4.1 Unit – III Searching and Sorting

Searching: Linear Search, Binary Search, Fibonacci search.

Bubble Sort, Selection Sort, Insertion Sort, Quick Sort, Merge Sort Merging, Iterative Merge Sort, Recursive Merge Sort, Heap Sort

4.1.1 Teaching Material / Teaching Aids as per above lecture plan.

4.1.1.1 Lecture-1: Linear Search

Linear Search

Searching techniques are used to retrieve a particular record from a list of records in an efficient manner so that least possible time is consumed. The list of records can have a key field, which can be used to identify a record. Therefore, the given value must be matched with the key value in order to find a particular record. If the searching technique identifies a particular record, then the search operation is said to be successful; otherwise, It is said to be unsuccessful.

The techniques which are used to search data from the memory of the computer are called internal searching techniques. The techniques that are used to search data from a storage device- such as a hard disk or a floppy disk-are called external searching techniques.
searching techniques:

- Linear search
- Binary search
- Fibonacci search

Linear Search
Linear search, also known as sequential search involves searching a data item from a given set of data items. These data items are stored in structures, such as arrays. Each data item is checked one by one in the order in which it exists in the structure to determine if it matches the criteria of search.

- It is a simple searching method. In this method, we begin searching from starting of the list to end of the list.
- This method is also called as sequential search.
- To implement this method we require a simple looping structure to find the element from 0 to n-1 in an array.
- In every iteration we have to check each element of the array with the key element.

- If the key element is found then stop the process and print the position of the key element in an array otherwise print the element is not found.

Advantages of Linear Searching

1. It is simple to implement
2. It does not require specific ordering before applying the method

Disadvantages of Linear Searching

1. It is less efficient.

```
/*Linear Search*/

#include<stdio.h>

#include<conio.h>

int a[20],i,n,key,count=0;

void main()

{

clrscr();

printf("\nEnter the size of the array");

scanf("%d",&n);

printf("Enter the array element");

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

scanf("%d",&a[i]);

printf("Enter the key");

scanf("%d",&key);

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

{

if(a[i]==key)

{

Count=count+1;
break;
}
}
if(count==1)
printf("The key element is found at position is %d",i+1);
else
printf("The key element is not found");
getch();
}
```

4.1.1.2 Lecture-2: Binary Search

Binary search

- Binary search is the fastest searching algorithm
- It works on the ordered list either in ascending order or descending order
- The principle used in this search is to divide the list into two halves and compare the key with the middle element.
- If the comparison results in true then print its position.
- If it is false, then check key element is greater than the middle element or less than the middle element.
- If the key element is greater than the middle element then search the key element in second half of the list.
- If the key is less than the middle element then search the key in the first half of the list
- Same process is repeated for both the sub lists depending upon the key present in half of the list or second half of the list.

Advantages of binary searching

1. It is an efficient technique.

Disadvantages of binary searching

1. It requires specific ordering before the applying the methods
2. It is complex to implement.

/*binary search*/

```
#include <stdio.h>

#include<conio.h>

void main()

{

int a[10],low,high,mid,i,n,key;

clrscr();

printf ("\nEnter the size of the array");

scanf("%d",&n);

printf("Enter the element in ascending order\n");

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

scanf("%d",&a[i]);

printf("Enter the key element");

scanf("%d",&key);

low=0;

high=n-1;

mid=(low+high)/2;

while(low<high)

{

if(key==a[mid])

break;

if(key>a[mid])

low=mid+1;
else
high=mid-1;
mid=(low+high)/2;
}
if(key==a[mid])
printf("The element is found at %d",mid+1);
else
printf("The element is not present in the list");
getch();
}
```

4.1.1.3 Lecture-31: Fibonacci search

Fibonacci Search

In binary search method we divide the number at mid and based on mid element i.e. by comparing mid element with key element we search either the left sublist or right sublist. Thus we go on dividing the corresponding sublist each time and comparing mid element with key element. If the key element is really present in the list, we can reach to the location of key in the list and this we get the message the “element is present in the list” otherwise get the message. “element is not present in the list”.

In fiboacci search rather than considering the mid element,we consider the indices as the numbers from Fibonacci series. As we know, the Fibonacci series is…

0 1 1 2 3 5 8 13 21….

To understand how Fibonacci search works, we will consider one example, suppose following is the list of elements.

Arr[]

10	20	30	40	50	60	70

1 2 3 4 5 6 7

Here n= Total number of elements =7

We will always compute 3 variables i.e a,b and f.

Initially f=n=7.

For setting a and b variables we will consider elements from Fibonacci series.

0 1 1 2 3 5 8

a=3 i.e. previous element of b

b=5 because this is the only element in the Fibonacci series which is <n i.e.<7

with these initial values we will start searching the key element from the list. Each time we will compare key element with arr[f]. that means

if (key<arr[f])

f=f-a

b=a

a=b-a

if (key>arr[f])

f=f+a

b=b-a

a=a-bsuppose we have f=7,b=5,a=3

		A	B			f
10	20	30	40	50	60	70
1	2	3	4	5	6	7

If key=20

i.e. key < arr [f]

i.e. 20<70

Therefore f=f-a=7-3=4

b=a=3

a=b-a=2

Again we compare **if (key < arr [f])**

i.e. 20 < arr [4]

i.e. if(20<40)->yes

at present f=4,b=3,a=2

Therefore f=f-a=4-2=2

b=a=2

a=b-a=3-2=1

Now we get f=2, b=2, a=1

a	f/b					
10	20	30	40	50	60	70
1	2	3	4	5	6	7

If (key < arr [f] i.e. if (20 < 20) -> no

If key> arr [f] i.e. if(20 > 20)->No

That means “element is present at f=2 location.”**If key=60**

i.e. key<arr[f]

i.e. 60<70

therefore f=f-a=7-3=4

b=a=3

a =b –a = 2

Again we compare **if (key > arr [f])** i.e. 60 > arr[f] i.e. 40

Therefore f=f + a = 4 + 2 = 6

b = b – a = 3 – 2 = 1

a = a – b = 2 – 3 = -1

a	b				F	
10	20	30	40	50	60	70
1	2	3	4	5	6	7

If (key < arr[f] i.e. if (60 < 60) -> no

If key > arr[f] i.e. if(60 > 60) ->No

That means “element is present at f=6 location.”

Analysis: the time complexity of Fibonacci search is **O(logn).**

```
/*Program to demonstrate Fibonacci Search*/

#include <stdio.h>

#include<conio.h>

void main()

{

int i,n,key,arr[10];

int Read_input(int arr[],int n);

void search(int arr[],int n,int key,int f,int a,int b);

int Fib(int n);

clrscr();

printf("\nProgram for Fibonacci search\n");

n=Read_input(arr,n);
```

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

scanf("%d",&key);

search(arr,n,key,n,Fib(n),Fib(Fib(n)));

getch();

}

int Read_input(int arr[10],int n)

{

int i;

printf("\nEnter the total number of elements:");

scanf("%d",&n);

printf("\nEnter the elements in ascending order\n");

for(i=1;i<=n;i++)

scanf("%d",&arr[i]);

return n;

}

int Fib(int n)/*obtaining fibonacci number*/

{

int a,b,f;

if(n<1)

return n;

a=0;

b=1;

while(b<n)

{

f=a+b;
a=b;

b=f;

}

return a;

}

void search (int arr[],int n,int key,int f,int b,int a)

{

if(f<1||f>n)

printf("\nThe number is not present");

else if(key<arr[f])

{

if(a<=0)

printf("\nThe element is not present");

else

search(arr,n,key,f-a,a,b-a);

}

else if(key>arr[f])

{

if(b<=1)

printf("\n The element is not presnet");

else

search(arr,n,key,f+a,b-a,a-b);

}

else

printf("\nThe element is present at %d",f);
```

}

4.1.1.4 Lecture-32: Exam on Searching Techniques/Revision

4.1.1.5 Lecture-33: Bubble Sort

Sorting is an important concept that is extensively used in the fields of computer science. Sorting is nothing but arranging the elements in some logical order.

For example, we want to obtain the telephone number of a person. If the tele phone directory is not arranged in alphabetical order, one has to search from the very first page to till the last page.

If the directory is sorted, we can easily search for the telephone number

Bubble sort

This is the simplest sorting technique when compare with all other sorting.

The bubble sort technique starts with comparing the first two data items of an array and swapping these two data items if they are not in a proper order, i.e. in descending or ascending order. This process continues until all the data items in the array are compared or the end of an array is reached.

Algorithm:

1. Compare the first two elements in the array, say A[1] and A[2], if A[2] is less A[1], then interchange the values.
2. Compare A[2] and A[3]. If A[3] is less than A[2], then interchange the values.
3. Continue this process till the last two elements are compared and interchanged.
4. Repeat the above steps for n-1 passes.

- After the completion of first pass the largest element is placed in n^{th} position.
- After second pass second largest element is placed in n-1th position.
- In the i^{th} pass i^{th} largest element is places in $n-(i-1)^{th}$ position.
- If the element are not in sorted order then it requires n-1 passes.
- If the element are in sorted order then it requires only one pass.

Example:

The elements are 50,40,30,20,10

After sorting the list is 10,20,30,40,50

Tracing: EX->exchange elements NC->no exchange

/*Bubble sort*/

