Searching and Sorting

Searching is used to find the location where an element is available. There are two types of search techniques. They are:

Linear or sequential search

Binary search

Sorting allows an efficient arrangement of elements within a given data structure. It is a way in which the elements are organized systematically for some purpose. For example, a dictionary in which words is arranged in alphabetical order and telephone director in which the subscriber names are listed in alphabetical order. There are many sorting techniques out of which we study the following.

Bubble sort

Quick sort

Selection sort and

Heap sort

There are two types of sorting techniques:

Internal sorting

External sorting

If all the elements to be sorted are present in the main memory then such sorting is called **internal sorting** on the other hand, if some of the elements to be sorted are kept on the secondary storage, it is called **external sorting**. Here we study only internal sorting techniques.

Linear Search:

This is the simplest of all searching techniques. In this technique, an ordered or unordered list will be searched one by one from the beginning until the desired element is found. If the desired element is found in the list then the search is successful otherwise unsuccessful.

Suppose there are 'n' elements organized sequentially on a List. The number of comparisons required to retrieve an element from the list, purely depends on where the element is stored in the list. If it is the first element, one comparison will do; if it is second element two comparisons are necessary and so on. On an average you need [(n+1)/2] comparison's to search an element. If search is not successful, you would need 'n' comparisons.

The time complexity of linear search is ***O(n)***.

Algorithm:

Let array a[n] stores n elements. Determine whether element 'x' is present or not.

```
linsrch(a[n], x)
{
        index = 0;
        flag = 0;
        while (index < n) do
        {
                if (x == a[index])
                {
                        flag = 1;
                        break;
```

```
            }
            index ++;
        }
        if(flag == 1)
            printf("Data found at %d position", index);
        else
            printf("data not found");

}
```

Example 1:

Suppose we have the following unsorted list: 45, 39, 8, 54, 77, 38, 24, 16, 4, 7, 9, 20

If we are searching for:

45, we'll look at 1 element before success
39, we'll look at 2 elements before success
8, we'll look at 3 elements before success
54, we'll look at 4 elements before success
77, we'll look at 5 elements before success
38 we'll look at 6 elements before success
24, we'll look at 7 elements before success
16, we'll look at 8 elements before success
4, we'll look at 9 elements before success
7, we'll look at 10 elements before success
9, we'll look at 11 elements before success
20, we'll look at 12 elements before success

For any element not in the list, we'll look at 12 elements before failure.

Example 2:

Let us illustrate linear search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Searching different elements is as follows:

Searching for x = 7Search successful, data found at 3rd position.

Searching for x = 82Search successful, data found at 7th position.

Searching for x = 42Search un-successful, data not found.

A non-recursive program for Linear Search:

```
include <stdio.h>
include <conio.h>
main()
{
    int number[25], n, data, i, flag = 0;
    clrscr();
    printf("\n Enter the number of elements: ");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for(i = 0; i < n; i++)
        scanf("%d", &number[i]);
    printf("\n Enter the element to be Searched: ");
```

```
        scanf("%d", &data);
        for( i = 0; i < n; i++)
        {
                if(number[i] == data)
                {
                        flag = 1;
                        break;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", i+1);
        else
                printf("\n Data not found ");
}
```

A Recursive program for linear search:

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

void linear_search(int a[], int data, int position, int n)
{
        if(position < n)
        {
                if(a[position] == data)
                        printf("\n Data Found at %d ", position);
                else
                        linear_search(a, data, position + 1, n);
        }
        else
                printf("\n Data not found");
}
void main()
{
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements: ");
        for(i = 0; i < n; i++)
        {
                scanf("%d", &a[i]);
        }
        printf("\n Enter the element to be seached: ");
        scanf("%d", &data);
        linear_search(a, data, 0, n);
        getch();
}
```

BINARY SEARCH

If we have 'n' records which have been ordered by keys so that $x_1 < x_2 < \ldots < x_n$. When we are given a element 'x', binary search is used to find the corresponding element from the list. In case 'x' is present, we have to determine a value 'j' such that a[j] = x (successful search). If 'x' is not in the list then j is to set to zero (un successful search).

In Binary search we jump into the middle of the file, where we find key a[mid], and compare 'x' with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then 'x' must be in that portion of the file that precedes a[mid]. Similarly, if

a[mid] > x, then further search is only necessary in that part of the file which follows a[mid].

If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of 'x' with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved after each comparison between 'x' and a[mid], and since an array of length 'n' can be halved only about $\log_2 n$ times before reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$.

Algorithm:

Let array a[n] of elements in increasing order, n ≥ 0, determine whether 'x' is present, and if so, set j such that x = a[j] else return 0.

```
binsrch(a[], n, x)
{
        low = 1; high = n;
        while (low ≤ high) do
        {
                mid =   (low + high)/2
                if (x < a[mid])
                        high = mid – 1;
                else if (x > a[mid])
                        low = mid + 1;
                else return mid;
        }
        return 0;
 }
```

low and *high* are integer variables such that each time through the loop either 'x' is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if 'x' is not present.

Example 1:

Let us illustrate binary search on the following 12 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*
Elements	4	7	8	9	16	20	24	38	39	45	54	77

If we are searching for x = 4: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4, ***found***

If we are searching for x = 7: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 1, high = 2, mid = 3/2 = 1, check 4
low = 2, high = 2, mid = 4/2 = 2, check 7, ***found***

If we are searching for x = 8: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8, ***found***

If we are searching for x = 9: (This needs 3 comparisons)

low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9, ***found***

If we are searching for x = 16: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 1, high = 5, mid = 6/2 = 3, check 8
low = 4, high = 5, mid = 9/2 = 4, check 9
low = 5, high = 5, mid = 10/2 = 5, check 16, ***found***

If we are searching for x = 20: (This needs 1 comparison)
low = 1, high = 12, mid = 13/2 = 6, check 20, ***found***

If we are searching for x = 24: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24, ***found***

If we are searching for x = 38: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 7, high = 8, mid = 15/2 = 7, check 24
low = 8, high = 8, mid = 16/2 = 8, check 38, ***found***

If we are searching for x = 39: (This needs 2 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39, ***found***

If we are searching for x = 45: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 10, high = 10, mid = 20/2 = 10, check 45, ***found***

If we are searching for x = 54: (This needs 3 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54, ***found***

If we are searching for x = 77: (This needs 4 comparisons)
low = 1, high = 12, mid = 13/2 = 6, check 20
low = 7, high = 12, mid = 19/2 = 9, check 39
low = 10, high = 12, mid = 22/2 = 11, check 54
low = 12, high = 12, mid = 24/2 = 12, check 77, ***found***

The number of comparisons necessary by search element:

20 – requires 1 comparison;
8 and 39 – requires 2 comparisons;
4, 9, 24, 54 – requires 3 comparisons and
7, 16, 38, 45, 77 – requires 4 comparisons

Summing the comparisons, needed to find all twelve items and dividing by 12, yielding 37/12 or approximately 3.08 comparisons per successful search on the average.

Example 2:

Let us illustrate binary search on the following 9 elements:

Index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*
Elements	-15	-6	0	7	9	23	54	82	101

Solution:

The number of comparisons required for searching different elements is as follows:

1. If we are searching for x = 101: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
8	9	8
9	9	9
		found

2. Searching for x = 82: (Number of comparisons = 3)

low	high	mid
1	9	5
6	9	7
8	9	8
		found

3. Searching for x = 42: (Number of comparisons = 4)

low	high	mid
1	9	5
6	9	7
6	6	6
	6	not found

Searching for x = -14: (Number of comparisons = 3)

low	high	mid
1	9	5
1	4	2
1	1	1
2	1	not found

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-15	-6	0	7	9	23	54	82	101
Comparisons	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

If x < a(1), a(1) < x < a(2), a(2) < x < a(3), a(5) < x < a(6), a(6) < x < a(7) or a(7) x < a(8) the algorithm requires 3 element comparisons to determine that 'x' is not present. For all of the remaining possibilities BINSRCH requires 4 element comparisons.

Thus the average number of element comparisons for an unsuccessful search is:

(3 + 3 + 3 + 4 + 4 + 3 + 3 + 3 + 4 + 4) / 10 = 34/10 = 3.4

Time Complexity:

The time complexity of binary search in a successful search is O(log n) and for an unsuccessful search is O(log n).

A non-recursive program for binary search:

```
  include <stdio.h>
  include <conio.h>
main()
{
        int number[25], n, data, i, flag = 0, low, high, mid;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements in ascending order: ");
        for(i = 0; i < n; i++)
                scanf("%d", &number[i]);
        printf("\n Enter the element to be searched: ");
        scanf("%d", &data);
        low = 0; high = n-1;
        while(low <= high)
        {
                mid = (low + high)/2;
                if(number[mid] == data)
                {
                        flag = 1;
                        break;
                }
                else
                {
                        if(data < number[mid])
                                high = mid - 1;
                        else
                                low = mid + 1;
                }
        }
        if(flag == 1)
                printf("\n Data found at location: %d", mid + 1);
        else
                printf("\n Data Not Found ");
}
```

A recursive program for binary search:

```
  include <stdio.h>
  include <conio.h>
  void bin_search(int a[], int data, int low, int high)
{
        int mid ;
        if( low <= high)
        {
                mid = (low + high)/2;
                if(a[mid] == data)
                        printf("\n Element found at location: %d ", mid + 1);
                else
                {
                        if(data < a[mid])
                                bin_search(a, data, low, mid-1);
```

```
                else
                        bin_search(a, data, mid+1, high);
                }
        }
        else
                printf("\n Element not found");
}
void main()
{
        int a[25], i, n, data;
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter the elements in ascending order: ");
        for(i = 0; i < n; i++)
                scanf("%d", &a[i]);
        printf("\n Enter the element to be searched: ");
        scanf("%d", &data);
        bin_search(a, data, 0, n-1);
        getch();
}
```

Bubble Sort:

The bubble sort is easy to understand and program. The basic idea of bubble sort is to pass through the file sequentially several times. In each pass, we compare each element in the file with its successor i.e., X[i] with X[i+1] and interchange two element when they are not in proper order. We will illustrate this sorting technique by taking a specific example. Bubble sort is also called as exchange sort.

Example:

Consider the array x[n] which is stored in memory as shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]
33	44	22	11	66	55

Suppose we want our array to be stored in ascending order. Then we pass through the array 5 times as described below:

Pass 1: (first element is compared with all other elements).

We compare X[i] and X[i+1] for i = 0, 1, 2, 3, and 4, and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	X[5]	Remarks
33	44	22	11	66	55	
	22	44				
		11	44			
			44	66		
				55	66	
33	22	11	44	55	66	

The biggest number 66 is moved to (bubbled up) the right most position in the array.

Pass 2: (second element is compared).

We repeat the same process, but this time we don't include X[5] into our comparisons. i.e., we compare X[i] with X[i+1] for i=0, 1, 2, and 3 and interchange X[i] and X[i+1] if X[i] > X[i+1]. The process is shown below:

X[0]	X[1]	X[2]	X[3]	X[4]	Remarks
33	22	11	44	55	
22	33				
	11	33			
		33	44		
			44	55	
22	11	33	44	55	

The second biggest number 55 is moved now to X[4].

Pass 3: (third element is compared).

We repeat the same process, but this time we leave both X[4] and X[5]. By doing this, we move the third biggest number 44 to X[3].

X[0]	X[1]	X[2]	X[3]	Remarks
22	11	33	44	
11	22			
	22	33		
		33	44	
11	22	33	44	

Pass 4: (fourth element is compared).

We repeat the process leaving X[3], X[4], and X[5]. By doing this, we move the fourth biggest number 33 to X[2].

X[0]	X[1]	X[2]	Remarks
11	22	33	
11	22		
	22	33	

Pass 5: (fifth element is compared).

We repeat the process leaving X[2], X[3], X[4], and X[5]. By doing this, we move the fifth biggest number 22 to X[1]. At this time, we will have the smallest number 11 in X[0]. Thus, we see that we can sort the array of size 6 in 5 passes.

For an array of size n, we required (n-1) passes.

Program for Bubble Sort:

```
#include <stdio.h>
#include <conio.h>
void bubblesort(int x[], int n)
```

```
{
        int i, j, temp;
        for (i = 0; i < n; i++)
        {
                for (j = 0; j < n–i-1 ; j++)
                {
                        if (x[j] > x[j+1])
                        {
                                temp = x[j];
                                x[j] = x[j+1];
                                x[j+1] = temp;
                        }
                }
        }
}
main()
{
        int i, n, x[25];
        clrscr();
        printf("\n Enter the number of elements: ");
        scanf("%d", &n);
        printf("\n Enter Data:");
        for(i = 0; i < n ; i++)
                scanf("%d", &x[i]);
        bubblesort(x, n);
        printf ("\n Array Elements after sorting: ");
        for (i = 0; i < n; i++)
                printf ("%5d", x[i]);
}
```

Time Complexity:

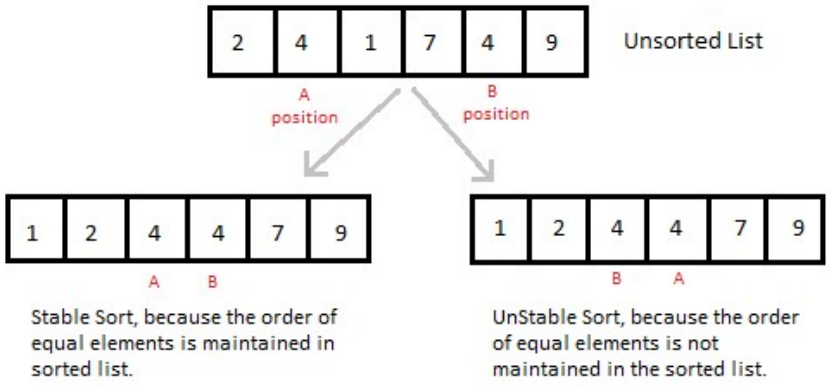
The bubble sort method of sorting an array of size n requires (n-1) passes and (n-1) comparisons on each pass. Thus the total number of comparisons is (n-1) * (n-1) = n^2 – 2n + 1, which is O(n^2). Therefore bubble sort is very inefficient when there are more elements to sorting.

Insertion Sort

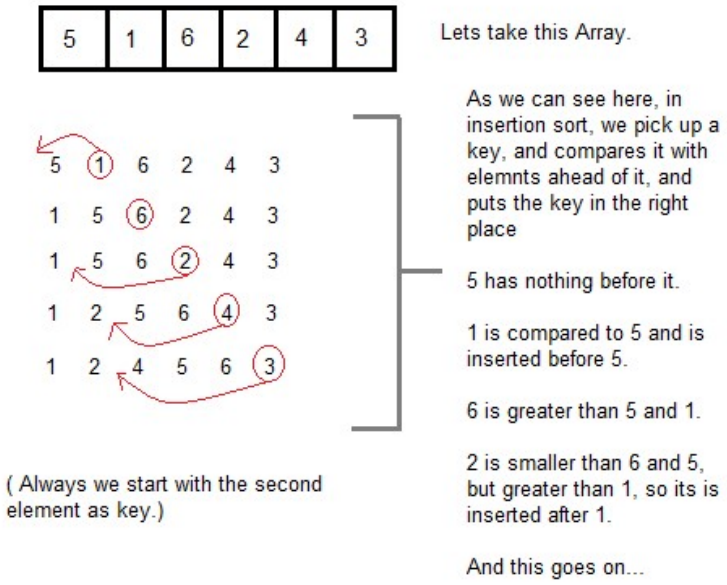
It is a simple Sorting algorithm which sorts the array by shifting elements one by one.

Following are some of the important characteristics of Insertion Sort.

1. It has one of the simplest implementation
2. It is efficient for smaller data sets, but very inefficient for larger lists.
3. Insertion Sort is adaptive, that means it reduces its total number of steps if given a partially sorted list, hence it increases its efficiency.
4. It is better than Selection Sort and Bubble Sort algorithms.
5. Its space complexity is less. Like Bubble Sorting, insertion sort also requires a single additional memory space.
6. It is a **Stable** sorting, as it does not change the relative order of elements with equal keys



How Insertion Sorting Works



Sorting using Insertion Sort Algorithm

```
int a[6] = {5, 1, 6, 2, 4, 3};
int i, j, key;
for(i=1; i<6; i++)
{
  key = a[i];
  j = i-1;
  while(j>=0 && key < a[j])
  {
    a[j+1] = a[j];
    j--;
  }
  a[j+1] = key;
}
```

Now lets, understand the above simple insertion sort algorithm. We took an array with 6 integers. We took a variable **key**, in which we put each element of the array, in each pass, starting from the second element, that is **a[1]**.

Then using the while loop, we iterate, until **j** becomes equal to zero or we find an element which is greater than **key**, and then we insert the key at that position.

In the above array, first we pick 1 as key, we compare it with 5(element before 1), 1 is smaller than 5, we shift 1 before 5. Then we pick 6, and compare it with 5 and 1, no shifting this time. Then 2 becomes the key and is compared with, 6 and 5, and then 2 is placed after 1. And this goes on, until complete array gets sorted.

Complexity Analysis of Insertion Sorting

Worst Case Time Complexity : O(n2)

Best Case Time Complexity : O(n)

Average Time Complexity : O(n2)

Space Complexity : O(1)

Selection Sort:

Selection sort will not require no more than n-1 interchanges. Suppose x is an array of size n stored in memory. The selection sort algorithm first selects the smallest element in the array x and place it at array position 0; then it selects the next smallest element in the array x and place it at array position 1. It simply continues this procedure until it places the biggest element in the last position of the array.

The array is passed through (n-1) times and the smallest element is placed in its respective position in the array as detailed below:

Pass 1: Find the location j of the smallest element in the array x [0], x[1], ... x[n-1], and then interchange x[j] with x[0]. Then x[0] is sorted.

Pass 2: Leave the first element and find the location j of the smallest element in the sub-array x[1], x[2], ... x[n-1], and then interchange x[1] with x[j]. Then x[0], x[1] are sorted.

Pass 3: Leave the first two elements and find the location j of the smallest element in the sub-array x[2], x[3], ... x[n-1], and then interchange x[2] with x[j]. Then x[0], x[1], x[2] are sorted.

Pass (n-1): Find the location j of the smaller of the elements x[n-2] and x[n-1], and then interchange x[j] and x[n-2]. Then x[0], x[1], ... x[n-2] are sorted. Of course, during this pass x[n-1] will be the biggest element and so the entire array is sorted.

Time Complexity:

In general we prefer selection sort in case where the insertion sort or the bubble sort requires exclusive swapping. In spite of superiority of the selection sort over bubble

sort and the insertion sort (there is significant decrease in run time), its efficiency is also $O(n^2)$ for n data items.

Example:

Let us consider the following example with 9 elements to analyze selection Sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*Remarks*
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	**50**	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	**50**	**55**	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	**50**	**55**	**60**	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	85	80	Find the eighth smallest element
							i	J	swap a[i] and a[j]
45	**50**	**55**	**60**	**65**	**70**	**75**	**80**	**85**	The outer loop ends.

Non-recursive Program for selection sort:

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

void selectionSort( int low, int high );

int a[25];

int main()
```

```
{
        int num, i= 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf("%d", &num);
        printf( "\nEnter the elements:\n" );
        for(i=0; i < num; i++)
                scanf( "%d", &a[i] );
        selectionSort( 0, num - 1 );
        printf( "\nThe elements after sorting are: " );
        for( i=0; i< num; i++ )
                printf( "%d    ", a[i] );
        return 0;
}

void selectionSort( int low, int high )
{
        int i=0, j=0, temp=0, minindex;
        for( i=low; i <= high; i++ )
        {
                minindex = i;
                for( j=i+1; j <= high; j++ )
                {
                        if( a[j] < a[minindex] )
                                minindex = j;
                }
                temp = a[i];
                a[i] = a[minindex];
                a[minindex] = temp;
        }
}
```

Recursive Program for selection sort:

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

int x[6] = {77, 33, 44, 11, 66};
selectionSort(int);

main()
{
        int i, n = 0;
        clrscr();
        printf (" Array Elements before sorting: ");
        for (i=0; i<5; i++)


                printf ("%d ", x[i]);
        selectionSort(n);                       /* call selection sort */
        printf ("\n Array Elements after sorting: ");
        for (i=0; i<5; i++)
                printf ("%d ", x[i]);
}

selectionSort( int n)
{
        int k, p, temp, min;
        if (n== 4)
                return (-1);
```

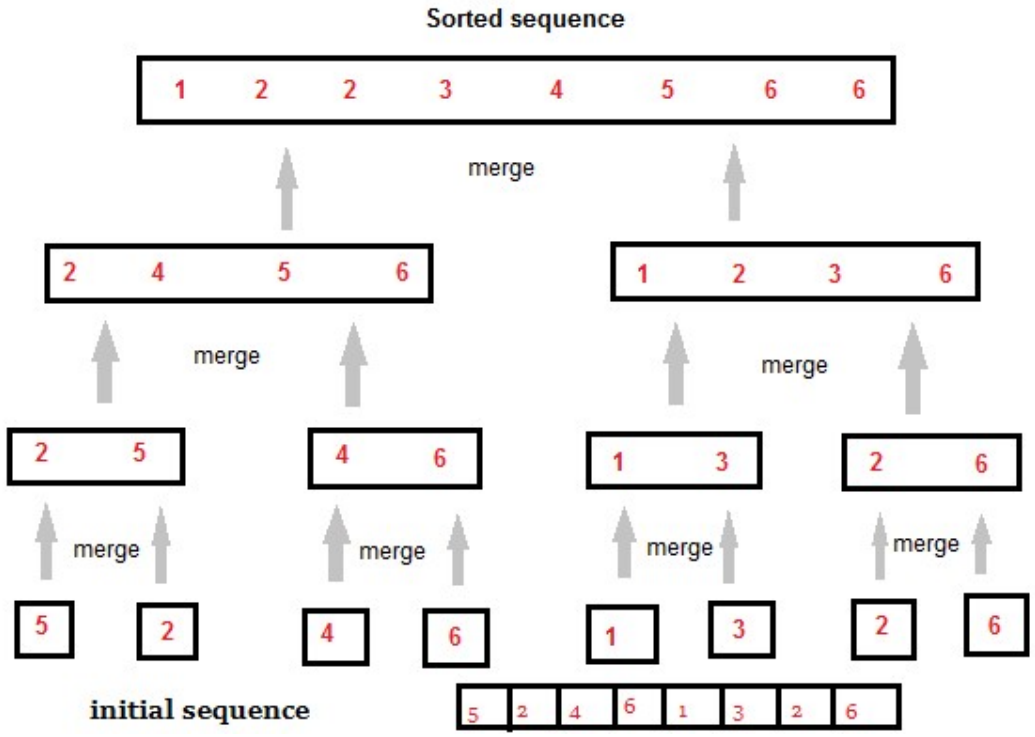
```
        min = x[n];
        p = n;
        for (k = n+1; k<5; k++)
        {
                if (x[k] <min)
                {
                        min = x[k];
                        p = k;
                }
        }
        temp = x[n];          /* interchange x[n] and x[p] */
        x[n] = x[p];
        x[p] = temp;
        n++ ;
        selectionSort(n);
}
```

Merge Sort

Merge Sort follows the rule of **Divide and Conquer**. In merge sort the unsorted list is divided into N sublists, each having one element, because a list consisting of one element is always sorted. Then, it repeatedly merges these sublists, to produce new sorted sublists, and in the end, only one sorted list is produced.

Merge Sort is quite fast, and has a time complexity of **O(n log n)**. It is also a stable sort, which means the "equal" elements are ordered in the same order in the sorted list.

How Merge Sort Works



Like we can see in the above example, merge sort first breaks the unsorted list into

sorted sublists, each having one element, because a list of one element is considered sorted and then it keeps merging these sublists, to finally get the complete sorted list.

Sorting using Merge Sort Algorithm

```
/*  a[] is the array, p is starting index, that is 0, and r is the last index of array.  */
//Lets take a[5] = {32, 45, 67, 2, 7} as the array to be sorted.
void mergesort(int a[], int p, int r)
{
  int q;
  if(p < r)
  {
    q = floor( (p+r) / 2);
    mergesort(a, p, q);
    mergesort(a, q+1, r);
    merge(a, p, q, r);
  }
}
void merge(int a[], int p, int q, int r)
{
  int b[5];     //same size of a[]
  int i, j, k;
  k = 0;
  i = p;
  j = q+1;
  while(i <= q && j <= r)
  {
    if(a[i] < a[j])
    {
      b[k++] = a[i++];       // same as b[k]=a[i]; k++; i++;
    }
    else
    {
      b[k++] = a[j++];
    }
  }
  while(i <= q)
  {
    b[k++] = a[i++];
```

```
 }
 while(j <= r)
 {
  b[k++] = a[j++];
 }
  for(i=r; i >= p; i--)
 {
  a[i] = b[--k];       // copying back the sorted list to a[]
 }
}
```

Complexity Analysis of Merge Sort

Worst Case Time Complexity : O(n log n)

Best Case Time Complexity : O(n log n)

Average Time Complexity : O(n log n)

Space Complexity : O(n)

- Time complexity of Merge Sort is O(n Log n) in all 3 cases (worst, average and best) as merge sort always divides the array in two halves and take linear time to merge two halves.
- It requires equal amount of additional space as the unsorted list. Hence its not at all recommended for searching large unsorted lists.
- It is the best Sorting technique used for sorting **Linked Lists**.

Quick Sort:

The quick sort was invented by Prof. C. A. R. Hoare in the early 1960's. It was one of the first most efficient sorting algorithms. It is an example of a class of algorithms that work by "divide and conquer" technique.

The quick sort algorithm partitions the original array by rearranging it into two groups. The first group contains those elements less than some arbitrary chosen value taken from the set, and the second group contains those elements greater than or equal to the chosen value. The chosen value is known as the *pivot* element. Once the array has been rearranged in this way with respect to the *pivot*, the same partitioning procedure is recursively applied to each of the two subsets. When all the subsets have been partitioned and rearranged, the original array is sorted.

The function partition() makes use of two pointers up and down which are moved toward each other in the following fashion:

Repeatedly increase the pointer 'up' until a[up] >= pivot.

Repeatedly decrease the pointer 'down' until a[down] <= pivot.

If down > up, interchange a[down] with a[up]

Repeat the steps 1, 2 and 3 till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and place pivot element in 'down' pointer position.

The program uses a recursive function quicksort(). The algorithm of quick sort function sorts all elements in an array 'a' between positions 'low' and 'high'.

It terminates when the condition low >= high is satisfied. This condition will be satisfied only when the array is completely sorted.

Here we choose the first element as the 'pivot'. So, pivot = x[low]. Now it calls the partition function to find the proper position j of the element x[low] i.e. pivot. Then we will have two sub-arrays x[low], x[low+1], ... x[j-1] and x[j+1], x[j+2], . . . x[high].

It calls itself recursively to sort the left sub-array x[low], x[low+1], ... x[j-1] between positions low and j-1 (where j is returned by the partition function).

It calls itself recursively to sort the right sub-array x[j+1], x[j+2], . . x[high] between positions j+1 and high.

The time complexity of quick sort algorithm is of ***O(n log n)***.

Algorithm

Sorts the elements a[p], ... ,a[q] which reside in the global array a[n] into ascending order. The a[n + 1] is considered to be defined and must be greater than all elements in a[n]; a[n + 1] = + ∝

