Nicholas Jones

COSC 3319 01

04/11/2019

Lab 04: Hashing & Collision Handling

A good hash algorithm is fast, randomizes the calculated addresses and uniformly distributes entries. However, collisions are common because of how difficult it is to create an ideal hash algorithm. This implies that designing an efficient and effective method for handling hash collisions is very important. There are many ways to handle collisions such as linear collision handling, random collision handling, and collision chaining.

Linear collision handling an easy way to handle collisions. However, the downside is that linear will start to develop primary clustering quickly as the table begins to fill. This was extremely apparent while implementing the “Burris” Hash algorithm. At a load factor of 50% the theoretical number of probes was estimated at E = (1 – a / 2) / (1 - a) where a = (the number of keys in the table) / (table size). With a table size of 2^7 =128 and the number of keys in the table = (128 \* .50) = 64 this Implies a = 64 / 128 = 0.5. Then by plugging this value into the theoretical equation = (1 – 0.5 / 2) / ( 1 - 0.5) = (1 – 0.25) / 0.5 = 0.75 / 0.5 = 1.5 expected probes to locate a given key. However, at 50% full the “Burris” hash received the follow empirical results. The for the first 30 values in the table the minimum number probes was 1 , maximum number of probes was a staggering 18 probes, and the average was 4.80 which is 320% more collisions then estimated by the theoretical formula. The last 30 was even worse with the minimum at 16, maximum at 59, and average at 36.367 which is is over 2000% more collisions then theoretically estimated. These extremely high empirical results are caused by primary clustering from continuous collisions. The values towards the end of the table are substantially higher because with linear collision handling doesn’t take steps to prevent primary clustering and the result is that with each new collision the number of probes for all hashing in that sector starts to grow exponentially. This is true even with an appropriate hashing algorithm but, collisions normally don’t hit an exponential growth rate until the table is about 75% full. The effect of the clustering is even more apparent once the table is at 90% full. The Theoretical results for a 90% table should be E = (1 – (128 \* .90) / 2) / (1 – (128 \* .90) = 5.50 probes. However, the empirical results from the “Burris” hashing algorithm resulted in, for the first 30 entries, a minimum of 1 and a maximum of 18 which averages to 4.80 probes. That value seems fine, However the last 30 keys in the table took a minimum of 66 probes and maximum of 113 probes which averages at about 89 probes. The implies that as the table begins to fill that the primary clustering becomes worse and an average of 89 probes is not good if the purpose of using hashing is to be fast.

Another way of handling collision is to use the random probe technique. This involves generating a set of pseudo random numbers from 1 through the table size – 1. Then using these random offsets, you place collisions in those offset positions instead of linearly placing collisions in the next sequentially available location. This helps to avoid the effect of primary clustering as described with linear hashing. The theoretical results of random probe collision handling follow the formula E = (1 / a) ln (1 – a) where a is (the number of keys in the table) / (Table size). For a table size of 128 and which is 50% full this equates to E = (1 / (128\*0.50 / 128)) ln (1 – (128\*0.50 / 128) = 1.39 expected probes to locate a key. Using the “Burris” algorithm with random probing resulted in, for the first 30 keys in the table, a minimum of 1, maximum of 4 and average of 1.733 probes. For the last 30 keys in the table, a minimum of 1, maximum of 15, and average of 7.2. These results are substantially less than the linear technique but, still higher than the theoretical values. When the table is 90% full the theoretical should be E = (1 / (128\*0.90 / 128))ln(1 – (128\*0.90 / 128) = 2.56 expected probes to locate a key. With the “Burris” algorithm the empirical results were as follows: for the first 30 entries the minimum was 1, maximum was 37, and average was 4.367 probes. For the last 30 entries, the minimum was 1, maximum was 35, and average was 15.567. Since the Linear and Random probing techniques went well over the theoretical values it’s safe to assume that the “Burris” algorithm is not efficient. The random probe technique was however was able to minimize the primary clustering enough to keep the probe count below the linear probe results. However, with some simple modifications to the Burris algorithm. The usage of a linear and random probe should be more effective.

--Put Table Here

After scrutinizing the “Burris” hashing algorithm the first detail I realized is that it’s not portable between compilers. The compiler I use is the Ada-core Community Edition which has 32-bit Integer’s and 64-bit Long Integers. The compiler used when developing this hash used 16-Bit Integers and 32-Bit Long Integers. There is not a check to validate the type of integer being used for conversion which lead to skewed values until I adjusted the algorithm slightly to cancel out the 16 high order bits. Having extra high order bits caused multiple changes to occur with my compiler. When slicing a string for example at 3,4 when the string value is “Burris” and using ada unchecked conversion would return an integer value (Burr) where B would be the leftmost 8 bits converted to its ascii binary value, the mid-left 8 bits would correspond to u, and the rightmost 16 bits would both correspond to r ascii values. Why? This Is caused by how unchecked conversion works in ada at the lower level. Ada converts a string which is stored in binary from right to left to an integer that is stored from left to right. I’m not sure exactly why ada reads in the additional values, B and u, into the leftmost bits and not keep those values untouched. But, with some critical thinking, usage of my exposure to assembly language, I’ve developed a hypothesis that my version of ada is using a register that is larger than 16 bits and so the minimum number of bits it can return is 32 bit which implies that when slicing in ada it handles this by cancelling out the high order bits upon assignment of the slice. However, since I used unchecked conversion before this check ada returns the entire register as an integer value. Now, I didn’t need to research exactly how my compiler performed the task and that would defeat the purpose of using a higher-level language in the first place. Instead I just canceled out the 16 high order bits when slicing only 2-character values from a string. Next, there is an issue with slicing a single character from a string and converting it using ada unchecked conversion. It would store this character is the mid-right 8 bits of a 32-bit integer. This meant I needed zero out the 16 high order bits and 8 low order bits to hash to prevent values other than what was intended to be stored in the 32-bit integer. However, the algorithm does / 256 which shifts the character value into the right most position but, without canceling the mid-left 8 bits then you still get an incorrect value. Moving past the issue of unintended values being passed through the “Burris” algorithm I noticed a few other issues. The next portion I’ll discuss is the Key[4..5] – Key[1..2] / 256. Key[4..5] takes two 8 bit character values and stores them at the right most 16-bits in a 32-bit integer. It subtracts these values 16 bits from each other than divides that result by 256. 256 is 2^8 which shifts 8 bits to the right. This only leaves 8 bits of non-zero values in the rightmost 8 bits. To add to this dilemma is the fact that subtraction is performed prior to this action which has a high probability of zeroing out the mid right 8 most bits in the first place. This alone contributes a great deal to the excessive amount of collisions caused by this hashing algorithm and causing the empirical results to far exceed the theoretical results. Moving on, the next issue that with Key[1..2] were only really using Key[1] and this is added to the value later in the algorithm and essentially canceling itself out. The only important bits in this algorithm is the mid right 8 bits because the other bits are being cancelled out through division and modulo computations. Using as many bits as possible and manipulating those bits in a random sequence will produce better results even if at the end of the algorithm the value is folded to 7 bits in order to produce values only within the table size. After realizing the effect of the “Burris” algorithm I built my own hash algorithm. It takes the first 4 characters in the string positions 1..4 into an integer value. I take this integer value, let’s call it HA, and split it into 4 separate integer values stored in an array of 4 integers. The first integer value at position 1 in the array is shifted to the rightmost 8 bits and the left 24 most bits are set to 0. Using input of “Burris” as an example this would mean that B’s ascii Value would be stored in position 1. The same type of operation is performed to allow u into position 2, r into position 3, and r(#2) into position 4. After zeroing out HA, Then I perform a loop of 4 iterations and start with position 1 and add it to the HA and multiple by i, where I is the current iteration, and continue through each position and iteration. After this is performed I take modulo 128 of the HA and that’s the value used for insertion. This should perform much better because I’m taking each character value at the beginning of the string and adding then together but, not only adding them together but, also weighting them different by multiplying by their position. I’ve also implemented the ability to adjust this algorithm based on the type of integer being used by the compiler. This gives each input a very unique address which is apparent when comparing the results in the table as follows. This algorithm also takes into account compilers with different integer lengths allowing it a level of portability between compilers.

The “Burris” algorithm and my “Jones” algorithm both use division remainder to keep the hash values within the table size range. They both use folding to manipulate the string into a value that can be used by the compiler’s integer through unchecked conversion with ada. The use of the techniques along with share a common goal and making unique hash addresses to store data input. An ideal hash function randomizes each key to a unique address. However, an ideal hash function are generally hard to create so using collision handling techniques can help with keeping collisions to a reasonable value. Overall, my hash algorithm developed with linear and random probing technique efficient in addressing string Values.