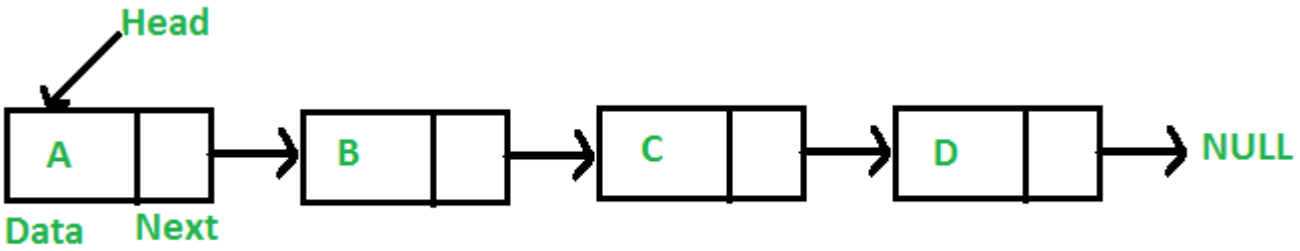
ISE Dept.
Transform Here

Mod-3 Linked list

Prepared by
Prof.S.Mahalakshmi
AP/ISE

- Linked List is a linear data structure.
- Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



Limitations of Arrays

- The size of the arrays is fixed.
- Inserting /deletion in an array of elements is expensive

Advantages of linked list

1) Dynamic size

2) Ease of insertion/deletion

Drawbacks:

- Random access is not allowed
- Extra memory space for a pointer is required with each element of the list.
- Not cache friendly

Representation:

- A linked list is represented by a pointer to the first node of the linked list.
- The first node is called the head. If the linked list is empty, then the value of the head is NULL.

  Each node in a list consists of at least two parts:

  1) data

  2) Pointer (Or Reference) to the next node

  In C, we can represent a node using structures

```
// A linked list node
struct node {
    int data;
    struct node  *next;
};
```

Note

- The variable 'first' contains the address of first node (initially NULL)
- All functions require first
- Functions that manipulate the linked list should return the address of the first node .
- first= NULL=> List is empty.
- first ->link= NULL=> There is one element in the List
- first ->link ! = NULL => There is more than one element in the-List

Types of Linked List

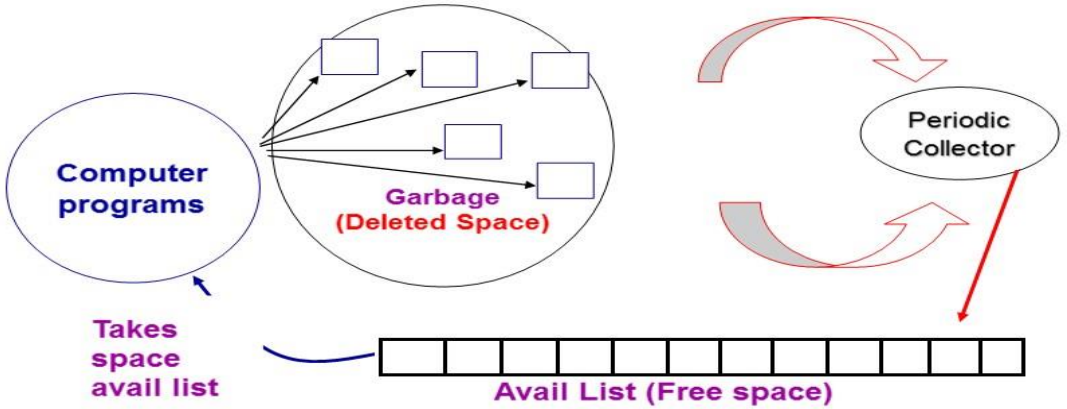
- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

Garbage Collection

- Suppose some memory space becomes reusable because a node is deleted from a list or an entire list is deleted from a program. So space is need to be available for future use.
- One way to bring this is to immediately reinsert the space into the free-storage list. However, this method may be too time-consuming for the operating system of a computer, which may choose an alternative method, as follows.
- The operating system of a computer may periodically collect all the deleted space onto the freestorage list. Any technique which does this collection is called garbage collection

Garbage collection takes place in two steps

- First the computer runs through all lists, tagging those cells which are currently in use
- And then the computer runs through the memory, collecting all untagged space onto the free-storage list.
- The garbage collection may take place when there is only some minimum amount of space or no space at all left in the free-storage list, or when the CPU is idle and has time to do the collection.



