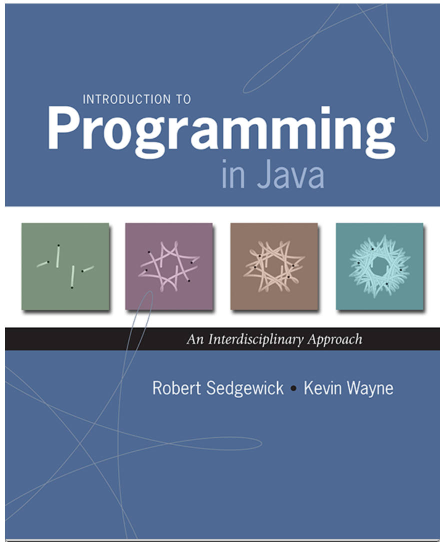
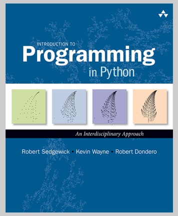
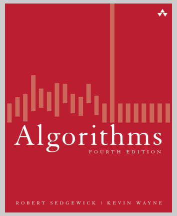


4.4 SYMBOL TABLES

This section under major construction.

Symbol tables. A *symbol table* is a data type that we use to associate *values* with *keys*. Clients can store (*put*) an entry into the symbol table by specifying a key-value pair and then can retrieve (*get*) the value corresponding to a particular key from the symbol table. For example, a university might associate information such as a student's name, home address, and grades (the value) with that student's social security number (the key), so that each student's records can be accessed by specifying a social security number. The same approach might be used by a scientist to organize data, by a business to keep track of customer transactions, or by an internet search engine to associate keywords with web pages.

API. A symbol table is a collection of key-value pairs. We use a generic type `Key` for keys and a generic type `Value` for values: every symbol-table entry associates a `Value` with a `Key`. In most applications, the keys have some natural ordering, so (as we did with sorting) we require the key type `Key` to implement Java's `Comparable` interface.

```
public class *ST<Key extends Comparable<Key>, Value>
             *ST()                      // create a symbol table
        void put(Key key, Value v)      // put key-value pair into the table
       Value get(Key key)               // return value paired with key
                                        // or null if no such value
     boolean contains(Key key)          // is there a value paired with key?
```

This API reflects several design decisions, which we now enumerate:

- *Comparable keys.* We take advantage of key ordering to develop efficient implementations of *put* and *get*. We also assume the keys do not change their value while in the symbol table. The simplest and most commonly used types of keys, `String` and built-in wrapper types like `Integer` and `Double`, are immutable.
- *Replace-the-old-value policy.* If when a key-value pair is inserted into the symbol table that already associates another value with the given key, we adopt the convention that the new value replaces the old one (just as with an array assignment statement).
- *Not found.* The method `get()` returns `null` if no entry with the given key has previously been put into the table. This choice has two implications, discussed next.
- *Null keys and values.* Clients are not permitted to use `null` as a key or value. This convention enables us to implement `contains()` as follows:

```
public boolean contains(Key key) {
   return get(key) != null;
}
```

- *Remove.* We do not include a method for removing keys from the symbol table. Many applications do require such a method, but we leave implementations as an exercise or for more advanced courses in data structures in algorithms. Since clients cannot associate `null` with a key, one simple interface is to take a *put* command with `null` as the value to mean *remove*.
- *Iterable.* As with most collections, it is best to implement `Iterable` and to provide an appropriate iterator that allows clients to visit the contents of the table. The natural order of iteration is in order of the keys, so we implement `Iterable<Key>` and use *get* to get values, if desired.
- *Variations.* Numerous other useful operations on symbol tables have been identified, and

APIs based on various subsets of them have been widely studied. We will consider several of these at the end of this section.

Symbol table clients. We consider two prototypical examples.

- *Dictionary lookup.* The most basic kind of symbol-table client builds a symbol table with successive *put* operations in order to be able to support *get* requests. The following list of familiar examples illustrates the utility of this approach.

Application	Action	Key	Value
phone book	look up phone number	person's name	phone number
dictionary	look up word	word	definition
Internet DNS	Look up website by IP address	website	IP address
Reverse DNS	Look up IP address by web site	IP address	Website
genomics	amino acid dictionary	codon	amino acid
Java compiler	Find properties of variable	Variable name	Value and type
stock quote	Look up price of stock	stock symbol	price
file share	find song to download	song name	machine
file system	find file on hard drive	file name	location on hard drive

Program Lookup.java builds a set of key-value pairs from a file of comma-separated values and then prints out values corresponding to keys read from standard input. The command-line arguments are the file name and two integers, one specifying the field to serve as the key and the other specifying the field to serve as the value.

Here are some sample data files: amino.csv (codons and amino acids), DJIA.csv (Dow Jones Industrial average by date), ip.csv (hostnames and IP addresses), morse.csv (Morse code), elements.csv (Periodic table of elements), mktsymbols.csv (market symbols and names), toplevel-domain.txt (top-level domain names and their country).

