Binomial heaps, Fibonacci heaps, & Applications

Fibonacci Heaps

- The Fibonacci heap data structure serves a dual purpose. First, it supports a set of operations that constitutes what is known as a "**mergeable heap.**" Second, several Fibonacci-heap operations **run in constant amortized time**, which makes this data structure well suited for applications that invoke these operations frequently.

Mergeable Heap

A. A mergeable heap is any data structure that supports the following five operations, in which each element has a key:
   1. MAKE-HEAP() creates and returns a new heap containing no elements.
   2. INSERT(H, x) inserts element x, whose key has already been filled in, into heap H.
   3. MINIMUM(H) returns a pointer to the element in heap H whose key is minimum.
   4. EXTRACT (DELETE) -MIN (H) **deletes** the element from heap H whose key is minimum, returning a pointer to the element.
   5. UNION(H_1, H_2) creates and returns a new heap that contains all the elements of heaps H_1 and H_2. Heaps H_1 and H_2 are "destroyed" by this operation.

B. In addition to the mergeable-heap operations above, Fibonacci heaps also support the following two operations:
   1) DECREASE-KEY(H, x, k) assigns to element x within heap H the new key value k, which we assume to be no greater than its current key value.
   2) DELETE.H (H, x) deletes element x from heap H.

Note: We assume our default mergeable heaps are mergeable min-heaps.

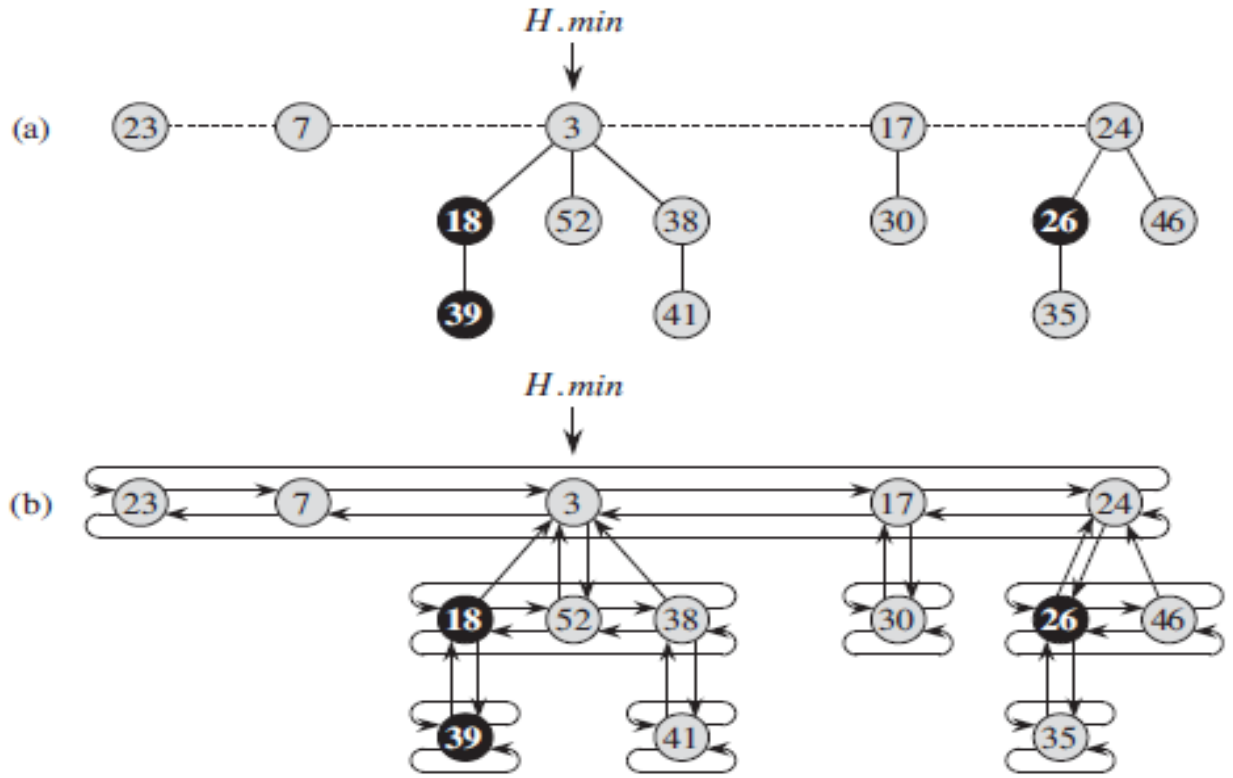
Running times for operations on Mergeable heaps

Procedure	Binary heap (worst-case)	Fibonacci heap (amortized)
MAKE-HEAP	$\Theta(1)$	$\Theta(1)$
INSERT	$\Theta(\lg n)$	$\Theta(1)$
MINIMUM	$\Theta(1)$	$\Theta(1)$
EXTRACT-MIN	$\Theta(\lg n)$	$O(\lg n)$
UNION	$\Theta(n)$	$\Theta(1)$
DECREASE-KEY	$\Theta(\lg n)$	$\Theta(1)$
DELETE	$\Theta(\lg n)$	$O(\lg n)$

Running times for operations on two implementations of mergeable heaps. The number of items in the heap(s) at the time of an operation is denoted by n.

