# Topics for Today

- CS Theory
  - Stack and heap recap
  - Tree data structure
  - Depth and Breadth first searches
- This week's lab
  - Designing a graph data structure
  - Finding connected components
  - Solving a sliding tile puzzle
  - Knight moves (chess)

# Stack and Heap

Stack

- Fast allocation
- Only exists in scope
- Automatically deleted
- Fast deletion

Heap

- Uses `new` keyword
- Slow allocation
- Programmer decides lifetime
- Manually deleted
- Can cause memory leaks

```
Node stackNode{ val_: 1, head};
```

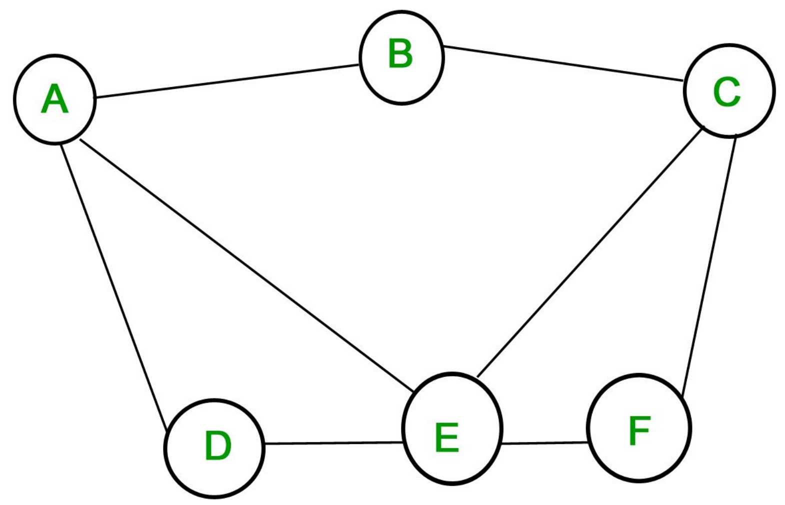
```
Node* heapNode = new Node{ val_: 2, head};
```

# Designing a Graph Data Structure

Most of the remainder of the course will be dealing with graph data structures. Let's take some time to think about how we can represent a graph through a data structure.

A graph is a collection of nodes, and the edges between the nodes.
So you need to think of a way of storing these two things.
And for this activity we'll look at ways of writing the following core functions:

- Add edge (u, v)
- Is edge (u, v)
- Adjacent To (v) - *get list edges on node v*
- Display - *print the graph*
- Constructor(number of nodes, directed?)



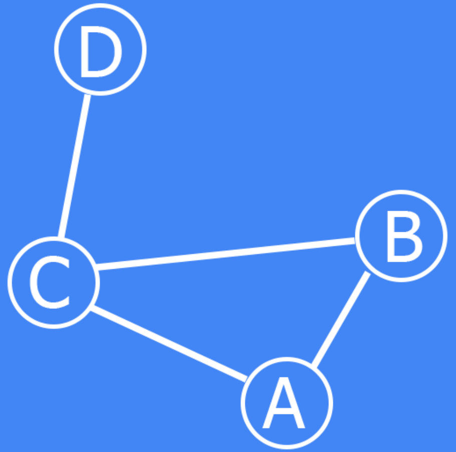
# Designing a Graph Data Structure

The approaches we will mostly be using in this course are:

1. Adjacency matrix - vector<vector<bool>>
   A vector of vectors, forming a matrix. Each node gets a row and a column, cells in which represent the edges in our graph. The matrix is always the same size, O(1) look up on edges.
2. Adjacency list - vector<list<int>> / vector<set<int>>
   A vector of lists/sets, each node has a set, containing the nodes it has an edge to. Tends to take up less space than the matrix, but has O(n)/O(logn) edge lookup.

```
   A B C D
A[[0,1,1,0],
B [0,0,1,0],
C [1,1,0,1],
D [0,0,1,0]]
```

```
A[{B,C},
B {A,C},
C {A,D},
D {C}]
```




Now go and implement a graph using one of these, we will be using it in the following exercise.

