Lecture 3

Searching

- Searching is the process of finding a given value in a list of values.
- It is the algorithmic process of finding a particular item in a collection of items.

(i) Linear Search:

Definition:

- It starts at the beginning of the list and checks every element of the list.
- i.e. It sequentially checks each element of the list until a match is found or the whole list has been searched. So it is also called sequential search.

Example:

- Let the elements are: 10,6,3,8,9,12,14
- The search element is : 12
- Now it compare 12 with each and every element.
- The 12 is available in 6^{th} place.
- So the searching process is success and element is found

Algorithm:

Step 1: Read elements in array
Step 2: Read the element to search
Step 3: Compare the element to sear and each element in array sequentially
Step 4: If match is found then the search success
Step 5: If match is not found upto the end then the search un success

Program:

```
#include <stdio.h>
int main()
{
        int a[100],n,i,s;
        printf("Enter Number of Elements in Array:\n");
        scanf("%d", &n);
        printf("Enter numbers:\n");
        for(i = 0; i < n; i++)
        scanf("%d",&a[i]);
        printf("Enter a number to search in Array:\n");
        scanf("%d", &s);
        for(i = 0; i < n; i++)
        {
                if(s==a[i])
                {
                        printf("Number found\n”);
                        break;
                }
        }
        if(i== n)
        printf("Number not found\n”);
        return 0;
}
```

Binary Search

- Binary search is the most popular Search algorithm. It is efficient and also one of the most commonly used techniques that are used to solve problems.
- Binary search sorts the records either in ascending or descending order to gain much better performance than linear search.
- Now suppose we have an ascending order record. At the time of search it takes the middle record/element, if the searching element is greater than middle element then the element mush be located in the second part

  else it is in the first half. In this way this search algorithm divides the records in the two parts in each iteration and thus called binary search.

Example:

- Let the elements in ascending order are
  2 4 6 8 10 12 15
- Let the element to search 12
- For searching it compare first middle element.
  2 4 6 8 10 12 15
- The middle element is 8 and is not equal to 12. Since 12 is greater than 8 search on right side part of 8. 12 is equal to right side part middle. So element is found.

Example:

- Let the elements in ascending order are
  2 4 6 8 10 12 15
- Let the element to search 12
- For searching it compare first middle element.
  2 4 6 8 10 12 15
- The middle element is 8 and is not equal to 12. Since 12 is greater than 8 search on right side part of 8. 12 is equal to right side part middle. So element is found.

Algorithm:

Step 1: Read sorted elements in array
Step 2: Read the element to search
Step 3: Compare the element to sear and middle element in array. If match is found the search success.
Step 4: If match is not found check the search element with middle element. If search element is greater than the middle element then search on right side of middle element otherwise search on left.
Step 5: This process is repeated for all elements in array. If no match is found upto the end then the search is not success.

Program:

```
#include <stdio.h>
int main()
{
        int i, first, last, middle, n, s, a[100];
        printf("Enter number of elements:\n");
        scanf("%d",&n);
        printf("Enter elements in ascending order:\n");
        for (i = 0; i < n; i++)
        scanf("%d",&a[i]);
        printf("Enter an element to search:\n");
        scanf("%d", &s);
        first = 0;
        last = n - 1;
        middle = (first+last)/2;
        while (first <= last)
        {
                if(s==a[middle])
                {
                        printf("Element is found at index: %d",middle);
                        break;
                }
                else if(s>a[middle])
                first = middle + 1;
                else if(s<a[middle])
                last = middle - 1;
                middle = (first + last)/2;
        }
        if (first > last)
        printf("Element is not found");
        return 0;
}
```

Sorting

- Sorting is a process of placing a list of elements from the collection of data in some order.
- It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order. It arranges the data in a sequence which makes searching easier

(i) Insertion Sort:

- In this sorting technique first elements are stored in an array.
- The process of sorting starts with second element.
- First the second element is picked and is placed in specified order Next third element is picked and is placed in specified order. Similarly the fourth, fifth, …n^{th} element .is placed in specified order.
- Finally we get the sorting elements.

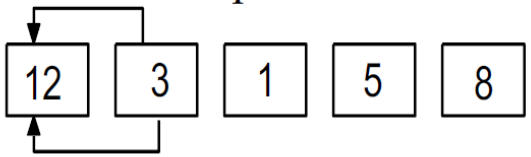
Algorithm:

Step 1: Check second element of array with element before it and insert it in proper position.
Step 2: Checking third element of array with element before it and inserting it in proper position.
Step 3: Repeat this till all elements are checked.
Step 4: Stop

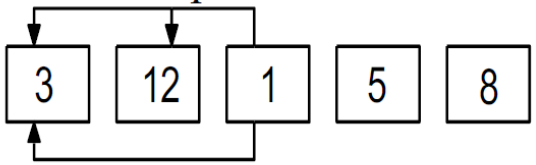
Example:

- Let us consider the elements: 12, 3, 1, 5, 8

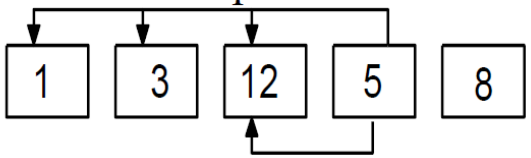
1. Checking second element of array with element before it and inserting it in proper position. In this case 3 is inserted in position of 12



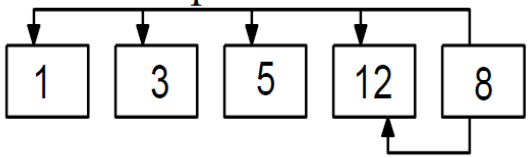
2. Checking third element of array with elements before it and inserting it in proper position. In this case 1 is inserted in position of 3



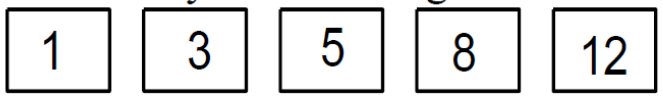
3. Checking fourth element of array with elements before it and inserting it in proper position. In this case 5 is inserted in position of 12



4. Checking fifth element of array with elements before it and inserting it in proper position. In this case 8 is inserted in position of 12



5. Sorted array in ascending order



Program:

```
#include<stdio.h>
int main()
{
        int n,a[30],key,i,j;
        printf("Enter total elements:\n");
        scanf("%d",&n);
        printf("Enter elements:\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        for(i=1;i<n;i++)
        {
                j=i;
                while(j>0 && a[j]<a[j-1])
                {
                        temp=a[j];
                        a[j]=a[j-1];
                        a[j-1]=temp;
                        j--;
                }
        }
        printf("After sorting is:\n");
        for(i=0;i<n;i++)
        printf(" %d",a[i]);
        return 0;
}
```

Selection Sort

- Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Example:

- The following figure shows the first pass of a selection sort.

First pass →

54	**26**	93	17	77	31	44	10	5	assign 54 min
26	54	**93**	17	77	31	44	10	5	assign 26 min
26	54	93	**17**	77	31	44	10	5	assign 26 min
17	54	93	26	**77**	31	44	10	5	assign 17 min
17	54	93	26	77	**31**	44	10	5	assign 17 min
17	54	93	26	77	31	**44**	10	5	assign 17 min
17	54	93	26	77	31	44	**10**	5	assign 17 min
10	54	93	26	77	31	44	17	**5**	assign 10 min
5	54	93	26	77	31	44	17	10	Exchange 10 and 5 after first pass

Algorithm:

Step 1 – Set min to the first location.
Step 2 – Search the minimum element in the array.
Step 3 – swap the first location with the minimum value in the array.
Step 4 – assign the second element as min.
Step 5 – Repeat the process until we get a **sorted** array.

- In first pass the first element is compared with all remaining elements and exchange element if first one is greater than second so that the smallest value is in first place. Leave this element.
- In second pass compare second element to all elements and put the next smallest value, in second place. Leave this element. This process is repeated till all the elements are placed.
- Now we get the sorted elements.

Bubble Sort

Example:

- The following figure shows the first pass of a bubble sort. In first pass the first element is compared with second and exchange element if first one is greater than second.
- Similarly second element is compared with third and exchange element if second one is greater than third.
- Repeat this so that at the end of first pass the largest value is in last place. Leave this element.

First pass →

54	**26**	93	17	77	31	44	55	20	Exchange
26	**54**	**93**	17	77	31	44	55	20	No Exchange
26	54	**93**	**17**	77	31	44	55	20	Exchange
26	54	17	**93**	**77**	31	44	55	20	Exchange
26	54	17	77	**93**	**31**	44	55	20	Exchange
26	54	17	77	31	**93**	**44**	55	20	Exchange
26	54	17	77	31	44	**93**	**55**	20	Exchange
26	54	17	77	31	44	55	**93**	**20**	Exchange
26	54	17	77	31	44	55	20	93	93 in place after first pass

- In second pass compare up to before last place value and put the next largest value, that before last place. Leave this element.

Algorithm:

Step 1: The first element is compared with second and exchange element if first one is greater than second
Step 2: Similarly second element is compared with third and exchange element if second one is greater than third
Step 3: Repeat this so that at the end the largest value is in last place
Step 4: Likewise sorting is repeated for all elements.

Program:

```
#include<stdio.h>
int main()
{
        int n,temp,i,j,a[20];
        printf("Enter total numbers of elements:\n");
        scanf("%d",&n);
        printf("Enter elements:\n");
        for(i=0;i<n;i++)
        scanf("%d",&a[i]);
        for(i=0;i<n;i++)
        {
              for(j=0;j<n-1;j++)
              {
                    if(a[j]>a[j+1])
                    {
                          temp=a[j];
                          a[j]=a[j+1];
                          a[j+1]=temp;
                    }
              }
        }
        printf("After sorting elements are:\n");
        for(i=0;i<n;i++)
        printf(" %d",a[i]);
        return 0;
}
```

Linked List

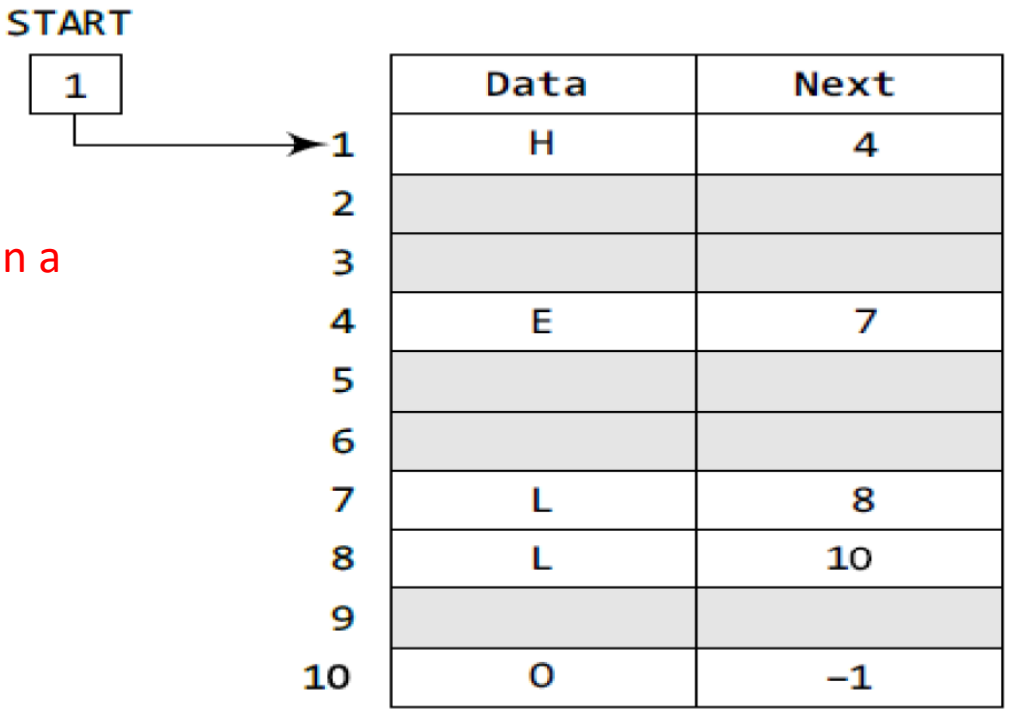
A linked list is a collection of data elements called nodes in which the linear representation is given by links from one node to the next node. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.

The elements in a linked list can be accessed only in a sequential manner. But like an array, insertions and deletions can be done at any point in the list in a constant time.

A linked list, in simple terms, is a linear collection of data elements. These data elements are called nodes. Linked list is a data structure which in turn can be used to implement other data structures.

Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations.

