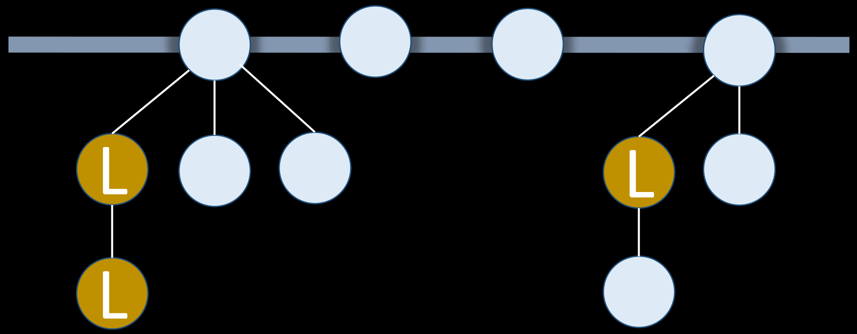
TODAY'S TOPIC 1/3

Analysis of the Fibonacci Heap



- Stores a collection of trees, each of them a heap
- Nodes that have lost one child are marked L and nodes that lose two children are disowned by their parents

floordrobe strategy
"keep trees bushy"

$\longrightarrow$ algorithm

algorithm $\longrightarrow \Phi$

algorithm $+ \Phi \longrightarrow$ amortized analysis

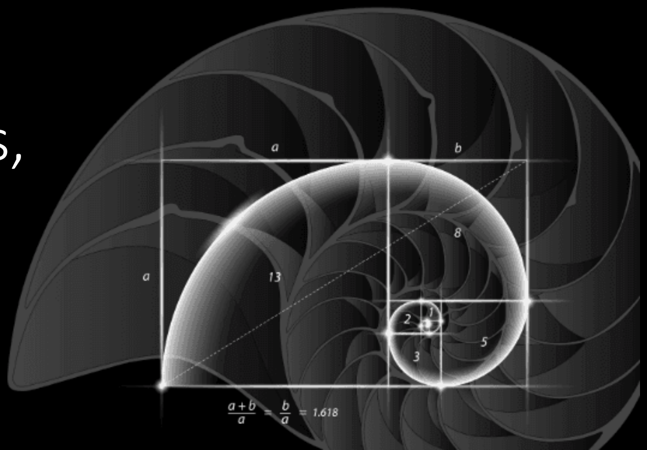
For good amortized costs, we want

$$\text{degree} = O(\log N).$$

Does our algorithm actually achieve this?

TODO: SHAPE THEOREM

In a Fibonacci heap with $\leq N$ items, every node has degree $\leq \log_\phi N$



SHAPE THEOREM

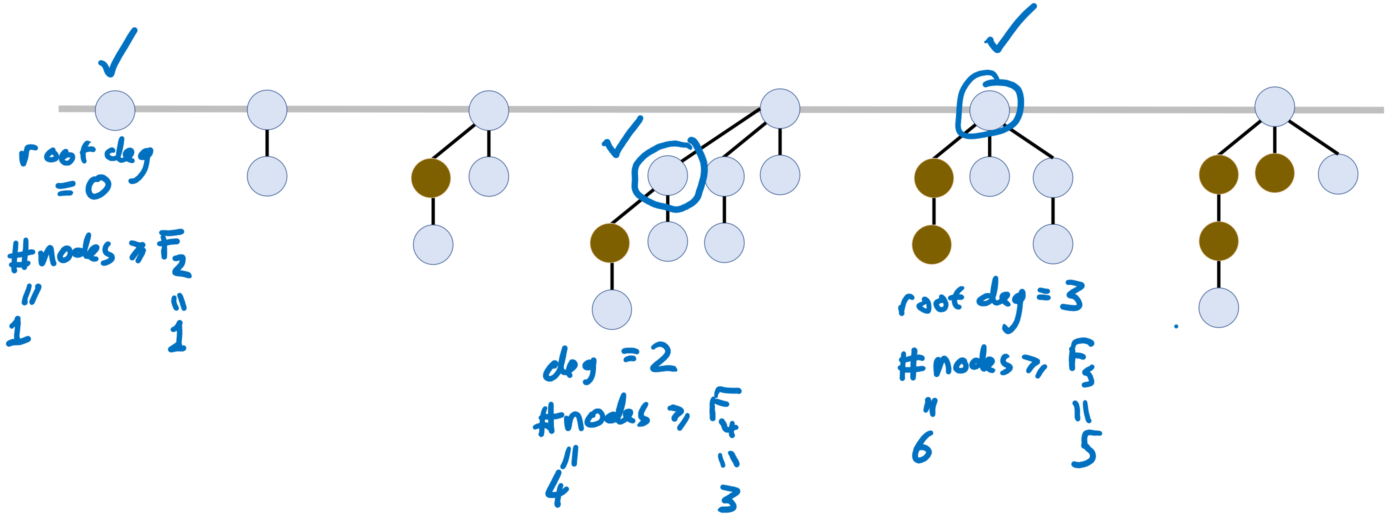
In a Fibonacci heap with N items, every node has degree $\leq \log_{\phi} N$

Fib. numbers

F_1	F_2	F_3	F_4	F_5	F_6	F_7
1	1	2	3	5	8	13

SHAPE LEMMA

Consider a subtree in a Fibonacci heap. If the subtree's root has d children, then the number of nodes in the subtree is $\geq F_{d+2}$ where $F_1, F_2, \ldots$ are the Fibonacci numbers



SHAPE THEOREM

In a Fibonacci heap with N items, every node has degree $\leq \log_\phi N$

Proof of theorem.
Pick a node with maximum degree, call it d,
and consider the subtree rooted at this node.

$$
\begin{aligned}
N &\geq \text{num.nodes in subtree} \\
&\geq F_{d+2} \quad \text{by lemma} \\
&\geq \phi^d \quad \text{linear algebra:}
\end{aligned}
$$

Hence $d \leq \log_\phi N$.

$$F_n = \frac{\varphi^n - (-\varphi)^n}{\sqrt{5}}.$$

SHAPE LEMMA

Consider a subtree in a Fibonacci heap. If the subtree's root has d children, then the number of nodes in the subtree is $\geq F_{d+2}$ where $F_1, F_2, \ldots$ are the Fibonacci numbers

Recall: in a binomial heap...

A binomial tree whose root has degree d has 2^d nodes.

