Three Flavors of Balance

How to guarantee efficient search trees has been an active area of data structure research.

We will explore three variations of "balancing":

- **AVL Trees**: Guaranteed balanced BST with only constant time additional overhead.
- **Splay Trees**: Ignore balance, focus on recency.
- **B Trees**: n-ary balanced search trees that work well with real world memory/disks.

Achieving a Balanced BST (part 1)

For a BST with n nodes inserted in arbitrary order:

- Average height is $O(\log n)$ – see text.
- Worst case height is $O(n)$.
- Simple cases, such as pre-sorted, lead to worst-case scenario.
- Inserts and removes can and will destroy any current balance.

Achieving a Balanced BST (part 2)

Shallower trees give better performance:

- This happens when the tree's height is $O(\log n)$ – like a perfect or complete tree.

Solution: Require a **Balance Condition** that:

1. Ensures depth is always $O(\log n)$.
2. Is easy to maintain.
Potential Balance Conditions

1. Left and right subtrees of the root have equal number of nodes

2. Left and right subtrees of the root have equal height

The AVL Balance Condition

Left and right subtrees of every node have heights differing by at most 1

Mathematical Definition:
For every node x, \(-1 \leq \text{balance}(x) \leq 1\) where
\[
\text{balance}(\text{node}) = \text{height(} \text{node.left} \text{)} - \text{height(} \text{node.right} \text{)}
\]

AVL Balance Condition

Ensures small depth
- Can prove by showing an AVL tree of height h must have nodes exponential in h

Efficient to maintain
- Requires adding a height parameter to the node class (Why?)
- Balance is maintained through constant time manipulations of the tree structure: single and double rotations

Calculating Height

What is the height of a tree with root r?

```java
int treeHeight(Node root) {
    if (root == null) return -1;
    return 1 + max(treeHeight(root.left), treeHeight(root.right));
}
```

Running time for tree with n nodes:
\(O(n)\) – single pass over tree

Very important detail of definition:
height of a null tree is \(-1\), height of tree with a single node is 0
**Height of an AVL Tree?**

Using the AVL balance property, we can determine the minimum number of nodes in an AVL tree of height \( h \).

**Recurrence relation:**

Let \( S(h) \) be the minimum nodes in height \( h \), then

\[
S(h) = S(h-1) + S(h-2) + 1
\]

where \( S(-1) = 0 \) and \( S(0) = 1 \).

**Solution of Recurrence:** \( S(h) \approx 1.62^h \)

---

**Minimal AVL Tree (height = 0)**

- 0

---

**Minimal AVL Tree (height = 1)**

- 1 node

---

**Minimal AVL Tree (height = 2)**

- 3 nodes

---

**Minimal AVL Tree (height = 3)**

- 7 nodes

---

**Minimal AVL Tree (height = 4)**

- 15 nodes
AVL Tree Operations

AVL find:
- Same as BST find

AVL insert:
- Starts off the same as BST insert
- Then check balance of tree
- Potentially fix the AVL tree (4 imbalance cases)

AVL delete:
- Do the deletion
- Then handle imbalance (same as insert)

Insert / Detect Potential Imbalance

Insert the new node (at a leaf, as in a BST)
- For each node on the path from the new leaf to the root
- The insertion may, or may not, have changed the node’s height

After recursive insertion in a subtree
- detect height imbalance
- perform a rotation to restore balance at that node

All the action is in defining the correct rotations to restore balance

The Secret

If there is an imbalance, then there must be a deepest element that is imbalanced
- After rebalancing this deepest node, every node is then balanced
- Ergo, at most one node needs rebalancing

Single Rotation

The basic operation we use to rebalance
- Move child of unbalanced node into parent position
- Parent becomes a "other" child
- Other subtrees move as allowed by the BST

Example

Insert(6)
Insert(3)
Insert(1)

Third insertion violates balance
What is a way to fix this?

Single Rotation Example: Insert(16)
Single Rotation Example: Insert(16)

Left-Left Case
Node imbalanced due to insertion in left-left grandchild (1 of 4 imbalance cases)
First we did the insertion, which made an imbalanced

Left-Left Case
So we rotate at a, using BST facts:
\( X < b < Y < a < Z \)
A single rotation restores balance at the node
- Node is same height as before insertion, so ancestors now balanced

Right-Right Case
Mirror image to left-left case, so you rotate the other way
- Exact same concept, but different code

The Other Two Cases
Single rotations not enough for insertions left-right or right-left subtree
- Simple example: insert(1), insert(6), insert(3)
First wrong idea: single rotation as before
The Other Two Cases
Single rotations not enough for insertions left-right or right-left subtree
- Simple example: insert(1), insert(6), insert(3)
  Second wrong idea: single rotation on child

Right-Left Case
Height of the subtree after rebalancing is the same as before insert
- No ancestor in the tree will need rebalancing
  Does not have to be implemented as two rotations; can just do:

Left-Right Case
Mirror image of right-left
- No new concepts, just additional code to write

Double Rotation
First attempt at violated the BST property
Second attempt did not fix balance
Double rotation: If we do both, it works!
- Rotate problematic child and grandchild
- Then rotate between self and new child

Memorizing Double Rotations
Easier to remember than you may think:
- Move grandchild to grandparent’s position
- Put grandparent, parent, and subtrees in the only legal position
Double Rotation Example: Insert(5)

Double Rotation Example: Insert(5)

Double Rotation Example: Insert(5)

Double Rotation Example: Insert(5)
**Summarizing Insert**

Insert as in a BST

Check back up path for imbalance for 1 of 4 cases:
- node's left-left grandchild is too tall
- node's left-right grandchild is too tall
- node's right-left grandchild is too tall
- node's right-right grandchild is too tall

Only one case can occur, because tree was balanced before insert

After rotations, the smallest-unbalanced subtree now has the same height as before the insertion
- So all ancestors are now balanced

**Efficiency**

Worst-case complexity of **find**: $O(\log n)$

Worst-case complexity of **insert**: $O(\log n)$
- Rotation is $O(1)$
- There's an $O(\log n)$ path to root
- Even without “one-rotation-is-enough” fact this still means $O(\log n)$ time

Worst-case complexity of **buildTree**: $O(n \log n)$

---

**Delete**

We will not cover delete in detail
- Read the textbook
- May cover in section

Basic idea:
- Do the delete as in a BST
- Where you start the balancing check depends on if a leaf or a node with children was removed
- In latter case, you will start from the predecessor/successor for the balancing check

**delete** is also $O(\log n)$

---

**Balancing Takes a Lot of Work**

To make AVL trees work, we needed:
- Extra info for each node
- Complex logic to detect imbalance
- Recursive bottom-up implementation

Can we do better with less work?

---

**Splay Trees**

Here's an insane idea:
- Let's take the rotating idea of AVL trees but do it without any care (ignore balance)
- Insert/Find always rotate node to the root

Seems crazy, right? But...
- Amortized time per operations is $O(\log n)$
- Worst case time per operation is $O(n)$ but is guaranteed to happen very rarely
Amortized Analysis

If a sequence of M operations takes $O(M f(n))$ time, we say the amortized runtime is $O(f(n))$

- Average time per operation for any sequence is $O(f(n))$
- Worst case time for any sequence of M operations is $O(M f(n))$
- Worst case time per operation can still be large, say $O(n)$

**Amortized complexity is a worst-case guarantee for a sequences of operations**

Interpreting Amortized Analyses

Is amortized guarantee any weaker than worst-case?
Yes, it is only for sequences of operations

Is amortized guarantee stronger than average-case?
Yes, it guarantees no bad sequences

Is average-case guarantee good enough in practice?
No, adversarial input can always happen

Is amortized guarantee good enough in practice?
Yes, due to promise of no bad sequences

The Splay Tree Idea

If you’re forced to make a really deep access:

Since you’re down there anyway, you might as well fix up a lot of deep nodes!

Find/Insert in Splay Trees

1. Find or insert a node $k$
2. **Splay $k$ to the root using:**
   - zig-zag, zig-zig, or plain old zig rotation

Splaying moves multiple nodes higher up in the tree (pushing some down too)

How do we do this?

Naïve Approach

One option is to repeatedly use AVL single rotation until node $k$ becomes the root:

Why this is bad:
- $r$ gets pushed almost as low as $k$ was
- Bad sequence: find($k$), find($r$), find($k$), etc.
Splay: Zig-Zag

Does this look familiar?
It's a double AVL rotation

Blue nodes are Helped
Red nodes are Hurt

Splay: Zig-Zig

Is this just two AVL single rotations in a row?
Not quite. We rotate g & p and then p & k

Blue nodes are Helped
Red nodes are Hurt

Splay: Zig-Zig

Why does this help?
Same number of nodes helped as hurt, but later rotations will help the whole subtree

Blue nodes are Helped
Red nodes are Hurt

Special Case for Root: Zig

Relative depth of p, Y, and Z?
Down one level
Relative depth of everyone else?
Much better!

