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 cool, trust me!)
Where We're Going

- **B-Trees (Today)**
  - A simple type of balanced tree developed for block storage.

- **Red/Black Trees (Today/Tuesday)**
  - The canonical balanced binary search tree.

- **Augmented Search Trees (Tuesday)**
  - Adding extra information to balanced trees to supercharge the data structure.
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.
Searching a BST
Searching a BST
Searching a BST
Searching a BST
Searching a BST
Searching a BST
Searching a BST

- 137
  - 73
    - 42
    - 60
  - 271
    - 161
    - 314
Searching a BST
Searching a BST

- 137
- 73
- 271
- 42
- 161
- 314
- 60
Searching a BST
Inserting into a BST
Inserting into a BST

```
             137
           /    
          73     271
         /   
        42   161   314
         /   
        60   
```

Inserting into a BST
Inserting into a BST
Inserting into a BST
Inserting into a BST
Inserting into a BST
Inserting into a BST

```
    137
   /   /
  73   271
 /    /    /
42   161   314
   /    \
  60    166
```
Deleting from a BST
Deleting from a BST

Delete 60 from this tree, then 73, and then 137.
Deleting from a BST
Deleting from a BST
Deleting from a BST
Deleting from a BST
Deleting from a BST
Case 0: If the node has just no children, just remove it.
Deleting from a BST
Deleting from a BST

```
    137
   /
  73   271
 /     /
42     161
     /
     314
```

166
Deleting from a BST

```
        137
       /   
      271   161
     /     / 
    314   166
   /     
  42     
```
Deleting from a BST
Deleting from a BST
Dealing from a BST

Case 1: If the node has just one child, remove it and replace it with its child.
Deleting from a BST
Deleting from a BST

```
137
/    /
42   271
\   /  \\
  161 \ 314
   /   \\
 166
```
Deleting from a BST

```
42

271
  161
  166
  314
```
Deleting from a BST

```
42
```

```
271
  161
  314
  166
```
Deleting from a BST
Deleting from a BST

```
137
/   \
42   271
   /   \
  161 314
     /   \
    166  
```
Deleting from a BST
Deleting from a BST
Deleting from a BST
Deleting from a BST

```
  161
 /   \
42    271
   /   /   /
   166 314
```
Deleting from a BST
Deleting from a BST

**Case 2:** If the node has two children, find its inorder successor (which has zero or one child), replace the node's key with its successor's key, then delete its successor.
Runtime Analysis

- The time complexity of all these operations is \( O(h) \), where \( h \) is the height of the tree.
  - That’s 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.
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.
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

![Red/Black Tree Diagram]
Mutating Red/Black Trees
Mutating Red/Black Trees
Mutating Red/Black Trees
Mutating Red/Black Trees
Mutating Red/Black Trees
Mutating Red/Black Trees
Mutating Red/Black Trees

```
    17
   /   \
  7     31
 /     /
3      23
|      /
11  37
```

Black nodes are shown as red for emphasis.
What are we supposed to do with this new node?
Mutating Red/Black Trees
Mutating Red/Black Trees
Mutating Red/Black Trees
Mutating Red/Black Trees
Mutating Red/Black Trees
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 rules for fixing up the tree?
  - How on earth did anyone come up with red/black trees in the first place?
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 rules for fixing up the tree?

- 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.
- A node with \( k \) keys splits the key space into \( k+1 \) regions, with subtrees for keys in each region.
Generalizing BSTs

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

- Surprisingly, it’s a bit easier to build a balanced multiway tree than it is to build a balanced BST. Let’s see how.
Balanced Multiway Trees

• In some sense, building a balanced multiway tree isn’t all that hard.

• We can always just cram more keys into a single node!
Balanced Multiway Trees

- In some sense, building a balanced multiway tree isn’t all that hard.
- We can always just cram more keys into a single node!
Balanced Multiway Trees

• In some sense, building a balanced multiway tree isn’t all that hard.

• We can always just cram more keys into a single node!
Balanced Multiway Trees

• In some sense, building a balanced multiway tree isn’t all that hard.
• We can always just cram more keys into a single node!
Balanced Multiway Trees

• In some sense, building a balanced multiway tree isn’t all that hard.
• We can always just cram more keys into a single node!
Balanced Multiway Trees

- In some sense, building a balanced multiway tree isn’t all that hard.
- We can always just cram more keys into a single node!
Balanced Multiway Trees

- In some sense, building a balanced multiway tree isn’t all that hard.
- We can always just cram more keys into a single node!
Balanced Multiway Trees

- In some sense, building a balanced multiway tree isn’t all that hard.
- We can always just cram more keys into a single node!

\[
\begin{array}{cccccccc}
26 & 31 & 41 & 53 & 58 & 59 & 97 \\
\end{array}
\]
Balanced Multiway Trees

- In some sense, building a balanced multiway tree isn’t all that hard.
- We can always just cram more keys into a single node!
Balanced Multiway Trees

- In some sense, building a balanced multiway tree isn’t all that hard.
- We can always just cram more keys into a single node!
Balanced Multiway Trees

- In some sense, building a balanced multiway tree isn’t all that hard.
- We can always just cram more keys into a single node!
Balanced Multiway Trees

• In some sense, building a balanced multiway tree isn’t all that hard.

