CSE332: Data Abstractions

Lecture 5: Binary Heaps, Continued

Tyler Robison

Summer 2010
Priority Queue ADT: **insert** comparable object, **deleteMin**

Binary heap data structure: Complete binary tree where each node has a lesser priority than its parent (greater value)

\[ O(\text{height-of-tree}) = O(\log n) \] \textit{insert} and \textit{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
Clever Trick: Array Representation of Complete Binary Trees

From node $i$:
- left child: $i \times 2$
- right child: $i \times 2 + 1$
- parent: $i / 2$

(wasting index 0 is convenient)

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>12</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>

We can use index 0 to store other info, such as the size
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
- For reasons you learn in CSE351 / CSE378, multiplying and dividing by 2 is very fast
- \text{size} \text{ is the index of the last node}

Minuses:

- Same might-be-empty or might-get-full problems we saw with array 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;
}
```

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

Note this pseudocode inserts ints, not useful data with priorities.

O(logn): Or is it…

<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>
Note this pseudocode deletes ints, not useful data with priorities

Pseudocode: deleteMin

```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;
}
```

O(log n)
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
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 \( O(\log n) \)

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

- **remove**: given pointer to object, take it out of the queue
  - **decreaseKey**: set to $-\infty$, then **deleteMin** \( O(\log n) \)

Running time for all these operations?
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 ~1/2 of all nodes
  - Second to bottom has ~1/4 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\textsuperscript{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
Suppose you started with \( n \) items to put in a new priority queue
- Call this the `buildHeap` operation

`create`, followed by \( n \) inserts works
- 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
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 by fixing the heap-order property**
   - Bottom-up: leaves are already in heap order, work up toward the root one level at a time

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

Example

- Say we start with 
  - $[12,5,11,3,10,2,9,4,8,1,7,6]$ 
- In tree form for readability
  - Red for node not less than descendants
    - Heap-order violation
  - Notice no leaves are red
  - Check/fix each non-leaf bottom-up (6 steps here)
Example

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
Example
But is it right?

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

```java
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, 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: Equivalent to the heap ordering property
Easy argument: \texttt{buildHeap} is $O(n \log n)$ where $n$ is size

- \texttt{size/2} loop iterations
- Each iteration does one \texttt{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 $2 \times \frac{\text{size}}{2}$ 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 $\theta(n \log n)$ 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
  - Makes heaps shallower
  - Approximate height of a complete $d$-ary tree with $n$ nodes?
  - How does this affect the asymptotic run-time (for small $d$’s)?
- Useful for huge tree data structures that are too large to fit in memory; accessing a node will require accessing the hard-drive (incredibly slow) – limit nodes accessed: B-Trees

Aside: How would we do a ‘merge’ for 2 binary heaps?
- Answer: Slowly; have to buildHeap; $O(n)$ time
  - Will always have to copy over data from one array
- Different data structures for priority queues that support a logarithmic time merge operation (impossible with binary heaps)
  - Leftist heaps, skew heaps, binomial queue: Insert & deleteMin defined in terms of merge
- Special case: How might you merge binary heaps if one heap is much smaller than the other?