Storing the word “HELLO” in a Linked list



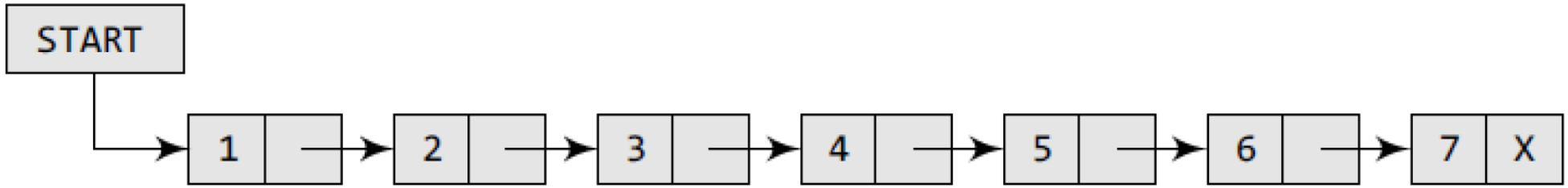
Linked Lists versus Arrays:

Both arrays and linked lists are a linear collection of data elements. But unlike an array, a linked list does not store its nodes in consecutive memory locations. Another point of difference between an array and a linked list is that a linked list does not allow random access of data. Nodes in a linked list can be accessed only in a sequential manner.

Another advantage of a linked list over an array is that we can add any number of elements in the list. This is not possible in case of an array.

Single Linked List

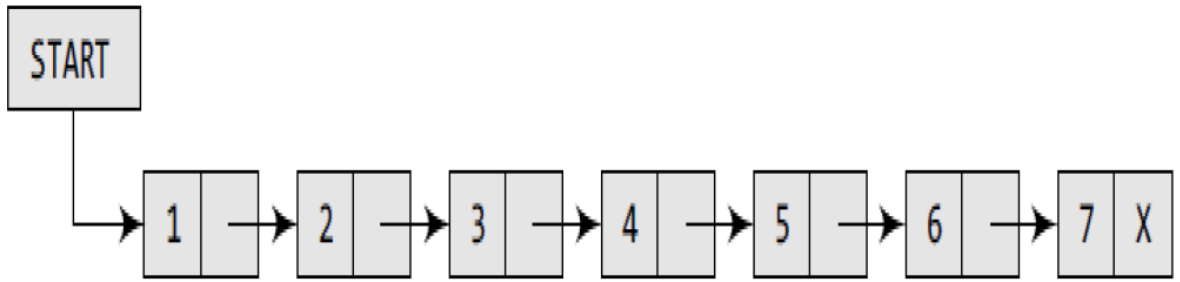
A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.



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.

Traversing a Linked List:

For traversing the linked list, we also make use of another pointer variable PTR which points to the node that is currently being accessed. Algorithm for traversing a linked list

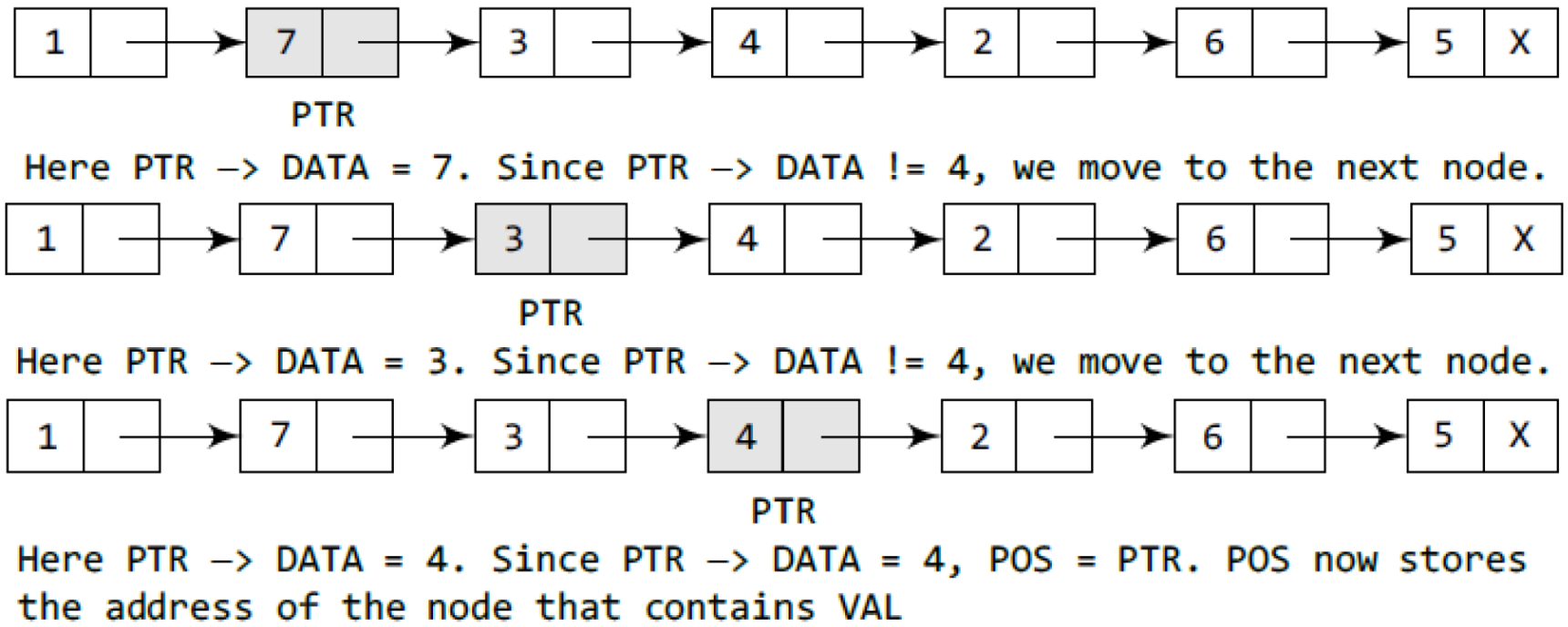



```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3:          Apply Process to PTR->DATA
Step 4:          SET PTR = PTR->NEXT
      [END OF LOOP]
Step 5: EXIT
```

Searching for a Value in a Linked List:

Searching a linked list means to find a particular element in the linked list. So searching means finding whether a given value is present in the information part of the node or not. If it is present, the algorithm returns the address of the node that contains the value. However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

Consider the linked list shown in below. If we have VAL = 4, then the flow of the algorithm can be explained as shown in the figure.

