Lab walk-through #9 - 2	
Topic	**Searching Algorithms**
Week	9

1. Introduction

Modern computing and the internet have made accessible a vast amount of information. The ability to efficiently search through this information is fundamental to processing it.

2. Lab Objectives

By the end of this lab the student will learn how to implement and time various search algorithms.

3. Lab Setup

Before beginning this lab, you should have:

1. The Eclipse and Java Runtime Environment on your computer
2. Completed Lab Walk-through #9 – 1 and its practice questions
3. Read Chapter 3 in your textbook

4. Lab Exercise

Section 1: Hashing

Many problems to solve only require the key to be an integer. Therefore, it would make more sense to use an array to implement an unordered symbol table, by interpreting the key as an array index so that we can store the value associated with key i in array entry i, ready for immediate access. Hashing is an extension of this simple method that handles more complicated types of keys. We reference key-value pairs using arrays by doing arithmetic operations to transform keys into array indices. Hashing is a space-time trade off.

Search algorithms that use hashing consist of two separate parts. The first part is to compute a **hash function** that transforms the search key into an array index. Ideally, different keys would map to different indices. This ideal is generally beyond our reach, so we have to face the possibility that two or more different keys may hash to the same array index. Thus, the second part of a hashing search is a **collision-resolution** process that deals with this situation. This is usually solved by using linear probing or separate chaining.

If there were no memory limitation, then we could do any search with only one memory access by simply using the key as an index in a (potentially huge) array. On the other hand, if there was no time limitation, then we can get by with only a minimum amount of memory by using sequential search in an unordered array. Hashing provides a way to use a reasonable amount of both memory and time to strike a balance between these two extremes.

With hashing, you can implement search and insert for symbol tables that require constant (amortized) time per operation in typical applications, making it the method of choice for implementing basic symbol tables in many situations.

Creating a hash function

This function needs to transform keys into array indices. If we have an array that can hold M key-value pairs, then we need a hash function that can transform any given key into an index into that array: an integer in the range [0, M – 1]. We seek a hash function that both is easy to compute and uniformly distributes the keys: for each key, every integer between 0 and M–1 should be equally likely (independently for every key).

The hash function depends on the **key type**. We need a different hash function for each key type that we use. For many common types of keys, we can make use of default implementations provided by Java. Java ensures that every data type inherits a method called `hashCode()` that returns a 32-bit integer.

The implementation of `hashCode()` for a data type must be consistent with equals. That is, if `a.equals(b)` is true, then `a.hashCode()` must have the same numerical value as `b.hashCode()`. Conversely, if the `hashCode()` values are different, then we know that the objects are not equal. If the `hashCode()` values are the same, the objects may or may not be equal, and we must use `equals()` to decide which condition holds.

Since our goal is an array index, not a 32-bit integer, we combine hashCode() with **modular hashing** in our implementations to produce integers between 0 and M – 1, as follows:

```
private int hash(Key x)
     { return (x.hashCode() & 0x7fffffff) % M; }
```

This code masks off the sign bit (to turn the 32-bit number into a 31-bit nonnegative integer) and then computes the remainder when dividing by M. Use a prime number for the hash table size M when using code like this, to attempt to make use of all the bits of the hash code.

For user-defined types, you have to override and then implement `hashcode()` on your own. The hashing function might be expensive to calculate. Additionally, a bad hashing function is possible, that perhaps doesn't distribute keys evenly or takes even longer than array comparisons.

Collision-resolution using separate chaining

Because finding a perfect hash function is almost impossible, the algorithm needs a collision-resolution technique: a strategy for handling the case when two or more keys to be inserted hash to the same index.

A straightforward and general approach to collision resolution is to build, for each of the M array indices, a linked list of the key-value pairs whose keys hash to that index. This method is known as separate chaining because items that collide are chained together in separate linked lists. The basic idea is to choose M to be sufficiently large that the lists are sufficiently short to enable efficient search through a two-step process: hash to find the list that could contain the key, and then sequentially search through that list for the key.

Task 1: Implement Separate Chaining Hash Search Tree

Since we have M lists and N keys, the average length of the lists is always N/M, no matter how the keys are distributed among the lists. One way to proceed is to expand SequentialSearchST to implement separate chaining using linked-list primitives. Create the following class in your package:

```
public class SeparateChainingHashST<Key, Value>
{
	private int N;                    // number of key-value pairs
	private int M;                    // hash table size
	private SequentialSearchST<Key, Value>[] st;// array of ST objs

	public SeparateChainingHashST(){ this(997); }

	public SeparateChainingHashST(int M)
	{ // Create M linked lists.
		this.M = M;
		st = (SequentialSearchST<Key, Value>[]) new
SequentialSearchST[M];
		for (int i = 0; i < M; i++)
			st[i] = new SequentialSearchST();
	}

	private int hash(Key key)
	{ return (key.hashCode() & 0x7fffffff) % M; }

	public Value get(Key key)
	{ return (Value) st[hash(key)].get(key); }

	public void put(Key key, Value val)
	{ st[hash(key)].put(key, val); }

	public Iterable<Key> keys(){}
	// See Exercise 3.4.19.
}
```

This basic symbol-table implementation maintains an array of linked lists, using a hash function to choose a list for each key. For simplicity, we use SequentialSearchST methods. We need a cast when creating st[] because Java prohibits arrays with generics. The default constructor specifies 997 lists, so that for large tables, this code is about a factor of 1,000 faster than SequentialSearchST.

