Week6

Studying Sorting through an Algorithmic Lens

In this section, we will study sorting as an algorithmic problem, addressing general issues about sorting algorithms.

Lower Bound for Sorting

Can we sort any faster than O(n log n) time?



[Explanation is in the book]

Linear-Time Sorting: Bucket-Sort and Radix-Sort

In the previous section, we showed that Ω(n log n) time is necessary, in the worst case, to sort an n-element sequence with a comparison-based sorting algorithm. Are there other kinds of sorting algorithms that can be designed to run asymptotically faster than O(n log n) time? Interestingly, such algorithms exist, but they require special assumptions about the input sequence to be sorted. Example:

- sorting integers from a known range,

- sorting character strings.

Bucket-Sort

Consider a sequence S of n entries whose keys are integers in the range [0,N−1], for some integer N ≥ 2, and suppose that S should be sorted according to the keys of the entries. In this case, it is possible to sort S in O(n+N) time.

[Check (both theory and implementation) online, preferably on udemy.]

[Check other than array implementations as well.]

Time Complexity:

Runs in O(n + N) time and uses O(n + N) space.

n : sequence size

N : range of values.

An important property of the bucket-sort algorithm is that it works correctly even if there are many different elements with the same key.

Stable Sorting

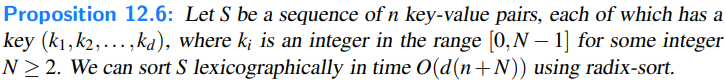
**Stable**: example: if we have the characters a-c-b-b, after sorting: a-b-b-c, but the two b characters did not change place with each other.

In bucket-sort stability is guaranteed as long as we ensure that all sequences act as queues, with elements processed and removed from the front of a sequence and inserted at the back. That is, when initially placing elements of S into buckets, we should process S from front to back, and add each element to the end of its bucket. Subsequently, when transferring elements from the buckets back to S, we should process each B[i] from front to back, with those elements added to the end of S.

Radix-Sort

[Check (both theory and implementation) online, preferably on udemy.]

[Check other than array implementations as well.]



Comparing Sorting Algorithms

Briefly:

O(n2) : insertion-sort and selection-sort

O(n log n) : heap-sort, merge-sort, and quick-sort[Read about it below.]

O(n) [Only for certain types of keys] : bucket-sort and radix-sort

Selection-Sort

Pretty bad in every case

Insertion-Sort

If implemented well, the running time of insertion-sort is O(n + m), where m is the number of inversions (that is, the number of pairs of elements out of order). Thus, insertion-sort is an excellent algorithm for sorting small sequences (say, less than 50 elements), because insertion-sort is simple to program, and small sequences necessarily have few inversions. Also, insertion-sort is quite effective for sorting sequences that are already “almost” sorted.

Apart from these special contexts, insertion-sort is also pretty bad.

Heap-Sort

The heap-sort runs in O(n log n) time in the worst case, which is optimal for comparison-based sorting methods. Heap-sort can easily be made to execute in-place, and is a natural choice on small- and medium-sized sequences, when input data can fit into main memory. However, heap-sort tends to be outperformed by both quick-sort and merge-sort on larger sequences. A standard heap-sort does not provide a stable sort, because of the swapping of elements.

Quick-Sort

Although its O(n2)-time worst-case performance makes quick-sort susceptible in real-time applications where we must make guarantees on the time needed to complete a sorting operation, we expect its performance to be O(nlog n) time, and experimental studies have shown that it outperforms both heap-sort and merge-sort on many tests. Quick-sort does not naturally provide a stable sort, due to the swapping of elements during the partitioning step.

Merge-Sort

Merge-sort runs in O(nlog n) time in the worst case. It is quite difficult to make merge-sort run in-place for arrays, and without that optimization the extra overhead of allocate a temporary array, and copying between the arrays is less attractive than in-place implementations of heap-sort and quick-sort for sequences that can fit entirely in a computer’s main memory.

Even so, merge-sort is an excellent algorithm for situations where the input is stratified across various levels of the computer’s memory hierarchy (e.g., cache, main memory, external memory). In these contexts, the way that merge-sort processes runs of data in long merge streams makes the best use of all the data brought as a block into a level of memory, thereby reducing the total number of memory transfers.

[Note: **Tim-Sort**: is a hybrid approach that is essentially a bottom-up merge-sort that takes advantage of initial runs in the data while using insertion-sort to build additional runs.]

Bucket-Sort and Radix-Sort

Finally, if an application involves sorting entries with small integer keys, character strings, or d-tuples of keys from a discrete range, then bucket-sort or radix-sort is an excellent choice, for it runs in O(d(n+N)) time, where [0,N −1] is the range of integer keys (and d = 1 for bucket sort). Thus, if d(n+N) is significantly “below” the nlog n function, then this sorting method should run faster than even quick-sort, heap-sort, or merge-sort.

Selection

There are a number of applications in which we are interested in identifying a single element in terms of its rank relative to the sorted order of the entire set. Eg. Finding minimum or maximum element, median, etc.

In this section, we discuss the general order-statistic problem of selecting the kth smallest element from an unsorted collection of n comparable elements. This is known as the **selection problem**.

Prune-and-Search [prune : ritkit]

We can indeed solve the selection problem in O(n) time for any value of k. The design pattern we use is known as **prune-and-search**.

