Lecture 14: Mini Midterm 2 (Robin Hood Hashing)

Sam McCauley

October 28, 2021

Williams College

Plan for today

- Mini midterm 2

- Then, finish up some previous topics

Mini-midterm 2: Hashing

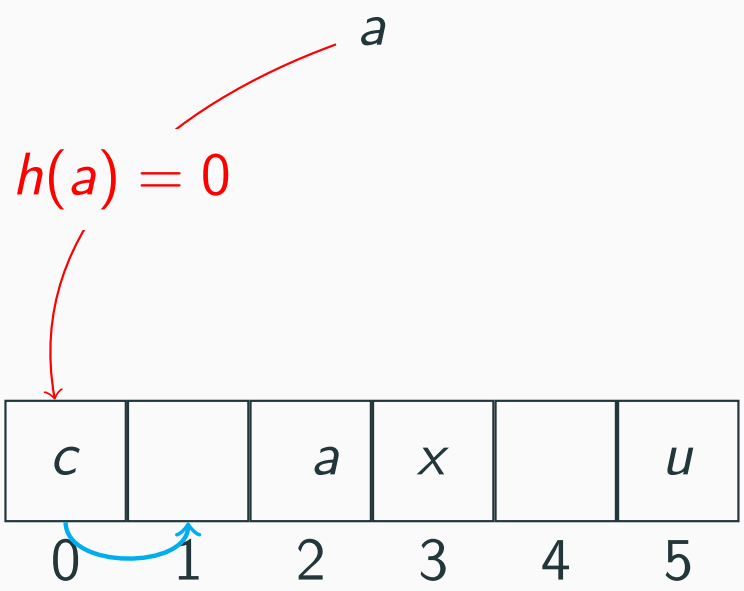
Idea for the midterm

- You'll be implementing a hash table (you'll be storing strings in the hash table)
- Using linear probing
- Put into place some common optimizations
- You may have implemented a hash table before; I'm asking for a specific methodology. Make sure you satisfy the constraints listed on the mini-midterm writeup.

Hashing with linear probing

- To resolve a collision on insert: find next empty slot
- Lookup: starting with hash position, compare to each slot until you find a matching item, or you find an empty slot
- No deletes on this assignment

Linear Probing Inserts



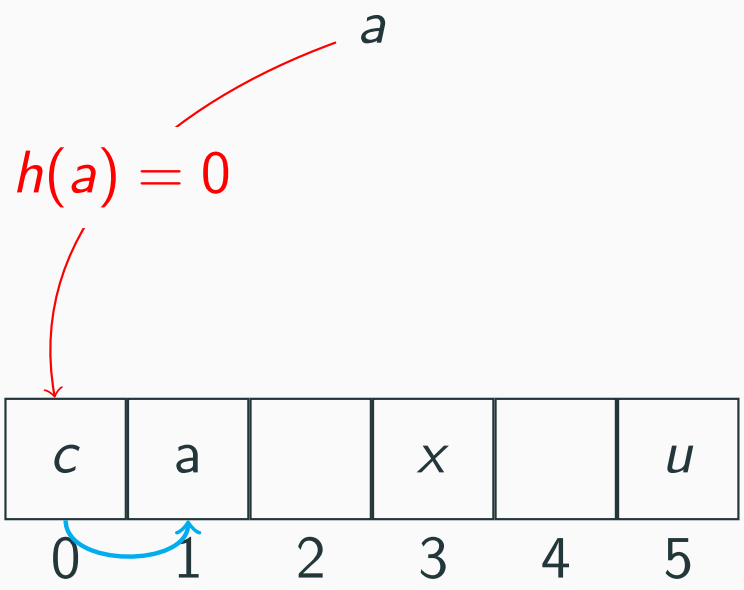
- Let's say we want to insert a new item a. How can we do that?
- Easy case: if $h_1(a)\%cn$ is free, can just store a immediately.
- Otherwise, look for next empty slot

Linear Probing Lookups

- What invariant can we guarantee about how items are stored?

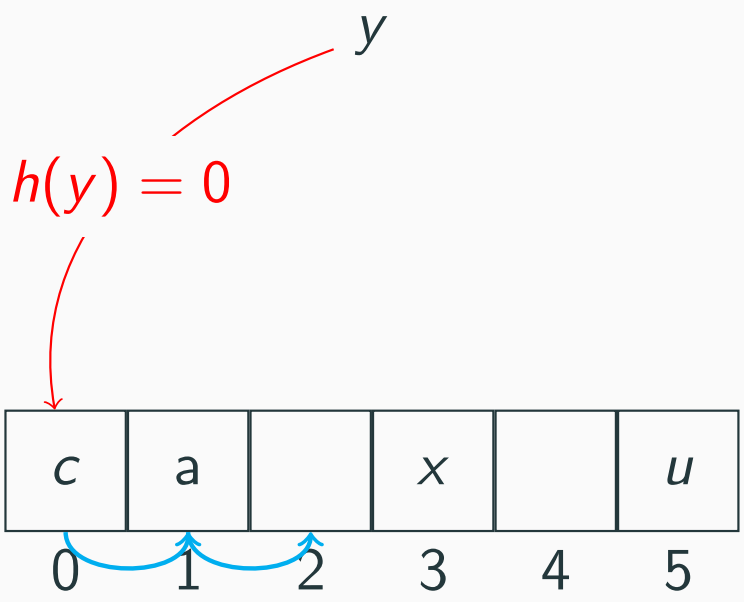
- (For now): Each item is stored in stored in a slot s such that all slots between s and its intended slot s_0 are full.

Linear Probing Lookups



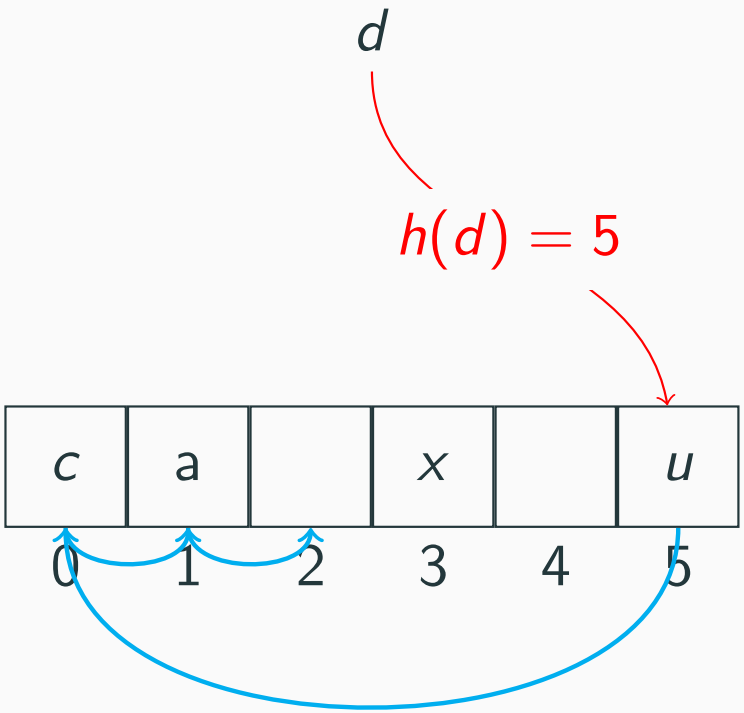
- Start with hash of element. Compare to element in that slot
- Keep comparing to element in successive slot. If find the query element, return 1 (element found)

Linear Probing Lookups



- Start with hash of element. Compare to element in that slot
- Keep comparing to element in successive slot. If find the query element, return 1 (element found)
- If find an empty slot, return 0 (element not found)

Insert falling off table



- What happens if there is no extra slot (we run out of table entries)?
- Answer for today: just loop around to the beginning!
- Invariant maintained, but "contiguous group of slots" wraps around the end of the array
- Lookups must also wrap around

Circular hash table

- Hopefully reasonably straightforward. Do lots of testing

- In mini-midterm: implement with if not mod
  - (If go off end, set variable to first in other side.)
  - I do think this is the way to go here for both speed and avoiding bugs; but in any case I'm asking all of you to do it this way for consistency.

Linear probing downsides

- Expected lookup time of linear probing is quite good
- Problem: lots of long runs, especially if table is very full
- Runs clump together to make longer runs!
- Our table in this mini-midterm will be 95% full; load factor $\alpha = .95$. (Unreasonably full.)
- Chaining avoids long runs, but has poor cache efficiency. Can we get the best of both worlds (or close to it?)

Why negative?

Expected time for a negative query:

Chaining: $O(1 + \alpha)$ Linear Probing: $O\left(1 + \frac{1}{(1-\alpha)^2}\right)$

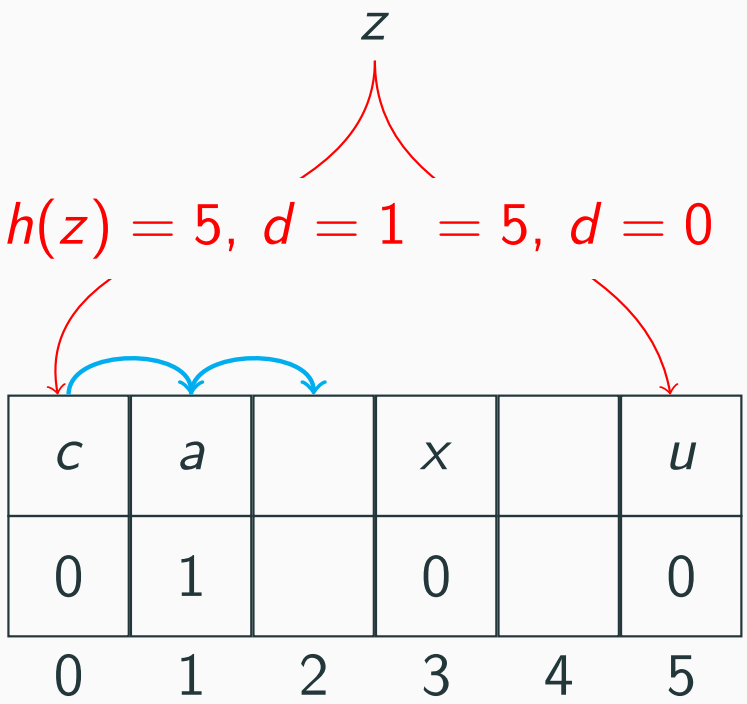
Wiggle room for linear probing

- We can maintain our linear probing invariant while improving our implementation
- Idea: we're always placing new elements at the end of the sequence of contiguous slots. Can we reorder elements to get better performance?
- In particular, some elements are really expensive (wind up far from their original slot). Can we balance this out a bit?
- Robin hood hashing: take from the cheap elements to give to expensive elements.
- In other words: put elements relatively close to their slot. Shift other elements down to make room (making each slightly more expensive)

Robin hood hashing details

- For each stored element, record its distance from its original slot
- When inserting, if find an element with smaller distance, insert our new element right away, shunting other elements down to make room.
- This increases the distance of those elements
- Let's do a quick example on the board
- New invariant?
- Elements are stored in order of their *original slot*

Robin hood hashing



Robin hood hashing

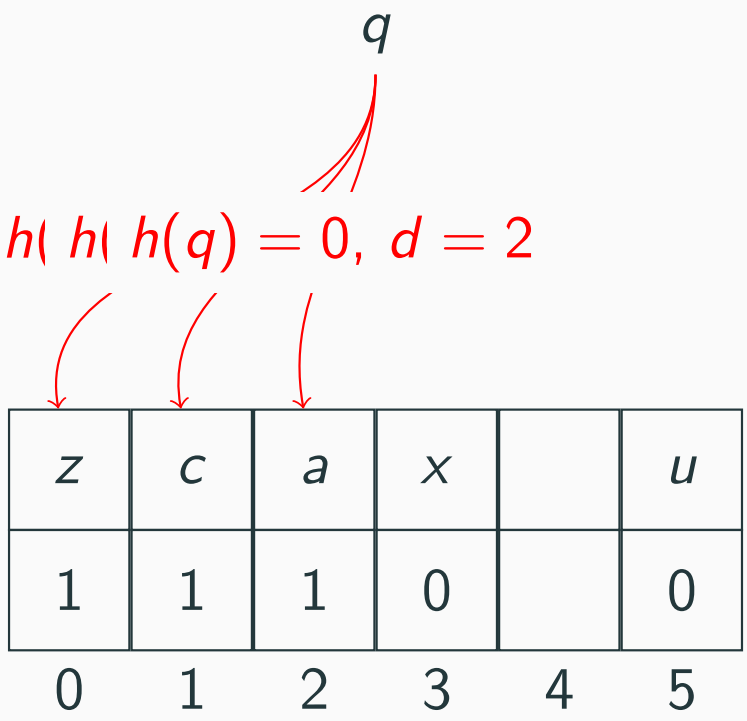
z	c	a	x		u
1	1	2	0		0
0	1	2	3	4	5

Robin hood hashing Lookups

z	*c*	*a*	*x*		*u*
1	1	2	0		0
0	1	2	3	4	5

- What happens when we query an element? Can we do better with our new information?
- Idea 1: only need to compare elements with matching distance
- Idea 2: if find an element with *smaller* distance than the query, can stop searching. (That element, and all elements after, hashed to a later slot.)

Robin hood hashing lookups



Don't need to compare q to z; they have different hashes

Compare q and c. They're not equal, so keep going

a must have $h(a) = 1$. That means that all further elements in this run of contiguous slots must have hashed to 1 or later, and we can return 0

Advantages of Robin Hood Hashing

- Worst-case is not so bad (shift elements to avoid worst-case lookups)
- Can also stop searches early
- Don't need to compare to all elements, only elements with matching hashes
- And: Inserts take same amount of time!
  - Elements we consider are exactly the same: all elements between the original slot we hashed to, and the next empty slot
  - Only disadvantage: need to shift elements down during inserts

Robin hood hashing performance

- Outperforms normal linear probing, especially when load factor is high

- Disadvantage: need to keep a little bit of distance information

Hashing on MM2

- You'll be storing 80-character DNA strings
- (expensive to compare!)
- Load factor .95

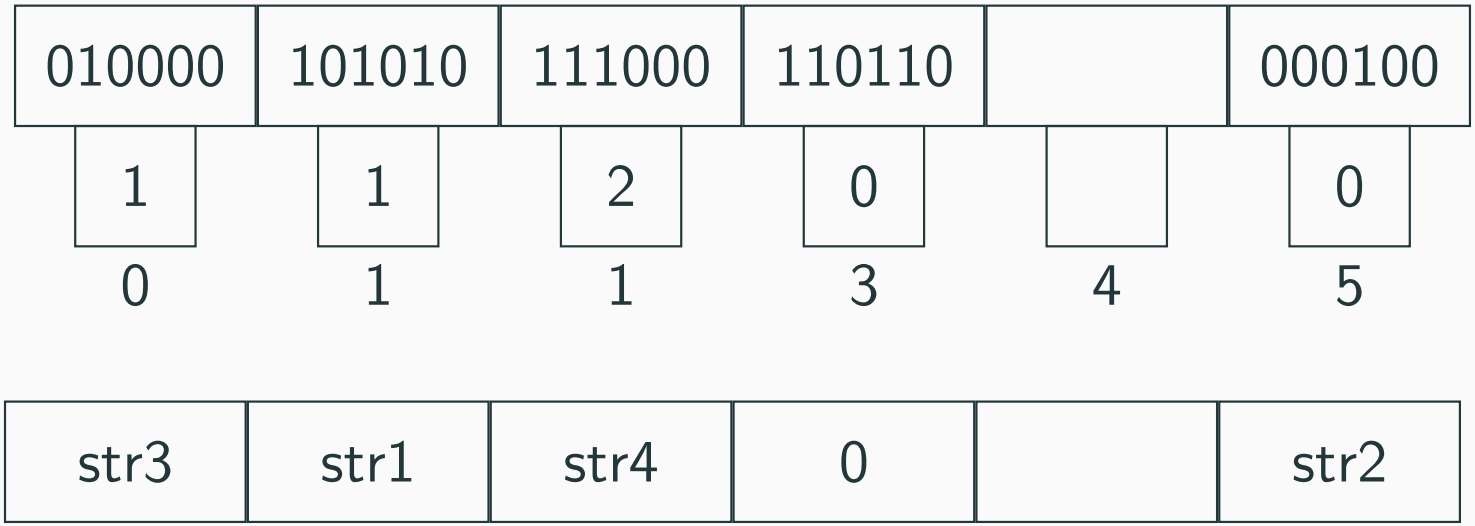
One last optimization

- Still need to do a decent number of comparisons in our use case, even for negative queries
- We also haven't talked about how to store distances
- How can we avoid comparing our long strings using hashing?
- New idea: store string *signatures*, compare only these signatures

