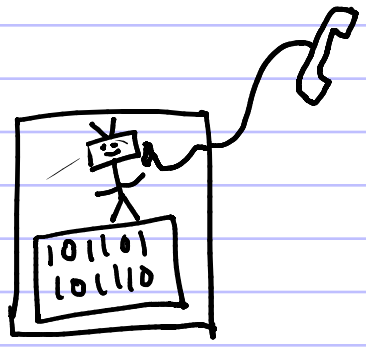
DATA STRUCTURES



"Add 7 to the set" `void ds.add(7);`

"Delete 19 from the list" `void ds.delete(19);`

"Is 3 in the table?" `bool ds.contains(3);`

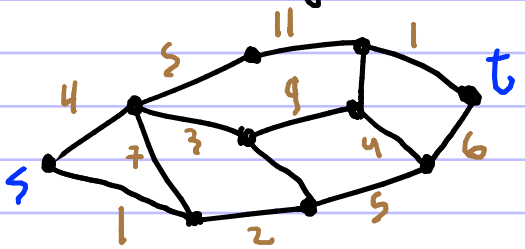
"How many things are in the list?" `int ds.size();`

OPERATIONS

Canonical data structures problem: design a data structure that has OPERATIONS (find, insert, delete) with RUNNING TIMES (O(log n), O(1), O(n)) that uses at most O(f(n)) SPACE. (f(n)=n, f(n)=n², f(n)=log n)

Example:
Design a data structure that supports insert(int), int minimum(), int deleteMinimum(), in O(log n), O(1), O(log n) that uses at most O(n) space.

Data structures are sometimes used as a standalone "database", but often are ingredients in algorithms.



G=(V,E)

Algorithmic problem

Shortest path problem: given a weighted undirected graph G=(V,E) and two vertices s,t∈V, find a minimum-weight path from s to t.

Dijkstra's Algorithm for Shortest Path Problem

Input: a weighted undirected graph $G=(V,E)$ and $s,t \in V$.

```
1. For all v∈V, D[v]=∞.
2. D[s]=0.
3. For all v∈V, P[v]=none.
4. Put all v∈V into a data structure Q, using Q.insert(v,D[v])
5. while(not Q.empty())
6.     c=Q.removeMin()
7.     for all neighbors v of c
8.         ω=weight of (c,v)∈E
9.         if (D[v]>D[c]+ω)
10.            D[v]=D[c]+ω
11.            P[v]=c
12.            Q.decreaseKey(v,D[v])
13. Return D[t] and, optionally, P.
```

Supports: void insert(vertex, int) bool empty()
void decreaseKey(vertex, int)
vertex removeMin()

Q: A priority Queue

Running time: Lines 1-3: $O(|V|)$ + Line 4: $O(|V| \cdot$ time of Q.insert()$)$ +
Line 5: $O(|V| \cdot ($time of Q.empty() + Line 6: time of Q.removeMin() + Line 7, Lines 8-11, Line 12: $|E| \cdot (O(1)$ + time of Q.decreaseKey()$))$
+ Line 13: $O(|V|)$

$= O(|V| \cdot ($time of Q.insert() + time of Q.empty() + time of Q.removeMin()$)$
$+ O(|E| \cdot$ time of Q.decreaseKey()$)$

Use a heap

insert → $O(\log n)$
empty → $O(1)$
removeMin → $O(\log n)$
decreaseKey → $O(\log n)$
$n = O(|V|)$

$= O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$

Are you happy with this?

$\Omega(|E|)$ for any algorithm

The Battle of the Operations

$|V| \cdot$ Q.insert()
Q.empty()
Q.removeMin()

vs. $|E| \cdot$ Q.decreaseKey()

Heaps (regular kind):

$|V| \cdot$ $O(\log|V|)$
$O(1)$
$O(\log|V|)$

vs. $|E| \cdot O(\log|V|)$

If $|E| = O(|V|)$ then equal, otherwise $O(|E|\log|V|)$ wins.

sparse graph

So "just" design a new Q with faster decreaseKey(), replacing choice of Q = heap in Dijkstra's. BOOM

Faster algorithm.

Data structures are recurring, modular, memory problems.

One approach to faster data structures: remove as much "structure" as possible.

