# Lab walk-through #6

| | |
|---|---|
| Topic | **Searching Algorithms** |
| Week | 7 – Session 1 |
| Week | 9 – Session 1 |

## 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 the Experimenting with Algorithms lab and its practice questions
3. Read Chapter 3 in your textbook

## 4. Lab Exercise
Search algorithms aim to find solutions or objects with specific properties and constraints in a search space or among a collection of objects. For our purposes, a solution will be a sub-structure of a given discrete structure of this search space. For example, if we use a linked list to store our information, then a solution will be a particular node of this linked list.

There are two types of information we want to store: a key and a value (do you see an ADT forming here?). A key is a piece of information to search by, and a value would be the information you would like to retrieve. For example, in a dictionary, one would search for a particular word (key) and retrieve its meaning (value). We will be using a structure called a symbol table to store this.

Search is so important to so many computer applications that symbol tables are available as high-level abstractions in many programming environments, including Java.

**Section 1: Symbol Tables**
A symbol table is described on page 362 in your textbook. It is a dictionary-type structure where a key (that we can search for) is stored along with its corresponding value (information we would like to retrieve). The type and nature of these keys and values depend on the application.

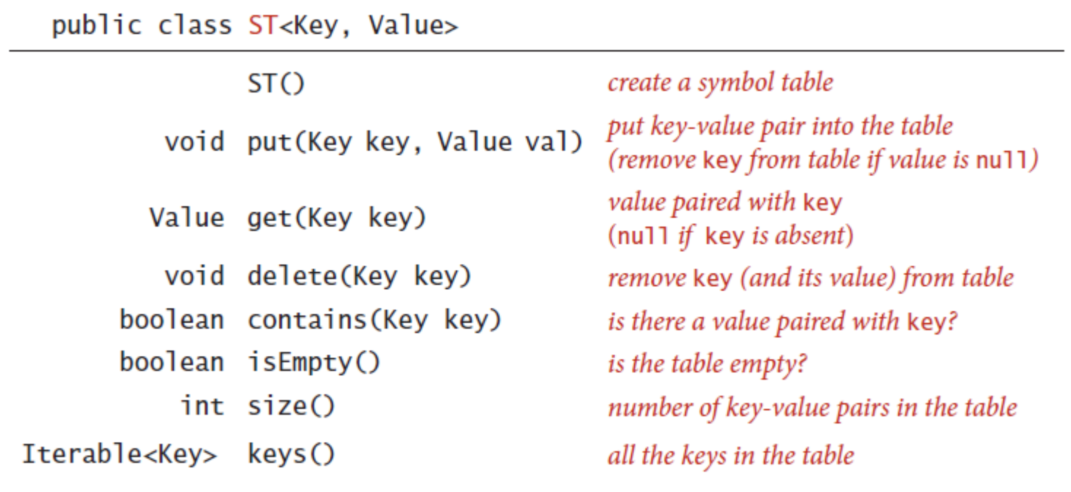
The primary purpose of a symbol table is to associate a value with a key. The client can insert key-value pairs into the symbol table with the expectation of later being able to search for the value associated with a given key, from among all of the key-value pairs that have been put into the table. To implement a symbol table, we need to define an underlying data structure and then specify algorithms for insert, search, and other operations that create and manipulate the data structure.

The symbol table is a prototypical abstract data type (Chapter 1 in your textbook): it represents a well-defined set of values and operations on those values, enabling us to develop clients and implementations separately. In a simple array the keys are indices and the values are array entries.

To start off:

1. Create a new Eclipse project called "cas2xb3_Lab6".
2. Create a package inside your src folder called "search". This is where we will implement our search algorithms.

We will be implementing the symbol table using the API on pages 363 and 366 of your textbook, depending on the problem being solved.



`public class ST<Key, Value>`

| | | |
|---|---|---|
| | `ST()` | *create a symbol table* |
| `void` | `put(Key key, Value val)` | *put key-value pair into the table (remove* `key` *from table if value is* `null`*)* |
| `Value` | `get(Key key)` | *value paired with* `key` *(*`null` *if* `key` *is absent)* |
| `void` | `delete(Key key)` | *remove* `key` *(and its value) from table* |
| `boolean` | `contains(Key key)` | *is there a value paired with* `key`*?* |
| `boolean` | `isEmpty()` | *is the table empty?* |
| `int` | `size()` | *number of key-value pairs in the table* |
| `Iterable<Key>` | `keys()` | *all the keys in the table* |

**API for a generic basic symbol table**

Take a careful look at this description. Notice the presence of the `Key extends Comparable<Key>` generic type variable in the class declaration, which specifies that the code depends upon the keys being Comparable and implements the richer set of operations. Together, these operations define for client programs an ordered symbol table. Find the API of this class on page 366 of your textbook.

An example client program for this symbol table could be:

```java
public static void main(String[] args)
{
    ST<String, Integer> st;
    st = new ST<String, Integer>();

    for (int i = 0; !StdIn.isEmpty(); i++)
    {
        String key = StdIn.readString();
        st.put(key, i);
    }

    for (String s : st.keys())
        StdOut.println(s + " " + st.get(s));
}
```

This client code above stores Strings as the keys and Integers as values. This is similar to what you needed in your Assignment 2 when you were storing words and scores as a pair, and needed to sort them. With a sorted symbol table, the extra step would be unnecessary and you could use the specialised functions to get the word with the maximum score. Other examples can be found on page 371 of your textbook.

The best uses of symbol tables are with problems that have the following characteristics:

1. Requires much more searching than insertion
2. The number of distinct keys are relatively large
3. Search and insert operations are jumbled
4. Search and insert operations are not random

Now we will implement these symbol tables using various searching techniques commonly used for searching in a number of applications. For each implementation, create a new class with the same methods that appear in the symbol table API.

**Section 2: Sequential Search**

One straightforward option for the underlying data structure for a symbol table is a linked list of nodes that contain keys and values. Create the following class:

```
public class SequentialSearchST<Key, Value>{
      private Node first; // first node in the linked list

      private class Node{ // linked-list node
            Key key;
            Value val;
            Node next;

            public Node(Key key, Value val, Node next){
                  this.key = key;
                  this.val = val;
                  this.next = next;
            }
      }
      public Value get(Key key){} // Search for key, return associated
      public void put(Key key, Value val){} // Search for key. Update
value if found; grow table if new.
}
```

This symbol table implementation uses a private inner class called Node to keep the keys and values in an unordered linked list. Let's take a look at what the get and put functions appear to be using sequential search:

```
public Value get(Key key){ // Search for key, return associated value.
      for (Node x = first; x != null; x = x.next)
            if (key.equals(x.key))
                  return x.val; // search hit
      return null; // search miss
}
```

To implement get(), we scan through the list, using equals() to compare the search key with the key in each node in the list. If we find the match, we return the associated value; if not, we return null.

```
public void put(Key key, Value val){ // Search for key. Update value
if found; grow table if new.
     for (Node x = first; x != null; x = x.next)
     if (key.equals(x.key))
          { x.val = val; return; } // Search hit: update val.

     first = new Node(key, val, first); // Search miss: add new node.
}
```

To implement put(), we also scan through the list, using equals() to compare the client key with the key in each node in the list. If we find the match, we update the value associated with that key to be the value given in the second argument; if not, we create a new node with the given key and value and insert it at the beginning of the list.

This method is known as sequential search: we search by considering the keys in the table one after another, and the order of the entries in the table is not considered.