• We can always just cram more keys into a single node!

23 26 31 41 53 58 59 62 84 93 97
Balanced Multiway Trees

• In some sense, building a balanced multiway tree isn’t all that hard.

• We can always just cram more keys into a single node!

• At a certain point, this stops being a good idea – it’s basically just a sorted array. What does “balance” even mean here?
Balanced Multiway Trees

- What could we do if our nodes get too big?
Balanced Multiway Trees

- What could we do if our nodes get too big?

- **Option 1:** Push keys down into new nodes.
Balanced Multiway Trees

- What could we do if our nodes get too big?
- **Option 1:** Push keys down into new nodes.
Balanced Multiway Trees

- What could we do if our nodes get too big?
- **Option 1:** Push keys down into new nodes.

| 23 26 31 41 53 58 59 62 84 93 97 |
Balanced Multiway Trees

- What could we do if our nodes get too big?
- **Option 1:** Push keys down into new nodes.
Balanced Multiway Trees

- What could we do if our nodes get too big?
  - **Option 1:** Push keys down into new nodes.
  - **Option 2:** Split big nodes, kicking keys higher up.
Balanced Multiway Trees

- What could we do if our nodes get too big?
  - **Option 1:** Push keys down into new nodes.
  - **Option 2:** Split big nodes, kicking keys higher up.
Balanced Multiway Trees

- What could we do if our nodes get too big?
  - **Option 1**: Push keys down into new nodes.
  - **Option 2**: Split big nodes, kicking keys higher up.
What could we do if our nodes get too big?

**Option 1:** Push keys down into new nodes.

**Option 2:** Split big nodes, kicking keys higher up.

What are some advantages of each approach? What are some disadvantages?
Balanced Multiway Trees

- What could we do if our nodes get too big?
  - **Option 1:** Push keys down into new nodes.
  - **Option 2:** Split big nodes, kicking keys higher up.
- Let’s assume that, during an insertion, we add keys to the deepest node possible.
- How do these options compare?
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

10 99 50 20 40 30 31 39 35 32 33 34
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2**: Split big nodes, kicking values higher up.
  - Keeps the tree balanced.
  - Keeps most nodes near the bottom.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

![Node values](image)
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

![Tree diagram with nodes 10, 50, 99, 20, 40, 30, 31, 39, 35, 32, 33, 34]
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

```
10 50 99
   20
```

```
40 30 31 39 35 32 33 34
```
Balanced Multiway Trees

• **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

```
10 50 99

20 30 40
```

```
31 39 35 32 33 34
```
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- Option 2: Split big nodes, kicking values higher up.
  - Keeps the tree balanced.
  - Keeps most nodes near the bottom.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

```
10 50 99
```
```
20 30 40
```
```
31
```
```
39 35 32 33 34
```
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
Balanced Multiway Trees

**Option 1:** Push keys down into new nodes.
- Simple to implement.
- Can lead to tree imbalances.

```
10 50 99
```
```
20 30 40
```
```
31 35 39
```
```
32 33 34
```
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

```
+---+---+---+
| 10|50|99 |
+---+---+---+
    |   |
+---+---+---+
| 20|30|40 |
+---+---+---+
    |   |
+---+---+---+
| 31|32|35|39 |
+---+---+---+
```

33 34
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

```
10 50 99
20 30 40
31 35 39
32
```

```
33 34
```
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

```
       10 50 99
       ↓    ↓
     20 30 40
     ↓    ↓
   31 35 39
   ↓
  32
```

```
       33
```

```
       34
```
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

![Tree diagram]

- **Option 2**: Split big nodes, kicking values higher up.
  - Keeps the tree balanced.
  - Keeps most nodes near the bottom.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

```
10 50 99
```
```
20 30 40
```
```
31 35 39
```
```
32 33 34
```
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2**: Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2**: Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2**: Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2**: Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1**: Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2**: Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.
- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.

Each existing node’s depth just increased by one.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **Option 1:** Push keys down into new nodes.
  - Simple to implement.
  - Can lead to tree imbalances.

- **Option 2:** Split big nodes, kicking keys higher up.
  - Keeps the tree balanced.
  - Slightly trickier to implement.
Balanced Multiway Trees

- **General idea:** Cap the maximum number of keys in a node. Add keys into leaves. Whenever a node gets too big, split it and kick one key higher up the tree.

