Analysis Report

Binary Search Tree Operations

The operations which can be implemented reasonably efficiently are using a binary search tree are:

- Searching the tree for the name of a particular city within the city. On my code this method runs in 0.001 seconds (CPU Clock) and 0.0013413 seconds (Wall Clock)
- Deletion if the node we are deleting is a leaf node as this means we do not have to rearrange the tree to make it comply to the standards of a binary search tree
- Deleting a node with one child node as then we only have one more operation to carry out by replacing the deleted node with its child.

Operations which take longer than O(log n) are:

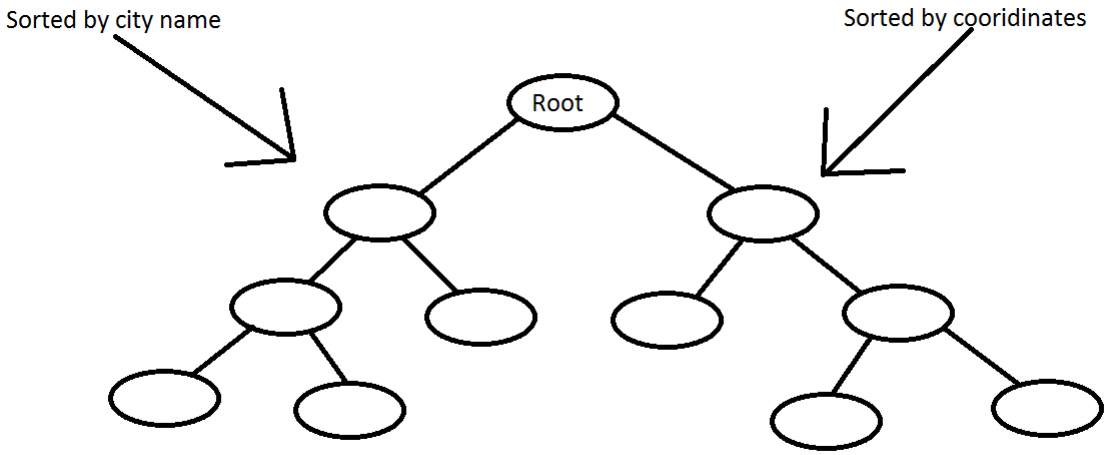
- Inserting a new record into the tree. However, due to the spec of this CA, it is an error to insert a record with identical coordinates of a city already in tree. This means that we must search through the entire tree looking to see if these coordinates exist meaning that the search is O(n) time operation. If this was not the case, then the insertion into the binary search tree would be O(log n). But in this case it is O(n). On my code this method runs in 0.001 seconds (CPU Clock) and 0.0014428 seconds (Wall Clock)
- Searching the tree by coordinates. This is due to the tree being organised by city name rather than coordinates. This means that we are unsure to go left or right when searching the tree. The solution to this is to search the left and right side of the tree until we find the node we are looking. This results in this search method running in O(n) time. On my code this method runs in 0.002 seconds (CPU Clock) and 0.0018635 seconds (Wall Clock)
- Traversing the tree. This visits each node in the tree to print its contents resulting in it running in O(n) time. On my code this method runs in 0.006 seconds (CPU Clock) and 0.0064386 seconds (Wall Clock)
- Finding the height of the tree would be O(n). On my code this method runs in 0.001 seconds (CPU Clock) and 0.0003328 seconds (Wall Clock)
- Printing all records within a given distance of a specified point runs in O(n) as we must calculate the distance for each node meaning we visit each node within the tree and output it if it meets the requirements. On my code this method runs in 0.011 seconds (CPU Clock) and 0.0101602 seconds (Wall Clock)
- Destructor as it also visits each node contained within the tree so it can destroy the node and free up the memory to prevent memory leaks.
- Deleting a node which has two child nodes as this requires a rearrangement of the tree

Stepping away from the spec of the CA and looking solely at binary search trees, the operations which can be implemented in O(log n) time are:

- Insertion into the tree
- Searching the tree
- Deletion if the node we are deleting is a leaf node as this means we do not have to rearrange the tree to make it comply to the standards of a binary search tree
- Deleting a node with one child node as then we only have one more operation to carry out by replacing the deleted node with its child

Improving the Database System

This database system can indeed be improved to make it more efficient for use to organise records by locations. The only way I can think of implementing this involves duplicating the data in the tree nodes and having two trees. The root node of the tree would have two pointers in it, one pointing to a tree sorted on city name and another one pointing to a tree which is sorted on the coordinates. Both trees would have the same data within them. So now when we search the tree on coordinates, instead of running in O(n) time as we need to search the whole tree, it would now work in O(log n) time as now we know that lower coordinate values will be to the left, and larger to the right. This is all well and good but now we have another problem, the two trees may go out of sync with each other. This isn't what we want. Now we have an extra overhead of completing this task. This would mean instead of doing one insert, we do two. In this case, this operation is still completed in O(log n) time in the average case. This also goes for the delete operations which still have the time complexity as mentioned in the above section. The tree would now look similar to the following:



Additional or different algorithms/data structures

Priority Queues

A problem with binary search trees are that we do not have instant access to a node via the index. By this I mean we cannot call code such as *"tree[5]"* to pick out the fifth element in the tree unlike arrays. If we wanted to pick out an element in the tree then we would have to iterate the tree looking for the specific element we want. This makes it less efficient for plucking out data we need as it is not done in constant time i.e. O(1). A solution to this problem is to implement the tree as a different yet similar data structure. We could use a Priority Queue and implement it as a Heap which would improve efficiency while maintaining a tree structure although this data structure is seen as a logical binary tree. To sort the heap using the Heap-Sort Algorithm, we can do this in O(n log n) time rather than our traditional bubble or selection sorting which is done in $O(n^2)$ time. A complete binary tree is always maintained when we insert into the heap and is completed in O(log n) time. This is also the case for deletion from the heap. A disadvantage of this data structure is searching for an element. The average case for searching the priority queue is done in O(n) time.

AVL Trees

This data structure is very similar to the binary search tree. One major difference is that the AVL tree remains balanced at all times. For searching, inserting, and deleting nodes in the tree there is no worst case as the average case and worst case are both equal in time to complete. These three operations are done in O(log n) time. With binary search trees, insertion, deletion, and searching has a worst case of O(n), thus making this data structure more efficient on time. One downside to this data structure is the need for rebalancing the tree when a delete or insertion occurs. Although even with this overhead, it still manages to complete this in O(log n) time.

The Red-Black Tree

This is another binary search tree which can provide us with some advantages. We assign nodes with an extra colour property which can be red or black. Some advantages of using this data structure are that because it is also a self-balancing tree, we can guarantee operations such as insert, delete, and search to run in O(log n) time in average and worst cases similar to AVL trees above. They perform very well when inserts and deletes are relatively frequent. Rebalancing the tree after an insertion is also improved as it only occurs once at most, deletion occurs at most twice which is an improvement to AVL trees. Some disadvantages are that they can be highly complex to implement from scratch. If the tree is only used for searching data then performance can be less efficient of that to the AVL tree.

Hash Table

The hash table data structure works as a key-value pair. This is a very quick data structure when we look at its average case which for search, insert, and delete is done in constant time i.e. O(1) which makes it quicker than the above data structures. However, if we look at its worst case scenarios we could suffer a big performance hit as those operations can be done in O(n) time making them quite inefficient if we get trapped in the worst case. But when we are in the average case the only overhead we have is the hashing function of the key and if that key comes into collision with an entry already in the table meaning the developer must account for these types of events that could occur. However, nothing is ordered in this data structure.

Daniel Buckley
X00109141