CSE373: Data Structures & Algorithms
Lecture 9: Priority Queues and Binary Heaps

Catie Baker
Spring 2015
Announcements

• Homework 3 is out and due Wednesday April 29th at 11pm
• Office Hours are updated - Check the website
• Lauren will be teaching next week while Catie is out of town
Priority Queue ADT

- A priority queue holds *compare-able* items
- Each item in the priority queue has a “priority” and “data”
  - In our examples, the lesser item is the one with the greater priority
  - So “priority 1” is more important than “priority 4”

- Operations:
  - *insert*: *adds* an element to the priority queue
  - *deleteMin*: *returns* and *deletes* the item with greatest priority
  - *is_empty*

- Our data structure: A *binary min-heap* (or *binary heap* or *heap*) has:
  - Structure property: A *complete* binary tree
  - Heap property: The priority of every (non-root) node is less important than the priority of its parent (*Not a binary search tree*)
Operations: basic idea

- **deleteMin:**
  1. Remove root node
  2. Move right-most node in last row to root to restore structure property
  3. “Percolate down” to restore heap property

- **insert:**
  1. Put new node in next position on bottom row to restore structure property
  2. “Percolate up” to restore heap property

Overall strategy:
- *Preserve structure property*
- *Break and restore heap property*
DeleteMin

Delete (and later return) value at root node
DeleteMin: Keep the Structure Property

• We now have a “hole” at the root
  – Need to fill the hole with another value

• Keep structure property: When we are done, the tree will have one less node and must still be complete

• Pick the last node on the bottom row of the tree and move it to the “hole”
DeleteMin: Restore the Heap Property

Percolate down:
- Keep comparing priority of item with both children
- If priority is less important, swap with the most important child and go down one level
- Done if both children are less important than the item or we’ve reached a leaf node

Run time?
Runtime is $O(\text{height of heap})$  \(O(\log n)\)
Height of a complete binary tree of \(n\) nodes = \(\lceil \log_2(n) \rceil\)
**Insert**

- Add a value to the tree
- Afterwards, structure and heap properties must still be correct
**Insert: Maintain the Structure Property**

- There is only one valid tree shape after we add one more node.
- So put our new data there and then focus on restoring the heap property.
Insert: Restore the heap property

Percolate up:
• Put new data in new location
• If parent is less important, swap with parent, and continue
• Done if parent is more important than item or reached root

What is the running time?
Like deleteMin, worst-case time proportional to tree height: $O(\log n)$
**Array Representation of Binary Trees**

From 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>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>
Judging the array implementation

Plusses:
• Non-data space: 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 $\text{size}$

Minuses:
• Same might-be-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 into binary heap

```c
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.
Pseudocode: deleteMin from binary heap

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

int percolateDown(int hole, int val) {
    while(2*hole <= size) {
        left = 2*hole;
        right = left + 1;
        if(right > size ||
            arr[left] < arr[right])
            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></td>
<td>10</td>
<td>20</td>
<td>80</td>
<td>40</td>
<td>60</td>
<td>85</td>
<td>99</td>
<td>700</td>
<td>50</td>
</tr>
</tbody>
</table>

Spring 2015

CSE 373

14
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th>16</th>
<th></th>
</tr>
</thead>
</table>
0  | 1 | 2 | 3 | 4 | 5 | 6  | 7 |
```
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

```
<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>32</th>
<th>16</th>
<th>67</th>
<th></th>
<th></th>
<th></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>
</tr>
</tbody>
</table>
```

![Binary Search Tree Example](image-url)
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>32</th>
<th>16</th>
<th>67</th>
<th>105</th>
<th></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>
</tr>
</tbody>
</table>

```
        4
       /
      32  16
     /    /
   67    105
```

Spring 2015
CSE 373
20
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>32</th>
<th>16</th>
<th>67</th>
<th>105</th>
<th>43</th>
<th></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>
</tr>
</tbody>
</table>

```
        4
       /  
      32   16
     /     /  
    67    105 43
```

Spring 2015 CSE 373
Example

1. insert: 16, 32, 4, 67, 105, 43, 2
2. deleteMin

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>32</th>
<th>4</th>
<th>67</th>
<th>105</th>
<th>43</th>
<th>16</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>
</tr>
</tbody>
</table>

```
2
  /\  \\
32 4
  /\  /
67 105
  /\  /
43 16
```
Other operations

- **decreaseKey**: given pointer to object in priority queue (e.g., its array index), lower its priority value by \( p \)
  - Change priority and percolate up

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

- **remove**: given pointer to object in priority queue (e.g., its array index), remove it from the queue
  - **decreaseKey** with \( p = \infty \), then **deleteMin**

Running time for all these operations?
Build Heap

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

• $n$ inserts works
  – Only choice if ADT doesn’t provide \texttt{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,…,\( n \)

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

```java
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

- Happens to already be less than children (er, child)
Step 2

- Percolate down (notice that moves 1 up)
Example

- Another nothing-to-do step
Example

- Percolate down as necessary (steps 4a and 4b)
Example

Step 5
Example

Step 6
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

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

**Loop Invariant:** For all $j > i$, $\text{arr}[j]$ is less than its children

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

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

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

- $size/2$ 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`

- `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 `(size/2)` total percolate steps: $O(n)$
Lessons from `buildHeap`

- Without `buildHeap`, our ADT already let clients implement their own in $O(n \log n)$ worst case.
- 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 $O(n)$.
Other branching factors

- $d$-heaps: have $d$ children instead of 2
  - Makes heaps shallower, useful for heaps too big for memory (or cache)

- Homework: Implement a 3-heap
  - Just have three children instead of 2
  - Still use an array with all positions from 1…heap-size used

<table>
<thead>
<tr>
<th>Index</th>
<th>Children Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2,3,4</td>
</tr>
<tr>
<td>2</td>
<td>5,6,7</td>
</tr>
<tr>
<td>3</td>
<td>8,9,10</td>
</tr>
<tr>
<td>4</td>
<td>11,12,13</td>
</tr>
<tr>
<td>5</td>
<td>14,15,16</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>