Fibonacci Heaps cont'd 10.11.

Fibonacci Heaps use "lazy" implementation of Insert, Min, and Union – if you never have an Extract Min, then its easy to just keep every node in the root list and a pointer to the Min.

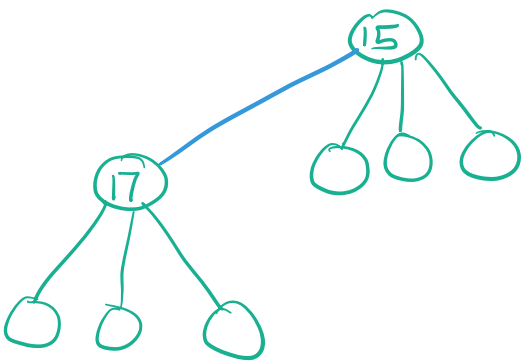
However, this means that the root list is largely unstructured – there is no way to find the next Min after an ExtractMin except to conduct a linear search of the root list.

General Amortization Strategy

When you have to do a large amount of work (like $O(r)$, where r is # nodes in the root list) also do $O(r^*)$ amount of "clean up" – ie reduce the potential for future work.

What we want CONSOLIDATE to accomplish?

- find the min root in rootlist
- merge trees in root list so there are fewer to search in future

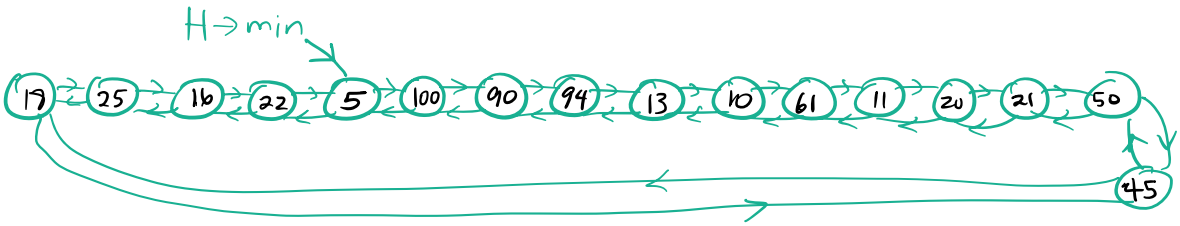


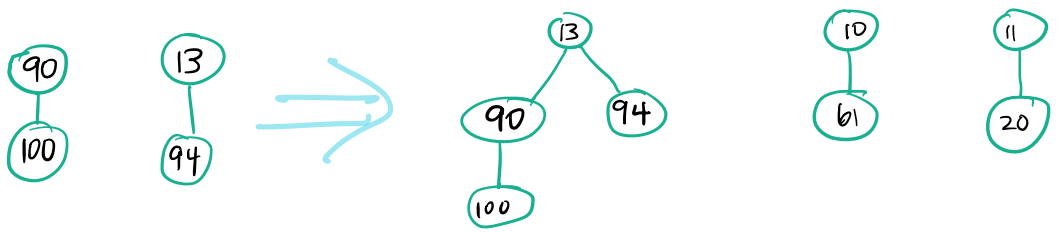
Find 2 trees of same degree d

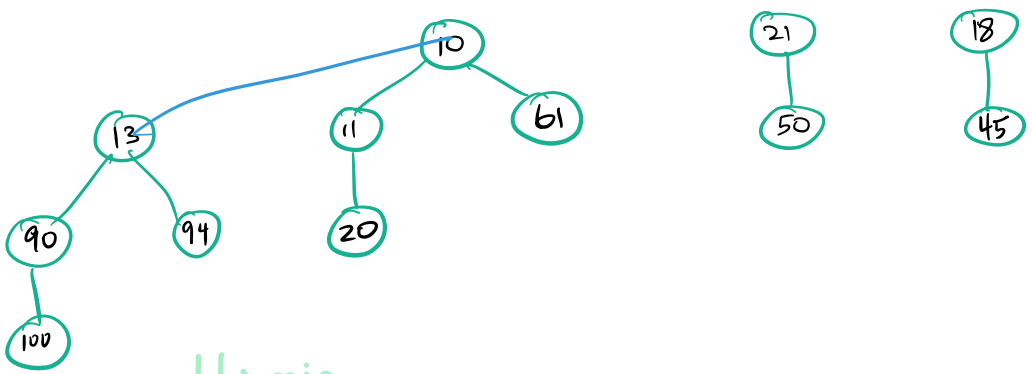
Make one be child of other, creating a d+1 degree tree

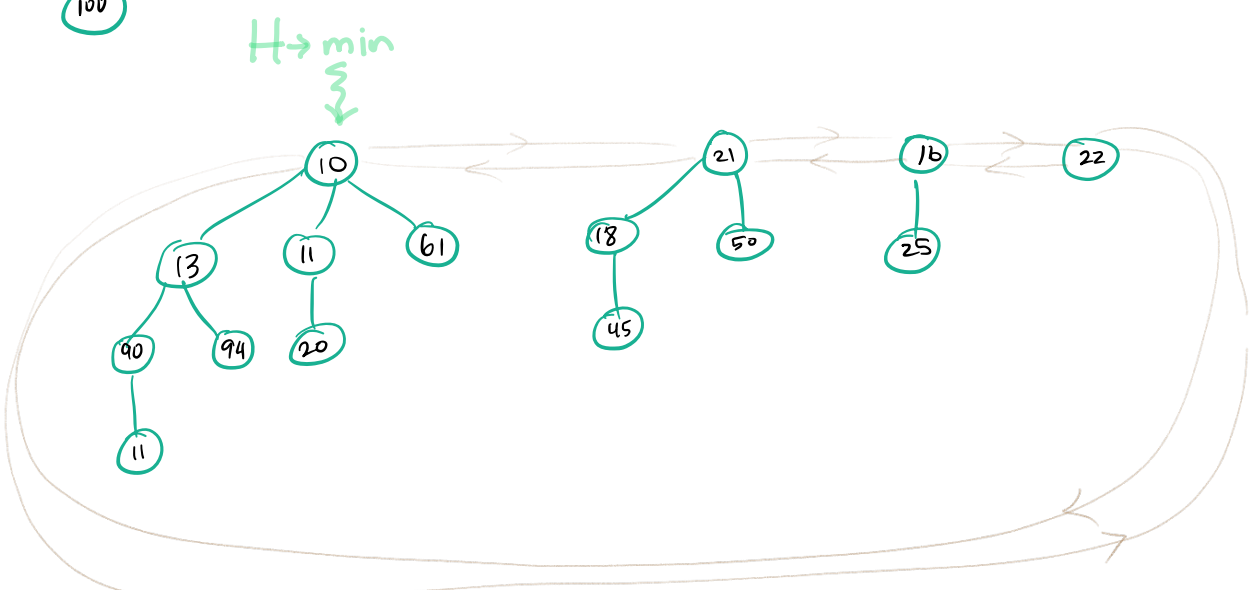
At the end, we want there to be 0 or 1 tree of each degree

How we want it to work, if we do 15 inserts and then an Extract Min:

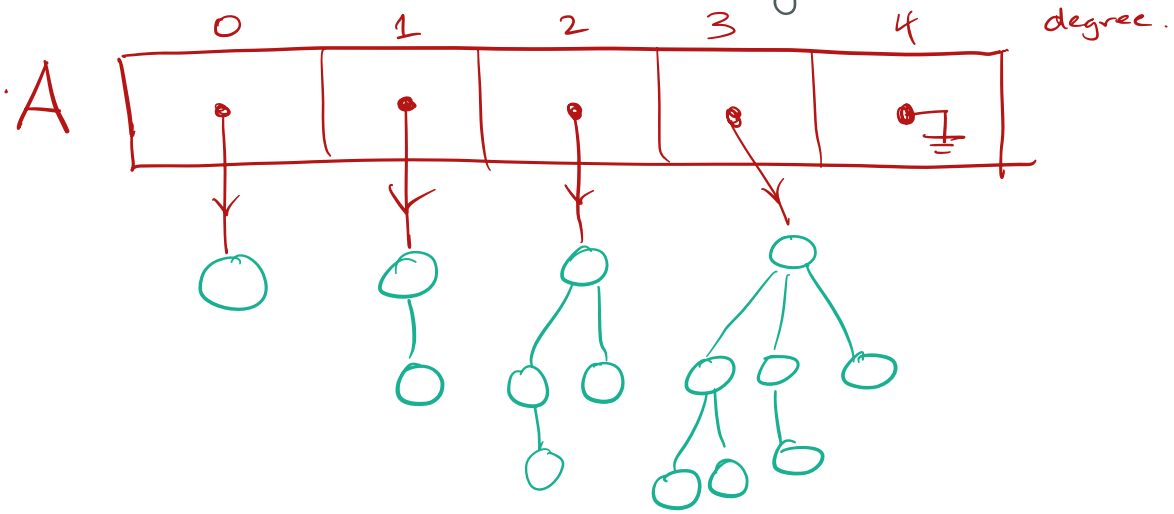








How do we accomplish this efficiently in code?



