**PROGRAMMING PATTERNS**

420-301-VA

ASSIGNMENT-3-

## Algorithms Efficiency

Review the Algorithm efficiency material and exercises then answer the questions.

1. Express in terms of Big O notation the running time of the following implementations and give a brief explanation:

|  |
| --- |
| public static void print(int n) {  for (int i = n; i >=1; i = i /2) {  System.out.println(i);  }  } |

|  |
| --- |
| O(log n) because we are reducing i by half after each iteration resulting in a log base 2 of n |

|  |
| --- |
| public static void print(int n) {  // Halving the input in each iteration.  for (int i = 1; i < n; i = i \* 2) {  System.out.println(i);  }  } |

|  |
| --- |
| O(log n) because we are doubling i after each iteration resulting in a log base 2 of n |

|  |
| --- |
| public static void logLinearTime(int[] arr) {  for (int i = 0; i < arr.length; i++) {  int temp = arr[i];  for (int j = 1; j < temp; j = j \* 2) {  System.out.println(j);  }  }  } |

|  |
| --- |
| O(n log n) because the outer loop itself is O(n) and the inner loop is O(log n). The product of both complexities result in O(n log n) |

1. Calculate the below for n=4, show the summation of the different terms and then the result:

|  |
| --- |
| i=0  (0+1)/2 = 0.5  i=1  (1+1)/2 = 1  i=2  (2+1)/2 = 1.5  i=3  (3+1)/2 = 2  i=4  (4+1)/2 = 2.5  0.5+1+1.5+2+2.5 = 7.5 |

## Java Collections

In Java, and other languages, the Classes that implement data structures are called collections.

Collections Classes are defined as Generic Types, thus they are called Generic Collections.

**Collections Interfaces in Java**

|  |  |
| --- | --- |
| **Interface** | **Description** |
| **Collection** | Enables you to work with groups of objects; it is at the top of the collections hierarchy. |
| **Deque** | Extends **Queue** to handle a double-ended queue. |
| **List** | Extends **Collection** to handle sequences (lists of objects). |
| **NavigableSet** | Extends **SortedSet** to handle retrieval of elements based on closest-match searches. |
| **Queue** | Extends **Collection** to handle special types of lists in which elements are removed only from the head. |
| **Set** | Extends **Collection** to handle sets, which must contain unique elements. |
| **SortedSet** | Extends **Set** to handle sorted sets. |

**Collections Classes that implement the interfaces**

|  |  |
| --- | --- |
| **Class** | **Description** |
| AbstractCollection | Implements most of the **Collection** interface. |
| AbstractList | Extends **AbstractCollection** and implements most of the **List** interface. |
| AbstractQueue | Extends **AbstractCollection** and implements parts of the **Queue** interface. |
| AbstractSequentialList | Extends **AbstractList** for use by a collection that uses sequential rather than random access of its elements. |
| LinkedList | Implements a linked list by extending **AbstractSequentialList.** |
| ArrayList | Implements a dynamic array by extending **AbstractList.** |
| ArrayDeque | Implements a dynamic double-ended queue by extending **AbstractCollection** and implementing the **Deque** interface. |
| AbstractSet | Extends **AbstractCollection** and implements most of the **Set** interface. |
| EnumSet | Extends **AbstractSet** for use with **enum** elements. |
| HashSet | Extends **AbstractSet** for use with a hash table. |
| LinkedHashSet | Extends **HashSet** to allow insertion-order iterations. |
| PriorityQueue | Extends **AbstractQueue** to support a priority-based queue. |
| TreeSet | Implements a set stored in a tree. Extends **AbstractSet.** |

You will implement some samples on collections that you haven’t used before so that you familiarize yourself with these collections and you are able to select the appropriate one based on a scenario.

#### **The HashSet Class**

**HashSet** extends **AbstractSet** and implements the **Set** interface. It creates a collection that uses a hash table for storage. **HashSet** is a generic class that has this declaration:

class HashSet<E>

Here, **E** specifies the type of objects that the set will hold.

As most readers likely know, a hash table stores information by using a mechanism called hashing. In *hashing*, the informational content of a key is used to determine a unique value, called its *hash code*. The hash code is then used as the index at which the data associated with the key is stored. The transformation of the key into its hash code is performed automatically—you never see the hash code itself. Also, your code can’t directly index the hash table. The advantage of hashing is that it allows the execution time of **add( )**, **contains( )**, **remove( )**, and **size( )** to remain constant even for large sets.

