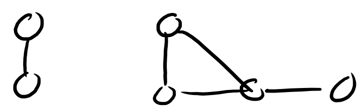
2019-11-07 Graphs

Thursday, November 7, 2019 9:05 AM

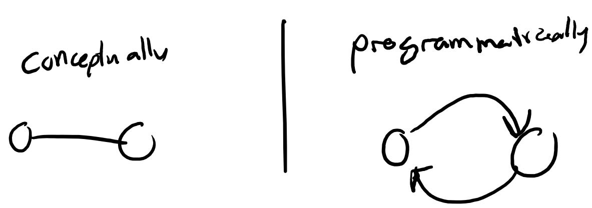
Graph Preliminaries

- Graphs are a "capstone" data structure in that they often will employ all of the other data structures discussed in this class. Depending on reason you are using a graph, you might encounter:
  - Hash tables
  - Priority queues
  - Vector
  - Linked Lists
  - Queues
  - Stacks
  - Recursion
- Graphs are just trees that can have multiple paths between two node
  - In math and more formal CS, graph nodes are called vertices
- Unlike trees, there may not exist a path between two nodes
  - Example:



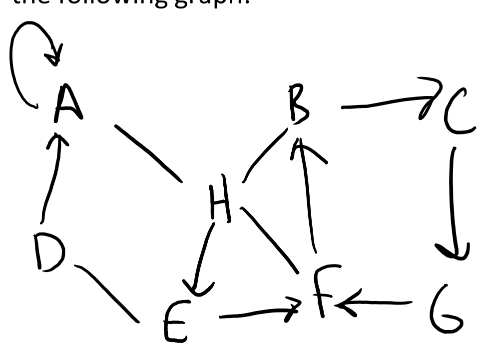
- All trees are graphs but not all graphs are trees
- In graph theory, edges can be unidirectional (one way) or bi-directional (two way)

- In applied computer science, all edges are unidirectional. Why?
  - Because in CS, edges are pointers and pointers can only point to one thing.
  - Implication: representing a bidirectional edge using two pointers



Programmatically representing a graph

- Consider the following graph:



Vector-based representation (Adjacency matrix)

- Adjacency matrix doesn't require any special custom classes for graph graph representation.
  - Hence, most examples on the web will use an adjacency matrix
- Matrices are typically in row-major order (we read across the matrix)
- In a basic adjacency matrix, we use 0 in a cell to represent not connected, 1 to represent connected

	A	B	C	D	E	F	G	H
A	1	0	0	0	0	0	0	1
B	0	0	1	0	0	0	0	1
C	0	0	0	0	0	0	1	0
D	1	0	0	0	1	0	0	0
E	0	0	0	1	0	1	0	0
F	0	1	0	0	0	0	0	1
G	0	0	0	0	0	1	0	0
H	1	1	0	0	1	1	0	0

- Sparse graph (matrix): Most cells contain 0 - not connected
- Representing a graph using an adjacency matrix costs O(N^2) memory
  - It's wasteful to represent a graph using an adjacency matrix when the graph is sparse (no need to represent NULLs)
- Instead, it is often more space efficient to use an edge list representation

Pointer-based (pseudo LL) Representation: Edge List

- (Adam's approach): Use a hashtable to store each node. Each item in the hash table has a hash table of pointers
  - In C++: unordered_map<string, unordered_map<string, int>>

Key	Value(HT<string,int>)
A	{A:1}, {H:1}
B	{C:1}, {H:1}
C	{G:1}
D	{A:1}, {E:1}
E	{D:1}, {F:1}
F	{B:1}, {H:1}
G	{F:1}
H	{A:1}, {B:1}, {E:1}, {F:1}

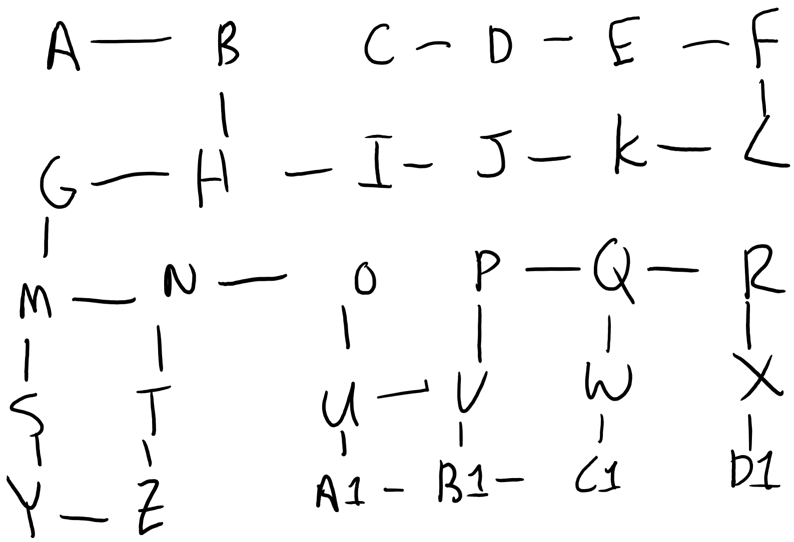
- Generally, as long as the graph is at least 50% empty, an edge list representation is more memory efficient

Searching a graph

- There are two strategies for searching a graph:

- (Stack / recursion) Depth first - Will touch nodes farther away before touching all nodes near the start of the search
- (Queue / iteration) Breadth first - "dropping a rock in pond" - We start with immediate neighbors ,then examine their neighbors, etc.

