UNIT-3: THE QUEUE AND LINKED LIST

QUEUE

A queue is logically a first in first out (**FIFO** or first come first serve) linear data structure. It is a homogeneous collection of elements in which new elements are added at one end called **rear**, and the existing elements are deleted from other end called **front**.

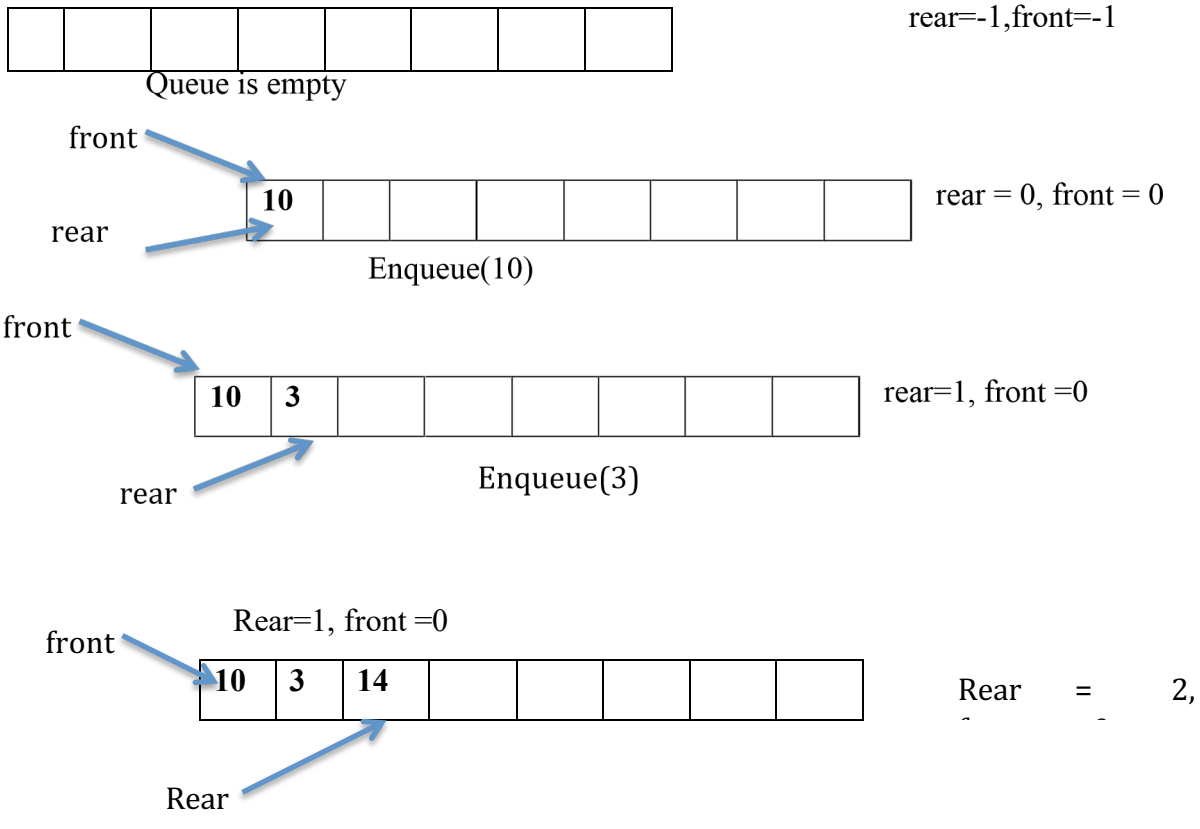
The basic operations (Primitive Operations) also called as **Queue ADT** that can be performed on queue are

1. **Enqueue() or Insert()** (or add) an element to the queue (push) from the **rear** end.
2. **Dequeue() or Delete()** (or remove) an element from a queue (pop) from **front** end.
3. **Front**: return the object that is at the front of the queue without removing it.
4. **Empty**: return true if the queue is empty otherwise return false.
5. **Size**: returns the number of items in the queue.

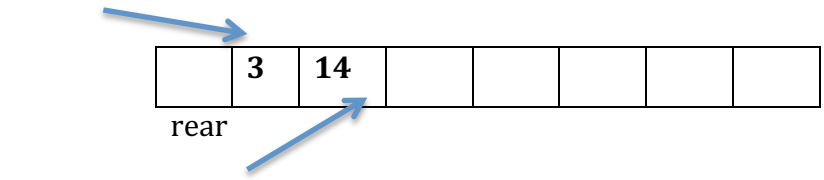
TYPES OF QUEUE

i) Linear Queue or Simple queue ii) Circular queue
iii) Double ended queue (de-queue)
iv) Priority queue: Priority queue is generally implemented using linked list

Enqueue operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. **Pop** operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Initially rear is set to -1 and front is set to -1. The queue is empty whenever **rear < front** or both the *rear and front is equal to -1*. Total number of elements in the queue at any time is equal to **rear-front+1**, when implemented using arrays. Below are the few operations in queue.



Enqueue(14)



Rear = 2,

x = Dequeue() (i.e. x = 10)

Note. During the insertion of first element in the queue we always increment the front by one.

Queue can be implemented in two ways:
1. Using arrays (static)
2. Using pointers (dynamic)

If we try to dequeue (or delete or remove) an element from queue when it is empty, **underflow** occurs. It is not possible to delete (or take out) any element when there is no element in the queue. Suppose maximum size of the queue (when it is implemented using arrays) is 50. If we try to Enqueue (or insert or add) an element to queue, overflow occurs. When queue is full it is naturally not possible to insert any more elements.

3.1. ALGORITHM FOR QUEUE OPERATIONS (Linear Queue)

Let Q be the arrays of some specified size say SIZE.

3.1.1. Inserting An Element Into The Queue

1. Initialize front=-1 rear = –1; [Create an Empty Queue]
2. Input the value to be inserted and assign to variable "data"
3. If (rear >= SIZE)
   i. Display "Queue overflow"
   ii. Exit
4. Else
   i. Rear = rear +1
5. Q [rear] = data
6. Exit

3.1.2. Deleting An Element From Queue

1. If (rear< front) or if(front ==-1 && rear == -1);[checking for queue is empty]
   i. front = -1, rear = –1;(optional)
   ii. Display "The queue is empty"
   iii. Exit
2. Else
   i. Data = Q [front]
3. Front = front +1
4. Exit

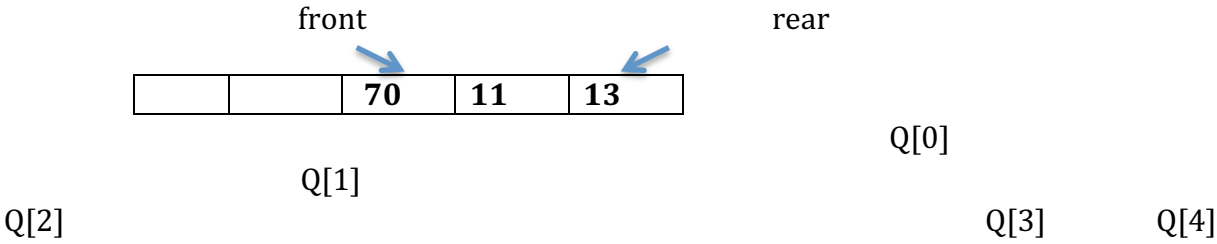
3.1.3. Program to Implement Queue Using Array

```
//PROGRAM TO IMPLEMENT QUEUE USING ARRAYS
#include<stdio.h>
#include <stdlib.h>
#define MAX 50
int queue_arr[MAX];
int rear = -1;
int front =-1;
//This function will insert an element to the queue
void Enqueue ()
{
int added_item;
if (rear==MAX-1)
{
printf("\nQueue Overflow\n");
return;     }
else     {
if(isEmpty() == 1) /*If queue is initially empty */
{
printf("Queue is Empty, Your are going to create a queue\n");
front = 0;
}
printf("\nInput the element for adding in queue:");
scanf("%d", &added_item);
 rear=rear+1;
//Inserting the element
 queue_arr[rear] = added_item ;
}
}/*End of insert()*/
//This function will delete (or pop) an element from the queue void
Dequeue()
{
if (isEmpty() == 1)
{
printf ("\nQueue Underflow or Queue is Empty\n");
 return;
}
else
{
//deleteing the element
printf ("\nElement deleted from queue is : %d\n",
queue_arr[front]);
front=front+1;
}
}/*End of del()*/
//Displaying all the elements of the queue void
Traverse()
{
int i;
//Checking whether the queue is empty or not
if (isEmpty() ==1)
{
printf ("\nQueue is empty\n");
return;
```

```
}     else
{
printf("\nFront is %d and Queue is :\n",front);
for(i=front;i<= rear;i++)
printf("%d ",queue_arr[i]);
printf("\n");
}
}/*End of display() */

/*Checks whether the Queue is Empty or not */ int
isEmpty()
{
if (front > rear) // if(front == -1 && rear == -1)
{
return 1; // True
}     else
return 0; //False

}
/*End if isEmpty() */

int main() {
int choice;
while(1)
{
printf("\n1.Enqueue\n");
printf("2.Dequeue\n");
printf("3.Traverse\n");
printf("4.Size\n");
printf("5.Quit");
printf("\nEnter your choice:");
scanf("%d", & choice);
switch(choice)
{
case 1 :
Enqueue();
break;
case 2:
Dequeue();
break;
case 3:
Traverse();
break;
case 4:
printf("The size of the queue is %d\n",rear-front+1);
break;
case 5:
exit(1);
default:
printf ("\n Wrong choice\n");

}/*End of switch*/
}/*End of while*/
return 0; }/*End of
main()*/
```

