Lecture Notes on
Data Structures

M1522.000900

Seoul National University

Fall 2022

Part V

Heap

Heap

What is the best strategy to find the maximum (or minimum) key among the following?

- unsorted list (either array or linked list)
- sorted list (either array or linked list)
- binary search tree

Heap: definition and implementation

A heap is a *complete binary tree* with the **heap property**:

- either a key value $\leq$ its child key values (Min-Heap)
- or a key value $\geq$ its child key values (Max-Heap)

Since a heap is a complete binary tree, an array is the best choice for the implementation.

Remark 1

A heap does not provide a total ordering for all elements but it does for the elements on the same root-to-leaf path.

Max Heap: Insert

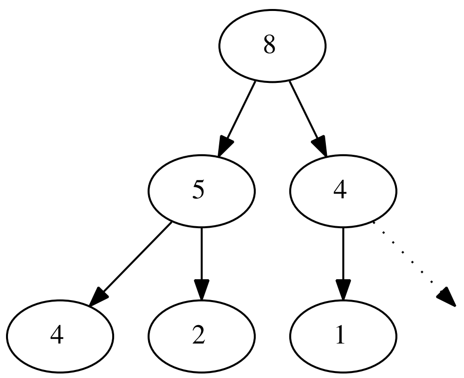
Algorithm 1 (Heap Insert)

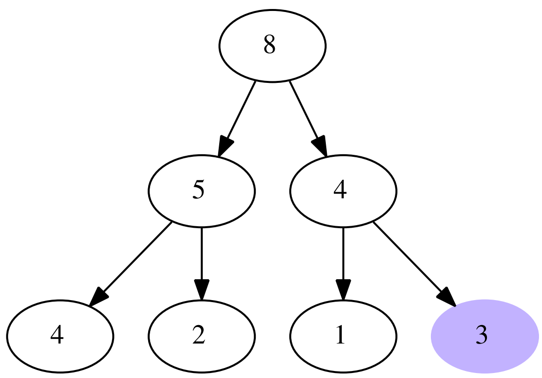
```
Insert(Heap,x)
   Heap[n++] = x;              // n = the # of elts in Heap
   for(k=n-1; k > 0 ; k=parent) {                // BOTTOM-UP
      parent = ⌊(k − 1)/2⌋;
      if (Heap[parent] >= Heap[k]) return;
      else swap(Heap[parent],Heap[k]);
   }
```

It may terminate even before reaching the root, because all the keys on the path to the root are in sorted order.

Example 1

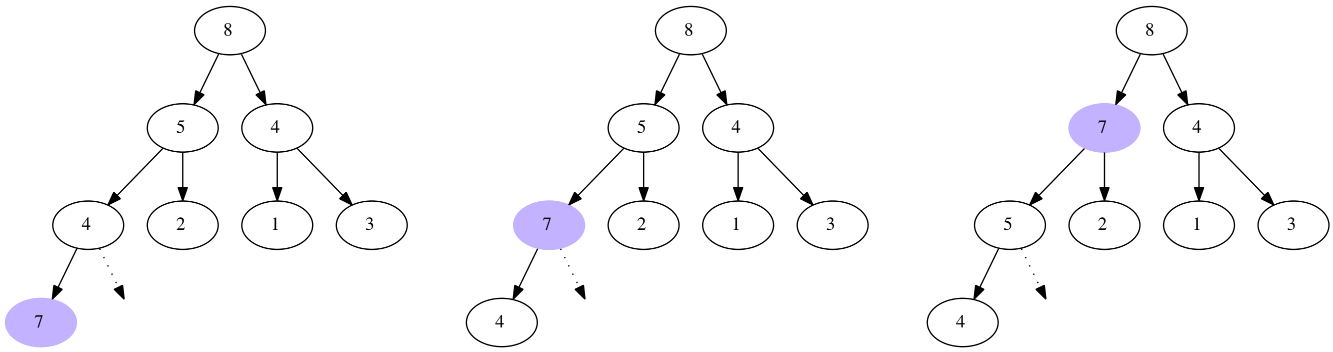
Insert the following keys into a max-heap in the figure: 3, 7, 9.



