Chapter 18: B-Trees

A B-tree is a balanced tree scheme in which balance is achieved by permitting the nodes to have multiple keys and more than two children.
**Definition** Let $t \geq 2$ be an integer. A tree $T$ is called a **B-tree** having **minimum degree** $t$ if the leaves of $T$ are at the same depth and each node $u$ has the following properties:

1. $u$ has at most $2t - 1$ keys.
2. If $u$ is not the root, $u$ has at least $t - 1$ keys.
3. The keys in $u$ are sorted in the increasing order.
4. The number of $u$’s children is precisely one more than the number of $u$’s keys.
5. For all $i \geq 1$, if $u$ has at least $i$ keys and has children, then every key appearing in the subtree rooted at the $i$-th child of $u$ is less than the $i$-th key and every key appearing in the subtree rooted at the $(i + 1)$-st child of $u$ is greater than the $i$-th key.
**Notation**

Let $u$ be a node in a B-tree. By $n[u]$ we denote the number of keys in $u$. For each $i$, $1 \leq i \leq n[u]$, $key_i[u]$ denotes the $i$-th key of $u$. For each $i$, $1 \leq i \leq n[u] + 1$, $c_i[u]$ denotes the $i$-th child of $u$.

**Terminology**

For the sake of simplicity we will introduce some terminology.

We say that a node is full if it has $2t - 1$ keys and we say that a node is lean if it has the minimum number of keys, that is $t - 1$ keys in the case of a non-root and 1 in the case of the root.

**2-3-4 Trees**

B-trees with maximum degree 2 are called 2-3-4 trees to signify that the number of children is two, three, or four.
**Depth of a B-tree**

**Theorem A**  Let $t \geq 2$ and $n$ be integers. Let $T$ be an arbitrary B-tree with minimum degree $t$ having $n$ keys. Let $h$ be the height of $T$. Then $h \leq \log_t \frac{n+1}{2}$.

**Proof** The height is maximized when all the nodes are lean. If $T$ is of that form, the number of keys in $T$ is

$$1 + \sum_{i=1}^{h} 2(t-1)t^{i-1} = 2(t-1)\frac{t^h - 1}{t - 1} + 1 = 2^h - 1.$$

Thus the depth of a B-tree is at most $\frac{1}{\log t}$ of the depth of an RB-tree.
**Searching** for a key $k$ in a B-tree

Start with $x = \text{root}[T]$.

1. If $x = \text{nil}$, then $k$ does not exist.
2. Compute the smallest $i$ such that the $i$-th key at $x$ is greater than or equal to $k$.
3. If the $i$-th key is equal to $k$, then the search is done.
4. Otherwise, set $x$ to the $i$-th child.

The number of examinations of the key is

$$O((2t - 1)h) = O(t \log_t(n + 1)/2).$$

Binary search may improve the search within a node

*How do we search for a predecessor?*
B-Tree-Predecessor \((T, x, i)\)

1: \(\triangleright\) Find a pred. of \(key_i[x]\) in \(T\)
2: \(\textbf{if } i \geq 2 \textbf{ then}\)
3: \(\qquad \textbf{if } c_i[x] = \text{nil} \textbf{ then return } key_{i-1}[x]\)
4: \(\triangleright\) If \(i \geq 2 \& x\) is a leaf
5: \(\triangleright\) return the \((i-1)\)st key
6: \(\textbf{else}\ {\{\}
7: \(\triangleright\) If \(i \geq 2 \& x\) is not a leaf
8: \(\triangleright\) find the rightmost key in the \(i\)-th child
9: \(\quad y \leftarrow c_i[x]\)
10: \(\quad \textbf{repeat}\)
11: \(\quad \quad z \leftarrow c_n[y]+1\)
12: \(\quad \quad \textbf{if } z \neq \text{nil} \textbf{ then } y \leftarrow z\)
13: \(\quad \textbf{until } z = \text{nil}\)
14: \(\quad \textbf{return } key_n[y][y]\)
15: \(\}\)
16:   else 
17:      \triangleright\text{Find } y \text{ and } j \geq 1 \text{ such that} 
18:      \triangleright\text{ } x \text{ is the leftmost key in } c_j[y] 
19:      \text{while } y \neq \text{root}[T] \text{ and } c_1[p[y]] = y \text{ do} 
20:      \quad y \leftarrow p[y] 
21:      j \leftarrow 1 
22:      \text{while } c_j[p[y]] \neq y \text{ do } j \leftarrow j + 1 
23:      \text{if } j = 1 \text{ then return } \text{“No Predecessor”} 
24:      \text{return } key_{j-1}[p[y]] 
25:   }
Basic Operations, Split & Merge

