Rutger Yager

Dr. Burris COSC 3319.01

Analysis of Hash functions

Lab 5, Grade Option A

**Bad Hash Function Analysis**

|  |  |  |  |
| --- | --- | --- | --- |
| **Bad Hash – Linear Probing at 78% full.** | **Bad Hash – Linear Probing at 95% full.** | **Bad Hash – Random Probing at 78% full.** | **Bad Hash – Random Probing at 95% full.** |
| Hash Table Contents:  # 1: [magnetic ]  # 2: [inlet ]  # 3: [constrain ]  # 4: [marsupial ]  # 5: [rationalize ]  # 6: [scat ]  # 7: [toluene ]  # 8: [wet ]  # 9: [sparse ]  # 10: [quandary ]  # 11: [dactyl ]  # 12: [nosegay ]  # 13: [option ]  # 14: [forgetful ]  # 15: [ratiocination ]  # 16: [privilege ]  # 17: [sponsor ]  # 18: [exhilarate ]  # 19: [guard ]  # 20: [noggin ]  # 21: [prologue ]  # 22: [seal ]  # 23: [seat ]  # 24: [tiller ]  # 25: [ichthyosaur ]  # 26: [lazy ]  # 27: [malfeasance ]  # 28: [compass ]  # 29: [diastase ]  # 30: [emperor ]  # 31: [history ]  # 32: [keep ]  # 33: [gangway ]  # 34: [labial ]  # 35: [sacrament ]  # 36: [taint ]  # 37: [ ]  # 38: [ ]  # 39: [ ]  # 40: [ ]  # 41: [ ]  # 42: [ ]  # 43: [ ]  # 44: [ ]  # 45: [ ]  # 46: [ ]  # 47: [ ]  # 48: [ ]  # 49: [ ]  # 50: [ ]  # 51: [ ]  # 52: [ ]  # 53: [ ]  # 54: [ ]  # 55: [ ]  # 56: [ ]  # 57: [ ]  # 58: [ ]  # 59: [ ]  # 60: [ ]  # 61: [ ]  # 62: [ ]  # 63: [ ]  # 64: [ ]  # 65: [Joseph ]  # 66: [Kelly ]  # 67: [Corey ]  # 68: [Adam ]  # 69: [Clayton ]  # 70: [Dustin ]  # 71: [Robert ]  # 72: [Kyle ]  # 73: [Scott ]  # 74: [Octavio ]  # 75: [Judy ]  # 76: [Derek ]  # 77: [Jeffrey ]  # 78: [Jordon ]  # 79: [Vinnela ]  # 80: [Lisa ]  # 81: [Todd ]  # 82: [Veronica ]  # 83: [Matthew ]  # 84: [Michael ]  # 85: [Akhila ]  # 86: [John ]  # 87: [Charles ]  # 88: [James ]  # 89: [Chris ]  # 90: [Wade ]  # 91: [Christopher ]  # 92: [Fernando ]  # 93: [Batbold ]  # 94: [Joel ]  # 95: [Fabulous ]  # 96: [Misogamist ]  # 97: [Maiden ]  # 98: [Eye ]  # 99: [Constriction ]  #100: [Necromancer ]  #101: [Syncopate ]  #102: [Yolk ]  #103: [Afterwards ]  #104: [Person ]  #105: [1234567890123456]  #106: [Northwest ]  #107: [Irreversible ]  #108: [Fabricate ]  #109: [Honor ]  #110: [Staple ]  #111: [Under ]  #112: [Jutty ]  #113: [Finagle ]  #114: [Cook ]  #115: [Rush ]  #116: [Wine ]  #117: [Screen ]  #118: [Perfect ]  #119: [mole ]  #120: [parasympathetic ]  #121: [poison ]  #122: [brutalize ]  #123: [cap ]  #124: [cauldron ]  #125: [prepossess ]  #126: [wince ]  #127: [orthodontist ]  #128: [live ] | Hash Table Contents:  # 1: [magnetic ]  # 2: [inlet ]  # 3: [constrain ]  # 4: [marsupial ]  # 5: [rationalize ]  # 6: [scat ]  # 7: [toluene ]  # 8: [wet ]  # 9: [sparse ]  # 10: [quandary ]  # 11: [dactyl ]  # 12: [nosegay ]  # 13: [option ]  # 14: [forgetful ]  # 15: [ratiocination ]  # 16: [privilege ]  # 17: [sponsor ]  # 18: [exhilarate ]  # 19: [guard ]  # 20: [noggin ]  # 21: [prologue ]  # 22: [seal ]  # 23: [seat ]  # 24: [tiller ]  # 25: [ichthyosaur ]  # 26: [lazy ]  # 27: [malfeasance ]  # 28: [compass ]  # 29: [diastase ]  # 30: [emperor ]  # 31: [history ]  # 32: [keep ]  # 33: [gangway ]  # 34: [labial ]  # 35: [sacrament ]  # 36: [taint ]  # 37: [withal ]  # 38: [oak ]  # 39: [preordain ]  # 40: [haberdasher ]  # 41: [crimp ]  # 42: [luculent ]  # 43: [pennywort ]  # 44: [prevalent ]  # 45: [monolith ]  # 46: [suffrage ]  # 47: [wiper ]  # 48: [zygote ]  # 49: [academy ]  # 50: [amputate ]  # 51: [brash ]  # 52: [consummation ]  # 53: [epicure ]  # 54: [indulgent ]  # 55: [minute ]  # 56: [job ]  # 57: [hypodermic ]  # 58: [meridian ]  # 59: [ ]  # 60: [ ]  # 61: [ ]  # 62: [ ]  # 63: [ ]  # 64: [ ]  # 65: [Joseph ]  # 66: [Kelly ]  # 67: [Corey ]  # 68: [Adam ]  # 69: [Clayton ]  # 70: [Dustin ]  # 71: [Robert ]  # 72: [Kyle ]  # 73: [Scott ]  # 74: [Octavio ]  # 75: [Judy ]  # 76: [Derek ]  # 77: [Jeffrey ]  # 78: [Jordon ]  # 79: [Vinnela ]  # 80: [Lisa ]  # 81: [Todd ]  # 82: [Veronica ]  # 83: [Matthew ]  # 84: [Michael ]  # 85: [Akhila ]  # 86: [John ]  # 87: [Charles ]  # 88: [James ]  # 89: [Chris ]  # 90: [Wade ]  # 91: [Christopher ]  # 92: [Fernando ]  # 93: [Batbold ]  # 94: [Joel ]  # 95: [Fabulous ]  # 96: [Misogamist ]  # 97: [Maiden ]  # 98: [Eye ]  # 99: [Constriction ]  #100: [Necromancer ]  #101: [Syncopate ]  #102: [Yolk ]  #103: [Afterwards ]  #104: [Person ]  #105: [1234567890123456]  #106: [Northwest ]  #107: [Irreversible ]  #108: [Fabricate ]  #109: [Honor ]  #110: [Staple ]  #111: [Under ]  #112: [Jutty ]  #113: [Finagle ]  #114: [Cook ]  #115: [Rush ]  #116: [Wine ]  #117: [Screen ]  #118: [Perfect ]  #119: [mole ]  #120: [parasympathetic ]  #121: [poison ]  #122: [brutalize ]  #123: [cap ]  #124: [cauldron ]  #125: [prepossess ]  #126: [wince ]  #127: [orthodontist ]  #128: [live ] | Hash Table Contents:  # 1: [inlet ]  # 2: [constrain ]  # 3: [Robert ]  # 4: [Finagle ]  # 5: [emperor ]  # 6: [Under ]  # 7: [marsupial ]  # 8: [prepossess ]  # 9: [brutalize ]  # 10: [malfeasance ]  # 11: [live ]  # 12: [Kyle ]  # 13: [seat ]  # 14: [toluene ]  # 15: [ratiocination ]  # 16: [Michael ]  # 17: [Cook ]  # 18: [ ]  # 19: [ ]  # 20: [gangway ]  # 21: [forgetful ]  # 22: [history ]  # 23: [Wine ]  # 24: [ ]  # 25: [Northwest ]  # 26: [Octavio ]  # 27: [Jutty ]  # 28: [poison ]  # 29: [scat ]  # 30: [ ]  # 31: [ ]  # 32: [rationalize ]  # 33: [Maiden ]  # 34: [Eye ]  # 35: [ ]  # 36: [Lisa ]  # 37: [wince ]  # 38: [Jordon ]  # 39: [Constriction ]  # 40: [Fernando ]  # 41: [Chris ]  # 42: [cap ]  # 43: [Fabulous ]  # 44: [ ]  # 45: [Perfect ]  # 46: [Yolk ]  # 47: [compass ]  # 48: [ ]  # 49: [Todd ]  # 50: [noggin ]  # 51: [ichthyosaur ]  # 52: [magnetic ]  # 53: [Staple ]  # 54: [orthodontist ]  # 55: [Matthew ]  # 56: [option ]  # 57: [Scott ]  # 58: [ ]  # 59: [Vinnela ]  # 60: [James ]  # 61: [Syncopate ]  # 62: [tiller ]  # 63: [nosegay ]  # 64: [Necromancer ]  # 65: [Joseph ]  # 66: [Kelly ]  # 67: [wet ]  # 68: [ ]  # 69: [Batbold ]  # 70: [ ]  # 71: [Corey ]  # 72: [ ]  # 73: [ ]  # 74: [Wade ]  # 75: [ ]  # 76: [sparse ]  # 77: [Akhila ]  # 78: [Dustin ]  # 79: [cauldron ]  # 80: [seal ]  # 81: [ ]  # 82: [Rush ]  # 83: [parasympathetic ]  # 84: [Misogamist ]  # 85: [Jeffrey ]  # 86: [Joel ]  # 87: [ ]  # 88: [Honor ]  # 89: [ ]  # 90: [dactyl ]  # 91: [ ]  # 92: [ ]  # 93: [Clayton ]  # 94: [mole ]  # 95: [Fabricate ]  # 96: [Adam ]  # 97: [labial ]  # 98: [sacrament ]  # 99: [Afterwards ]  #100: [exhilarate ]  #101: [ ]  #102: [privilege ]  #103: [taint ]  #104: [diastase ]  #105: [1234567890123456]  #106: [ ]  #107: [keep ]  #108: [Person ]  #109: [ ]  #110: [ ]  #111: [Christopher ]  #112: [Screen ]  #113: [guard ]  #114: [Veronica ]  #115: [Charles ]  #116: [ ]  #117: [ ]  #118: [ ]  #119: [prologue ]  #120: [Derek ]  #121: [quandary ]  #122: [Irreversible ]  #123: [sponsor ]  #124: [lazy ]  #125: [ ]  #126: [John ]  #127: [Judy ]  #128: [ ] | Hash Table Contents:  # 1: [inlet ]  # 2: [constrain ]  # 3: [Robert ]  # 4: [Finagle ]  # 5: [emperor ]  # 6: [Under ]  # 7: [marsupial ]  # 8: [prepossess ]  # 9: [brutalize ]  # 10: [malfeasance ]  # 11: [live ]  # 12: [Kyle ]  # 13: [seat ]  # 14: [toluene ]  # 15: [ratiocination ]  # 16: [Michael ]  # 17: [Cook ]  # 18: [brash ]  # 19: [job ]  # 20: [gangway ]  # 21: [forgetful ]  # 22: [history ]  # 23: [Wine ]  # 24: [monolith ]  # 25: [Northwest ]  # 26: [Octavio ]  # 27: [Jutty ]  # 28: [poison ]  # 29: [scat ]  # 30: [minute ]  # 31: [prevalent ]  # 32: [rationalize ]  # 33: [Maiden ]  # 34: [Eye ]  # 35: [haberdasher ]  # 36: [Lisa ]  # 37: [wince ]  # 38: [Jordon ]  # 39: [Constriction ]  # 40: [Fernando ]  # 41: [Chris ]  # 42: [cap ]  # 43: [Fabulous ]  # 44: [crimp ]  # 45: [Perfect ]  # 46: [Yolk ]  # 47: [compass ]  # 48: [epicure ]  # 49: [Todd ]  # 50: [noggin ]  # 51: [ichthyosaur ]  # 52: [magnetic ]  # 53: [Staple ]  # 54: [orthodontist ]  # 55: [Matthew ]  # 56: [option ]  # 57: [Scott ]  # 58: [pennywort ]  # 59: [Vinnela ]  # 60: [James ]  # 61: [Syncopate ]  # 62: [tiller ]  # 63: [nosegay ]  # 64: [Necromancer ]  # 65: [Joseph ]  # 66: [Kelly ]  # 67: [wet ]  # 68: [academy ]  # 69: [Batbold ]  # 70: [wiper ]  # 71: [Corey ]  # 72: [ ]  # 73: [meridian ]  # 74: [Wade ]  # 75: [ ]  # 76: [sparse ]  # 77: [Akhila ]  # 78: [Dustin ]  # 79: [cauldron ]  # 80: [seal ]  # 81: [amputate ]  # 82: [Rush ]  # 83: [parasympathetic ]  # 84: [Misogamist ]  # 85: [Jeffrey ]  # 86: [Joel ]  # 87: [consummation ]  # 88: [Honor ]  # 89: [luculent ]  # 90: [dactyl ]  # 91: [zygote ]  # 92: [hypodermic ]  # 93: [Clayton ]  # 94: [mole ]  # 95: [Fabricate ]  # 96: [Adam ]  # 97: [labial ]  # 98: [sacrament ]  # 99: [Afterwards ]  #100: [exhilarate ]  #101: [ ]  #102: [privilege ]  #103: [taint ]  #104: [diastase ]  #105: [1234567890123456]  #106: [ ]  #107: [keep ]  #108: [Person ]  #109: [indulgent ]  #110: [preordain ]  #111: [Christopher ]  #112: [Screen ]  #113: [guard ]  #114: [Veronica ]  #115: [Charles ]  #116: [ ]  #117: [suffrage ]  #118: [ ]  #119: [prologue ]  #120: [Derek ]  #121: [quandary ]  #122: [Irreversible ]  #123: [sponsor ]  #124: [lazy ]  #125: [oak ]  #126: [John ]  #127: [Judy ]  #128: [withal ] |

The following data is an analysis of the performance of the following hash function utilizing both linear probing and random probing for collision handling:

function GetBadHashAddress (

Key : in String16)

return Long\_Integer is

type String2 is new String

(1 .. 2);

function StringToInt is

new Ada.Unchecked\_Conversion(String2, Long\_Integer);

Address : Long\_Integer;

begin

Address := (abs (StringToInt(String2(Key(13..14))))) + (abs (StringToInt(String2(Key(15..16)))));

return (Address mod 128) + 1;

end GetBadHashAddress;

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Minimum Probes** | **Maximum Probes** | **Average Probes** | **Expected Probes** |
| **Linear Probing at 78%** | 1 | 100 | 49.87 | 2.77 |
| **Linear Probing at 95%** | 1 | 122 | 61.02 | 10.5 |
| **Random Probing at 78%** | 1 | 99 | 49.18 | 1.94 |
| **Random Probing at 95%** | 1 | 121 | 60.52 | 3.15 |

**Probing Results for First and Last 30 Keys Inserted**

**Linear Probing – 78%**

The expected number of probes to obtain a key that has been stored using linear probing for collision handling changes based on how full (alpha) the table is, and can be approximated using the following formula: (1 – (alpha / 2)) / (1 - alpha). In this case, our alpha value is 78%, or 0.78. Using the formula we approximate: (1 – (0.78 / 2)) / (1 – 0.78) = 0.61 / 0.22 = ~2.77 probes to locate a key. What we find when looking at the data for our bad hash function in this case is that probes ranged anywhere from 1 to 100 probes and on average took 49.87 probes to locate a key within the table. The reason for the strikingly bad performance can be explained by the fact that linear probing works under the assumption that you use a good hash function, in particular one that produces unique addresses for almost every key and produces addresses based entirely on the key. It becomes a weak method for collision handling when the hash function tends to produce certain addresses more than others, as this results in the formation of primary clusters, groups of keys that will cause a collision for any address within their range. Linear probing begins to fail when large primary clusters form, so a good hash algorithm that will only produce several small primary clusters across the table is needed for linear probing to be an effective method. The hash function used in this example unfortunately does not meet these requirements; it does not produce unique addresses for every key, and even worse, it only produces addresses based on the last four characters of the key. What makes this particularly nasty is that a text key is not usually sixteen characters in length, and text is usually left-justified, so these four characters more often than not turn out to be four space characters. In terms of our hash result for this particular function, any key that has four spaces as the last four characters will always hash to the address of 65. Indeed, when looking at the table results for this case, we can see the formation of a large group of keys that starts at address 65, goes up to address 128, loops back down to address 1, and then extends up to address 36. Looking at the keys, there are only three that do not have four spaces for the last four characters, which means this large group is actually one single primary cluster for address 65. It becomes quite clear then why our average probe amount to find a key was 49.87; because most keys hashed to address 65, the search function almost always starts searching at address 65. Because address 65 is the head of a single large primary cluster, this essentially equates to a linear search, meaning we can expect an average probe count of (n + 1) / 2, where n in this case is our primary cluster size (100). (100 + 1) / 2 = 50.5, approximately the average probe count we observed.

**Linear Probing – 95%**

As stated previously, the formula to calculate the expected number of probes to obtain a key that has been stored using linear hashing is: (1 – (alpha / 2)) / (1 - alpha). In this case, our alpha value is 95%, or 0.95. Using the formula, we approximate: (1 – (0.95 / 2)) / (1 – 0.95) = 0.525 / 0.05 = ~10.5 probes to locate a key. While we expect the amount of probes to go up as the table becomes full, we only expect a jump from 2.77 probes to 10.5 probes. Actual results show a decidedly worse jump from 49.87 probes to a very high 61.02 probes. The explanation for this is again due to the fact that most keys used in this example hashed to address 65, this time resulting in an even larger primary cluster starting at address 65 and extending around to address 58. Because there are still only three keys that do not have four spaces at the end, searches for almost all keys in the table will start at address 65, meaning this again equates to a linear search, this time with a primary cluster size of 122. (122 + 1) / 2 = 61.5, again approximately the observed average probe count.

**Random Probing – 78%**

Random probing is a modification of linear probing, and as such requires a different formula for approximating the expected number of probes to find a key. The formula is:   
-(1 / alpha) \* ln(1 - alpha). In this case our alpha value is 78%, or 0.78. Using the formula we approximate: -(1 / 0.78) \* ln(1 – 0.78) = -1.28205… \* ln(0.22) = ~1.94 probes to locate a key. This result is better than linear probing at 78% because of how random probing improves upon linear probing; by distributing keys randomly throughout the table, large primary clusters are avoided, meaning there is less chance of a collision happening on insertion into the table, and a primary cluster forming as a result. Instead, secondary clusters form, which are essentially the remnants of broken up primary clusters that would normally result using linear probing. These tend to be relatively small when compared to primary clusters, so they do not affect performance as much as primary clusters would. Again, however, this method for collision handling relies heavily on the use of a good hash function as defined previously. Also as shown previously, our hash function in this case does not meet the requirements of a good hash function, and tends to produce address 65 as a result because any key that contains four spaces at the end will hash to address 65. This means that any improved performance that would’ve been gained by distributing the primary cluster throughout the table is lost because almost all keys used in our test hash to address 65. Because of this, our search algorithm almost always starts at address 65, and because we use the same pseudo random sequence of addresses to both insert and search through the table, what we end up with in this case is again a linear search, where the next address is determined by a predefined sequence rather than simply adding 1. As stated previously, a linear search has an average comparison count of (n + 1) / 2. In this case, n is roughly the number of keys inserted into the table (100) as opposed to the size of the primary cluster. This is because in this particular case, we have a pseudo-primary-cluster of roughly the same size as we did using linear probing, due to almost all keys hashing to address 65 and then using the same pseudo-random sequence starting at address 65 to jump around the table. (100 + 1) / 2 = 50.5, once again approximately our observed average probes (49.18). One thing to note in this case is that our average of 49.18 probes using random probing is very slightly better than our previous average of 49.87 probes using linear probing. This is due to random probing reducing the chance that a collision will be caused by a cluster blocking the insertion spot as opposed to a previous hash already existing at that spot. Unfortunately there are only three keys in our example that could have benefited from this, so in this case the most performance could have been improved due to random probing is very small. (In this example, the performance increase was only 0.69 probes.)

**Random Probing – 95%**

As stated before, the formula to approximate the number of probes expected to find a key using random probing is: -(1 / alpha) \* ln(1 - alpha). At an alpha value of 95%, or 0.95, our expected number of probes is: -(1 / 0.95) \* ln(1 – 0.95) = -1.05263… \* ln(0.05) = ~3.15 probes to locate a key. This is result is much more optimistic than our actual average number of probes (60.52). Once again, our collision handling method falls victim to the inadequacy of a bad hash function. As explained previously, because almost all keys hash to address 65, we end up with a pseudo-primary-cluster starting at address 65. Because our search function starts from the hash address of whatever key we are searching for, our search function almost always starts at address 65 and then iterates through the same sequence of pseudo-random numbers until it arrives at an empty spot or the key, making it as if we were really just searching through a large primary cluster starting at address 65 using linear probing. Because our table is dominated by keys with a hash address of 65 (only three keys don’t have that hash address), this is the same as performing a linear search, and so our expected number of probes in this case would be (n + 1) / 2 where n is the number of keys in the table (only 3 keys so far have had a chance at not falling into the pseudo-primary-cluster, so in this case it is essentially all keys). (122 + 1) / 2 = 61.5, approximately the average number of probes observed. As with the previous random probe example, our observed average using random probing is slightly better than out observed average using linear probing (60.52 as opposed to 61.02). This difference can be attributed to the increased chance that an initial collision using random probing will not be the result of a cluster forming over the address. Again, only three keys could have taken advantage of this since almost all keys hashed to address 65, so the total performance gain is insignificant at best (a decrease of only 0.97 probes).

**Improved Hash Function Analysis**

The following data was obtained by running the same procedure used to generate the analysis data for the bad hash function, but instead replacing use of the bad hash algorithm with the following improved hashing algorithm:

with Interfaces; use Interfaces;

function GetGoodHashAddress (

Key : in String16)

return Long\_Integer is

type String4 is new String

(1 .. 4);

function StringToInt is

new Ada.Unchecked\_Conversion(String4, Long\_Integer);

Address : Long\_Integer;

begin

Address := (StringToInt(String4(Key(1..4)))/(2 \*\* 5)) + (StringToInt(String4(Key(5..8)))/(2 \*\* 5)) + (StringToInt(String4(Key(9..12)))/(2 \*\* 5)) + (StringToInt(String4(Key(13..16)))/(2 \*\* 5));

Address := Address \*\* 2;

Address := Long\_Integer(Shift\_Left(Unsigned\_32(Address), 12));

Address := Long\_Integer(Shift\_Right(Unsigned\_32(Address), 25));

return abs Address + 1;

end GetGoodHashAddress;

What makes this hash function better than the previous? The previous hash function relied on a weak folding procedure followed by the division-remainder technique. First, folding is best done utilizing all parts of the key so that a change to any part of the key will affect the outcome of the fold, as in theory this produces the most unique fold result. This is where the previous hash function first suffers; its folding procedure relies on the last 4 characters of the key only. This means that the first twelve characters of the key do not affect the address in any way. Even worse, because Strings tend to be left-aligned, characters farther to the right tend to be space characters. In this case, if the last 4 characters are spaces, the hash function produces the same address no matter what the first twelve characters are. The new hash function utilizes a much more effective folding technique; by breaking the key up into 32-bit chunks, shifting to the right five to avoid overflow, and then adding the numbers together, we are guaranteed a folding result that relies on the entirety of the key, making the fold more likely to be unique for every key. Second, the bad hash function uses a technique called the division-remainder technique to generate the final address for the table. While this technique is by no means bad, in this case it is not well suited to the job. The effectiveness of the division-remainder technique heavily depends on the size of the table being used. Theoretically, the best table sizes are prime numbers as they will do the best job at randomizing addresses. If a prime cannot be used, the next safest bet is an odd number. The worst table sizes are even numbers because they tend to evenly divide things too often or produce even addresses. Table sizes that should be avoided are ones which are a power of the numeric base being used. If we look at the table size for the bad hash function, it happens to be 128. This number is not prime, not odd but even, and worst of all is a power of the numeric base 2 (2^7). This means that when we obtain the remainder from our fold result, we are actually just chopping off the last 7 bits to use as an address. This strategy is ineffective because it means the bottom 7 bits of our fold determine our key, meaning any values in the fold beyond the first 7 bits will not affect the final address. This translates into characters 14 and 16 for big-endian systems or characters 13 and 15 for little-endian systems being the **only** characters in the entire 16 character key that have any effect on the final address. Our new hash function improves on this method by using a technique that is less dependent on table size: Square-and-Extract-N-bits. First, we square our fold value once it is obtained. Squaring does two things: it ensures our output is positive, and it produces a number entirely dependent on the fold. If we had extracted 7 bits from the middle before squaring, our address would rely too much on numbers towards the middle of the fold, and consequently letters towards the middle of their respective four letter chunks. While this would still be more dependent than the previous algorithm on the key, without squaring it will never be as dependent as it could be. Once squared, the bits that tend to be most evenly affected by values in the fold are those in the middle, so we shift left 12 and then right 25 to extract the middle 7 bits. The reason we don’t use bits on the outer edge is because those bits tend to not be as affected as the bits in the middle. Extracting the right 7 bits would be the same as the division remainder technique in this case, and would too often lead to numbers affected by space characters or the right most characters of each four letter chunk, while extracting from the left would tend to lead to numbers more affected by significant bits rather than insignificant ones, in this case the left most characters of each four letter chunk. Again, the more we rely on the entirety of the key, the better out hash will be, and in this case that means extracting from the middle. By utilizing a more effective folding technique and a more suited technique to produce the final outcome, we now have a hash function that severely outperforms the previous one.

|  |  |  |  |
| --- | --- | --- | --- |
| **Good Hash – Linear Probing at 78% full.** | **Good Hash – Linear Probing at 95% full.** | **Good Hash – Random Probing at 78% full.** | **Good Hash – Random Probing at 95% full.** |
| Hash Table Contents:  # 1: [marsupial ]  # 2: [ ]  # 3: [ ]  # 4: [malfeasance ]  # 5: [ ]  # 6: [Jordon ]  # 7: [mole ]  # 8: [ ]  # 9: [Necromancer ]  # 10: [Person ]  # 11: [sponsor ]  # 12: [seal ]  # 13: [seat ]  # 14: [prologue ]  # 15: [ ]  # 16: [Charles ]  # 17: [ ]  # 18: [ ]  # 19: [ ]  # 20: [cauldron ]  # 21: [ ]  # 22: [Syncopate ]  # 23: [compass ]  # 24: [ ]  # 25: [ ]  # 26: [constrain ]  # 27: [Joseph ]  # 28: [lazy ]  # 29: [Maiden ]  # 30: [diastase ]  # 31: [Judy ]  # 32: [Veronica ]  # 33: [Adam ]  # 34: [Kyle ]  # 35: [Wade ]  # 36: [Yolk ]  # 37: [John ]  # 38: [Christopher ]  # 39: [inlet ]  # 40: [James ]  # 41: [Fernando ]  # 42: [ichthyosaur ]  # 43: [Todd ]  # 44: [brutalize ]  # 45: [labial ]  # 46: [sacrament ]  # 47: [ ]  # 48: [ ]  # 49: [Rush ]  # 50: [ ]  # 51: [ ]  # 52: [sparse ]  # 53: [Scott ]  # 54: [quandary ]  # 55: [option ]  # 56: [Northwest ]  # 57: [Wine ]  # 58: [Michael ]  # 59: [Matthew ]  # 60: [poison ]  # 61: [forgetful ]  # 62: [wet ]  # 63: [guard ]  # 64: [Octavio ]  # 65: [orthodontist ]  # 66: [Constriction ]  # 67: [tiller ]  # 68: [history ]  # 69: [ ]  # 70: [ ]  # 71: [Afterwards ]  # 72: [noggin ]  # 73: [ratiocination ]  # 74: [ ]  # 75: [ ]  # 76: [ ]  # 77: [ ]  # 78: [ ]  # 79: [ ]  # 80: [prepossess ]  # 81: [ ]  # 82: [ ]  # 83: [Eye ]  # 84: [Misogamist ]  # 85: [Fabricate ]  # 86: [cap ]  # 87: [Fabulous ]  # 88: [live ]  # 89: [Robert ]  # 90: [Finagle ]  # 91: [Chris ]  # 92: [Jutty ]  # 93: [Under ]  # 94: [Cook ]  # 95: [magnetic ]  # 96: [Irreversible ]  # 97: [Dustin ]  # 98: [dactyl ]  # 99: [privilege ]  #100: [emperor ]  #101: [keep ]  #102: [gangway ]  #103: [1234567890123456]  #104: [ ]  #105: [Corey ]  #106: [Joel ]  #107: [Honor ]  #108: [Kelly ]  #109: [Perfect ]  #110: [Batbold ]  #111: [wince ]  #112: [rationalize ]  #113: [Jeffrey ]  #114: [Akhila ]  #115: [Screen ]  #116: [taint ]  #117: [Staple ]  #118: [exhilarate ]  #119: [nosegay ]  #120: [toluene ]  #121: [Derek ]  #122: [Lisa ]  #123: [scat ]  #124: [Vinnela ]  #125: [ ]  #126: [ ]  #127: [Clayton ]  #128: [parasympathetic ] | Hash Table Contents:  # 1: [marsupial ]  # 2: [ ]  # 3: [ ]  # 4: [malfeasance ]  # 5: [ ]  # 6: [Jordon ]  # 7: [mole ]  # 8: [crimp ]  # 9: [Necromancer ]  # 10: [Person ]  # 11: [sponsor ]  # 12: [seal ]  # 13: [seat ]  # 14: [prologue ]  # 15: [suffrage ]  # 16: [Charles ]  # 17: [preordain ]  # 18: [minute ]  # 19: [job ]  # 20: [cauldron ]  # 21: [hypodermic ]  # 22: [Syncopate ]  # 23: [compass ]  # 24: [ ]  # 25: [ ]  # 26: [constrain ]  # 27: [Joseph ]  # 28: [lazy ]  # 29: [Maiden ]  # 30: [diastase ]  # 31: [Judy ]  # 32: [Veronica ]  # 33: [Adam ]  # 34: [Kyle ]  # 35: [Wade ]  # 36: [Yolk ]  # 37: [John ]  # 38: [Christopher ]  # 39: [inlet ]  # 40: [James ]  # 41: [Fernando ]  # 42: [ichthyosaur ]  # 43: [Todd ]  # 44: [brutalize ]  # 45: [labial ]  # 46: [sacrament ]  # 47: [luculent ]  # 48: [academy ]  # 49: [Rush ]  # 50: [monolith ]  # 51: [oak ]  # 52: [sparse ]  # 53: [Scott ]  # 54: [quandary ]  # 55: [option ]  # 56: [Northwest ]  # 57: [Wine ]  # 58: [Michael ]  # 59: [Matthew ]  # 60: [poison ]  # 61: [forgetful ]  # 62: [wet ]  # 63: [guard ]  # 64: [Octavio ]  # 65: [orthodontist ]  # 66: [Constriction ]  # 67: [tiller ]  # 68: [history ]  # 69: [withal ]  # 70: [pennywort ]  # 71: [Afterwards ]  # 72: [noggin ]  # 73: [ratiocination ]  # 74: [wiper ]  # 75: [haberdasher ]  # 76: [amputate ]  # 77: [brash ]  # 78: [consummation ]  # 79: [epicure ]  # 80: [prepossess ]  # 81: [meridian ]  # 82: [ ]  # 83: [Eye ]  # 84: [Misogamist ]  # 85: [Fabricate ]  # 86: [cap ]  # 87: [Fabulous ]  # 88: [live ]  # 89: [Robert ]  # 90: [Finagle ]  # 91: [Chris ]  # 92: [Jutty ]  # 93: [Under ]  # 94: [Cook ]  # 95: [magnetic ]  # 96: [Irreversible ]  # 97: [Dustin ]  # 98: [dactyl ]  # 99: [privilege ]  #100: [emperor ]  #101: [keep ]  #102: [gangway ]  #103: [1234567890123456]  #104: [prevalent ]  #105: [Corey ]  #106: [Joel ]  #107: [Honor ]  #108: [Kelly ]  #109: [Perfect ]  #110: [Batbold ]  #111: [wince ]  #112: [rationalize ]  #113: [Jeffrey ]  #114: [Akhila ]  #115: [Screen ]  #116: [taint ]  #117: [Staple ]  #118: [exhilarate ]  #119: [nosegay ]  #120: [toluene ]  #121: [Derek ]  #122: [Lisa ]  #123: [scat ]  #124: [Vinnela ]  #125: [zygote ]  #126: [indulgent ]  #127: [Clayton ]  #128: [parasympathetic ] | Hash Table Contents:  # 1: [option ]  # 2: [forgetful ]  # 3: [ ]  # 4: [mole ]  # 5: [ ]  # 6: [Jordon ]  # 7: [ ]  # 8: [seat ]  # 9: [Necromancer ]  # 10: [Person ]  # 11: [labial ]  # 12: [ ]  # 13: [nosegay ]  # 14: [prologue ]  # 15: [Perfect ]  # 16: [Charles ]  # 17: [quandary ]  # 18: [inlet ]  # 19: [ ]  # 20: [cauldron ]  # 21: [poison ]  # 22: [Syncopate ]  # 23: [ ]  # 24: [Veronica ]  # 25: [rationalize ]  # 26: [constrain ]  # 27: [Joseph ]  # 28: [Fabricate ]  # 29: [Maiden ]  # 30: [ ]  # 31: [Judy ]  # 32: [ichthyosaur ]  # 33: [Adam ]  # 34: [Kyle ]  # 35: [gangway ]  # 36: [ ]  # 37: [John ]  # 38: [Christopher ]  # 39: [brutalize ]  # 40: [James ]  # 41: [ ]  # 42: [sponsor ]  # 43: [Todd ]  # 44: [ ]  # 45: [ ]  # 46: [sacrament ]  # 47: [ ]  # 48: [privilege ]  # 49: [Rush ]  # 50: [Akhila ]  # 51: [guard ]  # 52: [sparse ]  # 53: [Scott ]  # 54: [lazy ]  # 55: [tiller ]  # 56: [Northwest ]  # 57: [Wine ]  # 58: [Michael ]  # 59: [Matthew ]  # 60: [ ]  # 61: [orthodontist ]  # 62: [wet ]  # 63: [ ]  # 64: [Octavio ]  # 65: [ ]  # 66: [Constriction ]  # 67: [diastase ]  # 68: [Staple ]  # 69: [Fernando ]  # 70: [Yolk ]  # 71: [Afterwards ]  # 72: [magnetic ]  # 73: [ratiocination ]  # 74: [ ]  # 75: [keep ]  # 76: [ ]  # 77: [history ]  # 78: [ ]  # 79: [seal ]  # 80: [prepossess ]  # 81: [Cook ]  # 82: [malfeasance ]  # 83: [Eye ]  # 84: [Misogamist ]  # 85: [toluene ]  # 86: [cap ]  # 87: [Fabulous ]  # 88: [live ]  # 89: [Robert ]  # 90: [Finagle ]  # 91: [Chris ]  # 92: [Jutty ]  # 93: [Under ]  # 94: [ ]  # 95: [taint ]  # 96: [Irreversible ]  # 97: [Dustin ]  # 98: [dactyl ]  # 99: [emperor ]  #100: [ ]  #101: [noggin ]  #102: [ ]  #103: [1234567890123456]  #104: [ ]  #105: [Corey ]  #106: [Joel ]  #107: [Honor ]  #108: [Kelly ]  #109: [ ]  #110: [Batbold ]  #111: [wince ]  #112: [ ]  #113: [Jeffrey ]  #114: [ ]  #115: [ ]  #116: [ ]  #117: [Wade ]  #118: [exhilarate ]  #119: [Screen ]  #120: [marsupial ]  #121: [Derek ]  #122: [Lisa ]  #123: [scat ]  #124: [Vinnela ]  #125: [compass ]  #126: [ ]  #127: [Clayton ]  #128: [parasympathetic ] | Hash Table Contents:  # 1: [option ]  # 2: [forgetful ]  # 3: [ ]  # 4: [mole ]  # 5: [zygote ]  # 6: [Jordon ]  # 7: [crimp ]  # 8: [seat ]  # 9: [Necromancer ]  # 10: [Person ]  # 11: [labial ]  # 12: [ ]  # 13: [nosegay ]  # 14: [prologue ]  # 15: [Perfect ]  # 16: [Charles ]  # 17: [quandary ]  # 18: [inlet ]  # 19: [epicure ]  # 20: [cauldron ]  # 21: [poison ]  # 22: [Syncopate ]  # 23: [pennywort ]  # 24: [Veronica ]  # 25: [rationalize ]  # 26: [constrain ]  # 27: [Joseph ]  # 28: [Fabricate ]  # 29: [Maiden ]  # 30: [luculent ]  # 31: [Judy ]  # 32: [ichthyosaur ]  # 33: [Adam ]  # 34: [Kyle ]  # 35: [gangway ]  # 36: [ ]  # 37: [John ]  # 38: [Christopher ]  # 39: [brutalize ]  # 40: [James ]  # 41: [job ]  # 42: [sponsor ]  # 43: [Todd ]  # 44: [prevalent ]  # 45: [brash ]  # 46: [sacrament ]  # 47: [oak ]  # 48: [privilege ]  # 49: [Rush ]  # 50: [Akhila ]  # 51: [guard ]  # 52: [sparse ]  # 53: [Scott ]  # 54: [lazy ]  # 55: [tiller ]  # 56: [Northwest ]  # 57: [Wine ]  # 58: [Michael ]  # 59: [Matthew ]  # 60: [withal ]  # 61: [orthodontist ]  # 62: [wet ]  # 63: [wiper ]  # 64: [Octavio ]  # 65: [meridian ]  # 66: [Constriction ]  # 67: [diastase ]  # 68: [Staple ]  # 69: [Fernando ]  # 70: [Yolk ]  # 71: [Afterwards ]  # 72: [magnetic ]  # 73: [ratiocination ]  # 74: [ ]  # 75: [keep ]  # 76: [minute ]  # 77: [history ]  # 78: [ ]  # 79: [seal ]  # 80: [prepossess ]  # 81: [Cook ]  # 82: [malfeasance ]  # 83: [Eye ]  # 84: [Misogamist ]  # 85: [toluene ]  # 86: [cap ]  # 87: [Fabulous ]  # 88: [live ]  # 89: [Robert ]  # 90: [Finagle ]  # 91: [Chris ]  # 92: [Jutty ]  # 93: [Under ]  # 94: [consummation ]  # 95: [taint ]  # 96: [Irreversible ]  # 97: [Dustin ]  # 98: [dactyl ]  # 99: [emperor ]  #100: [ ]  #101: [noggin ]  #102: [hypodermic ]  #103: [1234567890123456]  #104: [suffrage ]  #105: [Corey ]  #106: [Joel ]  #107: [Honor ]  #108: [Kelly ]  #109: [academy ]  #110: [Batbold ]  #111: [wince ]  #112: [amputate ]  #113: [Jeffrey ]  #114: [monolith ]  #115: [indulgent ]  #116: [haberdasher ]  #117: [Wade ]  #118: [exhilarate ]  #119: [Screen ]  #120: [marsupial ]  #121: [Derek ]  #122: [Lisa ]  #123: [scat ]  #124: [Vinnela ]  #125: [compass ]  #126: [preordain ]  #127: [Clayton ]  #128: [parasympathetic ] |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Minimum Probes** | **Maximum Probes** | **Average Probes** | **Expected Probes** |
| **Linear Probing at 78%** | 1 | 14 | 2.62 | 2.77 |
| **Linear Probing at 95%** | 1 | 38 | 5.48 | 10.5 |
| **Random Probing at 78%** | 1 | 17 | 2.33 | 1.94 |
| **Random Probing at 95%** | 1 | 48 | 5.95 | 3.15 |

**Probing Results for First and Last 30 Keys Inserted**

**Linear Probing – 78%**

As we calculated before, the expected number of probes to find a key for a hash table utilizing linear collision at 78% full was ~2.77 probes. While our bad hash algorithm performed with a dismal 49.87 probes on average, our new hash functions averages out to a low 2.62 probes per key, a whole 0.15 probes less than our theoretical expectations. The important difference here is the hash function; we are now using a hash function that meets the qualifications previously laid out for a good hash function: it produces an address based entirely on the key, and it produces a unique address for almost every key. Because we now have an address distribution that more closely resembles unique randomness, the benefits of linear probing as a collision handling method begin take place. Across the table we can see the formation of numerous small primary clusters. While these clusters still have the ill effect of producing collisions for slots in the hash table that would’ve otherwise produced no collision, the important thing to note is that the clusters are much smaller than in our previous example, which had one gigantic primary cluster covering up almost the whole table, so the downsides brought upon by primary clusters are not as apparent at lower levels of fullness, and especially in this example. The slight outperformance of the theoretical results is most likely due to the sequence of keys fed into program, as a different set of keys, or even the same keys in a different order would produce a different key distribution and different instances of collision. In this case, the sequence of keys has actually made our algorithm perform slightly better than expected.

**Linear Probing – 95%**

Previous calculations for the theoretical expected number of probes using linear probing for a table at 95% capacity produced the result of ~10.5 probes per key. At 95%, our bad hash function went up to 61.02 average probes per key, but our new hash function has again improved on our bad hash function as well as theoretical prediction with a very low 5.48 probes on average. Again, the effects of a good hash function become increasingly prevalent; the more likely keys produce unique addresses, the less likely collisions occur and the less pronounced of an effect primary clusters have on table lookup. In this particular case, the set of keys used probably produced far less collisions and consequently primary clusters than theoretically expected. Other factors out of control of the hash function could also lead to very bad performance at this level of capacity (those factors include the keys used and the order they’re inserted).

**Random Probing – 78%**

Random probing at 78% should, theoretically and as calculated previously, only take on average 1.94 probes to find a key within a table that is using random probing for collision handling. Our old hash function remained barely affected in terms of performance by the use of random probing, with an average of 49.18 probes per key to find its location. Our new hash function improves on this greatly with an average of 2.33 probes per key. Here we can see the use of random probing as a performance boost over linear probing. Because the main problem with linear probing is the formation of primary clusters, the ability for random probing to break those clusters up into smaller secondary ones spread out over the table coupled with a better hash function that produces a more unique and random distribution of addresses means we should see a performance level better than that of not only our bad hash algorithm, but also of linear probing. While the jump in performance isn’t particularly astonishing (0.29 probes less than the linear probe average), the increase in performance large enough to justify the use of random probing, especially at lower average levels like these where a smaller jump is more significant. It should be noted that in this particular case, random probing actually failed to meet the expected result of 1.94 probes per key, and this again can be most likely attributed to this particular sequence of keys producing a result specific to this example. Had a different set of keys been used, a different order of keys be used, or a differ sample of keys be selected for analysis, such as the first 60, we might’ve seen an increase in performance of a greater magnitude, which might be closer to or even beat the theoretical prediction. In practice though, repeated analysis of multiple examples of this nature should on average yield an average probe count close to 1.94.

**Random Probing – 95%**

As determined in our previous analysis of the bad hash function using random probing at 95% capacity, the theoretical expected number of probes to find the location of a key is 3.15. Our bad hash function falls short of this theoretical mark, with a horrible average of 60.52 probes per key in this case. With our new hash function, the average number of probes needed to find a key was brought down to only 5.95. What is made apparent here is the effectiveness of our hash function at utilizing a collision handling method that relies on a good hash function. Random probing relies on a hash function that produces unique, random keys, and by inserting a new hashing algorithm that does that, our performance jumped down a very large 54.57 average probes. In fact, all cases analyzed so far have shown that by using a good hash function, collision handling techniques become much more effective and convenient. What is not apparent here, however, is how exactly random probing is a better method than linear probing. While our results certainly improved from one hash function to the next, the result of 5.95 average probes is twice the expected theoretical result. The most likely reasons for this apparent lack of performance can be attributed, again, to the keys used, the order they were inserted, and the keys selected as a sample for our analysis results. Another factor that might greatly affect things is the pseudo-random number generator used. In this case, the hash addresses might not have lined up as well as one would’ve liked with the random number sequence, and in the off chance this is the case, the result would be more collisions, more secondary cluster, and thus more probes to find key up retrieval.

**SOURCE CODE**

HashTest.adb

with Ada.Unchecked\_Conversion;

with Ada.Text\_IO;

use Ada.Text\_IO;

with Ada.Integer\_Text\_IO;

use Ada.Integer\_Text\_IO;

with Ada.Float\_Text\_IO;

use Ada.Float\_Text\_IO;

with HashTable128;

with Interfaces; use Interfaces;

procedure HashTest is

type String16 is new String

(1 .. 16);

package Long\_IO is new Ada.Text\_IO.Integer\_IO(Long\_Integer);

use Long\_IO;

function GetBadHashAddress (

Key : in String16)

return Long\_Integer is

type String2 is new String

(1 .. 2);

function StringToInt is

new Ada.Unchecked\_Conversion(String2, Long\_Integer);

Address : Long\_Integer;

begin

Address := (abs (StringToInt(String2(Key(13..14))))) + (abs (StringToInt(String2(Key(15..16)))));

return (Address mod 128) + 1;

end GetBadHashAddress;

function GetGoodHashAddress (

Key : in String16)

return Long\_Integer is

type String4 is new String

(1 .. 4);

function StringToInt is

new Ada.Unchecked\_Conversion(String4, Long\_Integer);

Address : Long\_Integer;

begin

Address := (StringToInt(String4(Key(1..4)))/(2 \*\* 5)) + (StringToInt(String4(Key(5..8)))/(2 \*\* 5)) + (StringToInt(String4(Key(9..12)))/(2 \*\* 5)) + (StringToInt(String4(Key(13..16)))/(2 \*\* 5));

Address := Address \*\* 2;

Address := Long\_Integer(Shift\_Left(Unsigned\_32(Address), 12));

Address := Long\_Integer(Shift\_Right(Unsigned\_32(Address), 25));

return abs Address + 1;

end GetGoodHashAddress;

procedure PutString16 (

Key : in String16) is

begin

Put(String(Key));

end PutString16;

begin

--Linear 78%

declare

package HashTable is new HashTable128(String16, GetBadHashAddress, PutString16);

use HashTable;

TestTable : LinearHashTable128;

Percent\_Full : Float;

Input : File\_Type;

Probes : Integer;

End\_Line : Integer := 1;

Min : Integer := 128;

Max : Integer := 0;

Average : Integer := 0;

Key : String16;

begin

SetupHashTable(TestTable, "Table1", " ");

Open(Input, In\_File, "Words200D16.txt");

loop

Get(Input, String(Key));

InsertLinearProbe(TestTable, Key, Percent\_Full);

End\_Line := End\_Line + 1;

exit when Percent\_Full > 0.78 or End\_Of\_File(Input);

end loop;

PrintTable(TestTable);

Put("Linear Table is ");

Put(Percent\_Full \* 100.0, 2, 2, 0);

Put\_Line("% Full.");

Close(Input);

Open(Input, In\_File, "Words200D16.txt");

Put\_Line("Probing First 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchLinearProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Set\_Line(Input, Positive\_Count(End\_Line - 30));

Put\_Line("Probing Last 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchLinearProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Close(Input);

Put("Minimum Probes: ");

Put(Min, 3);

New\_Line;

Put("Maximum Probes: ");

Put(Max, 3);

New\_Line;

Put("Average Probes: ");

Put(Float(Average) / 60.0, 2, 2, 0);

New\_Line;

New\_Line;

end;

--Linear 95%

declare

package HashTable is new HashTable128(String16, GetBadHashAddress, PutString16);

use HashTable;

TestTable : LinearHashTable128;

Percent\_Full : Float;

Input : File\_Type;

Probes : Integer;

End\_Line : Integer := 1;

Min : Integer := 128;

Max : Integer := 0;

Average : Integer := 0;

Key : String16;

begin

End\_Line := 1;

SetupHashTable(TestTable, "Table2", " ");

Open(Input, In\_File, "Words200D16.txt");

loop

Get(Input, String(Key));

InsertLinearProbe(TestTable, Key, Percent\_Full);

End\_Line := End\_Line + 1;

exit when Percent\_Full > 0.95 or End\_Of\_File(Input);

end loop;

PrintTable(TestTable);

Put("Linear Table is ");

Put(Percent\_Full \* 100.0, 2, 2, 0);

Put\_Line("% Full.");

Close(Input);

Open(Input, In\_File, "Words200D16.txt");

Put\_Line("Probing First 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchLinearProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Set\_Line(Input, Positive\_Count(End\_Line - 30));

Put\_Line("Probing Last 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchLinearProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Close(Input);

Put("Minimum Probes: ");

Put(Min, 3);

New\_Line;

Put("Maximum Probes: ");

Put(Max, 3);

New\_Line;

Put("Average Probes: ");

Put(Float(Average) / 60.0, 2, 2, 0);

New\_Line;

New\_Line;

end;

--Random 78%

declare

package HashTable is new HashTable128(String16, GetBadHashAddress, PutString16);

use HashTable;

TestTable : RandomHashTable128;

Percent\_Full : Float;

Input : File\_Type;

Probes : Integer;

End\_Line : Integer := 1;

Min : Integer := 128;

Max : Integer := 0;

Average : Integer := 0;

Key : String16;

begin

End\_Line := 1;

SetupHashTable(TestTable, "Table3", " ");

Open(Input, In\_File, "Words200D16.txt");

loop

Get(Input, String(Key));

InsertRandomProbe(TestTable, Key, Percent\_Full);

End\_Line := End\_Line + 1;

exit when Percent\_Full > 0.78 or End\_Of\_File(Input);

end loop;

PrintTable(TestTable);

Put("Random Table is ");

Put(Percent\_Full \* 100.0, 2, 2, 0);

Put\_Line("% Full.");

Close(Input);

Open(Input, In\_File, "Words200D16.txt");

Put\_Line("Probing First 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchRandomProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Set\_Line(Input, Positive\_Count(End\_Line - 30));

Put\_Line("Probing Last 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchRandomProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Close(Input);

Put("Minimum Probes: ");

Put(Min, 3);

New\_Line;

Put("Maximum Probes: ");

Put(Max, 3);

New\_Line;

Put("Average Probes: ");

Put(Float(Average) / 60.0, 2, 2, 0);

New\_Line;

New\_Line;

end;

--Random 95%

declare

package HashTable is new HashTable128(String16, GetBadHashAddress, PutString16);

use HashTable;

TestTable : RandomHashTable128;

Percent\_Full : Float;

Input : File\_Type;

Probes : Integer;

End\_Line : Integer := 1;

Min : Integer := 128;

Max : Integer := 0;

Average : Integer := 0;

Key : String16;

begin

End\_Line := 1;

SetupHashTable(TestTable, "Table4", " ");

Open(Input, In\_File, "Words200D16.txt");

loop

Get(Input, String(Key));

InsertRandomProbe(TestTable, Key, Percent\_Full);

End\_Line := End\_Line + 1;

exit when Percent\_Full > 0.95 or End\_Of\_File(Input);

end loop;

PrintTable(TestTable);

Put("Random Table is ");

Put(Percent\_Full \* 100.0, 2, 2, 0);

Put\_Line("% Full.");

Close(Input);

Open(Input, In\_File, "Words200D16.txt");

Put\_Line("Probing First 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchRandomProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Set\_Line(Input, Positive\_Count(End\_Line - 30));

Put\_Line("Probing Last 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchRandomProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Close(Input);

Put("Minimum Probes: ");

Put(Min, 3);

New\_Line;

Put("Maximum Probes: ");

Put(Max, 3);

New\_Line;

Put("Average Probes: ");

Put(Float(Average) / 60.0, 2, 2, 0);

New\_Line;

New\_Line;

end;

Put\_Line("==============================================================");

Put\_Line("= GOOD HASH FUNCTION HERE =");

Put\_Line("==============================================================");

New\_Line;

--Linear 78%

declare

package HashTable is new HashTable128(String16, GetGoodHashAddress, PutString16);

use HashTable;

TestTable : LinearHashTable128;

Percent\_Full : Float;

Input : File\_Type;

Probes : Integer;

End\_Line : Integer := 1;

Min : Integer := 128;

Max : Integer := 0;

Average : Integer := 0;

Key : String16;

begin

SetupHashTable(TestTable, "Table5", " ");

Open(Input, In\_File, "Words200D16.txt");

loop

Get(Input, String(Key));

InsertLinearProbe(TestTable, Key, Percent\_Full);

End\_Line := End\_Line + 1;

exit when Percent\_Full > 0.78 or End\_Of\_File(Input);

end loop;

PrintTable(TestTable);

Put("Linear Table is ");

Put(Percent\_Full \* 100.0, 2, 2, 0);

Put\_Line("% Full.");

Close(Input);

Open(Input, In\_File, "Words200D16.txt");

Put\_Line("Probing First 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchLinearProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Set\_Line(Input, Positive\_Count(End\_Line - 30));

Put\_Line("Probing Last 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchLinearProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Close(Input);

Put("Minimum Probes: ");

Put(Min, 3);

New\_Line;

Put("Maximum Probes: ");

Put(Max, 3);

New\_Line;

Put("Average Probes: ");

Put(Float(Average) / 60.0, 2, 2, 0);

New\_Line;

New\_Line;

end;

--Linear 95%

declare

package HashTable is new HashTable128(String16, GetGoodHashAddress, PutString16);

use HashTable;

TestTable : LinearHashTable128;

Percent\_Full : Float;

Input : File\_Type;

Probes : Integer;

End\_Line : Integer := 1;

Min : Integer := 128;

Max : Integer := 0;

Average : Integer := 0;

Key : String16;

begin

End\_Line := 1;

SetupHashTable(TestTable, "Table6", " ");

Open(Input, In\_File, "Words200D16.txt");

loop

Get(Input, String(Key));

InsertLinearProbe(TestTable, Key, Percent\_Full);

End\_Line := End\_Line + 1;

exit when Percent\_Full > 0.95 or End\_Of\_File(Input);

end loop;

PrintTable(TestTable);

Put("Linear Table is ");

Put(Percent\_Full \* 100.0, 2, 2, 0);

Put\_Line("% Full.");

Close(Input);

Open(Input, In\_File, "Words200D16.txt");

Put\_Line("Probing First 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchLinearProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Set\_Line(Input, Positive\_Count(End\_Line - 30));

Put\_Line("Probing Last 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchLinearProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Close(Input);

Put("Minimum Probes: ");

Put(Min, 3);

New\_Line;

Put("Maximum Probes: ");

Put(Max, 3);

New\_Line;

Put("Average Probes: ");

Put(Float(Average) / 60.0, 2, 2, 0);

New\_Line;

New\_Line;

end;

--Random 78%

declare

package HashTable is new HashTable128(String16, GetGoodHashAddress, PutString16);

use HashTable;

TestTable : RandomHashTable128;

Percent\_Full : Float;

Input : File\_Type;

Probes : Integer;

End\_Line : Integer := 1;

Min : Integer := 128;

Max : Integer := 0;

Average : Integer := 0;

Key : String16;

begin

End\_Line := 1;

SetupHashTable(TestTable, "Table7", " ");

Open(Input, In\_File, "Words200D16.txt");

loop

Get(Input, String(Key));

InsertRandomProbe(TestTable, Key, Percent\_Full);

End\_Line := End\_Line + 1;

exit when Percent\_Full > 0.78 or End\_Of\_File(Input);

end loop;

PrintTable(TestTable);

Put("Random Table is ");

Put(Percent\_Full \* 100.0, 2, 2, 0);

Put\_Line("% Full.");

Close(Input);

Open(Input, In\_File, "Words200D16.txt");

Put\_Line("Probing First 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchRandomProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Set\_Line(Input, Positive\_Count(End\_Line - 30));

Put\_Line("Probing Last 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchRandomProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Close(Input);

Put("Minimum Probes: ");

Put(Min, 3);

New\_Line;

Put("Maximum Probes: ");

Put(Max, 3);

New\_Line;

Put("Average Probes: ");

Put(Float(Average) / 60.0, 2, 2, 0);

New\_Line;

New\_Line;

end;

--Random 95%

declare

package HashTable is new HashTable128(String16, GetGoodHashAddress, PutString16);

use HashTable;

TestTable : RandomHashTable128;

Percent\_Full : Float;

Input : File\_Type;

Probes : Integer;

End\_Line : Integer := 1;

Min : Integer := 128;

Max : Integer := 0;

Average : Integer := 0;

Key : String16;

begin

End\_Line := 1;

SetupHashTable(TestTable, "Table8", " ");

Open(Input, In\_File, "Words200D16.txt");

loop

Get(Input, String(Key));

InsertRandomProbe(TestTable, Key, Percent\_Full);

End\_Line := End\_Line + 1;

exit when Percent\_Full > 0.95 or End\_Of\_File(Input);

end loop;

PrintTable(TestTable);

Put("Random Table is ");

Put(Percent\_Full \* 100.0, 2, 2, 0);

Put\_Line("% Full.");

Close(Input);

Open(Input, In\_File, "Words200D16.txt");

Put\_Line("Probing First 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchRandomProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Set\_Line(Input, Positive\_Count(End\_Line - 30));

Put\_Line("Probing Last 30 Keys...");

for X in 1..30 loop

Get(Input, String(Key));

