CSE 326: Data Structures

Priority Queues – Binary Heaps

Steve Seitz
Winter 2009

Administrative

- P1 due Wednesday
  - Electronic submission by end of Wednesday
- HW1 due beginning of class Friday
- Reading for this week: Chapter 6.
  - Optional (can skip): leftist, skew heaps

Recall Queues

- FIFO: First-In, First-Out
  - Print jobs
  - File serving
  - Phone calls and operators
  - Lines at the Department of Licensing...

Priority Queues

Often, we want to prioritize who goes first—a priority queue:

- Select print jobs in order of decreasing length
- Forward packets on routers in order of urgency
- Operating system can favor jobs of shorter duration or those tagged as having higher importance
- Greedy optimization: “best first” problem solving
Priority Queue ADT

- Need a new ADT
- Operations: Insert an Item, Remove the “Best” Item

Potential implementations

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>deleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list (Array)</td>
<td>$O(1)/O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Unsorted list (Linked-List)</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted list (Array)</td>
<td>$O(n)$ to find, $O(n)$ to delete</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Sorted list (Linked-List)</td>
<td>$O(n)$ to find, $O(n)$ to insert</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Search Tree (BST)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
</tbody>
</table>

Choosing the Right ADT

- Use an ADT that corresponds to your needs
- The right ADT is efficient, while an overly general ADT provides functionality you aren’t using, but are paying for anyways
- Today we look at using a **binary heap** (a kind of binary tree) for priority queues:
  - $O(\log n)$ worst case for both insert and deleteMin
  - $O(1)$ average insert
- What if priority is equal to seconds since creation of priority queue?
Tree Review

- root(T): A
- leaves(T): D, F, I, N
- children(B): D, E, F
- parent(H): G
- siblings(E): D, F
- ancestors(F): A, B
- descendents(G): H - N
- subtree(C):

More Tree Terminology

- depth(B): 1
- height(G): 2
- height(T): = height(A) = 4
- degree(B): 3
- children branching factor(T): 5
- max #children of any node: n-ary tree

Heap Structure Property

- A binary heap is a complete binary tree.

Complete binary tree – binary tree that is completely filled, with the possible exception of the bottom level, which is filled left to right.

Examples:

Height of a complete binary tree with n nodes?
Representing Complete Binary Trees in an Array

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

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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</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>
<tr>
<td>2</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>
<tr>
<td>3</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>
<tr>
<td>4</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>
<tr>
<td>5</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>
<tr>
<td>6</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>
<tr>
<td>7</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>
<tr>
<td>8</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>
<tr>
<td>9</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>
<tr>
<td>10</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>
<tr>
<td>11</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>
<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>
<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>

Why this approach to storage?
- Faster access
- Less space
- Better memo- locality

Heap Order Property

**Heap order property:** For every non-root node X, the value in the parent of X is less than (or equal to) the value in X.

not a heap

Heap Operations
- **findMin:** at root
- **insert(val):** percolate up.
- **deleteMin:** percolate down.
Modifying Heaps

- Main ops: insert, deleteMin
- Key is to maintain
  - Structure Property
  - Order Property

- Basic idea is to propagate changes up/down the tree, fixing order as we go

Heap – insert(val)

Basic Idea:
1. Put val at “next” leaf position
2. Percolate up by repeatedly exchanging node until no longer needed

Insert: percolate up

```
void insert(Object o) {
    assert(!isFull());
    size++;
    newPos =
    percolateUp(size, o);
    Heap[newPos] = o;
}
```

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

\[
\text{worst} \leq O(\log n) \Rightarrow O(\log n)
\]

\[
\text{runtime: on avg.} \sim 1.67 \text{ levels} \Rightarrow O(1)
\]

(Java code in book)
Heap – deleteMin

Basic Idea:
1. Remove root (that is always the min!)
2. Put “last” leaf node at root
3. Find smallest child of node
4. Swap node with its smallest child if needed.
5. Repeat steps 3 & 4 until no swaps needed.

DeleteMin: percolate down

DeleteMin Code (Optimized)

```java
Object deleteMin() {
    assert(!isEmpty());
    returnVal = Heap[1];
    size--;
    newPos =
    percolateDown(1,
                  Heap[size+1]);
    Heap[newPos] =
    Heap[size + 1];
    return returnVal;
}

int percolateDown(int hole, Object val) {
    while (2*hole <= size) {
        left = 2*hole;
        right = left + 1;
        if (right <= size &&
            Heap[right] < Heap[left])
            target = right;
        else
            target = left;
        if (Heap[target] < val) {
            Heap[hole] = Heap[target];
            hole = target;
        } else
            break;
    }
    return hole;
}
```
More Priority Queue Operations

**remove(objPtr):**
given a pointer to an object in the queue, remove it \( \mathcal{O}(\log n) \)

Binary heap: \( \text{decrease priority}(objPtr, \infty) \), delete

**findMax():**
Find the object with the highest value in the queue

Binary heap: \( \text{search leaves} \) \( \mathcal{O}(n) \)

Worst case running times?
Building a Heap: Take 1

Building a Heap: Take 2

BuildHeap: Floyd’s Method

Add elements arbitrarily to form a complete tree. Pretend it’s a heap and fix the heap-order property!

Red nodes need to percolate down
Finally…

```
private void buildHeap() {
    for (int i = currentSize/2; i > 0; i--) {
        percolateDown(i);
    }
}
```

runtime: