11/09/15 Project 2

Fall | 15

Algorithms in Bioinformatics

Nabil Azamy

An Analysis of alternative hashing methods using differing division/bucket standards and collision handling.

Hash Tables

At its simplest a hash table is simply an array containing a number of keys, which may be used to hold information. These keys typically pair to some other object and represent one half of a key pair value. Assigned keys can pair to primary source objects, the key serving as pointer, or even other data structures. In this way most hash tables function as address books for access and since their structure is relatively simple and exists simply as a series of indexes contained in an array hash tables can be extremely flexible in their implementations.

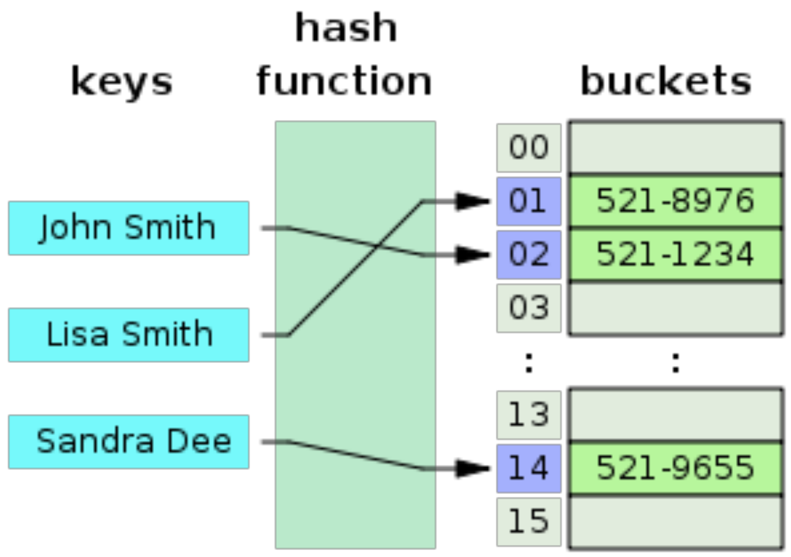


Figure 1: The canonical hash table

Typical implementations of hash tables require two functions to produce the data structure; hash functionality and collision handling. Hashing or has functionality is simply a preprocessing event whereby a key is converted to a numerical representation of the original key. This value is then further manipulated via one of several methods into an array specific index for which to be stored in. However, designing a perfect hashing function is extremely difficult and often requires previous or extensive knowledge of potential key values, which will be hashed for the tables. In instances of non-perfect hashing collisions may occur. A collision is defined as attempting to index a hashed key value somewhere within the hash table, which has already been occupied, by another key value. Strictly speaking avoiding a proper collision-handling algorithm can result in data loss both in terms of previous indexes being overridden or new ones being unable to be properly incorporated into the hash table. It is with these two features of hash tables in mind this project was assigned to test.

Nabil Azamy   Algorithms in Bioinformatics
11/09/15   Project 2

Implementation in Java

Despite a host of available hashing libraries here the goal was to design a hashing structure from scratch and capable of the following tasks:

- Hashing division modulo of 120 within equal table size with linear, quadratic, and chain collision handling
- Hashing division modulo of 113 within a table of size 120 with linear, quadratic, and chain collision handling
- Hashing division modulo of 41 within a table of size 40, a bucket size of 3 with linear and quadratic collision handling
- Hashing using a custom method with a table of size 120 with linear, quadratic, and chain collision handling

Hashing

The hashing functions used within this implementation primarily use of a modulo function for assignment of the position. The following is the example of the hashing function used by example 1:

```
public static HashingTypes div_mod_120 = new HashingTypes() {
    public int hash(int data) {
      return data % 120;
    }
```

In the piece of code above a hashing method is performed using a simple modulo function. The int value the key is pre processed into an index number to be inserted into the hash table. Modulo functions are perfect for this use because they only return the remainder value of the number beind divided which allows for the hash table to artificially accept key values far larger than its mod value. Note, this value does not reflect the size of the hash table in question, in instances of open addressing and quadratic probing it may be in your best interest to have a modulo much smaller than your intended hash table size.

