Mergeable Heaps

David Kauchak
CS140
Spring 2013

Binary heap

A binary tree where the value of a parent is greater than or equal to the value of its children

Additional restriction: all levels of the tree are complete except the last

Max heap vs. min heap

Binary heap - operations

Max - return the largest element in the set

ExtractMax – Return and remove the largest element in the set

Insert(val) – insert val into the set

IncreaseElement(x, val) – increase the value of element x to val

BuildHeap(A) – build a heap from an array of elements

Assignments

Assignment 2 graded
Assignment 4 – start working!
Binary heap representations

Heapify
Assume left and right children are heaps, turn current set into a valid heap

heapify(A, i)  
1  l ← Left(i)  
2  r ← Right(i)  
3  largest ← i  
5      largest ← l  
7      largest ← r  
8  if largest ≠ i  
9      swap A[i] and A[largest]  
10     heapify(A, largest)
Heapify

16 10 8 7 9 5 2 4 1
1 2 3 4 5 6 7 8 9 10

Heapify(A, i)
1 i <- Last(i)
2 r <- Right(i)
3 largest = i
4 if r ≤ heap-size[A] and A[r] > A[i] 
5 largest = r 
7 largest = r 
8 if largest ≠ i
9 swap A[i] and A[largest]
10 Heapify(A, largest)

9

Heapify

16 3 10 8 7 9 5 2 4 1
1 2 3 4 5 6 7 8 9 10

Heapify(A, i)
1 i <- Last(i)
2 r <- Right(i)
3 largest = i
4 if r ≤ heap-size[A] and A[r] > A[i] 
5 largest = i 
7 largest = r 
8 if largest ≠ i
9 swap A[i] and A[largest]
10 Heapify(A, largest)

10

Heapify

16 8 10 3 7 9 5 2 4 1
1 2 3 4 5 6 7 8 9 10

Heapify(A, i)
1 i <- Last(i)
2 r <- Right(i)
3 largest = i
4 if r ≤ heap-size[A] and A[r] > A[i] 
5 largest = i 
7 largest = r 
8 if largest ≠ i
9 swap A[i] and A[largest]
10 Heapify(A, largest)

11

Heapify

16 8 10 3 7 9 5 2 4 1
1 2 3 4 5 6 7 8 9 10

Heapify(A, i)
1 i <- Last(i)
2 r <- Right(i)
3 largest = i
4 if r ≤ heap-size[A] and A[r] > A[i] 
5 largest = i 
7 largest = r 
8 if largest ≠ i
9 swap A[i] and A[largest]
10 Heapify(A, largest)

12
Heapify

\[
\begin{array}{c}
16 & 10 & 8 & 4 \\
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 \\
\end{array}
\]

1. \textbf{Heapify}(A, i)
2. \text{left} = \text{left}(i)
3. \text{right} = \text{right}(i)
4. \text{largest} = i
5. \text{if} \ i \leq \text{left}(i) \text{ and } A[i] > A[\text{left}]
6. \text{largest} = \text{left}
7. \text{if} \ i \leq \text{right}(i) \text{ and } A[i] > A[\text{right}]
8. \text{largest} = \text{right}
9. \text{if} \ \text{largest} \neq i
10. \text{swap } A[i] \text{ and } A[\text{largest}]
11. \text{Heapify}(A, \text{largest})

Running time of Heapify

O(\text{height of the tree})

What is the height of the tree?
- Complete binary tree, except for the last level

\[2^h \leq n\]

\[h \leq \log_2 n\]

O(\log n)
**Binary heap - operations**

Max - return the largest element in the set

ExtractMax – Return and remove the largest element in the set

Insert(val) – insert val into the set

IncreaseElement(x, val) – increase the value of element x to val

BuildHeap(A) – build a heap from an array of elements

---

**Max**

What is the largest element in the set?

**Return A[1]**
**ExtractMax**
Return and remove the largest element in the set

![Heapify Diagram](image1)

**ExtractMax**
Return and remove the largest element in the set

![Heapify Diagram](image2)

**IncreaseElement**
Increase the value of element \(x\) to \(val\)

![Increase Element Diagram](image3)

**IncreaseElement**
Increase the value of element \(x\) to \(val\)

![Increase Element Diagram](image4)
IncreaseElement
Increase the value of element \( x \) to \( val \)

Runtime? \( O(\text{height of tree}) = O(\log n) \)
Insert
Insert val into the set

Insert
Insert val into the set

Insert
Insert val into the set

Insert
Insert val into the set
Building a heap

Can we build a heap using the functions we have so far?

- Max
- ExtractMax
- Insert(val)
- IncreaseElement(x, val)

For each element x in array:

insert(x)

Building a heap: take 2

Start with n/2 "one-node" heaps

call Heapify on element n/2-1, n/2-2, n/2-3 ...

take all children have smaller indices

building from the bottom up, makes sure that all the children are heaps

Running time of BuildHeap1

n calls to Insert – O(n log n)

Can we do better?

BUILD-HEAP1(A)
1 copy A to B
2 heap-size[A] ← 0
3 for i ← 1 to length[B]
4 INSERT(A, B[i])
Running time of BuildHeap2

\( n/2 \) calls to Heapify – \( O(n \log n) \)

Can we get a tighter bound?

```
BUILD-HEAP2(A)
1 heap.size[A] = (length[A])
2 for i = [(length[A])/2] + 1 to 1
3 HEAPIFY(A, i)
```
Running time of BuildHeap2

\[ T(n) = \sum_{d=1}^{\log_2 n} 2^d \Theta(d) \]

Nodes at height \( h \)

- \( h = 0 \) < ceil(n/2) nodes
- \( h = 1 \) < ceil(n/4) nodes
- \( h = 2 \) < ceil(n/8) nodes
- ...

