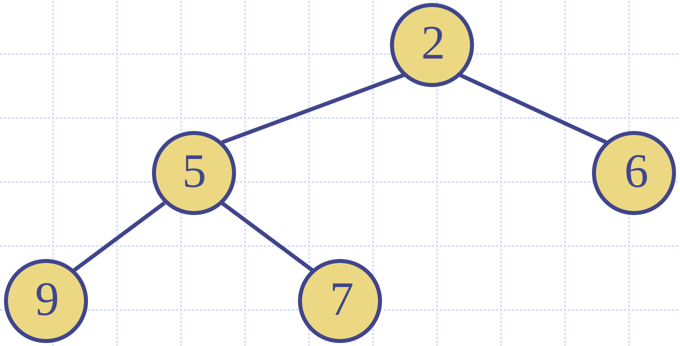
Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

Heaps



Recall Priority Queue ADT

insert(k, v)：插入一个 (key, value) 的条目
removeMin()：删除并返回当前 **最小 key 的条目**
min()：返回但不删除当前最小 key 的条目
size()：返回当前大小
isEmpty()：是否为空

- A priority queue stores a collection of entries
- Each entry is a pair (key, value)
- Main methods of the Priority Queue ADT
  - insert(k, v)
    inserts an entry with key k and value v
  - removeMin()
    removes and returns the entry with smallest key
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market
  - ...

Standby flyers（候补登机乘客排队）
Auctions（拍卖）
Stock market（股票市场，根据优先级处理交易）

Recall PQ Sorting

- We use a priority queue
  - Insert the elements with a series of insert operations
  - Remove the elements in sorted order with a series of removeMin operations
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives selection-sort: $O(n^2)$ time
  - Sorted sequence gives insertion-sort: $O(n^2)$ time
- Can we do better?

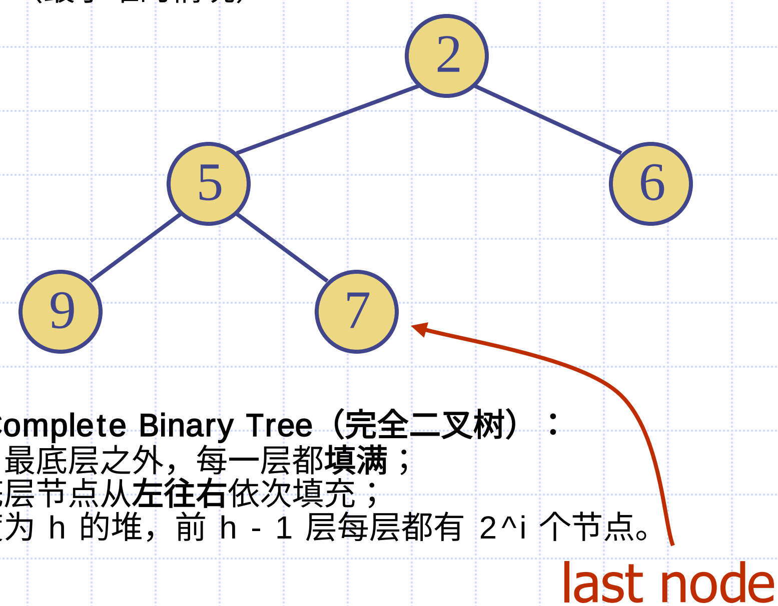
Algorithm ***PQ-Sort***(***S***, ***C***)
  Input sequence ***S***, comparator ***C*** for the elements of ***S***
  Output sequence ***S*** sorted in increasing order according to ***C***
  P ← priority queue with comparator ***C***
  while ¬***S.isEmpty*** ()
    e ← ***S.remove*** (***S. first*** ())
    P.insert (***e***, ***e***)
  while ¬***P.isEmpty***()
    e ← ***P.removeMin***().***getKey***()
    S.addLast(***e***)

Heaps (Sec.9.3)

- A heap is a *binary tree* storing keys at its nodes and satisfying the following properties:
- Heap-Order: for every node v other than the root, 意味着 **父节点的 key 最小**（最小堆的情况）
  $key(v) \geq key(parent(v))$
- Complete Binary Tree: let h be the height of the heap
  - levels $i = 0, \ldots, h-1$ have the maximal number of nodes, i.e., there are 2^i nodes at depth i.
  - The remaining nodes at level/depth h reside in the leftmost possible positions at that level/depth.

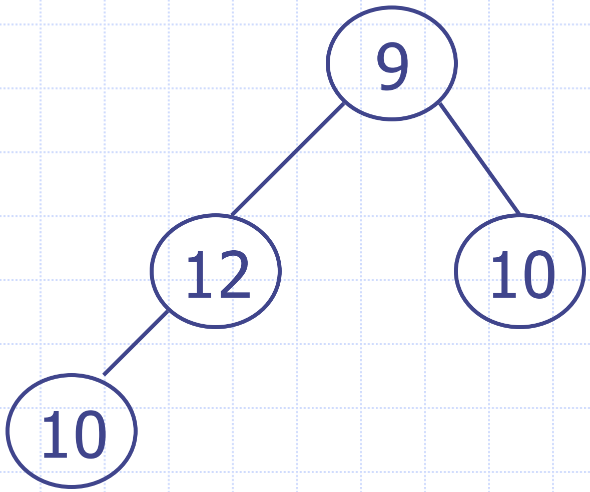
标出了**last node：最后一个节点是最深层最右边的节点**

- The last node of a heap is the rightmost node of maximum depth



2. Complete Binary Tree（完全二叉树）：
除了最底层之外，每一层都**填满**；
最底层节点从**左往右**依次填充；
高度为 h 的堆，前 h - 1 层每层都有 2^i 个节点。

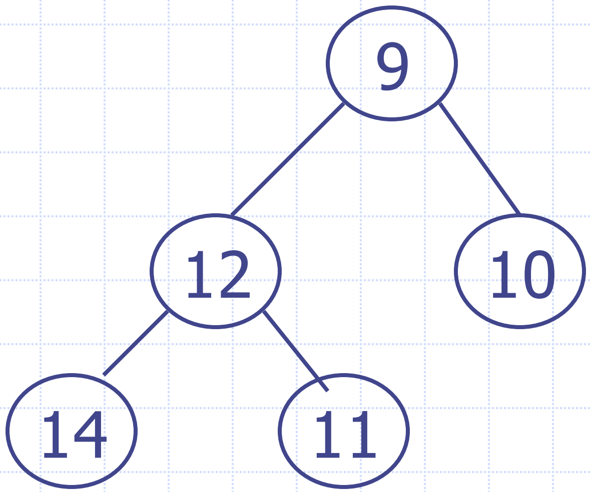
Exercise: is this a heap?



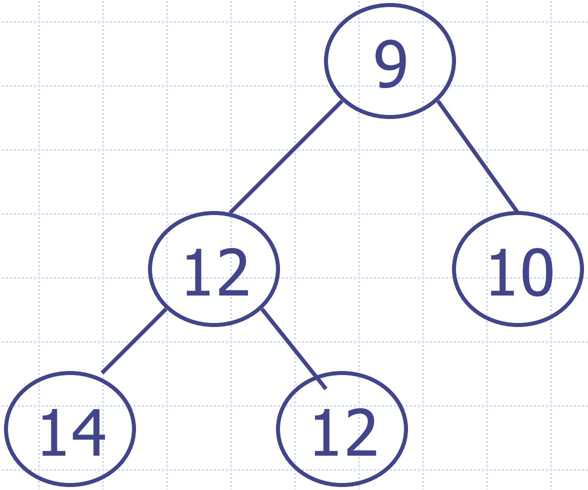
1.
堆序性（Heap-order）
：
每个节点的键值 key(v) 小于等于 key(parent (v))
即：子节点 父节点

2.
完全二叉树（Complete Binary Tree）
：
每层从左到右填满，最后一层从左到右依次填。

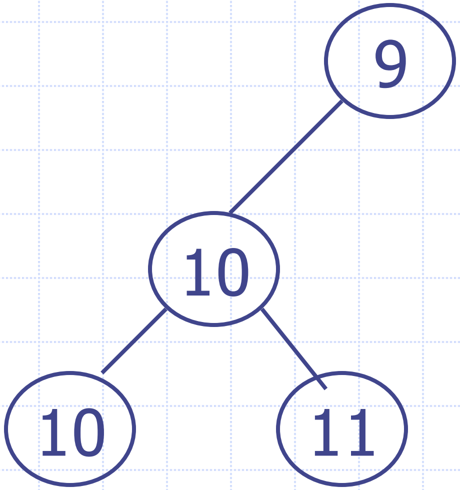
Exercise: is this a heap?



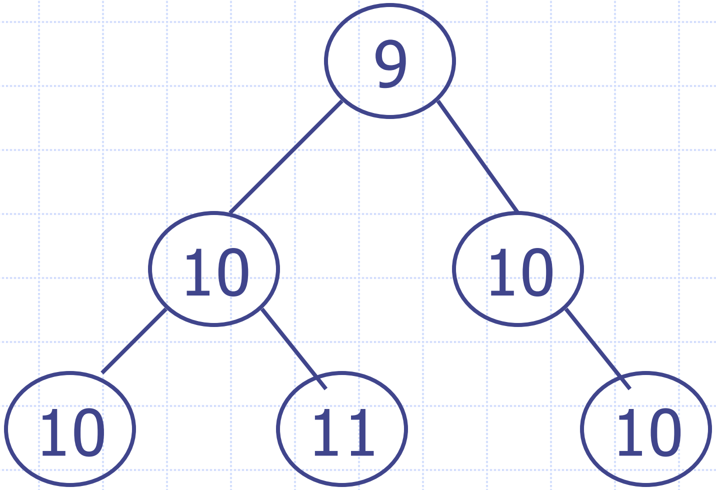
Exercise: is this a heap?



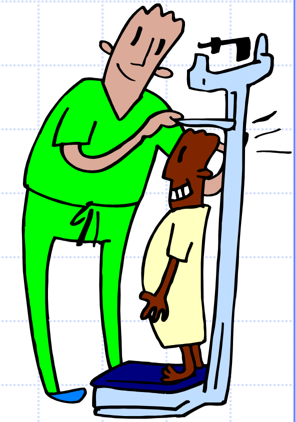
Exercise: is this a heap?



Exercise: is this a heap?

