2019-04-11 Hashtables pt 2

Thursday, April 11, 2019 8:59 AM

Recall from yesterday…

- We were exploring collision resolutions using open addressing
  - Open addressing -> If a box is already taken, try to find another box

Linear Probing Recap

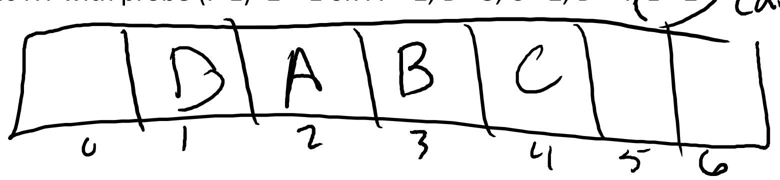
- If a box is full, check the next box over to see if it's full. Repeat until an empty box is found.
- Downside: linear probing results in collision clusters which hurt the performance of the hashtable

Quadratic Probing

- Rather than checking the next box over, we check for a box some quadratic distance away
- Next box to check = (current_index+1)^2 + (current_index+1)*3 + 2;
- This tends to reduce clustering
  - If a hashes to 1 and c hashes to 1, a will be placed at 1, c will be placed 12
    Furthermore, if a collision occurs at 2, the next check will be at 21
- Downside:
  - While faster in practice, quadratic probing cannot be guaranteed to find an empty box if the load factor is greater than .5 (50% full)

- Example of quadratic HT with probe (I+1)^2 + 1 on A-> 2, B->3, C->2, D->4 E->2

cannot be placed



$E \rightarrow 3^2 = 9 \% 6 = 3$

$4^4 = 16 \% 6 = 4$

$5^2 = 25 \% 6 = 1$

$2^2 = 4$

Double Hashing

- On collision, we use another (different) hashing function plus a "salt" to find the next box
- Example, hash1(x) = x^2 + x + 3; hash2(y, salt) = 5*y + 7*2*salt + 1
  - Salt is usually the index that hash1 produced
- Pros and cons are pretty similar to quadratic probing

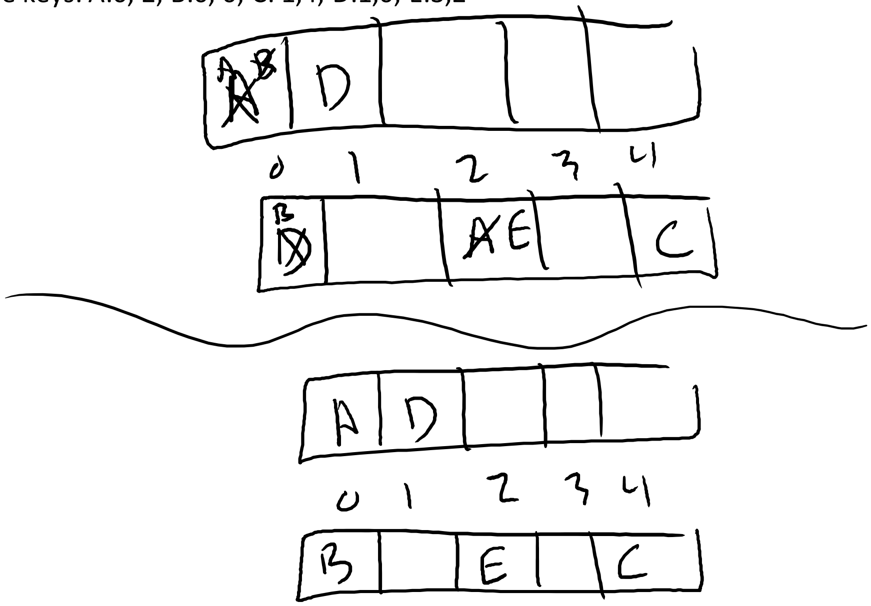
Modern approaches to Open Addressing

Cuckoo Hashing

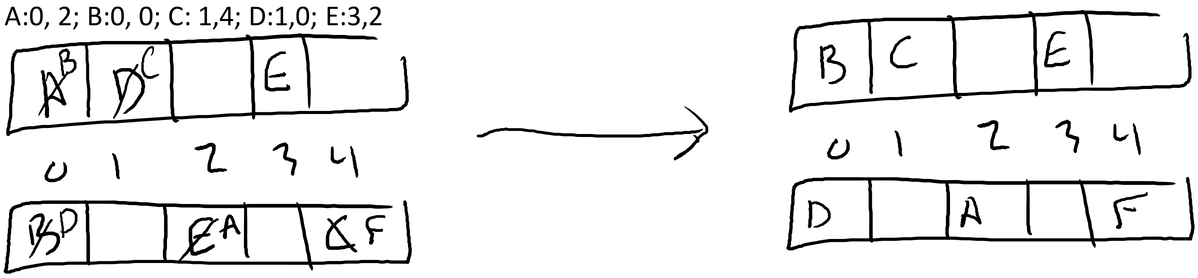
- First documented in 2001. Came about as a result of advances in probability theory.
- Similar to double hashing but instead utilizes parallel arrays

- Like double hashing, we have two hash functions for each array
- On 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

- Example keys: A:0, 2; B:0, 0; C: 1,4; D:1,0; E:3,2



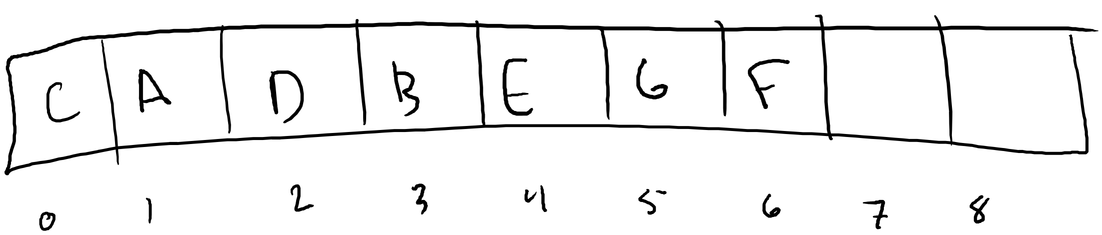
- What happens if we insert F:1,4 into the array?
- A:0, 2; B:0, 0; C: 1,4; D:1,0; E:3,2

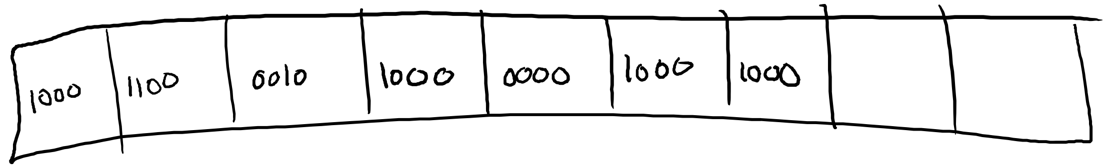


Hopscotch Hashing

- First paper published in 2009
- Similar to linear probing but it places a max limit on how for an item can be away from its original hash.
- Hopscotch hashing uses two parallel arrays: one for data, and one to track location of hashed item.
- Guarantees constant-time operations (new discovery)
  - Open question: is the extra bookkeeping of hopscotch hashing worth it?

- Example hopscotch hash table with max distance = 3



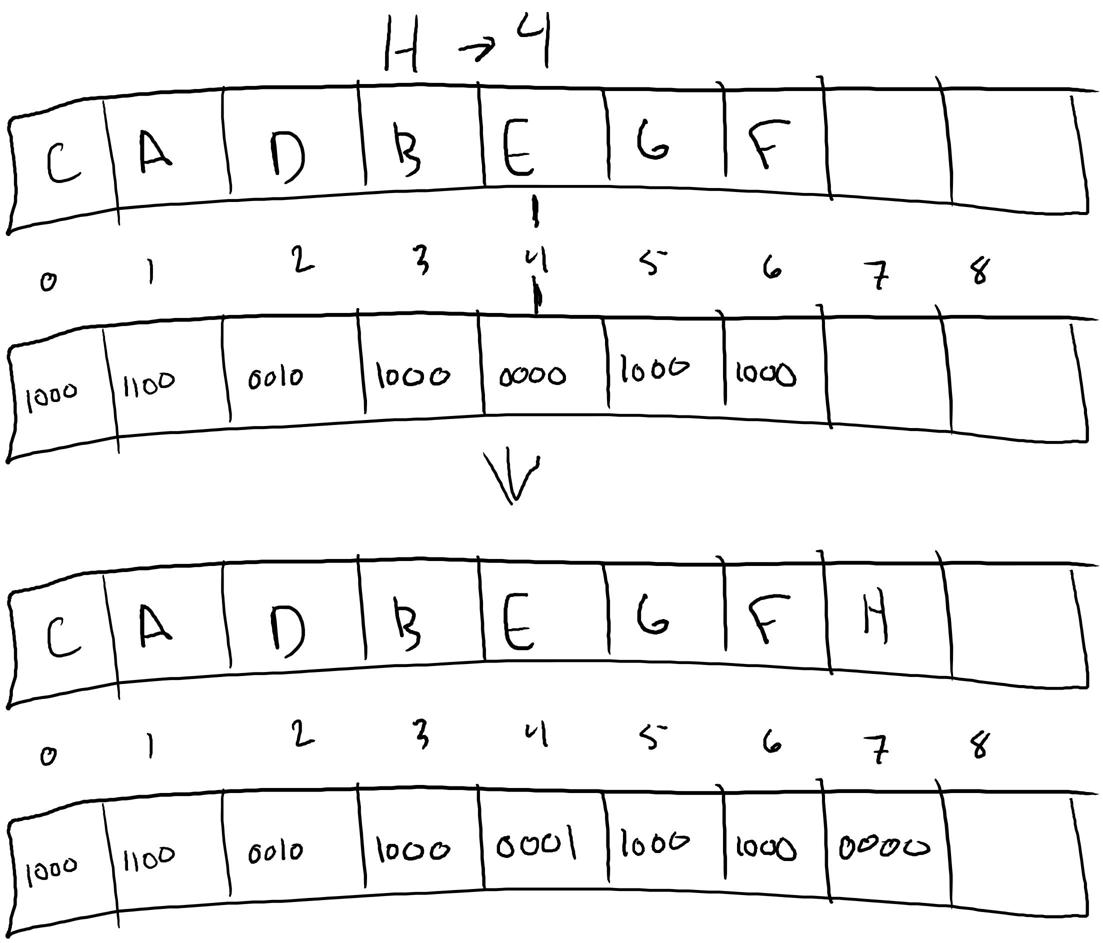


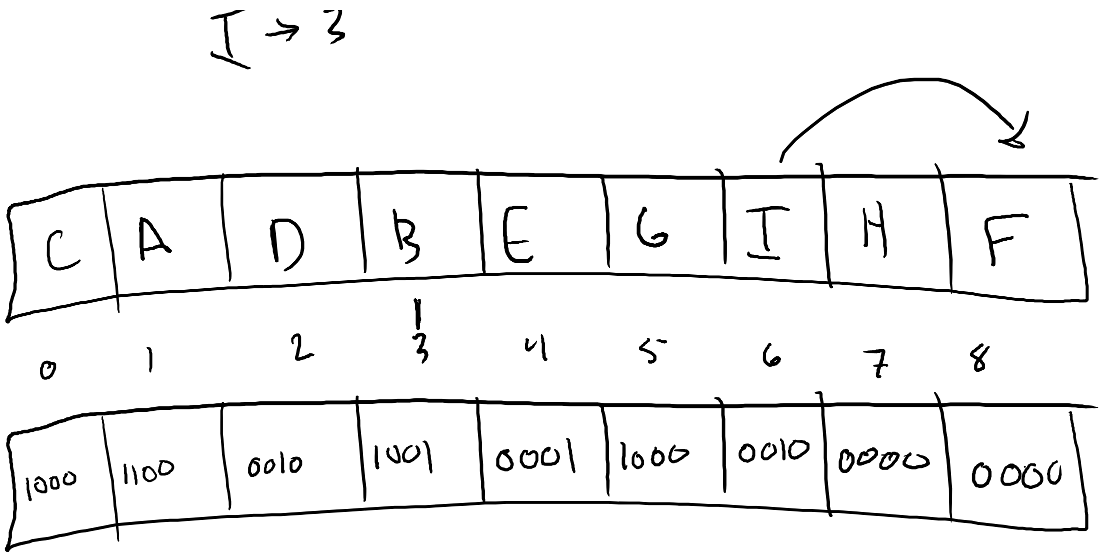
- Bits in the 2nd array track "distance from the origin"

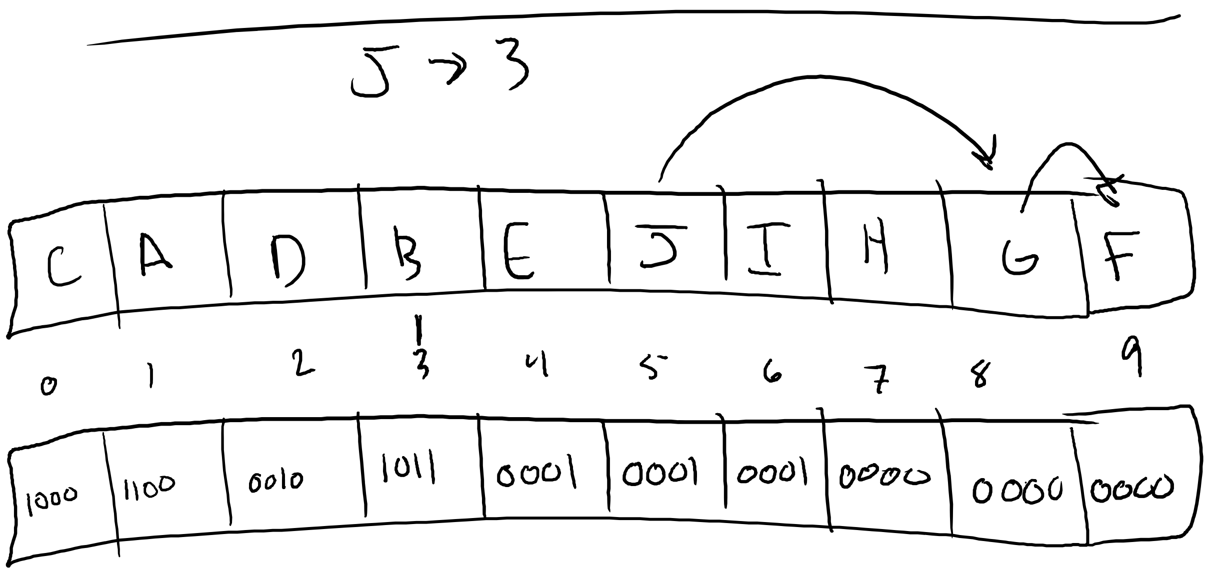
0/1	0/1	0/1	0/1
If 1, the item in the actual array hashes to this index	If 1, the item to the right of us hashes to this index	If 1, the item 2 spaces from us hashes to this index	If 1, the item 3 spaces from us hashes to this index

Algorithm

- If the original hashed box is empty, add to that box, update binary bits. Done.
- Otherwise, starting at the hashed value, try to find a box that is empty up to max distance. If successful, update the bits box.
- If all boxes up to max distance are occupied, find the first empty box up and work back to the hashed value.
  - During this process, try to move any value if possible to the empty box.
  - If this creates valid room, we're good. Otherwise, we couldn't find a spot so resize

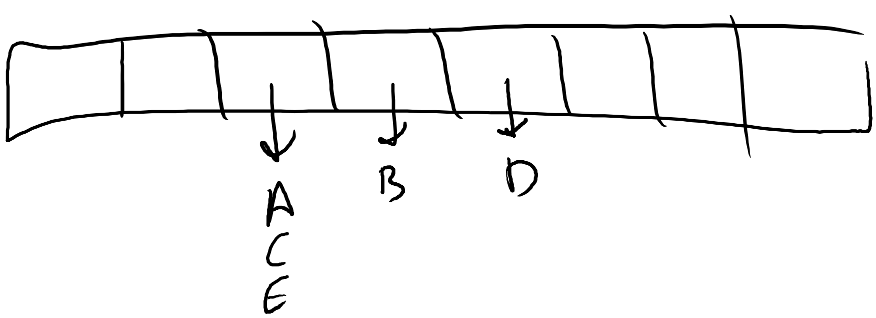






Separate Chaining

- The idea of having a single item in a box is an unnecessary limitation. Why not treat each box as a linear structure (e.g. vector or LL)?



- To add an item, simply go to the desired hash location, search linearly for the item.
  - If not found, push_back new item
  - If found, update value
- Visual C++ STL unordered_map uses this technique
- Implications

    - As each box takes on more elements (load factor goes up), performance slows down
    - Separate chaining can have load factors greater than 100%

Key Issue: How to remove items from a HT?

- Consider a linear probing HT w/ keys a->2, b->4, c->2, D->4

HT["C"]

- What would happen if I say HT.delete["A"]

- What would happen if I then try to access C?
- First thought, does removing trigger a rehash of all item?
    - Thankfully not!
- Instead, we perform a soft delete. We leave the key and data there but flag it as "deleted"

- Doing so allows us to recognize that C isn't at location 2 and that we must probe forward
- On resize, we delete instead of rehash soft deleted items.
- Documenting discussion: resize triggers a rehash of all items.