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

Linda Shapiro
Spring 2016
A new ADT: Priority Queue

• A priority queue holds compare-able data

  – Like dictionaries, we need to compare items
    • Given x and y, is x less than, equal to, or greater than y
    • Meaning of the ordering can depend on your data

  – Integers are comparable, so will use them in examples
    • But the priority queue ADT is much more general
    • Typically two fields, the priority and the data
Priorities

• Each item has a “priority”
  – In our examples, the lesser item is the one with the greater priority
  – So “priority 1” is more important than “priority 4”
  – (Just a convention, think “first is best”)

• Operations:
  – insert
  – deleteMin
  – is_empty

• Key property: deleteMin returns and deletes the item with greatest priority (lowest priority value)
  – Can resolve ties arbitrarily
Example

```
insert x1 with priority 5
insert x2 with priority 3  \ (x1,5) \ (x2,3)
insert x3 with priority 4  \ (x1,5) \ (x3,4) \ (x2,3)
a = deleteMin   // x2  \ (x1,5) \ (x3,4)
b = deleteMin   // x3  \ (x1,5)
insert x4 with priority 2
insert x5 with priority 6
c = deleteMin   // x4
d = deleteMin   // x1
```

- Analogy: **insert** is like **enqueue**, **deleteMin** is like **dequeue**
  - But the whole point is to use priorities instead of FIFO
Applications

Like all good ADTs, the priority queue arises often
  – Sometimes blatant, sometimes less obvious

• Run multiple programs in the operating system
  – “critical” before “interactive” before “compute-intensive”
  – Maybe let users set priority level

• Treat hospital patients in order of severity (or triage)
• Select print jobs in order of decreasing length?
• Forward network packets in order of urgency
• Select most frequent symbols for data compression
• Sort (first \texttt{insert} all, then repeatedly \texttt{deleteMin})
Finding a good data structure

• Will show an efficient, non-obvious data structure for this ADT
  – But first let’s analyze some “obvious” ideas for $n$ data items
  – All times worst-case; assume arrays “have room”

<table>
<thead>
<tr>
<th>Data</th>
<th>Insert Algorithm / Time</th>
<th>DeleteMin Algorithm / Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsorted array</td>
<td>add at end</td>
<td>O(1) search</td>
</tr>
<tr>
<td>unsorted linked list</td>
<td>add at front</td>
<td>O(1) search</td>
</tr>
<tr>
<td>sorted circular array</td>
<td>search / shift</td>
<td>O(n) move front</td>
</tr>
<tr>
<td>sorted linked list</td>
<td>put in right place</td>
<td>O(n) remove at front</td>
</tr>
<tr>
<td>binary search tree</td>
<td>put in right place</td>
<td>O(n) leftmost</td>
</tr>
<tr>
<td>AVL tree</td>
<td>put in right place</td>
<td>O(\log n) leftmost</td>
</tr>
</tbody>
</table>
Our data structure: the Binary Heap

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

So:
- Where is the most important item?
- What is the height of a heap with $n$ items?
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

```
    1
   / \
  4   3
 / \
7   5 8 9
/ \   / \    \
11 9 6 10
```
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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>E</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>G</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>H</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>I</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>J</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>K</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>L</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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 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”
This pseudocode uses ints. In real use, you will have data nodes with priorities.

**Pseudocode: insert into binary heap**

```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;
}
```

```
10
 /   \
20   80
 / \
40 60
 / \
40 85 99
 /      \
700 50
```

|    10    20    80    40    60    85    99    700    50 |
|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|-----------|
| 0 1 2 3 4 5 6 7 8 9 10 11 12 13 |
This pseudocode uses ints. In real use, you will have data nodes with priorities.

```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;
}
```
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;
}
Pseudocode: deleteMin from binary heap
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
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>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
4

32

67

16

4
```
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>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 2016

CSE 373 Data Structures & Algorithms
Exercise

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

- \( n \) \texttt{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,\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)

12/2 = 6
Example

Step 1

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

- 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

12

1

3

4 8 10 7 11

5

2

6

9

3

4

12 8 10 7 11

5

6

2

9

10
But is it right?

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

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

```c
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 \text{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

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...
Efficiency

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

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
- ...
- \(((1/2) + (2/4) + (3/8) + (4/16) + (5/32) + \ldots) < 2\) (page 4 of Weiss)
  - So at most \(2*(\text{size}/2)\) 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 $O(n)$
Exercise: Build the Heap using Floyd’s method

3 | 9 | 2 | 6 | 25 | 1 | 80 | 35
1 2 3 4 5 6 7 8