Tree Indexing

Introduction

A real-world database typically exhibits the following characteristics:

- Records are frequently updated
- A search can be performed using one or more keys
- Range and min/max queries are performed

Facing the Facts

- Linear indexing is not efficient for **updating**
- Hash tables are not efficient for **range queries**
- So... Tree Indexing?

BST Indexing

Use a BST to store primary and secondary indices?

- $O(\log n)$ to look up an index ✅
- $O(\log n)$ to perform a range query ✅
- $O(\log n)$ to insert or delete a record ✅

…assuming we fit it all in program memory.

Big BST Indexing

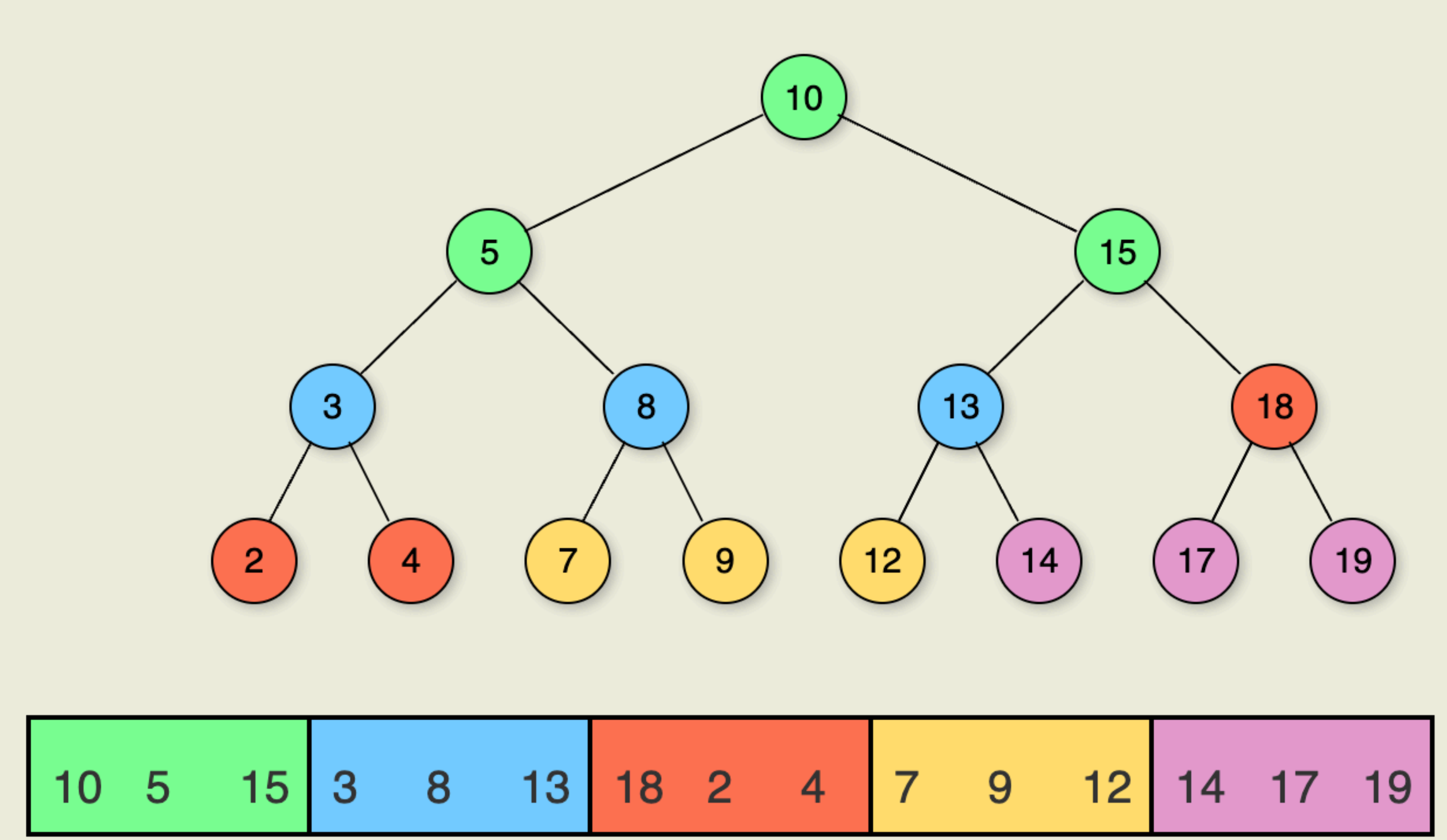
If a database is big enough, the index won't fit in program memory!

