SORTING

Sorting Basics

- Internal Sorting: Entire data is sorted in main memory (no. of data elements is limited)
- External Sorting: Data resides in disk or tape and sorting is performed on the data on such media.
- Assumed that data can be ordered based on some comparison operation.
- Time complexity depends on the no. of comparisons to perform the sort.

Insertion Sort

- Take the next element from the unsorted array. Insert it into the proper place in the already-sorted sub-array.

Example

0	*1*	*2*	*3*	*4*	*5*	*6*
Null	*G*	*D*	*Z*	*F*	*B*	*E*
Null	*D*	*G*	*Z*	*F*	*B*	*E*
Null	*D*	*G*	*Z*	*F*	*B*	*E*
Null	*D*	*F*	*G*	*Z*	*B*	*E*
Null	*B*	*D*	*F*	*G*	*Z*	*E*
Null	*B*	*D*	*E*	*F*	*G*	*Z*

Algorithm 1

```
a[0]   =  <a low sentinel value>
for  i   = 2 to n do
{
      temp   =  a[i];
      loc      =   i;
      while (a[loc-1] >temp) do
      {
            a[loc]   =   a[loc-1];
            loc      =    loc-1;
      }
    a[loc]   =    temp;
}
```

Algorithm 2

```
for i = 2 to n do
{
        temp  =  a[i];
        loc    =   i;
        while ((loc > 1) and (a[loc-1]>temp)) do
        {
                a[loc]   =  a[loc-1];
                loc      =    loc-1;
        }
      a[loc]   =    temp;
}
```

Analysis of Insertion Sort

Best case

Already ordered array:
The inner while loop is never entered.
Outer loop is processed n-1 times.
Thus, time complexity is O(n)

Worst Case

Array is initially in reverse order:
Inner while loop is executed i-1 times
Outer for loop is processed for i from 2 to n

$$\text{Thus, time complexity} = \sum_{i=2}^{n}(i-1) = \tfrac{1}{2}n^2 - \tfrac{1}{2}n$$
$$= O(n^2)$$

Analysis of Insertion Sort …

Average Case

For a randomly sorted array, on an average we will need to search through one-half of the sub-array to find a proper location for temp.

Thus inner while loop is executed (i-1)/2 times.

$$\text{Therefore time complexity} = \sum_{i=2}^{n} (i-1)/2 = 1/4\ n^2 - 1/4\ n$$

$$= O(n^2)$$

Selection Sort

- **Find the minimum of the unsorted Sub-array. Exchange this with first element of the sub-array.**

1	2	3	4	5	6
Z	G	E	D	B	F
B	G	E	D	Z	F
B	D	E	G	Z	F
B	D	E	G	Z	F
B	D	E	F	Z	G
B	D	E	F	G	Z

Algorithm

```
for  i = 1 to n-1 do
    {
        minindex  =  i;
        for j = i+1 to n do
         {
                if (a[j] < a[minindex])
                then minindex  =  j;
        }
     swap(a[i], a[minindex]);
    }
```

Analysis

For ordered, random or reverse ordered data:

The outer for loop executes n-1 times.

