http://recursivelyrecursive.wordpress.com/

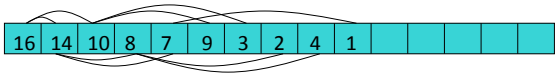
Heaps O' Fun

CS 140 HMC - Jim Boerkoel

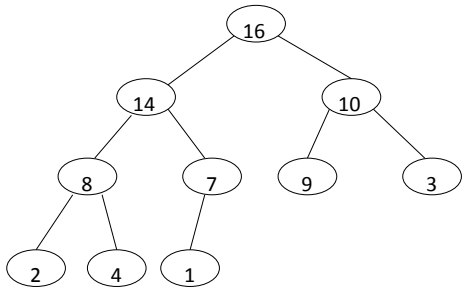
Today's Goals

- Mergeable Heaps
  - Review Binary Heaps
  - Linked-lists
  - Binomial Trees
  - Binomial Heaps
- Red-Black
  - Prove balance
  - Look at insert / delete

Binary heap representations

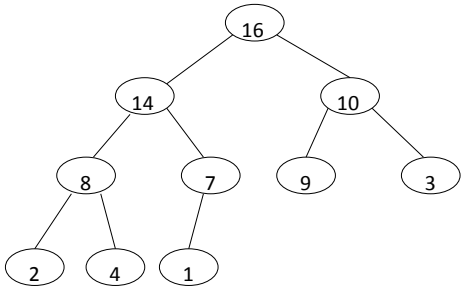


1 2 3 4 5 6 7 8 9 10



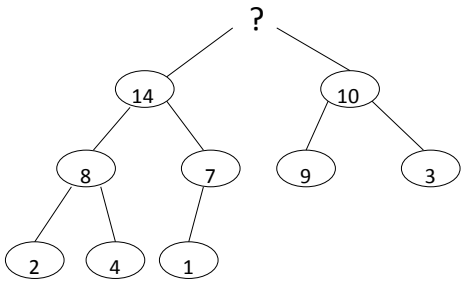
ExtractMax

Return and remove the largest element in the set



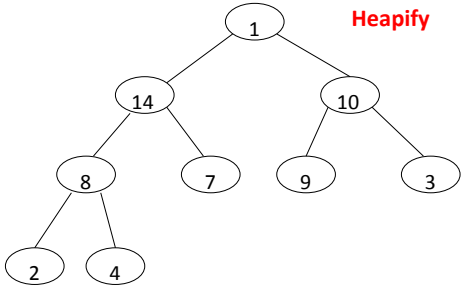
ExtractMax

Return and remove the largest element in the set



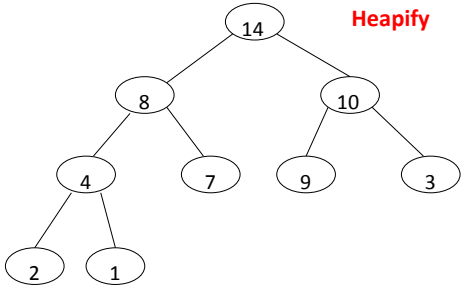
ExtractMax

Return and remove the largest element in the set



ExtractMax

Return and remove the largest element in the set



ExtractMax

Return and remove the largest element in the set

```
EXTRACT-MAX(A)
1  if heap-size[A] < 1
2          error
3  max ← A[1]
4  A[1] ← A[HEAP-SIZE[A]] − 1
5  heap-size[A] ← heap-size[A] − 1
6  HEAPIFY(A, 1)
7  return max
```

Mergeable Heap

A mergeable heap is any data structure that supports the following five operations, in which each element has a key:

- **MakeHeap()** - creates and returns a new heap containing no elements.
- **Insert(H,x)** - inserts element x, whose key has already been filled in, into heap H.
- **Minimum(H)** - returns a pointer to the element in heap H whose key is minimum.
- **Extract-Min(H)** - deletes the element from heap H whose key is minimum, returning a pointer to the element.
- **Union(H_1, H_2)** - creates and returns a new heap that contains all the elements of heaps H1 and H2. Heaps H1 and H2 are "destroyed" by this operation.
- ALSO: Increase-Element, Delete

