Data Structures

Chapter 1 : Introduction to Data Structures

Q. 1 Define ADT with an example.

Dec. 2013, May 2016

Ans. : Abstract Data Types (ADT)

A big program is never written as a monolithic piece of program, instead it is broken down in smaller modules (may be called a function or procedure) and each module is developed independently.

When the program is hierarchical organized as shown in the Fig. 1.1, then the "main program" utilizes services at the functions appearing at level 1. Similarly, functions written at level 1 utilizes services of functions written at level 2. Main program uses the services of the next level function without knowing their implementation details. Thus a level of **abstraction** is created. When an abstraction is created at any level, our concern is limited to "what it can do" and not "how it is done".

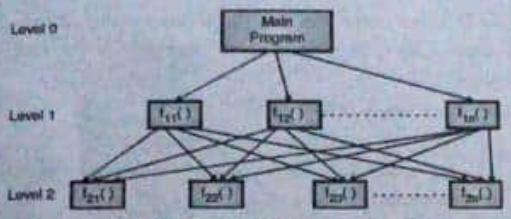


Fig. 1.1 : Hierarchical organized program

Abstraction in case of data

Abstraction for primitive types (char, int, float) is provided by the compiler. For example, we use integer type data and also, perform various operations on them without knowing them :

(1) Representation

(2) How various operations are performed on them.

Example :

```
int x, y, z ;
x = 13 ;
```

Constant 13 is converted to 2's complement and then stored in x. Representation is handled by the compiler

```
x = y + z ;
```

Meaning of the operation '+' is defined by the compiler and its implementation details remain hidden from the user. Implementation details (representation) and how various operation are implemented remain hidden from the user. User is only concerned about, how to use these operations. Objects such as lists, sets and graphs along with associated operations, can be viewed as abstract data type. Integer, char, real are primitive data types and there are set of operations associated with them. For the set ADT (Abstract data type), We might have operations like union, intersection, size and complement. Once the data type set is defined (representation and associated functions) then the ADT set can be used in any application program.

Q. 2 Explain different types of data structures with example.

Dec. 2013, May 2014, May 2015
Dec. 2016, May 2017

Ans. : Types of Data Structures

1. Primitive and Non-Primitive

Primitive

The integers, reals, logical data, character data, pointers and reference are primitive data structures. These data types are available in most programming languages as built in type. Data objects of primitive data types can be operated upon by machine level instructions.

Non-Primitive

These data structures are derived from primitive data structures. A set of homogeneous and heterogeneous data elements are stored together. Examples of Non-primitive data structures : Array, structure, union, linked-list, stack, queue, tree, graph. Some at the most commonly used operations that can be performed on data structures are shown in Fig. 1.2

Fig. 1.2 : Data structure operations

2. Linear and Non-Linear

Linear

Elements are arranged in a Linear fashion (one dimension). All one-one relation can be handled through Linear data structures. Lists, stacks and queues are examples of linear data structure.

Representation of Linear data structures in an array

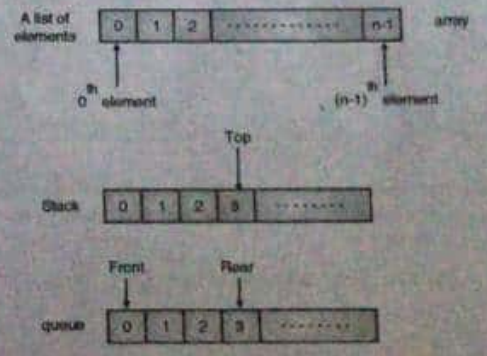


Fig. 1.3(a)

Representation of Linear data structures through Linked structure

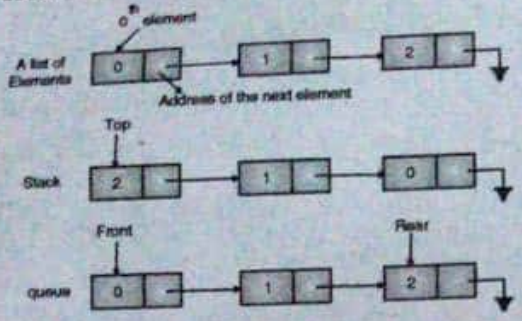
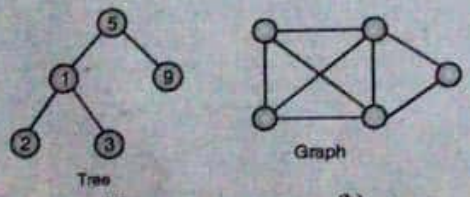


Fig. 1.3(b)

Non-Linear

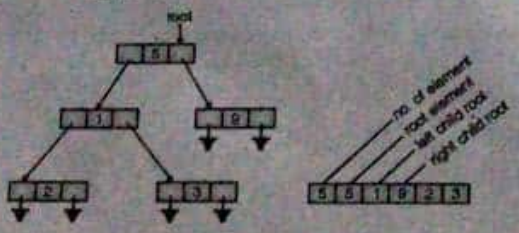
All one-many, many-one or many-many relations are handled through non-linear data structures. Every data element can have a number of predecessors as well as successors. Tree graphs and tables are examples of non-linear data structures.



5	9	1
6	4	13
2	5	0
9	8	11

Table
(c)

Fig. 1.4 : Non- Linear data structures



(a) Representation of the binary tree through linked structure

(b) Representation of the binary tree through an array

Fig. 1.5 : Representation of tree of Fig. 1.4(a)

3. Static and Dynamic

Static

In case of static data structure, memory for objects is allocated at the time of loading of the program. Amount of memory required is determined by the compiler during compilation.

Example : int a[50]

Memory for the array 'a' of 50 element will be allocated at the time of loading of the program. It may not always be possible to fix the size of the array in advance. Amount of data to be handle is often determined by the user and not by the programmer. Our initial judgment of size, if wrong, may cause failure of program (due to overflow) or wastage of memory space. Static data structure causes under utilization of memory (in case of over allocation). Static data structure may cause overflow (under allocation). No reusability of allocated memory. Difficult to guess the exact size of data at the time of writing of program.

Dynamic

In case of dynamic data structures, the memory space required by variables is calculated and allocated during execution. Dynamic memory is managed in 'C' through a set of library function.

Allocating a block of memory in "C"

ptr = (cast – type) malloc (byte – size) ;

The "malloc ( )" returns a pointer (of cast type) to an area of memory with size, byte – size

Example : x = (int *) malloc(100 * sizeof(int)) ;

A Linear data structure can be implemented either through static or dynamic data structures. Static data structured is preferred. All Linked structures are preferably implemented through dynamic data structures. Dynamic data structures provides flexibility in adding, deleting or rearranging data objects at run time. Additional space can be allocated at run time. Unwanted space can be released at run time. It gives re-usability of memory space.

Chapter 2 : Stack and Queues

Q. 1 **Write a 'C' program to convert decimal to binary using any appropriate data structure you have studied.** Dec. 2013

Ans. :

Program for conversion of decimal number to binary form.

```
#include<stdio.h>
#include<conio.h>
#define MAX 20
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
void init(stack *);
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *,int);
void main()
{
    stack s;
    int x;
    init(&s);
    printf("\nEnter decimal number:");
    scanf("%d",&x);
    while((x!=0))
    {
        if(!full(&s))
        {
            push(&s,x%2);
            x=x/2;
        }
        else
        {
            printf("\nstack overflow");
            exit(0);
        }
    }
    printf("\n");
    while(!empty(&s))
    {
        x=pop(&s);
        printf("%d",x);
    }
}
void init(stack *s)
{
    s->top=-1;
}
int empty(stack *s)
{
    if(s->top==-1)
        return(1);
    return(0);
}
int full(stack *s)
{
    if(s->top==MAX-1)
        return(1);
    return(0);
}
void push(stack *s,int x)
{
    s->top=s->top+1;
    s->data[s->top]=x;
}
int pop(stack *s)
{
    int x;
    x=s->data[s->top];
    s->top=s->top-1;
    return(x);
}
```

Output

```
Enter decimal number:12
1100
```

Q. 2 **Write a program to check whether a given string is a palindrome.**

Ans. :

```
# include <stdio.h>
# include <conio.h>
typedef struct stack
    {   char data[30];
        int top;
    }   stack;
void init (stack *p)
        {   p -> top = -1;
        }
void push ( stack *p, char x)
        {   p -> top = p -> top + 1;
            p -> data [p -> top] = x;
        }
char POP (stack *p)
        {   char x;
```

```
            x = p -> data [p -> top];
            p -> top = p -> top -1;
            return (x);
        }
int empty (stack *p)
        {   if (p -> top== = -1)
                return(1);
                return(0);
        }
void palindrome (char[]);
void main()
    {   stack s;
        char text [20];
        int i;
        int (&s);
        printf("\n Enter a string:");
        gets (text);
        palindrome (text);
    }
void palindrome (char text[])
        {   stack S;
            int i ;
            init (&S);
\* stack is being used to reverse the string. Original string and
the reversed string are compared character by character */
            for (i = 0; text[i]! = '\0'; i++)
                    push (&S, text[i]);
            for (i = 0; text[i] ! = '\0'; i++)
                    if (text[i] != POP(&S))
                            break;
if (text[i] != '\0') /* a mismatch found before end of the string
*/
printf("\n Not a palindrome");
else
            printf("\n A Palindrome");
            getch ();
        }
```

Output

```
Enter a string : MADAM
A palindrome.
```

Q. 3 Explain STACK as ADT ? Dec. 2015, Dec. 2016

Ans. :

Stack as an ADT

A stack is an ordered list with the restriction that insertions and deletions can be performed at only one position, namely, the front end of the list, called the top. The fundamental operation on the stack are :

(i) Push, equivalent to an insert

(ii) POP, equivalent to deleting the most recently inserted element.

A POP on an empty stack is considered to be an error. A push operation on stack that is full is considered to be an error.

Structure used for stack

```
typedef struct stack
  {  int data[SIZE];  // int has been taken for
                      convenience, it could be any data type.
     int top;
  } stack;
```

Operation on stack

```
initialize (stack *)        /*make a stack empty by setting top
                            equal to -1*/
boolean empty (stack)       /*determine, whether the stack is
                            empty */
boolean full (stack)        /*determine, whether the stack is full*/
int POP (stack *)           /*if the stack is not empty then POP
                            the top element */
push (stack*, int)          /*if the stack is not full then PUSH a
                            new data on top of the stack*/
int gettop(stack*)          /*if the stack is not empty then retrieve
                            the top element*/
```

Q. 4 Give application of stack. Dec. 2015

Ans. :

Applications of Stack

Stack data structure is very useful. Few of its usages are given below :

1. Expression conversion
   - (a) Infix to postfix
   - (b) Infix to prefix
   - (c) Postfix to infix
   - (d) Prefix to infix
2. Expression evaluation
3. Parsing
4. Simulation of recursion
5. Function call

Q. 5 Explain infix, postfix and prefix expressions with an example. May 2015

Ans. :

Expression Representation

There are three popular methods for representation of an expression.

(a) infix x + y operator between operands

(b) prefix + x y operator before operands

(c) postfix x y + operator after operands

Example :

Infix	x + y * z
Prefix	+ x * y z
Postfix	x y z * +

(a) Evaluation of an infix expression

Infix expressions are evaluated left to right but operator precedence must be taken into account. To evaluate x + y * z, y and z will be multiplied first and then it will be added to x.

Infix expressions are not used for representation of an expression inside computer, due to additional complexity of handling of precedence.

(b) Evaluation of a prefix expression

Consider an example for evaluation of prefix expression.

+ 5 * 3 2

find an operator from right to left and perform the operation.

+ 5 * 3 2

first operator

First operator is * and therefore, 3 * 2 are multiplied expression becomes + 5 6.

First operator is + and therefore, 5 and 6 are added.

expression becomes : 11

(c) Evaluation of postfix expression

5 3 2 * +

Find the first operator from left to right and perform the operation

first operator is * and therefore, 3 and 2 are multiplied

expression becomes 5 6 +

first operator is + and therefore, 5 and 6 are added

expression becomes : 11

Q. 6 Write program in 'C' to evaluate a postfix expression. Dec. 2013

Ans. :

```
Assumption— primary operators '-,+,*,/,%' operand — a
single digit
#include<stdio.h>
#include<conio.h>
#define MAX 20
typedef struct stack
{
    int data[MAX];
    int top;
}stack;
void init(stack *);
int empty(stack *);
int full(stack *);
int pop(stack *);
void push(stack *,int);
int evaluate(char x,int op1,int op2);
void main()
{
    stack s;
    char x;
    int op1,op2,val;
    init(&s);
    printf("\nEnter the expression(i.e 59+3*)\nsingle digit
operand and operators only:");
    while((x=getchar())!='\n')
    {
        if(isdigit(x))
            push(&s,x-48);
        else
        {
            op2=pop(&s);
            op1=pop(&s);
            val=evaluate(x,op1,op2);
            push(&s,val);
        }
    }
    val=pop(&s);
    printf("\nvalue of expression = %d",val);
}
int evaluate(char x,int op1,int op2)
{
    if(x=='+')
        return(op1+op2);
    if(x=='-')
        return(op1-op2);
    if(x=='*')
        return(op1*op2);
    if(x=='/')
        return(op1/op2);
    if(x=='%')
        return(op1%op2);
}
void init(stack *s)
{
    s->top=-1;
}
int empty(stack *s)
{
    if(s->top==-1)
        return(1);
    return(0);
}
int full(stack *s)
{
    if(s->top==MAX-1)
        return(1);
    return(0);
}
void push(stack *s,int x)
{
    s->top=s->top+1;
    s->data[s->top]=x;
}
int pop(stack *s)
{
    int x;
    x=s->data[s->top];
    s->top=s->top-1;
    return(x);
}
```

Output

```
Enter the expression(i.e. 59+3*)
single digit operand and operators only:653+9*+
value of expression = 78
```

Q. 7 Compare stacks and queues.

Ans. :

Comparison of stack and queue

Sr. No.	Stack	Queue
1.	Stack is last in first out (LIFO) i.e. which is entered last will be retrieved firstly.	Queue is first out (FIFO) i.e. which is entered first will be served first.
2.	Stack is Linear data structure which follows LIFO.	Queue is Linear data structure which follows FIFO.
3.	Insertion and deletions are possible through one end called top.	Insertions are at the rare end and deletions are from the front end in a queue.
4.	Example : Books in library.	Example : Cinema ticket counter.

Q. 8 Evaluate the following postfix expression. Show all steps :

ab * c + d – e + where a = 5, b = 4, c = 10, d = 15 and e = 6. Dec. 2015

Ans. :

Step	Expression	Stack
1.	ab*c + d – e +	[ ]
2.	b*c + d – e +	[5]
3.	*c + d – e+	[4, 5]
4.	c + d – e+	[20]
5.	+ d – e+	[10, 20]
6.	d – e +	[30]
7.	– e+	[15, 30]
8.	e+	[15]
9.	+	[6, 15]
10.	End	[21]

∴ Value of the expression = 21

Q. 9 Convert the following expression to postfix. (f – g)* * ((a + b)* (c – d)) /e May 2017

Ans. :

Sr. No.	Stack	Infix expression	Postfix expression
1.	Empty	(f – g) * ((a + b) * (c – d))/e	–
2.	(	f – g) * ((a + b) * (c – d))/e	–
3.	(	– g) * ((a + b) * (c – d))/e	f
4.	(–	g) * ((a + b) * (c – d))/e	f
5.	(–	) * ((a + b) * (c – d))/e	fg
6.	Empty	*((a + b) * (c – d))/e	fg-
7.	*	((a + b) * (c – d))/e	fg-
8.	*((	a + b)*(c – d))/e	Fg-
9.	*((	+ b) * (c – d))/e	fg – a
10.	*((+	b) * (c – d))/e	fg – a
11.	*((+	)*(c – d))/e	fg – ab
12.	*(	*(c – d))/e	fg – ab+
13.	*(*	(c – d))/e	fg – ab+
14.	*(*(	c – d))/e	fg – ab+
15.	*(*(	– d))/e	fg – ab+c
16.	*(*(–	d))/e	fg – ab + c
17.	*(*(–	))/e	fg – ab + cd
18.	*(*	)/e	fg – ab + cd-
19.	*	/e	fg – ab + cd – *
20.	*/	e	fg – ab + cd – *
21.	*/	–	fg – ab + cd – *e
22.	Empty	–	fg – ab + cd – * e/*

∴ Postfix Expression = fg – ab + cd – *e/*

Q. 10 Write a program for conversion of infix to its postfix form operators supported '+,-,*,/,%,^,(,) operands supported – all single character operands. May 2014, Dec. 2015

Ans. :

```
/* program for conversion of infix into its postfix form
   operators supported '+,-,*,/,%,^,(,)
   operands supported -- all single character operands
*/

#include<stdio.h>
#include<conio.h>
#include<ctype.h>
#define MAX 50
typedef struct stack
{
    int data[MAX];
    int top;
}stack;

int precedence(char);
```

```
void init(stack *);
int  empty(stack *);
int  full(stack *);
int  pop(stack *);
void push(stack *,int );
int  top(stack *); //value of the top element
void infix_to_postfix(char infix[],char postfix[]);

void main()
 { char infix[30],postfix[30];
  clrscr();
  printf("\nEnter an infix expression : ");
  gets(infix);
  infix_to_postfix(infix,postfix);
  printf("\nPostfix : %s ",postfix);
  getch();
 }
void infix_to_postfix(char infix[],char postfix[])
 {    stack s;
      char x;
      int i,j;//i-index for infix[],j-index for postfix
      char token;
      init(&s);
      j=0;
      for(i=0;infix[i]!='\0';i++)
       {    token=infix[i];
          if(isalnum(token))
               postfix[j++]=token;
          else
               if(token == '(')
                    push(&s,'(');
               else
                    if(token == ')')
                      while((x=pop(&s))!='(')
                        postfix[j++]=x;
                    else
                    {

    while(precedence(token)<=precedence(top(&s)) &&
!empty(&s))
                         {
                              x=pop(&s);
                              postfix[j++]=x;
                         }
                         push(&s,token);
                    }
       }
      while(!empty(&s))
      {
          x=pop(&s);
          postfix[j++]=x;
      }
postfix[j]='\0';
}
int precedence(char x)
{
     if(x == '(')                    return(0);
     if(x == '+' || x == '-')        return(1);
     if(x == '*' || x == '/' || x == '%') return(2);
     return(3);
}

void init(stack *s)
{
     s->top=-1;
}

int empty(stack *s)
{
     if(s->top==-1)  return(1);
     return(0);
}
int full(stack *s)
{
     if(s->top==MAX-1)   return(1);
     return(0);
}

void push(stack *s,int x)
{
     s->top=s->top+1;
     s->data[s->top]=x;
}

int pop(stack *s)
{
     int x;
     x=s->data[s->top];
     s->top=s->top-1;
     return(x);
}

int top(stack * p)
{
     return(p->data[p->top]);
}
```

Output

```
Enter infix expression:a*(b+c)/d+g
abc+*d/g+
```

Q. 11 Explain the term recursion with an example. Dec. 2014, May 2015, May 2017

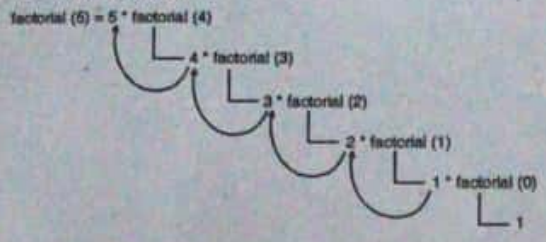
Ans. : Recursion

Recursion is a fundamental concept in mathematics. When a function is defined in terms of itself then it is called a recursive function. Consider the definition of factorial of a positive integer n.

$$\text{Factorial}(n) = \begin{cases} 1 & \text{if } (n=0) \\ n * \text{factorial}(n-1), & \text{otherwise} \end{cases}$$

Function "factorial( )" is defined in terms of itself for n > 0. Value of the function at n = 0 is 1 and it is called the base. Recursion terminates on reaching the base.

For example :



Recursion expands when n > 0 and it starts winding up on hitting the base (n = 0).

Q. 12 Write recursive functions to calculate GCD of 2 numbers. May 2014, May 2015

Ans. :

Recursive definition of GCD of two numbers

$$f(x, y) = \begin{cases} y & \text{if } (x \text{ is divisible by } y) \\ f(y, x \% y) & \text{otherwise} \end{cases}$$

Note : It is assumed that x ≥ y

C function for finding GCD

```
int GCD(int x, int y)
   {  int temp;
      if(x < y)      /* y intechange x and y if x < y y */
         { temp = x;
           x = y;
         }
      if(x % y) = = 0
            return(y);
      return(GCD(y, x % y));
   }
```

Q. 13 Write a recursive function in 'C' to find sum of digits of a number. May 2017

Ans. :

C function for finding sum of digits of a number

```
int Sum(int x, int t)
   {  if(x = =0)
      return(0);
      t=t+x%10;
      return(t);
   }
```

Q. 14 Write a 'C' function to calculate sum of elements of an array. Dec. 2014

Ans. :

'C' function for finding sum of the elements of an array

```
int sum( int a[ ], int n)
{
    if(n = = 1)
        return( a[0]);
    else
        return(a[n – 1] + sum(a, n – 1));
}
```

Q. 15 Explain use of stack in function call. May 2016

Ans. : Use of stack in function call

Nested calls of functions are managed through stack. When a function is invoked, memory is allocated on stack for Local variables and Return address.

On termination of execution of the function, memory is de-allocated. Details regarding the called functions are stored in a specific structure known as activation record. An activation record consists of :

1. Storage space for local variables
2. Definition of function
3. Return address
5. A pointer to activation record.

Activation records are stored on the stack. When a function is called, its activation record is pushed into the stack and control is transferred to the called program. When the function execution completes, the control returns to the caller function. It obtains the return address from the return address field of the activation record.

Q. 16 Discuss any two applications of Queue data structure. Dec. 2014, May 2017

Ans. : Application of Queues

Various features of operating system are implemented using a queue.

(a) Scheduling of processes (Round Robin Algorithm).
(b) Spooling (to maintain a queue of jobs to be printed).
(c) A queue of client processes waiting to receive the service from the server process.

Various application software using non-linear data structure tree or graph requires a queue for breadth first traversal. Simulation of a real life problem with the purpose of understanding its behaviour. The probable waiting time of a person at a railway reservation counter can be found through the technique of computer simulation if the following concerned factors are known :

(1) Arrival rate
(2) Service time
(3) Number of service counters

Q. 17 **Write a program in 'C' to implement a circular queue. The following operations should be performed by the program :**

(i) Creating the queue

(ii) Deleting from the queue

(iii) Inserting in the queue

(iv) Displaying all the elements of the queue.

Dec. 2013, May 2015, Dec. 2015, May 2016

Ans. :

```
#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct queue
{
    int data[MAX];
    int rear, front;
} queue;
void initialize (queue *p);
int empty (queue *p);
int full (queue *p);
void enqueue (queue *p, int x);
int dequeue (queue *p);
void print (queue *p);

void main ( )
{
    int x, op, n;
    queue q;
    initialize (&q);
    do
    {
        printf ("\n1)create \n2)insert \n3)delete \n4)print
\n5)Quit");
        printf("\n enter your choice :");
        scanf ("%d", &op);
        switch(op)
        { case 1 : printf("\n enter no. of elements :");
                   scanf("%d", &n);
                   initialize (&q);
                   printf("\n enter the data :");
                   for(i =0; i < n; i++)
                   {
                       scanf("%d", &x);
                       if (full(&q))
                       {
                           printf("\n queue is full ...");
                           exit(0);
                       }
                       enqueue (&q, x);
                   }
                   break;
        case 2 : printf("\n enter the element to be inserted
:");
                   scanf("%d", &x);
                   if (full(&q))
                   {
                       printf("\n queue is full ...");
                       exit(0);
                   }
                   enqueue (&q, x);
                   break;
        case 3 : if (empty(&q))
                   {
                       printf("\n queue is empty ...");
                       exit (0);
                   }
                   x = dequeue (&q);
                   printf("\n element =%d", x);
                   break;
        case 4 : print (&q);
                   break;
        default : break;
        }
    }while (op! = 5);
}
void initialize(queue *P)
{
    P ->rear = – 1;
    P ->front = –1;
}
int empty(queue *P)
{
    if(P ->rear == – 1)
        return(1);
    return(0);
}
int full(queue *P)
{
    if((P ->rear + 1)% MAX == P ->front)
        return(1);
    return(0);
}
void enqueue(queue *P, int x)
{
    if(empty(P))                    /* empty queue */
    {
        P ->rear = P ->front = =0;  /* rear and front will
                                       point to the same
                                       element */
        P ->data[P ->rear] = x;
    }
    else
    {
```

```
        P ->rear = (P ->rear + 1)% MAX;  /* Advance P in the
                                            circular array */
        P ->data[P ->rear] = x;
    }
}
int dequeue(queue *P)
{
        int x;
        x = P ->data[P ->front];
        if(P ->rear == P ->front)        /* deleted the last
                                            element */
            initialize(P);
        else
            P ->front =(P ->front + 1)% MAX;
        /* front advances to the next position in the circular
array */
        return(x);
}
void print(queue *P)
{
    int i;
    i = P ->front;
    while(i != P ->rear)
    {
        printf("\n%d", P ->data[i]);
        i = (i + 1) %MAX;
    }
    printf("\n%d", P ->data[P ->rear]);
}
```

Q. 18 **Write a 'C' program to implement a priority queue.** Dec. 2014

Ans. :

```
#include <stdio.h>
#include <conio.h>
#define MAX 30
typedef struct pqueue
{    int data [MAX];
     int rear, front;
} pqueue1
void initialize(pqueue *p);
int empty(pqueue *p);
int full(pqueue *p);
void enqueue(pqueue *p, int x);
int dequeue(pqueue *p);
void print(pqueue *p);
void main( )
{ int x, op, n;
  pqueue q;
  initialize (&q);
do
{

printf("\n1)create\n2)insert\n3)Delete\n4)print\n5)Quit");
    printf("\n enter your choice:");
    scanf ("%d", &op);
    switch (op)
    {
    case 1 : printf("\n enter no. of elements :");
          scanf("%d", &n);
          initialize (&q);
          printf("enter the data:");
          for (i = 0; i < n; i++)
          {
             scanf ("%d" &x);
             if (full (&q))
             {   printf("\n queue is full ...");
                 exit(0);
             }
             enqueue (&q, x);
          }
          break;
    case 2 : printf("\n enter the element to be inserted");
          scanf ("%d", &x);
          if (full(&q))
          {
             printf("\n queue is full ...");
             exit(0);
          }
          enqueue (&q, x);
          break;
          case 3 : if (empty (&q))
          {
                printf("\n queue is empty ...");
                exit(0);
          }
          x = dequeue (&q);
          printf("\n element = %d", x);
          break;
          case 4 : print(&q);
          break;
           default : break;
          }
    } while (op! = 5);
}
void initialize (pqueue *p)
{   p -> rear = – 1;
    p -> front = – 1;
}
/* A value of rear or front as –1, indicate that the queue is
empty. */
int empty (pqueue *p)
{ if (P->rear = = – 1)
        return (1);     /* queue is empty */
    return (0); /* queue is not empty */
}
```

```
int full (pqueue *p)
{ /* if front is next rear in the circular array then the queue is
full */
      if (p -> rear + 1)% MAX == p -> front)
         return (1); /* queue is full */
      return (0);
}
void enqueue (pqueue *p, int x)
{ int i;
     if (full (p))
     printf("\n overflow ...");
      else
     {   /* inserting in an empty queue */
         if (empty (p))
              {   p -> rear = p -> front = 0
                  p -> data [0] = x;
              }
         else
         {
         /* move all lower priority data right by one place */
     i = p -> rear ;
     while (x > p -> data [i])
     {   p -> data [(i + 1)%MAX] = p -> data [i];
         /* position i on the previous element */
         i = (i – 1 + MAX) % MAX; /* anticlock wise
movement inside the queue */
         if ((i + 1)% MAX == p -> front)
              break; /* if all elements have been moved */
     }
     /* insert x */
         i = (i + 1)% MAX;
         p -> data [i] = x;
     /* re-adjust rear */
         p -> rear = (p -> rear + 1) % MAX;
     }
 }
}
int dequeue (pqueue *p)
{   int x;
    if (empty (p))
              printf("\n underflow ...");
         else
         {   x = p -> data [p -> front];
             if (p -> rear == p -> front)      /* delete last
                                                  element */
                  initialize (p);
             else
                  p -> front = (p -> front + 1) % MAX;
         }
    return (x);
    }
void print (pqueue *p)
{   int i, x;
    i = p -> front;
    while ( i! = p -> rear)
    {   x = p data[i];
        printf ("\n%d", x);
        i = (i + 1) % MAX;
    }
    /* print the last data */
        x = p -> data[i];
        printf ("\n%d", x);
    }
```

Q. 19 Explain Double ended queue.

May 2018, Dec. 2016

Ans. : Dequeues

The word **dequeue** is a short form of double ended queue. It is general representation of both stack and queue and can be used as stack and queue. In a dequeue, insertion as well deletion can be carried out either at the rear end or the front end. In practice, it becomes necessary to fix the type of operation to be performed on front and rear end. Dequeue can be classified into two types :

(1) Input restricted Dequeue

The following operations are possible in an input restricted dequeue :

(i) Insertion of an element at the rear and
(ii) Deletion of an element from front end
(iii) Deletion of an element from rear end

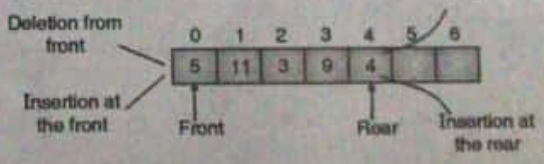
(2) Output restricted Dequeue

The following operations are possible in an output restricted dequeue.

(i) Deletion of an element from front end
(ii) Insertion of an element at the rear end
(iii) Insertion of an element at the front end

There are various methods to implement a dequeue.

(a) Using a circular array
(b) Using a singly linked list.
(c) Using a singly circular linked list.
(d) Using a doubly linked list.
(e) Using a doubly circular linked list.



dss(11.33)**Fig. 2.1 : A dequeue in a circular array**

Operations associated with dequeue :

(a) empty( ) : Whether the queue is empty ?
(b) full( ) : Whether the queue is full ?
(c) initialize( ) : Make the queue empty
(d) enqueueR( ) : Add item at the rear end of the queue.
(e) enqueueF( ) : Add item at the front end of the queue.
(f) dequeueR( ) : Delete item from the rear end of the queue.
(g) dequeueF( ) : Delete item from the front end of the queue.

Timing complexity of various dequeue operations :
enqueue R( ) – O(1) – constant time.
enqueue F( ) – O(1) – constant time.
dequeue R( ) – O(1) – constant time.
dequeue F( ) – O(1) – constant time.

Advantage of dequeue :

The dequeue is a general representation of both stack and queue it can be used both as stack or a queue.

Dequeue as an ADT

Data type for dequeue in an array

```
#define MAX 30 /* A queue with maximum of 30 elements */
typedef struct DQ
{
    int data [MAX];
    int rear, front;
} DQ;
```

Operations on a dequeue

(i) initialize( ) : Make the queue empty.
(ii) empty( ) : Determine if queue is empty.
(iii) full( ) : Determine if queue is full.
(iv) enqueueF( ) : Insert an element at the front end of the queue.
(v) enqueueR( ) : Insert an element at the rear end of the queue.
(vi) dequeueR( ) : Delete the rear element.
(vii) dequeueF( ) : Delete the front element.
(viii) print( ) : Print elements of the queue.

Prototype of functions used for various operations on queue

(i) void intialize (DQ *p);
(ii) int empty (DQ *p);
function returns 1 or 0, depending on whether the queue pointed by p is empty or not.
(iii) int full (DQ *p);
function returns 1 or 0, depending on whether the queue pointed by p is full or not.
(iv) void enqueueF (DQ *p, int x);
(v) void enqueueR (DQ *p, int x);
(vi) int deleteR (DQ *p);
(vii) int deleteF (DQ *p);
(viii) void print (DQ *p);

enqueueR( ) and enqueueF( ) will cause an overflow if the queue is full. dequeueR( ) and dequeueF( ) will cause an underflow if the queue is empty.

Chapter 3 : Linked List

Q. 1 What are the advantages of using linked lists over arrays ? May 2016

Ans. : Advantages of linked lists

i) Linked list is an example of dynamic data structure. They can grow and shrink during execution of the program.
ii) Representation of linear data structure. Inline data like polynomial, stack and queue can easily be represented using linked list.
iii) Efficient memory utilization. Memory is not pre-allocated like static data structure. Memory is allocated as per the need. Memory is deallocated when it is no longer needed.
iv) Insertion and deletions are easier and efficient. Insertion and deletion of a given data can be carried out in constant time.

Q. 2 Explain Linked list as an ADT. Dec 2015

Ans. : Representation

The linked list consists of a series of structures. They are not required to be stored in adjacent memory locations. Each structure consists of a data field and address field. Address field contains the address of its successors. Fig. 3.1 shows the actual representation of the structure.

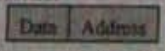


Fig. 3.1 : Representation of the structure

A variable of the above structure type is conventionally known as a **node**. Fig. 3.2 gives a representation of a linked list of three nodes.

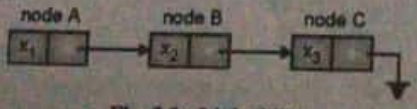


Fig. 3.2 : Linked list

A list consisting of three data x_1, x_2, x_3 is represented using a linked list. Node A stores the data x_1 and the address of the successor (next) node B. Node B stores the data x_2 and the address of its successor node C. Node C contains the data x_3 and its address field is grounded (NULL pointer), indicating it does not have a successor.

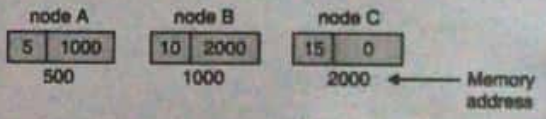


Fig. 3.3 : Memory representation of a linked list

Fig. 3.3 gives a memory representation of the linked list shown in Fig. 3.2. Nodes A, B and C happen to reside at memory locations 500, 1000 and 2000 respectively. $x_1 = 5$, $x_2 = 10$ and $x_3 = 15$.

Node A resides at the memory location 500, its data field contains a value 5 and its address field contains 1000, which is the address of its successor node. Address field of node C contains 0 as it has no successor.

Implementation

Structures in "C" can be used to define a node. Address of the successor node can be stored in a pointer type variable.

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

It is a **self referential structure** in "C". It is assumed that the type of data to be stored in each node is predefined to be of integer type. Next field of the structure is a pointer type variable, used for storing address of its successor node. Nodes are manipulated during run-time. A programming language must

provide following facilities for run time manipulation of nodes. Acquiring memory for storage during run-time. Freeing memory during execution of the program, once the need for the acquired memory is over.

In "C" programming language, memory can be acquired through the use of standard library functions malloc( ) and calloc( ). Memory acquired during runtime can be freed through the use of library function free( ).

```
/* 1 */   node *P;
/* 2 */   P = (node *) malloc(sizeof(node));
/* 3 */   P → data = 5;
/* 4 */   P → next = NULL
```

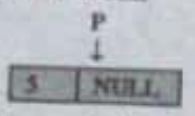


Fig. 3.4 : Memory for a node has been allocated during run-time. Its address is stored in the pointer P

In the program segment above, the line 1 declares a variable P with the storage class node *. It can store the address of a node that has been created dynamically. sizeof(node) in line 2 is storage requirement in number of bytes to store a node. malloc(sizeof(node)) returns the address of the allocated memory block and it is assigned to variable P.

The address returned by malloc(sizeof(node)) is type casted using type casting operator(node *) before assigning it to the pointer P.

P → data = 5, in line 3 stores a value of 5 in the data field of node whose address is stored in pointer P. P → next = NULL stores a value of NULL in the next field of the node whose address is stored in the pointer P.

A linked list with the address of the head node

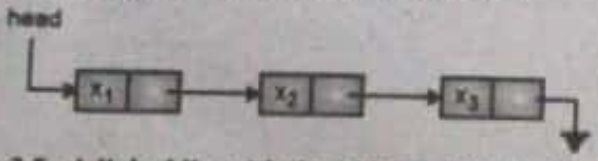


Fig. 3.5 : A linked list with the address of the head node

As an array is referenced by its starting address, a linked list is known by the address of its head node. Address of the head node is stored in a pointer variable (node * head) head. All manipulations on the linked list can be performed through the address of the starting node. Through the variable "head", first node can be accessed and through the address of the second node stored in the next field of the first node, second node can be accessed and so on.

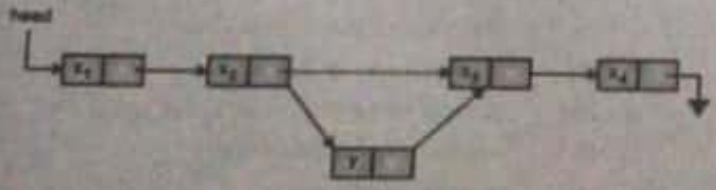


Fig. 3.6 : Insertion into a linked list

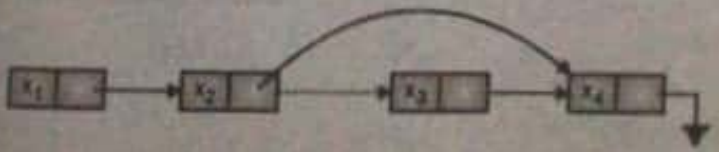


Fig. 3.7 : Deletion from a linked list

Insertion into a linked list requires obtaining a new node and then changing values of two pointers. The general idea is shown in Fig. 3.6. The dashed line represents the old pointer. It is changed to point to new node. Deletion of a node can be performed in one pointer change. Fig. 3.7 shows the result of deleting the node containing x_3. The next field of the node containing x_2 is changed to point to the node containing x_4. Node containing x_3 is freed using the library function free( ) subsequently.

Q. 3 List and explain types of linked list.

Ans. :

Types of Linked List

1. **Singly Linked List :** In this type of linked list two successive nodes of the linked list are linked with each other in sequential linear manner. Movement in forward direction is possible.

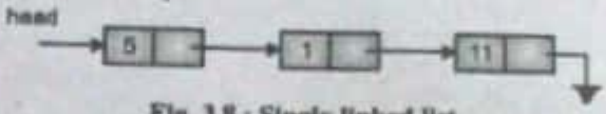


Fig. 3.8 : Singly linked list

2. **Doubly Linked List :** In this type of linked list each node holds two-pointer fields. In doubly linked list addresses of next as well as preceding elements are linked with current node.

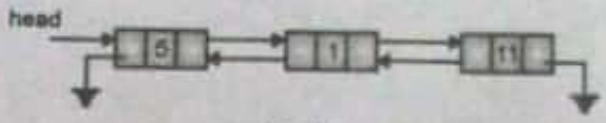


Fig. 3.9

3. **A Circular Linked List :** In a circular list the first and the last elements are adjacent. A linked list can be made circular by storing the address of the first node in the next field of the last node.

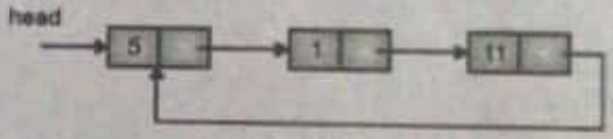


Fig. 3.10

Q. 4 State differences between Singly Linked List and Doubly Linked list data structures along with their applications. May 2015

Ans. :

Difference between Singly Linked List and Doubly Linked List

Sr. No.	Singly linked list	Doubly linked list
1.	It has one pointer, pointing to successor.	It has two pointers, one pointing to successor and another pointing to predecessor.
2.	Traversal is possible only in the forward direction.	One can traverse in both forward and backward directions.
3.	Less memory is required by a node.	More memory required by a node.

Sr. No.	Singly linked list	Doubly linked list
4.	Fewer pointer adjustment in delete and insert operation.	More pointer adjustment in delete and insert operation.
5.	In singly linked list, each node contains data and link.	In doubly linked list, each node contains data, link to next node and link to previous node.
6	**Applications :** 1. Representation of linear data structure. 2. Representation of Stack. 3. Representation of Queue. 4. Representation of Polynomial. 5. Representation of Sparse matrix.	**Applications :** 1. Representation of dequeue. 2. Memory management and garbage collection.

Q. 5 **Write program to create linked list interactively and print the list and total number of items in the list.** Dec. 2016

Ans. :

```
#include<conio.h>
#include<stdio.h>
typedef struct node
{
    int data;
    struct node *next;
}node;
node * create(int);
void print(node *);
int count(node *);
void main()
{
    node *HEAD;
    int n,number;
    printf("\n no. of items :");
    scanf("%d",&n);
    HEAD=create(n); //create function returns the address of first node
    print(HEAD);
    number=count(HEAD);
    printf("\n No of nodes = %d",number);
}
node * create(int n)
{
    node *head,*P;
    int i;
    head=(node*)malloc(sizeof(node));
    head->next=NULL;
    scanf("%d",&(head->data));
    P=head;
    //create subsequent nodes
    for(i=1;i<n;i++)
    {
        P->next=(node*)malloc(sizeof(node));
        //new node is inserted as the next node after P
        P=P->next;
        scanf("%d",&(P->data));
        P->next=NULL;
    }
    return(head);
}
void print(node *P)
{
    while(P!=NULL)
    {
        printf("<- %d ->",P->data);
        P=P->next;
    }
}
int count(node *P)
{
    int i=0;
    while(P!=NULL)
    {
        P=P->next;
        i++;
    }
    return(i);
}
```

Output :

```
no. of items :4
12
13
14
15
<- 12 -> <- 13 -> <- 14 -> <- 15 ->
No of nodes = 4
```

Q. 6 **Write a 'C' program to implement a singly Linked List which supports the following operations :**

(i) **Insert a node in the beginning**
(ii) **Insert a node in the end**
(iii) **Insert a node after a specific node**
(iv) **Deleting a specific node**
(v) **Displaying the list.**

Dec. 2014, May 2017

Ans. :

```
/*Operations on SLL(singly linked list) */
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
```

```
typedef struct node
  { int data;
    struct node *next;
  }node;
node *create();
node *insert_b(node *head,int x);
node *insert_e(node *head,int x);
node *insert_in(node *head,int x);
node *delete_b(node *head);
node *delete_e(node *head);
node *delete_in(node *head);
node *reverse(node *head);
void search(node *head);
void print(node *head);
node *copy(node *);
int count(node *);
node * concatenate(node *, node *);
void split(node *);
void main()
{ int op,op1,x;
 node *head=NULL;
 node *head1=NULL,*head2=NULL,*head3=NULL;
 clrscr();
 do
  {
   printf("\n\n 1)create\n 2)Insert\n 3)Delete\n 4)Search");
   printf("\n 5)Reverse\n 6)Print\n 7)Count\n 8)Copy\n
9)Concatenate");
   printf("\n 10)Split\n 11)Quit");
   printf("\n Enter your Choice:");
   scanf("%d",&op);
   switch(op)
 { case 1:head=create();break;
    case 2:printf("\n\t 1)Beginning\n\t 2)End\n\t 3)In
between");
        printf("\n Enter your choice : ");
        scanf("%d",&op1);
        printf("\n Enter the data to be inserted : ");
        scanf("%d",&x);
        switch(op1)
        { case 1: head=insert_b(head,x);
                 break;
          case 2: head=insert_e(head,x);
                 break;
          case 3: head=insert_in(head, x);
                 break;
        }
       break;
    case 3:printf("\n\t 1)Beginning\n\t 2)End\n\t 3)In
between");
        printf("\n Enter your choice : ");
        scanf("%d",&op1);
        switch(op1)
        { case 1:head=delete_b(head);
                 break;
          case 2:head=delete_e(head);
                 break;
          case 3:head=delete_in(head);
                 break;
        }
        break;
    case 4:search(head);break;
    case 5:head=reverse(head);
       print(head);
       break;
    case 6: print(head); break;
    case 7: printf("\nNo.of node = %d",count(head));break;
//count
    case 8: head1=copy(head);//copy
         printf("\nOriginal Linked List :");
         print(head);
         printf("\nCopied Linked List :");
         print(head1);
         break;
    case 9:printf("\nEnter the first linked list:");
         head1=create();
         printf("\nEnter the second linked list:");
         head2=create();
         head3=concatenate(head1,head2);
         printf("\nConcatenated Linked List :");
         print(head3);
         break;
         //concatenate
    case 10:printf("\nEnter a linked list : ");
         head1=create();
         split(head1);
         break;
         //split
   }
  }while(op!=11);
}
node *create()
{ node *head,*p;
 int i,n;
 head=NULL;
 printf("\n Enter no of data:");
 scanf("%d",&n);
 printf("\n Enter the data:");
 for(i=0;i<n;i++)
  {
   if(head==NULL)
     p=head=(node*)malloc(sizeof(node));
   else
    {
```

```
        p->next=(node*)malloc(sizeof(node));
        p=p->next;
       }
      p->next=NULL;
      scanf("%d",&(p->data));
    }
  return(head);
}
node *insert_b(node *head,int x)
{  node *p;
   p=(node*)malloc(sizeof(node));
   p->data=x;
   p->next=head;
   head=p;
   return(head);
}
node *insert_e(node *head,int x)
{  node *p,*q;
   p=(node*)malloc(sizeof(node));
   p->data=x;
   p->next=NULL;
   if(head==NULL)
     return(p);
   //locate the last node
   for(q=head;q->next!=NULL;q=q->next);
   q->next=p;
   return(head);
}
node *insert_in(node *head,int x)
{  node *p,*q;
   int y;
   p=(node*)malloc(sizeof(node));
   p->data=x;
   p->next=NULL;
   printf("\n Insert after which number ? : ");
   scanf("%d",&y);
   //locate the data 'y'
   for(q=head ; q != NULL && q->data != y ; q=q->next);
   if(q!=NULL)
    {
     p->next=q->next;
     q->next=p;
    }
   else
     printf("\n Data not found ");
  return(head);
}
node *delete_b(node *head)
{
 node *p,*q;
 if(head==NULL)
   {
     printf("\n Underflow....Empty Linked List");
     return(head);
   }
 p=head;
 head=head->next;
 free(p);
 return(head);
}
node *delete_e(node *head)
{
 node *p,*q;
 if(head==NULL)
   {
     printf("\n Underflow....Empty Linked List");
     return(head);
   }
 p=head;
 if(head->next==NULL)
   { // Delete the only element
     head=NULL;
     free(p);
     return(head);
   }
//Locate the last but one node
 for(q=head;q->next->next !=NULL;q=q->next)
 p=q->next;
 q->next=NULL;
 free(p);
 return(head);
}
node *delete_in(node *head)
{
 node *p,*q;
 int x,i;
 if(head==NULL)
   {
     printf("\n Underflow....Empty Linked List");
     return(head);
   }
 printf("\n Enter the data to be deleted : ");
 scanf("%d",&x);
 if(head->data==x)
   { // Delete the first element
     p=head;
     head=head->next;
     free(p);
     return(head);
   }
//Locate the node previous to one to be deleted
 for(q=head;q->next->data!=x && q->next
!=NULL;q=q->next )
   if(q->next==NULL)
```

```
    {
      printf("\n Underflow.....data not found");
      return(head);
    }
   p=q->next;
   q->next=q->next->next;
   free(p);
   return(head);
}
void search(node *head)
{ node *p;
 int data,loc=1;
 printf("\n Enter the data to be searched: ");
 scanf("%d",&data);
 p=head;
 while(p!=NULL && p->data != data)
 { loc++;
   p=p->next;
 }
 if(p==NULL)
  printf("\n Not found:");
 else
  printf("\n Found at location=%d",loc);
}
void print(node *head)
{ node *p;
 printf("\n\n");
 for(p=head;p!=NULL;p=p->next)
  printf("%d  ",p->data);
}
node *reverse(node *head)
 { node *p,*q,*r;
   p=NULL;
   q=head;
   r=q->next;
   while(q!=NULL)
    {
     q->next=p;
     p=q;
     q=r;
     if(q!=NULL)
       r=q->next;
    }
  return(p);
 }
node *copy(node *h)
 {
   node *head=NULL,*p;
   if(h==NULL)
    return head;
   //Copy the first node
   head=p=(node*)malloc(sizeof(node));
   p->data=h->data;
   while(h->next != NULL)
    {
     p->next=(node*)malloc(sizeof(node));
     p=p->next;
     h=h->next;
     p->data=h->data;
    }
   p->next=NULL;
   return (head);
 }
int count(node *h)
 {
   int i;
   for(i=0; h!=NULL; h=h->next)
     i++;
   return(i);
 }
node *concatenate( node *h1, node * h2)
  {
    node *p;
    if(h1==NULL)
      return(h2);
    if(h2==NULL)
      return(h1);
    p=h1;
    while(p->next != NULL)      //go to the end of the ist
                                linked list
      p=p->next;
    p->next=h2;
    return(h1);
  }
void split(node *h1)
 {
   node *p,*q,*h2;
   printf("\n Linked list to be split : ");
   print(h1);
   //linked list will be broken from the centre using the
pointers p and q
   if(h1==NULL)
    return;
   p=h1;
   q=h1->next;
   while(q!=NULL && q->next != NULL)
    {
     q=q->next->next;
     p=p->next;  //When q reaches the last node,p will
                   reach the centre node
    }
   h2=p->next;
   p->next=NULL;
   printf("\nFirst half : ");
```


```
 print(h1);
 printf("\nSecond half : ");
 print(h2);
}
```

