Lecturer: Andrew Parkes
http://www.cs.nott.ac.uk/~pszajp/

COMP2054-ADE

Priority Queues & Heaps

ADT and CDT

Terminology:

- Abstract Data Type (ADT)
  - Only concerned with specifying the interface
  - "Behaviour as seen from the outside"
  - Specifies the methods provided - and possibly requirements on their complexity, e.g. "must run in O(n)"
- Concrete Data Type (CDT)
  - Some structure and algorithm used for an actual implementation
  - "Behaviour as seen from the inside"

ADT and CDT utility

Standard object-oriented reasons:

- Separation of the "specification" from the "implementation details"
- Allows different CDTs to be explored for the same ADT
- Quickly swap or improve CDTs without users of the ADT having to change their code

- **Watch out for: difference between**
  - **names used for ADTs**
  - **names used for associated CDTs**

Priority Queue ADT

- 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
  - Note: unlike Map (seen later), there is no requirement of a method find(k).

- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key
  - size(), isEmpty()

- Applications:
  - Standby passengers
  - Auctions
  - Stock market
  - Printer queues

Portions of slides are © 2004 Goodrich, Tamassia

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined (between all different pairs)
  - Just need a "compare" operation between the keys and that behaves like <= or < as needed
- Two distinct entries in a priority queue can have the same key
  - i.e. that compare as being equal

Simple Implementations, CDT, of a Priority Queue

- Implementation with an unsorted list

- Performance:
  - insert takes $O(1)$ time since we can insert the item at the head or tail of the list
  - removeMin and min take $O(n)$ time since we have to traverse the entire list to find the smallest key

- Implementation with a sorted list

- Performance:
  - insert takes $O(n)$ time since we have to find the position to insert the item
  - removeMin and min take $O(1)$ time, since the smallest key is at the head

Exercise (offline): is singly-linked sufficient? Or is doubly-linked needed?

Portions of slides are © 2004 Goodrich, Tamassia

General Remark

- "Some operations are O(1) and others O(n), so the average is not too bad" ??
  - NO NO NO!!!
- The average could still be O(n)
- Suppose, that
  - 99% take time 1
  - 1% take time n

  The average (n+ 99)/100 is still O(n)
- This only improves if the worst-case operations are 'rare', otherwise the expensive operations still dominate
  - E.g. see the study of amortised complexity. If the resizing is "rare" then it does not dominate the cost.

Aim:

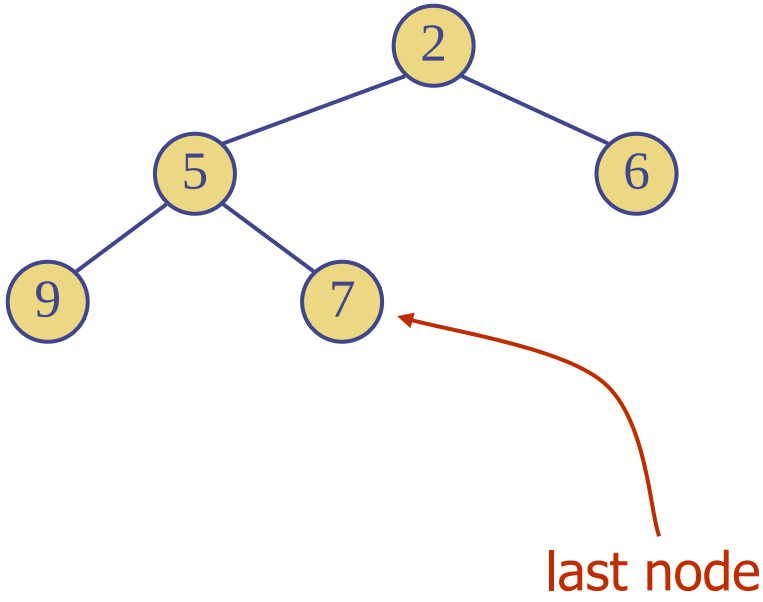
- Can we find an implementation of a PQ such that **all** operations are better than O(n)?
  - I.e. want all operations o(n).
  - E.g. aim for them all to be O(log n)

- We can do this by implementing a Priority Queue using a kind of binary tree called a 'heap':

Heaps

- A heap is a binary tree storing key-value pairs at its nodes and satisfying the following properties:
  - Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
  - 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 nodes are to the left of any 'missing nodes'
      - in this case the children of 6 are 'missing'

