Module 2-Queues & Linked Lists

Module 2 Syllabus	**QUEUES:** Queues, Circular Queues, Using Dynamic Arrays, Multiple Stacks and queues. **LINKED LISTS :** Singly Linked, Lists and Chains, Representing Chains in C, Linked Stacks and Queues, Polynomials

2.1 QUEUES

"A queue is an ordered list in which insertions (additions, pushes) and deletions (removals and pops) take place at different ends". The end at which new elements are added is called the rear, and that from which old elements are deleted is called the front. If the elements are inserted A, B, C, D and E in this order, then A is the first element deleted from the queue. Since the first element inserted into a queue is the first element removed, queues are also known as First-In-First-Out (FIFO) lists.

2.1.1 Array Representation of Queues

Queues can be easily represented using linear arrays. As stated earlier every queue has front and rear variables that point to the position from where deletions and insertions can be done, respectively.

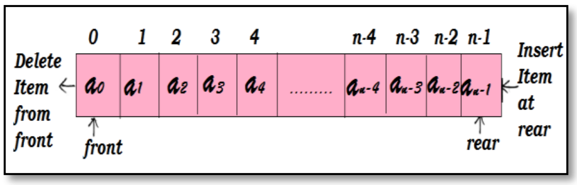


Fig. 2.1: Array Representation of a Queue

The array representation of a queue is shown in Fig. 2.1.Queues may be represented by one-way lists or linear arrays and it will be maintained by a linear array QUEUE and two pointer variables:

- **FRONT**-containing the location of the front element of the queue.
- **REAR**-containing the location of the rear element of the queue.

The condition FRONT = NULL will indicate that the queue is empty.

2.1.2 Queue Operations

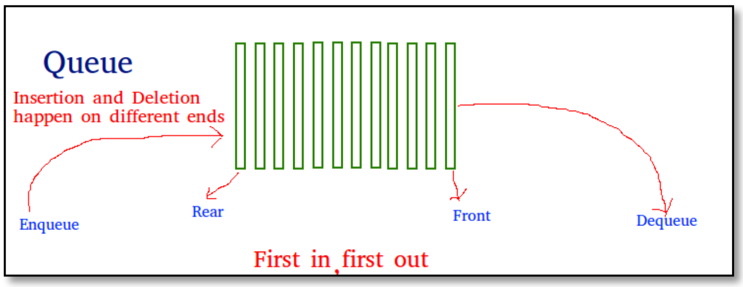


Fig. 2.2: Example for Insertion and Deletion from Stack

- **enqueue()** − add (store) an item to the queue from front end.
- **dequeue()** − remove (access) an item from the queue from rear end.
- **isfull() / overflow**− Checks if the queue is full. If queue is full and if we try to insert one more element, then overflow occurs.
- **isempty() /underflow**− Checks if the queue is empty.If queue is empty and if we try to remove one more element, then underflow occurs.

i)Queue Create

The maximum number of elements that can be entered into queue can be defined as:

```
#define MAX 5
```

Since queue is an array, we can declare it as:

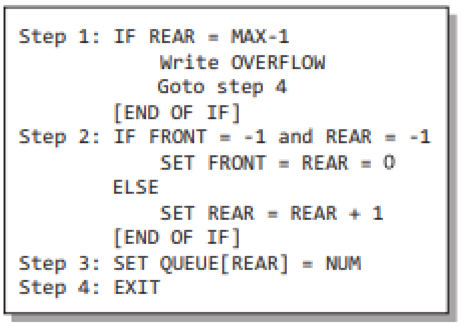
```
int q[MAX];
```

The variables 'front' and 'rear' are associated with q and holds the index of 1st element and index of rear element. Initially, front is initialized to -1 and rear to -1. When an item is to be inserted into queue, increment rear by 1 and insert item to that position where rear is pointed.

```
int front = -1, rear = -1;
```

ii) Queue Insert and Overflow

When we want to add elements in to the queue the first step is to check for the overflow condition. Next we check whether queue is empty, incase if the queue is empty we set front and rear both equal to 0,so that new value can be stored at 0th location. Otherwise, if the queue already has some values, then rear is incremented so that it points to the next location in the array. The algorithm to insert an element into the queue is stated below in the figure 2.3.



```
Step 1: IF REAR = MAX-1
            Write OVERFLOW
            Goto step 4
        [END OF IF]
Step 2: IF FRONT = -1 and REAR = -1
            SET FRONT = REAR = 0
        ELSE
            SET REAR = REAR + 1
        [END OF IF]
Step 3: SET QUEUE[REAR] = NUM
Step 4: EXIT
```

Fig. 2.3: Algorithm to insert an element in a queue

Example:

Elements 10, 20, 30, 40 and 50 i entered into queue as in Fig. 2.4.

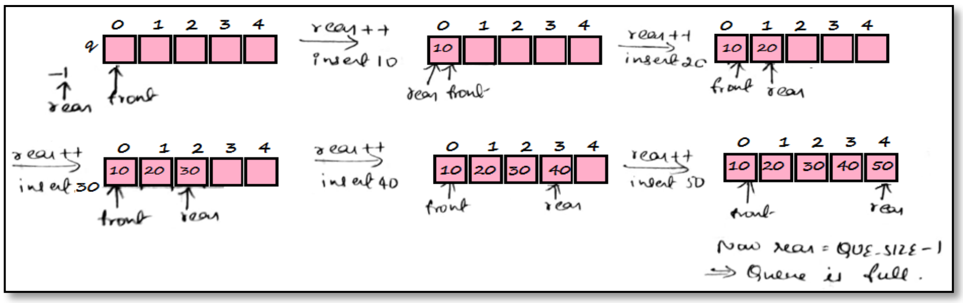


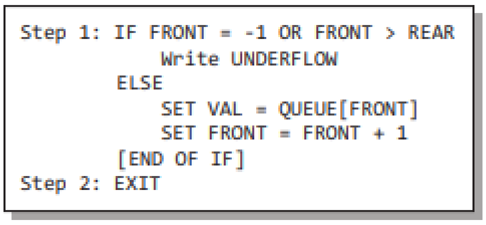
Fig. 2.4: Queue Insert

Program code for insert queue operation

```
void insert()
{
int item;
if(rear == MAX - 1)
{       printf("Queue Overflow n");
        return;
}
printf("Inset the element in queue : ");
scanf("%d", &item);
rear = rear + 1;
q[rear] = item;
}
}
```

iii) Queue Delete and Underflow

In Step 1, we check for underflow condition. An underflow occurs if FRONT = –1 or FRONT > REAR. However, if queue has some values, then FRONT is incremented so that it now points to the next value in the queue. The algorithm to delete an element from the queue is stated below in the figure 2.5.



```
Step 1: IF FRONT = -1 OR FRONT > REAR
            Write UNDERFLOW
        ELSE
            SET VAL = QUEUE[FRONT]
            SET FRONT = FRONT + 1
        [END OF IF]
Step 2: EXIT
```

Fig. 2.5: Algorithm to delete an element from the queue

Example:

Initially, rear points to -1. So when queue is empty, rear should point to -1 and front should point to 0 (Fig 2.6(a)). Elements are deleted from the front end. For example, delete 10, 20, 30, 40 and 50 from the queue in the fig. 2.4 will result in the queue in the fig.2.6(b). When front>rear it results empty queue. Hence we need to make the initial condition as front=0 and rear = -1 when front>rear.

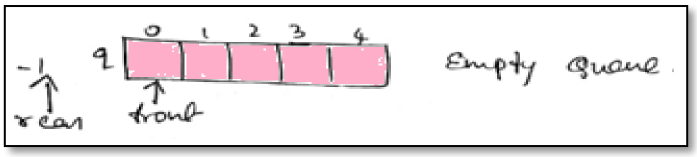


Fig. 2.6(a): Empty Queue

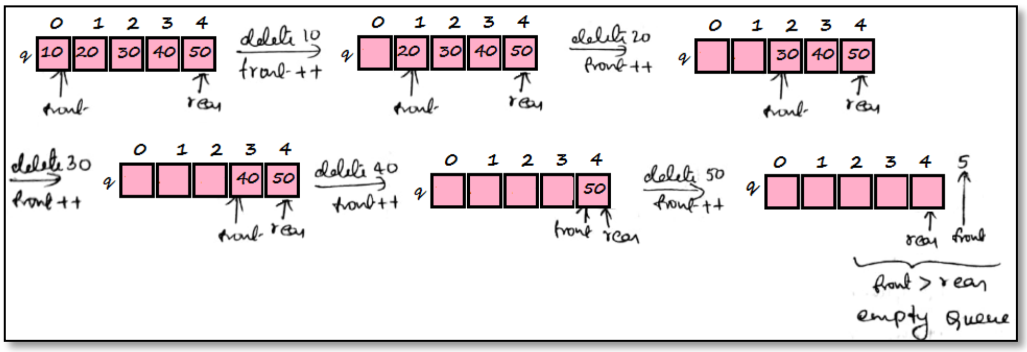


Fig.2.6(b): Delete from Queue

Program code for delete queue operation

```
void delete()
{
int item;
if(front>rear)
{         printf("Queue underflow n");
          return;
}
item=q[front];
printf("deleted element is:%d ", item);
front++;
if(front>rear)
{
          front=0;
          rear=-1;
}
}
```

iv) Queue Display

If queue is having some items, then it should be displayed one after the other. If there is no item, then it should display error message.

Case – 1: if the queue is empty then

```
if(front>rear)
{               printf("Queue is empty");
                return;
}
```

Case – 2: if the queue is having some items, then

```
for(i=front;i<rear;i++)
                printf("%d\t",q[i]);
```

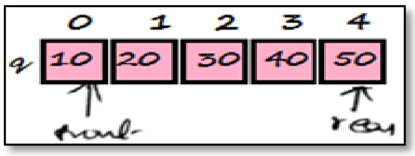


Fig.2.7. Queue with items

Algorithm for display queue operation

Step 1 − Check if the queue is empty.

Step 2 − If the queue is empty, produce empty queue error and exit.

Step 3 − If the queue is not empty, display from front to rear.

Program code for display queue operation

```
void display() {
int i;
if(front>rear)
{               printf("Queue is empty");
                return;
}

printf(" The elements of queue are:\n");
for(i=front;i<rear;i++)
                printf("%d\t",q[i]);
}
```

2.2 Circular Queues

- A circular queue in C stores the data in a very practical manner. It is a linear data structure. It is very similar to the queue. The only difference is that the last node is connected back to the first node. Thus it is called a circular queue. Circular Queue is also called ring Buffer.

