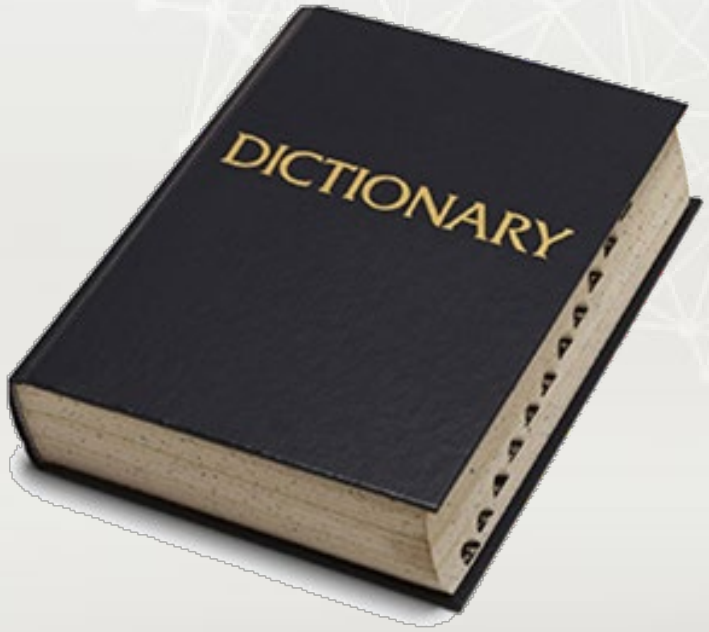
Algorithms 3 – Dictionaries & Graphs

by Hans Wichman

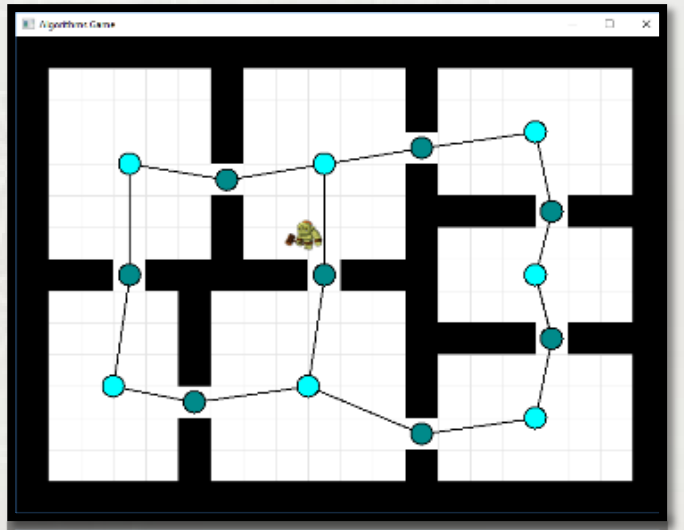
Algorithms 3

Dictionaries



What, why, when

Graphs



What, when, why +

Assignment 2 intro

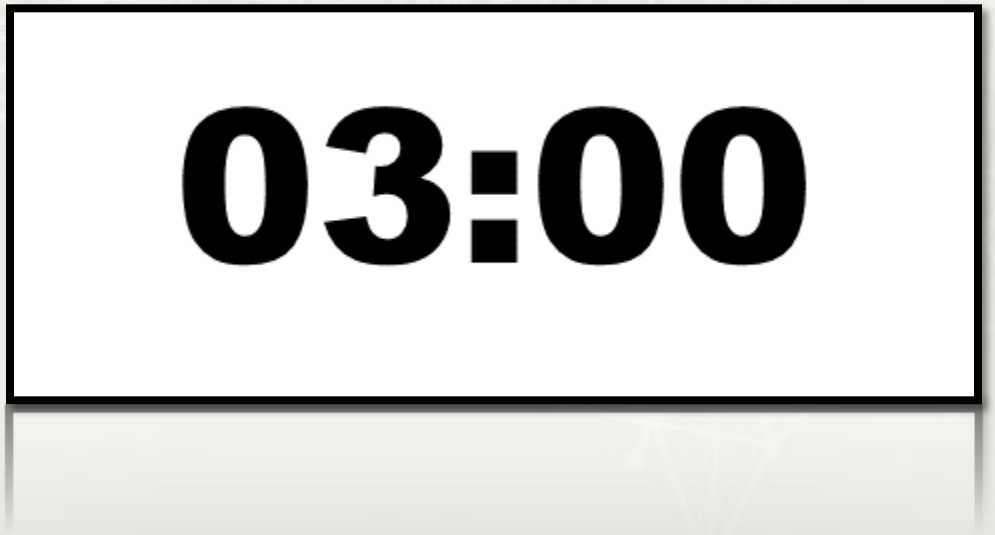
Coding ‘challenge’ – HighScoreManager

- You made an arcade game and now you want to keep track of (local) player HighScores
- On game over:
  - Players can enter their name
  - You will check for their HighScore and update/add it if necessary

- Challenge: *write down updateHighScore (string pPlayer, int pScore)*

Coding ‘challenge’ – HighScoreManager

- Challenge: *write down updateHighScore (string pPlayer, int pScore)*



Did you use Lists? For example, something like:

```
List<string>    playerNames = new List<string> ();
List<int>       playerScores = new List<int> ();

updateHighScore (string pName, int pScore) {
        int index = playerNames.IndexOf (pName);

        if (index == -1)
        {
                playerNames.Add (pName);
                playerScores.Add (pScore);
        }
        else if (playerScores[index] < pScore)
        {
                playerScores[index] = pScore;
        }
}
```

What are the ***downsides*** of using this approach?

- IndexOf is O(n)
- We have to keep the lists in sync ourselves (imagine having to present a sorted list of player highscores)

Did you use Player objects? E.g. something like:

```
List<Player> players = new ....;

updateHighScore (string pName, int pScore) {
        int index = -1;
        for (int i = 0; i < players.Count; ++i) {
                if (players[i].Name == pName) {
                        index = i;
                        break;
                }
        }
        if (index == etc ....
}
```

Downsides of this approach:
Using objects is more OO, but without 'hacks' we can't use IndexOf anymore:

- So we have all of the previous issues
- Plus actually more work,
  we are reinventing the wheel

Who used Dictionaries already?

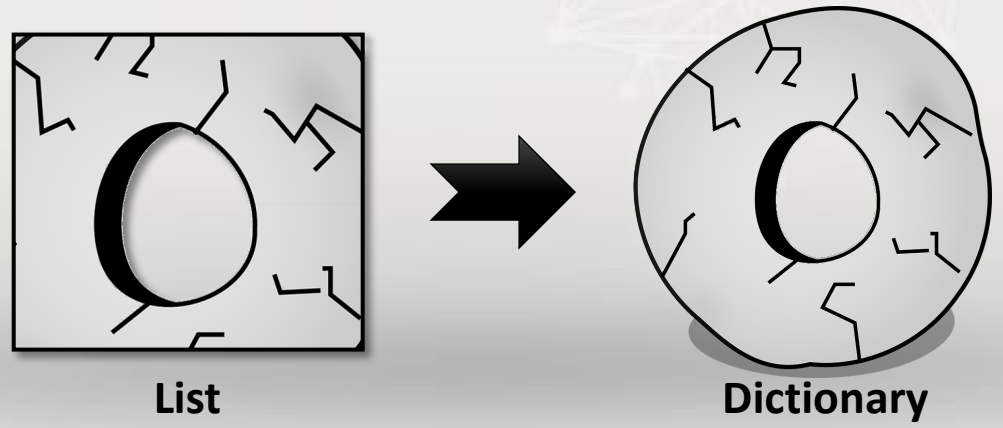
```
Dictionary<string, int> scores = new Dictionary<string, int>();

updateHighScore (string pName, int pScore) {
        if (!scores.ContainsKey(pName) || scores[pName] < pScore) {
                scores[pName] = pScore;
        }
}
```

Downsides of this approach:
None ☺

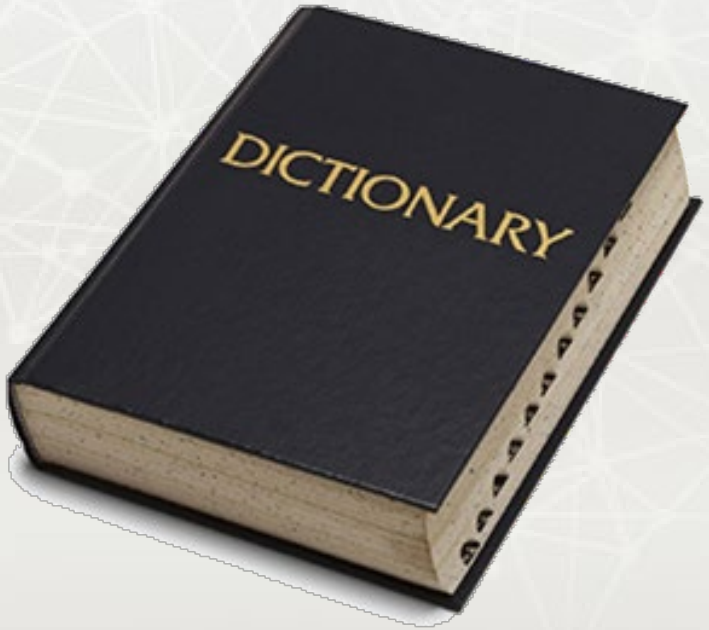
Short, sweet and to the point!

Moral of the story

- The problem we are trying to solve is a common one:
  - Relate a (possibly non integer) value to another value
  - Relate a (possibly non integer) value to another value ***quickly***
- We ***could*** use lists for that, but:
  - the list solutions we've just shown are all O(n),
  - we are basically reinventing the wheel this way, but not in a good way
  - we should use Dictionaries for this sort of problems instead



Dictionaries



The Dictionary<TKey, TValue> data structure

- Dictionary is a new (generic) data structure

- A **list** maps an ***integer*** key to a value of any chosen type:
  - List<string> players = new List<string>();
  - list[**0**] = "n00bZ3r";

- A **dictionary** maps a key ***of any chosen type*** to a value of any chosen type:
  - Dictionary <string, int> _playerScores = new Dictionary<string, int>();
  - _playerScores["**n00bZ3r**"] = 200;

- Also goes by the name of:
  - Map
  - HashMap
  - HashTable
  - LookupTable
  - Associative array (eg php)

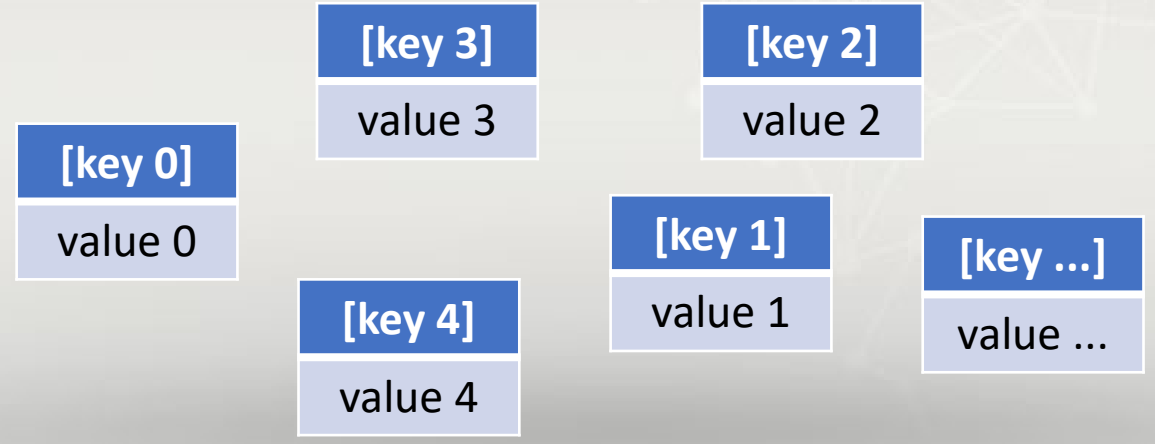
```php
<?php
$arr_course = array();
$arr_course["Course1"] = "PHP";
$arr_course["Course2"] = "MySQL";
$arr_course["Course3"] = "Java";


foreach($arr_course as $key_arr => $val_arr){
    echo $key_arr . " = " .$val_arr ."<br />";
}

?>
```

List vs Dictionary

List

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[...]
value 0	value 1	value 2	value 3	value 4	value 5	value 6	value 7	value 8	value ..

Dictionary



The Dictionary<TKey, TValue> data structure

"Represents an unordered collection of typed keys and values."