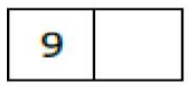


Case 1: Inserting a Node at the Beginning of a Linked List

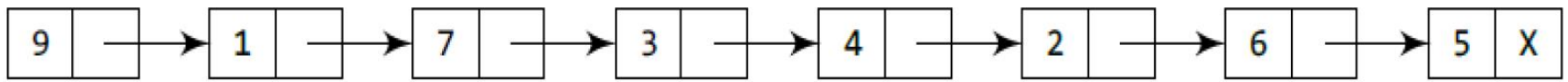
Inserting a Node at the Beginning of a Linked List. Consider the linked list shown in below figure Suppose we want to add a new node with data 9 and add it as the first node of the list.

START

Allocate memory for the new node and initialize its DATA part to 9.

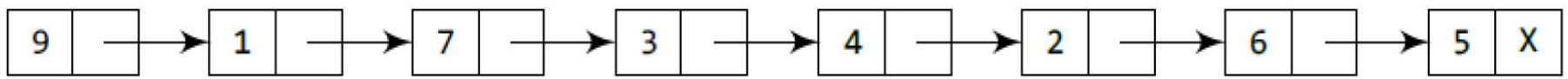


Add the new node as the first node of the list by making the NEXT part of the new node contain the address of START.



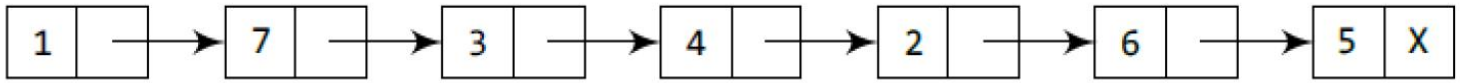
START

Now make START to point to the first node of the list.



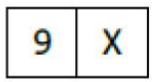
START

Case 2: Inserting a Node at the End of a Linked List

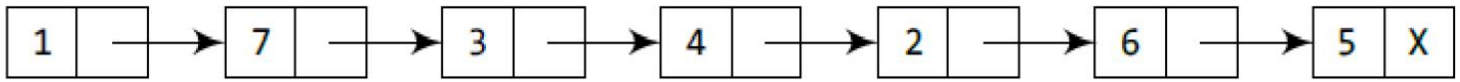


START

Allocate memory for the new node and initialize its DATA part to 9 and NEXT part to NULL.

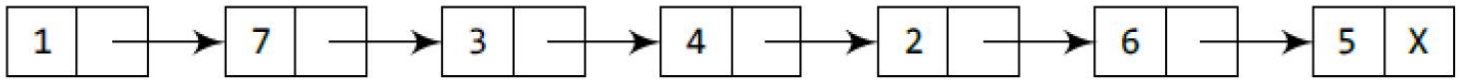


Take a pointer variable PTR which points to START.



START, PTR

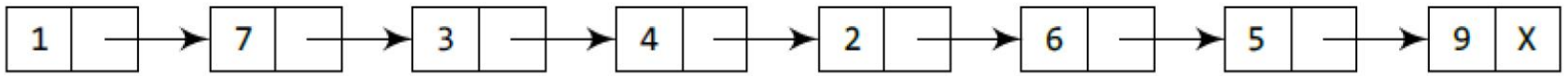
Move PTR so that it points to the last node of the list.



START

PTR

Add the new node after the node pointed by PTR. This is done by storing the address of the new node in the NEXT part of PTR.

