

# UNIT 1. INTRODUCTION TO DATA STRUCTURES

## 1.1 INFORMATION AND ITS MEANING

A computer is a device used to manipulate the information. The study of computer science includes the study of how information is stored in a computer how it can be edited, and how it can be used as per user's requirements. Even though the concepts of information is a very basic essential for the field of computer science, it is hardly difficult to define the term information. But it can be explained using the measurable quantities. The basic unit of information in Computer science is 'bit'; which will take any one of two values. The binary digits 0 and 1 are used to represent the two possible status of a particular bit of information.

## 1.2 ABSTRACT DATA TYPES (ADT)

The term abstract data type refers to the basic mathematical concept that defines the data type. ADT will specify the logical properties of a data type. It is a useful tool for implementers and programmers who wish to use the data type correctly. Whenever any new data type (user-defined data type) is to be created, a prototype of its nature, the possible operations on it etc. have to be thought of. In such a situation, ADT helps in forming a prototype. Note that, the representation of ADT do not follow the syntax of any programming language.

Even though there are several methods to specify ADT, we use the semiformal method, which will adopt C notations.

For illustration, consider rational number in the form –

$$\left\{ \frac{p}{q} / p, q \in Z \text{ and } q \neq 0 \right\}$$

The above equation indicates that any rational number will be in the form of p divided by q, where p and q are integers (the set Z) and the denominator q is not equal to zero.

In this case, the sum of two rational numbers ( $a_1/a_2$ ) and ( $b_1/b_2$ ) would be –

$$\frac{a_1}{a_2} + \frac{b_1}{b_2} = \frac{a_1 * b_2 + b_1 * a_2}{a_2 * b_2}$$

The specification of any ADT consists of two parts –

- **value definition**
  - here, we specify the set of possible values taken by the ADT along with some conditions or constraints with which the ADT bounds.
- **operator definition**
  - here, various operations which are imposed on ADT are defined. This part contains 3 sections viz. a header, the preconditions (which is optional) and

the postconditions. The term 'abstract' in the header indicates that this is not a C function; rather it is an ADT operator definition. This term also indicates that the role of an ADT is a purely logical definition of a new data type.

The following listing gives the ADT for rational numbers. The value definition here indicates the constraints on rational number. The operator definition parts contains the definition for various operations like creation of rational number, addition and multiplication of rational numbers and for checking the equality of two rational numbers.

```
// value definition
abstract typedef<integer, integer> RATIONAL;
condition RATIONAL[1] != 0;

//Operator definition
abstract RATIONAL createrational (a, b)
int a,b;
precondition b!=0;
postcondition createrational [0] == a;
                createrational [1] == b;

abstract RATIONAL add(a,b)
RATIONAL a, b;
postcondition add[0] == a[0]*b[1] + b[0]*a[1];
                add[1] == a[1]*b[1];

abstract RATIONAL mul(a,b)
RATIONAL a, b;
postcondition mul[0] == a[0]*b[0];
                mul[1] == a[1]*b[1];

abstract equal(a, b)
RATIONAL a,b;
postcondition
    equal == (a[0]*b[1] == b[0]*a[1]);
```

### 1.2.1 Sequences as Value Definitions

While developing the specification for various data types, to specify the value of an ADT, we use set notation or the notation of sequences. Basically, a sequence is an ordered set of elements denoted by,

$$S = \langle S_0, S_1, \dots, S_{n-1} \rangle$$

If  $S$  contains  $n$  elements then  $S$  is said to be of length  $n$ . We assume the functions  $\text{len}(S)$ ,  $\text{first}(S)$ ,  $\text{last}(S)$  etc to denote length, first element and last element respectively. A sequence with zero length is called *nilseq*.

Various syntaxes for ADT specifications are as below –

- To define an ADT viz. `adt1` whose values are the elements of a sequence, we write –

```
abstract typedef<<type1>> adt1;
```

Here, *type1* indicates the data type of elements in the sequence.

- To denote an ADT taking the values of different data types *type1*, *type2*, etc. we write –  

```
abstract typedef<type1,type2,...,type n> adt2;
```
- To denote an ADT having a sequence of length *n*, where all elements are of same data type, we write –  

```
abstract typedef<<type, n>> adt3;
```

Two sequences are said to be equal if their corresponding elements are equal. If *S* is a sequence, the function *sub(S,i,j)* refers to the subsequence of *S* starting at the position *i* in *S* and consisting of *j* consecutive elements. The concatenation of two sequences *S1* and *S2* is the sequence consisting of all elements of *S1* followed by the elements of *S2*.

### 1.2.2 ADT for varying length character strings

Using the sequence notation, the specification of ADT for the varying-length character string can be illustrated. Normally, these are four basic operations for strings viz.

- *length* is a function that returns length of string
- *concat* is returns concatenation of two strings
- *pos* is returns the first position of one string in the other
- *substr* is returns a substring of given string.

```
abstract typedef <<char>> STRING;
```

```
abstract length(s)
STRING s;
postcondition length = len(s);
```

```
abstract STRING concat(s1,s2)
STRING s1, s2;
postcondition concat == s1 + s2;
```

```
abstract STRING substr(s1, i, j)
STRING s1;
int i, j;
postcondition substr == sub(s1, i, j)
```

### 1.3 DATA TYPES

The C language has 4 basic data types viz. *int*, *float*, *char* and *double*. We can apply the qualifiers like *short*, *long* and *unsigned* to the data type *int*. The maximum sizes implied by *short*, *long* or *int* vary from machine to machine. The unsigned integer can store only positive integers. The declaration of variable in C does two tasks. First, it specifies the

amount of storage that must be allocated for objects declared with that type. For ex, an integer variable declaration tells the compiler to reserve 2 bytes (16 bit machine) of memory. Second, it specifies how data represented by strings of bits are to be interpreted.

**(NOTE: Detailed discussion of data types, pointers, arrays and strings in C is not done here, as the students would have studied it in 1<sup>st</sup> semester).**

## 1.4 POINTERS

A variable which will be used to hold the address of some other variable is called pointer variable. For ex, if *i* is declared as an integer, then *&i* refers to the location that has been set aside to contain the value of *i*. The declaration of pointer variables will be like-

```
int *pi;
float *pf;
char *pc;
```

Here, *pi* is a pointer which will contain the address of an integer variable. Similarly *pf* indicates that it can contain an address of floating point variable. The assignment of pointer variable can be like-

```
int a, *p;
a=10;
b=&a;
```

Here, in the pointer variable *p*, the address of the variable 'a' is going to be stored. And *\*p* will be nothing but the value of 'a' i.e. 10. If *pi* is a pointer to an integer, then (*pi +1*) is the pointer to the integer immediately following the integer *\*pi* in memory, whereas, (*pi-1*) is the pointer to the integer immediately preceding *\*pi*. Note that (*\*p+1*) refers to 1 added to the integer *\*pi*, while *\*(p+1)* refers to the integer following the integer at location *pi*.

## 1.5 DATA STRUCTURES

The study of data structures in C deals with the study of ***how the data is organized in the memory, how efficiently the data can be retrieved from the memory, how the data is manipulated in the memory and the possible ways in which different data items are logically related.*** Thus, we can understand that the study of data structure involves the study of memory as well.

The study of data structures also involves the study of how to implement the developed data structures using the available data structures in C. Since the problems that arise which implementing high-level data structures are quite complex, the study will allow to investigate the C language more thoroughly and to gain valuable experience in the use of this language. While implementing data structure, one should take care of efficiency, which involves two facts viz. time and space. That is, a careful evaluation of time complexity and space complexity should be made before data structure implementation.

Types of data structures can be depicted as given in Figure 1.1. Linear data structures are those in which the relationship between the elements is linear/sequential. If the relationship between the elements is hierarchical, then it is called as non-linear data structures.

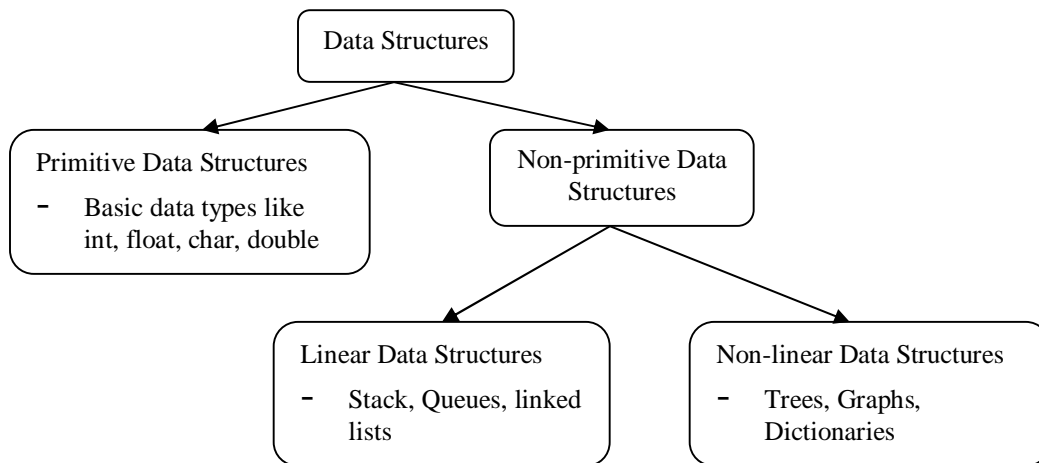


Figure 1.1 Types of Data Structures

## 1.6 ARRAYS

Array is one of the composite data structures which may be of 'n' dimension. In other words, array is an ordered set of elements, which are of same data types, having the finite number of elements. The general form of array declaration will be –

```
data_type array_name[size];
```

Each element of array is referred by its index. The smallest value of array index is known as *lower bound* and the largest value of array index is known as *upper bound* of an array. The number of elements that an array can hold is called as *range* or the *size* of an array and is given by (upper bound – lower bound + 1). Array can be passed to a function as a whole. Since an array variable in C is a pointer (base address of the array), while passing, it is passed by reference and not by value.

### 1.6.1 Array as an ADT

An array is a derived data type. Hence, it can be represented as an ADT. The following listing gives ADT for an array. It contains the value definition indicating how an array should look like. The operator definition part indicates two major operations of the array viz. extracting an element from the array and storing an element into the array.

```
//value definition part
Abstract typedef <<eltype, ub>> ARRTYPE (ub, eltype);
Condition type(ub)==int;
```

```
//operator definition part
abstract eltype extract(a,i)
ARRTYPE(ub, eltype) a;
int i;
precondition 0<= i <ub;
postcondition extract==a;
```

```
abstract store(a,i, elt)
```

```
ARRTYPE(ub, eltype) a;  
int i;  
eltype elt;  
precondition 0<= i <ub;  
Postcondition a[i]==elt;
```

### 1.6.2 Using One-dimensional Array

Following is a simple example for using one dimensional array. This program is to compute sum and average of n numbers.

#### Program 1.1 Example of 1-D array – finding sum and average of n numbers

```
#include<stdio.h>  
#include<conio.h>  
void main()  
{  
    int a[10], n, i, sum=0;  
    float avg;  
    clrscr();  
  
    printf("Enter size of array:");  
    scanf("%d", &n);  
    printf("\nEnter elements:\n");  
  
    for(i=0;i<n;i++)  
    {  
        scanf("%d",&a[i]);  
        sum+=a[i];  
    }  
    avg=(float)sum/n;  
    printf("\nSum=%d, and Avg=%f", sum,avg);  
}
```

### 1.6.3 Array as Parameters

The example given below indicates how to pass an array (in other words, base address of the array) to a function. This program is also for finding sum and average of n numbers.

#### Program 1.2 Passing array to a function – finding sum and average of n numbers

```
#include<stdio.h>  
#include<conio.h>  
  
float Avg(int x[], int n)  
{  
    int sum=0,i;  
    float avg;
```

```
        for(i=0;i<n;i++)
            sum+=x[i];

        avg=(float)sum/n;
        return avg;
    }

void main()
{
    int a[10], n, i;
    float a;
    clrscr();

    printf("Enter size of array:");
    scanf("%d", &n);

    printf("\nEnter elements:\n");

    for(i=0;i<n;i++)
        scanf("%d", &a[i]);

    a=Avg(a,n);
    printf("\n Avg=%f", avg);
}
```

#### 1.6.4 Character Strings

A character string (or just a string) in C is nothing but an array of characters. Following simple program is to read a string and to find the length of the string.

#### Program 1.3 Example of string – finding length of a string

```
#include<stdio.h>
#include<conio.h>
void main()
{
    char str[20];
    int i;

    printf("Enter a string:");
    scanf("%s", str);

    for(i=0;str[i]!='\0';i++);

    printf("\nLength of string is %d", i);
}
```

## UNIT 2. THE STACK

### 2.1 DEFINITION AND MEANING

Stack is non-primitive linear data structure into which, new items may be inserted and from which items may be deleted. In this data structure, the insertion and deletion of the elements are carried out at one end and is called as 'top' of the stack. In stack, the element last inserted will be the first to be deleted. Hence, stack is known as Last In First Out (LIFO) structure.

For example, consider the task of keeping books on a table one above the other. When a person wants to take the book, he has to take the book which was kept last. Thus, the first book kept on the table will be the last book to be taken out.

In the study of data structures, insertion of new item into the stack is called as **push** operation; whereas, deletion of an item at the top of the stack is **pop** operation. When the stack is full, we can't insert any more elements. This situation is called as **stack overflow**. Similarly, when stack is empty, we can't delete element from it. This condition is known as **stack underflow**.

An integer variable **top** is normally used for denoting current status of the stack – that is, the value of **top** gives the number of items of the stack. For programming purpose, an empty stack is denoted by setting the value of **top** as -1. Each time the **push** operation is encountered, the **top** will be incremented. When the **pop** operation is done, **top** will be decremented. Usually, we define the size of the stack at the beginning. For example, stack of 5 elements, stack of 10 elements etc. When the value of **top** becomes -1 during deletion, it is stack underflow. When the **top** reaches the predefined size, it is stack overflow.

Figure 2.1 depicts the example of primitive operation on an integer stack of size 3.

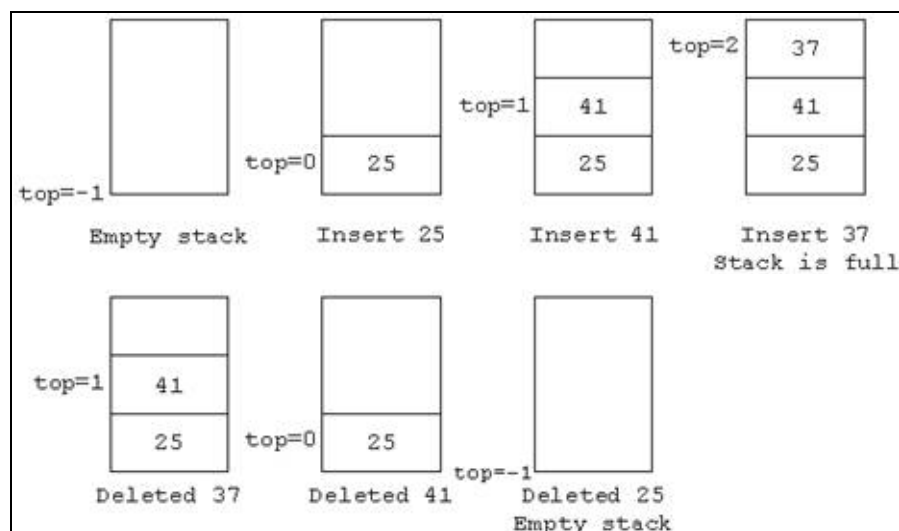


Figure 2.1 Demo of stack operations



**Program 2.1 Primitive operations on stack**

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

#define MAX 3

void push(int st[], int item, int *t)
{
    if(*t==MAX-1)
    {
        printf("Stack full!!");
        return;
    }
    st[++(*t)]=item;
}

int pop(int st[], int *t)
{
    int item;

    if(*t== -1)
    {
        printf("\nStack Empty!!");
        return ;
    }
    item=st[( *t) - -];
    return item;
}

void disp(int st[], int *t)
{
    int i;

    if(*t== -1)
    {
        printf("\nStack Empty!!");
        return;
    }

    printf("\nStack contents:\n");
    for(i=*t ; i>=0; i- -)
        printf("\n%d",st[i]);
}
```

```
void main()
{
    int st[MAX], top= -1, opt, item;

    for(;;)
    {
        printf("\n*****Stack Operations*****\n");
        printf("\n1. Push \n 2. Pop \n 3. Display \n 4. Exit\n");
        printf("Enter your option:");
        scanf("%d", &opt);

        switch(opt)
        {
            case 1: printf("\nEnter item:");
                    scanf("%d",&item);
                    push(st, item, &top);
                    break;
            case 2: item=pop(st, &top);
                    printf("\n Deleted item is %d", item);
                    break;
            case 3: disp(st, &top);
                    break;
            case 4:
            default:exit(0);
        }
    }
}
```

## 2.2 APPLICATIONS OF STACK

The concept of stacks has various applications in the field of computer science. Some of them includes conversion of arithmetic expressions, evaluation of expressions, recursion etc.

### 2.2.1 Conversion of Expressions

Let us consider an arithmetic expression

$$x \text{ op } y$$

Here, x and y are two arithmetic expressions or operands and 'op' is the arithmetic operator like +, -, \*, / etc. For example, consider a simple arithmetic expression, (x+y). Here x and y are operands and + is an operator. Representing the equation like this is known as 'infix' expression. There are two more representations for denoting an arithmetic expression:

xy+, known as postfix expression  
+xy, known as prefix expression

The words 'pre', 'post' and 'in' specifies the relative position of the operator in the expression.

Thus, any expression having an operator in between two operands is called as an ***infix expression***. Any expression having an operator followed by two operands is ***postfix expression***. An expression, in which the operator precedes the two operands, is a ***prefix expression***.

Note that an infix expression may have parentheses in-between. But, postfix and prefix expressions will not be having any parentheses. While converting a particular infix expression into either prefix or postfix expression, one should consider precedence of operators, as given below:

Precedence	Operator
1	( or ) -> parentheses
2	^ or \$ -> exponentiation
3	* or / -> Multiplication or division
4	+ or - -> Addition or subtraction

**Example:**

**Convert the following infix expression into postfix and prefix expression:**

**$((A+B)^C - (D/E) * F)$ .**

**Solution: Conversion into Postfix:**

- The given expression is  $((A+B) ^ C - (D/E) * F)$
- We have to resolve inner brackets first.
- Consider,  $(A+B)$ . This will be  $AB+$  in postfix. Let  $P = AB+$
- Consider,  $(D/E)$ . This is  $DE/$  in postfix. Let  $Q = DE/$
- Now, the expression will be  $(P^C - Q * F)$
- Now resolve operators with high precedence. That is  $^$  (power symbol) and  $*$  (multiplication)
- That is,  $P^C = PC^$  Let  $R = PC^$
- And,  $Q * F = QF^$  Let  $S = QF^$
- Then, expression is  $(R-S)$ , which when converted into postfix, gives  $RS -$
- Now, by replacing the values of R and S and in turn, the values of P and Q we will get –
  - $PC^QF^ -$
  - $AB+C^DE/F^ -$
- The required postfix expression is  **$AB+C^DE/F^ -$**

**Conversion into Prefix:**

- Consider,  $((A+B)^C - (D/E) * F)$
- As we did in conversion into postfix notation, here also, we make use of some temporary variables like P, Q etc.
- Consider  $(A+B)$ . In prefix notation, this will be  $+AB$ . Let  $P = +AB$ .
- Let  $(D/E) = /DE = Q$ .

- Now, the expression is  $(P\$C - Q*F)$
- Again,  $P\$C = \$PC = R$ , say and,  $Q*F = *QF = S$
- Then, expression is  $(R-S) = -RS$
- Now, putting the values of P, Q, R and S, we will get –
  - ⇒  $-\$PC*QF$
  - ⇒  **$-\$+ABC*/DEF$** , is the required prefix expression.

**Procedure for converting an Infix expression to Postfix expression programmatically:**

- For converting an infix expression to postfix expression, one has to analyze the precedence value of a symbol or element both in input and in stack.
- If an operator is left associative, then the input precedence value is less than the stack precedence value.
- If the operator is right associative, then, the input precedence value is greater than the stack precedence value.

The following table is used for writing an algorithm and program for conversion of infix to postfix.

Symbols	Input precedence (IP)	Stack precedence (SP)
+ or –	1	2
* or /	3	4
\$ or ^	6	5
Operands	7	8
(	9	0
)	0	-
#	-	-1

**Algorithm/Pseudo code for converting infix expression into postfix expression:**

1. Initialize empty stack with symbol '#'.  
i.e.  $st[0]='#'$
2. Read the character from infix expression.  
i.e.  $symbol=infix[ i]$
3. while  $SP(st[top]) > IP(symbol)$   
     $postfix [ j]=pop(st[top])$

```

4. if SP(st[top]) != IP(symbol)
    push(symbol)
    else
        pop(st[top])
    
```

5. Repeat the steps (2) to (4) till the last character of infix expression.


```

6. While stack becomes empty, //only for partially parenthesized expression
    postfix [ j]=pop(st[top])
    
```

Observe the following figures to understand the tracing of the above algorithm.

**Tracing an algorithm taking an example**

- Consider an infix expression  
(( A + B ) \* C - D )
- Initially, push '#' into the stack to denote an empty stack.
- Assume that we have defined the size of the stack as 15 characters.
- Then, the stack may look like this ----->

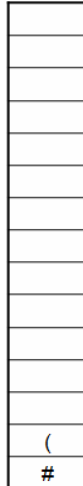


Next steps are as explained:

1. Sym = '('

- Check stack precedence of top of the stack i.e. '#' and input precedence of the current symbol i.e. '('.
- 1 > 9 ? No
- 1 != 9 ? Yes

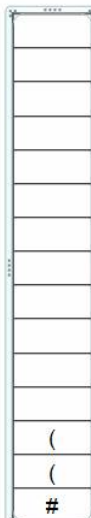
Push '(' into stack.



2. Sym = '('

- Check stack precedence of top of stack i.e. '(' and input precedence of the current symbol i.e. '('.
- 0 > 9? No
- 0 != 9? Yes


Push '(' into stack.



3. Sym = 'A'

- Check stack precedence of top of stack i.e. '(' and input precedence of the current symbol i.e. 'A'.
- 0 > 7? No
- 0 != 7? Yes

Push 'A' into stack.







