floordrobe, **noun.** A heap of clothing left on the floor of a room. **In computer science:** the most perfect design for an advanced data structure.
Pushing $N$ items is $O(N \log N)$ — but if we’re clever we can create a binary heap of $N$ items in $O(N)$.  

<table>
<thead>
<tr>
<th></th>
<th>popmin</th>
<th>push</th>
<th>decreasekey</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>binomial heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

$O(1)$ amortized

Dijkstra’s algorithm makes $O(E)$ calls to push / decreasekey, and only $O(V)$ calls to popmin.

**QUESTION1.** Can we make both push and decreasekey be $O(1)$?  

**QUESTION2.** What’s the binomial heap’s secret sauce that lets it have $O(1)$ push?
<table>
<thead>
<tr>
<th></th>
<th>Push</th>
<th>Popmin</th>
<th>Decreasekey</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>Binomial heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

- Pushing $N$ items is $O(N \log N)$ — but if we’re clever we can create a binary heap of $N$ items in $O(N)$.

- When we reheapify from depth $d$ it takes \(h - d\) work to bubble down, and there are \(\leq 2^d\) items that need this work.

- There are more items at greater depths, and it’s these items that take the least work.

- Total work is \(\sum_{d=0}^{h} 2^d (h - d)\)
  \[ \leq 2 \times 2^h = 2N \quad \text{[printed notes chapter 2.10]} \]
Pushing \( N \) items is \( O(N \log N) \) — but if we’re clever we can create a binary heap of \( N \) items in \( O(N) \).

<table>
<thead>
<tr>
<th></th>
<th>popmin</th>
<th>push</th>
<th>decreasekey</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary heap</td>
<td>( O(\log N) )</td>
<td>( O(\log N) )</td>
<td>( O(\log N) )</td>
</tr>
<tr>
<td>binomial heap</td>
<td>( O(\log N) )</td>
<td>( O(\log N) )</td>
<td>( O(\log N) )</td>
</tr>
</tbody>
</table>

\( O(1) \) amortized

SECRET SAUCE. Design your data structure so that most of the time it’s sufficient to only touch a small bit of it.

- The binary heap’s fast-heapification achieves this through doing its work in a batch (rather than push by push)
- The binomial heap achieves this by splitting up the heap into semi-isolatable trees
Pushing $N$ items is $O(N \log N)$ — but if we’re clever we can create a binary heap of $N$ items in $O(N)$.

<table>
<thead>
<tr>
<th></th>
<th>popmin</th>
<th>push</th>
<th>decreasekey</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>binomial heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
</tbody>
</table>

$O(1)$ amortized

Dijkstra’s algorithm makes $O(E)$ calls to push / decreasekey, and only $O(V)$ calls to popmin.

**QUESTION 1.** Can we make both push and decreasekey be $O(1)$?

**QUESTION 2.** What’s the binomial heap’s secret sauce that lets it have $O(1)$ push?
push is $O(1)$

push(new item)

push(new item)
Linked-list priority queue

decreasekey is $O(1)$

decreasekey(item, new key)

first

minitem

3 12 3 7 9 1 6 5 1

3 0 3 7 9 1 6 5 1

3 0 3 7 9 1 6 5 1
Linked-list priority queue

\[ \text{popmin is } O(N) \]

\[ \text{popmin()} \]
<table>
<thead>
<tr>
<th></th>
<th>popmin</th>
<th>push</th>
<th>decreasekey</th>
</tr>
</thead>
<tbody>
<tr>
<td>binary heap</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>binomial heap</td>
<td>$O(\log N)$</td>
<td>$O(1)$</td>
<td>$O(\log N)$</td>
</tr>
<tr>
<td>linked list</td>
<td>$O(N)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>$O(\log N)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Design strategy for the Fibonacci heap:

❖ Give your data enough structure that you only need to touch a little bit of it

❖ Be lazy: let mess accumulate

❖ Do cleanup in batches

batch-push is $O(N)$
SECTION 7.6
The Fibonacci Heap
- store a list of trees, each a heap
- trees can have any shape
- keep track of the minroot

```python
# Maintain a list of heaps (i.e. store a pointer to the root of each heap)
roots = []

# Maintain a pointer to the smallest root
minroot = None

def push(Value v, Key k):
    # create a new heap h consisting of a single item (v, k)
    h = [v, k]
    # add h to the list of roots
    roots.append(h)
    # update minroot if minroot is None or k < minroot.key
    if minroot is None or k < minroot[1]:
        minroot = h
```

push(new item)
```python
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     
16     # cleanup the roots
17     while there are two roots with the same degree:
18         merge those two roots, by making the larger root a child of the smaller
19         update minroot to point to the root with the smallest key
20     return the value and key we noted in line 13
```

**popmin()**

1. **extract min root**

   - **7**
   - **4**
   - **6**
   - **3**
   - **2**
   - **5**

2. **merge eq. roots**

   - **7**
   - **4**
   - **3**
   - **6**
   - **5**
   - **2**

3. **set M**

   - **3**
   - **4**
   - **7**
   - **5**
   - **6**

4. **merge eq. roots**

   - **3**
   - **4**
   - **7**
   - **5**
   - **6**

5. **merge eq. roots**

   - **3**
   - **5**
   - **2**
   - **6**
decreasekey(item, new key)

LAZY STRATEGY
Dump heap-violating nodes into the root list, to be cleaned up by the next popmin()

... but we might end up with a heap with wide shallow trees, which will make popmin() slow
Rule 1. Lose one child, and you’re marked a **LOSER**

Rule 2. Lose two children, and you’re dumped into the root list
# Every node will store a flag, $n$.loser = True / False

```python
def decreasekey($v$, $k'$):
    let $n$ be the node where this value is stored
    $n$.key = $k'$
    if $n$ violates the heap condition:
        repeat:
            $p$ = $n$.parent
            remove $n$ from $p$.children
            insert $n$ into the list of roots, updating minroot if necessary
            $n$.loser = False
            $n$ = $p$
        until $p$.loser == False
    if $p$ is not a root:
        $p$.loser = True
```

# Modify popmin so that when we promote minroot's children, we erase any loser flags
Sometimes it pays to let mess build up

Your parents want lots of grandchildren*

* and they’ll disown you if you don’t have enough
SECTION 7.8
Amortized analysis of the Fibonacci Heap

Take-away: this is an elegant use of potential functions to account for two separate unbounded-cost operations.
SHAPE THEOREM
Every node has degree \( \leq \log_\phi N \)

\( \phi \) is golden ratio \( \approx 1.618 \)

FIBONACCI HEAP
COMPLEXITY ANALYSIS

COMPLEXITY ANALYSIS
In a Fibonacci heap with \( N \) items, using the potential function
\[
\Phi = \text{num.roots} + 2 \times \text{num.losers},
\]
- push() has amortized cost \( O(1) \)
- decreasekey() has amortized cost \( O(1) \)
- popmin() has amortized cost \( O(\log N) \)

SHAPE THEOREM
The largest tree has degree \( \leq \log_2 N \)

BINOMIAL HEAP
COMPLEXITY ANALYSIS

In a binomial heap with \( N \) items
- push() is \( O(\log N) \)
- decreasekey() is \( O(\log N) \)
- popmin() is \( O(\log N) \)
\[ \Phi = \text{num.roots} + 2 \times \text{num.losers} \]

```python
def push(Value ν, Key k):
    create a new heap \( h \) consisting of a single item \((ν,k)\)
    add \( h \) to the list of roots
    update minroot if minroot is None or \( k < \text{minroot.key} \)
```

\[
c = O(1) \quad \Delta \Phi = 1 \quad \text{am. cost}
\]

\[
c' = c + \Delta \Phi = O(1)
\]
\[ \Phi = \text{num.roots} + 2 \times \text{num.losers} \]

\begin{verbatim}
32 def decreasekey(v, k):
33     let n be the node where this value is stored
34     n.key = k'
35     if n violates the heap condition:
36         repeat:
37             p = n.parent
38             remove n from p.children
39             insert n into the list of roots, updating minroot if necessary
40             n.loser = False
41             n = p
42         until p.loser == False
43     if p is not a root:
44         p.loser = True

Case I: no heap violation. \( c = O(1) \) \( \Delta \Phi = 0 \) \( c + \Delta \Phi = O(1) \).

