Set and Dictionary ADTs: Hash Tables
CSE 332 Spring 2020

Instructor: Hannah C. Tang

Teaching Assistants:
Annie Mao  Howard Xiao  Lucy Jiang
Chris Choi  Jade Watkins  Nathan Lipiarski
Ethan Knutson  Khushi Chaudhari  Richard Jiang
Announcements

❖ Exercises 6 and 7 due tonight

❖ P2 Checkpoint due tomorrow

❖ Tell us how Quiz #2 went: cse332-staff@cs or anonymous feedback

❖ As usual: questions at pollev.com/cse332
Learning Objectives

❖ Understand how to use *hashing* to implement *hash tables*

❖ Differentiate between collision *avoidance* and collision *resolution*

❖ Describe the difference between the major collision resolution strategies

❖ Implement Dictionary ADT operations for a separate-chaining hash table and an open-addressing linear-probing hash table
Lecture Outline

❖ Hash Table Introduction
  ▪ Collision *Avoidance* Concepts

❖ Collision Resolution: Separate Chaining

❖ Collision Resolution: Open Addressing
  ▪ Linear Probing
Review: Set and Dictionary Data Structures

- We’ve seen several implementations of the Set or Dictionary ADT

- Search Trees give good performance – log \( N \) – as long as the tree is reasonably balanced
  - Which doesn’t occur with sorted or mostly-sorted input
  - So we studied two categories of search trees whose heights are bounded:
    - **B-Trees** (eg, B+ Trees) which grow from the root and are “mostly full” M-ary trees
    - **Balanced BSTs** (eg, AVL Trees) which grow from the leaves but rotate to stay balanced

<table>
<thead>
<tr>
<th>Data Structure</th>
<th>Find</th>
<th>Add</th>
<th>Remove</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinkedList Dict</td>
<td>( \Theta(N) )</td>
<td>( \Theta(N) )</td>
<td>( \Theta(N) )</td>
</tr>
<tr>
<td>BST Dict</td>
<td>( h = \Theta(N) )</td>
<td>( h = \Theta(N) )</td>
<td>( h = \Theta(N) )</td>
</tr>
<tr>
<td>AVL Tree Dict</td>
<td>( h = \Theta(\log N) )</td>
<td>( h = \Theta(\log N) )</td>
<td>( h = \Theta(\log N) )</td>
</tr>
<tr>
<td>B+ Tree Dict</td>
<td>( h = \Theta(\log N) )</td>
<td>( h = \Theta(\log N) )</td>
<td>( h = \Theta(\log N) )</td>
</tr>
</tbody>
</table>
Hash Table: Idea (1 of 2)

- Thanks to hashing, we can convert objects to large integers

- Hash tables can use these integers as array indices

```java
HashTable h;
h.add("cat", 100);
h.add("bee", 50);
h.add("dog", 200);

hashFunction("cat") == 2;
hashFunction("bee") == 2525393088;
hashFunction("dog") == 9752423;
```
Hash Table: Idea (2 of 2)

- We can convert objects to large integers

- Hash Tables use these integers as array indices
  - To force our numbers to fit into a reasonably-sized array, we’ll use the modulo operator (%)

```java
HashTable h;
h.add("cat", 100);
h.add("bee", 50);
h.add("dog", 200);

hashFunction("cat") == 2;
2 % 5 == 2

hashFunction("bee") == 2525393088;
2525393088 % 5 == 3

hashFunction("dog") == 9752423;
9752423 % 5 == 3
```
How should we handle the “bee” and “dog” collision at index 3?

A. Somehow force “bee” and “dog” to share the same index
B. Overwrite “bee” with “dog”
C. Keep “bee” and ignore “dog”
D. Put “dog” in a different index, and somehow remember/find it later
E. Rebuild the hash table with a different size and/or hash function
F. I’m not sure ...
Implementing a hash table requires the following components:

```
HashTable h;
h.add("cat", 100);
h.add("bee", 50);
```

- `hashFunction("cat") == 2;`  
- `2 % 5 == 2`
- `hashFunction("bee") == 2525393088;`  
- `2525393088 % 5 == 3`
A Note on Terminology

- We and the book use the terms
  - “chaining” or “separate chaining”
  - “open addressing”

- Very confusingly
  - “open hashing” is a synonym for “chaining”
  - “closed hashing” is a synonym for “open addressing”

Reminder: a dictionary maps keys to values; an item or data refers to the (key, value) pair
Lecture Outline

❖ Hash Table Introduction
  ▪ Collision Avoidance Concepts

❖ Collision Resolution: Separate Chaining

❖ Collision Resolution: Open Addressing
  ▪ Linear Probing
Key Space vs Value Space vs Table Size

- There are \( m \) possible keys
  - \( m \) typically large, even infinite
- A hash function will map those keys into a large set of integers
- We expect our table to have only \( n \) items
  - \( n \) is much less than \( m \) (often written \( n << m \))

- Many dictionaries have this property
  - Database: All possible student names vs. students enrolled
  - AI: All possible chess-board configurations vs. those considered by the current player
  - ...

- ...
Collision Avoidance: Hash Function Input

- As usual: our examples use int or string keys, and omit values

- If you have aggregate/structured objects with multiple fields, you want to hash the “identifying fields” to avoid collisions
  - Hashing just the first name = bad idea
  - Hashing everything = too granular? Too slow?

```java
class Person {
    String first; String middle; String last;
    Date birthdate;
}
```

- As we saw earlier, the hard part is deciding what to hash
  - The how to hash is easy: we can usually use “canned” hash functions
Collision Avoidance: Table Size (1 of 3)

- With “x % TableSize”, the number of collisions depends on
  - the keys inserted (see previous slide)
  - the quality of our hash function (don’t write your own)
  - TableSize

- Larger table-size tends to help, but not always!
  - Eg: 70, 24, 56, 43, 10 with TableSize = 10 and TableSize = 60

- Technique: Pick table size to be prime. Why?
  - Real-life data tends to have a pattern
  - “Multiples of 61” are probably less likely than “multiples of 60”
  - Some collision resolution strategies do better with prime size
Collision Avoidance: Table Size (2 of 3)

- Examples of why prime table sizes help:

- If $\text{TableSize}$ is 60 and...
  - Lots of keys hash to multiples of 5, we waste 80% of table
  - Lots of keys hash to multiples of 10, we waste 90% of table
  - Lots of keys hash to multiples of 2, we waste 50% of table

- If $\text{TableSize}$ is 61...
  - Collisions can still happen, but multiples of 5 will fill table
  - Collisions can still happen, but multiples of 10 will fill table
  - Collisions can still happen, but multiples of 2 will fill table
Collision Avoidance: Table Size (3 of 3)

❖ If \( x \) and \( y \) are “co-prime” (means \( \gcd(x, y) == 1 \)), then
\[
(a \times x) \mod y == (b \times x) \mod y \quad \text{iff} \quad a \mod y == b \mod y
\]

❖ Given table size \( y \) and key hashes as multiples of \( x \), we’ll get a decent distribution if \( x \) & \( y \) are co-prime

- So choose a TableSize that has no common factors with any “likely pattern” \( x \)
- And choose a decent hash function
Lecture Outline

❖ Hash Table Introduction
  ▪ Collision *Avoidance* Concepts

❖ Collision Resolution: Separate Chaining

❖ Collision Resolution: Open Addressing
  ▪ Linear Probing
Separate Chaining Idea

❖ All keys that map to the same table location are kept in a list
  ▪ (a.k.a. a “chain” or “bucket”)

```
HashTable h;
h.add("cat", 100);
h.add("bee", 50);
h.add("dog", 200);
```

```
hashFunction("cat") == 2;
2 % 5 == 2
hashFunction("bee") == 2525393088;
2525393088 % 5 == 3
hashFunction("dog") == 9752423;
9752423 % 5 == 3
```
Separate Chaining: Add Example

- Add 10, 22, 107, 12, 42
  - Let hashFunction(x) = x
  - Let TableSize = 10

<table>
<thead>
<tr>
<th>K</th>
<th>hashFunction</th>
<th>int</th>
<th>%</th>
<th>table-index</th>
<th>collision?</th>
<th>resolved table-index</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td>10</td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td>22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td>12</td>
<td></td>
<td></td>
</tr>
<tr>
<td>42</td>
<td></td>
<td></td>
<td></td>
<td>42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>107</td>
<td></td>
<td></td>
<td></td>
<td>107</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Separate Chaining: Find

- You can probably figure this one out on your own
Separate Chaining: Remove

- Not too bad!
  - Find in table
  - Delete from bucket

- Example: remove 12

- What are the runtimes of these operations (add, find, remove)?
Separate Chaining Runtime: Load Factor

- The load factor $\lambda$, of a hash table is

$$\lambda = \frac{N}{\text{TableSize}}$$

$\leftarrow$ number of elements
Load Factor: Example

\[
\lambda = \frac{\text{number of elements}}{\text{number of slots}}
\]

- \( \lambda = 0.5 \) with 5 elements and 10 slots
- \( \lambda = 1.4 \) with 14 elements and 10 slots
Separate Chaining Runtime: Cases

- Under separate chaining:
  - The average number of elements per bucket is: \( \lceil \sqrt{N} \rceil \)
  - If we have some *random* inserts are followed by *random* finds, then:
    - How many keys does each *unsuccessful* find compare against? \( \lceil \sqrt{N} \rceil \)
    - How many keys does each *successful* find compare against? \( \lceil N/2 \rceil \)
  - If we have a sequence of *worst-case* adds, then:
    - What is the runtime of the next add? \( O(1) \)
    - What is the runtime of find? \( O(n) \)
    - What is the runtime of the next remove? \( O(n) \)

