A (B)tree that's balanced?
Keep thy leaves at the same depth
Across the tree's breadth

As you insert, watch the tree grow
Spreading and branching
Out wide

Though sometimes in height
But only when the root splits
And that too, by one

Frequently asked questions from the previous class survey

- Can poor indexing impact the performance of databases?
- Does a large fanout in B-Trees make searching slower?
- Can B-trees handle duplicates?
Topics covered in this lecture

- B-Trees
- B+Trees
- B*Trees

B-TREE LOOKUPS
Separator keys splitting a tree into subtrees

K<sub>s</sub> < K<sub>1</sub>
K<sub>1</sub> ≤ K<sub>s</sub> < K<sub>2</sub>
K<sub>2</sub> ≤ K<sub>s</sub> < K<sub>3</sub>
K<sub>s</sub> ≥ K<sub>3</sub>

Separator keys splitting a tree into subtrees

K<sub>s</sub> < 12
12 ≤ K<sub>s</sub> < 31
31 ≤ K<sub>s</sub> < 45
K<sub>s</sub> ≥ 45
B-tree Lookup Algorithm:

- To find an item in a B-Tree, we perform a single traversal from root to leaf.
- The objective of this search is to find the key or its predecessor.
  - Finding an exact match is used for point queries, updates, and deletions.
  - Finding its predecessor is useful for range scans and inserts.
B-tree Lookup Algorithm: [2/4]

- Index keys split the tree into **subtrees**
  - With boundaries between two neighboring keys
- The algorithm starts from the root and performs a binary search
  - This locates a subtree
- As soon as we find the subtree?
  - Follow the pointer that corresponds to it
  - Repeat search

---

B-tree Lookup Algorithm: [3/4]

- On each level, we get a more detailed view of the tree:
  - We start on the most coarse-grained level (the root of the tree)
  - Descend to the next level where keys represent more precise, detailed ranges
  - Until we finally reach **leaves**, where the data records are located
B-tree Lookup Algorithm:

- During the point query
  - Search completes after finding (or failing to find) the target key
- During the range scan
  - Sibling pointers are followed until the end of the range is reached or the range predicate is exhausted

B-Tree: Example: Looking for 53
B-Tree: Example: Looking for 20

\[
\begin{array}{c}
30 \\
1 \\
9 \\
20 \\
25 \\
4 \\
6 \\
7 \\
9 \\
17 \\
20 \\
24 \\
25 \\
28 \\
29 \\
30 \\
33 \\
37 \\
38 \\
40 \\
53 \\
55 \\
\end{array}
\]

B-Tree: Example: Looking for 20 < x < 53

\[
\begin{array}{c}
30 \\
1 \\
9 \\
20 \\
25 \\
4 \\
6 \\
7 \\
9 \\
17 \\
20 \\
24 \\
25 \\
28 \\
29 \\
30 \\
33 \\
37 \\
38 \\
40 \\
53 \\
55 \\
\end{array}
\]
I shall be telling this with a sigh
Somewhere ages and ages hence:
Two roads diverged in a wood, and I—
I took the one less traveled by,
And that has made all the difference.
The Road Not Taken, Robert Frost

To insert the value into a B-Tree?

☐ First locate the target and find the insertion point

☐ If the target node doesn’t have enough room available?
  ☐ We say that the node has overflowed
    ☑ Should be split in two to fit the new data
More precisely, the node is split if the following conditions hold

- For leaf nodes:
  - If the node can hold up to $N$ key-value pairs?
    - Inserting one more key-value pair brings it over its maximum capacity $N$

- For nonleaf nodes:
  - If the node can hold up to $N + 1$ pointers?
    - Inserting one more pointer brings it over its maximum capacity $N + 1$ pointers

Mechanics of splits

- Splits are done by allocating a new sibling node
  - Transferring half the elements from the splitting node to the new sibling node
  - Adding sibling node's first key and pointer to the parent node
  - We say that the key is promoted

- The index at which the split is performed is called the split point
  - All elements after the split point (including split point in the case of leaf node split) are transferred to the newly created sibling node
  - The rest of the elements remain in the splitting node
Mechanics of splits

- If the parent node is full?
  - The parent has to be split as well
  - This operation **might propagate recursively** all the way to the root

Node splits are done in **four steps**

- Allocate a **new sibling node**
- **Transfer** half the elements from **splitting node** to the **new sibling node**
- **Place the new element** into either the new sibling or the splitting node
- At the parent of the split node?
  - Add a separator key and a pointer to the new sibling node
B-Tree: Leaf node Splits during insertion of 11

Before Split

\[ \begin{array}{c}
10 \\
13 \\
15 \\
\end{array} \]

After Split

\[ \begin{array}{c}
10 \\
11 \\
13 \\
15 \\
\end{array} \]

Once the new sibling node is created, we use separator key invariants to decide where to insert the new element.

