Binomial & Fibonacci Heaps and Amortized Analysis

Main Reference: Chapter 19, Sections 17.1-17.3 and Chapter 20

- Motivation
- Mergable Heaps: Binomial Heaps
- An Introduction to Amortized Analysis
- Fibonacci Heaps

Most of this course deals with how to solve optimization problems using efficient algorithms. This section is different. It focuses on how to improve known algorithms by designing special data structures.

In the course of doing this we will introduce amortized analysis, a more sophisticated way of analyzing algorithms than just looking at the worst case time of individual operations.

A mergeable (Min) heap is a data structure supporting the following operations. (Heaps and nodes are passed and returned via pointers).

Make-Heap() creates & returns a new empty heap

Insert(H, x) inserts x into H

Minimum(H) returns pointer to smallest key in H

Extract-Min(H) removes minimum item from H and returns it to caller

Union(H_1, H_2) merges H_1 and H_2 into a new heap (destroying H_1, H_2 in process)

Decrease-Key(H, x, k) Reduces value of node x in H to key value k (assumes that old value was not less than k)

Delete(H, x) Removes node x from H

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
Extract-Min	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
Union	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
Decrease-Key	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
Delete	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Any combination of n *Insert*s followed by n *Extract-Min*s must take at least $\Omega(n \log n)$ time. (Why?)

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
Extract-Min	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
Union	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
Decrease-Key	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
Delete	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Dijkstra's algorithm for solving the single-source shortest path problem and Prim's similar algorithm for constructing a minimum spanning-tree use

$|V|$ *Insert*s; $|V|$ *Extract-Min*s; $|E|$ *Decrease-Key*s.

We usually learn that these algorithms require $\Theta((|E|+|V|)\log|V|)$ time.

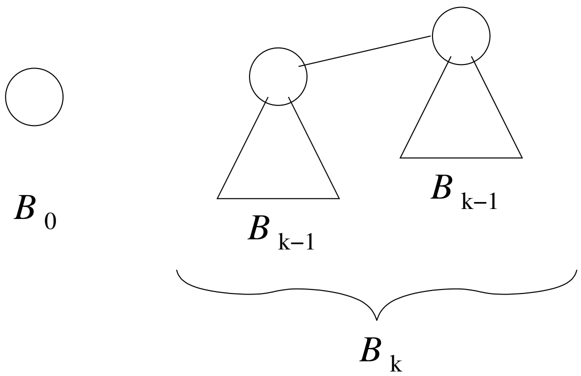
The *Insert*s and *Extract-Min*s do require $\Theta(|V|\log|V|)$. But, by the end of this section, we will see how to design a data structure where the $|E|$ *Decrease-Key*s only use $O(|E|)$ *amortized* time, leading to a total running time of only $\Theta(|V|\log|V|+|E|)$

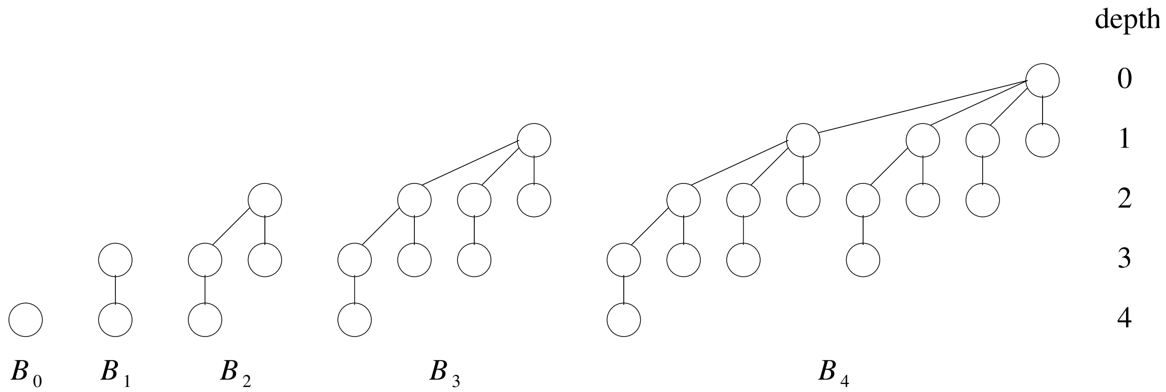
We will start by introducing *binomial heaps*, a variation on standard binary heaps. Binomial heaps use a standard trick that has proved very successful in the design of data-structures for dynamic data; Instead of maintaining *one* data structure, they maintain a *collection* of them. Whenever so much data has been added that our collection becomes “too large and messy” it is tided up, (essentially charging the cost to the previously added data).

Binomial heaps by themselves don’t provide enough power to improve the running times of Dijkstra’s and Prim’s algorithms. We therefore introduce *amortized analysis*. This is a useful technique that, given a sequence of operations, permits sharing the cost of a single expensive operation with many other cheaper ones.

We will then use this new way of looking at things to modify Binomial Heaps into Fibonacci Heaps, which do have good *amortized* time bounds.

A *binomial heap* is a collection of heap-ordered *binomial trees* so we must start with:

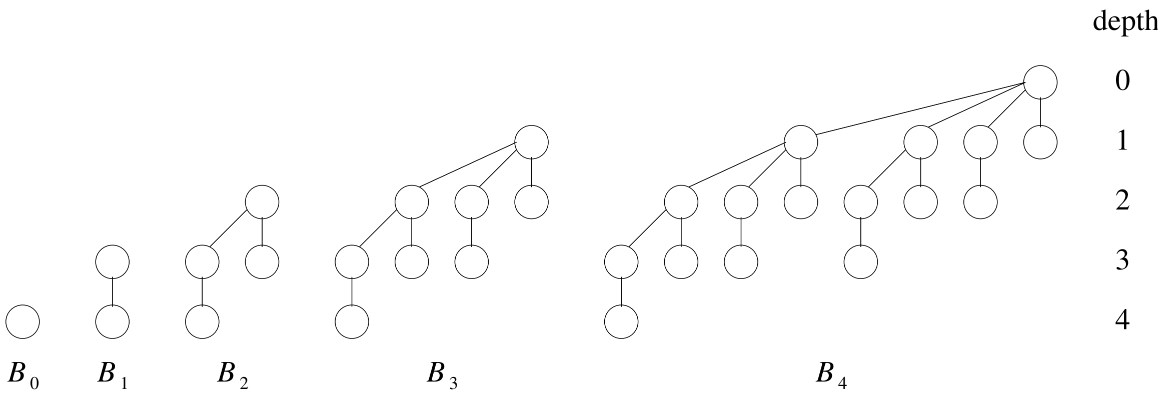




Definition: A binomial tree B_k is an ordered tree such that:

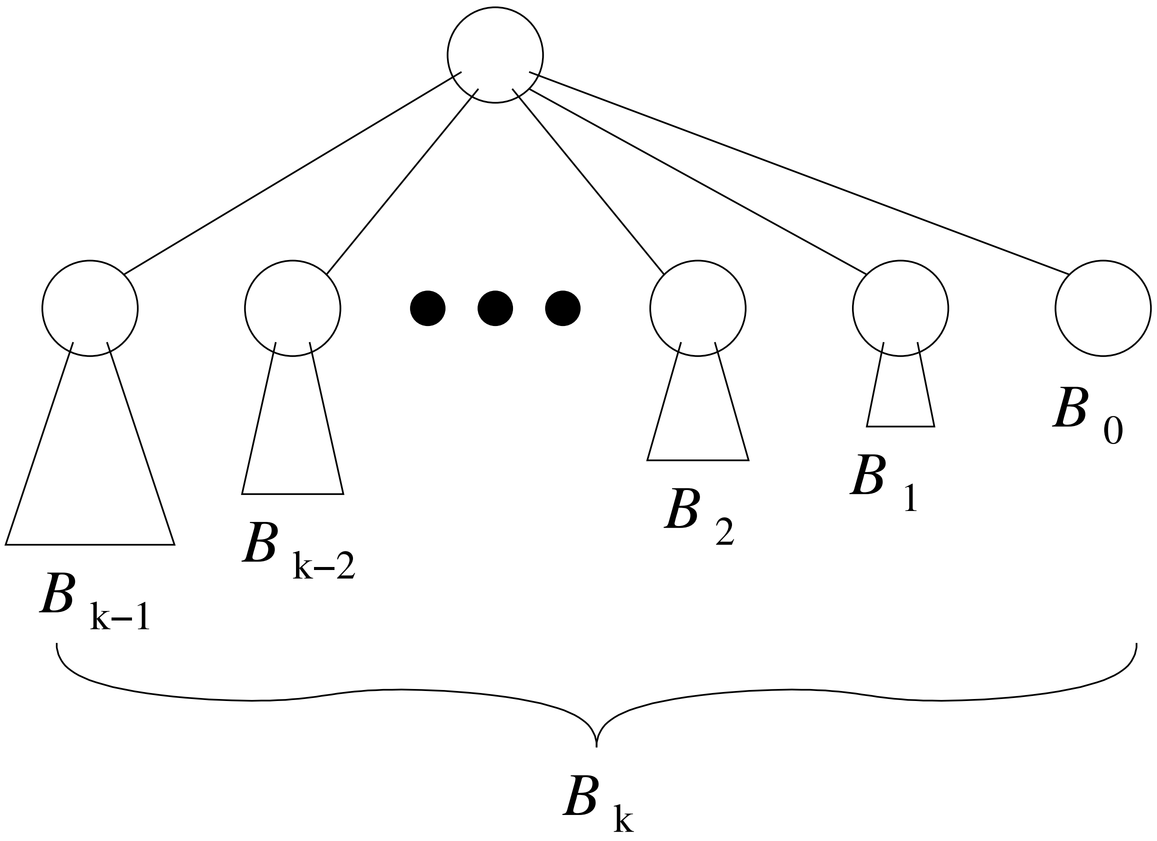
(i) B_0 is a single node.

(ii) B_k is two B_{k-1} trees with the root of the 1st being the leftmost child of the 2nd.



We need the following simple properties of B_k :

- It contains 2^k nodes.
- It has height k.
- It has exactly $\binom{k}{i}$ nodes at depth i.
- The root has degree k. The children of the root, from left to right, are the roots of $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$.
- The maximum degree of *any* node in an n node binomial tree is $\log_2 n$.



