Hash Tables and Graphs

Nick Galbreath ("nickg")

nickg@friendster.com
Friendster

I know you are going to ask anyways!

- Why is the site so slow?
- Are/When we charging?
- Were does the money come from?
- Are we shutting down for 3 months?
- Why do to kill Fakesters?
- It would be really cool if...
Notions of Equality

- Java has two notions of equality for objects and arrays, (e.g. int[]) reference equality and data equality.
- For “primitive types” (ints, longs, bytes), only data equality. 3 == 3 always.
Reference Equality

- Reference equality is when two objects or arrays refer to the same data in memory. They are aliases for each other.
  - `MyData a = new MyData(...);`
  - `MyData b = a;`

- This type of equality is tested by “==”
  - `(a == b)` is true in the example above.

- If `(a == b)` is true, then `a` and `b` are actually the same object. A change in one will result in a change in the other.
Data Equality

- Data equality for when two different objects have the same data but different locations in memory.
- This equality is tested with method
  - `boolean equals(Object o)`
- If `a.equals(b)` is true then they contain the same data, but may be different objects.
public class MyData {

private String name;
private int age

public MyData(String value, int age) {
    this.name = name;
    this.age = age
}
}
public static void main(String[] args) {

    MyData a = new MyData("Fred", 23);
    MyData b = new MyData("Fred", 23);
    MyData c = a;

    System.out.println("a == b " + (a == b));  // ref equality
    System.out.println("a == c " + (a == c)); // ref equality
    System.out.println("a.equals(b) " + (a.equals(b)));
    System.out.println("a.equals(c) " + (a.equals(c)));
}

a == b false
a == c true
a.equals(b) false    Huh???
a.equals(c) true
Equals Default Behavior

- By default, the class `Object` defines equals to be the same as reference equality.
- This means `equals` will return `false`, on objects with the same data but with different references!
- You must define your equals method, individually testing fields.
- You might not need to test every field.
public class MyData {

    private String name;
    private int age

    public MyData(String value, int age) {
        this.name = name;
        this.age = age
    }

    public boolean equals(Object o) { 
        MyData rhs = (MyData)o;
        return name.equals(rhs.name) && age == rhs.age;
    }
}
Test Harness and Output

```java
public static void main(String[] args) {
    MyData a = new MyData("Fred", 23);
    MyData b = new MyData("Fred", 23);
    MyData c = a;

    System.out.println("a == b  " + (a == b));
    System.out.println("a == c  " + (a == c));
    System.out.println("a.equals(b)  " + (a.equals(b)));
    System.out.println("a.equals(c)  " + (a.equals(c)));
}
```

```
a == b  false
a == c  true
a.equals(b) true   // good!
a.equals(c) true
```
When to Reference Equality

Use Reference Equality (==) when

– All the objects to be tested are already created and no more are being added.
– Preventing “self-assignment” (e.g. a = a)
– When the program is short-lived
– Preventing “double-counting” when iterating through a list.

```java
MyData a = from a collection.
MyData other = null
for (Iterator i = collection.iterator(); i.hasNext(); other = (MyData) i.next()) {
    if (a != other) { do something; }
}
```
When to use Data Equality

- Use Data Equality (equals) everywhere else!
  - Objects are being created from an external source and being compared with internal data
    - From the database
    - From user input
    - From a web query string or form
    - Any remote source

When the program is long-lived
Server environments

In most cases you won't be wrong to use data equality. You just might be a bit slower.
Hash Tables are the Most Important Data Structure!
You use them all the time!
Java has some tricks you need to know about!
Hash Tables and Hash Sets

- A Set in general, is a collection of unique objects, no duplicates!
- A Hash Tables is a collections of mappings between a key and a value. Keys are unique and form a set.
- A Hash Set is a special implementation of a set that uses hashing to quickly find elements. One way to think of a hash set is that it's a hash table but the keys and values are the same.
java.util.HashMap

- Standard “chained” implementation. Java does not have a native open addressing version.
- Has a number of “buckets” each of which has a (singularly linked) list holding data.
- “An Array of Linked Lists” (Question: why single instead of doubly linked list?)
- When accessing or adding data, HashMap converts the key into a number by use of `int hashCode();`
- The integer is then turned into a bucket number (i.e. an array index).
Hash Codes

- **hashCode** is defined in `java.lang.Object` which all objects are derived from
- It converts the object into a number (int) such that
  - Does it quickly
  - Is consistent and deterministic. Two objects with identical data (or are "equal") should produce the same hash code
  - Given a collection of objects, the hash code they produce should be essentially random – no clumping or repeats.
More Hash Codes

- Java defines a pretty good hash function for the `String` class based on the data the string contains.
- For custom classes you write, Java uses an internal memory reference for the hash. Just like the default `equals()`. Not so good.
- Write your own!
- Easiest way to use the `hashCode()` of a internal value (e.g. Name, Key, etc)

```java
public class Foo {
    protected String name;
    int hashCode() { return name.hashCode(); }
}
```
More Hash Codes

