Heaps

- Heap is a great data structure
- Any chance to make it even better?

<table>
<thead>
<tr>
<th></th>
<th>SortedVector</th>
<th>SortedList</th>
<th>Heap</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td></td>
<td>Binary search</td>
<td>Linear search</td>
<td>Percolate up</td>
</tr>
<tr>
<td></td>
<td>Slide data up</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>getMin</strong></td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td></td>
<td>get(0)</td>
<td>Returns firstLink val</td>
<td>Get root node</td>
</tr>
<tr>
<td><strong>removeMin</strong></td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td></td>
<td>Slide data down</td>
<td>removeFront()</td>
<td>Percolate down</td>
</tr>
</tbody>
</table>
Seeking an O(1) addition

• How do we obtain O(1) addition?
• Answer: only change the location of the root
• How to maintain heap structure?
  • Heap is partial order!
Robert Tarjan

• Professor of Princeton
• Probably the most influential DS researcher in the 1980s
• Many algorithms/advanced DS
  • Splay trees
  • Pairing heaps
  • Fibonacci heaps
  • Goldberg-Tarjan push-relabel max-flow
  • Hopcroft-Tarjan Planarity-testing
• Turing Award in 1986
Pairing heap \((\text{Fredman, Sedgewick, Sleator, Tarjan 1986})\)

- Maintain root and a list of subheaps

Instead of:

Maintain:

<table>
<thead>
<tr>
<th>Root</th>
<th>List of subheaps</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

```
3
|
3
9
14
12
11
16

5
7
8
```

Last filled position (not necessarily the last element added)

Root = Smallest element

Next open spot
Pairing heap: O(1) insertion

• Merge 2 heaps operation:

Root | List of subheaps
---|---
2 | 

| Insert |
---|---
6 | 

Root | List of subheaps
---|---
2 | 

3 | 9
12
14

5 | 7
8

10 | 16
11

6 | 

3 | 9
12
14

10 | 16
11

5 | 7
8

6 |
Pairing heap: $O(1)$ insertion
Pairing heap: $O(1)$ merging
Pairing heap: Deletion of root

• Am I cheating?
• What if I just inserted many elements?
• Deletion is going to be very difficult!
• No and yes
Pairing heaps: deletion (step 1)
Pairing heaps: deletion (step 2)
Pairing heaps: deletion (step 3)
Pairing heaps: formal deletion algorithm

```plaintext
function delete-min(heap)
    if heap == Empty
        error
    else
        return merge-pairs(heap.subheaps)
```

This uses the auxiliary function `merge-pairs`:

```plaintext
function merge-pairs(l)
    if length(l) == 0
        return Empty
    elseif length(l) == 1
        return l[0]
    else
        return merge(merge(l[0], l[1]), merge-pairs(l[2..]))
```
Pairing heaps: merging sequence

merge-pairs([H1, H2, H3, H4, H5, H6, H7])
=> merge(merge(H1, H2), merge-pairs([H3, H4, H5, H6, H7]))
    # merge H1 and H2 to H12, then the rest of the list
=> merge(H12, merge(merge(H3, H4), merge-pairs([H5, H6, H7])))
    # merge H3 and H4 to H34, then the rest of the list
=> merge(H12, merge(H34, merge(merge(H5, H6), merge-pairs([H7]))))
    # merge H5 and H6 to H56, then the rest of the list
=> merge(H12, merge(H34, merge(H56, H7)))
    # switch direction, merge the last two resulting heaps, giving H567
=> merge(H12, merge(H34, H567))
    # merge the last two resulting heaps, giving H34567
=> merge(H12, H34567)
    # finally, merge the first merged pair with the result of merging the rest
=> H1234567
Pairing heaps: deletion time

• Amortized log-n time:
  • $O((\log n)\text{+})$

• Analysis too complicated here
  • Basic point is that each deletion makes the heap more “binary” which makes subsequent ones faster

• Why would pairing heaps work?
  • Utilize multi-way trees
  • Data structures can get more complicated than class!
Many more heaps


<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>find-min</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>delete-min</td>
<td>Θ(log n)</td>
<td>Θ(log n)</td>
<td>O(log n)[b]</td>
<td>O(log n)[b]</td>
<td>O(log n)</td>
<td>O(log n)[b]</td>
<td>O(log n)</td>
</tr>
<tr>
<td>insert</td>
<td>Θ(log n)</td>
<td>Θ(1)[b]</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>decrease-key</td>
<td>Θ(log n)</td>
<td>Θ(log n)</td>
<td>Θ(1)[b]</td>
<td>o(log n)[b][c]</td>
<td>Θ(1)</td>
<td>Θ(1)[b]</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>merge</td>
<td>Θ(n)</td>
<td>O(log n)[c]</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
<td>Θ(1)</td>
</tr>
</tbody>
</table>
Practical Performance

• A Back-to-Basics Empirical Study of Priority Queues

• Pairing heap is the most efficient in practice!