CSCI 104
Hash Tables Intro

Mark Redekopp
David Kempe
Reviewed for Fall 2022
Motivation

Suppose a company has a unique 3-digit ID for each of its 1000 employees.

• We want a data structure that, when given an employee ID, efficiently brings up that employee’s record.

How should we implement this?

• An array gives $O(1)$ access time!

Alright, how do we obtain this runtime when the keys are no longer so nicely ordered or non-integers??
Maps/Dictionaries

Arrays

• An array maps **integers** to **values**
  
  – Given i, array[i] returns the value in O(1)

Maps/Dictionaries

• Dictionaries map **keys** to **values**
  
  – Given key, k, map[k] returns the associated value
  
  – Key can be anything provided...
    
    • It has a '<' operator defined for it (C++ map) or some other comparator functor (other languages require something similar)

Arrays associate an integer with some arbitrary type as the value (i.e. the key is always an integer)
Dictionary Implementation

• A dictionary/map can be implemented with a balanced BST
  – Insert, Find, Remove = O(_________)

• Can we do better?
  – Hash tables (unordered maps) offer the promise of O(____) access time
Hash Tables - Insert

• Can we use non-integer keys to index an array?
  • Yes. Let us convert (i.e. "hash") the non-integer key to an integer

• To **insert** a key, we hash it and place the key (and value) at that index in the array
  – For now, make the unrealistic assumption that each unique key hashes to a unique integer

• The conversion function is known as a **hash function, \( h(k) \)**

• A hash table implements a **set/map ADT**
  – insert(key) / insert(key,value)
  – remove(key)
  – lookup/find(key) => value

• **Question to address:** What should we do if two keys ("Jill" and "Erin") hash to the same location (aka a COLLISION)?

![Insertion Example](image)

**A map implemented as a hash table**
(key=name, value = GPA)

**Hash table parameter definitions:**

\[
\begin{align*}
\text{n} &= \# \text{ of keys entered (}=4 \text{ above}) \\
\text{m} &= \text{tableSize (}=6 \text{ above}) \\
\alpha &= \frac{n}{m} = \text{Loading factor} = (4/6 \text{ above})
\end{align*}
\]
Hash Tables - Find

- To **find** a key, we simply hash it again to find the index where it was inserted and access it in the array.

- How might we hash a string to an integer?
  - Use ASCII codes for each character and **add, multiply, or shift/mix** them.
  - We then can use simple a **modulo** $m$ operation to convert the sum to a value between $0$ to $m-1$ where $m$ is the table size.
  - Note: All data in a computer is already bits (1s and 0s). Any object can be viewed as a long binary number and hashed.

**We could sum the ASCII values.**

\[
\begin{align*}
'h' &= 104 \\
'e' &= 101 \\
'l' &= 108 \\
'o' &= 111 \\
\end{align*}
\]

\[
h("hello") = 532 \mod m
\]

Is this a good way to hash a string?
Hash Tables - Remove

- To **remove** a key, we simply hash the key and mark the location as "free" again
  - Could use a `bool` in the struct for each array entry (more later) to indicate it is free

- The **hash function, \( h(k) \), should**
  - Be **fast/easy** to compute
    - \( O(|k|) \) – where \( |k| \) is the length of the key
    - But in terms of \( n \) [# of keys in the set/map] this runtime is constant since \( |k| \ll n \) [e.g. \( O(1) \)]
  - Be **consistent** and output the same result any time it is given the same input
  - **Distribute** keys well
    - We'd like every unique key to map to a different index, but that turns out to be almost impossible.
    - We'll settle for a "**good**" hash function where the probability of a key mapping to any location \( x \) is \( 1/m \) (i.e. uniform)

**Hash table parameter definitions:**

\[
\begin{align*}
\text{n} &= \text{# of keys entered} \\
\text{m} &= \text{tableSize} \\
\alpha &= \frac{n}{m} = \text{Loading factor}
\end{align*}
\]
Possible Hash Functions

• Define $n = \# \text{ of keys stored}$, $m = \text{table size}$ and suppose $k$ is non-negative integer key

• Evaluate the following possible hash functions
  • $h(k) = 0$ ?
  • $h(k) = \text{rand()} \mod m$ ?
  • $h(k) = k \mod m$ ?

• Rules of thumb
  – The hash function should examine the entire search key (i.e. all bits/characters), not just a few digits or a portion of the key
  – When modulo hashing is used, the base should be prime
Hashing Efficiency

- If computing the hash function, $h(k)$, is $O(1)$ and the array access is $O(1)$,
- Then the runtime of the operations is $O(1)$
- What might prevent us from achieving this $O(1)$?
  - Collisions
**Ordered vs. Unordered**

**Ordered Map/Set**
- `map/set` *(implemented as balanced BST)*
- Log(n) runtime for insert/find/remove
- If we print each key via an in-order traversal of the tree, in what order will the keys be printed?

```
<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Jordan&quot;</td>
<td>Student object</td>
</tr>
<tr>
<td>&quot;Frank&quot;</td>
<td>Student object</td>
</tr>
<tr>
<td>&quot;Percy&quot;</td>
<td>Student object</td>
</tr>
<tr>
<td>&quot;Anne&quot;</td>
<td>Student object</td>
</tr>
<tr>
<td>&quot;Greg&quot;</td>
<td>Student object</td>
</tr>
<tr>
<td>&quot;Tommy&quot;</td>
<td>Student object</td>
</tr>
</tbody>
</table>
```

**Unordered Map/Set**
- `unordered_map/unordered_set` *(implemented as hash table)*
- Each uses a hash table for O(1) average runtime to insert, find, and remove
- New to C++11 and requires compilation with the `-std=c++11` option in g++
- Iteration will print the keys in an undefined order (unordered)
- Provides hash functions for basic types: int, string, etc. but for any other type you must provide your own hash function (like the operator `<` for BSTs)
Table Size and Collisions

- Suppose we want to store USC student info using their 10-digit USC ID as the key
  - The set of all POSSIBLE keys, S, has size $|S| = 10^{10}$
  - But the number of keys we'd actually store, $n$, is likely much less (i.e. $n \ll |S|$)
- So how large should the table size ($m$) be?
  
  \[
  \_\_\_\_\_\_ < \_\_\_\_\_\_\_\_\_\_\_\_ < \_\_\_\_\_\_\_\_\_\_\_\_\_\_
  \]
- But anything smaller than the size of all possible keys admits the chance of COLLISION
  - A collision is when two keys map to the same location [i.e. $h(k1) == h(k2)$]
  - The probability of this should be low
  - How we handle collisions is the major remaining question to answer
- You will see that table size ($m$) should usually be a prime number

```latex
\text{Conversion / Hash function}
\begin{array}{|c|c|c|c|c|c|}
\hline
& Bo & Ann & Jill & & Tim \\
\hline
conversion & 2.7 & 3.5 & 3.7 & & 3.8 \\
\hline
\end{array}
```

```
insert("Erin",3.2)
```

\begin{itemize}
  \item COLLISION!!
  \item $h("Jill") = h("Erin")$
\end{itemize}
Resolving Collisions

• Collisions occur when two keys, k1 and k2, are not equal, but h(k1) = h(k2).

• Collisions are inevitable if the number of entries, \( n \), is greater than table size, \( m \) (by pigeonhole principle) and are likely even if \( n < m \) (by the birthday paradox...more in our probability unit)

• Methods
  – Closed Addressing (e.g. buckets or chaining): Keys MUST live in the location they hash to (thus requiring multiple locations at each hash table index)
    • Methods: 1.) Buckets, 2.) Chaining
  – Open Addressing (aka probing): Keys MAY NOT live in the location they hash to (only requiring a single 1D array as the hash table)
    • Methods: 1.) Linear Probing, 2.) Quadratic Probing, 3.) Double-hashing
Closed Addressing Methods

• Make each entry in the table a fixed-size ARRAY (bucket) or LINKED LIST (chain) of items/entries so all keys that hash to a location can reside at that index
  – Close Addressing => A key will reside in the location it hashes to (it's just that there may be many keys (and values) stored at that location

• Buckets
  – How big should you make each array?
  – Too much wasted space

• Chaining
  – Each entry is a linked list (or, potentially, vector)
Open Addressing and Linear Probing

- With open addressing, we keep the hash table a 1D array (only one location per index) but when collisions occur we allow keys to reside in a location other than $h(k)$
  - **Open Addressing** => A key may NOT reside in the location it hashes to requiring extra searching in a process called **probing**
- For insertion: always start by checking location $h(k)$
  - If it is open, write the key (and value) there
  - Else "**probe**" for an empty location
- **Linear Probing (other techniques in a minute)**
  - Let $i$ be number of failed checks to find a blank location (for insertion) or the key we are looking (for find/remove)
  - $h(k,i) = (h(k)+i) \mod m$
  - Example: If $h(k)$ occupied (i.e. collision) then check $h(k)+1, h(k)+2, h(k)+3, ...$
Probing Impact on Find