- If you have a unique integer identifier (user id, social security number, account number), you can use that as is.
- Java's `HashMap` will also scramble further the result to make it “more random like”
HashMap and Equality

• After a bucket is selected, the corresponding linked list is searched to see if the object already exists.
• It's important to make sure your custom classes define the `boolean equals(Object o)`
• If you don't, Java uses “reference equality”, which is ok in some situations, NOT ok in most server situation.
public class MyData {

    private String name;
    private int age

    public MyData(String name, int age) {
        this.name = name;
        this.age = age
    }

    public int hashCode() {
        return name.hashCode(); // could be fancier but ok
    }

    public boolean equals(Object o) {
        MyData rhs = (MyData)o;
        return name.equals(rhs.val) && age == rhs.age;
    }
}
public static void main(String[] args) {

    HashSet map = new HashSet();
    MyData a = new MyData("Fred", 23);
    MyData b = new MyData("Fred", 23);
    MyData c = new MyData("Fred", 23);
    map.add(a);
    map.add(b);
    System.out.println("Hash Code for a = " + a.hashCode());
    System.out.println("Hash Code for b = " + b.hashCode());
    System.out.println("Hash Code for c = " + c.hashCode());
    System.out.println("Set size = " + map.size());
    System.out.println("C Exists? " + map.contains(c));
}

Test Harness
Test Output

Output:
  Hash Code for a = 110182
  Hash Code for b = 110182
  Hash Code for c = 110182
  Set size = 1
  C Exists? true

• What happens if we remove the equals() method? And why?
• What happens if we remove the hashCode() method? Any why?
• What happens if both equals() and hashCode() are missing?
HashMap Constructors

- `java.util.HashMap` has three constructors:
  - `HashMap() // default 16 buckets, 75% load`
  - `HashMap(int capacity) // default 75% load`
  - `HashMap(int capacity, float load); // you pick`

- `capacity` is the number of buckets
- `load` is average the number of elements each bucket can hold
- After that, a `resize` event happens.
- This happens `size() > capacity * load`
HashMap Resize Events

- Once the size exceeds the threshold, a resize event happens:
  - The number of buckets is doubled.
  - Every element is rehashed.
- No big deal for small maps, but every expensive with large maps (k or M).
- The HashMap default is only 16, with load of 75%. After 12 elements are added a resize event happens.
- Multiple resizes event can happen when adding a large data. Can be Slow.
HashMap Summary

- When creating new objects, always write `hashCode()`
  write `equals()`
  write `toString()`
- This is a good idea anyways!
- Understand the `HashMap/HashSet` constructors and set appropriately, especially when you know the data set is going to be large. It's ok to be generous.
Graphs

- An extension or generalization of trees.
- With trees, each node has exactly one parent.
- With graphs, each node can have many parents.
What do you use them for?

- Airline routes – for travel sites, and for the airlines themselves
- Logistics – shipping and delivery routes
- Manufacturing – robotic control and circuit boards – Try to minimize motion, increase speed.
- Information management and visualization (words, books, data, websites)
- and... social networks!
Nodes and Vertices

- With trees, a data point is a “node”
- With graphs, a data point is sometimes called a “vertex”, plural “vertices”
- It's perfectly fine to use “nodes” when talking about graphs.
- I like nodes better since it's easier to name methods, and it's faster to type!
What is a Graph?

- A graph $G = (V,E)$ is composed of:
  - $V$: set of vertices
  - $E$: set of edges connecting the vertices in $V$
- An edge $e = (u,v)$ is a pair of vertices
- Example:
  
  $V = \{a,b,c,d,e\}$
  
  $E = \{(a,b),(a,c),(a,d), (b,e),(c,d),(c,e), (d,e)\}$
Graph Terminology

• adjacent vertices: connected by an edge
• degree (of a vertex): # of adjacent vertices

\[ \sum_{v \in V} \deg(v) = 2(\# \text{ edges}) \]

• Since adjacent vertices each count the adjoining edge, it will be counted twice
Degree 1

- Nodes degree 1 away can be called
- Adjacent
- Neighbors
- Friends
**path**: sequence of vertices $v_1, v_2, \ldots, v_k$ such that consecutive vertices $v_i$ and $v_{i+1}$ are adjacent.
- **simple path**: no repeated vertices

- **cycle**: simple path, except that the last vertex is the same as the first vertex
Even More Terminology

- connected graph: any two vertices are connected by some path
- **subgraph**: subset of vertices and edges forming a graph

- **connected component**: maximal connected subgraph. E.g., the graph below has 3 connected components.
Connectivity

Let $n = \#\text{vertices}$
\[ m = \#\text{edges} \]

- **complete graph** - all pairs of vertices are adjacent

\[
m = \frac{1}{2} \sum_{v \in V} \text{deg}(v) = \frac{1}{2} \sum_{v \in V} (n - 1) = \frac{n(n-1)}{2}
\]

- Each of the $n$ vertices is incident to $n - 1$ edges, however, we would have counted each edge twice!!!
Therefore, intuitively, $m = \frac{n(n-1)}{2}$.

