Three Reasons for using B-trees

(1) storage-friendly:
   B-trees work well on any layer of the storage hierarchy

(2) good trade-off random/sequential:
   seeks/cache misses once for each node rather than once for each pivot step

(3) universal applicability:
   You have a fancy indexing problem?
   Did you try a B-tree (or a variant of it)?
   Did not work? Did you try really hard?
Intuition for B-Tree Indexes
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>Z</td>
<td>T</td>
<td>K</td>
</tr>
<tr>
<td>34</td>
<td>56</td>
<td>67</td>
<td>72</td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>V</td>
<td>B</td>
<td>D</td>
</tr>
<tr>
<td>83</td>
<td>84</td>
<td>87</td>
<td>105</td>
</tr>
</tbody>
</table>
Create Descriptor...

Node

Leaf
Redistribute...
Fix Descriptors...

![Diagram with numbers and letters]

11 15 34 56
M T S Z

67 72 73 83
T K A O

84 87 102 105
V B N D

$\mathcal{C}_{60;84}$
B-Tree Node and Leaf Sizes

<table>
<thead>
<tr>
<th></th>
<th>if not root:</th>
<th>if root:</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Nodes</strong></td>
<td>$n \in [k;2k]$ keys</td>
<td>$n \in [1;2k]$ keys</td>
</tr>
<tr>
<td></td>
<td>$\Rightarrow n+1$ children</td>
<td>$\Rightarrow n+1$ children</td>
</tr>
<tr>
<td>$4k &lt; 13$</td>
<td></td>
<td>$k = \frac{4}{3}$</td>
</tr>
<tr>
<td><strong>Leaves</strong></td>
<td>$[k^<em>;2k^</em>]$ key/value-pairs</td>
<td>$[1;2k^*]$ key/value-pairs</td>
</tr>
<tr>
<td>$k^* = \frac{2}{1}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B-Tree Properties

path from root to any leaf: same length

k and k* implicitly defined by data types and node size

no values in index nodes

all data sits in leaf nodes
B-Tree Properties

path from root to any leaf: same length

k and k* implicitly defined by data types and node size

no values in index nodes

all data sits in leaf nodes

(data inside nodes and leaves sorted by key)
B-Trees and Interval Partitioning

\[
\begin{align*}
\omega & = 60 \\
[0; 60] & \rightarrow [0; 123] \\
[60; 123] & \rightarrow 63, 103 \\
& \rightarrow [123; +\infty[ \\
& \rightarrow 137, 443
\end{align*}
\]
```yaml
<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>15</td>
<td>34</td>
<td>56</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67</td>
<td>72</td>
<td>73</td>
<td>83</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>84</td>
<td>87</td>
<td>102</td>
<td>105</td>
<td></td>
</tr>
</tbody>
</table>
```

**find_key(67)**

```
CloseSubtree(key <)
```

```
\[ g \]
```
find_range([34;72])
Index Sequential Access Method (ISAM)
find_range([34;72])

1. Point query
2. Scan along leaves