CHAPTER 14: Hashing

Java Software Structures: Designing and Using Data Structures

Third Edition
John Lewis & Joseph Chase
Chapter Objectives

• Define hashing
• Examine various hashing functions
• Examine the problem of collisions in hash tables
• Explore the Java Collections API implementations of hashing
Hashing

• In the collections we have discussed thus far, we have made one of three assumptions regarding order:
  – Order is unimportant
  – Order is determined by the way elements are added
  – Order is determined by comparing the values of the elements
Hashing

- Hashing is the concept that order is determined by some function of the value of the element to be stored
  - More precisely, the location within the collection is determined by some function of the value of the element to be stored
Hashing

- In hashing, elements are stored in a hash table with their location in the table determined by a hashing function.
- Each location in the hash table is called a cell or a bucket.
Hashing

- Consider a simple example where we create an array that will hold 26 elements.
- Wishing to store names in our array, we create a simple hashing function that uses the first letter of each name.
A simple hashing example

FIGURE 14.1 A simple hashing example
Hashing

- Notice that using a hashing approach results in the access time to a particular element being independent of the number of elements in the table.
- This means that all of the operations on an element of a hash table should be $O(1)$.
- However, this efficiency is only realized if each element maps to a unique position in the table.
Hashing

• A \textit{collision} occurs when two or more elements map to the same location

• A hashing function that maps each element to a unique location is called \textit{perfect hashing function}

• While it is possible to develop a perfect hashing function, a function that does a good job of distributing the elements in the table will still result in \(O(1)\) operations
Another issue surrounding hashing is how large the table should be.

If we are assured of a dataset of size $n$ and a perfect hashing function, then the table should be of size $n$.

If a perfect hashing function is not available but we know the size of the dataset ($n$), then a good rule of thumb is to make the table 150% the size of the dataset.
Hashing

- The third case is the case where we do not know the size of the dataset
- In this case, we depend upon dynamic resizing
- Dynamic resizing of a hash table involves creating a new, larger, hash table and then inserting all of the elements of the old table into the new one
Hashing

- Deciding when to resize is the key to dynamic resizing
- One possibility is simply to resize when the table is full
  - However, it is the nature of hash tables that their performance seriously degrades as they become full
- A better approach is to use a load factor
- The load factor of a hash table is the percentage occupancy of the table at which the table will be resized
Hashing Functions - Extraction

- There are a wide variety of hashing functions that provide good distribution of various types of data.
- The method we used in our earlier example is called extraction.
- Extraction involves using only a part of an element’s value or key to compute the location at which to store the element.
- In our previous example, we simply extracted the first letter of a string and computed its value relative to the letter A.
Hashing Functions - Division

• Creating a hashing function by division simply means we will use the remainder of the key divided by some positive integer p
  \[ \text{Hashcode(key)} = \text{Math.abs(key)} \mod p \]
• This function will yield a result in the range 0 to p-1
• If we use our table size as p, we then have an index that maps directly to the table
Hashing Functions in the Java Language

- The java.lang.Object class defines a method called hashcode that returns an integer based on the memory location of the object.
- This is generally not very useful.
- Class that are derived from Object often override the inherited definition of hashcode to provide their own version.
- For example, the String and Integer classes define their own hashcode methods.
- These more specific hashcode functions are more effective.
Hashing - Resolving Collisions

• If we are able to develop a perfect hashing function, then we do not need to be concerned about collisions or table size
• However, often we do not know the size of the dataset and are not able to develop a perfect hashing function
• In these cases, we must choose a method for handling collisions
Resolving Collisions - Chaining

- The chaining method simply treats the hash table conceptually as a table of collections rather than a table of individual elements.
- Thus each cell is a pointer to the collection associated with that location in the table.
- Usually, this internal collection is either an unordered list or an ordered list.
- Chaining can be accomplished using a linked structure or an array based structure with an overflow area.
The chaining method of collision handling

FIGURE 14.2 The chaining method of collision handling
Chaining using an overflow area

**FIGURE 14.3** Chaining using an overflow area
Resolving Collisions - Open Addressing

• The open addressing method looks for another open position in the table other than the one to which the element is originally hashed.
• The simplest of these methods is linear probing.
• In linear probing, if an element hashes to position $p$ and that position is occupied we simply try position $(p+1) \% s$ where $s$ is the size of the table.
• One problem with linear probing is the development of clusters of occupied cells called primary clusters.
Open addressing using linear probing
Resolving Collisions - Open Addressing

• A second form of open addressing is quadratic probing

• In this method, once we have a collision, the next index to try is computed by

  \[ \text{newhashcode}(x) = \text{hashcode}(x) + (-1)^i \cdot \left(\frac{i+1}{2}\right)^2 \]

• Quadratic probing helps to eliminate the problem of primary clusters
Open addressing using quadratic probing

**Figure 14.5** Open addressing using quadratic probing
Resolving Collisions - Open Addressing

- A third form of the open addressing method is double hashing
- In this method, collisions are resolved by supplying a secondary hashing function
- For example, if a key $x$ hashes to a position $p$ that is already occupied, then the next position $p'$ will be
  \[ p' = p + \text{secondaryhashcode}(x) \]
- Given the same example, if we use a secondary hashing function that uses the length of the string, we get the following result
Open addressing using double hashing

**Figure 14.6** Open addressing using double hashing
Deleting Elements from a Hash Table

- Removing an element from a chained implementation falls into one of five cases.
- If the element we are attempting to remove is the only one mapped to the location, we simply remove the element by setting the table position to null.
Deleting Elements from a Hash Table

• If the element we are attempting to remove is stored in the table but has an index into the overflow area
  – Replace the element and the next index value in the table with the element and next index value of the array position pointed to by the element to be removed
  – We then must also set the position in the overflow area to null and add it back to our list of free overflow cells
Deleting Elements from a Hash Table

• If the element we are attempting to remove is at the end of the list of elements stored at that location the table
  – Set its position in the table to null
  – Set the next pointer of the previous element to null
  – Add that position to the list of free overflow cells
Deleting Elements from a Hash Table

• If the element we are attempting to remove is in the middle of the list of elements stored at that location
  – Set its position in the overflow area to null
  – Set the next pointer of the previous element to point to the next element of the element being removed
  – Add the position back to the list of free overflow cells

• If the element we are attempting to remove is not in the table, then we must throw an exception
Deleting Elements from a Hash Table

- If we have chosen to implement our table using open addressing, the deletion creates more of a challenge.
- This is because we cannot remove a deleted item with possibly affecting our ability to find items that collided with it on entry.
- The solution to this problem is to mark items as deleted but not actually remove them from the table until some future point when either the deleted item is overwritten or the entire table is rehashed.
Open addressing and deletion

**FIGURE 14.7** Open addressing and deletion
Hash Tables in the Java Collections API

- The Java Collections API provides seven implementations of hashing (click on each for more information):
  - Hashtable
  - HashMap
  - HashSet
  - IdentityHashMap
  - LinkedHashSet
  - LinkedHashMap
  - WeakHashMap