Balanced Trees
Part One
Balanced Trees

• Balanced search trees are among the most useful and versatile data structures.

• Many programming languages ship with a balanced tree library.
  • C++: std::map / std::set
  • Java: TreeMap / TreeSet
  • Python: OrderedDict

• Many advanced data structures are layered on top of balanced trees.
  • We'll see them used to build y-Fast Tries later in the quarter. (They’re really really cool, trust me!)
Where We're Going

• **B-Trees**
  • A simple type of balanced tree developed for block storage.

• **Red/Black Trees**
  • The canonical balanced binary search tree.

• **Augmented Search Trees**
  • Adding extra information to balanced trees to supercharge the data structure.

• **Two Advanced Operations**
  • The split and join operations.
Outline for Today

• **BST Review**
  • Refresher on basic BST concepts and runtimes.

• **Overview of Red/Black Trees**
  • What we're building toward.

• **B-Trees and 2-3-4- Trees**
  • A simple balanced tree in depth.

• **Intuiting Red/Black Trees**
  • A much better feel for red/black trees.
A Quick BST Review
Binary Search Trees

- A **binary search tree** is a binary tree with the following properties:
  - Each node in the BST stores a **key**, and optionally, some auxiliary information.
  - The key of every node in a BST is strictly greater than all keys to its left and strictly smaller than all keys to its right.
- The **height** of a binary search tree is the length of the longest path from the root to a leaf, measured in the number of **edges**.
  - A tree with one node has height 0.
  - A tree with no nodes has height -1, by convention.
Runtime Analysis

- The time complexity of all these operations is $O(h)$, where $h$ is the height of the tree.
  - Represents the longest path we can take.
- In the best case, $h = O(\log n)$ and all operations take time $O(\log n)$.
- In the worst case, $h = \Theta(n)$ and some operations will take time $\Theta(n)$.

**Challenge:** How do you efficiently keep the height of a tree low?
A Glimpse of Red/Black Trees
Red/Black Trees

- A red/black tree is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.
Red/Black Trees

- A **red/black tree** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.
Red/Black Trees

- A **red/black tree** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.
Red/Black Trees

- A **red/black tree** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.
Red/Black Trees

A red/black tree is a BST with the following properties:

- Every node is either red or black.
- The root is black.
- No red node has a red child.
- Every root-null path in the tree passes through the same number of black nodes.
Red/Black Trees

• **Theorem:** Any red/black tree with $n$ nodes has height $O(\log n)$.
  • We could prove this now, but there's a much simpler proof of this we'll see later on.

• Given a fixed red/black tree, lookups can be done in time $O(\log n)$. 
Mutating Red/Black Trees

What are we supposed to do with this new node?
Mutating Red/Black Trees
Mutating Red/Black Trees

How do we fix up the black-height property?
Fixing Up Red/Black Trees

- **The Good News:** After doing an insertion or deletion, can locally modify a red/black tree in time $O(\log n)$ to fix up the red/black properties.

- **The Bad News:** There are a lot of cases to consider and they're not trivial.

- Some questions:
  - How do you memorize / remember all the different types of rotations?
  - How on earth did anyone come up with red/black trees in the first place?
B-Trees
Generalizing BSTs

- In a binary search tree, each node stores a single key.
- That key splits the “key space” into two pieces, and each subtree stores the keys in those halves.
Generalizing BSTs

- In a *multiway search tree*, each node stores an arbitrary number of keys in sorted order.

- In a node with $k$ keys splits the “key space” into $k + 1$ pieces, and each subtree stores the keys in those pieces.
A **B-tree of order** $b$ is a multiway search tree with the following properties:

- All leaf nodes are stored at the same depth.
- All non-root nodes have between $b - 1$ and $2b - 1$ keys.
- The root node has been 1 and $2b - 1$ keys.
- All root-null paths through the tree pass through the same number of nodes.
One Solution: **B-Trees**

- A **B-tree of order** \( b \) is a multiway search tree with the following properties:
  - All leaf nodes are stored at the same depth.
  - All non-root nodes have between \( b - 1 \) and \( 2b - 1 \) keys.
  - The root node has been 1 and \( 2b - 1 \) keys.
  - All root-null paths through the tree pass through the same number of nodes.
2-3-4 Trees

- A **2-3-4 tree** is a B-tree of order 2. The rules for 2-3-4 trees are really simple:
  - All leaf nodes are stored at the same depth.
  - All nodes have between 1 and 3 keys (between 2 and 4 children).
  - All root-null paths through the tree pass through the same number of nodes.
- These fellas will make a number of appearances later on. Stay tuned!
The Tradeoff

• Because B-tree nodes can have multiple keys, when performing a search, insertion, or deletion, we have to spend more work inside each node.

• Insertion and deletion can be expensive – for large $b$, we might have to shuffle thousands or millions of keys over!