- "A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle."
- In a normal Queue Data Structure, we can insert elements until queue becomes full. But once the queue becomes full, we cannot insert the next element until all the elements are deleted from the queue. For example, consider the queue in the Fig. 2.8(a).Now consider the Fig.2.8 (b), the situation after deleting three elements from the queue. This situation also says that Queue is full and we cannot insert the new element because '**rear**' is still at last position. In the above situation, even though we have empty positions in the queue we cannot make use of them to insert the new element. This is the major problem in a normal queue data structure. To overcome this problem we use a circular queue data structure.

- When the array is viewed as a circle, each array position has a next and a previous position. The position next to position MAX-1 is 0,and the position that precedes 0 is MAX-1,the next element is put into position 0.
- To work with the circular queue, we must be able to move the variables front and rear from their current position to the next position(clockwise).
- If front =-1 and rear =-1 we cannot distinguish between queue empty and queuefull. To avoid this confusion we set front=0 and rear =0 in circular queue.

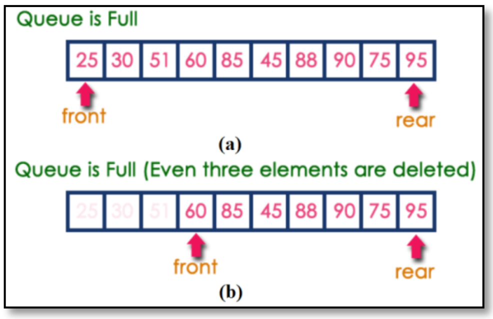


Fig. 2.8: Example for Normal Queue

Different operations of circular Queue.

1. Create Circular Queue

We have to define the maximum size of circular queue.

```
#define MAX 3
```

Initially both the front and rear points to 0 in a circular queue.

```
int front=0, rear=0;
```

Declare a queue array of size MAX as:

```
int q[MAX];
```

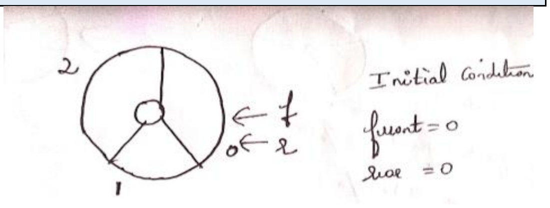


Fig. 2.9: Empty Circular Queue

2. Insert and Overflow in Circular Queue

In a circular queue, the new element is always inserted at **rear** position. Suppose we want to insert element 10, 20, 30 to the circular queue in the Fig. 2.9. When front = 0. Fig.2.10 shows the step by step procedure to insert into CQ.

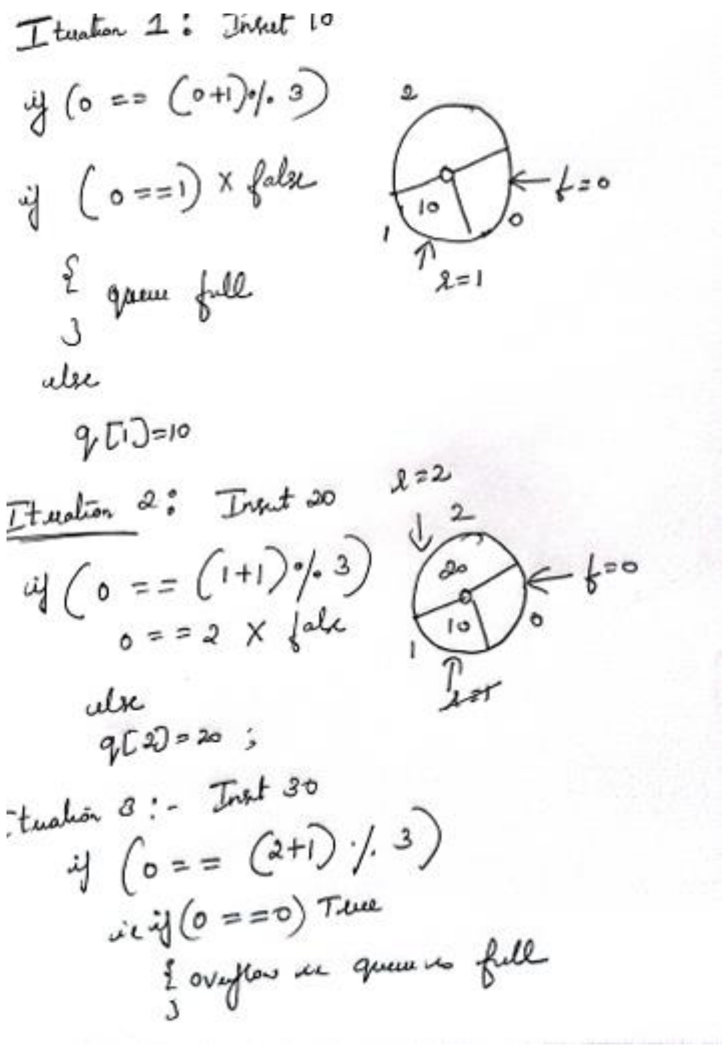


Fig.2.10: Insertion Operation of Circular Queue

Insert Function

```
void insert()
{
int item;
if (front==(rear+1)%MAX)
{
printf("\n Circular Queue overflow");
return;
}
else
{
rear=(rear+1)%MAX;
printf("\ n Enter the element to be inserted");
scanf("%d",&item);
q[rear]=item;
}
}
```

3. Delete and Underflow in Circular Queue

We continue deletion of element from the last CQ got in Fig. 2.10. In a circular queue, the element is always deleted at **front** position. So delete 10 first since element of q[front] is 10.Now front should point to 1, since next element to be deleted is 20. Fig.2.11 shows the step by step procedure to delete from CQ. When front=0 and we try to delete more items, then queue underflow occurs.

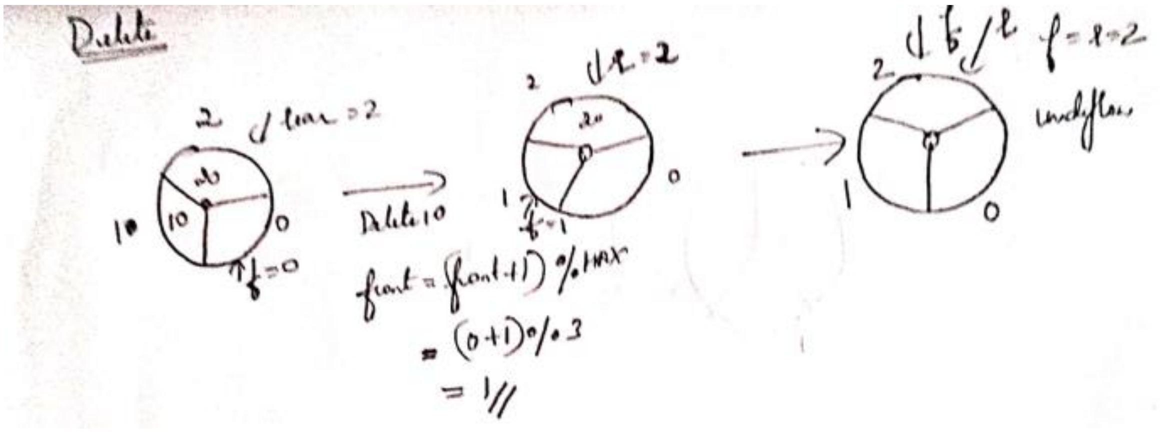
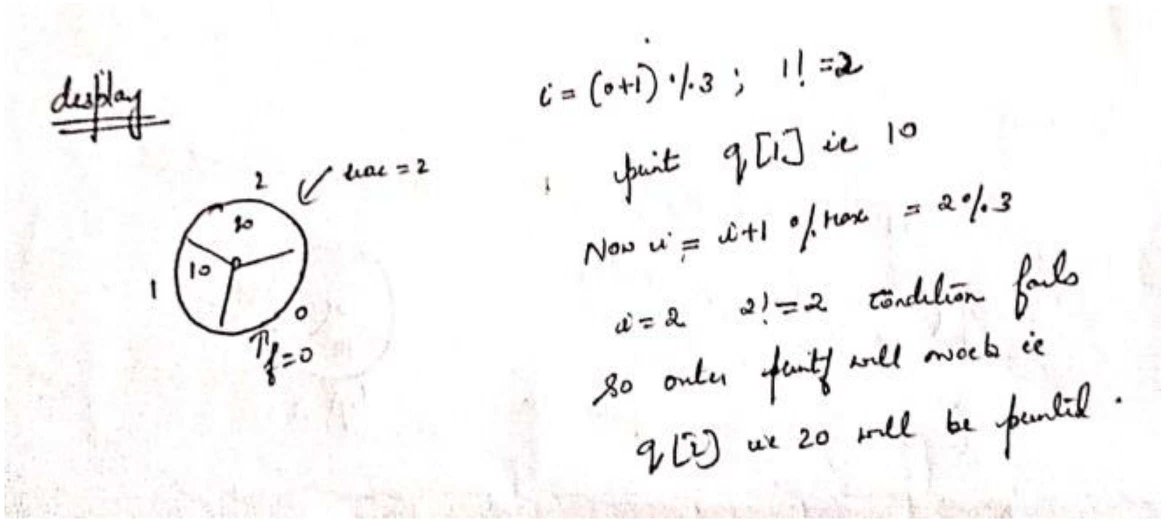


Fig. 2.11: Deletion Operation of Circular Queue

Delete Function

```
void delete()
{
if(front==rear)
{
printf(" Circular Queue underflow");
return;
}
else
{
front=(front+1)%MAX;
printf("\n deleted element is %d",q[front]);
}
}
```

4. Display in Circular Queue



i = (0+1) % 3 ; 1 != 2

print q[1] ie 10

Now i = i+1 % max = 2 % 3

i = 2   2 != 2 condition fails

so outer printf will work ie

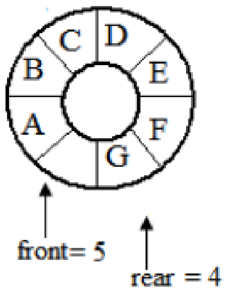
q[2] ie 20 will be printed.

