Module - 3

LINKED LISTS

Linked Lists: Definition, Representation of linked lists in Memory, Memory allocation; Garbage Collection. Linked list operations: Traversing, Searching, Insertion, and Deletion. Doubly Linked lists, Circular linked lists, and header linked lists. Linked Stacks and Queues. Applications of Linked lists – Polynomials, Sparse matrix representation. Programming Examples

Text 1: Ch4: 4.1 -4.8 except 4.6

Text 2: Ch5: 5.1 – 5.10

Linked lists are a way to store data with structures so that the programmer can automatically create a new place to store data whenever necessary. A linked list is called "linked" because each node in the series has a pointer that points to the next node in the list.

How Linked lists are different from arrays?

Consider the following points:

An array is a static data structure. This means the length of array cannot be altered at run time. While, a linked list is a dynamic data structure. In an array, all the elements are kept at consecutive memory locations while in a linked list the elements (or nodes) may be kept at any location but still connected to each other.

When to prefer linked lists over arrays? Linked lists are preferred mostly when you don't know the volume of data to be stored. For example, in an employee management system, one cannot use arrays as they are of fixed length while any number of new employees can join. In scenarios like these, linked lists (or other dynamic data structures) are used as their capacity can be increased (or decreased) at run time (as an when required).

How linked lists are arranged in memory?

Linked list basically consists of memory blocks that are located at random memory locations. Now, one would ask how are they connected or how they can be traversed? Well, they are connected through pointers. Usually a linked list is represented through a like this:

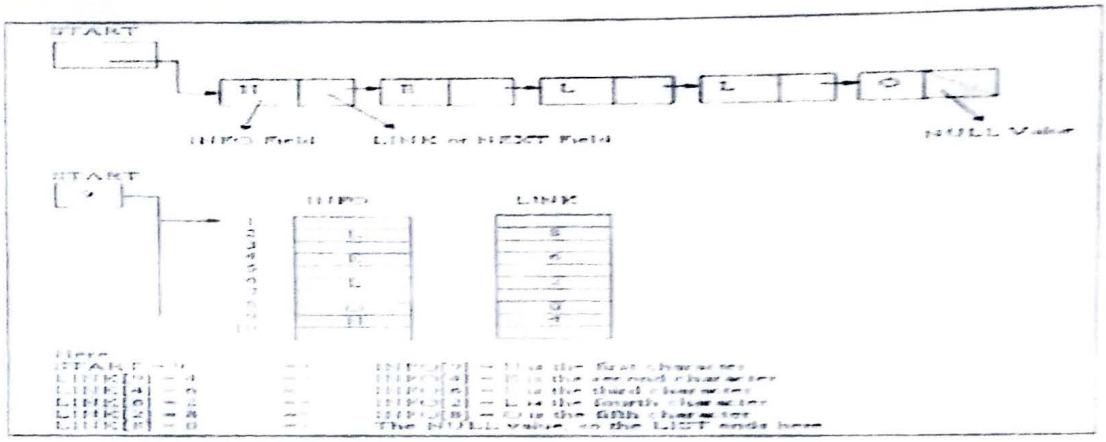


Fig 3.1 Linked list representation in memory

Memory Allocation and Garbage collection:

The maintenance of linked lists in memory assumes that here is a possibility of inserting new nodes into the list and hence requires mechanism which provides unused memory space for the new nodes. Hence together with the linked lists in the memory a special list is maintained which consist the memory cells which are unused. This list has its own pointer which points to the list of available **pace or the free storage list or free pool. When a new node has to be inserted first** the program **checks whether a node is available in list or pool, if it is available then the node will be allocated** from the **list if not the memory is allocated dynamically using functions like malloc.**

Garbage collection (GC) is a form of automatic memory management. The *garbage collector*, or just *collector*, attempts to reclaim *garbage*, or memory occupied by objects that are no longer in use by the program.

Single linked lists

A singly linked list is a concrete data structure consisting of a sequence of nodes. Each node stores

- element
- link to the next node

The diagram given below shows the structure of the node in a single linked lists.

Single linked list representation.

- The link filed of the last node is always null.