As per the assignment the modulo hashing functions within the implementation are as follows:

Example set A:

```
public static HashingTypes div_mod_120 = new HashingTypes() {
    public int hash(int data) {
      return data % 120;
    }
  };
```

Example set B:

```
public static HashingTypes div_mod_113 = new HashingTypes() {
   public int hash(int data) {
      return data % 113;
   }
};
```

Example set C:

```
public static HashingTypes div_mod_41 = new HashingTypes() {
   public int hash(int data) {
      return data % 41;
   }
};
```

Example set D:

```
public static HashingTypes custom = new HashingTypes() {
   public int hash(int data) {

      return foldShift(data) % fourKPlus(120);

   }
```

Each of the corresponding example sets pertain to the groupings in the assignment document, eg example set A applies for schemes 1-3. Examples sets A – C, each function through the same modulo based division method as previously described with a operational cost of O(1) for each hashing event required. However example set D, the custom set described follows an alternative hashing mechanism.

There are two methods, which both the key value and the table size must go through to arrive at the hash table position, that's foldShift and fourKPlus.

The foldShift preprocessing event is a method for introducing higher randomness into hashing functions via breaking up key values, converting them into integer values, and summing their component pieces to use as a key. This increase in randomness would function to increase the ability of the hash table to be more spread out which would function to decrease collision events.

```
public static int foldShift(int data) {
      int pseudoKey = 0;
      int n = 1;
      int cn = 0;
      int grouping = 0;
      String targetKey;

      targetKey = Integer.toString(data);
```

```
    char c[] = targetKey.toCharArray();

    while (cn < targetKey.length()) {
      grouping = grouping << 8;
      grouping += c[cn];
      cn++;

      if (n == 4 || cn == targetKey.length()) {
        pseudoKey += grouping;
        n = 0;
        grouping = 0;
      }
      n++;
    }
    return Math.abs(pseudoKey);
  }
```

This increased degree of randomness would also pair nicely with collision functions that increase that as well. In particular the quadratic probing method should produce the best results while linear probing combined with keys of small ranges would perform the worst.

In terms of operational time the foldShift method runs in an operational time of O(n) because it is dependent on the length of the key in question.

Collisions with Probing

Collisions are a regular part of hashing functions and require much attention to resolve their issues quickly and efficiently. However, given a hash table of any size when its contents reach a certain size threshold their operations will result in more and more frequent collisions searching for an open slot. One would expect the number of collisions to expect within a given hash table to grow logarithmically to the size of the table over time and for the most part that is the case. This state of hash table growth is often expressed as a tables load factor. It is defined as follows:

$$\alpha = \frac{n}{m}$$

Where m is the size of the hash table in question and n is the number of indexes occupied. A table, which is of size 100 and contains 70 keys is said to have a load factor of 0.7. This value is frequently measured and often after passing some threshold will result in a systemic expansion of hash table size in the interests of lowering the frequency of collisions.

Still, since hash tables cannot be infinitely large in size there are a series of probing algorithms designed to minimize the impact collisions may have in hash table insertion events.

Linear probing as it is implemented in the assignment is done as follows:

```
public int findFreeSlot(int startSlot)
  {
    int slot = startSlot;
    while (table.slotFull(slot))
    {
      numCollisions++;
      slot = (slot + 1) % numSlots;
      if (slot == startSlot)
      {
        return -1;  // table is full.
      }
    }
    return slot;
  }
```

This technique, in the event of a collision, slowly increments up in the hash table until it finds an open position and returns that value. One of the primary advantages of this probing method is that, assuming a table of large enough size and a hash function of sufficient randomness, this function can operate in O(1) time for insertion and fetch operations. However, linear probing can begin to slow increasingly as load factors increase and often don't scale well with hash table expansions.

An alternative take on linear probing is quadratic probing. Quadratic probing is an expansions of the ideas present in linear probing which instead of incrementally increasing until an open slot is acquired approaches the next slot attempt via a quadratic expansion. As implemented within the assignment it looks as follows:

```
public int findFreeSlot(int startSlot)
  {
    int slot = startSlot;
    int i = 0;
    while (table.slotFull(slot))
    {
      numCollisions++;
      i++;
      slot = (slot + (1/2 * i + 1/2 * i ^ 2)) % numSlots;
```

```
            }
            return slot;

        }
```

The primary position of interest is the slot assignment within the while loop using the following formula:

$$slot = (0.5i) + (0.5i^2)\ \%\ tableSize$$

There are a number of possible quadratic formulations possible, however, an ideal solution would be one which sufficiently has the ability to traverse the hash table visiting all positions with the least number of repeats but the effectiveness for quadratic probing is heavily linked to the size of the table in question. And often decreases in efficiency more quickly than linear probing as load factors increase.

Chaining on the other hand is an alternative to the linear/quadratic probing method, which tends to perform best under conditions of high load factor. Due to the presence of linked lists pointing from the hash table, chained probing is a favorite of smaller hash tables and through the implementation of intelligent growth functions they can be extremely fast.

Analysis

The output of each implementation of the hash techniques described above, sans hash table output, is as follows:

```
Number of items to be hashed:  60
****  Hashing Scheme: Division modulo 120 on a table size of 120 slots (bucket size
= 1), linear probing ****
Collisions:  30

****  Hashing Scheme: Division modulo 120 on a table size of 120 slots (bucket size
= 1), quadratic probing ****
Collisions:  67

****  Hashing Scheme: Division modulo 120 on a table size of 120 slots (bucket size
= 1), chaining ****
Collisions:  71
```

**** Hashing Scheme: Division modulo 113 on a table size of 120 slots (bucket size = 1), linear probing ****
Collisions: 30

**** Hashing Scheme: Division modulo 113 on a table size of 120 slots (bucket size = 1), quadratic probing ****
Collisions: 34

**** Hashing Scheme: Division modulo 113 on a table size of 120 slots (bucket size = 1), chaining ****
Collisions: 34

**** Hashing Scheme: Division modulo 41 on a table size of 40 slots (bucket size = 3), (120 total slots) linear probing ****
Collisions: 43

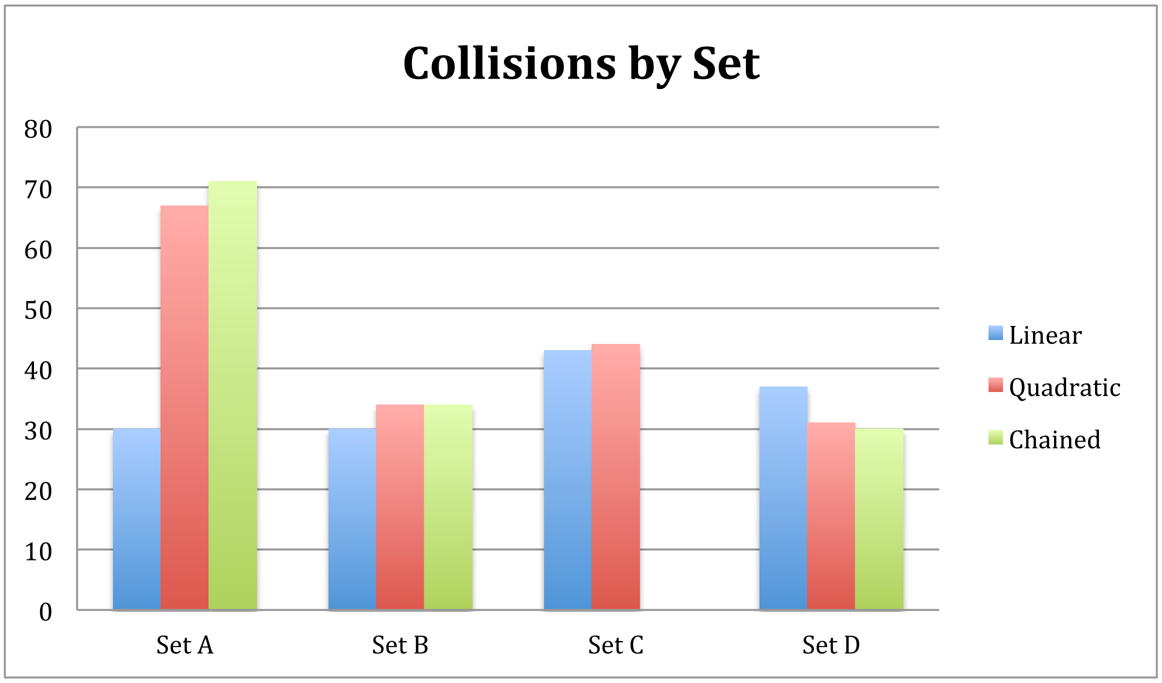
**** Hashing Scheme: Division modulo 41 on a table size of 40 slots (bucket size = 3), (120 total slots) quadratic probing ****
Collisions: 44

