Amortized analysis, Fibonacci heaps

CS240 Spring 2020

Rui Fan

Amortized analysis

- Suppose we want to bound the amount of time to perform n (possibly different) operations on a data structure.
- If max amount of time for each operation is f(n), $n \cdot f(n)$ is upper bound on the time for all the operations.
- But some operations might take more time than others.
  - Even the same operation can take different amounts of time each time it's executed.
  - So $n \cdot f(n)$ may overestimate actual amount of time taken.
  - The bound isn't tight.
- Amortized analysis looks at the average amount of time for each operation over all the operations.
  - The average is taken over the worst case execution, i.e. a sequence of operations with the highest average cost for the data structure.

Potential method

- To keep track of the true total cost of a sequence of operations, we use a potential function $\Phi: D \to \mathbb{R}$, where D is the set of states of the data structure.
- Let D_i be the state of the data structure after applying the i'th operation, and c_i be the cost of the i'th operation.
- Def The amortized cost for the i'th operation is $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1})$.
- Using the amortized cost, we sometimes overcharge and sometimes undercharge for operations.
  - I.e. when $\widehat{c_i} > c_i$, we overcharge, and when $\widehat{c_i} < c_i$ we undercharge.
- However, the total amortized cost is at least the total actual cost, i.e. $\sum_i \widehat{c_i} \geq \sum_i c_i$.
  - So total amortized cost is an upper bound on total actual cost.
- If we design the right potential function, we can keep track of the total cost by tracking the amortized costs.
  - The amortized cost is sometimes easier to analyze than directly keeping track of actual costs.
  - This leads to tight bounds for many data structures.

Potential method

- When we overcharge, i.e. $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) > c_i$, Φ increases.
  - We "store" the extra amortized cost $\widehat{c_i} - c_i$ we charged the i'th operation in Φ
  - Φ is also called "credit" or "potential" (energy).
- When we undercharge, i.e. $\widehat{c_i} < c_i$, Φ decreases.
  - We use some of the stored credit to pay for the $c_i - \widehat{c_i}$ amount of actual cost that the amortized cost doesn't account for.
- Lemma Suppose $\Phi(D_n) \geq \Phi(D_0)$. Then $\sum_{i=1}^{n} \widehat{c_i} \geq \sum_{i=1}^{n} c_i$.
- Proof
  $\sum_{i=1}^{n} \widehat{c_i} = \sum_{i=1}^{n}(c_i + \Phi(D_i) - \Phi(D_{i-1})) = (\sum_{i=1}^{n} c_i) + \Phi(D_n) - \Phi(D_0) \geq \sum_{i=1}^{n} c_i$.
  - The second equality follows because all the terms except $\Phi(D_n), \Phi(D_0)$ telescope away.
- A simple way to ensure $\Phi(D_n) \geq \Phi(D_0)$ is to design Φ so that $\Phi(D_0) = 0$, and $\Phi(D_i) \geq 0$ for all i.

Example: Binary counter

- Consider a k-digit binary counter. When we increment the counter, we flip some bits.
  - Suppose each bit flip costs 1 unit.
- What is the total cost for incrementing the counter n times, starting from 0?
- Since there are k digits, a trivial upper bound is O(nk).
- However, the actual number of bit flips is much less, because most increments only flip a few bits.
- We use the potential method to show the total cost is O(n).
  - In fact, it's at most 2n.

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

Example: Binary counter

- Let $\Phi(D_i) = b_i$, where D_i is the state of the counter after i increments, and b_i is the number of 1's in D_i.
- Suppose the i'th operation sets t_i bits from 1 to 0.
  - Then the actual cost is $c_i = t_i + 1$.
  - This sets t_i bits from 1 to 0, and one bit from 0 to 1 for the carry.
- If $b_i = 0$, then the i'th operation reset all the bits.
  - Also, all the bits were set in D_{i-1}.
  - So $t_i = b_{i-1} = k$.
- If $b_i > 0$, then $b_i = b_{i-1} - t_i + 1$.
  - t_i bits went from 1 to 0, and one carry bit went from 0 to 1.
- In both cases, $b_i \leq b_{i-1} - t_i + 1$.