```
        case '-' : return 2;
        case '*' :
        case '/' : return 4;
        case '^' :
        case '$' : return 5;
        case '(' : return 0;
        case '#' : return -1;
        default : return 8;
    }
}

void push (char item, int *top, char s[])
{
    s[++(*top)] = item;
}

char pop(int *top, char s[])
{
    return s[(--*top)];
}

void infix_to_postfix (char ifix[], char pfix[])
{
    int top = -1, i, j = 0;
    char s[30], sym;

    push('#', &top, s);

    for(i=0; i < strlen(ifix); i++)
    {
        sym = ifix[i];
        while (stackpre(s[top]) > inputpre(sym))
            pfix[j++] = pop(&top, s);

        if(stackpre(s[top]) != inputpre(sym))
            push(sym, &top, s);
        else
            pop(&top, s);
    }

    while(s[top] != '#')
        pfix[j++] = pop(&top, s);

    pfix[j] = '\0';
}
```



```

void main()
{
    char ifix[20], pfix[20];
    clrscr();

    printf("Enter valid infix expression\n");
    scanf("%s", ifix);
    infix_to_postfix (ifix,pfix);
    printf("The postfix expression is = %s", pfix);
}

```

### Procedure for converting an Infix expression to Prefix expression:

To convert an infix expression into prefix expression, we have to use the following precedence table.

Symbols	Input precedence (IP)	Stack precedence (SP)
+ or -	2	1
* or /	4	3
\$ or ^	5	6
Operands	7	8
(	0	--
)	9	0
#	-	-1

**Algorithm:** The algorithm to convert an infix expression into prefix expression is given below:

- Reverse the given infix expression.
- Follow the procedure (using above table) for obtaining postfix expression.
- Reverse the result obtained.

### Program 2.3 Converting a valid infix expression into the prefix expression:

```

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

int inputpre(char sym)
{
    switch(sym)
    {
        case '+':
        case '-': return 2;
        case '*':
        case '/': return 4;
    }
}

```

```
        case '^' :
        case '$' : return 5;
        case '(' : return 0;
        case ')' : return 9;
        default : return 7;
    }
}

int stackpre(char sym)
{
    switch(sym)
    {
        case '+' :
        case '-' : return 1;
        case '*' :
        case '/' : return 3;
        case '^' :
        case '$' : return 6;
        case ')' : return 0;
        case '#' : return -1;
        default : return 8;
    }
}

void push (char item, int *top, char s[])
{
    s[++(*top)] = item;
}

char pop(int *top, char s[])
{
    return s[(--*top)];
}

void infix_to_prefix (char ifix[], char pfix[])
{
    int top = -1, i, j = 0;
    char s[30] , sym;

    push('#', &top, s);
    strrev(ifix);

    for(i=0; i < strlen(ifix); i++)
    {
        sym = ifix[i];
```

```
        while (stackpre(s[top]) > inputpre(sym))
            pfix[j++] = pop(&top,s);

        if(stackpre(s[top]) != inputpre(sym))
            push(sym,&top,s);
        else
            pop(&top,s);
    }

    while(s[top] != '#')
        pfix[j++] = pop(&top,s);

    pfix[j] = '\0';
    strrev(pfix);
}

void main()
{
    char ifix[20], pfix[20];
    clrscr();

    printf("Enter valid infix expression\n");
    scanf("%s", ifix);
    infix_to_prefix (ifix,pfix);
    printf("The prefix expression is = %s", pfix);
}
```

**Evaluation of Postfix expression:**

To evaluate an infix expression, we will scan from left to right repeatedly. But if the expression contains parentheses, evaluation becomes complex as the parentheses changes the order of precedence. So, it is always easy to evaluate an infix expression by converting it into either prefix or postfix expression.

**Algorithm** to evaluate postfix expression:

1. Scan a symbol in postfix expression from left to right.
2. If the symbol is operand, push it into stack.
3. If the symbol is an operator, pop two elements from the stack and perform the operation indicated.
4. Push the result of step (3) into the stack.
5. Repeat the above steps till all the symbols get exhausted in the given postfix expression.
6. Now, pop the element from the stack, which will be the result of entire postfix expression.

Note that, here, a single digit is treated as an operand and scanned.



```
        case '-': return op1 - op2;
        case '*': return op1 * op2;
        case '/': if(op2== 0)
            {
                printf("Can't evaluate");
                exit(0);
            }
            return op1 / op2;
        case '^':
        case '$': return pow(op1,op2);
    }
}

void push(float item, int *top, float s[ ])
{
    s[++(*top)] = item;
}

float pop(int *top, float s[ ])
{
    return s[(--*top)];
}

void main()
{
    float s[20], result, op1, op2, x;
    int top = -1, i;
    char postfix[20], sym;

    printf("Enter valid postfix expression\n");
    scanf("%s",postfix);

    for(i=0;i<strlen(postfix);i++)
    {
        sym = postfix[i];

        if(isdigit(sym))
            push(sym-'0', &top, s); // character to digit conversion
        else if (isalpha(sym))
        {
            printf("Enter the value of %c: ", sym);
            scanf("%f",&x);
            push(x,&top,s);
        }
        else
        {
```

```
        op2 = pop(&top,s);
        op1 = pop(&top,s);
        result = oper(sym,op1,op2);
        push(result,&top,s);
    }
}
result = pop(&top,s);
printf("Result =%.4f",result);
}
```

**Sample Output 1:**

```
Enter valid postfix expression: 941-3*/
Result = 1.0000
```

**Sample Output 2:**

```
Enter valid postfix expression: abc-d*/
Enter value of a: -54
Enter value of b: 23
Enter value of c: 5
Enter value of d: 8
Result = -0.3750
```

**NOTE:** The **sample output 1** takes the postfix expression with digits. Hence, each digit has to be treated as one operand. Whereas, in the **Sample Output 2**, the postfix expression is a series of alphabets (variables). So, the program will allow you to read the values for each of these variables. And hence, the user can give an operand containing more than one digit, a floating point number, a negative number etc. In the **Sample Output2**, the meaning of expression (in infix format) will be –

$$- 54 / ((23-5)*8) = - 0.3750.$$

## UNIT 3. RECURSION

### 3.1 Definition and meaning

Recursion is a technique of defining something in terms of itself. In the field of mathematics and computer science, many concepts can be explained using recursion. In computer terminology, if a function calls itself then it is known as recursive function. If the function calls itself directly, then it is known as *direct recursion*.

Example:

```
void myfun()
{
    ----
    ----
    myfun();
    ----
    ----
}
```

If the function calls itself through another function, then it is known as *indirect recursion*. For example:

```
void myfun()
{
    ----
    ----
    fun();
    ----
    ----
}
void fun()
{
    -----
    -----
    myfun();
    -----
    -----
}
```

Here, the function myfun() is calling the function fun(), which in turn calls myfun().

### 3.2 Factorial of a Number

Factorial of a non-negative integer  $n$  is defined as the product of all integers from 1 to  $n$ . That is:

$$n! = n(n-1)(n-2)\dots 3.2.1, \text{ for all } n \geq 1 \text{ and} \\ 0! = 1.$$

By definition we have,  $n! = n(n-1)!$

Thus, factorial is a recursive function.

Consider one example of finding 5! –

$$\begin{aligned}
 5! &= 5*(5-1)! \\
 &= 5*4! \\
 &= 5*4*3! \\
 &= 5*4*3*2! \\
 &= 5*4*3*2*1! \\
 &= 5*4*3*2*1*0! \\
 &= 5*4*3*2*1*1 \\
 &= 120
 \end{aligned}$$

This procedure can be implemented through program given below:

```

#include<stdio.h>

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

void main()
{
    int n;

    printf("Enter the value of n");
    scanf("%d", &n);

    if(n<0)
        printf("Invalid input");
    else
        printf("\nFactorial of %d is %d ", n, fact(n)) ;
}

```

**The output would be:**

```

Enter the value of n: 6
The factorial of 6 is 720

```

### 3.3 Multiplication of two Natural Numbers

The product of two natural numbers say, m and n is nothing but the sum obtained by adding m for n number of times. That is,

$$m*n = m+m+\dots +m \text{ (for } n \text{ times).}$$

This can be given recursively as –

$$\begin{aligned}
 m*n &= m*(n-1) + m && \text{for } n>1 \\
 &= m && \text{for } n=1
 \end{aligned}$$



Consider one example of multiplying 4 and 3.

Let  $m=4$  and  $n=3$ . Then

$$\begin{aligned}
 4*3 &= 4*(3-1) + 4 \\
 &= 4*2 + 4 \\
 &= 4 * (2-1) + 4 + 4 \\
 &= 4 * 1 + 4 + 4 \\
 &= 12
 \end{aligned}$$

**Program:**

```

#include<stdio.h>
int mul(int a, int b)
{
    if(b==1)
        return a;
    return a + mul(a, b-1);
}

void main()
{
    int m, n;
    printf("Enter the values of m and n");
    scanf("%d%d", &m, &n);

    if(m<=0 || n<=0)
    {
        printf("Invalid input");
        return;
    }
    printf("Product of %d and %d is %d", m, n, mul(m,n));
}

```

### 3.4 Fibonacci Sequence

A sequence of integers is called Fibonacci sequence if an element of a sequence is the sum of its immediate two predecessors. That is:

$$\begin{aligned}
 f(x) &= 0 && \text{for } x=1 \\
 f(x) &= 1 && \text{for } x=2 \\
 f(x) &= f(x-2) + f(x-1) && \text{for } x>2
 \end{aligned}$$

Thus the Fibonacci sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, ....

Consider an example of finding 6<sup>th</sup> Fibonacci number.

$$\begin{aligned}
 f(6) &= f(4) + f(5) \\
 &= \underline{f(2) + f(3)} + \underline{f(3) + f(4)} \\
 &= 1 + \underline{f(1) + f(2)} + \underline{f(1) + f(2)} + \underline{f(2) + f(3)} \\
 &= 1 + 0 + 1 + 0 + 1 + 1 + f(1) + (2)
 \end{aligned}$$

$$\begin{aligned} &= 4 + 0 + 1 \\ &= 5 \end{aligned}$$

The above procedure can be illustrated through the program given below:

### Finding n<sup>th</sup> Fibonacci number

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

```
int fibo(int n)
{
    if (n==1)
        return 0;
    else if (n==2)
        return 1;
    return fibo(n-1) + fibo (n-2);
}

void main()
{
    int n;

    printf("Enter value of n");
    scanf("%d", &n);

    if(n<=0)
        printf("invalid input");
    else
        printf("%d th fibonacci number is %d ", n, fibo(n));
}
```

#### The output would be:

```
Enter the value of n : 10
10th fibonacci number is 34
```

## 3.5 Binary Search

This method is applied on the sorted array. Initially, the key element is compared with the middle element of the array. If they are equal then the search is successful. Otherwise, the array is divided into two parts viz. from the first element to middle element and from middle to last element. If the key element is greater than the middle element then the second sub-array is searched for. If the key is less than the middle then the first sub-array is searched. The procedure is repeated till the key is found or till the sub-array contains a non-matching single element.

#### Program for Binary Search:

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

```
#include<conio.h>

int binary(int item, int a[],int low,int high)
{
    int mid;

    if(low<=high)
    {
        mid=(low+high)/2;

        if(a[mid]==item)
            return mid+1;
        if(a[mid]>item)
            return binary(item,a,low,mid-1);

        return binary(item,a,mid+1,high);
    }
    return -1;
}

void main()
{
    int a[10],n,pos,item, i;

    printf("Enter the array size:");
    scanf("%d",&n);
    printf("Enter the elements in ascending order\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("Enter the key element:");
    scanf("%d",&item);

    pos=binary(item,a,0,n-1);

    if(pos==-1)
        printf("\nItem not found");
    else
        printf("\nItem found at position %d",pos);
}
```

### 3.6 Tower of Hanoi Problem

In the problem of Tower of Hanoi, there will be three poles, viz. A, B and C. Pole A (source pole) contains 'n' discs of different diameters and are placed one above the other such that larger disc is placed below the smaller disc. Now, all the discs from source (A) must be transferred to destination (C) using the pole B as temporary storage.

Conditions are:

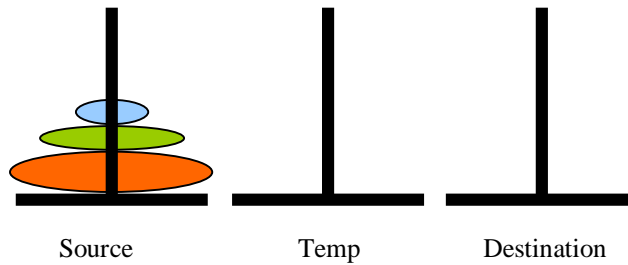
- Only one disc must be moved at a time
- smaller disc is on the top of larger disc at every step

Algorithm to move n discs from source to destination is as follows –

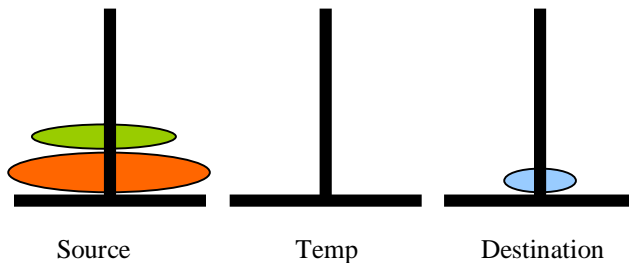
- move n-1 discs from source to temporary
- move n<sup>th</sup> disc from source to destination
- move n-1 disc from temporary to destination

For example, consider the number of discs = 3. Then, various steps involved in transferring 3 discs from source to destination are shown below –

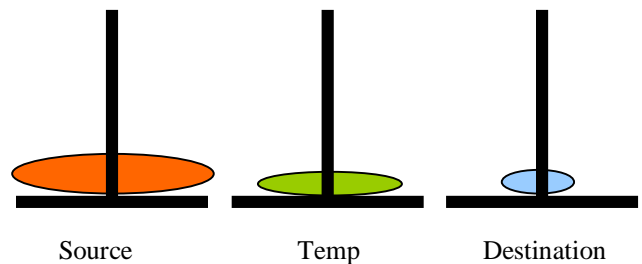
### Initial Stage



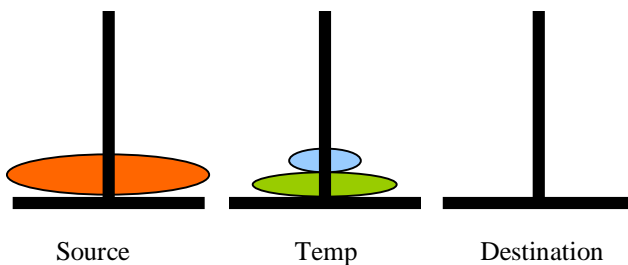
### Step 1:



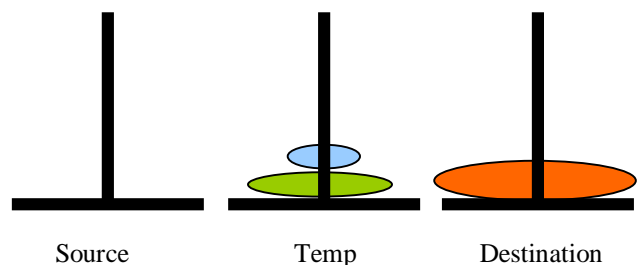
### Step 2:



### Step 3:



### Step 4:





```
    printf("Total number of moves =%d", count);  
}
```

**Output:** Enter the number of discs: 3  
Move disc 1 from A to C  
Move disc 2 from A to B  
Move disc 1 from C to B  
Move disc 3 from A to C  
Move disc 1 from B to A  
Move disc 2 from B to C  
Move disc 1 from A to C  
Total number of disc moves = 7

### 3.7 Properties of Recursive definition or algorithm

When a function calls itself, a new set of local variables and parameters are being allocated memory in the stack. Then the function code is executed from the top with these new variables. A recursive call does not make a new copy of the function, only the values being operated upon are new. When each recursive call returns, the old local variables and parameter are removed from the stack and the execution resumes at the point of the function call inside the function.

Some facts about recursive functions are:

- Recursive functions do not usually reduce the code size.
- They do not improve memory utilization compared to iterative functions.
- Many of the recursive functions may execute a bit slower compared to iterative functions because of the overhead of repeated function calls.
- They may cause a stack overrun, as each new call to the recursive function creates a new copy of variables and parameters and puts them into the stack.
- The advantage of using recursive functions is to create clearer and simpler programs.
- Moreover, some of the algorithms do require recursive functions as the iterative versions of them are quite difficult to write and implement.
- While writing a recursive function, a conditional statement has to be included which will terminate the recursive function without using a recursive function call.
- Otherwise, the recursive function will enter into an infinite loop and will not terminate at all.

Thus, any recursive function should satisfy the following conditions:

- In each and every call, the function must be nearer to the solution. (In other words, at every step, the problem size must reduce)
- There should be at least one non-recursive exit condition.

## UNIT 4. QUEUES

### 4.1 INTRODUCTION AND DEFINITION

In our day-to-day life, we come across many situations in which we have to stand in a queue like at a bank counter, at a cinema hall etc. In the field of computer science also we can give the examples for queue system such as printing a several files using only one printer, Processes waiting to be executing by CPU etc. Using this analogy, the data structure 'queue' is defined.

**Queue is a non-primitive linear data structure, where the elements are inserted at rear end and deleted from the front end.**

The item inserted first will be the first to be deleted. Hence it is known as **First In First Out (FIFO)** data structure.

The primitive operations of a queue structure include:

- Inserting an element into queue
- Deleting element from queue
- Displaying the contents of a queue

An attempt to insert an element into a full queue is called as **Queue overflow**. Trying to delete an element from an empty queue is known as **queue underflow**. The status of the queue is maintained by two variables viz. **front** and **rear**. Initially, both **front** and **rear** are set to -1 to indicate empty queue. While inserting the very first element into queue, both **front** and **rear** are incremented. After that, in every insertion, **rear** will be incremented. The **front** will be incremented at every deletion.

Consider the following illustration to understand the working of queue.

Empty Queue:



$f = r = -1$

Insert 14



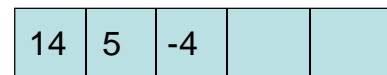
$f = r = 0$

Insert 5 :



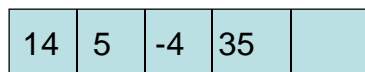
$f=0$   $r=1$

Insert -4:



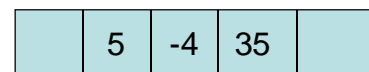
$f=0$   $r=2$

Insert 35:



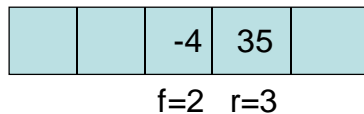
$f=0$   $r=3$

Delete :



$f=1$   $r=3$

Delete :



Note that, after every insertaion, **rear** is incremented and after every deletion, **front** is incremented.

### Program 4.1 Primitive operations on ordinary queue

```

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

#define MAX 3

void insert(int q[ ], int *f, int *r, int item)
{
    if(*r==MAX-1)
    {
        printf("\nQueue overflow");
        return;
    }
    q[++(*r)]=item;

    if(*f== -1)
        (*f)++;
}

void del(int q[], int *f, int *r)
{
    if(*f== -1 || *f>*r)
    {
        printf("\nQueue underflow");
        return;
    }
    printf("\nDeleted item is %d", q[( *f )++]);
}

void disp(int q[], int *f, int *r)
{
    int i;

    if(*f== -1 || *f>*r)
    {
        printf("\nNo elements to display!!");
    }
}

```



```
        return;
    }
    printf("\n Contents of queue:\n");

    for(i=*f;i<=*r;i++)
        printf("%d\t",q[i]);
}

void main()
{
    int q[MAX], f=-1, r=-1,item, opt;

    for(;;)
    {
        printf("\n****Queue operations****");
        printf("\n1.Insert\n 2.Delete\n 3.Display \n 4.Exit");
        printf("\nEnter your option: ");
        scanf("%d",&opt);

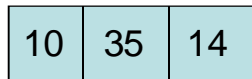
        switch(opt)
        {
            case 1: printf("\nEnter item to be inserted:");
                    scanf("%d",&item);
                    insert(q,&f,&r,item);
                    break;
            case 2: del(q, &f, &r);
                    break;
            case 3: disp(q, &f,&r);
                    break;
            case 4:
                    default:exit(0);
        }
    }
}
```

## 4.2 CIRCULAR QUEUE

We have seen in Program 4.1 that when an element is deleted from the queue, the **front** is incremented and when the element is inserted, **rear** is incremented. Such implementation of a queue has a drawback.

Assume, we have inserted the elements into queue up to its full capacity. That is, *rear* reached maximum size. Then, if we delete some elements from front end, there will be empty spaces at the beginning of a queue. But still, we can't insert elements into queue, as *rear* is already at *MAX*.

Let  $MAX=3$



$f=0$                        $r=2$

Delete twice:



$f=r=2$

Now, if we try to insert an item, it is not possible, as the condition ( $r==MAX-1$ ) is true. To overcome this problem, we adopt **circular queue**.

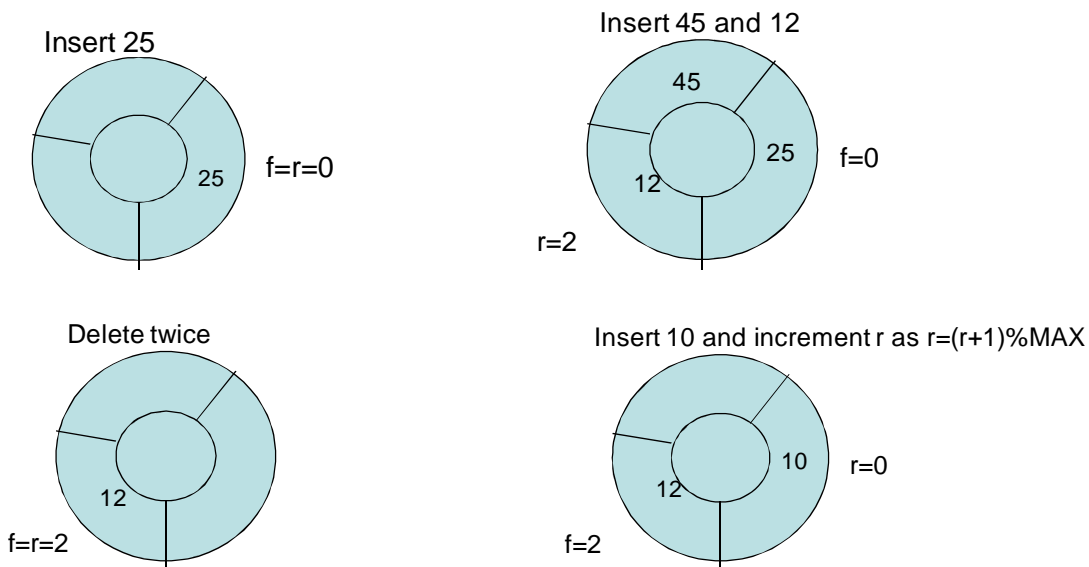
In ordinary queue, we will just increment **front** and **rear**. Hence, there is chance that **rear** will go beyond the range of the size of the queue. Hence, in circular queue, instead of using the statements like

$f = f + 1$                       and    $r = r + 1$ ,

we use

$f = (f + 1) \% MAX$                       and    $r = (r + 1) \% MAX$

This will ensure that both **front** and **rear** will fall within the range of  $MAX$  always. This can be illustrated as below –



Thus, we can overcome the problem with ordinary queue using circular queue.

### Program 4.2 Implementation of Circular Queue

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

#define MAX 3
```

```
void insert(int q[], int *f, int *r, int item)
{
    if(*f==( *r+1)%MAX)
    {
        printf("\nQueue overflow");
        return;
    }
    *r=( *r+1)%MAX;
    q[*r]=item;

    if(*f== -1)
        (*f)++;
}

void del(int q[], int *f, int *r)
{
    if(*f== -1 )
    {
        printf("\nQueue underflow");
        return;
    }

    printf("\nDeleted element is %d", q[*f]);

    if(*f==*r)
        *f=*r=-1;
    else
        *f=( *f+1)%MAX;
}

void disp(int q[], int *f, int *r)
{
    int i;

    if(*f== -1)
    {
        printf("\nNo elements to display!!");
        return;
    }
    printf("\n Contents of queue:\n");

    if(*f>*r)
    {
        for(i=*f;i<MAX;i++)
            printf("%d\t",q[i]);
    }
}
```

```
        for(i=0;i<=*r;i++)
            printf("%d\t", q[i]);
    }
    else
    {
        for(i=*f;i<=*r;i++)
            printf("%d\t",q[i]);
    }
}

void main()
{
    int q[10], f=-1, r=-1,item, opt;

    for(;;)
    {
        printf("\n*****Circular Queue operations*****");
        printf("\n1.Insert\n 2.Delete\n 3.Display \n 4.Exit");
        printf("\nEnter your option: ");
        scanf("%d",&opt);

        switch(opt)
        {
            case 1: printf("\nEnter item to be inserted:");
                    scanf("%d",&item);
                    insert(q,&f,&r,item);
                    break;
            case 2: del(q, &f, &r);
                    break;
            case 3: disp(q, &f,&r);
                    break;
            case 4:
            default:exit(0);
        }
    }
}
```

### 4.3 PRIORITY QUEUE

Priority Queue is a data structure in which the items are served (deleted) based on their priority levels. The insertion and deletion operations of priority queue are based on the priority of the elements. The element with highest priority is processed first and the element with second highest priority is processed next and so on. The use of this data structure is in job scheduling algorithms in the design of operating system.

