**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 andDouble, 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](http://introcs.cs.princeton.edu/java/44st/Lookup.java.html) 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](http://introcs.cs.princeton.edu/java/44st/amino.csv) (codons and amino acids), [DJIA.csv](http://introcs.cs.princeton.edu/java/44st/DJIA.csv) (Dow Jones Industrial average by date), [ip.csv](http://introcs.cs.princeton.edu/java/44st/ip.csv) (hostnames and IP addresses), [morse.csv](http://introcs.cs.princeton.edu/java/44st/morse.csv) (Morse code), [elements.csv](http://introcs.cs.princeton.edu/java/44st/elements.csv) (Periodic table of elements), [mktsymbols.csv](http://introcs.cs.princeton.edu/java/44st/mktsymbols.csv) (market symbols and names), [toplevel-domain.txt](http://introcs.cs.princeton.edu/java/44st/toplevel-domain.txt)(top-level domain names and their country).
* *Indexing.* Program [Index.java](http://introcs.cs.princeton.edu/java/44st/Index.java.html) 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 |

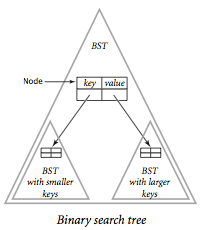
**Elementary implementations.**

 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](http://introcs.cs.princeton.edu/java/44st/BinarySearchST.java.html) 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](http://introcs.cs.princeton.edu/java/44st/LinkedListST.java.html) 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.



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.

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.

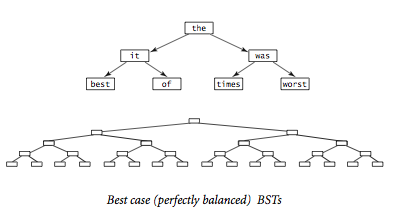
* *Search in a BST.* Suppose that you want to *search* for a node with a given key in a BST (or to *get* a value with a given key in a symbol table). There are two possible outcomes: the search might be successful (we find the key in the BST; in a symbol-table implementation we return the associated 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](http://introcs.cs.princeton.edu/java/44st/BST.java.html) is a symbol-table implementation based on these two recursive algorithms. Here is a useful [binary search tree application](http://introcs.cs.princeton.edu/java/GrowingTree).

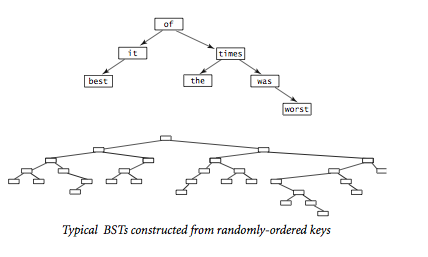
**Performance characteristics of BST.**

 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.

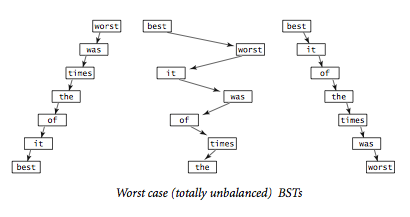
* *Best case.* In the best case, the tree is perfectly balanced (each Node has exactly two non-null children), with lg N nodes between the root and each leaf node. In such a tree, it is easy to see that the cost of an unsuccessful search is logarithmic, because that cost satisfies the same recurrence 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.



* *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.



* *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.



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](http://docs.oracle.com/javase/6/docs/api/java/util/TreeMap.html) implements a symbol table using this approach. Our symbol-table implementation [ST.java](http://introcs.cs.princeton.edu/java/44st/ST.java.html) 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:

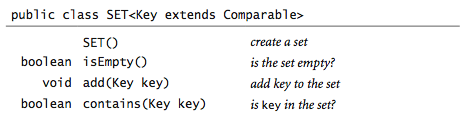
|  |
| --- |
| public void remove(Key key) {  if (contains(key))  put(key, null);  } |

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:



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](http://introcs.cs.princeton.edu/java/44st/SET.java.html) implements the set API. Program [DeDup.java](http://introcs.cs.princeton.edu/java/44st/DeDup.java.html) 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](http://www.zo.utexas.edu/faculty/antisense/tree.pdf), 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.