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

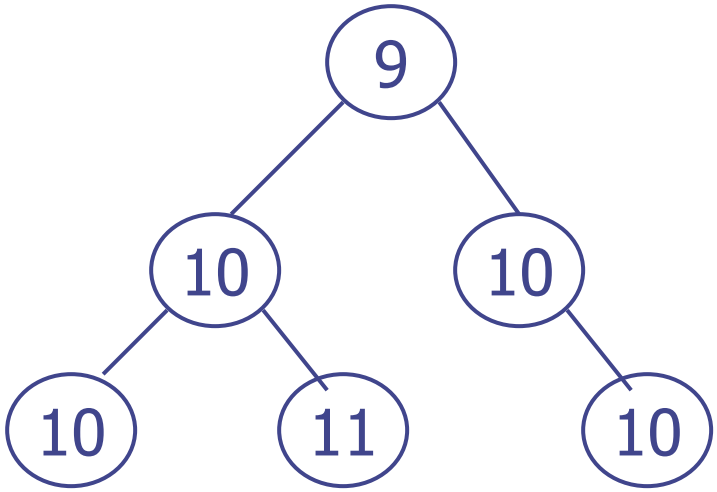


Portions of slides are © 2004 Goodrich, Tamassia

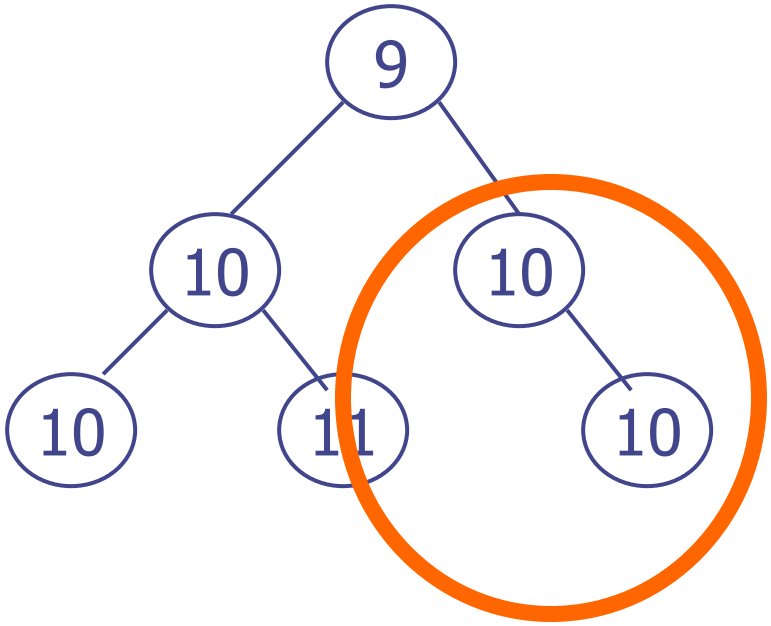
Note on terminology:

- The previous definition of "heap" is a standard one for a "Binary Heap"
  - It is used/assumed in this module.
- Other notions of "heap" do exist
  - they keep the numeric ordering of parent relative to children
  - but can have different conditions for the "shape"
    - E.g. "Binomial" and "Fibonacci" heaps

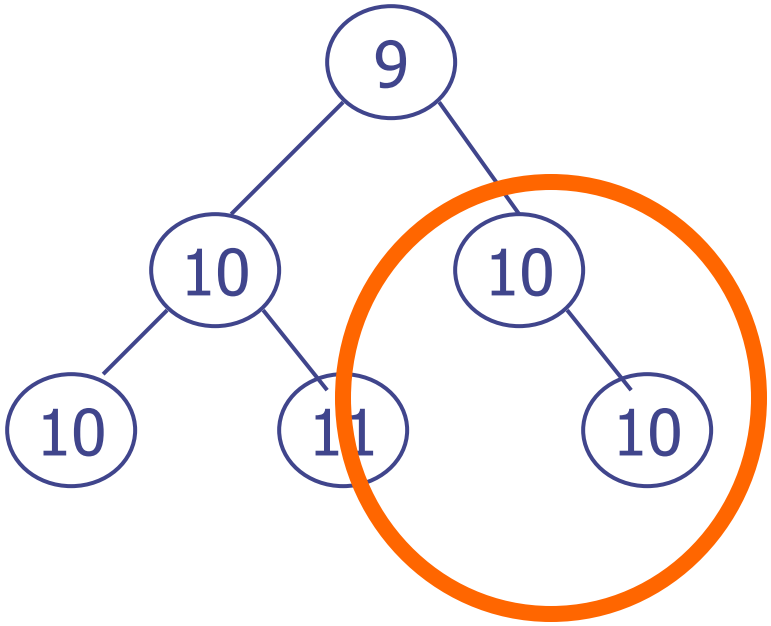
Exercise: is this a heap?



Exercise: is this a heap?



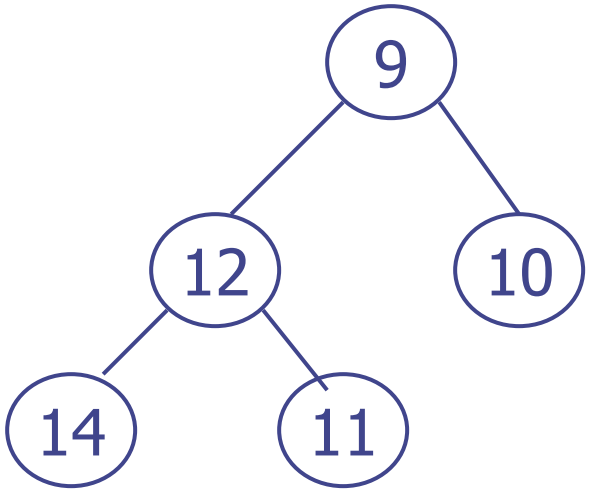
Exercise: is this a heap?



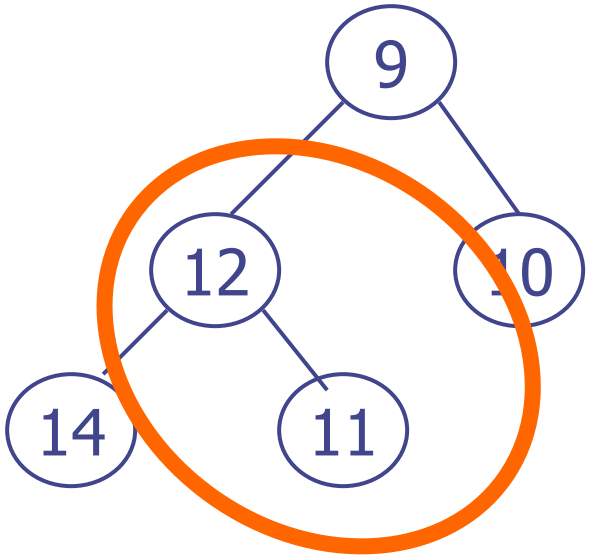
Answer is no, because the shape of the tree is wrong:

- The nodes on the bottom row are not "as leftwards as they can be"
- E.g. If a node has just one child it must be a left child.

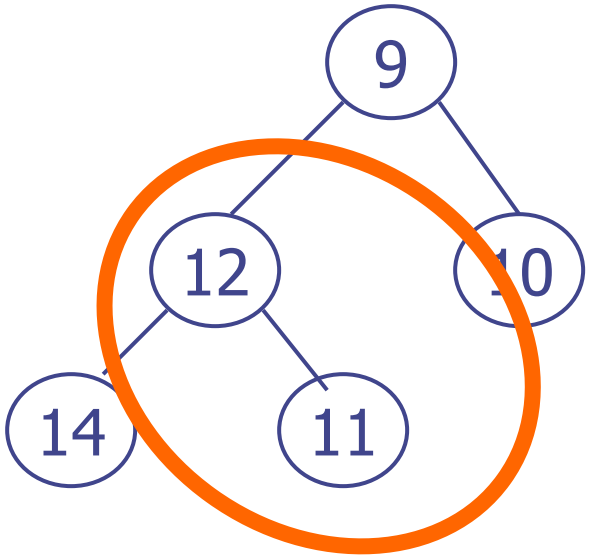
Exercise: is this a heap?



Exercise: is this a heap?



Exercise: is this a heap?



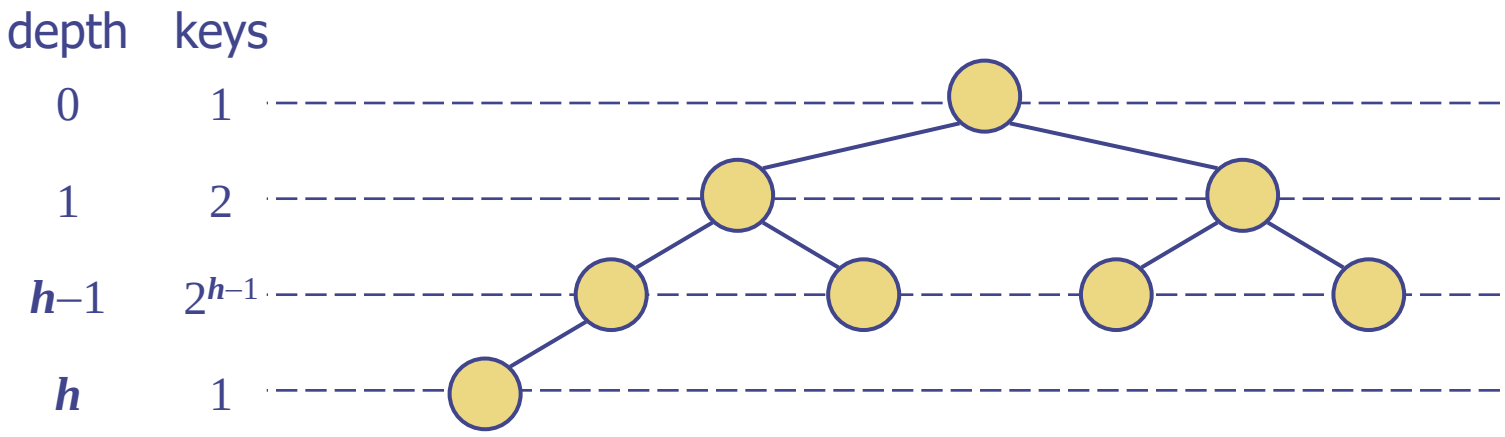
The answer is no:

- Because the numeric conditions are not satisfied
- In a heap that supports removal of a minimum element, then the children of a node can never be smaller.

Height of a Heap

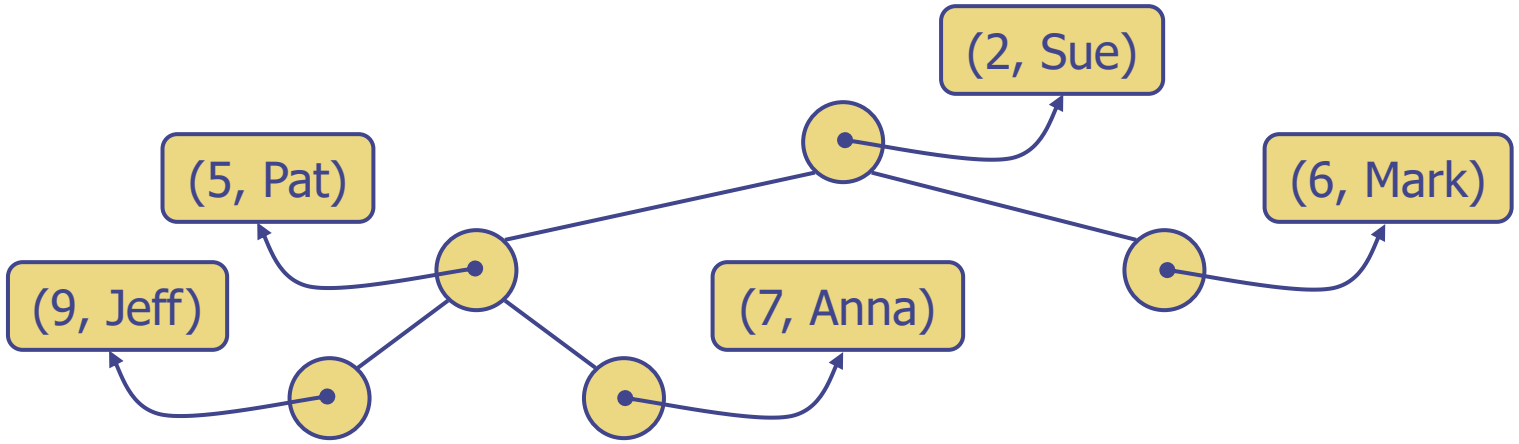
- Theorem: A heap storing n keys has height $O(\log n)$