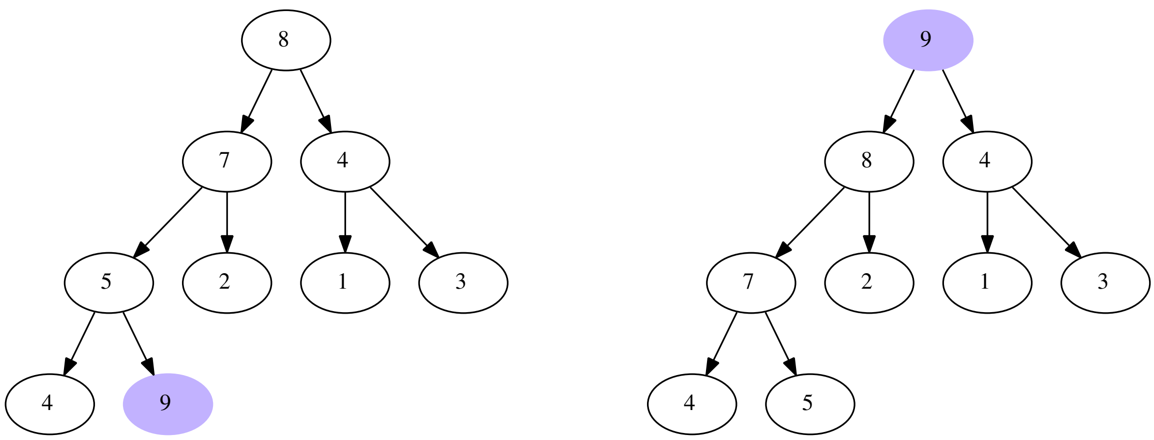


After inserting 3

After inserting 7:



After inserting 9:



Cost of Insertions

1. A single insertion: $\mathcal{O}(\log n)$ because $h = \Theta(\log n)$.
2. Cost of building a heap of n nodes:
   - If it is done by n insertions, then $\mathcal{O}(n \log n)$.
   - We can do better than that, can't we?

Max Heap: Delete Max

1. The max key is at the root. Remove it from the root.
2. Replace the root with the last element (in the array).
3. Both subtrees are still max-heaps.
4. *Sift* (or *percolate*) down along a path until the root key finds its place.
5. The heap property will be restored.

Algorithm 2 (Delete Max)

```
DeleteMax(Heap)
   Heap[0] = Heap[--n];   // Move the last one to the root
   SiftDown(Heap, 0);     // Start from the new root
```

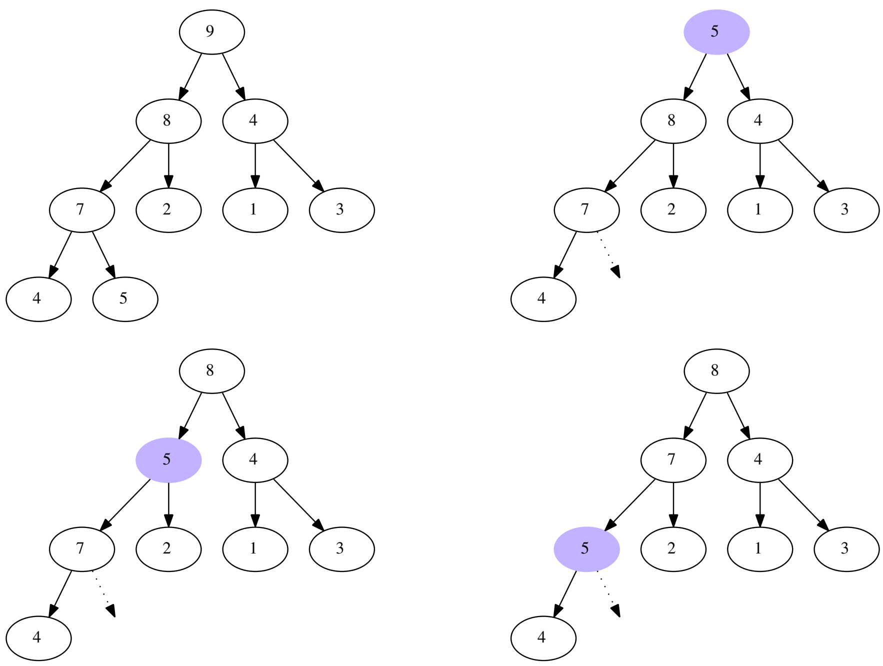
Algorithm 3 (Sift-down)

```
SiftDown(Heap, k)
   while(Heap[k] is not a leaf) {
      j = a child of Heap[k] with the larger key;
      if (Heap[k] >= Heap[j]) return;           // DONE
      swap(Heap[k],Heap[j]);
      k = j;                                    // TOP DOWN
   }
```

`SiftDown` may terminate before reaching a leaf node.

