Trees, Dictionaries, and Maps

Instructors: Sam McCauley and Dan Barowy

April 25, 2022

Admin

- Any questions?

Putting AVL trees together (from last week)

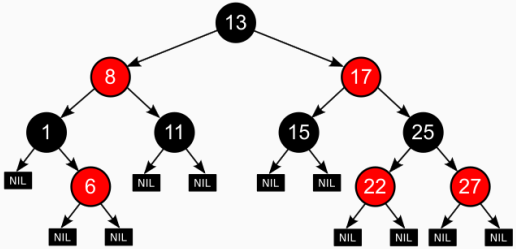
- Invariant: each node has balance -1, 0, or 1 (maintained through two rules for rotations)
- The lowest number of nodes in an AVL tree of height h is $w(h) \geq (3/2)^h$
- Therefore, if an AVL tree has n nodes, then the height of the tree must satisfy $n \geq (3/2)^h$, so $h \leq \log_{3/2} n$
- Therefore, $h = O(\log n)$!
- As you may have seen elsewhere, if $\phi = (1 + \sqrt{5})/2 \approx 1.62$, then $\log_\phi n$ is a much tighter bound.

Wrapping it Up

- AVL trees support `add()`, `contains()`, `remove()` (we didn't talk about remove; same idea but more complicated rules) all in $O(\log n)$ time
- Every other data structure we've seen requires at least $O(n)$ time!
- So on a data structure with a billion items, requires $\approx$ 30 operations rather than $\approx$ 1000000000.
- Incredible example of:
  - How more intricate data structures can improve performance (past what seemed possible)
  - How simple invariants can lead to performance improvements
  - How more-involved analysis can help us analyze complex data structures

Other Trees and Tree Operations

Red-black trees



- Another way to implement a Balanced Binary Search Tree
- Some advantages; some disadvantages in practice compared to the AVL tree
- Also get $O(\log n)$-time operations
- This is the balanced binary search tree implemented in `structure5`

Predecessor and Successor Queries

- A *predecessor query* for an item v in a set of items returns the largest v' satisfying $v' \leq v$
    - So if an item is located in the set, a predecessor query is equivalent to a `contains()` query
- A *successor query* for an item v in a set of items returns the smallest v' satisfying $v' \geq v$
- How can we implement these queries in a Binary Search Tree?

Predecessor Queries

- If query is larger than root, set the root as the current best predecessor. Then, search in right subtree.
- If query is smaller than root, recurse in left subtree
- If reach an empty node, return the stored predecessor
- What's the invariant of this approach?
  - The predecessor of the query is either the stored best predecessor, or in the subtree being searched
  - How can we prove this?
- Running time on a BBST?
  - $O(\log n)$

Range Queries

- Let's say we want to find all entries between two elements q_1 and q_2
- Say: "find all students with names in this range" or "find all companies with number of employees in this range" (very common type of database query)
- Let's say there are k entries in the range, and our data structure is of size n. Let's calculate performance in terms of n and k
- How long does this query take on an unsorted `Vector`?
  - $O(n)$
- What about a sorted `Vector`?
  - $O(k + \log n)$: binary search for q_1; list entries until q_2
- What about a BBST?
  - Also $O(k + \log n)$. (Essentially: search for q_1; do a careful traversal to q_2.)

Dictionaries and Maps

Dictionary data structure

- Store data associated with a set of *keys*
- Goal: for a given key, want to be able to look up the associated data (which we call a *value*)
- For example: let's say we have a list of words. We want to be able to look up the definition of any word.
  - keys are the words
  - definitions are the values
- For Google: given a keyword, find all websites that contain that keyword
- Given a course name, find the list of all students that are taking that course
- Given a *k*-gram, find its `FrequencyList`

Dictionary methods

- method `contains(key)` returns a `boolean`
- method `getValue(key)` should get the value associated with a key
- Want to be able to update dictionary: `add(key, value)` and `delete(key, value)`
- Each key should appear once. (Why?)
    - Unambiguous lookup! If a query a key, I should know exactly what value I'm getting

Dictionaries We've Seen

- Unordered vector: $O(1)$ `add`, $O(n)$ `delete`, `contains`, `getValue`
- Ordered vector: $O(n)$ `add`, $O(n)$ `delete`, $O(\log n)$ `contains`, `getValue`
- Balanced Binary Search Tree (i.e. AVL trees): $O(\log n)$ for all operations
    - And can also do predecessor, successor, range queries efficiently

A Dictionary Interface: Map

- Let's take a look at the `Map` interface

- (Map and Dictionary are essentially synonyms. The interface just happens to be called Map.)

