4 Static and Dynamic List

We have studied that an array is a linear collection of data elements in which the elements are stored in consecutive memory locations. While declaring arrays, we have to specify the size of the array, which will restrict the number of elements that the array can store. For example, if we declare an array as int marks[10], then the array can store a maximum of 10 data elements but not more than that. But what if we are not sure of the number of elements in advance? Moreover, to make efficient use of memory, the elements must be stored randomly at any location rather than in consecutive locations. So, there must be a data structure that removes the restrictions on the maximum number of elements and the storage condition to write efficient programs.

Linked list is a data structure that is free from the aforementioned restrictions. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it. However, unlike an array, a linked list does not allow random access of data. 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

- **Self-referential Structure**

It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type. Hence, a structure which contains a reference to itself is called self-referential structure. In general terms, this can be expressed as:

```
struct node
{
        member 1;
        member 2;
        …….
        struct node *ptr;
};
```

For example:

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

This is a structure of type node. The structure contains two members: a info integer member, and a pointer to a structure of the same type (i.e., a pointer to a structure of type node), called next. Therefore this is a self-referential structure.

- **LIST**

- A list is a sequential data structure. It differ from the stack and queue data structures in that additions and removals can be made at any position in the list.

4.1 Static Implementation of List

- Static implementation can be implemented using arrays. It is very simple method but it has Static implementation. Once a size is declared, it cannot be change during the program execution. It is also not efficient for memory utilization. When array is declared, memory allocated is equal to the size of the array. The vacant space of array also occupiers the memory space. In both cases, if we store fewer arguments than declared, the memory is wasted and if more elements are stored than declared, array cannot be expanded. It is suitable only when exact numbers of elements are to be stored.
- **Contiguous list:** If the elements are stored such that they share common adjacent memory boundary as in case of array then the list is known as contiguous list.

  Note:
  An array stores elements in successive order in memory. This means, elements are one after another consecutive in memory.
- The position of each element is given by an index from 0 to n-1, where n is the number of elements.
- Given any index, the element with that index can be accessed in constant time, i.e. the time to access does not depend on the size of the list.

- To add an element at the end of the list, the time taken does not depend on the size of the list. However, the time taken to add an element at any other point in the list does depend on the size of the list, as all subsequent element must be shifted up. Addition near the start of the list take longer than additions near the middle or end.
- When an element is removed, subsequent elements must be shifted down, so removals near the start of the list take longer than the removals near the middle or end.

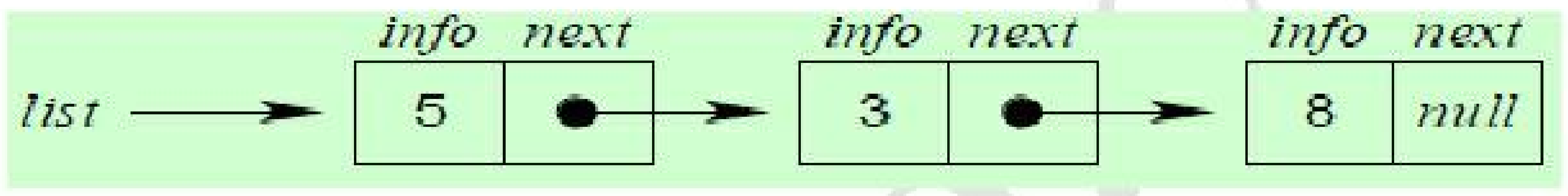
4.2 Dynamic Implementation of List

In static implementation of memory allocation, we cannot alter (increase or decrease) the size of an array and the memory allocation is fixed. So we have to adopt an alternative strategy to allocate memory only when it is required. There is a special data structure called linked list that provides a more flexible storage system and it does not required the use of array. **The advantage of a list over an array occurs when it is necessary to insert or delete an element in the middle of a group of other elements**.

4.3 Linked List

- **Definition**:
  Linear linked list is a data structure of explicit ordering of items and each item contains two portions; one is information portion and the other one is next address portion.
  In explicit ordering of items, each item contained within itself the address of another item. Such an explicit ordering gives rise to a data structure which is known as a linear linked list.

- A linked list is a collection of nodes, where each node consists of two parts:
  - **info**: the actual element to be stored in the list. It is also called data field.
  - **link**: one or two links that points to next and previous node in the list. It is also called next or pointer field.



  - The nodes in a linked list are not stored contiguously in the memory
  - You don't have to shift any element in the list.
  - Memory for each node can be allocated dynamically whenever the need arises.
  - The size of a linked list can grow or shrink dynamically

Advantages of Linked List
- **Linked lists are dynamic data structures.** That is, they can grow or shrink during the execution of a program
- **Efficient memory utilization.** Here, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (removed) when it is no longer needed.
- **Insertion and deletions are easier and efficient:** Linked lists provide flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
- Many complex applications can be easily carried out with linked lists.

Disadvantages of Linked List
- More memory: If the number of fields are more, then more memory space is needed.
- Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists efficiently with its default implementation.
- Extra memory space for a pointer is required with each element of the list.

4.4 Operations on linked list

The basic operations to be performed on the linked list are as follows:

- **Creation**: This operation is used to create a linked list
- **Insertion**: This operation is used to insert a new nose in a kinked list in a specified position. A new node may be inserted
  - ✓ At the beginning of the linked list
  - ✓ At the end of the linked list
  - ✓ At the specified position in a linked list

- **Deletion**: The deletion operation is used to delete a node from the linked list. A node may be deleted from
  - ✓ The beginning of the linked list
  - ✓ the end of the linked list
  - ✓ the specified position in the linked list.
- **Traversing**: The list traversing is a process of going through all the nodes of the linked list from one end to the other end. The traversing may be either forward or backward.
- **Searching** or **find**: This operation is used to find an element in a linked list. In the desired element is found then we say operation is successful otherwise unsuccessful.
- **Concatenation**: It is the process of appending second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1) th node in A. After concatenation A will contain (n+m) nodes
- **Display**: This operation is used to print each and every node's information.

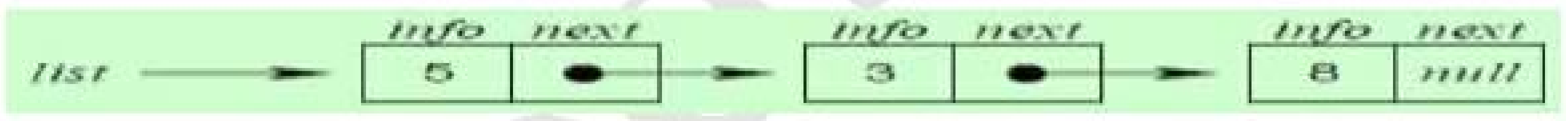
4.5 Types of Linked List

Basically we can put linked list into the following four types:

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

4.5.1 Singly Linked List

➢ A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made. In this type of linked list each node contains two fields one is info field which is used to store the data items and another is link field that is used to point the next node in the list. The last node has a NULL pointer.



➢ We can create a structure for the singly linked list the each node has two members, one is info that is used to store the data items and another is next field that store the address of next node in the list.

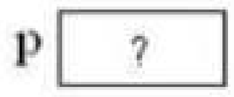
```
struct Node
{
        int info;
        struct Node *next;
};

typedef struct Node Lnode;
Lnode  *START;                    //START is a pointer type structure variable
```

Creating a Node:

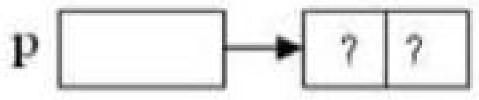
- ✓ To create a new node, we use the **malloc** function to dynamically allocate memory for the new node.
- ✓ After creating the node, we can store the new item in the node using a pointer to that node.

The following steps clearly shows the steps required to create a node and storing an item

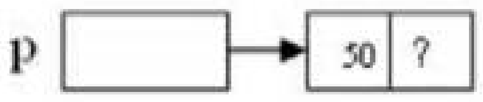
```
Lnode *p;
```



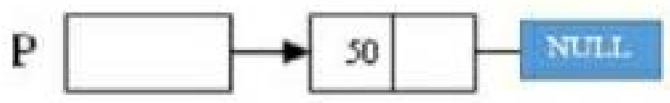
```
p = (Lnode*)malloc(sizeof(Lnode));
```



```
p->info = 50;
```



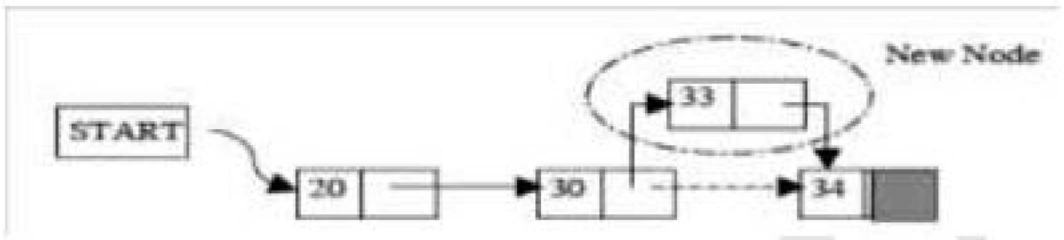
```
p->next = NULL;
```



Note that **p** is not a node; instead it is a pointer to a node.

