CSCI 3230 Data Structures

Priority Queue and Heap

Weitian Tong, Ph.D.

Department of Computer Science
Georgia Southern University
Website: www.weitiantong.com
Email:wtong@georgiasouthern.edu

Table of contents

Priority Queue

Priority Queue

A priority queue stores a collection of entries

Each entry is a pair (key, value)

Allow for efficient **insertion** and **removal** based on keys

```
class MinPriorityQueue {
  a collection of entries
  ------
  size();
  isEmpty();
  insert(K key, V value);
  min();
  removeMin();
}
```

```
class Entry {
  key
  value
  ------
  getKey()
  getValue()
}
```

Total Order Relations, Comparator ADT

Keys can be arbitrary objects on which an order is defined

When the priority queue needs a comparator to compare two keys
Mathematical concept of **total order relation** $\leq$

- **Comparability property**: either $x \leq y$ or $y \leq x$
- **Antisymmetric property**: $x \leq y$ and $y \leq x \Rightarrow x = y$
- **Transitive property**: $x \leq y$ and $y \leq z \Rightarrow x \leq z$

A **comparator** encapsulates the action of comparing two objects according to a given total order relation

Example Comparator

```
1  /** Class representing a point in the plane
2      with integer coordinates */
3  public class  Point2D {
4      protected int xc, yc; // coordinates
5      public  Point2D(int  x,  int  y)  {
6          xc = x;
7          yc = y;
8      }
9      public int getX() {
10         return xc;
11     }
12     public int getY() {
13         return yc;
14     }
15 }
```

Example Comparator

```
1  /** Comparator for 2D points under
2      the standard lexicographic order. */
3  public class Lexicographic implements Comparator{
4      int  xa, ya, xb, yb;
5      public int compare(Object a, Object b) {
6         xa = ((Point2D) a).getX();
7         ya = ((Point2D) a).getY();
8         xb = ((Point2D) b).getX();
9         yb = ((Point2D) b).getY();
10        if  (xa != xb)
11          return  (xb > xa);
12        else
13          return  (yb >= ya);
14     }
15 }
```

Sequence-based Priority Queue

Sequence-based Priority Queue

Implement with an unsorted list

Performance:

- **insert** takes $O(1)$ time
  since we can insert the item at the beginning or end of the sequence
- **removeMin** and **min** take $O(n)$ time
  since we have to traverse the entire sequence to find the smallest key

Implement with a sorted list

Performance:

- **insert** takes $O(n)$ time
  since we have to find the place where to insert the item
- **removeMin** and **min** take $O(1)$ time
  since the smallest key is at the beginning

Application: Priority Queue Sorting

Use a priority queue to sort a list of comparable elements

insert and **removeMin**

```
1  Algorithm PQ-Sort(S, C)
2  Input: list S, comparator C for the elements of S
3  Output: list S sorted in increasing order according to C
4
5  P ← priority queue with comparator C
6  while !S.isEmpty()
7    e ← S.remove(S.first())
8    P.insert (e, ∅)
9  while !P.isEmpty()
10   e ← P.removeMin().getKey()
11   S.addLast(e)
```

The running time of this sorting method depends on the priority queue implementation

Heap

Motivation for Heaps

Goal:

- $O(\log n)$ insertion
- $O(\log n)$ removal

Remember that $O(\log n)$ is almost as good as $O(1)$!
e.g., $n = 1,000,000,000 \rightarrow \log n \approx 30$

There are **min** heaps and **max** heaps.

We will assume **min** heaps.

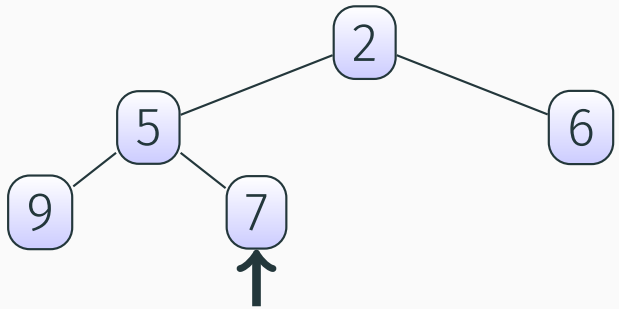
Min Heaps

A **min heap** is a binary tree storing keys at its nodes and satisfying the following properties:

- Heap-order: for every internal node v other than the root

$$key(v) \geq key(parent(v))$$

- (Almost) complete binary tree: let h be the height of the heap
  - for $i = 0, \ldots, h-1$, there are 2^i nodes of depth i
  - at depth $h-1$,
    - the internal nodes are to the left of the external nodes
    - Only the rightmost internal node may have a single child



The last node of a heapis the rightmost node of depth h

Height of a Heap (Optional)

Theorem

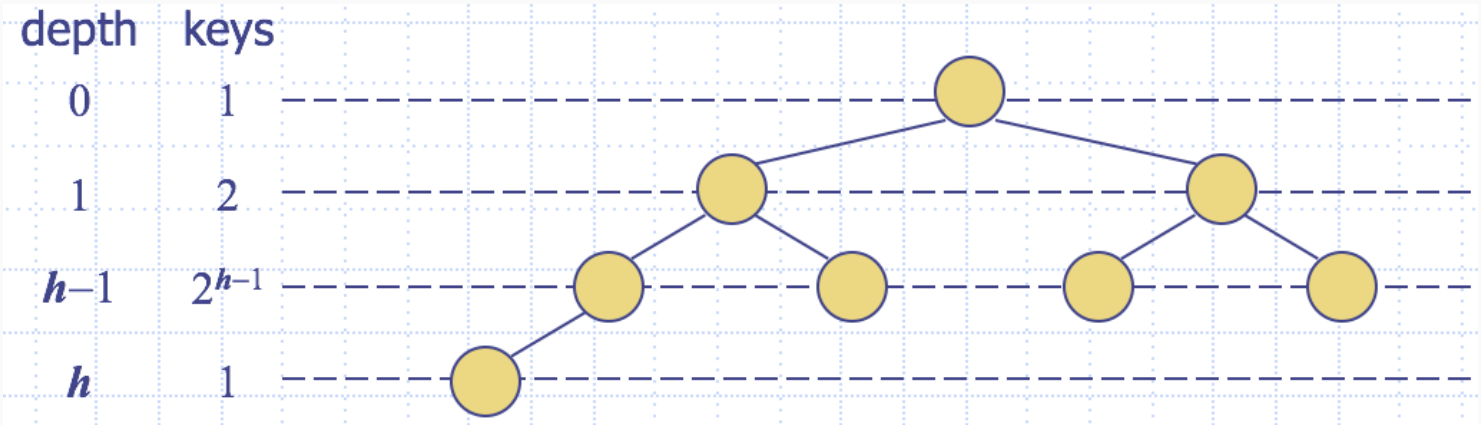
A heap storing n keys has height $O(\log n)$

