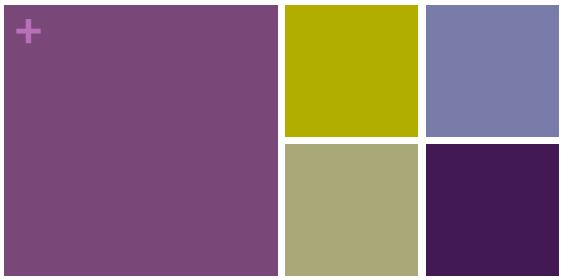


David Kauchak
cs312

Review

Midterm

- Will be posted online this afternoon
- You will have 2 hours to take it
  - watch your time!
  - if you get stuck on a problem, move on and come back
- Must take it by Friday at 6pm
- You may use:
  - your book
  - your notes
  - the class notes
  - ONLY these things
- Do NOT discuss it with anyone until after Friday at 6pm

Midterm

- General
  - what is an algorithm
  - algorithm properties
  - pseudocode
  - proving correctness
  - run time analysis
  - memory analysis

Midterm

- Big O
  - proving bounds
  - ranking/ordering of functions
- Amortized analysis
- Recurrences
  - solving recurrences
    - substitution method
    - recursion-tree
    - master method

Midterm

- Sorting
  - insertion sort
  - merge sort
  - quick sort
    - partition function
  - bubble sort
  - heap sort

Midterm

- Divide and conquer
  - divide up the data (often in half)
  - recurse
  - possibly do some work to combine the answer
- Calculating order statistics/medians
- Basic data structures
  - set operations
  - array
  - linked lists
  - stacks
  - queues

Midterm

- Heaps
  - binary heaps
  - binomial heaps
- Search trees
  - BSTs
  - B-trees

Midterm

- Other things to know:
  - run-times (you shouldn't have to look all of them up, though I don't expect you to memorize them either)
  - when to use an algorithm
  - proof techniques
    - look again an proofs by induction

Data structures so far

- When would we use:
  - Arrays
    - get and set particular indices in constant time
  - linked list
    - insert and delete in constant time
  - stack
    - LIFO
  - queue
    - FIFO

Data structures so far

- When would we use:
  - binary heap
    - max/min in log time
  - binomial heap
    - max/min in log time
    - supports the union operation
  - BST
    - search in log time
  - B-Tree
    - search on disk in log disk accesses

Recurrences: three approaches

- **Substitution method**: when you have a good guess of the solution, prove that it's correct
- **Recursion-tree method**: If you don't have a good guess, the recursion tree can help. Then solve with substitution method.
- **Master method**: Provides solutions for recurrences of the form:

$$T(n) = aT(n/b) + f(n)$$

Substitution method

- Guess the form of the solution
- Then prove it's correct by induction

$$T(n) = T(n/2) + d$$

- Halves the input then constant amount of work
- Similar to binary search:

Guess: $O(\log_2 n)$

Proof?

$$T(n) = T(n/2) + d = O(\log_2 n)?$$

Proof by induction!
- Assume it's true for smaller T(k)
- prove that it's then true for current T(n)

$$T(n) = T(n/2) + d$$

- Assume $T(k) = O(\log_2 k)$ for all $k < n$
- Show that $T(n) = O(\log_2 n)$
- From our assumption, $T(n/2) = O(\log_2 n)$:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

- From the definition of O: $T(n/2) \le c \log_2(n/2)$

$$T(n) = T(n/2) + d$$

- To prove that $T(n) = O(\log_2 n)$ we need to identify the appropriate constants:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

i.e. some constant c such that $T(n) \le c \log_2 n$

$$\begin{aligned} T(n) &= T(n/2) + d \\ &\le c\log_2(n/2) + d \\ &\le c\log_2 n - c\log_2 2 + d \\ &\le c\log_2 n \underbrace{-c + d}_{\text{residual}} \\ &\le c\log_2 n \end{aligned}$$

if $c \ge d$ ★

$$T(n) = T(n-1) + n$$

- Guess the solution?
  - At each iteration, does a linear amount of work (i.e. iterate over the data) and reduces the size by one at each step
  - $O(n^2)$
- Assume $T(k) = O(k^2)$ for all $k < n$
  - again, this implies that $T(n-1) \le c(n-1)^2$
- Show that $T(n) = O(n^2)$, i.e. $T(n) \le cn^2$