- Characteristics:
  - *Un*ordered collection
  - Maps *unique* keys to (possibly duplicate) *values*
  - Modifiable
  - Auto sizes
- Common operations:
  - Adding/updating a key/value pair
  - Removing a key/value pair
  - Checking whether a key exists
  - Clearing the whole dictionary

- MSDN Docs → Look up all the operations and compare them with Lists!

Some examples of these operations:

- Dictionary <string, int> _highScores = new Dictionary<string, int>();
- _highScores.Add ("n00bZ3r", 10); //adds and checks for duplicate
- _highScores["n00bZ3r"] = 10; //adds/overwrites on exist
- _highScores.Clear();
- _highScores.ContainsKey("n00bZ3r");
- _highScores.ContainsValue(10);
- _highScores.Remove ("n00bZ3r");
- _highScore.TryGetValue ("n00bZ3r", out score);

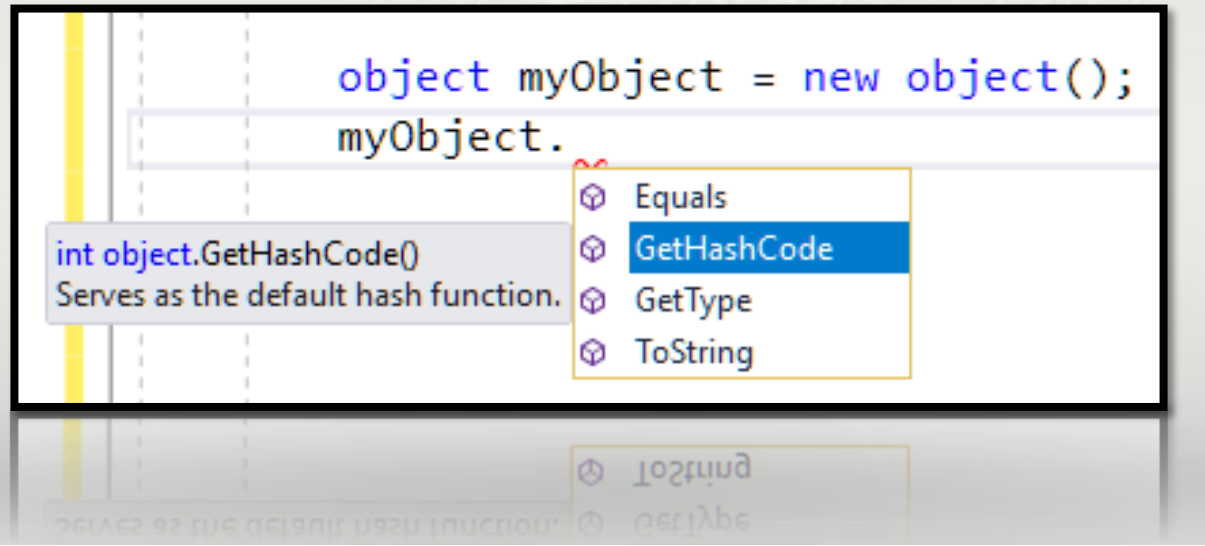
Things to note in the documentation:

- List.Add → O(1) / O(n) operation
- List.Contains → O(n) operation
- Dictionary.Add → O(1) / O(n) operation
- Dictionary.ContainsKey/Get → O(**1**)
- Dictionary.ContainsValue → O(n) operation

- How did 'they' accomplish the mapping of non integer keys to values?
- How did 'they' accomplish this mapping in O(1) time complexity?
- Why does this matter to us / why is this cool?

How does a Dictionary work internally?

- Last week we discussed lists:
  - Lists wrap an array, and arrays maps integer indices to values
- How do Dictionaries map (possibly non integer) keys to values?
- Through the use of HashCodes:



MSDN HashCode documentation

Object.GetHashCode Method

Namespace: System

Assemblies: System.Runtime.dll, mscorlib.dll, netstandard.dll

Serves as the default hash function.

C#

```
public virtual int GetHashCode ();
```

Returns

Int32

A hash code for the current object.

So what is a HashCode?

- Luckily, if we scroll down a bit...

A hash code is a numeric value that is used to insert and identify an object in a hash-based collection such as the Dictionary<TKey,TValue> class, the Hashtable class, or a type derived from the DictionaryBase class. The GetHashCode method provides this hash code for algorithms that need quick checks of object equality.

> **Note**
>
> For information about how hash codes are used in hash tables and for some additional hash code algorithms, see the **Hash Function** entry in Wikipedia.

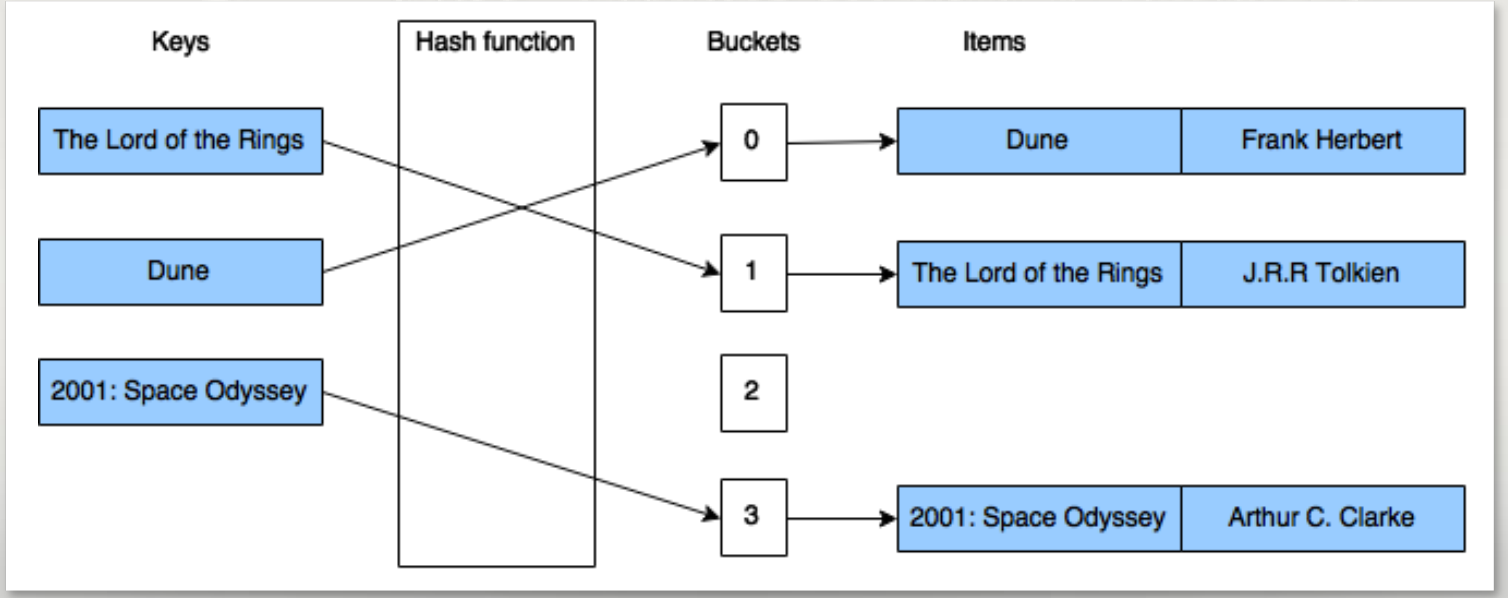
Two objects that are equal return hash codes that are equal. However, the reverse is not true: equal hash codes

etc etc etc

The basic idea is this...

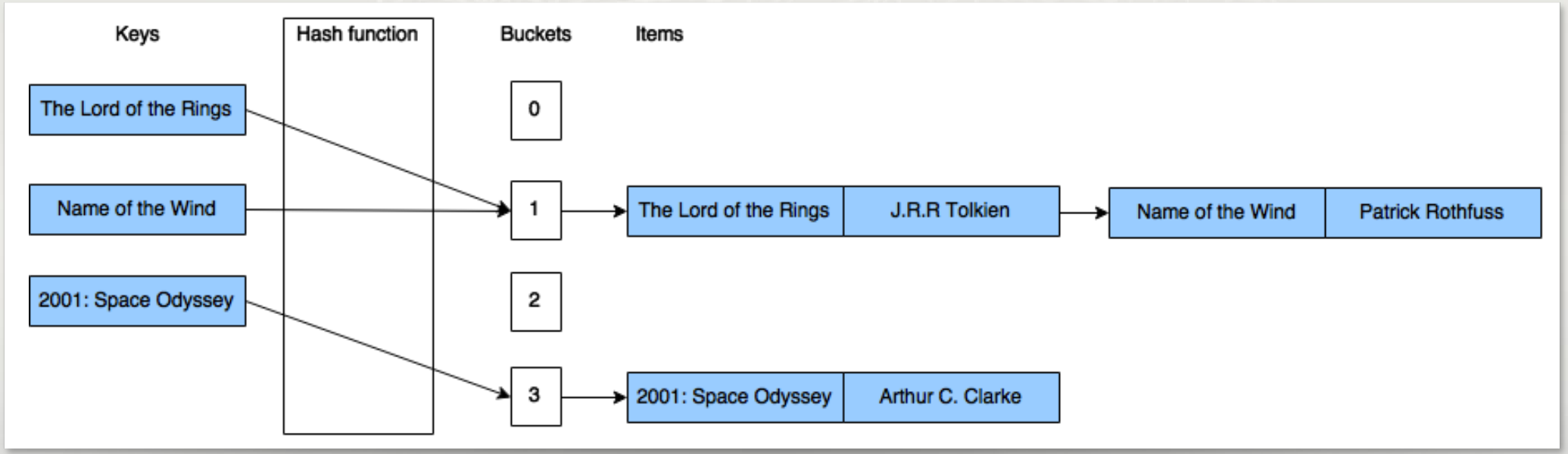
- Each object can return its HashCode
- A HashCode is a ***non unique*** integer which can be used as index in a list of 'buckets', for example in pseudo:

bucketIndex = object.GetHashCode() % bucketList.Count;



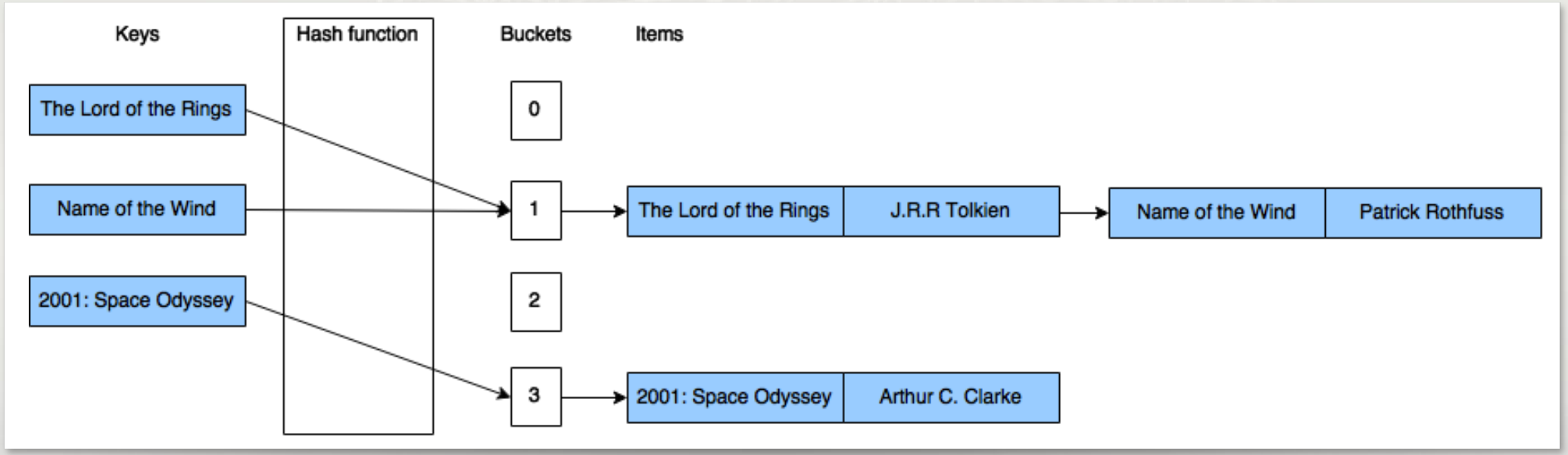
HashCode Collisions

- HashCodes are ***non unique*** integers, so 'collisions' might occur:
  - the buckets store linked entries, consisting of key **& value** pairs
    1. we search for the correct bucket using the key
    2. then we go through all entries in that bucket, looking for the matching key



HashCode Collisions

- Best case: every bucket only contains 1 key – value pair
- Worst case: all entries are in one bucket
  (by implementing a ***very bad*** GetHashCode)



Resizing

- Dictionaries have a load factor → ratio between buckets and entries
- To avoid too many collisions, we need enough buckets
- Dictionary uses an internal calculation to determine when a resize should happen
- But ***if** a resize happens:*
  - internal structure size is doubled (same as with list)
  - every key/value pair has to be re-inserted
- Self-check, can you now explain these values?
  - Dictionary.Add → O(1) / O(n) operation
  - Dictionary.ContainsKey → O(**1**)

List.Contains O(n) vs Dictionary.ContainsKey O(1) performance?

