

VE281

Data Structures and Algorithms

Priority Queues and Heaps

Learning Objectives:

- Know what a priority queue is
- Know what a min heap is
- Know how a min heap performs enqueue and extractMin operations
- Know how to efficiently initialize a min heap

Outline

- Priority Queue
- Min Heap and Its Operations
- Min Heap Initialization and Application

Priority Queues

- Two kinds of priority queues:
 - Min priority queue.
 - Max priority queue.
- We will focus on **min priority queue**.
 - The max priority queue is similar.

What Is Min Priority Queue?

- A collection of items.
- Each item has a key (or “**priority**”).
- Support the following operations:
 - **isEmpty**
 - **size**
 - **enqueue**: put an item into the priority queue.
 - **dequeueMin**: remove element with **min** key.
 - **getMin**: get item with **min** key.

Applications of Priority Queue

- Banking services
 - VIP customer who arrives later gets served first.
- Network bandwidth management
 - The prioritized traffic, such as real-time data, is forwarded with the least delay once it reaches the network router.
- Discrete event simulation
 - One event happening triggers a few others, which are put into a queue.
 - Simulating in the order of the **beginning time** of the events.

Min Priority Queue: Implementation

- A collection of items.
- Each item has a key (or “**priority**”).
- Support the following operations:
 - **isEmpty**
 - **size**
 - **enqueue**: put an item into the priority queue.
 - **dequeueMin**: remove element with **min** key.
 - **getMin**: get item with **min** key.

What's the time complexity for an unsorted array-based implementation?

Priority Queue Implemented with Heap

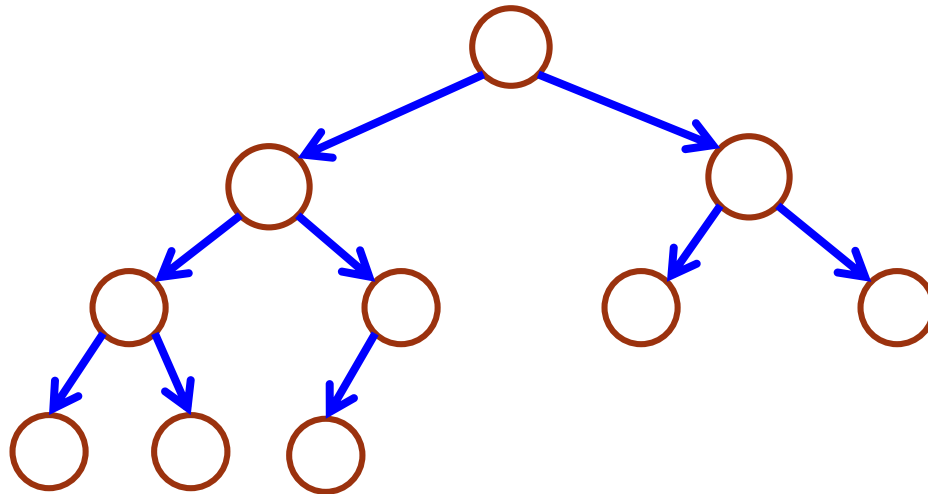
- Priority queues are most commonly implemented using **Binary Heaps** (will be shown soon).
- Complexity of the operation using heap implementation:
 - **isEmpty**, **size**, and **getMin** are $O(1)$ time complexity in the worst case.
 - **enqueue** and **dequeueMin** are $O(\log n)$ time complexity in the worst case, where n is the size of the priority queue.

Outline

- Priority Queue
- **Min Heap and Its Operations**
- Min Heap Initialization and Application

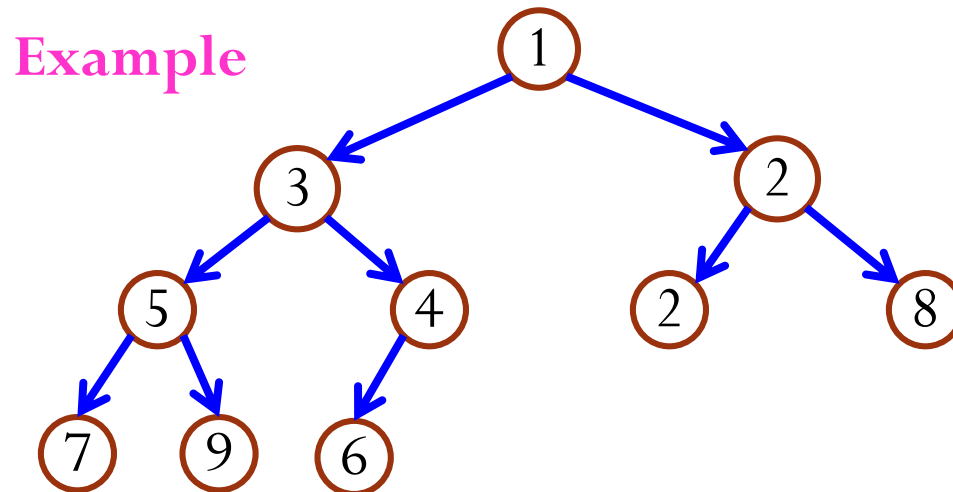
Binary Heap

- A **binary heap** is a **complete binary tree**.

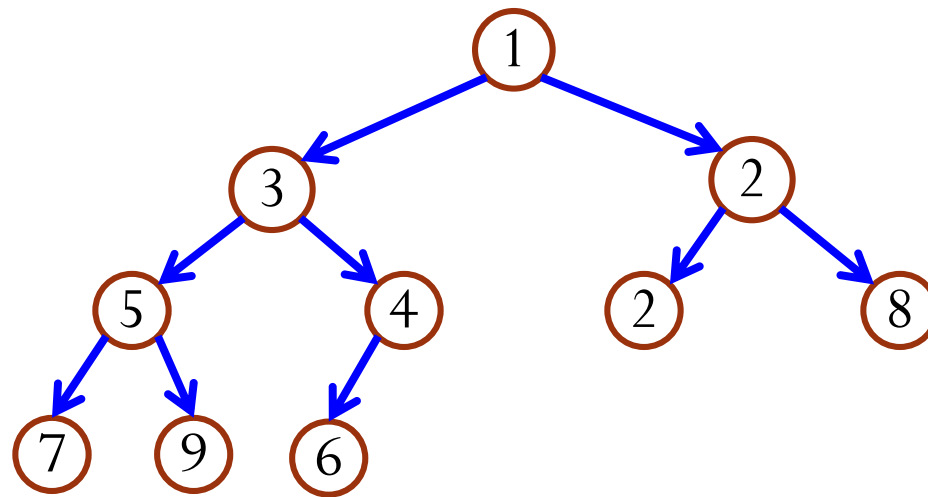


Min Heap

- A min heap is
 - a **binary heap**, and
 - a tree where for **any** node v , the key of v is smaller than or equal to (\leq) the keys of any **descendants** of v .
- Property: The key of the root of **any** subtree is always the smallest among all the keys in that subtree.



