A hash table is a data structure that allows elements to be stored in such a way that they can be retrieved, by key, in a constant amount of time, essentially independent of the number of elements in the table.

The basic idea is a variant of the notion of an array. Given an index into an array, one can recover the element stored at that index in constant time, because one can compute the address of the storage location directly from the origin of the array and the index. Similarly, in a hash table, one uses the key to compute the location in the table at which the desired element should be found.

If the range of possible keys is small, the computation is trivial; one simply uses the keys themselves as array subscripts. But there is a problem if the range of possible keys is vastly larger than the number of elements to be stored. For instance, at Grinnell College, student IDs are nine-digit numbers (that is, in the range from 000000000 to 999999999), while there are only about 1650 students. Allocating an array with a billion elements just to permit constant-time access to 1650 records would not be an effective use of storage.

So one must interpose some computation between the key and the array subscript — a computation that is typically encapsulated in a hash function that takes keys as arguments and returns subscripts into some more appropriately sized array as values. To find out where in a hash table the element with a given key is stored, one applies the hash function to the key and uses the result as an index into the array.

Of course, since the hash function maps a gigantic range of possible keys into a much smaller range of array subscripts, it can’t be one-to-one. On the contrary, it is inevitable that there will be cases in which the hash function assigns the same array subscript to different keys. When the distinct keys of two elements of the hash table are mapped to the same array subscript, collision occurs. The implementer of a hash table must provide some mechanism for resolving collisions, that is, for finding an alternative storage location for an element that cannot be stored in the (already occupied) position proposed by the hash function.

There are various mechanisms for resolving collisions. The earliest proposal was to use the array subscript returned by the hash function as the starting point for a linear search for an unused location within the table; as soon as the linear search encounters a position that is not already occupied, the incoming element can be inserted. If the end of the table is encountered before an unused location is reached, the search “wraps around” to the beginning of the array and continues from there.

This linear probing strategy, however, does not work well, because the data tend to clump together as the table fills up, leading to long stretches of occupied slots separated by sparsely occupied stretches. A better idea, called secondary hashing, applies another hash function to the key to figure out how many positions in the array to jump over, after finding an occupied position, before trying to insert a new element again.

Still another idea is to implement the hash table not as an array of elements but as an array of lists of elements. The hash function is applied to the key to determine which of these lists the new element should be added to; in the event of a collision, one simply puts all of the elements that hash to the same array subscript into the same list, or bucket, as it is sometimes called.

As compared with linear probing and secondary hashing, this method has the advantage that it can if necessary accommodate more elements than there are positions in the array, though with a progressive degradation of performance as the average list grows longer and the linear search down
such a list comes to occupy a larger fraction of the running time.

Also, it is far easier to delete an element from a hash table that uses buckets than from one that uses linear probing or secondary hashing as its collision-resolution mechanism. In many applications, however, deletions are never needed, or can be saved up and performed at a time when the hash table must be completely rebuilt anyway.

Nowadays most implementations of hash tables resolve collisions by using buckets and monitor the ratio between the number of elements currently stored in a hash table and the number of buckets provided for those elements. If this ratio gets too large, the implementation intervenes to allocate a new, larger array of buckets and to move all of the elements from the old array into the new one. This is a resource-intensive operation requiring an amount of time proportional to the number of elements currently stored in the hash table, since every key must be rehashed to determine which of the buckets in the new array it should be transferred to. So programmers who use hash tables are encouraged to estimate the maximum number of elements that their application will ever need to store simultaneously and to provide that estimate to the hash-table constructor so that it can generate a sufficiently large array to begin with, one that will never need resizing. Implementations also try to reduce the frequency of resizing operations by making the new array of buckets much larger than the old one whenever a resizing is performed.

R7RS Scheme does not support hash tables as a primitive data type, but there are hash-table libraries that one can import, and I have provided one in /home/reseda/computational-linguistics/code/cl/hash-tables.ss, using an interface specified by Panu Kalliokoski in his Scheme Request for Implementation entitled “Basic Hash Functions.” (This document is available on the World Wide Web at https://srfi.schemers.org/srfi-69/srfi-69.html.)

This interface provides specific hash functions for the most common types of keys (strings and symbols; for integer keys, one can use the standard modulo procedure as a hash function) and two generic hash functions that allow keys of any kind except procedures and records. (The procedure hash guarantees that any two values that satisfy equal? yield the same result when hashed; the procedure hash-by-identity restricts that guarantee to values that satisfy eq?.)

It provides a constructor with optional arguments that allow the caller to specify a customized binary predicate for determining whether two keys should be counted as equivalent, a customized hash function, and the number of buckets that should be allocated initially. When supplying a hash function and an equivalence predicate, the caller must ensure that the hash function yields the same result when given either of two values that satisfy the equivalence predicate.

There are also specialized constructors for hash tables with keys that are integers, symbols, or strings, and a separate constructor of hash tables keyed by strings that should be treated as case-insensitive (so that, for example, "foo", "FOO", "Foo", and "fOo" all count as the same key).

