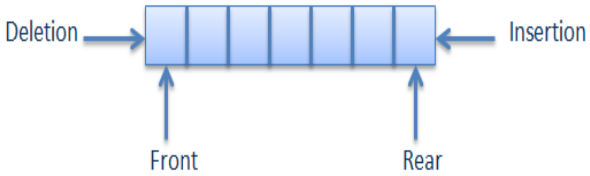
Module 2

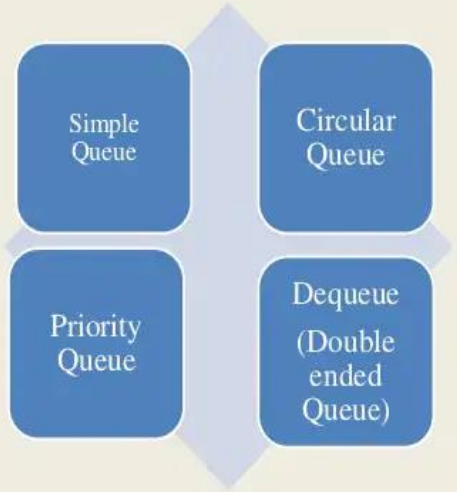
Queues

- A Queue is a special type of data structure where elements are inserted from one end and deleted from other end.
- The end at which elements are added is called rear
- The end at which elements are deleted is called front
- The first element inserted is the first element to be deleted out. Hence it is **First In First Out (FIFO) or Last In Last Out (LILO) data structure.**



FIFO/LILO

Type of queue



“How queues can be represented?”

The queues can be represented using following two data structures:

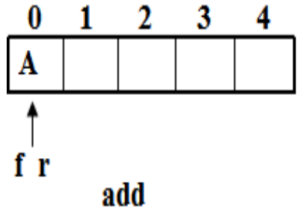
- Arrays
- Linked lists

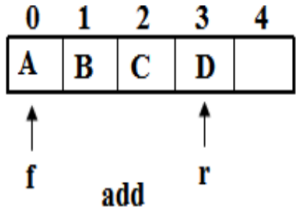
Linear queue (Ordinary queue)

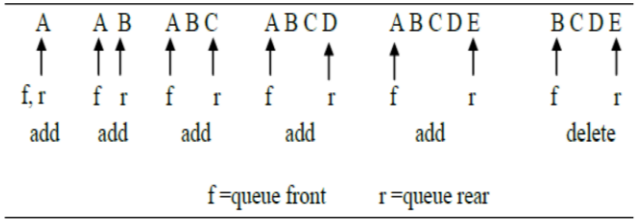
- The linear queue or ordinary queue or simply a queue are one and the same.
- A queue (linear queue or ordinary queue) is a special type of data
- structure (an ordered collection of items) where elements are inserted from one end and elements are deleted from the other end.
- hence, queue is also called First In First Out (FIFO) data structure.

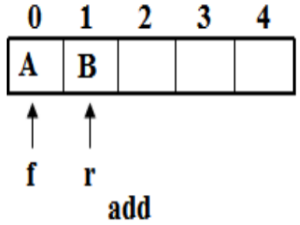
Array Representation

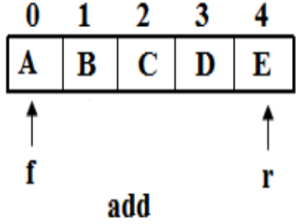
F=-1, r= -1

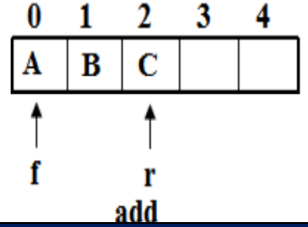


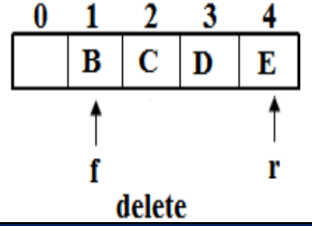












Queue operations

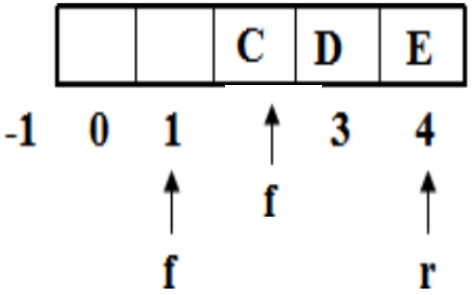
- Insert (An element is inserted from the rear end)
- Delete (An element is deleted from the front end)
- Display (Display the status of queue and queue contents)

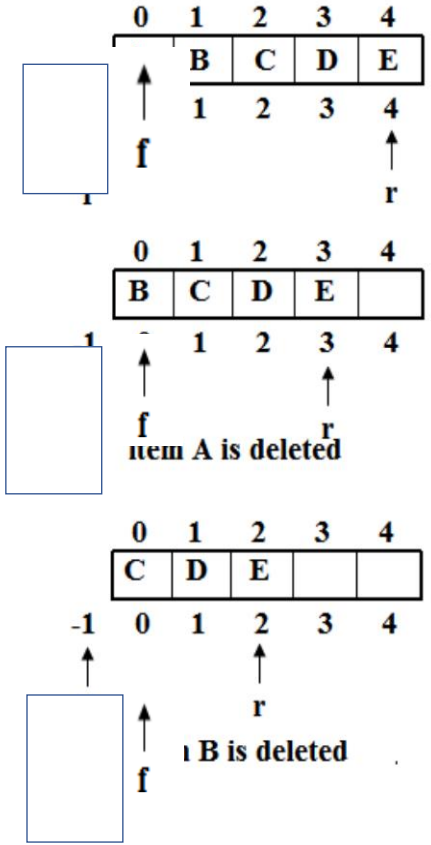
Functions to insert, delete and display operations on linear queue

```
#define MAX 10
int queue[MAX];
int f=-1;
int r=-1;

void insert()
{
    if(rear==MAX-1)
        printf("overflow");
    else
        if(f== -1)
        {
            f++;
        }
printf("enter element to be inserted");
        scanf("%d", &ele);
        Queue[++r]=ele;
}
```

```
void delete()
{
    If(front== -1 || front>rear)
        printf("underflow");
    else
        Printf("element deleted is %d",
queue[front]);
        front++;
}
  void display()
  {
      if(front==-1 || front>rear)
          Printf("display not possible");
      else
          Printf("queue elements are:");
          For(int i=f; i<=r; i++)
              Printf("%d\t", queue[i]);
  }
```

Disadvantage of queue





Solution 1:

Method 1: Shift left:

Solution 2:Using circular representation

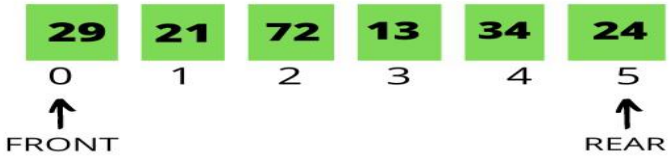
we increment rear by 1 and then insert the item as shown below:

rear = rear + 1;

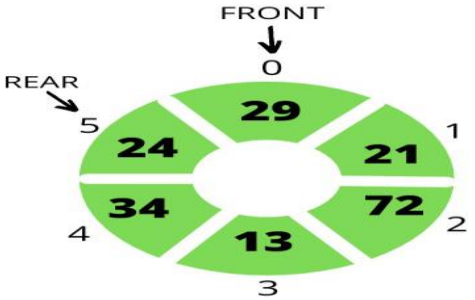
q[rear] = item;

In circular representation, we increment rear by using operator '%' as shown below:

rear = (rear + 1) % QUE_SIZE;

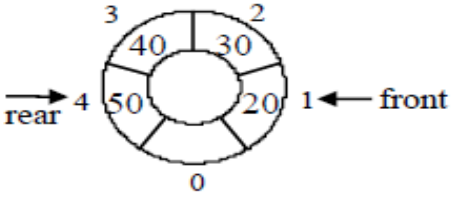


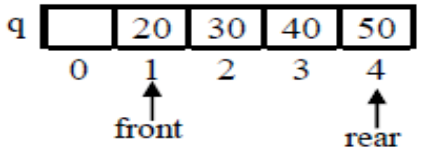
LINEAR QUEUE



CIRCULAR QUEUE

ent

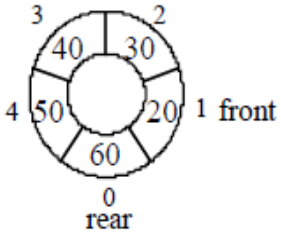


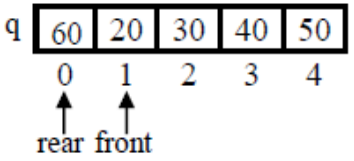


- Now if we want to insert an item 60, we have to increment rear by 1. In the
- above figure, the value of rear is 4. If we increment rear by 1, its value will be 5.
- Since we assumed it as circular queue, rear value is calsulated as

  rear = (rear + 1) % QUE_SIZE

- The value of rear will be 0 so that item 60 can be inserted at 0th position as shown below:

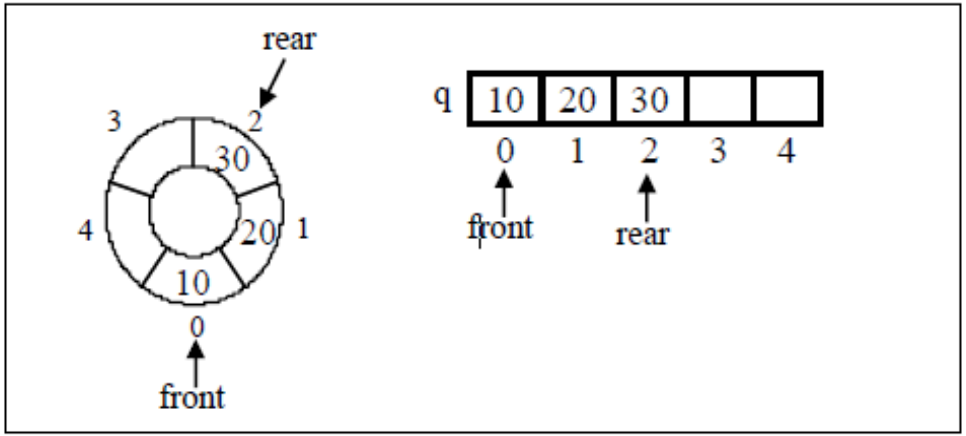




- The most fundamental operations in the queue ADT include:
  - enqueue()
  - dequeue()
  - peek()
  - isFull()
  - isEmpty().

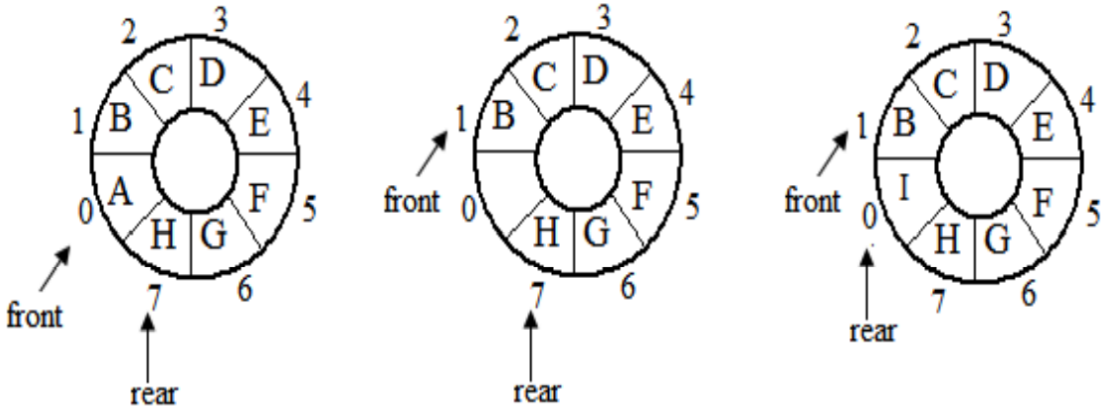
Circular Queues

- In circular queue, the elements of a given queue can be stored efficiently in an array so as to “wrap around”.
- The rear end of the queue is followed by the front of queue.
- The pictorial representation of a circular queue and its equivalent representation using an array are given side by side.

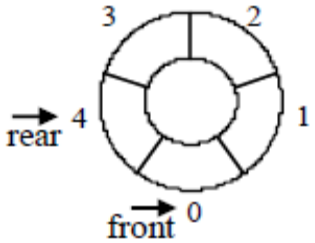


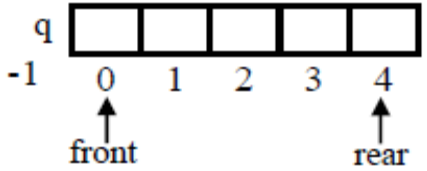
This circular representation allows the entire array to store the elements without shifting any data within the queue.

(rear +1) % MAX_QUEUE_SIZE



Empty queue: The circular representation and its equivalent array representation when queue is empty is shown below:





- When queue is empty, front = 0 and rear = -1.
- But, in circular queue, just before index 0 we have index 4.
- So, instead of rear = -1, we can write rear = 4

```
front = 0
rear = 4;                    // rear = 4. In general, rear = QUE_SIZE – 1
```

- Example: Show the contents of circular queue after performing each of the following operations”

  a) Empty queue

  b) Insert 10

  c) Insert 20 and 30

  d) Insert 40 and 50

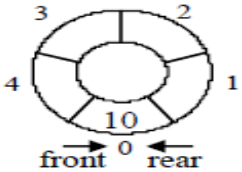
  e) Insert 60

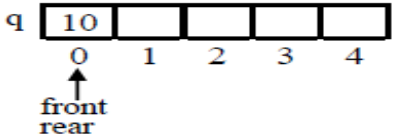
  f) Delete two items

  g) Insert 60 and 70

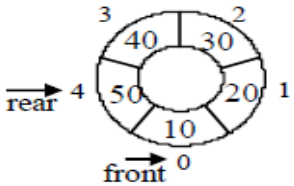
  h) Insert 80

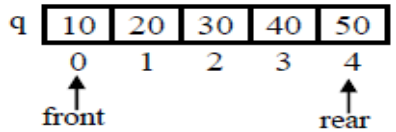
- Whenever front is 0 and rear is either -1 or QUE_SIZE – 1, queue is empty.
- Inserting 10: After incrementing rear by 1, 10 is inserted as shown below:



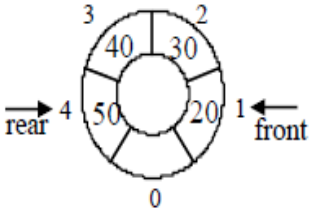


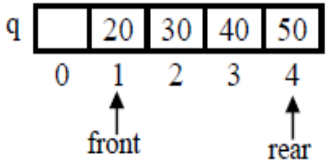
- Increment rear by 1 and insert 20. Again increment rear by 1 and insert 30.
- Inserting 40 and 50: Increment rear by 1 and insert 40. Again increment rear by 1 and insert 50. Now it looks like



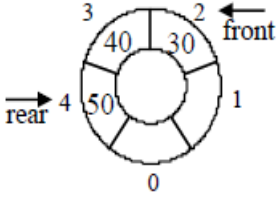


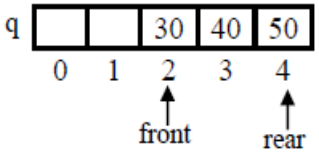
- Inserting 60: Queue is full. It is not possible to insert any element into queue. So, contents of queue have not been changed.
- **Delete**: An item has to be deleted always from the front end. So, 10 is deleted
- and contents of queue after deleting 10 is shown below:





Incrementing rear by 1, its value is 0 (because of its circular representation) and insert 60 at 0th location as shown below:





Functions to insert, delete and display operations on Circular queue

```
#define MAX 10
int queue[MAX];
int f=-1;
int r=-1;

void InsertQ()
{
/* Check for overflow of queue */
if (count == MAX)
{
printf("Queue overflow\n");
return;
}
rear = (rear + 1) % MAX;
q[rear] = item;
count++;
}
```

```
void delete()
{
if(count==0)
printf("queue underflow\n");
else
{
printf("deleted item is%c\n",q[f]);
 f=(f+1)%MAX;
count--;
}
}
```

```
void display()
{
intj=f,i;
if(count==0) printf("queue is empty\n");
else
{
printf("contents of circular queue\n");
for(i=1;i<=count;i++)
{
printf("%c\t",q[j]); j=(j+1)%MAX;
}
printf("total number of items=%d\n",count);
}
}
```

Double ended queue (Dequeue)

A Dequeue is a special type of data structure in which insertions are done from both ends and deletions are done at both ends.

Operations performed on dequeues:

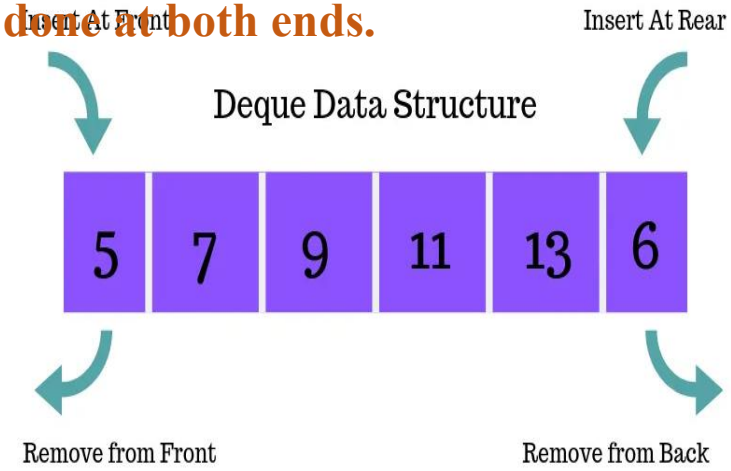
-> insert an item from front end

-> insert an item from rear end

-> delete an item from front end

-> delete an item from rear end

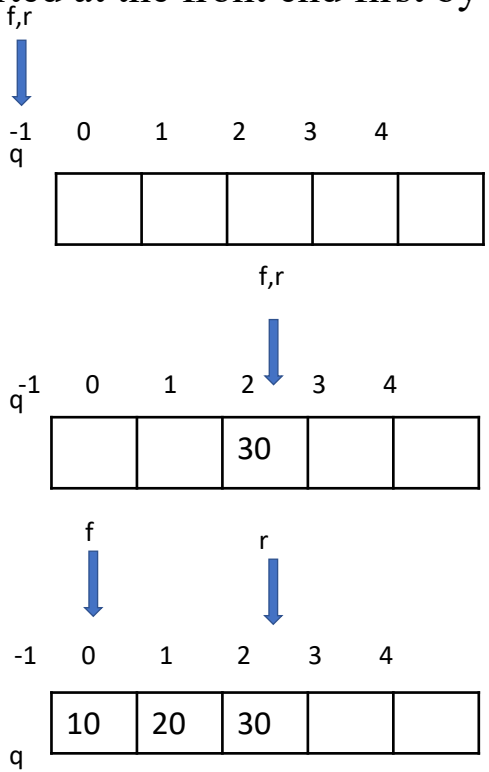
-> display the contents of queue



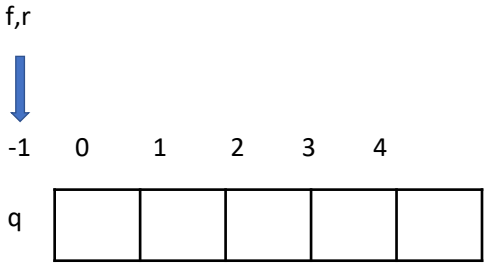
Insert at the front end

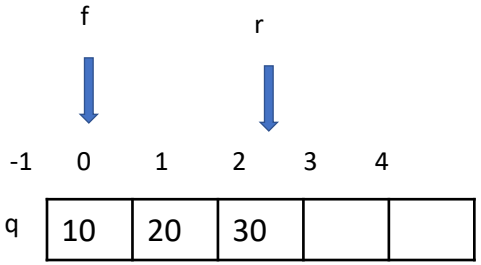
Queue empty: When queue is empty, an item can be inserted at the front end first by incrementing r by 1 and then insert an item.

```
void insertfront()          /* Case 1: Insert when Q empty */
{
    If(f==-1 && r==-1)
          f++;
          q[++rear]=item;

    if ( front != 0 ) /* Case 2: Insert when items are present */
{
q[--front] = item;
return;
}
 else
          Printf("front insertion not possible");
}
```



```
void deleterear()
{
    If(f==-1 && r==-1)
        Printf("no items");

    else
        Printf("item deleted is %d", q[r]);
        r--;
}
```





There are two variations of a deque

1. **Input-restricted deque** is a deque which allows insertions at only one end of the list

but allows deletions at both ends of thelist

2. **Output-restricted deque** is a deque which allows deletions at only one end of the list

but allows insertions at both ends of the list.

PRIORITY QUEUES

- A queue in which we are able to insert items or remove items from any position based on some priority is often referred to as a priority queue.
- Always an element with highest priority is processed before processing any of the lower priority elements.
- If the elements in the queue are of same priority, then the element, which is inserted first into the queue, is processed.

The priority queues are classified into two groups:

- Ascending priority queue
- Descending priority queue

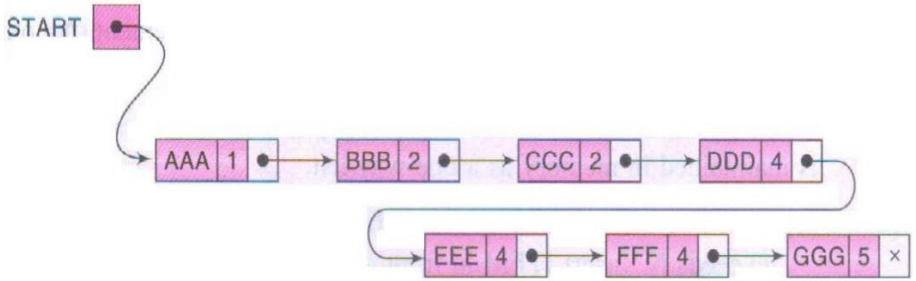
- **One way to maintain a priority queue in memory is by means of a one-way list, as follows:**

  1. Each node in the list will contain three items of information: an information field INFO, a priority number PRN and a link number LINK.

  2. A node X precedes a node Y in the list

  a. When X has higher priority than Y

  b. When both have the same priority but X was added to the list before Y.



Algorithm: This algorithm deletes and processes the first element in a priority queue which appears in memory as a one-way list.

1. Set ITEM:= INFO[START] [This saves the data in the first node.]
2. Delete first node from the list.
3. Process ITEM.
4. Exit.

Algorithm to add an element to priority queue

.

Algorithm: This algorithm adds an ITEM with priority number N to a priority queue which is maintained in memory as a one-way list.

1. Traverse the one-way list until finding a node X whose priority number exceeds N. Insert

ITEM in front of node X.