Min Heap



- However, the keys of nodes **across** subtrees have no required relationship.
 - Different from binary search trees, which we will show later.

? What's the Height of a Heap of n Nodes?

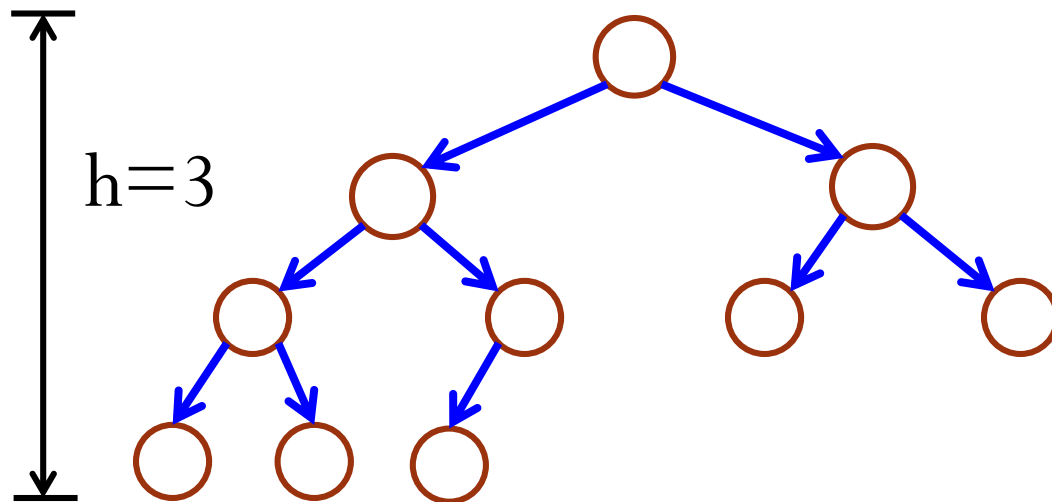
Select **all** the correct answers:

A. $\lfloor \log_2(n + 1) \rfloor - 1$

B. $\lfloor \log_2 n \rfloor - 1$

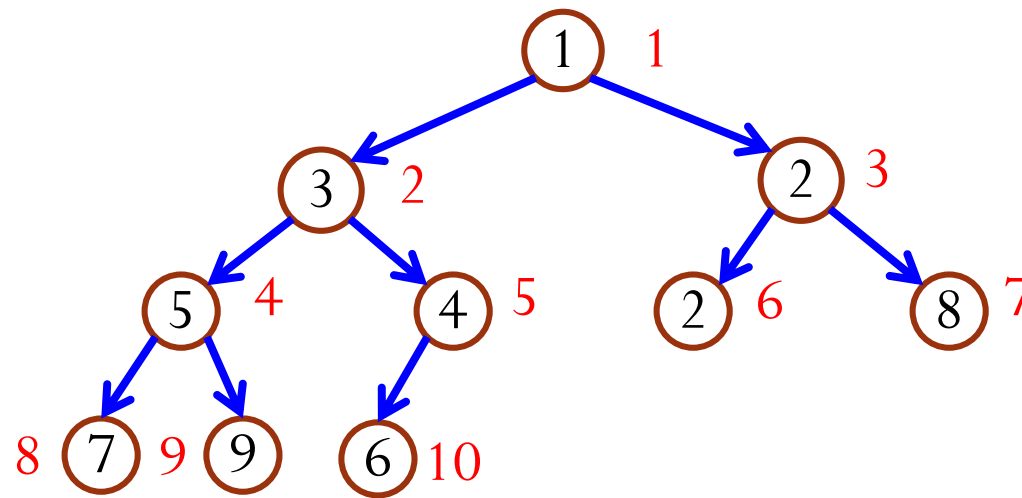
C. $\lfloor \log_2(n + 1) \rfloor - 1$

D. $\lfloor \log_2 n \rfloor$



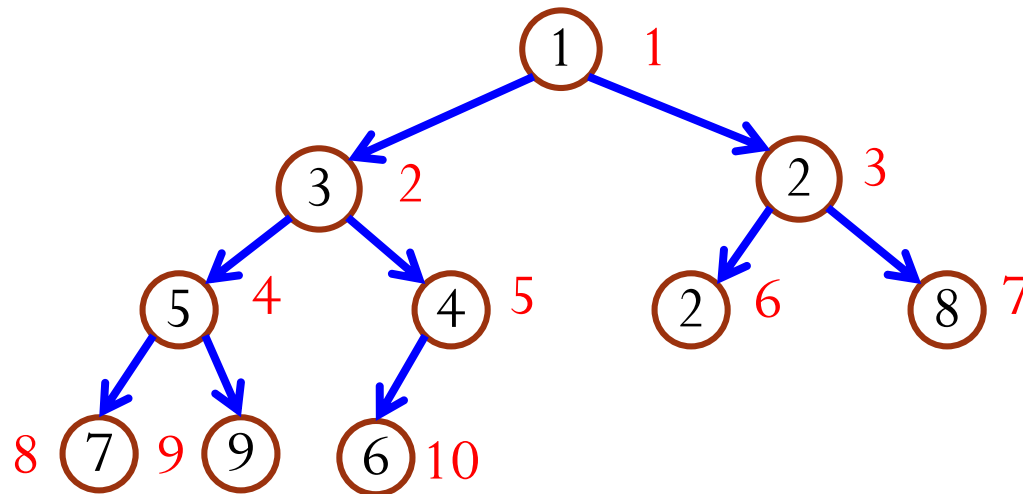
Binary Heap Implementation as an Array

- Store the elements in an array in the order produced by a level-order traversal.
- The first element is stored at index 1.