+

$$T(n) = T(n-1) + n$$
$$\le c(n-1)^2 + n$$
$$= c(n^2 - 2n + 1) + n$$
$$= cn^2 - 2cn + c + n \quad \text{residual}$$
$$\le cn^2$$

if $-2cn + c + n \le 0$

$$-2cn + c \le -n$$
$$c(-2n+1) \le -n$$
$$c \ge \frac{n}{2n-1}$$

which holds for any $c \ge 1$ for $n \ge 1$

$$c \ge \frac{1}{2 - 1/n}$$

+ Changing variables

$$T(n) = 2T(\sqrt{n}) + \log n$$

- Guesses?
- We can do a variable change: let $m = \log_2 n$ (or $n = 2^m$)

$$T(2^m) = 2T(2^{m/2}) + m$$

- Now, let $S(m)=T(2^m)$

$$S(m) = 2S(m/2) + m$$

+ Changing variables

$$S(m) = 2S(m/2) + m$$

- Guess? $S(m) = O(m \log m)$

$$T(n) = T(2^m) = S(m) = O(m \log m)$$

substituting m=log n

$$T(n) = O(\log n \log\log n)$$

+ Recurrences

$$T(n) = 2T(n/3) + d \qquad T(n) = 7T(n/7) + n$$

if $f(n) = O(n^{\log_b a - \varepsilon})$ for $\varepsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$
if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$
if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for $\varepsilon > 0$ and $af(n/b) \le cf(n)$ for $c < 1$
then $T(n) = \Theta(f(n))$

$$T(n) = T(n-1) + \log n \qquad T(n) = 8T(n/2) + n^3$$

Binary Search Trees

- BST – A binary tree where a parent's value is greater than all values in the left subtree and less than or equal to all the values in the right subtree

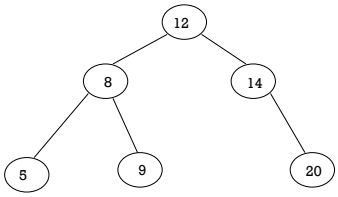
$$leftTree(i) < i \leq rightTree(i)$$

- the left and right children are also binary trees
- Why not?

$$leftTree(i) \leq i \leq rightTree(i)$$

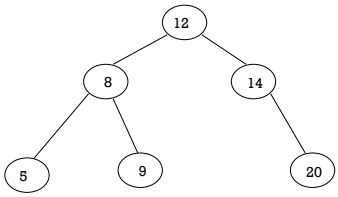
- Can be implemented with with pointers or an array

Example



Visiting all nodes

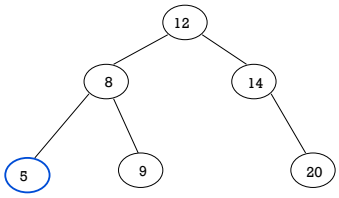
- In sorted order



Visiting all nodes

- In sorted order

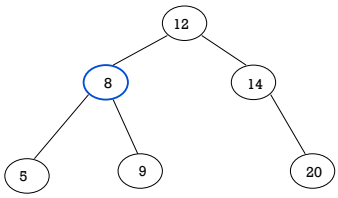
5



Visiting all nodes

- In sorted order

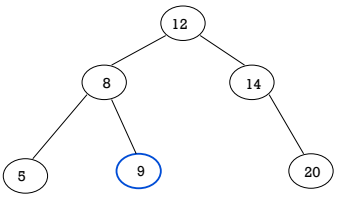
5, 8



Visiting all nodes

- In sorted order

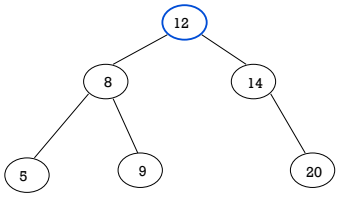
5, 8, 9



Visiting all nodes

- In sorted order

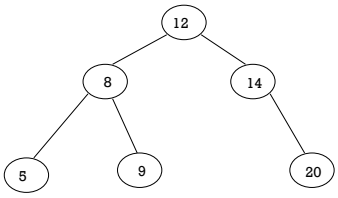
5, 8, 9, 12



Visiting all nodes

- What's happening?

