18 B-Trees

- B-trees are similar to RBTs, but they are better at minimizing disk I/O operations.
- Many database systems use B-trees, or variants of them, to store information.
- B-tree nodes may have many children, from a few to thousands.
- The branching factor of a B-tree can be quite large, although it usually depends on characteristics of the disk unit used.
• B-trees are similar to RBTs in that every $n$-node B-tree has height $O(\lg n)$
• The exact height of a B-tree can be considerably less than that of a RBT, because its branching factor – the base of the logarithm that expresses its height – can be much larger
• Therefore, we can also use B-trees to implement many dynamic-set operations in time $O(\lg n)$
• B-trees generalize BSTs in a natural manner
  – If an internal B-tree node $x$ contains $x.n$ keys, then $x$ has $x.n + 1$ children
18.1 Definition of B-trees

• A B-tree $T$ is a rooted tree (whose root is $T.root$) having the following properties:

1. Every node $x$ has the following attributes:
   a) $x.n$, the number of keys currently stored in $x$,
   b) the $x.n$ keys themselves, $x.key_1$, $x.key_2$, ..., $x.key_{x.n}$, stored in nondecreasing order, so that $x.key_1 \leq x.key_2 \leq \cdots \leq x.key_{x.n}$,
   c) $x.leaf$, a Boolean value that is TRUE if $x$ is a leaf and FALSE if $x$ is an internal node

2. Each internal node $x$ also contains $x.n + 1$ pointers $x.c_1$, $x.c_2$, ..., $x.c_{x.n+1}$ to its children. Leaf nodes have no children, and so their $c_i$ attributes are undefined

3. The keys $x.key_i$ separate the ranges of keys stored in each subtree: if $k_i$ is any key stored in the subtree with root $x.c_i$, then $k_1 \leq x.key_1 \leq k_2 \leq x.key_2 \leq \cdots \leq x.key_{x.n} \leq k_{x.n+1}$

4. All leaves have the same depth, which is the tree’s height $h$
5. Nodes have lower and upper bounds on the number of keys they can contain. We express these bounds in terms of a fixed integer \( t \geq 2 \) called the **minimum degree** of the B-tree:

a) Every node (except the root) must have at least \( t - 1 \) keys. Every internal node (except the root) thus has at least \( t \) children. If the tree is nonempty, the root must have at least one key.

b) Every node may contain at most \( 2t - 1 \) keys. Therefore, an internal node may have at most \( 2t \) children. We say that a node is **full** if it contains exactly \( 2t - 1 \) keys.

The height of a B-tree

- The simplest B-tree occurs when \( t = 2 \)
- Every internal node then has either 2, 3, or 4 children, and we have a 2-3-4 tree
- In practice, however, much larger values of \( t \) yield B-trees with smaller height

**Theorem 18.1** If \( n \geq 1 \), then for any \( n \)-key B-tree \( T \) of height \( h \) and minimum degree \( t \geq 2 \),

\[
h \leq \log_t \frac{n + 1}{2}
\]
Proof  The root of a B-tree $T$ contains at least one key, and all other nodes contain at least $t-1$ keys. Thus, $T$, whose height is $h$, has at least 2 nodes at depth 1, at least $2t$ nodes at depth 2, at least $2t^2$ nodes at depth 3, and so on, until at depth $h$ it has at least $2t^{h-1}$ nodes. Thus, the number $n$ of keys satisfies the inequality

$$n \geq 1 + (t - 1) \sum_{i=1}^{h} 2t^{i-1} = 1 + 2(t - 1)\left(\frac{t^h - 1}{t - 1}\right) = 2t^h - 1$$

We get $t^h \leq (n + 1)/2$. Taking base-$t$ logarithms of both sides proves the theorem. ■

Here is the power of B-trees, w.r.t. RBTs
- The height of the tree grows as $O(\log n)$ in both cases ($t$ is a constant)
- For B-trees the base of the logarithm can be many times larger
- Thus, B-trees save a factor of about $\log t$ over RBTs in the number of nodes examined for most tree operations
- Because we usually have to access the disk to examine an arbitrary node in a tree, B-trees avoid a substantial number of disk accesses
18.2 Basic operations on B-trees

- Searching a B-tree, at each internal node \( x \), we make an \( (x.n + 1) \)-way branching decision
- B-TREE-SEARCH is a simple generalization of the TREE-SEARCH procedure defined for BSTs
- B-TREE-SEARCH inputs are a pointer to the root node \( x \) and key \( k \) to be searched in that subtree
- The top-level call is B-TREE-SEARCH(\( T.root, k \))
- If \( k \) is in the tree, it returns the ordered pair \( (y, i) \) of a node \( y \) and an index \( i \) s.t. \( y.key_i = k \)

B-TREE-SEARCH\( (x, k) \)

1. \( i \leftarrow 1 \)
2. while \( i \leq x.n \) and \( k > x.key_i \)
3. \( i \leftarrow i + 1 \)
4. if \( i \leq x.n \) and \( k = x.key_i \)
5. return \( (x, i) \)
6. elseif \( x.leaf \)
7. return NIL
8. else DISK-READ\( (x.c_i) \)
9. return B-TREE-SEARCH\( (x.c_i, k) \)
• As in the TREE-SEARCH procedure for BSTs, the nodes encountered during the recursion form a simple path downward from the root of the tree.