Example: Binary counter

- Since $b_i \le b_{i-1} - t_i + 1$, then $\Phi(D_i) - \Phi(D_{i-1}) = b_i - b_{i-1} \le 1 - t_i$.
- So the amortized cost $\widehat{c_i} = c_i + \Phi(D_i) - \Phi(D_{i-1}) \le (t_i + 1) + (1 - t_i) = 2$.
- Finally, $\Phi(D_0) = 0$, since the counter is initially 0, and $\Phi(D_n) \ge 0$.
- Thus, by the lemma the total cost for all n increments is $\sum_{i=1}^{n} c_i \le \sum_{i=1}^{n} \widehat{c_i} \le 2n$.

Fibonacci heaps

- A Fibonacci heap is a type of heap that implements certain operations faster than a binary heap, in amortized time.
    - The time complexities of the circled operations are amortized. The rest are worst case.
- It can be used to speed up a number of graph algorithms asymptotically.
- Both Dijkstra's and Prim's algorithms take O((V+E) log V) time with a binary heap, and O(E + V log V) time with a Fibonacci heap.
    - Both algorithms decrease the key value heap items O(E) times.
    - This takes O(E log V) time on a binary heap, and O(E) time on a Fibonacci heap.
- Fibonacci heaps are more complicated than binary heaps, and often don't perform better in practice.
- When decreasing or deleting a key, assume we have a pointer to the node with the key.
    - Otherwise finding the node takes O(n) time, where n is the number of items.

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	$\Theta(\lg n)$	$O(\lg n)$

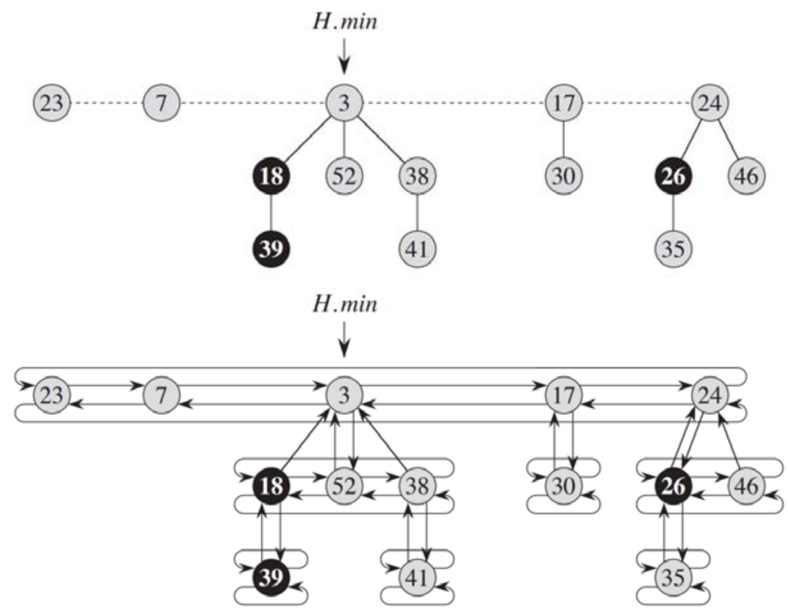
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for each vertex v ∈ G.Adj[u]
        RELAX(u, v, w)
```

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

