1.1

Explain the effect of the following method calls.

Set<String> s = new HashSet<String>();

s.add("hello");

s.add("bye");

s.addAll(s);

Set<String> t = new TreeSet<String>();

t.add("123");

s.addAll(t);

System.out.println(s.containsAll(t));

System.out.println(t.containsAll(s));

System.out.println(s.contains("ace"));

System.out.println(s.contains("123"));

s.retainAll(t);

System.out.println(s.contains("123"));

t.retainAll(s);

System.out.println(t.contains("123"));

Set<String> s = new HashSet<String>();

Creates a HashSet object to hold Strings and assignes a reference to it to the variable s.

s.add("hello");

Adds the string "hello" to the set.

s.add("bye");

Adds the string "bye" to the set.

s.addAll(s);

Copies the content of s into itself. Effectively doing nothing since sets do not contain duplicate values.

Set<String> t = new TreeSet<String>();

Creates a TreeSet object to hold Strings and assignes a reference to it to the variable t.

t.add("123");

Adds the string "123" to t.

s.addAll(t);

Adds the contents of t to s. s now contains {"hello", "bye", "123"}

System.out.println(s.containsAll(t));

Outputs "true" since all of the elements in t are included in s.

System.out.println(t.containsAll(s));

Outputs "false" since there are elements in s that are not in t.

System.out.println(s.contains("ace"));

Outputs "false" since "ace" is not in s.

System.out.println(s.contains("123"));

Outputs "true" since "123" is contained in s.

s.retainAll(t);

Removes the elements from s that are not in t.

System.out.println(s.contains("123"));

Outputs "true" since "123" is still in s.

t.retainAll(s);

Effectively nothing, since s now only contains the values in t.

System.out.println(t.contains("123"));

Outputs "true" since "123" is still in t.

1.3

What are the differences between the Set interface and the List interface?

Elements in a List have a position (index) and the List interface contains methods to access element using an index. Also the requirements on the equals and hashCode methods are different. The List also defines a ListIterator which allows bi-directional traversal.

2.1

If you were using a Map to store the following lists of items, which data field would you select as the key, and why?

**a.** textbook title, author, ISBN (International Standard Book Number), year, publisher

**b.** player’s name, uniform number, team, position

**c.** computer manufacturer, model number, processor, memory, disk size

**d.** department, course title, course ID, section number, days, time, room

a. ISBN Number. There may be multiple textbooks with the same year, publisher, year, or author, but the ISBN would be unique to the particular book.

b. There is not good key. None of these are unique.

c. Model number, because it is unique, but there could be many computer models with the same manufacturer, processor, memory, or disk size.

d. Course ID, because many courses can have the same department, title, section number, days, time, or room.

2.3

Explain the effect of each statement in the following fragment on the index built in Self-Check Exercise 2.

lines = index.get("this");

lines = index.get("that");

lines = index.get("line");

lines.add(4);

index.put("is", lines);

lines = index.get("this");

lines references the list [1]

lines = index.get("that");

lines is null.

lines = index.get("line");

lines references the list [1, 2, 3]

lines.add(4);

The value of 4 is added to the list referenced by lines and indexed by "line", it is now [1, 2, 3, 4]

index.put("is", lines);

The list [1, 2, 3, 4] is also indexed by "is". Note "line" and "is" both are paired with the same list.

3.1

For the hash table search algorithm shown in this section, why was it unnecessary to test whether all table entries had been examined as part of Step 5?

The table is never allowed to become full, therefore, the loop will always terminate by finding the requested item or by finding a null entry.

3.3

The following table stores Integer keys with the int values shown. Show one sequence of insertions that would store the keys as shown. Which elements were placed in their current position because of collisions? Show the table that would be formed by chaining.

|  |  |
| --- | --- |
| Index | Key |
| [0] | 24 |
| [1] | 6 |
| [2] | 20 |
| [3] |  |
| [4] | 14 |

Inserting in the order 6, 14, 24, 20 would result in the order shown. 24 was placed in it position because of a collision with 14. 20 is then placed in position [2] because of collision with 24 and 6.

If chaining were used the table would look as follows:

|  |  |
| --- | --- |
| [0] | 20 |
| [1] | 6 |
| [2] |  |
| [3] |  |
| [4] | 14 🡪 24 |

3.5

Explain what is wrong with the following strategy to reclaim space that is filled with deleted items in a hash table: When attempting to insert a new item in the table, if you encounter an item that has been deleted, replace the deleted item with the new item.

Assume that we take the table given in Example 7.7 and delete "Dick", marking it with "deleted". Next we try to re-insert "Sam" The search for "Sam" will start at [4] and then advance to [0] finding the "deleted" flag. We would then insert "Sam" at that position, resulting in duplicate values in the hash table.

3.7

One simple hash code is to use the sum of the ASCII codes for the letters in a word. Explain why this is not a good hash code.

Many word contain the same letters, but in a different order. For example “cat” and “tac”. These would have the same hash code.

3.9

Use the hash code in Self-Check Exercise 7 to store the words "cat", "hat", "tac", "act" in a hash table of size 10. Show this table using open hashing and chaining.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Letter | Code | Letter | Code | Letter | Code | Letter | Code |
| c | 99 | h | 104 | t | 116 | a | 97 |
| a | 97 | a | 97 | a | 97 | c | 99 |
| t | 116 | t | 116 | c | 99 | t | 116 |
| SUM | 312 |  | 317 |  | 312 |  | 312 |

|  |  |  |
| --- | --- | --- |
| Word | Code | Code % 10 |
| cat | 312 | 2 |
| hat | 317 | 7 |
| tac | 312 | 2 |
| act | 312 | 2 |

|  |  |  |  |
| --- | --- | --- | --- |
| Position | Open Hashing |  | Chaining |
| 0 |  |  |  |
| 1 |  |  |  |
| 2 | cat |  | cat 🡪 tac 🡪 act |
| 3 | tac |  |  |
| 4 | act |  |  |
| 5 |  |  |  |
| 6 |  |  |  |
| 7 | hat |  | hat |
| 8 |  |  |  |
| 9 |  |  |  |

4.1

The following table stores Integer keys with the int values shown. Where would each key be placed in the new table resulting from rehashing the current table?

|  |  |
| --- | --- |
| Index | Key |
| 0 | 24 |
| 1 | 6 |
| 2 | 20 |
| 3 |  |
| 4 | 14 |
| Index | Key |
| 0 | 20 |
| 1 | 6 |
| 2 | 14 |
| 3 |  |
| 4 | 24 |

5.1

Explain why the nested interface Map.Entry is needed.

The interface Map.Entry is needed for the return value of the entrySet method. This allows one to access the contents of the map and traverse it easily. It is not strictly necessary since one could obtain the keySet and then iterate through it and obtain the corresponding value for each key, but this would be inefficient since a look-up of each key would then be required.

6.1

Why did we make clones of the bit string code in recursive method buildCodeTable? What would happen if we didn’t?

If we did not clone the code parameter passed to the buildCodeTable method the values leftCode and rightCode would reference the original value of code. When the append method was called it would then append the value to the original.

7.1

What is displayed by the execution of the following program?

public static void main(String[] args) {

NavigableSet<Integer> s = new TreeSet<Integer>();

s.add(5); s.add(6); s.add(3); s.add(2); s.add(9);

NavigableSet<Integer> a = s.subSet(1, true, 9, false);

NavigableSet<Integer> b = s.subSet(4, true, 9, true);

System.out.println(a);

System.out.println(b);

System.out.println(s.higher(5));

System.out.println(s.lower(5));

System.out.println(a.first());

System.out.println(b.lower(4));

int sum = 0;

for (int i : a) {

System.out.println(i);

sum += i;

}

System.out.println(sum);

}

[2, 3, 5, 6]

[5, 6, 9]

6

3

2

null

2

3

5

6

16