Last class: 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, if any) in the hash table.

• The notation $h(x)$ was used for a hash function that returns a bucket number for key $x$, that is, an integer between 0 and $M-1$ where the hash table is of size $M$.

• But note that Java Object.hashCode() method may return an integer of any size. The HashMap/HashSet code computes a bucket number from this $h(x)$ by $bucket = hash \% M$. 
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.
• “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

Hashing with separate chaining for standard indexing client
Understanding Object Graphs

- The outer box represents one SequentialSearchST (code, 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 (an Integer). 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.
- The key = “A” is a String that actually has its own reference and contents outside the Node. Similarly, the value = 8 is an Integer object hanging off.
• First add: key S, value 0, hash 2, find HT[2] list, empty, add S-0
• 2\textsuperscript{nd} add: key E, val 1, hash 0, find HT[0] list, empty, add E-1
• 4\textsuperscript{th} add: key R, val 3, hash 4, find HT[4], empty, add R-3
• 5\textsuperscript{th} add: key C, val 4, hash 4, push C-4 onto H[4] list, now has 3 elements
• 7\textsuperscript{th} add: key E, val 6, hash 0, replace E’s value with this one, ←note ST behavior
• This is implementing a ST/Map: each key has only one value
• This is an array of objects, each of which contains a list.
• These are the collision lists of the hash table
• We generally try to keep them shorter than this!
• Each list is typed as 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.
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.
- Well, there is a really ugly part needed to set up the needed hash table array, as we will see
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 that works like h(x) described last class to find a bucket no.
• Get and put both hash the key, use that hashval to locate the bucket in the hashtable, and thus the bucket’s own little ST/map, ready for get or put.
• This is the neat-as-a-pin code mentioned on the last slide. Prepare yourself for ugliness…
S&W Hashing with separate chaining: ugly 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: need
    @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 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: 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.
```
FYI: Towards understanding Generic types and arrays

• A generic class like HashSet<String> doesn’t have its own source code (i.e. code for the HashSet of String case)

• Instead, it uses the common HashSet class (the “raw type”) that accepts elements of all object types via object references.
  • The HashSet code can call element.hashCode() since hashCode is an Object method.

• Java arrays are typed in a way so that if x is a subtype of y, the array of x is a subtype of array of y, so the array type handles the element type in a inheritance-sensitive way. Ref oreilly book

• That means an array element can’t directly be a generic type object, which dissolves into an object reference in use. It doesn’t have a strong-enough type identity.

• But we can create an array of the “raw type”, which yields an array of references, and then use those reference spots with refs to objects created via generics. You do what you have to do.

• The linked chapter shows another way here: use a wildcard generic. Since it doesn’t specify a type, it’s allowed here.  ... = new SequentialSearchST<?>[m]
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
        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.
Hashing with separate chaining: 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];

    With this setup, get and put have more work to do than we saw in the S&W code. After they locate the bucket, they need to search the list in it for the key. If it’s found, access the value. If not, fail on get or add to the list for put.

    This search code is inside the little bucket STs for the S&W code.
```
The other Hashtable scheme: 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

Trace of linear-probing ST implementation for standard indexing client
“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 key is null, look no further, set the key in keys[55] and val in values[55] and return. If that key.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.
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.
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)
Performance of Resizing

- 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.
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 (online), 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

public class Stack<Item> implements Iterable<Item> {
    ... instance variables of outer class ...
    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 don’t discuss inner classes much, just a little note on pg. 159.
private class Node {
    Item item;
    Node next;
}

• Other code in Stack.java using Nodes:
  public void push(Item item) {
    Node oldfirst = first;
    first = new Node();
    first.item = item;  \(\leftarrow\) 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 = ...
```
Note on Object types, Upcasting and Downcasting

• Consider `Point2Dim p = new Point2D(3, 4);`
  • For `Point2Dim` definition, see [class3](#), for `Point2D`, page 77
  • Here the reference variable `p` is of interface type `Point2Dim`, and it points to an object of type `Point2D`, a subtype of `Point2Dim`. We’re saying we intend to use the interface’s API with `p` in the code we’re writing, i.e., we don’t need the full API of the `Point2D` class. The code can be used for `PointXY` too.
  • This assignment is said to do an upcast, from `Point2D` to its supertype `Point2Dim`
  • Upcasts always work, and we don’t use a cast operator here
  • Downcasts are more problematic…
FYI: Downcasting

• From the last slide we had p, a variable of type Point2Dim, with which we can call Point2Dim methods but not some of Point2D’s.
• Now suppose we’re working with p, but want to use a method of Point2D not in the Point2Dim interface, say draw().
• To do this, we can downcast p to the needed subtype:
  Point2D p1 = (Point2D)p;  // Now p1.draw() is callable
• We need to use the cast here, and Java will check that p’s object is in fact a Point2D, or throw a ClassCastException if not.
• To avoid ClassCastException, we can check type with instanceof…
FYI: Downcasts and instanceof

Point2D p1 = (Point2D)p;

- This is a downcast from p’s Point2Dim to its subtype Point2D, which will throw a ClassCastException if p’s object is not in fact a Point2D as expected here.

- We can check p’s object’s type with instanceof:
  
  ```java
  if (p instanceof Point2D) {
      Point2D p1 = (Point2D)p;
      ... do something with p1, like p1.draw()
  }
  ```

- This way, we can code mostly with the more abstract type, and occasionally dip down to the more specific type just when needed. It’s clear what is needed to make PointXYs work too.
FYI: What about downcasting with generic types?

- We can still cast, but instanceof isn’t so helpful
- Example:
  
  ```java
  Set<String> x = new HashSet<String>();
  ```
- Then
  
  ```java
  HashSet<String> y = (HashSet<String>)x; // downcast back
  ```
- We try to check with instanceof:
  
  ```java
  if (x instanceof HashSet<String>) // won’t compile!
  ```
- Best we can do: check raw type by using wildcard:
  
  ```java
  if (x instanceof HashSet<?>) // works but only partial check
  ```
We have seen that a reference variable $p$ can have one type, while the object it points to can have another, though it must be a subtype of $p$’s type.

The type of the reference variable is call $p$’s static type, while the type of the object pointed-to is called $p$’s dynamic type.

This makes sense: the static type can’t change because it is compiled in at the time of $p$’s definition: `Point2Dim p;`

The dynamic type can change by pointing $p$ to an object of one subtype and then to another object of another subtype of its static type.
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.
- 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.
- 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);
```
FYI: 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>keys</th>
<th>hash table</th>
<th>Data records</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>memory</td>
<td>memory</td>
<td>memory</td>
<td>typical HashMap apps</td>
</tr>
<tr>
<td>memory</td>
<td>memory</td>
<td>disk</td>
<td>use HashMap to hold disk record locations as values</td>
</tr>
<tr>
<td>memory</td>
<td>disk</td>
<td>disk</td>
<td>hashed files, some database tables</td>
</tr>
</tbody>
</table>
FYI: 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