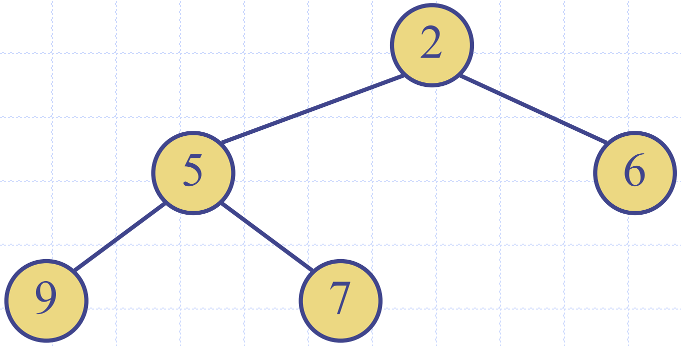
Heaps



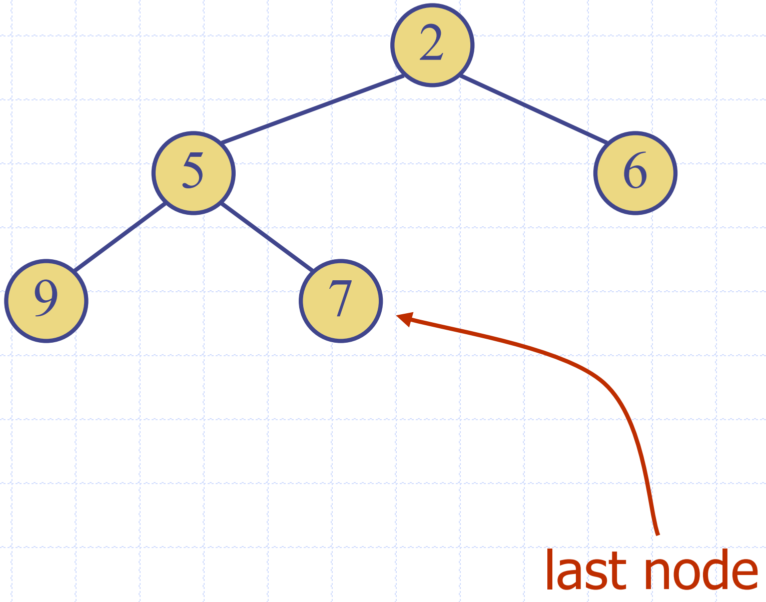
Recall Priority Queue ADT

- A priority queue stores a collection of items
- Each item is a pair (key, value)
- Main methods of the Priority Queue ADT
  - add(k, x) inserts an item with key k and value x
  - remove_min() removes and returns the item with smallest key
- Additional methods
  - min() returns, but does not remove, an item with smallest key
  - len(), is_empty()
- Applications:
  - Standby flyers
  - Auctions
  - Stock market

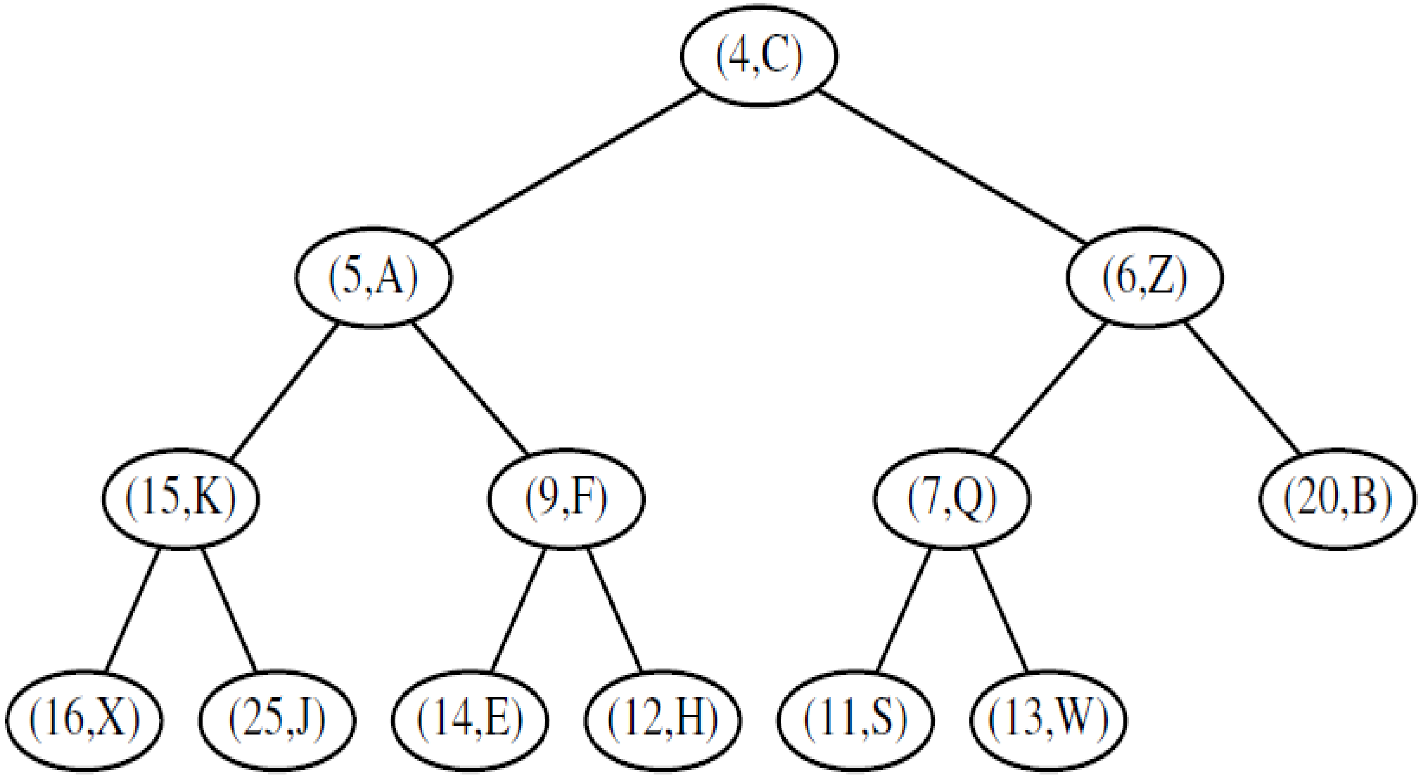
Heaps (Min-Heaps)

- A 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, $key(v) \geq key(parent(v))$
- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \ldots, h-1$, there are 2^i nodes at level i
  - at level h, the nodes reside in the leftmost possible positions

- The last node of a heap is the rightmost node in the last level



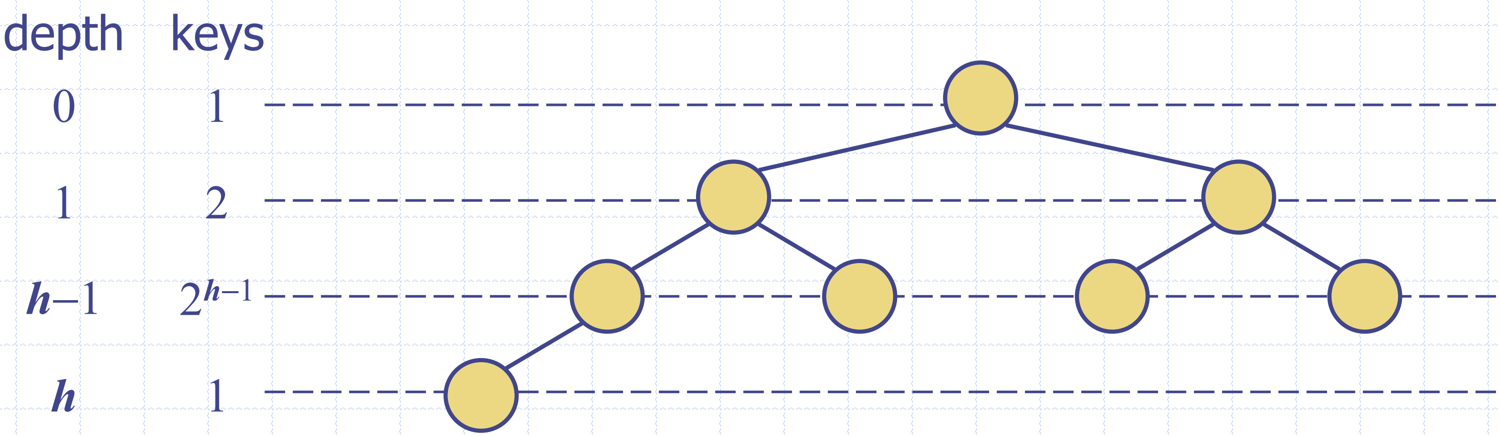
Example of a heap storing Key-Value pairs



Height of a Heap

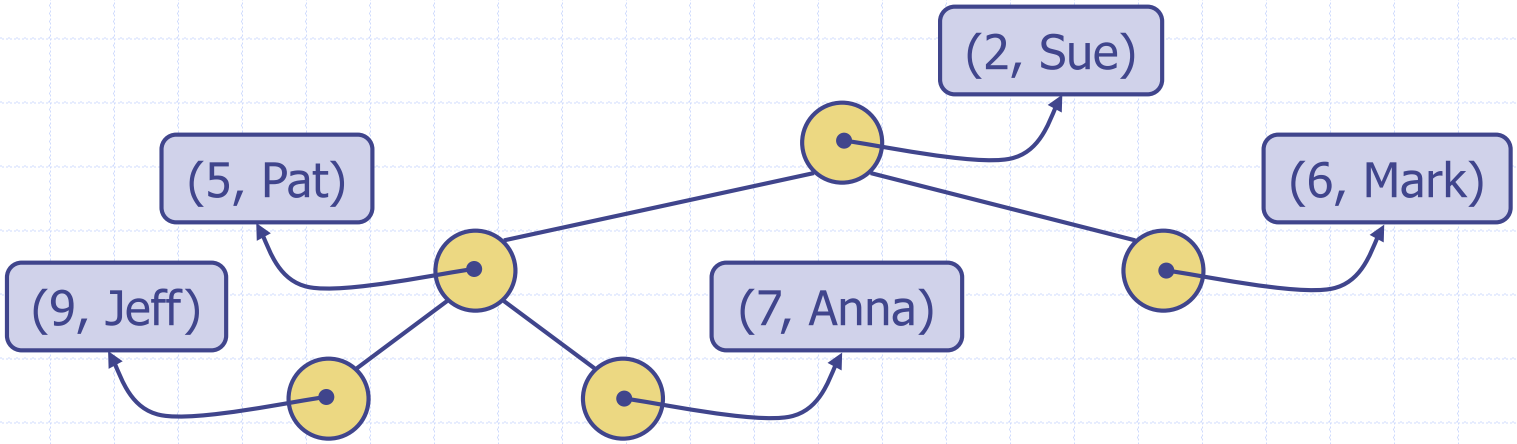
- **Theorem:** A heap storing n keys has height $O(\log n)$

  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., $h \leq \log n$



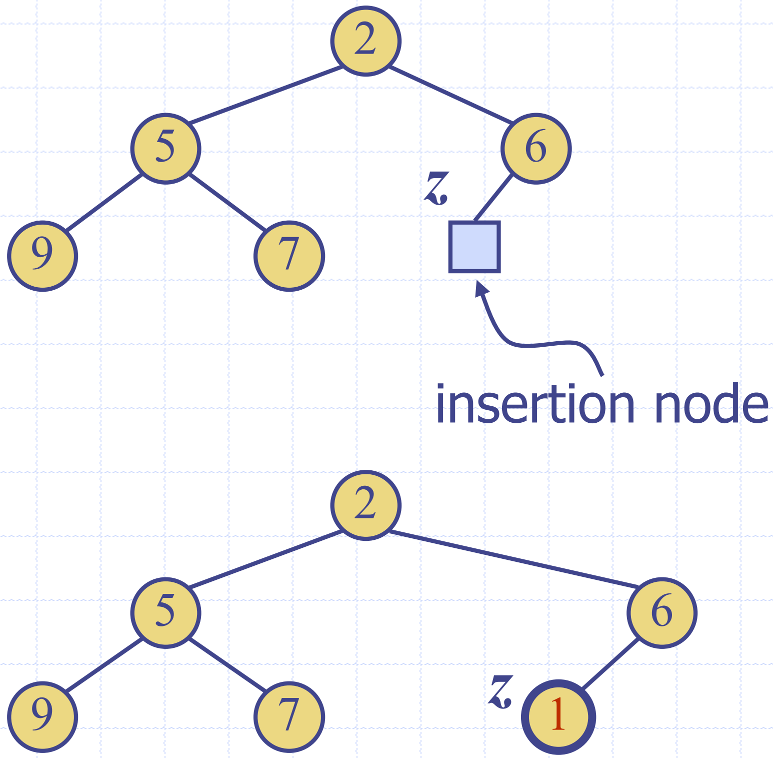
Heaps and Priority Queues

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



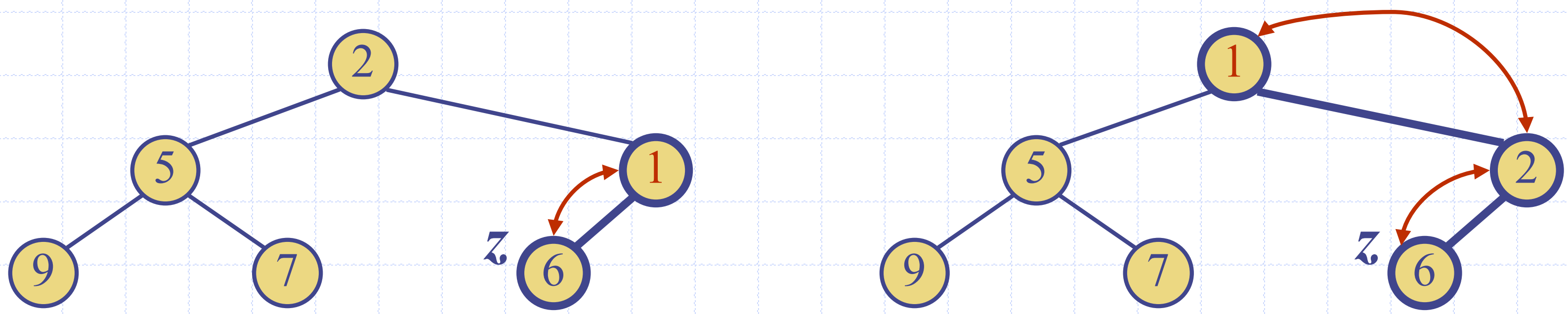
Insertion into a Heap

- Method add 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)

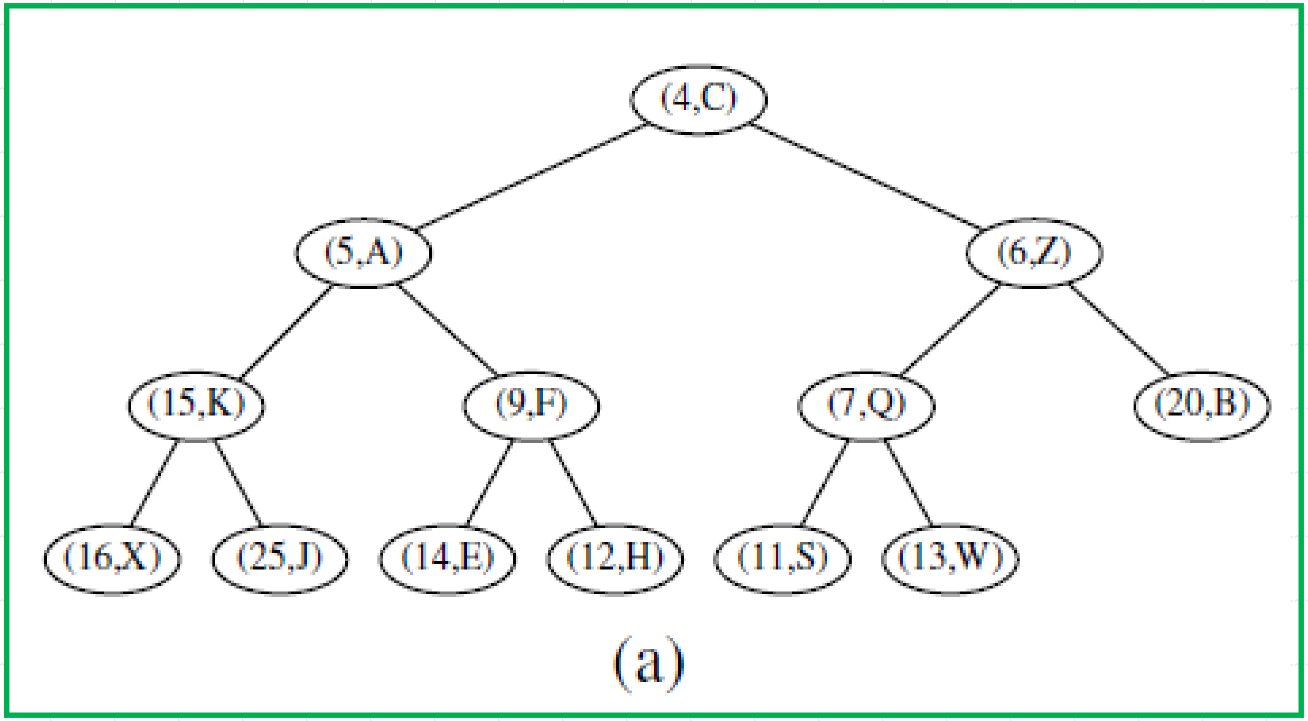


Upheap

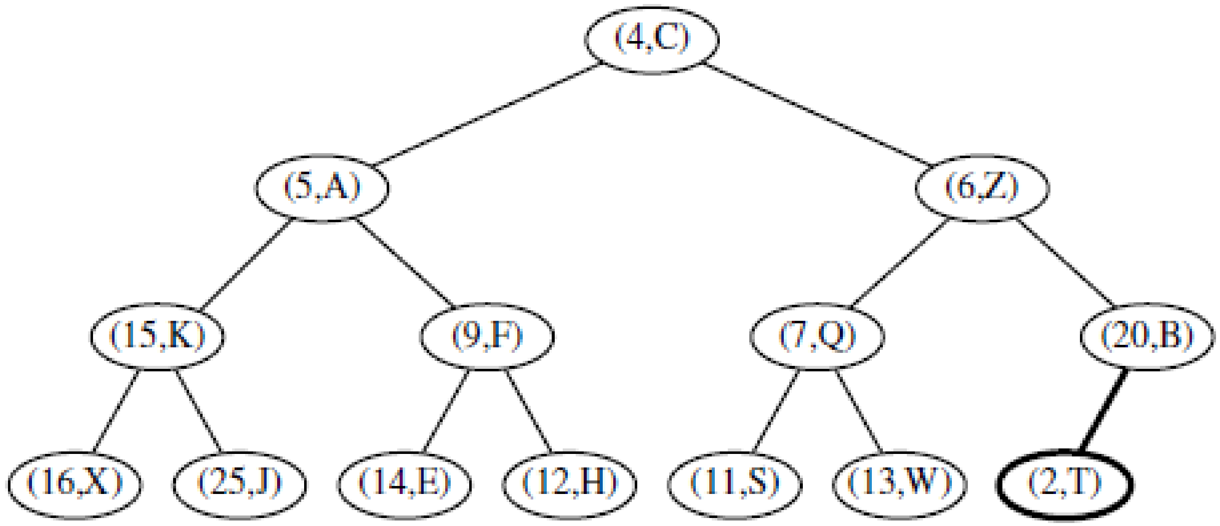
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k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



Let’s Insert (2,T)

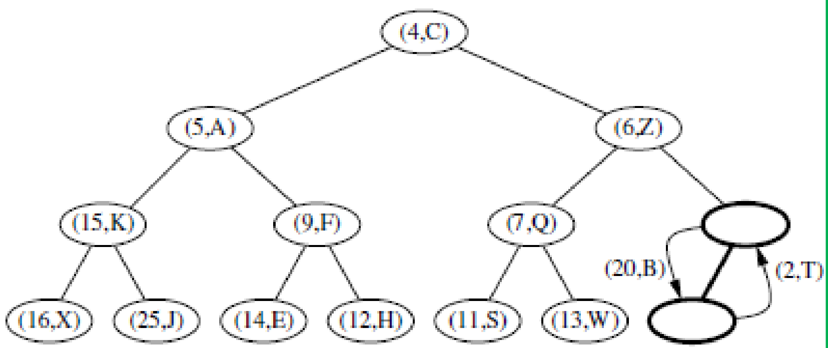


Let's Insert (2,T)

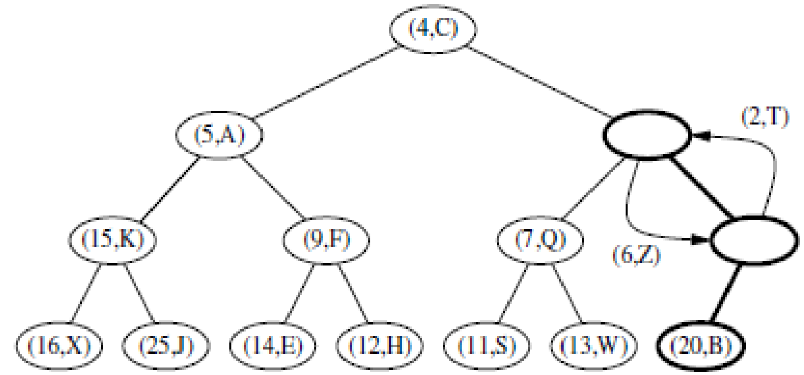


(b)

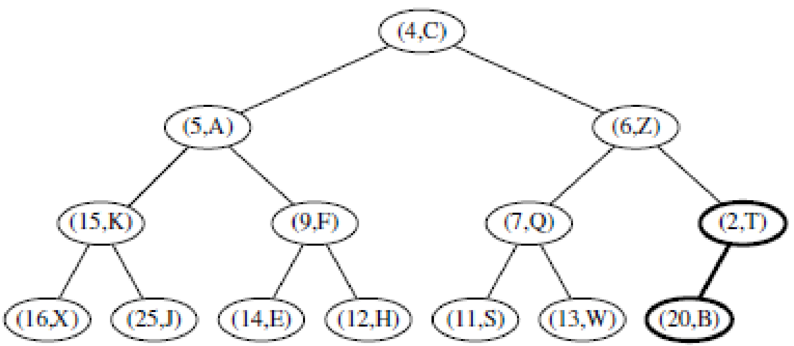
Let’s Insert (2,T)



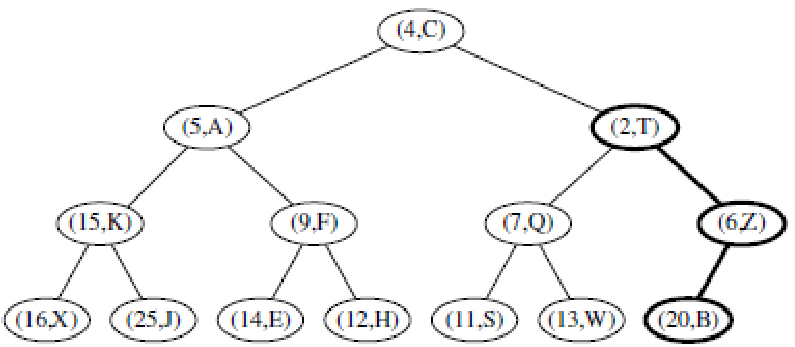
(c)



(e)

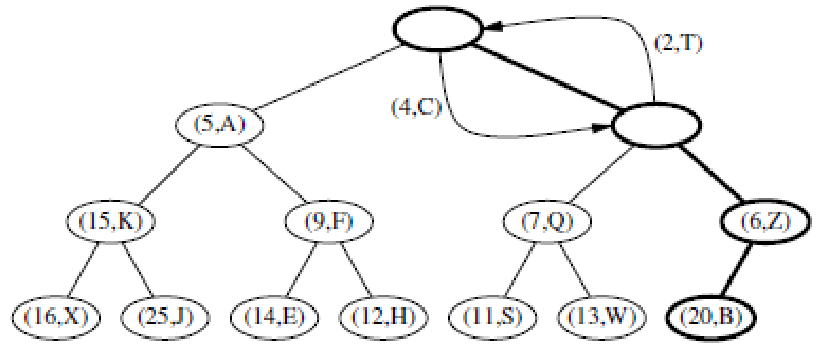


(d)

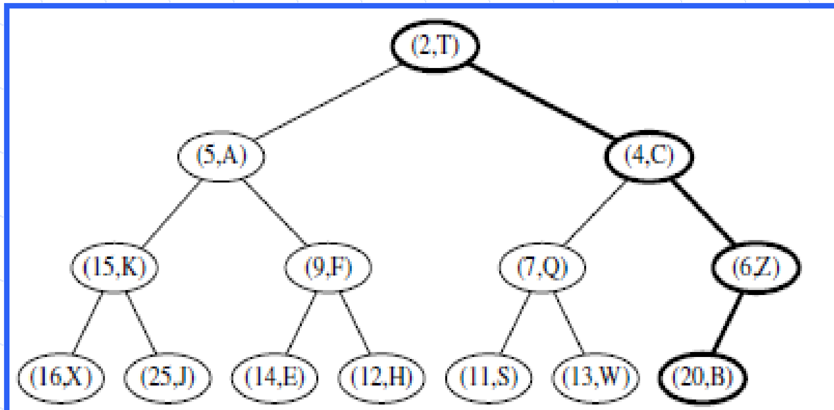


(f)

Let's Insert (2,T)



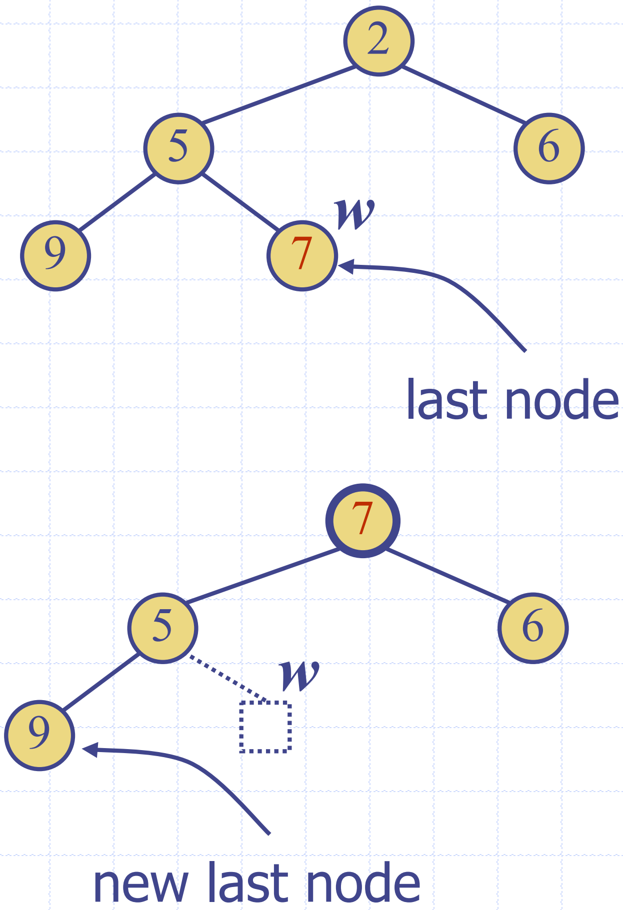
(g)



(h)

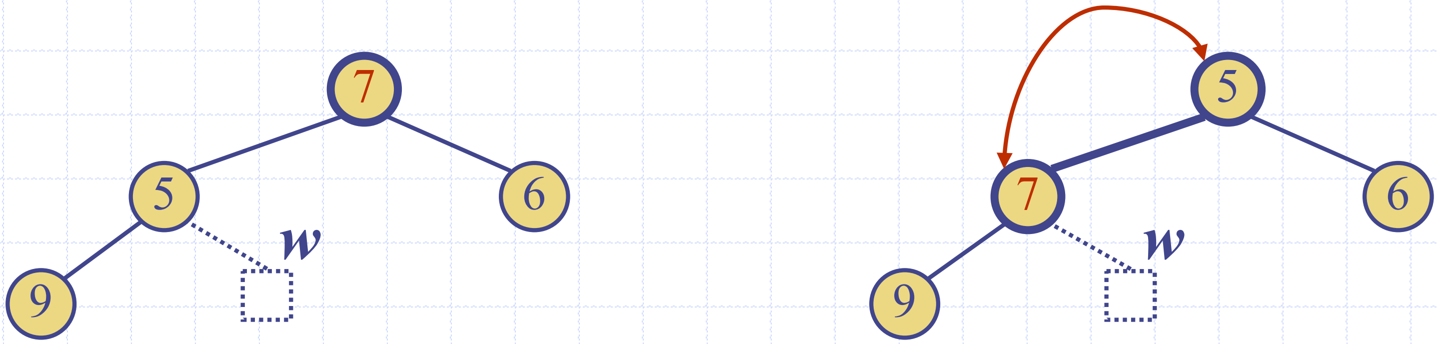
Removal from a Heap

- Method remove_min 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)

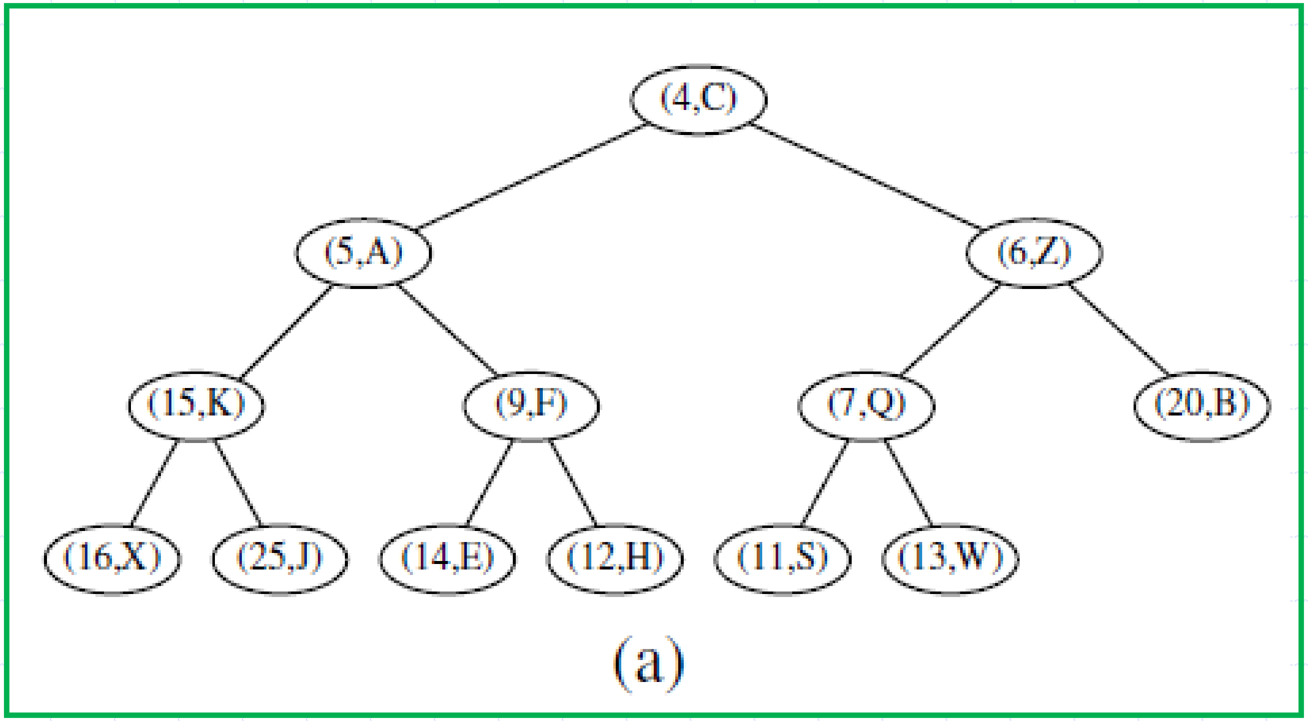


Downheap

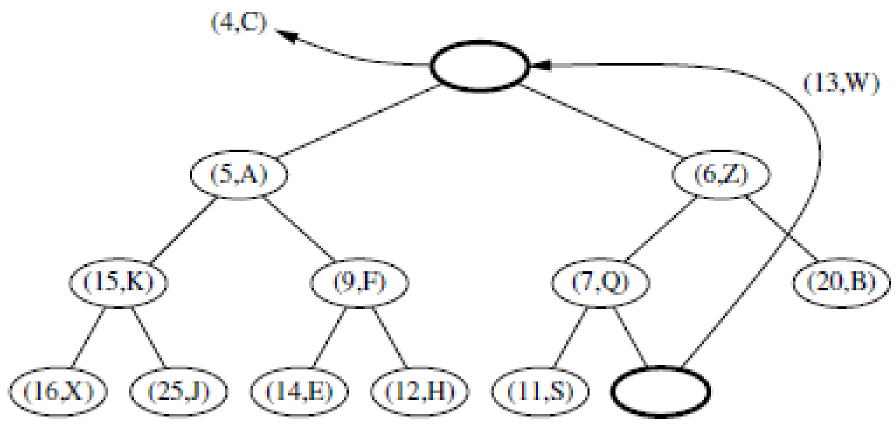
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater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



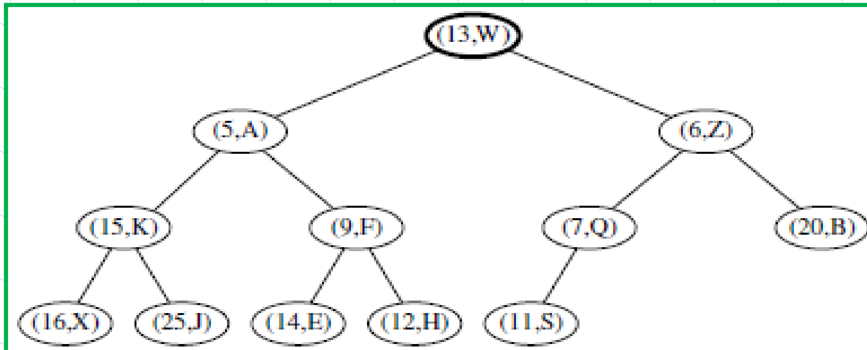
Let's Remove_min (4,C)



