Part 8. Hashing

CS 200 Algorithms and Data Structures

Outline

- Hashing
- Hash Functions
- Resolving Collisions
- Efficiency of Hashing
- Java Hashtable and HashMap

Table Implementations

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Add</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted array-based</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Unsorted array-based</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced Search Trees</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
</tbody>
</table>

Can we build a faster data structure?

Tables in $O(1)$

Suppose we have a magical address calculator...

```
tableInsert(in: newItem:TableItemType)
  i = index that the address calculator gives you
  for newItem's search key
  table[i] = newItem
```

Hash Functions and Hash Tables

Magical address calculators exist:
They are called **hash functions**

![Hash Table Diagram]
Outline

- Hashing
- **Hash Functions**
- Resolving Collisions
- Efficiency of Hashing
- Java Hashtable and HashMap

Simple Hash Functions

Credit card numbers
- 3: travel/entertainment cards (e.g. American Express and Diners Club)
  - Digits three and four are type and currency
  - Digits five through 11 are the account number
- 4: Visa
  - Digits two through six are the bank number
  - Digits seven through 12 or seven through 15 are the account number
- 5: Mastercard
  - Digits two and three, two through four, two through five or two through six are bank number
  - Till digits 15 are the account number
  - Digit 16 is a check digit
- To design a system to find an account based on the account number, we don’t need 16 or more digits of numbers on the credit card.

Other simple example

- Phone exchange: need quick access to record corresponding to phone #.
  \[ h(123-4567) = 34567 \]

Requirements (1/2)

- In the previous examples:
  - The hash function mapped each \( x \) to a **unique** integer \( h(x) \)
  - There was no empty space in the table
- We used domain knowledge to design the hash function
- We want general purpose hash functions!

Requirements (2/2)

- Desired properties:
  - Easy and fast to compute
  - Values evenly distributed
    - Within array size range

Hash function: Selecting digits

- \( h(001364825) = 35 \)
  - Select the fourth and last digits
- Simple and fast
  - Does not evenly distribute items
Hash function: Folding

• Suppose the search key is a 9-digit ID.
• Sum-of-digits:
  \[ h(001364825) = 0 + 0 + 1 + 3 + 6 + 4 + 8 + 2 + 5 \]
  satisfies: \( 0 \leq h(\text{key}) \leq 81 \)
• Grouping digits: \( 001 + 364 + 825 = 1190 \)
  \( 0 \leq h(\text{search key}) \leq 3*999=2997 \)

Hash function: Converting Strings (1/4)

• First step: convert characters to integers (e.g. using ASCII values)
  • Example: "NOTE"

Hash function: Converting Strings (3/4)

• Hashing the sequence of integers:
  – Sum the values representing the characters
  – Write the numeric values in binary and concatenate.
    \[ h(\text{"NOTE"}) = 100111 \ 10100 \ 100101 \ \ 100101 \]
    \( = 78 \ 64 \ 64 \ 64 \ 64 \ 64 \) = 20,776,261
  – Using only 1 through 26 to the letters A through Z
    \[ h(\text{"NOTE"}) = 01110 \ 01111 \ \ 10100 \ \ 00101 \]
    \( = 64 \ 64 \ 64 \ 64 \ 64 \ 64 \) = 474,757

  Can now apply \( x \mod \text{tableSize} \)

Hash function: Converting Strings (4/4)

• Hashing the sequence of integers:
  \( h(\text{"NOTE"}) = 14 \times 32^3 + 15 \times 32^2 + 20 \times 32^1 + 5 \times 32^0 \)
• Overflow can occur for long strings.
• Horner’s Rule:
  – Hash function can be expressed as:
    \[ h(\text{"NOTE"}) = (14 \times 32^2 + 15 \times 32^1 + 20 \times 32^0 + 5) \times 32^0 \]
• Prevent overflow by applying the modulo operation at each step
  \[ A = B \mod N, \text{then for any } C, A + C = B + C \mod N \]
  \[ A = B \mod N, \text{then for any } D, AD = BD \mod N \]
  In practice it is better to use a prime number instead of 32
The Birthday Problem (1/3)

- What is the minimum number of people so that the probability that at least two of them have the same birthday?