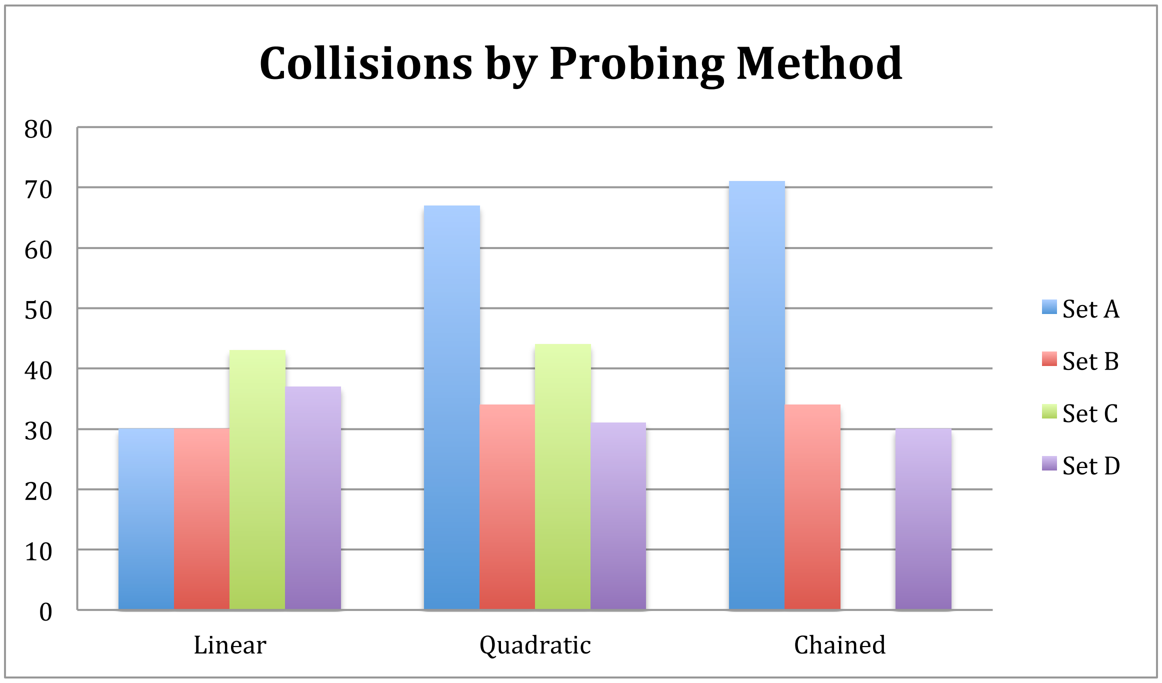
**** Hashing Scheme: Custom hash function on a table size of 120 slots (bucket size = 1), linear probing ****
Collisions: 37

**** Hashing Scheme: Custom hash function on a table size of 120 slots (bucket size = 1), quadratic probing ****
Collisions: 31

**** Hashing Scheme: Custom hash function on a table size of 120 slots (bucket size = 1), chaining ****
Collisions: 30

And are graphically represented below:





Nabil Azamy
11/09/15

Algorithms in Bioinformatics
Project 2

	Linear	Quadratic	Chained
Set A	30	67	71
Set B	30	34	34
Set C	43	44	N/A
Set D	37	31	30

From the results above it would seem the two overall best performing combinations were the division modulo paired with linear probing and the custom hashing scheme and chaining with 30 collisions. In fact the division modulo had 2 implementations which both scored 30 given the provided dataset.

One of the primary advantages of using a linear probing scheme paired with a load factor of ~0.5 is that its function has yet to reach the performance drop off which comes after a load factor of 0.75 has been reach. Running the two division linear probing pairs again with table sizes of 75 (maintaining a 0.8 load factor) their collisions jump up to 105 and 169 respectively while the chaining custom combination came out on top with 89 collisions. That is one of the primary advantages of the chain implementation; they are far more capable of working with larger load factors relative to other, open addressing methods.