Let h be the height of a heap storing n keys

Since there are 2^i keys at depth $i = 0, \ldots, h-1$ and at least one key at depth h, we have

$$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} + 1$$

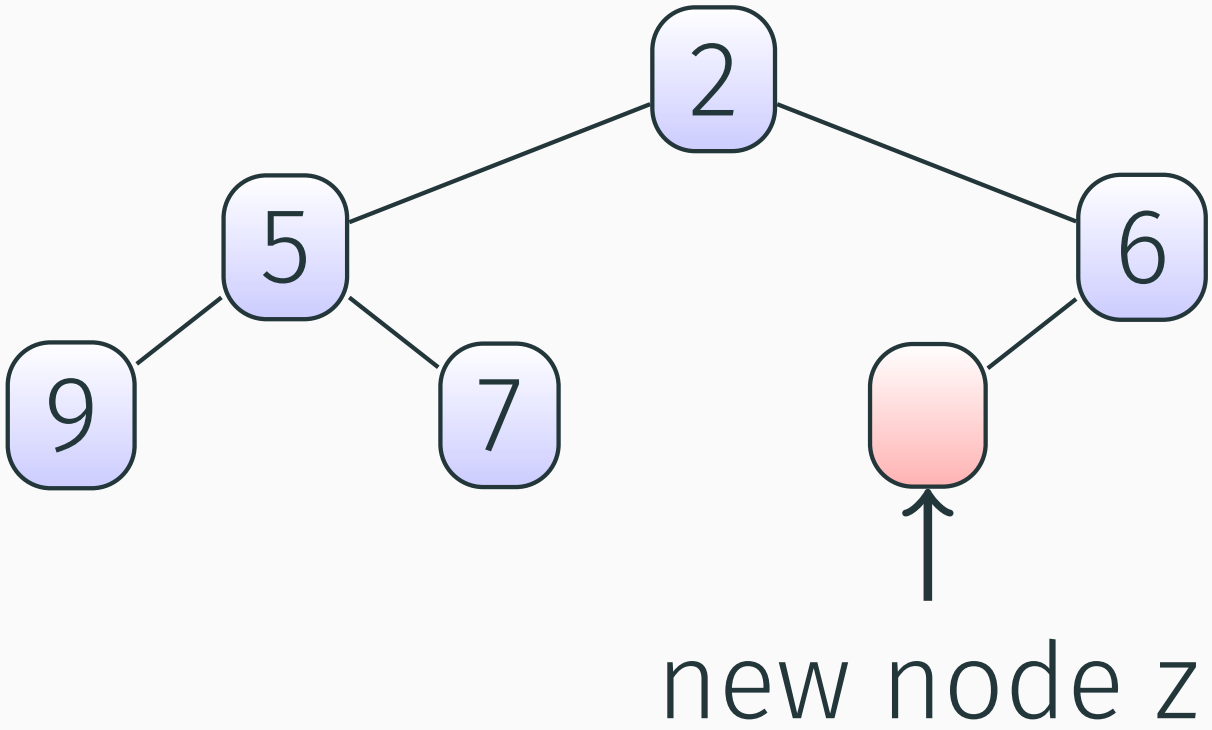
Thus, $n \geq 2^h$, *i.e.*, $h \leq \log n$



Insert into a Heap

The insertion algorithm consists of three steps

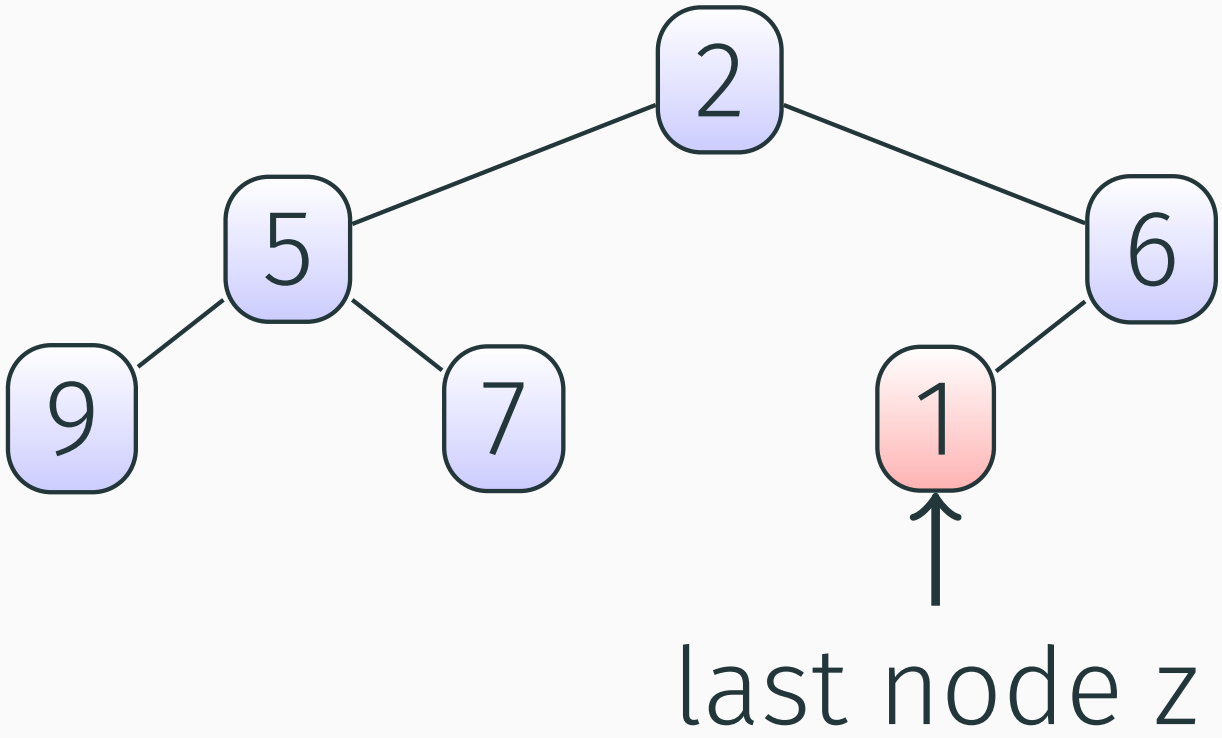
- Find the insertion node z (the new last node)
- Store k at z
- Restore the heap-order property (Upheap)



Insert into a Heap

The insertion algorithm consists of three steps

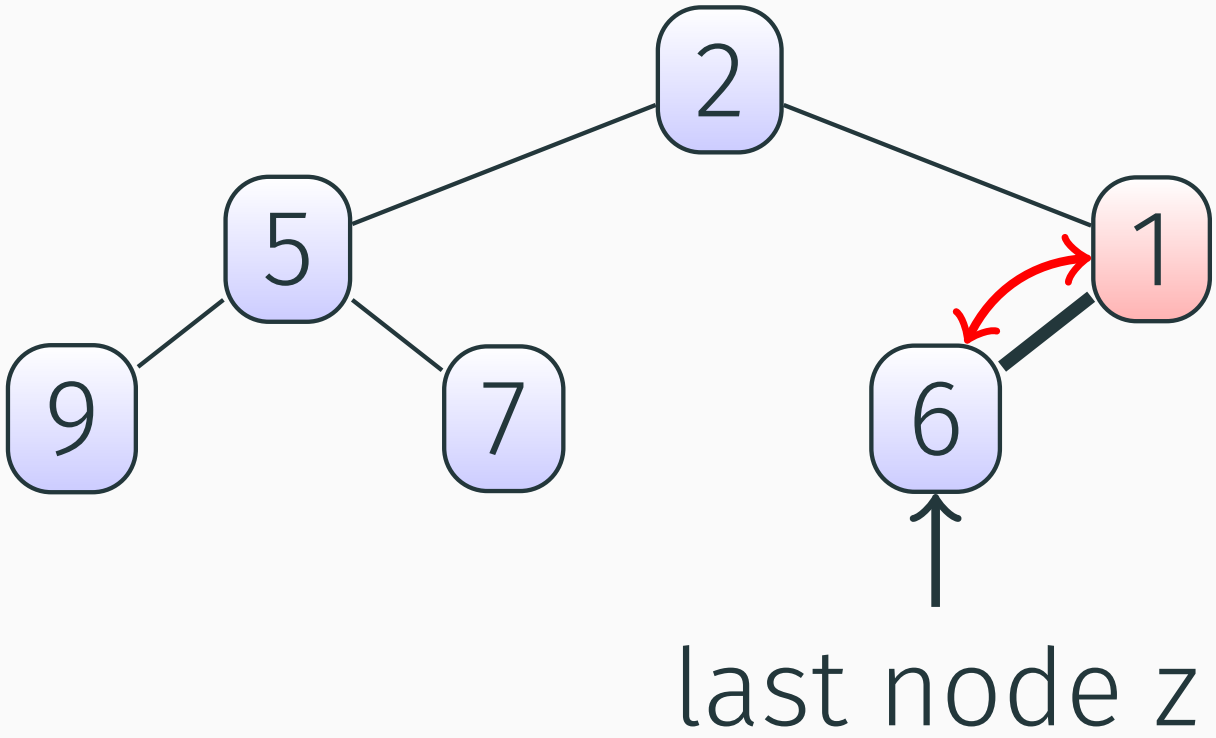
- Find the insertion node z (the new last node)
- Store k at z
- Restore the heap-order property (Upheap)



Insert into a Heap

The insertion algorithm consists of three steps

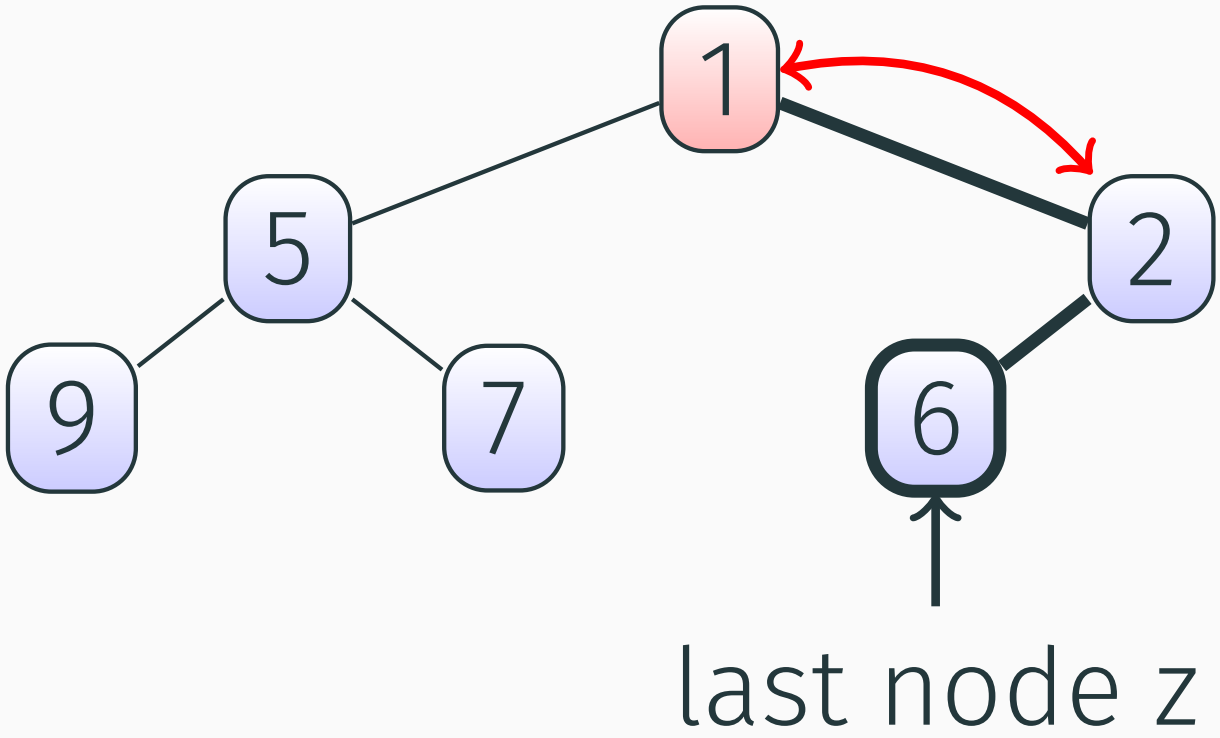
- Find the insertion node z (the new last node)
- Store k at z
- Restore the heap-order property (Upheap)



