CS206 Data Structures

Priority Queues I

Sung-eui Yoon (윤성의)
Department of Computer Science
KAIST

http://sglab.kaist.ac.kr/~sungeui

Korea Advanced Institute of Science and Technology
한국과학기술원

Class Objectives (Ch. 9)

- ☐ Understand concepts of priority queue and heap
- ☐ Know different implementations for the priority queue and operations of the heap

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, x)
    inserts an entry with key k and value x
  - 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:
  - 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
- ☐ 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: $x \leq y \wedge y \leq z \Rightarrow x \leq z$

Entry ADT

- An entry in a priority queue is simply a key-value pair
- Priority queues store entries to allow for efficient insertion and removal based on keys
- Methods:
  - key(): returns the key for this entry
  - value(): returns the value associated with this entry

- As a Java interface:

```
/**
 * Interface for a key-value
 * pair entry
 **/
public interface Entry {
    public Object key();
    public Object value();
}
```

Comparator ADT

- A comparator encapsulates the action of comparing two objects according to a given total order relation
- A generic priority queue uses an auxiliary comparator
- The comparator is external to the keys being compared
- When the priority queue needs to compare two keys, it uses its comparator

- The primary method of the Comparator ADT:
  - compare(a, b): Returns an integer i such that $i < 0$ if $a < b$, $i = 0$ if $a = b$, and $i > 0$ if $a > b$; an error occurs if a and b cannot be compared.

Priority Queue Sorting

- We can use a priority queue to sort a set of comparable elements
  - Insert the elements one by one with a series of insert operations
  - 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.isEmpty*** ()
    e ← ***S.removeFirst*** ()
    P.insert (***e***, ***0***)
  while ¬***P.isEmpty***()
    e ← ***P.removeMin***().***key***()
    S.insertLast(***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

Selection-Sort

- Selection-sort is the variation of PQ-sort where the priority queue is implemented with an unsorted sequence
- Running time of Selection-sort:
  - Inserting the elements into the priority queue with n insert operations takes O(n) time
  - Removing the elements in sorted order from the priority queue with n removeMin operations takes time proportional to

    $$1 + 2 + \ldots + n$$
- Selection-sort runs in $O(n^2)$ time

Selection-Sort Example

	Sequence S	*Priority Queue P*
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(7,4)
..	.. ..	
.	. .	
(g)	()	(7,4,8,2,5,3,9)
Phase 2		
(a)	(2)	(7,4,8,5,3,9)
(b)	(2,3)	(7,4,8,5,9)
(c)	(2,3,4)	(7,8,5,9)
(d)	(2,3,4,5)	(7,8,9)
(e)	(2,3,4,5,7)	(8,9)
(f)	(2,3,4,5,7,8)	(9)
(g)	(2,3,4,5,7,8,9)	()

Insertion-Sort

- ☐ Insertion-sort is the variation of PQ-sort where the priority queue is implemented with a sorted sequence
- ☐ Running time of Insertion-sort:
  - Inserting the elements into the priority queue with n insert operations takes time proportional to

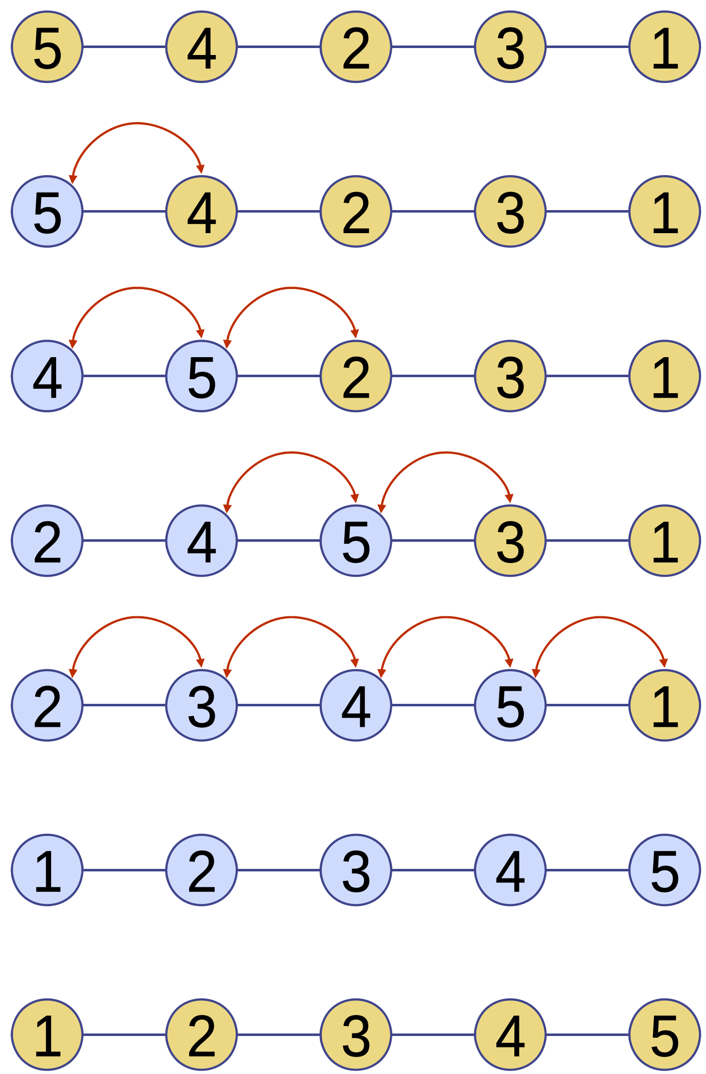
    $$1 + 2 + \ldots + n$$
  - Removing the elements in sorted order from the priority queue with a series of n removeMin operations takes $O(n)$ time
- ☐ Insertion-sort runs in $O(n^2)$ time