4.5.1.1 Inserting Nodes

- To insert an element or a node in a linked list, the following three things to be done:
  - Allocating a node
  - Assigning a data to info field of the node
  - Adjusting the pointer.
- Inserting a new node into the linked list has the following three instances:
  - Insertion at the beginning of the list
  - Insertion at the end of the list
  - Insertion at the specified position within the list



Algorithm to insert a node at the beginning of the singly linked list:

Let ***START** be the pointer to first node in the current list

```
1. Input DATA in newItem to be inserted
2. Create a new node from memory and assign its address to NewNode
        NewNode=(Lnode*)malloc(sizeof(Lnode));
3. Assign data to the info field of new node
        NewNode->info=newItem;
4. Set next of new node to START
        NewNode->next=START;
5. Set the START pointer to the new node
        START=NewNode;
6. End
```

Algorithm to insert a node at the end of the singly linked list:

Let ***START** be the pointer to first node in the current list and **TEMP** is a temporary pointer to hold the node address

```
1. Input DATA in newItem to be inserted
2. Create a new node from memory and assign its address to NewNode
        NewNode=(Lnode*)malloc(sizeof(Lnode));
3. Assign data to the info field of new node and Set NULL to the next  of new node
        NewNode->info=newItem;
        NewNode -> next = NULL
4. If (SATRT equal to NULL)
        START = NewNode
   Else
        TEMP = START
        While (TEMP -> next not equal to NULL)
            TEMP = TEMP->next
        TEMP ->next = NewNode
5. Exit
```

Algorithm to insert a node at the specified position in a singly linked list:

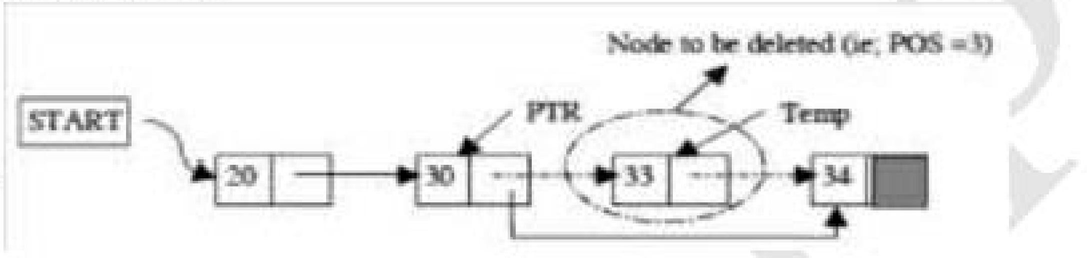
Let ***START** be the pointer to first node in the current list and **TEMP** is a temporary pointer to hold the node address

```
1. Input DATA and POS to be inserted
2. Initialize TEMP = START; and k = 0
3. Repeat the step 3 while (k is less than POS)
        I.   TEMP = TEMP->next
        II.  If (TEMP is equal to NULL)
                a) Display "Node in the list less than the position"
                b) Exit
        III. k = k + 1
```

```
4. Create a new node from memory and assign its address to NewNode
       NewNode=(Lnode*)malloc(sizeof(Lnode));
5. Assign data to the info field of new node and Set TEMP → next to the next  of new node
       NewNode->info=newItem;
       NewNode -> next = TEMP->next
6. TEMP → next = NewNode
7. End
```

4.5.1.2 Deleting Nodes

- Deleting a node from the linked list has the following three instances
  - Deleting the first node of the linked list
  - Deleting the last node of the linked list
  - Deleting the specified node within the linked list



Algorithm to delete the first node of the singly linked list

Let ***START** be the pointer to first node in the current list and **PTR** is a pointer to hold the node address

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 5
            [END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START->next
Step 4: Print, element deleted is PTR->info
Step 5: FREE PTR
Step 6: EXIT
```

Algorithm to delete the last node of the singly linked list

Let ***START** be the pointer to first node in the current list, **PTR** is a pointer to hold the node address and another pointer variable **PREPTR** such that it always points to one node before the **PTR**

```
Step 1: IF START = NULL
            Write UNDERFLOW
            Go to Step 10
            [END OF IF]
Step 2: SET PTR = START
Step 3: If PTR->next = NULL
                SET START = NULL
                Go to Step 8
              [END OF IF]
Step 4: Repeat Steps 5 and 6 while PTR ->next != NULL
Step 5: SET PREPTR = PTR
Step 6: SET PTR = PTR->next
            [END OF LOOP]
Step 7: SET PREPTR->next = NULL
Step 8: Print, element deleted is PTR->info
Step 9: free PTR
Step 10: EXIT
```

Algorithm to delete the node at the specified position of the singly linked list

Let ***START** be the pointer to first node in the current list, **PTR** is a pointer to hold the node address and another pointer variable **PREPTR** such that it always points to one node before the **PTR**

```
1. IF START = NULL
       print UNDERFLOW  and exit
2. Enter position of a node at which you want to delete a new node. Let this position is POS.
```

```
3.  Set PTR=START and k = 0
4.  Repeat the step 4 while (k is less than POS)
        I.    Set PREPTR = PTR
        II.   PTR = PTR->next
        III.  If (PTR is equal to NULL)
                  c)  Display "Node in the list less than the position"
                  d)  Exit
        IV.   k = k + 1

5.  Set PREPTR->next = PTR->next
6.  free PTR
7.  Exit
```

❖ **<u>Algorithm to print the number of nodes in a linked list</u>**

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

❖ **<u>Algorithm for traversing a linked list</u>**

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

❖ **<u>Algorithm to search a linked list</u>**

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Step 3 while PTR != NULL
Step 3: IF VAL = PTR->info
                    SET POS = PTR
                    Go To Step 5
        ELSE
                    SET PTR = PTR->next
                    [END OF IF]
        [END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

In Step 1, we initialize the pointer variable PTR with START that contains the address of the first node. In Step 2, a while loop is executed which will compare every node's DATA(info) with VAL for which the search is being made. If the search is successful, that is, VAL has been found, then the address of that node is stored in POS and the control jumps to the last statement of the algorithm. However, if the search is unsuccessful, POS is set to NULL which indicates that VAL is not present in the linked list.

Representation of Linked list in C

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
  int info;
  struct node *next;
};
typedef struct node Lnode;
Lnode *start = NULL;
void create_linkedlist();
void display();
void insert_beg();
void insert_end();
void insert_position();
void delete_beg();
void delete_end();
void delete_position();
void delete_linkedlist();
void search_linkedlist();
int main()
{
   int option,flag=1;
   printf("\n\n *****MAIN MENU *****");
  printf("\n 1: Create a list");
  printf("\n 2: Display the list");
  printf("\n 3: Add a node at the beginning");
  printf("\n 4: Add a node at the end");
  printf("\n 5: Add a node at nth position");
  printf("\n 7: Delete a node from the beginning");
  printf("\n 8: Delete a node from the end");
  printf("\n 9: Delete nth position node");
  printf("\n 10: Delete the entire list");
  printf("\n 11: Search in the linked list");
  printf("\n 12: EXIT");
  do
  {
          printf("\n\n Enter your option : ");
          scanf("%d", &option);
          switch(option)
          {
                  case 1:
                                  create_linkedlist();
                                  printf("\n LINKED LIST CREATED");
                                  break;
                  case 2:
                                  display();
                                  break;
                  case 3:
                                  insert_beg();
                                  break;
                  case 4:
                                  insert_end();
```

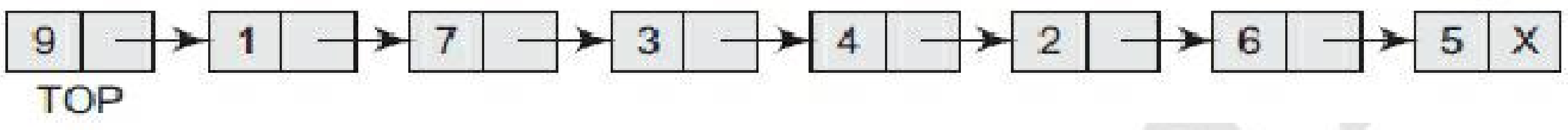
```
                                    break;
                    case 5:
                                    insert_position();
                                    break;
                    case 7:
                                    delete_beg(start);
                                    break;
                    case 8:
                                    delete_end();
                                    break;
                    case 9:
                                    delete_position();
                                    break;
                    case 10:
                                    delete_linkedlist();
                                    printf("\n LINKED LIST DELETED");
                                    break;
                    case 11:
                                    search_linkedlist();
                                    break;
                    case 12:
                               flag = 0;
                               break;
                    default:
                            printf("Invalide Choice");
            }
    }while(flag);
    getch();
    return 0;
}
void create_linkedlist()
{
    Lnode *new_node, *ptr;
    int data,ch;
    do
    {
            printf("\n Enter the data : ");
            scanf("%d",&data);
            new_node = (Lnode*)malloc(sizeof(Lnode));
      new_node->info=data;
            if(start==NULL)
            {
                    new_node->next = NULL;
                    start=new_node;
            }
            else
            {
                    ptr=start;
                    while(ptr->next!=NULL)
                            ptr=ptr->next;
                    ptr->next = new_node;
                    new_node->next=NULL;
            }
    printf("\n Do you want to add more data then press 1:");
    scanf("%d",&ch);
    }while(ch == 1);

}
void display()
{
    Lnode *ptr;
    if(start == NULL)
    {
            printf("\nList is empty\n");
            return;
    }
    ptr = start;
    printf("\nLinked List Data are:\n");
    while(ptr!=NULL)
    {
            printf("\t %d", ptr->info);
            ptr=ptr->next;
    }
}
void insert_beg()
{
    Lnode *new_node;
    int data;
    printf("\n Enter the data : ");
    scanf("%d", &data);
    new_node=(Lnode *)malloc(sizeof(Lnode));
    new_node->info=data;
    new_node->next=start;
    start=new_node;
}
void insert_end()
{
    Lnode *ptr, *new_node;
    int data;
    printf("\n Enter the data : ");
    scanf("%d", &data);
    new_node = (Lnode *)malloc(sizeof(Lnode));
    new_node->info = data;
    new_node->next = NULL;
    if(start == NULL)
            start = new_node;
    else
    {
            ptr = start;
            while(ptr->next != NULL)
                    ptr = ptr -> next;
            ptr->next = new_node;
    }
}
void insert_position()
{
    Lnode *new_node,*ptr,*preptr;
    int data,pos,k;
    printf("\n Enter the data :");
    scanf("%d", &data);
    printf("\n Enter the position at which the data has to be inserted : ");
    scanf("%d", &pos);
    ptr = start;
    k=0;
    while(k<pos)
    {
            ptr=ptr->next;
            if(ptr==NULL)
            {
                    printf("Node in the list less than the position");
                    return;
            }
            k++;
    }
    new_node = (Lnode *)malloc(sizeof(Lnode));
    new_node->info = data;
    new_node->next = ptr->next;
    ptr->next = new_node;
  }

  void delete_beg()
  {
    Lnode *ptr;
    if(start == NULL)
    {
            printf("\nList is empty\n");
            return;
    }
    ptr = start;
    start = start -> next;
    printf("\nDleted data from begining is:%d\n",ptr->info);
    free(ptr);
  }
  void delete_end()
  {
    Lnode *ptr,*preptr;
    ptr = start;
    if(start == NULL)
    {
            printf("\nList is empty\n");
            return;
    }
    while(ptr -> next != NULL)
    {
            preptr = ptr;
            ptr = ptr -> next;
    }
    preptr -> next = NULL;
    printf("\nDleted data from end is:%d\n",ptr->info);
    free(ptr);
  }
  void delete_position()
```

```
{
  Lnode *ptr,*preptr;
  int pos,k;
  if(start == NULL)
  {
          printf("\nList is empty\n");
          return;
  }
  printf("\n Enter the index position of the node which has to be deleted : ");
  scanf("%d",&pos);
  ptr = start;
  for(k=0;k<pos;k++)
  {
          preptr = ptr;
          ptr = ptr->next;
          if(ptr==NULL)
          {
                  printf("Node in the list is less than entered position");
                  return;
          }
  }
  preptr->next = ptr->next;
  printf("\n Deleted data at position %d is %d",pos,ptr->info);
  free(ptr);
}
void delete_linkedlist()                    //Delete complete linked list
{
  Lnode *ptr;
  if(start == NULL)
  {
          printf("\nList is empty\n");
          return;
  }
  while(start != NULL)
  {
          ptr=start;
          start = start->next;
          printf("\nDeleted data is %d\n", ptr->info);
          free(ptr);
  }
}
void search_linkedlist()
{
  Lnode *ptr,pos;
  int data;
  printf("\nEnter value you want to search:");
  scanf("%d",&data);
  ptr=start;
  while(ptr != NULL)
  {
          if(ptr->info == data)
          {
                  printf("\n%d found at address %u\n",ptr->info,ptr);
                  return;
```

```
            }
            ptr = ptr->next;
    }
    printf("\nEntered value is not found in the linked list\n");
}
```

4.5.2 Linked List Implementation of Stack

- We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.
- In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.



4.5.2.1 OPERATIONS ON A LINKED STACK

Push Operation

```
Step 1: Allocate memory for the new
        node and name it as NEW_NODE
Step 2: SET NEW_NODE -> DATA = VAL
Step 3: IF TOP = NULL
            SET NEW_NODE -> NEXT = NULL
            SET TOP = NEW_NODE
        ELSE
            SET NEW_NODE -> NEXT = TOP
            SET TOP = NEW_NODE
        [END OF IF]
Step 4: END
```

In Step 1, memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked list. This is done by checking if TOP = NULL. In case the IF statement evaluates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the new node is not the first node in the list, then it is added before the first node of the list (that is, the TOP node) and termed as TOP.

Pop Operation

```
Step 1: IF TOP = NULL
            PRINT "UNDERFLOW"
            Goto Step 5
        [END OF IF]
Step 2: SET PTR = TOP
Step 3: SET TOP = TOP -> NEXT
Step 4: FREE PTR
Step 5: END
```

In Step 1, we first check for the UNDERFLOW condition. In Step 2, we use a pointer PTR that points to TOP. In Step 3, TOP is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool

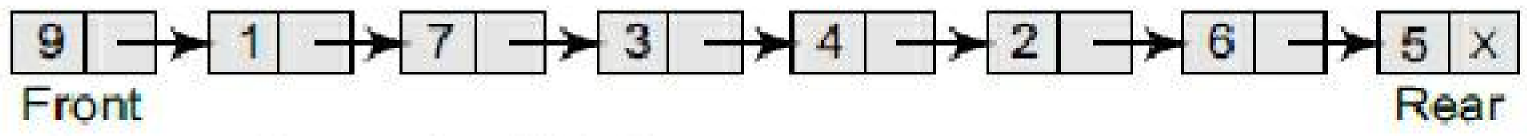
Representation Of Linked Stack in C

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct STACK
{
    int data;
    struct STACK *next;
};
typedef struct STACK stack;
stack *top = NULL;
void push(int);
void display();
void pop();
int main()
{
    int val,option,flag=1;
            printf("\n *****MAIN MENU*****");
            printf("\n 1. PUSH");
            printf("\n 2. POP");
            printf("\n 3. DISPLAY");
            printf("\n 4. EXIT");
    do
    {
            printf("\n Enter your option: ");
            scanf("%d", &option);
            switch(option)
            {
                    case 1:
                                    printf("\n Enter the number to be pushed on stack: ");
                                    scanf("%d", &val);
                                    push(val);
                                    break;
                    case 2:
                                    pop();
                                    break;
                    case 3:
                                    display();
                                    break;
                    case 4:
                                    flag=0;
                                    break;
                    default:
                            printf("Invalid Choice");
            }
    }while(flag);
    getch();
    return 0;
}
void push(int val)
{
    stack *ptr;
    ptr = (stack*)malloc(sizeof(stack));
```

```
        ptr->data = val;
        if(top == NULL)
        {
                ptr->next = NULL;
                top=ptr;
        }
        else
        {
                ptr->next=top;
                top=ptr;
        }
}
void display()
{
        stack *ptr;
        ptr = top;
        if(top == NULL)
                printf("\n STACK IS EMPTY");
        else
        {
                printf("\nData in stack are:\n");
                while(ptr != NULL)
                {
                        printf("\t %d", ptr->data);
                        ptr = ptr->next;
                }
        }
}
void pop()
{
        stack *ptr;
        ptr = top;
        if(top == NULL)
                printf("\n STACK UNDERFLOW");
        else
        {
                top = top->next;
                printf("\n Data being deleted is: %d", ptr->data);
                free(ptr);
        }
}
```

4.5.3 Linked list implementation of queue

- In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue. All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT = REAR = NULL, then it indicates that the queue is empty.



Algorithm to insert an element in a linked queue

```
Step 1: Allocate memory for the new node and name
        it as PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
            SET FRONT = REAR = PTR
            SET FRONT -> NEXT = REAR -> NEXT = NULL
        ELSE
            SET REAR -> NEXT = PTR
            SET REAR = PTR
            SET REAR -> NEXT = NULL
        [END OF IF]
Step 4: END
```

In Step 1, the memory is allocated for the new node. In Step 2, the DATA part of the new node is initialized with the value to be stored in the node. In Step 3, we check if the new node is the first node of the linked queue. This is done by checking if FRONT = NULL. If this is the case, then the new node is tagged as FRONT as well as REAR. Also NULL is stored in the NEXT part of the node (which is also the FRONT and the REAR node). However, if the new node is not the first node in the list, then it is added at the REAR end of the linked queue (or the last node of the queue).

Algorithm to delete an element from a linked queue