Q. 7 **Write a program in 'C' to implement circular queue using Link-list.** May 2014

Ans. :

```
//*********To implement circular queue using linked
list.*******//
#include<stdio.h>
#include<conio.h>
typedef struct node
{
 int data;
 struct node *next;
}node;
void init(node **R);
void enqueue(node **R,int x);
int dequeue(node **R);
int empty(node *rear);
void print(node *rear);
void main()
{
int x,option;
int n = 0,i;
node *rear;
init(&rear);
clrscr();
do
{
printf("\n 1. Insert\n 2. Delete\n 3. Print\n 4. Quit");
printf("\n your option:    ");
scanf("%d",&option);
switch(option)
{
 case 1 :
       printf("\n Number of Elements to be inserted");
       scanf("%d",&n);
       for(i=0;i<n;i++)
       {
        scanf("\n %d",&x);
        enqueue(&rear,x);
       }
      break;
case 2 : if(! empty(rear))
       {
       x = dequeue(&rear);
       printf("\n Element deleted = %d",x);
       }
       else
       printf("\n Underflow..... Cannot deleted");
       break;
 case 3 : print(rear);
       break;
 }
}while(option != 4);
getch();
}
void init(node **R)
{
 *R = NULL;
}
void enqueue(node **R,int x)
{
 node *p;
 p = (node *)malloc(sizeof(node));
 p->data = x;
 if(empty(*R))
 {
   p->next = p;
   *R = p;
 }
 else
 {
   p->next = (*R)->next;
   (*R)->next = p;
   (*R) = p;
 }
}
int dequeue(node **R)
{
 int x;
 node *p;
 p = (*R)->next;
 p->data = x;
 if(p->next == p)
 {
   *R = NULL;
   free(p);
   return(x);
 }
 (*R)->next = p->next;
 free(p);
 return(x);
}
void print(node *rear)
{
 node *p;
 if(!empty(rear))
 {
  p = rear->next;
 }
 p = p->next;
 do
```

```
  {
   printf("\n %d",p->data);
   p = p->next;
  }while(p != rear->next);
}
int empty(node *P)
{
    if(P->next== -1)
        return(1);
    return(0);
}
```

Output :

```
1. Insert
2. Delete
3. Print
4. Quit
your option :   1

Element to be inserted 4
                12  23  34   45
1. Insert
2. Delete
3. Print
4. Quit
your option :   3
              12  23  34   45
1. Insert
2. Delete
3. Print
4. Quit
your option :   2
Element deleted = 4
1. Insert
2. Delete
3. Print
4. Quit
your option :   3
              23  34  45
1. Insert
2. Delete
3. Print
4. Quit
your option :   4
```

Q. 8 State applications of Circular Linked List.

Ans. : Applications of Circular Linked List

Circular linked list can be used for storing a list of elements. It allows insertion in constant time at both ends of the list. In general, it can be used for following applications :

1. Stack
2. Queue
3. Dequeue
4. Priority queue
5. Polynomial
6. List
7. Sparse matrix.

Q. 9 Write a program for creation of a doubly linked list and printing its elements in forward and reverse direction.

Ans. :

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

dnode * create();
void print_forward(dnode *);
void print_reverse(dnode *);
void main()
{
    dnode *head;
    head=NULL;   // initially the list is empty
    head=create();
    printf("\nElements in forward direction :");
    print_forward(head);
    printf("\nElements in reverse direction :");
    print_reverse(head);
}
dnode *create()
{
    dnode *h,*P,*q;
    int i,n,x;
    h=NULL;
    printf("\nEnter no of elements :");
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        printf("\nEnter next data: ");
        scanf("%d",&x);
        q=(dnode*)malloc(sizeof(dnode));
        q->data=x;
        q->prev=q->next=NULL;
        if(h==NULL)
            P=h=q;
        else
            P->next=q;
            q->prev=P;
            P=q;
    }
    return(h);
}
void print_forward(dnode *h)
{
```

```
    while(h!=NULL)
    {
        printf("<- %d ->",h->data);
        h=h->next;
    }
}
void print_reverse(dnode *h)
{
    while(h->next!=NULL)
        h=h->next;
    while(h!=NULL)
    {
        printf("<- %d ->",h->data);
        h=h->prev;
    }
}
```

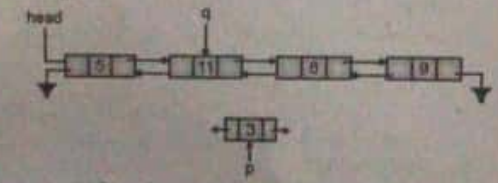
Output :

```
Enter no of elements : 4
Enter next data: 1
Enter next data: 2
Enter next data: 3
Enter next data: 4
Elements in forward direction :<-1-> <-2-><-3-><-4->
Elements in reverse direction :<-4-> <-3-><-2-><-1->
```

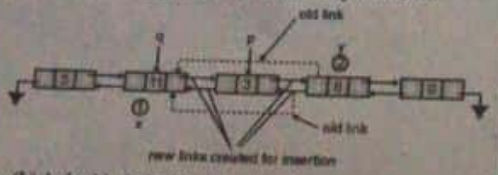
Q. 10 Write a function for insertion of a node in Doubly linked list. May 2014

Ans. :

Insertion and deletion are two basic operations on such lists. Consider that a new node pointed to by "P" is to be **inserted** after the node pointed to by q in a doubly linked list as shown in the Fig. 3.11.



(a) Insertion of a node in a doubly linked list



(b) A doubly linked list after the insertion of the node

Fig. 3.11

Links of two nodes are altered during insertion. In the Fig. 3.11(b), a new node pointed to by P is inserted between the two nodes x and y. Following links must be changed for proper insertion of node pointed by P between x and y.

(1) Right link of x
(2) Left link of y
(3) Left and right links of node pointed to by P

Instructions for making above changes

```
P → next = q → next;        //right link of P set to y
P → prev = q;               // left link of P set to x
q → next = P;               // right link of x set to P
(P → next) → prev = P;      // left link of y set to P
```

'C' function for inserting a node pointed to by P after a node pointed to by q :

```
void insert(dnode *P, dnode *q)
{
        P → next = q → next;
        P → prev = q;
        q → next = P;
        if(P → next ! = NULL)   /* insertion at the end */
            P → next → prev = P;
}
```

If the right pointer of x is NULL then the insertion is being performed at the end. Hence, there is no question of modifying the left pointer of y.

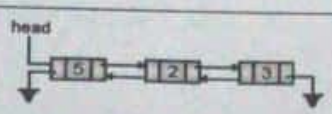
'C' function for inserting a node pointed to by p before a node pointed to by q :

```
dnode * insert (dnode * head, dnode *P, dnode * q)
    {
        if (q → prev == NULL)
        { // insert at the beginning
            P→ next = q;
            P→ prev = NULL;
            q→ prev = p;
            return (p);
        }
    else
    {   P → prev = q → prev;
        P → next = q;
        P → prev → next = p;
        q → prev = p;
        return (head);
    }
}
```

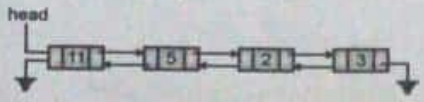
'C' function for inserting a value x, at the beginning of doubly linked list :

```
dnode * insert2(dnode * head, int x)
{
        dnode *P;
        P =(dnode *) malloc(sizeof(dnode));/* get memory
for the new node */
        P → data = x;
        P → prev = NULL;
        P → next = head;
```

```
    if(head ! = NULL)          /* not an empty list */
        head → prev = P;
    return(P);
}
```



(a) Original linked list



(b) Linked list after insertions of 11 at the front
Fig. 3.12

'C' function for inserting a value x, at the end of a doubly linked list :

```
dnode * insert3(dnode *head, int x)
{
    dnode *P, *q;
    P =(dnode *) malloc(sizeof(dnode));
    P → data = x;
    P → prev = P → next = NULL;
    if(head == NULL)                    /* empty list */
        return(P);
    /* go to the last node */
    q = head;
    while(q → next ! = NULL)
        q = q → next;
    P → prev = q
  q → next = P;
    return(head);
}
```

Q. 11 Write a function for deletion of a node from Doubly linked list. Dec. 2015

Ans. : Deletion of a Node

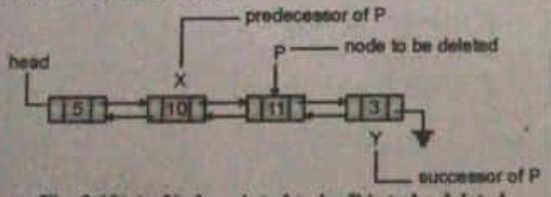


Fig. 3.13(a) : Node pointed to by P is to be deleted

When a node, pointed to by P is to be deleted than its predecessor node x and it's successors node y (as shown in Fig. 3.13(a)) will be affected.

Right link at x should be set to y.

Left link at y should be set to x.

Release the memory allocated to the node pointed to by P.

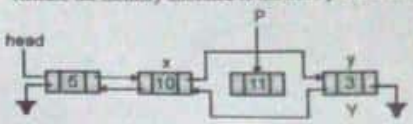


Fig. 3.13(b) : Links to be modified as shown above

C-instructions for deletions :

```
P → prev → next = P → next;  // right at x set to y
P → next → prev = P → prev;  // left link at y set to x
free(P);
```

'C' function for deletion of a node pointed by p in a doubly linked list.

```
dnode * delete_double(dnode *head, dnode *P)
{
    if(P = = head)      /* deleting the head node */
    {
        P → next → prev = NULL;
        head = P → next;
        free(P);
        return(head);
    }
    P → prev → next = P → next;
    if(P → next ! = NULL)        /* not the last node */
        P → next → prev = P → prev;
    free(P);
    return(head);
}
```

Special care should be taken to delete the first or the last node. If the node to be deleted is the first node then head should be advanced to the next node. If the node to be deleted is the last node then there is no node to its right.

Q. 12 Write a C program to implement a Doubly Linked List which performs the following operations : May 2016

(i) Inserting element in the beginning
(ii) Inserting element in the end
(iii) Inserting element after an element
(iv) Deleting a particular element
(v) Displaying the list

Ans. :

```
/*Accept input as a string and construct a doubly linked list
for the input string with each node containing as a data one
character from the string and perform :
a.  Insert                  b.  Delete
c.  Display forward         d.  Display backward */
/*Operations on a DLL( doubly linked list) */
/* Details :
```

Accept input as a string and construct a Doubly Linked List for the input string with each node contains, as a data one character from the string and perform :
a) Insert b) delete c)Display forward d) Display backward
*/

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
typedef struct student
{
 int rollno,marks;
 char name[15];
}student;

typedef struct node
 { student data;
  struct node *next,*prev;
 }node;
node *create();
node *insert_b(node *head,student x);
node *insert_e(node *head,student x);
node *insert_in(node *head,student x);
node *delete_in(node *head);
void displayforward(node *head);
void displaybackward(node *head);
void modify(node *head);
student read()
{ student x;
 printf("\n Enter name :");
 scanf("%s",x.name);
 printf("\n Enter roll no. and Marks : ");
 scanf("%d%d",&x.rollno,&x.marks);
 return x;
}
void print(student x)
{
 printf("\n%s\t%d\t%d",x.name,x.rollno,x.marks);
}
void main()
{ int op,op1;
 student x;
 node *head=NULL;
 clrscr();
 do
 { flushall();
   printf("\nSelect Option
    :\n1)Create\n2)Insert\n3)Delete\n4)Modify");
   printf("\n5)Display forward\n6)Display
backward\n7)Quit");
   printf("\nEnter your Choice: ");
   scanf("%d",&op);
   switch(op)
   { case 1:head=create();break;
    case 2:printf("\n\t1)Beginning\n\t2)End\n\t3)In
between");
       printf("\nEnter your choice : ");
       scanf("%d",&op1);
       printf("\nEnter the data to be inserted : ");
       flushall();
       x=read();
       switch(op1)
       { case 1: head=insert_b(head,x);
            break;
        case 2: head=insert_e(head,x);
            break;
        case 3: head=insert_in(head, x);
            break;
       }
      break;
    case 3: head = delete_in(head);
      break;
    case 4:modify(head);break;
    case 5:displayforward(head);break;
    case 6:displaybackward(head);break;
   }
 }while(op!=7);
}
node *create()
{ node *head,*p;
student x;
 int n,i;
 head=NULL;
 printf("\n Enter no. of students : ");
 scanf("%d",&n);
 printf("\n Enter records of students :");
 flushall();
 for(i=1;i<=n;i++)
 {
  x=read();
  if(head==NULL)
   {
     head=p=(node*)malloc(sizeof(node));
     head->next=head->prev=NULL;
```

```
    }
   else
    {
        p->next=(node*)malloc(sizeof(node));
        p->next->prev=p;
        p=p->next;
        p->next=NULL;
    }
  p->data=x;
 }
return(head);
}
node *insert_b(node *head,student  x)
{  node *p;
  p=(node*)malloc(sizeof(node));
  p->data=x;
 if(head==NULL)
    {
        head=p;
        head->next=head->prev=NULL;
    }
 else
    {
        p->next=head;
        head->prev=p;
        p->prev=NULL;
        head=p;
    }
 return(head);
}
node *insert_e(node *head,student  x)
{  node *p,*q;
  p=(node*)malloc(sizeof(node));
  p->data=x;
 if(head==NULL)
    {
        head=p;
        head->next=head->prev=NULL;
    }
 else
    {
        for(q=head; q->next != NULL ; q=q->next);
        q->next=p;
        p->prev=q;
        p->next=NULL;

    }
 return(head);
}
node *insert_in(node *head,student  x)
{  node *p,*q;
  int rollno;
  p=(node*)malloc(sizeof(node));
  p->data=x;
  printf("\Insert after which student ? : ");
  printf("\n Enter roll no. : ");
  scanf("%d",&rollno);
  flushall();
  for(q=head ; q != NULL && q->data.rollno != rollno;
q=q->next);
  if(q->data.rollno==rollno)
    {
        p->next=q->next;
        p->prev=q;
        p->next->prev=p;
        p->prev->next=p;
    }
  else
     printf("\nData not found ");
  return(head);
}
node *delete_in(node *head)
{
 node *p,*q;
 int rollno;
 if(head==NULL)
   {
    printf("\nUnderflow....Empty Linked List");
    return(head);
   }
 printf("\nEnter the roll no. of student  to be deleted : ");
 flushall();
 scanf("%d",&rollno);
 for(p=head ; p !=NULL && p->data.rollno != rollno ;
p=p->next);
  if(p->data.rollno!=rollno)
   {
      printf("\nUnderflow.....data not found");
      return(head);
   }
 if(p==head)
 {
      head=head->next;
      if(head != NULL)
          head->prev=NULL;
      free(p);
 }
 else
 {
    p->prev->next=p->next;
    p->next->prev=p->prev;
    free(p);
 }
```

```
 return(head);
}
void displayforward(node *head)
{ node *p;
 printf("\n");
 if(head != NULL)
   {
     for(p=head ;p!=NULL ; p=p->next)
         print(p->data);
   }
}
void displaybackward(node *head)
{ node *p;
 printf("\n");
 if(head != NULL)
   {  // go to the last node
     for(p=head ;p->next!=NULL ; p=p->next);
     for( ;p!=NULL ; p=p->prev)
         print(p->data);
   }
}
void modify(node *head)
{
 int rollno;
 node *p;
 printf("\Enter Roll no of the student : ");
 scanf("%d",&rollno);
 for(p=head;p!=NULL & p->data.rollno != rollno;p=p->next);
 if(p!=NULL)
   {
    printf("\Enter a new record : ");
    p->data=read();
   }
 else
  printf("Wrong roll no ....");
}
```

Q. 13 **Write a program in 'C' to implement Doubly Link-list with methods insert, delete and search.** May 2014, May 2017

Ans. :

```
/*Doubly Linked List :*/
#include <stdio.h>
#include <conio.h>
typedef struct dnode
{
  int data;
  struct dnode *next,*prev;
}dnode;
dnode *create()
{
  int i,n,x;
  dnode *head,*p,*q;
  head=NULL;
  printf("\nEnter no. of data : ");
  scanf("%d",&n);
  printf("\n enter data : ");
  for(i=1;i<=n;i++)
   {
    printf("\nEnter next data : ");
    scanf("%d",&x);
    q=(dnode*)malloc(sizeof(dnode));
    q->data=x;
    q->next=q->prev=NULL;
    if(head==NULL)
     {
      p=head=q;
     }
    else
     {
      p->next=q;
      q->prev=p;
      p=q;
     }
   }
  return(head);
}
void display(dnode *head)
 {
  printf("\n");
  for(;head!=NULL;head=head->next)
     printf("%d  ",head->data);
 }
dnode *Delete(dnode *head,int x)
 {
  dnode *p,*q;

  if(head==NULL)
   return(head);
  if(head->data==x)
   {
    p=head;
    head=head->next;
    head->prev=NULL;
    free(p);
```

```
        return(head);
      }
     p=head;
     while(p!=NULL && p->data !=x)
       p=p->next;
     if(p!=NULL)
      {
       if(p->next==NULL)
        {
         p->prev->next=NULL;
         free(p);
        }
       else
        {
         p->prev->next=p->next;
         p->next->prev=p->prev;
         free(p);
        }
      }
    return(head);
 }
int search(dnode *head, int x)
 {
  while(head!=NULL)
   {
    if(head->data==x)
      return (1);
      head=head->next;
   }
  return(0);
 }
void main()
 {
  dnode *head;
  int x;
  head=create();
  printf("\nEnter the data to be searched : ");
  scanf("%d",&x);
  if(search(head,x))
   printf("\nfound");
  else
   printf("\nNot found");
  printf("\nEnter the data to be deleted : ");
  scanf("%d",&x);
  head=Delete(head,x);
  printf("\nLinked list after deletion : ");
  display(head);
  getch();
 }
```

Chapter 4 : Trees

Q. 1 Define traversal of binary tree. Explain different types of traversals of Binary tree with example. May 2015

Ans. : Binary Tree Traversal :

Most of the tree operations require traversing a tree in a particular order. Traversing a tree is a process of visiting every node of the tree and exactly once. Since, a binary tree is defined in a recursive manner, tree traversal too could be defined recursively. For example, to traverse a tree, one may visit the root first, then the left subtree and finally traverse the right subtree. If we impose the restriction that left subtree is visited before the right subtree then three different combination of visiting the root, traversing left subtree, traversing right subtree is possible.

1. Visit the root, traverse, left subtree, traverse right subtree.
2. Traverse left subtree, visit the root, traverse right subtree.
3. Traverse left subtree, traverse right subtree, visit the root.

Types of traversals of binary tree :

These three techniques of traversal are known as preorder, inorder and postorder traversal of a binary tree.

1. Preorder Traversal (Recursive)

The functioning of preorder traversal of a non-empty binary tree is as follows :

1. Firstly, visit the root node (visiting could be as simple as printing the data stored in the root node).
2. Next, traverse the left subtree in preorder.
3. At last, traverse the right-subtree in preorder.

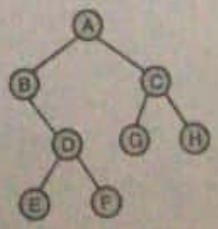
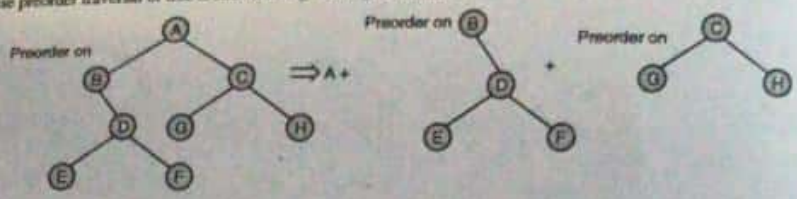


Fig. 4.1 : A sample binary tree

Stepwise preorder traversal of tree is shown in Fig. 4.2 is given below :



⇒ A + (B + Preorder on D) + (C + Preorder on G + Preorder on H)

⇒ A + (B + (D + Preorder on E + Preorder on F)) + (CGH)

⇒ A + (BDEF) + (CGH)

⇒ ABDEFCGH

Fig. 4.2

'C' Function for Preorder Traversal

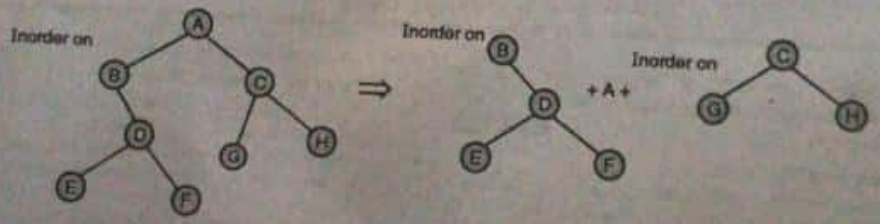
```
void preorder (node * T)
            /* address of the root node is passed in T */
{
      if (T! = NULL)
      {
            printf ("\n%d", T → data);   /* visit the root */
            preorder (T → left);
              /* preorder traversal on left subtree */
            preorder (T → right);
              /* preorder traversal on right subtree */
      }
}
```

2. Inorder Traversal (Recursive) :

The functioning of inorder traversal of a non-empty binary tree is as follows :

1. Firstly, traverse the left subtree in inorder.
2. Next, visit the root node.
3. At last, traverse the right subtree in inorder.

Stepwise inorder traversal of tree shown in Fig. 4.3.



= (B + Inorder on D) + A + ((Inorder on G) + C + (Inorder on H))

= (B + ((Inorder on E) + D + ((Inorder on F)) + A + (GCH)

= BEDFAGCH

Fig. 4.3

'C' Function for Inorder Traversal :

```
void inorder (node *T)
                    /* address of the root node is passed in T */
{
    if (T! =NULL)
      {
            inorder (T → left);
              /* inorder traversal on left subtree */
            printf ("\n%d", T → data);   /* visit the root */
            inorder (T → right);
                    /* inorder traversal on right subtree */
      }
}
```

3. Postorder Traversal (Recursive) :

The functioning of postorder traversal of a non-empty binary tree is as follows :

1. Firstly, traverse the left subtree in postorder.
2. Next, traverse the right subtree in postorder.
3. At last, visit the root node.

Stepwise postorder traversal of tree shown in Fig. 4.4 is given below.