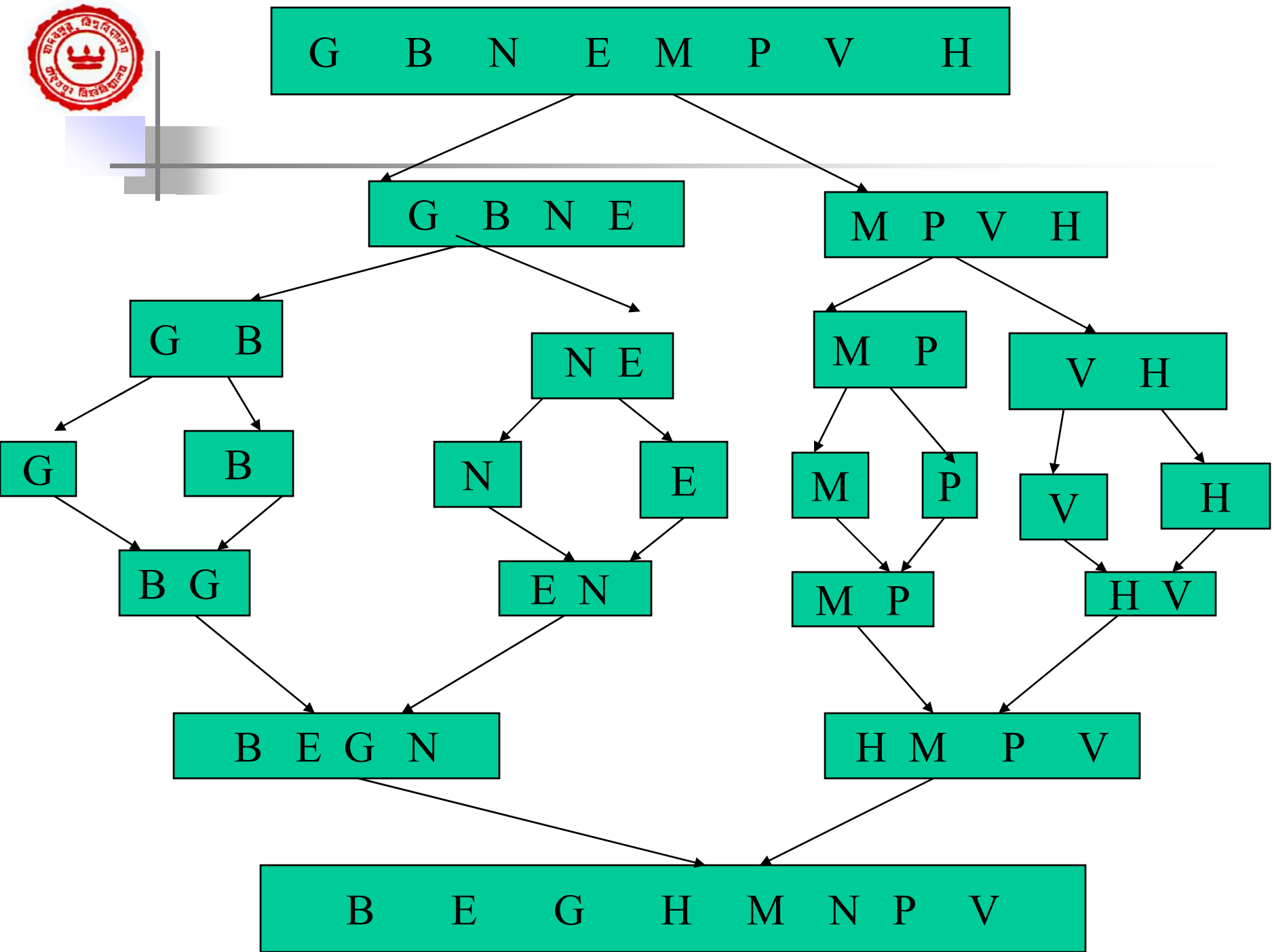
The inner for loop executes n-i times

$$\text{Time complexity} = \sum_{i=1}^{n-1}(n-i) = 1/2\ n^2 - 1/2\ n = O(n^2)$$

Merge Sort

Divide-Conquer-Recombine

Divide the array into two halves, sort both these sub-arrays using the same procedure recursively and combine the sorted sub-arrays by merging.


G B N E M P V H
G B N E
M P V H
G B
N E
M P
V H
G
B
N
E
M
P
V
H
B G
E N
M P
H V
B E G N
H M P V
B E G H M N P V

Algorithm

mergesort:

```
if (first < last )
    then
     {
          mid  =   (first + last)/2;
          mergesort(a, first, mid);
          mergesort(a, mid+1, last);
          merge(a, first, mid, last);
     }
```

Analysis

Merge sort repeatedly divides the problem in half. This generates a tree of $\lceil \log_2 n \rceil + 1$ levels.
At each level, approximately n comparisons are made to accomplish the merger of the sub-arrays.

