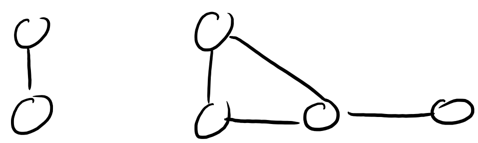
2019-04-18 Introduction to Graphs

Thursday, April 18, 2019 8:59 AM

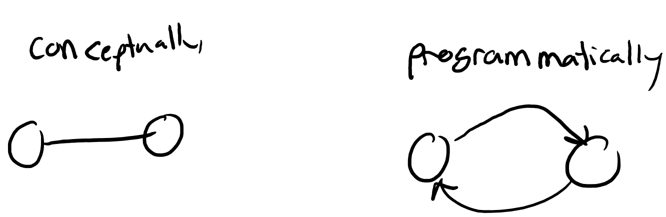
Graph Preliminaries

- Graphs are the "capstone" data structure in that it employs many of the data structures discussed in class. Depending on the problem, a graph might employ:
    - Hash tables
    - Priority queues
    - Vectors
    - Linked Lists
    - Queues
    - Stacks
- Graphs are trees that allow for multiple paths between two nodes
    - In math and more formal computer science, graph nodes are often called vertices
- Unlike trees in which all nodes are reachable, graphs may contain disconnected segments (not every node is reachable from every other node)
- Example:



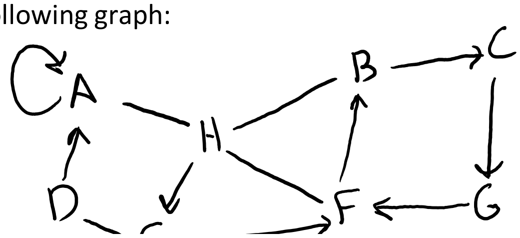
- All trees are graphs, but not all graphs are trees
- Graph edges can be unidirectional (one way) or bidirectional

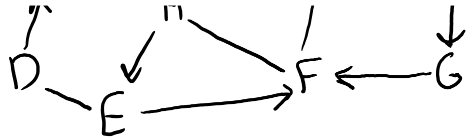
- In computer science, all edges are unidirectional? Why?
    - This is because we represent edges with pointers. Pointers can only point to one thing!
    - Thus, we need two pointers to represent a bidirectional edge.



Programmatically representing a graph

- Consider the following graph:





Vector-based representation (Adjacency matrix)

- Adjacency matrix doesn't require any special custom classes
- Matrix typically is in row-major order (we read across the matrix)
- In the matrix, 0 represents not connected, 1+ represents 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

- In an adjacency matrix, there is a quadratic relationship between the number of nodes and the number of cells.
- In many circumstances, graphs are "sparse" meaning that the number of possible edges that a node may far exceeds the actual number of edges.
  - Implication: we're wasting a lot of space
- When graphs are sparse and large, it is often more memory efficient to use an edge list representation

