UNIT –IV

SEARCHING:

Search is a process of finding a value in a list of values. In other words, searching is the process of locating given value position in a list of values.

Linear Search (Sequential Search):

Linear search algorithm finds given element in a list of elements with O(n) time complexity where n is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list. Linear search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Compare, the search element with the first element in the list.
- **Step 3:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 4:** If both are not matching, then compare search element with the next element in the list.
- **Step 5:** Repeat steps 3 and 4 until the search element is compared with the last element in the list.
- **Step 6:** If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function.

Example

Consider the following list of element and search element...

Step 1:

search element (12) is compared with first element (65)

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)

Both are matching. So we stop comparing and display element found at index 5.

Linear Search Program

```
#include<iostream>
using namespace std;
int  main()
{   int list[]={10,20,40,30,50};
    int size,i,s;
    size= sizeof(list)/sizeof(list[0]);
    cout<<"Enter the element to be Search";
    cin>>s;
  for(i = 0; i < size; i++)
  {
     if(s == list[i])
     {
        cout<<"Element is found at posdition"<<i+1;
        break;
     }
  }
  if(i == size)
     cout<<"Given element is not found in the list!!!";
 }
```

Binary Search:

Binary search algorithm finds given element in a list of elements with O(log n) time complexity where n is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in a order. The binary search can not be used for list of element which are in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sublist of the middle element. If the search element is larger, then we repeat the same process for right sublist of the middle element. We repeat this process until we find the search element in the list or until we left with a sublist of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list".

Binary search is implemented using following steps...

- **Step 1:** Read the search element from the user
- **Step 2:** Find the middle element in the sorted list
- **Step 3:** Compare, the search element with the middle element in the sorted list.
- **Step 4:** If both are matching, then display "Given element found!!!" and terminate the function
- **Step 5:** If both are not matching, then check whether the search element is smaller or larger than middle element.
- **Step 6:** If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sublist of the middle element.
- **Step 7:** If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sublist of the middle element.
- **Step 8:** Repeat the same process until we find the search element in the list or until sublist contains only one element.
- **Step 9:** If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.

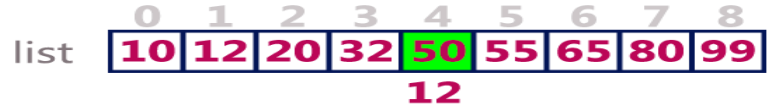
Example

Consider the following list of element and search element...

search element **12**

Step 1:

search element (12) is compared with middle element (50)



Both are not matching. And 12 is smaller than 50. So we search only in the left sublist (i.e. 10, 12, 20 & 32).

Step 2:

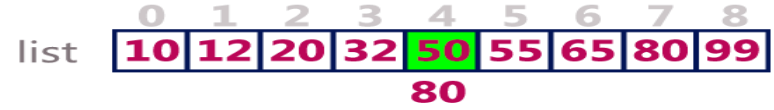
search element (12) is compared with middle element (12)

Both are matching. So the result is "Element found at index 1"

search element **80**

Step 1:

search element (80) is compared with middle element (50)



Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

Step 2:

search element (80) is compared with middle element (65)

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

Step 3:

search element (80) is compared with middle element (80)

Both are not matching. So the result is "Element found at index 7"

Binary Search Program

```
#include<iostream>
using namespace std;
int  main()
{
   int list[] = {10,20,30,40,50,60,70,80,80,100};
   int first, last, middle, size, i, s;
   size= sizeof(list)/sizeof(list[0]);
   cout<<"Enter the element to be Search";
   cin>>s;
   first = 0;
   last = size - 1;
   middle = (first+last)/2;
   while (first <= last)
   {
      if (list[middle] < s)
         first = middle + 1;
      else if (list[middle] == s)
          {
         cout<<"Element is found at posdition"<<middle+1;
         break;
      }
      else
         last = middle - 1;

      middle = (first + last)/2;
   }
   if (first > last)
      cout<<"Given element is not found in the list!!!";

}
```

SORTING:

Arranging the elements in a list in either ascending or descending order is called sorting. There are several types of sorting techniques:

1. **Bubble Sort:**

Bubble sort is also known as exchange sort. Bubble sort is a simplest sorting algorithm. In bubble sort algorithm array is traversed from 0 to the length-1 index of the array and compared one element to the next element and swap values in between if the next element is less than the previous element. In other words, bubble sorting algorithm compare two values and put the largest value at largest index. The algorithm follow the same steps repeatedly until the values of array is sorted.