Overflow

- Sometimes new data are to be inserted into a data structure but there is no available space, i.e., the free-storage list is empty. This situation is usually called ***overflow***.
- The programmer may handle overflow by printing the message OVERFLOW. In such a case, the programmer may then modify the program by adding space to the underlying arrays.
- Overflow will occur with linked lists when AVAIL = NULL and there is an insertion.

Underflow

- The term underflow refers to the situation where one wants to delete data from a data structure that is empty.
- The programmer may handle underflow by printing the message UNDERFLOW.
- The underflow will occur with linked lists when START = NULL and there is a deletion.

SLL operations

- **Creation**
- **Insertion** – Adds an element at the beginning/ middle/end of the list.
- **Deletion** – Deletes an element at the beginning /end/middleof the list.
- **Traversing**– Displays the complete list.
- **Search** – Searches an element using the given key.

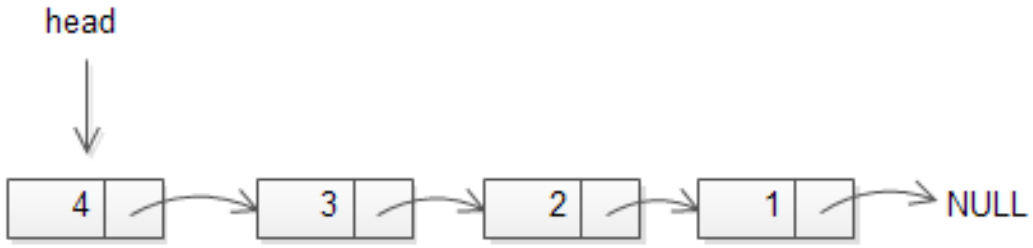
Traversing the list

In all of the examples, we will assume that the linked list has three nodes 1 --->2 --->3

```
void display(struct node* head)
{
struct node *temp = head;
printf("\n\nList elements are - \n");
while(temp != NULL)
{
printf("%d --->",temp->data);
temp = temp->next;
}
}
```

Add to beginning

```
void insertAtFront(struct node** headRef, int value)
{
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
newNode->next = head;
head = newNode;
*headRef = head;
}
```



Add to end

```
void insertAtEnd(struct node* head, int value)
{
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = value;
newNode->next = NULL;
struct node *temp = head;
while(temp->next != NULL){
temp = temp->next;
}
temp->next = newNode;
}
```

Add to middle


```
void insertAtMiddle(struct node *head, int position, int value)
{
struct node *newNode;
newNode = malloc(sizeof(struct node));
newNode->data = 4;
struct node *temp = head;
for(int i=2; i < position; i++)
 {
     if(temp->next != NULL)
      {    temp = temp->next;  }
}
newNode->next = temp->next;
temp->next = newNode;
}
```

Delete from beginning

```
void deleteFromFront(struct node** headRef)
{       struct node* head = *headRef;
        head = head->next;
        *headRef = head;   }
```

Delete from end

```
void deleteFromEnd(struct node* head)
{
struct node* temp = head;
while(temp->next->next!=NULL){
temp = temp->next;
}
temp->next = NULL;
}
```

Delete from middle

```
void deleteFromMiddle(struct node* head, int position)
{struct node* temp = head;
int i;
for(int i=2; i< position; i++)
 {
        if(temp->next!=NULL)
         {
           temp = temp->next;
         }
}
temp->next = temp->next->next;
}
```

Search an element

```
int search(struct node *head, int key)
{
        while (head != NULL)
        {
           if (head->data == key)
          {
            return 1;
          }
         head = head->next;
        }
  return 0;
}
```

Creation of list by adding at end

```
void create(struct node **head)
{
    int c, ch;
    struct node *temp, *rear;
   do
   {
    printf("Enter number: ");
    scanf("%d", &c);
    temp = (struct node *)malloc(sizeof(struct node));
    temp->num = c;
     temp->next = NULL;
    if (*head == NULL)
    {
        *head = temp;
    }
```