Signatures

- Let's store a hash of each string in the table (a signature)
- Rather than comparing the whole string, compare the signature of the query to the signature of the stored element
- If the signatures don't match (cheap!), can move on. If the signatures match, do the expensive string comparison
- Hash table itself stores distances and signatures. Keep a second hash table with pointers to the original strings.

Hashing with Signatures



How do signatures change our algorithm?

- Basic idea of robin hood hashing stays the same: elements are still stored in the same places; lookups stay essentially the same
- On a lookup, when the distance matches, now have a two-part process to check if string is there. First, compare the signature. Only if the signature matches, compare the string (using the second hash table)
- Remember: when shifting items during an insert, need to shift in *both tables*

Storing table entries in MM2

- Each table entry is 32 bits
- Some number of bits per distance. (8 works well)
  - Needs to be changeable
  - OK if it's in the code, but needs to be stored as a variable. Can't do `uint8_t` for 8-bit distances for example.
  - This means: need to use bit operations, not structs, for table entries
- Remaining entry in each table is the signature.
  - Signatures should have length 32 − distance bits.

Other things to look out for

- Remember to increment distances when shifting
- Careful with circular array with distances and shifts! For both lookups and inserts.
- It's possible that distances will overflow the bits you allocate for them. In that case, it's OK if your hash table fails. But it should fail, rather than returning wrong answers (i.e. you should detect this case)

Let's Look at the mini-Midterm

Assignment 3 Wrapup Discussion

Separating out into functions

- No optimization criteria for Assignment 3
- So: cleanest solution is best solution. (Best for me/to maintain. But also, easiest for you to debug.)
- This is *always true*. But it is true that sometimes to write fast code, you should take clean code and make it less clean (e.g. by manually inlining functions)
- Let's look at one good example of a very very clean Assignment 3

Get and set in bin

```
int binGet(Filter* filter, uint64_t h, int binNum) {
    int bin = filter->table[h];
    int res = (bin >> (binNum * 8)) & ((1 << filter->fingerprintLength) - 1);

    // printf("binGet(filter, %ld, %d) = %x, bin = %x\n", h, binNum, res, bin);
    return res;
}

void binSet(Filter* filter, uint64_t h, int binNum, int f) {
    // printf("binSet(filter, %ld, %d, %d)\n", h, binNum, f);

    // resets bits binNum*8..binNum*8+7
    filter->table[h] &= (-1 - (((1 << filter->fingerprintLength) - 1) << (binNum * 8)))
;

    // adds the bits from f to binNum*8..binNum*8+7
    filter->table[h] |= f << (binNum * 8);
}
```

Bin Insert

```
// tries to insert into a bin; returns 0 if fails, 1 if succeeds
int binInsert(Filter* filter, uint64_t h, int f) {
    for (int i = 0; i < filter->binSize; i++) {
        if (!binGet(filter, h, i)) {
            binSet(filter, h, i, f);
            return 1;
        }
    }
    return 0;
}
```

Cuckoo

```
void cuckoo(Filter* filter, uint64_t h, int f, int depth) {
    if (depth >= filter->maxIter) {
        printf("MAX ITER REACHED. CUCKOO HAS FAILED.\n");
        return;
    }

    int toCuckoo = filter->toCuckoo[h];
    filter->toCuckoo[h]++;
    int evictedF = binGet(filter, h, toCuckoo);
    uint64_t newBin = h ^ (hashFingerprint[evictedF - 1] % (filter->numBins - 1) + 1);

    binSet(filter, h, toCuckoo, f);

    if (!binInsert(filter, newBin, evictedF)) {
        cuckoo(filter, newBin, evictedF, depth + 1);
    }
}
```

One takeaway

- Writing more modular code is often the best way to make your code easier to work with

- Superior to comments; can even be superior to simplifying expressions with intermediate variables.

