Topic 10: B-Trees (and variants)

- B-Trees
- Search
- How full are the nodes?
- Insert & Delete
- Properties of B-Trees
- 2-3 Trees
- 2-3-4 Trees
B-Tree

- A **B-Tree** is a search tree with many children per node.
- To allow searching, each node needs 1 fewer key than children.
This is a B-Tree of order $k$. 

B-Trees
B-Trees

A B-Tree of order $k$, with height $h$, can have up to $k^h$ leaves, with $k-1$ keys per leaf.
If $k=1000$, then a height 4 B-Tree stores $10^{15}$ keys in the leaves alone!
B-Trees

• Why use B-Trees?

Because of this
A Typical Hard Drive

- Throughput (sequential reads/writes):
  > 100 MB/sec

- Head Seek Time:
  5-10 ms

- Block size:
  512 bytes

https://www.youtube.com/watch?v=3owqvmMf6No
A Typical Hard Drive

- If you read large chunks of data (many blocks):
  > 100 MB / sec

- If you read single 512-byte blocks:
  ~ 100 KB / sec
Imagine storing a BST on disk:
- Every node is a block
- Every link requires a read of the disk
BSTs and Disks

Imagine storing a BST on disk:
~ 15-30 ms to read three levels (tiny tree!)
B-Trees and Disks

Imagine storing a B-Tree on disk:
~ 15-30 ms to read any node
How is this Possible?

- Asymptotic cost measures how many **operations** occur.
  - We roughly assume that all operations take comparable amount of time

- But disk storage isn't like that
  - 5 ms head seek
  - 5 ms to do roughly 1 billion CPU instructions
How is this Possible?

When a data structure is on disk:

- The only thing that matters is the **number of head seeks**
- CPU time is “**almost free**”
So Why Not Ban Disks?

• If disk is so slow, why do we use them?
  - Memory: 16 GB = $90
  - Disk: 2 TB = $75
  - SSD: 2 TB = $649

• You can afford a lot of disks for the price of a little bit of RAM! (Not to mention power costs!)

Prices: [http://www.newegg.com](http://www.newegg.com), Spring 2016
B-Trees

- So why not use B-Trees for everything?
- Programming complexity
- Memory overhead for small trees
- Huge cost of searching each node for keys
- Huge cost of inserting/deleting keys in nodes
B-Trees: Implementation

• How are the keys stored in a node?
  – Array, usually (alternatives possible)

• How to search an array?

• How to insert / delete? (Expensive!)
  – How to make insert / delete inside a node faster?
  – Probably use a BST (ick)
Topic 10: B-Trees (and variants)

- B-Trees
- Search
- How full are the nodes?
- Insert & Delete
- Properties of B-Trees
- 2-3 Trees
- 2-3-4 Trees
B-Trees: Searching

- B-Trees have an invariant like BSTs:
Suppose that we were searching for the key 73.
B-Trees: Searching

We recurse to the node between 50 and 100.
Topic 10: B-Trees (and variants)

- B-Trees
- Search
- How full are the nodes?
- Insert & Delete
- Properties of B-Trees
- 2-3 Trees
- 2-3-4 Trees
B-Trees: How full are the nodes?

- In BSTs, every node contains exactly one key
  - Zero, one, or two children
  - Linked lists possible
B-Trees: How full are the nodes?

- In B-Trees, always exactly 1 fewer keys than children
  - Every key marks the boundary between two subtrees
  - Min 2 children per node (except leaves)
  - Linked lists impossible

- Partially full nodes possible (and normal)
Topic 10: B-Trees (and variants)

- B-Trees
- Search
- How full are the nodes?
- Insert & Delete
- Properties of B-Trees
- 2-3 Trees
- 2-3-4 Trees
B-Trees: Insert and Delete

- B-Trees never add nodes – they **split** them.

100 keys → 1 key, 50 keys, 49 keys
B-Trees: Insert and Delete

• When you split a node with a parent, the new key goes into the parent.

Remember!
The parent has lots of other children that we're not drawing here.

• This may cause a recursive split
B-Trees: Insert and Delete

- Likewise, we can **join nodes** when they have too few keys.
B-Trees: Insert and Delete

- When the parent has multiple keys, a join removes one key.

Remember!
The parent has lots of other children that we're not drawing here.
B-Trees: When to Join or Split?

- **Bad (but works):**
  - Split only when totally full
  - Join only when totally empty
  - Wasteful in practice

- **Good Idea:**
  - Set thresholds for split/join
  - Keep all nodes between the thresholds
Performing an Insert into a B-Tree

- We always **insert into leaves.**

**Top-Down Strategy:**
- Split nodes as we go down, “just in case”
- Add to leaf

**Bottom-Up Strategy:**
- Add to leaf
- Split leaf (recurse to parents) if leaf is overloaded
Performing an Delete from a B-Tree

• Like a BST, delete can be harder
  - Easy in leaves with many keys (delete one key)
  - Sometimes join a node with a sibling
  - Sometimes requires swap-with-successor

• Like insert, can use top-down or bottom-up
• Details to come (in the 2-3-4 section)
Topic 10: B-Trees (and variants)

- B-Trees
- Search
- How full are the nodes?
- Insert & Delete
- Properties of B-Trees
- 2-3 Trees
- 2-3-4 Trees
Height in B-Trees

• A B-Tree is always perfectly height balanced.