- List.Contains is O(n)
- Dictionary.ContainsKey is O(1)
  (O(n) worst case if you decide to mess up GetHashCode)

- Which is faster, List.Contains or Dictionary.ContainsKey ?
  - O(n)/O(1) says nothing about the actual performance,
    only about how the performance scales.
  - That said for any substantial collection, a dictionary is generally faster

Equals vs GetHashCode

- If you override the Equals method, you ***must override*** *GetHashCode*
  - *Objects that are equal, must return the same HashCode*
  - *Objects with the same HashCode, do not have to be equal*

- *Why?*
  - If you don't override GetHashCode for objects A & B for which A.Equals(B), A & B might end up in different buckets.
  - If two objects A & B are in *different* buckets, they can *never be equal according* to the Dictionary.
  - If two objects A & B are *in the same* bucket, they *can* be equal, but *don't* have to be.

(Horrible) combinations are also possible

- Dictionary <string, Dictionary<string, int>> slightlyUnreadable1;
- Dictionary <string, List<int>> slightlyBetter2;
- Dictionary <string, Dictionary<string, List<int>>> codingHorror3;
- List<Dictionary <string, Dictionary<string, int>>> eyesAreBleeding4;

- Constructions like this would love a little bit more documentation ☺
- Scrap that: constructions like that would like to not be written at all ☺

LIST vs DICTIONARY

List or dictionary?

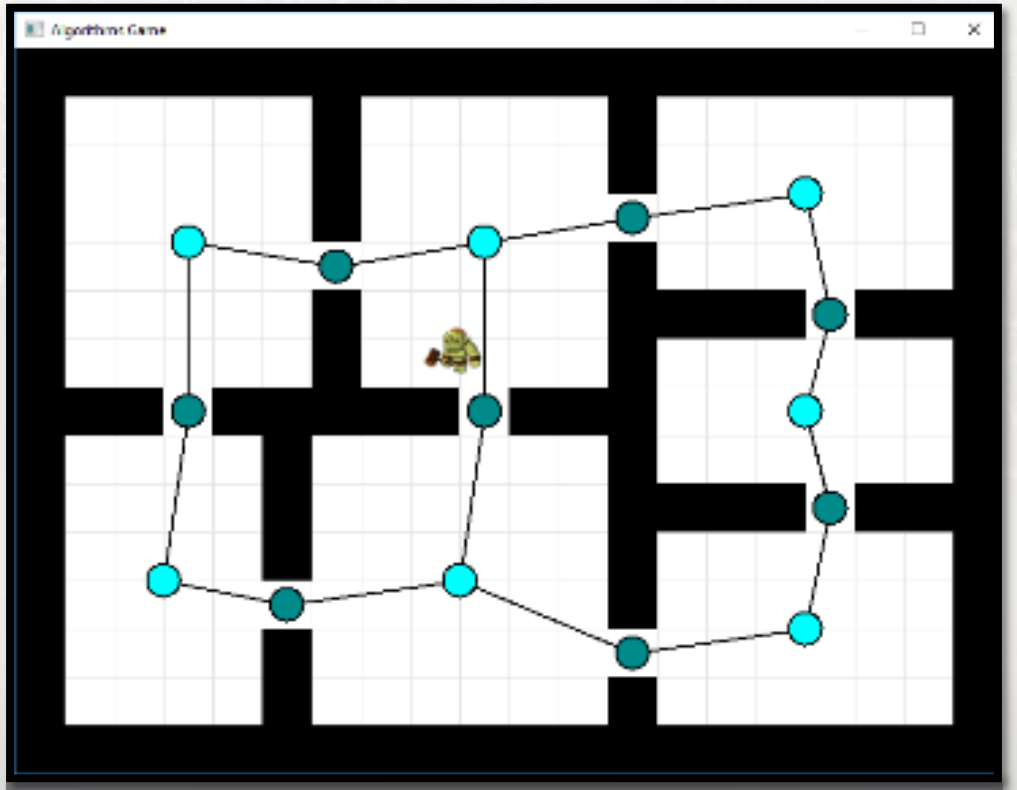
- What should we use for these 'problems'?:
  - Calculate how many unique cards there are in your the deck
  - Create a mana curve for your random deck
    (A mana curve tells you how many 0,1,2,3,4,etc mana cards you have)
  - Map file extension to application
  - Counting all cards in your HearthStone deck per minion type
  - Mapping current level blocks to possible next level blocks
  - Finding players through ip addresses in online multiplayer games
  - Map some integers to some object names
    - 1, 2, 3, 4, 5, etc
    - 1, 1000, 1000000, etc

Sorting dictionaries

- By key
- By value

- See the 001_dictionary_example in the downloads

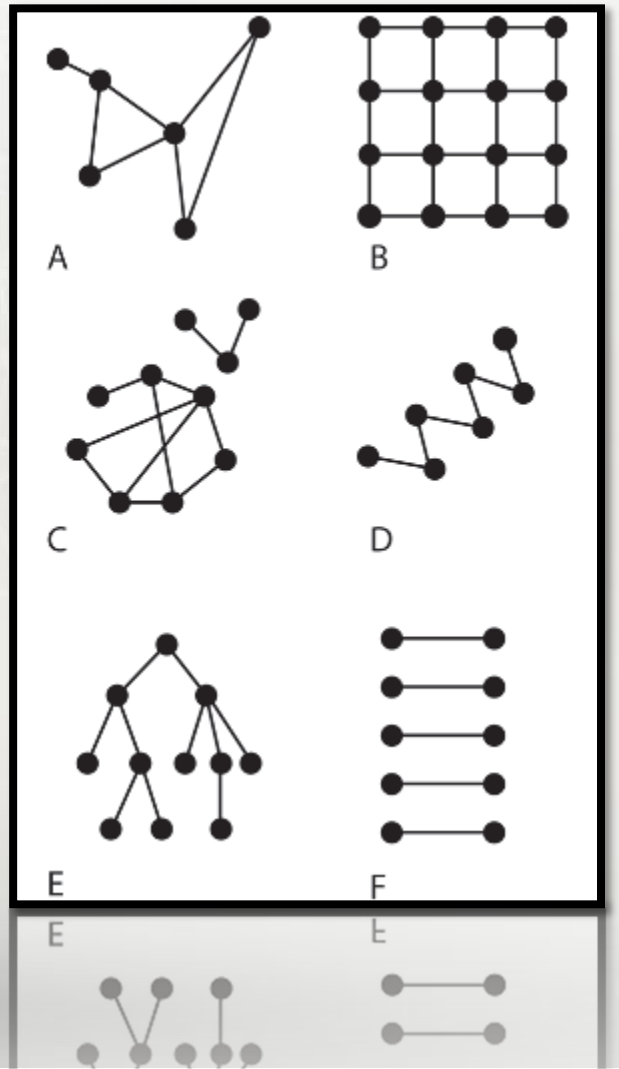
Graphs



What is a Graph?

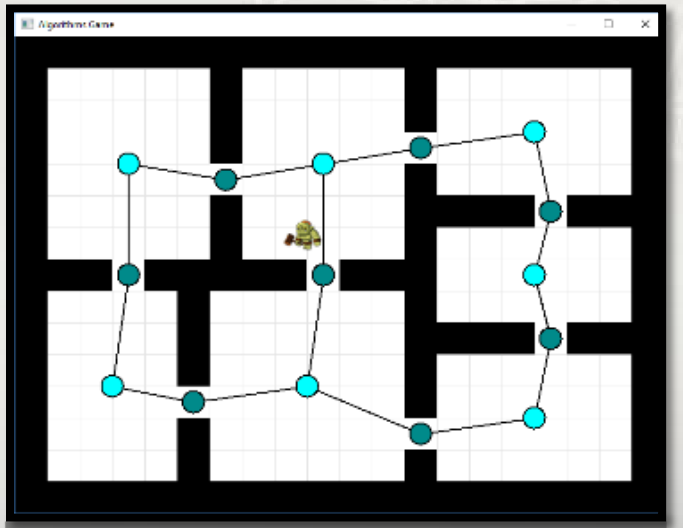
- A (data) structure consisting of connected nodes
- The simplest way to *display* a graph is using points and lines
- Using these nodes and the connections between them, we can use graphs to describe all sorts of *related* items.

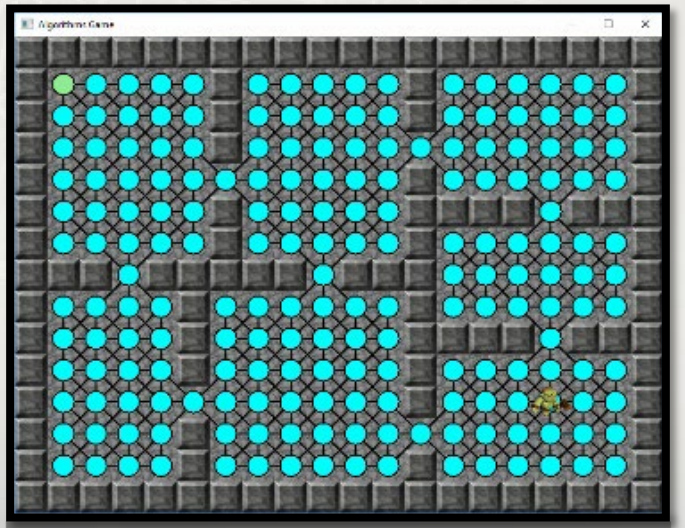
- Let's look at some examples!



Example 1 – A navigation Graph

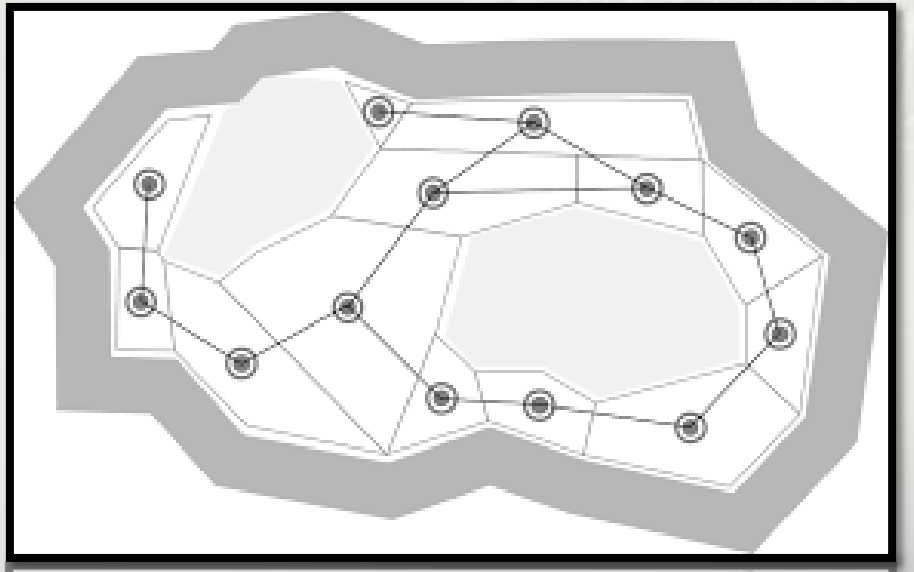
- Each node is a location
- Each connection indicates how to get to a location
- Level of detail is completely implementation dependent:





Side note on navigation in modern games

- Modern games often use a combination of navigation graphs for point-of-interest route planning & navigation meshes for actual path finding, but both have the concepts of nodes and connections at heart:

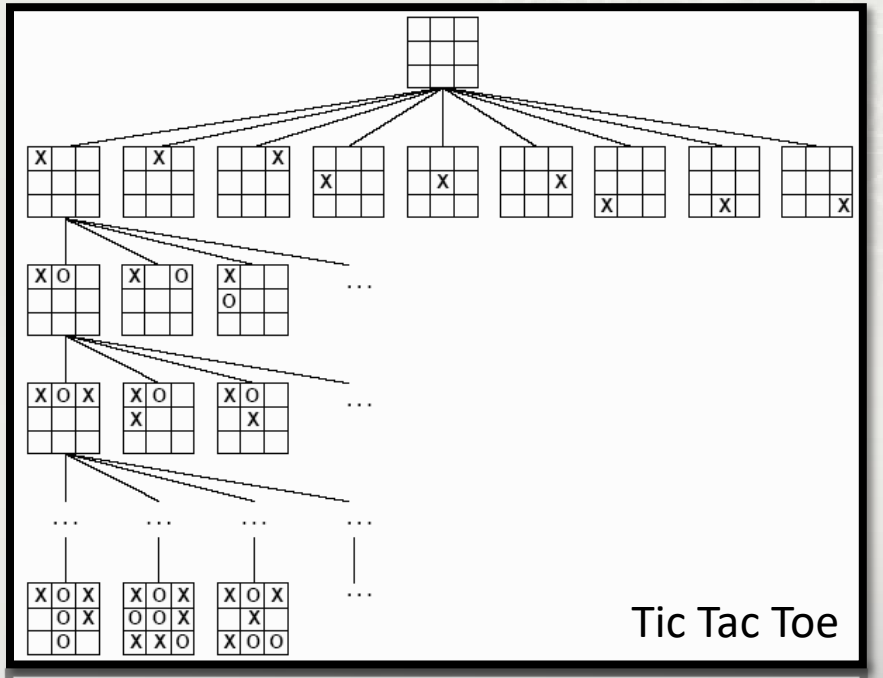


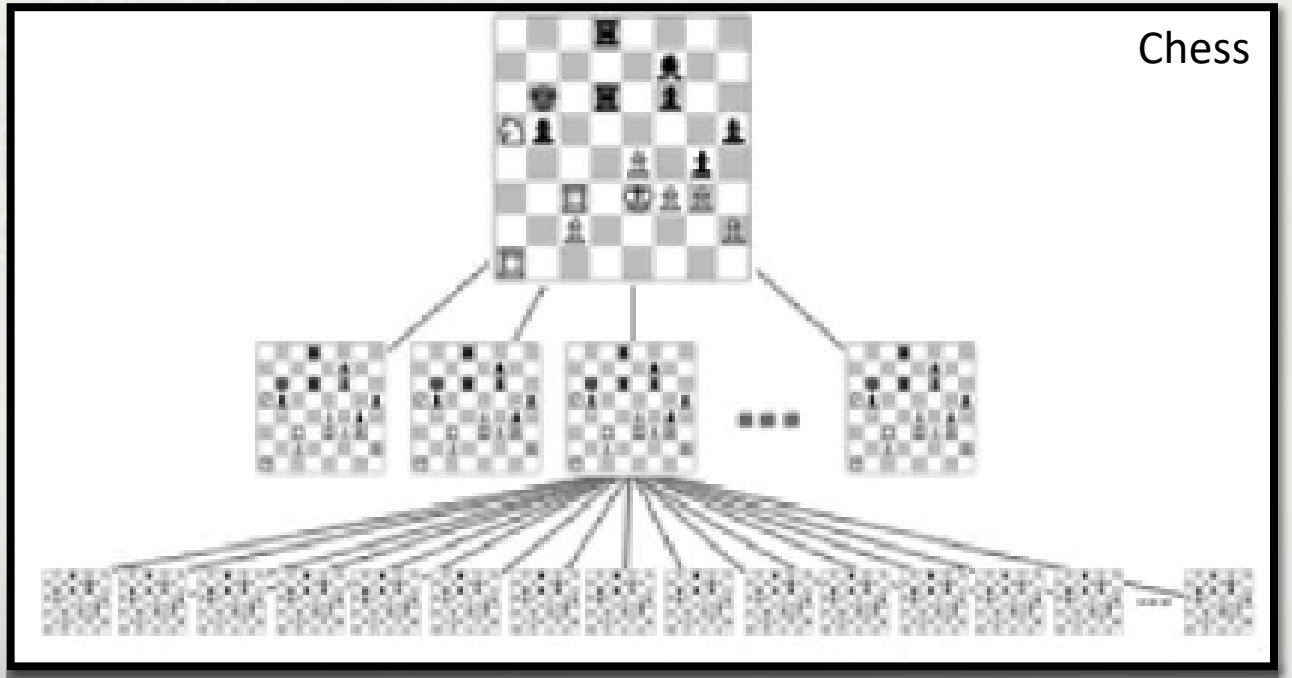
Example 2 – Talent/Skill trees

- Each node is a skill or talent
- Each connection describes the prerequisites for the skill or talent

Example 3 – Game state graph

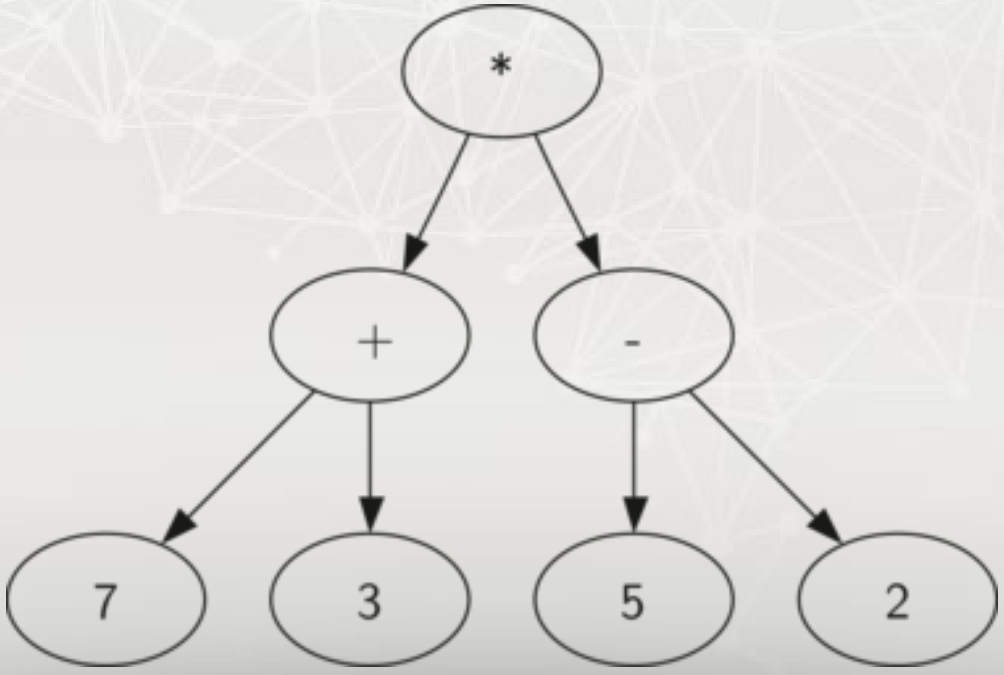
- Each node is a possible game state
- Each connection a possible move to reach that game state





Example 4 – Expression Parse trees

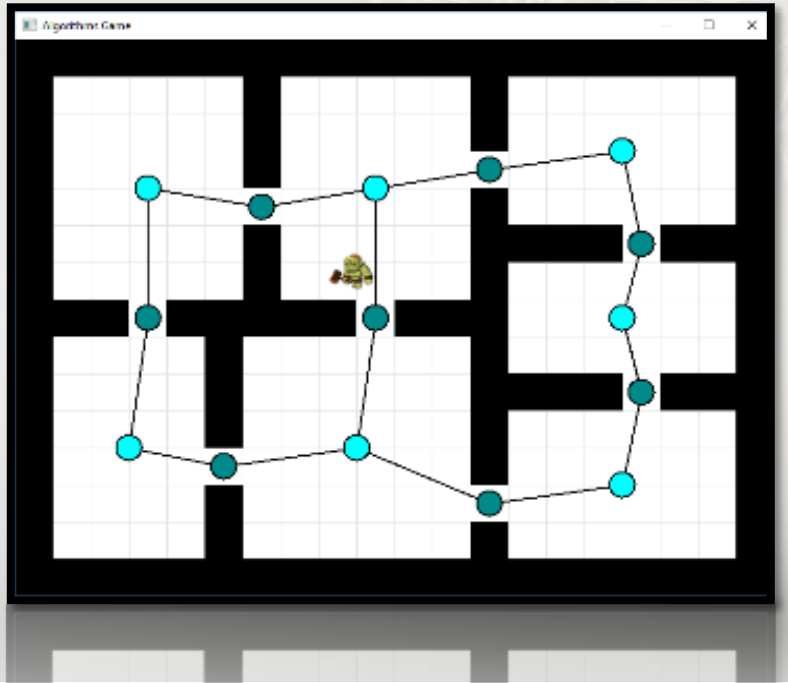
- Each node is a sub-expression (operator or number)
- Each connection a link to the (sub)-sub-expression

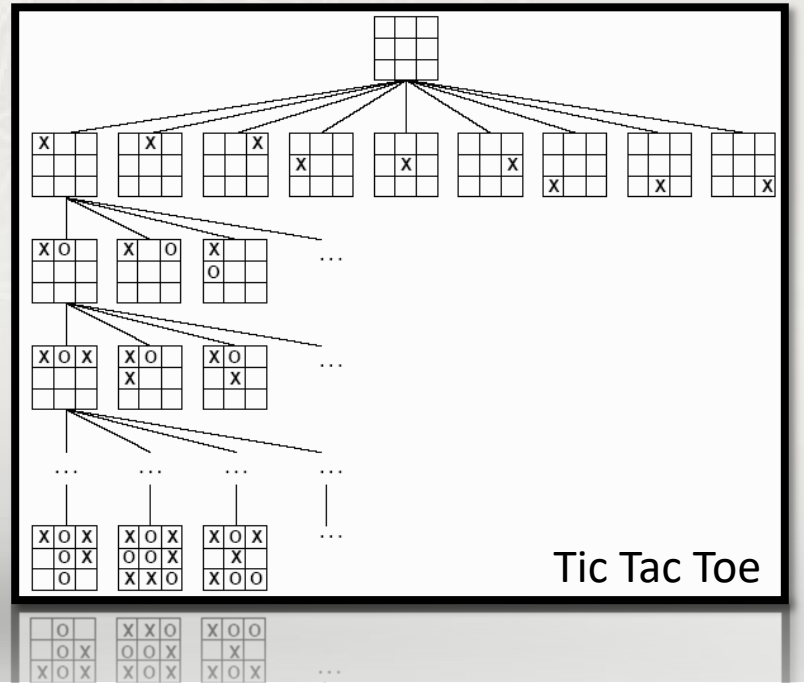


Graph types

Graph types - Explicit vs Implicit

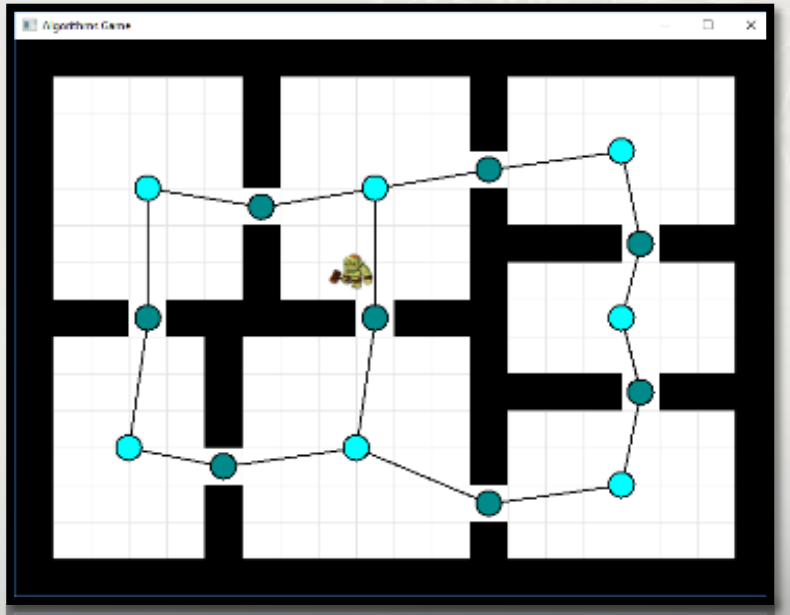
- Explicit: The board state is the graph, fully accessible in memory
- Implicit: The board state is just a node in the graph, reaching other nodes is done by cloning the board state and performing a 'move' on it

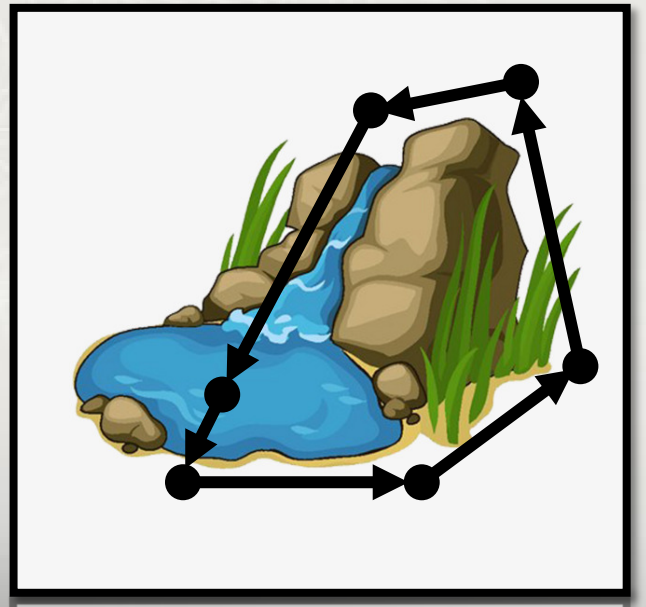




Graph types – Directed vs Undirected

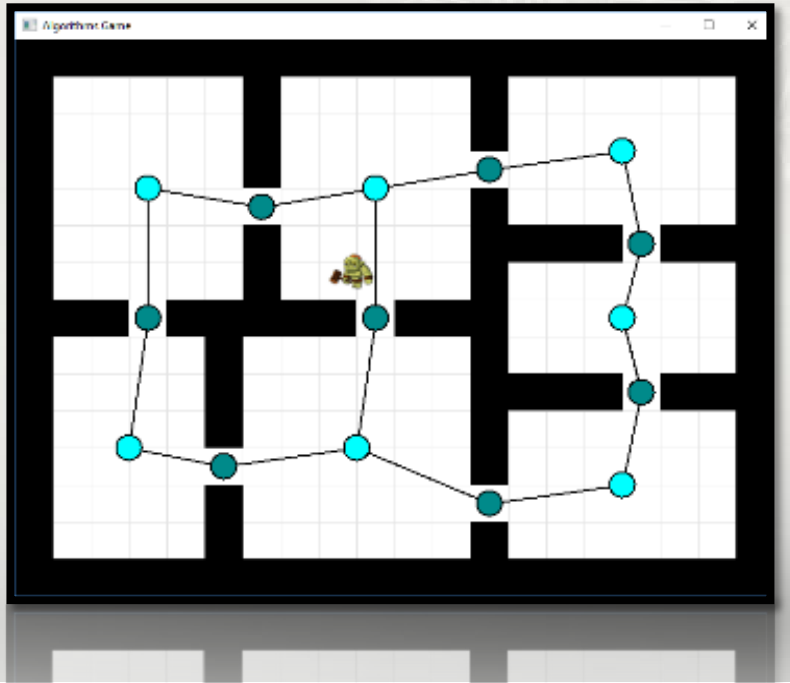
- Undirected: we can traverse connections in either direction
- Directed: we can traverse connections in a single direction
- A single graph could contain both types of connections of course

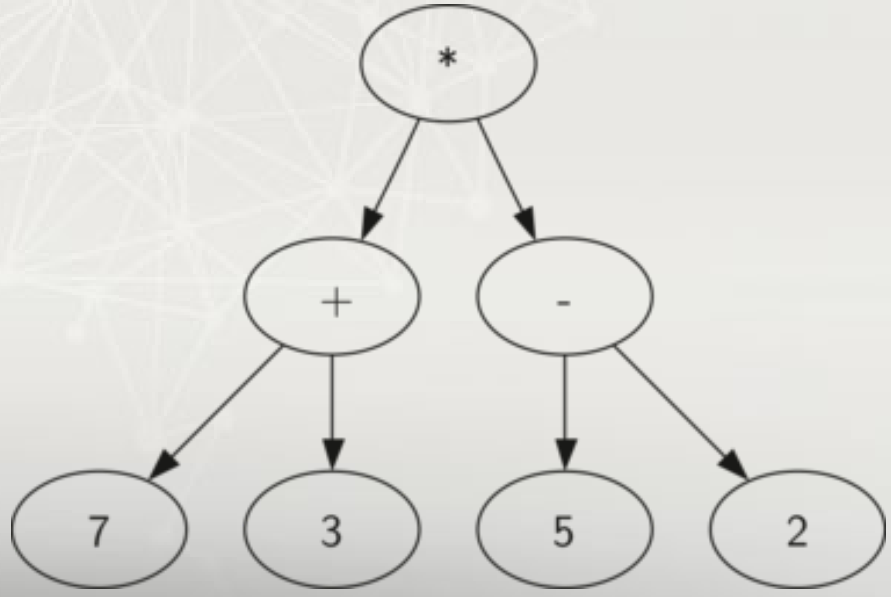




Graph types – Cyclic vs Acyclic

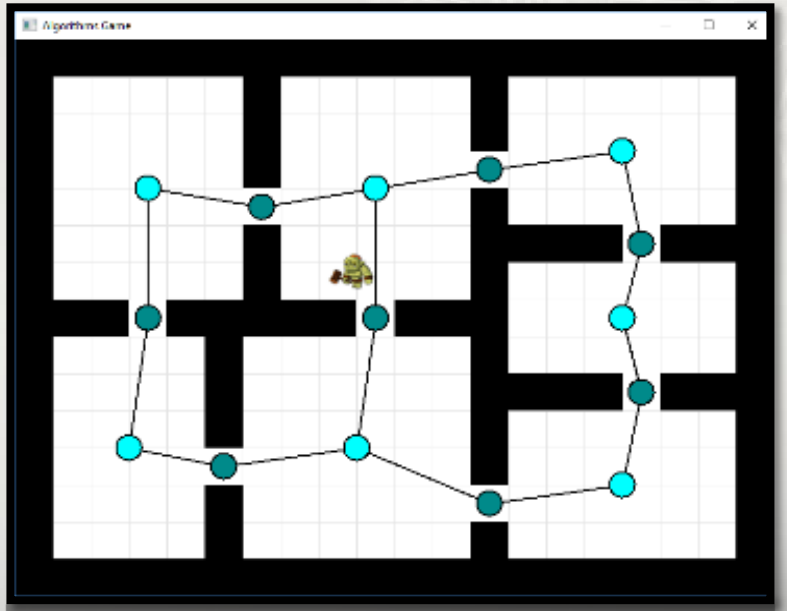
- Cyclic: There are cycles in the graph
- Acyclic: There are no cycles in the graph

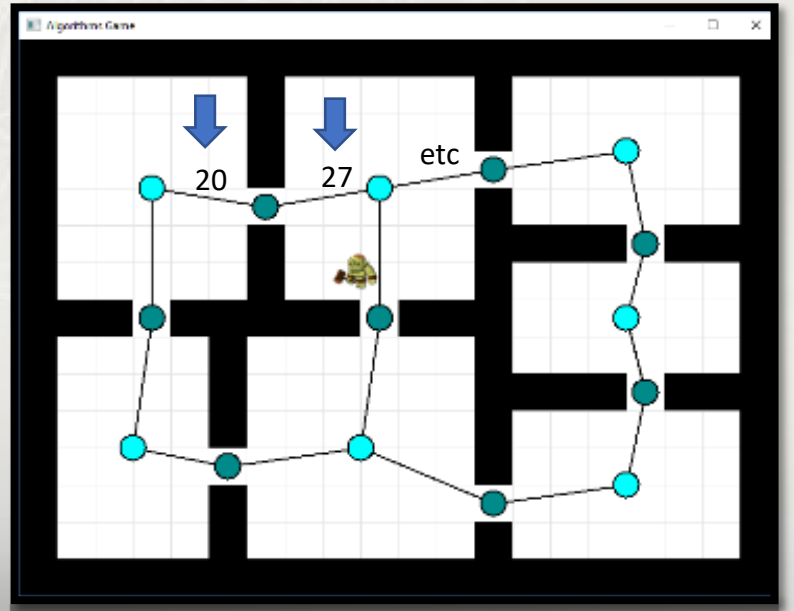




Graph types – Weighted vs Unweighted

- Unweighted: 'travelling' to a node costs 1 / cost is ignored
- Weighted: 'travelling' to a node costs x,
  where x is based on distance/terraintype or any other chosen value





Graphs as problem spaces

Graphs as a description of problem spaces

- When we look at graphs from a little bit more abstract perspective, we can also say that graphs can be used to describe problem spaces consisting of states and state transitions.

- What do we mean with problem space?
  - A description of a problem we are trying to solve
  - The relevant data / states within that problem (the nodes)
  - The transitions between those states (the connections)

Navigation graph problem space

- ***What problem are we trying to solve?***

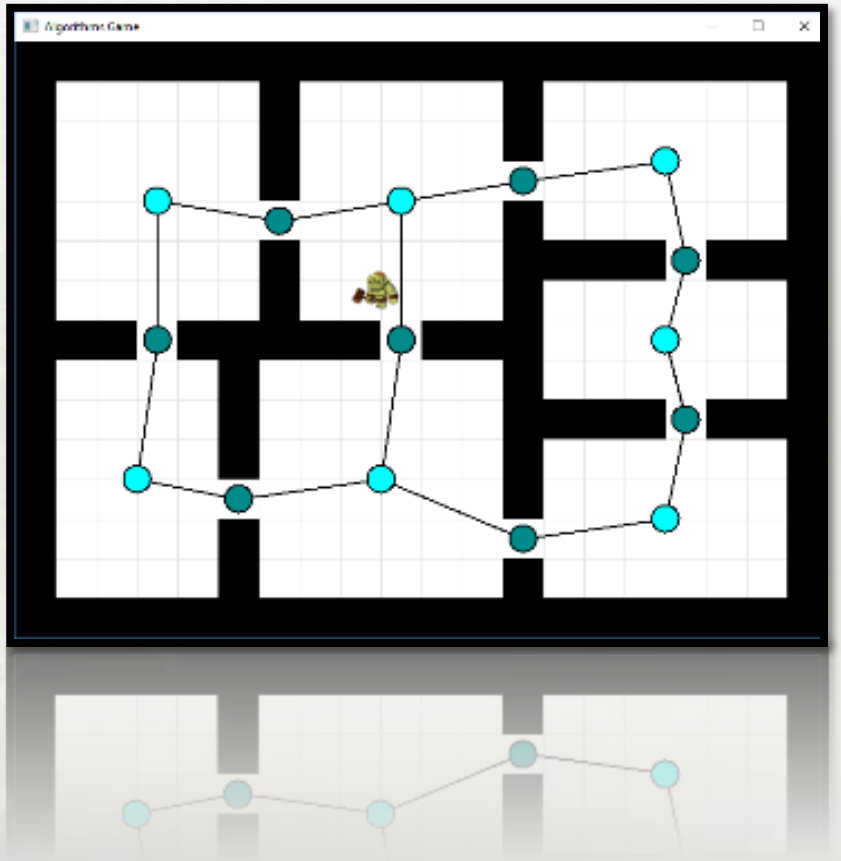
  Navigating from one place to another

- ***What are the nodes/states in the graph?***

  Locations

- ***What are the transitions?***

  Moving from one location to another,
  a link indicates that a move is possible.