Pointer-based (pseudo LL) Representation: Edge List

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

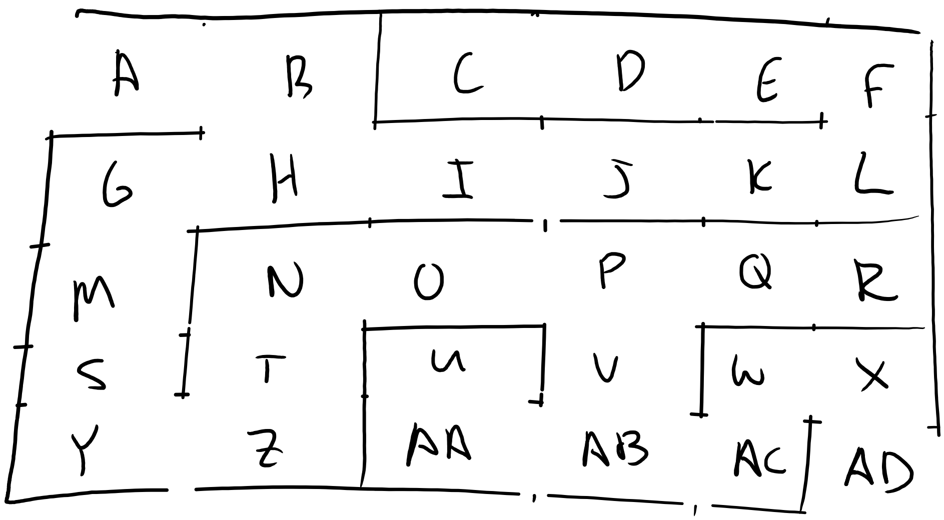
Key (string)	Value (HT<string,int>)
A	{A:1}, {H:1}
B	{C:1}, {H:1}
C	{G:1}
D	{A:1}, {E:1}
E	{D:1}, {F:1}
F	{B:1}, {H:1}
G	{F:1}
H	{A:1}, {B:1}, {E:1}, {F:1}

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

Searching a Graph

- There are two strategies for searching a graph:
    - Depth first - Will touch nodes farther away before touching all nodes near start of search
    - Breadth first - "dropping a rock in a pond" - We start with our immediate neighbors, then examine their neighbors, etc.

Example search on a maze



Example Adjacency Matrix

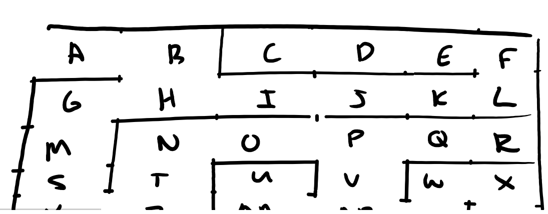
	A	B	C	D	E	F	G	...
A	1	1	0	0	0	0	0	0

Example Edge List

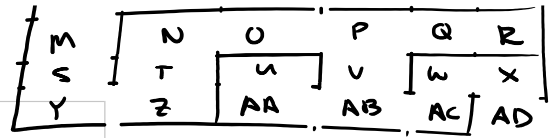
Key	Value
A	{"Self", A}, {"Right", B}
B	{"Left", A}, {"Self", B}, {"Below", H}

Depth-First Search

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



Visit Order

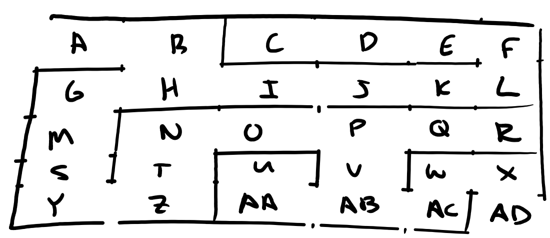


Visit Order

• NULL (A->L)	• Y (S->B)
• B (A->R)	• Z (Y->R)
• A (B->L) (seen, return)	• Y (Z->L) (seen)
• NULL (B->R)	• T (Z->U)
• H (B->B)	• Z (T->B) (seen)
• G (H->L)	• (ignore previously seen)
• NULL (G->L)	• N (T->U)
• H (G->R) (seen, return)	• O (N->R)
• M (G->B)	• P (O->R)
• NULL (M->L)	• Q (P->R)
• NULL (M->R)	• R (Q->R)
• S (M->B)	• V (P->B)
• (ignoring all nulls)	• AB (V->B)
	• AA (AB->B)
	• U (AA->A)
	• AC, W, X, AD

Breadth-First Search

Function BFS(MazeSpace start):
    Queue<MazeSpace> to_visit
    To_visit.push(start)
    While to_visit is not empty:
        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);



~~A~~, ~~B~~, ~~A~~, ~~H~~, ~~G~~, ~~I~~, ~~B~~, ~~H~~, ~~M~~, ~~H~~, ~~J~~, ~~S~~, G, I, K, Y, M

Visit order

- A, B, H, G, I, M, J, S,