CSE 373: B-trees

Michael Lee
Wednesday, Jan 31, 2018

Motivation

What we've done so far: study different dictionary implementations

- ArrayDictionary
- SortedArrayDictionary
- Binary search trees
- AVL trees
- Hash tables

They all make one common assumption: *all our data is stored in in-memory, on RAM.*

Motivation

New challenge: what if our data is too large to store all in RAM? (For example, if we were trying to implement a database?)

How can we do this efficiently?

Two techniques:

- A tree-based technique
  Excels for range-lookups (e.g. "find all users with an age between 20 and 30", where "age" is the key)
- A hash-based technique
  Excels for specific key-value pair lookups

A tree-based technique

Idea 1: Use an AVL tree

Suppose the tree has a height of 50. In the best case, how many disk accesses do we need to make? In the worst case?

In the best case, the nodes we want happen to be stored in RAM, so we need zero accesses.

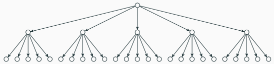
In the worst case, each node is stored on a different page on disk, so we need to make 50 accesses.

M-ary search trees

Idea 1:

- Instead of having each node have 2 children, make it have M children. Each node contains a **sorted** array of children nodes.
- Pick M so that each node fits into a single page

Example:



M-ary search trees

- What is the **height** of an M-ary search tree in terms of M and n? Assume the tree is balanced.
  The height is approximately $\log_M(n)$.
- What is the worst-case runtime of `get(...)`?
  We need to examine $\log_M(n)$ nodes.
  Per each node, we need to find the child to pick.
  We can do so using *binary search*: $\log_2(M)$
  Total runtime: $\text{height} \cdot \text{wordPerNode} = \log_M(n) \cdot \log_2(M)$.

M-ary trees

With M-ary trees, how many *disk accesses* do we make, assuming each node is stored on one page?

Is it $\log_M(n)$, or $\log_M(n)\log_2(M)$?

It's $\log_M(n)\log_2(M)$! When doing binary search, we need to check the child to see if its key is the one we should pick.

B-Trees

Idea 2:

- Rather then visiting each child, what if we stored the info we need in the parent – store keys?
- To avoid redundancy, store values only in leaf nodes.

Internal node

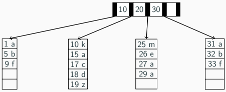
A node that stores only keys and pointers to children nodes

Leaf node

A node that stores only keys and values

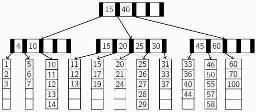
B-Trees

An example:



B-Trees

A larger example (values in leaf nodes omitted):



B-tree invariants

The B-tree invariants

1. The B-tree node type invariant
2. The B-tree order invariant
3. The B-tree structure invariant

The B-tree node type invariant

B-tree node type invariant

A B-tree has two types of node: **internal** nodes, and **leaf** nodes.

The B-tree node type invariant

B-tree internal node

An **internal node** contains M pointers to children and $M - 1$ **sorted** keys. Note: $M > 2$ must be true. Example of internal node where $M = 6$:

B-tree leaf node

A **leaf node** contains L key-value pairs, **sorted** by key. Example of leaf node where $L = 3$:

Note: M and L are parameters the creator of the B-tree must pick

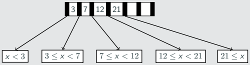
The B-tree order invariant

B-tree order invariant

For any given key k, all subtrees to the left may only contain keys x that satisfy $x < k$. All subtrees to the right may only contain keys x that satisfy $k \geq k$.

This means the subtree between two adjacent keys a and b may only contain keys x that satisfy $a \leq x < b$.

Example:



The B-tree structure invariant

B-tree structure when $n \leq L$

If $n \leq L$, the *root node* is a leaf:

B-tree structure when $n > L$

When $n > L$, the *root node* MUST be an internal node containing 2 to M children.

All *other* **internal** nodes must have $\lceil \frac{M}{2} \rceil$ to M children.

All **leaf** nodes must have $\lceil \frac{L}{2} \rceil$ to L children.

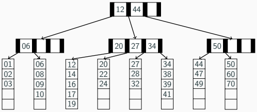
In other words: all nodes must be at least **half-full**. The only exception is the root, which can have as few as 2 children.

Why?

- Why must $M > 2$?
  Otherwise, we could end up with a linked list.
- Why do we insist almost all nodes must be at least half-full?
  It lets us ensure the tree stays *balanced*.
- Why is the root allowed to have as few as 2 children?
  If n is relatively small compared to M and L, it may not be possible for the root to actually be half-full.

B-tree get

Try running `get(6)`, `get(39)`



What's the worst-case runtime of `get(...)`? Num disk accesses?

Runtime roughly the same as M-ary trees: $\log_2(L) + \log_M(n)\log_2(M)$.

Number of disk accesses is $\log_M(n)$.

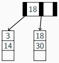
B-tree put

Suppose we have an empty B-tree where $M = 3$ and $L = 3$. Try inserting 3, 18, 14, 30:

After inserting 3, 18, 14:

We want to insert 30, but leaf node is out of space.

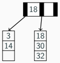
So, **SPLIT** the node:



B-tree put

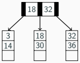
Next, try inserting 32 and 36.

After inserting 32:



We want to insert 36, but the leaf node is full!

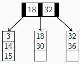
So, we **SPLIT** again:



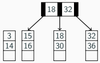
B-tree put

Next, try inserting 15 and 16.

After inserting 15:



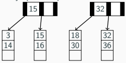
We try inserting 16. The node is full, so we **SPLIT:**



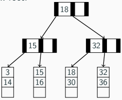
What do we do now?

B-tree put

Solution: Recursively split the parent!

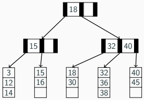


Then create a new root!



