Trees and Hashing

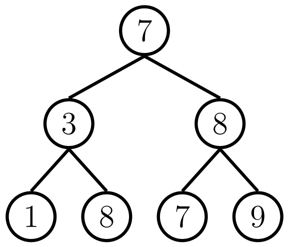
Abhinav Ashar
CS 61B: Data Structures and Algorithms

March 25, 2019

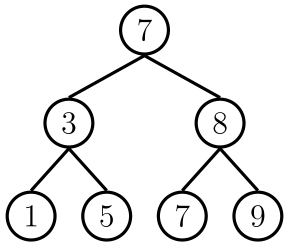
1 Trees

Binary Search Trees
A binary search tree (BST) is a specific type of binary tree that provides some ordering within itself. It can be extremely useful when working with data that needs to be relatively compared with other data. All BSTs follow this invariant: nodes smaller than a certain node will be stored to the left, and nodes greater than a certain node will be stored to the right. Any nodes equal can be arbitrarily stored to the left or right.

Invalid BST

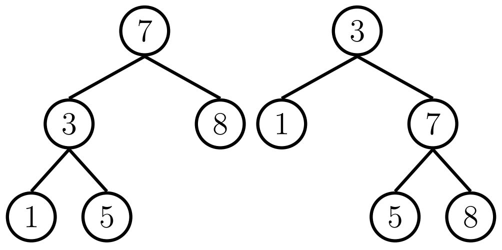


Valid BST



Let's look at the invalid BST. At first glance, everything seems to look correct. However, notice that 8 is on the left branch of 7, which breaks the rule of a binary search tree. Thus, this is an invalid BST. On other hand, look at the valid BST. There is no node in the left branch that is greater than 7, and there is no node in the right branch that is less than

7. Thus, this is a valid BST. Notice that BSTs can also face some of the same problems as regular binary trees: they can become spindly. One way this is possible is if you have a sorted list of numbers and add each number to the tree in order. Since each number is less than the next, all new nodes be added to the rightmost part of the tree, forming a spindly tree. The way we can deal with spindly, or unbalanced, trees is that we can rotate them such that we decrease the total height and make them more balanced. Here is an example of rotating a tree clockwise with respect to 2 (though it does not improve the height in this case):



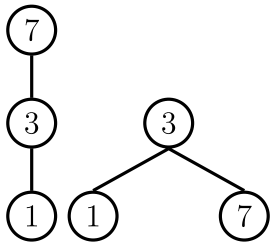
For rotating clockwise with respect to a certain node n, follow these rules:

1. Put n as the root of the tree
2. $n.left$ forms the root of the sub-tree for the left branch of n
3. *n.parent* forms the root of the sub-tree for the right branch of n
4. $n.right$ becomes the *left* child of the n's *right* child from step 3

For rotating counter-clockwise with respect to a certain node n, follow these rules:

1. Put n as the root of the tree
2. $n.right$ forms the root of the sub-tree for the right branch of n
3. *n.parent* forms the root of the sub-tree for the left branch of n
4. $n.left$ becomes the *right* child of the n's *left* child from step 3

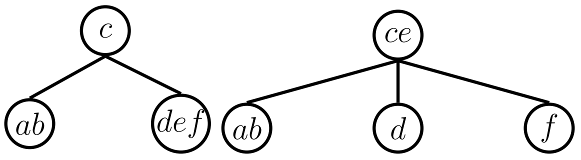
One way to know if a tree is unbalanced is if rotation can decrease the total height.



In this case, the height of the tree decreased from 2 to 1. A quicker way to know if a tree is unbalanced is to level-by-level horizontally, and check from left to right if there is a node. If we visit all the nodes that exist before encountering the first vacancy, then the tree is balanced. Else, it is not.

2-3 Trees

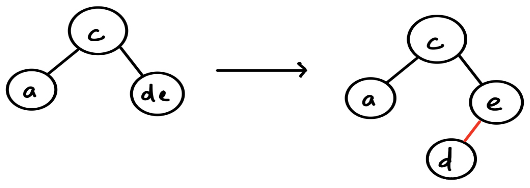
BSTs are very helpful, but they have a big problem: they need to be rotated in order to be balanced. This can be very inefficient, so 2-3 trees provide an answer to this problem by being self-balancing. 2-3 trees means that a node can have at most 2 values in it, and a node can have at most 3 children. Follow the same rules of BSTs when adding a value to a 2-3 tree, but now you can add multiple values to a node. Within a node, the values should be in increasing order from left to right. If the number of values in a node becomes 3, then restructure the tree by promoting the middle value and making the left and right values as children to the higher node.



This new 2-3 implies that $a \leq b \leq c$ and $c \leq d \leq e$ and $e \leq f$.

Red Black Trees

In this note, we have learned about BSTs and 2-3 trees. BSTs provide a defined structure, but they require rotation to balance. 2-3 trees are self-balancing, but are actually quite complicated to implement in code. As a result, we have Red Black trees, which are self-balancing and very easy to implement. Red Black trees are functionally the same as 2-3 trees but just look slightly different. This is because we take a node with multiple values in it and split it up, creating red "glue" links between these split values. The red link indicates that they are actually part of the same node, but have been split up for implementation purposes. When we split the node, the smaller of the two numbers becomes the *left* child of the larger number. This represents a Left-Leaning Red Black trees (LLRB).



Here are a few more properties of Red Black trees:

a) Despite adding the extra red edge, the red edge does not actually increase the height of the Red Black tree (real height of Red Black Tree is the height of the 2-3 tree)

b) Normally has no more than about 2 times the height of the 2-3 tree

c) No 2 red links can be next to each other because that would essentially represent a 3-node (not possible in a 2-3 tree)

d) When modifying a Red Black tree (such as adding), it is often easier to convert the

Red Black tree to a 2-3 Tree, carry out the operation, and convert back to a Red Black tree

2 Hashing

All the above data structures are great for accessing an item or checking whether an item exists in the data structure, but can we do better? In fact, we can with the concept of hashing. Hashing allows us to use the properties of an object to determine a hashcode that will allow quick access to the object. Think of it like a row of boxes, where we place an object in a certain box based on the properties of an object that determine the hashcode. Thus, if we wanted to look for that object afterwards, we don't need to look through all the boxes. We simply use the properties of the object we are looking for to calculate the hashcode and look in that specific box. There are some major rules when it comes to adding items to data structures that utilize hashing:

1. Never store objects that can change because this can cause an object to be located in box different from where it is expected
2. Never override equals without overriding the hashcode
3. Hashcodes must be consistent, have an equality constraint, and have uniqueness
    a) **Consistent:** the hashcode method must consistently return the same integer if nothing about the object changes
    b) **Equality Constraint:** if two objects are equal according to the equals() method, then their hashcodes must be equal
    c) **Uniqueness:** if two objects are unequal according to the equals() method, then their hashcodes must be unequal

Lastly, there is a load factor to keep track of. When we determine which box to look it, we simply need to take the time to look through all the objects in the box. However, if there are too many items in the box, this can be inefficient as well. This is referred to as having too many *collisions*. As a result, we need to increase the number of total boxes and re-determine the hashcode for every item. While this seems like it would be inefficient, it happens so infrequently that the runtime for hashing operations (add, contains, get etc.) is amortized constant time.