```
void bubblesort(int a[20],int n)
  {
    for(i=0;i<=n;i++)
    {
    for(j=0;j<=n-1;j++)
       {
         if(a[j]>=a[j+1])
            {
               temp=a[j];
              a[j]=a[j+1];
              a[j+1]=temp;
            }
        }
    }
}
```

4.1.1.6 Lecture-34: Selection Sort

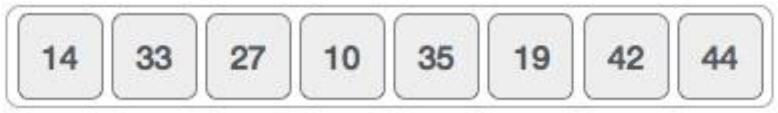
Selection sort:

Selection sort is a simple sorting algorithm. This sorting algorithm is a in-place comparison based algorithm in which the list is divided into two parts, sorted part at left end and unsorted part at right end. Initially sorted part is empty and unsorted part is entire list.

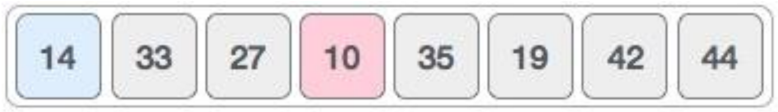
Smallest element is selected from the unsorted array and swapped with the leftmost element and that element becomes part of sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where n are no. of items.

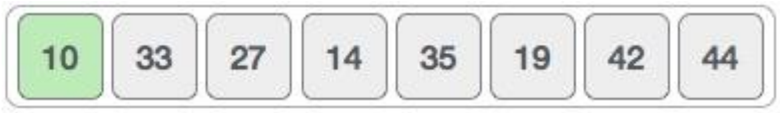
We take the below depicted array for our example.



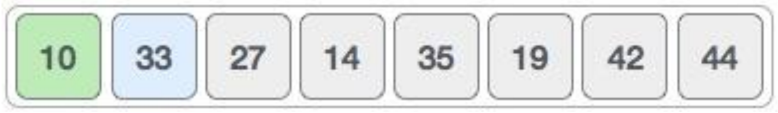
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.



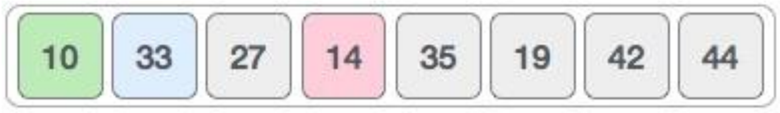
So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of sorted list.



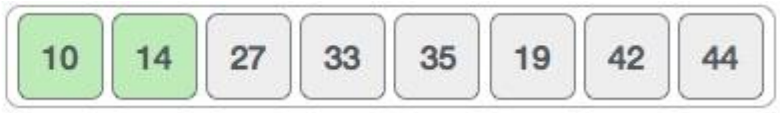
For the second position, where 33 is residing, we start scanning the rest of the list in linear manner.



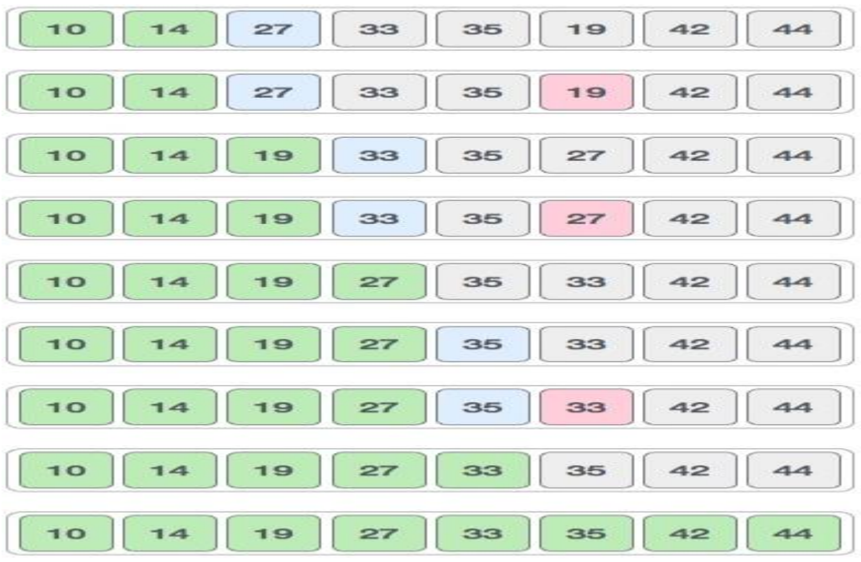
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.



After two iterations, two least values are positioned at the the beginning in the sorted manner.



The same process is applied on the rest of the items in the array. We shall see an pictorial depiction of entire sorting process −



Now, we should learn some programming aspects of selection sort.

Pseudocode

```
procedure selection sort
   list  : array of items
   n     : size of list

   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i

      /* check the element to be minimum */

      for j = i+1 to n
         if list[j] < list[min] then
            min = j;
         end if
      end for

      /* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if

   end for

end procedure
```

4.1.1.7 Lecture-35: Insertion Sort

Insertion sort:

The insertion sort works very well when the no of element are very less

This technique is similar to the way a librarian keeps the books in the shelf.

Initially all the books are placed in shelf according to their access number.

When a student returns the book to the librarian, he compares the access number of this book with all other books of access numbers and inserts it into the correct position, so that all books are arranged in order with respect to their access numbers.

Method:

- Pass1: the 1st element in the array is itself is sorted.
- Pass2: 2nd element is inserted either before or after 1st element, so that 1st and 2nd elements are sorted.
- Pass3: 3rd element is inserted into its proper place, that is before 1st and 2nd , or between 1st and 2nd elements, or after 1st and 2nd elements.
- Pass4: 4th element is inserted into its proper place in 1st , 2nd ,3rd so that 1st,2nd,3rd,4th element are sorted.
- Pass N: Nth element is inserted into its proper place so that total list is sorted.

```
Insertion_sort(int a[10])

{

 for(j=2;j<n;j++)

  {

    key=a[j];

    i=j-1;

 while(i>0 && a[i]>key)

  {

      a[i+1]=a[i];

       i=i-1;

    }
  a[i+1]=key;
 }
}
```

4.1.1.8 Lecture-36: Quick Sort

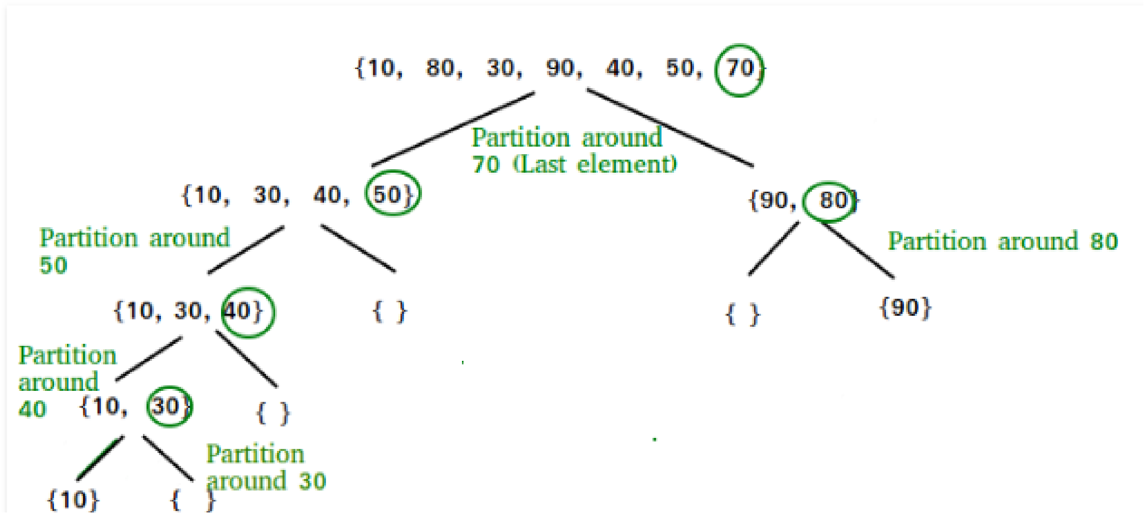
Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

1. Always pick first element as pivot.
2. Always pick last element as pivot (implemented below)
3. Pick a random element as pivot.
4. Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.

Pseudo Code for recursive QuickSort function :

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[pi] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```



Partition Algorithm

There can be many ways to do partition, following pseudo code adopts the method given in CLRS book. The logic is simple, we start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.

