UNIT I –LINEAR DATA STRUCTURES

Abstract Data Types (ADTs) – List ADT – Array-Based Implementation – Linked List Implementation – Singly Linked Lists – Circularly Linked List - Doubly-Linked Lists – Applications of Linked List

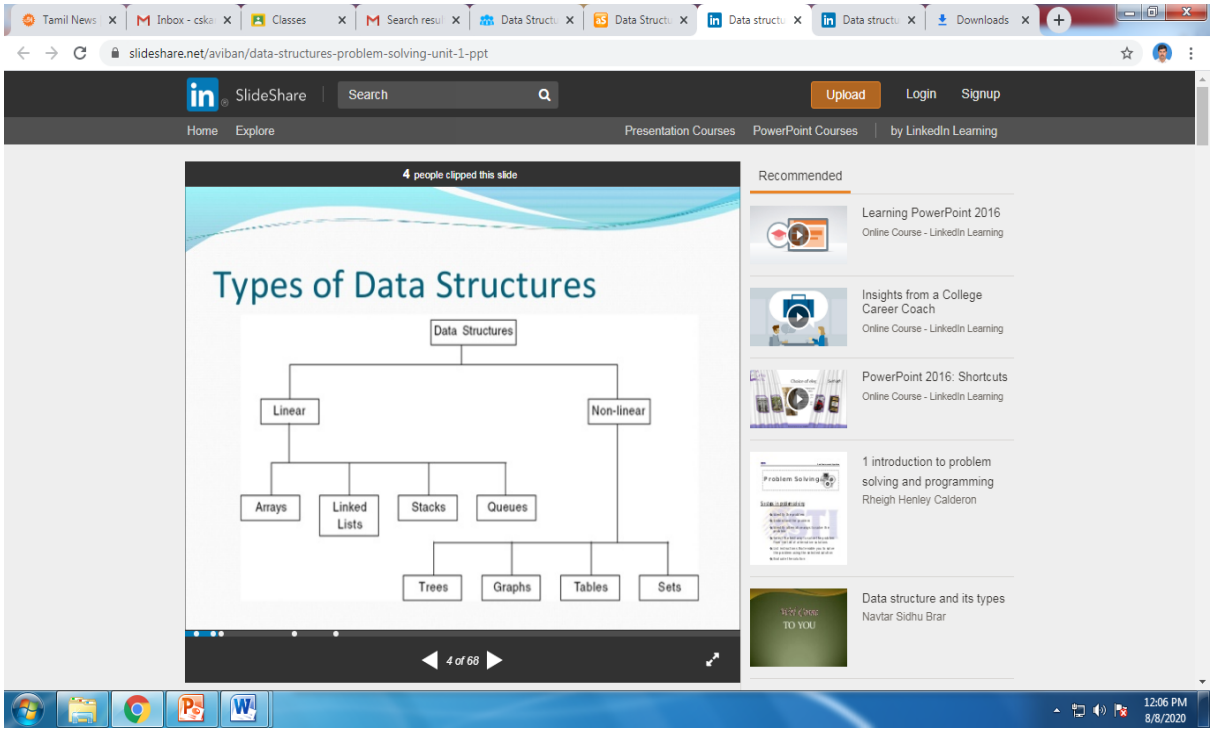
Data Structures

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 structures are used in almost every program or software system. Some common examples of data structures are arrays, linked lists, queues, stacks, binary trees, and hash tables. Data structures are widely applied in the following areas:

- Compiler design
- Operating system
- Statistical analysis package
- DBMS
- Artificial intelligence
- Graphics

Data structures are generally categorized into two classes: Linear and Non Linear data structures. If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. Examples include arrays, linked lists, stacks, and queues. Linear data structures 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. However, if the elements of a data structure are not stored in a sequential order, then it is a non-linear data structure. The relationship of adjacency is not maintained between elements of a non-linear data structure. Examples include trees and graphs.



Types of data Structures

Linear Data Structures	Non-linear Data Structures
In linear data structure, data elements are sequentially connected and each element is traversable through a single run.	In non-linear data structure, data elements are hierarchically connected and are present at various levels.
Linear data structures can be traversed completely in a single run.	Non-linear data structures are not easy to traverse and needs multiple runs to be traversed completely.
Linear data structures are not very memory friendly and are not utilizing memory efficiently.	Non-linear data structures uses memory very efficiently.
Time complexity of linear data structure often increases with increase in size.	Time complexity of non-linear data structure often remain with increase in size.
Array, List, Queue, Stack.	Graph, Map, Tree.

Comparison of Linear and Non Linear Data Structures

ABSTRACT DATA TYPE(ADT)

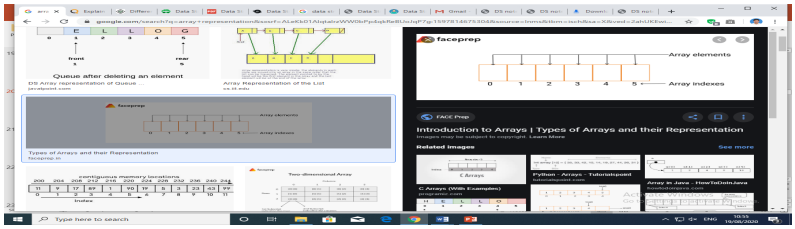
An *abstract data type* (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job. For example, stacks and queues are perfect examples of an ADT. We can implement both these ADTs using an array or a linked list. This demonstrates the 'abstract' nature of stacks and queues.

LIST ADT:

List is a linear data structures which consists of sequential collection of elements of same data type. List can be implemented using arrays and Linked List

LIST USING 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 (also known as the subscript). The subscript is an ordinal number which is used to identify an element of the array.



Basic Operations

- Creation
- Insert
- Delete
- Find/Search
- Display/Traversal

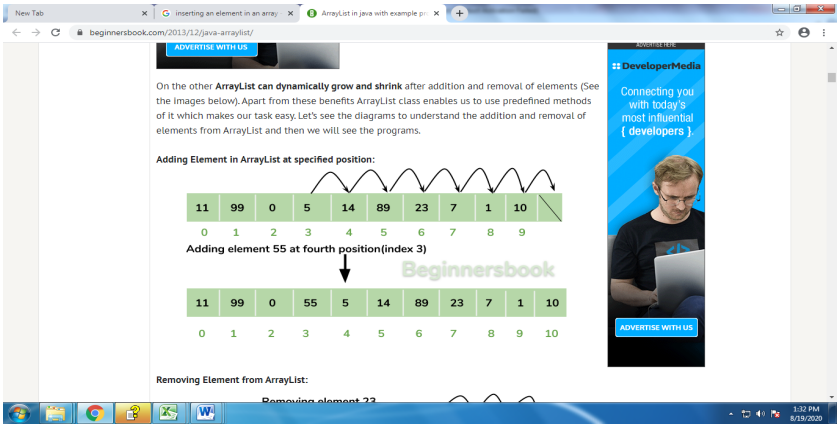
Creation:

```
# define maxsize 10
int lst[maxsize];
int size=0
```

The variable maxsize denotes the size of the array and size represents the number of elements in the List. Since the list is empty, size is 0.

