Resolving collisions

A collision happens when two keys hash to the same bucket, i.e., the same hash-value. What to do?

- **Separate chaining.** Make the hash table an array of linked lists.
- **“Open-addressing” hashing.** If the first spot is full, use another spot in the hash table.
  - linear probing: look in the next spot down/wrapped in the array.
  - “Quadratic” probing is also possible, but we won’t cover it. Weiss covers this case.

Separate Chaining: pp. 464-468

- Each hash array element is a linked list holding all the keys that hash to that bucket - all the collision participants. No further probing is needed, just list operations.
- This is the simplest way to make a hash table that works decently. Many hash tables work this way, including HashMap in the JDK.
- In some cases separate chaining may be a little slower than linear probing, because it causes memory references that hop around in memory more. This could be important for very large hash tables.
- In Java 8+, Hashmaps may switch a bucket from a linked list to a tree structure if the linked list gets too long. This mitigates bad hash function use.

Understanding Object Graphs

- This represents one SequentialSearchST (pg. 375) object.
- first: the instance variable of SequentialSearchST object that references the first Node object for the list.
- The box with A and 8 in little boxes: this represents one Node object for the list, with its instance variables key = “A” and val = 8. Also the arrow out of this box represents the ref called “next” of the Node object.
- The Node with E and 12 has no arrow out because its next = null, the end of the list.

public class SeparateChainingHashST<Key,Value> {
    private int m; // HT size
    private SequentialSearchST<Key,Value>[] st; // HT
    // ... in constructor, create the hash table: needs
    @SuppressWarnings({"rawtype", "unchecked"})
    st = [SequentialSearchST<Key,Value>[] new
    Array of
    Raw
    Generic type
    Array of
    Raw
    Generic type
    
- We can't directly create an array of generic type in Java. So we create this array of raw type and cast it to the generic type, which causes warnings. We can suppress those warnings.
- Very ugly. Same ugliness shows up in JDK HashMap code. Can’t be avoided if you want the result to be a generic type, as we definitely do.
Hashing with separate chaining: alternatively using array of LinkedList

```java
public class SeparateChainingHashMap<Key, Value> {
    private int m;  // HT size
    private List<Map.Entry<Key, Value>>[] st; // HT
    // ... in constructor, create the hash table: needs
    @SuppressWarnings({"rawtype", "unchecked"})
    st = (List<Map.Entry<Key, Value>>[]) new LinkedList[m];
    // Again, we can't directly create an array of generic type in Java. So we
    // create the array of raw type and cast it to the generic type, which causes
    // warnings. We can suppress those warnings. Very ugly.
```

S&W Hashing with separate chaining: code, pg. 465

```java
public class SeparateChainingHashMap<Key, Value> {
    private int m;  // HT size
    private SequentialSearchST<Key, Value>[] st; // HT
    // ... in constructor, create the hash table: needs
    @SuppressWarnings({"rawtype", "unchecked"})
    st = (SequentialSearchST<Key, Value>[]) new SequentialSearchST[m];
    // Now create the actual lists in the buckets--
    for (int i=0; i < m; i++)
        st[i] = new SequentialSearchST<Key, Value>();
    // Note that the book code has the raw type here, unnecessary. No array
```

Linear Probing: pp. 469-474

- If bucket b = h(x) is already in use, try b+1, then b+2, etc.,
  wrapping around to b = 0 if you hit M, the table size.
  - b = b+1 % M.
- As soon as you find an empty spot, take it.
- On lookup, hash the key and check if it matches the one in the
  hash slot, if not, try the next (wrapping if necessary), etc.,
  until you find the matching one or an empty one.
- The performance can suffer because stretches of the array get
  filled up and probes have to go further and further. Need to resize
  aggressively.
- Also, you can't delete entries in a simple way, just removing them,
  because they have been used as stepping stones in other key's
  probing sequences.

As on pg. 469, trace of HT load
“Open Addressing” HTs
LinearProbingHashST (pg. 470)

- Uses two parallel arrays, one for keys, one for values
  - Alternatively, could use array of Map.Entry, but this setup avoids the array of generics mess.
- So an entry stored at bucket 123 has key in keys[123], value in values[123].
- put(key, val) works like this:
  - It hashes the key to say bucket 55, then looks in keys[55]. If that is null, look no further. If that equals key, put val in values[55] and return, else step to bucket 56 and handle similarly.
  - If reach end of table, start over at bucket 0.
- End up with null spot if haven’t yet returned: put key and val at that bucket and return.
- delete(key) is tricky—see pg. 471

Resizing the hash table to keep it not too full.

