Heaps

Motivation for Heaps

- Binary trees
  - BST – imposing search property on binary trees
  - AVL trees – imposing search and height balance properties on binary trees
  - better suited for searching any key (k-th order statistic and rank)

- K-ary trees
  - B(k)-trees – imposing search and height balance properties on k-ary trees

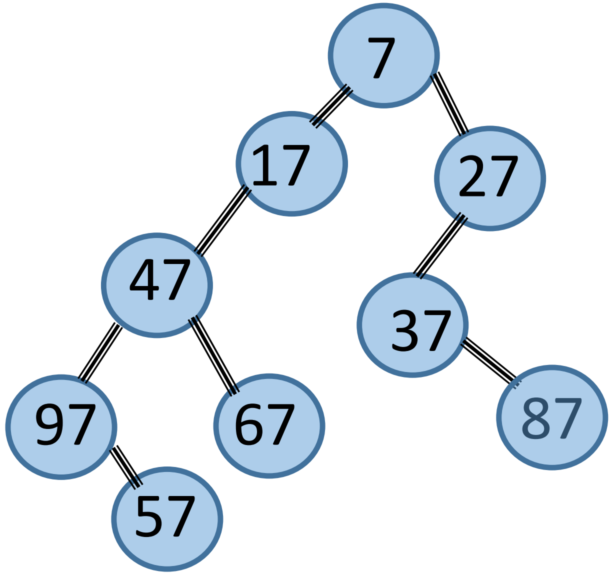
- Binary heaps
  - imposing heap property and structure properties (a severe kind of height balance – leading to array representation) on binary trees
  - better suited for searching the keys on the boundary (1st order or n-th order statistic, minimum or maximum [thereby for priority queues]

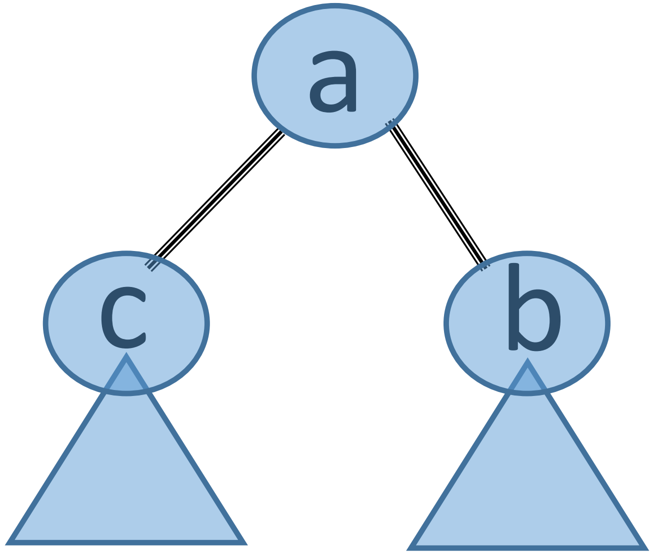
(Binary) (min-)Heap Property

- The key at a node is smaller than the keys at its subtrees (i.e., at its descendants)

≡

- A tree of height 0 is a heap; a tree of height h > 0 is a heap if
  - the key at the root is smaller than the keys at its children
  - its subtrees are heaps

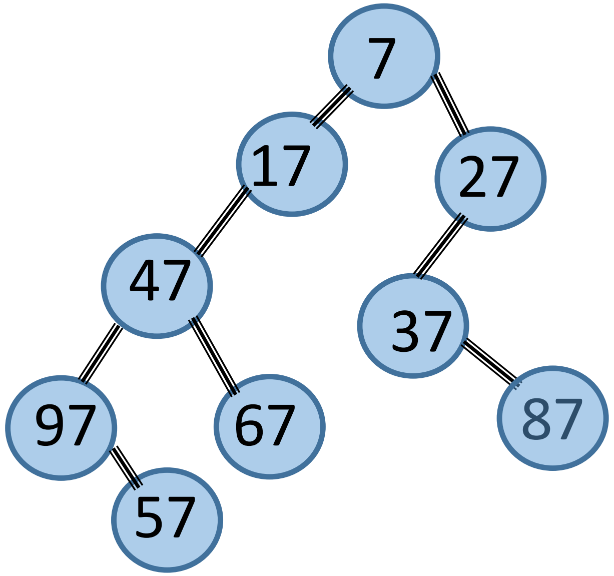




a < c & a < b & b ? c

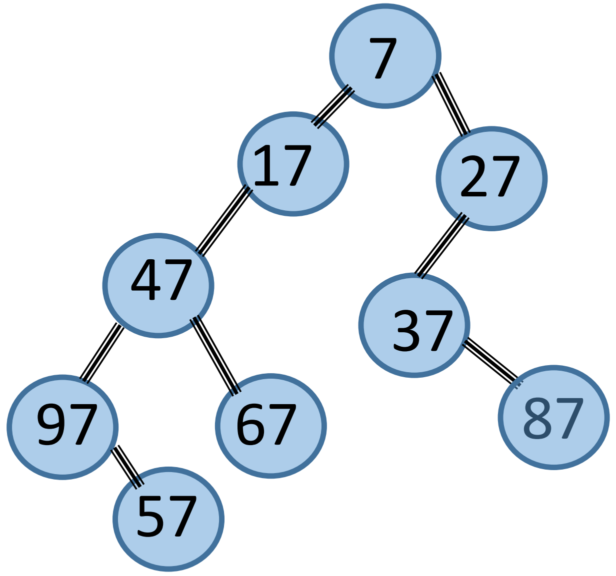
Heap Property - Features

- If the LST and RST at a node is exchanged, the tree continues to retain the heap property.
- The 1^{st} order statistic; i.e., the minimum key is at the root.
- The 2^{nd} order statistic will be a child of the root.



Heap Property – Search

- Searching for minimum and second minimum is trivial. $O(1)$
- Searching for k requires traversing the tree. $O(n)$

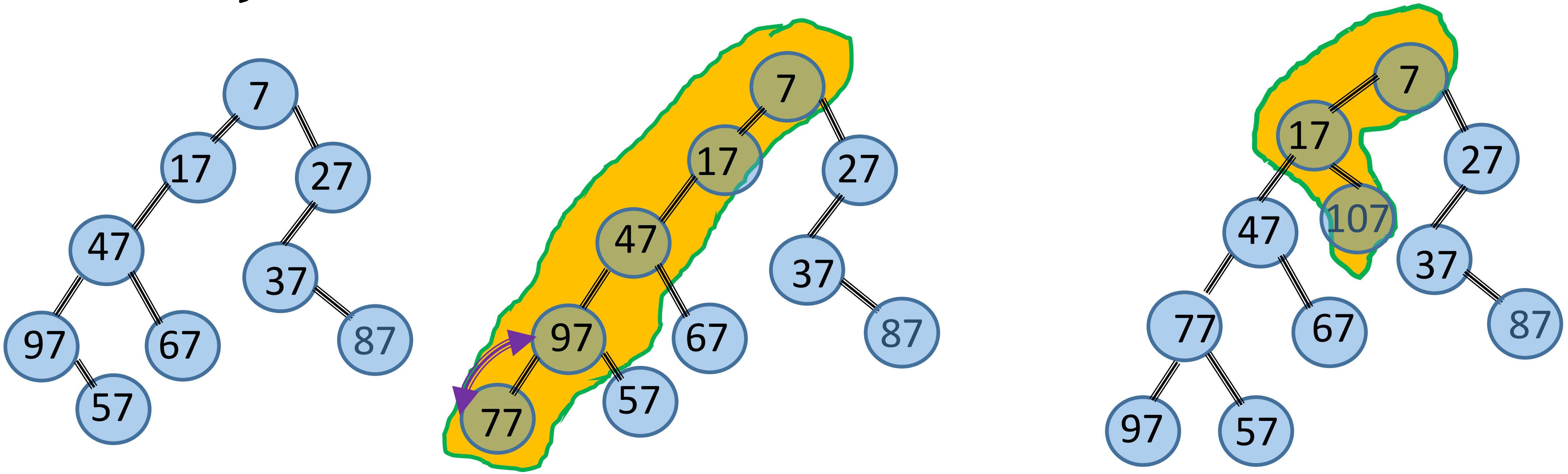


Heap Property - Insert

- Create a new leaf node with key k.
- Adjust the nodes on the leaf-root path by pushing the inserted key towards the root (to the extent required).

$O(h) = O(n)$

Insert 77, 107

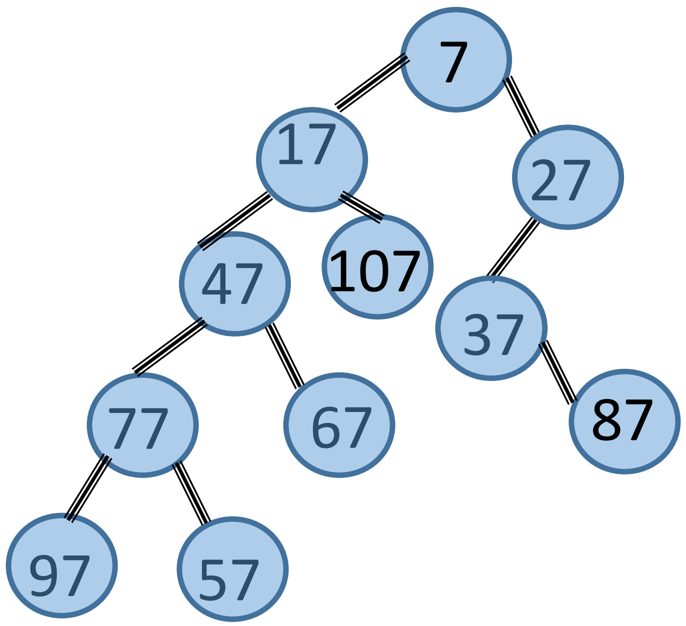


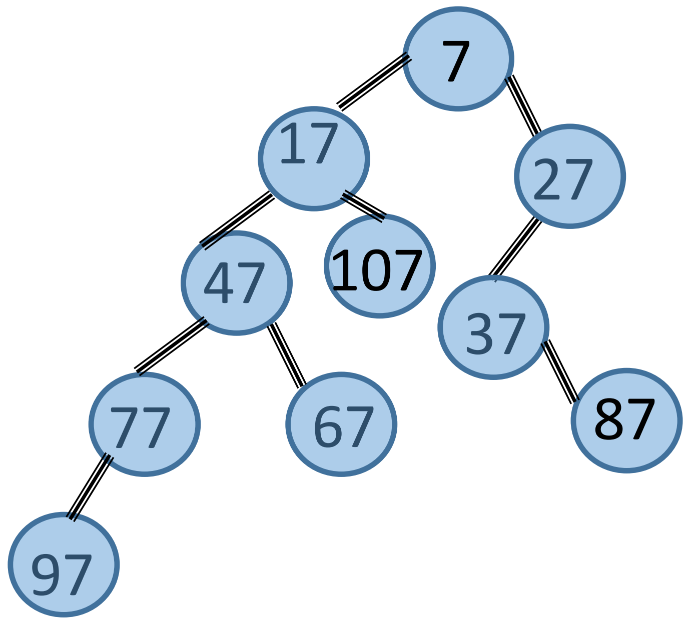
Heap Property – Delete (at a leaf)