Insert Operation

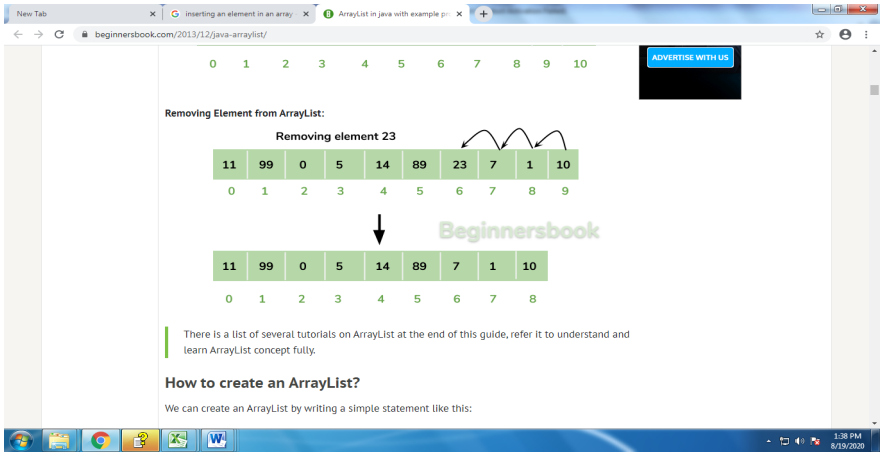
Inserting an element into list requires the position to be inserted. To insert an element at intermediate position requires movement of element after the position to be inserted by one position to its right. Insertion into the list cannot be done it the list is already full.




```
void insert(int pos, int val)
 {
      if(size<maxsize-1)
      {
            if( ( pos>=0) && (pos<=size ) )
            {
                  for (int k=size-1;k>=pos;k--)
                        lst[k+1]=lst[k];
                  lst[pos]= val;
                  size++;
                  printf(" %d has been inserted", val);
             }
            else
               printf("\n Invalid position");
      }
      else
         printf("\n List is full");
   }
```

Delete Operation:

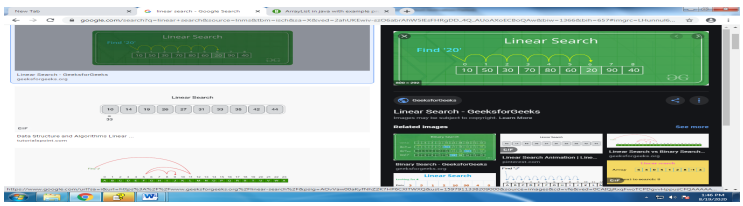
Deletion of an element from the list requires to find the position of the element to be deleted. Deleting an element at intermediate position requires movement of element after deletion to be moved by one position to its left.




```
void remove(int pos)
{
        if(pos>=0 && pos<size)
        {
                printf("%d is removed", lst[pos]);
                for(int k=pos;k<size;k++)
                {
                   lst[k]=lst[k+1];
                }
                size--;
        }
        else
        printf("\n Element Cannot be Removed");
     }
```

Search/Find Operation:

Searching an element in list is based on linear search. It sequentially checks each element of the list until a match is found or the whole list has been searched




```
void search(int val)
   {
      for(int k=0;k<size;k++)
      {
         if(lst[k]==val)
            printf("\n Element found at the position: %d",k);
```

```
            return;
        }
        printf("\n Element Not found");
    }
```

Display/Traverse

Traversing the element is done by iterating from first to last element directly by using the index of the list

```
void display()
{
        printf("\n List Elements are..");
        for(int i=0;i<size;i++)
            printf("\n %d",lst[i]);
 }
```

Disadvantages:

- Size of the array is fixed.
- Insertion and deletion requires movement of element and is time consuming
- Wastage of memory if allotted memory is not used.

LINKED LIST

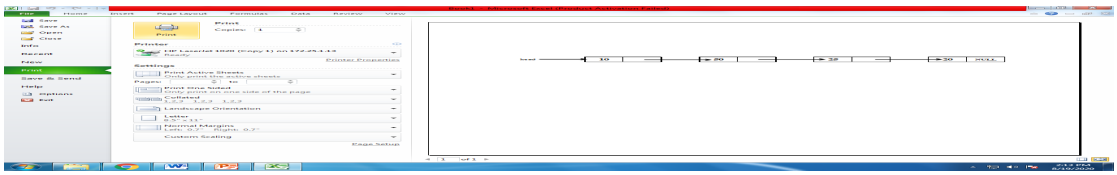
A linked list, in simple terms, is a linear collection of data elements. These data elements are called nodes. Linked list is a data structure which in turn can be used to implement other data structures. Thus, it acts as a building block to implement data structures such as stacks, queues, and their variations. A linked list can be perceived as a train or a sequence of nodes in which each node contains one or more data fields and a pointer to the next node. Every node has two parts: Data and a pointer (link)

Types of Linked List

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

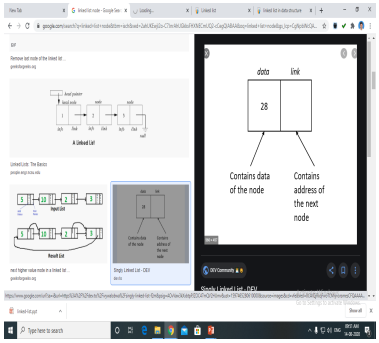
SINGLY LINKED LIST

A singly linked list is a type of linked list that is unidirectional, can be traversed in only one direction from head to the last node (tail). A single node contains data and a pointer to the next node.



Structure of Node:

Every node has a data part and a pointer that links to the next node




```
struct Node
{
        int data;
        struct Node *next;
};
```

Basic Operations:

- Insertion
- Deletion
- Traverse/Display
- Search/Find

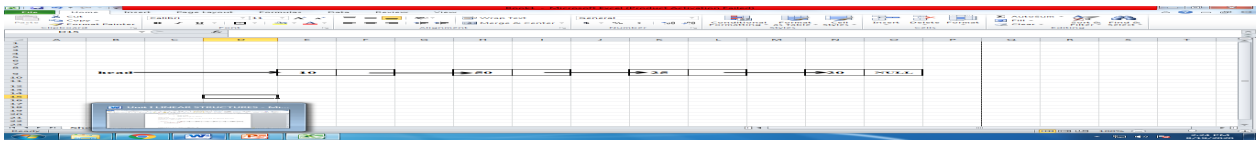
Insert Operation

Insertion is process of adding elements into the list. Based on the position to be inserted insertion is classified as:

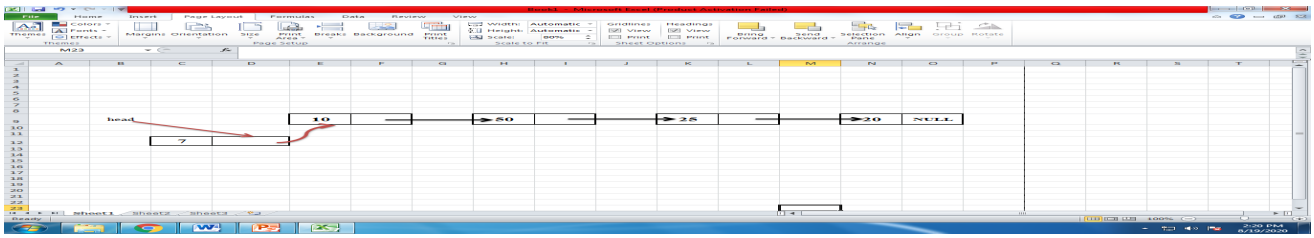
- Insert at First
- Insert at Last
- Insert after a specific element

Insert at First

Inserting an element at beginning requires creation of new node and changing the pointer assignment of head pointer



Before Insertion

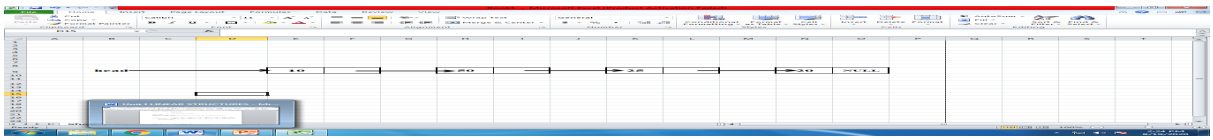


After Inserting 7

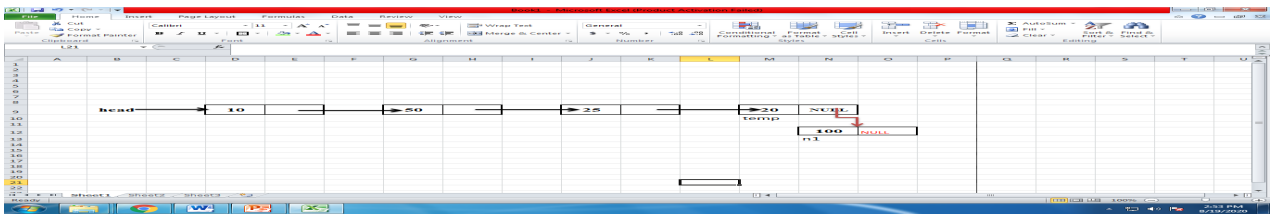
```
void insertatfirst(int val)
   {
      struct Node *n1= (struct Node *) malloc (sizeof (struct Node));
      n1-> data=val;
      n1- > next=head;
      head=n1;

   }
```

Insert at Last

Inserting an element at last requires creation of new node, traversing the list to the last node and changing the pointer assignment of last node to point to the new node.



Before Insertion



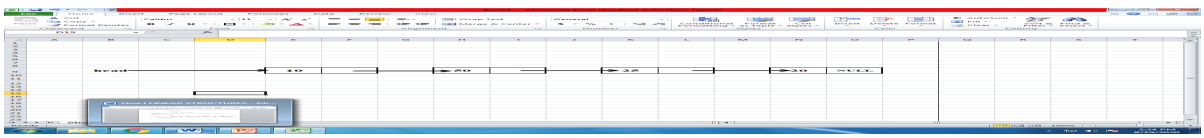
After Inserting 100

```
void insertlast(int val)
{
        struct Node *temp= (struct Node *) malloc (sizeof (struct Node));
        struct Node *n1= (struct Node *) malloc (sizeof (struct Node));
      temp=head;
```

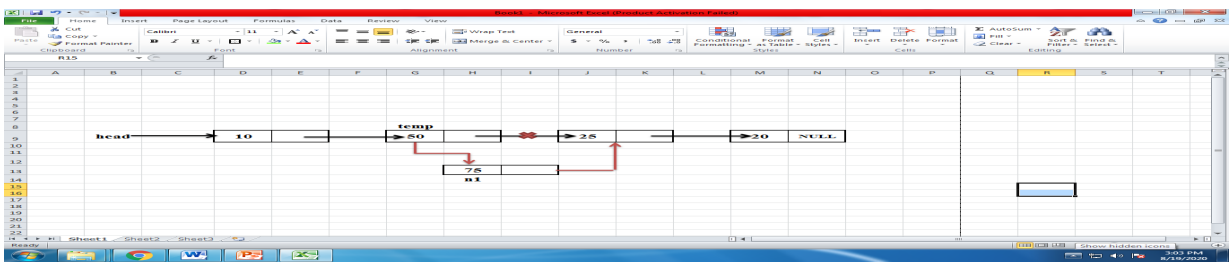
```
        n1->data=val;
        n1->next=NULL;
        while(temp->next!=NULL)
        {
                temp=temp->next;
        }
        temp->next=n1;

    }
```

Insert after a specific element

Inserting an element after a specific element needs to find the element, creation of new node and changing the pointer assignment accordingly.



Before Insertion



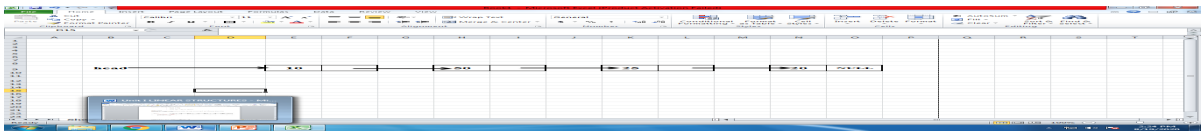
After Inserting 75

```
void insertafter(int aft,int val)
{

        struct Node *temp= (struct Node *) malloc (sizeof (struct Node));
        temp=head;
        while(temp!=NULL)
        {
           if(temp->data==aft)
           {
                        Node *n1=(struct Node *) malloc (sizeof (struct Node));
                        n1->data=val;
                        n1->next=temp->next;
                        temp->next=n1;
```

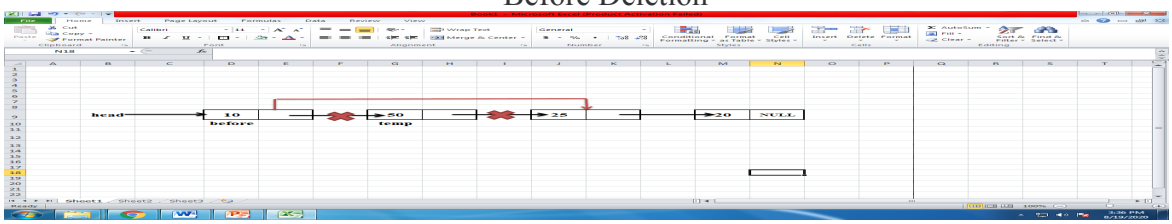
```
                return;
            }
            else
                temp=temp->next;
        }
        printf("\nInsertion Not possible");
    }
```

Deletion

Deletion of a node in singly linked list needs to find the element to be deleted and tracking its previous element in the list and changing the pointer of the previous node accordingly.



Before Deletion



After Deleting 50

```
void remove(int val)
{
        struct Node *temp= (struct Node *) malloc (sizeof (struct Node));
        struct Node *before= (struct Node *) malloc (sizeof (struct Node));
          temp=head;
        if(head->data==val)
        {
            head=head->next;
        }
        else
        {
                        before=head;
                while(temp  !=NULL)
                {
                        if(temp->data==val)
                        {
                            before->next=temp->next;
```

```
                    return;
                }
                else
                {
                    before=temp;
                    temp=temp->next;
                }
            }
        printf("\nElement cannot be deleted");
        }

    }
```

Find/Search Operation

Searching an element in list is based on linear search. It sequentially checks each element of the list until a match is found or the whole list has been searched.

```
void search(int val)
    {
        struct Node *temp= (struct Node *) malloc (sizeof (struct Node));
        temp=head;
        while(temp!=NULL)
        {
                if(temp->data==val)
                {
                    printf("\n Element found in the list");
                    return;
                }
                temp=temp->next;
        }
        printf("\n Element Not found in the list");

    }
```

Display/Traverse

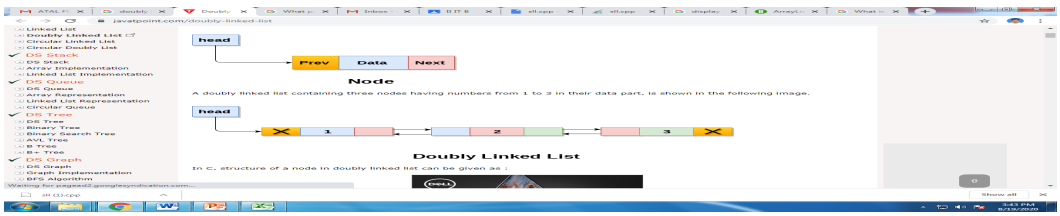
Traversing the element is done by iterating from first to last element directly by using the head pointer.

```
void display()
    {
```

```
    struct Node *temp= (struct Node *) malloc (sizeof (struct Node));
    temp=head;
    printf("\nList Elements are....");
    while(temp!=NULL)
    {
            printf("\n %d", temp->data);
            temp=temp->next;
    }
}
```

DOUBLY LINKED LIST

Doubly linked list is a type of linked list in which each node apart from storing its data has two links. The first link points to the previous node in the list and the second link points to the next node in the list



Basic Operations:

- Insertion
- Deletion
- Traverse/Display
- Search/Find

Structure of Node

Every node has a data part, a pointer that links to the next node and a pointer that refers to the previous node.

```
struct Node
{
        public:
        int data;
        struct Node * next;
```

```
        struct Node * prev;
};
```

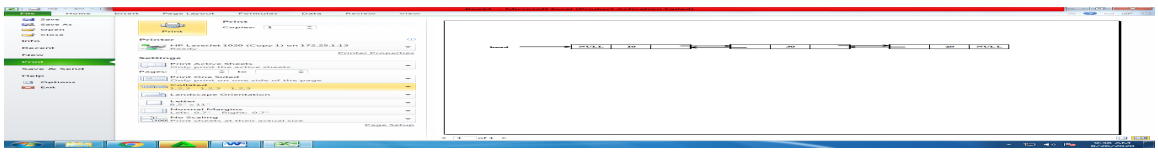
Insert Operation

Insertion is process of adding elements into the list. Based on the position to be inserted insertion is classified as:

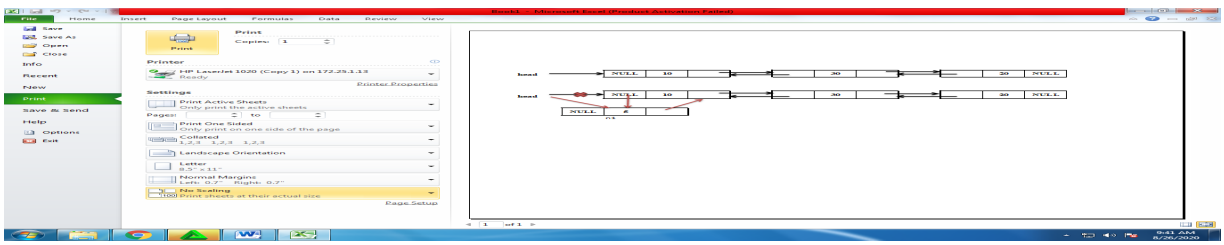
- Insert at First
- Insert at Last
- Insert after a specific element

Insert at First

Inserting an element at beginning requires creation of new node and changing the pointer assignment of head pointer



Before Insertion



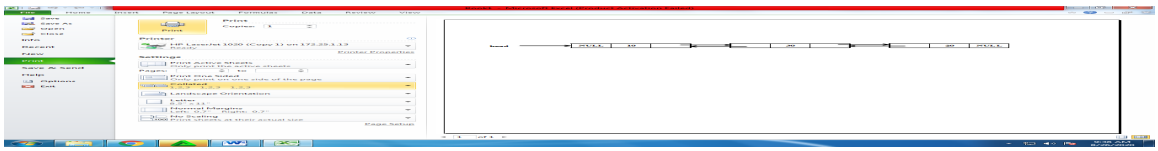
After Inserting 5

```
void insertatfirst(int val)
    {
            struct Node *n1=(struct Node *) malloc (sizeof (struct Node));
            n1->data=val;
            n1->prev=NULL;
            if(head==NULL)
            {
               n1->next=NULL;
            }
            else
            {
               n1->next=head;
```

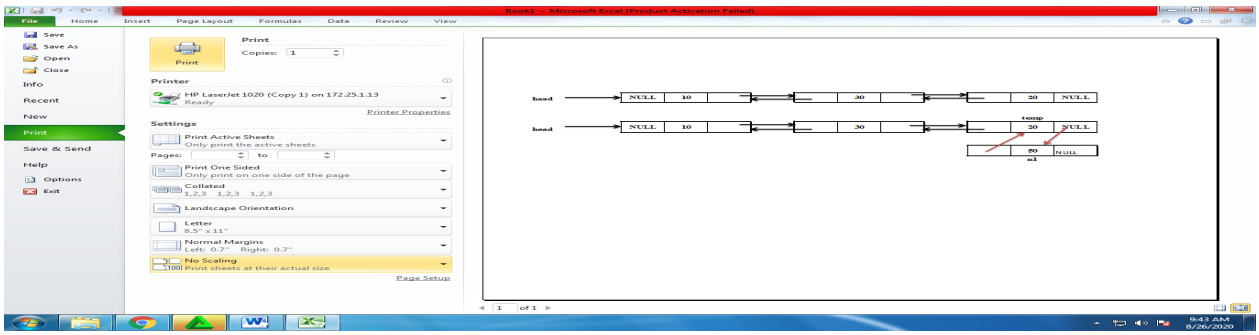
```
            head->prev=n1;
          }
         head=n1;
  }
```

