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></td>
<td></td>
</tr>
<tr>
<td>Unsorted list (Linked-List)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted list (Array)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sorted list (Linked-List)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary Search Tree (BST)</td>
<td></td>
<td></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):
- leaves(T):
- children(B):
- parent(H):
- siblings(E):
- ancestors(F):
- descendents(G):
- subtree(C):

More Tree Terminology

- depth(B):
- height(G):
- height(T):
- degree(B):
- branching factor(T):
- n-ary tree:

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”.

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: 13
- right child: 11
- parent: 9

implicit (array) implementation:

<table>
<thead>
<tr>
<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>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</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 $X$.

Heap Operations

- findMin:
- 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

Insert Code (optimized)

```java
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;
}
```

Insert Code (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)

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;
}

runtime:
(Java code in book)
Insert: 16, 32, 4, 69, 105, 43, 2

More Priority Queue Operations

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

Binary heap: change priority of node and ____________________

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

Binary heap: change priority of node and ____________________

Why do we need a pointer? Why not simply data value?

Worst case running times?

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.
  • Naïve solution:
    • Running time:

Can we do better with buildHeap?
Building a Heap: Take 1

12 5 11 3 10 6 9 4 8 1 7 2

Building a Heap: Take 2

12 5 11 3 10 6 9 4 8 1 7 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

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…

```
Buildheap pseudocode

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

`runtime:`