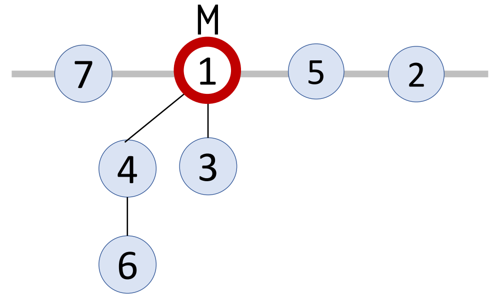
In a binomial heap,

$$\underset{\text{total \#nodes}}{N} \geq \underset{\text{largest tree}}{\text{\#nodes in}} = \underset{\substack{d = \text{root degree of}\\ \text{largest tree}}}{2^d} \Rightarrow \underset{\text{in entire heap}}{d = \text{largest deg.}}$$

max degree $\leq \log_2 N$.

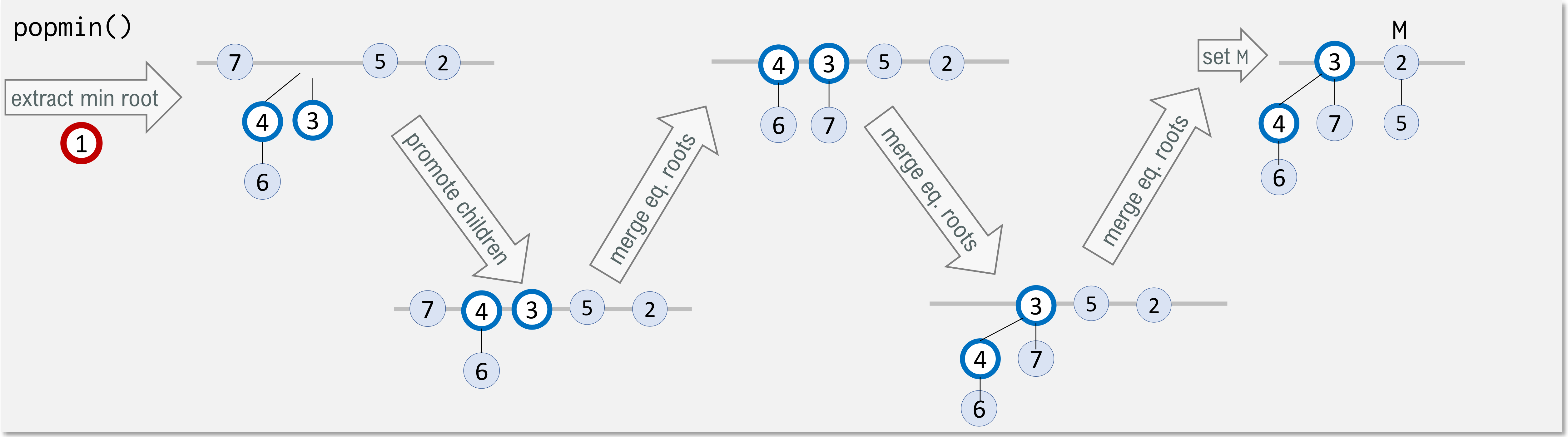
SHAPE LEMMA

Consider a subtree in a Fibonacci heap. If the subtree's root has d children, then the number of nodes in the subtree is $\geq F_{d+2}$ where $F_1, F_2, \ldots$ are the Fibonacci numbers




```
12  def popmin():
13      take note of minroot.value and minroot.key
14      delete the minroot node, and promote its children to be roots
15      # cleanup the roots
16      while there are two roots with the same degree:
17          merge those two roots, by making the larger root a child of the smaller
18      update minroot to point to the root with the smallest key
19      return the value and key we noted in line 13
```

This merging is the only mechanism through which a node can acquire children.



SHAPE LEMMA

Consider a subtree in a Fibonacci heap. If the subtree's root has d children, then the number of nodes in the subtree is $\geq F_{d+2}$ where $F_1, F_2, \ldots$ are the Fibonacci numbers

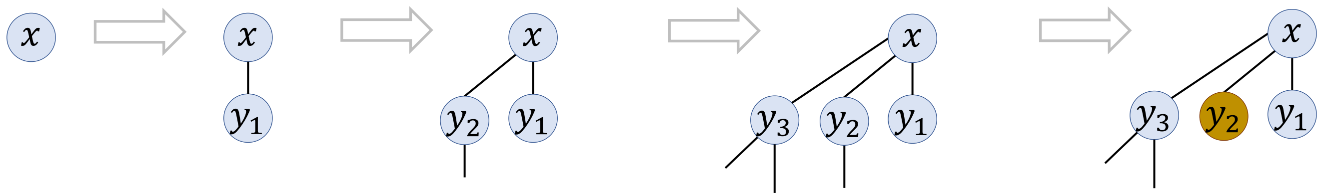
GRANDCHILD RULE

A node x is said to satisfy the grandchild rule if its children can be ordered, call them $y_1, \ldots, y_d$, such that for all $i \in \{1, \ldots, d\}$

$$\text{num. grandchildren of } x \text{ via } y_i \geq i - 2$$

ALGORITHMIC CLAIM

In a Fibonacci heap, at every instant in time, every node x satisfies the grandchild rule, when we order its children $y_1, \ldots, y_d$ by when they became children of x



when x acquired y_2, x had a child already, so y_2 did too

when x acquired y_3, x had two children already, so y_3 did too

each y_i might have lost a single child

y_1 now has ≥ 0 children
y_2 now has ≥ 0 children
y_3 now has ≥ 1 child
$\vdots$
y_d now has $\geq d - 2$ children

SHAPE LEMMA

Consider a subtree in a Fibonacci heap. If the subtree's root has d children, then the number of nodes in the subtree is $\geq F_{d+2}$ where $F_1, F_2, \ldots$ are the Fibonacci numbers

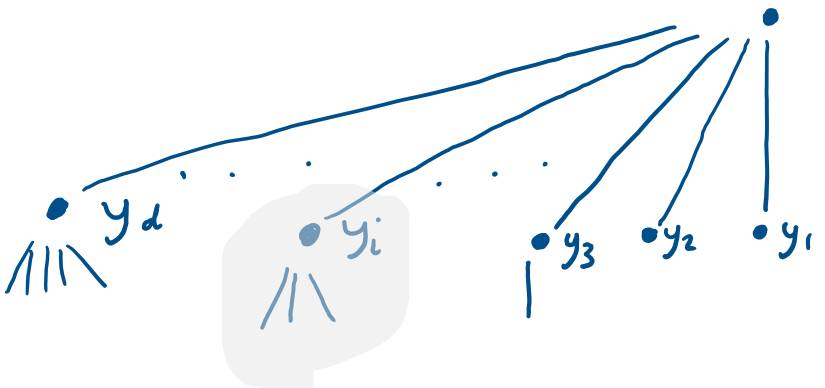
GRANDCHILD RULE

A node x is said to satisfy the grandchild rule if its children can be ordered, call them $y_1, \ldots, y_d$, such that for all $i \in \{1, \ldots, d\}$

num. grandchildren of x via $y_i \geq i - 2$

MATHEMATICAL CLAIM

Consider a tree where *all* nodes satisfy the grandchild rule. Let N_d be the smallest number of nodes in a tree whose root has d children. Then $N_d = F_{d+2}$.



num.nodes in tree $\geq N_{d-2} + N_{d-3} + \cdots + N_1 + N_0 + N_0 + 1$

child y_i has degree $\geq i - 2$,
so its subtree has $\geq N_{i-2}$ nodes

$$N_d = N_{d-2} + N_{d-3} + \cdots + N_0 + N_0 + 1$$

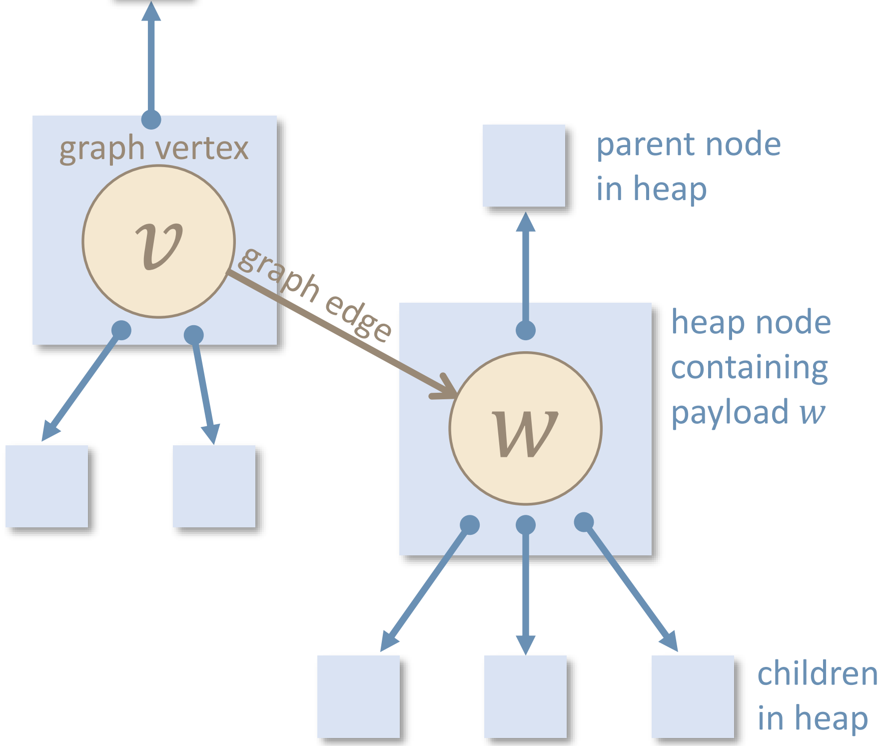
$$N_{d-1} = N_{d-3} + \cdots + N_0 + N_0 + 1$$

$$\Rightarrow \quad N_d = N_{d-2} + N_{d-1}$$

$\Rightarrow$ N_d is Fibonacci number

SECTION 7.7

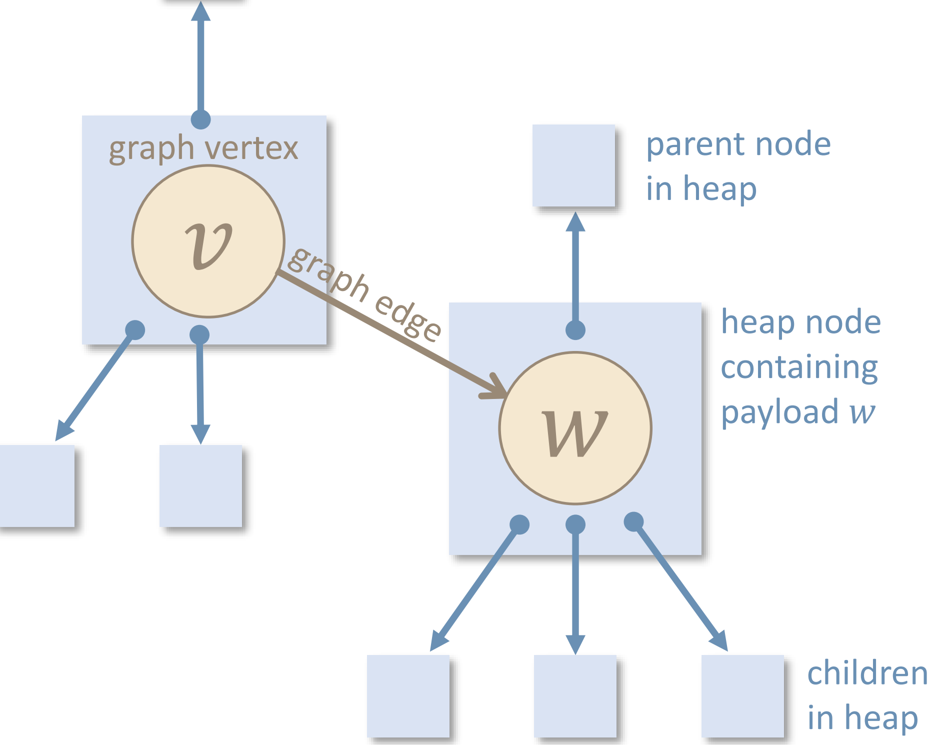
Implementing the Fibonacci heap




```
def dijkstra(g, s):
    ...
    toexplore = PriorityQueue()
    toexplore.push(s, key=0)

    while not toexplore.is_empty():
        v = toexplore.popmin()
        for (w,edgecost) in v.neighbours:
            dist_w = v.distance + edgecost
            ...
                toexplore.decreasekey(w, key=dist_w)
```

QUESTION. **How can decreasekey be $O(\log N)$?**

Doesn't it take $O(N)$ in the first place, to find the heap node that we want to decrease?




```
def dijkstra(g, s):
    ...
    toexplore = PriorityQueue()
    toexplore.push(s, key=0)

    while not toexplore.is_empty():
        v = toexplore.popmin()
        for (w,edgecost) in v.neighbours:
            dist_w = v.distance + edgecost
            ...
              toexplore.decreasekey(w, key=dist_w
```