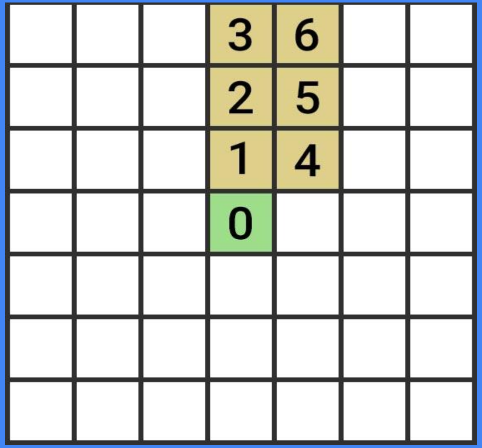
# Depth and Breadth First Searches

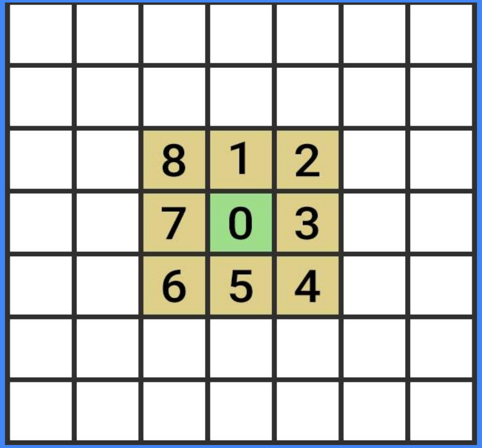
Depth-First Search

- Explores options to exhaustion before moving on
- Stack data structure

Breadth-First Search

- Explores options widely, looking at all options before moving in towards extremities.
- Queue data structure
- Vertices are entered into queue in order of their distance to start node.

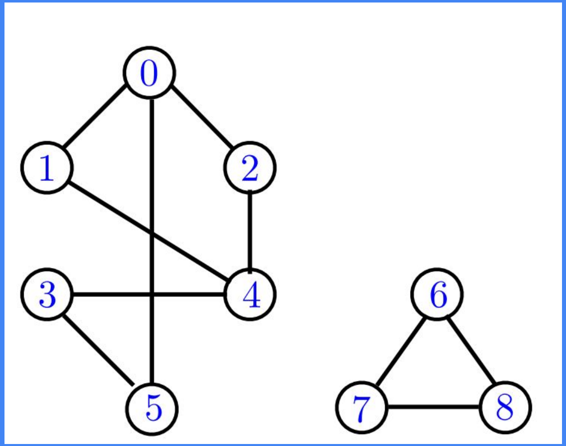


# Finding Connected Components

Using the code from the previous exercise, solve this problem.
In a graph, return how many connected components there are, and list the nodes in them.

A connected component is a group of nodes which can reach each other through any number of edges.




So for our example there are two connected components, {0,1,2,3,4,5}, and {6,7,8}

We will solve this using *Depth First Search,* which uses a stack to order its traversal.

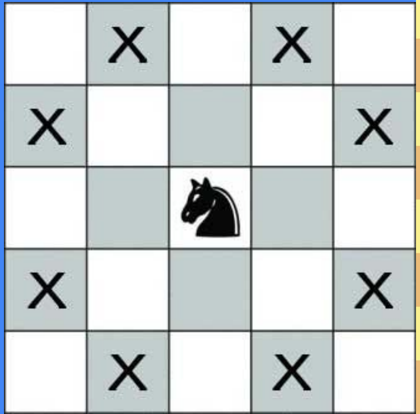
# Knight Moves

Using the properties of how a knight moves in chess (2 units in one direction, one unit in an orthogonal direction), find the fewest number of moves to move from (0,0) to a destination (x,y) on an infinite chess board.
We can represent this as an abstract graph. Each node represents an x-y position on the board, and the edges from that node represent the available moves, leading to new positions.

You have to write the function

```
std::vector<Point> Knight::minKnightMoves(const Point& dest)
```

Returning the smallest set of moves to get from the origin to a destination.
Solve this using *Breadth-first Search*
You also have to write a function to return the path, but I am happy to give that to you.



| | | | | | | |
|---|---|---|---|---|---|---|
| 3 | 2 | 3 | 2 | 3 | 2 | 3 |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 |
| 3 | 4 | 1 | 2 | 1 | 4 | 3 |
| 2 | 1 | 2 | 3 | 2 | 1 | 2 |
| 3 | 2 | 3 | 0 | 3 | 2 | 3 |
| 2 | 1 | 2 | 3 | 2 | 1 | 2 |
| 3 | 4 | 1 | 2 | 1 | 4 | 3 |
| 2 | 3 | 2 | 3 | 2 | 3 | 2 |