CS 3137 Class Notes

1 What is Hashing?

1. Problem: How do you implement a spell checker using a lookup in a dictionary file. For example, there are about 25,000 words in /usr/dict/words 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. It takes O(N) operations to find each word in our file.
   - What if /usr/dict/words is sorted (it is)? We can improve on this using binary search which uses O(Log N) operations.
2. Can we do better? Yes, 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
   - Typically Used when the set of possible keys is much greater than the set of addresses needed.
   - Example: Students at Columbia, ID = SSN. 10^9 possible SSN, yet only 20,000 students at Columbia. 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 known 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 sometimes 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).
   - A Hash table is sometimes referred to as **Scatter Table**. This is because we are trying to scatter the data throughout the table.
4. Criteria for a good Hash function: 1) Fast and easy to compute and 2) Distributes keys evenly in the array.
5. There are really only 3 decisions you need to make in hashing: 1) choosing a Hash function, 2) choosing a collision policy and 3) choosing a table size.
6. If key is an integer, a simple hash function is to take the key modulo the table size:$Key \,\%\, Tablesize$
7. Why Use a Prime Number for Table Size?
   - When choosing a table size, you should always choose a prime number larger than the desired table size. There are a number of reasons for this.
   - We compute the index as $index = Key \, mod \, N$. If the table size is an even number, and the key is even, then the index will be an even number. This can bias the entries in the table. A common practice in hashing is to use the object's computer address as part of the key, and these addresses can often begin on an even number (an address is often 4 bytes long on many machines.). Therefore only half the table slots may be used - no odd numbered indices will be computed.

- If we use a power of 2 as the table size, we are effectively just using the low order bits of the key, since Modular division by a power of 2 leaves us with just the low order (least significant) bits of a number. This may not be desirable. Note: We usually try to make the table size a prime number. This prevents anomalies that cause collisions, and it can help in distributing the keys after collisions.

8. 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 */

final int TABLE_SIZE = 5

int Badhashfunction(String word)
{
   int sum = 0;
   int len = word.length();
   for (int i=0, i<len; i++)
     sum += (int)word.charAt(i);
   return sum % TABLE_SIZE   /* 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. Why?

9. Here is a better method using polynomial coefficients(see Weiss, p. 174). This method spreads the strings better throughout the table by multiplying each character by a polynomial base value. The algorithm is:

$$Index = \sum_{i=0}^{len-1} c_{len-i-1} \cdot a^i$$

where c_i is character code at position i in a string of length len and a is the base value of the polynomial. Typically an odd base number such as $a = 37$ is used. For example, the word *card* would be computed as:

$$H = 'c' * 37^3 + 'a' * 37^2 + 'r' * 37^1 + 'd' * 37^0$$

Note that we can compute a polynomial like this very efficiently without using the relatively slow Math.pow() function using Horner's method.

$$H = ((('c') * 37 + 'a') * 37 +' r') * 37 + 'd'$$

```
/** test polynomial multiplier p. 174 Weiss for hash function on
     /usr/dict/words (contains 25143 words to be hashed */
import java.io.*;
import java.util.*;
import java.text.*;

public class hashweiss {
    private final int tableSize = 25147; //first  prime after 25143
    int index,collisions;
    int[] hashtable = new int[tableSize];

    public void performTest() throws Exception {
        for(int index=0;index<tableSize; index++)
           hashtable[index]=0;

        BufferedReader diskInput;
        String key;
        diskInput = new BufferedReader (new InputStreamReader (
                                          new FileInputStream (
                                          new File ("/usr/dict/words"))));
        key = diskInput.readLine(); //read next word
        while (key != null) {
          index= this.hash(key,tableSize);
          if(hashtable[index]>0) { // is it a collision?
              collisions++;
              System.out.println("word=" + key +  " index=" + index +
                                 " collisions=" + collisions);
          }
          hashtable[index]++;
          key = diskInput.readLine();
        }
    }

    public  int hash(String key, int tableSize){

        int hashVal = 0;  //uses Horner's method to evaluate a polynomial
        for( int i = 0; i < key.length( ); i++ )
           hashVal = 37 * hashVal + key.charAt( i );

        hashVal %= tableSize;
        if( hashVal < 0 )
           hashVal += tableSize; //needed if hashVal is negative
        return hashVal;
    }


    public static void main(String[] args) throws Exception {
        hashweiss app = new hashweiss();
        app.performTest();
    }
}
```

