ECE 250 *Algorithms and Data Structure*
Department of Electrical and Computer Engineering
University of Waterloo

9.1 Introduction to Hash Tables

In order to discuss hash tables, we will first consider the following supporting problem.

9.1.1 A Supporting Problem

Suppose that we had approximately 150 different error conditions and each is associated with a unique 8-bit identifier on the range 0, ..., 255. Suppose that each error condition is associated with a error-handling function that will resolve the problem and that function is supposed to be called immediately upon receiving the error-condition identifier.

9.1.1.1 A Sub-optimal Solution

The way in which this can be done is to create an array of 150 function pointers and to assign them as appropriate, as is demonstrated by this code:

```
#include <iostream>

void a() {
        std::cout << "Calling 'void a()'" << std::endl;
}

void b() {
        std::cout << "Calling 'void b()'" << std::endl;
}

int main() {
        void (*function_array[150])();

        function_array[0] = a;
        function_array[1] = b;

        function_array[0]();
        function_array[1]();

        return 0;
}
```

The output of the resulting executable is

```
% ./a.out
Calling 'void a()'
Calling 'void b()'
```

Unfortunately, because the identifiers are not necessarily uniformly distributed, we would need to find out which of the 150 array entries corresponds to identifier. This means that we would have to, somehow, perform a binary search requiring approximately 6 comparisons per call. Suppose, also, that we may be modifying what functions are called or what identifiers are used for which errors. We could not use an

array for this and thus, we would have to use an AVL tree. This would extend the worst-case number of comparisons to eight. If we are dealing with an embedded system, this may already be unacceptable.

9.1.1.2 A Simple and Optimal Solution

Instead, one question we must ask ourselves is: just because the identifiers are linearly ordered, do we even care about the linear ordering? For example, will be ever ask, what is the next valid error identifier after a given identifier? The likely answer is "no". Consequently, we don't even require the linear ordering. Instead, consider the alternate solution: just create an array of size 256 function pointers and only use those entries that correspond to known error identifiers. For example, suppose 5 and 8 are error identifiers. We could therefore use something like:

```
int main() {
        void (*function_array[256])();

        function_array[5] = a;
        function_array[198] = b;

        function_array[5]();
        function_array[198]();

        return 0;
}
```

Now, a function can be called immediately without any prior searching: $\Theta(1)$

Problem: there is some wasted memory.

9.1.2 Keys and Records

Given a collection of records (structures of related data), each record is usually associated with a unique identifying key. For example, we some examples in Table 1.

Table 1. Various keys and their ranges.

Collection	Identifier	Range	Sample
Canadian Employees	Social Insurance Number	9-digit decimal	123 456 789
UW Students	UW Student ID Number	8-digit decimal	20123456
Internet Devices	IP Address	32-bit binary	000.000.000.000

Database management systems (DBMSs) will invariably assign a unique *primary key* to each entry within a table of a given database often using an automatically incremented counter. We will, however, focus on IP addresses.

9.1.3 IP Addresses

Each device connected to a network that uses the Internet Protocol Version 4 (IPv4) is assigned an IP address of 32 bits allowing over four-billion addresses. It is possible to use the IP addresses directly, for example, the ECE web server can be connected to via http://129.97.56.100/ but this is not memorable and

prone to errors. Instead, *domain names* were introduced for human use. The Domain Name System (DNS) is hierarchical, as is shown in Figure 1.