- Locate the node
- Remove the node

$O(n)$

Delete 57



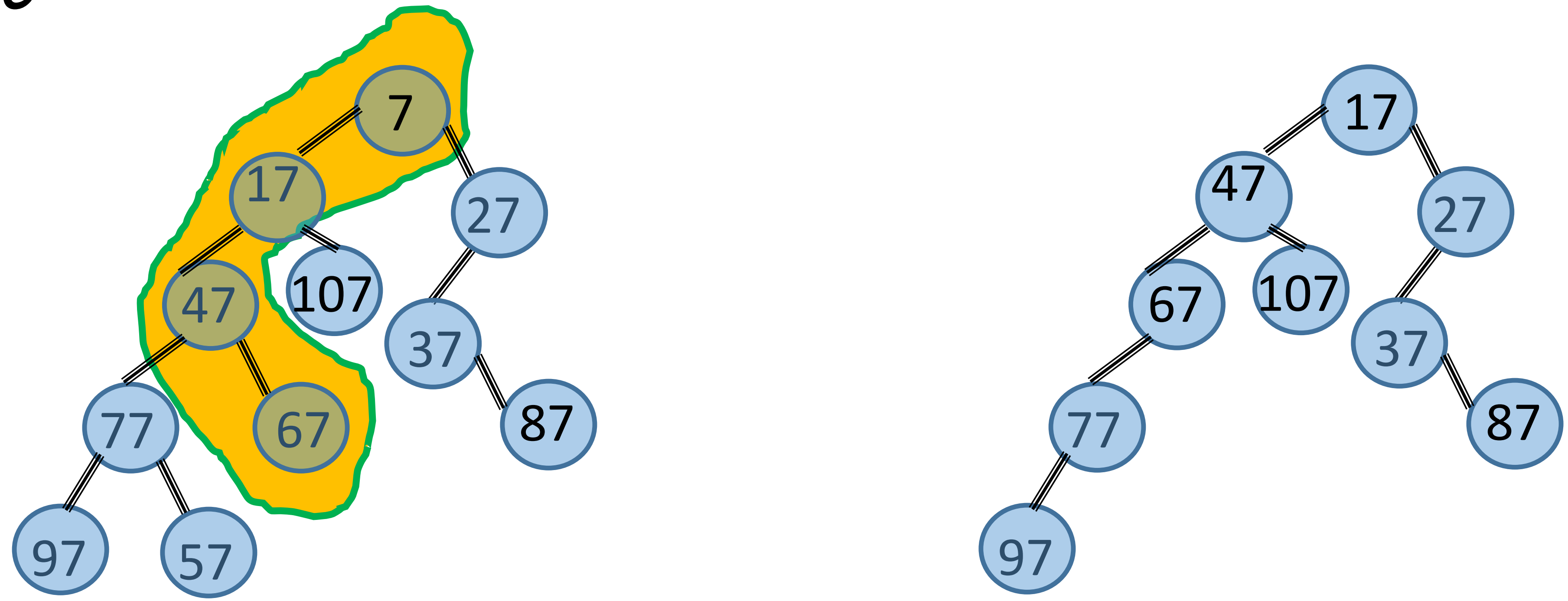


Heap Property – Delete (at root)

- The smallest child becomes the replacement key
- (Recursively) delete the smallest child

$O(n)$

Delete



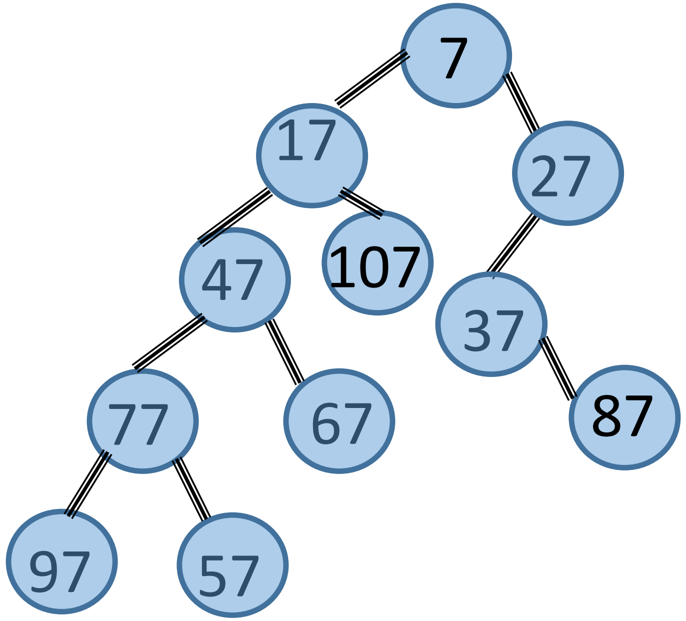
Eventually, number of leaves go down by one.

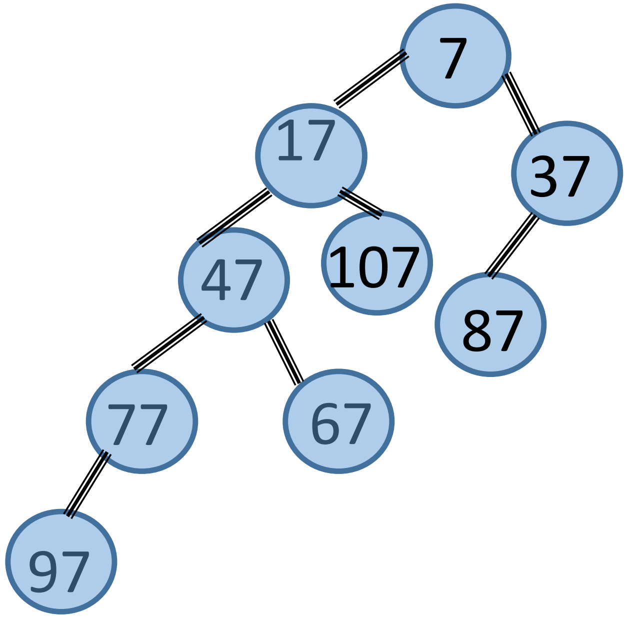
Heap Property – Delete (at an internal node)

- Locate the node
- Delete the node from the root of the subtree

$O(n)$

Delete 27





Eventually, the number of leaves go down by one.

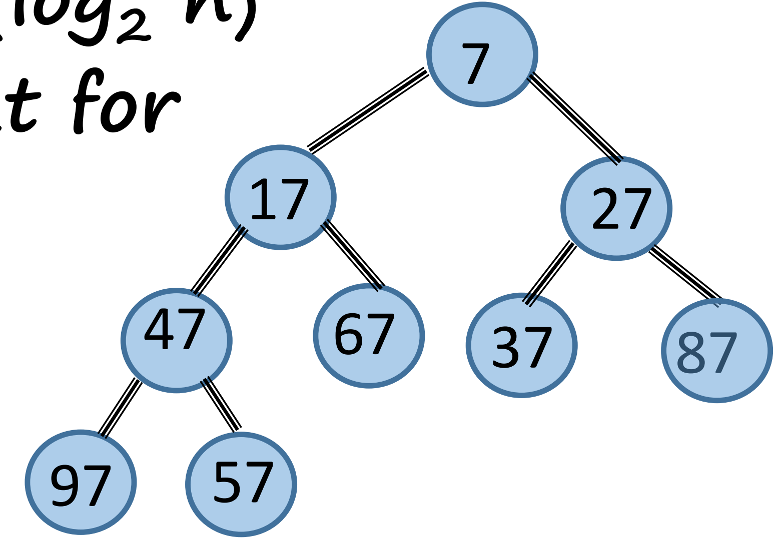
Binary (min-)Heap

Deletion only at the root

- A tree with the following properties
  - Binary (min-) heap property
  - Shape property: Complete binary tree
- In a binary (min) heap on n nodes with height h,

$$2^h-1+1 \le n \le 2^{h+1}-1;\ i.e.,\ h = O(\log_2 n)$$

- Can be implemented using an array such that for a key at position i
  - the left child is at 2i
  - the right child is at 2i+1
  - the parent is at floor(i/2)