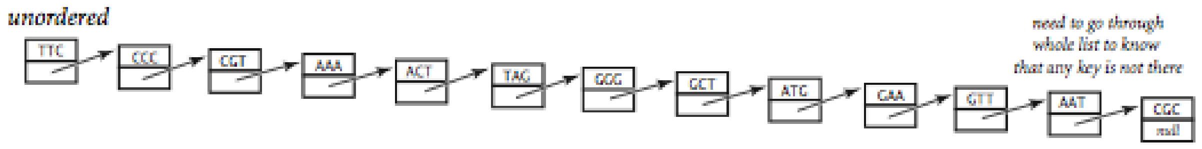
- *Indexing.* Program Index.java is a prototypical example of a symbol table client that uses an intermixed sequence of calls to `get()` and `put()`: it reads in a list of strings from standard input and prints a sorted table of all the different strings along with a list of integers specifying the positions where each string appeared in the input. We have a large amount of data and want to know where certain strings of interest are found. In this case, we seem to be associating multiple values with each key, but we actually associating just one: a queue. Again, this approach is familiar:

Application	Action	Key	Value
book index	search for terms	term	page numbers
genomics	find genetic markers	DNA substring	locations
web search	find information on web	keyword	websites
business	find transactions	customer name	transactions

Elementary implementations. Symbol-table implementations have been heavily studied, Many different algorithms and data structures have been invented for this purpose. We first describe two simple approaches.

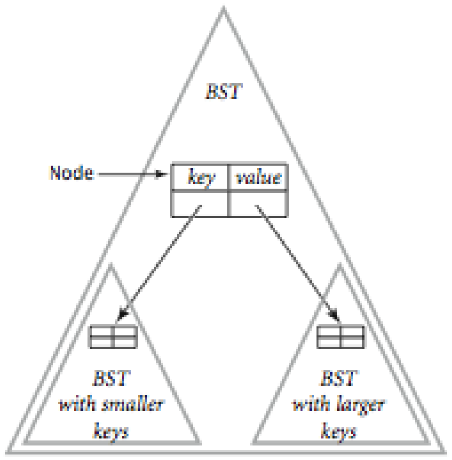
- *Binary search implementation.* Program BinarySearchST.java implements a symbol table by maintaining two parallel arrays of keys and values, keeping them in key-sorted order. It uses *binary search* for *get*. To maintain the keys in sorted order, it moves all of the larger elements over for *put*. The *get* operation is logarithmic, but the *put* operation takes linear time per operation.

- *Linked list implementation.* Program LinkedListST.java implements a symbol table with an (unordered) linked list. Both *put* and *get* take linear time per operation: to search for a key, we need to traverse its links; to put a key-value pair, we need to search for the given key.


unordered
TTC
CCC
CGT
AAA
ACT
TAG
GGG
GCT
ATC
GAA
GTT
AAT
CGC
null
need to go through whole list to know that any key is not there

To develop a symbol-table implementation that is feasible for use with clients like `Lookup` and `Index`, we need the flexibility of linked lists and the efficiency of binary search. Binary search trees, which we consider next, provide just this combination.

Binary search trees. The *binary tree* is a mathematical abstraction that plays a central role in the efficient organization of information. Like arrays and linked lists, a binary tree is a data type that stores a collection of data. Binary trees play an important role in computer programming because they strike an efficient balance between flexibility and ease of implementation.



Binary search tree

For symbol-table implementations, we use special type of binary tree to organize the data and to provide a basis for efficient implementations of the symbol-table *put* operations and *get* requests. A *binary search tree* (*BST*) associates `Comparable` keys with values, in a structure defined recursively as follows: A BST is either

- empty (`null`) or
- a node having a key-value pair and two references to BSTs, a left BST with smaller keys and a right BST with larger keys.

The key type must be `Comparable`, but the type of the value is not specified, so a BST node can hold any kind of data in addition to the (characteristic) references to BSTs. To implement BSTs, we start with a class for the node abstraction, which has references to a key, a value, and left and right BSTs:

```
private class Node {
   Key   key;
   Value val;
   Node  left, right;

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

}
```

This definition is like our definition of nodes for linked lists, except that it has *two* links, not just one.

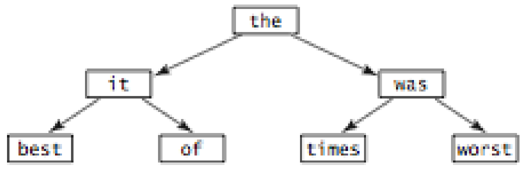
- *Search in a BST.* Suppose that you want to *search* for a node with a given key in a BST (or to *get* a value with a given key in a symbol table). There are two possible outcomes: the search might be successful (we find the key in the BST; in a symbol-table implementation we return the associated value) or it might be unsuccessful (there is no key in the BST with the given key; in a symbol-table implementation, we return `null`). A recursive algorithm is immediate: Given a BST (a reference to a `Node`), first check whether the tree is empty (the reference is `null`). If so, then terminate the search as unsuccessful (in a symbol-table implementation, return `null`). If the tree is non-empty, check whether the key in the node is equal to the search key. If so, then terminate the search as successful (in a symbol-table implementation, return the value associated with the key). If not, compare the search key with the key in the node. If it is smaller, search (recursively) in the left subtree; if it is greater, search (recursively) in the right subtree.

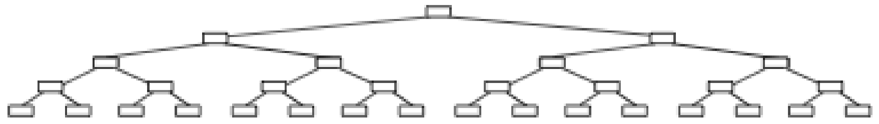
- *Inserting into a BST.* Suppose that you want to insert a new node into a BST (in a symbol-table implementation, *put* a new key-value pair into the data structure). The logic is similar to searching for a key, but the code is trickier: If the BST is empty, we create and return a new `Node` containing the key-value pair; if the search key is less than the key at the root, we set the left link to the result of inserting the key-value pair into the left subtree; if the search key is less, we set the right link to the result of inserting the key-value pair into the right subtree; otherwise if the search key is equal, we overwrite the existing value with the new value. Resetting the link after the recursive call in this way is usually unnecessary, because the link changes only if the subtree is empty, but it is as easy to set the link as to test to avoid setting it.

