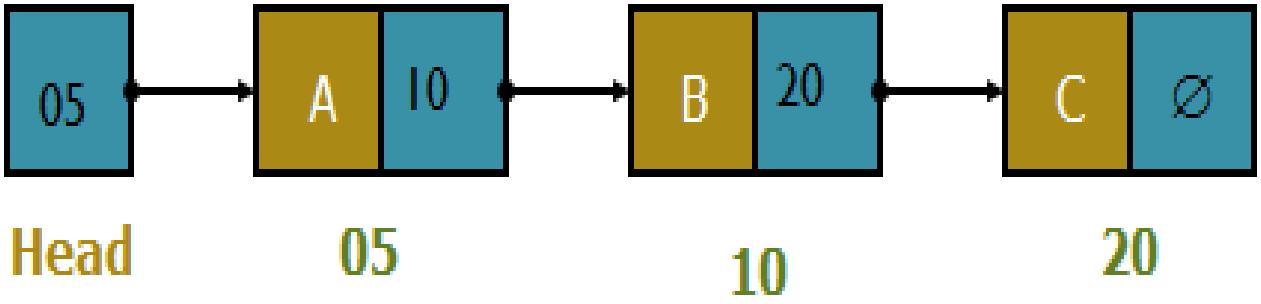
Algorithm for traversing a linked list

```
Step 1: [INITIALIZE] SET COUNT = 0
Step 2: [INITIALIZE] SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR != NULL
Step 4:             SET COUNT = COUNT + 1
Step 5:             SET PTR = PTR->NEXT
        [END OF LOOP]
Step 6: Write COUNT
Step 7: EXIT
```

Algorithm to print the number of nodes in a linked list

Example

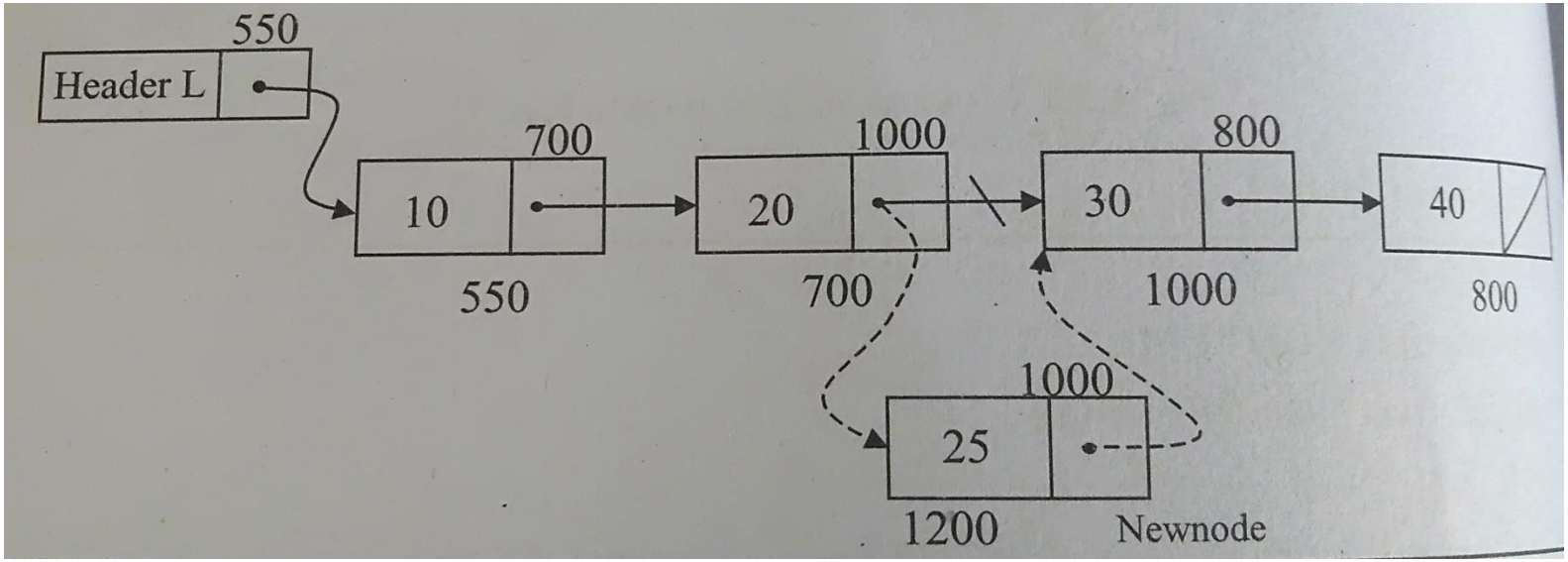
- A B C



Algorithm to Insert an element in the list

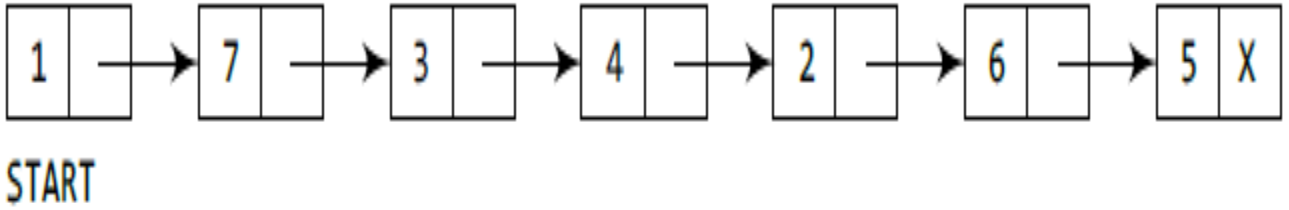
```
void Insert(int X, List L, Position P)
{
        position Newnode;
        Newnode=malloc(sizeof(Struct Node));
        If(Newnode!=NULL)
        {
                Newnode->Element=X;
                Newnode->Next=P->Next;
                P->Next=Newnode;
        }
}
```