Split takes a full node $x$ as part of the input. If $x$ is not the root, then its parent should not be full. The input node is split into three parts: the middle key, a node that has everything to the left of the middle key, and a node that has everything to the right of the middle key. Then the three parts replace the pointer pointing to $x$. As a result, in the parent node the number of children and the number of keys are both increased by one.
inserted

node y

node x

node y

node z

\text{t=4}
B-Tree-Split(T, x)
1: if n[x] < 2t − 1 then return "Can’t Split"
2: if x ≠ root[T] and n[p[x]] = 2t − 1 then
3: return "Can’t Split"
4: ▷ Create new nodes y and z
5: n[y] ← t − 1
6: n[z] ← t − 1
7: for i ← 1 to t − 1 do {
8: key_i[y] ← key_i[x]
9: key_i[z] ← key_{i+t}[x]
10: }
11: for i ← 1 to t do {
12: c_i[y] ← c_i[x]
13: c_i[z] ← c_{i+t}[x]
14: }
15: ▷ If x is the root then create a new root
16: if x = root[T] then {
17: ▷ Create a new node v
18: n[v] ← 0
19: c_1[v] ← x
20: p[x] ← v
21: root[T] ← v
22: }
23: ▷ Find the spot for insertion
24: \( j \leftarrow 1 \)
25: \textbf{while} \( c_j[p[x]] \neq x \) \textbf{do} \( j \leftarrow j + 1 \)
26: ▷ Open up space for insertion
27: \textbf{if} \( j \leq n[p[x]] \) \textbf{then}
28: \quad \textbf{for} \( i \leftarrow n[p[x]] \) \textbf{downto} \( j \) \textbf{do} \{ \\
29: \quad \quad key_{i+1}[p[x]] \leftarrow key_i[p[x]] \\
30: \quad \quad c_{i+2}[p[x]] \leftarrow c_{i+1}[p[x]] \\
31: \quad \}
32: ▷ Insertion
33: \quad key_j[p[x]] \leftarrow key_t[x] \\
34: \quad c_j[p[x]] \leftarrow y \\
35: \quad c_{j+1}[p[x]] \leftarrow z \\
36: \quad n[p[x]] \leftarrow n[p[x]] + 1 \\
37: \quad p[y] \leftarrow p[x] \\
38: \quad p[z] \leftarrow p[x] \\
39: ▷ Return the pointer to the parent
40: \textbf{return} \( p[x] \)
Merge takes as input a node and the position of a key. Then it merges the key and the pair of children flanking the key into one node.

What kind of properties must the input node and the children satisfy for such an operation be possible?
Well, the input node must not be lean, and the two children must be lean.

That’s right.
node $x$ dropped

node $z$

node $y$

$B$, $D$, $F$, $H$, $m$, $R$, $X$

$t=4$

$J$, $K$, $L$, $m$, $NP$, $Q$
B-Tree-Merge\( (T, x, i) \)

1: \( \triangleright \) Merge the \( i \)-th key of \( x \) and the two children flanking the \( i \)-th key

2: \( \triangleright \) children flanking the \( i \)-th key

3: \( y \leftarrow c_i[x] \) \( \triangleright \) \( y \) is the left child

4: \( z \leftarrow c_{i+1}[x] \) \( \triangleright \) \( z \) is the right child

5: \( \textbf{if} \ (n[y] > t - 1 \ \textbf{or} \ n[z] > t - 1) \ \textbf{then} \)

6: \( \quad \textbf{return} \ \text{“Can’t Merge”} \)

7: \( \text{key}_t[y] \leftarrow \text{key}_i[x] \) \( \triangleright \) Append the middle key

8: \( \quad \textbf{for} \ j \leftarrow 1 \ \textbf{to} \ t - 1 \ \textbf{do} \)

9: \( \quad \text{key}_{t+j}[y] \leftarrow \text{key}_j[z] \) \( \triangleright \) Copy keys from \( z \)

10: \( \quad \textbf{for} \ j \leftarrow 1 \ \textbf{to} \ t \ \textbf{do} \{ \)

11: \( \quad c_{t+j}[y] \leftarrow c_j[z] \) \( \triangleright \) Copy children from \( z \)

12: \( \quad p[c_j[z]] \leftarrow y \) \( \triangleright \) Fix the parent pointers

13: \( \text{\} } \)

14: \( n[x] \leftarrow n[x] - 1 \) \( \triangleright \) Fix the \( n \)-tag

15: \( \textbf{if} \ (n[x] = 0) \ \textbf{then} \{ \)

16: \( \quad \text{root}[T] \leftarrow y \) \( \triangleright \) and was lean, then