Structure of Fibonacci Heaps

- A Fibonacci heap is a collection of rooted trees that are min-heap ordered. That is, each tree obeys the min-heap property: the key of a node is greater than or equal to the key of its parent. Figure (a) shows an example of a Fibonacci heap.
- As Figure (b) shows, each node x contains a pointer x.p [up arrows] to its parent and a pointer x.child to any one of its children. The children of x are linked together in a circular, doubly linked list, which we call the child list of x. Each child y in a child list has pointers y.left and y.right that point to y's left and right siblings, respectively. If node y is an only child, then y.left = y.right = y. Siblings may appear in a child list in any order.

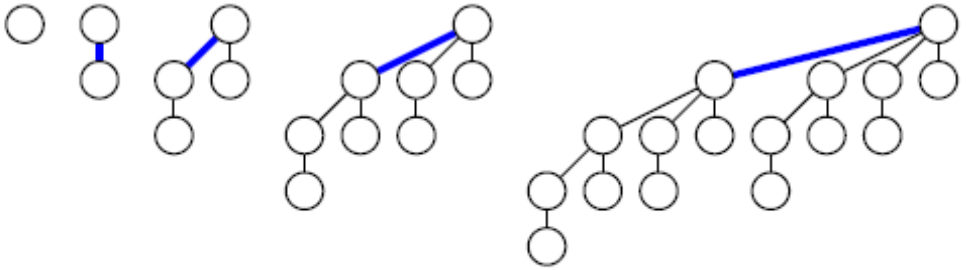


Structure of Fibonacci Heaps (contd.)

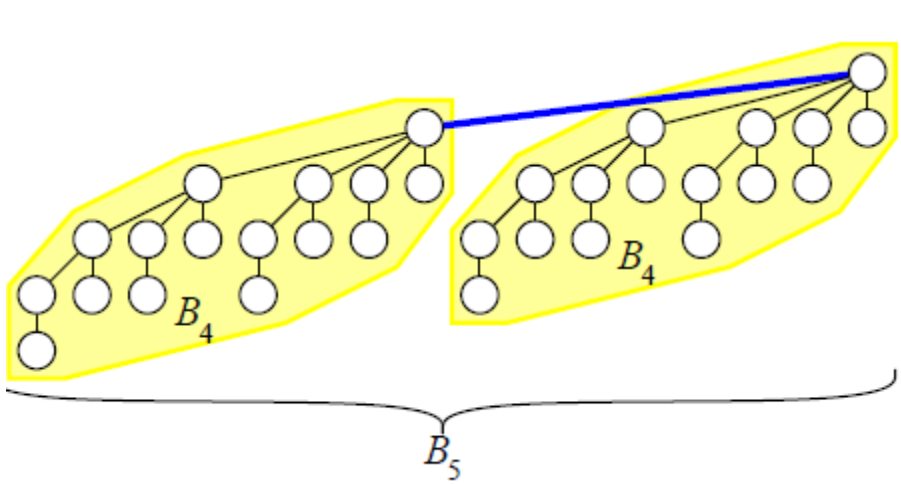
- Circular, doubly linked lists have two advantages for use in Fibonacci heaps.
  - First, we can insert a node into any location or remove a node from anywhere in a circular, doubly linked list in O(1) time.
  - Second, given two such lists, we can concatenate them (or "splice" them together) into one circular, doubly linked list in O(1) time. In the descriptions of Fibonacci heap operations, we shall refer to these operations informally.
- Each node has two other attributes: x.degree [# of children of x], The boolean-valued attribute x.mark [to indicates whether node x has lost a child since the last time x was made the child of another node].
- Newly created nodes are unmarked, and a node x becomes unmarked whenever it is made the child of another node. Initially, set all mark attributes to FALSE.
- We access a given Fibonacci heap H by a pointer H.min to the root of a tree containing the minimum key; we call this node the minimum node of the Fibonacci heap. If more than one root has a key with the minimum value, any such root may serve as the minimum node. When a Fibonacci heap H is empty, H.min is NIL.
- The roots of all the trees in a Fibonacci heap are linked together using their left and right pointers into a circular, doubly linked list called the root list of the Fibonacci heap. The pointer H.min thus points to the node in the root list whose key is minimum. [Trees may appear in any order within a root list].
- We rely on another attribute for a Fibonacci heap H, H.n, the number of nodes currently in H.

