Exercise  (Lower-bounding heap operations)

Theorem  Suppose only comparisons are allowed on heap keys.
Then for all heap implementations,

    Insert or \((\text{Extract and (Delete or Minimum)})\)

must take \(\Omega(\log n)\) amortized time on a heap of \(n\) elements.

Proof  Suppose not, i.e. that there is a comparison-based heap
for which

    Insert and \((\text{Extract or (Delete and Minimum)})\)

take \(o(\log n)\) amortized time. Then given an array \(A[1:n]\),
at least one of the following two algorithms,

\[
\begin{align*}
H &:= \text{Heap} \\
\text{for } i := 1 \text{ to } n \text{ do} \quad &\{ (i) \}
\text{    Insert } (A[i], A[i], H) \\
\text{ for } i := 1 \text{ to } n \text{ do} \quad &\{ (i) \}
\text{    } A[i] := \text{Extract } (H)
\end{align*}
\]

and

\[
\begin{align*}
H &:= \text{Heap} () \\
\text{for } i := 1 \text{ to } n \text{ do} \quad &\{ (2) \}
\text{    } P[i] := \text{Insert } (A[i], i, H) \\
\text{ for } i := 1 \text{ to } n \text{ do} \begin{align*}
&\text{begin} \\
&j := \text{Minimum } (H) \\
& A[i] := \text{Key } [P[j]] \\
& \text{Delete } (P[j], H)
\end{align*}
\end{align*}
\]

Sorts \(A\) in time \(\Theta(n) o(\log n) = o(n \log n)\), which is impossible,
since any comparison-based sorting algorithm must use
\(\Omega(n \log n)\) time.
Problem (Structure of Fibonacci heaps, II) Prove the following.

Definition The Fibonacci tree $T_k$, for $k \geq 0$, is

$$
T_k := \begin{cases} 
0, & k = 0; \\
1, & k = 1; \\
T_1, & k \geq 2.
\end{cases}
$$

Lemma For any Fibonacci heap $H$ and any $k \geq 0$, one can construct $H$ so that the Fibonacci tree $T_k$ is rooted at any specified node of $H$.

Proof We first show how to construct $T_k$ by Fibonacci heap operations, using induction on $k$. For $k = 0$, we do a single insert. For $k = 1$, we do three insert operations followed by one Extract. Thus the basis holds.

For $k \geq 2$, we:

(a) recursively (using the induction hypothesis)
Construct three heaps: $H_1 = T_{k-1}$, $H_2 = T_{k-1}$, $H_3 = T_0$

(b) Union $H_1$, $H_2$, $H_3$ together to form $H$.

(c) Delete the single node of $H_3$ in $H$, yielding

$$
\begin{array}{c}
S \\
T_k \\
T_{k-1}
\end{array} \quad \begin{array}{c}
T_k \\
T_{k-1}
\end{array}
$$

(d) Delete every node in the subtree rooted at the first child of $S$. 
**Prob. cont'd (Structure, II)**

**Proof cont'd**

By definition, $H_1$ has the form

So Step (d) converts $H_1$ into $T_{k-2}$ and thus $H$ into $T_k$.

Finally, to have $T_k$ rooted at any specified node $v$ of an arbitrary heap $H$, let $S$ be a sequence of Fibonacci heap operations that constructs $H$ except for the subtree rooted at $v$. At the point in $S$ at which $v$ is created, construct a separate heap $H' = T_k$ whose root has the same key as $v$, and union $H'$ with the heap into which $v$ was inserted. The remainder of $S$ will construct $H$ with $T_k$ rooted at $v$. \(\square\)
Problem (Fibonacci heaps on a pointer machine)

Show how to implement a Fibonacci heap on a pointer machine (i.e. on a random-access machine without arrays), so that all operations (in particular Extract) take the same amortized time.

Solution sketch

To efficiently perform Consolidate during Extract without using an array, maintain a linked list encoding the degree of a node:

- Every node in the heap has a pointer to the corresponding degree list. Since a node's degree changes by ≤ 1 during an operation, these pointers are easily maintained. Finding a root of the same degree is now easy during Consolidate.

- Also maintain $d$, the maximum degree of a node in the heap.

- On a Union of two heaps, merge the smaller of the degree lists into the larger.

