CIS 313: Intermediate Data Structure

week of Jan 21

third week of the term
binary heap implementation of PQ

- most common implementation
- operations are $O(\log n)$
- uses a binary tree structure
- except that the tree is stored in an array with no pointers
- it is an *implicit* tree, children and parents inferred from location in array

- PQSort becomes *heapsort*
binary heap

- stored in array
- item located in position \( i \)
  - parent in location \([i/2]\)
  - left child in position \(2i\)
  - right child in position \(2i + 1\)
- tree is complete
  - all nodes have two children, except maybe parent of “last” one
- tree maintains heap property
  - value stored at location \( i \) is greater than or equal to values stored in both its children
- fact: a binary heap with \( n \) elements has the height of \( \lceil \lg n \rceil \) (why?)
binary heap insertion

- put new value $x$ at end of array, extending its size by 1
- value $x$ is now viewed as being at the bottom of the tree
- if $x$ violates heap property (if larger than parent), swap with parent
- repeat until no violation
- time is proportional to height of tree, which is $O(lg \ n)$

- text handles this differently, they insert $-\infty$ and then use heap-increase-key to the new value
pseudo-code for insert

```
insert(x):

heapsize++
A[heapsize]=x

i = heapsize
while i>1 and A[i]>A[parent(i)]
    swap A[i] and A[parent(i)]
    i = parent(i)
```

sometimes called “sift-up” or “bubble-up”
Binary Heap: Insert Operation

Viewed as a binary tree:

- Left:
  - 1
  - 16
  - 2
  - 3
  - 11
  - 12
  - 4
  - 5
  - 6
  - 7
  - 8
  - 10
  - 9

- Right:
  - 1
  - 16
  - 2
  - 3
  - 14
  - 4
  - 5
  - 6
  - 7
  - 8
  - 10
  - 9
  - 12

Viewed as an array:

- Left:
  - 1 2 3 4 5 6 7
  - 16 11 12 8 10 9 14

- Right:
  - 1 2 3 4 5 6 7
  - 16 11 14 8 10 9 12
heap extract-max (deletion)

• similar but element moves down
• idea: remove and return root (in location 1 of the tree)
• move rightmost element into that empty location …
• … and reduce the heapsize
• tree shape is maintained but root location may violate heap property
• note: rest of tree still has heap property
• swap node with larger (why) of it’s children
• repeat while heap property violated until leaf hit
• called “sift-down” or “bubble-down”
text algorithm

```
MAX-HEAPIFY(A, i)

// Input: A: an array where the left and right children of i root heaps (but i may not), i: an array index
// Output: A modified so that i roots a heap
// Running Time: O(log n) where n = heap-size[A] - i
1  l ← LEFT(i)
2  r ← RIGHT(i)
4      largest ← l
5  else largest ← i
7      largest ← r
8  if largest ≠ i
9      exchange A[i] and A[largest]
10     MAX-HEAPIFY(A, largest)
```
first attempt at sorting

1. for each element $x$, insert $x$ into a heap
   - time per insert $O(lg n)$, total $O(n lg n)$
   - this can be made much faster

2. while the heap is not empty, extract-max
   - output is a sorted list (reversed)
   - each extract-max is $O(lg n)$, total $O(n lg n)$
   - cannot be made faster

BUILDHEAP uses deletion idea to get linear overall time
buildheap code

**Build-Max-Heap(A)**

```plaintext
// Input: A: an (unsorted) array
// Output: A modified to represent a heap.
// Running Time: O(n) where n = length[A]
1  heap-size[A] ← length[A]
2  for i ← [length[A]/2] downto 1
3      MAX-HEAPIFY(A, i)
```

correctness
- idea sort of clear, build heaps bottom up
- text uses loop invariant!!

time analysis
- if tree has height $H=\lg n$
- all nodes at level $k$ take time $H-k$ to sift down
- there are $2^k$ nodes at level $k$
- total time is $\sum_0^H 2^k (H - k)$
- can show this is at most $2n$
grinding through the time bound

\[ \sum_{k=0}^{H} 2^k (H - k) = 2^H \sum_{k=0}^{H} (2^k / 2^H)(H - k) \]

\[ = n \cdot \sum_{k=0}^{H} \frac{1}{2^{H-k}} (H - k) \]

\[ = n \cdot \frac{\sum_{i=0}^{H} \frac{i}{2^i}}{\sum_{i=0}^{\infty} \frac{i}{2^i}} \leq n \cdot \frac{\sum_{i=0}^{\infty} \frac{i}{2^i}}{2} = 2 \cdot n \]

why just 2?
- mentioned but not proved in appendix
- “fun” to derive
- can also take derivative of \( \sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \)

\[ 2^H \approx 2^{\log_2 n} = n \]

re-index
now heapsort

```
HEAP-SORT(A)
   // Input: A: an (unsorted) array
   // Output: A modified to be sorted from smallest to largest
   // Running Time: O(n log n) where n = length[A]
   1 BUILD-MAX-HEAP(A)
   2 for i = length[A] downto 2
      3   exchange A[1] and A[i]
      4   heap-size[A] ← heap-size[A] − 1
      5   MAX-HEAPIFY(A, 1)
```

- step 1: $\Theta(n)$ time
- steps 2-5: $\Theta(n \log n)$ time
other heap operation: increase-key

