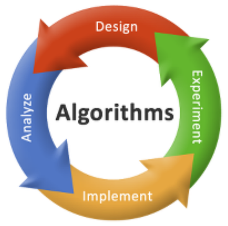
Sorting – Bucket Sort & Heap Sort

Course: Algorithms



Faculty: Dr. Rajendra Prasath

Autumn 2018

Sorting – Bucket Sort and Heap Sort

This lecture covers two sorting algorithms for sorting a set of n elements. The first one is the Bucket sort and the second one is the Heap sort. We present the algorithms and their complexity analysis. We also discuss the suitable data structures for each of the algorithm.

Recap: Sorting Algorithms

- Suggest a simple algorithm for Sorting n elements
  - Correctness: First Test whether will the algorithm work for a small set of the input? Then apply on a bigger set

  - Choose a suitable data structure
  - Perform complexity analysis
    - How much space and time required in the worst case?

  - Is the solution adaptable?
    - With the growing size of the input n

- How to get the tight bound of the sorting algorithm in terms of the running time?

Recap: Complexity Analysis

- Different Sorting Algorithms

Best	Best	average	Worst	Space
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(1)$

Bucket Sort

- called as Bin Sort (Why??)

- **How does it work?**
  - Distribute the elements into a number of buckets
  - Sort each bucket efficiently

- It is a Distribution Sort
- Can also be considered as a comparison sort

- Complexity depends on number of buckets

Bucket Sort

- Bucket Sort can be applied if the following two properties hold:
  - **Uniform distribution**
    - The input data must be uniformly distributed for a given range. Based on this distribution, n buckets are created to evenly partition the input range.
  - **Ordered hash function**
    - The buckets must be ordered. That is, if i<j, then elements inserted into bucket bi are lexicographically smaller than elements in bucket bj.

Bucket Sort – Basic Idea

- **Basic Steps**
  - Consider a set of empty **buckets**
    - Range of elements and Size of the bucket??
  - **Scatter:** From the given set of the input, put each element in the corresponding bucket
  - **Sort** the elements in each non-empty bucket
  - **Gather:** Each bucket is visited in order and collect all sorted elements and generate a sorted set of n elements (with respect to the given partial order)

Bucket Sort – Algorithm

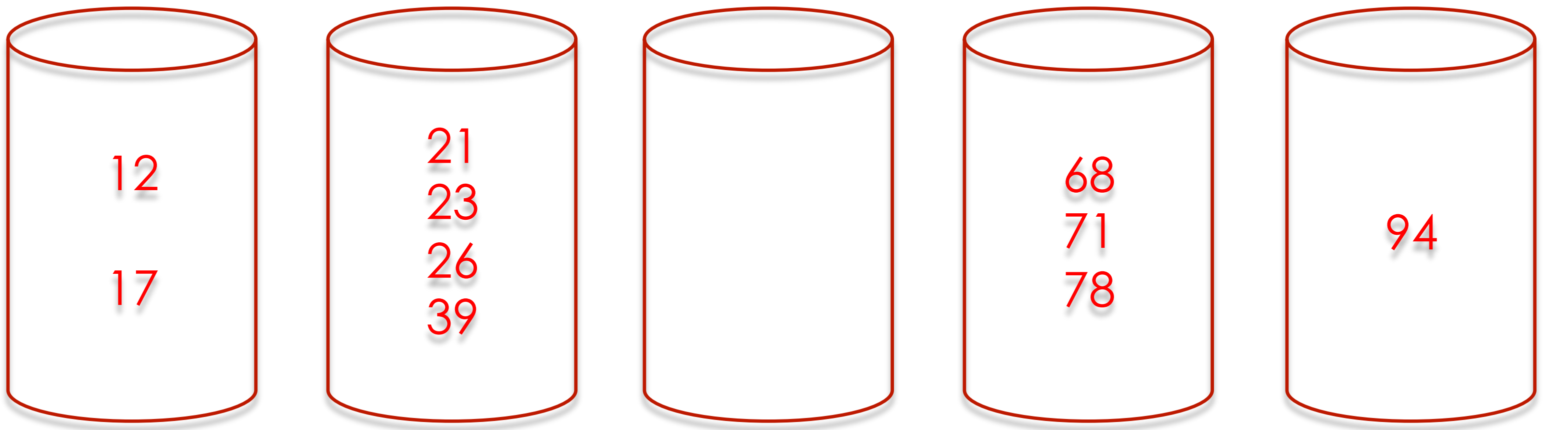
```
procedure **BucketSort()**
    Input Sequence, say A

    Begin
        Create n empty buckets (Or lists)
        Do following for every array element A[i]
        Insert a[i] into bucket[n*A[i]]
        Sort individual buckets using insertion sort
        Concatenate all sorted buckets
    End
```

Bucket Sort – Example

- **Input sequence to be sorted:**

78	17	39	26	71	94	21	12	23	68



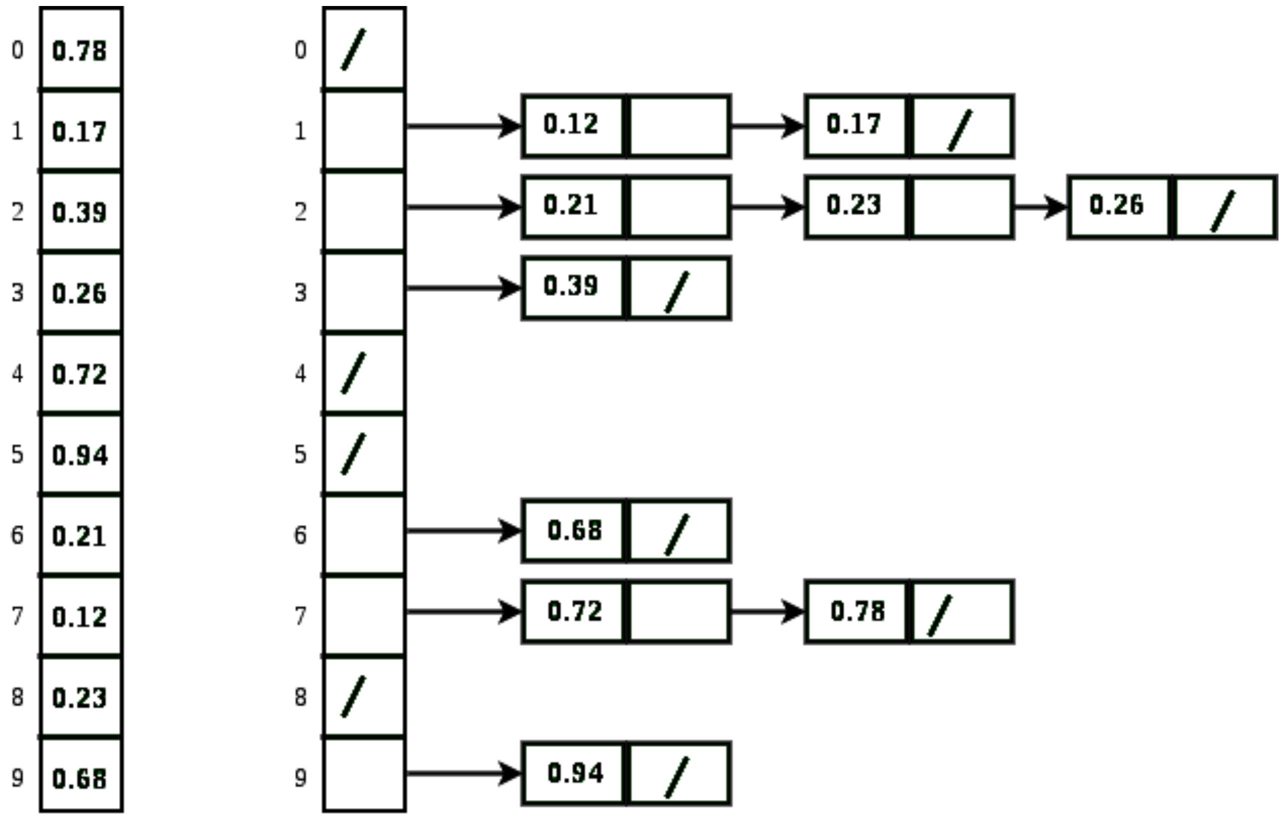
- **Sorted Sequence:**

12	17	21	23	26	39	68	71	78	94

Bucket Sort – Illustration

- **Sort:**

0.78	0.17	0.39	0.26	0.72	0.94	0.21	0.12	0.23	0.68



Bucket Sort – Facts

- **Is this the Fastest Sorting algorithm:**
  - When??
    - When the elements to be sorted can be uniformly partitioned using a fast hashing function
- **Stable:**
  - Yes provided the underlying sorting algorithm is stable.
- Bucket Sort is not appropriate for sorting arbitrary strings (why?)

Bucket Sort – Complexity

- Best Case
  - O(n)
- Average Case
  - O(n)
- Worst Case
  - $O(n^2)$
- Data Structures:
  - Arrays, Hashes, Adjacency Lists

Heap Sort - Introduction

- Comparison Based Sorting
- Thought of as an improved selection sort