- How big should TableSize be??
Separate Chaining Optimizations

- **Worst-case asymptotic runtime**
  - Only happens with really bad luck or bad hash function
  - Generally not worth avoiding (e.g., with balanced trees in each bucket)
    - Keep # of items in each bucket small
    - Overhead of AVL tree, etc. not worth it for small $n$

- Some simple modifications can improve constant factors
  - Linked list vs. array vs. a hybrid of the two
  - Move-to-front (part of Project 2)
  - Leave room for 1 element (or 2?) in the table itself, to optimize constant factors for the common case
    - A time-space trade-off...
A Time vs. Space Optimization

(only makes a difference in constant factors)

```
0  -  10
1  -  -
2  -  22  12  42
3  -  -
4  -  -
5  -  -
6  -  -
7  -  107
8  -  -
9  -  -
```

```
0  10  -
1  -  -
2  22  -  12  42
3  -  -
4  -  -
5  -  -
6  -  -
7  107  -
8  -  -
9  -  -
```
Lecture Outline

❖ Hash Table Introduction
  ▪ Collision Avoidance Concepts

❖ Collision Resolution: Separate Chaining
  - Linked List
  - Balanced trees

❖ Collision Resolution: Open Addressing
  ▪ Linear Probing
Open Addressing Idea

- Why not use up the empty space in the table?
  - Store directly in the array cell (no linked list)

- How to deal with collisions?
  - If $h(\text{key}) \% \text{TableSize}$ is already full, ...

```plaintext
HashTable h;
h.add("cat", 100);
h.add("bee", 50);
h.add("dog", 200);
```

```plaintext
hashFunction("cat") == 2;
2 % 5 == 2
hashFunction("bee") == 2525393088;
2525393088 % 5 == 3
hashFunction("dog") == 9752423;
9752423 % 5 == 3
```
Linear Probing: Add Example

- Our first option for resolving this collision is **linear probing**

- If \( h(\text{key}) \) is already full,
  - try \( (h(\text{key}) + 1) \mod \text{TableSize} \). If full,
  - try \( (h(\text{key}) + 2) \mod \text{TableSize} \). If full,
  - try \( (h(\text{key}) + 3) \mod \text{TableSize} \). If full...

- Example: add 38, 19, 8, 109, 10

<table>
<thead>
<tr>
<th>K</th>
<th>hashFunction</th>
<th>int</th>
<th>%</th>
<th>table-index</th>
<th>collision?</th>
<th>resolved table-index</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

0 8
1 109
2 10
3 
4 
5 
6 
7 
8 38
9 19
Open Addressing

- **Open addressing** resolves collisions by trying a sequence of other positions in the table
  - Trying the *next* spot is called **probing**
  - We just did linear probing:
    - $i^{th}$ probe: $(h(key) + i) \mod \text{TableSize}$
  - In general have some **probe function** $f$ and:
    - $i^{th}$ probe: $(h(key) + f(i)) \mod \text{TableSize}$

- Open addressing does poorly with high load factor $\lambda$
  - Typically want larger tables
  - Too many probes means no more $O(1)$ 😭😭😭
Linear Probing: find

- You can figure this one out too 😊
  - Must use same probe function to “retrace the trail” for the item
  - Unsuccessful search when reach empty position

- What is `find`’s runtime ...
  - If key is in table
  - If key is NOT there?
  - Worst case?
Linear Probing: Remove

- **Must** use “lazy deletion”
  - Marker/tombstone indicates “no item here, but don’t stop probing”

|   | 10 | ☠️ | - | 23 | - | - | 16 | ☠️ | 26 |

- As with lazy deletion on other data structures, spots marked “deleted” can be filled in during subsequent adds
Linear Probing: Primary Clustering

- It turns out linear probing is a *bad idea*, even though the probe function is quick to compute (a good thing)
- Tends to produce *clusters*, which lead to long probe sequences
- Called *primary clustering*
- Saw the start of a cluster in our linear probing example

[R. Sedgewick]
Linear Probing: Analysis (1 of 2)

❖ **Trivial fact**: For any $\lambda < 1$, linear probing will find an empty slot
   - It is “safe” in this sense: no infinite loop unless table is full

❖ **Non-trivial facts** we won’t prove:
   - Average # of probes given $\lambda$ (in the limit as $\text{TableSize} \to \infty$)
     - Unsuccessful search: $\frac{1}{2} \left( 1 + \frac{1}{(1 - \lambda)^2} \right)$
     - Successful search: $\frac{1}{2} \left( 1 + \frac{1}{1 - \lambda} \right)$

❖ This is pretty bad: need to leave sufficient empty space in the table to get decent performance
Linear Probing: Analysis (2 of 2)

- Linear-probing performance degrades rapidly as table gets full
  - (Formula assumes “large table” but point remains)

- By comparison, separate chaining performance is linear in $\lambda$
  and has no trouble with $\lambda>1$