CSCI 136
Data Structures & Advanced Programming

Fall 2017

Lecture 33

The 2070567s

Administrative Details

Reminders

- No lab this week
- Problem set 3
  - Due Wednesday at start of class
- Final exam
  - Thursday, December 14 at 9:30 in TBL 112
  - Covers everything, with strong emphasis on post-midterm
  - Study guide, sample exam will be posted
  - We will have a review (during reading period)

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

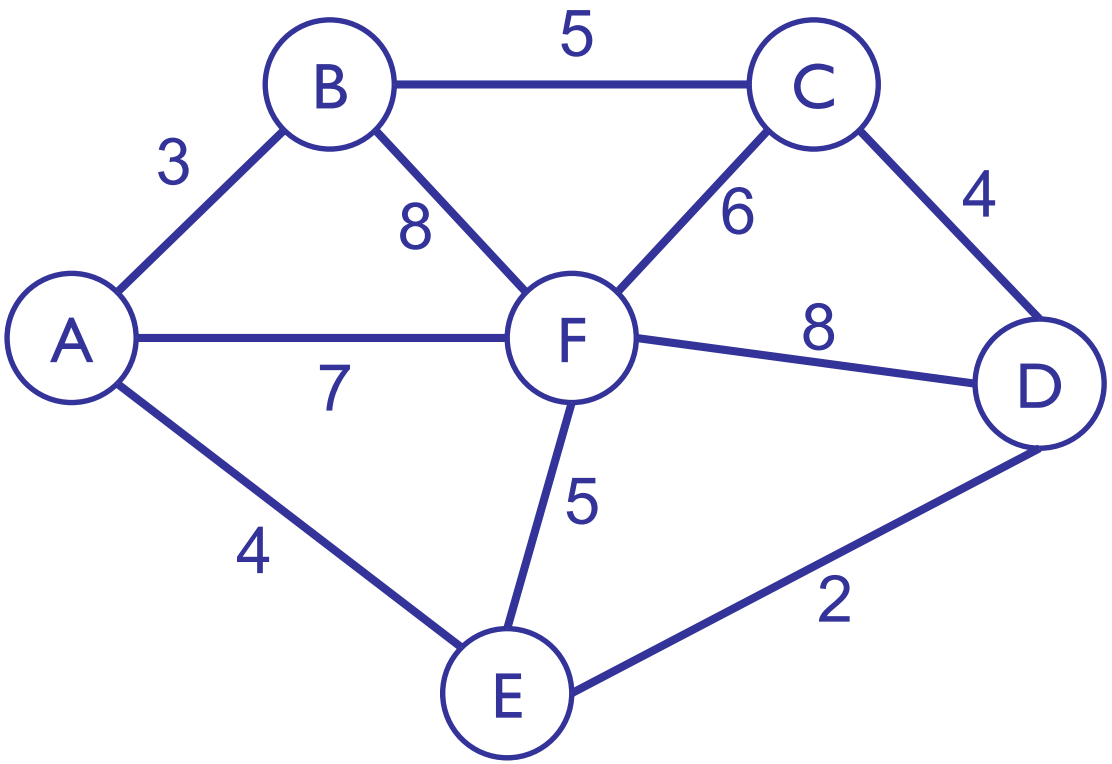
- toString() and printing Graphs
- Graph applications (more in Ch 16)
  - Prim's algorithm for MCST

Today's Outline

- Finish Prim's algorithm
- Maps (#2 Interface of all time)
  - 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