Algorithms tick: fib-heap

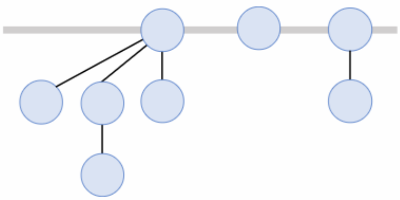
cl.cam.ac.uk/teaching/2223/Algorithm2/ticks/fib-heap.html

Algorithms tick: fib-heap

Fibonacci Heap

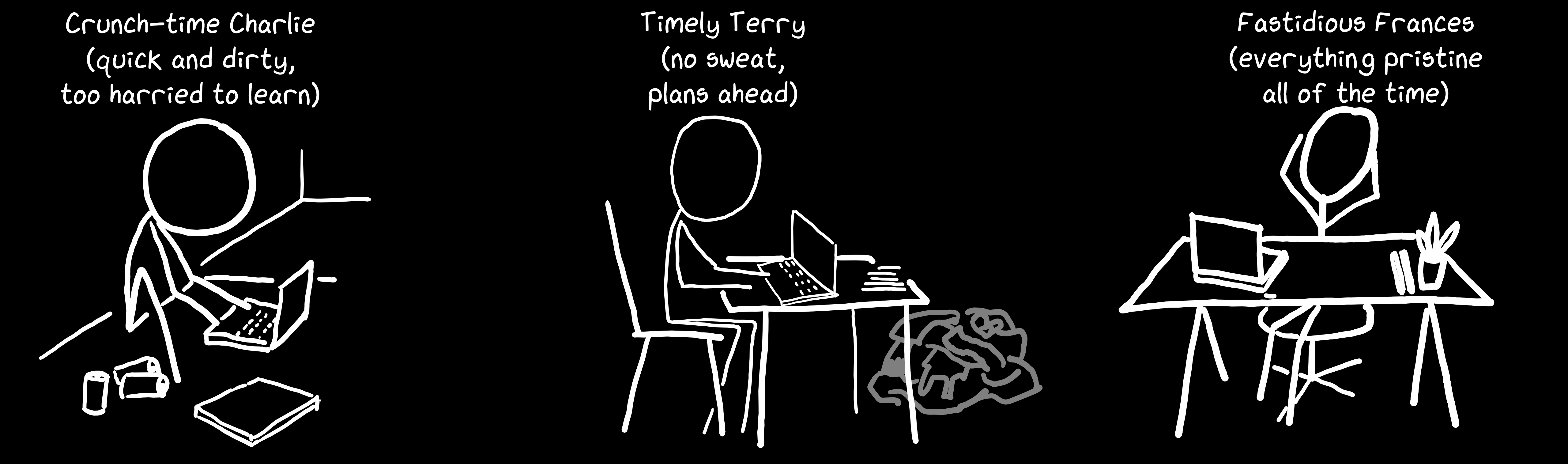
In this tick you will implement the Fibonacci Heap. This is an intricate data structure – for some of you, perhaps the most intricate programming you have yet programmed. If you haven't already completed the dis-set tick, that's a good warmup.

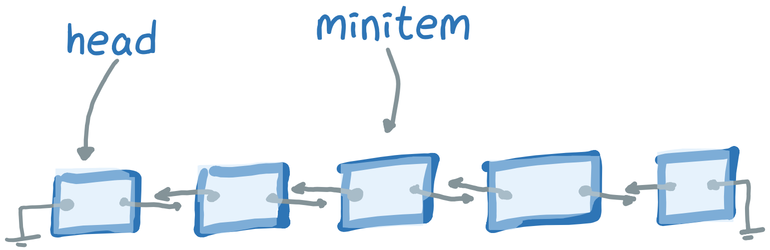
Step 1: heap operations

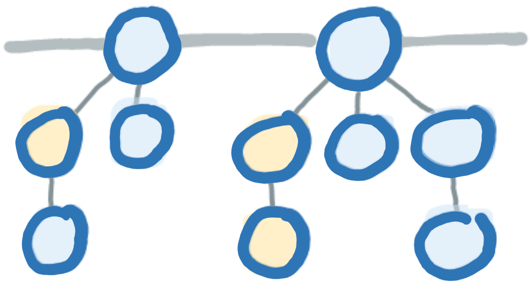


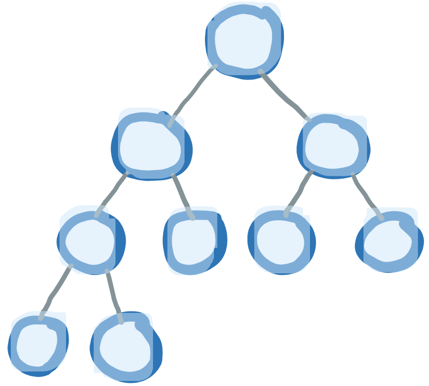
The first step is to implement a `FibNode` class to represent a node in the Fibonacci heap, and a `FibHeap` class to represent the entire heap. Each FibNode should store its priority key `k`, and the FibHeap should store a list of root nodes as well as the minroot.

