Data Structures and Algorithms
CS245-2017S-20

B-Trees

David Galles

Department of Computer Science
University of San Francisco
20-0: **Indexing**

- Operations:
  - Add an element
  - Remove an element
  - Find an element, using a key
  - Find all elements in a range of key values
20-1: **Indexing**

- **Sorted List**
  - Find / Find in Range fast
  - Add / Remove slow

- **Unsorted List / Hash Table**
  - Add, Find, Remove fast (hash)
  - Find in Range slow

- **Binary Search Tree**
  - All operations are fast \(O(\lg n)\)
  - *if* the tree is balanced
Indexing

- Generalized Binary Search Trees
  - Each node can store several keys, instead of just one
  - Values in subtrees between values in surrounding keys
  - For non leaves, # of children = # of keys + 1
20-3: 2-3 Trees

- Generalized Binary Search Tree
  - Each node has 1 or 2 keys
  - Each (non-leaf) node has 2-3 children
    - hence the name, 2-3 Trees
  - All leaves are at the same depth
Example 2-3 Tree

- Root: 6, 16
- Left child: 3
  - Left child: 2
  - Right child: 5
- Right child: 8, 13
  - Right child: 11, 12
- Right child: 14
- Right child: 17
- Right child: 18
- Right child: 20
Finding in 2-3 Trees

- How can we find an element in a 2-3 tree?
Finding in 2-3 Trees

- How can we find an element in a 2-3 tree?
  - If the tree is empty, return false
  - If the element is stored at the root, return true
  - Otherwise, recursively find in the appropriate subtree
20-7: Inserting into 2-3 Trees

- Always insert at the leaves
- To insert an element:
  - Find the leaf where the element would live, if it was in the tree
  - Add the element to that leaf
20-8: Inserting into 2-3 Trees

- Always insert at the leaves
- To insert an element:
  - Find the leaf where the element would live, if it was in the tree
  - Add the element to that leaf
    • What if the leaf already has 2 elements?
Always insert at the leaves

To insert an element:
  • Find the leaf where the element would live, if it was in the tree
  • Add the element to that leaf
    • What if the leaf already has 2 elements?
      • Split!
20-10: Splitting Nodes
20-11: Splitting Nodes

Too many elements
20-12: Splitting Nodes

Promote to parent

Left child of 6

Right child of 6
Splitting Nodes
20-14: Splitting Root

- When we split the root:
  - Create a new root
  - Tree grows in height by 1
Inserting elements 1-9 (in order) into a 2-3 tree
20-16: 2-3 Tree Example

- Inserting elements 1-9 (in order) into a 2-3 tree
20-17: 2-3 Tree Example

- Inserting elements 1-9 (in order) into a 2-3 tree

\[
\begin{array}{ccc}
1 & 2 & 3 \\
\end{array}
\]

Too many keys, need to split
Inserting elements 1-9 (in order) into a 2-3 tree
20-19: 2-3 Tree Example

- Inserting elements 1-9 (in order) into a 2-3 tree
20-20: 2-3 Tree Example

- Inserting elements 1-9 (in order) into a 2-3 tree

```
  2
 / \
1   3 4 5
```

Too many keys, need to split
20-21: 2-3 Tree Example

- Inserting elements 1-9 (in order) into a 2-3 tree
Inserting elements 1-9 (in order) into a 2-3 tree
2-3 Tree Example

- Inserting elements 1-9 (in order) into a 2-3 tree

```
  2  4
 /   |
1   3  5  6  7
```

Too many keys need to split
20-24: **2-3 Tree Example**

- Inserting elements 1-9 (in order) into a 2-3 tree

```
  2 4 6
  /  |
 1 3 5
```

Too many keys need to split
20-25: 2-3 Tree Example

- Inserting elements 1-9 (in order) into a 2-3 tree

```
    4
   /|
  2 6
 /|
1 3 5 7
```
20-26: 2-3 Tree Example

- Inserting elements 1-9 (in order) into a 2-3 tree
Inserting elements 1-9 (in order) into a 2-3 tree.
• Inserting elements 1-9 (in order) into a 2-3 tree
As with BSTs, we will have 2 cases:
- Deleting a key from a leaf
- Deleting a key from an internal node
20-30: Deleting Leaves

- If leaf contains 2 keys
  - Can safely remove a key
Deleting Leaves

- Deleting 7
Deleting Leaves

- Deleting 7
Deleting Leaves

- If leaf contains 1 key
  - Cannot remove key without making leaf empty
  - Try to steal extra key from sibling
Deleting 3 – we can steal the 5
20-35: Deleting Leaves

- Not a 2-3 tree. What can we do?
Deleting Leaves

- Steal key from sibling *through parent*
20-37: Deleting Leaves

- Steal key from sibling *through parent*
20-38: Deleting Leaves

- If leaf contains 1 key, and no sibling contains extra keys
  - Cannot remove key without making leaf empty
  - Cannot steal a key from a sibling
  - Merge with sibling
    - split in reverse
Removing the 4
• Removing the 4
• Combine 5, 7 into one node
20-41: Merging Nodes

Diagram:

- Node 8
- Node 5
- Node 7
- Node 11

Diagram shows the merging of nodes 5 and 7 into node 8, and then node 8 merging with node 11.
Merging Nodes

- Merge decreases the number of keys in the parent
  - May cause parent to have too few keys
- Parent can steal a key, or merge again
• Deleting the 3 – cause a merge
20-44: Merging Nodes

- Deleting the 3 – cause a merge
- Not enough keys in parent
Merging Nodes

- Steal key from sibling
Merging Nodes

- Steal key from sibling
When we steal a key from an internal node, steal nearest subtree as well.
• When we steal a key from an internal node, steal nearest subtree as well
Merging Nodes

- Deleting the 7 – cause a merge
Merging Nodes

- Parent has too few keys – merge again
20-51: Merging Nodes

- Root has no keys – delete
Merging Nodes
How can we delete keys from non-leaf nodes?

*HINT:* How did we delete non-leaf nodes in standard BSTs?
How can we delete keys from non-leaf nodes?

- Replace key with smallest element subtree to right of key
- Recursively delete smallest element from subtree to right of key
- (can also use largest element in subtree to left of key)
Deleting Interior Keys

- Deleting the 4
• Deleting the 4
• Replace 4 with smallest element in tree to right of 4
20-57: Deleting Interior Keys
20-58: Deleting Interior Keys

- Deleting the 5
Deleting Interior Keys

- Deleting the 5
- Replace the 5 with the smallest element in tree to right of 5
Deleting Interior Keys

- Deleting the 5
- Replace the 5 with the smallest element in tree to right of 5
- Node with two few keys
Deleting Interior Keys

- Node with two few keys
- Steal a key from a sibling
Deleting Interior Keys
• Removing the 6
• Removing the 6
• Replace the 6 with the smallest element in the tree to the right of the 6
20-65: Deleting Interior Keys

- Node with too few keys
  - Can’t steal key from sibling
  - Merge with sibling
### Deleting Interior Keys

- **Node with too few keys**
  - Can’t steal key from sibling
  - Merge with sibling
  - (arbitrarily pick right sibling to merge with)
Deleting Interior Keys
20-68: Generalizing 2-3 Trees

- In 2-3 Trees:
  - Each node has 1 or 2 keys
  - Each interior node has 2 or 3 children
- We can generalize 2-3 trees to allow more keys / node
20-69: **B-Trees**

- A B-Tree of maximum degree $k$:
  - All interior nodes have $\lceil k/2 \rceil \ldots k$ children
  - All nodes have $\lceil k/2 \rceil - 1 \ldots k - 1$ keys
- 2-3 Tree is a B-Tree of maximum degree 3
B-Trees

- B-Tree with maximum degree 5
- Interior nodes have 3 – 5 children
- All nodes have 2-4 keys
Inserting into a B-Tree
- Find the leaf where the element would go
- If the leaf is not full, insert the element into the leaf
- Otherwise, split the leaf (which may cause further splits up the tree), and insert the element
20-72: B-Trees

- Inserting a 6 ..
Inserting a 10 ..
20-75: B-Trees

- Promote 8 to parent (between 5 and 11)
- Make nodes out of (6, 7) and (9, 10)
- Promote 11 to parent (new root)
- Make nodes out of (5, 8) and (6, 19)
Note that the root only has 1 key, 2 children.

All nodes in B-Trees with maximum degree 5 should have at least 2 keys.

The root is an exception – it may have as few as one key and two children for any maximum degree.
B-Trees

- B-Tree of maximum degree $k$
  - Generalized BST
  - All leaves are at the same depth
  - All nodes (other than the root) have $\lceil k/2 \rceil - 1 \ldots k - 1$ keys
  - All interior nodes (other than the root) have $\lceil k/2 \rceil \ldots k$ children
20-79: B-Trees

- B-Tree of maximum degree $k$
  - Generalized BST
  - All leaves are at the same depth
  - All nodes (other than the root) have $\lceil k/2 \rceil - 1 \ldots k - 1$ keys
  - All interior nodes (other than the root) have $\lceil k/2 \rceil \ldots k$ children
- Why do we need to make exceptions for the root?
Why do we need to make exceptions for the root?
Consider a B-Tree of maximum degree 5 with only one element
B-Trees

• Why do we need to make exceptions for the root?
  • Consider a B-Tree of maximum degree 5 with only one element
  • Consider a B-Tree of maximum degree 5 with 5 elements
Why do we need to make exceptions for the root?

- Consider a B-Tree of maximum degree 5 with only one element
- Consider a B-Tree of maximum degree 5 with 5 elements
- Even when a B-Tree could be created for a specific number of elements, creating an exception for the root allows our split/merge algorithm to work correctly.
Deleting from a B-Tree (Key is in a leaf)
- Remove key from leaf
- Steal / Split as necessary
- May need to split up tree as far as root
• Deleting the 15
20-85: B-Trees

Too few keys
• Steal a key from sibling
20-87: B-Trees
• Delete the 11
20-89: B-Trees

```
   5   9   16   19
  /     /     /    /
1 3  7  8  12        17 18
  |    |              |    |
  22   23             |
```

Too few keys
20-90: B-Trees

*Combine into 1 node*

- Merge with a sibling (pick the left sibling arbitrarily)
20-91: B-Trees
Deleting from a B-Tree (Key in internal node)
- Replace key with largest key in right subtree
- Remove largest key from right subtree
- (May force steal / merge)
• Remove the 5
• Remove the 5
20-95: B-Trees
• Remove the 19
Remove the 19
Too few keys
• Merge with left sibling
20-100: B-Trees
Almost all databases that are large enough to require storage on disk use B-Trees.

Disk accesses are very slow:
- Accessing a byte from disk is 10,000 – 100,000 times as slow as accessing from main memory.
- Recently, this gap has been getting even bigger.

Compared to disk accesses, all other operations are essentially free.

Most efficient algorithm minimizes disk accesses as much as possible.
Disk accesses are slow – want to minimize them

Single disk read will read an entire sector of the disk

Pick a maximum degree $k$ such that a node of the B-Tree takes up exactly one disk block
  
  Typically on the order of 100 children / node
With a maximum degree around 100, B-Trees are very shallow.

Very few disk reads are required to access any piece of data.

Can improve matters even more by keeping the first few levels of the tree in main memory.

For large databases, we can’t store the entire tree in main memory – but we can limit the number of disk accesses for each operation to only 1 or 2.