2. If no such node is found, insert ITEM as the last element of thelist.

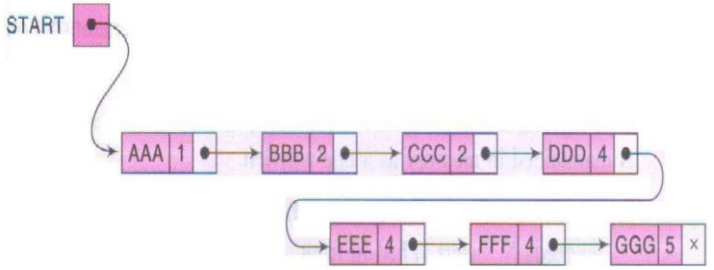
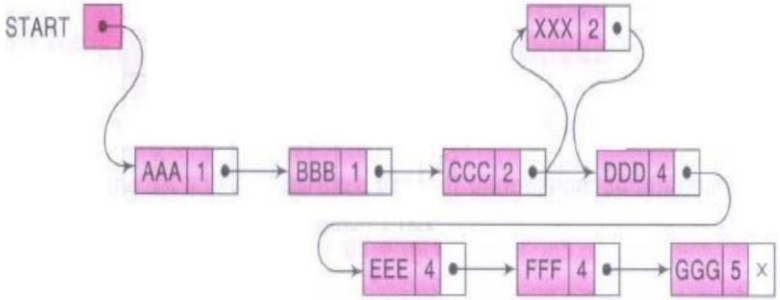


Fig (a)



Fig(b)

Circular Queue using Dynamic Arrays

"What is the disadvantage of using arrays to implement queue operations?"

- maximum size of the queue (QUE_SIZE) should be known during compilation.
- Once the size of the queue is fixed during compilation, it is not possible to alter the size of the queue during execution.

- This disadvantage we can overcome using dynamic arrays.
- Before inserting, we check whether sufficient space is available in the circular queue.

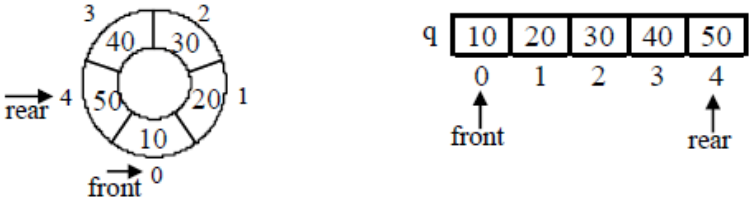
```
if (count == SIZE)
{
printf("Q full:\n");
return;
}
```

But, once the queue is full, instead of returning the control, we can increase the size of array using realloc() function.

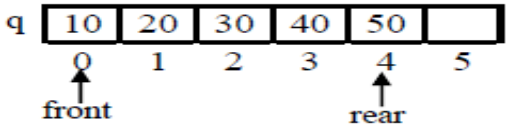
```
if (count == SIZE )
{
printf("Q Full: update size, insert item\n ");
SIZE++;
q = (int *) realloc(q, SIZE*sizeof(int) );
}
```

However, it is not sufficient to simply increase the size using realloc(). The queue contents have to be re-adjusted.

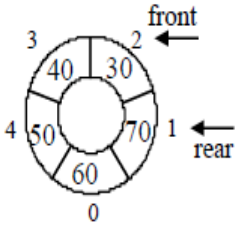
Case 1: The front index is less than the rear index: Consider the following circular queue with five elements whose capacity SIZE is 5.

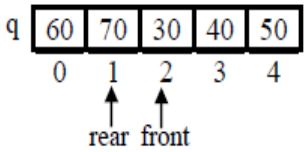


After increasing SIZE by 1 using realloc(), the contents of circular queue is shown below:

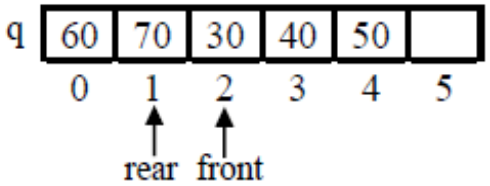


Case 2: The front index is greater than the rear index: Consider the following circular queue with five elements whose capacity SIZE is 5:





After increasing SIZE by 1 using realloc(), the contents of circular queue is shown below:



- To get the proper circular queue configuration, slide the elements 30, 40, 50 in above queue, to the right by one position using the following statements.
- q[5] = q[4]
- q[4] = q[3]
- q[3] = q[2] In general q[i+1]=q[i] for SIZE – 2 down to front.

shifting towards right by 1 position can be coded as shown below:

```
for (i = SIZE – 2; i >= front; i--)
{
q[i+1] = q[i];
}
```

- This re-adjusting is done only after re-allocating the memory. Now, the code can be written as shown below:

```
if (count == SIZE )
{
printf("Q Full: update size, insert item\n ");
SIZE++;
q = (int *) realloc(q, SIZE*sizeof(int) );
if (front > rear)
{
for (i = QUE_SIZE – 2; i >= front; i--)
{
q[i+1] = q[i];
}
front++; }}
```

Multiple Stack and Queues

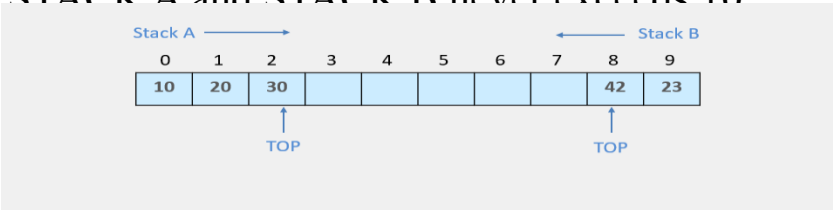
- A single stack is sometimes not sufficient to store a large amount of data.
- To overcome this problem, we can use multiple stack.
- A single array having more than one stack. The array is divided for multiple stacks.

Suppose there is an array STACK[n] divided into two stack STACK A and STACK B, where n = 10.

STACK A expands from the left to the right, i.e., from 0th element.

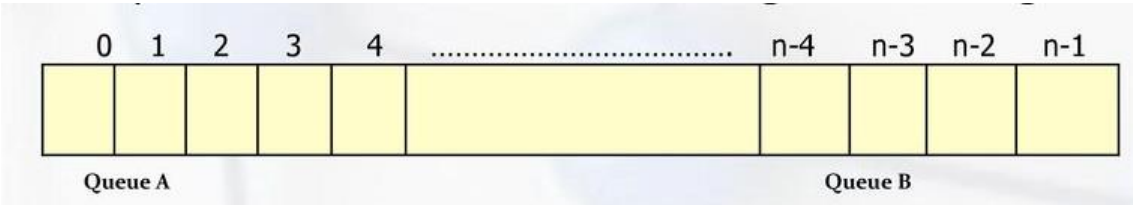
STACK B expands from the right to the left, i.e., from 10th element.