1	3	2	5	4	2	8	7	9	6
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]

Index Relation



Index relation allows us to move up and down a heap easily.

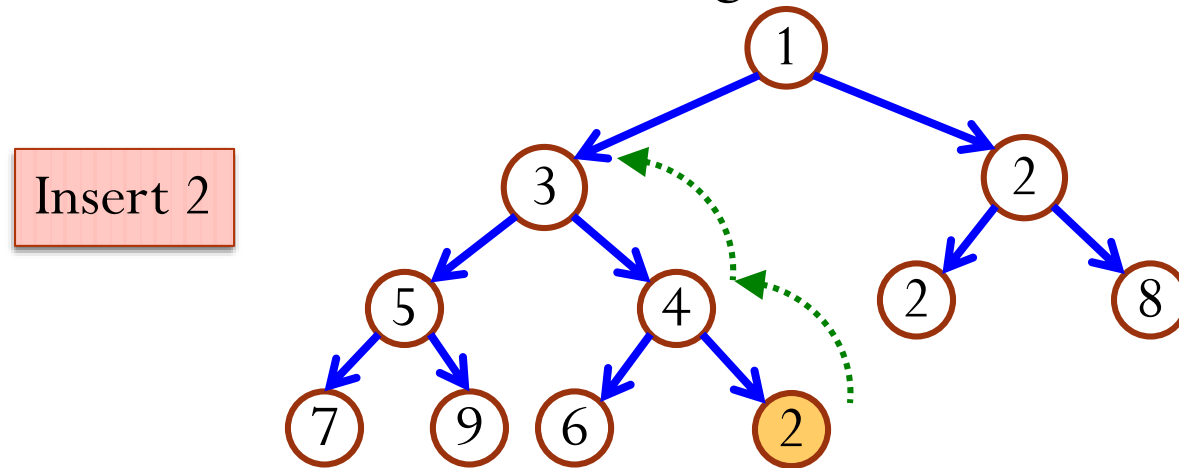
- A node at index i ($i \neq 1$) has its parent at index $\lfloor i/2 \rfloor$.
- Assume the number of nodes is n . A node at index i ($2i \leq n$) has its left child at $2i$.
 - If $2i > n$, it has no left child.
- A node at index i ($2i + 1 \leq n$) has its right child at $2i + 1$.
 - If $2i + 1 > n$, it has no right child.

Min Heap Implementation

- We also have a **size** variable to keep the number of nodes in the heap.
 - The heap elements are stored in `heap[1]`, `heap[2]`, ..., `heap[size]`.
- Operations
 - **isEmpty**: `return size==0;`
 - **size**: `return size;`
 - **getMin**: `return heap[1];`

Procedure of enqueue

- Insert **newItem** as the rightmost leaf of the tree.

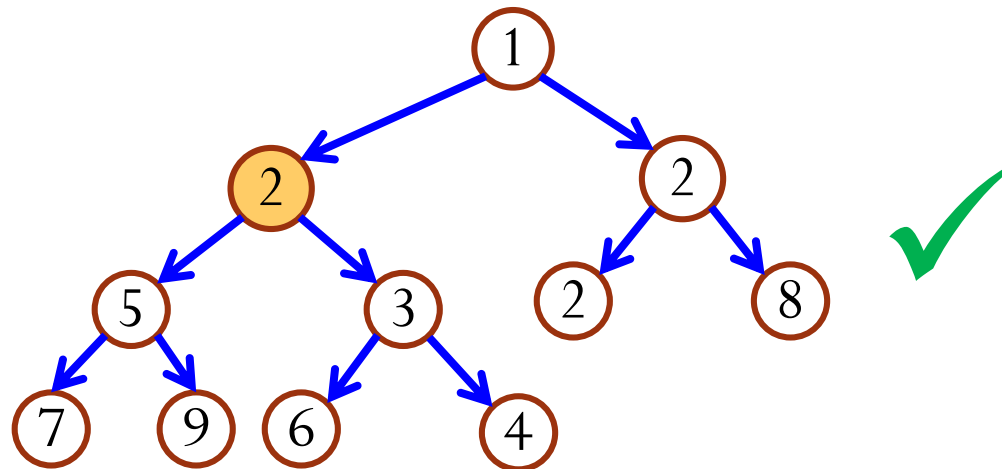
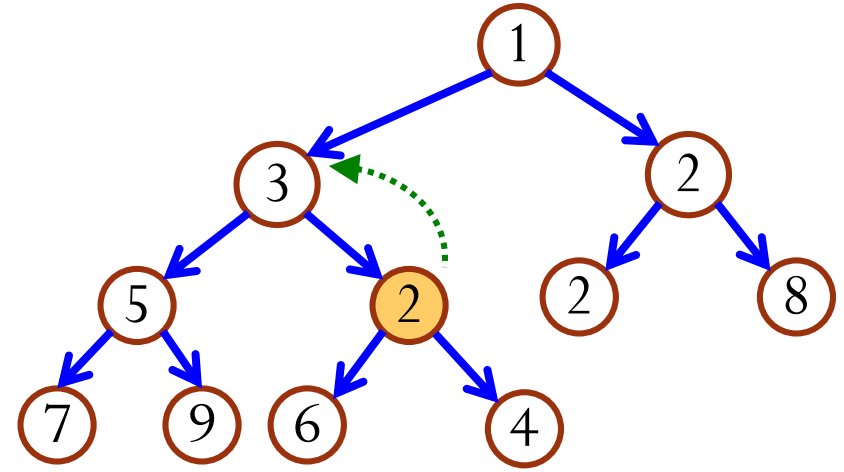
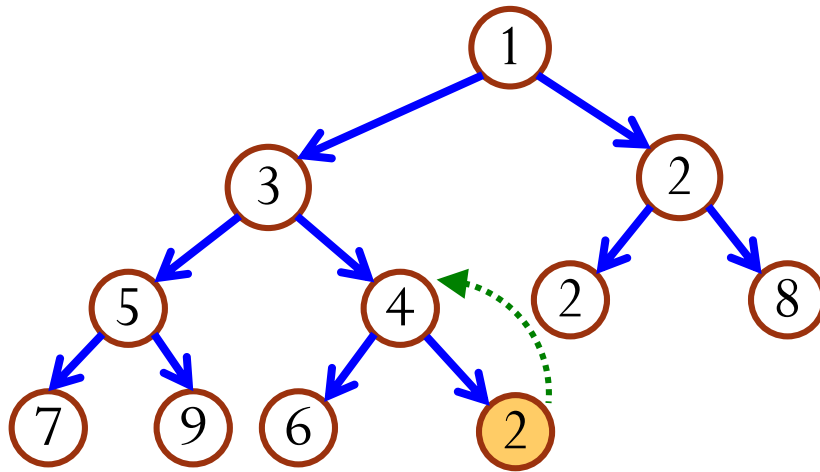


heap[++size] = newItem;

- The tree may no longer be a heap at this point!
- **Percolate up** **newItem** to an appropriate spot in the heap to restore the heap property.