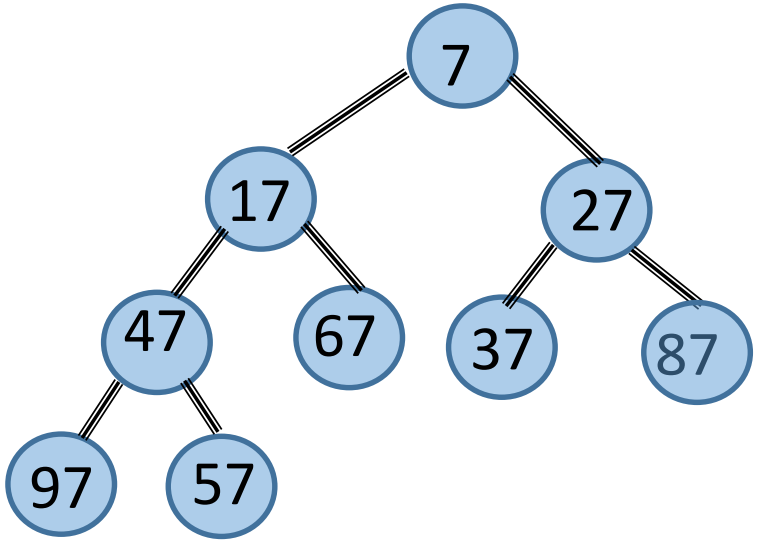
Binary max heap,
3-ary min heap,
3-ary max heap,
m-ary min heap,
m-ary max heap,
can be similarly defined.

1	2	3	4	5	6	7	8	9
7	17	27	47	67	37	87	97	57

Binary (min) Heap - Search

- Traverse the array

O(n)



1	2	3	4	5	6	7	8	9
7	17	27	47	67	37	87	97	57

Binary (min) Heap – Insert

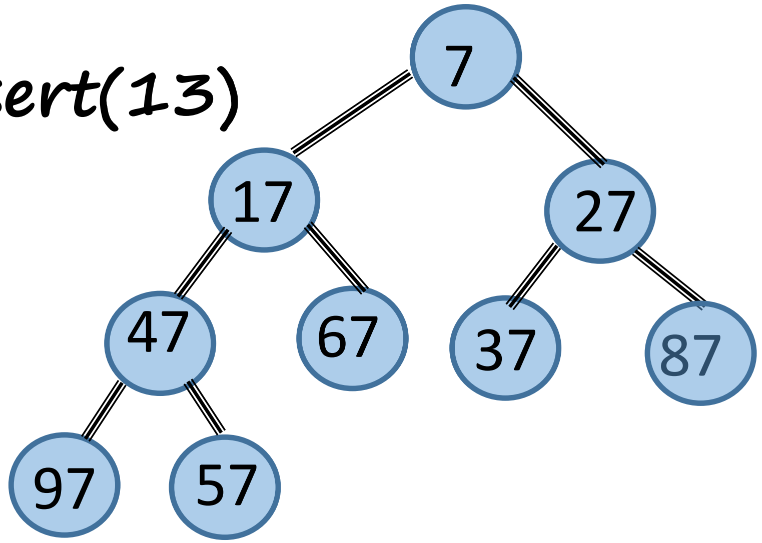
- Insert at the top/root and sift downwards

OR

- Insert at the last (leaf) and sift upwards
  - Create a hole at the end
  - If k can be placed in the hole (w/o violating heap property), then do so, else move the hole towards the root by sifting/sliding the key in the hole's parent downwards until k can be placed in the hole.

$O(h) = O(\log n)$

Insert(13)



1	2	3	4	5	6	7	8	9	10
7	~~17~~ 13	27	47	~~67~~ 17	37	87	97	57	~~?~~ 67

Binary (min) Heap - Insert

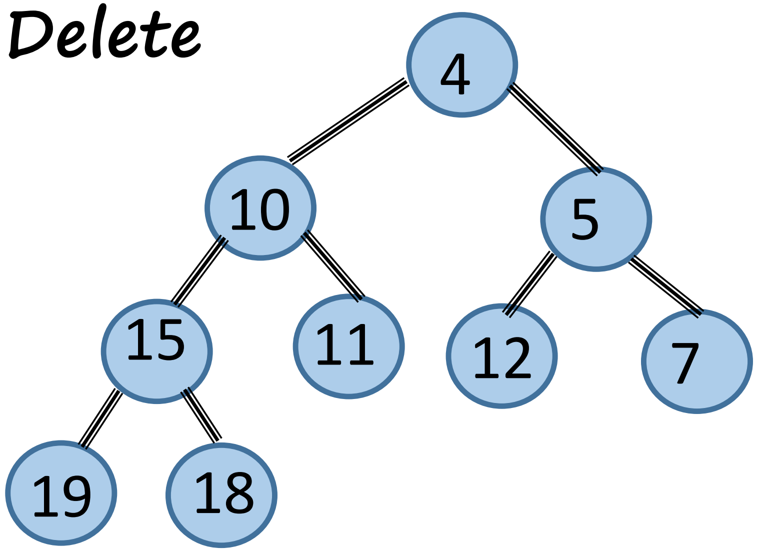
```
INSERT_MINHEAP(A, n, k)


n ← n+1
A[n] ← k
i ← n
While i > 1
      parent ← floor(i/2)
      if A[parent] > A[i]
            SWAP(A[parent], A[i])
            i ← parent
      else
            Return
```

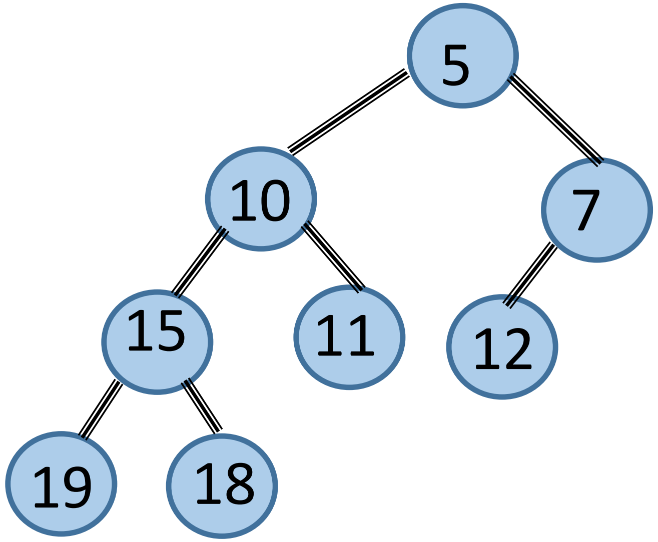
Binary (min) Heap – Delete/Extract

- Since deletion is at the root, can we delete as we did deletion of root in a binary tree with heap property? i.e., can we replace the root with the smallest child and (recursively) delete the smallest child?

Delete



1	2	3	4	5	6	7	8	9
~~4~~ 5	10	~~5~~ 7	15	11	12	~~7~~ ???	19	18



Binary (min) Heap – Delete/Extract

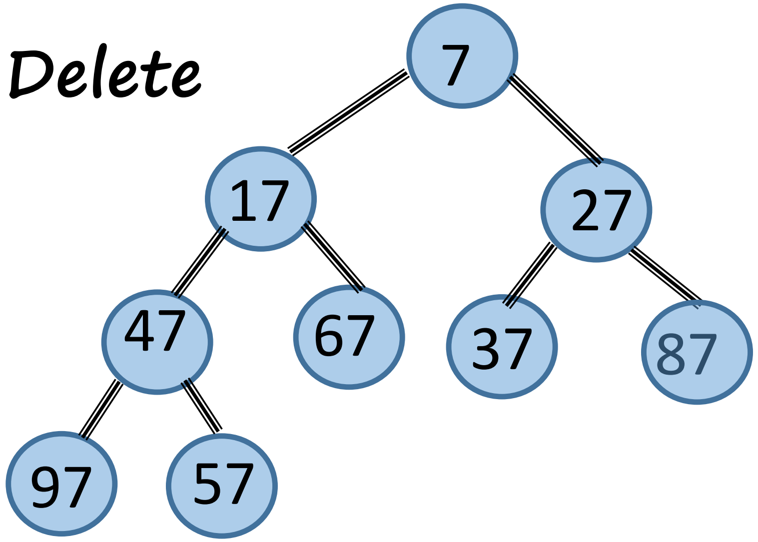
- Replace the root with the last (leaf) key and sift downwards
  - Create a hole at the root
  - If the last key k can be placed in the hole (w/o violating heap property), then do so, else move the hole downwards by sifting/sliding the key in the hole's children upwards until k can be placed in the hole.

