**Some hashing methods used:**

**Bit shifting:** We perform bitwise shifts of a single byte since the lowest level we can go to in C is the byte level – we can never move/modify single bits, only whole bytes at a time. But we can shift a number of bits. So,

The number 8 in binary is represented by 1000. If we wanted to shift 8 three bits to the right,

8 << 3 or 1000 << 3 == 0001 or 1.

This technique aids us in creating seemingly random hash values.

**XOR operator (^):** The XOR operator provides an even truth table which gives us somewhat uniform results.

**p|q 🡪 r**

0|0 🡪 0

0|1 🡪 1

1|0 🡪 1

1|1 🡪 0

As you can see, the truth table produces even results. Combining the XOR operator and bitwise manipulation, you can create some very interesting hashing algorithms that are actually quite simple to code.

I chose to create roughly six Hash functions and analyze the results. Some produce unexpected results that I do not completely understand. Some perform better at low occupancy, and some perform better at high occupancy. Also, it seems that some numbers work better for bit shifting, but I cannot figure out a specific methodology to these numbers. (maybe odd??)

**Initial Test – Reaching 80% occupancy**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size  MAX\_ELE\*BUCKET | Percentage of Occupancy | Initial Collisions | Subsequent Collisions | Total Collisions |
| 500\*6 = 3000 | 10% | 117 | 104 | 221 |
| 400\*6 = 2400 | 13% | 148 | 207 | 335 |
| 300\*6 = 1800 | 18% | 152 | 159 | 311 |
| 200\*6 = 1200 | 27% | 195 | 476 | 674 |
| 100\*6 = 600 | 54% | 243 | 1004 | 1247 |
| 75\*6 = 450 | 72% | 253 | 867 | 1120 |
| 68\*6 = 408 | 80% | 265 | 2342 | 2607 |

**Hashing algorithm used:**

int HashTable::Hash\_1(DataType data)

{

int wordLen = data.length();

int keyVal = 0;

for (int i = 0; i < wordLen; i++)

{

if (keyVal % 2 == 0)

keyVal += data[i];

else

{

keyVal -= data[i];

keyVal = keyVal << 3;

}

}

if (keyVal < 0)

keyVal = keyVal\*-1;

return keyVal % MAX\_ELEMENTS \* BUCKET\_SIZE;

}

**Summary of algorithm:**

Very simple initial algorithm. Data is a string that is passed in when the client calls Insert() and passes a new item. The string is taken and processes byte by byte (char by char). If the value current value of the HK is even, the current byte’s value is added to the HK. If the current HK is odd, the current byte’s value is subtracted from HK. Then, I shift three bits to the right.

**Second Test – Reaching 80% occupancy**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size  MAX\_ELE\*BUCKET | Percentage of Occupancy | Initial Collisions | Subsequent Collisions | Total Collisions |
| 500\*6 = 3000 | 10% | 263 | 1468 | 1731 |
| 400\*6 = 2400 | 13% | 263 | 1468 | 1731 |
| 300\*6 = 1800 | 18% | 263 | 1468 | 1731 |
| 200\*6 = 1200 | 27% | 263 | 1468 | 1731 |
| 100\*6 = 600 | 54% | 263 | 1468 | 1731 |
| 75\*6 = 450 | 72% | 285 | 26544 | 26289 |
| 68\*6 = 408 | 80% | 291 | 29998 | 30289 |

**Hashing algorithm used:**

int HashTable::Hash\_2(DataType data)

{

int wordLen = data.length();

int keyVal = 0;

for (int i = 0; i < wordLen; i++)

{

keyVal = (keyVal ^ data[i]);

}

return keyVal % MAX\_ELEMENTS \* BUCKET\_SIZE;

}

**Summary of algorithm:**

This was another simple algorithm that I just wanted to test. It takes the current keyValue and XORs it with the current byte of the string. The results that it produces are quite terrible, and odd.

**Third Test – Reaching 80% occupancy**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size  MAX\_ELE\*BUCKET | Percentage of Occupancy | Initial Collisions | Subsequent Collisions | Total Collisions |
| 500\*6 = 3000 | 10% | 288 | 26813 | 27101 |
| 400\*6 = 2400 | 13% | 288 | 26813 | 27101 |
| 300\*6 = 1800 | 18% | 288 | 26813 | 27101 |
| 200\*6 = 1200 | 27% | 288 | 26813 | 27101 |
| 100\*6 = 600 | 54% | 288 | 27147 | 27435 |
| 75\*6 = 450 | 72% | 288 | 27028 | 27316 |
| 68\*6 = 408 | 80% | 291 | 27322 | 27613 |

**Hashing algorithm used:**

int HashTable::Hash\_3(DataType data)

{

int wordLen = data.length();

int keyVal = 0;

for (int i = 0; i < wordLen; i++)

{

keyVal = (keyVal >> 33) ^ data[i];

}

if (keyVal < 0)

keyVal = keyVal\*-1;

return keyVal = keyVal % MAX\_ELEMENTS \* BUCKET\_SIZE;

}

**Summary of algorithm:**

This algorithm is another experimental algorithm. It takes the current keyValue and shifts the bits 33 spaces to the left. It produces terrible and odd results similar to test two. I believe that the extremely high number of subsequent collisions (for test two and three) is because the values are hashing to the first few possible places in the list. This is a good illustration of the problem with bucketing – it turns into linear probing very quickly because of clustering if the hash function is not effective. So, every time an element needs to be inserted at the top of the list, it has to probe all the way down towards the bottom of the list, and each time it moves down a spot the subsequent collision counter is incremented.

**Fourth Test – Reaching 80% occupancy**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size  MAX\_ELE\*BUCKET | Percentage of Occupancy | Initial Collisions | Subsequent Collisions | Total Collisions |
| 500\*6 = 3000 | 10% | 94 | 34 | 128 |
| 400\*6 = 2400 | 13% | 110 | 53 | 163 |
| 300\*6 = 1800 | 18% | 134 | 80 | 214 |
| 200\*6 = 1200 | 27% | 170 | 197 | 367 |
| 100\*6 = 600 | 54% | 232 | 668 | 900 |
| 75\*6 = 450 | 72% | 254 | 846 | 1100 |
| 68\*6 = 408 | 80% | 262 | 1645 | 1907 |

**Hashing algorithm used:**

int HashTable::Hash\_4(DataType data)

{

int wordLen = data.length();

int keyVal = 0;

for (int i = 0; i < wordLen; i++)

{

keyVal = keyVal ^ ((keyVal >> 3) \* (keyVal << 5) ^ data[i]);

}

if (keyVal < 0)

keyVal = keyVal\*-1;

return keyVal % MAX\_ELEMENTS \* BUCKET\_SIZE;

}

**Summary of algorithm:**

This algorithm first takes the current keyValue and does a bitwise shift to the left by 3, and a bitwise shift to the right by 5. Then this result is multiplied and XORed with the current byte (current character) and then that result is XORed with the current keyValue. This algorithm was particularly interesting because changing the number that we shift by changes the result significantly. For example, I changed the number 3 to 15 and the results were terrible. Changing the number to 3, as you can see, the results are better than the first iteration of my hashing algorithm. This is the best results I have had yet. It is interesting how the subsequent collisions are much lower than the first, but the initial collisions are not that far off.

**Fifth Test – Reaching 80% occupancy**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size  MAX\_ELE\*BUCKET | Percentage of Occupancy | Initial Collisions | Subsequent Collisions | Total Collisions |
| 500\*6 = 3000 | 10% | 87 | 25 | 112 |
| 400\*6 = 2400 | 13% | 107 | 49 | 156 |
| 300\*6 = 1800 | 18% | 126 | 67 | 193 |
| 200\*6 = 1200 | 27% | 162 | 118 | 280 |
| 100\*6 = 600 | 54% | 232 | 463 | 695 |
| 75\*6 = 450 | 72% | 254 | 939 | 1193 |
| 68\*6 = 408 | 80% | 261 | 1109 | 1370 |

**Hashing algorithm used:**

int HashTable::Hash\_5(DataType data)

{

int wordLen = data.length();

int keyVal = 0;

for (int i = 0; i < wordLen; i++)

{

keyVal = keyVal ^ ((keyVal >> 5) + (keyVal << 3) ^ data[i]);

}

if (keyVal < 0)

keyVal = keyVal\*-1;

return keyVal % MAX\_ELEMENTS \* BUCKET\_SIZE;

}

**Summary of algorithm:**

This algorithm I found particularly interesting. In comparison to Hash\_4, I thought that I may be losing some digits by multiplying and getting a resultant that is too large (out of range of an int) so I changed the multiplication to addition. The initial collision rate is very similar (in fact, the same in some cases), but what I found very interesting was the number of subsequent collisions dropped significantly, especially when nearing high occupancy (except at 72% I’m not very sure what that is about). I left rest of the algorithm the way it was, since I was getting good results with odd numbered bit shifts that were lower numbers. In conclusion, this is the best result I have received so far, with roughly 1400 collisions.

**Sixth Test – Reaching 80% occupancy**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Array Size  MAX\_ELE\*BUCKET | Percentage of Occupancy | Initial Collisions | Subsequent Collisions | Total Collisions |
| 500\*6 = 3000 | 10% | 81 | 20 | 101 |
| 400\*6 = 2400 | 13% | 96 | 35 | 131 |
| 300\*6 = 1800 | 18% | 119 | 62 | 181 |
| 200\*6 = 1200 | 27% | 163 | 176 | 339 |
| 100\*6 = 600 | 54% | 228 | 498 | 726 |
| 75\*6 = 450 | 72% | 253 | 877 | 1130 |
| 68\*6 = 408 | 80% | 259 | 2186 | 2445 |

**Hashing algorithm used:**

int HashTable::Hash\_6(DataType data)

{

int wordLen = data.length();

int keyVal = 0;

for (int i = 0; i < wordLen; i++)

{

if (keyVal % 2 == 0)

keyVal = keyVal ^ ((keyVal >> 5) + (keyVal << 3) ^ data[i]);

else

keyVal = keyVal ^ ((keyVal >> 3) - (keyVal << 5) ^ data[i]);

}

if (keyVal < 0)

keyVal = keyVal\*-1;

return keyVal % MAX\_ELEMENTS \* BUCKET\_SIZE;

}

**Summary of algorithm:**

With this algorithm I attempted to merge all the methods I used in the previous 5 tests, and the results were surprising. I expected this function to out-perform the rest, however once it passes 20% occupancy the performance drops. The number of subsequent collisions is very high, so the hash function is not distributing evenly. This particular algorithm performed very well at low occupancy, but somewhat bad when approaching full occupancy.

**In conclusion:**

This project was awesome. It is very interesting to see how small changes dramatically affect the distribution of hash keys. I am going to mess around more with Hash\_4, Hash\_5, and Hash\_6. I think I found effective bit shift numbers with 3 and 5, but I am sure there is a more efficient combination of bit shifts and XOR operators. I was also surprised on how effective my initial hash function was. I was not expecting Hash\_2 and Hash\_3 to be so terrible.

P.S. Thanks for an awesome semester, never have learned so much from a single class. Merry Christmas!