CSE 326: Data Structures

Priority Queues – Binary Heaps

Brian Curless

Spring 2008

Administrative

- HW1 due beginning of class Friday
- P1 due Wednesday
  - Electronic submission by end of Wednesday (11:59 +ε PM PDT, ε <0:01)
- Reading for this week: Chapter 6.

Recall Queues

- FIFO: First-In, First-Out
  - Print jobs
  - File serving
  - Phone calls and operators
  - Lines at the Department of Licensing…

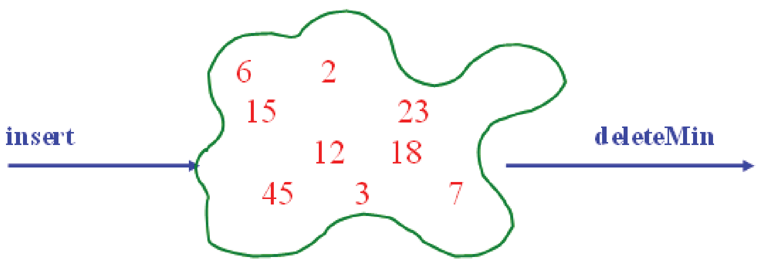
Priority Queues

Often, we want to allow skipping to front of the line – a **priority queue**:

- Select print jobs in order of decreasing length
- Forward packets on routers in order of urgency
- Operating system can favor jobs of shorter duration or those tagged as having higher importance
- Greedy optimization: "best first" problem solving

Priority Queue ADT

- Need a new ADT
- Operations: Insert an Item,
  Remove the "Best" Item



Priority Queue ADT

1. **PQueue <u>data</u>** : collection of data with priority
2. **PQueue <u>operations</u>**
   - insert
   - deleteMin

   (also: create, destroy, is_empty)
3. **PQueue <u>property</u>**: for two elements in the queue, x and y, if x has a **<u>lower</u>** priority value than y, x will be deleted before y

Potential implementations

Worst case

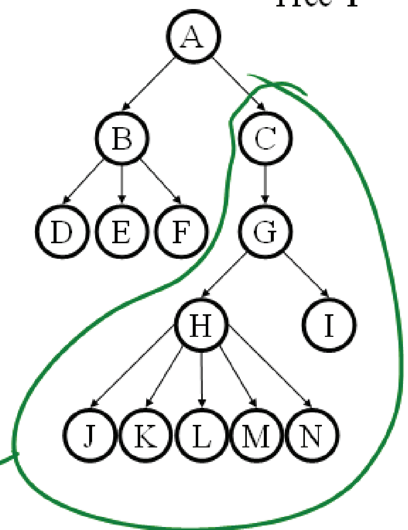
	insert	deleteMin
Unsorted list (Array)	O(1)	O(N)
Unsorted list (Linked-List)	O(1)	O(N)
Sorted list (Array)	O(N)	O(1)
Sorted list (Linked-List)	O(N)	O(1)
Binary Search Tree (BST)	O(N)	O(N)

Choosing the Right ADT

- Use an ADT that corresponds to your needs
- The right ADT is efficient, while an overly general ADT provides functionality you aren't using, but are paying for anyways
- Today we look at using a **binary heap** (a kind of binary tree) for priority queues:
  - O(log n) worst case for both insert and deleteMin
  - O(1) average insert
- What if priority is equal to seconds since creation of priority queue? queue

Tree Review

Tree T



root(**T**): A

leaves(**T**): D-F, I-N

children(B): DEF

parent(H): G

siblings(E): DF

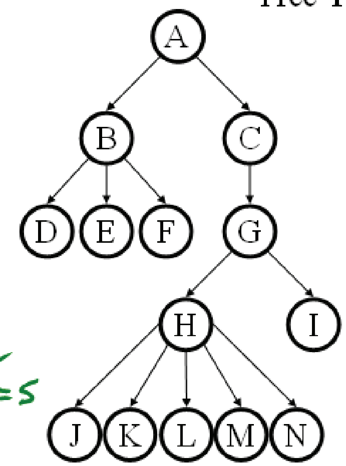
ancestors(F): B A

descendents(G): H-N

subtree(C):

