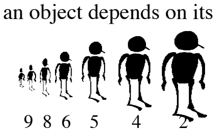
Priority queue ADT
Heaps

Lecture 21

Priority queue ADT

- Like a dictionary, a priority queue stores a set of pairs (key, info)
- The rank of an object depends on its priority (key)

Rear of queue  Front of queue

key: 9 8 6 5 4 2

- Allows only access to
  - Object findMin() //returns info of smallest key
  - Object removeMin() // removes smallest key
  - void insert(key k, info i) // inserts pair
- Applications: customers in line, Data compression, Graph searching, Artificial intelligence…

Priority queue ADT

(**4**,O_4)	(**5**, O_5)	(**8**,O_8)		

insert(9,O_9)

(**4**,O_4)	(**5**, O_5)	(**8**,O_8)	(**9**,O_9)	

remove()

(**5**, O_5)	(**8**,O_8)	(**9**,O_9)		

insert(6,O_6)

(**5**, O_5)	(**6**,O_6)	(**8**,O_8)	(**9**,O_9)	

insert(2,O_2)

(**2**, O_2)	(**5**, O_5)	(**6**,O_6)	(**8**,O_8)	(**9**,O_9)

Implementation of priority queue

Unsorted array of pairs (key, info)

- findMin(): Need to scan array O(n)
- insert(key, info): Put new object at the end O(1)
- removeMin(): First, findMin, then shift array O(n)

Sorted array of pairs (key, info)

- findMin(): Just return first element O(1)
- insert(key, info):
  - Use binary-search to find position of insertion. O(log n)
  - Then shift array to make space. O(n)

Implementation of priority queue

Using a sorted doubly-linked list of pairs (key, info)

findMin(): Return first element O(1)

insert(key, info):

First, find location of insertion.

Binary Search?

Slow on linked list.

Instead, we have to scan array O(n)

Then insertion is easy O(1)

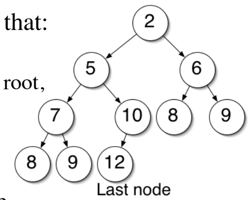
removeMin(): Remove first element of list O(1)

Heap data structure

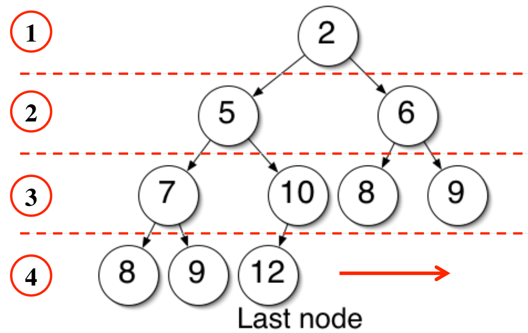
- A heap is a data structure that implements a priority queue:
  - findMin(): O(1)
  - removeMin(): O(log n)
  - insert(key, info): O(log n)
- A heap is based on a binary tree, but with a different property than a binary search tree
- heap ≠ binary *search* tree

Heap - Definition

- A **heap** is a binary tree such that:
  - For any node *n* other than the root, key(n) ≥ key(parent(n))
  - Let h be the height of the heap
    - First h-1 levels are full: For i = 0,…,h-1, there are 2^i nodes of depth i
    - At depth h, the leaves are packed on the left side of the tree



Heap - Example



Height of a heap

What is the maximum number of nodes that fits in a heap of height h?

$$\sum_{k=0}^{h} 2^k = 2^{h+1} - 1$$

What is the minimum number?

$$(2^h - 1) + 1 = 2^h$$

Thus, the height of a heap with n nodes is:

$$\lfloor \log(n) \rfloor$$

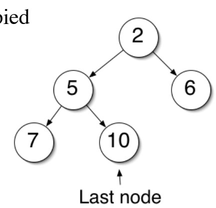
Heaps: findMin()

The minimum key is always at the root of the heap!

