

## **Introduction:**

**C** is a general-purpose computer programming developed in 1972 in the Bell Telephone Laboratories for use with the UNIX operating system. Although C was designed for implementing system software it is also widely used for developing portable application software. C is one of the most popular programming languages. It is widely used on many different system software, and there are few computer architecture for which a compiler does not exist. C has greatly influenced many other popular programming languages, most notably C++, which originally began as an extension to C.

C is an imperative (procedural) systems implementation language. It was designed to be compiled using a relatively straightforward compiler, to provide low-level access to memory, to provide language constructs that map efficiently to machine instructions, and to require minimal run time support. C was therefore useful for many applications that had formerly been coded in assembly language.

## **Main article: Operators in C**

C supports a rich set of operators, which are symbols used within an expression to specify the manipulations to be performed while evaluating that expression. C has operators for:

- arithmetic (+, -, \*, \, %)
- equality testing (=, !=)
- order relations (<, <=, >, >=)
- boolean logic (!, &&, ||)
- bitwise logic (~, &, |, ^)
- bitwise shifts (<<, >>)
- assignment (=, +=, -=, \*=, /=, %=, &=, |=, ^=, <<=, >>=)
- increment and decrement (++ , --)
- reference and dereference (&, \*, [ ])
- conditional evaluation (?, :)
- member selection (., ->)
- type conversion (( ))
- object size (sizeof)
- function argument collection (( ))
- sequencing (,)
- subexpression grouping (( ))

## **Characteristics**

Like most programming languages, C has facilities for structured programming and allows lexical variable scope and recursion, while a static type system prevents many unintended operations. In C, all executable code is contained within functions. Function parameters are always passed by value. Pass-by-reference is achieved in C by explicitly passing pointer values. Heterogeneous aggregate data types (`struct`) allow related data elements to be combined and manipulated as a unit. C program source text is free-format, using the semicolon as a statement terminator (not a delimiter).

## **C also exhibits the following more specific characteristics:**

- lack of nested function definitions
- variables may be hidden in nested blocks
- partially weak typing; for instance, characters can be used as integers
- low-level access to computer memory by converting machine addresses to typed pointers
- function and data pointers support run-time polymorphism
- array indexing as a secondary notion, defined in terms of pointer arithmetic
- a preprocessor for macro definition, source code file inclusion, and conditional compilation
- complex functionality such as I/O, string manipulation, and mathematical functions consistently delegated to library routines
- A relatively small set of reserved keywords

## **C does not have some features that are available in some other programming languages:**

- No direct assignment of arrays or strings (copying can be done via standard functions; assignment of objects having `struct` or `union` type is supported)
- No automatic garbage collection
- No requirement for bounds checking of arrays
- No operations on whole arrays
- No syntax for ranges, such as the `A . . . B` notation used in several languages
- No separate Boolean type (zero/nonzero is used instead)
- No formal closures or functions as parameters (only function and variable pointers)
- No generators or coroutines; intra-thread control flow consists of nested function calls, except for the use of the `longjmp` or `setcontext` library functions
- No exception handling; standard library functions signify error conditions with the global `errno` variable and/or special return values
- Only rudimentary support for modular programming
- No compile-time polymorphism in the form of function or operator overloading
- Only rudimentary support for generic programming
- Very limited support for object-oriented programming with regard to polymorphism and inheritance
- Limited support for encapsulation

- No native support for multithreading and networking
- No standard libraries for computer graphics and several other application programming needs

A number of these features are available as extensions in some compilers, or can be supplied by third-party libraries, or can be simulated by adopting certain coding disciplines.

## **Memory management**

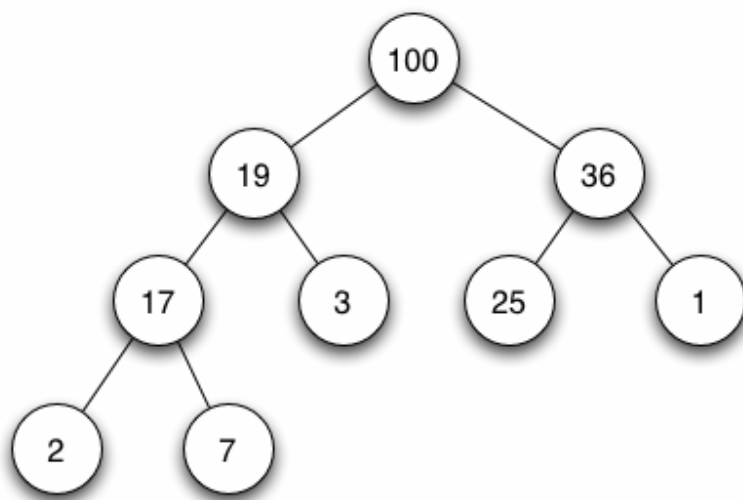
One of the most important functions of a programming language is to provide facilities for managing memory and the objects that are stored in memory. C provides three distinct ways to allocate memory for objects:

- Static memory allocation: space for the object is provided in the binary at compile-time; these objects have an extent (or lifetime) as long as the binary which contains them is loaded into memory
- Automatic memory allocation: temporary objects can be stored on the stack, and this space is automatically freed and reusable after the block in which they are declared is exited
- Dynamic memory allocation: blocks of memory of arbitrary size can be requested at run-time using library functions such as `malloc` from a region of memory called the heap; these blocks persist until subsequently freed for reuse by calling the library function `free`

## Heap Data Structure:

A Heap data structure is a binary tree with the following properties:

1. It is a complete binary tree; that is, each level of the tree is completely filled, except possibly the bottom level. At this level, it is filled from left to right.
2. It satisfies the heap-order property: The data item stored in each node is greater than or equal to the data items stored in its children.



Example of a full binary max heap

A **heap** is a specialized tree-based data structure that satisfies the *heap property*: if  $B$  is a child node of  $A$ , then  $\text{key}(A) \geq \text{key}(B)$ . This implies that an element with the greatest key is always in the root node, and so such a heap is sometimes called a *max-heap*. (Alternatively, if the comparison is reversed, the smallest element is always in the root node, which results in a *min-heap*.)

## Accessing the Heap Values

- Given the index  $i$  of a node, the indices of its parent  $Parent(i)$ , left-child  $LeftChild(i)$  and right child  $RightChild(i)$  can be computed simply :

Parent(i) return  $i/2$

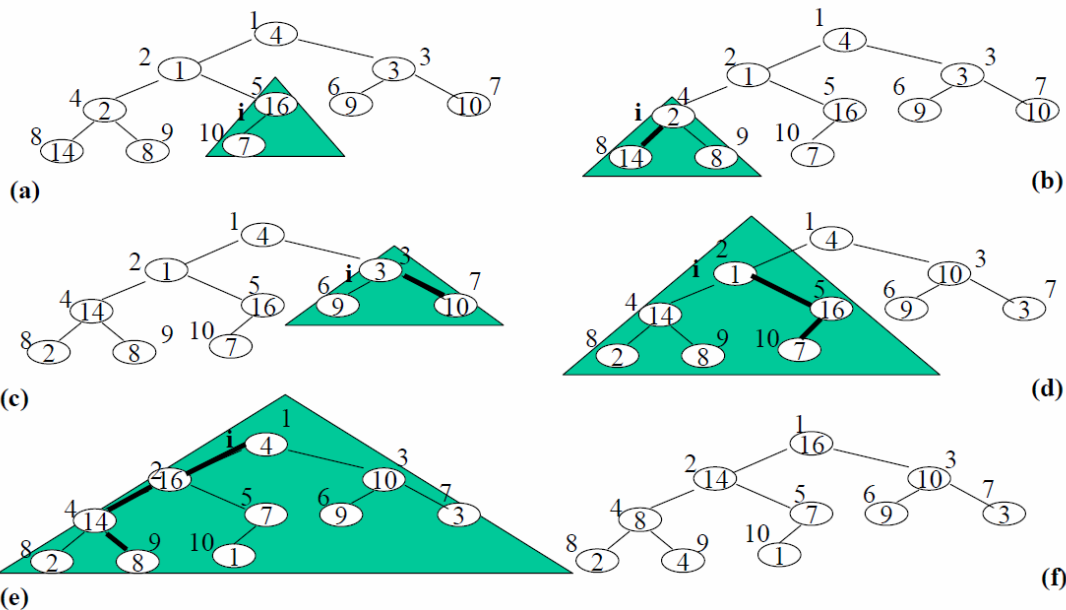
LeftChild(i) return  $2i$

RightChild(i) return  $2i+1$

## Example of Build Heap

**A**

4	1	3	2	16	9	10	14	8	7
---	---	---	---	----	---	----	----	---	---



### **Problem 1:-**

Write a program to take a list of integer value as input( values should be in heap order). Write a procedure that changes the value at a certain note (index). Write another procedure that orders the changed heap into a heap again.