- It is important to keep hash tables less than half full (or 75% full) for good performance.
- This can be quite critical for performance.
- With Java, it’s particularly easy to resize a hash table because you don’t have to carefully dismantle the old one—you can point to a new table and just leave the old table to the garbage collector to clean up.

Performance of Resizing

- Typically, we resize every time the number of entries exceeds 2n, starting at the original half-full point.
- Consider doing a sequence of inserts to an empty hash table of size 64+, prime (M = first prime over 64, that would be 67):
  - insert 1, 2, 3, ..., 31 – still less than half full.
  - insert 32—first resize, to M=128+, prime
  - insert 33, 34, 35, ..., 63 – still less than half full (32 = 2^5 ops)
  - insert 64: second resize, to M=256+, prime (64 = 2^6ops)

Intro to inner classes

- In general, during inserts 2n to 2n+1, we do 2n simple inserts, each O(1), plus 2n+1 relocations into a new hash table, each also O(1).
- This totals O(n) inserts.
- If we amortize this over 2n inserts, that is, average the total time over the total number of inserts between rehashes, 2n, we get O(1) per insert.
- Naturally there’s a “hiccup” when the expansion happens, and this can be unacceptable if it exceeds some threshold time such as human-perceivable time in interactive programs.
- There are more sophisticated “extensible hashing” methods that don’t hiccup so badly.

Resizing in LinearProbingHashST<Key, Value>

```java
private void resize(int capacity) {
    LinearProbingHashST<Key, Value> temp = new LinearProbingHashST<Key, Value>(capacity);
    for (int i = 0; i < m; i++) {
        if (keys[i] != null) {
            temp.put(keys[i], vals[i]);
        }
    }
    keys = temp.keys;
    vals = temp.vals;
    m = temp.m; // old value of m is GC'd
}
```

Performance of Resizing

- Typically, we resize every time the number of entries exceeds 2n, starting at the original half-full point.
- Consider doing a sequence of inserts to an empty hash table of size 64+, prime (M = first prime over 64, that would be 67):
  - insert 1, 2, 3, ..., 31 – still less than half full.
  - insert 32—first resize, to M=128+, prime
  - insert 33, 34, 35, ..., 63 – still less than half full (32 = 2^5 ops)
  - insert 64: second resize, to M=256+, prime (64 = 2^6ops)

Intro to inner classes

- Starting with Nodes for linked lists in S&W Sec. 1.3, we have seen private class definitions inside top-level classes.
- This kind of class with definition inside another class is called an inner class. There are two kinds of inner classes in Java:
  1. static inner classes, AKA-nested classes
  2. non-static inner classes, or plain “inner classes”
- S&W only uses inner classes (not static), and mostly so will we.
- Both kinds of inner classes can be used like ordinary classes.
- They can be hidden from client code with “private class…”.
- They are both used mainly for implementation details of the outer object.
public class Stack<Item> implements Iterable<Item> {
    ...
    private class Node {
        Item item;
        Node next;
    }
}

• This inner class inside Stack.java doesn't bother with “private Item item;” or “private Node next;” because the whole class is private inside Stack.
• And the variables of the inner class are always accessible to the outer class's code anyway.
• S&W doesn't discuss inner classes much, just a little note on pg. 159.

Other code in Stack.java using Nodes:
public void push(Item item) {
    Node oldfirst = first;
    first = new Node();
    first.item = item; // access to field within Node
    first.next = oldfirst;
    n++;
}

We don't usually “dot into” objects like this, but this Node object is a private construct of this class.

Inner class code's access to outer class's variables

• Code in an inner class (but not in a static inner class) can use the outer class’s instance variables.
• This is important in implementing iterators.
• See code on pg. 155 inside class Bag<Item>:
  private class ListIterator implements Iterator<Item> {
      private Node current = first;
      ...
  }

Note on inner classes (not static)

• An Inner class object is bound to its outer class object.
  • It contains a ref to its outer object.
  • So the outer object can't be garbage-collected as long as the inner class object is in use.
• Their creation always involves an outer object, and because of the close association, they know their outer object at any time.
• Their code can use the generic type parameter(s) of their outer class (here Item of Bag<Item>, pg. 155)
private class ListIterator implements Iterator<Item> {
    ...
    Item item = ...
}

JDK HashMap implementation

• The Java HashMap uses an array of Nodes, each with key, value, and Node implements Map.Entry.
• The Node for a hash bucket becomes the first Node of a singly-linked collision list, so they each have a next ref.
• The creation of the array has the same generic-type array problem as we saw with SeparateChainingST, with @SuppressWarnings("rawtypes","unchecked") to keep from showing a warning to users when the needed cast happens.
• The Java 8 HashMap also has a way to switch from using this linked collision list to a tree structure similar to the JDK TreeMap, to handle cases where the hash function sends too many entries into one hash bucket.

