CSC263H
Data Structures and Analysis

Prof. Bahar Aameri & Prof. Marsha Chechik

Winter 2024 – Week 2

ADT: Priority Queues

Queue:

- **Objects**: a collection of elements.
- **Operations**: *Enqueue*(Q, x), *Dequeue*(Q), *PeekFront*(Q).

Priority Queue:

- **Objects**: a set of elements, where each element has a **priority**.
- **Operations**:
  - *Insert*(PQ, x, p): *Add* x to the priority queue PQ with the priority p.
  - *FindMax*(PQ): *Return* the item in PQ with the *highest* priority.
  - *ExtractMax*(PQ): *Remove and return* the item from PQ with the *highest* priority.

    $k \geq x.p$
  - *IncreaseKey*(PQ, x, k): *Increases* the priority value p of the element x to the new value k (k assumed to be at least as large as p).

Applications of Priority Queues

- **Hospital Waiting Room:** More severe injuries and illnesses are generally treated before minor ones

- **Job Scheduling in Operating Systems**

- **Printer Queues**

- **Event-Driven Simulation Algorithms**

Unsorted List:

- *Insert*(PQ, x, p): $\Theta(1)$
- *FindMax*(PQ): $\Theta(n)$
- *ExtractMax*(PQ): $\Theta(n)$
- *IncreaseKey*(PQ, x, k): $\Theta(1)$ (assuming we know where x is placed)

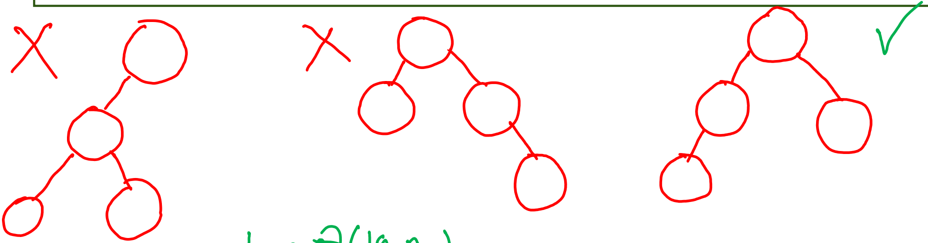
Sorted List (by priorities):

- *Insert*(PQ, x, p): $\Theta(n)$
- *FindMax*(PQ): $\Theta(1)$
- *ExtractMax*(PQ): $\Theta(1)$
- *IncreaseKey*(PQ, x, k): $\Theta(n)$

Data Structures for Priority Queues: Heaps

- **Complete Binary Tree:** A binary tree is complete iff it satisfies the following two properties:
  1. All of its levels are **full**, *except* possibly the **bottom** one.
  2. All of the nodes in the bottom level are **as far to the left as possible**.

There is only *one* complete tree *shape* for *each number of nodes*.



$h \in \Theta(\lg n)$

Data Structures for Priority Queues: Heaps

- **Max-Heap Property**: A tree satisfies the max-heap property iff for each node in the tree, the value of that node is greater than or equal to the value of all of its **descendants**.

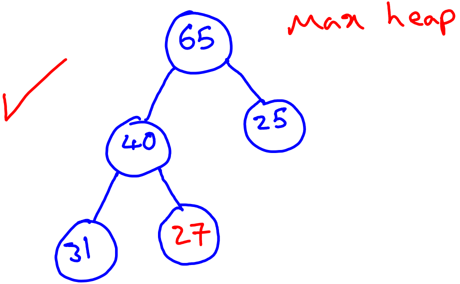
- **Min-Heap Property**: A tree satisfies the min-heap property iff for each node in the tree, the value of that node is less than or equal to the value of all of its **descendants**.

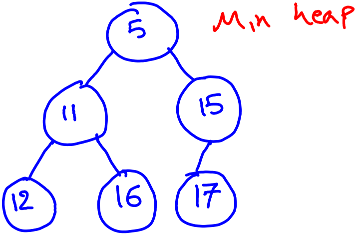
- **Max-Heap**: A complete binary tree that satisfies the max-heap property.

- **Min-Heap**: A complete binary tree that satisfies the min-heap property.

- **Implication**: Every *sub-tree* of a max-heap/min-heap is also a *max-heap/min-heap*.

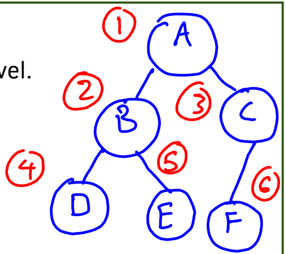
Data Structures for Priority Queues: Heaps





Storing a heap in an **array**:

- **Level Order Traversal**: from *left to right*, level by level.



For a node corresponding to index i
(assuming the items are stored starting at ***index 1***):

- *left child* is stored at index $2i$
- *right child* is stored at index $2i+1$
- *parent* is stored at index $\left\lfloor \frac{i}{2} \right\rfloor$

1. Items are stored in an array A.

2. Each item x has a key $x.p$ which represents its priority (x may have other fields).

3. A is a max heap based on priorities of its items.

Note: To simplify examples, from now on we assume that the only field a heap item has is its key (i.e., p).
This way we can assume that A stores only numerical values representing the keys.

FindMax(PQ): Return the item in PQ with the highest priority.

- *HeapMaximum*(A): Return the root of A.

return A[1]

- **Worst-Case Running Time:** $\Theta(1)$

Heaps: Implementing *IncreaseKey*

IncreaseKey(PQ,x,k): Increases the priority value p of the element x to the new value k (k assumed to be at least as large as p).

HeapIncreaseKey(A, i, k) (assuming that i is the index of x in the array):

1. Set the priority of x (stored at $A[i]$) to k.
2. Bubble-up x to a proper position, by *swapping* with parent until k is *not greater* than priority of parent of x.

Worst-case Running Time: $\Theta(\lg n)$

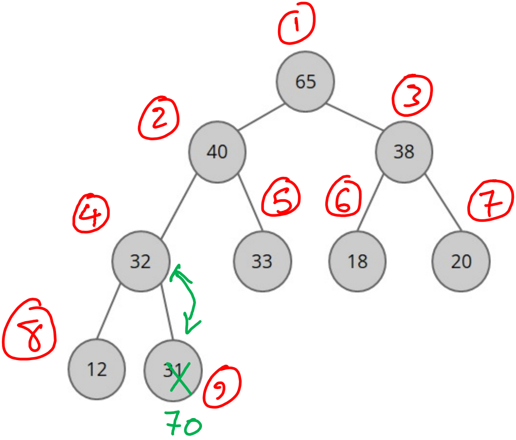
move a leaf all the way to the root.

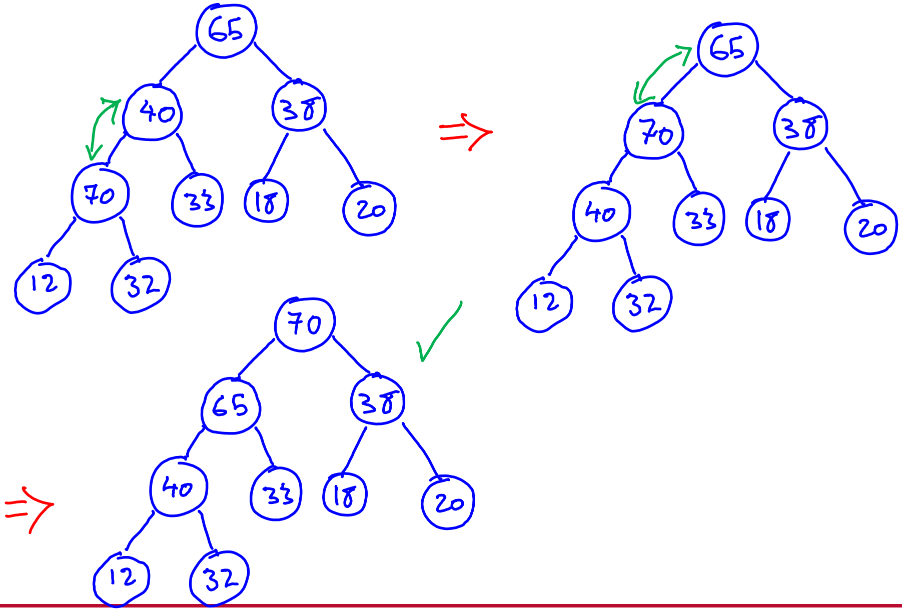
$\Theta(h) = \Theta(\lg n)$

h: height of the heap

n: number of nodes in the heap

HeapIncreaseKey(A, 9, 70)




65
40
38
70
33
18
20
12
32
⇒
65
70
38
40
33
18
20
12
32
✓
70
65
38
40
33
18
20
12
32
⇒

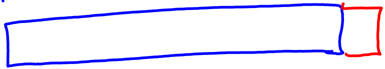
Insert(PQ, x, p): Add x to the priority queue PQ with the priority p.

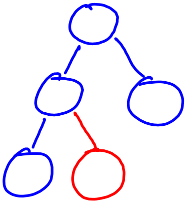
MaxHeapInsert(A, x):

1. Insert x at the (only) spot that keeps the tree a *complete binary tree*.
2. *Fix* the tree to maintain the *max-heap property*:
   - Bubble-up x to a proper position, by swapping with parent.

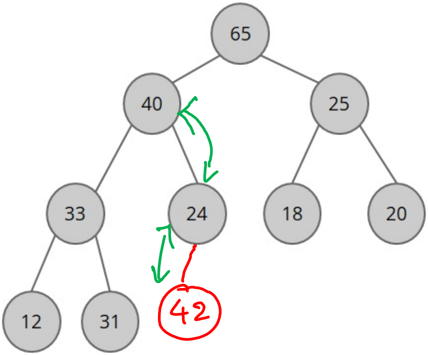
Worst-case Running Time: $\Theta(\lg n)$

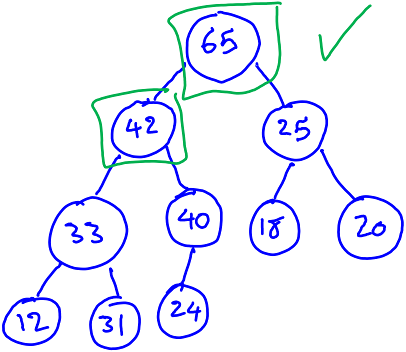
A



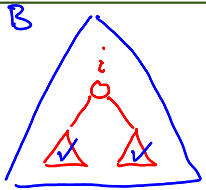


MaxHeapInsert(A, 42)




65
42
25
33
40
18
20
12
31
24

MaxHeapify(B,i):



- **Pre-conditions:** i is a node in a *complete binary tree* B.
  The binary trees rooted at *Left*(i) and *Right*(i) are *max-heaps*.
- **Post-condition:** The binary trees rooted at i is a *max-heap*.

Implementation Method: Bubble-down i to a proper position, by *swapping* with children:

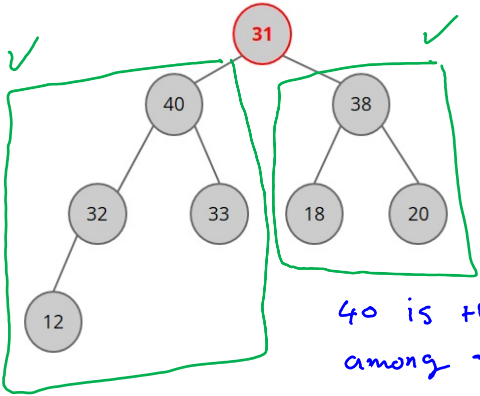
1. Compare the root and its children.
   If the root is the largest, then the three is already a Max-Heap.
   Otherwise, swap the root with largest child.

2. *Fix* the *subtree* rooted at swapped child to maintain the max-heap property by repeating *Step 1* for the sub-tree which its root has been swapped.