Example

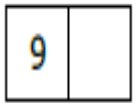

550
Header L
700
10
550
1000
20
700
800
30
1000
40
800
1000
25
1200
Newnode

Example

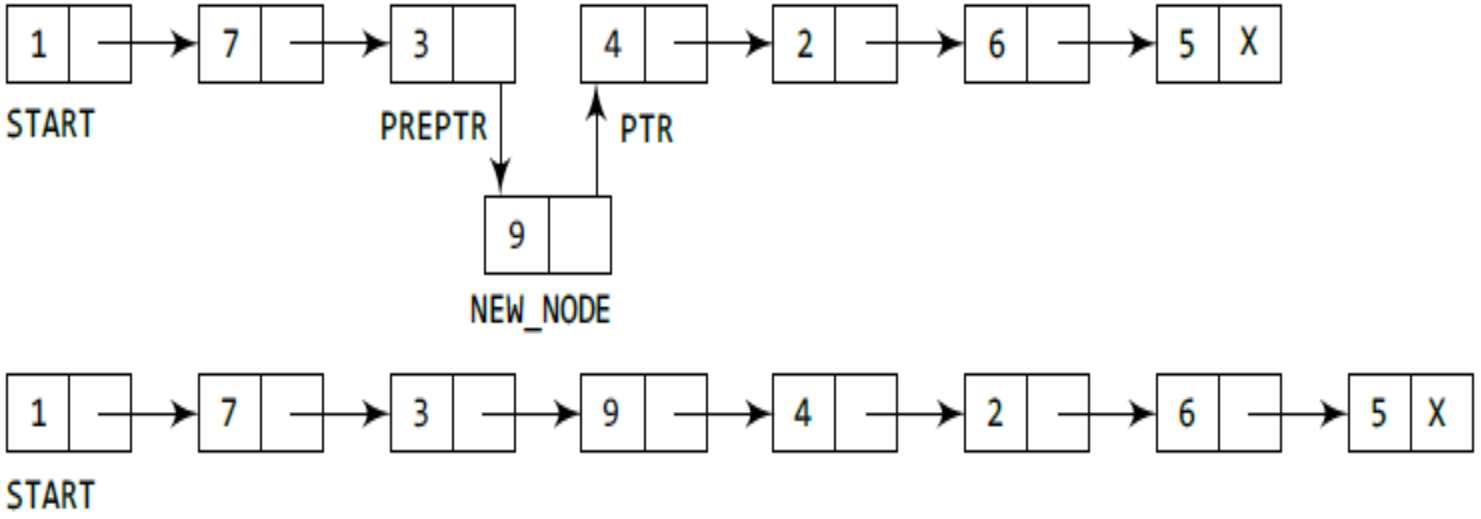
- Before Insertion



- To Insert: After 3

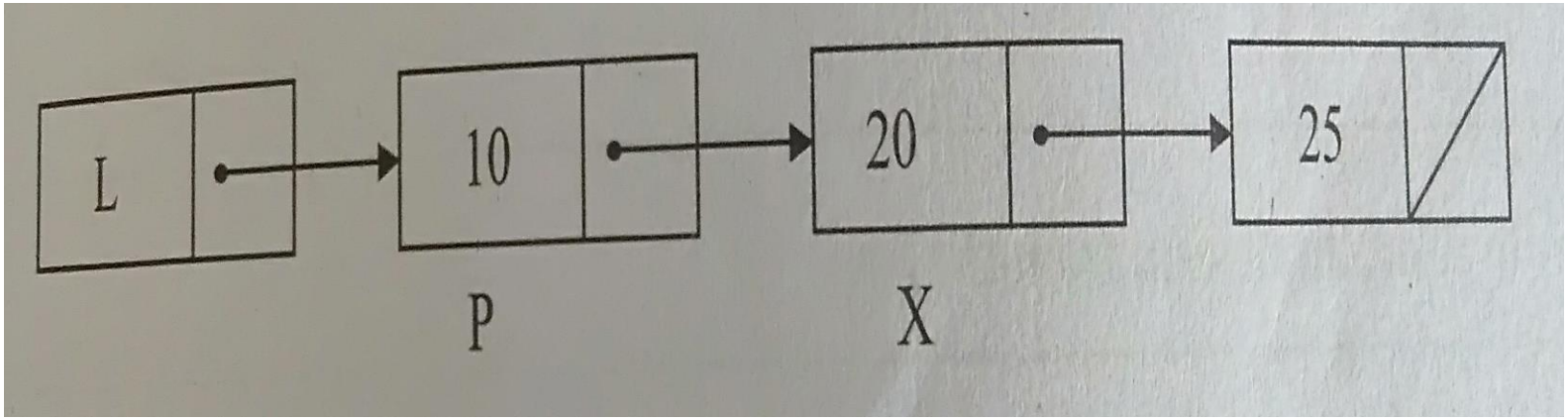


- After Insertion



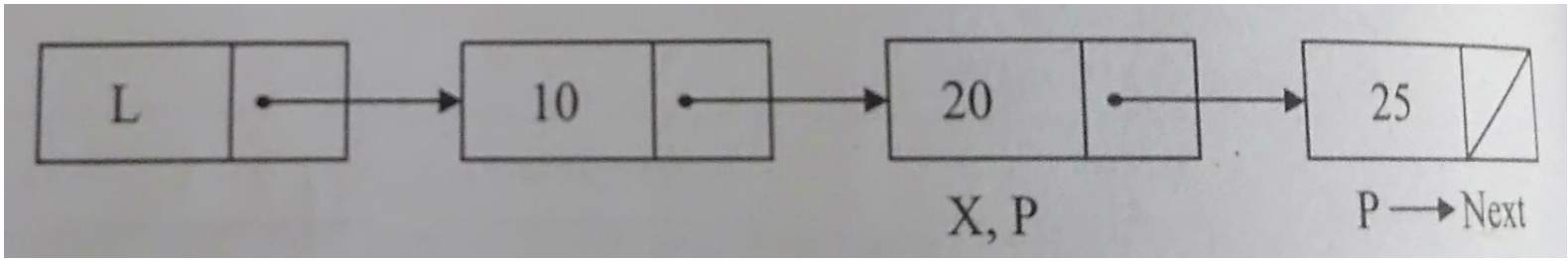
Find Routine(Algorithm)

```
Position Find(int X, List L)
{
        Position P;
        P=L->Next;
        while(P!=NULL && P->Element!=X)
                P=P->Next;
        return P;
}
```



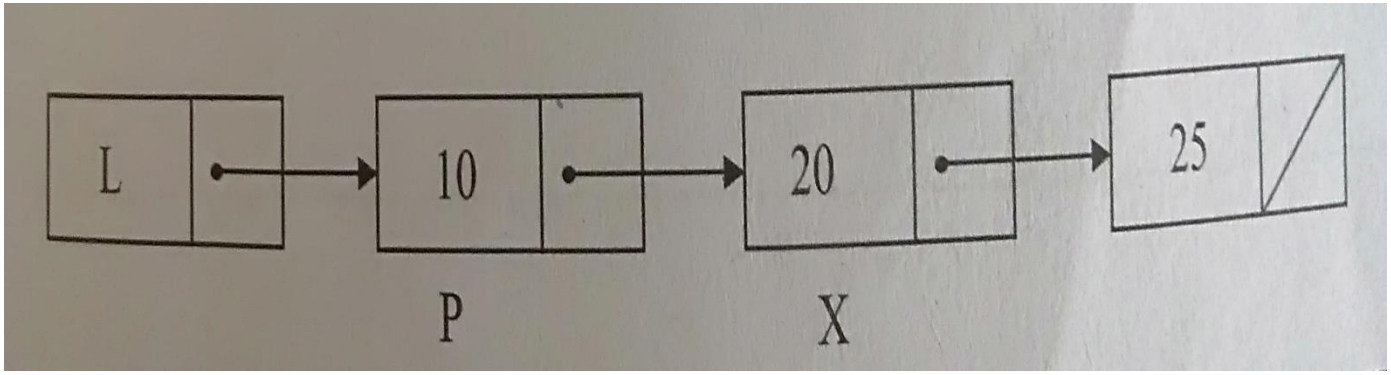
FindNext Algorithm

```
Position FindNext(int X, List L)
{
        Position P;
        P=L->Next;
        while(P->Next!=NULL && P->Element!=X)
                P=P->Next;
        return P->Next;
}
```



Find Previous Algorithm

```
Position FindPrevious(int X, List L)
{
        Position P;
        P=L;
        while(P->Next!=NULL && P->Next->Element!=X)
                P=P->Next;
        return P;
}
```

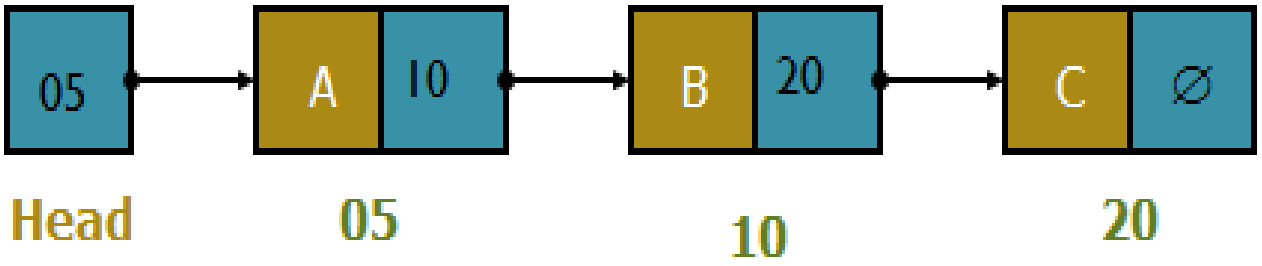


Deleting the Node

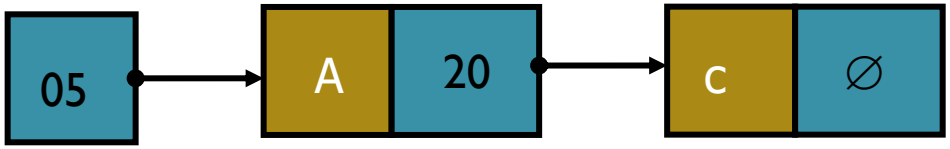
```
void Delete(int X, List L)
{
        Position=L->Next;
        PrePosition=Position;
        while(Position->Element!=X)
        {
                PrePosition=Position;
                Position=Position->Next;
        }
        Temp=Position;
        PrePosition->Next= Position->Next;
        Free Temp;
}
```

Example

- Before Deletion : X=B



- After Deletion:

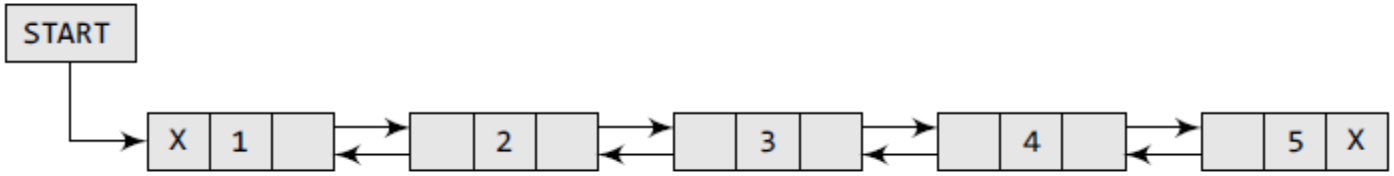


Merging Two SLL

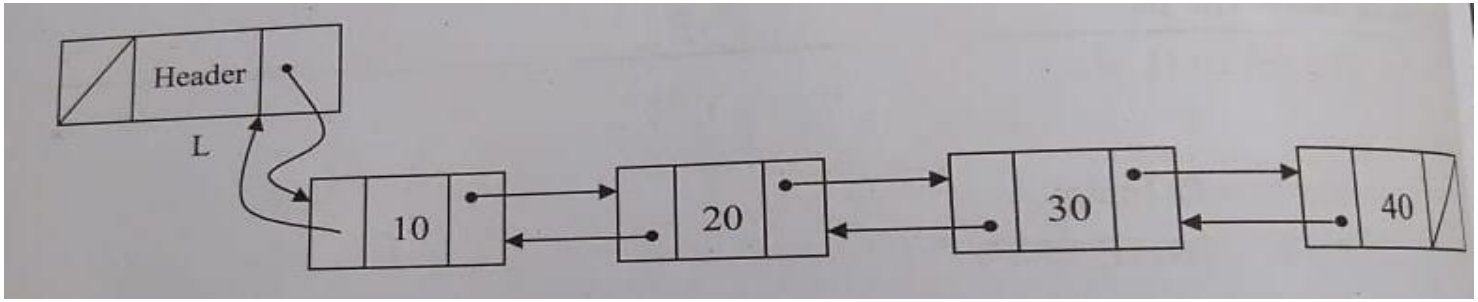
```
void MergeSLL( List L1,List L2)
{
        P=L1->Next;
        while(P->Next!=NULL)
        P=P->Next;
        P->Next=L2->Next;
        Free(L2->Next);
}
```

Doubly Linked Lists

- A doubly linked list or a two-way linked list is a more complex type of linked list.
- A DLL has Three fields namely
  - Data Field
  - Forward Link – Points to the successor node
  - Backward Link – Points to the predecessor node.



Doubly linked list



Doubly Linked Lists

- The BLINK field of the first node and the FLINK field of the last node will contain NULL.
- The BLINK field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.
- A DLL provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).
- The **main advantage** of using a doubly linked list is that it makes searching twice as efficient.

Structure Declartion

```
Struct Node
{
        int Element;
        Struct Node *FLINK;
        Struct Node *BLINK;
}
```

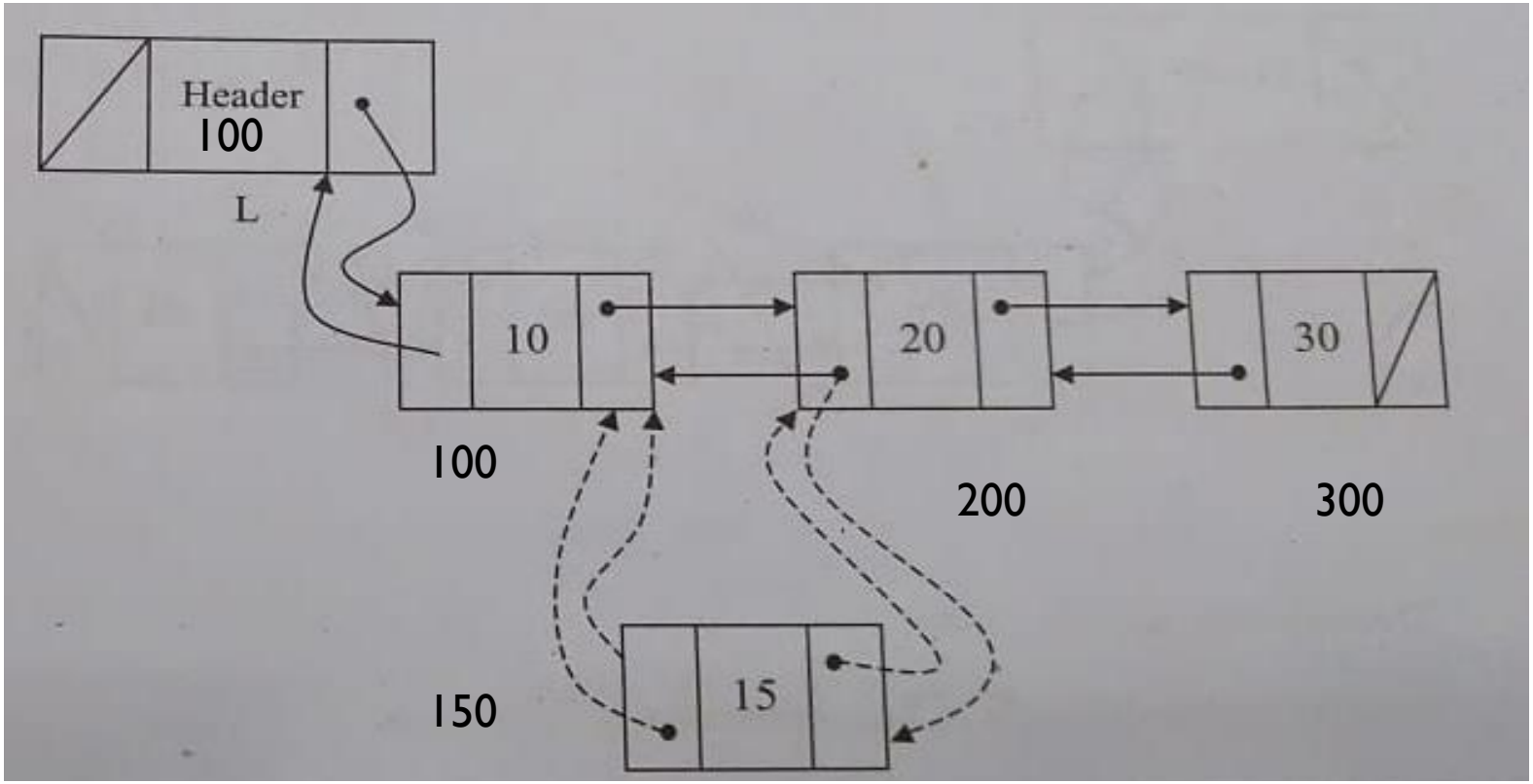
Operations on Doubly Linked List

- Inserting a New Node in a Doubly Linked List
  - Case 1: The new node is inserted at the beginning.
  - Case 2: The new node is inserted at the end.
  - Case 3: The new node is inserted after a given node.
  - Case 4: The new node is inserted before a given node.
- Deleting a New Node from a Doubly Linked List
  - Case 1: The first node is deleted.
  - Case 2: The last node is deleted.
  - Case 3: The node after a given node is deleted.
  - Case 4: The node before a given node is deleted.
  - Merging two linked List

Routine to Insert an Element in a DLL

```
void Insert(int X, List L, Position P)
{
        Struct Node *Newnode;
        Newnode=malloc(sizeof(Struct Node));
        if(Newnode!=NULL)
        {
                Newnode->Element=X;
                Newnode->FLINK=P->FLINK;
                P->FLINK->BLINK=Newnode;
                P->FLINK=Newnode;
                Newnode->BLINK=P;
        }
}
```

Example

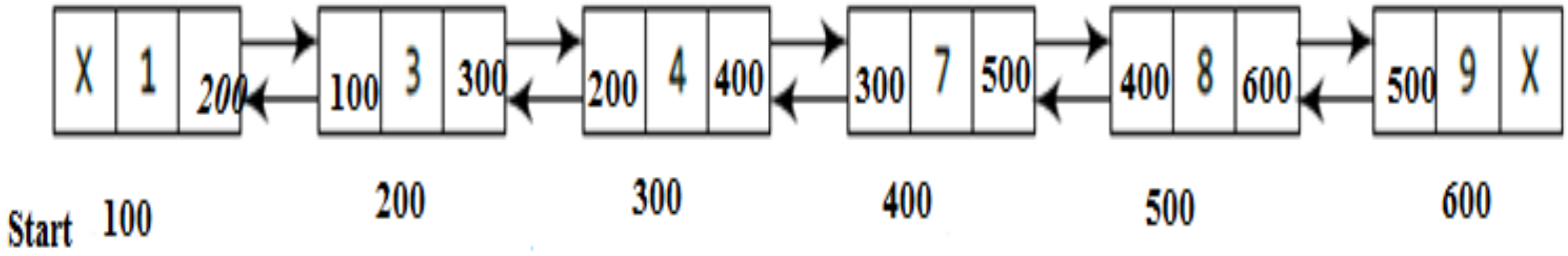


Routine to Delete an Element after a given node

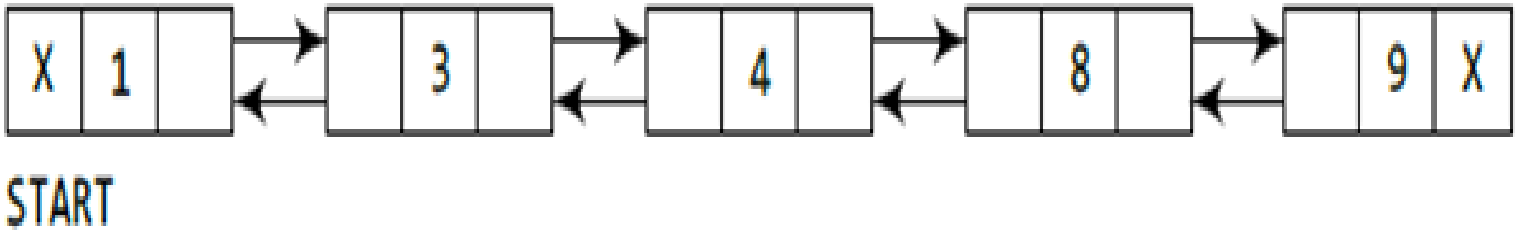
```
void Delete(int X, List L)
{
        Position P;
        P=L->Next;
         while (P->Element!= X)
                P=P->FLINK;
        P->BLINK->FLINK=P->FLINK;
        P->FLINK->BLINK=P->BLINK;
}
```

Example

- Before Deletion



- To Delete : 7
- After Deletion



Merging Two DLL

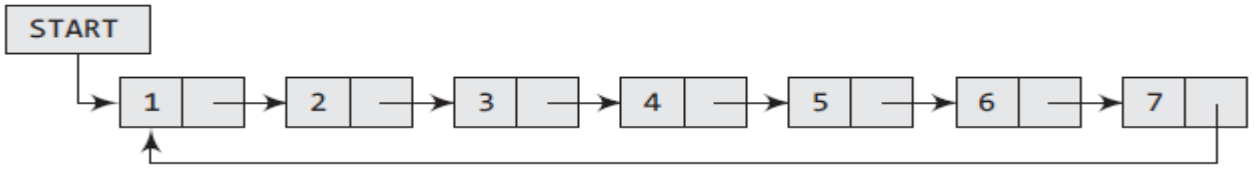
```
void MergeDLL( List L1,List L2)
{
        P1=L1->Next;
        P2=L2->Next;
        while(P1->FLINK!=NULL)
        P1=P1-> FLINK;
        P1-> FLINK =P2;
        P2->BLINK=P1
        Free(L2->Next);
}
```

Circular Linked List

- In a circular linked list, the last node contains a pointer to the first node of the list.
- CLL can be implemented with or without headers.
- Types:
  - Singly Linked Circular List
  - Doubly Linked Circular List
- A circular linked list has no beginning and no ending.
- Circular linked lists are widely used in operating systems for task maintenance.