3.2. Circular Queue

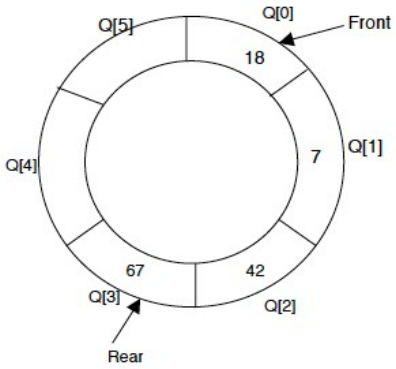
Suppose a queue Q has maximum size 5, say 5 elements pushed and 2 elements popped.



Now if we attempt to add more elements, even though 2 queue cells are free, the elements cannot be pushed. Because in a queue, elements are always inserted at the rear end and hence rear points to last location of the queue array Q[4]. That is queue is full (overflow condition) though it is empty. This limitation can be overcome if we use circular queue.

In circular queues the elements Q[0],Q[1],Q[2] .... Q[n – 1] is represented in a circular fashion with Q[1] following Q[n]. A circular queue is one in which the insertion of a new element is done at the very first location of the queue if the last location at the queue is full.

Suppose Q is a queue array of 6 elements. Enqueue() and Dequeue() operation can be performed on circular. The following figures will illustrate the same.



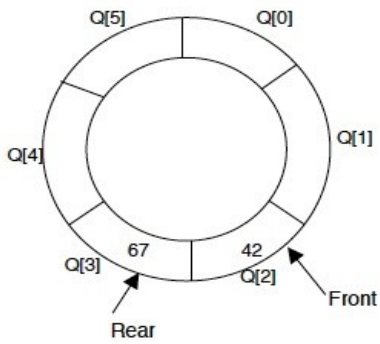


Fig1: A Circular Queue After inserting 18,7,42,67. **Fig2**: Circular Queue after popping 18,7.

After inserting an element at last location Q[5], the next element will be inserted at the very first location (i.e., Q[0]) that is circular queue is one in which the first element comes just after the last element.

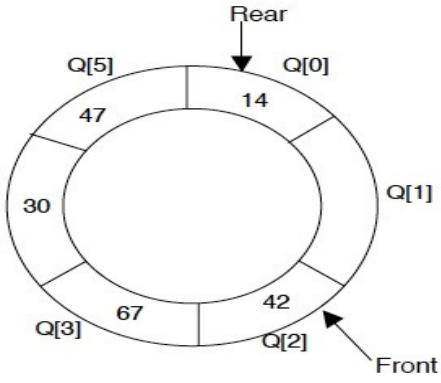


Fig3: Circular Queue after pushing 30, 47, 14

At any time the relation will calculate the position of the element to be inserted ***Rear = (Rear + 1) % SIZE***

After deleting an element from circular queue the position of the front end is calculated by the relation

Front= (Front + 1) % SIZE

After locating the position of the new element to be inserted, **rear**, compare it with **front**. If (**rear = front**), the queue has only one element.

3.2.1. ALGORITHMS

Let Q be the arrays of some specified size say SIZE. FRONT and REAR are two pointers where the elements are deleted and inserted at two ends of the circular queue. DATA is the element to be inserted.

Initialize FRONT = -1 and REAR = -1

3.2.1.1. Inserting an element to circular Queue

1. If ((FRONT is equal to 0 && rear = MAX-1) OR front = rear+1) //check for queue full i. Display "Queue is full"
   ii. Exit
2. If (FRONT is equal to – 1 && rear == -1) //Empty Queue (inserting first time) i. FRONT = 0
   ii. REAR = 0
3. Else
   i) If (REAR == MAX-1) //Rear is at last position
      REAR = 0;
   ii) Else
      REAR = (REAR + 1) % SIZE
4. Input the value to be inserted and assign to variable "DATA"
5. Q [REAR] = DATA
6. Repeat steps 2 to 7 if we want to insert more elements
7. Exit

3.2.1.2. Deleting an element from a circular queue

1. If (FRONT is equal to – 1 and rear == -1)
   i. Display "Queue is empty"
   ii. Exit
2. Else
   i. DATA = Q [FRONT]
3. If (REAR is equal to FRONT) //Queue has only one element
   i. FRONT = –1
   ii. REAR = –1
4. Else
   i. FRONT = (FRONT +1) % SIZE
5. Repeat the steps 1 to 4, if we want to delete more elements
6. Exit

3.2.1.3. Program to Implement Circular Queue Using Array

```
//PROGRAM TO IMPLEMENT CIRCULAR QUEUE USING ARRAY
#include<stdio.h>
#define MAX 50
//Global Variables

int cqueue_arr[MAX];
int front,rear;

    //initialize the variables
void circular_queue()
    {
        front = -1;
rear = -1;
    }


//Function to insert an element to the circular queue
void Enqueue()
{
    int added_item;
    //Checking for overflow condition
    if ((front == 0 && rear == MAX-1)
    {
        printf("\nQueue Overflow \n");

return;
    }
    if (front == -1 && rear == -1) /*If queue is empty */
    {
front = 0;
rear = 0;
    }
else
        if (rear == MAX-1)/*rear is at last position of queue */
rear = 0;
else
  rear = (rear + 1)%MAX;
   printf("\nInput the element for insertion in queue:");
scanf("%d",&added_item);
cqueue_arr[rear] = added_item;
}/*End of insert()*/

//This function will delete an element from the queue
void Dequeue()
{
    //Checking for queue underflow
if (front == -1 && rear == -1)
    {
        printf("\nQueue Underflow\n");
return;
    }
    printf("\nElement deleted from queue is: %d,\n",cqueue_arr[front]);
if (front == rear) /* queue has only one element */
    {
```

```
front = -1;
rear = -1;
    }
else
 if(front == MAX-1)
front = 0;
else
 front = (front + 1)%MAX;
}/*End of del()*/
//Function to display the elements in the queue void
Traverse()
{
    int front_pos = front,rear_pos = rear;
    //Checking whether the circular queue is empty or not
if (front == -1 && rear == -1)
    {
        printf("\nQueue is empty\n");
return;
    }
    //Displaying the queue elements
printf("\nQueue elements:\n");
if(front_pos <= rear_pos )
while(front_pos <= rear_pos)
        {
            printf("%d,",cqueue_arr[front_pos]);
front_pos++;
        }
else
{
        while(front_pos <= MAX-1)
        {
            printf("%d,", cqueue_arr[front_pos]);
front_pos++;
        }
front_pos = 0;
        while(front_pos <= rear_pos)
        {
            printf("%d,",cqueue_arr[front_pos]);
front_pos++;         }     }/*End of else*/
printf("\n"); }/*End of display() */ int main()
{
    int choice;

    //Creating the objects for the class
circular_queue();
while(1)
    {
        //Menu options
printf("\n1.Enqueue\n");
printf("2.Dequeue\n");
printf("3.Traverse\n");
printf("4.Quit\n");
printf("\nEnter your choice: ");
scanf("%d",&choice);
switch(choice)
        {
```

```
case 1:
Enqueue();
break;
case 2 :
Dequeue();
break;
case 3:
Traverse();
break;
case 4:
exit(1);
default:
printf("\nWrong choice\n");
        }/*End of switch*/
}/*End of while*/
return 0; }/*End of
main()*/
```

3.3. DEQUES

A deque is a homogeneous list in which elements can be added or inserted (called Enqueue operation) and deleted or removed from both the ends (which is called Dequeue operation). ie; we can add a new element at the rear or front end and also we can remove an element from both front and rear end. Hence it is called Double Ended Queue.

