MECHTRON 2MD3

Data Structures and Algorithms for Mechatronics

Winter 2022

27 Priority Queues and Heaps

Department of Computing and Software

Instructor:

Omid Isfahanialamdari

March 28, 2022

Admin

- Mid-Term 2:
  - Wednesday 30 March 2022
  - Duration: **1 hour**
  - **From 1:30 to 14:30 (lec. time)**
  - Location: **T13 123**
  - Covers: Topics from “Doubly Linked Lists” until the lecture of Wednesday 16 March 2022 (inclusive)

Priority Queue

What we have seen so far

- So far, we have seen “position-based” data structures
  - Stacks, queues, deques, lists, trees
  - Store elements at specific positions (linear or hierarchical)
  - Insertion and removal based on “position” (linear or hierarchical)
  - But, priority queue
    - Insertion and removal: **priority-based**

- Priority Queue
  - Data structure for storing a collection of prioritized elements
  - Supporting arbitrary element insertion
  - Supporting removal of elements in order of priority
    - How to express priority? with key

Priority Queue ADT

- A priority queue stores a collection of entries
- Typically, an entry is a pair (key, value), where the key indicates the priority
- Main methods of the Priority Queue ADT
  - insert(e): inserts an entry e
  - removeMin(): removes the entry with smallest key
- Additional methods
  - min()
    - returns, but does not remove, an entry with smallest key
  - size(), empty()
    - Applications:
      - Auctions
      - Stock market

Total Order Relations

- Keys in a priority queue can be arbitrary objects on which an order is defined
- Two distinct entries in a priority queue can have the same key
- Total ordering
  - Comparison rule should be defined for every pair of keys
- Satisfying the above three properties ensures:
  - We will never have a comparison contradiction

- Mathematical concept of total order relation $\leq$
  - Reflexive property: $x \leq x$
  - Antisymmetric property: $x \leq y \wedge y \leq x \Rightarrow x = y$
  - Transitive property: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Total Order Example

- 2D points with (x-coordinate, y-coordinate)
  - Define relation ‘>=’ based on first x, and then y
    - (4,3) >= (3,4),
    - (3,5) >= (3,4)
    - Total ordering
- What about defining relation ‘>=‘ based on both x and y
  - (4,3) >=(2,1), but (4,3) ??? (3,4)
  - Partial ordering
    - Comparison is not defined for some objects

- We assume that we define a comparison that leads to total ordering.

Comparator Design Pattern

- Integer, float, double
  - Quite clear on how to define “order”

- Student: id, sex, department
  - S1 is less than S2? In what sense?

- Flight Passengers: airplane number, seat number, sex
  - P1 is less than P2? In what sense?

- How to design “comparison logic” in a programming language?

Comparator Design Pattern

- Having different Priority Queues for different Objects?!
  - Simple, but not general
  - Many copies of the same code

- Template and Overloading
  - General enough for many situations
  - Cannot have multiple comparison methods for the same type
    - What about comparison based on first y, and then x?
- Separating Comparator
  - 2D points
    - Sometimes we want either of X-based comparison, Y-based comparison
    - Idea
      - Define a comparator class, e.g., "LeftRight" (x-based) and "BottomTop" (ybased)
      - Overload "()" operator

Comparator ADT

- Implements the boolean function isLess(p,q), which tests whether p < q
- Can derive other relations from this:
  - (p == q) is equivalent to
  - (!isLess(p, q) && !isLess(q, p))
- Can implement in C++ by overloading "()"

Two ways to compare 2D points:

```
class LeftRight { // left-right comparator
public:
    bool operator()(const Point2D& p,
      const Point2D& q) const
    { return p.getX() < q.getX(); }
};
class BottomTop { // bottom-top
public:
    bool operator()(const Point2D& p,
    const Point2D& q) const
    { return p.getY() < q.getY(); }
};
```

- Can use: leftRight(*p*,*q*) or bottomTop(p,q)

Comparator ADT

```
template <typename E, typename C>      // element type and comparator
void printSmaller(const E& p, const E& q, const C& isLess) {
  cout << (isLess(p, q) ? p : q) << endl;  // print the smaller of p and q
}
```

```
Point2D p(1.3, 5.7), q(2.5, 0.6);        // two points
LeftRight leftRight;                     // a left-right comparator
BottomTop bottomTop;                     // a bottom-top comparator
printSmaller(p, q, leftRight);           // outputs: (1.3, 5.7)
printSmaller(p, q, bottomTop);           // outputs: (2.5, 0.6)
```