17: \( \quad p[y] \leftarrow \text{nil} \) \( \triangleright \) \( y \) becomes the root

18: \( \text{\} } \)
19: ▶ If the middle key is not the last key
20: ▶ Fill the gap by moving things
21: \textbf{else if} \ i \leq \ n[x] \ \textbf{then} \ { 
21: \hspace{1em} \textbf{for} \ j \leftarrow \ i \ \textbf{to} \ n[x] \ \textbf{do} 
22: \hspace{2em} \text{key}_{j}[x] \leftarrow \text{key}_{j+1}[x] 
23: \hspace{1em} \textbf{for} \ j \leftarrow \ i \ \textbf{to} \ n[x] \ \textbf{do} 
24: \hspace{2em} c_{j}[x] \leftarrow c_{j+1}[x] 
25: \}
Insertion of a key

Suppose that a key $k$ needs to be inserted in the subtree rooted at $y$ in a B-tree $T$.

Before inserting the key we make sure that is room for insertion, that is, not all the nodes in the subtree are full. Since visiting all the nodes in the subtree is very costly, we will make sure only that $y$ is not full.

If $y$ is a leaf, insert the key. If not, find a child in which the key should go to and then make a recursive call with $y$ set to the child.
the initial tree

B inserted

Q inserted

L inserted

F inserted
B-Tree-Insert($T, y, k$)

1: $z \leftarrow y$
2: $f \leftarrow \text{false}$
3: while $f = \text{false}$ do {
4: \hspace{1em} if $n[z] = 2t - 1$ then
5: \hspace{2em} $z \leftarrow \text{B-Tree-Split}(T, z)$
6: \hspace{1em} $j \leftarrow 1$
7: \hspace{1em} while $\text{key}_j[z] < k$ and $j \leq n[z]$ do
8: \hspace{2em} $j \leftarrow j + 1$
9: \hspace{1em} if $c_j[z] \neq \text{nil}$ then $z \leftarrow c_j[z]$
10: \hspace{1em} else $f \leftarrow \text{true}$
11: }
12: for $i \leftarrow n[z]$ downto $j$ do
13: \hspace{1em} $\text{key}_{j+1}[z] \leftarrow \text{key}_j[z]$
14: \hspace{1em} $\text{key}_j \leftarrow k$
15: $n[z] \leftarrow n[z] + 1$
16: return $z$
Deletion

The task is to receive a key $k$ and a B-tree $T$ as input and eliminate it from $T$ if it is in the tree. To accomplish this, we will take an approach similar to that we took for binary search trees.

- Search for $k$. If the node containing $k$ is a leaf, eliminate $k$.
- Otherwise, search for the predecessor $k$ in the subtree immediate to the right of $k$. Relocate the predecessor to the position of $k$.

*What should we be careful about?*
Well, we should avoid removing a key from a lean leaf.

To avoid such a case, probably we should take a strategy similar to that we took in Insertion, that is, when a node is about to be visited, make sure that the node is not lean.

*That's correct.*
**Strategy**

When a lean node $x$ is about to be visited, do the following:

- In the case when $x$ is not the first child, if its immediate left sibling is not lean move the last key and the last child of the sibling to $x$; otherwise merge the sibling, $x$, and the key between them into one.

- In the case when $x$ is the first child, if its immediate right sibling is not lean move the first key and the first child of the sibling to $x$; otherwise merge the sibling, $x$, and the key between them into one.

We can then assume that if $x$ is not the root then its parent is not lean.
node y \rightarrow root \rightarrow node z \rightarrow new root
B-Tree-Delete($T, k$)

1: ▷ Search for $k$
2: $x \leftarrow \text{root}[T]$
3: \textbf{if} ($n[x] = 1 \text{ and } n[c_1[x]] = t - 1 \text{ and } n[c_1[x]] = t - 1$) \textbf{then} B-Tree-Merge($T, x, 1$)
4: $f \leftarrow \text{false}$
5: \textbf{while} ($x \neq \text{nil or } f = \text{false}$) \textbf{do} {
6: $q \leftarrow 1$
7: \textbf{while} ($q \leq n[x] \text{ and } key_q[x] \geq k$) \textbf{do}
8: $q \leftarrow q + 1$ ▷ Scan the keys
9: \textbf{if} ($q \leq n[x] \text{ and } key_q[x] = k$) \textbf{then}
10: $f \leftarrow \text{true}$ ▷ $k$ has been found
11: \textbf{else} {
12:}
if \( c_q[x] \neq \text{nil} \) and \( n[c_q[x]] = t - 1 \) then

