Hashtables

Lunch & Learn

Ethan Kent

February 7, 2020

Purpose

The goal of this presentation is to discuss what hashtables are, how they work, and the tradeoffs that go along with different designs.

Data structures and tradeoffs

Linked lists

O(1) insertion and popping; O(n) for search, access by position.

```
Node = Struct.new(:value, :next_node)

node1 =  Node.new(0)
node2 =  Node.new(1)
node3 =  Node.new(2)
node4 =  Node.new(3)

node1.next_node = node2
node2.next_node = node3
node3.next_node = node4
node4.next_node = :null_pointer

puts node1
```

Data structures and tradeoffs

Arrays

Linked lists have pointers to arbitrary other locations in memory. By contrast, arrays are contiguous locations in memory.

Arrays have O(1) insertion, popping, and access by position, and O(n) search.

```
int * my_array;

my_array = (int * ) malloc(sizeof(int) * 50);

for(i = 0; i < 50; ++i) {
    my_array[i] = 0;
}
```

Data structures and tradeoffs

The desire for constant-time “find” operations

- With a linked list, we can only find an element by starting at the beginning and searching every spot. We can't even go directly to an element if we know its index.
- With a vector, we can only find an element by starting at the beginning and searching every spot. But we *can* go directly to an element if we know its index.

It would be nice would be to have some way to use the value we're checking for *to tell us where to look*, so we could go directly there.

An analogy

What if we had a machine that could use the description of your backpack:

and tell you it belongs in locker 57?

An analogy

What else would our machine have to do? It should—

- Always give you the same answer given the same backpack.
- Not put every backpack in locker 57.
- Not put most backpacks in just a few lockers.
- Not put most blue backpacks in just a few lockers.
- Give you an answer pretty quickly.

If we had that, we could walk directly do locker 57 and get our backpack, without the need to search.

An analogy

But there are still some problems. What if—

- More than one backpack is assigned to the same locker (assume only one backpack can fit)?
- There are more backpacks than lockers?

Hashtables

We have now considered several aspects of a data structure called a *hashtable*.

- The machine that turns the backpack into a locker assignment is a hash function.
- The characteristics of a good backpack assigner are the characteristics of a good hash function:
    - Deterministic,
    - Fast,
    - Uniform, and
    - Avalanching.
- The problems you can run into are the same as those with a hashtable:
    - Collisions, and
    - Load factor.

The key idea

If we can use *the data itself* to tell us where the data belongs, and if we can go directly to where the data belongs, we get best-case `O(1)`. This compares favorably with:

- Fully sorting the data (`O(n log n)` for insertion).
- Traversing a linked list or array (`O(n)` for search).
- Using a balanced data structure (`O(log n)` for common operations).

The birthday problem

You're throwing a party. You invite one person at a time until two people have the same birthday. On average, how many people do you have to invite to your party (assuming birthdays are distributed evenly across a 365-day year?)

The birthday problem

```ruby
require "set"
year = (1..365).to_a
num_runs = 100_000

total = num_runs.times.reduce(0.0) do |total|
  set = Set.new

  loop do
    added = set.add?(year.sample)
    break unless added
  end

  total + set.length
end

puts total / num_runs
```

The birthday problem

About 24.

The birthday problem

Why am I telling you that?

Collisions

Remember the idea of more than one backpack being assigned to one locker? Even with a good hashing algorithm, collisions will occur earlier than we might expect:

- With 100 buckets, the average first collision occurs around the 13th element.
- With 1,000, it's around the 40th.
- With 10,000, it's around the 125th.

Collisions

We'll discuss two ways to deal with collisions:

1. Separate chaining.
2. Open addressing.

Separate chaining

Our backpack machine could send us to a locker. That locker could have a note telling us to go to another locker. If that locker is full, it could tell us to go to another locker. And so on. Once we find an empty locker, we put our backpack there.

In other words, the locker could contain a (pointer to) a linked list.

To see if the hashtable contains a value, we can find the right bucket by hashing the value, then search the linked list. If we get to the end and haven't seen our value, we know it isn't there.

Open addressing

Our backpack machine could send us to a locker. If that locker is full, we could keep checking the next locker, and the next, and the next, until we found an empty one, then put our backpack there.

In other words, the array can actually hold the data, rather than holding a pointer to the data.

To see if the hashtable contains a value, we can find the right bucket by hashing the value, then check the bucket, and if it's full, the next bucket, and so on, until we either find our value or find an empty bucket.

Open addressing and collisions

Open addressing is often preferable. Among other advantages, it can sometimes benefit from caching. But collisions are a bigger problem.

- Collisions with separate chaining are straightforward to handle: add to the linked lists.
- With open addressing, there are problems—
  - The table can become completely full.
  - Collisions cause more collisions.

Open addressing and deletions

When you delete an element from a hashtable, there's a problem. You create an empty space. But remember what an empty space can mean.

Load factor

When the load factor of a hashtable gets above a certain level, we may need to rehash: grow the hashtable and move elements around.

Optimizations

Robin Hood hashing is one optimization we'll discuss. There are others.