• B-TREE-SEARCH accesses $O(h) = O(\log_t n)$ disk pages, where $h$ is the height of the B-tree and $n$ is the number of keys.

• Since $x.n < 2t$, the while loop of lines 2–3 takes $O(t)$ time within each node, and the total CPU time is $O(th) = O(t \log_t n)$.

Creating an empty B-tree

• ALLOCATE-NODE allocates one disk page to be used as a new node in $O(1)$ time.

• It requires no DISK-READ, since there is as yet no useful information stored on the disk.

B-TREE-CREATE($T$)

1. $x \leftarrow$ ALLOCATE-NODE()
2. $x.leaf \leftarrow$ TRUE
3. $x.n \leftarrow 0$
4. DISK-WRITE($x$)
5. $T.root \leftarrow x$
Inserting a key into a B-tree

- We insert the new key into an existing leaf node.
- We need an operation that splits a full node $y$ (having $2t - 1$ keys) around its median key $y.key_l$ into two nodes having only $t - 1$ keys.
- Median key moves up into $y$'s parent to identify the dividing point between the two new trees.
- But if $y$'s parent is also full, we must split it before we can insert the new key; we could end up splitting full nodes all the way up the tree.

- As with a BST, we can insert a key into a B-tree in a single pass down from the root to a leaf.
- We do not wait to find out whether we will actually need to split a full node in order to do the insertion.
- As we travel down the tree, we split each full node we come to along the way (including the leaf itself).
- Thus whenever we want to split a full node $y$, we are assured that its parent is not full.
Splitting a node in a B-tree

- B-TREE-SPLIT-CHILD takes as input a nonfull internal node $x$ (in main memory) and an index $i$ such that $x.c_i$ (in main memory) is full.
- The procedure then splits this child in two and adjusts $x$ so that it has an additional child.
- To split a full root, we will first make the root a child of a new empty root node, so that we can use B-TREE-SPLIT-CHILD.
- The tree thus grows in height by one; splitting is the only means by which the tree grows.
B-TREE-SPLIT-CHILD\((x, i)\)

1. \(z \leftarrow \text{ALLOCATE-NODE()}
2. \(y \leftarrow x.c_i\)
3. \(z.\text{leaf} \leftarrow y.\text{leaf}\)
4. \(z.n \leftarrow t - 1\)
5. \(\text{for } j \leftarrow 1 \text{ to } t - 1\)
6. \(z.\text{key}_j \leftarrow y.\text{key}_{j+t}\)
7. \(\text{if not } y.\text{leaf}\)
8. \(\text{for } j \leftarrow 1 \text{ to } t\)
9. \(z.c_j \leftarrow y.c_{j+t}\)
10. \(y.n \leftarrow t - 1\)
11. \(\text{for } j \leftarrow x.n + 1 \text{ downto } i + 1\)
12. \(x.c_{j+1} \leftarrow x.c_j\)
13. \(x.c_{i+1} \leftarrow z\)
14. \(\text{for } j \leftarrow x.n \text{ downto } i\)
15. \(x.\text{key}_{j+1} \leftarrow x.\text{key}_j\)
16. \(x.\text{key}_i \leftarrow y.\text{key}_t\)
17. \(x.n \leftarrow x.n + 1\)
18. \(\text{DISK-WRITE}(y)\)
19. \(\text{DISK-WRITE}(z)\)
20. \(\text{DISK-WRITE}(x)\)

B-TREE-INSERT\((T, k)\)

1. \(r \leftarrow T.\text{root}\)
2. \(\text{if } r.n = 2t - 1\)
3. \(s \leftarrow \text{ALLOCATE-NODE()}\)
4. \(T.\text{root} \leftarrow s\)
5. \(s.\text{leaf} \leftarrow \text{FALSE}\)
6. \(s.n \leftarrow 0\)
7. \(s.c_1 \leftarrow r\)
8. \(\text{B-TREE-SPLIT-CHILD}(s, 1)\)
9. \(\text{B-TREE-INSERT-NONFULL}(s, k)\)
10.\(\text{else B-TREE-INSERT-NONFULL}(r, k)\)
B-TREE-INSERT-NONFULL \((x, k)\)
1. \(i \leftarrow x.n\)
2. if \(x.leaf\)
3. while \(i \geq 1\) and \(k < x.key_i\)
4. \(x.key_{i+1} \leftarrow x.key_i\)
5. \(i \leftarrow i - 1\)
6. \(x.key_{i+1} \leftarrow k\)
7. \(x.n \leftarrow x.n + 1\)
8. DISK-WRITE\((x)\)

