RV Institute of Technology and Management®

Go, change the world

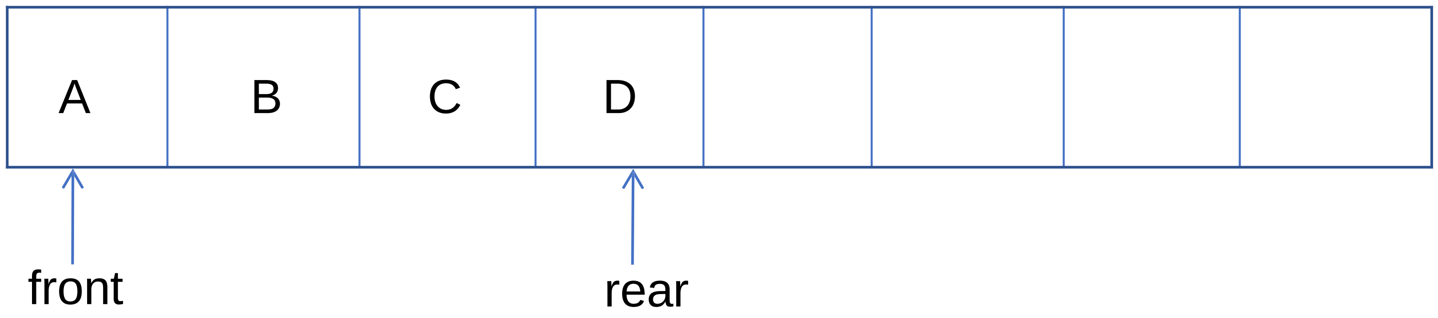
Course Name: Data Structures and Applications

Course Code: BCS304

Module 2

QUEUES

- Queue is a linear list of elements in which insertion and deletion take place at different ends. The end at which new element is inserted is called as rear and the end from which element is deleted is called as front
- Queue has the property - FIFO
- To implement queue front and rear variables are used



```
#define MAXQ 25
typedef struct {
        int front, rear;
        char item[MAX];
        }QUEUE;
QUEUE q;
q.rear = -1;
q.front = -1;
```

Primitive operations – Qinsert, Qdelete


```
Void Qinsert (char element)
{
   If (q.rear ==MAXQ-1) {
                    Queue full();
                    Exit(0);
                          }
                    q.item[++q.rear]=element;
            Return;
       }
Char Qdelete( )
{
If (q.front==q.rear)
Return(emptyQ( );
Return (q.item[++q.front]);
}
```

Disadvantage of ordinary queue

Once MAXQ items are added, even if we delete the items from Front, we can not add items in vacant places. This is because rear has reached MAXQ value and there is no way to come back to deleted element position.

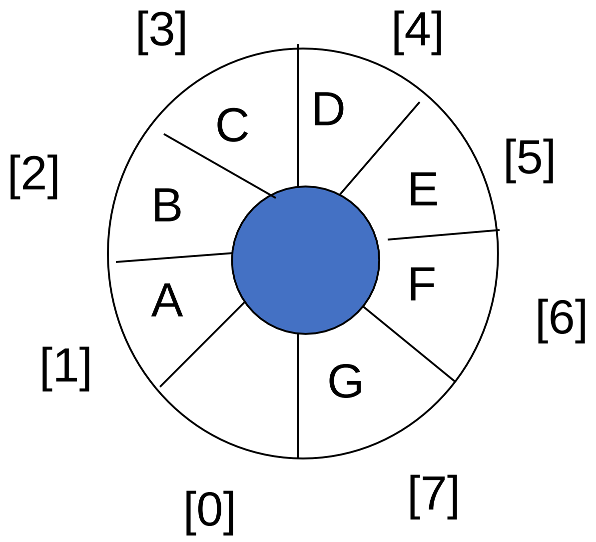
This problem may be solved by using array compaction on every item deletion or by viewing queue as circular not as straight line.

In array compaction method, Qdelete function should be changed as given below. In this case front is always 0.

