MAY 2015

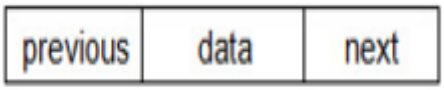
Q1 a.
State difference between singly and doubly link list (5)

difference between Singly Linked List and Doubly Linked List

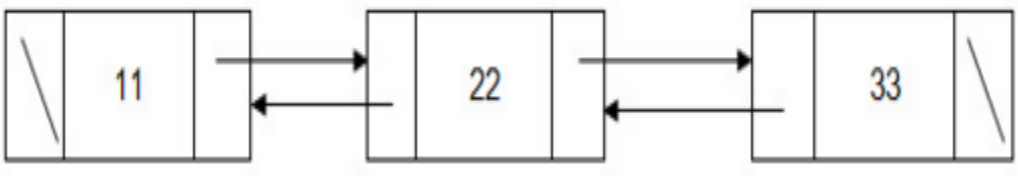
Each element in the singly linked list contains a pointer to the next element in the list, while each element in the doubly linked list contains pointer to the next element as well as the previous element in the list.

Doubly linked lists require more space for each element in the list and elementary operations such as insertion and deletion is more complex since they have to deal with two pointers. But doubly link lists allow easier manipulation since it allows traversing the list in forward and backward directions. Whereas singly link list can be traversed on in one direction.

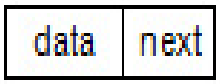
Structure of doubly link list.



Example of doubly link list



Structure of singly link list



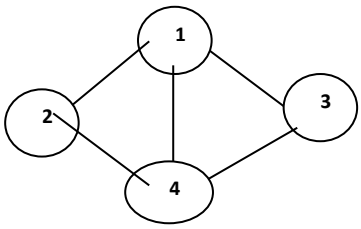
Example of singly link list

Q1b

What is Graph. What are methods to represent Graph (5)

Graph is a data structure which is collection of vertices and edges. Each edge connects two vertices of a graph.

Example graph is shown below



This Graph named G can be defined as follows

G = (V , E)
V = {1,2,3,4}
E = {(1,2) , (1,3) , (1,4) , (2,4) , (3,4)}

Representation of graph G in computer memory

a) Adjacency matrix method

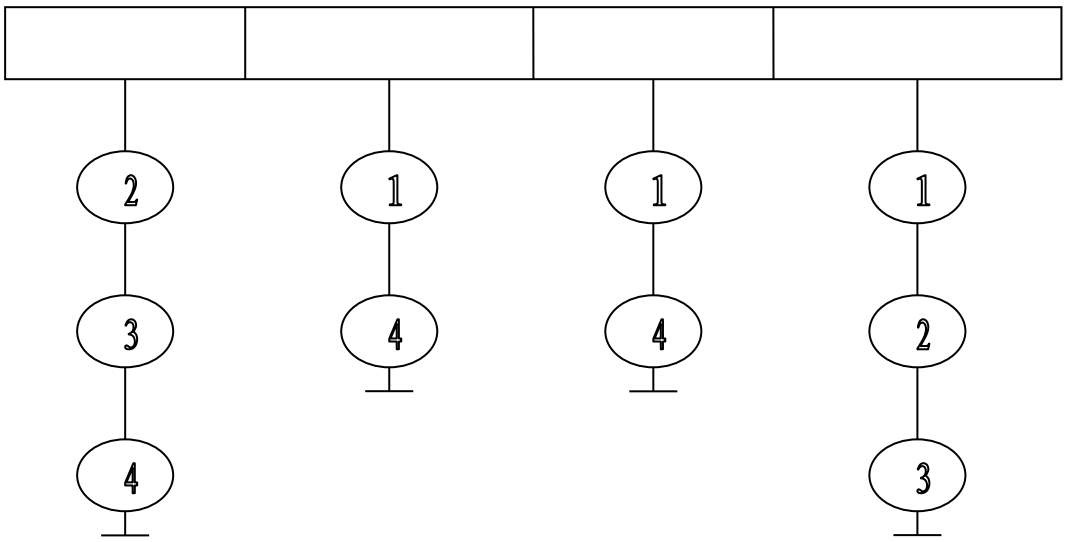
In this method a matrix (2 dimensional array) is declared of size V x V (V = number of vertices of the graph). The entry (i , j) is set to 1 if there is edge from vertex i to vertex j. Otherwise entry (i , j) is set to zero. The adjacency matrix G for the above graph is shown below.

Matrix G

	1	2	3	4
1	0	1	1	1
2	1	0	0	1
3	1	0	0	1
4	1	1	1	0

b) Adjacency List method

In this method a link list is used to store all adjacent vertices of a particular vertex. For a V vertex graph there will be V different link list. adjacency list for the above graph is shown below



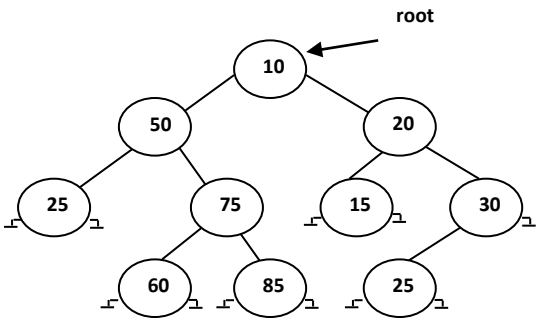
Q1 c

What is binary search tree. Explain with example (5)

It is a data structure which is collection of nodes. Each node has some data stored in it and each node has two references namely left and right. Left reference refers to the left child node and right reference refers to the right child node. Value stored in the left child node is always less than or equal to value of its parent node. Value stored in the right child node is always greater than value of its parent node.

Consider the following data which is set of numbers and this data is inserted in a binary search tree.

Data : 100 50 200 300 250 25 75 60 150 85



Q1 d

What is data structure. List out the areas in which data structures are used extensively.

Left as an exercise.

Q2 a

WAP to implement Quick sort (8)

```
//program for quick sort

int partition(int a[], int left , int right)
{
  int i,j,x,t;
  i = left;
  j = right;
  x = a[left];
```

```
  while (i < j)
  {
    while (i <= right && a[i] <= x)
      i++;

    while (a[j] > x)
      j--;

    if (i<j)
    {
      t = a[i];
      a[i] =a[j];
      a[j] = t;
    }
  } //end while

  //swap a[j] and a[left]
  t = a[j];
  a[j] = a[left];
  a[left] = t;

  return j;

} //end of partition


void quicksort(int a[] , int left , int right)
{
  int p;

  if (left < right)
  {
    p = partition(a, left , right);
    quicksort(a,left,p - 1);
    quicksort(a, p+1, right);
  }
}


void main()
{
  int a[100] , n , i;
```

```
 printf("Enter number of elements ");
 scanf("%d" , &n);

 //scan array from 0 to n-1 locations
 for (i = 0 ; i <= n-1 ; i++)
 {
   printf("Enter element %d " , (i+1));
   scanf("%d" , &a[i]);
 }

 //sort the array

 quicksort(a,0,n-1);

 //print the sorted array
 for (i=0 ; i <= n-1 ; i++)
   printf("%d " , a[i]);

} //end of main
```

Q2 b

Define traversal of binary tree. Explain different traversals with examples. (6)

Traversal means to visit each node of binary tree and print the data or do some operation on that node. Traversal of a tree is nothing but travelling the entire tree node by node.