The combined size of both STACK A and STACK B never exceeds 10



Multiple Queues

- When implementing 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, but this results in sheer wastage of memory.
- Thus, there lies a tradeoff between the frequency of overflows and the space allocated.
- A better solution to deal with this problem is to have multiple queues or to have more than one queue in the same array.



```
#define MEMORY-SIZE 100              /* size of memory*/
#define MAX-STACKS 10                /*max number of stacks plus 1 */
/* global memory declaration
element memory[MEMORY—SIZE];
int top[MAX—STACKS] ;
int boundary[MAX—STACKS] ;
int n;                                        /* number of stacks entered by the user */
/*To divide the array into roughly equal segments we use the following code*/
top[0] = boundary[0]= -1;
for(i=1;i < n;i++)
top[i] = boundary[i]= (MEMORY_SIZE/n)*i;
boundary[n] = MEMORY_SIZE-1;
```

LINKED LISTS

"What are various advantages of using arrays?"

- **Data accessing is faster:** Using arrays, the data can be accessed very efficiently just by specifying the array name and the corresponding index.
  - For example, we can access ith item in the array A by specifying A[i].
  - The time taken to access a[0] is same as the time taken to access a[10000].
- **Simple**: Arrays are simple to understand and use.

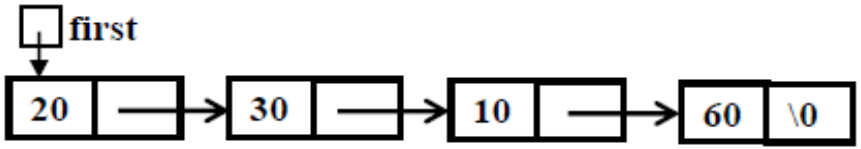
"What are the disadvantages of arrays?"

- The size of the array is fixed: The size of the array is fixed during compilation time only.
- Insertion and deletion operations involving arrays is tedious job
- The above disadvantages can be overcome using linked lists.

Linked list

- Now, let us see “What is a linked list?”
- Definition: Linear Non-Primitive **Dynamic** Data Structure which is a collection of **zero or more** nodes. Each **Nodes** has **Information** & **link** field
- info – This field is used to store the data or information to be manipulated
- link – This field contains address of the next node.

The pictorial representation of a singly linked list is shown below:

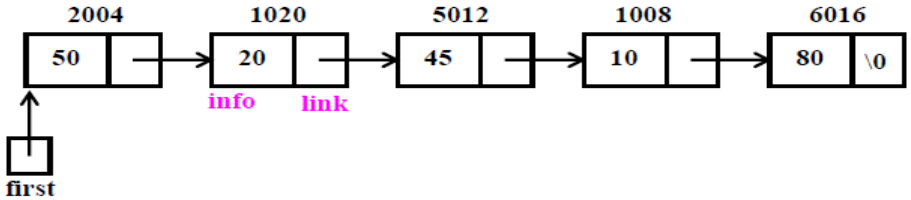


Types of Linked list

- Singly linked lists
- Doubly linked lists
- Circular singly linked lists
- Circular doubly linked lists

Singly Linked List

- A singly linked list is a collection of zero or more nodes where each node has two or more fields but only one link field which contains address of the next node.
- Each node in the list can be accessed using the link field which contains address of the next node.



- The variable first contains the address of the first node of the list.
- The entire list can be identified by the name first.
- Using the variable first, we can access any node in the list.

- An empty linked list is a pointer variable which contains NULL (\0).
- This indicates that linked list does not exist.
- For example, an empty list can be written as shown below:

Create an empty list

An empty list can be created by assigning NULL to a self-referential structure variable.

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

typedef struct node *NODE;

NODE first;                    // first is self-referential structure variable
```

- An empty list identified by the variable first is pictorially represented as shown below:

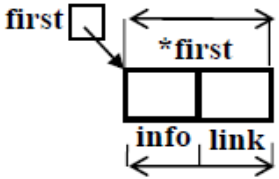
C Function to get a new node from the availability list

