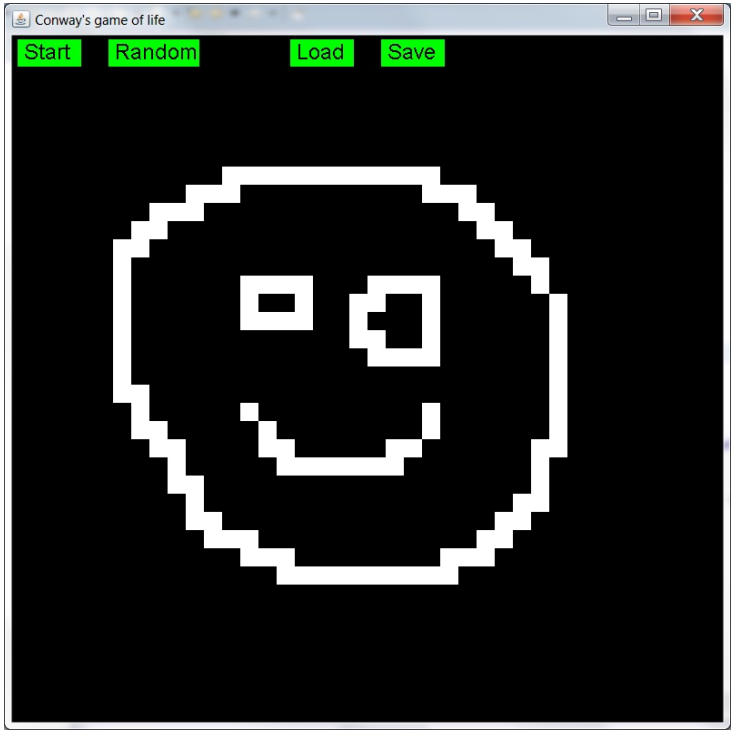
CT255
NGT2

Week 9
[2D Games in Java]

Dr. Sam Redfern
sam.redfern@universityofgalway.ie

Last week's assignment [Conway's Game of Life]

- Implement mouse dragging for game state setup
- Implement game state loading and saving (via 'buttons' as before)
- Read the following A* webpage for next week!



http://www.psychicsoftware.com/AStarForBeginners.html

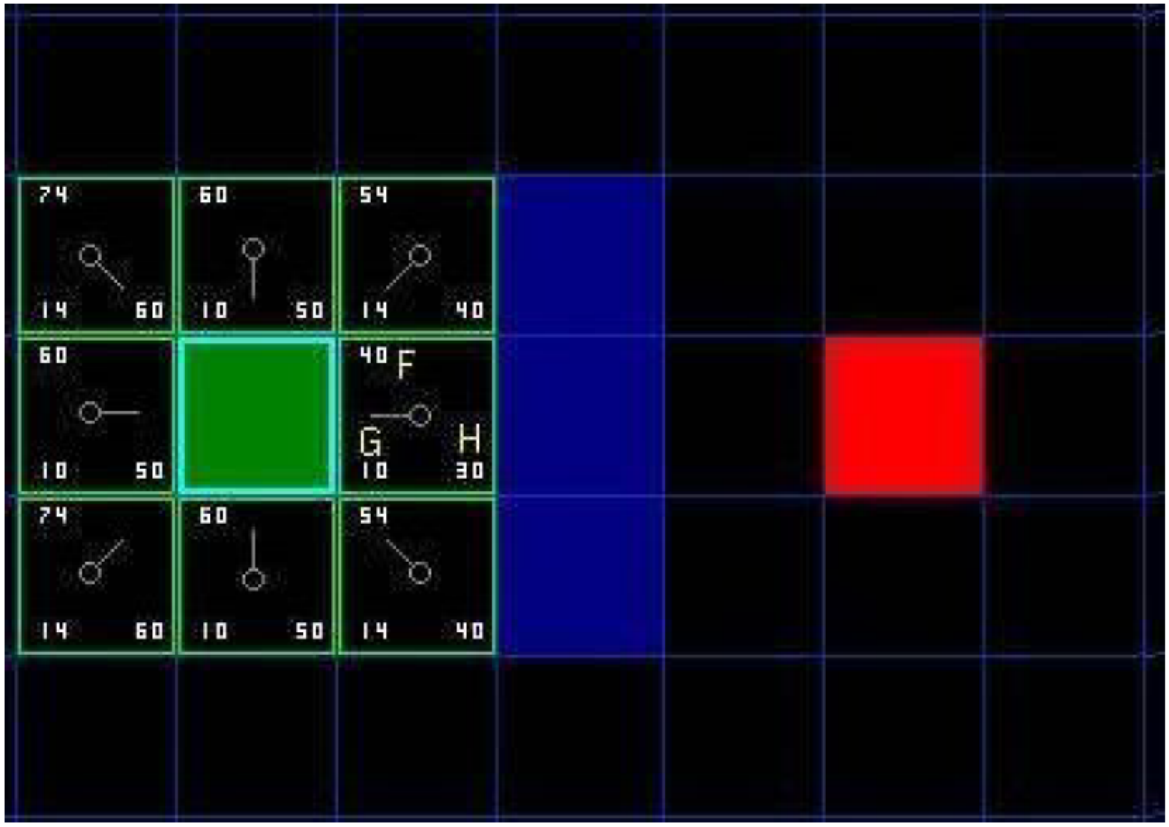
A* Pathfinding

- The fundamental operation of the A* algorithm is to traverse a map by exploring promising positions (nodes) beginning at a starting location, with the goal of finding the best route to a target location.

- Each node has four attributes other than its position on the map:
  - *g* is the cost of getting from the starting node to this node
  - *h* is the estimated (heuristic) cost of getting from this node to the target node. It is a best guess, since the algorithm doesn't (yet) know the actual cost
  - *f* is the sum of *g* and *h*, and is the algorithm's best current estimate as to the total cost of travelling from the starting location to the target location via this node
  - *parent* is the identity of the node which connected to this node along a potential solution path

A* Pathfinding

- The algorithm maintains two lists of nodes, the *open* list and the *closed* list.
- The OPEN LIST consists of nodes to which the algorithm has already found a route (i.e, one of its connected neighbours has been evaluated or *expanded*) but which have not themselves, yet, been expanded.
- The CLOSED LIST consists of nodes that have been expanded and which therefore should not be revisited.
- Progress is made by identifying the most promising node in the open list (i.e., the one with the lowest *f* value) and expanding it by adding each of its connected neighbours to the open list, unless they are already closed.
- As nodes are expanded, they are moved to the closed list.
- As nodes are added to the open list, their *f, g, h* and *parent* values are recorded.
- The *g* value of a node is, of course, equal to the *g* value of its parent plus the cost of moving from the parent to the node itself.



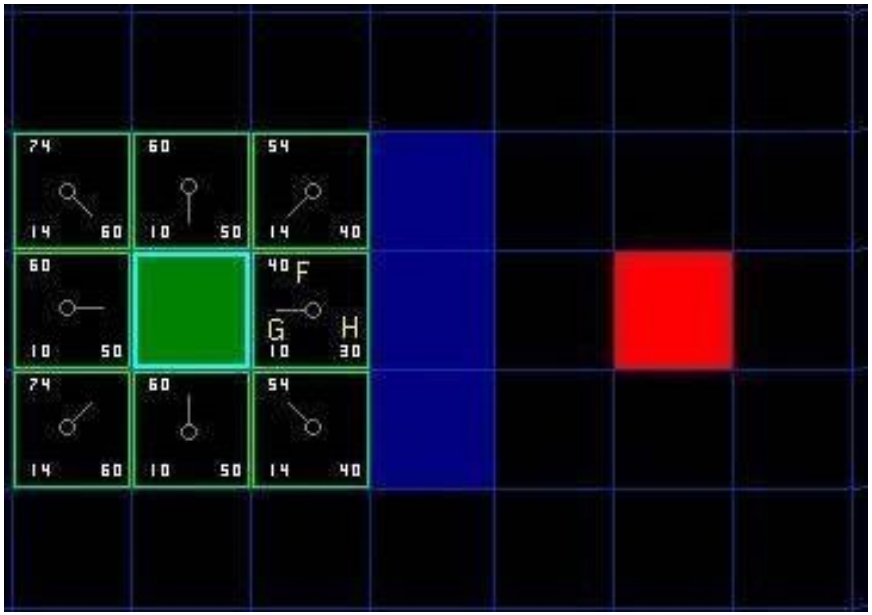
known cost from start to this node

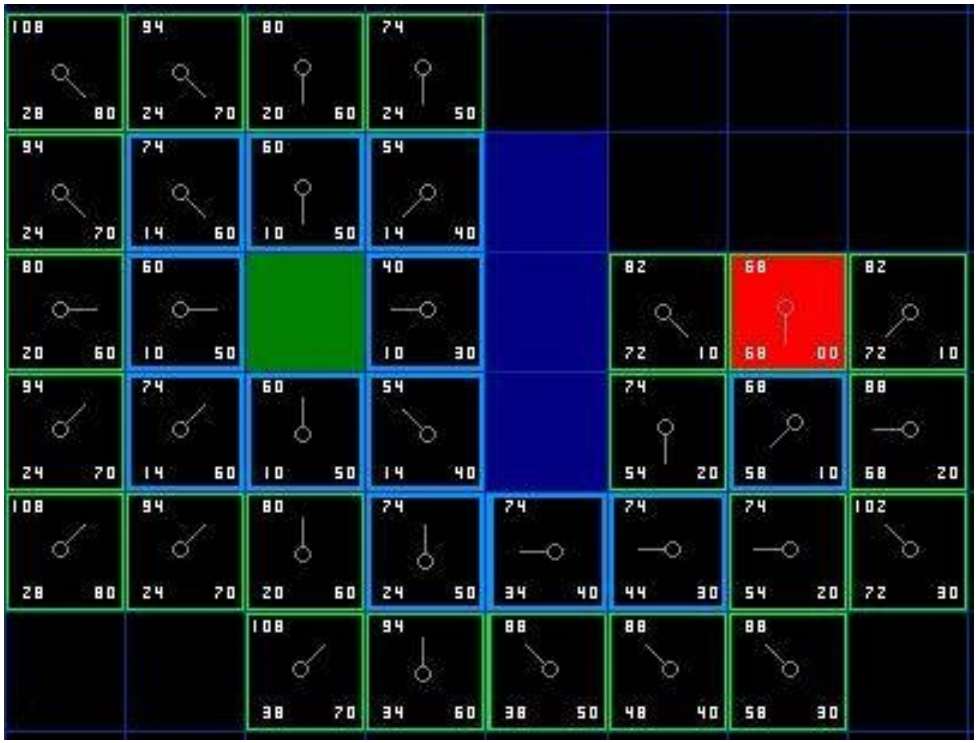
est. cost from this node to goal

$$f = g + h$$

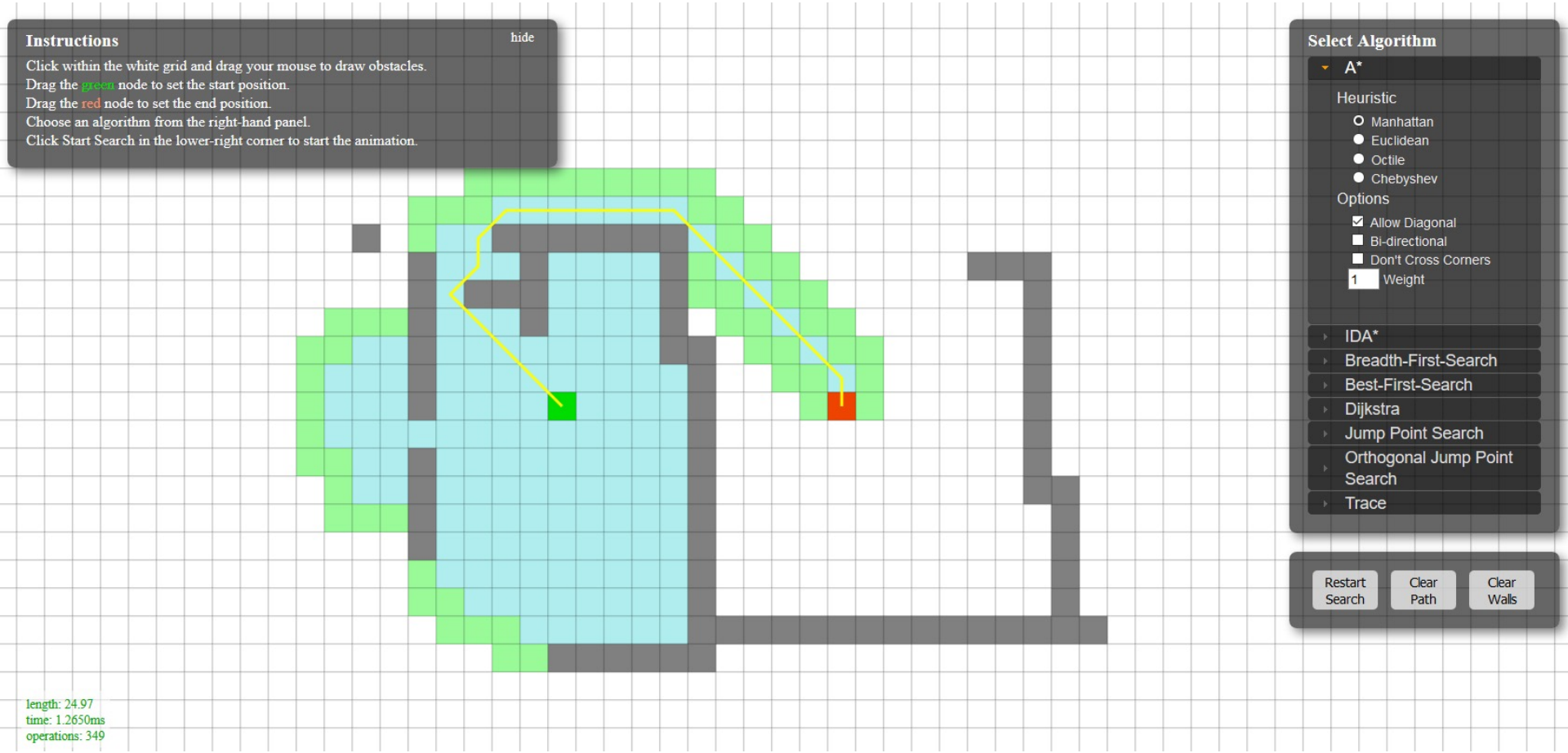
est. cost from start to goal via this node

images from: http://www.policyalmanac.org/games/aStarTutorial.htm


74
14 60
60
10 50
54
14 40
60
10 50
40 F
G H
10 30
74
14 60
60
10 50
54
14 40


108
28 80
94
24 70
80
20 60
74
24 50
94
24 70
74
14 60
60
10 50
54
14 40
80
20 60
60
10 50
40
10 30
82
72 10
68
68 00
82
72 10
94
24 70
74
14 60
60
10 50
54
14 40
74
54 20
68
58 10
88
68 20
108
28 80
94
24 70
80
20 60
74
24 50
74
34 40
74
44 30
74
54 20
102
72 30
108
38 70
94
34 60
88
38 50
88
48 40
88
58 30

https://qiao.github.io/PathFinding.js/visual/



(PathFinding.js.html)

Implementing A* Pathfinding..

- What data do we need? How might we structure the data?
  - Start loc, target loc
  - Nodes to map the game board (2D array of nodes)
  - Walkable/unwalkable map (i.e. our original 2Darray of booleans)
  - Open list (as linked list of nodes?)
  - Storage of final path (as a stack of nodes?)

- What are the initial conditions for this data?
  - Each wall node is unwalkable -> ‘closed’
  - All the rest are not open and not closed
  - Calculate f,g,h for the **starting node** and set to ‘open’

Implementing A* Pathfinding..

- What is the initial algorithmic step?
- What is the general algorithmic step?
  - Find open node with lowest f (call it X)
  - EXPAND: Look at its neighbours: any not closed and not open should become opened: calculate f,g,h and record parent position (i.e. position of X)
  - Close node X

- How will we know when we're finished?
  - If a neighbour is the target, we're done searching
  - If there are no open nodes, the maze is unsolvable

- How will we use what we found in order to have an AI-controlled '*badguy*' chase after a '*player*'?
  - Push target onto stack,
  - Push its parent onto stack
  - Push its parent onto stack
  - Etc.. Until we have pushed start node

Data for A*

- It makes sense to define a ‘node’ class, and to store nodes in specific kinds of data structures. I suggest:
  - a 2D array covering the whole game area (quick to find based on x,y)
  - a linked-list for the Open List (quick to add/remove members)
  - a stack storing the calculated path to follow (good for reversing order via LIFO)
- Of course, each node instance can happily exist in multiple data structures, since they're actually only storing pointers to it
- The nature of the A* algorithm means that we obtain our calculated path in the reverse order to how we need it
  - use a **2D array** to store all possible node cells during calculations, then when the target is found:
  - use a **stack** to store the path that a ‘badguy’ will follow, as this is a handy way to reverse the order of data
- The **linked list** is not strictly required, but since only a subset of all nodes will be Open at any given time, it's more efficient to store these in a separate data structure rather than have to search all nodes to find the best Open node to expand next

Stack

'Last in First Out' (LIFO)

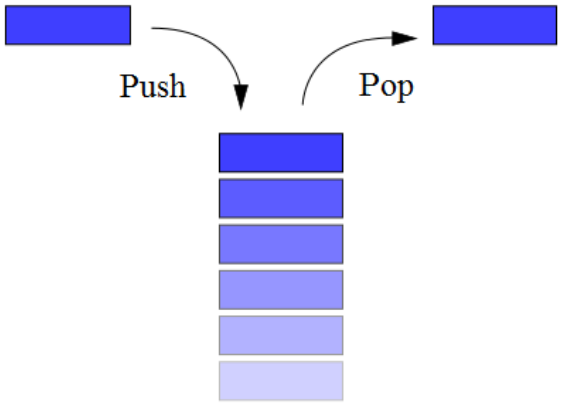
- In Java, use the Stack class:

```
import java.util.Stack;
```

Use the `push` and `pop` methods of this class

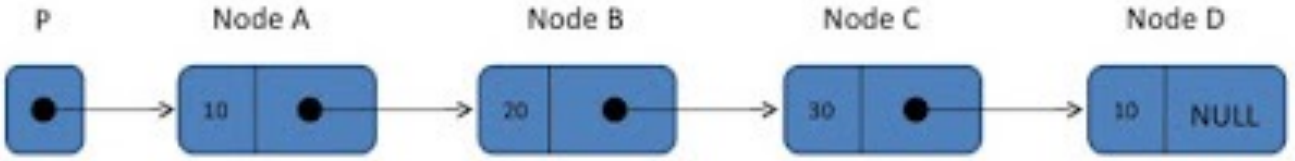
```
myStack.push(myObject);
myObject = (myClass)myStack.pop();
```

You can push any object you like onto a stack (e.g. our node object) and when popping it you must cast it to its correct data type



Linked List

- A list implemented by each item having a link to the next item.
- Head points to the first node.
- Last node points to NULL.



- Very efficient for insertion and deletion
- Can only be iterated sequentially (i.e. not random access)

```java
import java.util.*;
public class LinkedListDemo {

   public static void main(String args[]) {
      // create a linked list
      LinkedList ll = new LinkedList();

      // add elements to the linked list
      ll.add("F");
      ll.add("B");
      ll.add("D");
      ll.add("E");
      ll.add("C");
      ll.addLast("Z");
      ll.addFirst("A");
      ll.add(1, "A2");
      System.out.println("Original contents of ll: " + ll);

      // remove elements from the linked list
      ll.remove("F");
      ll.remove(2);
      System.out.println("Contents of ll after deletion: " + ll);

      // remove first and last elements
      ll.removeFirst();
      ll.removeLast();
      System.out.println("ll after deleting first and last: " + ll);

      // get and set a value
      Object val = ll.get(2);
      ll.set(2, (String) val + " Changed");
      System.out.println("ll after change: " + ll);
   }
}
```

This will produce the following result –

Output

```
Original contents of ll: [A, A2, F, B, D, E, C, Z]
Contents of ll after deletion: [A, A2, D, E, C, Z]
ll after deleting first and last: [A2, D, E, C]
ll after change: [A2, D, E Changed, C]
```

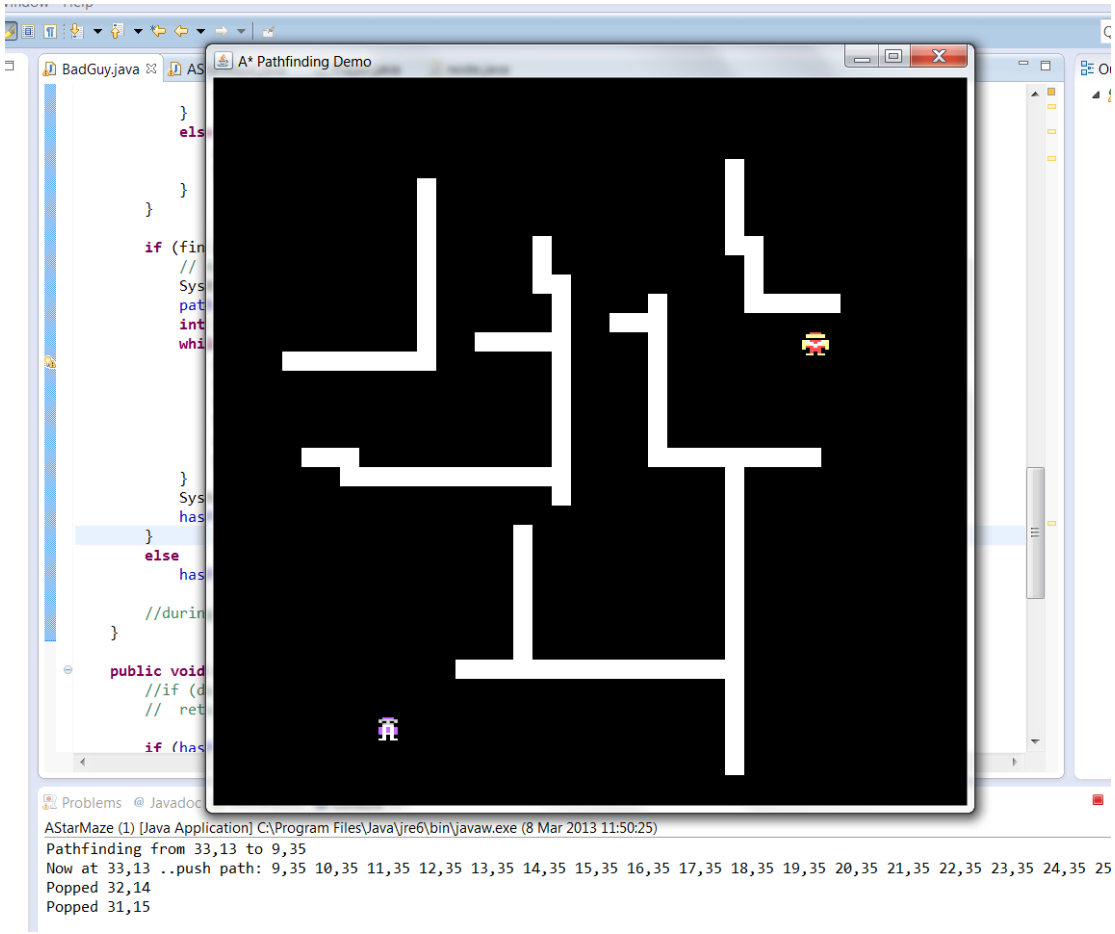
Iteration:

```
Iterator i = ll.iterator();
while (i.hasNext()) {
  string s = (String)i.next();
}
```

From: https://www.tutorialspoint.com/java/java_linkedlist_class.htm

Assignment

- Download base code for ‘badguy chases the player’ game
- This provides maze drawing, loading, saving
- It also moves the player with arrow keys, and badguy moves according to a dumb ‘straight line’ chase path – stops at walls
- Your goal is to implement A* pathfinding to make the badguy chase more effectively
- The A* path should be recalculated whenever the player moves or the maze is modified



Base code:

AStarDemoBaseCode.zip

(posted on Blackboard)

Debugging

- I'd recommend using **System.*out.print*** to debug the A* calculations and path following code