3 types of traversals are

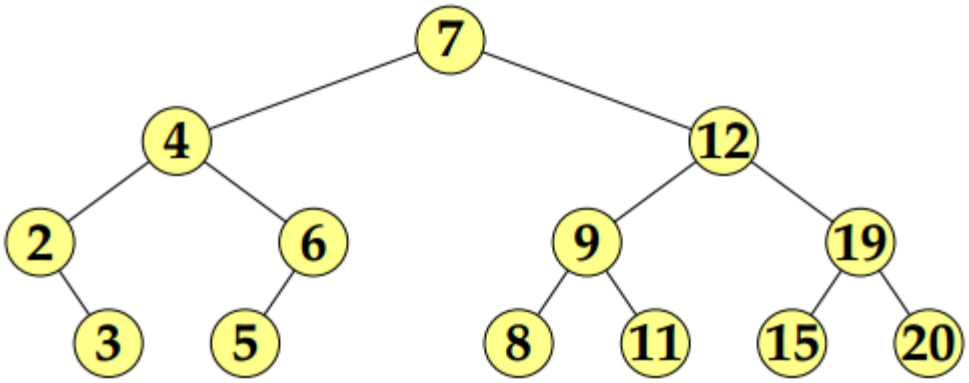
a) Inorder : visit the nodes in (L , V , R) method.
   i) visit left child node (L)
   ii) then visit vertex (V)
   iii) finally visit right child node (R)

b) Preorder : visit the nodes in (V , L , R) method.
   i) visit vertex first (V)
   ii) then visit left child (L)
   iii) finally visit right child node (R)

c) Postorder : visit the nodes in (L , R , V) method.
   i) visit left child first (L)

ii) visit right child node (R)
iii) then visit vertex (V)

Find out inorder , preorder and postorder for the following tree.



Inorder traversal gives: 2, 3, 4, 5, 6, 7, 8 , 9, 11, 12, 15, 19, 20.
Preorder traversal gives: 7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20.
Postorder traversal gives: 3, 2, 5, 6, 4, 8, 11, 9, 15, 20, 19, 12, 7.

Q2 c
Explain Infix , Prefix and Postfix expression with examples (6)
Any expression can be stated in 3 ways
a) infix : a / b (Operator is written between the 2 operands)
b) prefix : /ab (Operator is written before the 2 operands)
c) postfix : ab/ (Operator is written after the 2 operands)

Any infix expression can be translated into postfix or prefix and viceversa

Ex 1 : Convert following infix expression in postfix and prefix

The given infix expression is **infix : a + b * c**

postfix : a + [bc*]
abc*+

prefix : a + [*bc]

+a*bc

Ex 2 : Convert following infix expression to postfix and prefix
infix : a + b * c / d – e
postfix : a + [bc*] / d – e
a + [bc*d/] – e
[abc*d/+] – e
abc*d/+e-

prefix : a + [*bc] / d – e
a + [/*bcd] – e
[+a/*bcd] – e
-+a/*bcde

Q3 a
What is circular queue. WAP to implement circular queue (10)
Circular queue is a linear data structure. It follows FIFO principle. In **circular queue** the last node is connected back to the first node to make a **circle**. **Circular** linked list fallow the First In First Out principle. Elements are added at the rear end and the elements are deleted at front end of the **queue**.

Circular Queue

Circular queue can be implemented in 2 ways

i) Circular Link list : the pointer of the last node points to first node and hence circular queue is formed.

ii) Using array : we can use array which will hold data and two counters front and rear which are initially -1. Every insertion and deletion increments rear and front respectively.

Program for circular queue using array

```
#include <stdio.h>
#define SIZE 5
struct queue
{
 int a[SIZE];
 int front , rear;
};

 void insert(struct queue *p, int x)
 {
  int pos;

  pos = (p->rear + 1 ) % SIZE;
  if (pos == p->front)
   printf("queue overflow\n");
  else
  {
  p->rear = pos;
  p->a[p->rear] = x;


  if (p->front == -1)
   p->front=0;
  }
 }


int empty(struct queue *p)
{
 if(p->front == -1)
  return 1;
 else
  return 0;
}

int delete(struct queue *p)
 {
  int x;
```

```
  x = p->a[p->front];
  if (p->front == p->rear)
    p->front = p->rear = -1;
  else
    p->front = (p->front + 1) % SIZE;

  return x;

 }

 void display(struct queue *p)
 {
  int i;
  if (p->front == -1)
    printf("queue underflow\n");
  else
  {
  printf("Elements of queue are\n");

  i = p->front;
  while (i != p->rear)
  {
    printf("%d\n" , p->a[i]);
    i = (i + 1) % SIZE;
  }
  printf("%d\n" , p->a[p->rear]);
 }
}

 void destroy(struct queue *p)
 {
  p->front = p->rear = -1;
 }


void main()
{
  int ch , x , z;
  struct queue q;
  q.front = q.rear = -1;
```

```
 do
 {
 printf("Menu\n1.insert\n2.delete\n3.display\n4.destroy\n5.exit\n");
 printf("enter choice ");
 scanf("%d" , &ch);

 switch(ch)
 {
 case 1 : printf("Enter integer to insert ");
        scanf("%d" , &x);
        insert(&q , x);
          break;
 case 2 : if (empty(&q))
         printf("queue underflow\n");
        else
        {
          z = delete(&q);
          printf("Data deleted = %d \n" , z);
        }
          break;
 case 3 : display(&q);
          break;
 case 4 : destroy(&q);
        break;
 case 5 : break;
 default : printf("wrong choice\n");
 }
}
while(ch != 5);


} //end main
```

Q3 b

Explain linear and non linear data structures with examples (5)

A data structure is a method for organizing and storing data, which would allow efficient data retrieval and usage. Linear data structure is a structure that organizes its data elements one after the other. Nonlinear data structures are constructed by attaching a data element to several other data elements in such a way that it reflects a specific relationship among them.

Linear data structures

Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Data elements in a linear data structure are traversed one after the other and only one element can be directly reached while traversing. Some commonly used linear data structures are arrays, linked lists, stacks and queues. An arrays is a collection of data elements where each element could be identified using an index. A linked list is a sequence of nodes, where each node is made up of a data element and a reference to the next node in the sequence. A stack is actually a link list where data elements can only be added or removed from the top of the list. A queue is also a link list, where data elements can be added from one end of the list and removed from the other end of the list.

Application of linear data structure :

a) Linear data structures like stacks are used for handling recursive functions. In fact any recursive function uses a stack to store return address of the function.
b) Linear data structure like link list is used to store data more efficiently in the memory in such a way that link list will use optimum memory. Link list can grow in size if more data is added and link list can decrease in size if data is deleted.
c) Queues are used by applications where data has to be processed in FIFO manner. Such as print queue of the printer will print the list of documents in First Come First Serve basis.

Nonlinear data structures

In nonlinear data structures, data elements are not organized in a sequential fashion. A data item in a nonlinear data structure could be attached to several other data elements to reflect a special relationship among them and all the data items cannot be traversed in a single run. Data structures like multidimensional arrays, trees and graphs are some

examples of nonlinear data structures. A multidimensional array is simply a collection of one-dimensional arrays. A tree is a data structure that is made up of a set of linked nodes, which can be used to represent a hierarchical relationship among data elements. A graph is a data structure that is made up of a finite set of edges and vertices. Edges represent connections or relationships among vertices that stores data elements.

Applications of Non Linear Data Structures

a) Binary Search Trees are used for searching a data in much faster and efficient way.
b) Expression trees are used for storing mathematical expressions so that such expressions can be converted into postfix , prefix and infix expression.
c) Graphs can be used to model systems like computer networks or electrical networks.

Q3 c
Explain the term recursion with examples (5)

Recursion is a phenomenon in which a function calls itself. A function which calls itself is called recursive function. Eg. factorial function can call itself recursively to find factorial of integer n

```
long fact(int n)
{
  if (n == 0)
    return 1;
  else
    return n * fact(n-1);
}
```

The statement

```
    return n * fact(n-1);
```

contains a recursive call "fact(n-1)".

Hence fact(n) calls fact(n-1).
Every recursive function has base case. The recursion stops when the base case becomes true. In this example base case is

```
if (n == 0)
 return 1;
```

Suppose we call the fact function as “fact(5)” then the recursive calls can be shown in stack as follows

fact(5) will return 5 * fact(4)
fact(4) will return 4 * fact(3)
fact(3) will return 3 * fact(2)
fact(2) will return 2 * fact(1)
fact(1) will return 1 * fact(0)
fact(0) will return 1

Q4 a
WAP to convert infix to postfix expression. (10)

```
//program to convert infix expression to postfix expression
#include <stdio.h>
#define SIZE 50
//define a stack
struct stack
{
  char a[SIZE];
  int top;
};


  void push(struct stack *p , char x)
  {

    if (p->top ==  SIZE-1)
      printf("stack overflow\n");
    else
    {
    p->top++;
    p->a[p->top] = x;
    }
  }


  char pop(struct stack *p)
  {
    char x;
```

```
  x = p->a[p->top];
  p->top--;
  return x;
}


char stacktop(struct stack *p)
{
  return p->a[p->top];
}

int isp(char ch)
{
 int priority=0;
 switch(ch)
 {

   case '*' : priority = 4; break;
   case '/' : priority = 4; break;
   case '+' : priority = 3; break;
   case '-' : priority = 3; break;
   case '(' : priority = 2; break;
   case '~' : priority = 1; break;
  }
 return priority;
}

int icp(char ch)
{
 int priority=0;
 switch(ch)
 {

   case '*' : priority = 4; break;
   case '/' : priority = 4; break;
   case '+' : priority = 3; break;
   case '-' : priority = 3; break;
  }
 return priority;
}
```

```
void convertinfixtopostfix(char infix[] , char postfix[])
{
      char ch , t;
      int len , i;
      int j = 0;
      struct stack s;

      len = strlen(infix);
      s.top = -1;

      push(&s , '~');
      for (i = 0 ; i <= len-1 ; i++)
      {
        ch = infix[i];

        if (ch == '(')
             push(&s , ch);
        else
        if (ch == ')')
        {
           while (stacktop(&s) != '(')
           {
              t = pop(&s);
              postfix[j] = t;
              j++;
            }

            pop(&s);
         }
         else
         if (ch == '*' || ch == '-' || ch == '/' || ch == '+')
         {
             while (icp(ch) <= isp(stacktop(&s)))
             {
               t = pop(&s);
               postfix[j] = t;
               j++;
              }

              push(&s,ch);
         }
         else
```

```
        {
           postfix[j] = ch;
           j++;
        }

      } //end for

      //pop remaining
      while (stacktop(&s) != '~')
      {
        t = pop(&s);
        postfix[j] = t;
        j++;
      }
      postfix[j] = '\0';
  }


 void main()
 {
   char infix[50] , postfix[50] ;
   printf("Enter the infix expression\n");
   printf("The expression should be of the form 2+3*4\n");

   gets(infix);

   convertinfixtopostfix(infix,postfix);

   printf("given infix = %s\n" , infix);
   printf("Resultant postfix = %s\n" , postfix);
  }
```

Q 4 b

What is AVL tree. Construct AVL tree for following data. (10)

50 25 10 5 7 3 30 20 8 15

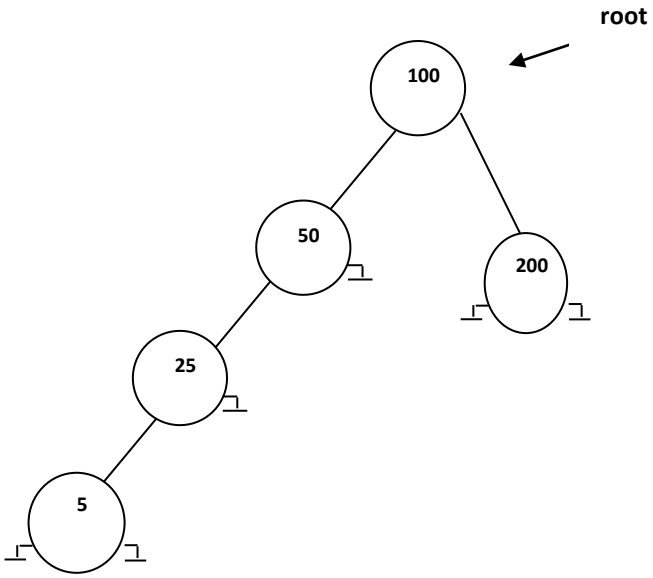
AVL tree stands for Adelson Velskii and Landis tree.

