Basic Data Structures

So far in the class you have only really come across one type of data storage - arrays. In practice there are much more sophisticated ways of storing data depending on the application.

Here we will go over a couple of basic types that turn up all the time (you will be surprised how often).

The first is the stack. The idea here is imagine a stack of paper, you add always to the top and remove from the top. This is the so called last in first out idea (LIFO).

In [1]:
```
struct Stack
    data
end

function Base.show( io::IO, stack::Stack )
   print( stack.data )
end
```

In [2]:
```
function Base.push!( stack::Stack, item )
    push!( stack.data, item )
end
```

In [3]:
```
stack = Stack( [1,2,3] )
```

Out[3]:
```
[1, 2, 3]
```

In [4]:
```
push!( stack, 4 )
```

Out[4]:
```
4-element Vector{Int64}:
 1
 2
 3
 4
```

In [5]:
```
function Base.pop!( stack::Stack )
    pop!( stack.data )
end
```

In [6]:
```
pop!( stack )
```

Out[6]:
```
4
```

In [7]:
```
stack
```

Out[7]:
```
[1, 2, 3]
```

The second type is something called a queue. Imagine you are queueing up, then you enter at the end but are processed from the front in the order you arrived. This is the so called first in first out (FIFO).

In [8]:
```
struct Queue
    data
end

function Base.show( io::IO, queue::Queue )
   print( queue.data )
end
```

In [9]:
```
function Base.push!( queue::Queue, item )
    prepend!( queue.data, item )
end
```

In [10]:
```
queue = Queue( [1,2,3] )
```

Out[10]:
```
[1, 2, 3]
```

In [11]:
```
push!( queue, 4 )
```

Out[11]:
```
4-element Vector{Int64}:
 4
 1
 2
 3
```

In [12]:
```
function Base.pop!( queue::Queue )
    pop!( queue.data )
end
```

In [13]:
```
pop!( queue )
queue
```

Out[13]:
```
[4, 1, 2]
```

Why are these useful? Many reasons, for example parts of computer memory are structured as a stack. We will see one use very soon.

Graph Algorithms

In [14]:
```
struct Vertex
    neighbors::Vector{Int}       # Indices of neighbors of this Vertex
    coordinates::Vector{Float64} # 2D coordinates of this Vertex - only
    Vertex(neighbors; coordinates=[0,0]) = new(neighbors, coordinates)
end

function Base.show(io::IO, v::Vertex)
    print(io, "Neighbors = ", v.neighbors)
end
```

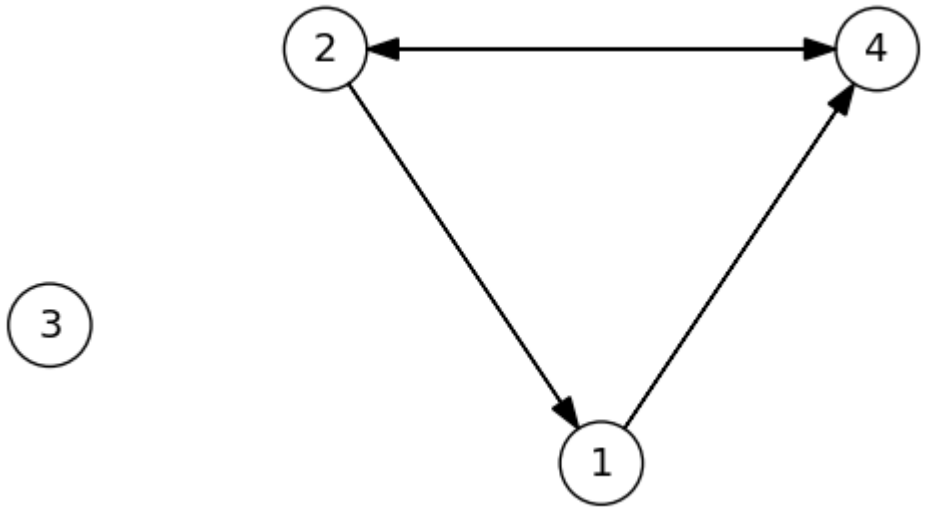
In [15]:
```
struct Graph
    vertices::Vector{Vertex}
end

function Base.show(io::IO, g::Graph)
    for i = 1:length(g.vertices)
        println(io, "Vertex $i, ", g.vertices[i])
    end
end
```