- Every time a BST node B is visited, it is necessary to visit all nodes along the path from the root to B
- Each node on this path must be retrieved from disk.
- Each disk access returns a block of information. 🐌
    - If a node is on the same block as its parent, then the cost to find that node is trivial once its parent is in main memory. 🐇

Still $O(\log n)$, but disk reads are **1,000,000X** slower than RAM operations.

Activity

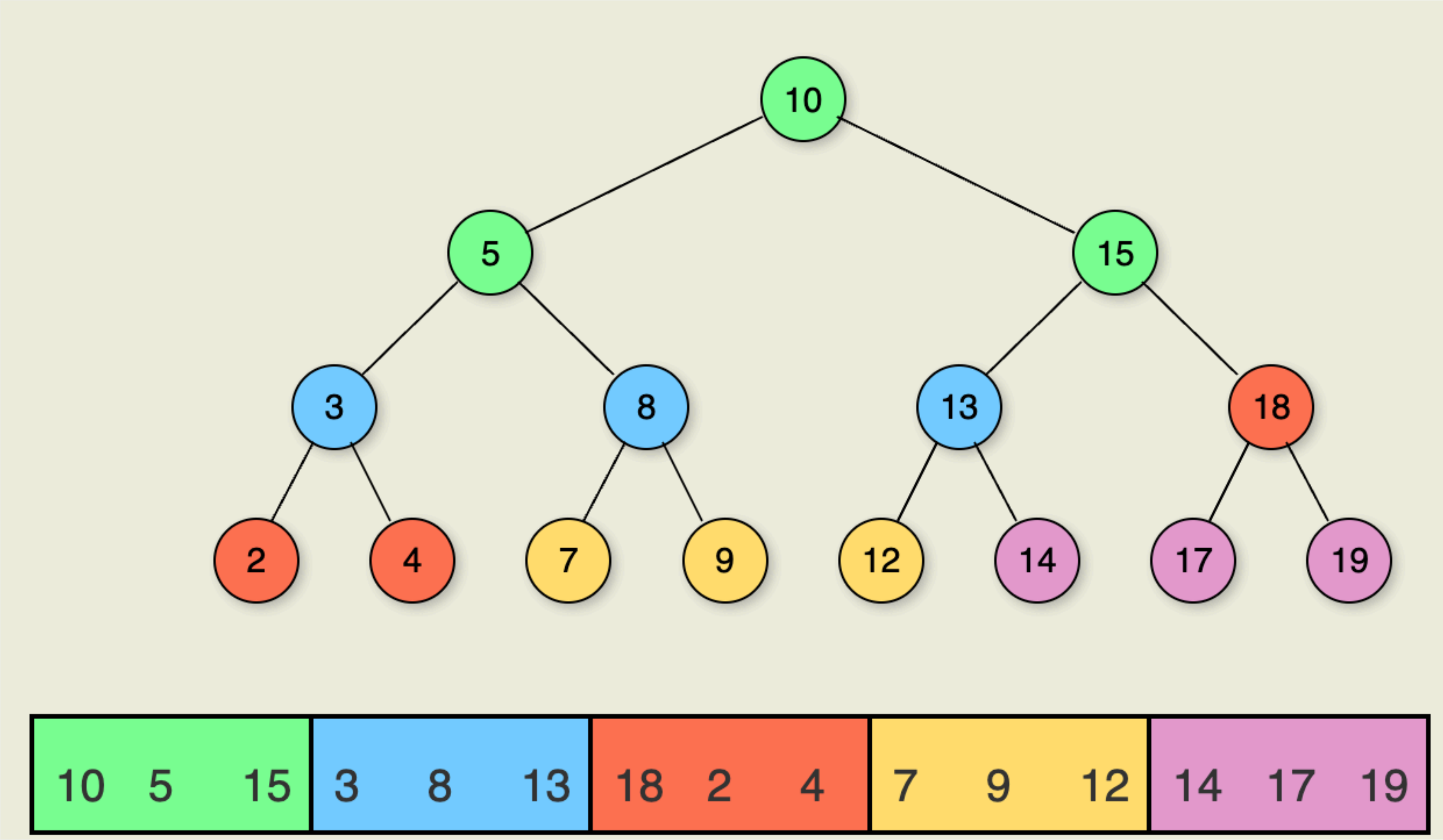
Number of block reads for finding key 12? (Nodes of the same color are found in the same block.)