```
quicksort (p, q)
{
      if ( p < q ) then
      {
           call j = PARTITION(a, p, q+1);     // j is the position of the partitioning
           element
           call quicksort(p, j – 1);
           call quicksort(j + 1 , q);
       }
}

partition(a, m, p)
{
                                              // a[m] is the partition
       v = a[m]; up = m; down = p;            element
       do
       {
              repeat
                     up = up + 1;
              until (a[up] ≥ v);
              repeat
                     down = down – 1;
              until (a[down] ≤ v);
              if (up < down) then call interchange(a, up,
        down); } while (up ≥ down);
       a[m] = a[down];
       a[down] = v;
       return (down);
}
```

```
interchange(a, up, down)
{
        p = a[up];
        a[up] = a[down];
        a[down] = p;
}
```

Example:

Select first element as the pivot element. Move 'up' pointer from left to right in search of an element larger than pivot. Move the 'down' pointer from right to left in search of an element smaller than pivot. If such elements are found, the elements are swapped.

This process continues till the 'up' pointer crosses the 'down' pointer. If 'up' pointer crosses 'down' pointer, the position for pivot is found and interchange pivot and element at 'down' position.

Let us consider the following example with 13 elements to analyze quick sort:

1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*	*12*	*13*	*Remarks*
38	08	16	06	79	57	24	56	02	58	04	70	45	
pivot				up						down			swap up & down
pivot				04						79			
pivot					up			down					swap up & down
pivot					02			57					
pivot						down	up						swap pivot & down
(24	08	16	06	04	02)	**38**	(56	57	58	79	70	45)	
pivot					down	up							swap pivot & down
(02	08	16	06	04)	**24**								
pivot, down	up												swap pivot & down
02	(08	16	06	04)									
	pivot	up		down									swap up & down
	pivot	04		16									
	pivot		down	Up									
	(06	04)	**08**	(16)									swap pivot

													& down
	pivot	down	up										
	(04)	**06**											swap pivot & down
	04 pivot, down , up												
				16 pivot, down , up									
(02	**04**	**06**	**08**	**16**	**24)**	38							

							(56	57	58	79	70	45**)**	
							pivot	up				down	swap up & down
							pivot	45				57	
							pivot	down	up				swap pivot & down
							(45)	**56**	(58	79	70	57)	
							45 pivot, down, up						swap pivot & down
									(58 pivot	79 up	70	57) down	swap up & down
										57		79	
										down	up		
									(57)	**58**	(70	79)	swap pivot & down
									57 pivot, down , up				
											(70	79)	
											pivot ,	up	swap pivot

											down		& down
											70		
												79 pivot, down, up	
							(45	**56**	**57**	**58**	**70**	**79)**	
02	**04**	**06**	**08**	**16**	**24**	**38**	**45**	**56**	**57**	**58**	**70**	**79**	

Recursive program for Quick Sort:

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

void quicksort(int, int);
int partition(int, int);
void interchange(int, int);
int array[25];

int main()
{
        int num, i = 0;
        clrscr();
        printf( "Enter the number of elements: " );
        scanf( "%d", &num);
        printf( "Enter the elements: " );
        for(i=0; i < num; i++)
                scanf( "%d", &array[i] );
        quicksort(0, num -1);
        printf( "\nThe elements after sorting are: " );
        for(i=0; i < num; i++)
                printf("%d ", array[i]);
        return 0;
}

void quicksort(int low, int high)
{
        int pivotpos;
        if( low < high )
        {
                pivotpos = partition(low, high + 1);
                quicksort(low, pivotpos - 1);
                quicksort(pivotpos + 1, high);
        }
}

int partition(int low, int high)
{
        int pivot = array[low];
        int up = low, down = high;

        do
        {
                do
```

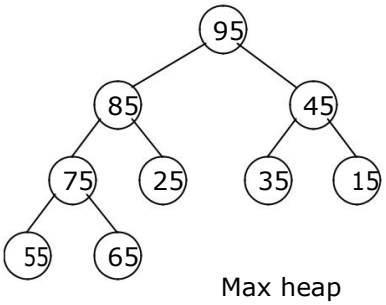
```
                up = up + 1;
            while(array[up] < pivot );

            do
                down = down - 1;
            while(array[down] > pivot);

            if(up < down)
                interchange(up, down);

        } while(up < down);
        array[low] = array[down];
        array[down] = pivot;
        return down;
}

void interchange(int i, int j)
{
        int temp;
        temp = array[i];
        array[i] = array[j];
        array[j] = temp;
}
```

Priority Queue, Heap and Heap Sort:

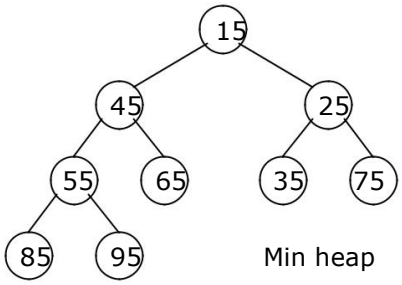
Heap is a data structure, which permits one to insert elements into a set and also to find the largest element efficiently. A data structure, which provides these two operations, is called a priority queue.

Max and Min Heap data structures:

A max heap is an almost complete binary tree such that the value of each node is greater than or equal to those in its children.



Max heap



Min heap

A min heap is an almost complete binary tree such that the value of each node is less than or equal to those in its children.

Representation of Heap Tree:

Since heap is a complete binary tree, a heap tree can be efficiently represented using one dimensional array. This provides a very convenient way of figuring out where children belong to.

The root of the tree is in location 1.

