Ordered Symbol Table

Ordered (Sorted) Array Implementation

In this we maintain a sorted array of keys and values.

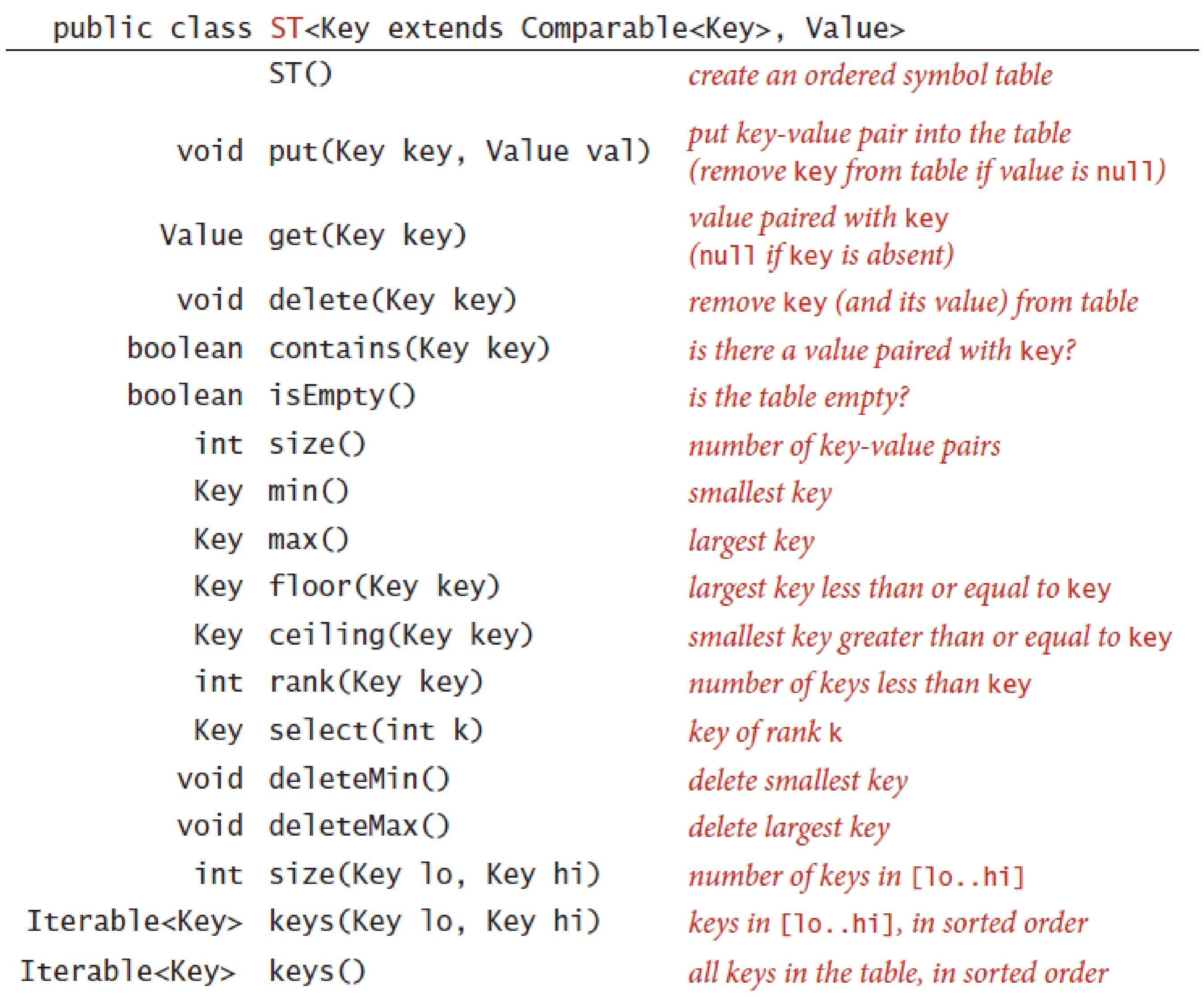
- Store in sorted order by key
- keys[i] = ith largest key
- values[i] = value associated with i^{th} largest key

Since the elements are sorted and stored in arrays, we can use a simple binary search for finding an element.

In typical applications, keys are Comparable objects, so the option exists of using the code a.compareTo(b) to compare two keys a and b.
Several symbol-table implementations take advantage of order among the keys that is implied by Comparable to provide efficient implementations of the put() and get() operations. More important, in such implementations, we can think of the symbol table as keeping the keys in order and consider a significantly expanded API that defines numerous natural and useful operations involving relative key order.

For example, suppose that your keys are times of the day. You might be interested in knowing the earliest
or the latest time, the set of keys that fall between two given times, and so forth.

API for ordered ST



`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

Minimum and maximum:
Perhaps the most natural queries for a set of ordered keys are to ask for the smallest and largest keys. In ordered symbol tables, we also have methods to delete the maximum and minimum keys (and their associated values). With this capability, the symbol table can operate like the IndexMin Priority Queue. The primary differences are that equal keys are allowed in priority queues but not in symbol tables and that ordered symbol tables support a much larger set of operations.

Floor and ceiling:
Given a key, it is often useful to be able to perform the floor operation (find the largest key that is less than or equal to
the given key) and the ceiling operation (find the smallest key that is greater than or equal to the given key). The nomenclature comes from functions defined on real numbers (the floor of a real number x is the largest integer that is smaller than or equal to x and the ceiling of a real number x is the smallest integer that is greater than or equal to x).

Rank and selection:
The basic operations for determining where a new key fits in the order are the rank operation (find the number of keys less than a given key) and the select operation (find the key with a given rank). Both property hold i == rank(select(i)) for all i between 0 and size() - 1 and all keys in the table satisfy key == select(rank(key)). For symbol tables, we need to perform these operations quickly, intermixed with insertions, deletions, and searches.

Range queries:
How many keys fall within a given range (between two given keys)? Which keys fall in a given range? The two-argument size() and keys() methods that answer these questions are useful in many applications, particularly in large databases. The capability to handle such queries is one prime reason that ordered symbol tables are so widely used in practice.

Exceptional cases:
When a method is to return a key and there is no key fitting the description in the table, our convention is to throw an exception (an alternate approach, which is also reasonable, would be to return null in such cases).

{ Implemented in code - BinarySearchTreeST.java / OrderedArrayST.java }