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
For good amortized costs, we want 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 $\leq \log_\phi N$

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.

![Diagram showing Fibonacci numbers and node counts for various root degrees.](image)
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.

\[ N \geq \text{num. nodes in subtree} \geq F_{d+2} \geq \phi^d \]

Hence \( d \leq \log_\phi N \).

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.

Recall: in a binomial heap...

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

In a binomial heap,

\[ N \geq \text{# nodes in largest tree} = 2^d \]

\[ d = \text{root degree} \Leftrightarrow \text{# nodes in largest tree} \geq \text{total # nodes} \]

\[ \max \text{ degree } \leq \log_2 N. \]
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.
def popmin():
    take note of minroot.value and minroot.key
    delete the minroot node, and promote its children to be roots
    # cleanup the roots
    while there are two roots with the same degree:
        merge those two roots, by making the larger root a child of the smaller
        update minroot to point to the root with the smallest key
    return the value and key we noted in line 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.

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 $x$.
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 $N_d = F_{d+2}$.

\[ N_d = N_{d-2} + N_{d-3} + \cdots + N_1 + N_0 + N_0 + 1 \]

\[ N_{d-1} = N_d - N_{d-2} \]

\[ N_d \text{ is Fibonacci number} \]
SECTION 7.7
Implementing the Fibonacci heap
QUESTION. How can \texttt{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 $(w, \text{edgecost})$ in $v$.neighbours:
            \text{dist}_w = v.distance + \text{edgecost}
            ...
            toexplore.decreasekey(w, key=\text{dist}_w)
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
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.
Crunch-time Charlie (quick and dirty, too harried to learn)

Timely Terry (no sweat, plans ahead)

Fastidious Frances (everything pristine all of the time)

**BINARY HEAP**
- push is slow, $O(\log N)$
- popmin is fast, $O(\log N)$

**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, $O(\log N)$
SECTION 7.9

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

**AbstractData**Type **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)
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

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

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

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

Implementations:

**IMPLEMENTATION 1 “FLAT FOREST”**

```python
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
```
quick and dirty
too harried to learn

everything pristine
all of the time

QUESTION. How can we design a DisjointSet so that merge is $O(1)$?
Sets are stored as trees
Use the root item to represent the set

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

**DEEP FOREST**
- $\text{get_set_with}$ is slower
- $\text{merge}$ is $O(1)$

**FLAT FOREST**
- $\text{get_set_with}$ is $O(1)$
- $\text{merge}$ 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.

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

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

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

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 $O(1)$

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

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

$O(m + N \log N)$

[Ex. sheet 6 q. 13]

Flat Forest
(with weighted-union)

Deep Forest
(with union-by-rank)

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

$O(m \log N)$

$O(m \alpha(N))$

$\alpha(N) = 0$ 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

Flat Forest
(with weighted-union)

Deep Forest
(with union-by-rank)

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

any $m$ operations on up to $N$ items takes $O(m + N \log N)$

$O(m \log N)$

$O(m \alpha(N))$

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

$\alpha(N) = 1$ for $N = 3$

$\alpha(N) = 2$ for $N = 4 \ldots 7$

$\alpha(N) = 3$ for $N = 8 \ldots 2047$

$\alpha(N) = 4$ for $N = 2048 \ldots 10^{11}$

WELL, THAT ESCALATED QUICKLY.
Aggregate complexity analysis

Any $m$ operations on up to $N$ items takes $O(m + N \log N)$

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 $N = 0, 1, 2$

$\alpha(N) = 1$ for $N = 3$

$\alpha(N) = 2$ for $N = 4 \ldots 7$

$\alpha(N) = 3$ for $N = 8 \ldots 2047$

$\alpha(N) = 4$ for $N = 2048 \ldots 10^{80}$
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