Program BST.java is a symbol-table implementation based on these two recursive algorithms. Here is a useful binary search tree application.

Performance characteristics of BST. The running times of algorithms on BSTs are dependent on the shape of the trees, and the shape of the trees is dependent on the order in which the keys are inserted.

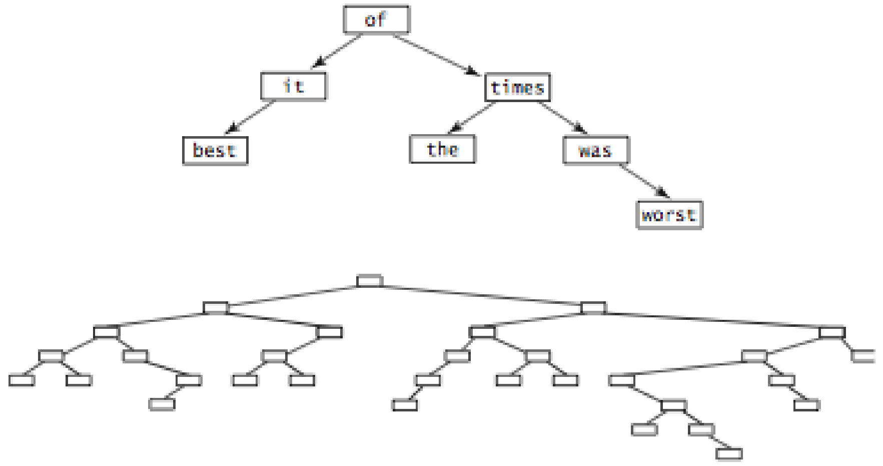
- *Best case.* In the best case, the tree is perfectly balanced (each `Node` has exactly two non-null children), with lg N nodes between the root and each leaf node. In such a tree, it is easy to see that the cost of an unsuccessful search is logarithmic, because that cost satisfies the same recurrence relation as the cost of binary search (see Section 4.2) so that the cost of every *put* operation and *get* request is proportional to lg N or less.





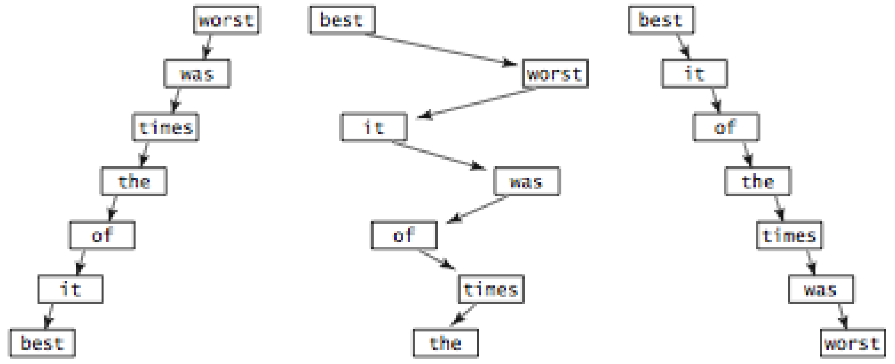
Best case (perfectly balanced) BSTs

- *Average case.* If we insert random keys, we might expect the search times to be logarithmic as well, because the first element becomes the root of the tree and should divide the keys roughly in half. Applying the same argument to the subtrees, we expect to get about the same result as for the best case. This intuition is indeed validated by careful analysis: a classic mathematical derivation shows that the time required for put and get in a tree constructed from randomly ordered keys is logarithmic. More precisely, the expected number of key comparisons is ~ 2 ln N for a random *put* or *get* in a key built from N randomly ordered keys.



Typical BSTs constructed from randomly-ordered keys

- *Worst case.* In the worst case, each node has exactly one null link, so the BST is like a linked list, where *put* operations and *get* requests take linear time. Unfortunately this worst case is not rare in practice - it arises, for example, when we insert the keys in order. Thus, good performance of the basic BST implementation is dependent on the keys being sufficiently similar to random keys that the tree is not likely to contain many long paths.



Worst case (totally unbalanced) BSTs

Remarkably, there are BST variants that eliminate this worst case and guarantee logarithmic performance per operation, by making all trees nearly perfectly balanced. One popular variant is known as a *red-black tree*. The Java library java.util.TreeMap implements a symbol table using this approach. Our symbol-table implementation ST.java uses this data structure to implement our symbol-table API. It is remarkably efficient: typically, it only accesses a small number of the nodes in the BST (those on the path from the root to the node sought or to the leaf to which the new node is attached) and it only creates one new `Node` and adds one new link for the *put* operation. Next, we show that put operations and get requests take logarithmic time (under certain assumptions).

Traversing a BST. Perhaps the most basic tree-processing function is known as *tree traversal*: Given a (reference to) a tree, we want to systematically process every key-value pair in the tree. For linked lists, we accomplish this task by following the single link to move from one node to the next. For trees, however, we have decisions to make, because there are generally two links to follow. But recursion comes immediately to the rescue. To process every key in a BST:

- process every key-value pair in the left subtree
- process the key-value pair at the root
- process every key-value pair in the right subtree

This approach not only processes every key pair in the BST, but it does so in key-sorted order. For example, the following method prints the key-value pairs in the tree rooted at its argument in key-sorted order.

```
private static void traverse(Node x) {
   if (x == null) return;
   traverse(x.left);
   StdOut.println(x.key + " " + x.val);
   traverse(x.right);
}
```

This remarkably simple method is worthy of careful study. It can be used as a basis for a `toString()` implementation for BSTs and also a starting point for developing an iterator.

Extended symbol table operations. The flexibility of BSTs enable the implementation of many useful additional operations beyond those dictated by the symbol table API.

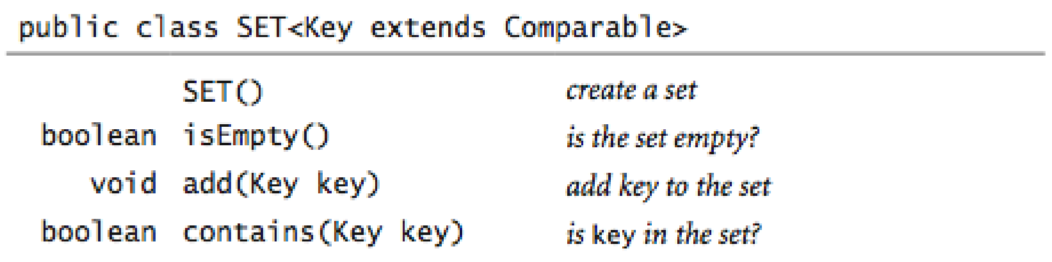
- *Minimum and maximum.* To find the smallest key in a BST, follow left links from the root until reaching null. The last key encountered is the smallest in the BST. The same procedure following right links lead to the largest key.
- *Size.* To keep track of the number of nodes in a BST, keep an extra instance variable `N` in BST that counts the number of nodes in the tree. Initialize it to 0 and increment it whenever creating a new `Node`.
- *Remove.* Many applications demand the ability to remove a key-value pair with a given key. You can find explicit code for removing a node from a BST in a book on algorithms and data structures. An easy lazy way to implement `remove()` relies on the fact that values cannot be `null`:

  ```
  public void remove(Key key) {
     if (contains(key))
        put(key, null);
  }
  ```

  This approach necessitates periodically cleaning out nodes in the BST with `null` values, because performance will degrade unless the size of the data structure stays proportional to the number of key-value pairs in the table.
- *Range search.* With a recursive method like `traverse()`, we can count the number of keys that fall within a given range or return all the keys falling into a given range.
- *Order statistics.* If we maintain an instance variable in each node having the size of the

subtree rooted at each node, we can implement a recursive method that returns the kth largest key in the BST.

Set data type. As a final example, we consider a data type that is simpler than a symbol table, still broadly useful, and easy to implement with BSTs. A *set* is an (unordered) collection of distinct comparable keys, defined by the following API:



`public class SET<Key extends Comparable>`

	`SET()`	*create a set*
`boolean`	`isEmpty()`	*is the set empty?*
`void`	`add(Key key)`	*add key to the set*
`boolean`	`contains(Key key)`	*is key in the set?*

A set is a symbol table with no values. We could use BST to implement SET, but a direct implementation is simpler, and client code that uses SET is simpler and clearer than it would be to use placeholder values and ignore them. Program SET.java implements the set API. Program DeDup.java is a SET client that reads in a sequence of strings from standard input and prints out the first occurrence of each string (thereby removing duplicates).

Perspective. The use of binary search trees to implement symbol tables is a sterling example of exploiting the tree abstraction, which is ubiquitous and familiar. Trees lie at the basis of many scientific topics, and are widely used in computer science. We are accustomed to many tree structures in everyday life including family trees, sports tournaments, the organization chart of a company, the tree of life, and parse trees in grammar. Trees also arise in numerous computational applications including function call trees, parse trees, and file systems. Many important applications are rooted in science and engineering, including phylogenetic trees in computational biology, multidimensional trees in computer graphics, minimax game trees in economics, and quad trees in molecular dynamics simulations.

Q + A.

Q. Why use immutable symbol table keys?

A. If we changed a key while it was in the BST, it would invalidate the ordering restriction.

Q. Why not use the Java library methods for symbol tables?

A. Now that you understand how a symbol table works, you are certainly welcome to use the industrial strength versions java.util.TreeMap and java.util.HashMap They follow the same basic API as BST, but they allow null keys and use the names `containsKey()` and `keySet()` instead of `contains()` and `iterator()`, respectively. They also contain additional methods such as `remove()`, but they do not provide any efficient way to add some of the additional methods that we mentioned, such as order statistics. You can also use `java.util.TreeSet` and `java.util.HashSet` which implement an API like our SET.

Exercises

1. Modify `Lookup` to to make a program `LookupAndPut` that allows *put* operations to be specified on standard input. Use the convention that a plus sign indicates that the next two strings typed are the key-value pair to be inserted.
2. Modify `Lookup` to make a program `LookupMultiple` that handles multiple values having the same key by putting the values on a queue, as in `Index`, and then printing them all out on a *get* request, as follows:

```
% java LookupMultiple amino.csv 3 0
Leucine
TTA TTG CTT CTC CTA CTG
```

3. Modify `Index` to make a program `IndexByKeyword` that takes a file name from the command line and makes an index from standard input using only the keywords in that file. Note : using the same file for indexing and keywords should give the same result as `Index`.
4. Modify `Index` to make a program `IndexLines` that considers only consecutive sequences of letters as keys and uses line numbers instead of word position as the value and to . This functionality is useful for programs, as follows:

```
% java IndexLines 6 0 < Index.java
continue 12
enqueue 15
Integer 4 5 7 8 14
parseInt 4 5
println 22
```

5. Develop an implementation BinarySearchST.java of the symbol-table API that maintains parallel arrays of keys and values, keeping them in key-sorted order. Use binary search for get and move larger elements to the right one position for put (use array doubling to keep the array size proportional to the number of key-value pairs in the table). Test your implementation with `Index`, and validate the hypothesis that using such an implementation for `Index` takes time proportional to the product of the number of strings and the number of distinct strings in the input.
6. Develop an implementation LinkedListST.java of the symbol-table API that maintains a linked list of nodes containing keys and values, keeping them in key-sorted order. Test your implementation with `Index`, and validate the hypothesis that using such an implementation for `Index` takes time proportional to the product of the number of strings and the number of distinct strings in the input.
7. Draw all the different BSTs that can represent the key sequence

```
best of it the time was.
```

8. Draw the BST that results when you insert items with keys

   ```
   E A S Y Q U E S T I O N
   ```

   in that order into an initially empty tree.
9. Suppose we have int values between 1 and 1000 in a BST and search for 363. Which of the following cannot be the sequence of keys examined.
   a. 2 252 401 398 330 363
   b. 399 387 219 266 382 381 278 363
   c. 3 923 220 911 244 898 258 362 363
   d. 4 924 278 347 621 299 392 358 363
   e. 5 925 202 910 245 363
10. Suppose that the following 31 keys appear (in some order) in a BST of height 5:

    ```
    10 15 18 21 23 24 30 30 38 41
    42 45 50 55 59 60 61 63 71 77
    78 83 84 85 86 88 91 92 93 94
    98
    ```

    Draw the top 3 nodes of the tree (the root and its two children).
11. Implement `toString()` for BST, using a recursive helper method like `traverse()`. As

usual, you can accept quadratic performance because of the cost of string concatenation. Extra credit : Write a linear-time toString() method for BST.

12. True or false. Given a BST, let *x* be a leaf node, and let *p* be its parent. Then either (i) the key of *p* is the smallest key in the BST larger than the key of *x* or (ii) the key of *p* is the largest key in the BST smaller than the key of *x*. *Answer*: true.
13. Modify the symbol-table API to use a wrapper type `STentry` that holds the keys and values (with accessor methods key() and value() to access them). Your iterator should return `STentry` objects in key-sorted order. Implement BST and Index as dictated by this API. Discuss the pros and cons of this approach versus the one given in the text.
14. Modify the symbol-table API to handle values with duplicate keys by having `get()` return a iterator for the values having a given key. Implement BST and Index as dictated by this API. Discuss the pros and cons of this approach versus the one given in the text.
15. Prove that the expected number of key comparisons for a random *put* or *get* in a BST build from randomly ordered keys is ~ 2 ln N.
16. Prove that the number of different BSTs that can be built from N keys is ~
17. Run experiments to validate the claims in the text that the *put* operations and *get* requests for Lookup and Index are logarithmic in the size of the table when using `BST`.
18. Modify BST to implement a symbol-table API where keys are arbitrary objects. Use `hashCode()` to convert keys to integers and use integer keys in the BST.
19. Modify BST to add methods `min()` and `max()` that return the smallest (largest) key in the table (or `null` if no such key exists).
20. Modify BST to add a method `size()` that returns the number of elements in the table. Use the approach of storing within each `Node` the number of nodes in the subtree rooted there.
21. Modify BST to add a method `rangeCount()` that takes two `Key` arguments and returns the number of keys in a BST between the two given keys. Your method should take time proportional to the height of the tree. Hint : First work the previous exercise.

22. Modify `SET` to add methods `floor()`, `ceil()`, and `nearest()` that take as argument a `Key` and return the largest (smallest, nearest) element in the set that is no larger than (no smaller than, closest to) the given `Key`.
23. Write a ST client GPA.java that creates a symbol table mapping letter grades to numerical scores, as in the table below, then reads from standard input a list of letter grades and computes their average (GPA).

A+	A	A-	B+	B	B-	C+	C	C-	D	F
4.33	4.00	3.67	3.33	3.00	2.67	2.33	2.00	1.67	1.00	0.00

Binary Tree Exercises

This list of exercises is intended to give you experience in working with binary trees that are not necessarily BSTs. They all assume a Node class with three instance variables: a integer value and two Node references.

1. Implement the following methods for a binary tree that each take as argument a `Node` that is the root of a binary tree.
    - `size()`: number of nodes in the tree.
    - `leaves()`: number of nodes whose links are both null
    - `total()`: sum of the key values in all nodes

    Your methods should all run in linear time.
2. Implement a linear-time method `height()` that returns the maximum number of nodes on any path from the root to a leaf node (the height of the `null` tree is 0; the height of a 1-node tree is 1).

3. A binary tree is *heap-ordered* if the key at the root is larger than the keys in all of its descendants. Implement a linear-time method `heapOrdered()` that returns `true` if the tree is heap-ordered, `false` otherwise.
4. A binary tree is *balanced* if both its subtrees are balanced and the height of its two subtrees differ by at most 1. Implement a linear-time method `balanced()` that returns `true` if the tree is balanced, `false` otherwise.
5. Two binary trees are *isomorphic* if only their key values differ (they have the same shape). Implement a linear-time static method `isomorphic()` that takes two tree references as parameters and returns `true` if they refer to isomorphic trees, `false` otherwise. Then implement a linear-time static method `eq()` that takes two tree references as parameters and returns `true` if they refer to identical trees (isomorphic with same key values), `false` otherwise.

```
public static boolean isomorphic(Node x, Node y) {
    if (x == null && y == null) return true;   // both null
    if (x == null || y == null) return false;  // exactly one null
    return isomorphic(x.left, y.left) && isomorphic(x.right, y.right);
}
```