TODAY’S TOPIC 3/3









LINKED LIST PRIORITY QUEUE

push is fast, $O(1)$

popmin is slow, $O(N)$

FIBONACCI HEAP

push is fast, $O(1)$

popmin is fast, $O(\log N)$

BINARY HEAP

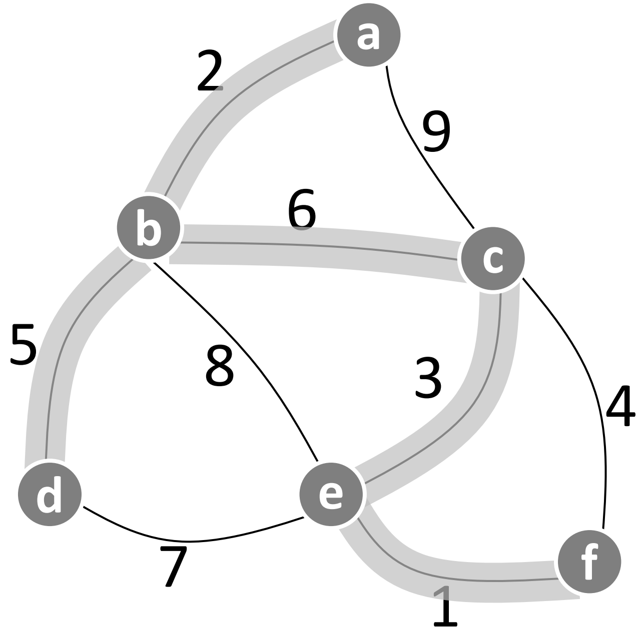
push is slow, $O(\log N)$

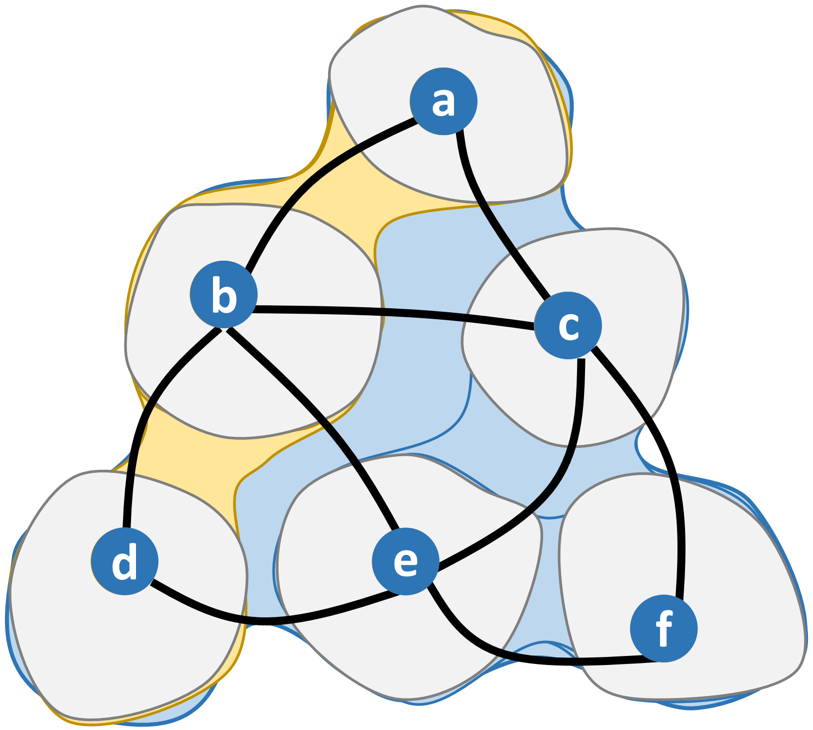
popmin is fast, $O(\log N)$

TODAY’S TOPIC 3/3

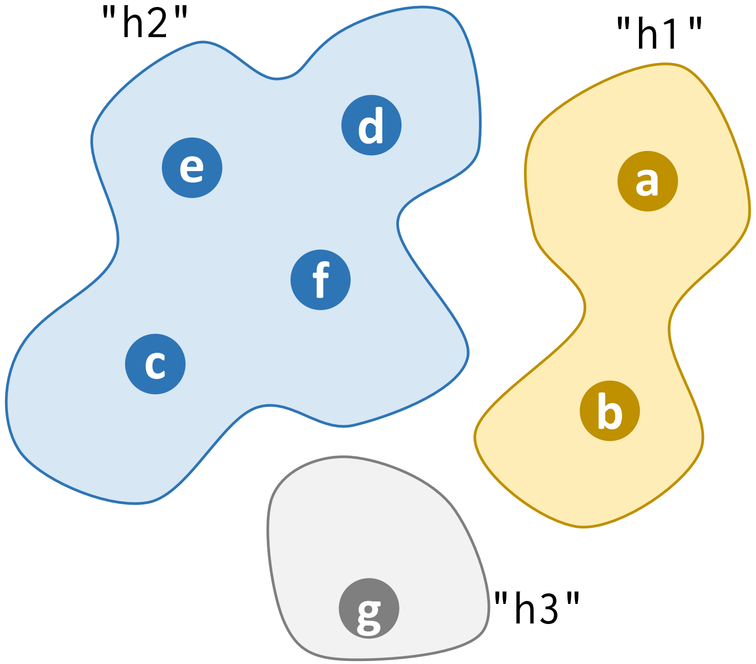
SECTION 7.9

Disjoint sets





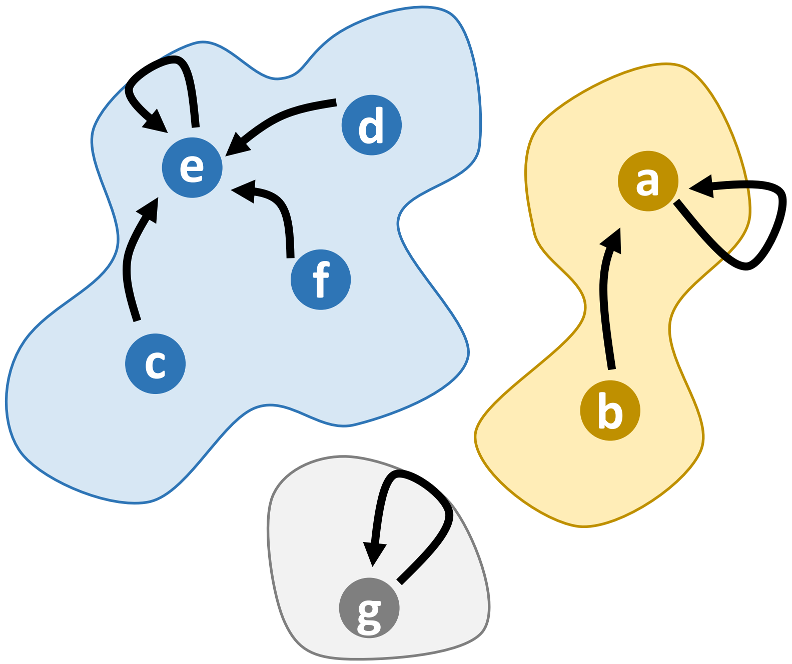

```
def kruskal(g):
    tree_edges = []
    partition = DisjointSet()
    for v in g.vertices:
        partition.add_singleton(v)
    edges = sorted(g.edges, sortkey = λ(u,v,weight): weight)

    for (u,v,edgeweight) in g.edges:
        p = partition.get_set_with(u)
        q = partition.get_set_with(v)
        if p != q:
            tree_edges.append((u,v))
            partition.merge(p, q)
```



IMPLEMENTATION 0

```
handles = {a:"h1", b:"h1", c:"h2", d:"h2", e:"h2", f:"h2", g:"h3"}

def merge(x,y):
    for every item in the entire collection:
        if the item's handle is y then update it to be x
```