Binary heaps

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

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

Mergeable heaps

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

(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

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

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

concatenate the arrays and then call Build-Heap

Worksheet:
(Doubly) linked-list heap

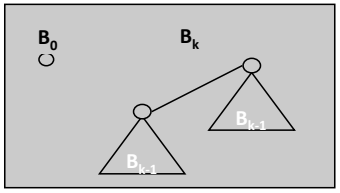
Procedure	Binary heap (worst-case)	Linked-list
BUILD-HEAP	$\Theta(n)$	
INSERT	$\Theta(\log n)$	
MAXIMUM	$\Theta(1)$	
EXTRAC-MAX	$\Theta(\log n)$	
UNION	$\Theta(n)$	
INCREASE-ELEMENT	$\Theta(\log n)$	
DELETE	$\Theta(\log n)$	

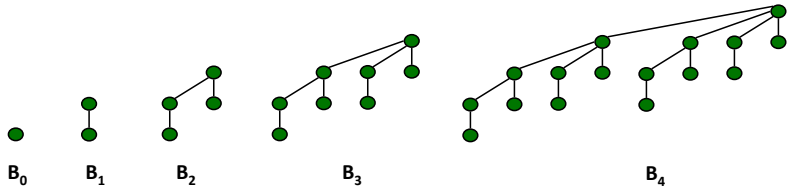
(adapted from Figure 19.1, pg. 456 [1])

Adapted from:
Kevin Wayne

Binomial Tree

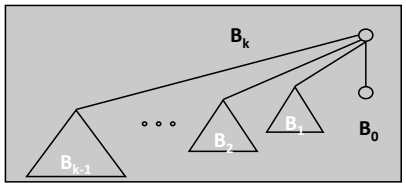
B_k: a binomial tree B_{k-1} with the addition of a left child with another binomial tree B_{k-1}

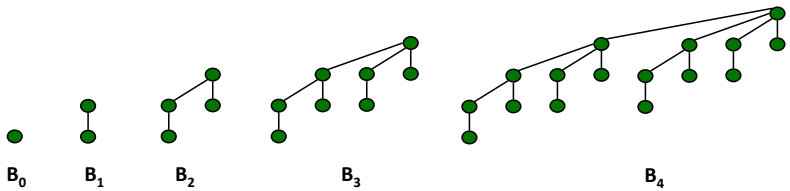




Binomial Tree

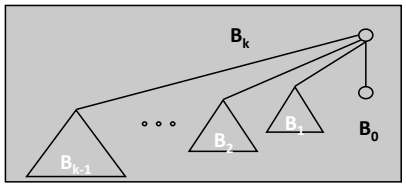
Number of nodes with respect to k?

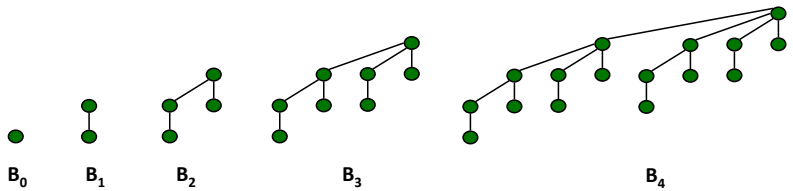




Binomial Tree

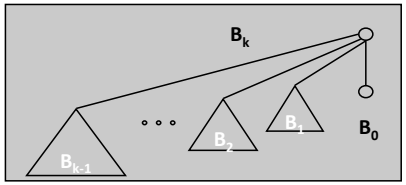
Height?

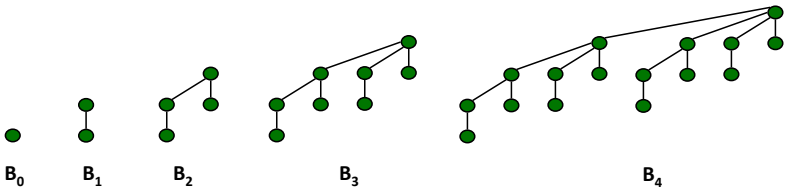




Binomial Tree

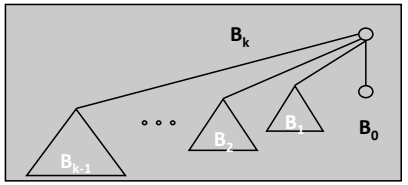
Degree of root node?