Insert into a Heap

The insertion algorithm consists of three steps

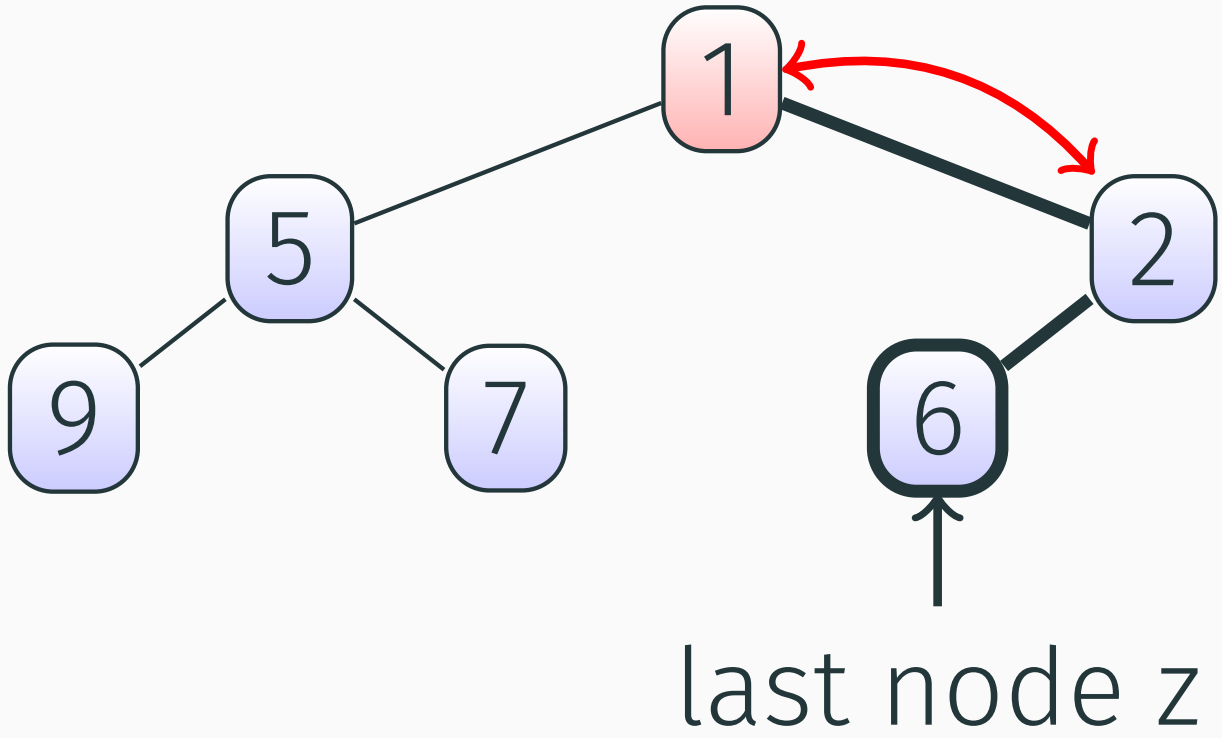
- Find the insertion node z (the new last node)
- Store k at z
- Restore the heap-order property (Upheap)



Insert into a Heap

The insertion algorithm consists of three steps

- Find the insertion node z (the new last node)
- Store k at z
- Restore the heap-order property (Upheap)
  Upheap runs in $O(\log n)$ time

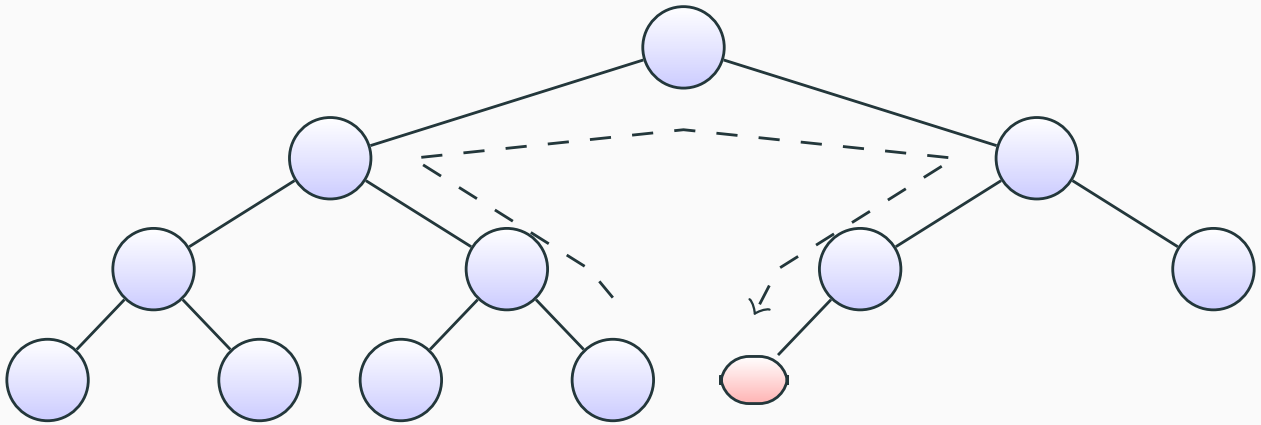


Updating the Last Node

The insertion node can be found by traversing a path of $O(\log n)$ nodes

- Go up until a left child or the root is reached
- If a left child is reached, go to the right child
- Go down left until a leaf is reached

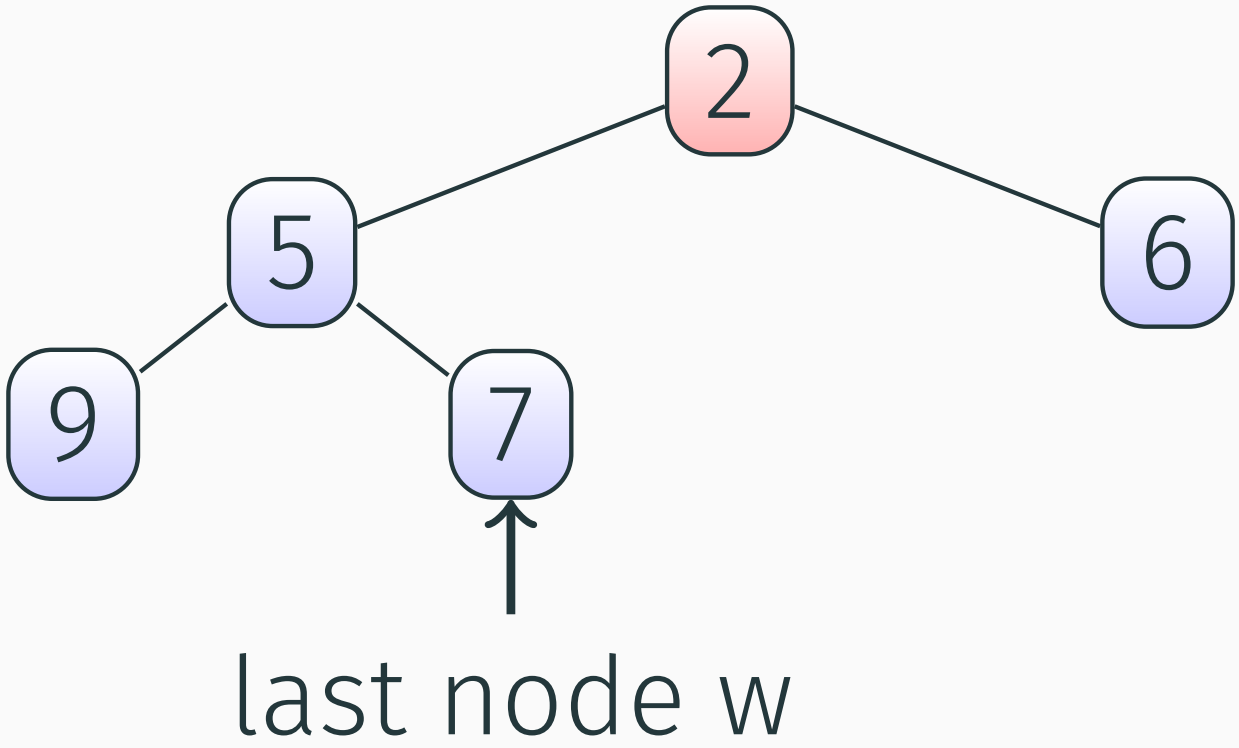
Similar algorithm for updating the last node after a removal



Removal from a Heap

The removeMin algorithm consists of three steps

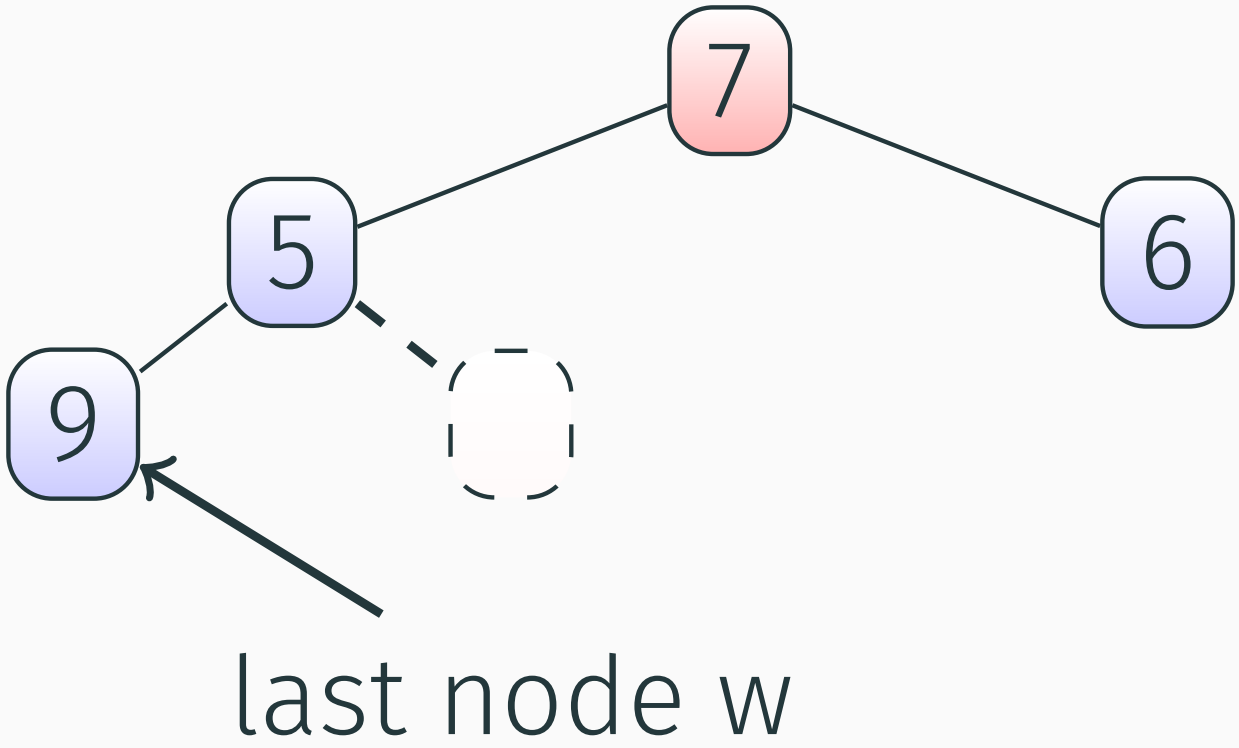
- Replace the root key with the key of the last node w
- Remove w
- Restore the heap-order property (DownHeap)



Removal from a Heap

The removeMin algorithm consists of three steps

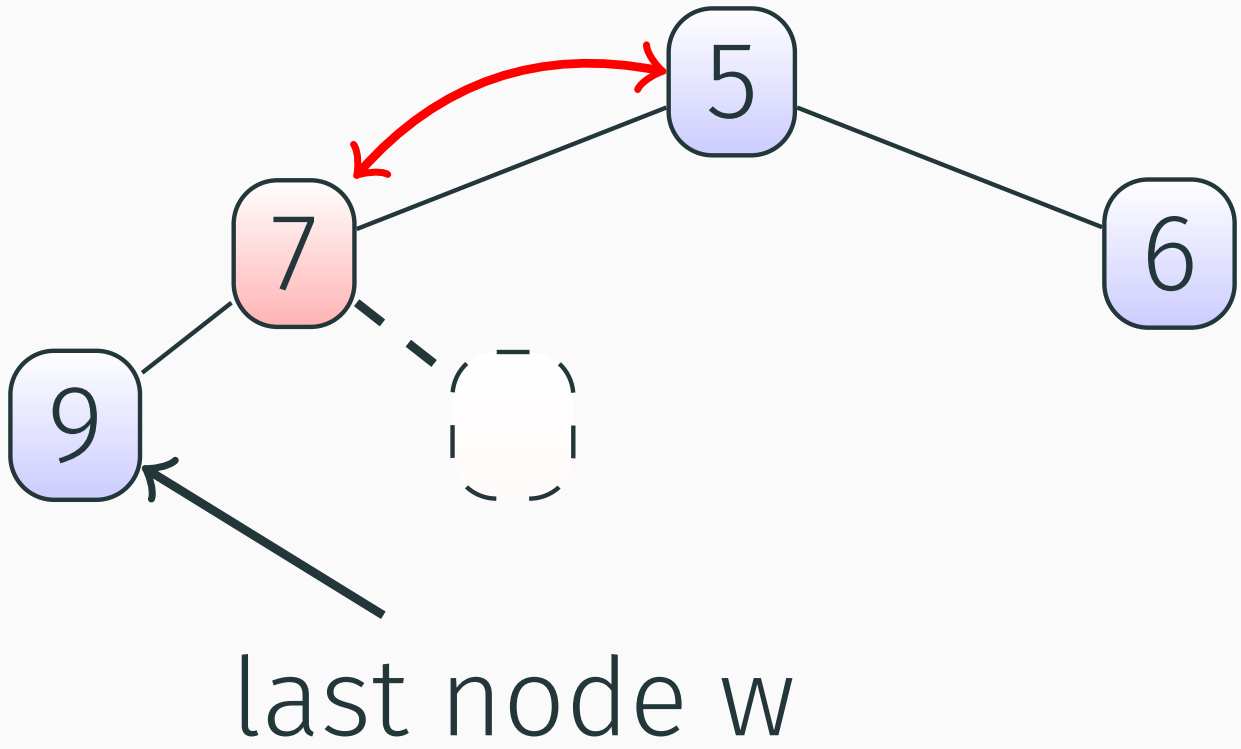
- Replace the root key with the key of the last node w
- Remove w
- Restore the heap-order property (DownHeap)



Removal from a Heap

The removeMin algorithm consists of three steps

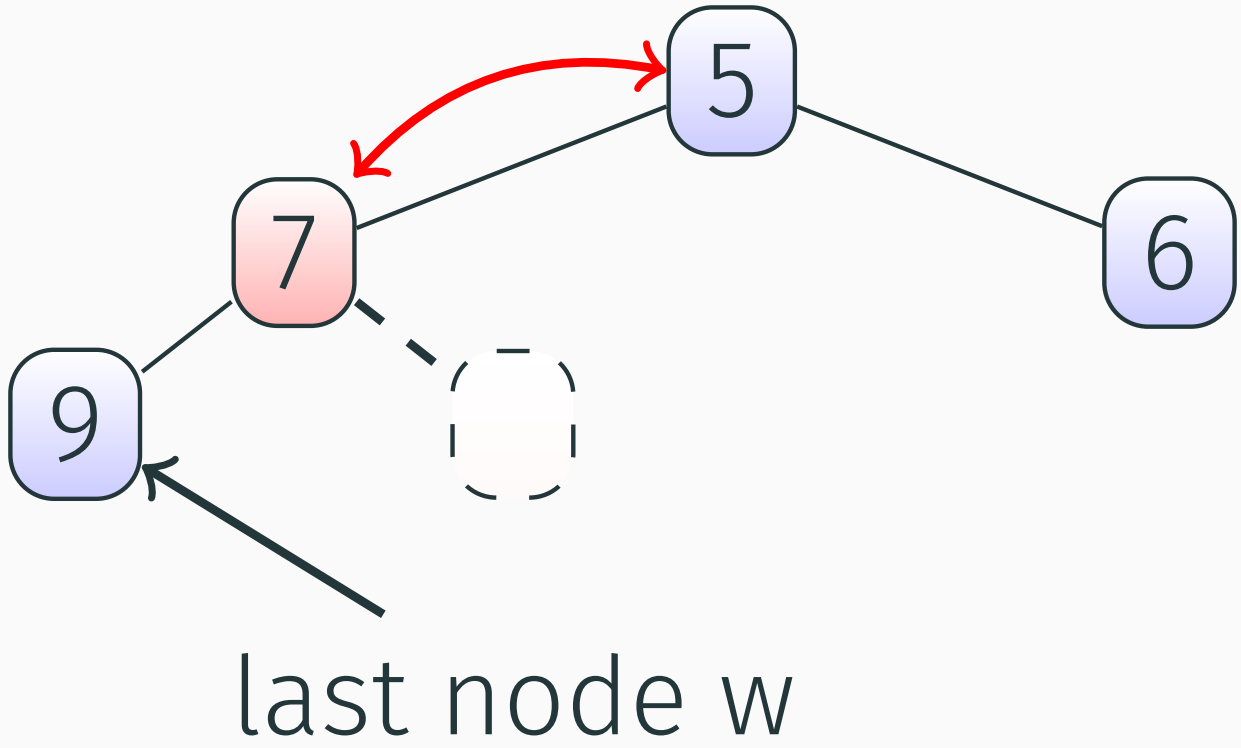
- Replace the root key with the key of the last node w
- Remove w
- Restore the heap-order property (DownHeap)



