CS200: Hash Tables

Walls & Mirrors Chapter 13

Table Implementations

Can we build a faster data structure?

<table>
<thead>
<tr>
<th></th>
<th>Search</th>
<th>Add</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorted</td>
<td>(O(\log n))</td>
<td>(O(n))</td>
<td>(O(n))</td>
</tr>
<tr>
<td>Unsorted</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>

Hash Functions and Hash Tables

Magical address calculators exist:
They are called hash functions

Simple Hash Functions

- In our Netflix movie database:
  - We know a user ID is between 0 and 6040
    \(h(x) = x\)
  - Phone exchange: need quick access to record corresponding to phone #.
    \(h(123-4567) = 34567\)

Hash Functions

- 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
  - Want general purpose hash functions!
Collisions

Hash Functions

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

Examples of Hash Functions

- 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:
  \[ 001 + 364 + 825 = 1190 \]

Hashing using modulo arithmetic

\[ h(x) = x \mod \text{tableSize} \]

\text{tableSize} is usually chosen as prime

Hashing Strings

- First step: convert characters to integers (e.g. using ASCII values)
- Hashing the sequence of integers:
  - Sum the values representing the characters (problem?)
  - Write the numeric values in binary and concatenate.
    \[ h(\text{"NOTE"}) = 01110\ 01111\ 10100\ 00101 \]
    \[ = 14 \times 32^3 + 15 \times 32^2 + 20 \times 32^1 + 5 \times 32^0 \]
    \[ = 474,757 \]

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

Hashing Strings

\[ h(\text{"NOTE"}) = 14 \times 32^3 + 15 \times 32^2 + 20 \times 32^1 + 5 \times 32^0 \]

- Overflow can occur for long strings.
- Hash function can be expressed as:
  \[ ((14 \times 32 + 15) \times 32 + 20) \times 32 + 5 \]
- Prevent overflow by applying the modulo operation at each step
  \[ h(x) = B \mod N \]
  \[ A + C = B + C \mod N \]

In practice it is better to use a prime number instead of 32
Collisions:

Collision: two keys map to the same index

The Birthday Problem

- What is the minimum number of people so that the probability that at least two of them have the same birthday is greater than \( \frac{1}{2} \)?
- Assumptions:
  - Birthdays are independent
  - Each birthday is equally likely
- \( p_n \) - the probability that all people have different birthdays

\[
p_n = \frac{365 \times 364 \times \cdots \times (366 - n + 1)}{366^n}
\]

\( n = 23 \rightarrow 1 - p_n \approx 0.506 \)

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

- Open addressing
  - probe for an empty slot in the hash table
- Chaining
  - slot corresponds to collection of data

Linear Probing

- If table[\( h(key) \)] is occupied check
  \( h(key) + 1, h(key) + 2, \ldots \)
  until finding an available position
- Retrieval?
  - Works until you need to delete.

Linear Probing

- 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 continues probing when encountering a deleted position. Insert into empty or deleted positions.
**Linear Probing**

- **Primary clustering**: items tend to cluster in the hash table.
- Large clusters tend to get larger.
- Decreases the efficiency of hashing.

**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

**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?)

**Open Addressing**

- 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 **rehash**
Open addressing

- Deleting
  - Find data
  - Set position to deleted

Chaining

- does not need special care (deleted) for removal as open addressing does
- how do find, add and delete work?

The Efficiency of Hashing

- Consider a hash table with $n$ items
  - Load factor $\alpha = n / \text{tableSize}$
  - measures difficulty of finding empty slots
- For open addressing
  - $\alpha \leq 1$
  - For chaining can have $\alpha > 1$
  - $\alpha$ represents space/time trade off

Hashing: Length of Probe Sequence

- Linear Probing
  - successful: $\frac{1}{2}$\left[ 1 \right. $1 - \alpha$]$
  - unsuccessful: $\frac{1}{2}$\left[ 1 - \alpha \right]$

- Quadratic Probing and Double Hashing
  - successful: $\frac{1}{2}$\left[ 1 \right. $1 - \alpha$]$
  - unsuccessful: $\frac{1}{2}$\left[ 1 - \alpha \right]$

- Chaining
  - successful: $1 + \alpha/2$
  - unsuccessful: $\alpha$
  - Note that $\alpha$ can be $> 1$
Comparison of Collision Resolution Methods

Traversal of Hash Tables

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

Hash Tables in Java

- Java provides hashing method for built-in objects:
  ```java
  public int hashCode();
  ```
- Using it to obtain a hash code:
  ```java
  int code = word.toLowerCase().hashCode();
  code = code % hashSize;
  ```

Application of Hashing: Sequence Alignment

- Alignment of strings of length \( n \) takes time which is \( O(n^2) \)
- If the objective is to determine if two sequences (protein or DNA) share significant level of similarity – no need for full alignment

Hash Tables: Summary

- Data stored in an array
- Location of data determined from key using hash function
- Close to constant time access
  □ Cost: extra space for unused slots