One note on coding style

- Classic computer science problem (especially in C): how to gracefully exit a loop
- More difficult when you want to exit *two* loops (because `break` doesn't work).
- There are several ways to do this, and people have strong opinions about the benefits of each.
  - I don't think there's one right way to do things...
  - But I do think there are some specific suboptimal things to watch out for

Nested loop

```
void cuckoo() {
   for(int iteration = 0; iteration < maxIter;
       iteration++){
      for(int slot = 0; slot < 4; slot++) {
         if(bin[slot] == 0) {
           /* what do we want to do here? */
         }
      }
   }
   /*cuckoo failed if never hit the above*/
}
```

- Many of you had code that looks like this
- When find empty slot, want to:
  - Update bin with new fingerprint
  - Get out of the whole function

Nested loop: 1st solution

```
void cuckoo() {
   for(int iteration = 0; iteration < maxIter;
       iteration++){
      for(int slot = 0; slot < 4; slot++) {
         if(bin[slot] == 0) {
           bin[slot] = fingerprint;
           return;
         }
      }
   }
   printf("failed!")
}
```

- In my opinion the best solution by far
- Only downside: multiple return points for function

Nested loop: 2nd solution (avoid return)

```
void cuckoo() {
   int flag = 1;
   for(int iteration = 0; iteration < maxIter && flag;
       iteration++){
      for(int slot = 0; slot < 4 && flag; slot++) {
         if(bin[slot] == 0) {
           bin[slot] = fingerprint;
           flag = 1;
         }
      }
   }
   if(!flag)
      printf("failed!")
}
```

- Also OK (IMO inferior, but not problematic)

Nested loop: 3rd solution (too much IMO)

```
void cuckoo() {
int flag = 1;
for(int iteration = 0; iteration < maxIter; iteration ++){
    for(int slot = 0; slot < 4; slot ++) {
        if(bin[slot] == 0) { flag = 1; break;
        }
    }
    if( flag ) break;
}
if( flag )
    printf(" failed !")
else
  bin[slot] = fingerprint ;
}
```

- Messy; extra ifs; unclear when last line runs. (not ideal)

Point to bring home

- Sometimes getting out of a loop requires messy control flow
- May require things you usually want to avoid: `break`, flag variables, early returns, even `goto`
- Rule of thumb: use what's simplest; do the least possible
- Multiple exit points:
  - Definitely don't want to have unnecessary exit points
  - In my opinion, not worth writing more complex code to avoid multiple exit points (though I won't penalize for that alone)
  - More important: not worth doing extra work to avoid multiple exit points; execute code immediately when you know you can

Randomization Practice: Fisher Yates Shuffle

Practice with randomization

- For minHash we saw a way to generate a random permutation
- Let's prove that it works

The algorithm

```
void shuffle_array(int* array, int length) {
   for(int i = length - 1; i > 0; i--) {
      int j = rand() % (i+1);
      int temp = array[j];
      array[j] = array[i];
      array[i] = temp;
   }
}
```

What do we want to prove?

- What do we need to show that each permutation has the same probability of occurring?

- Idea: let's fix any permutation P. Show that the probability that P occurs is $1/n!$

Let's break it down

- What is the probability that the last item of the array matches what we wanted (in P)?
- Well, we only have one chance to get the last item right. (Once we set that item we never revisit it.)
- Probability: $1/n$.
- OK, let's say that we got that one. What's the probability that the second to last item matches P?
- $1/(n-1)$

The algorithm

```
void shuffle_array(int* array, int length) {
   for(int i = length - 1; i > 0; i--) {
      int j = rand() % (i+1);
      int temp = array[j];
      array[j] = array[i];
      array[i] = temp;
   }
}
```

Finishing the analysis

- The probability that the i to last item of P is correct is $1/(n-i)$.
- Multipying: permutation P occurs with probability $1/n!$
- Pretty cool!
- Lots of similar shuffling techniques don't work. The reason this one does: we select each item once, uniformly at random among all remaining items.