Worst-case Running Time: $\Theta(\lg n)$

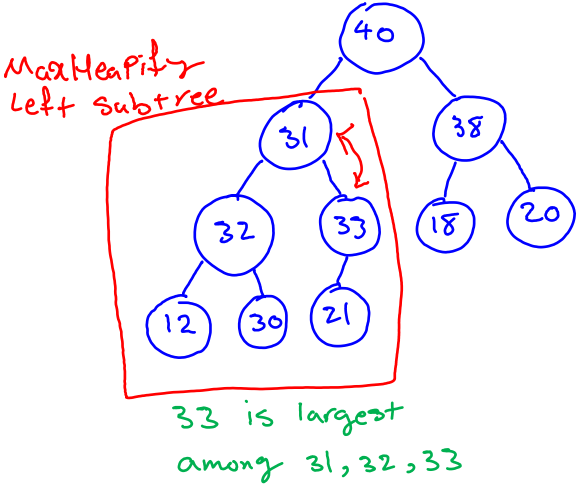
move the root all the way down to a leaf

MaxHeapify$(A, 1)$

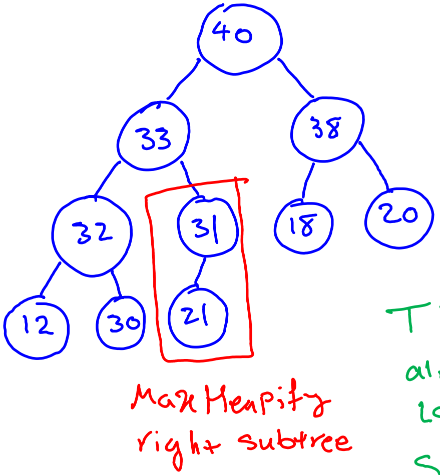


40 is the largest among 31, 40, 38

MaxHeapify: Another way of maintaining the max-heap property



MaxHeapify: Another way of maintaining the max-heap property



The root is already the largest.

So terminate

Heaps: Implementing *HeapExtractMax*

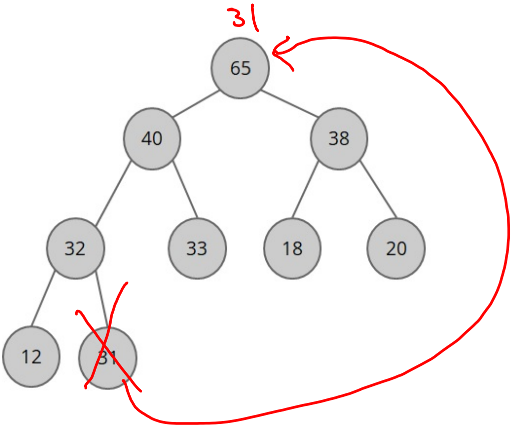
ExtractMax(PQ): Remove and return the item from PQ with the highest priority.

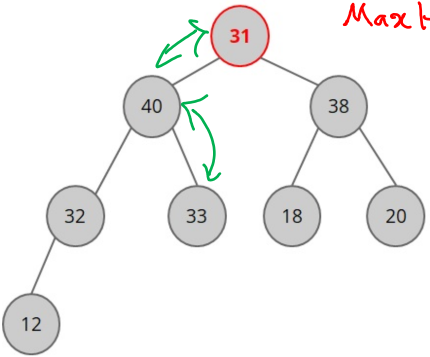
HeapExtractMax(H):

1. Return the root of the tree.
2. *Replace* the root with a node f in the heap so that the tree *remains a complete binary tree*.
3. *Fix* the tree to *maintain the max-heap property*:
   Bubble-down f to a proper position, by swapping with children.

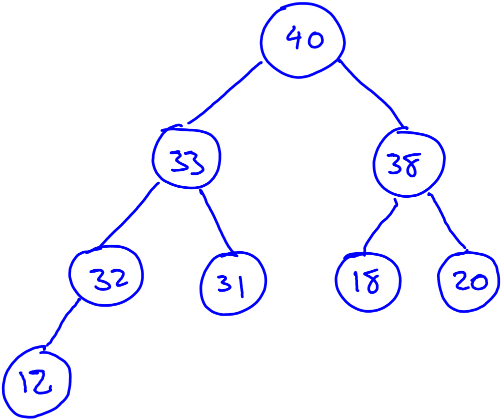
Worst-case Running Time: $\Theta(\lg n)$

HeapExtractMax(A)





MaxHeapify(A,1)


40
33
38
32
31
18
20
12

Heaps:

- *Insert*(PQ, x, p): $\Theta(\lg n)$
- *FindMax*(PQ): $\Theta(1)$
- *ExtractMax*(PQ): $\Theta(\lg n)$
- *IncreaseKey*(PQ,x,k): $\Theta(\lg n)$

- **Intuition:** Tree is partially sorted. Enough to make query operations fast while not requiring full sorting after each update.
- **Complete tree:** Ensures height is small.
- **Heap order:** Supports faster heap operations.

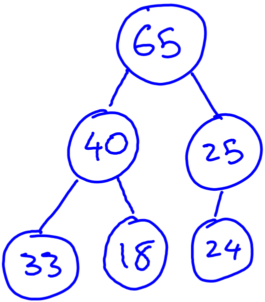
Heap Sort: General Idea

Given a max-heap H, how can we create a sorted array out of H?

- Keep extracting max element for n times. (HeapExtractMax)
- The extracted keys are sorted in non-ascending order.

Worst-case Running Time:

Heap Sort: General Idea



A

65	40	25	33	18	24

Heap Sort: Implementation Details

HeapSort(A):

- **Pre-conditions:** A is an arbitrary array of size n (starting index is 1).
- **Post-condition:** A is sorted in non-decreasing order.

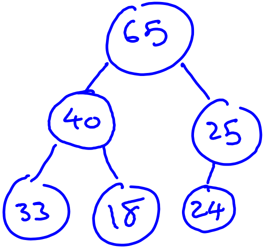
1. Convert A to a max-heap.
2. Let *count* point to the end of A.
3. Extract the max element:

   Count+1

   - Call *HeapExtractMax*(A[1:~~count~~]).
   - Put the max element where *count* points to.
   - Decrease *count* by one.
4. Repeat Step 3 until *count* is 0.

65	40	25	33	18	24

Count



40	33	25	24	18	**65**

Count

33	24	25	**18**	**40**	**65**

25	24	**18**	**33**	**40**	**65**

24	**18**	**25**	**33**	**40**	**65**

18	**24**	**25**	**33**	**40**	**65**

18	**24**	**25**	**33**	**40**	**65**

BuildMaxHeap: First Attempt

BuildMaxHeap(A): Given an unsorted array A, return a max-heap that includes all elements in A.

Idea $\#1$: Call *MaxHeapInsert* for every element in A.

Worst-case Running Time:

BuildMaxHeap: Second Attempt

BuildMaxHeap(A): Given an unsorted array A, return a max-heap that includes all elements in A.

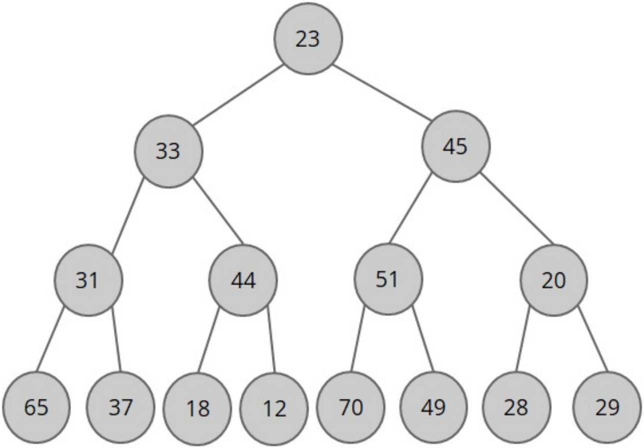
General Idea: Build a max heap *bottom-up*.

