Motivation

Old storage organization:

CPU → RAM → disk slow

Modern storage organization:

CPU → cache → RAM slow

Data stored in the slow device. But read/write is per block.
Want to read/write fewer blocks.
Want to take full advantage of each block.

B-Trees, e.g., (2,4)-Trees

Say, $B$ bytes can fit $d$ child pointers and $d - 1$ keys. Each (internal) node has $c$ children and $c - 1$ keys, $[d/2] \leq c \leq d$.
Keys are in increasing order: $k_1 < k_2 < \cdots < k_{c-1}$.

$k_{c-1} < (\text{keys in child subtree } T_i) < k_i$

All leaves (null, external nodes) are at the same depth.
(This example uses capital letters for keys, and $d = 4$.)

\[ \begin{array}{c}
H, P \\
B, C, E \\
J, L \\
S
\end{array} \]

\[ \begin{array}{c}
k_1, k_2, k_3, k_4 \\
v_1, v_2, v_3, v_4, v_5
\end{array} \rightarrow \begin{array}{c}
k_1 \\
v_1, v_2, v_3, v_4, v_5
\end{array} \]

\[ \begin{array}{c}
k_1, k_2, k_3, k_4 \\
v_1, v_2, v_3, v_4, v_5
\end{array} \rightarrow \begin{array}{c}
k_1, k_2 \\
v_1, v_2, v_3, v_4, v_5
\end{array} \]

This may cause the parent node to overflow—repeat.
There may be no parent—create new root.

\[ \begin{array}{c}
\ldots, k_p, \ldots \\
v_1', v_2', v_3', v_1
\end{array} \rightarrow \begin{array}{c}
\ldots, 30, \ldots \\
v_1', v_2', v_3', v_1
\end{array} \]

“Transfer”: steal from rich sibling, but go through parent.

find($k$)

Starting with the root:
Search for $k$ in a node. If $k$ is found, done.
If $k$ is not found, you still know which “gap” it would be ⇒ which child to pursue.

insert($k$, $x$)

Begin like find($k$). Should be not-found, but you know which bottom node you end at.
Insert ($k$, $x$) there. Insert new leaf.
If overflow, split node and promote a middle key to the parent:

\[ \begin{array}{c}
k_1, k_2, k_3, k_4 \\
v_1, v_2, v_3, v_4, v_5
\end{array} \rightarrow \begin{array}{c}
k_1 \\
v_1, v_2, v_3, v_4, v_5
\end{array} \]

This may cause the parent node to overflow—repeat.
There may be no parent—create new root.

remove($k$): part 1

Begin like find($k$). Should be found.
If not bottom node: replace by an appropriate key in a bottom node.

\[ \begin{array}{c}
\ldots, k_1, \ldots \\
v_i \\
40, 50
\end{array} \rightarrow \begin{array}{c}
\ldots, k_1, \ldots \\
v_i
\end{array} \]

E.g., replace $k_1$ by $50$.
Remove chosen key and a leaf at chosen bottom node.
If underflow and has parent…

remove($k$): part 2a

Underflow case (a): an immediate sibling has 2 or 3 keys.

\[ \begin{array}{c}
\ldots, k_p, \ldots \\
20, 30 \\
v_1', v_2', v_3', v_1
\end{array} \rightarrow \begin{array}{c}
\ldots, 30, \ldots \\
20
\end{array} \]

“Transfer”: steal from rich sibling, but go through parent.
remove($k$): part 2b

Underflow case (b): otherwise (both siblings have too few keys).

```
          k1, k2
         /   \
        20   
   v'_1  v'_2  v_1   \rightarrow  
          k1  20, k2
              /   \
         v'_1  v'_2  v_1
```

"Fusion": steal a key from parent, merge with a sibling.
May cause parent to underflow—repeat fixing underflow.

B-Tree Height

$n$ keys $\Rightarrow$ $n + 1$ leaves. (Structural induction on subtrees.)