	Sorted array	Balanced BST:	Heap:	Pile of junk w/pointers:
insert()	$O(n)$	$O(\log n)$	$O(\log n)$	$O(1)$
minimum()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
	LAME	SUCH STRUCTURE	MEH	LOL

Another approach is to assume more about how the data structure is used:

"I only have fast delete() on average"

AMORTIZED ANALYSIS

FIBONACCI HEAPS IN DIJKSTRA'S ALGORITHM

Like regular heap, fibonacci heap can act as a priority queue.

void insert(v, int)] O(1) vs. O(log n)

bool empty()] O(1) vs. O(1)

vertex removeMin()] O(log n) amortized vs. O(log n)

void decreaseKey(v, int)] O(1) amortized vs. O(log n)

Gives Dijkstra's algorithm $O(|V|\log|V| + |E|)$ running time vs.

$O(|E|\log|V|)$

If $|E| = \Theta(|V|)$, then $O(|E|\log|E|)$ vs. $O(|E|\log|E|)$ sparse

$|E| = \Theta(|V^2|)$, then $O(|E|)$ vs. $O(|E|\log|E|)$ dense

If $|E| = O(|V|\log|V|)$, then $O(|E|\log|E|)$ vs. $O(|E|\log|E|)$

$|E| = \Omega(|V|\log|V|)$, then $O(|E|)$ vs. $O(|E|\log|E|)$

Using Fibonacci Heap improves running time for most graphs by log factor.

Possible to achieve $o(|E|\log|E|)$ for sparse graphs by using an even better priority queue in Dijkstra's algorithm?

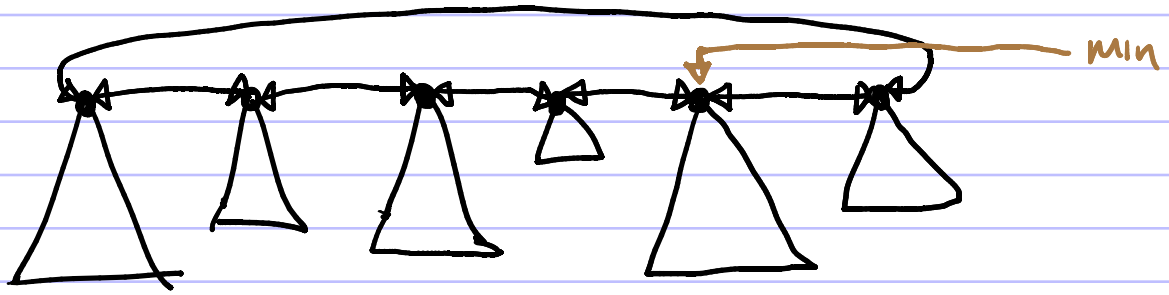
Homework: Show that no priority queue data structure will allow Dijkstra's to run in $o(|V|\log|V|)$. Hint: sorting takes $\Omega(n\log n)$ time.

Project: Many variants of shortest path algorithms using various assumptions exist. See "Recent results on the single-source shortest path problem", Raman, 1997. Survey one vein of these and the history of improvements.

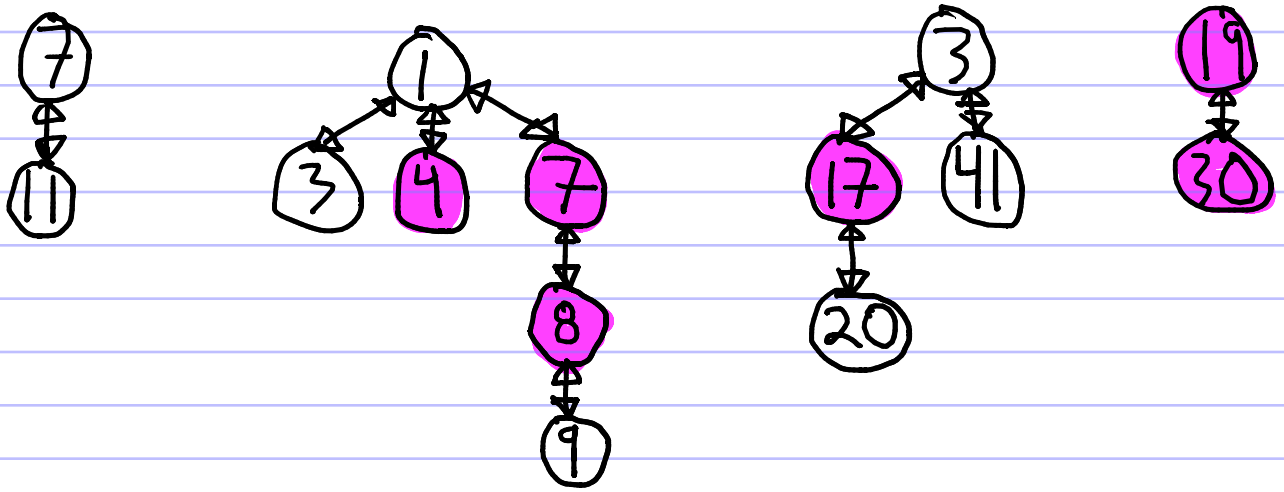
Thorup, Fredman & Willard, Raman

FIBONACCI HEAPS [Fredman, Tarjan 1987]

A forest of heaps in a doubly connected linked list by their roots.
Also a global pointer to min element (root of some heap):

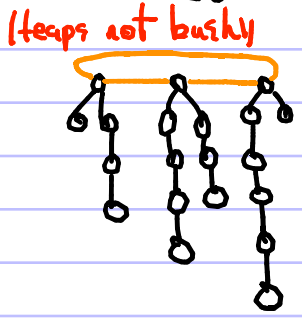


Each heap is <u>not</u> a usual binary heap. Nodes have a variety of # children (known by node), is doubly linked with parent, children. Each node is also marked or not.



Idea: Lazily insert(), decreaseKey(). Use deleteMin() to do cleanup.
Cleanup keeps heaps bushy and few.

BAD: Updating global min after deleteMin() is slow.



BAD: decreaseKey() is slow.

AMORTIZED ANALYSIS

"Lazily" insert() and decreaseKey()?
"Cleanup" with deleteMin()?

What Time Figures?

Fibonacci heap deleteMin() actual takes $\Theta(n)$ time in the worst case, e.g.

since we need to search through length n list to update global min. But we do cleanup so this doesn't happen often.

Worst case analysis doesn't care how often bad case happens. This makes sense if you only do it once.

In Dijkstra's, we call deleteMin() $|V|$ times. We only care about total time across all these calls.

Amortized analysis lets you measure across calls to give a more precise total running time.

Idea: measure running time + change in messiness of data structure.
"Amortized time" = running time + change in messiness.

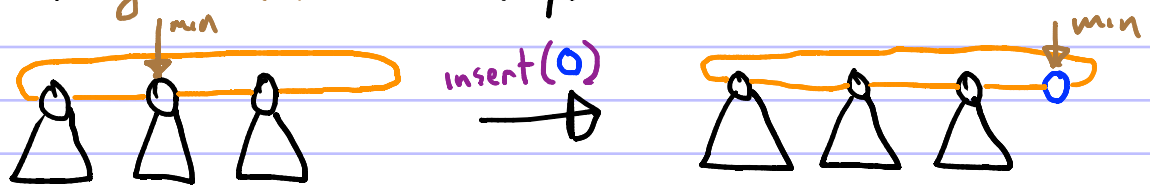
"Lazy" operations: small running time + positive change in messiness.
"Cleanup" operations: big running time + negative change in messiness.

Give cleanup operations the credit they deserve!
By taxing lazy operations for making a mess.

Fibonacci heap messiness: # heaps + 2x # marked nodes

insert()

1. Put new element in root linked list as a heap of one element.
2. Update global min if necessary.



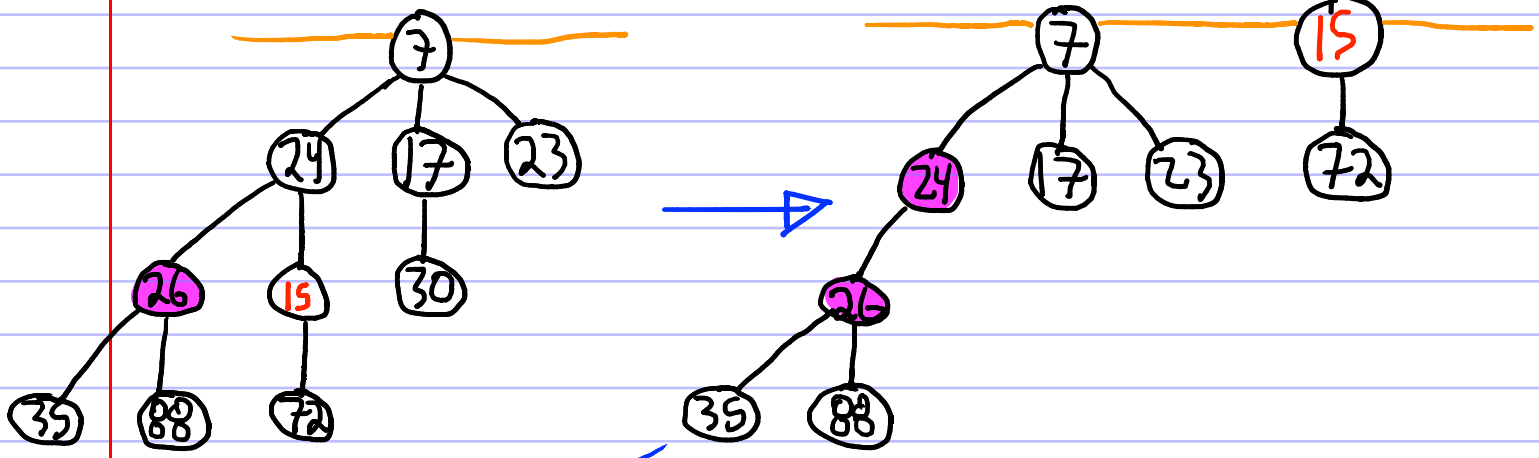
$O(1)$ time. Also $O(1+1)=O(1)$ amortized time.

t increased by 1

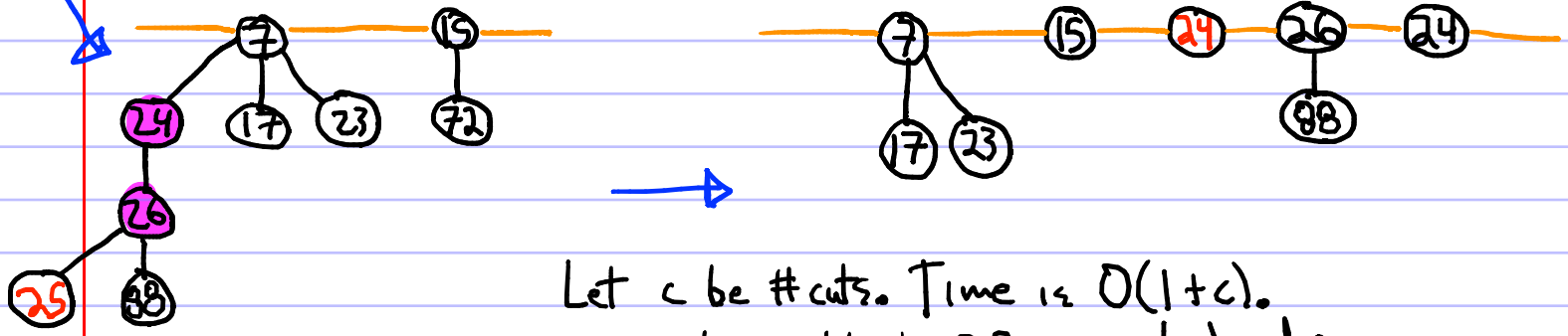
decreaseKey(v, int)

1. If change in value doesn't violate heap properties, check if global min needs to be updated, done. $O(1)$ time.

2. Otherwise, unmark v, cut subtree rooted at v and place it in root list. Then mark former parent of v. Update global min.



3. If parent is already marked, also cut parent's subtree and put it in root node list. Mark grandparent. If grandparent is already marked ...



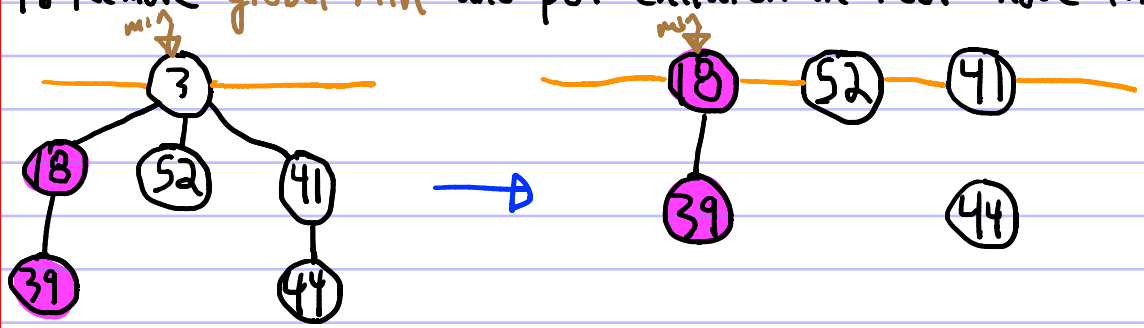
Let c be # cuts. Time is $O(1+c)$.

c more trees, at least c-2 fewer marked nodes.

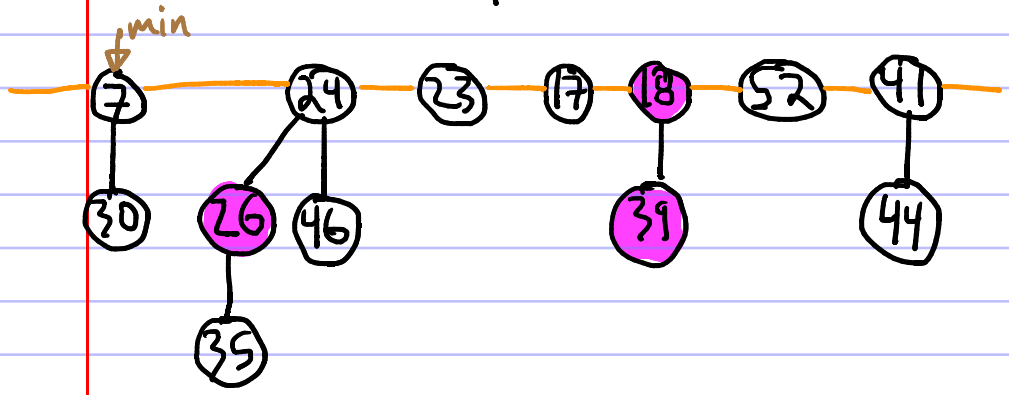
So $O(1+c+(c-2(c-2)))=O(1)$ amortized

deleteMin() The workhorse

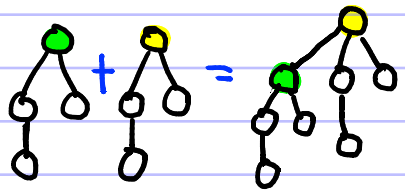
1. Remove global min and put children in root node list. Update globalmin

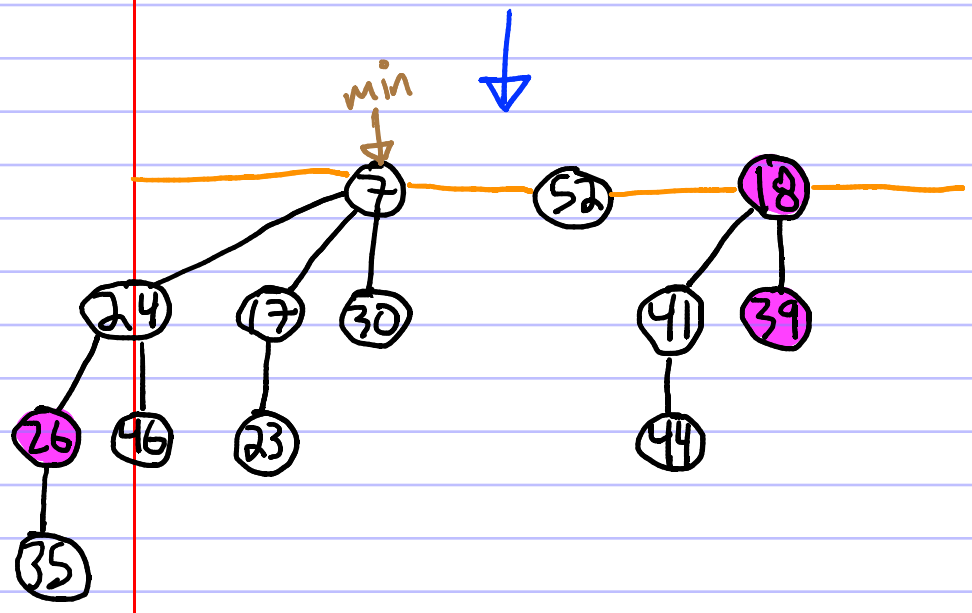


2. Consolidate heaps so no two roots have same # children.



Scan left to right, merge two trees when they have same root # children.





trees in $H' \leq D(n)+1$:
Each has a root with different degree between 0 & $D(n)$.

Let pre deleteMin() and post deleteMin() Fibonacci heaps be H, H'.
Let max # number of children of any node of a Fib. heap of size n be $D(n)$.

Time: $O($ remove g.m. $D(n)$ + Find new g.m. $\#\text{trees in } H + D(n)$ + consolidate $\#\text{trees in } H + D(n)) = O(\#\text{trees in } H + D(n))$

Change in messiness: $\#\text{trees in } H' - \#\text{trees in } H \leq D(n)+1-\#\text{trees in } H$

Amortized time: $O(\#\text{trees in } H + D(n) + D(n) + 1 - \#\text{trees in } H) = O(D(n))$

Summary of amortized costs

insert() $O(1)$ worst case

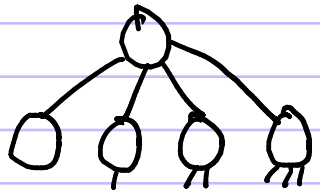
decreaseKey() $O(1)$ amortized

deleteMin() $O(D(n))$ amortized

And what is $D(n)$? (We ~~want~~ need $D(n) = O(\log n)$ for Dijkstra's)

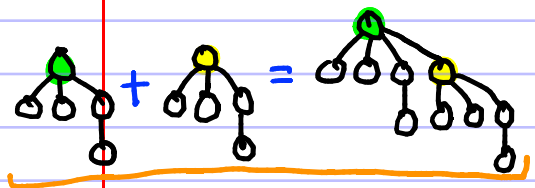
$D(n) = O(\log n)$ if $n = c^{D(n)}$, i.e. n is exponential in $D(n)$.

Consider a node in a Fibonacci heap and order children in the order they became children.] via consolidation



The ith child is the root of a heap with degree i, since this child had same degree as parent at consolidation

So the size of a heap whose root is degree i is $\geq$ 2 × size of a heap whose root is degree $i-1$.



consolidation

Let $F(i)$ be size of heap whose root is degree i.
Then $F(i) = 2F(i-1)$, $F(0) = 1$. So $F(i) = 2^i = n$ & $i = \log n$.
So $D(n) = \log_2(n)$.

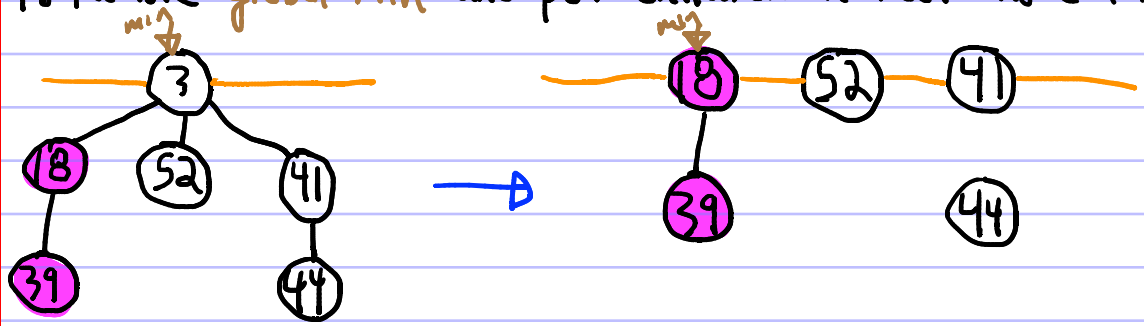
EXCEPT: May lose grandchild during decreaseKey() after consolidation. Marked child
So actually $F(i) = F(i-1) + F(i-2)$, $F(0) = 1$, $F(1) \geq 1$.

So $F(i) \geq F(i-1) + F(i-2)$, $F(0) = 1$, $F(1) \geq 2$.
So $F(i) \geq 1.618^i$ & $n \geq 1.618^{D(n)}$ & $\log_{1.618} n \geq D(n)$ & $D(n) = O(\log n)$.

