CS 310: Heapify and Heap Sort

Chris Kauffman

Week 15-1
Logistics

Schedule

- Mon 12/7: Heapify and Heap Sort
- Wed 12/9: Review, Evals
- Fri 12/11: Codefest!
- Sat 12/12: HW 4 Due
- Mon 12/14: Final Exams

Goals Today

- HW4 Discussion
- Heapsort

Reading: Weiss

21: Priority Queue/Binary Heap
HW 4: TripleStore

- **add(e, r, p):** $O(\log N)$
- **query(e, r, p):** $O(K + \log N)$
- **remove(e, r, p):** $O(K \times \log N)$

where
- $N$: number of records in DB
- $K$: number of matching records

Note:
- Correctness is easy for these methods
- Correctness + Target Complexity is trickier
Binary Heaps

- How are they different from binary trees?
- What are the efficient operations a binary heap supports?
- How efficient are they?
- What is a common use of binary heaps?
  - Hint: facilitates cutting in line...
// Binary Heap, 1-indexed
public class BinaryHeapPQ<T>{
    private T [] array;
    private int size;

    // Helpers
    static int root(){
        return 1;
    }
    static int left(int i){
        return i*2;
    }
    static int right(int i){
        return i*2+1;
    }
    static int parent(int i){
        return i / 2;
    }

    // Insert a data
    public void insert(T x){
        size++;
        ensureCapacity(size+1);
        array[size] = x;
        percolateUp(size);
    }

    // Remove the minimum element
    public void deleteMin(){
        array[root()] = array[size];
        array[size] = null;
        size--;
        percolateDown(root());
    }
}
void percolateUp(int xdx) {
    while (xdx != root()) {
        T x = array[xdx];
        T p = array[parent(xdx)];
        if (doCompare(x, p) < 0) {
            array[xdx] = p;
            array[parent(xdx)] = x;
            xdx = parent(xdx);
        } else { break; }
    }
}

void percolateDown(int xdx) {
    while (true) {
        T x = array[xdx];
        int cdx = left(xdx);
        // Determine which child
        // if any to swap with
        if (cdx > size) { break; } // No left, bottom
        if (right(xdx) < size && // Right valid
            doCompare(array[right(xdx)], array[cdx])
            cdx = right(xdx); // Right smaller
        }
        T child = array[cdx];
        if (doCompare(child, x) < 0) { // child smaller
            array[cdx] = x; // swap
            array[xdx] = child;
            xdx = cdx; // reset index
        } else { break; }
    }
}
Height Again...

Efficiency of Binary Heap PQs

- `findMin()` clearly $O(1)$
- `deleteMin()` worst case height
- `insert(x)` worst case height

Complete Binary Tree

- Min number of nodes in a complete tree of height $H$?
- Take a crack at it
- Derive a rough formula
Weiss’s Assertion: insert is $O(1)$

On average the percolation [up] terminates early: It has been shown that 2.6 comparisons are required on average to perform the add, so the average add moves an element up 1.6 levels.

- Weiss 4th ed pg 815

Precise results on the number of comparisons and data movements used by heapsort in the best, worst, and average case are given in (Schaffer and Sedgewick).

- pg 839

And how rumors resolve

Binary Heaps DO have $O(1)$ amortized insertion

Empirical Evaluation

Mathematical Proof

- Schaffer/Sedgewick (1991)
- → Carlsson (1987)
- → Porter/Simon (1974): "note pg. 15 in which the average case is stated to be upper-bound by lambda, which computes to 1.606691524"

Students in Fall 2013 did the above, edited Wikipedia to reflect this fact. You should have been there. It was awesome.
Binary Heaps

Great for building Priority Queues:

<table>
<thead>
<tr>
<th>Op</th>
<th>Worst Case</th>
<th>Avg Case</th>
</tr>
</thead>
<tbody>
<tr>
<td>findMin()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert(x)</td>
<td>$O(\log N)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>deleteMin()</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

What else can one do with binary heaps?
Exercise: Sort data using a Binary Heaps/PQ

- Sort an array of stuff with a PQ/Binary heap
- Define the following method

// Sort an array in place using a priority queue
class with static function heapSort<T extends Comparable>(data[]):
  void heapSort(T data[]):
    ...;

- T is Comparable
- data unsorted
- return: nothing, but data must be sorted
- Use pq.insert(x) and pq.deleteMin()
Out of Place Sort Sucks

Initial solution required data duplication
  - Copy from data to pq, then back
  - Out of place sorting, double memory requirement
For large arrays this hurts
  - Want truly in place sorting
  - Don’t make copy, avoid $O(N)$ space overhead
  - Suggest a solution: no duplication possible?
In-Place Sorting with Heaps: Three Problems

1. 1-indexing of heaps
   - Not a problem: heap root at index 0
   - Formulas for left(i), right(i), parent(i)?

2. Broken Heap
   - Can’t start with an empty heap
   - Start with unsorted array
   - Heap property doesn’t hold
   - How to fix it?

3. Pulling Stuff Out
   - Have a heap, repeatedly pull out min
   - Where to put it?
Heapify, a.k.a. buildHeap()

Converts an existing array into a heap (!)

```java
public void buildHeap() {
    for (int i = parent(this.size); i >= root(); i-- ){
        this.percolateDown( i );
    }
}

Build the heap bottom up, repeated percolateDown(i)
  ▶ Start one level above bottom (where for size N heap?)
  ▶ Work right to left, low to high
  ▶ If small guy is down low, will bubble up
Heapify Example

Level 3: Initial, percolateDown([63])

Level 3: percolateDown([45]), percolateDown([12]),
Heapify Example

Level 3: \texttt{percolateDown([20])}, Level 2: \texttt{percolateDown([21])},

Level 2: \texttt{percolateDown([47])}, Level 1: \texttt{percolateDown([92])},
How complex is Heapify / buildHeap()?

- Discuss with a neighbor

Try

- How many small moves versus big moves?
- Derive an expression for the worst possible number of moves
Complexity of Heapify

Heap size $n$, height $h$, assume complete (?)

Measure level from bottom

- Level 1 is bottom, has $2^{h-1}$ nodes,
- Level 2 is second from bottom, has $2^{h-2}$ nodes
- Level $i$ is $i$th from bottom, has $2^{h-i}$ nodes
- Level $h$ is root, has $2^{h-h} = 1$ node

Each level $i$ node can move $i$ down so

$$
\text{moves} = \sum_{i=1}^{h} i \times 2^{h-i} = \sum_{i=1}^{\log_2 n} i \times 2^{\log_2 n-i} \\
= \sum_{i=1}^{\log_2 n} i \times \frac{2^{\log_2 n}}{2^i} = n \sum_{i=1}^{\log_2 n} \frac{i}{2^i} \\
\leq n \times 2 = O(n)
$$

because

$$
\sum_{i=1}^{\infty} \frac{i}{2^i} \to 2
$$
In Place Heap Sort

After heapifying an array...

Space Available

- Remove an element from a heap
- Now open space at end of array (percolate down)
- Put the removed element at end of array
- Repeat until empty

Min and Max Heap

Above process orders the array

- For Min-Heap will result in biggest to smallest
- For Max-Heap will result in smallest to biggest
Summary of Heap Sort

Input: Array a
Output: a is sorted

Build Max Heap on a
for i=0 to length-1 a
  tmp = findMax(a)
  removeMax(a)
  a[length-i-1] = tmp
done