Homework # 12
due Monday, May 3, midnight

In this assignment, you will implement "depth-first," "breadth-first," and "uniform-cost" (based on path cost) search to find paths through mazes on hex boards, and then display them on the screen. The search code will use a Worklist interface so as to be agnostic over the kind of search that is done. This way, you can achieve a different search algorithm by changing the Worklist used. Please (re-)read Chapter 14 in the textbook.

1 Concerning the Worklist ADT

A worklist contains things that need to be done; getting something from the worklist implicitly also removes it from the worklist. A worklist also can be added to. Different kinds of worklists are distinguished by how they prioritize items within the list. We will use three kinds of worklists for this homework:

LIFO The most recently added item is the highest priority item.

FIFO The item that has been in the list longest is the highest priority item.

Priority The first (least) in order (using a comparator, passed to the constructor) is the next one to be removed.

You will implement these worklists from scratch! You should use Java standard library implementations of Stack, Queue and PriorityQueue to implement the respective worklists. We will learn about the heap data structure soon, which is (Ch. 10.1) classically used in PriorityQueue implementations, but the ADT should be pretty clear. The interface is almost the same as for Queue: use `add` and `remove` to enqueue and dequeue elements, respectively.

The Worklist ADT provides the following methods:

hasNext() Whether the worklist still has work to do.

next() Remove and return the next item to do. Throws `NoSuchElementException` is the worklist is empty.

add(x) Add an item to the worklist.

Since we build the Worklist ADT on top of the Iterator ADT, there is also a method `remove` which has no meaning and for which an "unsupported operation exception" should be thrown. In other words, you don't need to implement this method.

2 Concerning the HexPath ADT

The search algorithm will find paths on boards of hexagonal tiles. A path is a series of adjacent hexagonal coordinates. We implement the path with an *immutable* class: once an instance of the class is constructed it will never change. Immutable classes are easier to reason about since they don't change, and don't require invariant checking except at the end of their constructors,

Unusually for immutable list classes, a `HexPath` is structured for easy access to the *end* of the path, where it might grow as we try to find longer paths to reach a specified goal. Normally, we

would override `equals` and `hashCode` for such classes, but to keep this homework in bounds, we are omitting them from this assignment.

The `HexPath` class has two (public) constructors. (Additionally a private constructor is used by the invariant checker tester.) In Java, one constructor can delegate to another using the `this(...)` syntax. The constructor should refuse to create bad paths.

A hex path can be drawn on the screen: it should be drawn as a series of line segments (use `Graphics#drawLine`) from centers of hexagons along the path. The number of segments will equal the size. We use the default width in class `HexTile`.

3 Concerning the HexDirection Enumeration

The graph we are searching is given implicitly by a `HexBoard`. To check which `HexTile`s are adjacent, we need check if there is a terrain at any of the 6 neighboring locations. It is helpful to have a list of the six directions one can travel from a tile. The `HexDirection` enumeration provides this capability and most importantly a way to determine the coordinate one reaches when traveling this direction from a given tile. The best way to use this enumeration in this programming assignment is in a loop:

```
for (HexDirection d : HexDirection.values()) {
   ...
}
```

If you find yourself using the specific directions (e.g., `NORTHEAST`) you are probably doing the assignment wrong.

4 Concerning the Hex Path Coster

In order for a priority search to choose the "best" path, we need need some way to specify what makes one path better than another. The priority worklist uses a comparator, and so we need a comparator for hex paths. We do this in three layers: first, we indicate how expensive it is to travel over each kind of terrain; next we use this information to compute a cost for any hex path, and third we compare two hex paths by comparing their costs.

The `HexPathCoster` class has the following public methods:

HexPathCoster(board) Create a coster in which every terrain has the same minimal cost, except for "INACCESSIBLE" which has maximum cost.

setCost(Terrain,int) Set the cost for crossing the given terrain.

getCost(Terrain) Query the information set in the previous method.

getCost(HexPath) Compute the cost of a path. Each *half* tile of terrain costs according to the terrain on the board. For example, if a path starts in a city, goes through a forest tile and ends up on a mounting, the cost is total of the city cost, *twice* the forest cost and the mountain cost. The forest cost is doubled because we go into it and then go out of it. In Java integer mathematics can overflow; we do not want a very high-cost path to suddenly become very cheap because of overflow. If the cost would overflow, you should return `Integer.MAX_VALUE` instead. The first symptom of overload is a negative value (but if you continue adding without checking, it may end up positive).

compare(HexPath,HexPath) This method is the one that makes this class a comparator. Implement it using the relative costs of the two paths. Two hex paths whose costs overflow will be considered the same cost; don't worry about that.

5 Concerning Search Algorithms

Section 14.3 of the textbook describes two important graph algorithms: "depth-first search" and "breadth-first search." As described on page 731, these algorithms can be used to find a path from one node to another in a graph. In our case, the nodes are hexagonal coordinates and the graph is represented by a hex board. Two hex coordinates are adjacent if the distance between the two is one, and if there is a tile at each coordinate and neither tile is "INACCESSIBLE".

Depth-first search is implemented using a stack (LIFO) for the worklist, breadth-first using a queue (FIFO). Priority-based search uses a priority queue. In "Artificial Intelligence" (CompSci 422) you can learn about more efficient ways to find the best path.

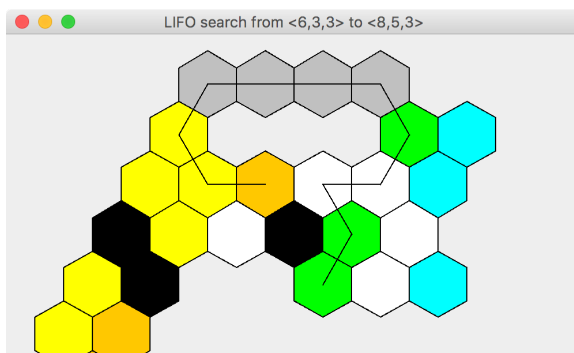
The search capability is in a class `Search`; it takes two coordinates (starting and destination coordinates) and a board (a map from coordinates to tiles). Assuming the first coordinate is occupied and accessible on the board, we start searching from there. Every coordinate that we traverse in the search is marked as "visited." which persists after the search is done. Unlike in the textbook, we break off the search if we find the destination node and return the path from the starting coordinate to the destination. If the destination is never found, we return null.

Do not check if the last node of a path is visited before you add the path to the worklist. Only check as it comes out of the worklist. Otherwise you will fail tests. More fundamentally, checking before you add to the worklist is not sufficient, and thus you would need checks in at least two places. It is cleaner to always check things in a single place.

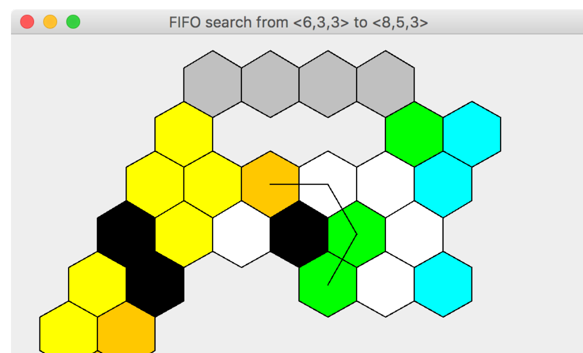
The main program `HexPathFinder` displays the board and after doing the search either displays the path, or shows all the hex tiles that were visited by drawing a large "X" on each tile (the bars of the "X" are one quarter of the length of the width of a hexagonal tile).

5.1 Examples

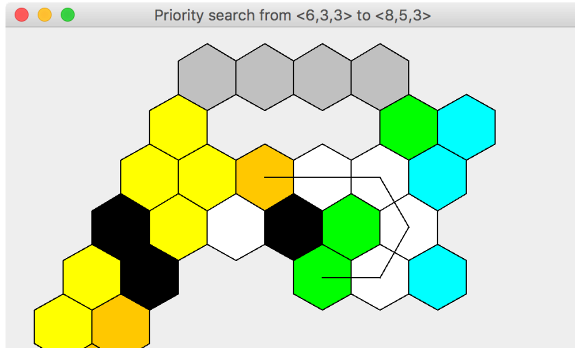
For each of the following examples, we show the arguments to use in the run configuration.



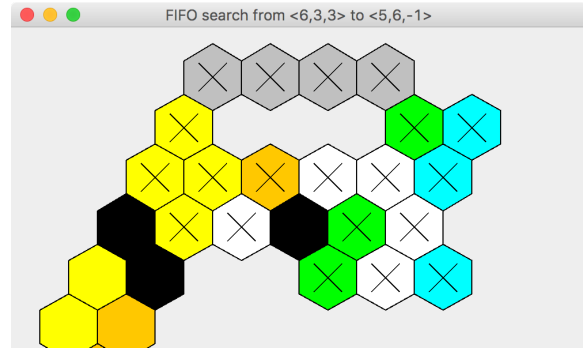
`lib/test.hex LIFO <6,3,3> <8,5,3>`



`lib/test.hex FIFO <6,3,3> <8,5,3>`



`lib/test.hex Priority <6,3,3> <8,5,3>`



`lib/test.hex FIFO <6,3,3> <5,6,-1>`

6 What You Need To Do

1. Implement `LIFOWorklist`
2. Implement `FIFOWorklist`
3. Implement `PriorityWorklist`
4. Complete `HexPath`
5. Complete `HexPathCoster`
6. Complete `Search`

When you commit your code, make sure to check the boxes for the new classes you are adding, or else these classes will *not* be added to your repository!

7 Files

The git repository has an Eclipse project with the following files:

src/Test{LIFO,FIFO,Priority}Worklist.java Units tests of three worklist classes.

src/TestHexPath.java Unit test of the HexPath ADT.

src/TestCoster.java Unit test of the `HexPathCoster` class.

src/TestSearch.java Unit test of the Search algorithm. It also can be run as a Java application to graphically debug the search algorithm by showing the board and a path computed by one of the tests (you choose which), and optionally shows the "visited" sets for one of the three algorithms (see "View" menu).

src/edu/uwm/cs351/HexPath.java Skeleton of HexPath ADT.

src/edu/uwm/cs351/HexPathCoster.java Skeleton of HexPathCoster implementation.

src/edu/uwm/cs351/Search.java Skeleton of Search algorithm.

lib/test.hex Sample hex board.