10	5	15	3	8	13	18	2	4	7	9	12	14	17	19

Activity

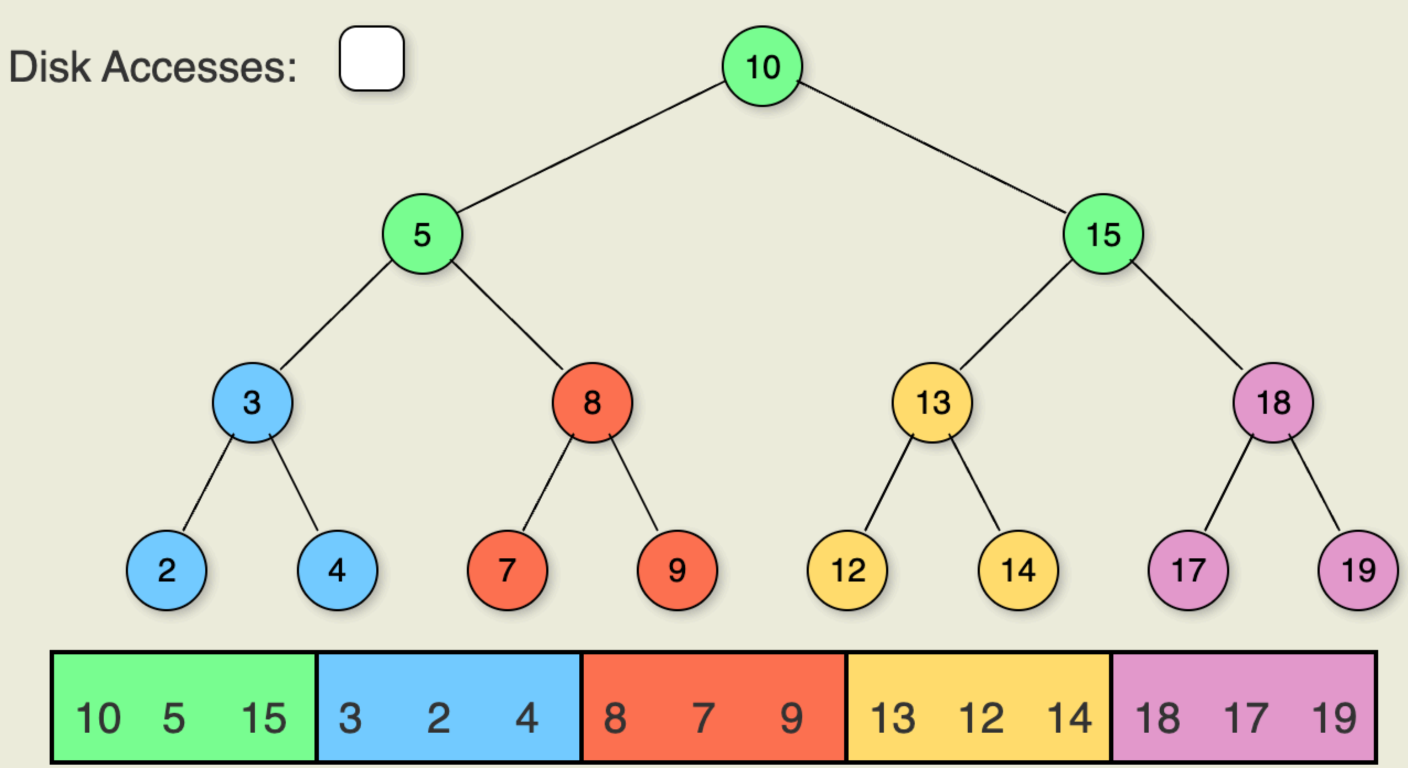
Assuming that a block can store only three keys, what would be the coloring of these nodes that leads to the lowest expected number of block reads per lookup?



10 5 15	3 8 13	18 2 4	7 9 12	14 17 19

Minimizing Block Reads

We prefer a structure that puts parents & children in the same block!



BST Indexing (challenges)

The BST must remain balanced after insertions and deletions

- Need to rearrange data within the tree to maintain $O(\log n)$ height.
- Difficult to maintain good block arrangement if nodes get rearranged.

➡️ We need a different tree structure.

2-3 Tree

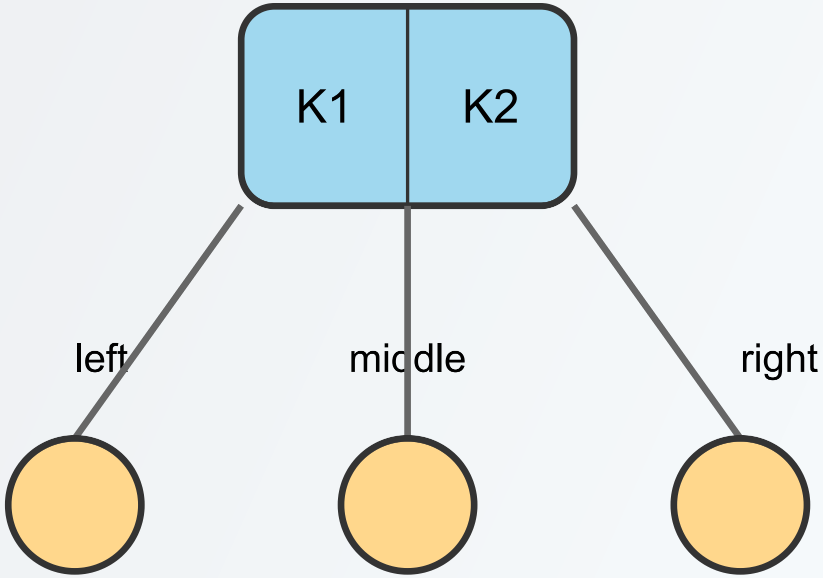
- Not a Binary Tree
- A node contains one or two keys
- Every internal node has either two children (if it contains one key) or three children (if it contains two keys)
- All leaves are at the same level in the tree, so the tree is always height-balanced

2-3 Tree Properties

For every node:

- The values in `left` are less than `K1`
- The values in `middle` are greater than `K1` and less than `K2`
- The values in `right` are greater than `K2`

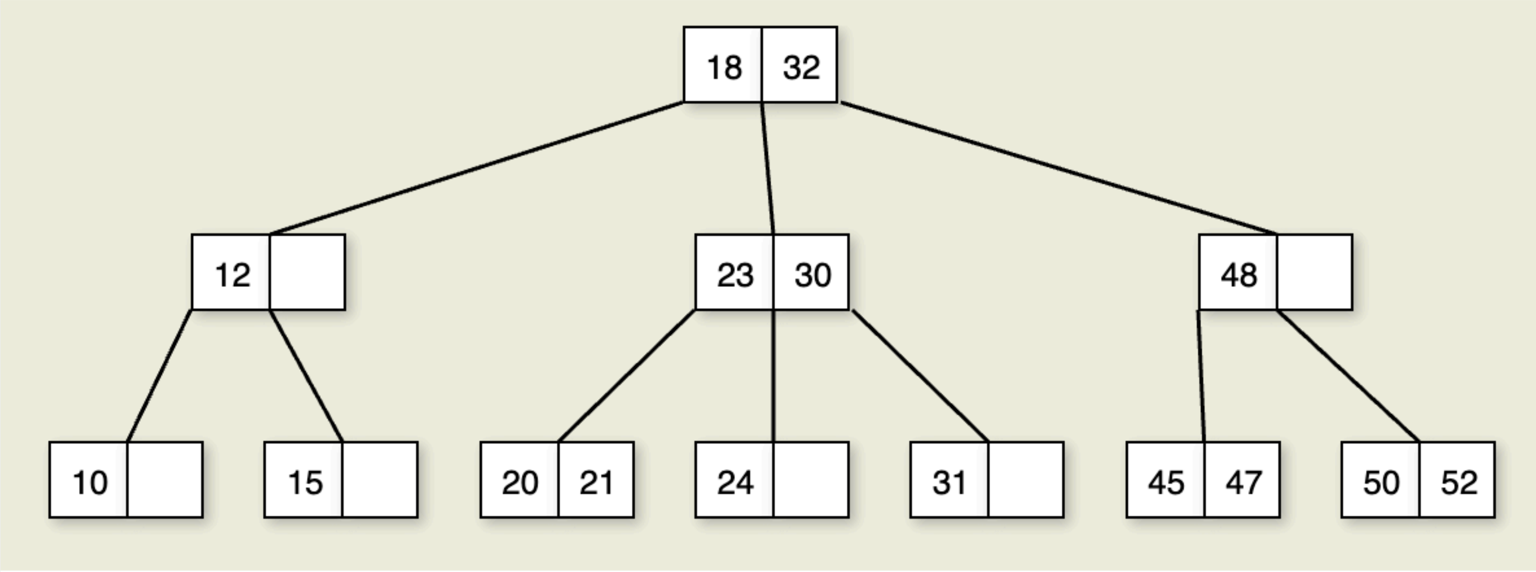
2-3 Tree Node



A 2-3 node with two keys (K1, K2) and three pointers

2-3 Tree Search

- Start at the root.
- While ∃ a node to search, check its keys.
- If a match is not found, proceed with search in the correct subtree.
- Continue until a null node is reached (failure) or the target key is found (success)



2-3 Tree Search

It's BST search, but with more options per step.

```
public E find(TTNode<Key, E> root, Key k) {
    if (root == null) return null;
    if (k.compareTo(root.lkey()) == 0) return root.lval();
    if (root.rkey() != null && k.compareTo(root.rkey()) == 0) {
        return root.rval();
    }
    if (k.compareTo(root.lkey()) < 0) {
        return find(root.left(), k);
    } else if (root.rkey() == null || k.compareTo(root.rkey()) < 0) {
        return find(root.middle(), k);
    } else {
        return find(root.right(), k);
    }
}
```

Shape of a 2-3 Tree

- All leaves are on the same level, so the tree is **height balanced.**
- Height balanced trees have an upper bound on their height of $O(\log_b n)$, where b is the **branching factor**, or number of children per node.
- Search in a 2-3 tree of height $O(\log n)$ takes $O(\log n)$ time.

Key challenge, therefore, is to keep the 2-3 tree height balanced always.

2-3 Tree: Insertion

- New record is stored in a leaf node (like BST)
- Algorithm:
    - Find correct leaf node L where the new key belongs
    - If that node has space, add the record there.
    - If that node doesn't have space, SPLIT and PROMOTE

SPLIT

Inserting `30`...

```
    [23 | ]
   /    |
[14| ] [29|33]
```

⬇⬇⬇⬇⬇

```
    [23   |    ]
   /
[14| ] [29| ] [33| ]
```

The middle element, `30` will now be PROMOTED.

