CSE 326: Data Structures

Priority Queues – Binary Heaps

James Fogarty

Autumn 2007

Lecture 4

Administrative

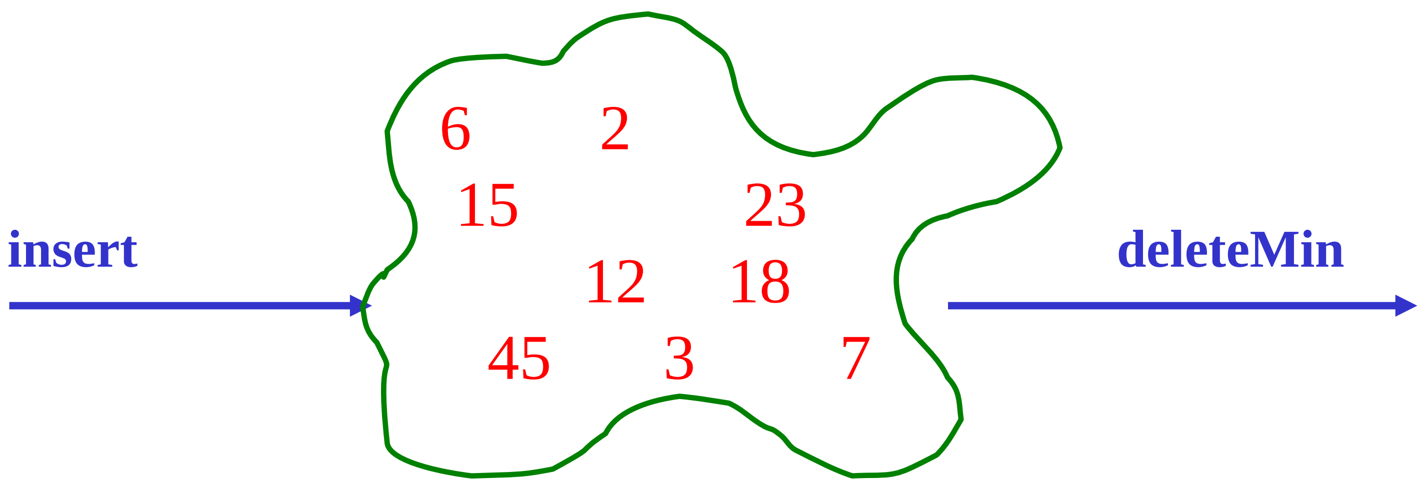
- HW1 due beginning of class Friday

Recall Queues

- FIFO: First-In, First-Out
- Some contexts where this seems right?
- Some contexts where some things should be allowed to skip ahead in the line?

Queues that Allow Line Jumping

- 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

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*

Applications of the Priority Queue

- Select print jobs in order of decreasing length
- Forward packets on routers in order of urgency
- Select most frequent symbols for compression
- Sort numbers, picking minimum first

- Anything ***greedy***

Potential Implementations

	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)

Recall From Lists, Queues, Stacks

- 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
- Heaps provide O(log n) worst case for both insert and deleteMin, O(1) average insert

Binary Heap Properties

1. Structure Property
2. Ordering Property

Tree Review

root(**T**):

leaves(**T**):

children(B):