  Proof: This uses just the complete binary tree property
  - Let h be the height of a heap storing n key
  - The perfect binary tree of height h-1 has $2^h - 1$ nodes
  - Our tree has at least one more, thus, $n \geq 2^h$, i.e., $h \leq \log n$



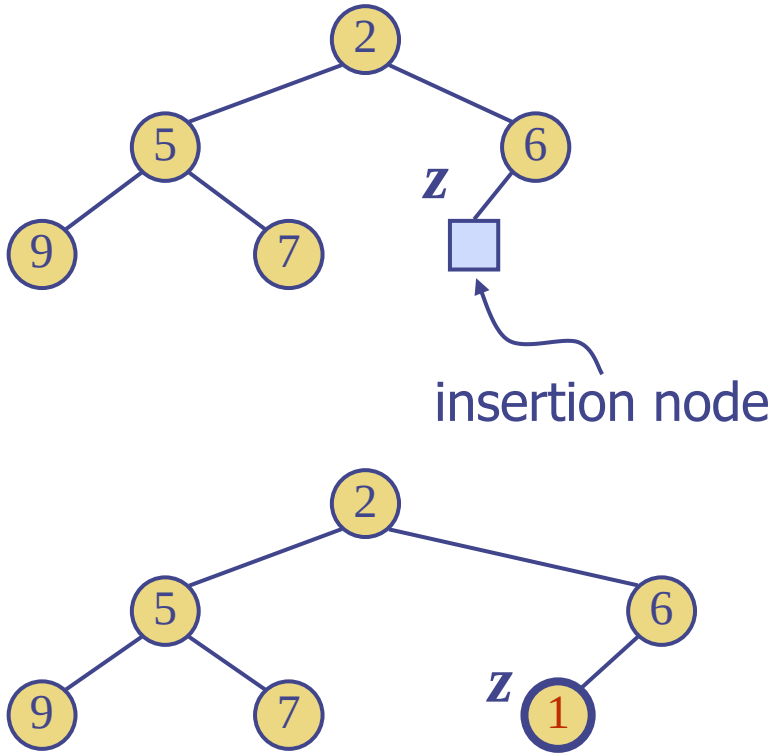
Heaps and Priority Queues

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



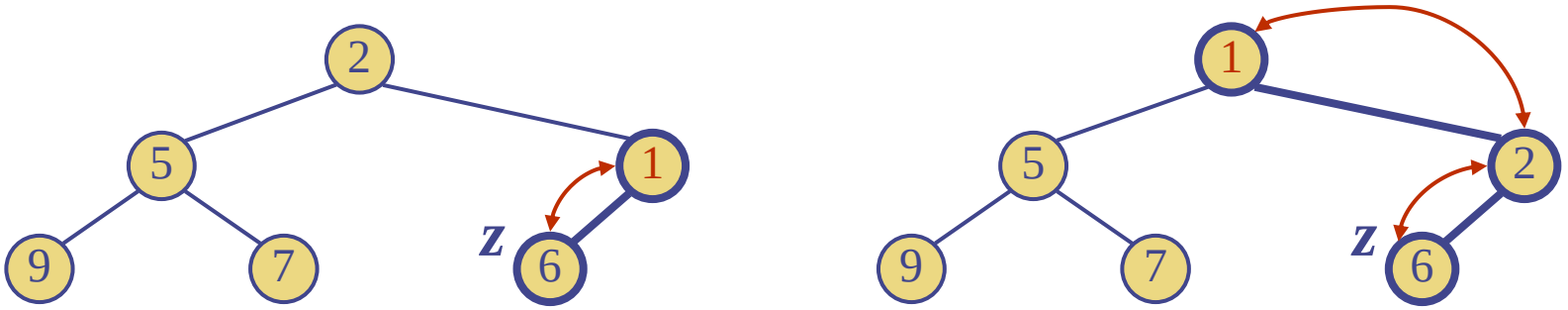
Insertion into a Heap

- 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)



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



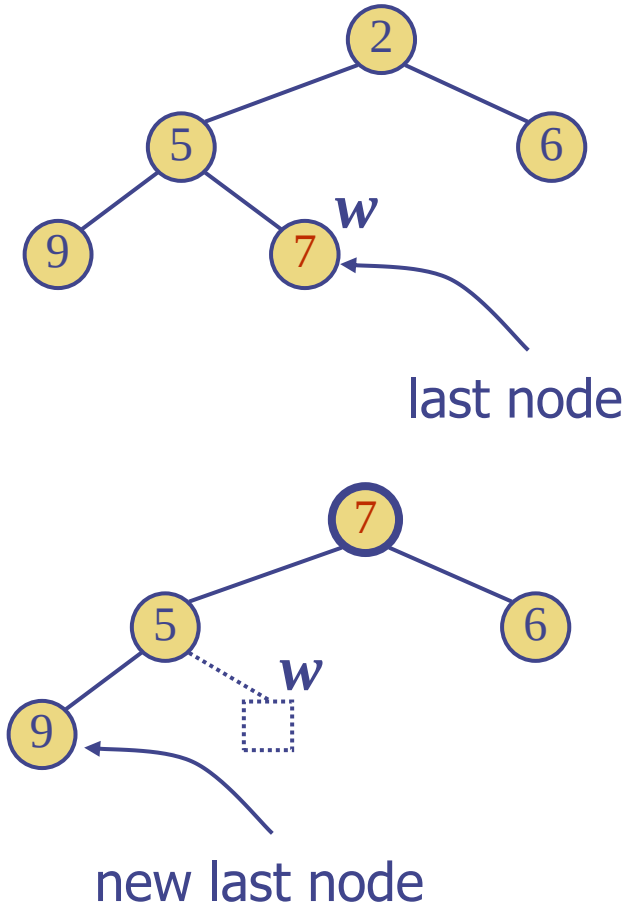
Exercises

- Convince yourself that the upheap method is correct
  - i.e. that it always results in the heap properties being satisfied after it is completed
  - It is typical in data structure algorithms that required properties are temporarily broken, but then restored