Hashing in Memory and on Disk

• The HashMap hash table is located in memory, supporting fast lookup to records in memory or even on disk.
• Some hash tables are on disk, supporting fast access to further disk.
• In fact, a disk-resident hash table that is in frequent use ends up being in memory because of the memory “caching” of disk pages in the file system.

<table>
<thead>
<tr>
<th>Type</th>
<th>hash table</th>
<th>Data records</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>memory</td>
<td>memory</td>
<td>typical HashMap app</td>
</tr>
<tr>
<td>memory</td>
<td>disk</td>
<td>disk</td>
<td>use HashMap to hold disk records locally, as interface</td>
</tr>
<tr>
<td>memory</td>
<td>disk</td>
<td>hash file, some database tables</td>
<td></td>
</tr>
</tbody>
</table>
Hashing packages in programming language libraries.

- The C library sticks to basics and has no collection support.
- We’ve looked at the Java Collections HashMap and HashSet.
- Python’s dictionaries and PHP’s associate arrays use hashing.
- The STL (Standard Template Library) of C++ provides Maps, but their performance (HP implementation) is logarithmic, and thus indicates they are implemented with trees, not hash tables.
- Microsoft’s .NET/C# provides collection classes

Set operations

- Consider set of integers: A = {1, 5, 3, 96}, or B = {17, 5, 1, 96} and a set of strings, (“Mary”, “contrary”, “quite”).
- Sets have no duplicates and order doesn’t count. (vs. List allows duplicates and order does count.)
- The Set interface contains just the basic set operations. We would like to use Union, Intersection, and Difference.

Basic set operations - reminder

\[ A \cup B = \{1, 3, 5, 17, 96\} \]

\[ A \cap B = \{1, 3, 5\} \]

A-B = \{3\}, B-A = \{17\}

A is contained in B

From Wikipedia

Set operation examples

- Consider Sets made with HashSet:
  - Set a = new HashSet(); ...
  - Union of Set a and Set b: we can “addAll” of b to a or vice versa, to form the union set:
    - a.addAll(b) // a is the union of a and b
    - b.addAll(a) // turns b into the union
  - If we don’t want to change a or b, make a new set from one of them first:
    - Set union = new HashSet(a);
    - union.addAll(b); // form a UNION b in own set
  - For subset containment, we just use containsAll:
    - a.containsAll(b); //returns true if b is a subset of a

Intersection example

```
a.retainAll(b);
// turns a into intersection of a and b
b.retainAll(a);
// turns b into intersection
Set intersection = new HashSet<T>(a);
intersection.retainAll(b);
// form a INTERSECT b in own set
```

Difference example

```
Notice that A – B and B – A are different.
a.removeAll(b);
// turns a into a – b
b.removeAll(a);
// turns b into b – a
Set<T> diffAB = new HashSet<T>(a);
diffAB.removeAll(b);
// form a – b in own set
Set<T> diffBA = new HashSet<T>(b);
diffBA.removeAll(a);
// form b – a in own set
```
Implementation of union, intersection

UNION via `a.addAll(b)`:
Iterate through b, checking elements for membership "contains()" in a, an O(1) op. Any that aren't there, add (by ref copy, as usual) to result set, also O(1). For N elements of B, O(N) operation.

INTERSECTION via `a.retainAll(b)`:
Iterate through a, checking elements for membership in b. If not in b, remove it (O(1)); For N elements of A, an O(N) operation.

How does it look with the element objects?

- In Java we allow many ref's to one object, and when we call "new HashSet(a)" we are not copying the element objects but just their refs, a "shallow" copy.
- Then the addAll copies more element refs to the new set.
- Element matching is done via element.equals().
- And if we're using HashSet, we need a consistent hashCode for the elements as well.

Implementation of difference

• For `a.removeAll(b)` it is possible to be a little smarter.
  if (a.size() < b.size())
    iterate through a, removing (from a) those that are in b
  else
    iterate through b, removing those from a
• This way, only the smaller set is scanned.

How does it look with the element objects?

