CSE373: Data Structures & Algorithms
Lecture 9: Binary Heaps, Continued

Kevin Quinn
Fall 2015
A priority queue is just an **abstraction** for an ordered queue.

A **binary heap** is a simple and concrete implementation of a priority queue.

It’s just one of many possible implementations!
Review

• **Priority Queue ADT**: insert comparable object, `deleteMin`
• **Binary heap data structure**: Complete binary tree where each node has priority value greater than its parent
• \(O(\text{height-of-tree}) = O(\log n)\) insert and `deleteMin` operations
  – `insert`: put at new last position in tree and percolate-up
  – `deleteMin`: remove root, put ‘last’ element at root and percolate-down
• But: tracking the “last position” is painful and we can do better
Array Representation of Binary Trees

Starting at node $i$

left child: $i \times 2$

right child: $i \times 2 + 1$

parent: $i / 2$

(wasting index 0 is convenient for the index arithmetic)

implicit (array) implementation:

<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></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>
</tr>
<tr>
<td></td>
<td>parent</td>
<td>i = 4</td>
<td></td>
<td>left</td>
<td></td>
<td>right</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
MAN, YOU'RE BEING INCONSISTENT WITH YOUR ARRAY INDICES. SOME ARE FROM ONE, SOME FROM ZERO.

DIFFERENT TASKS CALL FOR DIFFERENT CONVENTIONS. TO QUOTE STANFORD ALGORITHMS EXPERT DONALD KNUTH, "WHO ARE YOU? HOW DID YOU GET IN MY HOUSE?"

WAIT, WHAT?

WELL, THAT'S WHAT HE SAID WHEN I ASKED HIM ABOUT IT.

http://xkcd.com/163
Judging the array implementation

Positives:

• Non-data space is minimized: just index 0 and unused space on right
  – In conventional tree representation, one edge per node (except for root), so n-1 wasted space (like linked lists)
  – Array would waste more space if tree were not complete

• Multiplying and dividing by 2 is very fast (shift operations in hardware)

• Last used position is just index size

Negatives:

• Same might-by-empty or might-get-full problems we saw with stacks and queues (resize by doubling as necessary)

Plusses outweigh minuses: “this is how people do it”
Pseudocode: insert

```java
void insert(int val) {
    if (size == arr.length-1)
        resize();
    size++;
    i = percolateUp(size, val);
    arr[i] = val;
}

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

This pseudocode uses ints. In real use, you will have data nodes with priorities.

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>20</th>
<th>80</th>
<th>40</th>
<th>60</th>
<th>85</th>
<th>99</th>
<th>700</th>
<th>50</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</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>
<td>13</td>
</tr>
</tbody>
</table>
This pseudocode uses ints. In real use, you will have data nodes with priorities.

```c
int deleteMin() {  
    if(isEmpty()) throw...  
    ans = arr[1];  
    hole = percolateDown  
        (1,arr[size]);  
    arr[hole] = arr[size];  
    size--;  
    return ans;  
}
```

```c
int percolateDown(int hole,int val){  
    while(2*hole <= size) {  
        left = 2*hole;  
        right = left + 1;  
        if(arr[left] < arr[right]  
            || right > size)  
            target = left;  
        else  
            target = right;  
        if(arr[target] < val) {  
            arr[hole] = arr[target];  
            hole = target;  
        } else  
            break;  
    }  
    return hole;  
}
```

<table>
<thead>
<tr>
<th></th>
<th>10</th>
<th>20</th>
<th>80</th>
<th>40</th>
<th>60</th>
<th>85</th>
<th>99</th>
<th>700</th>
<th>50</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>
</tr>
<tr>
<td>1</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>20</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>80</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>60</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>85</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>700</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>13</td>
</tr>
</tbody>
</table>
Example

1. **insert**: 16, 32, 4, 69, 105, 43, 2
2. **deleteMin**

```plaintext
  0 1 2 3 4 5 6 7
```

```
  1
 / \   \
16 32 4
  / \   \
69 43 2
```

```plaintext
  0
 1
 2
 3
 4
 5
 6
 7
```
Example

1. **insert**: 16, 32, 4, 69, 105, 43, 2
2. **deleteMin**
Example

1. insert: 16, 32, 4, 69, 105, 43, 2
2. deleteMin
Example

1. **insert:** 16, 32, 4, 69, 105, 43, 2
2. **deleteMin**

```
   4
  / \
32   16
/     /
/     /
/     /
/     /
/     /
/     /
/     /
```

0 1 2 3 4 5 6 7
Example

1. **insert**: 16, 32, 4, 69, 105, 43, 2
2. **deleteMin**

```
  4   32   16   69
  0  1   2   3   4   5   6   7
```

```
  4
  / 
32  16
  / 
69  
```
Example

1. **insert**: 16, 32, 4, 69, 105, 43, 2
2. **deleteMin**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>4</td>
<td>32</td>
<td>16</td>
<td>69</td>
<td>105</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
        4
      /   \
    32    16
   /     /   \
  69    105    
```