Tic tac toe graph problem space

- ***What problem are we trying to solve?***

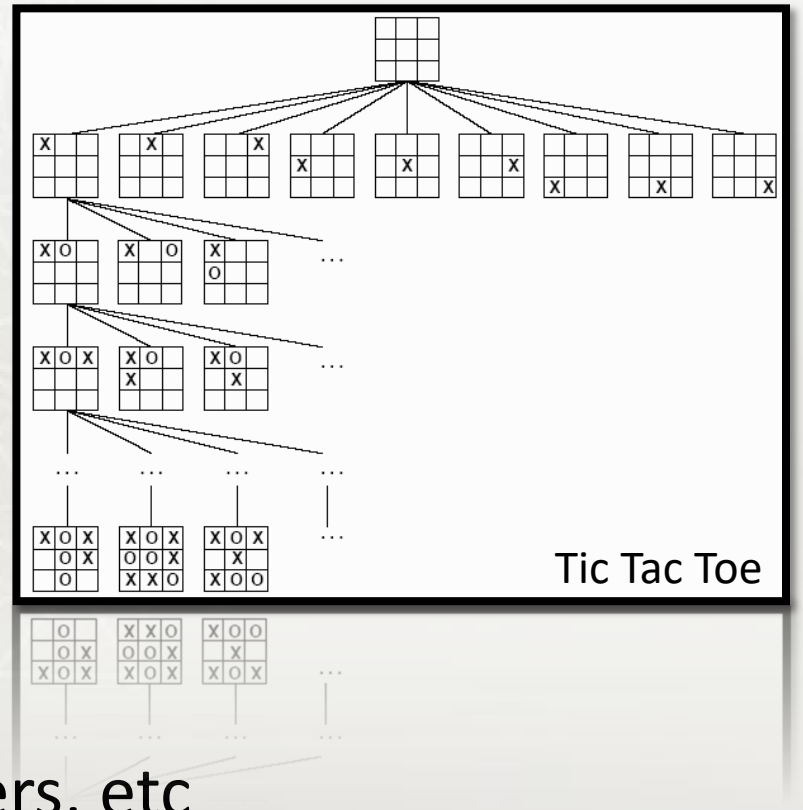
  Playing a game of Tic Tac Toe (trying to win!)

- ***What are the nodes/states in the graph?***

  Possible board states

- ***What are the transitions?***

  Valid moves from one board state to another



- Similar examples (of varying complexity):
  HearthStone, Chess, Towers of Hanoi, Checkers, etc

Your turn

- A farmer has a goat, wolf and a cabbage
- He needs to get them across the river, ***but***
  - he can't leave the wolf with the goat, nor the goat with the cabbage
  - he can only bring one 'thing' across with him in the boat
- In which order, should he bring what/who across?

- Answer these questions:
  - What problem are we trying to solve?
  - What do the nodes represent?
  - What do the connections represent?
  - Can you solve the problem?

Your turn

03:00

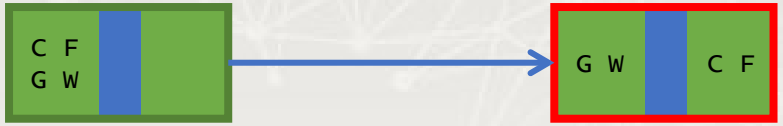
- A farmer has a goat, wolf and a cabbage
- He needs to get them across the river, ***but***
  - he can't leave the wolf with the goat, nor the goat with the cabbage
  - he can only bring one 'thing' across with him in the boat
- In which order, should he bring what/who across?

- Answer these questions:
  - What problem are we trying to solve?
  - What do the nodes represent?
  - What do the connections represent?
  - Can you solve the problem?

The Wolf, Goat, Cabbage and the Farmer

- What defines a state? Where everyone is:

- What is a transition? Moving the farmer and at max 1 'thing':



- How many states are there?

The Wolf, Goat, Cabbage and the Farmer states

- We have 4 objects: wolf, goat, cabbage, farmer
- Each object can be:
  - left or right, across or not across, true or false, 1 or 0 → different ways to say the same thing
- Whichever you choose: 2 * 2 * 2 * 2 possibilities → 16 possible states
- As a helper method you could list all binary numbers from 0 – 15 first:

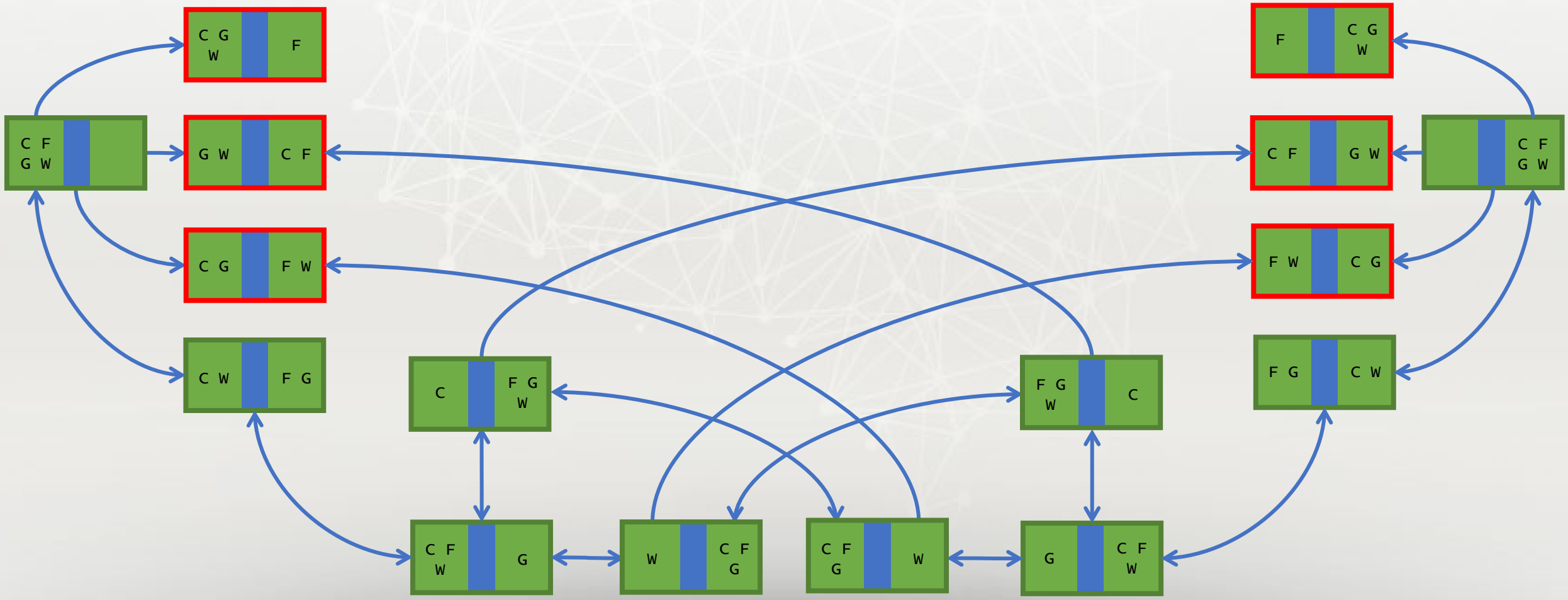
Binary number system

Base 10	Base 2
00	0000
01	0001
02	0010
03	0011
04	0100
05	0101
06	0110
07	0111
08	1000
09	1001
10	1010

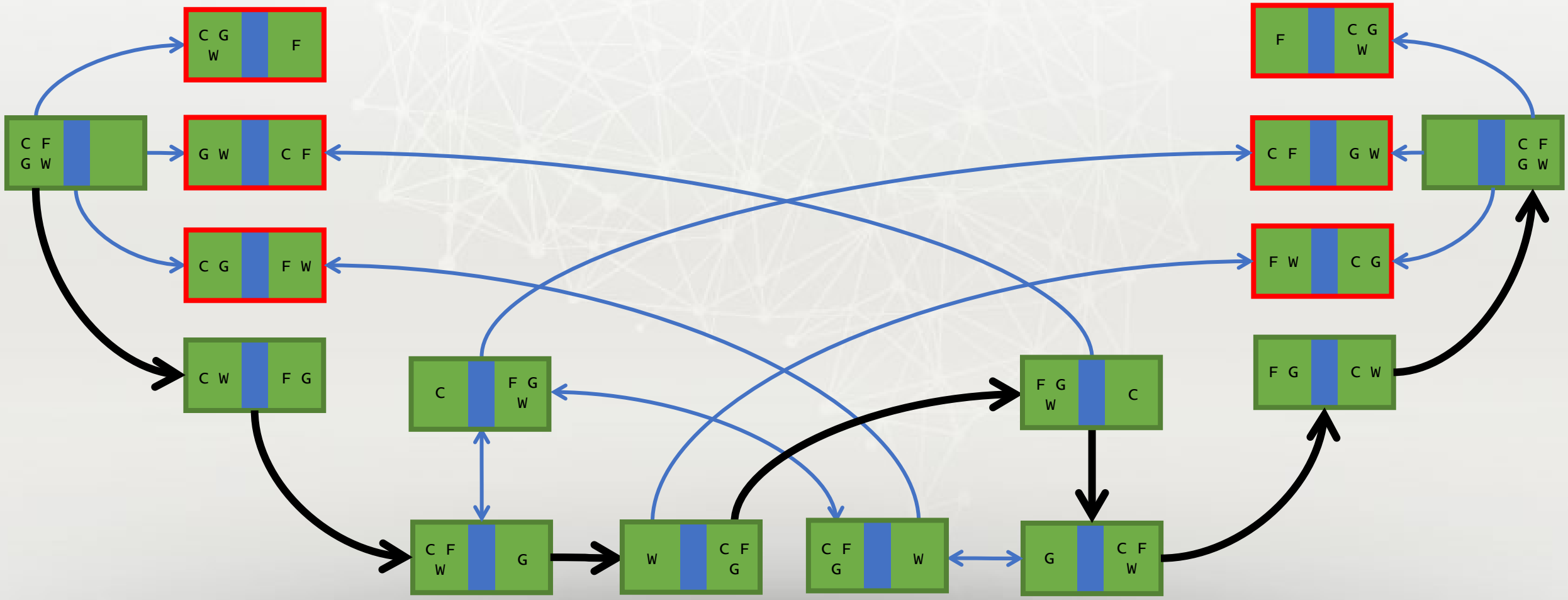
The Wolf, Goat, Cabbage and the Farmer states

Getting CFGW across the river...		
0	0000	CFGW -
1	0001	CFG - W
2	0010	CFW - G
3	0011	CF - GW
4	0100	CGW - F
5	0101	CG - FW
6	0110	CW - FG
7	0111	C - FGW
8	1000	FGW- C
9	1001	FG - CW
10	1010	FW - CG
11	1011	F - CGW
12	1100	GW - CF
13	1101	G - CFW
14	1110	W - CFG
15	1111	- CFGW

The Wolf, Goat, Cabbage and the Farmer graph



A possible solution



The hard part in describing a problem space

- What are the states and how do I represent them?
- What are the connections and how do I represent them?
- How do I translate all of this to code? (Continued in lecture 5 or 6)

- Luckily assignment 2 & 3 are way less complicated:
  - Each node is a position, each connection is a 'road'

Assignment 2

Create a Graph and navigate it (randomly)

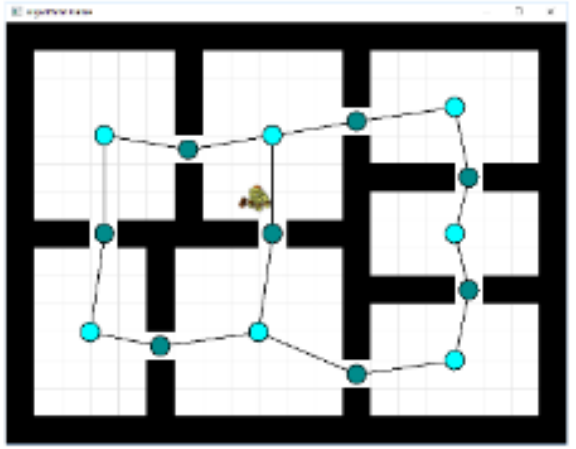
Assignment 2: Graphs, tiles & agents

In this second assignment you have to convert your Dungeon into graphs and tiles, learning how intermediate forms of data might help you solve a more difficult problem. In addition, you will practice general list manipulation a bit more by moving Morc-da-Orc through its new lair.

Continue in your code from assignment 1. Again, a walkthrough for new parts of the code will be provided during the lectures, demonstrating all the visualization options you have to (/can) use. This code only has to work with the 'sufficient' requirements from assignment 1.

