11/7/2025 Lecture: Hash Tables

Thursday, October 30, 2025 2:00 PM

With AVL Trees, we can insert items, delete items and search for items in O(lg n) time, where n is the number of items in the tree.

Can we do better?

Hash Table - expected O(1) performance for insert and search, and also delete (for one of the implementations)

Idea is as follows:

The table is long array, table size is p.
The table uses a function called a hash function.
A hash function is a many-to-one function. Output has to be an integer in the range of the size of the table (0 to p - 1). If it doesn't you can just mod by p.
A hash function takes in whatever items you're storing and returns an integer in range (0 to p - 1). The properties of a good hash function are:

1. Each output is equally likely.
2. Small changes in the input lead to unpredictable changes in the output
3. We want the probability of H(x) = H(y) for two randomly chosen strings x and y to be roughly 1/p.
4. Must be fast to compute.

Bad Hash Functions:

```
int hash(char* s) {
   int len = strlen(s);
   int res = 0;
   for (int i=0; i<len; i++)
      res = res + s[i];
   return res;
}
```

This is bad because all anagrams will give the same hash value. Also the sum of these values for regular words tends to be close to one another since ascii values are in such a small range.

Really bad:

```
int hash(char* s) {
   return s[0];
}
```

Let's say we're storing integers in the table of size 7, and our hash function is

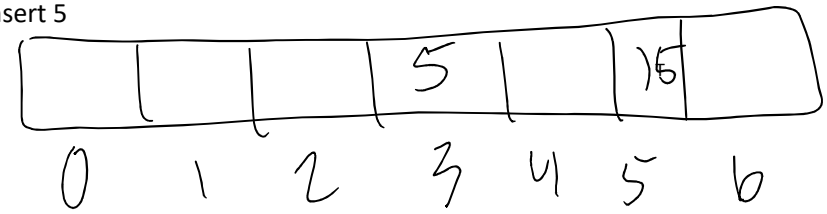
f(x) = (3x + 2) % 7

To insert something into the table, we calculate the input value's hash function and go to that index in the table.

$(3+5+2) = 17 \% 7$

Insert 5

Insert 5



Insert 15

To search for say 15, first calculate f(15) --> 5, then look in index 5.

Works great until???

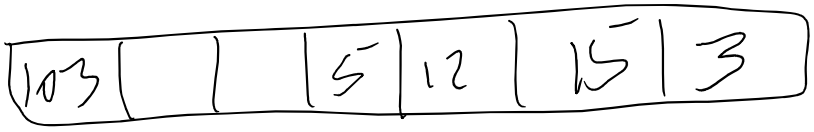
Insert 10 into the table… f(12) = (3*12 + 2)%7 = 38%7 = 3 --> ISSUE, 5 is already in slot 3…oops…COLLISION

A collision is when the index to insert a new item is the same as an index already storing an item in the table!!!

How to deal with collisions?

1) We don't, we evict the previous element in that index. (LOSSY)
2) Linear Probing
3) Quadratic Probing
4) Separate Chaining Hashing

Idea behind linear probing is that if a spot, idx is filled, then go to idx+1, then idx+2, … wrap around back to the beginning if necessary.



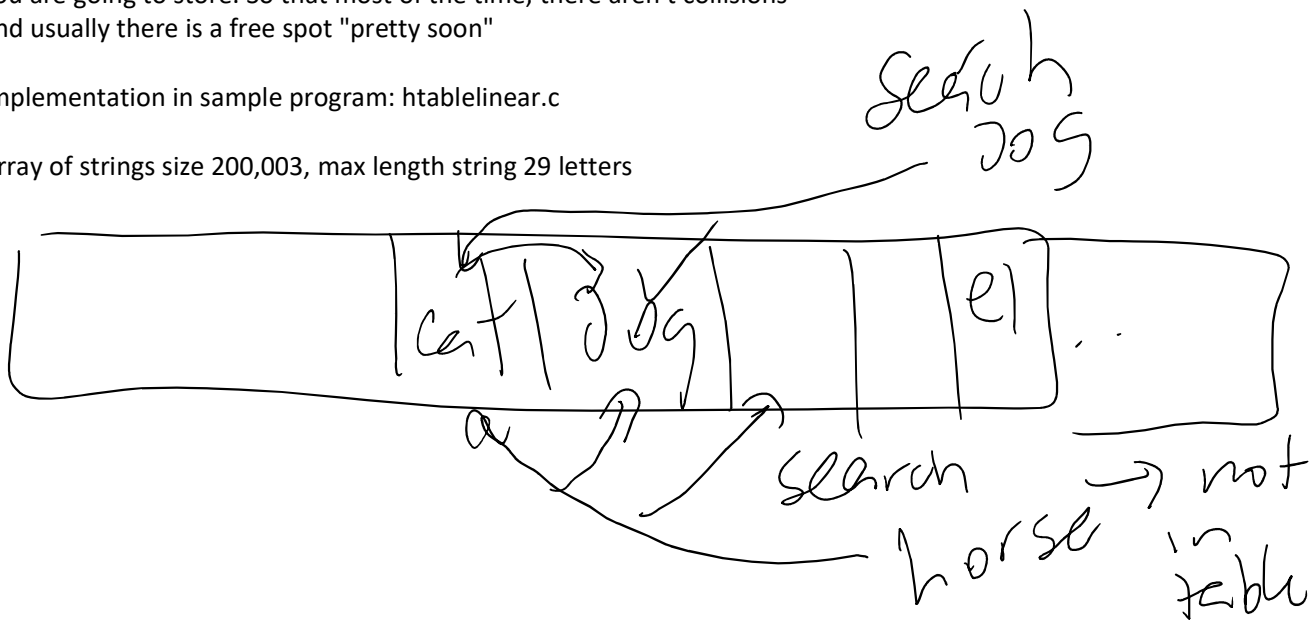
Search --> calculate hash value, then if that spot is full and the number is there, return yes it's in the table. If that spot is empty return it's not in the table. Otherwise, continue to the next spot over and over again until either you find the value or find an empty spot.