Let's Remove_min (4,C)

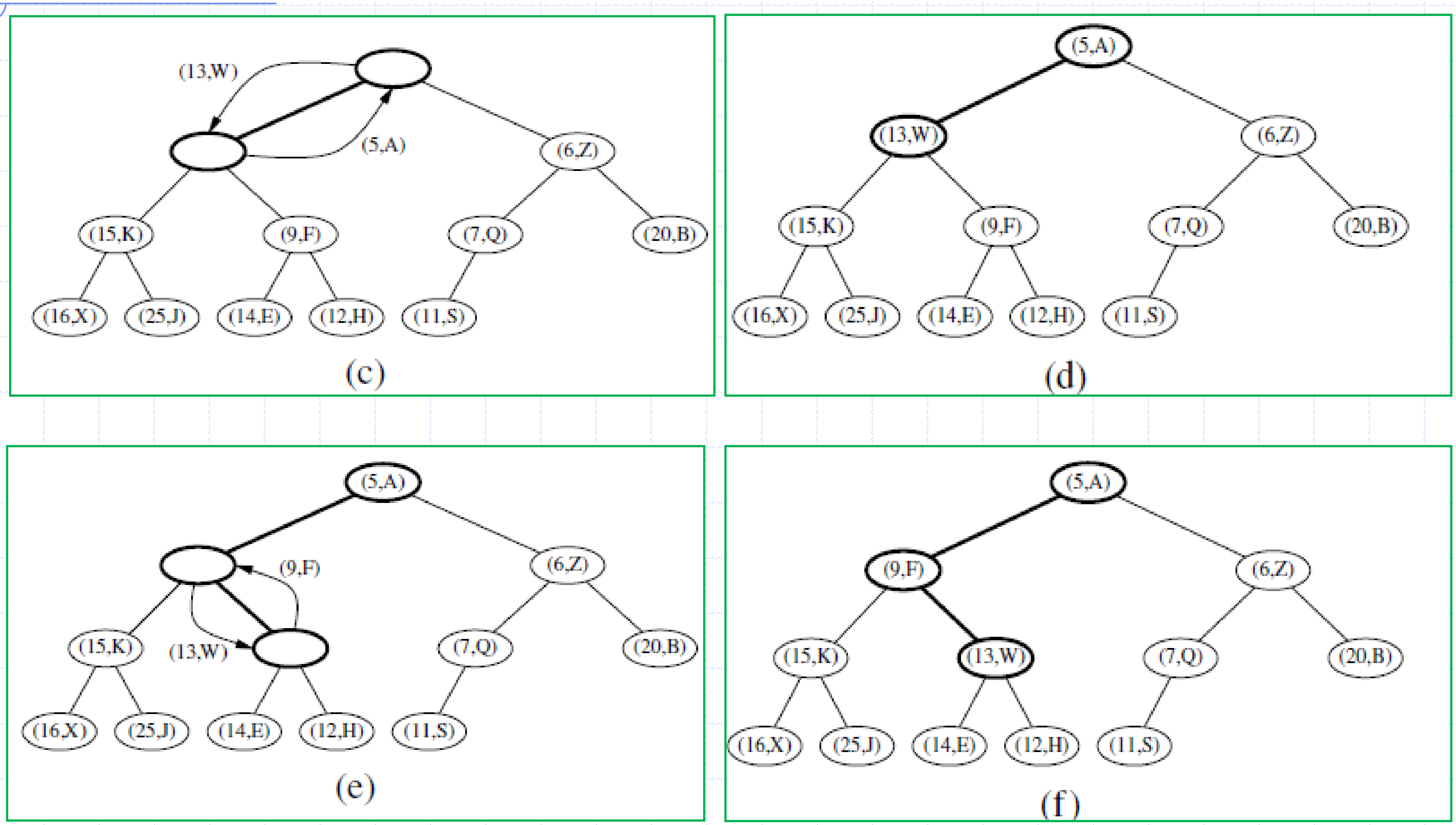


(a)

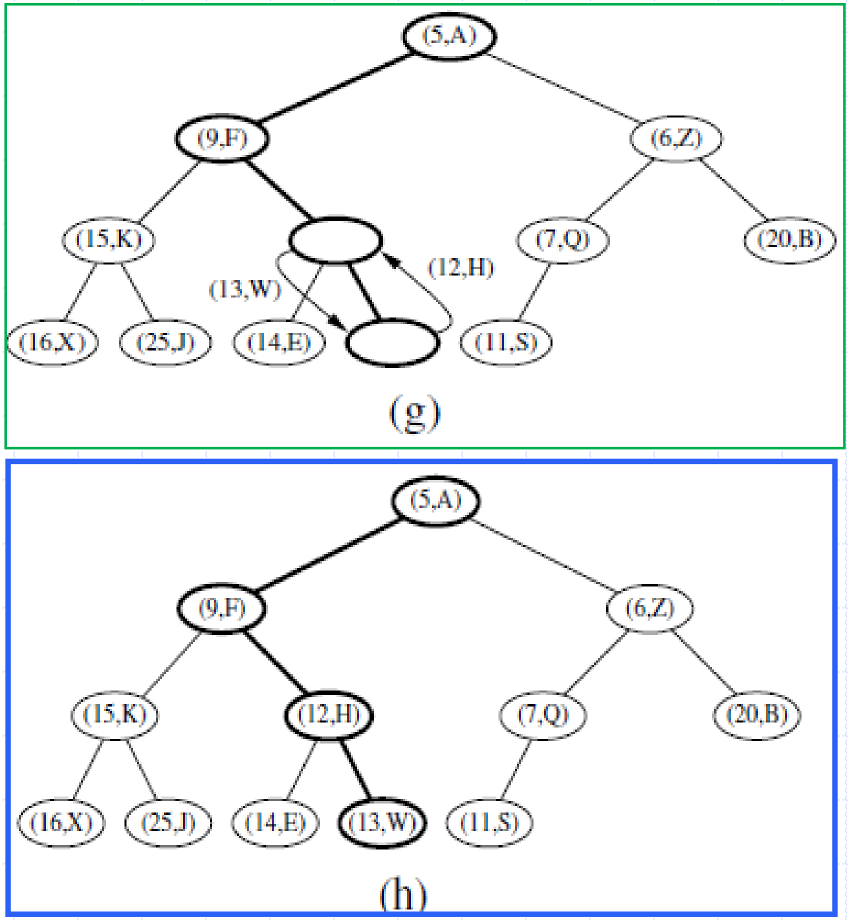


(b)

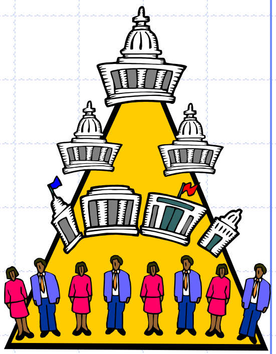
Let’s Remove_min (4,C)



Let's Remove_min (4,C)



