The Pigeonhole Principle and Hashing

It’s all gone to the birds, I say...

Learning Goals

After this unit, you should be able to...

- Define various forms of the pigeonhole principle; recognize and solve the specific types of counting and hashing problems to which they apply
- Provide examples of the types of problems that can benefit from a hash data structure
- Compare and contrast open addressing and chaining
- Evaluate collision resolution policies
- Describe the conditions under which hashing can degenerate from O(1) expected complexity to O(n)
- Identify the types of search problems that do not benefit from hashing (e.g. range searching) and explain why
- Manipulate data in hash structures both irrespective of implementation and also within a given implementation

Pigeonhole Principle

[Informal version] If $k+1$ pigeons fly into k pigeonholes, then some pigeonhole contains at least 2 pigeons.

But, we don't know *which* pigeonhole contains 2 or more pigeons; and, we don't know *how many* pigeons are in this pigeonhole.

[Formal version] Let X and Y be finite sets where $|X| > |Y|$. If $f{:}X \rightarrow Y$, then $f(x_1) = f(x_2)$ for some $x_1, x_2 \in X$, where $x_1 \neq x_2$.

Example 1:

(a) Suppose we have 5 colours of marbles, and that there are many marbles in a bag. How many marbles do we have to pull out of the bag if we want to be sure to get 2 marbles of the same colour?

(b) If there are 1000 marbles of each colour, how many do we need to pull to guarantee that we'll get 2 *green* marbles (assuming that green is one of the 5 colours)?

Example 2: If 5 points are placed in a 6 cm by 8 cm rectangle, argue that there are two points that are not more than 5 cm apart. (You may use the fact that 2 points in a 3 cm by 4 cm rectangle are not more than 5 cm apart.)

Hashing and Association Tables (Maps, Dictionaries)

A dictionary (or map or association table) is a collection of **key-value** pairs. In general, the objects in a table have no fixed order and each object is accessed by using the associated key.

For example, we could have a table of motor vehicle drivers, where each object is a driver's record having a name, address, number of points for bad driving, etc. Each of these objects has an associated *key*. What field below might make a suitable search key?

```
struct Driver
{
   string name;
   string address;
   int    points;
   int    licenseNumber;
};
```

Hashing

We have seen how to search an array using a linear or binary search algorithm; but generally it is faster to try to *index directly into an array*.

> **Example 1 (natural, numeric keys):** Suppose we are running a club and want to store data (e.g., telephone number, address, etc.) for each of the members.

If we assign membership numbers in the range [0]..[*N*-1] to each of the members, then we can store data for member *k* in slot *k* of an array. (Note: slot = cell = location).

Hence, the membership number (i.e., the **key**) can be used to index into the array and access the member's data record (i.e., the **value**) in O(1) time.

Example 2 (characters): Suppose we want to design a **frequency table** for counting the number of times that each character appears in a file. We can use the character's ASCII code as the **key** to index into an array of frequencies (the **values**).

In this case, note that our array of frequencies will have size 256: one for each of the 256 different ASCII characters. It may well be that not all of these 256 different characters are in a given file, and so we are potentially wasting memory. What we gain is the benefit of accessing frequencies in O(1) time. Is this a good time/space trade-off?

Suppose we make the call:

```
int freqA = getFrequency('A');
```

Since the ASCII code for 'A' has the numeric value 65, `freqA` is set to the frequency found at array index 65.

Example 3 (long numeric keys): Suppose we want to keep track of Visa credit card accounts. Visa card numbers are long: 16 digits—allowing for a total of 10^{16} different numbers.

If we were to build an array capable of holding 10^{16} accounts, we would have way more accounts than there are people on Earth! (We couldn't hold the entire table in memory, either.)

It is not reasonable to use the Visa credit card number to index into an array. We'll see how to solve this problem momentarily.

Example 4 (strings): What if the key is a *string*? For example, social insurance numbers in the United Kingdom are alphanumeric. We cannot use a non-numeric string to index into an array.

Hashing solves these problems. How? We define a function that *transforms* keys into a numeric array index. Such a function is called a **hash function**.

A hash function can be thought of as the composition of two functions:

1. Hash code map:
   $\boldsymbol{h}_1$: keys $\rightarrow$ integers
2. Compression map:
   $\boldsymbol{h}_2$: integers $\rightarrow$ $[0, \boldsymbol{N} - 1]$

The hash code map is applied first, and the compression map is then applied on the result, i.e.,

$$\boldsymbol{h}(\boldsymbol{x}) = \boldsymbol{h}_2(\boldsymbol{h}_1(\boldsymbol{x}))$$

Important: The goal of a hash function is to "disperse" the keys in an apparently random, uniform way. Why?

If the notion of uniform or random dispersion isn't present for a transformation, then rather than saying we "hash" a key, we sometimes use the term "map" or "quantize".

Strategies for Building Hash Functions

A) *Truncation*:

Use only part of the key as the hash index.

For example, if we want to build a table of student records at UBC, then rather than using the entire student number (8 digits) as the key, we could use only the first 5 digits. We would therefore require an array with space for up to 10^5 student records.

What happens when some students have the first 5 digits of their student number in common? (More about this later).

B) *Folding*:

Partition the key into parts and combine the parts using arithmetic operations.

For example, we could take a Visa number and partition it into one 6 digit number, plus two 5 digit numbers. We could then add these three numbers together to come up with a hash index.

C) *Modular Arithmetic*:

Use the modulus (remainder) operator (% in C++) to produce a *hash index* in a given range.

For example, suppose a company with 150 employees wants to use each employee's Social Insurance Number as a key. We might use an array of size 200 (leaving room for expansion). We can then generate a hash index from the Social Insurance Number as follows:

```
      hashIndex = SIN % 200;
```

Again, we would expect this hash function to lead to collisions.

What about Alphanumeric Keys?

Suppose we have a table capable of holding 5000 records, and whose keys consist of *strings* that are 6 characters long. We can apply numeric operations to the ASCII codes of the characters in the string in order to determine a hash index:

```
int hash( unsigned char * key )
{
    int hashCode = 0;
    int index = 0;

    while( key[index] != '\0' )
        hashCode += (int) key[index++];

    return hashCode % 5000;
}
```

What is a significant problem with this approach?

A solution:

```
while( key[index] != ‘\0’ )
   hashCode = 2 * hashCode + (int) key[index++];
```

Now, before the % operation, the `hashCode` is in the range 0 to 16065. After applying the modulus operator, we end up with a hash code in the range 0 to 4999.

A common, simple, and often effective mathematical expression for basic hash functions is: $h(k) = (ak + b) \bmod N$.

Hash Functions

By now, you may have realized that a hash table is *only as good as its hash function*

- If we had one bucket, we'd end up with O(n) complexity
- If we can exactly the right number of buckets, each item would have its own, unique bucket, we'd end up with O (1) complexity. Sweet!

But reality dictates that most hash tables lay somewhere in between...

The unique digital fingerprint...

The goal, then, with our hash function, is to be able to assign a unique digital fingerprint to each and every piece of data

This is of course impossible!

Consider a 32-bit hash function, which yields ~4.3 billion unique values

Even if this function were *perfect* it still would not be enough to represent the population of the planet, uniquely (which is about 6.5 billion people)

Hey wait a minute...

Hashing and the pigeonhole principle

If we have 4.3 unique locations, and we're trying to fit 6.5 billion pieces of data inside... the pigeonhole principle tells us that we are guaranteed to have **collisions**.

A rule of thumb for estimating the number of values you need to enter into a hash table before you have a 50% chance of collision (where n is the number of possible hash values that your function can map to):

sqrt(1.4 * n)

Collisions

If a hash function *h* maps two different keys *x* and *y* to the same index (i.e., $x \neq y$ and $h(x) = h(y)$), then *x* and *y* *collide*. What does this mean about *h* as a function?

A *perfect hash function* causes no collisions. Unfortunately, creating a perfect hash function requires knowledge of what keys will be hashed.

Even a hash function that distributes items randomly (why is this not a practical hash function?) will cause collisions—even when the number of items hashed is small. To see why, recall the birthday paradox from statistics:

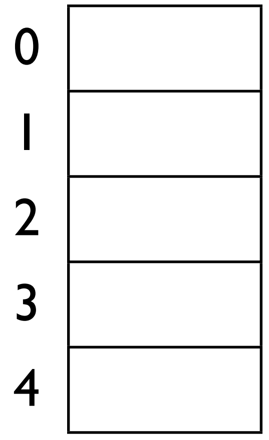
So, we must design a collision handling scheme.

General Hashing Method #1: Chaining

Each table location points to a linked list (*chain*) of items that hash to this location.

Example: Suppose $h(x) = \lfloor x/10 \rfloor \bmod 5$

Now hash: 12540, 51288, 90100, 41233, 54991, 45329, 14236



When we search for an item, we apply the hash function to get its hash index and then perform a linear search of the linked list. If the item is not in the list, then it's not in the table.

Advantage(s) of Chaining

- “Unlimited” size
- Easy to program
- Deletions are easy (this will be comparatively clear when we look at probing)

Disadvantage(s) of Chaining

- The overhead of pointers can be quite large
- As the table gets fuller, you risk becoming more inefficient. E.g. given a table of size N, the worst case searching time can be > N (how can this be?)

General Hashing Method #2: Open Addressing

There is no linked list here. We have a hash table containing key-value pairs, and we handle collisions by trying a sequence of table entries until either the item is found in the table or we reach an empty cell. The sequence is called a *probe sequence*. We have several alternatives.