```
Step 1: IF FRONT = NULL
            Write "Underflow"
            Go to Step 5
        [END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END
```

In Step 1, we first check for the underflow condition. If the condition is true, then an appropriate message is displayed, otherwise in Step 2, we use a pointer PTR that points to FRONT. In Step 3, FRONT is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

Representation of Linked Queue in C

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
struct Node
{
        int data;
        struct Node *next;
};
typedef struct Node node;
struct Queue
{
        node *front;
        node *rear;
};
typedef struct Queue queue;
queue *q;
```

```
void create_queue();
void insert(int);
void delete_element();
void display();
int main()
{
        int val,option,flag=1;
        create_queue();
                printf("\n *****MAIN MENU*****");
                printf("\n 1. INSERT");
                printf("\n 2. DELETE");
                printf("\n 3. DISPLAY");
                printf("\n 4. EXIT");
        do
        {
                printf("\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                        case 1:
                                printf("\n Enter the number to insert in the queue:");
                                scanf("%d", &val);
                                insert(val);
                                break;
                        case 2:
                                delete_element();
                                break;
                        case 3:
                                display();
                                break;
                        case 4:
                                flag=0;
                                break;
                        default:
                                        printf("\nInvalid Choice\n");
                }
        }while(flag);
        getch();
        return 0;
}
void create_queue()
{
        q = (queue*)malloc(sizeof(queue));
        q ->rear = NULL;
        q ->front = NULL;
}
void insert(int val)
{
        node *ptr;
        ptr = (node*)malloc(sizeof(node));
        ptr->data = val;
        if(q->front == NULL)
        {
                q->front = ptr;
                q->rear = ptr;
                q->front->next=q->rear->next = NULL;
        }
        else
        {
                q->rear->next=ptr;
                q->rear=ptr;
                q->rear->next=NULL;
        }
}
void display()
{
        node *ptr;
        ptr=q->front;
        if(ptr == NULL)
                printf("\n QUEUE IS EMPTY");
        else
        {
                printf("\nData in queue are:\n");
                while(ptr!=q->rear)
                {
                        printf("%d\t", ptr -> data);
                        ptr=ptr->next;
                }
                printf("%d\t", ptr->data);
        }
}
void delete_element()
{
        node *ptr;
        ptr= q->front;
        if(q->front == NULL)
                printf("\n UNDERFLOW");
        else
        {
                q->front= q->front->next;
                printf("\n The value being deleted is : %d", ptr->data);
                free(ptr);
        }
}
```

4.5.3.1 Advantages of Singly Linked List

- Singly linked list can store data in non-contiguous locations. Thus there is no need of compaction of memory, when some large related data is to be stored into the memory.
- Insertion and deletion of values is easier as compared to array, as no shifting of values is involved.

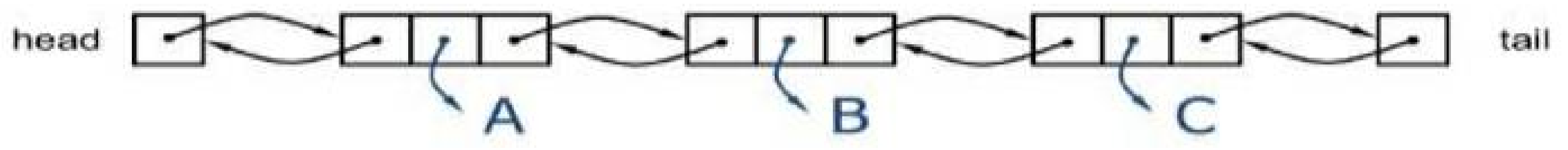
4.5.3.2 Disadvantages of Singly Linked List

- Nodes can only be accessed sequentially. That means, we cannot jump to a particular node directly.
- Because of the above disadvantage, binary search algorithm cannot be implemented on the singly linked list.
- There is no way to go back from one node to previous one. Only forward traversal is possible.

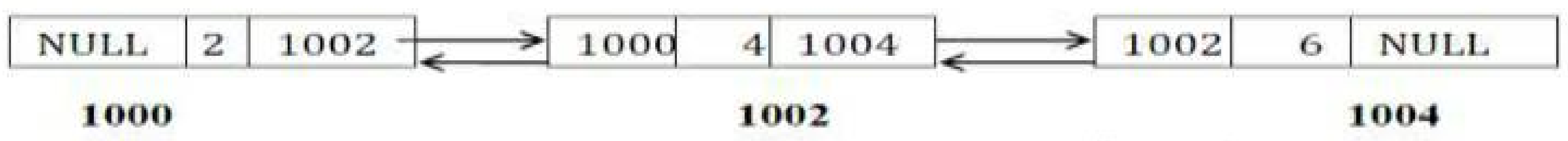
4.5.4 DOUBLY LINKED LIST

- A doubly-linked list is a linked data structure that consists of a set of data records, each having two special link fields that contain references to the previous and to the next record in the sequence. It can be viewed as two singly-linked lists formed from the same data items, in two opposite orders.
- A doubly-linked list whose nodes contain three fields: a data value, the link to the next node, and the link to the previous node.
- The two links allow walking along the list in either direction with equal ease. Compared to a singly-linked list, modifying a doubly-linked list usually requires changing more pointers, but is simpler because there is no need to keep track of the address of the previous node.
- In simpler terms, Doubly linked list
  - Pointers exist between adjacent nodes in both directions.
  - The list can be traversed either forward or backward.
  - Usually two pointers are maintained to keep track of the list, head and tail.

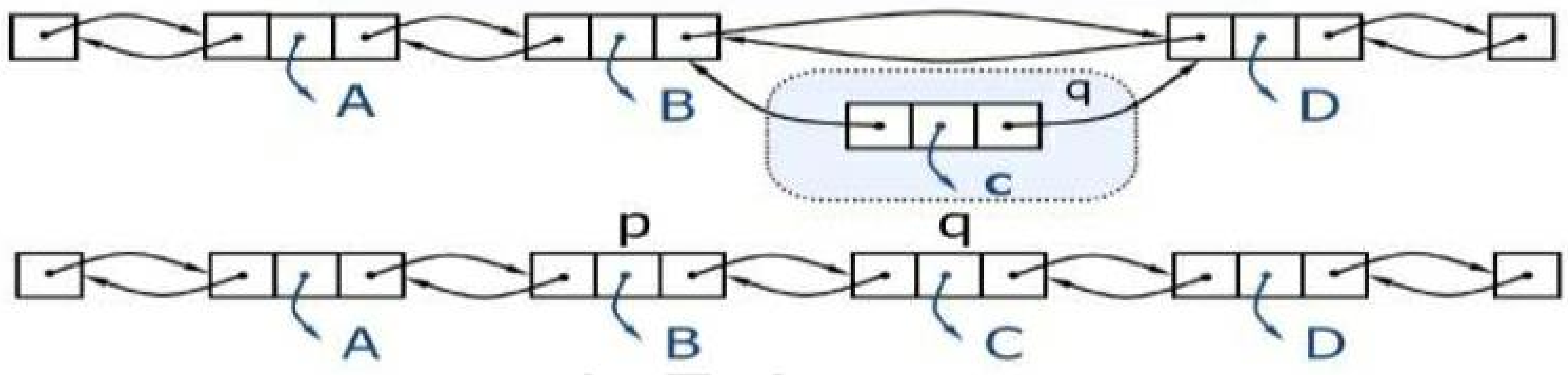
Representation of a doubly linked list



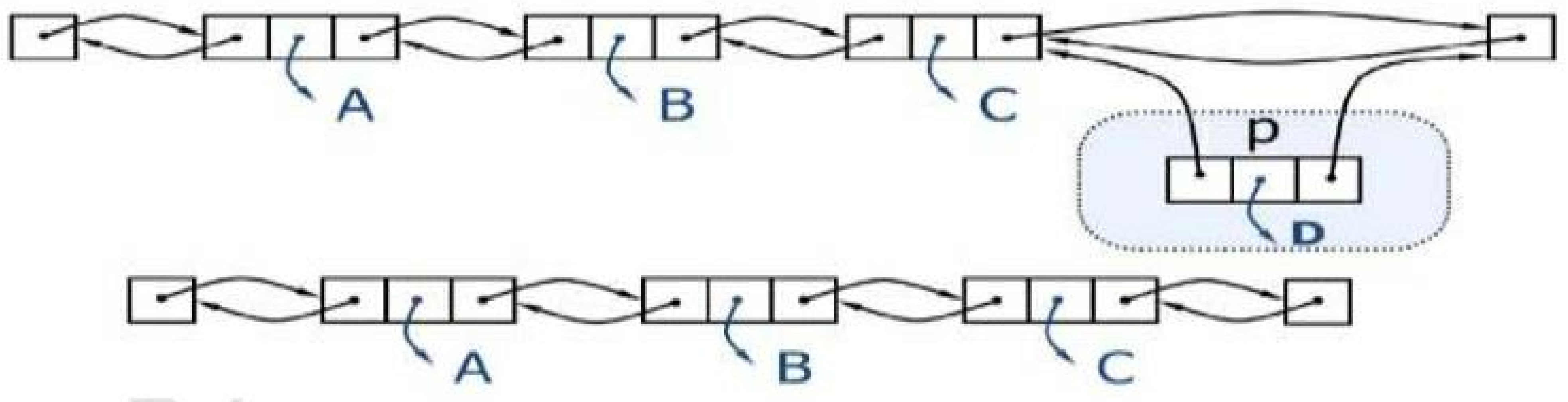
Example



Insertion of node in Doubly Linked List



Deletion of node in Doubly Linked List



- In C, the structure of a doubly linked list can be given as,

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
typedef struct node NodeType;
NodeType *head=NULL:
```

4.5.4.1 Inserting a New Node in a Doubly Linked List

Let * **head** be the pointer to first node in the current list, **PTR** is a pointer to hold the node address.

Algorithm to insert a node at the beginning of a doubly linked list :

```
1. Allocate memory for the new node as,
       newnode=(NodeType*)malloc(sizeof(NodeType))
2. Assign value to data field of a new node
       set newnode->data=item
3. set newnode->prev=newnode->next=NULL
4. set newnode->next=head
5. set head->prev=newnode
6. set head=newnode
7. End
```

Algorithm to insert a node at the end of a doubly linked list :

```
1. Allocate memory for the new node as,
       newnode=(NodeType*)malloc(sizeof(NodeType))
2. Assign value to data field of a new node
       set newnode->data=item
3. set newnode->next=NULL
4. if head==NULL
       set newnode->prev=NULL;
       set head=newnode;
5. if head!=NULL
       set PTR=head
       while(PTR->next!=NULL)
           PTR=PTR->next;
       end while
       set PTR->next=newnode;
       set newnode->prev=PTR
6. End
```

Algorithm to insert a node at the specified position in a doubly linked list :

```
1. Input DATA and POS to be inserted
2. Initialize PTR = head; and k = 0
3. Repeat the step 3 while (k is less than POS)
       I.   PTR = PTR->next
       II.  If (PTR is equal to NULL)
                a) Display "Node in the list less than the position"
                b) Exit
       III. k = k + 1
4. Create a new node from memory and assign its address to NewNode
       newnode =( NodeType *)malloc(sizeof(NodeType));
5. Assign DATA to the data field of new node and Set PTR → next to the next  of new node, PTR to
   the prev of newnode
       newnode ->data=newItem;
       newnode -> next = PTR->next
       newnode -> prev = PTR
6. PTR → next = NewNode
7. End
```

4.5.4.2 Deleting a Node from a Doubly Linked List

Let *__head__ be the pointer to first node in the current list, **PTR** is a pointer to hold the node address

- **Algorithm to delete a node from beginning of a doubly linked list:**

```
1. if head==NULL then
          print "empty list" and exit
   else
          set PTR=head
          set head=head->next
          set head->prev=NULL;
          free(PTR)
2. End
```

- **Algorithm to delete a node from end of a doubly linked list:**

```
1. if head==NULL then
          print "empty list" and exit
   else if(head->next==NULL) then
          set PTR=head
          set head=NULL
          free(PTR)
   else
          set PTR=head;
          while(PTR->next!=NULL)
                 PTR=PTR->next
          end while
          set PTR->prev->next=NULL
          free(PTR)
2. End
```

- **Algorithm to delete the node at the specified position of the doubly linked list:**

```
1. if head==NULL then
       print "empty list" and exit
2. Enter position of a node at which you want to delete a new node. Let this position is POS.
3. Set PTR=head and k = 0
4. Repeat the step 4 while (k is less than POS)
       I.   PTR = PTR->next
       II.  If (PTR is equal to NULL)
              a) Display "Node in the list less than the position"
              b) Exit
       III. k = k + 1

5. Set PTR->prev->next = PTR->next
6. Set PTR->next->prev = PTR->prev
7. free PTR
8. Exit
```

- **Representation of Doubly Linked List in C**

```c
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct Node
{
        int info;
        struct Node *next;
        struct Node *prev;
};
```

```
typedef struct Node node;
node *head = NULL;
void create_linkedlist();
void display();
void insert_beg();
void insert_end();
void insert_position();
void delete_beg();
void delete_end();
void delete_position();
int main()
{
  int option,flag=1;
  printf("\n\n *****MAIN MENU *****");
        printf("\n 1: Create a list");
        printf("\n 2: Display the list");
        printf("\n 3: Add a node at the beginning");
        printf("\n 4: Add a node at the end");
        printf("\n 5: Add a node at nth position");
        printf("\n 7: Delete a node from the beginning");
        printf("\n 8: Delete a node from the end");
        printf("\n 9: Delete nth position node");
        printf("\n 10: EXIT");
        do
        {
                printf("\n\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                        case 1:
                                        create_linkedlist();
                                        printf("\n LINKED LIST CREATED");
                                        break;
                        case 2:
                                        display();
                                        break;
                        case 3:
                                        insert_beg();
                                        break;
                        case 4:
                                        insert_end();
                                        break;
                        case 5:
                                        insert_position();
                                        break;
                        case 7:
                                        delete_beg();
                                        break;
                        case 8:
                                        delete_end();
                                        break;
                        case 9:
                                        delete_position();
                                        break;
```

```
                    case 10:
                            flag = 0;
                            break;
                    default:
                            printf("Invalide Choice");
                }
        }while(flag);
        getch();
        return 0;
}
void create_linkedlist()
{
        node *new_node, *ptr;
        int data,ch;
        do
        {
                printf("\n Enter the data : ");
                scanf("%d",&data);
                new_node = (node*)malloc(sizeof(node));
          new_node->info=data;
          new_node->next = NULL;
                if(head==NULL)
                {
                        new_node->prev = NULL;
                        head=new_node;
                }
                else
                {
                        ptr=head;
                        while(ptr->next!=NULL)
                                ptr=ptr->next;
                        ptr->next = new_node;
                        new_node->prev=ptr;
                }
        printf("\n Do you want to add more data then press 1:");
        scanf("%d",&ch);
        }while(ch == 1);

}
void display()
{
        node *ptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        ptr = head;
        printf("\nLinked List Data are:\n");
        while(ptr!=NULL)
        {
                printf("\t %d", ptr->info);
                ptr=ptr->next;
        }
```

```
}
void insert_beg()
{
        node *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node=(node *)malloc(sizeof(node));
        new_node->info=data;
        new_node->next=head;
        head->prev=new_node;
        new_node->prev=NULL;
        head=new_node;
}
void insert_end()
{
        node *ptr, *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node = (node *)malloc(sizeof(node));
        new_node->info = data;
        new_node->next = NULL;
        if(head == NULL)
        {
                head->prev = NULL;
                head = new_node;
        }
        else
        {
                ptr=head;
                while(ptr->next!=NULL)
                        ptr=ptr->next;
                ptr->next = new_node;
                new_node->prev=ptr;
        }
}
void insert_position()
{
        node *new_node,*ptr,*preptr;
        int data,pos,k;
        printf("\n Enter the data :");
        scanf("%d", &data);
        printf("\n Enter the index position at which the data has to be inserted : ");
        scanf("%d", &pos);
        ptr = head;
        k=0;
        while(k<pos)
        {
                ptr=ptr->next;
                if(ptr==NULL)
                {
                        printf("Node in the list less than the position");
                        return;
                }
                k++;
        }
        new_node = (node *)malloc(sizeof(node));
        new_node->info = data;
        new_node->next = ptr->next;
        new_node->prev = ptr;
        ptr->next = new_node;
}

void delete_beg()
{
        node *ptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        ptr = head;
        head = head -> next;
        head->prev=NULL;
        printf("\nDleted data from begining is:%d\n",ptr->info);
        free(ptr);
}
void delete_end()
{
        node *ptr,*preptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        else if(head->next == NULL)
        {
                ptr = head;
                head=NULL;
                printf("\nDleted data from end is:%d\n",ptr->info);
                free(ptr);
        }
        else
        {
                ptr = head;
                while(ptr->next != NULL)
                {
                        ptr = ptr->next;
                }
                (ptr->prev)-> next = NULL;
                printf("\nDleted data from end is:%d\n",ptr->info);
                free(ptr);
        }
}
void delete_position()
{
        node *ptr,*preptr;
        int pos,k;
        if(head == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        printf("\n Enter the position of the node which has to be deleted : ");
        scanf("%d",&pos);
        ptr = head;
        for(k=0;k<pos;k++)
        {
                ptr = ptr->next;
                if(ptr==NULL)
                {
                        printf("Node in the list is less than entered position");
                        return;
                }
        }
        (ptr->prev)->next = ptr->next;
        (ptr->next)->prev = ptr->prev;
        printf("\n Deleted data at position %d is %d",pos,ptr->info);
        free(ptr);
}
```

4.5.4.3 Application of Doubly Linked List

- Applications that have an MRU list (a linked list of file names)
- The cache in your browser that allows you to hit the BACK button (a linked list of URLs)
- Undo functionality in Photoshop or Word (a linked list of state)
- A stack, hash table, and binary tree can be implemented using a doubly linked list.
- A great way to represent a deck of cards in a game

4.5.4.4 Advantages of Doubly Linked List

- The primary advantage of a doubly linked list is that given a node in the list, one can navigate easily in either direction.
- This can be very useful, for example, if the list is storing strings, where the strings are lines in a text file (e.g., a text editor).
- One might store the ``current line'' that the user is on with a pointer to the appropriate node; if the user moves the cursor to the next or previous line, a single pointer operation can restore the current line to its proper value.
- Or, if the user moves back 10 lines, for example, one can perform 10 pointer operations (follow the chain) to get to the right line.
- For either of these operations, if the list is singly linked, one must start at the head of the list and traverse until the proper point is reached. This can be very inefficient for large lists.

4.5.4.5 Disadvantages of Doubly Linked List

- Each node requires an extra pointer, requiring more space
- The insertion or deletion of a node takes a bit longer (more pointer operations).

4.5.5 CIRCULAR SINGLY LINKED LIST

- Singly Linked List has a major drawback. From a specified node, it is not possible to reach any of the preceding nodes in the list. To overcome the drawback, a small change is made to the SLL so that the next field of the last node is pointing to the first node rather than NULL. Such a linked list is called a circular linked list.
    - Because it is a circular linked list, it is possible to reach any node in the list from a particular node.
    - There is no natural first node or last node because by virtue of the list is circular.
    - Therefore, one convention is to let the external pointer of the circular linked list, tail, point to the last node and to allow the following node to be the first node.
    - If the tail pointer refers to NULL, means the circular linked list is empty.

- We can declare the structure for the circular linked list in the same way as declared it for the linear linked list.

