Lecture 5: Hash Tables

Instructor: Olga Veksler

© 2004 Goodrich, Tamassia
Outline

- Hash Tables
  - Motivation
  - Hash functions
  - Collision handling
    - Separate Chaining (open addressing)
      - Linked list, any other container class
    - Closed addressing
      - Linear Probing
      - Double hashing
Dictionary

- Dictionary stores key-value pairs \((k,v)\), called entries
- Linked List implementation
  - \(O(1)\) to insert and \(O(n)\) to find an element
- Ordered array implementation
  - \(O(n)\) to insert and \(O(\log n)\) to find an element
- Can we have a more efficient dictionary?
  - insert, find, delete \(O(1)\)?
  - Hash tables have \(O(1)\) expected time
Hash Table Motivation

- Suppose know we will have at most \( N \) entries \((k,v)\)
- Suppose keys \( k \) are **unique** integers between 0 to \( N - 1 \)
- Create initially empty array \( A \) of size \( N \)
- Store \((k,v)\) in \( A[k] \)
- Example

  \[
  (1, `ape`) \quad (3, `or`) \quad (N-1, `sad`) 
  \]

  ![Hash Table Example](image)

- Main operations (insert, find, remove) are \( O(1) \)
- need \( O(N) \) space
Hash Table Motivation

- What if still have \( N \) keys, but they may be not unique? 
  \((1,A)\) \((1,R)\) \((1,C)\) \((3,D)\)

- Use **bucket** array:

  ![Bucket Diagram]

  A bucket can be implemented as a linked list
  - Assume have at most a constant number (say 3) of repeated keys
  - methods find(), remove(), insert() are still \( O(1) \)
Hash Table Motivation

1. What if we will have at most 100 entries with integer keys but the keys are in range 0 to 1,000,000,000?
   - still want $O(1)$ insert(), delete(), find(),
   - But do not want to use 1,000,000,000 memory cells to store only 100 entries

2. What if keys are not integers?
   - These 2 issues above motivate a Hash Table data structure
     - insert, find, delete are $O(1)$ expected (average) time
     - worst-case time is still $O(n)$
Hash Table: Main Idea

- Bucket array $A$ of size $N$
- Design function $h(k)$ that maps key $k$ into integer range $0, 1, ..., N - 1$
- Entry with key $k$ is stored at index $h(k)$ of array $A$

$0 \quad 1 \quad 2 \quad 3 \quad \cdots \quad N-1$

$h(k) = 1$
$h(k) = 3$

$(k,v)$
$(k,v)$
Hash Functions and Hash Tables

- A hash function $h$ maps keys of given type to integers in interval $[0, N - 1]$
- Example for integer keys $x$
  \[ h(x) = x \mod N \]
- The integer $h(k)$ is called the hash value of key $k$
- A hash table for a given key type consists of
  - hash function $h$
  - array (called table) of size $N$
- Store item $(k, v)$ at index $i = h(k)$
- Collision happens when for $k_1 \neq k_2$, $h(k_1) = h(k_2)$
  - bucket array is one way to handle collisions
Hash Table Example

- Company has 5,000 employees, store information with key = SSN
  - SNN is a nine digit positive integer
  - If stored info under full 10 digit SSN, need array of size 9,999,999,999
- Hash table for storing entries (SSN, info)
  - Array of size \( N = 10,000 \)
  - Hash function \( h(x) = \) last four digits of \( x \)
Hash Functions

- Specify hash function as composition of two functions

  Hash code
  \( h_1: \text{keys} \rightarrow \text{integers} \)

  Compression function
  \( h_2: \text{integers} \rightarrow [0, N - 1] \)

- Hash code is applied first, and compression function second

  \( h(x) = h_2(h_1(x)) \)

- Collision: two different keys assigned the same hash code

- To avoid collisions, the goal of the hash function is to “disperse” the keys in an apparently random way

  - both hash code and compression function should be designed so as to avoid collisions as much as possible
Hash Codes

