This sort of query can be answered efficiently with an index.

```sql
CREATE INDEX ind1 ON movies (year)
```

- An index contains a set of (key,value) pairs, ordered by key.
- It should support efficient search, as well as efficient insert / delete.
Crunch-time Charlie (quick and dirty, too harried to learn)

Timely Terry (no sweat, plans ahead)

Fastidious Frances (everything pristine all of the time)

Q. Can we design a roughly-balanced search tree, but without being obsessive about it?

FREE-FORM BINARY SEARCH TREE

BALANCED BINARY SEARCH TREE

N
O
I
S
D
P

search is fast, insert/delete slow

search can be slow, insert/delete fast
Genius idea: let’s keep the depth perfectly balanced, but let the nodes have a variable number of children.

E.g. let’s require that each non-leaf node have 2, 3, or 4 children.

To fit the standard BST design, let’s store 1, 2, or 3 items at each node, \( \text{#children} = \text{#items} + 1 \)
Genius idea: let’s keep the depth perfectly balanced, but let the nodes have a variable number of children.

E.g. let’s require that each non-leaf node have 2, 3, or 4 children.
To fit the standard BST design, let’s store 1, 2, or 3 items at each node, $\#\text{children} = \#\text{items} + 1$

Q1. Is this balanced enough to give $O(\log n)$ search, where $n$ is the number of (key,value) pairs?
Q2. Is this flexible enough that we can do insert/delete in $O(\log n)$, while maintaining the rough balance?
A B-tree is a perfectly height-balanced search tree, where each node has \( \#\text{keys} \in \{k_{\text{min}}, \ldots, k_{\text{max}}\} \) (apart perhaps from the root, which may have fewer).

For a node with \( m \) (key,value) pairs,

- There are \( m + 1 \) child subtrees (unless it’s a leaf)
- All keys \( \ell \) in child \( c_i \) satisfy \( k_{i-1} < \ell < k_i \) (with appropriate adjustment at \( i = 0 \) and \( i = m \))

What’s the smallest possible \( \#\text{keys} \) in a tree of height \( h \)?

- \( \text{root, depth 0: allowed to have 1 key} \) by (4)
- \( \text{depth 1: 2 nodes, each with } k_{\text{min}} \text{ keys} \)
- \( \text{depth 2: } 2 \times (k_{\text{min}} + 1)^2 \text{ nodes, each with } k_{\text{min}} \text{ keys} \)
- \( \text{depth 3: } 2 \times (k_{\text{min}} + 1)^3 \text{ nodes, each with } k_{\text{min}} \text{ keys} \)
- \( \cdots \)
- \( \text{depth } h: 2 \times (k_{\text{min}} + 1)^{h-1} \text{ nodes, each with } k_{\text{min}} \text{ keys} \)

So, for an arbitrary tree, \( \#\text{keys} \geq 1 + 2k_{\text{min}}(1 + (k_{\text{min}} + 1)^1 + \ldots + (k_{\text{min}} + 1)^{h-1}) = 2(k_{\text{min}} + 1)^h - 1 \)
QUESTION. For a tree with \( n \) keys in total, what’s the largest possible height?

We’ve just argued that \( n \gg 2^{(k_{\text{min}}+1)^h} - 1 \). Therefore \( h \leq \log_{k_{\text{min}}+1} \left( \frac{n+1}{2} \right) \). So \( h \) is \( O(\log n) \) for any \( k_{\text{min}} > 1 \).

If we didn’t bound \#keys, we might have a node with \( \Omega(n) \) keys. It’d take time \( \Omega(n) \) to scan through them, slowing down search & insert/delete. For any finite bound \( k_{\text{max}} < \infty \), we ensure that the work per node is \( O(1) \).

Putting these together, search is \( O(\log n) \).
SECTIONS 4.4 & 4.6

2-3-4 trees and B-trees
Insert \((k, v)\) into a tree with \(k_{\text{min}} = 1\) and \(k_{\text{max}} = 3\)

**Easy Case**
Simple insertion into a leaf

**Harder Case**
Insertion into a full leaf

**Question.** Where do we insert key \(A\)? And how?
The destination leaf is full, so split it by promoting its median key.
The destination leaf is full, so split it by promoting its median key.

To keep our tree balanced, excess keys need to be pushed *up*. 
**HARDEST CASE**

**insertion into a full leaf on a full path**

\[
\text{insert}(k,v) \text{ into a tree with } k_{\min} = 1 \text{ and } k_{\max} = 3
\]

**QUESTION. How do we insert key R?**

**NOTE.** This code is suitable for the general case. It may unnecessarily split some nodes at the top, but who cares?

```python
def insert(k, v):
    if root node is full:
        split it, and create new root
    \(x \leftarrow\) root node
    while \(x\) is not a leaf:
        # assert \(x\) is not full
        scan \(x\) to find which child \(y\) we want \(k\) in
        if \(y\) is not full:
            \(x \leftarrow y\)
        else:
            split \(y\) into \(y_1\) and \(y_2\) and promote a key
            \(x \leftarrow y_1\) or \(y_2\) as appropriate
    \(\text{insert } (k,v) \text{ into } x\)
```
QUESTION. Does this splitting operation constrain $k_{\text{min}}$ and $k_{\text{max}}$?