Insertion-Sort Example

	Sequence S	*Priority queue P*
Input:	(7,4,8,2,5,3,9)	()
Phase 1		
(a)	(4,8,2,5,3,9)	(7)
(b)	(8,2,5,3,9)	(4,7)
(c)	(2,5,3,9)	(4,7,8)
(d)	(5,3,9)	(2,4,7,8)
(e)	(3,9)	(2,4,5,7,8)
(f)	(9)	(2,3,4,5,7,8)
(g)	()	(2,3,4,5,7,8,9)
Phase 2		
(a)	(2)	(3,4,5,7,8,9)
(b)	(2,3)	(4,5,7,8,9)
..	..	..
.	.	.
(g)	(2,3,4,5,7,8,9)	()

In-Place Insertion-Sort

- Instead of using an external data structure, we can implement selection-sort and insertion-sort in-place
- A portion of the input sequence itself serves as the priority queue
- For in-place insertion-sort
  - We keep sorted the initial portion of the sequence
  - We can use swaps instead of modifying the sequence



Heaps

Recall 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, x)
    inserts an entry with key k and value x
  - 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:
  - Auctions
  - Stock market

Recall Priority Queue Sorting

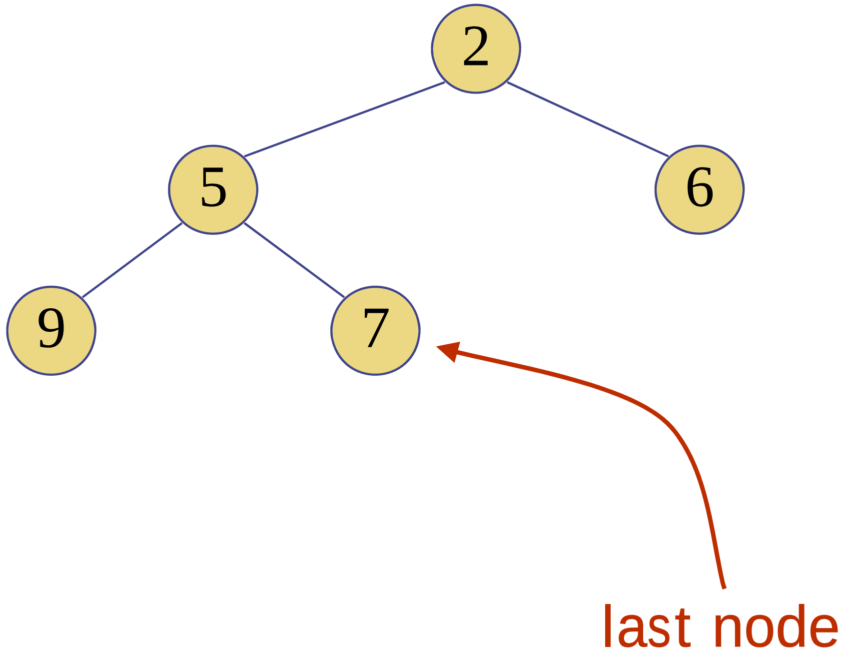
- We can use a priority queue to sort a set of comparable elements
  - Insert the elements with a series of insert operations
  - Remove the elements in sorted order with a series of removeMin operations
- The running time depends on the priority queue implementation:
  - Unsorted sequence gives selection-sort: $O(n^2)$ 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.insertItem(***e***, ***e***)
  while ¬***P.isEmpty***()
    e ← ***P.removeMin***()
    S.insertLast(***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) ≥ key(parent(v))
  - Complete Binary Tree: let h be the height of the heap
    - for i = 0, … , 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 depth h