Main Map Interface Methods

- `int size()` – returns number of entries in map
- `boolean isEmpty()` – true if there are no entries
- `void clear()` – remove all entries from map
- `boolean containsKey(K key)` – true if key exists in map
- `boolean containsValue(V val)` – true if val exists at least once in map
- `V get(K key)` – get value associated with key
- `V put(K key, V val)` – insert mapping from key to val, returns value replaced (old value) or null
- `V remove(K key)` – remove mapping from key to val

Other Map Interface Methods

- `void putAll(Map<K,V> other)` – puts all key-value pairs from an existing Map into the current map
- `Set<K> keySet()` – return set of keys in map
- `Structure<V> valueSet()` – return collection of values
- `Set<Association<K,V>> entrySet()` – return set of key-value pairs from map
- `boolean equals()` – true if two maps are entrywise equal
- `int hashCode()` – returns hash code associated with values in map (stay tuned...)

Simple `Map` implementation: `MapList`

- Implements a `Map` using a singly linked list
- How do you think `get(K key)` works?
- How about `put(K key, V val)`?
- What is the running time?
  - $O(n)$ ☹
- How fast can we get these operations implemented?
  - $O(\log n)$ using a BBST
- Is that optimal?

Hash Codes and Hash Tables

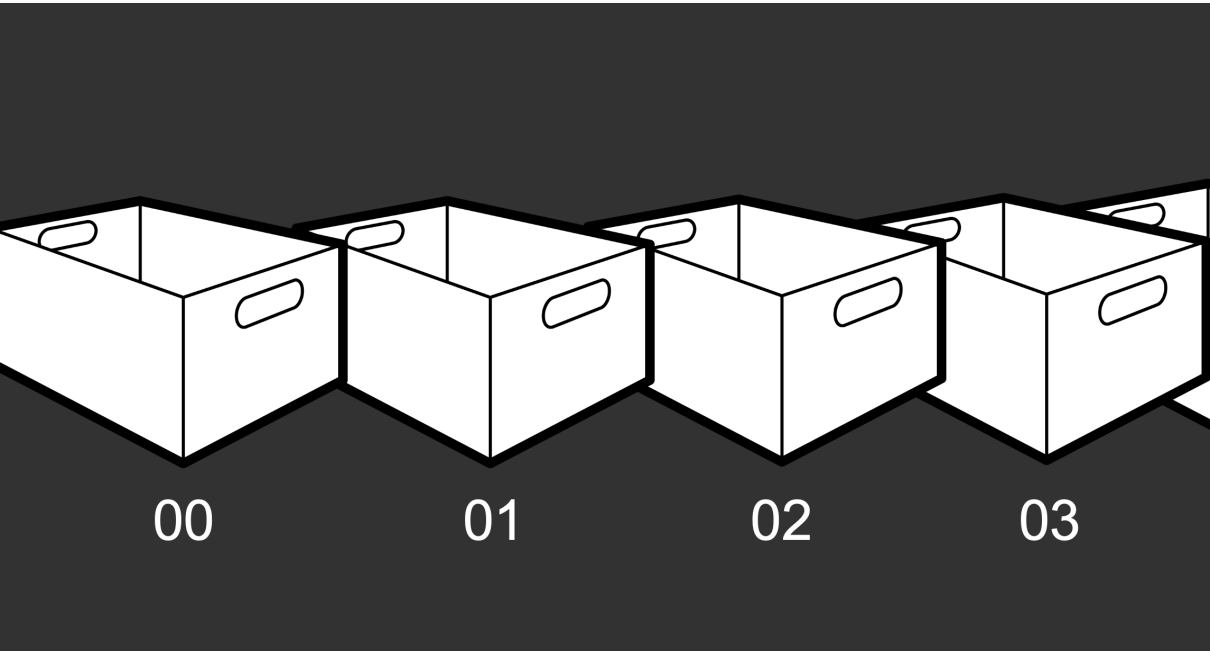
Hash Tables

- Hash tables can add an locate objects in roughly $O(1)$ time!!!
- Later this week we'll see why $O(1)$ is a bit of a fuzzy claim
- Let's look at a real-world example to help us think through the hash table strategy