### **Code:**

```
/*Including the header files*/
#include<stdio.h>
#include<conio.h>

void main()
{ //Declaing Variables and functions
    void crheap(int [],int);
    int a[20],i,n,x,y;
    printf("Enter the size of the array:-");
    scanf("%d",&n);
    printf("Enter the elements\n");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    printf("\n Heap create");
    printf("\nEnter number to enter :- ");
    scanf("%d",&x);
    printf("\nEnter position where to enter the number :- ");
    scanf("%d",&y);
    printf("\n");
    a[y]=x;
    //Calling the Heap Create function
    crheap(a,n);
    //Displaying the sorted array
    printf("\nArray after sorting:-\n");
    for(i=0;i<n;i++)
        printf("%d\t",a[i]);
    getch();
}

//Heap Create function is defined
void crheap(int a[],int n)
{
    int i,j,k=1,key;
    //Heap Create logic
    do
    {
        i=k;
        key=a[k];
        j=(int)((i-1)/2);
```

```

        while((i>0) && (key>a[j]))
        {
            a[i]=a[j];
            i=j;
            j=(int)((i-1)/2);
            if(j<0)
                j=0;
        }
        a[i]=key;
        ++k;
    }while(k<n);
}

```

Output:

Output1:

```

C:\> "D:\msc\c-ds\dipl\assignment\hpCreat\Debug\Hp-1.exe"
Enter the size of the array:-6
Enter the elements
99
24
17
7
23
13

Enter number to enter :- 63
Enter position where to enter the number :- 1

Array after sorting:-
99    63    17    7    24    13    _

```

Output 2:-

```
C:\ "D:\msc\c-ds\dipl\assignment\hpCreat\Debug\Hp-1.exe"
Enter the size of the array:-5
Enter the elements
82
19
12
3
14

Enter number to enter :- 102
Enter position where to enter the number :- 2

Array after sorting:-
102    19    82    3    14    _
```

## Algorithm:

from main

Calling the Heap Create function

we pass two parameters a and n

a is the array and n is the no. of element

crheap(a,n);

i hold the index of the child node

j holds the index of the parent node

key holds the value of the child node

crheap(a, n)

```
{
    do
    {
        i<-k
        key=a[k]
        j<-((i-1)/2)
        while((i>0) && (key>a[j]))
        {
            a[i]<-a[j]
            i<-j
            j<-((i-1)/2)
            if(j<0) then
                j<-0
        }
    }
```



```

        a[i]<-key
        ++k
    }while(k<n)
}

```

## Heap sort complexity

Let, there is a tree of level k, if we apply the heap create algorithm then for a node in (k-1) level the loop will execute 2 times. 1<sup>st</sup> time the condition of while will be satisfied. And it will enter to the loop again but this time the condition will not satisfied. So, in (k-1) level the loop will be execute  $2 \cdot 2^{(k-1)}$  times. Likewise in k-2 level for a single node the loop will be execute 3 times and so on. So we can write—

$$\begin{aligned}
 & 2 \cdot 2^{(k-1)} + 3 \cdot 2^{(k-2)} + 4 \cdot 2^{(k-3)} + \dots + (k+1) \cdot 2^0 \\
 &= -2^k + 2^k + 2 \cdot 2^{(k-1)} + 3 \cdot 2^{(k-2)} + 4 \cdot 2^{(k-3)} + \dots + (k+1) \cdot 2^0 \\
 &= -2^k + 2^{k+1} (2^{-1} + 2 \cdot 2^{-2} + 3 \cdot 2^{-3} + \dots) \\
 &= -2^k + 2^{k+1} (1/2 + 2 \cdot (1/2^2) + 3 \cdot (1/2^3) + \dots)
 \end{aligned}$$

Converge  $(-1 < r < 1)$

value at that range,  
 $(1/2) / (1 - (1/2)) = 2$

so,  $-2^k + 2^{k+2}$

$< 4n$  where n = no. of nodes.

So, time complexity of create heap is  $O(n)$ .

## Conclusion:

The algorithm is not written by the percolate() and shift-down() function. It just takes the inputs and makes the heap checking the inputs from the first.

# **Sorting**

In computer science and mathematics, a **sorting algorithm** is an algorithm that puts elements of a list in a certain order. The most-used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

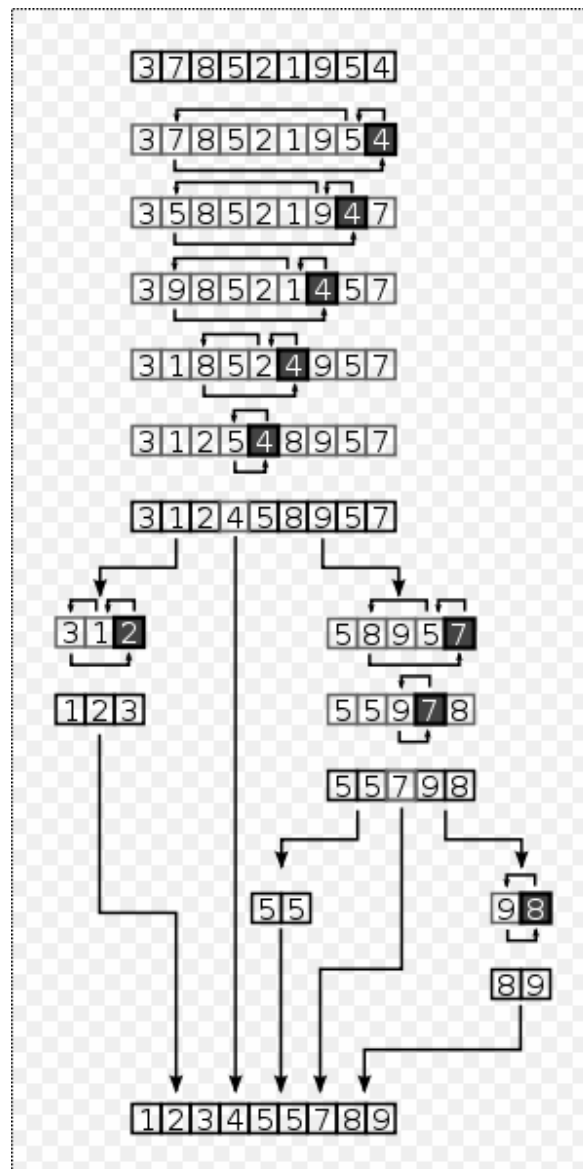
1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation, or reordering, of the input.

## Quick sort

Quick sort is often named as partition exchange sort. Let  $x$  be an array, and  $n$  the number of elements in the array to be sorted. We chose an element  $a$  from a specific position within the array. Suppose that the elements of  $x$  are partitioned so that  $a$  is placed into position  $j$ . The element  $a$  will be the pivot element.

The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the **partition** operation.
3. Sort the sub-list of lesser elements and the sub-list of greater elements.



Full example of quick sort on a random set of numbers. The boxed element is the pivot. It is here chosen as the last element of the partition.

## **Problem 2:-**

Create a structure in c that contains the following information about train:

i)train Id

ii)distance

iii)train Id

iv)destination

create a file name train.txt and add 10 train information in it manually. Write a program to sort the stored train information using quick sort (non-recursive) according to

i)distance

ii)destination

## **Code:**

```
/*Including the header files*/
#include<stdio.h>
#include<conio.h>
#include<string.h>

/*creating a structure train*/
struct train
{
    int id,dist,noc;
    char dest[10];
};

//quick sort function for distance sorting is defined
int quick1(struct train input[],int beginning, int end)
{
    int pos,left,right;
    struct train temp;
    left=beginning;
    right=end;
    pos=beginning;
    /* scan from right to left */
    do
    {
        while(input[pos].dist<=input[right].dist && pos!=right)
```

```

        {
            right--;
        }
        if(pos==right)
            return(pos);
        else
        {
            temp=input[pos];
            input[pos]=input[right];
            input[right]=temp;
            pos=right;
        }
        /*      scan from left to right      */
        while(input[pos].dist>=input[left].dist && pos!=left)
        {
            left++;
        }
        if(pos==left)
            return(pos);
        else
        {
            temp=input[pos];
            input[pos]=input[left];
            input[left]=temp;
            pos=left;
        }
    } while(left<=right);
}

```

//quick sort function for destination sorting is defined

```

int quick2(struct train input[],int beginning, int end)
{
    int pos,left,right;
    struct train temp;
    left=beginning;
    right=end;
    pos=beginning;
    /* scan from right to left      */
    do
    {
        while(input[pos].dest[0]<=input[right].dest[0] && pos!=right)
        {
            right--;
        }
        if(pos==right)
            return(pos);
    }
}

