CSE 332: Data Abstractions

Lecture 9: B Trees

Ruth Anderson
Spring 2014
Announcements

• **Project 2** –
  Partner selection due by Fri *at the latest*.

• **Homework 3** – due next Wednesday
Today

- Dictionaries
  - B-Trees
Our goal

- **Problem**: A dictionary with so much data most of it is on disk

- **Desire**: A balanced tree (logarithmic height) that is even shallower than AVL trees so that we can minimize disk accesses and exploit disk-block size

- **A key idea**: Increase the branching factor of our tree
M-ary Search Tree

- Build some sort of search tree with branching factor $M$:
  - Have an array of sorted children (Node[])
  - Choose $M$ to fit snugly into a disk block (1 access for array)

Perfect tree of height $h$ has $(M^{h+1}-1)/(M-1)$ nodes (textbook, page 4)

What is the height of this tree?
What is the worst case running time of find?
M-ary Search Tree

- # hops for `find`?
  - If we have a balanced M-ary tree:
  - Approx. $\log_M n$ hops instead of $\log_2 n$ (for balanced BST)
  - Example: $M = 256 (=2^8)$ and $n = 2^{40}$ that’s 5 hops instead of 40 hops
- Sounds good, but how do we decide which branch to take?
  - Binary tree: Less than/greater than node value?
  - M-ary: In range 1? In range 2? In range 3?... In range M?
- Runtime of `find` if balanced: $O(\log_2 M \log_M n)$
  - $\log_M n$ is the height we traverse.
  - $\log_2 M$: At each step, find the correct child branch to take using binary search among the M options!
Questions about M-ary search trees

• What should the order property be?
• How would you rebalance (ideally without more disk accesses)?
• Storing real data at inner-nodes (like we do in a BST) seems kind of wasteful…
  – To access the node, will have to load the data from disk, even though most of the time we won’t use it!!
  – Usually we are just “passing through” a node on the way to the value we are actually looking for.

So let’s use the branching-factor idea, but for a different kind of balanced tree:
  – Not a binary search tree
  – But still logarithmic height for any $M > 2$
B+ Trees (we and the book say “B Trees”)

- Two types of nodes: **internal nodes** & **leaves**
- Each **internal node** has room for up to $M-1$ keys and $M$ children
  - No other data; **all data at the leaves!**
- **Order property:**
  Subtree **between** keys $a$ and $b$
  contains only data that is $\geq a$ and $< b$ (notice the $\geq$)
- **Leaf** nodes have up to $L$ sorted data items
- As usual, we’ll ignore the “along for the ride” data in our examples
  - Remember no data at non-leaves
Find

• Different from BST in that we *don’t store data at internal nodes*

• But *find* is still an easy root-to-leaf recursive algorithm
  – At each internal node do binary search on (up to) M-1 keys to find the branch to take
  – At the leaf do binary search on the (up to) L data items

• But to get logarithmic running time, we need a balance condition…
Structure Properties

• **Root** (special case)
  – If tree has \( \leq L \) items, root is a leaf (occurs when starting up, otherwise unusual)
  – Else has between 2 and \( M \) children

• **Internal nodes**
  – Have between \( \lceil M/2 \rceil \) and \( M \) children, i.e., at least half full

• **Leaf nodes**
  – All leaves at the same depth
  – Have between \( \lceil L/2 \rceil \) and \( L \) data items, i.e., at least half full

Any \( M > 2 \) and \( L \) will work, but:
  We pick \( M \) and \( L \) **based on disk-block size**
Example

Suppose \( M = 4 \) (max # pointers in internal node) and \( L = 5 \) (max # data items at leaf)

- All internal nodes have at least 2 children
- All leaves have at least 3 data items (only showing keys)
- All leaves at same depth

Note on notation: Inner nodes drawn horizontally, leaves vertically to distinguish. Include empty cells
Balanced enough

Not hard to show height $h$ is logarithmic in number of data items $n$

• Let $M > 2$ (if $M = 2$, then a list tree is legal – no good!)

• Because all nodes are at least half full (except root may have only 2 children) and all leaves are at the same level, the minimum number of data items $n$ for a height $h>0$ tree is…

$$n \geq 2 \left\lceil \frac{M}{2} \right\rceil^{h-1} \left\lceil \frac{L}{2} \right\rceil$$

minimum number of leaves  minimum data per leaf

4/18/2014
Example: B-Tree vs. AVL Tree

Suppose we have 100,000,000 items

- Maximum height of AVL tree?

