Today

- Priority Queues, Ch 6, 6.1-6.3
- Binary Min Heap implementation

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 1:
- left child: \(i \times 2\)
- right child: \(i \times 2 + 1\)
- Parent: \(i / 2\)

implicit (array) implementation:

\[
\begin{array}{cccccccccccc}
A & B & C & D & E & F & G & H & I & J & K & L \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\end{array}
\]
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
• 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

```java
void insert(int val) {
    if (size == arr.length-1)
        resize();
    size++;
    i = percolateUp(size, val);
    arr[i] = val;
}
```

Pseudocode: deleteMin

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

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
- **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, take it out of the queue
  - decreaseKey with \( p = \infty \), then deleteMin

Running time for all these operations?

### Insert run-time: Take 2

- **Insert**: Place in next spot, percolate
  - How high do we expect it to go?
- **Aside**: Complete Binary Tree
  - Each full row has \( 2^k \) nodes of parent row
  - \( 1+2+4+8+...+2^k = 2^{k+1} - 1 \)
  - Bottom level has \(~1/2 of all nodes\)
  - Second to bottom has \(~1/4 of all nodes\)
- **Percolate 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 \( \Theta(\log n) \)
- **Expected case**: \( O(1) \)
- Of course, there’s no guarantee; it may percolate up to the root

### 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 by fixing the heap-order property
   - Bottom-up: leaves are already in heap order, work up toward the root one level at a time

```plaintext
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 problem
      - Notice no leaves are red
  - Check/fix each non-leaf
    - bottom-up (6 steps here)

- Happens to already be less than children (or, child)
Step 2
- Percolate down (notice that moves 1 up)

Step 3
- Another nothing-to-do step

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

Step 5

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

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

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

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...

Lessons from buildHeap

- Without buildHeap, our ADT already lets 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?