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.

Separate Chaining Data Structure from S&W website and pg. 464

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.

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

```java
public class SeparateChainingHashST<Key,Value> { 
    private int m;  // HT size 
    private SequentialSearchST<Key, Value>[][] st; // HT 
    // ... see pg. 375: singly linked list 
    // ... in constructor, create the hash table: needs @SuppressWarnings({"rawtype","unchecked"})
    st = (SequentialSearchST<Key,Value>[][]) new 
    // Array of Node 
    // We can handle a chain of any 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. 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:
    // @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 SeparateChainingHashST<Key, Value> {
    private int m;  // HT size
    private SequentialSearchST<Key, Value>[] st; // HT
    // ... in constructor, create the hash table:
    // @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
}
```

Hashing with separate chaining: using array of LinkedList

```java
public class SeparateChainingHashST<Key, Value> {
    // instance variables, constructor
    private int hash(Key key) {
        return (key.hashCode() & 0x7fffffff) % m;
    }
    public Value get(Key key) {
        return (Value) st[hash(key)].get(key);
    }
    public void put(Key key, Value val) {
        st[hash(key)].put(key, val);
    }
    // ...Note hash(key) is a private helper method
    // Get and put both hash the key, just that hashval to locate the bucket in the hashtable,
    // and then the bucket’s own ST/map, ready for get or put.
    // 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
}
```

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.

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

```java
public class SeparateChainingHashST<Key, Value> {
    private int m;  // HT size
    private SequentialSearchST<Key, Value>[] st; // HT
    // ... in constructor, create the hash table:
    // @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
```
“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.

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
}
```

- Here we create a T within a method of T.java (outside the constructor of course)
- We are using Java’s garbage collection (GC) to clean up the old data structure when we overwrite m, the master HT variable.

Performance of Resizing

- Typically, we resize every time the number of entries exceeds $2^n$, 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 ($64 = 2^6$ops)

- 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 “extendible hashing” methods that don’t hiccup so badly.

Performance of Resizing

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.
S&W’s first inner class: Stack.java, pg. 149

```java
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:
```java
public void push(Item item) {
    Node oldfirst = first;
    first = new Node();
    first.item = item;  // access to item 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>:
  ```java
  private class ListIterator implements Iterator<Item> {
      private Node current = first;
      ...
  }
  ```

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

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)

```java
private class ListIterator implements Iterator<Item> {
    private Node current = first;
    ...
    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 SeparateChainingHashST, with @SuppressWarnings({“rawtypes”, “unchecked”})
- 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.
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