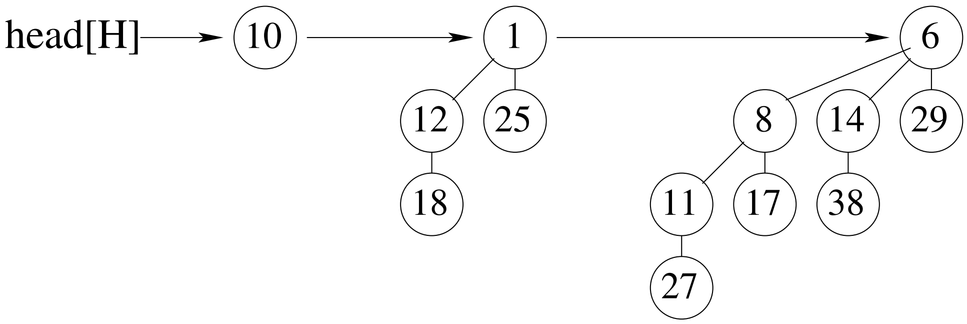
The root has degree k.

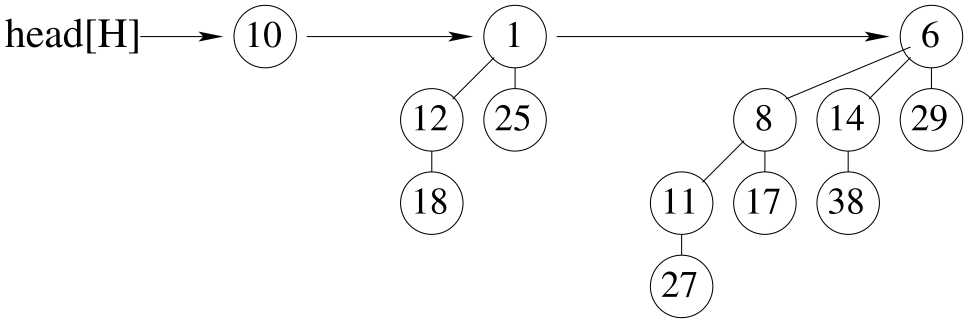
The children of the root, from left to right, are roots of $B_{k-1}, B_{k-2}, \ldots, B_1, B_0$.

A *Binomial heap* H is a set of *Binomial trees* that satisfies the *Binomial heap properties:*

- Each binomial tree obeys the *min-heap property:* the key of a node is $\geq$ key of its parent

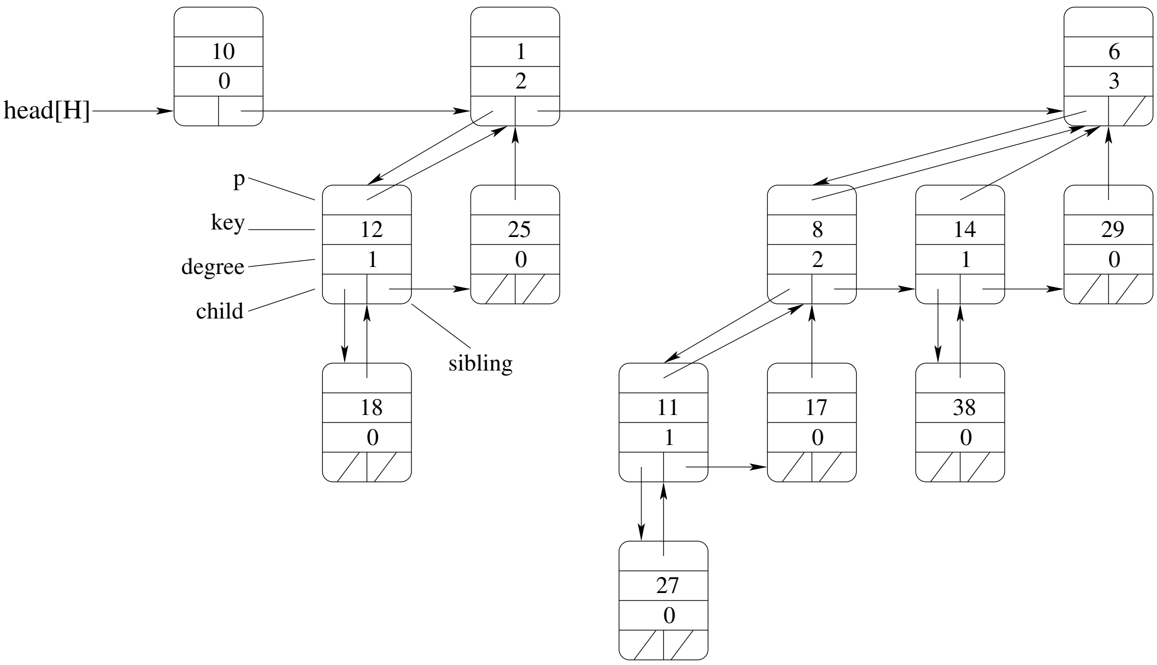
- For every integer k there is at most one B_k in H.



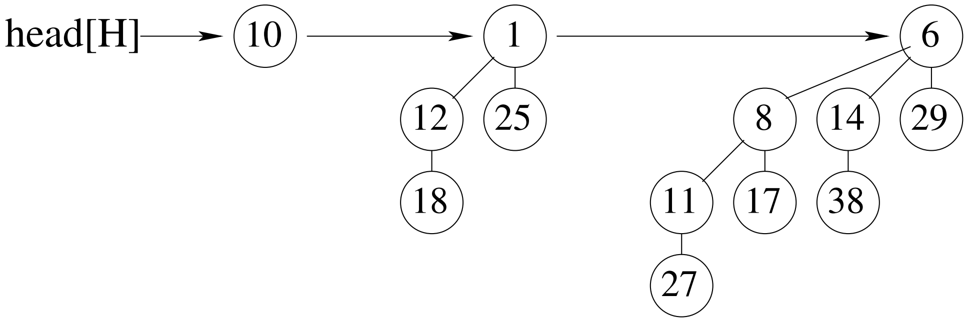


In practice, every node will contain a pointer to its: parent, right sibling, leftmost child

It will also know how many children it has (degree). Nodes in a sibling list will be sorted by degree.



Operations on Binomial Heaps



Make-Heap():
Creating all of the pointers can be done in $O(1)$ time.

Minimum(H):
The minimum must be in some root in the top list.
If there are n nodes in the heap there are at most $\lg n$ roots at the top,
at most one each of degree $0, 1, 2, \ldots, \lfloor \lg n \rfloor$,
so this can be found in $O(\lg n)$ time.

Union(H_1, H_2) is the most sophisticated of the binomial heap operations.

Let A_i (B_i) be unique b. tree of degree i in H_1 (H_2), If trees don't exists, set $A_i, B_i = \emptyset$.

Note that two binomial trees X_i and Y_i, both of degree i, can be merged together in $O(1)$ time to create a binomial tree of degree $i + 1$.

In the Union(H_1, H_2) we will create, C_i will be the unique binomial tree of degree i, if it exists.

Let $k = \lg(|H_1 + |H_2|)$. (limit on size of i).
For $i = 0$ **to** k, $\quad C_i = \emptyset$.
For $i = 0$ **to** k
  If all of $A_i, B_i, C_i \neq \emptyset$
    Link A_i, B_i to form C_{i+1}.
  If exactly two of $A_i, B_i, C_i \neq \emptyset$
    Link those two together to form C_{i+1}. Set $C_i = \emptyset$.
  If exactly one of $A_i, B_i, C_i \neq \emptyset$
    Set C_i to be that binomial tree.
Link the C_i together to form the merged binomial heap.

Let $k = \lg(|H_1 + |H_2|)$. (limit on size of i).

For $i = 0$ **to** k, $\quad C_i = \emptyset$.

For $i = 0$ **to** k

  If all of $A_i, B_i, C_i \neq \emptyset$

    Link A_i, B_i to form C_{i+1}.

  If exactly two of $A_i, B_i, C_i \neq \emptyset$

    Link those two together to form C_{i+1}. Set $C_i = \emptyset$.

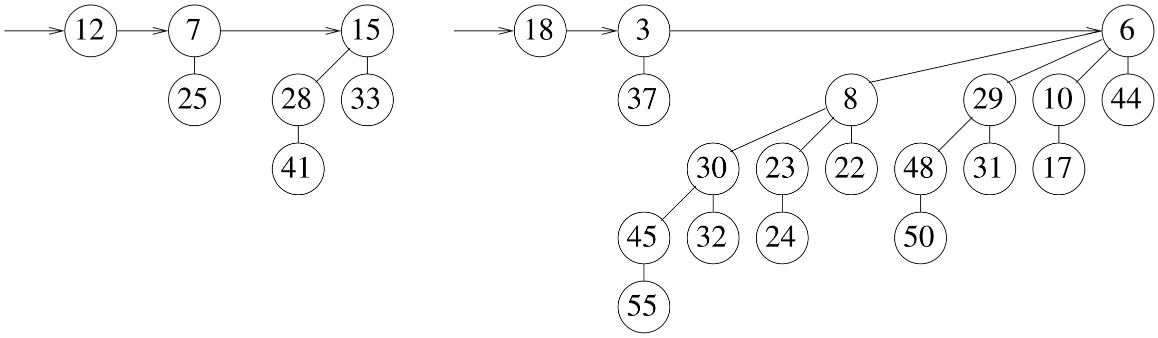
  If exactly one of $A_i, B_i, C_i \neq \emptyset$