```

```

        else
        {
            temp=input[pos];
            input[pos]=input[right];
            input[right]=temp;
            pos=right;
        }
        /*      scan from left to right      */
        while(input[pos].dest[0]>=input[left].dest[0] && pos!=left)
        {
            left++;
        }
        if(pos==left)
            return(pos);
        else
        {
            temp=input[pos];
            input[pos]=input[left];
            input[left]=temp;
            pos=left;
        }
    }while(left<=right);
}

void main()
{ //Declaing Variables and functions
    struct train ob[10];
    int n=0,lower[100],upper[100],i,ch;
    int beginning,end,pos,top;
    //declaring file pointer
    FILE *fp;
    //open train.txt in read mode
    fp=fopen("train.txt","r");
    if(fp==NULL)
    {
        printf("file not found");
        exit(0);
    }
    do
    {
        //reading data from file
        fscanf(fp,"%d|%d|%d|%s\n",&ob[n].id,&ob[n].dist,&ob[n].noc,&ob[n].dest);
        n++;
    }while(!feof(fp));
    //closing train.txt
    fclose(fp);
}

```

```

do
{
    printf("\n\QUICK SORT NON-RECURSION
    ");
    printf("\n\t1.distance sort");
    printf("\n\t2.destination sort");
    printf("\n\t3.Exit");
    printf("\n\n\tEnter Choice : - ");
    scanf("%d",&ch);
    switch(ch)
    {
        //Sub menu for distance wise sorting
        case 1:
            top=-1;
            if(n>=1)
            {
                top=top+1;
                lower[top]=0;
                upper[top]=n-1;
            }
            while(top!=-1)
            {
                beginning=lower[top];
                end=upper[top];
                top=top-1;
                //Calling the quick Sort function
                pos=quick1(ob,beginning,end);
                if(beginning+1<pos)
                {
                    top=top+1;
                    lower[top]=beginning;
                    upper[top]=pos-1;
                }
                if(end-1>pos)
                {
                    top=top+1;
                    lower[top]=pos+1;
                    upper[top]=end;
                }
            }
            //open test5.txt in write mode
            fp=fopen("test5.txt","w");
            if(fp == NULL)
            {
                printf("\nCannot open file");
            }
        }
    }
}

```

```

        exit(0);
    }
    for(i=0;i<n;i++)
    {
        fprintf(fp,"%d %d %d
%s\n",ob[i].id,ob[i].dist,ob[i].noc,ob[i].dest);
    }
    //closing test5.txt
    fclose(fp);
    break;

```

//Sub menu for destination wise sorting

case 2:

```

    top=-1;
    if(n>=1)
    {
        top=top+1;
        lower[top]=0;
        upper[top]=n-1;
    }
    while(top!=-1)
    {
        beginning=lower[top];
        end=upper[top];
        top=top-1;
        //Calling the quick Sort function
        pos=quick2(ob,beginning,end);
        if(beginning+1<pos)
        {
            top=top+1;
            lower[top]=beginning;
            upper[top]=pos-1;
        }
        if(end-1>pos)
        {
            top=top+1;
            lower[top]=pos+1;
            upper[top]=end;
        }
    }
    //open test6.txt in write mode
    fp=fopen("test6.txt","w");
    if(fp == NULL)
    {
        printf("\nCannot open file");
    }

```



```

        exit(0);
    }
    for(i=0;i<n;i++)
    { //write data into file
        fprintf(fp,"%d %d %d
%s\n",ob[i].id,ob[i].dist,ob[i].noc,ob[i].dest);
    }
    //closing test6.txt
    fclose(fp);
    break;
case 3:
    exit(0);
default:
    printf("\nWrong Input: Re Enter");
    break;
    }
}while(1);
}

```

Output:

```

C:\ "E:\Program\Programs\WSc_CompSc\mou\ds\Debug\train_qu_i.exe"
QUICK SORT NON-RECURSION
  1.distance sort
  2.destination sort
  3.Exit
Enter Choice : - 1
QUICK SORT NON-RECURSION
  1.distance sort
  2.destination sort
  3.Exit
Enter Choice : - 2
QUICK SORT NON-RECURSION
  1.distance sort
  2.destination sort
  3.Exit

```

Input:

```

2|23|2|kol
3|12|4|del
78|81|7|chen
45|5|10|mad
2|76|9|rajgir
7|20|19|gangtak
1|103|15|asam
60|83|8|bihar
23|97|7|indor
6|37|10|patna

```

output 1:  
45 5 10 mad  
3 12 4 del  
7 20 19 gangtak  
2 23 2 kol  
6 37 10 patna  
2 76 9 rajgir  
78 81 7 chen  
60 83 8 bihar  
23 97 7 indor  
1 103 15 asam

output 2:  
1 103 15 asam  
60 83 8 bihar  
78 81 7 chen  
3 12 4 del  
7 20 19 gangtak  
23 97 7 indor  
2 23 2 kol  
45 5 10 mad  
6 37 10 patna  
2 76 9 rajgir

## Algorithm

```
declare train id, distance  
no of compartment int type  
a character array for destination  
struct train  
{  
    int id,dist,noc  
    char dest[10]  
}
```

from main

**begining** and **end** are two variable

**lower** and **upper** are two stacks that hold the positions of lower and upper index respectively for the partions that are made on which the sorting is done

we use a stack for the sorting, and initialize the **top** with -1

**ob** is a struct type array that holds the values on which the sorting is to be applied and finally the sorted values are stored in this array

```

top<- -1
after sub dividing the array check if n(no. of element) greater than 1 or not
if(n>=1) then{
    top<-top+1
    lower[top]<-0
    upper[top]<-n-1
}
untill there is any element in the stack
while(top!=-1)
{
    beginning=lower[top]
    end=upper[top]
    top=top-1
    Calling the quick Sort function and return the position of the pivot element in variable
    pos
    pos=quick1(ob,beginning,end);
    if(beginning+1<pos)
    {
        top=top+1
        lower[top]=beginning
        upper[top]=pos-1
    }
    if(end-1>pos)
    {
        top=top+1
        lower[top]=pos+1
        upper[top]=end
    }
}

```

**input** is a struct type array  
**beginning** is the starting index  
**end** is the last index  
**temp** is a struct type array  
 quick1(input,beginning,end)
 {

```

        left<-beginning
        right<-end
        pos<-beginning
        do
        {
            while(input[pos].dist<=input[right].dist && pos!=right)
            {

```

```

        right--
    }
    if(pos==right) then
        return(pos)
    else
    {
        temp<-input[pos]
        input[pos]<-input[right]
        input[right]<-temp
        pos<-right
    }

    while(input[pos].dist>=input[left].dist && pos!=left)
    {
        left++
    }
    if(pos==left)
        return(pos)
    else
    {
        temp<-input[pos]
        input[pos]<-input[left]
        input[left]<-temp
        pos<-left
    }
} while(left<=right)
}

```

this algorithm we write for distance sorting, in case of destination sort we write input[pos].dest in the place of input[pos].dist.

### **Problem 3:-**

Create a structure in c that contains the following information about train:

- i)train Id
- ii)distance
- iii)train Id
- iv)destination

create a file name train.txt and add 10 train information in it manually. Write a program to sort the stored train information using quick sort (recursive) according to

- i)distance
- ii)destination

#### **Code:**

```
/*Including the header files*/

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

/*creating a structure train*/
struct train
{
    int id,dist,noc;
    char dest[10];
};

/*function for swapping*/
void swap(struct train *a,struct train *b)
{
    struct train t;
    t=*a;
    *a=*b;
    *b=t;
}

void main()
{
    //Declaing Variables and functions
    int i,n=0;
    struct train ob[10];
    int ch;
    void qk_srt1(struct train a[],int,int);
    void qk_srt2(struct train a[],int,int);
    //declaring file pointer
```

```

FILE *fp;
//open train.txt in read mode
fp=fopen("train.txt","r");
if(fp==NULL)
{
    printf("file not found");
    exit(0);
}
do
{
    //reading data from file
    fscanf(fp,"%d%d%d%s\n",&ob[n].id,&ob[n].dist,&ob[n].noc,&ob[n].dest);
    n++;
}while(!feof(fp));
//closing train.txt
fclose(fp);
do
{
    printf("\n QUICK SORT RECURSION");
    printf("\n\t1.distance sort");

    printf("\n\t2.destination sort");
    printf("\n\t3.Exit");
    printf("\n\n\tEnter Choice : - ");
    scanf("%d",&ch);
    switch(ch)
    { //Sub menu for distance wise sorting
        case 1:
            //Calling the quick Sort function
            qk_srt1(ob,0,n);
            //open test1.txt in write mode
            fp=fopen("test1.txt","w");
            if(fp == NULL)
            {
                printf("\nCannot open file");
                exit(0);
            }

            for(i=0;i<n;i++)
                { //write data into file

                    fprintf(fp,"%d %d %d
%s\n",ob[i].id,ob[i].dist,ob[i].noc,ob[i].dest);

                }
            //closing test1.txt

```

```

        fclose(fp);
        break;

        //Sub menu for destination wise sorting
case 2:
        //Calling the quick Sort function
        qk_srt2(ob,0,n);
        //open test2.txt in write mode
        fp=fopen("test2.txt","w");
        if(fp == NULL)
        {
                printf("\nCannot open file");
                exit(0);
        }
        for(i=0;i<n;i++)
        { //write data into file

                fprintf(fp,"%d %d %d
%s\n",ob[i].id,ob[i].dist,ob[i].noc,ob[i].dest);

        }
        //closing test2.txt
        fclose(fp);
        break;
case 3:
        exit(0);
default:
        printf("\nWrong Input: Re Enter");
        break;
    }
} while(1);
}

```

//quick sort function for distance sorting is defined

```
void qk_srt1(struct train a[],int m,int n)
```

```

{
    //Declaring Variables
    int i,j,k;
    //Quick sort logic
    if(m<n)
    {
        i=m;
        j=n;
        k=a[m].dist;
        do
        {
            do
            {
                i++;
            }
        }
    }
}

```

```

        }while(a[i].dist<k);
        do
        {
            j--;
        }while(a[j].dist>k);
        if(i<j)
            swap(&a[i],&a[j]);
    }while(i<j);
    swap(&a[m],&a[j]);
    qk_srt1(a,m,j-1);
    qk_srt1(a,j+1,n);
}
}

```

//quick sort function for destination sorting is defined

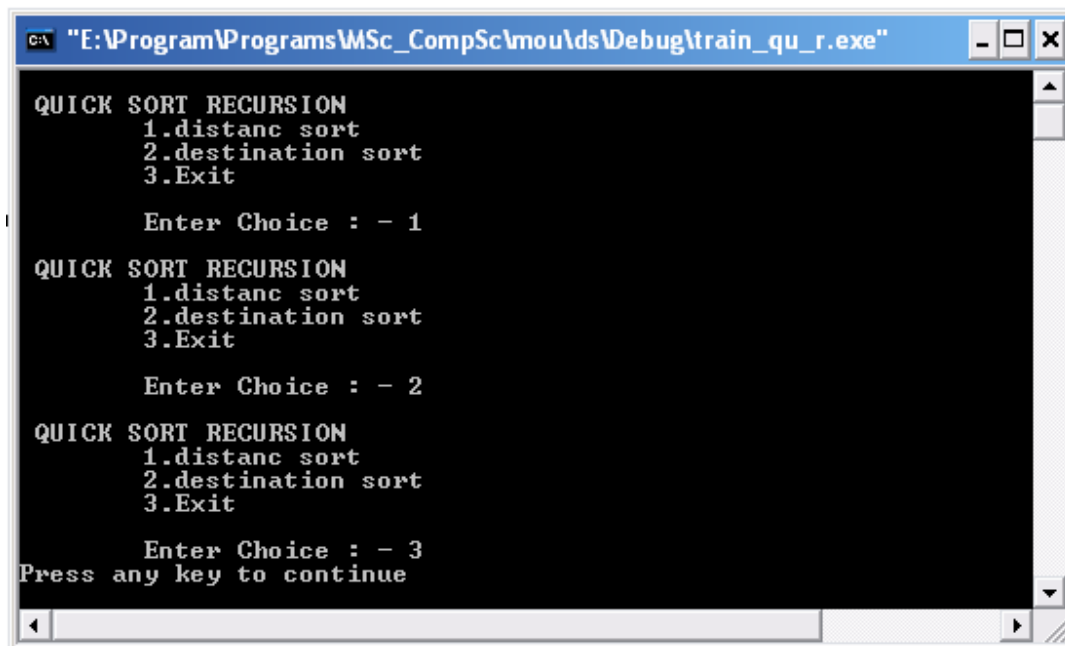
```

void qk_srt2(struct train a[],int m,int n)
{
    //Declaring Variables
    int i,j;
    char k[80];
    //Quick sort logic
    if(m<n)
    {
        i=m;
        j=n;
        strcpy(k,a[m].dest);
        do
        {
            do
            {
                i++;
            }while(a[i].dest[0]<k[0]);
            do
            {
                j--;
            }while(a[j].dest[0]>k[0]);
            if(i<j)
                swap(&a[i],&a[j]);
        }while(i<j);
        swap(&a[m],&a[j]);
        qk_srt2(a,m,j-1);
        qk_srt2(a,j+1,n);
    }
}
}

```



Output:



```
"E:\Program\Programs\WSc_CompSc\mou\ds\Debug\train_qu_r.exe"

QUICK SORT RECURSION
  1.distanc sort
  2.destination sort
  3.Exit

Enter Choice : - 1

QUICK SORT RECURSION
  1.distanc sort
  2.destination sort
  3.Exit

Enter Choice : - 2

QUICK SORT RECURSION
  1.distanc sort
  2.destination sort
  3.Exit

Enter Choice : - 3
Press any key to continue
```

Input:

2|23|2|kol  
3|12|4|del  
78|81|7|chen  
45|5|10|mad  
2|76|9|rajgir  
7|20|19|gangtak  
1|103|15|asam  
60|83|8|bihar  
23|97|7|indor  
6|37|10|patna

output 1:

45 5 10 mad  
3 12 4 del  
7 20 19 gangtak  
2 23 2 kol  
6 37 10 patna  
2 76 9 rajgir  
78 81 7 chen  
60 83 8 bihar  
23 97 7 indor  
1 103 15 asam

output 2:

1 103 15 asam

60 83 8 bihar

78 81 7 chen

3 12 4 del

7 20 19 gangtak

23 97 7 indor

2 23 2 kol

45 5 10 mad

6 37 10 patna

2 76 9 rajgir

## **Algorithm:**

declare train id distance

no of compartment int type

a character array for destination

struct train

{

int id,dist,noc

char dest[10]

}

from main

calling **quick sort** function

we pass **ob** a struct type array that holds the values on which the sorting is to be applied and finally the sorted values are stored in this array

**o** and **n** are the starting and ending index respectively

qk\_srt1(ob,o,n)

**a** is a struct type array

**m** is starting index

**n** is the ending index

qk\_srt1( a, m,n)

{

if(m<n)

then{

i<-m

j<-n

```

k<-a[m].dist
do
{
    do
    {
        i++
    }while(a[i].dist<k)
    do
    {
        j--
    }while(a[j].dist>k)
    if(i<j)
    then
        swap(&a[i],&a[j])
    }while(i<j)
    swap(&a[m],&a[j])
    qk_srt1(a,m,j-1)
    qk_srt1(a,j+1,n)
}
}

```

this algorithm we write for distance sorting, in case of destination sort we write a[m].dest in the place of a[m].dist.

the swap function is used to swap two data items of the structure type train which are passed from the quick sort

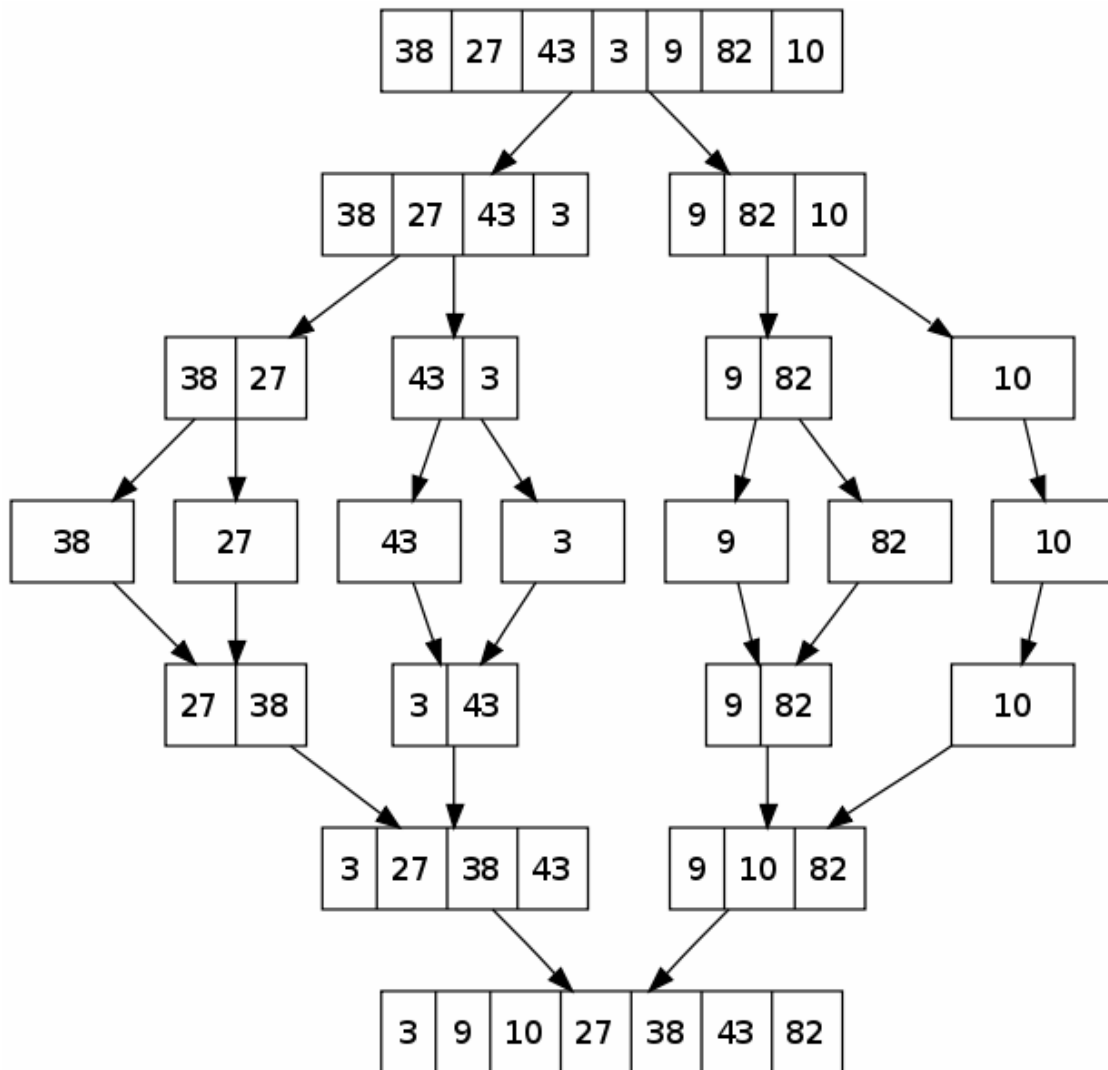
## **COMPLEXITY of QUICK SORT**

In the best case, the pivot element is in the middle.  
The best case complexity is  $O(N \log_2 N)$

The average case and the worst case complexity of quick sort is  $O(1.386 n \log_2 n)$

## Merge sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Merge sort has seen a relatively recent surge in popularity for practical



## **Problem 4**

Create a structure in c that contains the following information about train:

- i)train Id
- ii)distance
- iii)train Id
- iv)destination

create a file name train.txt and add 10 train information in it manually. Write a program to sort the stored train information using merge sort(non-recursive) according to

- i)distance
- ii)destination

## **Code:**

```
/*Including the header files*/
#include<stdio.h>
#include<stdlib.h>
#include<string.h>

/*creating a structure train*/
struct train
{
    int id,dist,noc;
    char dest[10];
};

/*function for swaping*/
void swap(struct train *a,struct train *b)
{
    struct train t;
    t=*a;
    *a=*b;
    *b=t;
}

void main()
{
    //Declaing Variables and functions
    int i,n=0;
    int ch;
    struct train ob[10];
    void mrg_srt1(struct train a[],int);
    void mrg_srt2(struct train a[],int);
    //declaring file pointer
```

```

    FILE *fp;
//open train.txt in read mode
    fp=fopen("train.txt","r");
    if(fp==NULL)
    {
        printf("file not found");
        exit(0);
    }
do
{
    //reading data from file
        fscanf(fp,"%d%d%d%d%s\n",&ob[n].id,&ob[n].dist,&ob[n].noc,&ob[n].dest);
        n++;
    }while(!feof(fp));
//closing train.txt
    fclose(fp);
do
{
    printf("\nMERGE SORT NON-RECURSION");
    printf("\n\t1.distance sort");
    printf("\n\t2.destination sort");
    printf("\n\t3.Exit");
    printf("\n\n\tEnter Choice : - ");
    scanf("%d",&ch);
    switch(ch)
    {
        //Sub menu for distance wise sorting
        case 1:
            //Calling the merge Sort function
            mrg_srt1(ob,n);
            //open test3.txt in write mode
            fp=fopen("test3.txt","w");
            if(fp == NULL)
            {
                printf("\nCannot open file");
                exit(0);
            }

            for(i=0;i<n;i++)
            {
                //write data into file

                fprintf(fp,"%d%d%d%d%s\n",ob[i].id,ob[i].dist,ob[i].noc,ob[i].dest);

            }
            //closing test3.txt
            fclose(fp);
            break;

```

```

        //Sub menu for destination wise sorting
case 2:
    //Calling the quick Sort function
    mrg_srt2(ob,n);
    //open test4.txt in write mode
    fp=fopen("test4.txt","w");
    if(fp == NULL)
    {
        printf("\nCannot open file");
        exit(0);
    }

    for(i=0;i<n;i++)
        { //write data into file

        fprintf(fp,"%d%d%d%s\n",ob[i].id,ob[i].dist,ob[i].noc,ob[i].dest);

        }
    //closing test4.txt
    fclose(fp);
    break;
case 3:
    exit(0);
default:
    printf("\nWrong Input");
    break;
    }
}while(1);
}

//merg sort function for distance sorting is defined
void mrg_srt1(struct train a[],int n)
{
    //Declaring Variables
    int la,ua,lb,ub,i,j,size,k;
    struct train aux[50];
    //Merge sort logic
    size=1;
    while(size<n)
    {
        k=0;
        la=0;
        while(la+size<=n)
        {
            lb=la+size;
            ua=lb-1;
            ub=(lb+size-1)<n?lb+size-1:n-1;

```

```

        for(i=la,j=lb;i<=ua && j<=ub;k++)
        {
            if(a[i].dist<a[j].dist)
            {
                aux[k]=a[i];
                i++;
            }
            else
            {
                aux[k]=a[j];
                j++;
            }
        }
        while(i<=ua)
        {
            aux[k]=a[i];
            k++;
            i++;
        }
        while(j<=ub)
        {
            aux[k]=a[j];
            k++;
            j++;
        }
        la=ub+1;
    }
    for(i=0;i<n;i++)
        a[i]=aux[i];
    size=size*2;
}
}

```

//merg sort function for destination sorting is defined

```

void mrg_srt2(struct train a[],int n)
{
    //Declaring Variables
    int la,ua,lb,ub,i,j,size,k;
    struct train aux[50];
    //Merge sort logic
    size=1;
    while(size<n)
    {
        k=0;
        la=0;
        while(la+size<=n)
        {
            lb=la+size;
            ua=lb-1;
            ub=(lb+size-1)<n?lb+size-1:n-1;
            for(i=la,j=lb;i<=ua && j<=ub;k++)
            {
                if(a[i].dest[0]<a[j].dest[0])
                {
                    aux[k]=a[i];
                    i++;
                }
            }
        }
    }
}

```

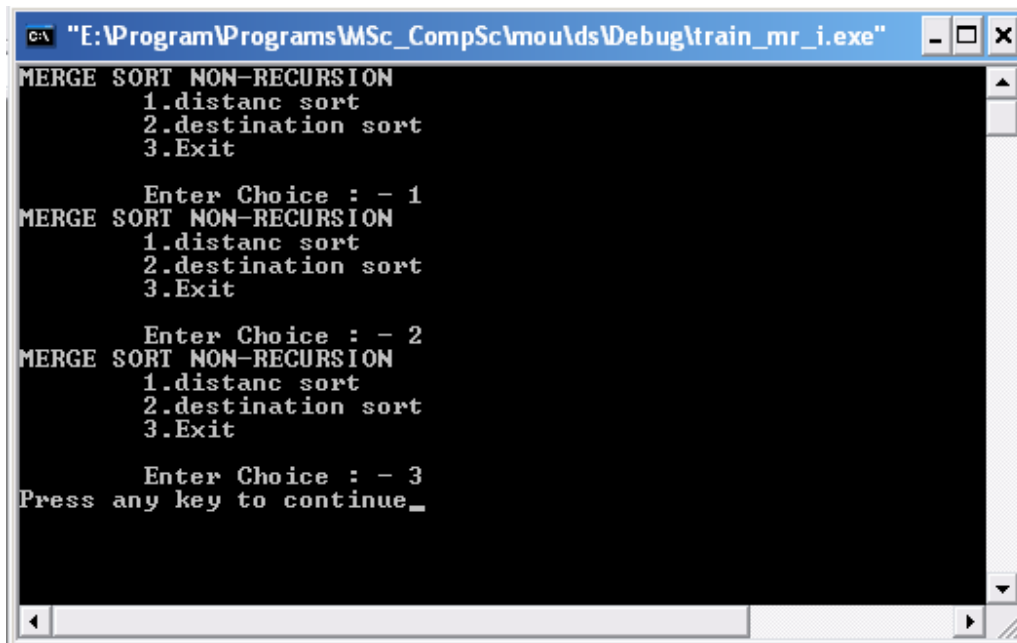


```

        }
        else
        {
            aux[k]=a[j];
            j++;
        }
    }
    while(i<=ua)
    {
        aux[k]=a[i];
        k++;
        i++;
    }
    while(j<=ub)
    {
        aux[k]=a[j];
        k++;
        j++;
    }
    la=ub+1;
}
for(i=0;i<n;i++)
    a[i]=aux[i];
size=size*2;
}
}