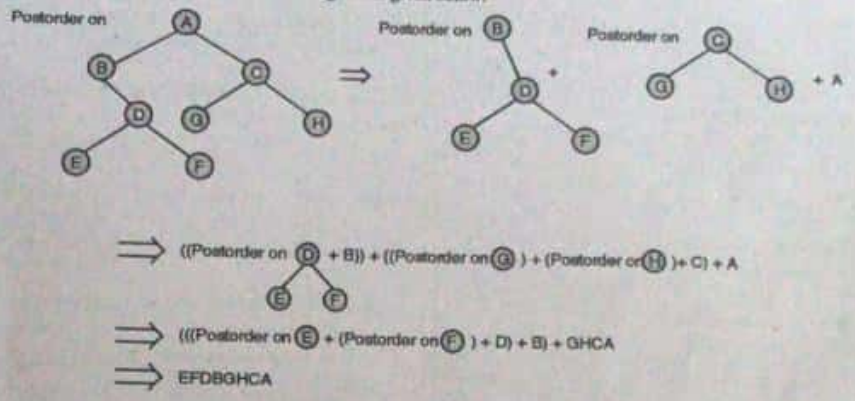


Fig. 4.4

'C' Function for Postorder Traversal :

```
void postorder (node * T)          /* address of the root node
passed in T */
{
    if (T! = NULL)
    {
        postorder (T → left);
                    /* postorder traversal on left subtree */
        postorder (T → right);
                /* postorder traversal on right subtree */
        printf ("\n%d", T → data);   /* visit the root */
    }
}
```

Q. 2 **Construct binary tree for the pre order and inorder traversal sequences :**

Preorder :	A	B	D	G	C	E	H	I	F
Inorder :	D	G	B	A	H	E	I	C	F

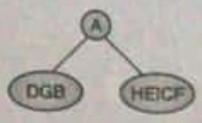
May 2014, Dec. 2016

Ans. :

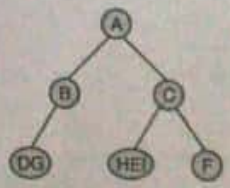
Preorder sequence : A B D G C E H I F

Inorder sequence : D G B A H E I C F

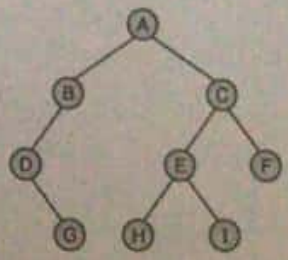
Step 1 :



Step 2 :



Step 3 :



Q. 3 **Write a program using object oriented programming using C++ to create a binary tree if inorder & preorder.** Dec. 2016

Ans. :

```
#include<iostream.h>
#include<string.h>
struct node
{ node *left,*right;
  char data;
};
class tree
{
  node *root;

  void inorder1(node *);
  void preorder1(node *);
  node *createpreorder(char *,char *);
                    //create a tree from preorder+inorder
  void dividepre(char *pre , char *in ,char *pre1,char
*pre2,char *in1,char *in2);
  public:
  void createpre(char *pre, char *in ) {
root=createpreorder(pre,in);}

  void preorder() { preorder1(root);}
  void inorder(){inorder1(root);}

};
int belongs(char x, char a[]);      //find whether the character
                                    x is in a[]

void tree::preorder1(node *p)
 {
  if(p!=NULL)
    {
      cout<<p->data;
      preorder1(p->left);
      preorder1(p->right);
    }
 }
void tree::inorder1(node *p)
 {
  if(p!=NULL)
   {
     inorder1(p->left);
     cout<<p->data;
     inorder1(p->right);
   }
}


node*  tree::createpreorder(char *pre, char *ino)
 {
  // preorder and inorder are preorder and inorder sequences
respectively.
    char pre1[20],pre2[20],in1[20],in2[20];
    node *p=NULL;
   //preorder sequence gives the root
   if(strlen(pre)==0)
     return(NULL);
   p=new node;
   p->data=pre[0];
   //divide the preorder and inorder sequence for left and
right subtrees.
   dividepre(pre,ino,pre1,pre2,in1,in2);
   p->left=createpreorder(pre1,in1);
   p->right=createpreorder(pre2,in2);
   return(p);
 }
void tree::dividepre(char *pre , char *in ,char *pre1,char
*pre2,char *in1,char *in2)
 {
    int i,j,k;
   for(i=0;in[i]!=pre[0];i++)        // left subtree, inorder
       in1[i]=in[i];
   in1[i]='\0';
   i++;
   for(j=0;in[i+j]!='\0';j++)   // right subtree, inorder
       in2[j]=in[i+j];
   // divide the preorder sequence
   in2[j]='\0';
   i=j=0;
   for(k=1;pre[k]!='\0';k++)
       if(belongs(pre[k],in1))     // belongs to left subtree
           pre1[i++]=pre[k];
       else
           pre2[j++]=pre[k]; // belongs to right subtree
   pre1[i]=pre2[j]='\0';
 }


int belongs(char x, char a[])
 { int i;
  for(i=0;a[i]!='\0';i++)
   if(a[i]==x)
    return(1);
  return(0);
 }
void main()
{
  char pre[20],in[20],post[20];
  tree T;
  cout<<"\n\n\nCreate a tree from preorder and inoder
sequence:\n";
  cout <<"\nEnter Preorder Sequence : ";
  cin>>pre;
```

```
cout<<"\nEnter inorder sequence : ";
cin >> in;
T.createpre(pre,in);
cout<<"\n\nPreorder on the tree :";
T.preorder();
cout<<"\nInorder on the tree : ";
T.inorder();
cout<<"\nEnter inorder sequence : ";
cin >> in;
T.createpost(post,in);
cout<<"\nPreorder on the tree :";
T.preorder();
cout<<"\nInorder on the tree : ";
T.inorder();

}
```

Output

```
Create a tree from preorder and inoder sequence:
Enter Preorder Sequence : -+a*bcd
Enter inorder sequence : a+b*c-d
Preorder on the tree :-+a*bcd
Inorder on the tree : a+b*c-d
Postorder on the tree :abc*+d-
```

Q. 4 What is an binary search tree ? Explain with an example. May 2015

Ans. : Binary Search Tree (BST)

A binary search tree is a binary tree, which is either empty or in which each node contains a key that satisfies the following conditions :

(1) All keys are distinct.

(2) For every node, X, in the tree, the values of all the keys in its left subtree are smaller than the key value in X.

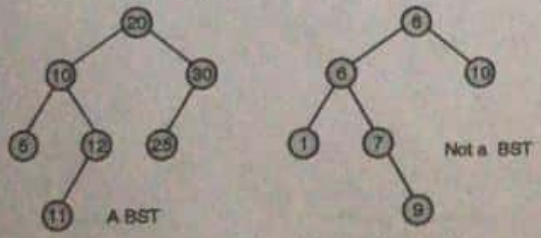


Fig. 4.5 : Left tree is a BST, right tree is not a binary search tree

(3) For every node, X, in the tree, the values of all the keys in its right subtree are larger than the key value in X.

Binary search tree finds its application in searching. In Fig. 4.5, the tree on the left is a binary search tree. Tree on the right is not a BST. Left subtree of the node with key 8 has a key with value 9.

Operations on a Binary Search Tree

(1)	Initialise	(2)	Find	(3)	Makeempty
(4)	Insert	(5)	Delete	(6)	Create
(7)	Findmin	(8)	Findmax		

Structure of node of a binary search tree

```
typedef struct BSTnode
{
    int data;
    struct BSTnode *left, *right;
} BSTnode;
```

1. Initialize Operation

Initially the tree is empty and hence the referencing pointer root should be set to NULL.

'C' function for initialise

```
BSTnode* initialize()
{
    return(NULL) ;
}
```

2. Find Operation

It is often required to find whether a key is there in the tree. If the key, X, is found in the node T, the function returns the address of T or NULL if there is no such node.

Recursive algorithm for find

return value	condition
Find (root, x) NULL	if root == NULL
root	if root → data == x
return (Find (root → right, x))	if x > root → data
return (Find (root → left, x))	if x < root → data

If the key, X, is found to be larger than the value stored in node T, we make a recursive call to function with the right subtree. Otherwise, we make a recursive call to function with the left subtree.

'C' Function for find() Recursive

```
BSTnode *find(BSTnode * root ,int x)
{
    if((root==NULL)
        return(NULL);
    if(root->data==x)
        return(root);
    if(x>root->data)
        return(find(root->right),x));
    return(find(root->left),x));
}
```

3. Make Empty Operation

This function deletes every node of the tree. It also releases the memory acquired by the node.

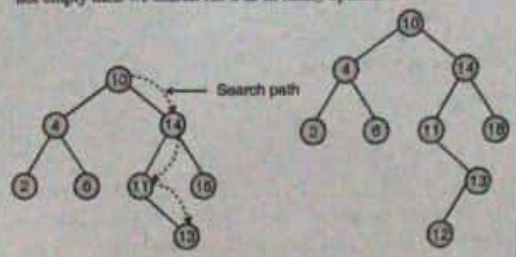
'C' Function to Release Memory :

```
BSTnode *makeempty (BSTnode *root)
{
    if (root ! = NULL)
        {
            makeempty (root → left);
            makeempty (root → right);
            free (root);
```

```
    }
    return (NULL);
}
```

4. Insert Operation

The function insert (T, x), adds element x to an existing binary search tree. T is tested for NULL, if so, a new node is created to hold x. T is made to point to the new node. If the tree is not empty then we search for x as in find() operation.



(a) Before insertion of the new key 12 **(b) After inserting 12**

Fig. 4.6 : Insertion operation into a binary search tree

If x is already there in the then insert() operation terminates without insertion as a BST is not allowed to have duplicate keys. If we find a NULL pointer during the find() operation. We replace it by a pointer to a new node holding x. Fig. 4.6 shows the insert operation.

'C' Function for insert() – Recursive :

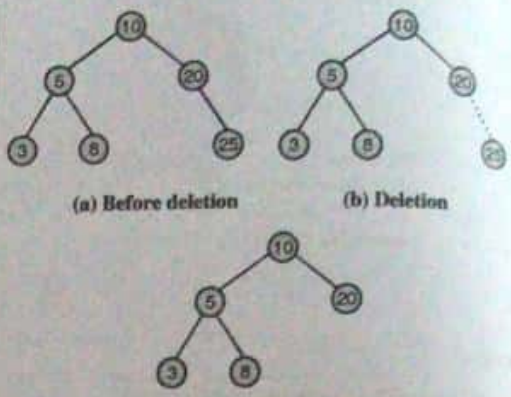
```
BSTnode *insert(BSTnode *T,int x)
{
    if(T==NULL)
    {
        T=(BST *)malloc(sizeof(BSTnode));
        T->data=x;
        T->left=NULL;
        T->right=NULL;
        return(T);
    }
    if(x > T->data)          // insert in right subtree
    {
        T->right=insert(T->right,x);
        return(T);
    }
    T->left=insert(T->left,x);   //insert in left subtree
    return(T);
}
```

5. Delete Operation

In order to delete a node, we must find the node to be deleted. The node to be deleted may be :

(a) A leaf node
(b) A node with one child
(c) A node with two children.

If the node is a leaf node, it can be deleted immediately by setting the corresponding parent pointer to NULL. For example, consider the tree of Fig. 4.7. Assume that the node (25) is to be deleted. node (25) is a right child of its parent node (20). Right child of node (20) is set to NULL.



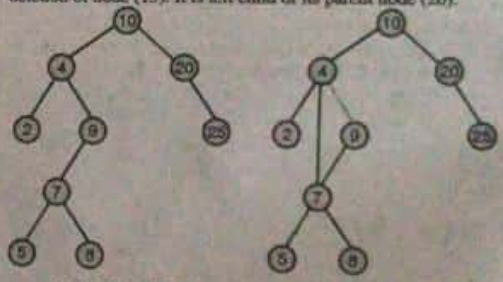
(a) Before deletion **(b) Deletion**

(c) After deletion

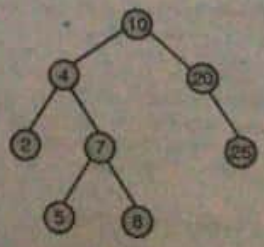
Fig. 4.7 : Deletion of node (25), a leaf node

Even when the node to be deleted has one child, it can be deleted easily. If a node q is to be deleted and it is right child of its parent node p. The only child of q will become the right child of p after deletion of q. Similarly, if a node q is to be deleted and it is left child of its parent node p. The only child of q will become the left child of p after deletion.

For example, consider the tree of Fig. 4.8. Assume that node (9) is to be deleted. Since node (9) is a right child of its parent node (4). The only child subtree of node (9), with node (7) will become the right subtree of node (4) after deletion. Fig. 4.9 shows deletion of node (15). It is left child of its parent node (20).

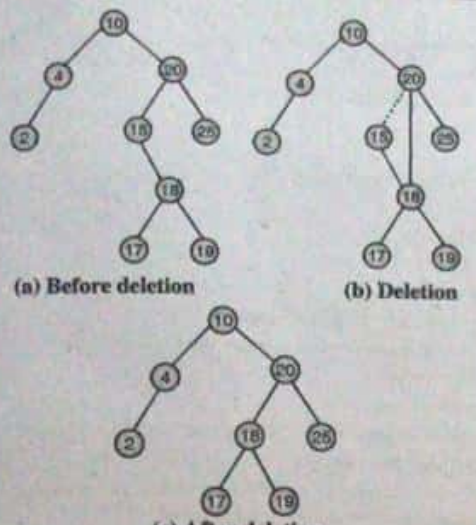


(a) Before deletion **(b) Deletion**



(c) After deletion

Fig. 4.8 : Deletion of a node (9) with one child

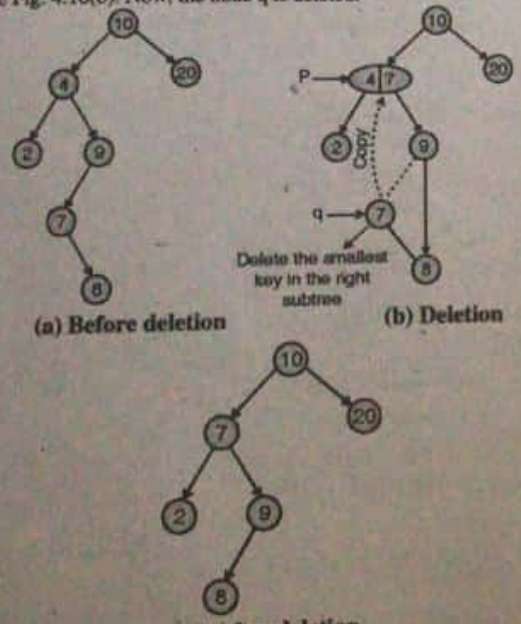


(a) Before deletion (b) Deletion

(c) After deletion

Fig. 4.9 : Deletion of a node (15) with one child

The case in which the node to be deleted has two children is a bit complicated. The general strategy is to replace the data of this node with the smallest data of the right subtree (inorder successor) and then delete the smallest node in the right subtree. The smallest node in right subtree will either be a leaf node ora node of degree 1. As a first step, the node with smallest value in the right subtree of P (address of node (4)) is found and its address is stored in q. Content of node q is copied in node P. As a second step, the node q is deleted, a node with one child. For example, consider the tree of Fig. 4.10. Assume that node (4) is to be deleted. Node (7) is the smallest node in the right subtree of node (4). Value 7 is copied in the node P (earlier node (4)) as shown in the Fig. 4.10(b). Now, the node q is deleted.



(a) Before deletion (b) Deletion

(c) After deletion

Fig. 4.10 : Deletion of a node (4) with two children

'C' Function for Deletion

```
BSTnode *delete(BSTnode *T,int x)
{
    if(T==NULL)
    {

    printf("\nElement not found :");
        return(T);
    }
    if(x < T->data)          // delete in left subtree
    {
        T->left=delete(T->left,x);
        return(T);
    }
    if(x > T->data)          // delete in right subtree
    {
        T->right=delete(T->right,x);
        return(T);
    }
    // element is found
    if(T->left==NULL && t->right==NULL)
                                    // a leaf node
    {
        temp=T;
        free(temp);
        return(NULL);
    }
    if(T->left==NULL)
    {
        temp=T;
        T=T->right;
        free(temp);
        return(T);
    }
    if(T->right==NULL)
    {
        temp=T;
        T=T->left;
        free(temp);
        return(T);
    }
    // node with two children
    temp=find_min(T->right);
    T->data=temp->data;
    T->right=delet(T->right, temp ->data);
    return(T);
}
```

6. Create :

A binary search tree can be created by making repeated calls to insert operation.

'C' Function for Tree Creation

```
BSTnode *create()
{
    int n,x,i;
    BSTnode *root;
    root=NULL;
    printf("\nEnter no. of nodes :");
    scanf("%d",&n);
    printf("\nEnter tree values :");
    for(i=0;i<n;i++)
    {
        scanf("%d",&x);
        root=insert(root,x);
    }
    return(root);
}
```

7. Find Min :

This function returns to address of the node with smallest value in the tree.

'C' function for finding the smallest value is a BST.

```
BSTnode *findmin (BSTnode *T)
{
        while (T→ left ! = NULL)
            T = T → left ;
        return (T);
}
```

8. Find Max

This function returns the address of the node with largest value in the tree.

'C' function for finding the largest value in a BST :

```
BSTnode *findmax (BSTnode * T)
{
        while (T → right ! = NULL)
            T = T → right ;
        return (T);
}
```

Q. 5 Write a Program in C for various operations on a binary search tree. Dec. 2013, May 2016

Ans. :

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
typedef struct BSTnode
{
    int data;
    struct BSTnode *left,*right;
}BSTnode;

BSTnode *initialise();
BSTnode *find(BSTnode *,int);
BSTnode *insert(BSTnode *,int);
BSTnode *delet(BSTnode *,int);
BSTnode *find_min(BSTnode *);
BSTnode *find_max(BSTnode *);
BSTnode *create();
void inorder(BSTnode *T);
void main()
{
    BSTnode *root,*p;
    int x;
    clrscr();
    initialise();
    root=create();
    printf("\n**** BST created ****");
    printf("\ninorder traversal on the tree ");
    inorder(root);
    p=find_min(root);
    printf("\n smallest key in the tree = %d",p->data);
    p=find_max(root);
    printf("\nlargest key in the tree = %d",p->data);
    printf("\n **** delete operation ****");
    printf("\nEnter the key to be deleted :");
    scanf("%d",&x);
    root=delet(root,x);
    printf("inorder traversal after deletion :");
    inorder(root);
}
void inorder(BSTnode *T)
{
    if(T!=NULL)
    {
        inorder(T->left);
        printf("%5d",T->data);
        inorder(T->right);
    }
}
BSTnode *initialise()
{
    return(NULL);
}
BSTnode *find(BSTnode *root,int x)
{
    while(root!=NULL)
    {
        if(x==root->data)
            return(root);
        if(x>root->data)
            root=root->right;
        else
        root=root->left;
    }
    return(NULL);
}
```

```
BSTnode *insert(BSTnode *T,int x)
{
    BSTnode *p,*q,*r;
    // acquire memory for the new node
    r=(BSTnode*)malloc(sizeof(BSTnode));
    r->data=x;
    r->left=NULL;
    r->right=NULL;
    if(T==NULL)
        return(r);
    // find the leaf node for insertion
    p=T;
    while(p!=NULL)
    {
        q=p;
        if(x>p->data)
            p=p->right;
        else
            p=p->left;
    }
    if(x>q->data)
        q->right=r;  // x as right child of q
    else
        q->left=r;  //x as left child of q
    return(T);
}
BSTnode *delet(BSTnode *T,int x)
{
    BSTnode *temp;
    if(T==NULL)
    {
        printf("\nElement not found :");
        return(T);
    }
    if(x < T->data)          // delete in left subtree
    {
        T->left=delet(T->left,x);
        return(T);
    }
    if(x > T->data)          // delete in right subtree
    {
        T->right=delet(T->right,x);
        return(T);
    }
    // element is found
    if(T->left==NULL && T->right==NULL)
                                    // a leaf node
    {
        temp=T;
        free(temp);
        return(NULL);
    }
    if(T->left==NULL)
    {
        temp=T;
        T=T->right;
        free(temp);
        return(T);
    }
    if(T->right==NULL)
    {
        temp=T;
        T=T->left;
        free(temp);
        return(T);
    }
    // node with two children
    temp=find_min(T->right);
    T->data=temp->data;
    T->right=delet(T->right,x);
    return(T);
}
BSTnode *create()
{
    int n,x,i;
    BSTnode *root;
    root=NULL;
    printf("\nEnter no. of nodes :");
    scanf("%d",&n);
    printf("\nEnter tree values :");
    for(i=0;i<n;i++)
    {
        scanf("%d",&x);
        root=insert(root,x);
    }
    return(root);
}
BSTnode *find_min(BSTnode *T)
{
    while(T->left!=NULL)
        T=T->left;
    return(T);
}
BSTnode *find_max(BSTnode *T)
{
    while(T->right!=NULL)
        T=T->right;
    return(T);
}
```

Output :

```
Enter no. of nodes : 5
Enter tree values : 34  11   2    99  6
**** BST created ****
inorder traversal on the tree   2  6  11  34  99
smallest key in the tree = 2
```

```
largest key in the tree = 99
**** delete operation ****
Enter the key to be deleted :11
inorder traversal after deletion :  2  6  34  99
```

Q. 6 **Write a program in 'C' to implement Binary search on sorted set of integers.** May 2014

Ans. :

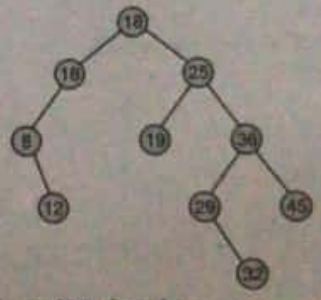
```
//*****To implement binary search :*****//

#include <stdio.h>
#include <conio.h>

//int stepcount=0,swapcount=0,compcount=0;

int binsearch(int a[],int i ,int j,int key);//Recursive
void main()
 {
  int a[30],key,n,i,result;
  clrscr();
  printf("\n Enter No. of elements : ");
  scanf("%d" ,&n);
  printf("\n Enter a sorted list of %d elements : ",n);
  for(i=0;i<n;i++)
     scanf("%d",&a[i]);
  printf("\n Enter the element to be searched : ");
  scanf("%d",&key);
  result=binsearch(a,0,n-1,key);
  if(result==-1)
     printf("\n Not found ");
  else
     printf("\n Found at location= %d",result+1);

/* printf("\n No. of steps= %d",stepcount);
  printf("\n No. of swaps= %d", swapcount);
  printf("\n No. of comparisons= %d",compcount);*/
  getch();
 }

int binsearch(int a[],int i, int j,int key)
{
 int c;
 if(i>j)
    {
    // compcount++;
     //stepcount+=2;
     return(-1);
    }
 c=(i+j)/2;
 if(key==a[c])
    {
     compcount++;
     stepcount+=2;
     return(c);
    }
 if(key>a[c])
    {
    // stepcount+=1;
     return(binsearch(a,c+1,j,key));
    }
// stepcount+=1;
 return(binsearch(a,i,c-1,key));
}
```

Output :

```
Enter No. of elements : 5

Enter a sorted list of 5 elements : 2
4
1
9
7

Enter the element to be searched : 9
Found at location= 4
```

Q. 7 **Consider the following list of numbers : 18, 25, 16, 36, 08, 29, 45, 12, 32, 19**
Create a binary search tree using these numbers and display them in a non-decreasing order. Write a 'C' program for the same. Dec. 2014

Ans. : Binary search tree of the given numbers



List in non-decreasing order
8 12 16 18 19 25 29 32 36 45

Program

```
#include <stdio.h>
#include <stadlib.h>
typedef struct BSTnode
    {
        int data;
        struct BSTnode * left, * right;
    } BSTnode;
BSTnode *insert (BSTnode *, int);
```

```
BSTnode * create();
void inorder (BSTnode * T)  ;
void main ()
    {
        BSTnode  * root;
        root = create ();
        inorder (root);
    }
void inorder (BSTnode * T)
    {
    if(T!=NULL)
    {
        inorder(T->left);
        printf("%5d",T->data);
        inorder(T->right);
    }
}
BSTnode *insert(BSTnode *T,int x)
{
    BSTnode *p,*q,*r;
    // acquire memory for the new node
    r=(BSTnode*)malloc(sizeof(BSTnode));
    r->data=x;
    r->left=NULL;
    r->right=NULL;
    if(T==NULL)
        return(r);
    // find the leaf node for insertion
    p=T;
    while(p!=NULL)
    {
        q=p;
        if(x>p->data)
            p=p->right;
        else
            p=p->left;
    }
    if(x>q->data)
        q->right=r;  // x as right child of q
    else
        q->left=r;   //x as left child of q
    return(T);
}
BSTnode *create()
{
    int n,x,i;
    BSTnode *root;
    root=NULL;
    printf("\nEnter no. of nodes :");
    scanf("%d",&n);
    printf("\nEnter tree values :");
    for(i=0;i<n;i++)
    {
        scanf("%d",&x);
        root=insert(root,x);
    }
    return(root);
}
```

Q. 8 Discuss AVL trees.

Dec. 2013, May 2015, May 2017

Ans. : AVL Trees

An AVL (Adelson-Velskii and Landis) tree is a height balance tree. These trees are binary search trees in which the heights of two siblings are not permitted to differ by more than one.

i.e. |Height of the left subtree – height of the right subtree| ≤ 1

Searching time in a binary search tree is O(h), where h is the height of the tree. For efficient searching, it is necessary that height should be kept to minimum. A full binary search tree with n nodes will have a height of $O(\log_2 n)$. In practice, it is very difficult to control the height of a BST. It lies between O(n) to $O(\log_2 n)$. An AVL tree is a **close approximation** of full binary search tree.

Height Balanced Tree

An empty tree is height balanced. A binary tree with h_l and h_r as height of left and right subtree respectively is height balanced if $|h_l - h_r| \leq 1$. A binary tree is height balanced if every subtree of the given tree is height balanced.

An AVL tree is a height balanced binary search tree.

Balance Factor

The balance factor, BF(T) of a node T in a binary tree is defined as $h_l - h_r$ where h_l and h_r are the heights of the left and the right subtrees of T. A binary search tree with balance factors is shown in the Fig. 4.11.

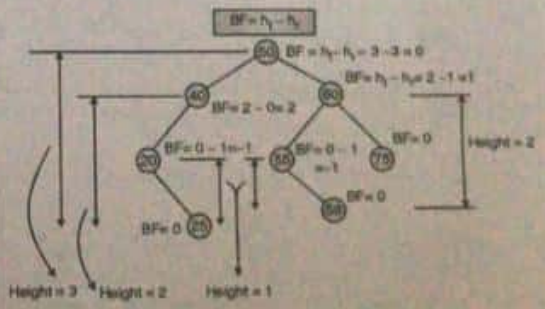


Fig. 4.11 : A sample BST with balance factors

Tree of Fig. 4.11 is not an AVL tree. The balance factor of the node with data 40 is +2. Trees of Fig. 4.12 are non-AVL trees. Trees of Fig. 4.13 are AVL-trees.

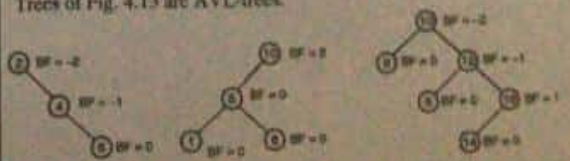


Fig. 4.12 : Non-AVL trees

The balance factor of a node in an AVL tree could be – 1, 0 or 1. If the balance factor of a node is 0 then the heights of the left and right subtrees are equal. If the balance factor of a node is +1 then the height of the left subtree is one more than the height of the right subtree. If the balance factor of a node is –1 then the height of the left subtree is one less than the height of the right subtree.

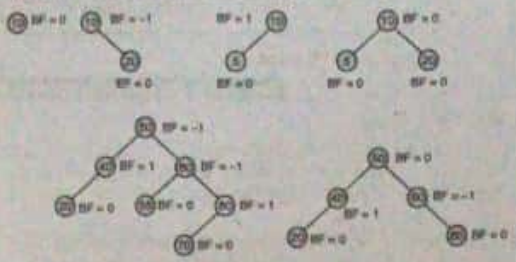


Fig. 4.13 : AVL trees

Structure of a Node in AVL Tree

Operations an AVL tree requires calculation of balance factors of nodes. To represent a node in AVL tree, a new field is introduced. This new field stores the height of the node. A node in AVL tree can be defined in the following way :

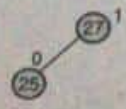
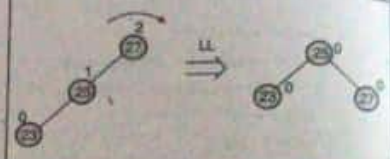
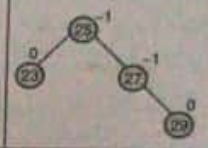
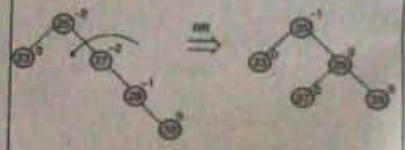
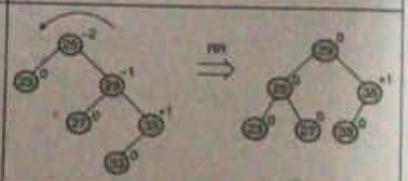
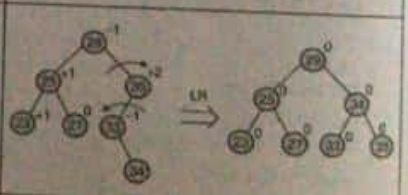
```
typedef struct node
    {
        int data ;
            struct node *left, *right;
            int height;
    } node ;
```

'C' Function for Finding the Balance Factor of a Node

