HEAPS (9.3)

EECS 2011

Priority Queue ADT (9.1, 9.2)

- 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, or null if the the priority queue is empty
- Additional methods
  - min()
    returns, but does not remove, an entry with smallest key, or null if the the priority queue is empty
  - size(), isEmpty()
- Applications:
  - Standby flyers
  - Auctions
  - Emergency room waiting list
  - Routing priority at routers in a network

Example

- A sequence of priority queue methods:

Method	Return Value	Priority Queue Contents
insert(5,A)		{ (5,A) }
insert(9,C)		{ (5,A), (9,C) }
insert(3,B)		{ (3,B), (5,A), (9,C) }
min( )	(3,B)	{ (3,B), (5,A), (9,C) }
removeMin( )	(3,B)	{ (5,A), (9,C) }
insert(7,D)		{ (5,A), (7,D), (9,C) }
removeMin( )	(5,A)	{ (7,D), (9,C) }
removeMin( )	(7,D)	{ (9,C) }
removeMin( )	(9,C)	{ }
removeMin( )	null	{ }
isEmpty( )	true	{ }

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:
  - getKey: returns the key for this entry
  - getValue: returns the value associated with this entry

- As a Java interface:

```
/**
 * Interface for a key-value
 * pair entry
 **/
public interface Entry<K,V> {
        K getKey();
        V getValue();
}
```

5

Implementations of PQs

Data structure	*insert*( )	*min*( )	*deleteMin*( )
Unsorted array	O( )	O( )	O( )
Sorted array	O( )	O( )	O( )
Unsorted doubly linked list	O( )	O( )	O( )
Sorted doubly linked list	O( )	O( )	O( )
AVL trees	O( )	O( )	O( )

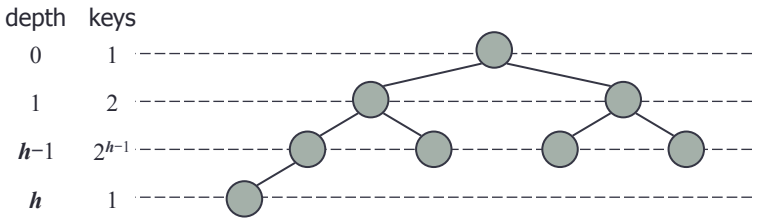
A data structure more efficient for PQs is **heaps**:

- *insert*( ), *deleteMin*( ): O(*logn*)
- *min*( ): O(1)

6

Complete Binary Trees

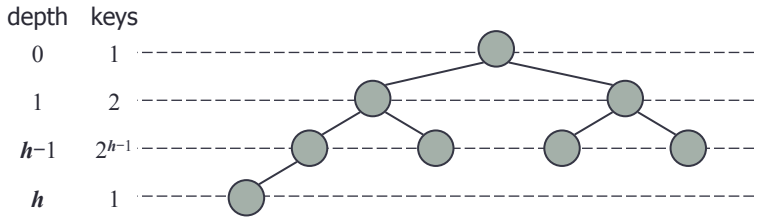
- Let h be the height of a binary tree.
  - for $i = 0, \ldots, h-1$, there are 2^i nodes at depth i.
    - that is, all levels except the last are full.
  - at depth h, the nodes are filled from left to right.



7

Complete Binary Trees (2)

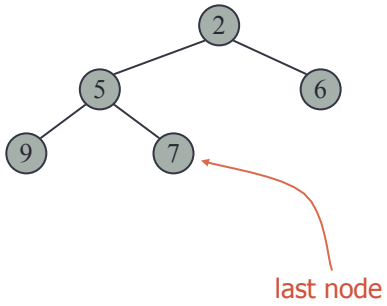
- Given a complete binary tree of height h and size n, $2^h \le n \le 2^{h+1} - 1$
- Which data structure is better for implementing complete binary trees, arrays or linked structures?



8

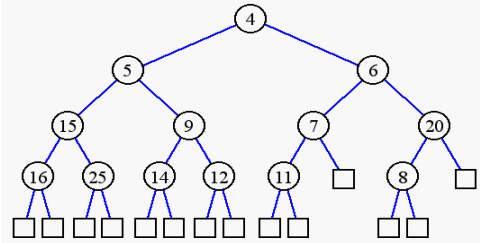
Heaps

- 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, $key(v) \ge key(parent(v))$
  - Complete Binary Tree: let h be the height of the heap
    - for $i = 0, \ldots, h - 1$, there are 2^i nodes at depth i.
    - at depth h, the nodes are filled from left to right.
- The last node of a heap is the rightmost node of depth h.
- Where can we find the smallest key in a min heap? The largest key?

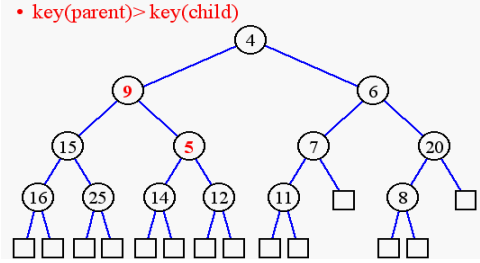


Examples that are not heaps

- bottom level is not left-filled



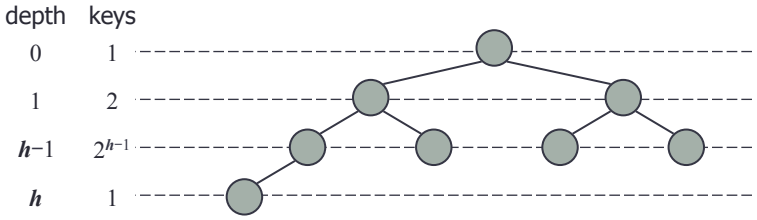
- key(parent)> key(child)



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., $h \leq \log n$



11

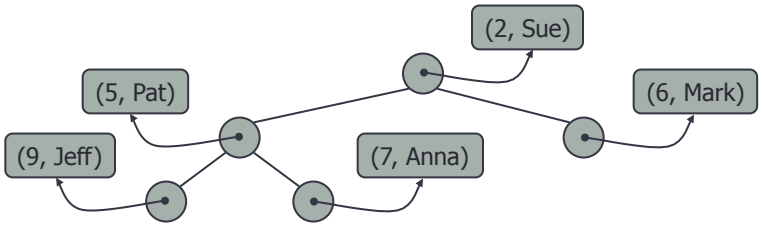
Max Heap

- The definition we just discussed is for a *min* heap.
- Analogously, we can declare a *max* heap if we need to implement *deleteMax* operation instead of *deleteMin*.

12

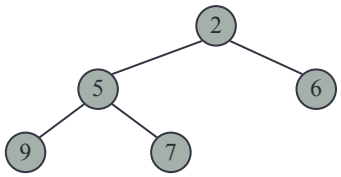
Heaps and Priority Queues

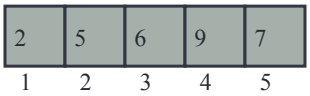
- We can use a heap to implement a priority queue
- We store a (key, value) item at each internal node
- We keep track of the position of the last node



Array-based Heap Implementation

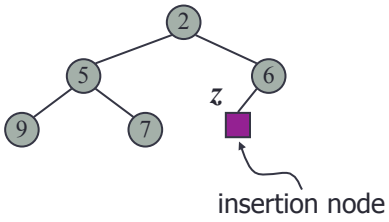
- We can represent a heap with n keys by means of an array of length n
- The root is at rank 1.
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$

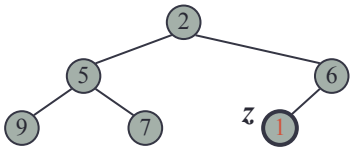




Insertion into a Heap

- 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
  1. Find the insertion node z (the new last node)
  2. Store k at z
  3. Restore the heap-order property (discussed next)

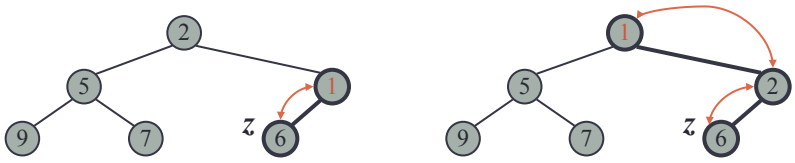




15

Upheap Percolation (Bubbling)

- 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

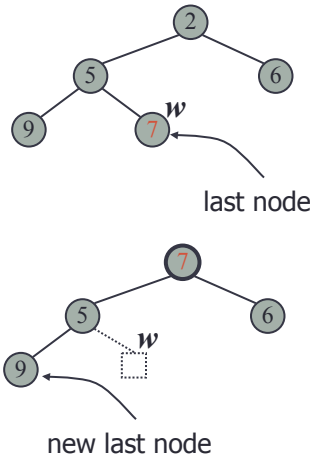


http://www.cs.usfca.edu/~galles/JavascriptVisual/Heap.html

16

Removal from a Heap (*deleteMin*)

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



Downheap Percolation

- 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

More Heap Operations

Assume a min heap.

- decreaseKey(i, k)
  - T[i] = T[i] - k, then percolate up.
  - Example: system admin boosts the priority of his/her jobs.
- increaseKey(i, k)
  - T[i] = T[i] + k, then percolate down.
  - Example: penalizing misbehaved processes.
- delete(i)
  - Perform decreaseKey(i, ∞) then deleteMin().
    - ∞ means a very large number, so T[i] = T[i] − ∞ has the highest priority (root)
  - Example: removing a print job from the priority queue.

- Note: searching for the element at index i takes $O(n)$ time in the worst case, but we expect not to use the above methods very often.

Next lecture …

- Heap Sort (9.4.2)