Data Structures and Algorithms

Indexing

Chris Brooks

Department of Computer Science
University of San Francisco
Often, when managing records, we want to do more than just find a record that matches a particular key.

- Find all records in a range, in order.
- Sort by one of several keys

Also, records may not all fit into memory
What are the operations we need?:

- Add an element
- Remove an element
- Find an element, using a key
- Find all elements in a range of key values
Some Possible Data Structures:

- **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 ($O(\log n)$)
  - *if* the tree is balanced

Trees look like a good choice - how to keep them balanced?
So far, we’ve assumed our entire data structure can fit into main memory.

What if it’s too large?

Need to minimize the number of disk accesses

- Speaking practically, reading from the disk is much slower than reading from memory
  - RAM: 50 nanoseconds
  - disk: 5 milliseconds (100,000 times slower)

- Strategies
  - Keep tree as shallow as possible
  - Store records with adjacent keys in adjacent blocks on disk
A 2-3 tree is a generalized Binary Search Tree

Not typically implemented, but a good intro
- 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
17-5: Example 2-3 Tree
How can we find an element in a 2-3 tree?
17-7: 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 (just like a BST)
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
17-9: 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!
17-11: Splitting Nodes

Diagram: A tree node labeled with 5, 6, and 7, with child nodes labeled 5 and 7.
17-12: Splitting Nodes

Too many elements
17-13: Splitting Nodes

Promote to parent

Left child of 6

Right child of 6
17-14: Splitting Nodes
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
Inserting elements 1-9 (in order) into a 2-3 tree
Inserting elements 1-9 (in order) into a 2-3 tree

1 2 3

Too many keys, need to split
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

Too many keys, need to split
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

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

Too many keys need to split
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
Inserting elements 1-9 (in order) into a 2-3 tree.
6 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
6 If leaf contains 2 keys
   ▲ Can safely remove a key
6 Deleting 7
Deleting 7
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
Not a 2-3 tree. (Left leaf greater than its parent.) What can we do?
6 Steal key from sibling *through parent*
6. Steal key from sibling *through parent*
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
17-42: Merging Nodes
Merging Nodes

6 Merge decreases the number of keys in the parent
   △ May cause parent to have too few keys
6 Parent can steal a key, or merge again
17-44: Merging Nodes

- Deleting the 3 – cause a merge
Deleting the 3 – cause a merge

Not enough keys in parent
6. Steal key from sibling
6. Steal key from sibling
When we steal a key from an internal node, steal nearest subtree as well
6 When we steal a key from an internal node, steal nearest subtree as well
Deleting the 7 – cause a merge
Parent has too few keys – merge again
6 Root has no keys – delete
17-53: 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 the 4
Deleting the 4

Replace 4 with smallest element in tree to right of 4
17-58: 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
Node with two few keys
Steal a key from a sibling
17-63: Deleting Interior Keys

```
     6
   /   \
  2     8
 /     /  \
1     3   7
      /    /  \
     7    9
```
Removing the 6
17-65: Deleting Interior Keys

Removing the 6

Replace the 6 with the smallest element in the tree to the right of the 6
Node with too few keys
- Can’t steal key from sibling
- Merge with sibling
Node with too few keys
  ▲ Can’t steal key from sibling
  ▲ Merge with sibling
  ▲ (arbitrarily pick right sibling to merge with)
17-68: Deleting Interior Keys
17-69: 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
A B-Tree of maximum degree $k$:

- All interior nodes have $\left\lceil k/2 \right\rceil \ldots k$ children
- All nodes have $\left\lfloor k/2 \right\rfloor - 1 \ldots k - 1$ keys

2-3 Tree is a B-Tree of maximum degree 3
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
Inserting a 6 ..
17-74: B-Trees

```
5 11 16 19

1 3
6 7 8 9
12 15
17 18
22 23
```
Inserting a 10
Too many keys need to split

- Promote 8 to parent (between 5 and 11)
- Make nodes out of (6, 7) and (9, 10)
Too many keys need to split

1. Promote 11 to parent (new root)
2. 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-Tree of maximum degree $k$

△ Generalized BST
△ All leaves are at the same depth
△ All nodes (other than the root) have $\left\lfloor k/2 \right\rfloor - 1 \ldots k - 1$ keys
△ All interior nodes (other than the root) have $\left\lfloor k/2 \right\rfloor \ldots k$ children
B-Tree of maximum degree $k$

- Generalized BST
- All leaves are at the same depth
- All nodes (other than the root) have $\left\lceil \frac{k}{2} \right\rceil - 1 \ldots k - 1$ keys
- All interior nodes (other than the root) have $\left\lfloor \frac{k}{2} \right\rfloor \ldots k$ 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

6 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
Too few keys
Steal a key from sibling
B-Trees
Delete the 11
17-90: B-Trees

Too few keys
Combine into 1 node

Merge with a sibling (pick the left sibling arbitrarily)
17-92: 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
17-96: B-Trees

```
1  3  
8  9  12
17 18
22 23
```

```
7  16  19
```
Remove the 19
Remove the 19
17-99: B-Trees

Too few keys
Merge with left sibling
17-101: B-Trees

Diagram of a B-Tree:

- Root node with values 7 and 16
- Child nodes with values 1, 3
- Child nodes with values 8, 9, 12
- Child nodes with values 17, 18, 22, 23

Department of Computer Science — University of San Francisco — p.103/108
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.
In practice, B-trees are typically implemented using a variant known as B+-trees.

All data kept at the leaves.

Internal nodes just contain key values.

Most operations are identical to B-trees, except search.

Since all elements are stored at leaves, search must continue down the entire tree.
We will often want a list of records whose keys are within some range.

- All students with GPA between 2.5 and 3.5

Solution: Keep a set of pointers between siblings.

Find the first element in the range, and then follow sibling pointers until we find the last element.