$A[0..D(n)]$ is an auxiliary array that is created during the CONSOLIDATE op'n

$D(n)$ is the maximum degree of any root in the heap of n nodes

CONSOLIDATE (H)

```
new A[0.. D(H,n)] of pointers to trees
for i = 0 to D(H,n)   A[i] = NULL
for each w in H→rootlist                    ← t(H)
    x = w;      d = x→degree
    while A[d] ≠ NULL
        y = A[d]
        if  x→key > y→key
            exchange x with y
            // now x should be merged tree's root
        FibHeapLink ( H, y, x )
        A[d] = NULL
        d = d+1
    A[d] = x
H→min = NULL
for i = 0 to D(n)
    if A[i] ≠ NULL
        if H→min == NULL
            create rootlist just containing A[i]
            H→min = A[i]
```

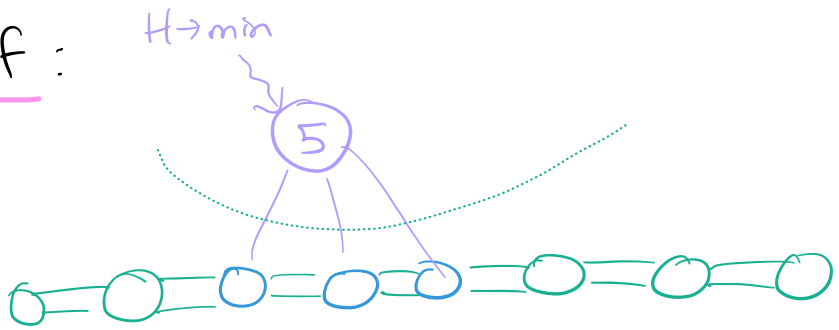
O(D(n))

```
else
    insert A[i] into H's rootlist
    if A[i]->key < H->min->key
        H->min = A[i]
```

```
FibHeapLink(H, y, x)
    remove y from rootlist of H
    make y a child of x
    x->degree++
    y->mark = FALSE        //"mark" only if node has lost
                           // 2 children since it got its current parent
```

Claim: Amortized cost of ExtractMin is $O(D(n))$

Proof:



Actual cost to add H->min's children to root list is $O(D(n))$

```
for each w in rootlist
    ....
```

Let's count up the work in this part - Aggregate Analysis.

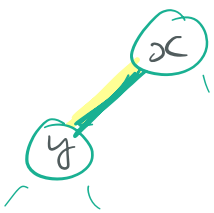
↳ Size of rootlist at this point is

$\leq t(H) - 1 + D(n)$

$t(H) - 1$: roots except extracted min

$D(n)$: new roots that were children of min

- The while loop is $O(1)$
- each time it gets executed, the number of trees in root list is diminished by 1



∴ total number of executions of while loop is $\leq t(H) - 1 + D(n)$

and each execution is $O(1)$

∴ Actual total work done in Extract Min is

$O(D(n) + t(H))$.

Also, $\Delta\phi = D(n) + 1 + 2m(H)$ (ϕ after)

$- (t(H) + 2m(H)) = D(n) + 1 - t(H)$ (ϕ before)

$$\therefore \text{work} + \Delta\Phi \in O(D(n) + t(H)) + O(D(n) + 1 - t(H))$$

$$\in O(D(n))$$

(we scale up the units of potential to dominate the constant hidden in $O(t(H))$)

Claim: $D(n) \in O(\lg n)$

Proof: later

Corollary: Extract Min has amortized running time $O(\lg n)$

Decrease key and Delete

```
Decrease key (H, x, k)
    if   k > x→key    error("new key not less")
    x→key = k
    y = x→p
    if  y ≠ NULL 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 y's child list; y→degree--
    add x to H's rootlist
    x→p = NULL
    x→mark = FALSE
```

$\Theta(1)$

```
CASCADING CUT ( H, y)
    z = y→p
    if  z ≠ NULL
        if  y→ mark == FALSE
```

y→mark = TRUE

else

CUT(H, y, z)

CASCADING_CUT(H, z)

How Decreasekey works.

- constant amount of work, $\Delta\phi = 0$ UNLESS it leads to violation of Heap Order.

