Announcements

• Homework 3 due today
• Homework 4 (hash tables) out today, partner programming assignment, find a partner by Monday
Motivating Hash Tables

For a dictionary with $n$ key, value pairs

<table>
<thead>
<tr>
<th></th>
<th>insert</th>
<th>find</th>
<th>delete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted linked-list</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Unsorted array</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Sorted array</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>Balanced tree</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Magic array</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Sufficient “magic”:

- Use key to compute array index for an item in $O(1)$ time [doable]
- Have a different index for every item [magic]
Hash Tables

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

- A hash table is an array of some fixed size

- Basic idea:

  hash function: \( \text{index} = h(\text{key}) \)

key space (e.g., integers, strings)
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 `findMin`, `findMax`, `predecessor`, and `successor` go from $O(\log n)$ to $O(n)$, `printSorted` from $O(n)$ to $O(n \log n)$
    • Why your textbook considers this to be a different ADT
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:
- Fast to compute
- “Rarely” hashes two “used” keys to the same index
  - Often impossible in theory but easy in practice
  - Will handle collisions in next lecture

\[
\text{hash function:} \quad \text{index} = h(\text{key})
\]

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

• Hash tables can be generic
  – To store elements of type $E$, we just need $E$ to be:
    1. *Hashable*: convert any $E$ to an int
    2. *Comparable*: order any two $E$ (only when dictionary)

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

  ![Diagram](image.png)

• 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 the two most common things to hash: ints and strings

– For objects with several fields, 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

• Bad/Good idea(?):
  – Use only first name
  – Use only middle initial

• Admittedly, what-to-hash-with is often unprincipled 😞
Hashing integers

- key space = integers

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

- Example:
  - \( TableSize = 10 \)
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)
Hashing integers

- key space = integers

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

- Example:
  - \( \text{TableSize} = 10 \)
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)
Hashing integers

- key space = integers

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

- Example:
  - \( \text{TableSize} = 10 \)
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)
Hashing integers

- key space = integers

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

- Example:
  - \textbf{TableSize} = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)
Hashing integers

- key space = integers

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

- Example:
  - \text{TableSize} = 10
  - Insert 7, 18, 41, 34, 10
  - (As usual, ignoring data “along for the ride”)
Hashing integers

• key space = integers

• Simple hash function:
  \[ h(\text{key}) = \text{key} \mod \text{TableSize} \]
  – Client: \( f(x) = x \)
  – Library \( g(x) = x \mod \text{TableSize} \)
  – Fairly fast and natural

• Example:
  – \text{TableSize} = 10
  – Insert 7, 18, 41, 34, 10
  – (As usual, ignoring data “along for the ride”)
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”
  – Next lecture shows one collision-handling strategy does provably well with prime table size
More on prime table size

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

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

This “table-filling” property happens whenever the multiple and the table-size have a **greatest-common-divisor** of 1
Okay, back to the client

- If keys aren’t *ints*, client must convert to an *int*
  - Trade-off: speed vs distinct keys hashing to distinct *ints*

- Strings
  - Key space $K = s_0s_1s_2...s_{m-1}$
    - (where $s_i$ are chars: $s_i \in [0,52]$ or $s_i \in [0,256]$ or $s_i \in [0,2^{16}]$)
  - Some choices: Which avoid collisions best?

1. $h(K) = s_0 \mod \text{TableSize}$
2. $h(K) = (\sum_{i=0}^{m-1} s_i) \mod \text{TableSize}$
3. $h(K) = (\sum_{i=0}^{m-1} s_i \cdot 31^i) \mod \text{TableSize}$
Specializing hash functions

How might you hash differently if all your strings were web addresses (URLs)?
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 $31^i$ works better than $256^i$
   – Example: “abcde” and “ebcda”

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
One expert suggestion

- int result = 17;
- foreach field f
  - int fieldHashcode =
    - boolean: (f ? 1: 0)
    - byte, char, short, int: (int) f
    - long: (int) (f ^ (f >>> 32))
    - float: Float.floatToIntBits(f)
    - double: Double.doubleToLongBits(f), then above
    - Object: object.hashCode()
  - result = 31 * result + fieldHashcode
Hashing and comparing

• Need to emphasize a critical detail:
  – We initially *hash* key `E` to get a table index
  – To check an item is what we are looking for, `compareTo E`
    • Does it have an equal key?

• So a hash table needs a hash function and a comparator
  – The Java library uses a more object-oriented approach:
    each object has methods `equals` and `hashCode`

```java
class Object {
    boolean equals(Object o) {...}
    int hashCode() {...}
    ...
}
```
Equal Objects Must Hash the Same

- The Java library make a crucial assumption clients must satisfy
  - And all hash tables make analogous assumptions

- Object-oriented way of saying it:
  \[
  \text{If } a \text{.equals}(b), \text{ then } a \text{.hashCode}() == b \text{.hashCode}()
  \]

- Why is this essential?

- Why is this up to the client?

- So always override \texttt{hashCode correctly} if you override \texttt{equals}
  - Many libraries use hash tables on your objects
By the way: comparison has rules too

We have not emphasized important “rules” about comparison for:

– Dictionaries
– Sorting (future major topic)

Comparison must impose a consistent, total ordering:

For all \(a, b, \) and \(c\),

\[
\text{– } a.\text{compareTo}(a) == 0
\]
\[
\text{– } \text{if } a.\text{compareTo}(b) < 0, \text{ then } b.\text{compareTo}(a) > 0
\]
\[
\text{– } \text{if } a.\text{compareTo}(b) == 0, \text{ then } b.\text{compareTo}(a) == 0
\]
\[
\text{– } \text{if } a.\text{compareTo}(b) < 0 \text{ and } b.\text{compareTo}(c) < 0, \text{ then } a.\text{compareTo}(c) < 0
\]

This is surprisingly awkward because of subclassing…
Example

class MyDate {
    int month;
    int year;
    int day;

    boolean equals(Object otherObject) {
        if (this==otherObject) return true; // common?
        if (otherObject==null) return false;
        if (getClass() != other.getClass()) return false;
        return month == otherObject.month
                && year == otherObject.year
                && day == otherObject.day;
    }
}
Example

class MyDate {
    int month;
    int year;
    int day;

    boolean equals(Object otherObject) {
        if (this==otherObject) return true; // common?
        if (otherObject==null) return false;
        if (getClass()! = other.getClass()) return false;
        return month = otherObject.month
        && year = otherObject.year
        && day = otherObject.day;
    }

    // wrong: must also override hashCode!
}
Tougher example

• Suppose you had a `Fraction` class where `equals` returned `true` for 1/2 and 3/6, etc.

• Then must override `hashCode` and cannot hash just based on the numerator and denominator
  – Need 1/2 and 3/6 to hash to the same int

• If you write software for a living, you are less likely to implement hash tables from scratch than you are likely to encounter this issue
Conclusions and notes on hashing

• The hash table is one of the most important data structures
  – Supports only **find**, **insert**, and **delete** efficiently
  – Have to search entire table for other operations

• Important to use a good hash function

• Important to keep hash table at a good size

• Side-comment: hash functions have uses beyond hash tables
  – Examples: Cryptography, check-sums

• Big remaining topic: Handling collisions