- Maximum height of B tree with $M=128$ and $L=64$?
Example: B-Tree vs. AVL Tree

Suppose we have 100,000,000 items

• **Maximum height of AVL tree?**
  – Recall \( S(h) = 1 + S(h-1) + S(h-2) \)
  – lecture7.xlsx reports: **37**

• **Maximum height of B tree** with \( M=128 \) and \( L=64 \)?
  – Recall \( (2 \left\lceil \frac{M}{2} \right\rceil^{h-1}) \left\lfloor \frac{L}{2} \right\rfloor \)
  – lecture9.xlsx reports: **5** (and 4 is more likely)
  – Also not difficult to compute via algebra
Disk Friendliness

What makes B trees so disk friendly?

- Many keys stored in one **internal node**
  - All brought into memory in one disk access
    - *IF* we pick \( M \) wisely
      - Makes the binary search over \( M-1 \) keys totally worth it (insignificant compared to disk access times)

- **Internal nodes** contain only keys
  - Any **find** wants only one data item; wasteful to load unnecessary items with internal nodes
  - So only bring one **leaf** of data items into memory
  - Data-item size doesn’t affect what \( M \) is
Maintaining balance

• So this seems like a great data structure (and it is)

• But we haven’t implemented the other dictionary operations yet
  – insert
  – delete

• As with AVL trees, the hard part is maintaining structure properties
  – Example: for insert, there might not be room at the correct leaf
Building a B-Tree (insertions)

The empty B-Tree (the root will be a leaf at the beginning)

\[ M = 3 \quad L = 3 \]

Just need to keep data in order
When we ‘overflow’ a leaf, we split it into 2 leaves
Parent gains another child
If there is no parent (like here), we create one; how do we pick the key shown in it?
   • Smallest element in right tree
Split leaf again

$M = 3 \quad L = 3$
Split the internal node (in this case, the root)

\[ M = 3 \quad \text{and} \quad L = 3 \]

What now?
Note: Given the leaves and the structure of the tree, we can always fill in internal node keys; ‘the smallest value in my right branch’
Insertion Algorithm

1. Insert the data in its leaf in sorted order

2. If the leaf now has \( L+1 \) items, overflow!
   - Split the leaf into two nodes:
     - Original leaf with \( \lceil (L+1) / 2 \rceil \) smaller items
     - New leaf with \( \lfloor (L+1) / 2 \rfloor = \lceil L / 2 \rceil \) larger items
   - Attach the new child to the parent
     - Adding new key to parent in sorted order

3. If step (2) caused the parent to have \( M+1 \) children, overflow!
   - ...
Insertion algorithm continued

3. If an internal node has $M+1$ children
   - Split the node into two nodes
     • Original node with $\lceil (M+1)/2 \rceil$ smaller items
     • New node with $\lfloor (M+1)/2 \rfloor = \lceil M/2 \rceil$ larger items
   - Attach the new child to the parent
     • Adding new key to parent in sorted order

Splitting at a node (step 3) could make the parent overflow too
   - So repeat step 3 up the tree until a node doesn’t overflow
   - If the root overflows, make a new root with two children
     • This is the only case that increases the tree height
Efficiency of insert

• Find correct leaf: $O(\log_2 M \log_M n)$
• Insert in leaf: $O(L)$
• Split leaf: $O(L)$
• Split parents all the way up to root: $O(M \log_M n)$

Total: $O(L + M \log_M n)$

But it’s not that bad:

– Splits are not that common (only required when a node is FULL, $M$ and $L$ are likely to be large, and after a split, will be half empty)
– Splitting the root is extremely rare
– Remember disk accesses were the name of the game: $O(\log_M n)$
B-Tree Reminder: Another dictionary

• Before we talk about deletion, just keep in mind overall idea:
  – Large data sets won’t fit entirely in memory
  – Disk access is slow
  – Set up tree so we do one disk access per node in tree
  – Then our goal is to keep tree shallow as possible
  – Balanced binary tree is a good start, but we can do better than \( \log_2 n \) height
  – In an M-ary tree, height drops to \( \log_M n \)
    • Why not set M really really high? Height 1 tree…
    • Instead, set M so that each node fits in a disk block
And Now for Deletion…

Delete(32)

Easy case: Leaf still has enough data; just remove

$M = 3 \quad L = 3$
Delete(15)

\[ M = 3 \quad L = 3 \]

4/18/2014

Is there a problem?
$M = 3 \quad L = 3$

Adopt from neighbor!
Delete(16)

$m = 3 \quad l = 3$

Is there a problem?
$M = 3 \quad L = 3$

But hey, Is there a problem?
\( M = 3 \quad L = 3 \)

Adopt from neighbor!

