CSC263 Assignment 2

Akhil Gupta, 1000357071

October 2015

1 Question 1

A set S of "coloured pixels" that are represented by triples (x, y, c), where x and y are positive integers denoting a position of a pixel on the screen, and $c \in \{R, B, G\}$ denotes a colour. Note that each position can be assigned up to three colours, for example, $(5, 3, R)$,$(5, 3, B)$ and $(5, 3, G)$ can co-exist in a PixelSet, but there cannot be more than one "$(5, 3, R)$".

Give a *detailed* description of how to use AVL trees to implement PixelSets. In particular, answer the following questions.

1. How many AVL trees are you using? What does each node correspond to? What information is stored in each node?

   Answer: In order to solve this problem, we should use two AVL trees. Each node in the tree corresponds to one position (x, y) and each node stores the set of colors $\{R, G, B\}$ that are assigned to that position.

2. What are the keys that you use for sorting each of the AVL trees? For each AVL tree, define carefully and precisely how you compare two pixels positioned at (x, y) and (x', y').

   Answer: For each of the AVL trees, the keys are the position pairs (x, y) and the keys for the first tree is sorted where the row number is x and column number is y. i.e. $(x, y) = (row, column)$. The second tree is sorted where the column number is x and row number is y. i.e. $(x, y) = (column, row)$. In the first tree, two keys $(row, column)$ and $(row', column')$ are compared by checking that $(row, column) < (row', column')$ if and only if $row < row'$ or ($row = row'$ and $column \leq column'$). Similarly for the second tree, two keys $(column, row)$ and $(column', row')$ are compared by checking that $(column, row) < (column', row')$ if and only if $row < row'$ or ($row = row'$ and $column \leq column'$). This method ensures that when the pixels are sorted, all the rows are arranged from smallest to largest and each column within the row is also arranged from smallest to largest.

3. For each of the above operations, describe in detail how it works, and argue why it works correctly and why its worst-case runtime is $O(\log n)$.

   Answer:
   a) ReadColour(S, x, y): Do an AVL-Search for the key (x, y) on any of the two trees and return the colour stored in the corresponding node. If the key does not exist, return the empty set. We know from lecture that AVL-Search takes $O(\log n)$ time.

   b) WriteColour(S, x, y, c): Do an AVL-Search for the key (x, y) on any of the two trees. If the node is found and c does not exist in its colour set, add the colour c to its colour set $\{c\}$. If the node already has colour c, do nothing. If the node is not found, we need to do AVL-Insert with key (x, y) and colour set $\{c\}$ on both AVL trees. We know that from lecture AVL-Insert takes $O(\log n)$ time. Since we have to search for the key first ($O(\log n)$) and then insert if key not found ($O(\log n)$), therefore, we do $O(\log n)$ work in total.

   c) NextInRow(S, x, y): For this case, we need to look at the first tree $(row, column)$ and do an AVL-Search for the coloured pixel with key (x, y). Let n be the node with key (x, y). Then we need to do TreeSuccessor algorithm to find the next coloured pixel after n in the tree. Let the next coloured pixel key be called (x', y').If the key does not exist in the tree, return $(0,0)$. Next we need to compare the row numbers of both keys. If $x' > x$, return $(0,0)$. Else if $x' < x$, it means that the new row number is bigger than the old one following the definition of comparing two pixels in part b), hence we return (x', y'). We know from lecture that the TreeSuccessor algorithm takes O(log$n$) time and AVL-Search takes O(log$n$) time as well. Therefore, NextInRow(S, x, y) takes O(log$n$) time in total.

d) NextInColumn(S, x, y): For this method, we follow a similar procedure as described for NextInRow(S, x, y) but instead of looking at the first tree, we need to perfom the operation on the second tree $(column, row)$.

e) RowEmpty(S, x): In order to check whether row x is empty or not, we need to check that the coloured pixel with key $(x, 0)$ does not exist in the tree. So we need to first insert the key $(x, 0)$ into the first tree using AVL-Insert $(row, 0)$ to check whether no coloured pixels exist in that row. Then we need to call NextInRow$(S, x, 0)$ on the first tree, if it returns $(0,0)$ it means that no such coloured pixel exists, hence it checks whether Row x is empty or not. Finally, we need to delete the key $(x, 0)$ from the tree using AVL-Delete. Since AVL-Insert, NextInRow and AVL-Delete take O(log$n$) time, therefore, RowEmpty(S, x) also takes O(log$n$) time.

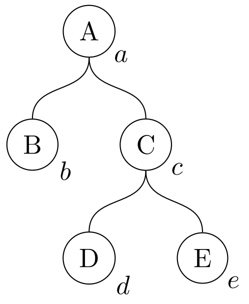
f) ColumnEmpty(S, y): For this method, we follow a similar procedure as described in RowEmpty(S, y) but instead of performing them on the first tree, we use the second tree and instead of inserting $(x, 0)$ = $(row, 0)$, we insert $(0, y)$ = $(0, column)$.

2 Question 2

We want to an augment AVL tree so that, in addition to the usual dictionary operations Insert, Delete and Search, it also supports the operation Average, defined as follows: given a pointer to any node x of T, Average(x) returns the average key-value in the subtree rooted at node x (including x itself). The query should work in worst-case time $\Theta(1)$.

(a) What extra information needs to be stored at each node?

Answer: In order to solve this problem, we need to store *node.size* and *node.sum* at each node. *node.size* stores the size of the subtree (as discussed in tutorial) and *node.sum* stores the sum of the values of the nodes in the subtree rooted at that node. At a leaf node, *node.size* = 1 and *node.sum* = value stored at the node. Hence, we can calculate the average for any node in the tree by dividing *node.sum* by *node.size*. This method ensures that the Average function takes $\Theta(1)$ time. Consider the below AVL tree with nodes A, B, C, D, E and corresponding sizes a, b, c, d, e:



Notice, $a = b + C.size + 1$, $C.size = 1 + d + e$ AND $A.sum = A + B.sum + C.sum$, $B.sum = B$, $C.sum = D.sum + E.sum$, $D.sum = D$, $E.sum = E$.

Recalling Theorem 14.1 of the Textbook and the 52nd lecture slide of Week 4, "If the additional information of a node only depends on the information stored in its children and itself, then this information can be maintained efficiently during Insert() and Delete() without affecting their O(log$n$) worst-case runtime. In our case, *node.size* and *node.sum* both depend on information stored in their children and itself. Therefore, AVL-Insert and AVL-Delete still maintain their

O(log$n$) worst-case runtime and the AVERAGE is calculated by dividing *node.sum* by *node.size* which takes O(1) time.

(b) Describe how to modify INSERT to maintain this information, so that its worst-case running time is still $O(\log n)$. Briefly justify your answer.

Answer: When a new node is inserted, we need to increase *node.size* of every node from insertion point back to the root. Also, we need to update *node.sum* of the parent node of that node recursively till the root. We know that AVL-insert takes O(log$n$) time and updating *node.sum* takes O(1) time. Therefore, AVL-insert still maintains its worst-case runtime of O(log$n$).

(c) Describe how to modify DELETE to maintain this information, so that its worst-case running time is still $O(\log n)$. Briefly justify your answer.

Answer: When a node is deleted, we need to decrease *node.size* of all affected nodes on way back. Also, we need to update *node.sum* of all nodes in the subtree rooted at that node and the parent node recursively. We know that AVL-delete takes O(log$n$) time and updating *node.sum* takes O(1) time. Therefore, AVL-delete still maintains its worst-case runtime of O(log$n$).