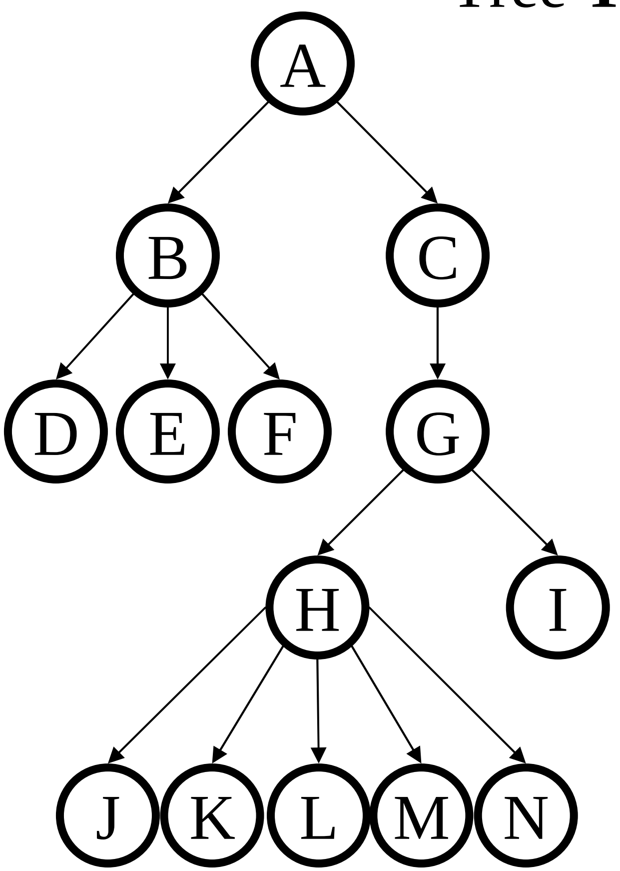
parent(H):

siblings(E):

ancestors(F):

descendents(G):

subtree(C):



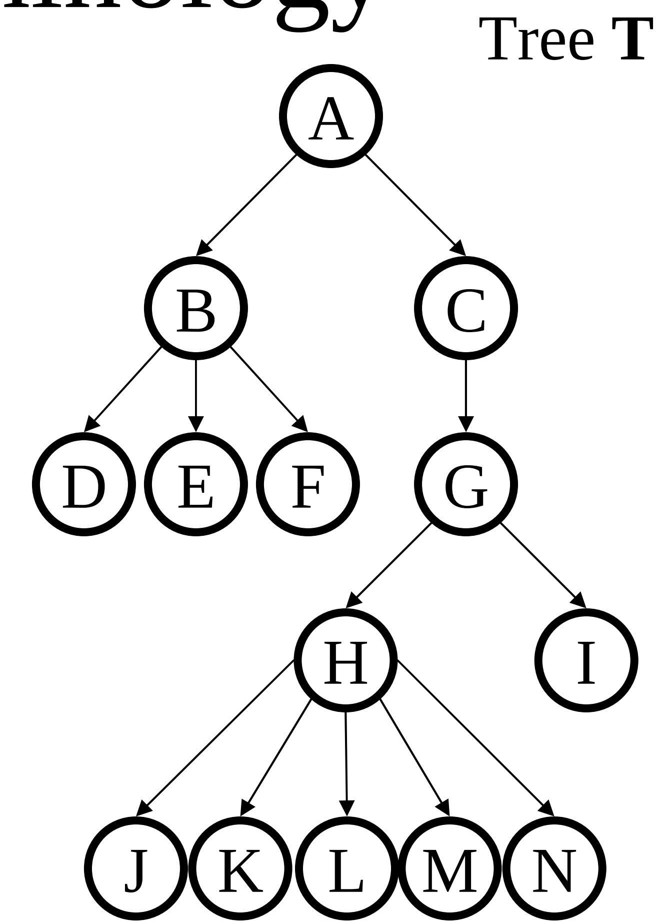
More Tree Terminology

depth(B):

height(G):

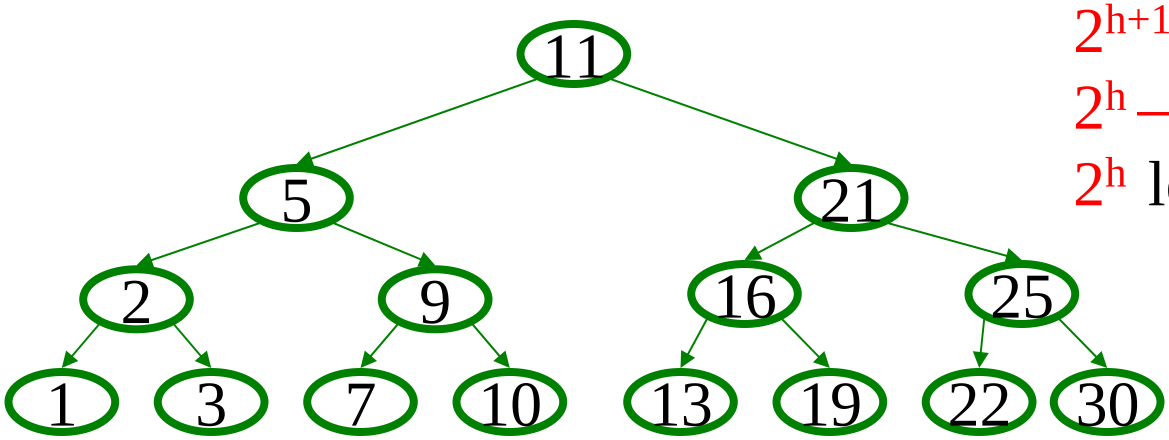
degree(B):

branching factor(**T**):



Brief interlude: Some Definitions:

A *Perfect* binary tree – A binary tree with all leaf nodes at the same depth. All internal nodes have 2 children.



height h

$2^{h+1} - 1$ nodes

$2^h - 1$ non-leaves

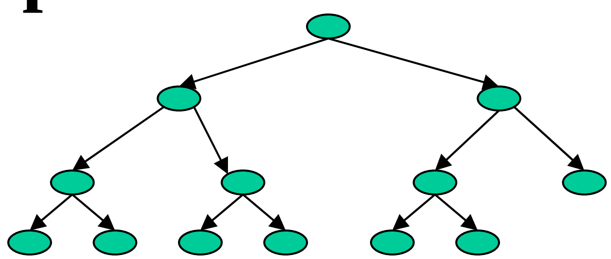
2^h leaves

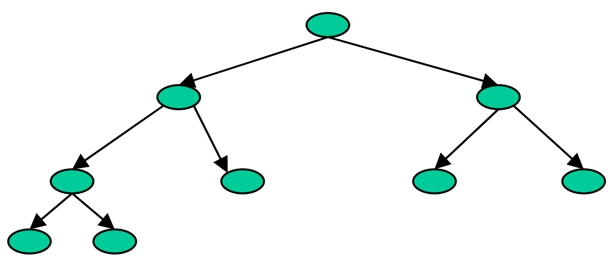
Heap **Structure** Property

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

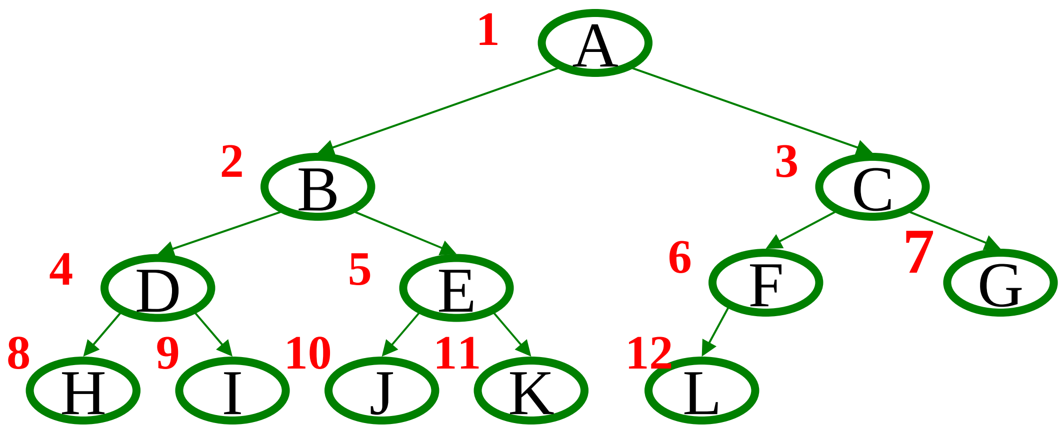
Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:





Representing Complete Binary Trees in an Array



From node **i**:

left child:
right child:
parent:

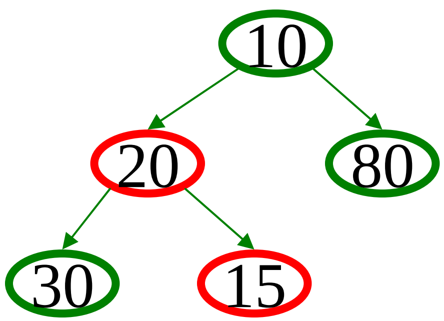
implicit (array) implementation:

	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

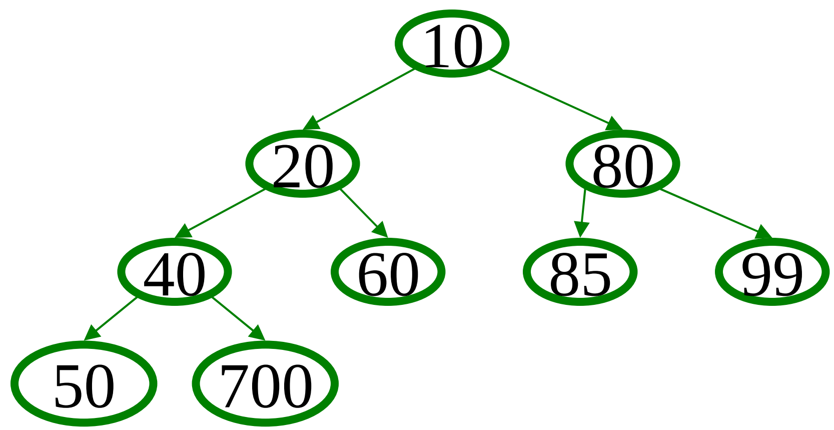
Why this approach to storage?

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.

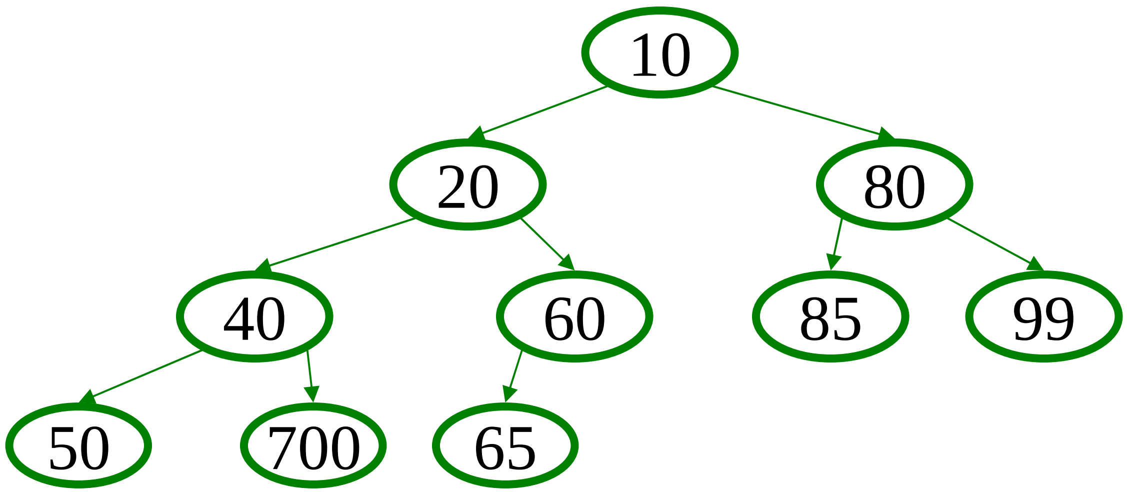


not a heap



Heap Operations

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

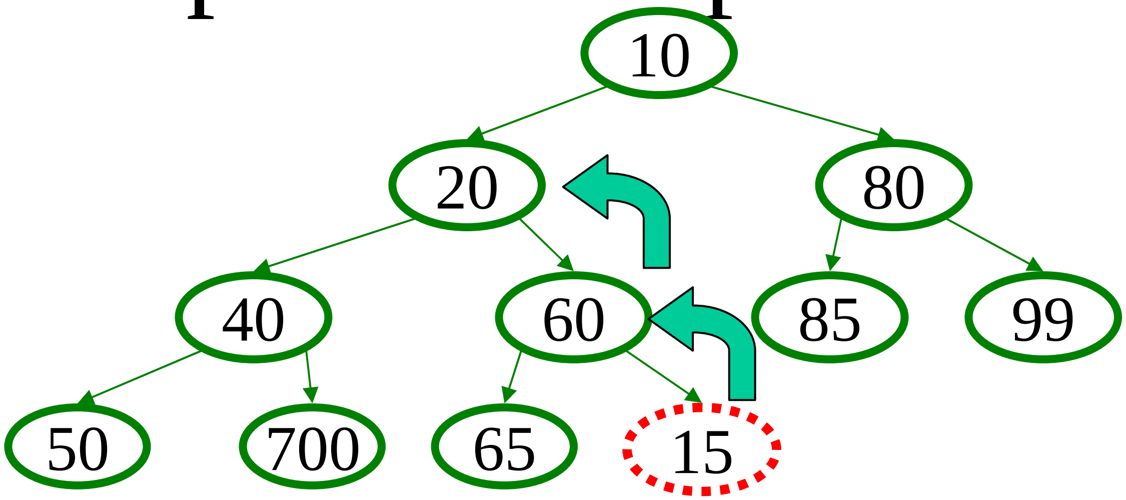


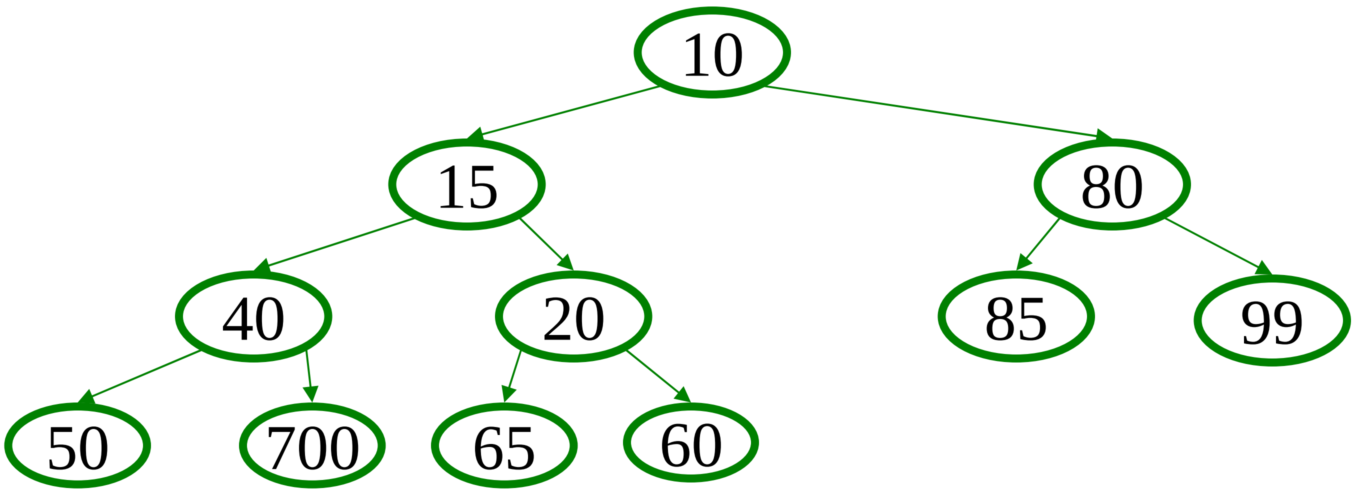
Heap – Insert(val)

Basic Idea:

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

Insert: percolate up





Insert 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:

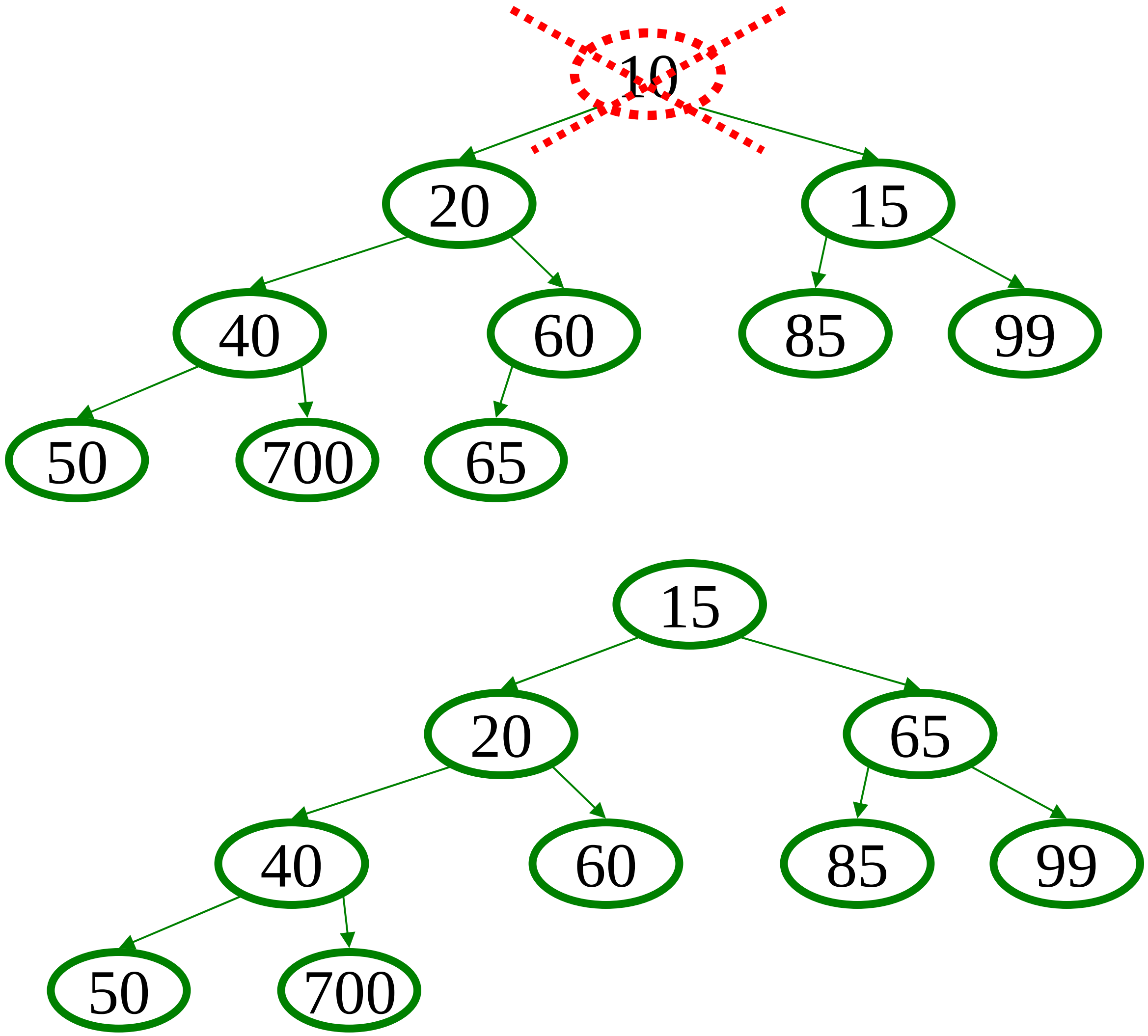
(Code in book)

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



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:

(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**