If $x \to key < x \to p \to key$ then

- "cut" x (from $x \to p$) (put x in rootlist)
- if $x \to p$ has already had a child cut then
  - "cut" $x \to p$ from $x \to p \to p$
  - if $x \to p \to p$ has already had a child cut
    - "cut" $x \to p \to p \to p$

⋮

We use "mark" to tell us whether a node has already had a child cut. — only allowed 1 child cut since it got its current parent

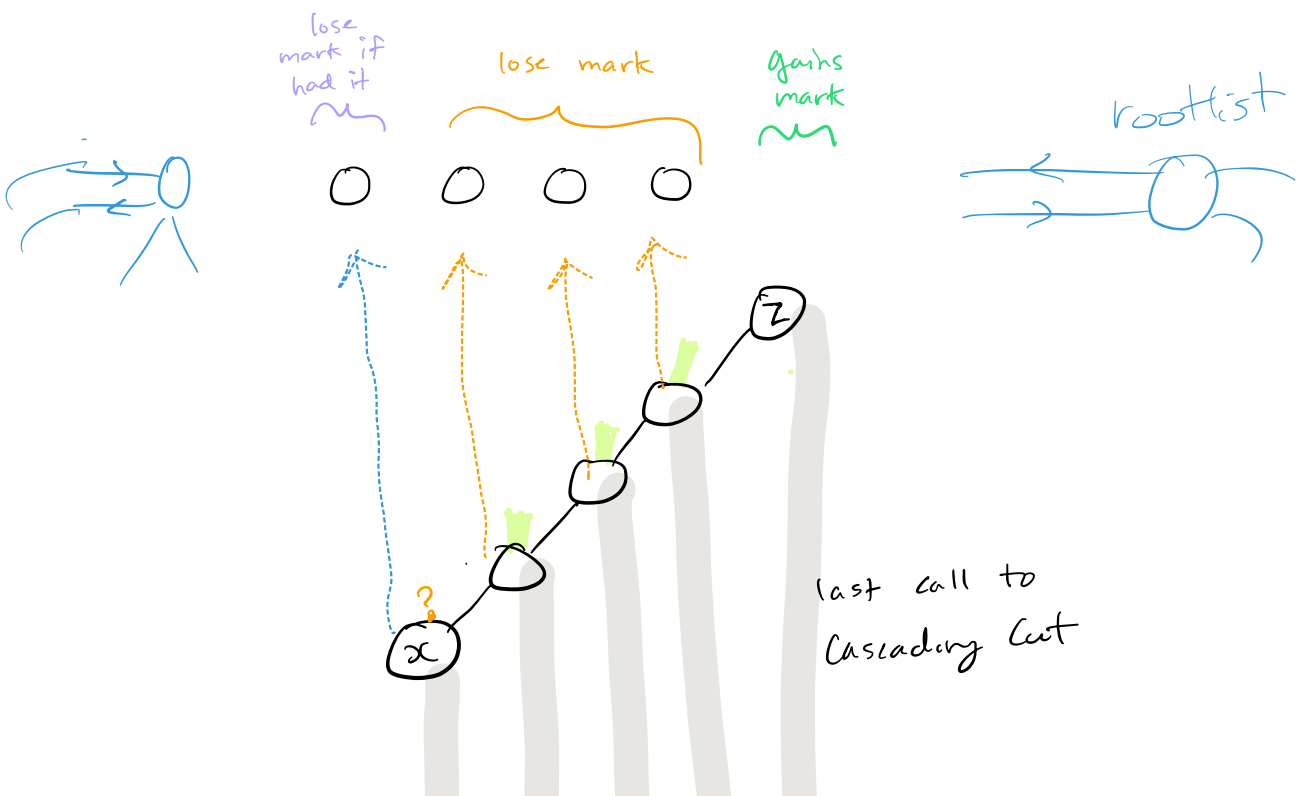
Claim: Amortized cost of DecreaseKey is $\Theta(1)$

Proof: Recall $\Phi = t(H) + 2m$

DecreaseKey is $\Theta(1)$ if no cascading cuts
(ie. either no cuts or just x is cut)
- reduces m by 0 or 1

Suppose DecreaseKey prompts C cascading cuts
(of $x \to p$, $x \to p \to p$, etc)

Of the C calls to cascading cut,
- the child's "mark" was true
and gets set to false



$\Delta\phi$ { t: +1 +1 +1 +1 0

2m: (0 or -2) -2 -2 -2 0 or 2 ?

work: +1 +1 +1 +1 +1 }

$2 + 0 + 0 + 0 + 3$

$\Delta\phi$ + work done $\leq$

$\therefore$ the DecreaseKey operation costs

$\text{work} + \Delta\phi = \Theta(1)$ } work in DecreaseKey not in CUT + CascadingCut

$+ \Theta(1) + 5$ } work in CUT and CASCADING CUT, aggregating the recursive calls.

$= \Theta(1)$ amortized time. ▨

CLRS

Delete (H, x)

DecreaseKey $(H, x, -\infty)$

ExtractMin (H)

DecreaseKey is $\Theta(1)$ amortized

ExtractMin is $\Theta(D(n))$ amortized

$\therefore$ Delete is $\Theta(D(n))$ amortized.