CS280 Notes 3/2

by Christian Sagel

Questions:
How does the algorithm deal when child nodes are null? When I do rotations on rotate right/left, I am losing nodes or whole sections of the list when one of the subtrees (u,v) left/right children are null.
Make sure to pass pointers by reference or by pointer when performing operations.

Hashing

We will always have collisions.
Performance characteristics of Linear Probing
We put things into container so they are spread out, the hash functions will minimize collisions and the clusters we will have.

Probability of hitting and forming a cluster with 1 item in the table when inserting a second item, given equal probability: 3/N
Probability of forming a cluster when they are 1 block away from the present: 2/N
Adding a value to the table more than 2 spaces away. N-5 / N

Given a 3/N chance to hit a cluster when there's 2 nodes already. 3/N * 4/N = 12/N^2
When hitting the 2/N cluster. 2/N * 5/N = 10/N^2
When hitting the N-5/N cluster
With 2 items in the table, 6/N
When trying multiple combinations, we can break it down to an average increase over time.
Donald Knuth performed analysis to figure it out with some real expert++ math.

Hit: The average number of probes to find an item.
Miss: The average number of probes to discover an item doesn’t exist.
The load factor is the number of items in the tale divided by the table size.
Knuth-derived formulas (non-trivially) that show how probing directly affected.
We will never let the hash table get full..
If we keep ⅓ of the table empty, we will cap at 2 comparisons. If it gets past the load factor we grow the table and balance out. Most professionals also pick this ratio, the sweet spot.

Other probing methods

3 other types of collision resolution:
The fundamental problem with linear probing is that all of the probes trace the same sequence.
In linear probing, the stride is 1.

- Quadratic probing: The first time we have a collision, we move 1. The next time after that we will move 2 over. And so on we will be increasing the stride in a quadratic way.

We think of the clusters as being physically close to each other.
Logical clusters: We will see the clusters being formed in the similar pattern, even though they won't be as close to each other as they were before.
With quadratic probing we are not guaranteed to see all the slots, only 2/3rds.
There's tweaks to it.

- Pseudo-random probing: Probe by a random value, must use key as the seed to insure repeatability, because they are unique.
- Double hashing: Use another (different(hash function to determine the probe sequence. Hash function P(K) gives starting point (index into array). Probe function: S(k) gives the stride (offset).

Typically hash functions consist of bitwise operations like xoring, anding to maximize performance.

We only calculate the secondary hash if there is a collision, and we only do the secondary hash once, and we use that as the stride going forward.
There is no guarantee that I can hit everything with the stride.
We will solve it by making the table size a prime number so that its not evenly divisible, no repeating patterns. A support function that given a number gives you the next largest prime number.
We will wrap about everything.
With double hashing we can get to 95% full with 3 .15 comparisons.

Mead will show us some professional grade hash functions that are used by Google, etc.
With small tables, linear probing can't be beat. (Similar to bubble sort.
With 99% full, we have up to 4.64 average comparisons.
Trees doing 11 comparisons with 10 levels.
On an openly-addressed table, we may hit 2-3 at most.

Expanding the Hash Table. (Dynamic Hash Tables)

The performance of the hash table algorithms depend on the load factor of the table.
Tables must not get full (or near full) or performance degrades.
Most compilers do 1.5 - 2x the size.

WIth a hash table we have to reinsert when growing the array.
Once a blue moon it may take some unlucky person 4 minutes to get the ATM to service them because the hash table is growing.
Run-time issues: Performance profile is erratic. Worst case is pretty bad, but rare and acceptable in certain situations. Can also “shrink” if it becomes too parse.

Using hash tables unwisely can get peopled killed.

Deleting items from a linear probing hash table
Deleting an item from a cluster presents a problem as the deleted item could be part of a linear probing sequence.

Handling deletions: Policies

1. Mark the slots as deleted (MARK). 3 states: occupied, unoccupied, deleted. The insertion/search algorithm will insert at the first deleted/unoccupied slot to reuse it. Also known as lazy deletion. Marking data as deleted be used by other data structures (e.g: balanced trees).

In double hashing there’s no way you know where your clusters are. We will mark them as deleted with open-addressing. We also don’t do double hashing with closed-addressing.
2. Adjusting the table (PACK). For each item after the deleted item that’s in the cluster, mark its slot as unoccupied and Insert it back into the table. On average, even with a million items in the table we only have to reinsert only a few items into the table. (Allowing for us not letting the table get full!)

The difference between linear probing and double hashing is 1 line. All that changes is setting the stride. If (double hashing) GetStrideFunc(), otherwise set it to 1.
We grow the table before inserting the new node when we calculate the load factor.
We will have to reinsert when growing the table, first walking through the known array.
In the grow function We would keep a temporary pointer to the current table, then insert into the new one.
In the MARK deletion method, we always walk to the end of the cluster to find if the item as there (because we have to overwrite it if need be). When reaching the end, we just jump back to the first deleted/unoccupied node and insert it back there .
The constructor will decide what size it is.