CSE 326: Data Structures
Lecture #13
Priority Queues and Binary Heaps
Nick Deibel
Winter Quarter 2002

Priority Queue ADT

- Priority Queue operations
  - create
  - destroy
  - insert
  - deleteMin
  - isEmpty
- Priority Queue 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

Not Quite Queues

- Consider applications
  - ordering CPU jobs
  - searching for the exit in a maze
  - emergency room admission processing
- Problems?
  - short jobs should go first
  - most promising nodes should be searched first
  - most urgent cases should go first

Applications of the Priority Q

- Hold jobs for a printer in order of length
- Store packets on network routers in order of urgency
- Sort numbers
- Simulate events
- Anything greedy

Discrete Event Simulation

- An event is a pair (x,t) where x describes the event and t is time it should occur
- A discrete event simulator (DES) maintains a set S of events which it intends to simulate in time order
repeat
  Find and remove (x_0,t_0) from S such that t_0 minimal;
  Do whatever x_0 says to do, in the process new events (x_1,t_1)…(x_k,t_k) may be generated;
  Insert the new events into S;

Emergency Room Simulation

- Two priority queues: time and criticality
- K doctors work in an emergency room
- events:
  - patients arrive with injury of criticality C at time t (according to some probability distribution)
  - Processing if no patients waiting and a free doctor, assign them to doctor and create a future departure event, else put patient in the Criticality priority queue
  - patient departs
    - If someone in Criticality queue, pull out most critical and assign to doctor
  - How long will a patient have to wait? Will people die?
Naïve Priority Queue Data Structures

- Unsorted list:
  - insert:
  - deleteMin:
- Sorted list:
  - insert:
  - deleteMin:

BST Tree Priority Queue Data Structure

- Regular BST:
  - insert:
  - deleteMin:
- AVL Tree:
  - insert:
  - deleteMin: Can we do better?

Binary Heap Priority Q Data Structure

- Heap-order property
  - parent’s key is less than children’s keys
  - result: minimum is always at the top
- Structure property
  - complete tree with fringe nodes packed to the left
  - result: depth is always $O(\log n)$; next open location always known

Nifty Storage Trick

- Calculations:
  - child:
  - parent:
  - root:
  - next free:

```
0 1 2 3 4 5 6 7 8 9 10 11 12
12 2 4 5 7 6 10 8 11 9 12 14 20
```

DeleteMin

```
pqueue.deleteMin();
```

Percolate Down
Finally…

DeleteMin Code

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

runtime:

```
I n s e r t □ C o d e
``` 

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

runtime:

```
P e r c o l a t e □ U p
```

Performance of Binary Heap

<table>
<thead>
<tr>
<th></th>
<th>Binary heap worst case</th>
<th>Binary heap avg case</th>
<th>AVL tree worst case</th>
<th>AVL tree avg case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>O(log n)</td>
<td>O(1)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Delete Min</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
</tbody>
</table>

- In practice: binary heaps much simpler to code, lower constant factor overhead
Changing Priorities

• In many applications the priority of an object in a priority queue may change over time
  – if a job has been sitting in the printer queue for a long time increase its priority
  – unix “renice”

• Must have some (separate) way of finding the position in the queue of the object to change (e.g. a hash table)

Other Priority Queue Operations

• decreaseKey
  – given the position of an object in the queue, reduce its priority value

• increaseKey
  – given the position of an object in the queue, increase its priority value

• remove
  – given the position of an object in the queue, remove it

• buildHeap
  – given a set of items, build a heap

DecreaseKey, IncreaseKey, and Remove

void decreaseKey(int pos, int delta) {
    temp = Heap[pos] - delta;
    newPos = percolateUp(pos, temp);
    Heap[newPos] = temp;
}

void increaseKey(int pos, int delta) {
    temp = Heap[pos] + delta;
    newPos = percolateDown(pos, temp);
    Heap[newPos] = temp;
}

void remove(int pos) {
    percolateUp(pos, 0);
    deleteMin();
}

BuildHeap

Floyd’s Method. Thank you, Floyd.

Build(this)Heap

Finally…
Complexity of Build Heap

- Note: size of a perfect binary tree doubles (+1) with each additional layer
- At most n/4 percolate down 1 level
- At most n/8 percolate down 2 levels
- At most n/16 percolate down 3 levels...

\[
\sum_{i=1}^{n} \frac{n}{2^{i+1}} \leq \frac{n}{2} (2) = n \quad \text{O(n)}
\]

Proof of Summation

\[
S = \sum_{i=1}^{n} \frac{i}{2^i} = \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \cdots + \frac{x-1}{2^{x-1}} + \frac{x}{2^x}
\]

\[
2S = 1 + \frac{2}{2} + \frac{3}{4} + \cdots + \frac{x}{2^{x-1}}
\]

\[
S = 2S - S = 1 + \left(\frac{2}{2} - \frac{1}{2}\right) + \left(\frac{3}{4} - \frac{2}{4}\right) + \cdots + \left(\frac{x}{2^{x-1}} - \frac{x}{2^x}\right)
\]

\[
S \leq 1 + \sum_{i=1}^{x} \frac{1}{2^i} \leq 1 + 1 = 2
\]

Heap Sort

- Input: unordered array A[1..N]
  1. Build a max heap (largest element is A[1])
  2. For i = 1 to N-1:
     A[N-i+1] = Delete_Max()

<table>
<thead>
<tr>
<th>7</th>
<th>50</th>
<th>12</th>
<th>15</th>
<th>4</th>
<th>40</th>
<th>20</th>
<th>10</th>
<th>35</th>
<th>23</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>40</td>
<td>20</td>
<td>25</td>
<td>15</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>35</td>
<td>20</td>
<td>25</td>
<td>15</td>
<td>10</td>
<td>7</td>
<td>4</td>
<td>17</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>25</td>
<td>20</td>
<td>22</td>
<td>7</td>
<td>15</td>
<td>10</td>
<td>4</td>
<td>40</td>
<td>50</td>
</tr>
</tbody>
</table>

Properties of Heap Sort

- Worst case time complexity O(n log n)
  - Build_heap O(n)
  - n Delete_Max’s for O(n log n)

- In-place sort – only constant storage beyond the array is needed

Thinking about Heaps

- Observations
  - finding a child/parent index is a multiply/divide by two
  - operations jump widely through the heap
  - each operation looks at only two new nodes
  - inserts are at least as common as deleteMins

- Realities
  - division and multiplication by powers of two are fast
  - looking at one new piece of data terrible in a cache line
  - with huge data sets, disk accesses dominate

Solution: d-Heaps

- Each node has \(d\) children
- Still representable by array
- Good choices for \(d\):
  - optimize performance based on # of insert/remove
  - choose a power of two for efficiency
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block

What do d-heaps remind us of???
Coming Up

• Thursday: Quiz Section is Midterm Review
  – Come with questions!
• Friday: Midterm Exam
  – Bring pencils
• Monday:
  – Mergeable Heaps
  – 3rd Programming project