6. Implement a linear-time method `isBST()` that returns `true` if the tree is a BST, `false` otherwise.

   Solution : This task is a bit more difficult than it might seem. We use an overloaded recursive method `isBST()` that takes two additional arguments `min` and `max` and returns `true` if the tree is a BST *and* all its values are between `min` and `max`.

```
public static boolean isBST() {
    return isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

private boolean isBST(Node x, int min, int max) {
    if (x == null) return true;
    if (x.val < min || x.val > max) return false;
    return isBST(x.left, min, x.val) && isBST(x.right, x.val, max);
}
```

7. Write a method `levelOrder()` that prints BST keys in *level order*: First print the root, then the nodes one level below the root, left to right, then the nodes two levels below the root (left to right), and so forth. *Hint* : Use a `Queue<Node>`.
8. Compute the value returned by mystery() on some sample binary trees, then formulate an hypothesis about its behavior and prove it.

```
public int mystery(Node x) {
    if (x == null) return 0;
    else return mystery(x.left) + mystery(x.right);
}
```

Answer: Returns 0 for any binary tree.

Creative Exercises

1. Spell checking. Write a `SET` client SpellChecker.java that takes as command-line argument the name of a file containing a dictionary of words, and then reads strings from standard input and prints out any string that is not in the dictionary. Use the 600,000+ word dictionary

words.utf-8.txt.

2. Spell correction. Write an `ST` client SpellCorrector.java that serves as a filter that replaces commonly misspelled words on standard input with a suggested replacement, printing the result to standard output. Take as command-line argument a file that contains common misspellings and corrections. Use the file misspellings.txt, which contains many common misspellings.
3. Web filter. Write a `SET` client `WebBlocker` that takes as command-line argument the name of a file containing a list of objectionable websites and then reads strings from standard input and prints out only those websites that should not be filtered.
4. Set operations. Add methods `union()` and `intersection()` to the set ADT. Do not worry about efficient BST implementations.
5. Frequency symbol table. Develop a data type FrequencyTable.java that supports the following operations: `click()` and `count()`, both of which take `String` arguments. The data-type value is an integer that keeps track of the number of times the `click()` operation is been called with the given `String` as argument. `The click()` operation increments the count by one, and the `count()` operation returns the value, possibly 0. Clients of this data type might include a web traffic analyzer, a music player that counts the number of times each song has been played, phone software for counting calls, and so forth.
6. 1D range searching. Develop a data type that supports the following operations: insert a date, search for a date, and count the number of dates in the data structure that lie in a particular interval. Use either Java's java.util.Date data type or java.util.GregorianCalendar data type to represent the date.
7. Non-overlapping interval search. Given a list of non-overlapping intervals of integers, write a function that takes an integer argument and determines in which if any interval that values lies, e.g., if the intervals are 1643-2033, 5532-7643, 8999-10332, 5666653-5669321, then the query point 9122 lies in the third interval and 8122 lies in no interval.
8. IP lookup by country. Write a `BST` client that uses the data file the data file ip-to-country.csv to determine what country a given IP address is coming from. The data file has five fields

(beginning of IP address range, ending of IP address range, two character country code, three character country code, and country name. The IP addresses are non-overlapping. Such a database tool can be used for: credit card fraud detection, spam filtering, auto-selection of language on a web site, and web server log analysis.

See also GeoIPCountryWhois.csv, the IP-to-country website or MaxMind GeoIP.

9. Inverted index of web. Given a list of web pages, create a symbol table of words contained in those web pages. Associate with each word a set of web pages in which that word appears. Program InvertedIndex.java takes a list of filenames as command-line inputs, reads in the text files, creates a symbol table of sets, and supports single-word queries by returning the set of web pages in which that query word appears.
10. Inverted index of web. Extend the previous exercise so that it supports multi-word queries. In this case, output the list of web pages that contain at least one occurrence of each of the query words.
11. Multiple word search. Write a program that takes k words from the command-line, reads in sequence of words from standard input, and identifies the smallest subsequence of text that contains all of the k words (not necessarily in the same order). Don???t consider partial words.

    Hint: for each index i, find the smallest subsequence [i, j] that contains the k query words. Keep a count of the number of times each of the k query words appear. Given [i, j], compute [i+1, j'] by decrementing the counter for word i. Then gradually increase j until the subsequence contains at least one copy of each of the k words.
12. Repetition draw in chess. In the game of chess, if a board position is repeated three times with the same side to move, the side to move can declare a draw. Describe how you could test this condition using a computer program.
13. Registrar scheduling. The Registrar at a prominent northeastern University recently scheduled an instructor to teach two different classes at the same exact time. Help the Registrar prevent future mistakes by describing a method to check for such conflicts. For

simplicity, assume all classes run for 50 minutes starting at 9, 10, 11, 1, 2, or 3.

14. Entropy. We define the *relative entropy* of a text corpus with N words, k of which are distinct as

$$E = 1 / (N \lg N) (p_1 \lg(k / p_1) + \ldots + p_k \lg(k / p_k))$$

where p_i is the fraction of times that word i appears. Write a program that reads in a text corpus and prints out the relative entropy. Convert all letters to lowercase and treat punctuation marks as whitespace.

15. Order statistics. Add to BST a method `select()` that takes an integer argument k and returns the kth largest key in the BST. Assume that the subtree sizes are maintained in each node. The running time should be proportional to the height of the tree.

16. Delete ith element. Create an ADT that supports the following operations: `isEmpty()`, `insert()`, and `delete(int i)`, where the deletion operation deletes and returns the ith least recently added object. Use a BST with a counter that records the total number of elements n inserted and give the nth element inserted the key n. Each BST node should also maintain the total number of nodes in the subtree rooted at that node. To find the ith least recently added item, search for the ith smallest element in the BST.

17. Mutable string. Create an ADT that supports the following operations on a string: `get(int i)`, `insert(int i, char c)`, and `delete(int i)`, where `get` returns the ith character of the string, `insert()` inserts the character c and makes it the ith character, and `delete()` deletes the ith character. Use a BST to implement operations in logarithmic time.

18. Sparse vectors and matrices. An N-by-N matrix is sparse if its number of nonzeros is proportional to N (or less). Goal: represent sparse matrix with space proportional to N by only implicitly storing the zero entries. Add two sparse vectors/matrices in time proportional to the number of nonzeros. Design ADTs SparseVector.java and SparseMatrix.java that support the following APIs.

```
public class SparseVector
   public SparseVector(int N)                   // 0-vector of length N
   public void (put i, double value)            // a[i] = val
   public double get(int i)                     // return a[i]
   public double dot(SparseVector b)            // vector dot product
   public SparseVector plus(SparseVector b)     // vector addition

public class SparseMatrix
   public SparseMatrix(int N)
   public void put(int i, int j, double val)     // a[i][j] = val
   public double get(int i, int j)               // return a[i][j]
   public SparseMatrix times(SparseMatrix b)     // matrix product
   public SparseMatrix plus(SparseMatrix b)      // matrix addition
```

19. Expression parser. Write a program to parse and evaluate expressions of the following form. Store the variable names in a symbol table.

```
A = 5
B = 10
C = A + B
D = C * C
Result: A = 5,  B = 10,  C = 15, D = 225
```

Consider adding more complicated expressions, e.g., E = 7 * (B + C * D).

20. Actor and actress aliases. Given this 10MB file containing a list of actors (with canonical name) and their aliases, write a program that reads in the name of an actor from standard input and prints out his canonical name.
21. Database joins. Given two tables, an inner join finds the "intersection" between the two tables.

```
Name       Dept ID        Dept       Dept ID
-----------------         ----------------
Smith        34           Sales         31
Jones        33           Engineering   33
Robinson     34           Clerical      34
Jasper       36           Marketing     35
Steinberg    33
Rafferty     31
```

The inner join on department ID is as follows.

```
Name       Dept ID    Dept
-------------------------------
Smith        34       Clerical
Jones        33       Engineering
Robinson     34       Clerical
Steinberg    33       Engineering
Rafferty     31       Sales
```

22. Molecular weight calculator. Write a program MolecularWeight.java that reads in a list of elements and their molecular weights, and then prompts the user for the molecular description of a chemical compound (e.g., CO.2 or Na.Cl or N.H4.N.O3 = ammonium nitrate) and prints out its molecular weight.

Web Exercises

1. Codon usage table. Describe how to hash codons (3 consecutive nucleotides) to an integer

between 0 and 63. Print out summary statistics for each codon, e.g.,

```
UUU 13.2(   724)  UCU 19.6(  1079)  UAU 16.5(   909)  UGU 12.4(   680)
UUC 23.5(  1290)  UCC 10.6(   583)  UAC 14.7(   808)  UGC  8.0(   438)
UUA  5.8(   317)  UCA 16.1(   884)  UAA  0.7(    38)  UGA  0.3(    15)
UUG 17.6(   965)  UCG 11.8(   648)  UAG  0.2(    13)  UGG  9.5(   522)

CUU 21.2(  1166)  CCU 10.4(   571)  CAU 13.3(   733)  CGU 10.5(   578)
CUC 13.5(   739)  CCC  4.9(   267)  CAC  8.2(   448)  CGC  4.2(   233)
CUA  6.5(   357)  CCA 41.0(  2251)  CAA 24.9(  1370)  CGA 10.7(   588)
CUG 10.7(   590)  CCG 10.1(   553)  CAG 11.4(   625)  CGG  3.7(   201)

AUU 27.1(  1491)  ACU 25.6(  1405)  AAU 27.2(  1495)  AGU 11.9(   653)
AUC 23.3(  1279)  ACC 13.3(   728)  AAC 21.0(  1151)  AGC  6.8(   374)
AUA  5.9(   324)  ACA 17.1(   940)  AAA 32.7(  1794)  AGA 14.2(   782)
AUG 22.3(  1223)  ACG  9.2(   505)  AAG 23.9(  1311)  AGG  2.8(   156)

GUU 25.7(  1412)  GCU 24.2(  1328)  GAU 49.4(  2714)  GGU 11.8(   650)
GUC 15.3(   843)  GCC 12.6(   691)  GAC 22.1(  1212)  GGC  7.0(   384)
GUA  8.7(   478)  GCA 16.8(   922)  GAA 39.8(  2185)  GGA 47.2(  2592)
```

2. Functions on trees. Write a function `count()` that takes a `Node x` as an argument returns the number of nodes in the subtree rooted at `x` (including `x`). The number of elements in an empty binary tree is 0 (base case), and the number of elements in a non-empty binary tree is equal to one plus the number of elements in the left subtree plus the number of elements in the right subtree.

```
public static int count(TwoNode x) {
   if (x == null) return 0;
```

```
    return 1 + count(x.left) + count(x.right);
}
```