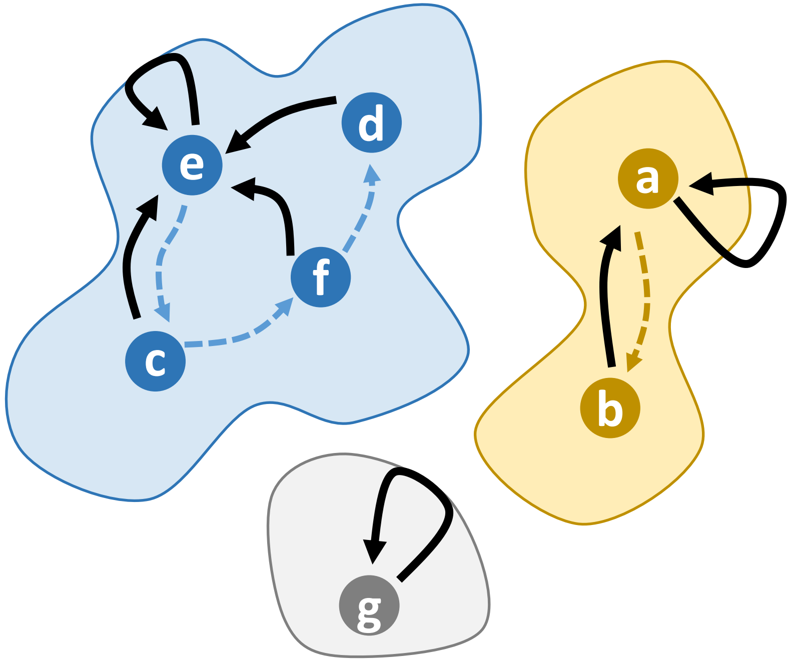
```
AbstractDataType DisjointSet:
    # Holds a dynamic collection of disjoint sets

    # Add a new set consisting of a single item (assuming it's not been added already)
    add_singleton(Item x)

    # Return a handle to the set containing an item.
    # The handle must be stable, as long as the DisjointSet is not modified.
    Handle get_set_with(Item x)

    # Merge two sets into one
    merge(Handle x, Handle y)
```



IMPLEMENTATION 0'

Each item points to a representative item for its set

`handles = {a:a, b:a, c:e, d:e, e:e, f:e, g:g}`



IMPLEMENTATION 1 "FLAT FOREST"

Each item points to a representative item for its set
Each set has a linked list, starting at its representative

so I can just walk through items that need to be updated, on merge.

```
def merge(x,y):
    for every item in set y:
        update it to belong to set x
```

worst-case O(N), N=# items.

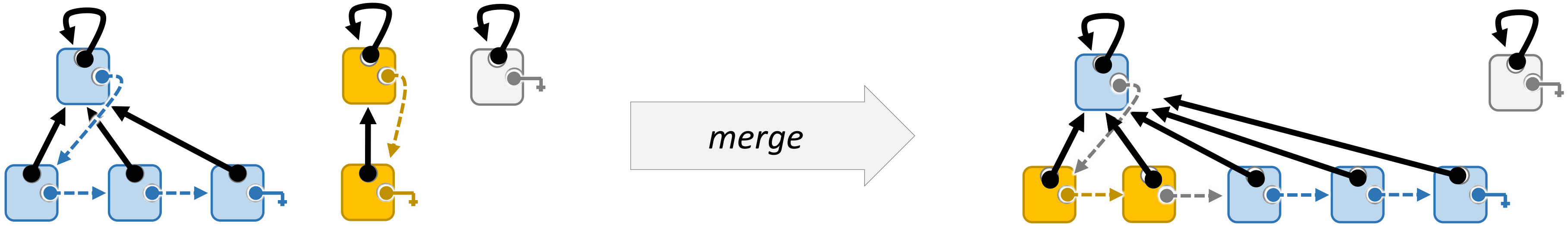
```
def get_set_with(x):
    return x's parent
```

O(1)

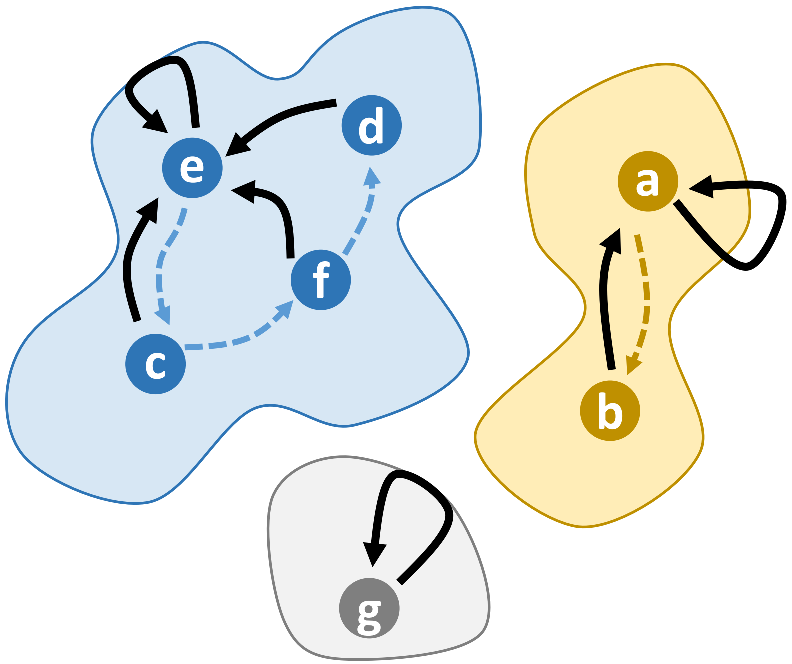
Speedup: let each repr. store the size of its set.

merge: we'll pick the smaller set to update.

(Doesn't change O(N) worst case, but is an improvement.)



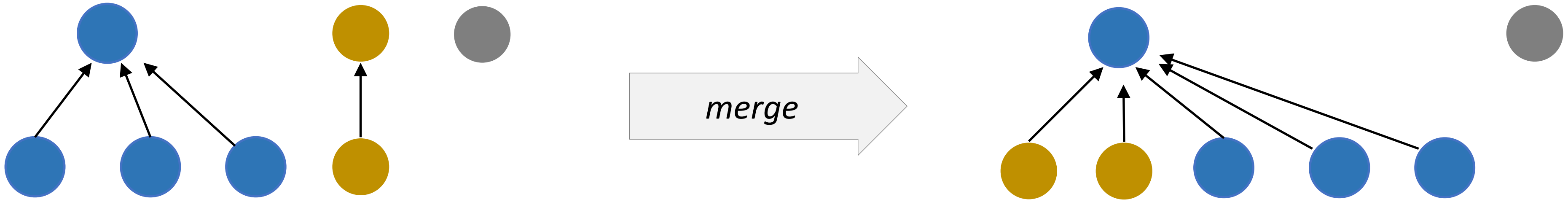
"Weighted union" heuristic.