Why not drop zig-zig and just zig all the way?
No! Zig helps one child subtree. Zig-zig helps two!

Splaying Example: find(6)

Still Splaying 6
Stay on target...

Splay Again: find(4)

Almost there...

Wait a sec...

What happened here?
- Didn’t the two find operations take linear time instead of logarithmic?
- What about the amortized $O(\log n)$ guarantee?

The guarantee still holds
- We must take into account the previous steps used to create this tree.
- The analysis says that some operations may be linear, but they average out in the long run.

Why Splaying Helps

If a node $k$ on the access path is at depth $d$ before the splay
- It’s at about depth $d/2$ after the splay

Overall, nodes which are low on the access path tend to move closer to the root.

Importantly, we fix up/balance the tree every time we do an expensive (deep) access
- This gives splaying its amortized $O(\log n)$ performance (Maybe not now, but soon, and for the rest of the operations)

Further Practical Benefits of Splaying

No heights to maintain/No imbalances to check
- Less storage per node
- Easier to code (seriously!)

Data accessed once is often soon accessed again
- Splaying does implicit caching to the root
- This important idea is known as locality
Splay Operations: find
1. Find the node in normal BST manner
2. Splay the node to the root
   ▪ if node not found, splay what would have been the node’s parent

What if we didn’t splay?
▪ The amortized guarantee would fail!
▪ Consider this sequence with k not in tree: find(k), find(k), find(k), ...
▪ Splaying would make the second find(k) a constant time operation

Splay Operations: Insert
To insert, could do an ordinary BST insert
▪ That would not fix up tree
▪ A BST insert followed by a find and splay?

Better idea: Splay before the insert!
▪ How? A combination of find and split
▪ What's split?

Splitting in Binary Search Trees
split(T, x) creates from T two BSTs L and R:
▪ All elements of T are in either subtree L or R (T = L ∪ R)
▪ All elements in L are ≤ x
▪ All elements in R are ≥ x
▪ L and R share no elements (L ∩ R = ∅)

Back to Insert
insert(x):
▪ Split on x
▪ Join subtrees using x as root

Insert Example: insert(5)
**Splay Operations: Delete**

The other operations splayed, so we’d better do that for delete as well.

`delete(x):`
- find `x` and splay to root
- if `x` is there, remove it
- ...?

Now what?

**Join Operation**

Join(L, R) merges two trees `L < R`
- Splay on the maximum element in `L`
  - then attach `R`

`L` `R`

Similar to BST delete:
find max = find element with no right child

---

**Splay Operations: Delete**

`delete(x):`
- find `x` and splay to root
- if `x` is there, remove it
- join the resulting subtrees

**Delete Example: delete(4)**

`5` `4` `3`

Find max

Why? It’s the fault of hardware!

**B TREES**

Technically, they are called B+ trees but their name was lowered due to concerns of grade inflation.
A Typical Memory Hierarchy

- CPU
  - L1 Cache: 128KB = 2^17
  - L2 Cache: 2MB = 2^21
  - Main memory: 2GB = 2^31
  - Disk: 1TB = 2^40

instructions (e.g., addition): \(2^{30}/\text{sec}\)

- get data in L1: \(2^{21}/\text{sec} = 2 \text{ insns}\)
- get data in L2: \(2^{25}/\text{sec} = 30 \text{ insns}\)
- get data in main memory: \(2^{30}/\text{sec} = 250 \text{ insns}\)
- get data from "new place" on disk: \(2^{27}/\text{sec} = 8,000,000 \text{ insns}\)
  - "streamed": \(2^{18}/\text{sec}\)

Moral of The Story

It is much faster to do:
- 5 million arithmetic ops
- 2500 L2 cache accesses
- 400 main memory accesses

Than:
- 1 disk access
- 1 disk access
- 1 disk access

Accessing the disk is EXPENSIVE!!!

