```python
from collections import deque .

example = "BB0     0   XX0        2    A2111 A2"
ex_hard = "02111A02BB4A02XX4CDDE 4C  EFF  HHJJ "

alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
evens = {character for character in alphabet[::2]}  # Even numbered/lettered vehicles will be used for up-down
odds = {character for character in alphabet[1::2]}  # Odd numbered/lettered vehicles will be used for left-right
cars = {character for character in alphabet[:26]}   # Letters are used for cars
trucks = {character for character in alphabet[26:]} # Numbers are used for trucks

# The dictionary below answers the question "If I move a character in a certain direction,
# how does that change its index in the string representing the game state?"
direction_to_index_shift = {"U": -6, "D": 6, "R": 1, "L": -1}

def print_puzzle(state):
    """ Prints out the puzzle, formatted nicely.  Uses ">" to mark the exit. """
    print("/------\\")
    for row in range(6):
        char = "|"
        if row == 2:
            char = ">"
        print("|" + state[row*6:(row+1)*6] + char)
    print("\\------/")

def build_puzzle():
    """ Allows the user to specify a starting state. Comments and prompts assume zero-indexing. """
    puzzle = []
    for index in range(36):
        puzzle.append(" ")
    red = int(input("From 0 to 4, left to right, what is the left-most index of the red car? "))
    puzzle[12 + red] = "X"  # The red car is always 'XX' and is always in row 2.
    puzzle[12 + red + 1] = "X"
    print_puzzle("".join(puzzle))
    v_car, h_car, v_truck, h_truck = 0, 1, 26, 27  # Keeps track of which index in the alphabet to use for each type
    while True:
        char = input("Input a size 2 car (C) or size 3 truck (T) or solve (S)? ")
        if char == "S":
            break
        elif char == "C":
            code = "C"
            size = 2
        elif char == "T":
            code = "T"
            size = 3
        else:
            continue
        char = input("Does it face up and down (U) or side to side (S)? ")
        if char == "U":
            code += "U"
            r = int(input("From 0 to 4, top to bottom, what is the top-most index of the vehicle? "))
            c = int(input("From 0 to 5, left to right, what is the index of the column of the vehicle? "))
            index = r * 6 + c
            step = 6  # How far is it from each index of this vehicle to the next one?  6 if vertical.
        elif char == "S":
            code += "S"
            r = int(input("From 0 to 5, top to bottom, what is the index of the row of the vehicle? "))
            c = int(input("From 0 to 4, left to right, what is the left-most index of the vehicle? "))
            index = r * 6 + c
            step = 1  # How far is it from each index of this vehicle to the next one?  1 if horizontal.
        else:
            continue
        if code == "CU":
            character = alphabet[v_car]
            v_car += 2  # Use the next available vertical car character next time the user places one.
        elif code == "CS":
            character = alphabet[h_car]
            h_car += 2
        elif code == "TU":
            character = alphabet[v_truck]
            v_truck += 2
        elif code == "TS":
            character = alphabet[h_truck]
            h_truck += 2
        for count in range(size):  # Place the car on the board.
            puzzle[index + count * step] = character
        print_puzzle("".join(puzzle))  # Let the user see the board each time.
    return "".join(puzzle)
```

```python
def move(st, index, dir, dist):
    """ Precondition: the input specifies a valid move. (dir is direction; dist is distance)
        Postcondition: returns the board state with that move made.
        Note: "index" specifies the leading index of the vehicle in the direction of movement;
        ie, if the car is moving left, "index" is the leftmost index of the car. """
    state = list(st)
    character = state[index]
    diff = direction_to_index_shift[dir]
    state[index] = " "                    # The next four lines delete the car from the board
    state[index - diff] = " "             # (Note the rest of this car is opposite the direction of movement)
    if character in trucks:
        state[index - 2 * diff] = " "
    state[index + diff * dist] = character         # The next four lines add the car in the new location
    state[index + diff * (dist - 1)] = character
    if character in trucks:
        state[index + diff * (dist - 2)] = character
    return "".join(state)

def get_spaces(state, index, dir):
    """ Given an index, this determines how many blank spaces are present between that index and
        the edge of the board in a certain direction.  Easier for up and down (too far is just out
        of bounds); harder for left to right, hence the additional if statements to check if we're
        at either the left or right edge of a row."""
    diff = direction_to_index_shift[dir]
    total = 0
    if dir == "L" and index % 6 == 0:
        return total
    if dir == "R" and index % 6 == 5:
        return total
    index += diff
    while index >= 0 and index < 36 and state[index] == " ":
        total += 1
        if dir == "L" and index % 6 == 0:
            return total
        if dir == "R" and index % 6 == 5:
            return total
        index += diff
    return total

def get_children(state):
    """ On any given board state, in this implementation, ODD numbered/lettered cars face left
        to right, and EVEN numbered/lettered cars face up and down.  This function finds all the
        ODD cars that can move left or right and generates those children, then all the EVEN cars
        that can move up or down and generates those children. """
    children = []
    for index, character in enumerate(state):
        if character in odds:
            for count in range(1, get_spaces(state, index, "L") + 1):  # Note if there are no spaces, this is skipped
                children.append(move(state, index, "L", count))  # Generate all children in this direction.
            for count in range(1, get_spaces(state, index, "R") + 1):
                children.append(move(state, index, "R", count))
        if character in evens:
            for count in range(1, get_spaces(state, index, "U") + 1):
                children.append(move(state, index, "U", count))
            for count in range(1, get_spaces(state, index, "D") + 1):
                children.append(move(state, index, "D", count))
    return children

def goal_test(state):
    """ Tests to see if this is a victory state; ie, is the red car (XX) immediately adjacent to
        the exit? """
    if state[16:18] == "XX":
        return True
    return False

def bfs_search(state):
    """ BFS search to find the ideal path; returns a tuple of visited boards along the way.
        Fun fact: this is completely identical to the code I used for peg solitaire! """
    visited = {state}
    fringe = deque()
    fringe.append((state, ()))
    while len(fringe) > 0:
        position, path = fringe.popleft()
        path = path + (position,)
        if goal_test(position):
            return path
        for child in get_children(position):
            if child not in visited:
                visited.add(child)
                fringe.append((child, path))


for state in bfs_search(build_puzzle()):
    print_puzzle(state)
```