In [16]:
```
using PyPlot

function PyPlot.plot(g::Graph; scale=1.0)
    fig, ax = subplots()
    ax.set_aspect("equal")

    xmin = minimum(v.coordinates[1] for v in g.vertices)
    xmax = maximum(v.coordinates[1] for v in g.vertices)
    ymin = minimum(v.coordinates[2] for v in g.vertices)
    ymax = maximum(v.coordinates[2] for v in g.vertices)
    sz = max(xmax-xmin, ymax-ymin)
    cr = scale*0.05sz
    hw = cr/2
    axis([xmin-2cr,xmax+2cr,ymin-2cr,ymax+2cr])
    axis("off")

    for i in 1:length(g.vertices)
        c = g.vertices[i].coordinates
        ax.add_artist(matplotlib.patches.Circle(c, cr, facecolor="none",
        ax.text(c[1], c[2], string(i),
                horizontalalignment="center", verticalalignment="center"
        for nb in g.vertices[i].neighbors
            cnb = g.vertices[nb].coordinates
            dc = cnb .- c
            L = sqrt(sum(dc.^2))
            c1 = c .+ cr/L * dc
            c2 = cnb .- cr/L * dc
            arrow(c1[1], c1[2], c2[1]-c1[1], c2[2]-c1[2],
                  head_width=hw, length_includes_head=true, facecolor="k
        end
    end
end
```

The code above is copied word for word from Prof. Persson's notes.

A graph is a set of vertices and edges connecting them. They are extremely widely used in pretty much everything. For example one application is traffic flow between cities, facebook friends where each node represents a person and an edge between two nodes representing the fact that they are friends, ... list goes on.

In [17]:
```
v1 = Vertex([4], coordinates=[1,0.5])
v2 = Vertex([1,4], coordinates=[0,2])
v3 = Vertex([], coordinates=[-1,1])
v4 = Vertex([2], coordinates=[2,2])
g = Graph([v1,v2,v3,v4])

plot(g)
```



This is called a directed graph, in that the edge goes from 2 to 1 but not the other way around. An undirected graph is when if 1 goes to 2 then 2 also goes to 1.

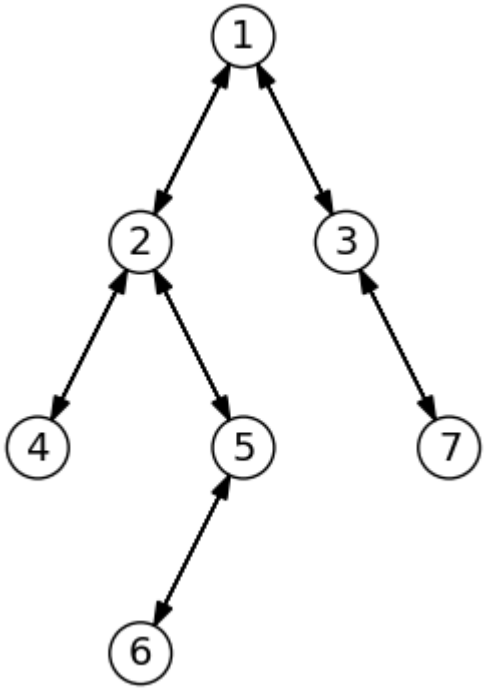
One of the most widely used graph algorithms is depth-first search (DFS). What DFS does is to search from each node to its neighbours, and traverses to each of its neighbours as far as possible before backtracking. If you think about it you will realise that the algorithm in Project 2 is actually DFS in disguise!