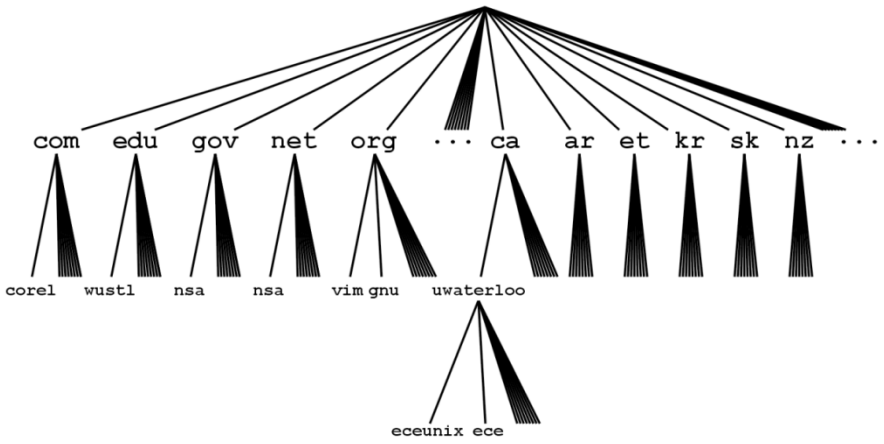

Figure 1. The DNS hierarchy.

There are a limited number of top-level domains where countries use ISO 1366 country codes. Each top-level domain is responsible for its 2nd-level domains, *etc*. You can use the Unix command host to translate between domain names and IP addresses:

```
% host uwaterloo.ca
uwaterloo.ca has address 129.97.128.40

% host ece.uwaterloo.ca
ece.uwaterloo.ca has address 129.97.56.100

% host www.uwaterloo.ca
www.uwaterloo.ca is an alias for info.uwaterloo.ca.
info.uwaterloo.ca has address 129.97.128.40

% host www.google.ca
www.google.ca is an alias for www.google.com.
www.google.com is an alias for www.l.google.com.
www.l.google.com has address 72.14.205.99
www.l.google.com has address 72.14.205.103
www.l.google.com has address 72.14.205.104
www.l.google.com has address 72.14.205.147
```

As you may note, the mapping is not one-to-one:

1. Some IP addresses may be associated with multiple domain names (*e.g.*, both www.uwaterloo.ca and info.uwaterloo.ca map to `129.97.128.40`), and
2. Some domain names may be associated with multiple IP addresses (*e.g.*, www.google.ca maps to `72.14.205.99`, `.103`, `.104`, and `.147`).

DNS allows a division of effort in name translation: a server in Korea (kr) does not need to know about the sub-domains of, for example, uwaterloo.ca and, at the same time, the University of Waterloo has

complete control over any IP addresses starting with `129.97`; *i.e.*, UW has access to $256^2 = 65\,535$ IP addresses. Therefore, if we wanted to associate UW IP addresses with their corresponding domain names, using the idea suggested before, we could create an array of 65 535 strings and index each IP address to the last 16 bits. For example, the last 16 bits of `129.97.90.209` are $0101101011010001_2 = 23249$. We could look up a table such as Table 1.

Table 2. IP addresses and their associated domain names.

Index	Address	Domain Name
23240	`129.97.90.200`	sidicsem.uwaterloo.ca
23241	`129.97.90.201`	watdist8.uwaterloo.ca
23242	`129.97.90.202`	NO DOMAIN NAME
23243	`129.97.90.203`	secure0.uwaterloo.ca
23244	`129.97.90.204`	msma.uwaterloo.ca
23245	`129.97.90.205`	ehab0.uwaterloo.ca
23246	`129.97.90.206`	calliope1.uwaterloo.ca
23247	`129.97.90.207`	calliope2.uwaterloo.ca
23248	`129.97.90.208`	dsip-lpt.uwaterloo.ca
23249	`129.97.90.209`	churchill.uwaterloo.ca

This would be a reasonably dense array: UW uses over two-thirds of the domain names assigned to it. Again, given an IP address, the translation to the domain name is $\Theta(1)$.

There are other, significantly larger problems:

1. What if the array size is very large relative to the number of entries?
2. Given a domain name, how do we access its IP address?
3. How does a router map an IP address onto the most appropriate route?