```
/* low  --> Starting index,  high  --> Ending index */
quickSort(arr[], low, high)
{
    if (low < high)
    {
        /* pi is partitioning index, arr[p] is now
           at right place */
        pi = partition(arr, low, high);

        quickSort(arr, low, pi - 1);  // Before pi
        quickSort(arr, pi + 1, high); // After pi
    }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places
   the pivot element at its correct position in sorted
    array, and places all smaller (smaller than pivot)
   to left of pivot and all greater elements to right
   of pivot */
partition (arr[], low, high)
{
    // pivot (Element to be placed at right position)

    pivot = arr[high];


    i = (low - 1)  // Index of smaller element


    for (j = low; j <= high- 1; j++)

    {

        // If current element is smaller than or

        // equal to pivot

        if (arr[j] <= pivot)

        {

            i++;    // increment index of smaller element

            swap arr[i] and arr[j]

        }

    }

    swap arr[i + 1] and arr[high])

    return (i + 1)

}
```

Illustration of partition() :

```
arr[] = {10, 80, 30, 90, 40, 50, 70}

Indexes:  0   1   2   3   4   5   6


low = 0, high =  6, pivot = arr[h] = 70

Initialize index of smaller element, i = -1

Traverse elements from j = low to high-1
j = 0 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j
                                     // are same

j = 1 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30

j = 3 : Since arr[j] > pivot, do nothing
// No change in i and arr[]

j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped
j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped
```

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.

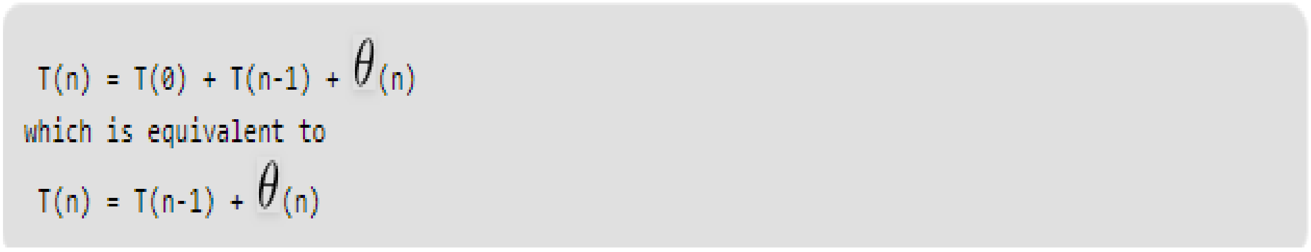
Analysis of QuickSort

Time taken by QuickSort in general can be written as following.

$$T(n)=T(k)+T(n-k-1)+\Theta(n)$$

The first two terms are for two recursive calls, the last term is for the partition process. k is the number of elements which are smaller than pivot.
The time taken by QuickSort depends upon the input array and partition strategy. Following are three cases.

Worst Case: The worst case occurs when the partition process always picks greatest or smallest element as pivot. If we consider above partition strategy where last element is always picked as pivot, the worst case would occur when the array is already sorted in increasing or decreasing order. Following is recurrence for worst case.



```
 T(n) = T(0) + T(n-1) + θ(n)
which is equivalent to
 T(n) = T(n-1) + θ(n)
```

The solution of above recurrence is $\Theta(n^2)$

Best Case: The best case occurs when the partition process always picks the middle element as pivot. Following is recurrence for best case.

$$T(n)=2T(n/2)+\Theta(n)$$

The solution of above recurrence is O(n log n). It can be solved using case 2 of Master Theorem.

Average Case:
To do average case analysis, we need to consider all possible permutation of array and calculate time taken by every permutation which doesn't look easy.
We can get an idea of average case by considering the case when partition puts O(n/9) elements in one set and O(9n/10) elements in other set. Following is recurrence for this case.

$$T(n)=T(n/9)+T(9n/10)+\Theta(n)$$

The solution of above recurrence is O(n log n). It can be solved using case 2 of Master Theorem.

4.1.1.9 Lecture-37: Merge Sort- Merging

Merge sort

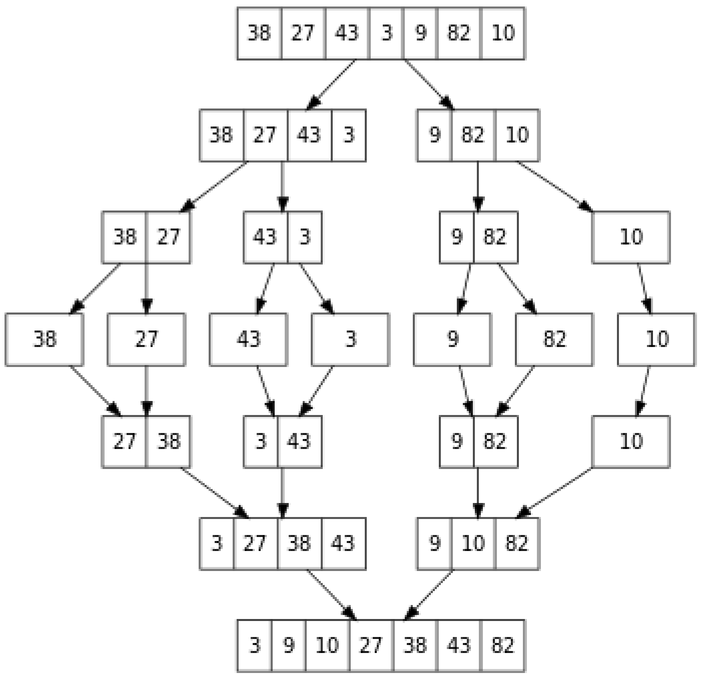
The merge sort technique combines two sorted arrays into one larger sorted array. The merge sort is a sorting algorithm that uses the divide and conquers strategy. In this method division is dynamically carried out.

Merge sort on an input array with n elements consists of three steps:

Divide: partition array into two sub lists s1 and s2 and with n/2 elements each.

Conquer: then sort sub list s1 and sub list s2.

Combine: merge s1 and s2 into a unique sorted group.