In [18]:
```
function dfs(g::Graph, start)
    visited = falses(length(g.vertices))
    tovisit = Stack( [start] )
    function visit(ivertex)
        visited[ivertex] = true
        println("Visiting vertex #$ivertex")
        for nb in g.vertices[ivertex].neighbors
            if !visited[nb]
                visited[nb] = true
                push!( tovisit, nb )
            end
        end
    end

    while length(tovisit.data) > 0
       visit( pop!(tovisit) )
    end
    return nothing
end
```

Out[18]:
```
dfs (generic function with 1 method)
```

Now let's generate a fairly complicated graph:

In [19]:
```
v1 = Vertex([2,3], coordinates=[0,0])
v2 = Vertex([1,4,5], coordinates=[-0.5,-1])
v3 = Vertex([1,7], coordinates=[0.5,-1])
v4 = Vertex([2], coordinates=[-1,-2])
v5 = Vertex([2,6], coordinates=[0,-2])
v6 = Vertex([5], coordinates=[-0.5,-3])
v7 = Vertex([3], coordinates=[1,-2])
g = Graph([v1,v2,v3,v4,v5,v6,v7])

plot(g)
```



Let's now run DFS:

In [20]:
```
dfs(g, 1)
```

```
Visiting vertex #1
Visiting vertex #3
Visiting vertex #7
Visiting vertex #2
Visiting vertex #5
Visiting vertex #6
Visiting vertex #4
```

In [21]:
```
g.vertices[1]
```

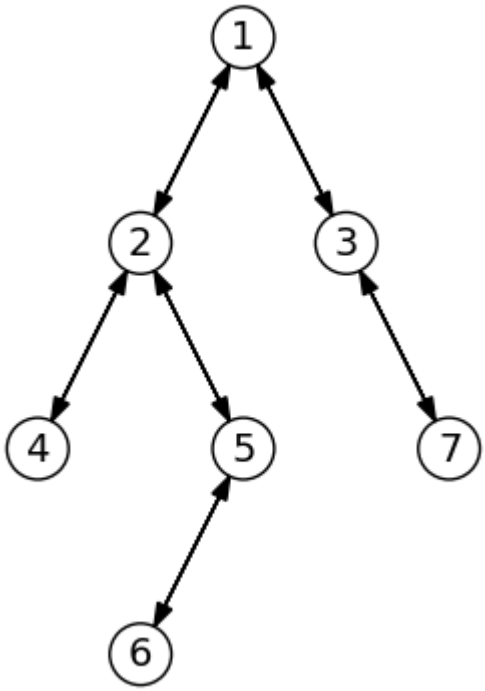
Out[21]:
```
Neighbors = [2, 3]
```

The second algorithm we will cover is breadth first search, or BFS. This instead of search all the way down the path of neighbours will search all neighbours of 1, then neighbours of those, and so on.

In [22]:
```
function bfs(g::Graph, start)
    visited = falses(length(g.vertices))
    tovisit = Queue( [start] )
    function visit(ivertex)
        visited[ivertex] = true
        println("Visiting vertex #$ivertex")
        for nb in g.vertices[ivertex].neighbors
            if !visited[nb]
                visited[nb] = true
                push!( tovisit, nb )
            end
        end
    end

    while length(tovisit.data) > 0
       visit( pop!(tovisit) )
    end
    return nothing
end
```

Out[22]:
```
bfs (generic function with 1 method)
```

In [23]:
```
plot(g)
bfs(g,1)
```




```
Visiting vertex #1
Visiting vertex #2
Visiting vertex #3
Visiting vertex #4
Visiting vertex #5
Visiting vertex #7
Visiting vertex #6
```

Notice the only difference in the two algorithsm above is the choice of data structure, DFS uses a stack, BFS uses a queue. The moral of the story is that a good data structure choice can make your algorithm much more efficient and also easier to implement!

