Administrative

• Turn in HW1
• HW2 available
• P1 Due next Wednesday
Priority Queue ADT

- Insert(v)
- DeleteMin()
Binary Heap data structure

• binary heap for priority queues:
  – $O(\log n)$ worst case for both insert and deleteMin

• Heap properties
  – Complete binary tree
  – Value of a node less than or equal to the values of its children
Insert: percolate up
DeleteMin: percolate down
Representing Complete Binary Trees in an Array

From node i:
left child:
right child:
parent:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
</tbody>
</table>
Why use an array?
DeleteMin Code

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

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;
}
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:

(Java code in book)
Insert: 16, 32, 4, 69, 105, 43, 2
More Priority Queue Operations

decreaseKey(nodePtr, amount):
given a pointer to a node in the queue, reduce its priority

Binary heap: change priority of node and ________________

increaseKey(nodePtr, amount):
given a pointer to a node in the queue, increase its priority

Binary heap: change priority of node and ________________

Why do we need a pointer? Why not simply data value?

Worst case running times?
More Priority Queue Operations

**remove(objPtr):**
- given a pointer to an object in the queue, remove it

  Binary heap: ______________________________________

**findMax( ):**
- Find the object with the highest value in the queue

  Binary heap: ______________________________________

Worst case running times?
More Binary Heap Operations

**expandHeap( ):**
If heap has used up array, copy to new, larger array.
  • Running time:

**buildHeap(objList):**
Given list of objects with priorities, fill the heap.
  • Running time:

We do better with `buildHeap`...
## Building a Heap: Take 1

<table>
<thead>
<tr>
<th>12</th>
<th>5</th>
<th>11</th>
<th>3</th>
<th>10</th>
<th>6</th>
<th>9</th>
<th>4</th>
<th>8</th>
<th>1</th>
<th>7</th>
<th>2</th>
</tr>
</thead>
</table>


BuildHeap: Floyd’s Method

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

Red nodes need to percolate down

Key idea: fix red nodes from bottom-up
BuildHeap: Floyd’s Method
Finally…

```
    1
   / \   \     \
3    2
   / \   / \\
4    5 6 9
 / \ / \ / \\
12 8 10 7 11
```
Buildheap pseudocode

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

runtime:
Buildheap Analysis

\[
\begin{align*}
\text{n/4 nodes percolate at most 1 level} \\
\text{n/8 percolate at most 2 levels} \\
\text{n/16 percolate at most 3 levels} \\
\cdots
\end{align*}
\]

runtime: