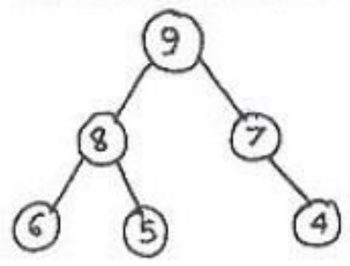
CMR
INSTITUTE OF
TECHNOLOGY

USN

Sub:	Data Structures and Applications							Code:	15CS33
Date:	03 / 11 / 2016	Duration:	90 mins	Max Marks:	50	Sem:	3-A,B	Branch:	CSE

		Marks Distribution		Max Marks
1 (a)	disadvantage of ordinary queue and how it is overcome in circular queue	[2]	2	10
(b)	Circular queue : addq function	[4]	8	
	deleteq function	[4]		
2	Operations of Dequeue : insert front, insert rear	[5]	10	10
	delete front, delete rear	[5]		
3 (a)	Example for the representation of two polynomials	[2]	4	10
	Example for addition of two polynomials using linked list representation.	[2]		
(b)	C function for addition of two polynomials using linked list	[6]	6	
4 (a)	doubly linked lists: advantages	[2]	4	10
	disadvantages	[2]		
(b)	Write a C function to delete a node from a doubly linked list. "ptr" is the pointer which points to the node to be deleted. Assume that there are nodes on either side of the node to be deleted.	[6]	6	
5	Give the node structure to create a linked list of integers and write a C function to perform the following	[2]	10	10
	i) Create a three node list with data 10,20,30	[2]		
	ii) Insert a node with data value 15 in between the nodes having the data values 10 and 20.	[2]		
	iii) Delete the node which is followed by a node whose data value is 20.	[2]		
	iv) Display the resulting single linked list	[2]		
6	Example of binary tree: array representation	[5]	10	10
	linked representation	[5]		
7	With reference to the fig , answer the following a. Is it a binary tree? b. Is it a complete tree? c. Give the preorder traversal d. Give the inorder traversal e. Give the postorder traversal 	[2] [2] [2] [2] [2]	10	10
8 (a)	Explain threaded binary tree.	[4]	4	10
(b)	Algorithm for inorder, postorder preorder traversal	[2] [2] [2]	6	

1 (a)	Give the disadvantage of ordinary queue and how it is overcome in circular queue

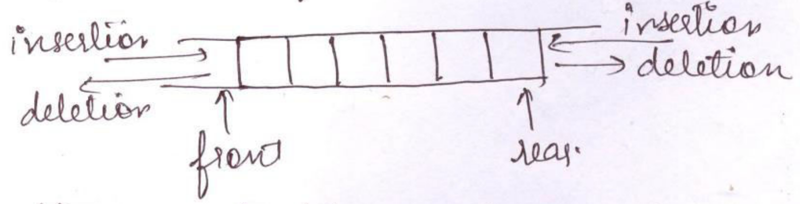
allocated arrays .
The disadvantage of ordinary queue is clearly shown in above
example even though we have some empty location in queue we
are not able to add elements into the queue to overcome
this short coming we will warp around. Flat Queue

(b)	Implement addq and deleteq functions for the circular queue

```
void addq(element item)
{
    if (front == (rear+1)%size)
        printf("queue is full \n");
    else
    {
        rear = (rear+1)%size;
        a[rear] = item;
    }
    if (front == -1) front ++;
}
element deleteq()
{
    element item;
    if (front == -1)
    {
        printf(" queue is empty");
    }
    else
    {
        item = queue [front];
        if (front == rear)
        {
            front = -1;
            rear = -1;
        }
        else
            front = (front +1) % size;
        return item;
    }
}
```

2	Give the implementation of the operations of Dequeue

A queue in which insertion is done at both the ends i.e rear and front end and deletion is also done from both the end i.e from front end and rear end is called Double ended queue.



Operations on double ended queue are:-

Insert at rear end:-

```
void insert(int x)
{  if (rear == size-1)
   queueFull();
   else
   {    queue[++rear]=x;     // increment rear then insert
                                x value in queue.
   }
}
```

Insert at front end:-

```
void insertfront(int x)
{    if (front ==0)
     printf("Insertion is not possible at front end");
```

```
 else
  {
     front--;          // decrement front
     queue[front]=x;   // insert value of x at front.
  }
}
```

Deletion at front end

```
int deletefront()
{
  int x;
  if(front == -1)
  {
    queueEmpty();
  }
  else
  {
     x= queue[front];
    if (front == rear)          // only one element in
    front = rear = -1;             queue.
    else
     {
       front++;             // increment front
     }
    return x;
  }
}
```

Deletion at rear end

```
int deleterear()
{
    if (rear == -1)
   printf("Deletion is not possible at rear end");
   else
    {  x = queue[rear];
      if (rear == front)   // only one element.
else
 {
   x = queue[rear];
   if (rear == front)   // only one element
   rear = front = -1;
   else
     rear --;        // decrement rear
  return x;
 }
}
```

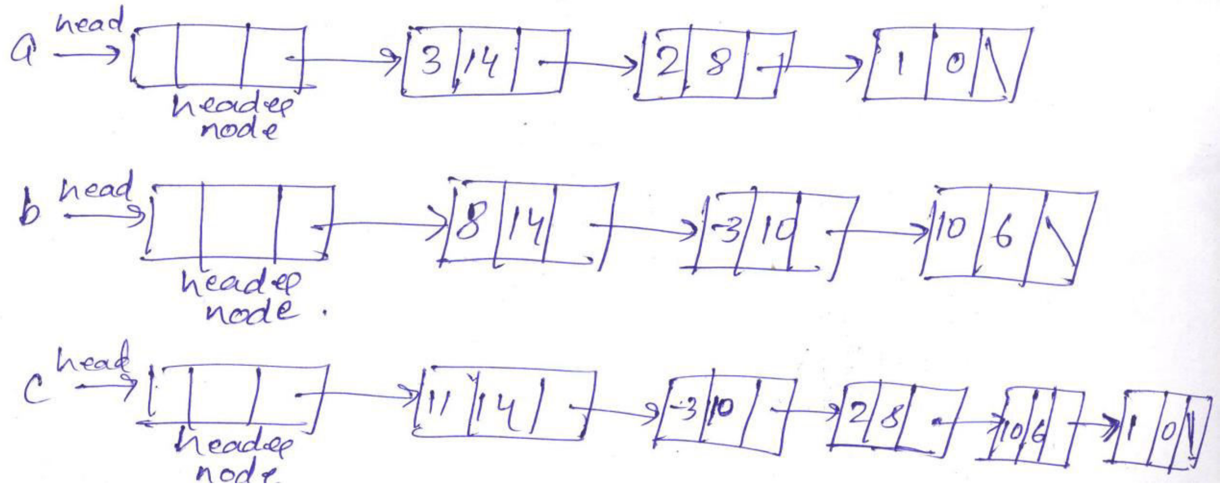
3 (a)	Explain with suitable example the addition of two polynomials using linked list representation.

Example.

$a = 3x^{14} + 2x^{8} + 1$

$b = 8x^{14} - 3x^{10} + 10x^{6}$

$c = 11x^{14} - 3x^{10} + 2x^{8} + 10x^{6} + 1$



(b)	Write a C function for addition of two polynomials using linked list

```
typedef struct node *nptr;
typedef struct
{
    float coef;
    int expo;
    nptr next;
}
nptr addpoly (nptr a, nptr b)
{
      nptr c, rear; int sum;
      while (a && b)
       {
        if (a->expo = = b->expo)
          {
             sum = a -> coef + b -> coef;
             attach (sum, a -> expo, & rear);
             a = a -> next;
             b = b -> next;
          }.
        else if ( a -> expo > b -> expo)
          {
              attach (a -> coef , a -> expo , & rear);
               a = a -> next;
          }
        else
          {
              attach (b -> coef , b -> expo, & rear);
              b = b -> next;
          }
       }
// copy rest of the terms into C
for(; a; a= a->next)
  attach (a-> coef, a-> expo, &rear);
 for(; b; b=b->next)
   attach(b -> coef , b-> expo, & rear);
   rear ->next; //NULL
   return c; // c is addition of 2 polynomials

}

void attach( float c, int e, nptr *ptr)

    {
      nptr new;
     new= (nptr) malloc (size of (struct node));
     if (new= =NULL)
          printf(" out of space");
        else
          {
            new -> coef =c;
             new ->expo =e;
             new -> next =NULL;
              (*ptr) ->next = new;
               ptr =new;
           }
    }
```

4 (a)	Describe the doubly linked lists with advantages and disadvantages.

Doubly linked lists are the lists in which each holds 2 addresses that is :

① It holds the address of the next node.

② It holds the address of the previous node.

Advantages:

① We can traverse in both directions, both from starting to the end and as well as from end to starting. It is easy to reverse the linked list.

② If we are at a node, then we can go to any node. But in linked list it is not possible to read the previous node.

Disadvantages:

① It requires more space because one extra field is required for pointer to previous node.

② Insertion & deletion will take more time than linear linked list because more pointer operation are required than linear linked list.

(b)	Write a C function to delete a node from a doubly linked list. "ptr" is the pointer which points to the node to be deleted. Assume that there are nodes on either side of the node to be deleted.

```
typedef struct node *ptr
typedef struct
{
int data;
ptr next;
```

```
  ptr prev;
 }node;
ptr find previous (ptr h, int n)
 {
 ptr p;
 p = h→ next
 while (p! = NULL)
  {
 if (p→ data = n)
 return p;
 else
 p = p →next;
  }
 }
void delete(ptr h, int n)
{
ptr temp, p;
p = find previous ( h, n);  /* previous node whose data is n*/
temp = p → next
temp → next → prev = temp → prev → next;
temp → next → prev = temp → prev;
temp → ~~next~~ prev → next = temp → next;
free (temp);
}
```

