

csci 210: Data Structures

Maps and Hash Tables




Summary

- Topics
 - the Map ADT
 - implementation of Map: hash tables
 - Hashing

Map ADT

- A Map is an abstract data structure (ADT)
 - it stores key-value (k,v) pairs
 - there cannot be duplicate keys
- Maps are useful in situations where a key can be viewed as a unique identifier for the object
 - the key is used to decide where to store the object in the structure. In other words, the key associated with an object can be viewed as the address for the object
 - maps are sometimes called associative arrays
- Note: Maps provide an alternative approach to searching

Map ADT

- size()
- isEmpty()
- get(k):  this can be viewed as searching for key k
 - if M contains an entry with key k, return it; else return null
- put(k,v):  this can be viewed as inserting key k
 - if M does not have an entry with key k, add entry (k,v) and return null
 - else replace existing value of entry with v and return the old value
- remove(k):  this can be viewed as deleting key k
 - remove entry (k,*) from M

Map example

(k,v) key=integer, value=letter

- $M = \{\}$
- put(5,A) $M = \{(5,A)\}$
- put(7,B) $M = \{(5,A), (7,B)\}$
- put(2,C) $M = \{(5,A), (7,B), (2,C)\}$
- put(8,D) $M = \{(5,A), (7,B), (2,C), (8,D)\}$
- put(2,E) $M = \{(5,A), (7,B), (2,E), (8,D)\}$
- get(7) return B
- get(4) return null
- get(2) return E
- remove(5) $M = \{(7,B), (2,E), (8,D)\}$
- remove(2) $M = \{(7,B), (8,D)\}$
- get(2) return null

Example

- Let's say you want to implement a language dictionary. That is, you want to store words and their definition. You want to insert words to the dictionary, and retrieve the definition given a word.
- Options:
 - vector
 - linked list
 - binary search tree
 - map
- The map will store (word, definition of word) pairs.
- key = word
 - note: words are unique
- value = definition of word
- get(word)
 - returns the definition if the word is in dictionary
 - returns null if the word is not in dictionary

Java.util.Map

- check out the interface
- additional handy methods
 - putAll
 - entrySet
 - containsValue
 - containsKey
- Implementation?

Class-work

- Write a program that reads from the user the name of a text file, counts the word frequencies of all words in the file, and outputs a list of words and their frequency.
 - e.g. text file: article, poem, science, etc
- Questions:
 - Think in terms of a Map data structure that associates keys to values.
 - What will be your <key-value> pairs?
 - Sketch the main loop of your program.

Map Implementations

- Arrays (Vector, ArrayList)
- Linked-list
- Binary search trees
- Hash tables

A LinkedList implementation of Maps

- store the (k,v) pairs in a doubly linked list
- get(k)
 - hop through the list until find the element with key k
- put(k,v)
 - Node x = get(k)
 - if (x != null)
 - replace the value in x with v
 - else create a new node(k,v) and add it at the front
- remove(k)
 - Node x = get(k)
 - if (x == null) return null
 - else remove node x from the list
 - Note: why doubly-linked? need to delete at an arbitrary position
- Analysis: $O(n)$ on a map with n elements

Map Implementations

- **Linked-list:**
 - get/search, put/insert, remove/delete: $O(n)$
- Binary search trees <----- we'll talk about this later
 - search, insert, delete: $O(n)$ if not balanced
 - $O(\lg n)$ if balanced BST
- **Hash tables:**
 - we'll see that (under some assumptions) search, insert, delete: $O(1)$

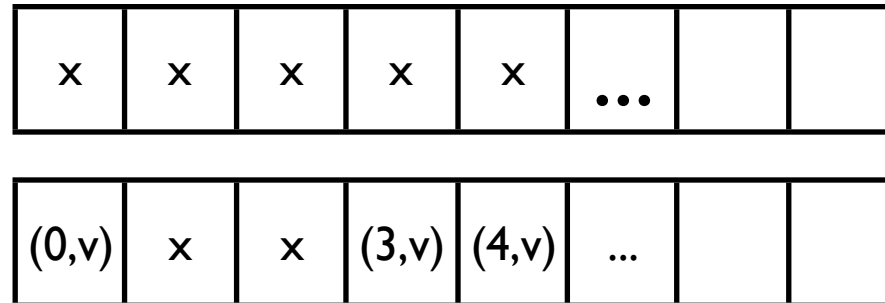
Hashing

- A completely different approach to searching from the comparison-based methods (binary search, binary search trees)
 - rather than navigating through a dictionary data structure comparing the search key with the elements, hashing tries to reference an element in a table directly based on its key
 - hashing transforms a key into a table address

Hashing

- If the keys were integers in the range 0 to 99
- The simplest idea:
 - store keys in an array $H[0..99]$
 - H initially empty

direct addressing:
store key k at index k



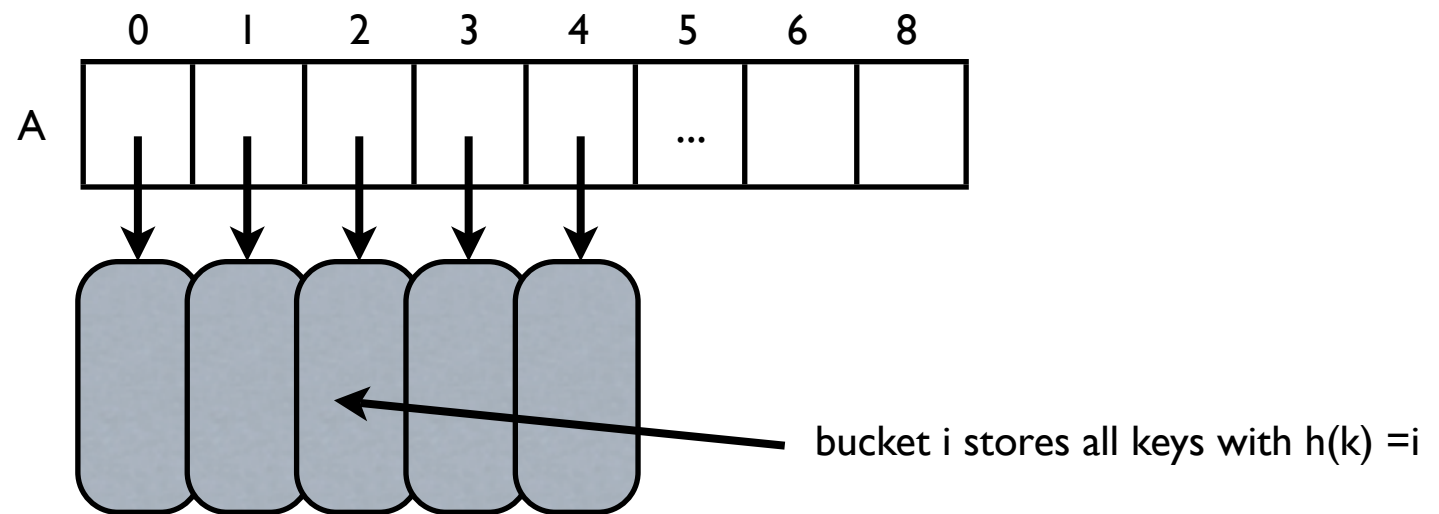
- $\text{put}(k, \text{value})$
 - store $\langle k, \text{value} \rangle$ in $H[k]$
- $\text{get}(k)$
 - check if $H[k]$ is empty

issues:

- keys need to be integers in a small range
- space may be wasted if H not full

Hashing

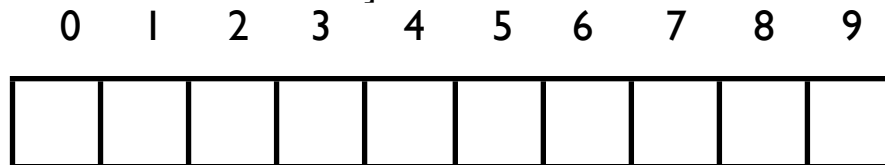
- Hashing has 2 components
 - the hash table: an array A of size N
 - each entry is thought of a bucket: a bucket array
 - a hash function: maps each key to a bucket
 - h is a function : $\{\text{all possible keys}\} \rightarrow \{0, 1, 2, \dots, N-1\}$
 - key k is stored in bucket $h(k)$



- The size of the table N and the hash function are decided by the user

Example

- keys: integers
- chose $N = 10$
- chose $h(k) = k \% 10$
 - [$k \% 10$ is the remainder of $k/10$]



- add (2,*), (13,*), (15,*), (88,*), (2345,*), (100,*)
- Collision: two keys that hash to the same value
 - e.g. 15, 2345 hash to slot 5
- Note: if we were using direct addressing: $N = 2^{32}$. Unfeasible.

Hashing

- $h : \{\text{universe of all possible keys}\} \rightarrow \{0,1,2,\dots,N-1\}$
- The keys need not be integers
 - e.g. strings
 - define a hash function that maps strings to integers
- The universe of all possible keys need not be small
 - e.g. strings
- Hashing is an example of space-time trade-off:
 - if there were no memory(space) limitation, simply store a huge table
 - $O(1)$ search/insert/delete
 - if there were no time limitation, use a linked list and search sequentially
- Hashing: use a reasonable amount of memory and strike a balance space-time
 - adjust hash table size
- Under some assumptions, hashing supports insert, delete and search in $O(1)$ time

Hashing

- Notation:

- U = universe of keys
- N = hash table size
- n = number of entries
 - note: n may be unknown beforehand

- Goal of a hash function:

- the probability of any two keys hashing to the same slot is $1/N$

called “universal hashing”



- Essentially this means that the hash function throws the keys uniformly at random into the table

- If a hash function satisfies the universal hashing property, then the expected number of elements that hash to the same entry is n/N

- if $n < N$: $O(1)$ elements per entry
- if $n \geq N$: $O(n/N)$ elements per entry

Hashing

- Choosing h and N
 - Goal: distribute the keys
 - n is usually unknown
 - If $n > N$, then the best one can hope for is that each bucket has $O(n/N)$ elements
 - need a good hash function
 - search, insert, delete in $O(n/N)$ time
 - If $n \leq N$, then the best one can hope for is that each bucket has $O(1)$ elements
 - need a good hash function
 - search, insert, delete in $O(1)$ time
 - If N is large \implies less collisions and easier for the hash function to perform well
 - Best: if you can guess n beforehand, choose N order of n
 - no space waste

Hash functions

- How to define a good hash function?
- An ideal hash function approximates a random function: for each input element, every output should be in some sense equally likely
- In general impossible to guarantee
- Every hash function has a worst-case scenario where all elements map to the same entry
- Hashing = transforming a key to an integer
- There exists a set of good heuristics

Hashing strategies

- Casting to an integer
 - if keys are short/int/char:
 - $h(k) = (\text{int}) k$;
 - if keys are float
 - convert the binary representation of k to an integer
 - in Java: $h(k) = \text{Float.floatToIntBits}(k)$
 - if keys are long long
 - $h(k) = (\text{int}) k$
 - lose half of the bits
- Rule of thumb: want to use all bits of k when deciding the hash code of k
 - better chances of hash spreading the keys

Hashing strategies

- Summing components
 - let the binary representation of key $k = \langle x_0, x_1, x_2, \dots, x_{k-1} \rangle$
 - use all bits of k when computing the hash code of k
 - sum the high-order bits with the low-order bits
 - $(\text{int}) \langle x_0, x_1, x_2, \dots, x_{31} \rangle + (\text{int}) \langle x_{32}, \dots, x_{k-1} \rangle$
 - e.g. String s ;
 - sum the integer representation of each character
 - $(\text{int})s[0] + (\text{int})s[1] + (\text{int})s[2] + \dots$

Hashing strategies

- summation is not a good choice for strings/character arrays
- e.g. $s_1 = \text{"temp10"}$ and $s_2 = \text{"temp01"}$ collide
- e.g. "stop" , "tops" , "pots" , "spot" collide
- Polynomial hash codes
 - $k = \langle x_0, x_1, x_2, \dots, x_{k-1} \rangle$
 - take into consideration the position of $x[i]$
 - chose a number $a > 0$ ($a \neq 1$)
 - $h(k) = x_0 a^{k-1} + x_1 a^{k-2} + \dots + x_{k-2} a + x_{k-1}$
 - experimentally, $a = 33, 37, 39, 41$ are good choices when working with English words
 - produce less than 7 collision for 50,000 words!!!
 - Java hashCode for Strings uses one of these constants

Hashing strategies

- Need to take into account the size of the table
- Modular hashing
 - $h(k) = i \bmod N$
 - If take N to be a prime number, this helps the spread out the hashed values
 - If N is not prime, there is a higher likelihood that patterns in the distribution of the input keys will be repeated in the distribution of the hash values
 - e.g. keys = {200, 205, 210, 215, 220, ... 600}
 - $N = 100$
 - each hash code will collide with 3 others
 - $N = 101$
 - no collisions