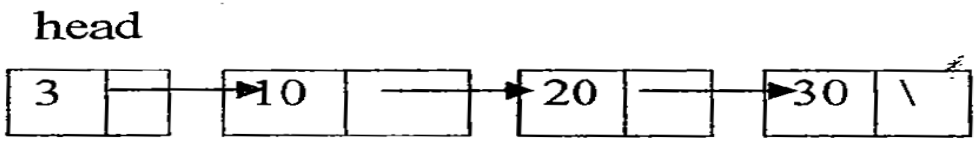
```
else
  {
    rear->next = temp;
  }
  rear = temp;
  printf("Do you wish to continue [1/0]: ");
  scanf("%d", &ch);
   } while (ch != 0);
  printf("\n");
  }
```

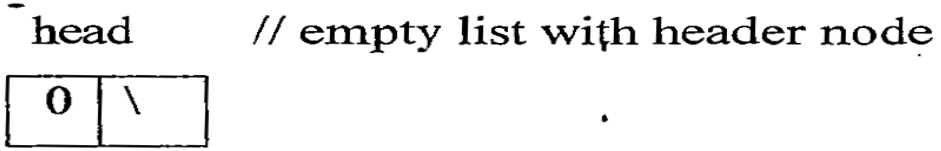
Reverse a singly linked list

```
void reverseList()
{       struct node *prevNode, *curNode;
        if(head != NULL)
        {       prevNode = head;
                curNode = head->next;
                 head = head->next;
                prevNode->next = NULL; // Make first node as last node
                while(head != NULL)
                {       head = head->next;
                        curNode->next = prevNode;
                        prevNode = curNode;
                         curNode = head;
                 }
        head = prevNode;     // Make last node as head
        printf("SUCCESSFULLY REVERSED LIST\n");
        }
}
```

Header Nodes

- Some times it is desirable to keep an extra node at the front of a list which simplifies the design process of some list operations.
- Such a node does not represent an item in the list and is called a header node or a list header.
- The info field of such a header node generally contains the global information of the entire list, like the number of nodes in the list.





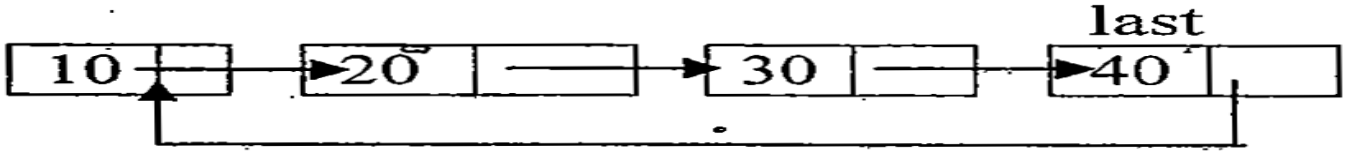
- Info field of head contains the total number of nodes in the list.
- Each time a node is added or deleted, the count in this field (i.e info field of the header) must be readjusted so as to contain actual number of nodes currently present which is certainly a overhead.

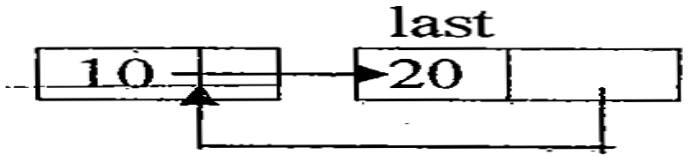
Note:

- If the list is empty, link field of the header node points to NULL. Otherwise, link field of the header node contains the address of the first node of the list and the link field of the last node contains NULL.
- Given a list with header node, any node· in the list can be accessed without the need for a pointer variable (first, which points to the first node as in before example).

Circular Singly Linked List

- In normal linked list The link part of the last node has NULL value, specifying that it is the last node.
- In a circular singly linked list the link field of the last node points to· the first node in the list, which makes the list a circular one.

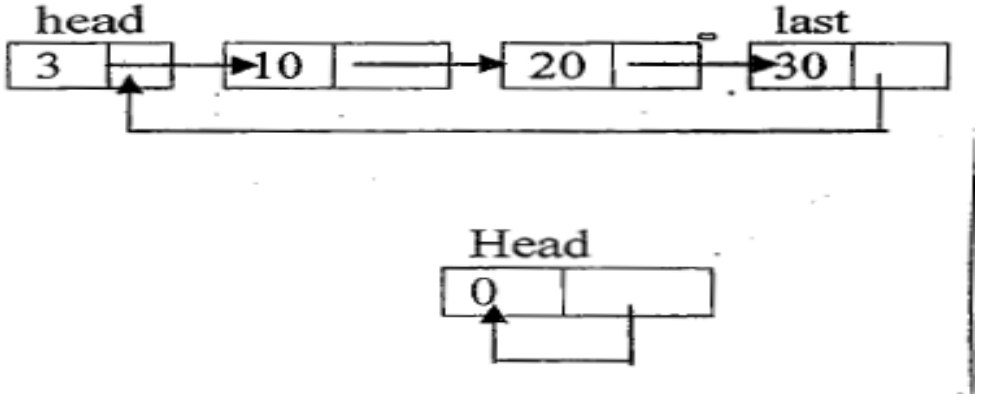




- Here the link field of the last node contains address of the first node .

Circular Singly Linked List with header node

- Here if the list is empty, link of head contains head.
- otherwise link field of header node contains address of the first node.
- The link field of last node contains address of header node



```
NODE insert_front(int item, NODE head)
{
        NODE temp;
        temp=(NODE)malloc(sizeof(struct node));
        temp->info = item;
        temp->link = head->link;
        head->link = temp;
        head->info = head->info +.1;
        return head;
}
```

Insert an element at rear end

```
NODE insert_rear(int item, NODE head)
{
        NODE temp, cur;
        temp=(NODE)malloc(sizeof(struct node));
        temp->info = item;
        cur = head->link;
        while(cur->link != head)
        {
                cur = cur->link;
        }
        cur->link = temp;
        temp->link = head;
        head->info = head->info + 1;
        return head;
}
```

Delete an element at front end

```
NODE delete_front(NODE head)
{
        NODE cur;
        if(head->link = = head)        // no node
        {
                printf("list is empty\n");
                return head;
        }
        cur = head->link;
        head->link = cur->link;
        printf("the deleted item is %d", cur->info);
        free(cur);
        head->info = head->info - 1;
        return(head);
}
```

Delete rear element

```
NODE delete_rear(NODE head)
{
        NODE prev, cur;
        if(head->link = = head)        // no node
        {
                printf("list is empty\n");
                return head;
        }
        prev = head;
        cur = head->link;
        while(cur->link != head)
        {
                prev = cur;
                cur = cur->link;
        }
        prev->link = head;
        printf("the deleted item is %d", cur->info);
        free(cur);
        head->info = head->info - 1;
        return(head);
}
```

ISE Dept.
Transform Here

Display the elements

```
void display(NODE head)
{
        NODE cur;
        if(head->link == head)
        {
                printf("the list is empty\n");
                return;
        }
        cur = head->link;
        printf("the contents of the list are:\n");
        while(cur != head)
        {
                printf("%d\n",cur->info);
                cur = cur->link;
        }
}
```

Doubly linked list (2 way list)

Disadvantages of Singly Linked List

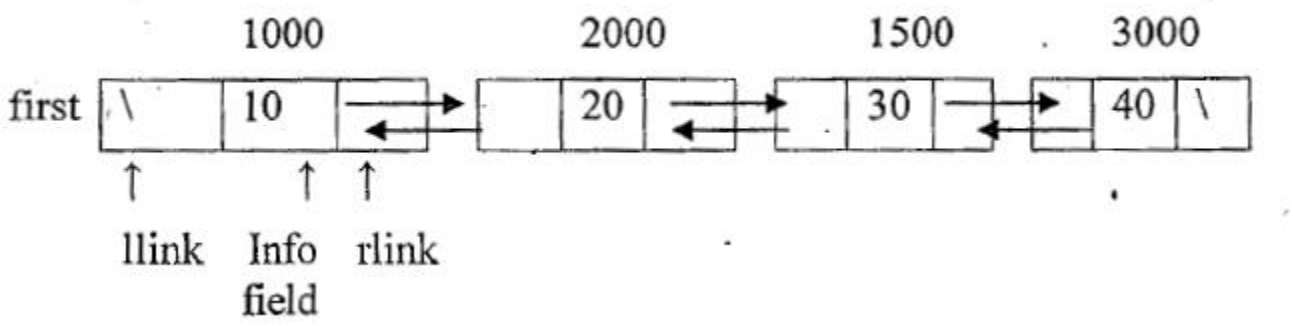
- Given the address of a node in the list, it is difficult to find the address of previous node .
- Traversing of list is possible only in the forward direction and hence singly linked list is also termed as one-way list.