Working of bubble sort algorithm:

Say we have an array unsorted A[0],A[1],A[2]..... A[n-1] and A[n] as input. Then the following steps are followed by bubble sort algorithm to sort the values of an array.

1.Compare A[0] and A[1] .
2.If A[0]>A[1] then Swap A[0] and A[1].
3.Take next A[1] and A[2].
4.Comapre these values.
5.If A[1]>A[2] then swap A[1] and A[2]
.....

at last compare A[n-1] and A[n]. If A[n-1]>A[n] then swap A[n-1] and A[n]. As we see the highest value is reached at n^{th} position. At next iteration leave n^{th} value. Then apply the same steps repeatedly on A[0],A[1],A[2]..... A[n-1] elements repeatedly until the values of array is sorted.

Ex:
12 9 4 99 120 1 3 10

The basic steps followed by algorithm:-
In the first step compare first two values 12 and 9.
12 9 4 99 120 1 3 10
As 12>9 then we have to swap these values
Then the new sequence will be
9 12 4 99 120 1 3 10
In next step take next two values 12 and 4
9 **12 4** 99 120 1 3 10
Compare these two values .As 12>4 then we have to swap these values.
Then the new sequence will be
9 ***4 12*** 99 120 1 3 10
We have to follow similar steps up to end of array. e.g.
9 4 **12 99** 120 1 3 10
9 4 12 **99 120** 1 3 10
9 4 12 99 **1 120** 3 10
9 4 12 99 1 **120 3** 10
9 4 12 99 1 3 120 10
9 4 12 99 1 3 10 120

When we reached at last index .Then restart same steps unlit the data is not sorted.
The output of this example will be : 1 3 4 9 10 12 99 120

Program :

```
#include<iostream>
using namespace std;
 void bubblesrt( int a[], int n )
{
    int i, j,t=0;
    for(i = 0; i < n; i++)
     for(j = 1; j < (n-i); j++)
      if(a[j-1] > a[j])
     {
         t = a[j-1];
         a[j-1]=a[j];
         a[j]=t;
       }
    }
int main()
{
    int i,size;
    int array[] = {12,9,4,99,120,1,3,10};
    size=sizeof(array)/sizeof(array[i]);
    cout<<"Values Before sorting:"<<endl;
    for(i = 0; i < size; i++)
    cout<< array[i]<<" ";
    cout<<endl;
    bubblesrt(array, size);
     cout<<"Values after  sorting:"<<endl;
    for(i = 0; i < size; i++)
    cout<< array[i]<<" ";
}
```

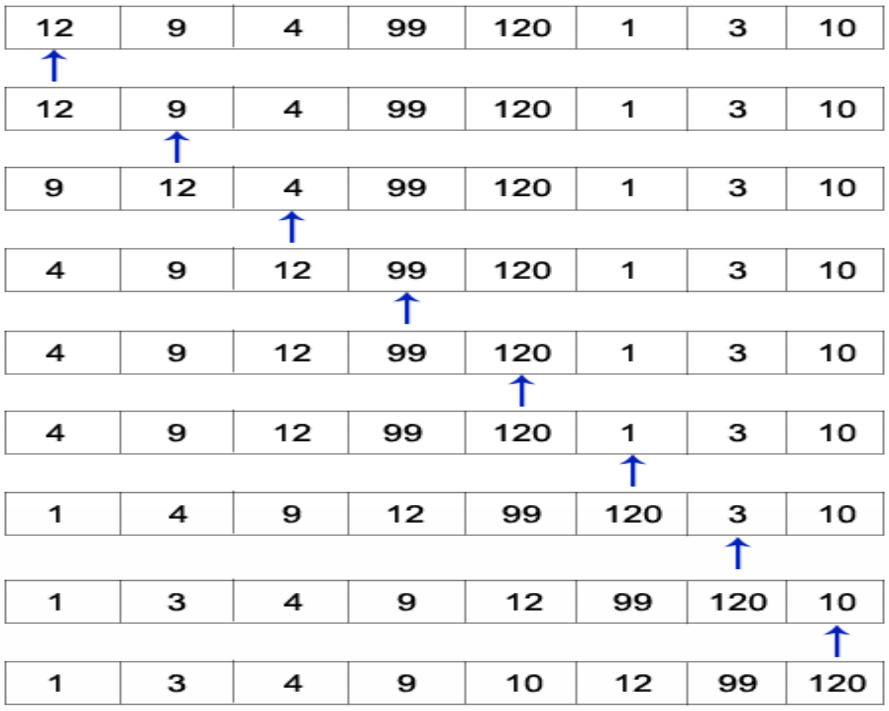
Insertion Sort:

Insertion sorting algorithm is similar to bubble sort. But insertion sort is more efficient than bubble sort because in insertion sort the elements comparisons are less as compare to bubble sort. In insertion sorting algorithm compare the value until all the prior elements are lesser than compared value is not found. This mean that the all previous values are lesser than compared value. This algorithm is more efficient than the bubble sort .Insertion sort is a good choice for small values and for nearly-sorted values.

Code description:

In insertion sorting take the element form left assign value into a variable. Then compare the value with previous values. Put value so that values must be lesser than the previous values. Then assign next value to a variable and follow the same steps relatively until the comparison not reached to end of array.

Ex:



Program:

```
#include<iostream>
using namespace std;
void insertionsrt(int array[], int n)
{   int i,j;
   for (i = 1; i < n; i++)
  {
     int j = i;
     int B = array[i];
     while ((j > 0) && (array[j-1] > B))
      {
       array[j] = array[j-1];
       j--;
      }
     array[j] = B;
   }
 }
```

```
int main()
{
   int i,size;
   int array[] = {12,9,4,99,120,1,3,10};
   size=sizeof(array)/sizeof(array[0]);
   cout<<"Values Before sorting:"<<endl;
   for(i = 0; i < size; i++)
   cout<< array[i]<<" ";
   cout<<endl;
   insertionsrt(array, size);
    cout<<"Values after  sorting:"<<endl;
   for(i = 0; i < size; i++)
   cout<< array[i]<<" ";

 }
```

Selection Sort:

In selection sorting algorithm, find the minimum value in the array then swap it first position. In next step leave the first value and find the minimum value within remaining values. Then swap it with the value of minimum index position. Sort the remaining values by using same steps. Selection sort is probably the most intuitive sorting algorithm to invent.

Code description:

In selection sort algorithm to find the minimum value in the array. First assign minimum index in key (index_of_min=x). Then find the minimum value and assign the index of minimum value in key (index_of_min=y). Then swap the minimum value with the value of minimum index. At next iteration leave the value of minimum index position and sort the remaining values by following same steps.

Working of the selection sort :

Say we have an array unsorted A[0],A[1],A[2]..... A[n-1] and A[n] as input. Then the following steps are followed by selection sort algorithm to sort the values of an array . (Say we have a key index_of_min that indicate the position of minimum value)

1. Initially variable index_of_min=0;

2.Find the minimum value in the unsorted array.

3. Assign the index of the minimum value into index_of_min variable.

4. Swap minimum value to first position.

5. Sort the remaining values of array (excluding the first value).

The code of the program :

```
#include<iostream>
using namespace std;
void selectionsort(int array[], int n)
 {
          int x,indexofmin;
   for(x=0; x<n; x++)
          {
     indexofmin = x;
     for(int y=x; y<n; y++)
            if(array[indexofmin]>array[y])
                    indexofmin = y;
     int temp = array[x];
     array[x] = array[indexofmin];
     array[indexofmin] = temp;
  }
}
int main()
{
   int array[] = {12,9,4,99,120,1,3,10};
   int i,size;
   size=sizeof(array)/sizeof(array[0]);
   cout<<"Values Before sorting:"<<endl;
   for(i = 0; i < size; i++)
          cout<< array[i]<<" ";
          cout<<endl;
   selectionsort(array, size);
    cout<<"Values after  sorting:"<<endl;
   for(i = 0; i < size; i++)
   cout<< array[i]<<" ";
 }
```

Quick Sort:

Quick sort is a comparison sort. The working of quick sort algorithm is depending on a divide-and-conquer strategy. A divide and conquer strategy is dividing an array into two sub-arrays. Quick sort is one of the fastest and simplest sorting algorithm.