More Tree Terminology

Tree T



depth(B): 1

height(G): 2

height(**T**): height(root) = height(A) = 4

degree(B): 3

branching factor(**T**): Max degree over all nodes = 5

n-ary tree: n = max branching factor(T)

5-ary tree

Binary Heap Properties

A binary heap is a binary tree with two important properties that make it a good choice for priority queues:

1. **Structure Property**
2. **Ordering Property**

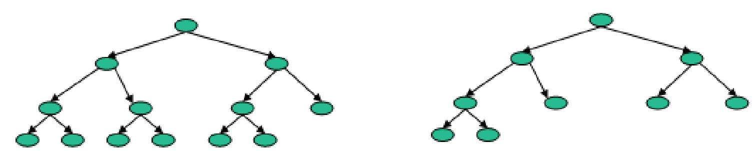
Note: we will sometimes refer to a binary heap as simply a "heap".

Heap **Structure** Property

- A binary heap is a ***complete*** binary tree.

Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:



Height of a complete binary tree with n nodes?

3. $h < \log_2(n+1) \le h+1$

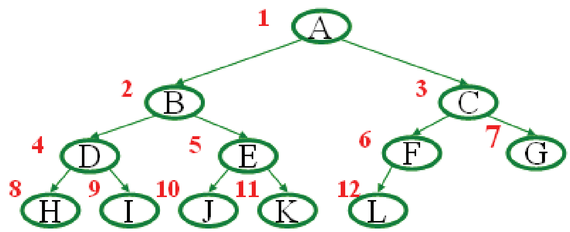
4. $h+1 = \lceil \log_2(n+1) \rceil$

1. $2^h - 1 < n \le 2^{h+1} - 1$

2. $2^h < n+1 \le 2^{h+1}$

5. $h = \lceil \log_2(n+1) \rceil - 1$

Representing Complete Binary Trees in an Array



From node **i**:

left child: $2i$
right child: $2i+1$
parent: $\lfloor i/2 \rfloor$

implicit (array) implementation:

	A	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Why this approach to storage?

Easier ...

Random access

Less memory

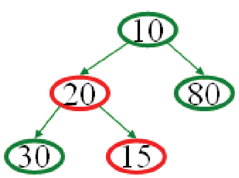
... better memory performance

Easy to serialize

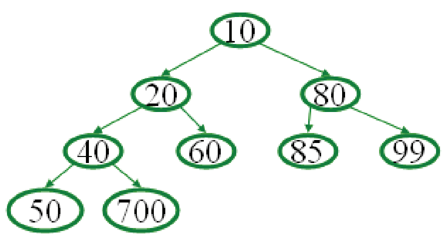
Easy to get to parent

Heap **Order** Property

Heap order property: For every non-root node X, the value in the parent of X is less than (or equal to) the value in X.

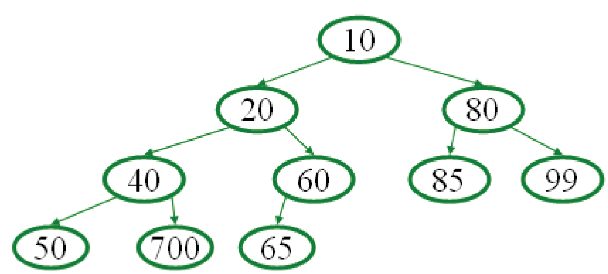


not a heap



Heap Operations

- findMin:
- insert(val): percolate up.
- deleteMin: percolate down.



Working on Heaps

- What are the two properties of a heap?
  - Structure Property
  - Order Property
- How do we work on heaps?
  - Fix the structure
  - Fix the order