Idea $\#2$**:**

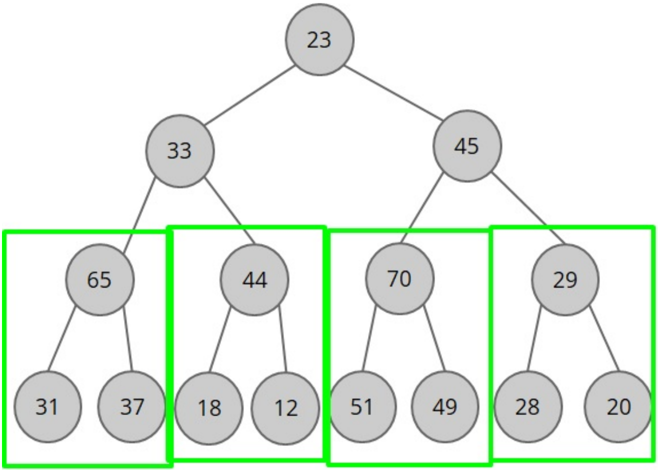
1. Interpret the list as the level order of a complete binary tree.

2. Starting from *bottom* of the tree, call *MaxHeapify* for all *non-leaf nodes*.

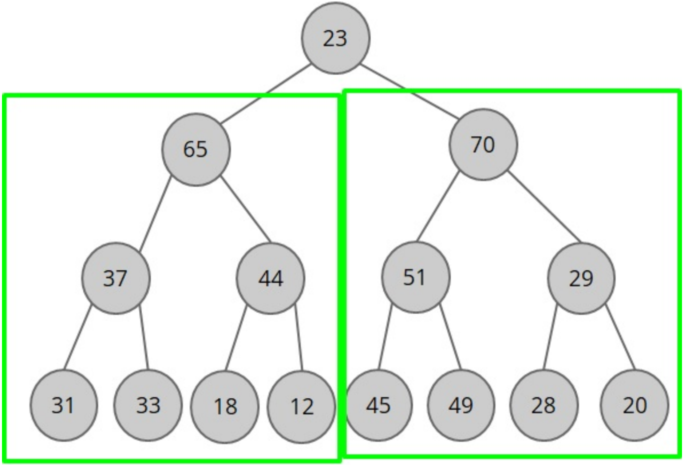
$$A = \{23, 33, 45, 31, 44, 51, 20, 65, 37, 18, 12, 70, 49, 28, 29\}$$



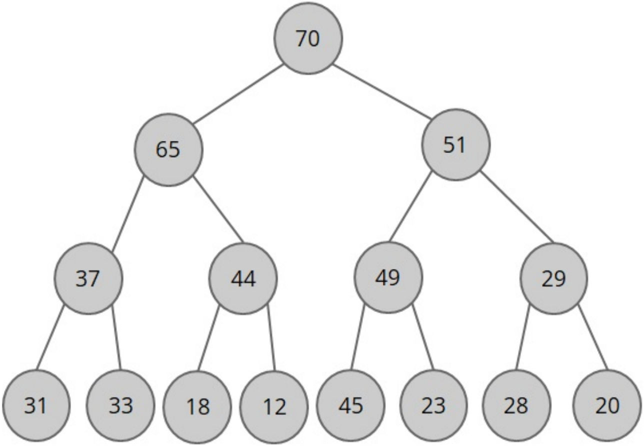
BuildMaxHeap: Second Attempt



BuildMaxHeap: Second Attempt



BuildMaxHeap: Second Attempt



BuildMaxHeap: Worst-case Running Time

BuildMaxHeap: Worst-case Running Time

Algorithm visualizer

https://visualgo.net/en/heap

- **Detailed implementation of heap algorithms:**
  - If you understand the algorithms we discussed conceptually, you should be able to implement them easily. To verify your implementations, please read the text-book to see the detailed algorithms written down.
- **Formal Correctness proof of heap algorithms:**
  - All discussed algorithms can be implements either as an iterative or recursive algorithm.
    Try to prove the correctness of one of them (suggestion: *Max-Heapify*) using the techniques you learned in CSC236.
- **After-lecture readings:**
  - Chapter 2 of the Course Notes, Chapter 6 of CLRS
- **Self-Test Exercises:**
  - Problems at the end of Chapter 2 of the Course Notes, Problems 6.1-1, 6.1-4, 6.2-6 from CLRS.
- **Heap Visualizer:** https://visualgo.net/en/heap