CS 61B Fall 2024 — Data Structures, HKN Tutoring

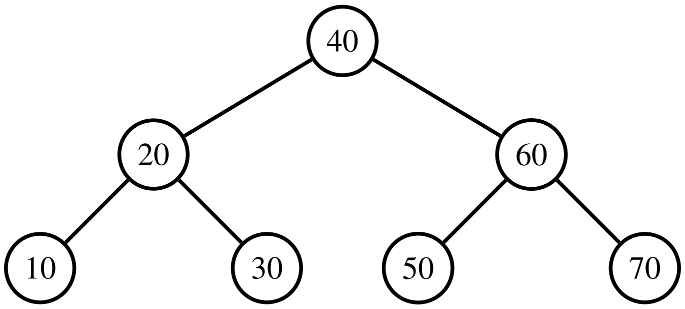
B-Trees

TLDR

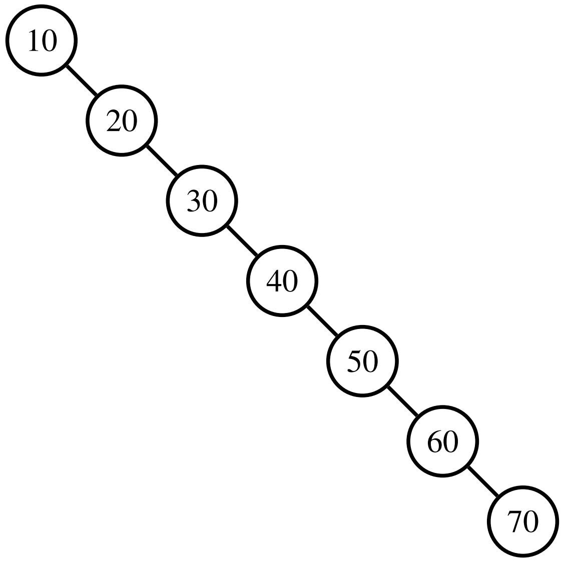
Although Binary Search Trees (BSTs) allow for possible efficient data storage and access, its time complexity depends heavily on the elements added and accessed. The idea of Balanced Trees (a.k.a. B-Trees) ensures a close-to balanced tree structure through "overstuffing" nodes and pushing values up.

1 Motivation

For BSTs, we strive for a bushy structure, such as:



But, sometimes adding items in a specific order can lead to a "degenerate" or spindly BST. For example adding items in the order $10, 20, 30, ..., 70$ forms a BST with a structure similar to that of a Linked List:



The adding rules of B-Trees that we will explore later allow us to avoid the latter BST such that a bushy structure is always possible to achieve.

Observation. *BST Height & Operation Time Complexity*

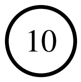
1. **Height for N items:** best case $\Theta(logN)$, worst case $\Theta(N)$
2. **Adding N items:** best case $\Theta(NlogN)$, worst case $\Theta(1+2+\cdots+N) \equiv \Theta(N^2)$
3. **Adding Nth item:** best case $\Theta(logN)$, worst case $\Theta(N)$
4. **Searching for Nth item:** best case $\Theta(logN)$, worst case $\Theta(N)$

2 Adding

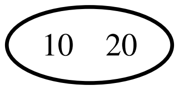
2.1 Standard Adding

Now, we will explore how adding into B-Trees — specifically 2-3-4 Trees — works, using the same sequence of numbers as above i.e. 10, 20, 30, ..., 70. For each `add` operation, we will show the initial step and subsequent steps to simplify the tree to maintain the invariants.

`add(10):`

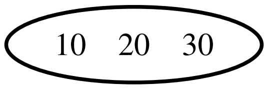


`add(20):`



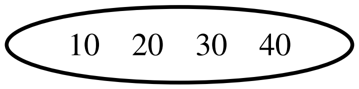
Each node can hold up to three values, so instead of making 20 a right child 10 (as in a BST), we stuff it into the existing node for 10. We follow standard adding rules for trees where lesser values belong on the left and greater values belong on the right.

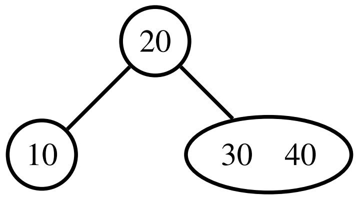
`add(30):`



Similar to the last step, we stuff 30 into the existing node.

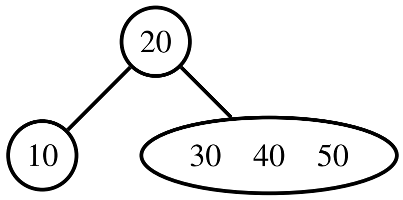
`add(40):`





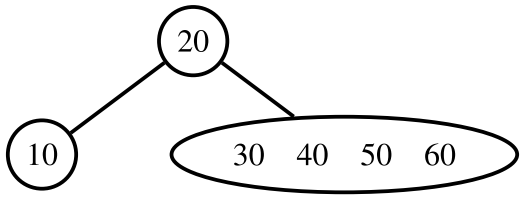
One node cannot hold more than three values, so we have push a value up. We arbitrarily choose the left middle value: 20 in this case. Notice how this maintains a valid tree structure where the left children (10) are less than the parent and the right children (30 and 40) are greater than the parent.

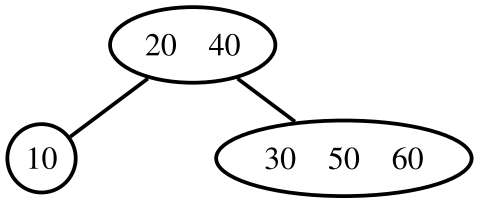
`add(50):`



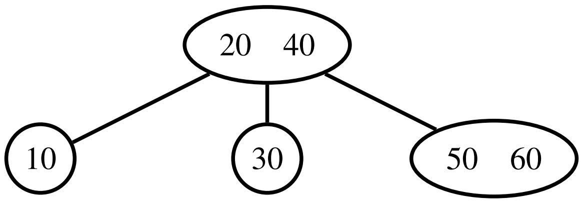
We check that 50 is greater than the root node, 20, and move to the right sub-tree. The right child node can fit one more value, so we stuff it to the right of 30 and 40.

`add(60):`



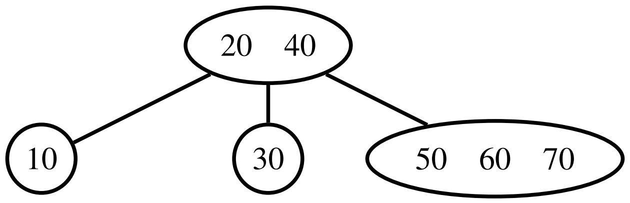


After stuffing 50, we have to push 40 upwards into the root node, placing it to the right of 20. But, the current structure of the tree says that 30 is greater than 40, since the node that contains 30 is the right child of the node that contains 40.



We move 30 to be the middle child of the root node because it is greater than 20 and less than 40.

`add(70):`

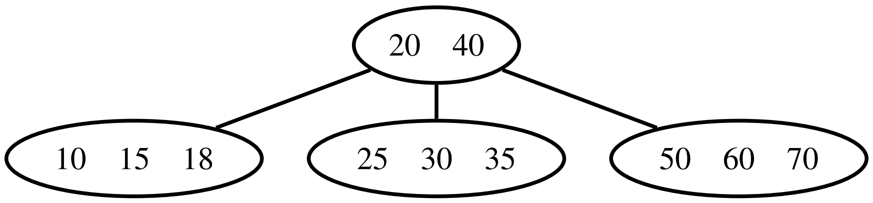


After adding all values, the B-Tree is balanced and not spindly, contrary to the original BST we saw earlier.

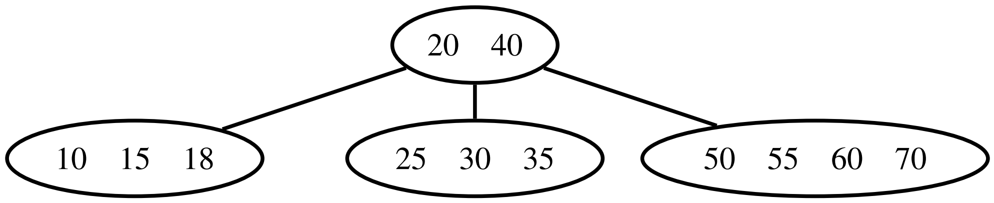
2.2 Complex Adding

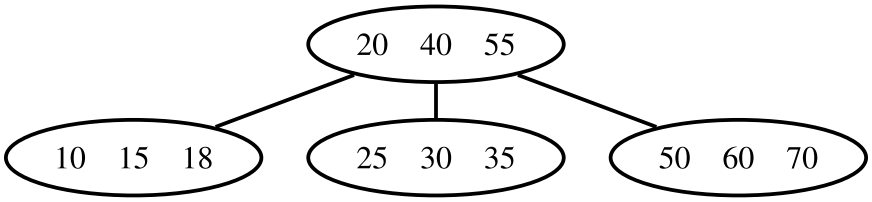
Continuing from the above, we will explore more complicated adding that require chain-reaction node splitting and pushing values up.

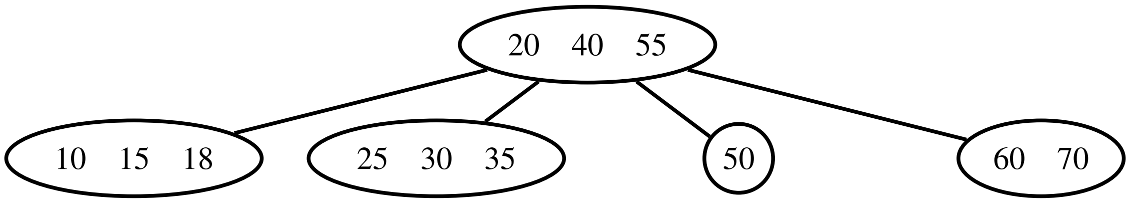
To fill up the tree more, `add(15, 18, 25, 35):`



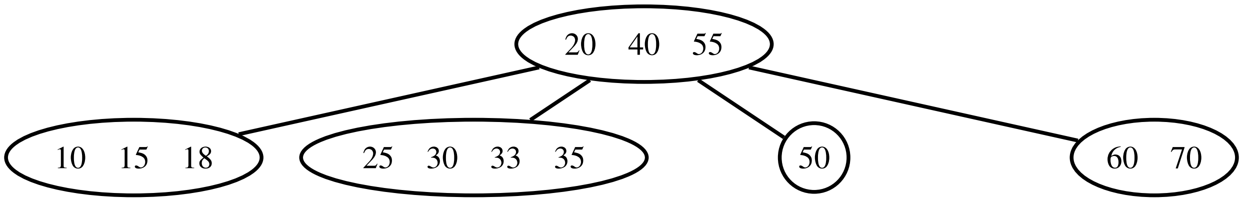
`add(55):`

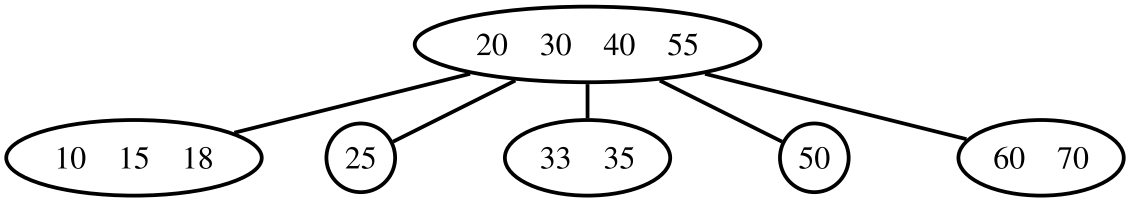




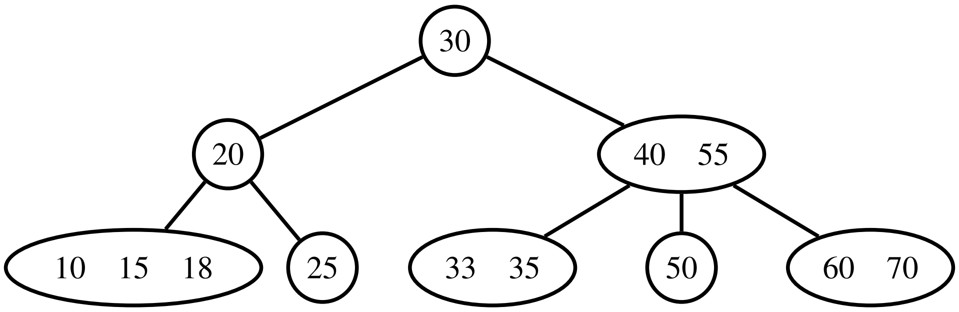


`add(33):`





After adding 33, we must push 30 up into its parent node (i.e. the root node) and then split the node that it came from. But, we still have to fix the root node, since it has four values in it right now.



This is our final B-Tree.

Observation. *Splitting Trees*

1. Splitting a non-root node does not change the height of tree.
2. Splitting the root node adds 1 to the height of the tree.

These observations are important when considering balance and time complexity.

Warning. *Chain Reaction Splitting*

Make sure to split the node you pushed the current value up from (if necessary) so that left, middle, and right child nodes are in the correct positions before continuing to push up values. Do chain reaction splitting in steps.

3 Terminology

The umbrella term "B-Trees" encompasses data structures 2-3 Trees and 2-3-4 Trees, and others. The above example follows a 2-3-4 Tree (a.k.a. 2-4 Tree); this naming comes from the fact that a node is able to have two, three, and up to four children. 2-3 Trees are very common and follow same adding, stuffing, and splitting logic with the only difference being any node can have two or three children only.

4 Invariants

To guarantee the balanced/bushy attribute of B-Trees, there exists two invariants:

1. Non-leaf nodes that contain k values have $k+1$ child nodes.
2. All leaf nodes are the same distance away from the root node.

5 Time Complexity

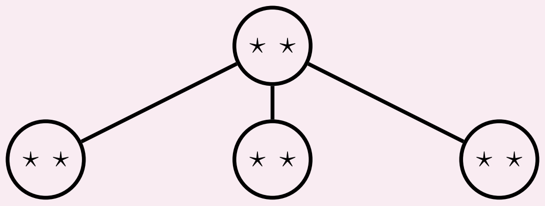
5.1 Height

Given a limit L values per node and N total values in the B-Tree, smallest height is achieved when each node has L values; contrarily, largest height is achieved when each non-leaf node has one value. Height ranges from $log_{L+1}(N)$ to $log_2(N)$, so is therefore $H = \theta(logN)$

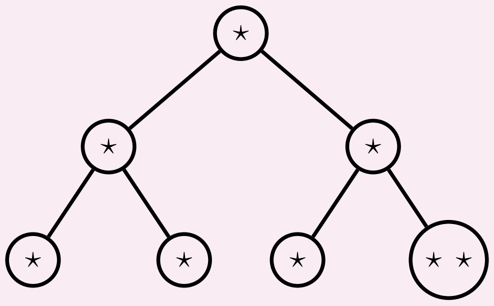
Example. *2-3 Tree Height*

For 2-3 Trees with $L = 2$, 8 items (arbitrarily represented with "$\star$"):

Smallest/best case height, grows with $\sim log_{L+1}(N) = log_3(N)$



Largest/worst case height, grows with $\sim log_2(N)$



The first 2-3 Tree is fully stuffed while the latter resembles a BST and is not stuffed at all.

5.2 `contains`

In the worst case, we have to inspect $H+1$ nodes and check L items per node. So, `contains` is upper bounded by $O(HL) \equiv O(L\, logN) \equiv O(logN)$. Remember for runtime analysis, we can drop constants when simplifying.

5.3 `add`

Similarly, in the worst case, we have to inspect $H+1$ nodes and check L items per node. But, we have to consider splitting nodes, which may occur a maximum of $H+1$ times. So, `add` is upper bounded by $O(HL) \equiv O(L\, logN) \equiv O(logN)$. Note that is the same runtime as `contains`.

ACKNOWLEDGMENTS

Inspiration taken from UC Berkeley's CS 61B Lecture Slides and Textbook.

Template by Patrick Mendoza (Spring 2024)
Crib Sheet by Robert Tian (Fall 2024)