Choice of Data Structures/Algorithms

For this project we were required to choose an ADT to store N transaction records such that we can perform the standard ordered functions (min(), floor(), sorted() etc. This meant that we would need some data structure which could not only support continuous updates in the memory allocation, but also perform these operations within a suitable time. The basic options which can be implemented through arrays (linear data structures-stacks, queues etc) were considered, however they didn’t support the increase of size efficiently due to their use of contiguous memory. Since we are handling large sets of data, it made more sense to use a non-linear data structure such as a tree which would utilise computer memory more efficiently.

Our choice was to go for a balanced search tree, in particular the Left Leaning Red-Black Binary Search Tree (LLRB BST). The other option was the 2-3 search tree, but since it is isomorphic to the LLRB and proved harder to implement we decided to go with the latter. This also ensured perfect balance and O(log(n)), tree height, time complexity in virtually all the operations we considered, through rotations and recolouring of nodes.

We decided upon having twelve trees in total, one for each of the top twelve FTSE-100 companies, due to the context. The key was the ‘transaction’ and the value was the ‘trade value’. The key was linked to an array of which contained the four required pieces of information. (It is also worth noting that if we were to have used a linked-list, we wouldn’t have had to create separate data structures for each of the companies as we could have been able to sort by company followed by trade value. However this would have meant sorting upon input call which has O(NlogN) minimum whereas LLRB BST has O(log(N)).)

Space Complexity Analysis

Originally, we had approached programming the functions in a recursive manner to keep the code more readable. Upon further inspection, we decided to modify most functions to be iterative as this would reduce the stack space from logarithmic to constant (stack memory, in the context of our code, is used to handle processes/instructions that are being run by our program by pushing it onto the stack and popping it from the stack once the instruction has been carried out). A recursive call would have meant the stack would increase in size for each recursion, which in our case would have meant traversing down the tree; the worst case would have meant log(N) total recursions on the stack. This could have resulted in a possible stack overflow due to the stack’s finite space, which would have threatened the entire program. Iterative code bypasses this problem since the function would be put onto the stack once and repeat over its own code multiple times. In terms of main memory storage, there is little difference between iterative and recursive, however it is clear that taking the iterative approach is more space efficient for the stack.

1. min/maxTransaction()-accepts two inputs and returns one output; the functions min() and max() are called respectively which are iterative so therefore a constant stack space. The overall space complexity is constant ~ O(1).
2. logTransaction()-coded iteratively, two operations are run simultaneously at any one point; the put() operation and one auxiliary function from rotateRight(), rotateLeft(), recolour() and isColourRed(). In terms of main memory space complexity, logTransaction() has two inputs, and two auxiliary variables in addition to put(). The put() also requires two inputs. There are six occurrences of isColourRed() (which takes 2 inputs and has 1 output), which can each call any the other auxiliary functions once. The rotate functions can also be called which accept 1 input and have 1 output created in the function, while recolour() has 1 input only. This means overall, there are 30-37 items stored. Thus the space complexity is not reliant on the number of elements in the tree and overall is constant ~ O(1).
3. floor/ceilingTransaction()-Called iteratively to ensure a constant space complexity. Three inputs are received and one output is displayed (as well as the floor/ceiling() function call which takes an input and returns and output) therefore it is a ~ O(1) space complexity.
4. sortedTransaction()-accepts two inputs and returns an ouput array of N records, where N is the number of records in the tree. The sorted() function is called, which takes one input and creates an array of N records as the output. We decided to run it recursively due to difficulties in implementing a readable iterative function. This would result in a stack build up. The worst-case would be a tree consisting of alternating red-black nodes, hence 2\*height recursions, so a stack space of 2log(N), with the average-case being log(N), meaning O(log(N)).
5. rangeTransaction()-similar to sortedTransaction(), programmed recursively. The worst-case would be handling an array of up to N records. The worst case on the stack would be 2log(N), average case 0.5log(N) and O(log(N)).

As discussed, the stack space could be reduced by writing the algorithms iteratively, however the linear space main space complexity dominates the space complexity as N > logN and N > 1 so either way the overall space complexity (both on average and worst case) is linear ~ O(N).

Each Node contains 6 attributes: transaction record, trade value, isRed, left child, right child and parent. This means all Nodes also have 0, 1 or 2 references to it. Therefore, each attribute has constant size and there are a constant number of attributes so each Node has a 6N space requirement ~ O(N) linear space complexity for the trees.

Each Node is stored in one tree once so there is no increased storage. Each tree also has no residual data except for the dictionary key pointing toward it (object of class StockTrading with attribute self.transactions is a dictionary holding all one tree for each company - key is company name, definition is the tree). Therefore overall, Space complexity is O(N) linear.

References:

https://stackoverflow.com/questions/46658483/splay-tree-worst-case-search-time