Hashing strategies

- Combine modular and multiplicative:
 - $h(k) = a k \% N$
 - chose $a =$ random value in $[0,1]$
 - advantage: the value of N is not critical and need not be prime
- empirically:
 - a popular choice is $a = 0.618033$ (the golden ratio)
 - chose $N =$ power of 2

Hashing strategies

- If keys are not integers
 - transform the key piece by piece into an integer
 - need to deal with large values
- e.g. key = string
 - $h(k) = (s_0a^{k-1} + s_1a^{k-2} + \dots + s_{k-2}a + s_{k-1}) \% N$
 - for e.g. $a = 33$
 - Horner's method: $h(k) = (((((s_0a + s_1)* a + s_2)*a + s_3)*a + \dots)*a + s_{k-1})$

```
int hash (char[] v, int N) {  
    int h = 0, a = 33;  
    for (int i=0; i< v.length; i++)  
        h = (a *h + v[i])  
    return h % N;  
}
```

- the sum may produce a number than we can represent as an integer
- take $\%N$ after every multiplication



```
int hash (char[] v, int N) {  
    int h = 0, a = 33;  
    for (int i=0; i< v.length; i++)  
        h = (a *h + v[i]) %N  
    return h;  
}
```

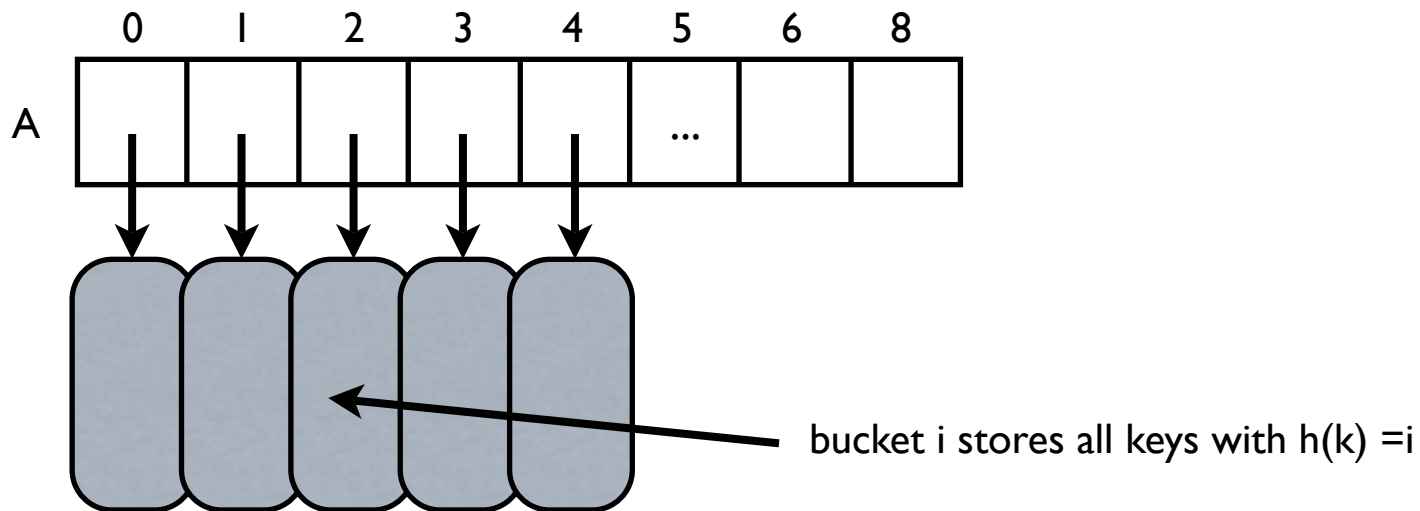

Hashing strategies

- Universal hashing
 - chose N prime
 - chose p a prime number larger than N
 - chose a, b at random from $\{0, 1, \dots, p-1\}$
 - $h(k) = ((a k + b) \bmod p) \bmod N$
 - gets very close to having two keys collide with probability $1/N$
 - i.e. to throwing the keys into the hash table randomly
- Many other variations of these have been studied, particularly has functions that can be implemented with efficient machine instructions such as shifting