Code description:

In quick sort algorithm pick an element from array of elements. This element is called the pivot. Then compare the the values from left to right until a greater element is find then swap the values. Again start comparison from right with pivot. When lesser element is find then swap the values. Follow the same steps until all elements which are less than the pivot come before the pivot and all elements greater than the pivot come after it. After this partitioning, the pivot is in its last position. This is called the partition operation. Recursively sort the sub-array of lesser elements and the sub-array of greater elements.

Ex:

Input:12 9 4 99 120 1 3 10 13

Output:1 3 4 10 12 13 99 120

Program:

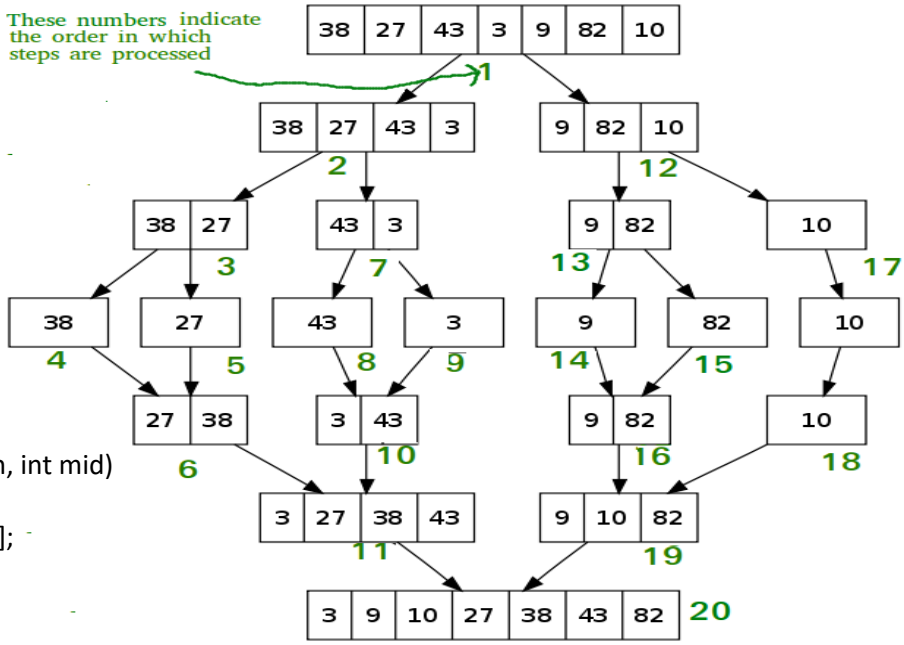
```
#include<iostream>
using namespace std;
void quicksort(int array[],int low, int n)
{           int mid,t;
   int lo = low;
   int hi = n;
   if (lo >= n)
           return;
   mid = array[(lo + hi) / 2];
   while (lo < hi)
           {
    while (lo<hi && array[lo] < mid)
      lo++;
   while (lo<hi && array[hi] > mid)
      hi--;
   if (lo < hi)
           {
      t = array[lo];
      array[lo] = array[hi];
      array[hi] = t;
   }
   }
   if (hi < lo)
           {
    int t = hi;
    hi = lo;
    lo = t;
   }
   quicksort(array, low, lo);
   quicksort(array, lo == low ? lo+1 : lo, n);
 }

 int main()
 {
   int array[] = {12,9,4,99,120,1,3,10,13};
   int i,size;
   size=sizeof(array)/sizeof(array[0]);
   cout<<"Values Before sorting:"<<endl;
   for(i = 0; i < size; i++)
   cout<< array[i]<<" ";
   cout<<endl;
   quicksort(array,0,size-1);
   cout<<"Values after  sorting:"<<endl;
   for(i = 0; i < size; i++)
   cout<< array[i]<<" ";
 }
```

Merge Sort:

Like QuickSort, Merge Sort is a Divide and Conquer algorithm. It divides input array in two halves, calls itself for the two halves and then merges the two sorted halves. **The merge() function** is used for merging two halves. The merge(arr, l, m, r) is key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

The following diagram from shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. The array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.