```
MST-PRIM(G, w, r)
for each u ∈ G.V
    u.key = ∞
    u.π = NIL
r.key = 0
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    for each v ∈ G.Adj[u]
        if v ∈ Q and w(u, v) < v.key
            v.π = u
            v.key = w(u, v)
```

Structure of Fibonacci heap

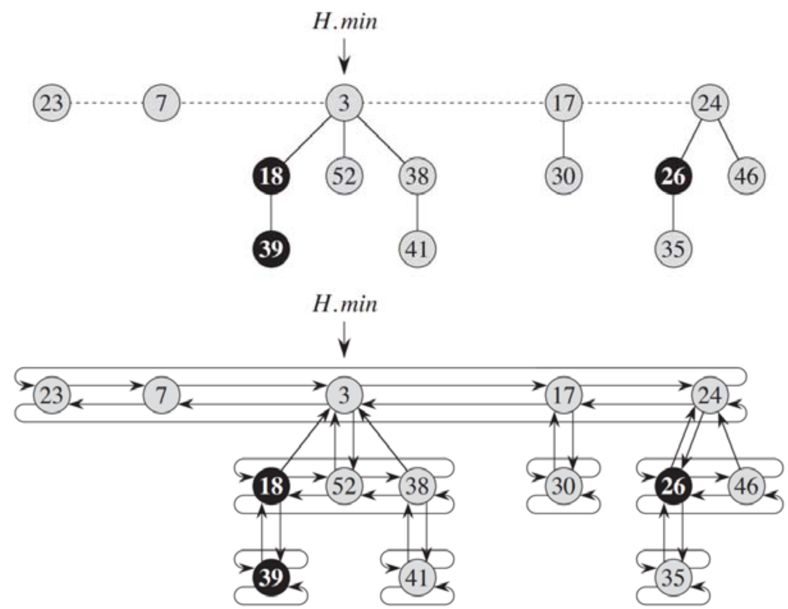
- A Fibonacci heap H consists of a set of rooted trees.
- Each tree satisfies the min heap property, i.e. each node's key is less than those of all its children.
- The trees are linked in a doubly-linked root list.
  - These roots are connected by the dashed line in the top figure.
- The minimum node is a root, and is pointed to by H.min.
- H.n stores the total number of nodes in all trees.
- Within each tree, the nodes at each level are also linked in a doubly-linked list.
- The two figures show the same Fibonacci heap, but the top figure avoids showing the linked list for clarity.



Potential function

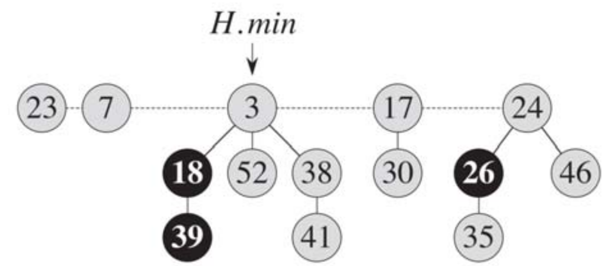
- Some nodes are marked.
  - These are shown in black.
- Marks are only used during decrease key and deletion operations, and are described later.
  - They help ensure each node has a large number of children.
  - This ensures each tree is not too tall, and so each operation is fast.
- Suppose H has t(H) root nodes and m(H) marked nodes.
- The potential of H is

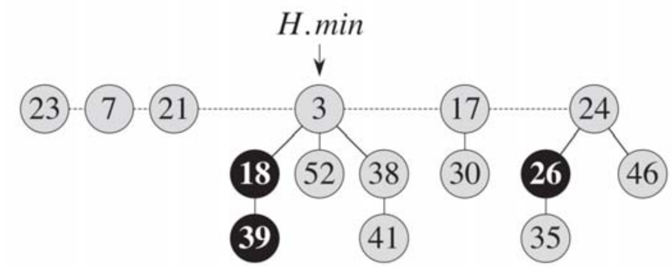
$$\Phi(H) = t(H) + 2m(H)$$



Basic operations

- Let H and H' denote the heap before and after an operation.
- Make-Heap
  - Make an empty heap. Set H'.n=0, H'.min=NIL.
  - Cost = O(1).
  - Amortized cost = O(1), since $\Phi(H) = \Phi(H') = 0$.
- Insert a node
  - Add the new node to the root list, left of the min node.
  - Change H.min if new node's key is smaller.
  - Cost = O(1).
  - Amortized cost = O(1).
    - Number of roots increases by 1.
    - So $\Phi(H') - \Phi(H) = \big(t(H) + 1 + 2m(H)\big) - \big(t(H) + 2m(H)\big) = O(1)$.