```
int BF(node *T)
{
    int lh,rh;
    if(T->left==NULL)
        lh=0;
    else
        lh=1+T->left->height;
    if(T->right==NULL)
        rh=0;
    else
        rh=1+ T->right->height;
    return(lh-rh);
}
/* the function height(),for finding the height of node is given
below */
int height(node *T)
{
    int lh,rh;
    if(T->left==NULL)
        lh=0;
    else
        lh=1+ T->left->height
    if(T->right==NULL)
        rh=0;
    else
        rh=1+ T->right->height
    if(lh>rh)
        return(lh);
    return(rh);
}
```

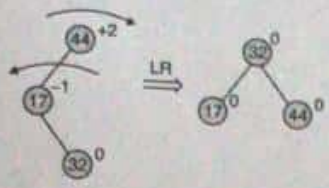
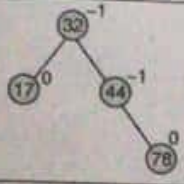
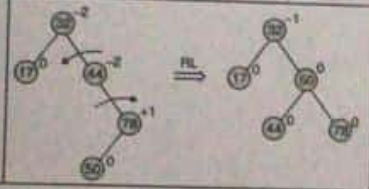
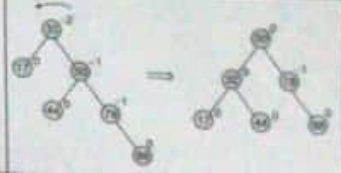
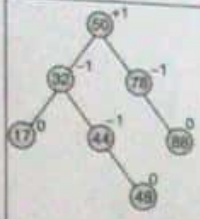
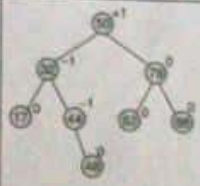
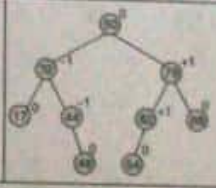
Q. 9 **Insert the following elements in a AVL search tree : 27, 25, 23, 29, 35, 33, 34.** Dec. 2012

Ans. :

Sr. No.	Data	Tree after insertion	Tree after rotation
1.	27		
2.	25		
3.	23		
4.	29		
5.	35		
6.	33		
7.	34		

Q. 10 **Insert the following elements in AVL tree : 44, 17, 32, 78, 50, 88, 48, 62, 54. Explain the different rotations that will be used.** Dec. 2014

Ans. :

Sr. No.	Data to be inserted	Tree after insertion	Tree after rotation
1.	44		
2.	17, 32		
3.	78		
4.	50		
5.	88		
6.	48		
7.	62		
8.	54		

Q. 11 **Construct AVL tree for the following data : 50, 25, 10, 5, 7, 3, 30, 20, 8, 15.** May 2015

Ans. : AVL Trees

An AVL (Adelson-Velskii and Landis) tree is a height balance tree. These trees are binary search trees in which the heights of two siblings are not permitted to differ by more than one.

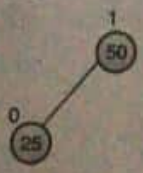
i.e. |Height of the left subtree – height of the right subtree| ≤ 1

Searching time in a binary search tree is O(h), where h is the height of the tree. For efficient searching, it is necessary that height should be kept to minimum. A full binary search tree with n nodes will have a height of $O(\log_2 n)$. In practice, it is very difficult to control the height of a BST. It lies between O(n) to $O(\log_2 n)$. An AVL tree is a **close approximation** of full binary search tree.

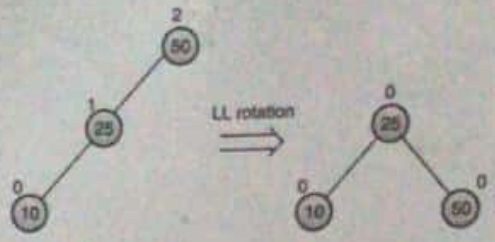
AVL Tree

Insert 50

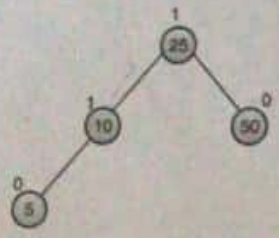
Insert 25



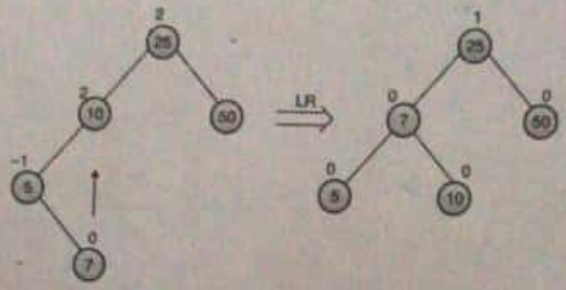
Insert 10



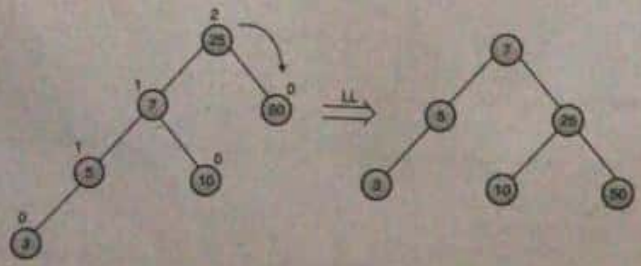
Insert 5



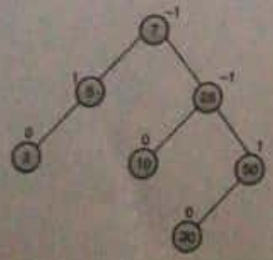
Insert 7



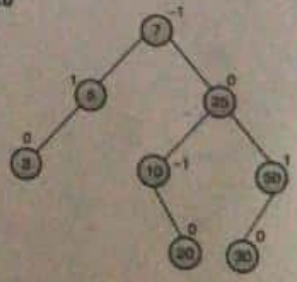
Insert 3



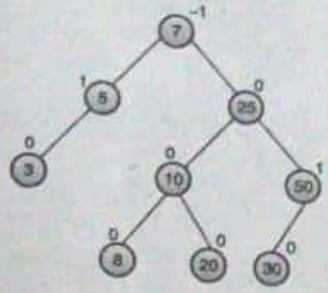
Insert 30



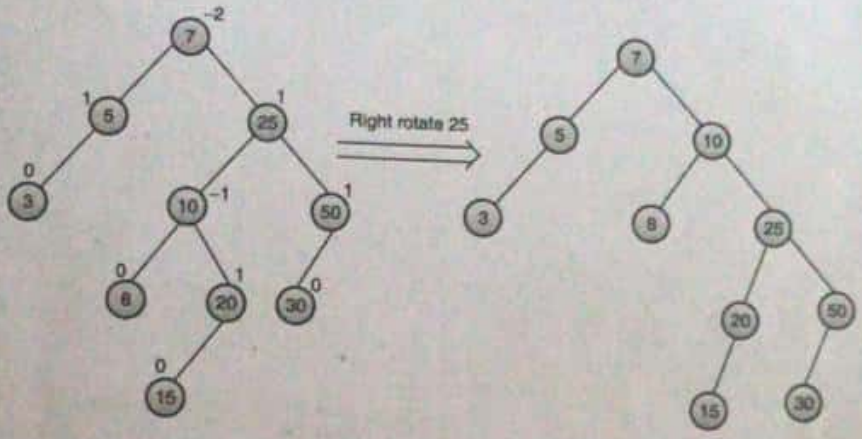
Insert 20

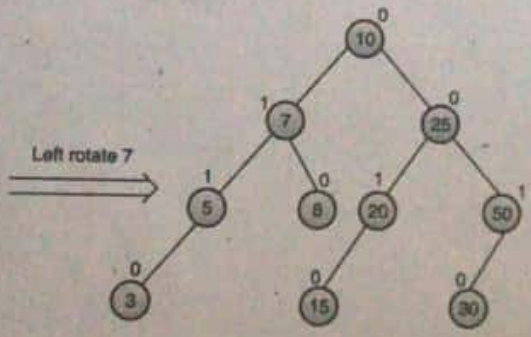


Insert 8



Insert 15

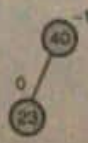


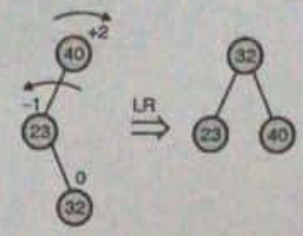
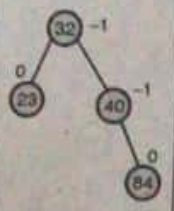
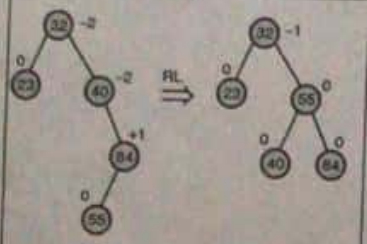
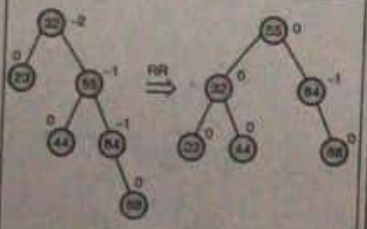
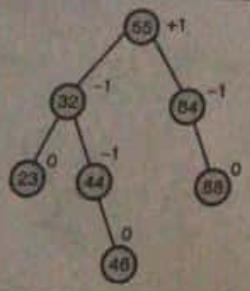


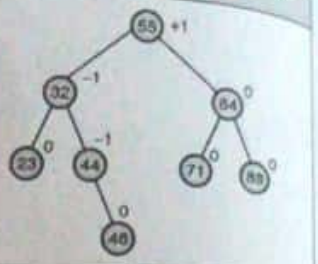
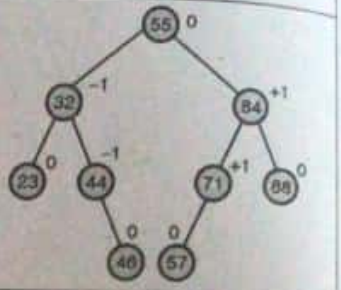
Q. 12 **Insert the following elements in a AVL search tree : 40, 23, 32, 84, 55, 88, 46, 71, 57**
Explain different rotations used in AVL trees
Dec. 2016

Ans. :

Sr. No.	Data to be inserted	Tree after insertion	Tree after rotation
1.	40		

Sr. No.	Data to be inserted	Tree after insertion	Tree after rotation
2.	23		

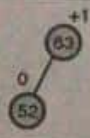
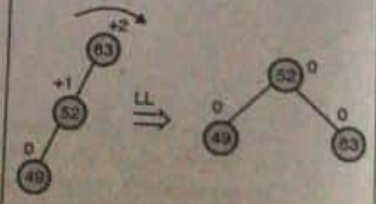
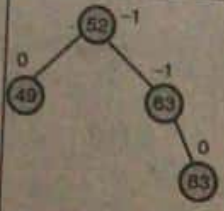
Sr. No.	Data to be inserted	Tree after insertion	Tree after rotation
3.	32		
4.	84		
5.	55		
6.	88		
7.	46		

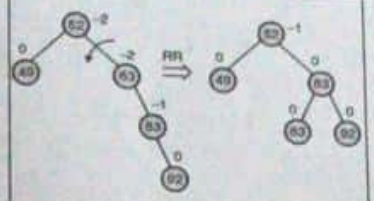
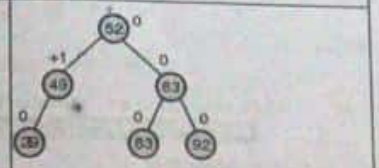
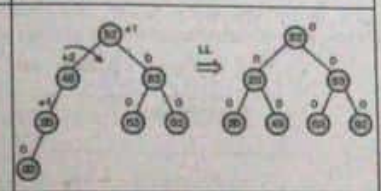
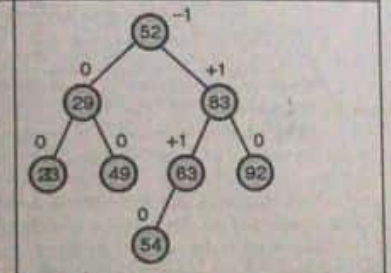
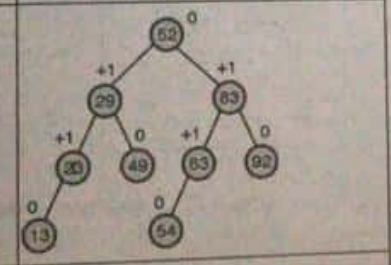
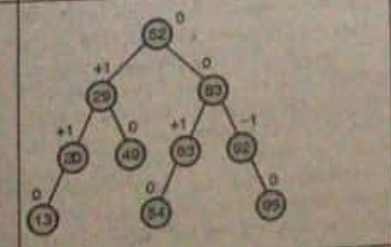
Sr. No.	Data to be inserted	Tree after insertion	Tree after rotation
8.	71		
9.	57		

Q. 13 **Insert the following elements in a AVL search tree : 63, 52, 49, 83, 92, 29, 23, 54, 13, 99.**

May 2017

Ans. :

Sr. No.	Data to be inserted	Tree after insertion	Tree after rotation
1.	63		
2.	52		
3.	49		
4.	83		

Sr. No.	Data to be inserted	Tree after insertion	Tree after rotation
5.	92		
6.	29		
7.	23		
8.	54		
9.	13		
10.	99		

Q. 14 Describe expression Tree with an example.

May 2014, May 2016, Dec. 2016, May 2017

Ans. : Expression Trees :

When an expression is represented through a tree, it is known as an expression tree. The leaves of an expression tree are operands, such as constants or variables names and all internal nodes contain operations.

Fig. 4.14 gives an example of an expression tree.

$(a + b * c) * e + f$

A preorder traversal on the expression tree gives prefix equivalent of the expression. A postorder traversal on the expression tree gives postfix equivalent of the expression.

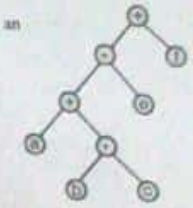


Fig. 4.14

Prefix (expression tree of Fig. 4.14) = + * + a * b c e f

Postfix (expression tree of Fig. 4.14) = a b c * + e * f +

Constructing an expression tree :

In case an expression tree is to be converted from infix expression, infix expression should be converted to postfix.

Algorithm :

We read our expression one symbol at a time. If the symbol is an operand, we create one node tree and push a pointer to it onto a stack. If the symbol is an operator, we pop pointer to two trees T_2 and T_1 from the stack and form a new tree whose root is the operator and whose left and right children point to T_1 and T_2 respectively. A pointer to this new tree is then pushed onto the stack.

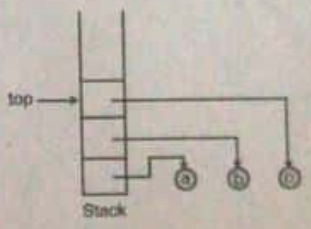


Fig. 4.15

As an example, suppose the input is = abc * + e * f +

The first three symbols are operands, so we create one-node trees push pointers to them onto a stack. Next, 4 * is read, so two pointers to tree are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

Next, a + is read, so two pointers to tree are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

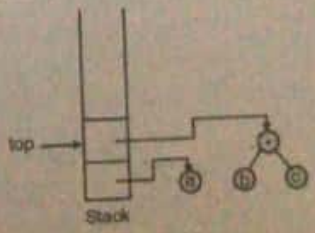


Fig. 4.16

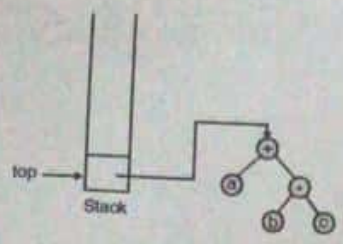


Fig. 4.17

Next, e is read, one node tree is created and a pointer to it is pushed onto the stack.

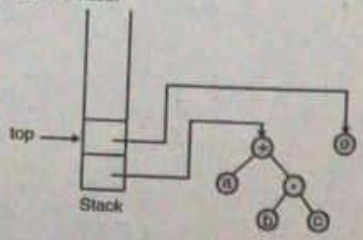


Fig. 4.18

Next, a, * is read, so two pointers to tree are popped, a new tree is formed, and a pointer to it is pushed onto the stack.

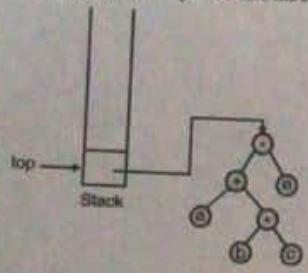


Fig. 4.19

Continuing, f is read, a one node tree is created and a pointer to it is pushed onto the stack.

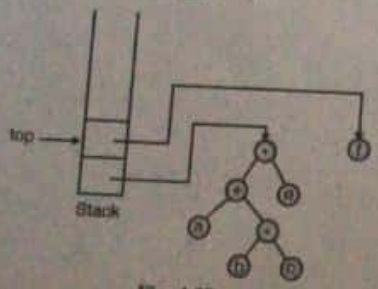


Fig. 4.20

Finally, a + is read, two trees are merged, and a pointer to the final tree is pushed on the stack.

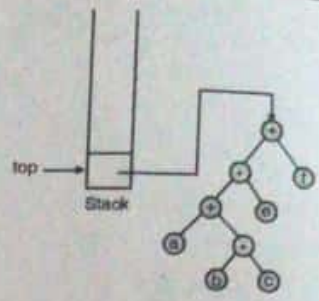


Fig. 4.21

Q. 15 Write short note on : Huffman coding.

Dec. 2013, May 2015, Dec. 2016, May 2017

Ans. : Huffman Algorithm

A text message can be converted into a sequence of 0's and 1's by replacing each character of the message with its code.

Table 4.1 : Binary code

Input symbol	Code
a	000
b	001
c	010
d	011
e	100

An input message bbca can be encoded as 001001010000 using the codes given in the Table 4.1.

Huffman coding technique is based on variable length code size. All character in a file, do not appear with the same frequency. There could be a large disparity between the most frequent and least frequent characters. A large data file may have large amount of digits, blanks and etc. but very few alphabets. A simple strategy of providing **short codes** to most frequently occurring characters can considerably reduce the size of the encoded message.

Symbol	Frequency	Code1	Code2
a	.12	000	000
b	.40	001	11
c	.15	010	01
d	.08	011	001
e	.25	100	10

Fig. 4.22 : Two binary codes

Message → b c b e b c

Encoded using code 1 → 001010001100001010 (length = 18)

Encoded using code 2 → 110111101101 (length = 12)

Encoding of the message using code1 (as given in Fig. 4.22) requires 18 bits of storage space, whereas if code 2 is used for encoding, only 12 bits will be required. Thus, there is a saving of 33% of storage space, if code 2 is used for encoding of the message "bcbebc".

Huffman code has prefix property. Prefix property allows one to decode a string of 0's and 9's by repeatedly deleting prefixes of the string that are codes for characters.

Example : Decoding 1101 11 101101 using code 2.

(a) Prefix 11 encoded as b and deleted from the message

Input	Output
01 111 011 01	b
11101101	bc[01(c) is deleted]
101101	bcb[11(b) is deleted]
1101	bcbe[10(e) is deleted]
01	bcbeb[11(b) is deleted]
–	bcbebc[01(c) is deleted]

Fig. 4.22(a) : Decoding using code 2[Prefix property]

Q. 16 Write a function to implement an Huffman coding given a symbol and its frequency.

May 2015, Dec. 2015

Ans. : Representation of binary codes as a binary tree

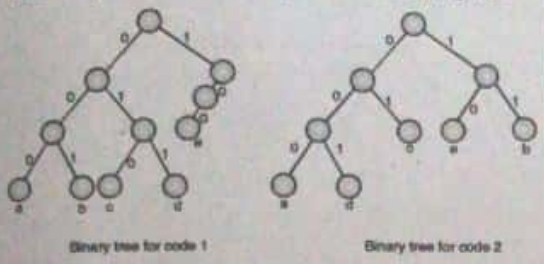


Fig. 4.23 : Binary tree representation of codes with prefix property

Tree in Fig. 4.23 has data only at leaves. Code of each character can be found by starting from the root of the tree and recording the path.0 indicates a left branch and 1 indicates a right branch. For instance 'd' using code 2 is reached by going left, then left and finally right. This is encodes as 001.

If the characters are placed only at the leaves, any sequence of bits can be decoded unambiguously (Prefix property).

Huffman's algorithm

It is an algorithm for finding the best possible prefix code for given frequency of occurrences of input alphabets in any message. Maintain a forest of trees. Each character is converted into a single node tree. Each node is labelled by its frequency (Probability). The weight of a tree is equal to the sum of the frequencies of its leaves. Select the two trees in the forest that have smallest weights. Combine these two trees into one weight of the combined tree = sum of the weights of the two trees. This process continues until one tree remains.

Q. 17 Construct the Huffman Tree and determine the code for the following characters whose frequencies are as given :

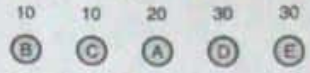
Characters	A	B	C	D	E
Frequency	20	10	10	30	30

Dec. 2013

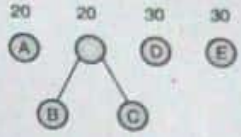
Ans. :

Construction of Huffman tree

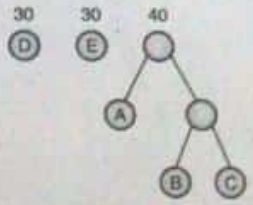
Step 1 : Each node is represented as a tree. These trees are stored in a priority queue.



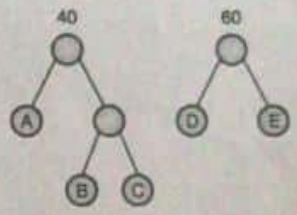
Step 2 : Merge B, C [Trees of minimum weights]



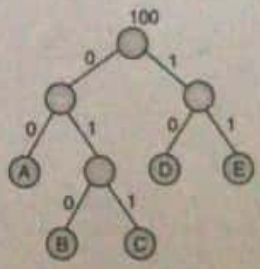
Step 3 : Merge A and (B, C)



Step 4 : Merge D, E



Step 5 : Merge the two trees



Data Item	Code
A	00
B	010
C	011
D	10
E	11

Q. 18 Apply Huffman Coding for the wor 'MALAYALAM'. Give the Huffman code f each symbol.

May 201

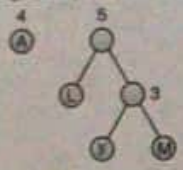
Ans. :

Data	Weight
M	2
A	4
L	2
Y	1

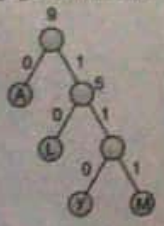
Step 1 : Initial forest of trees

Step 2 : Merging Y and M

Step 3 : Merging L and (Y, M)



Step 4 : Merging A and (L, Y, M)



Step 5 :

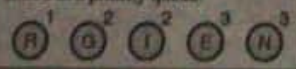
Data	Huffman Code
A	0
L	10
Y	110
M	111

Q. 19 Construct the Huffman Tree and determine the code for each symbol in the sentence "ENGINEERING". May 2017

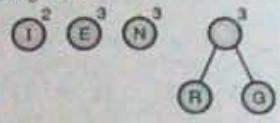
Ans. : Frequency of each letter in "ENGINEERING"

Symbol	Frequency
E	3
N	3
G	2
I	2
R	1

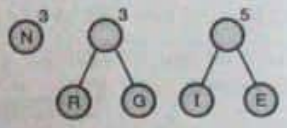
Step 1 : Each node is represented as a tree. These trees are stored in a priority queue.



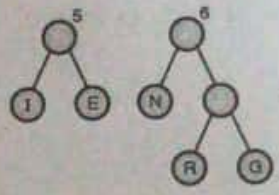
Step 2 : Merge R, G



Step 3 : Merge I, E

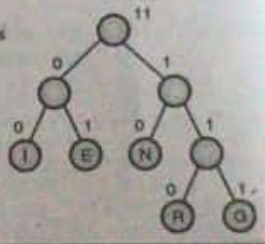


Step 4 : Merge N, (R, G)



Step 5 : Merge the two trees

Code for each symbol
I – 00
E – 01
N – 10
R – 110
G – 111



Q. 20 What is Multiway search Tree ? Explain with an example. Dec. 2014

Ans. : B-Trees

B-tree is another very popular search tree. The node in a binary tree like AVL tree contains only one record. AVL tree is commonly stored in primary memory. In database application, where huge volume of data is handled, the search tree cannot be accommodated in primary memory. B-trees are primarily meant for secondary storage.

A B-tree is a M-way tree. An M-way tree can have maximum of M children.

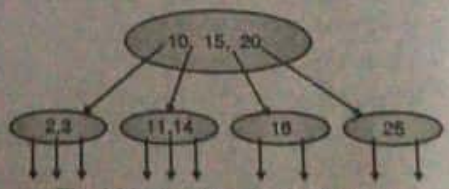


Fig. 4.24 : An example of 4-way tree

An M-way tree contains multiple keys in a node. This leads to reduction in overall height of the tree. If a node of M-way tree holds K number of keys then it will have K+1 children.

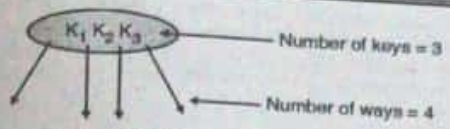
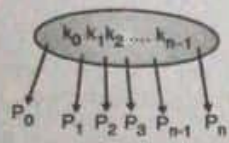


Fig. 4.25 : An M-way tree with 3 keys and 4 children

Definition :

A B-tree of order M is a M-way search tree with the following properties :

1. The root can have 1 to M–1 keys.
2. All nodes (except the root) have between [(M–1)/2] and M–1 keys.
3. All leaves are at the same depth.
4. If a node has t number of children then it must have (t – 1) number of keys.
5. Keys of a node are stored in ascending order.



6. $K_0, K_1, K_2 \ldots K_{n-1}$ are the keys stored in the node. Subtrees are pointed by $P_0, P_1 \ldots P_n$.

   then $K_0 \geq$ all keys of the subtree P_0

   $K_1 \geq$ all keys of the subtree P_1

   :

   :

   $K_{n-1} \geq$ all keys of the subtree P_{n-1}

   $K_{n-1} <$ all keys of the subtree P_n.

   An example of B-tree of order 4 is shown in Fig. 4.26.

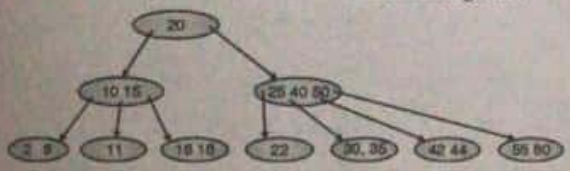
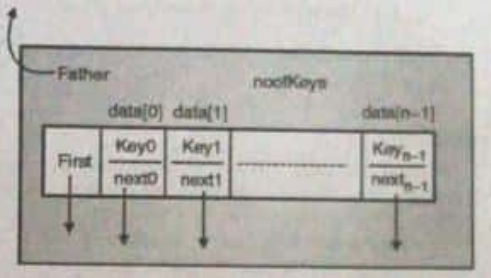


Fig. 4.26

Representation of a node of B-tree

