Programming 2 Revision

Jonathan Windle

University of East Anglia

J.Windle@uea.ac.uk

May 27, 2017

Overview I

ArrayList

Using sets

- Contains no duplicate elements in the collection.
- Order of items also doesn't matter.

```java
ArrayList<String> myList = new ArrayList<>();
myList.add("Bob");
myList.add("Bob"); // Won't add to the set
myList.add("Alice");
myList.add("Fred");
// Can use a List as a constructor argument
Set<String> mySet = new HashSet<>(myList); // Size is 3

if(mySet.add("Alice"))// Returns false as Alice already in set
        // Do things
```

Set Implementations

- `TreeSet`
  - Stores the elements in a red-black tree, orders the elements based on their values.
- `HashSet`
  - Stores elements in a `HashTable`
- `LinkedHashSet`
  - Implemented as a `HashTable` with a linked list running through it, orders elements in insertion order.

TreeSet

- Stores elements in a `Tree` structure in order to maintain the elements in order.
- `TreeSet` can be used with `Comparable` classes or by providing a `Comparator`.
- These are useful when you need to extract elements from a set in a sorted manner.
- `first()` - returns the smallest element.
- `pollFirst()` which removes and returns the smallest element.
- Search, insert and delete takes $O(logn)$ time.
- Iterating in sorted order is $O(n)$ time iterating inorder traversal.

HashSet

- Stores the elements in a Hash Table.
- Allows user to set initial capacity and load factor.
- `HashSet` just contains a `HashMap` with keys defined by the `hashCode()`.
- `HashSet` is generally the `Set` you would use unless you need to access the data in sorted order.
- Insert, delete and contains are all $O(1)$ time.

HashSet - Behind the scenes

1. Call the `hashCode()` function.
2. Apply the `hash()` function to find the index.
3. If the position index is empty, add to the head of the list, otherwise find the end of the list and add there.

- After the `hash` is complete it is then bitwise & with $arrLength - 1$ where $arrLength$ is a power of two.
- 71638%8 is equivalent to 71638&7 and compensates for negative hash codes.

HashSet notes

- Two parameters that affect its performance:
  - initial capacity (Default 16, increases to nearest power of 2).
  - load factor (Default to 0.75). This determines when to resize the array, e.g. with capacity 16, when the 12th element is added, the array is resized, when this happens:
    - All indeces are recalculated
    - Chains are removed/reduced.
    - This is expensive.
- The performance of the `hash` function has a large effect on the overall performance.
- As chains get long, performance decreases to $O(m)$
- Extra efficiency is obtained at the cost of memory.
- Useful if you want to query a lot, using `contains`.

LinkedHashSet

- Is the same as HashSet, however includes a doubly linked list. This means it can be traversed in insertion order too.

Map

- Allows you to associate a key with one or more values and then quickly retrieve those values using the key.
- A map cannot contain duplicate keys: Each key can map to at most one value. **It can contain duplicate values**.
- A set is simply a map with the key equal to the value.

Set : {**Fred, Alice, Bob**}
Map: {**(1,Fred), (2,Alice), (3,Bob)**}
Set as a Map: {**(Fred, Fred), (Alice, Alice), (Bob, Bob)**}

- Maps are a collection of `Entry` objects, these store both the key and the value for a given map entry.

HashMap

```java
// String key and Student value
HashMap<String, Student> hm = new HashMap<>();
hm.put("BobKey", new Student(33, "Bob"));
```

1. A new `Entry` object is created containing the key and the value.
2. The `hashCode` function is called on the key to find the location in the hash table.

`HashSet` is just a `HashMap` where the value is used as the key.

TreeMap

- Behaves just as the `TreeSet` class, except the key is used to determine the location of the value.

1. The key must be comparable.
2. the `compareTo` method is used to insert the entry to the correct position.
3. `TreeSet` simply contains a `TreeMap` with the values set to the key.

Queues and Deques

- Java uses a linked list for queues and deques.
- Priority queues store elements in a heap data structure (a complete binary tree).
- Duplicates are allowed.
- $O(logn)$ time for insertion methods (`offer`,`poll`, `remove` and `add`)
- $O(n)$ time for `remove(Object)` and `contains`
- (1) time for retrieval methods, `peek`, `element` and `size`.
- Iterator not guaranteed to traverse in any order.

- Useful methods:
  - `frequency(Collection<?>c, Object o)` - Returns occurences of o in c.
  - `rotate(Collection<?> list, int distance)` - Shifts all the elements right in a logical circular way.
  - `shuffle(List<?>, list)` - Performs n normal swaps.
- Sorting:
  - `sort(List<T> list)` - A modified merge sort.
  - Mergesort that does not perform the merge operation if not required.
  - It dumps the specified list into an array,sorts the array, and iterates over the list resetting each element from the corresponding position in the array.
  - This avoids the $n^2 log(n)$ performance that would result from attempting to sort a linked list in place.

The End