Today

- Dictionaries
  - B-Trees
  - Hashing
Motivating Hash Tables

For dictionary with $n$ key/value pairs

- Unsorted linked-list
- Unsorted array
- Sorted linked list
- Sorted array
- Balanced tree
- Big Array
- Hash Tables

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>O(1)</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>O(n)</td>
<td>O(1)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
</tbody>
</table>

"Average"
Keys = student IDs
0 to 99999999

Key = 7

We Insert? O(1)
Find? O(1)
Delete? O(1)

N = 2 + students

"Big Array"
Hash Tables

- Aim for constant-time (i.e., $O(1)$) find, insert, and delete
  - “On average” under some reasonable assumptions

- A hash table is an array of some fixed size

- Basic idea:

  key space (e.g., integers, strings)       hash function: $h(\text{key}) \rightarrow \text{index}$

  hash table

  0

  ...
Aside: Hash Tables vs. Balanced Trees

- In terms of a Dictionary ADT for just insert, find, delete, hash tables and balanced trees are just different data structures
  - Hash tables $O(1)$ on average (*assuming* few collisions)
  - Balanced trees $O(\log n)$ worst-case

- Constant-time is better, right?
  - Yes, but you need “hashing to behave” (must avoid collisions)
  - Yes, but what if we want to findMin, findMax, predecessor, and successor, printSorted?
    - Hashtables are not designed to efficiently implement these operations
    - Your textbook considers Hash tables to be a different ADT
    - Not so important to argue over the definitions
Hash Tables

• There are $m$ possible keys ($m$ typically large, even infinite)
• We expect our table to have only $n$ items
• $n$ is much less than $m$ (often written $n << m$)

Many dictionaries have this property

  – **Compiler:** All possible identifiers allowed by the language vs. those used in some file of one program

  – **Database:** All possible student names vs. students enrolled

  – **AI:** All possible chess-board configurations vs. those considered by the current player

  – ...
Hash functions

An ideal hash function:
- Is fast to compute
- “Rarely” hashes two “used” keys to the same index
  - Often impossible in theory; easy in practice
  - Will handle collisions a bit later

key space (e.g., integers, strings)
Who hashes what?

- Hash tables can be generic
  - To store keys of type $E$, we just need to be able to:
    1. Test equality: are you the $E$ I'm looking for?
    2. Hashable: convert any $E$ to an int

- When hash tables are a reusable library, the division of responsibility generally breaks down into two roles:

  ![Diagram showing the roles of client and hash table library]

- We will learn both roles, but most programmers “in the real world” spend more time as clients while understanding the library
More on roles

Some ambiguity in terminology on which parts are “hashing”

Two roles must both contribute to minimizing collisions (heuristically)

- **Client** should aim for different ints for expected items
  - Avoid “wasting” any part of \( E \) or the 32 bits of the int
- **Library** should aim for putting “similar” ints in different indices
  - conversion to index is almost always “mod table-size”
  - using prime numbers for table-size is common
What to hash?

- We will focus on two most common things to hash: ints and strings

- If you have objects with several fields, it is usually best to have most of the “identifying fields” contribute to the hash to avoid collisions

- Example:

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

- An inherent trade-off: hashing-time vs. collision-avoidance
  - Use all the fields?
  - Use only the birthdate?
  - Admittedly, what-to-hash is often an unprincipled guess
Hashing integers

key space = integers

Simple hash function:
- \( h(key) = key \mod TableSize \)
- Client \( f(x) = x \)
- Library \( g(x) = f(x) \mod TableSize \)
- Fairly fast and natural

Example:
- TableSize = 10
- Insert 7, 18, 41, 34, 10
- (As usual, ignoring corresponding data)
Hashing integers (Soln)

key space = integers

Simple hash function:
  \[ h(key) = key \mod TableSize \]
  
  - Client: \( f(x) = x \)
  - Library \( g(x) = f(x) \mod TableSize \)
  - Fairly fast and natural

Example:
  
  - TableSize = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring corresponding data)
Collision-avoidance

- With “x % TableSize” the number of collisions depends on
  - the ints inserted (obviously)
  - TableSize

- Larger table-size tends to help, but not always
  - Example: 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”
  - We’ll see some collision strategies do better with prime size
More arguments for a prime table size

If TableSize is 60 and...
- Lots of data items are multiples of 5, wasting 80% of table
- Lots of data items are multiples of 10, wasting 90% of table
- Lots of data items are multiples of 2, wasting 50% of table

If TableSize is 61...
- Collisions can still happen, but 5, 10, 15, 20, … will fill table
- Collisions can still happen but 10, 20, 30, 40, … will fill table
- Collisions can still happen but 2, 4, 6, 8, … will fill table

In general, if \(x\) and \(y\) are “co-prime” (means \(\gcd(x, y) = 1\)), then
\[
(a \times x) \mod y = (b \times x) \mod y \text{ if and only if } a \mod y = b \mod y
\]
- Given table size \(y\) and keys as multiples of \(x\), we’ll get a decent distribution if \(x \& y\) are co-prime
- So good to have a TableSize that has no common factors with any “likely pattern” \(x\)
What if the key is not an int?

- If keys aren’t \texttt{ints}, the \texttt{client} must convert to an \texttt{int}.
  - Trade-off: speed and distinct keys hashing to distinct \texttt{ints}

- Common and important example: Strings
  - Key space $K = s_0s_1s_2\ldots s_{m-1}$
  - where $s_i$ are chars: $s_i \in \{0,256\}$
  - Some choices: Which avoid collisions best?

1. $h(K) = s_0$
2. $h(K) = \left( \sum_{i=0}^{m-1} s_i \right) \pmod{\text{Tablesize}}$
3. $h(K) = \left( \sum_{i=0}^{m-1} s_i \cdot 37^i \right)$

Then on the \texttt{library} side we typically mod by Tablesize to find index into the table.

Similar to positional numbers

$s_0 \cdot 37^0 + s_1 \cdot 37^1 + s_2 \cdot 37^2 + …$
Specializing hash functions

How might you hash differently if all your strings were web addresses (URLs)?
Aside: Combining hash functions

A few rules of thumb / tricks:

1. Use all 32 bits (careful, that includes negative numbers)
2. Use different overlapping bits for different parts of the hash
   - This is why a factor of $37^i$ works better than $256^i$
3. When smashing two hashes into one hash, use bitwise-xor
   - bitwise-and produces too many 0 bits
   - bitwise-or produces too many 1 bits
4. Rely on expertise of others; consult books and other resources
5. If keys are known ahead of time, choose a perfect hash
Collision resolution

Collision:
When two keys map to the same location in the hash table.

We try to avoid it, but number-of-possible-keys exceeds table size.

So hash tables should support collision resolution
- Ideas?
Flavors of Collision Resolution

Separate Chaining

Open Addressing
- Linear Probing
- Quadratic Probing
- Double Hashing
Separate Chaining

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

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and Table Size = 10
Separate Chaining

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

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10
Separate Chaining

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

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10
Separate Chaining

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

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10
Separate Chaining

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

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10
Separate Chaining

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

As easy as it sounds

Example: insert 10, 22, 107, 12, 42 with mod hashing and TableSize = 10

Worst case time for find? $O(N)$
Thoughts on separate chaining

- Worst-case time for find?
  - Linear
  - But only with really bad luck or bad hash function
  - So not worth avoiding (e.g., with balanced trees at each bucket)
    - Keep # of items in each bucket small
    - Overhead of AVL tree, etc. not worth it for small n

- Beyond asymptotic complexity, some “data-structure engineering” can improve constant factors
  - Linked list vs. array or 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...
Time vs. space (constant factors only here)

0
1
2
3
4
5
6
7
8
9

0 10 /
1 /
2 
3
4
5
6
7
8
9

10
42
12
22
107

0 10 /
1 /
2 42
3 /
4 /
5 /
6 /
7 107 /
8 /
9 /
More rigorous separate chaining analysis

Definition: The load factor, \( \lambda \), of a hash table is

\[
\lambda = \frac{N}{\text{TableSize}} \quad \leftarrow \text{number of elements}
\]

Under chaining, the average number of elements per bucket is \( \lambda \)

So if some inserts are followed by \textit{random} finds, then on average:

- Each unsuccessful \texttt{find} compares against \( \lambda \) items
- Each successful \texttt{find} compares against \( \frac{\lambda}{2} \) items

- How big should \texttt{TableSize} be??

The book recommends \( N \), for separate chaining
More rigorous separate chaining analysis

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

$$\lambda = \frac{N}{\text{TableSize}} \leftarrow \text{number of elements}$$

Under chaining, the average number of elements per bucket is $\lambda$

So if some inserts are followed by random finds, then on average:
- Each unsuccessful find compares against $\lambda$ items
- Each successful find compares against $\lambda/2$ items
- If $\lambda$ is low, find & insert likely to be $O(1)$
- We like to keep $\lambda$ around 1 for separate chaining
Load Factor?

\[
\lambda = \frac{n}{\text{Table Size}} = \frac{5}{10} = \frac{1}{2}
\]
Load Factor?

$\lambda = \frac{n}{\text{TableSize}} = \frac{5}{10} = 0.5$
Load Factor?

\[
\lambda = \frac{n \geq 2^1}{TableSize} = ? \quad \frac{2^1}{10} \approx 0.2
\]
Load Factor?

\[
\lambda = \frac{n}{TableSize} = \frac{21}{10} = 2.1
\]
Separate Chaining Deletion?
Separate Chaining Deletion

- Not too bad
  - Find in table
  - Delete from bucket
- Say, delete 12
- Similar run-time as insert