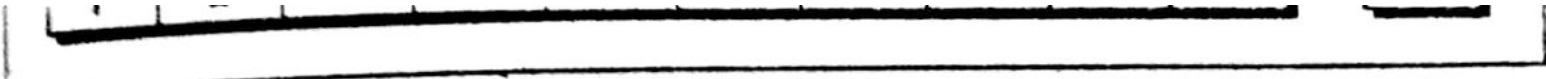

Figure 10-2. *Linear search in a sorted array.*

The following program implements linear search on a sorted array.

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

int main( )
{
    int arr[10] = { 1, 2, 3, 9, 11, 13, 17, 25, 57, 90 } ;
    int i, num ;

    system ( "cls" ) ;

    printf ( "Enter number to search: " ) ;
    scanf ( "%d", &num ) ;
```

```
    for ( i = 0 ; i <= 9 ; i++ )
    {
        if ( arr[9] < num || arr[i] >= num )
        {
            if ( arr[i] == num )
                printf ( "The number is at position %d in the array.\n", i ) ;
            else
                printf ( "Number is not present in the array.\n" ) ;
            break ;
        }
    }

    return 0 ;
}
```

Output:

```
Enter number to search: 57
The number is at position 8 in the array.
```

... number of comparisons in binary search is limited to $\log_2 n$. Following program implements binary search algorithm for a sorted array.

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

int main( )
{
    int arr[10] = { 1, 2, 3, 9, 11, 13, 17, 25, 57, 90 } ;
    int mid, lower = 0 , upper = 9, num, flag = 1 ;

    system ( "cls" ) ;

    printf ( "Enter number to search: " ) ;
    scanf ( "%d", &num ) ;

    for ( mid = ( lower + upper ) / 2 ; lower <= upper ;
        mid = ( lower + upper ) / 2 )
    {
        if ( arr[mid] == num )
        {
            printf ( "The number is at position %d in the array.\n",
                mid ) ;
            flag = 0 ;
```

```
            break ;
        }
        if ( arr[mid] > num )
            upper = mid - 1 ;
        else
            lower = mid + 1 ;
    }

    if ( flag )
        printf ( "Element is not present in the array.\n" ) ;

    return 0 ;
}
```

Output:

```
Enter number to search: 57
The number is at position 8 in the array.
```

Algorithm	Worst Case	Average Case	Best Case
Quick Sort	$O(n^2)$	$\log_2 n$	$\log_2 n$

Table 10-4. *Complexity of quick sort.*

The program given below implements the Quick sort algorithm.

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

void quicksort ( int *, int, int ) ;
int split ( int *, int, int ) ;

int main( )
{
    int arr[10] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 } ;
    int i ;

    system ( "cls" ) ;

    printf ( "Quick sort.\n\n" ) ;
    printf ( "Array before sorting:\n") ;

    for ( i = 0 ; i <= 9 ; i++ )
        printf ( "%d\t", arr[i] ) ;
```

```
    quicksort ( arr, 0, 9 ) ;

    printf ( "\n" ) ;
    printf ( "Array after sorting:\n") ;

    for ( i = 0 ; i <= 9 ; i++ )
        printf ( "%d\t", arr[i] ) ;

    return 0 ;
}

void quicksort ( int a[ ], int lower, int upper )
{
    int i ;
    if ( upper > lower )
    {
        i = split ( a, lower, upper ) ;
        quicksort ( a, lower, i - 1 ) ;
        quicksort ( a, i + 1, upper ) ;
    }
}

int split ( int a[ ], int lower, int upper )
{
    int i, p, q, t ;

    p = lower + 1 ;
    q = upper ;
    i = a[lower] ;

    while ( q >= p )
    {
        while ( a[p] < i )
            p++ ;
```

```
        if ( q > p )
        {
            t = a[p] ;
            a[p] = a[q] ;
            a[q] = t ;
        }
    }

    t = a[lower] ;
    a[lower] = a[q] ;
    a[q] = t ;

