CSE 332: Data Abstractions

Lecture 5: Binary Heaps, Continued

Ruth Anderson

Winter 2015
Announcements

• **Project 1** – phase B due Thurs via catalyst
• **Homework 1** – due Fri via catalyst
• **Homework 2** – due next Fri
Today

- Binary Min Heap implementation
  - Insert
  - Deletemin
  - Buildheap
Review

- Priority Queue ADT: \texttt{insert} comparable object, \texttt{deleteMin}
- Binary heap data structure: Complete binary tree where each node has priority value greater than its parent
- $O(\text{height-of-tree})=O(\log n)$ \texttt{insert} and \texttt{deleteMin} operations
  - \texttt{insert}: put at new last position in tree and percolate-up
  - \texttt{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

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>
<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>
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.
Pseudocode: insert

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

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

```cpp
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(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;
}
```

This tree represents the data structure used in the pseudocode.
Example

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

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

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

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>32</th>
<th>16</th>
<th>69</th>
<th>105</th>
<th>43</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>

![]
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?
Evaluating the Array Implementation…

Advantages:

**Minimal amount of wasted space:**
- Only index 0 and any unused space on right in the array
- No "holes" due to complete tree property
- No wasted space representing tree edges

**Fast lookups:**
- Benefit of array lookup speed
- Multiplying and dividing by 2 is extremely fast (can be done through bit shifting (see CSE 351))
- Last used position is easily found by using the PQQueue’s size for the index

Disadvantages:

- What if the array gets too full (or wastes space by being too empty)? Array will have to be resized.

**Advantages outweigh Disadvantages:** This is how it is done!
So why $O(1)$ average-case insert?

- Yes, insert's **worst case** is $O(\log n)$
- The trick is that it all depends on the order the items are inserted (What is the worst case order?)
- Experimental studies of randomly ordered inputs shows the following:
  - Average 2.607 comparisons per insert (# of percolation passes)
  - An element usually moves up 1.607 levels
- `deleteMin` is average $O(\log n)$
  - Moving a leaf to the root usually requires re-percolating that value back to the bottom
Aside: Insert run-time: Take 2

• Insert: Place in next spot, percUp
• How high do we expect it to go?
• Aside: Complete Binary Tree
  – Each full row has \(2x\) nodes of parent row
  – \(1+2+4+8+\ldots+2^k = 2^{k+1}-1\)
  – Bottom level has \(\sim1/2\) of all nodes
  – Second to bottom has \(\sim1/4\) of all nodes
• PercUp Intuition:
  – Move up if value is less than parent
  – Inserting a random value, likely to have value not near highest, nor lowest; somewhere in middle
  – Given a random distribution of values in the heap, bottom row should have the upper half of values, \(2^{nd}\) from bottom row, next \(1/4\)
  – Expect to only raise a level or 2, even if \(h\) is large
• Worst case: still \(O(\log n)\)
• Expected case: \(O(1)\)
• Of course, there’s no guarantee; it may percUp to the root
Building a Heap

Suppose you have $n$ items you want to put in a new priority queue
• A sequence of $n$ insert operations works
• Runtime?

Can we do better?
• If we only have access to insert and deleteMin operations, then NO.
• There is a faster way - O(n), but that requires the ADT to have a specialized buildHeap operation

Important issue in ADT design: how many specialized operations?
– Tradeoff: Convenience, Efficiency, Simplicity
Floyd’s buildHeap Method

Recall our general strategy for working with the heap:
- Preserve structure property
- Break and restore heap ordering property

Floyd’s buildHeap:
1. Create a complete tree by putting the n items in array indices 1, . . . , n
2. Treat the array as a heap and fix the heap-order property
   - Exactly how we do this is where we gain efficiency
Thinking about buildHeap

- Say we start with this array:
  \[12, 5, 11, 3, 10, 2, 9, 4, 8, 1, 7, 6\]

- To “fix” the ordering can we use:
  - percolateUp?
  - percolateDown?
Floyd’s buildHeap Method

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;
    }
}
```
**buildHeap Example**

- Say we start with this array:
  \[12, 5, 11, 3, 10, 2, 9, 4, 8, 1, 7, 6\]

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

- Happens to already be less than child
buildHeap Example

Step 2

- Percolate down (notice that moves 1 up)
**buildHeap Example**

- Another nothing-to-do step
**buildHeap Example**

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

![Binary Heap Diagram]

**Step 5**

1. **Example**: Build a binary heap from the given array.

2. **Algorithm**: The heap property must be maintained throughout the process.

3. **Implementation**: Use a recursive approach to ensure the heap property is preserved.

4. **Analysis**: Understand the time complexity of the buildheap operation.

5. **Visualization**: Use a diagram to illustrate the heap structure at each step.
buildHeap Example

Step 6

12

1

3

1

5

2

6

9

3

4

5

12

8

10

7

11

6

10

7

11

9

4

8

10

7

11

2
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

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, 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
**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
- 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

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

```java
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… etc.
- \( ((1/2) + (2/4) + (3/8) + (4/16) + (5/32) + \ldots) = 2 \) (page 4 of Weiss)
  - So at most \( 2 \cdot (\text{size}/2) \) total percolate steps: $O(n)$
  - Also see Weiss 6.3.4, sum of heights of nodes in a perfect tree
Lessons from `buildHeap`

- Without `buildHeap`, our ADT already let clients implement their own in $\theta(n \log n)$ worst case
  - Worst case is inserting lower priority values later
- By providing a specialized operation internally (with access to the data structure), 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 $O(n \log n)$
    - A “tighter” analysis shows same algorithm is $O(n)$
What we’re skipping (see text if curious)

• **d-heaps**: have $d$ children instead of 2 (Weiss 6.5)
  – Makes heaps shallower, useful for heaps too big for memory
  – How does this affect the asymptotic run-time (for small $d$’s)?
• **Leftist heaps, skew heaps, binomial queues** (Weiss 6.6-6.8)
  – Different data structures for priority queues that support a logarithmic time **merge** operation (impossible with binary heaps)
  – **merge**: given two priority queues, make one priority queue
  – Insert & deleteMin defined in terms of merge

Aside: How might you merge *binary* heaps:

• If one heap is much smaller than the other?
• If both are about the same size?