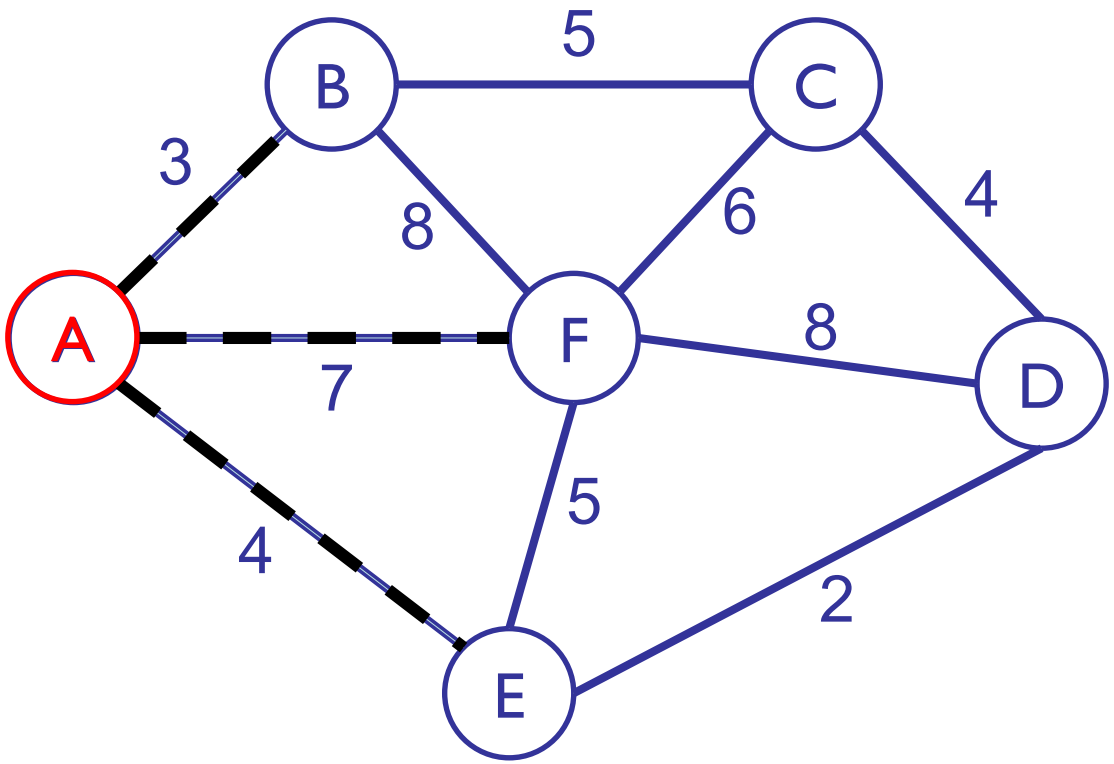
Prim's Algorithm

Priority Queue



MCST

Prim's Algorithm



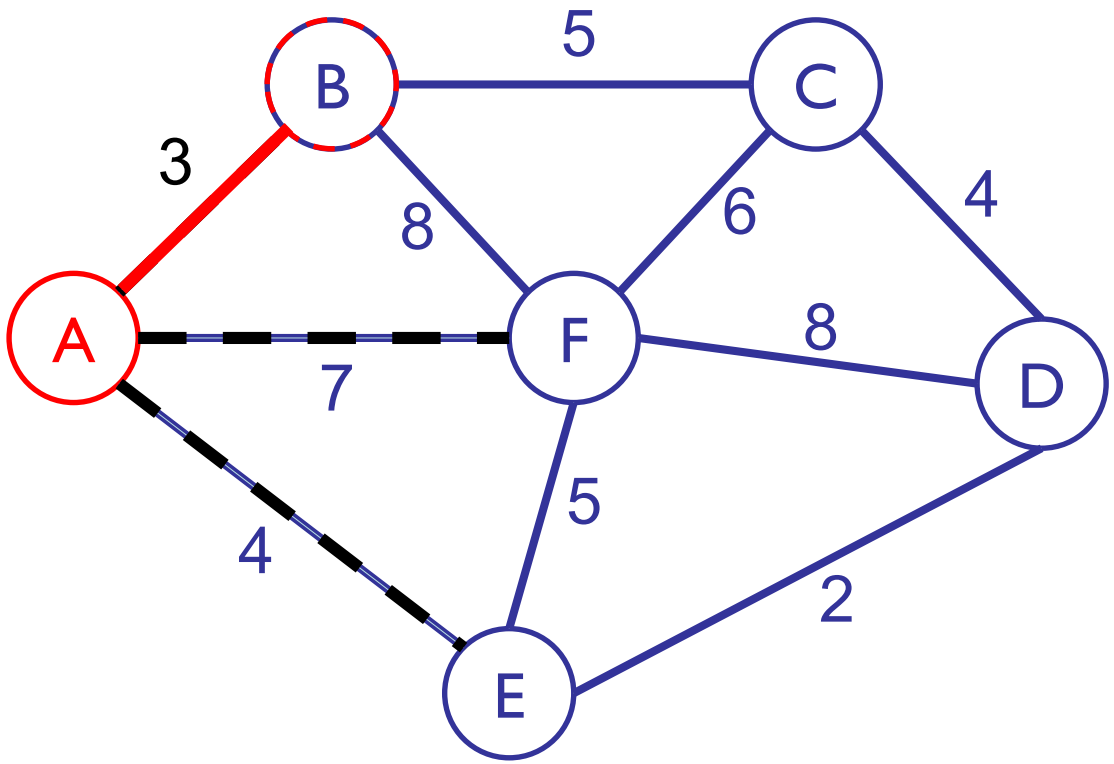
Priority Queue

A-B: 3
A-E: 4
A-F: 7

MCST

Visit A, then
Add all outgoing edges to priority queue

Prim's Algorithm



Priority Queue

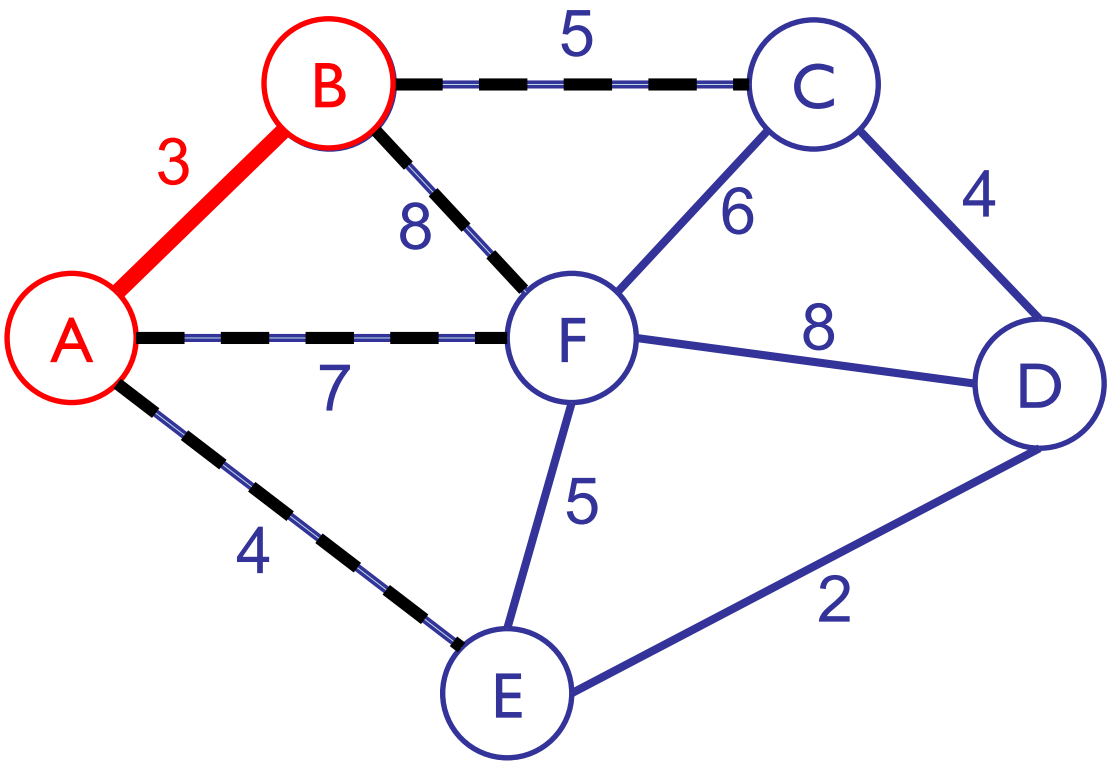
A-B: 3
A-E: 4
A-F: 7

MCST

A-B: 3

Pop elements from Priority queue until one takes us somewhere 'new'. Add to MCST

Prim's Algorithm



Visit B, then
Add all outgoing edges to priority queue

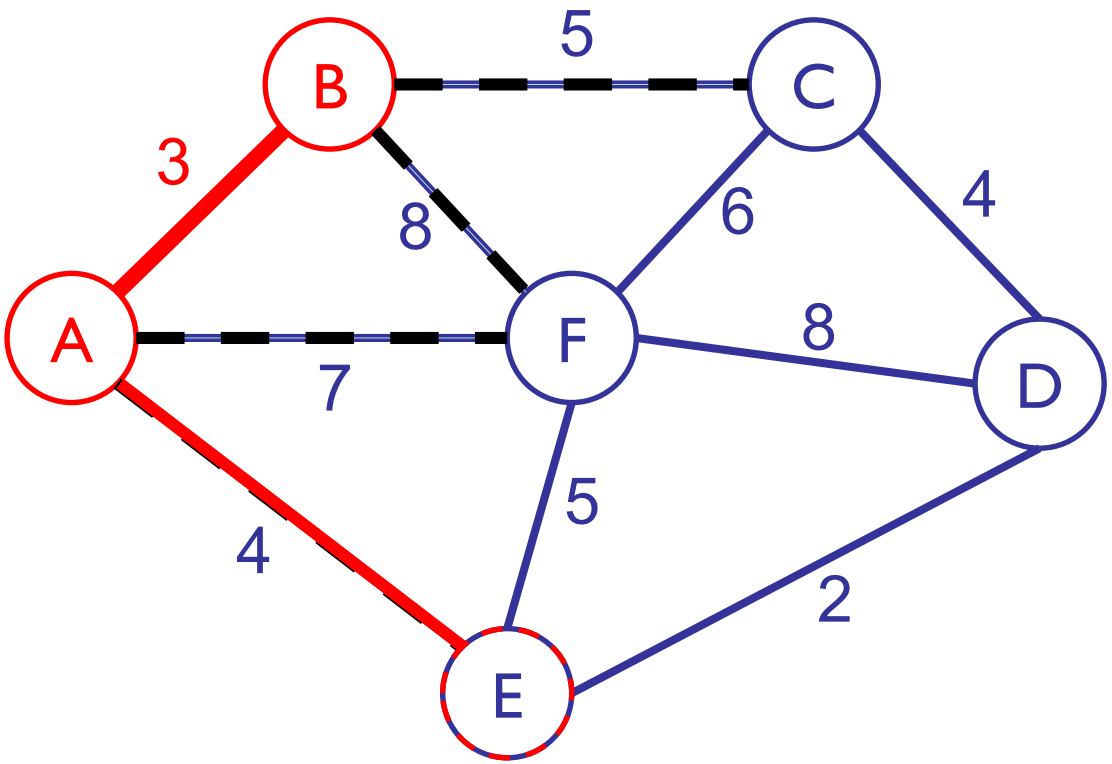
Priority Queue

A-E: 4
B-C: 5
A-F: 7
B-F: 8

MCST

A-B: 3

Prim's Algorithm



Priority Queue

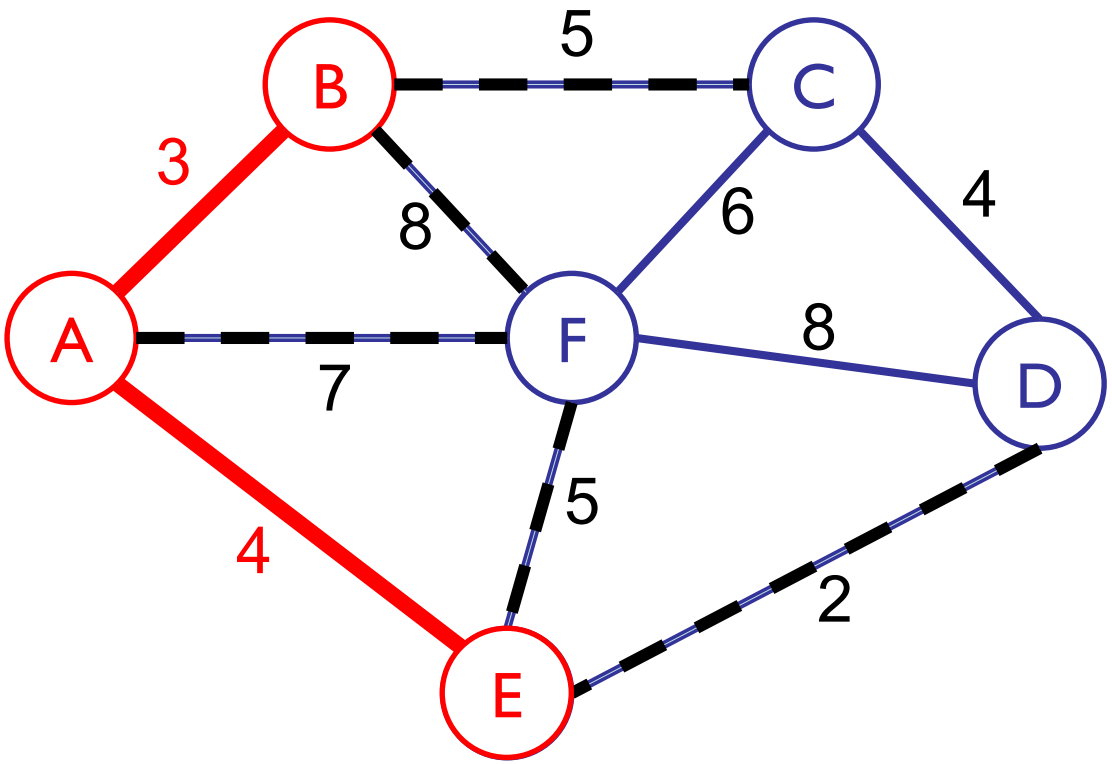
A-E: 4
B-C: 5
A-F: 7
B-F: 8

MCST

A-B: 3
A-E: 4

Pop elements from Priority queue until one takes us somewhere 'new'. Add to MCST

Prim's Algorithm



Visit E, then
Add all outgoing edges to priority queue

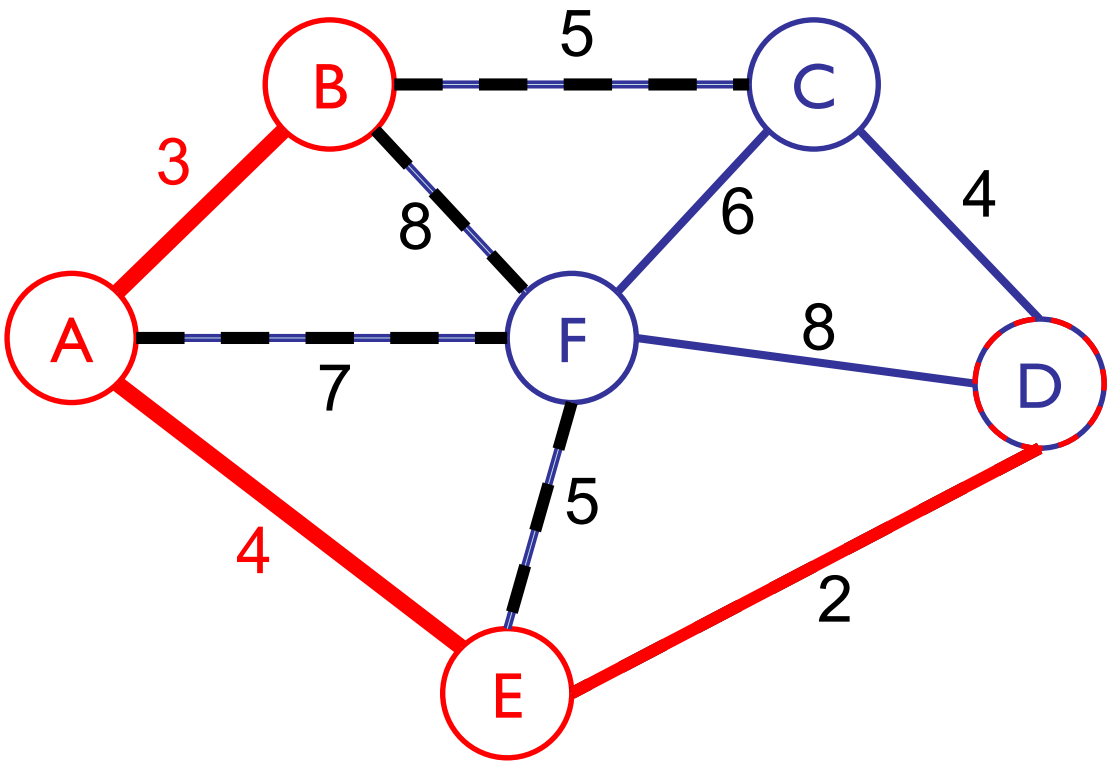
Priority Queue

E-D: 2
B-C: 5
E-F: 5
A-F: 7
B-F: 8

MCST

A-B: 3
A-E: 4

Prim's Algorithm



Priority Queue

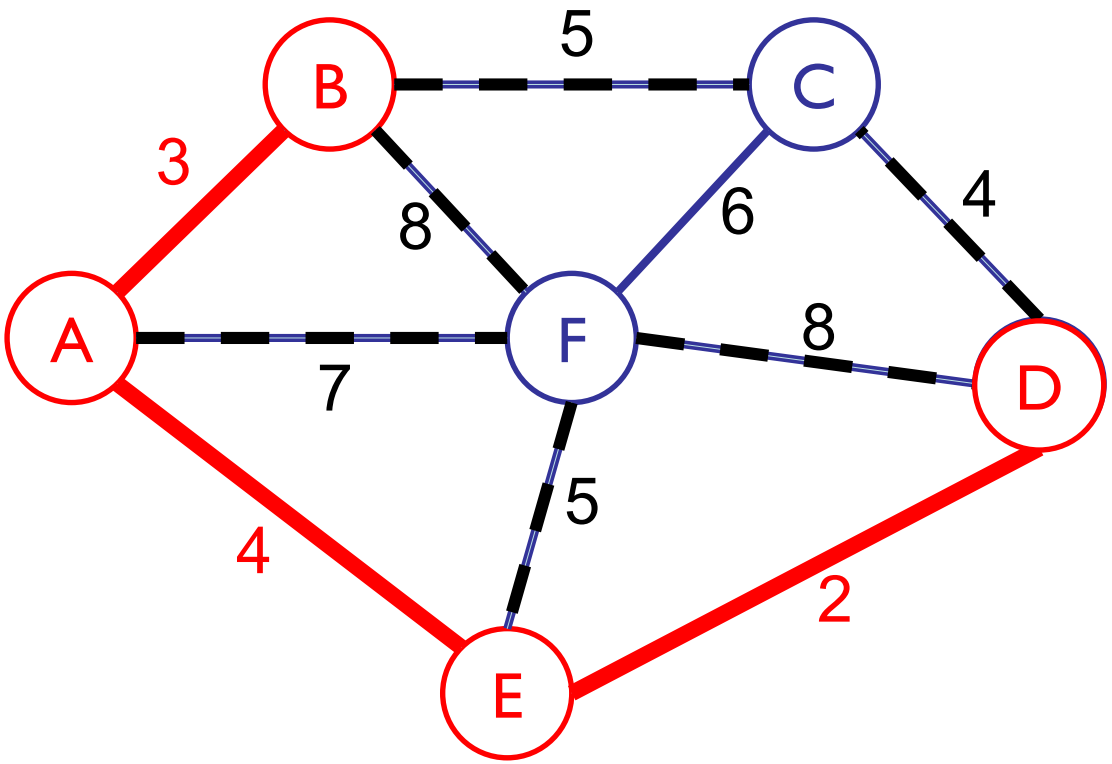
E-D: 2
B-C: 5
E-F: 5
A-F: 7
B-F: 8

MCST

A-B: 3
A-E: 4
E-D: 2

Pop elements from Priority queue until one takes us somewhere 'new'. Add to MCST

Prim's Algorithm



Visit D, then
Add all outgoing edges to priority queue

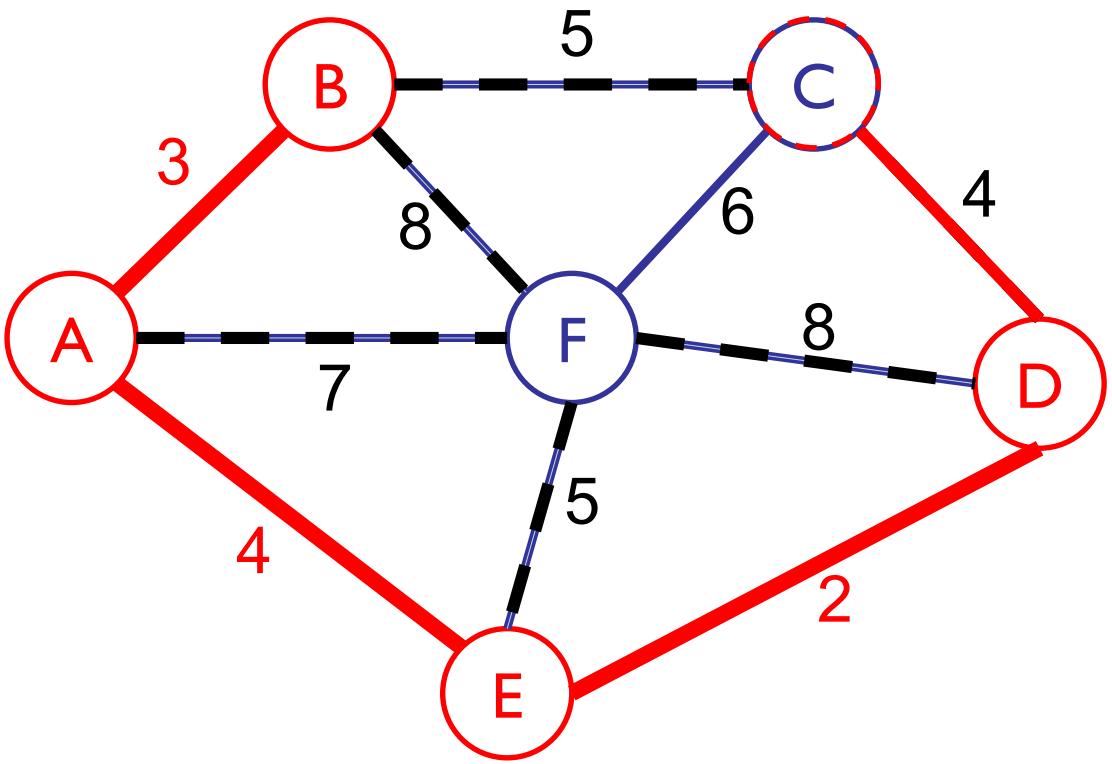
Priority Queue

D-C: 4
B-C: 5
E-F: 5
A-F: 7
B-F: 8
D-F: 8

MCST

A-B: 3
A-E: 4
E-D: 2

Prim's Algorithm



Pop elements from Priority queue until one takes us somewhere 'new'. Add to MCST

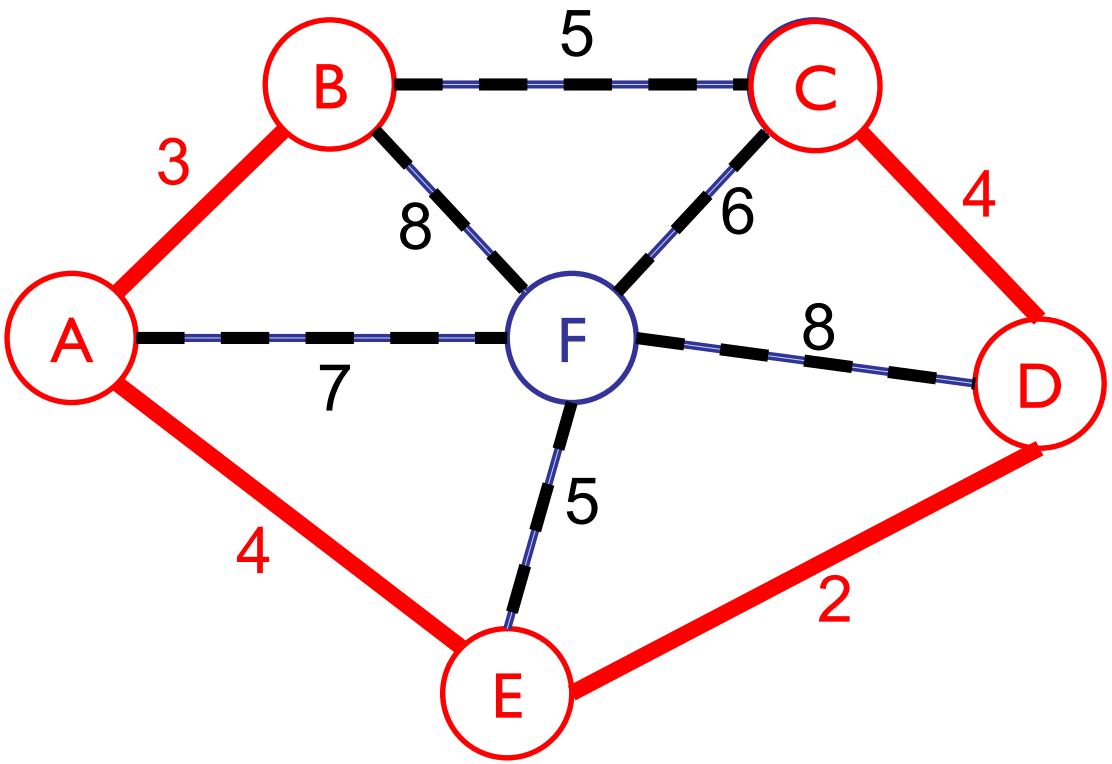
Priority Queue

D-C: 4
B-C: 5
E-F: 5
A-F: 7
B-F: 8
D-F: 8

MCST

A-B: 3
A-E: 4
E-D: 2
D-C: 4

Prim's Algorithm



Visit C, then
Add all outgoing edges to priority queue

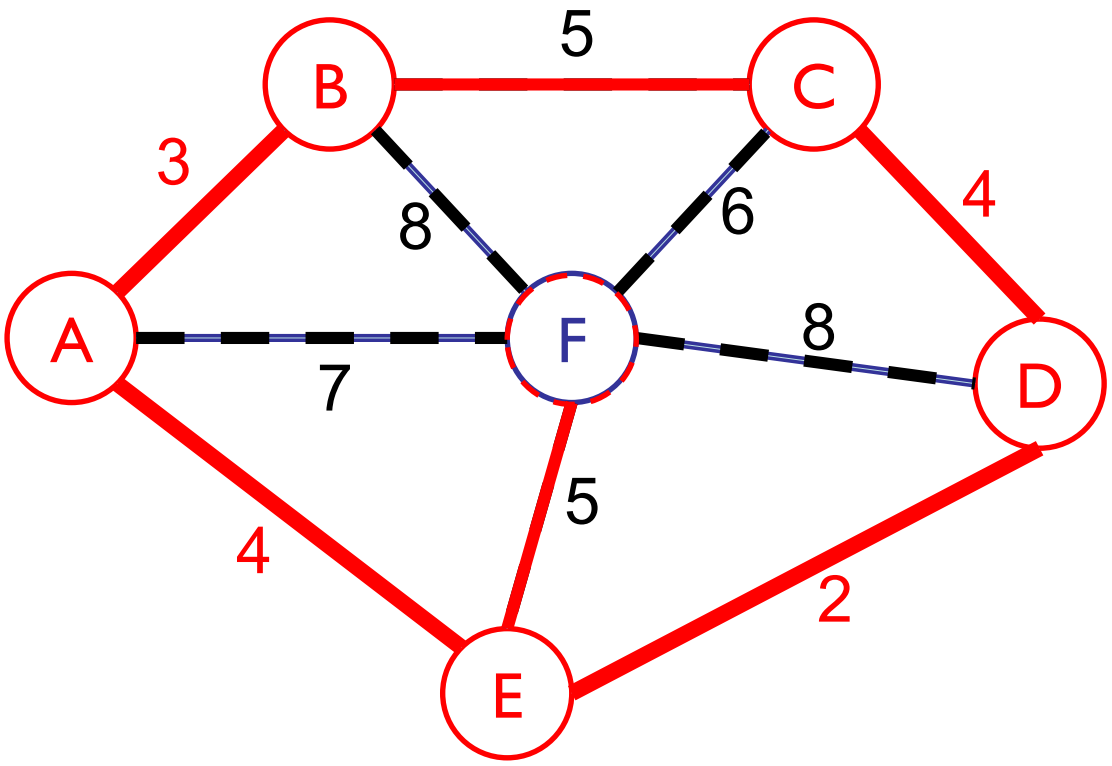
Priority Queue

B-C: 5
E-F: 5
C-F: 6
A-F: 7
B-F: 8
D-F: 8

MCST

A-B: 3
A-E: 4
E-D: 2
D-C: 4

Prim's Algorithm



Priority Queue

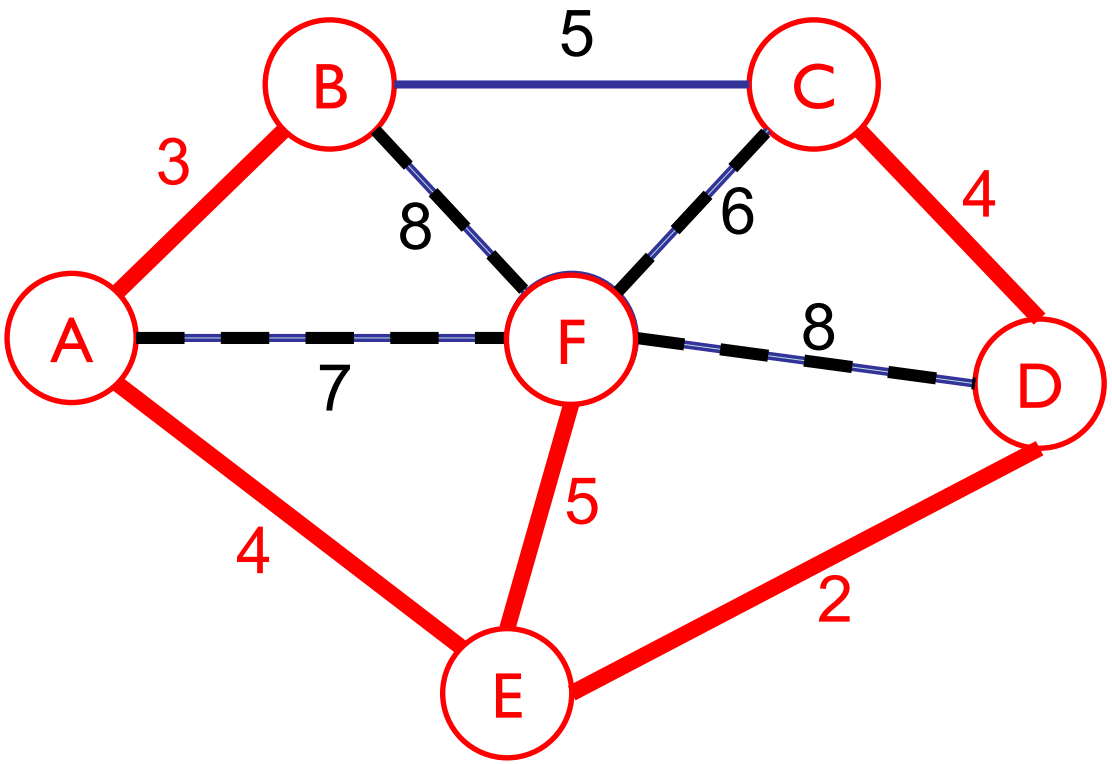
B-C: 5
E-F: 5
C-F: 6
A-F: 7
B-F: 8
D-F: 8

MCST

A-B: 3
A-E: 4
E-D: 2
D-C: 4

Pop elements from Priority queue until one takes us somewhere 'new'. Add to MCST

Prim's Algorithm



Visit F, then
Add all outgoing edges to priority queue

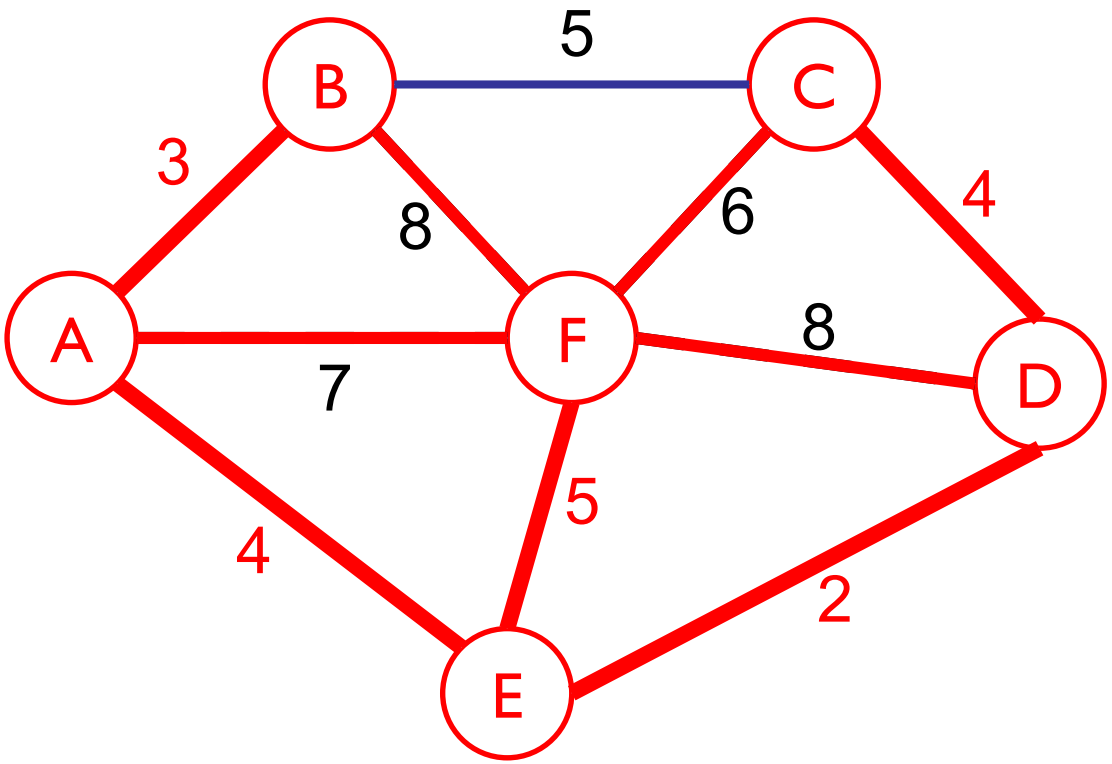
Priority Queue

C-F: 6
A-F: 7
B-F: 8
D-F: 8

MCST

A-B: 3
A-E: 4
E-D: 2
D-C: 4
E-F: 5

Prim's Algorithm



Priority Queue

C-F: 6
A-F: 7
B-F: 8
D-F: 8

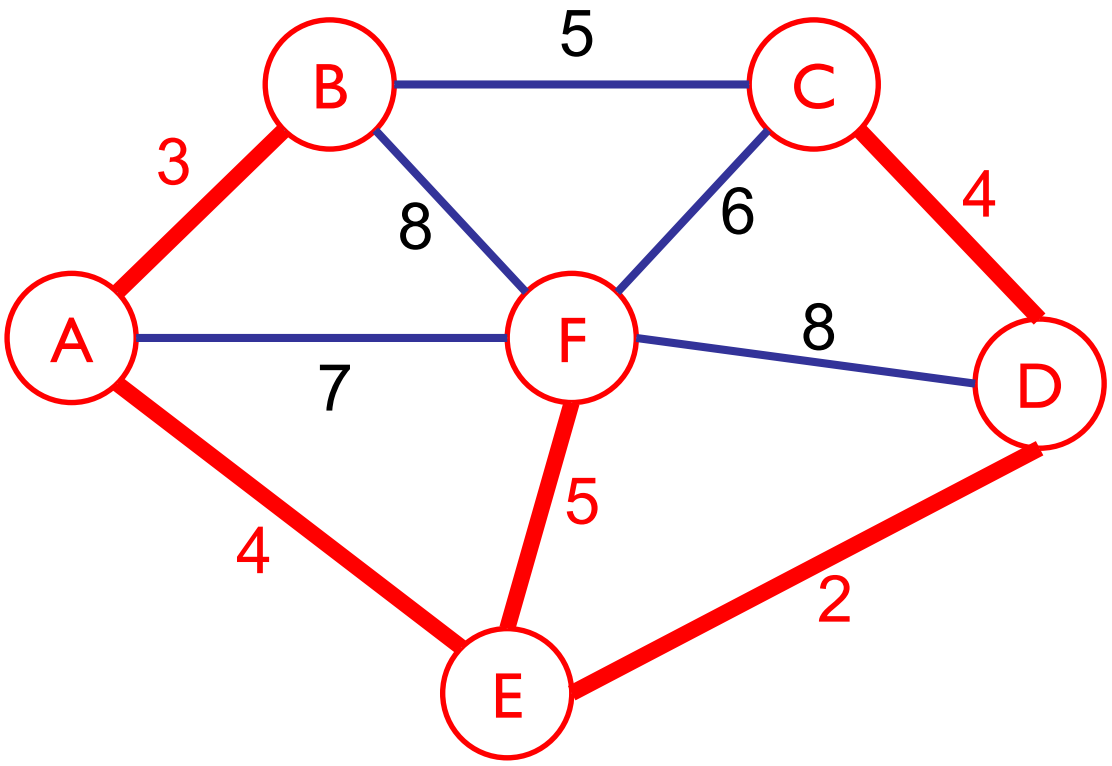
MCST

A-B: 3
A-E: 4
E-D: 2
D-C: 4
E-F: 5

Pop elements from Priority queue until one takes us somewhere 'new'.

Prim's Algorithm

Priority Queue



Priority queue is empty. We are done!

MCST

A-B: 3
A-E: 4
E-D: 2
D-C: 4
E-F: 5

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 p. queue: O(deg(v) log |E|)
  - Iterator operations are O(1)
  - Adding an edge to the priority queue is O(log |E|)
- Find next edge: O(# edges checked * log |E|)
  - Removing an edge from p. 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|

Maps and Hash Tables

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 values 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 dictionary needs?

`successor(key)`, `predecessor(key)`

These are very hard to implement in a hashtable!

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 *roughly* O(1) time!
    - (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

hashCode() rules

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()

Implementing HashTable

- How do we add objects to the array?
  - `int hash = Math.abs(o.hashCode());`
    `array[hash % array.length] = o;`
  - Does this always work?
- Collisions make life hard
- Two approaches
  - Linear probing (open addressing)
  - External chaining

Linear Probing

- Inserting: If a collision occurs at a given bin, just scan forward (linearly) until an empty slot is available
  - We will call a contiguous region of full bins a *run*
- If you are looking for a target KV-pair, scan linearly through the run until you find target or reach the end of the run
- Let's implement put(key, val) and get(key)…

First Attempt: put(K)

```
public V put (K key, V value) {
   int bin = key.hashCode() % data.length;
   while (true) {
      Association<K,V> slot = (Association<K,V>) data[bin];
      if (slot == null) {
         data[bin] = new Association<K,V>(key,value);
         return null;
      }
      if (slot.getKey().equals(key)) { // already exists!
         V old = slot.getValue();
         slot.setValue(value);
         return old;
      }
      bin = (bin + 1) % data.length;
   }
}
```

First Attempt: get(K)

```
public V get (K key) {
   int bin = key.hashCode() % data.length;
   while (true) {
      Association<K,V> slot = (Association<K,V>) data[bin];
      if (slot == null)
         return null;

      if (slot.getKey().equals(key))
         return slot.getValue();

      bin = (bin + 1) % data.length;
   }
}
```

Linear Probing

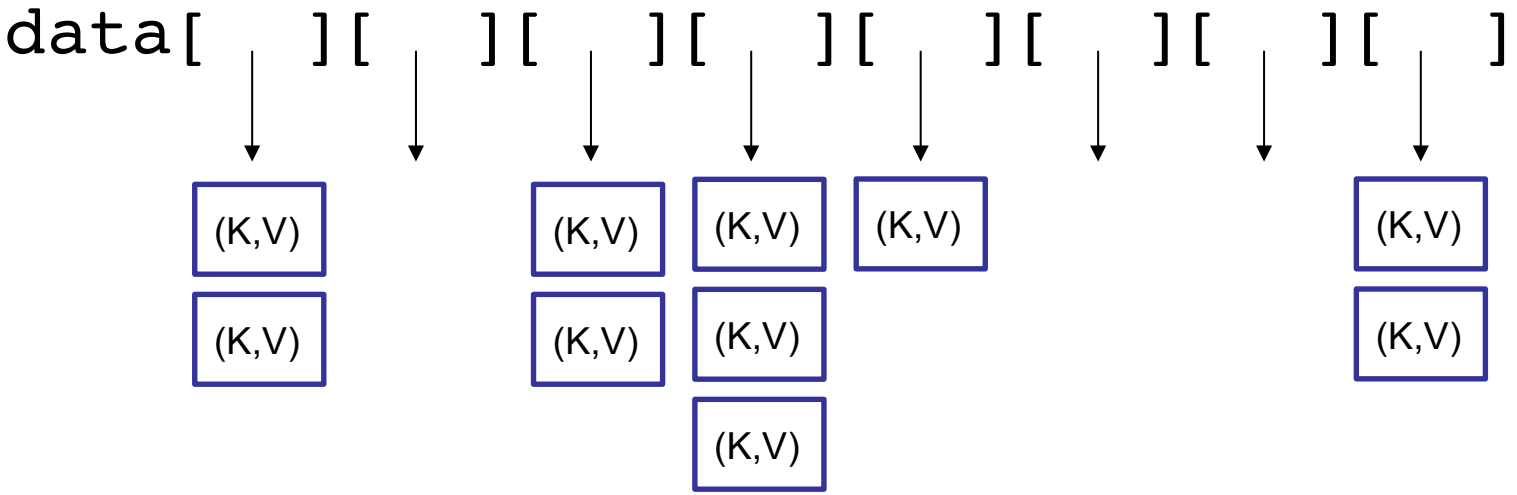
- In NaiveProbing.java, we:
  - Specify a dummy hash function: index of first letter of word
  - Set the initial array size = 8
  - Add "air hockey" to hash table
  - Add "doubles ping pong"
  - Add "quidditch"
- What happens when we remove "air hockey", and then lookup "quidditch"?
  - Our *run* was broken up!
  - We need a "placeholder" for removed values to preserve runs…
- See Hashtable.java in structure5

Linear Probing

- Downsides of linear probing?
  - What if array is almost full?
    - Loooong runs for every lookup…
    - Items out of place if we don't re-index after removing (placeholders are correct, but they defer work)
- How can we avoid these problems?
  - Keep all values that hash to same bin in a `Collection`
    - Usually a SLL
  - *External chaining* "chains" objects with the same hash value together

External Chaining

- Instead of runs, we store a list in each bin



- Everything that hashes to bin_i goes into $list_i$
  - get(), put(), and remove() only need to check one slot's list
  - No placeholders!

Probing vs. Chaining

What is the performance of:

- `put(K, V)`
  - LP: O(1 + run length)
  - EC: O(1 + chain length)
- `get(K)`
  - LP: O(1 + run length)
  - EC: O(1 + chain length)
- `remove(K)`
  - LP: O(1 + run length)
  - EC: O(1 + chain length)

- Runs/chains are important. Ho do we control cluster/chain length?