9. else while \(i \geq 1\) and \(k < x.key_i\)
10. \(i \leftarrow i - 1\)
11. \(i \leftarrow i + 1\)
12. DISK-READ\((x.c_i)\)
13. if \(x.c_i.n = 2t - 1\)
14. B-TREE-SPLIT-CHILD\((x, i)\)
15. if \(k > x.key_i\)
16. \(i \leftarrow i + 1\)
17. B-TREE-INSERT-NONFULL\((x.c_i, k)\)
For a B-tree of height $h$, B-TREE-INSERT performs $O(h)$ disk accesses, since only $O(1)$ DISK-READ and DISK-WRITE operations occur between calls to B-TREE-INSERT-NONFULL.

The total CPU time used is $O(th) = O(t \log_t n)$.

B-TREE-INSERT-NONFULL is tail-recursive, and can alternatively be implemented as a while loop:

- the number of pages that need to be in main memory at any time is $O(1)$

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)

$t = 3$, therefore $\text{min} = 2$ and $\text{max} = 5$
(e) $D$ deleted: case 3b
Let us sketch how deletion works

1. If the key $k$ is in 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 $x$. 

![Diagram of deletion process]
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; we expect that in practice deletions are most often used to delete keys from leaves
• B-TREE-DELETE acts in one downward pass through the tree, without having to back up
• 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)
• 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 \( H \)
### 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.

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Binary Heap (worst-case)</th>
<th>Fibonacci Heap (amortized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAKE-HEAP</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>INSERT</td>
<td>( \Theta(\lg n) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>( \Theta(1) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>EXTRACT-MIN</td>
<td>( \Theta(\lg n) )</td>
<td>( \Theta(\lg n) )</td>
</tr>
<tr>
<td>UNION</td>
<td>( \Theta(n) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>DECREASE-KEY</td>
<td>( \Theta(\lg n) )</td>
<td>( \Theta(1) )</td>
</tr>
<tr>
<td>DELETE</td>
<td>( \Theta(\lg n) )</td>
<td>( \Theta(\lg n) )</td>
</tr>
</tbody>
</table>
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 ones 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 show that when we support \textsc{Decrease-Key} and \textsc{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 consist of just a root list of \( k \) nodes
- **Trade-off:** if we then perform **EXTRACT-MIN** on \( H \), after removing the node that \( H.\min \) 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

\[
\text{FIB-HEAP-INSERT}(H, x)
\]

1. \( x.\text{degree} \leftarrow 0 \)
2. \( x.p \leftarrow \text{NIL} \)
3. \( x.\text{child} \leftarrow \text{NIL} \)
4. \( x.\text{mark} \leftarrow \text{FALSE} \)
5. \textbf{if} \( H.min = \text{NIL} \)
6. \quad create a root list for \( H \) containing just \( x \)
7. \quad \( H.min \leftarrow x \)
8. \textbf{else} insert \( x \) into \( H \)'s root list
9. \textbf{if} \( x.key < H.\text{min.key} \)
10. \quad \( H.min \leftarrow x \)
11. \( H.n \leftarrow H.n + 1 \)
To determine the amortized cost of `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:

\[ O(1) + 1 = O(1) \]
FIB-HEAP-UNION$(H_1,H_2)$
1. $H \leftarrow \text{MAKE-FIB-HEAP}()$
2. $H\.\text{min} \leftarrow H_1\.\text{min}$
3. concatenate the root list of $H_2$ with the root list of $H$
4. if $(H_1\.\text{min} = \text{NIL})$ or $(H_2\.\text{min} \neq \text{NIL}$ and $H_2\.\text{min}.\text{key} < H_1\.\text{min}.\text{key})$
5. $H\.\text{min} \leftarrow H_2\.\text{min}$
6. $H\.n \leftarrow H_1\.n + H_2\.n$
7. return $H$

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 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 CONSOLIDATE

```
FIB-HEAP-EXTRACT-MIN(H)
1. z ← H.min
2. if z ≠ NIL
3. for each child x of z
4. add x to the root list of H
5. x.p ← NIL
6. remove z from the root list of H
7. if z = z.right
8. H.min ← NIL
9. else H.min ← z.right
10. CONSOLIDATE(H)
11. H.n ← H.n − 1
12. return z
```
CONSOLIDATE\((H)\) reduces the number of trees in the Fibonacci heap

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. 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 \(y\)