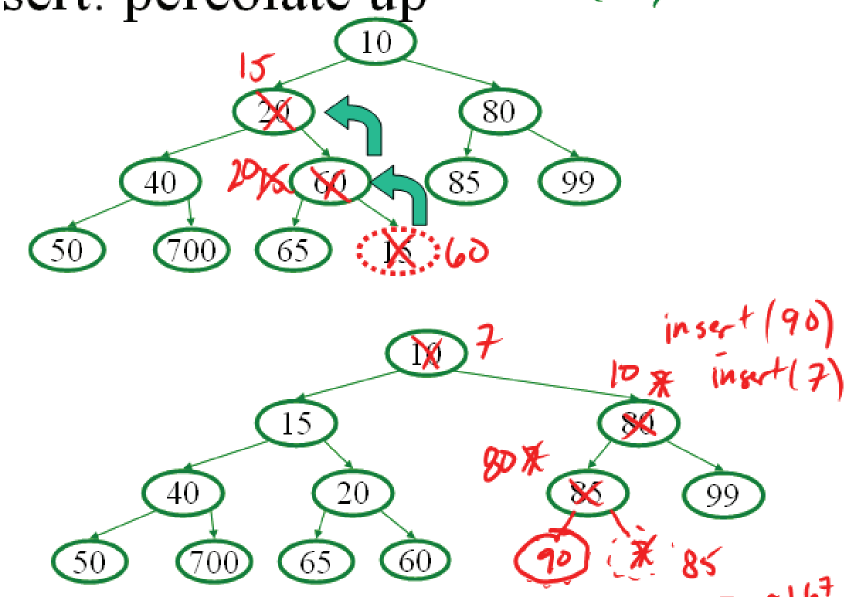
Heap – insert(val)

Basic Idea:

1. Put val at "next" leaf position
2. Percolate up by repeatedly exchanging node until no longer needed

Insert: percolate up

insert (15)



$T_{worst} \in \Theta(\log n)$ $T_{avg} \approx 1.67 \in \Theta(1)$

Insert Code (optimized)

```
void insert(Object o) {
  assert(!isFull());
  size++;
  newPos =
    percolateUp(size,o);
  Heap[newPos] = o;
}
```

```
int percolateUp(int hole,
                Object val) {
  while (hole > 1 &&
         val < Heap[hole/2])
    Heap[hole] = Heap[hole/2];
    hole /= 2;
  }
  return hole;
}
```

runtime: $T_{worst} = \Theta(\log n)$

$T_{avg} = \Theta(1)$

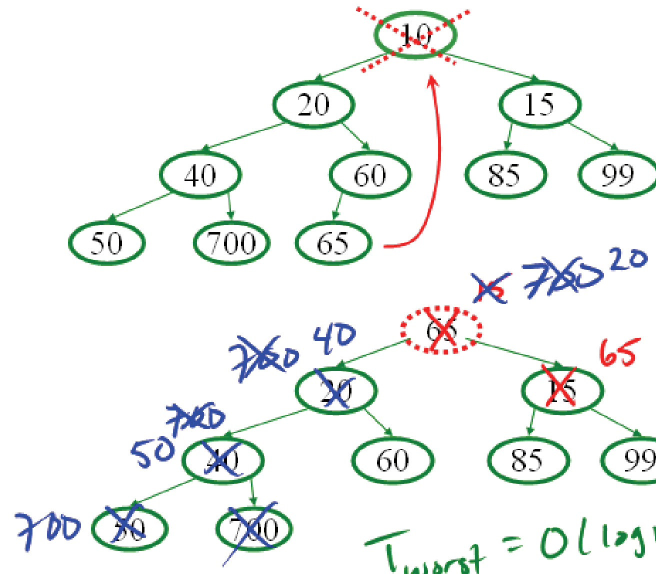
(Java code in book)

Heap – deleteMin

Basic Idea:

1. Remove root (that is always the min!)
2. Put "last" leaf node at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

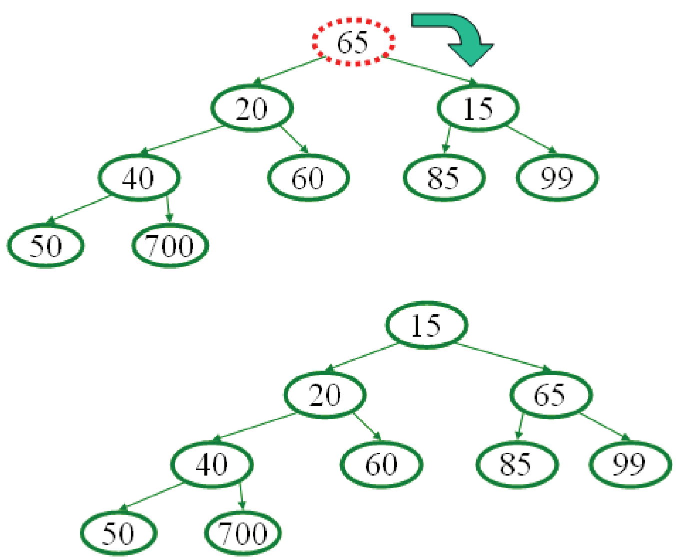
DeleteMin: percolate down



$T_{worst} = O(\log n)$

$T_{avg} = O(\log n)$

DeleteMin: percolate down



DeleteMin Code (Optimized)

```
Object deleteMin() {
  assert(!isEmpty());
  returnVal = Heap[1];
  size--;
  newPos =
    percolateDown(1,
        Heap[size+1]);
  Heap[newPos] =
    Heap[size + 1];
  return returnVal;
}
```

runtime:

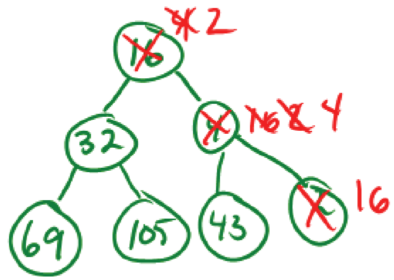
(Java code in book)

```
int percolateDown(int hole,
                  Object val) {
while (2*hole <= size) {
    left = 2*hole;
    right = left + 1;
    if (right ≤ size &&
        Heap[right] < Heap[left])
      target = right;
    else
      target = left;

    if (Heap[target] < val) {
      Heap[hole] = Heap[target];
      hole = target;
    }
    else
      break;
  }
  return hole;
}
```

Insert: 16, 32, 4, 69, 105, 43, 2

0	1	2	3	4	5	6	7	8
	2	32	4	69	105	43	16	



More Priority Queue Operations

decreaseKey(objPtr, amount): ← > 0

given a pointer to an object in the queue, reduce its priority value

Binary heap: change priority of node and percolateUp
(→ subtract amount) O(log n)

increaseKey(objPtr, amount):

given a pointer to an object in the queue, increase its priority value

Binary heap: change priority of node and percolateDown
(→ add amount) O(log n)

Why do we need a *pointer*? Why not simply data value?
avoid searching

Worst case running times?

More Priority Queue Operations

remove(objPtr):

given a pointer to an object in the queue, remove it

Binary heap: decreaseKey(objPtr, ∞), deleteMin O(log n)

findMax( ):

Find the object with the highest value in the queue

Binary heap: search the leaves O(n)

Worst case running times?

More Binary Heap Operations

expandHeap( ):

If heap has used up array, copy to new, larger array.

- Running time: O(n)

buildHeap(objList):

Given list of objects with priorities, fill the heap.

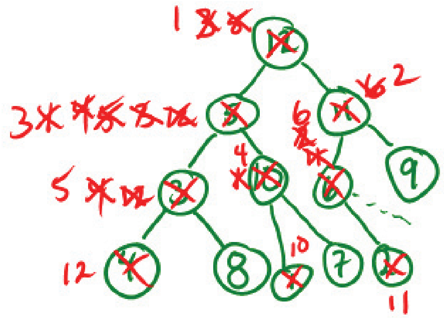
- Naïve solution: insert one at a time
- Running time: $\Theta(n \log n)$

$$\sum_{i=1}^{n} \log i \ge \sum_{i=n/2}^{n} \log i \ge \sum_{i=n/2}^{n} \log n/2 = \frac{n}{2} \cdot \log n/2$$

Can we do better with **buildHeap**?

Building a Heap: Take 1

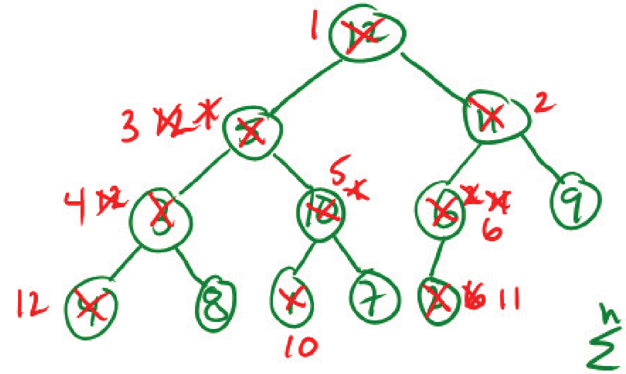
12	5	11	3	10	6	9	4	8	1	7	2



$O(n \log n)$

Building a Heap: Take 2

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



$O(n)$

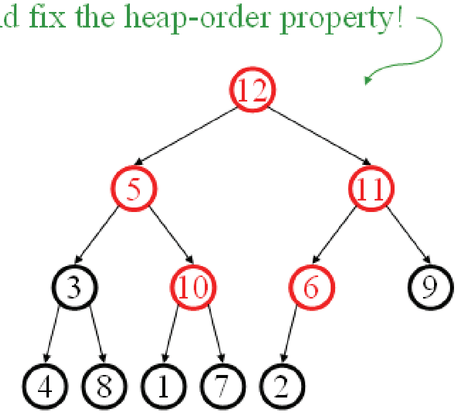
$$\sum_{i=0}^{h} 2^i (h-i) \sim O(n)$$

$h \sim \log_2 n$

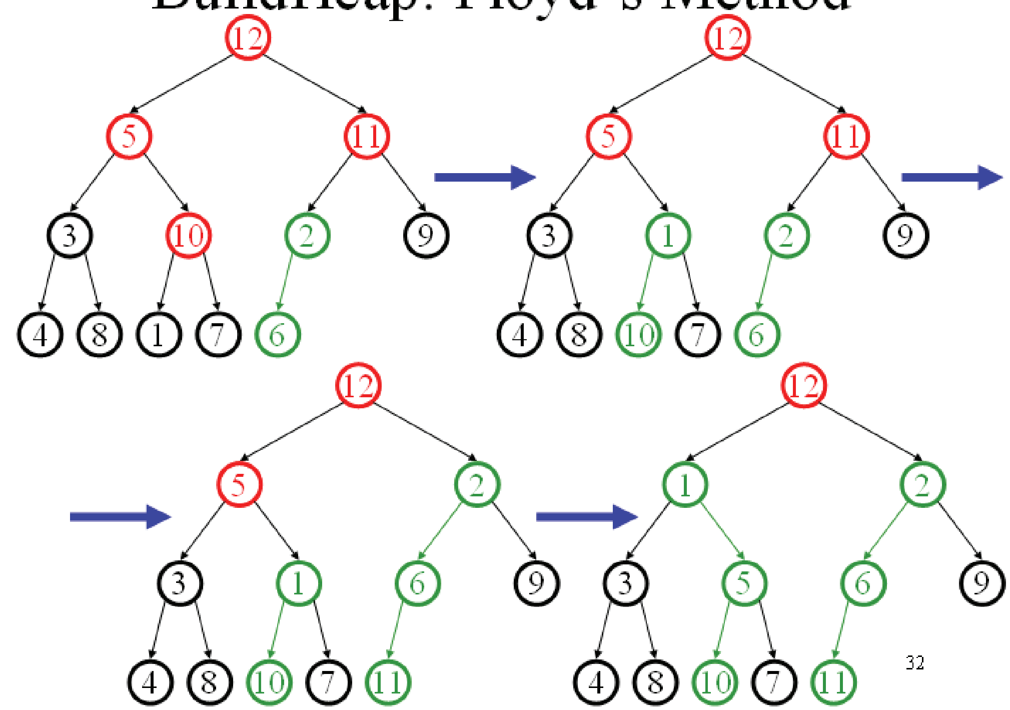
BuildHeap: Floyd's Method

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

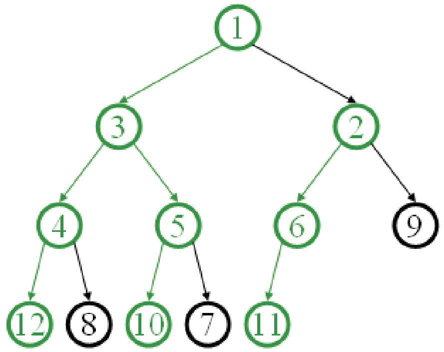
Add elements arbitrarily to form a complete tree.
Pretend it's a heap and fix the heap-order property!



BuildHeap: Floyd's Method



Finally…



Buildheap pseudocode

```
private void buildHeap() {
  for ( int i = currentSize/2; i > 0; i-- )
      percolateDown( i );
}
```

runtime: $O(n)$