Chris Good

Dan Schepers

Ethan Veatch

**Scrabble Datastructures and Implementation Plan**

***Scrabble Dictionary***

One of the major concerns for Scrabble is how to store the dictionary to enable quicker searches. We have decided to use a 2D Array of size 26 x 26 that will hold the valid word choices in the form of ArrayLists of strings. Each column in the 2D Array will correspond to words that begin with the letter at that position in the alphabet, starting with ‘A’ at column index 0 going through ‘Z’ at index 25. Each row will also represent a letter starting with ‘A’ at row index 0 going through ‘Z’ at row index 25. Each location within the column will contain an ArrayList made up of words that start with the corresponding column letter and containing the letter for the row. As a basic example we will look at the word tow. Tow would be found at Array[19][14] and at Array[19][22]. The first index is 19 in both situations because ‘t’ is the 20th letter of the alphabet and we must subtract 1 due to 0-based indexing. The second indexes correspond to the location of ‘o’ and ‘w’ in the alphabet when adjusted for 0-based indexing. While this is not incredibly space efficient the other benefits should outweigh the cost of space used.

Table 1: A partial diagram of the 2D Array for the dictionary. The ArrayList at each position stores words beginning with the capital letter and containing the lower case letter. The actual structure will have cells for each letter.

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Aa | Ba | Ca | Da | Ea | Fa | Ga | Ha | Ia | Ja | Ka | La | Ma |
| Ab | Bb | Cb | Db | Eb | Fb | Gb | Hb | Ib | Jb | Kb | Lb | Mb |
| Ac | Bc | Cc | Dc | Ec | Fc | Gc | Hc | Ic | Jc | Kc | Lc | Mc |
| Ad | Bd | Cd | Dd | Ed | Fd | Gd | Hd | Id | Jd | Kd | Ld | Md |
| Ae | Be | Ce | De | Ee | Fe | Ge | He | Ie | Je | Ke | Le | Me |
| Af | Bf | Cf | Df | Ef | Ff | Gf | Hf | If | Jf | Kf | Lf | Mf |
| Ag | Bg | Cg | Dg | Eg | Fg | Gg | Hg | Ig | Jg | Kg | Lg | Mg |
| Ah | Bh | Ch | Dh | Eh | Fh | Gh | Hh | Ih | Jh | Kh | Lh | Mh |
| Ai | Bi | Ci | Di | Ei | Fi | Gi | Hi | Ii | Ji | Ki | Li | Mi |
| Aj | Bj | Cj | Dj | Ej | Fj | Gj | Hj | Ij | Jj | Kj | Lj | Mj |
| Ak | Bk | Ck | Dk | Ek | Fk | Gk | Hk | Ik | Jk | Kk | Lk | Mk |
| Al | Bl | Cl | Dl | El | Fl | Gl | Hl | Il | Jl | Kl | Ll | Ml |
| Am | Bm | Cm | Dm | Em | Fm | Gm | Hm | Im | Jm | Km | Lm | Mm |

Our 2D Array of ArrayList<String>’s will then be populated by creating a new file reader to accept the dictionary (.sd) file. From there we will use a BufferedReader to add Strings representing each line in the file to the correct ArrayList(s) within the Array. This setup allows us to quickly create our dictionary without using unreasonable amounts of memory. The creation of the 2D Array using this method should be fairly fast and allow for quick lookups and comparisons later.

To find the list of playable words we would initially populate a list with words based on tiles currently held in the hand. The source for this list would be the Array we used to hold the dictionary. For each letter in the hand we would copy the words from the corresponding row and column to an ArrayList<String>, allowing duplications for having multiple of the same letter tile. We could then use that ArrayList along with the tiles on the board to create a full list of possible playable words and remove those that do not work for each position on the board.

***Scrabble***

Given the list of the possible words from the hand that is created by the ScrabbleDictionary class, we can now determine which words are actually playable. First we would further expand the list to include possible words for the available letters on the board as well. An open letter is any letter on the board that can be used to make a word. We can help to keep the list somewhat shorter by checking to see if a letter on the board can be used within the word or only at the beginning. If the letter can only be used at the beginning of the word it is only necessary to add words from the ArrayLists within that letter’s column. Similarly if the letter can only be used as the end of a word, only the words from the ArrayLists within that letter’s row need be added. These situations are depicted in Figure 1.

In Figure 1 tiles that would be classified as available are shown with a color background. Blue tiles can occur anywhere within a word- beginning, middle, or end. Green blocks can occur only at the beginning of a word and red blocks only at the end. By being able to establish this information we can keep the memory usage for the ArrayList holding possible moves as low as possible.

Figure 1: Available tile configurations

In addition to the major data structures already mentioned, it is worth noting that several other fields and local variables will be needed. It will be necessary to have Boolean values to distinguish between horizontal and vertical, declare a placement valid or invalid, and to mark the first turn of the game. The need for other Boolean values may arise during implementation, but these are all that we can foresee. Many methods will also require a temporary ArrayList<String > to be made, usually as a copy of an array stored as a field in order to perform actions that would manipulate the field without losing information. It is also worth noting that many data structure choices were already made in the original project and we intend to use most, if not all, of the pre-existing fields and local variable