PROMOTE

```
    [23   | 30  ]
   /     /     \
[14| ] [29| ] [33| ]
```

Split & Promote

Split: If the leaf node is full, there are two keys present and one key to be inserted. Keep the smallest key in the existing leaf, create a new leaf for the greatest key, and **promote** the middle key.

Promote: Inserting a middle key into the parent node, recursively **splitting** and **promoting**.

Effects

Like a BST, the new key is always inserted into a leaf node.

Unlike a BST, we never create a leaf node that is deeper than the other leaf nodes.

- When does the depth of a leaf change? When we have to split the root!
- But then all leaves get pushed down one level deeper.

So, height of the tree increases **rarely** and balance is maintained **always**.

➡️ $O(\log n)$ time to insert and maintain $O(\log n)$ height.

Deletion

Three major cases:

1. Deletion from a leaf with two keys
2. Deletion from a leaf with one key
3. Deletion from an internal node

If **(1)**, just remove the key. If **(2) or (3)** find a replacement within the tree, and then recursively delete that replacement from its previous position.

B Tree

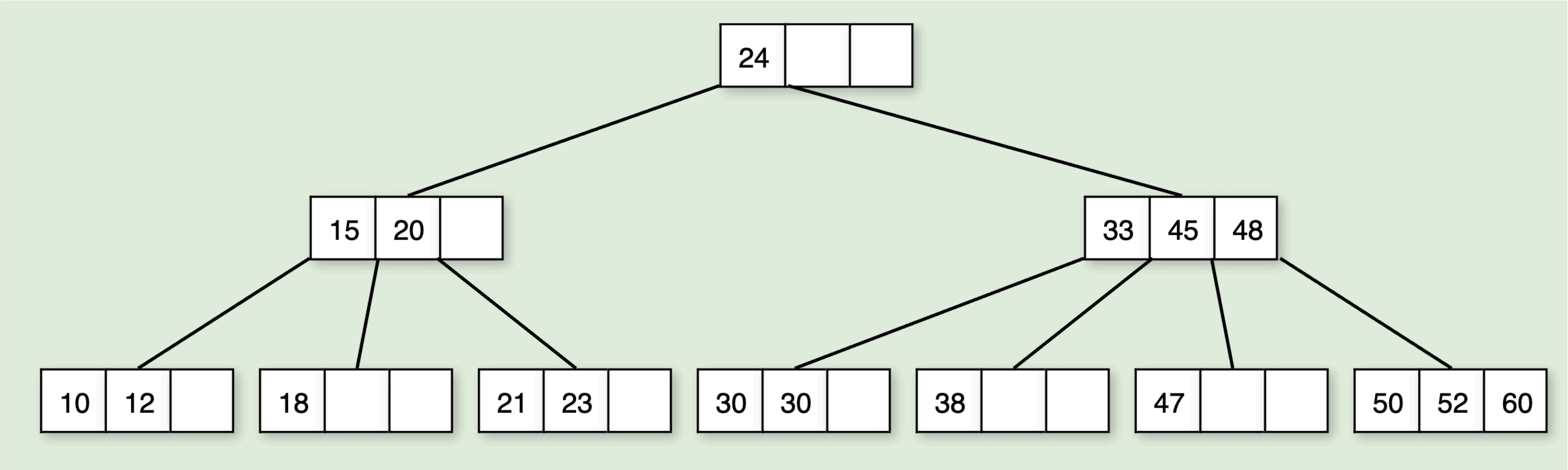
- Generalization of the 2-3 tree
- Invented by R. Bayer and E. McCreight
- Used to implement most modern file systems (Linux, Windows, Apple)
- Used to index tables in relational database management systems

B-Tree: Main Idea

- Keep all leaves at the same level like a 2-3 tree
- Set the size of node to be the size of a block—probably get lots of keys per node this way!
- If a block stores m keys, we will have between $\lceil m/2 \rceil$ and m children per internal node.

Few block reads! Low tree height! Wow!

B-Tree of Order Four



It's like a 2-3 tree, but each node is one bigger.