9.1.3.1 Large Spaces

Currently, UW uses two thirds of its allocated IP addresses. The standard IPv6 uses 128-bit addresses allowing 340 undecillion unique address (that is approximately 1.1 IP addresses for every cubic metre inside a sphere the radius of which equals the radius of Neptune. Alternatively, that's 3.4 octillion IP addresses for every star in the Milky Way galaxy. Suppose UW was assigned 2^{32} of these addresses. In this case, we could no longer store our 45 000 IP addresses in an array as suggested before. We could sort these addresses by their values, but this would require, again, a binary search: each lookup would require lg(45 000) or approximately 16 comparisons. Can we somehow get back to $\Theta(1)$ lookups?

A Simpler Example

Suppose I have 100 students in my class. Suppose I want to store grades so that I can quickly look them up. Now, each student is assign a unique UW Student ID Number, but creating an array of 10^8 when I only have 100 students in my class would be a waste of memory—one million empty addresses per student.

Instead, create an array of size 1000 and map (or *hash*) each UW Student ID Number to the last three digits. Thus, the student 20123456 would have his or her grade stored in array entry 456.

Problem: multiple students will have the same last three digits. In fact, what is the probability that out of 100 students, none of them will have the same last digits?

$$\prod_{k=1}^{100}\left(1-\frac{k-1}{1000}\right)\approx 0.005959\approx 0.60\,\%\,.$$

This is related to the birthday question: in a group of 23 students, what is the probability that no two students will have the same birthday?

$$\prod_{k=1}^{23}\left(1-\frac{k-1}{366}\right)\approx 0.4937\approx 49\,\%\,.$$

Therefore, it is almost certain that we will have to deal with two students having the last three digits. If two students have the same last three digits, we will call it a *collision*. Even if we took the last four digits of 100 students, there would still be only a 60 % chance that each student will have a unique last four digits. Thus, collisions are more-or-less inevitable.

9.1.3.2 Mapping Domain Names to IP Addresses

Suppose we want to take a domain name and map it to an IP address. We do not want to keep a lexicographically sorted list of domain names and then perform a search. Looking at the previous idea, we took an 8-digit number and mapped it to its last three digits. Is there a way that we could take a domain name and map it onto a number, say, on the range 0 through 131 071 (= $2^{17} - 1$)? If this were possible

9.1.3.3 Mapping IP Addresses to Routes

The Internet *backbone* is connected via *core routers*. These routers must accept communications at rates of 10 Gbit/s and it must read, interpret, and forward each packet to the most appropriate neighbouring router getting that packet to its destination in the shortest number of jumps. These routers must quickly look up IP addresses as well as dealing with problems such as other core routers being taken down for service.

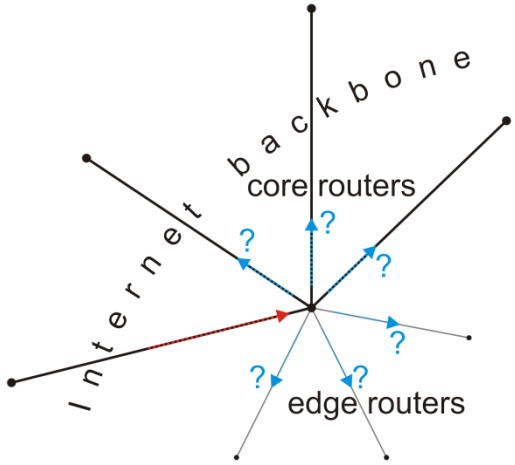

Figure 2. Core routers forwarding incoming packets.

9.1.4 Strategy

We will use the idea suggested by UW Student ID Numbers:

1. We will take an object (be it a UW Student ID Number, an IP address or a string) and convert it to a 32-bit integer (the techniques vary according to the object),
2. we will then map that 32-bit integer onto a range 0, ..., $M - 1$ (using modulus or the mid-square, multiplicative, or Fibonacci techniques), and
3. Deal with collisions with
    a. Chained hash tables, or
    b. Open addressing (linear and double probing).

You can see that this is three independent steps:

$$\text{Object} \rightarrow \text{32-bit integer} \rightarrow \text{Map down to } 0, ..., M - 1 \rightarrow \text{Deal with collisions}$$