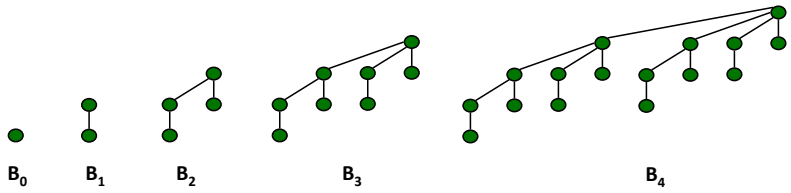




Binomial Tree

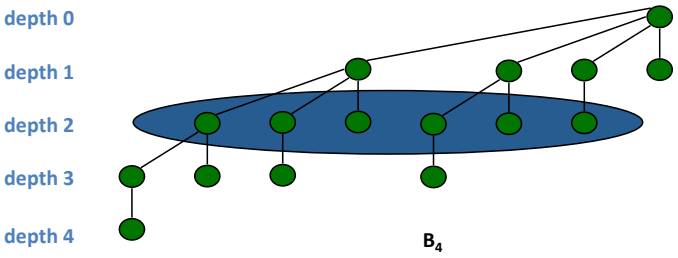
What are the children of the root?





Binomial Tree

Why is it called a binomial tree?

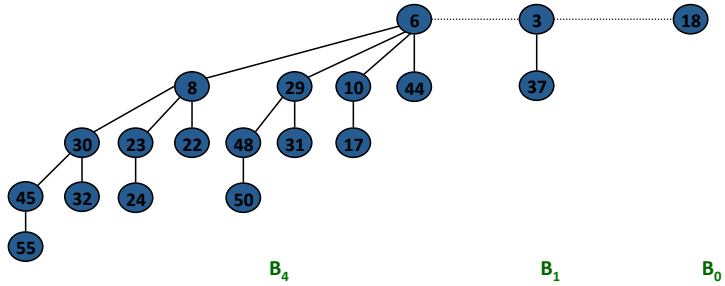


Binomial Heap

•Binomial heap Vuillemin, 1978.

Sequence of binomial trees that satisfy binomial heap property:

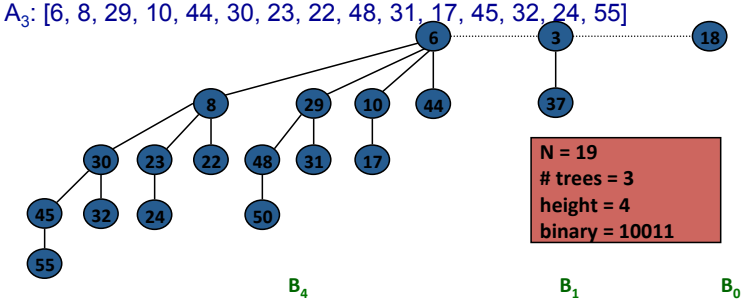
- each tree is min-heap ordered
- top level: full or empty binomial tree of order k
- which are empty or full is based on the number of elements



Binomial Heap

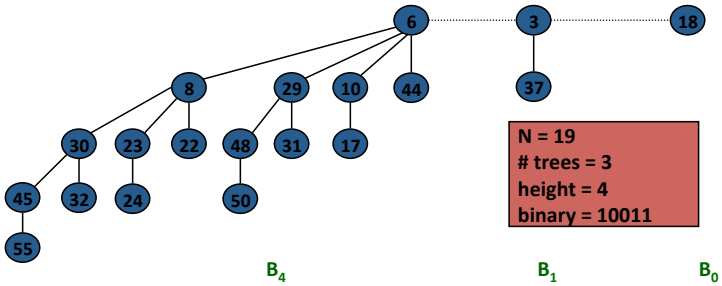
Represented as a series of arrays (each representing a min-heap)

A_0: [18]
A_1: [3, 7]
A_2: empty
A_3: empty
A_3: [6, 8, 29, 10, 44, 30, 23, 22, 48, 31, 17, 45, 32, 24, 55]



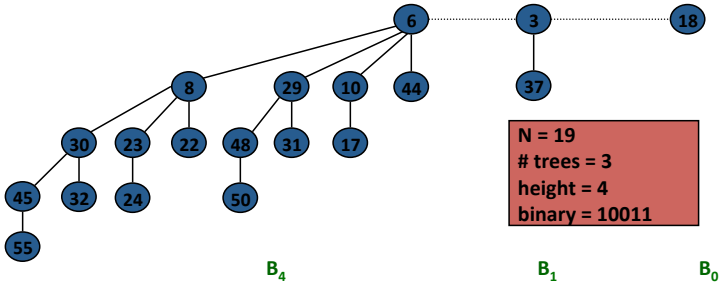
Binomial Heap: Properties

How many heaps?



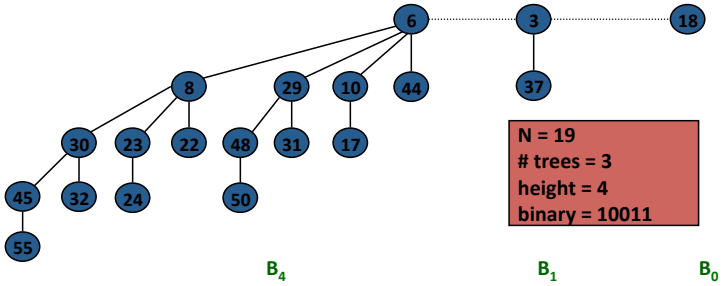
Binomial Heap: Properties

Where is the max/min?



Binomial Heap: Properties

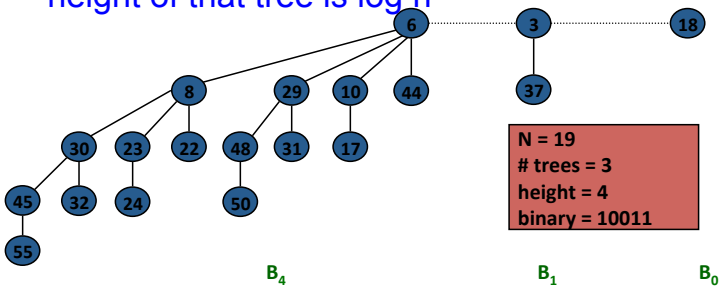
Runtime of max/min?



Binomial Heap: Properties

Height?

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

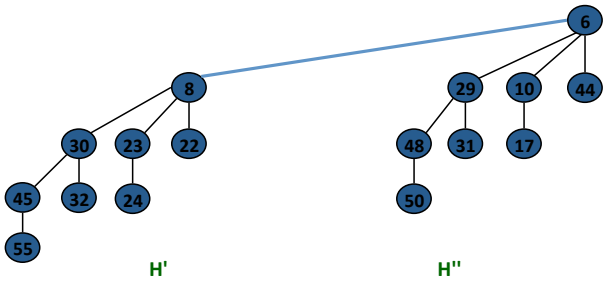


Binomial Heap: Union

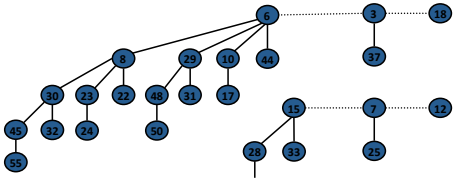
•How can we merge two binomial tree heaps of the same size (2^k)?

- connect roots of H' and H''
- choose smaller key to be root of H

Runtime?