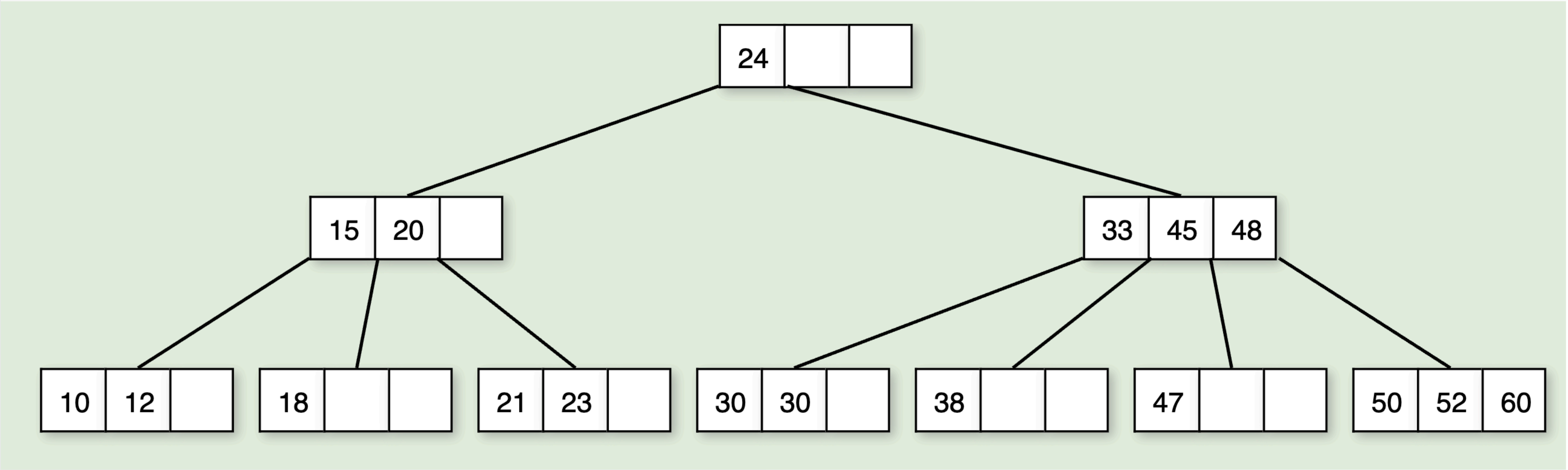
B-Tree Search

- Start at the root.
- While ∃ a node to search, check its keys using **binary search**—nodes are sorted.
- If a match is not found, proceed with search in the correct subtree.
- Continue until a null node is reached (failure) or the target key is found (success)

It's the same as the 2-3 tree search!

B-Tree of Order Four

Find 47!



B-Tree Search

Trees are still height-balanced with a height of $O(\log_m n)$

- Still get $O(\log n)$ runtime for search
- The bigger the value of m, the shallower the tree (improvement by a constant factor)
- The bigger values of m do mean slower search *within a node*, but...
  - do binary search instead of linear search
  - the node size is constant, so who cares

B-Tree: Insertion

- New record is stored in a leaf node
- Algorithm:
    - Find correct leaf node L where the new key belongs
    - If that node has space, add the record there.
    - If that node doesn't have space, SPLIT and PROMOTE

It's the same, but SPLIT is just a little different

B-Tree Split & Promote

Split: If the leaf node is full, there are m keys present and one key to be inserted. Keep the lower half of keys in the existing leaf, create a new leaf for the upper half of key, and **promote** the middle key.

Promote: Inserting a middle key into the parent node, recursively **splitting** and **promoting**.

Same effect of inserting leaves at the same level to maintain balance and rarely increase height.