Percolate Up

Illustration



Percolate Up

Code

```
void minHeap::percolateUp(int id) {  
    while(id > 1 && heap[id/2] > heap[id]) {  
        swap(heap[id], heap[id/2]);  
        id = id/2;  
    }  
}
```

- Pass index (**id**) of array element that needs to be percolated up.
- Swap the given node with its parent and move up to parent until:
 - we reach the root at position 1, or
 - the parent has a smaller or equal key.

enqueue

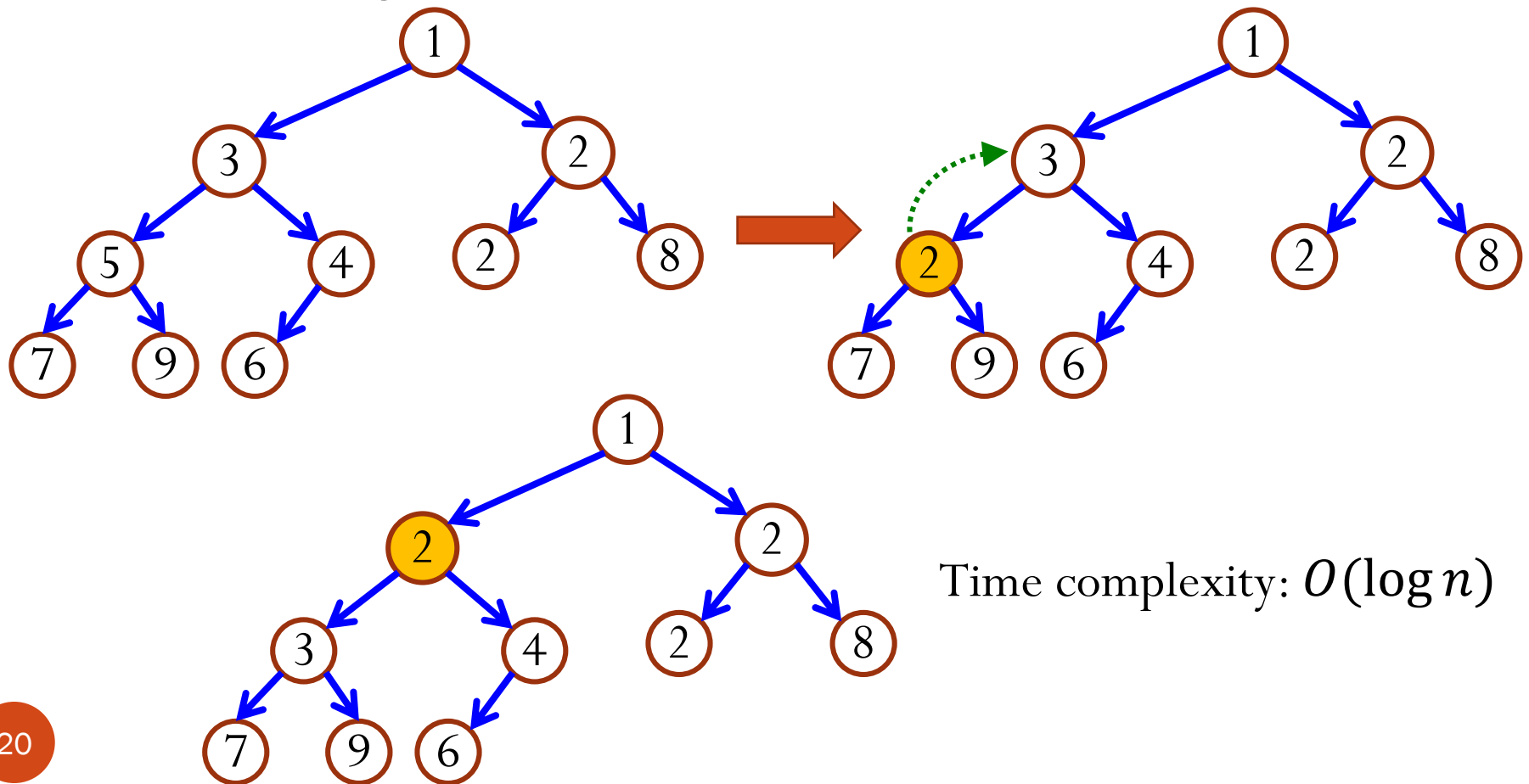
Code

```
void minHeap::enqueue(Item newItem) {  
    heap[++size] = newItem;  
    percolateUp(size) ;  
}
```

- What is the time complexity?
 - $O(\log n)$

Aside: Decrease Key

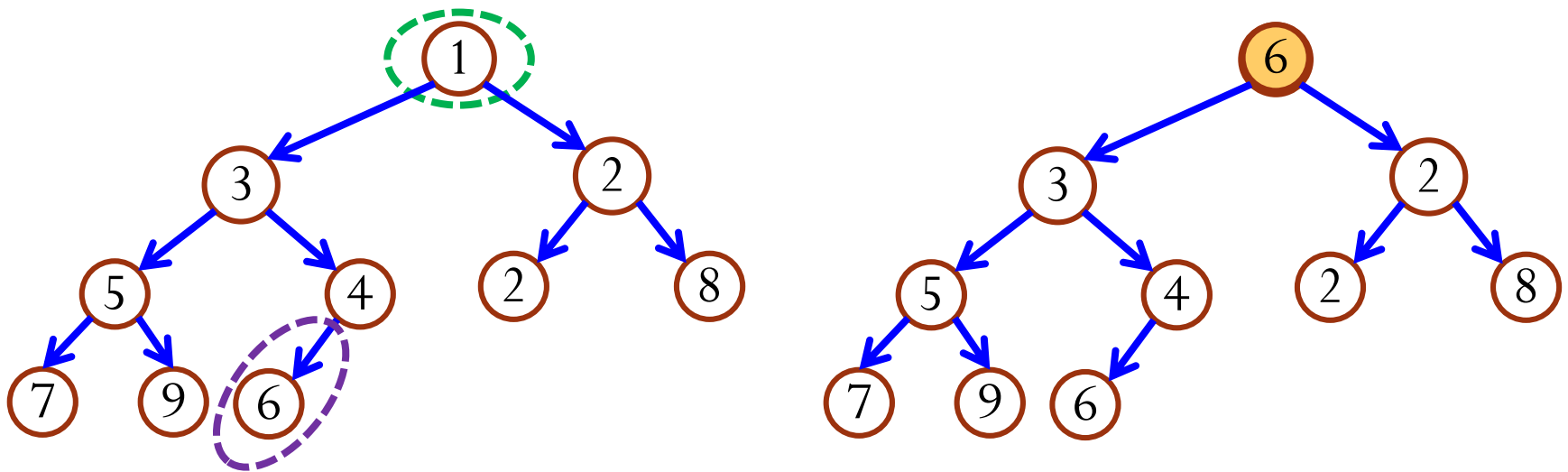
- Percolating-up can also be exploited to implement the decreasing-key operation



Procedure of dequeueMin

- The min item is at the root. Save that item to be returned.
- Move the item in the rightmost leaf of the tree to the root.

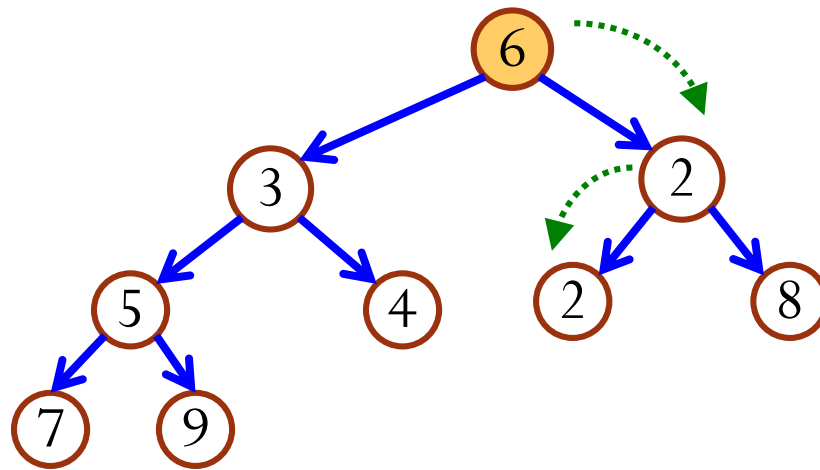
swap(heap[1], heap[size--]);



- The tree may no longer be a heap at this point!

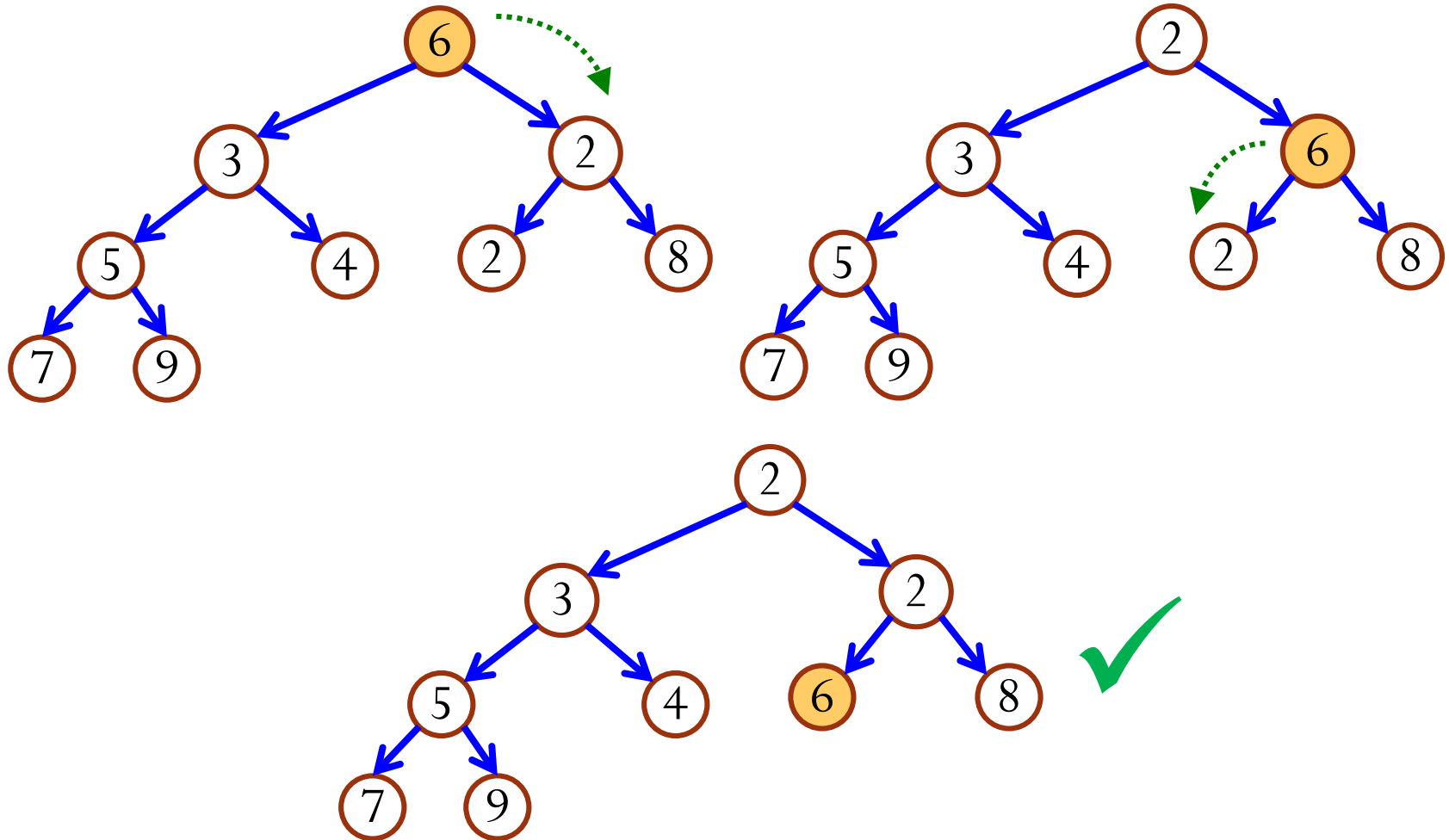
Procedure of dequeueMin

- **Percolate down** the recently moved item at the root to its proper place to restore heap property.
 - For each subtree, if the root has a **larger** search key than **either of its children**, swap the item in the root with that of the **smaller** child.



Percolate Down

Illustration



Percolate Down

Code

```
void minHeap::percolateDown(int id) {  
    for(j = 2*id; j <= size; j = 2*j) {  
        if(j < size && heap[j] > heap[j+1]) j++;  
        if(heap[id] <= heap[j]) break;    find the smaller child  
        swap(heap[id], heap[j]);  
        id = j;  
    }  
}
```

- Pass index (**id**) of array element that needs to be percolated down.
- Swap the key in the given node with the smallest key among the node's children, moving down to that child, until:
 - we reach a leaf node, or
 - both children have larger (or equal) key

dequeueMin

Code

```
Item minHeap::dequeueMin() {  
    swap(heap[1], heap[size--]);  
    percolateDown(1);  
    return heap[size+1];  
}
```

- What is the time complexity?
 - $O(\log n)$

Outline

- Priority Queue
- Min Heap and Its Operations
- Min Heap Initialization and Application

Initializing a Min Heap

- How do we initialize a min heap from a set of items?
- Simple solution: insert each entry one by one.
 - The worst case time complexity for inserting the k -th item is $O(\log k)$, so creating a heap in this way is $O(n \log n)$.
- Instead, we can do better by putting the entries into a **complete** binary tree and running **percolate down** intelligently.

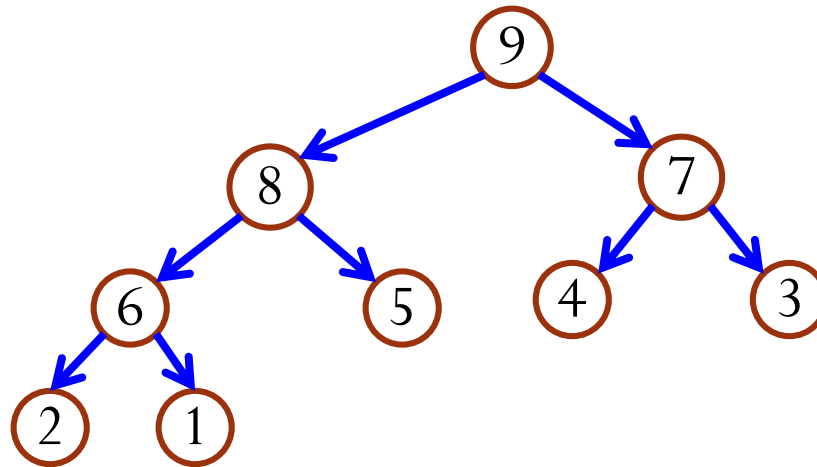
Initializing a Min Heap

- Put all the items into a complete binary tree.
 - Implemented using an array.
- Starting at the rightmost array position **that has a child**, percolate down all nodes in **reverse** level-order.
 - The rightmost array position **that has a child** is **size/2**.
- Procedure:
For **$i = \text{size}/2$** down to **1**
 percolateDown(i) ;

Initializing a Min Heap

Illustration

- Input items: 9, 8, 7, 6, 5, 4, 3, 2, 1
- First step: put all the items into a complete binary tree.

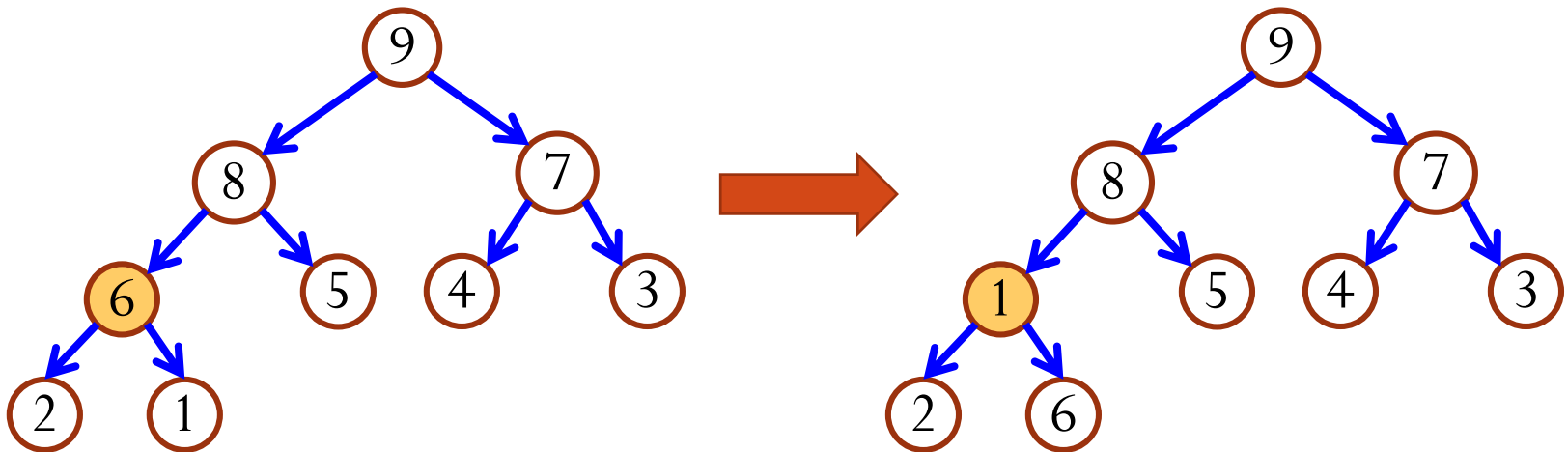


Initializing a Min Heap

Illustration

- Starting at the rightmost array position **that has a child**, percolate down all nodes in **reverse** level-order.