AVL tree is defined as a balanced binary search tree in which every node must satisfy the following criteria

Height of left subtree - Height of right subtree <= 1

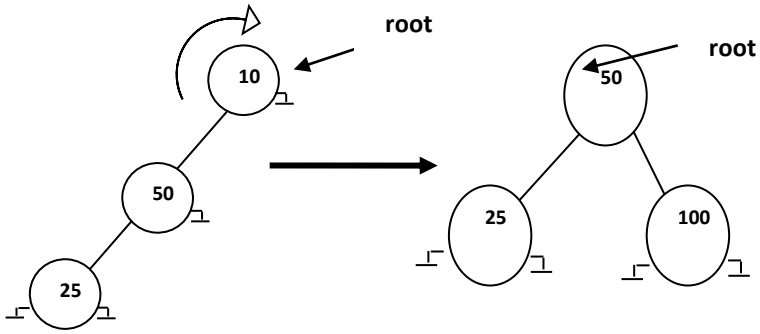
If any node does not satisfy this criteria then the entire tree said to be not balanced.

For example the following tree is not balanced because 50 has height of left subtree = 2 and height of right subtree = 0. The difference comes out to be 2 which proves that node storing 50 is unbalanced.



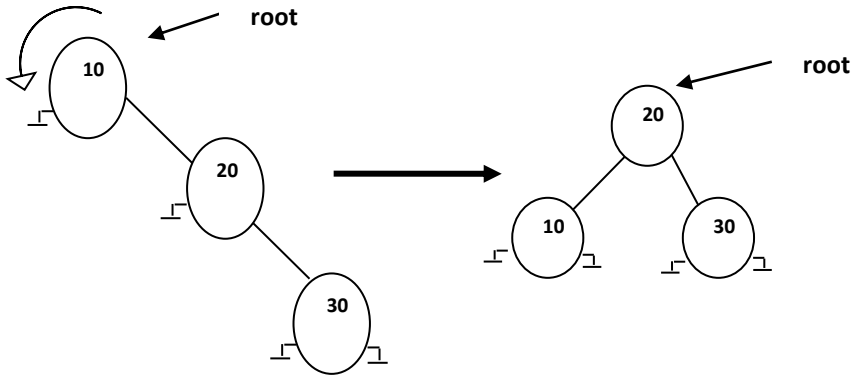
AVL gives 4 rotation algorithms which can transform an unbalanced tree into a balanced tree. These 4 rotation algorithms can be explained by using 4 cases.

Case I : Left – Left case



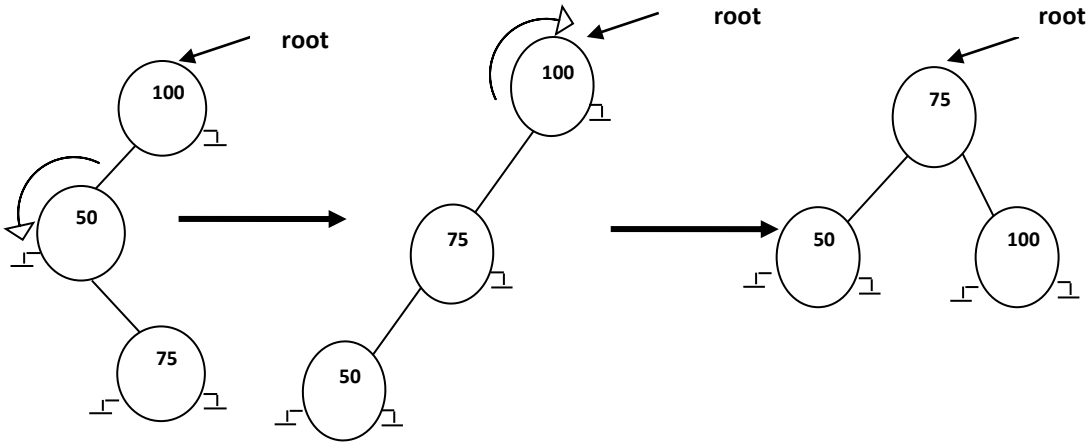
As seen from the above diagram 100 is not balanced due to left-left case. Hence 100 is rotated to the right of 50. The resultant tree shown is clearly balanced.

Case II : Right – Right case



As seen from the above diagram 100 is not balanced due to right-right case. Hence 100 is rotated to the right of 200. The resultant tree shown is clearly balanced.

Case III : Left-Right case

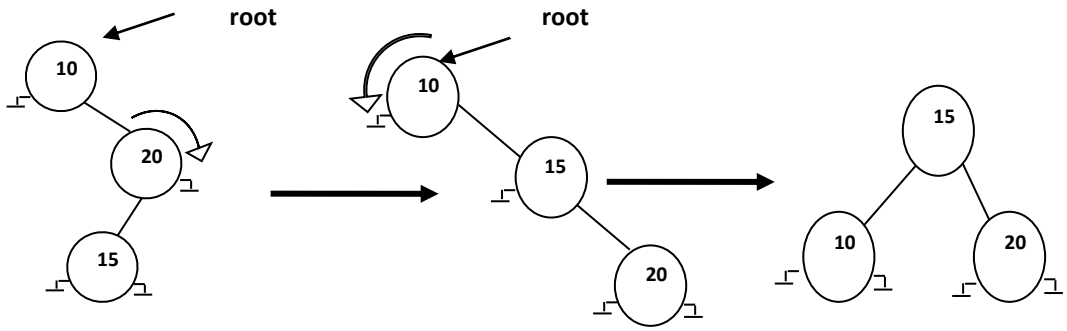


As seen from the above diagram 100 is not balanced due to left-right case. To solve this case of unbalancing we need a double rotation. First we rotate 50 to left of 75(that is 50 is made left child of 75). Next we rotate 100 to the right of 75. The resultant tree is clearly balanced.

Case IV : Right-Left case

As seen from the above diagram 100 is not balanced due to right-left case. To solve this case of unbalancing we need a double rotation. First we rotate 200 to right of 150(that is

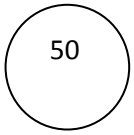
200 is made right child of 150). Next we rotate 100 to the left of 150. The resultant tree is clearly balanced.