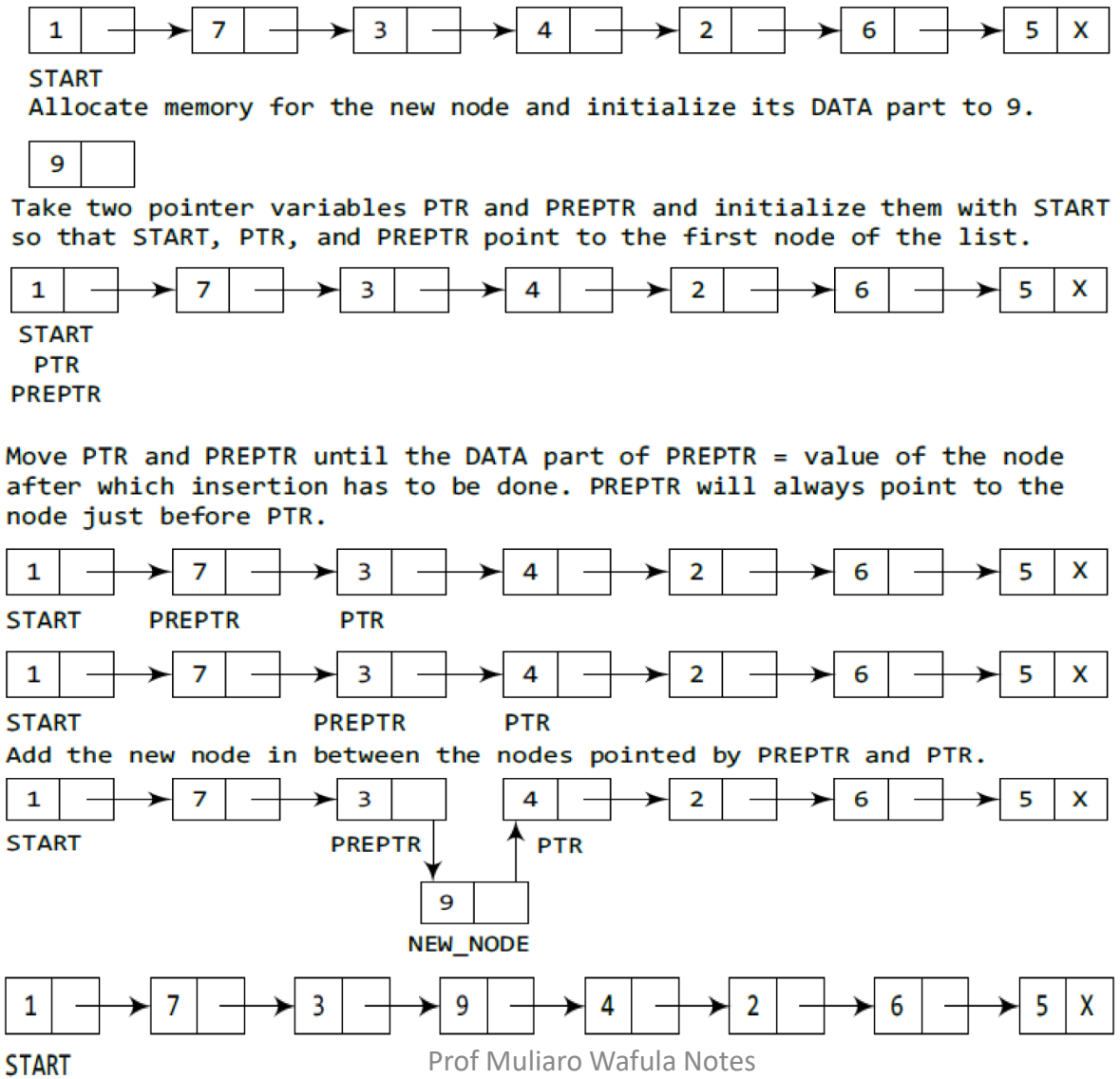


START

PTR

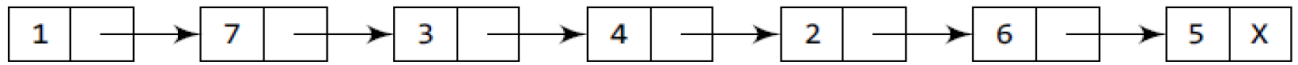
Case 3: Inserting a Node After a Given Node in a Linked List

Consider the linked list shown in below figure. Suppose we want to add a new node with value 9 after the node containing 3.



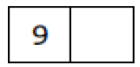
Case 4: Inserting a Node Before a Given Node in a Linked List

Consider the linked list shown in below figure. Suppose we want to add a new node with value 9 before the node containing 3.

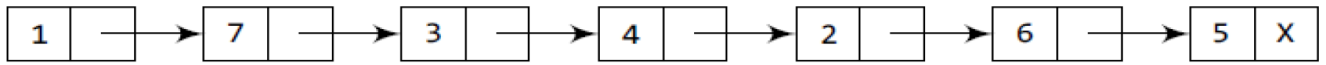


START

Allocate memory for the new node and initialize its DATA part to 9.

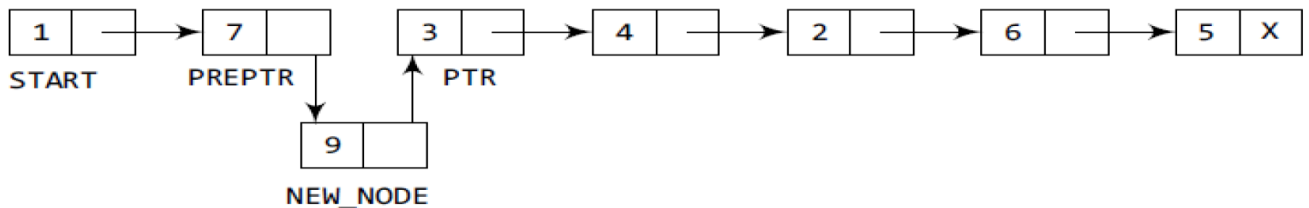


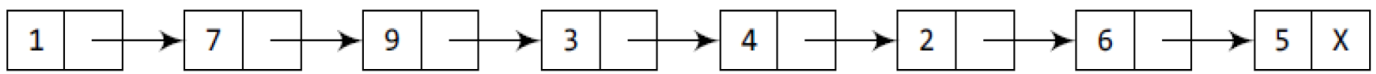
Initialize PREPTR and PTR to the START node.



START
PTR
PREPTR

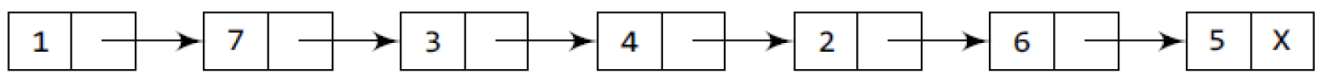
Insert the new node in between the nodes pointed by PREPTR and PTR.





START

Move PTR and PREPTR until the DATA part of PTR = value of the node before which insertion has to be done. PREPTR will always point to the node just before PTR.



START PREPTR PTR

Deleting a Node from a Linked List:

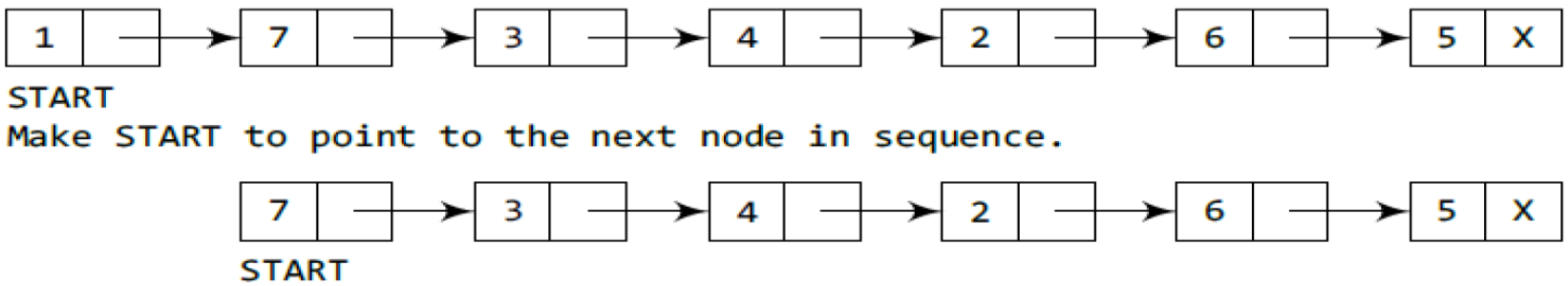
We will discuss how a node is deleted from an already existing linked list. We will consider three cases and then see how deletion is done in each case.

Case 1: The first node is deleted.

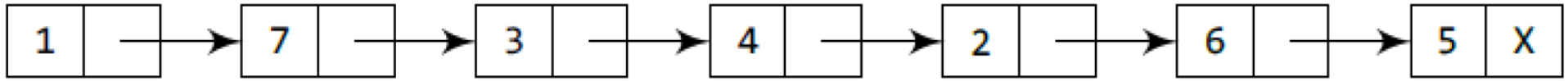
Case 2: The last node is deleted.

Case 3: The node after a given node is deleted.