```
FIB-HEAP-INSERT(H, x)
x.degree = 0
x.p = NIL
x.child = NIL
x.mark = FALSE
if H.min == NIL
      create a root list for H containing just x
      H.min = x
else insert x into H's root list
      if x.key < H.min.key
            H.min = x
H.n = H.n + 1
```

Basic operations

- Find the minimum
  - Return H.min.
  - Cost = amortized cost = O(1).
- Union of two heaps
  - Concatenate the root lists of the two heaps H_1 and H_2.
  - Set H.min to $\min(H_1.min, H_2.min)$
  - Cost = O(1).
  - Since new root list is the union of the two old root lists, the change in potential is $\Phi(H') - \big(\Phi(H_1) + \Phi(H_2)\big) = t(H') + 2m(H') - \big(t(H_1) + 2m(H_1) + t(H_2) + 2m(H_2)\big) = 0.$
  - Thus the amortized cost = O(1).

```
FIB-HEAP-UNION(H1, H2)
H = MAKE-FIB-HEAP()
H.min = H1.min
concatenate the root list of H2 with the root list of H
if (H1.min == NIL) or (H2.min ≠ NIL and H2.min.key < H1.min.key)
    H.min = H2.min
H.n = H1.n + H2.n
return H
```

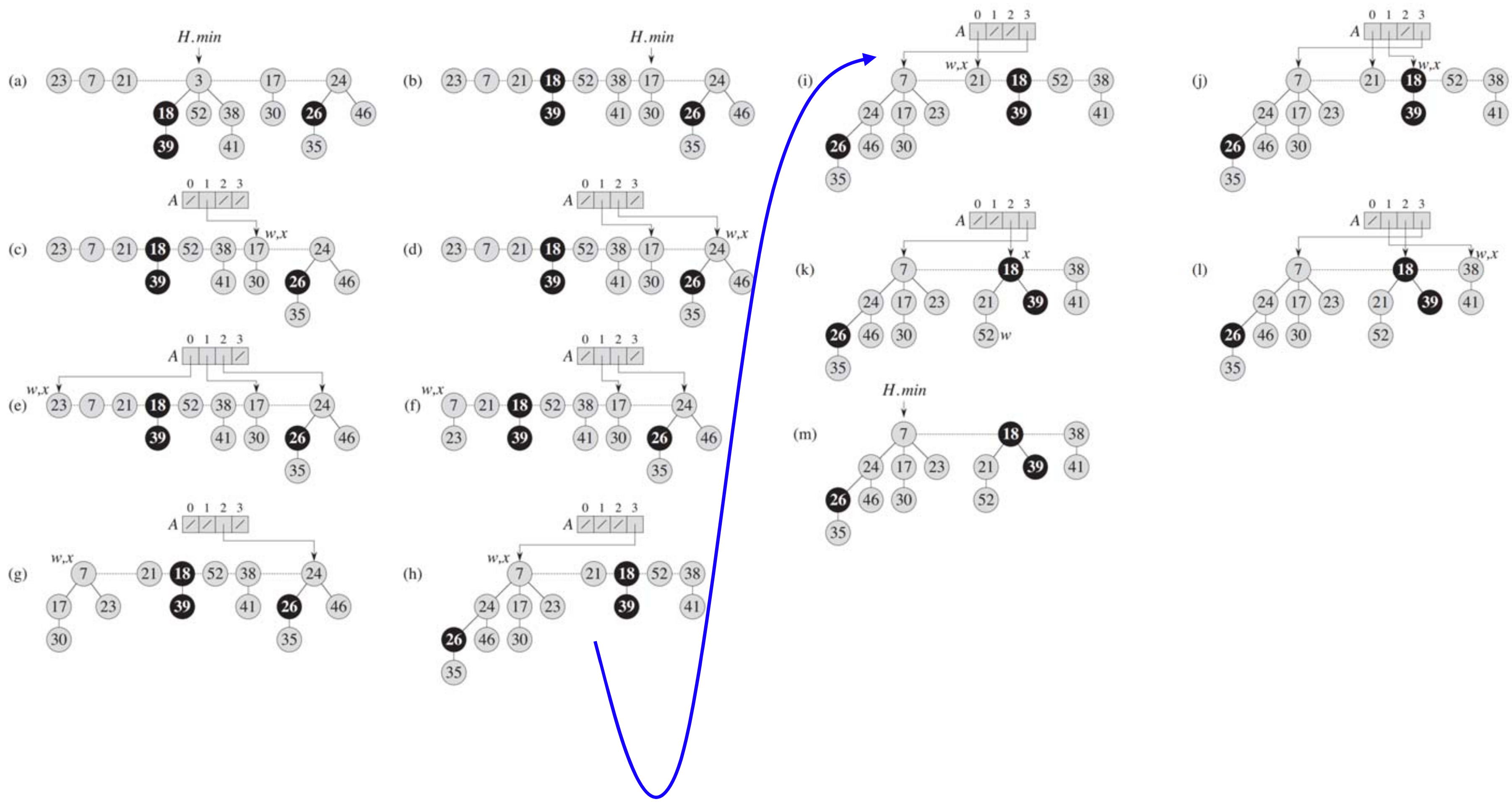
Extract min node

- First remove the H.min node.
- Then add each of its children (along with its subtree) to the root list.
  - There may now be many trees in the root list.
  - To find the new H.min, we need to iterate through the roots of all the trees, which may be slow.
  - So we want to decrease the number of trees in the root list.
- Def The degree of a node is its number of children.
- We merge some of the trees in the root list, so that none of the roots have the same degree.
  - The merging function is called CONSOLIDATE.

CONSOLIDATE

- Let D(H.n) be upper bound on the degree of any node in a Fibonacci heap with n nodes.
  - We show later D(H.n) = O(log n).
- Use an array A of size D(H.n)+1.
  - A[i] points to a tree in the root list with degree i.
- Iterate through all the trees in the root list.
  - If the current tree x we process has degree d, and $A[d] = y \neq NIL$, then there's already a tree y in the root list with degree d.
  - Since we don't want two trees in the root list with the same degree, we link the roots of x and y.
    - This creates a tree in the root list with degree d+1, and removes the tree with degree d.
    - The direction we link depends on which root has the smaller key.
  - Then set A[d]=NIL, and set A[d+1] to point to newly linked tree.
  - If the new root is marked, clear the mark.
- Finally, iterate through A array, and set H.min to the min root value.

Example: Extract min



Pseudocode for extract min