- Singly linked lists are also called chains. A chain may comprise of zero or more nodes. When the chain contains zero nodes the chain is empty. The nodes of an non empty chain are ordered so that the first node, links to the second node; the second to the third and so on.

Representation of chains in C

We need following capabilities to make linked representations possible:

(1) Mechanism for defining a nodes structure that is the field it contains. Self referential structures are used for this purpose.

(2) A way to create nodes. The malloc function is used for this purpose.

(3) A way to remove the nodes that we don't need, free is used for this purpose.

self referential structure for node:

```
struct node
{
   int info;
struct node *link;
};
```

allocating memory using malloc

```
struct node *p;
p=(struct node*)malloc(sizeof(struct node);
```

Linked list operations:

The following operations can be performed on the linked lists they are:

- Traversing, Searching, Insertion, and Deletion

  Traversing is operation where all the nodes of the list are visited. let us assume that there is first pointer which points to the first node in the linked list, then traversing can be achieved as follows:

  algorithm: traversing

  step 1: set p=first

  step 2: repeat 3 and 4 while !=NULL

  step 3: print info[p];

  step 4: set p=p→link;

  [end of step 2]

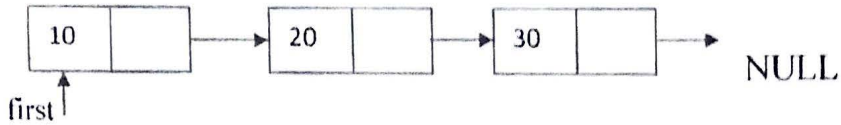
  step 5: exit;

Inserting a new node into the chain

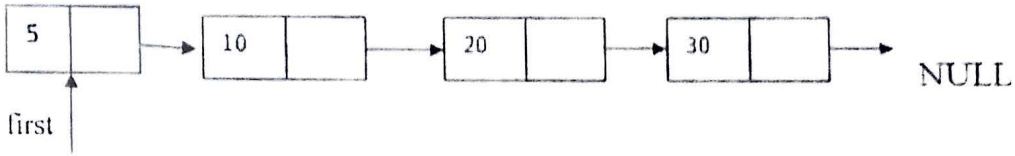
Inserting an element at the front

- Allocate a new node
- Insert new element

- Make new node point to first node.
- Update **first** to point to the new node.

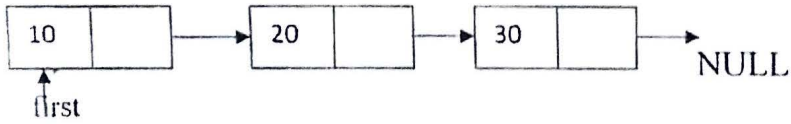


After inserting a new node with value 5 at the front.

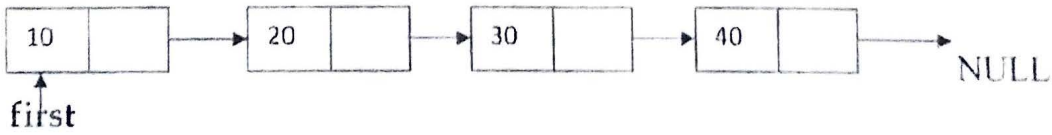


Inserting an element at the end

- Allocate a new node
- Insert new element
- Make last node point to new node.
- Update link of new node to NULL.



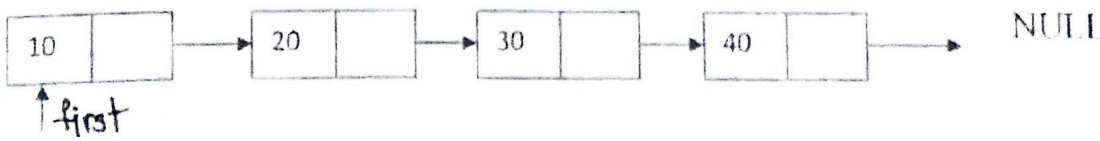
After inserting a new node at the last.



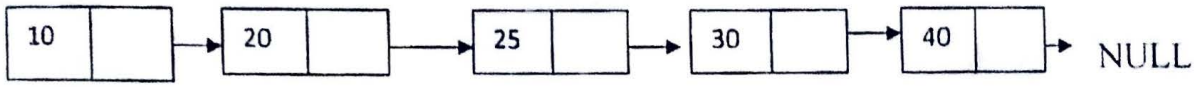
Inserting an element based on key

inserting after the key

- Allocate a new node
- Insert new element
- Make node with the key point to new node.
- Update link of new node to point to the next node where the key was found.

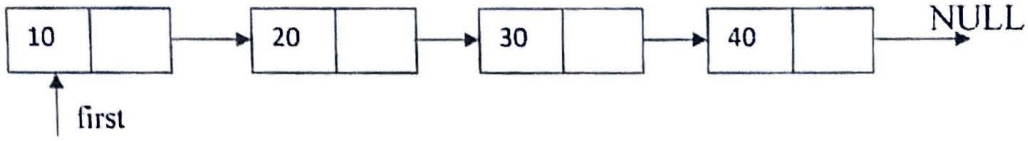


after inserting a node after key 20

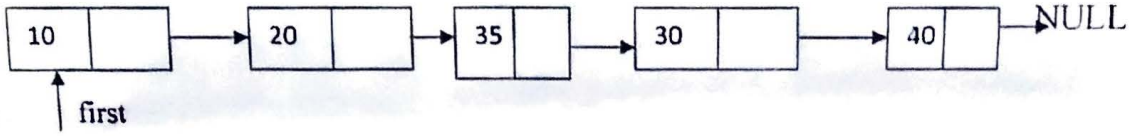


Inserting before the key

- Allocate a new node
- Insert new element
- Make new node point to node with the key.
- Update link of new node to point to the node where the key was found.



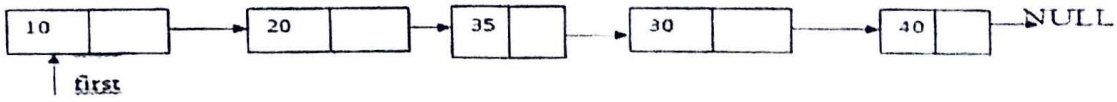
after inserting a new node before the node 30



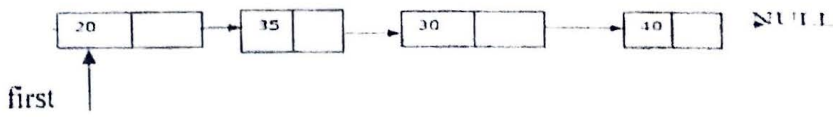
Deleting A Node From The Chain(SLL)

Deleting an element at the front

- Delete the first node
- Make first pointer point to next of first.
- while deleting the node check if it is only node left out,if so then set first to NULL.
- before deleting the node

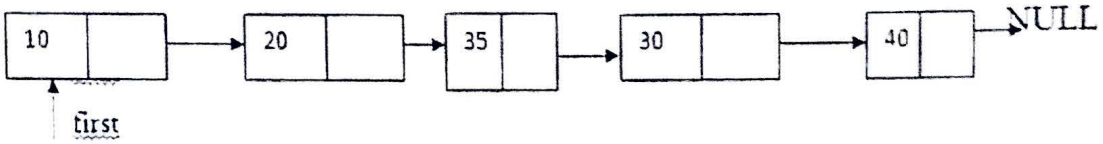


after deleting from front

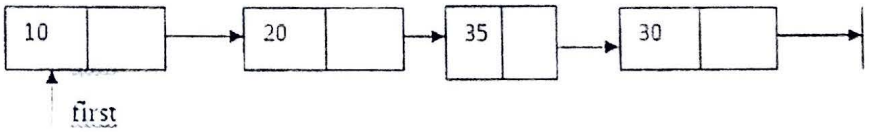


Deleting an element at the end

- Delete the last node
- Update the link of the node which contains the address of the last node to set to NULL.



After deleting the last node



Searching

Two types of searching

1) searching in sorted list:
   a) Suppose if the key is less than the value of the node which is compared currently, stop the searching process and report key not found.
   b) Else repeat the search comparing the key with all the nodes in the linked list till the last node or till key is found.
2) searching in unsorted list

   Comparing the key with all the nodes in the linked list till the last node or until the key is found.