Big Ideas

We can optimize a tree index by expanding the node size

- fewer block reads
- $O(\log n)$ many lookups required

A Linear Index was still very helpful for:

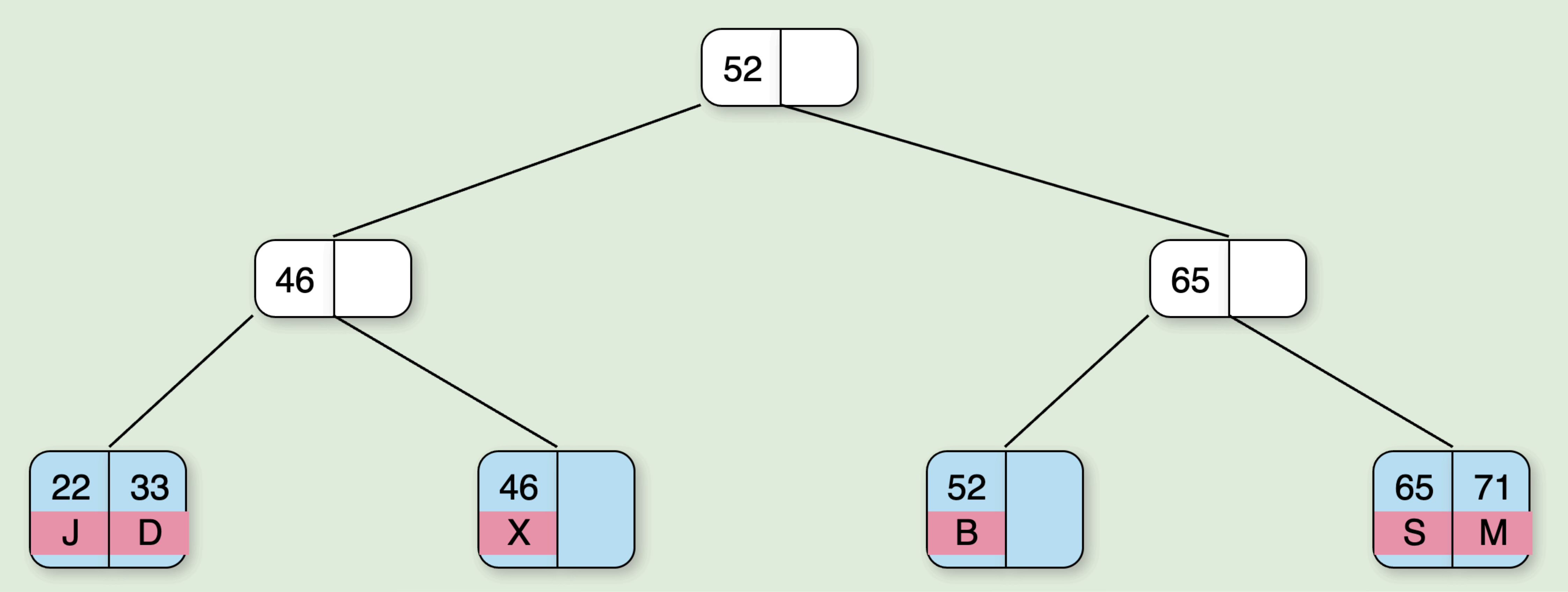
- range queries
- low overhead
- $O(\log n)$ search when sorted

Can we combine the two ideas?

B+ Tree (structure)

- B+ tree stores records only at the leaf nodes
- Internal nodes store search keys
    - used to guide the search
- The leaf nodes store the records and are linked together to form a doubly linked list
    - The entire collection of records can be traversed in sorted order by visiting all the leaf nodes on the linked list

B+ Tree of Order 3



B+ Tree: Analysis

- All operations run in $O(\log_b n)$
- The base of the log is the (average) branching factor of the tree
- Database applications use extremely high branching factors, >=100
  - b = 100 implies that a B+ tree with a height of four stores between 250k and 100 million records
  - Overhead? Not so much! ~1/100 of nodes will be internal nodes
    - 1/k rule for k-ary trees

B+ Tree: Analysis (continued)

To minimize disk accesses:

- Upper levels (internal nodes) of the B+ tree stored in memory
- Internal nodes require little space (do not store records)
- Fewer internal nodes
- Leaf nodes stored on disk

Leaf nodes resemble a linked list divided into blocks such that insertion and deletion only requires shifting of about a block's worth of elements instead of potentially all n.