5	Give the node structure to create a linked list of integers and write a C function to perform the following i) Create a three node list with data 10,20,30 ii) Insert a node with data value 15 in between the nodes having the data values 10 and 20. iii) Delete the node which is followed by a node whose data value is 20.

	iv) Display the resulting single linked list

```
#include <stdio.h>
typedef struct node *nptr;
typedef  struct
 {
   int data;
   struct  node * next;
 } node;

nptr createnode()
{
  nptr new= (nptr) malloc( sizeof( struct node));
   if (new == NULL)
   {printf (" Out of space\n"); ~~return~~ exit(0); }
   else  new->next= NULL;
   return new;
 }
```

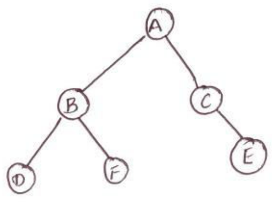
```
void  add(nptr h, int x)
{ nptr last= findlast(h);
  nptr  temp = createnode( );
   temp -> data= x;
    temp -> next = last -> next;
     last -> next  =  temp;
 }
nptr  findlast (nptr h)
 {
   nptr  p= h;
     while ( p -> next != NULL)
     {
      p = p -> next;
     }
     return  p;
  }
 nptr findnode( nptr h, int n)
 {
   nptr   p= h -> next;
    while  ( p -> data != n)
     p= p -> next;
      return p;
   }
void   insert ( nptr  h, int  n, int x)
{
  ~~p = find~~ nptr   p= findnode (h, n);
   nptr new= createnode( );
    new -> data = x;
   new -> next = p -> next;
    p -> next  =  new;
 }
void  delete ( n ptr   h, int n)
{
 nptr  ~~temp~~ p = findnode(h, n);
  nptr  temp = p -> next;
  p -> next = temp -> next;
   free ( temp);
 }
```

```
Void display(nptr h)
{
   nptr p = h->next;
    while (p-> next != NULL)
    {
      printf (" %d -> ", p->data);
    }
    printf (" %d ", p->data);

}

int  main( )
{
  ~~node~~node ~~nptr~~ n1 = createnode();
 ~~n1.~~add (n1, 10);
 ~~n1.~~ add (n1, 20);
  ~~n1.~~add (n1, 30);
  ~~n1.~~ insert (n1, 10 , 15);
    ~~n1.~~ delete (n1, 20);
    printf (" The linked list is\n");
      display(n1);
    return 0;
 }
```

6	With an example show array representation and linked representation of binary tree.

Binary tree is a tree in which each node (parent) has maximum of 2 subnodes (children) ie max degree 2.

Ex :-



Array Representation :-

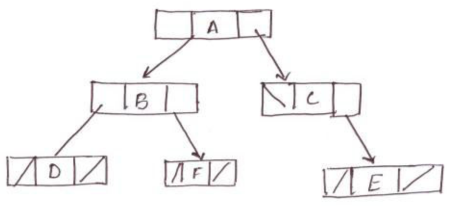
Parent - i

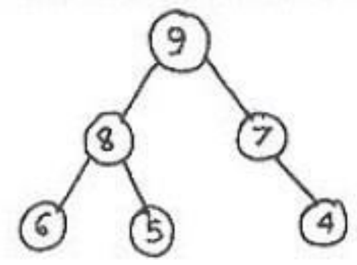
left child - $2 * i$

Right child - $(2 * i) + 1$.

0	1	2	3	4	5	6	7	8
	A	B	C	D	F		E	

Linked list representation :-

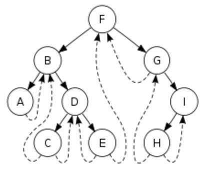


7	With reference to the fig , answer the following a. Is it a binary tree? b. Is it a complete tree? c. Give the preorder traversal d. Give the inorder traversal e. Give the postorder traversal 

a. Yes
b. No
c. 9,8,6,5,7,4
d. 6,8,5,9,7,4
e. 6,5,8,4,7,9

8 (a)	Explain threaded binary tree.

a **threaded binary tree** is a binary tree variant that allows fast traversal: given a pointer to a node in a threaded tree, it is possible to cheaply find its in-order successor (and/or predecessor).



Algorithm traverse(*t*):

- Input: a pointer *t* to a node (or nil)
- If *t* = nil, return.
- Else:
    - traverse(left-child(*t*))
    - Visit *t*
    - traverse(right-child(*t*))

(b)	Give the algorithm for inorder, postorder and preorder traversal

```
Algorithm Inorder(tree)
   1. Traverse the left subtree, i.e., call Inorder(left-subtree)
   2. Visit the root.
   3. Traverse the right subtree, i.e., call Inorder(right-subtree)
Algorithm Preorder(tree)
   1. Visit the root.
   2. Traverse the left subtree, i.e., call Preorder(left-subtree)
   3. Traverse the right subtree, i.e., call Preorder(right-subtree)
Algorithm Postorder(tree)
   1. Traverse the left subtree, i.e., call Postorder(left-subtree)
   2. Traverse the right subtree, i.e., call Postorder(right-subtree)
   3. Visit the root.
```