Heaps: Insert

Insert(key k, info i). Two steps:

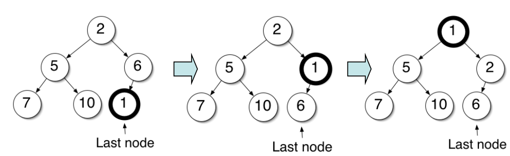
1. Find the left-most unoccupied node and insert (k,i) there temporarily.
2. Restore the heap-order property (see next)



Heaps: Bubbling-up

Restoring the heap-order property:

- Keep swapping new node with it' s parent as long as it' s key is smaller than it' s parent' s key



Running time? $O(h) = O(\log(n))$

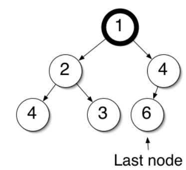
Insert pseudocode

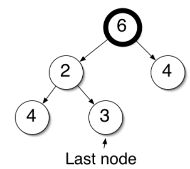
Algorithm insert(key k, info i)
Input: The key k and info i to be added to the heap
Output: (k,i) is added

lastNode ← nextAvailableNode(lastNode)
lastNode.key ← k,   lastNode.info ← i
n ← lastnode
while (n.getParent()!=null **and** n.getParent().key > k) **do**
    swap (n.getParent(), n)

Heaps: RemoveMin()

- The minimum key is always at the root of the heap!
- Replace the root with last node



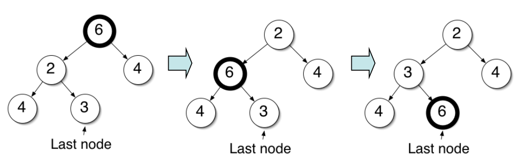


- Restore heap-order property (see next)

Heaps: Bubbling-down

Restoring the heap-order property:

– Keep swapping the node with its smallest child as long as the node's key is larger than it's child's key



Running time? $O(h) = O(\log(n))$

removeMin pseudocode

Algorithm removeMin()
Input: The key k and info I to be added to the heap
Output: (k,i) is added

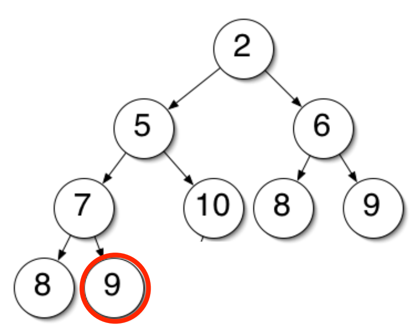
swap(lastNode, root)
Update lastNode
n ← root
while (n.key > min(n.getLeftChild().key, n.getRightChild().key)) **do**
    if (n.getLeftChild().key < n.getRightChild().key) **then**
        swap(n, n.getLeftChild)
    else swap(n, n.getRightChild)

Finding nextAvailableNode

nextAvailableNode(lastNode) finds the location where the next node should be inserted. It runs in time O(n).

n <-- lastNode;
while (n is the right child of its parent && n.parent!=null) do
    n <-- n.parent
if (n.parent == null) then nextAvailableNode is the left child of the leftmost node of the tree
else
    n <-- n.parent   // go up one more level
    if (n has no right child) then nextAvailableNode is the right child of n
    else
        n <-- n.rightChild   // go down the right child
        while (n has a left child) do   n <-- n.leftChild
        nextAvailableNode is the left child of n

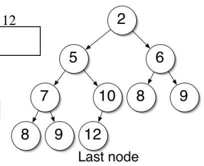
NextAvailableNode - Example



Array representation of heaps

- A heap with n keys can be stored in an array of length n+1

0	1	2	3	4	5	6	7	8	9	10	11	12
-	2	5	6	7	10	8	9	8	9	12		



- For a node at index i,
  - The parent (if any) is at index $\lfloor i/2 \rfloor$
  - The left child is at index 2*i
  - The right child is at index 2*i + 1
- lastNode is the first empty cell of the array. To update it, either add or subtract one

HeapSort

Algorithm heapSort(array A[0…n-1])
Heap h ← new Heap()
for i=0 **to** n-1 **do**
  h.insert(A[i])
for i=0 **to** n-1 **do**
  A[i] ← h.removeMin()

Running time: O(n log n) in worst-case
Easy to do in-place: Just use the array A to store the heap