```

Output:



```

C:\ "E:\Program\Programs\WSc_CompSc\mou\ds\Debug\train_mr_i.exe"
MERGE SORT NON-RECURSION
1.distanc sort
2.destination sort
3.Exit
Enter Choice : - 1
MERGE SORT NON-RECURSION
1.distanc sort
2.destination sort
3.Exit
Enter Choice : - 2
MERGE SORT NON-RECURSION
1.distanc sort
2.destination sort
3.Exit
Enter Choice : - 3
Press any key to continue_

```

Input:

2|23|2|kol  
3|12|4|del  
78|81|7|chen  
45|5|10|mad  
2|76|9|rajgir  
7|20|19|gangtak  
1|103|15|asam  
60|83|8|bihar  
23|97|7|indor  
6|37|10|patna

output 1:

45 5 10 mad  
3 12 4 del  
7 20 19 gangtak  
2 23 2 kol  
6 37 10 patna  
2 76 9 rajgir  
78 81 7 chen  
60 83 8 bihar  
23 97 7 indor  
1 103 15 asam

output 2:

1 103 15 asam  
60 83 8 bihar  
78 81 7 chen  
3 12 4 del  
7 20 19 gangtak  
23 97 7 indor  
2 23 2 kol  
45 5 10 mad  
6 37 10 patna  
2 76 9 rajgir

## **Algorithm:**

```
declare train id distance
no of compartment int type
a character array for destination
struct train
{
    int id,dist,noc
```

```

        char dest[10]
    }

```

from main

Calling the **merge Sort** function

pass a struct type array **ob** that holds the values on which the sorting is to be applied and finally the sorted values are stored in this array

and **n** the no. of elements

mrq\_srt1(ob,n)

**temp1** and **aux** are two struct type array

**n** the no. of elements

**la, lb** are the starting and ending index of the lower sub array

**ua, ub** are the starting and ending index of the upper sub array

mrq\_srt1(a, n)

```

{
    size<-1
    while(size<n)
    {
        k<-0
        la<-0
        while(la+size<=n)
        {
            lb<-la+size
            ua<-lb-1
            ub<-(lb+size-1)<n?lb+size-1:n-1
            for i from la to ua & j from lb to ub
            {
                if(a[i].dist<a[j].dist) then
                    aux[k++]<-a[i++]
                else
                    aux[k++]<-a[j++]
            }
            while(i<=ua)
                aux[k++]<-a[i++]
            while(j<=ub)
                aux[k++]<-a[j++]
            la<-ub+1
        }
        for i from 0 to n
            a[i++]<-aux[i++]
        size<-size*2
    }
}

```

### **Problem 5.**

Create a structure in c that contains the following information about train:

- i)train Id
- ii)distance
- iii)train Id
- iv)destination

create a file name train.txt and add 10 train information in it manually. Write a program to sort the stored train information using merge sort(recursive) according to

- i)distance
- ii)destination

### **Code:**

```
/*Including the header files*/
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
#include<string.h>

/*Declarations of the global function for merge sort*/
void merge_sort1(struct train [ ],int,int);
void merge_sort2(struct train [ ],int,int);
void merge(struct train [],struct train [],int,int,int);
void mergee(struct train [],struct train [],int,int,int) ;
main()
{ //Declaing Variables and functions
    struct train t[20];
    int i,s=0,n=0,ch;
//declaring file pointer
    FILE *fp;
//open train.txt in read mode
    fp=fopen("train.txt","r");
    if(fp==NULL)
    {
        printf("file cannot be opened");

        exit(1);
    }
    do
    {
        //reading data from file
        fscanf(fp,"%d|%d|%d|%s\n",&t[n].train_id,&t[n].dist,&t[n].no_of_comp,&t[n].dest);
        n=n+1;
```

```

    }
    while(!feof(fp)) ;
//closing train.txt
fclose(fp);
printf("\n\n");
do
{
    printf("\n MERGE SORT RECURSION");
    printf("\n\t1.distance sort");
    printf("\n\t2.destination sort");
    printf("\n\t3.Exit");
    printf("\n\n\tEnter Choice : - ");
    scanf("%d",&ch);
    switch(ch)
    { //Sub menu for distance wise sorting
        case 1:
            //Calling the quick Sort function
            merge_sort1(t,s,n-1);
            //open test3.txt in write mode
            fp=fopen("test3.txt","w");
            if(fp == NULL)
            {
                printf("\nCannot open file");
                exit(0);
            }

            for(i=0;i<n;i++)
            //write data into file

            fprintf(fp,"%d %s %d %d\n",t[i].train_id,t[i].dest,t[i].dist,t[i].no_of_comp);
            //closing test3.txt
            fclose(fp);
            break;
        //Sub menu for destination wise sorting
        case 2:
            //Calling the quick Sort function
            merge_sort2(t,s,n-1);
            //open test4.txt in write mode
            fp=fopen("test4.txt","w");
            if(fp == NULL)
            {
                printf("\nCannot open file");
                exit(0);
            }
    }
}

```

```

                for(i=0;i<n;i++)
                    //write data into file
                fprintf(fp,"%d%s%d%d\n",t[i].train_id,t[i].dest,t[i].dist,t[i].no_of_comp);
                //closing test4.txt
                fclose(fp);
                break;

            case 3:
                exit(0);
            default:
                printf("\nWrong Input");
                break;
        }
    }while(1);
}

```

//merge sort function for distance sorting is defined

```
void merge_sort1(struct train temp1[ ],int s,int t)
```

```

{
    int m,i;
    struct train a[100];
    if(s<t)
    {
        m=(s+t)/2;
        merge_sort1(temp1,s,m);
        merge_sort1(temp1,m+1,t);
        merge(temp1,a,s,t,m);
        for(i=s;i<=t;i++)
            temp1[i]=a[i];
    }
}

```

//function merge()

```
void merge(struct train temp1[],struct train a[],int s,int t,int m)
```

```

{
    int i,j,k;
    i=s;
    j=m+1;
    k=s;
    while(i <= m && j <= t)
    {
        if(temp1[i].dist<temp1[j].dist)
        {
            a[k]=temp1[i];

```

```

        i++;
    }
    else
    {
        a[k]=temp1[j];
        j++;
    }
    k++;
}
if(i<=m)
{
    while(i<=m)
    {
        a[k]=temp1[i];
        k++;
        i++;
    }
}
else
{
    while(j<=t)
    {
        a[k]=temp1[j];
        k++;
        j++;
    }
}
}

```

//merge sort function for destination sorting is defined

```
void merge_sort2(struct train temp1[ ],int s,int t)
```

```

{
    int m,i;
    struct train a[100];
    if(s<t)
    {
        m=(s+t)/2;
        merge_sort1(temp1,s,m);
        merge_sort1(temp1,m+1,t);
        mergee(temp1,a,s,t,m);
        for(i=s;i<=t;i++)
            temp1[i]=a[i];
    }
}

```

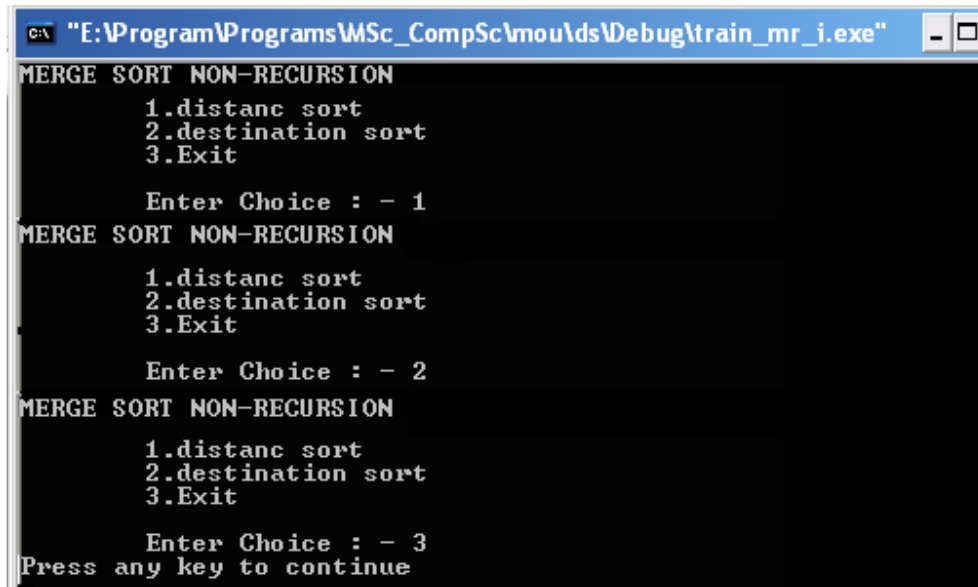
```

//function mergee()
void mergee(struct train temp1[],struct train a[],int s,int t,int m)
{
    int i,j,k;
    i=s;
    j=m+1;
    k=s;
    while(i <= m && j <= t)
    {
        if(temp1[i].dest[0]<temp1[j].dest[0])
        {
            a[k]=temp1[i];
            i++;
        }
        else
        {
            a[k]=temp1[j];
            j++;
        }
        k++;
    }
    if(i<=m)
    {
        while(i<=m)
        {
            a[k]=temp1[i];
            k++;
            i++;
        }
    }
    else
    {
        while(j<=t)
        {
            a[k]=temp1[j];
            k++;
            j++;
        }
    }
}

