CSCI 240 -- Lecture Notes: Hashing

Dominick Atanasio

Introduction to Hashing

Hashing is a method of turning any type of data into a relatively small integral number that may serve as a digital "fingerprint" of the data. The hashing algorithm manipulates the data to create such fingerprints, called hash values. These hash values are usually used as indices into **hash tables**. Remember that indexing into an array only takes O(1) time (random access), thus, if we have a fast hashing algorithm to generate the index corresponding to the data (keys/values), then we could achieve constant time for the basic operations in a map.

The array itself is called a **hash table**. For example, the 911 emergency system can take your phone number, convert it to a suitable integer *i*, and store a reference to your street address in the array element *A*[*i*]. We say that the telephone number (the search key) maps or hashes to the index *i*.

In general, a **hash table** consists of two major components, a **bucket array** and a **hash function**, where a bucket array is used to store the data (key-value entries) according to their computed indices and a hash function, *h,* maps keys of a given type to integers in a fixed interval [0, *n*-1]. For example: $h(x) = x \bmod n$ is a hash function for integer keys and the integer $h(x)$ is called the hash value of key *x*.

Hash functions are designed to be fast and to yield few hash collisions in expected input domains. Collisions are inevitable, however. In hash tables, collisions inhibit the distinguishing of data, making records more costly to find. As we will see, the worst-case running time of map operations in an *n*-entry hash table is O(*n*), a hash table can usually perform these operations in O(1) expected time.

Bucket Arrays

A **bucket array** for a hash table is an array *A* of size *n*, where each cell of *A* is thought of as a "bucket" (that is, a collection of key-value pairs) and the integer *n* defines the **capacity** of the array. An entry *e* with a key *k* is simply inserted into the bucket $A[h(k)]$, where $h(x)$ is a hash function.

If each $h(k)$ returns a unique integer in the range [0, *n* - 1], then each bucket holds at most one entry. Thus, searches, insertions, and removals in the bucket array take O(1) time.

Drawbacks?

Hash Functions

A hash function maps each key to an integer in the range [0, *n* -1], where *n* is the capacity of the bucket array for the hash table. The main idea is to use the hash value, $h(k)$, as an index into our bucket array, *A*, instead of the key *k* (which is most likely inappropriate for use as a bucket array index). That is, we store the entry (*k*, *v*) in the bucket $A[h(k)]$.

A hash function is usually specified as the composition of two functions:

Hash code: h_1: keys --> integers

Compression function: h_2: integers --> [0, *n*-1]

The hash code is applied first, and the compression function is applied next on the result, i.e., $h(x) = h_2(h_1(x))$. Example: design a hash table for a map storing entries as (SSN, Name), where SSN is a 9-digit positive integer.

Potential problems?

If there are two or more keys with the same hash value, then two different entries will be mapped to the same bucket in *A*. In this case, we say a **collision** has occurred. The way to handle collisions is: (1) avoid them; and (2) handle them.

Any function can be a hash function if it produces an integer that is suitable as an array index. But not every function is a good hash function. A good hash function should:

"Distribute" the entries uniformly throughout the hash table to minimize collisions
Be fast to compute

Hash Codes

The first action that a hash function performs is to take an arbitrary key *k* and assign it an integer value, which is called the **hash code** for *k*. This integer needs not be in the range [0, *n*-1], and may even be negative. The goal is to generate a set of hash codes assigned to our keys that avoid collisions as much as possible. For if the hash codes of our keys cause collisions, then there is no hope for our compression function to avoid them. In addition, the same keys should result in the same hash code.

Memory address: We reinterpret the memory address of the key object as an integer. The generic Object class defined in Java comes with a default hashCode() method that maps each object instance to an integer that is a representation of that object. Since every class is a subclass of Object, all classes inherit this method. But unless a class overrides hashCode(), the method will return an **int** value based on the invoking object's memory address. The default hash code usually is not appropriate for hashing, because equal but distinct objects will have different hash codes.

Good in general, but works poorly with numeric numbers, character strings, etc.

In fact, the Java String class overrides the hashCode method of the Object class to be something more appropriate for character strings.

Casting to an Integer: We interpret the bits of the key as an integer. For example, for Java base types **byte**, **short**, **int**, **char**, and **float**, we can achieve a good hash code simply by casting this type into **int**. For a variable *x* of a base type **float**, we can convert *x* to an integer using a call to Float.floatToIntBits(*x*).