Function in C to search for a key element in SLL (unsorted list)

Assume first is pointer which points to first node of the linked list.

```
void serach()
{
int key,found=0;
 printf("enter the key to be searched");
scanf("%d",&key);
struct node *temp;
temp=first;
while(temp!=NULL  && found==0)
{
```

```
if(temp→info=key)
found=1;
else
temp=temp→link;
        }
        if(found==1)
        printf("key found\n");
        else
        printf("key not found");
        }
```

Function in C to search for a key element in SLL (sorted list)

Assume first is pointer which points to first node of the linked list.

```
void serach()
{
int key.found=0;
 printf("enter the key to be searched");
 scanf("%d",&key);
 struct node *temp;
 temp=first;
 while(temp!=NULL  && found==0)
 {
 if(temp→info==key)
 found=1;
 else if(temp→info >key)
   break;
 else
 temp=temp→link;
          }
          if(found==1)
          printf("key found\n");
          else
          printf("key not found");
          }
```

Function in C to reverse element in SLL.

```
void reverse()
{
struct node *trail,*middle*lead;
lead=first;
middle=NULL;
while(lead)
{
trail=middle;
middle=lead;
lead=lead→link;
middle→link=trail;

first=middle;
printf("reversed list is\n");
temp=first;
while(temp!=NULL)
{
printf("%d",temp→info);
temp=temp→link;
}
}
```

Function in C to concatenate two SLL.

```
struct node * concatenate(sturct node *first1,struct node *first2)
{
if(first1==NULL)
return first2;
if(first2==NULL)
return first1;
for(temp=first1;temp!=NULL;temp=temp→link)

temp→link=first2;
}
```

Polynomials

Polynomial Representation

A polynomial $A(x) = a x^{e_{m-1}} + \dots a_0 x^{e_0}$, where a_i are non-zero co-efficients and e_i are non-negative integer exponents, each term is represented as a node containing co-efficient and exponent fields as well as pointer to next term.

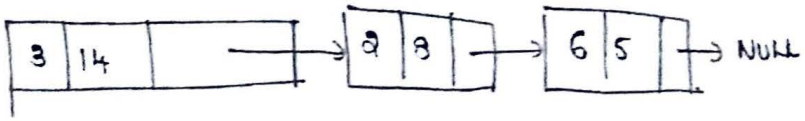
Definition of structure to represent node:

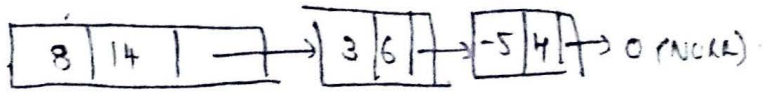
```
Struct node
{
   int coe;
   int exp;
   struct node *link;
};
```

Node : | coe | exp | link |

Ex: Suppose $a = 3x^{14} + 2x^8 + 6x^5$

$b = 8x^{14} + 3x^6 - 5x^4$





Adding two polynomials

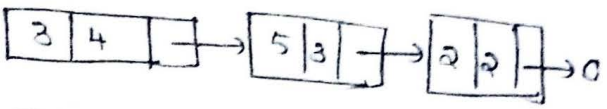
To add two polynomials examine their starting nodes pointed to by a and b. If the exponents of the two terms are equal, we add co-efficients and create a new term of the result. We also

move pointers a and b to the next nodes. If the exponent of the current term in a is less than the exponent of the current term in b, then create a node copy co-efficient and exponent of b, the move pointer b. If exponent of a is greater than repeat the same for a.

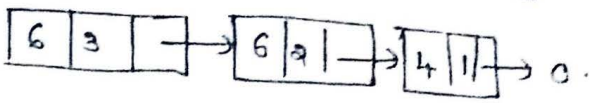
Ex: $a = 3x^4 + 5x^3 + 2x^2$

$b = 6x^3 + 6x^2 + 4x$ add two polynomials.