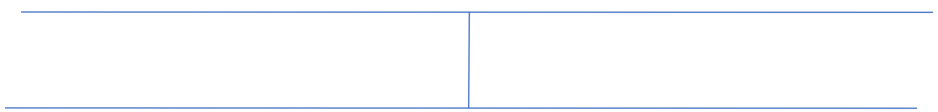
```
Char Qdelete( ){  If(q.rear>=0){  X=q.items[0];

For(i=0; i< q.rear; i++)
        q.items[i]=q.items[i+1];

q.rear--;

}}
```

Circular Queues

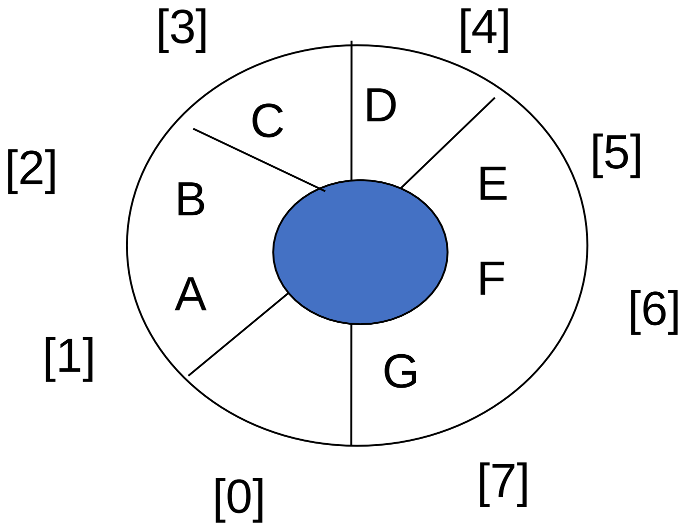
When an array is used to have a circular queue : Front is one position counterclockwise from the first element and rear is current end



Front=0, Rear=7



Circular Queue



Front=0, Rear=7

A	B	c	A	A	A	A	A

Algorithm for Circular Queue using Dynamically allocated Array

1. Create a new array newQueue of twice the capacity.
2. If front==0

   Copy queue[front+1] through queue[capacity-1] to positions in newQueue beginning at 0 and go to step 5.
3. Copy the second segment (ie, queue[front+1] through queue[capacity-1])to positions in newQueue beginning at 0.
4. Copy the first segment (ie, elements queue[0] through queue[rear] to positions in newQueue beginning at capacity-front-1.
5. Update rear=capacity -2,front=2*capacity-1, capacity=2*capacity

- Void addcq(char item)

  {

  rear=(rear+1)% capacity;

If (front==rear) qfull( );

Cq[rear]=item;

}

Note: The function Copy(a, b, c) copies elements from locations a through b-1 to locations beginning at c

```
Void qfull( ){ // queue is the current circular queue
char *new Queue;
new Queue= (char*)malloc(2*capacity*sizeof(*queue));
int start =(front+1)%capacity;
If(start<2)
Copy(queue+start, queue+start+capacity-1, new Queue);
Else {
Copy(queue+start, queue+capacity, new Queue);
Copy(queue, queue+start+rear+1, new Queue+capacity-start);
   }
Front=2*capacity-1;
Rear=capacity-2;
Capacity =2* capacity;
Free(queue);
Queue= newQueue;
 }
```

Multiple stacks and queues

Two stacks

m[0],   m[1],   …,   m[n-2],   m[n-1]

bottommost stack 1   bottommost stack 2

More than two stacks (n)