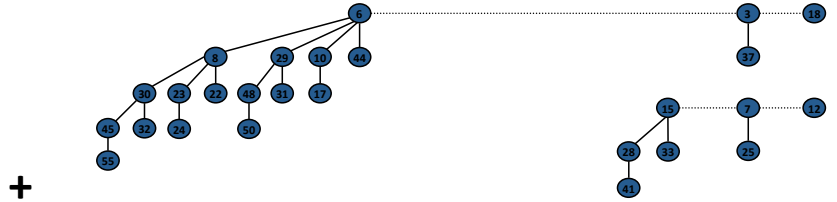
Binomial Heap: Union



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

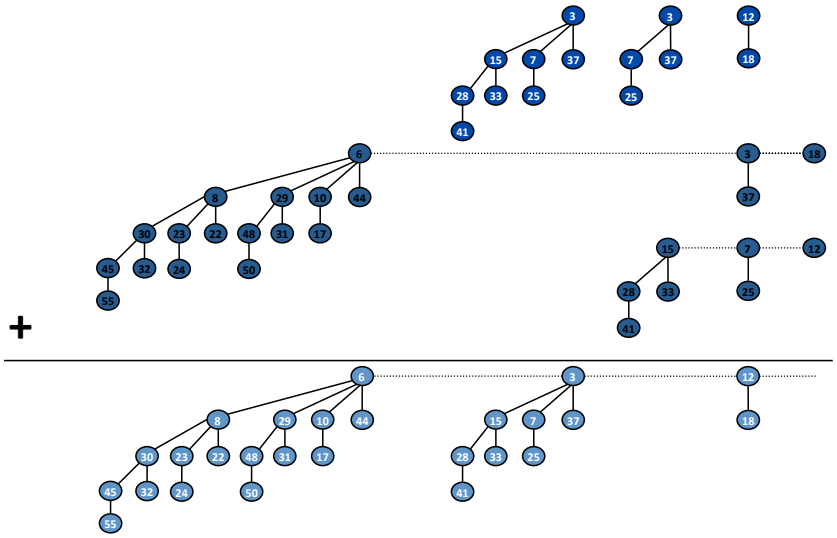
Binomial Heap: Union

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



19 + 7 = 26

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



Binomial Heap: Union

Analogous to binary addition

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

19 + 7 = 26

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

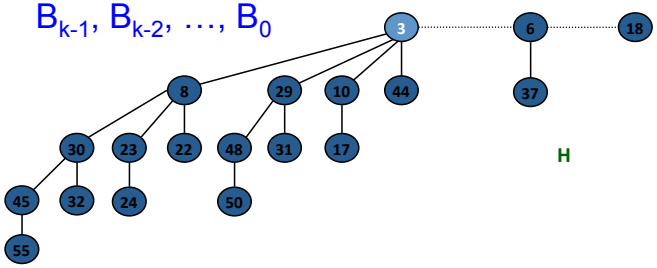
Binomial Heap: Delete Min/Max

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

How can we extract it?

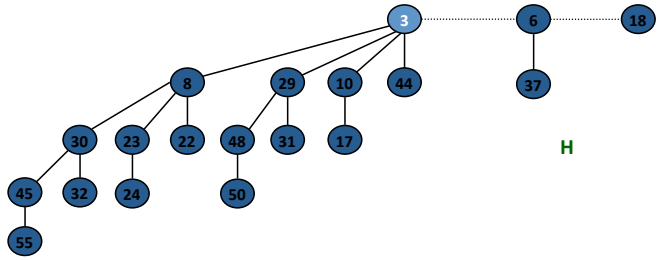
Hint: B_k consists of binomial trees:

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



Binomial Heap: Delete Min

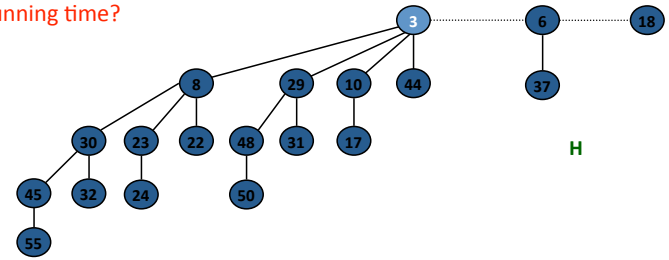
- Delete node with minimum key in binomial heap H.
  - Find root x with min key in root list of H, and delete
  - H' ← broken binomial trees
  - H ← Union(H', H)



Binomial Heap: 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)

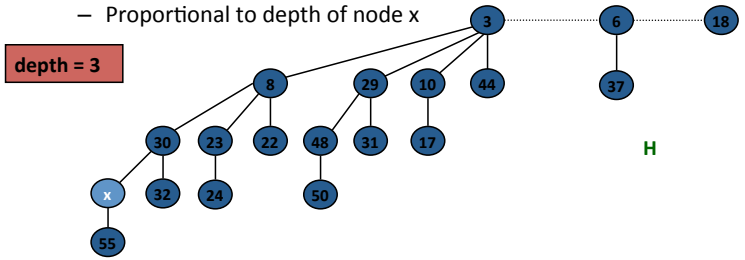
•Running time?



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:
- Proportional to depth of node x



Binomial Heap: Delete

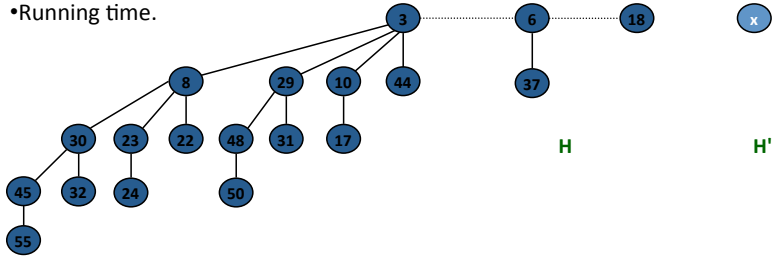
•Delete node x in binomial heap H
- Decrease key of x to -∞
- Delete min

•Running time:

Binomial Heap: Insert

•Insert a new node x into binomial heap H
- H' ← MakeHeap(x)
- H ← Union(H', H)

•Running time.



Build-Heap

•Call insert n times

Runtime?

Can we get a tighter bound?

Build-Heap

•Call insert n times

times cost

Consider inserting *n* numbers

- **how many times will B_0 be empty?**
- **how many times will we need to merge with B_0?**
- **how many times will we need to merge with B_1?**
- **how many times will we need to merge with B_2?**
- **…**
- **how many times will we need to merge with $B_{\log n}$?**

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])

BST refresher

- Data structures that support many dynamic-set operations
- Can be used as a both a dictionary and as a priority queue
- Basic operations take time proportional to the height of the tree
  - Complete binary tree with *n* nodes: worst case Θ(log *n*)
  - For linear chain of *n* nodes: worst case Θ(*n*)
- Red-black trees
  - Variation of binary search tree
  - Balanced: height is *O*(log *n*)
  - Operations take *O*(log *n*) in the worst case

Red-Black Tree

- A variation of binary search trees
- ***Balanced***: height is $O(\log n)$
- Operations will take $O(\log n)$ in the worst case

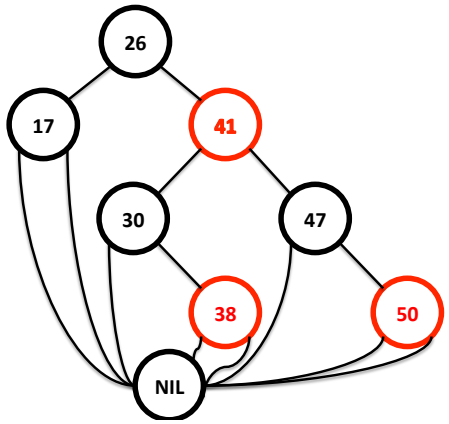
Red-black tree

- A **red-black** tree is a binary search tree +1 bit per node: a *color* attribute, which is either red or black
- All leaves are empty and colored black
  - Use single black node, *T.nil*, for root and all leaves of tree
- Inherits all other properties of a binary search tree

Red-Black Tree Properties

1. Every node is either red or black.
2. The root is black.
3. Every (null) leaf is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

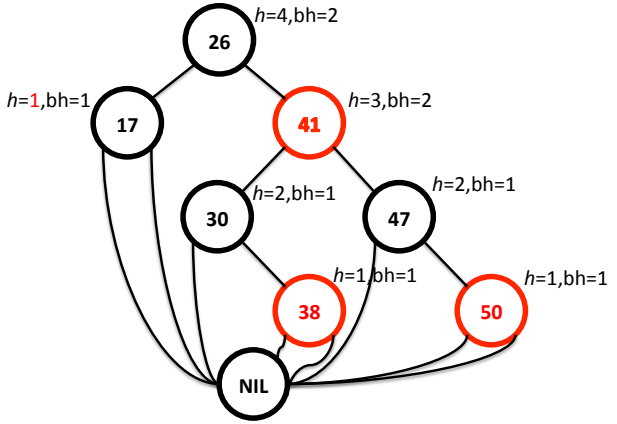
Red-Black Tree Example



Height of a Red-Black Tree

- **Height of a node:** is the number of edges in a longest path to a leaf
- **Black-height of a node *x***: bh(*x*) is the number of black nodes (including NIL) on the path from *x* to a leaf, not including *x*.
  - Well-defined due to property 5

Red-Black Tree Example



Worksheet: Prove

- **Claim 1**: Any node with height *h* has black-height ≥ *h*/2.

Worksheet: Prove

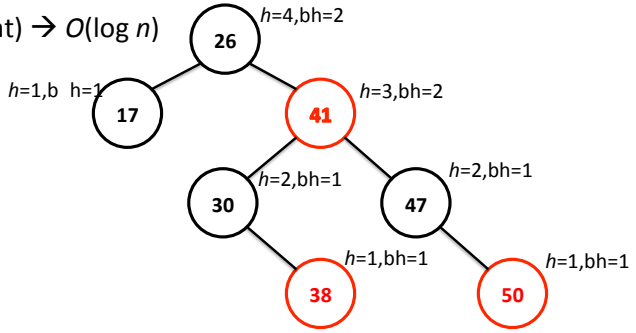
- **Claim 2**: The subtree rooted at node *x* contains ≥ $2^{bh(x)\ -1}$

The key to success

- **Lemma:** A red black tree with n internal nodes has height $\leq 2\log(n+1)$.
- **Proof:** Let h and b be the height and black-height of the root. Then,