```



## Output:



```
C:\ "E:\Program\Programs\WSc_CompSc\mou\ds\Debug\train_mr_i.exe"
MERGE SORT NON-RECURSION
    1.distanc sort
    2.destination sort
    3.Exit

    Enter Choice : - 1
MERGE SORT NON-RECURSION
    1.distanc sort
    2.destination sort
    3.Exit

    Enter Choice : - 2
MERGE SORT NON-RECURSION
    1.distanc sort
    2.destination sort
    3.Exit

    Enter Choice : - 3
Press any key to continue
```

Input:

2|23|2|kol  
3|12|4|del  
78|81|7|chen  
45|5|10|mad  
2|76|9|rajgir  
7|20|19|gangtak  
1|103|15|asam  
60|83|8|bihar  
23|97|7|indor  
6|37|10|patna

output 1:

45 5 10 mad  
3 12 4 del  
7 20 19 gangtak  
2 23 2 kol  
6 37 10 patna  
2 76 9 rajgir  
78 81 7 chen  
60 83 8 bihar  
23 97 7 indor  
1 103 15 asam

output 2:

1 103 15 asam

60 83 8 bihar

78 81 7 chen

3 12 4 del

7 20 19 gangtak

23 97 7 indor

2 23 2 kol

45 5 10 mad

6 37 10 patna

2 76 9 rajgir

## Algorithm:

declare train id distance

no of compartment int type

a character array for destination

struct train

```
{  
    int id,dist,noc  
    char dest[10]  
}
```

from main

Calling the merge Sort function

pass a struct type array **t**

**s** and **n-1** are the starting and ending index

merge\_sort1(t,s,n-1);

**temp1** is a struct type array

**s** is the starting index

**t** is the ending index.

merge\_sort1(temp1,s,t)

```
{  
    if(s<t)  
        then{  
            m<-(s+t)/2  
            merge_sort1(temp1,s,m
```

```

        merge_sort1(temp1,m+1,t)
        merge(temp1,a,s,t,m)
        for i from s to t
            temp1[i]<-a[i]
    }
}

```

**temp1** is a temporary struct type array

**a** is an struct type array where final sorted list will be placed

**s** is the starting index

**t** is the ending index

merge(temp1,a,s,t,m)

```

{
    i<-s
    j<-m+1
    k<-s
    while(i <= m && j <= t)
    {
        if(temp1[i].dist<temp1[j].dist)
            then
                a[k++]<-temp1[i++]
        else
            a[k++]<-temp1[j++]
        if(i<=m)
            then
            {
                while(i<=m)
                    a[k++]<- temp1[i++]
            }
        else
            {
                while(j<=t)
                    a[k++]<-temp1[j++]
            }
    }
}

```

this algorithm we write for distance sorting, in case of destination sort we write temp1[i].dest in the place of temp1[i].dist.

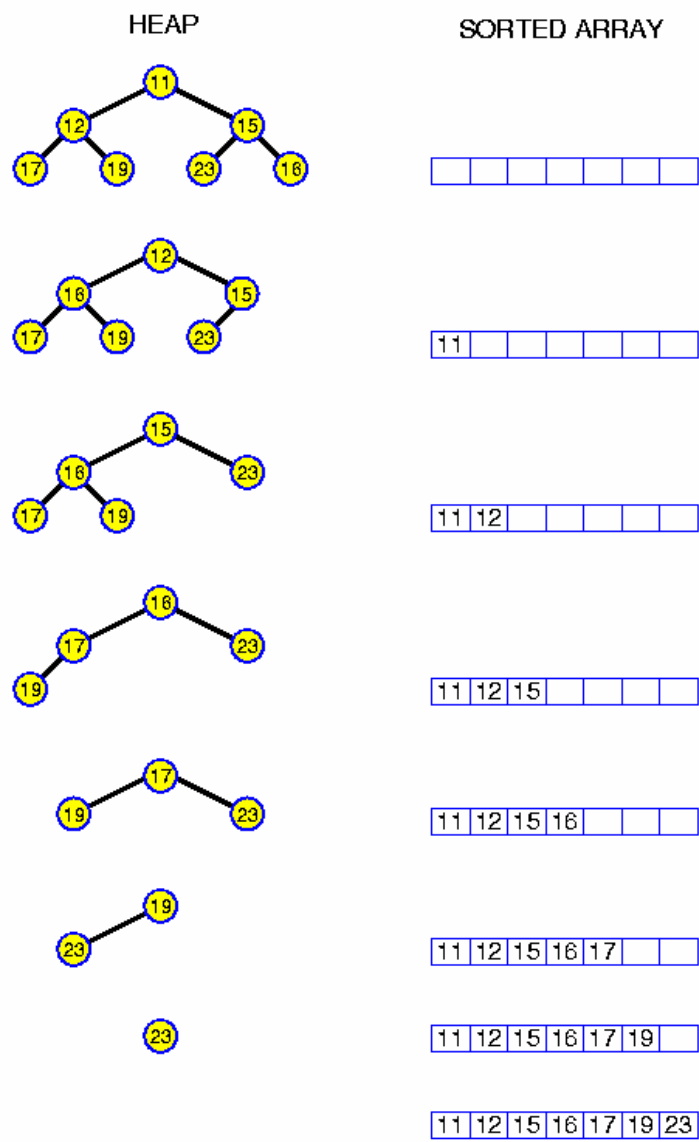
## COMPLEXITY of MERGE SORT

The complexity of merge sort is  $O(N \log_2 N)$

# Heap sort

Heap sort works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Such a way, the nodes are arranged in a non decreasing (or non increasing) order. And we get a sorted list.

## HEAPSORT



## **Problem 6**

Create a structure in c that contains the following information about train:

- i)train Id
- ii)distance
- iii)train Id
- iv)destination

create a file name train.txt and add 10 train information in it manually. Write a program to sort the stored train information using heap sort(recursive) according to

- i)distance
- ii)destination

### **Code:**

```
/*Including the header files*/
#include<stdio.h>
#include<conio.h>

/*creating a structure train*/
struct train
{
    int id,dist,noc;
    char dest[10];
};

void main()
{ //Declaing Variables and functions
    void crheap1(struct train [],int);
    void heapsort1(struct train [],int);
    void crheap2(struct train [],int);
    void heapsort2(struct train [],int);
    int i,n=0;
    int ch;
    struct train ob[10];
    //declaring file pointer
    FILE *fp;
    //open train.txt in read mode
    fp=fopen("train.txt","r");
    if(fp==NULL)
    {
        printf("file not found");
        exit(0);
    }
    do
    { //reading data from file
        fscanf(fp,"%d%d%d%s\n",&ob[n].id,&ob[n].dist,&ob[n].noc,&ob[n].dest);
```

```

        n++;
    }while(!feof(fp));
//closing train.txt
fclose(fp);
do
{
    printf("\n HEAP SORT");
    printf("\n\t1.distanc sort");
    printf("\n\t2.destination sort");
    printf("\n\t3.Exit");
    printf("\n\n\tEnter Choice : - ");
    scanf("%d",&ch);
    switch(ch)
    {
        //Sub menu for distance wise sorting
        case 1:
            crheap1(ob,n);
            //Calling the Heap Sort function
            heapsort1(ob,n);
            //open test7.txt in write mode
            fp=fopen("test7.txt","w");
            if(fp == NULL)
            {
                printf("\nCannot open file");
                exit(0);
            }
            for(i=0;i<n;i++)
            {
                //write data into file

                fprintf(fp,"%d %d %d%s\n",ob[i].id,ob[i].dist,ob[i].noc,ob[i].dest);
            }
            //closing test7.txt
            fclose(fp);
            break;

        //Sub menu for destination wise sorting
        case 2:
            crheap2(ob,n);
            //Calling the Heap Sort function
            heapsort2(ob,n);
            //open test8.txt in write mode
            fp=fopen("test8.txt","w");
            if(fp == NULL)
            {
                printf("\nCannot open file");
                exit(0);
            }
    }
}