memory is divided into n equal segments

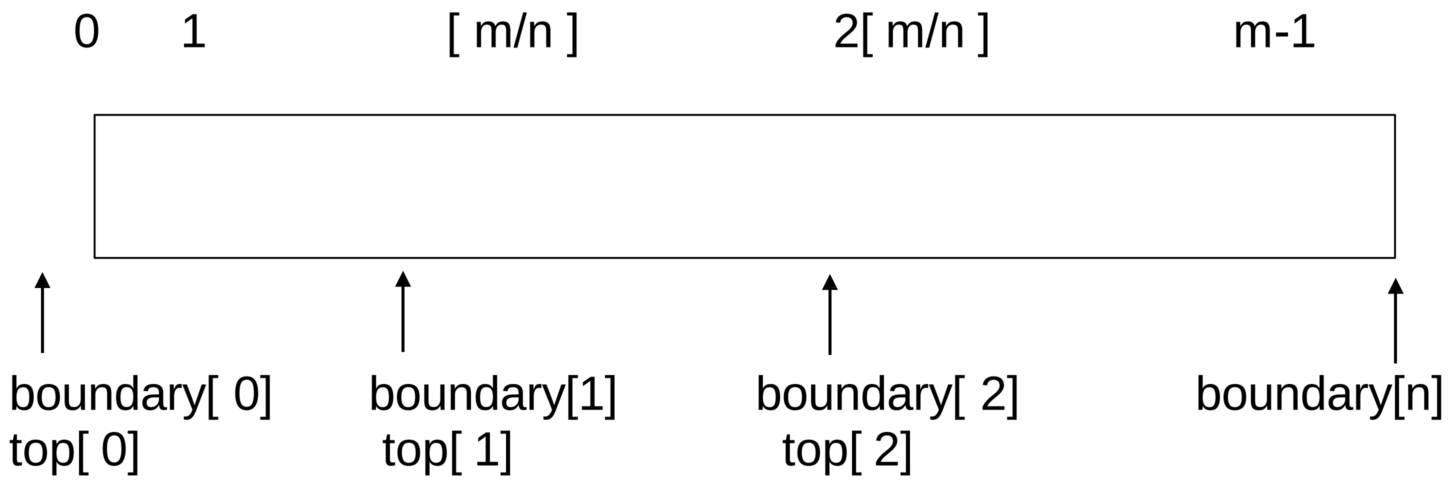
boundary[stack_no]

$0 \leq \text{stack\_no} < \text{MAX\_STACKS}$

top[stack_no]

$0 \leq \text{stack\_no} < \text{MAX\_STACKS}$

Initially, boundary[i]=top[i].