Removal from a Heap

The removeMin algorithm consists of three steps

- Replace the root key with the key of the last node w
- Remove w
- Restore the heap-order property (DownHeap)
  DownHeap runs in $O(\log n)$ time

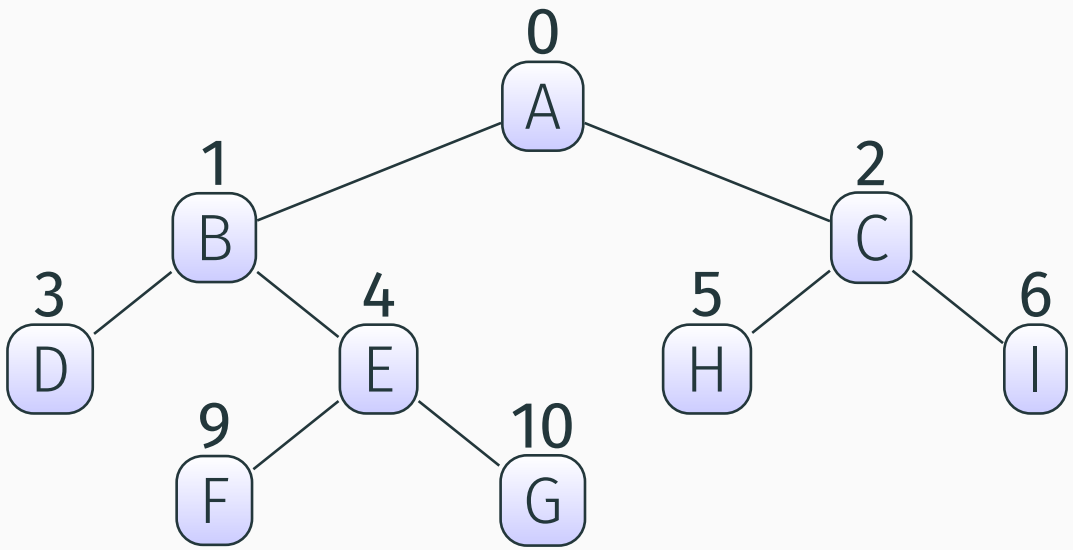


Heap

Array-based Heap Implementation

Recall: Array-Based Representation of Binary Trees

0	1	2	3	4	5	6	7	8	9	10	11
A	B	C	D	E	H	I			F	G	

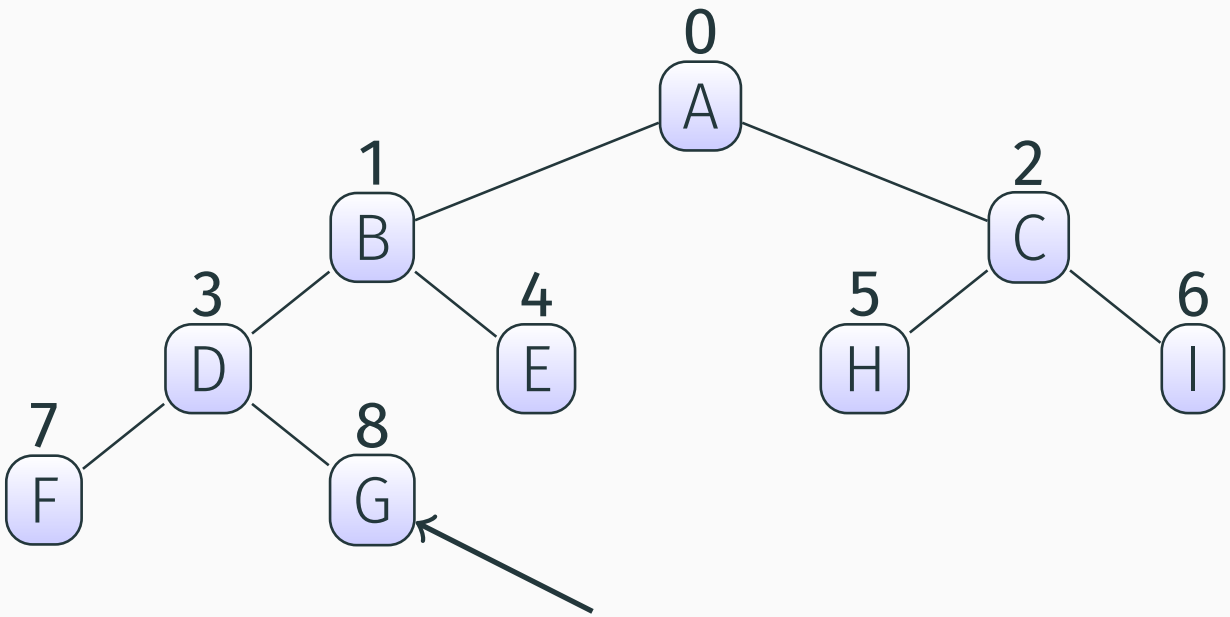


Array-based Heap Implementation

Heap

A **heap** is an (Almost) complete binary tree storing keys at its nodes and satisfying Heap-order.