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. Remove the elements in sorted order with a series of removeMin operations
- The running time of this sorting method depends on the priority queue implementation

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.empty*** ()

  e ← ***S.front***(); ***S.eraseFront***()

  P.insert (***e***, ∅)

while ¬***P.empty***()

  e ← ***P.removeMin***()

  S.insertBack(***e***)

Sequence-based Priority Queue

- Implementation 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

- Implementation 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

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  1. Insert the elements one by one with a series of insert operations
  2. 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(n2) time**
  - **Sorted sequence gives insertion-sort: O(n2) 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.empty*** ()

  e ← ***S.front***(); ***S.eraseFront***()

  P.insert (***e***, ∅)

while ¬***P.empty***()

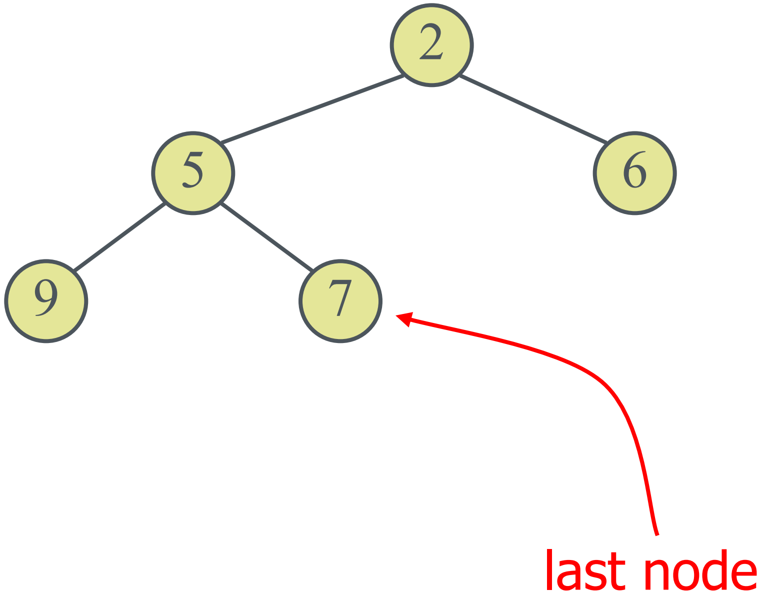
  e ← ***P.removeMin***()

  S.insertBack(***e***)

Heaps

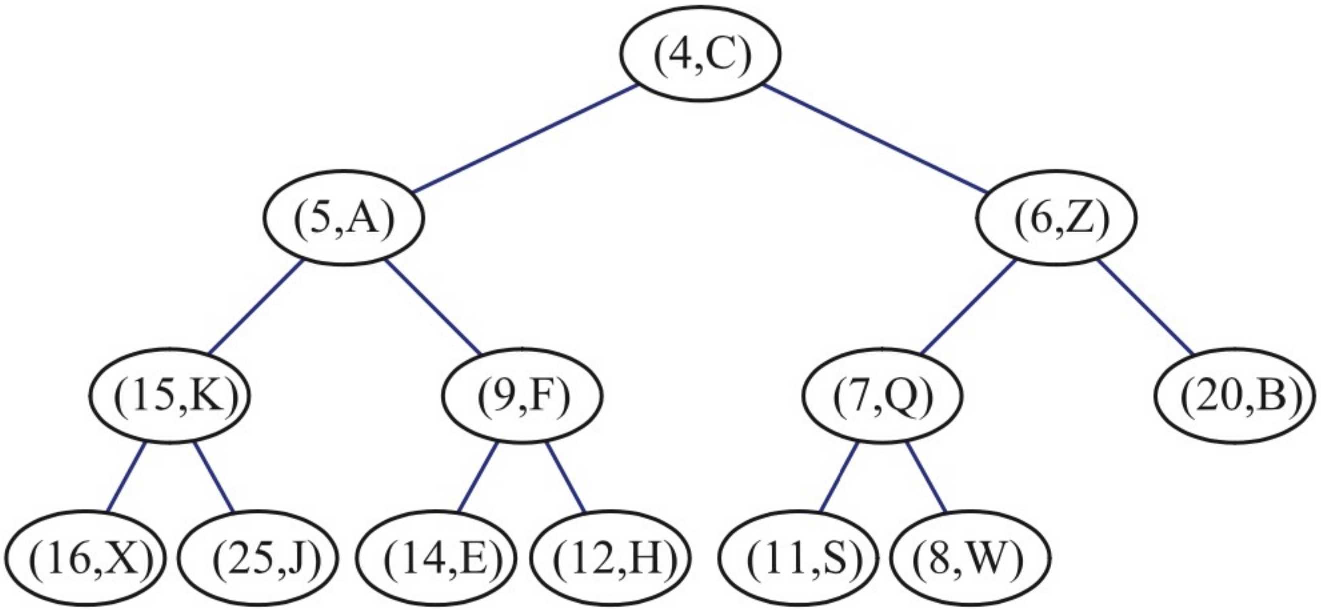
- A heap is a binary tree storing keys at its nodes and satisfying the following properties:
- Heap-Order: for every internal node v other than the root, $key(v) \geq key(parent(v))$
- Complete Binary Tree: let h be the height of the heap
  - for $i = 0, \dots, h - 1$, there are 2^i nodes of depth i
  - at depth $h - 1$, the internal nodes are to the left of the external nodes