• Why would you use a B-tree?
Memory Tradeoffs

- There is an enormous tradeoff between speed and size in memory.
- SRAM (the stuff registers are made of) is fast but very expensive:
  - Can keep up with processor speeds in the GHz.
  - As of 2010, cost is $5/MB. *(Anyone know a good source for a more recent price?)*
  - Good luck buying 1TB of the stuff!
- Hard disks are cheap but very slow:
  - As of 2018, you can buy a 4TB hard drive for about $100.
  - As of 2018, good disk seek times for magnetic drives are measured in ms (about two to four million times slower than a processor cycle!)
The Memory Hierarchy

**Idea:** Try to get the best of all worlds by using multiple types of memory.

![Memory Hierarchy Diagram]

- **Registers:** 256B - 8KB, 0.25 - 1ns
- **L1 Cache:** 16KB - 64KB, 1ns - 5ns
- **L2 Cache:** 1MB - 4MB, 5ns - 25ns
- **Main Memory:** 4GB - 256GB, 25ns - 100ns
- **Hard Disk:** 1TB+, 3 - 10ms
- **Network (The Cloud):** Lots, 10 - 2000ms
Why B-Trees?

- Because B-trees have a huge branching factor, they're great for on-disk storage.
  - Disk block reads/writes are glacially slow.
  - The high branching factor minimizes the number of blocks to read during a lookup.
  - Extra work scanning inside a block offset by these savings.
- Major use cases for B-trees and their variants (B+-trees, H-trees, etc.) include
  - databases (huge amount of data stored on disk);
  - file systems (ext4, NTFS, ReFS); and, recently,
  - in-memory data structures (due to cache effects).
The Height of a B-Tree

- What is the maximum possible height of a B-tree of order $b$?

![Diagram of a B-tree]

1

$2(b - 1)$

$2b(b - 1)$

$2b^2(b - 1)$

...
The Height of a B-Tree

- **Theorem:** The maximum height of a B-tree of order $b$ containing $n$ nodes is $\log_b \left( \frac{n + 1}{2} \right)$.

- **Proof:** Number of nodes $n$ in a B-tree of height $h$ is guaranteed to be at least

  \[
  1 + 2(b - 1) + 2b(b - 1) + 2b^2(b - 1) + \ldots + 2b^{h-1}(b - 1)
  \]

  \[
  = 1 + 2(b - 1)(1 + b + b^2 + \ldots + b^{h-1})
  \]

  \[
  = 1 + 2(b - 1)((b^h - 1) / (b - 1))
  \]

  \[
  = 1 + 2(b^h - 1) = 2b^h - 1.
  \]

  Solving $n = 2b^h - 1$ yields $h = \log_b \left( \frac{n + 1}{2} \right)$. ■

- **Corollary:** B-trees of order $b$ have height $\Theta(\log_b n)$. 
Searching in a B-Tree

- Doing a search in a B-tree involves
  - searching the root node for the key, and
  - if it's not found, recursively exploring the correct child.

Time complexity is $O(\log_{b} n \cdot \log_{b} n) = O(\log_{b} n)$.
Searching in a B-Tree

- Doing a search in a B-tree involves
  - searching the root node for the key, and
  - if it's not found, recursively exploring the correct child.
- Using binary search within a given node, can find the key or the correct child in time $O(\log \text{number-of-keys})$.
- Repeat this process $O(\text{tree-height})$ times.
- Time complexity is
  $$O(\log \text{number-of-keys} \cdot \text{tree-height})$$
  $$= O(\log b \cdot \log_b n)$$
  $$= O(\log b \cdot (\log n / \log b))$$
  $$= O(\log n)$$
- Requires reading $O(\log_b n)$ blocks; this more directly accounts for the total runtime.
The Trickier Cases

- What happens if you insert a key into a node that's too full?
- **Idea:** Split the node in two and propagate upward.
- Here's a 2-3-4 tree (each node has 1 to 3 keys).
The Trickier Cases

- What happens if you insert a key into a node that's too full?

- **Idea:** Split the node in two and propagate upward.

- Here's a 2-3-4 tree (each node has 1 to 3 keys).
The Trickier Cases

- What happens if you insert a key into a node that's too full?
- **Idea:** Split the node in two and propagate upward.
- Here's a 2-3-4 tree (each node has 1 to 3 keys).
The Trickier Cases

- What happens if you insert a key into a node that's too full?
- **Idea:** Split the node in two and propagate upward.
- Here's a 2-3-4 tree (each node has 1 to 3 keys).
The Trickier Cases

- What happens if you insert a key into a node that's too full?
  - **Idea:** Split the node in two and propagate upward.
- Here's a 2-3-4 tree (each node has 1 to 3 keys).

**Note:** B-trees grow upward, not downward.
Inserting into a B-Tree

To insert a key into a B-tree:

- Search for the key, insert at the last-visited leaf node.
- If the leaf is too big (contains 2b keys):
  - Split the node into two nodes of size b each.
  - Remove the largest key of the first block and make it the parent of both blocks.
  - Recursively add that node to the parent, possibly triggering more upward splitting.

Time complexity:

- O(b) work per level and O(log_b n) levels.
- Total work: O(b log_b n)
- In terms of blocks read: O(log_b n)
The Trickier Cases

- How do you delete from a leaf that has only $b - 1$ keys?
- **Idea:** Steal keys from an adjacent node, or merge the nodes if both are empty.
- Again, a 2-3-4 tree:
Deleting from a B-Tree

- If not in a leaf, replace the key with its successor from a leaf and delete out of a leaf.
- To delete a key from a node:
  - If the node has more than \( b - 1 \) keys, or if the node is the root, just remove the key.
  - Otherwise, find a sibling node whose shared parent is \( p \).
    - If that sibling has more than \( b - 1 \) keys, move the max/min key from that sibling into \( p \)'s place and \( p \) down into the current node, then remove the key.
    - Otherwise, fuse the node and its sibling into a single node by adding \( p \) into the block, then recursively remove \( p \) from the parent node.
- Work done is \( O(b \log_b n) \): \( O(b) \) work per level times \( O(\log_b n) \) total levels. Requires \( O(\log_b n) \) block reads/writes.
Time-Out for Announcements!
Problem Sets

• Problem Set One solutions are now available up on the course website.
  • We’re working on getting them graded – stay tuned!

• Problem Set Two is due next Tuesday.
  • Have questions? Ask them on Piazza or stop by our office hours!
Back to CS166!
So... red/black trees?
Red/Black Trees

- A **red/black tree** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.
Red/Black Trees

- A **red/black tree** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.
Red/Black Trees

- A **red/black tree** is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.
Red/Black Trees

- A *red/black tree* is a BST with the following properties:
  - Every node is either red or black.
  - The root is black.
  - No red node has a red child.
  - Every root-null path in the tree passes through the same number of black nodes.
Data Structure Isometries

- Red/black trees are an *isometry* of 2-3-4 trees; they represent the structure of 2-3-4 trees in a different way.

- Many data structures can be designed and analyzed in the same way.

- **Huge advantage:** Rather than memorizing a complex list of red/black tree rules, just think about what the equivalent operation on the corresponding 2-3-4 tree would be and simulate it with color flips and rotations.
The Height of a Red/Black Tree

**Theorem:** Any red/black tree with \( n \) nodes has height \( O(\log n) \).

**Proof:** Contract all red nodes into their parent nodes to convert the red/black tree into a 2-3-4 tree. This decreases the height of the tree by at most a factor of two. The resulting 2-3-4 tree has height \( O(\log n) \), so the original red/black tree has height \( 2 \cdot O(\log n) = O(\log n) \). ■
Exploring the Isometry

- Nodes in a 2-3-4 tree are classified into types based on the number of children they can have.
  - **2-nodes** have one key (two children).
  - **3-nodes** have two keys (three children).
  - **4-nodes** have three keys (four children).
- How might these nodes be represented?
Exploring the Isometry
Red/Black Tree Insertion

- **Rule #1:** When inserting a node, if its parent is black, make the node red and stop.

- **Justification:** This simulates inserting a key into an existing 2-node or 3-node.
Tree Rotations

- **Rotate Right**: A → B
- **Rotate Left**: B → A
This applies any time we're inserting a new node into the middle of a “3-node.”

By making observations like these, we can determine how to update a red/black tree after an insertion.
Building Up Rules

- All of the crazy insertion rules on red/black trees make perfect sense if you connect it back to 2-3-4 trees.
- There are lots of cases to consider because there are many different ways you can insert into a red/black tree.
- **Main point:** Simulating the insertion of a key into a node takes time $O(1)$ in all cases. Therefore, since 2-3-4 trees support $O(\log n)$ insertions, red/black trees support $O(\log n)$ insertions.
- The same is true of deletions.
My Advice

• **Do** know how to do B-tree insertions and deletions.
  - You can derive these easily if you remember to split and join nodes.

• **Do** remember the rules for red/black trees and B-trees.
  - These are useful for proving bounds and deriving results.

• **Do** remember the isometry between red/black trees and 2-3-4 trees.
  - Gives immediate intuition for all the red/black tree operations.

• **Don't** memorize the red/black rotations and color flips.
  - This is rarely useful. If you're coding up a red/black tree, just flip open CLRS and translate the pseudocode. 😊
Next Time

- **Augmented Trees**
  - Building data structures on top of balanced BSTs.

- **Splitting and Joining Trees**
  - Two powerful operations on balanced trees.