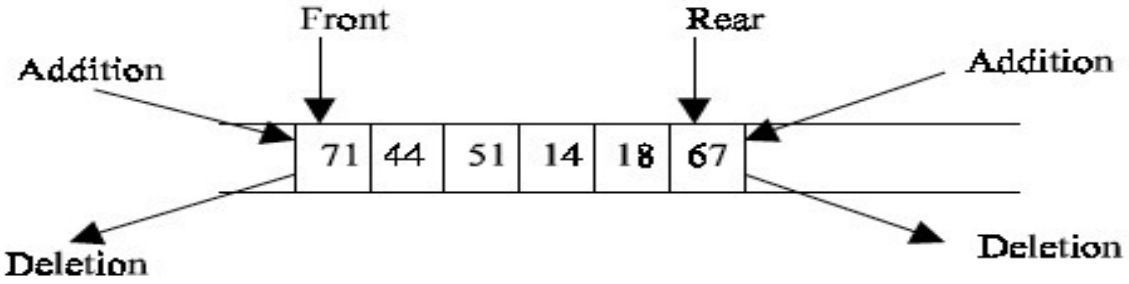


Fig4: A Deque

There are two types of deque depending upon the restriction to perform insertion or deletion operations at the two ends. They are

1. **Input restricted deque:** An input restricted deque is a deque, which allows insertion at only 1 end, rear end, but allows deletion at both ends, rear and front end of the lists.
2. **Output restricted deque:** An output-restricted deque is a deque, which allows deletion at only one end, front end, but allows insertion at both ends, rear and front ends, of the lists.

The possible operation (Deque ADT) performed on deque is

1. Add an element at the rear end (insert_rear)
2. Add an element at the front end (insert_front)
3. Delete an element from the front end (delete_front)
4. Delete an element from the rear end (delete_rear)

Only 1st, 3rd and 4th operations are performed by input-restricted deque and 1st, 2nd and 3rd operations are performed by output-restricted deque.

3.3.1. Algorithms For Inserting An Element

Let Q be the array of MAX elements. front (or left) and rear (or right) are two array index (pointers), where the addition and deletion of elements occurred. Let DATA be the element to be inserted. Before inserting any element to the queue left and right pointer will point to the -1.

Insert an element at the *right side (rear end)* of the de-queue

1. Input the DATA to be inserted
2. If ((front == 0 && rear == MAX–1) | | (front == rear + 1))
   i. Display "Queue Overflow"
   ii. Exit
3. If (front == –1 && rear == -1)
   i. front = 0
   ii. rear = 0
   iii. Q[rear] = DATA
4. Else
   i. if (rear != MAX –1) //Deque already contain more than one element
      a) rear = rear+1
      b) Q[rear] = DATA ii. Else //rear has reach at the end of array
      [Shift all the elements from position front until rear back by 1 position] for(i = front; i<=rear;i++)
      Q[i-1] = Q[i]
      Q [rear] = DATA
5. Exit

Insert An Element At The Left Side (Front) Of The De-Queue

1. Input the DATA to be inserted
2. If ((front == 0 && rear == MAX–1) | | (front == rear+1))
   i. Display "Queue Overflow"
   ii. Exit
3. If (front == – 1 && rear == -1)
   i. front = 0
   ii. rear = 0
   iii. Q [front] = DATA
4. Else
   i. if (front != 0) // Deque already contain more than one element
      a) front = front– 1
      b) Q [front] = DATA ii. Else
      a. Shift all the elements from position front until rear ahead by 1 position for(i=rear; i>=front;i--)
         Q[i+1] = Q [i]
      b. Q[front] = DATA
5. Exit

3.3.2. ALGORITHMS FOR DELETING AN ELEMENT

Let Q be the array of MAX elements. front (or left) and rear (or right) are two array index (pointers), where the addition and deletion of elements occurred. DATA will contain the element just deleted.

Delete An Element From The *Right Side (Rear side)* Of The De-Queue

1. If (front == – 1 && rear == -1)
   i. Display "Queue Underflow"
   ii. Exit
2. If (front == rear) //Deque has only one element (last element)
   i) DATA = Q [rear]
   a) front = – 1

b) rear = – 1

3. Else //deque has more than one element
   i. DATA = Q [rear]
      a) rear = rear -1 4.

Exit

Delete An Element From The *Left Side (Front Side)* Of The De-Queue

1. If (front == – 1 && rear == -1)
   i. Display "Queue Underflow"
   ii. Exit
2. If(front == rear)
   i. DATA = Q [front]

      a) front = – 1
      b) rear = – 1
3. Else // Dequeue has more than one element
   i) DATA = Q [front]
      a) Front = front +1
4. Exit

3.4. PRIORITY QUEUES

Priority Queue is a queue where each element is assigned a priority. In priority queue, the elements are deleted and processed by following rules.

1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were inserted to the queue.

For example, Consider a manager who is in a process of checking and approving files in a first come first serve basis. In between, if any urgent file (with a high priority) comes, he will process the urgent file next and continue with the other low urgent files.

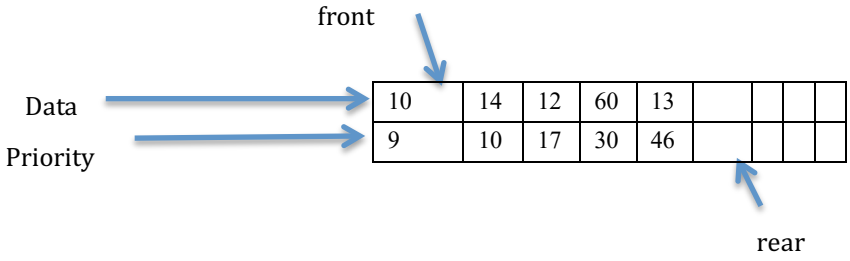


Fig5: Priority Queue representation using Array

Above Fig5 gives a pictorial representation of priority queue using arrays after adding 5 elements (10,14,12,60,13) with its corresponding priorities (9,10,17,30,46). Here the priorities of the data(s) are in ascending order. Always we may not be pushing the data in an ascending order. From the mixed priority list it is difficult to find the highest priority element if the priority queue is implemented using arrays. Moreover, the implementation of priority queue using array will yield n comparisons (in liner search), so the time complexity is much higher than the other queue for inserting an element. So it is always better to implement the priority queue using linked list - where a node can be inserted at anywhere in the list.

3.5. APPLICATION OF QUEUES

1. Round robin techniques for processor scheduling is implemented using queue.
2. Printer server routines (in drivers) are designed using queues.
3. All types of customer service software (like Railway/Air ticket reservation) are designed using queue to give proper service to the customers.
4. Disk Driver: maintains a queue od disk input/output requests
5. Scheduler (e.g, in operating system): maintains a queue of processes awaiting a slice of machine time.
6. Call center phone systems will use a queue to hold people in line until a service representative is free.
7. Buffers on MP3 players and portable CD players, iPod playlist. Playlist for jukebox - add songs to the end, play from the front of the list.
8. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
9. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

STATIC AND DYNAMIC LIST

4.1. STATIC IMPLEMENTATION OF LIST

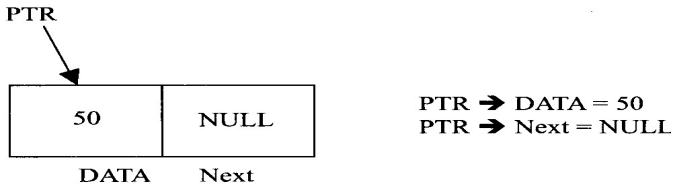
Static implementation can be implemented using arrays. It is very simple method but it has Static implementation. Once a size is declared, it cannot be change during the program execution. It is also *not efficient for memory utilization*. When array is declared, memory allocated is equal to the size 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.

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.*

LINKED LIST

A linked list is a linear collection of specially designed data structure, called **nodes**, linked to one another by means of **pointer**. Each node is divided into 2 parts: the first part contains information of the element and the second part contains address of the next node in the link list.



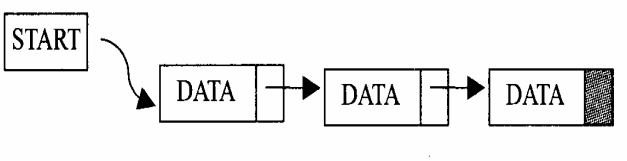


Fig1: Node Fig2: Linked List

Above Fig2 shows that the schematic diagram of a linked list with 3 nodes. Each node is pictured with two parts. The left part of each node contains the data items and the right part represents the address of the next node. The next pointer of the last node contains a special value, called the **NULL** pointer, which does not point to any address of the node. That is NULL pointer indicates the end of the linked list. START pointer will hold the address of the 1st node in the list START =NULL if there is no list (i.e.; NULL list or empty list).

Representation of Linked List

Suppose we want to store a list of integer numbers using linked list. Then it can be schematically represented as

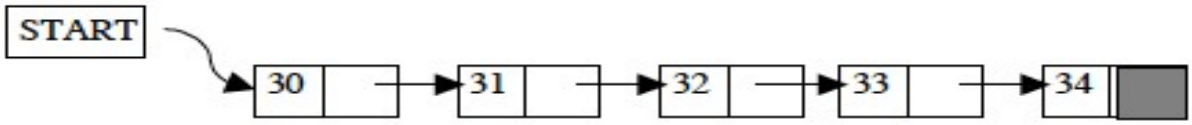


Fig3: Linked list representation

We can declare linear linked list as follows

```
struct Node{
   int data; //instead of data we also use info
struct Node *Next; //instead of Next we also use link
};
typedef struct Node *NODE;
```

Advantages and Disadvantages of Linked List

Linked list have many advantages and some of them are:

1. Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program.
2. Efficient memory utilization: In linked list (or dynamic) representation, memory is not preallocated. Memory is allocated whenever it is required. And it is deallocated (or removed) when it is not needed.
3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Many complex applications can be easily carried out with linked list.

Linked list has following disadvantages

1. More memory: to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.
2. Access to an arbitrary data item is little bit cumbersome and also time consuming.

Operations on Linked List (Primitive Operations)

The primitive operations performed on Linked list is as follows

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Searching
6. Find Previous
7. Concatenation

Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.

i. At the beginning of the linked list
ii. At the end of the linked list
iii. At any specified position in between in a linked list

Deletion operation is used to delete an item (or node) from the linked list. A node may be deleted from the

i. Beginning of a linked list
ii. End of a linked list
iii. Specified location of the linked list

Traversing is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from left to right, forward traversing, nodes only. But in doubly linked list forward and backward traversing is possible.

Concatenation is the process of appending the 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

Types of Linked List

Following are the types of Linked list depending upon the arrangements of the nodes.

1. Singly Linked List
2. Doubly linked List
3. Circular Linked List
    a. Circular Singly Linked List
    b. Circular Doubly Linked List

4.2.1. SINGLY LINKED LIST

All the nodes in a singly linked list are arranged sequentially by linking with a pointer. A singly linked list can grow or shrink, because it is a dynamic data structure. The following figure explains the different operations on Singly Linked List.

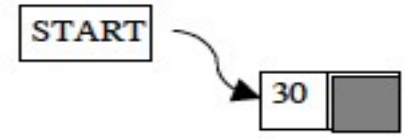


Fig4: Create a node with Data (30)

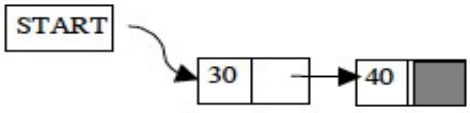


Fig5: Insert a node with Data (40) at the end

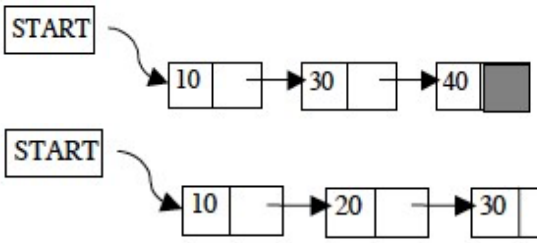


Fig6: Insert a node with Data (10) at the beginning

Fig7: Insert a node with Data (20) at 2nd position

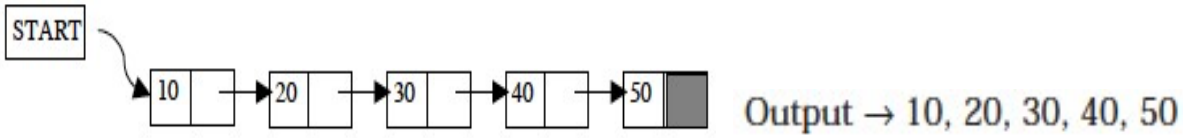


Fig8: Insert a node with Data (50) at the end

Fig9: Traversing the nodes from left to right

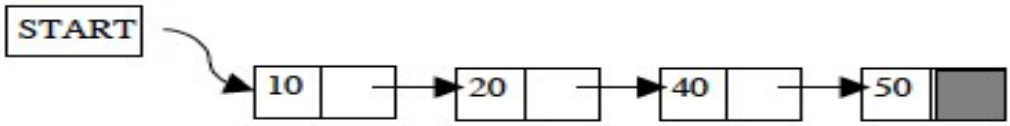


Fig10: Delete 3rd node from the

4.2.1.1. Algorithm For Inserting A Node

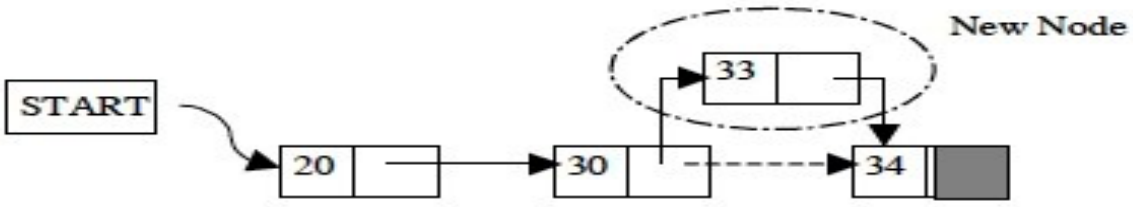


Fig11: Insertion of New Node at specific position

Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the new node is to be inserted. TEMP is a temporary pointer to hold the node address.

1) Insert a Node at the beginning of Linked List

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. If (SATRT equal to NULL)
   (a) NewNode → next = NULL
5. Else
   (a) NewNode → next = START
6. START = NewNode
7. Exit

2) Insert a Node at the end of Linked List

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
8. If (SATRT equal to NULL)
    (a) START = NewNode
9. Else
    (a) TEMP = START
    (b) While (TEMP → Next not equal to NULL)
    (i) TEMP = TEMP → Next
10. TEMP → Next = NewNode
11. Exit

3) Insert a Node at any specified position of Linked List

1. Input DATA and POS to be inserted
2. Initialize TEMP = START; and j = 0
3. Repeat the step 3 while (k is less than POS)
    (a) TEMP = TEMP->Next
    (b) If (TEMP is equal to NULL)
        i. Display "Node in the list less than the position"
        ii. Exit
    (c) k = k + 1
4. Create a New Node
5. NewNode → DATA = DATA
6. NewNode → Next = TEMP → Next
7. TEMP → Next = NewNode
8. Exit

4.2.1.2 Algorithm For Display All Nodes

Suppose START is the address of the first node in the linked list. Following algorithm will visit all nodes from the START node to the end.

1. If (START is equal to NULL)
    (a) Display "The list is Empty"
    (b) Exit
2. Initialize TEMP = START
3. Repeat the step 4 and 5 until (TEMP == NULL)
4. Display "TEMP → DATA"
5. TEMP = TEMP → Next
6. Exit

Program (Single Linked List Insertion Operation)

```
#include<stdio.h> #include
<stdlib.h> struct node
{
    int data; //instead of data we can use info
struct node *next;
};
typedef struct node NODE; NODE
*start;
void createmptylist(NODE *start)
{
    start = (NODE*)malloc(sizeof(NODE)); //start = NULL;
```

```
}

void insert_at_begin(int item)
{
    NODE *ptr;
    ptr=(NODE *)malloc(sizeof(NODE));
ptr->data=item;     if(start==(NODE
*)NULL)          ptr->next=(NODE
*)NULL;     else
        ptr->next=start;
start=ptr;
}
void insert_at_end(int item)
{
    NODE *ptr,*loc;
    ptr=(NODE *)malloc(sizeof(NODE));
ptr->data=item;     ptr->next=NULL;
if(start==NULL)          start=ptr;
else     {          loc=start;
        while(loc->next!=(NODE *)NULL)
loc=loc->next;          loc->next=ptr;
    } }
void insert_spe(int item,int pos)
{
    struct node *tmp,*q;
int i;     q=start;
    //Finding the position to add new element to the linked list
for(i=0;i<pos-1;i++)
    {          q=q-
>next;
if(q==NULL)
{
             printf ("\n\n There are less than %d elements",pos);
return;
        }
    }/*End of for*/
    tmp=(struct node*)malloc(sizeof (struct node));
tmp->next=q->next;     tmp->data=item;     q-
>next=tmp;
}
//This function will display all the element(s) in the linked list void
traverse(NODE *start)
{
    NODE *tmp;
     if(start == NULL)
     {
            printf("List is empty\n");
            return;
     }
      tmp = start;
      while(tmp != NULL)
     {
         printf("%d\n",tmp->info);
tmp=tmp->next;
```

```
    }
}
int main() {
    int choice,item,pos;
createemptylist(start);
while(1)     {
        printf("1. Insert element at begin \n");
printf("2. Insert element at end positon\n");
printf("3. Insert at specific position\n");
printf("4. Travers the list in order\n");
printf("5. Exit\n");
printf(" Enter your choice\n");
scanf("%d",&choice);
switch(choice)
        {
            case 1: printf("Enter the item\n");
scanf("%d",&item);
insert_at_begin(item);
break;
            case 2: printf("Enter the item\n");
scanf("%d",&item);
insert_at_end(item);
break;
            case 3: printf("Enter the item and pos\n");
scanf("%d%d",&item,&pos);
insert_spe(item,pos);
break;
            case 4: printf("\ntravers the list\n");
traverse(start);
break;
case 5:
exit(0);
        }
}
   } //End of main()
```

4.2.1.3 Algorithm For Deleting A Node

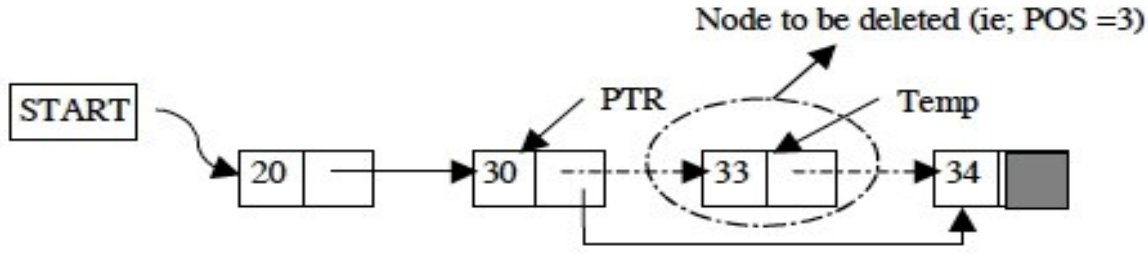


Fig12: Deletion of Node

Suppose START is the first position in linked list. Let DATA be the element to be deleted. TEMP, HOLD is a temporary pointer to hold the node address.

1. Input the DATA to be deleted
2. If (START → DATA is equal to DATA) // Data at first node
   (a) TEMP = START
   (b) START = START → Next
   (c) Set free the node TEMP, which is deleted (d) Exit
3. HOLD = START

4. while ((HOLD → Next → Next) not equal to NULL)) (a) if ((HOLD → NEXT → DATA) equal to DATA)
    (i) TEMP = HOLD → Next
    (ii) HOLD → Next = TEMP → Next
    (iii) Set free the node TEMP, which is deleted
    (iv) Exit
   (b) HOLD = HOLD → Next
5. if ((HOLD → next → DATA) == DATA)
   (a) TEMP = HOLD → Next
   (b) Set free the node TEMP, which is deleted
   (c) HOLD → Next = NULL
   (d) Exit
6. Disply "DATA not found"
7. Exit

Program: Deleting a Node from single linked list

```
//Delete any element from the linked list void
Delete(int item)
{
    struct node *tmp,*q;
if (start->info == data)
    {
tmp=start;
        start=start->next; /*First element deleted*/
free(tmp);
return;     }
q=start;
    while(q->next->next ! = NULL)
    {
        if(q->next->data == item) /*Element deleted in between*/
        {
            tmp=q->next;
q->next=tmp->next;
free(tmp);
return;         }
q=q->next;     }/*End of while
*/
    if(q->next->data==info) /*Last element deleted*/
    {
        tmp=q->next;
free(tmp);
q->next=NULL;
return;
    }
    printf ("\n\nElement %d not found",data);
}/*End of del()*/
```

4.2.1.4. Algorithm For Searching A Node

Suppose START is the address of the first node in the linked list and DATA is the information to be searched. After searching, if the DATA is found, POS will contain the corresponding position in the list.

1. Input the DATA to be searched
2. Initialize TEMP = START; POS =1;
3. Repeat the step 4, 5 and 6 until (TEMP is equal to NULL)
4. If (TEMP → DATA is equal to DATA)
   (a) Display "The data is found at POS"
   (b) Exit
5. TEMP = TEMP → Next
6. POS = POS+1
7. If (TEMP is equal to NULL)
   (a) Display "The data is not found in the list"
8. Exit

Code: Searching an element

```
//Function to search an element from the linked list
void Search(int item)
{
    struct node *ptr = start;
int pos = 1;
    //searching for an element in the linked list
while(ptr!=NULL)
    {
        if (ptr->data==item)
        {
            printf ("\n\nItem %d found at position %d", data, pos);
getch();              return;
        }
        ptr = ptr->next;
pos++;     }
    if (ptr == NULL)
        printf ("\n\nItem %d not found in list",data);
}
```

4.2.2. Stack Using Linked List

The following figure shows that the implementation of stack using linked list.

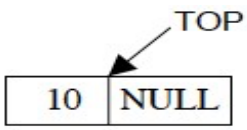


Fig13: Push (10)

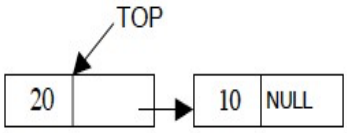


Fig14: Push (20)

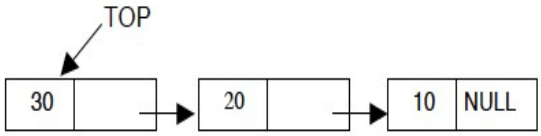


Fig15: Push (30)

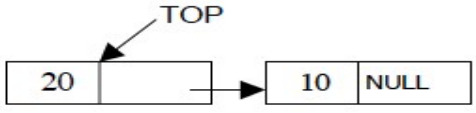


Fig16: X = Pop (i.e. X=30)

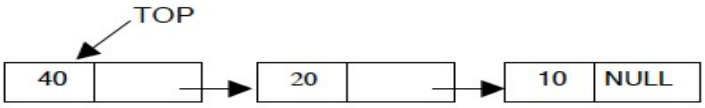


Fig17: Push (40)

4.2.2.1 Algorithm For Push Operation

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. DATA is the data item to be pushed.

1. Input the DATA to be pushed
2. Creat a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = TOP
5. TOP = NewNode
6. Exit

4.2.2.2. Algorithm For POP Operation

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.

1. if (TOP is equal to NULL)
   (a) Display "The stack is empty"
2. Else
   (a) TEMP = TOP
   (b) Display "The popped element TOP → DATA"
   (c) TOP = TOP → Next
   (d) TEMP → Next = NULL
   (e) Free the TEMP node
3. Exit

Program: Push, Pop and Display using Linked list on Stack

```
#include<stdio.h>
#include<stdlib.h>
//Structure is created a node struct
node
{     int
data;
    struct node *next;//A link to the next node
};
//A variable named NODE is been defined for the structure typedef
struct node *NODE;
//This function is to perform the push operation
NODE push(NODE top)
{
    NODE NewNode;
int pushed_item;
    //A new node is created dynamically
NewNode = (NODE)malloc(sizeof(NODE));
    printf("\nInput the new value to be pushed on the stack:");
scanf("%d",&pushed_item);
    NewNode->data=pushed_item;//Data is pushed to the stack
    NewNode->next=top;//Link pointer is set to the next node
top=NewNode;//Top pointer is set     return(top); }/*End of push()*/
//Following function will implement the pop operation
NODE pop(NODE top)
{
    NODE tmp;
    if(top == NULL)//checking whether the stack is empty or not
printf ("\nStack is empty\n");     else     {
```

```
        tmp=top;//popping the element
printf("\nPopped item is %d\n",tmp->data);
top=top->next;//resetting the top pointer
tmp->next=NULL;
        free(tmp);//freeing the popped node
    }
return(top);
}/*End of pop()*/
//This is to display the entire element in the stack
void display(NODE top)
{
    NODE tmp;
    if(top==NULL)
        printf("\nStack is empty\n");
else
    {    tmp = top;
        printf("\nStack elements:\n");
while(tmp != NULL)
        {
            printf("%d\n",tmp->data);
tmp = tmp->next;
        }/*End of while */
    }/*End of else*/
}/*End of display()*/
int main() {     char
opt;     int choice;
NODE Top=NULL;
while(1){
        printf("\n1.PUSH\n");
printf("2.POP\n");
printf("3.DISPLAY\n");
printf("4.EXIT\n");
printf("\nEnter your choice:");
scanf("%d", &choice);
switch(choice)
        {
case 1:
 Top=push(Top);
break;
case 2:
Top=pop(Top);
break;
case 3:
display(Top);
break;
case 4:
exit(1);
default:
printf("\nWrong choice\n");
        }/*End of switch*/
    }
}/*End of main() */
```

4.2.3. Queue Using Linked List

The following figure shows that the implementation issues of Queue using linked list.

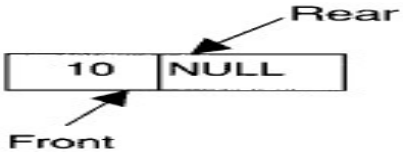


Fig18: Enqueue (10)

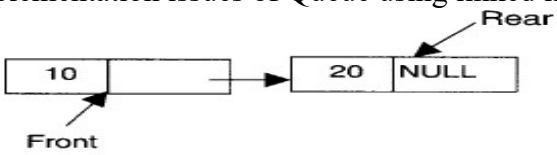
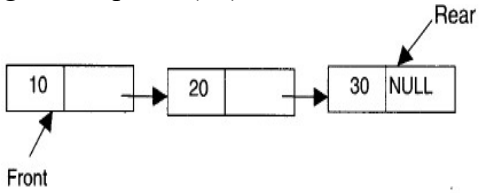


Fig19: Enqueue (20)



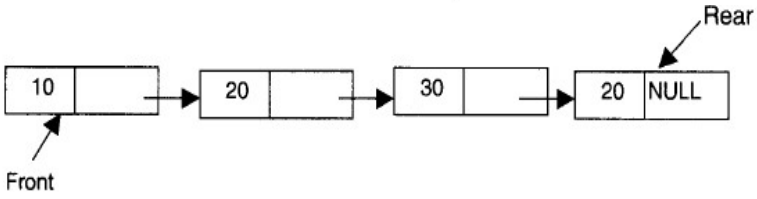


Fig20: Enqueue (30)

Fig21: Enqueue (20)

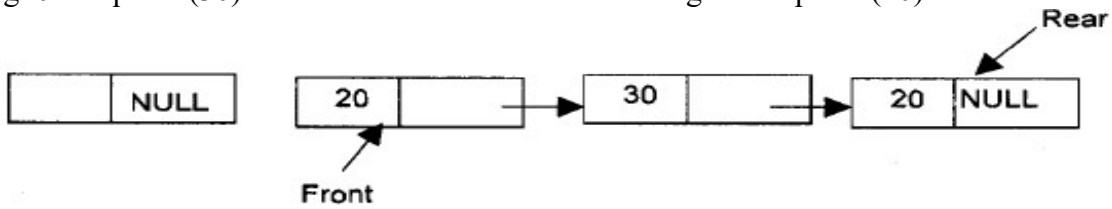


Fig22: X = Dequeue (i.e. X = 10)

4.2.3.1. Algorithm for pushing an element into a Queue

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element to be pushed.

1. Input the DATA element to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
5. If(REAR not equal to NULL)
   (a) REAR → next = NewNode;
6. REAR =NewNode;
7. Exit

4.2.3.2. Algorithms for popping an element from the queue

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element popped from the queue.

1. If (FRONT is equal to NULL)
   (a) Display "The Queue is empty"
2. Else
   (a) Display "The popped element is FRONT → DATA" (b) If (FRONT is not equal to REAR)
   (i) FRONT = FRONT → Next
   (c) Else
   (d) FRONT = NULL;
3. Exit

Code: Queue using Linked List (Enqueue, Dequeue and Traverse)

```
//A structure is created for the node in queue
#include <stdio.h>
#include <stdlib.h>
struct queue {
int data;
    struct queue *next;//Next node address
};
typedef struct queue *NODE;
//This function will Enqueue an element into the queue
NODE Enqueue(NODE rear)
{
    NODE NewNode;
    //New node is created to push the data
NewNode=(NODE)malloc(sizeof(struct queue));
printf ("\nEnter the no to be pushed = ");
    scanf ("%d",&NewNode->data);
NewNode->next=NULL;
//setting the rear pointer
if (rear != NULL)         rear-
>next=NewNode;     rear=NewNode;
return(rear);
}
//This function will Dequeue the element from the queue
NODE Dequeue(NODE f,NODE r)
{
    //The Queue is empty when the front pointer is NULL
if(f==NULL)
        printf ("\nThe Queue is empty");
else     {
        printf ("\nThe poped element is = %d\n",f->data);
if(f != r)             f=f->next;         else
f=NULL;
    }
return(f);
}
//Function to display the element of the queue void
Traverse(NODE fr,NODE re)
{
    //The queue is empty when the front pointer is NULL
if (fr==NULL)
        printf ("\nThe Queue is empty");
else     {
        printf ("\nThe element(s) is/are = ");
while(fr != re)
        {
            printf("%d ",fr->data);
fr=fr->next;
        };
        printf ("%d",fr->data);
printf("\n");
    }
}  int
main() {
```

```
    int choice;
    //declaring the front and rear pointer
    NODE front, rear;
    //Initializing the front and rear pointer to NULL
front = rear = NULL;     while(1)     {
        printf ("1. Enqueue\n");
printf ("2. Dequeue\n");         printf ("3.
Traverse\n");         printf ("4. Exit\n");
printf ("\n\nEnter your choice = ");
scanf ("%d",&choice);         switch(choice)
        {   case 1:
                //calling the Enqueue function
rear = Enqueue(rear);                 if
(front==NULL)
                {
                    front=rear;
                }
break;             case
2:
                //calling the Dequeue function by passing
                //front and rear pointers
front = Dequeue(front,rear);
if (front == NULL)                     rear
= NULL;                 break;
case 3:
                Traverse(front,rear);
break;             case 4:
                exit(0);
default:                 printf("Try
Again\n");
        }
    }
}
```

Advantages of Singly Linked List

1. Accessibility of a node in the forward direction is easier.
2. Insertion and deletion of node are easier.

Disadvantages of Singly Linked List

1. Can insert only after a referenced node
2. Removing node requires pointer to previous node
3. Can traverse list only in the forward direction

4.2.4. DOUBLY LINKED LIST

A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list. Every nodes in the doubly linked list has three fields: LeftPointer (Prev), RightPointer (Next) and DATA.

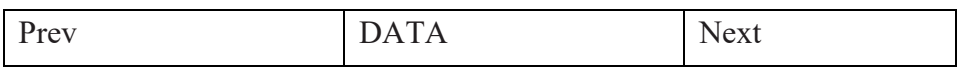


Fig23: Typical Doubly Linked list node

Prev will point to the node in the left side (or previous node) i.e. **Prev** will hold the address of the previous node. **Next** will point to the node in the right side (or next node), i.e. **Next** will hold the address of next node. DATA will store the information of the node.

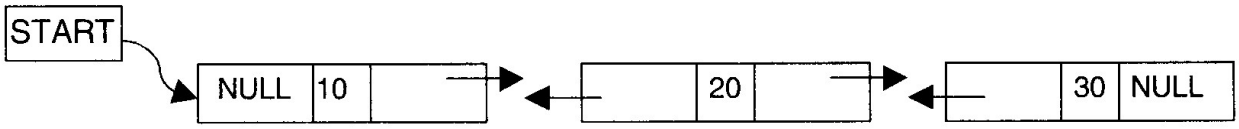


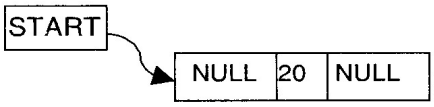
Fig24: Doubly Linked List

Representation of Doubly Linked List

Following declaration can represent doubly linked list

```
struct Node
{     int data;
struct Node *Next;
struct Node *Prev;
};
typedef struct Node *NODE;
```

All the primitive operations performed on singly linked list can also be performed on doubly linked list. The following figures shows that the insertion and deletion of nodes.



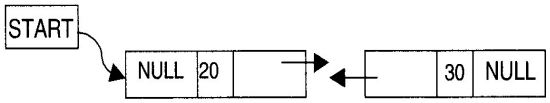


Fig25: Add (20)

Fig26: Insert (30) at the end

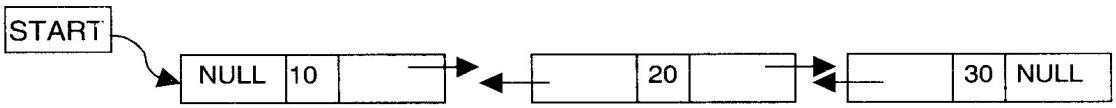


Fig27: Insert (10) at the Beginning

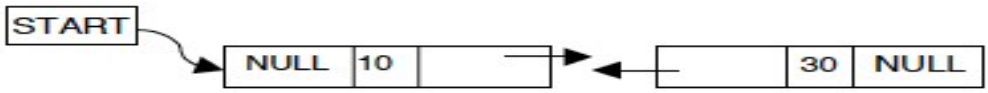


Fig28: Delete a node at the 2nd position (Delete 20 at 2nd position)

4.2.4.1. Algorithm for Inserting a Node

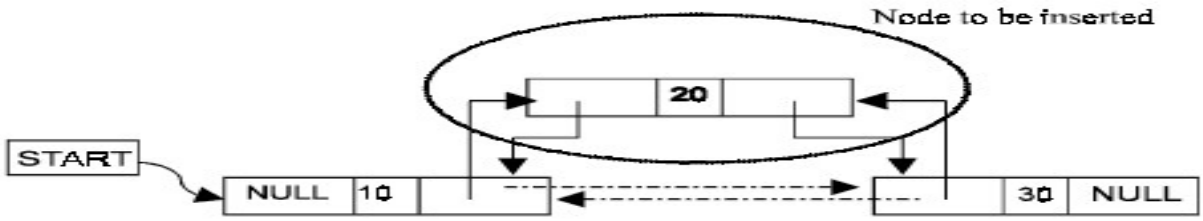


Fig29: Insert a node at the 2nd position

Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the NewNode is to be inserted. TEMP is a temporary pointer to hold the node address.