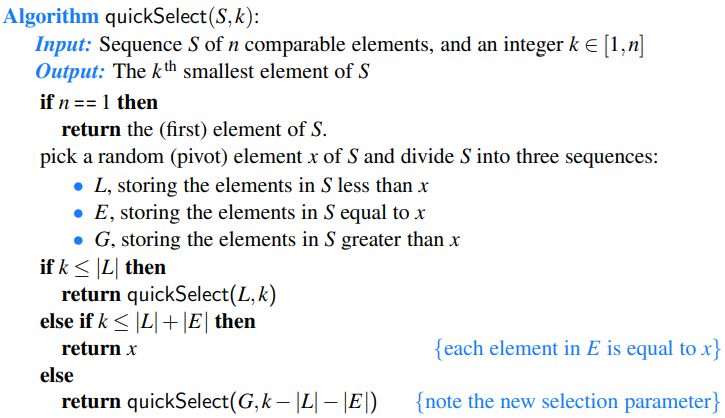
In applying this design pattern, we solve a given problem that is defined on a collection of n objects by pruning away a fraction of the n objects and recursively solving the smaller problem. When we have finally reduced the problem to one defined on a constant-sized collection of objects, we then solve the problem using some brute-force method. Returning back from all the recursive calls completes the construction. In some cases, we can avoid using recursion, in which case we simply iterate the prune-and-search reduction step until we can apply a brute-force method and stop. Incidentally, the binary search method described earlier [in Week1] is an example of the prune-and-search design pattern.

Randomized Quick-Select

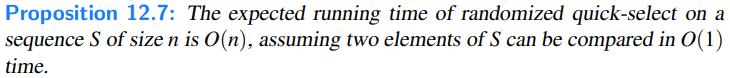
In applying the prune-and-search pattern to finding the k th smallest element in an unordered sequence of n elements, we describe a simple and practical algorithm, known as **randomized quick-select**. This algorithm runs in O(n) **expected** time. We note though that randomized quick-select runs in O(n2) time in the worst case.

Suppose we are given an unsorted sequence S of n comparable elements together with an integer k ∈ [1,n]. At a high level, the quick-select algorithm for finding the kth smallest element in S is similar to the randomized quick-sort algorithm described earlier. We pick a “pivot” element from S at random and use this to subdivide S into three subsequences L, E, and G, storing the elements of S less than, equal to, and greater than the pivot, respectively. In the prune step, we determine which of these subsets contains the desired element, based on the value of k and the sizes of those subsets. We then recur on the appropriate subset, noting that the desired element’s rank in the subset may differ from its rank in the full set.

Pseudocode:



Analyzing Randomized Quick-Select



[Explanation with recurrence equation is in the book.]

Maps

A map stores keyvalue pairs (k,v), which we call entries, where k is the key and v is its corresponding value. Keys are required to be unique, so that the association of keys to values defines a mapping.

[It is like an array, but the key doesn’t need to be a number, and not related to the position in the array.]

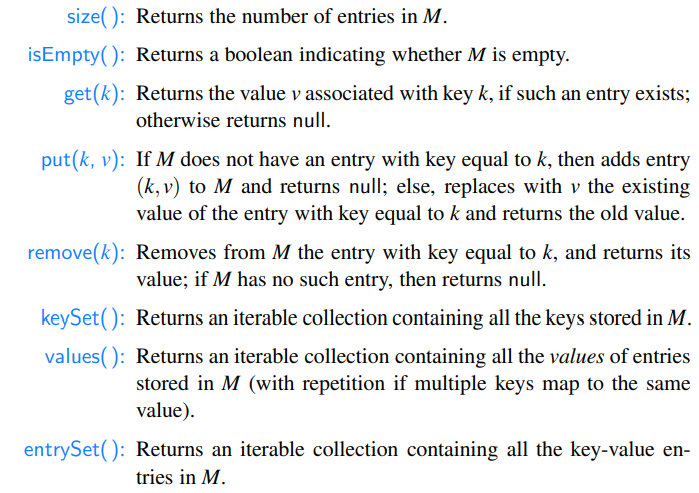
Examples:

studentID -> student’s name, student’s address, student’s grades

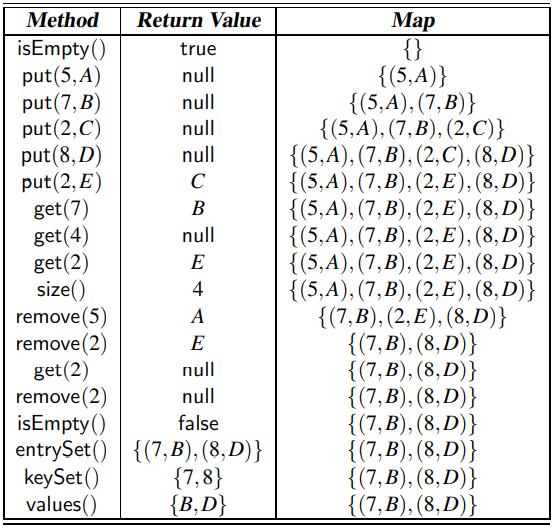
[www.wiley.com](http://www.wiley.com) -> 208.215.179.146

color(turquoise)-> r(64), g(224), b(208)

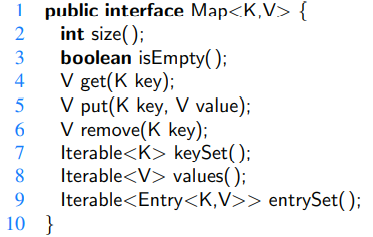
Methods of the Map ADT



Example Usage:

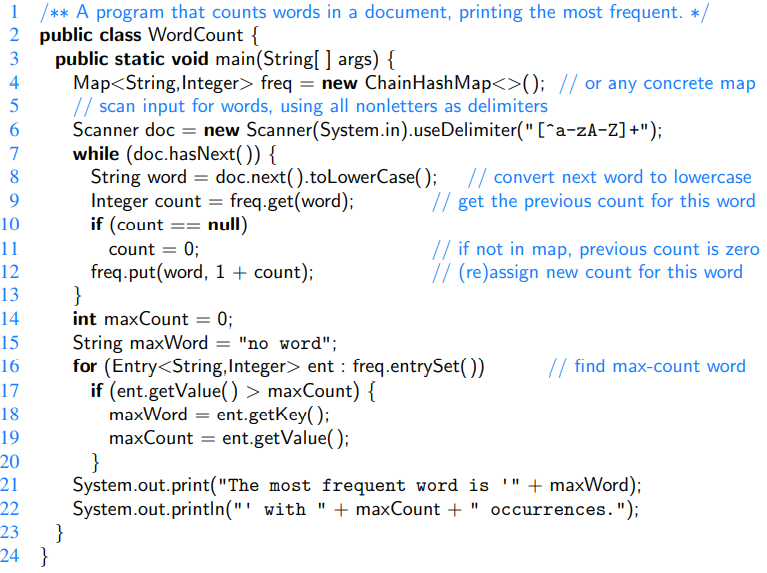


A Java Interface for the Map ADT



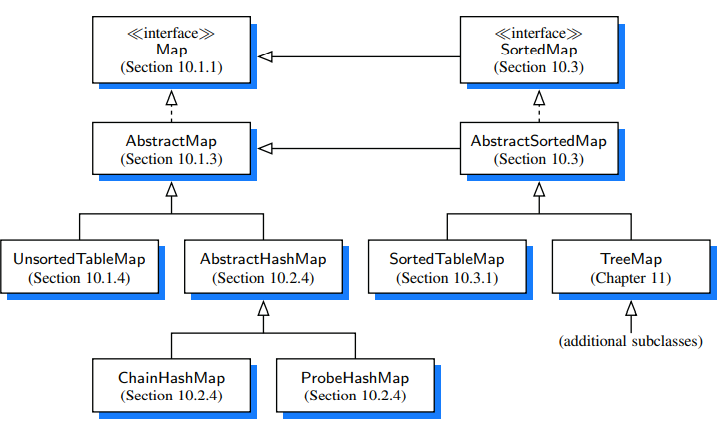
Application: Counting Word Frequencies

The problem of counting the number of occurrences of words in a document. We can use words as keys and word counts as values.

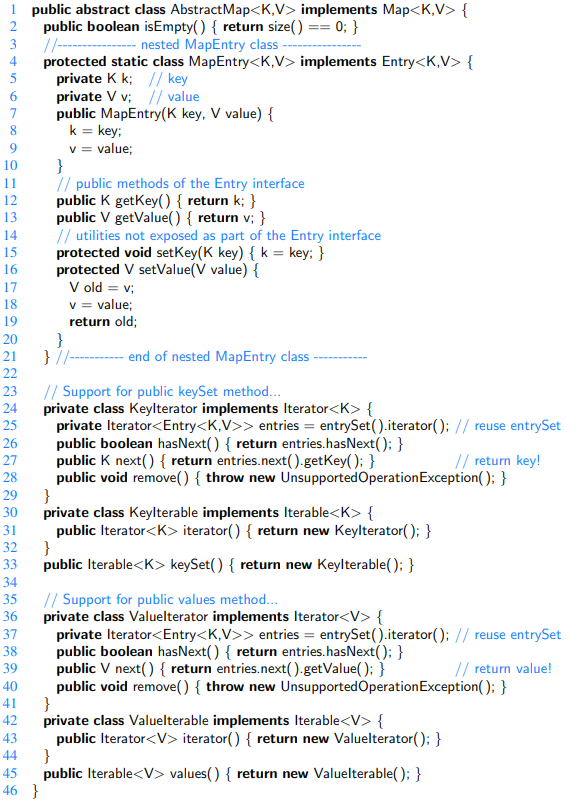


An AbstractMap Base Class

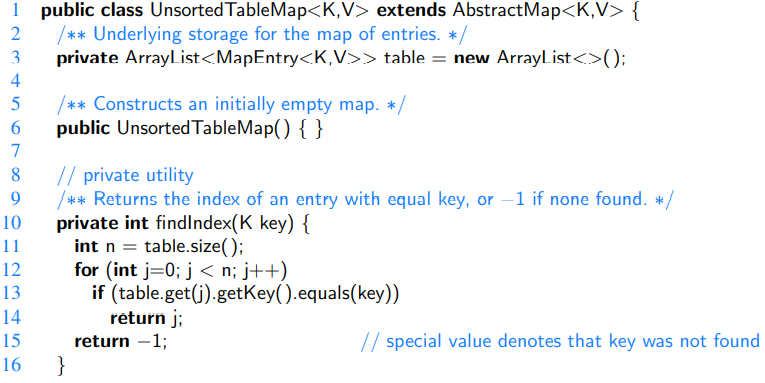
We will implement different types of maps:



It is good to have a base class, to help ourselves with code reusability.



A Simple Unsorted Map Implementation

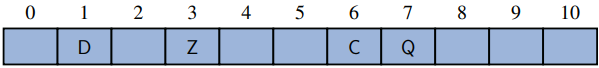




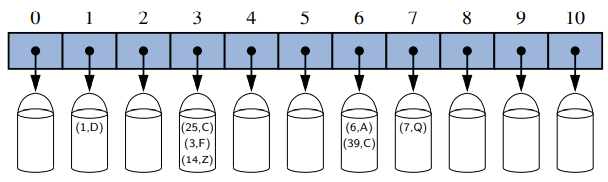
Unfortunately, the UnsortedTableMap class on the whole is not very efficient. On a map with n entries, each of the fundamental methods takes O(n) time in the worst case because of the need to scan through the entire list when searching for an existing entry.

Hash Tables

A lookup table with length 11 for a map containing entries (1,D), (3,Z), (6,C), and (7,Q):



It can happen that more than one keys are mapped to the same index by a hash function (called collision). In this case we can use a bucket array. A bucket array of capacity 11 with entries (1,D), (25,C), (3,F), (14,Z), (6,A), (39,C), and (7,Q), using a simple hash function:

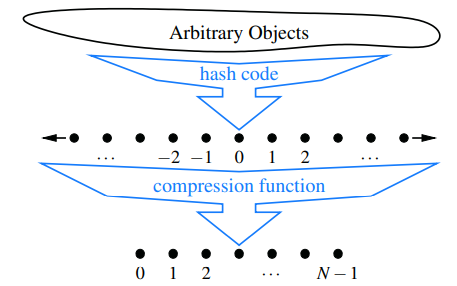


Hash Functions

The goal of a hash function, h, is to map each key k to an integer in the range [0,N − 1], where N is the capacity of the bucket array for a hash table. h(k) is an index in our bucket array. We store the entry (k,v) in the bucket A[h(k)].

Let’s always try to minimize collisions.

It is common to view the evaluation of a hash function, h(k), as consisting of two portions—a hash code that maps a key k to an integer, and a compression function that maps the hash code to an integer within a range of indices, [0,N −1], for a bucket array.



Hash Codes

The first action that a hash function performs is to take an arbitrary key k in our map and compute an integer that is called the hash code for k. We desire that the set of hash codes assigned to our keys should avoid collisions as much as possible.

Treating the Bit Representation as an Integer

For any data type X that is represented using at most as many bits as our integer hash codes, we can simply take as a hash code for X an integer interpretation of its bits. Eg. In Java if it is an int that we need a hash code for, because java uses 32 bit hash codes, we can assign a value to each int.

What if it is a long or a double, which are 64 bits? Then we can take the higher order 32 bits and the lower order 32 bits of the double or long, and we “comb”them together, by eg. adding them together, or using a xor operator(in Java: ^). This way we get 32 numbers instead of 64, and now we can assign a value to each of them, just like at the int example.

Polynomial Hash Codes



What if the length is not always the same? Eg. Strings. The above mentioned technique doesn’t work. Instead, we can take the components (x0,x1,...,xn−1) of an object x as its coefficients.

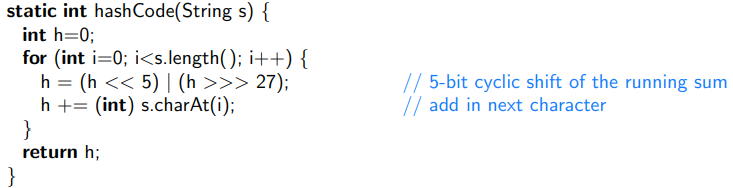
This polynomial can be computed by Horner’s rule:



Cyclic-Shift Hash Codes

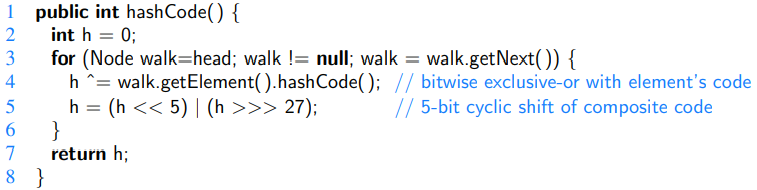
For example, a 5-bit cyclic shift of the 32-bit value **00111**101100101101010100010101000 is achieved by taking the leftmost five bits and placing those on the rightmost side of the representation, resulting in 101100101101010100010101000**00111**.

In java:



Hash Codes in Java

As an example of how to properly implement hashCode for a user-defined class, we will revisit the SinglyLinkedList class. We defined the equals method for that class, so that two lists are equivalent if they represent equal-length sequences of elements that are pairwise equivalent. We can compute a robust hash code for a list by taking the exclusive-or of its elements’ hash codes, while performing a cyclic shift.



Compression Functions

The hash code for a key k will typically not be suitable for immediate use with a bucket array, because the integer hash code may be negative or may exceed the capacity of the bucket array. Thus, once we have determined an integer hash code for a key object k, there is still the issue of mapping that integer into the range [0,N −1]. This computation, known as a compression function, is the second action performed as part of an overall hash function. A good compression function is one that minimizes the number of collisions for a given set of distinct hash codes.

The Division Method



where N, the size of the bucket array, is a fixed positive integer. Additionally, if we take N to be a prime number, then this compression function helps “spread out” the distribution of hashed values. Indeed, if N is not prime, then there is greater risk that patterns in the distribution of hash codes will be repeated in the distribution of hash values, thereby causing collisions. For example, if we insert keys with hash codes {200,205,210,215,220,... ,600} into a bucket array of size 100, then each hash code will collide with three others. But if we use a bucket array of size 101, then there will be no collisions.

Choosing N to be a prime number is not always enough tho.

The MAD method

Multiply-Add-and-Divide method. This method maps an integer i to



where N is the size of the bucket array, p is a prime number larger than N, and a and b are integers chosen at random from the interval [0, p− 1], with a > 0.

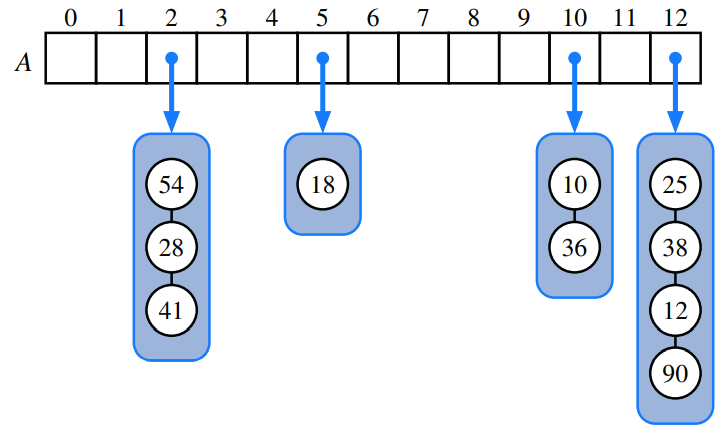
Collision-Handling Schemes

The main idea of a hash table is to take a bucket array, A, and a hash function, h, and use them to implement a map by storing each entry (k,v) in the “bucket” A[h(k)]. This simple idea is challenged, however, when we have two distinct keys, k1 and k2, such that h(k1) = h(k2). The existence of such collisions prevents us from simply inserting a new entry (k,v) directly into the bucket A[h(k)]. It also complicates our procedure for performing insertion, search, and deletion operations.

Separate Chaining

On the figure below:

A hash table of size 13, storing 10 entries with integer keys, with collisions resolved by separate chaining. The compression function is h(k) = k mod 13. For simplicity, we do not show the values associated with the keys.



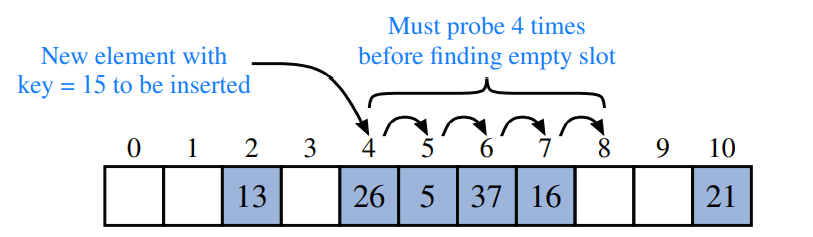
Load factor: λ = n/N. [Explanation in the book]

Open Addressing

Open addressing requires that the load factor is always at most 1 and that entries are stored directly in the cells of the bucket array itself. There are several variants of this approach, collectively referred to as open addressing schemes, which we discuss next.

Linear Probing and Its Variants

With this approach, if we try to insert an entry (k,v) into a bucket A[ j] that is already occupied, where j = h(k), then we next try A[(j +1) mod N]. If A[(j +1) mod N] is also occupied, then we try A[(j + 2) mod N], and so on, until we find an empty bucket that can accept the new entry.



To implement a deletion, we cannot simply remove a found entry from its slot in the array. A typical way to get around this difficulty is to replace a deleted entry with a special “defunct” sentinel object.

Linear probing suffers from an additional disadvantage. It tends to cluster the entries of a map into contiguous runs, which may even overlap. Such contiguous runs of occupied hash cells cause searches to slow down considerably.

Another open addressing strategy, known as **quadratic probing**, iteratively tries the buckets A[(h(k)+ f(i)) mod N], for i = 0,1,2,..., where f(i) = i 2, until finding an empty bucket. (It solves the clustering problem of linear probing, but creates its own clustering.)

An open addressing strategy that does not cause clustering of the kind produced by linear probing or the kind produced by quadratic probing is the **double hashing** strategy.

In this approach, we choose a secondary hash function, h′ , and if h maps some key k to a bucket A[h(k)] that is already occupied, then we iteratively try the buckets A[(h(k) + f(i)) mod N] next, for i = 1,2,3,..., where f(i) = i · h′ (k). In this scheme, the secondary hash function is not allowed to evaluate to zero; a common choice is h′ (k) = q−(k mod q), for some prime number q < N. Also, N should be a prime.

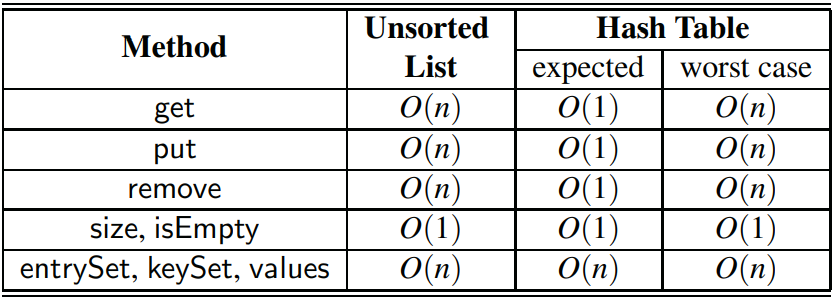
Another approach to avoid clustering with open addressing is to iteratively try buckets A[(h(k) + f(i)) mod N] where f(i) is based on a pseudorandom number generator, providing a repeatable, but somewhat arbitrary, sequence of subsequent probes that depends upon bits of the original hash code.

Load Factors, Rehashing, and Efficiency

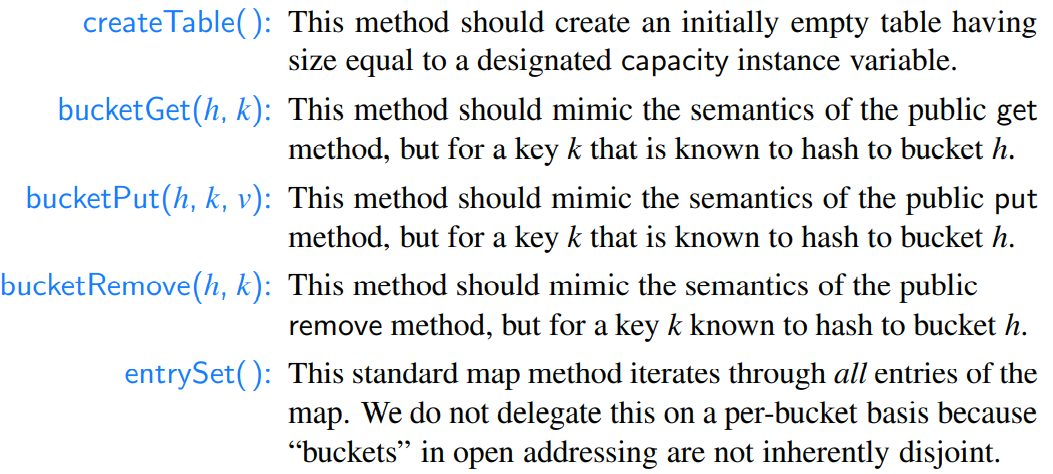
In the hash table schemes described thus far, it is important that the load factor, λ = n/N, be kept below 1. With separate chaining, as λ gets very close to 1, the probability of a collision greatly increases, which adds overhead to our operations, since we must revert to linear-time list-based methods in buckets that have collisions.

Efficiency of Hash Tables

We let n denote the number of entries in the map, and we assume that the bucket array supporting the hash table is maintained such that its capacity is proportional to the number of entries in the map.



Java Hash Table Implementation

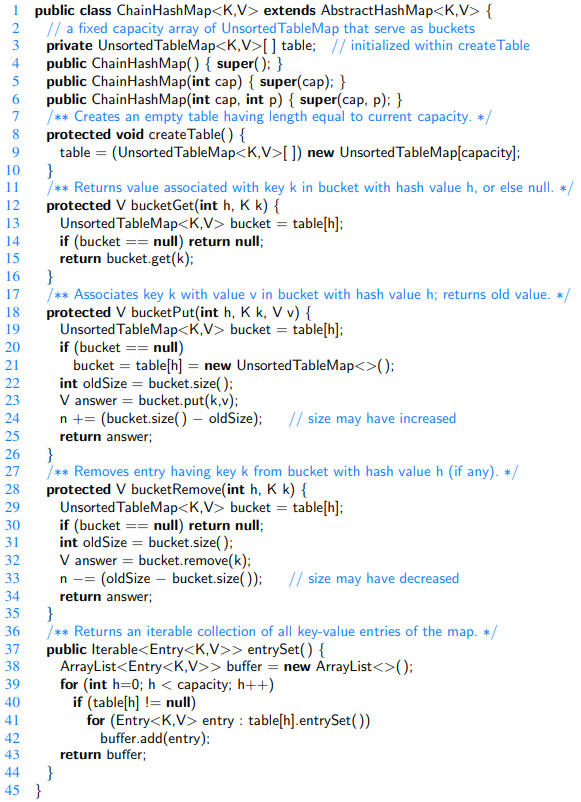


A base class for our hash table implementations, extending the AbstractMap class from earlier:

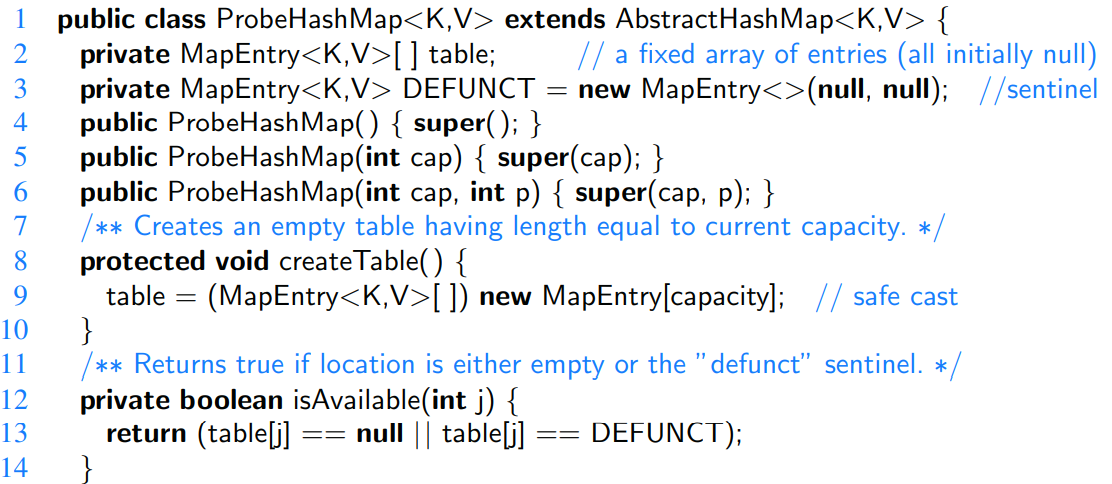


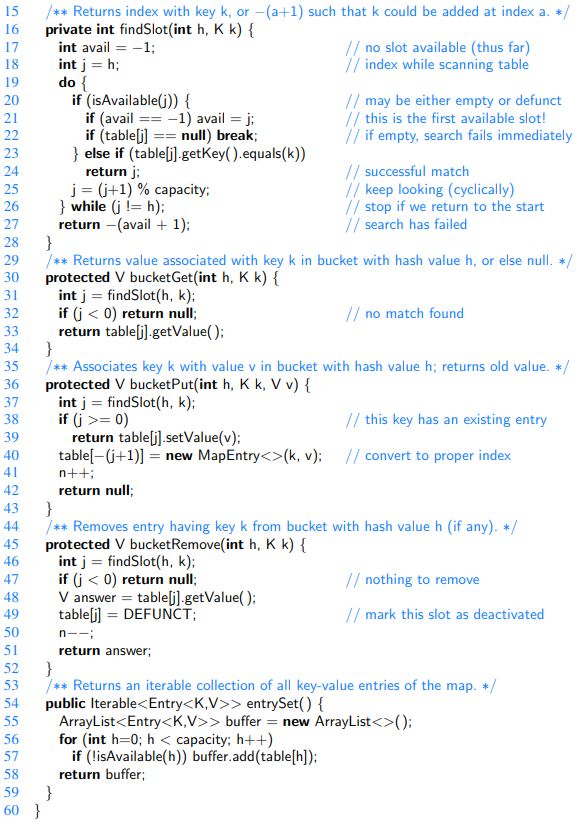
Separate Chaining

To represent each bucket for separate chaining, we use an instance of the simpler UnsortedTableMap class from earlier. The advantage of using a map for each bucket is that it becomes easy to delegate responsibilities for top-level map operations to the appropriate bucket.



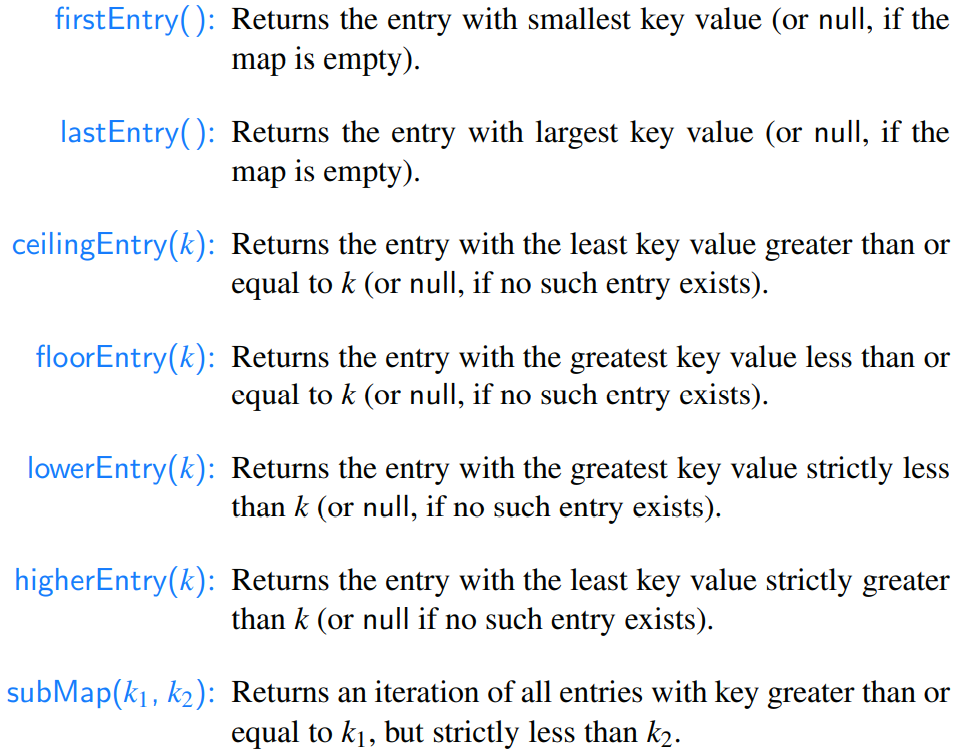
Linear Probing





Sorted Maps

The sorted map ADT includes all behaviors of the standard map, plus the following:

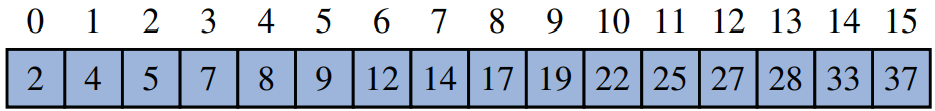


To motivate the use of a sorted map, consider a computer system that maintains information about events that have occurred (such as financial transactions), with a time stamp marking the occurrence of each event. The (unsorted) map ADT does not provide any way to get a list of all events ordered by the time at which they occur, or to search for which event occurred closest to a particular time.

Sorted Search Tables

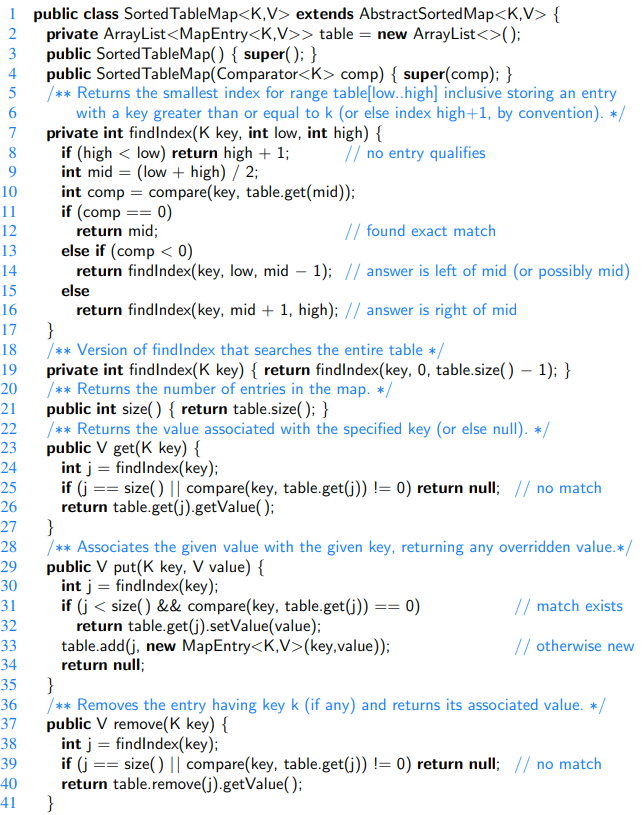
Several data structures can efficiently support the sorted map ADT. In this section, we will begin by exploring a simple implementation of a sorted map. We store the map’s entries in an array list A so that they are in increasing order of their keys. This is a Sorted Search Table.

On the figure: Realization of a map by means of a sorted search table. We show only the keys for this map, so as to highlight their ordering.



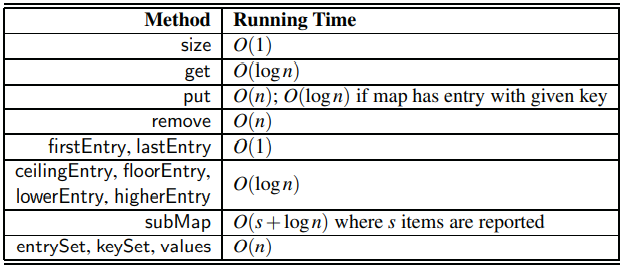
Has a space requirement that is O(n). Advantage: The primary advantage of this representation, and our reason for insisting that A be array-based, is that it allows us to use the binary search algorithm for a variety of efficient operations.

Binary Search and Inexact Searches





Time Complexities



Two Applications of Sorted Maps

To apply a sorted map, keys must come from a domain that is totally ordered. Furthermore, to take advantage of the inexact or range searches afforded by a sorted map, there should be some reason why nearby keys have relevance to a search.

[Check both in the book, pg 433(451)]

Flight Databases

Maxima Sets

Sets, Multisets and Multimaps

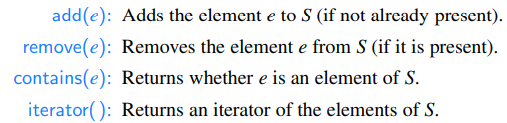
• A set is an unordered collection of elements, without duplicates, that typically supports efficient membership tests. In essence, elements of a set are like keys of a map, but without any auxiliary values.

• A multiset (also known as a bag) is a set-like container that allows duplicates.

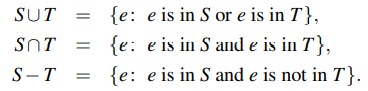
• A multimap is similar to a traditional map, in that it associates values with keys; however, in a multimap the same key can be mapped to multiple values. For example, the index of this book (page 714) maps a given term to one or more locations at which the term occurs elsewhere in the book.

The Set ADT

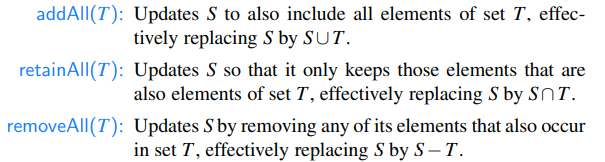
Methods:



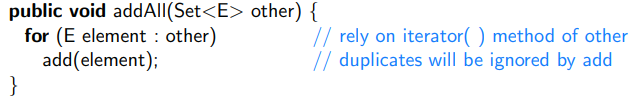
Set operations:



In java.util.Set:

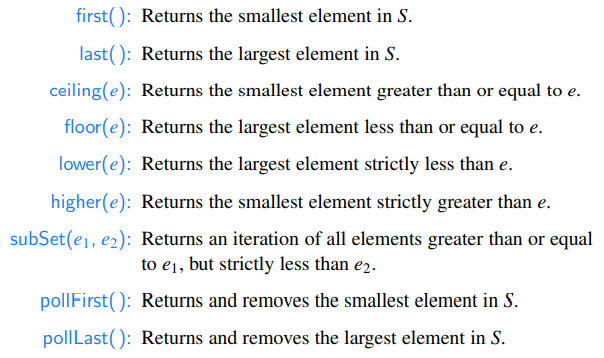


The **template method pattern** can be applied to implement each of the methods addAll, retainAll, and removeAll using only calls to the more fundamental methods add, remove, contains, and iterator. An example:



Sorted Sets

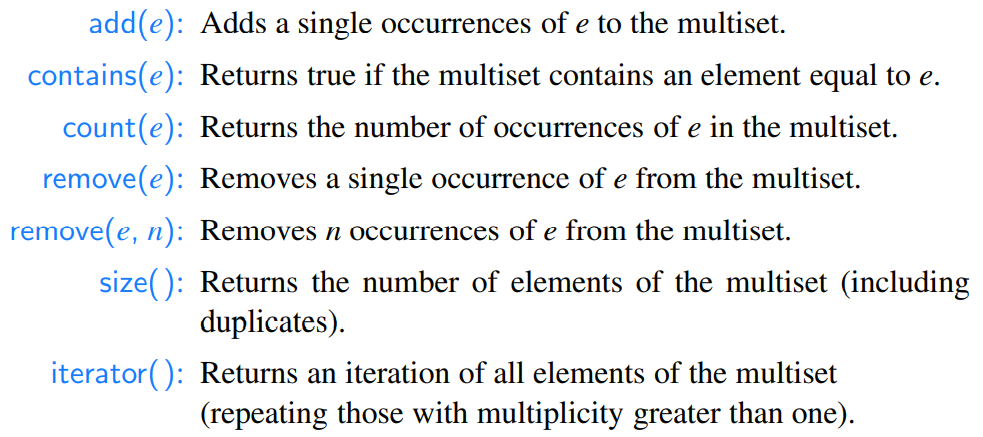
Methods:



Although a set is a completely different abstraction than a map, the techniques used to implement the two can be quite similar. In effect, a set is simply a map in which (unique) keys do not have associated values. Therefore, any data structure used to implement a map can be modified to implement the set ADT with similar performance guarantees.

The Multiset ADT

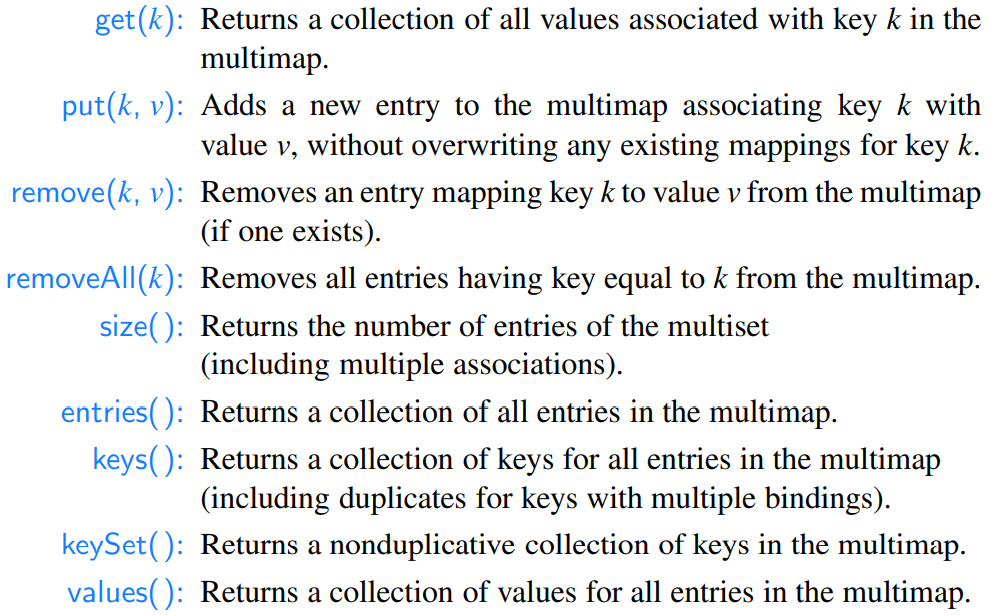
Methods:



The Multimap ADT

Like a map, a multimap stores entries that are key-value pairs (k,v), where k is the key and v is the value. Whereas a map insists that entries have unique keys, a multimap allows multiple entries to have the same key, much like an English dictionary, which allows multiple definitions for the same word. That is, we will allow a multimap to contain entries (k,v) and (k,v′ ) having the same key.

Methods:



Implementation:

