2018-10-25 Hashtables

Thursday, October 25, 2018 3:00 PM

Recall from last lecture

- Hash tables expose a vector-like structure to the programmer that accepts non-numeric keys
- When designing a hash table, 3 considerations must be made:
  - How do we efficiently convert from a non-integer into an integer (hashing function)?
  - How do we handle hash collisions (when two items hash to the same number; inevitable)?
  - How full do we allow our hash table to become before resizing (load factor, expressed as a %)
    - Normal vectors can wait until a load factor of 100% before resizing
    - Some hash table implementations cannot be more than 50% full

Collision Resolution Mechanisms

Approach #1: Open Addressing

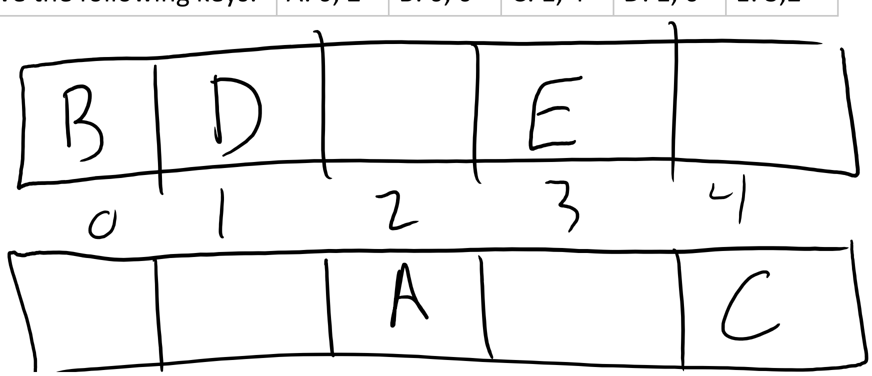
- General idea: if the box is full, find another box
- Last lecture we looked at linear probing (look at next box until empty found)
  - Positive: Linear probing allows for 100% load factor (will be slow, however)
  - Negative: Collisions create a clustering effect, increasing the probability of future collisions
- We also looked at quadratic probing (use quadratic formula to find next box)
  - Positive: Less clustering effect, thus tends to require less probes than linear probing
  - Negative: Requires a lot of empty space; more than 50% full means we cannot guarantee that we will find an empty box.
- Next approach: Double Hashing
  - On collision, use another, different hashing function plus a "salt" to find the next box
  - Pros and Cons are very similar to quadratic hashing
  - Example hash1(x) = x^2 * x + 3 hash2(y,salt) = 5*y + 7 * 2*salt + 1;
  - In Adam's benchmarks, double hashing tends to be the slowest between linear, quadratic, and double

Modern Approaches to Open Addressing

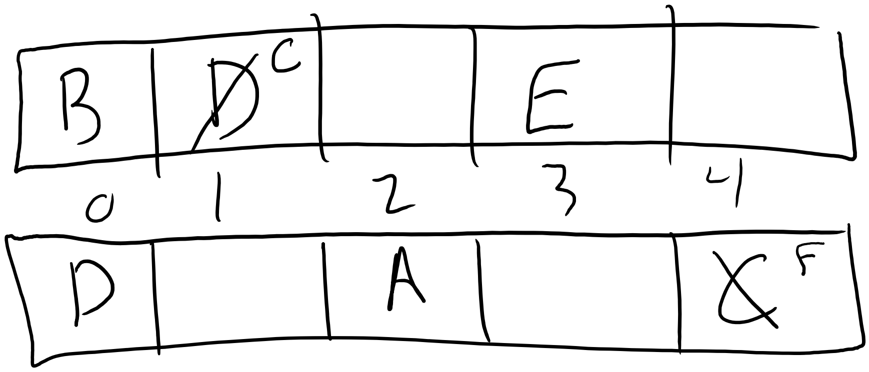
Cuckoo Hashing

- Invented in 2001
- Came about as a result of advancements in probability theory
- Similar to double hashing but instead utilizes two parallel arrays
- Like double hashing, there are two hashing functions, one for each array
- On an insert, randomly select one of the arrays to place the item in.
  - If something is already there (collision), take its place and force it to find a new home

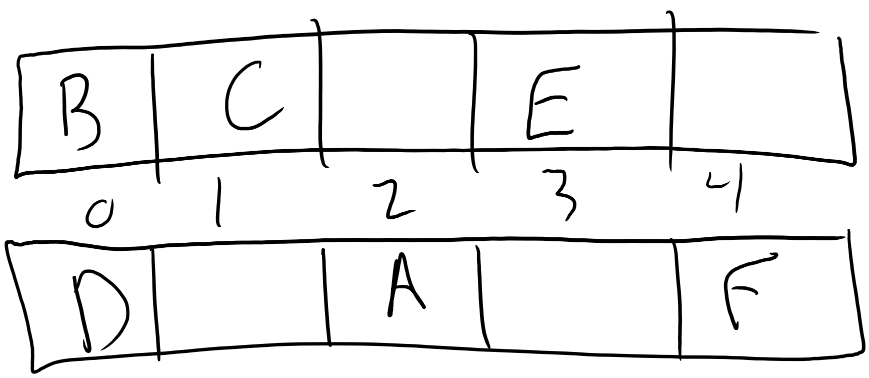
Assume we have the following keys:	A: 0, 2	B: 0, 0	C: 1, 4	D: 1, 0	E: 3,2



- Insert F: 3, 4 (say we randomly picked the 2nd array to insert into)

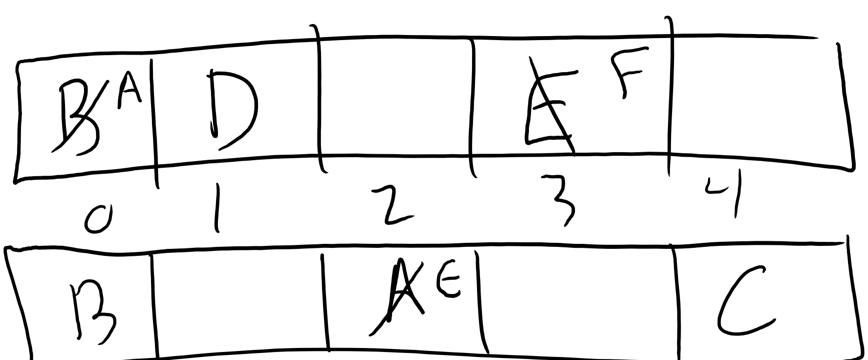


Final result:



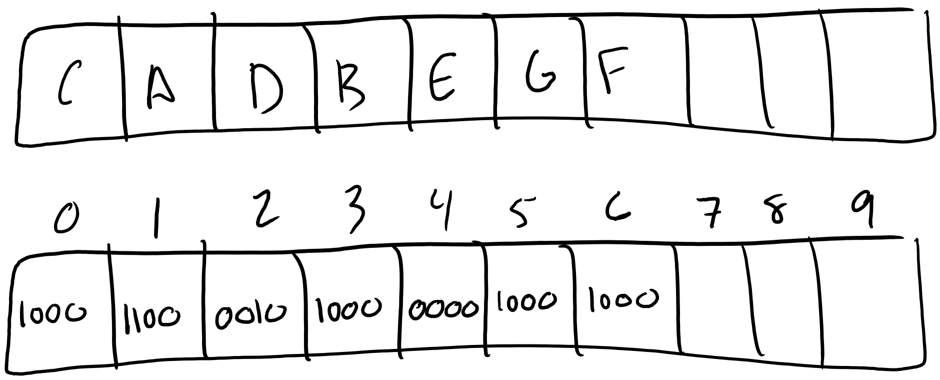
What if instead F went into the first array?

Assume we have the following keys:	A: 0, 2	B: 0, 0	C: 1, 4	D: 1, 0	E: 3,2	F:3,4



Hopscotch Hashing

- First paper published in 2009
- Similar idea to linear probing but it places a max limit on how far an item can be away from its original hash.
    - In our examples, max distance is up to 3 spaces away
    - Guaranteed O(1). Open question: is the extra bookkeeping worth it?
- Like cuckoo hash tables, hopscotch uses two arrays: one data array and one distance array



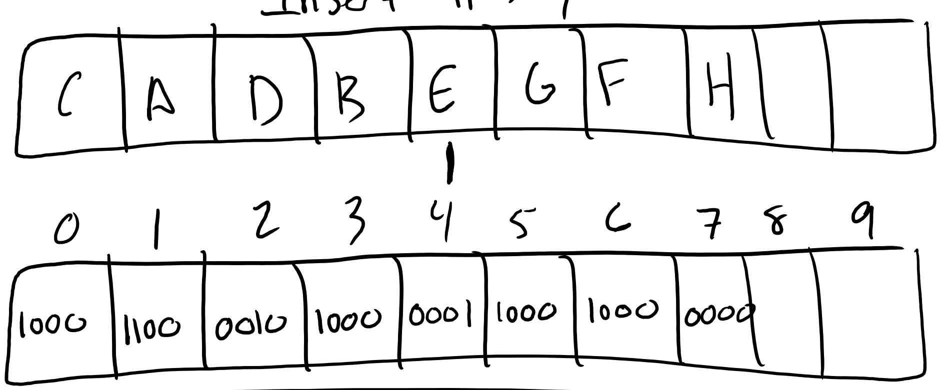
- Bits in 2nd array track "distance from the origin"

0/1	0/1	0/1	0/1
The item that is presently in the corresponding box hashes to this value	Item that is +1 away hashes to this value	+2 away…	+3 away…

Algorithm (from wikipedia)

1. If the original hashed box is empty, add to that box. Update bits. **DONE.**
2. Otherwise, starting at the hashed value, try to find a box that is empty up to max distance. Update bits box.
3. If all boxes up to max distance are occupied, working from first empty box back to the origin, try to move values in other boxes farther if allowable.
4. If #3 allows us to make room, we're good. If we get back to the origin and could not find an available slot, resize!

Insert H → 4



I → 3

C	A	D	B	E	G	F	H		

0	1	2	3	4	5	6	7	8	9
1000	1100	0010	1000	0001	1000	1000	0000		

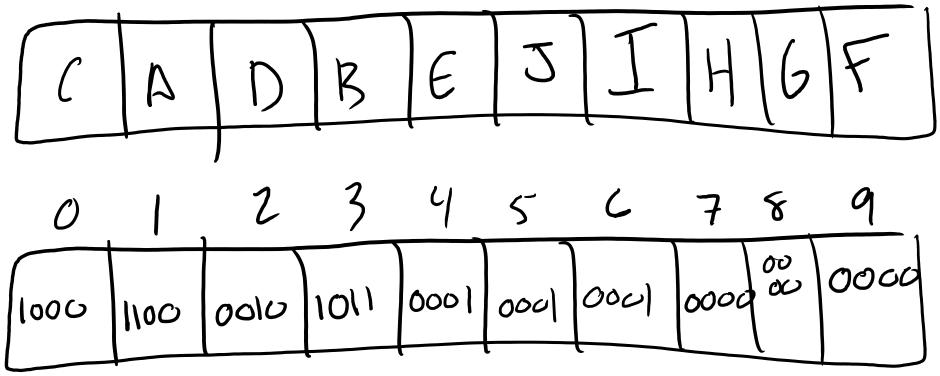
C	A	D	B	E	G	I	H	F	

0	1	2	3	4	5	6	7	8	9
1000	1100	0010	1001	0001	1000	0010	0000	0000	

J → 3

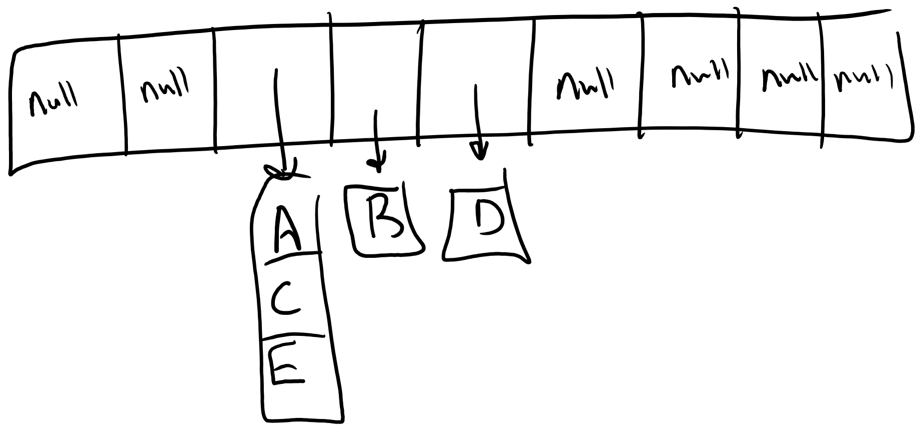
C	A	D	B	E	G	I	H	F	

0	1	2	3	4	5	6	7	8	9
1000	1100	0010	1001	0001	1000	0010	0000	0000	



Separate Chaining

- Why do we allow only one value in a given box?
- Why not treat each box as a linear structure (e.g. vector or LL)?

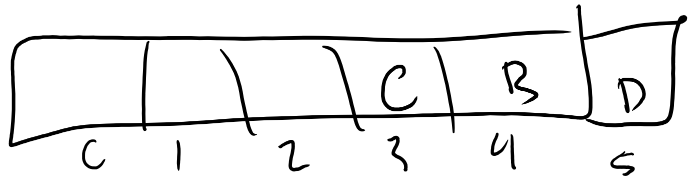


- To add an item, just go to the original hash location, search (linearly) for the item. If it is not there, add to the end. Otherwise, update value.
- At present, the STL unordered_map uses this technique
- Implications
  - As buckets get more full, performance becomes more linear
  - Bucket hash tables can have a load factor greater than 100%

Key Issue: Removing items from a HT

- Consider a linear probing hash table with the following keys:
- A->2, B->4, C->2, D->4

- What happens if we were to remove A?



- Consider, what happens if we try to now retrieve C?
- A "hard delete" ends up ruining the probing algorithm (makes things appear to not be in the table)
- As a workaround, we perform a "soft delete" where we mark the value as having been deleted, but it actually still takes up space

- Items won't actually get removed from the HT until a resize occurs
- Resizing is also interesting
  - Every time you resize, the hash mod by amount changes. Thus, what once hashed to location 1 may hash to location 20
  - To solve this, every item in the hash table must be rehashed and replaced.