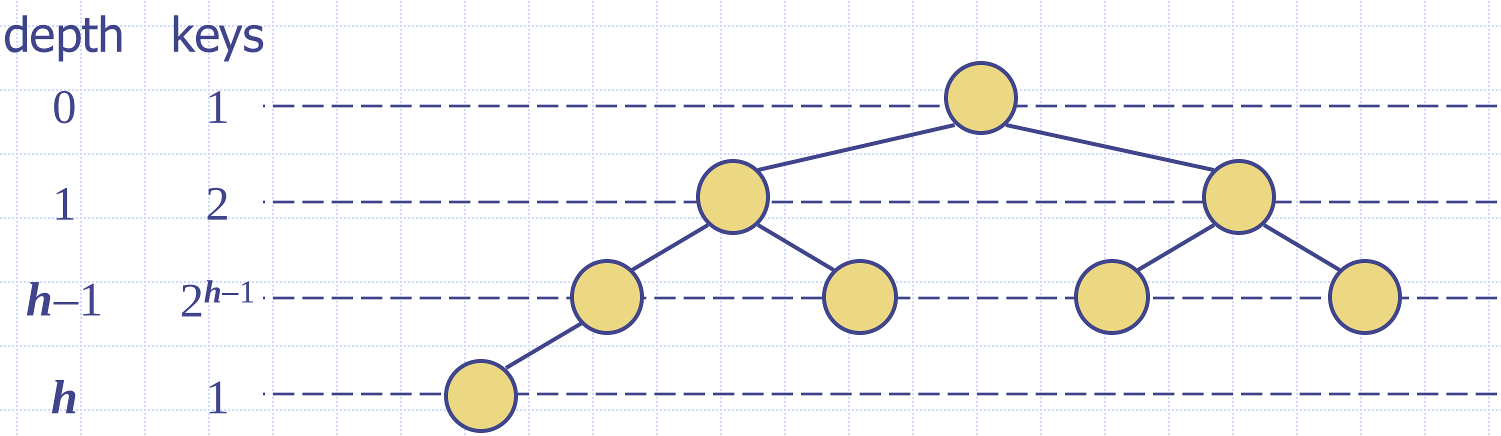


Height of a Heap



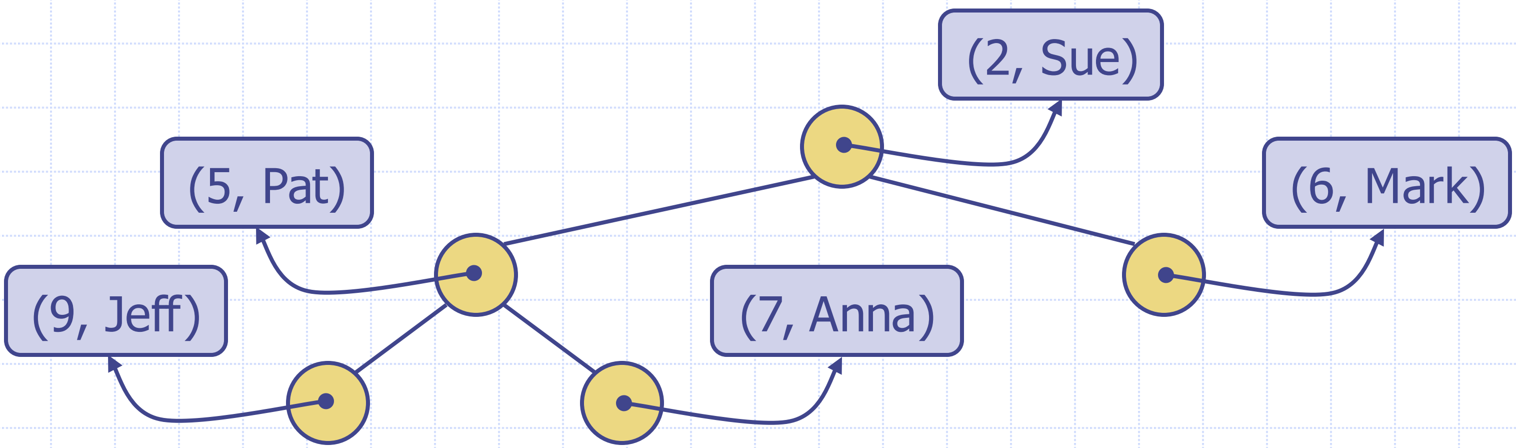
- **Theorem:** A heap storing n keys has height $O(\log n)$

  Proof: (we apply the complete binary tree property)
  - 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., $h \leq \log n$



Heaps and Priority Queues

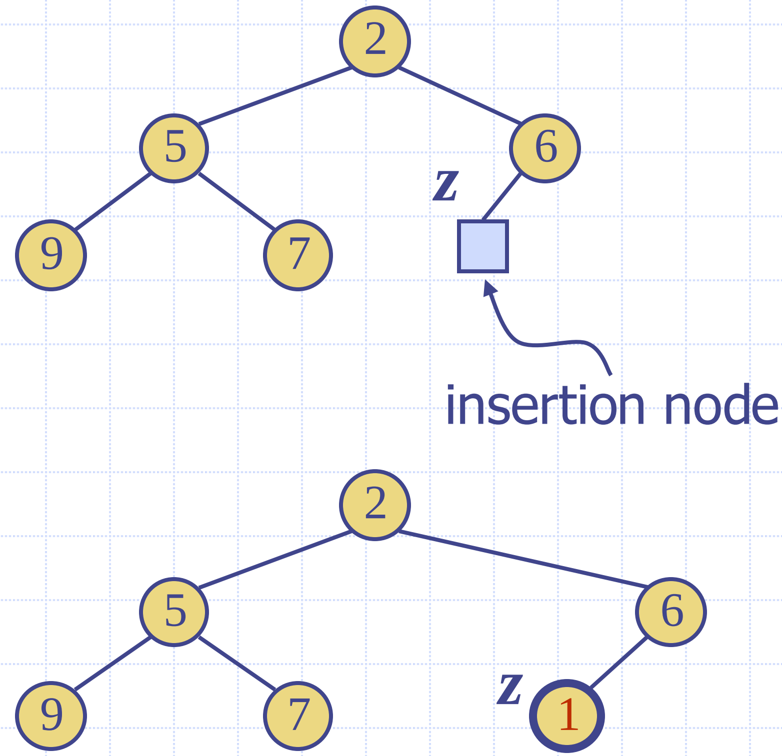
- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node