  - Using "Heaps" rather than linear time search to find the maximum (or minimum) element

- **Basic Idea**
  - Two Step sorting algorithm
    - Build a heap from the given n elements
    - Sort the elements by removing the largest element from the heap

Heaps

- What is a HEAP?
  - A complete binary tree (balanced or unbalanced)
  - Ordering property
    - How to define this ordering?
  - Different types of ordering?

- A binary heap
  - A COMPLETE binary tree which satisfies the heap ordering property

Binary Tree → Heaps

- When does a binary tree become a heap?

  - It should be complete
  - Ordering property

- Value(root) ≤ values(children)

  → → Min Heap

- Value(root) ≥ values(children)

  → → Max Heap

Heap Property

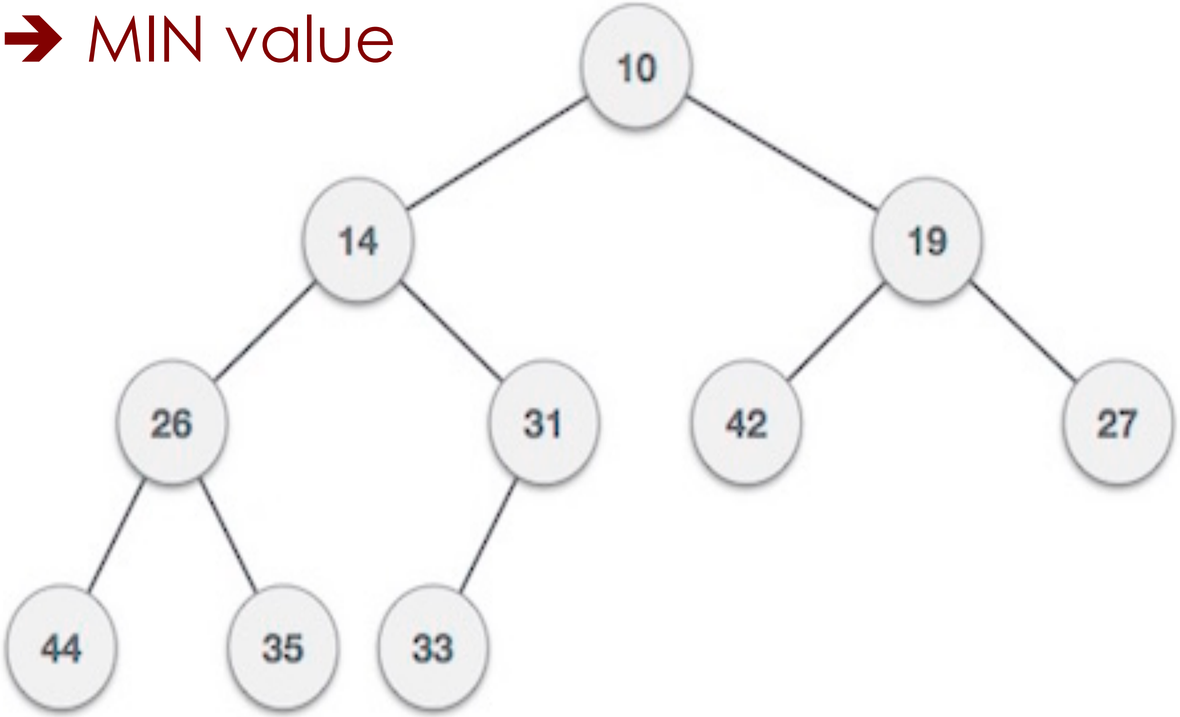
- A HEAP is a complete binary tree
  - Has a smallest possible height – How?
  - A heap with n nodes ➔ O(log n) height

- Heap Property ?
  - To order elements in the heap
  - How to define Heap Property ?
  - Is ordering WELL – DEFINED??

- Two Types:
  - min-heap property
  - max-heap property

Min-Heap

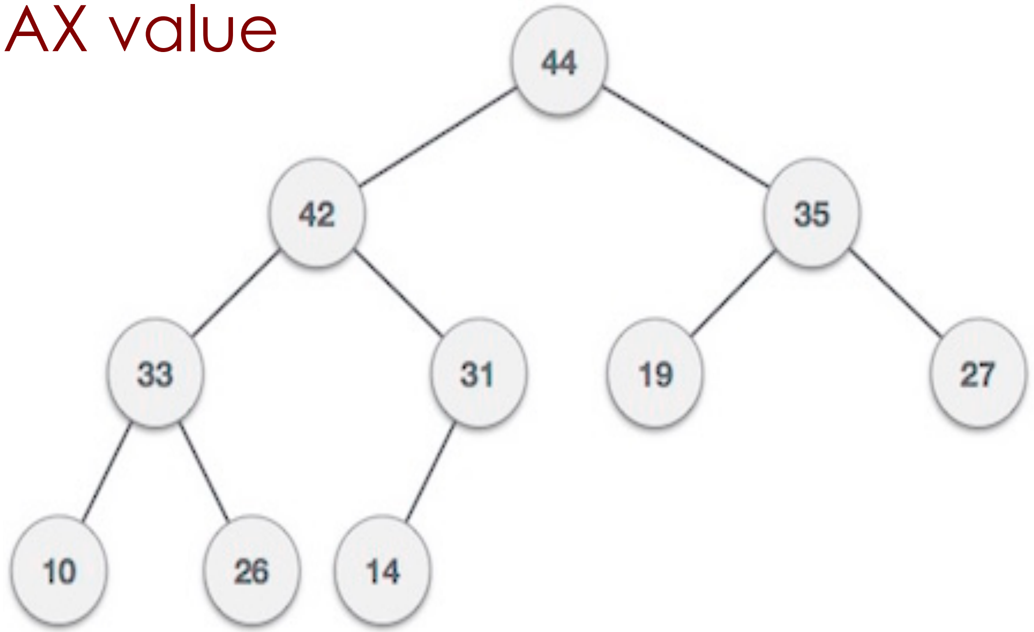
- Min-Heap Property ?
- Value of each node
  ≥ value of its parent.
- Root ➔ MIN value



Max-Heap

- Max-Heap Property ?
- Value of each node

  $\leq$ value of its parent.
- Root $\rightarrow$ MAX value



Building Heaps

- How to build Heaps from the given array or list of n elements

- MAX-Heap Property ?

- Check the following property
  - Value of each node ≤ value of its parent.
  - root ➔ MAX value

BuildHeaps: Steps

- We assume that we build MAX-heap (can be MIN-heap as well)

- MAX-Heap
  - Create a new node at the end of heap
  - Assign new value to the node
  - Compare the value of this child node with its parent
  - If value of parent < the value of the child node, then swap them
  - Repeat above two steps until the Heap property holds

Heap Sort – Algorithm

Given an array A of n elements (e.g., integers):

Algorithm HeapSort

- Build Heap
- Heapify

- Sort the elements based on Min or Max heap

Heapify

- Heapify picks the largest child key and compare it to the parent key. If parent key is larger than heapify quits, else it swaps the parent key with the largest child key

```
Heapify(A, i) {
    l ← left(i)
    r ← right(i)
    if l <= heapsize[A] and A[l] > A[i]
        then largest ←l
        else largest ← i
    if r <= heapsize[A] and A[r] > A[largest]
        then largest ← r
    if largest != i
        then swap A[i] ←→ A[largest]
            Heapify(A, largest)
}
```

Build Heap

- We can use the procedure 'Heapify' in a bottom-up fashion to convert an array A[1 . . *n*] into a heap. Since the elements in the subarray A[*n*/2 +1 . . *n*] are all leaves, the procedure BUILD_HEAP goes through the remaining nodes of the tree and runs 'Heapify' on each one. The bottom-up order of processing node guarantees that the subtree rooted at children are heap before 'Heapify' is run at their parent.

```
Buildheap(A) {
        heapsize[A] ← length[A]
        for i ← |length[A]/2  //down to 1
            do Heapify(A, i)
}
```

Heap Sort – Algorithm

- BUILD-HEAP to build a heap on the input array A[1 . . *n*].
- Since the maximum element of the array stored at the root *A*[1], it can be put into its correct final position by exchanging it with *A*[*n*] (the last element in *A*).
- If we now discard node n from the heap than the remaining elements can be made into heap

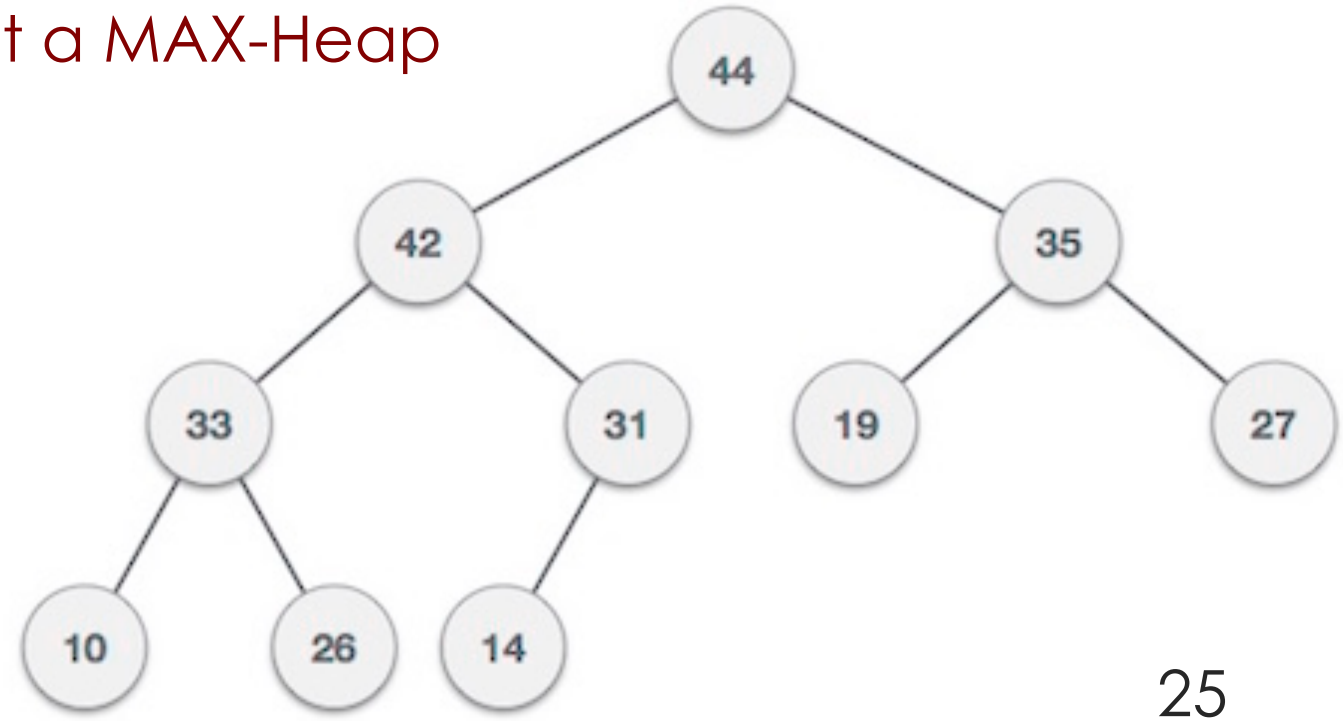
```
Heapsort(A) {
        Buildheap(A)
        for i ← length[A] //down to 2
            do swap A[1] ←→ A[i]
            heapsize[A] ← heapsize[A] - 1
            Heapify(A, 1)
}
```

Heap Sort – Example

- Consider the following elements

35 33 42 10 14 19 27 44 26 31

- Construct a MAX-Heap



Heap Sort – Implementation

- Using Array indices
  - Consider kth element of the array

  - left child→ 2*k index
  - right child → 2*k+1 index
  - Parent → k/2 index

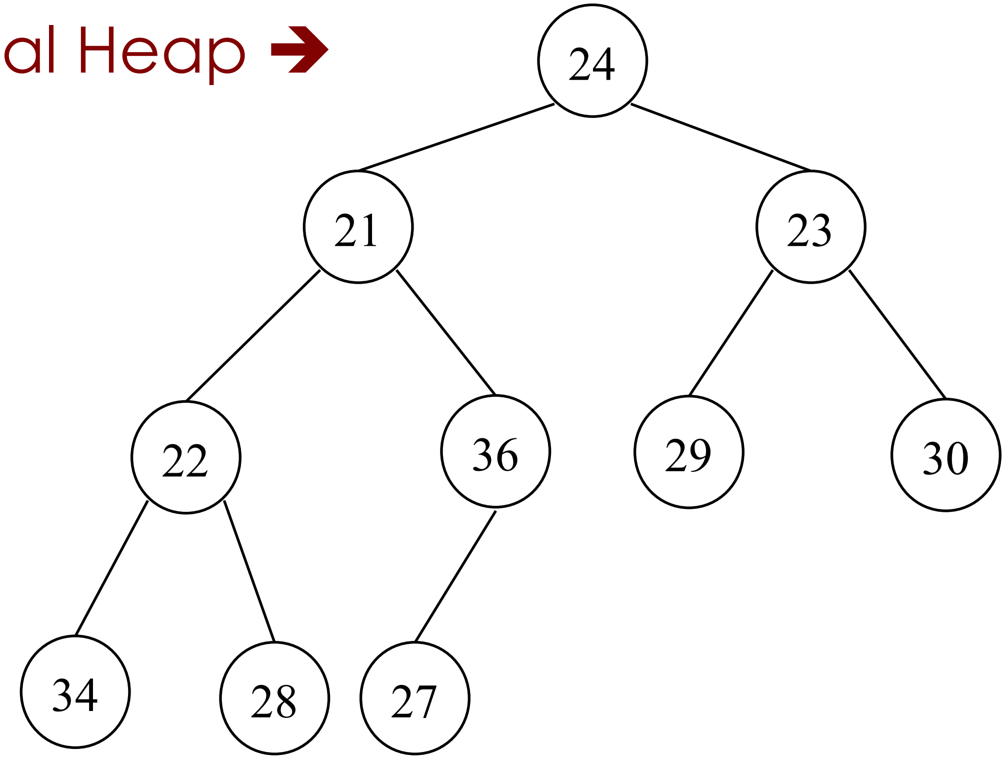
- Using a counter – Count the nodes in the heap and dynamically create a new node and insert it into heap preserving the heap property

Heap Sort – An Example

- Sort:

24	21	23	22	36	29	30	34	28	27

- Initial Heap ➔

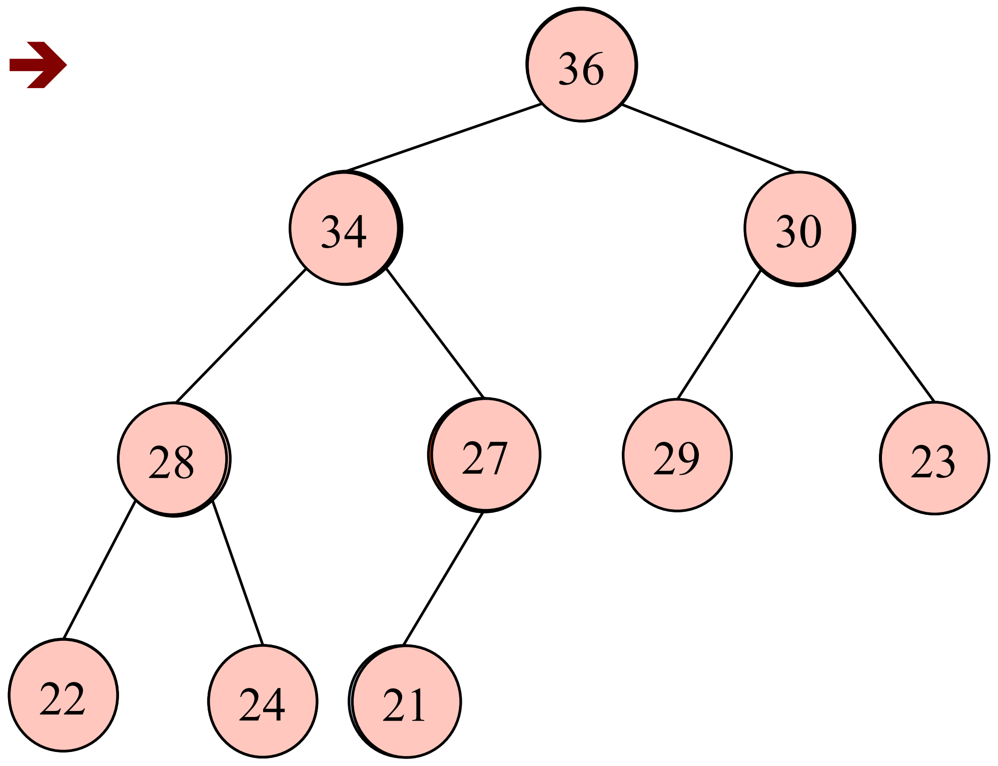


Building Max-Heap

- Sort:

24	21	23	22	36	29	30	34	28	27

- Max-Heap ➔



Heap Sort – Operations

- Insert
  - A new element into the heap preserving the heap property
- Delete a specific element
  - Delete an element – deleteMIN() / deleteMAX()
- Remove
  - One element or All elements
- Find MIN / MAX
  - Find a specific element

Applications

- Priority Queues – Goals scored by a player
  - MIN – Priority Queue
  - MAX - Priority Queue

- Order statistics: Efficiently find the kth smallest (or largest) element

- Sorting of Elements
  - Limited use … QUICKSORT is better in practice …
  - Running time is larger than Quicksort?

Heap Sort – Facts

- STABLE:
  - No, it is not a stable sorting algorithm
    - (Frequent movements of the elements)

- Take Home Assignment:
- Can you estimate the actual running time of the Heap sort algorithms with the varying n (=10k, 20k, 50k, and 100k)?

Heap Sort - Analysis

- In-place Sorting Algorithm
- Best, Average and the Worst case running time:
  - O(n logn) !!!

- How?
  - Time complexity of heapify is O(Logn)
  - Time complexity of createAndBuildHeap() is O(n)

➔ The Overall time complexity is O(nLogn)

- Data Structures: Array, list, linked list

Computational Complexity

- Different Sorting Algorithms

Best	**Best**	**average**	**Worst**	**Extra Space**
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O(n)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Quick Sort	$O(nlogn)$	$O(nlogn)$	$O(n^2)$	$O(1)$
Bucket Sort	**$O(n)$**	**$O(n)$**	**$O(n^2)$**	**$O(1)$**
Heap Sort	**$O(nlogn)$**	**$O(nlogn)$**	**$O(nlogn)$**	**$O(1)$**

Help among Yourselves?