OR

- Sift replacement key(s) upwards until the root is replaced

O(h) = O(log n)

Delete



1	2	3	4	5	6	7	8	9
~~7~~ ~~57~~ 17	~~17~~ ~~57~~ 47	27	~~47~~ 57	67	37	87	97	~~57~~ ???

Heap – Exercise

- Write the pseudocode for deletion/extraction in a binary min-heap.
- What is the procedure to delete and arbitrary key from a binary min-heap?
- What is the procedure to increase or decrease a key in a binary min-heap?
- How to use a binary (min-) heap to sort a set?
- Write the pseudocode for search, insert, and extract operations in a binary max-heap.
- Write the pseudocode for search, insert, and extract operations in an m-ary min-heap. [m > 2]
- What is the run-time for search, insert, and extract operations in an m-ary min-heap? [m > 2]

Sorting using a Heap

- To sort a set of n keys
  - Build a binary min-heap — O(log n) per key, so O(n log n)
  - Delete from the binary min-heap until it become empty — as above, O(n log n)

O(n log n)

Sort {15, 20, 7, 9, 30}

Build a binary min-heap

1	2	3	4	5
15				
15	20			
~~15~~ 7	20	~~7~~ 15		
7	~~20~~ 9	15	~~9~~ 20	
7	9	15	20	30

Empty the heap (into a new array)

7, 9, 15, 20, 30

Is it unique?

Build a binary max-heap

1	2	3	4	5
15				
~~15~~ 20	~~20~~ 15			
20	15	7		
20	15	7	9	
~~20~~ 30	~~15~~ ~~30~~ 20	7	9	~~30~~ 15

Empty the heap

30, 20, 15, 9, 7

Heapify

- Building by a sequence of insertions takes O(n log n) time.
- Run-time can be reduced to O(n) – using heapify – converting non-heap CBT into a heap.
- (min)Heapify
  - Represent the set through the array representation of a complete binary tree by placing the keys arbitrarily. [Heap property may not be satisfied now.]
  - Heapify (sift the key downwards) non-leaf nodes (by starting at the last non-leaf node)
    - In the array representation of a complete binary tree, the leaf nodes are from floor(n/2)+1 to n; i.e., nodes 1 through floor(n/2) are non-leaf nodes. So, heapify the nodes at floor(n/2), floor(n/2)-1, …, 1.

To min-Heapify {15, 5, 20, 17, 1, 10}

1	2	3	4	5	6
15	5	20	17	1	10
Heapify at position 3					
15	5	~~20~~ 10	17	1	~~10~~ 20

1	2	3	4	5	6
Heapify at position 2					
15	~~5~~ 1	10	17	~~1~~ 5	20

1	2	3	4	5	6
Heapify at position 1					
~~15~~ 1	~~1~~ ~~15~~ 5	10	17	~~5~~ 15	20

(max)Heapify

```
MAX_HEAPIFY(A, n, i)

largest ← i
lc ← 2i
rc ← 2i+1
While (lc ≤ n) and (A[lc] > A[largest])
      largest ← lc
While (rc ≤ n) and (A[rc] > A[largest])
      largest ← rc
If largest ≠ i
      SWAP(A[largest], A[i])
      MAX_HEAPIFY(A, n, largest)
```

Run-time to build a heap:

1. Run-time for heapifying one subtree (= for a given i) is the number of swaps to sift the root to its appropriate place and is hence O(h)
2. Since the height of the heap is floor(log n), the number of nodes at height h $\leq 2^{floor(\log n)}/2^h \leq n/2^h$
3. Cost of heapifying all the subtrees is the sum of cost of heapifying all the subtrees at height 1 to the root

$$\sum_{h=1}^{floor(\log n)} \frac{n}{2^h} O(h) = O\left(n \sum_{h=1}^{floor(\log n)} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

Sorting using a Heap (w/o deleting into new array)

```
HEAP_SORT(A, n)

for i=floor(n/2) downto 1
        MAX_HEAPIFY(A, n, i)
for i=n downto 1
        SWAP(A[1], A[i])
        MAX_HEAPIFY(A, n, 1)
```

Building a binary (max)-heap

Moving the current max to its appropriate position (in the sorted order) and re-building the heap