Node at index $9/2 = 4$

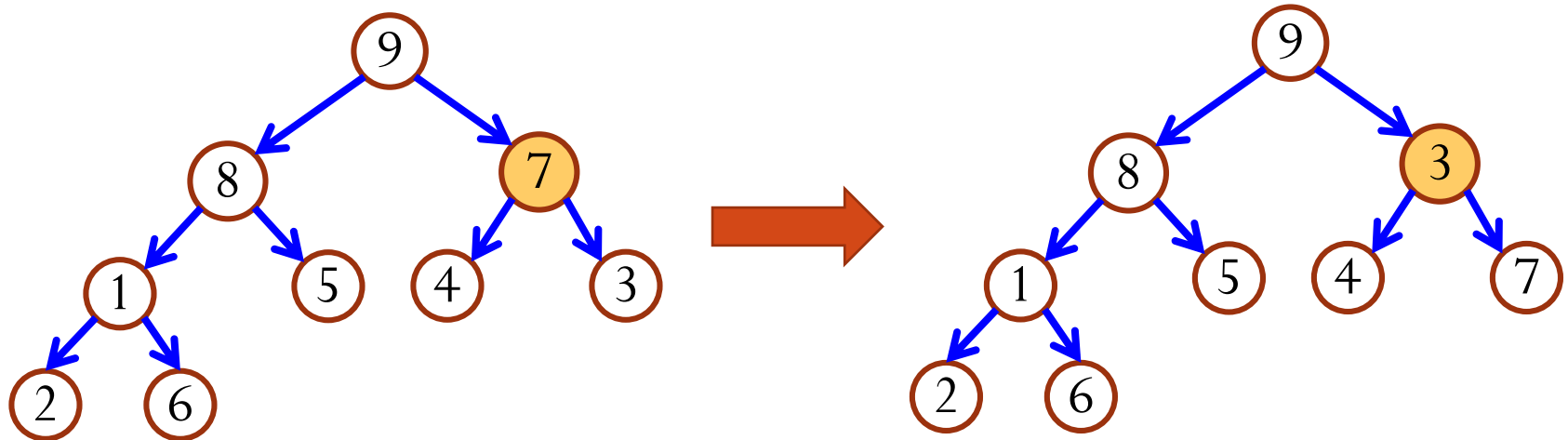


Move to next lower array position.

Initializing a Min Heap

Illustration

Node at index 3

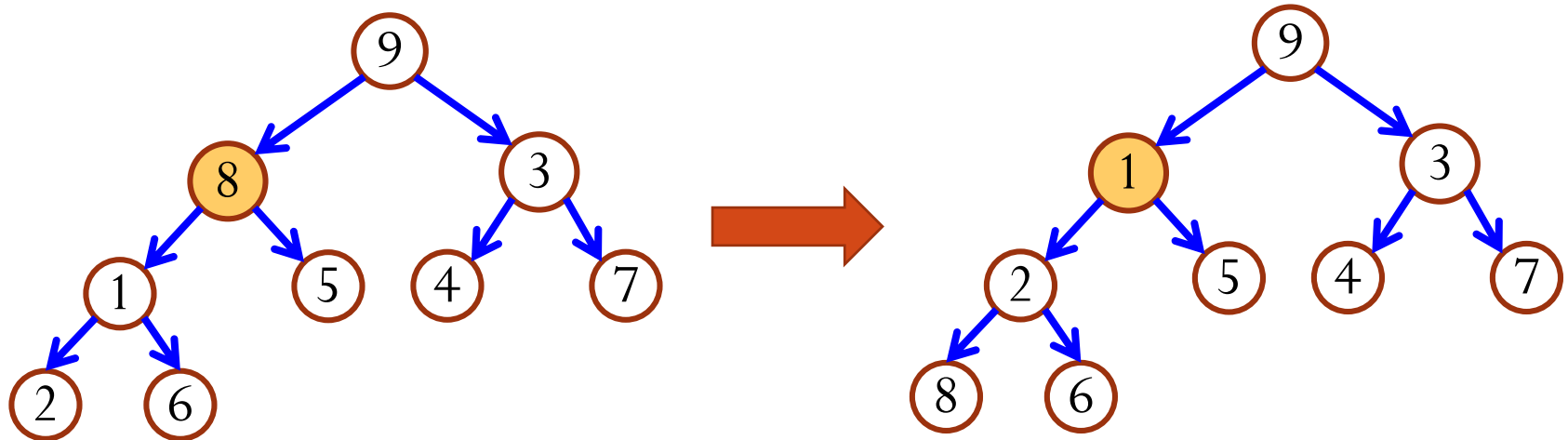


Move to next lower array position.

Initializing a Min Heap

Illustration

Node at index 2

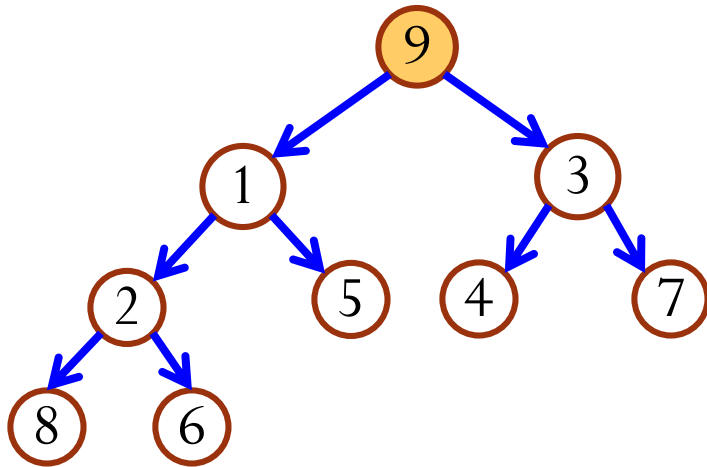


Move to next lower array position.