2 A Hash Collision Experiment

If we increase TableSize, we have more places to store the indices, and we reduce our load factor, usually lessening collisions.

```
/** An application that determines the number of collisions among
    25,143 distinct words in /usr/dict/words using the hashCode()
    function of class String and hash tables of increasing size.  The
    table size is increased by 25,143 each time (we use the next prime
    number larger than this value for the actual table size) */

import java.io.*;
public class CollisionExperiment {
    // Fields
    private final int NUMBER_WORDS = 25143;
    private final int MAX_PRIMES = 500000;
    String [] table;

    public void computeCollisions () {
        int hashIndex = 0;
        try {
            System.out.println ("TableSize\tCollisions");
            System.out.println ("---------" + "\t" + "----------");
            // generate proime numbers
            PrimeGenerator pr= new PrimeGenerator(MAX_PRIMES);
            pr.computePrimes();
            for (int size = pr.get_next_prime(NUMBER_WORDS);
                size<10*NUMBER_WORDS;
                    size=pr.get_next_prime(size+=NUMBER_WORDS)){

              table = new String[size];
              BufferedReader diskInput = new BufferedReader (
                                      new InputStreamReader (
                                      new FileInputStream (
                                      new File("/usr/dict/words"))));
 // Insert each line of disk input file into hash table
                int collisions = 0;
                String line = diskInput.readLine();
                while (line != null && line.length() > 0) {
                    int hx=line.hashCode();
                    hashIndex = Math.abs(hx) % size;
                    if (table[hashIndex] == null)
                        table[hashIndex] = line;
                    else
                        collisions++;
                    line = diskInput.readLine();
                }   //end while
                System.out.println (size + "\t\t" + collisions);
             } //end for
        }
        catch (Exception ex) {
        System.out.println("error!");}
    }
    static public void main (String[] args)  {
        CollisionExperiment test = new CollisionExperiment();
        test.computeCollisions();
    }
}
```

```
TableSize       Collisions
---------       ----------
25147           9215
50291           5345
75437           3723
100591          2895
125737          2388
150881          1937
176041          1782
201193          1470
226337          1305
```

