**External Documentation**

**Overview**

Construct crossword is such that where words are allowed to be filled should be pre-filled with “.” if no word is there, and if word can be filled at those position, then fill that with that word, and positions where the words are not allowed to be filled, should be kept empty.

The main goal of the crossword is to ensure that all the words given in the list are filled in the appropriate area where “.” is there and all the empty spaces where no word can’t be filled is kept as it is.

**Input**: no of rows in the puzzle, no of columns in the puzzle, no. of words in the puzzle, column and row position from where the word starts in the puzzle and no. of words that can be filled in that, the orientation (horizontal or vertical) in which those words will get filled.

**Files and External Data**

1. FillInPuzzle.java - class that implements the data structure. It implements two interfaces – searchable.java and TreeDebug.java

2. Test.java - class that creates node for placing the values of the data-structure.

Input is taken from a String Reader and output is stored in new file – solution.txt

Initially, I thought to create following separate classes

1. **Grid** – which creates the 2d grid and stores all the relevant values in it.
2. **Solution** – which solves and prints the solution.
3. **Word –** which adds the word and iterates over it.

**Data Structures and their relation to each other.**

**loadPuzzle(BufferedReader instream)**

- This just reads the stream, iterates over part 2, and part 3, and stores the values into variables and arraylist.

For loading the puzzle, have used following variables

**PART 1**

n\_columns\_puzzle: Holds the no. of columns

n\_rows\_puzzle: Holds the no. of rows

n\_words\_puzzle: Holds the no. of words given as an input to the puzzle

**PART 2**

start\_col\_pos = holds the starting column index of the word.

start\_row\_pos = Holds the starting row index of the word.

n\_letters\_word = Holds the no. of letters in that word.

word\_orient = Holds the orientation of the word.

Adds all the 3 values of the variables above iteratively in the following list –

start\_col\_positions, start\_row\_positions, alln\_letters\_word and all\_word\_orient

**PART 3**

words – stores the arraylist of all the words. Iterates uptil n\_words\_puzzle

**construct2dgrid()**

- There are 2 constraints inherent in the problem statement because of which I somewhat struggled with this

1. Column and row numbers start at 0. The bottom left corner is column 0, row 0. So, considering the input given in the assignment – the rowIndex would be 4,3,2,1,0 in the reverse order. It prints the grid with column 0, row 0 in the bottom left corner (so prints the maximum row first).

2. The words in the puzzle all fill from left to right or from top to bottom.

- For ensuring the grid is in proper format, I used the current input as the reference and on the basis of that, I found pattern in it by doing the following iterations on pen and paper and also on keyboard.

**For word\_orient - horizontal**

if(start\_row\_positions.get(0) == 0 && start\_col\_positions.get(0) == 0)

{

for(int k=0;k<alln\_letters\_word.get(0);k++) {

b[4][k] = ".";

}

}

if(start\_row\_positions.get(1) == 2 && start\_col\_positions.get(1) == 1)

{

for(int k=1;k<alln\_letters\_word.get(1)+1;k++) {

b[2][k] = ".";

}

}

**For word\_orient - vertical**

if(start\_row\_positions.get(2) == 4 && start\_col\_positions.get(2) == 1)

{

for(int k=0;k<=4;k++) {

b[k][1] = ".";

}

}

if(start\_row\_positions.get(3) == 3 && start\_col\_positions.get(3) == 4)

{

for(int k=1;k<=alln\_letters\_word.get(3);k++) {

b[k][4] = ".";

}

}

By iterating it row by row and column by column and taking small steps, I was able to see the pattern in that and on the basis of that I have written the code for this.

**Assumptions**

None

**Choices**

- Have used string as the grid.

- Have used arraylist for storing words.

**Limitations**

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

- The current design accepts just the input given on the assignment. It’s not tested against any other inputs. So, when any other input other than the one given in the assignment is given, please test it. Will test it in the revised version.

- I was able to implement the code to read in the puzzle, and also was able to print the solution, but due to time constraint, was not able to write the code to solve the puzzle. Will do it in the revised version.

**Test Cases**

**Boolean loadPuzzle (BufferedReader stream)**

**Input Validation**

- stream is null or empty, return false

**Invalid input scenarios**

- In part 1, no. of words (n) in the puzzle is 4, but only 3 words are provided in part 3, return false.

- In part 1, no. of words (n) in the puzzle is 4, but only 1 word is provided in part 3, return false.

- In part 1, no. of words (n) in the puzzle are 4, but no words are provided in part 3, return false.

- In part 1, no. of words (n) in the puzzle are 4, but 5 words are provided in part 3, return false.

- Puzzle just has part 1, return false.

- Puzzle just has part 2, return false.

- Puzzle just has part 3, return false.

- Puzzle has just part 1 and part 2, return false.

- Puzzle has just part 2 and part 3, return false.

- Puzzle has just part 1 and part 3, return false.

- In part 2, no. of words that a position can contain are 5, but in part 3, there are no words with 5 letters, return false.

- No. of columns is not given in part 1, return false.

- No. of columns is non-negative, return true.

- No. of rows is not given in part 1, return false.

- No. of rows is non-negative, return true.

- No. of words is not given in part 1, return false.

- 3 Integers are separated by spaces in part 1, return true.

- Part 2 consist of exact n rows (no. of words), return true.

- Part 2 has 3 integers followed by one letter, return true.

- All the 3 integers in the entire part 2 is non-negative, return true.

- The letter at the end in part 2 can only be either h or v, return true.

- All integers in part 1, part 2 and part 3 are non-negative, return false.

- No word is repeated in part 3, return true.

**Boundary Validation**

- There are more than 3 integers in part 1, return false.

- There is just 1 integer in part 1, return false.

**Control flow**

- First read the part 1, then part 2, and then part 3 in a sequential manner.

**void print( PrintWriter solvedPuzzle)**

**Input validation**

- solvedPuzzle is null, false

- grid is null, false

**-** value of i<0 and value of j<0**,** false

- grid is empty, false.

- test against large no. of rows and large no. of columns, if everything works as expected, return true.

**Boundary condition**

- grid is a string array

- rows and columns inside the grid are integer values >= 0.

**Control flow**

**Data Flow**

**Boolean solve()**

- not able to fill the word, return false.

- successfully able to fill the word, return true.

**int choices()**

- Not getting the exact choices made during the execution of solve, return false.

**Develop a strategy on how you will solve the puzzle before you finalize and start coding your data structure(s) for the puzzle**

We’ll use backtracking to solve the crossword puzzle.

1. Start with the grid with empty spaces and “.” arranged in proper places, as per the input.
2. Start with the first word and Iterate over the entire grid with the condition that if grid[i][j] == “.” Or the first character of any word matches with the first character in the grid.
3. If the no. of letters in the word exactly and perfectly matches the no. of dots in that particular row, then place it horizontally.
4. After the word is placed successfully in horizontal orientation, then start with the next word. Continue from 2 to 4 until all the words are placed successfully. Once all the words are fit successfully to the number of dots in the grid, then increment the no. of guesses variable by 1.
5. If the no. of letters in the word exactly and perfectly matches the no. of dots in that particular column, then place it vertically.
6. After the word is placed successfully in vertical orientation, then start with the next word. Continue from 2,5, and 6 until all the words are placed successfully in vertical orientation. Once all the words are fit successfully to the number of dots in the grid, then increment the no. of guess variable by 1.
7. Word doesn’t fit in any of the available and matching grid positions, then backtrack to the previous word and try different position for that previous word and continue doing that until a solution is found.