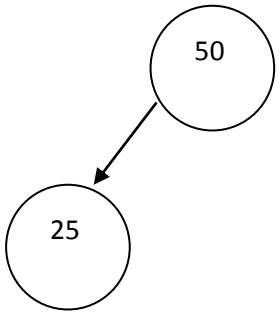
The given data is

50 25 10 5 7 3 30 20 8 15

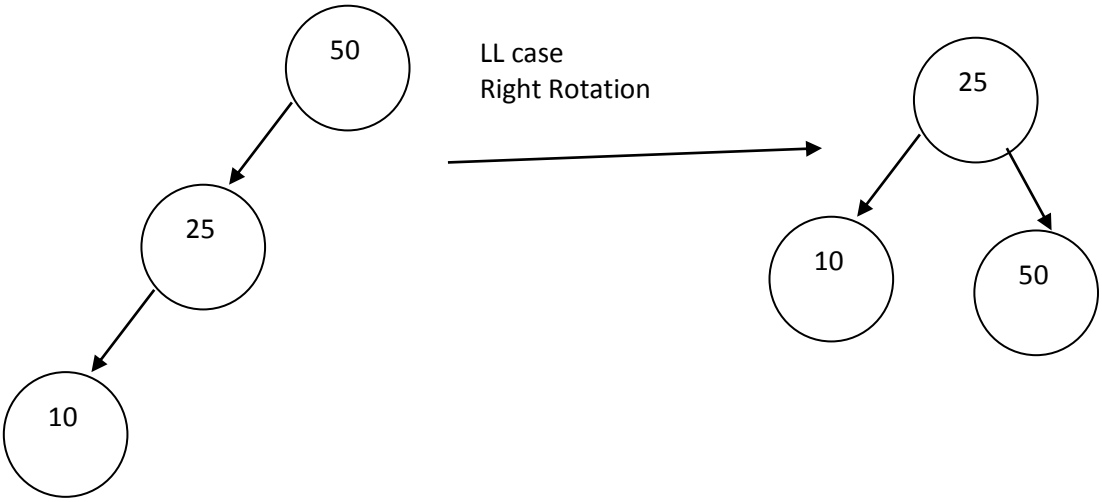
i) Insert 50



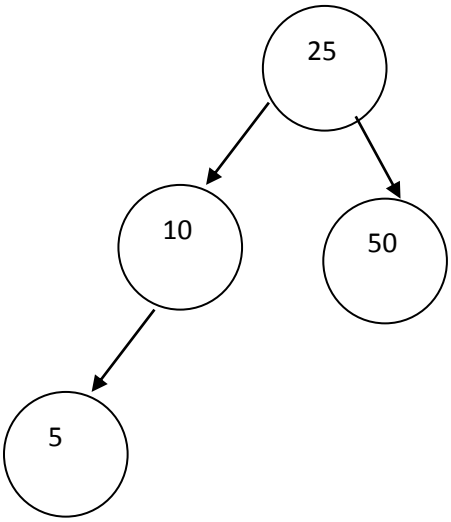
ii) Insert 25



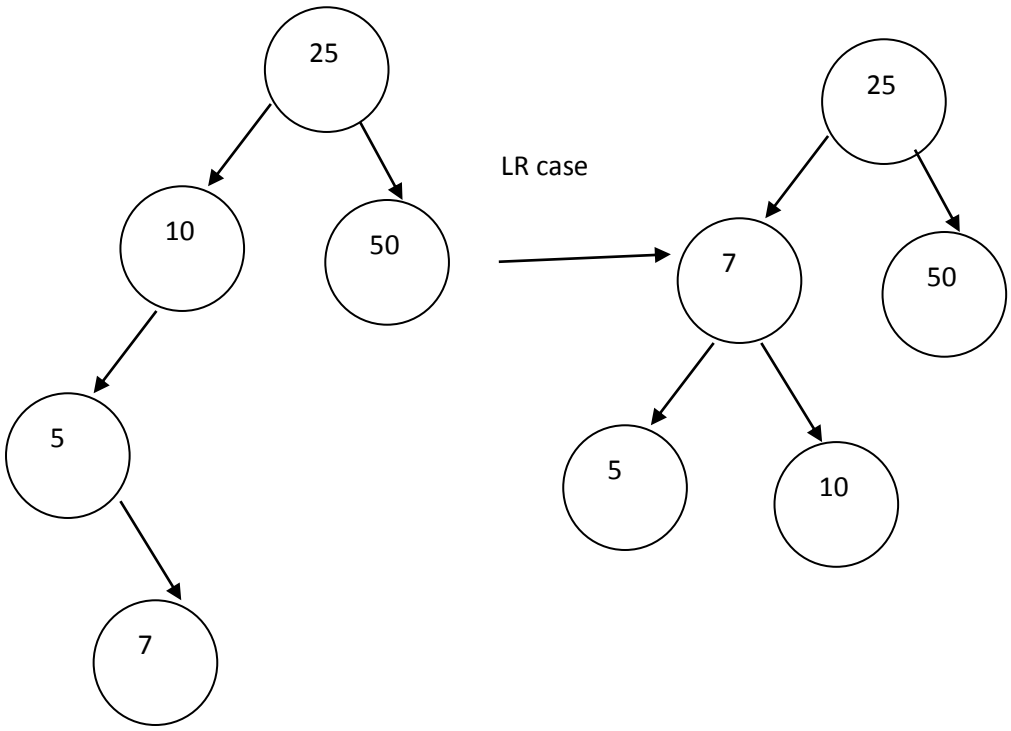
iii) Insert 10



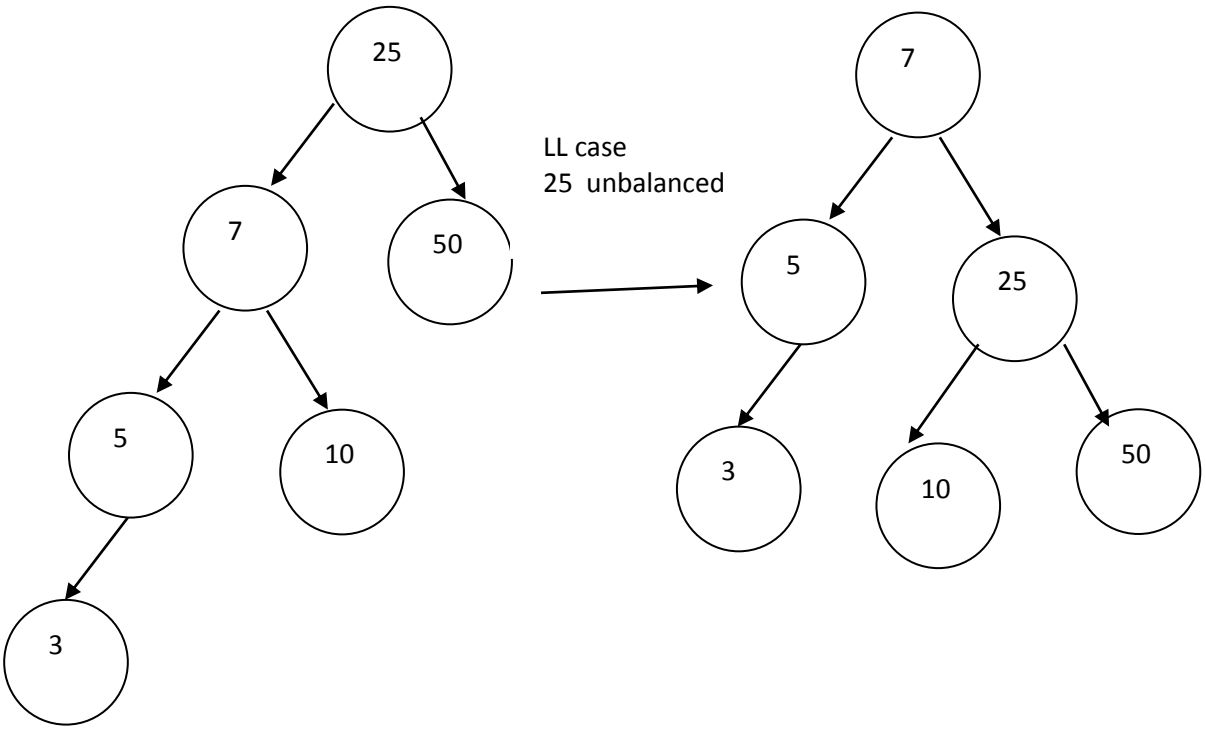
iv) Insert 5



v) Insert 7



vi) Insert 3



Remaining question is left as an exercise

Q6 b
Explain Huffman algorithm with example (5)