Insert at Last

Inserting an element at last requires creation of new node, traversing the list to the last node and changing the next pointer assignment of last node to point to the new node, prev pointer of new node to point to the existing last node



Before Insertion

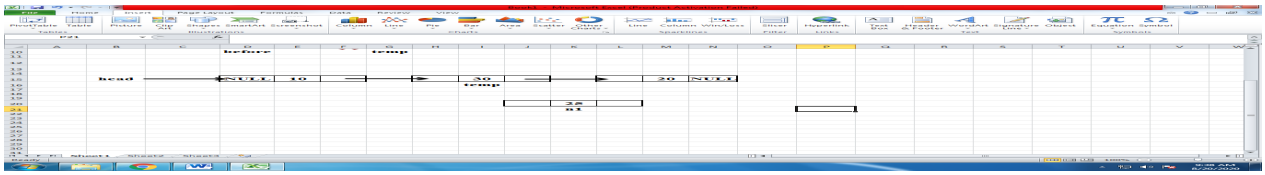


After Inserting 50

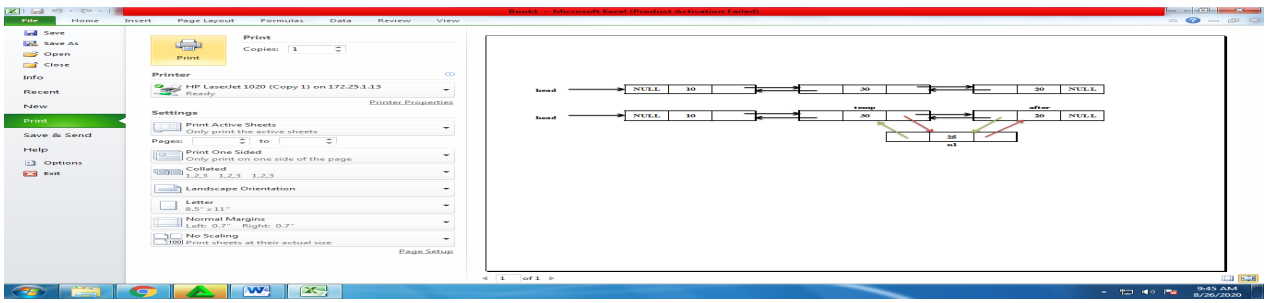
```
void insertlast(int val)
{
        struct Node *temp==(struct Node *) malloc (sizeof (struct Node));
         struct Node *n1==(struct Node *) malloc (sizeof (struct Node));
        temp=head;
        n1->data=val;
        n1->next=NULL;
        while(temp->next!=NULL)
        {
          temp=temp->next;
        }
        temp->next=n1;
        n1->prev=temp;
  }
```

Insert after a specific element

Inserting an element after a specific element needs to find the element, creation of new node and changing the next and prev pointer assignment accordingly.



Before Insertion

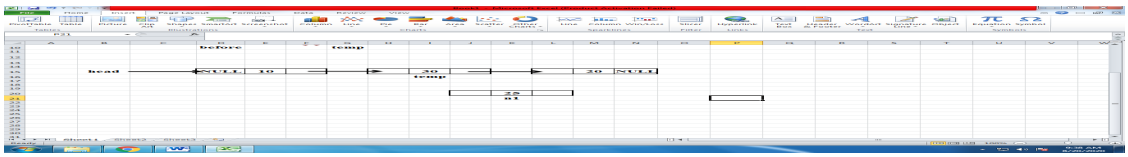


Inserting 25 after 30

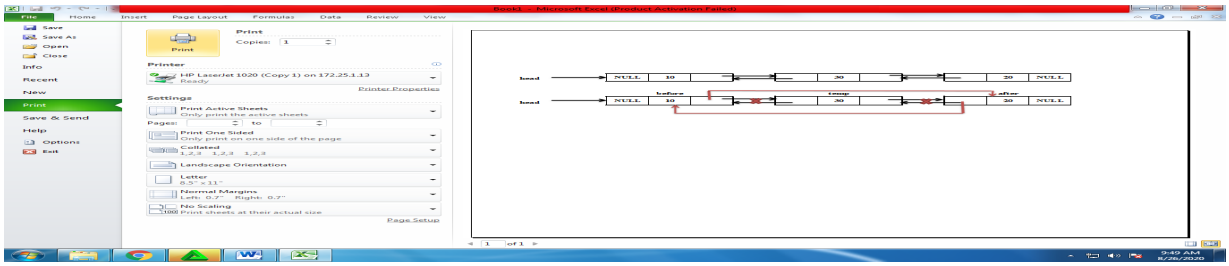
```
void insertafter(int aft,int val)
{
   struct Node *temp==(struct Node *) malloc (sizeof (struct Node));
   temp=head;
   while(temp!=NULL)
   {
      if(temp->data==aft)
      {
         struct Node *n1==(struct Node *) malloc (sizeof (struct Node));
         n1->data=val;
         n1->next=temp->next;
         temp->next=n1;
         n1->prev=temp;
         return;
      }
      else
         temp=temp->next;
   }
   printf("\nInsertion Not possible");
}
```

Delete Operation

Deleting an element in the list requires finding the element, changing the next and prev pointer of its adjacent nodes accordingly.



Before deletion



After deleting 30

```
void remove(int val)
{
        struct Node *temp==(struct Node *) malloc (sizeof (struct Node));
        struct Node *before==(struct Node *) malloc (sizeof (struct Node));
        struct Node *after==(struct Node *) malloc (sizeof (struct Node));
        temp=head;
        while(temp !=NULL)
        {
              if(temp->data==val)
              {

                 before=temp->prev;
                 after=temp->next;
                 if(temp==head)
                 {
                    head=after;
                    after->prev=before;
                 }
                 else if(after==NULL)
                    before->next=after;
                 else
                 {
```

```
                before->next=after;
                after->prev=before;
                }
              free(temp);
              return;
            }
            else
            {
               temp=temp->next;
            }
         }
   printf("\n Element cannot be deleted ");
}
```

Find/Search Operation

Searching an element in list is based on linear search. It sequentially checks each element of the list until a match is found or the whole list has been searched

```
void search(int val)
{
      Node *temp;
      temp=head;
      while(temp!=NULL)
      {
         if(temp->data==val)
         {
            printf("Element found in the list");
            return;
         }
         temp=temp->next;
      }
      printf("Element Not found in the list");

   }
```

