18.3 Deleting a key from a B-tree

- B-TREE-DELETE deletes the key $k$ from the subtree rooted at $x$
- We design it to guarantee that whenever it calls itself recursively on a node $x$, the number of keys in $x$ is at least the minimum degree $t$
- This condition requires one more key than the minimum required by usual B-tree conditions
  - Sometimes a key may have to be moved into a child node before recursion descends to that child

- The strengthened condition allows us to delete a key in one downward pass without having to “back up” (with one exception)
- Interpret the following specification for deletion from a B-tree with the understanding that
  - if the root node $x$ ever becomes an internal node having no keys (this situation can occur in cases 2c and 3b),
  - then we delete $x$, and $x$’s only child $x.c_1$ becomes the new root of the tree,
  - decreasing the height of the tree by one and
  - preserving the property that the root of the tree contains at least one key (unless it is empty)
(a) initial tree

\[
\begin{array}{cccccc}
A & B & C & G & M & P \\
D & E & F & J & K & L \\
O & Q & R & S & U & V & Y & Z \\
\end{array}
\]

(b) \( F \) deleted: case 1

\[
\begin{array}{cccccc}
A & B & C & G & M & P \\
D & E & J & K & L & N & O \\
Q & R & S & U & V & Y & Z \\
\end{array}
\]

(c) \( M \) deleted: case 2a

\[
\begin{array}{cccccc}
A & B & C & G & L & P \\
D & E & J & K & N & O \\
Q & R & S & U & V & Y & Z \\
\end{array}
\]
(c) \( M \) deleted: case 2a

\[
\begin{array}{cccccccc}
P & \rightarrow & C & \rightarrow & G & \rightarrow & L & \rightarrow & \text{TX} \\
\downarrow & & \downarrow & & \downarrow & & \downarrow & & \\
A & \rightarrow & B & \rightarrow & D & \rightarrow & E & \rightarrow & J & \rightarrow & K & \rightarrow & N & \rightarrow & O & \rightarrow & Q & \rightarrow & R & \rightarrow & S & \rightarrow & U & \rightarrow & V & \rightarrow & Y & \rightarrow & Z
\end{array}
\]

(d) \( G \) deleted: case 2c

\[
\begin{array}{cccccccc}
P & \rightarrow & C & \rightarrow & L & \rightarrow & \text{TX} \\
\downarrow & & \downarrow & & \downarrow & & \\
A & \rightarrow & B & \rightarrow & D & \rightarrow & E & \rightarrow & J & \rightarrow & K & \rightarrow & N & \rightarrow & O & \rightarrow & Q & \rightarrow & R & \rightarrow & S & \rightarrow & U & \rightarrow & V & \rightarrow & Y & \rightarrow & Z
\end{array}
\]

(e) \( D \) deleted: case 3b

\[
\begin{array}{cccccccc}
P & \rightarrow & C & \rightarrow & L & \rightarrow & \text{TX} \\
\downarrow & & \downarrow & & \downarrow & & \\
A & \rightarrow & B & \rightarrow & E & \rightarrow & J & \rightarrow & K & \rightarrow & N & \rightarrow & O & \rightarrow & Q & \rightarrow & R & \rightarrow & S & \rightarrow & U & \rightarrow & V & \rightarrow & Y & \rightarrow & Z
\end{array}
\]
(e) $D$ deleted: case 3b