Fig. 2.12: Display in Circular Queue

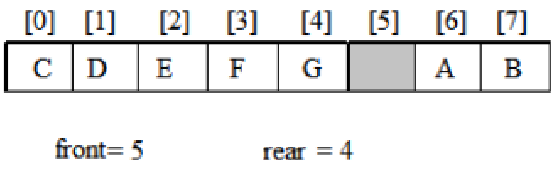
```
void display()
{
int i;
if(front==rear)
{
printf("circular queue is empty\n");
}
else
{
printf("\n contents of circular queue");
for(i=(front+1)%MAX ;i!=rear;i=(i+1)%MAX)
{
printf("%d\t",q[i]);
}
printf("%d\t",q[i]);
}
}
```

2.3 Circular Queues using Dynamic Arrays

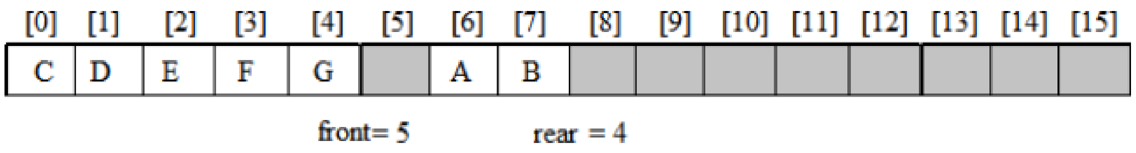
- A dynamically allocated array is used to hold the queue elements. Let capacity be the number of positions in the array queue.
- To add an element to a full queue, first increase the size of this array using a function realloc. As with dynamically allocated stacks, array doubling is used
- Consider the full queue of figure (a). This figure shows a queue with seven elements in an array whose capacity is 8. A circular queue is flatten out the array as in Figure (b). Figure (c) shows the array after array doubling by realloc



(a) A full circular queue



(b) Flattened view of circular full queue



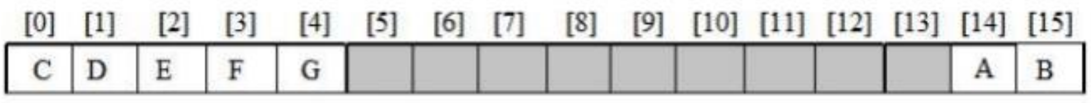
(c) After array doubling

- To get a proper circular queue configuration, slide the elements in the right segment (i.e., elements A and B) to the right end of the array as in figure (d).
- To obtain the configuration as shown in figure (e), follow the steps

  1) Create a new array newQueue of twice the capacity.

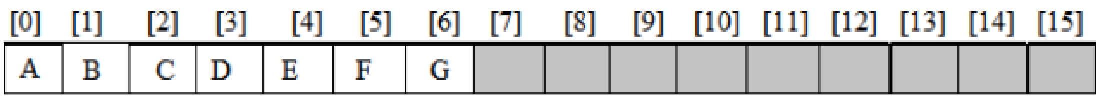
  2) Copy the second segment (i.e., the elements queue [front +1] through queue [capacity-1]) to positions in newQueue beginning at 0.

  3) Copy the first segment (i.e., the elements queue [0] through queue [rear]) to positions in newQueue beginning at capacity – front – 1.



front= 13 rear = 4

(d After shifting right segment



front= 15 rear = 6

(e) Alternative configuration

- Below program gives the code to add to a circular queue using a dynamically allocated array.

```
void addq( element item)
{
/* add an item to the queue
rear = (rear +1) % capacity;
if(front == rear)
queueFull( ); /* double capacity */
queue[rear] = item;
 }
```

- Below program obtains the configuration of figure (e) and gives the code for queueFull. The function copy (a,b,c) copies elements from locations a through b-1 to locations beginning at c.

```
void queueFull( ) {
/* allocate an array with twice the capacity */
element *newQueue;
MALLOC ( newQueue, 2 * capacity * sizeof(* queue));
/* copy from queue to newQueue */
 int start = ( front + 1 ) % capacity; if ( start < 2)
/* no wrap around */
copy( queue+start, queue+start+capacity-1,newQueue);
else
{
 /* queue wrap around */
copy(queue+start, queue+capacity, newQueue);
copy(queue, queue+rear+1, newQueue+capacity-start); }
```

```
 /* switch to newQueue*/
front = 2*capacity – 1;
rear = capacity – 2;
capacity * =2;
free(queue);
queue= newQueue;
 }
```

2.4 Mutiple Stacks

- While implementing a stack using an array, we had seen that the size of the array must be known in advance. If the stack is allocated less space, then frequent OVERFLOW conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array.
- In case we allocate a large amount of space for the stack, it may result in sheer wastage of memory. Thus, there lies a trade-off between the frequency of overflows and the space allocated. So, a better solution to deal with this problem is to have multiple stacks or to have more than one stack in the same array of sufficient size. In Fig. 2.13(a) an array STACK[n] is used to represent two stacks, Stack A and Stack B. The value of n is such that the combined size of both the stacks will never exceed n.
- While operating on these stacks, it is important to note one thing—Stack A will grow from left to right, whereas Stack B will grow from right to left at the same time. Extending this concept to multiple stacks, a stack can also be used to represent n number of stacks in the same array. That is, if we have a STACK[n], then each stack I will be allocated an equal amount of space bounded by indices b[i] and e[i]. This is shown in Fig. 2.13(b)

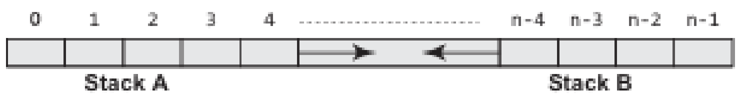


Fig. 2.13(a) Multiple Stack

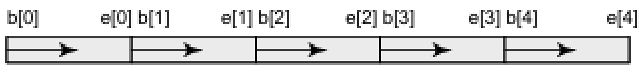
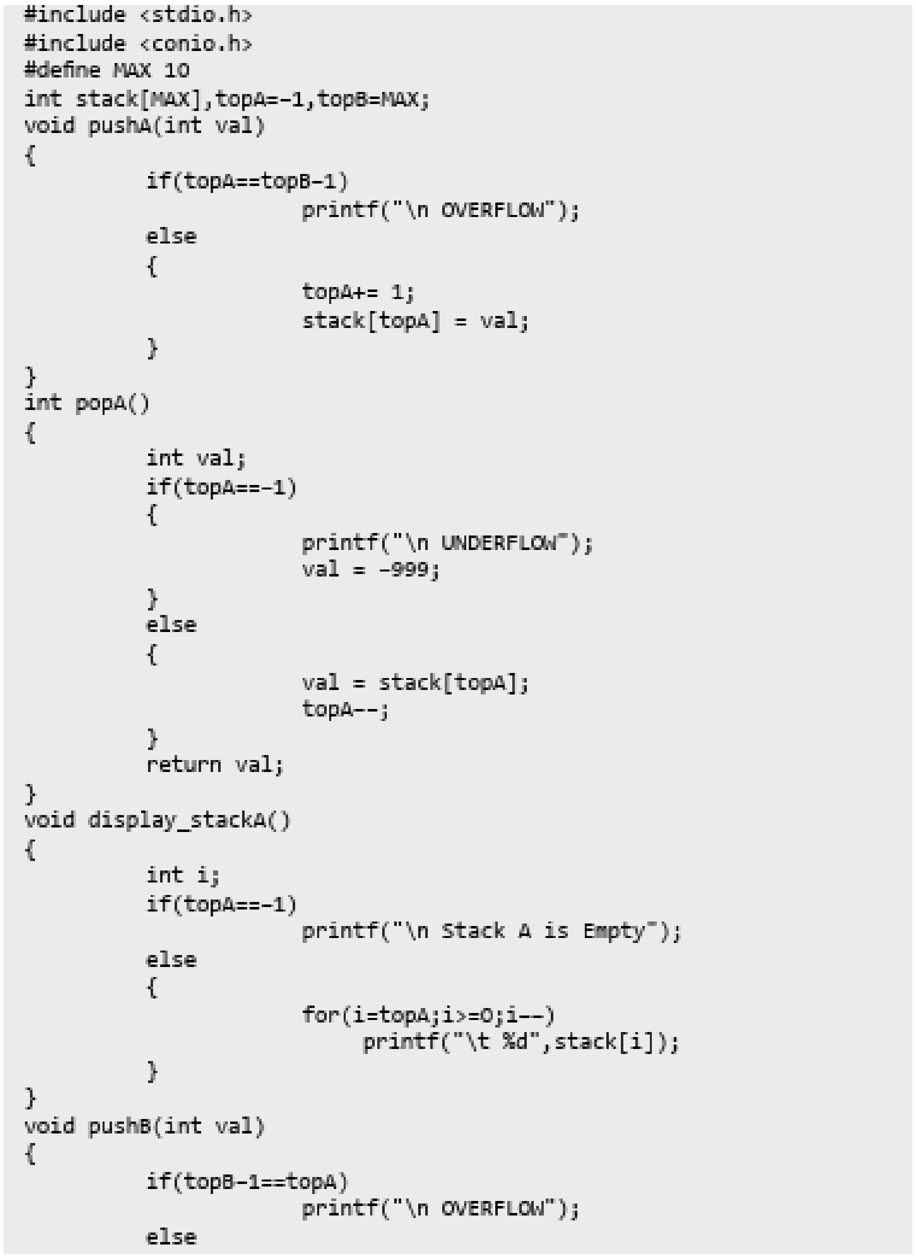


Fig. 2.13(b) Multiple Stack

Example:



```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int stack[MAX],topA=-1,topB=MAX;
void pushA(int val)
{
        if(topA==topB-1)
                printf("\n OVERFLOW");
        else
        {
                topA+= 1;
                stack[topA] = val;
        }
}
int popA()
{
        int val;
        if(topA==-1)
        {
                printf("\n UNDERFLOW");
                val = -999;
        }
        else
        {
                val = stack[topA];
                topA--;
        }
        return val;
}
void display_stackA()
{
        int i;
        if(topA==-1)
                printf("\n Stack A is Empty");
        else
        {
                for(i=topA;i>=0;i--)
                    printf("\t %d",stack[i]);
        }
}
void pushB(int val)
{
        if(topB-1==topA)
                printf("\n OVERFLOW");
        else
```

```
            {
                        topB -= 1;
                        stack[topB] = val;
            }
    }
    int popB()
    {
            int val;
            if(topB==MAX)
            {
                        printf("\n UNDERFLOW");
                        val = -999;
            }
            else
            {
                        val = stack[topB];
                        topB++;
            }
    }
    void display_stackB()
    {
            int i;
            if(topB==MAX)
                        printf("\n Stack B is Empty");
            else
            {
                        for(i=topB;i<MAX;i++)
                            printf("\t %d",stack[i]);
            }
    }
```

```
void main()
{
          int option, val;
          clrscr();
          do
          {
                      printf("\n *****MENU*****");
                      printf("\n 1. PUSH IN STACK A");
                      printf("\n 2. PUSH IN STACK B");
                      printf("\n 3. POP FROM STACK A");
                      printf("\n 4. POP FROM STACK B");
                      printf("\n 5. DISPLAY STACK A");
                      printf("\n 6. DISPLAY STACK B");
                      printf("\n 7. EXIT");
                      printf("\n Enter your choice");
                      scanf("%d",&option);
                      switch(option)
                      {
                          case 1: printf("\n Enter the value to push on Stack A : ");
                                  scanf("%d",&val);
                                  pushA(val);
                                  break;
                          case 2: printf("\n Enter the value to push on Stack B : ");
                                  scanf("%d",&val);
                                  pushB(val);
                                  break;
                          case 3: val=popA();
                                  if(val!=-999)
                                    printf("\n The value popped from Stack A = %d",val);
                                  break;
```

```
                        case 4: val=popB();
                                if(val!=-999)
                                  printf("\n The value popped from Stack B = %d",val);
                                break;
                        case 5: printf("\n The contents of Stack A are : \n");
                                display_stackA();
                                break;
                        case 6: printf("\n The contents of Stack B are : \n");
                                display_stackB();
                                break;
                    }
          }while(option!=7);
          getch();
   }
```

Output

```
*****MAIN MENU*****
1. PUSH IN STACK A
2. PUSH IN STACK B
3. POP FROM STACK A
4. POP FROM STACK B
5. DISPLAY STACK A
6. DISPLAY STACK B
7. EXIT
Enter your choice : 1
Enter the value to push on Stack A : 10
Enter the value to push on Stack A : 15
Enter your choice : 5
The content of Stack A are:
15      10
Enter your choice : 4
UNDERFLOW
Enter your choice : 7
```

2.6 Multiple Queues

- When we implement a queue using an array, the size of the array must be known in advance. If the queue is allocated less space, then frequent overflow conditions will be encountered. To deal with this problem, the code will have to be modified to reallocate more space for the array. In case we allocate a large amount of space for the queue, it will result in sheer wastage of the memory.
- Thus, there lies a tradeoff between the frequency of overflows and the space allocated. So a better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array of sufficient size. Figure 2.14(a) illustrates this concept. In the figure, an array QUEUE[n] is used to represent two queues, QUEUE A and QUEUE B.
- The value of n is such that the combined size of both the queues will never exceed n. While operating on these queues, it is important to note one thing: QUEUE A will grow from left to right, whereas QUEUE B will grow from right to left at the same time. Extending the concept to multiple queues, a queue can also be used to represent n number of queues in the same array. That is, if we have a QUEUE[n], then each QUEUE I will be allocated an

equal amount of space bounded by indices b[i] and e[i]. This is shown in Fig. 2.14(b)

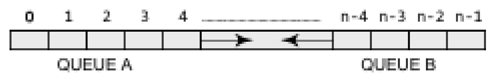


Fig. 2.14(a) Multiple Queue

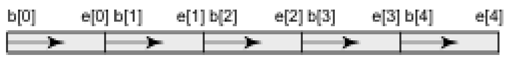


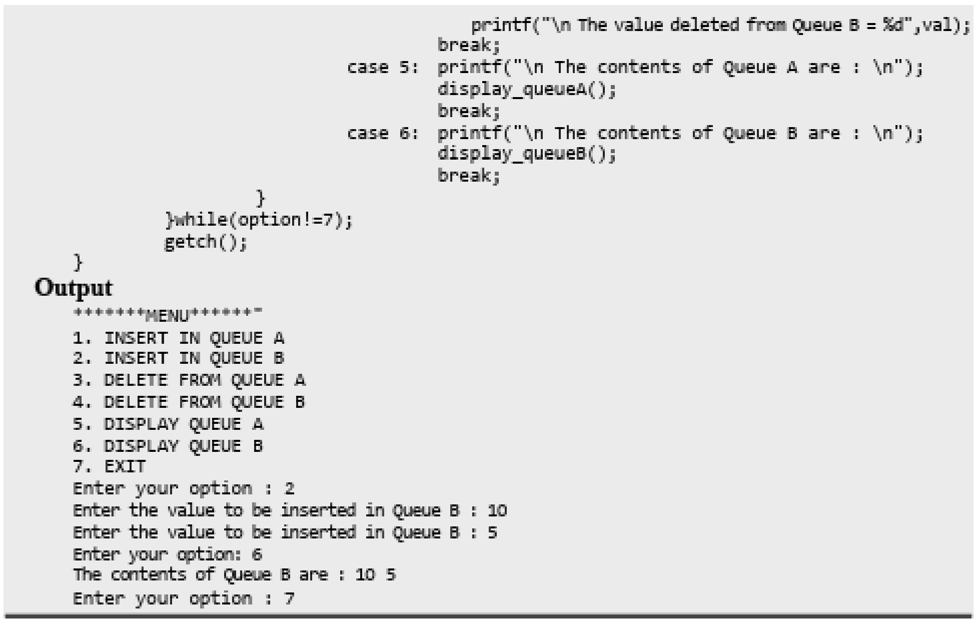
Fig. 2.14(b) Multiple Queue

Example:

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
int QUEUE[MAX], rearA=-1,frontA=-1, rearB=MAX, frontB
void insertA(int val)
{
        if(rearA==rearB -1)
                printf("\n OVERFLOW");
        else
        {
                if(rearA ==-1 && frontA == -1)
                {       rearA = frontA = 0;
                        QUEUE[rearA] = val;
                }
                else
                        QUEUE[++rearA] = val;
        }
}
int deleteA()
{
        int val;
        if(frontA==-1)
        {
                printf("\n UNDERFLOW");
                return -1;
        }
        else
        {
                val = QUEUE[frontA];
                frontA++;
                if (frontA>rearA)
                        frontA=rearA=-1
                return val;
        }
}

void display_queueA()
{
        int i;
        if(frontA==-1)
                printf("\n QUEUE A IS EMPTY");
        else
        {
                for(i=frontA;i<=rearA;i++)
                        printf("\t %d",QUEUE[i]);
        }
}
```

```
void insertB(int val)
{
        if(rearA==rearB-1)
                printf("\n OVERFLOW");
        else
        {
                if(rearB == MAX && frontB == MAX)
                {       rearB = frontB = MAX-1;
                        QUEUE[rearB] = val;
                }
                else
                        QUEUE[--rearB] = val;
        }
}

int deleteB()
{
        int val;
        if(frontB==MAX)
        {
                printf("\n UNDERFLOW");
                return -1;
        }

        else
        {
                val = QUEUE[frontB];
                frontB--;
                if (frontB<rearB)
                        frontB=rearB=MAX;
                return val;
        }
}

void display_queueB()
{
        int i;
        if(frontB==MAX)
                printf("\n QUEUE B IS EMPTY");
        else
        {
                for(i=frontB;i>=rearB;i--)
                        printf("\t %d",QUEUE[i]);
        }
}
```

```
int main()
{
        int option, val;
        clrscr();
        do
        {
                printf("\n *******MENU******");
                printf("\n 1. INSERT IN QUEUE A");
                printf("\n 2. INSERT IN QUEUE B");
                printf("\n 3. DELETE FROM QUEUE A");
                printf("\n 4. DELETE FROM QUEUE B");
                printf("\n 5. DISPLAY QUEUE A");
                printf("\n 6. DISPLAY QUEUE B");
                printf("\n 7. EXIT");
                printf("\n Enter your option : ");
                scanf("%d",&option);
                switch(option)
                {
                        case 1:  printf("\n Enter the value to be inserted in Queue A : ");
                                 scanf("%d",&val);
                                 insertA(val);
                                 break;
                        case 2:  printf("\n Enter the value to be inserted in Queue B : ");
                                 scanf("%d",&val);
                                 insertB(val);
                                 break;
                        case 3:  val=deleteA();
                                 if(val!=-1)
                                    printf("\n The value deleted from Queue A = %d",val);
                                 break;
                        case 4 : val=deleteB();
                                 if(val!=-1)
```



```
                                printf("\n The value deleted from Queue B = %d",val);
                              break;
                    case 5:   printf("\n The contents of Queue A are : \n");
                              display_queueA();
                              break;
                    case 6:   printf("\n The contents of Queue B are : \n");
                              display_queueB();
                              break;
                }
        }while(option!=7);
        getch();
}
```

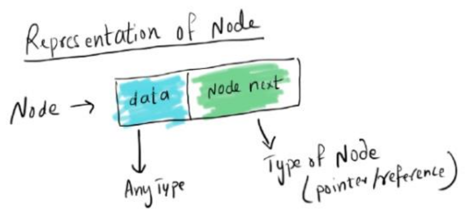
Output