```
# define MAX 5
class node;
struct pair
{
        node *next;
        int key;
};
class node
{
        public :
            node * first;
            node *father
            pair data [MAX];
            int noofkeys;
    } ;
```



Structure pair is being used to combine a key and the associated tree pointer. Class node can store a maximum of MAX pairs of (key, next). A node with MAX number of keys will give rise to MAX + 1 ways. The additional tree pointer is designated as 'first'. 'noofkeys' gives the actual number of keys stored in a node. The pointer 'father' points to the father of a node. 'father' pointer will be NULL for the root

Chapter 5 : Graphs

Q. 1 What is a graph ? Explain its types in brief. May 2015

Ans. :

Graph

A graph G is a set of vertices (V) and set of edges (E). The set V is a finite, nonempty set of vertices. The set E is a set of pair of vertices representing edges.

$$G = (V, E)$$

V(G) = Vertices of graph G

E(G) = Edges of graph G

An example of graph is shown in Fig. 5.1.

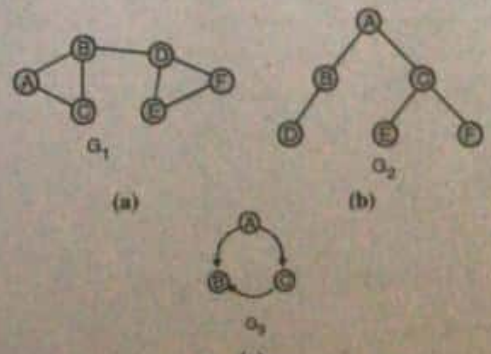


Fig. 5.1 : Graphs

The set representation for each of these graphs is given by

$V(G_1)$ = {A, B, C, D, E, F}
$V(G_2)$ = {A, B, C, D, E, F}
$V(G_3)$ = {A, B, C}
$E(G_1)$ = {(A, B),(A, C),(B, C),(B, D), (D, E),(D, F),(E, F)}
$E(G_2)$ = {(A,B),(A, C),(B, D),(C, E),(C, F)}
$E(G_3)$ = {(A, B),(A, C),(C, B)}

Undirected Graph

A graph containing unordered pair of vertices is called an undirected graph. In an undirected graph, pair of vertices (A, B) and (B, A) represent the same edge.

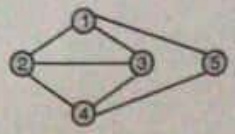


Fig. 5.2 : Example of an undirected graph

The set of vertices V = {1, 2, 3, 4, 5}.
The set of edges E = {(1, 2), (1, 3), (1, 5), (2, 3), (2, 4), (3, 4), (4, 5)}.

Directed Graph

A graph containing ordered pair of vertices is called a directed graph. If an edge is represented using a pair of vertices (V_1, V_2) then the edge is said to be directed from V_1 to V_2.

The first element of the pair, V_1 is called the start vertex and the second element of the pair, V_2 is called the end vertex. In a directed graph, the pairs (V_1, V_2) and (V_2, V_1) represent two different edges of a graph. Example of a directed graph is shown in Fig. 5.3.

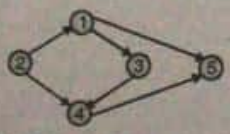


Fig. 5.3 : Example of a directed graph

The set of vertices V = {1, 2, 3, 4, 5, 5}.
The set of edges E = {(1, 3), (1, 5), (2, 1), (2, 4), (3, 4), (4, 5)}.

A Complete Graph

An undirected graph, in which every vertex has an edge to all other vertices is called a complete graph.

A complete graph with N vertices has $\frac{N(N-1)}{2}$ edges.

Example of a complete graph is shown in Fig. 5.4.

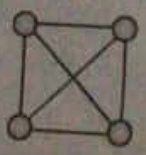


Fig. 5.4 : A complete graph

Weighted Graph

A weighted graph is a graph in which edges are assigned some value. Most of the physical situations are shown using weighted graph. An edge may represent a highway link between two cities. The weight will denote the distance between two connected cities using highway. Weight of an edge is also called its cost. The graph of Fig. 5.5 is an example of a weighted graph.

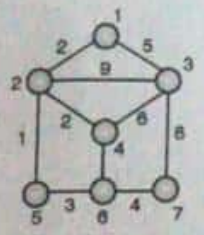


Fig. 5.5 : A weighted graph

Adjacent Nodes

Two vertices V_1 and V_2 are said to adjacent if there is an edge between V_1 and V_2.

Path

A path from vertex V_0 to V_n is a sequence of vertices V_0, V_1, $V_2 \cdots V_{n-1}$, V_n. Here, V_0 is adjacent to V_1, V_1 is adjacent to V_2 and V_{n-1} is adjacent to V_n. The **length** of a path is the number of edges on the path. A path with n vertices has a length of n – 1. A path is simple if all vertices on the path, except possibly the first and last, are distinct.

Cycle

A cycle is a simple path that begins and ends at the same vertex. Fig. 5.6 is an example of a graph with cycle.

A B D A is a cycle of length 3
B D C B is a cycle of length 3
A B C D A is a cycle of length 4

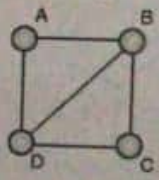


Fig. 5.6 : A graph with cycles

Connected Graph

A graph is said to be connected if there exists a path between every pair of vertices V_i and V_j.

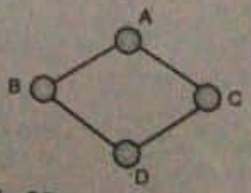


Fig. 5.7 : A connected graph

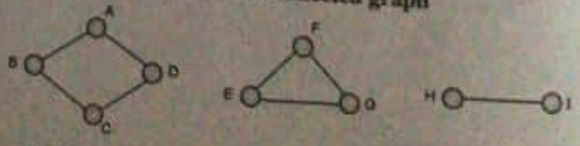
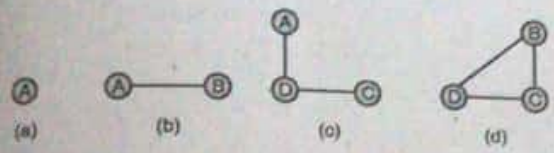


Fig. 5.8 : A disconnected graph with 3 components

The graph in Fig. 5.7 is a connected graph, but the graph in Fig. 5.8 is not connected as there is no path between D and E or between G and H. The graph of Fig. 5.8 consists of 3 connected components.

Subgraph

A subgraph of G is a graph G_1 such that V (G_1) is a subset of V (G) and E (G_1) is a subset of E (G).



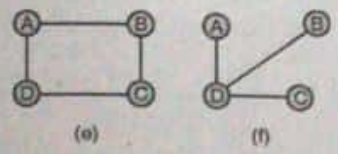


Fig. 5.9 : Some subgraphs of the graph of Fig. 5.6

Component

A component H of an undirected graph is a maximal connected subgraph. The graph of Fig. 5.10 has three components H_1, H_2 and H_3.

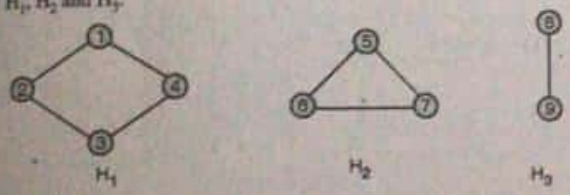


Fig. 5.10 : A graph with three components

Degree of a Vertex

The total number of edges linked to a vertex is called its degree. The **indegree** of a vertex is the total number of edges coming to that node. The **outdegree** of a node is the total number of edges going out from that node. A vertex, which has only outgoing edges and no incoming edges, is called a **source**. A vertex having only incoming edges and no outgoing edges is called a **sink**. When indegree of a vertex is one and outdegree is zero then such a vertex is called a **pendant** vertex. When the degree of a vertex is 0, it is an **isolated** vertex.

Self Edges or Self Loops

An edge of the form (V, V) is known as self edge or self loop. An example of self edge is shown in the Fig. 5.11.

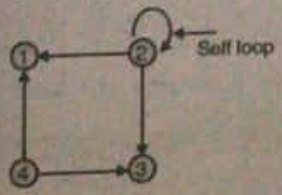


Fig. 5.11 : A graph with self loop

Multigraph :

A graph with multiple occurrences of the same edge is known as a multigraph. An example of multigraph is shown in the Fig. 5.12.

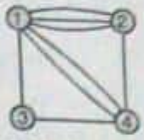


Fig. 5.12 : A multigraph

Tree

A tree is a connected graph without any cycle. The graphs of Fig. 5.13 are not trees as they contain cycles. The graphs of Fig. 5.14 are example of trees.

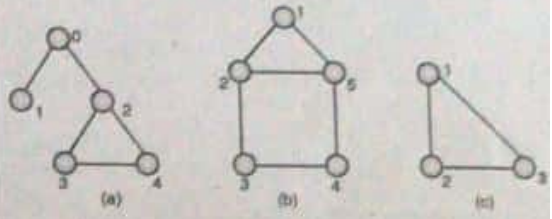


Fig. 5.13 : Graphs are not trees

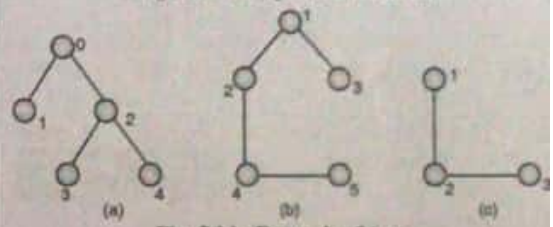


Fig. 5.14 : Example of trees

Spanning Trees

A spanning tree of a graph G = (V, E) is a subgraph of G having all vertices of G and no cycles in it. If the graph G is not connected then there is no spanning tree of G. A graph may have multiple spanning trees. Fig. 5.16 gives some of the spanning trees of the graph shown in Fig. 5.15.

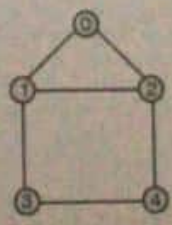


Fig. 5.15 : A sample connected graph

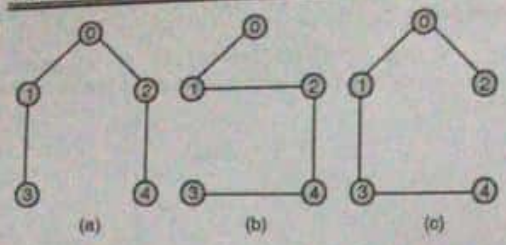


Fig. 5.16 : Spanning trees of the graph of Fig. 5.15

Minimal Spanning Tree

The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a group G = (V, E) is called a minimal cost spanning tree or simply the minimal spanning tree of G if its cost is minimum.

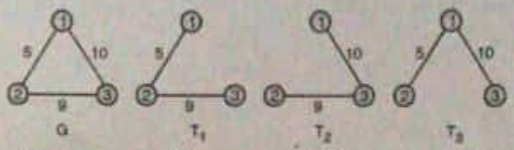


Fig. 5.17 : An example of minimal spanning tree

$G \rightarrow$ A sample weighted graph

$T_1 \rightarrow$ A spanning tree of G with cost 5 + 9 = 14

$T_2 \rightarrow$ A spanning tree of G with cost 10 + 9 = 19

$T_3 \rightarrow$ A spanning tree of G with cost 5 + 10 = 15

Therefore, T_1 with cost 14 is the minimal cost spanning tree of the graph G.

Q. 2 Explain Adjacency Matrix technique of graph representation. Dec. 2013, Dec. 2014, May 2015, Dec. 2016, May 2017

Ans. : Adjacency Matrix

A two dimensional matrix can be used to store a graph. A graph G = (V, E) where V = {0, 1, 2, ... n – 1} can be represented using a two dimensional integer array of size n × n.

int adj[20][20]; can be used to store a graph with 20 vertices.

adj[i][j] = 1, indicates presence of edge between two vertices i and j

= 0, indicates absence of edge between two vertices i and j

A graph is represented using a square matrix. Adjacency matrix of an undirected graph is always a symmetric matrix, i.e. an edge (i, j) implies the edge (j, i).

Adjacency matrix of a directed graph is never symmetric adj[i][j] = 1, indicates a directed edge from vertex i to vertex j.

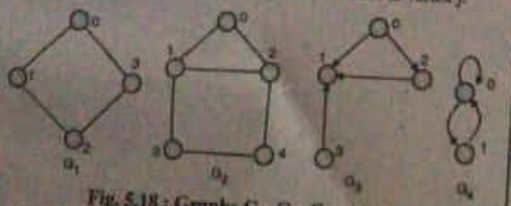
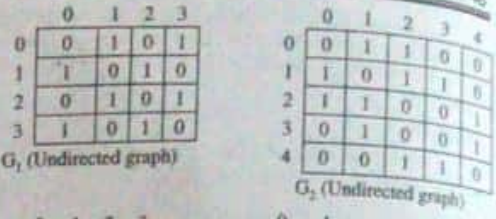


Fig. 5.18 : Graphs G_1, G_2, G_3 and G_4



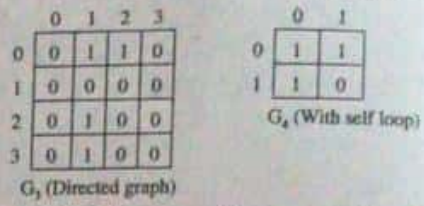


Fig. 5.19 : Adjacency matrix representation of graphs G_1, G_2, G_3 and G_4 of Fig. 5.18

Adjacency matrix representation of a weighted graph

For weighted graph, the matrix adj[ ][ ] is represented as :

If there is an edge between vertices i and j then Adj[i][j] = weight of the edge (i, j) otherwise,

Adj[i][j] = 0

Fig. 5.20 is an example of representation of a weighted graph.

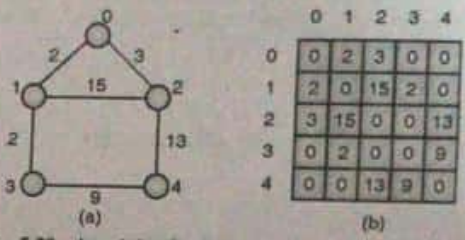


Fig. 5.20 : A weighted graph and its adjacency matrix

Adjacency matrix representation of graphs is very simple to implement.

Memory requirement : Adjacency matrix representation of a graph wastes lot of memory space. Such matrices are found to be very sparse. Above representation requires space for n^2 elements for a graph with n vertices. If the graph has e number of edges then $n^2 - e$ elements in the matrix will be 0.

Presence of an edge between two vertices V_i and V_j can be checked in constant time.

if(adj[i][j] == 1)

edge is present between vertices i and j

else

edge is absent between vertices i and j.

Degree of a vertex can easily be calculated by counting all non-zero entries in the corresponding row of the adjacency matrix.

Q. 3 Explain Adjacency List technique of graph representation. Dec. 2013, Dec. 2014, Dec. 2016, May 2017

Ans. : Adjacency List

A graph can be represented using a linked list. For each vertex, a list of adjacent vertices is maintained using a linked list. It creates a separate linked list for each vertex V_i in the graph G = (V, E).

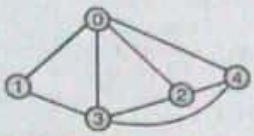


Fig. 5.21 : A graph

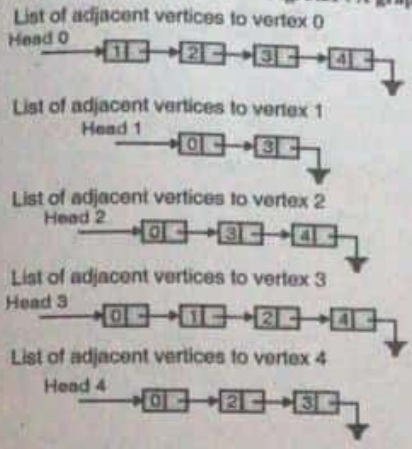


Fig. 5.22 : Adjacency list for each vertex of graph of Fig. 5.21

Adjacency list of a graph with n nodes can be represented by an array of pointers. Each pointer points to a linked list of the corresponding vertex. Fig. 5.23 shows the adjacency list representation of graph of Fig. 5.21.

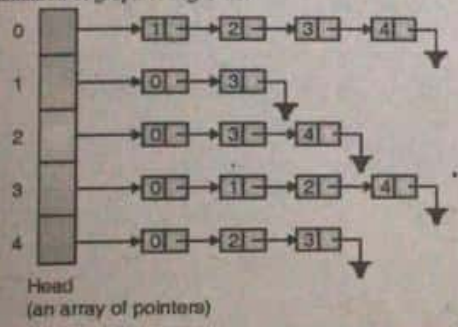


Fig. 5.23 : Adjacency list representation of the graph of Fig. 5.21

Adjacency list representation of a graph is very memory efficient when the graph has a large number of vertices but very few edges. For an undirected graph with n vertices and e edges, total number of nodes will be n + 2e. If e is large then due to overhead of maintaining pointers, adjacency list representation does not remain cost effective over adjacency matrix representation of a graph.

Degree of a node in an undirected graph is given by the length of the corresponding linked list. Finding in-degree of a directed graph represented using adjacency list will require O(e) comparisons. Lists pointed by all vertices must be examined to find the indegree of a node in a directed graph. Checking the existence of an edge between two vertices i and j is also time consuming. Linked list of vertex i must be searched for the vertex j.

A graph can be represented using a structure as defined below :

```
# define MAX 30        /* graph has maximum of 30 nodes */
typedef struct node
{
    struct node * next;
    int vertex;
}node;
node * head[MAX];
```

If a weighted graph is to be represented using a adjacency list, then structure "node" should be modified to include the weight of a edge.

Thus the definition of 'struct node' is modified as below :

```
typedef struct node
{
    struct node * next;
    int vertex;
    int weight;
}node;
```

Q. 4 Explain Path Matrix.

Dec. 2013, Dec. 2014, Dec. 2016, May 2017

Ans. : Path Matrix

A path matrix for a graph G = (V, E) with n vertices is defined given below.

A path matrix is an n × n matrix whose elements are given by

$$a_{ij} = \begin{cases} 1, & \text{if there is path from } V_i \text{ to } V_j \\ 0, & \text{otherwise} \end{cases}$$

Example :

A path matrix for graph given in Fig. 5.24 is given in Fig. 5.25.

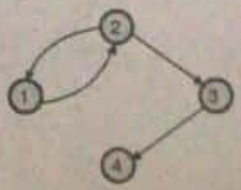


Fig. 5.24 : A sample graph

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

Fig. 5.25 : Path matrix for graph appearing in Fig. 5.24

Path matrix can be found by calculating $A^2, A^3, \ldots, A_n$ and then adding them together. The matrix A is the adjacency matrix.

If $B = A + A^2 + A^3 + \ldots + A^n$

then the path matrix P can be obtained from B by replacing each non-zero element by 1.

Q. 5 Write a function for DFS traversal of graph. Explain its working with an example.

Dec. 2013, Dec. 2014, Dec. 2015

Ans. :

Depth First Search (DFS) :

It is like preorder traversal of tree. Traversal can start from any vertex, say V_i. V_i is visited and then all vertices adjacent to V_i are traversed recursively using DFS.

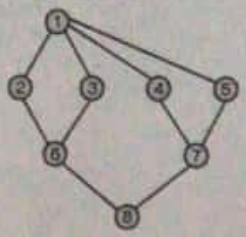


Fig. 5.26 : Graph G

DFS (G, 1) is given by

(a) Visit (1)
(b) DFS (G, 2)
DFS (G, 3)
DFS (G, 4)
DFS (G, 5)
} all nodes adjacent to 1

Since, a graph can have cycles. We must avoid re-visiting a node. To do this, when we visit a vertex V, we mark it visited. A node that has already been marked as visited, should not be selected for traversal. Marking of visited vertices can be done with the help of a global array visited[]. Array visited[] is initialized to false (0).

Algorithm for DFS (Recursive)

