B Trees (aka B+ Trees)

- Two types of nodes:
  - Internal Nodes
    - Sorted array of $M - 1$ keys
    - Has $M$ children
    - No other data!
  - Leaf Nodes
    - Sorted array of $L$ key-value pairs

- Subtree between values $a$ and $b$ must contain only keys that are $\geq a$ and $< b$
  - If $a$ is missing use $-\infty$
  - If $b$ is missing use $\infty$
Find

- Start at the root node
- Binary search to identify correct subtree
- Repeat until you reach a leaf node
- Binary search the leaf to get the value
B Tree Structure Requirements

• Root:
  • If the tree has \( \leq L \) items then root is a leaf node
  • Otherwise it is an internal node

• Internal Nodes:
  • Must have at least \( \left\lceil \frac{M}{2} \right\rceil \) children (at least half full)

• Leaf Nodes:
  • Must have at least \( \left\lceil \frac{L}{2} \right\rceil \) items (at least half full)
  • All leaves are at the same depth
Insertion Summary

• Binary search to find which leaf should contain the new item
• If there’s room, add it to the leaf array (maintaining sorted order)
• If there’s not room, **split**
  • Make a new leaf node, move the larger $\left\lceil \frac{L+1}{2} \right\rceil$ items to it
  • If there’s room in the parent internal node, add new leaf to it (with new key bound value)
  • If there’s not room in the parent internal node, **split** that!
    • Make a new internal node and have it point to the larger $\left\lceil \frac{M+1}{2} \right\rceil$
    • If there’s room in the parent internal node, add this internal node to it
    • If there’s not room, repeat this process until there is!
Insertion TLDR

• Find where the item goes by repeated binary search
• If there’s room, just add it
• If there’s not room, split things until there is
Insert Example

Insert 22
Insert Example

Insert 22
Insert Example

Insert 26
Insert Example

Insert 26
Insert Example

Insert 8
Insert Example

Insert 8

Split!

Split!

Split!
Insert Example

Insert 8
Insert Example

Insert 8
Let’s do it together!

- $M = 3, L = 3$
- Inserts all of these:
Running Time of Find

• Maximum number of leaves:
  • $\frac{2n}{L}$
  • $\Theta \left( \frac{n}{L} \right)$
• Maximum height of the tree:
  • $2 \log_M \frac{2n}{L}$
  • $\Theta \left( \log_M \frac{n}{L} \right)$
• Find:
  • One binary search per level of the tree
    • $\Theta(\log_2 M)$ per search
  • One binary search in the leaf
    • $\Theta(\log_2 L)$

Overall: $\Theta \left( \log_2 M \cdot \log_M \frac{n}{L} + \log_2 L \right)$
Usually simplified to: $\Theta(\log_2 M \cdot \log_M n)$
Running Time of Insert

• Find:
  • $\Theta(\log_2 M \cdot \log_M n)$

• Add item to leaf:
  • $\Theta(L)$

• Split a leaf
  • $\Theta(L)$

• Split one internal node:
  • $\Theta(M)$

Overall: $\Theta(L + M \cdot \log_M n)$
Usually simplified to:
$\Theta(\log_2 M \cdot \log_M n)$
Delete

• Recall: all nodes must be at least half full (except root at startup)

delete 50
Delete

• Recall: all nodes must be at least half full (except root at startup)

delete 24
Delete

- Recall: all nodes must be at least half full (except root at startup)

```plaintext
delete 24
```

```
3 5
1 3 5
2
```
```
9
7 8 9
```
```
17 25
13 14 17 20 25 27 30
```
```
55
38 40 50 55
```

```
Delete

- Recall: all nodes must be at least half full (except root at startup)

delete 5

```
    7 13 38
   /  \
  17 25
 /    \\    
13  17  20 25 27
 /  \
14  16
    /   \    
   17  20  27  30
```

```
      3  5
     /  \
    9
```

```
      1  3  5
     /  \
    7  9
   /  \
  7  9
```

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

```
       38  40  50  55
      /  \
    38  40  50  55
```
Delete

• Recall: all nodes must be at least half full (except root at startup)

delete 5
Delete

- Recall: all nodes must be at least half full (except root at startup)

delete 1
Delete

- Recall: all nodes must be at least half full (except root at startup)

```
delete 1
```
Delete

• Recall: all nodes must be at least half full (except root at startup)

delete 1
Delete Summary

• Find the item

• Remove the item from the leaf
  • If that causes the leaf to be underfull, adopt from a neighbor
  • If that would cause the neighbor to be underfull, merge those two leaves
  • Update the parent
    • If that causes the parent to be underfull, adopt from a neighbor
    • If that causes the neighbor to be underfull, merge
    • Update the parent
      • ...
Delete TLDR

• Find and remove from leaf

• Keep doing this until everything is “full enough”:
  • If the node is now too small, adopt from a neighbor
  • If the neighbor is too small then merge
Aside: Implementation

• What an internal node class might look like:
  • int M
  • int[] keys
  • Node[] children
  • int num_children

• What a leaf node class might look like:
  • int L
  • E[] data
  • int num_items
Next topic: Hash Tables

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Time to insert</th>
<th>Time to find</th>
<th>Time to delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Unsorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Array</td>
<td>$\Theta(n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Sorted Linked List</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Binary Search Tree</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>AVL Tree</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td>Hash Table (Worst case)</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>Hash Table (Average)</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Two Different ideas of “Average”

• Expected Time
  • The expected number of operations a randomly-chosen input uses
  • Assumed randomness from somewhere
    • Most simply: from the input
    • Preferably: from the algorithm/data structure itself
  • \( f(n) = \text{sum of the running times for each input of size } n \text{ divided by the number of inputs of size } n \)

• Amortized Time
  • The long-term average per-execution cost (in the worst case)
  • Rather than look at the worst case of one execution, look at the total worst case of a sequential chain of many executions
    • Why? The worst case may be guaranteed to be rare
  • \( f(n) = \text{the sum of the running times from a sequence of } n \text{ sequential calls to the function divided by } n \)
Amortized Example

• ArrayList Insert:
  • Worst case: $\Theta(n)$
Amortized Example

• ArrayList Insert:
  • First 8 inserts: 1 operation each
  • 9\textsuperscript{th} insert: 9 operations
  • Next 7 inserts: 1 operation each
  • 17\textsuperscript{th} insert: 17 operations
  • Next 15 inserts: 1 operation each
  • ...

\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}

Do \( x \) operations with cost 1
Do 1 operation with cost \( x \)
Do \( x \) operations with cost 1
Do 1 operation with cost 2\( x \)
Do 2\( x \) operations with cost 1
Do 1 operation with cost 4\( x \)
Do 4\( x \) operations with cost 1
Do 1 operation with cost 8\( x \)
...

Amortized: each operation cost 2 operations \( \Theta(1) \)
Hash Tables

• Motivation:
  • Why not just have a gigantic array?
Hash Tables

• Idea:
  • Have a small array to store information
  • Use a **hash function** to convert the key into an index
    • Hash function should “scatter” the keys, behave as if it randomly assigned keys to indices
  • Store key at the index given by the hash function
  • Do something if two keys map to the same place (should be very rare)
    • Collision resolution

Key Object

\[ h(k) \]

Index between 0 and size-1

Insert / find / delete

& value
Example

- Key: Phone Number
- Value: People
- Table size: 10
- $h(phone) = \text{number as an integer} \mod 10$
- $h(8675309) = 9$
What Influences Running time?