Binomial Trees and Fibonacci Heaps

- Fibonacci heap is a circular doubly linked list, with a pointer to the minimum key, but the elements of the list are not single keys. Instead, we collect keys together into structures called binomial heaps. Binomial heaps are trees that satisfy the heap property – every node has a smaller key than its children and have the following special structure.



Binomial trees of order 0 through 5.

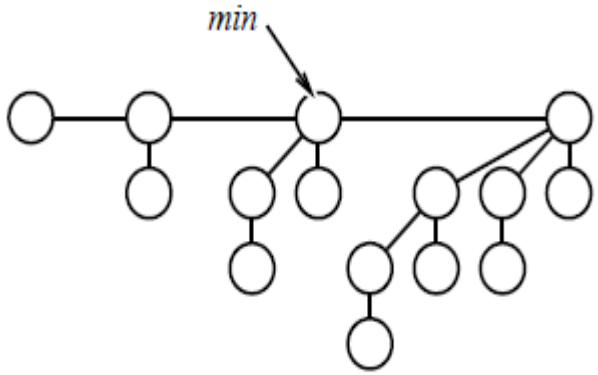


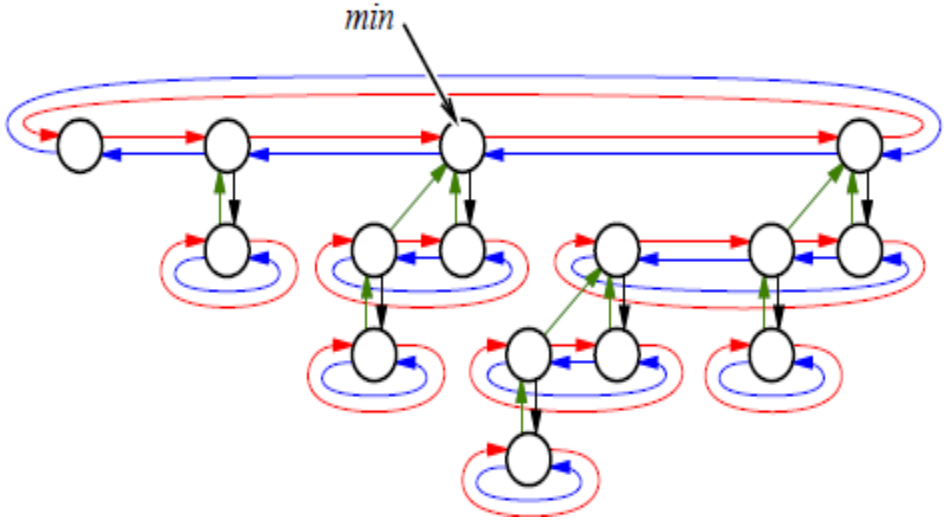
Properties of Binomial Trees

- The root of B_k has degree k.
- The children of the root of B_k are the roots of $B_0, B_1, \ldots, B_{k-1}$.
- B_k has height k.
- B_k has 2^k nodes.
- B_k can be obtained from B_{k-1} by adding a new child to every node.
- B_k has $\binom{k}{d}$ nodes at depth d, for all $0 \leq d \leq k$.
- B_k has 2^{k-h-1} nodes with height h, for all $0 \leq h < k$, and one node (the root) with height k.

Note: Every node in a Fibonacci heap points to four other nodes: its parent, its `next' sibling, its "previous" sibling, and one of its children. The sibling pointers are used to join the roots together into a circular doubly-linked root list. In each binomial tree, the children of each node are also joined into a circular doubly-linked list using the sibling pointers.

Example





- With this representation, we can add or remove nodes from the root list, merge two root lists together, link one two binomial tree to another, or merge a node's list of children with the root list, in constant time, and we can visit every node in the root list in constant time per node.
- Having established that these primitive operations can be performed quickly, we never again need to think about the low-level representation details.
- Every node in a Fibonacci heap points to four other nodes: its parent, its `next' sibling, its `previous' sibling, and one of its children. The sibling pointers are used to join the roots together into a circular doubly-linked root list. In each binomial tree, the children of each node are also joined into a circular doubly-linked list using the sibling pointers.

Operations on Fibonacci Heaps

- The **Insert, Merge, and FindMin** algorithms for Fibonacci heaps are exactly like the corresponding algorithms for linked lists.
- Since we maintain a pointer to the minimum key, **FindMin** is trivial.
- To insert a new key, we add a single node (which we should think of as a B_0) to the root list and (if necessary) update the pointer to the minimum key.
- To **merge** two Fibonacci heaps, we just merge the two root lists and keep the pointer to the smaller of the two minimum keys. Clearly, all three operations take O(1) time.
- DeleteMin is a bit more complicated.
  - First, we remove the minimum key from the root list and splice its children into the root list. Except for updating the parent pointers, this takes O(1) time.
  - Then we scan through the root list to find the new smallest key and update the parent pointers of the new roots. This scan could take $\Theta(n)$ time in the worst case.
  - To bring down the amortized deletion time, we apply a **Cleanup algorithm**, which links pairs of equal-size binomial heaps until there is only one binomial heap of any size.

Cleanup Algorithm

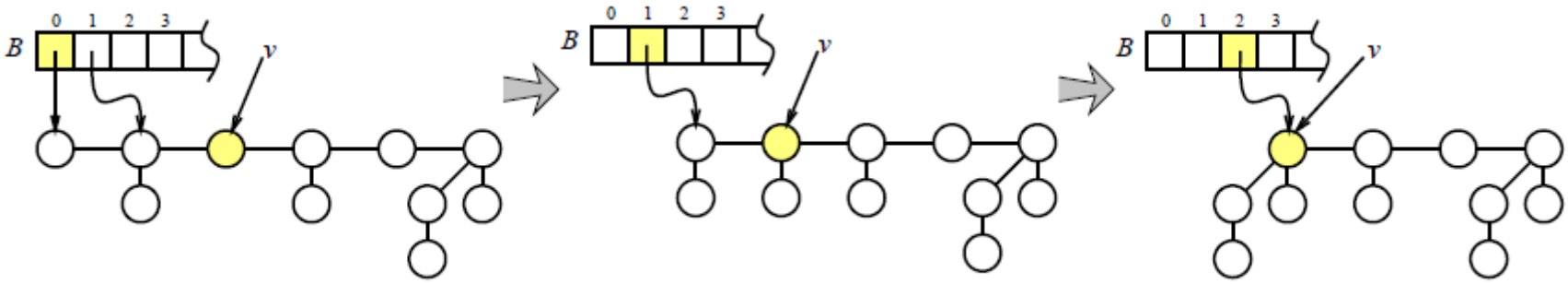
- We will see the Cleanup algorithm in more detail, so we can analyze its running time.
- The following algorithm maintains a global array B[1…⌊lg n⌋], where B[i] is a pointer to some previously-visited binomial heap of order i, or Null if there is no such binomial heap. Notice that Cleanup simultaneously resets the parent pointers of all the new roots and updates the pointer to the minimum key. We split off the part of the algorithm that merges binomial heaps of the same order into a separate subroutine MergeDupes.

```
CLEANUP:
  newmin ← some node in the root list
  for i ← 0 to ⌊lg n⌋
      B[i] ← NULL

  for all nodes v in the root list
      parent(v) ← NULL    (⋆)
      if key(newmin) > key(v)
          newmin ← v
      MERGEDUPES(v)
```

```
MERGEDUPES(v):
  w ← B[deg(v)]
  while w ≠ NULL
      B[deg(v)] ← NULL
      if key(v) ≤ key(w)
          swap v ⇆ w
      remove w from the root list    (⋆⋆)
      link w to v
      w ← B[deg(v)]
  B[deg(v)] ← v
```

Cleanup Algorithm



MERGEDUPES(v), ensuring that no earlier root has the same degree as v.

- Notice that MergeDupes is careful to merge heaps so that the heap property is maintained – the heap whose root has the larger key becomes a new child of the heap whose root has the smaller key. This is handled by swapping v and w if their keys are in the wrong order.
- The running time of Cleanup is O(r'), where r' is the length of the root list just before Cleanup is called. The easiest way to see this is to count the number of times the two starred lines can be executed: line (*) is executed once for every node v on the root list, and line (**) is executed at most once for every node w on the root list. Since DeleteMin does only a constant amount of work before calling Cleanup, the running time of DeleteMin is O(r') = O(r + deg(min)) – NOT GOOD. where **r is the number of roots before DeleteMin begins, and min is the node deleted.**
- **Note:** Although deg(min) is at most lg n, we can still have r = Θ(n) (for example, if nothing has been deleted yet), so the worst-case time for a DeleteMin is Θ(n). After a DeleteMin, the root list has length O(log n)

Amortized Analysis of DeleteMin

- To bound the amortized cost, observe that each insertion increments r (number of roots before DeleteMin begins). If we charge a constant `cleanup tax' for each insertion and use the collected tax to pay for the Cleanup algorithm, the unpaid cost of a DeleteMin is only O(deg(min)) = O(log n).
- More formally, **define the potential of the Fibonacci heap to be the number of roots**. Recall that the amortized time of an operation can be defined as its actual running time plus the increase in potential, provided the potential is initially zero (it is) and we never have negative potential (we never do). Let r be the number of roots before a DeleteMin and let r'' denote the number of roots afterwards. The actual cost of DeleteMin is r + deg(min), and the number of roots increases by r'' – r, so the amortized cost is r'' + deg(min). Since r'' = O(log n) and the degree of any node is O(log n), **the amortized cost of DeleteMin is O(log n).**
- Each **Insert** adds only one root, so its amortized cost is still constant. A Merge doesn't change the number of roots, since the new Fibonacci heap has all the roots from its constituents and no others, so its **amortized cost is also constant**.

Decreasing Keys

- In some applications of heaps, we also need the ability to delete an arbitrary node. The usual way to do this is to decrease the node's key to $-\infty$, and then use DeleteMin. Here is an algorithm to decrease the key of a node in a Fibonacci heap; the algorithm will take O(log n) time in the worst case, but the amortized time will be only O(1). Our algorithm for decreasing the key at a node v follows two simple rules.
  - 1. Promote v up to the root list. (This moves the whole subtree rooted at v.)
  - 2. As soon as two children of any node w have been promoted, immediately promote w.
- In order to enforce the second rule, we now mark certain nodes in the Fibonacci heap. Specifically, a node is marked if exactly one of its children has been promoted. If some child of a marked node is promoted, we promote (and unmark) that node as well. Whenever we promote a marked node, we unmark it; this is the only way to unmark a node. (Specifically, splicing nodes into the root list during a DeleteMin is not considered a promotion.)
- We provide a more formal description of the algorithm next. The input is a pointer to a node v and the new value k for its key.

Decreasing Keys

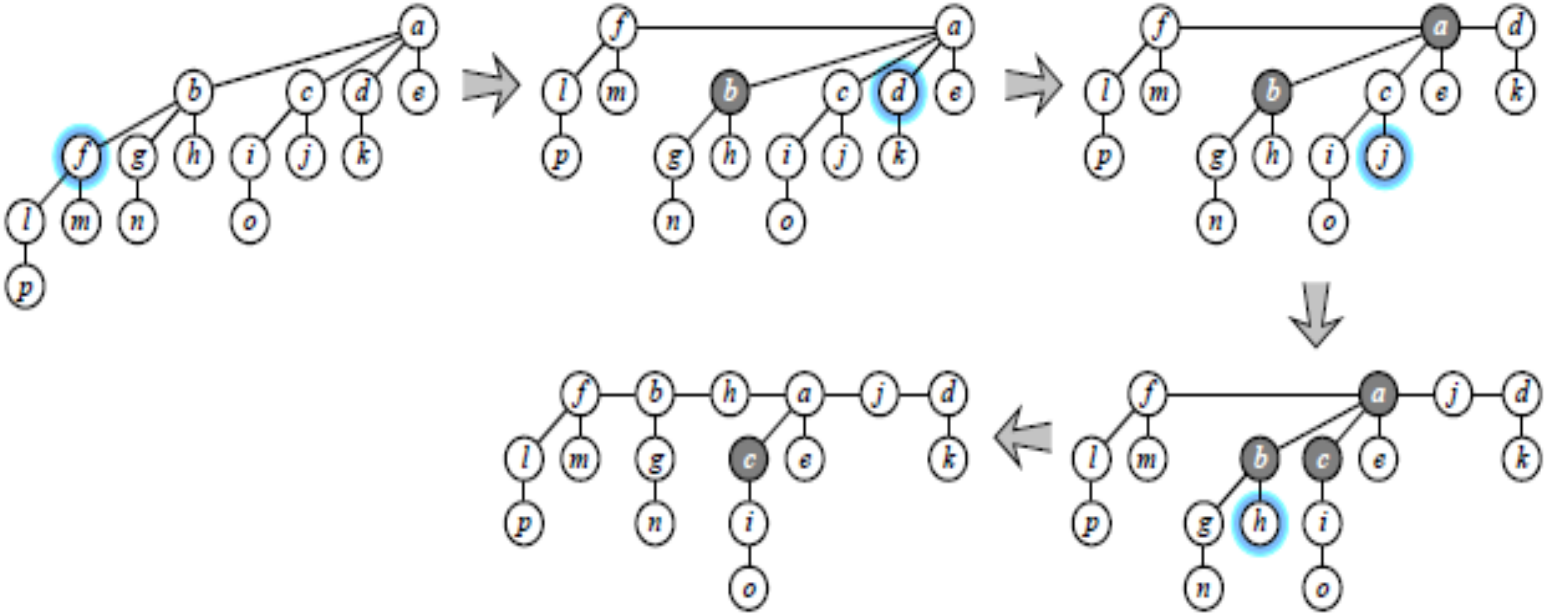
- The input is a pointer to a node v and the new value k for its key.

```
DecreaseKey(v, k):
    key(v) ← k
    update the pointer to the smallest key
    Promote(v)
```

```
Promote(v):
    unmark v
    if parent(v) ≠ Null
        remove v from parent(v)'s list of children
        insert v into the root list
        if parent(v) is marked
            Promote(parent(v))
        else
            mark parent(v)
```

