Peter Murphy

DE CS II

Professor Kelly

31 March 2018

TicTacToe Hash Function

**Part 1**

Description: The hash function used in part 1 of the assignment maps any of the 19,683 possible 9 character String representations of a given TicTacToe Board into a single, unique integer ranging from 0-19,682 which corresponds to boolean array for O(1) lookups to verify whether or not the Board is a valid winner or not.

The core function of the algorithm works by assigning any of the 3 possible tokens of a board (‘ ‘, ‘x’, and ’o’) to an integer value 0, 1, or 2 respectively. Each of the 9 characters of the Board String are evaluated, and then multiplied by 3 raised to the power of their position in the string (0, 8). The hashcode of the Board is the sum of all of the evaluations of said characters, resulting in the unique integer value, with the lowest possible value of 0 corresponding to a blank Board ‘ ‘, and the highest possible value of 19,682 corresponding to a board full of ‘o’s ‘ooooooooo’.

Analysis: This method provides consistent linear-speed mapping of all possible Board values at the expense of a relatively large lookup table. Using this method, there are no collisions.

**Part 2**

Description: The hash function used in part 2 of the assignment is very similar to the function used in part 1, with the defining distinction being an attempt to reduce the size of the lookup table to a smaller number such as 36 or 37. This consolidation is achieved by applying the modulus operator to the value yielded by the first hash function and the new desired size of the look-up table 3n : n < 9.

This method relies on sufficiently quick resolution of colliding hash code values of unique boards. My second hash function implements the lookup table as an array of TreeNodes rather than simple boolean values so as to aim for O(1) and O(log n) as best and worst case run times while simultaneously cutting down on the size of the lookup table. I found that using values of 36 or 37 for the size of the winners array yielded the best ratio of collisions to chain size as represented in the attached analysis. The biggest challenge of reducing the size of the lookup table was retaining even distribution of values throughout the winners array which was achieved by using the % operator. Some visible grouping still occurred in the first and third quadrants of the lookup table. Although smaller array sizes such as 35 and 36 suffered from higher rates of collision, the worst possible case (every single Board sharing the same hashcode resulting in a single chain of size 19,683) would still only be O(log10 39) = 4.29, roughly four times slower than linear time. This function could be improved by modifying the algorithm for even distribution so that fewer spaces in the lookup table are left empty, and so that the average chain length would decrease.

Analysis:

Table[243] created for 1400 boards with 1172 collisions

loadFactor (numCollisions/SIZE) = 4.823045267489712

228 / 243 Slots filled --> 15 Empty slots

Entries per Quarter: [73, 49, 73, 48]

Collisions per Tenth: [12, 16, 17, 20, 20, 22, 20, 22, 23, 20]

Lowest Index Entry: 1, Highest index entry: 241

Chain Info:

Collisions: 1172

Biggest Chain: 136

Average Chain: 28.0

Chains with > 108 collisions: 5

-------------------------------------------------------

Table[729] created for 1400 boards with 956 collisions

loadFactor (numCollisions/SIZE) = 1.3113854595336076

444 / 729 Slots filled --> 285 Empty slots

Entries per Quarter: [219, 146, 219, 145]

Collisions/Tenth: [15, 36, 53, 22, 34, 54, 42, 36, 51, 43]

Lowest Index Entry: 4, Highest index entry: 724

Chain Info:

Collisions: 956

Biggest Chain: 47

Average Chain: 10.0

Chains with > 37 collisions: 14

-------------------------------------------------------

Table[2187] created for 1400 boards with 550 collisions

loadFactor (numCollisions/SIZE) = 0.25148605395518975

850 / 2187 Slots filled --> 1337 Empty slots

Entries per Quarter: [657, 437, 656, 437]

Collisions/Tenth: [11, 20, 26, 24, 38, 48, 66, 45, 77, 33]

Lowest Index Entry: 13, Highest index entry: 2176

Chain Info:

Collisions: 550

Biggest Chain: 14

Average Chain: 3.0

Chains with > 11 collisions: 54

**Part 3**

Description: The hash function used in part 3 of the assignment uses Java 8’s built in HashMap class, storing the Board Strings as the keys of the structure, and their respective boolean value (winner/loser : true/false) as the values. This method improves upon the functions used in parts 1 and 2 by utilizing the refined Java HashMap data structure largely due to the fact that the final size of the HashMap is known and defined as the number of “winner” Board Strings: ~1400 (accounting for missing/incorrect winner Boards in the supplied project file). Strings which do not exist within the winners HashMap are null → false → loser Boards. Collision chains within the winners HashMap whose size exceeds the given TREEIFY\_THRESHOLD (the default threshold is 8) are automatically converted from LinkedLists to Balanced Search Trees. The resulting best and worst case search times go from O(1) and O(n) to O(1) and O(log n)[[1]](#footnote-0). If the final size of the HashMap was unknown, this solution could become comparably slow as either of its two predecessors as the contents of the HashMap would have to be copied, rehashed, and collisions resolved across a new, larger HashMap which would drive the runtime speed upwards depending on the insertion time for each of the collisions.

Analysis:

Still figuring it out.

1. javarevisited.blogspot.sg/2016/01/how-does-java-hashmap-or-linkedhahsmap-handles.html?m=1 [↑](#footnote-ref-0)