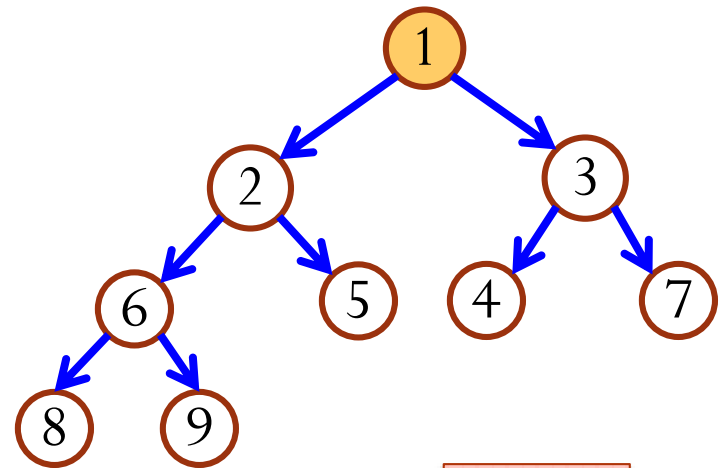
Initializing a Min Heap

Illustration

Node at index 1

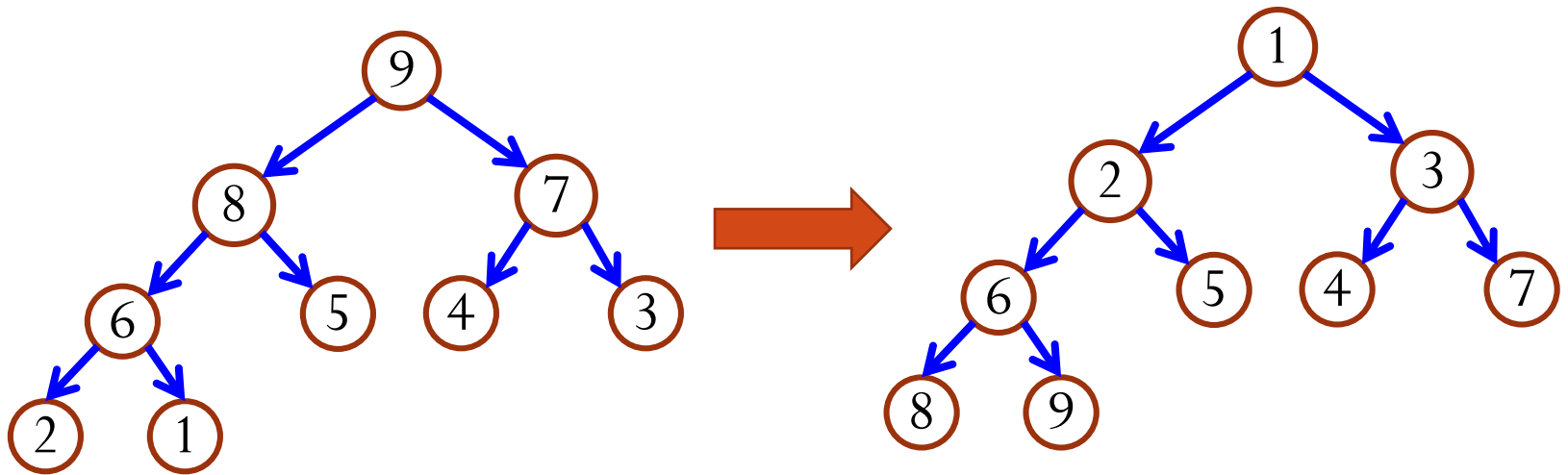


Exercise: What's the result?



Done!

Time Complexity Analysis



- Suppose: the **height** of the heap is h .
- Note: Number of nodes at level k ($0 \leq k \leq h$) is $\leq 2^k$.
- Note: The worst case time complexity of percolating down a node at level k is $O(h - k)$.

Time Complexity Analysis

$$T(h) \leq \sum_{k=0}^{h-1} 2^k O(h-k) = O\left(\sum_{k=0}^{h-1} 2^k (h-k)\right)$$

- What is $S(h) = \sum_{k=0}^{h-1} 2^k (h-k)$?

$$S(h) = 2^0 h + 2^1 (h-1) + 2^2 (h-2) + \dots + 2^{h-1} \cdot 1$$

$$2S(h) = 2^1 h + 2^2 (h-1) + \dots + 2^{h-1} \cdot 2 + 2^h \cdot 1$$

$$2S(h) = \quad 2^1 h \quad + 2^2 (h-1) + \dots + 2^{h-1} \cdot 2 + 2^h \cdot 1$$

$$S(h) = 2S(h) - S(h) = 2^1 + 2^2 + \dots + 2^h - h = 2^{h+1} - 2 - h$$

Time Complexity Analysis

$$T(h) \leq O(2^{h+1} - 2 - h)$$

- For a complete binary tree, we have

$$h = \lceil \log_2(n + 1) \rceil - 1 \leq \log_2(n + 1)$$

where n is the number of nodes.

- Therefore, the algorithm for initializing a min heap with n nodes has worst case time complexity $T(n) = O(n)$.
 - Better than the way to enqueue entry one by one.

Application of Heap: Sorting

- Procedure:
 1. Initialize a min heap with all the elements to be sorted
Complexity: $O(n)$
 2. Repeatedly call **dequeueMin** to extract elements out of the heap.
Complexity: $O(n \log n)$
- The resulting elements are sorted by their keys.
- What is the time complexity? $O(n \log n)$
- This is known as **heap sort**.

Application: Median Maintenance

- Input: a sequence of numbers x_1, x_2, \dots, x_n , one-by-one
- Output: at each time step i , the median of x_1, x_2, \dots, x_i
- Problem: how to do this with $O(\log i)$ time at each step i ?
- Hint: using two heaps, one min heap and one max heap
- Key idea: maintain the smallest half ($\lceil \frac{n}{2} \rceil$) in max heap and the largest half ($\lfloor \frac{n}{2} \rfloor$) in the min heap
- Question: How do you get the median (i.e., the $\lceil \frac{n}{2} \rceil$ -th smallest item)?
 - Answer: get max from the max heap

How to Insert a New Item?

- Key problem: maintain the **invariant** that the smallest half ($\lceil \frac{n}{2} \rceil$) in max heap and the largest half ($\lfloor \frac{n}{2} \rfloor$) in the min heap
 - To maintain balance between the two heaps
- If n (before insertion) is even
 - If new item $\leq \min(\text{minHeap})$, insert it into maxHeap
 - Else (new item $> \min(\text{minHeap})$), first extract min value from minHeap, then insert that value in maxHeap, and finally insert new item into minHeap
- If n (before insertion) is odd
 - If new item $\geq \max(\text{maxHeap})$, insert it into minHeap
 - Else (new item $< \max(\text{maxHeap})$), first extract max value from maxHeap, then insert that value in minHeap, and finally insert new item into maxHeap

Time complexity is $O(\log i)$