Michael Tangy

CS 340-001

FALL 2015

Project 2 – Crossword Hunter: Report

Binary Search Trees are a data structure that store elements (variables, strings objects etc) in an organized and sorted fashion by assigning each element a key (which is typically an integer) and then using that key to organize the positioning of each element. During insertion to a binary search tree the keys are used to distinguish whether or not to place the element to the right or left of its parent. If the key of the element being inserted is less than that of its parent than the element is placed to the left of that parent if the key is greater than the parent it’s placed to right of it. This process causes Binary Search trees to have at most two children per node (which is what makes them a “Binary” Search Tree), the process also causes the node with biggest key to be the rightmost leaf and element with the smallest key to be the left most leaf.

I implemented an iterative BST insert function that iterates a pointer starting at the BST’s root and goes down the tree until it reaches the leaves where it creates a new node and places it either to the right or left of the leaf node depending on its key value. I also implemented an iterative search function that starts at the root of the tree and iterates down while constantly comparing the current nodes key with the key passed to the function. When that key is found it returns that node. I also implemented a wild card search function that performs an in order traversal of the BST starting at the root once the traversal reaches a node who’s key is less than the minimum key value (which is the wild card word with the wildcard characters replaced by a’s) it starts to print all the nodes (as well as counts all the nodes) until it reaches a node who’s key value is greater than maximum wild card value (which is the same as the minimum however it replaces the characters with z’s instead of a’s). Both of my functions were implemented in an iterative manner which a lot different than many of typical recursive implementations because it has less overhead by needing a smaller stack size for the programs process. The iterative can also handle a larger number of elements without needing a massive stack.

The insertion and search functions of a binary search tree both have a time complexity of O(log n) for their average case which applies to the permutated word list in this situation. This can be seen by the plots of the timing samples taken from executing the function. A time complexity of O(log n) is desirable because the number of operations needed to execute an increasing amount of elements is proportionally not nearly as much as many other Big O functions. Both functions also have a Worst case of O(n) for the pre-sorted word list. This time complexity is worse than its average case but not as bad as some sorting algorithm like insertion sort.

Red black Tree:

The red black tree is a self-balancing BST where all the nodes have a color (red or black) associated with them. Both children of a red node are black as well as all leaf nodes and the node pointing to the root. This property causes every path from a node to one of its decedent leaf nodes to have the same number of black nodes. This property is needed to rotate the nodes in order to maintain the balance of the tree.

I implemented an iterative red-black insert function that iterates a pointer starting at the RBT’s root and goes down the tree until it reaches the leaves where it creates a new node and places it either to the right or left of the leaf node depending on its key value. It then performs a fix-up to maintain the RBT’s color and balancing properties by adjusting the nodes colors and performing rotations. I also implemented an iterative look up function that starts at the root of the tree and iterates down while constantly comparing the current nodes key with the key passed to the function. When that key is found it returns that node. I also implemented a wild card search function that’s the same as the BST’s. These functions also have the same iteratice benefits as the BST’s functions.

Red-Black tree’s time complexity is the same as the average case of the BST’s functions since RBT’s are always balanced. The worst case only occurs with BST when all the nodes are sorted and the tree essentially becomes a linked list and that is impossible with a properly implemented RBT.

Data:

The time complexities for all the data were all as expected. The x-axis is number of elements and the y-axis is time in seconds for all plots.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Column1** | **Column2** | **Column3** | **Column4** | **Column5** | **Column6** | **Column7** |
| BST |  |  |  |  |  |  |
|  | unsorted | sorted |  |  | unsorted | sorted |
| 15000 | 0.09225 | 23.556 |  | 75000 | 0.00438 | 1.112 |
| 30000 | 0.21912 | 107.35 |  | 150000 | 0.005128 | 2.512 |
| 45000 | 0.3535 | 248.94 |  | 225000 | 0.006227 | 4.385 |
| 60000 | 0.514703 | 451.87 |  | 300000 | 0.007145 | 6.27 |
| 75000 | 0.6369 | 702.89 |  | 375000 | 0.00867 | 9.5638 |
|  |  |  |  |  |  |  |
| RBT |  |  |  |  |  |  |
|  | unsorted | sorted |  |  | unsorted | sorted |
| 15000 | 0.15385 | 0.195 |  | 75000 | 0.00634 | 0.00675 |
| 30000 | 0.3192 | 0.4387 |  | 150000 | 0.007112 | 0.007439 |
| 45000 | 0.515 | 0.6096 |  | 225000 | 0.008457 | 0.008829 |
| 60000 | 0.7772 | 0.9382 |  | 300000 | 0.009165 | 0.009563 |
| 75000 | 0.9304 | 1.2976 |  | 375000 | 0.010837 | 0.01107 |

BST

Searching:

RBT:

Searching: