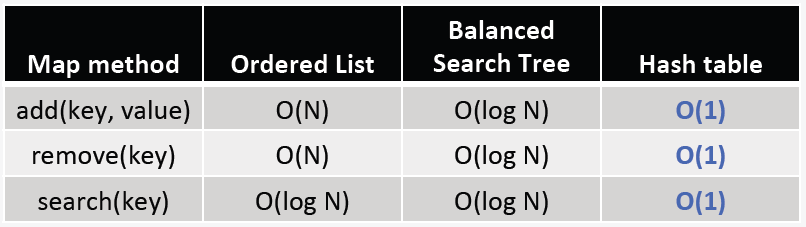
COMP 2210 – Exam 4 Study Guide Robert Sanek

**HASH TABLES**

**Map** (aka *associative array*, *dictionary*) – an abstract data type composed of key-value pairs, where each possible key occurs at most once.

**Hash Table** – a data structure that implements the map behavior by using a mathematical function (the hash function) to associate keys with the location in a table where that key and its associated value are stored.

* **Perfect hashing** – the hash function maps each distinct key onto a different index (can require infeasible amounts of memory)
* **Typical hashing** – two distinct keys will hash to the same index – a collision occurs.
  + Collisions degrade hash table performance: hash functions should be written to avoid collisions when possible.
* Good hash functions:
  + are **deterministic** – always give same index for same key
  + provide a **uniform distribution** of keys over indexes (each table index is equally likely for each key).
  + **fast**
* Hash functions convert the key into an integer hash code, and map that code onto a legal index value.

public int hash(Object key) {

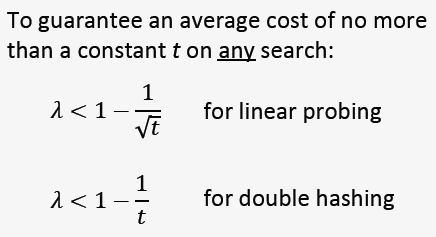
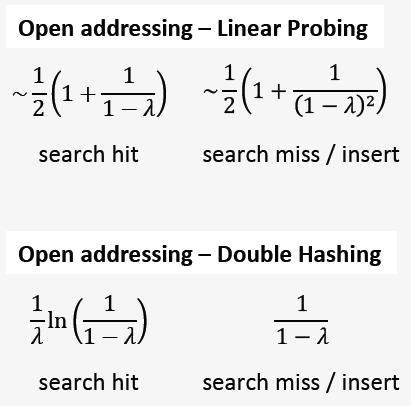
int hashCode = key.hashCode();

int index = (hashCode & Integer.MAX\_VALUE) % table.length;

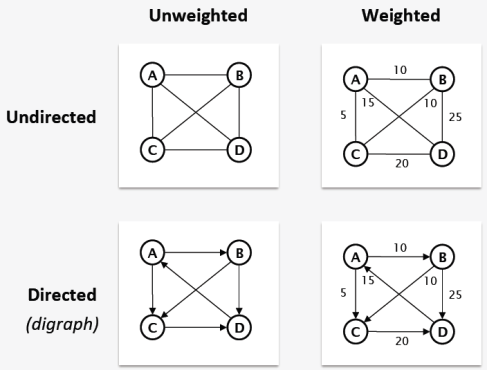
return index; }

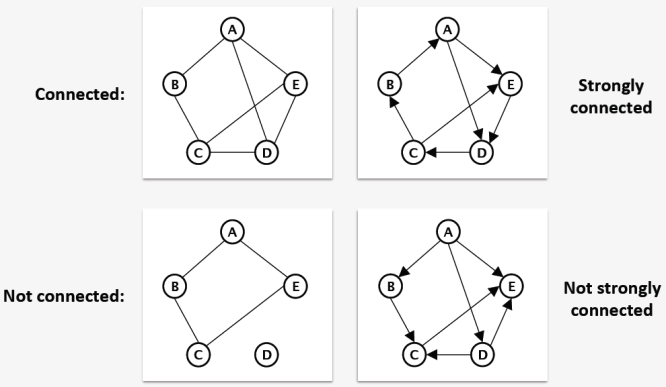
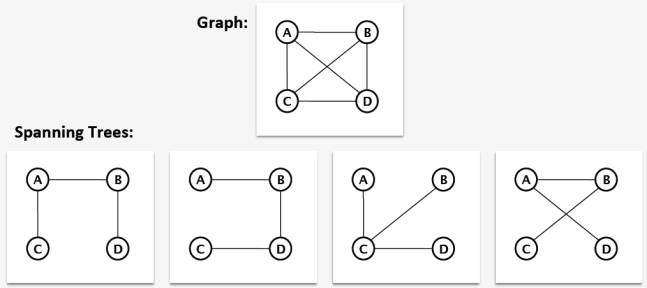
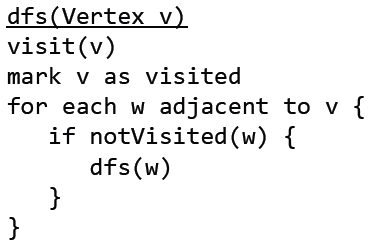
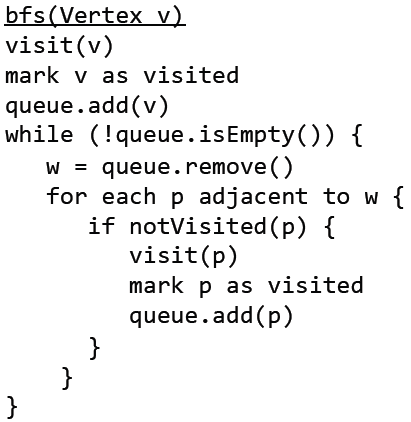
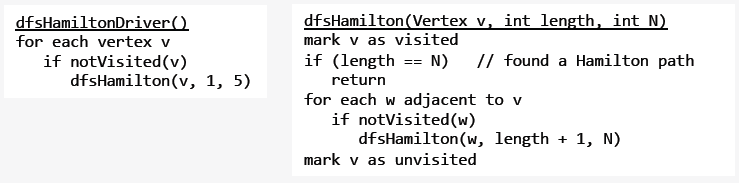
* + Use **hashCode** method inherited from Object to compute hash code, then use mod division to map the hash code onto a legal index value (“**modular hashing**”)
  + Problems arise when dealing with negative hash codes – use bitwise AND (&) operator to ensure valid index locations.
* **Uniform hashing** – for a hash table with M indexes and N keys, each index has to within a small constant factor N/M keys hashed to it. Being able to attain O(1) time complexity depends on uniform hashing.

**Collisions:** Open vs. Closed addressing collision resolution; applicable during insertion, search, & deletion

* **Open Addressing** – find an alternate to the home index *inside* the hash table. Each index contains at most one element reference.
  + Find an open index in the table (called **probing**)
  + Probing repeatedly applies following formula to resolve collision: index = (home + i\*C) % M
  + home: index at which collision occurred | i: number of probe attempts so far | C: constant multiplier | M: table.length
  + Linear Probing (C = 1), Quadratic Probing (C = i) & Double hashing (C = h2(hashcode) = 1 + (hashcode % (M-1))).
  + **Search**
    1. Hash the key that is being searched to find its home index.
    2. If the home index is empty, return not found.
    3. If the home index contains the key being searched for, return found.
    4. If the home index contains anything else, follow the probe sequence.
  + As the table becomes fuller, collisions become more likely.
    - **Load factor λ** – the ratio of elements in the table to the capacity of the table.
      * λ = N/M | empty table λ = 0, full table λ = 1
      * Most hash tables maintain the load factor between 0.5 and 0.8 (0.75 for HashMaps).
      * Once load factor is surpassed, rehash.
  + **Removing elements**
    - Linear Probing: make index empty, rehash elements that are possible collisions OR mark index as “unoccupied but not empty”
    - Double Probing: use “sentinel key” approach
* **Closed Addressing** – store all elements *outside* the hash table. Each index contains a reference to an auxiliary data structure (such as a linked list) that stores all the elements that hash to a given index.
  + **Removing elements**: just delete the element from the list
* Performance Analysis
  + Open Addressing: Double hashing provides the same performance in less space.
  + Closed Addressing: By managing λ appropriately, we can ensure that no search will require more than some constant amount of comparisons.

**GRAPHS**

**Graphs** represent pairwise relationships between objects, represented as vertices & edges.

* **Undirected** edges represent symmetric relationships, and are indicated by a line.
* **Directed** edges represent asymmetric relationships, and are indicated by an arrow.
* Edges can have numeric values (**weights**) associated with them.
* Adjacency
  + **Undirected**: two vertices are adjacent to each other iff there is an edge between them.
  + **Directed**: Node B is adjacent to node A **iff there is an edge from A to B**.
  + An adjacency matrix is a two-dimensional table where both the rows and the columns represent the vertices of the graph. Cell (*i, j*) indicates if a vertex *j* is adjacent to vertex *i*. For weighted graphs, weights are stored as table values.
    - Requires O(n2) memory for graphs with n vertices. Good for edge-existence questions: O(1)
  + An adjacency list is a one-dimensional table where each entry represents the vertices of the graph. Entry *k* stores a linked list of all the vertices that are adjacent to vertex *k*. For weighted graphs, nodes in linked list contain associated weights.
    - Requires O(*n* + *e*) memory for graphs with *n* vertices and *e* edges. Better for directed than undirected graphs.
* **self loop** – an edge that links back to itself
* **simple graph** – a graph with no self loops
* **path** – a sequence of vertices/edges from a start vertex to an end vertex.
* **simple path** – a path that does not cross the same edge twice.
* **cycle** – a simple path that starts and ends at the same vertex.
* **acyclic graph** – a graph with no cycles
* **connected graph** – a graph in which there is a simple path between any two pair of vertices.
* **connected component** – maximal subgraph that is connected
* **complete graph** – a simple graph in which every pair of vertices is adjacent to each other. Undirected edge count: N(N-1)/2, Directed: N(N-1)
* **Spanning tree** (for connected, undirected graphs) – a connected, acyclic subgraph that contains all of the vertices of the graph.
* **Graph Traversals**
  + Depth-First: simple recursive formulation. O(V + E) time complexity
  + Breadth-First: typically implemented iteratively with a FIFO queue. O(V + E) time complexity
* **Topological Sorting** – list the nodes in an order such that for every edge (u, v), u appears before v in the order.
* **Hamilton path** – a path in a graph that visits each vertex in the graph exactly once.