(e') tree shrinks in height

(f) $B$ deleted: case 3a
• Let us sketch how deletion works
1. If the key $k$ is a leaf node $x$, delete $k$ from $x$
2. If $k$ is in an internal node $x$, do the following:
   a) If the child $y$ that precedes $k$ in node $x$ has at least $t$ keys, then find the predecessor $k'$ of $k$ in the subtree rooted at $y$. Recursively delete $k'$, and replace $k$ by $k'$ in $x$. (We can find $k'$ and delete it in a single downward pass.)
   b) If $y$ has fewer than $t$ keys, then, symmetrically, examine the child $z$ that follows $k$ in node $x$. If $z$ has at least $t$ keys, then find the successor $k'$ of $k$ in the subtree rooted at $z$. Recursively delete $k'$, and replace $k$ by $k'$ in $x$.
   c) Otherwise, if both $y$ and $z$ have only $t-1$ keys, merge $k$ and all of $z$ into $y$, so that $x$ loses both $k$ and the pointer to $z$, and $y$ now contains $2t-1$ keys. Then free $z$ and recursively delete $k$ from $y$.
3. If the key $k$ is not present in internal node $x$, determine the root $x.c_i$ of the appropriate subtree that must contain $k$, if $k$ is in the tree at all. If $x.c_i$ has only $t-1$ keys, execute step 3a or 3b as necessary to guarantee that we descend to a node containing at least $t$ keys. Then finish by recursing on the appropriate child of $x$. 
a) If \( x.c_i \) has only \( t - 1 \) keys but has an immediate sibling with at least \( t \) keys, give \( x.c_i \) an extra key by moving a key from \( x \) down into \( x.c_i \), moving a key from \( x.c_i \)'s immediate left or right sibling up into \( x \), and moving the appropriate child pointer from the sibling into \( x.c_i \).

b) If \( x.c_i \) and both of \( x.c_i \)'s immediate siblings have \( t - 1 \) keys, merge \( x.c_i \) with one sibling, which involves moving a key from \( x \) down into the new merged node to become the median key for that node.

Most of the keys in a B-tree are in the leaves and we may expect that in practice deletions are most often used to delete keys from leaves.

- B-TREE-DELETE then acts in one downward pass through the tree, without having to back up.
- When deleting a key in an internal node, the procedure may have to return to replace the key with its predecessor or successor (2a and 2b).
- This involves only \( O(h) \) disk operations for a B-tree of height \( h \), since only \( O(1) \) calls to DISK-READ and DISK-WRITE are made between recursive invocations of the procedure.
- The CPU time required is \( O(th) = O(t \log_t n) \).
19 Fibonacci Heaps

1. The Fibonacci heap data structure supports a set of operations that constitutes what is known as a “mergeable heap”
2. Several Fibonacci-heap operations run in constant amortized time, which makes this data structure well suited for applications that invoke these operations frequently

Mergeable heaps

- Support the following operations, each element has a key:
  - MAKE-HEAP() creates and returns a new empty heap
  - INSERT(H, x) inserts element x, whose key has already been filled in, into heap H
  - MINIMUM(H) returns a pointer to the element in heap H whose key is minimum
  - EXTRACT-MIN(H) deletes the element from heap H whose key is minimum, returning a pointer to the element
• **UNION**($H_1, H_2$) creates and returns a new heap that contains all the elements of heaps $H_1$ and $H_2$. Heaps $H_1$ and $H_2$ are “destroyed” by this operation.
• Fibonacci heaps also support the following two operations:
  • **DECREASE-KEY**($H, x, k$) assigns to element $x$ within heap $H$ the new key value $k$, which we assume to be no greater than its current key value.
  • **DELETE**($H, x$) deletes element $x$ from heap $H$.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary Heap (worst-case)</th>
<th>Fibonacci Heap (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MAKE-HEAP</strong></td>
<td>$\theta(1)$</td>
<td>$\theta(1)$</td>
</tr>
<tr>
<td><strong>INSERT</strong></td>
<td>$\min(\log n)$</td>
<td>$\theta(1)$</td>
</tr>
<tr>
<td><strong>MINIMUM</strong></td>
<td>$\theta(1)$</td>
<td>$\theta(1)$</td>
</tr>
<tr>
<td><strong>EXTRACT-MIN</strong></td>
<td>$\theta(\log \pi)$</td>
<td>$\theta(\log \pi)$</td>
</tr>
<tr>
<td><strong>UNION</strong></td>
<td>$\theta(\pi)$</td>
<td>$\theta(1)$</td>
</tr>
<tr>
<td><strong>DECREASE-KEY</strong></td>
<td>$\theta(\log \pi)$</td>
<td>$\theta(1)$</td>
</tr>
<tr>
<td><strong>DELETE</strong></td>
<td>$\theta(\log \pi)$</td>
<td>$\theta(\log \pi)$</td>
</tr>
</tbody>
</table>
Fibonacci heaps in theory and practice

