CSE 373: Data Structures and Algorithms

Lecture 13: Finish Binary Heaps

Instructor: Lilian de Greef
Quarter: Summer 2017
Announcements

- **Midterm on Friday**
  - Will start at 10:50, will end promptly at 11:50 (even if you’re late), so be early
  - Anything we’ve covered is fair game (including this lecture)
  - Only bring pencils and erasers
  - Turn off / silence and put away any devices (e.g. phone) before exam

- **Section**
  - Will go over solutions for select problems from practice set
  - Practice set posted on course webpage (under Sections)
  - Recommendation: do the practice problems, then use section to go over the questions you found hardest (there isn’t enough time to cover all of them)

- **Homework 3 grades come out today!**
- **Course feedback today! (anonymous, confidential, something I have set up)**
Binary Trees Implemented with an Array

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)
Judging the array implementation

Pros:
• 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

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

Pros outweigh cons: min-heaps almost always use array implementation
Heap insert:

1. Put new data in new location *(preserve structure property)*
2. **Percolate up:** *(restore heap property)*
   - If higher priority than parent, swap with parent
   - Repeat until parent is more important or reached root
Semi-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;
}
```

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

1. Remove (and later return) item at root
2. “Move” the last item in bottom row to the root  *(preserve structure property)*
3. **Percolate down:** *(restore heap property)*
   - If item has lower priority, swap with the most important child
   - Repeat until both children have lower priority or we’ve reached a leaf node
Semi-Pseudocode: `deleteMin` from binary heap

```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(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;
}
```
Example

1. insert (in this order): 16, 32, 4, 67, 105, 43, 2
2. deleteMin once

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

1. insert (in this order): 16, 32, 4, 67, 105, 43, 2
2. deleteMin once

```
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
  - \( \text{decreaseKey with } p = \infty, \text{ then deleteMin} \)

Running time for all these operations?
Build Heap

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

- $n$ inserts
  - Only choice if ADT doesn’t provide `buildHeap` explicitly
  - Run time:

- Why would an ADT provide this unnecessary operation?
  - Convenience
  - Efficiency: an $O(n)$ algorithm
  - Common issue in ADT design: how many specialized operations
heapify (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. Fix the heap-order property
   - Bottom-up: percolate down starting at nodes one level up from leaves, work up toward the root
**heapify (Floyd’s Method): Example**

1. Use $n$ items to make any complete tree you want
2. Fix the heap-order property from bottom-up

Which nodes break the heap-order property?

Why work from the bottom-up to fix them?

Why start at one level above the leaf nodes?

Where do we start here?
heapify (Floyd’s Method): Example
heapify (Floyd’s Method): Example
heapify (Floyd’s Method)

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

But is it right? ... it “seems to work”
- Let’s *prove* it restores the heap property
- Then let’s *prove* its running time
Correctness

Loop Invariant: For all $j > i$, arr[$j$] is higher priority than its children

- True initially: If $j > \frac{\text{size}}{2}$, 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 arr[$i$] higher priority than children without breaking the property for any descendants

So after the loop finishes, all nodes are less than their children
Easier argument: `buildHeap()` is where \( n \) is size loop iterations

- Each iteration does one `percolateDown`, each is

This is correct, but there is a more precise ("tighter") analysis of the algorithm...
Efficiency

Better argument: buildHeap is where $n$ is size

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

$$\sum_{i=1}^{\infty} \frac{i}{2^i} = 2$$

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

- **Without providing** `buildHeap`, clients can implement their own that runs in **worst case**

- By providing a specialized operation (with access to the internal data), we can do **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 (easier) analysis proved it was $O(n \log n)$
    - Tighter analysis shows same algorithm is $O(n)$
Other branching factors for Heaps

*d-heaps*: have $d$ children instead of 2
- Makes heaps shallower

Example: 3-heap
- Only difference: three children instead of 2
- Still use an array with all positions from 1 ... heapSize

<table>
<thead>
<tr>
<th>Index</th>
<th>Children Indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Wrapping up Heaps

• What are heaps a data structure for?

• What is it usually implemented with? Why?

• What are some example uses?