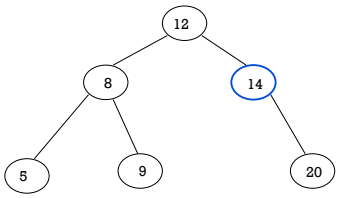
5, 8, 9, 12



Visiting all nodes

- In sorted order

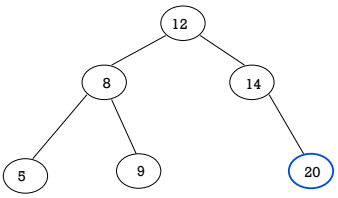
5, 8, 9, 12, 14



Visiting all nodes

- In sorted order

5, 8, 9, 12, 14, 20



Visiting all nodes in order

```
INORDERTREEWALK(x)
1  if x ≠ null
2        INORDERTREEWALK(LEFT(x))
3        print x
4        INORDERTREEWALK(RIGHT(x))
```

Visiting all nodes in order

```
INORDERTREEWALK(x)
1  if x ≠ null
2        INORDERTREEWALK(LEFT(x))
3        print x
4        INORDERTREEWALK(RIGHT(x))
```

any operation

Is it correct?

```
INORDERTREEWALK(x)
1  if x ≠ null
2        INORDERTREEWALK(LEFT(x))
3        print x
4        INORDERTREEWALK(RIGHT(x))
```

- Does it print out all of the nodes in sorted order?

$$left(i) < i \leq right(i)$$

Running time?

```
INORDERTREEWALK(x)
1  if x ≠ null
2        INORDERTREEWALK(LEFT(x))
3        print x
4        INORDERTREEWALK(RIGHT(x))
```

- How much work is done for each call?
- How many calls?
- $\Theta(n)$

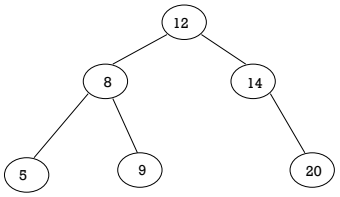
What about?

```
TREEWALK(x)
1  if x ≠ null
2        print x
3        TREEWALK(LEFT(x))
4        TREEWALK(RIGHT(x))
```

Preorder traversal

```
TREEWALK(x)
1  if x ≠ null
2        print x
3        TREEWALK(LEFT(x))
4        TREEWALK(RIGHT(x))
```

12, 8, 5, 9, 14, 20



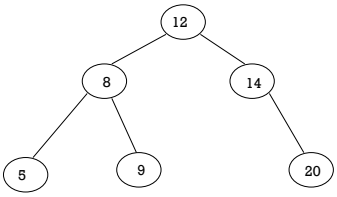
What about?

```
TreeWalk(x)
1  if x ≠ null
2        TreeWalk(Left(x))
3        TreeWalk(Right(x))
4        print x
```

Postorder traversal

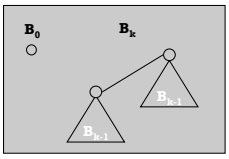
```
TreeWalk(x)
1  if x ≠ null
2        TreeWalk(Left(x))
3        TreeWalk(Right(x))
4        print x
```

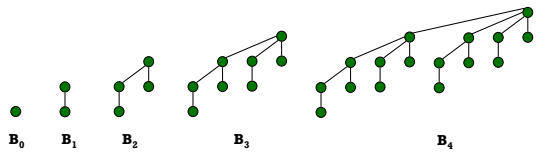
5, 9, 8, 20, 14, 12



Binomial Tree

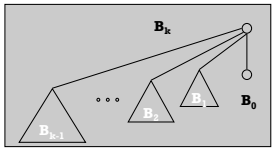
B_k is a binomial tree B_{k-1} with the addition of a left child with another binomial tree B_{k-1}





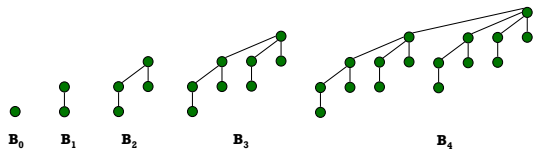
Binomial Tree

Height?



$H(B_o) = 1$

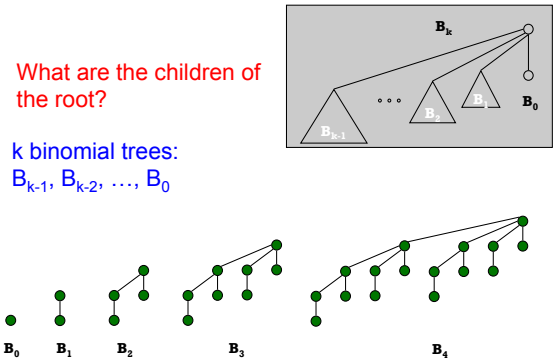
$H(B_k) = 1 + H(B_{k-1}) = k$



Binomial Tree

What are the children of the root?

