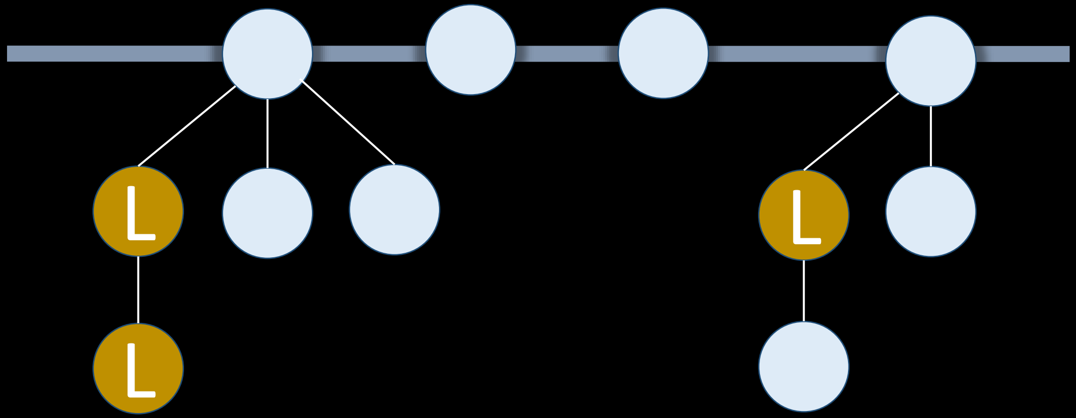
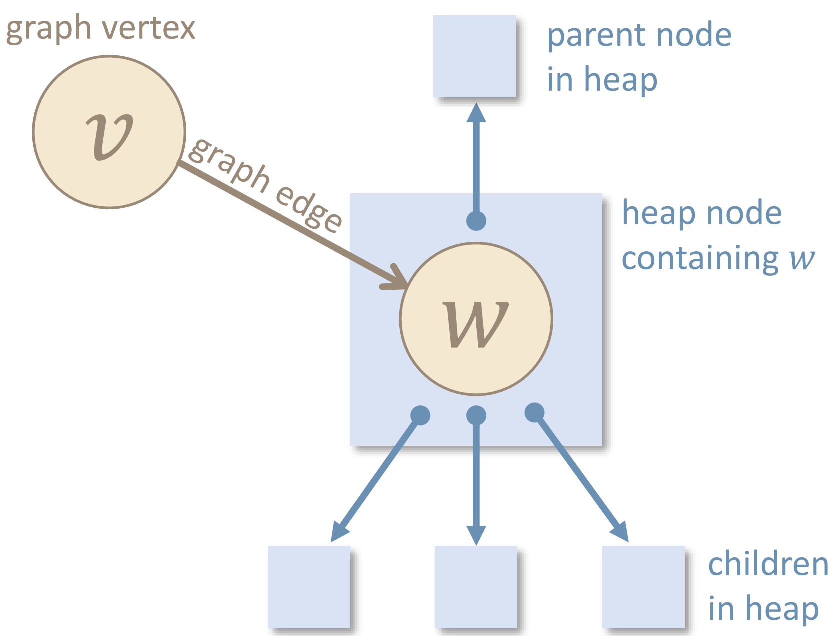
SECTION 7.6