```
FIB-HEAP-EXTRACT-MIN(H)
z = H.min
if z ≠ NIL
    for each child x of z
        add x to the root list of H
        x.p = NIL
    remove z from the root list of H
    if z == z.right
        H.min = NIL
    else H.min = z.right
        CONSOLIDATE(H)
    H.n = H.n − 1
return z
```

```
FIB-HEAP-LINK(H, y, x)
remove y from the root list of H
make y a child of x, incrementing x.degree
y.mark = FALSE
```

```
CONSOLIDATE(H)
let A[0 . . D(H.n)] be a new array
for i = 0 to D(H.n)
    A[i] = NIL
for each node w in the root list of H
    x = w
    d = x.degree
    while A[d] ≠ NIL
        y = A[d]        // another node with the same degree as x
        if x.key > y.key
            exchange x with y
        FIB-HEAP-LINK(H, y, x)
        A[d] = NIL
        d = d + 1
    A[d] = x
H.min = NIL
for i = 0 to D(H.n)
    if A[i] ≠ NIL
        if H.min == NIL
            create a root list for H containing just A[i]
            H.min = A[i]
        else insert A[i] into H's root list
            if A[i].key < H.min.key
                H.min = A[i]
```

Complexity for extract min

- Let H denote the heap before the extract min.
- The real cost includes
  - O(D(n)) for moving children of H.min to root list.
  - The for loop in lines 4-14 of CONSOLIDATE operate on a list of size at most D(n)+t(H)-1.
  - Every time through the while loop in lines 7-13, we link two of the trees in the root list.
    - Each tree can be linked (to a tree whose root has a smaller key) at most once.
    - So the total number of iterations of the while loop is at most the root list size, i.e. O(D(n)+t(H)).
  - So the real cost is O(D(n)+t(H)).
- For the amortized cost, the potential before the extract min is at most t(H)+2m(H).
- The potential after extract is ≤ (D(n)+1)+2m(H).
  - All trees in root list of H' have different degrees, and max degree is D(n).
  - No new nodes get marked during extract.
- So amortized cost is

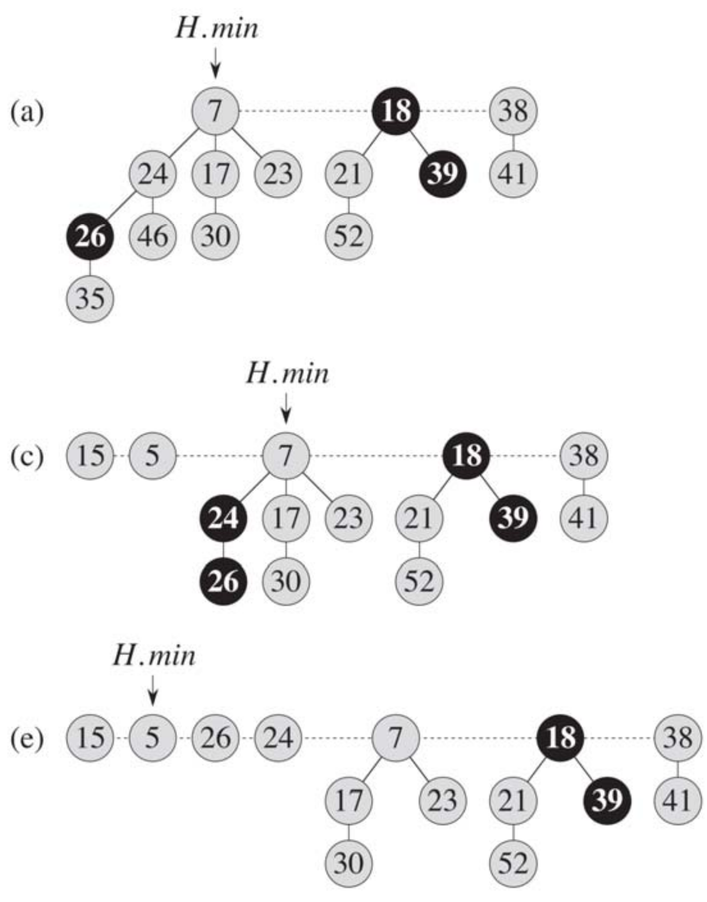
$$O\big(D(n)+t(H)\big)+\big((D(n)+1)+2m(H)\big)-\big(t(H)+2m(H)\big)$$
$$=O(D(n))+O(t(H))-t(H)=O(D(n)).$$

  - The last equality follows because we can scale up the units of the potential to cancel out the hidden constant in O(t(H)).

Decreasing key and marking

- To decrease a node x's key, check if the new key violates the heap property.
  - If not, we're done.
  - Otherwise, move x and its subtree to the root list.
    - We say we cut out x (and its subtree).
    - Unmark x, if it's marked.
- A node is marked if one of its children has been cut, since the last time it's been cut.
- The second time a node's children is cut out, we move the node (and its subtree) to the root list.
- Let y be x's parent.
  - If y is not marked, mark y, since we cut one of its children.
  - If y is already marked, move y and its subtree to the root list, and then unmark y.
  - Let z be y's parent.
  - If z is not marked, stop. Otherwise, cut z and move it to the root list, and repeat the previous steps for z's parent, etc.
- One decrease key can create a sequence of cascading cuts.

Example: decrease key



- (a) shows the original Fibonacci heap.
- (b) shows the heap after node 46 is decreased to 15.
- (c)-(e) show the cascading cuts after node 35 is decreased to 5.

Pseudocode for decrease key and delete

```
FIB-HEAP-DECREASE-KEY(H, x, k)
if k > x.key
    error "new key is greater than current key"
x.key = k
y = x.p
if y ≠ NIL and x.key < y.key
    CUT(H, x, y)
    CASCADING-CUT(H, y)
if x.key < H.min.key
    H.min = x
```

```
CUT(H, x, y)
remove x from the child list of y, decrementing y.degree
add x to the root list of H
x.p = NIL
x.mark = FALSE
```

```
CASCADING-CUT(H, y)
z = y.p
if z ≠ NIL
    if y.mark == FALSE
        y.mark = TRUE
    else CUT(H, y, z)
        CASCADING-CUT(H, z)
```