Ex:



Program:

```
#include <iostream>
using namespace std;
void  Merge(int *a, int low, int high, int mid)
{
        int i, j, k, temp[high-low+1];
        i = low;
        k = 0;
        j = mid + 1;
        while (i <= mid && j <= high)
        {               if (a[i] < a[j])
                {
                        temp[k] = a[i];
                        k++;
                        i++;
                }
                else
                {
                        temp[k] = a[j];
                        k++;
                        j++;
                }
        }
        while (i <= mid)
        {
                temp[k] = a[i];
                k++;
                i++;
        }

        while (j <= high)
        {
                temp[k] = a[j];
                k++;
                j++;
        }

        for (i = low; i <= high; i++)
        a[i] = temp[i-low];
}
```

```
void MergeSort(int *a, int low, int high)
{
        int mid;
        if (low < high)
        {
                mid=(low+high)/2;
                MergeSort(a, low, mid);
                MergeSort(a, mid+1, high);
                Merge(a, low, high, mid);
        }
}

int main()
{
        int array[] = {12,9,4,99,120,1,3,10,13};
   int i,size;
   size=sizeof(array)/sizeof(array[0]);
   cout<<"Values Before sorting:"<<endl;
   for(i = 0; i < size; i++)
        cout<< array[i]<<" ";
        cout<<endl;
   MergeSort(array,0,size-1);
   cout<<"Values after  sorting:"<<endl;
   for(i = 0; i < size; i++)
   cout<< array[i]<<" ";
}
```

Comparisons of sorting techniques:

Table 9.6 Comparison of sorting techniques

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable?	Pros	Cons
Bubble sort	Repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order	$O(n^2)$	$O(n^2)$	No extra space needed	Yes	1. A simple and easy method 2. Efficient for small lists $n > 100$	Highly inefficient for large data
Selection sort	Finds the minimum value in the list and then swaps it with the value in the first position, repeats these steps for the remainder of the list (starting at the second position and advancing each time)	$O(n^2)$	$O(n^2)$	No extra space needed	No	1. Recommended for small files 2. Good for partially sorted data	Inefficient for large lists
Insertion sort	Every repetition of insertion sort removes an element from the input data, inserts it into the correct position in the already sorted list until no input elements remain. The choice of which element to remove from the input is arbitrary and can be made using almost any choice of algorithm	$O(n)$	$O(n^2)$	No extra space needed	Yes	1. Relatively simple and easy to implement 2. Good for almost sorted data	Inefficient for large lists
Quick sort	Picks an element, called a pivot, from the list. Reorders the list so that all elements with values less than the pivot come before the pivot, whereas all elements with values 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. Recursively sorts the sub-list of the lesser elements and the sub-list of the greater elements.	$O(n\log_2 n)$	$O(n^2)$	No extra space needed	No	1. Extremely fast 2. Inherently recursive	Very complex algorithm

Heaps:

BASIC CONCEPTS:

Def:

A *heap* is a binary tree having the following properties:

1. It is a *complete binary tree*, that is, each level of the tree is completely filled, except the bottom level, where it is filled from left to right.
2. It satisfies the heap-order property , that is, the key value of each node is greater than or equal to the key value of its children, or the key value of each node is lesser than or equal to the key value of its children.

All the binary trees of Fig. 12.1 are heaps, whereas the binary trees of Fig. 12.2 are not. The second condition is violated in Fig. 12.2(a) as the content of the child node 80 is greater than its parent node 70. The first condition is violated in Fig. 12.2(b) as at level 2, 30 has a right child but no left child, that is, at this level, it should be filled from left to right.

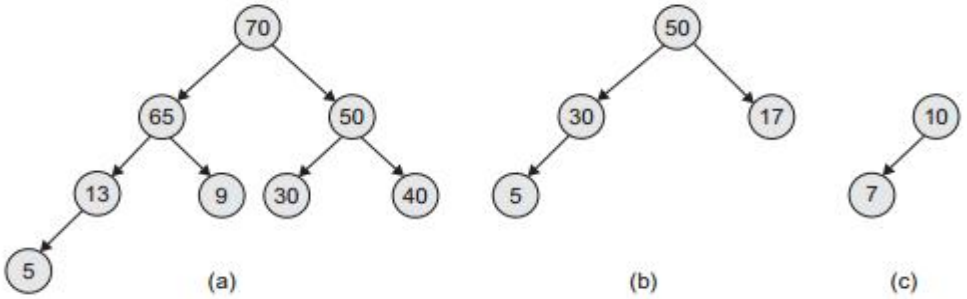


Fig. 12.1 Sample heaps (a) Heap with height three (b) Heap with height two (c) Heap with height one

