CSCI 136
Data Structures &
Advanced Programming

Fall 2019

Lecture 33

2070567 & 82879

Administrative Details

Reminders

- No lab this week
- Final exam
    - Monday, December 16 at 9:30 in TCL 123 (Wege)
    - Covers everything, with strong emphasis on post-midterm
    - Study guide, sample exam will be posted on handouts page

Topics Covered

- Vectors (and arrays)
- Complexity (big O)
- Recursion + Induction
- Searching
- Sorting
- Linked Lists (SLL & DLL)
- Stacks
- Queues
- Iterators
- Bitwise operations
- Comparables/Comparators
- OrderedStructures
- Binary Trees
- Priority Queues
- Heaps
- Binary Search Trees
- Graphs
- Maps/Hashtables

Last Time

- Graph applications (more in Ch 16)
  - Dijkstra's Algorithm for shortest paths
    - Single source
  - Prim's algorithm for MCST

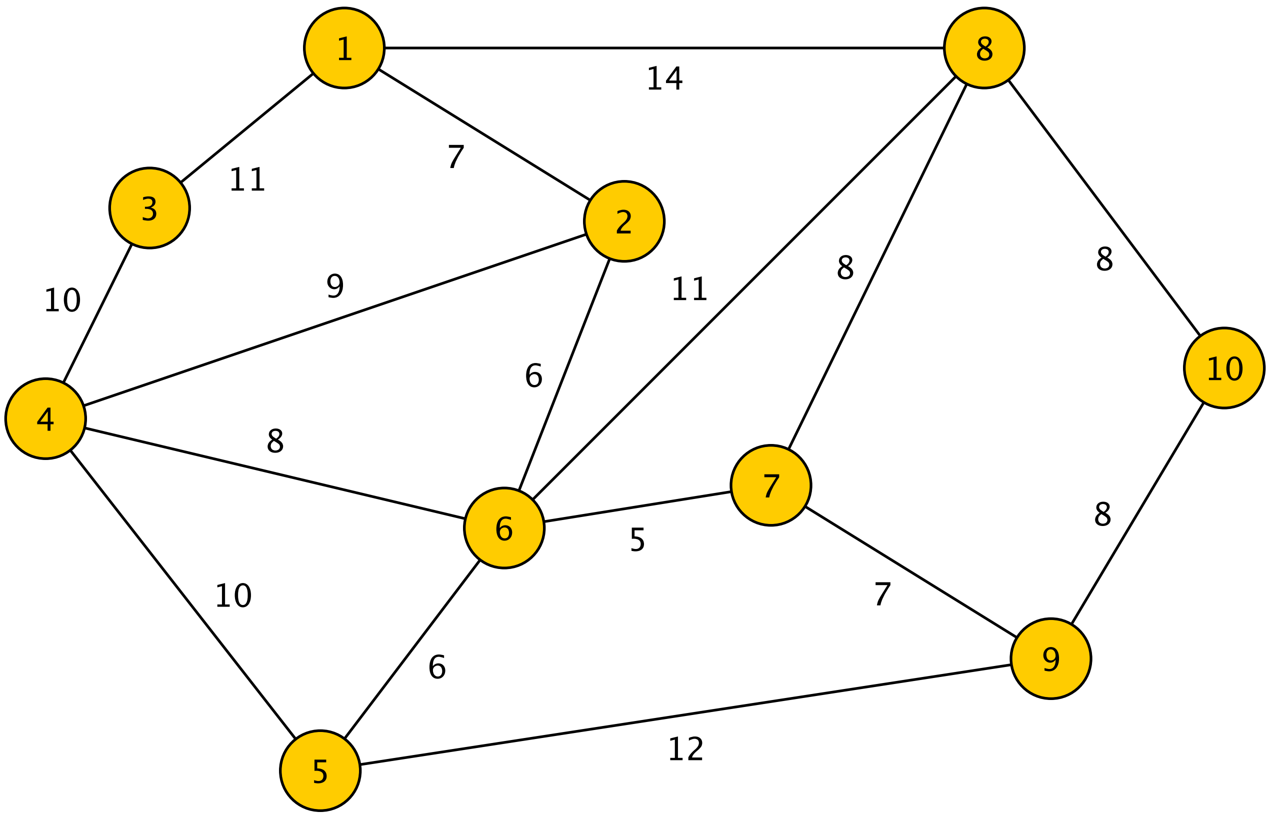
Today's Outline

- Finish MCST Discussion
- Maps
  - Revisit Naïve implementation from Lab 2
  - structure5.Hashtable (finally)
    - Hash functions
    - "Load factor"
    - Collisions and how to handle them
  - You should also read Ch 15 for more info

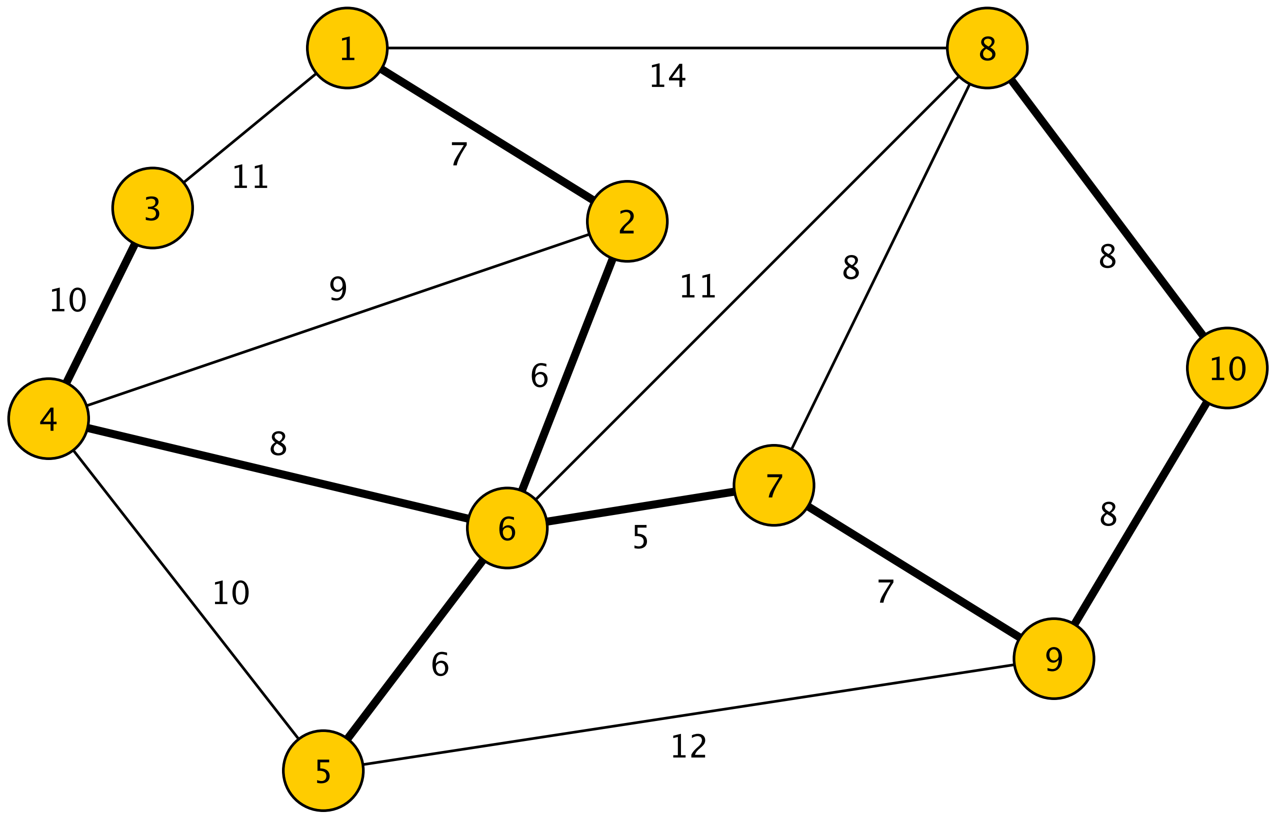
A Famous Problem

- Given a connected, undirected graph G=(V,E) with non-negative edge weights, find a minimum-weight, connected, spanning subgraph of G.
- Note: Such a subgraph must be a spanning tree!
- Frequently, we refer to the edge weights as *costs* and so this problem becomes:
- Given an undirected graph G with edge costs, compute a minimum-cost spanning tree of G.

Minimum-Cost Spanning Trees



Minimum-Cost Spanning Trees



Finding a MCST

Suppose we just wanted to find a PCST (pretty cheap spanning tree), here's one idea:

Grow It Greedily!

- Pick a vertex and find its cheapest incident edge. Now we have a (small) tree
- Repeatedly add the cheapest edge to the tree that keeps it a tree (connected, no cycles)
- This method is called *Prim's Algorithm*
- How close might this get us to the MCST?

An Amazing Fact

Thm: (Prim 1957) The greedy tree-growing algorithm always finds a minimum-cost spanning tree for any connected graph.

Contrast this with the greedy exam scheduling algorithm, which does *not* always find a minimum schedule (coloring)

Why does this work?

The Key

Def: Sets V_1 and V_2 form a *partition* of a set V if

$$V_1 \cup V_2 = V \text{ and } V_1 \cap V_2 = \emptyset$$

Lemma: Let G=(V,E) be a connected graph and let V_1 and V_2 be a partition of V. *Every* MCST of G contains a cheapest edge between V_1 and V_2

- Let e be a cheapest edge between V_1 and V_2
- Let T be a MCST of G. If $e \notin T$, then $T \cup \{e\}$ contains a cycle C and e is an edge of C
- Some other edge e' of C must also be between V_1 and V_2; e is a cheapest edge, so w(e') = w(e) [Why?]

Using The Key to Prove Prim

We'll assume all edge costs are distinct

Otherwise proof is slightly less elegant

Let T be the tree produced by the greedy algorithm and suppose T* is a MCST for G

Claim: T = T*

Idea of Proof: Show that every edge added to the tree T by the greedy algorithm is in T*

Clearly the first edge added to T is in T*

Why? Use the key!

Using The Key

Now use induction!

- Suppose, for some $k \geq 1$, that the first k edges added to T are in T*. These form a tree T_k
- Let V_1 be the vertices of T_k and let $V_2 = V - V_1$
- Now, the greedy algorithm will add to T the cheapest edge e between V_1 and V_2
- But any MCST contains the (only!) cheapest edge between V_1 and V_2, so e is in T*
- Thus the first k+1 edges of T are in T*

Prim's Algorithm

prim(G) // finds a MCST of connected G=(V,E)

let v be a vertex of G; set $V_1 \leftarrow \{v\}$ *and* $V_2 \leftarrow V_1 - \{v\}$

let A be the set of all edges between V_1 *and* V_2

while($|V_1| < |V|$)

    let $e \leftarrow$ *cheapest edge in A between* V_1 *and* V_2

    add e to MCST

    let $u \leftarrow$ *the vertex of e in* V_2

    move u from V_2 *to* V_1*;*

    add to A all edges incident to u

    // note: A now may have edges with both ends in V_1

Prim's Algorithm (Variant)

prim(G) // finds a MCST of connected $G=(V,E)$

let v be a vertex of G; set $V_1 \leftarrow \{v\}$ *and* $V_2 \leftarrow V_1 - \{v\}$

let $A \leftarrow \emptyset$ *// A will contain ALL edges between* V_1 *and* V_2

while $|V_1| < |V|$

    add to A all edges incident to v

    repeat

        remove cheapest edge e from A

    until e is an edge between V_1 *and* V_2

    add e to MCST

    let $v \leftarrow$ *the vertex of e in* V_2

    move v from V_2 *to* V_1;

Prim's Algorithm (Variant)

- Note: If G is not connected, A will eventually be empty even though $|V_1| < |V|$
- We fix this by
  - Replacing *while*($|V_1| < |V|$) with
    - *repeat … until* $|V_1| = |V|$ *or* $A = \emptyset$
  - Replacing *until e is an edge between* V_1 *and* V_2 with
    - *until* $A=\emptyset$ *or e is an edge between* V_1 *and* V_2
- Then Prim will find the MCST for the component containing v

Prim's Algorithm (Variant)

prim(G) // finds a MCST of connected G=(V,E)

let v be a vertex of G; set $V_1 \leftarrow \{v\}$ *and* $V_2 \leftarrow V_1 - \{v\}$

let A $\leftarrow \emptyset$ *// A will contain ALL edges between* V_1 *and* V_2

repeat *// Assume G contains at least 2 vertices….*

add to A all edges incident to v

repeat

remove cheapest edge e from A

until A is empty || e is an edge between V_1 *and* V_2

if e is an edge between V_1 *and* V_2

let v $\leftarrow$ *the vertex of e in* V_2

move v from V_2 *to* V_1*;*

until $|V_1| == |V|$ *or* $|A| = 0$

Implementing Prim's Algorithm

- We'll "build" the MCST by marking its edges as "visited" in G
- We'll "build" V_1 by marking its vertices visited
- How should we represent A?
  - What operations are important to A?
    - Add edges
    - Remove cheapest edge
  - A priority queue!
- When we remove an edge from A, check to ensure it has one end in each of V_1 and V_2

ComparableEdge Class

- Values in a PriorityQueue need to implement Comparable
- We wrap edges of the PQ in a class called ComparableEdge
  - It requires the label used by graph edges to be of a Comparable type

Prim's Algorithm (Variant)

```
prim(G) // finds a MCST of connected G=(V,E)
let v be a vertex of G; set V1←{v} and V2←V1 - {v}
let A ←∅          // A will contain ALL edges between V1 and V2
repeat            // Assume G contains at least 2 vertices....
        add to A all edges incident to v
        repeat
                remove cheapest edge e from A
        until A is empty || e is an edge between V1 and V2
        if e is an edge between V1 and V2
                let v←the vertex of e in V2
                move v from V2 to V1;
untill |V1|=|V| or |A| = 0
```

MCST: The Code

```
PriorityQueue<ComparableEdge<String,Integer>> q =
  new SkewHeap<ComparableEdge<String,Integer>>();

String v = null;           // current vertex
Edge<String,Integer> e;    // current edge
boolean searching;         // still building tree
g.reset();                 // clear visited flags

// select a node from the graph, if any
Iterator<String> vi = g.iterator();
if (!vi.hasNext()) return;
v = vi.next();
```

MCST: The Code

```
do {
     // visit the vertex and add all outgoing edges
     to the priority queue
     g.visit(v);
     Iterator<String> ai = g.neighbors(v);
     while (ai.hasNext()) {
           // turn it into outgoing edge
           e = g.getEdge(v,ai.next());
           // add the edge to the queue
           q.add(new
             ComparableEdge<String,Integer>(e));
     }
     ...
```

MCST: The Code

```
     searching = true;
     while (searching && !q.isEmpty()) {
           // grab next shortest edge
           e = q.remove();
           // Is e between V1 and V2 (subtle code!!)
           v = e.there(); // does e connect V1 to V2?
           if (g.isVisited(v)) v = e.here();
           if (!g.isVisited(v)) {
                 searching = false;
                 g.visitEdge(g.getEdge(e.here(),
                       e.there()));
           }
     }
} while (!searching);
```

Prim : Space Complexity

- Graph: O(|V| + |E|)
  - Each vertex and edge uses a constant amount of space
- Priority Queue O(|E|)
  - Each edge takes up constant amount of space
- Every other object (including the neighbor iterator) uses a constant amount of space
- Result: O(|V| + |E|)
  - Optimal in Big-O sense!

Prim : Time Complexity

Assume Map ops are O(1) time (not quite true!)

For each iteration of do ... while loop

- Add neighbors to queue: O(deg(v) log |E|)
  - Iterator operations are O(1) [Why?]
  - Adding an edge to the queue is O(log |E|)
- Find next edge: O(# edges checked * log |E|)
  - Removing an edge from queue is O(log |E|) time
  - All other operations are O(1) time

Prim : Time Complexity

Over *all* iterations of do ... while loop

Step I: Add neighbors to queue:

- For each vertex, it's O(deg(v) log |E|) time
- Adding over all vertices gives

$$\sum_{v \in V} \deg(v) \log |E| = \log |E| \sum_{v \in V} \deg(v) = \log |E| * 2 |E|$$

- which is O(|E| log |E|) = O(|E| log |V|)
  - $|E| \leq |V|^2$, so $\log |E| \leq \log |V|^2 = 2 \log |V| = O(\log |V|)$

Prim : Time Complexity

Over *all* iterations of do ... while loop

Step 2: Find next edge: O(# edges checked * log |E|)

- Each edge is checked at most once
- Adding over all edges gives O(|E| log |E|) again

Thus, overall time complexity (worst case) of Prim's Algorithm is O(|E| log |V|)

- Typically written as O(m log n)
  - Where m= |E| and n = |V|

Final Topic: Maps and Hashing

Map Interface

Methods for `Map<K, V>`

- `int size()` - returns number of entries in map
- `boolean isEmpty()` - true iff there are no entries
- `boolean containsKey(K key)` - true iff key exists in map
- `boolean containsValue(V val)` - true iff val exists at least once in map
- `V get(K key)` - get value associated with key
- `V put(K key, V val)` - insert mapping from key to val, returns value replaced (old value) or null
- `V remove(K key)` - remove mapping from key to val
- `void clear()` - remove all entries from map

Map Interface

Other methods for Map<K,V>:

- `void putAll(Map<K,V> other)` - puts all key-value pairs from Map other in map
- `Set<K> keySet()` - return set of keys in map
- `Set<Association<K,V>> entrySet()` - return set of key-value pairs from map
- `Structure<V> valueSet()` - return set of values
- `boolean equals()` - used to compare two maps
- `int hashCode()` - returns hash code associated with data in map (stay tuned…)

Dictionary.java

```
public class Dictionary {

    public static void main(String args[]) {
        Map<String, String> dict = new Hashtable<String, String>();
        …
        dict.put(word, def);
        …
        System.out.println("Def: " + dict.get(word));
    }

}
```

What's missing from the Map API that a BST provides?

`successor(key), predecessor(key)`

Maps do NOT preserve order!

Simple Implementation: MapList

- Uses a SinglyLinkedList of Associations as underlying data structure
  - Think back to Lab 2, but a List instead of a Vector
- How would we implement `get(K key)`?
- How would we implement `put(K key, V val)`?

MapList.java

```
public class MapList<K, V> implements Map<K, V>{

    //instance variable to store all key-value pairs
    SinglyLinkedList<Association<K,V>> data;

    public V put (K key, V value) {
        Association<K,V> temp =
                            new Association<K, V> (key, value);
        // Association equals() just compares keys
        Association<K,V> result = data.remove(temp);

        data.addFirst(temp);
        if (result == null)
                return null;
        else
                return result.getValue();
    }
}
```

Simple Map Implementation

- What is MapList’s running time for:
  - containsKey(K key)?
  - containsValue(V val)?
- Bottom line: not O(1)!

Search/Locate Revisited

- How long does it take to search for objects in Vectors and Lists?
  - O(n) on average
- How about in BSTs?
  - O(log n)
- Can this be improved?
  - Hash tables can locate objects in *really quickly*!
    - (we will cover two reasons that O(1) performance is a fuzzy claim)

Example from Bailey

"We head to a local appliance store to pick up a new freezer. When we arrive, the clerk asks us for the last two digits of our home telephone number! Only then does the clerk ask for our last name. Armed with that information, the clerk walks directly to a bin in a warehouse of hundreds of appliances and comes back with the freezer in tow."

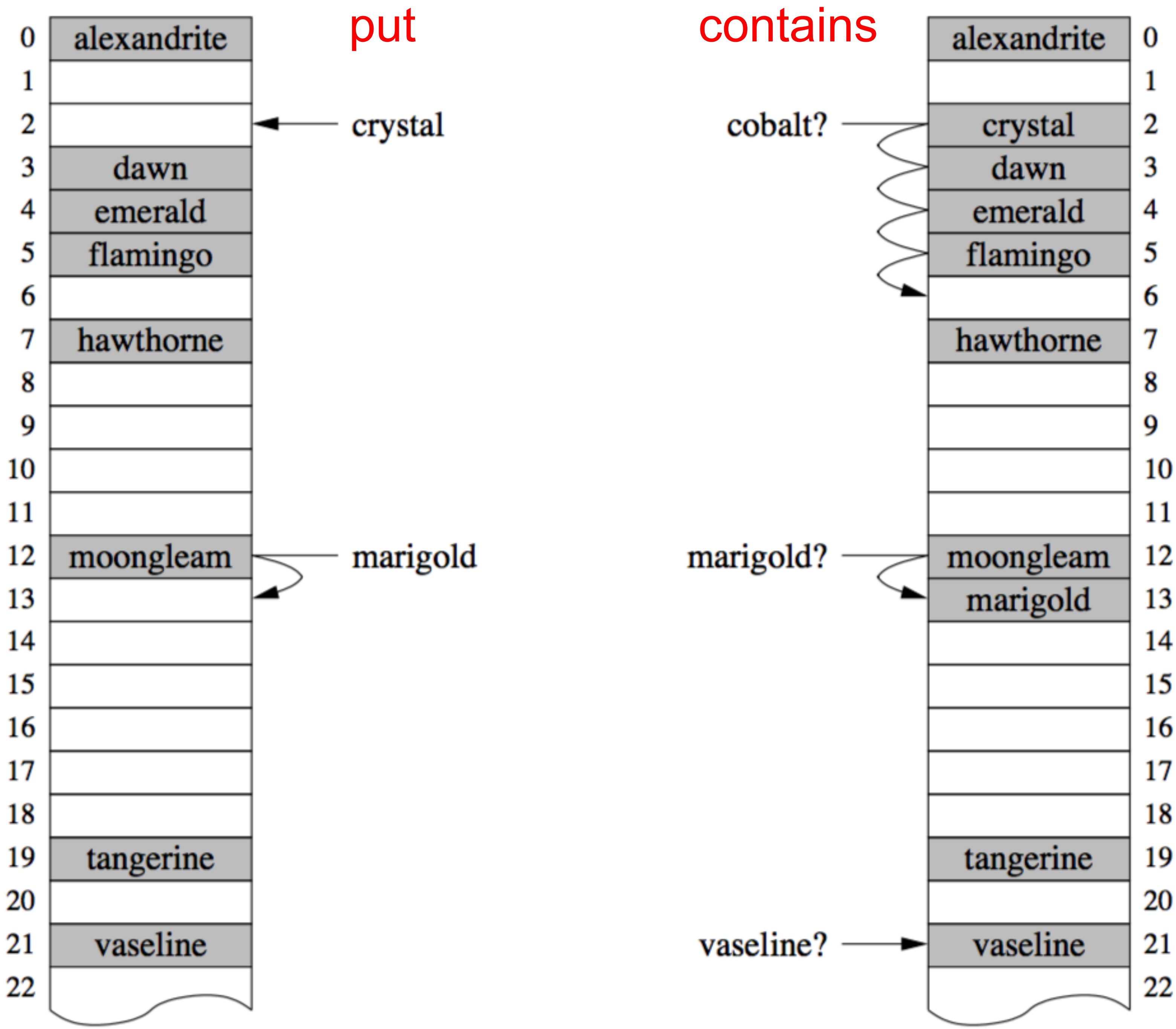
- Thoughts?
  - What is Key? What is Value?
  - Are names evenly distributed?
  - Are the last 2 phone digits evenly distributed?

Hashing in a Nutshell

- Assign objects to “bins” based on key
- When searching for object, go directly to appropriate bin (and ignore the rest)
- If there are multiple objects in bin, then search for the correct one
- Important Insight: Hashing works best when objects are evenly distributed among bins
  - Phone numbers are randomly assigned, last names are not (there were a lot of Smiths in Smithsville!)

Implementing a HashTable

- How can we represent bins?
- Slots in array (or Vector, but arrays are faster)
    - Initial size of array is a prime number
- How do we find a key's bin number?
    - We use a *hash function* that converts keys into integers
    - In Java, all Objects have `public int hashCode()`
        - Hashing function is one way: key fingerprint
        - Hashing function is deterministic



put
contains
0 alexandrite
1
2 crystal
3 dawn
4 emerald
5 flamingo
6
7 hawthorne
8
9
10
11
12 moongleam
13 marigold
14
15
16
17
18
19 tangerine
20
21 vaseline
22
crystal
marigold
cobalt?
marigold?
vaseline?