Case 1: Deleting a First Node from a Linked List

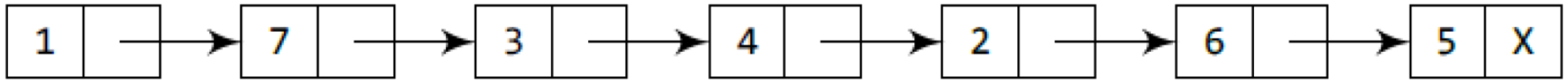


Case 2: Deleting the Last Node from a Linked List



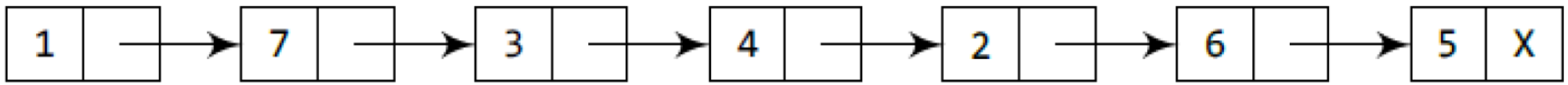
START

Take pointer variables PTR and PREPTR which initially point to START.



START
PREPTR
PTR

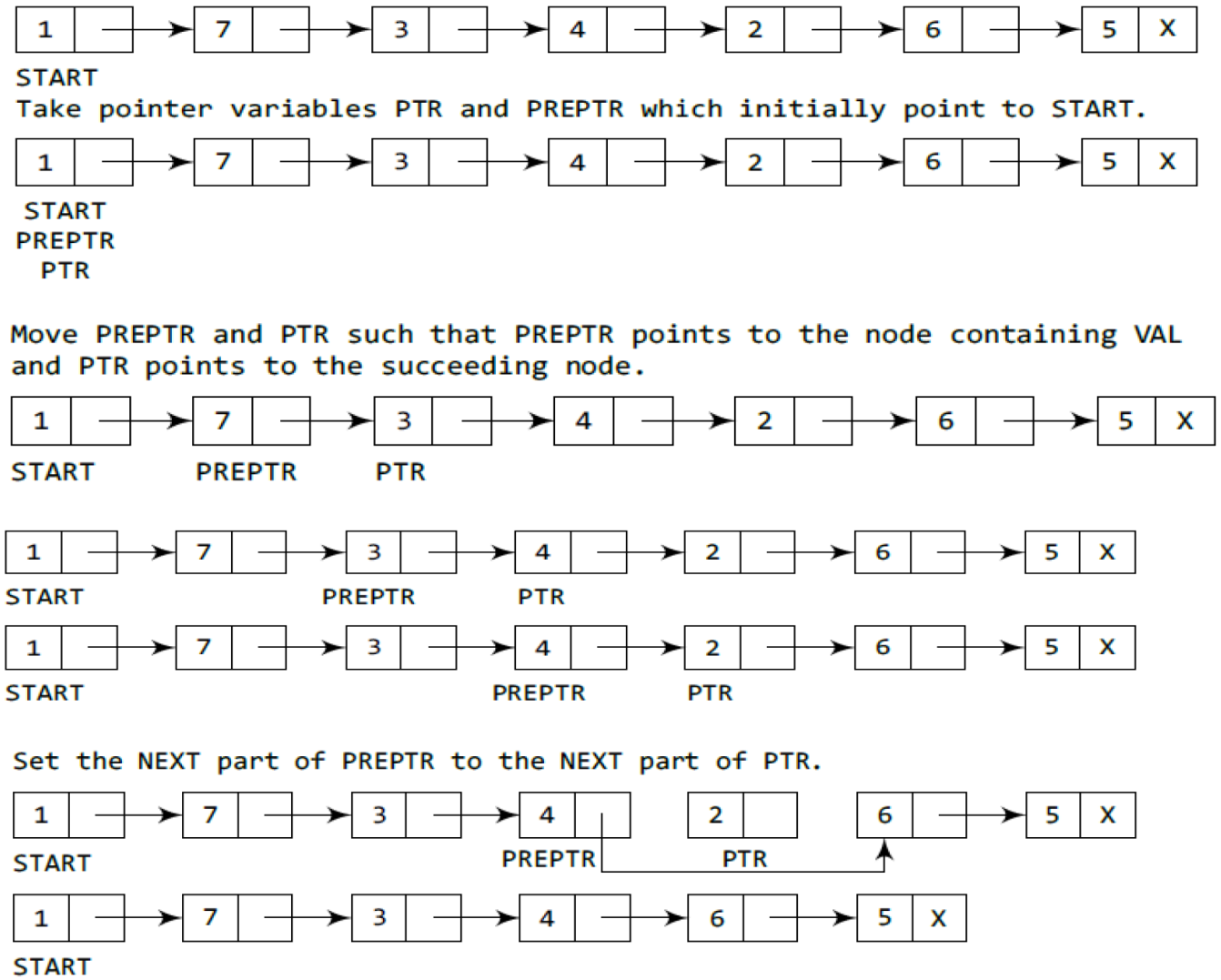
Move PTR and PREPTR such that NEXT part of PTR = NULL. PREPTR always points to the node just before the node pointed by PTR.



START PREPTR PTR

Case 3: Deleting After a Given Node in a Linked List

Consider the linked list shown in below figure. Suppose we want to delete the node that succeeds the node which contains data value 4.



Application of Linked Lists

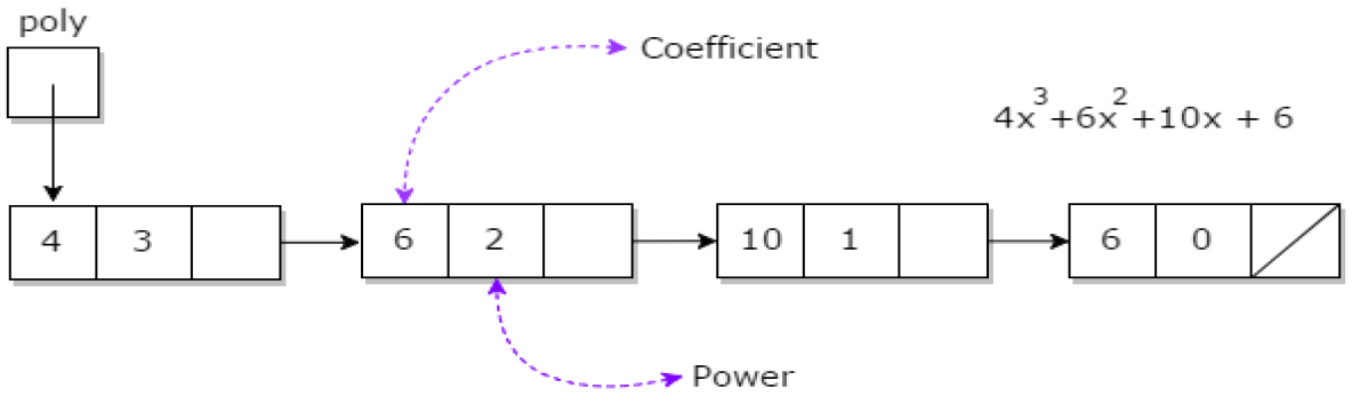
- ✓ Implementation of stacks and queues.
- ✓ Implementation of graphs: Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.
- ✓ Dynamic memory allocation: We use linked list of free blocks.
- ✓ Maintaining directory of names. Performing arithmetic operations on long integers
- ✓ Manipulation of polynomials by storing constants in the node of linked list. Representing sparse matrices

Polynomial Expression Representation:

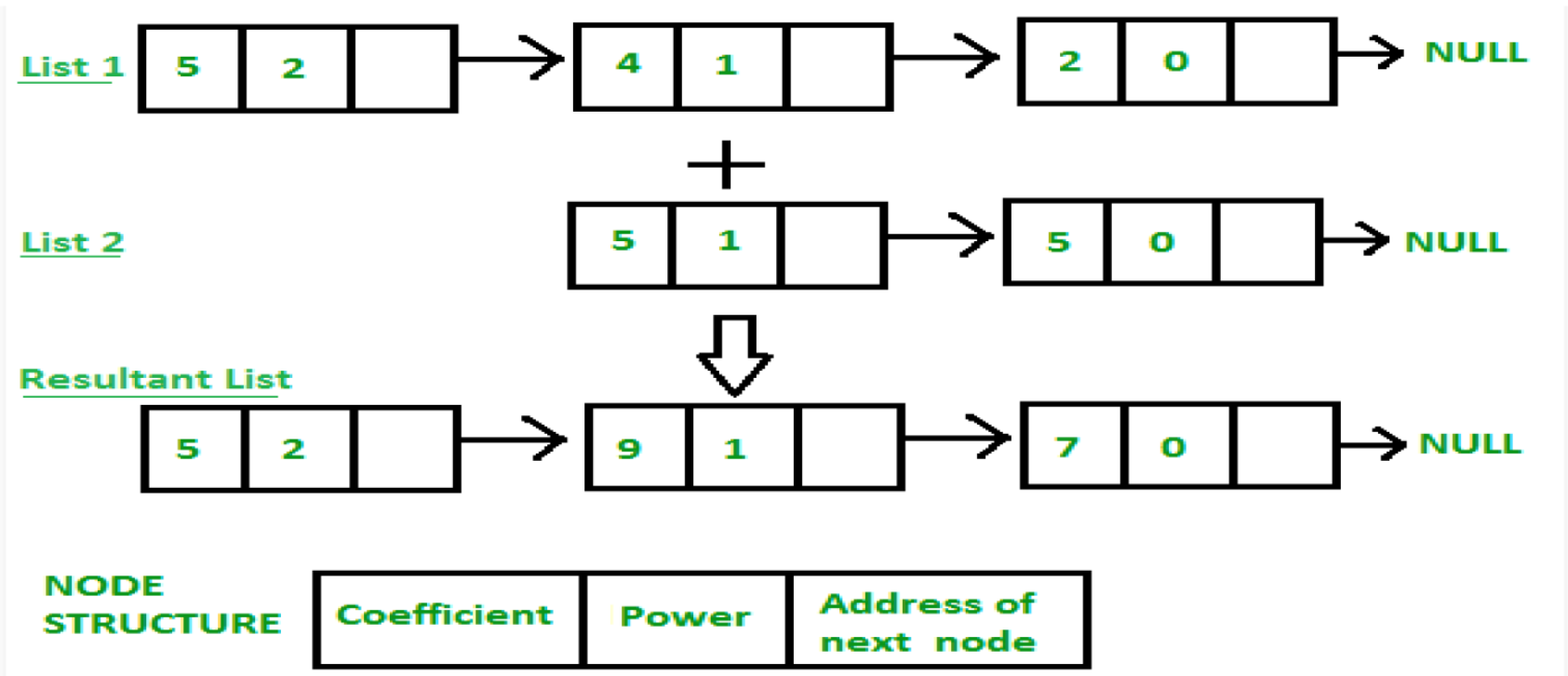
A polynomial is composed of different terms where each of them holds a coefficient and an exponent. An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts: one is the coefficient and other is the exponent

Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 are its exponential values.



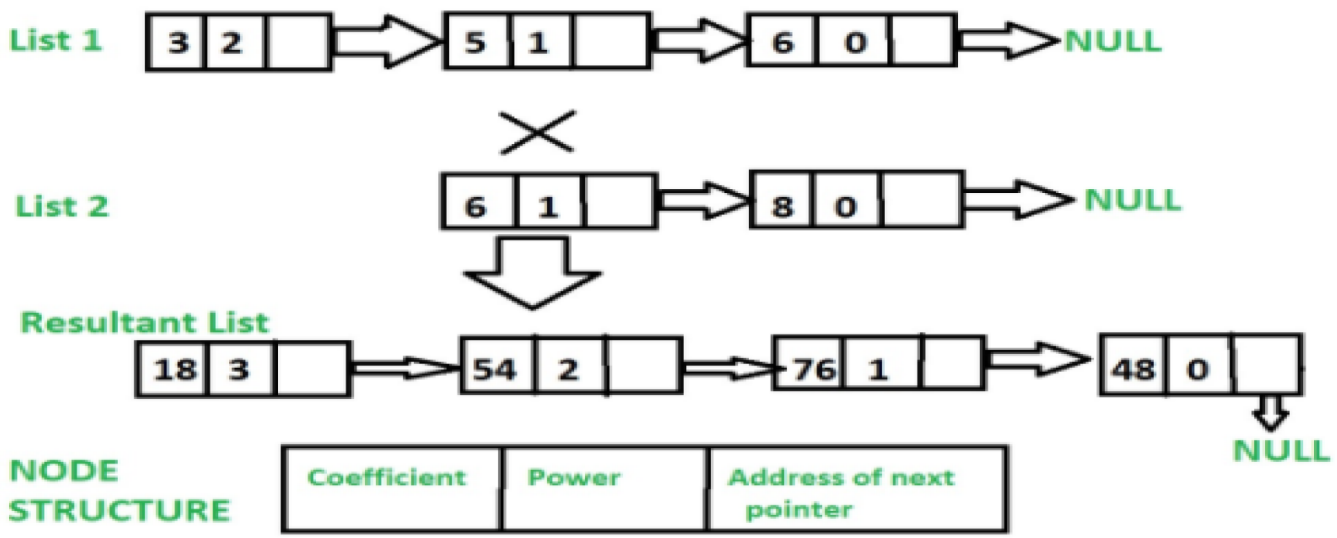
```
Input:
      1st number = 5x^2 + 4x^1 + 2x^0
      2nd number = 5x^1 + 5x^0
Output:
         5x^2 + 9x^1 + 7x^0
Input:
      1st number = 5x^3 + 4x^2 + 2x^0
      2nd number = 5x^1 + 5x^0
Output:
         5x^3 + 4x^2 + 5x^1 + 7x^0
```



