CMPT 225
Hashing – Part 1
Hash Functions
Last Lecture

- Heap Sort
Learning Outcomes

At the end of this unit, a student will be able to:

- define hashing as well as chained and open addressed hash table
- discuss tradeoffs in designing hash functions and between collision resolution strategies
- demonstrate and trace operations on hash table
Our goal in this set of lecture notes is to
- define hashing and hash functions
Fundamental Operations

- So far, we have looked at value-oriented data collection ADT’s that allow us to perform fundamental operations such as insertion, deletion and retrieval in an increasingly “time efficient” fashion.
Overview of value-oriented data collection ADT’s so far

<table>
<thead>
<tr>
<th></th>
<th>Unsorted List</th>
<th>Sorted List</th>
<th>BST</th>
<th>AVL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>insert:</strong></td>
<td>O(1)</td>
<td>O(n)</td>
<td>O((\log_2 n))</td>
<td>O((\log_2 n))</td>
</tr>
<tr>
<td><strong>remove:</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O((\log_2 n))</td>
<td>O((\log_2 n))</td>
</tr>
<tr>
<td><strong>retrieve:</strong></td>
<td>O(n)</td>
<td>O((\log_2 n))</td>
<td>O((\log_2 n))</td>
<td>O((\log_2 n))</td>
</tr>
<tr>
<td><strong>traverse:</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
</tbody>
</table>
Can We Do Better?

• **Question:**
  For example, would it be possible to search for an element and retrieve it in O(1)?

• **Answer:**
So far, array-based implementations elements \([\text{index}]\)

where \(\text{index}\) unrelated to data stored @ index

define array 

```
famousDogs = [0, 1, 2, 3, 4, 5, 6]
```

```
famousDogs[2] = "Snoopy"
```
Introducing *mapping*

- Idea: use part of the element’s data as an “indexing key” into the array-based data collection
  - Allows us to access that element directly [ ]
  - We use array-based underlying data structures to implement our data collection ADT class (as opposed to link-based) because array-based offer “direct access” to their elements (as opposed to “sequential access”)

CMPT 225 - Anne Lavergne
2 Flavours of Mapping

- “1-to-1” map
  - -> indexing key maps to only 1 index in data collection
- “many-to-1” map
  - -> 2 or more distinct indexing keys map to 1 index in data collection
Problem Statement

- School in Spences Bridge wants you to develop a student registration system.
- Students are given unique student #.
- Student # → 3 digits (001-100)
- Max. of 100 students in that school.
Example #1  
Design of data collection ADT

- **Strategy:**
  - Size of “indexing key” space (# of indexing keys) is used to determine size of mapping table

- **Insert, retrieve and remove (i.e., search) -> O(1)**
  - 1-to-1 mapping
  - Mapping: `arrayOfStudents[ ]`
Problem Statement - Take 2

University of British Columbia

School in Vancouver, BC

wants you to develop a student registration system.

- Students are given unique student IDs.
- Student IDs are 3 digits (001-100).
- Estimate of 100 students in that school.
- Max. of 100,000 students.
- Std IDs are not sequentially assigned.
Student Numbers Not Sequentially Assigned?

\[
\begin{align*}
\text{eg:} & \\
01256710 & \rightarrow \text{never assigned} \\
01256711 & \rightarrow \text{assigned 52nd} \\
01256712 & \\
01256713 & \{ \text{assigned later} \} \\
01256714 & \\
01256715 & \\
01256716 & \rightarrow \text{assigned 1st} \\
\vdots & \\
\end{align*}
\]
Example #2
Design of data collection ADT

- If we use the same strategy:
  - Size of “indexing key” space is used to determine size of mapping table

  - 8 digits

  - Array index # 0 to 99,999,999
Problem with this Strategy

**Strategy:**
- Use size of “indexing key” space (# of indexing keys) to determine size of mapping table
  - … cannot be used for large “indexing key” spaces where not all key values are used
Possible Solution to Example #2

- **Strategy:**
  - Size of mapping table now determined by the number of elements expected i.e., number of students -> \( n \)

- **Mapping:**
  - Use hash function that produces array indices in the range of 0 to 99,999
Effect of Hashing

- Reduce the range of array indices hence reducing the size of the mapping table such that, potentially, a greater # of cells end up being occupied
  - less space wasted

- Mapping table now referred to as hash table
Hash Function

- **Definition:**
  - Function that maps (transforms) indexing keys (from a large indexing key space) to hash table indices (smaller hash table index space)
  - The purpose of a hash function is to reduce the size of hash table index space
Characteristics of a Good Hash Function

- A **good** hash function should
  1. Be easy to compute  -> Why?
  2. Give an even distribution of the indexing keys across the range of array indices
Even Distribution of Indexing Keys Across Range of Array Indices

Hash fcn:

indexing key space

hash table (array)
index space
Our goal in creating hash functions is to ...

where $k_1 \neq k_2$

where $i_1 \neq i_2$
Types of Hash Function

- Some common ways of building hash functions:
  1. Modular arithmetic: use modulo operator \( \% \) (or other means)
  2. Folding: partition indexing key into parts and combine these parts using arithmetic operations
  3. Truncation: use only part(s) of indexing key that is unique to this element
  4. Using strings/characters as indexing keys: apply arithmetic operations on the numerical code (Unicode/ASCII) of each character
Example of Modular Arithmetic

Hash (key)
{
    indexing
    hash code = key % array size;
    return hash code;
}

array index within bound
Example of Folding

- **Key**: indexing

- **Diagram**:
  - Credit Card #
  - 16 digits
  - Hash function
  - Array index

- **Example**:
  - Partition credit card # into
    - 6-digit #
    - 2×5-digit #’s
  - Add these 3 #’s together.
2 Types of Folding

1. Shift

2. Boundary
Example of Truncation

- Corp. XYZ has 5000 employees
- Employee’s key: “bpqrcs” where
  - “b” is the branch number (1, 2, or 3)
  - “pqrs” are 4 key digits unique to an employee
  - “c” is a check digit
Example of Truncation

• A simple hash function for XYZ’s hash table may look like:

```c
int hash( int key )
{
    int hashCode;
    hashCode = ( ( key/100 ) % 1000 );
    hashCode = hashCode * 10 + ( key % 10 );
    return ( hashCode % capacity );
}
```
Let’s Give Truncation a Try!
String or Characters-based Indexing Keys

- What is the indexing key is a string?

Problem statement:
Build inventory system of machine parts.

Implementation of soln:

| Element   | Name of Part | Price of Part | Quantity kept in inventory |
Example

Suppose...
- array size → 5000
- keys → STRING of 6 characters
- Hash function:

\[
\text{Alg:} \\
\text{HashCode} = 0 \\
\text{For each char in key} \\
\text{HashCode} + = \text{ASCII value of char.} \\
\text{return HashCode} \mod \text{array size}
\]
How Good Is This Hash Function?

**EXPERIMENTATION**

What is the range of indices?

If use 'A' to 'Z' & 'a' to 'z' then

<table>
<thead>
<tr>
<th>Key</th>
<th>Range of indices</th>
</tr>
</thead>
<tbody>
<tr>
<td>AAAAAA to ZZZZZZ</td>
<td>390 to 540</td>
</tr>
<tr>
<td>aaaaa2 to zzzzzz</td>
<td>582 to 732</td>
</tr>
</tbody>
</table>
How Good Is This Hash Function?

It does not spread resulting array-indices (various values of hash code) well (uniformly) over array-index range.

Observation #1

Array indices

0 to 540
582 to 782

OBSERVATION

0

4999
How Good Is This Hash Function?

"Easy to compute"

- Even though simple, hash function will map many-to-1.

Eg: key ABAAAAA will produce same HashIndex as key AABAAAA
Improving Our Hash Function

**How can we modify our hash function so that array indices are more uniformly spread?**

**One Possibility:**

\[
\text{HashCode} = 2 \times \text{HashCode} + \text{Ascii value of char}
\]
In General: Hash Functions Are ...

- 2 – step process:

  - **STEP 1**: Key -> # -> valid index in Hash Table
    - **Folding**
    - **Extraction ( truncation ) of unique part of key**
    - **String**

  - **STEP 2**: % size of hash table
    - Purpose: produce array indices that are within valid range of valid array indices.
Problem with Hashing?

-> Collision

• **Definition**: Collision occurs when **multiple distinct** indexing keys are hashed to the same location in the hash table (i.e. the same hash table index is produced for each of these distinct indexing keys)

• These multiple distinct indexing keys are called **synonyms**
Two factors that may minimize the number of collisions are:
- Goodness of hash function
- Size of the table

but they cannot completely eliminate them
Summary

- Defined hashing and hash functions
Next Lecture

- Hashing – Part 2 – Collision Resolution Strategies