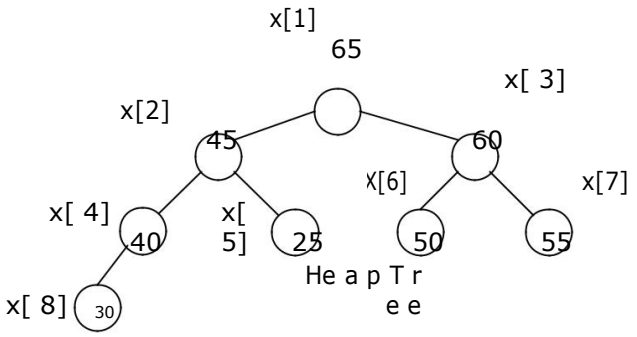
The left child of an element stored at location *i* can be found in location 2**i*.

The right child of an element stored at location *i* can be found in location 2**i*+1.

The parent of an element stored at location *i* can be found at location floor(*i*/2).

The elements of the array can be thought of as lying in a tree structure. A heap tree represented using a single array looks as follows:

X[1]	X[2]	X[3]	X[4]	X[5]	X[6]	X[7]	X[8]
65	45	60	40	25	50	55	30



Operations on heap tree:

The major operations required to be performed on a heap tree:

Insertion,

Deletion and

Merging.

Insertion into a heap tree:

This operation is used to insert a node into an existing heap tree satisfying the properties of heap tree. Using repeated insertions of data, starting from an empty heap tree, one can build up a heap tree.

Let us consider the heap (max) tree. The principle of insertion is that, first we have to adjoin the data in the complete binary tree. Next, we have to compare it with the data in its parent; if the value is greater than that at parent then interchange the values. This will continue between two nodes on path from the newly inserted node to the root node till we get a parent whose value is greater than its child or we reached the root.

For illustration, 35 is added as the right child of 80. Its value is compared with its parent's value, and to be a max heap, parent's value greater than child's value is satisfied, hence interchange as well as further comparisons are no more required.

As another illustration, let us consider the case of insertion 90 into the resultant heap tree. First, 90 will be added as left child of 40, when 90 is compared with 40 it requires interchange. Next, 90 is compared with 80, another interchange takes place. Now, our process stops here, as 90 is now in root node. The path on which these comparisons and interchanges have taken places are shown by *dashed line*.

The algorithm Max_heap_insert to insert a data into a max heap tree is as follows:

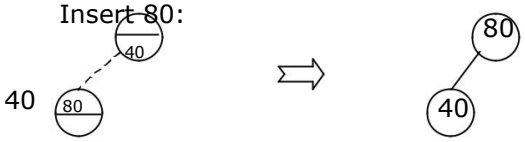
```
Max_heap_insert (a, n)
{
        //inserts the value in a[n] into the heap which is stored at a[1] to a[n-1]
        int i, n;
        i = n;
        item = a[n];
        while ( (i > 1) and (a[   i/2   ] < item ) do
        {
                a[i] = a[   i/2   ] ;                          // move the parent down
                i =   i/2    ;
        }
        a[i] = item ;
        return true ;
}
```

Example:

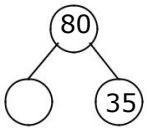
Form a heap using the above algorithm for the data: 40, 80, 35, 90, 45, 50, 70.

Insert 40:

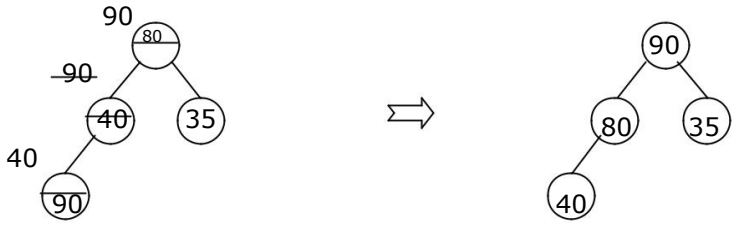
Insert 80:



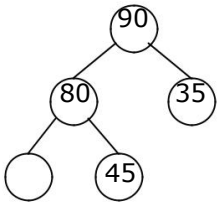
Insert 35:



Insert 90:

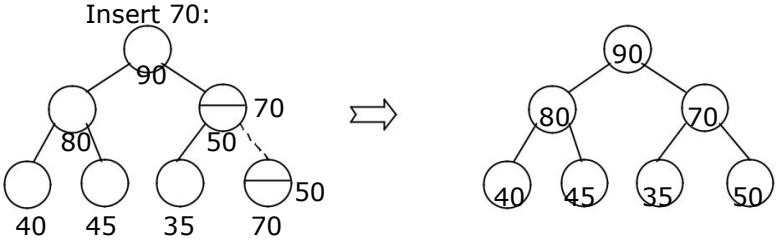


Insert 45:



Insert 50:

Insert 70:



Deletion of a node from heap tree:

Any node can be deleted from a heap tree. But from the application point of view, deleting the root node has some special importance. The principle of deletion is as follows:

Read the root node into a temporary storage say, ITEM.

Replace the root node by the last node in the heap tree. Then re-heap the tree as stated below:

Let newly modified root node be the current node. Compare its value with the value of its two child. Let X be the child whose value is the largest. Interchange the value of X with the value of the current node.

Make X as the current node.

Continue re-heap, if the current node is not an empty node.

The algorithm for the above is as follows:

delmax (a, n, x)

delete the maximum from the heap a[n] and store it in x

```
{
        if (n = 0) then
        {
                write ("heap is empty");
                return false;
        }
        x = a[1]; a[1] = a[n];
        adjust (a, 1, n-1);
        return true;
}
```

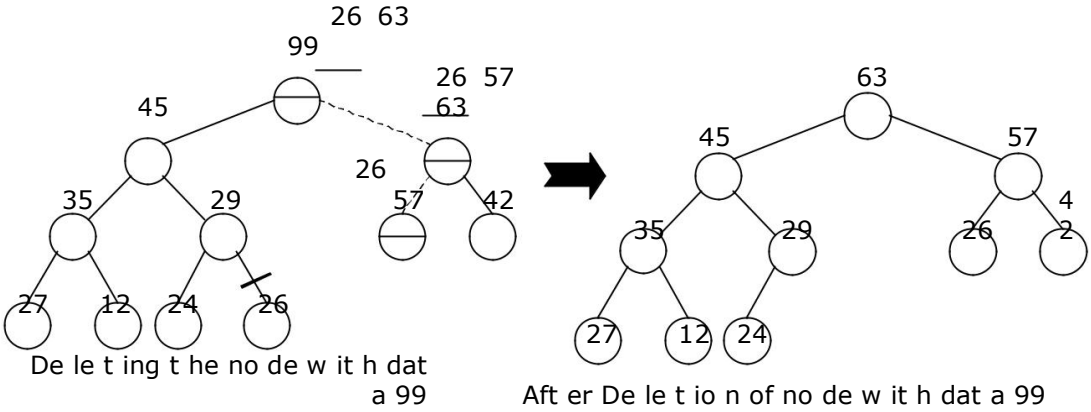
adjust (a, i, n)

The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to form a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //

```
{
        j = 2 *i ; item
        = a[i] ; while (j
        ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j □ j + 1;
                                compare left and right child and let j be the larger
                child if (item ≥ a (j)) then break;
                                                        a position for item is found
                else a[   j / 2   ] = a[j]        // move the larger child up a level
                j = 2 * j;
        }
        a [   j / 2   ] = item;
}
```

Here the root node is 99. The last node is 26, it is in the level 3. So, 99 is replaced by 26 and this node with data 26 is removed from the tree. Next 26 at root node is compared with its two child 45 and 63. As 63 is greater, they are interchanged. Now,

26 is compared with its children, namely, 57 and 42, as 57 is greater, so they are interchanged. Now, 26 appears as the leave node, hence re-heap is completed.