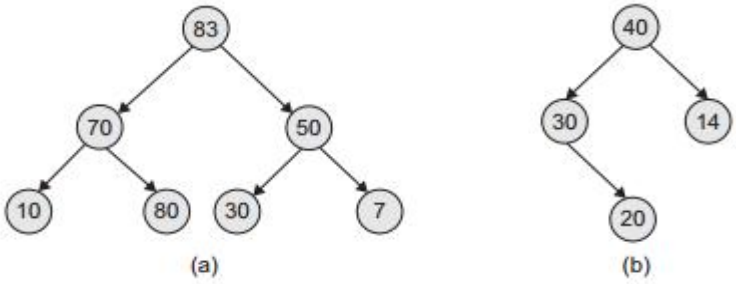


Fig. 12.2 Binary trees but not heaps (a) Sample 1 (b) Sample 2

Min-heap and Max-heap

Min-heap

The structure shown in Fig. 12.3 is called *min-heap*. In min-heap, the key value of each node is lesser than or equal to the key value of its children. In addition, every path from root to leaf should be sorted in ascending order.

Figure 12.4 is an example of a min-heap.

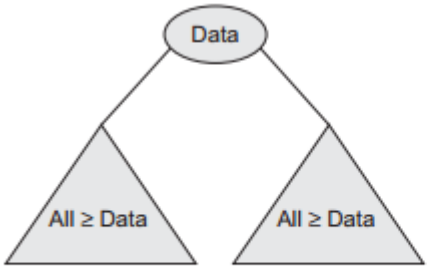


Fig. 12.3 Structure of min-heap

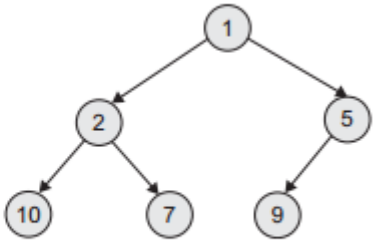


Fig. 12.4 An example of a min-heap

Max-heap

A max-heap is where the key value of a node is greater than or equal to the key value of its children. In general, whenever the term 'heap' is used by itself, it refers to a max-heap as shown in Fig. 12.5. In addition, every path from the root to leaf should be sorted in descending order. Figure 12.6 is an example of a max-heap.

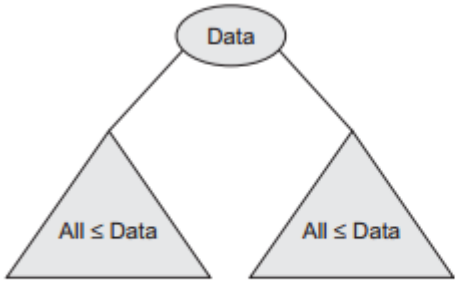


Fig. 12.5 A max-heap

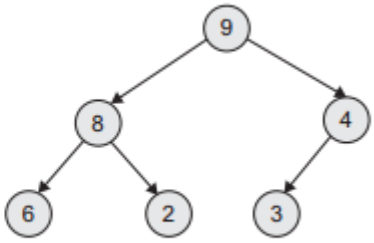


Fig. 12.6 An example of a max-heap

IMPLEMENTATION OF HEAP:

To implement heaps using array is an easy task. We simply number the nodes in the heap from top to bottom, number the nodes on each level from left to right, and store the i_{th} node in the i_{th} location of the array.

The root of the tree is stored at index 0, its left child at index 1, its right child at index 2, and so on.

For example, consider Fig 12.7.

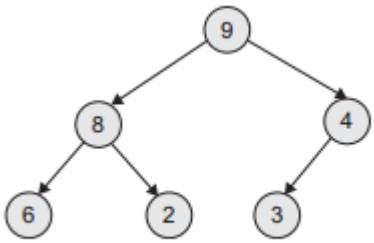
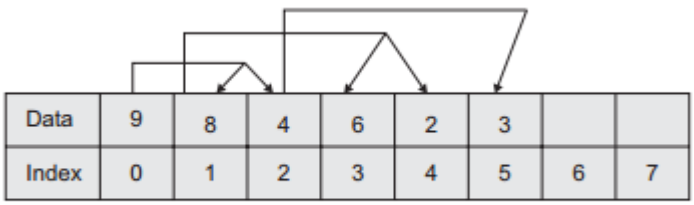


Fig. 12.7 Sample heap

Figure 12.8 shows the corresponding array representation of the heap.