1. Input the DATA and POS
2. Initialize TEMP = START; i = 0
3. Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
4. TEMP = TEMP → Next; i = i +1
5. If (TEMP not equal to NULL) and (i equal to POS)
    a) Create a New Node
    b) NewNode → DATA = DATA
    c) NewNode → Next = TEMP → Next
    d) NewNode → Prev = TEMP
    e) (TEMP → Next) → Prev = NewNode
    f) TEMP → Next = New Node
6. Else
    a) Display "Position NOT found"
7. Exit

4.2.4.2. Algorithm for Deleting a Node

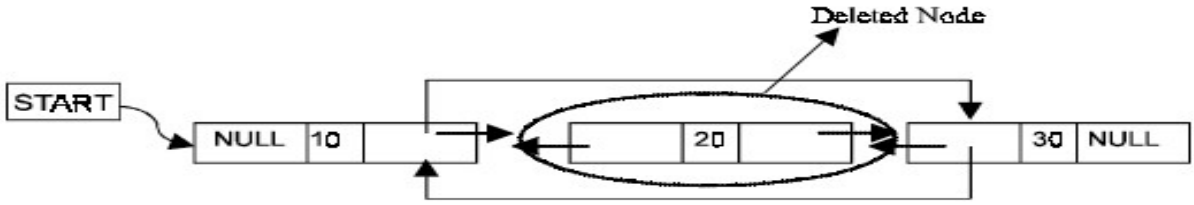


Fig30: Delete a node at the 2nd position

Suppose START is the address of the first node in the linked list. Let POS is the position of the node to be deleted. TEMP is the temporary pointer to hold the address of the node. After deletion, DATA will contain the information on the deleted node.

1. Input the POS
2. Initialize TEMP = START; i = 0
3. Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
4. TEMP = TEMP → Next; i = i +1
5. If (TEMP not equal to NULL) and (i equal to POS)
    a. Create a New Node
    b. NewNode → DATA = DATA
    c. NewNode → Next = TEMP → Next
    d. NewNode → Prev = TEMP
    e. (TEMP → Next) → Prev = NewNode
    f. TEMP → Next = New Node
6. Else
    a. Display "Position NOT found" 7. Exit

Source code to Implement the primitives operations on Doubly Linked List

```
//Structure is created for the node
struct node {
    struct node *prev;
int data;
struct node *next;
};
struct node *NODE;
NODE *start;
//fucntion to create a doubly linked list void
create_list(
{

    //a new node is created
    start=(NODE)malloc(sizeof(struct node));
start->next=NULL;
start->prev=NULL;
}/*End of create_list()*/
//Functcion to add new node at the beginning
void addatbeg(int num)
{
    NODE tmp;
    //a new node is created for inserting the data
tmp=(NODE)malloc(sizeof(struct node));
tmp->prev=NULL;
tmp->data=num;
tmp->next=start;
start->prev=tmp;
start=tmp; }/*End of addatbeg()*/
//This fucntion will insert a node in any specific position void
addafter(int num,int pos)
{
    NODE tmp,q;
int i;
q=start;
    //Finding the position to be inserted
for(i=0;i<pos-1;i++)
    {
 q=q->next;
if(q==NULL)
{
            printf ("\nThere are less than %d elements\n",pos);
return;
        }
    }
    //a new node is created
    tmp=(NODE)malloc(sizeof(struct node) );
tmp->data=num;
q->next->prev=tmp;
tmp->next=q->next;
tmp->prev=q;
q->next=tmp;
}/*End of addafter() */
```

```
//Function to delete a node void
del(int num)
{
    NODE tmp,q;
if(start->data==num)
    {
tmp=start;
        start=start->next; /*first element deleted*/
start->prev = NULL;
        free(tmp);//Freeing the deleted node
return;     }
q=start;
    while(q->next->next!=NULL)
    {
        if(q->next->data==num) /*Element deleted in between*/
        {
tmp=q->next;
q->next=tmp->next;
tmp->next->prev=q;
free(tmp);
return;         }
q=q->next;
    }
    if (q->next->data==num) /*last element deleted*/
    { tmp=q->next;
free(tmp);
q->next=NULL;
return;     }
    printf("\nElement %d not found\n",num);
}/*End of del()*/

//Displaying all data(s) in the node
void display() {
NODE q;     if(start==NULL)
    {
        printf("\nList is empty\n");
return;     }
q=start;
    printf("\nList is :\n");
while(q!=NULL)
    {
        printf("%d ", q->data);
q=q->next;
    }
    printf("\n"); }/*End
of display() */
```

4.2.5. CIRCULAR LINKED LIST

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node. Circular linked lists also make our implementation easier, because they eliminate the boundary conditions associated with the beginning and end of the list, thus eliminating the special case code required to handle these boundary conditions.

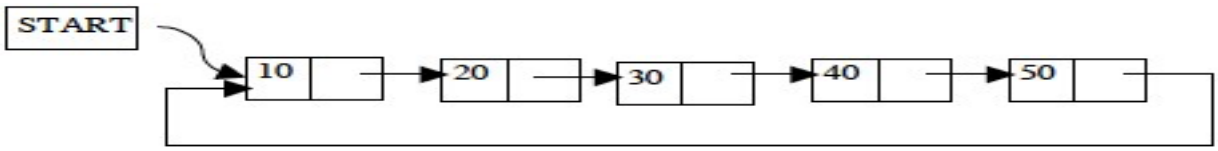


Fig31: Circular Linked List

A circular doubly linked list has both the successor pointer and predecessor pointer in circular manner as shown in the following Fig.

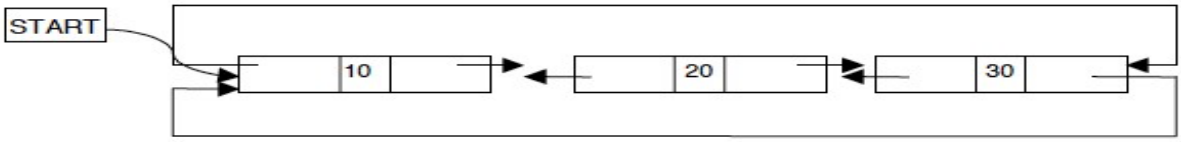


Fig32: Circular Doubly Linked List

4.2.5.1. Algorithm

1. Inserting a node at the beginning (*Insert_First(START,Item***)**

1. Create a Newnode
2. IF START is equal to NULL then
    a. Set Newnode->data = item
    b. Set NewNode->next = Newnode
    c. Set START = Newnode
    d. Set Last = Newnode
3. Set Newnode ->data = Item
4. Set Newnode->next = Start
5. Set START = Newnode
6. Set last->next = Newnode

2. Inserting a node at the End (*Insert_End(START,Item***)**

1. Create a Newnode
2. If START is equal to NULL, then
    a. Newnode->data = Item
    b. Newnide->next = Newnode
    c. Last = Newnode
    d. START = Newnode
3. Newnode->data = Item
4. Last->next = Newnode
5. Last = Newnode
6. Last->next = START

3. Deleting a node from beginning (*Delete_First(START***)**

1. Declare a temporary node, ptr
2. If START is equal to NULL, then
    a. Display empty Circular queue
    b. Exit
3. Ptr = START
4. START = START->next
5. Print, Element deleted is , ptr->data
6. Last->next = START
7. Free (ptr)

4. Deleting a Node from End (*Delete_End(START***)**

1. Declare a temporary node , ptr

2. If START is equal to NULL , then
    a. Print Circular list empty
    b. Exit
3. Ptr = START
4. Repeat step 5 and 6 until ptr !=Last
5. Ptr1 = ptr
6. Ptr = ptr->next
7. Print element deleted is ptr->data
8. Ptr1->next = ptr->next
9. Last = ptr1
10. Return START

Program:

```
/*Program to demonstrate circular singly linked list. This program will add
an element at the beginning of Linked list. */
#include <stdio.h>
#include <stdlib.h>
 struct node {
 int data;
struct node *next;
};
struct node *head = NULL, *tail = NULL;

struct node * createNode(int item)
{     struct node *newnode;
    newnode = (struct node *)malloc(sizeof (struct node));
newnode->data = item;
newnode->next = NULL;
return newnode;
}

/* create dummy head and tail to make insertion and deletion operation
simple.While processing data in our circular linked list. */

void createDummies() {
head = createNode(0);
tail = createNode(0);
head->next = tail;
tail->next = head;
}

/* insert data next to dummy head */
void circularListInsertion(int data) {
struct node *newnode, *temp;
newnode = createNode(data);
temp = head->next;
head->next = newnode;
newnode->next = temp;
}

/* Delete the node that has the given data */
void DeleteInCircularList(int data)
{
struct node *temp0, *temp;
```

```
    if (head->next == tail && tail->next == head) {
printf("Circular Queue/list is empty\n");
    }
temp0 = head;
temp = head->next;
while (temp != tail)
{
if(temp->data == data) {
temp0->next = temp->next;
free(temp);
 break;
}
temp0 = temp;
temp = temp->next;
    }
return;
}

/* travese the circular linked list. */
void traverse()
{     int n = 0;
    struct node *tmp = head->next;
if (head->next == tail && tail->next == head) {
printf("Circular linked list is empty\n");
return;
 }
while (tmp !=tail)
{
printf("%d->", tmp->data);
tmp = tmp->next;
    }
return;
}
  int main() {
int data, ch;
createDummies();
while (1) {
        printf(" \n1.Insertion\t2. Deletion\n");
printf("3. Display\t4. Exit\n");
printf("Enter ur choice:");
scanf("%d", &ch);
switch (ch) {
case 1:
printf("Enter the data to insert:");
scanf("%d", &data);
circularListInsertion(data);
break;
case 2:
printf("Enter the data to delete:");
scanf("%d", &data);
DeleteInCircularList(data);
break;
case 3:
traverse();
break;
```

```
case 4:
exit(0);
default:
printf("Pls. enter correct option\n");
break;
        }
}
return 0;
}
```

4.2.6. 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.

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

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

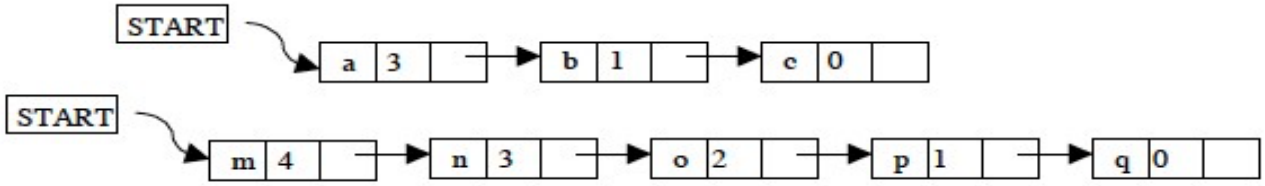


Fig33: Polynomial Representation in Linked List These two polynomials can be added by

$h(x) = f(x) + g(x) = mx^4 + (a + n)\,x^3 + ox^2 + (b + p)x + (c + q)$

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

Fig34: Addition of polynomials

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.

Program:

```
#include<stdio.h>
#include<stdlib.h>
//structure is created for the node struct
node
{     int coef;
int expo;
struct node *link;
};
typedef struct node *NODE;
//Function to add any node to the linked list
NODE insert(NODE start,float co,int ex)
{
    NODE ptr,tmp;
    //a new node is created
    tmp= (NODE)malloc(sizeof(struct node));
tmp->coef=co;
tmp->expo=ex;
    /*list empty or exp greater than first one */
if(start==NULL || ex>start->expo)
    {
        tmp->link=start;//setting the start
start=tmp;
    }
else
{
        ptr=start;
        while(ptr->link!=NULL && ptr->link->expo>ex)
ptr=ptr->link;
tmp->link=ptr->link;
ptr->link=tmp;
        if(ptr->link==NULL) /*item to be added in the end */
tmp->link=NULL;
    }
return start; }/*Endof insert()*/
//This function is to add two polynomials
NODE poly_add(NODE p1,NODE p2)
{
    NODE p3_start,p3,tmp;
p3_start=NULL;
if(p1==NULL && p2==NULL)
return p3_start;
while(p1!=NULL && p2!=NULL )
    {
        //New node is created
        tmp=(NODE)malloc(sizeof(struct node));
if(p3_start==NULL)
        {
```

```
            p3_start=tmp;
p3=p3_start;
        }
else
{
            p3->link=tmp;
p3=p3->link;
        }
        if(p1->expo > p2->expo)
        {
            tmp->coef=p1->coef;
tmp->expo=p1->expo;
p1=p1->link;
        }
else
            if(p2->expo > p1->expo){
tmp->coef=p2->coef;
 tmp->expo=p2->expo;
p2=p2->link;
            }
else
                if(p1->expo == p2->expo)
                {
                    tmp->coef=p1->coef + p2->coef;
tmp->expo=p1->expo;
p1=p1->link;
p2=p2->link;
                }
}/*End of while*/
while(p1!=NULL)
    {
        tmp=(NODE)malloc(sizeof(struct node));
tmp->coef=p1->coef;
tmp->expo=p1->expo;
        if (p3_start==NULL) /*poly 2 is empty*/
        {
            p3_start=tmp;
p3=p3_start;
        }
        else         {
p3->link=tmp;
p3=p3->link;
        }
p1=p1->link;
}/*End of while */
while(p2!=NULL)
    {
        tmp=(NODE)malloc(sizeof(struct node));
tmp->coef=p2->coef;
tmp->expo=p2->expo;
        if (p3_start==NULL) /*poly 1 is empty*/
        {
p3_start=tmp;
p3=p3_start;
```

```
        }         else{
p3->link=tmp;
p3=p3->link;
        }
p2=p2->link;     }/*End
of while*/
p3->link=NULL;
return p3_start; }/*End
of poly_add() */
//Inputting the two polynomials
NODE enter(NODE start)
{
int i,n,ex;
int co;
    printf("\nHow many terms u want to enter:");
scanf("%d",&n);
for(i=1;i<=n;i++)
    {
        printf("\nEnter coeficient for term %d:",i);
scanf("%d",&co);
        printf("Enter exponent for term %d:",i);
scanf("%d",&ex);
start=insert(start,co,ex);
    }
return start;
}/*End of enter()*/
//This function will display the two polynomials and its
//added polynomials void
display(NODE ptr)
{
    if (ptr==NULL)
    {
        printf ("\nEmpty\n");
return;     }
    while(ptr!=NULL)
    {
        printf ("%dx^%d ", ptr->coef,ptr->expo);
ptr=ptr->link;
if(ptr != NULL)
printf(" +");
    }
}/*End of display()*/
int main()
{
    NODE p1_start,p2_start,p3_start;
p1_start=NULL;
p2_start=NULL;
p3_start=NULL;
    printf("\nPolynomial 1 :\n");
p1_start=enter(p1_start);
printf("\nPolynomial 2 :\n");
p2_start=enter(p2_start);
    //polynomial addition function is called
p3_start=poly_add(p1_start,p2_start);
```

```
    printf("\nPolynomial 1 is:");
display(p1_start);
printf ("\nPolynomial 2 is:");
display(p2_start);
    printf ("\nAdded polynomial is:");
display(p3_start);
}/*End of main()*/
```

Advantages

1. Polynomials of any degree can be represented.
2. Memory allocated to the nodes only when it is required and deallocated when the nodes are no more required.

Generalized Linked List

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')

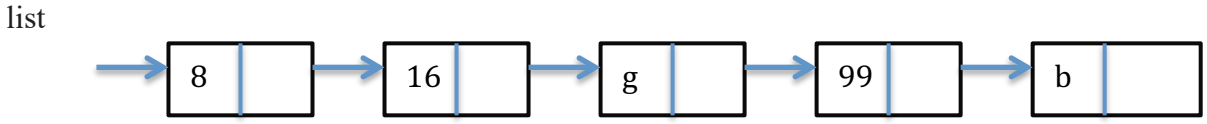


Fig35 (a): Generalized Linked List

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))

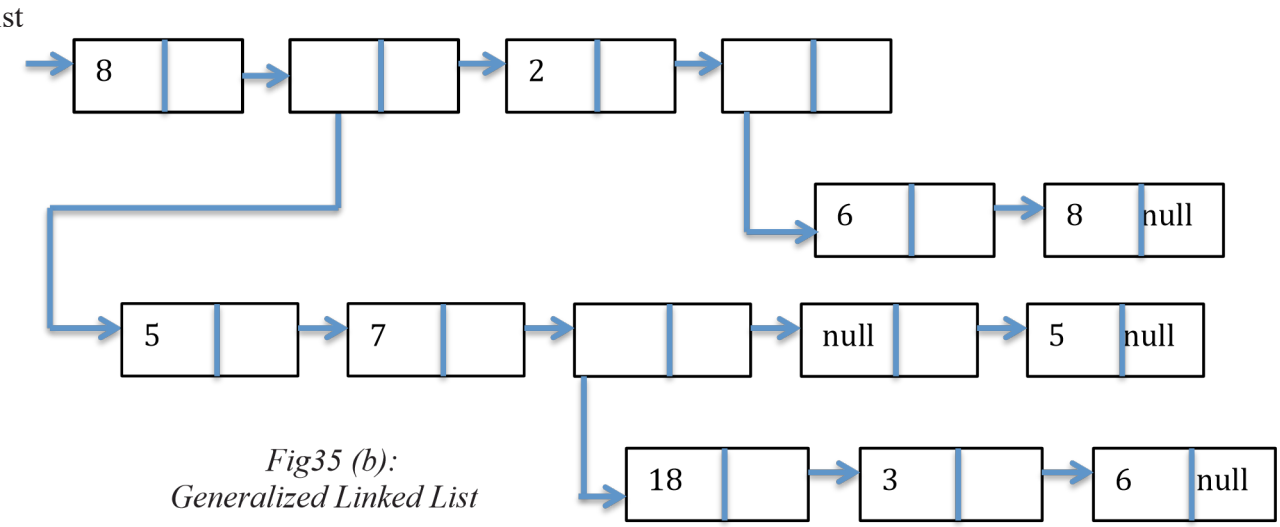


Fig35 (b): Generalized Linked List