1) Data is stored in IBM PC in ASCII format(American standard code for information interchange). The size of the ASCII code is 8 bit or 1 byte. For example letter A in ASCII format is 01000001(which is 65). Hence every character in stored in the memory using 8 bit code. Note that every character has constant length code that is 8 bit code.

2) Data compression can be achieved if we can store characters using variable length code. That is every character will not be coded in fixed length of 8 bits. But the length of the code will be dependent on frequency of the character. Shorter length code will be assigned to characters having larger frequency.

3) Consider an example file which has following characters and their frequency

Character	Frequency
A	50
B	35
C	15
D	10
E	5
F	1

The characters in the file are listed in decreasing order of their frequency.

The Huffman's algorithm for finding variable length codes is as follows
Comments
Given 'n' characters and their frequencies. Each character and frequency is stored in the node of a binary tree. Hence 'n' different binary trees are created each tree having only one node.

Algorithm Huffman
{
Create a forest of 'n' trees each tree has only 1 node. The node stores character and frequency

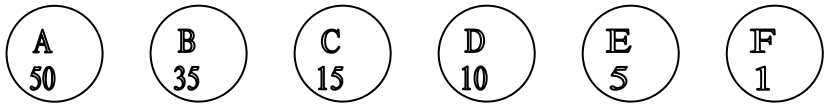
while (number of trees in forest is not equal to 1)

```
    {
        sort all characters in decreasing order of frequency
        take 2 least frequencies f1  and  f2
        create a new node  newrec such that
            newrec.frequency = f1 + f2
            newrec.left = f1
            newrec.right = f2
    }
} // end algorithm
```

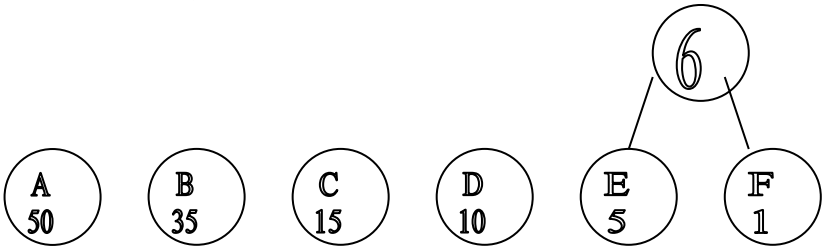
The algorithm creates one single tree. label all left branches as 0 and right branches as 1. These labels will give codes for every character.

The demonstration of the algorithm is done on the above table of characters and frequencies.

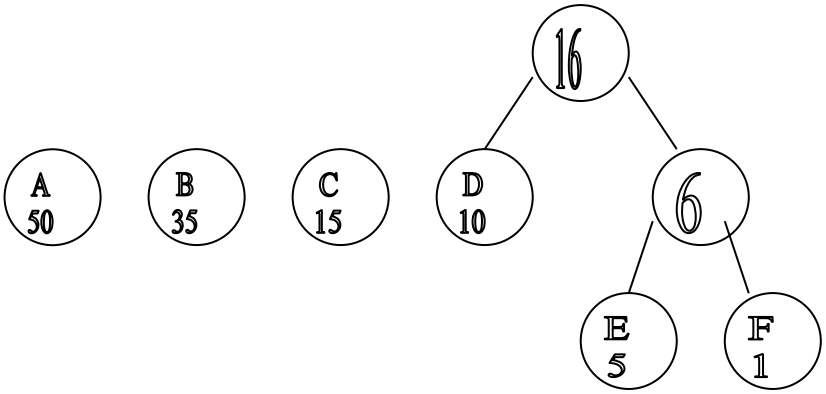
1) Create forest of 6 trees



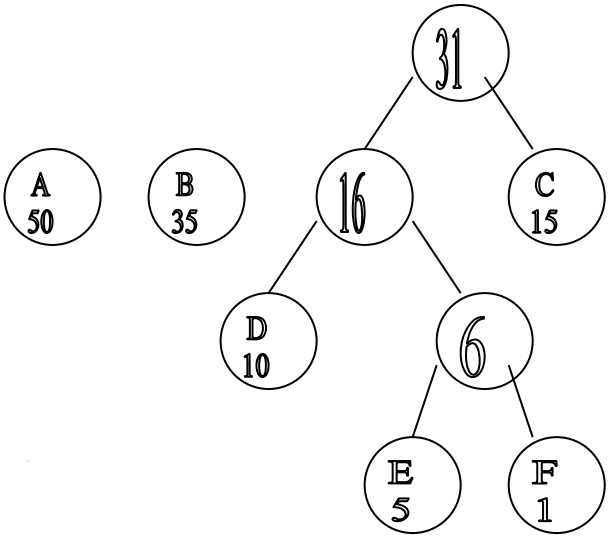
2) Take 2 least frequencies E,5 and F,1.



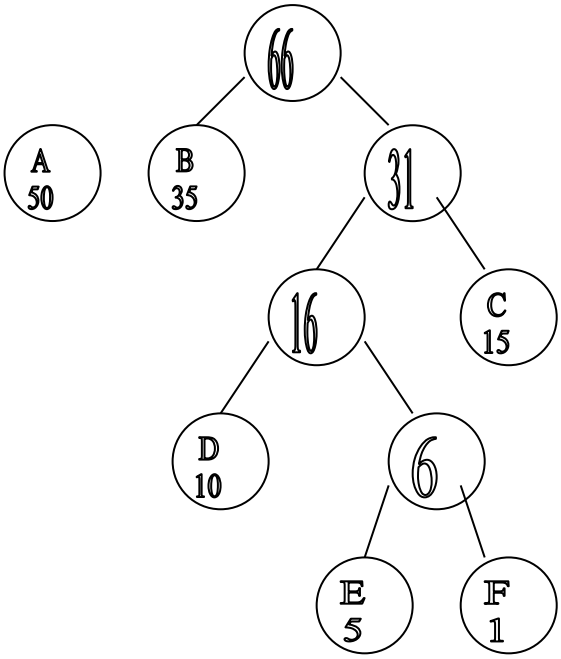
3) Arrange all frequencies in decreasing order and take 2 least frequencies D,10 and 6



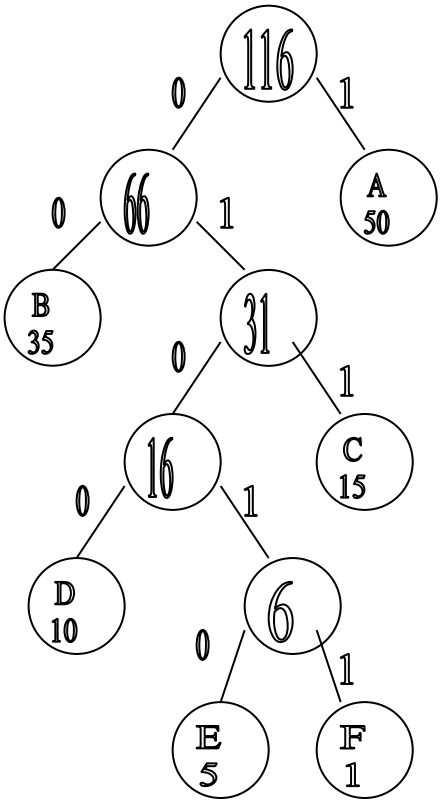
4) Arrange all frequencies in decreasing order and take 2 least frequencies 16 and C,15



4) Arrange all frequencies in decreasing order and take 2 least frequencies B,35 and 31



4) Arrange all frequencies in decreasing order and take 2 least frequencies 66 and A,50



The characters and their Huffman codes are now listed as follows

Character	Frequency	Huffman code
A	50	1
B	35	00
C	15	011
D	10	0100
E	5	01010
F	1	01011

Q6 c
What is a file? Explain various file handling operation in C. (5)

A file represents a sequence of bytes on the disk where a group of related data is stored. File is created for permanent storage of data. It is a readymade structure. In C language, we use a structure pointer of file type to declare a file. FILE *fp;

Opening a File or Creating a File The fopen() function is used to create a new file or to open an existing file.
General Syntax : *fp = FILE *fopen(const char *filename, const char *mode);

Here filename is the name of the file to be opened and mode specifies the purpose of opening the file. Mode can be of following types, *fp is the FILE pointer (FILE *fp), which will hold the reference to the opened (or created) file.

mode	description
r	opens a text file in reading mode
w	opens or create a text file in writing mode.
a	opens a text file in append mode

Closing a File The fclose() function is used to close an already opened file.

fgetc() is a function to read single character from file.
fputc()is a function to write a single character to the file.

Following program shows how to read contents of a text file and display all the contents on the screen.

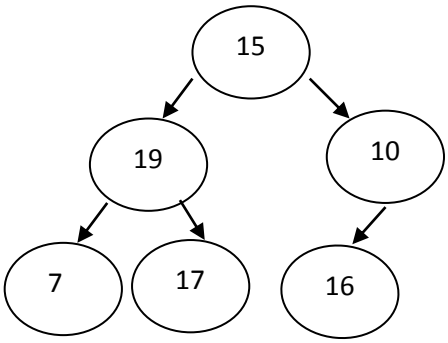
```
#include <stdio.h>
void main()
{
File *fp;
char ch;
fp = fopen("abc.txt" , "r");

while (1)
{
  ch = fgetc(fp);
  if ( ch == EOF)
    break;
 else
   printf("%c" , ch);
}

fclose(fp);
}
```

Q6 a
What is heap. Sort these numbers using Heap sort
15 19 10 7 17 16

Heap : Heap is defined as a sequential binary tree in which all values are added from left to right. Any array can be represented in the form of a Heap. Consider an array of 6 elements and it's heap representation.



Heap always satifies following 2 criteria
a) for any parent p, left child = p*2 , right child = p*2+1
b) for any child c , parent p = c/2

Max Heap : It is a sequential binary tree in which every parent node has a value greater than the value of both the children.

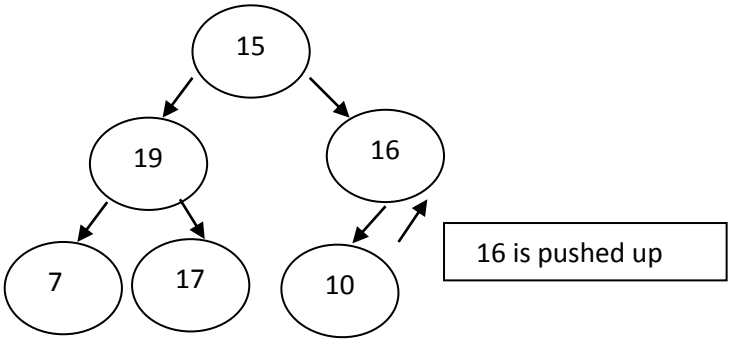
Min Heap : It is a sequential binary tree in which every parent node has a value less than the value of both the children.

Process of Heap Sort
Heap sort works in 2 stages.
Stage 1 of heap sort
Convert the array into max heap. That is every parent must be larger than its children. So for the above array following conditions must satisfy.

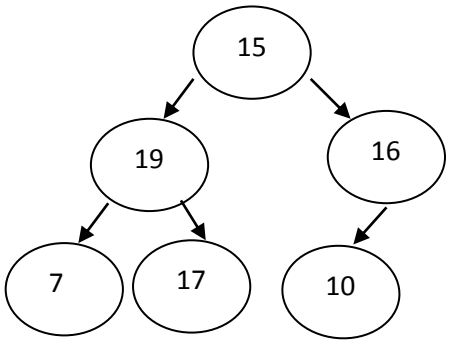
a[1] > a[2], a[3]
a[2] > a[4], a[5]
a[3] > a[6]

These conditions can be met by adjusting value of all parents starting with last parent first.

Adjust values of parent 3
Let x = a[3] = 10

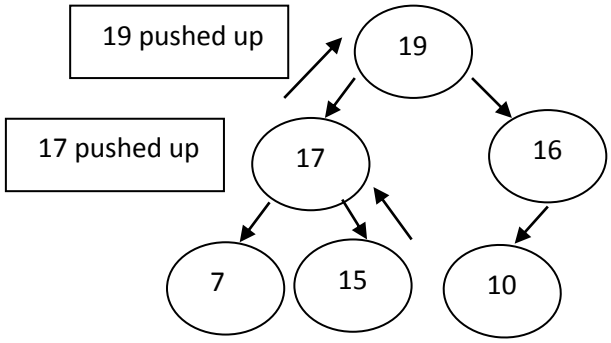


Adjust values of parent 2
Let x = a[2] = 19



But a[2] = 19 is already greater than both the children 7 & 17.

