SECTION 7.5
Three priority queues
AbstractDataType PriorityQueue

# Holds a dynamic collection of items
# Each item has a value v, and a key/priority k

# Extract the item with the smallest key
Pair<Key, Value> popmin()

# Add v to the queue, and give it key k
push(Value v, Key k)

# For a value already in the queue, give it a new (lower) key
decreasekey(Value v, Key k')

# Sometimes we also include methods for
Pair<Key, Value> peekmin()
delete(Value v)
merge_with(PriorityQueue q)
The binary heap

The heap property
every node’s key is $\leq$ those of its children
The binary heap

```
0
1
5
6
1
6
9
12
7
3
```

**popmin()**

1. extract root
2. replace root
3. bubble down
4. bubble down

```
popmin()
```

```
extract root
```

```
replace root
```

```
bubble down
```

```
bubble down
```

```
1
```

```
5
```

```
3
```

```
1
```

```
5
```

```
3
```

```
6
```

```
1
```

```
6
```

```
9
```

```
6
```

```
9
```

```
3
```

```
1
```

```
5
```

```
6
```

```
9
```
The binary heap

push(new item)

append

bubble up

bubble up

bubble up

bubble up
The binary heap

push(new item)

bubble up

append

decreasekey(item, new key)
SHAPE LEMMA
The height is $O(\log N)$
where $N$ is the number of items in the heap

COMPLEXITY ANALYSIS
All operations are $O(\log N)$,
Binomial trees

2 a tree of degree 0

2 5
two trees of degree 0
merge to give a tree of degree 1

2 6 9
2 5 9
two trees of degree 1
merge to give a tree of degree 2

2 6 5 9
2 3 7 12
two trees of degree 2
merge to give a tree of degree 3
The binomial heap

- a list of binomial trees, at most one of each degree
- each tree is a heap

push(*new item*)

append

merge trees of equal degree
The binomial heap

decreasekey \((item, new\ key)\)

bubble up
The binomial heap

popmin()

extract min root

merge eq. trees

promote children
The binomial heap

SHAPE THEOREM
- A binomial tree of degree $k$ has $2^k$ items
- In a binomial heap with $N$ items, the binary digits of $N$ tell us which binomial trees are present

Also, in a binomial tree of degree $k$,
- the root has degree $k$
- its $k$ children are binomial trees
- the height is $k$

COMPLEXITY ANALYSIS
- $\text{push()}$ is $O(\log N)$
  we have to merge $O(\log N)$ trees
- $\text{decreasekey()}$ is $O(\log N)$
  in the worst case we have to bubble up from the bottom of the largest tree
- $\text{popmin()}$ is $O(\log N)$
  scan $O(\log N)$ trees; promote $O(\log N)$ children; do $O(\log N)$ merges to recover the heap
<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>

*And what about aggregate costs?*
<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>

And what about aggregate costs?
<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>

And what about aggregate costs?
<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>

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

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

[Ex. sheet 6 q. 2, 4]
Linked-list priority queue

push is $O(1)$

push(new item)
Linked-list priority queue

**decreasekey** is $O(1)$

decreasekey(*item*, *new key*)
Linked-list priority queue

popmin is $O(N)$

popmin()
Be lazy

Do cleanup in batches

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

---

<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)) amort</td>
<td>(O(\log N))</td>
</tr>
<tr>
<td>linked list</td>
<td>(O(N))</td>
<td>(O(1)) amort</td>
<td>(O(1)) amort</td>
</tr>
<tr>
<td>Fibonacci heap</td>
<td>(O(\log N)) amort</td>
<td>(O(1)) amort</td>
<td>(O(1)) amort</td>
</tr>
</tbody>
</table>

*but \(N\) pushes are only \(O(N)\)* (see heapsort, §2.10)
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)
    add h to the list of roots
    update minroot if minroot is None or k < minroot.key
```
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
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
decreasekey(item, new key)

**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

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
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
Every node has degree \( \leq \log_\phi N \)

BINOMIAL HEAP
COMPLEXITY ANALYSIS

COMPLEXITY ANALYSIS
In a binomial heap with \( N \) items
- push() is \( O(\log N) \)
- decreasekey() is \( O(\log N) \)
- popmin() is \( O(\log N) \)

SHAPE THEOREM
The largest tree has degree \( \leq \log_2 N \)
\[ \Phi = \text{num.roots} + 2 \times \text{num.losers} \]

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

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