```
FIB-HEAP-DELETE(H, x)
FIB-HEAP-DECREASE-KEY(H, x, −∞)
FIB-HEAP-EXTRACT-MIN(H)
```

To delete a key, simply decrease its value to $-\infty$ and then do a extract-min.

Complexity for decrease key

- Let H denote the heap before the decrease key operation.
- Cutting out a node takes O(1) time.
- Suppose a decrease key operation creates c cascading cuts.
- Then the actual cost is O(c).
- For the amortized cost
  - Each cut creates one more tree in the root list.
  - It also removes one marked node.
  - After the decrease key, the root list contains t(H)+c trees.
  - It also contains $\leq m(H) - c + 2$ marked nodes.
    - c-1 nodes were unmarked by cascading cuts, and the last call to CASCADING-CUT may have marked a node.
  - So the change in potential is $\left(\left(t(H)\right) + c + 2(m(H) - c + 2)\right) - \left(\left(t(H) + 2m(H)\right)\right) = 4 - c$.
  - So the amortized cost is $O(c) + 4 - c = O(1)$ by scaling the hidden constant in the potential appropriately.

Bounding the max degree

- So far, all the operations have O(1) amortized cost, except extract-min (and delete, which calls extract-min).
- extract-min has amortized cost O(D(H)), where D(H) is the max degree of any node in the Fibonacci heap.
- Def The golden ratio is $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$.
  - ϕ is the positive solution to the equation $x^2 = x + 1$.
- Recall the Fibonacci F_n is defined by $F_0 = 0, F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for $n \geq 2$.
  - The sequence is 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...
- Fact 1 $F_n = \lfloor \frac{\phi^n}{\sqrt{5}} + \frac{1}{2} \rfloor$.
  - For a proof, see section 3.2 of *Introduction to Algorithms.*
- Fact 2 $F_{n+2} = 1 + \sum_{i=0}^{n} F_i$.
- Fact 3 $F_{n+2} \geq \phi^n$.
- We show $D(n) \leq \lfloor \log_\phi n \rfloor$.
- Def For any node let x.deg denote its degree, and size(x) be the number of nodes in x’s subtree (including x).

Bounding the max degree

- Lemma 1 Let x be a node in a Fibonacci heap, and suppose $x.deg = k$. Let $y_1, \ldots, y_k$ be the children of x, in the order they were linked to x, from earliest to latest. Then $y_1.deg \geq 0$, and $y_i.deg \geq i - 2$ for $i = 2, \ldots, k$.
- Proof Obviously $y_1.deg \geq 0$.
  - For $i \geq 2$, when y_i was linked to x, $y_1, \ldots, y_{i-1}$ were already children of x, and so x had degree $\geq i - 1$.
  - y_i was linked to x during CONSOLIDATE.
    - So when y_i was linked, we had $y_i.deg = x.deg \geq i - 1$.
  - Since y_i was linked to x, it could have lost at most one child.
    - As soon as y_i loses two children, it's cut and moved to the root list.
  - So $y_i.deg \geq i - 2$.

Bounding the max degree

- Lemma 2 Let x be a node in a Fibonacci heap, and suppose $x.deg = k$. Then $size(x) \geq F_{k+2} \geq \phi^k$.
- Proof We use induction on k. The bound holds for k = 0, 1. For higher k, let $y_1, \ldots, y_k$ denote the children of x.
  - By Lemma 1, $y_i.deg \geq i-2$ for $i \geq 2$.
  - So by induction, $size(y_i) \geq F_i$, for $i \geq 2$.
    - Also, $size(y_0), size(y_1) \geq 1$.
  - We have $size(x) \geq \sum_{i=0}^{k} size(y_i) \geq 2 + \sum_{i=2}^{k} size(y_i) \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2} \geq \phi^k$.
    - The last two equalities follow by Facts 2 and 3.
- Cor For any n node Fibonacci heap H, the max degree $D(H) = O(\log n)$.
- Proof Let x be any node in H, and let $k = x.deg$.
  - We have $n \geq size(x) \geq \phi^k$.
  - So $k \leq \lfloor \log_\phi n \rfloor$.