3 Hashing Methods

1. **Hash Method I: Separate Chaining** (also called Direct Chaining). Each table 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, make a new linked list pointing from this table entry and insert the 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.
2. 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 (this is the *load factor*. If N=B, O(N/B) = O(1), load factor is 1, and we get good performance. If B is smaller than N (fewer table entries 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(i.e doubling the table size doesn't really affect performance in separate chaining).
3. **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] == null  /* 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*/
    report ``not found'' /* entire table is full */
```

4. H(key) = 0: this is a hash function that creates linear search!
5. Problem with linear probing: Clustering in regions of the table where the keys collide. This effect can lead to hashing becoming sequential search.
6. Deletions in open addressing are a problem. An open spot breaks the continuity of a search for a key. Lazy deletion is used instead, which means we need another bit to mark each cell as alive or dead.
7. **Quadratic Probing** spreads out search. Probe(i)= $Hash(Key) + i^2$ where i is probe number. Sequence of probes is 1, 4, 9, 16,..... However, you 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 table is at least half empty (see Weiss p. 181). So we choose a table size that is prime and greater than twice the number of elements.

8. **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:

$$H_1(Key) \;=\; Key \,\%\, B$$

where B is the number of entries in the table, we can also define a second hash function $H_2(Key)$ that defines a probe increment:

$$H_2(Key) \;=\; Probe\,Increment$$

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* - they share no common divisors other than 1, then you can prove that the probe sequence will in fact cover the entire table. 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.

9. **Guaranteeing Table will be Covered with a Probe Increment**

   (a) If we use a probing strategy with a non prime table size, we may not be able to guarantee that the probes will always find an open slot.

   (b) Example: Choose a table size of 6, and a second hash function that maps into the range increment 1-5. Do the probe sequences for each increment, assuming we have an initial hash to index 0:

   Probe incr and probe sequence: Initial hash = 0, Tablesize=6 (Non-Prime)

Probe Incr.					
1	1	2	3	4	5
2	2	4	0	2	4
3	3	0	3	0	3
4	4	2	0	4	2
5	5	4	3	2	1

   Note that some of the probe sequences only cover a small part of the table before they return to the original hash location and start repeating.

   (c) However, we can guarantee that if the table size and probe increment are *relatively prime* - they share no common divisors other than 1, then the probe sequence will cover the table without repeating any locations. Hence, we need to choose a second hash function that will be in the range $Max(1, TableSize - 1)$. One way to achieve this is to use a prime table size B, and a second hash function as:

$$Probe\,Increment \;==\; H_2(Key) \;=\; Max(1,\,(Key\%B))$$

   This guarantess that the probe increment will be relatively prime with B.

   (d) Example: Choose a table size of 7 (a prime number). Now choose a probe increment in the range 1-6. Any of these increments will be relatively prime with a table size of 7. Now do the probe sequences for each increment, assuming we have an initial hash to index 0:

   Probe incr. and probe sequence: initial hash =0, Tablesize=7 (Prime)

Probe Incr.						
1	1	2	3	4	5	6
2	2	4	6	1	3	5
3	3	6	2	5	1	4
4	4	1	5	2	6	3
5	5	3	1	6	4	2
6	6	5	4	3	2	1

10. **Rehashing:** No matter how good our hash functions are, we still run into problems if the table size B is too small. If table gets near full, we increase search time looking for an empty space. A simple solution is to **Rehash** - create a new table twice as big. To rehash, we simply recalculate the table entries with new value of B, typically doubling the size of the table. The cost of rehashing is O(N). The rehashing algorithm is simple: just take each entry in the old table, and compute its hash function using the new tablesize. The cost is $O(N)$, since we have to rehash each entry, but we do it only once, as the original hash table gets full. Another nice aspect to rehashing is you can easily make it an automatic procedure. Once you start seeing the load factor of your table increase to say 75%, you can automatically invoke a rehash. Or you may invoke if you find an insertion into the table requires more than a specified number of probes, signifying clustering.

11. 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.

12. 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.

4 Summary of Collison Resolution Policies

1. If we are using Separate Chaining, then collisions are simply added on to the linked list for the bucket that the key hashes to.

2. With open addressing, we need a probe strategy if the initial hash location is being used.

   (a) Linear Probing: Probe(i) = $Hash_1(Key) + i$.

   (b) Quadratic Probing: Probe(i)= $Hash_1(Key) + i^2$.

   (c) Double Hashing: Probe(i) = $Hash_1(Key) + i * Hash_2(Key)$

5 Computing Prime Numbers Using Bit Operations in Java

An application program that computes the prime numbers from 2 to 1 million using bit sets and the Sieve of Eratosthenes algorithm. Initially, all numbers are marked as prime using a large bit vector. We start at number 2 and mark all multiples of 2 up to the number SIZE as not prime. It then finds the next number after 2 that is marked as prime and crosses out multiples of that number as not prime. The algorithm stops when we try to test multiples of a prime number that is greater than Sqrt(SIZE) (why...????). The bit vector is then scanned and any unmarked location is a prime number. Prime numbers are output to the console up to the value $printTo$.