- Hash code maps a key $k$ to an integer
  - not necessarily in the desired range $[0,\ldots,N-1]$
  - may be even negative
- We assume a hash code is a 32 bit integer
- Hash code should be designed to avoid collisions as much as possible
  - if hash code causes collision, then compression function will not resolve this collision
Memory Address Hash Code

- Interpret memory address of the key object as integer
  - sometimes done in Java implementations
  - any object inherits hashCode() method
- Sometimes sufficient, but often does not make sense
- Usually want objects with equal content to have the same hash code, this may not happen with inherited hashCode()
  - want s1 = “Hello” and s2 = “Hello” to have same hash code
  - s1 and s2 are stored at different memory locations
  - they do not have the same hashCode()
An object is represented as a sequence of bits

For objects with the same content, bit representation is equal
- hash code of objects with the same content will be the same

Reinterpret sequence of bits representing a key as an integer

Suitable with no modification for keys with bit representation of length less than or equal to 32
- for `byte`, `short`, `int`, `char` cast into type `int`
- for `float`, use `Float.floatToIntBits()`

What about types longer than 32 bits, i.e. `long` or `double`?
- can cast into `int`, but upper half of the bits will not be used

<table>
<thead>
<tr>
<th>first 32 bits</th>
<th>next 32 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- many collisions if most of the key differences are in those lost bits
Component Sum Hash Code

- Partition the bits of the key into components of fixed length of 32 bits and we sum the components ignoring overflows.

- Suitable for numeric keys of fixed length greater than or equal to the number of bits of the integer type.
  - e.g. long and double in Java
Component Sum Hash Code

- Can do this for object of any length
- Suppose we have 128 bit object
  
  ![Diagram showing 128 bit object divided into four 32-bit parts](image)

  - first 32 bits
  - next 32 bits
  - next 32 bits
  - last 32 bits

- Add the parts

  ![Diagram showing addition of the parts](image)

  - hash code
Can do this even for variable length objects

variable-length object can be viewed as tuple

$$(x_0, x_1, \ldots, x_{k-1})$$

Component sum hash code is

$$x_0 + x_1 + \ldots + x_{k-1}$$
Component Sum Hash Code: Strings

- Convenient to break strings into 8 bit components
  - i.e. characters
- Sum up characters in String $s$
- “abracadabra” = `a’ + `b’+...+ `a’
- Many collisions
  - “stop”, “tops”, “pots”, “spot” have the same hash code
- Problem: position of individual characters is important, but not taken into account by the component sum hash code
Hash Code: Polynomial Accumulation

- Object is a variable length tuple \((x_0, x_1, \ldots, x_{k-1})\) and the order of \(x_i\)'s is significant.
- Choose non-zero constant \(c \neq 1\).
- Polynomial hash code is:
  \[
p(c) = x_0 + x_1 c + x_2 c^2 + \ldots + x_{k-1} c^{k-1}
\]
  - Overflows are ignored.
- Multiplication by \(c\) makes room for each component in a tuple of values, while also preserving a characterization of the previous components.
Hash Code: Polynomial Accumulation

- Especially suitable for strings
- Experimentally, good choices \( c = 33, 37, 39, 41 \)
- Choice \( c = 33 \) gives at most 6 collisions on a set of 50,000 English words
- Let \( a = 33 \)
  - “top” = ‘t’+’o’*33+’p’*33^2=116+111*33+112*33^2=125747
  - “pot” = ‘p’+’o’*33+’t’*33^2=112+111*33+116*33^2=130099
Polynomial $p(c)$ evaluated in $O(k)$ time using Horner’s rule

$$p(c) = x_0 + x_1 c + x_2 c^2 + \ldots + x_{k-1} c^k$$

$$= x_0 + c(x_1 + c(x_2 + \ldots c(x_{n-3} + c(x_{n-2} + x_{n-1} c)))) \ldots$$

Compute:

- $p_1 = x_{n-1}$
- $p_2 = c p_1 + x_{n-2}$
- \ldots
- $p_i = c p_{i-1} + x_{n-i}$

Algorithm

```
p = x[k-1]
i = k - 2
while i \geq 0 do
    p = p*c + x[i]
i = i - 1
```
Hash Code: Polynomial Accumulation

\[ p(c) = x_0 + x_1 c + x_2 c^2 + ... + x_{k-1} c^k \]

- Alternatively, can accumulate powers of \( c \)
- Also \( O(k) \) efficiency

**Algorithm**

\[
\begin{align*}
p &= x[0] \\
\text{accum} &= c \\
\text{for } i &= 1 \text{ to } k-1 \text{ do} \\
& \quad p = p + x[i]*\text{accum} \\
& \quad \text{accum} = \text{accum}*c
\end{align*}
\]
Compression Functions

- Now know how to map objects to integers using a suitable hash code
- The hash code for key $k$ will typically not be in the legal range $[0,\ldots,N-1]$
- Need **compression function** to map the hash code into the legal range $[0,\ldots,N-1]$
- Good compression function will minimize the number of collisions
Division Compression Function

- \( h_2(y) = |y| \mod N \)
- \( N \) should be a prime number
  - helps to spread out the hashed values
  - the reason is beyond the scope of this course
- Example: \{200,205,210,300,305,310,400,405,410\}
  - \( N=100 \), hashed values \{0,5,10,0,5,10,0,5,10\}={0,5,10}
  - \( N=101 \), hashed values \{99,3,8,98,2,7,97,1,6\}
- \( \mod N \) compression does not work well if there is repeated pattern of hash codes of form \( pN+q \) for different \( p \)'s
MAD compression Function (Multiply Add and Divide)

- \( h_2(y) = |ay + b| \ mod \ N \)
- \( N \) is a prime number
- \( a > 0 \) and \( b \geq 0 \) are integers such that
  - \( a \ mod \ N \neq 0 \)
  - otherwise, every integer would map to the same value, namely \( (b \ mod \ N) \)
  - \( a \) and \( b \) are usually chosen randomly at the time when a MAD compression function is chosen

- This compression function spreads hash codes fairly evenly in the range \([0,\ldots,N-1]\)
- MAD compression is better than mod \( N \)
Collision Handling via Separate Chaining

- Handle collisions with bucket array
  - Bucket is a linked list or any other container data structure
- This is called **Separate Chaining**

![Bucket Array Diagram]

- 0: Empty
- 1: Linked list containing 025-612-0001
- 2: Empty
- 3: Empty
- 4: Linked list containing 451-229-0004 and 981-101-0004
Load Factor

- Useful to keep track of load factor of the hash table
  - Tells us if the hash array too small to store current number of entries
- Suppose bucket array has size $N$ and there are $n$ entries
- The load factor is defined as $l = \frac{n}{N}$
- Suppose that the hash function is good
  - spreads keys evenly in the array $A[0,...N]$  
- Then $\frac{n}{N}$ is expected number of items in each bucket
  - find, insert, remove, take $O(\frac{n}{N})$ expected time
- Ideally, each bucket should have at most 1 item
- Should keep the load factor $l < 1$
  - for separate chaining, recommended $l < 0.9$
- If load factor becomes too large, rehash
  - make hash array larger (at least twice) and re-insert all entries into the new hash array
Dictionary Methods with Separate Chaining

- Implement each bucket as list-based dictionary

**Algorithm** `insert(k,v):`

**Input:** A key k and value v

**Output:** Entry (k,v) is added to dictionary D

```plaintext
if (n+1)/N > 1 then  // Load factor became too large
double the size of A and rehash all existing entries

  e = A[h(k)].insert(k,v)  //A[h[k]] is a linked list
  n = n+1                //n is number of entries in hash table

return e
```
Algorithm findAll(k):
Input: A key k
Output: An iterator of entries with key equal to k

return A[h(k)].findAll(k)
Algorithm remove(e):
Input: an entry e
Output: The (removed) entry e or null if e was not in dictionary D

t = A[h(k)].remove(e)  // delegate the remove to dictionary at A[h(k)]
if t ≠ null then  // e was found
    n = n - 1  // update number of entries in hash table
return t
Open Addressing

- Separate chaining
  - Advantage: simple implementation
  - Disadvantage: additional storage requirements for auxiliary data structure (list)
- Can handle collisions without additional data structure, i.e., can store entries in hash array $A$
- This is called **open addressing**
  - $A$ is not a bucket array in this case
  - Load factor has to be always at most 1
Open Addressing: Linear Probing

- Handle collisions by placing the colliding item in the next (circularly) available table cell
- Each table cell inspected is referred to as a *probe*
- Example
  - $h(x) = x \mod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, 5, 6

```
   6  41  18  44  59  32  22  31  73  5
  0  1  2  3  4  5  6  7  8  9 10 11 12
```
find with Linear Probing

Algorithm find(k)
    hashVal ← h(k)
p ← 0
repeat
    i ← (hashVal + p) mod N
c ← A[i]
    if c = null
        return null
    if c.key() = k
        return c
    p ← p + 1
until p = N
return null

- start at cell h(k)
- probe consecutive locations until one of the following
  - item with key k is found, or
  - empty cell is found, or
  - N cells have been unsuccessfully probed
What about remove?

- \( h(x) = x \mod 13 \)

remove(18) \[\rightarrow\] remove(18)

find(31) \[\rightarrow\] find(31)

- Oops, 31 is not found now!
Solution for remove

- Replace deleted entry with special marker to signal that an entry was deleted from that cell
  - `null` is a special marker, but we already use it for another purpose
  - create an Entry object, let us call it `AVAILABLE`
    - make sure to instantiate `AVAILABLE` only once in your hash table

Entry `AVAILABLE = new Entry(someKey, someValue)`

- do not care what the `someKey` and `someValue` are, we never use them
- we use only the reference (address) of object `AVAILABLE`
  - if `A[index] == AVAILABLE`
Solution for remove

- remove(e)
  - search for entry e
  - if entry e is found, we replace it with AVAILABLE
  - else return null

Example: \( h(x) = x \mod 13 \)

```
0 1 2 3 4 5 6 7 8 9 10 11 12
A 44 59 32 22 31 73
```

\[
\text{remove(18)} \quad \rightarrow \quad 0 1 2 3 4 5 6 7 8 9 10 11 12
\]

```
0 1 2 3 4 5 6 7 8 9 10 11 12
A 44 59 32 22 31 73
```

\[
\text{find(31)} \quad \rightarrow \quad 0 1 2 3 4 5 6 7 8 9 10 11 12
\]

```
0 1 2 3 4 5 6 7 8 9 10 11 12
A 44 59 32 22 31 73
```
Linear Probing: fixed find

Algorithm find(k)
    hVal ← h(k)
    p ← 0
    repeat
        i ← (hVal + p) mod N
        c ← A[i]
        if c = null
            return null
        if c != AVAILABLE
            if c.key() = k
                return c
            p ← p + 1
        until p = N
    return null

- Do not access A[i].key before making sure A[i] ≠ AVAILABLE
**insert with Linear Probing**

- **insert**(*k*,*v*)
  - rehash if load factor becomes too large
  - start at cell *h*(*k*)
  - probe consecutive cells until find null or AVAILABLE cell found
  - insert new entry at that cell

**Example:** *h(x) = x mod 13*

```
0 1 2 3 4 5 6 7 8 9 10 11 12
```

- **insert(57)**
  ```
  0 1 2 3 4 5 6 7 8 9 10 11 12
  41 57 44 59 32 22 31 73
  ```

- **insert(4)**
  ```
  0 1 2 3 4 5 6 7 8 9 10 11 12
  41 4 57 44 59 32 22 31 73
  ```
Problems with Linear Probing

