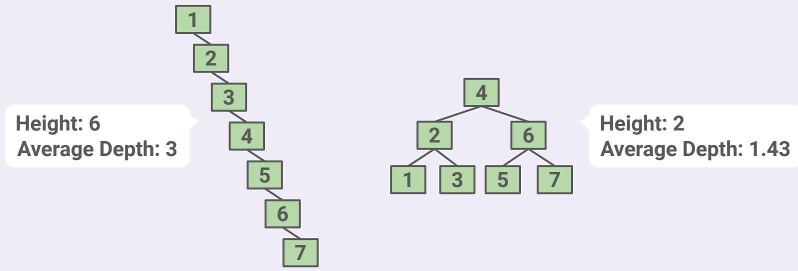
Best Case and Worst Case Height

Suppose we want to build a BST out of the numbers 1, 2, 3, 4, 5, 6, 7.

Give a sequence of add operations that result in (1) a **spindly tree** and (2) a **bushy tree**.



Q1: Give a sequence of add operations that result in (1) a **spindly tree** and (2) a **bushy tree**.

Good News and Bad News

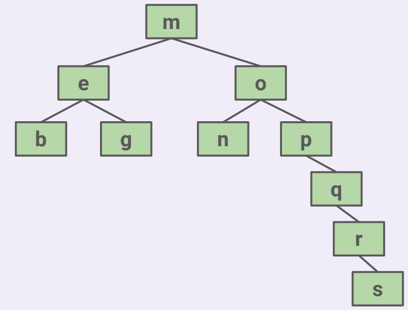
Good news.

BSTs have great a runtime if we insert keys randomly.

Θ(log N) per insertion.

Bad news.

We can't always insert our keys in a random order. Why?



Q1: We can't always insert our keys in a random order. Why?

Algorithm Design Process

Hypothesize. How do invariants affect the behavior for each operation?

Identify. What strategies have we used before? What examples can we apply?

Plan. Propose a new way from findings.

Analyze. Does the plan do the job? What are potential problems with the plan?

Create. Implement the plan.

Evaluate. Check implemented plan.

ArrayList Invariant.

data is an array of items, never null. The i-th item in the list is always stored in data[i].

Iterative Refinement

ArrayQueue Invariant.

data is an array of items, never null. The i-th item in the list is always stored in data[(start + i) mod length].

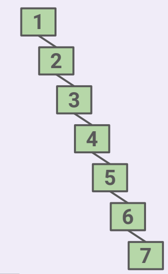
Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance (Loksa et al./CHI '16)

Let's zoom in on the Data Structure and Implementation Details. We need to optimize the worst case height of our binary search tree.

Iterative Refinement. Like the debugging process we learned earlier, information is key and motivates how we improve our invariants. As with debugging, the solutions are often very closely related to a particular framing of the problem. That's why there are lots of unsolved problems in theoretical CS. Oftentimes, we don't have the right understanding or perspective–hence why it's so easy to get stuck.

?: How have we applied iterative refinement before?

Q Rewriting Invariants



Hypothesis. Worst-case height trees are spindly trees.

Identify.

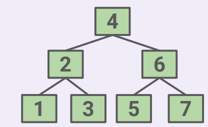
Spindly tree: all nodes have either 0 children (leaf) or 1 child.

Bushy tree: all nodes have either 0 children (leaf) or 2 children.

Plan. Say we have a BST in which every node has either 0 or 2 children.

Analyze.

1. What is the worst case search time in this case?
2. What do worst case trees look like?



Say we have a BST in which every node has either 0 or 2 children.

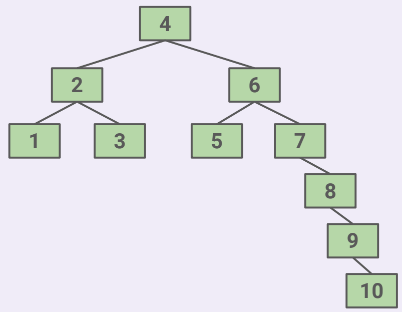
Q1: What is the worst case search time in this case?

Q2: What do worst case trees look like?

Q A Different Hypothesis

Hypothesis. Unbalanced growth leads to worst-case height trees.

How does adding a new node affect the height of a tree? Explain in terms of the height of the left and right subtrees.

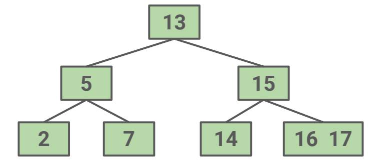


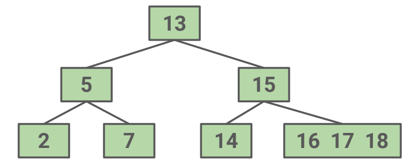
Overstuffing Leaves

Problem. New keys are added as leaves.

Avoid adding new leaves by overstuffing existing leaves.

What's the problem with this idea?





Q1: How does adding a new node affect the height of a tree? Explain in terms of the height of the left and right subtrees.

?: Does this suggestion increase the height of the tree?

?: What's the problem with this idea?

Promoting Keys

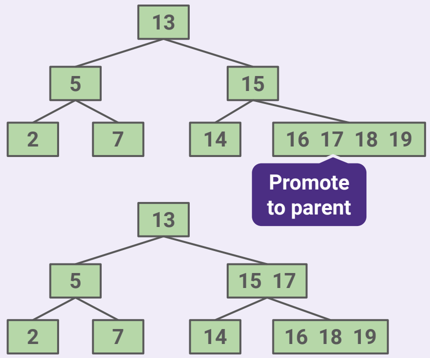
Height is balanced but leaves are too full.

