Data Structures and Algorithms

Spring 2009-2010

Outline

Announcements

- Mid-term: 15.00,Thursday, Week08 in SR3006.

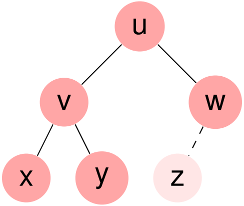
Outline

1. Announcements

2. Priority Queues (contd.)
   - Binary Heaps
   - Binary Heap Operations

Binary Heaps: Introduction

A heap is a special type of binary tree that is **completely filled** except for the bottom level, on which level the nodes are filled from left to right



Heap Height and Representations

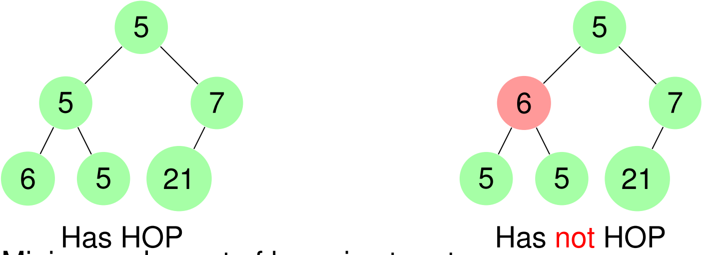
- A heap of height h has a maximum of $\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$ nodes (a full bottom level) and a minimum of $2^h (= 1 + \sum_{i=0}^{h-1} 2^i)$ nodes
- Thus, a heap of n nodes has height $\lfloor \log n \rfloor = O(\log n)$
- Heaps can be represented in an array

	u	*v*	*w*	*x*	*y*	*z*		
0	*1*	*2*	*3*	*4*	*5*	*6*		

- For any element in position i of the array its relatives' locations are:
  - left child in position $2i$
  - right child in position $2i + 1$
  - parent in position $\lfloor i/2 \rfloor$

Heap Order Property

- A heap is said to have the **heap order property** if the smallest element of every subtree is at the root of the subtree
- A BST does not have this property



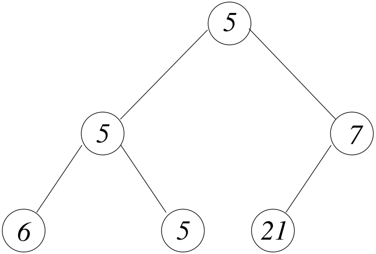
- ➜ Minimum element of heap is at root
- ➜ To check for heap property just need to trace from every leaf node back to root; no need to compare siblings
- ➜ A heap is a much looser structured Data Structure than a Binary Search Tree

Outline

1. Announcements
2. Priority Queues (contd.)
   - Binary Heaps
   - Binary Heap Operations

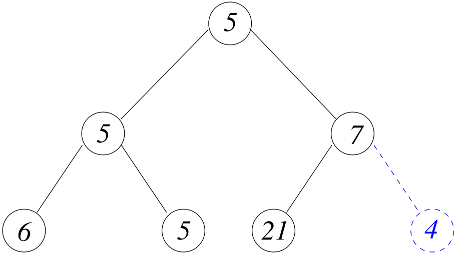
Heap Operations: `insert()`

- If there are *n* elements in the heap, insert the new element at the first empty location in the array
- If the heap property is not violated, we're finished
- Otherwise, by swapping the new (and offending) element with its parent percolate it up the tree until the heap order property is satisfied
- Inserting the value **4** into the heap below:



Heap Operations: `insert()` (contd.)

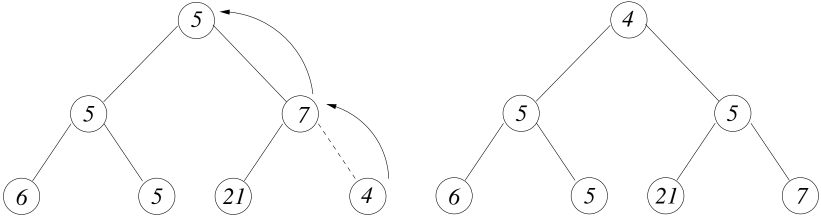
- Insert at end of array (leftmost free slot on bottom level of tree):



- Percolate up:
- `void BinaryHeap<Comparable>::insert(const Comparable & x )` is here

Heap Operations: `insert()` (contd.)