The Fibonacci Heap



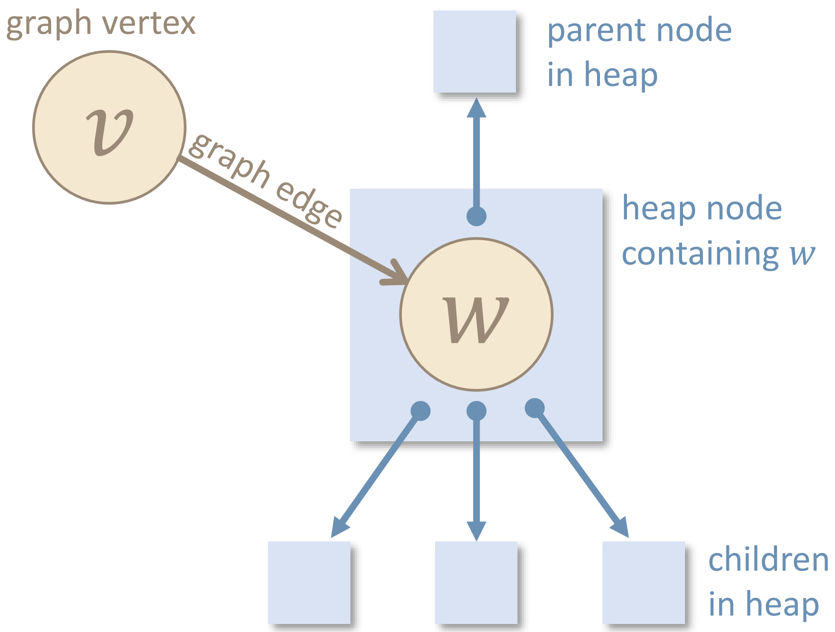
- push() — $O(1)$ amortized
  Lazy, just adds singleton nodes to the rootlist
- decreasekey() — $O(1)$ amortized
  Does some work to keep the trees in shape
  Adds singleton nodes to the rootlist
- popmin() — $O(\log N)$ amortized
  Cleans up the rootlist
  (at most one tree of any given degree)




```
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

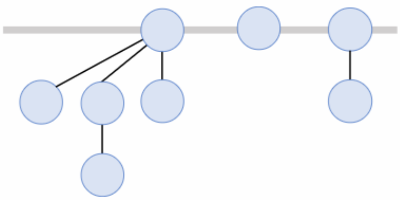
cl.cam.ac.uk/teaching/2223/Algorithm2/ticks/fib-heap.html

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.

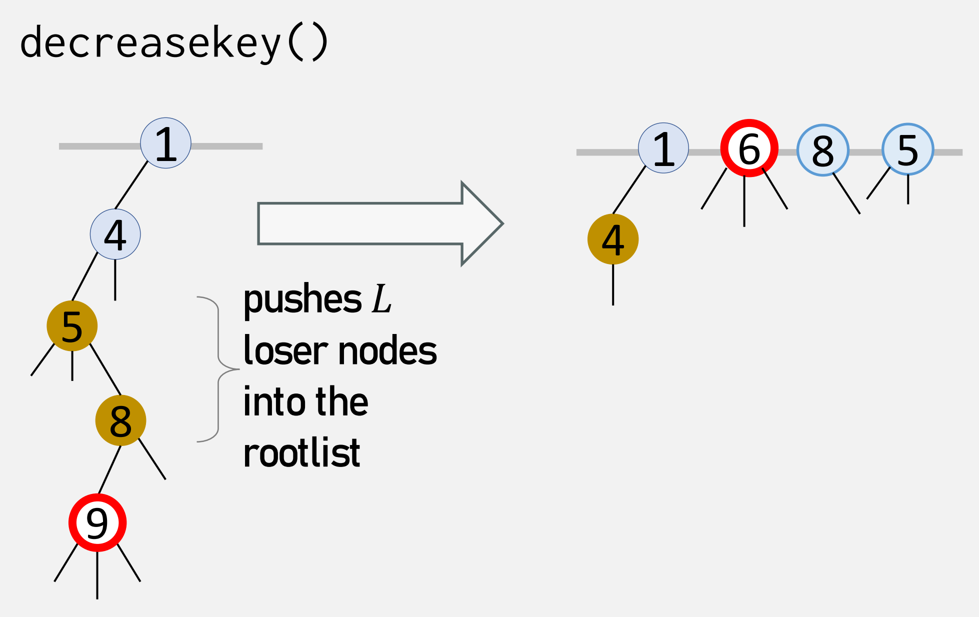
Step 1: heap operations

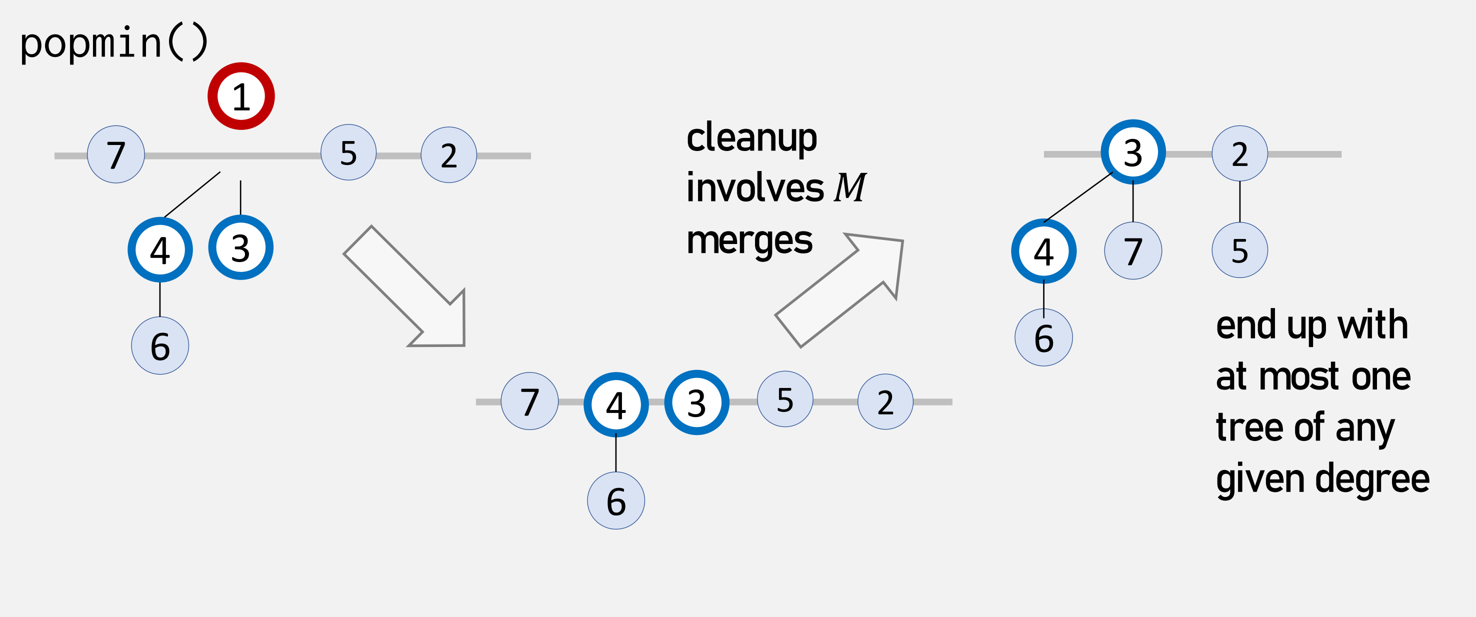