Display/Traverse

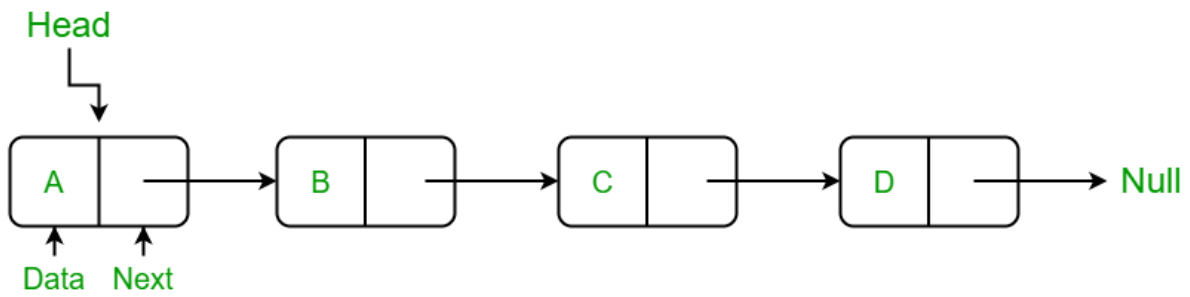
Traversing the element is done by iterating from first to last element directly by using the head pointer.

```
void display()
```

```
{
    Node *temp;
    temp=head;
    printf("List Elements are....");
    while(temp!=NULL)
    {
       printf("%d",temp->data);
       temp=temp->next;
    }
}
```

Applications of linked list:

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



Applications of linked list in computer science:

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

Applications of linked list in the real world:

1. Image viewer – Previous and next images are linked and can be accessed by the next and previous buttons.
2. Previous and next page in a web browser – We can access the previous and next URL searched in a web browser by pressing the back and next buttons since they are linked as a linked list.

3. Music Player – Songs in the music player are linked to the previous and next songs. So you can play songs either from starting or ending of the list.
4. GPS navigation systems- Linked lists can be used to store and manage a list of locations and routes, allowing users to easily navigate to their desired destination.
5. Robotics- Linked lists can be used to implement control systems for robots, allowing them to navigate and interact with their environment.
6. Task Scheduling- Operating systems use linked lists to manage task scheduling, where each process waiting to be executed is represented as a node in the list.
7. Image Processing- Linked lists can be used to represent images, where each pixel is represented as a node in the list.
8. File Systems- File systems use linked lists to represent the hierarchical structure of directories, where each directory or file is represented as a node in the list.
9. Symbol Table- Compilers use linked lists to build a symbol table, which is a data structure that stores information about identifiers used in a program.
10. Undo/Redo Functionality- Many software applications implement undo/redo functionality using linked lists, where each action that can be undone is represented as a node in a doubly linked list.
11. Speech Recognition- Speech recognition software uses linked lists to represent the possible phonetic pronunciations of a word, where each possible pronunciation is represented as a node in the list.
12. Polynomial Representation- Polynomials can be represented using linked lists, where each term in the polynomial is represented as a node in the list.
13. Simulation of Physical Systems- Linked lists can be used to simulate physical systems, where each element in the list represents a discrete point in time and the state of the system at that time.

Polynomial Manipulation:

A polynomial is composed of different terms where each of them holds a coefficient and an exponent. This tutorial chapter includes the representation of polynomials using linked lists and arrays.

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + \ldots. + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- one is the coefficient
- other is the exponent

Example:

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

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent

Representation of Polynomial

Polynomial can be represented in the various ways. These are:

- By the use of arrays
- By the use of Linked List

By the use of Linked List

A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two-tuple which holds two pieces of information:

- The exponent part
- The coefficient part

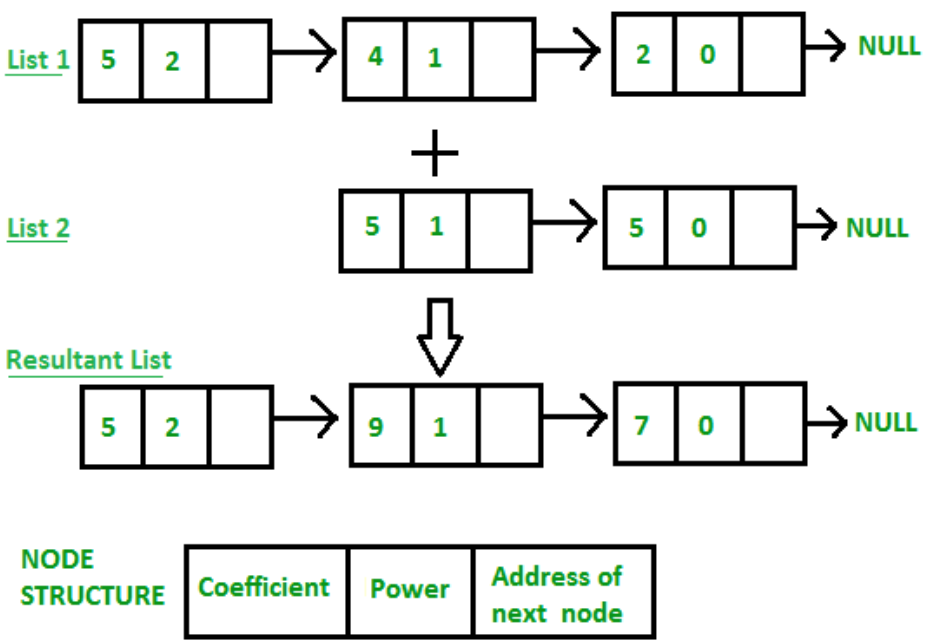
Polynomial operations:

In polynomial we can perform the following operations like

- Polynomial addition
- Polynomial subtraction
- Polynomial multiplication

In this chapter we are going to discuss polynomial addition in detail.

Polynomial addition:



Polynomial Manipulation

Polynomial manipulations are one of the most important applications of linked lists. Polynomials are an important part of mathematics not inherently supported as a data type by most languages. A polynomial is a collection of different terms, each comprising coefficients, and exponents. It can be represented using a linked list. This representation makes polynomial manipulation efficient.

While representing a polynomial using a linked list, each polynomial term represents a node in the linked list. To get better efficiency in processing, we assume that the term of every polynomial is stored within the linked list in the order of decreasing exponents. Also, no two terms have the same exponent, and no term has a zero coefficient and without coefficients. The coefficient takes a value of 1.