Insertion into a Heap

找到插入位置z 即新的 last node，要保持完全二叉树的结构。
将新 key 插入该节点 如图中将 1 插入到了 z 节点。
恢复堆序性（heap-order） 因为插入的新 key 可能破坏了堆的顺序，需要**上滤（upheap）**调整。

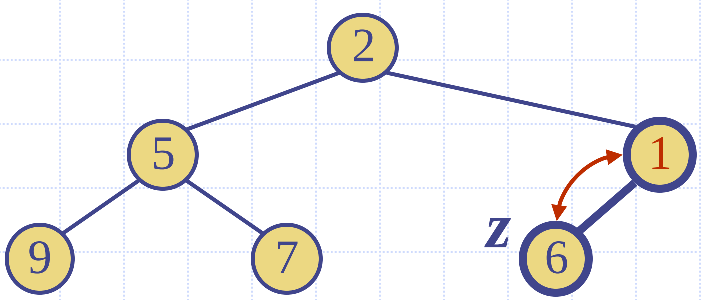
- Method *insertItem* of the priority queue ADT corresponds to the insertion of a key ***k*** to the heap.
- The insertion algorithm consists of three steps
  - Find the insertion node ***z*** (the new last node)
  - Store ***k*** at ***z***
  - Restore the heap-order property (discussed next)

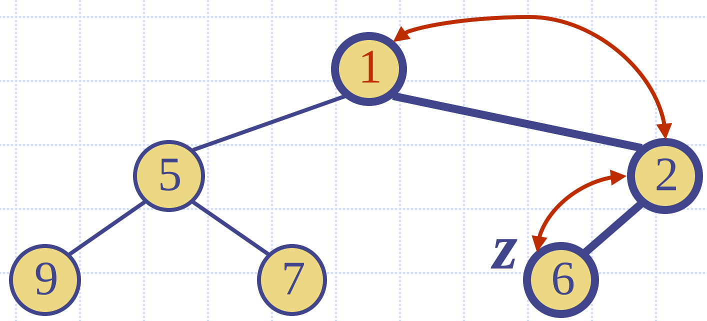


当我们把一个新的 key k 插入堆之后，它被放在“最后一个位置”——但这个 key 可能违反了堆的顺序（heap-order）。
Upheap 就是一个向上走的过程，**不断与父节点交换，直到满足堆序性**。

Upheap

- After the insertion of a new key k, the heap-order property may be violated.
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node.
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k.
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time.





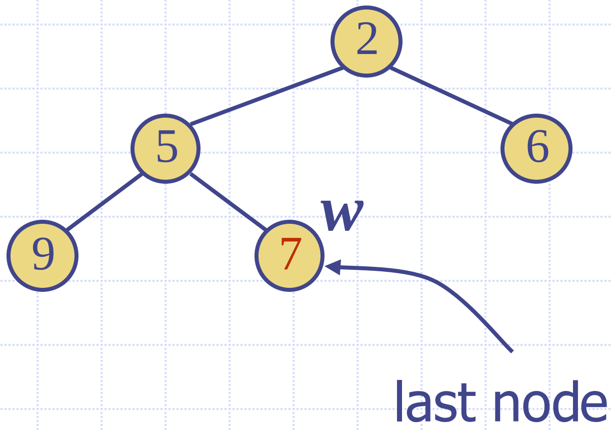
Exercise

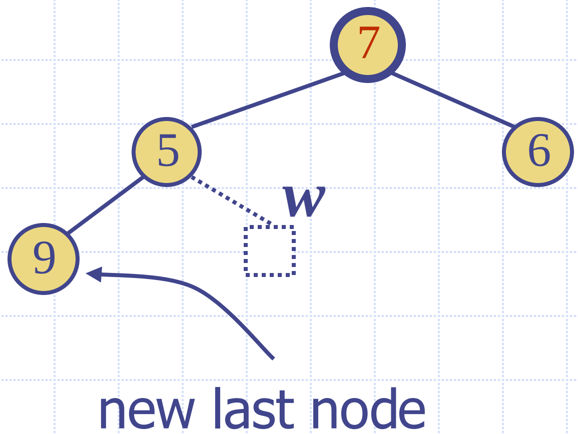
- Convince yourself that the upheap method is correct.
- This is, after the upheap method is completed, the result is a heap.

Step 1：把最后一个节点的 key（记为 w）放到根节点
原本最小的 key 在 root（这里是 2）。最后一个节点是 7 将其替代 root 的值。
Step 2：删除最后一个节点 w
现在 7 处于根节点。删除原来最后的那个节点，使树仍然是完全二叉树。
Step 3：恢复堆序性（Heap Order Property）原来最小的元素被删了，现在根节点是 “ 临时放上来的大元素 ” 很可能不满足堆序性。使用 **downheap**（也叫 heapify、sift down）方法，把这个值向下交换，直到它小于两个孩子。

Removal from a Heap

- Method *removeMin* of the priority queue ADT corresponds to the removal of the root key from the heap.
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)

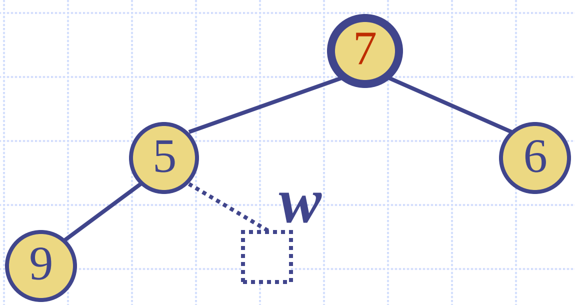


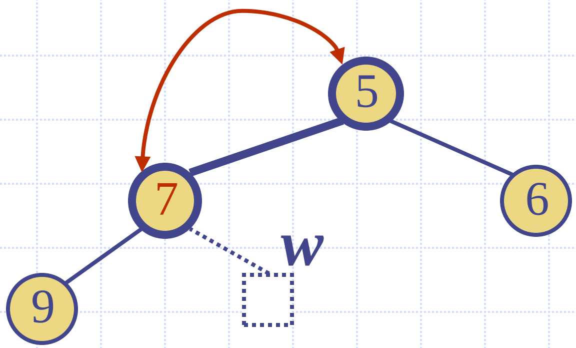


downheap 是在删除最小元素（根）后执行的操作。我们把最后一个节点的值移动到根，然后将其 “ 向下移动 ” 到合适的位置，以恢复堆序性。

Downheap

- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping the key k along a downward path from the root
- Downheap terminates when the key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time

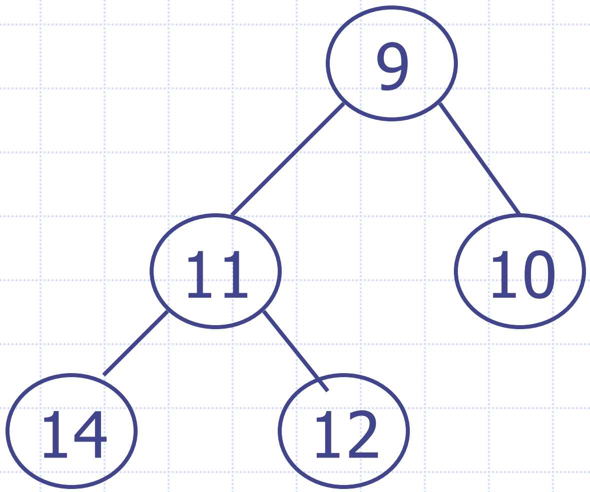




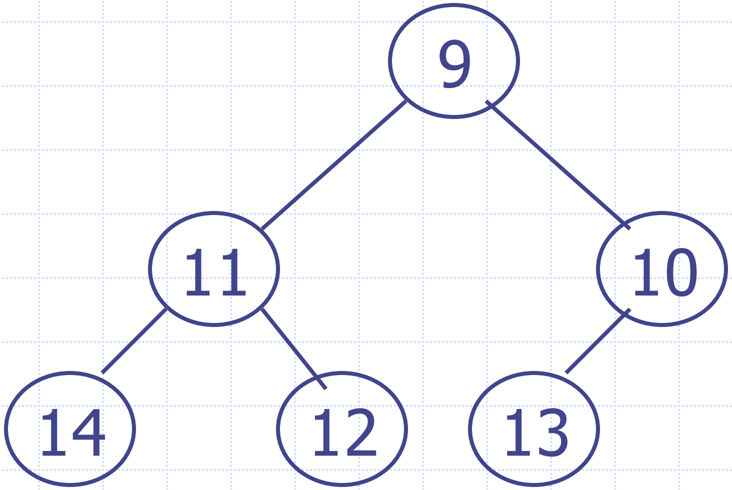
Exercise

- Convince yourself that the downheap method is correct.
- Do we need to scan the entire tree? Why?

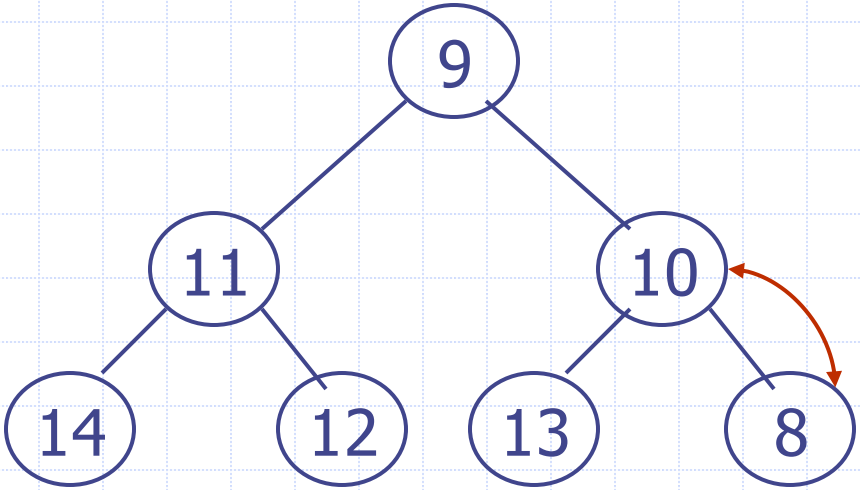
Exercise: insert 13, 8, 7



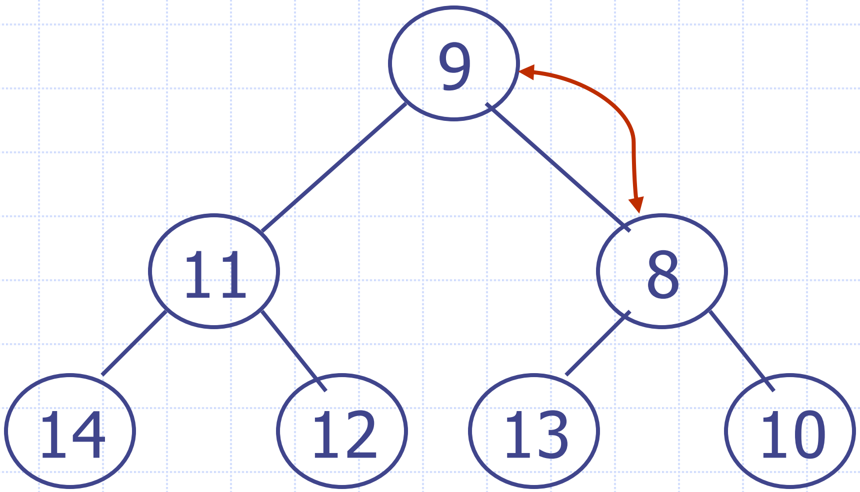
Exercise: insert 13



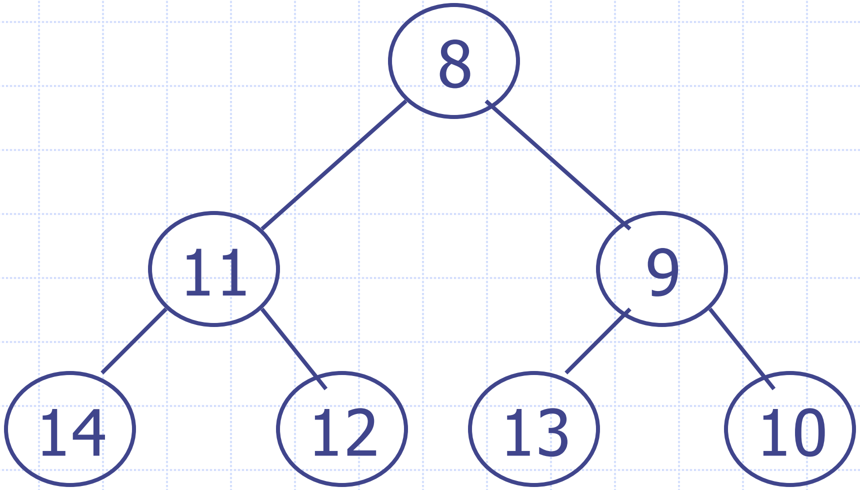
Exercise: insert 8



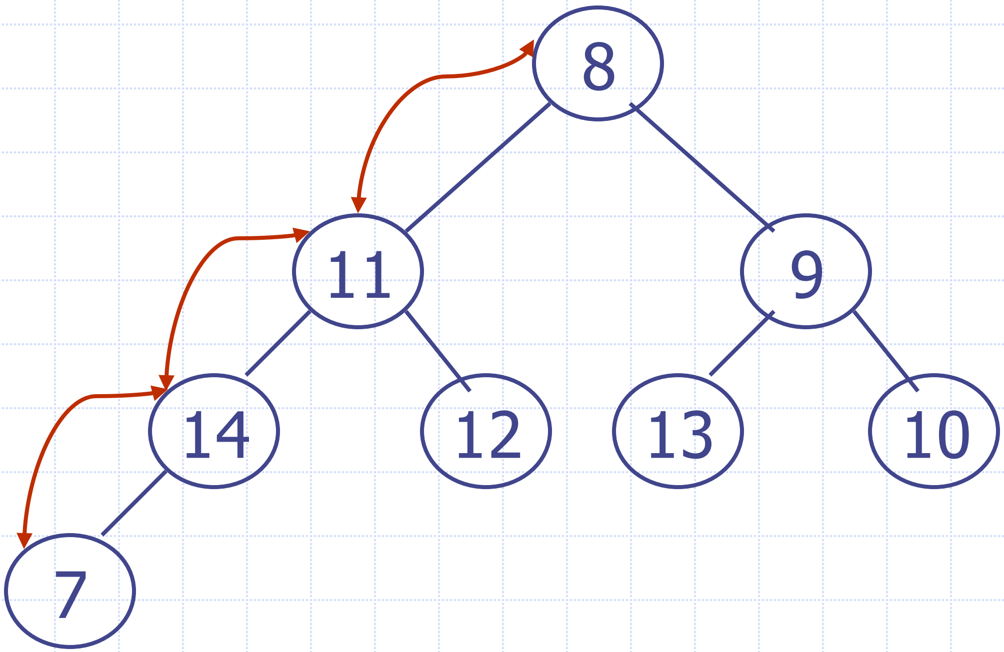
Exercise: insert 8



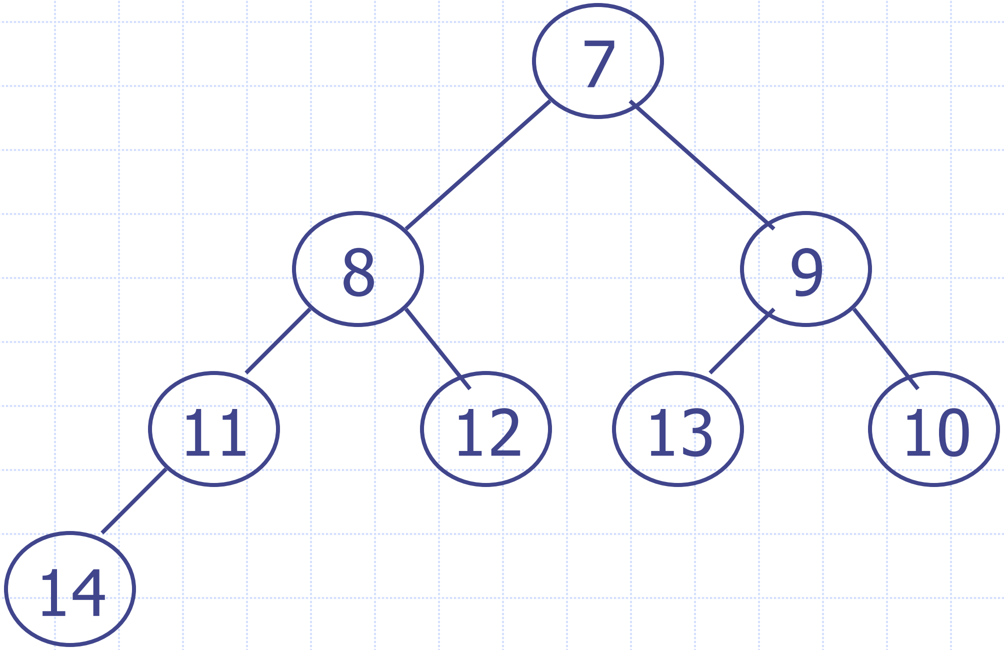
Exercise: insert 8



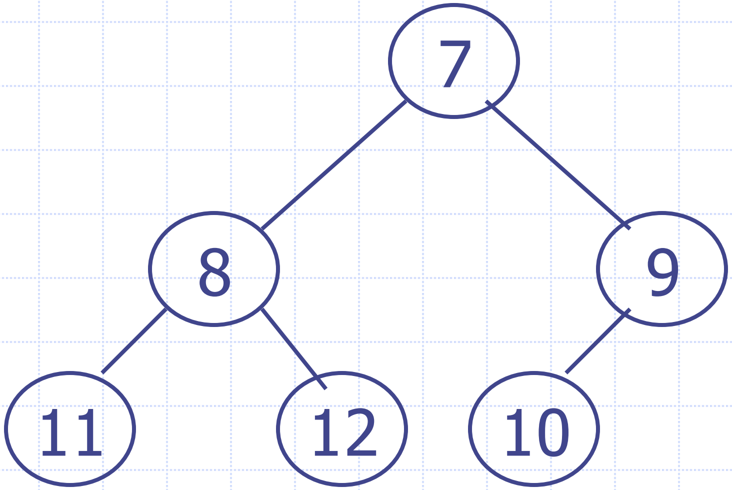
Exercise: insert 7



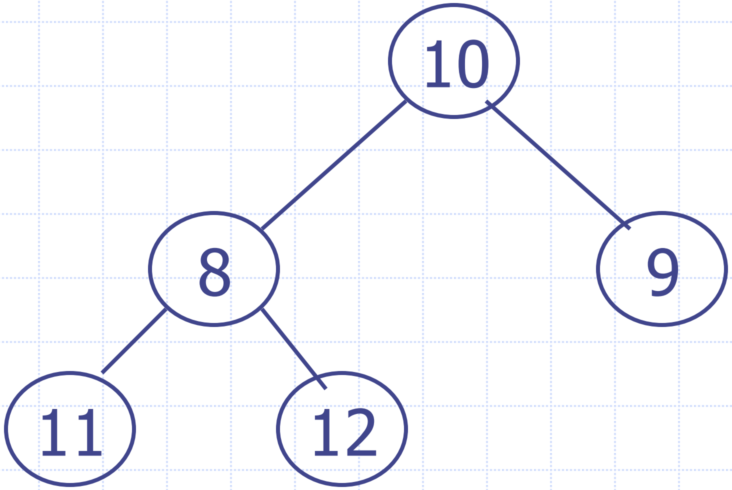
Exercise: insert 7



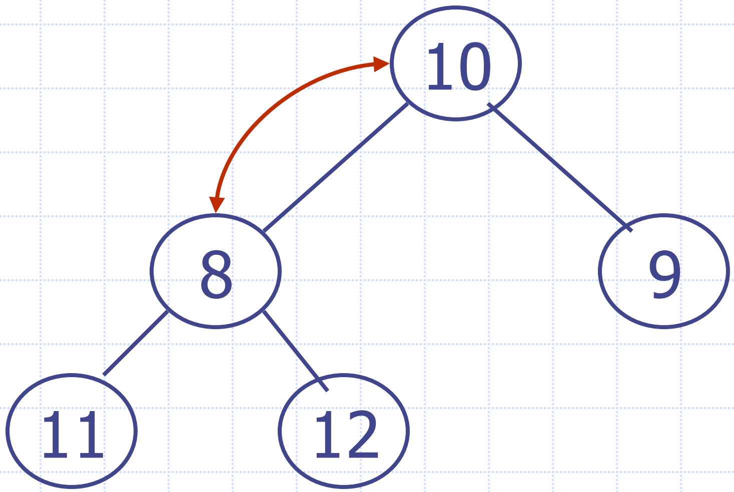
Exercise: remove 7



Exercise: remove 7

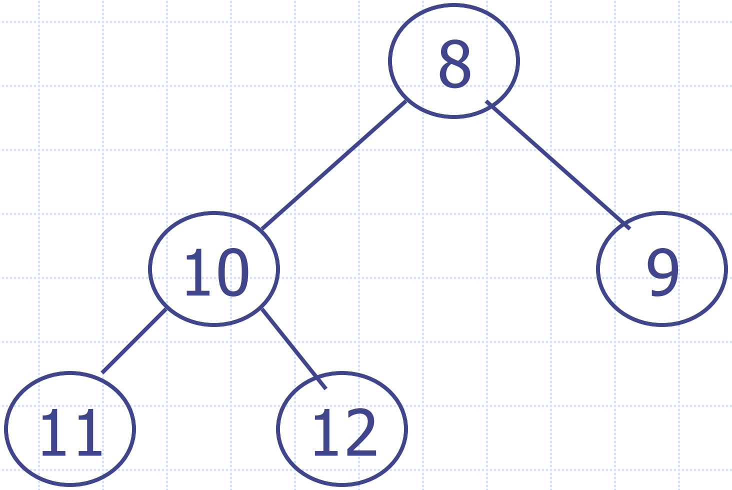


Exercise: remove 7



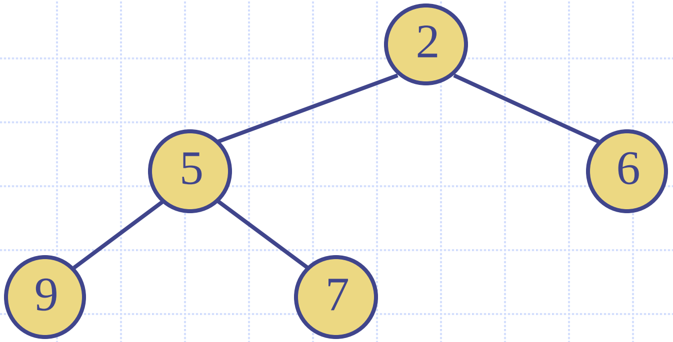
Always swap with the smaller child!

Exercise: remove 7



Array-based Heap Implementation

- We can represent a heap with n keys by means of a vector or ArrayList of length $n + 1$
- The cell of at index 0 is not used
- Links between nodes are not explicitly stored
- For the node at index i
  - the left child is at index $2i$