These are two different types of priority queues viz.

- Ascending priority queue.

- Descending priority queue.

In both the methods the items are inserted in any order. But in ascending priority queue the smallest element is deleted first, while in the descending priority queue, the largest element is deleted first.

However, the term *smallest* in the above statement not necessarily mean the value of the element, but it may be any quantity associated with the element. For example, we can think of stack as a priority queue, in which the elements are deleted based on the time of insertion as a priority level. That is, the item inserted most recently (i.e. least time) will be deleted first. Similarly, ordinary queue can be thought of as a priority queue, where deletion is based on maximum time spent by the element in a queue. That is, the item which spent maximum time (inserted at the beginning of the process) will be deleted first.

One should note that, the meaning of deletion here (in the study of queue data structures, as well) indicates providing the service for which the item/element is waiting in a queue.

Priority queue can be implemented using arrays. In further discussion with priority queue, we assume a queue of integers where, the value of item itself indicates its priority. That is, smallest item has higher priority.

As discussed earlier, in priority queue, the insertion of elements can be in any order, but the deletion is based on priority. That is, in ascending priority queue, the smallest item must be searched for, and then it should be deleted. Hence, element present at any in between position of the array must be deleted and rest of elements must be re-adjusted. Thus, the deletion process becomes difficult.

So to avoid this problem and for the sake of simplicity, we assume that the elements are inserted in an ascending order, so that the deletion requires only deleting the element at front end. Thus, in the programmatic implementation of priority queue, during insertion, we should see that items are inserted in a proper position to maintain ascending order.

### Program 4.3 Implementation of Priority Queue

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#define MAX 3

void insert(int PQ[], int *f, int *r, int item)
{
    int j;

    if (*r==MAX-1)
    {
        printf("\nQueue overflow");
```

```
        return;
    }

    j=*r;

    while( j>=0 && item<PQ[ j ])
    {
        PQ[ j+1]=PQ[ j];
        j --;
    }

    PQ[ j+1]=item;
    (*r)++;

    if(*f== -1)
        (*f)++;
}

void del(int PQ[ ], int *f, int *r)
{
    if(*f== -1 || *f>*r)
    {
        printf("\nQueue underflow");
        return;
    }

    printf("\nDeleted item is %d", PQ[( *f )++]);
}

void disp(int PQ[], int *f, int *r)
{
    int i;

    if(*f== -1 || *f>*r)
    {
        printf("\nQueue underflow");
        return;
    }

    printf("\nContents of priority queue:");

    for(i=*f;i<=*r;i++)
        printf("%d\t", PQ[i]);
}
```

```
void main()
{
    int PQ[MAX], f=-1,r=-1, item, opt;

    for(;;)
    {
        printf("\n****Priority Queue****\n");
        printf("1.Insert\n2.Delete\n3.Display\n4.Exit\n");
        printf("\nEnter your option:");
        scanf("%d",&opt);

        switch(opt)
        {
            case 1: printf("\nEnter the item to be inserted:");
                    scanf("%d",&item);
                    insert(PQ,&f,&r,item);
                    break;
            case 2: del(PQ, &f, &r);
                    break;
            case 3: disp(PQ, &f, &r);
                    break;
            case 4:
            default:
                    exit(0);
        }
    }
}
```

#### 4.4 DOUBLE – ENDED QUEUE

A double ended queue or deque (pronounced as deck) is a set of items from which items may be deleted from either end and items may be inserted at either end. A deque can have some sub-types:

- **Input restricted deque:** deletion can be made at both the ends, but insertion can be made at only one end.
- **Output restricted deque:** deletion is at only one end and insertion can be made at both the ends.

Considering the definition of deque, the stack and queue data structures can be thought of as special cases of deque. Deque can be implemented either using arrays or linked lists. The implementation of deque using linked list will be discussed along with linked list chapter.

## UNIT 5. LINKED LISTS

### 5.1 INTRODUCTION

Array is the most popular and frequently used data structure. Though we have developed the data structures like stack and queue using array, it has certain limitations as discussed below.

The conventional array uses static memory allocation. For example, the declaration

```
int a[100];
```

will allocate the memory for 100 integers, say 200 bytes. During runtime of the program, neither the size of the array can be reduced, nor can it be increased. In case, the program uses only 10 integers out of 100, the space allocated for rest of 90 integers (180 bytes) will be wasted. On the other hand, if the program requires more integers, say 120, during run time, it can not allocate and it will face shortage of memory.

Thus, the static memory allocation for arrays may create the problem of either shortage of memory or wastage of memory. This problem can be avoided by using dynamic arrays. For example, we can consider dynamic memory allocation for array as –

```
int *p, n;  
p=(int *) malloc(sizeof(int)*n);
```

Now, memory for n integers, as requested by the user during execution time, will be allocated on heap. If the program requires more space, one can use *realloc()* function.

But, the problem still persists: array requires contiguous memory blocks and *malloc()* and *realloc()* may fail **if heap doesn't contain enough free space continuously !!**

To understand this problem, consider the following illustration:

```
int *p;  
p=(int *) malloc(sizeof(int)*100);
```

The above allocation requires 200 bytes of memory (if int requires 2 bytes) in a contiguous blocks. Now, assume that heap has total of 300 bytes free space, but with chunks of 50, 80, 90, 70 and 10 bytes at different locations. But, our request of 200 bytes cannot be served, as heap doesn't contain 200 bytes at a stretch.

So, *malloc()* or *realloc()* will return NULL and hence, the program cannot be continued further. Thus, in a nutshell, we can say that usage of arrays either with static memory allocation or with dynamic memory allocation does not serve for all practical applications.

To overcome these problems with arrays, a new data structure called as **linked list** has been designed. Linked list uses dynamic memory allocation and each element can be stored any where in the heap.

In linked list, we have different categories based on their structure and path of accessibility of elements:

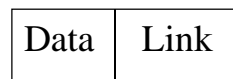


- Singly linked list
- Circular single linked list
- Doubly linked list
- Circular double linked list

Also, based on type of data that we store in the linked list, we categorize a list as homogeneous list and heterogeneous list. A list which contains single type of data (like int or char or float etc) is called as **homogeneous** list. Where as, a list with multiple types of data like combination of int, char, float etc. is called as **heterogeneous** list. In a list, each element is called as a **node**.

## 5.2 SINGLE (or SINGLY) LINKED LIST

In a singly linked list, every node consists of two parts viz. data field and link field as shown:



The data field consists of the item to be stored in the list. The link field of every node contains the address of next node in the list, and the link field of last node contains NULL to indicate end of the list. Thus, every time we need a new node, we request memory from heap.

The memory required for one node =  
memory required for the item + memory required for a pointer

The diagrammatic representation of singly linked list may look like as shown in Figure 5.1.

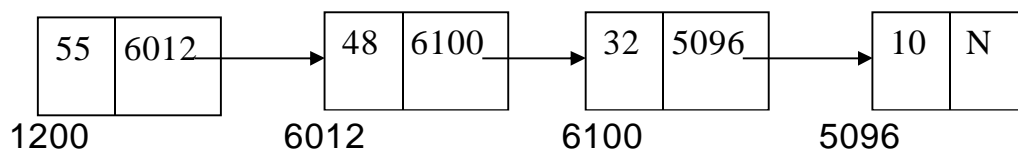


Figure 5.1 Representation of singly linked list

Various operations that can be performed on any linked list:

- Insert at front
- Insert at rear
- Delete from front
- Delete from rear
- Display the contents
- Insert at any position
- Delete from any position
- Search for a particular item
- Delete a particular item
- Creating ordered (ascending or descending) list

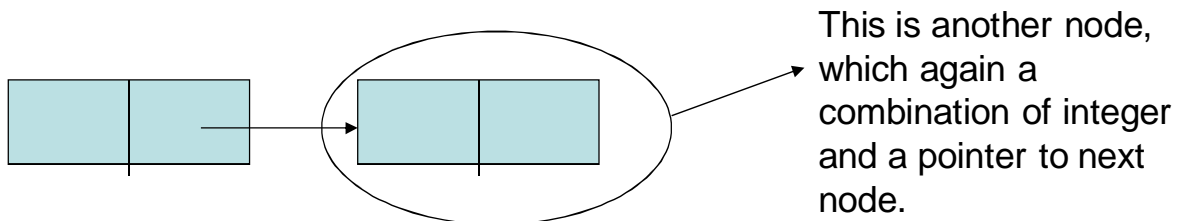
To perform various operations, first we should construct a linked list, or better to say a node. As discussed earlier, a node in singly linked list consists of a data part and link part.

For the initial stage of discussion, let us assume that we are going to create a linked list of integers. The one element (that is, node) in the list need to contain one integer and one **pointer to another node**.

Since we need two entities which are related to each other, but are of different types (integer and a pointer), we use a structure to design a node. That is, we can use something similar to the following:

```
struct node
{
    int data;
    _____ *link;
};
```

Here, we have to think, **what is the type of the pointer “link”?** The question is: the pointer *link* is going to store the address of what type of element?. The answer is: it is going to store the address of another node. Because, the linked list looks like this:



Thus, the structure looks like:

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

Now, refer Figure 5.1. It clearly indicates that, we are interested in the address of every node of linked list, which is the most important information needed to maintain any linked list.

So, for doing any operation, we need to create a pointer to **struct node**. For example,

```
struct node *n1;
struct node *n2; etc.
```

To reduce the typing job, we can use *typedef* as –

```
typedef struct node *NODE;
```

Now, whenever we need to create a node in linked list, we can just declare –

```
NODE n1;  
NODE n2;      etc.
```

We know that, each time we need a new node, we should request memory from the heap using *malloc()* function. To avoid repetitive task, we will write a function called **getnode()** to get heap memory and thus to create a new node.

```
NODE getnode()  
{  
    NODE x;  
    x=(NODE) malloc(sizeof(struct node));  
  
    if(x==NULL)  
    {  
        printf("no memory in heap");  
        exit(0);  
    }  
    return x;  
}
```

On successful allocation of memory, the above function returns the address of one block of memory, which is equivalent to the size of one node.

Similarly, to free a node, we can create a function like:

```
void freenode(NODE x)  
{  
    free(x);  
}
```

But, since the **freenode()** function contains only one statement, it makes no sense to call this function, instead, it is better to use a built-in function **free()**.

By referring to the Figure 5.1, it can be observed that, if we have the address of first node in our hand, we can trace the entire list. Thus, for doing all operations on singly linked list, we keep starting node as a reference, and we declare that node as –

**NODE start;**

Now, we will start implementing the code for various operations on singly linked list. Initially, we will implement 5 basic operations viz. insert\_front, insert\_rear, delete\_front, delete\_rear and display.

**Program 5.1 Implementation of Singly linked list**

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

struct node
{
    int data;
    struct node *link;
};
typedef struct node *NODE;

NODE getnode()
{
    NODE x;
    x=(NODE) malloc(sizeof(struct node));
    if(x==NULL)
    {
        printf("no memory in heap");
        exit(0);
    }
    return x;
}

NODE insert_front(int item, NODE start)
{
    NODE temp;
    temp = getnode();
    temp->data=item;
    temp->link=start;
    return temp;
}

NODE delete_front(NODE start)
{
    NODE temp;

    if(start==NULL)
    {
        printf("no element to delete\n");
        return start;
    }

    temp=start;
```

```
    printf("Deleted item=%d", temp->data);
    start=start->link;
    free(temp);
    return start;
}
```

```
NODE insert_rear(int item, NODE start)
{
```

```
    NODE temp, cur;
    temp=getnode();
    temp->data=item;
    temp->link=NULL;
```

```
    if (start==NULL)
        return temp;
```

```
    cur=start;
    while(cur->link!=NULL)
        cur=cur->link;
```

```
    cur->link=temp;
    return start;
```

```
}
```

```
void display(NODE start)
```

```
{
    NODE temp;
    if(start==NULL)
    {
        printf("No element to display\n");
        return ;
    }
```

```
    printf("The contents of list:\n");
```

```
    temp=start;
```

```
    while(temp!=NULL)
    {
        printf("%d\n", temp->data);
        temp=temp->link;
    }
```

```
}
```

```
NODE delete_rear(NODE start)
```

```
{
    NODE prev, cur;
```

```
    if(start==NULL)
    {
        printf("no element to delete\n");
        return start;
    }

    if(start->link==NULL)
    {
        printf("\nDeleted element is%d", start->data);
        free(start);
        return NULL;
    }
    prev=NULL;
    cur=start;

    while(cur->link!=NULL)
    {
        prev=cur;
        cur=cur->link;
    }
    printf("\nDeleted element is %d", cur->data);
    free(cur);
    prev->link=NULL;
    return start;
}

void main()
{
    int opt, item;
    NODE start=NULL;

    for(;;)
    {
        printf("1.Insert Front\n 2.Insert Rear\n 3. Display\n");
        printf(" 4.Delete Front\n 5.Delete Rear\n");
        printf("enter your option:");
        scanf("%d",&opt);

        switch(opt)
        {
            case 1: printf("\nenter item");
                    scanf("%d",&item);
                    start=insert_front(item,start);
                    break;
            case 2: printf("\nenter item");
```

```

        scanf("%d",&item);
        start=insert_rear(item,start);
        break;
    case 3: display(start);
        break;
    case 4: start=delete_front(start);
        break;
    case 5: start=delete_rear(start);
        break;
    default: exit(0);
}
}
}

```

### 5.2.1 Creating Ordered Linked List

A linked list in which all the items are stored in some specified order viz. ascending or descending is known as an ordered linked list. Note that after the insertion of one item into an ordered list, the order should be maintained. Thus, the insertion process must be done as explained below:

- If the item to be inserted is less than the first item of the existing list, then this 'new node' should become the 'start' of resulting list.
- If the item to be inserted is greater than the last item of the existing list, then 'new node' must be the last node of the resulting list.
- But if the item is somewhere between the linked list then it must be inserted at appropriate position using the 'link' field of the nodes.

**NOTE:** Instead of insert\_front() and insert\_rear() functions in the linked list program, if you use the following function i.e. insert\_order(), then you will get an ordered linked list.

#### Code Snippet for creating ordered linked list:

```

NODE insert_order(int item, NODE start)
{
    NODE temp, prev, cur;
    temp=getnode();
    temp->data = item;
    temp->link = NULL;

    if(start == NULL)
        return temp;

    if(item < start ->data)
    {
        temp->link =start;
        return temp;
    }
}

```

```
    prev = NULL;
    cur = start;
    while(cur != NULL && item >= cur->data)
    {
        prev = cur;
        cur = cur->Link;
    }

    prev->link = temp;
    temp->link=cur;
    return start;
}
```

### 5.2.2 Stack and Queue using Linked lists

We have discussed earlier that stack is LIFO data structure and queue is a FIFO data structure. And, they can be implemented using either arrays or linked lists. Array implementation of these data structures have been discussed in the previous chapters. The linked-list representation can be done as give below:

- **Stack:** Implement *insert\_front()*, *delete\_front()* and *display()* functions using singly linked list. That is, insertion and deletion from the same end is nothing but stack.
- **Queue:** Implement *insert\_rear()*, *delete\_front()* and *display()* functions.
- **Priority Queue:** Implement *insert\_order()*, *delete\_front()* and *display()* functions.
- **Deque:**
  - **General Deque:** Implement *insert\_front()*, *insert\_rear()*, *delete\_front()*, *delete\_rear()* and *display()* functions.
  - **Input Restricted Deque:** Implement *insert\_rear()*, *delete\_front()*, *delete\_rear()* and *display()* functions.
  - **Output Restricted Deque:** Implement *insert\_rear()*, *insert\_front()*, *delete\_front()* and *display()* functions.

### 5.2.3 Searching for a particular node

This operation is used to find whether a given key item is present in the list or not. If the item is there in the list, it is known as successful search, otherwise an unsuccessful search. A function to search for an item in the existing list is given below. Use appropriate *main()* function and *insert()* functions and then use 'search' function for a working of full program.

#### Code snippet for searching a particular node:

```
void search( int key, NODE start)
{
    NODE cur;
    int pos;

    if (start == NULL)
```



```
{
    printf("List is empty");
    return;
}
cur = start;
pos = 1;

while(cur!=NULL && key!=cur->data)
{
    cur = cur->link;
    pos++;
}
if(cur==NULL)
{
    printf("key not found");
    return;
}
printf("Key is found at position %d", pos);
}
```

#### 5.2.4 Deletion of a node whose data field is given

Sometimes, we may need to delete a particular node from the list, based on its value. That is, when we are given a 'data' field of some node, then the entire list must be traversed to find that node, and then it must be deleted.

Following function is for deleting a particular node. Use appropriate *main()* function and *insert()* function for the complete implementation.

```
NODE delete_data(int item, NODE start)
{
    NODE prev, cur;

    if(start == NULL)
    {
        printf("No item to delete");
        return start;
    }

    if(item == start->data)
    {
        cur = start;
        start=start->link;
        free (cur);
        return start;
    }
}
```

```
prev = NULL;
cur = start;

while(cur!=NULL && item !=cur->data)
{
    prev = cur;
    cur= cur->link;
}

if(cur == NULL)
{
    printf("Item not found);
    return start;
}

prev->link = cur->link;
free(cur);

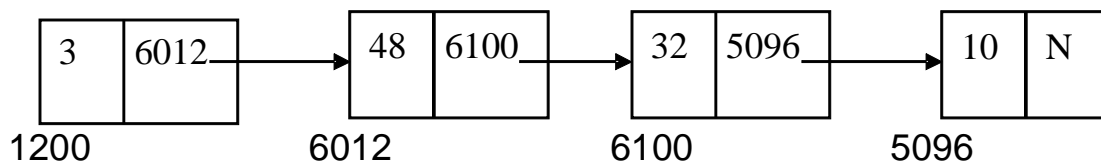
return start;
}
```

### 5.2.5 Header Nodes

To simplify the design of linked list, sometimes we will have a special node at the beginning of the list. Usually, the 'data' field of this node would be empty, and it will not represent any item of the linked list. Such a node is called as **header node**.

Sometimes the integer value representing the number of nodes present in the list will be stored in header node. But, in this case each time an insertion or deletion occurs, the 'data' field of header node must be updated to keep track of the actual information. If the list is empty, then link field of header node contains NULL or else it will contains the address of first node of the linked list.

For example, following diagram shows a singly linked list with a header node, where the data field of header node contains total number of nodes in the list.



Here, the first node with the address 1200 is a header node. Its data field contains the number 3 indicating there are 3 nodes in the list.

### 5.2.6 Non-homogenous List

All the examples that we discussed till now were having the 'data' field of a node as an integer variable. But, in fact, a 'data' field of a node in a singly linked list may also contain the variables of some other data type. Moreover, a node can contain more than one value in its data field. That is, it is capable of storing various information.

For example, we can define our structure as-

```
struct student
{
    char name[20];
    int stdid;
    int sem;
    struct student *link;
};
```

Then, each of the member field can be stored in 'data' field of a linked list. That is, one node of the linked list may look like –

name	stdid	sem	link
------	-------	-----	------

Suppose we have declared a variable like-

```
struct student *temp;
```

Then, the member variables can be accessed as

```
temp -> name;
temp -> stdid;
temp -> sem;      etc.
```

The operations for non-homogenous or non-integer linked list are same as those of integer linked list. But the functions like *insert-front()* and *insert-rear()* should have the parameters like char, int etc. instead of only one parameter.

Consider following example program to illustrate non-homogenous list.