- Remark:
  - You should always strive to convince yourself of the correctness for all algorithms given to you
  - If nothing else, it encourages deeper thinking about the algorithm, and this makes it easier to
    - 'fill in the gaps' if forget parts of them ☺
    - Extend the ideas to other cases

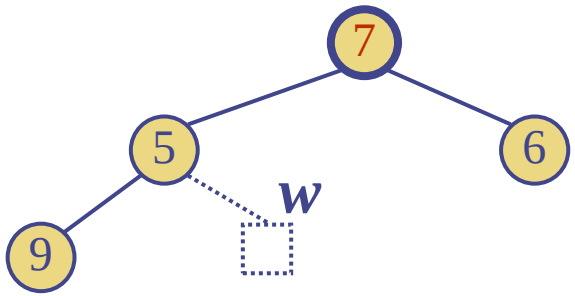
Removal from a Heap

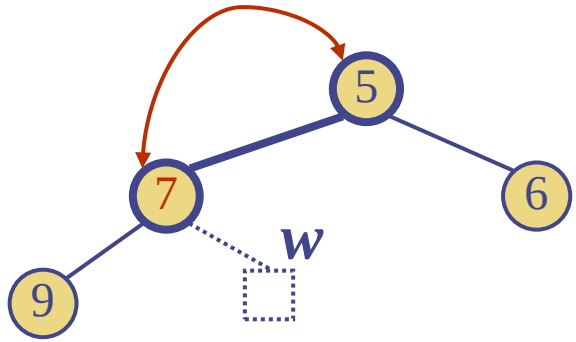
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)



Downheap

- 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 key k along a particular downward path from the root
- Downheap terminates when 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

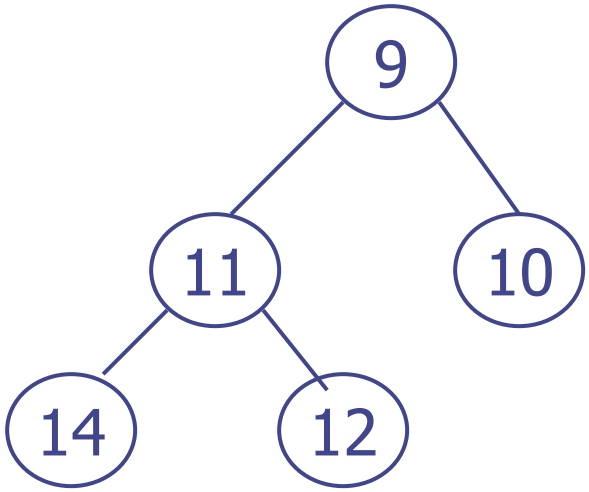




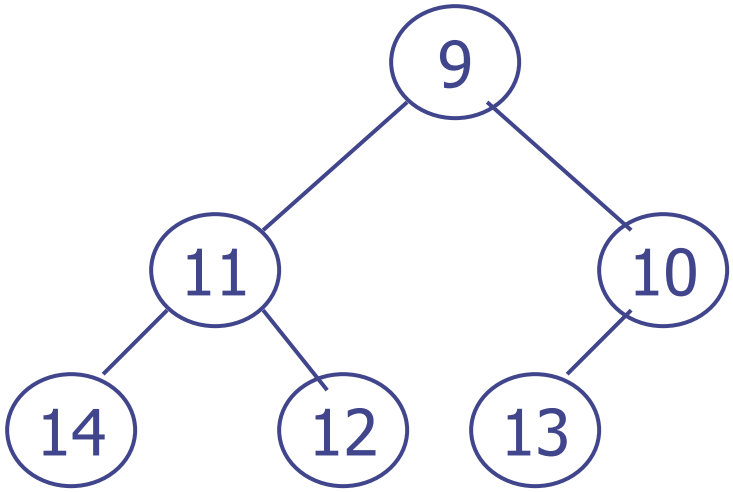
Exercise (offline)

- Be sure that you understand why the upheap and downheap operations are correct.
- In particular, why do we only need to fix nodes on the path from the "adjusted nodes" back up to the root node?
  - I.e. Why do we not need to scan the entire tree?
  - Otherwise, the operations would not be O(h)=O(log(n)) but would be O(n)
    - (that is they would be exponentially slower)