```
void partition(int arr[],int low,int high)
{
     int mid;
    if(low<high)
```

```
       {
         mid=(low+high)/2;
         partition(arr,low,mid);
         partition(arr,mid+1,high);
         mergeSort(arr,low,mid,high);
      }
}
void mergeSort(int arr[],int low,int mid,int high)
{
    int i,m,k,l,temp[MAX];
    l=low;
    i=low;
    m=mid+1;
    while((l<=mid)&&(m<=high))
        {
         if(arr[l]<=arr[m])
          {
             temp[i]=arr[l];
             l++;
         }
         else
         {
             temp[i]=arr[m];
             m++;
         }
         i++;
    }
    if(l>mid)
      {
         for(k=m;k<=high;k++)
            {
             temp[i]=arr[k];
             i++;
            }
   }
    else
      {
         for(k=l;k<=mid;k++)
          {
             temp[i]=arr[k];
            i++;
          }
      }
        for(k=low;k<=high;k++)
        {
           arr[k]=temp[k];
     }
}
```

4.1.1.10 Lecture-38: Iterative Merge Sort

4.1.1.11 Lecture-39: Recursive Merge Sort

4.1.1.12 Lecture-40: Heap Sort

Heap Sort

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible (Source Wikipedia)

A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater(or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.

2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.

3. Repeat above steps while size of heap is greater than 1.

How to build the heap?
Heapify procedure can be applied to a node only if its children nodes are heapified. So the heapification must be performed in the bottom up order.

Heaps can be used in sorting an array. In max-heaps, maximum element will always be at the root. Heap Sort uses this property of heap to sort the array.

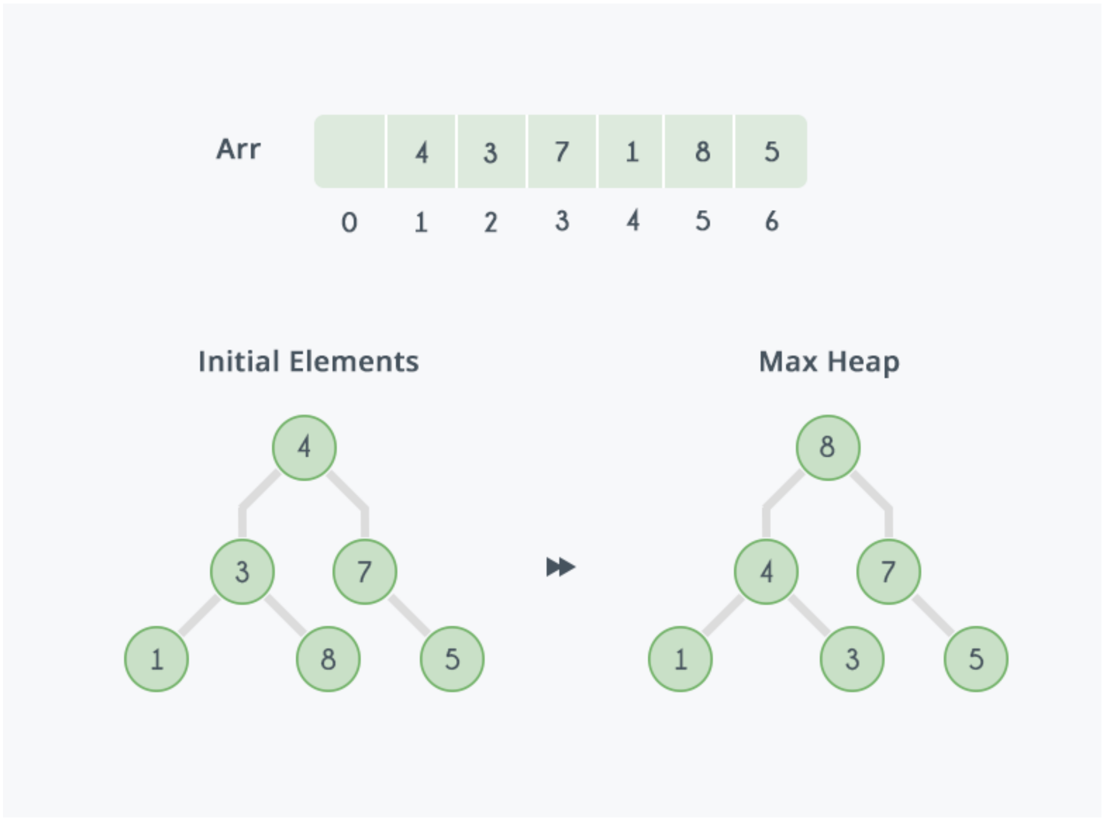
Consider an array Arr which is to be sorted using Heap Sort.

- Initially build a max heap of elements in Arr.
- The root element, that is Arr[1], will contain maximum element of Arr. After that, swap this element with the last element of Arr and heapify the max heap excluding the last element which is already in its correct position and then decrease the length of heap by one.
- Repeat the step 2, until all the elements are in their correct position.

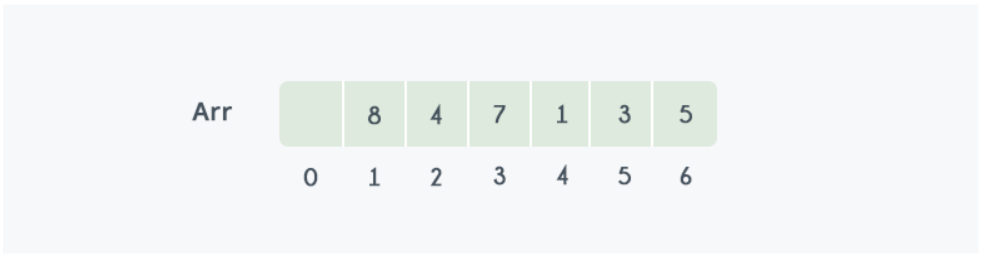
Implementation:

```
void heap_sort(int Arr[ ])

{
   int heap_size = N;

   build_maxheap(Arr);
   for(int i = N; i >= 2 ; i-- )
   {
      swap|(Arr[ 1 ], Arr[ i ]);
      heap_size = heap_size - 1;
      max_heapify(Arr, 1, heap_size);
   }
}
```

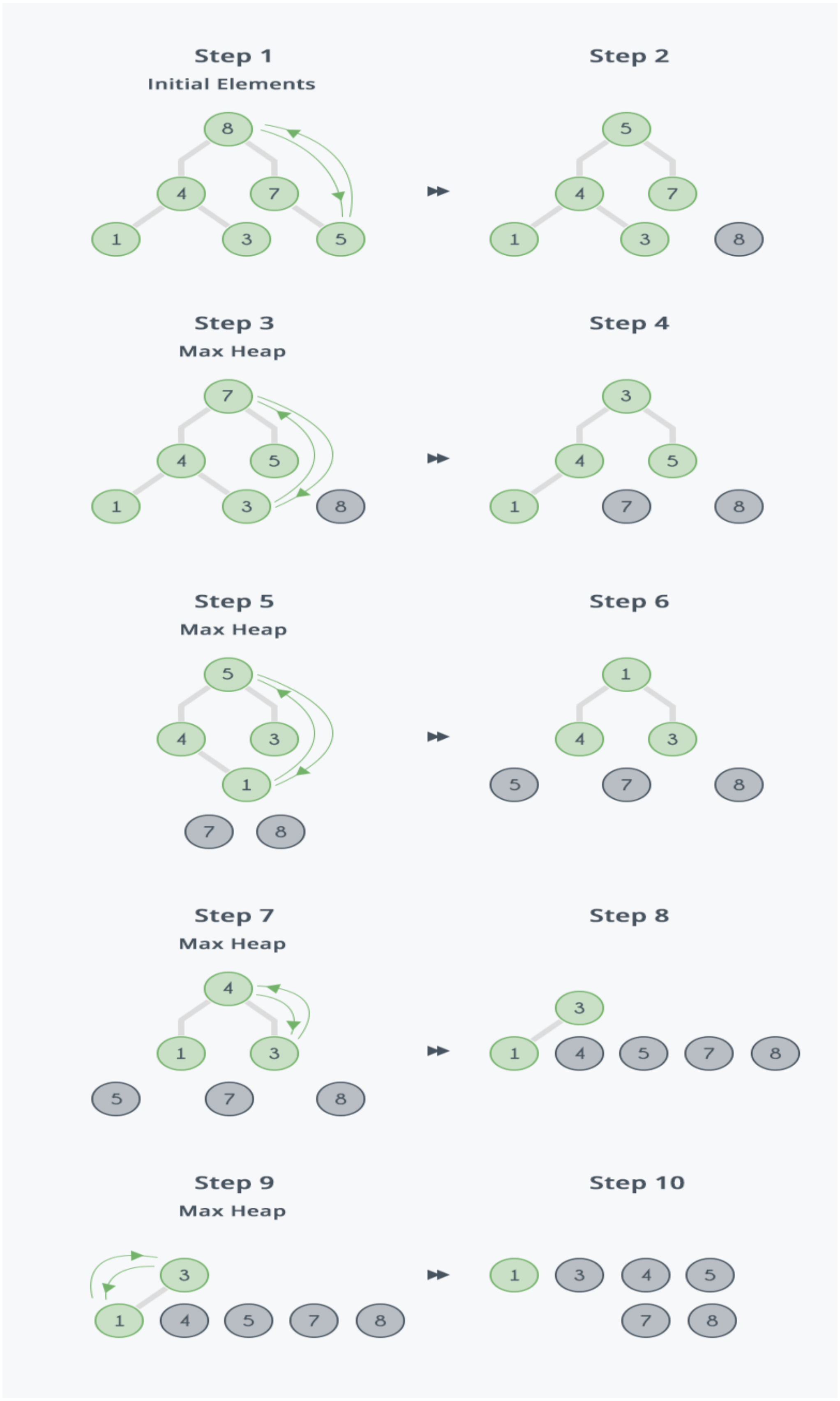
Lets understand with the help of an example:



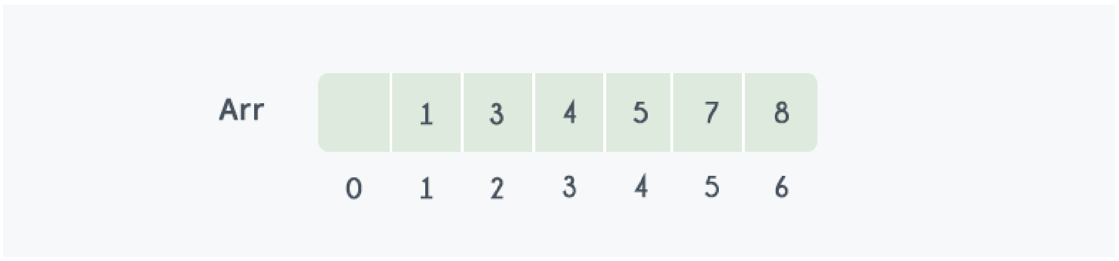
After building max-heap, the elements in the array Arr will be:



Step 1: 8 is swapped with 5.
Step 2: 8 is disconnected from heap as 8 is in correct position now and.
Step 3: Max-heap is created and 7 is swapped with 3.
Step 4: 7 is disconnected from heap.
Step 5: Max heap is created and 5 is swapped with 1.
Step 6: 5 is disconnected from heap.
Step 7: Max heap is created and 4 is swapped with 3.
Step 8: 4 is disconnected from heap.
Step 9: Max heap is created and 3 is swapped with 1.
Step 10: 3 is disconnected.


Step 1
Initial Elements
Step 2
Step 3
Max Heap
Step 4
Step 5
Max Heap
Step 6
Step 7
Max Heap
Step 8
Step 9
Max Heap
Step 10

After all the steps, we will get a sorted array.



```
// C++ program for implementation of Heap Sort
#include <iostream>
using namespace std;

// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
    int largest = i;  // Initialize largest as root
    int l = 2*i + 1;  // left = 2*i + 1
    int r = 2*i + 2;  // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i)
    {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// main function to do heap sort
void heapSort(int arr[], int n)
{
    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        swap(arr[0], arr[i]);

        // call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
    for (int i=0; i<n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
}

// Driver program
int main()
{
    int arr[] = {12, 11, 13, 5, 6, 7};
    int n = sizeof(arr)/sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is \n";
    printArray(arr, n);
}
```
Run on IDE

Output:

```
Sorted array is
5 6 7 11 12 13
```

Here is previous C code for reference.

Notes:

Heap sort is an in-place algorithm.

Its typical implementation is not stable, but can be made stable (See this)

Time Complexity: Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

Applications of HeapSort

1. Sort a nearly sorted (or K sorted) array

2. k largest(or smallest) elements in an array

Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.