

**CSCI 136**  
**Data Structures &**  
**Advanced Programming**

**Hashtables & Collisions**

# Video Outline

- Hashtables
  - Recap “big picture”
- Collision resolution strategies
  - External chaining
  - Linear probing/open addressing

# Hash Table Implementation

**General idea:** Use an array to represent “bins”

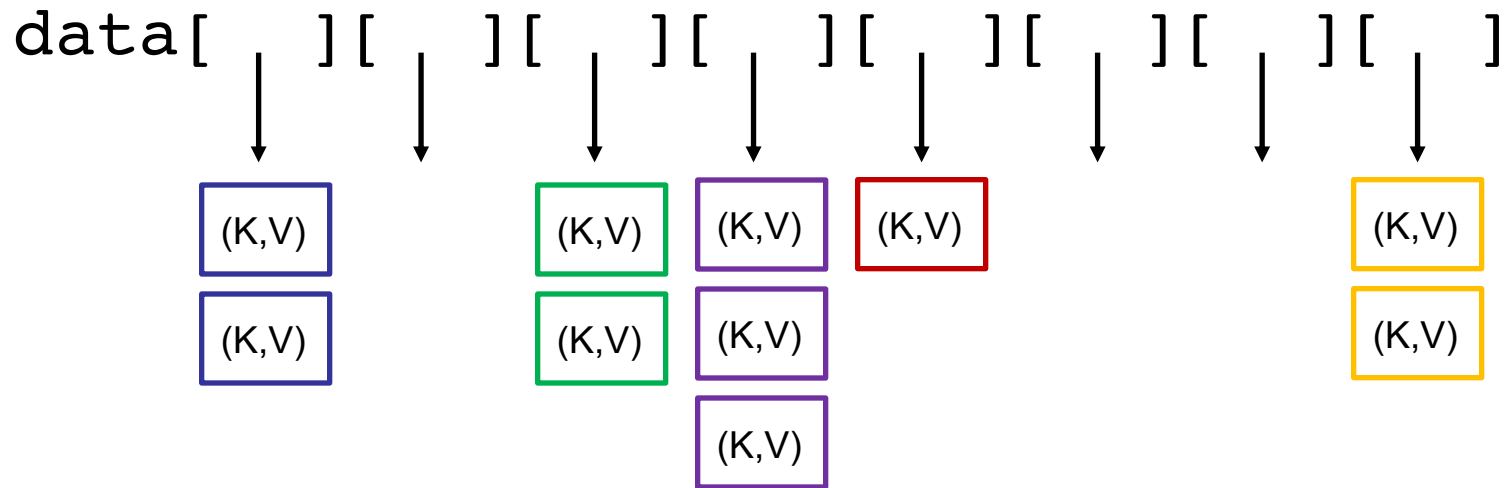
- **V get (K key):**
  - use key's hashcode to identify bin (% array length)
  - Search bin for item with matching key
- **V put (K key, V val):**
  - use key's hashcode to identify bin
  - Search bin for item with matching key:
    - If a match exists, replace old value with val
    - If no match exists, add new (key,value) pair

# Navigating HashTable Collisions

- **Problem:** *collisions* occur when two unique items are mapped to the same bin
  - This is a problem in arrays, because we can only store one item per index
  - Thus, collision management isn't *just* a performance issue, it is a correctness issue
- We'll discuss two strategies to resolve collisions
  - *Linear probing* (sometimes called open addressing)
  - *External chaining*

# Idea 1: External Chaining

- **Idea:** Instead of mapping individual items to bins, we store a *list* in each bin



- `get()`, `put()`, and `remove()`, then, need to (a) identify the bin, then (b) check bin's list

# Hash Table Implementation w/ External Chaining

```
public V get(K key) {
    int bin = key.hashCode() % table.length;
    // search for value in bin
    Association<K, V> temp = new Association<K,V>(key);
    Association<K, V> ret = table[bin].remove(temp);

    if (ret != null) { // if found, return value
        // restore value to bin so don't modify table
        table[bin].add(ret);
        // return the value we found
        return ret.getValue();
    }
    return null;
}
```

# Hash Table Implementation w/ External Chaining

```
public V put(K key, V val) {
    int bin = key.hashCode() % table.length;

    // search for old value in bin and remove if found
    Association<K, V> toAdd = new Association<>(key, val);
    Association<K, V> old = table[bin].remove(toAdd);

    // add our new K,V pair
    table[bin].add(toAdd);

    if (old != null) {
        // if old value found, return val we're replacing
        return old.getValue();
    }
    // not found, return null
    return null;
}
```

# Downsides to External Chaining

- Each slot in our Hashtable's array stores a list, even if the slot is empty
  - This consumes extra space
- Potentially poor *locality*
  - Not something we've talked about so far in this course, but a general rule of thumb: it is faster to access things that are near to each other than it is to access things that are far away.
    - Array elements are always contiguous (near)
    - List elements may be scattered throughout mem (far)



# Rethinking Collisions

- Let's define an item's **canonical slot** as the place where the item belongs *ignoring collisions*
  - If no two items map to the same canonical slot, we don't have any problems
  - If multiple items do map to the same canonical slot, we need to figure out:
    - Among the set of colliding items, which one belongs in the canonical slot
    - Where do the "losing" items belong so that we still can find them in the future?