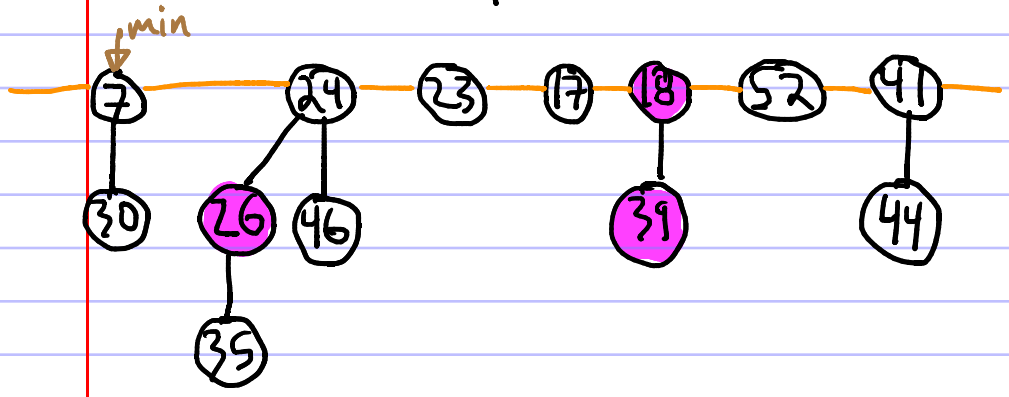
So deleteMin() runs in $O(\log n)$ amortized time.

deleteMin() The workhorse

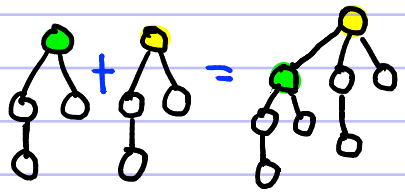
1. Remove global min and put children in root node list. Update globalmin

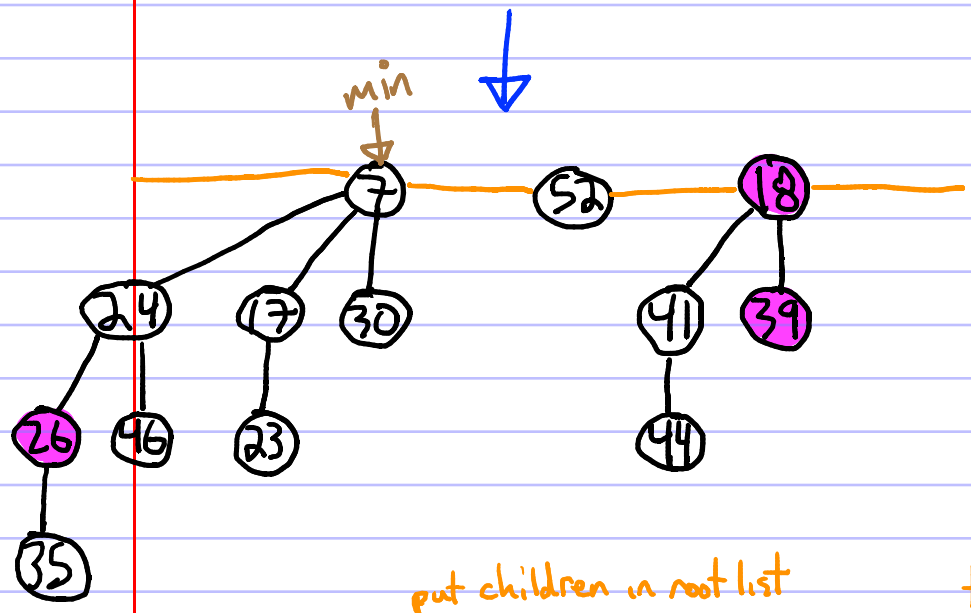


2. Consolidate heaps so no two roots have same # children.



Scan left to right, merge two trees when they have same root #children.





put children in root list — Find new global min + consolidate heaps

Time: $O(\#\text{children of global min}) + O(\#\text{children of global min} + \#\text{heaps})$

Change in messiness: $\#\text{consolidated heaps} - \#\text{heaps} \leq \max_{\text{root}} \#\text{children of any root} - \#\text{heaps}$

Amortized time: $O(\#\text{children of global min} + \max \#\text{children of any root})$