- The Promote algorithm calls itself recursively, resulting in a `cascading promotion'. Each consecutive marked ancestor of v is promoted to the root list and unmarked, otherwise unchanged.
- The lowest unmarked ancestor is then marked, since one of its children has been promoted.
- The time to decrease the key of a node v is O(1+#consecutive marked ancestors of v). Binomial heaps have logarithmic depth, so if we still had only full binomial heaps, the running time would be O(log n). Unfortunately, promoting nodes destroys the nice binomial tree structure; our trees no longer have logarithmic depth! In fact, DecreaseKey runs in Θ(n) time in the **worst** case.

Example



Decreasing the keys of four nodes: first f, then d, then j, and finally h. Dark nodes are marked. DECREASEKEY(h) causes nodes b and a to be recursively promoted.

Amortized cost of Decreasing Keys

- To compute the amortized cost of DecreaseKey, we'll use the potential method, just as we did for DeleteMin.
- We need to find a potential function Φ that goes up a little whenever we do a little work and goes down a lot whenever we do a lot of work.
- DecreaseKey unmarks several marked ancestors and possibly also marks one node. So, the number of marked nodes might be an appropriate potential function here.
  - Whenever we do a little bit of work, the number of marks goes up by at most one; whenever we do a lot of work, the number of marks goes down a lot.
  - More precisely, let m and m′ be the number of marked nodes before and after a DecreaseKey operation. The actual time (ignoring constant factors) is t = 1 + #consecutive marked ancestors of v and if we set Φ = m, the increase in potential is $m' - m \leq 1 -$ #consecutive marked ancestors of v.
  - Since $t + \Delta\Phi \leq 2$, **the amortized cost of DecreaseKey is O(1)**.

Bounding the Degree

- Now we have a problem with our earlier analysis of DeleteMin. The amortized time for a DeleteMin is still O(r+deg(min)). To show that this equaled O(log n), we used the fact that the maximum degree of any node is O(log n), which implies that after a Cleanup the number of roots is O(log n). But now that we don't have complete binomial heaps, this `fact' is no longer obvious!
- So, let's prove it. For any node v, let |v| denote the number of nodes in the subtree of v, including v itself. Our proof uses the following lemma, which finally tells us why these things are called Fibonacci heaps.
- **Lemma:** For any node v in a Fibonacci heap, $|v| \geq F_{deg(v)+2}$.
- **Proof:** [Skip the proof if you want to] Label the children of v in the chronological order in which they were linked to v. Consider the situation just before the i^{th} oldest child w_i was linked to v. At that time, v had at least i – 1 children (possibly more). Since Cleanup only links trees with the same degree, we had $deg(w_i) = deg(v) \geq i - 1$. Since that time, at most one child of w_i has been promoted away; otherwise, w_i would have been promoted to the root list by now. So currently we have $deg(w_i) \geq i - 2$.
- We also quickly observe that $deg(w_i) \geq 0$.

Bounding the Degree (contd.)

- Let s_d be the minimum possible size of a tree with degree d in any Fibonacci heap. Clearly
- $s_0 = 1$; for notational convenience, let $s_{-1} = 1$ also. By our earlier argument, the i^{th} oldest child of the root has degree at least $\max\{f_0, i-2\}$, and thus has size at least $\max(1, s_{i-2}) = s_{i-2}$. Thus, we have the following recurrence:

$$s_d \geq 1 + \sum_{i=1}^{d} s_i - 2$$

- If we assume inductively that $s_i \geq F_{i+2}$ for all $-1 \leq i < d$ (with the easy base cases $s_{-1} = F_1$ and $s_0 = F_2$), we have [Remember, by definition $|v| \geq s_{deg(v)}$]

$$s_d \geq 1 + \sum_{i=1}^{d} F_i = F_{d+2}$$

- We can easily show, by using induction, that $F_{k+2} > \phi^k$ where $\phi = (1+\sqrt{5})/2 \approx 1.618$ is the **golden ratio**. Thus, by our previous lemma, we get $deg(v) \leq \log_\phi |v| = O(\log |v|)$.
- Thus, since the size of any subtree in an n-node Fibonacci heap is obviously at most n, the degree of any node is O(log n), which is exactly what we wanted. Our earlier analysis is still good.

Analyzing Everything Together

- Unfortunately, our analyses of DeleteMin and DecreaseKey used two different potential functions. Unless we can find a single potential function that works for both operations, we can't claim both amortized time bounds simultaneously. So, we need to find a potential function Φ that goes up a little during a cheap DeleteMin or a cheap DecreaseKey, and goes down a lot during an expensive DeleteMin or an expensive DecreaseKey.
- Let's look a little more carefully at the cost of each Fibonacci heap operation, and its effect on both the number of roots and the number of marked nodes, the things we used as out earlier potential functions. Let r and m be the numbers of roots and marks before each operation and let r' and m' be the numbers of roots and marks after the operation.

operation	actual cost	$r' - r$	$m' - m$
INSERT	1	1	0
MERGE	1	0	0
DELETEMIN	$r + \deg(min)$	$r' - r$	0
DECREASEKEY	$1 + m - m'$	$1 + m - m'$	$m' - m$

- If we guess that the correct potential function is a linear combination of our old potential functions r and m and play around with various possibilities for the coefficients, we will eventually stumble across the correct answer: **$\phi = r + 2m$**

Analyzing Everything Together

- To see that this potential function gives us good amortized bounds for every Fibonacci heap operation, let's add two more columns to our table.

operation	actual cost	$r' - r$	$m' - m$	$\Phi' - \Phi$	amortized cost
INSERT	1	1	0	1	2
MERGE	1	0	0	0	1
DELETEMIN	$r + \deg(\min)$	$r' - r$	0	$r' - r$	$r' + \deg(\min)$
DECREASEKEY	$1 + m - m'$	$1 + m - m'$	$m' - m$	$1 + m' - m$	2

- Since Lemma 1 implies that r' + deg(min) = O(log n), we're finally done!

Fibonacci Trees

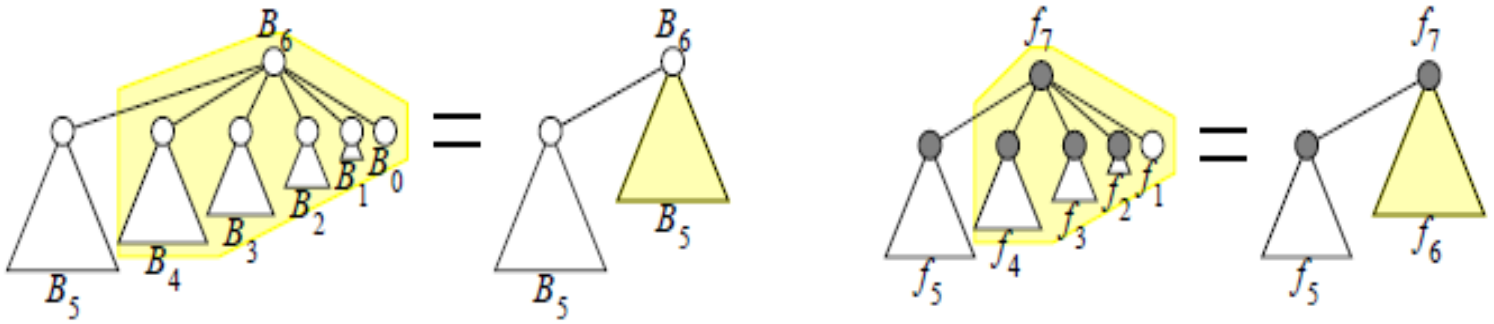
- To get a little more intuition about how Fibonacci heaps behave, let's look at a worst-case construction for our Lemma.
- Suppose we want to remove as many nodes as possible from a binomial heap of order k, by promoting various nodes to the root list, but without causing any cascading promotions. The most damage we can do is to promote the largest subtree of every node. Call the result a Fibonacci tree of order k + 1 and denote it f_{k+1}. As a base case, let f_1 be the tree with one (unmarked) node, that is, $f_1 = B_0$. The reason for shifting the index should be obvious after a few seconds.



Fibonacci trees of order 1 through 6. Light nodes have been promoted away; dark nodes are marked.

Fibonacci Trees

- Recall that the root of a binomial tree B_k has k children, which are roots of B_0, B_1 …, B_{k-1}. To convert B_k to f_{k+1}, we promote the root of B_{k-1}, and recursively convert each of the other subtrees B_i to f_{i+1}. The root of the resulting tree f_{k+1} has degree k .. 1, and the children are the roots of smaller Fibonacci trees f_1, f_2, … , f_{k-1}. We can also consider B_k as two copies of B_{k-1} linked together. It's quite easy to show that an order-k Fibonacci tree consists of an order k – 2 Fibonacci tree linked to an order k – 1 Fibonacci tree.



Comparing the recursive structures of B_6 and f_7.