Hashing

- Hashing
 1. hash function
 - convert keys into table addresses
 2. collision handling
 - Collision: two keys that hash to the same value
 - Decide how to handle when two keys hash to the same address
- Note: if $n > N$ there must be collisions
- Collision with chaining
 - bucket arrays
- Collision with probing
 - linear probing
 - quadratic probing
 - double hashing

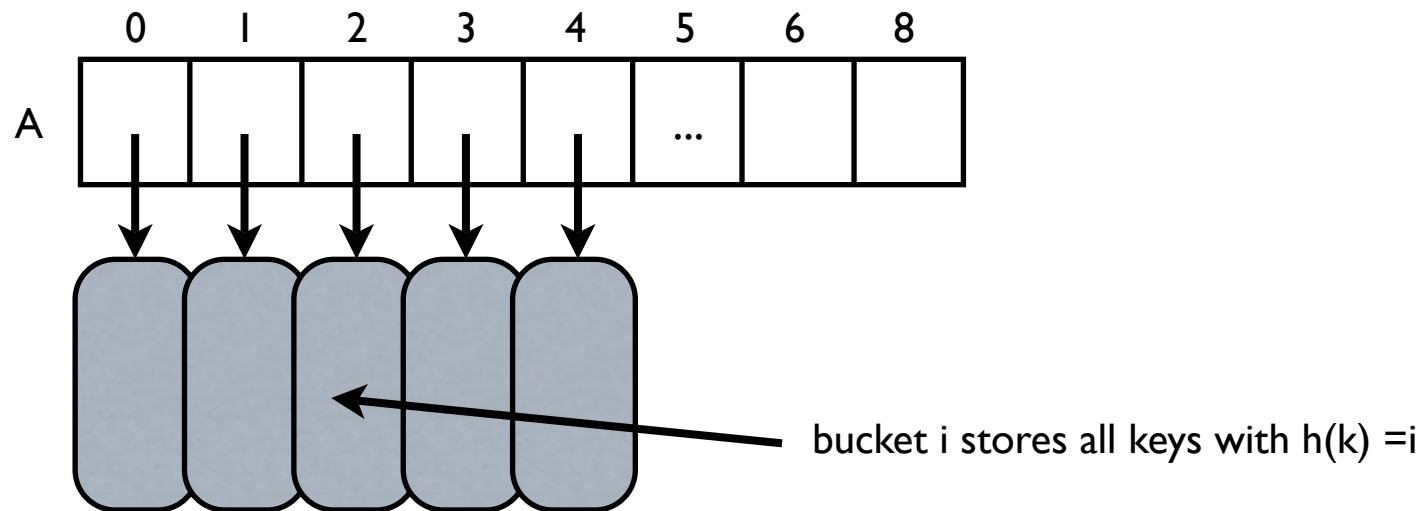
Collisions with chaining



- Store all elements that hash to the same entry in a linked list (array/vector)
- Can chose to store the lists in sorted order or not
- **Insert(k)**
 - insert k in the linked list of $h(k)$
- **Search(k)**
 - search in the linked list of $h(k)$

under universal hashing:
each list has size $O(n/N)$ with high probability
insert, delete, search: $O(n/N)$

Collisions with chaining



- Pros:
 - can handle arbitrary number of collisions as there is no cap on the list size
 - don't need to guess n ahead: if N is smaller than n , the elements will be chained
- Cons: space waste
 - use additional space in addition to the hash table
 - if N is too large compared to n , part of the hash table may be empty
- Choosing N : space-time tradeoff

Collisions with probing

- Idea: do not use extra space, use only the hash table
- Idea: when inserting key k , if slot $h(k)$ is full, then try some other slots in the table until finding one that is empty
 - the set of slots tried for key k is called the probing sequence of k
- Linear probing:
 - if slot $h(k)$ is full, try next, try next, ...
 - probing sequence: $h(k), h(k) + 1, h(k) + 2, \dots$
 - insert(k)
 - search(k)
 - delete(k)
- Example: $N = 10, h(k) = k \% 10$, collisions with linear probing
- insert 1, 7, 4, 13, 23, 25, 25

Linear probing

- Notation: $\alpha = n/N$ (load factor of the hash table)
- In general performance of probing degrades inversely proportional with the load of the hash
 - for a sparse table (small α) we expect most searches to find an empty position within a few probes
 - for a nearly full table (α close to 1) a search could require a large number of probes
- Proposition:

Under certain randomness assumption it can be shown that the average number of probes examined when searching for key k in a hash table with linear probing is $\frac{1}{2} (1 + \frac{1}{1 - \alpha})$

 - [No proof]
 - $\alpha = 0$: 1 probe
 - $\alpha = 1/2$: 1.5 probes (half-full)
 - $\alpha = 2/3$: 2 probes (2/3 full)
 - $\alpha = 9/10$: 5.5 probes
- Collisions with probing: cannot insert more than N items in the table
 - need to guess n ahead
 - if at any point n is $> N$, need to re-allocate a new hash table, and re-hash everything. Expensive!

Linear probing

- Pros:
 - space efficiency
- Con:
 - need to guess n correctly and set $N > n$
 - if α gets large \implies high penalty
 - the table is resized and all objects re-inserted into the new table
- Rule of thumb: good performance with probing if α stays less than $2/3$.

Double hashing

- Empirically linear hashing introduces a phenomenon called clustering:
 - insertion of one key can increase the time for other keys with other hash values
 - groups of keys clustered together in the table
- Double hashing:
 - instead of examining every successive position, use a second hash function to get a fixed increment
 - probing sequence: $h_1(k), h_1(k) + h_2(k), h_1(k) + 2h_2(k), h_1(k) + 3h_2(k), \dots$
- Chose h_2 so that it never evaluates to 0 for any key
 - would give an infinite loop on first collision
- Rule of thumb:
 - chose $h_2(k)$ relatively prime to N
- Performance:
 - double hashing and linear hashing have the same performance for sparse tables
 - empirically double hashing eliminates clustering
 - we can allow the table to become more full with double hashing than with linear hashing before performance degrades

Java.util.Hashtable

- This class implements a hash table, which maps keys to values. Any non-null object can be used as a key or as a value.
- [java.lang.Object](#)
- [java.util.Dictionary](#)
- **java.util.Hashtable**
- **implements Map**
- [check out Java docs]
- implements a Map with linear probing; uses .75 as maximal load factor, and rehashes every time the table gets fuller
- Example

```
//create a hashtable of <key=string, value=number> pairs
Hashtable numbers = new Hashtable();
numbers.put("one", new Integer(1));
numbers.put("two", new Integer(2));
numbers.put("three", new Integer(3));

//retrieve a string
Integer n = (Integer)numbers.get("two");
if (n != null) {
    System.out.println("two = " + n);
}
```

Hash functions in Java

- The generic `Object` class comes with a default `hashCode()` method that maps an `Object` to an integer
 - `int hashCode()`
- Inherited by every `Object`
- The default `hashCode()` returns the address of the `Object`'s location in memory
 - too generic
 - poor choice for most situations
- Typically you want to override it
- e.g. class `String`
 - overrides `String.hashCode()` with a hash function that works well on `Strings`

Perspective

- Best hashing method depends on application
- Probing is the method of choice if n can be guessed
 - Linear probing is fastest if table is sparse
 - Double hashing makes most efficient use of memory as it allows the table to become more full, but requires extra time to compute a second hash function
 - rule of thumb: load factor $< .66$
- Chaining is easiest to implement and does not need guessing n
 - rule of thumb: load factor $< .9$ for $O(1)$ performance, but not vital
- Hashing can provide better performance than binary search trees if the keys are sufficiently random so that a good hash function can be developed
 - when hashing works, better use hashing than BST
- However
 - Hashing does not guarantee worst-case performance
 - Binary search trees support a wider range of operations

Exercises

- What is the worst-case running time for inserting n key-value pairs into an initially empty map that is implemented with a list?
- Describe how to use a map to implement the basic ops in a dictionary ADT, assuming that the user does not attempt to insert entries with the same key
- Describe how an ordered list implemented as a doubly linked list could be used to implement the map ADT.
- Draw the 11-entry hash that results from using the hash function $h(i) = (2i+5) \bmod 11$ to hash keys 12, 44, 13, 88, 23, 94, 11, 39, 20, 16, 5.
 - (a) Assume collisions are handled by chaining.
 - (b) Assume collisions are handled by linear probing.
 - (c) Assume collisions are handled with double hashing, with the secondary hash function $h'(k) = 7 - (k \bmod 7)$.
- Show the result of rehashing this table in a table of size 19, using the new hash function $h(k) = 2k \bmod 19$.
- Think of a reason that you would not use a hash table to implement a dictionary.