Disjoint sets and their union:

A={1,5}  B={17,96}

AUB={1,5,17,96}

How does it look with the element objects?

Sets with shared elements and their union:

A={1,5}  B={1,17,96}

AUB={1,5,17,96}

How does it look with the element objects?

Sets with equal but not shared elements and their union:

A={1,5}  B={1,17,96}

AUB={1,5,17,96}
Toy example revisited: vowels

- The original pseudocode said
  ```java
  if (c is a vowel)
  count it
  ```

- How did we code the condition here?
  ```java
  String vowelStr = "aeiou";
  Set vowels = new HashSet();
  for (int i=0;i<vowelStr.length(); i++)
      vowels.add(new Character(vowelStr.charAt(i)));
  ```

- Later: // c a char: O(1) lookup
  ```java
  if (vowels.contains(c))
  count++;
  ```

Toy example revisited: vowels

- Of course for 5 items this is not important, but for large sets, it really matters.
- We have gone from an O(N) search to an O(1) lookup, a big savings.
- So when you see a loop of tests, think "could I use a Set here?"
- We just saw how set-thinking allowed us to replace an O(N) search with an O(1) lookup when we needed to classify something.
- But to see the full pay-off of set-thinking we need to use the set-crunching operations of union, intersection, etc.

Example: monitoring users

- Suppose we have 500 office connections, or "lines", each line has a Set of usernames seen on the line, so this defines 500 Sets of usernames.
- For example, the set of usernames (Strings) seen on line 1, set of users seen on line 2, etc.
- We could hold this all in an Array<Set<String>> if we wanted.
- Or in an array of Sets<String>, using the trick of casting from raw type to generic type:
  ```java
  Set<String> lines[] =
      new Set<String>[HashSet][500];
  ```

Example: monitoring users

- We have 500 sets of String usernames: users on line 1, users on line 2, etc.
- How do you find all users ever seen on any line?
  - Answer: take the union of all the sets.
- How do you find if anyone (or several users) logged in on all lines?
- Suppose one line is for a public station suspected of break-in attempts. The set of users who logged in on that line are considered suspicious. How do you find all the lines that any of these users logged in on?

Application idea – movie rentals

- Netflix and other webapps offer movies for rent. Thinking of new features…
- Suppose we get our patrons to rate movies they have just seen: buttons for "liked movie" and "disliked movie".
- Then our webapp could enter this data into the customer movie database. Movies have well-defined identifiers, movie titles, or title and year if needed.
- We want our webapp to output suggestions, based on our like-dislike data from our own customers, or choose the next movie for binge-watching.

Application – movie rentals

- After a while we would have access to the movie like-sets and dislike sets of our customers.
- This is now happening in a lot of industries, and is summarized by the phrase "market of one" data.
- Looking from the movie angle, we have sets of customers who liked it and disliked it.
- The simplest analysis is simple movie popularity:
  ```java
  popularity = Size(likeset)/[Size(likeset) + Size(dislikeset)].
  ```
Application – movie rentals

- But what about these predictions from X to Y?
- In the population, some people liked both X and Y, some liked X only and some liked Y only and some disliked both. If the first and last of these are relatively large compared to the middle ones, we have correlation between them: they tend to go together, like or dislike.
- But we don’t actually care about the dislike X case for our predictions. We are interested in all the cases where people liked X—then what about Y? The chance that such a person will like Y is (statistically)

\[ \text{probability(like Y | like X)} = \frac{\text{Size(like(X)} \cap \text{like(Y)})}{\text{Size(like(X)}}) \]

Application – movie rentals

- After a while our customers think we’re great and really come to depend on our predictions. Then they become more predictable themselves.
- Although it is impossible to keep enough movies on hand to keep everything available, we now have a notion of a good substitute:
  - If X predicts Y, then Y is a good substitute for X. Each movie has a substitute list based on some predictability level we’ve established.

How do we implement this?

- Each movie has a substitute set. movie → {substitutes} i.e., a Map.
- We also have movie qoh, quantity on-hand, its inventory. We can combine these into one master Map, using notation (x,y) for an object with fields x,y:
  
  
  movie → {qoh, {substitutes}}

- Here the domain element is a movie, id’d by String movie title. The range element is an object with an int qoh and a Set of movies, i.e., Set of Strings.
- Pretty easy to work with since String comes with equals and hashCode, ready for HashMap.
- Implementation: HashMap<String, InventoryData> where InventoryData is a class which has a Set<String> as well as an int "qoh".