Implement heap with array or arraylist.



the last node is the rightmost node in the array

Heap Construction

A trivial way is to keep inserting (key, element) pairs. **Time:** $O(n \log n)$

Heap Construction

A trivial way is to keep inserting (key, element) pairs. **Time:** $O(n \log n)$

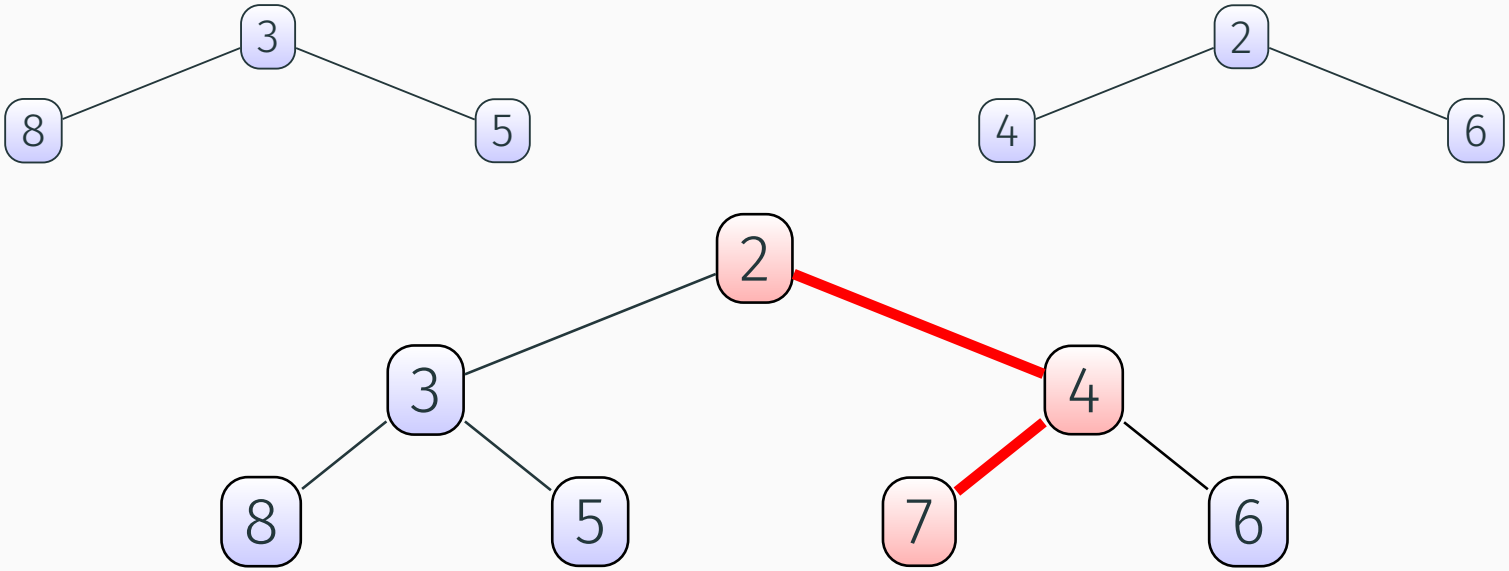
Can we make it faster?

Heap Construction

A trivial way is to keep inserting (key, element) pairs. **Time:** $O(n \log n)$

Can we make it faster?

Divide and Conquer? **Hint:** $O(n)$

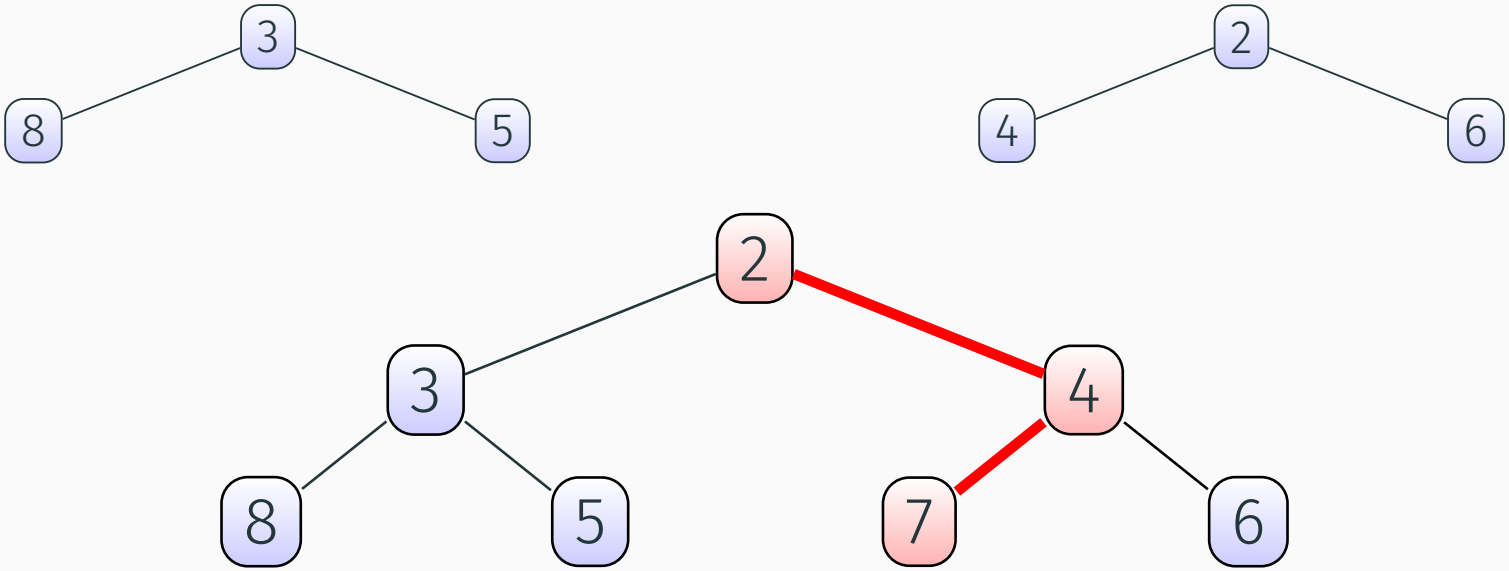


Heap Construction

A trivial way is to keep inserting (key, element) pairs. **Time:** $O(n \log n)$

Can we make it faster?

Divide and Conquer? **Hint:** $O(n)$



Time: $O(n)$

Thank you!

Questions?