Binary heaps

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILD-HEAP</td>
<td>( \Theta(n) )</td>
</tr>
<tr>
<td>INSERT</td>
<td>( \Theta(\log n) )</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>EXTRACT-MAX</td>
<td>( \Theta(\log n) )</td>
</tr>
<tr>
<td>UNION</td>
<td>( \Theta(\log n) )</td>
</tr>
<tr>
<td>INCREASE-ELEMENT</td>
<td>( \Theta(\log n) )</td>
</tr>
<tr>
<td>DELETE</td>
<td>( \Theta(\log n) )</td>
</tr>
</tbody>
</table>

(adapted from Figure 19.1, pg. 456 [1])
Mergeable heaps

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILD-HEAP</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>EXTRACT-MAX</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>UNION</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>INCREASE-ELEMENT</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

(Adapted from Figure 19.1, pg. 456 [1])

- Mergeable heaps support the union operation
- Allows us to combine two heaps to get a single heap
- Union runtime for binary heaps?

Union for binary heaps

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILD-HEAP</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>EXTRACT-MAX</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>UNION</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>INCREASE-ELEMENT</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

(Adapted from Figure 19.1, pg. 456 [1])

- Concatenate the arrays and then call Build-Heap

Linked-list heap

- Store the elements in a doubly linked list
- Insert:
- Max:
- Extract-Max:
- Increase:
- Union:

Linked-list heap

- Store the elements in a doubly linked list
- Insert: add to the end/beginning
- Max: search through the linked list
- Extract-Max: search and delete
- Increase: increase value
- Union: concatenate linked lists
**Linked-list heap**

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
<th>Linked-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILD-HEAP</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>UNION</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>INCREASE-ELEMENT</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

(adapted from Figure 19.1, pg. 436 [1])

Faster Union, Increase, Insert and Delete… but slower Max operations

**Binomial Tree**

$B_k$, a binomial tree $B_{k-1}$ with the addition of a left child with another binomial tree $B_{k-1}$.

---

**Binomial Tree**

Number of nodes with respect to $k$?

$N(B_0) = 1$

$N(B_k) = 2 \cdot N(B_{k-1}) = 2^k$

---

**Binomial Tree**

Height?

$H(B_0) = 1$

$H(B_k) = 1 + H(B_{k-1}) = k$
Binomial Tree

Degree of root node?

$k$, each time we add another binomial tree

What are the children of the root?

$k$ binomial trees: $B_{k-1}, B_{k-2}, \ldots, B_0$

Why is it called a binomial tree?

$B_k$ has $\binom{k}{i}$ nodes at depth $i$. 

82

83

84

85
**Binomial Heap**

Sequence of binomial trees that satisfy binomial heap property:
- each tree is min-heap ordered
- top level: full or empty binomial tree of order k
- which are empty or full is based on the number of elements

[Diagram of binomial heap]

**Binomial Heap: Properties**

How many heaps?

O(log n) – binary number representation

[Diagram of binomial heap]

**Binomial Heap: Properties**

Where is the max/min?

Must be one of the roots of the heaps

[Diagram of binomial heap]
Binomial Heap: Properties

Runtime of max/min?

$O(\log n)$

Binomial Heap: Properties

Height?

$\log_2 n$
- largest tree = $B_{\log_2 n}$
- height of that tree is $\log n$

Binomial Heap: Union

How can we merge two binomial tree heaps of the same size ($2^k$)?
- connect roots of $H'$ and $H''$
- choose smaller key to be root of $H$

Runtime?
$O(1)$

Binomial Heap: Union

How can we combine/merge binomial heaps (i.e. a combination of binomial tree heaps)?
Binomial Heap: Union

Go through each tree size starting at 0 and merge as we go.

19 + 7 = 26

94

Binomial Heap: Union

95

96

97
Binomial Heap: Union

Analogous to binary addition

Running time?
- Proportional to number of trees in root lists $2 \Theta(\log N)$
- $O(\log N)$

$19 + 7 = 26$
Binomial Heap: Delete Min/Max

We can find the min/max in $O(\log n)$.

How can we extract it?

Hint: $B_k$ consists of binomial trees:

$B_{k-1}, B_{k-2}, \ldots, B_0$

Binomial Heap: Delete Min

Delete node with minimum key in binomial heap $H$.

- Find root $x$ with min key in root list of $H$, and delete
- $H' \leftarrow$ broken binomial trees
- $H \leftarrow \text{Union}(H', H)$

Running time? $O(\log N)$

Heaps

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
<th>Biomial heap (worst-case)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Build-Heap</td>
<td>$\Theta(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Insert</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Maximum</td>
<td>$\Theta(1)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Extract-Max</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Union</td>
<td>$\Theta(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Increase-Element</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Delete</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

(adapted from Figure 19.1, pg. 456 [2])
Fibonacci Heaps

Similar to binomial heap
- A Fibonacci heap consists of a sequence of heaps
  - More flexible
- Heaps do not have to be binomial trees
  - More complicated

More complicated

Should you always use a Fibonacci heap?

Heaps

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary heap (worst-case)</th>
<th>Binomial heap (worst-case)</th>
<th>Fibonacci heap (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BUILD-HEAP</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>INSERT</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>$\Theta(1)$</td>
<td>$O(\log n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>EXTRACT-MAX</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>UNION</td>
<td>$\Theta(n)$</td>
<td>$O(\log n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>INCREASE-ELEMENT</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

(Adapted from Figure 19.1, pg. 466 [1])

- Extract-Max and Delete are $O(n)$ worst case
- Constants can be large on some of the operations
- Complicated to implement

Can we do better?