```
*******MENU******"
1. INSERT IN QUEUE A
2. INSERT IN QUEUE B
3. DELETE FROM QUEUE A
4. DELETE FROM QUEUE B
5. DISPLAY QUEUE A
6. DISPLAY QUEUE B
7. EXIT
Enter your option : 2
Enter the value to be inserted in Queue B : 10
Enter the value to be inserted in Queue B : 5
Enter your option: 6
The contents of Queue B are : 10 5
Enter your option : 7
```

2.7 Linked 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. A linked list does not store its elements in consecutive memory locations and the user can add any number of elements to it.

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.There are 2 fields in linked list

1. Data
2. Link to next node

So we can write the structure of linked list as in Fig.2.15.

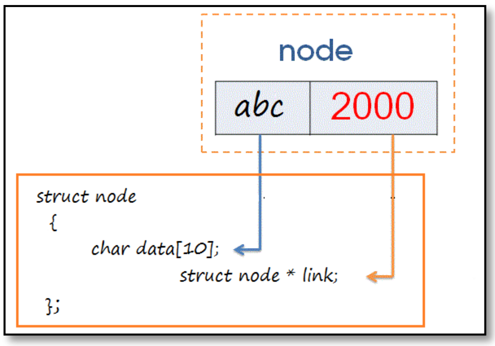


Fig. 2.15: Linked List Structure

2.5.1 Linked list Definition

A linked list (Fig.2.17), or one-way list, is a linear collection of data elements, called nodes, where the linear order is given by means of pointers. That is, each node is divided into two parts:

- The first part contains the information of the element, and
- The second part, called the ***link field* or *nextpointer*** field, contains the address of the next node in the list.

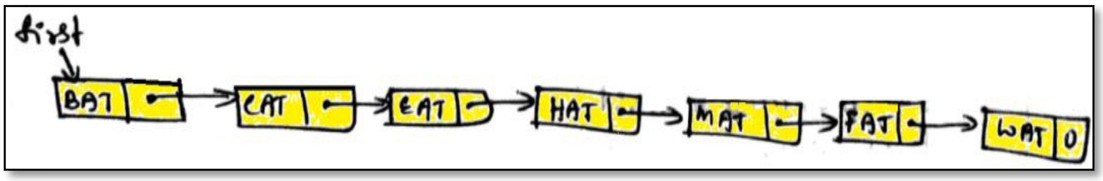


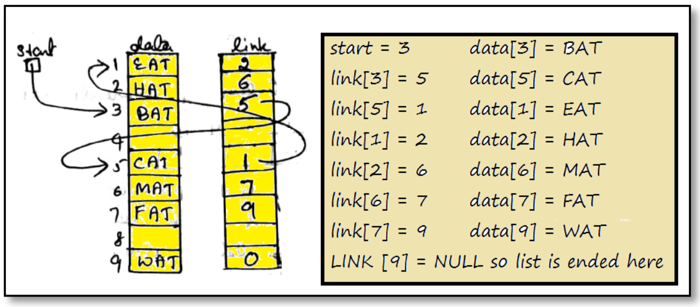
Fig2.17: representation of linked list

2.5.2 Representation of linked lists in Memory

Let LIST be a linked list. Then LIST will be maintained in memory as follows.

1. LIST requires two linear arrays such as INFO and LINK-such that INFO[K] and LINK[K] contains the information part and the next pointer field of a node of LIST.
2. LIST also requires a variable name such as START which contains the location of the beginning of the list, and a next pointer sentinel denoted by NULL-which indicates the end of the list.
3. The subscripts of the arrays INFO and LINK will be positive, so choose NULL = 0,unless otherwise stated.

The Fig.2.17 can be represented as Fig. 2.18 which indicates that the nodes of a list need not occupy adjacent elements in the arrays INFO and LINK, and that more than one list may be maintained in the same linear arrays INFO and LINK. However, each list must have its own pointer variable giving the location of its first node.



start = 3	data[3] = BAT
link[3] = 5	data[5] = CAT
link[5] = 1	data[1] = EAT
link[1] = 2	data[2] = HAT
link[2] = 6	data[6] = MAT
link[6] = 7	data[7] = FAT
link[7] = 9	data[9] = WAT

LINK [9] = NULL so list is ended here

Fig. 2.18: Linked List Representation in Memory.

2.6 Linked List Operations(Representing Chains in C)

1. Create

Initially we make ‘first’ as ‘NULL’ . When we create a new node, name it as ‘temp’ and store the value in its data field. For example, enter 10 to linked list as in Fig.1. Now connect the new node to ‘first’ as in Fig1. Now make ‘temp’ as ‘first’ so that we can connect next new node to ‘first’ directly as in Fig. We can create ‘n’ number of nodes in the same way.

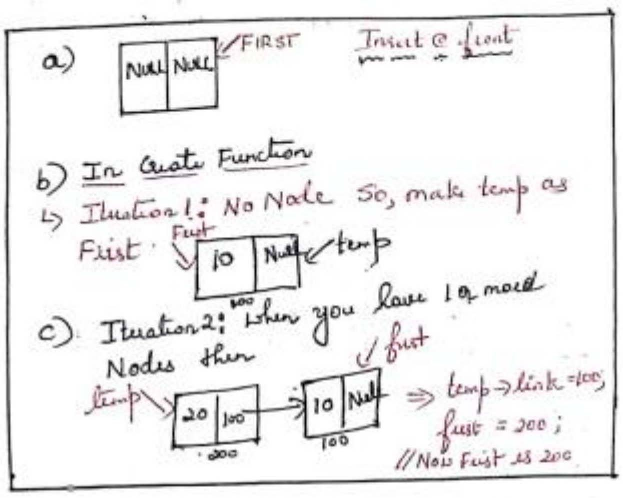


Fig. 1: Create a node with insertion front.

Program code for create operation

```
struct node
{
int data;
struct node *link;
};
struct node*first=NULL,*temp,*last;

void create()
{
        printf("\n enter the no of elements to be inserted into the list\n");
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                temp = (struct node *) malloc(sizeof (struct node));
                printf("Enter the data to be inserted:\n");
                scanf("%d",temp->data);
                temp->link=NULL;
                if(first==NULL)
                first=temp;
                else
                {
                temp->link=first;
                first=temp;
                }
        }
}
```

2.Insert to Front

It works same as create function, by inserting new node to front. Here only one node can be inserted at a time.

Program code for insert front operation

```
void insert_front()
{

        temp = (struct node *) malloc(sizeof (struct node));
        printf("Enter the data to be inserted:\n");
        scanf("%d",temp->data);
        temp->link=NULL;
        }
        if(first==NULL)
        first=temp;
        else
        {
        temp->link=FIRST;
        FIRST=temp;
        }
```

3.Insert to End

Suppose we have empty list, then first is equal to NULL. Then directly create new node and make it as first. But suppose we have list as in Fig.1. Now if we want to perform insert end, then new node to be attached to right side of the last node (right of 10 here). Create a new node and name it as temp and read new data(30) to temp as in Fig.3. Check if last→link=NULL and if it is not equal to NULL make last=last→link and last→link=NULL as in Fig.3 and then connect new node 'temp' to 'last→link' as in Fig.3.

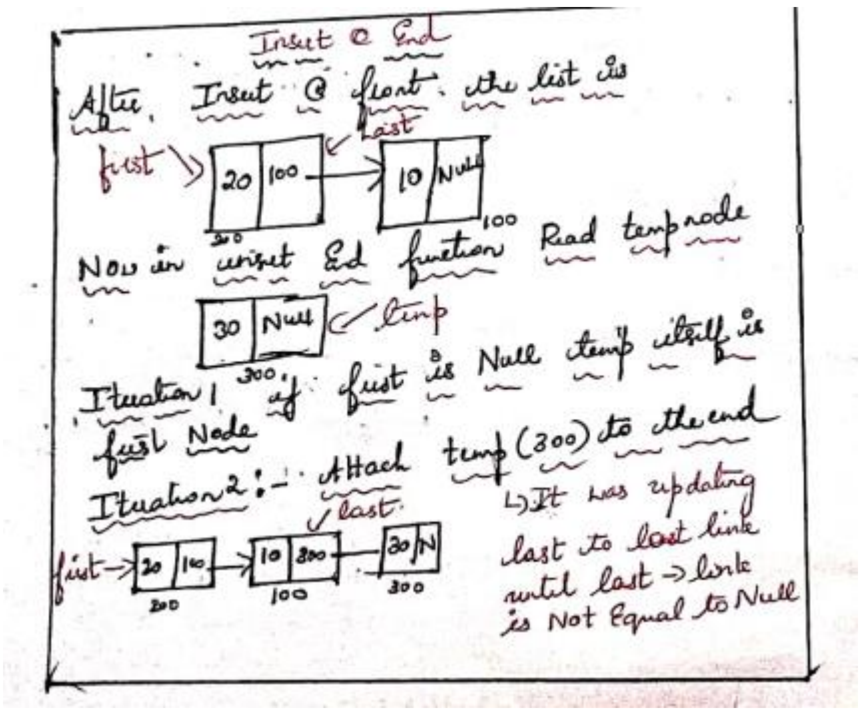

Insert @ End
After Insert @ front, the list is
first
20
100
200
last
10
NULL
100
Now in insert End function Read temp node
30
NULL
300
temp
Iteration 1: if first is Null temp itself is first Node
Iteration 2: Attach temp (300) to the end
It was updating last to last link until last -> link is Not Equal to Null
first
20
100
200
10
300
100
last
30
N
300

Fig. 3: Insert a New Node at the End of a Linked List

Program code for insert end operation

```
void insert_end()
{
        last=first;
        temp = (struct node *) malloc(sizeof (struct node));
        printf("Enter the data to be inserted:\n");
        scanf("%d",temp->data);
        temp->link=NULL;
        if(first==NULL)
        first=temp;
        else
        {
        while(last->link!=NULL)
        {
        last=last->link;
        }
        last->link=temp;
        }
```

}

4.Delete from Front

Suppose if we want to delete a node from front from the Fig.3, (delete node contains data 20), then make 'temp' as 'first' and delete temp→data and make 'temp→link' as 'first'. If first=NULL, then that means there is no element in the list.

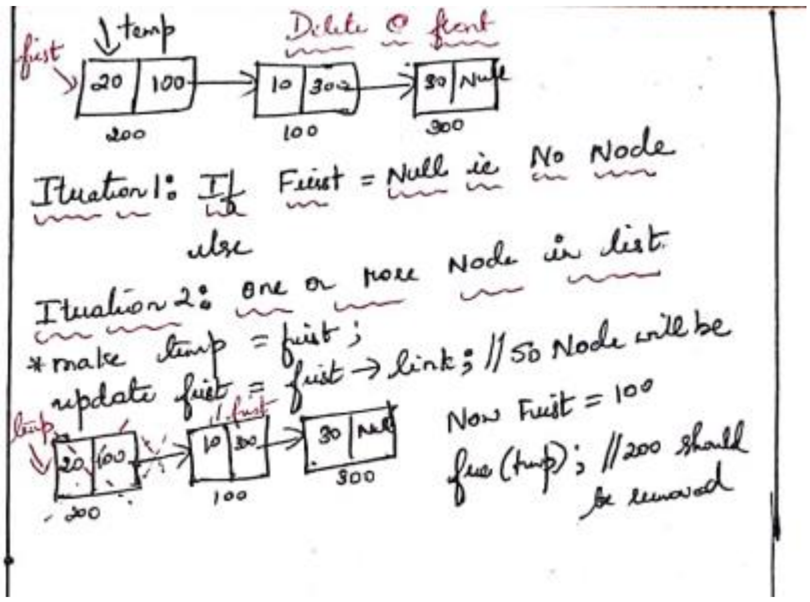

Delete @ front
first
temp
20 | 100
10 | 300
30 | Null
200
100
300
Iteration 1: If First = Null ie No Node
else
Iteration 2: one or more Node in list
* make temp = first;
update first = first → link; // so Node will be
Now First = 100
free (temp); // 200 shall be removed

Fig. 4: Example Linked list for performing Delete Front

Program code for delete front operation

