# MA 515: Introduction to Algorithms & MA353 : Design and Analysis of Algorithms [3-0-0-6]

**Lecture 11**

http://www.iitg.ernet.in/psm/indexing_ma353/y09/index.html

**Partha Sarathi Manal**

psm@iitg.ernet.in

Dept. of Mathematics, IIT Guwahati

Mon 10:00-10:55 Tue 11:00-11:55 Fri 9:00-9:55
Class Room : 2101

# Heap Sort

***Input*:** An *n*-element array A (unsorted).

***Output*:** An *n*-element array A in sorted order, smallest to largest.

**HeapSort(A)**

1. **Build-Max-Heap(A)** /* put all elements in heap */
2. **for** *i* ← length(A) **downto** 2
3. **do** swap A[1] ↔ A[*i*]/*puts max in *i*-th array position*/
4. heap-size[A] ← heap-size[A] - 1
5. **Max-Heapify(A,1)** /* restore heap property */

# Heap Sort

**HeapSort(A)**

1. **Build-Max-Heap(A)**
2. **for** i ← length(A) **downto** 2
3. **do** swap A[1] ↔ A[i]
4. heap-size[A] ← heap-size[A] – 1
5. **Max-Heapify(A,1)**

**Build-Max-Heap(A)**

1. heapsize(A) ← length(A)
2. for $i \leftarrow \lfloor length(A)/2 \rfloor$ downto 1
3. do Max-Heapify(A, i)

**Max-Heapify(A,i)**

0. left ← 2i
1. right ← 2i + 1
   ▶indices of left & right children of A[i]
2. largest ← i
3. if left ≤ heapsize(A) and A[left] > A[i]
4. then largest ← left
5. if right ≤ heapsize(A) and A[right] > A[largest]
6. then largest ← right
7. if largest ≠ i
8. then exchange A[i] ↔ A[largest])
9. Max-Heapify(A, largest)

# Max-Heapify: Maintaining the Heap Property

- Max-Heapify(A, i)

  Assumption: subtrees rooted at left and right children of A[i] are max-heaps and the roots of these subtrees are A[2i] and A[2i + 1].

- ...but subtree rooted at A[i] might not be a max-heap (that is, A[i] may be smaller than its left or right child)

- Max-Heapify(A, i) will cause the value at A[i] to "float down" in the heap so that subtree rooted at A[i] becomes a max-heap.

# Max-Heapify: Maintaining the Heap Property

```
Max-Heapify(A,i)
0.  left ← 2i
1.  right ← 2i + 1
      ▶indices of left & right children of A[i]
2.  largest ← i
3.  if  left ≤ heapsize(A) and A[left] > A[i]
4.        then largest ← left
5.  if  right ≤ heapsize(A) and A[right] > A[largest]
6.        then largest ← right
7.  if  largest ≠ i
8.        then exchange A[i] ↔ A[largest])
9.        Max-Heapify(A, largest)
```

# Max-Heapify: Running Time

Running Time of Max-Heapify:

- Every line is $\Theta(1)$ time -- except the recursive call
- In the worst-case of the recursion, Max-Heapify takes $O(h)$ time when node A[$i$] has height $h$ in the heap, therefore
- $T(n) = O(\lg n)$

# Build-Max-Heap

- **Intuition:** uses Max-Heapify in a bottom-up manner to convert unordered array A into a heap.
- Key point is that the leaves are already heaps. Elements A[($\lfloor n/2 \rfloor$ + 1) ... n] are all leaves.
- So the work starts at parents of leaves...then, grandparents of leaves...etc.

```
Build-Max-Heap(A)
1. heapsize(A) ← length(A)
2. for i ← ⌊length(A)/2⌋ downto 1
3.     do Max-Heapify(A, i)
```

# Build-Max-Heap

**Running Time of Build-Max-Heap**

- Approximately $n/2$ calls to Max-Heapify ($O(n)$ calls)
- Simple upper bound: Each call takes $O(\lg n)$ time & $O(n \lg n)$ time total.
- Is it possible to make some tighter bound for Build-Max-Heap?
- What is the tighter bound for Build-Max-Heap ?
- Answer is $O(n)$.
- Now question is how ?

```
Build-Max-Heap(A)
1. heapsize(A) ← length(A)
2. for i ← ⌊length(A)/2⌋ downto 1
3.     do Max-Heapify(A, i)
```

# Build-Max-Heap

- Proof of tighter bound ($O(n)$) relies on following theorem:
- **Theorem 1:** The number of nodes at height $h$ in a maxheap $\lceil n/2^{h+1} \rceil$.