    return q ;
}
```

Output:

Quick sort.

Array before sorting:

11 2 9 13 57 25 17 1 90 3

Array after sorting:

1 2 3 9 11 13 17 25 57 90

Table 10-6. *Complexity of binary tree sort.*

The following program implements the binary tree sort algorithm.

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

struct btreenode
{
    struct btreenode *leftchild ;
    int data ;
    struct btreenode *rightchild ;
};

void insert ( struct btreenode **, int ) ;
void inorder ( struct btreenode * ) ;

int main( )
{
    struct btreenode *bt ;
    int arr[10] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 } ;
    int i ;

    bt = NULL ;
```

```
    system ( "cls" ) ;

    printf ( "Binary tree sort.\n\n" ) ;

    printf ( "Array:\n" ) ;
    for ( i = 0 ; i <= 9 ; i++ )
        printf ( "%d\t", arr[i] ) ;

    for ( i = 0 ; i <= 9 ; i++ )
        insert ( &bt, arr[i] ) ;

    printf ( "\n" ) ;
    printf ( "In-order traversal of binary tree:\n" ) ;
    inorder ( bt ) ;

    return 0 ;
}

void insert ( struct btreenode **sr, int num )
{
    if ( *sr == NULL )
    {
        *sr = ( struct btreenode * ) malloc ( sizeof ( struct btreenode ) ) ;

        ( *sr ) -> leftchild = NULL ;
        ( *sr ) -> data = num ;
        ( *sr ) -> rightchild = NULL ;
    }
    else
    {
        if ( num < ( *sr ) -> data )
            insert ( &( ( *sr ) -> leftchild ), num ) ;
        else
            insert ( &( ( *sr ) -> rightchild ), num ) ;
    }
}
```

```
void inorder ( struct btreenode *sr )
{
    if ( sr != NULL )
    {
        inorder ( sr -> leftchild ) ;
        printf ( "%d\t", sr -> data ) ;
        inorder ( sr -> rightchild ) ;
    }
}
```

Output:

Binary tree sort.

Array:

11 2 9 13 57 25 17 1 90 3

In-order traversal of binary tree:

1 2 3 9 11 13 17 25 57 90

Heap Sort

In this method, a tree structure called heap is used. A heap is type of a binary tree. An ordered balanced binary tree is called a **min-heap** where the value at the root of any sub-tree is less than or equal to the value of either of its children. An ordered balanced binary tree is called a **max-heap** when the value at the root of any sub-tree is more than or equal to the value of either of its children. It is not necessary that the two children must be in some order. Sometimes the value in left child may be more than the value at right child and some other times it may be the other way round.

Heap sort is basically an improvement over the binary tree sort. It does not create nodes as in case of binary tree sort. Instead it builds a heap by adjusting the position of elements within the array itself.

Basically, there are two phases involved in sorting the elements using heap sort algorithm. They are as follows:

(a) Construct a heap by adjusting the array elements.

The following program implements the heap sort algorithm:

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

void makeheap ( int [ ], int ) ;
void heapsort ( int [ ], int ) ;

int main( )
{
    int arr[10] = { 11, 2, 9, 13, 57, 25, 17, 1, 90, 3 } ;
    int i ;

    system ( "cls" ) ;
    printf ( "Heap Sort\n\n" ) ;
    makeheap ( arr, 10 ) ;

    printf ( "Before Sorting:\n" ) ;
    for ( i = 0 ; i <= 9 ; i++ )
        printf ( "%d\t", arr[i] ) ;

    heapsort ( arr, 10 ) ;
    printf ( "\n" ) ;
    printf ( "After Sorting:\n" ) ;
    for ( i = 0 ; i <= 9 ; i++ )
        printf ( "%d\t", arr[i] ) ;
```

```
    return 0 ;
}

/* creates heap from the tree*/
void makeheap ( int x[ ], int n )
{
    int i, val, s, f ;
    for ( i = 1 ; i < n ; i++ )
    {
        val = x[i] ;
        s = i ;
        f = ( s - 1 ) / 2 ;
        while ( s > 0 && x[f] < val )
        {
            x[s] = x[f] ;
            s = f ;
            f = ( s - 1 ) / 2 ;
        }
        x[s] = val ;
    }
}

/* sorts heap */
void heapsort ( int x[ ], int n )
{
    int i, s, f, ivalue ;
    for ( i = n - 1 ; i > 0 ; i-- )
    {
        ivalue = x[i] ;
        x[i] = x[0] ;
        f = 0 ;

        if ( i == 1 )
            s = -1 ;
        else
            s = 1 ;
```

```
        if ( i > 2 && x[2] > x[1] )
            s = 2 ;

        while ( s >= 0 && ivalue < x[s] )
        {
            x[f] = x[s] ;
            f = s ;
            s = 2 * f + 1 ;

            if ( s + 1 <= i - 1 && x[s] < x[s + 1] )
                s++ ;
            if ( s > i - 1 )
                s = -1 ;
        }
        x[f] = ivalue ;
    }
}
```

Output:

```
Heap Sort

Before Sorting:
90    57    25    13    11    9    17    1    2    3

After Sorting:
1    2    3    9    11    13    17    25    57    90
```

Table 10-8. *Complexity of merge sort.*

The following program implements the merge sort algorith

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

int main( )
{
    int a[5] = { 11, 2, 9, 13, 57 } ;
    int b[5] = { 25, 17, 1, 90, 3 } ;
    int c[10] ;
    int i, j, k, temp ;
```

```
system ( "cls" ) ;

printf ( "Merge sort:\n\n" ) ;

printf ( "First array:\n" ) ;
for ( i = 0 ; i <= 4 ; i++ )
      printf ( "%d\t", a[i] ) ;

printf ( "\n\n" ) ;
printf ( "Second array:\n" ) ;
for ( i = 0 ; i <= 4 ; i++ )
      printf ( "%d\t", b[i] ) ;

for ( i = 0 ; i <= 3 ; i++ )
{
      for ( j = i + 1 ; j <= 4 ; j++ )
      {
            if ( a[i] > a[j] )
            {
                  temp = a[i] ;
                  a[i] = a[j] ;
                  a[j] = temp ;
            }
            if ( b[i] > b[j] )
            {
                  temp = b[i] ;
                  b[i] = b[j] ;
                  b[j] = temp ;
            }
      }
}

for ( i = j = k = 0 ; i <= 9 ; )
{
      if ( a[j] <=  b[k] )
            c[i++] = a[j++] ;
      else
            c[i++] = b[k++] ;
```

```
        if ( j == 5 || k == 5 )
            break ;
    }

    for ( ; j <= 4 ; )
        c[i++] = a[j++] ;

    for ( ; k <= 4 ; )
        c[i++] = b[k++] ;

    printf ( "\n\n" ) ;
    printf ( "Array after sorting:\n" ) ;
    for ( i = 0 ; i <= 9 ; i++ )
        printf ( "%d\t", c[i] ) ;

    return 0 ;
}
```

The following program implements the external sort algorithm.

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

void shownums ( char * ) ;
void split ( char * ) ;
void sort ( char * ) ;

int main( )
{
    char str[67] ;

    system ( "cls" ) ;

    printf ( "Enter file name: " ) ;
    scanf ( "%s", str ) ;

    printf ( "Numbers before sorting:\n" ) ;
```

```
    shownums ( str ) ;

    split ( str ) ;
    sort ( str ) ;

    printf ( "\n" ) ;
    printf ( "Numbers after sorting:\n" ) ;
    shownums ( str ) ;

    return 0 ;
}

/* Displays the contents of file */
void shownums ( char *p )
{
    FILE *fp ;
    int i ;

    fp = fopen ( p, "rb" ) ;
    if ( fp == NULL )
    {
        printf ( "Unable to open file.\n" ) ;
        exit ( 0 ) ;
    }

    while ( fread ( &i, sizeof ( int ), 1, fp ) != 0 )
        printf ( "%d \t", i ) ;

    fclose ( fp ) ;
}

/* Splits the original file into two files */
void split ( char *p )
{
    FILE *fs, *ft ;
    long int l, count ;
    int i ;
```

```
fs = fopen ( p, "rb" ) ;
if ( fs == NULL )
{
     printf ( "Unable to open file.\n" ) ;
     exit ( 0 ) ;
}

ft = fopen ( "temp1.dat", "wb" ) ;
if ( ft == NULL )
{
     fclose ( fs ) ;
     printf ( "Unable to open file.\n" ) ;
     exit ( 1 ) ;
 }

 fseek ( fs, 0L, SEEK_END ) ;
 l = ftell ( fs ) ;
 fseek ( fs, 0L, SEEK_SET ) ;

 l = l / ( sizeof ( int ) * 2 ) ;

 for ( count = 1 ; count <= l ; count++ )
 {
     fread ( &i, sizeof ( int ), 1, fs ) ;
     fwrite ( &i, sizeof ( int ), 1, ft ) ;
 }

 fclose ( ft ) ;

 ft = fopen ( "temp2.dat", "wb" ) ;
 if ( ft == NULL )
 {
     fclose ( fs ) ;
     printf ( "Unable to open file.\n" ) ;
     exit ( 2 ) ;
 }