- **Perspective Students** (having CGPA above 8.5 and above)
- **Promising Students** (having CGPA above 6.5 and less than 8.5)

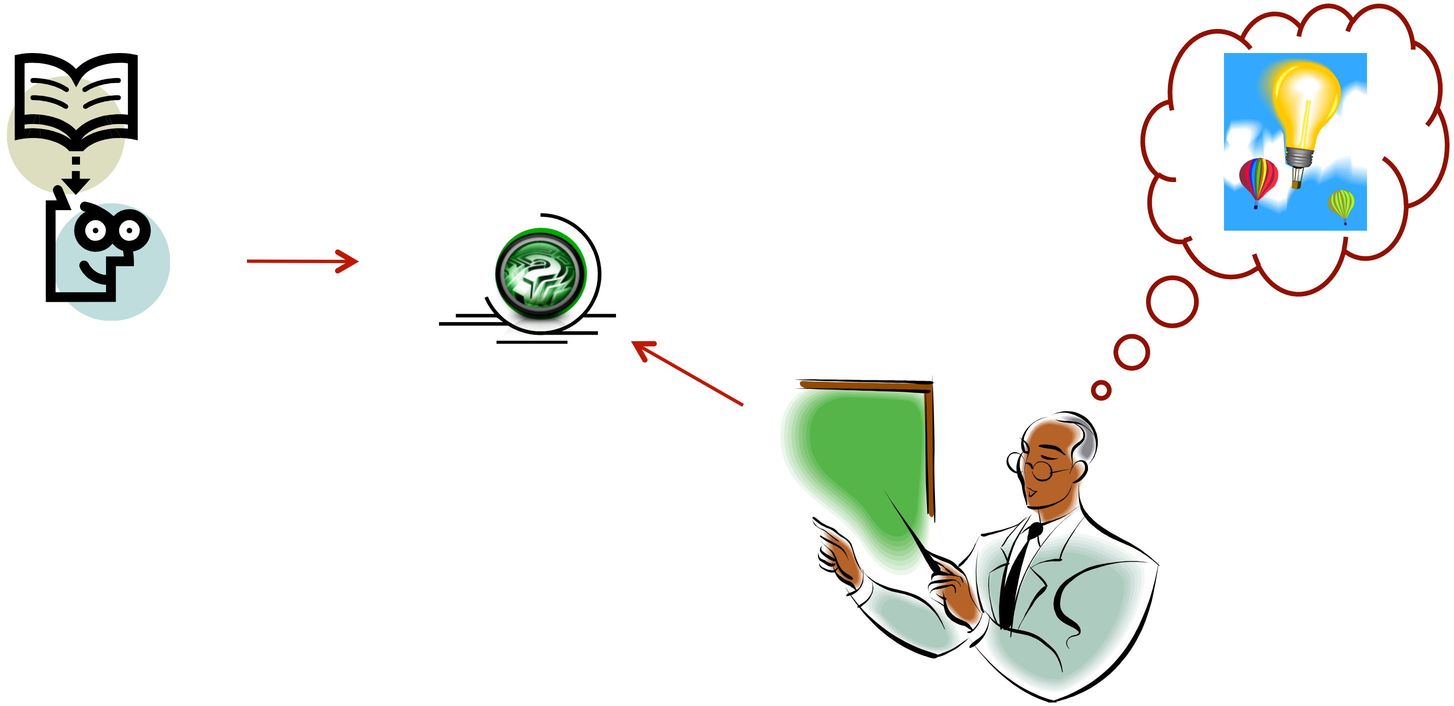
- **Needy Students** (having CGPA less than 6.5)
  - Can the above group help these students? (Your work will also be rewarded)

- You may grow a culture of **collaborative learning** by helping the needy students

Assistance

- You may post your questions to me at any time
- You may meet me in person on available time or with an appointment
- TA s would assist you to clear your doubts.
- You may leave me an email any time (email is the best way to reach me faster)

Thanks ...



... Questions ???