B-tree put

Now, try inserting 12, 40, 45, and 38.



Note: make sure to always fill "signpost" with **smallest value to right**

B-tree put

1. Insert data in correct leaf in **sorted order**.
2. If leaf has $L+1$ items, overflow.
   Split leaf into two new nodes:
   - Original leaf gets $\left\lceil \frac{L+1}{2} \right\rceil$ smaller items
   - New leaf gets $\left\lceil \frac{L}{2} \right\rceil$ larger items

   Attach new child and key to the parent (preserving sorted order).
3. Recursively continue overflowing if necessary. Note: for internal nodes, split using M instead of L.
4. If root overflows, make a new root.

B-tree put analysis

What is the worst-case runtime?

- Time to find correct leaf: $\Theta\left(\log_M(n)\log_2(M)\right)$
- Time to insert into leaf: $\Theta\left(L\right)$
- Time to split leaf: $\Theta\left(L\right)$
- Time to split parent: $\Theta\left(M\right)$
- Number of parents we might have to split: $\Theta\left(\log_M(n)\right)$

Overall runtime:

$$\text{timeFindLeaf} + \text{timeModifyLeaf} + \text{timeModifyParents}$$

Putting it all together:

$$\Theta\left(\log_M(n)\log_2(M) + L + M\log_M(n)\right) \quad = \quad \Theta\left(L + M\log_M(n)\right)$$

B-tree put analysis

Note:

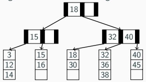
Runtime in the worst case is $\Theta(L + M \log_M(n))$.

However, splits are very rare! And splitting all the way to the root is even rarer. This means the average runtime is often better (often, just $\Theta(1)$ or $\Theta(L)$.

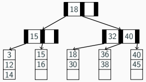
And at the end of the day, number of disk accesses matter more: it's still $\Theta(\log_M(n))$ no matter how many splits we do.

B-tree remove

Now, try deleting 32 then 15. The starting B-tree:

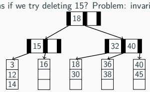


After deleting 32:

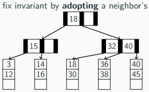


B-tree remove

What happens if we try deleting 15? Problem: invariant is broken!

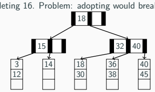


Solution: We fix invariant by **adopting** a neighbor's child!

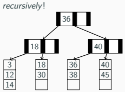


B-tree remove

Now, try deleting 16. Problem: adopting would break invariant!

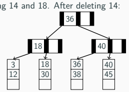


Solution: adopt *recursively*!

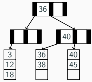


B-tree remove

Now, try deleting 14 and 18. After deleting 14:

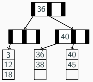


We try and delete 18....

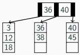


B-tree remove

Problem: invariant is broken, adopting recursively doesn't work:



Solution: Merge!



B-tree remove

1. Remove data from correct leaf
2. If leaf has $\left\lceil \frac{L}{2} \right\rceil$ items, underflow

   If neighbor has more then $\left\lceil \frac{L}{2} \right\rceil$, adopt one!

   Otherwise, **merge** with neighbor.
3. If we merged, parent has one fewer child. Recursively underflow if necessary (note: for internal nodes, we use M instead of L).
4. If we merge all the way up to the root and the root now has only one child, delete root and make child the root.

B-tree remove analysis

What is the worst-case runtime?

- Time to find correct leaf: $\Theta(\log_M(n)\log_2(M))$
- Time to remove from leaf: $\Theta(L)$
- Time to adopt/merge with neighbor: $\Theta(L)$
- Time to adopt/merge in parent: $\Theta(M)$
- Number of parents we might have to fix: $\Theta(\log_M(n))$

Putting it all together:

$$\Theta(L + M\log_M(n))$$

As before, average case runtime is frequently better because merges are very rare.

Picking M and L

Our original goal: make a disk-friendly dictionary.

Why are B-trees so disk-friendly?

- All relevant information about a single node fits in one page.
- We use as much of the page we can: each node contains many keys that are all brought in at once with a single disk access, basically "for free".
- The time needed to do a binary search within a node is insignificant compared to disk access time.

Picking M and L

So, how do we make sure a B-tree node *actually* fits in one page? How do we pick M and L?

Suppose we know the following:

1. One key is k bytes
2. One pointer is p bytes
3. One value is v bytes

Two questions:

- What is the size of an internal node?
  $Mp + (M-1)k$
- What is the size of a leaf node?
  $L(k+v)k$

Picking M and L

We know $Mp + (M-1)k$ is the size of one internal node, and $L(k+v)$ is the size of a leaf node.

Let's say one page (aka one *block*) takes up B bytes.

Goal: pick the largest M and L that satisfies these two inequalities:

$$Mp + (M-1)k \le B \qquad\qquad L(k+v) \le B$$

If we do the math:

$$M = \left\lfloor \frac{B+k}{p+k} \right\rfloor \qquad\qquad L = \left\lfloor \frac{B}{k+v} \right\rfloor$$

Summary

What we've done so far: study different dictionary implementations.

These implementations all assume data is all stored in RAM.

- ArrayDictionary
- SortedArrayDictionary
- Binary search trees
- AVL trees
- Hash tables

What if we have a lot of data that must be stored on disk?

Use a B-tree, which we *intentionally designed* to take advantage of how memory is accesssed in computers.

Summary

What you should know for midterm:

- The motivation behind why we made B-trees
- How to pick an optimal M and L
- A high level understanding of the B-tree invariants (e.g. be able to recognize when a B-tree is broken)
- The `get` algorithm

What you should know for final:

- The `put` and `remove` algorithms
- A more detailed understanding of the B-tree invariants