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



Heap Order

- The keys encountered on a path from the root to a leaf T are nondecreasing
- A minimum key: always at the root

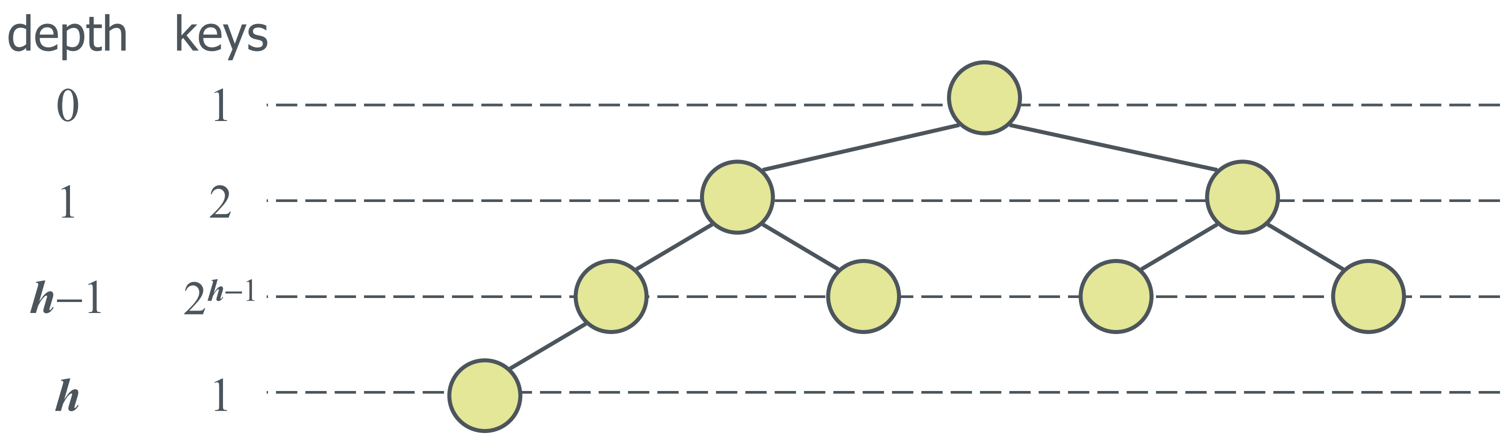


Height of a Heap

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

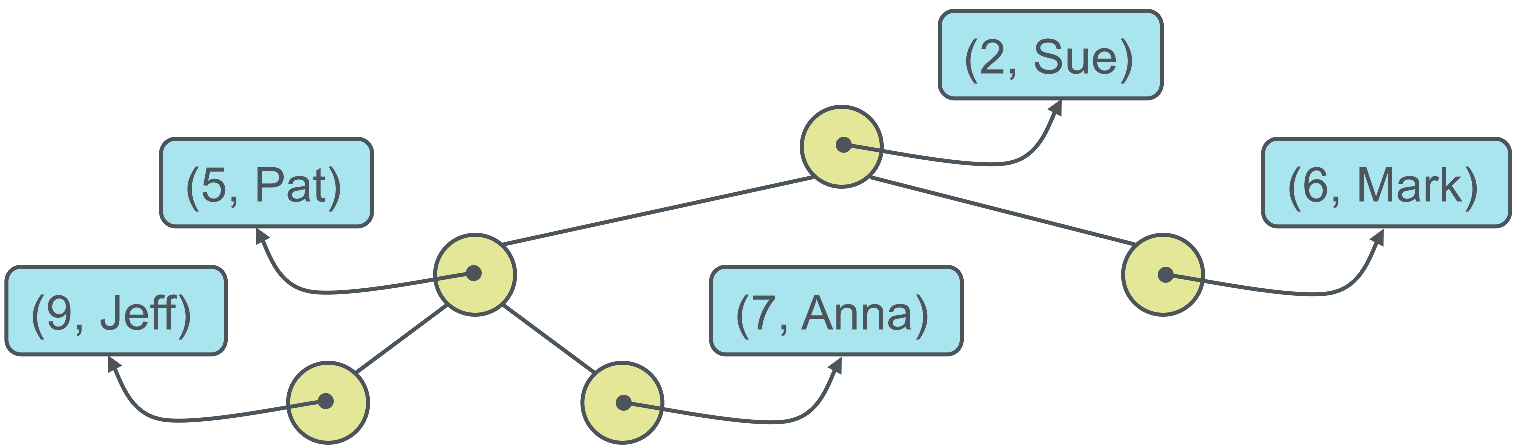
  Proof: (we apply the complete binary tree property)