```
void delete_front()
{
        temp = first;
        if(first == NULL)
        printf("\n list is empty");
else
{
printf("deleted element is %d",temp->data);
first=first->link;
free(temp);
}
}
```

5.Delete from End

Suppose if we want to delete a node from the end from the Fig.5 i.e. delete node contains data 30. If first=NULL, that means there is no element in the list. Otherwise make ‘temp’ as ‘first’ as in Fig.5. Then check if temp→link= NULL, if it is not equal to NULL, then make next node as temp and Once againcheck if temp→link=NULL, if not, repeat the same previous operation until temp→link=NULL as in Fig.5. When temp→link=NULL, delete temp→data and make last→link to point to NULL as in Fig.5.

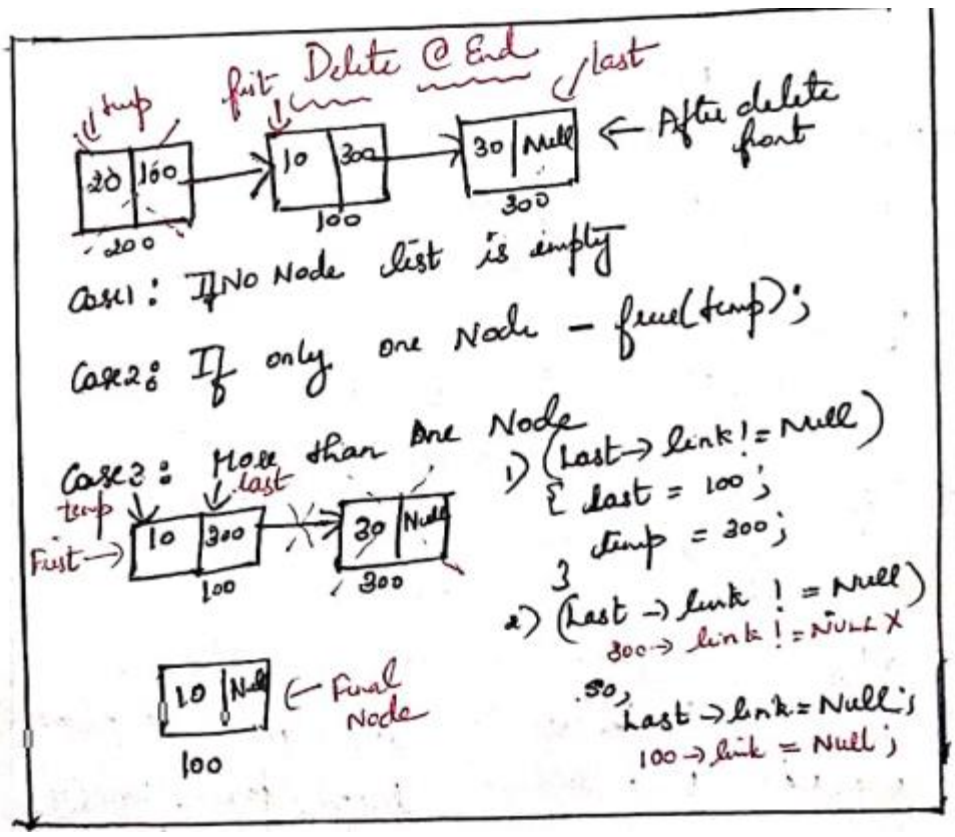


Fig. 5: Delete a Node from the End of a Linked List

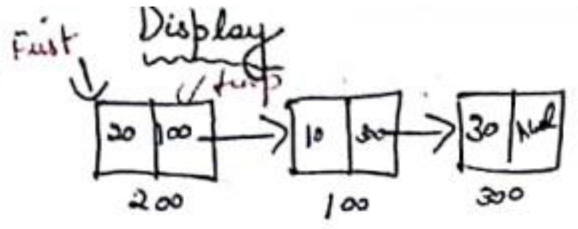
Program code for delete end operation

```
void delete_end()
{
last=NULL;
 temp=first;
 if(first==NULL) // Check for empty list
    printf("List is empty\n");
 else if(first->link==NULL) // // Check for single node in SLL
       {
        printf("Deleted element is %d\n",temp->data);
```

```
                 free(first);
                 first=NULL;
                }
               else
                 {
                    while(temp->link!=NULL)
                    {
                     last=temp;
                     temp=temp->link;
                    }
                   last->link=NULL;
                   printf("Deleted element is %d\n",temp->data);
                   free(temp); // delete last node
                   }
                return;
}
```

6.Traverse (Display) the linked list

Consider Fig. 6. Suppose if ‘first’ is equal to NULL, then no need of traversing. Directly display that list is empty. Otherwise we need to traverse the whole list by traversing node by node. For that make ‘first’ as ‘temp’ and start traversing from first and display the traversed node data i.e. temp→data. Then make next node as temp for traversing next node and so on until we reach NULL. If we want to count the number of nodes, then put the counter. Initially set the count value to one and increment the count value after traversing each node.



1) If First == Null
No Node
else
2) temp -> link (100) != Null (T)
Count -1 ;
print 20 ; update temp to temp -> link;
3) Now temp is 100 ;
check temp -> link (300) != Null (T)
Count = 2; print 10; update temp = 300
4) temp -> link (NULL) ! = Null (F) Print 30

Fig. 6: Display of a Linked List

Program code for traverse operation

```
void display_count()
{
 int count=1;
 temp=first;
 printf("Student details:\n");
 if(first==NULL) // check for empty list
   printf("Student detail is NULL and count is 0\n");
 else
   {
     printf("\nUSN\tNAME\tBRANCH\tPHNO\tSEMESTER\n");
     while(temp->link!=NULL) // iterate nodes of SLL until end node
     {
       count++;
       printf( "\n %d”,temp->data);
       temp=temp->link; // next node
     }
       printf( "\n %d”,temp->data);
 printf("\n  node count is %d\n",count);
    }
  return;
}
```

2.7 Linked Stacks

- 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. The linked representation of a stack is shown in Fig.3.9.1

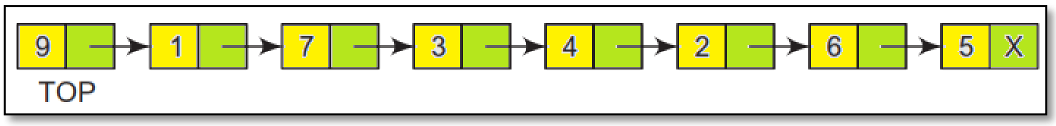


Fig. 3.9.1: Linked Stack.

- The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in Fig. 3.9.2(a). To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP! =NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP. Thus, the updated stack becomes as shown in Fig. 3.9.2(b).

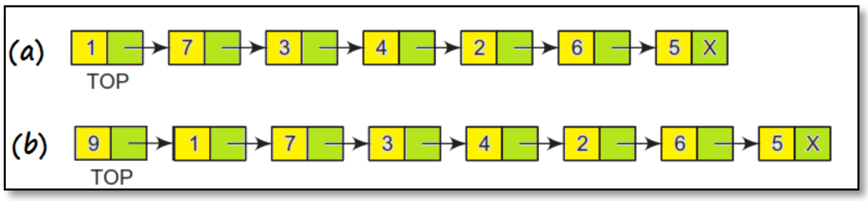


Fig. 3.9.2: Linked Stack Push Operation

- Figure 3.9.3 shows the algorithm to push an element into a linked stack. 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 valuates to true, then NULL is stored in the NEXT part of the node and the new node is called TOP. However, if the newnode 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.

```
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
```

Fig. 3.9.3: Algorithm to Insert an Element in a Linked Stack

- The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack shown in Fig. 3.9.4 (a). In case TOP! =NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown in Fig. 3.9.4 (b).

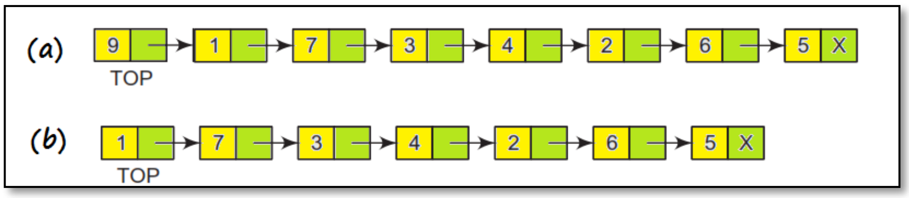


Fig. 3.9.4: Linked Stack Pop Operation.

- Figure 3.9.5 shows the algorithm to delete an element from a stack. 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.

```
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
```

Fig. 3.9.5: Algorithm to Delete an Element in a Linked Stack

SLL STACK PROGRAM

```
#include<stdio.h>
#include<stdlib.h>
 struct Node
    {
    int data;
    struct Node *next;
    };

  struct Node *top =NULL;
void push(int);
void pop();
void display();
void main()
{
    int choice,value;
    printf("\n Stack using Linked List\n");
    while(1)
    {
    printf("\n****** MENU ******\n");
    printf("1. Push\n2. Pop\n3.Display \n4.Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
    switch(choice)
    {
    case 1: printf("Enter the value to be insert: ");
            scanf("%d", &value);
            push(value);
            break;
    case 2: pop();
            break;
    case 3: display();
            break;
    case 4: exit(0);
            break;
```

```
   default: printf("\nWrong selection!!! Please try again!!!\n");
   }
   }
}

void push(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(top == NULL)
   newNode->next = NULL;
   else
   newNode->next = top;
   top = newNode;
}

void pop()
{
   if(top == NULL)
   printf("\nStack is overflow!!!\n");
   else
   {
      struct Node *temp = top;
      printf("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   }
   }

   void display()
   {
   if(top == NULL)
   printf("\nStack is Empty!!!\n");
   else
```

```
{
struct Node *temp = top;
while(temp->next != NULL)
{
printf("%d--->",temp->data);
temp = temp -> next;
}
printf("%d--->NULL",temp->data);
}
}
```

2.8 Linked Queue

- We have seen how a queue is created using an array. Although this technique of creating a queue is easy, its drawback is that the array must be declared to have some fixed size.
- If we allocate space for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will be wasted. And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time.
- In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative ,i.e. the linked representation is used.
- In a linked queue, every element has two parts, one that stores the data and another which holds 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 front end. If FRONT=REAR=NULL, then it indicates that the queue is empty. The linked representation of queue is shown in Fig. 3.10.1.

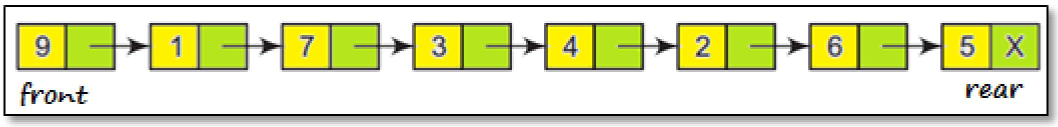


Fig. 3.10.1: Linked Stack.

- The insert operation is used to insert an element into the queue. The new element is added as the last element of the queue. Consider the linked queue shown in Fig. 3.10.2 (a). To insert an element with value 9, we first check if FRONT=NULL. If this is the case, then we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called FRONT and REAR. However, if FRONT! =NULL, then we insert the new node at the rear end of the linked queue and name this new node as REAR. Thus, the updated stack becomes as shown in Fig. 3.10.2(b).

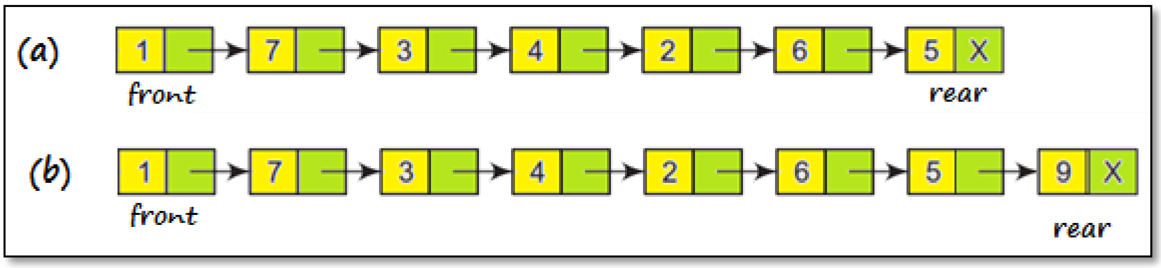
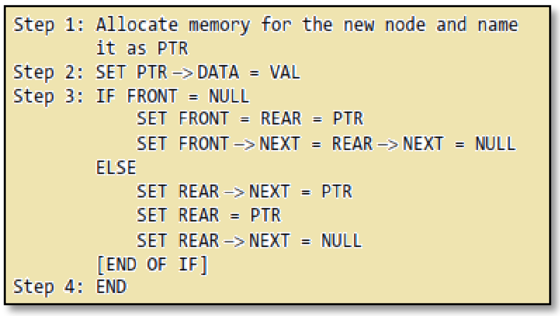


Fig. 3.10.2: Linked Queue Insert Operation.

- Figure 3.10.3 shows the algorithm to insert an element into a linked queue. 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 queue. This is done by checking if FRONT = NULL. In case the new node is tagged as FRONT and REAR. Also NULL is stored in the NEXT part of the 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.



```
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
```

Fig. 3.10.3: Algorithm to Insert an Element in a Linked Queue.

Delete Operation

- The delete operation is used to delete the element that is first inserted

- The delete operation is used to delete the element that is first inserted into the queue. i.e. the element whose address is stored at FRONT.
- However, before deleting the value, we must first check if FRONT=NULL, because if this is the case, then it means that the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed.
- Consider the stack shown in Fig. 3.10.4(a). To delete an element, we first check if FRONT=NULL. If it is false, then we delete the 1st node pointed by the FRONT. The FRONT will now point to the 2nd element of the linked queue. Thus the updated queue becomes as shown in Fig. 3.10.4(b).

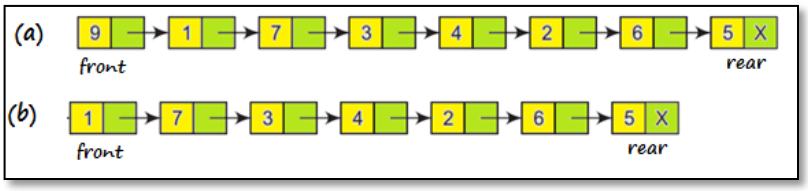
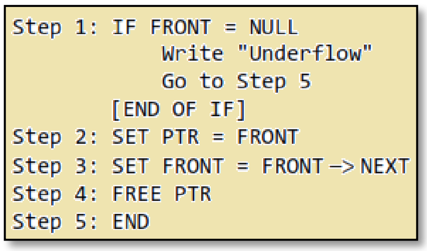


Fig. 3.10.4: Linked Queue Delete Operation.

- Figure 3.10.5 shows the algorithm to delete an element froma queue. In Step 1, we first check for the UNDERFLOW condition. 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.



```
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
```

Fig. 3.10.5: Algorithm to Delete an Element in a Linked Queue.

- **Example: Create a SLL queue of N Students Data.**

```
#include<stdio.h>
#include<stdlib.h>
struct Node
{

   char usn[20],name[10],branch[5];
   unsigned long long int phno;
   int sem;
   struct Node *next;
};
typedef struct Node * NODE;
NODE temp,front = NULL,rear = NULL;
void insert();
void delete();
void display();
void main()
{
  int choice, value;

  printf("\n Queue Implementation using Linked List \n");
  while(1){
    printf("\n****** MENU ******\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
    printf("Enter your choice: ");
    scanf("%d",&choice);
```

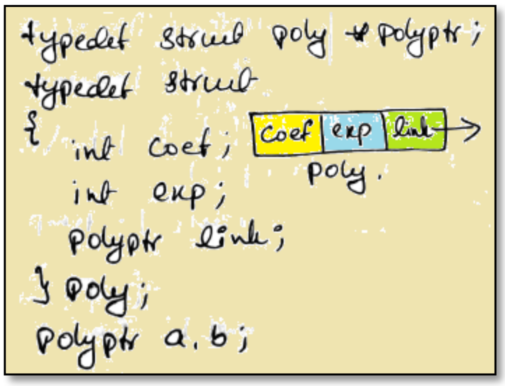
```
      switch(choice){
             case 1:insert();
                         break;
             case 2: delete();
                        break;
             case 3: display();
                         break;
             case 4: exit(0);
             default: printf("\nWrong selection!!! Please try again!!!\n");
      }
   }
}

void insert()
{
NODE newNode;
newNode=(NODE)malloc(sizeof(struct Node));
printf("Enter USN: ");
 scanf("%s",newNode->usn);
 printf("Enter NAME: ");
 scanf("%s",newNode->name);
 printf("Enter Branch: ");
 scanf("%s",newNode->branch);
 printf("Enter phone Number: ");
 scanf("%llu",&newNode->phno);
 printf("Enter Semester: ");
 scanf("%d",&newNode->sem);
newNode->next=NULL;
if(front == NULL)
    {

     front = rear = newNode;
      front->next=NULL;
      rear->next=NULL;
 }
```

```
  else{
    rear -> next = newNode;
    rear = newNode;
    rear->next=NULL;
  }
  printf("\nInsertion is Success!!!\n");
}

void delete()
{

  if(front == NULL)
    printf("\nQueue is Underflow!!!\n");
  else{
      temp = front;
  front = front -> next;
printf("\nDeleted node is with usn: %s", temp->usn);

    free(temp);
  }
}

void display()
{
  if(front == NULL)
    printf("\nQueue is Empty!!!\n");
  else{
    temp = front;
    while(temp->next != NULL){
printf("The Student information in the node is\n");
printf("\nUSN:%s\nNAME:%s\nBRANCH:%s\nPHONE
NO.:%llu\nSEM:%d\n",temp->usn,temp->name,temp->branch,temp-
>phno,temp->sem);

         temp = temp -> next;
```

```
        }
        printf("\nUSN:%s\nNAME:%s\nBRANCH:%s\nPHONE
NO.:%llu\nSEM:%d\n",temp->usn,temp->name,temp->branch,temp-
>phno,temp->sem);
    }
}
```

2.9 Applications of Linked lists: Polynomials

Linked lists can be used to represent polynomials and the different operations that can be performed on them. A polynomial is a combination of coefficients and exponents. We represent a polynomial, each term as a node containing coefficients and exponent field, as well as a pointer to next term. The type declarations are shown in Fig. 3.11.



```
typedef struct poly *polyptr;
typedef struct
{
    int coef;
    int exp;
    polyptr link;
} poly;
polyptr a,b;
```

Coef
exp
link
poly.

2.9.1 Polynomial representation

Let us see how a polynomial is represented in the memory using a linked list. Consider a polynomial $6x^3 + 9x^2 + 7x + 1$. Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively. Every term of a polynomial can be represented as a node of the linked list. Figure 3.11.1 showsthe linked representation of the terms of the above polynomial.

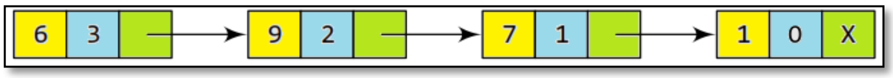


Fig. 3.11.1: Linked Representation of a Polynomial

2.9.2 Polynomial Addition

Consider an example polynomial shown in Fig. 3.11.2. To add polynomial, we examine their terms starting at the nodes pointed by a and b. There are three cases:

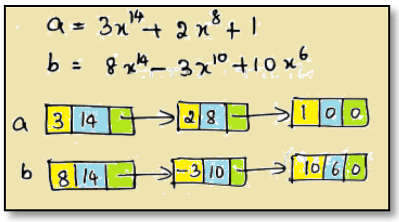


Fig. 3.11.2: Example of a Polynomial

Case 1: If the exponent of a and b are equal

Consider the Fig. 3.11.2.1(a). Here a→exp = b→exp. So add the coefficients of a and b and store the result in the new link called 'res' as in Fig. 3.11.2.1 (b).

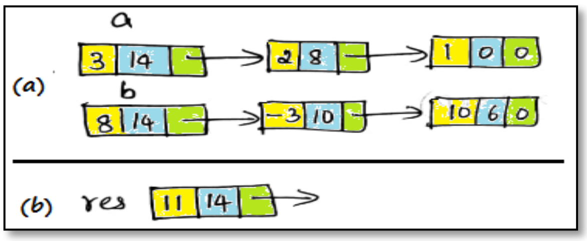


Fig. 3.11.2.1: Polynomial Addition if a→exp = b→exp.

Case 2: If the exponent of a is less than exponent of b

Consider the Fig3.11.2.2 (a). Here a→exp < b→exp. So copy the coefficient and exponent of the bigger term i.e. 'b' on to the new link called 'res' as in Fig. 3.11.2.2 (b).

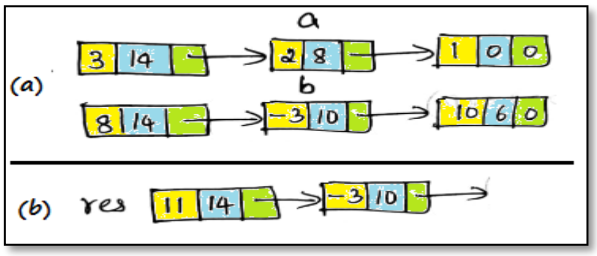


Fig. 3.11.2.2: Polynomial Addition if a→exp < b→exp.

Case 3: If the exponent of a is greater than exponent of b

Consider the Fig. 3.11.2.3(a). Here a→exp > b→exp. So copy the coefficient and exponent of the bigger term i.e. 'a' on to the new link called 'res' as in Fig. 3.11.2.3 (b).Final polynomial is shown in the Fig. 3.11.2.4.

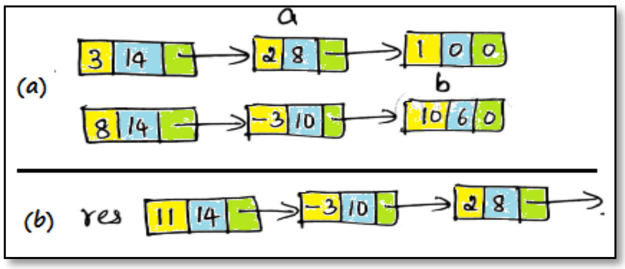


Fig. 3.11.2.3: Polynomial Addition if a→exp > b→exp

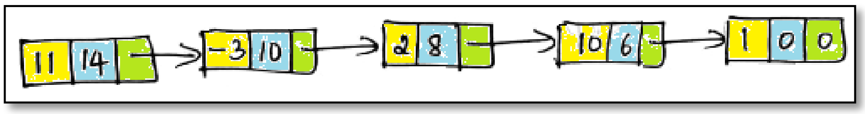


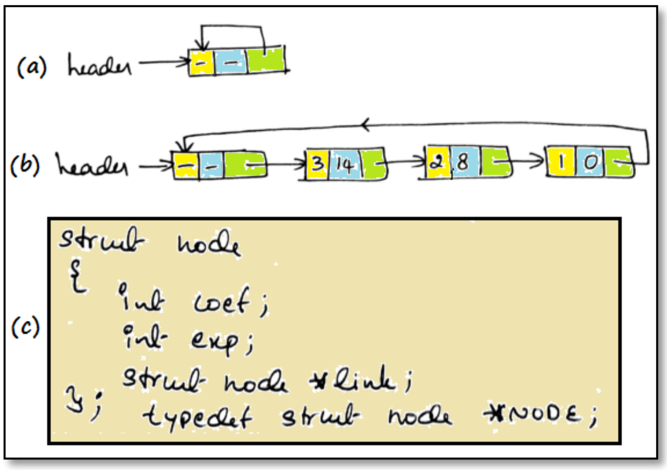
Fig. 3.11.2.4: Resultant Polynomial after Polynomial Addition.

• Program Code for polynomial addition