```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
            \( n.\text{loser} = \text{False} \)
            \( n = p \)
        until \( p.\text{loser} == \text{False} \)
    if \( p \) is not a root:
        \( p.\text{loser} = \text{True} \)
```

**CASE I: no heap violation**

\[ c = O(1) \quad \Delta \Phi = 0 \quad \Rightarrow \quad c + \Delta \Phi = O(1) \]

**CASE II: heap violation**

1. Move \( a \) to root list
   \[ c = O(1) \quad \Delta \Phi = 1 \quad \text{or} \quad \Delta \Phi = -1 \quad \text{if} \quad a \text{ was loser} \]
   \[ \Rightarrow \quad c + \Delta \Phi = O(1) \]

2. Move up \( L \) losers also
   \[ c = O(L) \quad \Delta \Phi = +L - 2L = -L \]
   \[ \Rightarrow \quad c + \Delta \Phi = O(1) \]

3. Mark \( d \) as a loser
   \[ c = O(1) \quad \Delta \Phi = 2 \quad \text{unless} \quad d \text{ is root}, \Delta \Phi = 0 \]
   \[ \Rightarrow \quad c + \Delta \Phi = O(1) \]

in both cases, total amortized cost is \( O(1) \)
\[ \Phi = \text{num.roots} + 2 \times \text{num.losers} \]

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

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: \( c = O(\# \text{ children}) \) \( \Delta \Phi \leq -1 + \# \text{ children} \) \( \Rightarrow c + \Delta \Phi = O(\log N) \)

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

3. fix minroot, by scanning the cleaned-up root list:
   there's at most one tree of each degree; max degree = \( O(\log N) \) \( \Rightarrow c = O(\log N) \)

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
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
            \( x = \text{merge}(x, u) \)
            root_array[\( x \).degree] = \( u \)
    roots = list of non-None values from root_array

\( \Phi = \text{num.roots} + 2 \times \text{num.losers} \)

degree \leq \log_\Phi N
\[ \Phi = \text{num.roots} + 2 \times \text{num.losers} \]

At the end of cleanup, we want to have \( \leq 1 \) tree of any given degree.

**SHAPE THEOREM**

Every node has degree \( \leq \log_\Phi N \)

To fit these trees, we’ll need an array of size \( \leq \log_\Phi N + 1 \)

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

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

$\text{degree} \leq \log_\Phi N$

updated roots:

$\begin{array}{c}
0 & 1 & 2 & 3 \\
\hline
\text{root_array} & & & \\
\end{array}$

Suppose we start with $x$ trees, do $M$ merges, and end up with $y$ trees.

$c = O(x + M + \log N) = O(y + 2M + \log N) = O(2M + 2\log N) = O(M + \log N)$

$\Delta \Phi = -M$

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

$\text{degree} \leq \log_\Phi N$

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

$\text{degree} \leq \log_\Phi N$

$\text{def cleanup(roots):}$

$\text{root_array} = [\text{None, None, ...}]$

for each tree $t$ in roots:

$x = t$

while root_array[x.degree] is not None:

$u = \text{root_array}[x.d}$

root_array[x.degree] = None

$x = \text{merge}(x, u)$

root_array[x.degree] = $u$

roots = list of non-$\text{None}$ values from root_array

$\text{am. case is}$

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

$\cdot O(\log N)$
\[ \Phi = \text{num.roots} + 2 \times \text{num.losers} \]

In both cases, total amortized cost is \( \mathcal{O}(1) \)

def decreasekey(v, k):
    let n be the node where this value is stored
    if n violates the heap condition:
        parent = n.parent
        remove n from p.children
        insert n into the list of roots, updating external if necessary
        if p.losser = False:
            p.losser = True
            p = p.parent
        while p is not a root
            p.losser = True
    c = 0
    \( \Delta\Phi = c + \Delta\Phi \cdot c \)

def cleanups(roots):
    root_array = [None, None, ...]
    for each tree t in roots:
        n = t
        while root_array[degree(n)] is not None:
            n = root_array[degree(n)]
        root_array[degree(n)] = n
        if n.losser:
            c = 0
            \( \Delta\Phi = c + \Delta\Phi \cdot c \)

pays in advance for these “uncontrolled” iterations

def popmin:
    M merges

def decreasekey:
    had to move
    L nodes to root
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

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.