- Fibonacci heaps are especially desirable when the number of EXTRACT-MIN and DELETE operations is small relative to the number of other operations performed.
- E.g., some algorithms for graph problems may call DECREASE-KEY once per edge.
- For dense graphs, with many edges, the $\Theta(1)$ amortized time of each call of DECREASE-KEY is a big improvement over the $\Theta(\lg n)$ worst-case time of binary heaps.
- Fast algorithms for problems such as computing minimum spanning trees and finding single-source shortest paths make essential use of Fibonacci heaps.

- The constant factors and programming complexity of Fibonacci heaps make them less desirable than ordinary binary (or $k$-ary) heaps for most applications, except for certain applications that manage large amounts of data.
- Thus, Fibonacci heaps are predominantly of theoretical interest.
- If a much simpler data structure with the same amortized time bounds as Fibonacci heaps were developed, it would be of practical use as well.
- Fibonacci heaps are based on rooted trees
- We represent each element by a node within a tree, and each node has a key attribute
- We use the term “node” instead of “element”
- We also ignore issues of allocating nodes prior to insertion and freeing nodes following deletion
- A Fibonacci heap is a collection of rooted trees that are min-heap ordered
- I.e., each tree obeys the min-heap property:
  - the key of a node is greater than or equal to the key of its parent
• Each node $x$ contains a pointer $x.p$ to its parent and a pointer $x.child$ to any one of its children
• The children of $x$ are linked together in a circular, doubly linked list – the child list of $x$
• Each child $y$ in a child list has pointers $y.left$ and $y.right$ that point to $y$‘s left and right siblings, respectively
• If $y$ is an only child, then $y.left = y.right = y$
• Siblings may appear in a child list in any order
We store the number of children in the child list of node \( x \) in \( x.\text{degree} \). The Boolean attribute \( x.\text{mark} \) indicates whether node \( x \) has lost a child since the last time \( x \) was made the child of another node. Newly created nodes are unmarked, and a node \( x \) becomes unmarked whenever it is made the child of another node. Until we look at the DECREASE-KEY operation we will just set all mark attributes to FALSE. We access a given Fibonacci heap \( H \) by a pointer \( H.\text{min} \) to the root of a tree containing the minimum key.

When a Fibonacci heap \( H \) is empty, \( H.\text{min} \) is NIL. The roots of all the trees in a heap are linked together using their left and right pointers into a circular, doubly linked list called the root list. The pointer \( H.\text{min} \) thus points to the node in the root list whose key is minimum. Trees may appear in any order within a root list. We rely on one other attribute for a Fibonacci heap \( H: H.\text{n} \), the number of nodes currently in \( H \).
**Potential function**

- We use the potential method to analyze the performance of Fibonacci heap operations.
- Let $t(H)$ be the number of trees in the root list of Fibonacci heap $H$ and $m(H)$ the number of marked nodes in $H$.
- We define the potential $\Phi(H)$ of heap $H$ by $\Phi(H) = t(H) + 2m(H)$.
- For example, the potential of the Fibonacci heap shown above is $5 + 2 \cdot 3 = 11$.

The potential of a set of Fibonacci heaps is the sum of the potentials of its constituent heaps.
- We assume that a unit of potential can cover the cost of any of the specific constant-time pieces of work that we might encounter.
- Fibonacci heap application begins with no heaps.
- The initial potential, therefore, is 0, and the potential is nonnegative at all subsequent times.
- An upper bound on the total amortized cost thus provides an upper bound on the total actual cost for the sequence of operations.
Maximum degree

- Amortized analyses we perform assume that we know an upper bound $D(n)$ on the maximum degree of any node in an $n$-node Fibonacci heap
- When only the mergeable-heap operations are supported $D(n) \leq \lfloor \lg n \rfloor$
- We shall show that when we support DECREASE-KEY and DELETE as well, $D(n) = O(\lg n)$