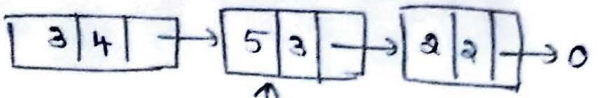
a :



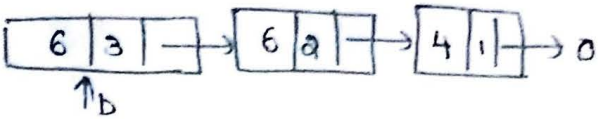
b:



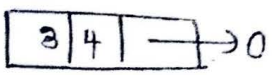
Compare exponent of a and b. Here a > b, therefore copy a and advance a.



(Here zero represents null).

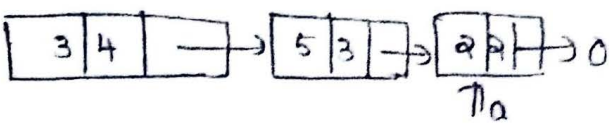


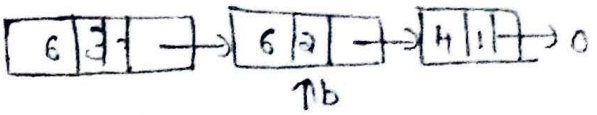
res:



a→exp = b→exp, add co-eff, move aand b.

res:





Again a→exp = b→exp, add co-eff move a and b

Doubly linked list

A **doubly linked list** is a **linked** data structure that consists of a set of sequentially **linked** records called nodes. Each node contains two fields, called links,that are references to the previous and to the next node in the sequence of nodes.

Following are advantages/disadvantages of doubly linked list over singly linked list.

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.

2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

3) We can quickly insert a new node before a given node.

In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Operations on DLL

Insertion

A node can be added in four ways

1) At the front of the DLL

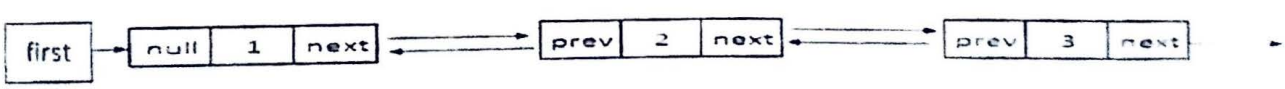
2) After a given node.

3) At the end of the DLL

4) Before a given node.

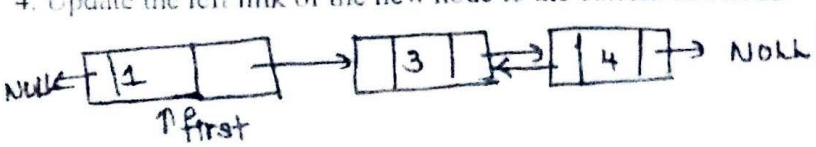
Insertion at front:

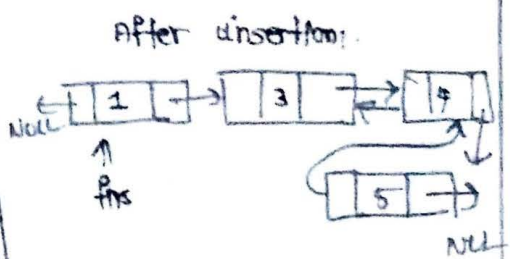
1. Create a new node
2. Set the right link of the new node to first
3. Update first to point to the new node.



Insertion at end:

1. Create a new node
2. Set the right link of the new node to NULL
3. Update the right link of the current last node to the new node.
4. Update the left link of the new node to the current last node.





Insertion based on the key(before the key)

1. Traverse the list till the key is found. if found

   a.Create a new node

   b. Set the right link of the new node to key

   c. Update the left link of key to the new node.

   d. Update the right link of the node before the key to the new node.

   e. Update the left link of the new node to the node before the key.

2. If the key not found . insertion fails.

Insertion based on the key (after the key)

1. Traverse the list till the key is found. if found

   a. Create a new node

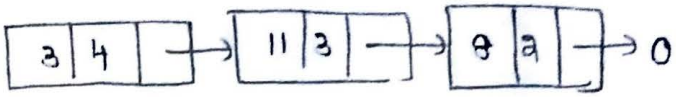
   b. Set the left link of the new node to key

   c. Update the right link of key to the new node.