Note:

- To increase the performance and efficiency of algorithms, it is required to traverse the list in either forward or backward direction

Doubly Linked List Representation

- A list where each node has 2 links
  - Left link (llink)- Contains the address of the left node
  - Right link (rlink)- Contains the address of the right node



Structure definition

```
struct node
{
int info;
struct node * llink;
struct node *rink;
}
typedefstruct node* NODE;
```

Disadvantages of Doubly Linked List

- Each node in the list requires an extra link and hence more memory is consumed .
- If a node is inserted or deleted both llink and rlink should be manipulated

DLL- insert at front end

```
NODE insertfront(int item, NODE first)
{
        NODE temp;
        temp = (NODE) malloc(sizeof(struct node));
        temp->info = item;
        temp->llink = NULL;
        temp->rlink = NULL;
        if(first == NULL)
                return temp;
        temp->rlink = first;
        first->llink = temp;
        return temp;
}
```

DLL- Insert at rear end

```
NODE insertrear(int item, NODE first)
{
        NODE temp, cur;
        temp = (NODE) malloc(sizeof(struct node));
        temp->info = item;
        temp->llink = NULL;
        temp->rlink = NULL;
        if(first == NULL)
                return temp;
        cur = first;
        while(cur->rlink != NULL)
                cur = cur->rlink;
        cur->rlink = temp;
        temp->llink = cur;
        return first;
}
```

```
NODE deletefront( NODE first)
{
        NODE cur;
        if(first = = NULL)
        {
                printf("list is empty\n");
                return first;
        }
        if(first->rlink = = NULL)
        {
                printf("item deleted is %d\n", first->info);
                free(first)
                return NULL;
        }
        cur = first;
       first =first->rlink;
       printf("item deleted is %d\n", cur->info);
       free(cur);
       frist->llink = NULL;
       return first;
}
```

DLL-delete at rear end

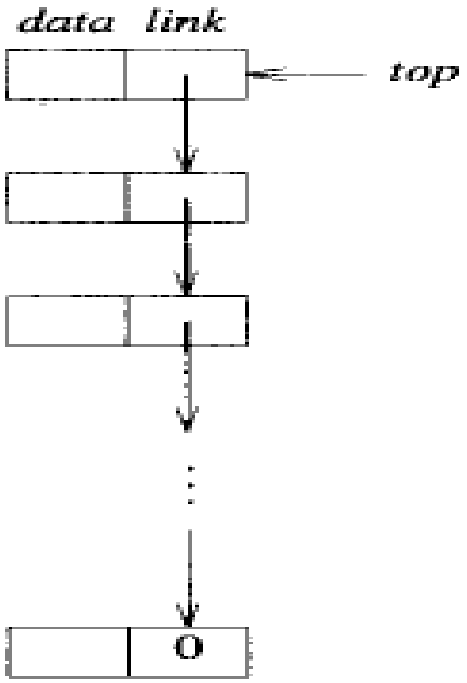
```
NODE deleterear( NODE first)
{
        NODE prev, cur;
        if(first == NULL)
        {
                printf("list is empty\n");
                return first;
        }
        if(first->rlink == NULL)
        {
                printf("item deleted is %d\n", first->info);
                free(first)
                return NULL;
        }
        cur = first;
        while(cur->rlink != NULL)
                cur = cur->rlink;
        prev =cur->llink;
        printf("item deleted is %d\n", cur->info);
        free(cur);
        prev->rlink = NULL;
        return first;
}
```

DLL- Display

```
void display(NODE first)
{
        NODE cur;
        if(first == NULL)
        {
                printf("the list is empty\n");
                return;
        }
        printf("the contents of the list are:\n");
        cur = first;
        while(cur != NULL)
        {
                printf("%d\n",cur->info);
                cur = cur->rlink;
        }
}
```

Linked stacks

- Implementing a stack using a linked list is particularly easy because all accesses to a stack are at the top.
- The main advantage of using linked list over an arrays is that
  - it is possible to implements a stack that can shrink or grow as much as needed.
  - Here each new node will be dynamically allocate. so overflow is not possible.



