Lecture 11: Hashtables

Wholeness of the Lesson

Hashtables provide even faster access to elements in a collection than a BST, but at the price of losing the sorted status of the elements. If maintaining order among the elements of a collection is not required, hashtables are the most efficient data structure for storing elements in memory, when insertion, deletion, and lookup operations are needed. Hashtables give concrete expression to the ability of pure intelligence to know any one thing instantaneously.

The Hashtable ADT

- A Hashtable is a generalization of an array in which any object can be used as a key instead of just integers. A Hashtable has *keys* which are used to look up corresponding *values.* A typical example is to store records from a database in memory. A key field from the database is often used as a key in the hashtable, and the corresponding record is the value in the hashtable.
- Two basic operations:

```
void put(Object key, Object value);
Object get(Object key)
```

(usually also have a `remove(Object key)` operation)

- Character Table from the Labs

<u>User's view</u>

Char key	**String value**
`'a'`	`"Adam"`
`'b'`	`"Bob"`
`'c'`	`"Charlie"`
`'w'`	`"William"`

```
//insert into table
table.put('c', "Charlie");
//retrieve from table
table.get('c');  //returns "Charlie"
```

<u>Implementation:</u>

put(c, s):

- `c` → `(int)c` [obtain hash code]
  → `i = (int)c - (int)'a'` [obtain hash value]
- insert `new Entry(c, s)` into `table[i]`

get(c):

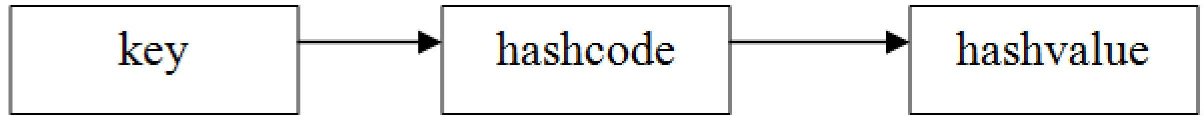
- `c` → `(int)c` [obtain hash code]
  → `i = (int)c - (int)'a'` [obtain hash value]
- `Entry e = table[i]; return e.value`

<u>Pattern:</u>

KEY (non-number) → HASHCODE (number) → HASH VALUE (array index)

Creating a Hashtable

- Steps:
  a. Devise a way of converting keys to integers so that different keys are mapped to different integers. This is what Java's `hashCode()` function is for.
  b. Devise a way of converting hashcodes to smaller integers (called *hash values*) that will be the indices of a smaller array, called the *table*. To do this, you usually need to decide on the `tableSize`, which is the length of the array. A typical way (note: Java does it differently) of making hashvalue from hashcode is by the formula
     `hashvalue = hashcode % tableSize`



`put` operation

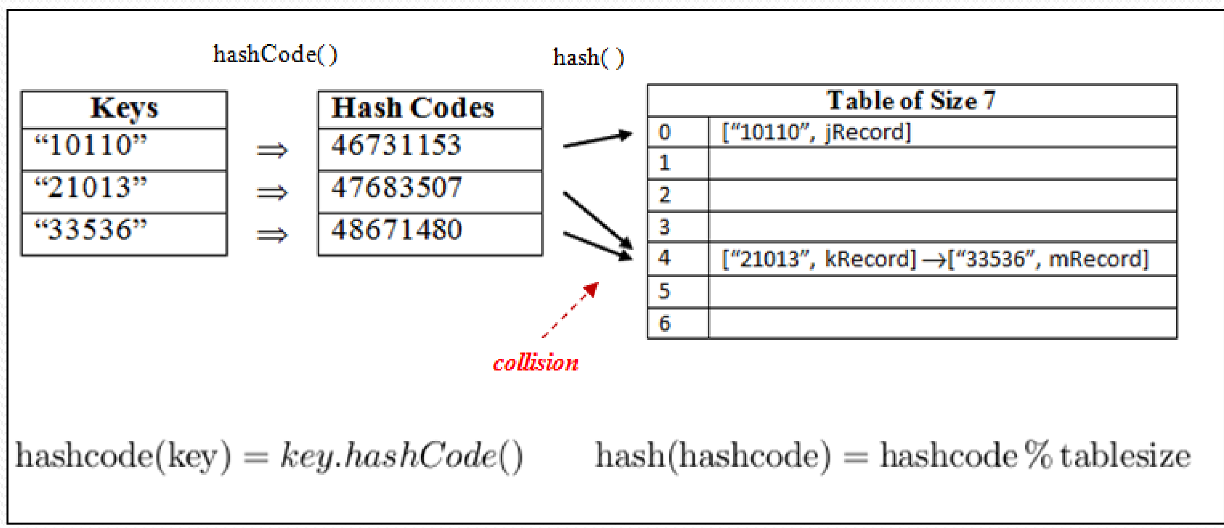
Step 1: Compute index in table where data will be placed

```
- Entry e = [k, val]
- hashcode = hashcode(k)
- h = hashvalue(hashcode)
    = hashcode % tableSize
```
(typically)

Step 2: Place the `Entry` object in the table.

- Cannot simply store `e` in `table[h]` because of potential collisions.
- *Solution*: Each table slot stores a `LinkedList`. Store `e` by:

```
table[h].add(e)
```



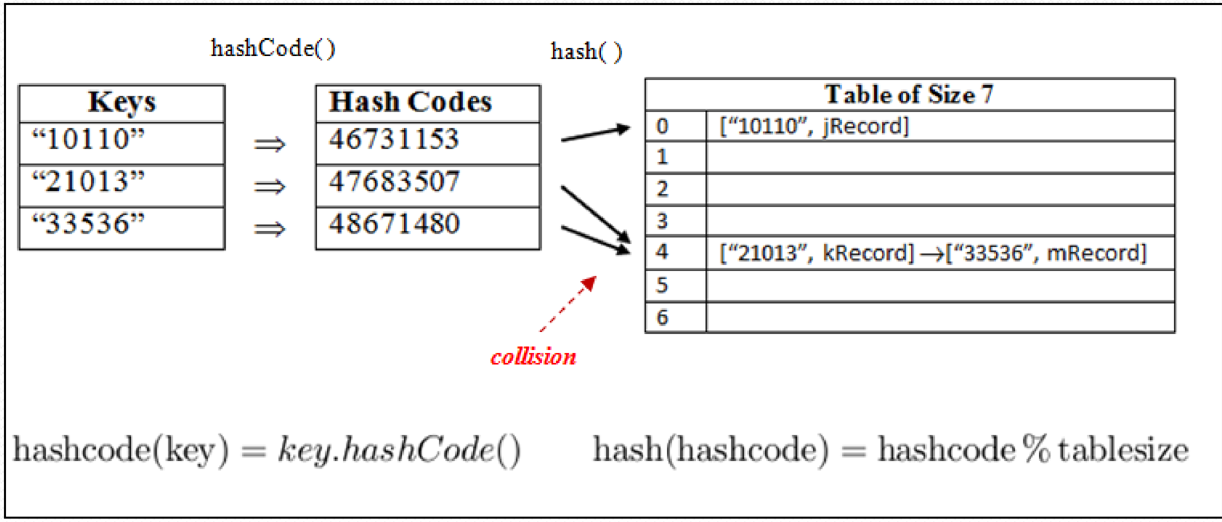
`get` operation

Step 1: Compute index in table where data will be found

```
- hashcode = hashcode(k)
- h = hashvalue(hashcode)
    = hashcode % tableSize
```
(typically)

Step 2: Read the `Entry` object from the table.

```
- LinkedList l = table[h]
- Entry e = l.find(k)
- return e.val
```



Overriding the hashCode() Method

- Any implementation of the Hashtable ADT in Java will make use of the `hashCode()` function as the first step in producing a hash value (or table index) for an object that is being used as a key.
- Default implementation of `hashCode()` provided in the Object class is not generally useful – gives a numeric representation of the memory location of an object. See package `lesson11.defaulthashcode`

  Example: We wish to use pairs (firstName, lastName) as keys for Person objects in a hashtable.
 (See package `lesson11.needoverridehashcode`)

  Demo show default `hashCode` method is not useful. If two `Pair` objects, created at different times, are equal (using the `equals` method), we would expect them to have the same `hashCodes`, so that, after hashing, they are sent to the same table slot. But default `hashCode` method does not take into account the fields used by `equals` method, so equal `Pair` objects may be assigned different slots in the table.

hashCode() Rules

- To use an object as a key in hashtable, you must override `equals()` and `hashCode()`
- If k_1, k_2 are keys and k_1`.equals(`k_2`)` then it must be true that [k_1`.hashCode() ==` k_2`.hashCode()`]
  This means that you must include the same information in your `hashCode` definition as you include in your implementation of `equals`.

Creating Good Hash Codes When Overriding hashCode()

- There are two general rules for creating hash codes:
  I. (Primary Hashing Rule) Equal keys must be given the same hash code (otherwise, the same key will occupy different slots in the table)
  If `k1.equals(k2) then k1.hashCode() == k2.hashCode()`

  II. (Secondary Hashing Guideline) Different keys should be given different hash codes (potential danger: in the worst case, if every key is given the same hash code, then all keys are sent to the same slot in the table; in this case, its performance degrades dramatically.
  Best Practice: The hash codes should be distributed as evenly as possible (this means that one integer occurs as a hash code approximately just as frequently as any other)

Creating Hash Codes from Object Data (Legacy Approach)

You are trying to define a hash code for each instance variable of a class. Suppose `f` is such an instance variable.

- If `f` is boolean, compute `(f ? 1 : 0)`
- If `f` is a byte, char, short, or int, compute `(int) f`.
- If `f` is a long, compute `(int) (f ^ (f >>> 32))`
- If `f` is a float, compute `Float.floatToIntBits(f)`
- If `f` is a double, compute `Double.doubleToLongBits(f)` which produces a long `f1`, then return `(int) (f1 ^ (f1 >>> 32))`

Creating Hash Codes from Object Data (Legacy Approach)

- *Hashing Objects.* Good hashCode functions will create a hashCode based on the instance variables of the object (using the same data as is used to override equals). Here is standard practice (described by J. Bloch) for combining the hashCodes of the instance variables.

- Suppose your class has instance variables u, v, w and corresponding hashCodes hash_u, hash_v, hash_w (if u, v or w is a primitive hash_u, hash_v, hash_w would be obtained as described in the last slide; otherwise if, say, u is an object of some kind, hash_u = u.hashCode()). Then:

```
@Override
public int hashCode() {
  int result = 17;
  result += 31 * result + hash_u;
  result += 31 * result + hash_v;
  result += 31 * result + hash_w;
  return result;
}
```

Creating a Hash Value from Object Data (Modern way)

- Use the following method in `Objects` class to compute hash code.

```
public static int hash(Object... values)
```

- For example, if an object has three fields, x, y, and z, we could compute hash code in this way:

```
@Override
public int hashCode() {
  return Objects.hash(x, y, z);
}
```

Overriding HashCode for Pair class

Example. *Overriding hashCode in the Person-Pair example.* We must take in account the same fields in computing hashCode as those used in overriding equals. The fields in Pair are Strings, and Java already provides hashCodes for Strings. So we make use of these and combine them to produce a complex hashCode for Pair.

```
//modern way
public int hashCode() {
    return Objects.hash(first, second);

}
```

```
//legacy approach
public int hashCode() {
    int result = 17; //seed
    int hashFirst = first.hashCode();
    int hashSecond = second.hashCode();
    result += 31 * result + hashFirst;
    result += 31 * result + hashSecond;
    return result;

}
```

HashCode in Java's String Class

Example. *How Java overrides hashCode in the String class*: Any Java String is converted to an integer via `hashCode()` by this formula:

Given a `String s` of length `k+1`

`s.hashCode()` **equals** 31^{k} `* s.charAt(0) +`
31^{k-1} `* s.charAt(1) +`
31^{k-2} `* s.charAt(2) +. . .+`
31^{0} `* s.charAt(k)`

Since every character in the String is taken into account, equal Strings must have equal hashCodes. Because of the formula, it is highly unlikely that two distinct Strings will be assigned the same hashCode (though it's possible)

Java's Implementation of Hashtables

- HashMap and Hashtable (HashMap is preferred; Hashtable is "legacy")

Feature	java.util.HashMap	java.util.Hashtable
Allows null key	Yes	No
Allows null values	Yes	No
Allows duplicate keys	No	No
Synchronized (for safe multithreading)	No	Yes

Java's Implementation of Hashtables

- Pre-j2se5.0

```
HashMap map = new HashMap();
map.put("Bob", new Employee("Bob",40000,1996,10,2));
Employee emp = (Employee)map.get("Bob");
```

- j2se5.0 version is parametrized:

```
HashMap<String, Employee> map =
        new HashMap<String,Employee>();
map.put("Bob", new Employee("Bob",40000,1996,10,2));
//no downcasting required
Employee emp = map.get("Bob");

map.put("Bob", new Employee("Bob",50000,1996, 10, 2));
//The Employee Bob with salary 50000 will be returned.
emp = map.get("Bob");
```

Main Point

The most common implementation of hashtables uses separate chaining; each hashvalue is an index in an array of Lists; all objects with colliding hashvalue *i* are stored in the *i*th list in the array. The solution to the problem of avoiding hashvalue collisions (namely, by using a List in each of the array slots) illustrates the principle of the second element: Using a List in each array slot provides a way to "harmonize" objects that were apparently in conflict because of identical hashvalues.

Hashtable Application #1: Removing Duplicates

- The most common use of hashtables is as in-memory look-up tables. For example, Employee records from a database could be stored by using Employee ID as key and the entire Employee record as value.