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

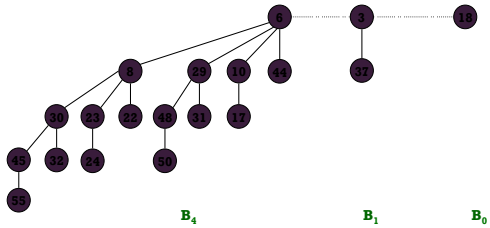


Binomial Heap

- Binomial heap Vuillemin, 1978.

Sequence of binomial trees that satisfy binomial heap property:

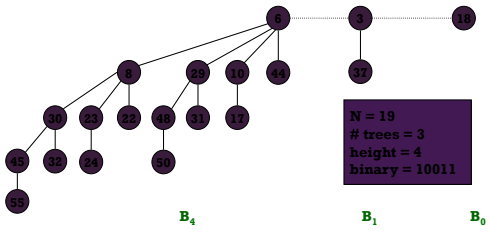
- each tree is min-heap ordered
- top level: full or empty binomial tree of order k
- which are empty or full is based on the number of elements



Binomial Heap: Properties

How many heaps?

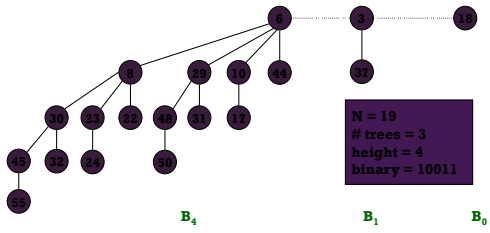
O(log n) – binary number representation



Binomial Heap: Properties

Where is the max/min?

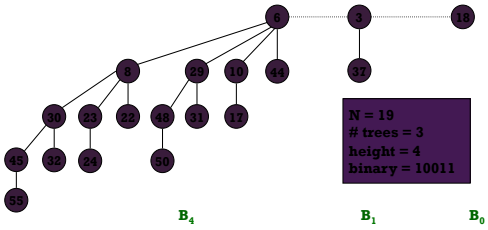
Must be one of the roots of the heaps



Binomial Heap: Properties

Runtime of max/min?

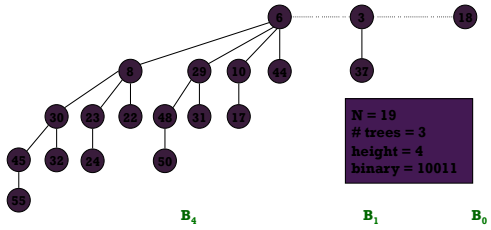
O(log n)



Binomial Heap: Properties

Height?

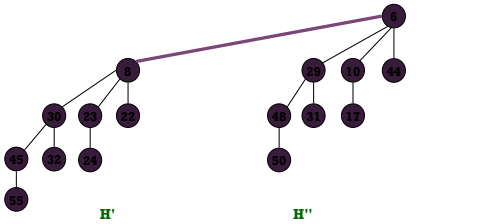
floor($\log_2$ n)
- largest tree = $B_{\log n}$
- height of that tree is log n



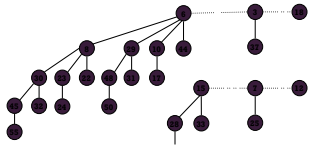
Binomial Heap: Union

- 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)



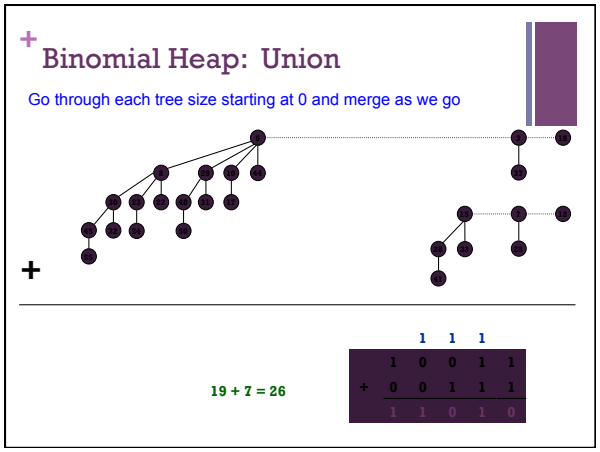
Binomial Heap: Union



What if they're not they're not the simple heaps of size 2^K?