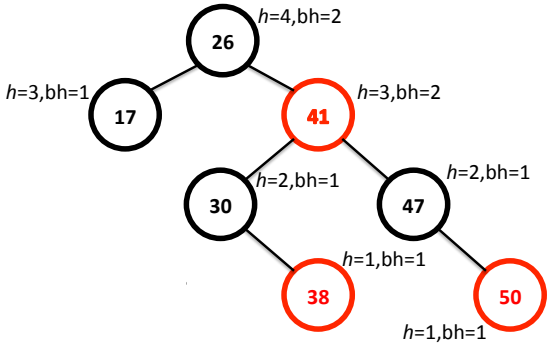
Operations on Red-black trees

- Non-modifying operations run as before
  - Minimum, Maximum, Successor, Predecessor, Search
  - O(height) → O(log n)



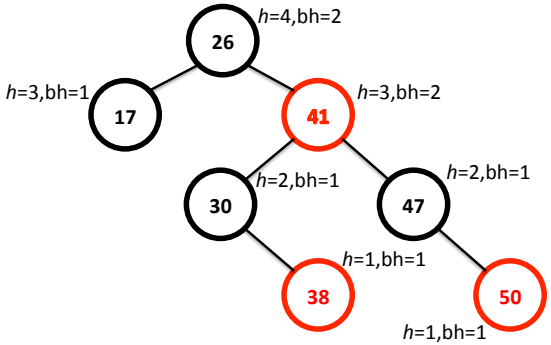
Inserting

- When inserting a new key, e.g. 40, should it be:
  - **Red?**
  - **Black?**



Deleting

- When deleting an existing key, what happens if it's:
  - **Red?**
  - **Black?**



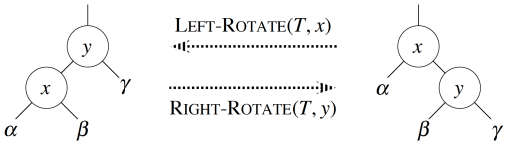
Rotations

- The basic tree-restructuring operation
- Needed to maintain red-black trees as balanced binary search trees
- Changes only the local pointer structure.
- Does not impact binary tree property
- Both left and right rotations (inverse of each other)
- Takes as input: red-black tree and a node within the tree

Rotate Analysis

```
LEFT-ROTATE(T, x)
y = x.right
x.right = y.left
if y.left ≠ T.nil
    y.left.p = x
y.p = x.p
if x.p == T.nil
    T.root = y
elseif x == x.p.left
    x.p.left = y
else x.p.right = y
y.left = x
x.p = y
```

- Time:
- Used in AVL and splay trees
- Can also talk about rotating on edge, rather than node