Example 2

Delete the maximum key from a max-heap below.



Bottom-Up Approach to Building a Max Heap

Algorithm 4 (Build a Max Heap from a complete BT)

```
MaxHeapBottomUp(Heap,n)
   for(i=n/2-1; i >= 0; i--) SiftDown(Heap,i);
```

- We can do better than $\mathcal{O}(n \log n)$ if all keys are available.
- Starting from the non-leaf and farthest node from the root (in the array), heapify each subtree rooted by a non-leaf node.
- How do you know you can start from `Heap[n/2-1]`?
  - We can infer it from:

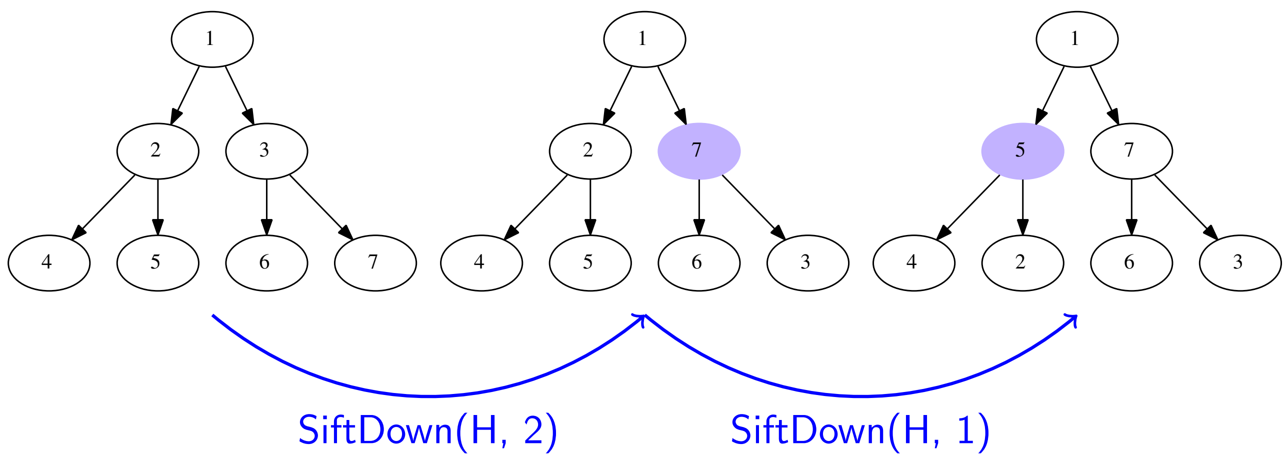
    $2 \times i + 1 = n - 1$ if the left child of H[i] is the last node

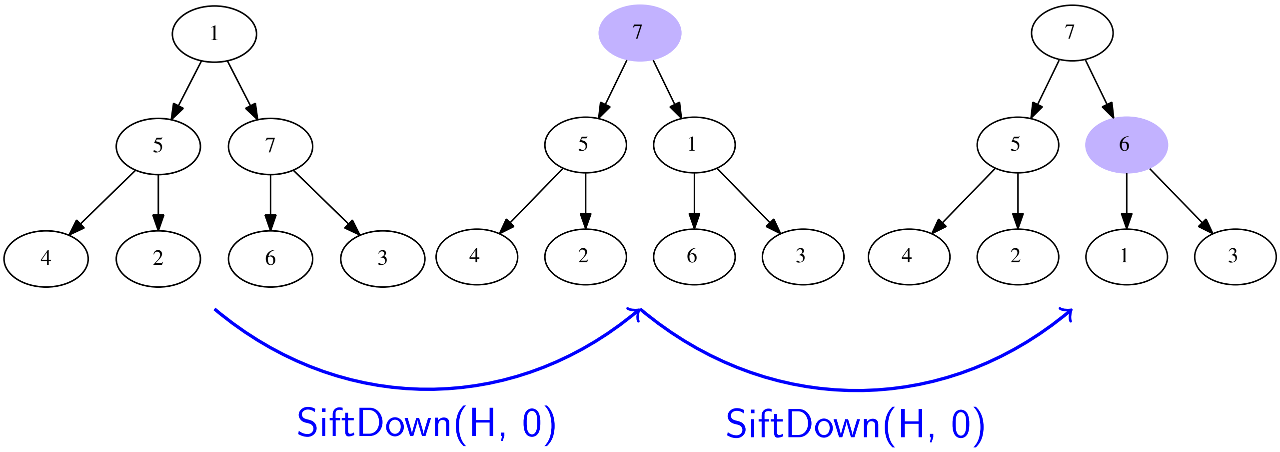
    $2 \times i + 2 = n - 1$ if the right child of H[i] is the last node

  - More formally? Theorem 3 shows that there are $\lceil n/2 \rceil$ leaf nodes and $\lfloor n/2 \rfloor$ internal nodes in any heap of n nodes.