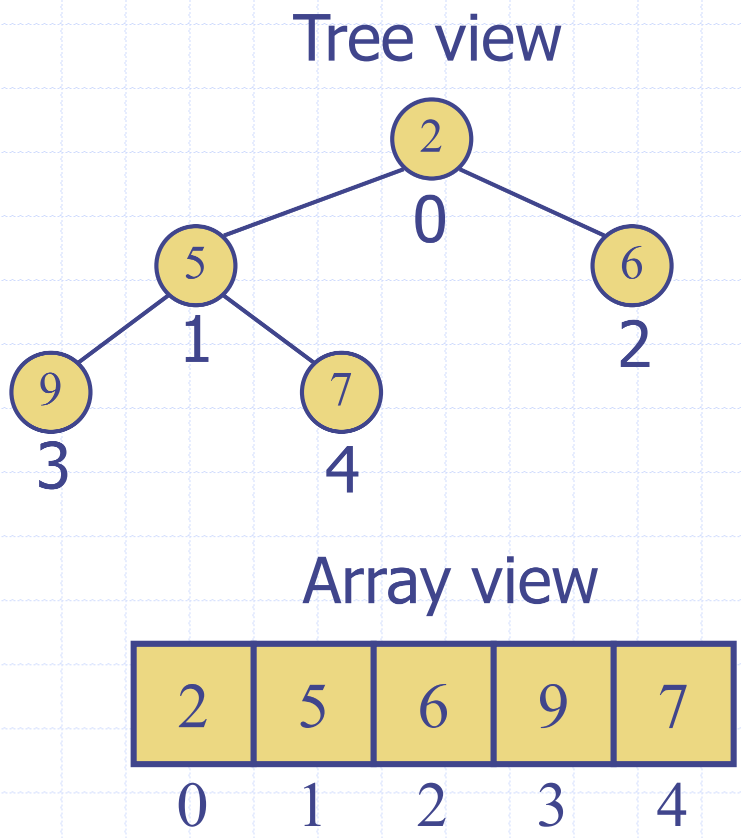
Heap-Sort



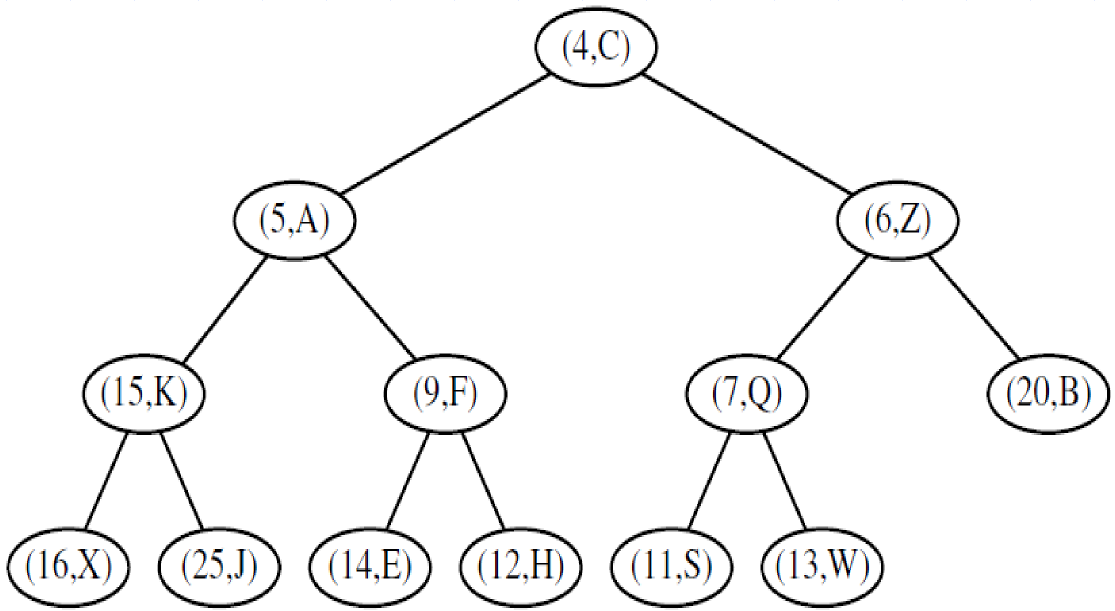
- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods add and remove_min take $O(\log n)$ time
  - methods len, is_empty, and min take time $O(1)$ time
- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$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

Array-based Heap Implementation

- We can represent a heap with n keys by means of an array of length n
- For the node at rank i
  - the left child is at rank $2i + 1$
  - the right child is at rank $2i + 2$
- Links between nodes are not explicitly stored
- Operation add corresponds to inserting at rank n
- Operation remove_min corresponds to removing at rank $n-1$
- Yields in-place heap-sort



Array based Implementation of a heap



(4,c)	(5,A)	(6,Z)	(15,K)	(9,F)	(7,Q)	(20,B)	(16,X)	(25,J)	(14,E)	(12,H)	(11,S)	(13,W)
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**

In-class exercise

- Download heap_priority_queue.py from Brightspace
- Implement basic functions in heap_priority_queue.py (see TODO)
- Submit your code to Gradescope

Python Heap Implementation

```
class HeapPriorityQueue(PriorityQueueBase): # base class defines _Item
  """A min-oriented priority queue implemented with a binary heap."""
  #------------------------------ nonpublic behaviors ------------------------------
  def _parent(self, j):
    return (j-1) // 2

  def _left(self, j):
    return 2*j + 1

  def _right(self, j):
    return 2*j + 2

  def _has_left(self, j):
    return self._left(j) < len(self._data)       # index beyond end of list?

  def _has_right(self, j):
    return self._right(j) < len(self._data)      # index beyond end of list?

  def _swap(self, i, j):
    """Swap the elements at indices i and j of array."""
    self._data[i], self._data[j] = self._data[j], self._data[i]

  def _upheap(self, j):
    parent = self._parent(j)
    if j > 0 and self._data[j] < self._data[parent]:
      self._swap(j, parent)
      self._upheap(parent)                       # recur at position of parent

  def _downheap(self, j):
    if self._has_left(j):
      left = self._left(j)
      small_child = left                         # although right may be smaller
      if self._has_right(j):
        right = self._right(j)
        if self._data[right] < self._data[left]:
          small_child = right
      if self._data[small_child] < self._data[j]:
        self._swap(j, small_child)
        self._downheap(small_child)              # recur at position of small child

  #------------------------------ public behaviors ------------------------------
  def __init__(self):
    """Create a new empty Priority Queue."""
    self._data = [ ]

  def __len__(self):
    """Return the number of items in the priority queue."""
    return len(self._data)

  def add(self, key, value):
    """Add a key-value pair to the priority queue."""
    self._data.append(self._Item(key, value))
    self._upheap(len(self._data) - 1)            # upheap newly added position

  def min(self):
    """Return but do not remove (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    item = self._data[0]
    return (item._key, item._value)

  def remove_min(self):
    """Remove and return (k,v) tuple with minimum key.

    Raise Empty exception if empty.
    """
    if self.is_empty():
      raise Empty('Priority queue is empty.')
    self._swap(0, len(self._data) - 1)           # put minimum item at the end
    item = self._data.pop( )                     # and remove it from the list;
    self._downheap(0)                            # then fix new root
    return (item._key, item._value)
```

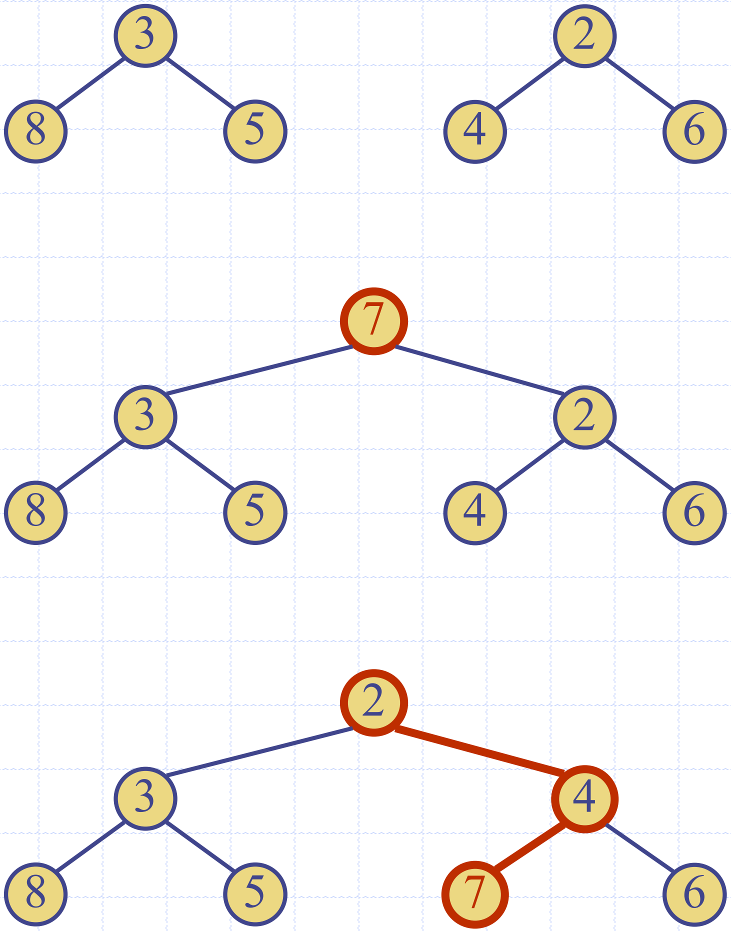
Analysis of a Heap-Based Priority Queue

Operation	Running Time
len(P), P.is_empty()	$O(1)$
P.min()	$O(1)$
P.add()	$O(\log n)^*$
P.remove_min()	$O(\log n)^*$

