binary trees

dillon freeman

october 12. 2021

types

binary tree when thinking of a tree in regards to collections, a tree moreso resembles what is beneath the tree (its roots). each "vertex" on a tree is labeled a node, while each edge is the connection between these two nodes. there can not be an edge without an incoming node(meaning, if you were to create an edge, it must point to some node in memory). A tree also must be open, it cannot be closed.(this means that for example, a square cannot be a representation of a tree.) a root is a point of reference for the entire tree. every child can traverse to eventually end up on the root node. when speaking in the context of binary trees, a binary tree *is a tree that has no more than 2 child nodes per parent node.* a parent node can have none, 1 or two child nodes but no more than 2. there is also the balanced binary tree, *in that during a* ***traversal****, there would be a height distance of no more than 1 between the subtrees.* What this means ties into a topic that will be covered later. To continue, a binary search tree is *a tree where the roots subtrees will not only be subtrees, but they will also have nodes greater or less than the root node, respectively* to explain this more clearly, think of a root node of 100. not only will one side have values less than 100 and one greater than 100, but the subtree of that root node(which is a subtree of the actual root) will also result in a similar behavior(left and right follow similar behavior). An AVL tree is different from a binary search tree in that it is self balancing. how this functionality is done is thru a series of logic and rebalancing. the "logic" here is that every child subtree differ no more than a value of 1 from the height of any other subtree. what this means is that if one were to a traversal on a subtree, the value of the height will either be 1, -1 or 0. the self balancing comes in play when doing an insertion or a deletion where rebalancing logic will ensure collection integrity. there is also an avl red-black tree, however even the knowledge for avl tree is within the recesses of my mind(i cant recall how to differentiate the subtree pathing, etc.)

methods

binary trees support basic collection functionality. insertion and delete are predefined as a part of a library for most programming languages.

traversal

to search for a value in a binary tree, one would make use of a good traversal algorithm. typically breadth-first search(traversal) will fit the bill. breadth-first search essentially starts at the root and searches for the key at depth 1,2,3...d-1,d until it reaches d to find the key, where d is the max depth of the tree(hence the word, breadth-first). The depth of the tree is expressed as the number of verticies plus the edges. If the key does not exist then at worst, one would traverse all verticies and nodes. searching in this fashion is at worse linear to the depth of the tree. since it is dependent on the depth, it is faster when the size of the tree is smaller. this is at worse O(n)

insertion

when using insertion, there is the tree and the key. the key is the var that will be stored in the binary tree after the function is done executing. this can be done using a similar concept with breadth-first search, where one will search for a node which has either left and right nodes as null or one or the other. if it can however, it will prefer to put it to the left. to make this simpler of an explanation, think of what was said above for search. one will search a tree to find a key by traversing every possible outcome on depth d-x until it reaches d, where x specifies how far off the max depth we're currently searching at. instead of looking for that key, one would be attempting to find a place to put that key(insert). this is at wost O(n)

delete

for deletion(assuming one had known where the key is) one would find the deepest rightmost node(if using inorder traversal) and replace the data from the rightmost node with the key. then (since the key is in memory but not in the tree anymore) we can disconnect the rightmost node(or delete). at worst, this has O(n) time complexity.

pros vs cons

although a hash map functionality for insertion, search and delete is in constant time and therefore is on average faster than using a binary tree a binary tree is easier to implement in code than a binary tree. if one had to reinvent the wheel, it'd be easier to implement a binary tree because its not easy to implement hashing in the way its intended for a hashmap. a hashmap is on average constant

time and at worse is significantly worse when collisions occur or when resizing the collection(which undoubtedly will result in collisions). when it comes to use cases aside from being better or worse than similar collections, trees (especially multi-level trees) are great for indexing databases. multi-level indexing essentially breaks down indices into smaller indices so that it can use less space and still be called fast despite the index table size. a table index in sql allows quick data retrieval. the larger the table index, the more likely a binary tree will be needed to keep the swift data retrieval.