Class 6: hash functions

- A hash function accepts a key and returns an integer value for it, which is used to place the key and its value in the hash table by computing a bucket number, `bucketNo`, that is, an integer between 0 and M-1 where the hash table is of size M.
- The notation `h(x)` is used in S&W for a hash function that directly returns a bucket number for key `x`. So `bucketNo = h(x)`.
- But note that Java `Object.hashCode()` method may return an integer of any size, so that value needs to be further processed to determine a `bucketNo` for `HashMap` (project 2).

hash functions for HashMap

- Java `Object.hashCode()` method may return an integer of any size, so it needs to be further processed to specify a `bucketNo`.
- The JDK `HashMap` code (project 2) computes a bucket number from the hash by a formula equivalent to `bucketNo = hash % M` (where M is a power of 2).
- Because M is a power of 2, say $2^{10} = 1024$, this formula just extracts the low 10 bits of `hash` to specify `bucketNo`.
- The actual formula is $i = (n-1) & hash$, where for $n=2^{10}$, $n-1 = 1111111110 \ (10 \ 1's)$, which also just extracts the low 10 bits of `hash` for `i`, the bucketNo.

Resolving collisions in hash tables

- 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. This is what `HashMap` uses.
  - "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. pa2: use `ArrayList<Node>` for the collision list. Node holds a key-value pair.
- 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, the full 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. pa2: this coding has been removed.

Separate Chaining Data Structure

For pa2 picture replace the SSSTs with little ArrayLists, one for each bucket.
• First add: key S, value 0, hash 2, find HT[2], list, empty, add S-0
• 2nd add: key E, val 1, hash 0, find HT[0], list, empty, add E-1
• 4th add: key R, val 3, hash 4, find HT[4], empty, add R-3
• 5th add: key C, val 4, hash 4, push C-4 onto H[4] list, now has 3 elements
• 7th add: key E, val 6, hash 0, replace E's value with this one, note ST behavior

• Each list is type SequentialSearchST, meaning it's meant to be an ST, i.e. a Map: what's the domain and range?
• Keys are strings, like 'A', values are numbers, like 8, so 'A' \rightarrow 8 in the first SSST, etc.
• So we're making a big ST out of an array of little STs, each implemented by a singly-linked list. We're using the hash function to assign elements pseudo-randomly to the little STs.
• Pa2: we're making a big ST out of an array of lists, so we need to iterate through the collision list to locate the key

Implementation of Separate Chaining for ST

• This S&W design of disguising linked lists as little Maps makes for neat-as-a-pin code, but you need to remember that the little st.get has to loop down the list.
• Pa2: we do have to loop down the collision list.
• Well, there is a really ugly part needed to set up the needed hash table array, as we will see

Java code for Separate Chaining HashST

```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 SequentialSearchST[m];
    // We can't directly create an array of generic type in Java. So we create an 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.
    Pa2: Has this ugliness too, see next slide.
```
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.

### Hashing with separate chaining: using array of LinkedList

```java
class SeparateChainingHashMap<Key, Value> {
    private int m; // HT size
    private LinkedList<Map.Entry<Key, Value>>[] st; // HT
    // … in constructor, create the hash table: needs @SuppressedWarnings("rawtype", "unchecked")
    st = new LinkedList[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, unnecessarily. No array type here to cause problems. Works either way.
}
```

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

```java
class SeparateChaining HashMap<Key, Value> {
    private int m; // HT size
    private SequentialSearchST<Key, Value>[] st; // HT
    // … in constructor, create the hash table: needs @SuppressedWarnings("rawtype", "unchecked")
    st = 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, unnecessarily. No array type here to cause problems. Works either way.
}
```

### 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.

FYI: Towards understanding Generic types and arrays

- A generic class like HashTable<String> doesn’t have its own source code (i.e. code for the HashTable of String class).
- Instead, it uses the common HashTable class (the “raw type”) that accepts elements of all object types to an object reference.
  - The HashTable uses raw element (as only Java can do) using HashCode is an Object method.
- Java arrays are typed in a way so that if it is a subtype of , the array of it is a subtype of .
- The array type handles the element type in a inheritance-sensitive way. Ref.
- That means an array element isn’t directly be a generic type object, which dissolves into an object reference in use. It doesn’t have a strong-type identity.
- But we can create an array of the “raw type”, which handles an array of references, and then use those reference copies with raw objects created in a generic. You do what you want to do.
- The linked chapter shows another way here: use a wildcard generic. Since it doesn’t specify a type, it’s allowed here, ...

### Hashing with separate chaining: using array of LinkedList

```java
class SeparateChainingHashMap<Key, Value> {
    private int m; // HT size
    private LinkedList<Map.Entry<Key, Value>>[] st; // HT
    // … in constructor, create the hash table: needs @SuppressedWarnings("rawtype", "unchecked")
    st = new LinkedList[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, unnecessarily. No array type here to cause problems. Works either way.
}"
```

### Pg. 474: 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; // old values of keys, vals are GC’d
    vals = temp.vals;
    m = temp.m;
    // Here we create a T within a method of T (outside the constructor of course!)
    We are using Java’s garbage collection (GC) to clean up the old data structure when we overwrite keys and vals, the master HT variables.
    Pg2: This is one way to do it. See “raps” doc.
}"
```
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
  • insert 33, 34, 35, ..., 63 — still less than half full (32 = 2^5 ops)
  • insert 64: second resize, to M=256 (64 = 2^6 ops)

In general, during inserts n+1 to 2n, we do n simple inserts, each O(1), plus 2n relocations into a new hash table, each also O(1).

This totals O(n) inserts and O(n) relocations.

If we amortize this over the n inserts, that is, average the total time over the total number of inserts between rehashes, n, 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.

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.
• Specifically, the "little STs" in buckets of the separate chaining HTs, with code on pg. 375 [link], use an inner class Node.
• 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;
    }
    ...
    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.

Other code in Stack.java using Nodes:

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

pa2: has Node inner class.

```java
private class Node {
    Item item;
    Node next;
}
```
• 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++;
}
```

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;
    ...
  }
  ```
  Outer class variable
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> {
    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") to keep from showing a warning to users when the needed cast happens.
- Note: the HashMap constructor can accept an initial size for the hash table, saving on all those resizes in the case you can estimate the final needed HT size:

```java
Map<String> map = HashMap<String>(1000000);
```