Adjust values of parent 1
Let x = a[1] = 15



Now the array is in the form of Max heap.
Array a

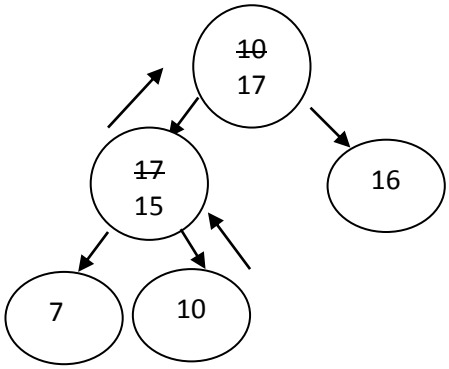
19	17	16	7	15	10
1	2	3	4	5	6

Stage 2 of heap sort
The max heap has first element the largest element. So we swap a[1] and a[6]. this makes a[6] the largest element but disturbs the max heap. So the heap is reconstructed by adjusting value of a[1].

Array after swapping a[1] , a[6]
Array a

10	17	16	7	15	19
1	2	3	4	5	6

Reconstruction of heap by adjusting a[1]



Now the array is

Array a

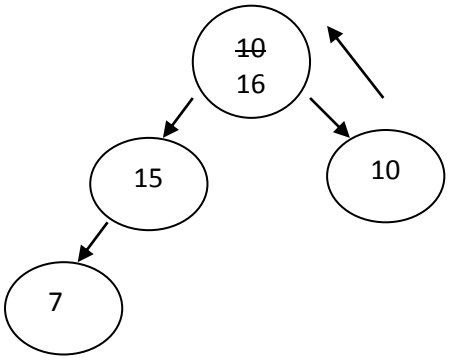
17	15	16	7	10	19
1	2	3	4	5	6

Swap a[1] and a[5] which gives following array

Array a

10	15	16	7	17	19
1	2	3	4	5	6

Reconstruction of heap by adjusting a[1]



Now the array is

Array a

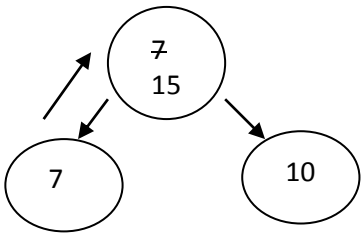
16	15	10	7	17	19
1	2	3	4	5	6

swap a[1] and a[4] which gives following array

Array a

7	15	10	16	17	19
1	2	3	4	5	6

Reconstruction of heap by adjusting a[1]



Now the array is

Array a

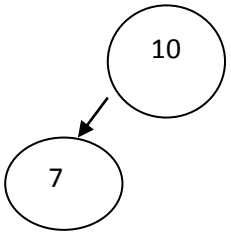
15	7	10	16	17	19
1	2	3	4	5	6

swap a[1] and a[3] which gives array as

Array a

10	7	15	16	17	19
1	2	3	4	5	6

Reconstruction of heap by adjusting a[1]



This is already in the form of max heap because parent (10) is greater than child (7)

Finally swap a[1] with a[2]

Array a

7	10	15	16	17	19
1	2	3	4	5	6

This is sorted array.

Q5 a

WAP to create doubly link list. The program must perform following operations (10)

1. Insert at beginning
2. Insert at a position
3. Delete at beginning
4. Delete at a position.

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

struct DLL
{
struct node *first , *last;
};

void insertbegin(struct DLL *p , int x)
{
 struct node *newrec;
 newrec = (struct node *) malloc (sizeof(struct node));
 newrec->data = x;
 newrec->prev = NULL;
 newrec->next = p->first;
 p->first->prev = newrec;
 p->first = newrec;
}

void insertatpos(struct DLL *p , int x , int pos)
{
 struct node *newrec, *a, *b;
 int i;
 newrec = (struct node *) malloc (sizeof(struct node));
 newrec->data = x;
 a = p->first;
 for (i = 1 ; i <= pos-1 ; i++)
```

```
   a = a->next;

  b = a->prev;

  b->next = newrec;
  newrec->prev= b;
  newrec->next = a;
  a->prev = newrec;

}

void deletebegin(struct DLL *p)
{
   struct node *a;
   a = p->first;
   p->first = a->next;
   p->first->prev = NULL;
   free(a);
}

void deleteatpos(struct DLL *p , int pos)
{
   struct node *a,*b;
   int i;
   a = p->first;
  for (i = 1 ; i <= pos-1 ; i++)
   a = a->next;

  b = a->prev;

   a->next->prev= b;
   b->next = a->next;
   free(a);
}


void display(struct DLL *p)
{
struct node *a;
a = p->first;
```

```
while (a != NULL)
{
  printf("%d " , a->data);
  a = a->next;
}
}

void main()
{
  struct DLL d;
  d.first = d.last = NULL;
  insertbegin(&d, 10);
  insertbegin(&d, 5);
  insertbegin(&d, 2);
  insertatpos(&d, 7 , 3);
  printf("after inserting 10 5 2 at begin and 7 at posn 3 \n\n");
  display(&d);

  deletebegin(&d);
  deleteatpos(&d , 2);
  printf("after deleting first node and node at posn 2\n\n");
  display(&d);
}
```

Q5 b

What is indexed sequential search. WAP to implement indexed sequential search. (10)

Indexed sequential search is modification of sequential search.
The data to be searched must be sorted in increasing order.
Suppose we have sorted array of 10 elements

10 20 30 40 50 60 70 80 90 100

To search this array we create 2 more arrays which act as index tables. Let the names be kin and pin.

kin array will store the keys of original array but only those keys which are at distance of 3 locations are stored in kin array.

pin array will store the position of the keys which are stored in the kin array.

kin array is
10 40 70 100

pin array is
0 3 6 9

Suppose we want to search x = 30. Then we take help of kin array and we find that x = 30 can be found only between element 10 and element 40 which are at position 0 and 3 respectively. Hence we sequentially search x = 30 between the locations 0 and 3.

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int d[100],kin[20],pin[20],temp,k,i,j=0,n,n1=0,start,end;
clrscr();
printf("Enter the number of elements:");
scanf("%d",&n);
for(i=0;i<n;i++)
scanf("%d",&d[i]);
printf("Enter the number to be searched:");
scanf("%d",&k);
for(i=0;i<n;i+=3)
{
kin[n1]=d[i];
pin[n1]=i;
n1++;
}
if(k < kin[0])
{
printf("element not found");
return;
}
else
{
for(i=1;i<=n1;i++)
if(k < kin[i] )
```

```
{
start=pin[i-1];
end=pin[i];
break;
}
}
for(i=start;i<=end;i++)
{
if(k==d[i])
{
j=1;
break;
}
}
if(j==1)
printf("element found at position %d",i);
else
printf("element not found");
}
```