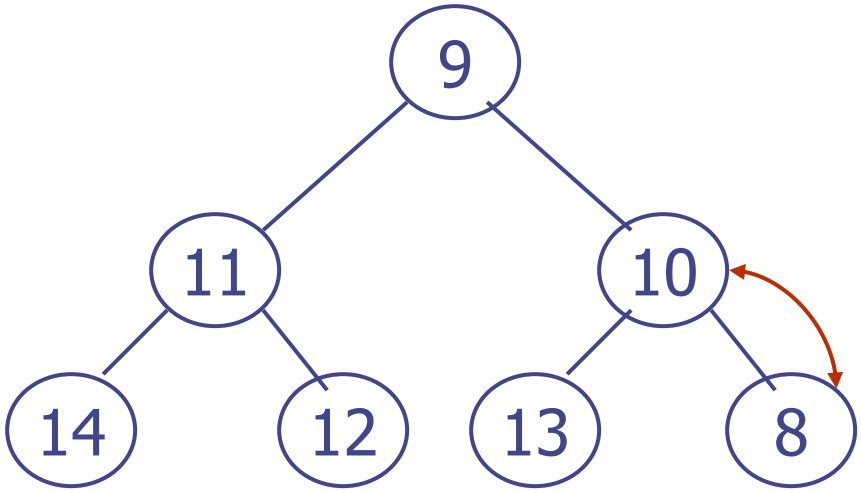
Exercise: insert 13, 8, 7



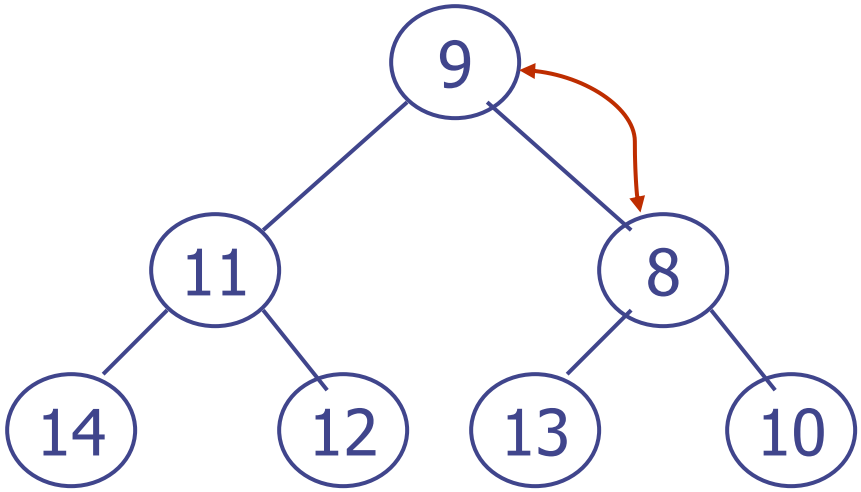
Exercise: insert 13



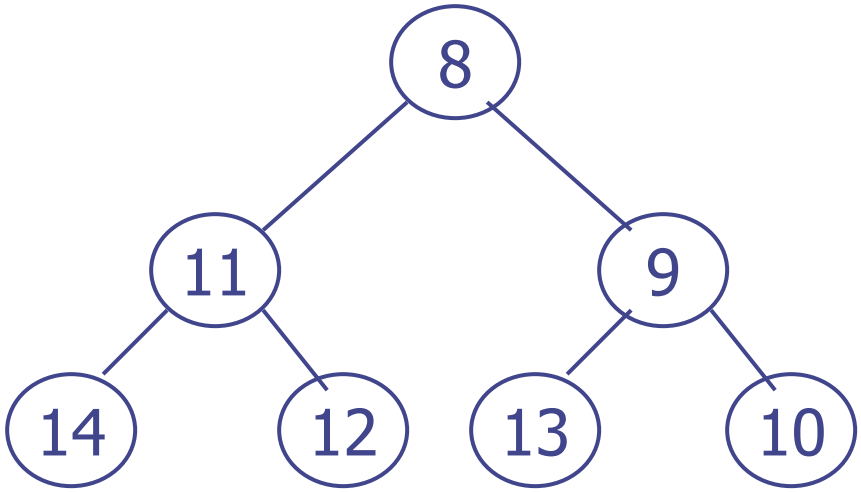
Exercise: insert 8



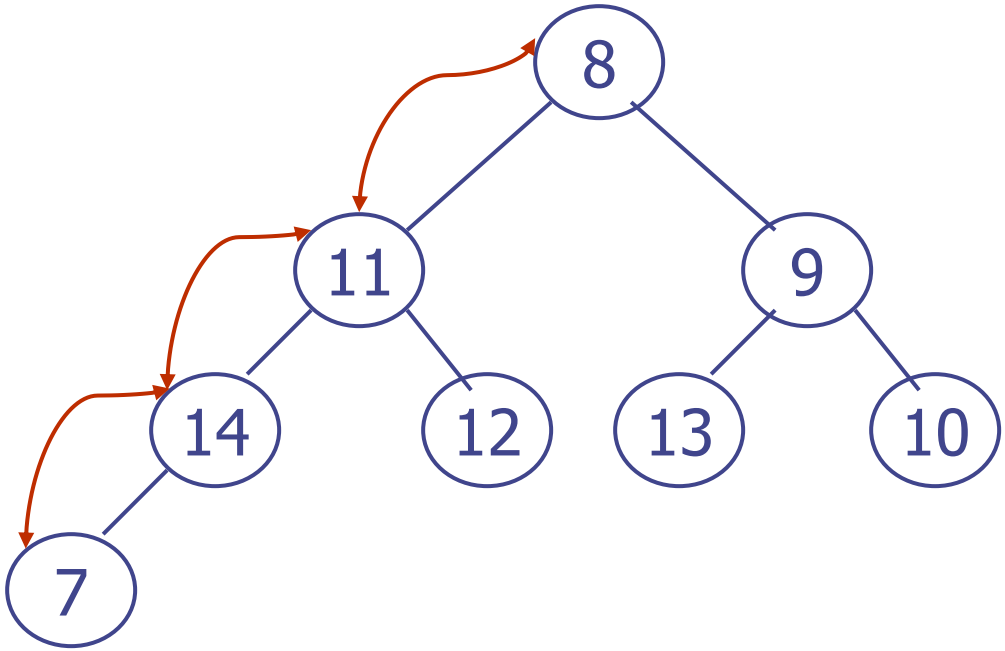
Exercise: insert 8



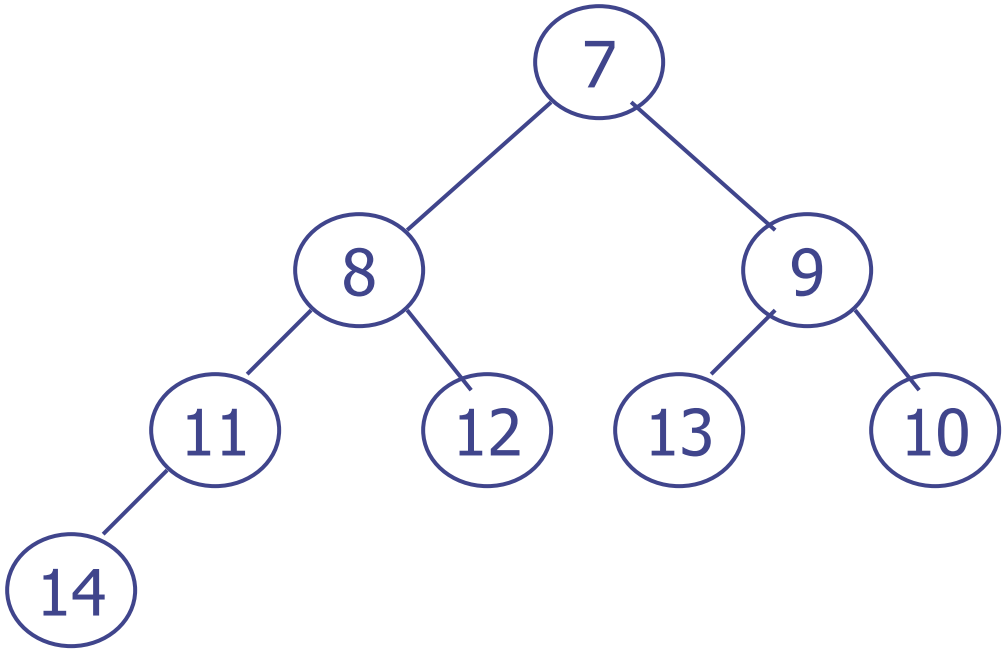
Exercise: insert 8



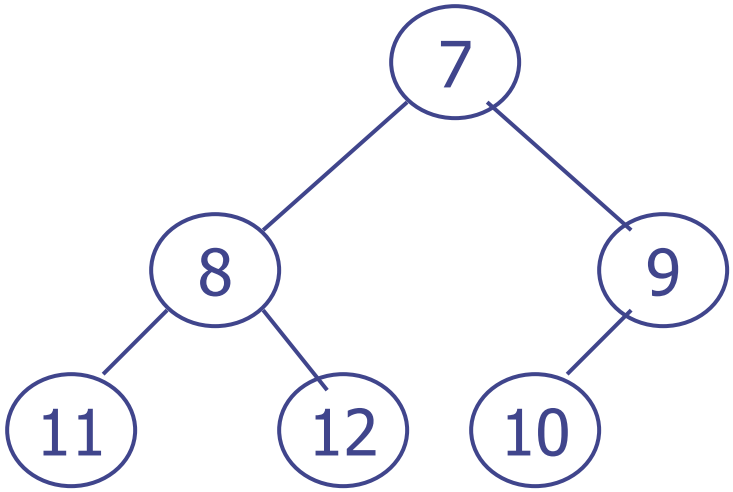
Exercise: insert 7



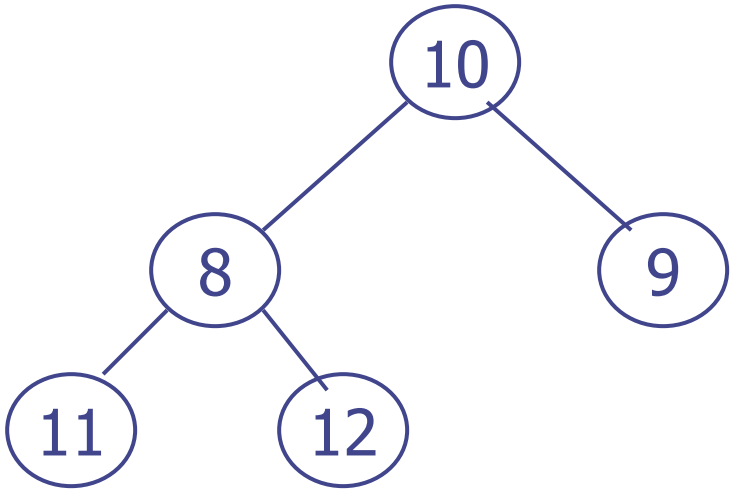
Exercise: insert 7



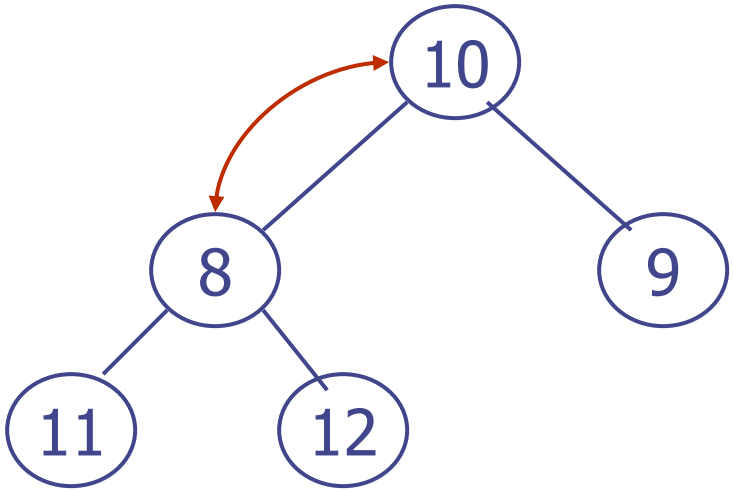
Exercise: remove 7



Exercise: remove 7



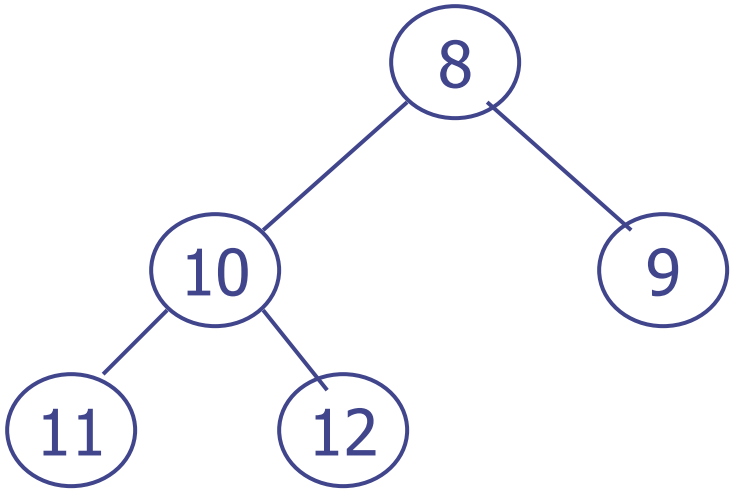
Exercise: remove 7



Always swap with the smallest child!

Exercises (offline):
Why is this needed?
Why is it always correct?
E.g. Try some other rule, such as
"swap with the left child if it is smaller" (BAD RULE)
and find an example where the bad rule fails.

Exercise: remove 7



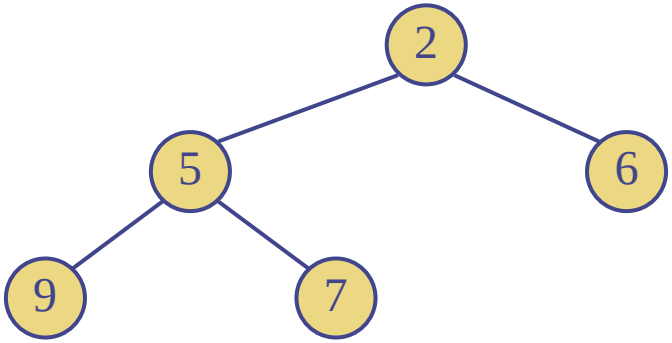
Final result is a valid version of the data structure

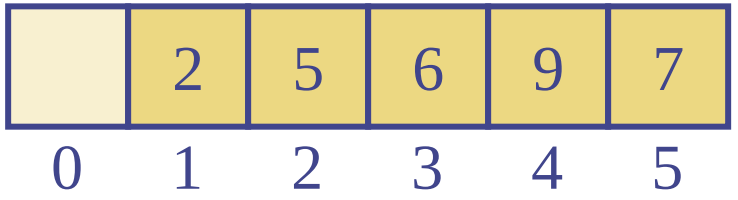
Always check this!!

(In code one could have an assertion to check it, and that is activated in DEBUG mode.)