(a) ***Linear probing***:

$h(k),\ h(k) + 1,\ h(k) + 2,\ h(k) + 3, \ldots, h(k) + (N - 1)$ (all mod N)

Example: Suppose $h(x) = x \bmod 10$

Hash these keys: 4, 44, 444, 6, 5.

A drawback is **clustering**. Adjacent clusters tend to join together to form composite clusters.

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

(b) ***Probing every R'th location***:

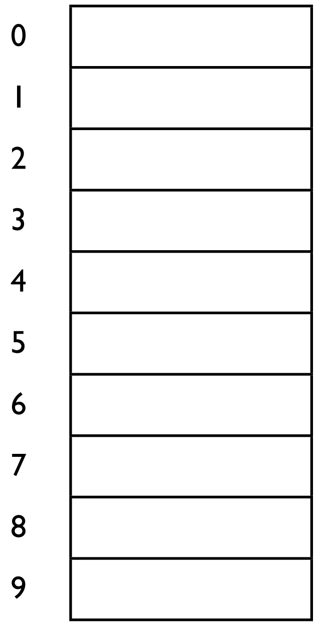
$$h(k),\ h(k)+R,\ h(k)+2R,\ \ldots,\ h(k)+(N-1)R \quad (\text{all mod } N)$$

To guarantee that we probe every table location, R must be relatively prime to N (the table size).

Example: $h(x) = x \bmod 10$, $R = 2$

Hash these keys: 4, 14, 114, 1114, 11114.

We still have clustering, but the clusters are not consecutive locations.



(c) ***Quadratic Probing***:

$$h(k), h(k) + 1^2, h(k) + 2^2, \ldots, h(k) + (N-1)^2 \qquad \text{(all mod } N\text{)}$$

This method avoids consecutive clustering.

Drawback: $i^2 = (N-i)^2 \pmod N$ so the probe sequence examines only half the table.

(Some authors use: $h(k), h(k) + 1^2, h(k) - 1^2, h(k) + 2^2, h(k) - 2^2, \ldots$)

Example: $h(x) = x \bmod 10$

Hash these keys: 4, 44, 444, 6, 5.

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

(d) ***Pseudo-Random Probing***: We first generate a sequence of random numbers

$$r_1, r_2, \ldots, r_{N-1}$$

The probe sequence is then:

$$h(k), h(k) + r_1, h(k) + r_2, \ldots, h(k) + r_{N-1} \qquad \text{(all mod } N\text{)}$$

Drawback: We must store the random numbers. Why?

Up to now, we've seen that two keys that hash to the same location have the same probe sequence. Now, here's a better solution, assuming the two keys are different…

(e) ***Double Hashing:***

$h(k), h(k) + 1h_2(k), h(k) + 2h_2(k), \ldots, h(k) + (N-1)h_2(k)$ (all mod N)

… *but* if $h_2(k) = 0$, then set $h_2(k) = 1$

A common choice is $h_2(k) = q - (k \bmod q)$ where q is a prime $< N$.

The hope is that if $h(x) = h(y)$, then $h_2(x) \neq h_2(y)$.

More hashing examples will be in your tutorials.

Disadvantages of Open Addressing

- Clusters can run into one another.
- The hash table must be at least as large as the number of items hashed, and preferably much larger.
- Deletions can be a problem. This will cause problems when searching for an existing key in a possibly long probe sequence.

Solution: *Tombstones*

Advantage of Open Addressing

- No memory is wasted on pointers

Performance of Hashing

Performance depends on:

- the quality of the hash function
- the collision resolution algorithm
- the available space in the hash table

To estimate how full a table is, we calculate the **load factor** α:

$$\alpha = (\text{ \# entries in table }) / (\text{ table size })$$

For open addressing, $0 \leq \alpha \leq 1$.

For chaining, $0 \leq \alpha < \infty$.

Donald Knuth has estimated the expected number of probes for a successful search as a function of the load factor α:

Collision Resolution Strategy	**Expected Number of Probes**
Linear probing	$\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)$
Quadratic probing, Random probing	$\frac{1}{\alpha}\ln\left(\frac{1}{1-\alpha}\right)$
Chaining	$1+\frac{\alpha}{2}$

Learning Goals

After this unit, you should be able to...

- Define various forms of the pigeonhole principle; recognize and solve the specific types of counting and hashing problems to which they apply
- Provide examples of the types of problems that can benefit from a hash data structure
- Compare and contrast open addressing and chaining
- Evaluate collision resolution policies
- Describe the conditions under which hashing can degenerate from O(1) expected complexity to O(n)
- Identify the types of search problems that do not benefit from hashing (e.g. range searching) and explain why
- Manipulate data in hash structures both irrespective of implementation and also within a given implementation