Chapter 9

SETS AND MAPS
Chapter Objectives

- To understand the C++ `map` and `set` containers and how to use them
- To learn about hash coding and its use to facilitate efficient search and retrieval
- To study two forms of hash tables—open addressing and chaining—and to understand their relative benefits and performance trade-offs
Chapter Objectives (cont.)

- To learn how to implement both hash table forms
- To be introduced to the implementation of maps and sets
- To see how two earlier applications can be implemented more easily using map objects for data storage
Introduction

- In Chapter 4 we studied the C++ containers, focusing on the sequential containers `vector` and `list`.

- Searching for a particular value in a sequential container is generally $O(n)$.
  - An exception is a binary search of a sorted object, which is $O(\log n)$. 
In this chapter, we consider another part of the container framework, the associative containers.

Associative containers:
- are not indexed
- do not reveal the order of insertion of items
- enable efficient search and retrieval of information
- allow removal of elements without moving other elements around
Associative containers include the set and the map.

The set is an implementation of the Set ADT.

The map facilitates efficient search and retrieval of entries that consist of pairs of objects:
- The first object in the pair is the key.
- The second object is the information associated with that key.
Section 9.1

Associative Container Requirements
The Set Hierarchy

Concept: associative_container

Subclasses:
- set
- multiset
- map
- multimap
The set Abstraction

- A set is a collection that contains no duplicate elements and at most one null element
  - adding "apples" to the set
    \{"apples", "oranges", "pineapples"\}
    results in the same set (i.e. no change)

- Operations on sets include:
  - testing for membership
  - adding elements
  - removing elements
  - union \(A \cup B\)
  - intersection \(A \cap B\)
  - difference \(A - B\)
  - subset \(A \subset B\)
The set Abstraction (cont.)

- The union of two sets A, B is a set whose elements belong either to A or B or to both A and B
  Example: $\{1, 3, 5, 7\} \cup \{2, 3, 4, 5\}$ is $\{1, 2, 3, 4, 5, 7\}$

- The intersection of sets A, B is the set whose elements belong to both A and B
  Example: $\{1, 3, 5, 7\} \cap \{2, 3, 4, 5\}$ is $\{3, 5\}$

- The difference of sets A, B is the set whose elements belong to A but not to B
  Examples: $\{1, 3, 5, 7\} - \{2, 3, 4, 5\}$ is $\{1, 7\}; \{2, 3, 4, 5\} - \{1, 3, 5, 7\}$ is $\{2, 4\}$

- Set A is a subset of set B if every element of set A is also an element of set B
  Example: $\{1, 3, 5, 7\} \subset \{1, 2, 3, 4, 5, 7\}$ is true
The multiset

- The **multiset** is the same as the **set** except that it does not impose the requirement that the items be unique.
- The **insert** function always inserts a new item, and duplicate items are retained.
- However, the **erase** function removes all occurrences of the specified item because there may be duplicates.
Maps and Multimaps

Section 9.2
Maps and Multimaps

- The map is related to the set
- Mathematically, a map is a set of ordered pairs whose elements are known as the key and the value
- Keys must be unique, but values need not be unique
- You can think of each key as a “mapping” to a particular value
- A map provides efficient storage and retrieval of information in a table
- A map can have many-to-one mapping: \((B, \text{Bill}), (B2, \text{Bill})\)

\[
\{(J, \text{Jane}), (B, \text{Bill}), (S, \text{Sam}), (B1, \text{Bob}), (B2, \text{Bill})\}
\]
Maps and Multimaps (cont.)

- In an onto mapping, all the elements of values have a corresponding member in keys.
- A map can be used to enable efficient storage and retrieval of information in a table.
- The key is a unique identification value associated with each item stored in a table.
- The “value” in the key/value pair might be an object, or even a pointer to an object, with objects in the class distinguished by some attribute associated with the key that is then mapped to the value.
Maps and Multimaps (cont.)

- When information about an item is stored in a table, the information should have a unique ID
- A unique ID may or may not be a number
- This unique ID is equivalent to a key

<table>
<thead>
<tr>
<th>Type of item</th>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>University student</td>
<td>Student ID number</td>
<td>Student name, address, major, grade point average</td>
</tr>
<tr>
<td>Online store customer</td>
<td>E-mail address</td>
<td>Customer name, address, credit card information, shopping cart</td>
</tr>
<tr>
<td>Inventory item</td>
<td>Part ID</td>
<td>Description, quantity, manufacturer, cost, price</td>
</tr>
</tbody>
</table>
In comparing maps to indexed collections, you can think of the keys as selecting the elements of a map, just as indexes select elements in a vector object.

