**Exercise 1**

1. Game State

The game state is the current state of the game, where pacman is, what he’s doing, where the ghosts and food are and in what state they are. It also contains the world of the game, the layout of the level, the score at a certain point in time.

The game state contains a lot of accessor methods to acces a GameStateData object. If one of the things in the level has to make a certain action, the corresponding method in an object gets called.

1. a
2. A – III : similarly to a stack, the crate on top of the pallet is removed first, and new crates will be added on top. It is not possible to remove a crate from the middle of the pallet either.

B – II : a queue in code form works like the line in a supermarket (provided no-one is being polite and letting someone go before them), where the first one to get in line is the first one who will be helped by the cashier.

C – I : a hospital room works like a priority qeue, the person with the worst injury will be helped first.

1. StayEastSearch is intended to prefer a path through the eastern part of a level. This is represented in the code by using uniform cost search and setting the cost function costFn used to calculate the cost of a step to lambda pos: .5 \*\* pos[0]. CostFn is given (x,y) as input. This formula take the x component and increases the cost by calculating ½ to the power of x. The result for a higher x (which is the east side of the level) is much lower than the result for a lower x. Uniform cost chooses the step with the lowest cost, and will thus go for the fields in the east side of the level.

StayWestSearch is intended to prefer the west side of a level. Very similar to StayEastSearch, the difference is in the costFn. For StayWestSearch, it is lambda pos: 2 \*\* pos[0]. This achieves the opposite of StayEastSearch because for higher values of x the cost 2 to the power of x will be a lot bigger than for lower values of x.

Both these agents have an exponential increase in cost, so the differnence between costs in one half of the level and the other will be absolutely huge and both will stick to their preferred side of the level.

**Exercise 2**

Our general structure is pasted below.

**(4)** We use a fringe/frontier, which can be any of the things found in util.py, depending on the a search algorithm.

**(5-6)** For the solution and the tracking of the explored nodes we use lists.

We fill the solution with Directions which pacman can use to move around.

For the explored nodes, we use a list named closed. A list comes with a lot of useful functions, such as checking if a node is already in there or not.

**(9)** We represent nodes as (currentLocation, (previousLocation, directionFromPrevToCurr)). This makes it easier for us to construct a solution when the algorithm finds the goal. For something as uniform cost search, cost would be included as well, making nodes look like :  
(currentLocation, (previousLocation, directionFromPrevToCurr, costForThisStep)).

**(11)** There’s and empty tuple named goal. This will later be used to store the goal node, and it is also used to construct the solution list.

**(13-24)** Our program then enters a while loop, executing its steps as long as the fringe is not empty.

The pop() action is called for the fringe.

A check is made to see if this node is already in the closed list. If not, it is added. closed is changed into a dict here, which Kevin is changing

A second check is made to see if the node is the goal node. If this is the case, the goal is set to be current node we are working on. The while loop is now interruped and the solution list will be build.

If not, we get the successors of the current node. These are added to the fringe using the push() action. This means there will probably be duplicates in the fringe. Because we check the closed list before working on a node, these will be discarded when they are popped from the fringe.

**(26)**  The closed list is converted to a dictionary

from game import Directions

s = Directions.STOP

fringe = ???

closed = []

solution = []

startLocation = problem.getStartState()

node = (startLocation, (startLocation, s))

fringe.push(node)

goal = ()

while not fringe.isEmpty():

node = fringe.pop()

if node[0] not in dict(closed):

closed.append(node)

if problem.isGoalState(node[0]):

goal = node

break

successors = problem.getSuccessors(node[0])

for successor in successors:

fringe.push((successor[0], (node[0], successor[1])))

closed = dict(closed)

while goal[0] in closed and goal[1] != s:

solution.append(closed[goal[0]][1])

goal = closed[goal[0]]

solution.reverse()

solution = solution[1:]

return solution

**Exercise 3**

1. Our solution is complete. Because of the graph search, infinite depth in trees is not a problem. The algorithm will explore the entire tree of nodes that it generates. A solution will be found.
2. It is not a least cost solution. Because the algorithm ‘dives’ into the tree as far as it can before trying a different way, it might return a terribly long, but working, path. A shorter, more efficient path might exist. The cost of steps is also ignored, which is also a problem.
3. **Missing.** Pacman does not visit all the explored tiles on the way to his goal if the goal was not found in one try. Otherwise, there is a straight path of explored nodes through the tree of nodes.

**Exercise 4**