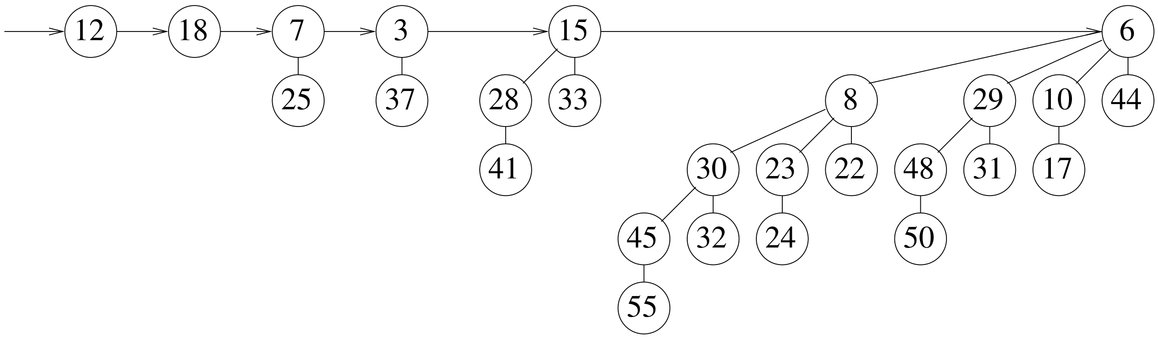
    Set C_i to be that binomial tree.

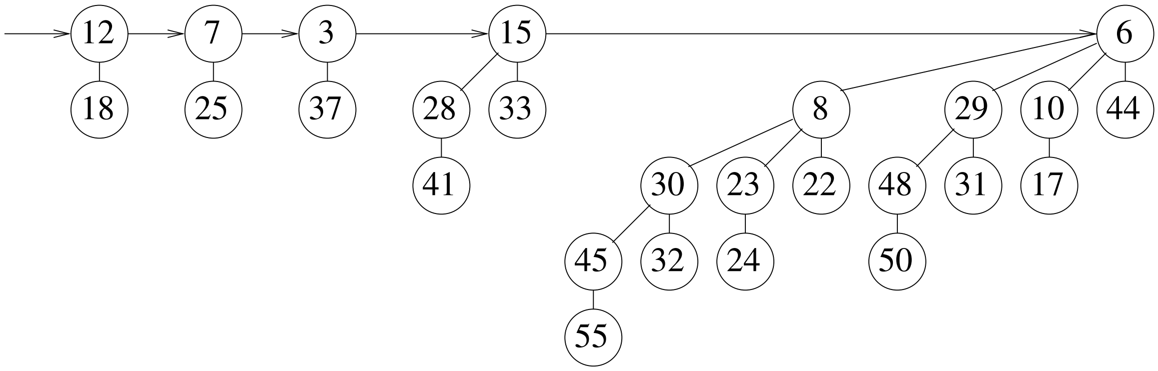
Link the C_i together to form the merged binomial heap.

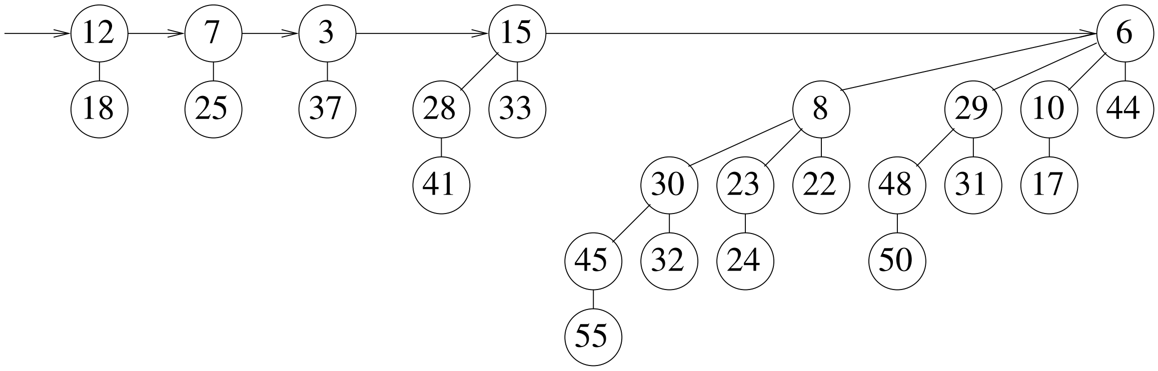
Let $n = |H_1 + |H_2|$.
Every iteration of the for loop runs in $O(1)$ time so the full algorithm obviously runs in $O(k) = O(\lg n)$time.

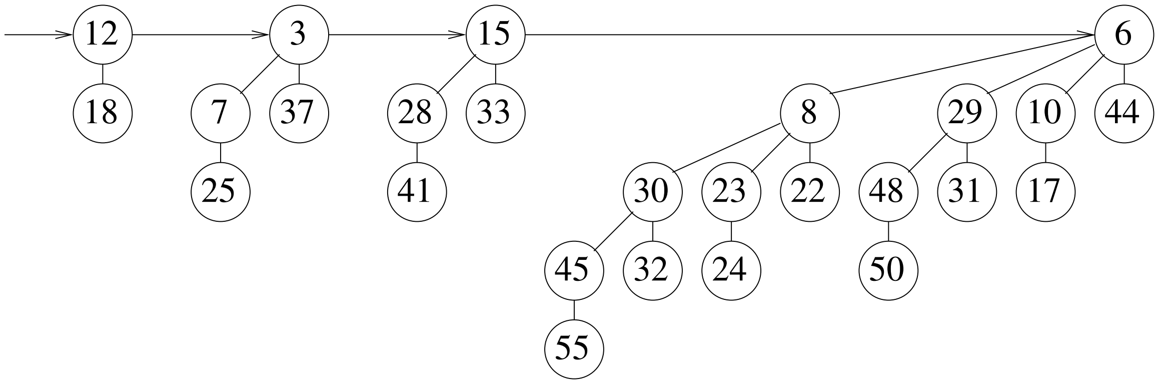
In practice one does not have to record the A_i, B_i, C_i that are $\emptyset$. Instead, the algorithm starts with an $O(\lg n)$ merge of the binomial trees, sorted by degree. This is followed by a linear scan through the merged list, always linking two trees of the same size.

