Algorithms and Data Structures 2

17 – Hash tables

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

Outline

- **Map ADT**
- **Implementations**
- **Direct-address tables**
- **Hash table data structure**
- **Hash functions**

Map ADT

- **A map models a searchable collection of key/value pairs**
  - Other names: associative array, symbol table, dictionary
  - Multiple entries with the same key are not allowed (keys must be unique)

- **Main map operations**
  - INSERT(M,k,v): add a pair (k,v) to map M
  - DELETE(M,k): remove key k and its value from map M
  - SEARCH(M,k): find a pair with key k in map M

- **Auxiliary map operation**
  - MAP-EMPTY(M): test whether no key/value pairs are stored in map M

Example

- **Map an integer k to a character v**
  - ASCII code

M = {}

  - INSERT(M, 65, ‘A’)
  - INSERT(M, 71, ‘G’)
  - INSERT(M, 113, ‘q’)
  - INSERT(M, 109, ‘m’)
  - SEARCH(M, 65)
  - INSERT(M, 83, ‘S’)
  - DELETE(M, 113)
  - SEARCH(M, 113)

Example

- **Map an integer k to a character v**
  - ASCII code

M = {}

  - INSERT(M, 65, ‘A’)

M = {(65, ‘A’)}

  - INSERT(M, 71, ‘G’)
  - INSERT(M, 113, ‘q’)
  - INSERT(M, 109, ‘m’)
  - SEARCH(M, 65)
  - INSERT(M, 83, ‘S’)
  - DELETE(M, 113)
  - SEARCH(M, 113)

Example

- **Map an integer k to a character v**
  - ASCII code

    M = {}

  - INSERT(M, 65, ‘A’) M = {(65, ‘A’)}
  - INSERT(M, 71, ‘G’) M = {(65, ‘A’), (71, ‘G’)}
  - INSERT(M, 113, ‘q’)
  - INSERT(M, 109, ‘m’)
  - SEARCH(M, 65)
  - INSERT(M, 83, ‘S’)
  - DELETE(M, 113)
  - SEARCH(M, 113)

Example

- **Map an integer k to a character v**
  - ASCII code

    M = {}

  - INSERT(M, 65, ‘A’) M = {(65, ‘A’)}
  - INSERT(M, 71, ‘G’) M = {(65, ‘A’), (71, ‘G’)}
  - INSERT(M, 113, ‘q’) M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}
  - INSERT(M, 109, ‘m’)
  - SEARCH(M, 65)
  - INSERT(M, 83, ‘S’)
  - DELETE(M, 113)
  - SEARCH(M, 113)

Example

- **Map an integer k to a character v**
  - ASCII code

    M = {}

  - INSERT(M, 65, ‘A’) — M = {(65, ‘A’)}
  - INSERT(M, 71, ‘G’) — M = {(65, ‘A’), (71, ‘G’)}
  - INSERT(M, 113, ‘q’) — M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}
  - INSERT(M, 109, ‘m’) — M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’)}
  - SEARCH(M, 65)
  - INSERT(M, 83, ‘S’)
  - DELETE(M, 113)
  - SEARCH(M, 113)

Example

- **Map an integer k to a character v**
  - ASCII code

	M = {}
– INSERT(M, 65, ‘A’)	M = {(65, ‘A’)}
– INSERT(M, 71, ‘G’)	M = {(65, ‘A’), (71, ‘G’)}
– INSERT(M, 113, ‘q’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}
– INSERT(M, 109, ‘m’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’)}
– SEARCH(M, 65)	return (65, ‘A’)
– INSERT(M, 83, ‘S’)	
– DELETE(M, 113)	
– SEARCH(M, 113)	

Example

- **Map an integer k to a character v**
  - ASCII code

	M = {}
– INSERT(M, 65, ‘A’)	M = {(65, ‘A’)}
– INSERT(M, 71, ‘G’)	M = {(65, ‘A’), (71, ‘G’)}
– INSERT(M, 113, ‘q’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}
– INSERT(M, 109, ‘m’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’)}
– SEARCH(M, 65)	return (65, ‘A’)
– INSERT(M, 83, ‘S’)	M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’), (83, ‘S’)}
– DELETE(M, 113)	
– SEARCH(M, 113)	

Example

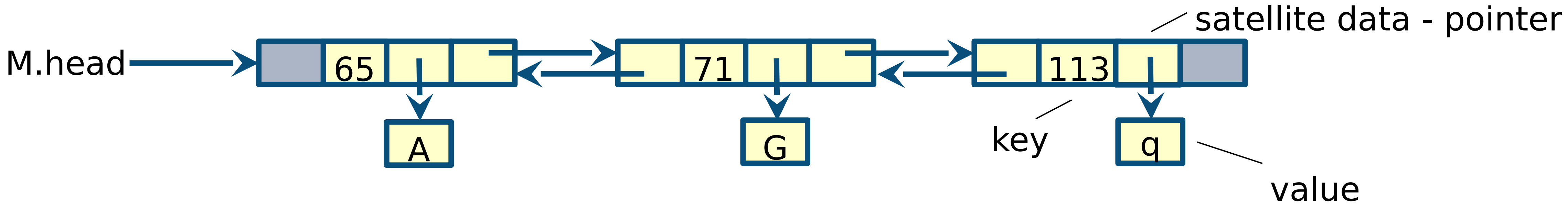
- **Map an integer k to a character v**
  - ASCII code — M = {}
  - INSERT(M, 65, ‘A’) — M = {(65, ‘A’)}
  - INSERT(M, 71, ‘G’) — M = {(65, ‘A’), (71, ‘G’)}
  - INSERT(M, 113, ‘q’) — M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}
  - INSERT(M, 109, ‘m’) — M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’)}
  - SEARCH(M, 65) — return (65, ‘A’)
  - INSERT(M, 83, ‘S’) — M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’), (83, ‘S’)}
  - DELETE(M, 113) — M = {(65, ‘A’), (71, ‘G’), (109, ‘m’), (83, ‘S’)}
  - SEARCH(M, 113)

Example

- **Map an integer k to a character v**
  - ASCII code
    M = {}
  - INSERT(M, 65, ‘A’) M = {(65, ‘A’)}
  - INSERT(M, 71, ‘G’) M = {(65, ‘A’), (71, ‘G’)}
  - INSERT(M, 113, ‘q’) M = {(65, ‘A’), (71, ‘G’), (113, ‘q’)}
  - INSERT(M, 109, ‘m’) M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’)}
  - SEARCH(M, 65) return (65, ‘A’)
  - INSERT(M, 83, ‘S’) M = {(65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’), (83, ‘S’)}
  - DELETE(M, 113) M = {(65, ‘A’), (71, ‘G’), (109, ‘m’), (83, ‘S’)}
  - SEARCH(M, 113) return NIL

List-based implementation

- **We can implement a map M using an unsorted doubly-linked list**
  - Values are stored as satellite data (attribute if small, pointer for larger structures)

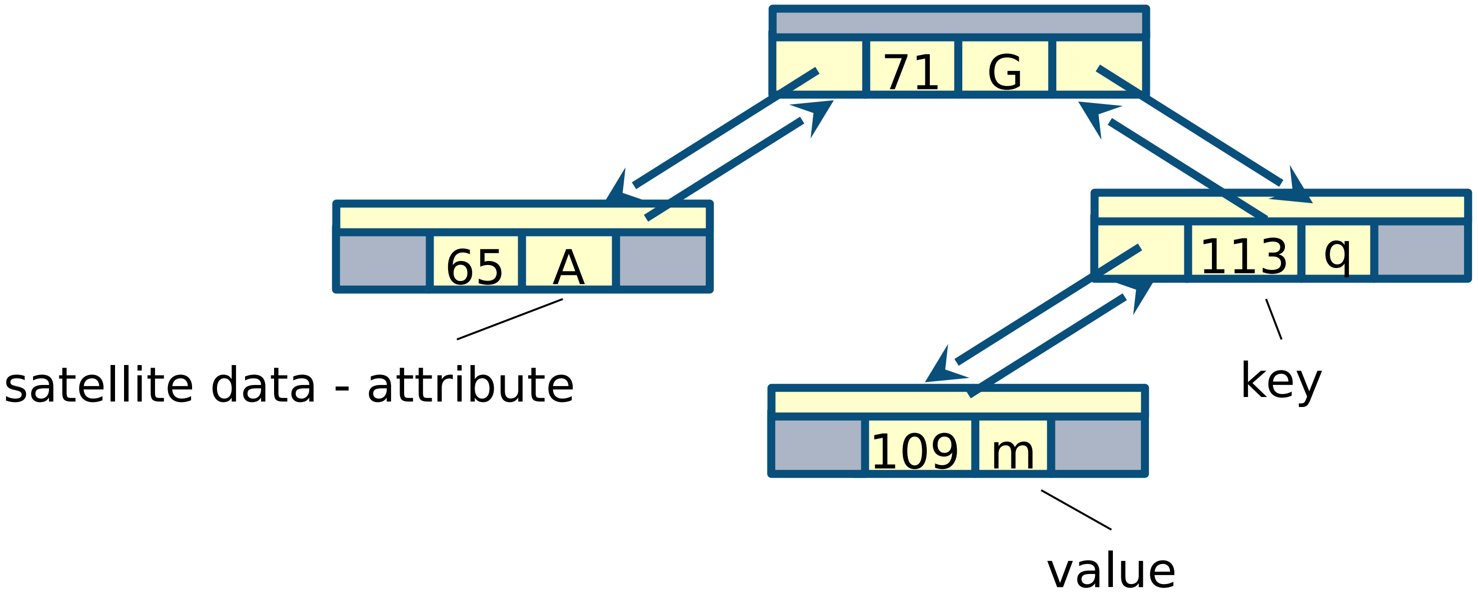


- **Performance**
  - INSERT takes $O(1)$ time ($O(n)$ if we first check for duplicates)
  - SEARCH and DELETE take $O(n)$ time since in the worst case (the item is not found) we traverse the entire list to look for an item with the given key

- **The list-based implementation is effective only for maps of small size**

Tree-based implementation

- **Self-balancing trees guarantee a worst-case time complexity of O(log n) for all the main operation of the Map ADT**
  - Inorder traversal allows us to get a sorted sequence of all the pairs stored in the map

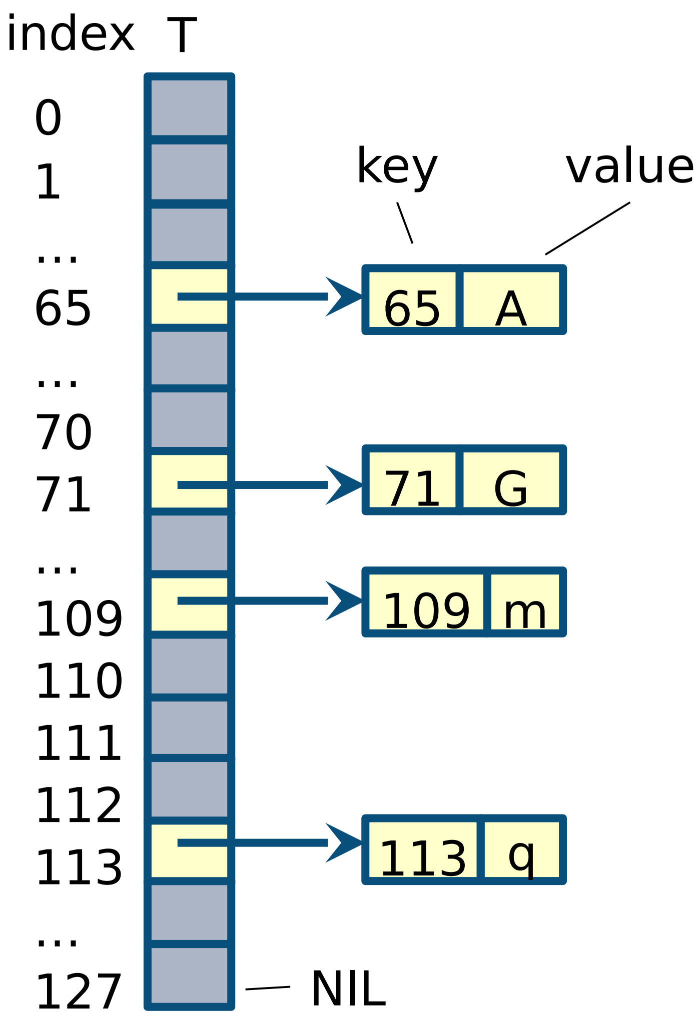


- **Can we do better?**

Direct-address tables

- **Assumptions**
  - Each element of the map has an integer key drawn from the universe U = {0,1,..., m – 1}, where m is not too large
  - No two elements have the same key

- **We use an array or direct-address table T[0,..,m - 1] to represent the map**
  - Each position (also called slot or bucket) in T corresponds to a key in the universe U
  - Slot k points to to an element in the map with key k
  - If no element has key k, then T[k] = NIL



Direct-address tables (cont.)

- **Operations are trivial to implement**
  - Each operation takes O(1) time
- **Space requirement is the size of the universe of keys U = {0,1,…, m – 1}**
  - Impractical unless m is small
- **In some applications key/value pairs are stored directly in T**
  - To further save space, the key is not stored as knowing the index is enough to determine the key of an object

```
DIRECT-ADDRESS-INSERT(T,x)
  T[x.key] := x
```

```
DIRECT-ADDRESS-SEARCH(T,k)
  return T[k]
```

```
DIRECT-ADDRESS-DELETE(T,x)
  T[x.key] := NIL
```

Hash tables

- **Data structure invented by H. P. Luhn in 1953 for the storage of key/value pairs**
  - Generalisation of direct-address tables

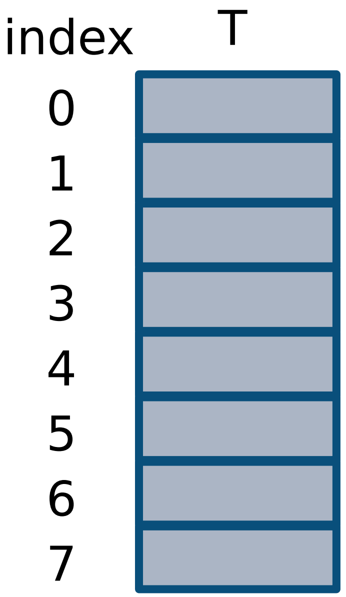
- **It consists of two main components**
  1. An array T[0,..,m - 1] of fixed size (called hash table or bucket array)
  2. A hash function h: U → {0,1,…,m - 1} mapping keys to slots in T, where m << |U|

- **When two keys are mapped to the same position in array T, we have a hash collision**
  - Ideally hash function is easy to compute and no collisions occur

- **Hash tables support INSERT, DELETE and SEARCH operations in O(1) time**

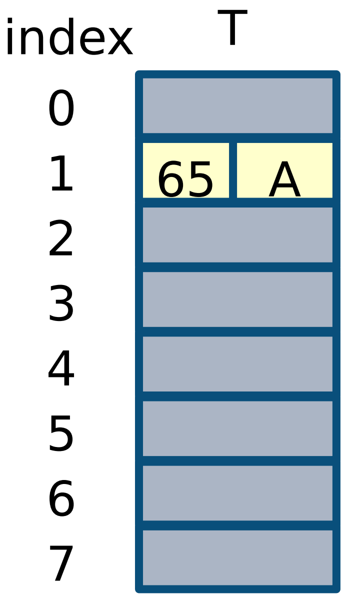
Example

- **ASCII table with hashing function h(k) = k mod 8**
  - U = {0,…127} and size of hash table T is m = 8
- **Insert (65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’) and (83, ‘S’) in hash table T**



Example

- **ASCII table with hashing function h(k) = k mod 8**
  - U = {0,...127} and size of hash table T is m = 8
- **Insert (65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’) and (83, ‘S’)**



- INSERT(T, (65, ‘A’))
- h(65) = 65 mod 8 = 1
- Insert element in slot 1

Example

- **ASCII table with hashing function h(k) = k mod 8**
  - U = {0,…127} and size of hash table T is m = 8
- **Insert (65, 'A'), (71, 'G'), (113, 'q'), (109, 'm') and (83, 'S')**

index	T	
0		
1	65	A
2		
3		
4		
5		
6		
7	71	G

- INSERT(T, (71, 'G'))
- h(71) = 71 mod 8 = 7
- Insert element in slot 7

Example

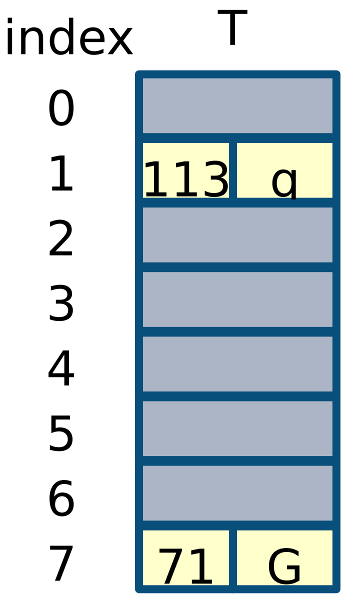
- **ASCII table with hashing function h(k) = k mod 8**
  - U = {0,…127} and size of hash table T is m = 8
- **Insert (65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’) and (83, ‘S’)**

index	T	
0		
1	65	A
2		
3		
4		
5		
6		
7	71	G

- INSERT(T, (113, ‘q’))
- h(113) = 113 mod 8 = 1
- Insert element in slot 1, but slot 1 is already occupied
- We say that keys 65 and 113 collide

Example

- **ASCII table with hashing function h(k) = k mod 8**
  - U = {0,…127} and size of hash table T is m = 8
- **Insert (65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’) and (83, ‘S’)**



- INSERT(T, (113, ‘q’))
- h(113) = 113 mod 8 = 1
- A (bad) strategy to resolve collisions is to store only the most recent key/value
- We will study more sophisticated strategies to resolve collisions later in this lecture

Example

- **ASCII table with hashing function h(k) = k mod 8**
  - U = {0,…127} and size of hash table T is m = 8
- **Insert (65, ‘A’), (71, ‘G’), (113, ‘q’), (109, ‘m’) and (83, ‘S’)**

index	T	
0		
1	113	q
2		
3		
4		
5	109	m
6		
7	71	G

- INSERT(T, (109, ‘m’))
- h(109) = 109 mod 8 = 5
- Insert element in slot 5

Example

- **ASCII table with hashing function h(k) = k mod 8**
  - U = {0,…127} and size of hash table T is m = 8
- **Insert (65, 'A'), (71, 'G'), (113, 'q'), (109, 'm') and (83, 'S')**

index	T	
0		
1	113	q
2		
3	83	S
4		
5	109	m
6		
7	71	G

- INSERT(T, (83, 'S'))
- h(83) = 83 mod 8 = 3
- Insert element in slot 3

Interpreting keys as natural numbers

- **Most hash functions assume the universe of keys $U = \mathbb{N}$ (the set of natural numbers)**

- **There are several methods (called hash codes) to convert an arbitrary value to an integer**
  - Memory address
  - Integer cast
  - Component sum
  - Polynomial accumulation

- **We briefly describe them one by one**

Memory address

- **Convert the memory address of the key into an integer**
  - Default hash code of all Java objects

- **Good in general, except for numeric and string keys**

Integer cast

- **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**
- **In Java, the int data type is 32-bit**
  - This method can be used cast keys of type byte (8-bit), short (16-bit), char (16-bit), and float (32-bit)
  - Keys of type long (64-bit) and double (64-bit) cannot use this method

Component sum

- **We partition the bits of the key into components of fixed length (16 or 32 bits) and we sum the components**
  - Overflows are ignored

- **Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type**
  - long and double in Java

Polynomial accumulation

- **The bits of the key are partitioned into a sequence of components of fixed length $a_0a_1\ldots a_{n-1}$**
  - Typical lengths are 8, 16 or 32 bits

- **Evaluate the polynomial $p(z) = a_0 + a_1z + a_2z^2 + \ldots + a_{n-1}z^{n-1}$ at a fixed value z**
  - If the key is long, it may cause an overflow

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

Example

- **A compiler uses a hash table to efficiently map identifiers to values**

- **Consider the identifier specified by string “tmp”**
- **There are three 16-bit partitions (one for each char) easily mappable to integers using the ASCII code**
  - a_0 = t = 116
  - a_1 = m = 109
  - a_2 = p = 112

- **“tmp” can be converted to an integer by evaluating $p(z) = 116 + 109z + 112z^2$**
  - p(33) = 125,681 p(7) = 6367 p(3) = 1451 p(128) = 1,849,076

Horner's rule

- **Optimal method to evaluate polynomial $p(z) = a_0 + a_1 z + a_2 z^2 + \ldots + a_{n-1} z^{n-1}$ in $O(n)$ time**

- **The following polynomials are successively computed, each from the previous one in $O(1)$ time**
  - $p_0(z) = a_{n-1}$
  - $p_i(z) = a_{n-i-1} + z p_{i-1}(z)$ $(i = 1, 2, \ldots, n - 1)$

- **We have $p(z) = p_{n-1}(z)$**

```
HORNER(A,z)
  key := 0
  for i=n-1 downto 0
    key := A[i] + z * key
```

Hash functions

- **The hash function should be simple to compute**

- **Purpose is to spread random data as evenly as possible over the indices of array T**
  - It should yield each index with equal probability for random data

- **We will look at three different classes of hash function**
  - Truncation
  - Division
  - Multiplication

- **Will assume our keys are integers (we already discussed conversion methods)**

Truncation

- **Take the first few or last few digits**
  - It generates many collisions if there are regularities in the input keys

- **Example**
  - Student IDs consisting of 6 digits. Numbers are assigned sequentially, so at any given time only a small subset of possible numbers are in use. Students in a given class will tend to have IDs close together, and all beginning with the same first few digits
  - Suppose we take first three digits as hash value and hash into a table of size 1000. Likely that they all start with same 1 or 2 digits (say a 4 or a 5, for example). So at most 200 of 1000 values actually used. Frequent collisions, and lots of empty space!
  - Taking the last three digits will probably work ok

Division

- **Map a key k into one of m slots by taking the remainder of k divided by m**
  - The hash function is $h(k) = k \bmod m$
  - Quite fast since it requires only a single division operation

- **When using the division method, we usually avoid certain values of m**
  - m should not be a power of 2, since if $m = 2^p$, then $h(k)$ is just the p lowest-order bits of k

- **To ensure that data is fairly distributed, choose table size to be**
  - Prime
  - Not too close to an exact power of 2

Example

- **Suppose we wish to allocate a hash table to hold roughly 5000 keys**

- **We pick m to be a prime close to 5000 but not near any power of 2**
  - $2^{12} = 4069$
  - $2^{13} = 8192$

- **Primes near 5000**
  - 4987, 4993, 4999, 5003, 5009

- **Our hash function would be h(k) = k mod 5003**

Multiplication

- **This method consists of two steps**
  1. Multiply key k by a constant $0 < A < 1$ (real number) and extract the fractional part of kA. Recall $\mathrm{frac}(kA) = kA - \mathrm{floor}(kA)$
  2. Multiply $\mathrm{frac}(kA)$ by m and take the floor

- **The hash function is defined as $h(k) = \mathrm{floor}(m\ \mathrm{frac}(kA))$**

- **An advantage of the multiplication method is that the value of m is not critical**
  - Typically a power of 2 is chosen
  - Knuth suggests to set $A \approx (\sqrt{5}-1)/2 = 0.6180339887...$

Conjugate of the golden ratio

Example

- **From Cormen book (11.3.2)**
  - Concrete implementation of the hashing function

- **Suppose the word size of the machine is w bits and k fits into a single word**
- **We restrict A to be in the form $s/2^w$ with $0 < s < 2^w$**

- **After multiplication $k * s$ the result is a fixed point $2w$-bit value $r_1 2^w + r_0$**
  - r_1 is the high-order word while r_2 is the low-order word

- **We then extract the p most significative bits of r_0 to obtain the final hash**
  - With $m = 2^p$

Example (cont.)

- **Compute the hash of key $k = 3278$ using the multiplication method**
  - Assume $p = 11$, $m = 2^{11} = 2048$ and $w = 32$

Example (cont.)

- **Compute the hash of key k = 3278 using the multiplication method**
  - Assume $p = 11$, $m = 2^{11} = 2048$ and $w = 32$

- **Pick A as the fraction $s/2^w$ that is closest to $(\sqrt{5}-1)/2$**
  - $s = \text{floor}((\sqrt{5}-1)/2 * 2^w) = \text{floor}((\sqrt{5}-1)/2 * 2^{32}) = 2654435769$
  - Usually this value is precomputed for a given architecture (16, 32, 64 bits)

Example (cont.)

- **Compute the hash of key $k = 3278$ using the multiplication method**
  - Assume $p = 11$, $m = 2^{11} = 2048$ and $w = 32$
- **Pick A as the fraction $s/2^w$ that is closest to $(\sqrt{5}-1)/2$**
  - $s = 2654435769$
- **After multiplication $k * s$ the result is 8,701,240,450,782**
  - Splitting in two 32-bit words we obtain $(2025 * 2^{32}) + 3{,}931{,}676{,}382$ with $r_1 = 2025$ and $r_0 = 3{,}931{,}676{,}382$
  - $r_0 = (k*s) \bmod 2^w = (k*s) \bmod 2^{32} = 3{,}931{,}676{,}382$
  - $r_1 = ((k*s) - r_0)/2^w$ This is usually not computed as it is not required for the computation of the final hash

Example (cont.)

- **Compute the hash of key k = 3278 using the multiplication method**
  - Assume $p = 11$, $m = 2^{11} = 2048$ and $w = 32$
- **Pick A as the fraction $s/2^w$ that is closest to $(\sqrt{5}-1)/2$**
  - $s = 2654435769$
- **After multiplication $k * s$ the result is 8,701,240,450,782**
  - $r_0 = 3{,}931{,}676{,}382$
- **The final hash is obtained by extracting the $p = 11$ most significative bits of r_0**
  - $h(k) = (r_0 – (r_0 \bmod 2^{w-p}))/ 2^{w-p} = (r_0 – (r_0 \bmod 2^{21}))/ 2^{21} = 1874$

Universal hashing

- **Any fixed hash function is vulnerable to malicious adversary choosing n keys that all hash to the same slot thus generating a lot of collisions**

- **In universal hashing at the beginning of execution we randomly select the hash function from a set of functions**
  - Randomization guarantees that no single input will always evoke worst-case behaviour

- **Let H be a finite collection of hash functions that map a given universe U of keys into the range $\{0, 1, \ldots, m - 1\}$**
- **Such a collection is said to be universal if for each pair of distinct keys $k,l \in U$, the number of hash functions $h \in H$ for which $h(k) = h(l)$ is at most $|H|/m$**

Universal class of hash functions

- **Choose a prime number p large enough so that every possible key k is in the range 0 to $p - 1$, inclusive**
  - $\mathbb{Z}_p = \{0,1,\ldots,p-1\}$, $Z_p^* = \{1,2,\ldots,p-1\}$ and $p > m$

- **Then define the hash function h_{ab} for any $a \in \mathbb{Z}_p^*$ and any $b \in \mathbb{Z}_p$ as follows**

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m$$

- **The family of all such functions is $H_{pm} = \{h_{ab} : a \in \mathbb{Z}_p^* \text{ and } b \in \mathbb{Z}_p\}$**

  - Size m of the output range is arbitrary—not necessarily prime
  - H_{pm} contains $p(p - 1)$ hash functions since we have $p - 1$ choices for a and p choices for b

Other hashing functions

- **Cyclic redundancy checks (CRCs)**
  - Good for longer keys
  - CRCs are about 3-4 times slower than multiplicative hashing
- **Cryptographic hash functions**
  - Cryptographic hash functions are hash functions that try to make it computationally infeasible to invert them
  - Examples: MD5 and SHA-1
  - MD5 is typically about twice as slow as using a CRC
- **Precomputing hash codes**
  - If the same values are being hashed repeatedly, one trick is to precompute their hash codes and store them with the value

Summary

- Map ADT
- Implementations
- Direct-address tables
- Hash table data structure
- Hash functions