For this split to result in legal nodes, we need $k_{\text{max}} - 1 \geq 2k_{\text{min}}$. 
To keep our tree balanced, excess keys need to be pushed *up*.

From time to time, we may have to add a new node *at the top*. The tree becomes higher, but it remains perfectly height-balanced.
delete($k$) from a tree with $k_{\text{min}} = 1$ and $k_{\text{max}} = 3$

**EASY CASE**

Simple deletion from a leaf

A

B

D

G

L

M

Simple delete

**HARDER CASE**

Deletion from a bare-bones leaf (i.e., one with only $k_{\text{min}}$ keys)

We can "fatten" the leaf by stealing a key from a sibling, and "rotating".

A

B

E

I

D

G

L

M

A

D

E

I

B

G

L

M

A

D

I

simple delete

simple delete
delete($k$) from a tree with $k_{\text{min}} = 1$ and $k_{\text{max}} = 3$

**HARDER CASE**
deletion from a bare-bones leaf, with bare-bones siblings

Merge with a sibling, by stealing a key from the parent.
QUESTION. Does this merging operation constrain $k_{\text{min}}$ and $k_{\text{max}}$?

For this merging to result in a legal node, we need $2k_{\text{min}} + 1 \leq k_{\text{max}}$.

(This is exactly the same inequality we saw when we checked splitting.)
delete($k$) from a tree with $k_{\text{min}} = 1$ and $k_{\text{max}} = 3$

**Hardest Case**
deletion from a bare-bones leaf on a bare-bones path

```
def delete($k$):
    x ← root node
    while x is not a leaf:
        # assert x has $> k_{\text{min}}$ keys (or is root)
        if x has key $k$:
            ...
            # See trick from last lecture on how to delete keys from internal nodes.
        else:
            scan x to find the child $y$ that has $k$
            if $y$ has $k_{\text{min}}$ keys:
                either rotate or merge to make $y$ fatter
                $x ← y$
            delete $k$ from $x
```
To keep our tree balanced, deletions *suck in* keys from beside or above.

From time to time, we may suck down the root when merging its children. The tree becomes shorter, and it remains perfectly height-balanced,
How should we choose $k_{\text{min}}$ and $k_{\text{max}}$?

- Height $= O(\log n)$ as long as $k_{\text{min}} \geq 1$
- The work at each node is $O(1)$ as long as $k_{\text{max}} < \infty$
- We need $k_{\text{max}} \geq 2k_{\text{min}} + 1$ for merging/splitting to work

Are there any other considerations that can guide us to specific choices?
Rethinking B-tree block sizes on SSDs

One of the first questions to answer when running databases on SSDs is what B-tree block size to use. There are a number of factors that affect this decision:

- The type of workload
- I/O time to read and write the block size
- The size of the cache

That's a lot of variables to consider. For this blog post we assume a fairly common OLTP scenario - a database that's dominated by random point queries. We will also sidestep some of the more subtle caching effects by treating the caching algorithm as perfectly optimal, and assuming the cost of lookup in RAM is insignificant.

Experimenting with different node sizes, they found that max-size = 4kB gave best performance (for a database stored on SSD).

<table>
<thead>
<tr>
<th>Block size</th>
<th>IOPS</th>
<th>IOPS</th>
<th>IOPS</th>
<th>IOPS</th>
<th>IOPS</th>
<th>IOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1kB (32)</td>
<td>5.98 hops</td>
<td>4.98 hops</td>
<td>4.27 hops</td>
<td>3.74 hops</td>
<td>3.32 hops</td>
<td>2.98 hops</td>
</tr>
<tr>
<td>2kB (64)</td>
<td>765 q/sec</td>
<td>854 q/sec</td>
<td>885 q/sec</td>
<td>854 q/sec</td>
<td>668 q/sec</td>
<td>593 q/sec</td>
</tr>
<tr>
<td>4kB (128)</td>
<td>4579 keys</td>
<td>4254 keys</td>
<td>3780 keys</td>
<td>3197 keys</td>
<td>2186 keys</td>
<td>1789 keys</td>
</tr>
<tr>
<td>8kB (256)</td>
<td>16kB (512)</td>
<td>32kB (1024)</td>
<td>64kB (2048)</td>
<td>1334</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

So, if we have no cache the optimal block size is 4kB.
If we’re storing our index on an SSD:

- An SSD consists of many *blocks*, each made of many *pages*
- We read and write an entire page at a time
- Reading and writing to an SSD is very slow, compared to main memory access

⇒ Choose $k_{\text{max}}$ so that a node takes up an entire page, and choose $k_{\text{min}}$ as large as possible, i.e. $k_{\text{min}} = (k_{\text{max}} - 1)/2$, to keep pages full

(This explains the experimental finding from RethinkDb, that 4kB nodes are best.)

This is called a B-tree.
If we’re storing our index in main memory ... we’ll make a different choice of $k_{\text{min}}$ and $k_{\text{max}}$. 