```
struct node
{
        int info;
        struct node *next;
};
typedef struct node NodeType;
NodeType *start=NULL:
NodeType *last=NULL:
```

- **Algorithm to insert a node at the beginning of a circular linked list:**

```
1.  Create a new node as
            newnode=(NodeType*)malloc(sizeof(NodeType));
2.  if start==NULL then
            set newnode->info=item
            set newnode->next=newnode
            set start=newnode
            set last = newnode
    else
            set newnode->info=item
            set newnode->next=start
            set start=newnode
            set last->next=newnode
3.  Exit
```

- **Algorithm to insert a node at the end of a circular linked list:**

```
1.  Create a new node as
            newnode=(NodeType*)malloc(sizeof(NodeType));
2.  if start==NULL then
            set newnode->info=item
            set newnode->next=newnode
            set start=newnode
            set last = newnode
    else
            set newnode->info=item
            set last->next=newnode
            set last=newnode
            set last->next=start
3.  Exit
```

- **Algorithm to delete a node from the beginning of a circular linked list:**

```
1.  if start==NULL then
            "empty list" and exit
    else if start==last
            set PTR = start;
            print the deleted element=PTR->info
            free(PTR)
            set start = last = NULL;
    else
            set PTR=start
            set start=start->next
            set last->next=start;
            print the deleted element=PTR->info
            free(PTR)
2.  Exit
```

Algorithm to delete a node from the end of a circular linked list:

```
1. if start==NULL then
        "empty list" and exit
2. if start==last
        set PTR=start
        print deleted element=PTR->info
        free(PTR)
        start=last=NULL
   else
        set temp=start
        while( temp->next!=last)
            set temp=temp->next
        end while
        set PTR=last
        set last=temp
        set last->next=start
            print the deleted element=PTR->info
        free(PTR)
3. Exit
```

Representation of Circular Singly Linked List in C

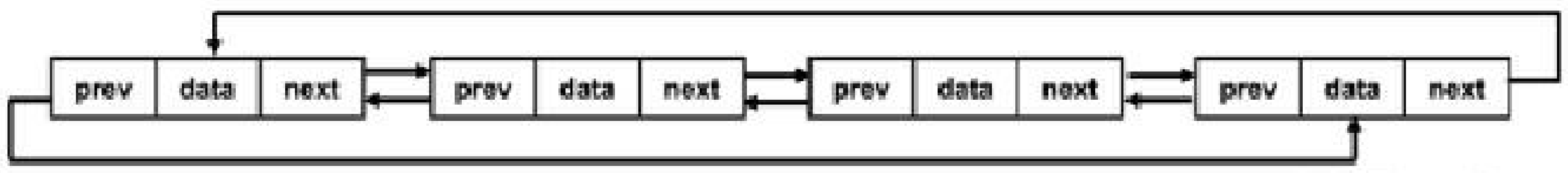
```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct node
{
        int info;
        struct node *next;
};
typedef struct node Lnode;
Lnode *start = NULL;
Lnode *last = NULL;
void create_linkedlist();
void display();
void insert_beg();
void insert_end();
void delete_beg();
void delete_end();
int main()
{
  int option,flag=1;
  printf("\n\n *****MAIN MENU *****");
        printf("\n 1: Create a list");
        printf("\n 2: Display the list");
        printf("\n 3: Add a node at the beginning");
        printf("\n 4: Add a node at the end");
        printf("\n 5: Delete a node from the beginning");
        printf("\n 6: Delete a node from the end");
        printf("\n 7: EXIT");
        do
        {
                printf("\n\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                        case 1:
                                        create_linkedlist();
                                        printf("\n LINKED LIST CREATED");
                                        break;
                        case 2:
                                        display();
                                        break;
                        case 3:
                                        insert_beg();
                                        break;
                        case 4:
                                        insert_end();
                                        break;
                        case 5:
                                        delete_beg();
                                        break;
                        case 6:
                                        delete_end();
                                        break;
                        case 7:
                                        flag = 0;
                                        break;
                        default:
                                printf("Invalide Choice");
                }
        }while(flag);
        getch();
        return 0;
}
void create_linkedlist()
{
        Lnode *new_node, *ptr;
        int data,ch;
        do
        {
                printf("\n Enter the data : ");
                scanf("%d",&data);
                new_node = (Lnode*)malloc(sizeof(Lnode));
                new_node->info=data;
                if(start==NULL)
                {
                        new_node->next = new_node;
                        start=new_node;
                        last=new_node;
                }
                else
                {
                        last->next = new_node;
                        last = new_node;
                        last->next=start;
                }
        printf("\n Do you want to add more data then press 1:");
        scanf("%d",&ch);
        }while(ch == 1);

}
void display()
{
        Lnode *ptr;
        if(start == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        ptr = start;
        printf("\nLinked List Data are:\n");
        while(ptr!=last)
        {
                printf("\t %d", ptr->info);
                ptr=ptr->next;
        }
        printf("\t %d", ptr->info);
}
void insert_beg()
{
        Lnode *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node=(Lnode *)malloc(sizeof(Lnode));
        new_node->info=data;
        if(start==NULL)
        {
                new_node->next=new_node;
                start=new_node;
                last=new_node;
        }
        else
        {
                new_node->next=start;
                start = new_node;
                last->next=new_node;
        }

}
void insert_end()
{
        Lnode *ptr, *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node = (Lnode *)malloc(sizeof(Lnode));
        new_node->info = data;
        if(start==NULL)
        {
                new_node->next=new_node;
                start=new_node;
```

```
			last=new_node;
		}
		else
		{
			last->next = new_node;
			last = new_node;
			last->next=start;
		}
	}
	void delete_beg()
	{
		Lnode *ptr;
		if(start == NULL)
		{
			printf("\nList is empty\n");
		}
		else if(start==last)
		{
			ptr = start;
			printf("\nDleted data from end is:%d\n",ptr->info);
			free(ptr);
			start = last = NULL;
		}
		else
		{
			ptr = start;
			start = start -> next;
			last->next = start;
			printf("\nDleted data from begining is:%d\n",ptr->info);
			free(ptr);
		}
	}
	void delete_end()
	{
		Lnode *ptr,*temp;
		if(start == NULL)
		{
			printf("\nList is empty\n");
			return;
		}
		else if(start==last)
		{
			ptr = start;
			printf("\nDleted data from end is:%d\n",ptr->info);
			free(ptr);
			start = last = NULL;
		}
		else
		{
			temp = start;
			while(temp -> next != last)
			{
				temp = temp -> next;
			}
```

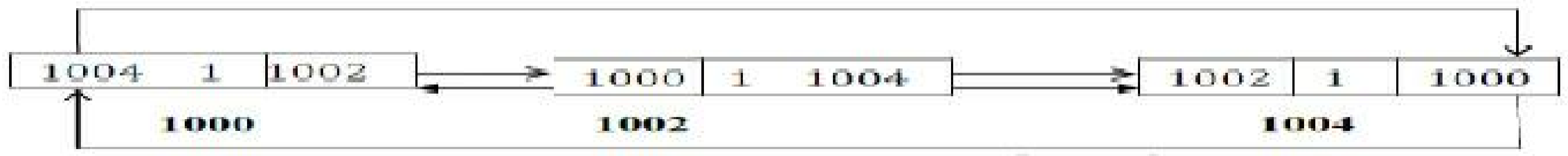
```
                ptr=last;
                last=temp;
                last->next=start;
                printf("\nDleted data from end is:%d\n",ptr->info);
                free(ptr);
            }
        }
```

4.5.6 Circular Doubly Linked List

- A Circular Doubly Linked List (CDL) is a doubly linked list with first node linked to last node and vice-versa.
- **The 'prev' link of first node contains the address of last node and 'next'** link of last node contains the address of first node.
- Traversal through Circular Singly Linked List is possible only in one direction.
- The main advantage of Circular Doubly Linked List (CDL) is that, a node can be inserted into list without searching the complete list for finding the address of previous node.
- We can also traversed through CDL in both directions, from first node to last node and vice-versa.



Example:

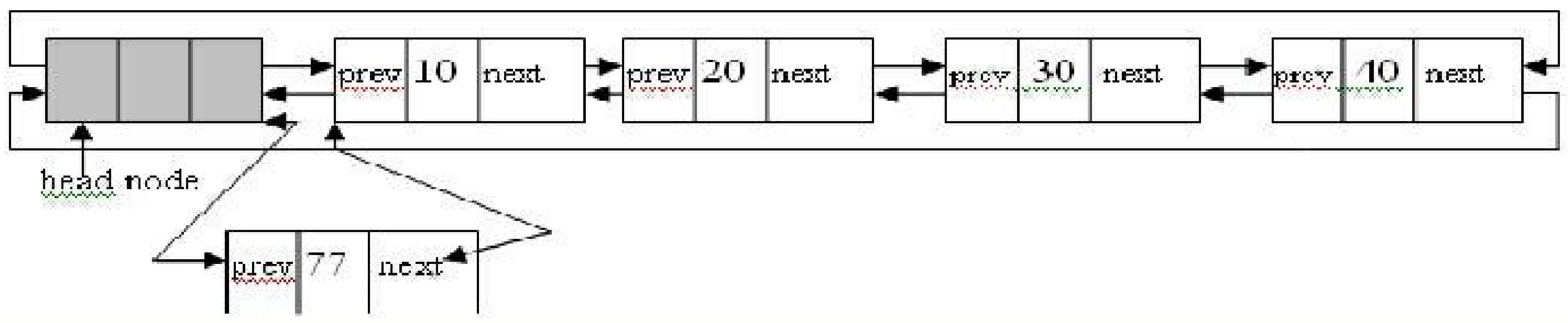


- We can declare the structure for the circular doubly linked list in the same way as declared it for the double linked list