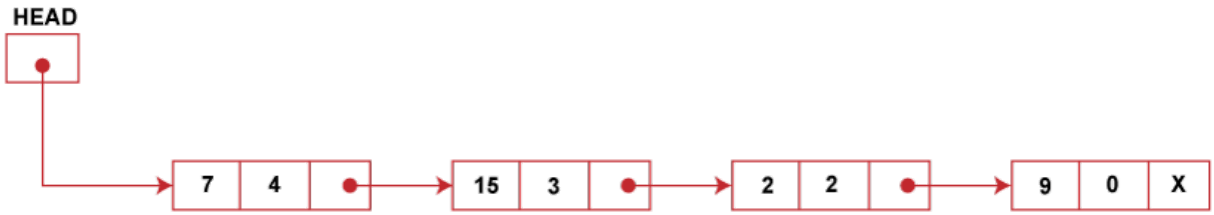
Each node of a linked list representing polynomial constitute three parts:

- The first part contains the value of the coefficient of the term.
- The second part contains the value of the exponent.
- The third part, LINK points to the next term (next node).

The structure of a node of a linked list that represents a polynomial is shown below:

Node representing a term of a polynomial

Consider a polynomial P(x) = $7x^4 + 15x^3 - 2x^2 + 9$. Here 7, 15, -2, and 9 are the coefficients, and 4,3,2,0 are the exponents of the terms in the polynomial. On representing this polynomial using a linked list, we have



Linked representation of polynomial P(x)

Observe that the number of nodes equals the number of terms in the polynomial. So we have 4 nodes. Moreover, the terms are stored to decrease exponents in the linked list. Such representation of polynomial using linked lists makes the operations like subtraction, addition, multiplication, etc., on polynomial very easy.

Addition of Polynomials:

To add two polynomials, we traverse the list P and Q. We take corresponding terms of the list P and Q and compare their exponents. If the two exponents are equal, the coefficients are added to create a new coefficient. If the new coefficient is equal to 0, then the term is dropped, and if it is not zero, it is inserted at the end of the new linked list containing the resulting polynomial. If one of the exponents is larger than the other, the corresponding term is immediately placed into the new linked list, and the term with the smaller exponent is held to be compared with the next term from the other list. If one list ends before the other, the rest

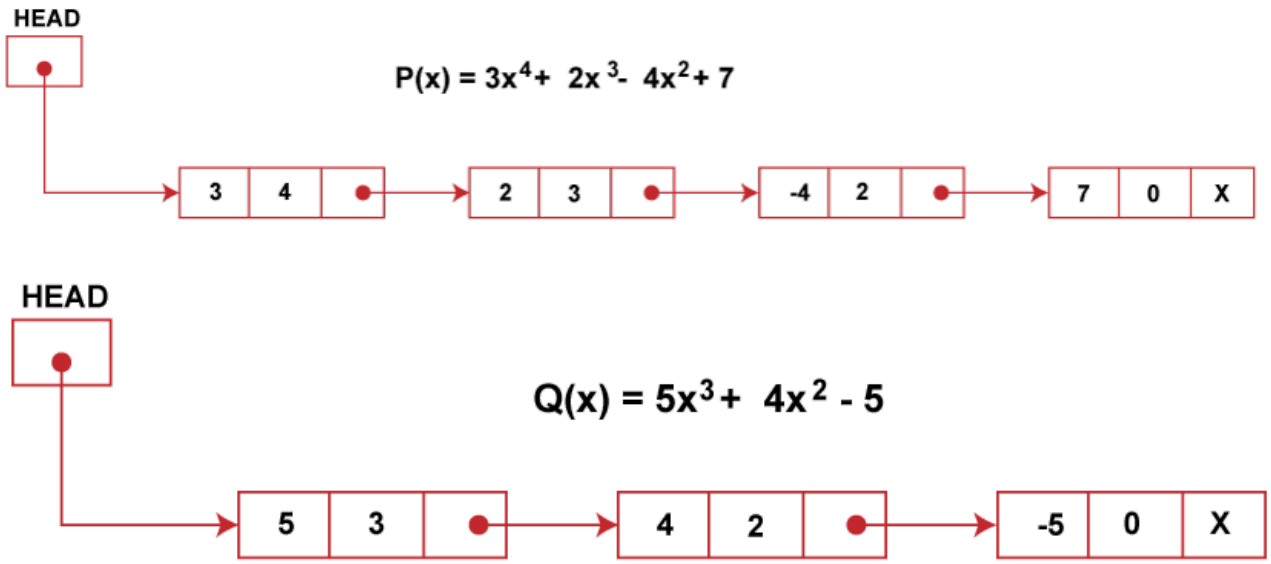
of the terms of the longer list is inserted at the end of the new linked list containing the resulting polynomial.

Let us consider an example to show how the addition of two polynomials is performed,

$P(x) = 3x^4 + 2x^3 - 4x^2 + 7$

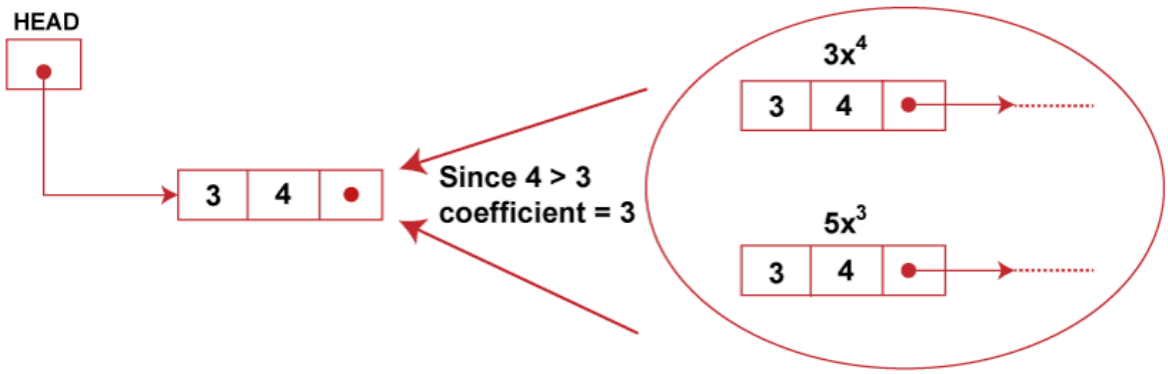
$Q(x) = 5x^3 + 4x^2 - 5$

These polynomials are represented using a linked list in order of decreasing exponents as follows:



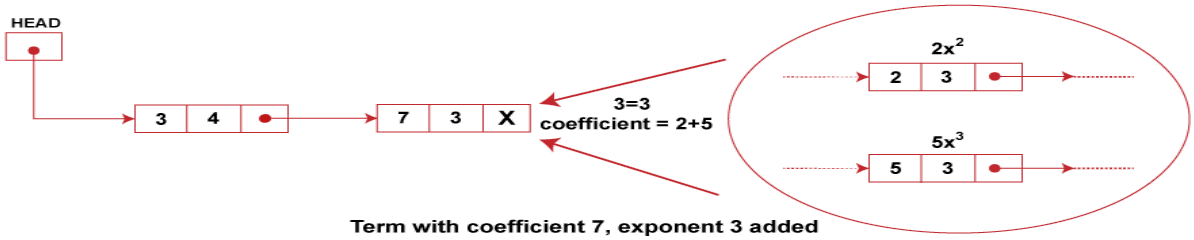
To generate a new linked list for the resulting polynomials that is formed on the addition of given polynomials P(x) and Q(x), we perform the following steps,