At most $d$ children per node
$\Rightarrow n + 1 \leq d^{\text{height}}$
$\Rightarrow \log_d(n + 1) \leq \text{height}$

At least $d/2$ children per node, and all leaves have the same depth
$\Rightarrow n + 1 \geq (d/2)^{\text{height}}$
$\Rightarrow \log_{d/2}(n + 1) \geq \text{height}$
$\Rightarrow \log_d(n + 1)/(1 - \log_d(2)) \geq \text{height}$

$\text{height} \in \Theta(\log_d(n))$
If $d = B/r$, then $\log_d(n) = \log_B(n)/(1 - \log_B(r))$.
$\text{height} \in \Theta(\log_B(n)) = \Theta(\log_2(n)/\log_2(B))$
and the hidden constant factor is just slightly bigger than 1.

B-Tree Time Costs

find, insert, remove each reads/writes $\Theta(\log_B(n))$ nodes.

We choose $d$ so that one node is in one block.
find, insert, remove each reads/writes $\Theta(\log_B(n))$ blocks.

This is better than binary search trees, which may read/write $\Theta(\log_2(n))$ blocks because different keys happen to scatter over different blocks.

Cache-Oblivious Data Structures

Block size $B$ varies with computers.
In fact, even varies within the same computer:

```
CPU -- cache  RAM
    
```

In recent years, cache-oblivious data structures are invented to do well with caching, without knowing $B$.

Cache-Oblivious Lookahead Array: Basic

Basic version (no lookahead):

- Array $#i$ has length $2^i$, either empty or full.
- If you have $n$ keys, express $n$ in binary.
- Array $#i$ is full iff bit $#i$ is on.
- Arrays are sorted (can do binary search).

Example: $n = 1011_2$

#0: 24
#1: 67, 81
#2:
#3: 13, 36, 48, 51, 57, 63, 72, 89

Insert

$n = 1011_2$

#0: 24
#1: 67, 81
#2: 75
#3: 13, 36, 48, 51, 57, 63, 72, 89

Insert 75.

$n = 1100_2$

#0:
#1:
#2: 24, 67, 75, 81
#3: 13, 36, 48, 51, 57, 63, 72, 89

Change in $n$’s binary form says which arrays to merge and where they go.
Amortized Time of Insert

We count the number of block accesses.

Over \( n \) inserts leading to \( n \) keys:

- Every time a merge merges \( s \) keys: \( O(s/B) \) accesses per merge.
- Amortize that over those \( s \) keys: \( O(1/B) \) accesses per key per merge.
- A key is involved in \( O(\log_2(n)) \) merges only.

Amortized \( O(\log_2(n)/B) \) accesses per key (or per insert).

This is even better than B-trees' \( O(\log_2(n)/\log_2(B)) \).

Find

Basic version (no lookahead): Find is not as rosy.

For each array, do a binary search.

\( \Theta((\log_2(n))^2) \) in the worst case.

The real version adds redundant data to fix this.

Cache-Oblivious Lookahead Array

Real version (has lookahead):

In array \( k \geq 2 \): Sacrifice half of its cells to:

- duplicate 1/8 of the content from array \( k+1 \), with pointers to where they are in \( k+1 \).
  call these pointers "real lookahead pointers"
- every 4th cell is two pointers: to the duplicate on the left, to the duplicate on the right
  call these pointers "duplicate lookahead pointers"

Find

With lookahead, examine at most 8 consecutive cells per array:

For the smaller arrays: obvious.

For the larger arrays: If not found in array \( k \):

- a nearby duplicate lookahead pointer tells you a duplicate on the left and a duplicate on the right
- their real lookahead pointers bring you to a segment in \( k+1 \)
- only need to examine that segment
- the segment has only 8 cells

Find takes \( O(\log_2(n)) \) steps or block accesses.

One Last Thing

That is still not the real real version.

The real real version is more sophisticated to reduce worst-case cost of insert (from \( O(n/B) \) to \( O(\log_2(n)) \)).

There are other cache-oblivious data structures with different strengths and weaknesses. There are even a few versions of cache-oblivious B-trees.