Today’s Outline

• Announcements
  – Assignment #2 due NOW.
  – Midterm #1: Friday, Jan 30th
  – Assignment #3 due Thurs, Feb 5th.

• Today’s Topics:
  – Dictionary
    • Balanced Binary Search Trees - (AVL Trees)
  – Priority Queues
    • Binary Min Heap

Priority Queue ADT

1. PQueue data : collection of data with priority
2. PQueue operations
   – insert
   – deleteMin
   (also: create, destroy, is_empty)
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 Q

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

Implementations of Priority Queue ADT

<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>
Representing Complete Binary Trees in an Array

From node i:
- left child:
- right child:
- parent:

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>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>
<td>12</td>
</tr>
</tbody>
</table>

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.

Heap – Insert(val)

Basic Idea:
1. Put val at “next” leaf position
2. Repeatedly exchange node with its parent if needed

Insert pseudo 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;
}
```

Runtime:

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 pseudo Code (Optimized)

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

Other Priority Queue Operations

- **decreaseKey**
  - given a pointer to an object in the queue, reduce its priority value
  
    Solution: change priority and __________________________

- **increaseKey**
  - given a pointer to an object in the queue, increase its priority value
  
    Solution: change priority and __________________________

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

Other Heap Operations

- **decreaseKey(objPtr, amount)**: raise the priority of an object, percolate up
- **increaseKey(objPtr, amount)**: lower the priority of an object, percolate down
- **remove(objPtr)**: remove an object, move to top, then delete.
  1) decreaseKey(objPtr, ∞)
  2) deleteMin()

Worst case Running time for all of these:
FindMax?
ExpandHeap – when heap fills, copy into new space.
Binary Min Heaps (summary)

- **insert**: percolate up. $\Theta (\log N)$ time.
- **deleteMin**: percolate down. $\Theta (\log N)$ time.
- **Build Heap?**

BuildHeap: Floyd’s Method

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

**Facts about Binary Min Heaps**

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

**Realities:**
- division/multiplication by powers of two are equally fast
- looking at only two new pieces of data: bad for cache!
- with huge data sets, disk accesses dominate