- Add a term of $+d$ to $\Phi$, the potential function, to cancel out the merge time, so that a Union still takes $O(1)$ amortized time.
Problem (Flow with vertex capacities)

Find a maximum flow over a graph $G = (V, E, \delta)$ with source $s$ and sink $t$, in addition to edge capacities, each vertex $v \in V$ has a lower bound on its capacity that limits the flow out of $v$:

$$\sum_{w \in V} f(v, w) \leq c(v).$$  \hspace{1cm} (1)

Lower bound $f(v, w) \geq 0$

Algorithm

We reduce flow with vertex capacities to ordinary flow without vertex capacities by constructing a new graph $\tilde{G} = (V, \tilde{E}, \tilde{\delta})$. Each vertex $v \in V$ is mapped to two vertices $v_{in}$, $v_{out}$ and $e \in E$ is follows:

$$u \xrightarrow{c(v)} v \xleftarrow{\delta(u)} w \quad \rightarrow \quad \tilde{G} : v_{in} \xrightarrow{c(v)} v_{out} \xleftarrow{\delta(u)} w_{in}$$

The edge $(v_{in}, v_{out})$ has edge capacity $c(v)$.

In $\tilde{G}$, the flow from $v_{in}$ to $w_{in}, \ldots, w_{in}$ from $v_{out}$ will be $\leq c(v)$ which makes the vertex-capacity flow constraint (1) in $\tilde{G}$.

So a maximum flow in $\tilde{G}$ will give a maximum flow in $G$ satisfying (1).

$G$ has $O(n)$ vertices and $O(m)$ edges, $O(n + m) = O(m)$ edges, so assuming $m > n$, where $n = |V|$ and $m = |E|$. This reduces the time to compute a maximum flow in $\tilde{G}$ by computing a maximum flow in $G$.
Problem (Maximum cardinality bipartite matching)

Reduce maximum cardinality bipartite matching to maximum flow.

Lemma

g) Proposition

Let \( H = (V, E, c, s, t) \) be a flow instance in which all capacities are integers.

Then \( H \) has a maximum flow in which all flow values are integers that is integer-valued.

Proof Consider running the Preflow-Push algorithm to compute a maximum flow \( f \) for \( H \). All arithmetic operations performed by this algorithm will have integer results, given integer capacities. So it will terminate with a maximum flow that is integer-valued.

(b) Given an bipartite graph \( G = (V, E) \) with vertex bipartition \( X, Y \), we construct a maximum flow instance \( H \) as follows:

\[ H \]

\[ \begin{array}{c}
\text{Theory: A maximum flow of } H \text{ yields a maximum cardinality bipartite matching of } G.
\end{array} \]

Proof By the lemma from Part (a), \( H \) has an integer-valued maximum flow, \( f \). Let \( M \) be the subset of \( E \) that go between \( X, Y \).

By flow conservation, each vertex in \( X \) and \( Y \) is touched by \( f \) edge in \( M \). So \( M \) is a matching in \( G \).
Problem (Path and cycle cover)

Given a directed, edge-weighted graph \( G = (V, E, w) \), find a maximum-weight path and cycle cover:

- a subset \( C \subseteq E \) s.t. \( \forall (v, c) \in (V, c) \) every vertex has
  - in-degree \( \leq 1 \) and out-degree \( \leq 1 \)
  - and \( \sum w(c) \) is maximum.

Algorithm

To given graph \( G \), we construct an undirected bipartite graph \( \tilde{G} = (\tilde{V}, \tilde{E}, \tilde{w}) \) as follows.

Each vertex \( v \in V \) is mapped to two vertices \( v_{in}, v_{out} \).

Each directed edge \( (v, w) \in E \) is mapped to an undirected edge \( (v_{in}, w_{out}) \in \tilde{E} \).

Notice \( \tilde{G} \) is bipartite:

For a matching \( M \subseteq \tilde{E} \) corresponds to a path and cycle

in \( \tilde{G} \), where each in-vertex in \( M \) is

touched by \( \leq 1 \) edge in \( M \), and each out-vertex in \( M \)
simultaneously.

which captures the in- and out-degree constraints

in a path and cycle cover.

So, a maximum-weight bipartite matching in \( \tilde{G} \) yields a

maximum-weight path and cycle cover in \( G \).