Problem: Linear searches for items within a data structure are too slow.
Solution: Store items in a sorted, balanced data structure to allow for $O(\log n)$ lookup.

But can we do better?

1 Hashtables

• The idea is to provide direct access to any element in constant time
• Provided that we know where an item should reside in the table (an array) and there are only a constant number of items at that location, lookup should be fast
• How do we decide where an item should reside in the array? Using a hash function, which is a mapping from that type of data to an integer value, which is then used to choose the correct array element
• Items are hashed (by a hash function) by their keys to buckets (cells in an array)
• We must ensure that the number of keys stays within a constant factor of the number of buckets and that the keys are almost evenly distributed across all the buckets
• With the above conditions, lookup, insertion, and deletion of items runs in $O(1)$ time

2 Simple hash functions

• Provide a mapping from key $\rightarrow$ bucket by taking some information from the key, manipulating it as an integer value, and choosing a bucket number
• A simple hash function for int values would be $x \% A.length$ where $x$ is an int and $A$ is the array in which we’d like to place int values
• If two items are hashed by their keys to the same location, a collision occurs (which isn’t as bad as it sounds), and the items are externally chained (that is, the item is attached to the linked-list that exists at that array index)

3 Hash function issues

• Problem with simple int hash function – if the keys of the items being placed in the hashtable are not relatively prime with the size of the array, the items are all placed in the same bucket, resulting in $O(n)$ lookup rather than the $O(1)$ lookup we want
• We want the load factor to remain constant. The load factor = the number of keys / the size of the array
• If the hash function is distributing keys evenly across the hashtable, the load factor is the average number of elements in each bucket
• Once the load factor exceeds a certain value, resize array (to twice the size) – although this operation
takes $O(n)$, its amortized cost per item is $O(1)$ (just like a Vector)
• When resizing the hashtable, all keys must be *rehashed* – after the array size is changed, all hash
functions (hash codes) must be reevaluated and items must be reassigned to buckets

4 A simple hash function on Strings
• Suppose we want to write a hashcode for string values – if we convert the characters of the string into
integer values (which they are), then we can produce a numeric hashcode which we then mod by the
size of the array
• Let’s hash strings by adding the values of all constituent characters: a string "$s_0s_1s_2s_3...s_{n-1}$" is
hashed by computing $s_0 + s_1 + s_2 + s_3 + ... + s_{n-1}$
• This seems to be pretty good – strings will end up evenly distributed...or will they?
• All anagrams will be hashed to the same bucket – ok, but not what we intented

5 Announcements
• Exam #2: Tuesday, July 30, 6:40PM, 2050 VLSB
• Reading: Data Structures (Into Java) – Ch. 7; Goodrich and Tamassia – 8.3

6 Question
Suppose we rewrite the string hashcode as $s_0 + 32^1s_1 + 32^2s_2 + 32^3s_3... + 32^{n-1}s_{n-1}$. We can quickly com-
pute each value using shifts. How well does this hashcode produce unique values for any strings we use it on?

White: It works well because we take into account the position of characters in the string.

Blue: It works well because character values never can be greater than 32.

Yellow: It doesn’t work well because all strings will be hashed to the same bucket.

Green: It doesn’t work well because all strings with the same first 7 characters will be hashed to the
same bucket.

Pink: It doesn’t work well because strings and their reversals will be hashed to the same bucket.

Green is the correct answer – we end up throwing away the bits from $s_7$ onwards because the shift
operation to compute $32^7$ causes the bits to be thrown away (since an int can only hold 32 bits).

7 A good hash function on Strings
• Java uses the following: $31^{n-1}s_0 + 31^{n-2}s_1 + 31^{n-3}s_2 + ... + s_{n-1}$
8 Goals of good hash functions

- All items of equal value should hash to the same bucket
- All (or almost all, probabilistically speaking) items of unequal value should hash to different buckets
- The hash code should be easy to compute (constant across all instances of the data type)
- Takes into account all variations of the data type’s information such that no important defining information is lost (for example, the string question in which we lost all characters after the 7th when computing the hash code)

9 Java’s Hashtable/HashMap/HashSet classes

- The class Object has a method hashCode() which is overridden by standard Java classes to provide a good hash code for each data type
- The classes java.util.Hashtable, java.util.HashMap, and java.util.HashSet use the method hashCode() and equals() to assign keys to buckets
- These classes map keys to values, although the value that is being inserted may itself be the key by which we look up the item
- Typically you’ll want to use Hashtable – HashMap and HashSet are almost the same as Hashtable except they allow null values but are unsynchronized (meaning that their use from multiple threads may result in undefined behavior)

Example

```java
Hashtable h = new Hashtable();
/* Insert items into the Hashtable */
h.put("Barath", new Integer(20));
h.put("Jack", new Integer(21));
h.put("Amir", new Integer(20));

/* Get items from the Hashtable */
Integer age = (Integer) h.get("Amir");
```

10 Hash table performance

- insert, find, and remove are all $\Theta(1)$
- range query (find all items with values between $X$ and $Y$), getMax, getMin are all $\Theta(n)$ because there isn’t a guaranteed ordering in the hash table
- How do we decide on hash codes besides experimenting? Can use universal hashing (which we won’t get into in this class), which allows us to choose a hash function randomly from a collection
11 Memoization

- Can use hashtables to store values of computations that will likely be redundant, thereby reducing time spent recomputing

- Storage is cheap, especially in small operations as opposed to recomputation

- By simply storing, we can reduce the complexity of a program from exponential time to polynomial time (sometimes)

Example
Suppose we are writing a simulation for a rover that can move forwards and backwards and turn in various directions. The rover is moving on an integer cell grid, moving one cell at a time. The rover is on a mission to pick up items in a certain order and determine all the different ways to pick up the items in that order. Suppose in the example below, the rover starts at the top left corner and must pick up items in the order 0012. How can we simulate the rover?

012
010
212