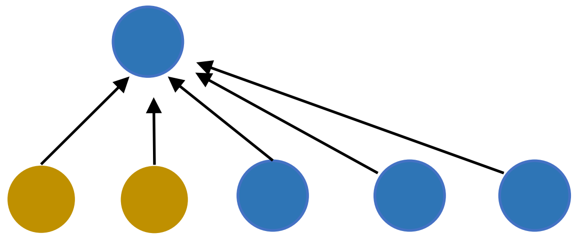
IMPLEMENTATION 1 "FLAT FOREST"

Each item points to a representative item for its set
Each set has a linked list, starting at its representative

```
def merge(x,y):
    for every item in set y:
        update it to belong to set x


def get_set_with(x):
    return x's parent
```



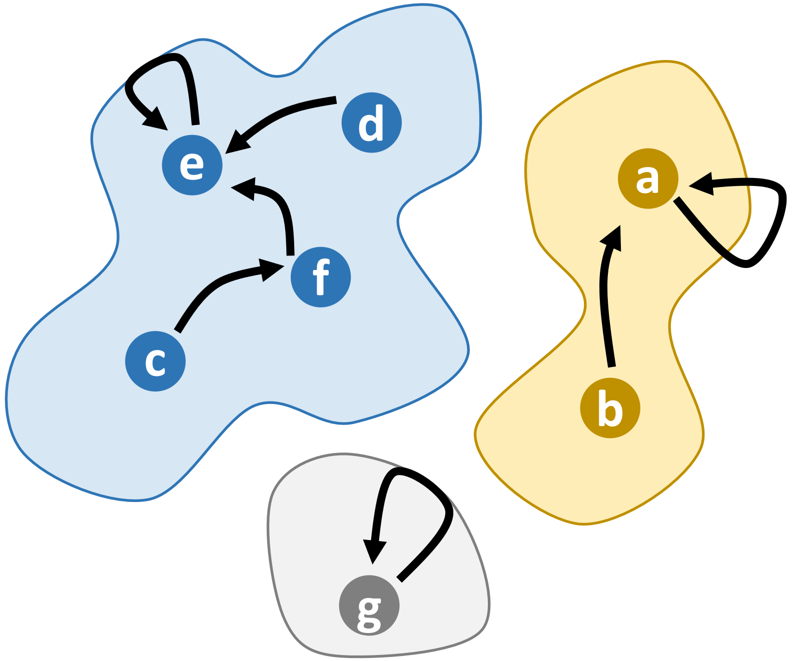
QUESTION. How can we design a DisjointSet so that `merge` is $O(1)$?



FLAT FOREST

`get_set_with` is $O(1)$

`merge` is $O(N)$



IMPLEMENTATION 2 "DEEP FOREST"

Sets are stored as trees
Use the root item to represent the set

```
def merge(x,y):
    update one of the roots to point to the other
```

$O(1)$

```
def get_set_with(x):
    walk up the tree from x to the root
    return this root
```

$O(N)$ in worst case

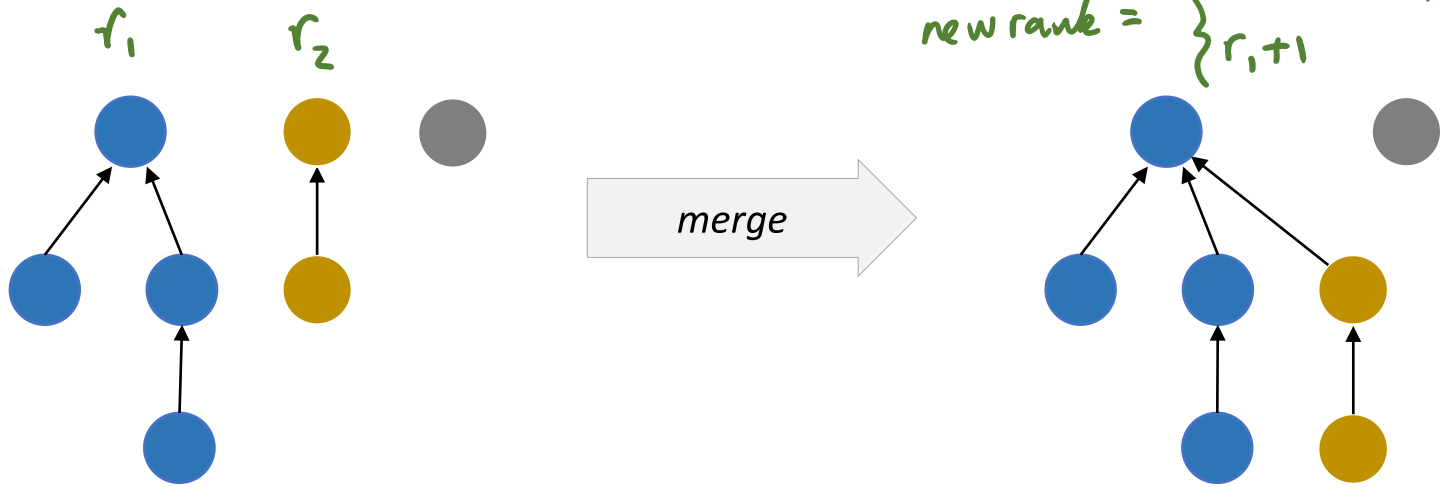
Speedup:

repr. stores the height of its tree (called "rank")

merge makes the low-rank tree go under the high-rank root.

"Union by rank" heuristic.

$$\text{new rank} = \begin{cases} \max(r_1, r_2) & \text{if } r_1 \neq r_2 \\ r_1 + 1 & \text{if } r_1 = r_2 \end{cases}$$

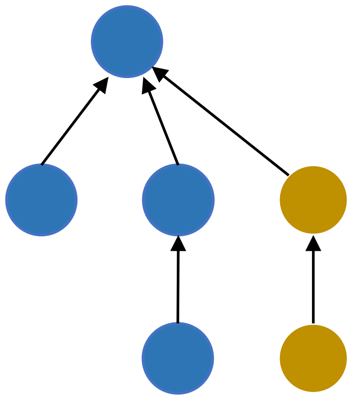


QUESTION. What's a sensible heuristic for `merge`, to speed up `get_set_with`?

quick and dirty
too harried to learn

no sweat
plans ahead

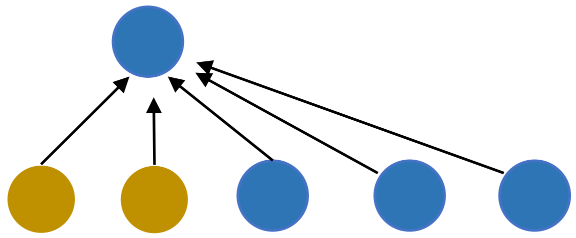
everything pristine
all of the time



DEEP FOREST

get_set_with is slower
merge is $O(1)$

QUESTION. **Can we have merge be $O(1)$, and also <u>manifest</u> our get_set_with working so that subsequent operations benefit?**



FLAT FOREST

get_set_with is $O(1)$
merge is $O(N)$

Can we 'manifest' our workings so that subsequent operations benefit?