Singly Linked Circular List

- A Singly Linked Circular List is a Linked List in which the last node of the list points to the first node.



Circular linked list

START
1

	DATA	NEXT
1	H	4
2		
3		
4	E	7
5		
6		
7	L	8
8	L	10
9		
10	0	1

Memory representation of a circular linked list

Operations on SLCL

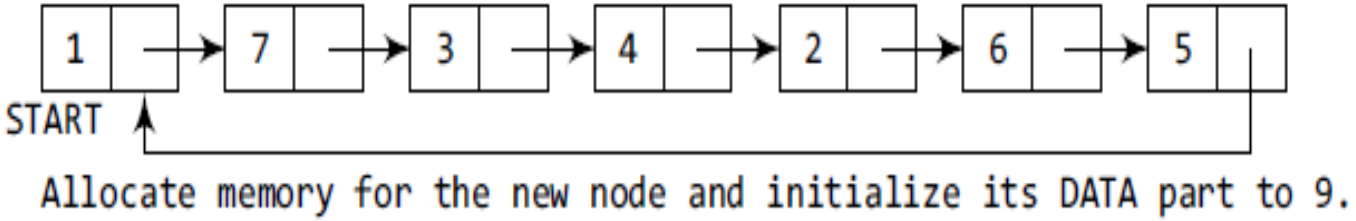
- Inserting a New Node in a Circular Linked List
  - Case 1: The new node is inserted at the beginning of the circular linked list.
  - Case 2: The new node is inserted at the end of the circular linked list.
- Deleting a Node from a Circular Linked List
  - Case 1: The first node is deleted.
  - Case 2: The last node is deleted.

Insertion at First

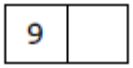
```
void Insertion(List L, int X)
{
        Newnode=malloc(sizeof(Struct node));
        if(Newnode!=NULL)
        {
                Newnode->Element=X;
                P=L->Next;
                while(P->Next!=P)
                        P=P->Next;
                P->Next=Newnode;
                Newnode->Next=L->Next;
                L->Next=Newnode;
        }
}
```

Example

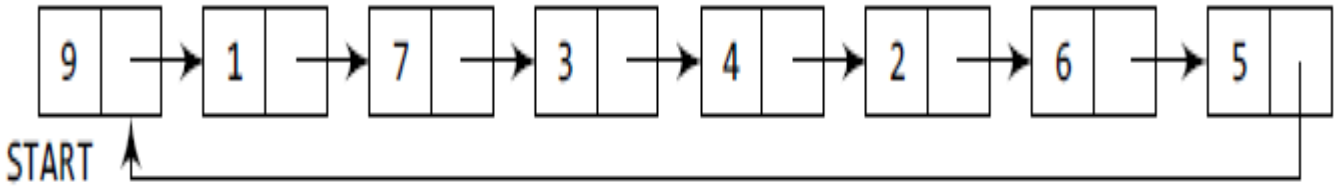
- Before Insertion



- To insert at first



- After Insertion

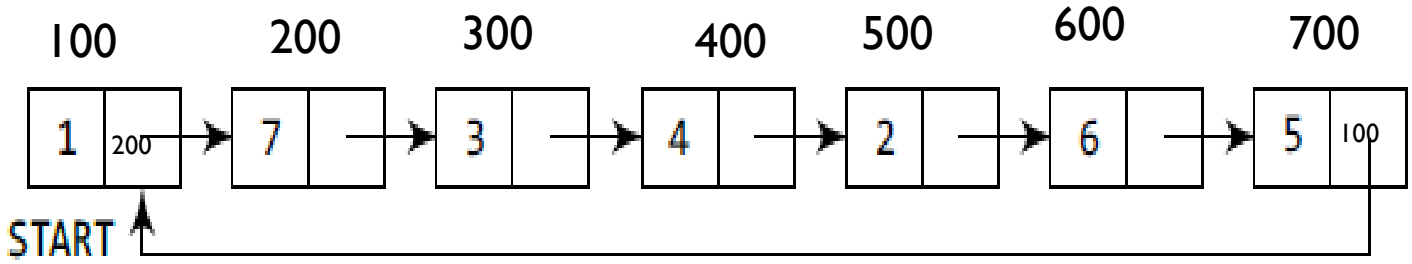


Deletion at First

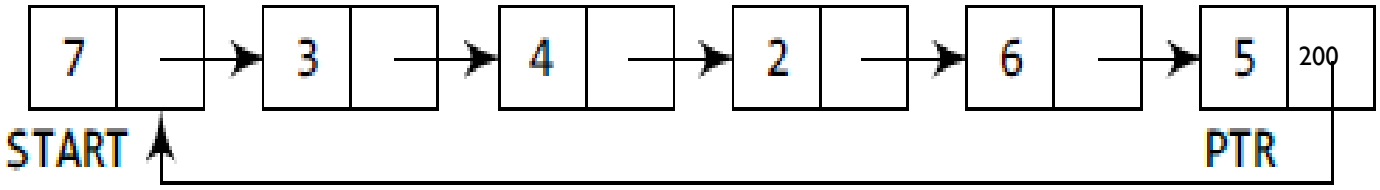
```
void Delete(List L)
{
        P=L->Next;
        START=P;
        Temp=P->Next;
        while(P->Next!=START)
                P=P->Next;
        P->Next=Temp;
        L->Next=Temp;
        Free(Temp);
        Free(START);
}
```