Set a limit L on number of keys, e.g. L=3.

If any node has more than L keys, give a key to the parent, e.g. the left-middle key.

However, now 16 is to the right of 17.

Suggest a way to resolve this problem.

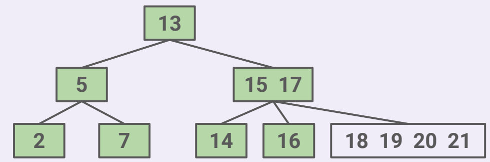


Q1: Suggest a way to resolve this problem.

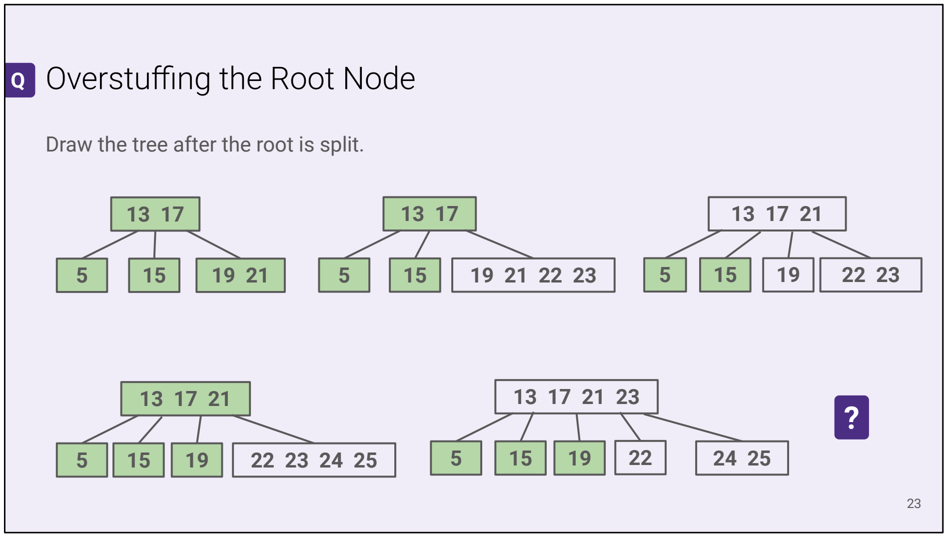
Adding More Keys

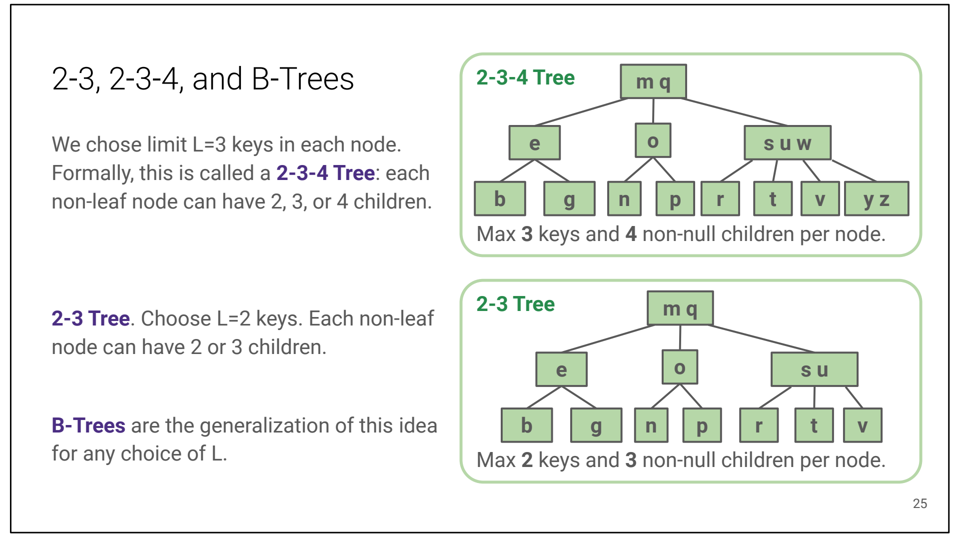
Suppose we add the keys 20 and 21.

If our cap is at most L=3 keys per node, draw the post-split tree.



Q1: If our cap is at most L=3 keys per node, draw the post-split tree.





Q1: Draw the tree after the root is split.

B-Trees are popular in two contexts.

- Small L (L=2 or L=3). Used as a conceptually simple balanced search tree as we saw today.
- Large L (in the thousands). Used in practice for databases and filesystems with very large records.

Tree Insertion

Give an insertion order for the keys 1, 2, 3, 4, 5, 6, 7 that results in a **max-height** 2-3 Tree.

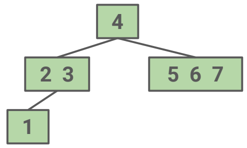
What about for a **min-height** 2-3 Tree?

B-Tree Invariants

1. All leaves must be the same depth from the root.
2. A non-leaf node with k keys must have exactly k + 1 non-null children.

These invariants guarantee bushy trees.

The tree to the right is not a possible B-Tree based on these invariants.



?: What is the least number of keys we can stuff into a 2-3 Tree node? The greatest number of keys?

Q1: Give an insertion order for the keys 1, 2, 3, 4, 5, 6, 7 that results in a max-height 2-3 Tree.

Q2: Do the same for a min-height 2-3 Tree.

?: Why is the tree to the right impossible? Which invariants does it violate?

?: Based on our algorithm design principle, explain to yourself why these invariants must be true.