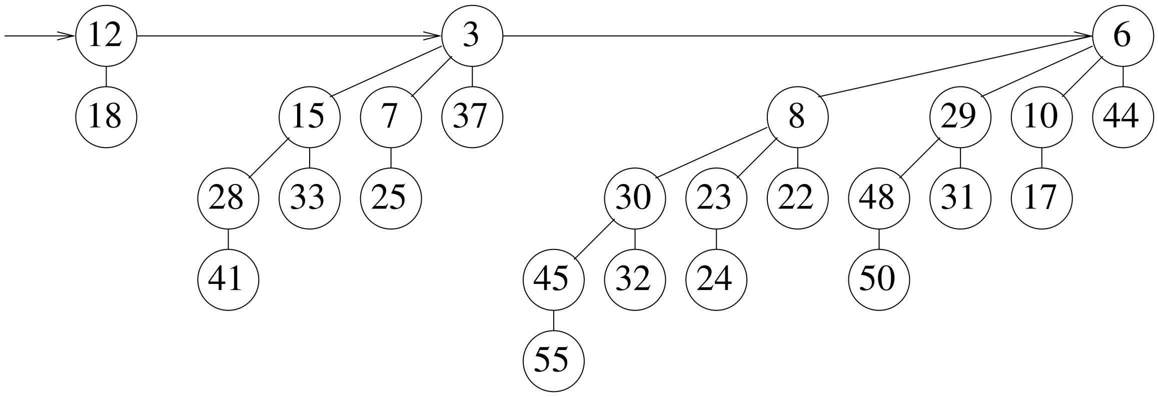

12 7 15 18 3 6
25 28 33 37 8 29 10 44
41 30 23 22 48 31 17
45 32 24 50
55


12 18 7 3 15 6
25 37 28 33 8 29 10 44
41 30 23 22 48 31 17
45 32 24 50
55


12 7 3 15 6
18 25 37 28 33 8 29 10 44
41 30 23 22 48 31 17
45 32 24 50
55


12
7
3
15
6
18
25
37
28
33
8
29
10
44
41
30
23
22
48
31
17
45
32
24
50
55


12
3
15
6
18
7
37
28
33
8
29
10
44
25
41
30
23
22
48
31
17
45
32
24
50
55


12
3
6
18
15
7
37
8
29
10
44
28
33
25
30
23
22
48
31
17
41
45
32
24
50
55

Insert(H, x):

Can be done by first performing an $O(1)$ Make-Heap() to create H_2 and then inserting x into H_2 followed by a $O(\lg n)$ Union(H, H_2).

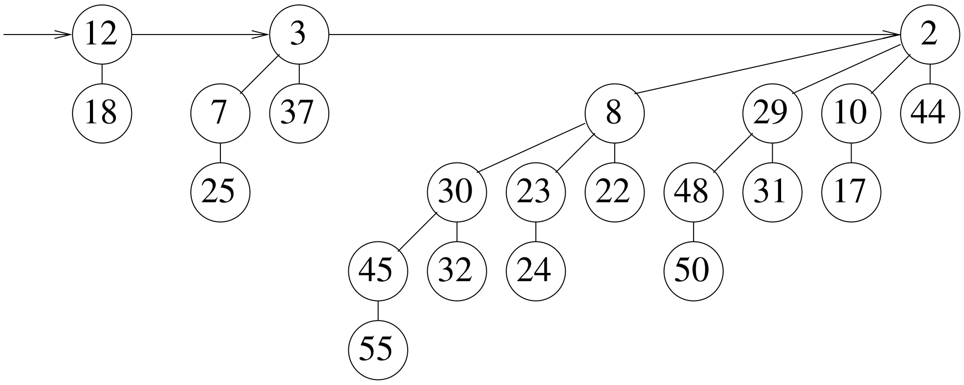
Extract-Min(H):

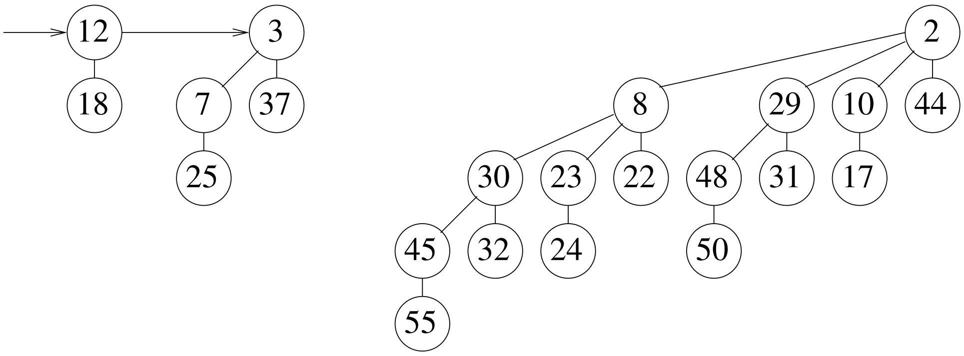
In $O(\lg n)$ time find the root x with the minimum value.