Array Heap Implementation

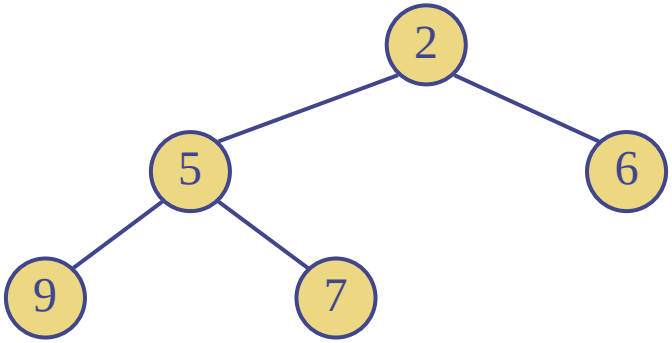
- (See the "trees" lecture on representing binary trees within an array)
- We can represent a heap with n keys by means of an Array (or Vector) of length $n + 1$
- Links between nodes are not explicitly stored, instead:
- For the node at index i
  - the left child is at index $2i$
  - the right child is at index $2i + 1$
  - the parent is at index $i/2$
- The cell at index 0 is not used
  - Makes the equations easier/faster
- Notice that there are no "gaps" when storing a heap

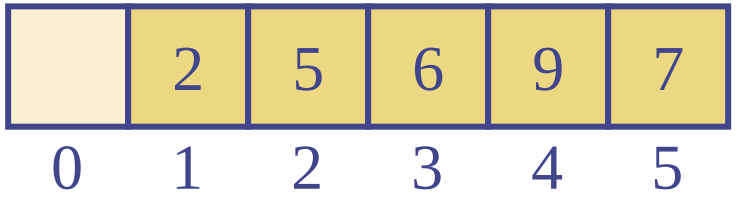




Array Heap Implementation

- For the node at index i
  - the left child is at index $2i$
  - the right child is at index $2i + 1$
  - the parent is at index $i/2$
- Operation insert corresponds to inserting at index $n + 1$
- Operation removeMin corresponds to moving index n to index 1
- Up- and down-heap operations just swap appropriate elements within the array
- Together with the lack of gaps, this makes the implementation very efficient, and this is the standard way to implement a Heap





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.

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 2

The algorithm has two stages:

1. Insert all elements into the heap one by one.
   This takes n insertions with O(log n) each for a total of O(n log n)
2. Remove all elements one by one, using removeMin(), hence obtaining them in sorted order.
   This takes n removals with O(log n) each for a total of O(n log n).

Hence, the overall cost is O(n log n)

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 requires O(n) time.
  - For the heap implementation, all methods run in O(log n).

IMPORTANT

Do NOT confuse

"Binary Tree implementations of heaps"
(this lecture – used for PQ ADT)

with

"Binary Search Trees"
(other lectures – used for Map ADT)

Minimal Expectations

- Thoroughly know the definitions
  - Be able to reason about them and explain their motivations, not only be able to reproduce them
- Know the Binary tree implementation
  - **Do NOT confuse "Binary Tree implementations of heaps" with "Binary Search Trees" (later lectures)**
- Know
  - The insertion/deletion procedures
  - Array-based implementations