De le t ing t he no de w it h dat a 99

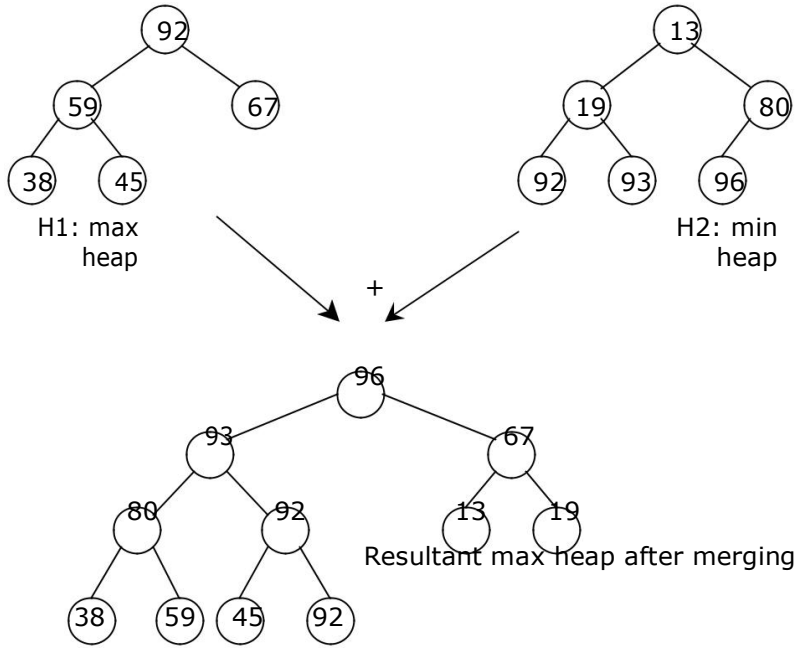
Aft er De le t io n of no de w it h dat a 99

Merging two heap trees:

Consider, two heap trees H1 and H2. Merging the tree H2 with H1 means to include all the node from H2 to H1. H2 may be min heap or max heap and the resultant tree will be min heap if H1 is min heap else it will be max heap. Merging operation consists of two steps: Continue steps 1 and 2 while H2 is not empty:

Delete the root node, say x, from H2. Re-heap H2.

Insert the node x into H1 satisfying the property of H1.



H1: max heap

H2: min heap

Resultant max heap after merging H1 and H2

Application of heap tree:

They are two main applications of heap trees known are:

Sorting (Heap sort) and

Priority queue implementation.

HEAP SORT:

A heap sort algorithm works by first organizing the data to be sorted into a special type of binary tree called a heap. Any kind of data can be sorted either in ascending order or in descending order using heap tree. It does this with the following steps:

Build a heap tree with the given set of data.

a.Remove the top most item (the largest) and replace it with the last

element in the heap.

Re-heapify the complete binary tree.

Place the deleted node in the output.

Continue step 2 until the heap tree is empty.

Algorithm:

This algorithm sorts the elements a[n]. Heap sort rearranges them in-place in non-decreasing order. First transform the elements into a heap.

heapsort(a, n)

```
{
        heapify(a, n);
        for i = n to 2 by – 1 do
        {
                temp = a[i];
                a[i] = a[1];
                a[1] = temp;
                adjust (a, 1, i – 1);
        }
}
```

heapify (a, n)

```
//Readjust the elements in a[n] to form a heap.
{
        for i □   n/2   to 1 by – 1 do adjust (a, i, n);
}
```

adjust (a, i, n)

The complete binary trees with roots a(2*i) and a(2*i + 1) are combined with a(i) to form a single heap, 1 ≤ i ≤ n. No node has an address greater than n or less than 1. //

```
{
        j = 2 *i ; item
        = a[i] ; while (j
        ≤ n) do
        {
                if ((j < n) and (a (j) < a (j + 1)) then j □ j + 1;
                                compare left and right child and let j be the larger
                child if (item ≥ a (j)) then break;
                                                        a position for item is found
                else a[   j / 2   ] = a[j]        // move the larger child up a level
                j = 2 * j;
        }
        a [   j / 2   ] = item;
}
```

Time Complexity:

Each 'n' insertion operations takes O(log k), where 'k' is the number of elements in the heap at the time. Likewise, each of the 'n' remove operations also runs in time O(log k), where 'k' is the number of elements in the heap at the time.

Since we always have k ≤ n, each such operation runs in O(log n) time in the worst case.

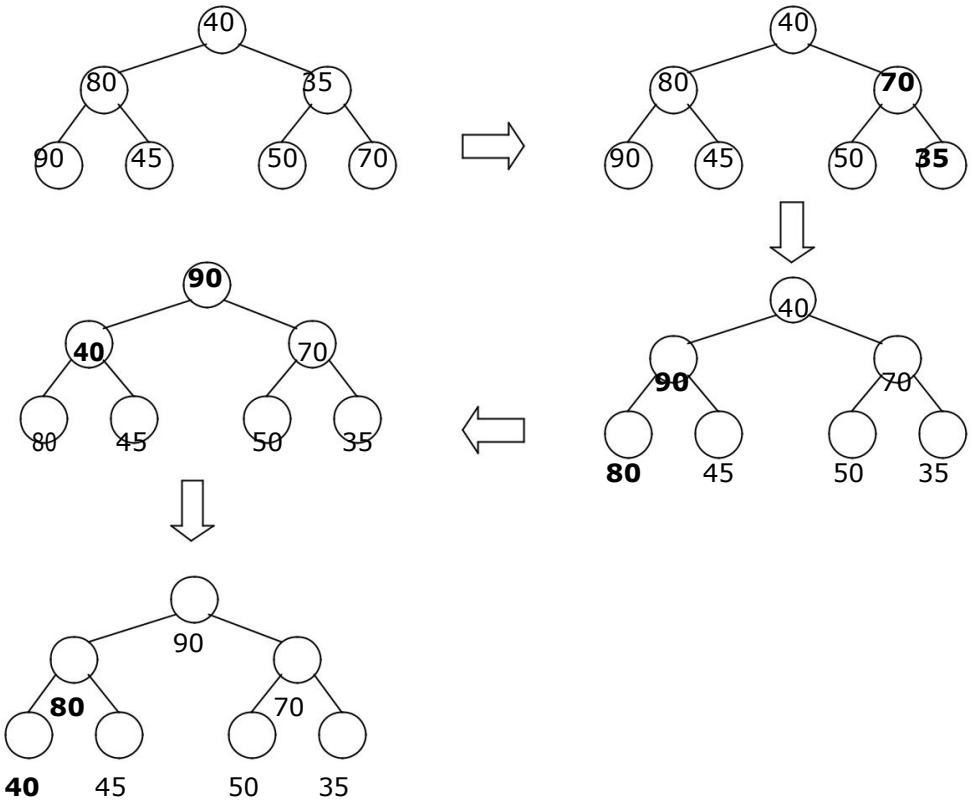
Thus, for 'n' elements it takes O(n log n) time, so the priority queue sorting algorithm runs in O(n log n) time when we use a heap to implement the priority queue.