SelectSort

Repeatedly scan for the largest remaining item, and move it to the sorted-chunk at the end.

A	B	C	D
key=3	key=1	key=5	key=4

1. Find the largest key, and put it at the end

- Start with largest-so-far = A
- Is B.key > A.key? No.
- Is C.key > A.key? Yes.
- Is D.key > C.key? No.
- Swap C and D

We had a useful piece of information, but we didn't keep it for the 2nd pass.

A	B	D	C
key=3	key=1	key=4	key=3

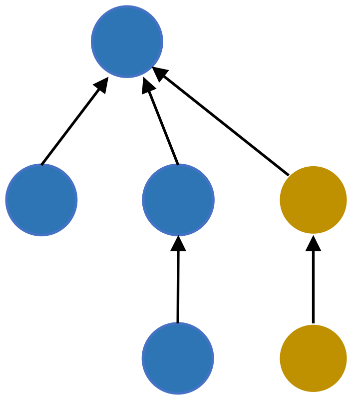
2. Find the largest out of [A,B,D]

The heap is a way to manifest what we've learnt so far, so we can re-use it in later passes. That's why HeapSort is better than SelectSort.

quick and dirty
too harried to learn

no sweat
plans ahead

everything pristine
all of the time

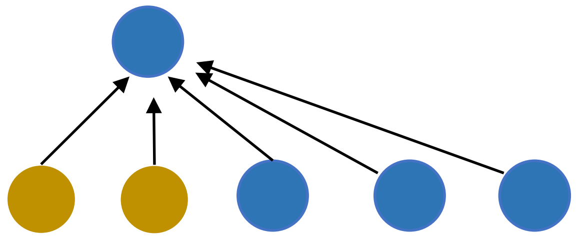


DEEP FOREST

`get_set_with` is slower

`merge` is $O(1)$

QUESTION. **Can we have `merge` be $O(1)$, and also <u>manifest</u> our `get_set_with` working so that subsequent operations benefit?**



FLAT FOREST

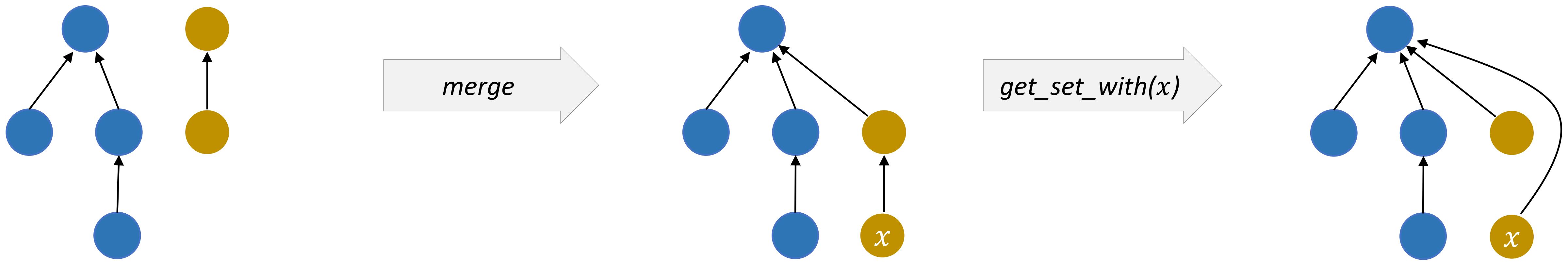
`get_set_with` is $O(1)$

`merge` is $O(N)$

IMPLEMENTATION 3 "LAZY FOREST"

```
def merge(x,y):
    as before, using the Union by Rank heuristic

def get_set_with(x):
    walk up the tree from x to the root
    walk up again, and make items in this path point to root
    return this root
```

"Path Compression heuristic".



Aggregate complexity analysis

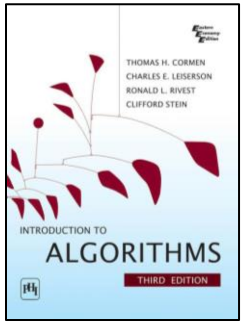
Any m operations on up to N items takes

Flat Forest
(with weighted-union)

$O(m + N \log N)$
[Ex. sheet 6 q. 13]

Deep Forest
(with union-by-rank)

$O(m \log N)$



Lazy Forest
(with union-by-rank + path compression)

$O(m\,\alpha(N))$

$$
\begin{aligned}
\alpha(N) &= 0 && \text{for } N = 0,1,2 \\
&= 1 && \text{for } N = 3 \\
&= 2 && \text{for } N = 4 \,..\, 7 \\
&= 3 && \text{for } N = 8 \,..\, 2047 \\
&= 4 && \text{for } N = 2048 \,..\, 10^{80}
\end{aligned}
$$

Flat Forest
(with weighted-union

Deep Forest
(with union-by-rank)

Lazy Forest
(with union-by-rank +

$\times \; O(m)$ each operation is $O(1)$ amortized.

Aggregate complexity analysis

Any m operations on up to N items takes

Flat Forest (with weighted-union)	$O(m + N\log N)$
Deep Forest (with union-by-rank)	$O(m\log N)$
Lazy Forest (with union-by-rank + path compression)	$O(m\,\alpha(N))$

$\alpha(N) = 0$ for $N = 0,1,2$

$= 1$ for $N = 3$

$= 2$ for $N = 4 .. 7$

$= 3$ for $N = 8 .. 2047$

$= 4$ for $N = 2048 .. 10^{80}$

1. take a handsome stoat
2. define a graph
   vertices on a grid, and edges between adjacent grid cells
3. assign edgeweights
   weight=low means vertices have similar colours
4. run Kruskal
   and find clusters of similar colour

Fastidious Frances | Crunchtime Charlie | Timely Terry