- **Advantage 1:** The tree is always balanced.
- **Advantage 2:** Insertions and lookups are pretty fast.
Balanced Multiway Trees

- We currently have a *mechanical description* of how these balanced multiway trees work:
  - Cap the size of each node.
  - Add keys into leaves.
  - Split nodes when they get too big and propagate the splits upward.
- We currently don’t have an *operational definition* of how these balanced multiway trees work.
  - e.g. “A Cartesian tree for an array is a binary tree that’s a min-heap and whose inorder traversal gives back the original array.”
  - e.g. “A suffix tree is a Patricia trie with one node for each suffix and branching word of $T$.”
B-Trees

- A **B-tree of order** $b$ is a multiway search tree where
  - each node has (roughly) between $b$ and $2b$ keys, except the root, which may only have between 1 and $2b$ keys;
  - each node is either a leaf or has one more child than key; and
  - all leaves are at the same depth.

- Different authors give different bounds on how many keys can be in each node. The ranges are often $[b-1, 2b-1]$ or $[b, 2b]$. For the purposes of today’s lecture, we’ll use the range $[b-1, 2b-1]$ for the key limits, just for simplicity.
Analyzing Multiway Trees
The Height of a B-Tree

- What is the maximum possible height of a B-tree of order $b$ that holds $n$ keys?

**Intuition:** The branching factor of the tree is at least $b$, so the number of keys per level grows exponentially in $b$. Therefore, we’d expect something along the lines of $O(\log_b n)$.
The Height of a B-Tree

- What is the maximum possible height of a B-tree of order $b$ that holds $n$ keys?

$$2^{h-1}(b - 1)$$
The Height of a B-Tree

- **Theorem:** The maximum height of a B-tree of order $b$ containing $n$ keys is $O(\log_b n)$.

- **Proof:** Number of keys $n$ 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 $h = \log_b ((n + 1) / 2)$, so the height is $O(\log_b n)$. ■
Analyzing Efficiency

• Suppose we have a B-tree of order $b$.
• What is the worst-case runtime of looking up a key in the B-tree?
• **Answer:** It depends on how we do the search!
Analyzing Efficiency

- To do a lookup in a B-tree, we need to determine which child tree to descend into.
- This means we need to compare our query key against the keys in the node.
- **Question:** How should we do this?
Analyzing Efficiency

- **Option 1**: Use a linear search!
- Cost per node: \(O(b)\).
- Nodes visited: \(O(\log_b n)\).
- Total cost:
  
  \[
  O(b) \cdot O(\log_b n) = O(b \log_b n)
  \]
Analyzing Efficiency

- **Option 2**: Use a binary search!
- Cost per node: \( O(\log b) \).
- Nodes visited: \( O(\log_b n) \).
- Total cost:
  \[
  O(\log b) \cdot O(\log_b n)
  = O(\log b \cdot \log_b n)
  = O(\log b \cdot (\log n) / (\log b))
  = O(\log n).
  \]

**Intuition**: We can’t do better than \( O(\log n) \) for arbitrary data, because it’s the information-theoretic minimum number of comparisons needed to find something in a sorted collection!
Suppose we have a B-tree of order $b$.

What is the worst-case runtime of inserting a key into the B-tree?

Each insertion visits $O(\log_b n)$ nodes, and in the worst case we have to split every node we see.

**Answer:** $O(b \log_b n)$. 
Analyzing Efficiency

- The cost of an insertion in a B-tree of order $b$ is $O(b \log_b n)$.
- What's the best choice of $b$ to use here?
- Note that

\[
\begin{align*}
    b \log_b n &= b \left( \frac{\log n}{\log b} \right) \\
    &= \left( \frac{b}{\log b} \right) \log n.
\end{align*}
\]

- What choice of $b$ minimizes $b / \log b$?
- **Answer:** Pick $b = e$.

**Fun fact:** This is the same time bound you’d get if you used a $b$-ary heap instead of a binary heap for a priority queue.
A 2-3-4 tree is a B-tree of order 2. Specifically:

- each node has between 1 and 3 keys;
- each node is either a leaf or has one more child than key; and
- all leaves are at the same depth.

You actually saw this B-tree earlier! It’s the type of tree from our insertion example.
The Story So Far

• A B-tree supports
  • lookups in time $O(\log n)$, and
  • insertions in time $O(b \log_b n)$.
• Picking $b$ to be around 2 or 3 makes this optimal in Theoryland.
  • The 2-3-4 tree is great for that reason.
• **Plot Twist:** In practice, you most often see choices of $b$ like 1,024 or 4,096.
• **Question:** Why would anyone do that?
Theoryland

IRL
The Memory Hierarchy
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.
  • SRAM units can’t be easily combined together; increasing sizes require better nanofabrication techniques (difficult, expensive!)

