CSE 326: Data Structures

Priority Queues – Binary Heaps

Eric McCambridge
Spring 2009
Lecture 5

Algorithms to find these things in graphs...

For graphs with $m$ edges, $n$ nodes:
- Simple: $O(mn^2)$
- Cooper, et al.: $O(mn^2)$ – better constants & space
- Leighton & Tarjan
  - Standard: $O(m \log \log n)$
  - Simple: $O(m \log_2 \log \log n)$
- Georgiadis: $O(n^2)$

Perspective: Kinds of Analysis

- Running time may depend on actual data input, not just length of input
- Distinguish
  - Worst Case
  - Amortized
  - Average Case
  - Best Case
- You choose input

Types of Analysis

Two orthogonal axes:
- Bound Flavor
  - Upper bound ($O$, $o$)
  - Lower bound ($\Omega$, $\omega$)
  - Asymptotically tight ($\Theta$)
- Analysis Case
  - Worst Case (Adversary)
  - Average Case
  - Best Case
  - Amortized

Administrative

- P1 due tonight
  - Electronic submission by midnight
- HW1 due beginning of class Friday
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

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

Priority Queue ADT

1. PQueue data: collection of data with priority
2. PQueue operations
   - insert
   - deleteMin
3. PQueue property: for two elements in the queue, x and y, if x has a lower priority value than y, x will be deleted before y

Applications of the Priority Queue

- Select print jobs in order of decreasing length
- Forward packets on routers in order of urgency
- Select most frequent symbols for compression
- Sort numbers, picking minimum first
- Anything greedy

Potential Implementations

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>deleteMin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted list (Array)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Unsorted list (Linked-List)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted list (Array)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Sorted list (Linked-List)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Recall From Lists, Queues, Stacks

- 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

More Tree Terminology

- depth(B): 1
- height(G): 2
- degree(B): 3
- branching factor(T): 5

Binary Heap Properties

A binary heap is a binary tree with two important properties that make it a good choice for priority queues:

1. Structure Property
2. Ordering Property

Note: we will sometimes refer to a binary heap as simply a “heap”.

Brief interlude: Some Definitions:

A Perfect binary tree – A binary tree with all leaf nodes at the same depth. All internal nodes have 2 children.

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:
Representing Complete Binary Trees in an Array

From node i:
- left child: \(2i + 1\)
- right child: \(2i + 2\)
- 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?

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:** return \(\{1\}\)
- **insert(val):** percolate up.
- **deleteMin:** percolate down.

Heap – Insert(val)

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

Average = 1.67

Insert: percolate up
Insert Code (optimized)

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

```
runtime: $O(\log n)
```

DeleteMin: percolate down

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

```
runtime: $O(\log n)$
```

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 smallest child of node if needed.
5. Repeat steps 3 & 4 until no swaps needed.

DeleteMin Code (Optimized)

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

```
runtime: $O(\log n)$
```

More Priority Queue Operations

decreaseKey(objPtr, amount):
given a pointer to an object in the queue, reduce its priority value by amount

Binary heap: change priority of node and percolate up

increaseKey(objPtr, amount):
given a pointer to an object in the queue, increase its priority value by amount

Binary heap: change priority of node and percolate down
More Priority Queue Operations

**remove(objPtr):**
Given a pointer to an object in the queue, remove it.

**Binary heap:**
- `decrease_key(i, val)`
- `delete_min` in $O(\log n)$

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

**Binary heap:** `search O(n)`

---

More Binary Heap Operations

**expandHeap():**
If heap has used up array, copy to new, larger array.
- Running time

**buildHeap(objList):**
Given list of objects with priorities, fill the heap.
- Naive solution.
- Running time

Can we do better with `buildHeap`?

---

BuildHeap: Floyd’s Method

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

Red nodes need to percolate down

---

BuildHeap pseudocode

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

---

Finally...