CS121 Data Structures A, C
Hash Tables

Varduhi Yeghiazaryan
vyeghiazaryan@aua.am

Fall 2021

Maps

A **map** is an abstract data type designed to efficiently store and retrieve values based on a uniquely identifying **search key** for each

A map stores key-value pairs (k, v), called **entries**, where k is the key and v is its corresponding value

Keys are required to be **unique**, i.e. multiple entries with the same key are **not** allowed

For example, the mapping from URLs (key) to page content (value)

The main operations of a map are for searching, inserting, and deleting items

Maps are also known as **associative arrays**, because the entry's key serves somewhat like an index into the map, in that it assists the map in efficiently locating the associated entry

The Map ADT

A map M is a collection of key-value pairs and supports:

size(): Returns the number of entries in M

isEmpty(): Returns a boolean indicating whether M is empty

get(k): Returns the value v associated with key k, if such an entry exists; otherwise returns `null`

put(k, v): If M does not have an entry with key equal to k, then adds entry (k, v) to M and returns `null`; else, replaces with v the existing value of the entry with key equal to k and returns the old value

remove(k): Removes from M the entry with key equal to k, and returns its value; if M has no such entry, then returns `null`

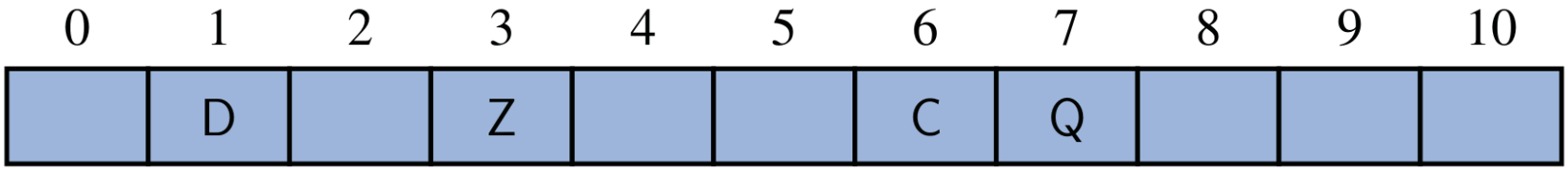
keySet(): Returns an iterable collection containing all the keys stored in M

values(): Returns an iterable collection containing all the *values* of entries stored in M (with repetition if multiple keys map to the same value)

entrySet(): Returns an iterable collection containing all the key-value entries in M

Lookup Tables

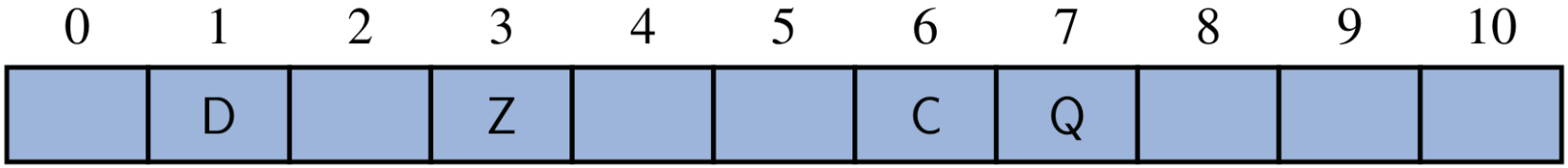
If the map uses integer keys in the range from 0 to $N - 1$ for some $N \geq n$, then we can represent the map using a **lookup table** of length N



This is a lookup table of length ?? for a map containing entries (?, ?), (?, ?), (?, ?), and (?, ?)

Lookup Tables

If the map uses integer keys in the range from 0 to $N - 1$ for some $N \geq n$, then we can represent the map using a **lookup table** of length N



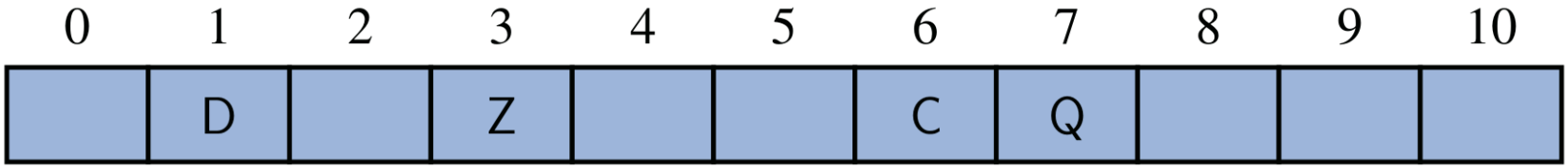
This is a lookup table of length 11 for a map containing entries (1, D), (3, Z), (6, C), and (7, Q)

We store the value associated with key k at index k of the table (presuming that we have a distinct way to represent an empty slot)

Worst-case running times of operations `get`, `put`, and `remove`:

Lookup Tables

If the map uses integer keys in the range from 0 to $N - 1$ for some $N \geq n$, then we can represent the map using a **lookup table** of length N



This is a lookup table of length 11 for a map containing entries (1, D), (3, Z), (6, C), and (7, Q)

We store the value associated with key k at index k of the table (presuming that we have a distinct way to represent an empty slot)

Worst-case running times of operations `get`, `put`, and `remove`: $O(1)$

Hash Tables

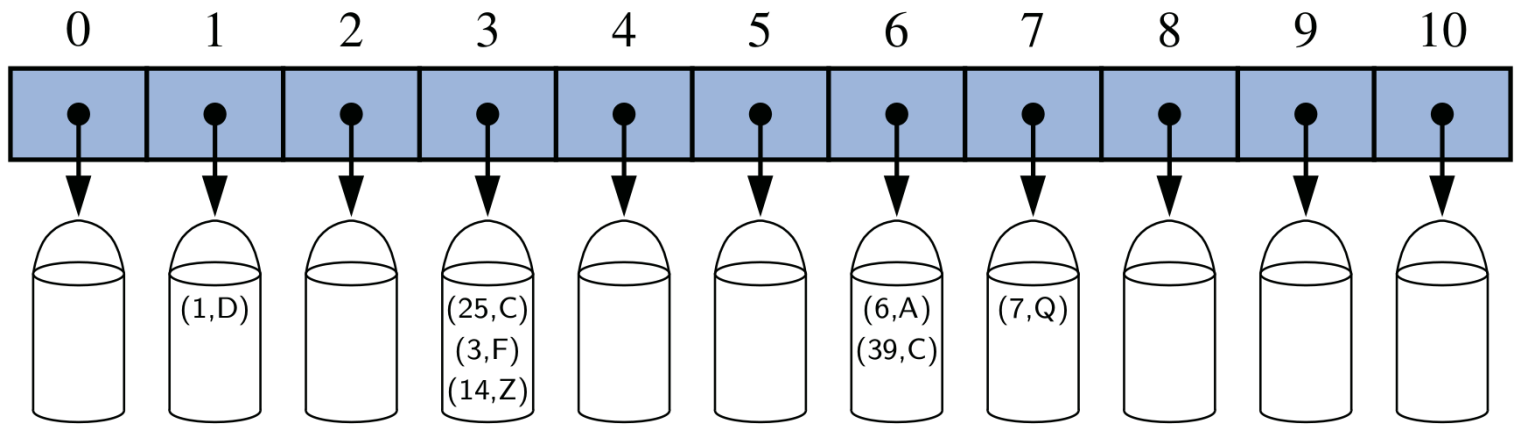
Problems when generalizing to all maps:

- N may be too large compared to n, i.e. $N \gg n$: **using too much memory!**
- the map's keys are **not necessarily integers**

Idea: use a **hash function** to map general keys to corresponding indices in a table

Note that there may be two or more distinct keys that get mapped to the same index (*Is this good?*)

Hash tables are one of the most efficient data structures for a map

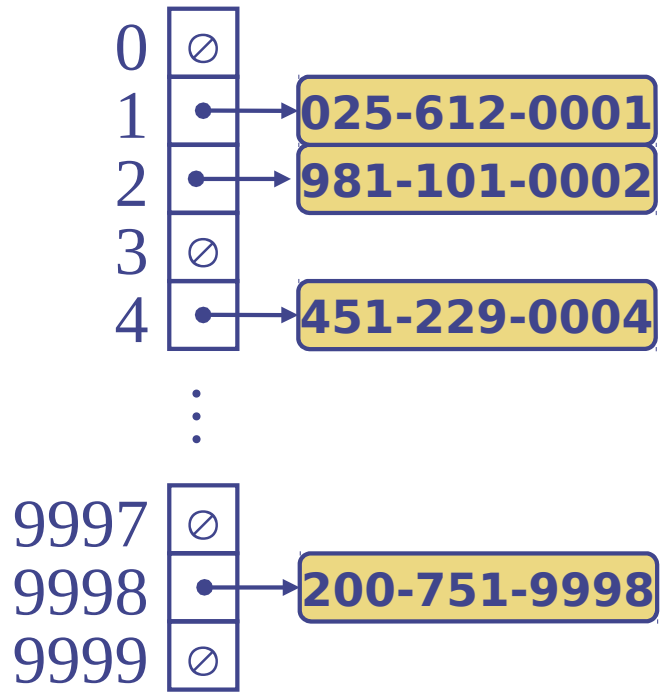


Can you guess the hash function here?

Example

We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a ten-digit positive integer

Our hash table uses an array of size $N = 10,000$ and the hash function $h(x) =$ last four digits of x



Hash Functions

The goal of a **hash function** h is to map each key k to an integer in the range $[0, N-1]$

N is the capacity of the bucket array A for a hash table

The hash function value $h(k)$ is the index into our bucket array A

The entry (k, v) is stored in the bucket $A[h(k)]$

If two or more keys have the same hash value and, hence, two or more different entries are mapped to the same bucket, a **collision** has occurred

We try to avoid collisions!

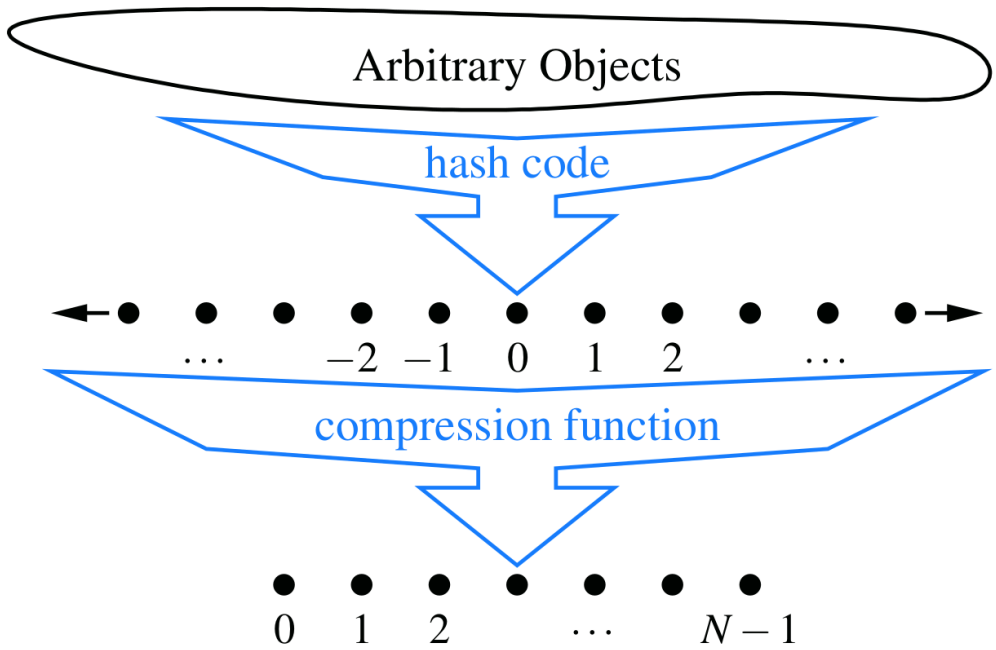
A hash function is 'good' if:

- it minimizes collisions when mapping keys
- it is fast and easy to compute

Calculating Hash Functions

The calculation of $h(k)$ consists of two parts:

1. a **hash code** maps a key k to an integer
2. a **compression function** maps the hash code to an integer within a range of indices $[0, N-1]$ for a bucket array

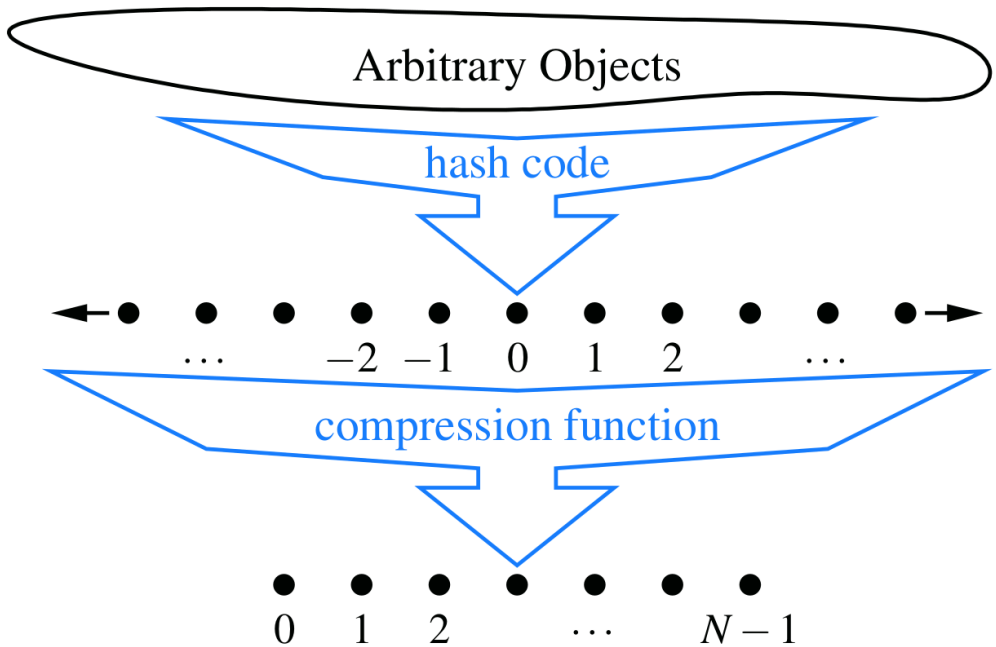


Why do we split it into two steps?

Calculating Hash Functions

The calculation of $h(k)$ consists of two parts:

1. a **hash code** maps a key k to an integer
2. a **compression function** maps the hash code to an integer within a range of indices $[0, N-1]$ for a bucket array



Why do we split it into two steps? Hash code computation is independent of a specific hash table size

Hash Codes: Integer Cast

In the first step of a hash function, the integer **hash code** for an arbitrary key k is computed

- We reinterpret the bits of the key as an integer
- Suitable for keys of length less than or equal to the number of bits of the integer type (e.g. `char`, `short`, `int` and `float`)

How can we reinterpret 64-bit keys?

Solution: use only the high-order 32 bits (or the low-order 32 bits)

Does this result in a good hash function?

Hash Codes: Integer Cast

In the first step of a hash function, the integer **hash code** for an arbitrary key k is computed

- We reinterpret the bits of the key as an integer
- Suitable for keys of length less than or equal to the number of bits of the integer type (e.g. `char`, `short`, `int` and `float`)

How can we reinterpret 64-bit keys?

Solution: use only the high-order 32 bits (or the low-order 32 bits)

Does this result in a good hash function?

A better approach combining the high-order and low-order portions: add the two components as 32-bit numbers, or take the exclusive-or of the two components

General case: $\sum_{i=0}^{n-1} x_i$ or $x_0 \hat{} x_1 \hat{} \cdots \hat{} x_{n-1}$ for an n-tuple $(x_0, x_1, \ldots, x_{n-1})$

Polynomial Hash Codes

Let's apply a 16-bit hash code for a character string s that sums the Unicode values of the characters in s

It produces lots of unwanted collisions: "temp01" and "temp10"; "stop", "tops", "pots", and "spot"

We should take into account the positions of the x_i's in $(x_0, x_1, \ldots, x_{n-1})$, e.g. **polynomial hash code**:

$$x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1}$$

for some nonzero constant $a \neq 1$

How can we compute this polynomial?

Polynomial Hash Codes

Let's apply a 16-bit hash code for a character string s that sums the Unicode values of the characters in s

It produces lots of unwanted collisions: "temp01" and "temp10"; "stop", "tops", "pots", and "spot"

We should take into account the positions of the x_i's in $(x_0, x_1, \ldots, x_{n-1})$, e.g. **polynomial hash code**:

$$x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1}$$

for some nonzero constant $a \neq 1$

How can we compute this polynomial? By Horner's rule:

Polynomial Hash Codes

Let's apply a 16-bit hash code for a character string s that sums the Unicode values of the characters in s

It produces lots of unwanted collisions: "temp01" and "temp10"; "stop", "tops", "pots", and "spot"

We should take into account the positions of the x_i's in $(x_0, x_1, \ldots, x_{n-1})$, e.g. **polynomial hash code**:

$$x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1}$$

for some nonzero constant $a \neq 1$

How can we compute this polynomial? By Horner's rule:

$$x_{n-1} + a(x_{n-2} + a(x_{n-3} + \cdots + a(x_2 + a(x_1 + ax_0)) \cdots))$$

Polynomial Hash Codes (cont'd)

On a typical computer, evaluating a polynomial will be done using the finite bit representation for a hash code

How should we handle overflows?

Polynomial Hash Codes (cont'd)

On a typical computer, evaluating a polynomial will be done using the finite bit representation for a hash code

How should we handle overflows? Since we are more interested in a good spread of the object x with respect to other keys, we simply ignore such overflows

Still, we should choose the constant a so that it has some nonzero, low-order bits, which will serve to preserve some of the information content even as we are in an overflow situation

Experiments suggest that 33, 37, 39, and 41 are good choices for a when working with character strings that are English words

Each of them produces fewer than 7 collisions in a list of over 50,000 English words

Hash Codes in Java

The `Object` class includes a default `hashCode()` method that returns a 32-bit `int`

The default version is just an integer representation derived from the object's memory address

We want equivalent keys in a map to have the same hash code to guarantee the same bucket for them

If `x.equals(y)` then `x.hashCode() == y.hashCode()`

Example `hashCode` implementation for the `SinglyLinkedList`

```
1  public int hashCode() {
2    int h = 0;
3    for (Node walk=head; walk != null; walk = walk.getNext()) {
4      h *= 33;              // product of constant and composite code
5      h += walk.getElement().hashCode(); // add element's code
6    }
7    return h;
8  }
```

Compression Functions: The Division Method

The integer hash code may be negative or may exceed the capacity of the bucket array

There is the issue of **compression function**—the mapping of integer hash code into the range $[0, N-1]$

The **division method** maps an integer i to $i \bmod N$

If N is prime, the risk of collisions is smaller

E.g. each hash code in $\{205, 210, 215, 220, \ldots, 600\}$ collides with three others for $N = 100$

If $N = 101$ there will be no collisions

If a hash function is chosen well, it should ensure that the probability of two different keys getting hashed to the same bucket is $1/N$

Compression Functions: The MAD Method

If the hash codes are of the form $pN + q$ for several different p's then choosing a prime N is not enough

The **Multiply-Add-and-Divide** ('MAD') method maps an integer i to $[(ai + b) \mod p] \mod N$

N is the size of the bucket array

p is a prime number larger than N

$a > 0$ and b are integers chosen at random from the interval $[0, p - 1]$

Collision-Handling

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 $A[h(k)]$

We may have distinct keys k_1 and k_2 s.t. $h(k_1) = h(k_2)$

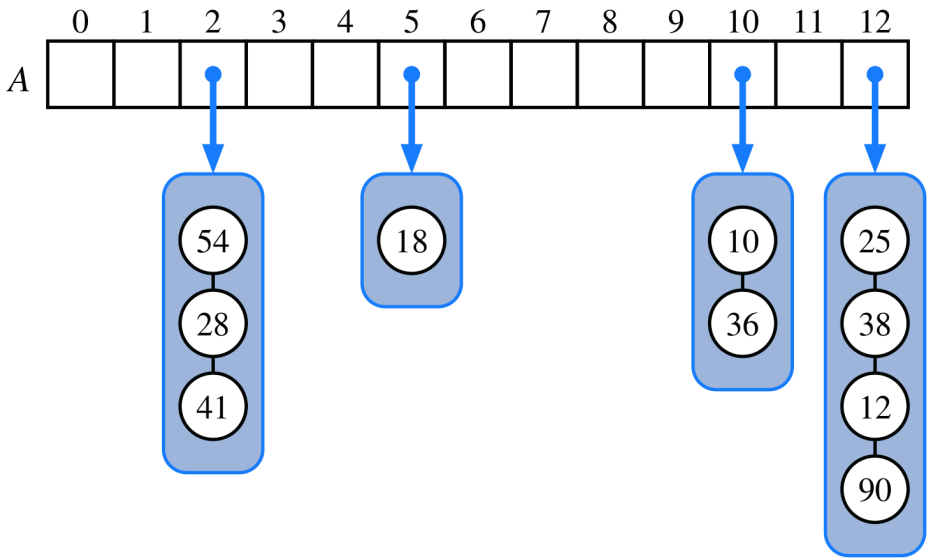
These collisions complicate our procedure for performing insertion, search and deletion operations

Collisions can be dealt with in a few ways considered next

Separate Chaining

In **separate chaining** each bucket $A[j]$ stores its own secondary container, holding all entries (k, v) s.t. $h(k) = j$

The secondary container is usually a small map instance implemented using an unordered list



Given a good hash function, the core map operations run in $O(\lceil n/N \rceil)$ since the expected size of a bucket is n/N

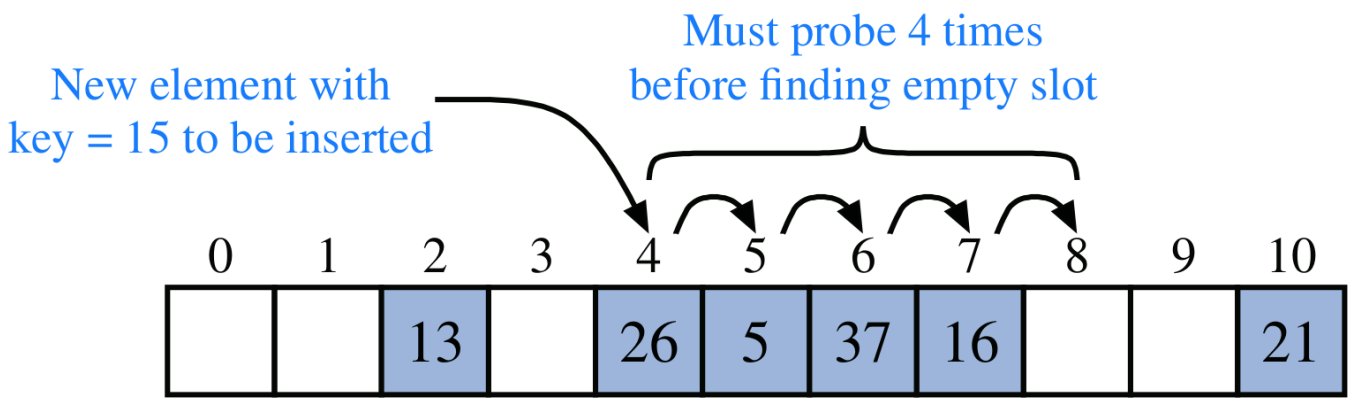
The ratio $\lambda = n/N$ is called the **load factor** of the hash table. If λ is $O(1)$ the core operations run in $O(1)$ *expected* time

Linear Probing

In **open addressing** each entry is stored directly in the cells of the bucket array (no auxiliary data structure to hold entries with colliding keys) and the load factor is always at most 1

In **linear probing**—a variant of open addressing—if we try to insert an entry (k, v) into a bucket $A[j]$ that is already occupied, $j = h(k)$:

- we next try $A[(j+1) \mod N]$, $A[(j+2) \mod N]$ and so on
- once we find an empty bucket that can accept the new entry, we simply insert the entry there



Quadratic Probing and Double Hashing

Quadratic probing (another open addressing strategy) iteratively tries the buckets $A[(h(k) + i^2) \mod N]$ for $i = 0, 1, 2, \ldots$ until finding an empty bucket

In **double hashing** (yet another open addressing strategy) we choose a secondary hash function h' and if the bucket $A[h(k)]$ is already occupied, then we iteratively try the buckets $A[(h(k) + i \cdot h'(k)) \mod N]$ for $i = 1, 2, 3, \ldots$

The secondary hash function is not allowed to evaluate to zero

A common choice: $h'(k) = q - (k \mod q)$ for some prime $q < N$ where N is also prime

Load Factors and Rehashing

In the hash table schemes described thus far, it is important that the load factor $\lambda = n/N$ be kept below 1

Experiments suggest the following bounds:

Separate chaining: maintain $\lambda < 0.9$

Open addressing with linear probing: maintain $\lambda < 0.5$ (only a bit higher for other open addressing schemes)

If an insertion causes the load factor of a hash table to go above the specified threshold, then we resize the table (to regain the specified load factor) and reinsert all objects into this new table

We need to reapply a new compression function that takes into consideration the size of the new table

It is a good requirement for the new array's size to be a prime number approximately double the previous size (*why?*)

Efficiency of Hash Tables

If our hash function is good, we expect the entries to be uniformly distributed in the N cells of the bucket array

Thus, to store n entries, the expected number of keys in a bucket would be $\lceil n/N \rceil$, which is $O(1)$ if n is $O(N)$

Method	**Unsorted List**	**Hash Table**	
		expected	worst case
get	$O(n)$	$O(1)$	$O(n)$
put	$O(n)$	$O(1)$	$O(n)$
remove	$O(n)$	$O(1)$	$O(n)$
size, isEmpty	$O(1)$	$O(1)$	$O(1)$
entrySet, keySet, values	$O(n)$	$O(n)$	$O(n)$

If the capacity N is proportional to the number of entries n

Java Hash Table Implementation

We develop two implementations of a hash table:

- using separate chaining
- using open addressing with linear probing

While these approaches to collision resolution are quite different, there are many higher-level commonalities to the two hashing algorithms

For that reason, we extend the `AbstractMap` class to define a new `AbstractHashMap` class for the common functionality

Summary

Reading

Section 10.2 Hash Tables

Questions?