# Linear Probing

- **General idea:** store each key-value pair in the first **open slot** on or after its **canonical slot**
- **Insertion:** If a collision occurs at a given bin, just scan forward (linearly) until an empty slot is available, and store it there
  - We “wrap around” at the end of the array
  - We will call a contiguous region of full bins a *run*
- **Lookup:** To find a KV-pair, scan linearly through the run until you find it or reach the end of the run
- Let’s take a first try at implementing `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) { // Found and empty bin!
            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 filled. Check the next bin...
        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) // Found an empty bin. End of the run
            return null;

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

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

# Linear Probing Gotchas

- Let's look at NaiveProbing.java
  - We specify a dummy hash function: index of first letter of word
  - Initial array size = 8
  - Add “atlanta” to hash table
  - Add “detroit”
  - Add “queens”
- What happens when we remove “atlanta”, and then lookup “queens”?
  - Our *run* was broken up!
  - We need a “placeholder” for removed values to preserve runs...

# Linear Probing Challenge

- When we delete an element from a **run**, we create a “hole”
  - **Challenge**: How do we tell if the run has ended, or if the hole is from a deletion?
  - **Solution**: Insert a “placeholder”
    - If we see the placeholder during a lookup, we treat it as a collision, and keep scanning until we find a true hole
    - If we see the placeholder during insertion, we treat it as an open spot
      - (We must still scan the whole run to see if our key is present)

# Hashtable.java

```
public class Hashtable<K,V> implements Map<K,V>, Iterable<V> {  
  
    /* A single key-value pair to be used as a token  
     * indicating a reserved location in the hashtable.  
     * Reserved locations are available for insertion,  
     * but cause collisions on lookup. */  
    protected static final String RESERVED = "RESERVED";  
  
    /* The data associated with the hashtable. */  
    protected Vector<HashAssociation<K,V>> data;
```

# Hashtable.java

```
protected int locate(K key) {
    // initial hash code
    int hash = Math.abs(key.hashCode() % data.size());
    // keep track of first unused slot, in case we need it
    int reservedSlot = -1;
    boolean foundReserved = false;
    while (data.get(hash) != null) {
        // loop until end of run OR find target key
        if (data.get(hash).reserved()) {
            // remember reserved slot if we fail to locate value
            if (!foundReserved) {
                reservedSlot = hash;
                foundReserved = true;
            }
        } else {
            // value located? return the index in table
            if (key.equals(data.get(hash).getKey())) return hash;
        }
        hash = (1+hash)%data.size();
    }
    // return first empty slot we encountered
    if (!foundReserved)
        return hash;
    else
        return reservedSlot;
}
```



# Hashtable.java

```
public V get(K key) {  
    // find bin where key lives (after resolving collisions)  
    int hash = locate(key);  
  
    // if the key is not found, the resulting location  
    // is either null or "RESERVED"  
    if (data.get(hash) == null ||  
        data.get(hash).reserved())  
        return null;  
  
    // key was found, so return associated value  
    return data.get(hash).getValue();  
}
```

# Hashtable.java

```
public V remove(K key) {
    // find bin where key lives (after resolving collisions)
    int hash = locate(key);

    // if the key is not found, the resulting location
    // is either null or "RESERVED"
    if (data.get(hash) == null ||
        data.get(hash).reserved())
        return null;

    // key was found, so remove, then return old value
    count--;
    V oldValue = data.get(hash).getValue();
    data.get(hash).reserve();
    return oldValue;
}
```

# Linear Probing Observations

- Code becomes more complicated, but manageable
- The length of a **run** dictates the performance
- Reserving elements does not “shrink” the **run**—it defers the work to other operations
  - Keeping our runs small is important, so we may want to reexamine design decisions if we expect a lot of deletions

# Linear Probing Observations

- Downsides of linear probing?
  - What if array is almost full?
    - Looooong runs for every lookup...
    - Items out of place if we don't re-index after removing (placeholders are correct, but they defer work)
- Does external chaining avoid these problems?
  - Recall, *External chaining* “groups” objects with the same hash value together in same bin in a `Collection` (usually a SLL)
    - Only scan collisions, not an entire run
    - Never scans more items than linear probing
    - Worse cache behavior (locality)

# Summary: Probing vs. Chaining

What is the performance of:

- $\text{put}(K, V)$ 
  - LP:  $O(l + \text{run length})$
  - EC:  $O(l + \text{chain length})$
- $\text{get}(K)$ 
  - LP:  $O(l + \text{run length})$
  - EC:  $O(l + \text{chain length})$
- $\text{remove}(K)$ 
  - LP:  $O(l + \text{run length})$
  - EC:  $O(l + \text{chain length})$
- Parting Question: how do we control cluster/chain length?