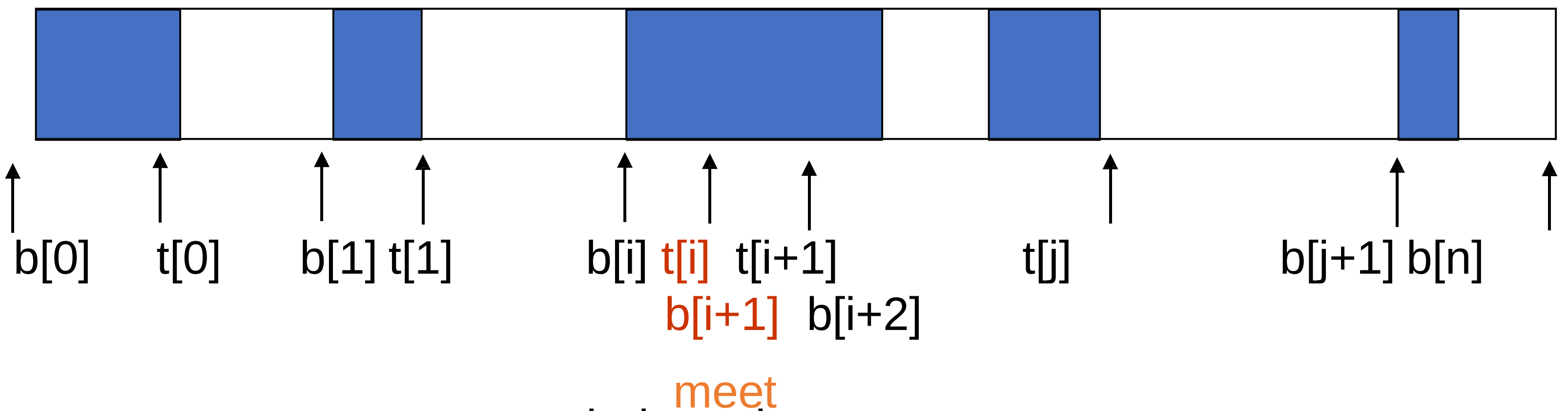


All stacks are empty and divided into roughly equal segments.

```
#define MEMORY_SIZE 100     /* size of memory */
#define MAX_STACK_SIZE 100 /* 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 */
top[0] = boundary[0] = -1;
for (i = 1; i < n; i++)
  top[i] =boundary[i] =(MEMORY_SIZE/n)*i;
boundary[n] = MEMORY_SIZE-1;
```

```
void add(int i, element item)
{     /* add an item to the ith stack */
    if (top[i] == boundary [i+1])
        stack_full(i);   may  have unused storage
        memory[++top[i]] = item;
}

element delete(int i)
{    /* remove top element from the ith stack */
    if (top[i] == boundary[i])
       return stack_empty(i);
    return memory[top[i]--];
}
```

Find j, stack_no < j < n such that top[j] < boundary[j+1]
or, $0 \leq j <$ stack_no



b=boundary, t=top

Linked List

Definition

Linked list is a collection of one or more nodes, where each node is a collection of one or more data field and link fields

Representing chains in C

- Structure definition to define a node to hold integer data is

```
typedef struct listNode *listPointer;
typedef struct
{
    int data;
    listPointer link;
}listNode;
```

Representing chains in C

- Structure definition to define a node to hold decimal data is

```
typedef struct listNode *listPointer;
typedef struct
{    int data1;
     float data2;
     listPointer link;
}listNode;
```

Representing chains in C

- After defining the node’s structure, we create a new empty list. This is accomplished by the statement:

```
listPointer first=NULL;
```

Types of Linked list

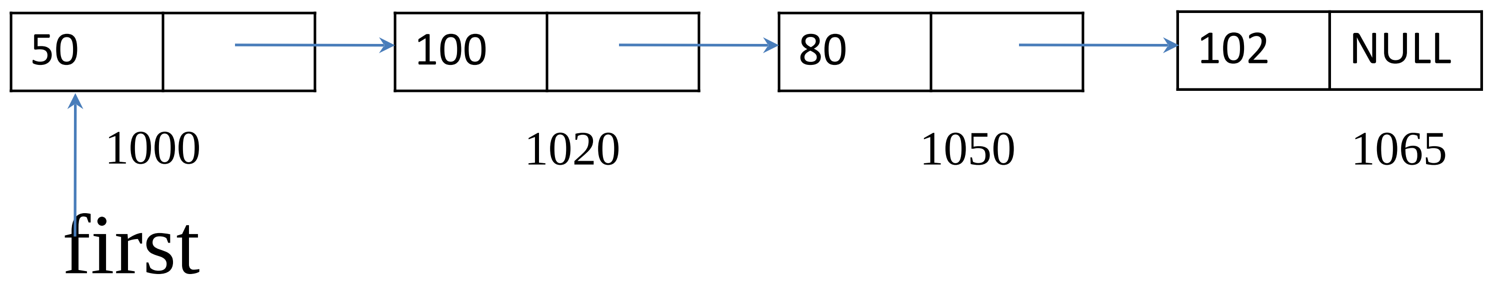
- Singly Linked List (SLL)
- Doubly Linked List(DLL)
- Circular Linked list(CLL)
  - Circular Singly Linked List(CSLL)
  - Circular Doubly Linked List(CDLL)

Singly Linked List

- Definition:

  Singly linked list is a linear list of data with exactly one link field and one or more data field in the node.

Example:



Linked list

- List can be generated in two directions
  - The Front end
  - The Rear end

Linked list : Basic operations

- The different basic operations can be performed on Linked list are:

  - Inserting a node at front end
  - Inserting a node at rear end

  - Deleting a node at font end
  - Deleting a node at rear end

  - Displaying the contents

Function: main()

```
void main()
{
      listPointer first=NULL;
      int  item,choice;
      while(1)
      {
              switch(choice)
              {
                      case 1:   printf("Enter the data to be inserted\n");
                                sacnf("%d",&item);
                                first=inert_front(item,first);
                                break;
                      case 2: display(first);
                              break;
              }
      }

}
```

SLL: C function to insert a node at the front end

Algorithm: Steps to follow in insert front

1. Allocate memory to the node temp
2. Load the fields with suitable data
3. Check list emptiness, if list is empty make the temp node as first
4. If list is not empty, link the temp node to first node of the existing node

```
listPointer insert_front(int item, listPointer first)
{
        listPointer temp;
        temp=(listPointer)malloc(sizeof(listPointer)
        );
        temp->info=item;
        temp->link=NULL;
        if(first!=NULL)

                temp->link=first;  return

        temp;
}
```

SLL: C function to display linked list

Algorithm:

Steps to follow

1. Check for list emptiness, if list is empty print suitable message
2. If list is not empty, print the data from first node to last node

```
display(listPointer first)
{
        listPointer temp;
         if(first==NULL)
        {
                printf("List is Empty\n");
                return;
         }
         else
        {
                temp=first;
                printf("The Linked List contents
are\n");
                while(temp!=NULL)
                {
                        printf("%d\t",temp->data);
                        temp=temp->link;
                }
```

Algorithm:

Steps to follow

1. Allocate memory to the node temp
2. Load the fields with suitable data of temp
3. Check list emptiness, if list is empty make the temp node as first
4. If list is not empty, search for the list's last node, once found: attach the 'temp' node to it

SLL: C function to insert a node at the rear end

```
listPointer insert_rear(int item,listPointer first)
{
        listPointer temp;
        temp=(listPointer)malloc(sizeof(listPointer)
);
        temp->data=item;
        temp->link=NULL;
        if(first==NULL)
                return temp;
        else
        {
                listPointer cur=first;
                if(cur!=NULL)
                {
                        cur=cur->link;
                }
                cur->link=temp;
                return first;
        }
```

SLL: C function to delete a node at the front end

Algorithm:

Steps to follow

1. Check for list emptiness, if list is empty print suitable message

1. If list is not empty, print the deleted data to the user in the front node and make the second node as first node

```
listPointer  delete_front(listPointer  first)
{
        listPointer  temp;
        if(first==NULL)
        {
                printf("List is Empty\n");
                return first;
        }
        temp=first;
        temp=temp->link;
        printf("Deleted data is \n %d \n",first->data);
        free(first);
        return temp;
}
```

SLL: C function to delete a node at the rear end

Algorithm:

Steps to follow

1. List is empty print the error message
2. If the list contains only one delete it and return NULL
3. If list contains more than one node:
   - Find the last node
   - Assign the 'link' field of the previous node to NULL
   - Delete the last

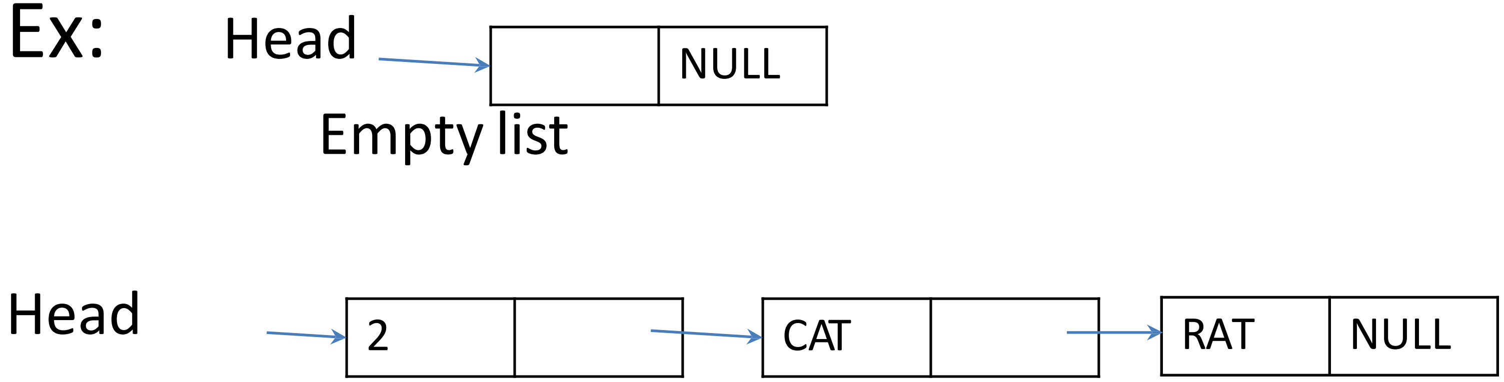
```
listPointer delete_rear(listPointer first)
{   if(first==NULL)
    {       printf("List is empty: Can not delete\n");
            return first;
    }
    if(first->link==NULL)
    {       printf("Data deleted %d\n",first->data);
            free(first);
            return NULL;
    }
    listPointer prev=NULL,cur=first;
    while(cur->link!=NULL)
    {       prev=cur;
            cur=cur->link;
    }
    printf("Date deleted is %d\n",cur->data);
    prev->link=NULL;
    free(cur);              cur=NULL;
}\*end of function*\
```