- If h(k) is occupied with another key, then probe
- **Insert**: probe until we find a blank location
- **Find/Remove**: probe until we...
  - Find the key we are looking for ..OR..
  - _______________________________ ..OR..
  - _______________________________
Probing Impact on Find

- If \( h(k) \) is occupied with another key, then probe
- **Insert**: probe until we find a blank location
- **Find/Remove**: probe until we...
  - Find the key we are looking for **.OR.**
  - We reach a free location **.OR.**
  - We have looked in all possible locations (i.e. wrapped back to \( h(k) \) or alternatively we've performed \( m \) probes)
Removal

- Many implementations exist but we will show one simple way for illustration
- Each location stores two bools
  - Valid: a stored key exists in this location (or else is free)
  - Removed: a key was erased at this location (so it is free for insertion, but probing must continue for find/remove)
- Progression:
  - Initially: \( V=0, R=0 \) (Free/Never used),
  - On insert: \( V=1, R=0 \),
  - On erasure: \( V=0, R=1 \) (can return to \( V=1, R=0 \) on insert)
- For performance, we can periodically rebuild/rehash the hash table after some number of erasures to effectively return locations to free/never used
Linear Probing & Primary Clustering

- Suppose a hash table \((m=10)\) with integer keys and \(h(k) = k \mod m\)

- Insert: 11, 21, 2, 31, 3
  - Notice, that the collisions of 11, 21, and 31 cause collisions for 2 and 3 which then may cause collisions for other nearby hash locations

- This is known as primary clustering (a few collisions to one location and the resulting probing cause collisions for other keys that would not have collided)
Quadratic Probing

- If certain data patterns lead to many collisions, linear probing leads to clusters of occupied areas in the table called *primary clustering*.

- **Quadratic probing** tends to spread out data across the table by taking larger and larger steps until it finds an empty location.

- **Quadratic Probing**
  - (Again, let $i$ be number of *failed* probes)
  - $h(k,i) = (h(k)+i^2) \mod m$
  - If $h(k)$ occupied, then check $h(k)+1^2$, $h(k)+2^2$, $h(k)+3^2$, ...
Linear vs. Quadratic Probing

- If certain data patterns lead to many collisions, linear probing leads to clusters of occupied areas in the table called **primary clustering**
- How would quadratic probing help fight primary clustering?
  - Quadratic probing tends to spread out data across the table by taking larger and larger steps until it finds an empty location
Quadratic Probing Practice

• Use the hash function $h(k) = k \% 9$ to find the contents of a hash table ($m=9$) after inserting keys 36, 27, 18, 9, 0 using quadratic probing

• If your **loading factor** rises above 0.5, bad things can happen!

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Use the hash function $h(k) = k \% 7$ to find the contents of a hash table ($m=10$) after inserting keys 14, 8, 21, 2, 7 using quadratic probing

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Quadratic probing only works well for prime table sizes, and keeping the load factor < 0.5
Double Hashing

• Note: In linear and quadratic probing, if two keys hash to the same place \( h_1(k_1) == h_1(k_2) \) we will probe the same sequence

• Could we probe a different sequence even if two keys have collided?
  - Let's use ANOTHER hash function, \( h_2(k) \) to choose the step size of our probing sequence

• Double Hashing
  - (Again, let \( i \) be number of failed probes)
  - Pick a second hash function \( h_2(k) \) in addition to the primary hash function, \( h_1(k) \)
  - \( h(k,i) = [ h_1(k) + i*h_2(k) ] \mod m \)

Sequence:
  - Start at \( h_1(k) \),
  - If needed, probe \( h_1(k) + h_2(k) \)
  - If needed, probe \( h_1(k) + 2*h_2(k) \)
  - If needed, probe \( h_1(k) + 3*h_2(k) \)
Double Hashing

• Assume
  – \( m=13 \),
  – \( h_1(k) = k \% 13 \)
  – \( h_2(k) = 5 - (k \% 5) \)

• What sequence would I probe if \( k = 31 \)
  – \( h_1(31) = \), \( h_2(31) = \)

– Seq: 

– Notice we \( \) in the table. Why? A _____
  table size!
Double Hashing

- Assume
  - $m=13$,
  - $h_1(k) = k \% 13$
  - $h_2(k) = 5 - (k \% 5)$

- What sequence would I probe if $k = 31$
  - $h_1(31) = 5$
  - $h_2(31) = 5 - (31 \% 5) = 4$ (which is the step size)
  - $5 + 0*4 = 5 \% 13 = 5$
  - $5 + 1*4 = 9 \% 13 = 9$
  - $5 + 2*4 = 13 \% 13 = 0$
  - $5 + 3*4 = 17 \% 13 = 4$
  - And then onto 8, 12, 3, 7, 11, 2, 6, 10, 1
  - Notice we visited each index in the table. Why? A prime table size!
Rehashing

• For probing (open-addressing), as $\alpha$ approaches 1 the expected number of probes/comparisons will get very large
  – Capped at the tableSize, $m$ (i.e. $O(m)$)

• Similar to resizing a vector, we can allocate a larger prime size table/array
  – Must **rehash** items to location in new table size and **cannot just copy**
    items to corresponding location in the new array
  – Example: $h(k) = k \% 7 \neq h(k) = k \% 11$ (e.g. $k=9$)
    – For quadratic probing if table size $m$ is prime, then first $m/2$ probes
      will go to unique locations

• **General guideline for probing:** keep $\alpha < ____$

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>9</td>
<td>38</td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

$h(k) = k \% 7$  
$h(k) = k \% 11$
Rehashing

• For probing (open-addressing), as \( \alpha \) approaches 1 the expected number of probes/comparisons will get very large
  – Capped at the tableSize, \( m \) (i.e. \( \mathcal{O}(m) \))

• Similar to resizing a vector, we can allocate a larger prime size table/array
  – Must **rehash** items to location in new table size and **cannot just copy** items to corresponding location in the new array
  – Example: \( h(k) = k \mod 7 \) != \( h(k) = k \mod 11 \) (e.g. \( k=9 \))
  – For quadratic probing if table size \( m \) is prime, then first \( m/2 \) probes will go to unique locations

• **General guideline for probing:** keep \( \alpha < 0.5 \)

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|   | 1 | 9 | 38 | 18 |   |   |   |   | 1 |   |   |   | 38 | 18 |   |   |   |   |   |

\[ h(k) = k \mod 7 \] \hspace{2cm} \[ h(k) = k \mod 11 \]
Probing Technique Summary

- If \( h(k) \) is occupied with another key, then probe
- Let \( i \) be number of failed probes
- Linear Probing
  - \( h(k,i) = (h(k)+i) \mod m \)
- Quadratic Probing
  - \( h(k,i) = (h(k)+i^2) \mod m \)
  - If \( h(k) \) occupied, then check \( h(k)+1^2, h(k)+2^2, h(k)+3^2, \ldots \)
- Double Hashing
  - Pick a second hash function \( h_2(k) \) in addition to the primary hash function, \( h_1(k) \)
  - \( h(k,i) = [ h_1(k) + i*h_2(k) ] \mod m \)
Hash Function Goals

• A "perfect hash function" should map each of the $n$ keys to a unique location in the table
  – Recall that we will size our table to be larger than the expected number of keys...i.e. $n < m$
  – Perfect hash functions are not practically attainable

• A "good" hash function
  – Is easy and fast to compute
  – Scatters data uniformly throughout the hash table
    • $P( h(k) = x ) = 1/m$ (i.e. pseudorandom)
Hashing Efficiency

• Loading factor, $\alpha$, defined as:
  – $\alpha = \frac{n}{m}$ (Really it is just the fraction of locations currently occupied)
  – $n$=number of items in the table, $m$=tableSize

• For open addressing, $\alpha \leq 1$
  – Good rule of thumb: resize and rehash after $\alpha > 0.5$

• For closed addressing (chaining), $\alpha$, can be greater than 1
  – This is because $n > m$
  – What is the average length of a chain in the table (e.g. 10 total items in a hash table with table size of 5)?
    – Need to keep $\alpha$ constant (usually $\alpha \leq 1$)
Hashing Efficiency

• Loading factor, $\alpha$, defined as:
  – $\alpha = \frac{n}{m}$ (Really it is just the fraction of locations currently occupied)
  – $n$ = number of items in the table, $m$ = tableSize

• For open addressing, $\alpha \leq 1$
  – Good rule of thumb: resize and rehash after $\alpha > 0.5$

• For closed addressing (chaining), $\alpha$, can be greater than 1
  – This is because $n > m$
  – What is the average length of a chain in the table (e.g. 10 total items in a hash table with table size of 5)?
    • Average length of chain will be $\alpha = \frac{n}{m}$
  – Need to keep $\alpha$ constant (usually $\alpha \leq 1$)
Hash Tables are Awesome!

Hash tables provide a very lucrative potential runtime. However, they are **probabilistic**.

- There was a similar problem with Splay Trees: they had a good **average** runtime, but a poor **worst**-case runtime.

As of this moment, we do not have the necessary mathematical framework to analyze either of these structures.

- We’re going to start remedying that... now.