Why are computers built this way?
- Physical realities of speed of light and relative closeness to CPU
- Cost (price per byte of different technologies)
- Disks get much bigger not much faster
  - 7200 RPM spin is slow compared to RAM
  - Disks unlikely to spin faster in the future
- Solid-state drives are faster than disks but still slower due to distance, write performance, etc.
- Speedups at higher levels generally make lower levels relatively slower

Dealing with Latency

Moving data up the memory hierarchy is slow because of latency

We can do better by grabbing surrounding memory with each request
- It is easy to do since we are there anyways
- Likely to be asked for soon (locality of reference)

As defined by the operating system:
- Amount moved from disk to memory is called block or page size
- Amount moved from memory to cache is called the line size

M-ary Search Tree

Build a search tree with branching factor M:
- Have an array of sorted children (Node[])
- Choose M to fit snugly into a disk block (1 access for array)

Perfect tree of height \(h\) has \((M^{h+1}-1)/(M-1)\) nodes

# hops for find: Use \(\log_M n\) to calculate
- If \(M=256\), that's an 8x improvement
- If \(n = 2^{10}\), only 5 levels instead of 40 (5 disk accesses)

Runtime of find if balanced: \(O(\log_J M \log_M n)\)

Problems with M-ary Search Trees

- What should the order property be?
- How would you rebalance (ideally without more disk accesses)?
- Any "useful" data at the internal nodes takes up disk-block space without being used by finds moving past it
- Use the branching-factor idea, but for a different kind of balanced tree
  - Not a binary search tree
  - But still logarithmic height for any \(M > 2\)
**B+ Trees (will just say “B Trees”)**

Two types of nodes:
- **Internal** nodes and **leaf** nodes

Each internal node has room for up to \( M-1 \) keys and \( M \) children.
- All data are at the leaves!

**Order property:**
- Subtree between \( x \) and \( y \)
  - Data that is \( \geq x \) and \( < y \)
- Notice the \( \geq \)

Leaf has up to \( L \) sorted data items

---

**Example**

Suppose: \( M=4 \) (max # children in internal node)
- L=5 (max # data items at leaf)
- All internal nodes have at least 2 children
- All leaves at same depth with at least 3 data items

---

**B Tree Find**

We are used to data at internal nodes
- At an internal node, binary search on the \( M-1 \) keys
- At the leaf do binary search on the \( \leq L \) data items

But find is still an easy root-to-leaf algorithm
- At an internal node, binary search on the \( M-1 \) keys
- At the leaf do binary search on the \( \leq L \) data items

To ensure logarithmic running time, we need to guarantee balance!

What should the balance condition be?
What makes B trees so disk friendly?

Many keys stored in one **internal node**
- All brought into memory in one disk access
- But only if we pick $M$ wisely
- Makes the binary search over $M-1$ keys worth it (insignificant compared to disk access times)

**Internal nodes** contain only keys
- Any `find` wants only one data item; wasteful to load unnecessary items with internal nodes
- Only bring one leaf of data items into memory
- Data-item size does not affect what $M$ is

Maintaining Balance

So this seems like a great data structure

It is

But we haven’t implemented the other dictionary operations yet
- `insert`
- `delete`

As with AVL trees, the hard part is maintaining structure properties

Building a B-Tree

$M = 3 \quad L = 3$

The empty B-Tree (the root will be a leaf at the beginning)

Simply need to keep data sorted

When we ‘overflow’ a leaf, we split it into 2 leaves
- Parent gains another child
- If there is no parent, we create one

How do we pick the new key?
- Smallest element in right subtree
Insertion Algorithm

1. Insert the data in its leaf in sorted order

2. If the leaf now has L+1 items, overflow!
   a. Split the leaf into two nodes:
      - Original leaf with \(\lceil (L+1)/2 \rceil\) smaller items
      - New leaf with \(\lfloor (L+1)/2 \rfloor = \lceil L/2 \rceil\) larger items
   b. Attach the new child to the parent
      - Adding new key to parent in sorted order

3. If Step 2 caused the parent to have M+1 children, overflow the parent!

Worst-Case Efficiency of Insert

Find correct leaf: \(O(1)\)
Insert in leaf: \(O(L)\)
Split leaf: \(O(L)\)
Split parents all the way to root: \(O(M \log_M n)\)
Total: \(O(L + M \log_M n)\)

But it’s not that bad:
- Splits are rare (only if a node is FULL)
- M and L are likely to be large
- After a split, nodes will be half empty
- Splitting the root is thus extremely rare
- Reducing disk accesses is name of the game: inserts are thus \(O(\log_M n)\) on average

Adoption for Insert

We can sometimes avoid splitting via a process called adoption

Example:

- Notice correction by changing parent keys
- Implementation not necessary for efficiency
Deletion

\[ M = 3 \quad L = 3 \]

\[ \text{delete}(32) \]

\[ \begin{array}{c}
\text{18} \\
\text{15} \\
\text{3, 15} \\
\text{12, 16} \\
\text{14, 14} \\
\end{array} \quad \begin{array}{c}
\text{32, 32} \\
\text{30, 36} \\
\text{40, 40} \\
\text{40, 45} \\
\text{40, 45} \\
\end{array} \]

\[ \begin{array}{c}
\text{18} \\
\text{15} \\
\text{3, 15} \\
\text{12, 16} \\
\text{14, 14} \\
\end{array} \quad \begin{array}{c}
\text{32, 32} \\
\text{30, 36} \\
\text{40, 40} \\
\text{40, 45} \\
\text{40, 45} \\
\end{array} \]

Are you using that 14?
Can I borrow it?

\[ M = 3 \quad L = 3 \]

\[ \text{delete}(15) \]

\[ \begin{array}{c}
\text{18} \\
\text{15} \\
\text{3, 15} \\
\text{12, 16} \\
\text{14, 14} \\
\end{array} \quad \begin{array}{c}
\text{32, 32} \\
\text{30, 36} \\
\text{40, 40} \\
\text{40, 45} \\
\text{40, 45} \\
\end{array} \]

Are you using that 14? Yes
Are you using that 18? Yes

\[ M = 3 \quad L = 3 \]

\[ \text{delete}(16) \]

\[ \begin{array}{c}
\text{18} \\
\text{14} \\
\text{14} \\
\text{14} \\
\text{14} \\
\end{array} \quad \begin{array}{c}
\text{3, 16} \\
\text{12, 16} \\
\text{14, 14} \\
\text{14, 14} \\
\text{14, 14} \\
\end{array} \]

Are you using that 12? Yes
Are you using that 18? Yes

\[ M = 3 \quad L = 3 \]

Well, let's just consolidate our leaves since we have the room

Oops. Not enough leaves
Are you using that 18/30?
Delete Algorithm

1. Remove the data from its leaf

2. If the leaf now has $\lceil L/2 \rceil - 1$, underflow!
   - If a neighbor has $>\lceil L/2 \rceil$ items, adopt and update parent
   - Else merge node with neighbor
     - Guaranteed to have a legal number of items $\lceil L/2 \rceil + \lceil L/2 \rceil = L$
     - Parent now has one less node

1. If Step 2 caused parent to have $\lceil M/2 \rceil - 1$ children, underflow!
Deletion Algorithm

4. If an internal node has \( \lceil M/2 \rceil - 1 \) children
   - If a neighbor has \( > \lceil M/2 \rceil \) items, adopt and update parent
   - Else merge node with neighbor
     - Guaranteed to have a legal number of items
     - Parent now has one less node, may need to continue underflowing up the tree

Fine if we merge all the way up to the root
- If the root went from 2 children to 1, delete the root and make child the root
- This is the only case that decreases tree height

Worst-Case Efficiency of Delete

Find correct leaf: \( O(\log_2 M \log_M n) \)
Insert in leaf: \( O(L) \)
Split leaf: \( O(L) \)
Split parents all the way to root: \( O(M \log_M n) \)
Total: \( O(L + M \log_M n) \)
But it's not that bad:
- Merges are not that common
- After a merge, a node will be over half full
- Reducing disk accesses is name of the game: deletions are thus \( O(\log_M n) \) on average

Implementing B Trees in Java?

Assuming our goal is efficient number of disk accesses, Java was not designed for this

This is not a programming languages course

Still, it is worthwhile to know enough about “how Java works” and why this is probably a bad idea for B trees

The key issue is extra levels of indirection...

What that looks like

BTreeNode (3 objects with “header words”)

BTreeLeaf (data objects not in contiguous memory)

The moral

The point of B trees is to keep related data in contiguous memory

All the red references on the previous slide are inappropriate
- As minor point, beware the extra “header words”

But that is “the best you can do” in Java
- Again, the advantage is generic, reusable code
- But for your performance-critical web-index, not the way to implement your B-Tree for terabytes of data

Other languages better support “flattening objects into arrays”
**Conclusion: Balanced Trees**

Balanced trees make good dictionaries because they guarantee logarithmic-time find, insert, and delete

- Essential and beautiful computer science
- But only if you can maintain balance within the time bound and the underlying computer architecture

Another great balanced tree which we sadly will not cover (but easy to read about)

- Red-black trees: all leaves have depth within a factor of 2