- Assumptions:
  - Birthdays are independent
  - Each birthday is equally likely

The Birthday Problem (2/3)

- $p'_n$ – the probability that all people have different birthdays
  
  \[ p'_n = 1 \times (1 - 1/365) \times (1 - 2/365) \times \ldots \times (1 - (n-1)/365) \]

  The event of at least two of the $n$ persons having the same birthday:
  
  \[ p_n = 1 - p'_n \]

The Birthday Problem (3/3)

<table>
<thead>
<tr>
<th>N (# of people)</th>
<th>$p_n$ (Probability that at least two of the $n$ persons have the same birthday)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>11.7 %</td>
</tr>
<tr>
<td>20</td>
<td>41.1 %</td>
</tr>
<tr>
<td>23</td>
<td>50.7 %</td>
</tr>
<tr>
<td>30</td>
<td>70.6 %</td>
</tr>
<tr>
<td>50</td>
<td>97.0 %</td>
</tr>
<tr>
<td>57</td>
<td>99.0%</td>
</tr>
<tr>
<td>100</td>
<td>99.99997%</td>
</tr>
<tr>
<td>200</td>
<td>99.999999999999999999999998%</td>
</tr>
<tr>
<td>366</td>
<td>100%</td>
</tr>
</tbody>
</table>

Probability of Collision

- How many items do you need to have in a hash table so that the probability of collision is greater than $\frac{1}{2}$?

- For a table of size 1,000,000 you only need 1178 items for this to happen!
Methods for Handling Collisions

• Approach 1: Open addressing
  – probe for an empty slot in the hash table

• Approach 2: Restructuring the hash table
  – Change the structure of the array table

Approach 1: Open addressing

• A location in the hash table that is already occupied
  – Probe for some other empty, open, location in which to place the item.
  – Probe sequence
    • The sequence of locations that you examine

Open addressing: Linear Probing (1/3)

• If \( \text{table}[h(key)] \) is occupied check
  \( h(key) + 1, h(key) + 2, \ldots \) until we find an available position

• Retrieval?

• Works until you need to delete.

Open addressing: Linear Probing (2/3)

• Deletion: The empty positions created along a probe sequence could cause the retrieve method to stop, incorrectly indicating failure.

• Resolution: Each position can be in one of three states occupied, empty, or deleted. Retrieve then continue probing when encountering a deleted position. Insert into empty or deleted positions.

Open addressing: Linear Probing (3/3)

• Primary clustering: Items tend to cluster in the hash table.
• Large clusters tend to get larger.
• Decreases the efficiency of hashing.

Open Addressing: Quadratic Probing

• check
  \( h(key) + 1^2, h(key) + 2^2, h(key) + 3^2, \ldots \)

• Eliminates the primary clustering phenomenon

• Secondary clustering: two items that hash to the same location have the same probe sequence
Open Addressing: Double Hashing

Use two hash functions:
- \( h_1(key) \) – determines the position
- \( h_2(key) \) – determines the step size for probing
  - the secondary hash \( h_2 \) needs to satisfy:
    - \( h_2(key) \neq 0 \)
    - \( h_2 \neq h_1 \) (why?)

- Rehashing
  - Using more than one hash functions

Double Hashing

Example:
\[
\begin{align*}
  h_1(key) &= \text{key mod 11} \\
  h_2(key) &= 7 - (\text{key mod 7})
\end{align*}
\]
Insert 58, 14, 91

Open Addressing: Increasing the size

- Increasing the size of the table: as the table fills the likelihood of a collision increases.
  - Cannot simply increase the size of the table – need to run the hash function again

Approach 2: Restructuring the Hash table

- Change the structure of the hash table to resolve collisions.
  - The hash table can accommodate more than one item in the same location

Restructuring: Buckets

- Each location \( \text{table}[i] \) is itself an array called a bucket.
  - Store items that hash into \( \text{table}[i] \) in this array.