Example Search on some Maze



Adjacency Matrix

	A	B	C	D	E	F	...
A	1	1	0	0	0	0	0

Edge List

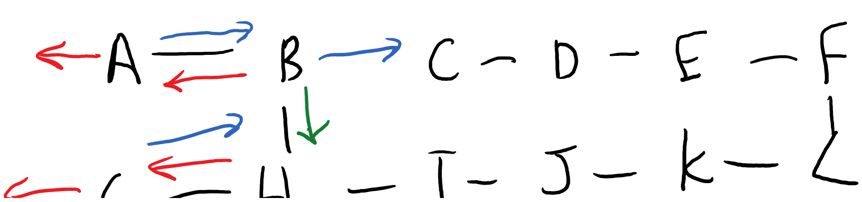
Key	Value
A	{"Self", A}, {"Right", B}
B	{"Left", A}, {"Down", "H"}, {"Self", B}

Depth-first search

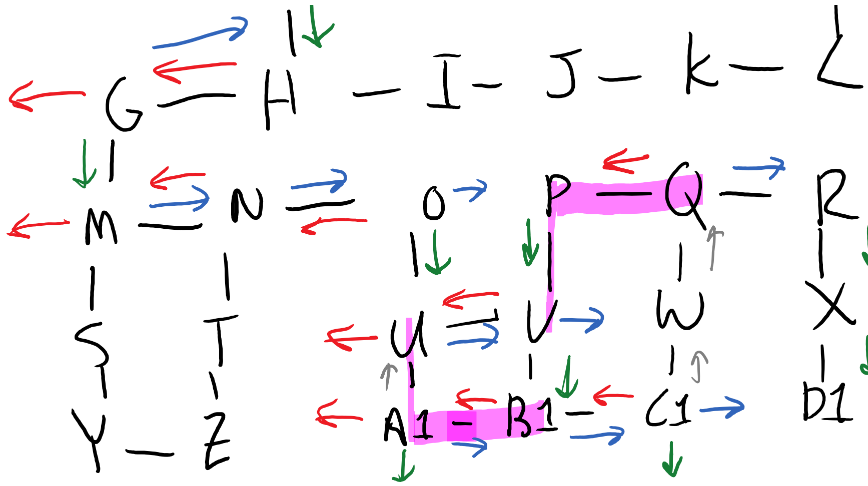
Function search(MazeSpace):

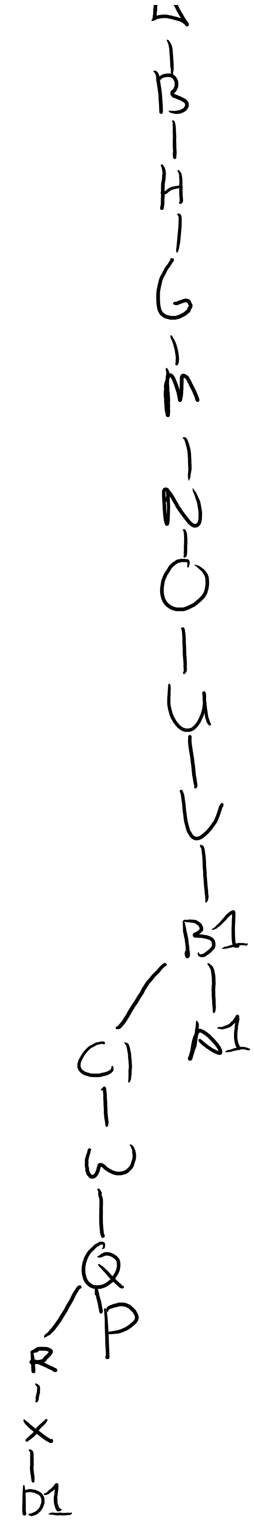
  If MazeSpace is null : return
  If MazeSpace seen before : return
  If MazeSpace is end: Done
  Search(MazeSpace->Left)
  Search(MazeSpace->Right)
  Search(MazeSpace->Below)
  Search(MazeSpace->Up)

Visit Order on DFS



DFS Tree





Breadth-first search

```
Function BFS(MazeSpace start):
	Queue<MazeSpace> to_vist{};
	to_visit.push(start)
	While to_visit.empty() == false:
		Front = to_visit.pop()
		If front is end: done
		If front not seen and front not null:
			To_visit.push(front->left)
			To_visit.push(front->right)
			To_visit.push(front->down)
			To_visit.push(front->up)
```