Hashing and Hash Tables

Weiss, chapter 20 (pages 773–806)
Hash Map

• Two implementations for Map so far
  ▪ Association list: linear time lookup $O(N)$
  ▪ Tree: logarithmic lookup $O(\log N)$

➢ Array: constant time lookup
  ▪ Keys are indices (integers) and values are the elements of the array
  ▪ Array is called hash map

• Example
  ▪ Consider an inventory. Build a map that relates product IDs to their counts in the inventory.
  ▪ IDS vary in a range from 0 to 999. IDs are unique.
  ▪ Implement the inventory with an array. The array size is 1000.
Hash Map

• Array does not work if the range of key values is not known.
  - Space of keys is sparse. A Few keys are required from the range.
  - An array list wastes a lot of space. Array list needs to be as large as the largest key.

```java
[null, null, ..., v]
```

```
| 10000 |
```
Hashing (cont.)

• Solution
  ▪ Use modulo operator with the fixed size array
  ▪ Assume an array of size 10
  ▪ Compute mod of key by 10:
    o Values 0, 10, 20, 30, ... map to index 0
    o Values 1, 11, 21, 31, ... map to index 1
      [ null, null, v1, null, null, v2, null, null, v3, null ]
      key232      key555      key8

• Problem
  ▪ Storing an entry for key 1992 causes a collision.
  ▪ Two keys use the same position in the array

• Two methods for resolving collisions: probing and chaining
Linear Probing

- When an index is already taken by another key, search for an open position to store the value.
- Linear probing
  - Start from the original (preferred) index corresponding to the key. Search successive positions until an open spot is found.
  - Also store the key in array cell. Comparing a target key with keys stored in the array.
- For example, add k8 to the array
  Before adding k8: [ null, null, k236, null, null, k555, null, null, k18, null]
  After adding k8: [ null, null, k236, null, null, k555, null, null, k18, k8]
- If searching reaches the end of array, *wrap around* the array (e.g., for adding k9)
Linear Probing

• Removing a key-value pair invalidates subsequent lookups.
• Consider looking up k88. By default, the lookup operation stops when it finds null in a position.

Before removing k8: [k88, null, k236, null, null, k555, null, null, k18, k8]
After removing k8: [k88, null, k236, null, null, k555, null, null, k18, null]

• Instead of physically deleting a pair, mark it as a deleted. This is called lazy deletion.

• Cost of Map operations: lookup and put
  ▪ $O(1)$ without collision
  ▪ $O(N)$ with collision: linear scan of array to find the key
Chaining

- With linear probing, a significant portion of a *dense* array needs to be scanned by the lookup operation.
- Instead, keys that are associated with the same index are stored at that index in the array.
- Each spot in the array looks like a chain (list) of key-value pairs.
- The method of collision resolution is called *chaining*.
- Each array spot is also called a bucket that has an index.
- Scan buckets instead of array spots. Time to scan a bucket is proportional to the numbers of keys that are collided in that bucket.
Chaining

• In the best case, each bucket has only one key-value pair. Lookup takes O(1) time.
• In the worst case, all keys are in the same bucket. Lookup takes O(N) time.

• When the number of keys is much greater than the buckets, the array is resized.
  ➢ The keys are rehashed to buckets. Indices of keys are different.
• Resizing depends on load factor, which measures the fullness of array. It computes the ratio between the number of key-value pairs and the number of buckets. Load factor is between zero and one.
• A low load factor wastes a lot of space in the array.
• A high load factor increases the collision. The look up becomes O(N).

➢ Java library uses chaining with 0.75 load factor.
Hashing

- Converts keys of type $T$ to integers. It does that by a *hash function*.
  
  \[ h: T \rightarrow \text{int} \]

- Hash function has some restrictions.

- Consider a hash function converts characters (keys) to zero:
  
  \[ h(c) = 0 \]

- It is not a good hash function. All keys are collided in the same bucket.
  
  ➢ This hash function does not *distribute* keys effectively.

- Consider a hash function that converts characters to random integers:
  
  \[ h(c) = \text{rand}(0, 100) \]

  ➢ This hash function is not *consistent*. For the same character, the function generates different integers.
Hashing for Primitives

• A good hash function
  ▪ distributes key evenly over the space of possible integers.
  ▪ consistently generates the same integer for the same keys
    if k1 = k2 then h(k1) = h(k2)
  ➢ For the same integers, keys may be different. This is key collision
    (constant hash function has too many collisions).

• For characters in Java, hash function returns the integer values.
  public static int hashChar(char ch) { return (int) ch; }
• For integers, hash function returns integers: h(n) = n
• For booleans, hash function returns two integers:
  h(true) = 1, h(false) = 0
• For floats, `Float.floatToIntBits` returns an integer equivalent of bits that
  represent float numbers.
  Float. floatToIntBits(3.14) = 1078523331
Hashing for Objects

- In Java, Object class has `hashCode` method for hashing. Other classes override `hashCode`.
- If two objects (keys) are *equal*, `hashCode` returns the same hash value. Two unequal objects may have the same hash value. ➤ It is hash collision, which should be avoided as much as possible.
- Object class returns memory address of objects as hash values. `equals()` checks equality using memory addresses (object identity).
- Derived classes need to override `equals` with respect to the structure (fields) of objects.
- To preserve *consistency*, classes must override `hashCode`. 
public class Point {

    private int x;
    private int y;

    // constructor without argument
    public Point() {

    }

    // constructor with two arguments
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o) {
        if (o instanceof Point) {
            Point rhs = (Point) o;
            return this.x == rhs.x && this.y == rhs.y;
        } else {
            return false;
        }
    }
}
Hashing for Objects

```java
@Override
public int hashCode() {
    return x + y;
}
```

- The above hash function generates the same integer for (0,1) and (1,0) objects.
- The hash function below was suggested by Joshua Bloch:

```java
@Override
public int hashCode() {
    int result = 1091;  // a random non-zero integer
    result = 31 * result + x;
    result = 31 * result + y;
    return result;
}
```

- 31 is an odd prime number. It reduces key collisions and speeds up multiplication.