CS280 Notes 3/7

by Christian Sagel

Counting nodes: When do we add counts. I try adding counts before I call recursive insert into left/right nodes, but end up with 1 more. When should I be incrementing the count?

The subscript is in inorder traversal.
Whenever we rotate with AVL, we have to recount the nodes.

Open-Addressing Hash Table
Today we will be talking about a lot of hash functions.
Things are stored in the table with a hash value.
When the hash value matches the slot, it means its in the right position. If it's not, there was a collision for it to get there.
Mead's demo application is pretty nice! It only took him 4 hours to do the application.

Collision Resolution by Chaining

This is the one we will be using for the homework.
While in open-addressing scheme the data is stored in the hash table, with chaining the data is stored outside of the hash table. This is called chaining. Instead of storing items in the hash table (in the slot indexed by the hashed key) we store them on a single linked list.
The hash table simply contains pointers to the first item in each list.
We will still have collisions (what we do with them determines the efficiency)
If we keep it sorted, we don't have to walk the list if we walk past the possible index. We may get a little performance increase, but it may not be worth it because the lists won't get big enough for sorting to matter much.
We will insert to the front (because users don't have control, we choose the front because it's more efficient).
When you add in the homework, do it to the front even if the example does otherwise!
Once we find that's it not on the list, we can insert to the front of the list.
We will keep these lists short enough.
We could have splay (caching) tables by moving items to the front of the linked list after looking them up, because its a constant time operating to swap pointers.
It will be more expensive because we do more allocations per node (alleviated because we use an object allocator). Implementing insert/delete is trivial.
Complexity depends on the hash function. Poor hash functions give O(N), while good ones O(N/M)

Complexity of chaining

With chaining, there is no concept of ⅔ full. Load factor can be calculated as the average length of the lists (given a good hash function).
WIth chaining we will see the
When the average load factor of the lists gets past, we will grow the hash table vertically to the next larger prime number, and re-insert into the new table.
We will see clusters when they grow horizontally to the right.
Load factor tells us how many comparisons we have on average.
Growth factor 1.5, Load Factor 2.0. When growing, multiply by the growth factor and round up to the nearest prime.
Remember: In the homework after getting to the back of the list to look for a duplicate, we insert to the front.
With 4 items, 3 lists the LF is 1.33.
When growing the tables, we reinsert all existing items into the new table with new hashes. We will iterate over the table, one list at a time.
When you are reinserting items into the table, you do not have to check for duplicates. (There can be no duplicates!) Do not walk the whole list while adding.
When adding new items into the table,
You are also reusing nodes. When moving the items into the new table, hook those pointers in instead of having to make new nodes.

If the TA sees two different functions for inserting on a new item/reinsert, we will be docked hard. Just have a flag on the method or a member on the class to tell it what inserting it should use.
Remember, even though we don't free the nodes, we do have to free the previous table.
With chaining, the performance degrades much more gracefully.
The lists are managed by structs have a count of how many nodes are on the list.
We are going to keep track of the number of probes. If we can, have a probes variable and increment only once in the class.
We are going to be using null-terminated strings for all our keys.
We will pass in a key and the size of the table, we will get an index back.
Check that FreeProc exists before blindly calling it when we are about to free the data. So before we delete the node we will pass the node's data to FreeProc, then we will delete the node.

3 main methods:

- Insert: Throws exception if we find a duplicate
- Remove:
- Clear: It will remove all nodes from every list, and reinitialize the lists. (Not destroy them!)

We don't really need to know how the hash functions work in order to use them, just know how to call them.
Once nodes have been created, we will move them to newly allocated lists.

If we delete an item, we can delete the node (because the allocator will recycle it)
Use SLLs.

Hashing Strings

Let’s make hashing great again. - Mead
Many algorithms exist for hashing non-numeric keys.
CRC: Cyclic redundancy check algorithms can hash entire files. Turning characters and scrambling them.
You can’t reverse engineer passwords encrypted through hashes
With hashing small changes in the input will lead to drastically different outputs.
Hash functions are typically more math than programming.
Divisions and modulo are the most expensive.
The best hash functions do only bitwise operations.

PJWWash: What the math ?????
We want hash functions to be fast and provide good distribution.