B-Tree: Nonleaf node splits during insertion of 11

Before Split

\[ \begin{array}{c}
8 \\
10 \\
13 \\
16 \\
\end{array} \]

After Split

\[ \begin{array}{c}
8 \\
10 \\
11 \\
13 \\
15 \\
\end{array} \]

\[ \begin{array}{c}
16 \\
\end{array} \]
Summarizing insertions

- We first proceed down the tree, searching for the position to insert the new key.
- We return back up the tree, splitting nodes that have become overfull.

Insertions and why B-trees are balanced

- Each split increases the branching factor of a node, but not necessarily the height.
- In fact, the only time we increase the height of the tree is when we split the root node itself.
- Because we only increase the height by splitting the root node (adding a depth of 1 to every leaf simultaneously).
  - We can guarantee that the tree always remains balanced.
Merges

- **Deletions** are done by first locating the target leaf
  - When the leaf is located?
    - The key and the value associated with it are removed
  - If neighboring nodes have too few values (i.e., their occupancy falls under a threshold)?
    - This situation is called an *underflow*
    - Sibling nodes are *merged*
Merges: If two adjacent nodes have a common parent

- And their contents fit into a single node?
  - Their contents should be merged (concatenated)

- If their contents do not fit into a single node?
  - Keys are redistributed between them to restore balance

More precisely, two nodes are merged if the following conditions hold

- For leaf nodes
  - If a node can hold up to N key-value pairs, and a combined number of key-value pairs in two neighboring nodes is less than or equal to N
    - We move elements from one of the siblings to the other one

- For nonleaf nodes:
  - If a node can hold up to N + 1 pointers, and a combined number of pointers in two neighboring nodes is less than or equal to N + 1
B-Tree: Leaf node merge during the deletion of 16

Before Merge

<table>
<thead>
<tr>
<th>11</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>

After Merge

<table>
<thead>
<tr>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>25</td>
</tr>
</tbody>
</table>

Generally, elements from the right sibling are moved to the left one, but it can be done the other way around as long as key order is preserved.

Node merges are done in three steps, assuming the element is already removed

- Copy all elements from the right node to the left one
- Remove the right node pointer from the parent (or demote it in the case of a nonleaf merge)
- Remove the right node
B-Tree: Nonleaf node merge during the deletion of 10

Before Merge

After Merge

Rebalancing
Rebalancing

- To improve space utilization, *instead of splitting* the node on overflow
  - We can *transfer* some of the elements to one of the sibling nodes and make space for the insertion

- Similarly, during delete, instead of merging the sibling nodes
  - We may choose to move some of the elements from the neighboring nodes to ensure the node is at least half full

Tree balancing: Distributing elements between the more occupied node to the less occupied one

**Before Balancing**

<table>
<thead>
<tr>
<th>10</th>
<th>18</th>
</tr>
</thead>
</table>
| 10 | 13 | 16
| 18 |

**After Balancing**

<table>
<thead>
<tr>
<th>10</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>16</td>
<td>18</td>
</tr>
</tbody>
</table>
B+Tree

- Internal nodes do not store any pointers to records
  - All pointers to records are stored in the leaf nodes
  - Each node, consequently, can hold more keys
    - Causing the tree to be shallower, and thus, faster to search

- Leaves form a linked list
  - A leaf node includes a pointer to the next leaf node to speed sequential access
B*tree

- Focuses on keeping internal nodes more densely packed
- Attempts to keep internal nodes 2/3 full
  - Instead of 1/2
- Most expensive component of inserting a node in B-tree is splitting the node
  - B*tree try to postpone splitting operation as long as they can

B*tree: Postponing the splitting operation

- Instead of immediately splitting up a node when it gets full,
  - Keys are shared with a node next to it
- Spill operation is less expensive than a split because it:
  - Requires only shifting the keys between existing nodes
  - Does not entail allocating memory for a new one
- When both sibling nodes are full?
  - The two sibling nodes are split into three (2/3<sup>rd</sup> occupancy!)
  - One more key is shifted up the tree, to the parent node
Any other variants?

- Yes, the B*+Tree
- The B*+Tree combines features of the B+Tree and the B*Tree

The contents of this slide-set are based on the following references