• Hard disks are cheap but very slow:
  • As of 2020, you can buy a 4TB hard drive for about $70.
  • As of 2020, 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.
The Memory Hierarchy

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

<table>
<thead>
<tr>
<th>Memory Level</th>
<th>Size Range</th>
<th>Access Time Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>256B - 8KB</td>
<td>0.25 – 1ns</td>
</tr>
<tr>
<td>L1 Cache</td>
<td>16KB - 64KB</td>
<td>1ns – 5ns</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>1MB - 4MB</td>
<td>5ns – 25ns</td>
</tr>
<tr>
<td>Main Memory</td>
<td>4GB - 256GB</td>
<td>25ns – 100ns</td>
</tr>
<tr>
<td>Hard Disk</td>
<td>1TB+</td>
<td>3 – 10ms</td>
</tr>
<tr>
<td>Network (The Cloud)</td>
<td>Lots</td>
<td>10 – 2000ms</td>
</tr>
</tbody>
</table>
The Memory Hierarchy

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

- **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
Suppose you have a data set that’s \textit{way} too big to fit in RAM. The data structure is on disk and read into RAM as needed. Data from disk doesn’t come back one \textit{byte} at a time, but rather one \textit{page} at a time. 

\textbf{Goal:} Minimize the number of disk reads and writes, not the number of instructions executed.
Suppose you have a data set that’s way too big to fit in RAM.
The data structure is on disk and read into RAM as needed.
Data from disk doesn’t come back one byte at a time, but rather one page at a time.

**Goal:** Minimize the number of disk reads and writes, not the number of instructions executed.

Calculate...
Think...
Compute...
External Data Structures

- Suppose you have a data set that’s *way* too big to fit in RAM.
- The data structure is on disk and read into RAM as needed.
- Data from disk doesn’t come back one *byte* at a time, but rather one *page* at a time.
- **Goal:** Minimize the number of disk reads and writes, not the number of instructions executed.

“Please give me 4KB starting at location *addr2*”

001101010001010001010001…
External Data Structures

- Suppose you have a data set that’s *way* too big to fit in RAM.
- The data structure is on disk and read into RAM as needed.
- Data from disk doesn’t come back one *byte* at a time, but rather one *page* at a time.
- **Goal:** Minimize the number of disk reads and writes, not the number of instructions executed.
Suppose you have a data set that’s way too big to fit in RAM.

The data structure is on disk and read into RAM as needed.

Data from disk doesn’t come back one byte at a time, but rather one page at a time.

**Goal:** Minimize the number of disk reads and writes, not the number of instructions executed.
Analyzing B-Trees

- Suppose we tune $b$ so that each node in the B-tree fits inside a single disk page.
- We only care about the number of disk pages read or written.
  - It’s so much slower than RAM that it’ll dominate the runtime.
- **Question:** What is the cost of a lookup in a B-tree in this model?

- **Question:** What is the cost of inserting into a B-tree in this model?
Analyzing B-Trees

• Suppose we tune $b$ so that each node in the B-tree fits inside a single disk page.

• We only care about the number of disk pages read or written.
  • It’s so much slower than RAM that it’ll dominate the runtime.

• **Question:** What is the cost of a lookup in a B-tree in this model?
  • Answer: The height of the tree, $O(\log_b n)$.

• **Question:** What is the cost of inserting into a B-tree in this model?
  • Answer: The height of the tree, $O(\log_b n)$. 
External Data Structures

• 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).
Analyzing B-Trees

• The cost model we use will change our overall analysis.

• Cost is number of operations:
  \[ O(\log n) \text{ per lookup}, \; O(b \log_b n) \text{ per insertion.} \]

• Cost is number of blocks accessed:
  \[ O(\log_b n) \text{ per lookup}, \; O(\log_b n) \text{ per insertion.} \]

• Going forward, we’ll use operation counts as our cost model, though looking at caching effects of data structures would make for an awesome final project!
The Story So Far

- We’ve just built a simple, elegant, balanced multiway tree structure.
- We can use them as balanced trees in main memory (2-3-4 trees).
- We can use them to store huge quantities of information on disk (B-trees).
- We’ve seen that different cost models are appropriate in different situations.
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.

```
1  2  3  4  5  6  7  8  9  10  11  12
```
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.
- After we hoist red nodes into their parents:
  - Each “meta node” has 1, 2, or 3 keys in it. (No red node has a red child.)
  - Each “meta node” is either a leaf or has one more child than key. (Root-null path property.)
  - Each “meta leaf” is at the same depth. (Root-null path property.)

This is a 2-3-4 tree!
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 BST operations.
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 \( 2 \cdot O(\log n) = O(\log n) \). ■
Next Time

• **Deriving Red/Black Trees**
  • Figuring out rules for red/black trees using our isometry.

• **Tree Rotations**
  • A key operation on binary search trees.

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