```

```

        for(i=0;i<n;i++)
        { //write data into file

                fprintf(fp,"%d %d %d
%s\n",ob[i].id,ob[i].dist,ob[i].noc,ob[i].dest);
        }
        //closing test8.txt
        fclose(fp);
        break;
    case 3:
        exit(0);
    default:
        printf("\nWrong Input: Re Enter");
        break;
    }
} while(1);
}

```

//Heap Create function is defined  
void crheap1(struct train a[],int n)

```

{
    int i,j,k;
    struct train key;
    //Heap Create logic
    for(k=1;k<n;++k)
    {
        i=k;
        key=a[k];
        j=(int)(i/2);
        while((i>0) && (key.dist>a[j].dist))
        {
            a[i]=a[j];
            i=j;
            j=(int)(i/2);
            if(j<0)
                j=0;
        }
        a[i]=key;
    }
}

```

//Heap Sort function is defined  
void heapsort1(struct train a[],int n)

```

{
    int k,i,j;
    struct train temp,key;
    //Heap Sort logic
    for(k=n-1;k>=1;--k)
    {
        temp=a[0];
        a[0]=a[k];
    }
}

```

```

        a[k]=temp;
        i=0;
        key=a[0];
        j=1;
        if((j+1)<k)
            if(a[j+1].dist>a[j].dist)
                j=j+1;
        while((j<=(k-1)) && (a[j].dist>key.dist))
        {
            a[i]=a[j];
            i=j;
            j=2*i;
            if((j+1)<k)
            {
                if(a[j+1].dist>a[j].dist)
                    j=j+1;
                else if(j>n)
                    j=n;
            }
            a[i]=key;
        }
    }
}

```

//Heap Create function is defined  
void crheap2(struct train a[],int n)

```

{
    int i,j,k;
    struct train key;
    //Heap Create logic
    for(k=1;k<n;++k)
    {
        i=k;
        key=a[k];
        j=(int)(i/2);
        while((i>0) && (key.dest[0]>a[j].dest[0]))
        {
            a[i]=a[j];
            i=j;
            j=(int)(i/2);
            if(j<0)
                j=0;
        }
        a[i]=key;
    }
}

```

//Heap Sort function is defined  
void heapsort2(struct train a[],int n)

```

{
    int k,i,j;
    struct train temp,key;

```

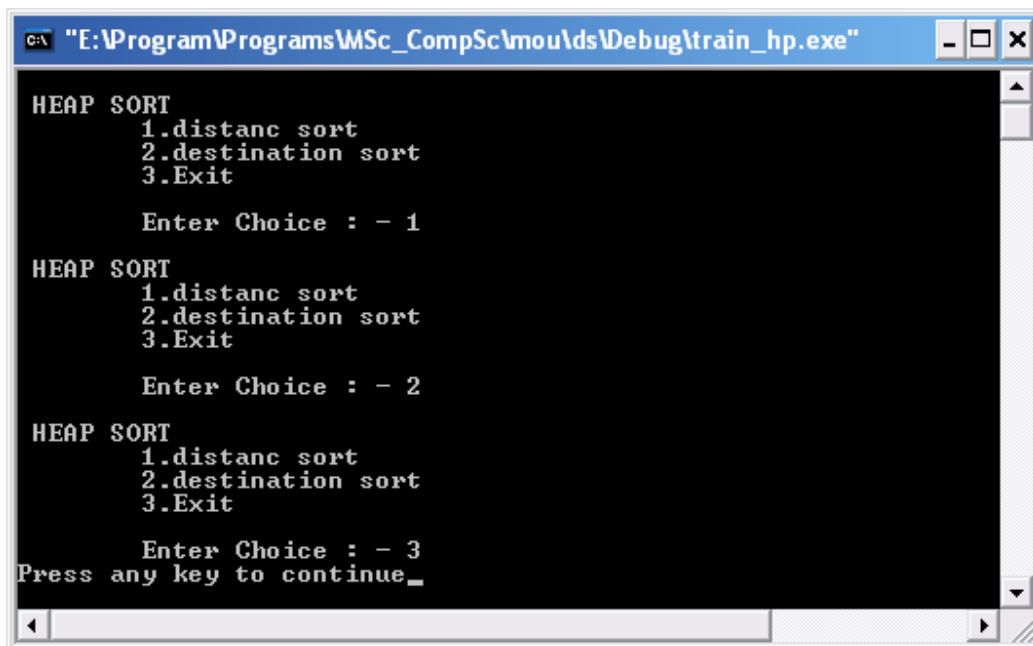


```

//Heap Sort logic
for(k=n-1;k>=1;--k)
{
    temp=a[0];
    a[0]=a[k];
    a[k]=temp;
    i=0;
    key=a[0];
    j=1;
    if((j+1)<k)
        if(a[j+1].dest[0]>a[j].dest[0])
            j=j+1;
    while((j<=(k-1)) && (a[j].dest[0]>key.dest[0]))
    {
        a[i]=a[j];
        i=j;
        j=2*i;
        if((j+1)<k)
        {
            if(a[j+1].dest[0]>a[j].dest[0])
                j=j+1;
            else if(j>n)
                j=n;
        }
        a[i]=key;
    }
}
}

```

Output:



```
"E:\Program\Programs\WSc_CompSc\mou\ds\Debug\train_hp.exe"

HEAP SORT
  1.distance sort
  2.destination sort
  3.Exit

Enter Choice : - 1

HEAP SORT
  1.distance sort
  2.destination sort
  3.Exit

Enter Choice : - 2

HEAP SORT
  1.distance sort
  2.destination sort
  3.Exit

Enter Choice : - 3
Press any key to continue_
```

Input:

2|23|2|kol  
3|12|4|del  
78|81|7|chen  
45|5|10|mad  
2|76|9|rajgir  
7|20|19|gangtak  
1|103|15|asam  
60|83|8|bihar  
23|97|7|indor  
6|37|10|patna

output 1:

45 5 10 mad  
3 12 4 del  
7 20 19 gangtak  
2 23 2 kol  
6 37 10 patna  
2 76 9 rajgir  
78 81 7 chen  
60 83 8 bihar  
23 97 7 indor  
1 103 15 asam

output 2:

1 103 15 asam

60 83 8 bihar

78 81 7 chen

3 12 4 del

7 20 19 gangtak

23 97 7 indor

2 23 2 kol

45 5 10 mad

6 37 10 patna

2 76 9 rajgir

## **Algorithm:**

from main

calling heap create function

pass a struct type array ob that holds the values on which the sorting is to be applied and finally the sorted values are stored in this array

and n(no. of element)

crheap1(ob,n)

Calling the Heap Sort function

heapsort1(ob,n)

i hold the index of the child node

j holds the index of the parent node

key holds the value of the child node

crheap(a, n)

```
{
    do
    {
        i<-k;
        key=a[k]
        j<-((i-1)/2);
        while((i>0) && (key>a[j]))
        {
            a[i]<-a[j];
            i<-j;
            j<-((i-1)/2);
            if(j<0) then
                j<-0
        }
        a[i]<-key
    }
```

```

        ++k
    }while(k<n)
}

```

Heap Sort function

pass a struct type array s,n

temp and key are another struct type array

i holds the index of the parent

j holds the index of the child

heapsort1(a,n)

```

{
    for k from n-1 to 1
    {
        temp<-a[0]
        a[0]<-a[k]
        a[k]<-temp
        i<-0
        key<-a[0]
        j<-1
        if((j+1)<k) then
            if(a[j+1].dist>a[j].dist)
                j<-j+1
        while((j<=(k-1)) && (a[j].dist>key.dist))
        {
            a[i]<-a[j]
            i<-j
            j<-2*i
            if((j+1)<k)
            {
                if(a[j+1].dist>a[j].dist)
                    j<-j+1
                else if(j>n)
                    j<-n
            }
            a[i]<-key
        }
    }
}

```

## Complexity

In heap sort program, we 1<sup>st</sup> create the heap i.e.  $O(n)$ . then we make loop that starts from last index to index 1 i.e.  $O(n)$  again. Then we swap the root with the last index i.e.  $O(1)$ . Then we again rearrange the heap which is  $O(\log n)$ .

So the complexity of the heap sort is  $O(n \log n)$ .

## **Conclusion:**

- 1) If the file is not opened in the correct mode then the data in the file maybe overwritten and this can result in loss of data. As a result the files must be opened in the correct mode.
- 2) If the files are not closed before their next access then the data maybe lost when the file is opened for next access. So the files must be close before next access.
- 3) The data stored in these files are separated by delimiters to distinguish between the different fields which are another disadvantage.