  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \dots, h-1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$



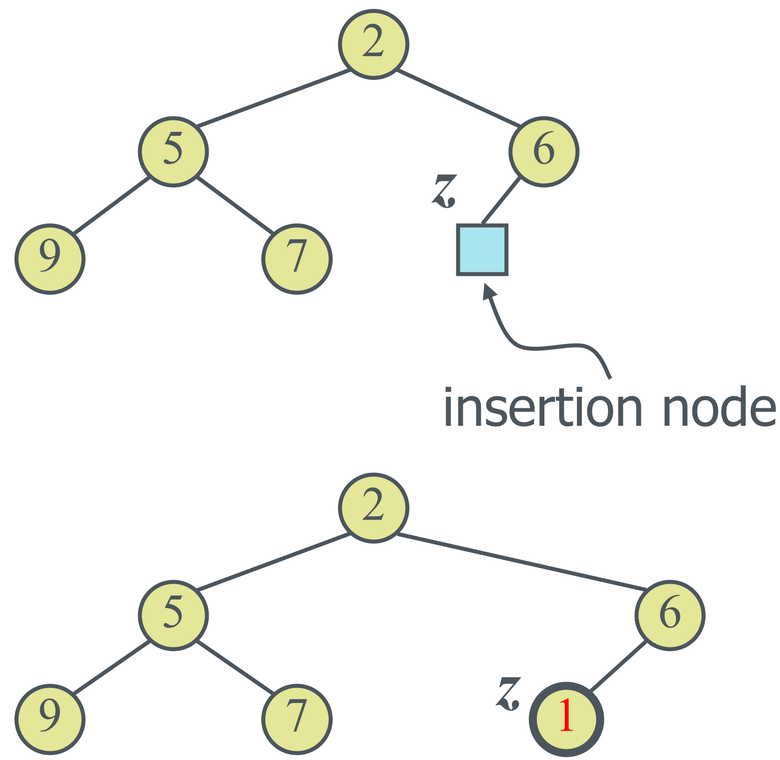
Heaps and Priority Queues

- 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



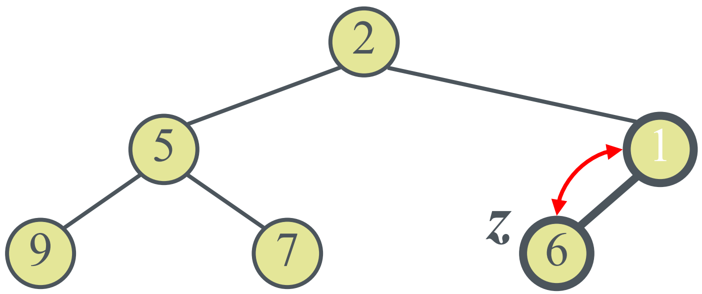
Insertion into a Heap

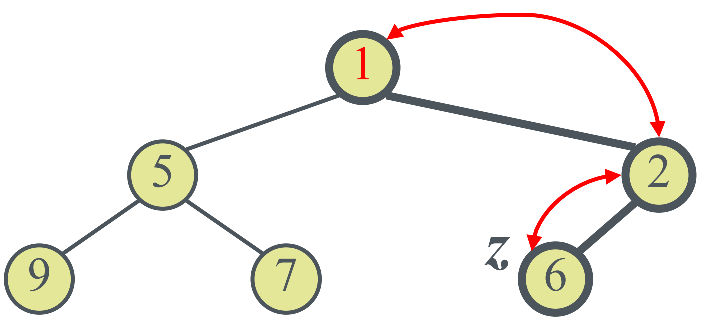
- Method insert of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)



