Priority Queue ADT

“store key-value pairs and support operations: insert, delete-min, and reduce-key”

**insert**\((k, v)\)

adds key-value pair \((k, v)\) to queue.

**delete-min() \(\Rightarrow (k, v)\)**

Removes key-value pair \((k, v)\) with minimum key value \(k\) and returns it.

**reduce-key**\((k, v)\)

Finds the key in queue with value \(v\) and reduces its key to \(k\).

**Goal:** implement all priority queue ADT operations in \(O(\log n)\).

Binary Heaps

**Def:** a binary heap is a tree that satisfies

- **heap order:** the key at any node is at least the key of its parent.

- **heap structure:** it’s a complete binary tree.

Example:

```
Example:

Questions:

- How to represent in memory?
- How to maintain heap properties?

Idea: Complete tree can be represented in array.

Example:

```

```

Traversals

- root() = 1
- leftchild\((i)\) = \(2i\)
- rightchild\((i)\) = \(2i + 1\)
- parent\((i)\) = \([i/2]\)

Add and Delete

- add-key\((k)\)
“add element with key $k$”

1. put key in position $n + 1$.
2. set $n = n + 1$.
3. fix violated heap property by “percolating up” key $k$.

- **percolate-up($i$)**
  “moves key at position $i$ up tree until it doesn’t violate heap property.”

  1. if key at parent($i$) > key at $i$
     (a) swap keys.
     (b) recursively percolate-up(parent($i$))

**Example:** add-key(1)

- Initially:
  1. Move key “5” to position 1

- Percolate up at key “1”:

- Percolate up at key “1”:
– Percolate-down key “5”:

– Percolate-down key “5” ⇒ done!

Priority Queue via Binary Heap

P.Q. Representation

heap: “length c array of key-value pairs”
aux: “length c array mapping values to nodes”
n: “number of keys in queue”

P.Q. Operations

• insert(k,v)
  1. run add-key(k) on heap.
  2. update aux with location of (k,v).
  3. update aux with location of all moved keys.

• delete-min() ⇒ (k,v)
  1. (k,v) = key and value at root.
  2. run delete-node(root()) on heap.
  3. update aux with location of all moved keys.

• reduce-key(k,v)
  1. i = aux[v]
  2. update key at i to k.

3. run percolate-up(i) on heap
4. update aux with location of all moved keys.

Runtimes: Θ(logn) (each operation).

Build-heap

Fact: sum of heights of nodes = Θ(n).
Algorithm: build-heap
Input: unordered array of keys, size n.

1. view array as complete binary tree.
2. for i = n down to 1
   percolate-down(i)

Example:

• Input unordered array.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>5</td>
<td>4</td>
<td>9</td>
<td>1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• view as tree.

• percolate-down(3)
Claim: output of build-heap is heap-ordered.

Proof: by induction on $i$. trivial.

Claim: build-heap runs in $\Theta(n)$ time.

Proof: runtime = sum of node heights = $\Theta(n)$.

Proof: (of fact)

- runtime = “sum of heights”
- $S = \text{sum of heights}$

\[
S = \sum_{j=0}^{h} j \times \text{num. nodes at height } h
= \sum_{j=0}^{h} j \times 2^{h-j}
\]

$\blacksquare$