Fall 2015 CSE373: Data Structures & Algorithms
Example

1. **insert**: 16, 32, 4, 69, 105, 43, 2
2. **deleteMin**

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>16</td>
<td>69</td>
<td>105</td>
<td>43</td>
</tr>
</tbody>
</table>

```
        4
       / \      
      32   16
     /  \  /   \
    69  105 43
```

Fall 2015  CSE373: Data Structures & Algorithms
Example

1. **insert**: 16, 32, 4, 69, 105, 43, 2
2. **deleteMin**

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

```
  2
 /  \
32   4
/   /  \
69  105  43  16
```
Other operations

- **decreaseKey**: given pointer to object in priority queue (e.g., its array index), lower its priority value. *Remember lower priority value is *better* (higher in tree).*
  - Change priority and percolate up

- **increaseKey**: given pointer to object in priority queue (e.g., its array index), raise its priority value.
  - Change priority and percolate down

- **remove**: given pointer to object in priority queue (e.g., its array index), remove it from the queue.
  - Percolate up to top and removeMin

Running time for all these operations?
Build Heap

• Suppose you have $n$ items to put in a new (empty) priority queue
  – Call this operation `buildHeap`

• $n$ distinct `inserts` works (slowly)
  – Only choice if ADT doesn’t provide `buildHeap` explicitly
  – $O(n \log n)$

• Why would an ADT provide this unnecessary operation?
  – Convenience
  – Efficiency: an $O(n)$ algorithm called Floyd’s Method
  – Common issue in ADT design: how many specialized operations
Floyd’s Method

1. Use \( n \) items to make any complete tree you want
   - That is, put them in array indices 1,\ldots,\( n \)

2. Treat it as a heap and fix the heap-order property
   - Bottom-up: leaves are already in heap order, work up toward the root one level at a time

```c
void buildHeap() {
    for (i = size/2; i > 0; i--) {
        val = arr[i];
        hole = percolateDown(i, val);
        arr[hole] = val;
    }
}
```
Example

• In tree form for readability
  – Purple for node not less than descendants
    • heap-order problem
  – Notice no leaves are purple
  – Check/fix each non-leaf bottom-up (6 steps here)
Example

Step 1

- Happens to already be less than children (\(e_r, \text{child}\))
Example

Step 2

- Percolate down (notice that moves 1 up)
Example

Step 3

- Another nothing-to-do step
- Percolate down as necessary (steps 4a and 4b)
Example
Example

Step 6

1
3
5
6
2
9
4
8
10
7
11

1
3
5
6
2
9
4
8
10
7
11

Fall 2015  
CSE373: Data Structures & Algorithms
But is it right?

• “Seems to work”
  – Let’s *prove* it restores the heap property (correctness)
  – Then let’s *prove* its running time (efficiency)

```c
void buildHeap() {
    for(i = size/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i,val);
        arr[hole] = val;
    }
}
```
Correctness

Loops Invariant: For all j > i, arr[j] is less than its children
- True initially: If j > size/2, then j is a leaf
  - Otherwise its left child would be at position > size
- True after one more iteration: loop body and percolateDown make arr[i] less than children without breaking the property for any descendants

So after the loop finishes, all nodes are less than their children
Efficiency

```cpp
void buildHeap() {
    for(i = size/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i,val);
        arr[hole] = val;
    }
}
```

Easy argument: buildHeap is $O(n \log n)$ where $n$ is size

- $\frac{size}{2}$ loop iterations
- Each iteration does one percolateDown, each is $O(\log n)$

This is correct, but there is a more precise ("tighter") analysis of the algorithm…
Better argument: `buildHeap` is $O(n)$ where $n$ is size

- $\text{size}/2$ total loop iterations: $O(n)$
- $1/2$ the loop iterations percolate at most 1 step
- $1/4$ the loop iterations percolate at most 2 steps
- $1/8$ the loop iterations percolate at most 3 steps
- ...
- $\left(\frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \frac{5}{32} + \ldots\right) < 2$ (page 4 of Weiss)
  - So at most $2(\text{size}/2)$ total percolate steps: $O(n)$

```c
void buildHeap() {
    for(i = size/2; i>0; i--) {
        val = arr[i];
        hole = percolateDown(i, val);
        arr[hole] = val;
    }
}
```
Lessons from buildHeap

• Without buildHeap, our ADT already let clients implement their own in $O(n \log n)$ worst case
  – Worst case is inserting better priority values later

• By providing a specialized operation internal to the data structure (with access to the internal data), we can do $O(n)$ worst case
  – Intuition: Most data is near a leaf, so better to percolate down

• Can analyze this algorithm for:
  – Correctness:
    • Non-trivial inductive proof using loop invariant
  – Efficiency:
    • First analysis easily proved it was $O(n \log n)$
    • Tighter analysis shows same algorithm is $O(n)$
What we are skipping

- **merge**: given two priority queues, make one priority queue
  - How might you merge binary heaps:
    - If one heap is much smaller than the other?
    - If both are about the same size?
  - Different pointer-based data structures for priority queues support logarithmic time **merge** operation (impossible with binary heaps)
    - Leftist heaps, skew heaps, binomial queues
    - Worse constant factors
    - Trade-offs!