The following constructors are defined:

HashSet( )

HashSet(Collection<? extends E> c)

HashSet(int *capacity*)

HashSet(int *capacity*, float *fillRatio*)

The first form constructs a default hash set. The second form initializes the hash set by using the elements of *c*. The third form initializes the capacity of the hash set to *capacity*. (The default capacity is 16.) The fourth form initializes both the capacity and the fill ratio (also called *load factor*) of the hash set from its arguments. The fill ratio must be between 0.0 and 1.0, and it determines how full the hash set can be before it is resized upward. Specifically, when the number of elements is greater than the capacity of the hash set multiplied by its fill ratio, the hash set is expanded. For constructors that do not take a fill ratio, 0.75 is used.

**HashSet** does not define any additional methods beyond those provided by its superclasses and interfaces.

It is important to note that **HashSet** does not guarantee the order of its elements, because the process of hashing doesn’t usually lend itself to the creation of sorted sets. If you need sorted storage, then another collection, such as **TreeSet**, is a better choice.

### Use Case: filtering unique values

#### Assume that we have the list of strings with duplicate values, add the products to a HashSet and then print out the HashSet and interpret the result:

* + “product 1”, “product 2”, product 3”, product 1”, “product 2”, “product 5”

The content of the HashSet is:

|  |
| --- |
| [Product 1, Product 2, Product 3, Product 5] |

#### Assume that we have the products in anArrayList, use the HashSet(Collection<? extends E> c) constructor to store only the unique values in the HashSet and then print out the HashSet and interpret the result:

* + “product 1”, “product 2”, product 3”, product 1”, “product 2”, “product 5”

The content of the HashSet is:

|  |
| --- |
| [Product 1, Product 2, Product 3, Product 5] |

#### What is the running time of removing the duplicates in the ArrayList by updating the ArrayList itself, expressed in Big O notation?:

|  |
| --- |
| O(n^2) |

#### What is the running time of removing the duplicates by adding the elements from the ArrayList into the HashSet, expressed in Big O notation?:

|  |
| --- |
| O(n) |

1. **HashMap**

The **HashMap** class extends **AbstractMap** and implements the **Map** interface. It uses a hash table to store the map. This allows the execution time of **get( )** and **put( )** to remain constant even for large sets. **HashMap** is a generic class that has this declaration:

class HashMap<K, V>

Here, **K** specifies the type of keys, and **V** specifies the type of values.

The following constructors are defined:

HashMap( )

HashMap(Map<? extends K, ? extends V> *m*)

HashMap(int *capacity*)

HashMap(int *capacity*, float *fillRatio*)

The first form constructs a default hash map. The second form initializes the hash map by using the elements of *m*. The third form initializes the capacity of the hash map to *capacity*. The fourth form initializes both the capacity and fill ratio of the hash map by using its arguments. The meaning of capacity and fill ratio is the same as for **HashSet**, described earlier. The default capacity is 16. The default fill ratio is 0.75.

**HashMap** implements **Map** and extends **AbstractMap**. It does not add any methods of its own.

You should note that a hashmap does not guarantee the order of its elements. Therefore, the order in which elements are added to a hash map is not necessarily the order in which they are read by an iterator.

### Use Case: maintain a frequency map which stores an element and its occurrence count

A common and practical use for a HashMap is to count the frequency of items in a collection, such as counting the occurrences of each word in a string. This is also known as a word counter or frequency map

Assume that we have a text and we want to log the repeated words and their count, for example the text:

“I'll dance dance dance With my hands hands hands Above my head head head”

* Store all the words of the text in a HashMap where each key is the word itself and the value is the count of the word in the text. Note that you can loop over the HashMap and use the getOrDefault() method to get the value to which the specified key is mapped (which is the count), or 0 if this map contains no mapping for the key, then each time increment the value by using the put() method repeatedly.

Implement and test the program.

The output is:

|  |
| --- |
| word: head, counted: 3x  word: with, counted: 1x  word: hands, counted: 3x  word: above, counted: 1x  word: my, counted: 2x  word: dance, counted: 3x  word: i'll, counted: 1x |