The first step is to implement a `FibNode` class to represent a node in the Fibonacci heap, and a `FibHeap` class to represent the entire heap. Each FibNode should store its priority key `k`, and the FibHeap should store a list of root nodes as well as the minroot.

SECTION 7.8

Amortized analysis of the Fibonacci Heap





decreasekey has true cost $O(L)$
so we want $\Delta\Phi = -L$ to pay for it

popmin merges trees in its cleanup phase, true cost $O(M)$
so we want $\Delta\Phi = -M$ to pay for it

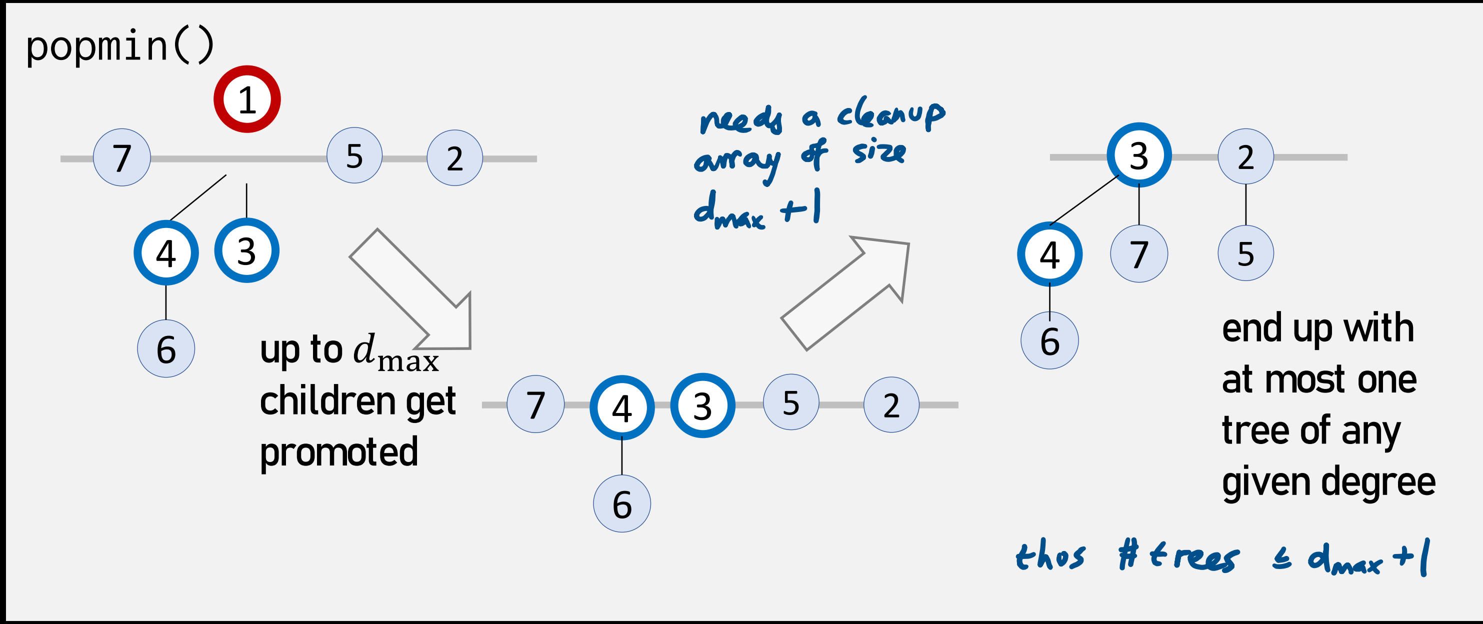
$\Phi = \text{num.roots} + 2 \times \text{num.losers}$ pays in advance for these "uncontrolled" iterations

SECTION 7.8

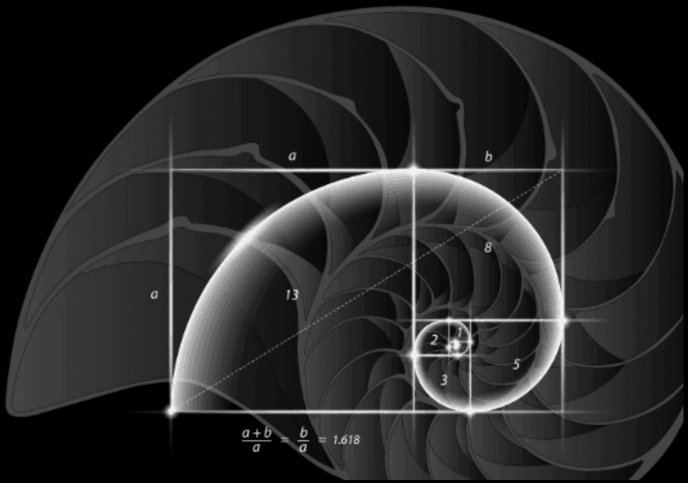
Amortized analysis of the Fibonacci Heap

SHAPE THEOREM

In a Fibonacci heap with N items, every node has degree $\leq \log_\phi N$ where ϕ is the golden ratio.



popmin also has to do $O(d_{\max})$ work where $d_{\max}$ is the maximum possible degree in a heap with N items

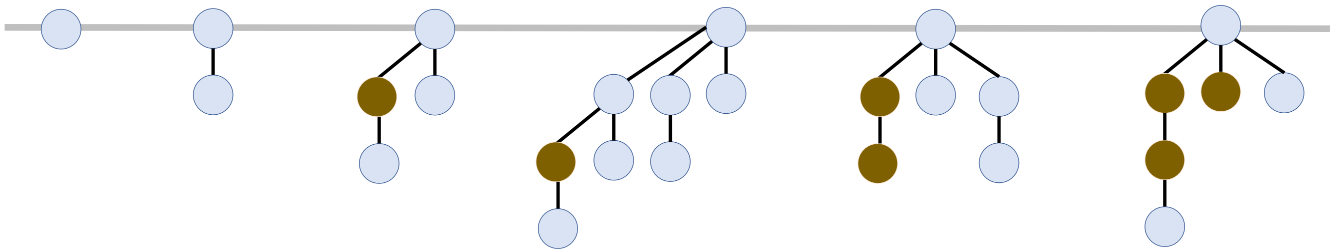


SHAPE THEOREM

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

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



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.

$N \geq$ num.nodes in subtree
$\geq F_{d+2}$ by lemma
$\geq \phi^d$ linear algebra:

Hence $d \leq \log_\phi N$. $F_n = \frac{\phi^n - (-\phi)^n}{\sqrt{5}}$.

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

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

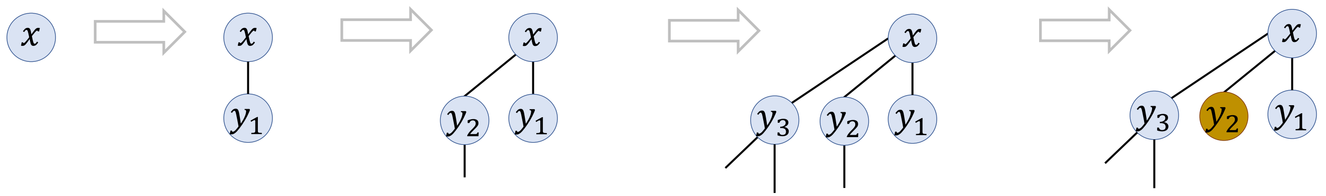
GRANDCHILD RULE

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