Example 1:

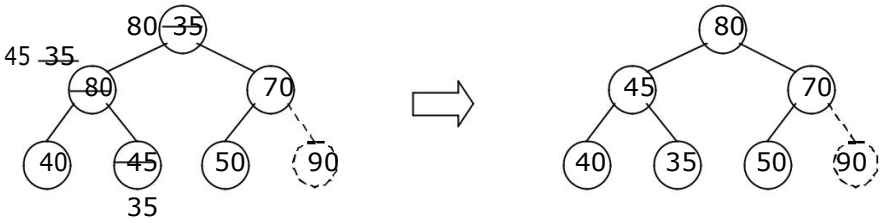
Form a heap from the set of elements (40, 80, 35, 90, 45, 50, 70) and sort the data using heap sort.

Solution:

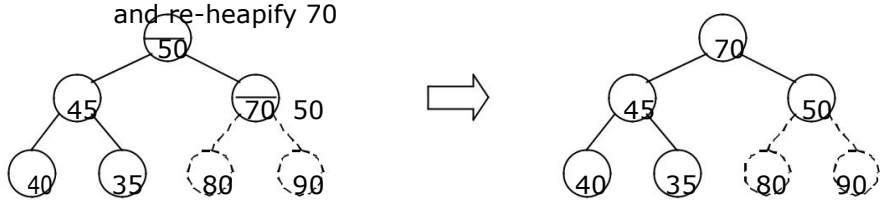
First form a heap tree from the given set of data and then sort by repeated deletion operation:



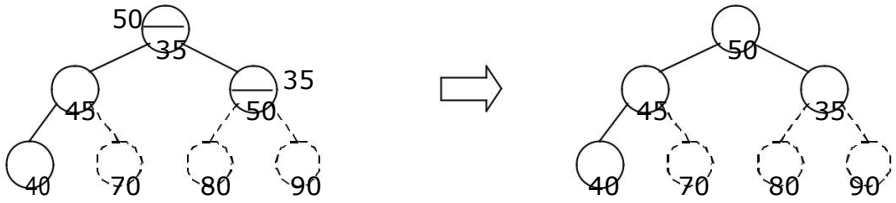
Exchange root 90 with the last element 35 of the array and re-heapify



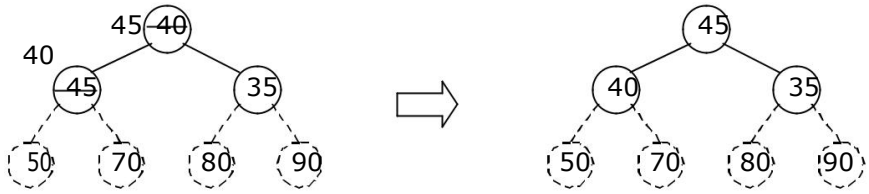
Exchange root 80 with the last element 50 of the array and re-heapify 70



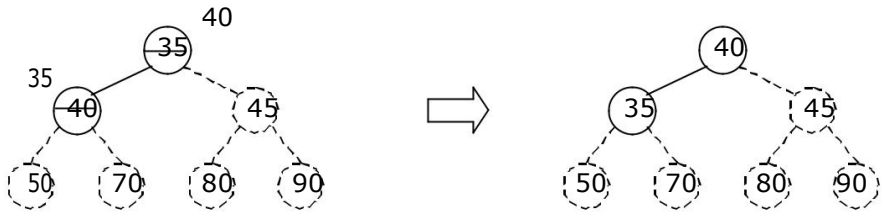
Exchange root 70 with the last element 35 of the array and re-heapify



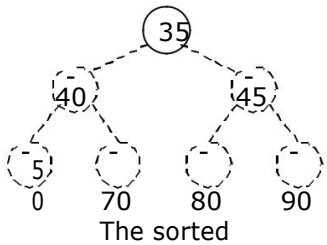
Exchange root 50 with the last element 40 of the array and re-heapify



Exchange root 45 with the last element 35 of the array and re-heapify



Exchange root 40 with the last element 35 of the array and re-heapify



The sorted tree

Program for Heap Sort:

```
void adjust(int i, int n, int a[])
{
        int j, item;
        j = 2 * i;
        item = a[i];
        while(j <= n)
        {
                if((j < n) && (a[j] < a[j+1]))
                        j++;
                if(item >= a[j])
                        break;
                else
                {
                        a[j/2] = a[j];
                        j = 2*j;
                }
        }
        a[j/2] = item;
}

void heapify(int n, int a[])
{
        int i;
        for(i = n/2; i > 0; i--)
                adjust(i, n, a);
}

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

void main()
{
        int i, n, a[20];
        clrscr();
        printf("\n How many element you want: ");
        scanf("%d", &n);
        printf("Enter %d elements: ", n);
        for (i=1; i<=n; i++)
                scanf("%d", &a[i]);
        heapsort(n, a);
        printf("\n The sorted elements are: \n");
        for (i=1; i<=n; i++)
                printf("%5d", a[i]);
        getch();
}
```

Priority queue implementation using heap tree:

Priority queue can be implemented using circular array, linked list etc. Another simplified implementation is possible using heap tree; the heap, however, can be represented using an array. This implementation is therefore free from the complexities of circular array and linked list but getting the advantages of simplicities of array.

As heap trees allow the duplicity of data in it. Elements associated with their priority values are to be stored in from of heap tree, which can be formed based on their priority values. The top priority element that has to be processed first is at the root; so it can be deleted and heap can be rebuilt to get the next element to be processed, and so on. As an illustration, consider the following processes with their priorities:

Process	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P10
Priority	5	4	3	4	5	5	3	2	1	5

These processes enter the system in the order as listed above at time 0, say. Assume that a process having higher priority value will be serviced first. The heap tree can be formed considering the process priority values. The order of servicing the process is successive deletion of roots from the heap.