CS280 Notes 3/21

By Christian Sagel

Choosing the “Right” Hash Tale
When performance matters more than memory.
With double hashing we could not delete items because there’s no clusters to insert.
With linear probing, 66% full, double hashing up to 90%.
With chaining, performance degrades more gracefully as each list grows. There is no clustering.
With chaining, you can use OTS algorithms and containers.
Hash tables rely on the fact that the data is uniformly and randomly distributed.
Since the data is not sorted in any meaningful way, operations such as displaying all items in a sorted fashion are expensive. (Using another data structure for that operation)
Hash tables are often used for databases, caches, string pools. For filesystems, with many many files to keep track of.
Perfect hash function generator: Given a known amount of keys, it will generate a function that will map them perfectly with zero collisions.

Introduction to graphs

One of the most useful data structures.
A lot of graph algorithms are named after people.
Trees are a a special, much simpler kind of graph. Graphs are general, less simple.
Games: Forcing moves by looking ahead.

A graph is essentially a collection of points connected by line segments.
The points are referred to as nodes or vertices. The segments are called edges.
If the edges have a direction (arrowheads in a diagram), the graph is a directed graph or digraph.

THere is no such thing as a root in a graph. No node is the starting point. To traverse the graph you have pick some node and walk through it. (WHereas in a tree you are guaranteed to hit every node only once)> With graphs you can end up going in cycles.
One of the fundamental things we have to know is whether two nodes are adjacent. Two nodes are adjacent if there is an edge conecting them.
Every node has

Priority queues will be very useful for storing the edges since they will be stored in order.
A path is a contiguous sequence of edges. We can have two types of connectivity:
Strongly connected: There is a path from each node to every other node.
Weakly connected: There is not a path from each node to every other node.

Cycles: A path w source and destination node are the same.
A cycle is simple if all nodes on the path are distinct. We don't repeat anything.
Sometimes its useful to know you won't get cycles in a graph. You can reduce overhead.
When we travel nodes, we have a boolean flag to note whether we have visited the node before lest we enter an infinite loop.
When it doe not have cycles, it is acyclic.

Degrees: For an undirected graph, the number of edges connecting to a node.
For a directed graph: in-degree is the number of incoming edges, out-degree outgoing edges.

Adjacency graph: Constant time lookups to find whether something is adjacent. The outdegree of a node can be read in the rows, the indegree in the column.
A sparse graph has few edges, a dense many.
Adjacency lists: A graph G with M nodes represented by an array of M linked lists. For each x and y, if xGy is true, y is on x's list. Much more expensive to find the in-degree.
The number of nodes in the lists is dependent on the number of edges.

Graph traversal:
The container C. Depending on what kind of container you use, you get different traversals. The different kinds are: stacks and queues. Depth-first or breadth-first traversal. It's a templatized function where you pass in the container that it will use.
On an unweighted graph, the order of traversal won't matter. If its weighted, the correct order will matter.