*amortized, if array-based

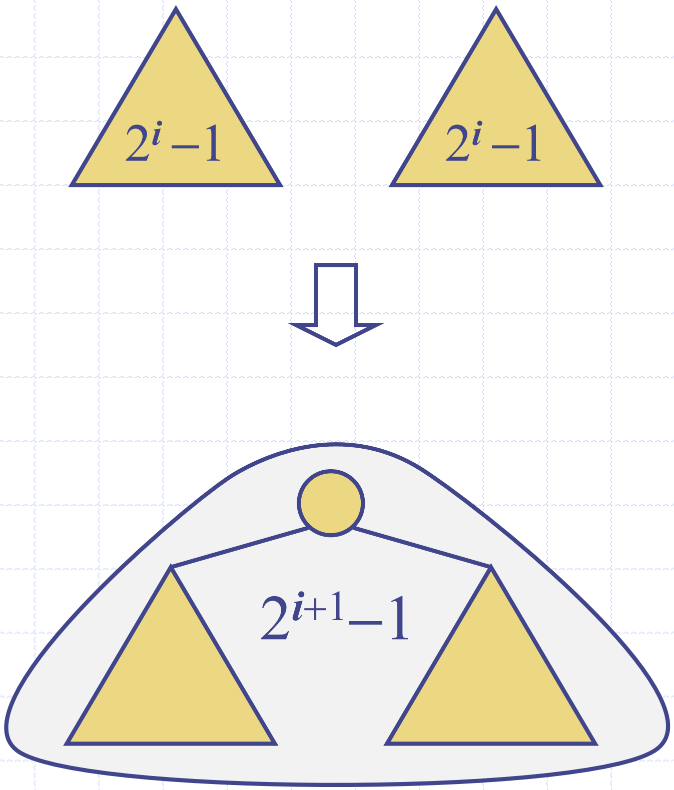
Merging Two Full Heaps

- We are given two full heaps and a key k
- We create a new heap with the root node storing k and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



Bottom-up Heap Construction

- We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with 2^i-1 keys are merged into heaps with $2^{i+1}-1$ keys

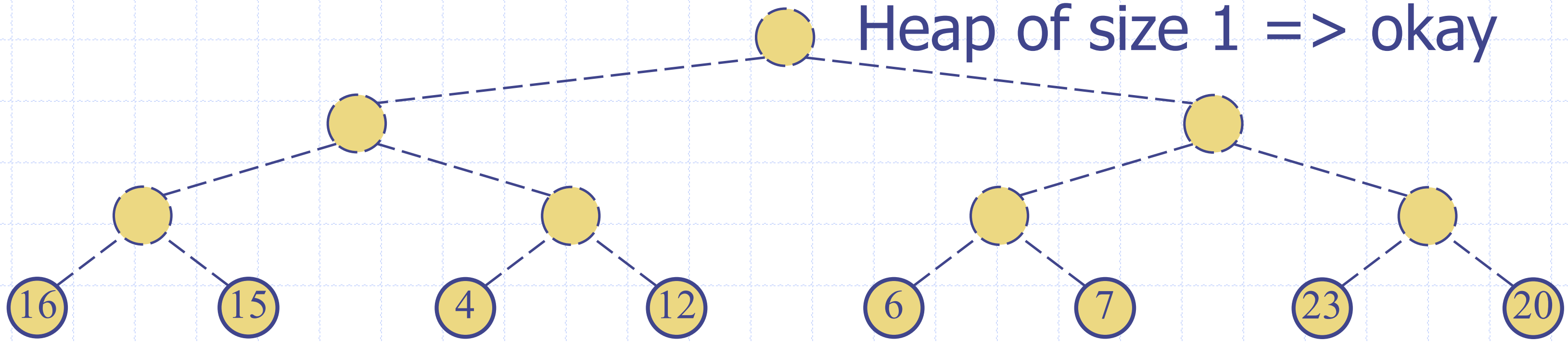


Heapify

- A complete tree can be represented using an array
- Given an array [10, 7, 8, 25, 5, 11, 27, 16, 15, 4, 12, 6, 7, 23, 20], how to convert it into a heap?
- Useful formulae:
  - Last node index: N-1
  - parent(i) = (i-1) // 2
- Bottom-up processing of internal nodes

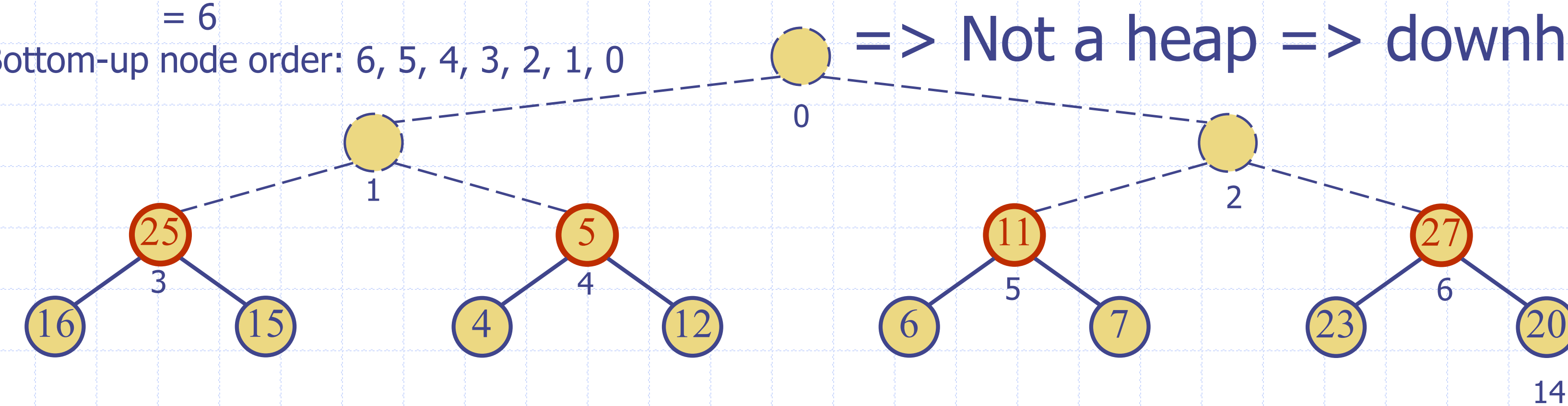
Example

Start from the bottom layer
Heap of size 1 => okay



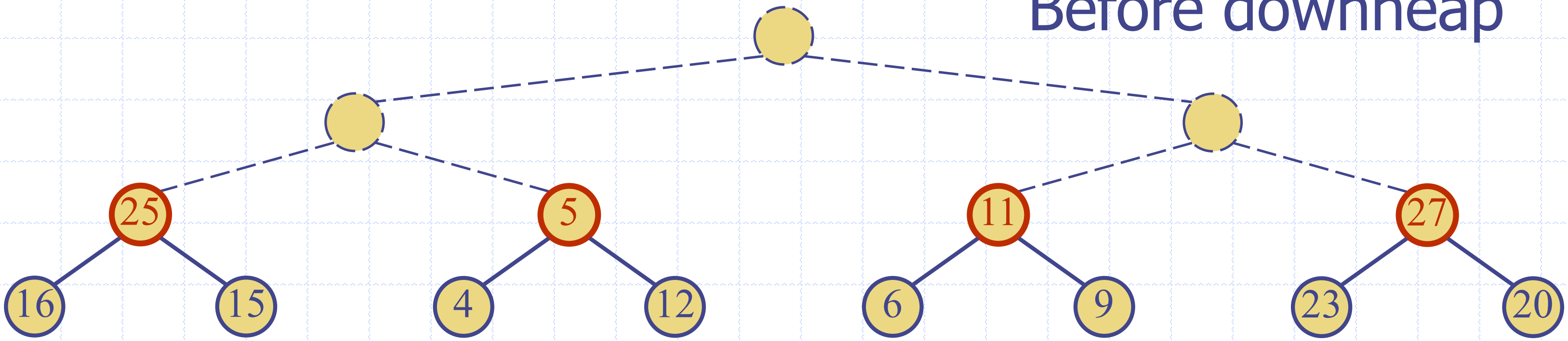
Locate the last internal node to start:
parent(14) = (14-1) // 2
= 6
Bottom-up node order: 6, 5, 4, 3, 2, 1, 0

Add bottom-1 layer
=> Not a heap => downheap

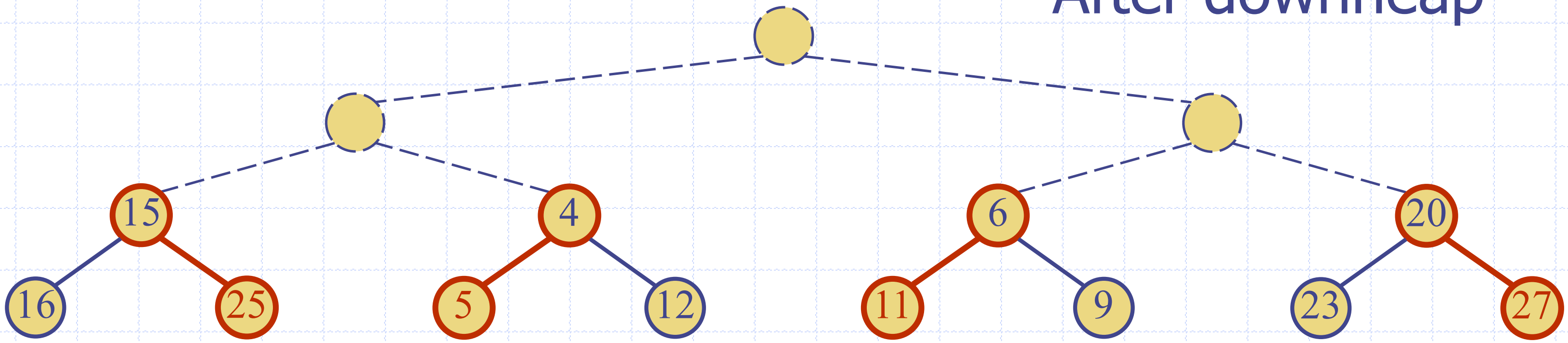


Example (contd.)