```
NODE getnode()
{
        NODE x;
        x = ( NODE ) malloc(sizeof(struct node));          /* allocate the memory space */
        if ( x == NULL )                          /* Free nodes don't exist */
        {
        printf("Out of memory\n");                          /* Allocation failed */
        exit(0);                                  /* Terminate the program */
        }
        return x;                                 /* allocation successful */
        }
```

Create a node with specified item

1. get a node: A node which is identified by variable first with two fields: info and link fields can be created using getnode() function.

first = getnode();

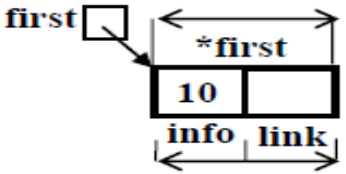


Observe from the above figure that:

- Using the variable first we can access the address of the node
- Using *first we can access entire contents of node
- Using (*first).info or first->info, we can access the data stored in info field
- Using (*first).link or first->link, we can access the link field.

2. The data item 10 can be stored in the info field using the following statement:

first->info = 10;



or

(*first).info = 10;

3. Store NULL character: After creating the above node, if we do not want link field to contain address of any other node, we can store '\0' (NULL) in link field.

first->link = NULL;

or

(*first).link = NULL;

Delete a node

- A node which is no longer required can be deleted using free() function in C as shown below:

  free (variable);

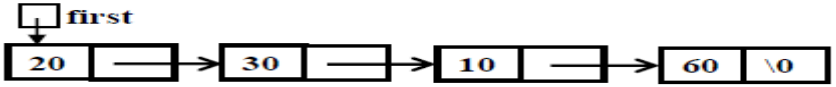
- For example: **free(first);**

Operations on singly linked lists

- IsEmpty: determine whether or not the list is empty
- InsertNode: insert a new node at a particular position
- FindNode: find a node with a given value
- DeleteNode: delete a node with a given value
- DisplayList: print all the nodes in the list

Insert a node at the front end

Consider the list as shown below

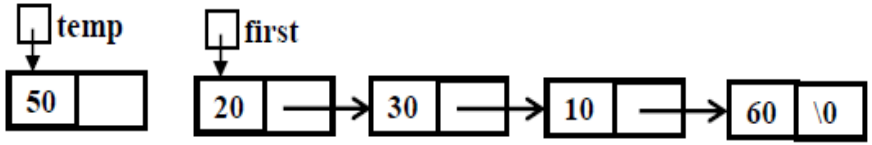


Step 1: Create a node using getnode() function as shown below:

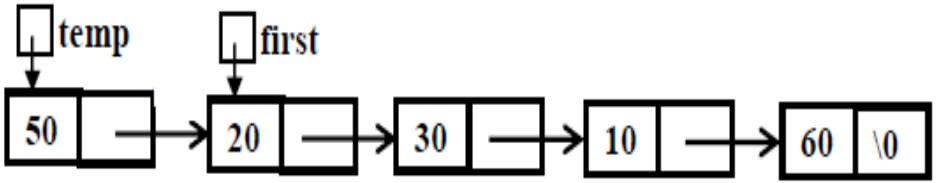
temp = getnode()

Step 2: Copy the item 50 into info field of temp using the following statement:

temp->info = item;



Step 3: Copy address of the first node of the list stored in pointer variable first into link field of temp using the statement: **temp->link = first;**



Step 4: Now, a node temp has been inserted and observe from the figure that temp is the first node. Let us always return address of the first node using the statement:

return temp;

Inserting a Node @ First/Front

```
NODE insert_first( int item, NODE first )
{
        NODE temp;
        temp = getnode( ) ;
        temp->info = item;
      temp->link = first;
      return temp;
}
```

```
case 1: printf("Enter the item to be inserted:");
        scanf("%d",&item);
        first = insert_first(item,first);
        break;
```

Displaying a Node

```
void display ( NODE first  )
{          NODE temp;
           if(first == NULL)
           {
           printf ("List is Empty");   return ;
          }
         else
          {  temp = first;
               while( temp != NULL )
               {   printf(" %d \t %p" , temp->info, temp->link  );
                   temp = temp->link;
```

Inserting a Node @ Last/Rear

```
NODE insert_rear( int item, NODE first )
{       NODE temp;                              /* Points to newly created node */
        NODE cur;                               /* To hold the address of the last node */
        temp = getnode ( );                     /* Obtain a new node and copy the item */
        temp -> info =  item;
        temp -> link = NULL;
/* If list is empty return new node as the first node*/
        if (first == NULL ) return temp;
        cur = first;                            /* If list exists, obtain address of the last node */
        while ( cur -> link != NULL )
        {
          cur = cur -> link;
        }
        cur -> link = temp;                             /* Insert the node at the end */
```

Deleting a Node @ Front

```
NODE delete_front( NODE first )
{       NODE temp;
        if ( first == NULL )
        {
        printf("List is Empty, Deleting is not possible");
        return first;
       }
        else
       {
          temp = first;                    /* Retain address of the node to be deleted */
          temp = temp -> link;             /* Obtain address of the second node */
          printf ("Deleted Item is %d", first -> info);
          free (first);
```

Deleting a Node @ Last/Rear

```
NODE delete_rear( NODE first )
{
        NODE cur, prev;
        if( first== NULL )
        {
          printf("List is Empty, Deleting is not possible");
          return first ;
        }
        if( first->link == NULL)                    // deleting very first node
         {
           printf ("deleted item %d",  first->info );
           free( first );
```

Deleting a Node @ Rear

```
prev = NULL;  cur = first;
while( cur->link   !=  NULL)
 {
   prev = cur;
   cur = cur->link ;
 }
printf ("Deleted Item %d",  cur->info);
free( cur);
prev->link = NULL;
return first;
```

Application of Linked list in Computer Science

- Representation of Polynomials.
- Arithmetic operations on Polynomials.
- Addition of Long Positive integers.
- Implementation of Queue Data structure.
- Implementation of Stack Data structure.
- Representation Adjacency List.

REPRESENTING CHAIN IN C

The following capabilities are needed to make linked representation

1. A mechanism for defining a node's structure, that is, the field it contains. So selfreferential structures can be used.

2. A way to create new nodes, so MALLOC functions can do this operation

3. A way to remove nodes that no longer needed. The FREE function handles this operation.

```
struct node
{
   int data;
   struct node *next;
};
struct node *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

STACK using SLL

stack is a special type of data structure where elements are inserted at one end and elements are deleted from the same end.

That is, if an element is inserted at front end, an element has to be deleted from front end.

If an element is inserted at rear end, an element has to be deleted from the rear end.

Operations:

- Push : insert_front( ) OR insert_rear()
- Pop: delete_front( ) OR delete_rear()
- Display: display( ) OR display()

Exceptional Conditions

- Stack Underflow
- Stack Overflow

QUEUE using SLL

queue is a special type of data structure where elements are inserted at one end and elements are deleted at the other end.

if an element is inserted at front end, element has to be deleted from rear end. If an element is inserted at rear end, element has to be deleted from the front end.

