CMPT 225

Lectures 28 and 29 – Hashing – Part 1 – Hash Functions
Last Lectures

- We saw how to...
  - Understand how heap sort works
  - Sort an array using heap sort
  - Analyze time/space efficiency of heap sort
Learning Outcomes

- At the end of these lectures, 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
Today’s menu

- Our goal is to
  - define hashing and hash functions
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>

<table>
<thead>
<tr>
<th></th>
<th>Average/Best case scenario</th>
<th>Worst case scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted List</td>
<td>$O(n)$</td>
<td></td>
</tr>
<tr>
<td>Sorted List</td>
<td>$O(\log_2 n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>BST</td>
<td>$O(n)$</td>
<td>$O(\log_2 n)$</td>
</tr>
<tr>
<td>AVL</td>
<td>$O(\log_2 n)$</td>
<td>$O(\log_2 n)$</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 index unrelated to data stored @ index

e.g.: famousDogs $[2]$ → Snoopy with master Charlie Brown

famousDogs:

- Milou
- Tintin
- Laika
- Spoutnik
- Snoopy
- Charlie Brown

Name of famous dog
Owner of famous dog
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 using [ ]
- We use an array as the underlying data structure when we implement our data collection ADT class (as opposed to link-based) because arrays offer “direct access” to their elements (as opposed to “sequential access”)
2 flavours of mapping

<table>
<thead>
<tr>
<th>Indexing Keys</th>
<th>Mapping Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range of indexing keys</td>
<td>Range of array indices</td>
</tr>
</tbody>
</table>

- **“1-to-1” map**
  - \(1 \times 1^\) map
  - Indexing key maps to only 1 index in data collection

- **“many-to-1” map**
  - \(\text{many} \times 1\) 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

<table>
<thead>
<tr>
<th>Indexing Keys</th>
<th>Mapping</th>
<th>Mapping Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student #</td>
<td></td>
<td>Array index #</td>
</tr>
<tr>
<td>001 to 100 (3 digits)</td>
<td></td>
<td>0 to 99</td>
</tr>
</tbody>
</table>

**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: \texttt{arrayOfStudents}[
      
  ]
Problem Statement  Take 2

• University in Burnaby wants you to develop a student registration system.

• Students are given unique student #.

• Student # → 8 digits
  estimate 1,000,000
  max. of 100 students in that school.

• Student numbers are not sequentially assigned
Student numbers not sequentially assigned?

Example:

01256710  (never assigned)
01256711  (assigned 52nd)
01256712
01256713
01256714
01256715
01256716  (assigned 1st)

...
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

<table>
<thead>
<tr>
<th>Indexing Keys</th>
<th>Mapping</th>
<th>Mapping Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student # 8 digits</td>
<td></td>
<td>Array index # 0 to 99,999,999</td>
</tr>
</tbody>
</table>
Problem with this strategy

The strategy is to:

- Use size of “indexing key” space (# of indexing keys) to determine size of mapping table

So, it 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 $\rightarrow n$

- **Mapping:**
  - Use hash function that produces array indices in the range of 0 to 999,999
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
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)
{
    hash code = key % array size;
    return hash code;
}
```

array index within bound
Example of folding

**indexing**

**Key**

**CREDIT CARD #**

16 digits

**ARRAY INDEX**

**HASH FN**

**Eg:**
- 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 solution:

Element
- Name of part
- Price of part
- Quantity kept in inventory.
Example

Suppose...

- array size → 5000
- keys → string of 6 characters
- Hash function:

\[
\text{Alg}:
\]

Hash Code = 0
For each char in key
Hash Code += ASCII value of char.
return Hash Code % array size

max 5000 parts ever?
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>aaaaaa to zzzzzz</td>
<td>582 to 732</td>
</tr>
</tbody>
</table>
How good is this Hash function?

Observation #1

It does not spread resulting array-indices (various values of hash code) well (uniformly) over array-index range.
How good is this Hash function?

“Easy to compute”

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

Eg: key ABAAAA 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:

A

\[ \text{HashCode} = 2 \times \text{HashCode} + \text{Ascii value of char} \]
In general: Hash functions are ...

- 2-step process:

  1. Key -> # -> valid index in Hash Table

  2. **Purpose**: produce array indices that are within valid range

     - Folding
     - Extraction (truncation) of unique part of key
     - String

     \[ \text{Intermediate hash code} \times 2 \text{– step process} \]
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*
Minimizing collisions

- Two factors that may minimize the number of collisions are:
  - Goodness of hash function
  - Size of the table

but they cannot completely eliminate them.
√ Learning Check

- We can now ...
  - define hashing and hash functions
Next Lectures

- Hashing – Part 2 – Collision Resolution Strategies