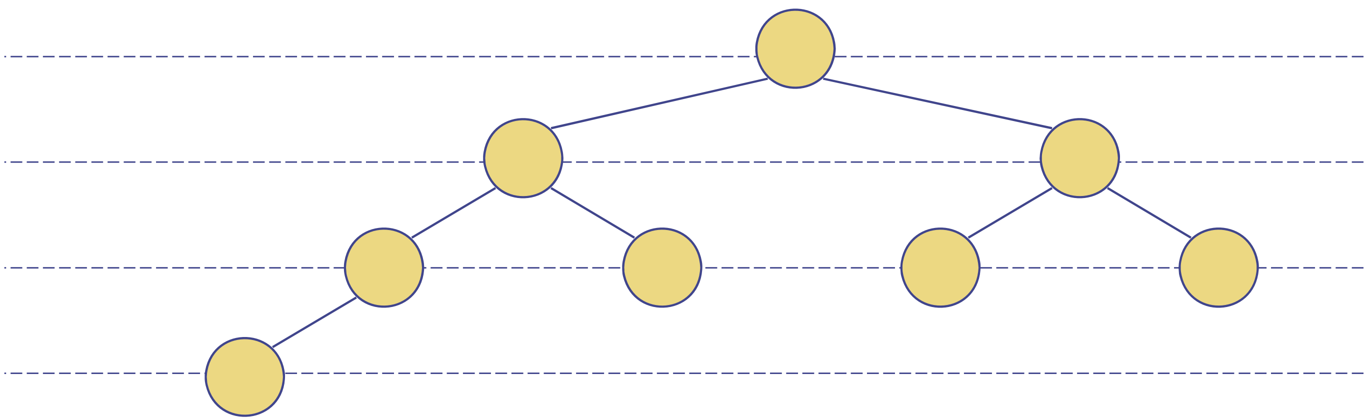
Height of a Heap

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

  Proof: (we apply the complete binary tree property)

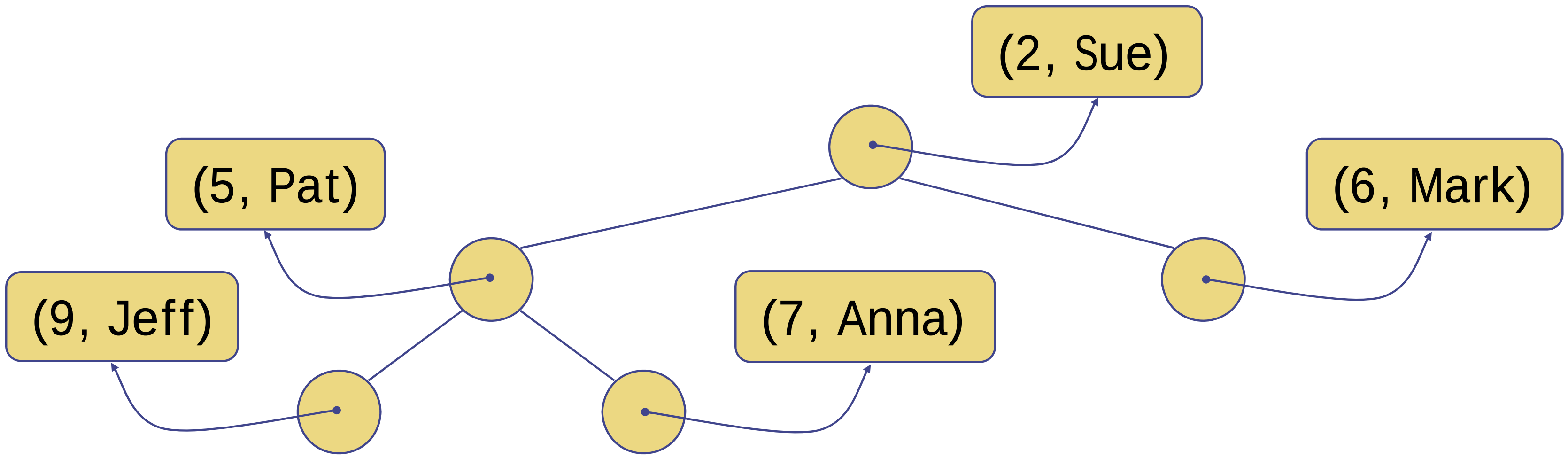
  - Let h be the height of a heap storing n keys
  - Since there are 2^i keys at depth $i = 0, \dots, h - 1$ and at least one key at depth h, we have $n \geq 1 + 2 + 4 + \dots + 2^{h-1} + 1$
  - Thus, $n \geq 2^h$, i.e., $h \leq \log n$

depth	keys
0	1
1	2
$h-1$	2^{h-1}
h	1



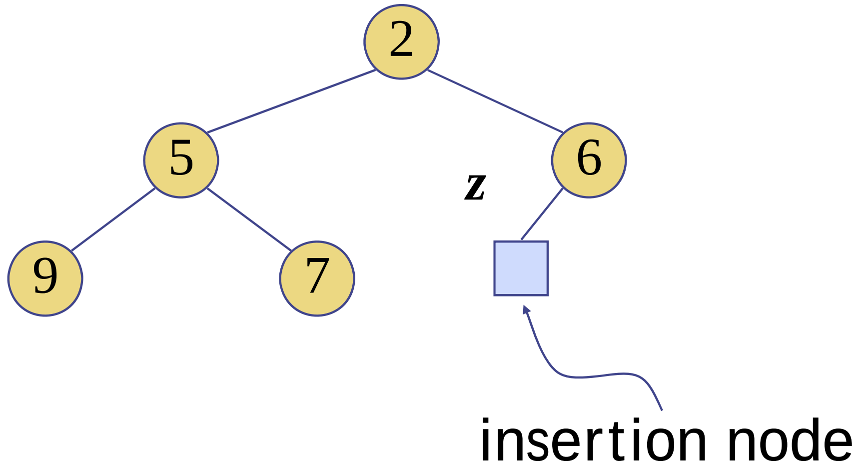
Heaps and Priority Queues

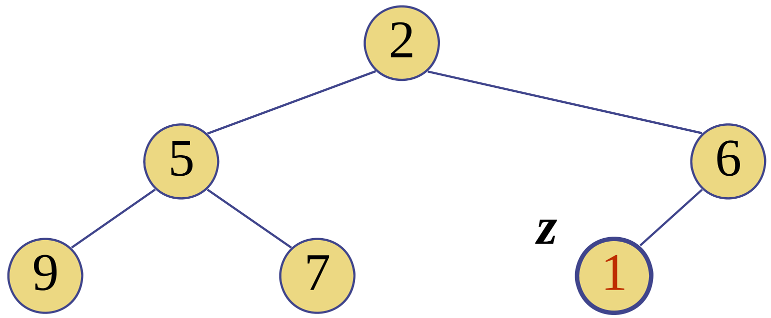
- We can use a heap to implement a priority queue
- We store a (key, element) item at each node
- We keep track of the position of the last node
- For simplicity, we show only the keys in the pictures



Insertion into a Heap

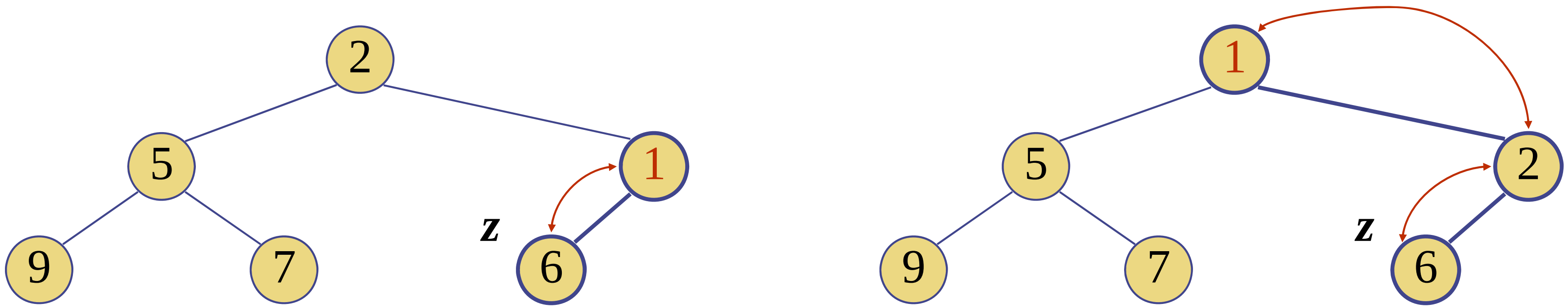
- Method insertItem of the priority queue ADT corresponds to the insertion of a key k to the heap
- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)