- Entries tend to cluster into contiguous regions
- Results in many probes per find, insert, and remove methods
- The more probes per find, insert, remove, the slower the code
Open Addressing: Double Hashing

- Linear Probing places item in first available cell in series
  \[( h(k) + p \cdot 1 ) \mod N \quad \text{for } p = 0, 1, \ldots, N - 1 \]

- *Double hashing* uses secondary hash function \( h'(k) \) and places item in first available cell in the series:
  \[( h(k) + p \cdot h'(k) ) \mod N \quad \text{for } p = 0, 1, \ldots, N - 1 \]

  - must have \( 0 < h'(k) < N \)
  - table size \( N \) *must* be a prime to allow probing of all cells
  - linear probing is a special case of double hashing with
    \[ h'(k) = 1 \quad \text{for all } k \]
  - double hashing spreads entries more evenly through hash array
Open Addressing: Double Hashing

- Good choice for secondary hash function
  
  \[ h'(k) = q - k \mod q \]

- where
  
  - \( q < N \)
  
  - \( q \) is a prime

- Possible values for \( h'(k) \) are
  
  \( 1, 2, \ldots, q \)

- Assumed \( k \) is integer

- If not, use \( h'(k) = q - |\text{HashCode}(k)| \mod q \)
Double Hashing Example

- \((h(k) + p*h'(k)) \mod N\) for \(p = 0, 1, \ldots, N - 1\)
  - \(N = 13\)
  - \(h(k) = k \mod 13\)
  - \(h'(k) = 7 - k \mod 7\)

- Insert keys in order 18, 41, 22, 44, 59, 32, 31, 73

- Insert 18
  - \(h(18) = 18 \mod 13 = 5\)
  - \(h'(18) = 7 - 18 \mod 7 = 3\)
  - first empty location in sequence
    - \((5 + 3\times0) \mod 13\)
    - \((5 + 3\times1) \mod 13\)
    - \((5 + 3\times2) \mod 13\)
Double Hashing Example

- Insert 41
  - $h(41) = 41 \mod 13 = 2$
  - $h'(41) = 7 - 41 \mod 7 = 1$
  - First empty location in sequence
    - $(2 + 1*0) \mod 13, (2 + 1*1) \mod 13, (2 + 1*2) \mod 13, ...$
Insert 22

- \( h(22) = 22 \mod 13 = 9 \)
- \( h'(22) = 7 - 22 \mod 7 = 6 \)
- first empty location in sequence: \( (9 + 6*0) \mod 13, (9 + 6*1) \mod 13, (9 + 6*2) \mod 13, \ldots \)
Double Hashing Example

- Insert 44
  - $h(44) = 44 \mod 13 = 5$
  - $h'(44) = 7 - 44 \mod 7 = 5$
  - first empty location in sequence: $(5 + 5*0) \mod 13$, $(5 + 5*1) \mod 13$, $(5 + 5*2) \mod 13$, ...

Insertion into the hash table:
Double Hashing Example

- Insert 59
  - \( h(59) = 59 \mod 13 = 7 \)
  - \( h'(59) = 7 - 59 \mod 7 = 4 \)
  - first empty location in sequence: \((7 + 4*0) \mod 13\), \((7 + 4*1) \mod 13\), \((7 + 4*2) \mod 13\), ...
Double Hashing Example

- **Insert 59**
  - $h(59) = 59 \mod 13 = 7$
  - $h'(59) = 7 - 59 \mod 7 = 4$
  - first empty location in sequence: $(7 + 4*0) \mod 13$, $(7 + 4*1) \mod 13$, $(7 + 4*2) \mod 13$, ...
Double Hashing Example

- Insert 32
  - \( h(32) = 32 \mod 13 = 6 \)
  - \( h'(32) = 7 - 32 \mod 7 = 3 \)
  - first empty location in sequence: \( (6 + 3*0) \mod 13, (6 + 3*1) \mod 13, (6 + 3*2) \mod 13, \ldots \)
Insert 31