```
import java.util.*;
public class Primes {
    private int SIZE;
    private final static int printTo = 500;
    private  BitSet b ;
    Primes(int size) {
        int i;
        SIZE=size;
        b= new BitSet (SIZE);  //bit vector with SIZE entries
//Mark all the bits from 2 to SIZE as prime (set bit=1);
        for (i = 2; i < SIZE; i++)
            b.set (i);
    }
    public void computePrimes () {
        int i;
        int count=0;

//Mark all  multiples of prime numbers as not prime (set bit =0);
        i = 2;        // start with smallest prime number
        while (i * i < SIZE) {
            if (b.get(i)) {
                count++; // counts number of primes found
                int k = 2 * i;
                while (k <= SIZE) { //mark all multiples of this prime
                    b.clear (k);
                    k += i;
                }
            }
            i++;
        }
           // display results
        System.out.println ("\nNumber of primes between 2 and " + SIZE +
                            " = " + count);
         }
        for (i = 2, count = 0; i < SIZE; i++)
            if (b.get (i)) {
                if (i <= printTo)
                    System.out.print (i + "  ");
                count++;
                }
    public static void main (String[] args) {
        Primes app = new Primes(500);
        app.computePrimes();
    }
}
---------------------------------------------------------
Number of primes between 2 and 500 = 95
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67
71 73  79  83  89  97  101  103  107  109  113  127  131  137  139
149  151  157  163  167  173  179  181  191  193  197  199  211  223
227  229  233  239  241  251  257  263  269  271  277  281  283  293<
307  311  313  317  331  337  347  349  353  359  367  373  379  383
389  397  401  409  419  421  431  433  439  443  449  457  461  463
467  479  487  491  499
```

6 Hashing, Collisions and Probability

How *random* are collisions? We can use an analogy with birthdays to find out. The birthday problem asks whether any of the people in a given group has a birthday matching any of the others. Presuming all birthdays are equally probable, the probability of a given birthday for a person chosen from the entire population at random is 1/365. If two people out of a group have the same birthday we can think of this as a collision.

How do we calculate the probability that someone has the same birthday? We use the fact that:

Prob(at least 2 people with same bday) = 1 - P(none same bday).

So we compute the complement probability, that no people have the same birthday, and then subtract from 1 to find the probability that at least 2 people have the same birthday.

Do the math for two people: P(my birthday is not same as the other person's)= $\frac{364}{365}$

