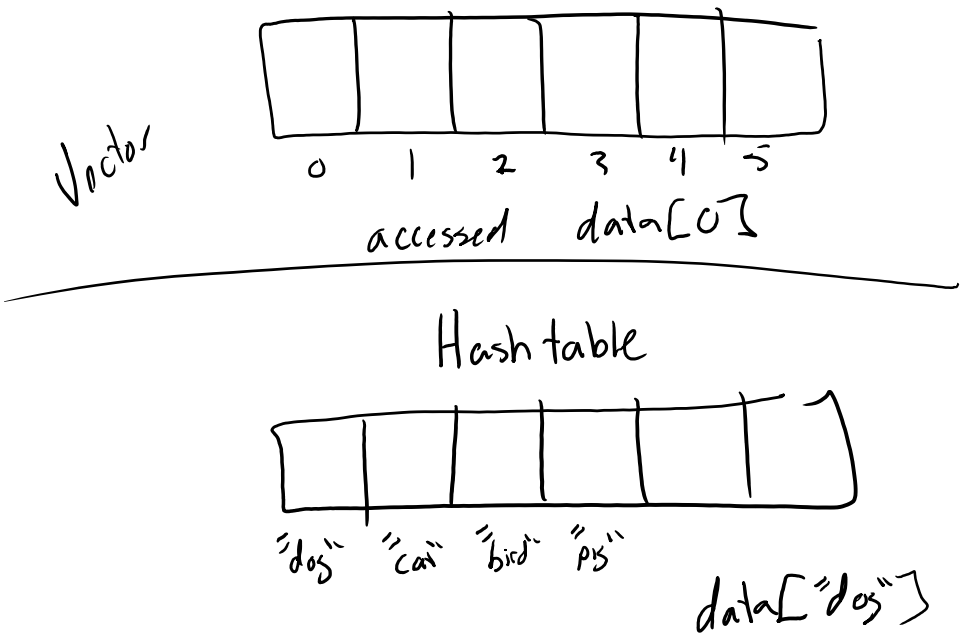
2019-04-09 Intro to Hashtables

Tuesday, April 9, 2019 9:01 AM

- A hashtable is a vector-like structure that allows a programmer to use non-numeric keys for indexing.



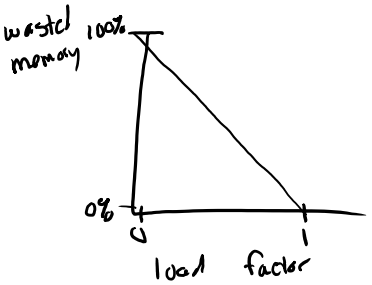
- From a programmer's perspective, operating on a hash table is pretty easy, especially when HTs are first-order data types (e.g. Python, JavaScript, PHP)
- Unfortunately for us, HTs are a bit more difficult to use in C++
- In C++, the HT data type is called unordered_map. There is also a "map" data type. This **IS NOT** a hash table. It's a red-black tree (similar to AVL tree).
- Depending on the task for which the data structure is used, maps can be significantly slower than unordered_maps
- As the name states, entries in an unordered_map are not ordered!
- On average, HTs are just as fast as vectors for inserts and finds and are generally faster for removes. However, they do tend to take up a bit more space (20-50% more)
- **CAVEAT:** Because HTs are unordered, you cannot use standard FOR loop. Instead, must use modern FOR auto loop.
    - In the case of a HT, auto is a key-value pair w/ .first being the key and .second being the value.

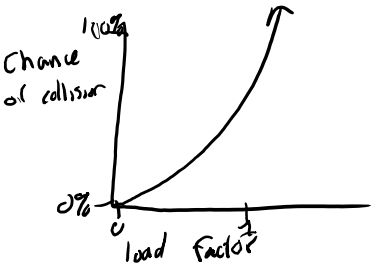
- A coding interlude…

When programming your own hash table, you must consider the following three factors

1. How do you turn non-numeric data into numeric data in a predictable manner. This process is called hashing. The quality and complexity of the hashing function directly impacts the speed of the HT.
    a. It impossible to create a hashing function that guarantees a one-to-one correspondence between a non-numeric value and an integer
    b. When two different items hash to the same value, we call this a "collision"

2. What happens when two keys collide (expect to go into the same vector box)? This problem is called a collision resolution mechanism.
3. Load factor, or fullness of HT's underlying vector. Load factor ranges from 0 (empty) to 1 (completely full). In general, the higher the load factor, the more likely a collision will occur. Higher load factor means less wasted memory, but higher collision rate. Lower load factor means more wasted memory, but lower collision chance.





Hashing functions

- A hashing function should ideally be capable of creating very large numbers with a reasonable amount of spread between like keys.

A really bad string hashing function

```
int keyFromString(string s){
   int hash_value = 1;
   for(auto ch : s){
      hash_value += ch;
   }
   return hash_value;
}
```

- Not great because similar keys (e.g. abc and cba) hash to the exact same value
- Not much distance between similar chars (e.g. a, b)
- Not very big numbers are generated from this hash

A better hashing function

```
int keyFromString(string s){
   int hash_value = 1;
   int index = 1;
   for(auto ch : s){
      hash_value *= 2^ch + 5 * ch + ch * index;
      index ++;
   }
   return hash_value;
}
```

Collision Resolution Mechanisms

- One of two categories
  - Open Addressing: If a box is full, somehow find a new box
    - Finding a new box is called probing
  - Separate chaining: Treat the HT as a 2D vector. Thus each box can accommodate infinite collisions
    - As a given box grows deep, the cost to find, update, and remove HT items goes up

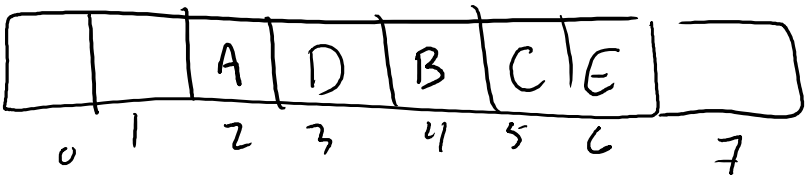
Open Addressing

Linear Probing

- If the box is full (e.g. box 2), try the next one over (i.e. box 3). Do this until

we find an empty box.

- Example: given the following key and hash values, what would the HT look like?
    - A -> 2, B -> 4, C -> 5, D -> 3, E -> 3



- With linear probing, high load factors result in a more linear access time (bad!)
- Linear probing tends to create collision clusters -> collision in one area tends to affect keys that are not close to the collision
    - E.g. F->6