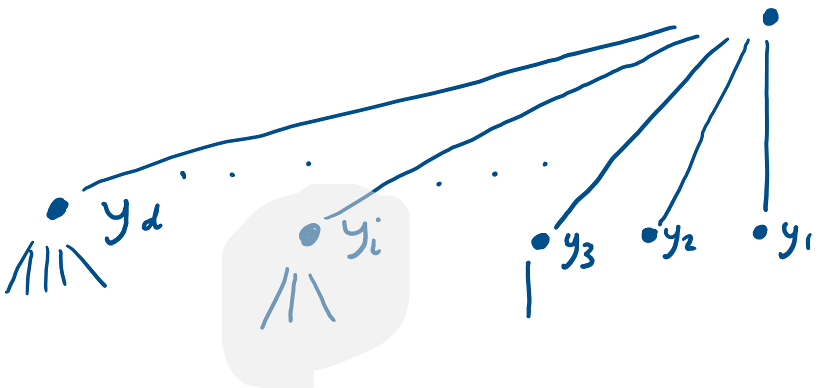
GRANDCHILD RULE

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 $i \in \{1, \ldots, d\}$

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

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 $\geq N_{d-2} + N_{d-3} + \cdots + N_1 + N_0 + N_0 + 1$

$N_d = N_{d-2} + N_{d-3} + \cdots + N_2 + N_0 + 1$

$N_{d-1} = N_{d-3} + \cdots + N_2 + N_0 + 1$

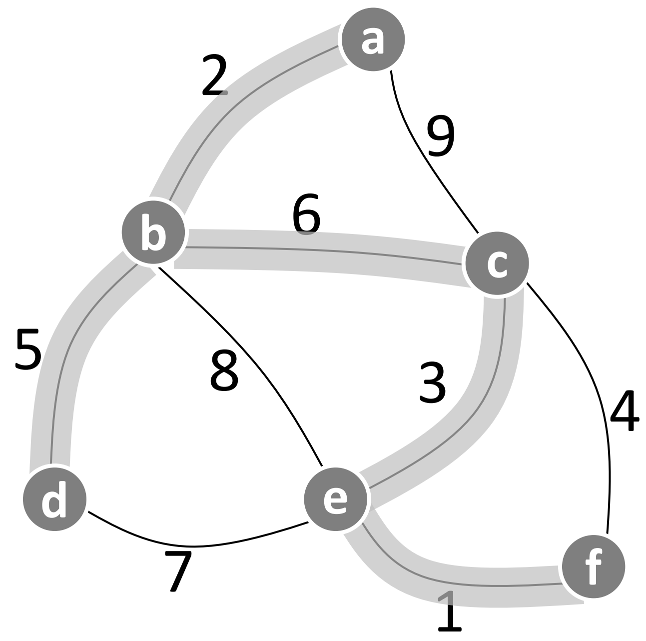
$\Rightarrow N_d = N_{d-2} + N_{d-1}$

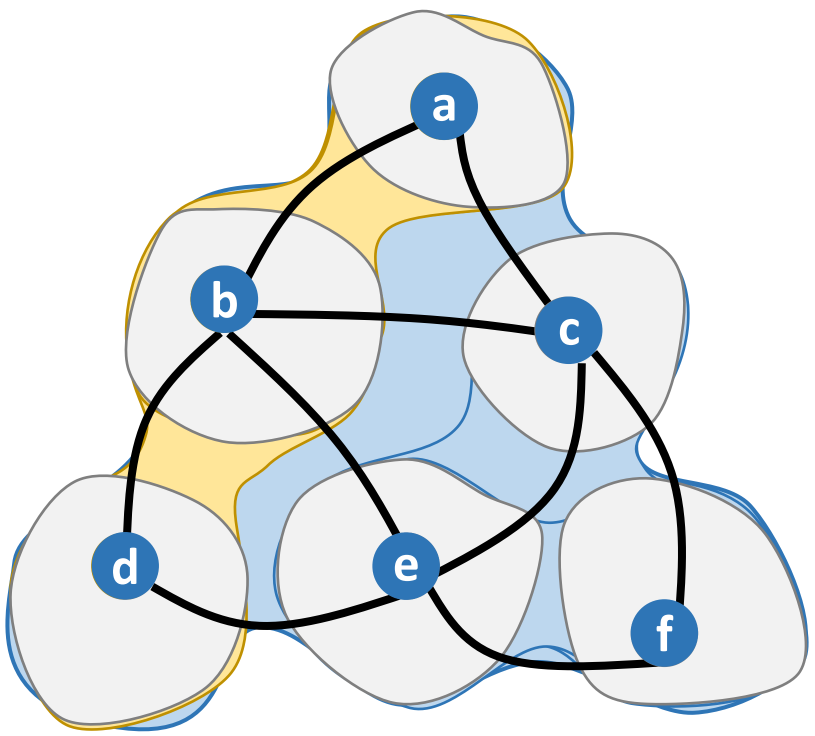
$\Rightarrow N_d$ is Fibonacci number

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

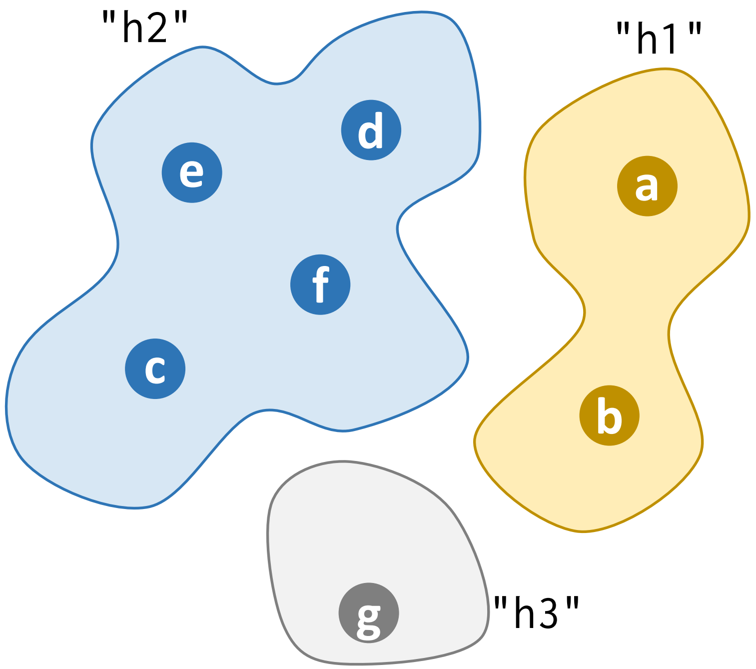
SECTION 7.9

Disjoint sets





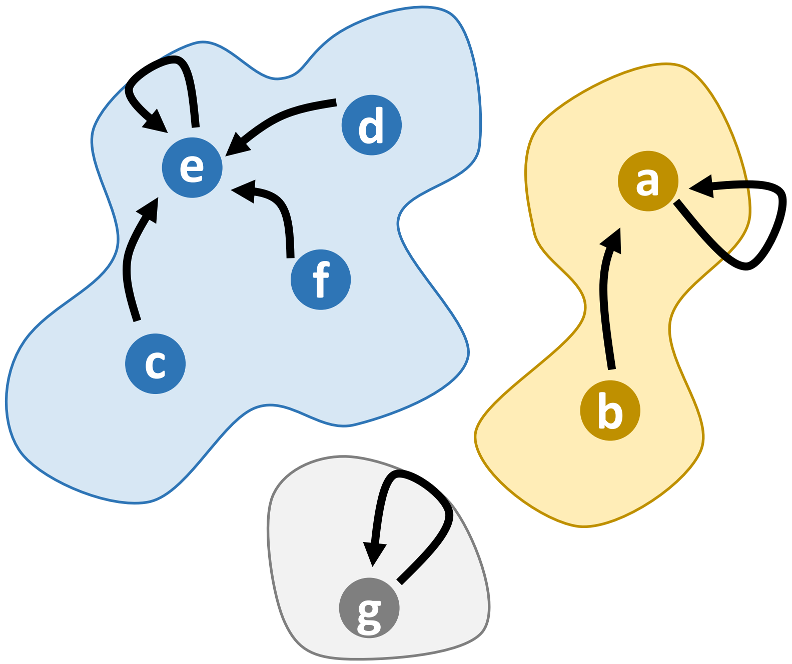

```
def kruskal(g):
    tree_edges = []
    partition = DisjointSet()
    for v in g.vertices:
        partition.add_singleton(v)
    edges = sorted(g.edges, sortkey = λ(u,v,weight): weight)

    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

```
mysets = {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 set is y then update it to be x
```

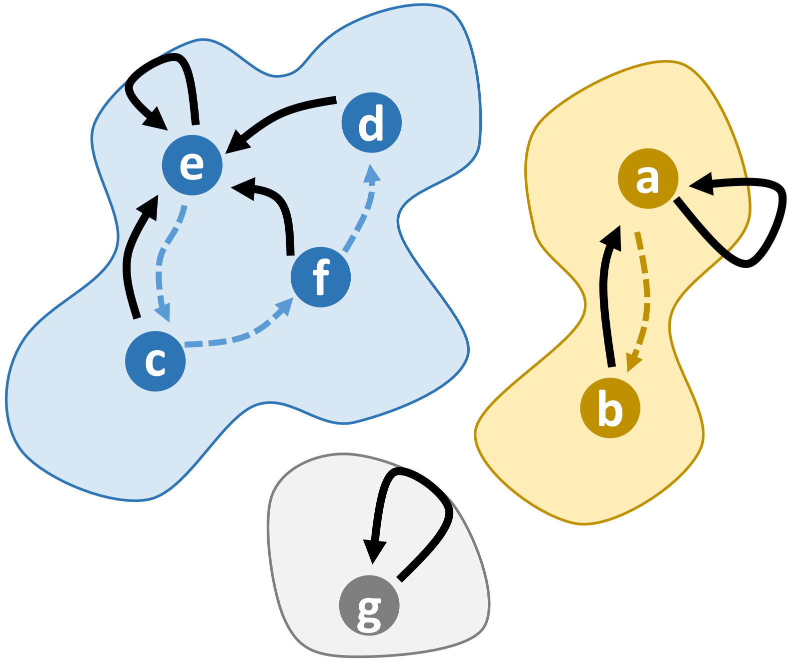
```
AbstractDataType DisjointSet:
    # Holds a dynamic collection of disjoint sets

    # 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

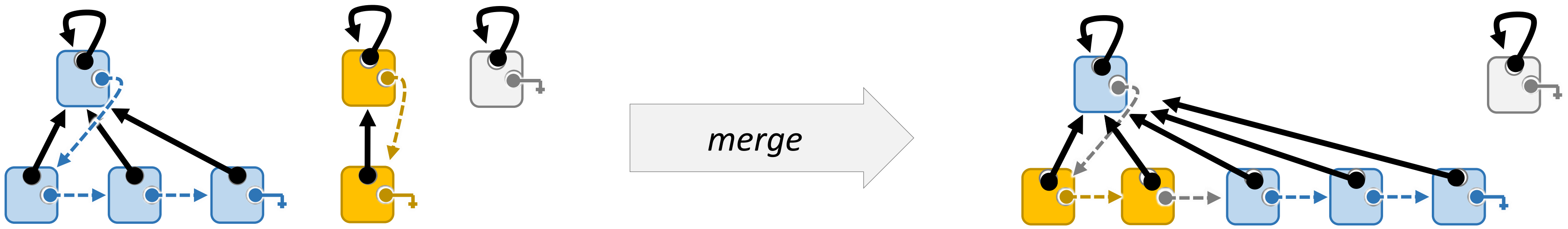
```
mysets = {a:a, b:a, c:e, d:e, e:e, f:e, g:g}
```

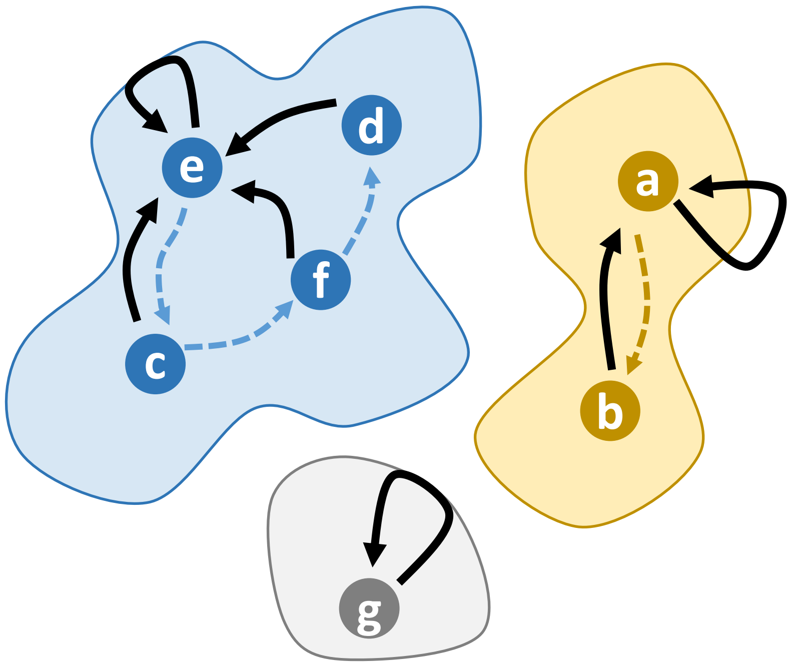


IMPLEMENTATION 1 “FLAT FOREST”

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

```
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
```

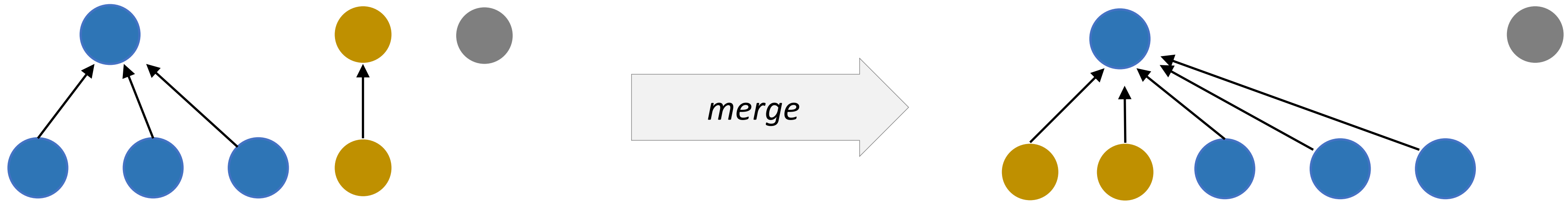


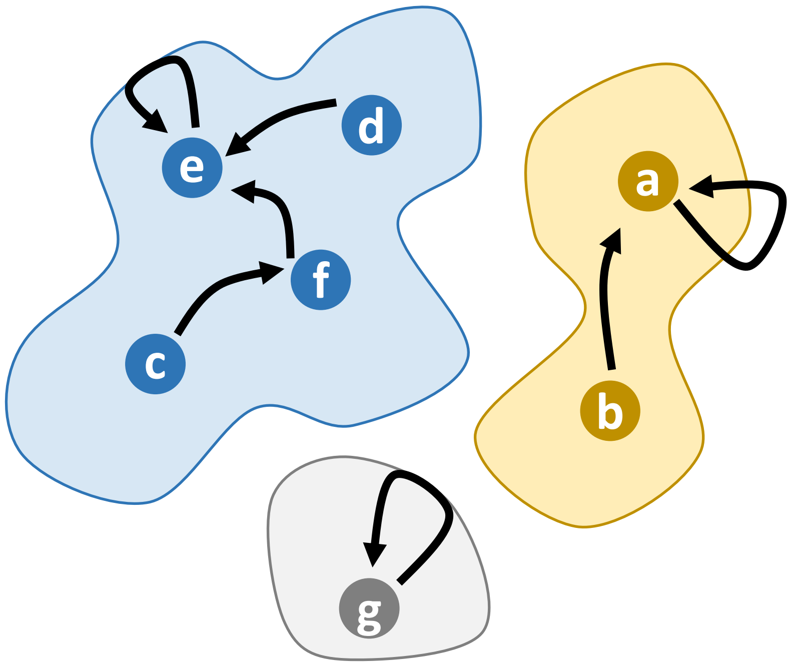


IMPLEMENTATION 1 "FLAT FOREST"

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
```

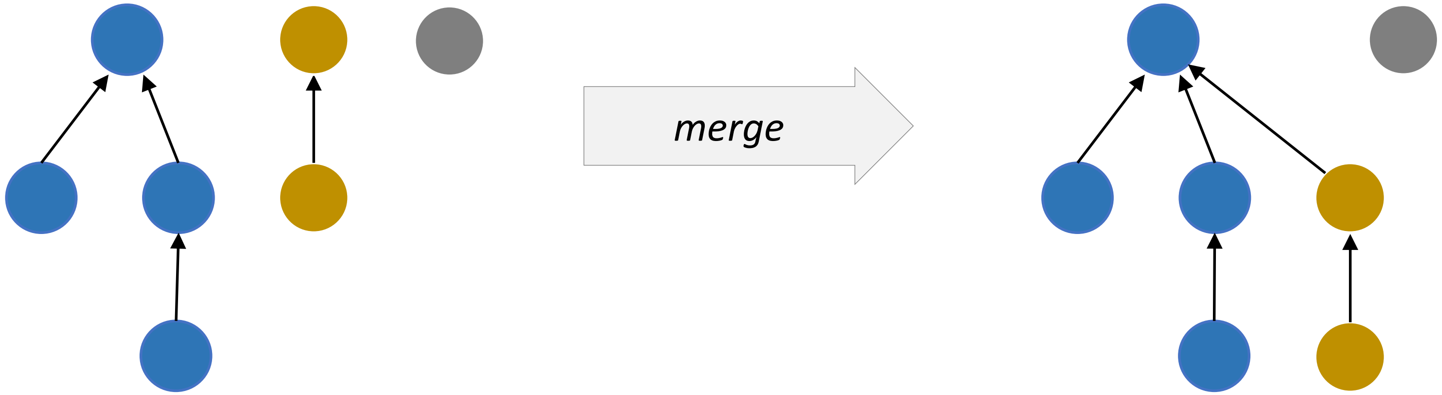




IMPLEMENTATION 2 “DEEP FOREST”

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

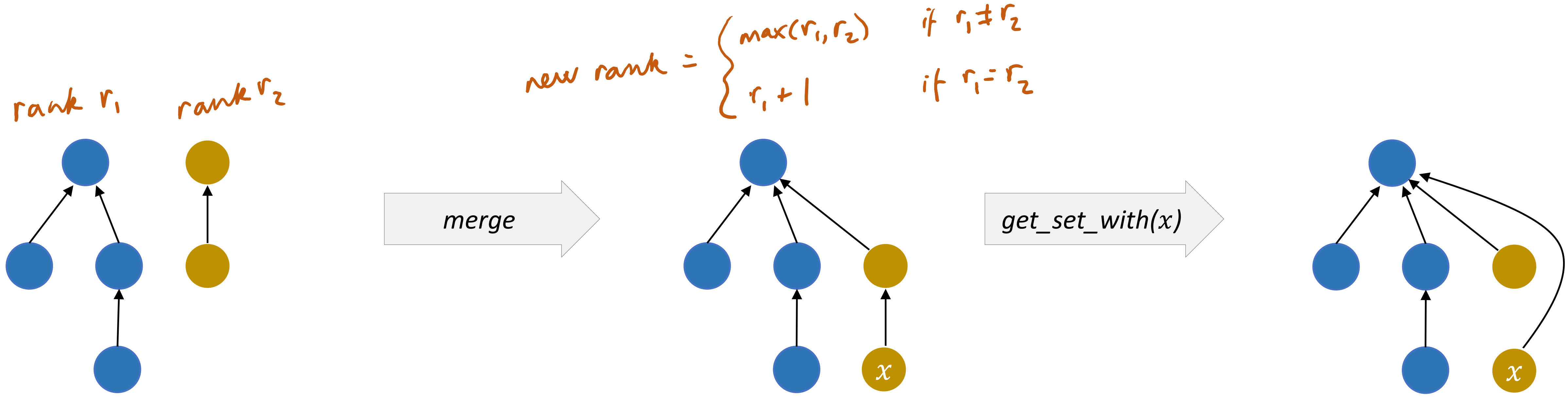
```
def merge(x,y):
    update one of the roots to point to the other

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



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

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 all items point to root
    return this root
```