Question: How on earth is this thing O(1) expected run time???

Make the table large, at least twice as big as the number of elements you are going to store. So that most of the time, there aren't collisions and usually there is a free spot "pretty soon"

Implementation in sample program: htablelinear.c

Array of strings size 200,003, max length string 29 letters



This definitely works, but it has issues…in the last example, we had a ton of numbers that didn't necessarily have the same hash function consecutively placed in the array, causing really long search times if you hit that group. This idea is called clustering, and it tends to happen because of how linear probing works.

No matter where something hits in a cluster, you are guaranteed to grow it.

To avoid clustering, we must not look space by space, but instead, "jump" somehow.

The idea is quadratic probing.
In linear probing, the indexes we look at if the hash value is x are

x, x + 1, x + 2, x + 3, x + 4 …, (mod p)

In quadratic probing, the indexes we look at are:

x, x + 1, x + 4, x + 9, x + 16, …, x + i^2, in general
Gaps are +1, +3, + 5, +7

The complicating issue here is what if this pattern loops between the same spots and gets stuck and doesn't discover open spots.

Solution: Make the size of the table a prime number, p, and we can prove that the first (p-1)/2 spots that quadratic probing looks at are all different. If you know you are going to add upto n elements, then choose a table size that is a prime number that is at least 2*n + 1.

Assume to the contrary that x + i^2 and x + j^2, where i and j are not equal and both are 0 < i, j <= (p-1)/2, map to the same place in the

Assume to the contrary that $x + i^2$ and $x + j^2$, where i and j are not equal and both are $0 < i, j <= (p-1)/2$, map to the same place in the array, ie, their mod is equivalent mod p.

$x + i^2 \equiv x + j^2 \pmod p$

$(i^2 - j^2) \equiv 0 \pmod p$

$(i-j)(i+j) \equiv 0 \pmod p$

$p \mid ((i-j)(i+j))$, because $p \in$ prime

$p \mid (i-j)$ OR $p \mid (i+j)$

$i \neq j$.

$i + j \leq p-1$

$i + j > 0$

example

$f(x) = (x^2 + 3x + 5) \bmod 11$

insert 3, 6, 14, -5, 5, 16, ~~8~~

0	1	2	3	4	5	6	7	8	9	10
	3	14			6	-5	16			5

$f(3) = 23 \bmod 11 = 1$

$f(6) = 4$ $(36 + 18 + 5 = 59 \bmod 11 = 4)$

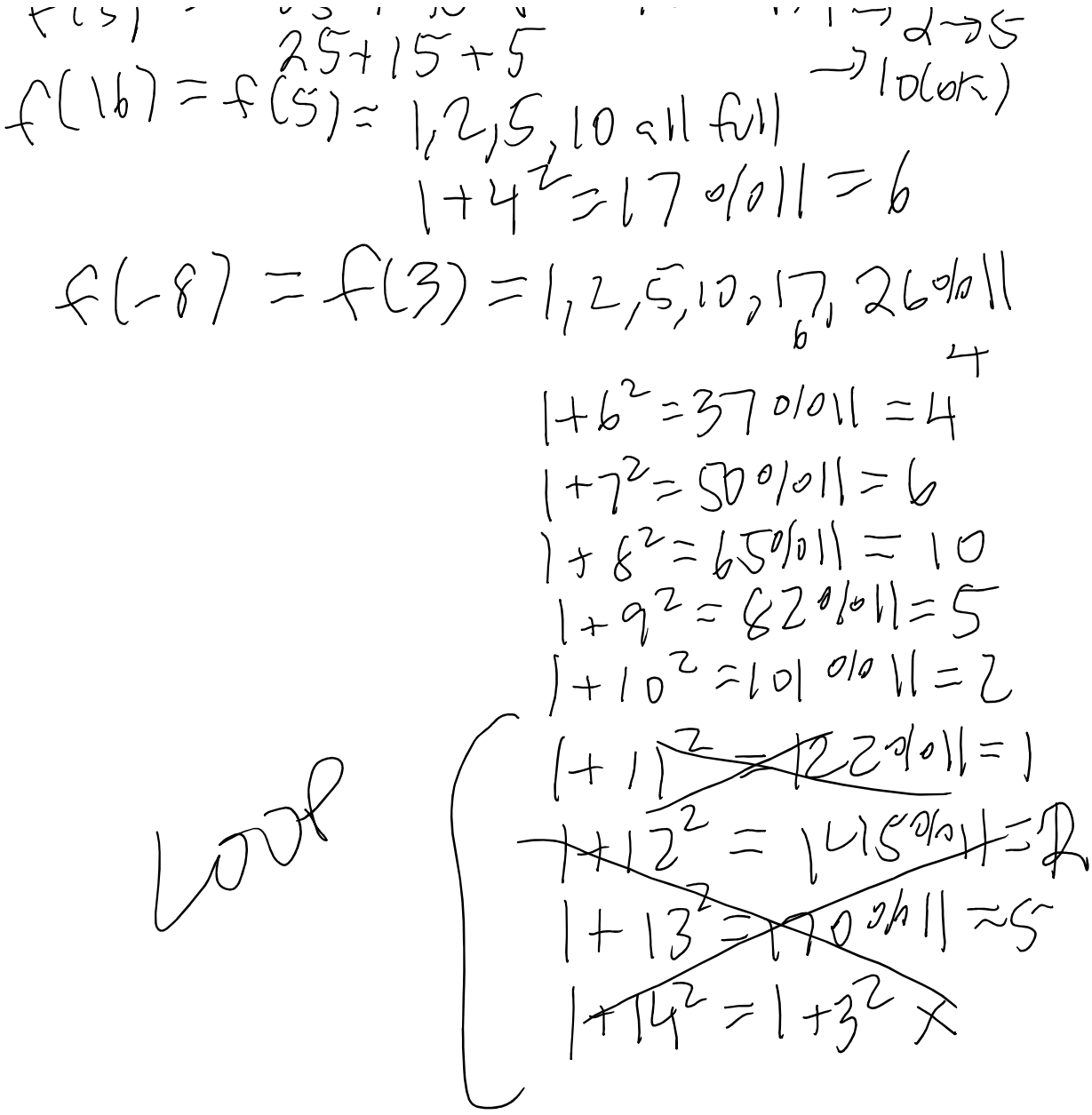
$f(14) = 1$ full $\rightarrow 2$

$f(-5) = 25 - 15 + 5 = 15 \bmod 11 = 4$ full $\rightarrow 5$

$f(5) = 45 \bmod 11 = 1 \rightarrow 2 \rightarrow 5$

$25 + 15 + 5$

$\rightarrow$ Infinite



Obvious idea: store multiple items at one array slot so you're not jumping around between slots hoping that you don't loop infinitely!

Separate Chaining Hashing:

Make each array slot a linked list, and then when searching, just go to the right index and for loop through the linked list…

Why do people do the other options???

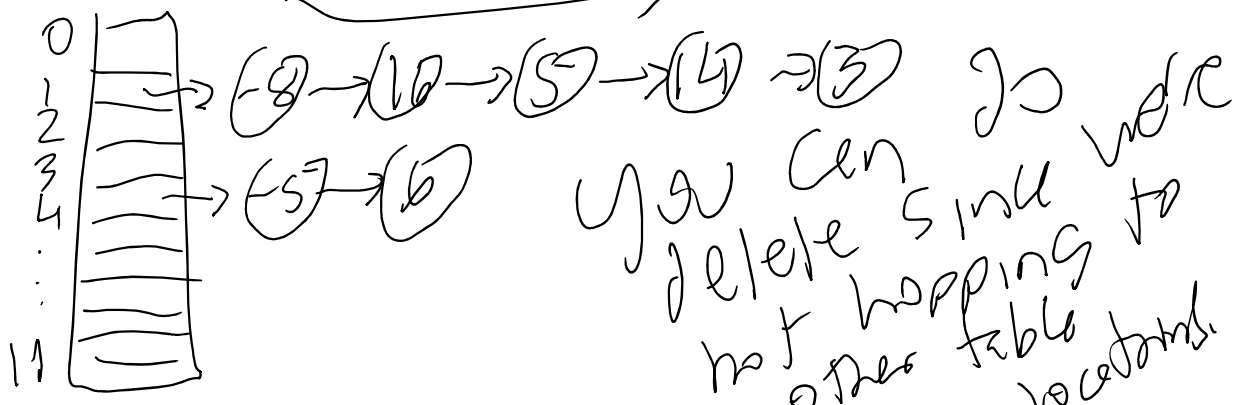
Answer: SPEED. Linked lists require dynamically allocating memory for each insertion, and the list sizes are small, but just managing the lists is a bunch of overhead that adding to array indexes is not. (Memory for each link…)

$f(x) = (x^2 + 3x + 5) \bmod 11$

$f(x) = (x^2 + 3x + 5) \bmod 11$

Insert: 3, 6, 14, −5, 5, 16, −8

3 → 1, 6 → 4, 14 → 1, −5 → 4, 5 → 1, 16 → 1, −8 → 1



You can do more delete simultaneously by hopping to other table locations.