It is also possible to construct a hash table by applying the alist->hash-table procedure to an association list. The car of each pair in the association list is treated as a key and the corresponding cdr is the value associated with that key.

The interface provides a type predicate, hash-table?, and selectors to determine what hash function a given hash table is using, what equivalence predicate it is applying to determine whether keys match, and how many elements it is currently storing.

The hash-table-keys procedure returns a list of all the keys in a given hash table. Similarly, hash-table-values returns a list of all the values, and hash-table->alist a list of all the key-value pairs. The order of the elements in these lists is unspecified and is unlikely to reflect either the order in which elements were added to the hash table or any “natural” ordering of the keys or values.

To get the value that a hash table associates with a given key, one uses the hash-table-ref, which performs the lookup. It is an error to call this procedure with a key that isn’t present in the hash table at all, but the implementation provides two ways to cope with this situation:
The caller can provide, as an optional third argument, a zero-argument “failure procedure” that is invoked when the “no such key” error would otherwise occur. If this third argument is present, `hash-table-ref` returns the value that the failure procedure produces if it can’t find the key. Alternatively, there is a separate `hash-table-ref/default` procedure that always takes three arguments and uses the third as a “default value,” to be returned if no element in the hash table has the specified key.

The `hash-table-set!` procedure puts an entry into a given hash table. The caller specifies the hash table and the key and the value for the new entry. If the hash table already has an entry indexed by a key that matches the one that the caller supplies, the new value overwrites the old one irreversibly and the number of elements stored in the hash table remains the same. Otherwise, the new key-value association is added to the appropriate bucket and the number of elements increases by one. (This is the point at which the hash table may need to be resized, so `hash-table-set!` is not guaranteed to be a constant-time operation on each use.)

You can find out whether a given hash table already contains an entry with a given key by applying the predicate `hash-table-exists?` to the hash table and the key.

In many cases, the programmer wants to use the value associated with a key that is already present in a hash table to help compute a new value that will replace the old one. It’s not too difficult to code this up by calling `hash-table-ref` and using the result in a call to `hash-table-set!`, but it’s such a common operation that the interface designer provided a `hash-table-update!` procedure to make it easy. Like `hash-table-set!`, `hash-table-update!` takes a hash table and a key as its first two arguments, but its third argument is an update function that takes one argument, the old value, and returns the new value. The `hash-table-update!` procedure extracts the old value from the table, applies the update function to it, and stores the result back into the table, overwriting the old value.

One common application is for hash tables is to tally occurrences of various key values as they arrive from some source that generates them. In that case, the update function is simply \( (r-s + 1) \), the operator section that adds 1 to its argument.

To deal with missing keys, `hash-table-update!` provides the same three alternatives as `hash-table-ref`. The programmer can choose to treat an attempt to update an entry that isn’t there to begin with as an error, or can supply a failure procedure as an additional argument, or can invoke `hash-table-update!/default` with a default value to fall back on if the hash table doesn’t have an entry with the specified key. Note in this last case that the update function is still applied to the default value before it is stored in the hash table. For example, the call `(hash-table-update!/default ht "foo" (r-s + 1) 0)` adds an entry for the key "foo" to the hash table `ht` if no such entry is already present, but the associated value is 1, not 0, since the update function is applied to the default value 0 before the new key-value association is put into the table.

There is a `hash-table-copy` procedure that takes a hash table as its argument and returns a freshly allocated hash table storing the same elements, and a `hash-table-merge!` procedure that takes two hash tables as arguments and adds all of the elements of the second into the first. This process is destructive, in the sense that an existing element in the first hash table can be overwritten by an element from the second hash table with the same key. A programmer who needs both of the original hash tables and the combined one can write something like

```
(let (((combined (hash-table-copy ht1)))
       (hash-table-merge! combined ht2)
     ...
)
```
to get this effect.

Finally, there are two higher-order procedures for operating on all of the elements of a hash
table. One is \texttt{hash-table-walk}, which is the analogue of the list operation \texttt{for-each}. It applies a given binary procedure to the key and value of each of the associations in a given hash table for the side effects only.

The other is \texttt{hash-table-fold}, which is a little like the fold operations for lists in CSC 151. Given a hash table, a three-argument procedure, and a starting value, the \texttt{hash-table-fold} procedure goes through all of the elements of the hash table, successively applying the three-argument procedure to the key and value stored in each hash node and to an “accumulator” value that is initially equal to the starting value. During this process, each value returned by the three-argument procedure becomes the accumulator argument for the call that processes the next element. The value returned by the final call, when the last element is processed, is the value returned by the \texttt{hash-table-fold} procedure.

The designer of this interface hoped to provide a repertoire of procedures that would include all of the common hash-table operations and make it straightforward to develop procedures for as many as possible of the less common ones. We’ll have occasion to test his design work in several applications for natural language processing that use hash tables to implement data structures such as language models and context-free grammars.