Data	9	8	4	6	2	3		
Index	0	1	2	3	4	5	6	7

Fig 12.8 Array representation of heap in Fig. 12.7

In this array,

1. parent of the node at index i is at index $(i - 1)/2$

2. left child of the node at index i is at index $2 \infty i + 1$

3. right child of the node at index i is at index $2 \infty i + 2$

For example, in Fig. 12.8,

1. the node having value 8 is at the 1st location.

2. Its parent is at 0/2, that is, at the 0th location (value is 9).

3. Its left child is at $2 \infty 1 + 1$, that is, at the 3rd location (value is 6).

4. Its right child is at $2 \infty 1 + 2$, that is, at the 4th location (value is 2).

Let us consider the heap tree in Fig. 12.9 in its logical form.

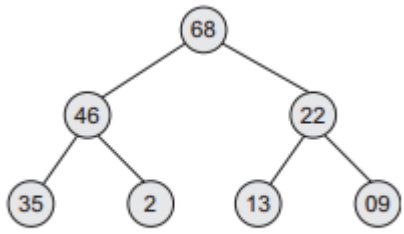


Fig. 12.9 A heap tree

The physical representation of the heap tree of Fig. 12.9 is shown in Fig. 12.10. We represent the tree using an array as in Fig. 12.10 using the rules stated.

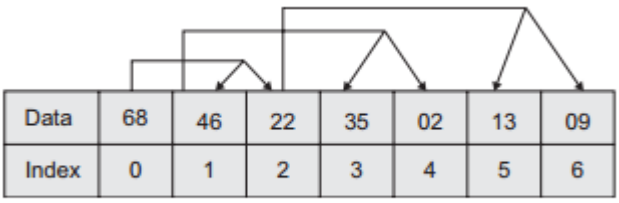


Fig. 12.10 Representation of heap in Fig. 12.9 as array

HEAP AS ABSTRACT DATA TYPE:

A heap is a complete binary tree, which satisfies the heap-order property, that is, the key value of each node is greater than or equal to the key value of its children (or the key value of each node is lesser than or equal to the key value of its children). The basic operations on heap are insert, delete, max-heap, and min-heap.

```
ADT Heap

1.Create()∅Heap

2.Insert(Heap, Data)∅Heap

3.DeleteMaxVal(Heap)∅Heap

4.ReHeapDown(Heap, Child)∅Heap

5.ReHeapUp(Heap, Root)∅Heap

End
```

The C++ class declaration for this ADT is as follows:

```
class HeapNode

{

int A[max];

int n; //No. of elements heap contains

};

class Heap

{

private:

HeapNode *Root;

void ReHeapUp(int i);

void ReHeapDown(int i);

public:

Heap();

{

for(int i = 0; i < max; i++)

A[i] = 0;

}

void Create();

void Insert(int i);

void DeleteMaxVal();

};
```

Heap Sort:

```
#include <iostream>
using namespace std;
void MaxHeapify(int a[], int i, int n)
{
	int j, temp;
	temp = a[i];
	j = 2*i;

	while (j <= n)
	{
		if (j < n && a[j+1] > a[j])
		j = j+1;
		if (temp > a[j])
			break;
		else if (temp <= a[j])
		{
			a[j/2] = a[j];
			j = 2*j;
		}
	}
	a[j/2] = temp;
	return;
}
void HeapSort(int a[], int n)
{
	int i, temp;
	for (i = n; i >= 2; i--)
	{
		temp = a[i];
		a[i] = a[1];
		a[1] = temp;
		MaxHeapify(a, 1, i - 1);
	}
}
void Build_MaxHeap(int a[], int n)
{
	int i;
	for(i = n/2; i >= 1; i--)
		MaxHeapify(a, i, n);
}
int main()
{
	int n, i,size;
	int array[] = {12,9,4,99,120,1,3,10,13};
    size=sizeof(array)/sizeof(array[0]);
    cout<<"Values Before sorting:"<<endl;
    for(i = 1; i < size; i++)
	cout<< array[i]<<" ";
	cout<<endl;
	Build_MaxHeap(array, size-1);
	HeapSort(array, size-1);
    cout<<"Values after  sorting:"<<endl;
    for(i = 1; i < size; i++)
    cout<< array[i]<<" ";
}
```