Case II: heap violation.
1. move a to root list. \( c = O(1) \) \( \Delta \Phi = 1 \) or \( \Delta \Phi = -1 \) if a was loser.
2. move up to losers \( c = O(L) \) \( \Delta \Phi = +L - 2L = -L. \) \( c + \Delta \Phi = O(1) \)
3. move d as loser. \( c = O(1) \) \( \Delta \Phi = 2 \) or \( \Delta \Phi = 0 \) if d was a root \( c + \Delta \Phi = O(1) \)
\end{verbatim}
\[ \Phi = \text{num.roots} + 2 \times \text{num.losers} \]

\[ \text{degree} \leq \log_\Phi N \]

```python
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
```

1. cut out minroot, promote its children

2. cleanup. we'll see \[ c + \Delta \Phi = O(\log N) \]

3. fix minroot. \[ c = O(\log N) \quad \Delta \Phi = 0 \quad c + \Delta \Phi = O(\log N) \]

\[ \Phi = \text{num.roots} + 2 \times \text{num.losers} \]

\[ \text{degree} \leq \log_\Phi N \]

at most one tree of any degree.

\[ \Phi \leq 1 + \log N \]

\[ \text{largest possible root-degree is } \lceil \frac{\log N}{\log_\Phi} \rceil \]

The total for these three steps is \( O(\log N) \) amortized cost.

num.roots dec. by 1

num.roots inc. by # children

num losers decreases, maybe
```
Φ = num.roots + 2 × num.losers

\[ \text{degree} \leq \log_\Phi N \]

\[
\Phi = \text{num.roots} + 2 \times \text{num.losers}
\]

```

```
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
```

```
def cleanup(roots):
    root_array = [None, None, ....]
    for each tree t in roots:
        x = t
        while root_array[x.degree] is not None:
            u = root_array[x.degree]
            root_array[x.degree] = None
            x = merge(x, u)
            root_array[x.degree] = u
    roots = list of non-None values from root_array
```

\[ \Phi = \text{num.roots} + 2 \times \text{num.losers} \]

for each \( t \) in roots:

\[
\text{updated roots:}
\]

Suppose I started with \( x \) trees, do \( M \) merges, end up with \( y \) trees, so \( y = x - M \Rightarrow x = y + M \).

\[
c = O(x + M + \log N) = O(y + 2M + \log N) = O(2M + 2\log N) = O(M + \log N),
\]

\[ \Delta \Phi = -M \text{ since \# roots has decreased by } M \]

Thus \( c + \Delta \Phi = O(M + \log N) - M = O(\log N) \).

```python
def cleanup(roots):
    root_array = [None, None, ... ]
    for each tree \( t \) in roots:
        \( x = t \)
        while root_array[x.degree] is not None:
            \( u = \text{root_array}[x.degree] \)
            root_array[x.degree] = None
            \( x = \text{merge}(x, u) \)
            root_array[x.degree] = u
    roots = list of non-None values from root_array
```
\[ \Phi = \text{num.roots} + 2 \times \text{num.losers} \]

**page 73**

**Algorithm**

**for each** \( t \) in roots:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- \( root_array \)
- \( \Delta \Phi = -M \)

**def** clean(root):

- \( root_array = [None, None, ...] \)
- \( \text{empty array of length 2} \)

**def** decreasekey(v, k):

- \( p = \text{parent} \)
- \( \text{move } k \text{ to parent} \)
- \( \text{insert } k \text{ into list of roots, updating minroot if necessary} \)

- **cases**:
  - **no heap violation**:
    - \( \Delta \Phi = 0 \)
  - **heap violation**:
    1. move up \( L \) times
    2. move \( L \) times
    3. move \( L \) times

**popmin**

- had to do \( M \) merges

**decreasekey**

- had to move \( L \) nodes to root