1. Traverse the two lists P and Q and examine all the nodes.
2. We compare the exponents of the corresponding terms of two polynomials. The first term of polynomials P and Q contain exponents 4 and 3, respectively. Since the exponent of the first term of the polynomial P is greater than the other polynomial Q, the term having a larger exponent is inserted into the new list. The new list initially looks as shown below:



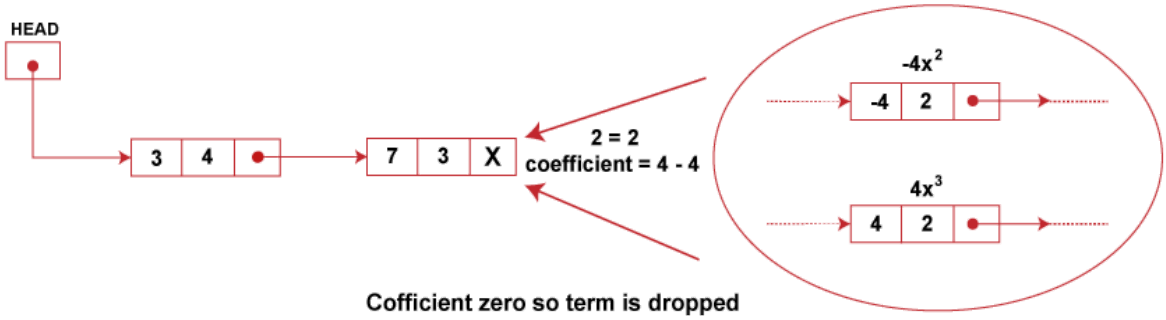
Term with coefficient 3, exponent 4 added

3. We then compare the exponent of the next term of the list P with the exponents of the present term of list Q. Since the two exponents are equal, so their coefficients are added and appended to the new list as follows:



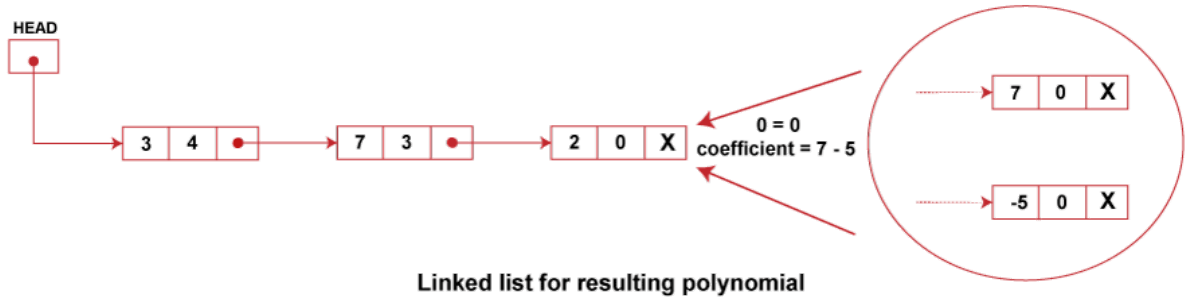
Term with coefficient 7, exponent 3 added

4. Then we move to the next term of P and Q lists and compare their exponents. Since exponents of both these terms are equal and after addition of their coefficients, we get 0, so the term is dropped, and no node is appended to the new list after this,



Cofficient zero so term is dropped

5. Moving to the next term of the two lists, P and Q, we find that the corresponding terms have the same exponents equal to 0. We add their coefficients and append them to the new list for the resulting polynomial as shown below:



Linked list for resulting polynomial

Example:

Program for Polynomial addition using C

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
int coef;
int exp;
struct Node* next;
};
typedef struct Node Node;
void insert(Node** poly, int coef, int exp) {
Node* temp = (Node*) malloc(sizeof(Node));
temp->coef = coef;
temp->exp = exp;
```

```
temp->next = NULL;
if (*poly == NULL) {
*poly = temp;
return;
}
Node* current = *poly;
while (current->next != NULL) {
current = current->next;
}
current->next = temp;
}
void print(Node* poly) {
if (poly == NULL) {
printf("0\n");
return;
}
Node* current = poly;
while (current != NULL) {
printf("%dx^%d", current->coef, current->exp);
if (current->next != NULL) {
printf(" + ");
}
current = current->next;
}
printf("\n");
}
Node* add(Node* poly1, Node* poly2) {
Node* result = NULL;
while (poly1 != NULL && poly2 != NULL) {
if (poly1->exp == poly2->exp) {
insert(&result, poly1->coef + poly2->coef, poly1->exp);
poly1 = poly1->next;
poly2 = poly2->next;
} else if (poly1->exp > poly2->exp) {
insert(&result, poly1->coef, poly1->exp);
poly1 = poly1->next;
} else {
insert(&result, poly2->coef, poly2->exp);
poly2 = poly2->next;
}
}
while (poly1 != NULL) {
insert(&result, poly1->coef, poly1->exp);
poly1 = poly1->next;
```

```
}
while (poly2 != NULL) {
insert(&result, poly2->coef, poly2->exp);
poly2 = poly2->next;
}
return result;
}
int main() {
Node* poly1 = NULL;
insert(&poly1, 5, 4);
insert(&poly1, 3, 2);
insert(&poly1, 1, 0);
Node* poly2 = NULL;
insert(&poly2, 4, 4);
insert(&poly2, 2, 2);
insert(&poly2, 1, 1);
printf("First polynomial: ");
print(poly1);
printf("Second polynomial: ");
print(poly2);
Node* result = add(poly1, poly2);
printf("Result: ");
print(result);
return 0;
}
```

GATE Questions:

1. What does the following function do for a given Linked List with first node as head?

```
 void fun1(struct node* head)
 {
 if(head == NULL)
   return;

 fun1(head->next);
 printf("%d  ", head->data);
}
```

a) Prints all nodes of linked lists
b) **Prints all nodes of linked list in reverse order**
c) Prints alternate nodes of Linked List
d) Prints alternate nodes in reverse order

2. Which of the following sorting algorithms can be used to sort a random linked list with minimum time complexity?

a) Insertion Sort
b) Quick Sort
c) Heap Sort
d) **Merge Sort**

3. What is the worst case time complexity of inserting n elements into an empty linked list, if the linked list needs to be maintained in sorted order?

a) **$O(n^2)$**
b) O(1)
c) O(n)
d) **O(n(n+1))**

4. In a circular linked list organization, insertion of a record involves modification of :

a) **One pointer**
b) Two pointer
c) multiple pointer
d) no pointer

5. Linked lists are not suitable data structures of which one of the following problems?

a) **Binary search**
b) Insertion sort
c) Radix sort
d) Polynomial manipulation