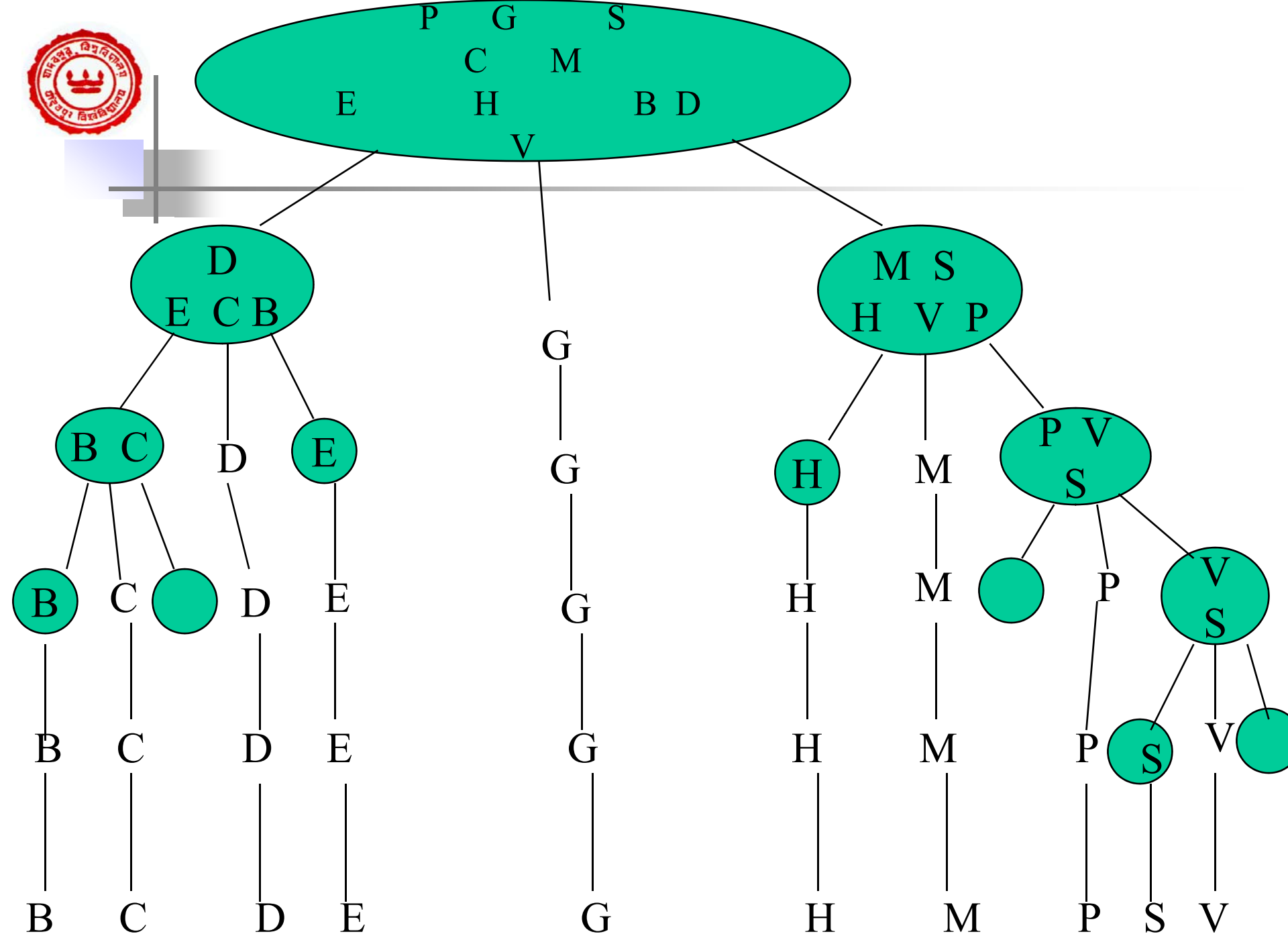
Thus, Time Complexity = O(n log n)
Regardless of the input permutations.

If n is a power of 2,
$T(2n) = 2T(n) + O(n)$, $T(2) = 1$; (recurrence relation)
Solution is $T(n) = O(n \log n)$

Quick Sort

Divide and Conquer

Select a pivot element.
Partition the array into two sub-arrays, one containing elements smaller then the pivot and the other containing elements greater then the pivot. Sort the sub-arrays using the same procedure recursively.