Linked List with Header Node

- What is a Header Node?

  Header node in a Linked List is a special node which contains the address of the first node with no data.

- Ex:



Linked List with Header Node

- It is specially named as “Head”

- If the Header node contains any data it will be always the “size of the List”

Linked List with Header Node

- Advantages of Header node
  - The implementation of the basic operations becomes easy with no preconditions checked in Insertion.
  - Renaming of “temp” as “first” is not required after every insertion at front end.
  - Size of the Linked list will be easily calculated (as it is the only content of the data field of the Header node).

Categories of the Linked list with Header node

- Singly Linked List with Header node
- Doubly Linked List with Header node
- Circular Singly Linked List with Header node
- Circular Doubly Linked List with Header node

Basic operations
on
Linked List with Header Node

- Inserting a node at the Front end
- Inserting a node at the Rear end

- Deleting a node at the Front end
- Deleting a node at the Rear end

- Displaying the list contents

RV Institute of Technology and Management®

Go, change the world

SLL with Header node: C function to insert a node at the front end

Algorithm: Steps to follow

Algorithm:isert_front_SLL_HN

//Input:data to be inserted as //item and the name of the list //as head

//Output: Linked list "head" //after inserting the data

BEGIN

1.Allocate memory to the node temp

2.Load the fields with suitable data

3.Create the links between :

- **Head node and the temp node**
- **Temp node and the first node of the existing list**

END

```
listPointer insert_front(int item, listPointer head)
{
    listPointer temp;
    temp=(listPointer)malloc(sizeof(listPointer));
    temp->info=item;
    temp->link=NULL;
    temp->link=head->link;
    head->link=temp;
    return head;
}
```

SLL with Header node : C function to insert a node at the rear end

Algorithm: Steps to follow

Algorithm: Insert_rear_SLL_HN

//Input: Data to be inserted as
//item and the name of the list
//as head

//Output: Linked list "head"
//after inserting the data

BEGIN

1.Allocate memory to the node temp

2.Load the fields with suitable data of temp

3.Find the last node of the list and then insert the temp node next to it

4.Return the address of the Head node.

END

```
listPointer insert_rear(int item,listPointer head)
{
    listPointer temp;
    temp=(listPointer)malloc(sizeof(listPointer));
    temp->data=item;
    temp->link=NULL;
    listPointer cur=head;
    while(cur->link!=NULL)
    {
            cur=cur->link;
    }
    cur->link=temp;
    return head;

}
```

SLL with Header node: C function to delete a node at the front end

Algorithm: Steps to follow

1. Check for list emptiness, if list is empty print suitable message

1. If list is not empty, print the deleted data to the user in the front node and make the second node as first node

```
listPointer  delete_front_SLL(listPointer  first)
{
        listPointer temp;
        if(head->link==NULL)
        {
                printf("List is Empty\n");
                return first;
        }
        temp=head->link;
        head->link=temp->link;
        printf("Deleted data is \n %d \n",temp->data);
        free(temp);
        temp=NULL;
        return head;
}
```

SLL with header node: C function to delete a node at the rear end

Algorithm:Steps to follow

Algorithm:
Insert_rear_SLL_HN
//Input: Data to be inserted as
//item and the name of the list
//as head
//Output: Linked list "head"
//after inserting the data

