README.md

Session 9

Table of Contents

The Object Class, equals and hashCode

1. The Object class
2. hashCode
3. equals

The Collections Framework - Part 2

Maps

Maps are know by various names in different contexts/languages; dictionaries, associative arrays, symbol table, etc. It is simply a **collection** of key-value pairs.

Maps are not part of the Iterable-Collection *hierarchy*, but they are still part of the framework.

A Map is internally implemented as an array of buckets. Each buckets is in itself a linked list (though this detail can change for the sake of efficient, e.g. a

bst instead of a linked list).

Each key-value pair in the bucket is stored as an Entry. The entry class looks something like this:

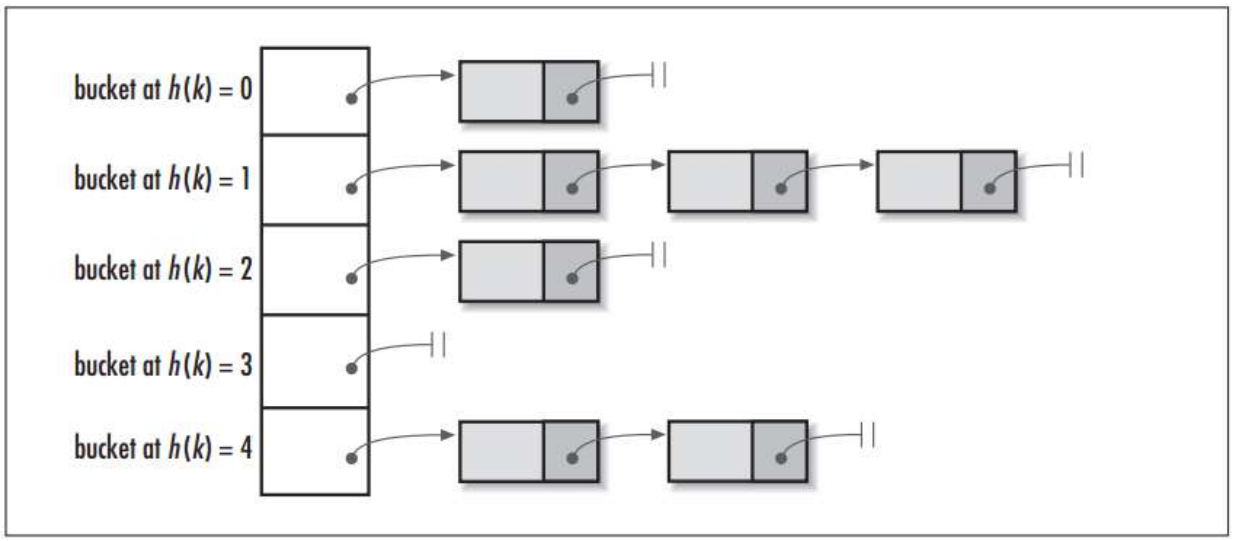
```
private static class Entry<K,V> implements Map.Entry<K,V> {
    final K key;
    final int hash;
    V value;
    Entry<K,V> next;
}
```

You can clearly make out the linked list nature of the Entry class because of the Entry<K,V> next field.

There are two main operations that you typically perform on a Map:

1. put(key, value)
   When you call the put method, the following steps occur:
   i. The hash of the key is calculated using the hashCode() method
   ii. An index i is derived from the resulting hash.
   iii. An instance of the Entry class (e) is created, with the relevant fields.
   iv. If there is no entry at the array location i, the instance e is set at the location.
   v. If these *is* an entry at the array location i, then:
      a. If the key passed to the put method as an argument is equal to the key of the existing entry, then the value is updated, and the old value is returned.
      b. If the key passed to the put method as an argument is *not* equal to the key of the existing entry, then the instance e is appended to the linked list.
2. get(key)
   When you call the get method, the following steps occur:
   i. The hash of the key is calculated using the hashCode() method
   ii. An index i is derived from the resulting hash.
   iii. A linear search for the key is conducted in the linked list at the location i, with equals being used to determine if the element is found or not.
   iv. If no element exists in the Map with this key, null is returned.

A simple visualization of the internal structure of the HashMap is:



Sample usage:

```
// A map from name to house
HashMap<String, String> houseMap = new HashMap<>();
houseMap.put("jaime", "lannister");
houseMap.put("robert", "baratheon");
houseMap.put("theon", "greyjoy");
houseMap.put("bran", "stark");
houseMap.put("ned", "stark"); // RIP

System.out.println(houseMap.get("theon"));    // greyjoy
System.out.println(houseMap.get("thoros"));   // null
```

As usual, one of the best places to understand Maps is the JavaDocs

Problems

1. https://www.codechef.com/problems/MIME2

Graphs

A graph is a set of finite set of vertices, along with edges that connect these vertices.

Graphs are a vast topic, with dozens of algorithms that can be performed on them.

Applications

1. Social Networks
2. Geographical Data (Google Maps, Air Navigation Routes)
3. Networking (Routers, Switches, Hubs)
4. Databases (Neo4J)
5. Web Scraping (PageRank)
6. Gaming (Path solving)

Classification

Graphs can be categorized on the basis of

1. Weightedness
    i. Unweighted - All edges have the same weight / unit weight.
    ii. Weighted - Each edge can have a different weight.
2. Directedness
    i. Undirected - If u is reachable from v, then vice versa is also true.
    ii. Directed - If u is reachable from v, then v may or may not be reachable from u, i.e. each edge has a direction.