### **Program 5.2 Illustration of non-homogenous (or heterogenous) list**

**Question:** Write a C program to construct a singly linked list consisting of the following information in each node: student\_id (integer), student\_name (character string) and semester (integer). Perform insert front and display operations on this non-homogenous linked list

**Answer:**

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

```
struct node
{
    int sid;
    char sname[15];
    int sem;
    struct node *link;
};
typedef struct node *NODE;

NODE getnode()
{
    NODE x;
    x = (NODE) malloc (sizeof (struct node));
    if (x == NULL)
    {
        printf("No memory space\n");
        exit(0);
    }
    return x;
}

NODE insert_front (int stuid, char stuname[], int semester, NODE start)
{
    NODE temp;
    temp = getnode();
    temp->sid = stuid;
    strcpy(temp->sname, stuname);
    temp->sem = semester;
    temp->link = start;

    return temp;
}

void display (NODE start)           //Display the contents of the list.
{
    NODE temp;

    if (start == NULL)
    {
        printf("List is empty\n");
        return;
    }

    printf("\nThe contents of the list are :\n");
    temp = start;
```

```
printf("S-ID \t\t\t S-NAME \t\t\t SEMESTER\n");

while (temp != NULL)
{
    printf("%d \t %s \t \t %d\n", temp->sid, temp->sname, temp->sem);
    temp = temp->link;
}

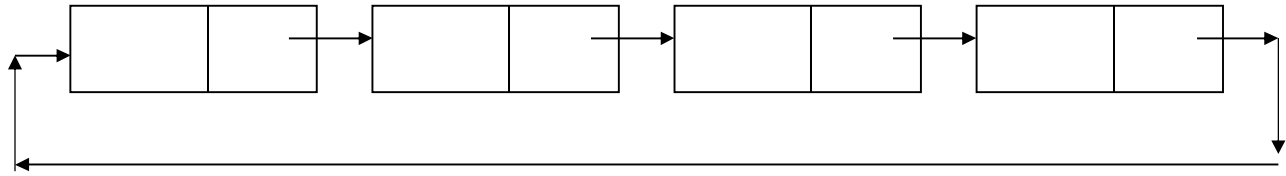
void main()
{
    NODE start = NULL;
    int opt, stuid, semester;
    char stuname[15];

    for(;;)
    {
        printf("1: Insert \n 2: Display \n 3: EXIT\n");
        printf("Enter your option\n");
        scanf("%d", &opt);

        switch (opt)
        {
            case 1: printf("Enter the student id \n");
                    scanf("%d", &stuid);
                    printf("Enter the student name \n");
                    scanf("%s", stuname);
                    printf("Enter the semester\n");
                    scanf("%d", &semester);
                    start = insert_front(stuid, stuname, semester, start);
                    break;
            case 2: display(start); break;
            case 3:
            default: exit(0);
        }
    }
}
```

### 5.3 CIRCULAR LINKED LISTS

Circular singly linked list is quite similar to singly linked list. The only difference is – the 'link' field of a last node in a circular singly linked list contains the address of first node, instead of NULL. The diagrammatic representations is -



In a singly linked list, we can trace the list in one direction. That is, if we are at 10<sup>th</sup> node, we can't trace back to access 9<sup>th</sup> node, instead, we have trace from the beginning once again. This is time consuming. Hence, we make use of circular list to avoid this problem up to some extent.

Because, in a circular list, from any node we can trace rest of the nodes processing in a forward direction (refer diagram given in previous slide). Theoretically, any node in a circular list can be treated as a first node, and its previous node as a last node. But, since, there won't be NULL in any node to indicate end of list, the circular lists have to be processed properly, otherwise, it may lead to infinite loop.

In most of the real time applications, we will be interested in first node and the last node of a linked list. So, if we keep the first node in our hand, we have to trace the whole list to get a last node, as we did in singly linked list. So, in a circular list, **we will keep last node as a tool in our hand**, whose next node itself will be the first node, and hence, we can get both the nodes without much trace.

All the operations performed on an ordinary singly linked list can be implemented on circular singly linked list.

### Program 5.3 Operations on Circular linked lists

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

struct node
{
    int data;
    struct node *link;
};
typedef struct node *NODE;

NODE getnode()
{
    NODE x;
    x=(NODE) malloc(sizeof(struct node));
    if(x==NULL)
    {
        printf("no memory in heap");
    }
}
```

```
        exit(0);
    }
    return x;
}

NODE insert_front(int item, NODE last)
{
    NODE temp;
    temp = getnode();

    temp->data=item;
    temp->link=temp;

    if(last==NULL)
        return temp;

    temp ->link = last->link;
    last ->link=temp;
    return last;
}

NODE insert_rear(int item, NODE last)
{
    NODE temp;
    temp=getnode();

    temp->data=item;
    temp->link=temp;

    if (last==NULL)
        return temp;

    temp->link=last->link;
    last->link=temp;
    return temp;
}

void display(NODE last)
{
    NODE temp;

    if(last==NULL)
    {
        printf("No element to display\n");
        return ;
    }
}
```

```
temp=last->link;
printf("The contents of list:\n");

while(temp!=last)
{
    printf("%d\n" temp->data);
    temp=temp->link;
}

printf("%d", temp->data);
}

NODE delete_front(NODE last)
{
    NODE temp;
    if(last==NULL)
    {
        printf("no element to delete\n");
        return NULL;
    }
    if(last->link==last)
    {
        printf("The item deleted is %d", last->data);
        free(last);
        return NULL;
    }
    temp=last->link;
    last->link=temp->link;
    printf("Item deleted is %d", temp->data);
    free(temp);
    return last;
}

NODE delete_rear(NODE last)
{
    NODE prev;
    if(last==NULL)
    {
        printf("no element to delete\n");
        return NULL;
    }
    if(last->link==last)
    {
        printf("The item deleted is %d", last->data);
        free(last);
    }
}
```



```
        return NULL;
    }
    prev=last->link;
    while(prev->link!=last)
        prev=prev->link;

    prev->link=last->link;
    printf("\nDeleted element is %d", last->data);
    free(last);
    return prev;
}

void main()
{
    int opt, item;
    NODE last=NULL;

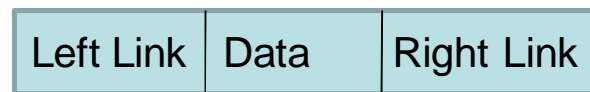
    for(;;)
    {
        printf("1.Insert Front\n    2.Insert Rear\n 3.Display\n")
        printf(" 4.Delete Front\n    5.Delete Rear\n");
        printf("Enter your option:");
        scanf("%d",&opt);

        switch(opt)
        {
            case 1: printf("\nenter item");
                    scanf("%d",&item);
                    last=insert_front(item,last);
                    break;
            case 2: printf("\nenter item");
                    scanf("%d",&item);
                    last=insert_rear(item,last);
                    break;
            case 3: display(last);
                    break;
            case 4: last=delete_front(last);
                    break;
            case 5: last=delete_rear(last);
                    break;
            default: exit(0);
        }
    }
}
```

## 5.4 DOUBLE LINKED LISTS

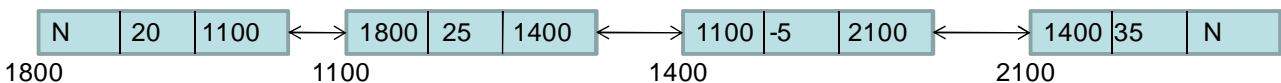
In a singly linked list and a circular singly linked list, we can traverse a list in a forward direction. That is, when we are at  $n^{\text{th}}$  node and would like to access  $(n-1)^{\text{th}}$  node, then we have to traverse in a forward direction only. This is time consuming.

To avoid such problem, we will go for double linked list in which each node consists of two link fields (viz. left link and right link) and one data field as shown:

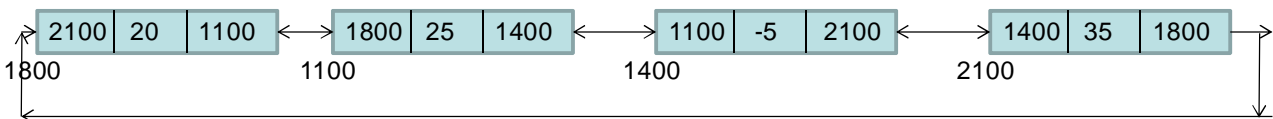


Here, right link is used to store the address of next node and left link is used to store the address of previous node. In ordinary doubly linked list, the left link field of first node and the right link field of last node contain NULL.

A typical doubly linked list may look like this –



A circular doubly linked list may look like this –



**NOTE:** All the primitive operations done using singly linked list can be applied for doubly linked list and circular doubly linked list.

The structure declaration for a node will be –

```
struct node
{
    struct node *llink;
    int data;
    struct node *rlink;
};
```

## UNIT 6. SORTING

### 6.1 INTRODUCTION

Sorting is a process of arranging a set of data in some order. Usually, sorting will be either in ascending order or in descending order. Sorting technique can be mainly divided into two categories viz. internal sorting and external sorting. If all the data to be sorted all stored in the main memory, then it is called as internal sorting. If the data are stored in the auxiliary storage i.e. in floppy, tape etc, then the sorting is said to be external sorting. Let us discuss about different internal sorting techniques one by one.

### 6.2 SELECTION SORT

Selection sort is a simplest method of sorting technique. To sort the given list in ascending order, we will compare the first element with all the other elements. If the first element is found to be greater than the compared element, then they are interchanged. Thus at the end of first interaction, the smallest element will be stored in first position, which is its proper position. Then in the second interaction, we will repeat the procedure from second element to last element. The algorithm is continued till we get sorted list. If there are  $n$  elements, we require  $(n-1)$  iterations, in general.

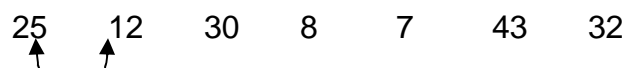
Consider the example----

25    12    30    8    7    43    32

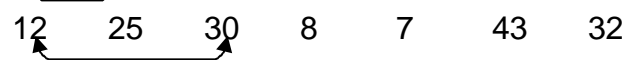


First: Iteration

25    12    30    8    7    43    32



12    25    30    8    7    43    32




12    25    30    8    7    43    32



8    25    30    12    7    43    32



7    25    30    12    8    43    32

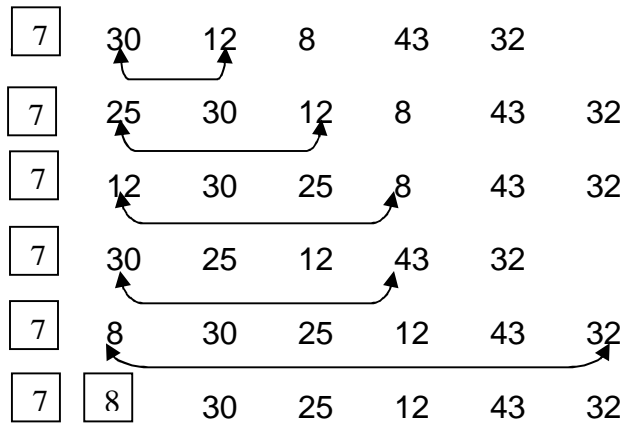


7    25    30    12    8    43    32

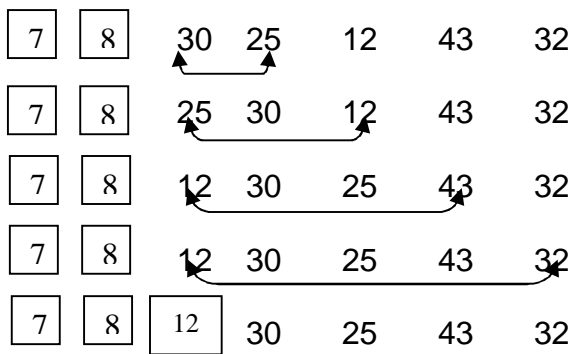


7    25    30    12    8    43    32

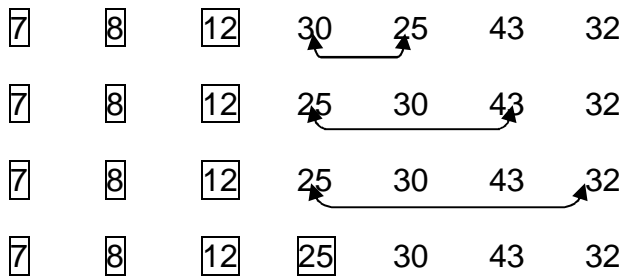
**Second Iteration**



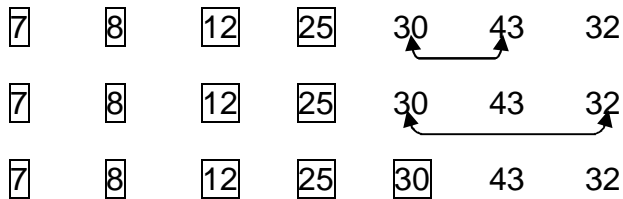
**Third Iteration**



**Fourth Iteration**



**Fifth Iteration**



**Sixth Iteration**

7	8	12	25	30	43	32
7	8	12	25	30	32	43

↖
↗

Thus sorted list is:

7, 8, 12, 25, 30, 32, 43

**Program:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10],n,i,temp,j;
    clrscr();
    printf("Enter the size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    for(i=0;i<n-1;i++)
    {
        for(j=i+1;j<n;j++)
        {
            if(a[i]>a[j])
            {
                temp=a[i];
                a[i]=a[j];
                a[j]=temp;
            }
        }
    }
    printf("\nSorted list is:\n");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    getch();
}
```

**6.3 BUBBLE SORT**

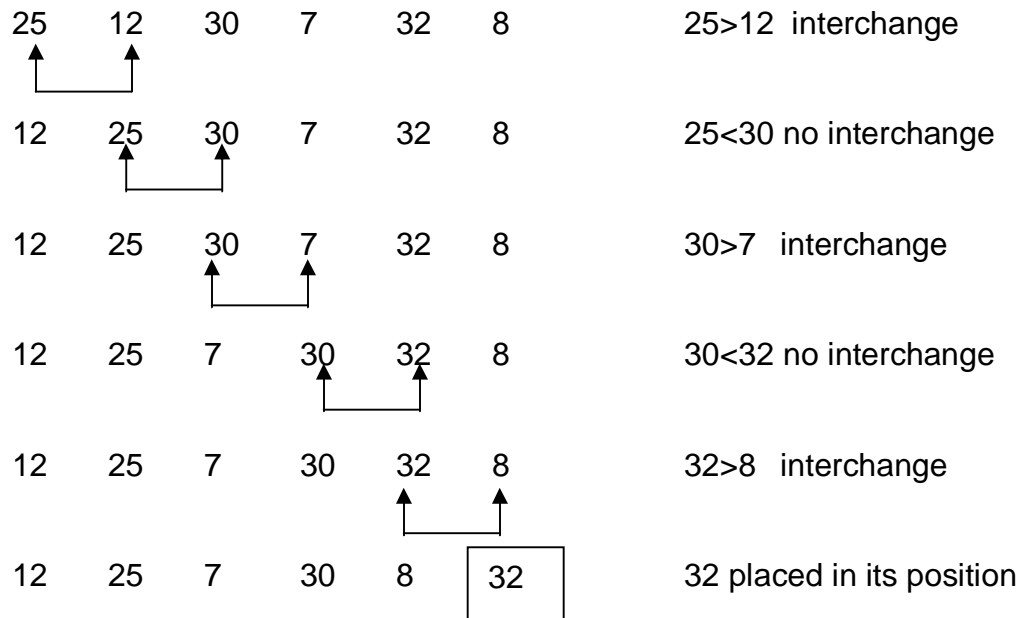
The bubble sort technique for sorting a list of data in ascending order is as follows: In the first iteration, the first element of the array is compared with the second element. If the first element is found to be greater than the second element, they are interchanged. Now, the

second element is compared with the third and interchanged if required. In the same way, comparison is done till the last element. At the end of first iteration, the largest element will be stored at the last position. In the second iteration, again the comparison is done from the first element to last-but-one element. At the end of this iteration, the second largest element will be placed in its proper position. If there are 'n' elements in the given list, then after (n-1) iterations, the array gets sorted.

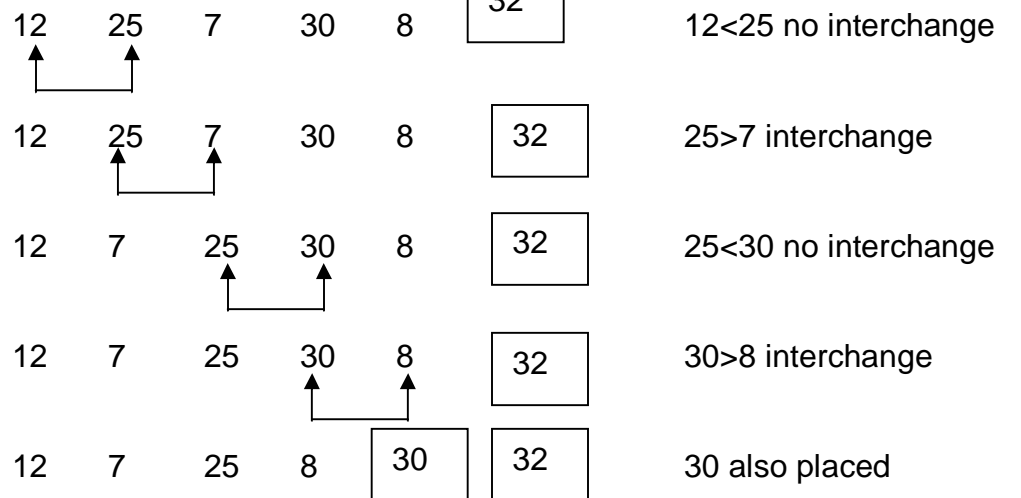
Consider the following list of integers to be sorted:

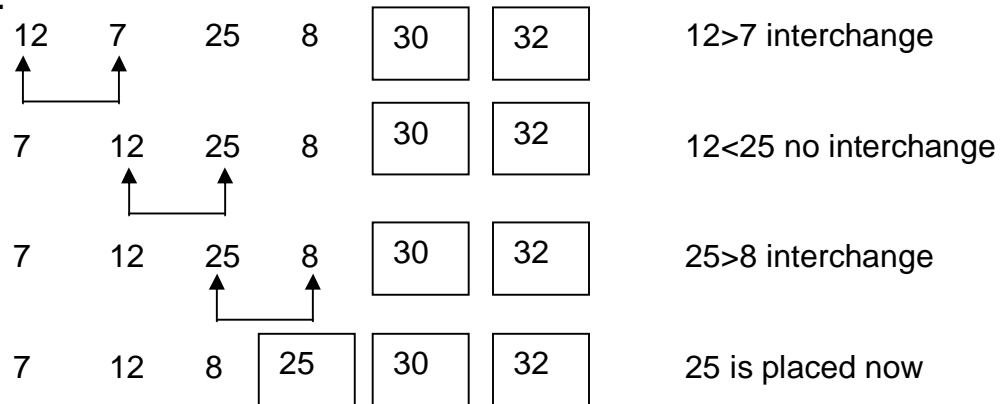
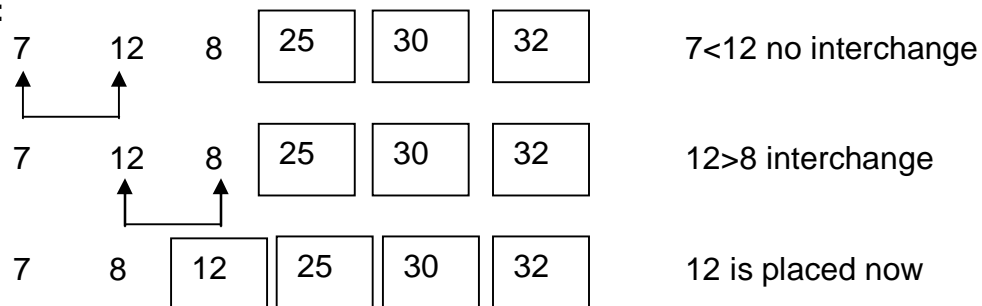
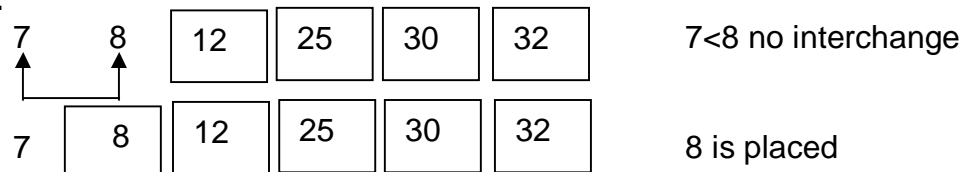
25 12 30 7 32 8

**1<sup>st</sup> Iteration:**



**2<sup>nd</sup> Iteration:**



**3<sup>rd</sup> Iteration:****4<sup>th</sup> Iteration:****5<sup>th</sup> Iteration:**

Thus, the sorted list is:

7   8   12   25   30   32

**Program:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10],n,i,temp,j;
    clrscr();
    printf("Enter the size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
```

```

for(i=0;i<n;i++)
{
    for(j=0;j<n-i-1;j++)
    {
        if(a[j]>a[j+1])
        {
            temp=a[j];
            a[j]=a[j+1];
            a[j+1]=temp;
        }
    }
}
printf("\nSorted list is:\n");
for(i=0;i<n;i++)
    printf("%d\t",a[i]);
getch();
}

```

## 6.4 INSERTION SORT

This sorting technique involves inserting a particular element in proper position. In the first iteration, the second element is compared with the first. In second iteration, the third element is compared with second and then the first. Thus in every iteration, the element is compared with all the elements before it. If the element is found to be greater than any of its previous elements, then it is inserted at that position and all other elements are moved to one position towards right, to create the space for inserting element. The procedure is repeated till we get the sorted list.

### Consider an example

25   12   30   8   7   43   32

#### I iteration


 25   12   30   8   7   43   32   (12<25, so insert 12 at first position)

12   25   30   8   7   43   32

#### II iteration


 12   25   30   8   7   43   32   (30>25, so don't compare 30 with 12)

12   25   30   8   7   43   32



**III iteration**

12 25 30 8 7 43 32 (8<30,25,12 so insert 8 at 1<sup>st</sup> position)

**IV iteration**

8 12 25 30 7 43 32 (7<30,25,12,8. So insert 7 at 1<sup>st</sup> pos)

**V iteration**

7 8 12 25 30 43 32 (43>30. So, don't compare 43 with other)

**VI iteration**

7 8 12 25 30 43 32 (32<43 but 32>30. So, insert in-between)

Sorted list:

7 8 12 25 30 32 43

**Program:**

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10],n,i,item,j;
    clrscr();
    printf("Enter the size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    for(i=1;i<n;i++)
    {
        item=a[i];
        for(j=i-1;j>=0 && item<a[j];j--)
            a[j+1]=a[j];
        a[j+1]=item;
    }
}
```

```
printf("\nSorted list is:\n");
for(i=0;i<n;i++)
    printf("%d\t",a[i]);
getch();
}
```

## 6.5 QUICK SORT

As the name suggests, Quick Sort is a technique that will sort a list of data significantly faster than any other sorting techniques. This algorithm is based on the fact that – it is always easier and faster to sort two small arrays than one single big array.

Here, the given array is divided into two sub-arrays such that the elements at the left-side of some key element are less than the key element and the elements at the right-side of the key element are greater than the key element.