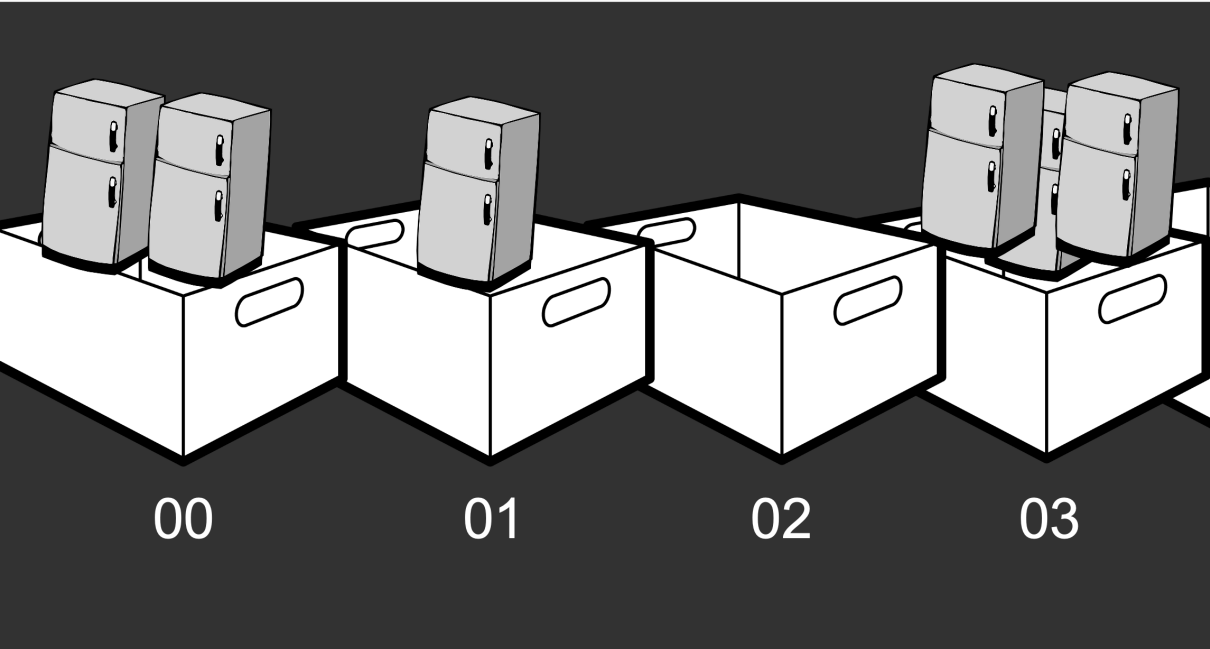
Example from Bailey

> *We head to a local appliance store to pick up a new freezer. When we arrive, the clerk asks us for the last two digits of our home telephone number! Only then does the clerk ask for our last name. Armed with that information, the clerk walks directly to a bin in a warehouse of hundreds of appliances and comes back with the freezer in tow.*