- Insert at end of array (leftmost free slot on bottom level of tree):
- Percolate up:



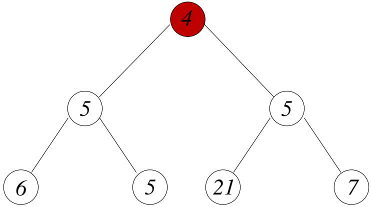
- `void BinaryHeap<Comparable>::insert(const Comparable & x )` is here

Heap Operations: `insert()` (contd.)

- Insert at end of array (leftmost free slot on bottom level of tree):
- Percolate up:
- `void BinaryHeap<Comparable>::insert(const Comparable & x )` is here

Heap Operations: `delete_min()`

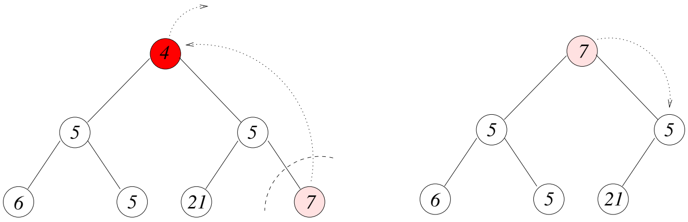
- It is easy to extract the min element from a heap – will be at $i = 1$ in array
- Hard part in deletions is reconstructing the heap
- If there were n elements in the heap prior to deletion, take the nth element and place it at $i = 1$
- Trickle *down* the offending element until heap order is no longer violated
- First, remove root element



(Before)

Heap Operations: `delete_min()` (contd.)

- Then, remove **last** element of array and place at root position

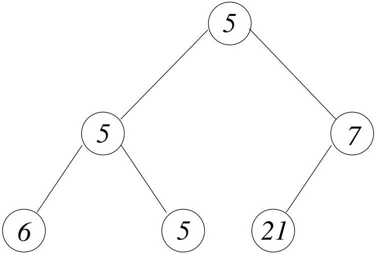


(i) (ii)

- Trickle down this element; need to look at **both** offspring:
- `void BinaryHeap<Comparable>::deleteMin(const Comparable & x )` is here

Heap Operations: `delete_min()` (contd.)

- Then, remove **last** element of array and place at root position
- Trickle down this element; need to look at **both** offspring:



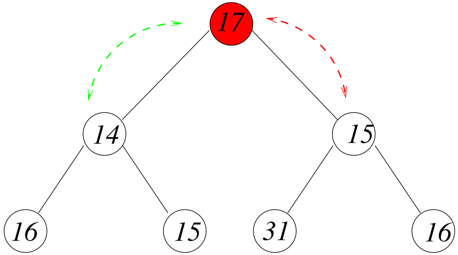
- `void BinaryHeap<Comparable>::deleteMin(const Comparable & x )` is here

Heap Operations: `delete_min()` (contd.)

- Then, remove **last** element of array and place at root position
- Trickle down this element; need to look at **both** offspring:
- `void BinaryHeap<Comparable>::deleteMin(const Comparable & x )` is here

Heap Operations: `delete_min()` (contd.)

- When trickling down, we must make sure that we swap with the *smaller* of the two children



- How many levels we will have to trickle down depends on which side we trickle down and we cannot always predict this
- Sometimes difficult to know how to break a tie:

✓ But we know that we will have to do at most $O(\log n)$ trickles, in the worst case

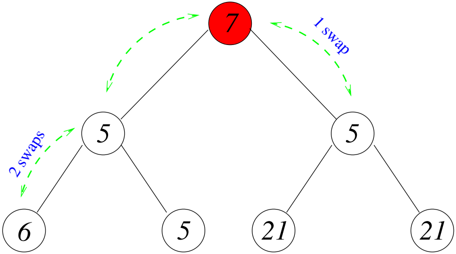
Heap Operations: `delete_min()` (contd.)

- When trickling down, we must make sure that we swap with the *smaller* of the two children
- How many levels we will have to trickle down depends on which side we trickle down and we cannot always predict this
- Sometimes difficult to know how to break a tie:

✓ But we know that we will have to do at most $O(\log n)$ trickles, in the worst case

Heap Operations: `delete_min()` (contd.)

- When trickling down, we must make sure that we swap with the *smaller* of the two children
- How many levels we will have to trickle down depends on which side we trickle down and we cannot always predict this
- Sometimes difficult to know how to break a tie:



✓ But we know that we will have to do at most $O(\log n)$ trickles, in the worst case

Heap Operations: `delete_min()` (contd.)

- When trickling down, we must make sure that we swap with the *smaller* of the two children
- How many levels we will have to trickle down depends on which side we trickle down and we cannot always predict this
- Sometimes difficult to know how to break a tie:

✓ But we know that we will have to do at most $O(\log n)$ trickles, in the worst case