**SortedSet and SortedMap interfaces**

Java 5.0’s SortedSet interface extends Set by providing the user with an ordered view of the elements with the ordering defined by a compareTo method

Because the elements are ordered, additional methods can return the first and last elements and define subsets

The ability to define subsets was limited because subsets always had to include the starting element and exclude the ending element

SortedMap interface provides an ordered view of a map with elements ordered by key value

Hashing is not used to implement SortedSet and SortedMap because there is no ordering in the hash table.

TreeSet is implementation of SortedSet, TreeMap is implementation of SortedMap.

**NavigableSet and NavigableMap interfaces**

Java 6 added NavigableSet and NavigableMap interfaces as extensions to SortedSet and SortedMap

Java retains SortedSet and SortedMap for compatibility with existing software

The new interfaces allow the user to specify whether the start or end items are included or excluded

They also enable the user to specify a subset or submap that is traversable in the reverse order

NavigableSet Interface

Text

Description automatically generated

Calendar

Description automatically generated with low confidence

Table

Description automatically generated

descendingSet creates a view of the set that we have in our hand.

headset, tailSet, subset also create a view.

View is not a separate set.

NavigableMap Interface

Table

Description automatically generated

**Classes TreeMap and TreeSet**

Besides HashMap and HashSet, the Java Collections Framework provides classes TreeMap and TreeSet

TreeMap and TreeSet use a Red-Black tree, which is a balanced binary search tree

Search, retrieval, insertion and removal are performed better using a hash table (expected O(1)) than using binary search tree (O(logn))

However, a BST can be traversed in sorted order while a hash table cannot be traversed in any meaningful way

Application of a NavigableMap Interface

Given a NavigableMap in which the keys represent years and the values are some statistics for the year, we can generate a table of averages covering different periods

computeAverage computes the average of the values defined in a Map

computeSpans creates a group of submaps of a NavigableMap and passes each submap to computeAverage

Example: Given a map of tropical storms representing the number of tropical storms from 1960 to 1969

List<Number> stormAverage = computeSpans(storms, 2)



Calculates the average number of tropical storms for each successive pair of years

computeSpans will return list of average values

1960-1961 | 1962-1963 | 1964-1965 | 1966-1967 | 1968-1969

Graphical user interface, text, application, letter, email

Description automatically generated

Text, letter

Description automatically generated

Additional Applications of Maps

Problem:

* A cell phone manufacturer wants a Java program to maintain of list of contacts (phone numbers) for each cell phone owner (<Name, PhoneNumber>)
* The manufacturer has provided the software interface

Table

Description automatically generated

addOrChangeEntry 🡪 very similar to put in Map  
lookUpEntry 🡪 very similar to get in Map  
removeEntry 🡪 very similar to remove in Map

So these methods can be easily delegated to Map methods.

Analysis

A map will associate the name (the key) with a list of phone numbers (value)

Implement ContactListInterface by using a Map<String, List<String>> object for the data type

Design

Text

Description automatically generated

Implementation

Writing the required methods using the Map methods is straightforward

Testing

Write a main function that creates a new MapContactList object

Apply the addOrChangeEntry() method several times with new names and numbers to build the initial contact list

Display and update the list to verify that all methods are functioning correctly

CHECK BOOK PAGE 360