Before downheap

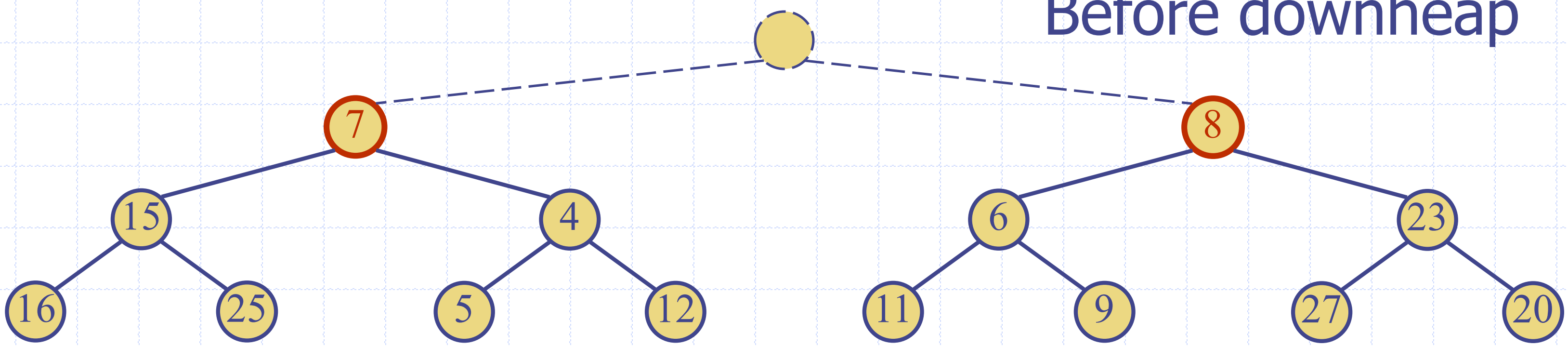


After downheap

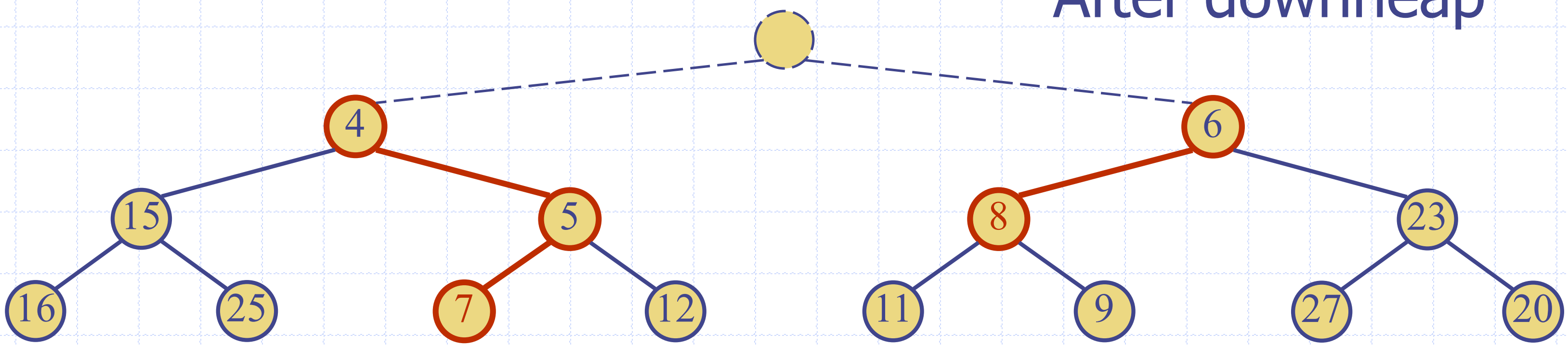


Example (contd.)

Before downheap

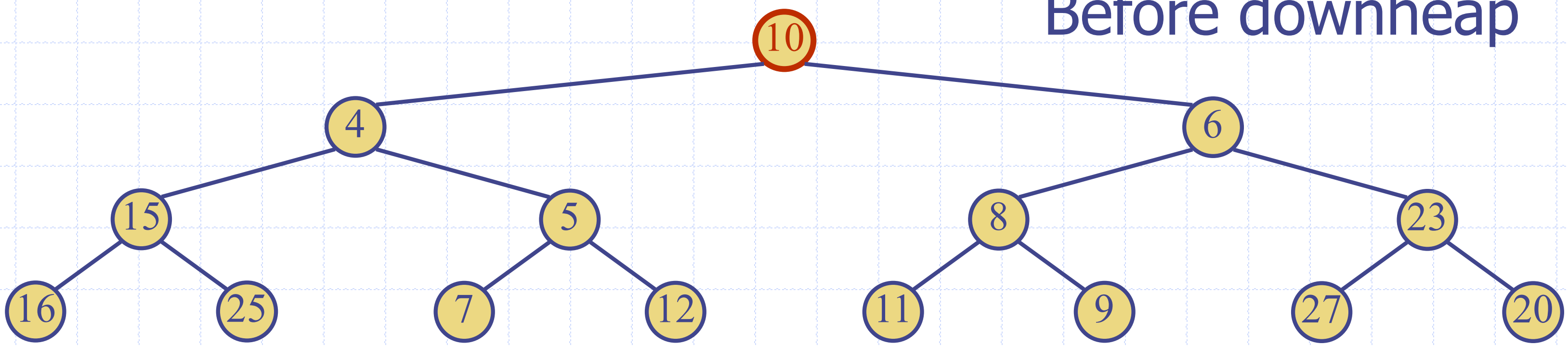


After downheap

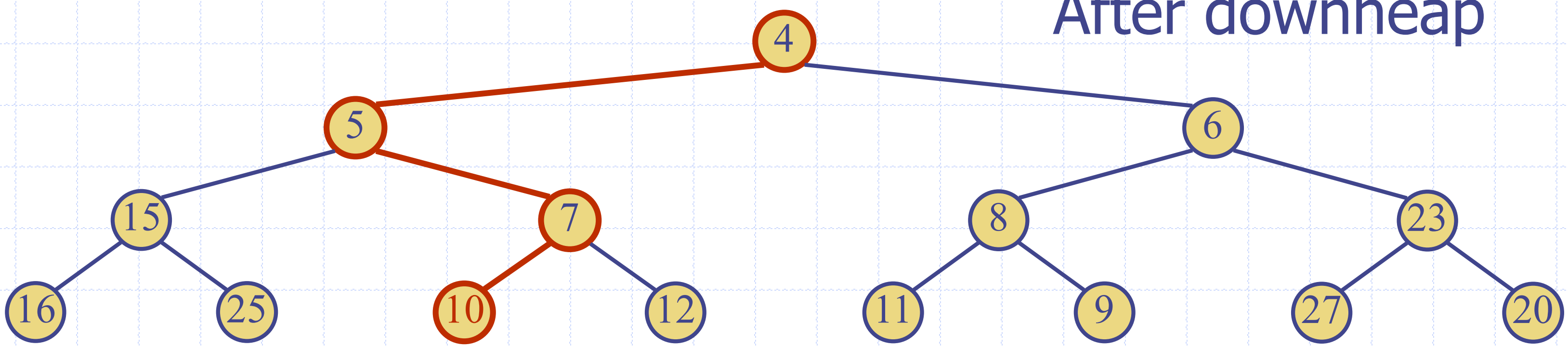


Example (end)

Before downheap



After downheap



Analysis of Heap Construction

- We visualize the worst-case time of a downheap with a proxy path that goes first right and then repeatedly goes left until the bottom of the heap (this path may differ from the actual downheap path)
- Since each node is traversed by at most two proxy paths, the total number of nodes of the proxy paths is $O(n)$
- Thus, bottom-up heap construction runs in $O(n)$ time
- Bottom-up heap construction is faster than n successive insertions and speeds up the first phase of heap-sort