3. Random element. Add a symbol table function `random` to a BST that returns a random element. Assume that the nodes of your BST have integer size fields that contain the number of elements in the subtree rooted at that node. The running time should be proportional to the length of the path from the root to the node returned.
4. Markov language model. Create a data type that supports the following two operations: `add` and `random`. The `add` method should insert a new item into the data structure if it doesn't yet exists; if it already exists, it should increase its frequency count by one. The `random` method should return an element at random, where the probabilities are weighted by the frequency of each element.
5. Queue with no duplicates. Create a data type that is a queue, except that an element may only appear on the queue at most once at any given time. Ignore requests to insert an item if it is already on the queue.
6. Bayesian spam filter. Follow the ideas in A Plan for Spam. Here is a place to get test data.
7. Symbol table with random access. Create a data type that supports inserting a key-value pair, searching for a key and returning the associated value, and deleting and returning a random value. *Hint*: combine a symbol table and a randomized queue.
8. Random phone numbers. Write a program that takes a command line input N and prints N random phone numbers of the form (xxx) xxx-xxxx. Use a symbol table to avoid choosing the same number more than once. Use this list of area codes to avoid printing out bogus area codes.
9. Unique substrings of length L. Write a program that reads in text from standard input and calculate the number of unique substrings of length L that it contains. For example, if the input is `cgcgggcgcg` then there are 5 unique substrings of length 3: `cgc`, `cgg`, `gcg`, `ggc`, and `ggg`. Applications to data compression. *Hint*: use the string method `substring(i, i + L)`

to extract ith substring and insert into a symbol table. Test it out on the first million digits of π. or 10 million digits of π.

10. The great tree-list recursion problem. A binary search tree and a circular doubly linked list are conceptually built from the same type of nodes - a data field and two references to other nodes. Given a binary search tree, rearrange the references so that it becomes a circular doubly-linked list (in sorted order). Nick Parlante describes this as one of the neatest recursive pointer problems ever devised. *Hint*: create a circularly linked list A from the left subtree, a circularly linked list B from the right subtree, and make the root a one node circularly linked list. Them merge the three lists.
11. Optimal BST.
12. Password checker. Write a program that reads in a string from the command line and a dictionary of words from standard input, and checks whether it is a "good" password. Here, assume "good" means that it (i) is at least 8 characters long, (ii) is not a word in the dictionary, (iii) is not a word in the dictionary followed by a digit 0-9 (e.g., hello5), (iv) is not two words separated by a digit (e.g., hello2world)
13. Reverse password checker. Modify the previous problem so that (ii) - (v) are also satisfied for reverses of words in the dictionary (e.g., olleh and olleh2world). *Simple solution*: insert each word and its reverse into the symbol table.
14. Cryptograms. Write a program to read in a cryptogram and solve it. A cryptogram is ancient form of encryption known as a substitution cipher in which each letter in the original message is replaced by another letter. Assuming we use only lowercase letters, there are 26! possibilities, and your goal is to find one that results in a message where each word is a valid word in the dictionary. Use Permutations.java and backtracking.
15. Frequency counter. Write a program FrequencyCounter.java that reads in a sequence of strings from standard input and *counts* the number of times each string appears.
16. Unordered array symbol table. Write a program SequentialSearchST.java that implements a symbol-table using (unordered) parallel arrays.
17. Exception filter. The client program ExceptionFilter.java reads in a sequence of strings from

a whitelist file specified as a command-line argument, then it prints out all words from standard input not in the whitelist.

18. Give the preorder traversal of some BST (not including null nodes). Ask the reader to reconstruct the tree.
19. Write an ADT IterativeBST.java that implements a symbol table using a BST, but uses iterative versions of `get()` and `put()`.
20. Largest absolute value.
21. Tree reconstruction. Given the following traversals of a binary tree (only the elements, not the null nodes), can you can you reconstruct the tree?
    a. Preorder and inorder.
    b. Postorder and inorder.
    c. Level order and inorder.
    d. Preorder and level order.
    e. Preorder and postorder.

    Answers

    a. Yes. Scan the preorder from left to right, and use the inorder traversal to identify the left and right subtrees.
    b. Yes. Scan the postorder from right to left.
    c. Yes. Scan the level order from left to right.
    d. No. Consider two trees with A as the root and B as either the left or right child. The preorder traversal of both is AB and the level order traversal of both is AB.
    e. No. Same counterexample as above.
22. Highlighting browser hyperlinks. Browsers typically denote hyperlinks in blue, unless they've already been visited, in which case they're depicted in purple Write a program `HyperLinkColorer.java` that reads in a list of web addresses from standard input and output `blue` if it's the first time reading in that string, and `purple` otherwise.

23. Spam blacklist. Insert known spam email addresses into a SET.java ADT, and use to blacklist spam.
24. Inverted index of a book. Write a program that reads in a text file from standard input and compiles an alphabetical index of which words appear on which lines, as in the following input. Ignore case and punctuation.

```
It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness,

age 3-4
best 1
foolishness 4
it 1-4
of 1-4
the 1-4
times 1-2
was 1-4
wisdom 4
worst 2
```

    Hint: create a symbol table whose key is a `String` that represents a word and whose value is a `Sequence<Integer>` that represents the list of pages on which the word appears.
25. Randomly generated identities. The files names20k.csv and The file names20k-2.csv each contain 20,000 randomly generated identities (number, gender, first name, middle initial, last name, street address, city, state, zip code, country, email address, telephone number, mother's maiden name, birthday, credit card type, credit card number, credit card expiration date, Social security number) from fakenamegenerator.com.

26. Distance between zip codes. Write a data type Location.java that represents a named location on the surface of the earth (a name, latitude, and longitude). Then, write a client program that takes the name of a ZIP code file (such as zips.txt) as a command-line argument, reads the data from the file, and stores it in a symbol table. Then, repeatedly read in pairs of ZIP codes from standard input and output the great-circle distance between them (in statute miles). This distance is used by the post office to calculate shipping rates.

Last modified on October 23, 2012.