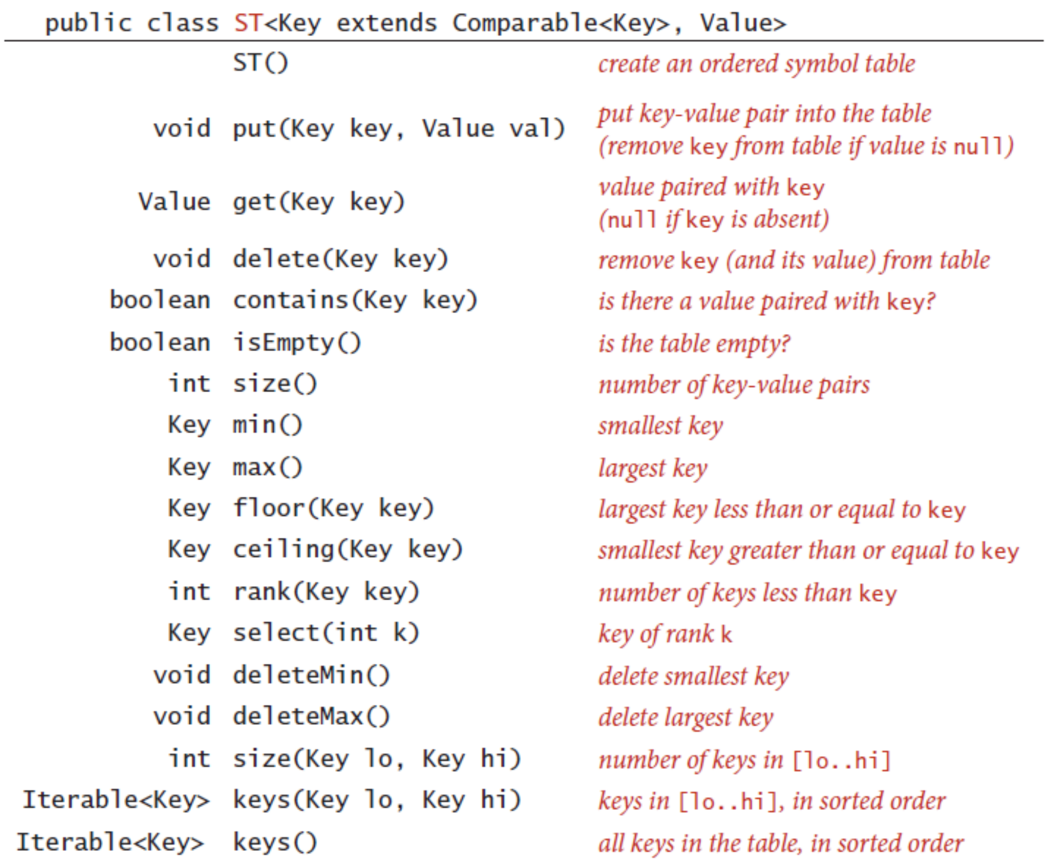
***Task 1:***
*Now implement the functions size(), keys() and delete() according to the API found on page 2 of this walkthrough.*
*Look at the implementation of these methods more carefully. What are the growth rates (complexity) for each method? Remember that put() first has to check whether the entry is already in the symbol table or not.*

**Section 3: Binary Search**
A binary search algorithm finds the position of a specified input key within an array. For binary search, the array should already be sorted.

In each step, the algorithm compares the search key with the key of the middle element of the array. If the keys match, then a matching element has been found and its value is returned. Otherwise, if the search key is less than the middle element's key, then the algorithm repeats its action on the sub-array to the left of the middle element or, if the search key is greater, on the sub-array to the right. If the remaining array to be searched is empty, then the key cannot be found in the array and a "not found" indication is returned.

A binary search halves the number of items to check with each iteration, so locating an item (or determining its absence) takes logarithmic time. A binary search is a dichotomic divide-and-conquer search algorithm.