If \( c_q[x] \) exists and is lean, then do the following

if \( q \neq n[x] + 1 \) then

If \( c_q[x] \) has the immediate right sibling

if \( n[c_{q+1}[x]] = t - 1 \) then

If the sibling is lean, merge it with \( c_q[x] \)

B-Tree-Merge\((T, x, q)\)

Otherwise, ship a key and a child from it

else B-Tree-GetFromRight\((T, x, q)\)

else

If \( c_q[x] \) is the rightmost child

If the immediate left sibling is lean

if \( n[c_{q-1}[x]] = t - 1 \) then

merge it with \( c_q[x] \) and fix \( q \)

\( q \leftarrow q - 1 \)

B-Tree-Merge\((T, x, q)\)

Otherwise, ship a key and a child from it

else B-Tree-GetFromLeft\((T, x, q)\)

\( x \leftarrow c_q[x] \)

}
34: \textbf{if} \ x = \text{nil} \textbf{ then return} “Key Not Found”
35: \textbf{if} \ c_q[x] = \text{nil} \textbf{ then} \\
36: \quad n[x] \leftarrow n[x] - 1 \\
37: \quad \textbf{for} \ i \leftarrow q \textbf{ to } n[x] - 1 \textbf{ do} \\
38: \quad \quad \text{key}_i[x] \leftarrow \text{key}_{i+1}[x] \\
39: \quad \textbf{return} \\
40: \quad y \leftarrow x \quad \quad \quad \quad \quad \quad \quad \text{▷ Now find the predecessor} \\
41: \quad r \leftarrow q \\
42: \quad \textbf{while} \ c_r[y] \neq \text{nil} \textbf{ do} \\
43: \quad \quad \textbf{if} \ n[c_r[y]] = t - 1 \textbf{ then} \\
44: \quad \quad \quad \textbf{if} \ r \neq n[y] + 1 \textbf{ then} \\
45: \quad \quad \quad \quad \textbf{if} \ n[c_{r+1}[y]] = t - 1 \textbf{ then} \\
46: \quad \quad \quad \quad \quad \text{B-Tree-Merge}(T, y, r) \\
47: \quad \quad \quad \quad \textbf{else} \text{ B-Tree-GetFromRight}(T, y, r) \\
48: \quad \quad \quad \textbf{else if} \ n[c_{r-1}[y]] = t - 1 \textbf{ then} \\
49: \quad \quad \quad \quad \quad r \leftarrow r - 1 \\
50: \quad \quad \quad \text{B-Tree-Merge}(T, y, r) \\
51: \quad \quad \textbf{else} \text{ B-Tree-GetFromLeft}(T, y, r) \\
52: \quad \quad y \leftarrow c_r[y] \\
53: \quad \quad r \leftarrow n[y] + 1 \\
54: \quad \textbf{return} \text{key}_q[x] \leftarrow \text{key}_r[y] \quad \quad \text{▷ Relocate the key} \\
55: \quad n[y] \leftarrow n[y] - 1
B-Tree-GetFromRight($T, x, i$)

1: $y \leftarrow c_i[x]$
2: $z \leftarrow c_{i+1}[x]$
3: $n[y] \leftarrow n[y] + 1$
4: $key_{n[y]}[y] \leftarrow key_i[x]$
5: $key_i[x] \leftarrow key_1[z]$
6: $c_{n[y]+1}[y] \leftarrow c_1[z]$
7: $p[c_1[x]] \leftarrow y$
8: $n[z] \leftarrow n[z] - 1$
9: for $i \leftarrow 1$ to $n[z]$ do $key_i[z] \leftarrow key_{i+1}[z]$
10: for $i \leftarrow 1$ to $n[z] + 1$ do $c_i[z] \leftarrow c_{i+1}[z]$
B-Tree-GetFromLeft$(T, x, i)$

1: $y \leftarrow c_i[x]$
2: $z \leftarrow c_{i-1}[x]$
3: for $i \leftarrow n[y]$ downto 1 do
4: \hspace{1em} $key_{i+1}[z] \leftarrow key_i[z]$
5: for $i \leftarrow n[y] + 1$ downto 1 do
6: \hspace{1em} $c_{i+1}[z] \leftarrow c_i[z]$
7: $n[y] \leftarrow n[y] + 1$
8: $key_1[y] \leftarrow key_{i-1}[x]$
9: $key_{i-1}[x] \leftarrow key_{n[z][z]}$
10: $c_1[y] \leftarrow c_{n[z]+1}[z]$
11: $p[c_{n[z]+1}[z]] \leftarrow y$
12: $n[z] \leftarrow n[z] - 1$
the initial tree

G deleted

A deleted

E deleted

O deleted