Suitable for keys of lengths less than or equal to the number of bits of the integer type (the above base

types). A class should define its own version of hashCode that adheres to the following guidelines:

If a class overrides the method equals, it should override hashCode.
If the method equals considers two objects equal, hashCode must return the same value for both objects. If an object invokes hashCode more than once during the execution of a program, and if the object's data remains the same during this time, hashCode must return the same hash code.
An object's hash code during one execution of a program can differ from its hash code during another execution of the same program.

Summing components: We partition the bits of the key into components of a fixed length (e.g., 16 or 32 bits) and we sum the components (ignoring overflows). For base types, such as **long** and **double**, whose bit representation is double that of a hash code, the previous approaches would not be appropriate. By casting a long to an integer, ignoring half of the information present in the original value, there will be many collisions if those numbers only differ in the bits being ignored. Instead of ignoring a part of a long search key, we can divide it into several pieces, then combine the pieces by using either **addition** or a bit-wise operation such as **exclusive or**.
An alternative hash code is to sum an integer representation of the high-order bits with an integer representation of the low-order bits.

```
int hash(long i) { return static_cast<int>(((i >> 32) + static_cast<int>(i); }
```

```
int hash(double d) {
    long long* bits{ reinterpret_cast<long long*>(&d) };
     return (int) (*bits ^ (*bits >> 32));
}
```

This approach of summing components can be further extended to any object x whose binary representation can be viewed as a k-tuple $(x_0, x_1, ..., x_{k-1})$ of integers, for we can then form a hash code by summing x_i.

Suitable for numeric keys of a fixed length greater than or equal to the number of bits of the integer type (**long** and **double**).

These computations of hash codes for the primitive types are actually used by the corresponding Java wrapper classes in their implementations of the method hashCode.

Polynomial hash codes: The summation hash code, described above, is not a good choice for character strings or other variable-length objects that can be viewed as a tuple of $(x_0, x_1, ..., x_{k-1})$, where the order of x_i's is significant. For example, the strings "stop" and "pots" collide using the above hash function. A better hash code should take into account the positions of x_i's.

We choose a nonzero constant, a != 1, and calculate $(x_0a^{k-1} + x_1a^{k-2} + ... + x_{k-2}a + x_{k-1})$ as the hash code, ignoring overflows. Mathematically speaking, this is simply a polynomial in a that takes the components $(x_0, x_1, ..., x_{k-1})$ of an object x as its coefficients. Since we are more interested in a good spread of the object x with respect to other keys, we simply ignore such overflows.

Experiments have shown that 31, 33, 37, 39, and 41 are particularly good choices for a when working with character strings that are English words. In fact, in a list of over 50,000 English words, taking a to be 33, 37, 39, or 41 produced less than 7 collisions in each case.

Many Java implementations choose the polynomial hash function, using one of these constants for a, as a default hash code for strings. For the sake of speed, however, some Java implementations only apply the polynomial hash function to a fraction of the characters in long strings.

How to evaluate the polynomial? What's the running time? -- By using the Horner's rule. Here is the code performing this evaluation for a string s and a constant a. Java's default uses a = 31.

```
int hash(std::string s)
{
     int hash {0};

     for (int i = 0; i < s.length(); ++i)
         hash = 31* hash + s[i];

    return hash;
 }
```

This computation can cause an overflow, especially for long strings. Overflows are ignored and, for an appropriate choice of a, the result will be a reasonable hash code. The current implementation of the method hashCode in Java's class String uses this computation.

Cyclic shift hash codes: A variant of the polynomial hash code replaces multiplication by a with a cyclic shift of a partial sum by a certain number of bits.

```
int hash(std::string s)
{
  int h{0};
  for (int i{0}; i < s.length(); ++i) {
    // 5-bit cyclic shift of the running sum
    h += static_cast<int>(s[i]);    // add in next char
```

```
    h = (h << 5) | (h >> 27);
  }
  return h;
}
```

Experiments have been done to calculate the number of collisions over 25,000 English words. It is shown that 5, 6, 7, 9, and 13 are good choices of shift values.

Compression Functions

The hash code for a key *k* will typically not be suitable for an immediate use with a bucket array since the hash code may be out of bounds. We still need to map the hash code into range [0, *n*-1]. The goal is to have a compression function that minimizes the possible number of collisions in a given set of hash codes.

The Division method: $\mathbf{h_{2(y)} = y\ MOD\ n}$

The size *n* of the hash table is usually chosen to be a prime number, to help "spread out" the distribution of hash values. For example, think about the hash values {200, 205, 210, 215, 220, ..., 600} with *n* = 100 or 101. The reason has to do with the number theory and is beyond the scope of this course. Choosing *n* to be a prime number is not always enough, for if there is a repeated pattern of hash codes of the form *pn* + *q* for several different *p*'s, then there will still be collisions.

The MAD method: $\mathbf{h2(y) = [(ay + b)\ mod\ p]\ mod\ n}$, where *n* is the size of the hash table, *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.

Evaluating Hashing Functions

The following five hashing functions will be considered:

1. t1: using the length of the string as its hash value
2. t2: adding the components of the string as its hash value
3. t3: hashing the first three characters of the string with polynomial hashing
4. t4: hashing the who string with polynomial hashing
5. t5: bit shifting by 5 bits

The compression function just simply uses the division method. The input file (input1.txt) is a list of 4000 random names. The input file (input2.txt) is a list of 4000 unique words from the C code. Here is the code for comparing the above 5 hashing functions (Compare.java). The following data measures the percentage of collisions.

t1	Hash Table Size			
	4000 8000 16000 100000			
input1	99.50 %	99.50 %	99.50 %	99.50 %
input2	98.75 %	98.75 %	98.75 %	98.75 %
t2	Hash Table Size			
	4000 8000 16000 100000			
input1	78.40 %	78.40 %	78.40 %	78.40 %

input2	58.46 %	58.46 %	58.46 %	58.46 %
t3	Hash Table Size			
	4000 8000 16000 100000			
input1	85.95 %	85.40 %	85.18 %	85.13 %
input2	67.91 %	64.58 %	62.56 %	60.73 %
t4	Hash Table Size			
	4000 8000 16000 100000			
input1	36.69 %	20.99 %	11.15 %	2.12%
input2	36.04 %	21.09 %	11.02 %	2.52%
t5	Hash Table Size			
	4000 8000 16000 100000			
input1	37.47 %	21.74 %	12.30 %	2.02%

input2 40.41% 24.29% 14.50% 2.42%

Resolving Collision

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, k_1 and k_2, such that $h(k_1) = h(k_2)$. When two distinct keys are mapped to the same location in the hash table, you need to find extra spots to store the values. There are two choices:

Use another location in the hash table
Change the structure of the hash table so that each array location can represent more than one value **Open Addressing**

Finding an unused, or open, location in the hash table is called **open addressing**. The process of locating an open location in the hash table is called **probing**, and various probing techniques are

available.

Linear Probing: A simple open addressing method that handles collisions by placing the colliding item in the next (circularly) available table cell. In this method, if we try to insert an entry (k, v) into a bucket $A[i]$ that is already occupied, where $i = h(k)$, then we try next at $A[(i+1) \bmod n]$. This process will continue until we find an empty bucket that can accept the new entry.

For example, we want to add the following (phone, address) entries to an addressBook with size 101:

```
addressBook.add("869-1214", "8-128");
addressBook.add("869-8131", "9-101");
addressBook.add("869-4294", "8-156");
addressBook.add("869-2072", "9-101");
```

Assume the hash function is $h(k) = (k \% 10000) \% 101$, all of the above keys (phone numbers) map to index 52. By linear probing, all entries will be put to indices 52 - 55.

With this collision resolution strategy, we also need to change the implementation of the *get*, *put*, and *remove* methods.

get(k): we must examine consecutive buckets, starting from $A[h(k)]$, until we either find an entry with its key equal to k or we find an empty bucket.
put(k, v): we must examine consecutive buckets, starting from $A[h(k)]$, until we find an empty bucket or the hashtable is full.
remove(k): There are two ways to handle this method.
The simplest way is to place null in that cell in the hash table. Problems?
The other way is to distinguish among three kinds of locations in a hash table:
Occupied: the location references an entry in the hash table
Empty: the location contains null
Available: the location's entry was removed from the hash table

Linear probing saves space, but it complicates removals. Colliding entries lump together, causing future collisions to cause a longer sequence of probes.

Let's take a look at a specific implementation of linear probing.

A potential problem with linear probing is **clustering**, where collisions that are resolved with linear probing cause groups of consecutive locations in the hash table to be occupied. Each group is called a **cluster**, and the phenomenon is known as **primary clustering**. Each cluster is a probe sequence that you must search when
adding, removing, or retrieving a table entry. When few collisions occur, probe sequence remains short and can be searched rapidly. But during an addition, a collision within a cluster increases the size of the cluster. Bigger clusters mean longer search times. As the clusters grow in size, they can merge into even larger clusters, compounding the problem. You can avoid primary clustering by changing the probe sequence.

Quadratic Probing: This open addressing strategy involves iteratively trying the buckets $A[(i + f(j)) \bmod n]$, for $j = 0, 1, 2, ...$, where $f(j) = j^2$, until finding an empty bucket. However, this approach creates its own kind of clustering, called **secondary clustering**, where the set of filled array cells "bounces" around the array in a fixed pattern. This secondary clustering is usually not a serious problem. This strategy may not find an empty slot even when the array is not full.

An advantage of linear probing is that it can reach every location in the hash table. This property is important since it guarantees the success of the *put* operation when the hash table is not full. Quadratic probing can only guarantee a successful *put* operation when the hash table is at most half full and its size is a prime number.

Double Hashing: In this approach, we choose a secondary hash function, h', and if h maps some key k to a bucket $A[i]$, with $i = h(k)$, that is already occupied, then we iteratively try the bucket $A[(i + f(j)) \bmod n]$

next, for $j = 1, 2, 3, ...,$ where $f(j) = j*h'(k)$. In this scheme, the secondary hash function is not allowed to evaluate to zero; a common choice is $h'(k) = q - (k \bmod q)$, for some prime number $q < n$. Also n should be a prime.

Double hashing uses a second hash function to compute these increments in a key-dependent way. Thus, double hashing avoids both primary and secondary clustering.

The second hash function should:

Differ from the first hash function
Depend on the search key
Have a nonzero value

Double hashing is able to reach every location in the hash table, if the size of the table is a prime

number. **Separate Chaining**

A simple and efficient way for dealing with collisions is to have each bucket $A[i]$ store a list of (k, v) pairs with $h(k) = i$. For each fundamental map operation, involving a key k, the separate-chaining approach delegates the handling of this operation to the miniature list-based map stored at $A[h(k)]$.

put(k, v): it will scan this list looking for an entry with a key equal to k; if it finds one, it replaces its value with v (replace the old value), otherwise, it puts (k, v) at the end of this list.
get(k): it searches through this list until it reaches the end or finds an entry with a key equal to k. remove(k): it performs a similar search but additionally remove an entry after it is found.

A good hash function will try to minimize collisions as much as possible, which will imply that most of our buckets are either empty or store just a single entry. Assume we use a good hash function to index the n entries of our map in a bucket array of capacity n, we expect each bucket to be of size n_b/n. This value, called the **load factor** of the hash table, should be bounded by a small constant, preferably below 1. For, given a good hash function, the expected running time of operations *get*, *put*, and *remove* in a map implemented with a hash table that uses this function is $O(n_b/n)$. Thus, we can implement these operations to run in **O(1) expected time**, provided that n is $O(n)$.

Java implementation provides a constant-time performance for the basic operations (get and put), assuming the hash function disperses the elements properly among the buckets. There are two parameters that affect the hash table's performance:

1. **Initial capacity**: the number of buckets in the hash table, and the initial capacity is simply the capacity at the time the hash table is created. The default value is 16.
2. **Load factor**: the measure of how full the hash table is allowed to get before its capacity is automatically increased. When the number of entries in the hash table exceeds the product of the load factor and the current capacity, the capacity is roughly doubled by calling the "rehash" method.

As a general rule, the default load factor (.75) offers a good tradeoff between time and space costs. Higher values decrease the space overhead but increase the lookup cost (get and put methods). The expected number of entries in the map and its load factor should be taken into account when setting its initial capacity, so as to minimize the number of rehash operations. If the initial capacity is greater than the maximum number of entries divided by the load factor, then no rehash operations will ever occur.

Separate chaining is simple but requires additional memory outside the table and an auxiliary data structure - a list - to hold entries with colliding keys.

The opening addressing schemes save some space over the separate chaining method, but they are not necessarily faster. In experimental and theoretical analyses, the chaining method is either competitive or faster than the other methods, depending on the load factor of the bucket array. So if memory space is not a major issue, the collision-handling method of choice seems to be separate chaining.

Exercises

What would be a good hash code function for a vehicle identification number, that is a string of numbers and letters of the form "9X9XX99X999", where '9' represents a digit and 'X' represents a letter? What is the worst-case time for putting n entries in an initially empty hash table, with collisions resolved by chaining? What is the best case?

Draw the 11-entry hash table that results from using the hash function, $h(i) = (2i + 5) \bmod 11$, to hash the keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, and 5, assuming collisions are handled by (1) separate chaining, (2) linear probing, (3) quadratic probing, and (4) double hashing with a secondary hash function $h'(k) = 7 - (k \bmod 7)$.