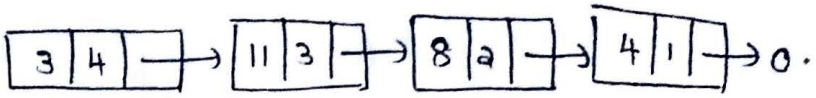
   d. Update the right link of the new node to node after key.

   e. Update the left link of node after the key to new node .

2. If the key not found. insertion fails.

∴ res: 

Now a is reached end, but did not reach, therefore copy remaining terms of b to res:-

∴ res: 

Function to add two polynomials

```
Void add( struct node *a, struct node *b)
{
   struct node * temp, * prev;
      Prev = NULL;
   While ( (a != NULL & b != NULL))
   {     temp = Create_node();
   if (a→exp == b→exp)
      {
        temp→ exp = a→exp;
        temp→ coe = a→ coe + b→ coe;
        a = a→link;
        b = b→ link;
        prev→ link = temp;
          Prev = temp;
      }
   else if (a→exp > b→exp)
      {  temp→exp = a→exp;
         temp→coe = a→ coe;
         a = a→link;
         prev→link = temp;
              Prev = temp;
      }
```

```
    else
      {
        temp→exp = b→exp;
        temp→coe = b→coe;
        b = b→link;
        Prev→link = temp;
        Prev = temp;
      }
  }
  While (a! = NULL)
    {
      temp = create_node();
      temp→coe = a→coe;
      temp→exp = a→exp;
      a = a→link;
      prev→link = temp;
      prev = temp;
    }
While (b! = NULL)
  {
    temp = create node();
    temp→coe = b→coe;
    temp→exp = b→exp;
    b = b→link;
    Prev→link = temp;
    Prev = temp;
  }
}
```

Erasing polynomials

Suppose if we perform an operation on polynomial as e(x) = a(x) * b(x) - d(x), the the sequence of function would be as follows.

```
a = read_poly();
b = read_poly();
d = read_poly();
temp = mult(a,b);
e = add(temp,d);
```

Now temp is a temporary polynomial, hence we can clear after the operation is performed.

```
void erase(struct node *p)
{
    struct node *t;
    while(*p)
    {
        t = p;
        p = p->link;
        free(t);
    }
}
```

Now, call erase(temp) to erase temp.

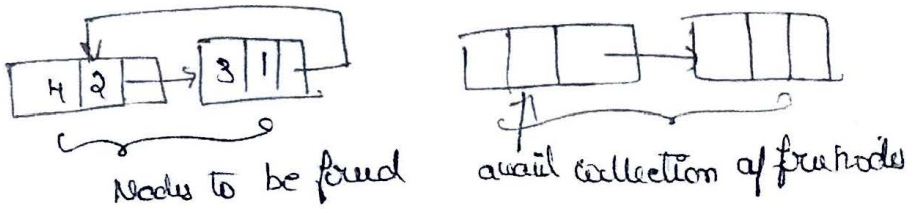
Linked list representation of polynomials

Here the link field of last node points to the first node in the list.

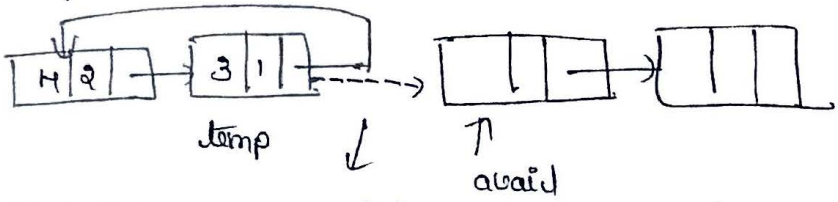
Eg: $a(x) = 3x^3 + 5x$

Erasing polynomials represented using circular list.

Ex: Suppose if we had to perform operation on polynomials such as $d(x) = a(x) * b(x) + c(x)$. Now if temp holds the result of operation $a(x) * b(x)$ and we want to free temp. But avail be the pointer pointing to free node which can be used later.