Representation

Graphs can be represented either by means of

1. Adjacency Matrix - A 2D matrix that holds the weight from every ith vertex to every jth vertex. Infinity implies the vertices are not connected.

```
class Graph {
    int[][] adjacencyMatrix;

    public boolean isAdjacent(int u, int v) {
        return adjacencyMatrix[u][v] != Integer.MAX_VALUE;
    }

    ...
}
```

2. Adjacency Lists - Each Vertex maintains a list of its adjacent vertices

```
class Vertex {
   Set<Vertex> neighbors;

   public Set<Vertex> getAdjacentVertices() {
      return new HashSet<>(neighbors);
   }
}

class Graph {
   Set<Vertex> allVertices;

   public boolean isAdjacent(Vertex u, Vertex v) {
      return u.getAdjacentVertices().contains(v);
   }
}
```

Terminology

1. Indegree - The number of vertices from which this vertex is reachable
2. Outdegree - The number of vertices reachable from this vertex
3. Sink Vertex - A vertex with outdegree = 0

Traversals

1. Breadth First Search

   This is typically used to find the shortest path from one node to another, or to search for a particular node. The following psuedo code is a general piece of code that you can use for any application of the algorithm

   ```
   bfs(Graph g, source s)
      Let Q be a queue
      Let visited be a set

      Q.enqueue(s)
      put s into the visited set

      while Q is not empty:
         current = Q.dequeue()
         for all neighbors x of current:
            Q.enqueue(x)
            put x into the visited set
   ```

   BFS will always visit vertices at a particular depth i before all vertices at depth > i.

2. Depth First Search

   This is typically used for backtracking algorithms. One of the most popular problems solved by using DFS is the n-Queen problem.

   ```
   dfs(Graph g, source s)
      Let St be a stack
      Let visited be a set

      St.push(s)
      Put s into the visited set

      while St is not empty:
         current = St.pop()
         for all neighbors x of current:
            St.push(x)
            put x into the visited set
   ```

Note that both the algorithms are identical, except for a single data structure choice (queue vs stack)

Problems

1. One of the constraints on using an adjacency matrix as a choice of representation of a graph is that the vertices have to be numbered. Can you think of a similar solution if the vertices are not numbered (A, B, C instead of 1,2,3)?
2. Come up with at least three examples for each combination of weighted/unweighted and directed/undirected graphs, i.e. you should have a total of 12 examples. (You're free to use Google search, the idea is to be aware of various applications)
3. Implement the BFS algorithm for an unweighted, undirected graph. It should accept an adjacency matrix, a source vertex, and a goal vertex. It should print out three things: a. Whether or not the goal vertex is reachable from the source vertex b. The distance to be traversed to reach the goal vertex (if the goal vertex is indeed reachable) c. The shortest path from the source vertex to the goal vertex.

   Write your code step by step, using the pseudo code above as a starting point. Think and reason about whether your algorithm would work for directed graphs as well. Then try it out.
4. Implement the DFS algorithm for an unweighted, undirected graph, to determine whether or not there are cycles in the graph. (Hint: you will ALWAYS encounter a vertex that is already on the stack in case there is a cycle)
5. Study and implement Djikstra's algorithm for the shortest path in a weighted, directed graph. (It is pronounced as Dyke-stra)

An Extensive List of Graph Topics

Go through the topic list for the Graph section at GeeksForGeeks, and read the following topics thoroughly:

1. Introduction, DFS and BFS
2. Detecting cycles
3. Topological sorting
4. Minimum Spanning Tree (Either of Prim/Kruskal should suffice)
5. n-Queen and Knight's tour problems
6. m Coloring problem
7. Finding the number of islands (This can be rephrased in a lot of different ways, for example, see this)

Assignments

HackerRank

1. https://www.hackerrank.com/challenges/phone-book
2. https://www.hackerrank.com/challenges/bfsshortreach
3. https://www.hackerrank.com/challenges/the-quickest-way-up

Miscellaneous

1. Companies have several positions that an employee can be at. Employees at the same position get the same salary. Given a list of employees and their salaries, you have to determine how many positions are there at a given company. The input spec is as follows:
   i. The first line is a number (say N)
   ii. N lines follow. Each line contains the name of an employee, and his salary (space separated). The salary may range from 10^5 to 10^9.
   iii. Display the number of positions at this company
2. You have probably heard of the mobile application called TrueCaller. By looking at a phone number, it tells you who is calling. It works like a reverse look-up phone directory. You have been asked to implement this application.
   i. The first line is a number (say N)
   ii. N lines follow
   iii. Each line contains a String denoting the name of a person (which may have spaces) and his/her number in a space separated manner
   iv. Store the names and numbers
   v. Now start a menu-driven flow to:

a. Accept a number from the user
    a. If the number has a name associated with it, display it
    b. If it does not, inform the user that it does not exist, and give him the option to provide a name. Store the name and number.
b. Display the entire reverse mapping, in the following format:
    a. Number1: Name
    b. Number2: Name
c. Change a particular phone number's owner
d. Quit the program