Example 3

Convert the following binary tree into a max-heap.





How many times is `swap` executed?

Analysis of MaxHeapBottomUp

- Count the swap operations.
- The maximum number of swaps by `SiftDown(H,i)` is determined by the level of node `H[i]`.

$$\#\text{swaps(SiftDown(H,i))} \le \textit{height} - \textit{level}(H[i]) - 1$$

- Specifically,

$$\begin{array}{ll} \text{at level } 0, & \#\text{swaps} \le h-1 \text{ and } \#\text{nodes} = 2^0. \\ \text{at level } 1, & \#\text{swaps} \le h-2 \text{ and } \#\text{nodes} = 2^1. \\ & \cdots \\ \text{at level } h-2, & \#\text{swaps} \le 1 \text{ and } \#\text{nodes} = 2^{h-2}. \\ \text{at level } h-1, & \#\text{swaps} \le 0 \text{ and } \#\text{nodes} \le 2^{h-1}. \end{array}$$

Assume $n = 2^h - 1$ (h is the height of the tree). The total number of swaps by the MaxHeapBottomUp algorithm is

$$
\begin{aligned}
&\le \sum_{i=0}^{h-1} i \times 2^{h-1-i} \\
&= 2^{h-1} \times \sum_{i=0}^{h-1} i/2^i \\
&= 2^{h-1} \times \left(2 - \frac{h}{2^{h-1}}\right) \\
&= 2^h - h \\
&\in \mathcal{O}(n)
\end{aligned}
$$

Lemma 1

Let n_k be the number of nodes with k children in a binary tree. For any binary tree, $n_0 = n_2 + 1$.

Proof. Let n and b denote the total number of nodes and the number of branches in a binary tree, respectively. Then,

$$
\begin{aligned}
n &= b + 1 \\
b &= n_1 + 2 \times n_2
\end{aligned}
$$

From $n = n_0 + n_1 + n_2$, it follows that

$$
\begin{aligned}
n_0 &= (b+1) - (n_1 + n_2) \\
&= (n_1 + 2 \times n_2 + 1) - (n_1 + n_2) \\
&= 1 + n_2.
\end{aligned}
$$

□

Lemma 2

Let n_k be the number of nodes with k children in a binary tree. For a <u>*complete*</u> *binary tree of n nodes, $n_1 = 0$ if n is odd and $n_1 = 1$ otherwise.*

<u>*Proof.*</u> The number of nodes in a complete binary tree excluding the bottom level (*i.e.*, the $(h-1)^{th}$ level) is always odd.

$$2^0 + 2^1 + 2^2 + \ldots + 2^{h-2} = 2^{h-1} - 1$$

Thus, if n is odd, then the number of nodes at the bottom level is even, and there is no node having only a single child. If n is even, then the number of nodes at the bottom level is odd, and there is exactly one node having only a single child. $\square$

Theorem 3

Let n_k be the number of nodes with k children in a binary tree. For a <u>*complete*</u> *binary tree of n nodes, $n_0 = \lceil n/2 \rceil$ and $n_2 + n_1 = \lfloor n/2 \rfloor$.*

n_2	n_1	n_0

<u>*Proof.*</u> From $n = n_0 + n_1 + n_2$ and Lemma 1,

$$n = n_0 + n_1 + n_2 = n_0 + n_1 + (n_0 - 1).$$

We obtain $n_0 = (n + 1 - n_1)/2$.

From Lemma 2, if n is odd, $n_0 = (n + 1 - 0)/2 = \lfloor (n+1)/2 \rfloor = \lceil n/2 \rceil$.
Otherwise, $n_0 = (n + 1 - 1)/2 = \lfloor n/2 \rfloor = \lceil n/2 \rceil$.

Therefore, $n_0 = \lceil n/2 \rceil$ and $n_2 + n_1 = n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ for all n. $\square$