Example

- Before Deletion

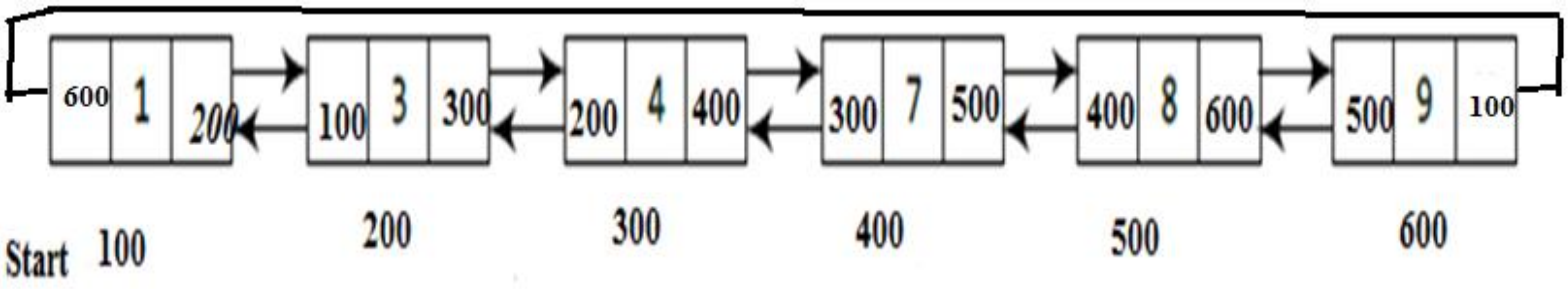


- After Deletion



Circular Doubly Linked List

- A circular doubly linked list or a circular two-way linked list which contains a pointer to the next as well as the previous node in the sequence.
- The circular doubly linked list does not contain NULL in the previous field of the first node and the next field of the last node.
- Rather, the next field of the last node stores the address of the first node of the list, i.e., START.
- Similarly, the previous field of the first field stores the address of the last node.



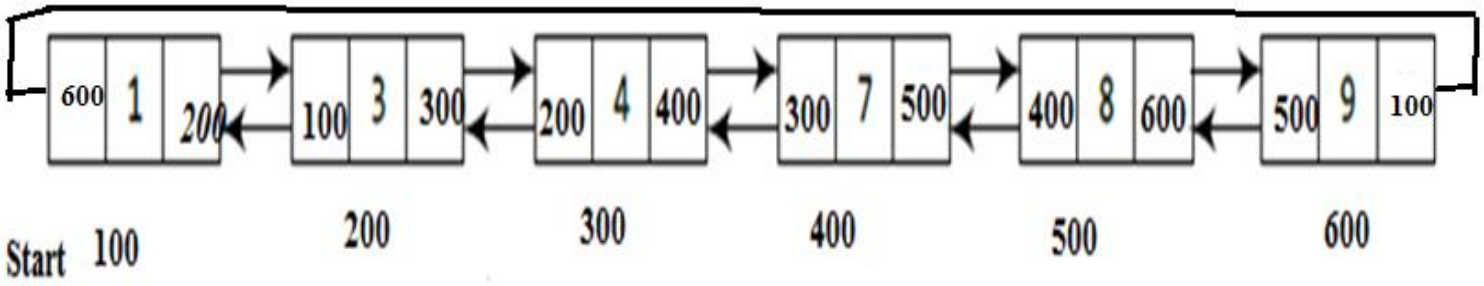
Insertion at First

```
Step 1: IF AVAIL = NULL
            Write OVERFLOW
            Go to Step 13
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 8: SET PTR -> NEXT = NEW_NODE
Step 9: SET NEW_NODE -> PREV = PTR
Step 10: SET NEW_NODE -> NEXT = START
Step 11: SET START -> PREV = NEW_NODE
Step 12: SET START = NEW_NODE
Step 13: EXIT
```

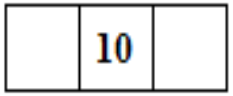
Algorithm to insert a new node at the beginning

Example

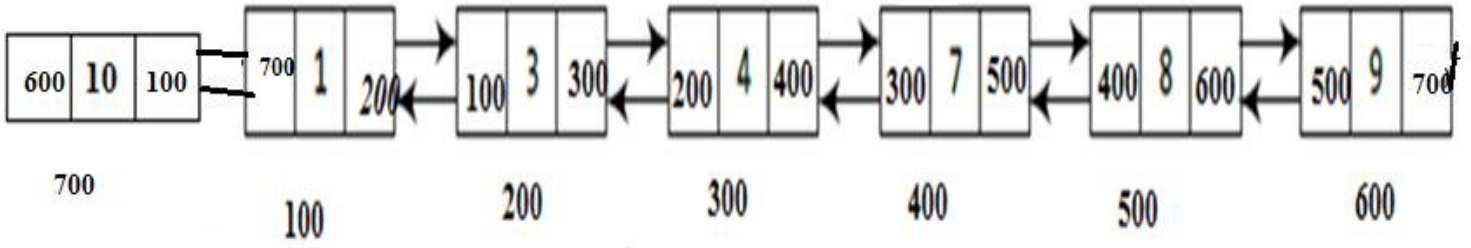
- Before Insertion



- To Insert : Address: 700



- After Insertion



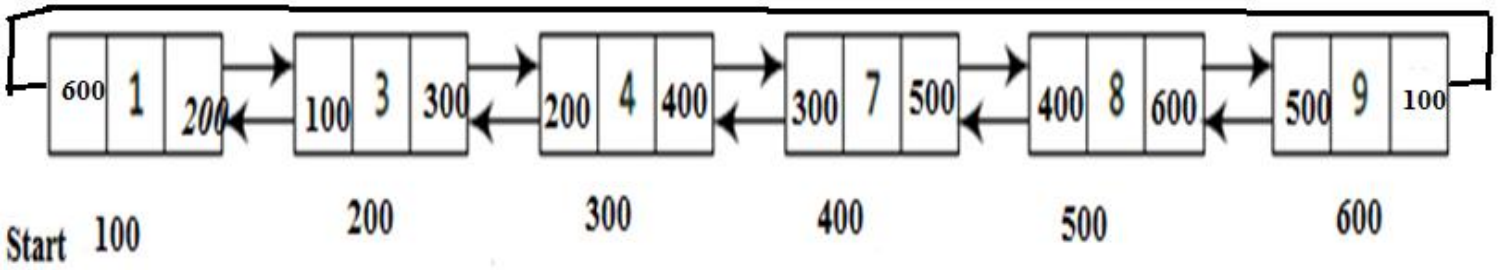
Deletion at Last

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 8
        [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4  while PTR -> NEXT != START
Step 4:     SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> PREV -> NEXT = START
Step 6: SET START -> PREV = PTR -> PREV
Step 7: FREE PTR
Step 8: EXIT
```

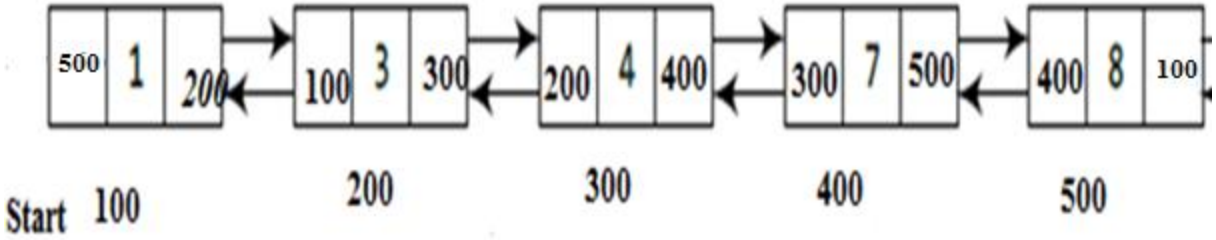
Algorithm to delete the last node

Example

- Before Deletion



- To Delete: 9
- After Deletion



Applications of Linked List

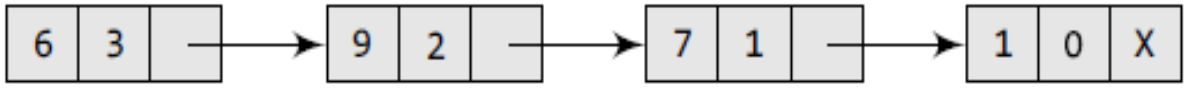
- Polynomial ADT
- Radix Sort
- Multilist

Polynomial ADT

- Linked lists can be used to represent polynomials and the different operations that can be performed on them.

Polynomial Representation

- Every individual term in a polynomial consists of two parts, a coefficient and a power.
- Consider a polynomial $6x^3 + 9x^2 + 7x + 1$.
- Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively.
- Every term of a polynomial can be represented as a node of the linked list.



Linked representation of a polynomial

Structure Declaration for Polynomial ADT

```
Struct Poly
{
        int coeff;
        int power;
        Struct Poly *Next;
}*list1,*list2,*list3;
```

Creation of the Polynomial

```
poly create(poly *head1,poly *newnode1)
{
        poly *ptr;
        if(head1==NULL)
        {
                head1=newnode1;
                return(head1);
        }
        else
        {
                ptr=head1;
                while(ptr→next!=NULL)
                ptr=ptr→next;
                ptr→next=newnode1;
        }
        return head1;
}
```

Addition of Two Polynomial

```
Void add()
{
        poly  *ptr1,*ptr2,*newnode;
        ptr1=list1;
        ptr2=list2;
        while(ptr1!=NULL && ptr2!=NULL)
        {
                newnode=malloc(sizeof(Struc poly));
        if(ptr1→power==ptr2→power)
        {
                newnode→coeff=ptr1→coeff+ptr2→coeff;
                newnode→power=ptr1→power;
                newnode→next=NULL:
                list3=create(list3,newnode);
                ptr1=ptr1→next;
                ptr2=ptr2→next;
        }
```

Addition of Two Polynomial (Cont.,)

```
else
{
        if(ptr1→power>ptr2→power)
        {
                newnode→coeff=ptr1→coeff;
                newnode→power=ptr1→power;
                newnode→next=NULL:
                list3=create(list3,newnode);
                ptr1=ptr1→next;
        }
```

Addition of Two Polynomial (Cont.,)

```
else
{
                newnode→coeff=ptr2→coeff;
                newnode→power=ptr2→power;
                newnode→next=NULL:
                list3=create(list3,newnode);
                ptr2=ptr2→next;
}
}
}
```

Example

ptr1→$6x^3 + 9x^2 + 7x + 1$.

ptr2→$3x^3 + 6x^2 + 2x + 8$.

ptr3→$9x^3 + 15x^2 + 9x + 9$.

Part B

1. Explain about classification of DS.
2. Write short notes on List ADT.
3. How can you insert an element after a position in a SLL? Explain with suitable example and algorithm.
4. Define DLL. Write the operations performed on that.
5. Write the operations performed on Doubly Linked Circular List.
6. Write the algorithm to insert a new node at First into the Singly Linked Circular List. Explain with an example.

How to answer Part B and Part C Questions?

- Definition - Should be very Clear
- Algorithm
- Example – Should comprise like Before insertion, To insert, After Insertion
- Neat Diagram Representation (If necessary)