  Height of a node = longest distance from a leaf.

  Depth of a node = distance from the root.

  – Let H be the height of the tree. If the heap is not a complete binary tree (because the bottom level is not full), then the nodes at a given depth don't all have the same height. Eg., although all the nodes with depth H have height 0, the nodes with depth H-1 may have either height 0 or 1.

```
Build-Max-Heap(A)
1. heapsize(A) ← length(A)
2. for i ← ⌊length(A)/2⌋ downto 1
3.     do Max-Heapify(A, i)
```

**Lemma 1:** The # of internal nodes in a proper binary tree is equal to the # of leaves in the tree - 1.

Defn: In a *proper binary tree*, each node has exactly 0 or 2 children.

Let *I* be the # of internal nodes and let *L* be the # of leaves in a proper binary tree. The proof is by induction on the *height of the tree*.

**Basis:** h=0. I = 0 and L = 1. Now I = L - 1 = 1 - 1 = 0, so the lemma holds.

**Inductive Step:** Assume true for proper binary trees of height *h* and show for proper binary trees of height *h* + 1.

Consider a proper binary tree T of height *h*+1. It has left and right subtrees (L and R) of height at most *h*.

$I_T = (I_L + I_R) + 1 = (L_L - 1) + (L_R - 1) + 1 =$

$(L_L + L_R - 2) + 1 = L_L + L_R - 1$. Since $L_T = L_L + L_R$ we have that $I_T = L_T - 1$.

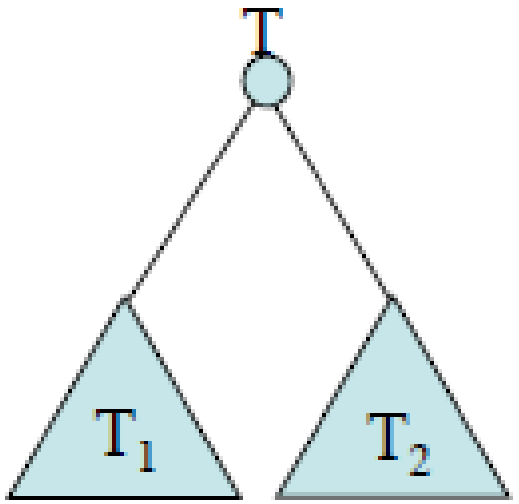
#Internal nodes in T

= #Internal nodes in $T_1$ + #Internal nodes in $T_2$ + 1

= (#Leaves in $T_1$ - 1) + (#Leaves in $T_2$ -1) + 1

= (#Leaves in $T_1$ + #Leaves in $T_2$) - 2 + 1

=#Leaves in T - 1



**Theorem :** The number of nodes at height $h$ in a maxheap $\lceil n/2^{h+1} \rceil$.

***Proof*:** Let H be the height of the heap. Proof is by induction on $h$, the height of each node. The number of nodes in the heap is $n$.

**Basis:** Show the thm holds for nodes with $h = 0$. The tree leaves (nodes at height 0) are at depths H and H-1.

Let $x$ be the # of nodes at depth H, that is, the # of leaves assuming that $n$ is a complete binary tree, i.e. $n = 2^{h+1}-1$

Note that $n-x$ is odd, because a complete binary tree has an odd # of nodes.

# Proof of the *Thm.*

If $n$ is odd, $x$ is even, so all nodes have siblings (all internal nodes have 2 children.)
By *Lemma 1*, the # of internal nodes = the # of leaves - 1.

So $n$ = # of nodes = # of leaves + # internal nodes = 2(# of leaves) - 1. Thus, the # of leaves = $(n+1)/2 = \lceil n/2^{0+1} \rceil$ because $n$ is odd.

Thus, the # of leaves = $\lceil n/2^{0+1} \rceil$ and the thm holds for the base case.

# Cont… Proof of the *Thm*.

**Inductive step**: Show that if the thm holds for height $h$-1, it holds for $h$.

Let $n_h$ be the number of nodes at height $h$ in the $n$-node tree $T$.

Consider the tree T' formed by removing the leaves of T. It has $n' = n - n_0$ nodes. We know from the base case that $n_0 = \lceil n/2 \rceil$, so $n' = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$.

Note that the nodes at height h in T would be at height $h$-1 if the leaves of the tree were removed--i.e., they are at height $h$-1 in T'. Letting $n'_{h-1}$ denote the number of nodes at height $h$-1 in T', we have $n_h = n'_{h-1}$

$n_h = n'_{h-1} \leq \lceil n'/2^h \rceil$ (by the IHOP) $= \lceil \lfloor n/2 \rfloor / 2^h \rceil \leq \lceil (n/2)/2^h \rceil = \lceil n/2^{h+1} \rceil$

# Cont… Proof of the *Thm.*

- Since the time of Max-Heapify when called on a node of height $h$ is $O(h)$, the time of B-M-H is

$$\sum_{h=0}^{\lg n} \frac{n}{2^{h+1}} O(h) = O(n \sum_{h=0}^{\lg n} \frac{h}{2^h})$$

- and since the last summation turns out to be a constant, the running time is $O(n)$.
- Therefore, we can build a max-heap from an unordered array in linear time.

# Running time of HeapSort

We'll see that

- *Build-Max-Heap*(A) takes
  $O(|A|) = O(n)$ time
- *Max-Heapify*(A,1) takes
  $O(\lg|A|) = O(\lg n)$ time

Running time of *HeapSort*:

- One call to *Build-Max-Heap*()
  $\Rightarrow O(n)$ time
- *n-1* calls to *Max-Heapify*() each takes $O(\lg n)$ time
  $\Rightarrow O(n\lg n)$ time

# Heap Sort

**HeapSort(A)**

1. **Build-Max-Heap(A)** // O(*n*)
2. **for** i ← length(A) **downto** 2 //O(*n*)
3. **do** swap A[1] ↔ A[i] //O(n)
4. heap-size[A] ← heap-size[A] – 1 //O(n)
5. **Max-Heapify(A,1)** // O(*n*lg *n*)

**Build-Max-Heap(A)**

1. heapsize(A) ← length(A)
2. for *i* ← ⌊*length*(A)/2⌋ downto 1
3. do Max-Heapify(A, i)

**Max-Heapify(A,i)**

0. left ← 2i
1. right ← 2i + 1
   ▶indices of left & right children of A[i]
2. largest ← i
3. if left ≤ heapsize(A) and A[left] > A[i]
4. then largest ← left
5. if right ≤ heapsize(A) and A[right] > A[largest]
6. then largest ← right
7. if largest ≠ i
8. then exchange A[i] ↔ A[largest])
9. Max-Heapify(A, largest)

# Heapsort Time and Space Usage

- An array implementation of a heap uses $O(n)$ space
  - one array element for each node in heap.
- Heapsort uses $O(n)$ space and is **in place.**
- Running time is as good as merge sort, $O(n \lg n)$ in worst case.

# Inserting Heap Elements

- **Inserting an element into a heap:**
  - increment heapsize and add new element to the highest numbered position of array.
  - walk up tree from new leaf to root, swapping values. Insert input key when a parent key larger than the input key is found.

```
Max-Heap-Insert(A, key)
1. heapsize(A) ← heapsize(A) +1
2. i ← heapsize(A)
3. while i > 1 and A[parent(i)] < key
4.       do A[i] ← A[parent(i)]
5.          i ← parent(i)
6. A[i] ← key
```

- **Running time of Max-Heap-Insert:**
  - $O(\lg n)$, time to traverse leaf to root path (height = $O(\lg n)$)