- Since f_1 and f_2 both have exactly one node, the number of nodes in an order-k Fibonacci tree is exactly the kth Fibonacci number! Since f_1 and f_2 both have exactly one node, the number of nodes in an order-k Fibonacci tree is exactly the k^{th} Fibonacci number!

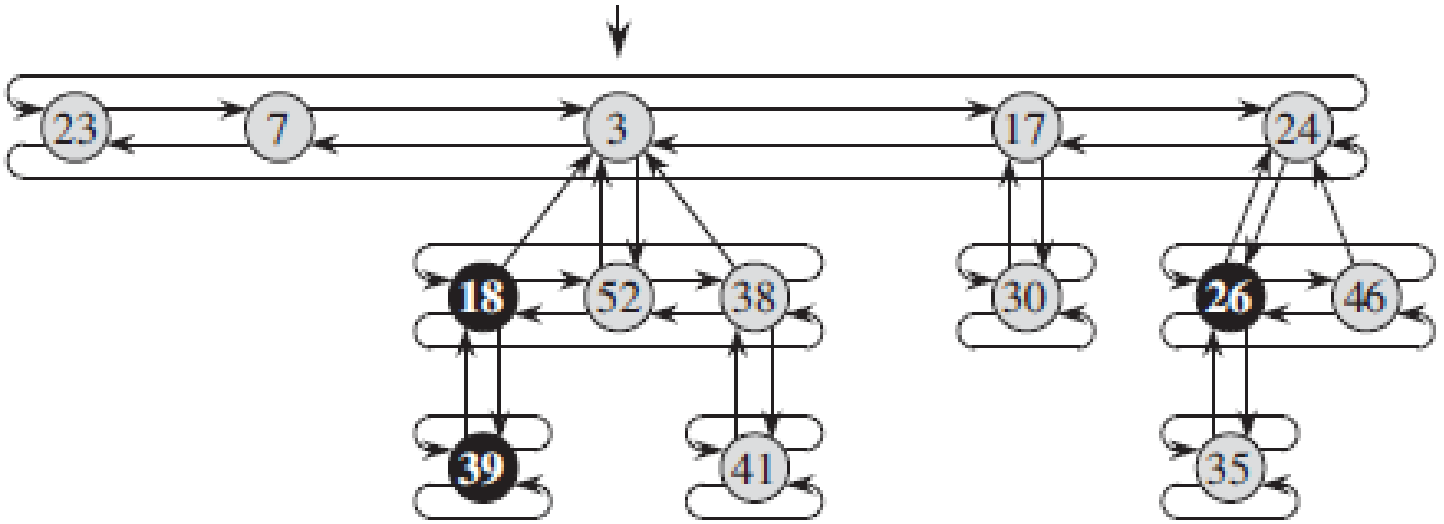
Properties of Fibonacci Trees

- Like binomial trees, Fibonacci trees have lots of other nice properties that easy to prove by induction.

  - The root of f_k has degree $k-2$.
  - f_k can be obtained from f_{k-1} by adding a new unmarked child to every marked node and then marking all the old unmarked nodes.
  - f_k has height $\lceil k/2 \rceil - 1$.
  - f_k has F_{k-2} unmarked nodes, F_{k-1} marked nodes, and thus F_k nodes altogether.
  - f_k has $\binom{k-d-2}{d-1}$ unmarked nodes, $\binom{k-d-2}{d}$ marked nodes, and $\binom{k-d-1}{d}$ total nodes at depth d, for all $0 \le d \le \lfloor k/2 \rfloor - 1$.
  - f_k has F_{k-2h-1} nodes with height h, for all $0 \le h \le \lfloor k/2 \rfloor - 1$, and one node (the root) with height $\lceil k/2 \rceil - 1$.

Potential function

- We use the potential method to analyze the amortized performance of Fibonacci heap operations. For a given Fibonacci heap H, we indicate by t(H) the **number of trees in the root list** of H and by m(H) the **number of marked nodes** in H. We then define the potential Φ of Fibonacci heap H by $\Phi(H) = t(H) + 2m(H)$.
- The potential of the Fibonacci heap shown is 5 + 2.3 =11. The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent Fibonacci heaps. We assume that a unit of potential can pay for a constant amount of work.
- We assume that a Fibonacci heap application begins with no heaps. The initial potential, therefore, is 0, and so, the potential is nonnegative at all subsequent times.
- An upper bound on the total amortized cost provides an upper bound on the total actual cost for the sequence of operations.



References

- https://www.geeksforgeeks.org/insert-operation-in-b-tree/ Insert
- https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1126/lectures/05/Slides05.pdf
- https://www.cs.princeton.edu/~wayne/teaching/fibonacci-heap.pdf
- https://www.programiz.com/dsa/decrease-key-and-delete-node-from-a-fibonacci-heap