- $h(31) = 31 \mod 13 = 5$
- $h'(31) = 7 - 31 \mod 7 = 4$
- first empty location in sequence: $(5 + 4 \times 0) \mod 13, (5 + 4 \times 1) \mod 13, (5 + 4 \times 2) \mod 13, ...$

Double Hashing Example
Double Hashing Example

- Insert 73
  - $h(73) = 73 \mod 13 = 8$
  - $h'(73) = 7 - 73 \mod 7 = 4$
  - first empty location in sequence: $(8 + 4*0) \mod 13$, $(8 + 4*1) \mod 13$, $(8 + 4*2) \mod 13$, ...

```
31 41 18 59 22 44
0 1 2 3 4 5 6 7 8 9 10 11 12
```


```
31 41 18 59 73 22 44
0 1 2 3 4 5 6 7 8 9 10 11 12
```
Double Hashing Example Summary

- Probe summary
  - 8 insertions, 11 probes
  - 11/8 probes on average

- Want average number of probes to be small

<table>
<thead>
<tr>
<th>$k$</th>
<th>$h(k)$</th>
<th>$h'(k)$</th>
<th>Probes</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>5</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>22</td>
<td>9</td>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>44</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>59</td>
<td>7</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>32</td>
<td>6</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>5</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>73</td>
<td>8</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

$$k$$ and $$h'(k)$$ are the original key and its hash, respectively. The table shows the number of probes for each key. The goal is to keep the average number of probes small.
Algorithm `find(k)`

- `hVal ← h(k)`
- `hVal2 ← h’(k)`
- `p ← 0`

\[ i ← (hVal + p) \mod N \]

\[ h’(k) = q - |\text{HashCode}(k)| \mod q \]

\[ \text{repeat} \]

\[ i ← |hVal + p*hVal2| \mod N \]

\[ c ← A[i] \]

\[ \text{if } c = \text{null} \]

\[ \text{return null} \]

\[ \text{if } c \neq \text{AVAILABLE} \]

\[ \text{if } c.key() = k \]

\[ \text{return } c \]

\[ p ← p + 1 \]

\[ \text{until } p = N \]

\[ \text{return null} \]
Open Addressing Performance

- Worst case: find(), insert() and remove() are $O(n)$
  - worst case occurs when all inserted keys collide
- Load factor $l = n/N$ affects performance
- Assuming that hash values are like random numbers, can show that expected number of probes for insert(), find(), remove() is

$$\frac{1}{1-l}$$

- Recommended to keep $l < 0.5$, then

$$\frac{1}{1-l} \leq 2$$
Separate Chaining vs. Open Addressing

- Open addressing saves space over separate chaining
- Separate chaining is usually faster (depending on load factor of the bucket array) than the open addressing, both theoretically and experimentally
- Thus, if memory space is not a major issue, use separate chaining, otherwise use open addressing
Separate Chaining vs. Open Addressing

- Open addressing saves space over separate chaining
- Separate chaining is usually faster (depending on load factor of the bucket array) than the open addressing, both theoretically and experimentally
- Thus, if memory space is not a major issue, use separate chaining, otherwise use open addressing
public interface HashCode<K> {
    public int giveCode(K k);}

public class HashDict<K,V> implements Dict<K,V> {
    private HashCode<K> hCode;
    public HashDict(HashCode<K> inCode, float maxLFactor){
        hCode  = inputCode;
        ...
    }
    public Entry<K,V> find(K key){
        int h = hCode.giveCode(key);
        ...
    }
}
public class WordPuzzle {
    StringHashCode = hC new StringHashCode();
    HashDict<String,Integer> D = new hashDict<String,Integer>(hC,(float)0.6);
}

public class StringHashCode implements HashCode<String>
{
    public int giveCode(String key)
    {
        ......
    }
}