Binomial Heap: Union

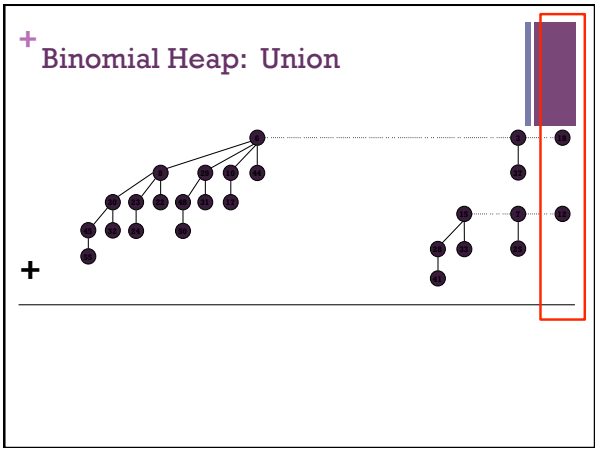
Go through each tree size starting at 0 and merge as we go



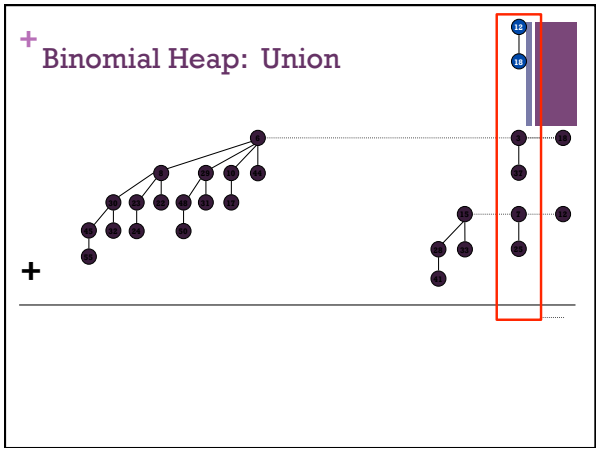
19 + 7 = 26

		1	1	1	
	1	0	0	1	1
+	0	0	1	1	1
	1	1	0	1	0

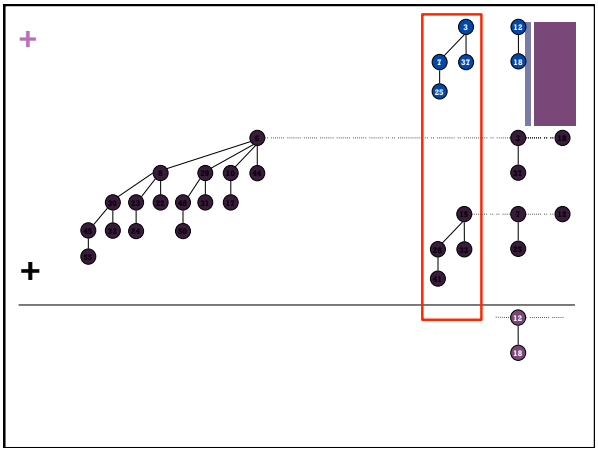
Binomial Heap: Union

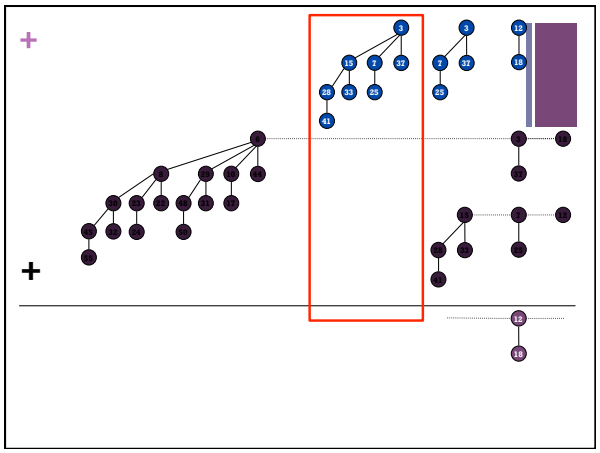


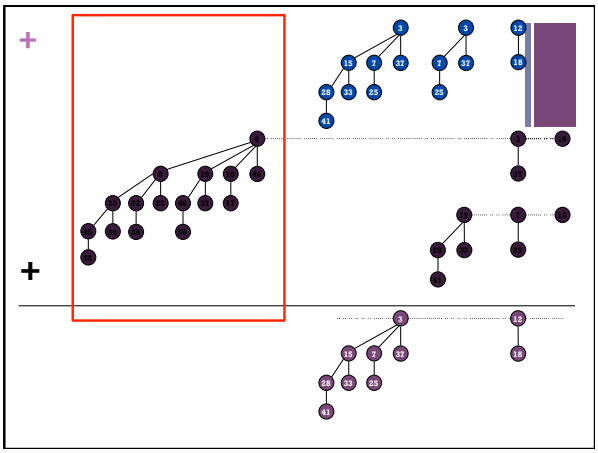
Binomial Heap: Union

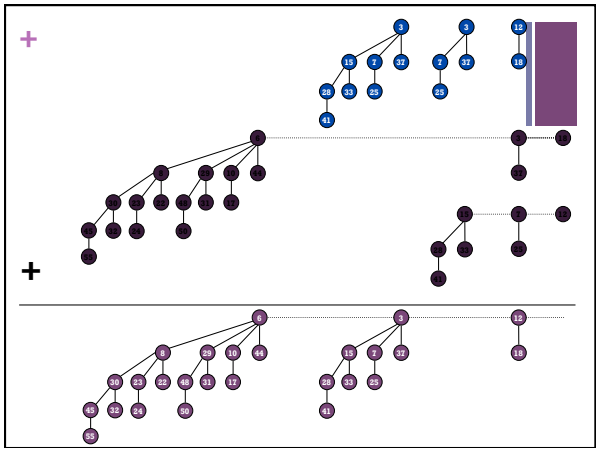


Binomial Heap: Union









Binomial Heap: Union

Analogous to binary addition

- Running time?
  - Proportional to number of trees in root lists 2 $O(\log_2 N)$
  - $O(\log N)$

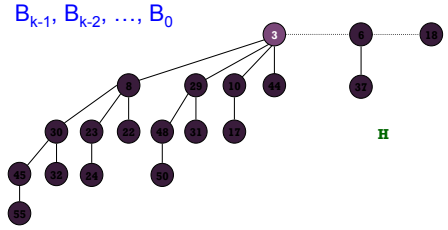
19 + 7 = 26

		1	1	1	
	1	0	0	1	1
+	0	0	1	1	1
	1	1	0	1	0

Binomial Heap: Delete Min/Max

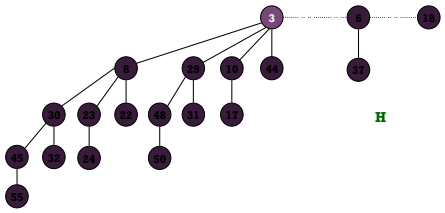
We can find the min/max in O(log n).
How can we extract it?

Hint: B_k consists of binomial trees: $B_{k-1}, B_{k-2}, \ldots, B_0$



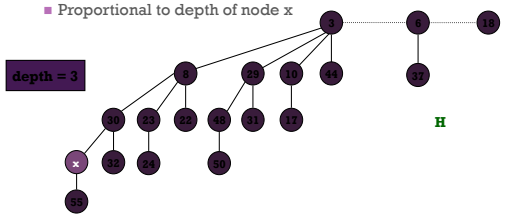
Binomial Heap: Delete Min

- Delete node with minimum key in binomial heap H.
  - Find root x with min key in root list of H, and delete
  - H' ← broken binomial trees
  - H ← Union(H', H)



Binomial Heap: Decrease Key

- Just call Decrease-Key/Increase-Key of Heap
  - Suppose x is in binomial tree B_k
  - Bubble node x up the tree if x is too small
- Running time: O(log N)
  - Proportional to depth of node x

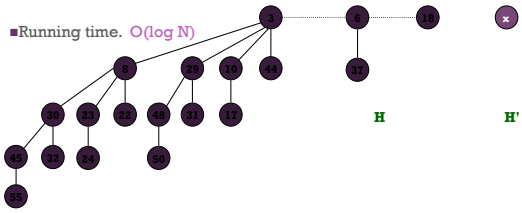


Binomial Heap: Delete

- Delete node x in binomial heap H
  - Decrease key of x to -∞
  - Delete min
- Running time: O(log N)

Binomial Heap: Insert

- Insert a new node x into binomial heap H
  - H' ← MakeHeap(x)
  - H ← Union(H', H)
- Running time. O(log N)



Heaps

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
BUILD-HEAP	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
INSERT	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$
MAXIMUM	$\Theta(1)$	$O(\log n)$	$\Theta(1)$
EXTRAC-MAX	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$
UNION	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$
INCREASE-ELEMENT	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
DELETE	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$

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