CSE 373
Data Structures & Algorithms

Lecture 08
Binary Heaps (Part I)
Recall Queues

• FIFO: First-In, First-Out

• Some contexts where this seems right?

• Some contexts where some things should be allowed to skip ahead in the line?
Queues that Allow Line Jumping

• 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></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</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AVL Trees</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# 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)</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)</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) worst</td>
<td>O(n) worst</td>
</tr>
<tr>
<td>AVL Trees</td>
<td>O(log n)</td>
<td>O(log n)</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

• Heaps provide $O(\log n)$ worst case for both insert and deleteMin, $O(1)$ average insert
Binary Heap Properties

1. Structure Property
2. Ordering Property
root(T):
leaves(T):
children(B):
parent(H):
siblings(E):
ancestors(F):
descendents(G):
subtree(C):
More Tree Terminology

depth(B):

height(G):

degree(B):

branching factor(T):
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.

- height $h$
- $2^{h+1} - 1$ nodes
- $2^h - 1$ non-leaves
- $2^h$ leaves
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:**

![Binary Tree Examples]

---

10/19/2009
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>

10/19/2009
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
```

```
10
  / 
20   80
 / 
30   15

40   60
 / 
50   700

10
  / 
20
  / 
40
  / 
50
  /
80
  /
85
  /
99
```
Heap Operations

- **findMin:**
- **insert**(val): percolate up.
- **deleteMin:** percolate down.

```plaintext
  10
 /  \
20   80
 / \
40   60
 / \
50   65
 / \
700
```
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

10

20

40

50 700 65 15

60

80

85

99

10

15

40

20

50 700 65 60

80

85

99
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:

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

(On the white board...)

10/19/2009
More 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?
More Priority Queue Operations

• **Remove(objPtr)**
  - given a pointer to an object in the queue, remove the object from the queue

  **Solution**: set priority to negative infinity, percolate up to root and deleteMin

• **FindMax**
Facts about 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
Priority Queue Operations

- `insert(obj)`
- `deletemin(obj)`
- `decreaseKey(objPtr, amount)`
- `increaseKey(objPtr, amount)`
- `remove(objPtr)`
- `findMax( )`
- `expandHeap( )`
- `buildHeap(objList)`
Building a Heap

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

12 → 5 → 3 → 5 → 3

12  5  3  11

12

5

11

12

10
Building a Heap

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

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

Diagram:

- Initial array: 12  5  11  3  10  6  9  4  8  1  7  2
- After heapify: 1  4  3  5  6  9  11  8  12
- Final array: 1  3  6  4  5  9  11  8  12
Building a Heap

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

• At every point, the new item may need to percolate all the way through the heap

• Adding the items one at a time is $O(n \log n)$ in the worst case (what is the worst case?)

• Next lecture we get clever and do it in $O(n)$