- Another application of hashtables is for "bookkeeping" purposes. A simple example is an efficient procedure for removing duplicates from a list. The "naïve" way to remove duplicates is to use nested loops: For each element `e` in the list, use an inner loop to look at all elements preceding `e` in the list to see if `e` has occurred before; if so, remove this second occurrence of `e`.

- A more efficient approach is to do the following: Create an auxiliary hashtable `H`. For each `e` in the list, check to see if `e` is a key in `H`. If so, remove `e` from the list. If not, add the entry `<e,e>` to H.
- See `package lesson11.removedups`

Hashtable Application #2: The Set ADT

1. Mathematically, a ***set*** is (roughly) a collection of objects. Two sets are said to be equal if they have the same elements.

   For example:

   `{1, 1, 3} = {1, 3} = {3, 1}`

   because all have the same elements.

   A set does not impose an ordering of elements (the set may contain elements that have their own natural order, like integers, but the set itself does not impose an order)

2. We can represent the mathematical notion of a set as an ADT called `Set`. In order to faithfully represent the properties of the mathematical idea, the `Set` ADT must have the following characteristics:

- It does not allow duplicate elements
- Iterating through a set does not guarantee any special order on its elements (in particular, there is no guarantee that the order in which an Iterator will provide elements corresponds to the order in which the elements were added in the first place)
- Its overridden `equals()` method declares two `Sets` to be equal if and only if they have the same elements

3. Java provides a `Set` interface (and in j2se5.0, the parametrized version `Set<E>`) as part of the Collections API.
4. Two implementations
   - `HashSet` Using a `HashMap` to implement Sets prevents duplicate elements from being added in the `Set`. Does not guarantee order in which elements are stored
   - `TreeSet`. Uses a red-black tree to store elements, so when they are output, they are in sorted order
   - Demo: `lesson11.sets`

Main Point

The Hashtable ADT is a generalization of the concept of an array. It supports (nearly) random access of table elements by looking up with a (possibly) non-integer key. In the usual implementations, objects used as keys in a hashtable are "hashed", producing hashcode (a numeric value) and hashvalue (numeric value reduced in size to be less than the table size). The hashvalue is an index in an array that can be used to locate or insert an object. Hashtables illustrate the principle of Do less and accomplish more – they provide an incredibly fast implementation of the main List operations.

Guidelines For Use of Common Data Structures

1. **Array List**
   - Use When: Main need for a list is random access reads, relatively infrequent adds (beyond initial capacity) and/or number of list elements is known in advance.
   - Avoid When: Many inserts and removes will be needed and/or when many adds expected, but number of elements unpredictable. Also: Maintaining data in sorted order is very inefficient.

2. **Linked List**
   - Use When: Insertions and deletions are frequent, and/or many elements need to be added, but total number is unknown in advance. There is no faster data structure for repeatedly adding new elements than a Linked List (if elements are always added to the front).
   - Avoid When: There is a need for repeated access to ith element as in binary search – random access is not supported.

3. **Binary Search Tree**
   - Use When: Data needs to be maintained in sorted order. Faster than Linked Lists for insertions and deletions, but ordinary adds are slower. Provides very fast search for keys.
   - Avoid When: The extra benefit of keeping data in sorted order is not needed and rapid read access is needed (Array List provides faster read access by index and hashtables provide faster read access by key)
4. **Hashtable/Hashmap**
   - Use When: Random access to objects is needed but array indexing is not practical. Provides fastest possible insertion and deletion (faster than BST's).
   - Avoid When: The order of data must be preserved or "find Max" or "find Min" operations are needed.
5. **Sets**
   - Use When: Duplicates should be disallowed, and there is no need for rapid lookup of individual set elements. To order the elements, use `TreeSet`.

Connecting the Parts of Knowledge With the Wholeness of Knowledge

Random access expanded from integer index to arbitrary index, from "point" to "infinity"

1. Arrays and ArrayLists provide highly efficient index-based access to a collection of elements.
2. The Hashtable ADT generalizes the behavior of an array by allowing non-integer keys (in fact, any object type can be used for a key), while retaining essentially random access efficiency for insertions, deletions, and lookups.

---

3. **Transcendental Consciousness:** TC is the home of all knowledge. The Upanishads declare "Know that by which all else is known" – this is the field of pure consciousness.
4. **Wholeness moving within itself**: In Unity Consciousness, one sees that the "key" to accessing complete knowledge of any object is the infinite value of that object, pure consciousness, which is known in this state to be one's own Self. Knowing that level of the object, it then becomes possible to know any more relative level of the object as well.