(a) Linked stack

Stack Operations:

- **push() :** Insert the element into linked list nothing but which is the top node of Stack.
- **pop() :** Return top element from the Stack and move the top pointer to the second node of linked list or Stack.
- **peek():** Return the top element.
- **display():** Print all element of Stack.

Multiple stacks using linked list

```
#define MAX_STACKS 10 /* maximum number of stacks */
typedef struct stack *stackPointer;
typedef struct
 {
int data;
stackPointer link;
} stack;
stackPointer top[MAX_STACKS];
```

The initial condition for the stacks is:

$$top[i] = NULL,\ 0 \le i < MAX\_STACKS$$

The boundary condition is: top [i] = NULL iff the ith stack is empty

Add element to linked stack

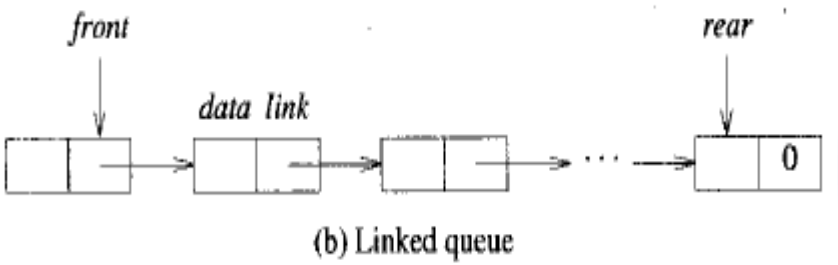
```
void push(int i, element item)
    { /* add item to the ith stack */
        stackPointer temp;
        MALLOC(temp, sizeof(*temp));
        temp→data = item;
        temp→link = top[i];
        top[i] = temp;
     }
```

Pop element from linked stack

```
element pop(int i)
{ /* remove top element from the ith stack */
        stackPointer temp = top[i];
        element item;
        if (! temp)
        return stackEmpty();
        item = temp→data;
        top[i] = temp→link;
        free (temp) ;
        return item;
}
```

Linked Queue

```
#define MAX-QUEUES 10
/* maximum number of queues */
Typedef struct queue *queuePointer;
typedef struct {
        int data;
        queuePointer link;
} queue;
queuePointer front[MAX_QUEUES], rear[MAX_QUEUES];
```



(b) Linked queue

The initial condition for the queues is:

front[i] = NULL, $0 \leq i < MAX\_QUEUES$

The boundary condition is:

front[i] = NULL iff the ith queue is empty

Add to the rear of a linked queue

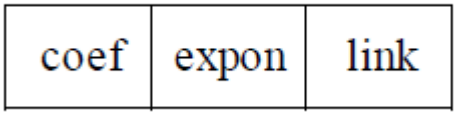
```
void addq(i, item)
{ /* add item to the rear of queue i */
        queuePointer temp;
        MALLOC(temp, sizeof(*temp));
        temp→data = item;
        temp→link = NULL;
        if (front[i])
                rear[i] →link = temp;
        else
                front[i] = temp;
                rear[i] = temp;
}
```

Delete from the front of a linked queue

```
element deleteq(int i)
{ /* delete an element from queue i */
        queuePointer temp = front[i];
        element item;
        if (! temp)
                return queueEmpty();
        item = temp→data;
        front[i]= temp→link;
        free (temp) ;
        return item;
}
```

APPLICATIONS OF LINKED LISTS – POLYNOMIALS

```
typedef struct polyNode *polyPointer;
typedef struct
{
        int coef;
        int expon;
        polyPointer link;
} polyNode;
polyPointer a,b;
```