Then, P(my birthday is the SAME as the other person's) = $1 - \frac{364}{365}$

We can easily extend this to three or more people: For three people to not share a birthday, the first person can choose any of 365 out of 365 days and not have a collision, then the second person can choose any of 364 out of 365 days, and the third person can choose any of 363 out of 365 days, and so on. Each event is independent, so we can multiply the probabilities of each person not having same birthday to get the overall probability:

P(Given 3 people, they DON'T have same birthday) = $\frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365}$

P(Given 3 people at least 2 have same birthday) = 1- $\frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365}$

Generalize for N people:

P(Given N people, they DON'T have same birthday) = $\frac{365}{365} \cdot \frac{364}{365} \cdot \frac{363}{365} \cdot \frac{362}{365} \ldots \frac{(365-N+1)}{365}$

We can turn this into a more general forumla as:

P(Given N people, they DON'T have the same birthday) = $\frac{365!}{(365^N)(365-N)!}$

The surprising result is that you only need 23 people in a room to have better than .5 probability of a same birthday, and with 50 people the probability is .97

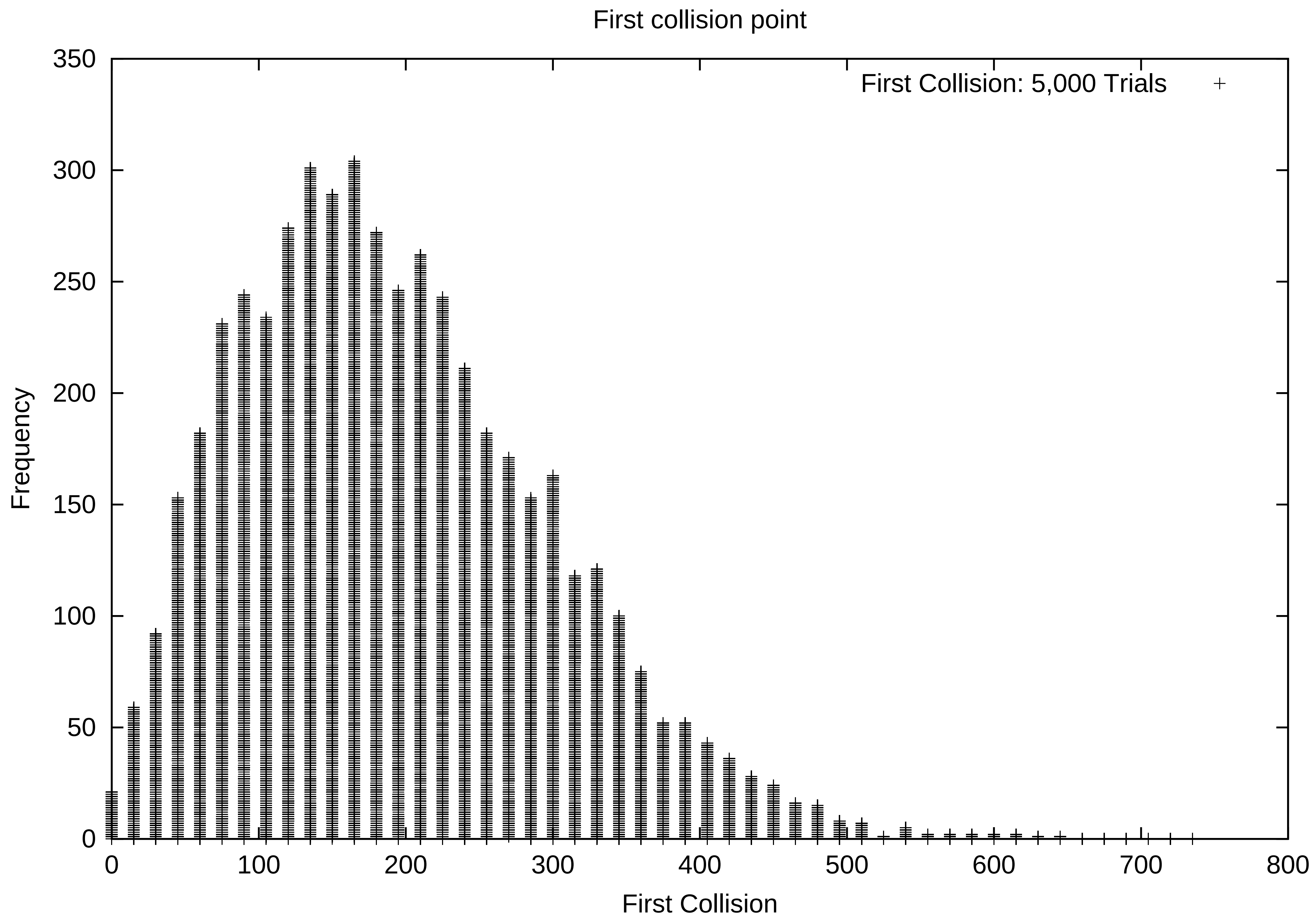
We can use the same analysis on hashing. Assuming our keys are random and equally probable, can we calculate the numberof expected collisions? Using the unix dictionary example, let us generate a random number between 0 and 25142 (there are 25143 words in /usr/dict/words), and see if how they collide. We can think of this as generating random polynomial hashcodes that map to locations in a 25143 element table.

The result is that about 9,000 collisions occur with totally random data. That isn't much better than our polynomial hash code, which has about 9,237 collisions on the unix words. This shows that the polynomial hashcode method is pretty good at scattering the data at random.

Further, we can use the birthday proability analysis to find out the number of samples we need to generate a first collision out of 25,143 random numbers.

Prob(no collisions in 25,143 entry hash table in N samples) =
$\frac{25143}{25143} \cdot \frac{25142}{25143} \cdot \frac{25141}{25143} \cdots \frac{(25143-N+1)}{25143}$

It turns out that the probability of a collision is greater than .5 after just 188 samples. The graph on the next page shows the number of samples needed before the first collision run over 5000 experiments. You can see the first collision peaks before 200 samples, and has a very short tail after that.


First collision point
First Collision: 5,000 Trials
Frequency
First Collision
0
50
100
150
200
250
300
350
0
100
200
300
400
500
600
700
800