Multiplication of two polynomials:

```
Input: Poly1: 3x^2 + 5x^1 + 6, Poly2: 6x^1 + 8
Output: 18x^3 + 54x^2 + 76x^1 + 48
On multiplying each element of 1st polynomial with
elements of 2nd polynomial, we get
18x^3 + 24x^2 + 30x^2 + 40x^1 + 36x^1 + 48
On adding values with same power of x,
18x^3 + 54x^2 + 76x^1 + 48

Input: Poly1: 3x^3 + 6x^1 + 9, Poly2: 9x^3 + 8x^2 + 7x^1 + 2
Output: 27x^6 + 24x^5 + 75x^4 + 135x^3 + 114x^2 + 75x^1 + 18
```



Example:

$$(2x^2 + 3x + 1) \times (x^2 - x + 4)$$

- Multiply each term in the first polynomial by each term in the second:

$$(2x^2) \times (x^2 - x + 4) = 2x^4 - 2x^3 + 8x^2$$

$$(3x) \times (x^2 - x + 4) = 3x^3 - 3x^2 + 12x$$

$$(1) \times (x^2 - x + 4) = x^2 - x + 4$$

- Now, combine all terms:

$$2x^4 - 2x^3 + 8x^2 + 3x^3 - 3x^2 + 12x + x^2 - x + 4$$

- Combine like terms:

$$2x^4 + (-2x^3 + 3x^3) + (8x^2 - 3x^2 + x^2) + (12x - x) + 4$$

$$2x^4 + x^3 + 6x^2 + 11x + 4$$

Sparse Matrix Representation using Linked List:

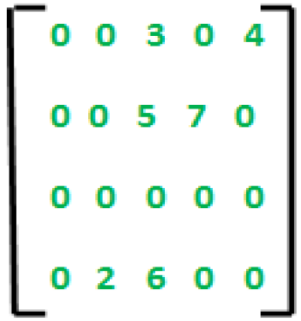
A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have 0 value, then it is called a sparse matrix.

Example:
0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0

Sparse Matrix Representation using arrays:

Method 1: Using Arrays

2D array is used to represent a sparse matrix in which there are three rows named as
Row: Index of row, where non-zero element is located
Column: Index of column, where non-zero element is located
Value: Value of the non zero element located at index – (row, column)



Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Sparse Matrix Representation using Linked List:

Method 2: Using Linked Lists

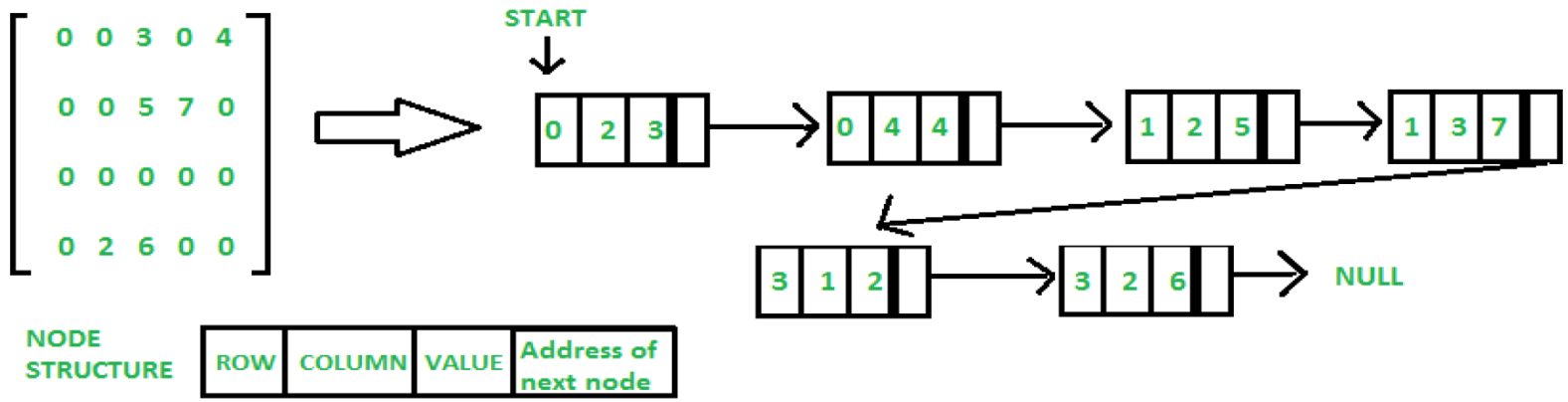
In linked list, each node has four fields. These four fields are defined as:

Row: Index of row, where non-zero element is located

Column: Index of column, where non-zero element is located

Value: Value of the non zero element located at index – (row, column)

Next node: Address of the next node



Advantages of Single Linked list:

- Insertions and Deletions can be done easily.
- It does not need movement of elements for insertion and deletion.
- Space is not wasted as we can get space according to our requirements.
- Its size is not fixed. It can be extended or reduced according to requirements.
- Elements may or may not be stored in consecutive memory available, even then we can store the data in computer.
- It is less expensive.

Disadvantages of Single Linked list:

- It requires more space as pointers are also stored with information.
- Different amount of time is required to access each element.
- If we have to go to a particular element then we have to go through all those elements that come before that element.
- We cannot traverse it from last & only from the beginning.
- It is not easy to sort the elements stored in the linear linked list.