Repr of polynomials using linked list

Figure shows how we would store the polynomials

$$a = 3x^{14} + 2x^8 + 1$$

and

$$b = 8x^{14} - 3x^{10} + 10x^6$$

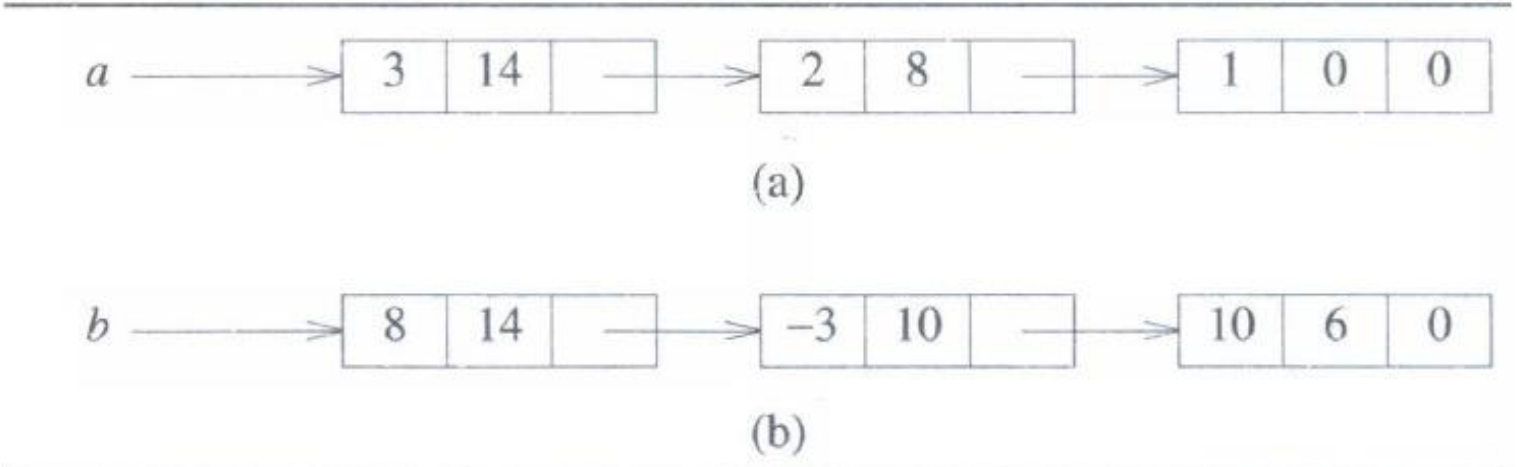


Figure: Representation of $3x^{14}+2x^8+1$ and $8x^{14}-3x^{10}+10x^6$

```
polyPointer padd(polyPointer a, polyPointer b)
{/* return a polynomial which is the sum of a and b */
   polyPointer c, rear, temp;
   int sum;
   MALLOC(rear, sizeof(*rear));
   c = rear;
   while (a && b)
      switch (COMPARE(a→expon,b→expon)) {
         case -1: /* a→expon < b→expon */
                  attach(b→coef,b→expon,&rear);
                  b = b→link;
                  break;
         case 0: /* a→expon = b→expon */
                  sum = a→coef + b→coef;
                  if (sum) attach(sum,a→expon,&rear);
                  a = a→link;  b = b→link; break;
         case 1: /* a→expon > b→expon */
                  attach(a→coef,a→expon,&rear);
                  a = a→link;
      }
   /* copy rest of list a and then list b */
   for (; a; a = a→link) attach(a→coef,a→expon,&rear);
   for (; b; b = b→link) attach(b→coef,b→expon,&rear);
   rear→link = NULL;
   /* delete extra initial node */
   temp = c; c = c→link;  free(temp);
   return c;
}
```

```
void attach(float coefficient, int exponent,
            polyPointer *ptr)
{/* create a new node with coef = coefficient and expon =
    exponent, attach it to the node pointed to by ptr.
    ptr is updated to point to this new node */
   polyPointer temp;
   MALLOC(temp, sizeof(*temp));
   temp→coef = coefficient;
   temp→expon = exponent;
   (*ptr)→link = temp;
   *ptr = temp;
}
```

Program : Attach a node to the end of a list

Sparse Matrix Representation

- In data representation, each column of a sparse matrix is represented as a circularly linked list with a header node. A similar representation is used for each row of a sparse matrix.
- Each node has a tag field, which is used to distinguish between header nodes and entry nodes.

Header Node:

- Each header node has three fields: down, right, and next as shown in figure (a).
- The down field is used to link into a column list and the right field to link into a row list.
- The next field links the header nodes together.
- The header node for row i is also the header node for column i, and the total number of header nodes is max {number of rows, number of columns.}

- **Element node:**
  - Each element node has five fields in addition in addition to the tag field: row, col, down, right, value as shown in figure (b).
  - The down field is used to link to the next nonzero term in the same column and the right field to link to the next nonzero term in the same row. Thus, if aij ≠ 0, there is a node with tag field = entry, value = aij, row = i, and col = j as shown in figure (c).
  - We link this node into the circular linked lists for row i and column j. Hence, it is simultaneously linked into two different lists.

