2018-10-18 Hash Tables (not Graphs!)

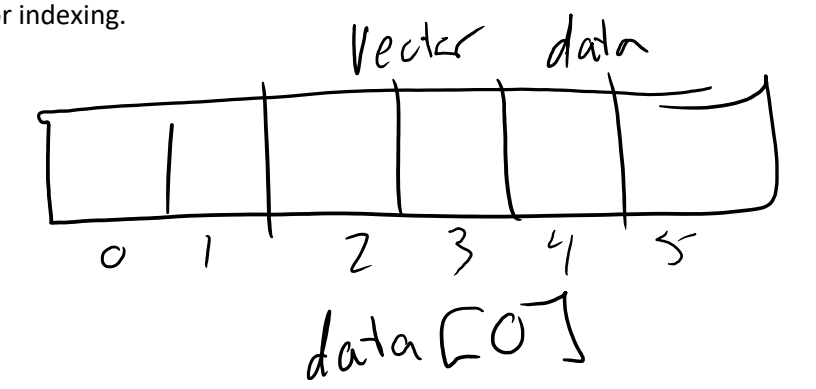
Thursday, October 18, 2018 3:07 PM

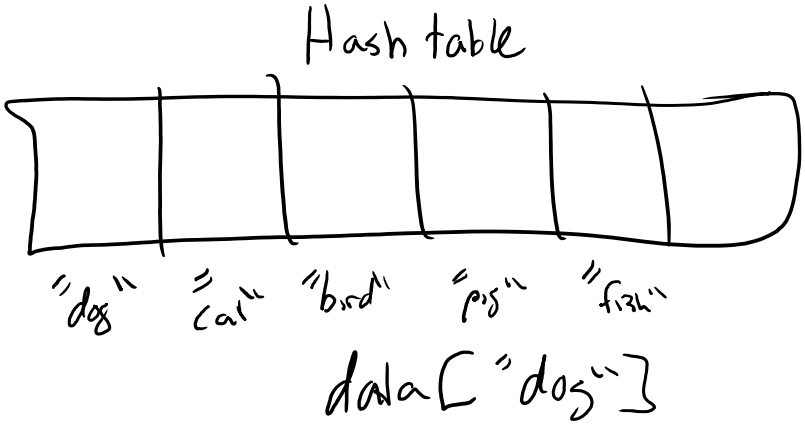
Graph Preliminaries

- Graphs are a "capstone" data structure in that employ many of the data structures discussed in class. Depending on the problem, we may use:
  - Hash tables
  - Priority Queues
  - Vectors
  - Linked Lists
  - Queues
  - Stacks

Hashtables

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





- From a programmers perspective, operating on a hash table is pretty easy, especially when HTs are first-order data types (e.g. Python, JavaScript, PHP)
- Unfortunately, the syntax in C++ is a little gnarly.
- In C++ the hash table data type is called unordered_map. There is also a map structure that works and looks just like unordered_map. They are not same, do not use map unless you have a very good reason (tends to be a lot slower!)
- Entries in an unordered_map are not ordered in any particular way (obvious!), but somehow through magic (which we will learn), unordered_maps allow for

O(1) loop, insert, and removal, just like a Vector

- Because HTs are unordered, you CANNOT use a standard FOR loop to access data. Instead, you MUST use the modern C++ FOR loop
  - For(auto item : data)
  - item is a STL pair, which has two properties .first -> key and .second -> value
- Adam's personal data structure usage breakdown:
  - 50% vectors, 40% HTs (includes graphs), 10% rest

Three factors that must be considered when designing a hash table

1. How do you turn non-numeric data into numeric data in a predictable manner. This process is called hashing. The quality of the hashing function directly impacts the speed of the HT.
   a. When two items hash to the same value, we call this a hash collision
2. What do you do when two keys hash to the same box? This is called a collision resolution mechanism.
3. Load factor, which means how full is the underlying HT's vector. 0-1; 0 = 0 empty vector, 1 = 1 full vector. In general, the higher the load factor, the more likely a collision will occur. Higher load factor means less memory but higher collision %. Lower load factor means more memory but lesser collision %.

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 hashing function

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

Lots of collisions
(adf, acg, bah)
Not very big numbers
("ZZZ....Z") → 4515

A slightly better hashing function

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

Collision Resolution Mechanism

- What do you do when two keys want to store data in the same box?

Open Addressing

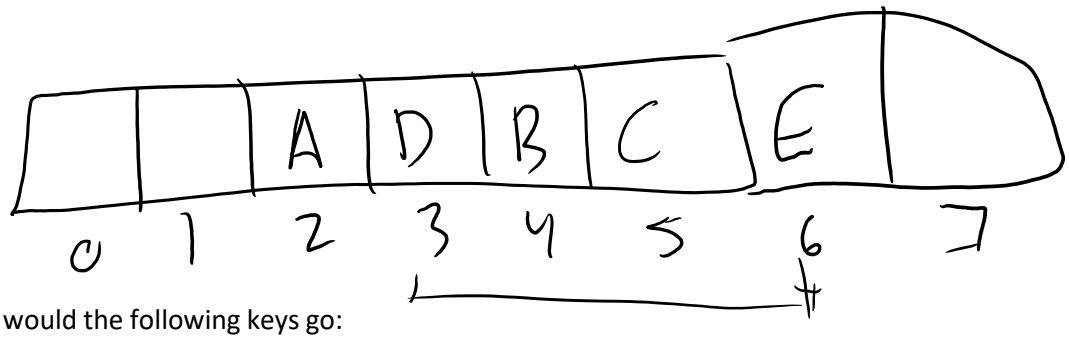
- If a box is full, somehow find an empty box (called probing)

Linear Probing

- If my box is full (e.g. box 2), try the next one over (e.g. box 3). Keep going until we find an empty box.

Linear Probing

- If my box is full (e.g. box 2), try the next one over (e.g. box 3). Keep going until we find an empty box.



- Where would the following keys go:
  - A-> 2
  - B -> 4
  - C -> 5
  - D -> 3
  - E -> 3
- With linear probing, higher load factors result in more linear access times
- Linear probing tends to create collision clusters. Collision in one area tend to affect keys that aren't even "close" to the collision.
  - E.g. F -> 6

Quadratic Probing

- Rather than looking to the next box, probe using a quadratic formula (e.g. (index + 3)^2 + 2*index + 5)
- Tends to reduce clustering effect
- Mathematically certain to get "stuck" in a probing cycle is load factor exceeds a certain amount
  - ~50% load factor
- Example HT probe = I^2 + 1 on A->2, B->3, C->2, D->4, E->2