This quick solution is an easy way to get good performance when you have some idea of the number of key-value pairs to be put() by a client. A more robust solution is to use array resizing to make sure that the lists are short no matter how many key-value pairs are in the table (see page 474 and Exercise 3.4.18).

It is important to note that everything depends on the uniformity of the hash function being used. If the hash function is not uniform and independent, the search and insert cost could be proportional to N, no better than with sequential search. With hashing, we are assuming that each and every key, no matter how complex, is equally likely to be hashed to one of M indices.

Task 2: Implement Linear Probing Hashing

Another approach to implementing hashing is to store *N* key-value pairs in a hash table of size *M* > *N*, relying on empty entries in the table to help with collision resolution. Such methods are called *open-addressing* hashing methods.

The simplest open-addressing method is called *linear probing*: when there is a collision (when we hash to a table index that is already occupied with a key different from the search key), then we just check the next entry in the table (by incrementing the index).

We hash the key to a table index, check whether the search key matches the key there, and continue (incrementing the index, wrapping back to the beginning of the table if we reach the end) until finding either the search key or an empty table entry.

Create the following class in your package:

```
public class LinearProbingHashST<Key, Value>
{
      private int N;                         // number of key-value pairs
      private int M = 16;                    // hash table size
      private Key[] keys;                    // keys
      private Value[] vals;                  // values

      public LienarProbingHashST()
      {
            keys = (Key[])   new Object[M];
            vals = (Value[]) new Object[M];
      }

      private int hash(Key key)
      { return (key.hashCode() & 0x7fffffff) % M; }

      private void resize()                  // see page 474

      public void put(Key key, Value val)
      {
            if (N >= M/2) resize(2*M);

            int i;
            for (i = hash(key); keys[i] != null; i = (i + 1) % M)
                  if (keys[i].equals(key)) { vals[i] = val; return; }
            keys[i] = key;
            vals[i] = val;
            N++;
      }

      public Value get(Key key)
      {
            for (int i = hash(key); keys[i] != null; i = (i + 1) % M)
                  if (keys[i].equals(key))
                        return vals[i];
            return null;
      }
}
```

Task 3: Deleting a key-value pair from a linear-probing table.

How do we delete a key-value pair from a linear-probing table? If you think about the situation for a moment, you will see that setting the key's table position to null will not work, because that might prematurely terminate the search for a key that was inserted into the table later.

Implement the following delete method in your package:

```
public void delete(Key key)
{
      if (!contains(key)) return;
      int i = hash(key);
      while (!key.equals(keys[i]))
            i = (i + 1) % M;
      keys[i] = null;
      vals[i] = null;
      i = (i + 1) % M;
      while (keys[i] != null)
      {
            Key keyToRedo = keys[i];
            Value valToRedo = vals[i];
            keys[i] = null;
            vals[i] = null;
            N--;
            put(keyToRedo, valToRedo);
            i = (i + 1) % M;
      }
      N--;
      if (N > 0 || N == M/8) resize(M/2);
}
```

Section 2: Some Applications of Search Algorithms

The advantages of hashing over BST implementations are that the code is simpler and search times are optimal (constant), if the keys are of a standard type or are sufficiently simple that we can be confident of developing an efficient hash function for them that (approximately) satisfies the uniform hashing assumption. The advantages of BSTs over hashing are that they are based on a simpler abstract interface (no hash function need be designed) and they support a wider range of operations (such as rank, select, sort, and range search). As a rule of thumb, most programmers will use hashing except when one or more of these factors is important.

Set APIs Some symbol-table clients do not need the values, just the ability to insert keys into a table and to test whether a key is in the table. Because we disallow duplicate keys, these operations correspond to the following API where we are just interested in the *set* of keys in the table, not any associated values:

`public class SET<Key>`

	`SET()`	*create an empty set*
`void`	`add(Key key)`	*add* `key` *into the set*
`void`	`delete(Key key)`	*remove* `key` *from the set*
`boolean`	`contains(Key key)`	*is* `key` *in the set?*
`boolean`	`isEmpty()`	*is the set empty?*
`int`	`size()`	*number of keys in the set*
`String`	`toString()`	*string representation of the set*

API for a basic set data type

You can turn any symbol-table implementation into a SET implementation by ignoring values or by using a simple wrapper class.

Dictionary clients The most basic kind of symbol-table client builds a symbol table with successive *put* operations in order to support *get* requests. Many applications also take advantage of the idea that a symbol table is a *dynamic* dictionary, where it is easy to look up information *and* to update the information in the table. Dictionaries are characterized by the idea that there is one value associated with each key, so the direct use of our ST data type, which is based on the associative-array abstraction that assigns one value to each key, is appropriate.

For some examples, refer to page 492 of the textbook.

6. Further Practice Problems

For further practice, here are some suggested questions from your textbook:
3.4.9, 3.4.18, 3.4.19, 3.4.26