Operations:

- Enqueue: insert_front( )		insert_rear( )
- Dequeue: delete_rear( )	OR	delete_front()
- Display: display( )		display( )

Exceptional Conditions

- Queue Empty

```
struct node
{
   int data;
   struct node *next;
};
struct node *front;
struct node *rear;
void insert();
void delete();
void display();
void main ()
{
   int choice;
while(choice != 4)
   {
 printf("\n1.insert an element\n2.Delete an element\n3.Display
the queue\n4.Exit\n");
       printf("\nEnter 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("\nEnter valid choice??\n");
       }  }  }
```

```
void insert()
{
   struct node *ptr;
   int item;
  ptr = (struct node *) malloc (sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW\n");
      return;
   }
   else
   {
      printf("\nEnter value?\n");
      scanf("%d",&item);
```

```
      ptr -> data = item;
         if(front == NULL)
         {
            front = ptr;
            rear = ptr;
            front -> next = NULL;
            rear -> next = NULL;
         }
         else
         {
            rear -> next = ptr;
            rear = ptr;
            rear->next = NULL;
         }
   }
```

```
void delete ()
{
   struct node *ptr;
   if(front == NULL)
   {
      printf("\nUNDERFLOW\n");
      return;
   }
   else
   {
      ptr = front;
      front = front -> next;
      free(ptr);
```

```
void display()
{
   struct node *ptr;
   ptr = front;
   if(front == NULL)
   {
      printf("\nEmpty queue\n");
   }
   else
   {   printf("\nprinting values .....\n");
      while(ptr != NULL)
      {
         printf("\n%d\n",ptr -> data);
         ptr = ptr -> next;
      }
```

APPLICATIONS OF LINKED LISTS – POLYNOMIALS

A polynomial is a collection of different terms, each comprising coefficients, and exponents.

It can be represented using a linked list. This representation makes polynomial manipulation efficient.

Each node of a linked list representing polynomial constitute three parts:

The first part contains the value of the coefficient of the term.

The second part contains the value of the exponent.

The third part, LINK points to the next term (next node).

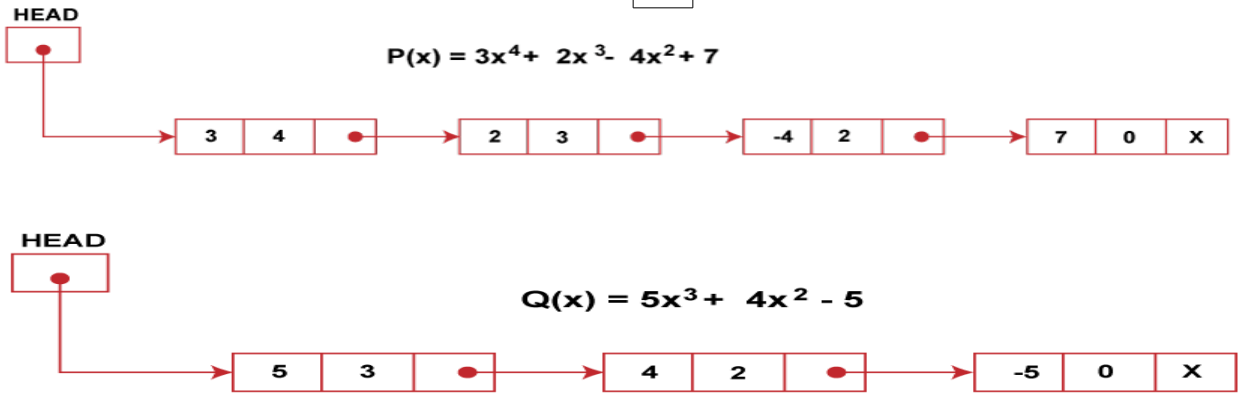
Node representing a term of a polynomial

Let us consider an example an example to show how the addition of two polynomials is performed,

P(x) = 3x4 + 2x3 - 4 x2 + 7

Q (x) = 5x3 + 4 x2 - 5

These polynomials are represented using a linked list in order of decreasing exponents as follows:



Adding Polynomials

To add two polynomials, examine their terms starting at the nodes pointed to by a and b.

• If the exponents of the two terms are equal, then add the two coefficients and create a new term for the result, and also move the pointers to the next nodes in a and b.

• If the exponent of the current term in a is less than the exponent of the current term in b, then create a duplicate term of b

• If the exponent of the current term in b is less than the exponent of the current term in a, then create a duplicate term of a, attach this term to the result, c, and advance the pointer to the next term in a.

```
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int coef;
    int exp;
    struct Node* next;
};
typedef struct Node Node;
void insert(Node** poly, int coef, int exp)
{
    Node* temp = (Node*) malloc(sizeof(Node));
    temp->exp = exp;
    temp->next = NULL;

    if (*poly == NULL) {
        *poly = temp;
        return;
    }

    Node* current = *poly;

    while (current->next != NULL) {
        current = current->next;
    }

    current->next = temp;
}
```

```
void print(Node* poly) {
    if (poly == NULL) {
        printf("0\n");
        return;
    }
    Node* current = poly;
    while (current != NULL) {
        printf("%dx^%d", current->coef, current->exp);
        if (current->next != NULL) {
            printf(" + ");
        }
        current = current->next;
    }
    printf("\n");
}
```

```
Node* add(Node* poly1, Node* poly2) {
    Node* result = NULL;
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exp == poly2->exp) {
            insert(&result, poly1->coef + poly2->coef, poly1->exp);
            poly1 = poly1->next;
            poly2 = poly2->next;
        } else if (poly1->exp > poly2->exp) {
            insert(&result, poly1->coef, poly1->exp);
            poly1 = poly1->next;
        } else {
            insert(&result, poly2->coef, poly2->exp);
            poly2 = poly2->next;
        }
```

```
while (poly1 != NULL) {
    insert(&result, poly1->coef, poly1->exp);
    poly1 = poly1->next;
  }

  while (poly2 != NULL) {
    insert(&result, poly2->coef, poly2->exp);
    poly2 = poly2->next;
  }

  return result;
}
```

```
int main() {
    Node* poly1 = NULL;
    insert(&poly1, 5, 4);
    insert(&poly1, 3, 2);
    insert(&poly1, 1, 0);

    Node* poly2 = NULL;
    insert(&poly2, 4, 4);
    insert(&poly2, 2, 2);
    insert(&poly2, 1, 1);

    printf("First polynomial: ");
    print(poly1);
    printf("Second polynomial: ");
    print(poly2);

    Node* result = add(poly1, poly2);
    printf("Result: ");
    print(result);

    return 0;
}
```