- an item can be increased in $O(\lg n)$ time
- after the increase, it would need to be sifted up as in the insert method
- the same applies to the decrease-key operation in a min heap
- this operation is a crucial step in Dijkstra’s method (shortest path) and Prim’s method (minimum spanning tree)
- it can be implemented in $O(1)$ amortized time using Fibonacci heaps

- we will not cover Fibonacci heaps, but next we look at a similar and simpler structure: binomial heaps
<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>INSERT</td>
<td>Θ(lg n)</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>Θ(1)</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>Θ(lg n)</td>
</tr>
<tr>
<td>UNION</td>
<td>Θ(n)</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>Θ(lg n)</td>
</tr>
<tr>
<td>DELETE</td>
<td>Θ(lg n)</td>
</tr>
</tbody>
</table>
union of binary heaps

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
<th>Binomial heap (worst-case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\Theta(\lg n)$</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>$\Theta(1)$</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
</tr>
<tr>
<td>UNION</td>
<td>$\Theta(n)$</td>
<td>$O(\lg n)$</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\Theta(\lg n)$</td>
<td>$\Theta(\lg n)$</td>
</tr>
</tbody>
</table>
small digression: ordered trees

ordered tree:

- tree has designated root
- a node can have any number of children
- if a node has k children, they are ordered
  - 1\textsuperscript{st} child, 2\textsuperscript{nd} child, ..., k\textsuperscript{th} child
- good representation involves two pointers per node:
  - first-child and next-sibling
  - so the children of a node are in a linked list
binomial trees

- A *binomial heap* will be a collection of binomial trees with the heap property
- So we need to define a *binomial tree* first
- A binomial tree is defined recursively:
  - A $B_0$ tree is a single node (height 0)
  - A $B_k$ tree consists of a $B_{k-1}$ tree whose root has another $B_{k-1}$ tree as a child

- A $B_k$ tree contains $2^k$ nodes
- The height of the tree is $k$
- The number of nodes on level $j$ of a $B_k$ tree is the binomial coefficient $\binom{k}{j}$
- The root has degree $k$, which is greater than the degree of any other node; moreover, if the children of the root are numbered from left to right by $k - 1, k - 2, \ldots, 0$, then child $i$ is the root of a subtree $B_i$
these trees will be represented using the first-child next sibling representation of ordered trees. One node will have three points:
• one points to the parent of the node
• one points to its leftmost child
• one points to its sibling immediately on the right
binomial heap

- collection of binomial trees, satisfying:
  - each binomial tree satisfy the heap property (values of parents less or equal to values at the children)
  - there is at most one binomial tree whose root has a given degree
- min value could be at root of one of the trees
- if $n$ nodes stored, then $\lg n$ trees used, corresponds to binary representation of $n$
- example: if $n=13$, need $B_0, B_2, B_3$ trees (containing 1, 4, 8 nodes)
- $n=13=(1101)_2$ in base 2
merge two $B_k$ trees

- two $B_k$ trees can be merged into a $B_{k+1}$ tree
- look at the two roots ...
- ... the root with larger value becomes child of root with smaller value
- easy since children of root given in linked list
- result is $B_{k+1}$ tree
main operation: union of two binomial heaps

• two heaps of sizes n and m can be merged in time $O(lg n + lg m)$

• idea is simple:
  • for $k=0, 1, 2, \ldots$
  • scan through each heap’s tree list
  • if there are two $B_k$ trees, merge them together into a $B_{k+1}$ tree
  • (*note 1*: one of the $B_k$ trees might be the result of an earlier merge)
  • (*note 2*: there might be three $B_k$ trees, one each from the two heaps and one from an earlier merge – pick the two later trees– similar to a carry bit)

• operation parallels closely addition in binary
main operation: union of two binomial heaps
Case 1

Case 2

key[x] ≤ key[next-x]

Case 3

key[x] > key[next-x]

Case 4
example union

\[
\begin{array}{c}
1011 \\
+ 0011 \\
\hline
1110
\end{array}
\]

\[
\begin{array}{c}
11 \\
+ 3 \\
\hline
14
\end{array}
\]
other operations “reduce” to union

• insertion:
  • to insert x into heap H
  • create heap $H'$ consisting of only x
  • perform union of $H$ and $H'$
• time $O(\log n)$
  • actually not so bad if many insertions performed
  • a sequence of $n$ insertions into an initially empty heap uses $O(n)$ time
  • similar: $n$ increments (by one) of a binary counter (initially zero) makes $O(n)$ bit flips
  • analysis: we saw something like \( \sum_{i=0}^{\log n} i2^{-i} = O(n) \) with the BuildHeap routine
extract-min

- the min is the root of one of the trees in the binomial heap \( H \)
- suppose it’s a \( B_k \) tree with root \( x \)
  - pull the tree with root \( x \) out of \( H \)
  - the children of \( x \) form a binomial heap \( H' \)
  - \( (H' \) will have one each of a \( B_0, B_1, ..., B_{k-1} \) tree)
  - perform a union \( H' \) and the reminder of \( H \)
  - return key of \( x \)
- \( O(\lg n) \) time
1 is the min of H

pull out the tree with 1 as root

remove 1 and look at its child list as heap H'

get union of H' with remaining heap H