BEGIN
1.Allocate memory to the node temp
2.Load the fields with suitable data of temp
3.Find the last node and previous node of the list
4.Insert NULL into link field of previous node and delete current node
5.Return the address of the Head node.
END

```
listPointer delete_rear(listPointer head)
{
    if(head->link==NULL)
    {       printf("List is empty: Can not delete\n");
            return head;
    }
    listPointer prev=NULL,cur=head->link;
    while(cur->link!=NULL)
    {       prev=cur;
            cur=cur->link;
    }
    prev->link=NULL;
    printf("Deleted data is \n %d \n",cur->data);
    free(cur);
    cur=NULL;
    return head;
}\*end of function*\
```

SLL with header node: C function to display the list

Algorithm: display_SLL_HN
Algorithm: Steps to follow
//Input: Data to be inserted as
//item and the name of the list as
//head
//Output: Linked list "head" after
//inserting the data

BEGIN
1.Check list emptiness: if list is empty print suitable message, else go to next step
2.Display the content of the list from first node to last node

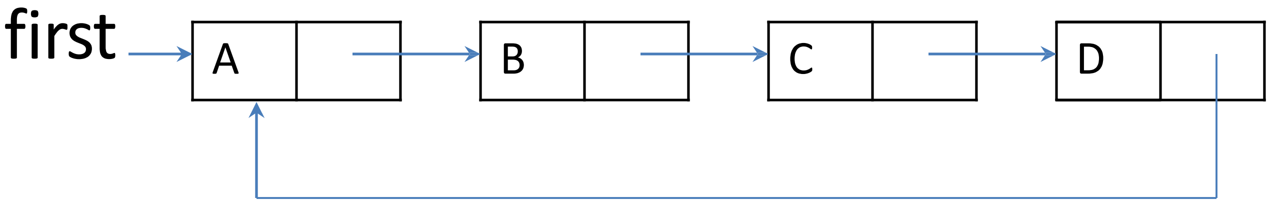
```
display(listPointer head)
{
        listPointer temp;
         if(head->link==NULL)
        {
                printf("List is Empty\n");
                return;
         }
         else
        {
                temp=head->link;
                printf("The Linked List contents are\n");
                while(temp!=NULL)
                {
                        printf("%d\t",temp->data);
                        temp=temp->link;
                }
         }
}
```

Circular Linked List

- Definition:

  Circular Linked List is a linear homogeneous list of zero or more nodes where the last node is followed by the first node

- Example:

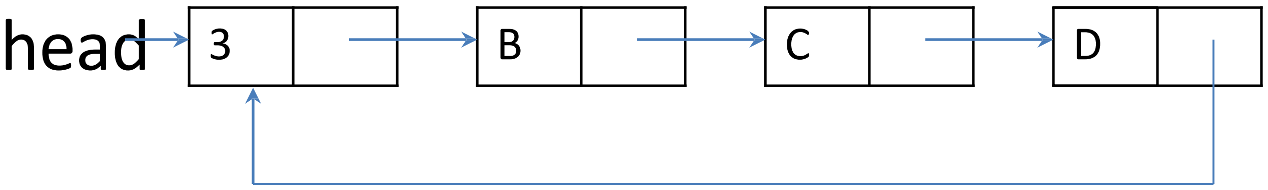


Circular Singly Linked List with Header Node

- Definition:

  Circular Singly Linked List is a linear homogeneous list of zero or more nodes with a special node called the “head”; where the last node is followed by the head node and each node consisting of exactly one link field

- Example:

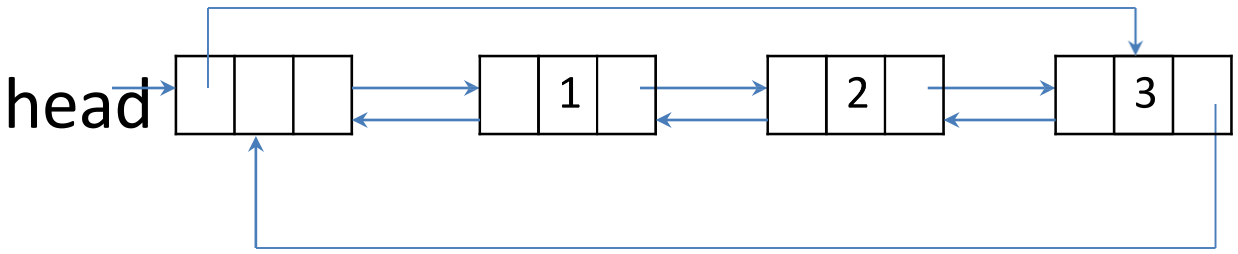


Circular Doubly Linked List with Header Node

- Definition:

  Circular Linked List is a linear homogeneous list of zero or more nodes with a special node called the “head”; where the last node is followed by the head node and each node consisiting of exactly two link fields

- Example:



Basic operations

- Inserting a node at the front end
- Inserting a node at the rear end

- Deleting a node at the front end
- Deleting a node at the rear end

- Displaying the list

Algorithm: Steps to follow

Algorithm: inertFront_CDLL

//Input: the name of the Circular doubly

//linked list as "head"

//Output: the newly created list "head"

//after inserting the data at front end

BEGIN

1. Allocate memory to node "temp"
2. Read the data to be stored
3. Load the fields of the node "temp" with suitable data
4. Identify the first node of the list as "cur"
5. Link the temp node to head node and temp node to current node

6.Return the head node

END

Circular DLL with Header node:C function to insert node at front end

```
listPointer inertFront_CDLL(ListPointer head)
{   listPointer temp,cur;
    temp=(listPointer)malloc(sizeof(listPointer));

    printf("Enter the item to be stored\n");
    scanf("%d",&item);

    temp->data=item;
    temp->llink=temp->rlink=NULL;

    cur=head->rlink;

    head->rlink=temp;
    temp->llink=head;
    temp->rlink=cur;
    cur->llink=temp;

    return head;
}
```

Linked Stacks

- Stack data structure created using Linked List
- Procedure
  - Stacks prefer the data should be entered at one end and deleted at the same end
  - Since Linked List has two ends, we can use any one end
    - Insert the data and delete the data at front end i.e.,
      - Pick Insert front and Delete front operations of Linked list
    - Insert the data and delete the data at rear end i.e.,
      - Pick Insert rear and Delete rear operations of Linked list

Linked Queues

- Queues data structure created using Linked List
- Procedure
  - Queues prefer the data should be entered at one end and deleted at the other end
  - Since Linked List has two ends, we can use any both ends
    - Insert the data at rear end and delete the data at front end i.e.,
      - Pick Insert rear and Delete front operations of Linked list
    - Insert the data at front end and delete the data at rear end i.e.,
      - Pick Insert front and Delete rear operations of Linked list

POLYNOMIAL

- Definition

  Polynomial is a collection of terms, each term taking the form

  $$ax^e$$

  where, a is the coefficient, x is the variable and e is the exponent

- Example

  $A(x) = 40x^{500} + 4x^3 - 3x^2 + 20$

  $B(x) = 5x^6 + 2x^2 + 1$

RV Institute of Technology and Management®

Go, change the world

Polynomial Representation

- Polynomial could be stored in two ways into memory of a computer

1. Using Structure consisting of an array
2. Using Array of Structures

Storage: Using Structure consisting of an Array

```
#define MAX_DEGREE 101 typedef struct
 {
     int degree;
     float coef[MAX_DEGREE];
} polynomial;

polynomial A,B;
```

- advantage: easy implementation
- disadvantage: waste space when sparse

Storage: Using Array of Structures

- To overcome the disadvantage of the previous storage, we can use one global array to store all the polynomials
- Structure definition #define MAX_DEGREE 101 typedef struct

```
  {
          int degree;
           float coef;
     } polynomial;
 polynomial P [MAX_DEGREE];
```

Data structure 2: use one global array to store all polynomials

Array representation of two polynomials

$A(X)=2X^{1000}+1$

$B(X)=X^4+10X^3+3X^2+1$

	starta	finisha	startb			finishb	avail
coef	2	1	1	10	3	1	
exp	1000	0	4	3	2	0	
	0	1	2	3	4	5	6

specification	representation
poly	<start, finish>
A	<0,1>
B	<2,5>

End of Module-2