```
polyptr addpoly (polyptr a, polyptr b)
{
polyptr c, *temp;
while( a->link && b->link)
{
        if( a->exp>b->exp)
        {
                c->exp=a->exp;
        c->coef=a->coef;
        a=a->link;
}
else if( a->exp<b->exp)
{
        c->exp=b->exp;
        c->coef=b->coef;
        b=b->link;
}
else
{
        c->exp=a->exp;
        c->coef=a->coef + b->coef;
        a=a->link;
        b=b->link;
}
c->link=(struct   poly*)malloc(sizeof(struct
poly));
c=c->link;
c->link = NULL;
while (a->link || b->link)
{
if(a->link)
{
        c->exp=a->exp;
        c->coef=a->coef;
        a=a->link;
}
if(b->link)
{
        c->exp=b->exp;
        c->coef=b->coef;
        b=b->link;
}
c->link= (struct poly*)malloc(sizeof(struct
poly));
c=c->link;
c->link = NULL;
}
```

}

2.9.3 Circular list representation of Polynomial



```
struct node
{
    int coef;
    int exp;
    struct node *link;
};
    typedef struct node *NODE;
```

Fig. 3.12 : Circular list Representation of Polynomial

The zero polynomial is represented as in Fig. 3.12 (a);while a(x) = $3x^{14}+2x^{8}+1$ can be represented as in Fig 3.12 (b). here link field of last node points to the 1st node in the list. So we call this a circular linked list. To simplify the addition algorithm for polynomial represented as circular linked list, we set coef and exp field of header node to -1. The structure can be written as in Fig. 3.12 (c).

Program Example for polynomial

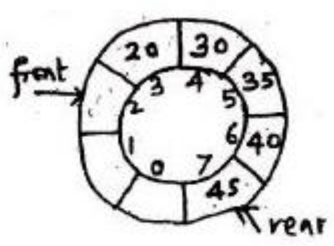
```
#include<stdio.h>
#include<stdlib.h>
#include<math.h>
 struct node // polynomial node
{
        int coef;
        int x,y,z;
        struct node  *link;
};
typedef struct  node *NODE;

NODE getnode() // create a node
{
        NODE x;
        x=(NODE)malloc(sizeof(struct node));
        return x;
} // end of getnode

NODE readpoly()
{
        NODE temp,head,cur;
        char ch;
        head=getnode(); // create a head node and set all values to -1 it is similar to FIRST in
SLL program
        head->coef=-1;
        head->x=-1;
        head->y=-1;
        head->z=-1;
        head->link=head; // self reference
        do
        {
                temp=getnode(); // create a polynomial node
                printf("\nEnter the coefficient and exponent in decreasing order\n");
                scanf("%d%d%d%d",&temp->coef,&temp->x,&temp->y,&temp->z );
                cur=head;
                while(cur->link!=head) // find the last node
                                cur=cur->link;
                cur->link=temp; // connect new node to the last node
                temp->link=head; // point back to head
                printf("\nDo you want to enter more coefficients(y/n)");
                fflush(stdin); // to clear the stdin buffer
                scanf("%c",&ch);
        } while(ch =='y' || ch == 'Y');
        return  head; // return the polynomial list
} // end of readpoly

int compare(NODE a,NODE b) // function to compare the A and B polynomial nodes
{
        if(a->x > b->x)
                return 1;
        else if(a->x < b->x)
                return -1;
        else if(a->y > b->y)
                return 1;
        else if(a->y < b->y)
                return -1;
        else if(a->z > b->z)
                return 1;
        else if(a->z < b->z)
                return -1;
                return 0;
} // end of compare

void attach(int cf,int x1,int y1, int z1, NODE *ptr) // function to attach the A and B
polynomial node to C Polynomial
{
      NODE temp;
      temp=getnode();
      temp->coef=cf;
      temp->x=x1;
      temp->y=y1;
      temp->z=z1;
      (*ptr)->link=temp;
      *ptr=temp;
} // end of attach

NODE addpoly(NODE a,NODE b) // function to add polynomial A and B i.e, C=A+B
{
        NODE  starta,c ,lastc;
        int sum,done=0;
        starta=a;
        a=a->link;
        b=b->link;
        c=getnode(); // create list C to store A+B
        c->coef=-1;
        c->x=-1;
        c->y=-1;
        c->z=-1;
        lastc=c;
        do{
        switch(compare(a,b))
        {
          case -1:attach(b->coef,b->x,b->y,b->z,&lastc);
            b=b->link;
            break;
          case 0:if(starta==a) done=1;
            else{
              sum=a->coef+b->coef;
            if(sum)
              attach(sum,a->x, a->y,a->z,&lastc);
              a=a->link;b=b->link;
                }
            break;
          case 1:  if(starta==a) done=1;
            attach(a->coef,a->x, a->y,a->z,&lastc);
            a=a->link;
            break;
        }
        }while(!done); // repeate until not done
        lastc->link=c; // point back to head of C
        return c; // return answer
    }

void print(NODE ptr) // to print the polynomial
{
        NODE cur;
        cur=ptr->link;
        while(cur!=ptr) // To print from HEAD node till END node
        {
                        printf("%d*x^%d*y^%d*z^%d",cur->coef,cur->x, cur->y, cur->z);
                        cur=cur->link; // move to next node
                        if (cur!=ptr)
                        printf(" + ");
                }
   } // end of print

 void evaluate(NODE ptr) // function to evaluate the final polynomial
 {
        int res=0;
        int x,y,z, ex,ey,ez,cof;
        NODE cur;
        printf("\nEnter the values of x, y,z"); // read values of X, Y and Z
        scanf("%d", &x);
        scanf("%d", &y);
        scanf("%d", &z);
   cur=ptr->link; // start with HEAD
   while(cur!=ptr) // Repeat until the end of list
        {
                        ex=cur->x; // exponent of x
                        ey=cur->y; // exponent of y
                        ez=cur->z; // exponent of z
                        cof=cur->coef; // coefficient
                        res+=cof*pow(x,ex)*pow(y,ey)*pow(z,ez); // compute result for each
polynomial
                        cur=cur->link; // move to next node
                }
        printf("\nresult: %d",res);
} // end of evaluate

void main(void)
```

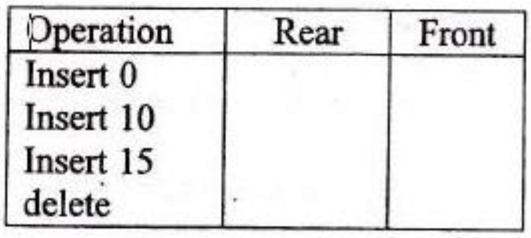
```
{
        int i, ch;
        NODE a=NULL,b,c;
        while(1)
   {
                printf("\n1: Represent first polynomial A");
                printf("\n2: Represent Second polynomial B");
                printf("\n3: Display the polynomial A");
                printf("\n4: Display the polynomial B");
                printf("\n5: Add A & B polynomials"); // C=A+B
                printf("\n6: Evaluate polynomial C");
                printf("\n7: Exit");
                printf("\n Enter your choice: ");
                scanf("%d",&ch);
                switch(ch)
                {
                        case 1: printf("\nEnter the elements of the polynomial A");
                                        a=readpoly();
                                        break;
                        case 2:printf("\nEnter the elements of the polynomial B");
                                        b= readpoly();
                                        break;
                        case 3: print(a); // display polynomial A
                                break;
                        case 4:print(b); // display polynomial A
                                break;
        case 5: c=addpoly(a,b); // C=A+B
                                        printf("\nThe sum of two polynomials is: ");
                                        print(c); // display polynomial C
                                        printf("\n");
                                        break;
                        case 6:evaluate(c); // Evaluate polynomial C
             break;
                        case 7: return;
```

```
                    default: printf("\nInvalid choice!\n");
                } //end of switch
            } // end of while
} // end of main
```

Question Bank

1. Give the disadvantage of ordinary queue and how it can be solved using circular queue. Explain with suitable example, how would you implement circular queue using dynamically allocated array
2. Write insert and delete functions of circular queue.
3. Explain multiple stacks and queues with program.
4. Define Queue. Implement the operations of queue using arrays.
5. What is circular queue? Explain how it is differ from linear queue. Write a C program for primitive operations of circular queue
6. For the given circular queue shown in Fig. write the values of front and rear in the table after each specified operation is performed. Queue full empty conditions must be considered. 0 - 7 indicate the array indices.





Operation	Rear	Front
Insert 0		
Insert 10		
Insert 15		
delete		

7. What is linked list? Explain the different types of linked list with examples.

8.Give a node structure to create a linked list of integers and write a C function to perform the following.

a. Create a three-node list with data 10, 20 and 30
b. Inert a node with data value 15 in between the nodes having data values 10 and 20
c. Delete the node which is followed by a node whose data value is 20
d. Display the resulting singly linked list.

9. With node structure show how would you store the polynomials in linked lists? Write C function for adding two polynomials represented as circular lists.

10. Write a C function to add two-polynomials represented as circular list with header node.

11. Write a C function to perform the following i. Reversing a singly linked list ii. Concatenating singly linked list. iii. Finding the length of the circular linked list. iv. To search an element in the singly linked list
12. Write a node structure of linked stack. Write a function to perform push and pop operations on linked stack.

13.Write a function for singly linked lists with integer data, to search an element in the list that is unsorted and a list that is sorted.

14. Write a node structure of linked queue. Write a function to perform enqueue and dequeue operations on linked stack.