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

```
def selectSort(a):
    """BEHAVIOUR: Run the selectsort algorithm on the integer
    array a, sorting it in place.

    PRECONDITION: array a contains len(a) integer values.

    POSTCONDITION: array a contains the same integer values as before,
    but now they are sorted in ascending order."""

    for k from 0 included to len(a) excluded:
        # ASSERT: the array positions before a[k] are already sorted

        # Find the smallest element in a[k:END] and swap it into a[k]
        iMin = k
        for j from iMin + 1 included to len(a) excluded:
            if a[j] < a[iMin]:
                iMin = j
        swap(a[k], a[iMin])
```

	A	B	C	D
$a =$	val=2	val=5	val=3	val=1

1. Find the lowest value, and put it at the front

- Is B.val < A.val? No.
- Is C.val < A.val? No.
- Is D.val < A.val? Yes.
- Swap A and D

	D	B	C	A
$a =$	val=1	val=5	val=3	val=2

2. Find the second-lowest in [B,C,A]

we had two useful pieces of information, but we didn't keep them :(

Aggregate complexity analysis

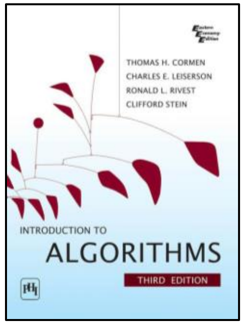
Any m operations on up to N items takes

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)

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

$\alpha(N) = 0$	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}

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 $N = 0,1,2$

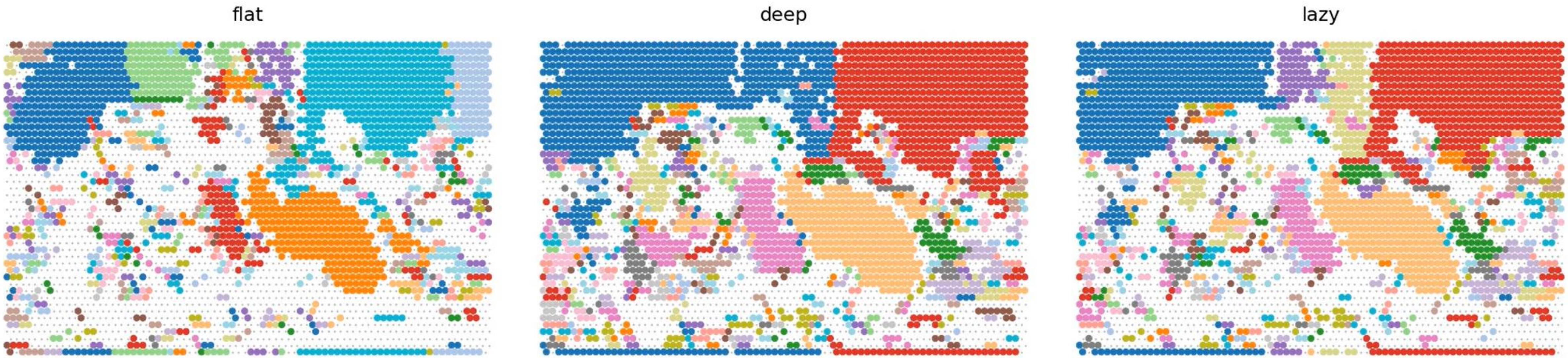
$= 1$ for $N = 3$

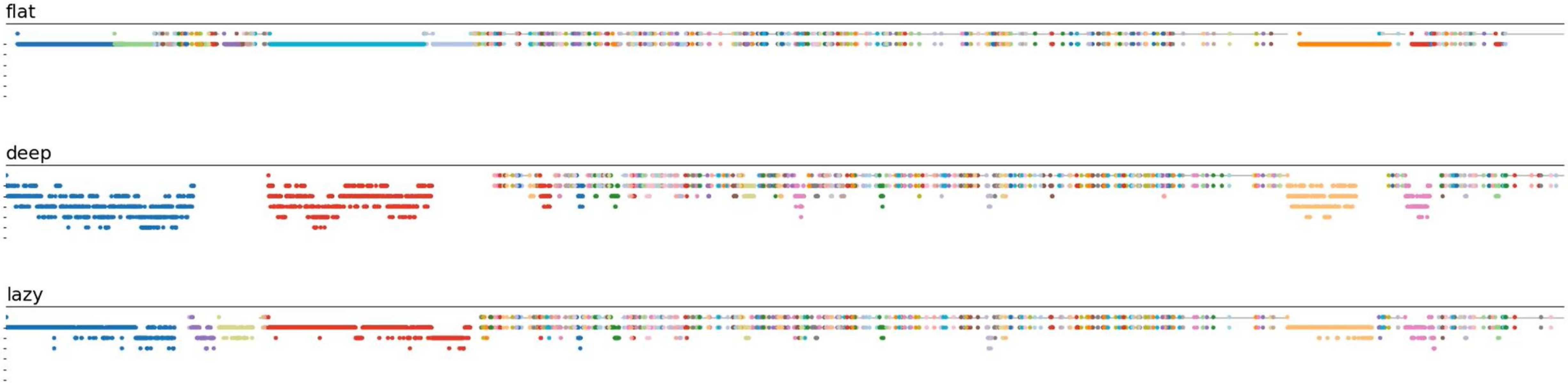
$= 2$ for $N = 4 .. 7$

$= 3$ 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


flat
deep
lazy


flat
deep
lazy