Time complexity analysis of quadTree.sml

Lucas Arnström & Gustav Dyrssen

February 4, 2014

Contents

1 Function emptyQtree(e)

Independently of the input size, emptyQtree will always execute the code in the same time, ergo its time complexity is O(1).

2 Function newIfEmpty(q, r)

The function newIfEmpty will always execute the code in the same runtime regardless of the input, which means its time complexity is O(1) as well.

3 Function insert(q, r)

3.1 Preamble

For this calculation, the n parameter in T(n) is equal to the area/extent of the tree. Since this calculation is for the worst case scenario, the rectangle that is to be inserted is of the smallest possible size (1x1) and can be found in one of the four corners, thus it will be inserted at the very last possible recursive call of the function.

3.2 Calculation

For each recursive call the extent of the tree will be divided by four. This is only possible as long as the extent's area is divisible by four without any remainders, meaning, the shortest side of the extent is bigger or equal to two.

The function keeps making recursive calls until the rectangle can be inserted in the vertical or horizontal centre line. In the case of a rectangle with the extent/area 1x1 and that can be located in any of the four corners, the function will recursively divide the tree's extent until it becomes small enough for the rectangle to cover any of the two centre lines, in this case the area would be any of these three combinations: 1x1, 1x2, 2x1.

Binding values, comparing them and inserting the rectangle in either the horizontal or vertical centre line will always execute in constant time and will be referred to as C_1. Making a call to newIfEmpty to create a new sub-tree if there is none, or simply returning the sub-tree if one is already there, will always execute in constant time and will be referred to as C_2. The recursive formula for the function can thus be defined as:

$$T(n) = \begin{cases} C_1 & \text{if } n = 1 \land n = 2 \\ T\left(\frac{n}{4}\right) + C_2 & \text{if } n > 2 \end{cases} \tag{1}$$

To explain the formula, if the extent/area is equal to 1 or 2 the rectangle has to bee inserted since the extent can no longer be divided by four, and since this is the worst case scenario, the rectangle will not be inserted until the very last possible recursion.

By expanding/rewriting the formula above we can form an idea on how to form the time complexity of the function:

$$\begin{aligned} T(n) &= T\left(\frac{n}{4}\right) + C_2 \\ &= T\left(\frac{n}{16}\right) + C_2 + C_2 \\ &= T\left(\frac{n}{64}\right) + C_2 + C_2 + C_2 \\ &= T\left(\frac{n}{256}\right) + C_2 + C_2 + C_2 + C_2 \end{aligned} \tag{2}$$

As we can see, the recursion grows as following, where k stands for the amount of recursions:

$$T(n) = T\left(\frac{n}{4^k}\right) + k \cdot C_2 \tag{3}$$

Since the rectangle will be inserted at the very last possible recursion, when n = 1 or n = 2, we can use that to form the closed form. In this case, we will use n = 1 since that is the simpler alternative to work with:

$$\frac{n}{4^k} = 1 \Leftrightarrow n = 4^k \Rightarrow k = \log_4(n) \tag{4}$$

For $\frac{n}{4^k}$ to be equal to one, k must be equal to $\log_4(n)$. By inducting this case on the formula we can form the closed form:

$$\begin{aligned} T(n) &= T\left(\frac{n}{4^k}\right) + k \cdot C_2 \\ &= T\left(\frac{n}{4^{\log_4(n)}}\right) + \log_4(n) \cdot C_2 \\ &= T\left(\frac{n}{n}\right) + \log_4(n) \cdot C_2 \\ &= T(1) + \log_4(n) \cdot C_2 \\ &= C_1 + \log_4(n) \cdot C_2 \\ &= \log_4(n) \cdot C_2 + C_1 \end{aligned} \tag{5}$$

Since $\log_4(n)$ is the highest order term, the worst case time complexity of the function is thus:

$$\Theta(\log(n)) \tag{6}$$

4 Function retrieveRects(l, x, y)

The function will recursively call itself to loop over each element/rectangle in the given list and check each rectangle if the point defined by x and y can be found within it. Thus it has a time complexity of O(n).

5 Function retrieveQuadrant(q, x, y)

Regardless the functions input values, the code will always execute in constant time and therefore its time complexity is O(1).

6 Function query(q, x, y)

6.1 Preamble

6.1.1 Size of the tree is constant

The n parameter in T(n) is equal to the height of the tree for this calculation, which is also made under the assumption that the size of the tree will be constant for each call to the function. Since this calculation is for the worst case scenario, the point (x,y) can be found within every single rectangle in the tree, meaning, all rectangles stored in the tree will be found along the way to the bottom of the tree, and the point (x,y) can be found within every single one of them.

6.1.2 Perfectly balanced tree

For this calculation, the amount of elements in each node's lists will be constant, meaning, the concatenation between the horizontal list and the vertical list will be of constant length for each node as the function traverse the tree. Thus it can be viewed as an operation with constant time. As well as that, all paths from the root node to the leaves are of the same length. The n parameter in T(n) is equal to the amount of nodes in the tree.

6.2 Calculation

6.2.1 Size of the tree is constant

When the code is executed it will go through all elements in the horizontal list one time when concatenating it with the vertical list, it will then go through all elements in the concatenated list in the retrieveRects function, and lastly it will go through all elements in the list that was returned from the retrieveRects function when it concatenates it with what was returned from the recursive call. To summarize it: $2 \cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}|$

The recursion stops when it can no longer traverse the tree, meaning, when the call to the function retrieveQuadrant returns an EmptyQuadTree. When this happens it will return an empty list with the constant time C. When there is one node (or more) it will traverse the node's lists as explained above. The recursive formula can thus be defined as:

$$T(n) = \begin{cases} C & \text{if } n = 0 \\ T(n-1) + 2 \cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| & \text{if } n \geq 1 \end{cases} \quad (7)$$

To explain the formula, when the height of the tree is equal to zero (n = 0) it returns with the constant time C. If the height is greater or equal to one, it will traverse the lists and make a recursive call to itself.

By expanding/rewriting the formula above we can form an idea on how to form the time complexity of the function. For each recursive call, the height of the tree will decrease by one.

$$\begin{aligned} T(n) &= T(n-1\cdot 1) + 1\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \\ &= T(n-1\cdot 2) + 2\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \\ &= T(n-1\cdot 3) + 3\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \\ &= T(n-1\cdot 4) + 4\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \\ &= T(n-1\cdot 5) + 5\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \end{aligned} \tag{8}$$

As we can see by expanding the formula, it grows as following, where k stands for the amount of recursions:

$$T(n) = T(n-1\cdot k) + k\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \tag{9}$$

Since every single rectangle stored in the tree will be in the direct path the function is going to traverse (due to that this is for the worst case scenario) the function will not stop until it has traversed the entire height of the tree. Thus k must be equal to n, since n is the height of the tree. By inducting this case on the formula, we can define the time complexity of the function:

$$\begin{aligned} T(n) &= T(n-1\cdot k) + k\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \\ &= T(n-1\cdot n) + n\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \\ &= T(n-n) + n\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \\ &= T(0) + n\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \\ &= C + n\cdot 2\cdot |\text{horizontalList}| + |\text{verticalList}| + |\text{retrieved}| \end{aligned} \tag{10}$$

Since n is the highest order term, the time complexity of the function is linear, where the n parameter is the height of the tree:

$$\Theta(n) \tag{11}$$

But, as stated in the preamble for this calculation, the size of the tree will be constant for each call to this function. And thus, the time complexity of the function will be of constant time.

$$\Theta(1) \tag{12}$$

6.2.2 Perfectly balanced tree

The recursive formula for this calculation is very similar to the one in the previous section regarding a constant sized tree. But instead of n being equal to the height of the tree, n is equal to the amount of nodes in the tree. As well as that, as stated in the preamble for this calculation, traversing each node's lists can be viewed as a constant time operation and will be referred to as C_2. C in the previous formula has been renamed to C_1.

$$T(n) = \begin{cases} C_1 & \text{if } n = 0 \\ C_2 + T(0) & \text{if } n = 1 \\ T\left(\frac{n}{4}\right) + C_2 & \text{if } n > 1 \end{cases} \tag{13}$$

To explain the formula, when there are no nodes to traverse in the tree, the function returns with the constant time C_1. If there are one or more nodes in the tree, it will

traverse that node's lists with constant time C_2 and make a call to retrieveQuadrant to find the next node to work on. In the case of n = 1 (there is only one node), it will traverse the lists then return with constant time C_1 due to the fact that the call to the function retrieveQuadrant returns EmptyQuadTree (the recursive call will be T(0)). By expanding/rewriting the formula we can form an idea of the time complexity of the function:

$$\begin{aligned} T(n) &= T\left(\frac{n}{4}\right) + C_2 \\ &= T\left(\frac{n}{16}\right) + C_2 + C_2 \\ &= T\left(\frac{n}{64}\right) + C_2 + C_2 + C_2 \\ &= T\left(\frac{n}{256}\right) + C_2 + C_2 + C_2 + C_2 \end{aligned} \tag{14}$$

The formula grows as following, where k stands for the amount of recursions:

$$T(n) = T\left(\frac{n}{4^k}\right) + k \cdot C_2 \tag{15}$$

The function will keep making recursive calls until there is no nodes left to traverse, this happens at the very last node e.g. the leaf. For this to happen n must be equal to one:

$$\frac{n}{4^k} = 1 \Leftrightarrow n = 4^k \Rightarrow k = \log_4(n) \tag{16}$$

When $\frac{n}{4^k}$ is equal to one, k is equal to $\log_4(n)$. By inducting this case on the formula we can define the time complexity of the function:

$$\begin{aligned} T(n) &= T\left(\frac{n}{4^k}\right) + k \cdot C_2 \\ &= T\left(\frac{n}{4^{\log_4(n)}}\right) + \log_4(n) \cdot C_2 \\ &= T\left(\frac{n}{n}\right) + \log_4(n) \cdot C_2 \\ &= T(1) + \log_4(n) \cdot C_2 \\ &= (C_2 + T(0)) + \log_4(n) \cdot C_2 \\ &= (C_2 + C_1) + \log_4(n) \cdot C_2 \\ &= \log_4(n) \cdot C_2 + C_2 + C_1 \end{aligned} \tag{17}$$

Since $\log_4(n)$ is the term of greatest order the time complexity of the function is thus:

$$\Theta(\log(n)) \tag{18}$$