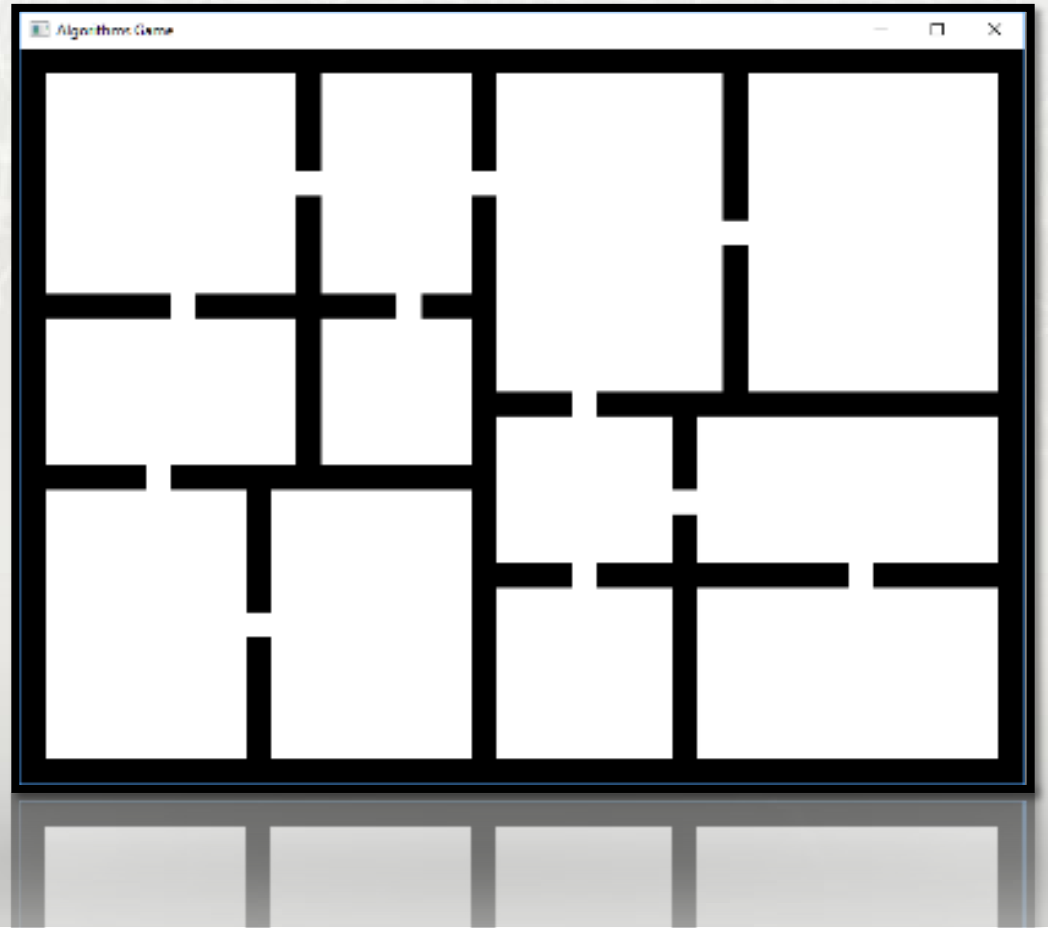
2.1 'Sufficient' requirements:

- Your graph replicates the dungeon structure with a node for every room and door with connections between them.
- Make sure that Morc queues clicked nodes and visits them in order. Morc is not allowed to go 'off graph' anymore, so make sure you ignore the clicked nodes that would cause this.



Paper first!

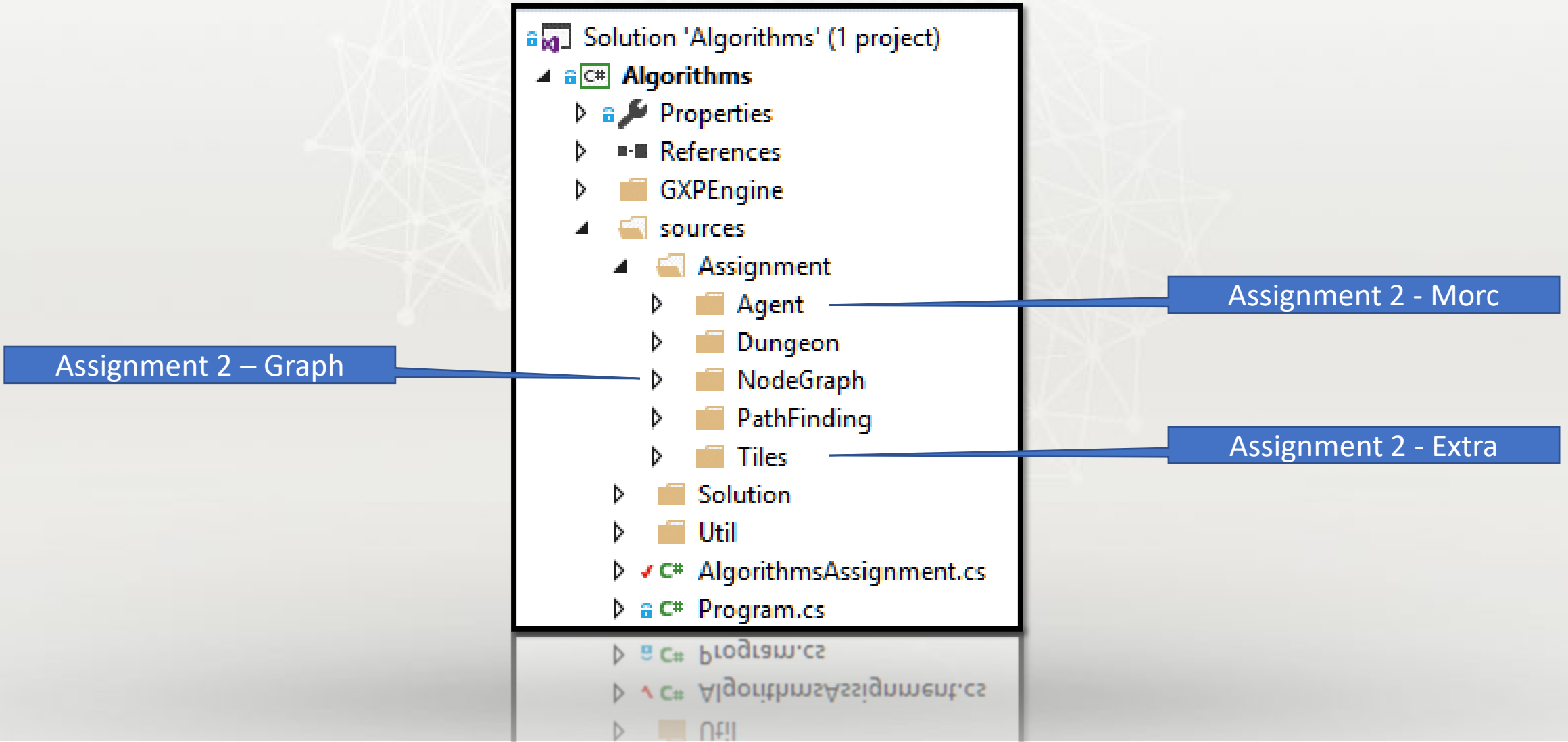
If you can't draw/describe it, you can't program it...



Part 1 – Implementing the Node Graph

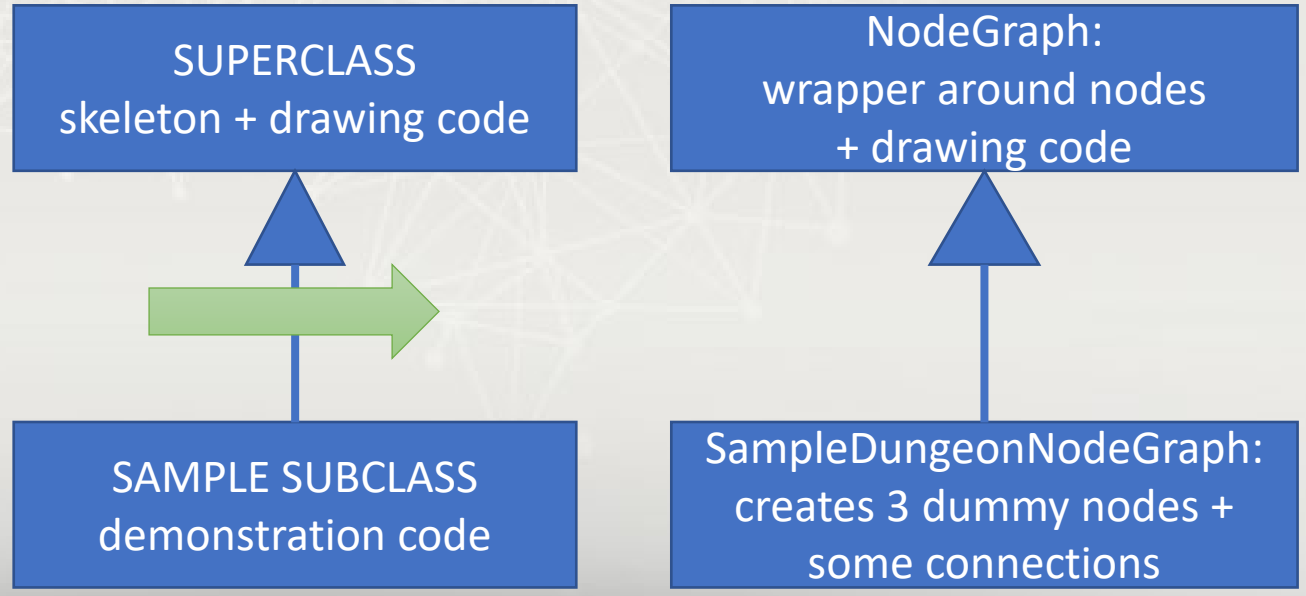
```
//////////////////////////////////////////////////////////////////////////////////
/// ASSIGNMENT 2 : GRAPHS, AGENTS & TILES
///
/// SKIP THIS BLOCK UNTIL YOU'VE FINISHED ASSIGNMENT 1 AND ASKED FOR TEACHER FEEDBACK !

//////////////////////////////////////////////////////////////
//Assignment 2.1 Sufficient (Mandatory) High Level NodeGraph
//
//TODO: Study assignment 2.1 on blackboard
//TODO: Study the NodeGraph and Node classes
//TODO: Study the SampleDungeonNodeGraph class and try it out below
//TODO: Comment out the SampleDungeonNodeGraph again, implement a HighLevelDungeonNodeGraph class and uncomment it below

//_graph = new SampleDungeonNodeGraph(_dungeon);
//_graph = new HighLevelDungeonNodeGraph(_dungeon);
if (_graph != null) _graph.Generate();
```

NodeGraph code



What does the starting code provide?

- Minimal data structures (or the skeleton of it) for every assignment
- Drawing code for debugging/visualization for every assignment
- Setup for each package is the same:



NodeGraph class

- Wraps all Nodes
- Has AddConnection convenience method
- Abstract generate method you have to override
- Drawing code to (re)draw the graph
- Click detection:
  - public Action<Node> OnNodeLeftClicked = delegate {};
  - public Action<Node> OnNodeRightClicked = delegate {};
  - …

Implementing (explicit) graphs

- On the outside your graph only has a couple of requirements:
  - Add/Get nodes (count)
  - Add/Get connections (count)

Option 1 : Lists

```
Node { List<Node> connections; }

Node a = new Node();
Node b = new Node();
Node c = new Node();
a.AddNode (b);
b.AddNode (c)
```

Option 2 : Dictionaries

```
Dictionary<Node, List<Node>>

Node a = new Node();
Node b = new Node();
Node c = new Node();
dictionary[a] = b;
dictionary[b] = c;
```

Option 3 : A Matrix

2D grid with **weights**

	a	b	c
a	-	1	-
b	-	-	1
c	-	-	-

Node class

- Very simple node class:
  - List<Node> connections;
  - Point location;
  - string id;

- Might require extension in future assignments

SampleDungeonNodeGraph class

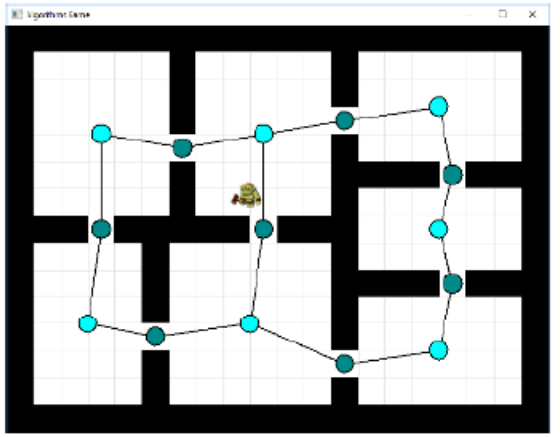
- Just adds some dummy nodes and connections
- Subclasses NodeGraph with some convenience methods:
  - getRoomCenter
  - getDoorCenter
- Use them, they'll make your life easier
  (Just like the Rectangle intersection / intersects methods ;))

Assignment 2: Graphs, tiles & agents

In this second assignment you have to convert your Dungeon into graphs and tiles, learning how intermediate forms of data might help you solve a more difficult problem. In addition, you will practice general list manipulation a bit more by moving Morc-da-Orc through its new lair.

Continue in your code from assignment 1. Again, a walkthrough for new parts of the code will be provided during the lectures, demonstrating all the visualization options you have to (/can) use. This code only has to work with the 'sufficient' requirements from assignment 1, but getting it to work with all of assignment 1 will earn you another candybar (or apple).

2.1 'Sufficient' requirements:

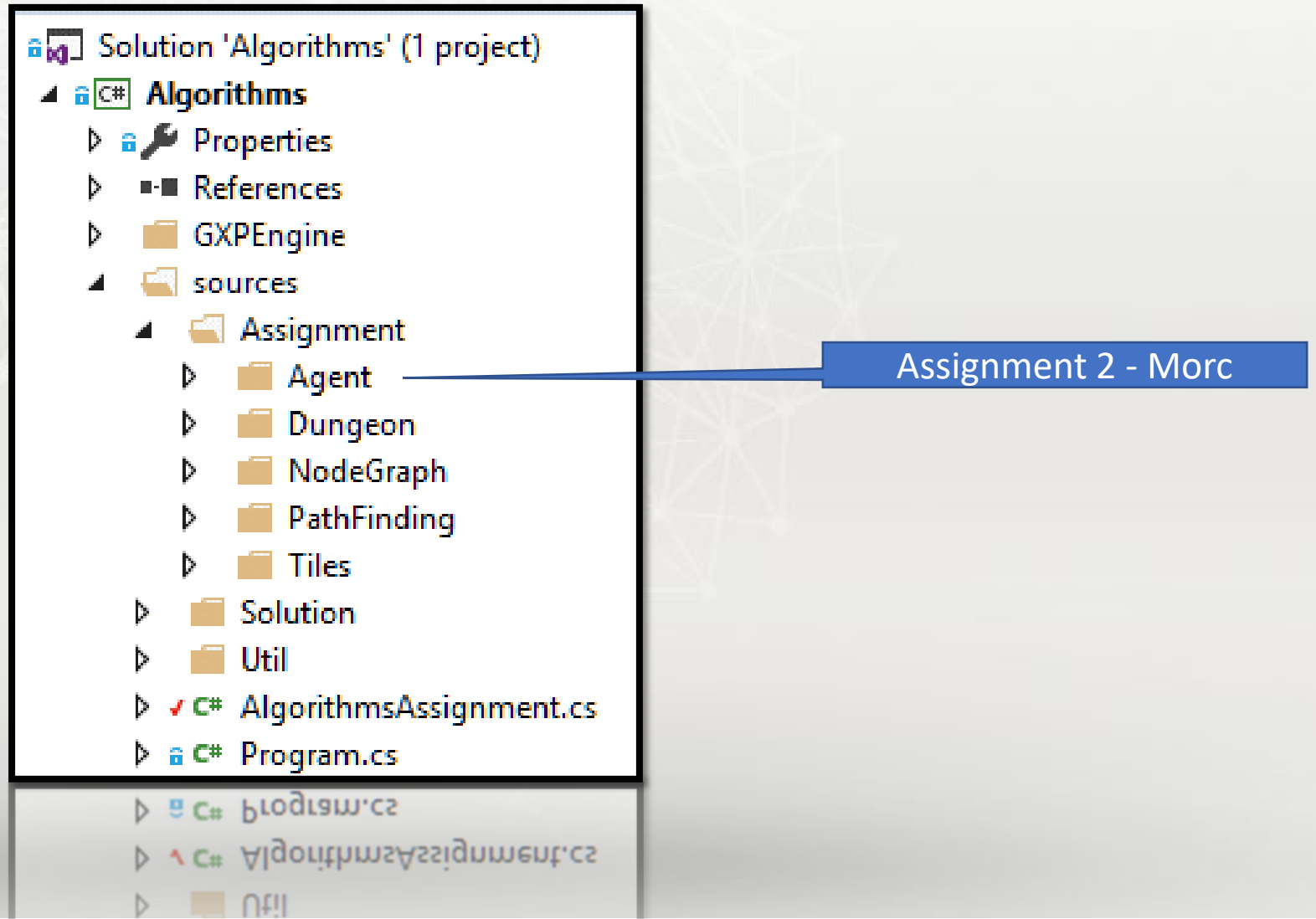


- Your graph replicates the dungeon structure with a node for every room and door with connections between them.
- Make sure that Morc queues clicked nodes and visit them in order. Morc is not allowed to go 'off graph' anymore, so make sure you ignore the clicked nodes that would cause this.

Part 2 – Morc the Orc

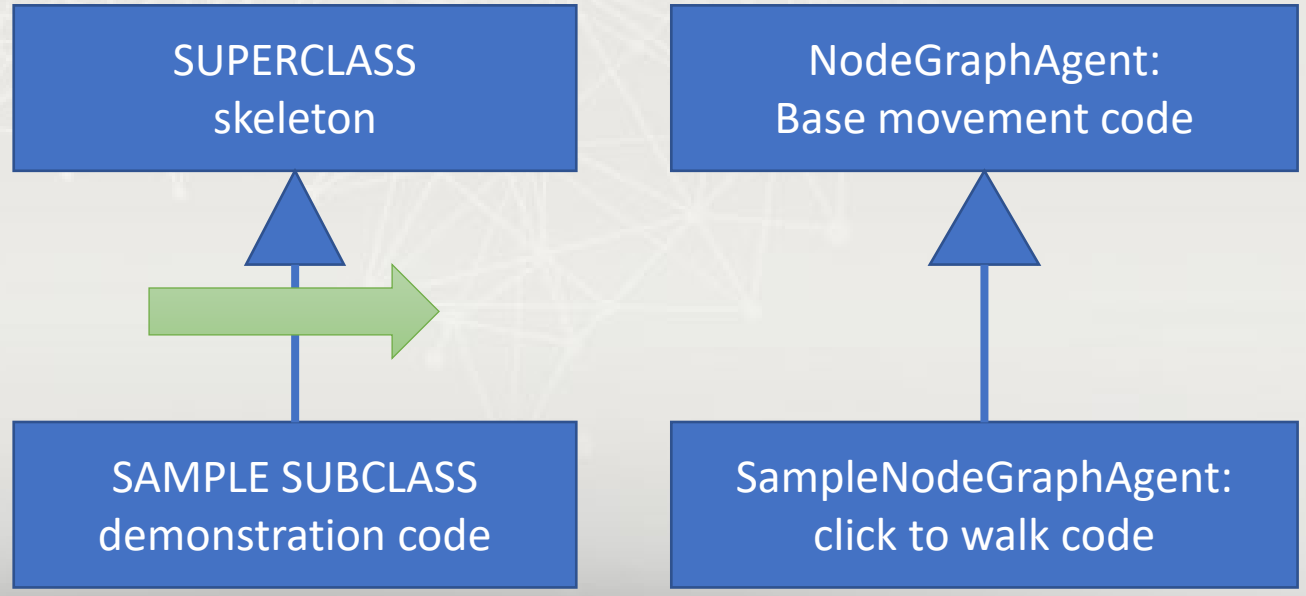
```
//////////////////////////////////////////////////////////////////
//Assignment 2.1 Sufficient (Mandatory) OnGraphWayPointAgent
//
//TODO: Study the NodeGraphAgent class
//TODO: Study the SampleNodeGraphAgent class and try it out below
//TODO: Comment out the SampleNodeGraphAgent again,
//      implement an OnGraphWayPointAgent class and uncomment it below

//_agent = new SampleNodeGraphAgent(_graph);
//_agent = new OnGraphWayPointAgent(_graph);
```

Agent code


Solution 'Algorithms' (1 project)
Algorithms
Properties
References
GXPEngine
sources
Assignment
Agent
Dungeon
NodeGraph
PathFinding
Tiles
Solution
Util
AlgorithmsAssignment.cs
Program.cs
Assignment 2 - Morc

What does the starting code provide?

- Minimal data structures (or the skeleton of it) for every assignment
- Drawing code for debugging/visualization for every assignment
- Setup for each package is the same:



Delegates

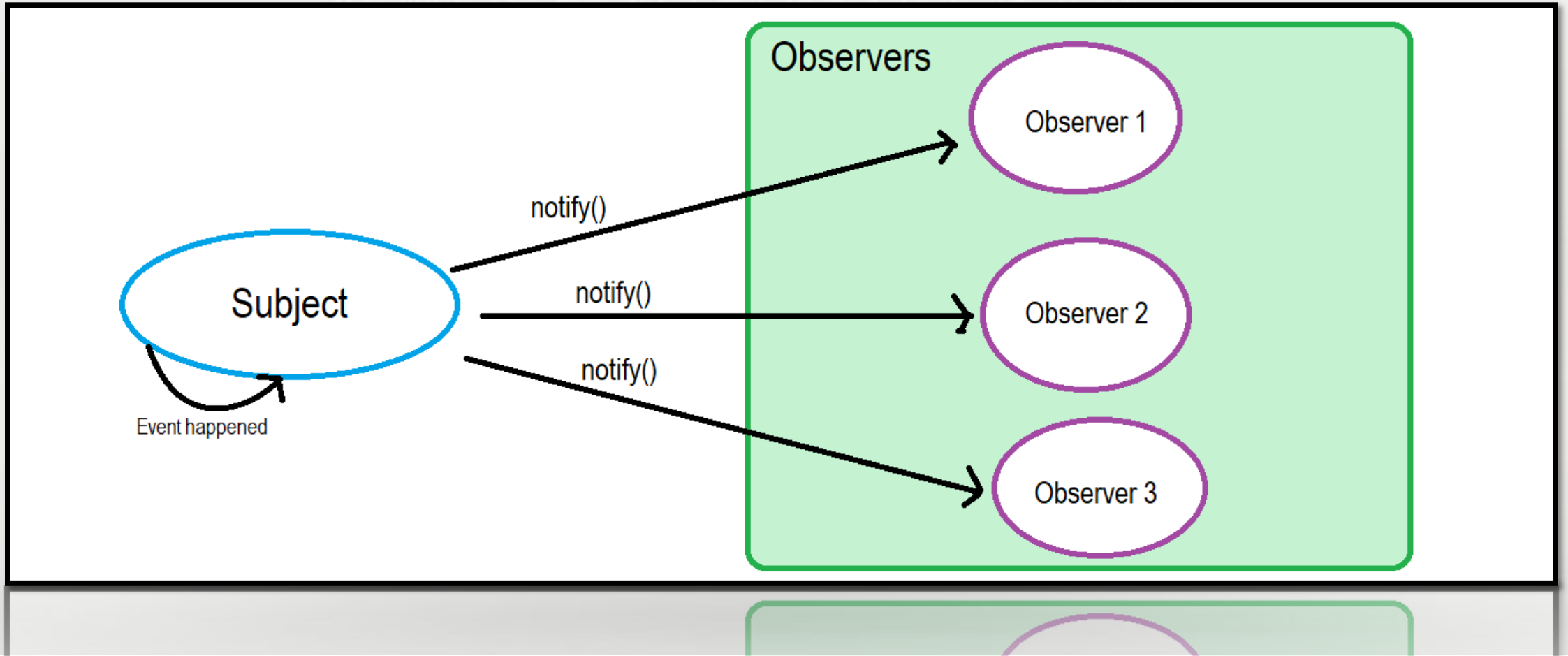
- Delegates are a variable type (we haven't seen yet before ?)
- A delegate variable is a function pointer, it stores a (list of) functions.
- For example the *Action<T>* definition:

```
namespace System
{
    ...public delegate void Action<in T>(T obj);
}
```

- This means that Action<T> points to a function that takes 1 argument and has no return values.

Observer/Observable – Publisher/Subscriber

- See 002_delegate_example



Closing words

Summing up: what did we cover today?

- Dictionary: what they are and how/when to use them
- Graphs: what they are and how/when to use them
- Assignment 2
  - Different ways to implement a Graph
  - Implementing a simple agent in a graph environment
  - Using Delegates and the Observer/Observable pattern

Interesting reads / credits

- https://blog.markvincze.com/back-to-basics-dictionary-part-1/
- https://www.loganfranken.com/blog/692/overriding-equals-in-c-part-2/
- https://docs.microsoft.com/en-us/archive/blogs/ericlippert/guidelines-and-rules-for-gethashcode
- https://en.wikipedia.org/wiki/Navigation_mesh

What's next?

- This week lab/homework:
  - Start with assignment 2
- Next week:
  - all about ***Recursion & Search Algorithms***
  - Introduction Assignment 3.1 Recursive part