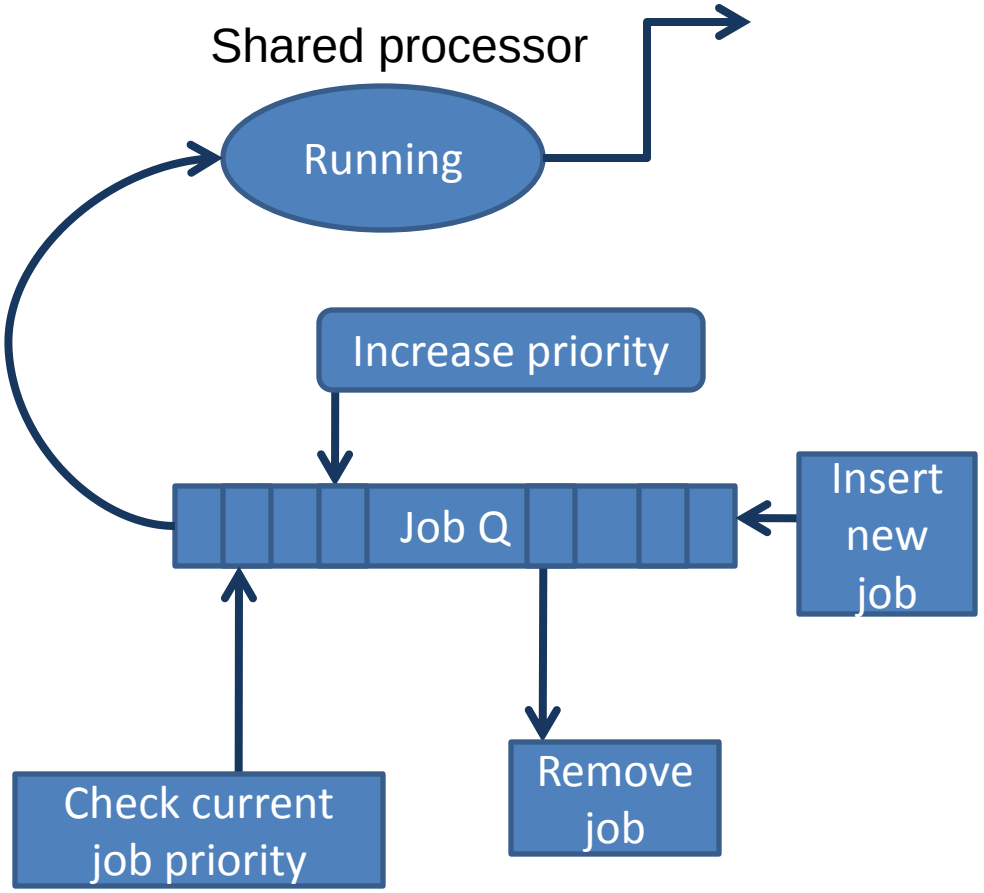
# Priority Queues

- **Definition:** A priority queue is a data structure for maintaining a set S of elements, each with an associated key.
- A max-priority-queue gives priority to keys with larger values and supports the following operations.
  1. *insert(S, x)* inserts the element x into set S.
  2. *max(S)* returns element of S with largest key.
  3. *extract-max(S)* removes and returns element of S with largest key.
  4. *increase-key(S,x,k)* increases the value of element x's key to new value k (assuming k is at least as large as current key's value).

# Priority Queues: Application for Heaps

An application of max-priority queues is to schedule jobs on a shared processor. Need to be able to

- **check current job's priority**
  **Heap-Maximum(A)**
- **remove job from the queue**
  **Heap-Extract-Max(A)**
- **insert new jobs into queue**
  **Max-Heap-Insert(A, key)**
- **increase priority of jobs**
  **Heap-Increase-Key(A,i,key)**



# Priority Queues: Application for Heaps

An application of max-priority queues is to schedule jobs on a shared processor. Need to be able to

| | |
|---|---|
| **check current job's priority** | **Heap-Maximum(A)** |
| **remove job from the queue** | **Heap-Extract-Max(A)** |
| **insert new jobs into queue** | **Max-Heap-Insert(A, key)** |
| **increase priority of jobs** | **Heap-Increase-Key(A,i,key)** |

Initialize PQ by running Build-Max-Heap on an array A.

A[1] holds the maximum value after this step.

**Heap-Maximum(A)** - returns value of A[1].

**Heap-Extract-Max(A)** - Saves A[1] and then, like Heap-Sort, puts item in A[heapsize] at A[1], decrements heapsize, and uses Max-Heapify(A, 1) to restore heap property.

# Heap-Increase-Key

**Heap-Increase-Key(A, i, key)** - If key is larger than current key at A[i], floats inserted key up heap until heap property is restored.

An application for a min-heap priority queue is an event driven simulator, where the key is an integer representing the number of seconds (or other discrete time unit) from time zero (starting point for simulation).