Administration

- Released today: Project 2, phase B
- Due today: Homework 1
- Released today: Homework 2
- I have office hours tomorrow

BuildHeap: Floyd’s Method

Add elements arbitrarily to form a complete tree. Pretend it’s a heap and fix the heap-order property!

Buildheap pseudocode

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

Finally…
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

Representing Complete Binary Trees in an Array

From node $i$:
- left child: $2i$ + 1
- right child: $2i$ + 2
- parent: $\left\lfloor \frac{i-1}{2} \right\rfloor$

implicit (array) implementation:

\[
\begin{array}{cccccccccccccc}
A & B & C & D & E & F & G & H & I & J & K & L \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13
\end{array}
\]

A Solution: $d$-Heaps

• Each node has $d$ children
• Still representable by array
• Good choices for $d$:
  - choose a power of two for efficiency
  - fit one set of children in a cache line
  - fit one set of children on a memory page/disk block

Operations on $d$-Heap

- Insert: runtime =
- deleteMin: runtime =

Does this help insert or deleteMin more?

One More Operation

- Merge two heaps. Ideas?
New Operation: Merge
Given two heaps, merge them into one heap
  › first attempt: insert each element of the smaller heap into the larger.
    runtime:
  › second attempt: concatenate binary heaps’ arrays and run buildHeap.
    runtime:

Merging heaps
• Binary Heap is a special purpose hot rod
  › FindMin, DeleteMin and Insert only
  › does not support fast merges of two heaps
• For some applications, the items arrive in prioritized clumps, rather than individually
• Is there somewhere in the heap design that we can give up a little performance so that we can gain faster merge capability?

Binomial Queues
• Binomial Queues are designed to be merged quickly with one another
• Using pointer-based design we can merge large numbers of nodes at once by simply pruning and grafting tree structures
• More overhead than Binary Heap, but the flexibility is needed for improved merging speed

Worst Case Run Times
<table>
<thead>
<tr>
<th></th>
<th>Binary Heap</th>
<th>Binomial Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>(\Theta(\log N))</td>
<td>(\Theta(\log N))</td>
</tr>
<tr>
<td>FindMin</td>
<td>(\Theta(1))</td>
<td>(O(\log N))</td>
</tr>
<tr>
<td>DeleteMin</td>
<td>(\Theta(\log N))</td>
<td>(\Theta(\log N))</td>
</tr>
<tr>
<td>Merge</td>
<td>(\Theta(N))</td>
<td>(O(\log N))</td>
</tr>
</tbody>
</table>

Binomial Queue with 5 Trees

Binomial Queues
• Binomial queues give up simplicity in order to provide \(O(\log N)\) merge performance
• A binomial queue is a collection (or forest) of heap-ordered trees
  › Not just one tree, but a collection of trees
  › each tree has a defined structure and capacity
  › each tree has the familiar heap-order property
**Structure Property**

- Each tree contains two copies of the previous tree
  - the second copy is attached at the root of the first copy
- The number of nodes in a tree of depth $d$ is exactly $2^d$

<table>
<thead>
<tr>
<th>Depth</th>
<th>Number of Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$2^0 = 1$</td>
</tr>
<tr>
<td>1</td>
<td>$2^1 = 2$</td>
</tr>
<tr>
<td>2</td>
<td>$2^2 = 4$</td>
</tr>
</tbody>
</table>

**Powers of 2**

- Any number $N$ can be represented in base 2
  - A base 2 value identifies the powers of 2 that are to be included

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Binary</th>
<th>$2^0$</th>
<th>$2^1$</th>
<th>$2^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Numbers of nodes**

- Any number of entries in the binomial queue can be stored in a forest of binomial trees
- Each tree holds the number of nodes appropriate to its depth, ie $2^d$ nodes
- So the structure of a forest of binomial trees can be characterized with a single binary number
  - $100_2 \rightarrow 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 4$ nodes

**Structure Examples**

- Any number of entries in the binomial queue can be stored in a forest of binomial trees
- Each tree holds the number of nodes appropriate to its depth, ie $2^d$ nodes
- So the structure of a forest of binomial trees can be characterized with a single binary number
  - $100_2 \rightarrow 1 \cdot 2^2 + 0 \cdot 2^1 + 0 \cdot 2^0 = 4$ nodes

**What is a merge?**

- There is a direct correlation between
  - the number of nodes in the tree
  - the representation of that number in base 2
  - and the actual structure of the tree
- When we merge two queues, the number of nodes in the new queue is the sum of $N_1 + N_2$
- We can use that fact to help see how fast merges can be accomplished
Example 2.
Merge BQ.1 and BQ.2
This is an add with a carry out.
It is accomplished with one comparison and one pointer change: $O(1)$

Example 3.
Part 1 - Form the carry.

$BQ.1 + BQ.2 = \text{carry}$

Part 2 - Add the existing values and the carry.

$BQ.1 + BQ.2 = BQ.3$

Merge Algorithm
- Just like binary addition algorithm
- Assume trees $X_0,\ldots,X_n$ and $Y_0,\ldots,Y_n$ are binomial queues
  $\Rightarrow X$ and $Y$ are of type $B$ or null

$C_0 := \text{null}$; //initial carry is null//
for $i = 0$ to $n$ do
  combine $X_i$, $Y_i$, and $C_i$ to form $Z_i$ and new $C_{i+1}$
  $Z_{i+1} := C_{i+1}$

$O(\log N)$ time to Merge
- For $N$ keys there are at most $\lceil \log_2 N \rceil$ trees in a binomial forest.
- Each merge operation only looks at the root of each tree.
- Total time to merge is $O(\log N)$. 
Insert

- Create a single node queue $B_0$ with the new item and merge with existing queue
- $O(\log N)$ time

DeleteMin

1. Assume we have a binomial forest $X_0, \ldots, X_m$
2. Find tree $X_k$ with the smallest root
3. Remove $X_k$ from the queue
4. Remove root of $X_k$ (return this value)
   - This yields a binomial forest $Y_0, Y_1, \ldots, Y_{k-1}$.
5. Merge this new queue with remainder of the original (from step 3)
   - Total time = $O(\log N)$

Implementation

- Binomial forest as an array of multiway trees
  - FirstChild, Sibling pointers
  - Subtrees in decreasing sizes

DeleteMin Example

Old forest

New forest

Return this value

Merge
Why Binomial?

\[
\binom{n}{k} \cdot \frac{1}{(d-k)!} \cdot \frac{1}{(d-k)!} = B_0 \quad B_1 \quad B_2 \quad B_3 \quad B_4
\]

\[
\text{tree depth } d = 4 \quad 3 \quad 2 \quad 1 \quad 0
\]

\[
\text{nodes at depth } k = 1, 4, 6, 4, 1 \quad 1, 3, 1, 1 \quad 1, 2, 1 \quad 1, 1 \quad 1
\]

Other Priority Queues

- **Leftist Heaps**
  - \(O(\log N)\) time for insert, deletemin, merge
- **Skew Heaps**
  - \(O(\log N)\) amortized time for insert, deletemin, merge
- **Calendar Queues**
  - \(O(1)\) average time for insert and deletemin
  - Assuming insertions are “random”

Exercise Solution

\[
\begin{array}{ccccc}
4 & 9 & + & 2 & 4 \\
5 & 7 & & 8 & 4 \\
\end{array}
\]