19.2 Mergeable-heap operations

- The operations delay work as long as possible; various operations have performance trade-offs
- E.g., we insert a node by adding it to the root list, which takes just constant time
- If we insert $k$ nodes to an empty Fibonacci heap $H$, the heap consists of just a root list of $k$ nodes
- Trade-off: if we then perform EXTRACT-MIN on $H$, after removing the node that $H.\text{min}$ points to, we have to look through each of the remaining $k - 1$ nodes to find the new minimum node
As long as we have to go through the entire root list during the EXTRACT-MIN operation,
  – we also consolidate nodes into min-heap-ordered trees to reduce the size of the root list

We shall see that, no matter what the root list looks like before a EXTRACT-MIN operation,
  – afterward each node in the root list has a degree that is unique within the root list, which leads to a root list of size at most $D(n) + 1$

---

**Creating a new Fibonacci heap**

- To make an empty Fibonacci heap, the MAKE-FIB-HEAP procedure allocates and returns the Fibonacci heap object $H$, where $H.n = 0$ and $H.min = \text{NIL}$; there are no trees in $H$
- Because $t(H) = 0$ and $m(H) = 0$, the potential of the empty Fibonacci heap is $\Phi(H) = 0$
- The amortized cost of MAKE-FIB-HEAP is thus equal to its $O(1)$ actual cost
**Fib-Heap-Insert**($H, x$)

1. $x\text{.degree} = 0$
2. $x\text{.p} = \text{NIL}$
3. $x\text{.child} = \text{NIL}$
4. $x\text{.mark} = \text{FALSE}$
5. **if** $H\text{.min} == \text{NIL}$
6. create a root list for $H$ containing just $x$
7. $H\text{.min} = x$
8. **else** insert $x$ into $H$’s root list
9. **if** $x\text{.key} < H\text{.min.key}$
10. $H\text{.min} = x$
11. $H\text{.n} = H\text{.n} + 1$
To determine the amortized cost of \textsc{Fib-Heap-Insert}, let $H$ be the input Fibonacci heap and $H'$ be the resulting Fibonacci heap.

Then, $t(H') = t(H) + 1$ and $m(H') = m(H)$, and the increase in potential is 

$$((t(H) + 1) + 2m(H)) - (t(H) + 2m(H)) = 1.$$

Since the actual cost is $O(1)$, the amortized cost is 

$$O(1) + 1 = O(1).$$

\begin{algorithm}
\textsc{Fib-Heap-Union}(H_1, H_2)
\begin{enumerate}
\item $H = \text{Make-Fib-Heap}()$
\item $H.\text{min} = H_1.\text{min}$
\item concatenate the root list of $H_2$ with the root list of $H$
\item \textbf{if} ($H_1.\text{min} == \text{NIL}$ \textbf{or} ($H_2.\text{min} \neq \text{NIL}$ \textbf{and} $H_2.\text{min}.\text{key} < H_1.\text{min}.\text{key}$)
\item $H.\text{min} = H_2.\text{min}$
\item $H.n = H_1.n + H_2.n$
\item \textbf{return} $H$
\end{enumerate}
\end{algorithm}
The change in potential is

\[
\Phi(H) - (\Phi(H_1) + \Phi(H_2)) \\
= (t(H) + 2m(H)) - ((t(H_1) + 2m(H_1)) + (t(H_2) + 2m(H_2))) \\
= 0
\]

because \( t(H) = t(H_1) + t(H_2) \) and \( m(H) = m(H_1) + m(H_2) \)

The amortized cost of \textsc{Fib-Heap-Union} is therefore equal to its \( O(1) \) actual cost.