For binary search, we will implement an ordered symbol table this time, which keeps the entries in the table sorted and has other useful methods implemented. The underlying data structure is a pair of parallel arrays, one for the keys and one for the values. The API for such a table can be found on page 366 of your textbook and is as follows:



`public class ST<Key extends Comparable<Key>, Value>`

| | | |
|---|---|---|
| | `ST()` | *create an ordered symbol table* |
| `void` | `put(Key key, Value val)` | *put key-value pair into the table (remove* `key` *from table if value is* `null`*)* |
| `Value` | `get(Key key)` | *value paired with* `key` *(*`null` *if* `key` *is absent)* |
| `void` | `delete(Key key)` | *remove* `key` *(and its value) from table* |
| `boolean` | `contains(Key key)` | *is there a value paired with* `key`*?* |
| `boolean` | `isEmpty()` | *is the table empty?* |
| `int` | `size()` | *number of key-value pairs* |
| `Key` | `min()` | *smallest key* |
| `Key` | `max()` | *largest key* |
| `Key` | `floor(Key key)` | *largest key less than or equal to* `key` |
| `Key` | `ceiling(Key key)` | *smallest key greater than or equal to* `key` |
| `int` | `rank(Key key)` | *number of keys less than* `key` |
| `Key` | `select(int k)` | *key of rank* `k` |
| `void` | `deleteMin()` | *delete smallest key* |
| `void` | `deleteMax()` | *delete largest key* |
| `int` | `size(Key lo, Key hi)` | *number of keys in* `[lo..hi]` |
| `Iterable<Key>` | `keys(Key lo, Key hi)` | *keys in* `[lo..hi]`*, in sorted order* |
| `Iterable<Key>` | `keys()` | *all keys in the table, in sorted order* |

**API for a generic ordered symbol table**

To implement this API, create a new class called BinarySearchST in the following way:

```
public class BinarySearchST<Key extends Comparable<Key>, Value>
{
	private Key[] keys;
	private Value[] vals;
	private int N;

	public BinarySearchST(int capacity){}

	public int size()
	{ return N; }

	public Value get(Key key){}
	public int rank(Key key){}
	public void put(Key key, Value val){}
	public void delete(Key key){}
	public boolean isEmpty(){}
}
```

This code maintains parallel arrays of keys and values. It also carries the inconvenience of having to create a Key array of type Comparable and a Value array of type Object, and to cast them back to Key[] and Value[] in the constructor. As usual, we can use array resizing so that clients do not have to be concerned with the size of the array (noting, as you shall see, that this method is too slow to use with large arrays).

Look at the partial implementation on pages 379-382 for more details. You will need to re-size the two arrays as necessary when adding or removing from them. Make sure you know how to do this - you can find this information on page 141. Let's discuss some of the more important methods.

```java
public int rank(Key key){
      int lo = 0, hi = N-1;

      while (lo <= hi){
      int mid = lo + (hi - lo) / 2;
      int cmp = key.compareTo(keys[mid]);

      if          (cmp < 0) hi = mid - 1;
      else if (cmp > 0) lo = mid + 1;
      else return mid;
      }
      return lo;
}
```

The heart of the implementation is this `rank()` method, which returns the number of keys smaller than a given key. We maintain indices into the sorted key array that delimit the sub-array that might contain the search key.

To search, we compare the search key against the key in the middle of the sub-array. If the search key is less than the key in the middle, we search in the left half of the sub-array; if the search key is greater than the key in the middle, we search in the right half of the sub-array; otherwise the key in the middle is equal to the search key.

This recursive `rank()` preserves the following properties:

1. If key is in the table, `rank()` returns its index in the table, which is the same as the number of keys in the table that are smaller than key.
2. If key is not in the table, `rank()` also returns the number of keys in the table that are smaller than key.

```java
public Value get(Key key){
      if (isEmpty()) return null;
      int i = rank(key);
      if (i < N && keys[i].compareTo(key) == 0) return vals[i];
      else return null;
}
```

For `get()`, the rank tells us precisely where the key is to be found if it is in the table (and, if it is not there, that it is not in the table).

```
public void put(Key key, Value val){
      int i = rank(key);
      if (i < N && keys[i].compareTo(key) == 0)
      { vals[i] = val; return; }

      for (int j = N; j > i; j--)
      { keys[j] = keys[j-1]; vals[j] = vals[j-1]; }
      keys[i] = key; vals[i] = val;
      N++;
}
```

For `put()`, the rank tells us precisely where to update the value when the key is in the table, and precisely where to put the key when the key is not in the table. We move all larger keys over one position to make room (working from back to front) and insert the given key and value into the proper positions in their respective arrays.

***Task 2:***
*Complete the implementation of the ordered symbol table. You can find some of the other implementations of methods in your textbook. Then calculate the growth rates of the algorithm. Which method is the most important to look at here?*

To discuss the growth rates of this algorithm, the `rank()` method is the most important to discuss. A call to `rank(key, 0, N-1)` does the same sequence of compares as a call to the non-recursive implementation, but this alternate version better exposes the structure of the algorithm. Since the keys are kept in an ordered array, most of the order-based operations are compact and straightforward.

However, `put()` is too slow (and `delete()`, for that matter). For a mostly static table that is already sorted, binary search is typically far better than sequential search. But typical modern search clients require symbol tables that can support fast implementations of both search and insert. That is, we need to be able to build huge tables where we can insert (and perhaps remove) key-value pairs in unpredictable patterns, intermixed with searches.

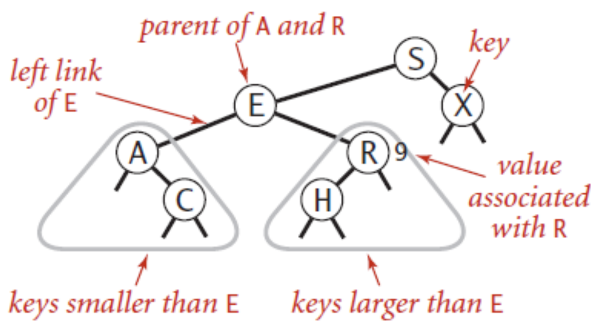
To support efficient searching in conjunction with insertion, it seems that we need a linked structure. But a singly linked list forecloses the use of binary search, because the efficiency of binary search depends on our ability to get to the middle of any sub-array algorithm quickly via indexing (and the only way to get to the middle of a singly linked list is to follow links).

## Section 4: Binary Search Trees

To combine the efficiency of binary search with the flexibility of linked structures, we need more complicated data structures. That combination is provided both by binary search trees and by hash tables. Specifically, using two links per node (instead of the one link per node found in linked lists) leads to an efficient symbol-table implementation based on the binary search tree data structure.

A binary search tree (BST) is a binary tree where each node has a Comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left sub-tree and smaller than the keys in all nodes in that node's right sub-tree.

We are working with data structures made up of nodes that contain links that are either null or references to other nodes. In a binary tree, we have the restriction that every node is pointed to by just one other node, which is called its parent (except for one node, the root, which has no nodes pointing to it), and that each node has exactly two links, which are called its left and right links, that point to nodes called its left child and right child, respectively.



**Anatomy of a binary search tree**

Although links point to nodes, we can view each link as pointing to a binary tree, the tree whose root is the referenced node. Thus, we can define a binary tree as a either a null link or a node with a left link and a right link, each references to (disjoint) subtrees that are themselves binary trees. In a binary search tree, each node also has a key and a value, with an ordering restriction to support efficient search.

Create a new class called BST.java and include the following implementation:

```
public class BST<Key extends Comparable<Key>, Value>{
     private Node root;                    // root of BST

     private class Node{
          private Key key;                 // key
          private Value val;               // associated value
          private Node left, right;        // links to sub-trees
          private int N;              // # nodes in sub-tree rooted here

          public Node(Key key, Value val, int N)
          { this.key = key; this.val = val; this.N = N; }
     }

     public int size() { return size(root); }

     private int size(Node x){
          if (x == null) return 0;
          else return x.N;
     }
     public Value get(Key key){}
     public void put(Key key, Value val){}
```

```
      // See page 407 for min(), max(), floor(), and ceiling().
      // See page 409 for select() and rank().
      // See page 411 for delete(), deleteMin(), and deleteMax().
      // See page 413 for keys().
}
```

This implementation of the ordered symbol-table API uses a binary search tree built from Node objects that each contain a key, associated value, two links, and a node count N. Each Node is the root of a subtree containing N nodes, with its left link pointing to a Node that is the root of a sub-tree with smaller keys and its right link pointing to a Node that is the root of a sub-tree with larger keys. The instance variable root points to the Node at the root of the BST (which has all the keys and associated values in the symbol table).

More than one BST can represent the same symbol table, depending on which node is considered the root of the tree. We take advantage of the flexibility inherent in having many BSTs represent this sorted order to develop efficient algorithms for building and using BSTs.

Let's look at the implementation of the `put()` and `get()` methods.

```
public Value get(Key key) { return get(root, key);}

private Value get(Node x, Key key){
      // Return value associated with key in the subtree rooted at x;
      // return null if key not present in subtree rooted at x.
      if (x == null) return null;
      int cmp = key.compareTo(x.key);
      if (cmp < 0) return get(x.left, key);
      else if (cmp > 0) return get(x.right, key);
      else return x.val;
}
```

A recursive algorithm to search for a key in a BST follows immediately from the recursive structure: if the tree is empty, we have a search miss; if the search key is equal to the key at the root, we have a search hit. Otherwise, we search (recursively) in the appropriate sub-tree, moving left if the search key is smaller, right if it is larger.

The procedure stops either when a node containing the search key is found (search hit) or when the current sub-tree becomes empty (search miss). Starting at the top, the search procedure at each node involves a recursive invocation for one of that node's children, so the search defines a path through the tree. For a search hit, the path terminates at the node containing the key. For a search miss, the path terminates at a null link.

```
public void put(Key key, Value val){
      // Search for key. Update value if found; grow table if new.
      root = put(root, key, val);
}

private Node put(Node x, Key key, Value val){
      // Change key's value to val if key in subtree rooted at x.
```

```
        // Otherwise, add new node to subtree associating key with val.
        if (x == null) return new Node(key, val, 1);

        int cmp = key.compareTo(x.key);

        if (cmp < 0) x.left = put(x.left, key, val);
        else if (cmp > 0) x.right = put(x.right, key, val);
        else x.val = val;

        x.N = size(x.left) + size(x.right) + 1;
        return x;
}
```

A more important essential feature of BSTs is that insert is not much more difficult to implement than search. Indeed, a search for a key not in the tree ends at a null link, and all that we need to do is replace that link with a new node containing the key. Make sure you can understand the recursive mechanics of this implementation.

***Task 3:***
*Complete the implementation of the BST class with the code snippets found in your textbook on pages 407-413. Notice that most of the methods use the same concepts we have applied in searching the tree (traversals of the tree). At each node, ideally half the options are being thrown away which should increase the performance of the algorithm a lot. Work out the growth rates of each method?*

**Section 5: 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.

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 4:***
*Complete the SeparateChainingHashST class and implement the LinearProbingHashST class as found on page 470 of your textbook. Work out the growth rates for each algorithm. Which technique is better? What simple modifications can you do to improve both implementations?*

**Section 6: Experimenting with Search Algorithms**
Now that you have implemented all these different searching techniques, it is necessary to do some experiments to learn a bit more about these algorithms. You might have noticed that most of these techniques are only useful with a large number of data to search from.

For your input, you can use Java's Random utility class for the primitive integer types or for a challenge; you can use the input file from your assignment 2 so that your implementations have to accommodate a user-defined type. Use 500 entries(`put()`) for your symbol table and time your implementations for about 4000 `get()` calls. This is because in almost all applications, the number of searches outnumber the inserts.

***Task 5:***
*Run and Time your implementations of searching algorithms. Don't forget to use Java's **Random** class as well as **Stopwatch** to time your code. Create a table which shows the various running times of 500 entries in your symbol table. One test should have all calls to `put()` at the start, and another test should have `put()` and `get()` interspersed with each other. Which algorithms are better in each situation?*

## 6. Further Practice Problems
For further practice, here are some suggested questions from your textbook:
3.1.5, 3.1.16, 3.1.17, 3.1.22, 3.1.28, 3.2.6, 3.2.25, 3.2.39, 3.4.9, 3.4.18, 3.4.19, 3.4.26