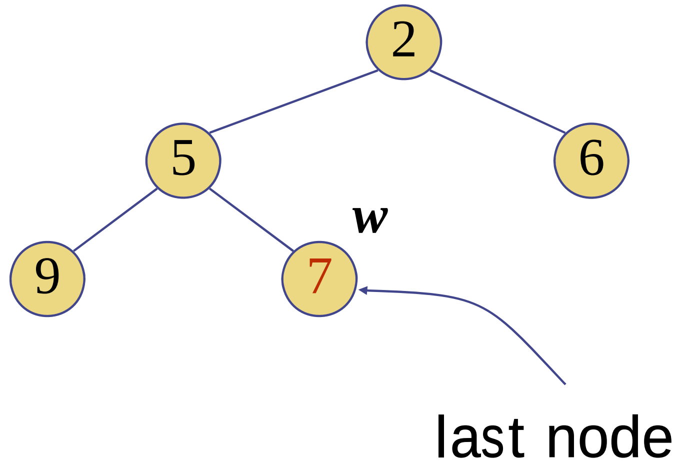
Upheap

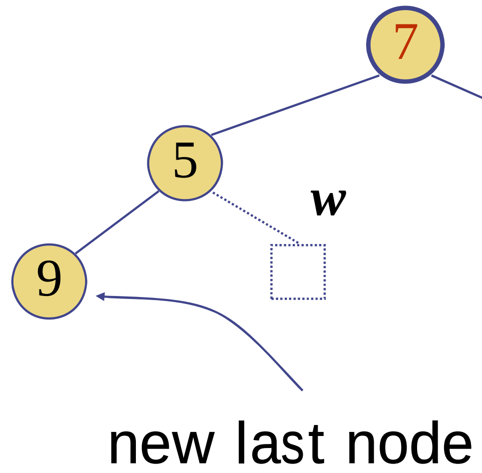
- ☐ After the insertion of a new key k, the heap-order property may be violated
- ☐ Algorithm upheap restores the heap-order property by swapping k along an upward path from the insertion node
  - Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal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



Removal from a Heap

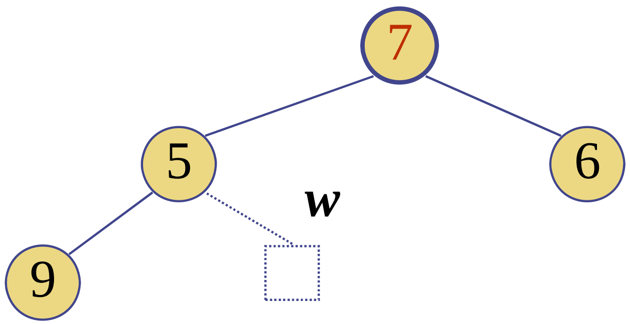
- ☐ Method removeMin of the priority queue ADT corresponds to the removal of the root key from the heap
- ☐ The removal algorithm consists of three steps
  - Replace the root key with the key of the last node w
  - Remove w
  - Restore the heap-order property (discussed next)

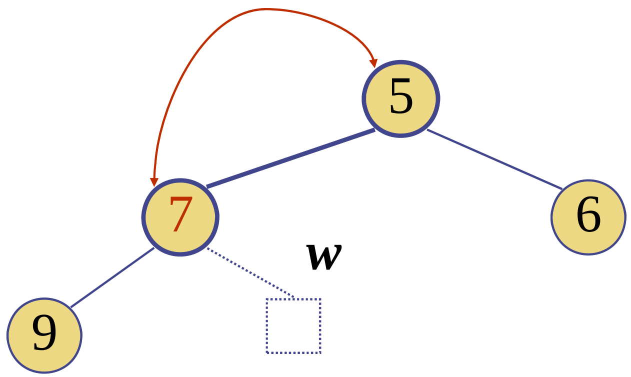




Downheap

- 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
  - 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





Class Objectives were:

- ☐ Understand concepts of priority queue and heap
- ☐ Know different implementations for the priority queue and operations of the heap

Next Time

- ☐ Heap sort

- ☐ Questions:
  - Come up with one question on what we have discussed in the class and submit at the end of the class
  - 1 for typical questions and 2 for questions with thoughts or that surprised me
  - Write questions at least 4 times; you can type at KLMS

- ☐ HW:
  - Go over the next lecture slides before the class
  - Just 10 min ~ 20 min should be okay