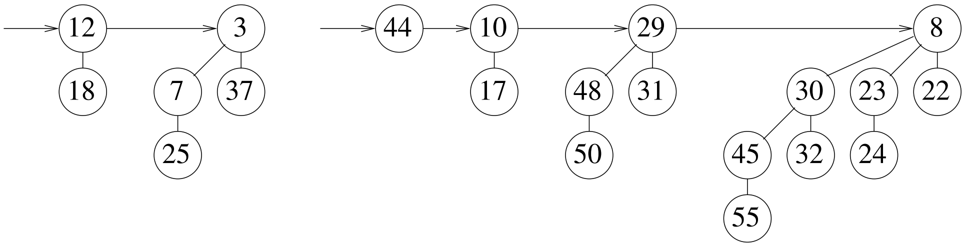
Let A be the tree of which x is the root. Let H_1 be the binomial heap remaining when A is removed from H and H_2 be the binomial heap left over when x is deleted from A.

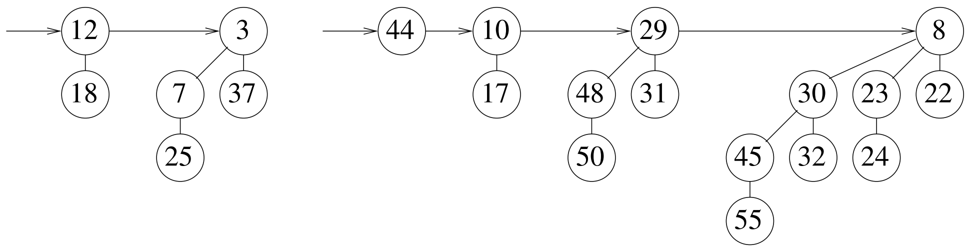
Both H_1 and H_2 can be created in $O(\lg n)$ time. In another $O(\lg n)$ time do Union(H_1, H_2). What results is a binomial heap concatenating all of the items in the original H except for x.

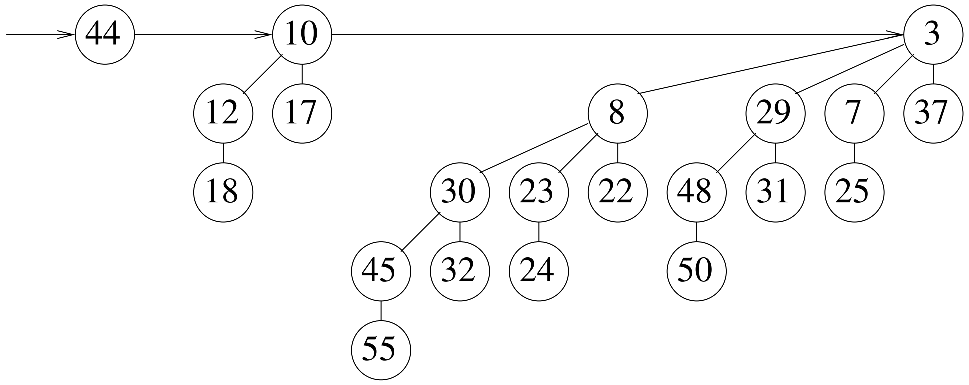
This entire process took only $O(\lg n)$ time.


