CSCI 235 Class Notes: Hashing

1. Problem: How do you implement a spell checker using a lookup in a dictionary file. For example, there can be thousands of words in a typical on-line dictionary. The basic operation is to see if a word is in the list of legal words or not - a classic **SEARCH** operation.

   - Sequential search is slow. O(N) operations for each word in our file.
   - What if the dictionary of words is sorted? We can improve on this using binary search which uses O(Log N) operations.

2. Can we do better? Maybe, with **HASHING**.

   - Hashing can deliver constant time access - O(1) operations to find an entry.
   - Hash function H maps keys to addresses (i.e. array indices)
     H: Keys $\rightarrow$ Array indices
   - Used when the set of possible keys is much greater than the set of addresses needed.
   - Example: Students at Hunter, ID = SSN. 10^9 possible SSN, yet only 20,000 students at Hunter. If we use a hash function: H(SSN) = SSN % 20,000 yields an array index in the range 0-19,999
   - Example: Words (up to 10 letters long) in a dictionary have $26^{10} + 26^9 + 26^8 + ...26$ possible keys, but we only need to store a small subset of these possible keys at any one time, since most combinations are not actual words. Can we do a fast access yet distribute these words in an array evenly?
   - However, there may be 2 or more keys that map to the same index! This is know as a $collision$. We need to find a method to handle this.

3. Nomenclature

   - **Table Size** is the number of locations in the Hash Table. Each slot in the array is often called a Bucket. The actual size of the array that constitutes the hash table.
   - **Collision**: two keys ”hash” to the same index.
   - **Load Factor** is defined to be the ratio of number of items N to be hashed over Table Size B (N/B).
   - Each entry in the hash table is sometimes called a **Bucket**.
   - A Hash table is sometimes referred to as **Scatter Table**.

4. Criteria for a good Hash function: 1) Fast and easy to compute and 2) Distributes keys evenly in the array. There are really only 2 decisions in hashing: choosing a Hash function and a collision policy

5. If the key is an integer, a simple hash function is to just take the key modulo the table size:
   KEY % TABLESIZE
   Note: We usually try to make the table size, or number of Buckets, a prime number. This prevents anomalies that cause collisions, and it can help in distributing the keys after collisions.

6. What if key is a string and not a number (as in the dictonary example)? We can use a simple hash function to compute the index from the string such as adding up character codes.

```
/* returns index based on string’s character code values */

const int BUCKET_SIZE = 5

int Badhashfunction(String word)
{
   int sum = 0;
   int len = word.length();
   for (int i=0; i<len; i++)
     sum += (int)word[i];
   return sum % BUCKET_SIZE   /* B is Table Size - number of Buckets *
}
Example of function on words below using a table size of 5:

        Word        Sum        Bucket
        -----------------------------
        anyone      650        0
        lived       532        2
        in          215        0
        a            97        2
        pretty      680        0
        how         334        4
        town        456        1
```

   Note: the above hash function is bad when we have a large number of words to hash.

7. **Hash Method I: Separate Chaining** (also called Direct Chaining). Each bucket entry contains a linked list pointer that points to all entries who hash to this bucket.

   - Find: H(key) = index, and table[index] is a pointer to linked list of entries that all hash to this location. Can be sequentially searched at this point.
   - Insert: H(key) = index and table[index] is a pointer to linked list of entries that all hash to this location. If NULL pointer, insert his item, else insert using standard Linked List insertion.
   - Delete: H(key) = index and table[index] is a pointer to linked list of entries that all hash to this location. Now use Linked List **find** method to delete this item.

8. Running time of Separate Chaining Hash operations. Assuming the hash function you use is uniform and random (it distributes the keys equally), running time is O(1) to compute index, and O(N/B) to find element where N= number of elements in hash table and B= Table Size. If N=B, O(N/B) = O(1), and we get good performance. If B is smaller than N (fewer buckets than items to insert) then the sequential search of the linked list becomes longer as the load factor is greater than 1. Note that increasing B $>$ N does not significantly help performance.

9. **Hash Method II: Open Addressing with Linear Probing**. Think of the Hash table as a big circular queue (implemented as a “wrap-around” array). Hash function gets you to to the bucket where the element is, and then you do a sequential search (**probes**) from this point in the table to find the element.

   Assume a Table size of 2* Number of elements (this leaves some open space around an item to be inserted). To insert an item we hash it. If there is already a table entry at that spot we continue by increasing the index by 1 until we find a) an empty slot, or b) we return to the start position.

   Probe sequence is: hash(key), hash(key) + incr, hash(key) + 2 * incr, hash(key) + 3 * incr, ... , etc. In linear probing, we use an increment of 1 each time.

```
pseudocode for find using linear probing
find(key)
    try = H = hash (key)
    i=1 /*increment per linear probe */
    do
       if table[try] == 0  /* empty slot at key index */
           report "not found"
       else if table[try]==key
           report "found"
       try = (try+i)MOD B /*B= Table Size*/
    while try!=H /*prevents returning to the beginning*/
```

10. Problem with linear probing: Clustering in places where keys map. Can become sequential search.

11. H(key) = 0: this is a hash function that creates linear search!

12. If table gets near full, we increase search time. Solution: **Rehash** - create new table twice as big. To rehash, we simply recalculate the table entries with new value of B cost is O(N), but we do it only once!

13. **Double Hashing** In this method, we use another function to choose the probe increment, so different keys probe different amounts to “spread” the data around the hash table. For example, if we have a hash function:

$$H(Key) \; = \; Key \,\%\, B$$

where B is the number of entries in the table, we can also define a probe increment for this key as:

$$ProbeIncr(Key) \;=\; Max(1, (Key/B)\,\%\,B)$$

So depending upon the key, we probe a different amount each time to spread the data around if their is a collision.

An important question with double hashing is this: does the probe increment guarantee that we will eventually find an open spot if an open spot exists? It turns out that if the probe increment and the table size are relatively prime, then you can prove that the probe sequence will in fact cover the entire table (see p. 476 in Standish). So by choosing a table size B that is a prime number, and choosing a probe increment that is modulo B, we guarantee a useful and complete probe sequence.

14. **Quadratic Probing** spreads out search. However, must guarantee that you can eventually find an empty spot for an insertion. You can prove this will be the case if table size is prime and less than half full,

15. Deletions in open addressing are a problem. An open spot breaks the continuity of a search for a key. Lazy deletion is often used instead.

16. Other methods of generating a Hash Function

    - **Folding:** Take a multiple digit key and break it into groups of digits, and then treat these groups as the entities to be used in calculating the index. If key is 123456789, break it into 3 groups of 3 digit numbers 123 - 456 - 789 and add them up, or possibly multiply each group by a base number.
    - **Mid-Squaring:** Take a multiple digit key, choose the middle digits and square them to get some randomization (scatter) in the table.

17. Obviously, combinations of these techniques can be used to create a hash function. Keep in mind the design goals: fast to compute, scatters the keys in the table.