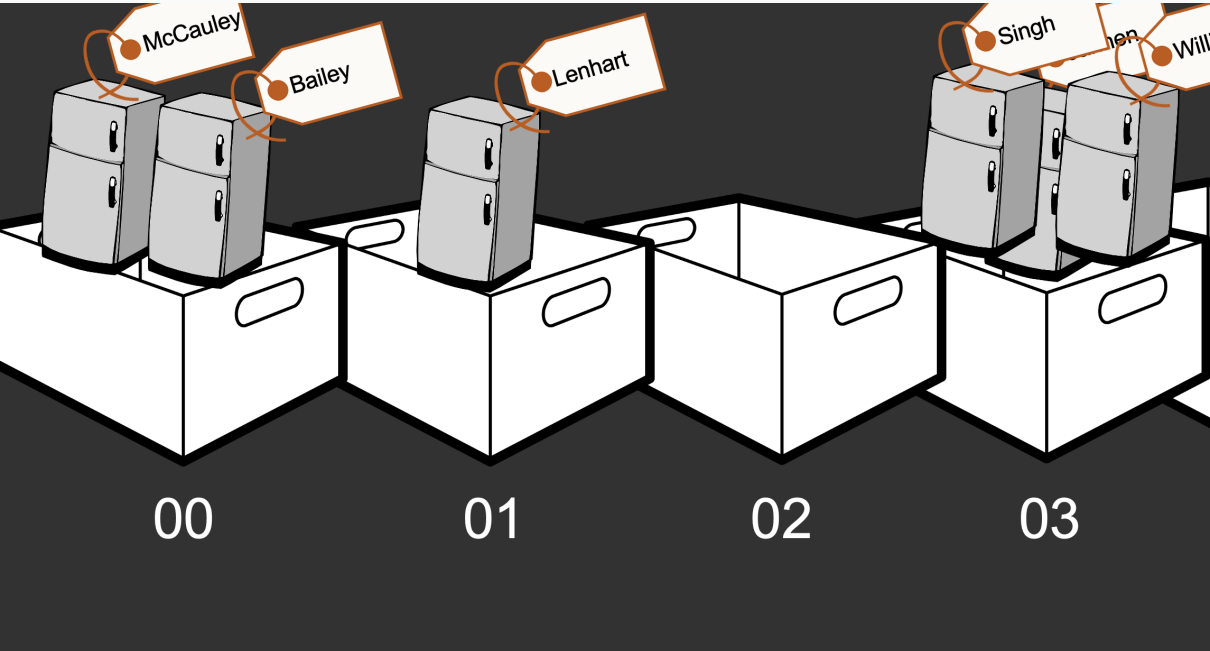
Example from Bailey



Example from Bailey



Example from Bailey



Example from Bailey

> *We head to a local appliance store to pick up a new freezer. When we arrive, the clerk asks us for the last two digits of our home telephone number! Only then does the clerk ask for our last name. Armed with that information, the clerk walks directly to a bin in a warehouse of hundreds of appliances and comes back with the freezer in tow.*

- How does this relate to the `Map` interface?
- What is the `Key`? What is the `Value`?
- Why those choices?
  - Are names evenly distributed?
  - Are the last 2 phone digits evenly distributed?

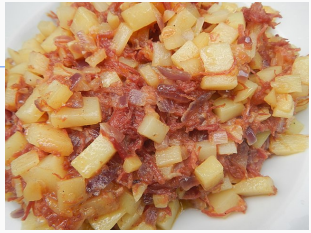
Hashing in a Nutshell

- Assign objects to “bins” based on key
- When searching for an object, jump directly to the appropriate bin (and ignore the rest)
- If there are multiple objects assigned to the target bin, then search for the right object
- Important Insight: Hashing works best when objects are *evenly distributed* among bins
- Phone numbers are randomly assigned, last names are not!

Implementing a Hash Table

- How can we represent bins?
  - Slots in an array! (We'll talk about how to grow later.)
- How do we find a key's bin?
  - We use a *hash function* that converts keys (of type `K`) into `int`s
- In Java, all `Objects` have a method `public int hashCode()`

`hashCode()` properties



- Return type `int`
- Hashing function is *one way:*
  - Can convert a key into a hashCode
  - May not be able to convert a `hashCode` into a key
- Hashing function is deterministic

Hash Code Rules

The general contract of `hashCode` is:

- Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is *not* required that if two objects are unequal according to the `equals(java.lang.Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer results for unequal objects may improve the performance of hash tables.

https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()

Implementing HashTable

- `hashCode()` allows us to jump directly to the bin for a particular key object
- How do we add `Associations` to our array?
- Problem 1: `hashCode()` yields an int, but our array may be relatively small.
  - How do we convert arbitrary ints to array locations?
- Problem 2: We can represent 2^{32} unique `int`, but there may be infinitely many values that an object can take on (e.g., String).
- By the pigeonhole principle, some Strings will have to “share” a hashcode!

Fitting Items Into Array

- Use mod (in Java: %) to map the object to an array index
- Something like:
- `array[o.hashCode() % array.length] = o;`
- That way, every object fits to some slot in our array using its hash code

Objects with the same hashcode

- If two objects map to the same slot in the array (after taking mod of the hashcode), it is called a *collision*
- Could be two objects with the same hashcode, or two objects with different hash codes that map to the same slot
- Can we guarantee that collisions can't happen?
  - No: for any hash code we write, *some* pairs of objects will have a collision
- Instead: create a strategy for storing items (extending the "mod" idea above) that can handle collisions!

On Wednesday

- See how to store items that share the same hashcode
- We'll talk a little bit about analysis