Upheap

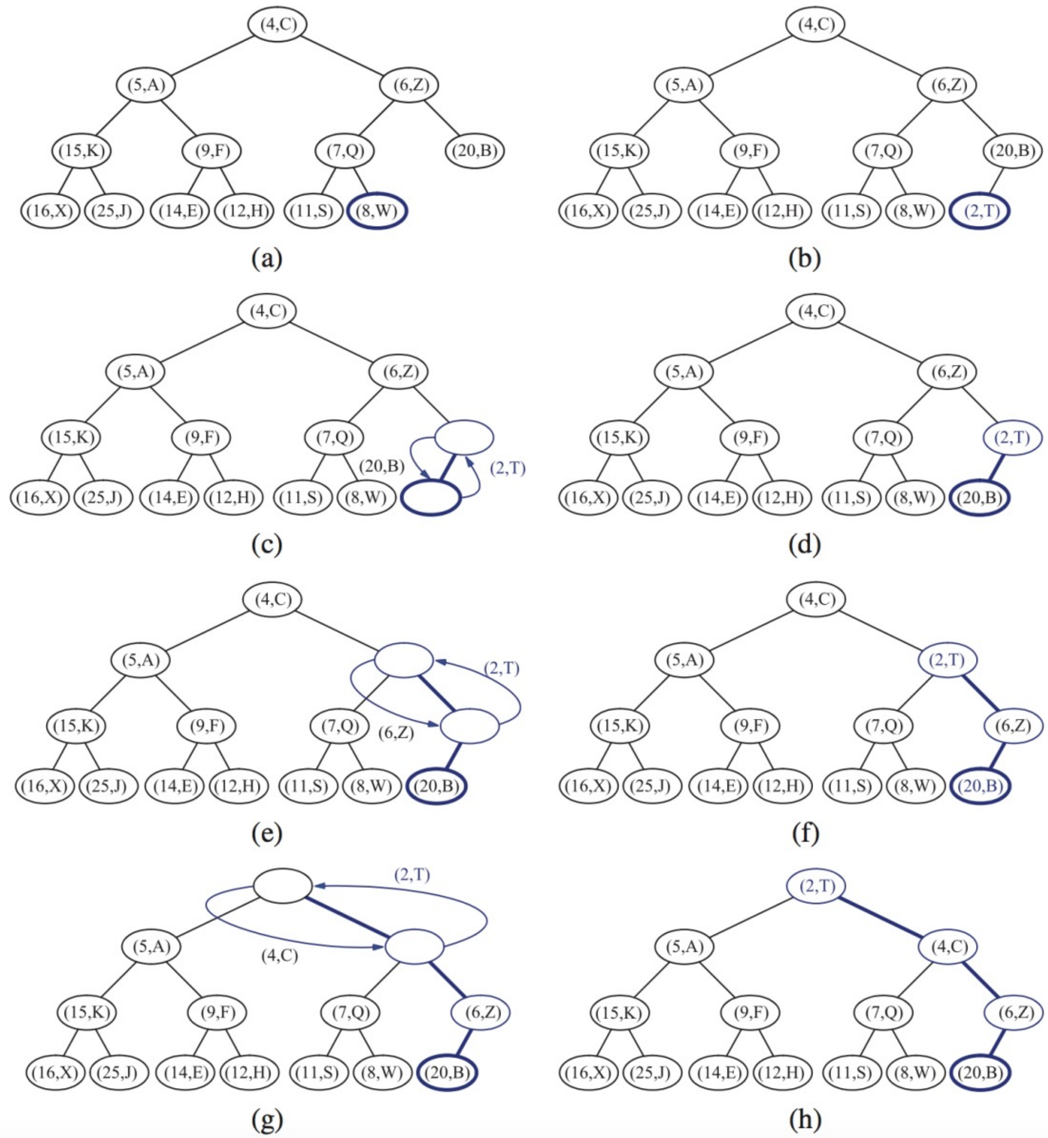
- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k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