12
18
3
7
37
25
2
8
29
10
44
30
23
22
48
31
17
45
32
24
50
55


12
18
3
7
37
25
2
8
29
10
44
30
23
22
48
31
17
45
32
24
50
55


12
18
3
7
37
25
44
10
17
29
48
31
50
8
30
23
22
45
32
24
55

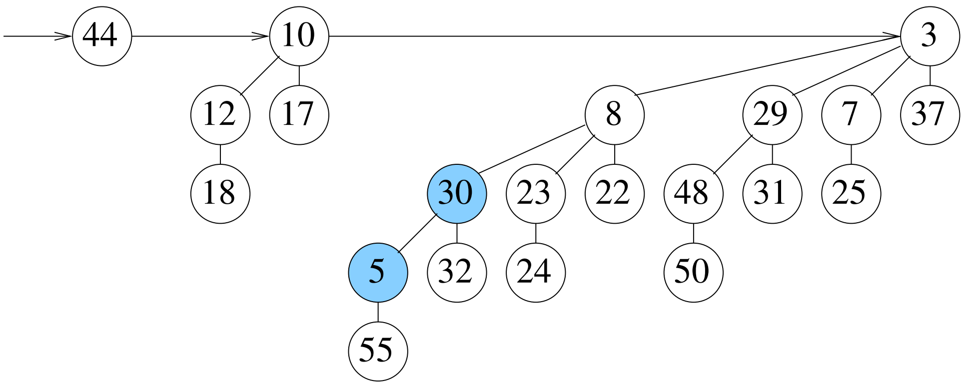

12
3
44
10
29
8
18
7
37
17
48
31
30
23
22
25
50
45
32
24
55

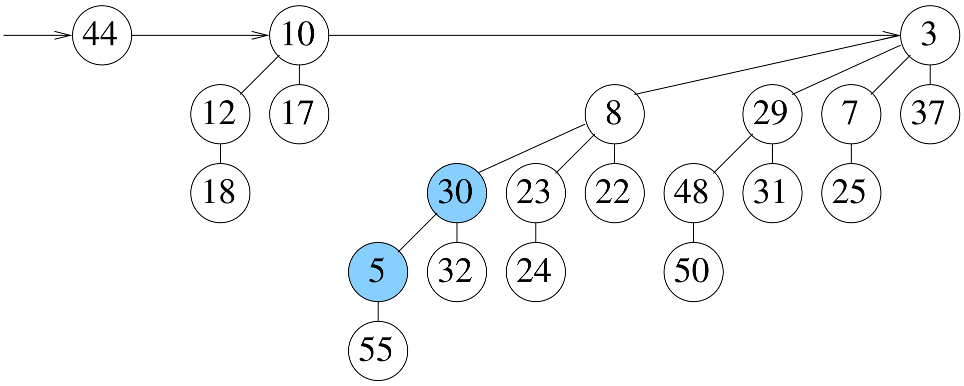

44
10
3
12
17
8
29
7
37
18
30
23
22
48
31
25
45
32
24
50
55

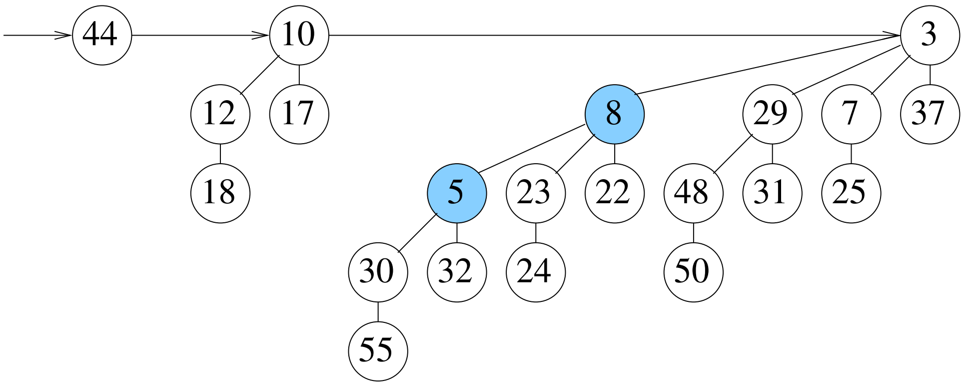
Decrease-Key(H, x, k):
This works exactly the same as in the regular binary heap.
Node x is continuously compared with its parent and, if it's smaller, it is swapped upwards.

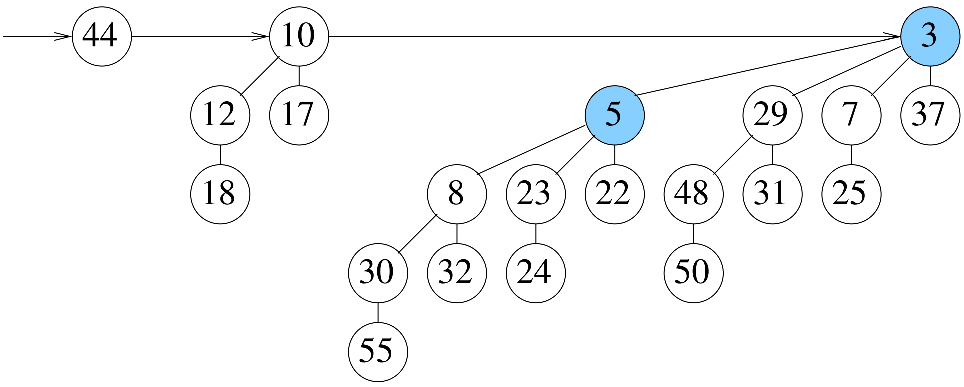
Since the height of any tree is $O(\lg n)$, this takes at most $O(\lg n)$.

In what follows, the key containing a 5 previously contained a 25.




44 10 3
12 17 8 29 7 37
18 30 23 22 48 31 25
5 32 24 50
55


44 10 3
12 17 8 29 7 37
18 5 23 22 48 31 25
30 32 24 50
55


44 10 3
12 17 5 29 7 37
18 8 23 22 48 31 25
30 32 24 50
55

Delete(H, x):
This can be done by performing the two following operations:

Decrease-Key($H, x, -\infty$) $O(\lg n)$
Extract-Min(H) $O(\lg n)$

leading to a $O(\lg n)$ algorithm.

Procedure	Binary heap (worst-case)	Binomial heap (worst-case)	Fibonacci heap (amortized)
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
Extract-Min	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
Union	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
Decrease-Key	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
Delete	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

We have just seen how to build Binomial heaps with the claimed time bounds. Essentially, we managed to substantially decrease the time for Union by slightly increasing the time for Minimum.