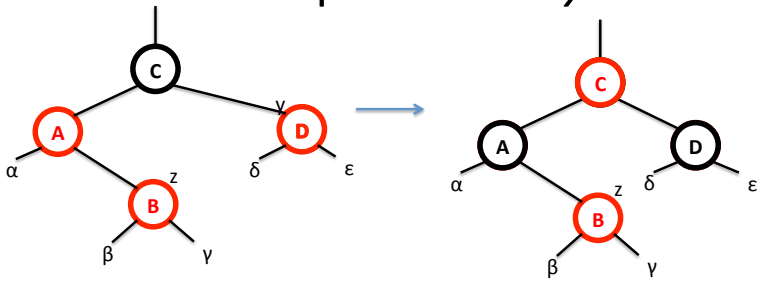



```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

Insertion

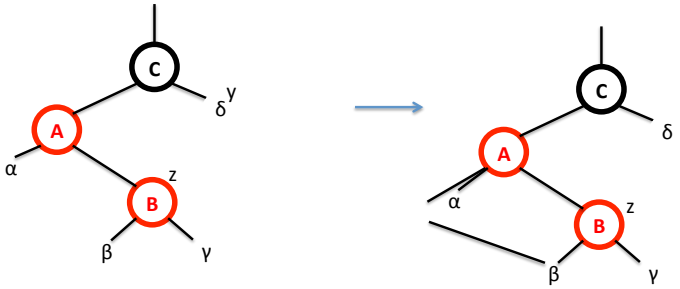
- Idea:
  - Run insertion as before
  - Set z to red
  - Fix possible violations:
  1.
  2.
  3.
  4.
  5.

Insert-Fixup: Case 1 – *y* is red



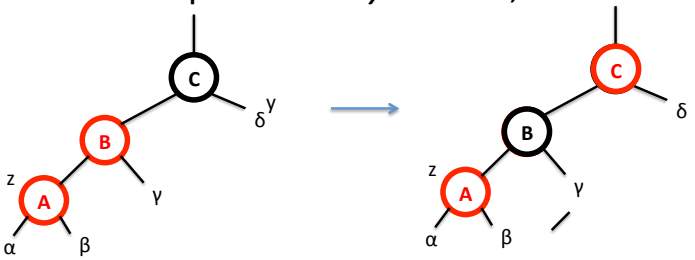
- *z.p.p* must be black
- Make *z.p* and *y* black
- Make *z.p.p* red
- Set *z.p.p* as the new *z*

Insert-Fixup: Case 2 – *y* is black, z is right child



- Rotate *z* and *z.p*

Insert-Fixup: Case 3 – *y* is black, z is left child



- Make *z.p* black and *z.p.p* red
- Rotate on *z.p.p*
- No longer have 2 reds in a row
- *z.p* is now black → no more iterations

RB-Insert Analysis

- *O*(log *n*) time for original insert
- Within RB-Insert-Fixup
  - Each iteration takes *O*(1) time
  - Each iteration either fixes the tree or moves *z* up two levels
  - There are at most 2 rotations overall
  - *O*(log *n*) levels → *O*(log *n*) time

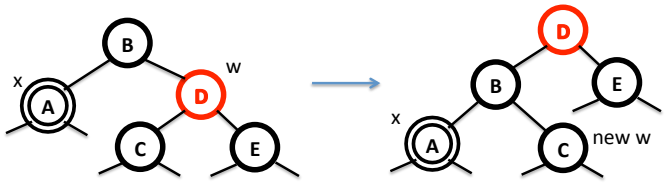
```
RB-DELETE(T, z)
  y = z
  y-original-color = y.color
  if z.left == T.nil
      x = z.right
      RB-TRANSPLANT(T, z, z.right)
  elseif z.right == T.nil
      x = z.left
      RB-TRANSPLANT(T, z, z.left)
  else y = TREE-MINIMUM(z.right)
      y-original-color = y.color
      x = y.right
      if y.p == z
          x.p = y
      else RB-TRANSPLANT(T, y, y.right)
          y.right = z.right
          y.right.p = y
      RB-TRANSPLANT(T, z, y)
      y.left = z.left
      y.left.p = y
      y.color = z.color
  if y-original-color == BLACK
      RB-DELETE-FIXUP(T, x)
```

Deletion

- Idea:
  - Run deletion (mostly) as before
  - If y's original color black, fix possible violations:

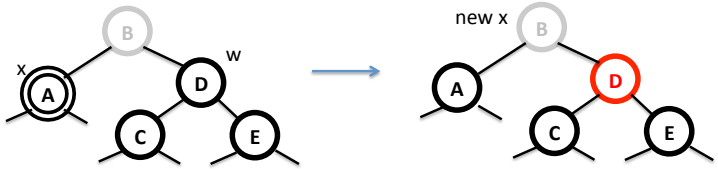
1.
2.
3.
4.
5.

Delete-Fixup: Case 1 – *w* is red



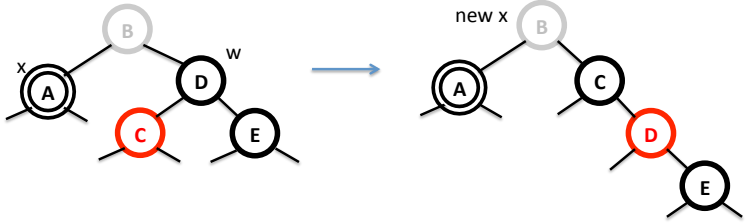
- W must have black children
- Make w black and x.p red
- Left rotate on x.p
- New sibiling of x must now be black
- Goto case 2,3, or 4

Delete-Fixup: Case 2 – *w* is black & both children are black



- Take 1 black off *x* and *w*
- Move black to *x.p*
- Do next iteration with *x.p* as new *x*
- If *new x* had been red, break and *new x* to black

Delete-Fixup: Case 3 – *w* is black & *w*'s left child is red, right child is black



- Make *w* red and *w*'s left child black
- Right rotate on *w*
- → case 4

Delete-Fixup: Case 4 – *w* is black & *w*'s right child is red

- Make *w* be *x.p*'s color
- Make *x.p* black and *w*'s right child black
- Left rotate on *w*
- Remove extra black on x
- Set *x* to root

RB-Delete Analysis

- $O(\log n)$ time to delete
- RB-Delete-Fixup
  - Only Case 2 iterates
    - (all others progress to next case or halt)
    - *X* moves up one level
  - Each of cases 1, 3, and 4 has 1 rotation → ≤ 3 rotations in all
  - Hence $O(\log n)$ overall