```
struct node
{
  int info;
  struct node *prev;
  struct node *next;
};
typedef struct node NodeType;
NodeType *head=NULL;
```

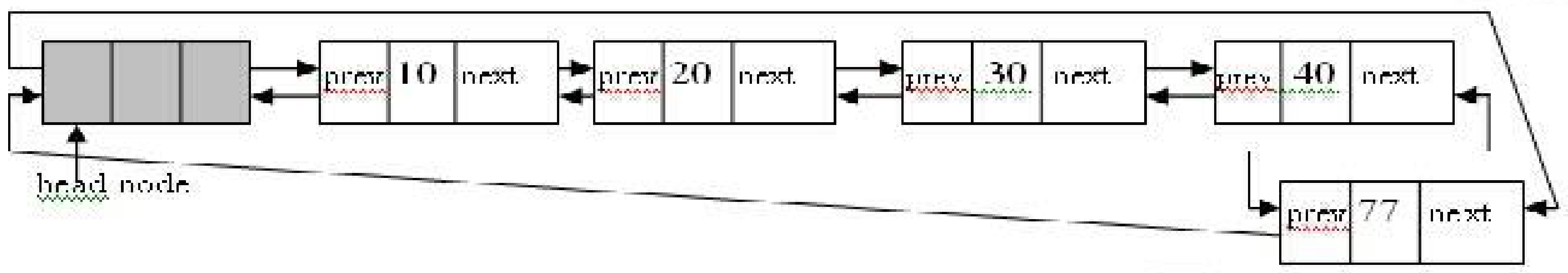
<u>Algorithm to insert a node at the beginning of a circular doubly linked list</u>

```
1. Allocate memory for the new node as,
        newnode=(NodeType*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node
        set newnode->info=item
3. If head == NULL
        head = newnode
        head->next = head->prev = newnode
   else
        set newnode->next=head
        set newnode->prev=head->prev
        set (head->prev)->next = newnode
        set head = newnode
4. Exit
```



Algorithm to insert a node at the end of a circular doubly linked list :

```
1. Allocate memory for the new node as,
        newnode=(NodeType*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node
        set newnode->info=item
5. If head == NULL
        head = newnode
        head->next = head->prev = newnode
   else
        set (head->prev )->next=newnode
        set newnode->prev=(head->prev)
        set newnode->next=head
        set head->prev=newnode
3. Exit
```



Algorithm to delete a node from the beginning of a circular doubly linked list

```
1. if head==NULL then
        print "empty list" and exit
   else if(head->next==head)
        PTR = head;
        head=NULL;
        print "Deleted data from end= PTR ->info";
        free(PTR);
   else
        set PTR=head;
        set (PTR->next)->prev = (PTR->prev)
        set (PTR->prev)->next = PTR->next
        set head = PTR->next
        free(PTR)
2. Exit
```

Algorithm to delete a node from the end of a circular doubly linked list:

```
1. if head ==NULL then
        print "empty list" and exit
   else if(head->next==head)
        PTR = head;
        head=NULL;
        print "Deleted data from end= PTR ->info";
        free(PTR);
   else
        set PTR=head->prev
        set (PTR->prev)->next = head
        set head->prev = PTR->prev
        free(PTR)
2. Exit
```

Representation of circular doubly linked list in C

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
struct Node
{
        int info;
        struct Node *next;
        struct Node *prev;
};
typedef struct Node node;
node *head = NULL;
void create_linkedlist();
void display();
void insert_beg();
void insert_end();
void delete_beg();
void delete_end();
int main()
{
  int option,flag=1;
  printf("\n\n *****MAIN MENU *****");
        printf("\n 1: Create a list");
        printf("\n 2: Display the list");
        printf("\n 3: Add a node at the beginning");
        printf("\n 4: Add a node at the end");
        printf("\n 5: Delete a node from the beginning");
        printf("\n 6: Delete a node from the end");
        printf("\n 7: EXIT");
        do
        {
                printf("\n\n Enter your option : ");
                scanf("%d", &option);
                switch(option)
                {
                        case 1:
                                        create_linkedlist();
                                        printf("\n LINKED LIST CREATED");
                                        break;
                        case 2:
                                        display();
                                        break;
                        case 3:
                                        insert_beg();
                                        break;
                        case 4:
                                        insert_end();
                                        break;
                        case 5:
                                        delete_beg();
                                        break;
                        case 6:
                                        delete_end();
                                        break;
                        case 7:
                                flag = 0;
                                break;
                        default:
                               printf("Invalide Choice");
                }
        }while(flag);
        getch();
        return 0;
}
void create_linkedlist()
{
        node *new_node, *ptr;
        int data,ch;
        do
        {
                printf("\n Enter the data : ");
                scanf("%d",&data);
                new_node = (node*)malloc(sizeof(node));
                new_node->info=data;
                if(head==NULL)
                {
                        head=new_node;
                        head->next=head->prev = new_node;
                }
                else
                {
                        (head->prev )->next=new_node;
                        new_node->prev=(head->prev);
                        new_node->next=head;
                        head->prev=new_node;
                }
        printf("\n Do you want to add more data then press 1:");
        scanf("%d",&ch);
        }while(ch == 1);

}
void display()
{
        node *ptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
                return;
        }
        ptr = head;
        printf("\nLinked List Data are:\n");
        while(ptr!=head->prev)
        {
                printf("\t %d", ptr->info);
                ptr=ptr->next;
        }
        printf("\t %d", ptr->info);
}
```

```
void insert_beg()
{
        node *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node=(node *)malloc(sizeof(node));
        new_node->info=data;
                if(head==NULL)
                {
                        head=new_node;
                        head->next=head->prev = new_node;
                }

                else
                {
                        (head->prev )->next=new_node;
                        new_node->prev=(head->prev);
                        new_node->next=head;
                        head->prev=new_node;
                }
}
void insert_end()
{
        node *ptr, *new_node;
        int data;
        printf("\n Enter the data : ");
        scanf("%d", &data);
        new_node = (node *)malloc(sizeof(node));
        new_node->info = data;
                if(head==NULL)
                {
                        head=new_node;
                        head->next=head->prev = new_node;
                }
                else
                {
                        (head->prev )->next=new_node;
                        new_node->prev=(head->prev);
                        new_node->next=head;
                        head->prev=new_node;
                }
}
void delete_beg()
{
        node *ptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
        }
        else if(head->next==head)
        {
                ptr = head;
                head=NULL;
                printf("\nDleted data from end is:%d\n",ptr->info);
                free(ptr);
        }
        else
        {
                ptr = head;
                (ptr->next)->prev = (ptr->prev);
                (ptr->prev)->next = ptr->next;
                head = ptr->next;
                printf("\nDleted data from begining is:%d\n",ptr->info);
                free(ptr);
        }
}

void delete_end()
{
        node *ptr,*preptr;
        if(head == NULL)
        {
                printf("\nList is empty\n");
        }
        else if(head->next==head)
        {
                ptr = head;
                head=NULL;
                printf("\nDeleted data from end is:%d\n",ptr->info);
                free(ptr);
        }
        else
        {
                ptr=head->prev;
                (ptr->prev)->next = head;
                head->prev = ptr->prev;
                printf("\nDleted data from end is:%d\n",ptr->info);
                free(ptr);
        }
}
```