```
        n ← number of nodes
(i)     Initialize visited[] to false (0)
        for (i = 0; i < n; i++)
                visited[i] = 0;
(ii)    void DFS (vertex i) [DFS starting from i]
        {
                visited[i] = 1;
                for each w adjacent to i
                        if(! visited[w])
          DFS(w);//Recursive call to DFS from an adjacent node.
        }
```

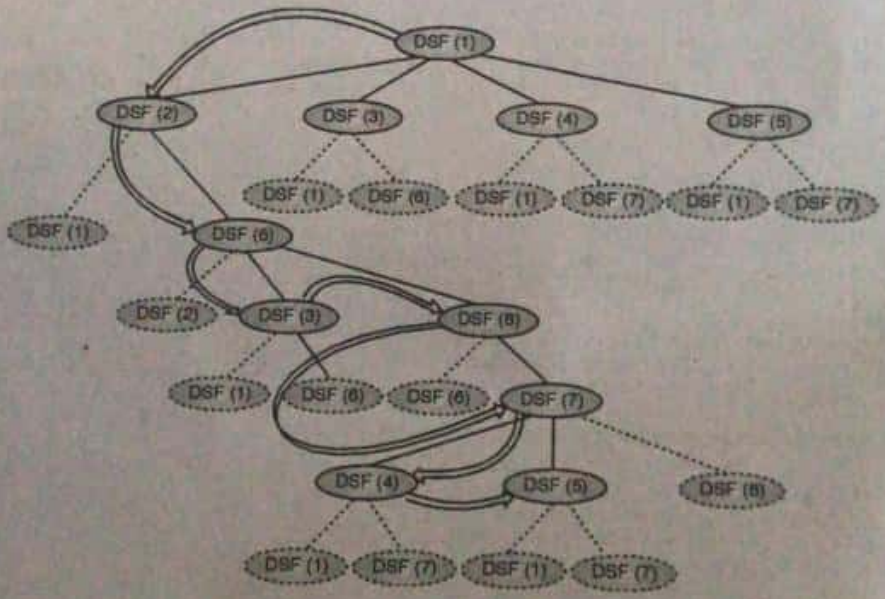


Fig. 5.27 : DFS traversal on graph of Fig. 5.26

DFS traversal on graph of Fig. 5.26.

DSF(i) Node i can be used for recursive traversal using DFS().

DFS(i) Node i is already visited

Traversal start from vertex 1. Vertices 2, 3, 4 and 5 are adjacent to vertex 1. Vertex 1 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	0	0	0	0	0	0	0

Out of the adjacent vertices 2, 3, 4 and 5, vertex number 2 is selected for further traversal. Vertices 1 and 6 are adjacent to vertex 2. Vertex 2 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	0	0	0	0	0	0

Out of the adjacent vertices 1 and 6, vertex 1 has already been visited. Vertex number 6 is selected for further traversal. Vertices 2, 3 and 8 are adjacent to vertex 6. Vertex 6 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	0	0	0	1	0	0

Out of the adjacent vertices 2, 3 and 8, vertex 2 is already visited. Vertex number 3 is used for further expansion. Vertices 1 and 6 are adjacent to vertex 3. Vertex 3 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	0	0	1	0	0

Vertices 1 and 6 are already visited, therefore it goes back to vertex 6. (Vertex 6 is predecessor of vertex 3 in DFS() sequence). Out of the adjacent vertices 2, 3 and 8, 2 and 3 are visited. It selects vertex 8 for further expansion. Vertices 6 an 7 are adjacent to vertex 8. Vertex 8 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	0	0	1	0	1

Out of the adjacent vertices 6 and 7, vertex 6 is already visited. Vertex number 7 is used for further expansion. Vertices 4, 5 and 8 are adjacent to vertex number 7. Vertex 7 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	0	0	1	1	1

Out of the adjacent vertices 4, 5 and 8, vertex 4 is selected for further expansion. Vertices 1 and 7 are adjacent to vertex 4. Vertex 4 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	1	0	1	1	1

Adjacent vertices 1 and 7 are already visited. It goes back to vertex 7 and selects the next unvisited node 5 for further expansion. Vertex 5 is marked as visited.

	1	2	3	4	5	6	7	8
Visited →	1	1	1	1	1	1	1	1

DFS, traversal sequence → 1, 2, 6, 3, 8, 7, 4, 5

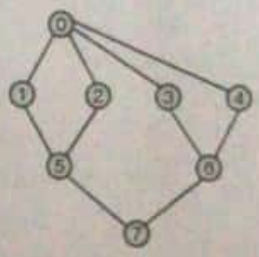
Q. 6 Write a program to implement DFS traversal on a graph represented using an adjacency matrix.

Ans. :

```
#include<conio.h>
#include<stdio.h>
void DFS(int);
int G[10][10],visited[10],n;
// n->no of vertices
// graph is stored in array G[10][10]
void main()
{
    int i,j;
    printf("\nEnter no of vertices: ");
    scanf("%d",&n);
    // read the adjecency matrix
    printf("\nEnter adjecency matrix of the graph :");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d",&G[i][j]);

    // visited is initialise to zero
    for(i=0;i<n;i++)
        visited[i]=0;
    DFS(0);
}

void DFS(int i)
{
    int j;
    printf("\n%d",i);
    visited[i]=1;
    for(j=0;j<n;j++)
        if(!visited[j] && G[i][j]==1)
            DFS(j);
}
```

Output

```
Enter no of vertices : 8
Enter adjecency matrix of the graph : 0 1 1 1 1 0 0 0
1 0 0 0 0 1 0 0
1 0 0 0 0 1 0 0
1 0 0 0 0 0 1 0
1 0 0 0 0 0 1 0
0 1 1 0 0 0 0 1
0 0 0 1 1 0 0 1
0 0 0 0 0 1 1 0
0
1
5
2
7
6
3
4
```



Graph used for input

Q. 7 Write a function for BFS traversal of graph.

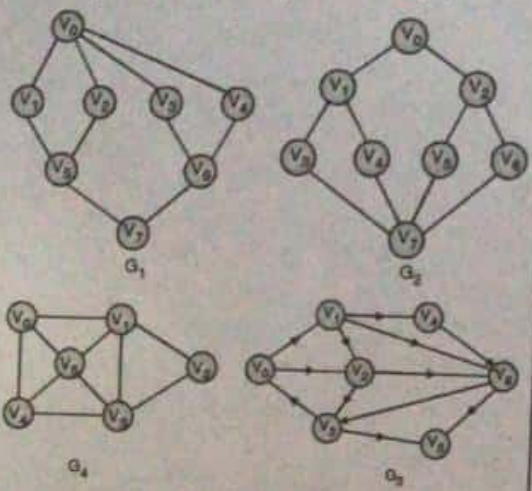
Dec. 2013, Dec. 2014, Dec. 2015

Ans. : Breadth First Search (BFS)

It is another popular approach used for visiting the vertices of a graph. This method starts from a given vertex V_0. V_0 is marked as visited. All vertices adjacent to V_0 are visited next. Let the vertices adjacent to V_0 are V_{01}, V_{11}, V_{12} ... V_{1t}. V_{11}, V_{12} ... V_{1t} and they are marked as visited. All unvisited vertices adjacent to V_{11}, V_{12} .....V_{1t} are visited next. The method continues until all vertices are visited. The algorithm for BFS has to maintain a list of

vertices which have been visited but not explored for adjacent vertices. The vertices which have been visited but not explored for adjacent vertices can be stored in queue. Initially the queue contains the starting vertex. In every iteration, a vertex is removed from the queue and its adjacent vertices which are not visited as yet are added to the queue. The algorithm terminates when the queue becomes empty.

Fig. 5.28 gives the BFS sequence on various graphs.



BFS sequence :

$G_1 \rightarrow V_0 | V_1 V_2 V_3 V_4 | V_5 V_6 | V_7$

$G_2 \rightarrow V_0 | V_1 V_2 | V_3 V_4 V_5 V_6 | V_7$

$G_3 \rightarrow V_0 | V_1 V_2 | V_4 V_6 V_5 | V_5 V_4$

$G_4 \rightarrow V_0 | V_1 V_4 V_2 | V_2 V_3$

Fig. 5.28 : BFS traversal on G_1, G_2, G_3 and G_4

Q. 8 Explain BFS algorithm with examples. May 2014

Ans. : Algorithm for BFS :

```
/* Array visited[] is initialize to 0 */
/* BFS traversal on the graph G is carried out beginning at
vertex V */
void BFS(int V)
{
        q : a queue type variable;
        initialize q;
        visited[v] = 1;      /* mark v as visited */
        add the vertex V to queue q;
        while(q is not empty)
        {
            v ← delete an element from the queue;
            for all vertices w adjacent from V
            {
                if(!visited[w])
                {
                    visited[w] = 1;
                    add the vertex w to queue q;
                }
            }
        }
}
```

Q. 9 Show the working of BFS algorithm on the following graph. May 2014, May 2016

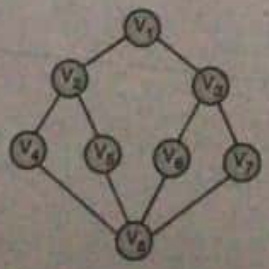


Fig. 5.29 : A sample graph

Ans. :

Queue	Visited[ ]	Vertex visited	Action
	1 2 3 4 5 6 7 8		
NULL	0 0 0 0 0 0 0 0		

Queue	Visited[ ]	Vertex visited	Action
V_1	1 2 3 4 5 6 7 8 1 0 0 0 0 0 0 0	V_1	add (q, V_1) visit (V_1)
$V_2 V_3$	1 2 3 4 5 6 7 8 1 1 1 0 0 0 0 0	$V_1 V_2 V_3$	delete (q), add and visit adjacent vertices
$V_3 V_4 V_5$	1 2 3 4 5 6 7 8 1 1 1 1 1 0 0 0	$V_1 V_2 V_3 V_4 V_5$	delete (q), add and visit adjacent vertices
$V_4 V_5 V_6 V_7$	1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 0	$V_1 V_2 V_3 V_4 V_5 V_6 V_7$	delete (q), add and visit adjacent vertices
$V_5 V_6 V_7 V_8$	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q), add and visit adjacent vertices
$V_6 V_7 V_8$	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q)
$V_7 V_8$	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q), add and visit adjacent vertices
V_8	1 2 3 4 5 6 7 8 1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	delete (q)
NULL	1 1 1 1 1 1 1 1	$V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8$	algorithm terminates as the queue is empty

10 Write a program in C to implement the BFS traversal of a graph. May 2016, Dec. 2016

s. :

ogram to implement BFS traversal on a graph resented using adjacency matrix.

```
clude<conio.h>
clude<stdio.h>
fine MAX 10
def struct Q

 int R,F;
 int data[MAX];

mpty(Q *P);
ull(Q *P);
 enqueue(Q *P,int x);
dequeue(Q *P);
 BFS(int);
```

```
int G[MAX][MAX];
int n;
void main()
{
    int i,j,v;
    printf("\nEnter no of vertices : ");
    scanf("%d",&n);
    printf("\nEnter the adjecendy matrix of graph : ");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d",&G[i][j]);
    printf("\nEnter the starting vertex for BFS");
    scanf("%d",&v);
    BFS(v);
    getch();
}
void BFS(int v)
{
```

```
    int visited[MAX],i;
    Q q;
    q.R=q.F=-1;      // initialize the queue
    for(i=0;i<n;i++)
        visited[i]=0;
    enqueue(&q,v);
    printf("\n visit\t%d",v);
    visited[v]=1;
    while(!empty(&q))
    {
        v=dequeue(&q);
        // visit and add adjacent vertices
        for(i=0;i<n;i++)
            if(visited[i]==0 && G[v][i]!=0)
            {
                enqueue(&q,i);
                visited[i]=1;
                printf("\nvisit\t%d",i);
            }
    }
}
int empty(Q *P)
{
    if(P->R==-1)
        return(1);
    return(0);
}

int full(Q *P)
{
    if(P->R==MAX-1)
        return(1);
    return(0);
}
void enqueue(Q *P,int x)
{
    if(P->R==-1)
    {
        P->R=P->F=0;
        P->data[P->R]=x;
    }
    else
    {
        P->R=P->R+1;
        P->data[P->R]=x;
    }
}
int dequeue(Q *P)
{
    int x;
    x=P->data[P->F];
    if(P->R==P->F)
    {
        P->R=-1;
        P->F=-1;
    }
    else
        P->F=P->F+1;
    return(x);
}
```

Output

```
Enter no of vertices : 8
Enter the adjecendy matrix of  graph :
0 1 1 1 1 0 0 0
1 0 0 0 0 1 0 0
1 0 0 0 0 1 0 0
1 0 0 0 0 0 1 0
1 0 0 0 0 0 1 0
0 1 1 0 0 0 0 1
0 0 0 1 1 0 0 1
0 0 0 0 0 1 1 0
Enter the starting vertex for BFS 0
 visit  0
visit  1
visit  2
visit  3
visit  4
visit  5
visit  6
visit  7
```

Chapter 6 : Sorting and Searching

Q. 1 Write short note on sequential search.

Ans. : Sequential Search

In sequential search elements are examined sequentially starting from the first element. The process of searching terminates when the list is exhausted or a comparison results in success.

Algorithm for searching an element 'key' in an array 'a[ ]' having n elements :

The search algorithm starts comparison between the first element of a[ ] and "key".

As long as a comparison does not result in success, the algorithm proceeds to compare the next element of "a[ ] with "key". The process terminates when the list is exhausted or the element is found.

'C' function for sequential search.

```
int sequential (int a[ ] , int key, int n)
{
   int i = 0 ;
   while (i < n)
   {
      if (a[i] == key)
          return (i) ;
      i++ ;
   }
   return (– 1) ;
}
```

The function returns the index of the element in 'a[ ]' for successful search. A value –1 is returned if the element is not found.

Analysis of sequential search algorithm

Number of comparisons required for a successful search is not fixed. It depends on the location (place) being occupied by the key element. An element at i^{th} location can be searched after i-comparisons. Number of comparisons required is probabilistic in nature. We could best calculate the average number of comparisons required for searching an element.

Let P_i is the probability that the element to be searched will be found at i^{th} position. i number of comparisons will be required to search the i^{th} element.

∴ Expected number of comparisons for a successful search

$C = 1.P_1 + 2.P_2 + \ldots + nPn$

Since the element could be found at any location with equal probability.

$P_1 = P_2 \ldots = P_n = 1/n$

$$\therefore \quad C = \frac{1}{n} + \frac{2}{n} + \ldots + \frac{n}{n} = \frac{1}{n}(1 + 2 + \ldots + n) = \frac{n(n+1)}{2n} = \frac{n+1}{2}$$

Time complexity of sequential search = O(n).

Q. 2 Write short note on binary search.

Ans. :

Binary Search

Linear search has a time complexity O(n). such algorithms are not suitable for searching when number of elements is large. Binary search exhibits much better timing behaviour in case of large volume of data with timing complexity

$O(\log_2 n)$

Number of comparisons for $n = 2^{20}$ (1 million)

Sequential search (in worst case) = 2^{20} comparisons.

Binary search (in worst case) = $\log_2 2^{20} = 20$ comparisons.

Linear search (sequential search) may need 1 million comparisons for searching an element in an array having 1 million elements. Binary search will require, just 20 comparisons for the same task. Binary search uses a much better method of searching. Binary search is applicable only when the given array is sorted.

This method makes a comparison between the "key" (element to be searched) and the middle element of the array.

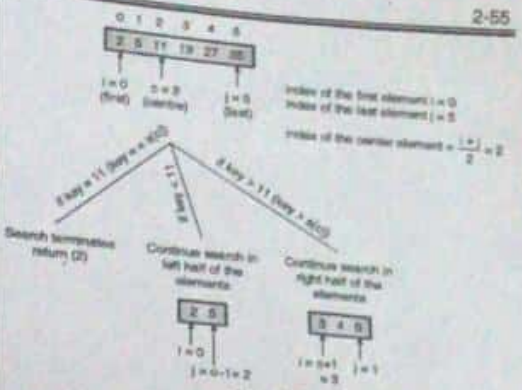


Fig. 6.1

Since elements are sorted, comparisons may result in either a match or comparison could be continued with either left half of elements or right half of the elements.

Left half of elements could be selected by simply making j = c–1

Right half of element could be selected by simply making i = c+1

Process of selecting either the left half or the right half continues until the element is found or element is not there.

Q. 3 Apply binary search on the following numbers stored in array from A [0] to A [10]

9, 17, 23, 38, 45, 50, 57, 76, 79, 90, 100 to search numbers – 10 to 100.

Ans. : Searching 10 :

0	1	2	3	4	5	6	7	8	9	10						
9	17	23	38	45	50	57	76	79	90	100				i	j	k
↑					↑					↑				0	10	5
i					k					j						

Step 1 : Since 10 < A [5], j = K – 1 = 4

0	1	2	3	4										i	j	k
9	17	23	38	45										0	4	2
↑		↑		↑												
i		k		j												

Step 2 : Since 10 < A [2], j = K – 1 = 1

0	1											
9	17									i	j	k
↑	↑									0	1	0
↑												
i	j											
k												

Step 3 : Since 10 > A [0], i = K + 1 = 1

										i	j	k
1										1	1	1
17												
↑	↑	↑										
i	j	k										

Step 4 : Since 10 < A [1], j = K – 1 = 0

As i becomes less than j, element 10 is not in the array A [ ]

Searching 100

0	1	2	3	4	5	6	7	8	9	10				
9	17	23	38	45	50	57	76	79	90	100		i	j	K
↑					↑					↑		0	10	5
i					k					j				

Step 1 : Since 100 > A [5], i = K + 1 = 6

6	7	8	9	10										
57	76	79	90	100								i	j	K
↑		↑		↑								6	10	8
i		k		j										

Step 2 : Since 100 > A [8], i = K + 1 = 9

9	10													
90	100											i	j	K
↑ ↑	↑											9	10	9
i k	j													

Step 3 : Since 100 > A [9], i = K + 1 = 10

10														
100												i	j	K
↑ ↑ ↑												10	10	10
i j k														

Since, the element to be searched is found at A [10], search terminates with a success.

Q. 4 List and explain types of sorting.

Ans. : Sorting

Sorting is a process of ordering a list of elements in either ascending or descending order. Sorting can be divided into two categories.

(a) **Internal sorting** takes place in the main memory of the computer. Internal sorting can take advantage of the random access nature of the main memory.

Elements to be sorted are stored in an integer array.

(b) **External sorting** is carried on secondary storage. External sorting becomes a necessity if the number of elements to be sorted is too large to fit in main memory.

External sorting algorithms should always take into account that movement of data between secondary storage and main memory is best done by moving a block of contiguous elements.

Simple sorting algorithms like bubble sort, insertion sort take $O(n^2)$ time to sort n elements. More complicated algorithms like quick sort, merge sort, heap sort take O(n log n) time to sort n elements. Sorting algorithms like bubble sort, insertion sort, merge sort are stable. Whereas quick sort and heap sort are unstable. A sorting algorithm is said to be **stable** if after sorting, identical elements appear in the same sequence as in the original unsorted list.

Q. 5 Write short note on Insertion Sort.

Ans. : Insertion Sort

An element can always be placed at a right place in sorted list of element for example

List of elements(sorted) 5 9 10 15 20

Element to be placed = 6

If the element 6 is to be inserted in a sorted list of elements(5, 9, 10, 15, 20), its rightful place will be between 5 and 9. Elements with value > 6 should be moved right by one place. Thus creating a space for the incoming element.

5 | 9 10 15 20 | → Moved right by 1 place

5 6 9 10 15 20 → Element 6 is inserted between 5 and 9

Insertion sort is based on the principle of inserting the element at its correct place in a previously sorted list. It can be varied from 1 to n – 1 to sort the entire array.

Index - 0 1 2 3 4 5 6 Initial unsorted list

5	0 1 9 2 6 4

A list of sorted element (a list of single element is always sorted)

a list of unsorted element

1st iteration (place element at location '1' i.e. a[1], at its correct place)

0	1	2	3	4	5	6
0	5	1	9	2	6	4

Sorted unsorted

2nd iteration (place a[2] at its correct place)

0	1	5	9	2	6	4

Sorted unsorted

3rd iteration (place a[3] at its correct place)

0	1	5	9	2	6	4

Sorted unsorted

4th iteration (Place a[4] at its correct place)

0	1	2	5	9	6	4

sorted Unsorted

5th iteration (Place a[5] at its correct place)

0	1	2	5	6	9	4

Sorted unsorted

6th iteration (Place a[6] at its correct place)

0 1 2 4 5 6 9

Fig. 6.2 : Sorting of elements using insertion sort

Insertion sort requires n-1 passes to sort an array having n elements. Before the i^{th} pass starts, elements at position 0 through i-1 are already sorted.

Index →	0	1	2	3	4	5	6
a List →	1	5	9	11	6	2	5	1
					↑		

at the beginning of pass = 4

Element at position i = 4 can be inserted at its correct place after the following operations

temp = a[4]

a[4] = a[3]

a[3] = a[2]

a[2] = temp

All elements larger than 6 are moved right by 1 place. i^{th} element is saved in the variable temp to protect its value before it is overwritten due to data movement.

i^{th} element can also be inserted at its correct place with the help of the following program segment.

```
temp = a[i] ;
for( j = i – 1 ; j> = 0 && temp < a[ j] ; j – –)
    a[ j + 1] = a[j]; /*move input */
a[ j + 1] = temp;
```

'C' function for insertion sort :

```
void insertion_sort(int a[ ], int n)
{
    int i, j, temp;
    for(i=1;i<n;i++)          /* passes 1 through n – 1 */
    {
        temp = a[i];
        for( j = i – 1; j > = 0 && a[ j] > temp; j – –)
            a[j + 1] = a[ j];
        a[ j + 1] = temp;
    }
}
```

Q. 6 Write a program in C to implement insertion sort. May 2016

Ans. :

```
#include<conio.h>
#include<stdio.h>
void insertion_sort(int[],int);
void main()
{
    int a[50],n,i;
    printf("\nEnter no of elements :");
    scanf("%d",&n);
    printf("\nEnter array elements :");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    insertion_sort(a,n);
  printf("\nSorted array is :");
    for(i=0;i<n;i++)
        printf("%d",a[i]);
    getch();
}
void insertion_sort(int a[],int n)
{
    int i,j,temp;
    for(i=1;i<n;i++)
    {
        temp=a[i];
        for( j=i-1;j>=0 && a[ j]>temp;j--)
            a[ j+1]=a[ j];
        a[j+1]=temp;
    }
}
```

Output :

```
Enter no of elements  :  5
Enter array elements  :  57  89  64  56  77  333
Sorted array is       :  56  57  64  77  89  333
```


Q. 7 With help of example, explain bubble sort.

Ans. : Bubble Sort :

Bubble sort is one of the simplest and the most popular sorting method. The basic idea behind bubble sort is as a bubble rises up in water, the smallest element goes to the beginning.

This method is based on successive selecting the smallest element through exchange of adjacent element.

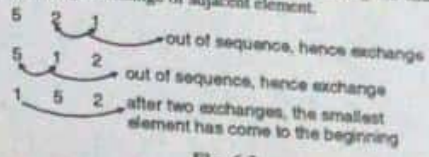


Fig. 6.3

First pass of the bubble sort

Let n be the number of elements in an array a[ ]. The first pass begins with the comparison of a[n – 1] and a[n – 2]. If a[n – 2] is larger than a[n – 1], the two elements are exchanged.

The smaller elements, now at a[n – 2] is compared with a[n – 3] and if necessary the elements are exchanged to place the smaller one in a[n – 3]. Comparison progresses backward and after the last comparison of A[1] and A[0] and possible exchange the smallest element will be placed at a[0].

As a variation, the first pass could begin comparison with a[0] and a[1]. If a[0] is larger than a[1], the two elements are exchanged.

Larger one will be placed in a[1] after the first comparison. Comparison can work forward and after the last comparison of a[n – 2] and a[n – 1], the larger element will be placed in a[n –1].

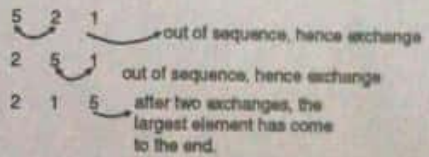


Fig. 6.4

First pass of the bubble sort with comparison in the forward direction.

The **second pass** is an exact replica of the first pass except that this time, the pass ends with the comparison and possible exchange of a[n – 3] and a[n – 2]. After the end of second pass, the second largest element will be placed at a[n – 2].

Bubble sort requires a total of n – 1 passes. If n – 1 elements are arranged (one element each pass) in ascending order from a[1] to a[n – 1], smallest element will finally left at a[0].

Loop for sorting an array a[ ] having n elements :

```
for(i = 1; i < n; i ++)
    for(j = 0; j < n – i; j ++)
        if(a[j] > a[j + 1])
            exchange a[j] and a[j + 1]
```

Outer loop is for passes and the inner loop is for comparisons.
In pass 1 (i = 1), there will be n – 1 comparisons (j = 0 to n – 2)
In pass 2 (i = 2), there will be n – 2 comparisons (j = 0 to n – 3)
:
:
In pass n – 1 (i = n – 1), there will be 1 comparison (j = 0 to 0)

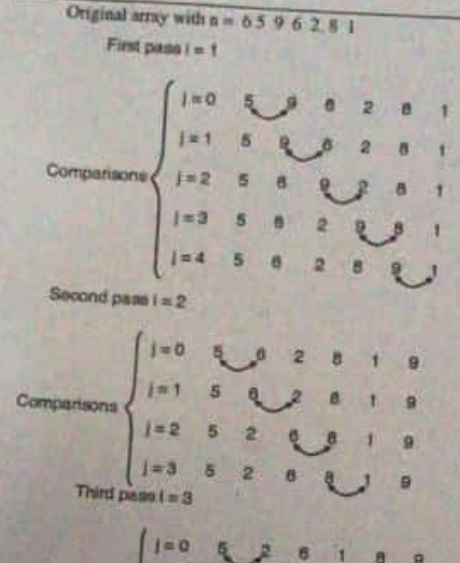


Illustration of bubble sort.

(Contd.) Fig. 6.5

Q. 8 Write C function for selection sort.

Ans. :

'C' function for selection sort.