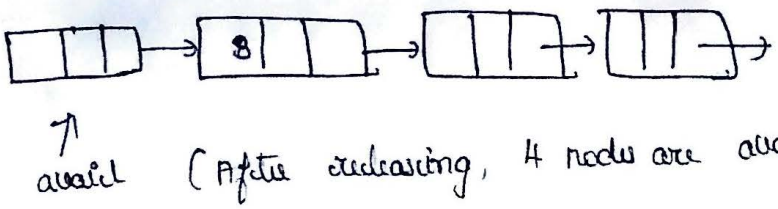


Set temp = last node of the list of nodes to be freed.



update temp → link = avail.

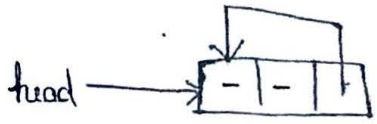
Update avail so that it points to beginning.



(After erasing, 4 nodes are available).

```
void cerase (struct node *p)   //code for erasing a circular polynomial
{
  Struct node *temp;
  if(p)
  {
    temp= p->link;
    p->link = avail;
    avail = temp;
    p= NULL;
  }
}
```

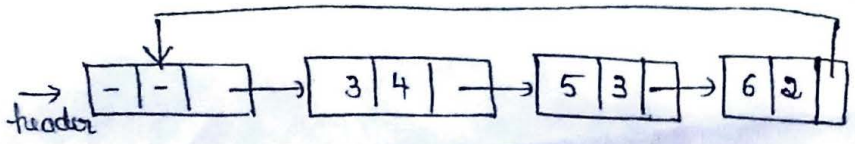
Code for erase, fnill function well exapt for a zero polynomial. A polynomid which does not contain any co-efficient and exponent.



Zero polynomial

To solve this special issue an additional node called header node is added to the circular linked list, this node does not contain any information, but it may contain the information about number of nodes present in the list. The last node points to the header node.

Ex:- Let $a = 3x^4 + 5x^3 + 6x^2$



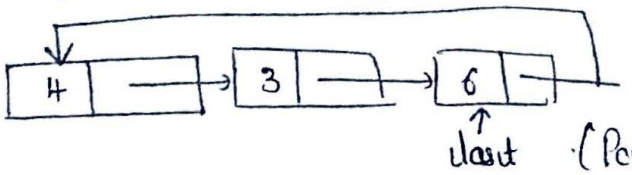
Polynomial represented using circular linked list containing header.

Operations for circularly linked lists

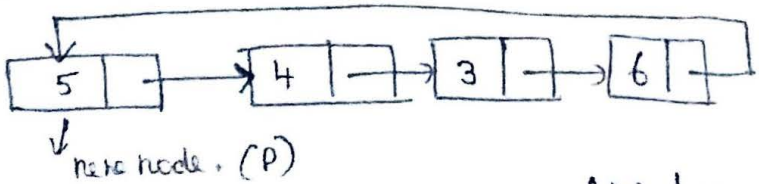
a) Insertion at fronct of list.

Insertion can be done easily in circular list, if there is a pointer pointing to the last node of the list.

Ex:-



To insert a node with value 5 at Fnont.



P→link = last→link; (∵ because last was containing address of node 4 earlier).

last→link = P;

Code for inserting at front.

```
Void insert_front()
{
  Struct node *p;
    p=create_node();
  if(last!=NULL) // check if list is empty or not
  {
    p->link= last->link;
    last->link=p;
  }
else
  {
    last=p;      // list empty.
  p->link= p;
  }
}
```

(b) Finding length of circular list.

```
int length(struct node *last)
{
  Struct node *temp;
  int count=0;
  if(last!=null)
  {
    temp=last;
  ~~while(temp!=last)~~ do
  {
    count++;
  temp=temp->link;
  }while(temp!=Last)
}return count;
```

Sparse Matrices

(1) Each column of a sparse matrix is represented as linked list with a header node.

(2) Data element node has five fields row, col, down, right, val

(a) header node (b) element node.

Down contains the address of node below, and right contains the address of node to right (i.e. elements of same row).

Ex: Represent the given sparse matrix using linked list.

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 4 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 \\ 8 & 0 & 0 & 1 \\ 0 & 0 & 6 & 0 \end{bmatrix}$$