4.5.6.1 Advantages of Circular Linked Lists:

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- Useful for implementation of queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last.
- Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.
- Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

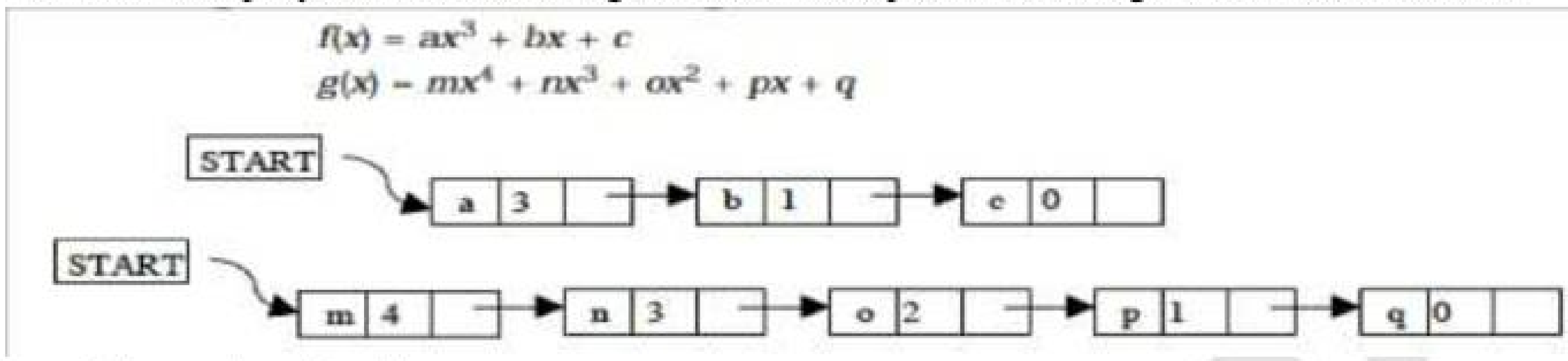
4.5.7 Polynomials using singly linked list (Application of Linked List)

Different operations, such as addition, subtraction, division and multiplication of polynomials can be performed using linked list. Following example shows that the polynomial addition using linked list.

In the linked representation of polynomials, each term is considered as a node. And such a node contains three fields: coefficient field, exponent field and Link field.

```
struct polynode
{
        int coeff;
        int expo;
        struct polynode *next;
};
```

- Consider two polynomials f(x) and g(x); it can be represented using linked list as follows.



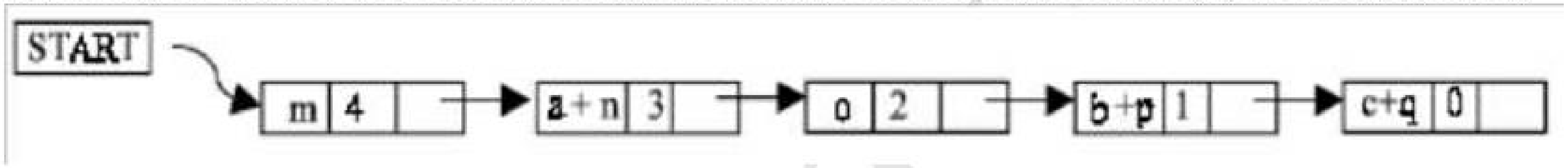
$f(x) = ax^3 + bx + c$

$g(x) = mx^4 + nx^3 + ox^2 + px + q$

two polynomials can be added by

h(x) = f(x) + g(x) = mx4 + (a + n) x3 + ox2 + (b + p)x + (c + q)

i.e.; adding the constants of the corresponding polynomials of the same exponentials. h(x) can be represented as



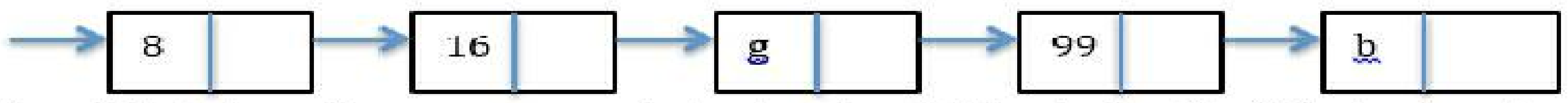
Steps involved in adding two polynomials

1. Read the number of terms in the first polynomial, f(x).
2. Read the coefficient and exponents of the first polynomial.
3. Read the number of terms in the second polynomial g(x).
4. Read the coefficients and exponents of the second polynomial.
5. Set the temporary pointers p and q to traverse the two polynomial respectively.
6. Compare the exponents of two polynomials starting from the first nodes.
   a. If both exponents are equal then add the coefficients and store it in the resultant linked list.
   b. If the exponent of the current term in the first polynomial p is less than the exponent of the current term of the second polynomial is added to the resultant linked list. And move the pointer q to point to the next node in the second polynomial Q.
   c. If the exponent of the current term in the first polynomial p is greater than the exponent of the current term in the second polynomial Q then the current term of the first polynomial is added to the resultant linked list. And move the pointer p to the next node.
   d. Append the remaining nodes of either of the polynomials to the resultant linked list.

- **<u>Generalized Linked List</u>**

  Linked list data structure can be viewed as abstract data type which consists of sequence of objects called elements. Associated with each list element is value. We can make distinction between an element, which is an object as part of a list and the elements value, which is the object considered individually. For example, the number 8 may appear on a list twice. Each appearance is a distinct element of the list, but the value of the two elements the number 8 is same. An element may be viewed as corresponding to a node in the linked list representation, where a value corresponds to the node's contents.

  There is a convenient notation for specifying abstract general lists. A list may be denoted by a parenthesized enumeration of its elements separated by commas. For example, the abstract list represented by following figure may be represented as: list = (8,16,'g', 99,'b')



The null list is denoted by an empty parenthesis pair such as ( ). Thus the list of the following figure is represented as:
list = (8,(5,7,(18,3,6),( ),5),2,(6,8))

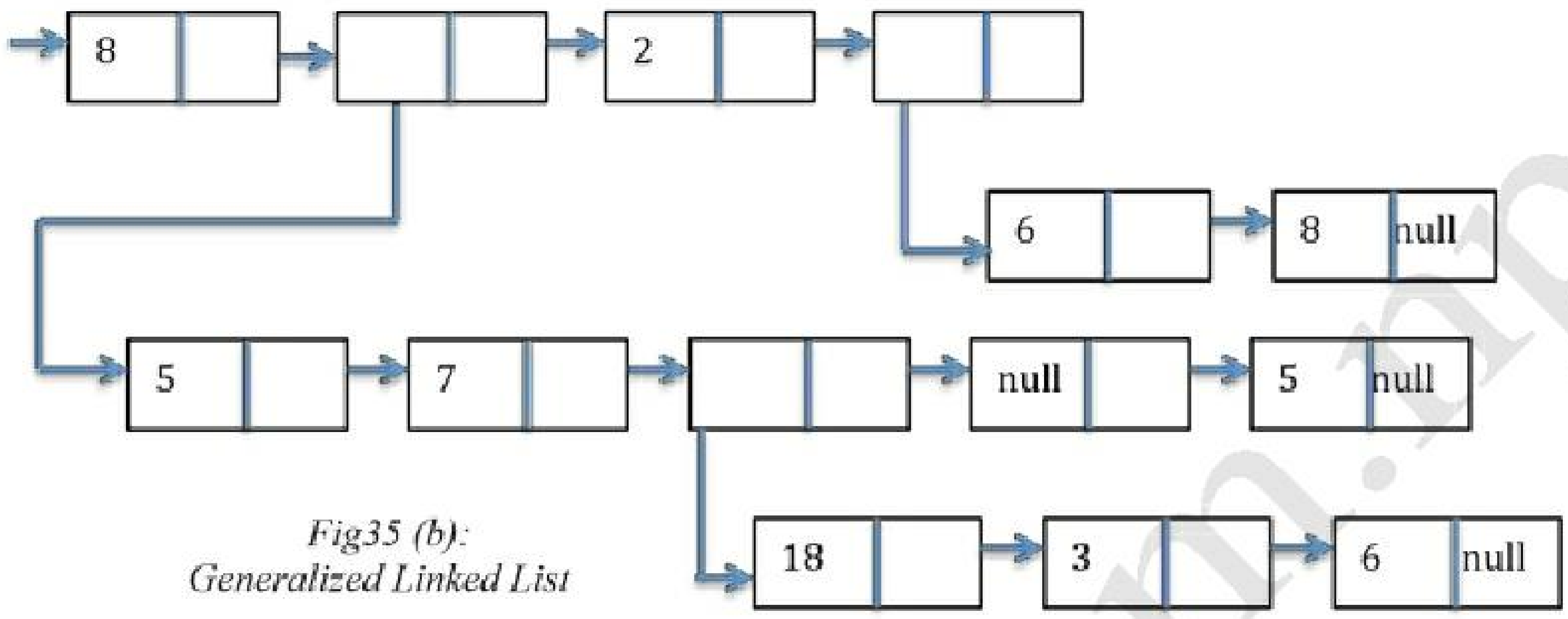


Fig35 (b): Generalized Linked List