CSSE 230 Day 17

Introduction to graphs
and their common representations

Hash Table Implementation

Reminders/Announcements

- Doublets partner evaluation due Wednesday at noon
- WA6 due Thursday at 8
  - One actual written problem
  - Queens problem from Session 16
  - A couple more methods for ThreadedBinarySearchTree
- EditorTrees Milestone 1 due Monday
  - Recall that Milestone 1 requires much less than half of the total project effort
- Exam 2 Tuesday May 8, 7-9 PM.
- **Your questions?**
  - EditorTree requirements
  - Anything else

Graphs

- Terminology
  Representations
  Algorithms

Example Graph

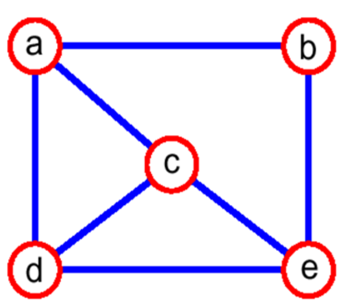
A **graph G** = (**V**,**E**) is composed of:

V: set of *vertices*

E: set of *edges* connecting the *vertices* in **V**

An **edge e** = (u,v) is a pair of vertices

Example:



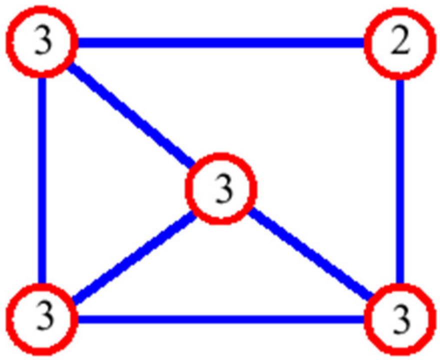
V= {a,b,c,d,e}

E= {(a,b),(a,c),(a,d),(b,e),(c,d),(c,e),(d,e)}

Graph Terminology

also called "neighbors"

- adjacent vertices: connected by an edge
- degree (of a vertex): # of adjacent vertices



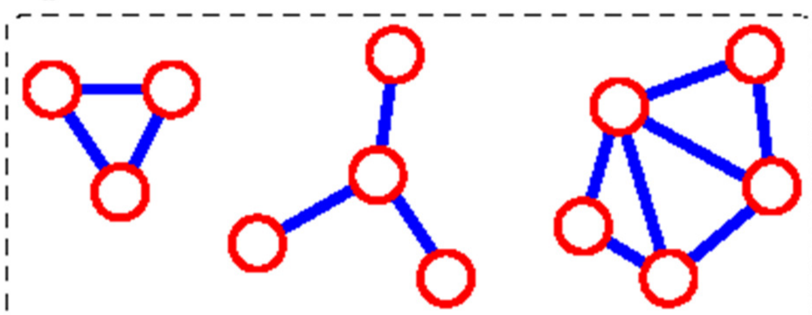
$$\sum_{v \in V} \deg(v) = 2(\#\ \text{edges})$$

- Since adjacent vertices each count the adjoining edge, it will be counted twice

Continuing Graph Terminology

1

connected component: maximal connected subgraph. E.g., the graph below has 3 connected components.

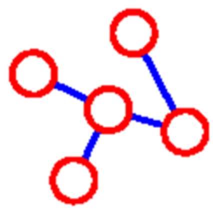


More Connectivity

n = #vertices
m = #edges

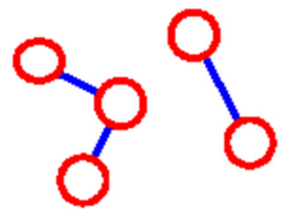
For a tree m = n - 1

A necessary but not sufficient condition for a graph to be a tree.



n = 5
m = 4

If m < n - 1, G is not connected



n = 5
m = 3

We represent vertices using a collection of objects

- Each Vertex object contains information about itself
- Examples:
  - City name
  - IP address
  - People in a social network

There are many options for representing edges of a graph

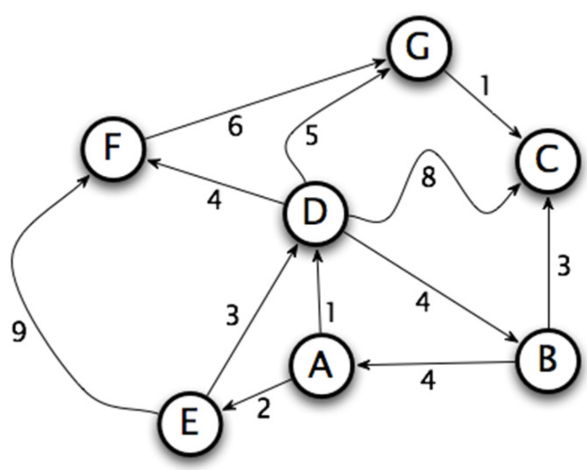
2-4

- Adjacency matrix
- Adjacency list. Each vertex stores…
  - pointers to other vertices?
  - named vertices using a HashMap<Name,Vertex>
  - An index into an array of the Vertex objectsI
    n each case, we need a way to store the vertex collection
- Edge list

To consider:
  Why not just use a triangular "matrix"?
  Does a boolean adjacency matrix make sense?
  What are the problems with the object-oriented approach?

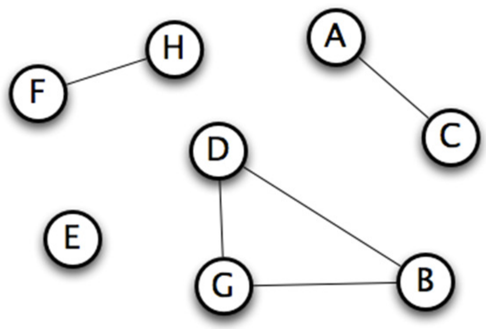
Sample graph problem: Weighted Shortest Path

- What's the cost of the shortest path from A to each of the other nodes in the graph?



Largest Connected Component

- What's the size of the largest connected component?



For much more on graphs, take MA/CSSE 473 or MA 477

Hashing

- Efficiently putting 5 pounds of data in a 20 pound bag

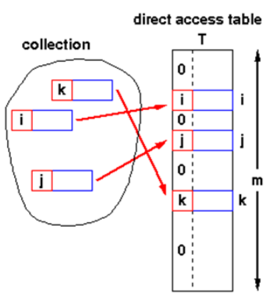
HashMap is a fast approach to dictionary storage

5

- **Functionality:** A HashMap implements a *finite function* H: K→V
  - domain of H is the set K of possible *keys*,
  - range is the set V of possible *values*
- **Main operations:** put(k, v), get(k), remove(k)
- **Representation:** Actual table data is stored in a large **array** of key-value pairs
- A HashSet uses a HashMap internally
  - Pay attention to keys; ignore the values.
- **Speed:** Insertion and lookup are constant time
  - with a good "hash function"
  - and a large storage array

On average

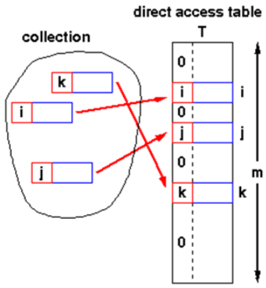
First approach: Direct Address Table



Contents of this slide are from John Morris, University of Western Australia. Adapted by Claude Anderson

- If we have a collection of n key-value pairs whose keys are unique integers in the range 0 .. m-1, where m >= n,
- then we can store the items in a *direct address* table, **T[m]**,
  - where **T[k]** is either null or contains the key-value pair for key k.
- Searching a direct address table is clearly an **O(1)** operation:
  - if T[k] is not null, get(k) returns **T[k].value**
  - otherwise returns `null`

First approach: Direct Address Table



- There are two main constraints:
  1. keys must be positive integers
  2. the set of possible keys must be severely bounded
     - largest key must be less than table size

The second constraint is often impossible to meet

And what if the domain of our map is some non-integer type?

We attempt to find a unique integer for each key by applying a hashCode( ) function …

6

key ➔ hashCode( ) ➔ integer

A good hashCode( ) function evenly distributes the keys, like:

hashCode("ate")= 48594983
hashCode("ape")= 76849201
hashCode("awe") = 14893202

Each class has its own hashCode() method. Default method is inherited from Object class.

Starting point for determining index in the array for this key.

What can go wrong?

...and then take that integer mod the table size (m) to get an index into the array.

- Example: if m = 100:

hashCode("ate")= 48594983 → mod → 83
hashCode("ape")= 76849201 → mod → 01
hashCode("awe") = 1489036 → mod → 36

Index is calculated from the object itself, not from a comparison with other objects in table

7–8

- Every Java object has a **hashCode** method that returns an integer H
  - It uses H % m as the index into the array

"ate"→ hashCode() → 48594983→ Mod 100 → 83

...
82
83 ate
84
...

  - Unless this position is already occupied

a "collision"

`Object` implements a default **hashCode** method

- Should we just inherit it?
- JDK classes override the **hashCode()** method
- If you plan to use instances of your class as keys in a hash table, you probably should too!

Choosing a hashCode( ) method for a class

- Should be fast to compute
- Should distribute keys as evenly as possible
- These two goals are often contradictory; we need to achieve a balance

A simple hash function for strings is a function that uses every character in its computaton

```
// This could be in the String class
public static int hash(String s) {
 int total = 0;
 for (int i=0; i<s.length(); i++)
   total = total + s.charAt(i);
 return Math.abs(total);
}
```

- Advantages?
- Disadvantages?

A better hash function for Strings also uses place value, but with a base that's prime

```
// This could be in the String class
public static int hash(String s) {
 int total = 0;
 for (int i=0; i<s.length(); i++)
   total = total*23 + s.charAt(i);
 return Math.abs(total);
}
```

- Spreads out the values more, and anagrams not an issue.
- We can't entirely avoid collisions. Why?
- What about overflow during computation?
- Note: **String** already has a reasonable `hashCode()` method; we don't have to write it ourselves.

Hash Table Caveats

- Objects that are equal (based on the equals method) **MUST** have the same hashCode values
- As much as possible, different objects should have different hashCodes
- Beware of mutable keys!
  - Python disallows mutable keys
- Hash tables don't maintain sorted order
  - So what's cost to find min or max element?

Collisions are Inevitable

- A hash table implementation (like HashMap) provides a "collision resolution mechanism"
- There are a variety of approaches to collision resolution
- Fewer collisions lead to faster performance

Collision Avoidance

10

- Just make hashCode unique?
- Possible key values >> capacity of table
  - Example: A key may be an array of 16 characters
  - How many different values could there be?
- Table size << possible hashCode values
- hashCode values are taken **mod** the current table size

Collision Resolution: Linear Probing

11

- Collision? Use the next available space:
  - Try H+1, H+2, H+3, …
  - Wrap around when we reach the end of the array
- Problem: Clustering
- Animation:
  - http://www.cs.auckland.ac.nz/software/AlgAnim/hash_tables.html

```
hash ( 89, 10 ) = 9
hash ( 18, 10 ) = 8
hash ( 49, 10 ) = 9
hash ( 58, 10 ) = 8
hash (  9, 10 ) = 9
```

	After insert 89	*After insert 18*	*After insert 49*	*After insert 58*	*After insert 9*
0			49	49	49
1				58	58
2					9
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Figure 20.4
Linear probing hash table after each insertion

Linear Probing Efficiency

12

- Depends on Load Factor, λ:
  - Ratio of the number of items stored to table size
  - $0 \le \lambda \le 1$.
- For a given λ, what is the expected number of probes before an empty location is found?

Rough Analysis of Linear Probing

13

- For a given λ, what is the expected number of probes before an empty location is found?
- Assume all locations are equally likely to be occupied, and equally likely to be the next one we look at.
- Then the probability that a given cell is full is λ and probability that a given cell is empty is 1-λ.
- What's the expected number?

$$\sum_{p=1}^{\infty} \lambda^{p-1}(1-\lambda)p = \frac{1}{1-\lambda}$$

Better Analysis of Linear Probing

14

- "Equally likely" probability is not realistic
- Clustering!
  - Blocks of occupied cells are formed
  - Any collision in a block makes the block bigger
- Two sources of collisions:
  - Identical hash values
  - Hash values that hit a cluster
- Actual average number of probes for large λ:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

For a proof, see Knuth, The Art of Computer Programming, Vol 3: Searching Sorting, 2nd ed, Addision-Wesley, Reading, MA, 1998.

Why consider linear probing?

- Easy to implement
- Simple code has fast run time per probe
- Works well when load is low
  - It could be more efficient to just get a bigger table and compute new locations for each item when table starts to fill.
  - Typically done in practice: rehash to an array that is double in size once the load factor goes over 0.75
- What about other fast, easy-to-implement strategies?

Quadratic Probing

- Linear probing:
  - Collision at H?   Try H, H+1, H+2, H+3,...
  - Guaranteed to succeed if array not completely full?
- Quadratic probing:
  - Collision at H?   Try H, $H+1^2$. $H+2^2$, $H+3^2$, ...
  - Eliminates primary clustering, but can cause "secondary clustering"
  - Will it always succeed?

Quadratic Probing Tricks (1/2)

15

- **Choose a prime number p for the array size**
- Then if $\lambda \le 0.5$:
  - Guaranteed insertion
    - If there is a "hole", we'll find it
  - No cell is probed twice
- See proof of Theorem 20.4 (done in CSSE 473):
  - Suppose that we repeat a probe before trying more than half the slots in the table
  - See that this leads to a contradiction
    - Contradicts fact that the table size is prime

Quadratic Probing Tricks (2/2)

- Use an algebraic trick to calculate next index to try
  - Replaces **mod** and general multiplication
  - Difference between successive probes yields:
    - Probe i location, $H_i = (H_{i-1} + 2i - 1)\ \%\ M$
  - Just use bit shift to "multiply" i by 2
  - Don't need mod, since i is at most M/2, so
    - ```
      probeLoc= probeLoc + (i << 1) - 1;
      if (probeLoc >= M)
              probeLoc -= M;
      ```
      ```

Quadratic probing analysis

- No one has been able to analyze it!
- Experimental data shows that it works well
  - Provided that the array size is prime, and is the table is less than half full

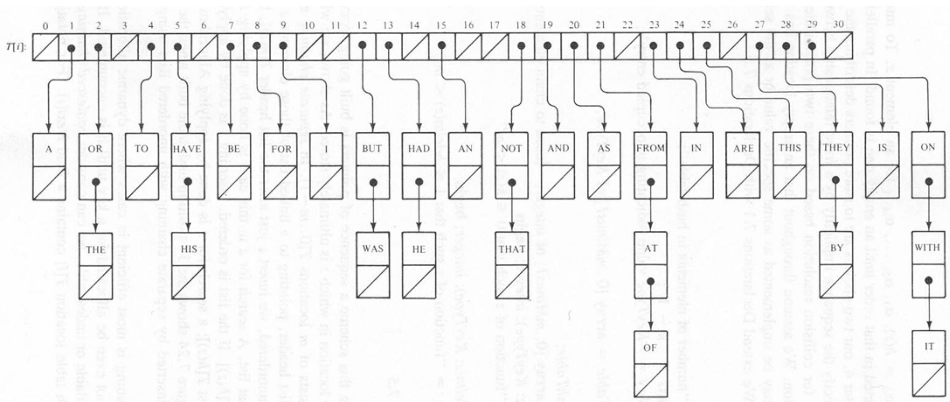
Another Approach: Separate Chaining

- Use an array of **linked lists**
- How would that help resolve collisions?

Hashing with Chaining

16–18

- Use an array of **linked lists**



Work Time

- WA6 or Editor Trees