Figure: Node structure for sparse matrices

- Consider the sparse matrix, as shown in below figure (2).

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

Figure (2): 4×4 sparse matrix a

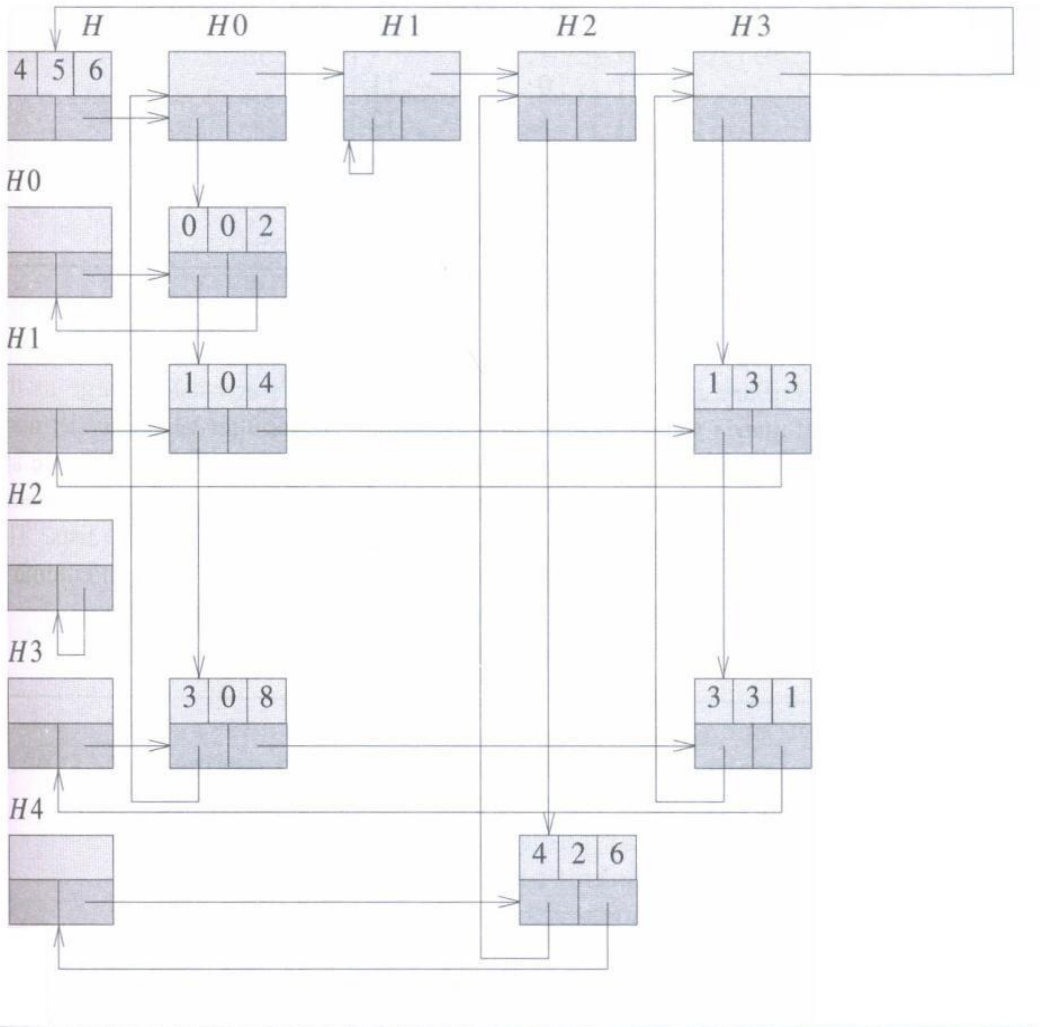


Figure (3) : Linked representation of the sparse matrix of Figure (2) (the *head* field of a node is not shown)

```
#define MAX-SIZE 50 /*size of largest matrix*/
typedef enum {head, entry} tagfield;
typedef struct matrixNode *matrixPointer;
 typedef struct
{
            int row;
            int col;
            int value;
    } entryNode;
 typedef struct
{
   matrixPointer down;
   matrixPointer right;
   tagfield tag;
   union
     {
     matrixPointer next;
      entryNode entry;
     } u;
  } matrixNode;
matrixPointer hdnode[MAX-SIZE];
```