SearchRandomProbe(TestTable, Key, Probes);

if Probes < Min then

Min := Probes;

end if;

if Probes > Max then

Max := Probes;

end if;

Average := Average + Probes;

end loop;

Close(Input);

Put("Minimum Probes: ");

Put(Min, 3);

New\_Line;

Put("Maximum Probes: ");

Put(Max, 3);

New\_Line;

Put("Average Probes: ");

Put(Float(Average) / 60.0, 2, 2, 0);

New\_Line;

New\_Line;

end;

end HashTest;

For the purposes of this lab, I created a very specific hash table object to help facilitate testing of the hashing algorithms:

HashTable128.ads

generic

type ITEM is private;

with function GetHash (

Key : in ITEM)

return Long\_Integer;

with procedure Put (

Key : in ITEM);

package HashTable128 is

type HashTable128 is tagged private;

type LinearHashTable128 is new HashTable128 with private;

type RandomHashTable128 is new HashTable128 with private;

procedure SetupHashTable (

HashTable : in out HashTable128'Class;

Table\_Name : in String;

Default\_Table\_Value : in ITEM);

procedure InsertLinearProbe (

HashTable : in out LinearHashTable128;

Key : in ITEM;

Percent\_Full : out Float);

procedure InsertRandomProbe (

HashTable : in out RandomHashTable128;

Key : in ITEM;

Percent\_Full : out Float);

procedure SearchLinearProbe (

HashTable : in out LinearHashTable128;

Key : in ITEM;

Number\_Of\_Probes : out Integer);

procedure SearchRandomProbe (

HashTable : in out RandomHashTable128;

Key : in ITEM;

Number\_Of\_Probes : out Integer);

procedure PrintTable (

HashTable : in HashTable128'Class);

private

type HashTable128 is tagged

record

Table : String(1..6);

Empty : ITEM;

Count : Integer := 0;

end record;

type LinearHashTable128 is new HashTable128 with null record;

type RandomHashTable128 is new HashTable128 with null record;

type RandomAddress128

(Initial\_Address : Long\_Integer) is

record

Address : Long\_Integer := ((Initial\_Address - 1) \* 4) + 1;

end record;

function GetAddress128 (

RA : in RandomAddress128)

return Long\_Integer;

procedure NextAddress128 (

RA : in out RandomAddress128);

end HashTable128;

HashTable128.adb

with Ada.Text\_IO;

with Ada.Direct\_IO;

package body HashTable128 is

function GetAddress128 (

RA : in RandomAddress128)

return Long\_Integer is

begin

return (RA.Address / 4) + 1;

end GetAddress128;

procedure NextAddress128 (

RA : in out RandomAddress128) is

begin

RA.Address := RA.Address \* 5;

RA.Address := RA.Address mod 512;

end NextAddress128;

procedure SetupHashTable (

HashTable : in out HashTable128'Class;

Table\_Name : in String;

Default\_Table\_Value : in ITEM) is

package Io\_Direct is new Ada.Direct\_IO(ITEM);

use Io\_Direct;

File : File\_Type;

begin

HashTable.Empty := Default\_Table\_Value;

HashTable.Table := Table\_Name;

Create(File, Inout\_File, HashTable.Table);

for I in Positive\_Count range 1..128 loop

Write(File, Default\_Table\_Value, I);

end loop;

Close(File);

end SetupHashTable;

procedure InsertLinearProbe (

HashTable : in out LinearHashTable128;

Key : in ITEM;

Percent\_Full : out Float) is

package Io\_Direct is new Ada.Direct\_IO(ITEM);

use Io\_Direct;

File : File\_Type;

Address : Long\_Integer;

Table\_Val : ITEM;

begin

Open(File, Inout\_file, HashTable.Table);

Address := GetHash(Key);

Read(File, Table\_Val, Positive\_Count(Address));

if Table\_Val = HashTable.Empty then

Write(File, Key, Positive\_Count(Address));

HashTable.Count := HashTable.Count + 1;

else

--Handle Collision

loop

Address := (Address mod 128) + 1;

Read(File, Table\_Val, Positive\_Count(Address));

exit when Table\_Val = HashTable.Empty or Address = GetHash(Key);

end loop;

if Address /= GetHash(Key) then

Write(File, Key, Positive\_Count(Address));

HashTable.Count := HashTable.Count + 1;

end if;

end if;

Percent\_Full := Float(HashTable.Count) / 128.0;

Close(File);

end InsertLinearProbe;

procedure InsertRandomProbe (

HashTable : in out RandomHashTable128;

Key : in ITEM;

Percent\_Full : out Float) is

package Io\_Direct is new Ada.Direct\_IO(ITEM);

use Io\_Direct;

File : File\_Type;

Address : RandomAddress128 (GetHash (Key));

Table\_Val : ITEM;

begin

Open(File, Inout\_File, HashTable.Table);

Read(File, Table\_Val, Positive\_Count(GetAddress128(Address)));

if Table\_Val = HashTable.Empty then

Write(File, Key, Positive\_Count(GetAddress128(Address)));

HashTable.Count := HashTable.Count + 1;

else

loop

NextAddress128(Address);

Read(File, Table\_Val, Positive\_Count(GetAddress128(Address)));

exit when Table\_Val = HashTable.Empty or GetAddress128(Address) = GetHash(Key);

end loop;

if GetAddress128(Address) /= GetHash(Key) then

Write(File, Key, Positive\_Count(GetAddress128(Address)));

HashTable.Count := HashTable.Count + 1;

end if;

end if;

Percent\_Full := Float(HashTable.Count) / 128.0;

Close(File);

end InsertRandomProbe;

procedure SearchLinearProbe (

HashTable : in out LinearHashTable128;

Key : in ITEM;

Number\_Of\_Probes : out Integer) is

package Io\_Direct is new Ada.Direct\_IO(ITEM);

use Io\_Direct;

File : File\_Type;

Table\_Val : ITEM;

Address : Long\_Integer;

begin

Address := GetHash(Key);

Open(File, Inout\_File, HashTable.Table);

Read(File, Table\_Val, Positive\_Count(Address));

Number\_Of\_Probes := 1;

if Table\_Val = Key or Table\_Val = HashTable.Empty then

--Key was found or does not exist, 1 probe

null;

else

--Search for key until we are at key or empty spot

Number\_Of\_Probes := 2;

loop

Address := (Address mod 128) + 1;

Read(File, Table\_Val, Positive\_Count(Address));

exit when Table\_Val = HashTable.Empty or Table\_Val = Key or Address = GetHash(Key);

Number\_Of\_Probes := Number\_Of\_Probes + 1;

end loop;

end if;

Close(File);

end SearchLinearProbe;

procedure SearchRandomProbe (

HashTable : in out RandomHashTable128;

Key : in ITEM;

Number\_Of\_Probes : out Integer) is

package Io\_Direct is new Ada.Direct\_IO(ITEM);

use Io\_Direct;

File : File\_Type;

Table\_Val : ITEM;

Address : RandomAddress128 (GetHash (Key));

begin

Open(File, Inout\_File, HashTable.Table);

Read(File, Table\_Val, Positive\_Count(GetAddress128(Address)));

Number\_Of\_Probes := 1;

if Table\_Val = Key or Table\_Val = HashTable.Empty then

null;

else

Number\_Of\_Probes := 2;

loop

NextAddress128(Address);

Read(File, Table\_Val, Positive\_Count(GetAddress128(Address)));

exit when Table\_Val = HashTable.Empty or Table\_Val = Key or GetAddress128(Address) = GetHash(Key);

Number\_Of\_Probes := Number\_Of\_Probes + 1;

end loop;

end if;

Close(File);

end SearchRandomProbe;

procedure PrintTable (

HashTable : in HashTable128'Class) is

package Long\_Int\_IO is new Ada.Text\_IO.Integer\_IO(Long\_Integer);

use Long\_Int\_IO;

package Io\_Direct is new Ada.Direct\_IO(ITEM);

use Io\_Direct;

File : File\_Type;

Table\_Val : ITEM;

begin

Open(File, Inout\_file, HashTable.Table);

Ada.Text\_IO.Put\_Line("Hash Table Contents:");

for I in Long\_Integer range 1..128 loop

Ada.Text\_IO.Put("#");

Put(I, 3);

Ada.Text\_IO.Put(": [");

Read(File, Table\_Val, Positive\_Count(I));

Put(Table\_Val);

Ada.Text\_IO.Put\_Line("]");

end loop;

Close(File);

end PrintTable;

end HashTable128;