4/18/2014
Delete(14)

$M = 3$  $L = 3$
Delete(18)

\[ M = 3 \quad L = 3 \]

Is there a problem?

4/18/2014
Merge with neighbor!

But hey, Is there a problem?
$M = 3 \quad L = 3$

Merge with neighbor!

But hey, Is there a problem?
Pull out the root!

$M = 3 \quad L = 3$
Deletion Algorithm, part 1

1. Remove the data from its leaf

2. If the leaf now has \([I/2] - 1\), underflow!
   - If a neighbor has > \([I/2]\) items, adopt and update parent
   - Else merge node with neighbor
     • Guaranteed to have a legal number of items
     • Parent now has one less node

3. If step (2) caused the parent to have \([M/2] - 1\) children, underflow!
   - ...
Deletion algorithm (continued)

3. If an internal node has \(\lceil \frac{M}{2} \rceil - 1\) children
   – If a neighbor has > \(\lceil \frac{M}{2} \rceil\) items, adopt and update parent
   – Else merge node with neighbor
     • Guaranteed to have a legal number of items
     • Parent now has one less node, may need to continue up the tree

If we merge all the way up through the root, that’s fine unless the root went from 2 children to 1
   – In that case, delete the root and make child the root
   – This is the only case that decreases tree height
Worst-Case Efficiency of Delete

- Find correct leaf: $O(\log_2 M \log_M n)$
- Remove from leaf: $O(L)$
- Adopt from or merge with neighbor: $O(L)$
- Adopt or merge all the way up to root: $O(M \log_M n)$

Total: $O(L + M \log_M n)$

But it’s not that bad:
- Merges are not that common
- Disk accesses are the name of the game: $O(\log_M n)$
Insert vs delete comparison

Insert
• Find correct leaf: \(O(\log_2 M \log_M n)\)
• Insert in leaf: \(O(L)\)
• Split leaf: \(O(L)\)
• Split parents all the way up to root: \(O(M \log_M n)\)

Delete
• Find correct leaf: \(O(\log_2 M \log_M n)\)
• Remove from leaf: \(O(L)\)
• Adopt/merge from/with neighbor leaf: \(O(L)\)
• Adopt or merge all the way up to root: \(O(M \log_M n)\)
**B Trees in Java?**

For most of our data structures, we have encouraged writing high-level, reusable code, such as in Java with generics.

It is worthwhile to know enough about “how Java works” to understand why this is probably a bad idea for B trees.

- If you just want a balanced tree with worst-case logarithmic operations, no problem
  - If $M=3$, this is called a 2-3 tree
  - If $M=4$, this is called a 2-3-4 tree
- Assuming our goal is efficient number of disk accesses
  - Java has many advantages, but it wasn’t designed for this

The key issue is extra *levels of indirection*...
Naïve approach in Java

Even if we assume data items have `int` keys, you cannot get the data representation you want for “really big data”

```java
interface Keyed {
    int getKey();
}
class BTreeNode<E implements Keyed> {
    static final int M = 128;
    int[] keys = new int[M-1];
    BTreeNode<E>[] children = new BTreeNode[M];
    int numChildren = 0;
    ...
}
class BTreeLeaf<E implements Keyed> {
    static final int L = 32;
    E[] data = (E[]) new Object[L];
    int numItems = 0;
    ...
}
```
What that looks like in Java

**BTreeNode** (Interior node)

- keys
- children
- numChildren: 70

**BTreeLeaf** (Leaf node)

- data
- numItems: 20

All the red references indicate “unnecessary” indirection that might be avoided in another programming language.

- (array of M-1 ints)
- (array of M refs to BTreeNodes)
- (array of L refs to data objects)

Note: data objects not in contiguous memory.
The moral

• The whole idea behind B trees was to keep related data in contiguous memory

• But that’s “the best you can do” in Java
  – Again, the advantage is generic, reusable code
  – But for your performance-critical web-index, not the way to implement your B-Tree for terabytes of data

• Other languages (e.g., C++) have better support for “flattening objects into arrays”

• Levels of indirection matter!
Conclusion: Balanced Trees

- **Balanced** trees make good dictionaries because they guarantee logarithmic-time **find**, **insert**, and **delete**
  - Essential and beautiful computer science
  - But only if you can maintain balance within the time bound

- **AVL trees** maintain balance by tracking height and allowing all children to differ in height by at most 1

- **B trees** maintain balance by keeping nodes at least half full and all leaves at same height

- Other great balanced trees (see text; worth knowing they exist)
  - **Red-black trees**: all leaves have depth within a factor of 2
  - **Splay trees**: self-adjusting; amortized guarantee; no extra space for height information