The dividing procedure is done with the help of two index variables and one key element as explained below –

- i) Usually the first element of the array is treated as *key*. The position of the second element is taken as the first index variable *left* and the position of the last element will be the index variable *right*.
- ii) Now the index variable *left* is incremented by one till the value stored at the position *left* is greater than the *key*.
- iii) Similarly *right* is decremented by one till the value stored at the position *right* is smaller than the *key*.
- iv) Now, these two elements are interchanged. Again from the current position, *left* and *right* are incremented and decremented respectively and exchanges are made appropriately, if required.
- v) This process is continued till the index variables crossover. Now, exchange *key* with the element at the position *right*.
- vi) Now, the whole array is divided into two parts such that one part is containing the elements less than the *key* element and the other part is containing the elements greater than the *key*. And, the position of *key* is fixed now.
- vii) The above procedure (from step i to step vi) is applied on both the sub-arrays. After some iteration we will end-up with sub-arrays containing single element. By that time, the array will be sorted.

Let us illustrate this algorithm using an example. Consider an array

$$a[7] = \{25, 12, 30, 8, 7, 43, 32\}$$

Let key = 25

left = 1, the position of 12

right = 6, the position of 32

**First step:** Compare *key* with *a[left]*

<i>key</i>	<i>left</i>					<i>right</i>
25	12	30	8	7	43	32

Now,  $key > a[left]$  (i.e.  $25 > 12$ ) is true.

So increment *left*. So, now *left* will be at the element 30.

**Second step:** Compare *key* with *a[left]*

<i>key</i>		<i>left</i>				<i>right</i>
25		12	30	8	7	43
						32

Now,  $key > a[left]$  (i.e.  $25 > 30$ ) is false.

So stop incrementing *left*.

**Third step:** Compare *key* with *a[right]*.

<i>key</i>		<i>left</i>				<i>right</i>
25		12	30	8	7	43
						32

Now,  $key < a[right]$  (i.e.  $25 < 32$ ) is true.

So, decrement *right*. Thus, *right* will be at the element 43 now.

**Fourth step:** Compare *key* with *a[right]*.

<i>key</i>		<i>left</i>			<i>right</i>	
25		12	30	8	7	43
						32

Now,  $key < a[right]$  (i.e.  $25 < 43$ ) is true.

So, decrement *right*. Thus, *right* will be now at 7.

**Fifth step:** Compare *key* with *a[right]*.

<i>key</i>		<i>left</i>		<i>right</i>		
25		12	30	8	7	43
						32

Now,  $key < a[right]$  (i.e.  $25 < 7$ ) is false.

So, stop decrementing *right*.

**Sixth step:** Exchange the values of *a[left]* (30) and *a[right]* (7). Thus, array will be –

<i>key</i>		<i>left</i>		<i>right</i>		
25		12	7	8	30	43
						32

**Seventh step:** Again start the procedure from beginning. That is, compare *key* with *a[left]*.

<i>key</i>		<i>left</i>		<i>right</i>		
25		12	7	8	30	43
						32

Now,  $key > a[left]$  (i.e.  $25 > 7$ ) is true.

So, increment *left*. Now, *left* will be at 8.

**Eighth step:** Compare *key* with *a[left]*.

<i>key</i>			<i>left</i>	<i>right</i>		
25		12	7	8	30	43
						32

Now,  $key > a[left]$  (i.e.  $25 > 8$ ) is true.  
So, increment *left*. Now, *left* will be at 30.

**Ninth step:** Compare *key* with  $a[left]$ .

<i>key</i>				<i>left, right</i>		
25	12	7	8	30	43	32

Now,  $key > a[left]$  (i.e.  $25 > 30$ ) is false.  
So, stop incrementing *left*.

**Tenth step:** Compare *key* with  $a[right]$ .

<i>key</i>				<i>left, right</i>		
25	12	7	8	30	43	32

Now,  $key < a[right]$  (i.e.  $25 < 30$ ) is true.  
So, decrement *right*. Thus, *right* will be at 8 now.

**Eleventh step:** The array looks like –

<i>key</i>			<i>right</i>	<i>left</i>		
25	12	7	8	30	43	32

As the index variables *left* and *right* cross-over, exchange *key* (25) with  $a[right]$  (8).

The array would be –

8	12	7	25	30	43	32
---	----	---	----	----	----	----

Thus, all the elements at the left-side of *key* (i.e. 25) are less than *key* and all the elements at the right-side of *key* are greater than *key*. Hence, we have got two sub-arrays as –

{8, 12, 7}    25    {30, 43, 32}

Now, the position of 25 will not get changed. But, we have to sort two sub-arrays separately, by referring the above explained steps.

Proceeding like this, we will get the sorted list.

**Program:**

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

```
quick_sort(int x[], int low, int high)        //Function to apply quick sort technique
{
    int pos;
    if (low < high)
    {
        pos = partition(x,low,high);
        quick_sort(x,low,pos-1);
        quick_sort(x,pos+1,high);
    }
    return;
}
```

```
int partition(int x[], int low, int high)           //Function for partitioning the array
{
    int key, temp, true = 1;
    int left, right;

    key = x[low];
    left = low +1;
    right = high;

    while(true)
    {
        while ((left < high) && (key >= x[left]))
            left++;
        while(key < x[right])
            right--;
        if(left < right)
        {
            temp = x[left];
            x[left] = x[right];
            x[right] = temp;
        }
        else
        {
            temp = x[low];
            x[low] = x[right];
            x[right] = temp;
            return(right);
        }
    }
    return 0;
}

void main()
{
    int a[10],n,i,low,high;
    clrscr();

    printf("Enter array size\n");
    scanf("%d",&n);

    printf("Enter the elements\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    low = 0;
```

```

high = n-1;

quick_sort(a,low,high);

printf("The sorted list is \n");
for(i=0;i<n;i++)
    printf("%d\t",a[i]);

getch();
}

```

**NOTE:** The bubble sort and quick sort techniques are usually called as exchange sort techniques. Because, both of these involves the procedure of exchanging the elements in some or the other situation.

## 6.6 SHELL SORT

This sorting technique is almost similar to that of insertion sort. Instead of comparing adjacent element as in insertion sort, here we will compare the element at equal distance.

In this sorting technique, first we will divide the array into sub-arrays by taking the elements at equal distances. Usually the distance will be  $n/2$ , where  $n$  is the size of the array. Now, these sub arrays are sorted using insertion sort. Again in the next step we will reduce the distance to the half of the previous distance and create sub-arrays. Again, these arrays are sorted. By continuing this way, at the end, are sub-arrays will be of single element and then after sorting, the entire array will get sorted.

**NOTE** that, if the array size is small, we can take the initial distance  $d=n/2$  and later  $d=d-1$  also.

### Consider an example

25	12	30	8	7	43	32
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Here,  $n=7$ . Let  $d = 3$

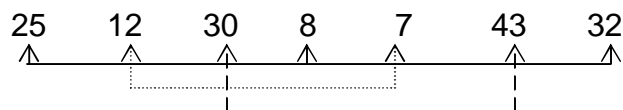
Now, sub-arrays will be

a[0], a[3], a[6]

a[1], a[4]

a[2], a[5].

Tracing of algorithm is a follows



Now, three sub-arrays are:

25, 8, 32

12, 7

30, 43

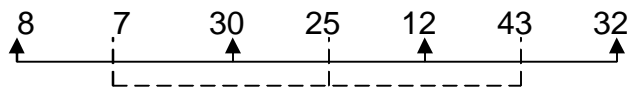
Sorting these arrays using insertion sort, we will get

8, 25, 32

7, 12

30, 43

Now take  $d = 2$



Now, the sub arrays are:

8, 30, 12, 32

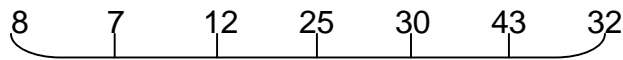
7, 25, 43

Sort these arrays using insertion sort. We get –

8, 12, 30, 32

7, 25, 43

Now take  $d = 1$



Now, every element is at the distance 1, sort this array using insertion sort. We will get –

7 8 12 25 30 32 43

### Program:

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

```
#include<conio.h>
```

```
void ShellSort(int n, int a[10])
```

```
{
```

```
    int i,j,d,item, k;
```

```
    for(d=(n-1)/2;d>0;d=d/2)
```

```
    {
```

```
        for(k=0;k<d;k++)
```

```
        {
```

```
            for(i=0;i<n;i=i+d)
```

```
            {
```

```
                item=a[i];
```

```
                j=i-d;
```

```

        while(j>=0 && item<a[j])
        {
            a[j+d]=a[j];
            j=j-d;
        }
        a[j+d]=item;
    }
}

void main()
{
    int a[10],n,i;
    clrscr();
    printf("Enter the size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    ShellSort(n,a);

    printf("\nSorted list is:\n");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    getch();
}

```

## 6.7 MERGE SORT

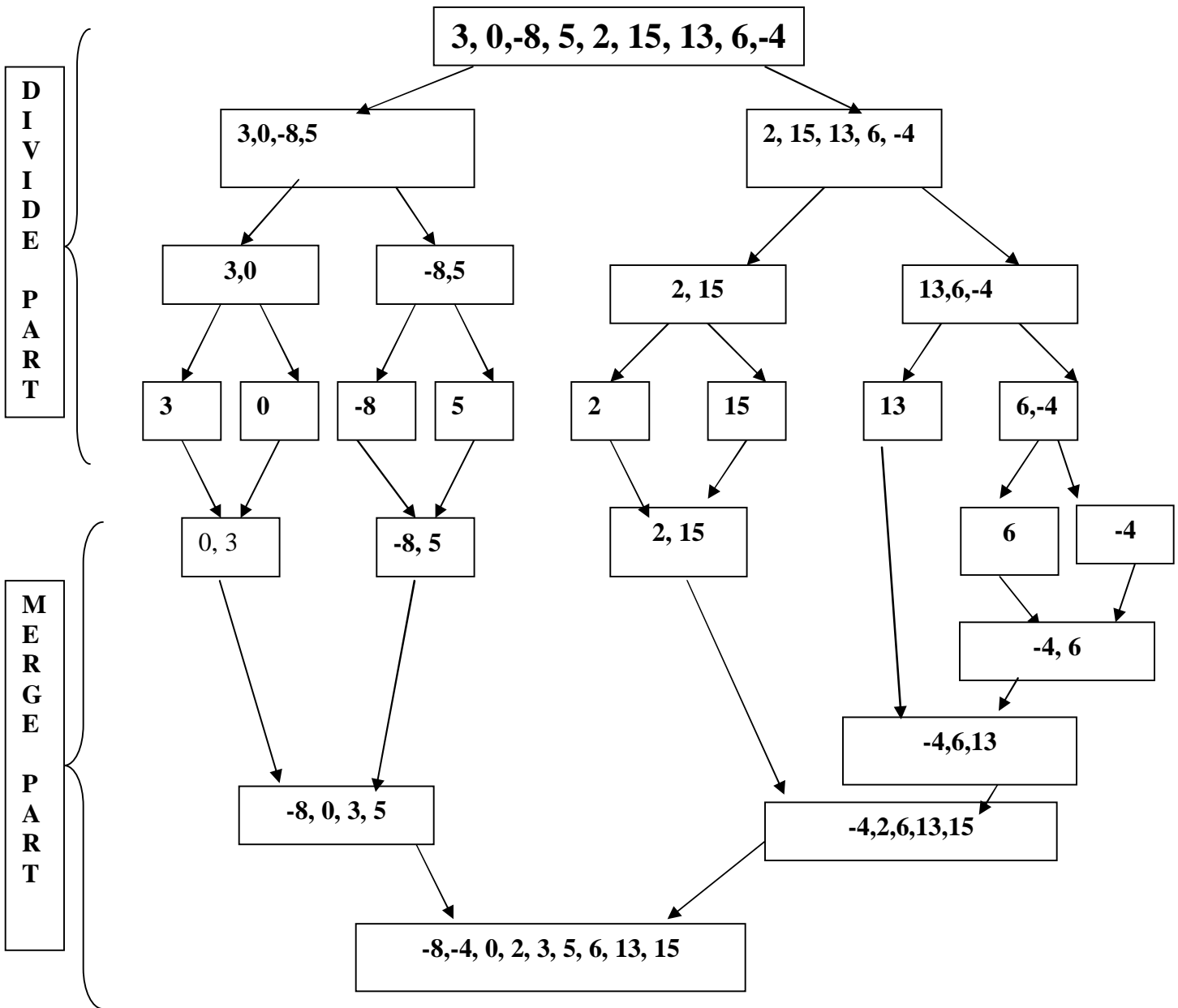
The procedure for merge sort contains two main parts viz. divide and merge.

- **Divide:** The original array is divided into two equal parts. Then each sub-array is divided into two equal parts. This method is continued till each sub array contains only one element.
- **Merge:** The first element of first sub-array is compared with first element of the second sub-array. The lesser among these is put into result-array. The remaining element is compared with the second element of the other array. The procedure is continued till both the arrays get exhausted.

The divide and merge parts are done recursively on given array to get sorted list.

Consider an array: 3, 0, -8, 5, 2, 15, 13, 6, -4





**Program:**

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

```
void Merge(int b[10], int c[10],int a[20],int p, int q)
{
    int i=0,j=0,k=0;
```

```
while(i<p && j<q)
{
    if(b[i]<c[j])
    {
        a[k]=b[i];
        i++;
    }
    else
    {
        a[k]=c[j];
        j++;
    }
    k++;
}
while(i<p)
{
    a[k]=b[i];
    i++;
    k++;
}
while(j<q)
{
    a[k]=c[j];
    j++;
    k++;
}
}

void MergeSort(int a[20],int n)
{
    int b[20],c[20],i,j,p,q;

    if(n>1)
    {
        for(i=0;i<n/2;i++)
            b[i]=a[i];
        p=i;

        for(i=n/2,j=0;i<n;i++,j++)
            c[j]=a[i];
        q=j;

        MergeSort(b,p);
        MergeSort(c,q);
        Merge(b,c,a,p,q);
    }
}
```

```

    return;
}

void main()
{
    int a[20],n,i;
    clrscr();
    printf("Enter the size of array:");
    scanf("%d",&n);
    printf("\nEnter elements:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    MergeSort(a,n);

    printf("\nSorted list is:\n");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    getch();
}

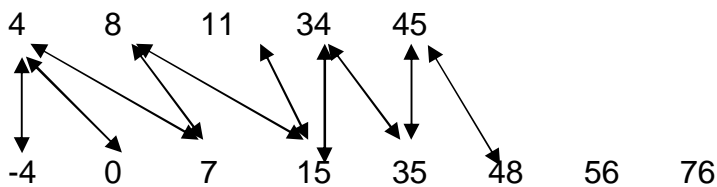
```

**NOTE:**

When there are two sorted arrays, then also we can sort them using merge sort. This technique is called as merging the sorted array. The procedure is explained as below using an example –

Sorted Array (i) : 4, 8, 11, 34, 45  
 Sorted Array (ii) : -4, 0, 7, 15, 35, 48, 56, 76

Procedure:



Result:

-4	0	4	7	8	11	15	34	35	45	48	56	76
----	---	---	---	---	----	----	----	----	----	----	----	----

The program for implementing merge sort when two sorted arrays are given, is as below.

**Program:**

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

void MergeSort(int a[10], int b[10],int c[20],int m, int n)
{
    int i=0,j=0,k=0;

    while(i<m && j<n)
    {
        if(a[i]<b[j])
        {
            c[k]=a[i];
            i++;
            k++;
        }
        else
        {
            c[k]=b[j];
            j++;
            k++;
        }
    }
    while(i<m)
    {
        c[k]=a[i];
        i++;
        k++;
    }
    while(j<n)
    {
        c[k]=b[j];
        j++;
        k++;
    }
}

void main()
{
    int a[10],b[10],c[20],m,n,i;
    clrscr();
    printf("Enter the size of first array:");
    scanf("%d",&m);
    printf("\nEnter first array(in sorted order):\n");
    for(i=0;i<m;i++)
        scanf("%d",&a[i]);
```

```

printf("Enter the size of second array:");
scanf("%d",&n);
printf("\nEnter second array(in sorted order):\n");
for(i=0;i<n;i++)
    scanf("%d",&b[i]);

MergeSort(a,b,c,m,n);

printf("\nSorted list is:\n");
for(i=0;i<m+n;i++)
    printf("%d\t",c[i]);
getch();
}

```

## 6.8 RADIX SORT

The radix sort is based on the idea that the number which is having the highest significant digit greater than the corresponding digit of another number will be larger. That is, if we consider two numbers viz. 673 and 512, compare highest significant digits i.e. 6 and 5. Then as 6 is greater than 5, the number 673 is greater than 512. If the two digits are same, we will take next digit and compare. The process continues till we get the result. For a sorting method by radix sort; we assume that all the numbers to be sorted are having equal number of digits i.e. all are two digit numbers or all are three digit numbers etc. There will be ten packets numbering from 0 to 9. Initially all the packets are empty. We will scan all the numbers to be sorted one by one and separate the least significant digit and insert that number into appropriate packet. If the packet is non-empty, the element is inserted into rear-end of the packet. Once all the elements have been scanned, they are removed from the packets. The process is repeated till the array gets sorted.

Consider an example for radix sort. We have to sort the numbers-

212, 310, 451, 117, 256, 813, 514, 315, 789, 618, 912, 513.

**Step (i):** Scan each number and put it into the appropriate packets based on last digit (least significant digit) of a number as below-

0	1	2	3	4	5	6	7	8	9
310	451	212	813	514	315	256	117	618	789
		912	513						

**Step (ii):** Now, remove elements from the packets-

310, 451, 212, 912, 813, 513, 514, 315, 256, 117, 618, 789

Put these elements into packets based on 2<sup>nd</sup> digit as follows –

0	1	2	3	4	5	6	7	8	9
	310				451			789	
	212				256				
	912								
	813								
	513								
	315								
	117								
	618								

Again remove elements from packets:

310, 212, 912, 813, 513, 514, 315, 117, 618, 451, 256, 789.

**Step (iii)** Now consider highest significant digit (first digit) and put the elements into packets.

0	1	2	3	4	5	6	7	8	9
	117	212	313	451	513	618	789	813	912
		256	315		514				

Remove from the packets-

117, 212, 256, 313, 315, 451, 513, 514, 618, 789, 813, 912

This is a sorted list.

To implement radix sort in C programming, we need a function which will separate the required significant digits, a function to insert an element at the rear-end of the list and to find the largest. For these, we go for linked list data structure as below –

```
# include<stdio.h>
# include<math.h>
# include<process.h>
# include<alloc.h>

struct node
{
    int data;
    struct node *link;
};
typedef struct node *NODE;

/* Use a function getnode() from linked list example*/
/* Use a function insert_rear() which is used for single list program*/

int separate(int item, int j)
{
    return item/(int) pow(10, j-1)%10;
}
```

```
int largest(int a[], int n)
{
    int i, big;
    big=a[0];
    for(i=1;i<n;i++)
        if(a[i]>big)
            big=a[i];
    return big;
}
void radix_sort(int a[], int n)
{
    int i, j, k, m, big, digit;
    NODE p[10], temp;
    big=largest(a,n);
    m=log 10(big)+1;
    For(j=1;j<=m;j++)
    {
        for(i=0;i<=9;i++) //0-9 packets
            p[i]=NULL;
        for(i=0;i<n;i++)
        {
            digit=separate (a[i],j);
            p[digit]=insert_rear(a[i], p[digit]);
        }
        k=0;
        for(i=0;i<=9;i++)
        {
            temp=p[i];
            while(temp!=NULL)
            {
                a[k++]=temp->data;
                temp=temp->link;
            }
        }
    }
}

void main()
{
    int n, i, a[20];
    printf("enter array size");
    scanf("%d",&n);
    printf("enter elements:\n");
    for(i=0;i<n;i++)
        scanf("%d", &a[i]);
}
```

```
radix_sort(a,n);
printf("\n sorted list");
for(i=0;i<n;i++)
    printf("%d",a[i]);
}
```

## 6.9 ADDRESS CALCULATION SORT

**NOTE: The readers are advised to understand the Hashing technique which is discussed in the next chapter (Searching) before reading this sorting technique.**

The address calculation sort is also known as sorting by hashing. In this method, a function  $f$  is applied on each element, where the function  $f$  should be such that,

$$\text{if } x \leq y, \text{ then } f(x) \leq f(y)$$

Such function is called as order preserving function. In this sorting method also, we assume that all the numbers to be sorted are having equal number of digits i.e. all are two digit numbers or all are three digit numbers etc. That is, if we consider two numbers viz. 673 and 512, compare highest significant digits i.e. 6 and 5. Then as 6 is greater than 5, the number 673 is greater than 512. If the two digits are same, we will take next digit and compare. The process continues till we get the result.

With the above assumption, the simplest function satisfying the above given equation, would be considering most significant digit of every number. That is,

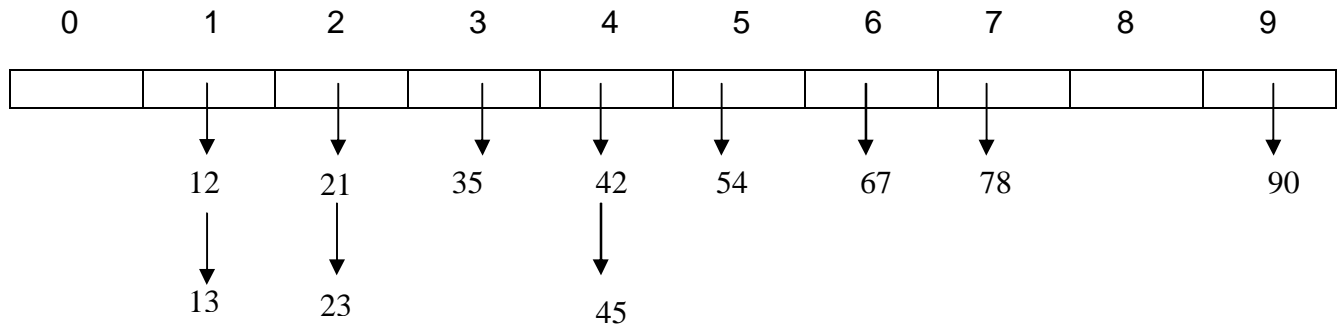
$$f(452) = 4, \\ f(127) = 1 \text{ etc.}$$

The sorting procedure is as follows: There will be ten packets numbering from 0 to 9 (similar to hash table). Initially all the packets are empty. Then, most significant digit of every item is computed, the item is hashed into respective packet. In case of hash collision, we use, open hashing (or separate chaining) using linked list. While inserting new item into a packet, we should see that the items in that packet are inserted in a sorted manner. After hashing all items into a hash table, just remove all the numbers from packet 0 till packet 9 sequentially. This will be a sorted list.

For example, consider the numbers: 21, 45, 13, 67, 42, 90, 78, 23, 54, 12, 35

Considering a hash function which results in most significant digit of a given number, the hash table would look like:





Observe that, while inserting 42, the number 45 was already hashed. But, 42 is inserted before 45. So, during each insertion into a packet, we should maintain an ordered list. Now, remove all the elements from packet 0 in an order. We will get a sorted list as:  
12, 13, 21, 23, 35, 42, 45, 54, 67, 78, 90

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

struct node //taken from linked list chapter
{
    int data;
    struct node *link;
};
typedef struct node *NODE;

NODE list[10]={NULL};

NODE getnode() //taken from linked list chapter
{
    NODE x;
    x=(NODE) malloc(sizeof(struct node));
    if(x==NULL)
    {
        printf("no memory in heap");
        exit(0);
    }
    return x;
}

NODE insert(int item, NODE start) //insert_order() function of linked lists
{
    NODE temp, cur, prev;
    temp = getnode();
```

```
temp->data=item;
temp->link=NULL;

if(start == NULL)
    return temp;

if(item < start ->data)
{
    temp->link =start;
    return temp;
}
prev = NULL;
cur = start;
while(cur != NULL && item >= cur->data)
{
    prev = cur;
    cur = cur->link;
}
prev->link = temp;
temp->link=cur;
return start;
}

int msd(int x)          //hash function to retrieve most significant digit of number
{
    while(x>10)
        x=x/10;

    printf("\n%d",x);
    return x;
}

void addcal(int arr[], int n)
{
    int i, j=0, pos;

    for(i=0;i<n;i++)
    {
        pos=msd(arr[i]);
        list[pos]=insert(arr[i],list[pos]);
    }

    for(i=0;i<10;i++)
    {
        while(list[i]!=NULL)
        {
```

```
        arr[j++]=list[i]->data;
        list[i]=list[j]->link;
    }
}

printf("\nSorted list is:\n");
for(i=0;i<n;i++)
    printf("%d\t",arr[i]);

}

void main()
{
    int n, a[20], i;
    clrscr();
    printf("Enter size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    addcal(a,n);
}
```

## UNIT 7. SEARCHING

### 7.1 INTRODUCTION

Searching is an operation which finds the location of a given element in the list. The search is said to be successful if the element is found in the list, otherwise the search is said to be unsuccessful. Some of the search techniques that we are going to discuss:

- Sequential or linear search
- Searching an ordered table
- Indexed sequential search
- Binary Search (already discussed during Recursion chapter)
- Interpolation Search
- Tree search

### 7.2 SEQUENTIAL / LINEAR SEARCH

Linear search method searches for a key element in the given list by comparing key with every element of an array. Here, each element is compared with the key element till the end of the list is reached or the element is found in between. In general, the linear search takes  $n$  comparisons for an array consisting of  $n$  elements.