P G S
C M
E H B D
V
D
E C B
G
M S
H V P
B C
D
E
H
M
P V
S
B
C
D
E
G
H
M
P
V
S
B
C
D
E
G
H
M
P
S
V
B
C
D
E
G
H
M
P
S
V

Algorithm

- **Quicksort:** if (first <last)
  then{ Partition(a, first, last, loc);
  Quicksort(a, first, loc-1);
  Quicksort(a, loc+1, last);
  }

- **Partition:** i = first;
  loc = last+1;
  pivot = a[first];
  while i<loc do
  {
  repeat
  i = i+1;
  until (a[i] >= pivot);
  repeat
  loc = loc-1;
  until (a[loc] ≤ pivot);
  if (i<loc)
  then swap(a[i], a[loc]);
  }
  swap (a[first], a[loc]);

Analysis

Running time depends on input permutation and pivot selection.

If the pivot always partitions the array into two equal parts,

$T(n) = 2T(n/2) + O(n)$. $T(2)=1$;

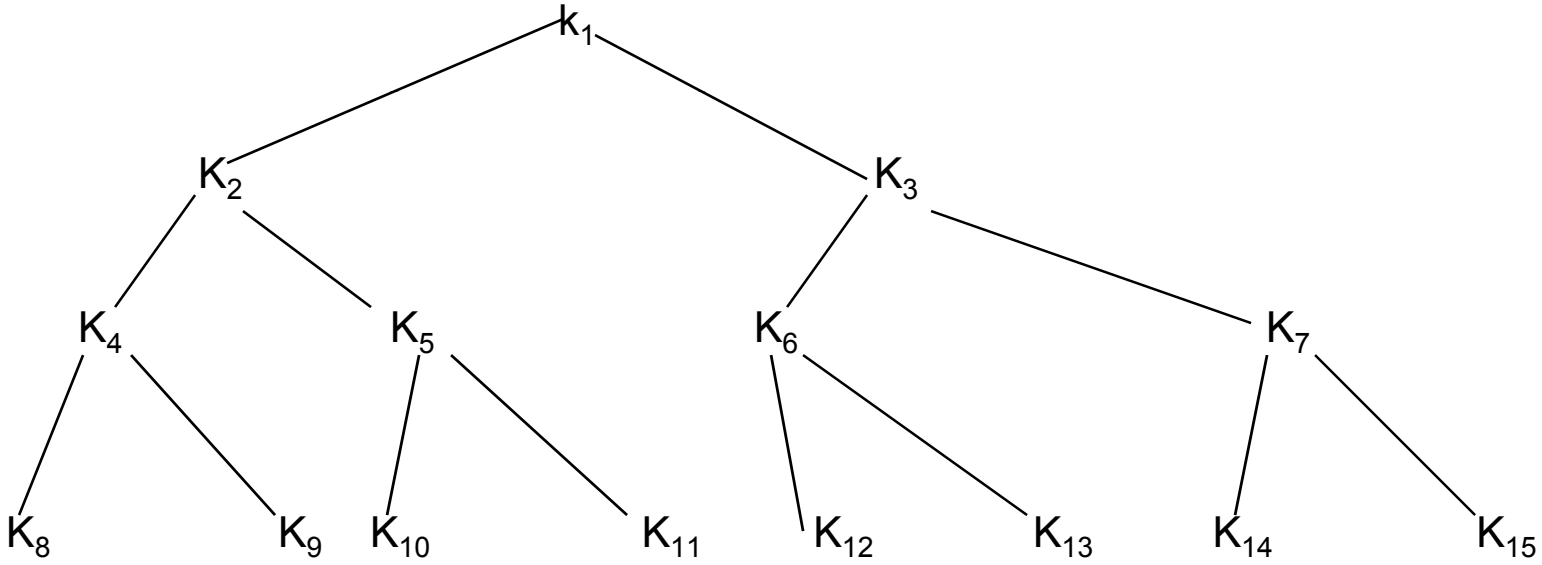
Solution is $T(n)=O(n \log n)$

If the array is already sorted and pivot is a[first], Time complexity becomes $O(n^2)$!!!(Quicksort becomes "Slowsort" - Remedy??)

Heap

A heap is a sequence of elements $K_1, K_2 \ldots\ldots, K_n$
where $K_i \geq K_{2i}$ & $K_i \geq K_{2i+1}$ $\quad 1 \leq i \leq n$ $\quad$ (Maxheap)

A sequence can be arranged as a tree:-



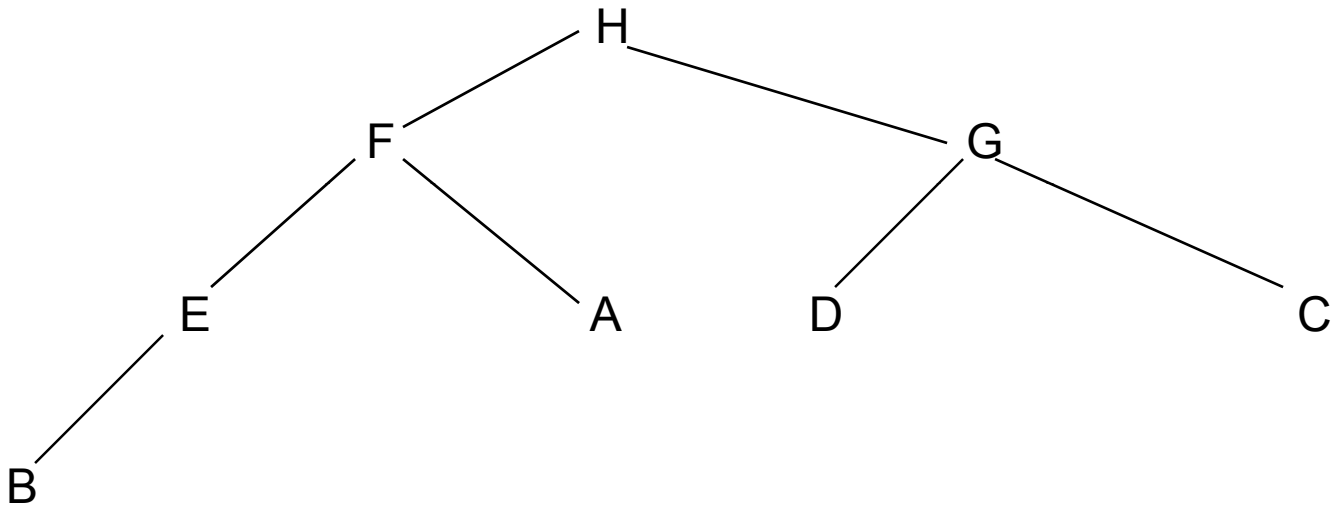
The heap property : the value of each node is greater than or equal to its children.

Heap Sort

If the input array is arranged as a heap, the first element of the array is the largest. We exchange the first and last element. Thus, now the last element is in proper position. We rearrange the new heap consisting of the first through the last but one element. Heap Sort is an iteration of this procedure.

Heap Example

H F G E A D C B



Algorithm

```
Percolate –down (A, i, N):
For (tmp = A[i];  leftchild(i) < N;  i = child) {
        child = leftchild(i);
        if (child != N-1 && A[child+1] > A[child])
                child++;
        if (tmp < A[child])
                A[i] = A[child];
        else    break;
        }
        A[i] = tmp;
```

Algorithm…

Heapsort:

```
for (i = N/2; i>=0; i--)
        Percolate-down(A, i, N);  // Build-heap

for (i = N-1; i > 0; i--) {
        Swap (A[0], A[i]);

        Percolate-down(A, 0, i);//rearrange heap
}
```

Analysis

- Percolate-down requires O(log N) iterations

- Bulidheap iterates O(N) times
- Time complexity of Buildheap is O(NlogN)

- Swapping max with the last is done O(N) times
- Each rearrange heap takes O(logN) time
- Time complexity of second for loop is O(NlogN)

- Time complexity of Heapsort is O(NlogN)

Bucket Sort

- Input array A consists of only positive integers smaller than M
- Keep an integer array *Count* of size M, initialized with all 0s
- Read array element A[i] and increment *Count*[A[i]]
- After all input array elements are read, scan the array *Count* and write *Count*[i] no. of element i in A
- Requires O(M) extra space
- Time complexity is O(M+N)
- If M is O(N), total time complexity is O(N)

Other issues

- For sorting large structures, swapping may take a lot of time
- Use pointers to such structures and swap the pointers instead of the actual structures

- Stable Sort: Sorting algorithms where input data elements with the same value appear in the output array in the same order as they do in the input array, are called Stable sorting algorithms.
- Stable sorting is required while sorting on multiple keys.