The keys for a map, however, can have arbitrary values (not restricted to 0, 1, 2, and so on, as for indexes).

The subscript operator is overloaded for the map class, so you can have statements of the form:

```
v = a_map[k]; // Assign to v the value for key k
a_map[k] = v1; // Set the value for key k to v1
```

where \( k \) is of the key type, \( v \) and \( v1 \) are of the value type, and \( a\_map \) is a map.
The map Functions

- A map is effectively defined as a set whose items are pairs.
- The member functions defined for both are the same except for the type of the parameters.
- The map is a template class that takes the following template parameters:
  - `Key_Type`: The type of the keys contained in the key set.
  - `Value_Type`: The type of the values in the value set.
  - `Compare`: A function class that determines the ordering of the keys; by default this is the less-than operator.
  - `Allocator`: The memory allocator for key objects; we will use the library-supplied default.
The map Functions (cont.)

The following statements build a map object:

```cpp
map<string, string> a_map;
a_map["J"] = "Jane";
a_map["B"] = "Bill";
a_map["S"] = "Sam";
a_map["B1"] = "Bob";
a_map["B2"] = "Bill";
```
cout << "B1 maps to " << a_map["B1"] << endl;

displays:

B1 maps to Bob
Map Interface (cont.)

cout << "Bill maps to " << a_map["Bill"] << endl;

displays:

Bill maps to

- (a side effect of this statement is that "Bill" would now be a key in the map associated with the empty string)
The `multimap`

- Like the `multiset`, the `multimap` removes the restriction that the keys are unique.
- The subscript operator is not defined for the `multimap`.
Section 9.3

Hash Tables
Hash Tables

- The C++ standard library uses a special type of binary search tree, called a *balanced binary search tree*, to implement the `set` and `map` classes.
- This provides access to items in $O(\log n)$ time.
- Sets and maps can also be implemented using a data structure known as a *hash table*, which has some advantages over balanced search trees.
The goal of hash table is to be able to access an entry based on its key value, not its location.

We want to be able to access an entry directly through its key value, rather than by having to determine its location first by searching for the key value in an array.

Using a hash table enables us to retrieve an entry in constant time (on average, $O(1)$).
Hash Codes and Index Calculation

- The basis of hashing is to transform the item’s key value into an integer value (its hash code) which is then transformed into a table index.
Consider the Huffman code problem from the last chapter.

If a text contains only ASCII values, which are the first 128 Unicode values, we could use a table of size 128 and let its Unicode value be its location in the table.

```c
int index = ascii_char;
```
However, what if all 65,536 Unicode characters were allowed?

If you assume that on average 100 characters were used, you could use a table of 200 characters and compute the index by:

```java
int index = uni_char % 200
```
If a text contains this snippet:

. . . mañana (tomorrow), I'll finish my program. . .

Given the following Unicode values:

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Name</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0029</td>
<td>41</td>
<td>right parenthesis</td>
<td>)</td>
</tr>
<tr>
<td>0x00F1</td>
<td>241</td>
<td>small letter n with tilde</td>
<td>ň</td>
</tr>
</tbody>
</table>

The indices for letters 'ñ' and ')' are both 41

41 % 200 = 41 and 241 % 200 = 41

This is called a collision; we will discuss how to deal with collisions shortly
In most applications, a key will consist of strings of letters or digits (such as a Social Security Number, an email address, or a partial ID) rather than a single character.

The number of possible key values is much larger than the table size.

Generating good hash codes typically is an experimental process.

The goal is a random distribution of values.

Simple algorithms sometimes generate lots of collisions.
Functions for Generating Hash Codes (cont.)

- For strings, simply summing the `char` values of all characters returns the same hash code for "sign" and "sing"

- One algorithm that has shown good results uses the following formula:

\[ s_0 \times 31^{(n-1)} + s_1 \times 31^{(n-2)} + \ldots + s_{n-1} \]

where \( s_i \) is the \( i \)th character of the string, and \( n \) is the length of the string

- “Cat” has a hash code of:

\[ 'C' \times 31^2 + 'a' \times 31 + 't' = 67,510 \]

- 31 is a prime number, and prime numbers generate relatively few collisions
Because there are too many possible strings, the integer value returned by the function can't be unique.

However, the probability of two strings having the same hash code value is relatively small, because this function distributes the hash code values fairly evenly throughout the range of int values.
Because the hash codes are distributed evenly throughout the range of `int` values, this function appears to produce a random value that can be used as the table index for retrieval.

If the object is not already present in the table, the probability that the table slot with this index is empty is proportional to how full the table is.
Although the hash function result appears to be random and gives a random distribution of keys, keep in mind that the calculation is deterministic.

You always get the same hash code for a particular key.

A good hash function should be relatively simple and efficient to compute.

It doesn't make sense to use an $O(n)$ hash function to avoid doing an $O(n)$ search.
We now consider two ways to organize hash tables:

- *open addressing*
- *chaining*

In open addressing, *linear probing* can be used to access an item (type `Entry_Type*`) in a hash table:

- If the index calculated for an item's key is occupied by an item with that key, we have found the item.
- If that element contains an item with a different key, increment the index by one.
- Keep incrementing until you find the key or a NULL entry (assuming the table is not full).
Table Wraparound and Search Termination

- As you increment the table index, your table should wrap around as in a circular array.
- This enables you to search the part of the table before the hash code value in addition to the part of the table after the hash code value.
- But this could lead to an infinite loop.
- How do you know when to stop searching if the table is full and you have not found the correct value?
  - Stop when the index value for the next probe is the same as the hash code value for the object.
  - Ensure that the table is never full by increasing its size after an insertion when its load factor exceeds a specified threshold.
Hash Code Insertion Example

Tom Dick Harry Sam Pete

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

[0] [1] [2] [3] [4] [Tom]
Hash Code Insertion Example (cont.)

Dick  Harry  Sam  Pete

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>
Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

Harry Sam Pete

[0] Dick
[1] 
[2] 
[3] 
Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Dick</th>
<th>[0]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[2]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>Harry</td>
<td></td>
</tr>
</tbody>
</table>
Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496484</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

Diagram:
- Sam
- Pete
- [0] Dick
- [1] 
- [2] 
- [3] Harry
- [4] Tom
Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

```
Sam
[0] Dick
[1] 
[2] 
[3] Harry
```
Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

Diagram:
Sam
[0] Dick
[1] Sam
[2]
[3] Harry

Pete
Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

| [0]  | Dick   |
| [1]  | Sam    |
| [2]  | Harry  |
| [3]  | Tom    |
| [4]  | Pete   |
Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn() % 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

[0] Dick  
[1] Sam  
[2]   
[3] Harry  

Pete
Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

Pete

[0] Dick
[1] Sam
[2]
[3] Harry
Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

Pete
[0] Dick
[1] Sam
[2]
[3] Harry
Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>4</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>3</td>
</tr>
</tbody>
</table>

Retrieval of "Tom" or "Harry" takes one step, $O(1)$

Because of collisions, retrieval of the others requires a linear search.
## Hash Code Insertion Example (cont.)

<table>
<thead>
<tr>
<th>Name</th>
<th>hash_fcn()</th>
<th>hash_fcn()%11</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Tom&quot;</td>
<td>84274</td>
<td>3</td>
</tr>
<tr>
<td>&quot;Dick&quot;</td>
<td>2129869</td>
<td>5</td>
</tr>
<tr>
<td>&quot;Harry&quot;</td>
<td>69496448</td>
<td>10</td>
</tr>
<tr>
<td>&quot;Sam&quot;</td>
<td>82879</td>
<td>5</td>
</tr>
<tr>
<td>&quot;Pete&quot;</td>
<td>2484038</td>
<td>7</td>
</tr>
</tbody>
</table>
The best way to reduce the possibility of collision (and reduce linear search retrieval time because of collisions) is to increase the table size.
Traversing a Hash Table

You cannot traverse a hash table in a meaningful way since the sequence of stored values is arbitrary.
Deleting an Item Using Open Addressing

- When an item is deleted, you cannot simply set its table entry to null.
- If we search for an item that may have collided with the deleted item, we may conclude incorrectly that it is not in the table.
- Instead, store a dummy value or mark the location as available, but previously occupied.
- Deleted items reduce search efficiency which is partially mitigated if they are marked as available.
- You cannot replace a deleted item with a new item until you verify that the new item is not in the table.
Reducing Collisions by Expanding the Table Size

- To reduce collisions, use a prime number for the size of the table.
- A fuller table results in more collisions, so, when a hash table becomes sufficiently full, a larger table should be allocated and the entries reinserted.
- You must reinsert (rehash) values into the new table; do not copy values as some search chains which were wrapped may break.
- Deleted items are not reinserted, which saves space and reduces the length of some search chains.
Reducing Collisions by Expanding the Table Size (cont.)

- Algorithm for Rehashing
  1. Allocate a new hash table with twice the capacity of the original
  2. Reinsert each old table entry that has not been deleted into the new hash table
  3. Reference the new table instead of the original
Reducing Collisions Using Quadratic Probing

- Linear probing tends to form clusters of keys in the hash table, causing longer search chains
- **Quadratic probing** can reduce the effect of clustering
  - Increments form a quadratic series \(1 + 2^2 + 3^2 + \ldots\)

```java
probeNum++;
index = (startIndex + probeNum * probeNum) % table.length
```

- If an item has a hash code of 5, successive values of index will be 6 (5+1), 9 (5+4), 14 (5+9), . . .
The disadvantage of quadratic probing is that the next index calculation is time-consuming, involving multiplication, addition, and modulo division.

A more efficient way to calculate the next index is:

```c
k += 2;
index = (index + k) % table.size();
```
Problems with Quadratic Probing (cont.)

- Examples:
  - If the initial value of $k$ is -1, successive values of $k$ will be 1, 3, 5, ...
  - If the initial value of index is 5, successive value of index will be 6 ($= 5 + 1$), 9 ($= 5 + 1 + 3$), 14 ($= 5 + 1 + 3 + 5$), ...

- The proof of the equality of these two calculation methods is based on the mathematical series:
  
  $$n^2 = 1 + 3 + 5 + ... + 2n - 1$$
Problems with Quadratic Probing (cont.)

- A more serious problem is that not all table elements are examined when looking for an insertion index; this may mean that
  - an item can't be inserted even when the table is not full
  - the program will get stuck in an infinite loop searching for an empty slot
- If the table size is a prime number and it is never more than half full, this won't happen
- However, requiring a half empty table wastes a lot of memory
Chaining

- **Chaining** is an alternative to open addressing
- Each table element references a linked list that contains all of the items that hash to the same table index
  - The linked list often is called a *bucket*
  - The approach sometimes is called *bucket hashing*
Chaining (cont.)

- Advantages relative to open addressing:
  - Only items that have the same value for their hash codes are examined when looking for an object
  - You can store more elements in the table than the number of table slots (indices)
  - Once you determine an item is not present, you can insert it at the beginning or end of the list
  - To remove an item, you simply delete it; you do not need to replace it with a dummy item or mark it as deleted
Performance of Hash Tables

- **Load factor** is the number of filled cells divided by the table size.
- Load factor has the greatest effect on hash table performance.
- The lower the load factor, the better the performance as there is a smaller chance of collision when a table is sparsely populated.
- If there are no collisions, performance for search and retrieval is $O(1)$ regardless of table size.
Performance of Open Addressing versus Chaining

- Donald E. Knuth derived the following formula for the expected number of comparisons, $c$, required for finding an item that is in a hash table using open addressing with linear probing and a load factor $L$

\[
c = \frac{1}{2} \left( 1 + \frac{1}{1 - L} \right)
\]
Using chaining, if an item is in the table, on average we must examine the table element corresponding to the item’s hash code and then half of the items in each list

\[ c = 1 + \frac{L}{2} \]

where \( L \) is the average number of items in a list (the number of items divided by the table size)
### Performance of Open Addressing versus Chaining (cont.)

<table>
<thead>
<tr>
<th>$L$</th>
<th>Number of Probes with Linear Probing</th>
<th>Number of Probes with Chaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>0.25</td>
<td>1.17</td>
<td>1.13</td>
</tr>
<tr>
<td>0.5</td>
<td>1.50</td>
<td>1.25</td>
</tr>
<tr>
<td>0.75</td>
<td>2.50</td>
<td>1.38</td>
</tr>
<tr>
<td>0.85</td>
<td>3.83</td>
<td>1.43</td>
</tr>
<tr>
<td>0.9</td>
<td>5.50</td>
<td>1.45</td>
</tr>
<tr>
<td>0.95</td>
<td>10.50</td>
<td>1.48</td>
</tr>
</tbody>
</table>