```
void selectionsort(int a[ ], int n)
{
    int i, j, k, temp;
    /*  i – outer loop
        j → inner loop
        k →index of the smallest element
        temp → for swapping */
    for(i = 0; i < n – 1; i ++)
        {
            k = i ;
        /* i^th element is assumed to be the smallest */
            for(j = i + 1; j < n; j ++)
                if(a[j] < a[k])
                    k = j;
            if(k ! = i)
            {
                        temp = a[i];
                        a[i] = a[k];
                        a[k] = temp;
            }
        }
}
```

Analysis of selection sort :

Selection sort is not data sensitive. In i^{th} pass, n – i comparisons will be needed to select the smallest element.

Thus, the number of comparisons needed to sort an array having n elements.

$$= (n-1) + (n-2) + \ldots + 2 + 1$$

$$= \frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n) = O(n^2)$$

Q. 9 Compare the three sorting algorithms
(a) Bubble sort (b) Selection sort
(c) Insertion sort

Ans. : A sorting algorithm can be judged on the basis of following parameters :

1. **Simplicity of algorithm :** All the three sorting algorithms are equally simple to write.
2. **Timing complexity :** Bubble sort and selection sort are not data sensitive. Both of them have a timing requirement of $O(n^2)$. Insertion sort is data sensitive. It works much faster if the data is partially sorted.
   Best case behaviour (when input data is sorted) = O(n).
   Worst case behaviour (when input data is in descending order) = $O(n^2)$.
3. **Sort stability :** All the three sorting algorithms are stable.
4. **Storage requirement :** No additional storage is required. All of them are in-place sorting algorithms.
5. All of them can be used for internal as well as external sorting.

Q. 10 Write short note on Quick Sort. Also write 'C' function for sorting an array of element using quick sort.

Ans. :

Quick Sort :

Quick sort is the fastest internal sorting algorithm with the time complexity = O(n log n). The basic algorithm to sort an array a[ ] of n elements can be described recursively as follows :

1. If n < = 1, then return

2. Pick any element V in a[ ]. This is called the pivot. Rearrange elements of the array by moving all elements $x_i > V$ right of V and all elements $x_i \leq V$ left of V. If the place of the V after re-arrangement is j, all elements with value less than V, appear in a[0], a[1] ... a[j – 1] and all those with value greater than V appear in a[j + 1] ... a[n – 1].

3. Apply quick sort recursively to a[0] ..... a[j – 1] and to a[j + 1] .... a[n – 1]

Entire array will thus be sorted by as selecting an element V.

(a) Partitioning the array around V.

(b) Recursively, sorting the left partition.

(c) Recursively sorting the right partition.

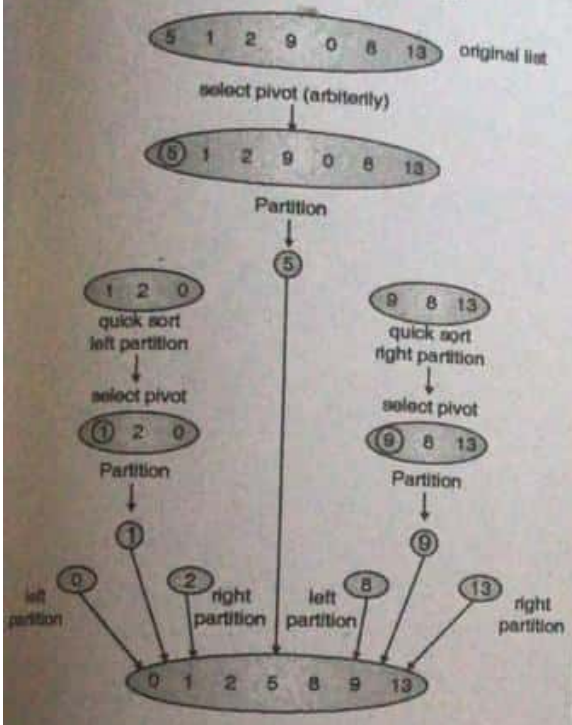


Fig. 6.6 : Illustration of quick sort

Picking a Pivot

Choose the popular choice of the first element as the pivot. There are several strategies for pivoting.

Choose the first element as the pivot

Partitioning :

Let the elements of the array a[l], a[l + 1] .... a[n – 1], a[n] are to be partitioned around the pivot V = a[l]. Initial value of l will be l and that of n will be n – 1.

As recursion proceeds, these values will change. Index of the pivot (say K) will satisfy the following conditions

$K \geq l$

$K \leq n$

Two cursors (index variable) are initialized with

$i = l + 1$ and

$j = n - 1$

i moves towards right searching for an element greater than V. j moves towards left searching for an element smaller than V. When these elements are found they are interchanged.

Again i moves towards right and j towards left and exchange is made whenever necessary. The process ends when i is equal to j or i > j.

j gives the index of the pivot element after termination of the above process.

e.g. Let the array of number be

0	1	2	3	4	5	6	7	8	9	10	11	12
30	35	10	15	20	34	5	18	6	11	13	26	38

Initially $l = 0$, $j = 12$, $V = a[l]$ (i.e. V = 30)

i is set to $l + 1$ and j is set to n – 1 as shown below

30	35	10	15	20	34	5	18	6	11	13	26	38

↑ i = 1 ↑ j = 12

Now i moves right, while a[i] < 30. Hence i does not move further to right as a[1] > 30.

Then, j moves left, while a[j] > 30.

30	35	10	15	20	34	5	18	6	11	13	26	38

i=1 j=11 interchange

At this point a[i] and a[j] are interchanged and the movement of i and j resumes.

0	1	2	3	4	5							
30	26	10	15	20	34	5	18	6	11	13	35	38

i=5 j=10

i moves right to a[5] as 34 > 30. j moves left to a[10] as 13 < 30. Once again a[i] and a[j] are swapped.

0	1	2	3	4	5	6	7	8	9	10	11	12
30	26	10	15	20	13	5	18	6	11	34	35	38

j = 9 i = 10

Now, i moves to a[10] as 34 > 30 and j moves to a[9] as 11 < 30.

Since, i > j, the process ends. j gives the location of the pivot element. Pivot element a[l with $l = 0$] is interchanged with a[i] to partition the array.

11	26	10	15	20	13	5	18	6	30	34	35	38

left partition Pivot right partition

All element a[i] with $1 \leq i < j$ are less than 30 and all elements a[j] with $u \geq i > j$ are greater than 30.

'C' function for the above partitioning algorithm :

```
int partition(int a[],int l,int u)
{
  int v,i,j,temp;
  v=a[l];
```

```
    i=l;
    j=u+1;
    do
    {
        do
            i++;
        while(a[i]<v && i<=u);
            do
                j--;
            while(v<a[j]);
                        if(i<j)
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
    }while(i<j);
    a[l]=a[j];
    a[j]=v;
    return(j);
}
```

'C' function for sorting an array of element using quick sort :

```
void quick_sort(int a[],int l,int u)
{
    int j;
    if(l<u)
    {
        j=partition(a,l,u);
        quick_sort(a,l,j-1);
        quick_sort(a,j+1,u);
    }
}
```

Q. 11 Write a program in 'C' to perform quick sort. May 2014, May 2015, Dec. 2016, May 2017

Ans. :

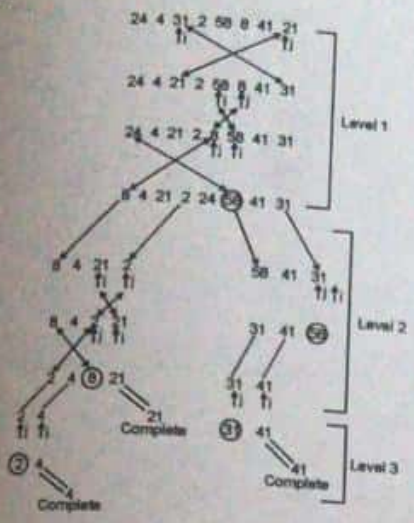
```
#include<conio.h>
#include<stdio.h>
void quick_sort(int[],int,int);
int partition(int[],int,int);
void main()
{
    int a[30],n,i;
    printf("\nEnter no of elements :");
    scanf("%d",&n);
    printf("\nEnter array elements :");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    quick_sort(a,0,n-1);
    printf("\nSorted array is :");
    for(i=0;i<n;i++)
        printf("%d",a[i]);
    getch();
}
void quick_sort(int a[],int l,int u)
{
    int j;
    if(l<u)
    {
        j=partition(a,l,u);
        quick_sort(a,l,j-1);
        quick_sort(a,j+1,u);
    }
}
int partition(int a[],int l,int u)
{
    int v,i,j,temp;
    v=a[l];
    i=l;
    j=u+1;
    do
    {
        do
            i++;
        while(a[i]<v && i<=u);

        do
            j--;
        while(v<a[j]);
        if(i<j)
        {
            temp=a[i];
            a[i]=a[j];
            a[j]=temp;
        }
    }while(i<j);
    a[l]=a[j];
    a[j]=v;
    return(j);
}
```

Output :

```
Enter no of elements   :   4
Enter array elements   :   23  1   55   33
Sorted array is   : 1   23   33   55
```

Q. 12 Sort the following sequence of elements using quick sort method. 24, 4, 31, 2, 58, 8, 41, 21 May 2014, Dec. 2016

Ans. :



(Best Sellers) Fig. 6.7

Q. 13 Write a program in 'C' to perform quick sort. May 2014

Ans. :

Program for non-recursive quick sort :

```
#include<stdio.h>
# define MAX 20
#include<conio.h>
typedef struct stack
{
   int data[50];
   int top;
}stack;
int empty(stack *);
void init(stack *);
void push(stack *,int);
int pop(stack *);
int partition(int[],int,int);
void quick_sort(int[],int,int);
void main()
{
  int a[30],n,i;
  printf("\n No. of elements :");
  scanf("%d",&n);
  printf("Enter elements :");
  for(i=0;i<n;i++)
     scanf("%d",&a[i]);
     quick_sort(a,0,n-1);
     printf("\n Sorted data ->\n");
     for(i=0;i<n;i++)
        printf("%d",a[i]);
        getch();
}
void quick_sort(int a[],int l,int u)
{
     stack s;
     int i,j;
     init(&s);
     while(i<u)
     {
        j=partition(a,l,u);
        push(&s,j+1);
        push(&s,u);
        u=j-1;
     }
     while(!empty(&s))
     {
        l=pop(&s);
        u=pop(&s);
        while(l<u)
        {
           j=partition(a,l,u);
           push(&s,j+1);
           push(&s,u);
           u=j-1;
        }
     }
}
void init(stack *s)
{
     s->top=-1;
}
int empty(stack *s)
{
     if(s->top==-1)
        return(1);
     return(0);
}
void push(stack *s,int x)
{
     s->top=s->top+1;
     s->data[s->top]=x;
}
int pop(stack *s)
{
     int x;
     x=s->data[s->top];
     s->top=s->top-1;
     return(x);
}
int partition(int a[],int l,int u)
{
  int v,i,j,temp;
```

```
v=a[l];
i=l;
j=u+1;
do
{
  do
     i++;
  while(a[i]<v && i<=u);
  do
     j--;
  while(v<a[ j]);
  if(i<j)
  {
     temp=a[i];
     a[i]=a[ j];
     a[ j]=temp;
  }
 }while(i<j);
 a[l]=a[ j];
 a[ j]=v;
 return( j);
}
```

Output

```
Enter elements : 234   11   555   54
Sorted data   -> 11    54   234   555
```

Q. 14 Explain role of Pivot in Efficiency of Quick Sort.

Ans. : Role of pivot in efficiency of quick sort

A choice of pivot, always influences the efficiency of quick sort algorithm. The popular choice of pivot as the first element is acceptable when input data is random. If the input data is in either ascending or descending order, pivot will provide a poor partition. Quick sort will require quadratic time for sorting.

Quick sort will give its best behaviour when partition is found near center of the input data. In such a case, quick sort will sort the data in (n log n) time. Median of the array is good choice for partition. Unfortunately, it is hard to calculate and would slow down quick sort. A good estimate can be obtained by picking three elements randomly and using the median of these three as pivot. As an alternative, input data can be randomised and subsequently, first element can be taken as pivot.

Q. 15 Compare sorting algorithm w.r.t. (i) sort stability (ii) efficiency (iii) passes.

Dec. 2014, May 2017

Ans. :

Comparison of sorting algorithm w.r.t. (i) sort stability (ii) efficiency(iii) passes

Sorting algorithm	Efficiency	Passes	Sort stability
Bubble sort	$O(n^2)$	n – 1	Stable
Selection sort	$O(n^2)$	n – 1	Stable
Insertion sort			
Best case	O(n)	n – 1	Stable
Worst case	$O(n^2)$	n – 1	
Quick sort			
Best case	O(n log n)	log n	Unstable
Worst case	$O(n^2)$	n – 1	
Merge sort	O(n log n)	log n	Stable
Shell sort			
Best case	O(n)	log n	Unstable
Worst case	$O(n^2)$	log n	
Radix sort	O(n)	No. of digits in the largest number	Stable

Q. 16 "External sorting calls for internal sorting as well". Justify.

Ans. :

Every sorting algorithm is based on passes. Inside a pass, records are compared on key values and the records can be shuffled based on the outcome of comparison. In case of external sorting, records to be compared are read into memory variables and they are re-written at appropriate places.

Pseudo code for bubble sort for external sorting

```
n ← no of records in the file ;
rec1, rec2, : record type ;
for (i = 1 ; i < n ; i ++)
   for (j = 0 ; j < n – i ; j ++)
      {  rec1 ← read the jth record ;
         rec2 ← read the ( j + 1)th record ;
         if (rec1 · key > rec2 · key)
           {  write rec2 at jth place ;
              write rec1 at ( j + 1) th place ;
           }
      }
```

Q. 17 What is Hashing ?

May 2014, Dec. 2016

Ans. : Hashing

Sequential search requires, on the average O(n) comparisons to locate an element. So many comparisons are not desirable for a large database of elements.

Binary search requires much fewer comparisons on the average O (log n) but there is an additional requirement that the data should be sorted. Even with best sorting algorithm, sorting of elements require O(n log n) comparisons. There is another widely used technique for storing of data called "hashing".

It does away with the requirement of keeping data sorted (as in binary search) and its best case timing complexity is of constant order (O(1)). In its worst case, hashing algorithm starts behaving like linear search. Best case timing behaviour of searching using hashing = O(1). Worst case timing Behaviour of searching using hashing = O(n).

Since, there is a large gap between its best case O(1) and worst case O(n) behaviour. It should be implemented properly to get an average case behaviour close to O(1). In hashing , the record for a key value "key", is directly referred by calculating the address from the key value. Address or location of an element or record, x, is obtained by computing some arithmetic function f.f(key) gives the address of x in the table.

(a) Table used for storing of records is known as hash table. Function f(key)is known as hash function.

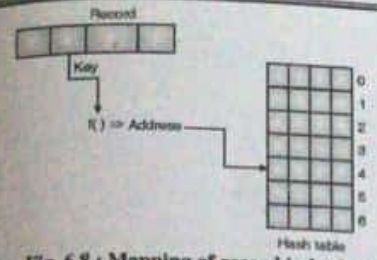


Fig. 6.8 : Mapping of record in hash table

Example

Suppose, we wish to implement a hash table for a set of records where the key is a member of set. Set K of strings.

K= {"aaa", "bbb", "ccc", "ddd" "eee", "fff", "ggg"}

A function f : key → Index is given by the following table :

N	f(x)
"aaa"	0
"bbb"	1
"ccc"	2
"ddd"	3
"eee"	4
"fff"	5
"ggg"	6

Hash table can be implement using an array of records of length n = 7.

To store a record with key x, we simply store it at position f(x) in the array. Similarly, to locate the record having key = x, we simply check to see if it is found at position f(x).

Hash Table Data Structure

There are two different forms of hashing.

(a) Open hashing or external hashing

(b) Close hashing or internal hashing

Open or external hashing, allows records to be stored in unlimited space (could be a hard disk). It places no limitation on the size of the tables. Closed or internal hashing, uses a fixed space for storage and thus limits the size of hash table.

(a) Open Hashing Data Structure :

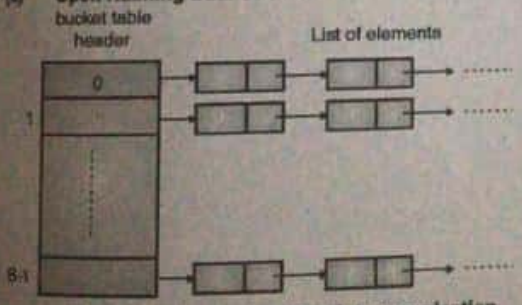


Fig. 6.9 : The open hashing data organization

Fig. 6.9 gives the basic data structure for open hashing.

The basic idea is that the records [elements] are partitioned into B classes, numbered 0,1,2.. B–1. Hashing function f(x) maps a record with key x to an integer value between 0 and B–1. if a record is mapped to location 1 then we say the record is mapped to bucket 1 or the record belongs to class 4. Each bucket in the bucket table is the head of the linked list of records mapped to that bucket.

(b) Closed Hashing Data Structure :

A closed hash table keeps the elements in the bucket itself. Only one element can be put in the bucket. If we try to place an element in the bucket f(n) and find it already holds an element, then we say that a collision has occurred. In case of collision, the element should be rehashed to alternate empty location $f_1(x)$, $f_2(x)$, ... within the bucket table. In closed hashing, collision handling is a very important issue.

Fig. 6.10 : Partially filled hash table

Q. 18 What is meant by collisions ?

May 2014, Dec. 2016

Ans. :

Collision Resolution Strategies (Synonym Resolution)

Collision resolution is the main problem in hashing. If the element to be inserted is mapped to the same location, where an element is already inserted then we have a collision and it must be resolved.

There are several strategies for collision resolution. The most commonly used are :

(a) Separate chaining – used with open hashing.

(b) Open addressing – used with closed hashing.

(a) Separate Chaining

In this strategy, a separate list of all elements mapped to the same value is maintained. Fig. 6.11 shows an implementation of separate chaining. Separate chaining is based on collision avoidance. If memory space is tight, separate chaining should be avoided. Additional memory space for links is wasted in storing address of linked elements.

Hashing function should ensure even distribution of elements among buckets, otherwise the timing behaviour of most operations on hash table will deteriorate.

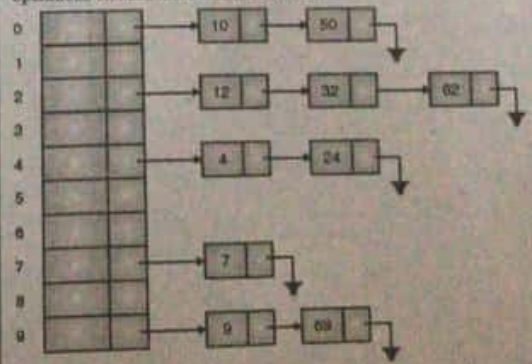


Fig. 6.11 : A separate chaining hash table

In the Fig. 6.11 :

(a) A simple hash function hash (x)= x mod 10 is taken.
Bucket table has a size of 10.

To perform a **find**, hash function is used to find the list to be traversed for searching. Then the list is traversed in normal manner. To perform an **insert**, the appropriate list is traversed to ensure that the element to be inserted is not present in the list. If the element is not present then it is inserted at the front.

Data structure for separate chaining :

```
#define MAX 10
typedef struct node
{
    int data;
    struct node *next;
};
node *hashtable[MAX]; //bucket table header
```

Operations on hash table :

(a) Initialise() - Header of all lists are set to NULL.

```
void initialise(node *hashtable[])
{
    int i;
    for(i=0;i<MAX;i++)
        hashtable[i]=NULL;
}
```

(b) Insert ( ) - If the item is already present then do nothing; otherwise, place the item at the front of the list.

```
void insert(node *hashtable[] , int x)
    {
        int loc;
        node *p,*q;
        loc=x % MAX;  // Mapped location
        //get memory for new node
        q=(node*)malloc(sizeof(node));
        q->data=x;
        q->next=NULL;
        if(hashtable[loc]==NULL)
            hashtable[loc]=q;
        else
            { // locate the last node of the linked list
                for(p=hashtable[loc];p->next!=NULL;p=p->next) ;
                p->next=q;
            }
    }
```

(c) Find ( ) – The function returns a pointer to the node containing the search key.

```
node *find(node *hashtable[],int x)
{
    int loc;
    node *p;
    loc=x % MAX;
    p=hashtable[loc];
    while(p!=NULL && x!=p->data)
        p=p->next;
    return(p);
}
```

Q. 19 **Using linear probing and quadratic probing insert the following values in a hash table of size 10. Show how many collisions occur in each technique : 99, 33, 23, 44, 56, 43, 19.** Dec. 2013, May 2014, Dec. 2016

Ans. :

1. Linear probing :

	Empty table	After 99	After 33	After 23	After 44	After 56	After 43	After 19	
0								19*	* = Collision
1									No. of collisions = 4
2									
3			33	33	33	33	33	33	
4				23*	23*	23*	23*	23*	
5					44*	44*	44*	44*	
6						56	56	56	
7							43*	43*	
8									
9		99	99	99	99	99	99	99	

2. **Quadratic probing :**

	Empty table	After 99	After 33	After 23	After 44	After 56	After 43	After 19
0								19*
1								
2								
3			33	33	33	33	33	33
4				23*	23*	23*	23*	23*
5					44*	44*	44*	44*
6						56	56	56
7							43*	43*
8								
9		99	99	99	99	99	99	99

* = Collision
No. of collisions = 4

Q. 20 **Hash the following in table of size 11. Use any two collision resolution techniques**
23, 55, 10, 71, 67, 32, 100, 18, 10, 90, 44

May 2016

Ans. :

Method I : Linear Probing

1. 23 % 11 = 1, 55 % 11 = 0 10 % 11 = 10, 71 % 11 = 5
67 % 11 = 1 32 % 11 = 10 100 % 11 = 1, 18 % 11 = 7
10 % 11 = 10, 90 % 11 = 2 44 % 11 = 0

2.

	23(1)	55(0)	10(10)	71(5)	67(1)	32(10)	100(1)	18(7)	10(10)	90(2)	44(0)
0	–	55	55	25	25	25	25	25	25	25	25
1	23	23	23	23	23	23	23	23	23	23	23
2	–	–	–	–	67	67	67	67	67	67	67
3	–	–	–	–	–	32	32	32	32	32	32
4	–	–	–	–	–	–	100	100	100	100	100
5	–	–	–	71	71	71	71	71	71	71	71
6	–	–	–	–	–	–	–	–	10	10	10
7	–	–	–	–	–	–	–	18	18	18	18
8	–	–	–	–	–	–	–	–	–	90	90
9	–	–	–	–	–	–	–	–	–	–	44
10	–	–	10	10	10	10	10	10	10	10	10

Method 2 : Separate chaining hash function = data % 11.

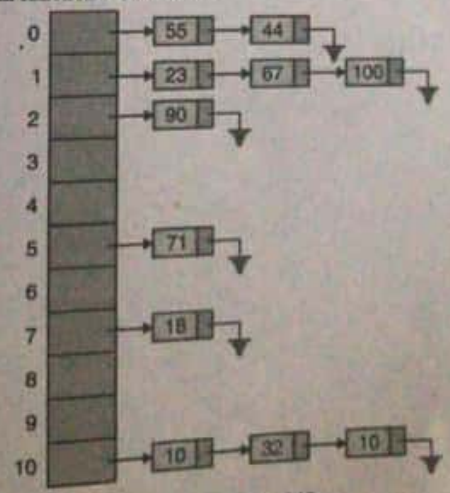


Fig. 6.12

Q. 21 **Using Linear probing and Quadratic probing insert the following values in a hash table of size 10. Show how many collisions occur in each iteration : 28, 55, 67, 11, 10, 90, 44.** May 2017

Ans. :

1. Linear probing :

	Empty table	After 28	After 55	After 71	After 67	After 1	After 10	After 90	After 44
0							10	10	10
1				71	71	71	71	71	71
2						1	1	1	1
3								90	90
4									44
5			55	55	55	55	55	55	55
6									
7					67	67	67	67	67
8	28	28	28	28	28	28	28	28	28
9									

Number of collisions = 2

2. Quadratic Probing :

	Empty table	After 28	After 55	After 71	After 67	After 1	After 10	After 90	After 44	
0							10	10	10	
1				71	71	71	71	71	71	
2						1	1	1	1	*
3								90	9	*
4										
5			55	55	55	55	55	55	55	
6										
7					67	67	67	67	67	
8		28	28	28	28	28	28	28	28	
9									44	*