In Class:
Why is this the case?
Height in B-Trees

• A B-Tree is **always perfectly height balanced.**

**Answer:**

• We never add or remove nodes, only split/join
  - If the split node has a parent, new key goes into the parent (can recurse)
  - If the split node is the root, new key becomes the new root (**adds height**)
  - Same argument for joins
Properties of B-Trees

**Size:**

- A B-Tree is never more empty than a perfectly balanced BST
- A B-Tree is never more full than a perfectly balanced $k$-ary tree.
Properties of B-Trees

\[ n = 2^{h+1} - 1 \]
\[ h = \Theta(\lg n) \]

In Class:
How many keys are there in a perfectly balanced \( k \)-ary tree of height \( h \)?
Properties of B-Trees

What can we conclude about the asymptotic relationship of $n$ and $h$ in a B-Tree?

$n = 2^{h+1} - 1$

$h = \Theta(lg\ n)$

$n = k^{h+1} - 1$

$h = \Theta(lg\ n)$
Properties of B-Trees

$n = 2^{h+1} - 1$
$h = \Theta(lg n)$

$h = \Theta(lg n)$

$n = k^{h+1} - 1$
$h = \Theta(lg n)$
Properties of B-Trees

Cost:

• Since the $h = \theta(\lg n)$, operations are also $\theta(\lg n)$
  - Obviously true for search
  - Will examine insert/delete next
Topic 10: B-Trees (and variants)

- B-Trees
- Search
- How full are the nodes?
- Insert & Delete
- Properties of B-Trees
- 2-3 Trees
- 2-3-4 Trees
B-Trees in Review

- **B-Tree**
  - \(k\)-ary tree
    - 2-3: B-Tree of order 3
    - 2-3-4: B-Tree of order 4
  - Nodes have 1 fewer key than child
  - All leaves at same height
    \[ h = \Theta(\log n) \]
    - Only increase height at root, keeps all balanced
  - Huge-\(k\) trees awesome for disk storage
2-3 Trees

- A “2-3” Tree is a B-Tree of order 3
- Small enough to be easy to code, fast
- Has the nice B-Tree properties
  - All leaves same height
  - Always balanced
2-3 Trees

- Each node has space for 2 keys

```
< 100  100 < x < 200  > 200
  100  200
```

```
2-3 Trees

- Some nodes have only 2 children, and only 1 key. (These work like binary tree nodes.)
2-3 Trees

- A leaf in a 2-3 tree can store 1 or 2 keys.
2-3 Trees

- An internal node in a 2-3 tree must have 2 or 3 children.
2-3 Trees

- An internal node always has 1 fewer keys than children

2 keys, 3 children

1 key, 2 children
2-3 Trees

- All leaves in a 2-3 tree are at the same height.
Insertion in 2-3 Trees

Suppose we want to insert 7 into the tree...

7

10 20

5

15

25
Insertion in 2-3 Trees

We always **insert** into leaves.
Now we'll try to add 12.
Insertion in 2-3 Trees

Now we'll try to add 12.
Insertion in 2-3 Trees

22

22 is next...

```
      10  20
   /     \
(5 7)   (12 15)
    |
   25
```
Insertion in 2-3 Trees

22 is next...
Insertion in 2-3 Trees

What happens if a leaf is full? Let's insert 6.

I've removed one of the child nodes, for now.
We'd like to insert into this leaf, but there's no space to store it.
We split our node into two pieces, and send a single key up to the parent.

The key must always be the middle value.

Note that it's not always the one we just inserted.
Insertion in 2-3 Trees

What happens when the parent is full?

Let's insert 18 to find out.
Insertion in 2-3 Trees