  - the right child is at index $2i + 1$
- Operation *insert* corresponds to inserting at index $n + 1$
- Operation *removeMin* corresponds to moving index n *to* index 1



	2	5	6	9	7
0	1	2	3	4	5

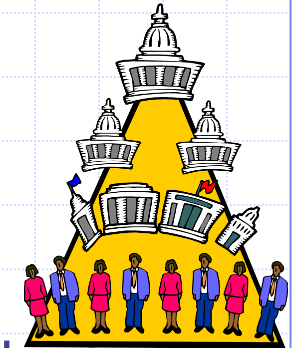
Implementing Priority Queue with a Heap

- To create a priority queue, initialise a heap
- To insert in the priority queue, insert in the heap
- To get the value with the minimal key, ask for the value of the root of the heap
- To dequeue the highest priority item, remove the root and return the value stored there.

初始化优先队列创建一个空的最小堆（min-heap）来存储键值对 (key, value)。
插入元素使用 insert 将新元素插入堆中，并通过 upheap 操作维护堆序性。
访问最小元素（但不删除）查看堆顶元素即可获取当前最小的 key 对应的值（优先级最高项）。
删除最小元素（dequeue）移除堆顶元素，将最后一个元素移动到顶部，再通过 downheap 恢复堆序性，返回该值。

插入 / 删除：**O(logn)**
查询最小值：**O(1)**

Heap-Sort



- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods insert and removeMin take $O(\log n)$ time
  - methods size, isEmpty, and min take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time.
- The resulting algorithm is called *heap-sort*.
- Heap-sort is much faster than quadratic sorting algorithms, such as insertion-sort and selection-sort.

Heap-Sort Time Complexity

- Heap-sort is the variation of PQ-sort where the priority queue is implemented with a heap.
- Running time of Heap-sort:
  1. Inserting the elements into the priority queue with n insert operations takes time proportional to

     $$\log 1 + \log 2 + \cdots + \log n$$

  2. Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes time proportional to

     $$\log n + \log(n-1) + \cdots + \log 1$$

- Heap-sort runs in $O(n \log n)$ time.

Conclusion

- Priority Queue ADT can be implemented using an unsorted list, a sorted list, or a heap.
- In the first two cases, one of the methods insert and removeMin has to run in $O(n)$ time.
- For the heap implementation, both methods run in $O(\log n)$.