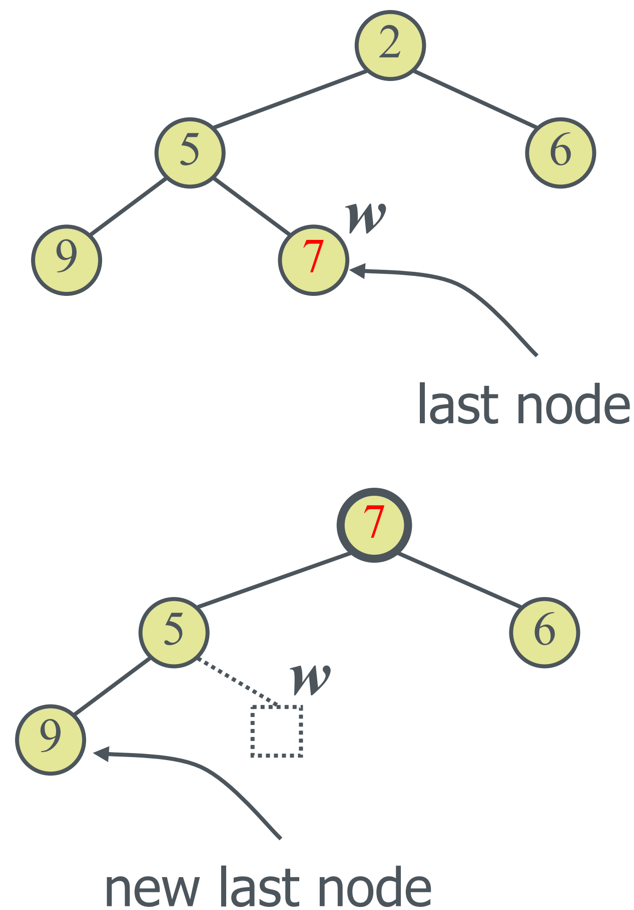
Upheap - Another Example

Insert: (2,T)



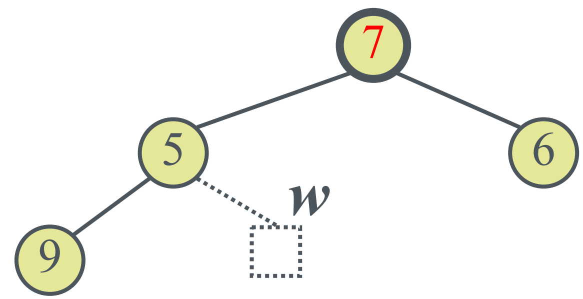
Removal from a Heap

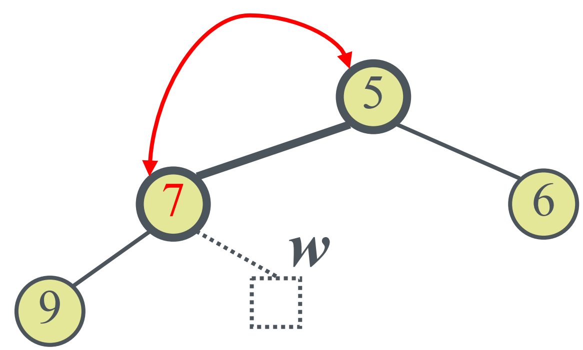
- Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)



Downheap

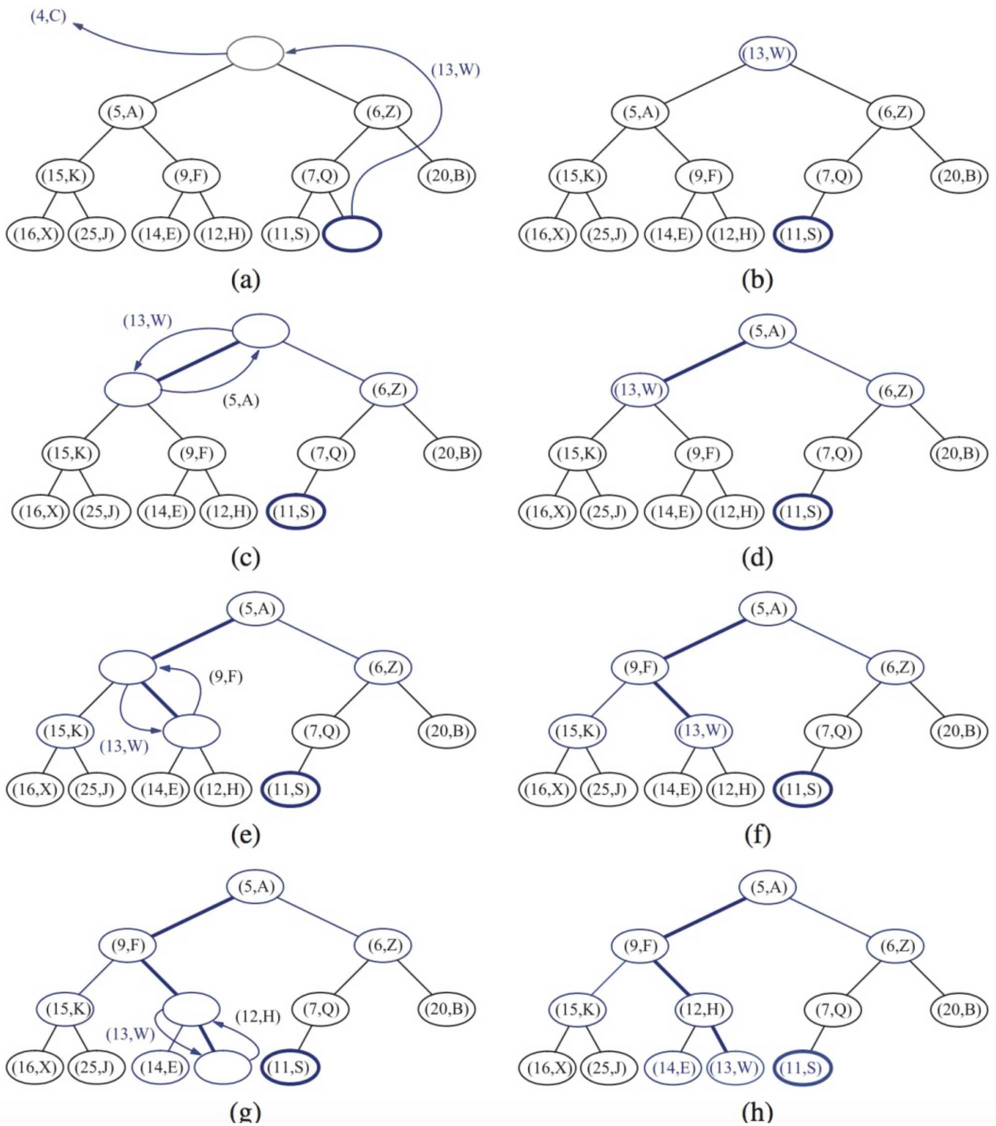
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap restores the heap-order property by swapping key k along a downward path from the root
- Upheap terminates when key k reaches a leaf or a node whose children have keys greater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





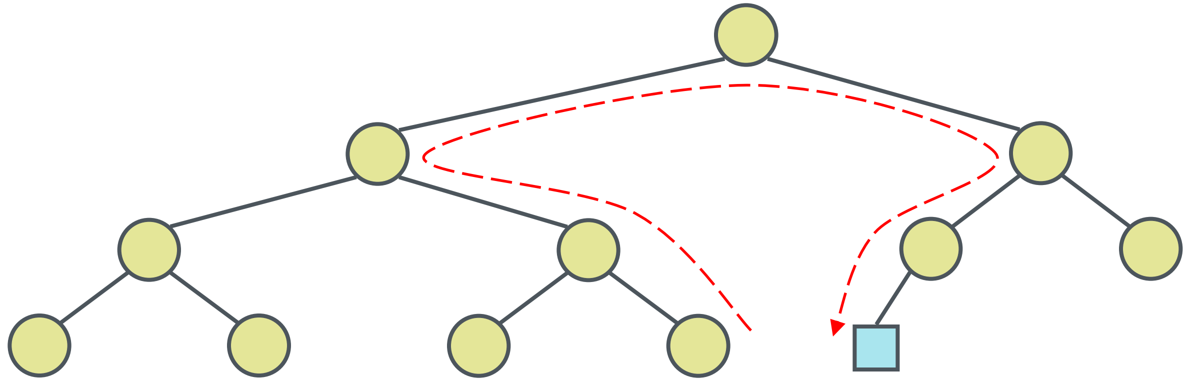
Downheap - Another Example

removeMin



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



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, 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.

Questions?