```
  18

 6   10

 5    7

12 15 (18)
```
As I mentioned, the value we push up is not always the one we inserted. 

How do we fix the parent node?

Recursion!
Insertion in 2-3 Trees

If there was a parent, then 10 is pushed up.

If not, then we create a new layer.
Insertion in 2-3 Trees

Cost of insertion:
- Search: $O(\log n)$
- Split: $O(\log n)$
Insetion in 2-3 Trees

- Insertion in 2-3 Trees must always be “bottom-up” (fix overloaded nodes after the fact).
- Top-down algorithms are impossible in 2-3 Trees.

In Class:
Why is this?
Insertion in 2-3 Trees

- Insertion in 2-3 Trees must always be “bottom-up” (fix overloaded nodes after the fact).
- Top-down algorithms are impossible in 2-3 Trees.
  - With only 2 keys per node, you can never split it
  - Need at least 3 keys per node to split
Deletion in 2-3 Trees

- Deletion is Hard!
  - Easy in a leaf w/ 2 keys
  - Harder in a leaf w/ 1 key
  - Harder still in an internal node
- Yes, it's possible to solve it (go look it up), but...
- Let's do 2-3-4 trees instead!
Topic 10: B-Trees (and variants)

- B-Trees
- Search
- How full are the nodes?
- Insert & Delete
- Properties of B-Trees
- 2-3 Trees
- 2-3-4 Trees
2-3-4 Trees

• A 2-3-4 Tree is a B-Tree of order 4.

• All of the same B-Tree properties hold.
• All of the same B-Tree algos work.
  – Including bottom-up insertion, which we just showed in a 2-3 tree
Bottom-Up vs. Top-Down

- Bottom-up is often more efficient
  - Only make modifications when necessary
- Top-down is more elegant
  - Easier to understand
  - Fewer cases
  - No parent pointers required

- We'll use 2-3-4 trees to show the top-down algos
Top-Down: Concept

• In top-down operations
  – Split/join nodes *as you search down*
  – Split/join is never required after you make a change

• Simpler operation
  – Always safe to insert/delete

• Changes too often
  – Sometimes, a split/join is unnecessary
This is a 2-3-4 tree. Up to 4 children per node. Up to 3 keys per node. keys=children-1
Top-Down Insertion in 2-3-4 Trees

Let's insert 7 into the tree.

We start at the root, and see that it has 3 keys.

We split it, just in case we later need to push a key up.
Top-Down Insertion in 2-3-4 Trees

The 3 keys get split into 3 different nodes.

We restart the insertion process at the new root.
We recurse through the tree, just like bottom-down, until we find another node which has 3 keys in it. We split that node.
Top-Down Insertion in 2-3-4 Trees

After the split, we return to the parent node, and keep recursing down.
When we finally reach a leaf, we place the new value into that leaf.
Why it Works

• We never have to split a leaf after insertion
  - If the leaf was full, we would have split it already

• When we split a node, the split never recurses
  - If the parent was full, we would have split it already
Top-Down Deletion in 2-3-4 Trees

• So how do we Delete?
  - Do proactive **joins** on parents as we go down the tree (like proactive splits in Insert)
  - If key is in a leaf node, simply remove it
    • Because of proactive join, it's impossible that the leaf has only one node (unless it's the root)
  - If key is in an internal node, swap with successor, and delete the other
Let's delete 15 from the tree.

We search for it, and check each node. Each node has at least two keys, so we're OK.
Top-Down Deletion in 2-3-4 Trees

Let's delete 15 from the tree.

We search for it, and check each node. Each node has at least two keys, so we're OK.
Top-Down Deletion in 2-3-4 Trees

We've deleted it from the leaf. The leaf still has 1 key left over.
Now, let's delete 150.

One of the nodes has only one value, so we pre-join, just in case we'll need to steal from it later.
In this case, one of our siblings has 2 keys...so we can solve this with a rotation.
Top-Down Deletion in 2-3-4 Trees

50 moved up.
100 moved down.
The 75 leaf changed its parent.

We can resume the search (at the parent of the node we stopped at).
Top-Down Deletion in 2-3-4 Trees

50 moved up. 100 moved down.
The 75 leaf changed its parent.
We can resume the search (at the parent of the node we stopped at).
Top-Down Deletion in 2-3-4 Trees

The 150 leaf has siblings, but none have multiple keys. We solve this with a join.
To perform the join, we had to steal 200 from the parent above.

We now resume the deletion process.
Top-Down Deletion in 2-3-4 Trees

We have now deleted 150 from the tree.
Top-Down Deletion in 2-3-4 Trees

The root is a special case.

Since it has no siblings, rotations and joins are impossible.

But that's OK – if we join at the next level, just steal from the root.

If the root disappears, that just means that tree got shorter!
Top-Down Deletion in 2-3-4 Trees

One last example: we'll delete 100 (which is in an internal node) from the tree.

Like a BST, we'll swap with the successor.
Top-Down Deletion in 2-3-4 Trees

We searched for 100 and found it in an internal node.
Top-Down Deletion in 2-3-4 Trees

We swapped 100 with its successor.

Note that this search process would do top-down joins as well.
This is the tree after we delete 100.
Remember!

- Bottom-up can be used in any B-tree
- Top-down can be used in 2-3-4 and larger (but not 2-3)
- Bottom-up more efficient
- Top-down easier to understand
Topic 10: B-Trees (and variants)

- B-Trees
- Search
- How full are the nodes?
- Insert & Delete
- Properties of B-Trees
- 2-3 Trees
- 2-3-4 Trees

Summary