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

int linear(int a[], int key, int n)
{
    int i;

    for(i=0;i<n;i++)
        if(a[i]==key)
            return i+1;

    return -1;
}

void main()
{
    int a[20], n, key, i, pos;

    printf("Enter size of the array:");
    scanf("%d",&n);
    printf("\nEnter array elements:\n");

    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    printf("\nEnter the key to be searched:");
```

```
scanf("%d",&key);

pos=linear(a, key, n);
if(pos==-1)
    printf("\nUnsuccessful search!!");
else
    printf("\nKey found at position %d",pos);
}
```

### 7.2.1 Searching an Ordered Table

To reduce the time complexity (that is, the number of comparisons) of linear search, we can apply linear search technique on a sorted list – this method is known as searching an ordered table.

Here, the key is compared with the elements in the array till the match is found or the key becomes greater than any in-between element.

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

int linear(int a[], int key, int n)
{
    int i;

    for(i=0;i<n;i++)
        if(a[i]==key)
            return i+1;
        else if(a[i]>key)
            break;

    return -1;
}

void main()
{
    int a[20], n, key, i, pos;

    printf("Enter size of the array:");
    scanf("%d",&n);
    printf("Enter array elements in ascending order:");

    for(i=0;i<n;i++)
        scanf("%d",&a[i]);

    printf("\nEnter the key to be searched:");
    scanf("%d",&key);
```

```

pos=linear(a, key, n);
if(pos==-1)
    printf("\nUnsuccessful search!!");
else
    printf("\nKey found at position %d",pos);
}

```

### 7.3 INDEXED SEQUENTIAL SEARCH

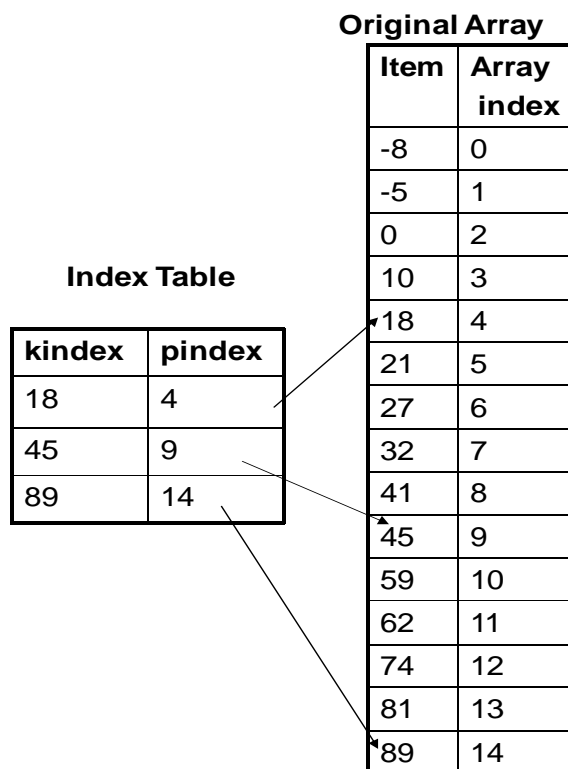
This technique is also used on sorted list, to reduce time complexity. But, this technique uses extra space.

Here, apart from original input array (which must be a sorted list), we will maintain another array called as **index table**. *Index table* consists of two parts:

- only few elements (called as *kindex*) of original array, which are equidistant.
- A pointer (called *pindex*) to the element in original array which corresponds to *kindex*.

The index table also must be a sorted list. Refer the diagram given in next slide to understand this. While searching for a key element, the index table is searched for. Then, only the respective sub-array of the original array is searched in a sequential manner.

Refer the following diagram:



Now, if we want to search for the key element 10, the *Index table* is searched for.

Since  $10 < 18$ , the first subarray of the original array  $\{-8, -5, 0, 10, 18\}$  is searched using a sequential search method. This is a successful search.

Now, assume we would like to search 75. Search the *Index Table* for 75. Since, 75 is in-between 45 and 89, we have to search in 3<sup>rd</sup> subarray :  $\{59, 62, 74, 81, 89\}$ . This is unsuccessful search.

## 7.4 INTERPOLATION SEARCH

This searching technique is also applied on a sorted list. If the elements are uniformly distributed between the first element and the last element then this algorithm works better than the binary search algorithm.

Consider the array:

$$a_0, a_1, a_2, a_3, \dots, a_{n-1}$$

Now, if  $a_1 - a_0 = a_2 - a_1 = a_3 - a_2$  and so on...., then we can say that array elements are uniformly distributed. Note, that, the difference between every two elements need not be exactly equal, but it can be almost equal. Now, the interpolation search follows almost same procedure as that of binary search, except for the calculation of middle element.

$$mid = low + (high - low) * \frac{(key - a[low])}{(a[high] - a[low])}$$

### //Interpolation Search

```
int interpol(int item, int a[],int low,int high)
{
    int mid;

    if(low<=high)
    {
        mid=low+(high-low)*((item-a[low])/(a[high]-a[low]));

        if(a[mid]==item)
            return mid+1;
        if(a[mid]>item)
            return interpol(item,a,low,mid-1);

        return interpol(item,a,mid+1,high);
    }
    return -1;
}
```

```
void main()
{
    int a[10],n,pos,item, i;

    printf("Enter the array size:");
    scanf("%d",&n);
    printf("Enter items in ascending order\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
}
```

```

printf("Enter the key element:");
scanf("%d",&item);

pos=interpol(item,a,0,n-1);
if(pos==-1)
    printf("\nItem not found");
else
    printf("\nItem found at %d",pos);
}

```

## 7.5 HASHING

Hashing is a way of representing dictionaries. Dictionary is an abstract data type with a set of operations searching, insertion and deletion defined on its elements. The elements of dictionary can be numeric or characters or most of the times, records. Usually, a record consists of several fields; each may be of different data types. For example, student record may contain student id, name, gender, marks etc. Every record is usually identified by some **key**.

Here we will consider the implementation of a dictionary of  $n$  records with keys  $k_1, k_2 \dots k_n$ . Hashing is based on the idea of distributing keys among a one-dimensional array

$H[0 \dots m-1]$ , called **hash table**.

For each key, a value is computed using a predefined function called **hash function**. This function assigns an integer, called **hash address**, between 0 to  $m-1$  to each key. Based on the hash address, the keys will be distributed in a hash table.

For example, if the keys  $k_1, k_2, \dots, k_n$  are integers, then a hash function can be

$$h(K) = K \bmod m.$$

Let us take keys as 65, 78, 22, 30, 47, 89. And let hash function be,

$$h(k) = k \% 10.$$

Then the hash addresses may be any value from 0 to 9. For each key, hash address will be computed as –

$$h(65) = 65 \% 10 = 5$$

$$h(78) = 78 \% 10 = 8$$

$$h(22) = 22 \% 10 = 2$$

$$h(30) = 30 \% 10 = 0$$

$$h(47) = 47 \% 10 = 7$$

$$h(89) = 89 \% 10 = 9$$

Now, each of these keys can be hashed into a hash table as –

0	1	2	3	4	5	6	7	8	9
30		22			65		47	78	89



In general, a hash function should satisfy the following requirements:

- A hash function needs to distribute keys among the cells of hash table as evenly as possible.
- A hash function has to be easy to compute.

### 7.5.1 Hash Collisions

Let us have  $n$  keys and the hash table is of size  $m$  such that  $m < n$ . As each key will have an address with any value between 0 to  $m-1$ , it is obvious that more than one key will have same hash address. That is, two or more keys need to be hashed into the same cell of hash table. This situation is called as **hash collision**.

In the worst case, all the keys may be hashed into same cell of hash table. But, we can avoid this by choosing proper size of hash table and hash function. Anyway, every hashing scheme must have a mechanism for resolving hash collision. There are two methods for hash collision resolution, viz.

- Open hashing
- closed hashing

### 7.5.2 Open Hashing (or Separate Chaining)

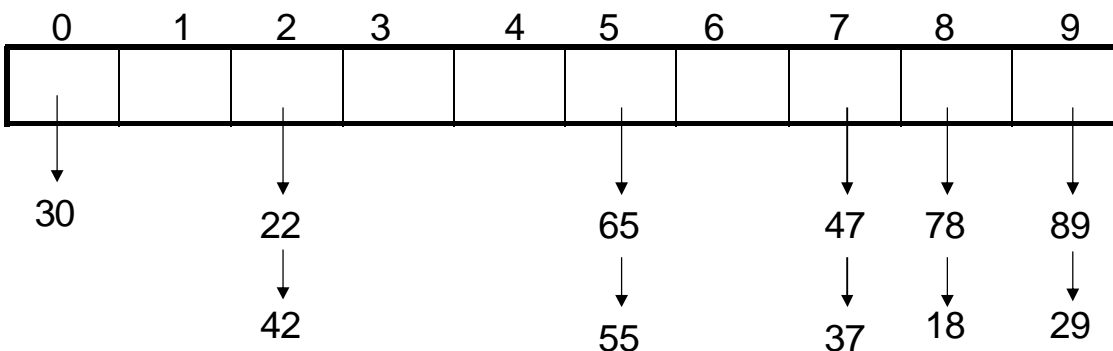
In open hashing, keys are stored in linked lists attached to cells of a hash table. Each list contains all the keys hashed to its cell. For example, consider the elements

65, 78, 22, 30, 47, 89, 55, 42, 18, 29, 37.

If we take the hash function as  $h(k) = k \% 10$ , then the hash addresses will be –

$h(65) = 65 \% 10 = 5$	$h(78) = 78 \% 10 = 8$
$h(22) = 22 \% 10 = 2$	$h(30) = 30 \% 10 = 0$
$h(47) = 47 \% 10 = 7$	$h(89) = 89 \% 10 = 9$
$h(55) = 55 \% 10 = 5$	$h(42) = 42 \% 10 = 2$
$h(18) = 18 \% 10 = 8$	$h(29) = 29 \% 10 = 9$
$h(37) = 37 \% 10 = 7$	

The hash table would be –



### Operations on Hashing:

- **Searching:** Now, if we want to search for the key element in a hash table, we need to find the hash address of that key using same hash function. Using the obtained hash address, we need to search the linked list by tracing it, till either the key is found or list gets exhausted.
- **Insertion:** Insertion of new element to hash table is also done in similar manner. Hash key is obtained for new element and is inserted at the end of the list for that particular cell.
- **Deletion:** Deletion of element is done by searching that element and then deleting it from a linked list.

### 7.5.3 Closed Hashing (or Open Addressing)

In this technique, all keys are stored in the hash table itself without using linked lists. Different methods can be used to resolve hash collisions. The simplest technique is *linear probing*.

This method suggests to check the next cell from where the collision occurs. If that cell is empty, the key is hashed there. Otherwise, we will continue checking for the empty cell in a circular manner. Thus, in this technique, the hash table size must be at least as large as the total number of keys. That is, if we have  $n$  elements to be hashed, then the size of hash table should be greater or equal to  $n$ .

Example:

Consider the elements 65, 78, 18, 22, 30, 89, 37, 55, 42

Let us take the hash function as  $h(k) = k \% 10$ , then the hash addresses will be –

$$\begin{array}{ll} h(65) = 65 \% 10 = 5 & h(78) = 78 \% 10 = 8 \\ h(18) = 18 \% 10 = 8 & h(22) = 22 \% 10 = 2 \\ h(30) = 30 \% 10 = 0 & h(89) = 89 \% 10 = 9 \\ h(37) = 37 \% 10 = 7 & h(55) = 55 \% 10 = 5 \\ h(42) = 42 \% 10 = 2 & \end{array}$$

Since there are 9 elements in the list, our hash table should at least be of size 9. Here we are taking the size as 10.

Now, hashing is done as below –

0	1	2	3	4	5	6	7	8	9
30	89	22	42		65	55	37	78	18

### Drawbacks:

- Searching may become like a linear search and hence not efficient.

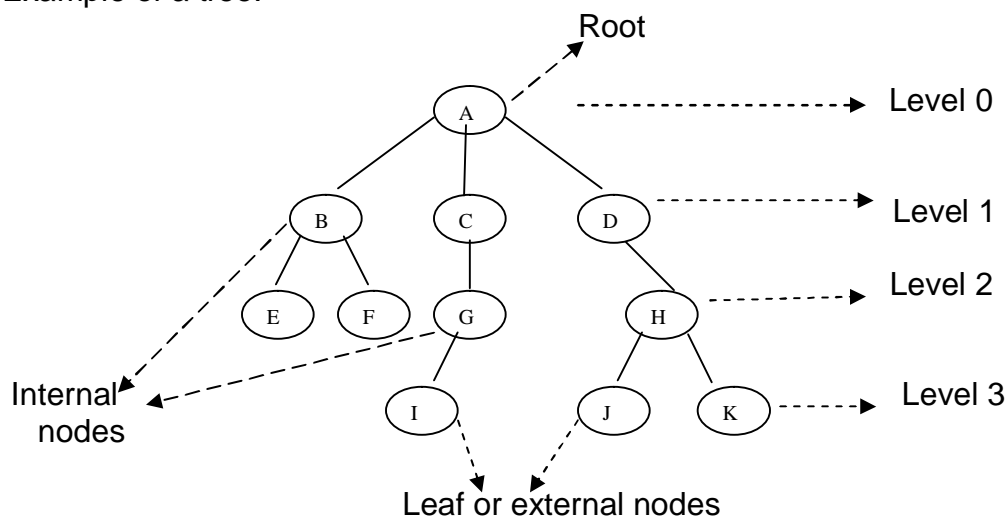
## UNIT 8. TREES

### 8.1 INTRODUCTION

The data structures that we studied till now such as stack, queue, linked list etc, were of linear in nature. That is, the inter-relationship between the elements of those data structures is linear. But, in the field of computer science, we come across many situations where the data are interrelated in hierarchical structure. In this case, a linear representation of data is not possible. The solution for this problem is a data structure called *tree*, which is a non-primitive non-linear data structure.

Tree is a data structure used to represent hierarchical relationship existing among several data items. Here, each data item is referred to as *node*. Each node may be empty or may be connected to some other nodes.

Example of a tree:



Referring to the above diagram, let us define some of the terminologies used in trees. In the diagram,

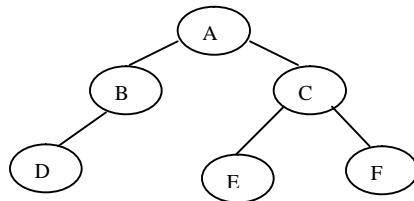
- A, B, C etc. are known as **nodes** of a tree.
- A is known as the **root**.
- B, C and D are called as **children** of A. Similarly, J and K are children of H and so on.
- A is referred as **father** of B, C and D.
- B, C and D are known as **siblings** of each other.
- The node, which is not having any children is called as **leaf** or **terminal** or **external node**.
- A tree structure, which is connected to root is known as a **subtree**.
- The number of subtrees connected to a node is known as a **degree of that node**. In the figure, A is having degree 3, B is of degree 2, C is of degree 1 etc.
- The maximum number representing the degree of any node in a tree is called as **degree of a tree**. Here, degree of tree is 3.

- The entire tree structure is leveled such that the root is at **level 0** and any other node is having the level one more than the level of its father.
- The **depth** or **height** of a tree is the maximum level of that tree.
- Collection of disjoint trees is known as the **forest**.

## 8.2 BINARY TREES

Binary tree is a finite set of data items, which is either empty or partitioned into three disjoint subsets. The first subset contains only one item known as root. The other two subsets are themselves binary trees known as *left subtree* and *right subtree*. Thus, in a binary tree, maximum degree of any node is at most 2.

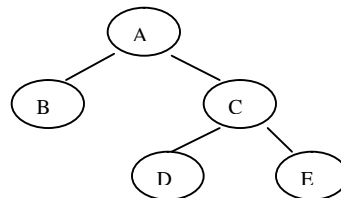
Ex:



In the above figure, A is the root of a binary tree. A tree structure having B as a root is known as left subtree of the tree with root A. Similarly, the tree structure having C as root is the right subtree of the given tree.

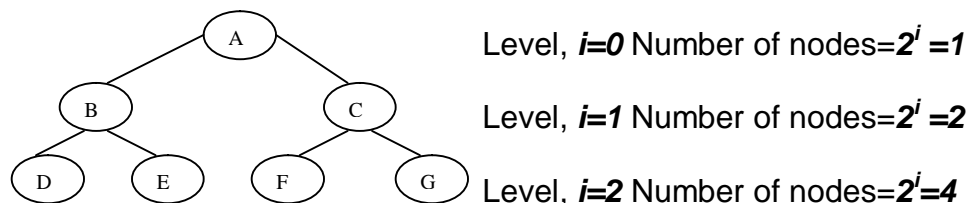
A binary tree in which any node is either empty or consisting of both left subtree and right subtree is known as **strictly binary tree**.

For ex:



A strictly binary tree is which the number of nodes at any level  $i$  is  $2^i$  then the tree is said to be **complete binary tree**.

Ex:



### 8.2.1 Applications of Binary Trees

Binary tree is a useful data structure while taking two-way decisions at each point of a process. Consider an example to illustrate this fact.

Suppose an array of numbers is given. The problem is to find duplicate numbers in the array. If the array is large, comparing each element with the other is quite difficult and is time consuming. So, for the sake of simplicity the applications of tree concept can be used. The procedure is as follows: Take a first number in the array and put it as a root of the binary tree. Now, take the second number compare it with the root. If the match is found, we can conclude that the number is duplicate. If the number is smaller than the root put it as a left subtree otherwise put it as a right subtree. Now the next number is compared with root then with left subtree and right subtree if necessary. The procedure is repeated till the array is completed.

Binary tree concept can be used for the efficient method of searching and sorting also. It can also be used for evaluating an expression by creating an expression tree.

### 8.2.2 Operations on Binary Trees

Few of important operations on trees are-

- Traversing a tree
- Insertion of a node
- Deletion of a node
- Searching for a node

### 8.2.3 Traversing a Tree

Traversal of a tree is a method of visiting each node of a tree exactly once in a systematic way. There are three different methods for tree traversal, viz.

- i) Pre-order traversal
- ii) In-order traversal
- iii) Post-order traversal

The rules for these traversals are as below-

#### Pre-order traversal:

- i) Visit the root.
- ii) Traverse left subtree using pre-order traversal method.
- iii) Traverse right subtree using pre-order traversal method.

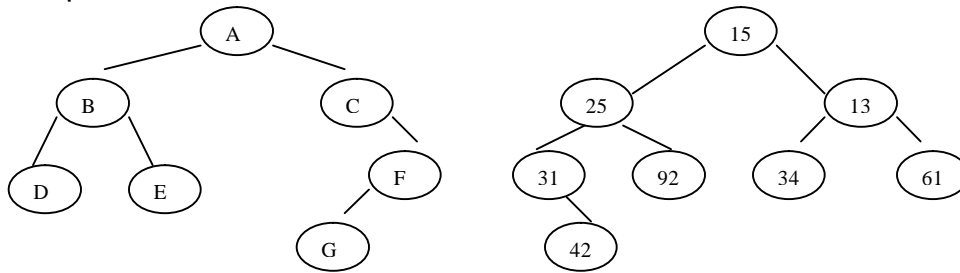
#### In-order traversal:

- i) Traverse left subtree using in-order traversal method.
- ii) Visit the root.
- iii) Traverse right subtree using in-order traversal method.

#### Post-order traversal:

- i) Traverse left subtree using post-order traversal method.
- ii) Traverse right subtree using pos-order traversal method.
- iii) Visit the root.

Example:



Pre-order:

ABDECFG

15, 25, 31, 42, 92, 13, 34, 61

In-order:

DBEACGF

31, 42, 25, 92, 15, 34, 13, 61

Post-order:

DEBGFCA

42, 31, 92, 25, 34, 61, 13, 15

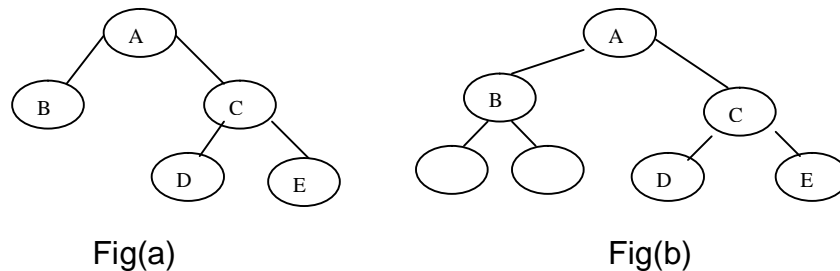
### 8.2.4 Representation of Binary Trees

For programmatic implementation, tree can be represented in two ways.

- **Array Representation**

Here, each element of a tree is treated as an element of the array i.e. in a binary tree, assumption is made that each node is present at all the levels and leaving blank space for absent nodes, the array elements are filled-up.

Ex: Consider a binary tree in fig(a).



Since this binary tree is having a maximum level of 2, fill-up all the nodes till level 2 as in fig (b). Now, the array representations will be-

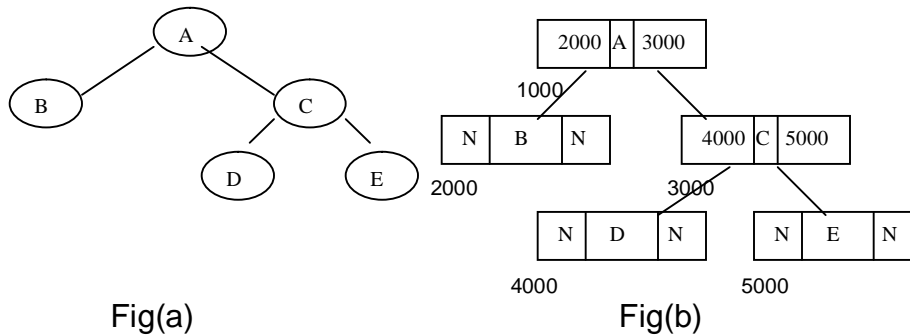
A	B	C			D	E
---	---	---	--	--	---	---

But, as we know, the array requires contiguous memory blocks to store the elements; we will go for node representations of trees.

- **Node/Linked list Representation**

Here, each of the elements in a binary tree is treated as a node in doubly linked list having three fields viz, left link, data and right link.

Ex: For the tree in fig (a), the node representation will be as in fig (b).



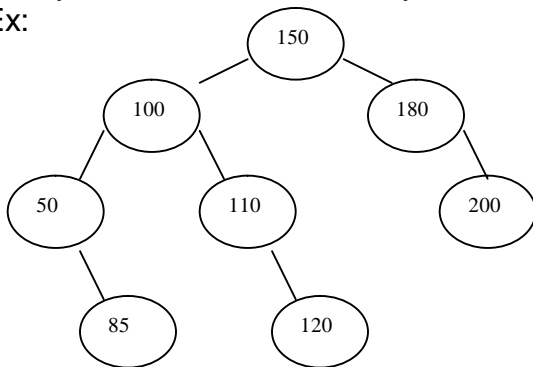
For the programming purpose, we create a node in a tree as-

```
struct node
{
    struct node *llink;
    int data;
    struct node *rlink;
};
typedef struct node * NODE;
```

### 8.3 BINARY SEARCH TREES

Binary search tree is a binary tree in which for each node, the elements in left subtree are less than it and the elements in right subtree are greater than or equal to it. Every node of binary search tree must satisfy this condition.

Ex:



To insert a new node into BST, it is compared with root. If it is less than the root, traverse towards left subtree. If it is greater than or equal to root, traverse towards right subtree and insertion is made appropriately. Care to be taken such that after insertion also, the tree remains BST. Similarly, for deletion of a node also, such precaution is to be taken.

Note that in-order traversal of a BST will give the sorted list in ascending order. So, binary tree sort is nothing but a creation of BST and then its in-order traversal.

**Program: Implementation (creation) of Binary Search Trees and Traversal Techniques**

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

struct node
{
    struct node *llink;
    struct node *rlink;
    int data;
};
typedef struct node *NODE;

NODE getnode()          //Function to get memory from heap
{
    NODE x;
    x = (NODE) malloc (sizeof (struct node));
    if (x == NULL)
    {
        printf("No memory space\n");
        exit(0);
    }
    return x;
}

NODE insert (int item, NODE root)      //Function to create binary search tree by inserting
elements
{
    NODE temp,prev,cur;

    temp = getnode();
    temp->data = item;
    temp->rlink = NULL;
    temp->llink = NULL;

    if(root == NULL)
        return temp;

    prev = NULL;
    cur = root;

    while (cur != NULL)
    {
        prev = cur;
```



```
        cur = (item < cur->data) ? cur->llink : cur->rlink;
    }
    if (item < prev->data)
        prev->llink = temp;
    else
        prev->rlink = temp;

    return root;
}

void preorder(NODE root)        //Function to traverse tree in pre-order
{
    if(root != NULL)
    {
        printf("%d\t",root->data);
        preorder(root->llink);
        preorder(root->rlink);
    }
}

void inorder(NODE root)        //Function to traverse tree in in-order
{
    if(root != NULL)
    {
        inorder(root->llink);
        printf("%d\t",root->data);
        inorder(root->rlink);
    }
}

void postorder(NODE root)        //Function to traverse tree in post-order
{
    if(root != NULL)
    {
        postorder(root->llink);
        postorder(root->rlink);
        printf("%d\t",root->data);
    }
}

void main()
{
    NODE root = NULL;
    int opt,item;
```

```
for(;;)
{
    printf("\nCreating a Binary Tree and Traversing the tree\n");
    printf("Enter your option\n");
    printf("1:Insert an element to tree\n");
    printf("2:Pre-Order Traversal\n");
    printf("3:In-Order Traversal\n");
    printf("4:Post-Order Traversal\n");
    printf("5:Exit\n");
    scanf("%d",&opt);
    switch(opt)
    {
        case 1: printf("Enter the element to be inserted \n");
                scanf("%d",&item);
                root = insert(item,root);
                break;
        case 2: printf("PREORDER TRAVERSAL\n");
                preorder(root);
                break;
        case 3: printf("INORDER TRAVERSAL\n");
                inorder(root);
                break;
        case 4: printf("POSTORDER TRAVERSAL\n");
                postorder(root);
                break;
        case 5:
        default: exit(0);
    }
}
}
```

**Following is a function for deleting a node from BST. This function can be included in the above program, if you want delete option.**

```
/* Function to delete a node whose data field is given*/
NODE deletenode(int key, NODE root)
{
    NODE cur, parent, suc, q;

    if(root==NULL)
    {
        printf("empty tree");
        return root;
    }

    parent=NULL;
```

```
cur=root;

while(cur!=NULL && key!=cur->data)
{
    parent=cur;
    cur=(item<cur->data)? cur->llink : cur->rlink;
}

if(cur==NULL)
{
    printf("key not found");
    return root;
}

if(cur->llink==NULL)
    q=cur->rlink;
else if(cur->rlink==NULL)
    q=cur->llink;
else
{
    suc=cur->rlink;
    while(suc->llink!=NULL)
        suc=suc->llink;

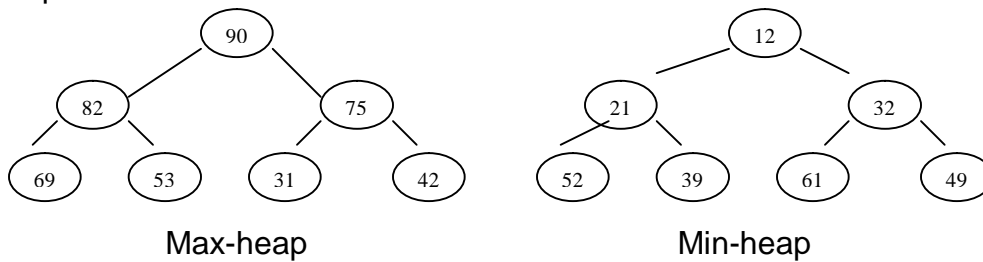
    suc->llink=cur->llink;
    q=cur->rlink;
}

if(parent==NULL)
    return q;
if(cur==parent->llink)
    parent->llink=q;
else
    parent->rlink=q;
freenode(cur);
return root;
}
```

## 8.4 HEAPS

Heap is a complete binary tree. There are two different types of heap viz. max-heap and min-heap. In a tree, if every node is greater than its children, then it is a max-heap or descending heap. If every node is less than its children, then it is a min-heap or ascending heap.

Example:



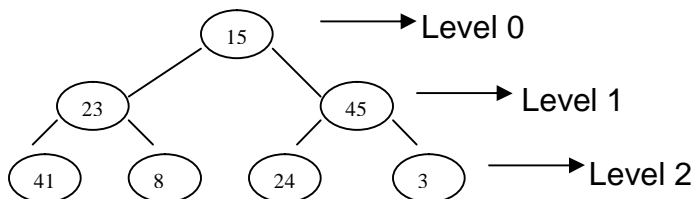
While inserting an element into the heap and deleting an element from the heap proper care should be taken so that after insertion/deletion, the resulting tree should be a heap of the same type as before.

### 8.4.1 Construction of a Heap

To construct a heap from a given set of elements, a binary tree is formed first. Then, starting from the level one less than the maximum level of a binary tree, the heap must be formed, considering the children of each node. The procedure is repeated till entire tree becomes a help of a specific type.

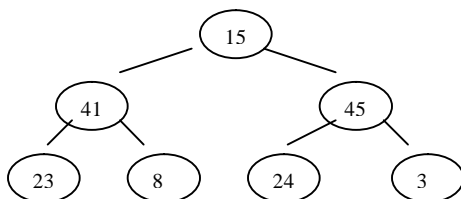
Ex. Consider the elements: 15, 23, 45, 41, 8, 24, 3. The problem is to construct a max-heap.

Step (i) Form a tree-

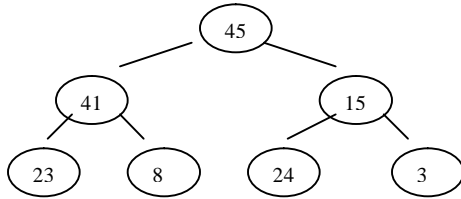


Step(ii) Consider the element 45. As 45 is greater than both of its children (24 and 3), no change occurs. Now compare the children of 23.

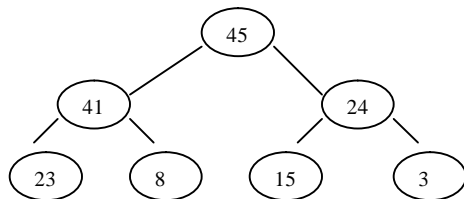
$41 > 8$  and  $41 > 23$  also. So, exchange 41 with 23. Thus, the tree would be –



Step(iii) Now, compare 41 and 45. As  $45 > 15$ , they are exchanged and the tree will be –



Step (iv) Again compare 15 with 24 and 3. Since  $24 > 15$ , they are exchanged.



which is a max-heap

Using same methodology, one can build min-heap also.

### 8.4.2 Heap Sort

Following are the steps for sorting a set of  $n$  elements using heap sort.

- i) Construct a max-heap from the given elements.
- ii) Exchange the first and last elements of this heap.
- iii) Discard the last element now, and re-construct the heap using remaining  $n-1$  elements.
- iv) Repeat step (ii) and step (iii) till only one element remains in the heap.
- v) Write all the discarded elements in the reverse order and this will be the sorted list.

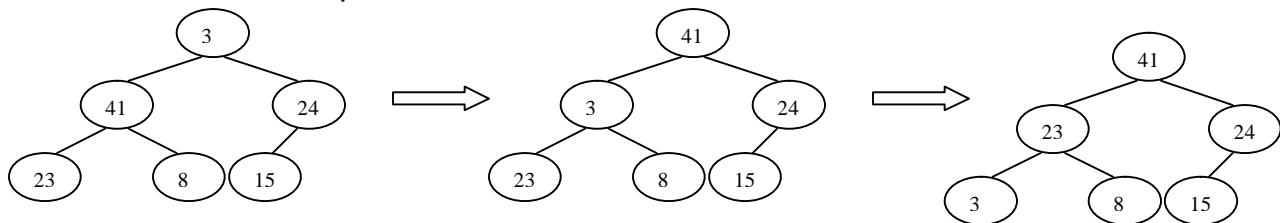
For example, consider the elements used in the previous section. After creating a heap, the elements will be –

45, 41, 24, 23, 8, 15, 3

Now, exchange 45 and 3. Eliminate 45. So, the list becomes–

3, 41, 24, 23, 8, 15

Now, create a max-heap–

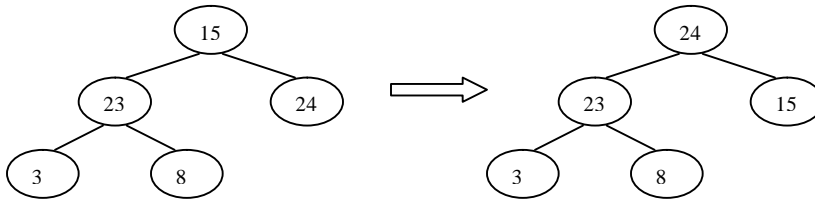


Write in-order: 41, 23, 24, 3, 8, 15

Exchange 41&15 and eliminate 41. So, the list becomes –

15, 23, 24, 3, 8.

Create a heap:

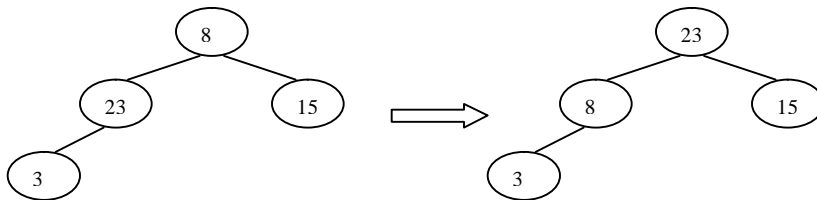


Write in-order: 24, 23, 15, 3, 8

Exchange 24 & 8. Eliminate 24. The remaining list is –

8,23,15,3

Create a heap:

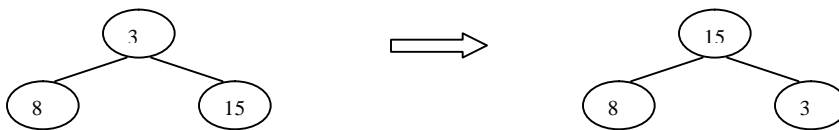


Write in the order: 23, 8, 15, 3.

Exchange 23 and 3 and then eliminate 23. The list remained is –

3, 8, 15

Create heap:

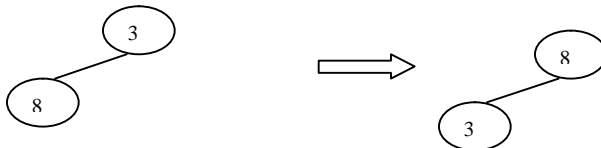


Write in the order: 15, 8, 3

Exchange 15 and 3 and eliminate 15. The list is –

3, 8

Create a heap:



Write in order: 8, 3

Exchange 8 and 3 and eliminate 8. Now the list contains only one element viz. 3

Now, write all the eliminated items –

45, 41, 24, 23, 15, 8, 3

The reverse of this list is the sorted list.

**/\*Program for heap sort\*/**

```
# include<stdio.h>
void createheap(int a[], int n)
{
    int i,j,k,item;
    for(k=1;k<n;k++)
    {
        item=a[k];
        i=k;
        j=(i-1)/2;

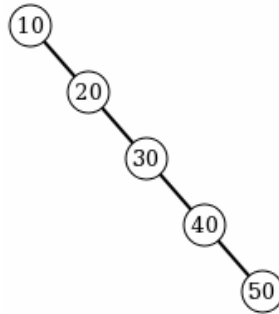
        while(i>0 && item>a[j])
        {
            a[i]=a[j];
            i=j;
            j=(i-1)/2;
        }
        a[i]=item;
    }
}

void heapsort(int a[],int n)
{
    int i, temp;
    createheap(a,n);
    for(i=n-1;i>0;--)
    {
        temp=a[0];
        a[0]=a[i];
        a[i]=temp;
        createheap(a,i);
    }
}