*n = 5*
\[ m = \frac{(5 \times 4)}{2} = 10 \]
More Connectivity

\[ n = \#\text{vertices} \]
\[ m = \#\text{edges} \]

- For a tree \( m = n - 1 \)

\[ n = 5 \]
\[ m = 4 \]

- If \( m < n - 1 \), \( G \) is not connected

\[ n = 5 \]
\[ m = 3 \]
Connected Components

**Connected Graph**: any two vertices connected by a path

connected
not connected

**Connected Component**: maximal connected subgraph of a graph
Spanning Tree

• A **spanning tree** of $G$ is a subgraph which
  - is a tree
  - contains all vertices of $G$

![Graph $G$ and spanning tree of $G$]

• Failure on any edge disconnects system (least fault tolerant)
Textbook Representation of Nodes

- At least from the second edition, page 620:

```java
class Vertex {
    public char label; // label, e.g. 'A'
    public boolean wasVisited; // huh???

    public Vertex(char lab) {
        label = lab;
        wasVisited = false;
    }
}
```
What does “lastVisited” have anything to do with a node? Is anything in the definition of a node involve if the node was 'visited' or not?

It is a temporary variable for use by another (as of yet unspecified) algorithm.

Mixing *algorithms* into the *data structure* is in general a bad idea. This is ok, when doing user interface, but for servers!

We will revisit the books choice in a bit, since it provides a good negative example.
Node Representation

- Let's start over. All a node really needs is a “key” or sometype of unique identifier. For simplicity, let's just use a String.

```java
Class Node {
    public String key;
    ... other data here...

    public Node(String key, ...other data...) {
        this.key = key;  //etc.
    }
}
```

- Why is `key` public? Should we use a method instead?
- Do we need to use a `String`? Can we use plain `Object`?
Adjacentcy Lists

• An adjacentcy list is simply a list that contains what other nodes are neighbors (friends).
• The map could be as simple as an array.

\[ \text{Nodes}[] \ adjList = \{\text{node1, node2, node3}\} \]
Adjacentcy Lists with Dynamic Storage

- More complicated version can have dynamic-sized storage.
- ArrayList
- LinkedList
  
  ```java
  ArrayList adjList = new ArrayList();
  foreach neighbor
    adjList.add(the neighbor);
  ```
- Remember to cast when retrieving an element in the list!
  
  ```java
  Node n = (Node) adjList.get(2); // get second neighbor
  ```
The most common data structure for a graph is a map from a node or node's key to it's adjacency list.

Typically a HashMap is used.
Node Abstraction

- Previous we assumed the Node object contained the key and the data, and the graph was mapping between a node and it's adjacency list, which was a list of Node objects.
- Frequently you don't want or can't store all the information in the actual graph.
- In this case, the graph is a mapping between the keys to adjacency list of just keys.
- The map is String to a List of Strings.
Comparison

- Full Node Objects

  Node("Mary", age=30) mapsTo
  {Node("Fred", age=10),
   Node("Alice", age 22),
   Node("Bob", age 43)}

- Pure Keys. Just uses Strings.
  "Mary" mapsTo "Fred", "Alice", "Bob"

- Sorry this is a terrible slide
If represent a graph just using keys you need some way of turning the keys into more useful data.

- Another map (keys to DataObjects)
- Or in the database
  
  “select * from table where key=Mary”
Embedded Adjacentcy Lists

- One can also add the adjacentcy list into the node itself.
  ```java
  public class Node {
    public String key;
    public Node[] neighbors;
    ...
  }
  ```

- Maybe ok in some cases, but I don't like mixing the data structure of the graph in with the data structure of the node.
Adjacency Matrix

Who's adjacent to who is represented as a big matrix, with a “1” indicated a connection.

```
  a b c d e
a 0 0 1 0 1
b 0 0 0 0 0
c 1 0 0 0 0
d 0 0 0 0 0
e 1 0 0 0 0
```

ex: a is connected to c and e

Center diagonal is normally all zeros. Normally nodes aren't self-connected. “Lower triangle” is redundant in undirected graphs.
More on Matrix

- Typically not as efficient as using plain adjacency lists.
- Is used or preferred in some algorithms.
- Maybe useful, since the matrix form is always the same size in memory regardless of how connected the graph is.
- If you need to use this form, take a look at java.util.BitSet.
Graph Abstraction

- Regardless of what internal representation you decide on, the user of the graph shouldn't care.
- What should a graph class do?
  - Add or delete an edge
  - Add a node
  - Get adjacent nodes
  - Delete a node (and remove it's edges)
  - Report the number of nodes and edges
  - Provide a textual representation (toString)
  - (bonus: be able to dump and read itself to disk)
A Simple Graph Interface

```java
import java.util.ArrayList;
public interface Graph {
    public getAdjacent(String key, ArrayList list)
    public ArrayList setAdjacent(String key)
    public int nodes();
    public int edges();
    public void addNode(String key);
    public void deleteNode(String key);
    public void addEdge(String key1, String key2);
    public String toString(); // dump it to text
    public int hashCode(); // ??
    public boolean equals(); // ??
}
```