---

Extracting the minimum node

- The process of extracting the minimum node is the most complicated of the operations so far
- It is also where the delayed work of consolidating trees in the root list finally occurs
- The following code assumes that when a node is removed, pointers remaining in the linked list are updated, but pointers in the extracted node are left unchanged
- It also calls the auxiliary procedure \textsc{Consolidate}
FIB-HEAP-EXTRACT-MIN\((H)\)

1. \(z = H.\text{min}\)
2. if \(z \neq \text{NIL}\)
3. for each child \(x\) of \(z\)
4. add \(x\) to the root list of \(H\)
5. \(x.p = \text{NIL}\)
6. remove \(z\) from the root list of \(H\)
7. if \(z == z.\text{right}\)
8. \(H.\text{min} = \text{NIL}\)
9. else \(H.\text{min} = z.\text{right}\)
10. CONSOLIDATE\((H)\)
11. \(H.n = H.n - 1\)
12. return \(z\)
• The next step reduces the number of trees in the Fibonacci heap, CONSOLIDATE($H$) accomplishes this.

• Consolidating the root list consists of repeatedly executing the following steps until every root in the root list has a distinct degree value:
  1. Find two roots $x$ and $y$ in the root list with the same degree. Without loss of generality, let $x.key \leq y.key$
  2. Link $y$ to $x$: remove $y$ from the root list, and make $y$ a child of $x$ by calling the Fib-HEAP-LINK procedure. This procedure increments the attribute $x.degree$ and clears the mark on $y$. 

Decreasing a key

**FIB-HEAP-DECREASE-KEY**($H, x, k$)

1. if $k > x.key$
2. error “new key is greater than current key”
3. $x.key = k$
4. $y = x.p$
5. if $y \neq \text{NIL}$ and $x.key < y.key$
6. CUT($H, x, y$)
7. CASCADING-CUT($H, y$)
8. if $x.key < H.min.key$
9. $H.min = x$
\textbf{Cut}(H,x,y)

1. remove \(x\) from the child list of \(y\), decrementing \(y\. \text{degree}\)
2. add \(x\) to the root list of \(H\)
3. \(x\. p = \text{NIL}\)
4. \(x\. \text{mark} = \text{FALSE}\)

\textbf{Cascading-Cut}(H,y)

1. \(z = y\. p\)
2. \textbf{if} \(z \neq \text{NIL}\)
3. \textbf{if} \(y\. \text{mark} == \text{FALSE}\)
4. \(y\. \text{mark} = \text{TRUE}\)
5. \textbf{else} \textbf{Cut}(H,y,z)
6. \textbf{Cascading-Cut}(H,z)
• **Fib-Heap-Decrease-Key** creates a new tree rooted at node \( x \) and clears \( x \)'s mark bit

• Each of the \( c \) calls of **Cascading-Cut**, except the last one, cuts a marked node and clears the mark bit

• Afterward, the heap contains \( t(H) + c \) trees
  - the original \( t(H) \) trees, \( c - 1 \) trees produced by cascading cuts, and the tree rooted at \( x \)
  - and at most \( m(H) - c + 2 \) marked nodes
  - \( c - 1 \) were unmarked by cascading cuts and the last call of **Cascading-Cut** may have marked a node

• The change in potential is therefore at most
  \[
  \left( (t(H) + c) + 2(m(H) - c + 2) \right) - (t(H) + 2m(H)) = 4 - c
  \]

• Thus, the amortized cost of **Fib-Heap-Decrease-Key** is at most \( O(c) + 4 - c = O(1) \), since we can scale up the units of potential to dominate the constant hidden in \( O(c) \)

• When a marked node \( y \) is cut by a cascading cut, its mark bit is cleared, which reduces the potential by 2

• One unit of potential pays for the cut and the clearing of the mark bit, and the other unit compensates for the unit increase in potential due to node \( y \) becoming a root
Deleting a node

- We assume that there is no key value of $-\infty$ currently in the Fibonacci heap

$\text{FIB-HEAP-DELETE}(H, x)$
1. $\text{FIB-HEAP-DECREASE-KEY}(H, x, -\infty)$
2. $\text{FIB-HEAP-EXTRACT-MIN}(H)$

- The amortized time of $\text{FIB-HEAP-DELETE}$ is the sum of the $O(1)$ amortized time of $\text{FIB-HEAP-DECREASE-KEY}$ and the $O(D(n))$ amortized time of $\text{FIB-HEAP-EXTRACT-MIN}$