void main()
{
    int a[20],n, i;
    printf("enter number of elements in array");
    scanf("%d", &n);
    printf("enter elements:\n");
    for(i=0;i<n;i++)
        scanf("%d", &a[i]);
    heapsort(a,n);
    printf("\n sorted list:\n");
    for(i=0;i<n;i++)
        printf("%d\n", a[i]);
}
```

## 8.5 BALANCED SEARCH TREES (AVL TREES)

We have seen that, major application of binary search tree is for searching. But, if our binary search tree is skewed at one side, as shown in the below diagram, then search process will be linear, and it is not efficient.



To overcome this, problem, we balance the height of BST such a way that, the difference between heights of left subtree and right subtree of any node is either 0 or 1 or -1. Such a tree is called as balanced search tree. Balanced Search Tree is also known as AVL (G M Adelson-Velsky and E M Landis) trees.

**Balanced Search Tree is a Binary Search tree in which the balance factor for every node is either -1, 0 or 1.**

**Balance factor = height of left subtree – height of right subtree**

After every insertion and deletion, we have to compute the balance factor for every node. If tree is unbalanced, then we have to balance it and then proceed further for next insertion/deletion.

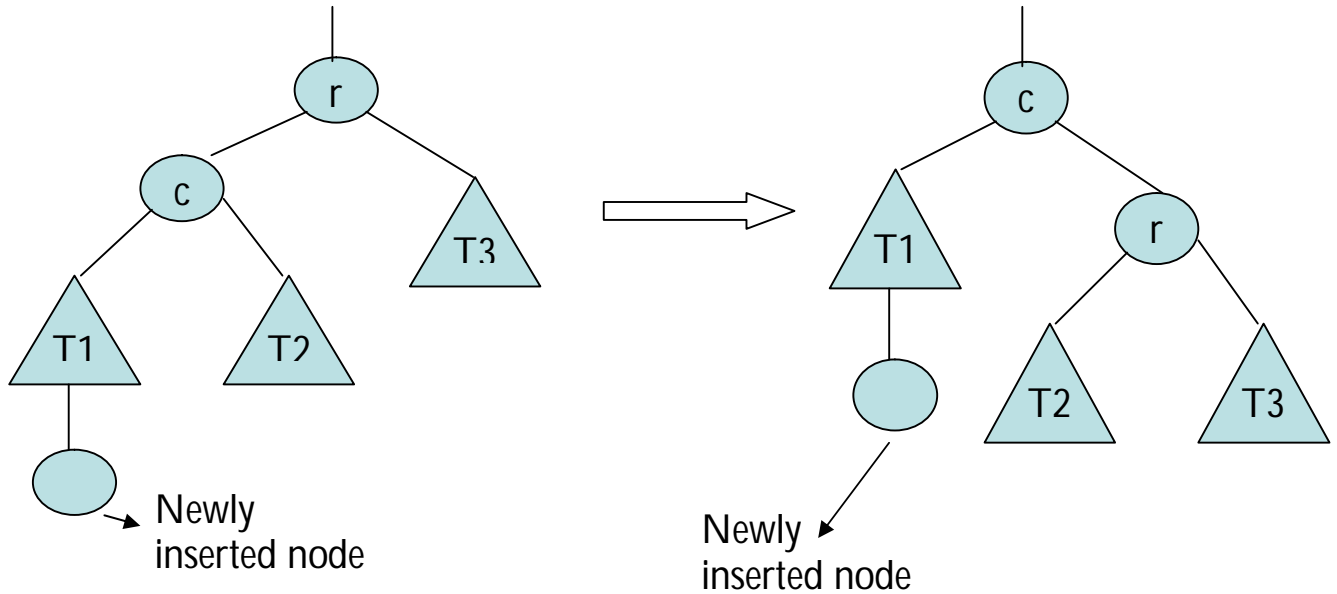
To balance a tree, we transform the tree by rotation. A rotation in an AVL tree is a local transformation of its sub tree rooted at a node whose balance has become either +2 or -2. If there are several such nodes, we rotate the tree rooted at the unbalanced node that is nearer to the newly inserted node (normally, from bottom of the tree).

There are 4 types of rotations.

### Single Right Rotation (R-Rotation):

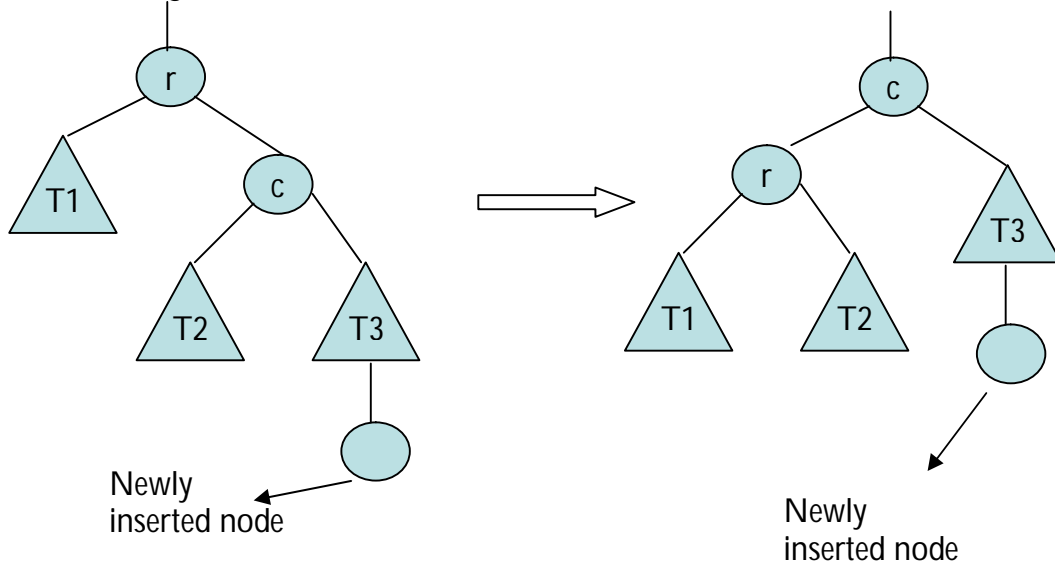
- This rotation is performed after a new element is inserted into left sub tree of the left child of tree, whose root had the balance factor 1 before insertion.
- The general form is –





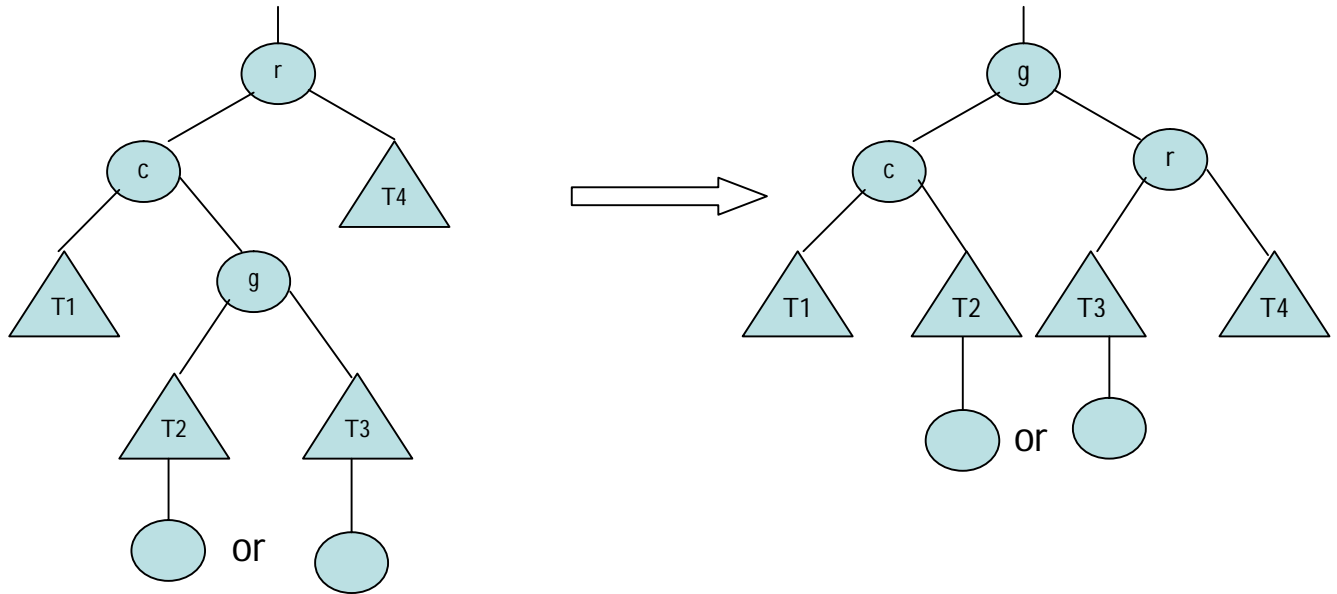
**Single Left Rotation (L-Rotation):**

- This rotation is performed when a new element is inserted into right sub tree of the right child of tree, whose root had the balance factor -1 before insertion.
- The general form is –



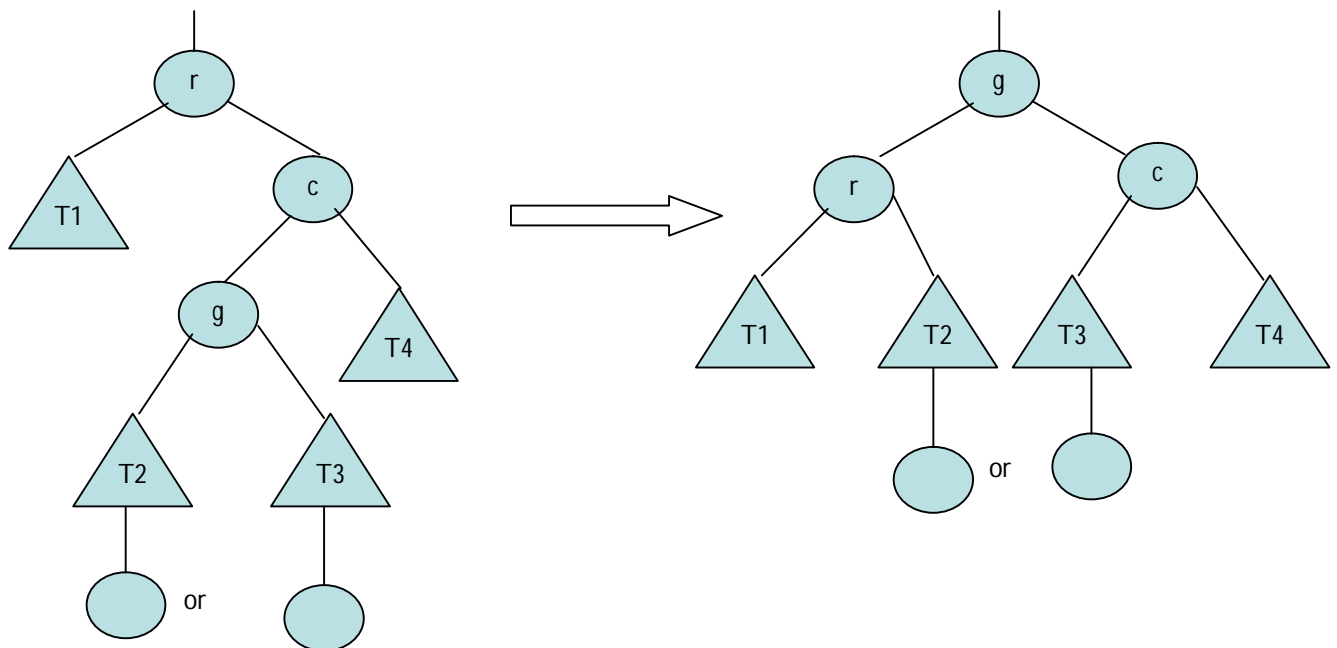
**Double Left-Right Rotation (LR-Rotation):**

- It is a combination of left and right rotations.
- We will perform L-rotation of the left sub tree of root *r* followed by R-rotation of the new tree rooted at *r*.
- This rotation is done when a new node is inserted into right sub tree of the left child of a tree whose root had the balance factor of 1 before insertion.



**Double Right-Left Rotation (RL-Rotation):**

- It is a mirror image of LR rotations.
- This rotation is done when a new node is inserted into left sub tree of the right child of a tree whose root had the balance factor of -1 before insertion.



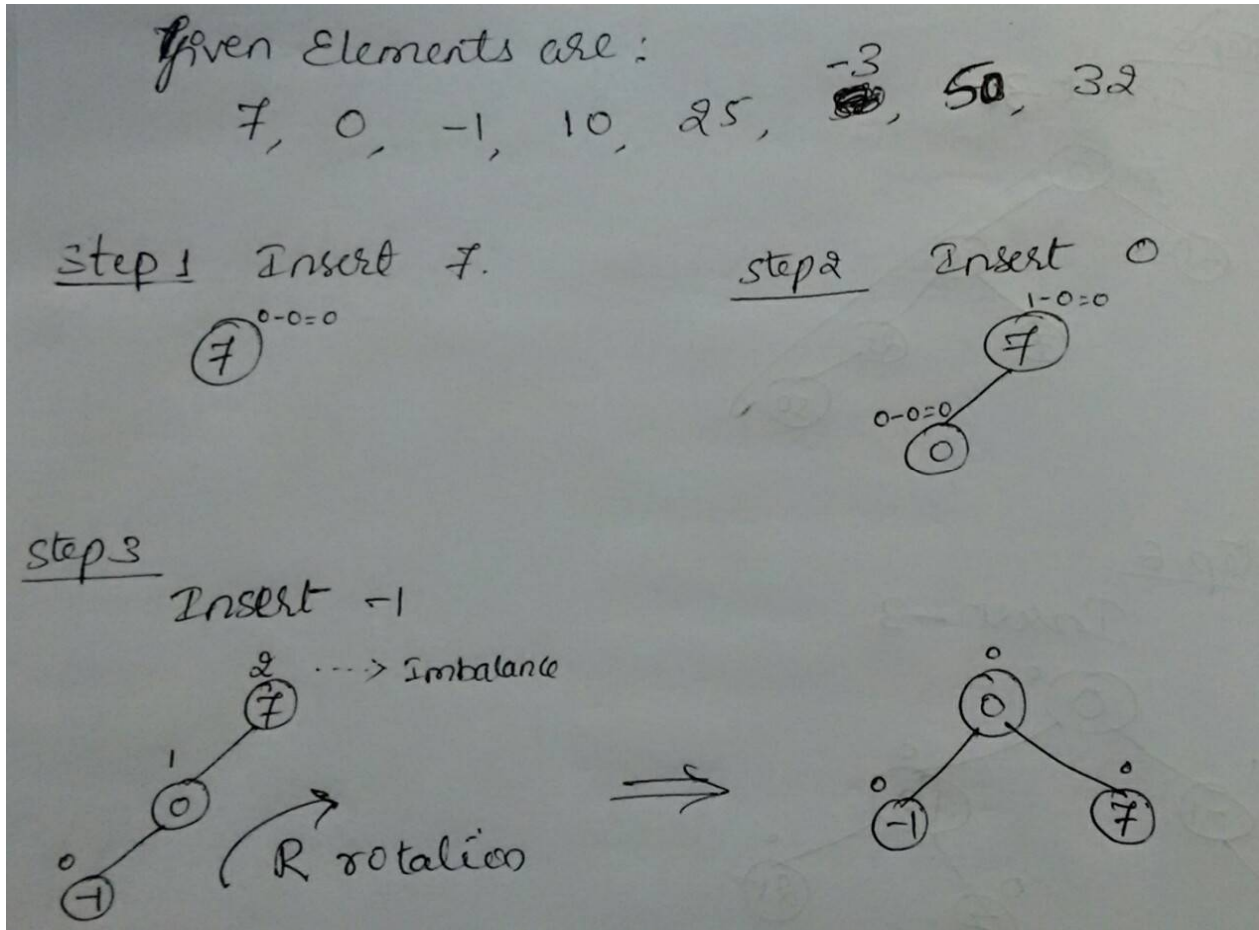
**Creation of AVL Tree:**

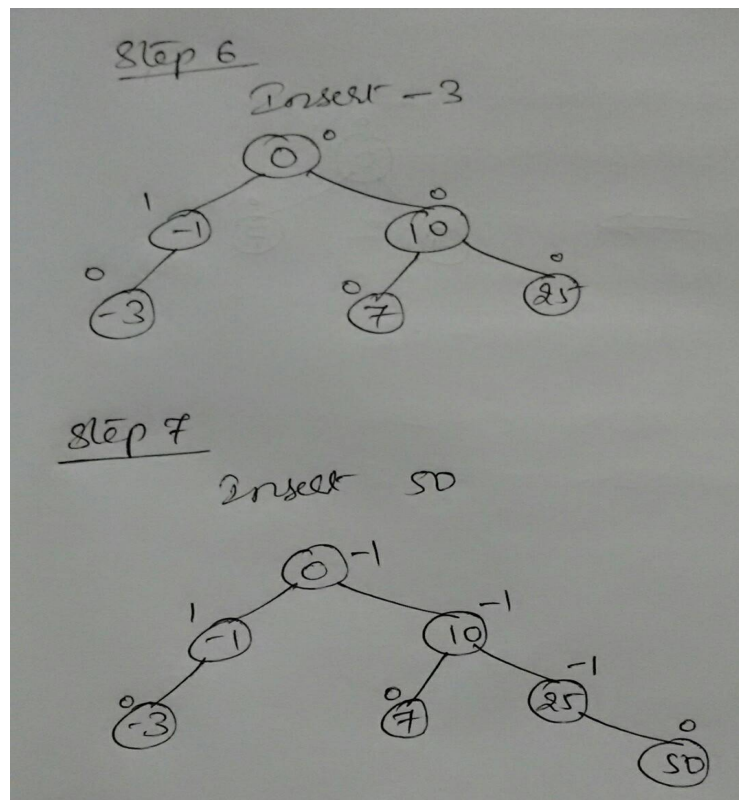
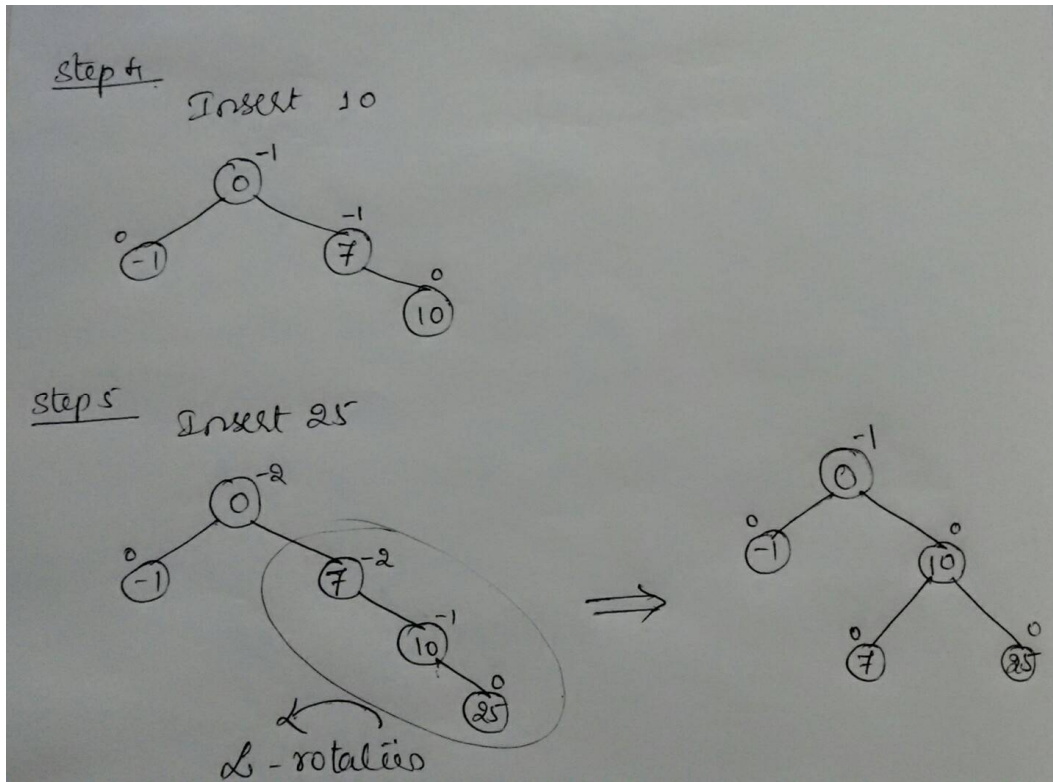
Given a set of numbers, the creation of AVL tree has following steps:

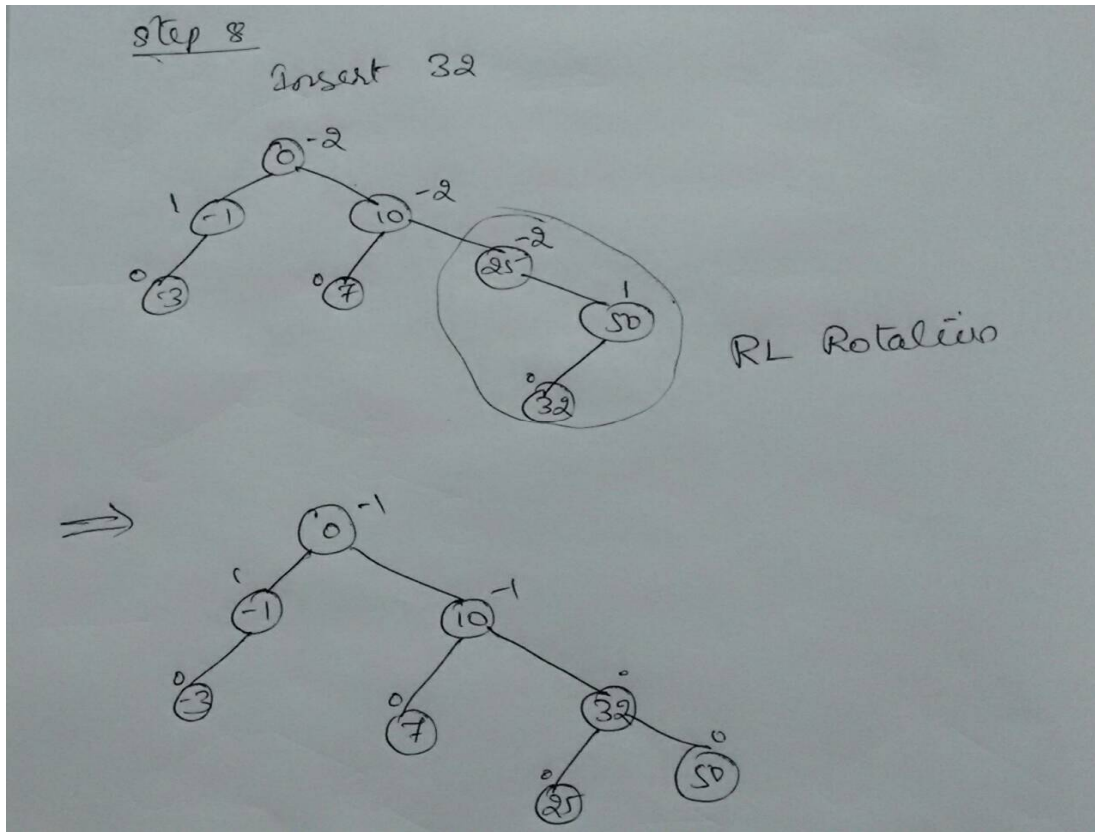
1. Take first element as root.

2. Compare the next element with the root and place it as a left child if it is lesser than the root. If the element is greater than the root, place it as a right child. (Same as Binary Search Tree).
3. Compute the balance factor for every node. If there is an imbalance at any node, apply suitable rotation only on that portion and balance the tree.
4. Apply steps (2) and (3) till you insert all the elements into an AVL tree.

Consider an example as shown below:

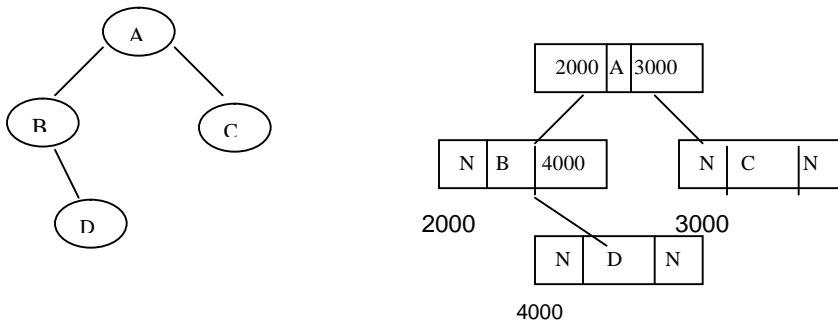






### 8.6 THREADED BINARY TREES

Since each of the traversal methods are recursive in nature, every time the program needs to push the items into the stack and to pop-up from the stack. This is a time consuming process. Consider the example –



The inorder traversal of this tree would be B D A C. To traverse this tree, program control reaches 'A' first. Then as it is found to have left subtree, A is pushed into the stack. Then as B don't have any left subtree, it gets printed. Then its right subtree, D gets printed. Now, program control will visit the stack to search for the elements and as A is found in it, it gets printed. In this manner, the flow continues, and all the elements get printed.

Note the point that if we would have had the address of the node A in the right link of D, searching in the stack was not necessary i.e if the node for D was like-

N	D	1000
---	---	------

then we would have directly printed A. But, during program execution it may not be possible to identify that whether the address 1000 is of right child of the node D or it is of the immediate inorder-successor of D. Hence, we will introduce one more field to indicate this concept. As a solution for this problem, we will construct the structure as-

```
struct node
{
    int data;
    int lchild;
    int rchild;
    struct node *llink;
    struct node *rlink;
};
typedef struct node *NODE;
```

Here, lchild and rchild will take the values 0 or 1. The field lchild having the value 0 indicates the address at the field llink is not the address of left child but it is of immediate in-order predecessor. The value 1 indicates the address at llink is of left child. Similarly, the same concept holds for right child i.e. rchild field and rlink field.

The operations on threaded binary tree are done depending on the values of lchild and rchild and keeping the logic as that of ordinary binary tree.

In general, the link fields of a tree can be used to store the address of higher-level nodes. The link field which keeps the address of higher-level node is called as *thread*. A binary tree having threads is *threaded binary tree*.

Threading may correspond to any of the three traversals.

- In-order Threading
- Pre-order Threading
- Post-order Threading

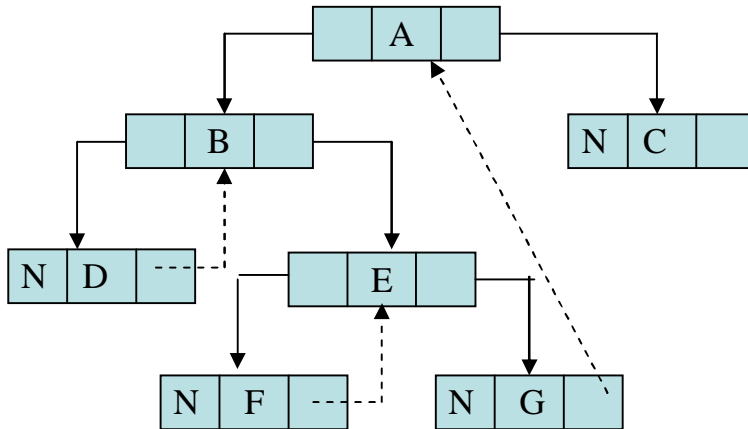
Each of these type may be of

- One-way threading
- Two-way threading

### In-Order Threading

- **One-way In-order Threading:** The right link field of the node will have the address of its in-order successor.

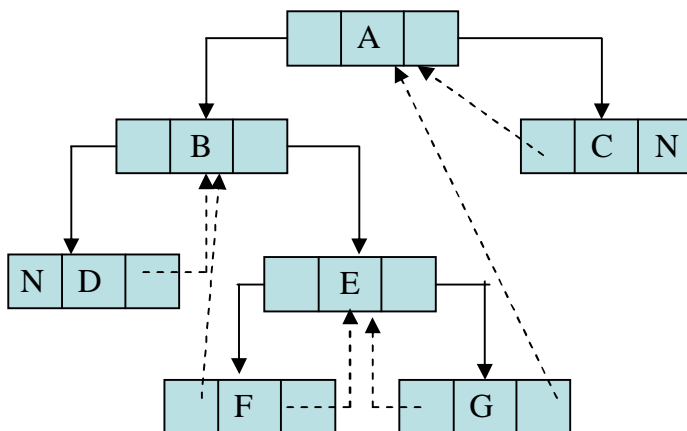
For example, In-order traversal of tree is D B F E G A C



Here, dotted lines indicate the link fields containing the address of their in-order successors.

- **Two-way In-order Threading:** The right link field of the node will have the address of its in-order successor. And the left link field of the node will have the address of its in-order predecessor. It is also known as **Fully threaded binary tree**.

For example, In-order traversal of tree is D B F E G A C



In the similar manner, we can create threaded binary trees for pre-order and post-order.