 while (   fread ( &i, sizeof ( int ), 1, fs ) != 0 )
```

```
            fwrite ( &i, sizeof ( int ), 1, ft ) ;

        fcloseall ( ) ;
}

/* Sorts the file */
void sort ( char *p )
{
        FILE *fp[4] ;
        char *fnames[ ] =
                                {
                                    "temp1.dat",
                                    "temp2.dat",
                                    "final1.dat",
                                    "final2.dat"
                                } ;

        int i, j = 1, i1, i2, flag1, flag2, p1, p2 ;
        long int l ;

        while ( 1 )
        {
            for ( i = 0 ; i <= 1 ; i++ )
            {
                fp[i] = fopen ( fnames[i], "rb+" ) ;
                if ( fp[i] == NULL )
                {
                    fcloseall( ) ;
                    printf ( "Unable to open file.\n" ) ;
                    exit ( i ) ;
                }

                fseek ( fp[i], 0L, SEEK_END ) ;
                l = ftell ( fp[i] ) ;
                if ( l == 0 )
                    goto out ;
                fseek ( fp[i], 0L, SEEK_SET ) ;

            }
```

```
for ( i = 2 ; i <= 3 ; i++ )
{
    fp[i] = fopen ( fnames[i], "wb" ) ;
    if ( fp[i] == NULL )
    {
        fcloseall( ) ;
        printf ( "Unable to open file.\n" ) ;
        exit ( i ) ;
    }
}

i = 2 ;
i1 = i2 = 0 ;
flag1 = flag2 = 1 ;

while ( 1 )
{
    if ( flag1 )
    {
        if ( fread ( &p1, sizeof ( int ), 1, fp[0] ) == 0 )
        {
            /* If first file ends then the whole content of
               second file is written in the respective target file */
            while ( fread ( &p2, sizeof ( int ), 1, fp[1] ) !=
                0 )
                fwrite ( &p2, sizeof ( int ), 1, fp[i] ) ;
            break ;
        }
    }

    if ( flag2 )
    {
        if ( fread ( &p2, sizeof ( int ), 1, fp[1] ) == 0 )
        {
            /* If second file ends then the whole content
               of first file is written in the respective target  file */
            fwrite ( &p1, sizeof ( int ), 1, fp[i] ) ;
```

```
            while ( fread ( &p1, sizeof ( int ), 1, fp[0] ) !=
                0 )
                fwrite ( &p1, sizeof ( int ), 1, fp[i] ) ;
            break ;
        }
    }

    if ( p1 < p2 )
    {
        flag2 = 0 ;
        flag1 = 1 ;
        fwrite ( &p1, sizeof ( int ), 1, fp[i] ) ;
        i1++ ;
    }
    else
    {
        flag1 = 0 ;
        flag2 = 1 ;
        fwrite ( &p2, sizeof ( int ), 1, fp[i] ) ;
        i2++ ;
    }

    if ( i1 == j )
    {
        flag1 = flag2 = 1 ;
        fwrite ( &p2, sizeof ( int ), 1, fp[i] ) ;
        for ( i2++ ; i2 < j ; i2++ )
        {
            if ( fread ( &p2, sizeof ( int ), 1, fp[1] ) != 0 )
                fwrite ( &p2, sizeof ( int ), 1, fp[i] ) ;
        }
        i1 = i2 = 0 ;
        i == 2 ? ( i = 3 ) : ( i = 2 ) ;
    }

    if ( i2 == j )
    {
        flag1 = flag2 = 1 ;
```

```
                fwrite ( &p1, sizeof ( int ), 1, fp[i] ) ;
                for ( i1++ ; i1 < j ; i1++ )
                {
                    if ( fread ( &p1, sizeof ( int ), 1, fp[0] ) != 0 )
                        fwrite ( &p1, sizeof ( int ), 1, fp[i] ) ;
                }
                i1 = i2 = 0 ;
                i == 2 ? ( i = 3 ) : ( i = 2 ) ;
            }
        }

        fcloseall( ) ;
        remove ( fnames[0] ) ;
        remove ( fnames[1] ) ;
        rename ( fnames[2], fnames[0] ) ;
        rename ( fnames[3], fnames[1] ) ;
        j *= 2 ;
    }

    out :

    fcloseall( ) ;
    remove ( p ) ;
    rename ( fnames[0], p ) ;
    remove ( fnames[1] ) ;
}
```