- If the bucket size is too small?
  - Collisions will happen soon

- If the bucket size is too large?
  - Waste of storage

Restructuring: Separate Chaining (1/3)

- Separate chaining:
  - Design the hash table as an array of linked lists.
Restructuring: Separate Chaining

• Does not need special care (deleted) for removal as open addressing does

• How do find, add and delete work?

Outline

• Hashing
• Hash Functions
• Resolving Collisions
• Efficiency of Hashing
• Java Hashtable and HashMap

The Efficiency of Hashing

• Consider a hash table with \( n \) items
  – Load factor \( \alpha = \frac{n}{\text{tableSize}} \)
  – \( n \): current number of items in the table
  – \( \text{tableSize} \): maximum size of array
  – \( \alpha \): a measure of how full the hash table is.
    • measures difficulty of finding empty slots

• Efficiency decreases as \( n \) increases

Size of Table

• Determining the size of Hash table
  – Estimate the largest possible \( n \)
  – Select the size of the table to get the load factor small.
  – Load factor should not exceed 2/3.

Hashing: Length of Probe Sequence

• Average number of comparisons that a search requires,
  – Linear Probing
    • successful \( \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right) \)
    • unsuccessful \( \frac{1}{2}\left(\frac{1}{1-\alpha}\right) \)
  – Quadratic Probing and Double Hashing
    • successful \( \frac{\ln(1-\alpha)}{\alpha} \)
    • unsuccessful \( \frac{1}{1-\alpha} \)
**Hashing: Length of Probe Sequence**

- **Chaining**
  - Successful: $1 + \alpha/2$
  - Unsuccessful: $\alpha$
  - Note that $\alpha$ can be $> 1$

---

**Comparison of Collision Resolution Methods**

![Graph showing comparison of successful and unsuccessful searches using different methods like linear probing, quadratic probing, and separate chaining.](image)

- **Successful search**
  - Linear probing
  - Quadratic probing (double hashing)
  - Separate chaining

- **Unsuccessful search**
  - Linear probing
  - Quadratic probing (double hashing)
  - Separate chaining

---

**Good Hash Function?**

- Easy and fast to compute
- Scatter the data evenly
  - Perfect hash function: Impractical to construct
  - Each chain should contain approximately the same number of items

---

**How well does the HF scatter random data?**

- Compare
  - $h(x) = \text{first two digits of } x \mod 40$
  - $h(x) = x \mod 101$

---

**How well does the hash function scatter nonrandom data?**

- Data can have patterns.
- **Example**
  - $h(x) = \text{first two digits of } x$
  - Employee IDs are according to department:
    - 10xxxxx Sales
    - 20xxxxx Customer Relations
    - 90xxxxx Data Processing
  - Only 9 entries will be used (because all of the second digits are 0)
  - Larger departments will have more crowded entries.

---

**Calculation of the hash function should involve the entire search key.**

- Comparing a modulo of the entire ID number is much safer than using only its first two digits.
- If a hash function uses modulo arithmetic, the base should be prime.
  - $H(x) = x \mod \text{tableSize}$
  - tableSize should be a prime number
  - This can avoid subtle types of patterns in the data.
Impact of using prime number

Traversal of Hash Tables

• If you need to traverse your tables by the sorted order of keys – hash tables may not be the appropriate data structure.

Outline

• Hashing
• Hash Functions
• Resolving Collisions
• Efficiency of Hashing
• Java Hashtable and HashMap

Hash Tables in Java

```java
public class Hashtable<K,V> extends Dictionary<K,V> implements Map<K,V>!
public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>!
public HashMap(int initialCapacity, float loadFactor)  // default loadFactor: 0.75

• HashMap is a newer implementation, and is the recommended one to use
```