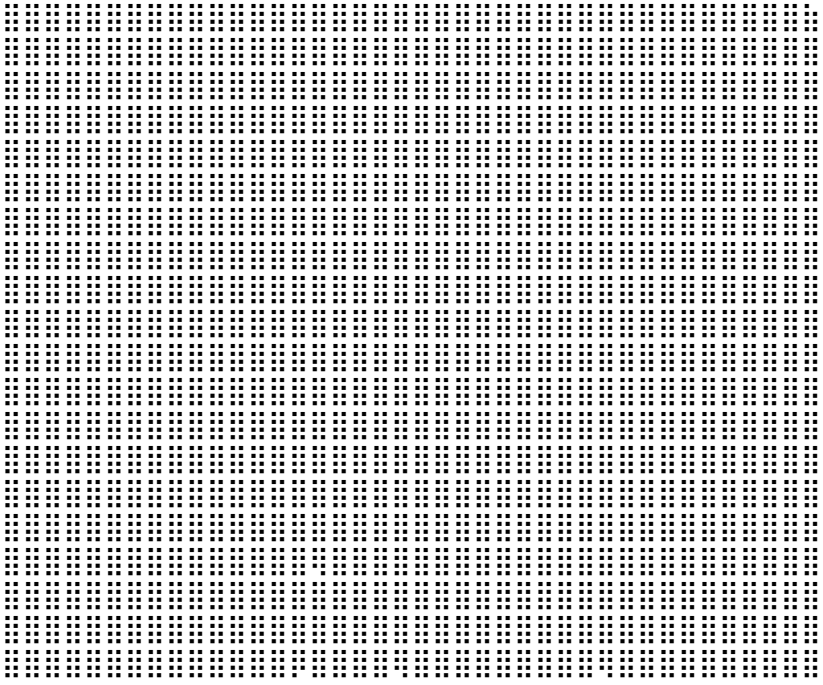
Sparse Matrices

One of the very important applications of graphs and good data structure usage is for sparse matrices. A sparse matrix is one where most of the entries are zero - meaning that we should not be storing them as it would be a waste of space. For a lot of applications, matrices tend to be sparse (~5% non-zero values). Specialised data structures and algorithms have been designed specially for this.

To show why this is a good idea:

In [24]:
```
using SparseArrays
```

In [25]:
```
mat1 = sprand( 1000,1000,0.05 )
```

Out[25]:
```
1000×1000 SparseMatrixCSC{Float64, Int64} with 50523 stored entries:
```


So I have generate an one million entry matrix where only 5 percent of the entries are non-zero. Let's compare the speed of running this in sparse form compared to dense form:

In [26]:
```
vec = rand( 1000 );
```

In [27]:
```
@time mat1 * vec;
```

```
  0.078659 seconds (70.11 k allocations: 4.290 MiB, 98.44% compilation
time)
```

In [28]:
```
@time Matrix( mat1 ) * vec;
```

```
  0.185208 seconds (376.31 k allocations: 30.134 MiB, 12.26% gc time,
96.41% compilation time)
```

So you can see that it is magnitudes more expensive in memory and in time. Bad idea.

The way sparse matrices are stored differs from language to language. Julia uses the Compressed Sparse Column format, for example:

In [29]:
```
rows = [1,3,4,2,1,3,1,4,1,5]
cols = [1,1,1,2,3,3,4,4,5,5]
vals = [5,-2,-4,5,-3,-1,-2,-10,7,9]

A = sparse(rows, cols, vals, 5, 5)
```

Out[29]:
```
5×5 SparseMatrixCSC{Int64, Int64} with 10 stored entries:
  5   ·   -3   -2   7
  ·   5    ·    ·   ·
 -2   ·   -1    ·   ·
 -4   ·    ·  -10   ·
  ·   ·    ·    ·   9
```

What Julia would store is that in column 1, there are 3 entries 5,-2,-4 in rows 1,3,4. And that in column 2 there is one entry in row 2. And so on. This way per the amount of storage instead of being n^2 is just 3 times the number of non-zero entries in the matrix.

One big application of sparse matrices is to represent graphs. Each row/column represents a node, and a one in the ij position means there is a connection from node i to node j. This is how for example search engines store links between websites etc.

It turns out that the eigenvector corresponding to the largest eigenvalue of this matrix will give you the node with the most neighbours linking to it. This is how the google pagerank algorithm works.

In [ ]:
```
```