Hayes Crowley

Exercise 3

CS-255-C01 Objects & Algorithms / Prof. Murashkina

09/26/2021

***Collision Resolution Methods for Hash Maps***

The purpose of this document is to briefly describe three common collision resolution methods (linear probing, quadratic probing, and double hashing) used when building hash tables in hash map implementations.

A hash map is an unordered collection of key/value pairs that allow for O(1) constant access, inserts, and deletions. The distinguishing feature of a hash map is that its keys can be any immutable type (string, float, integer, etc.), which allows for quick look ups of values whose keys are not integers (for example, ISO country codes, names of persons, etc.).

The way this is accomplished is by using something called a “hash table,” which is essentially a dynamic bucket array in which each “bucket” contains a key/value pair (a tuple or some user-defined class). Keys are “hashed” into integers using a hash function (typically some sort of polynomial operation involving modular arithmetic and large prime numbers) and inserted into their respective buckets. When getting items out of a hash map, a key is hashed and then accessed from the hash table.

When a hashed key’s bucket is already occupied during insert, we have what is called a “collision”. We do not want to overwrite the current item in the bucket, so we must find the next available bucket to insert our key/value pair into. To do so, we can do something called “probing”, where we increment our hashed key value until we find the next available slot. “Double hashing” is a similar approach, however it involves using an additional hashing function along with the increment. (Note: collision resolution techniques also take place during access, when a key/value pair in an accessed bucket doesn’t match the key lookup.)

**The following collision resolution examples use the given input: [7, 421, 12, 72, 46, 379, 58, 34, 42] and the following primary hash function:**

*// Primary hash function used in all examples*

*procedure hash(key):*

*return key % length(hashTable)*

**The length of our hash table (length(hashTable)) is 11 for all examples.**

1. **Linear Probing Example**

In linear probing, upon encountering a collision, we increment the hashed key + *n*, where *n* is the number of collisions or iterations in the probe, and modulate it with the length of the hash table. The pseudo code for inserts looks like:

*item = new Item(key, someValue)*

*hashedKey = hash(key)*

*n = 0*

*While True:*

*n += 1*

*If hashTable[hashedKey].key == key:*

*hashTable[hashedKey].value = item.value*

*return*

*else If hashTable[hashedKey] is not occupied:*

*hashTable[hashedKey] = item*

*return*

*// iterate cyclically using linear probing*

*hashedKey = (hashedKey + n) % length(hashTable)*

Using my example input, **[7, 421, 12, 72, 46, 379, 58, 34, 42]**, this would look like the following in a tabular representation.

We have no collisions when hashing and inserting 7, 421, 12, 72, 46, and 379 into our table:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | **3** | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|  | 12 | 46 | **421** |  | 379 | 72 | 7 |  |  |  |

However, we encounter a collision when hashing and inserting 58, because hash(58) = 57 % 11 = 3, and the bucket in hashTable[3] is occupied.

Using linear probing, then, we increment the hashedKey (3) plus the number of collisions or iterations in this cycle (1), and modulate that by the length of our table (11).

*3 + 1 % 11 = 4 % 11 = 4*

Our bucket at hashTable[4] is empty, so we can go ahead and insert 58 into this position.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | **4** | 5 | 6 | 7 | 8 | 9 | 10 |
|  | 12 | 46 | 421 | **58** | 379 | 72 | 7 |  |  |  |

Another collision occurs when we try to insert the next item, 34, into the bucket 34 % 11 = 1, which is already occupied. Using linear probing, we would then increment until we reach the available bucket at hashTable[8]. This would result in **7 more collisions** to find the next available bucket a position 8, for a total of **8 collisions** when inserting item 34. That brings our total collisions to **9.**

Inserting the final item, 42, would not result in a collision, as 42 % 11 = 9, and hashTable[9] is available.

**Total collisions = 9**

1. **Quadratic probing**

In quadratic probing, upon encountering a collision, we increment the hashed key + *n*^2, where *n* is the number of collisions or iterations in the prove, and modulate it with the length of the hash table.

The pseudo code for inserts looks just like the linear probing example above, except instead of iterating cyclically with linear probing, we use quadratic probing:

*// iterate cyclically using quadratic probing*

*hashedKey = (hashedKey + n^2) % length(hashTable)*

Using my example input, **[7, 421, 12, 72, 46, 379, 58, 34, 42]**, this would look like the following in a tabular representation.

We have no collisions when hashing and inserting 7, 421, 12, 72, 46, and 379 into our table:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | **3** | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|  | 12 | 46 | **421** |  | 379 | 72 | 7 |  |  |  |

However, we encounter a collision when hashing and inserting 58, because hash(58) = 57 % 11 = 3, and the bucket in hashTable[3] is occupied.

Using quadratic probing, then, we increment the hashedKey (3) plus the number of collisions or iterations in this cycle (1) squared, and modulate that by the length of our table (11).

*3 + 1^2 % 11 = 4 % 11 = 4*

Our bucket at hashTable[4] is empty, so we can go ahead and insert 58 into this position.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | **4** | 5 | 6 | 7 | 8 | 9 | 10 |
|  | 12 | 46 | 421 | **58** | 379 | 72 | 7 |  |  |  |

The next item we attempt to insert, 34, produces a collision, as 34 % 11 = 1, and hashedTable[1] is occupied. Using quadratic probing, we get:

// first iteration produces another collision

1 + 1^2 % 11 = 2 % 11 = 2

// second iteration produces another collision

2+ 2^2 % 11 = 6 % 11 = 6

// the third iteration prodiuces yet another collision

6 + 3^2 % 11 = 15 % 11 = 4

// the fourth iteration is successful

4 + 4^2 % 11 = 20 % 11 = 9

**Total collisions for inserting item 34 is 5**, which is a little better than linear probing in this circumstance, which produced 7. That brings our total collisions overall up to **6.**

The bucket at hashTable[9] is empty, so we can go ahead and insert 34 into hashTable[9]:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | **9** | 10 |
|  | 12 | 46 | 421 | 58 | 379 | 72 | 7 |  | **34** |  |

The final item we attempt to insert, 42, produces a collision (42 % 11 = 9), as the bucket in hashTable[9] is not available.

Using quadratic probing:

// first iteration is successful

9 + 1 ^ 2 % 11 = 10 % 11 = 10

The bucket at hashTable[10] is available, so we can insert our final item into this position. This involved only one collision.

**Total collisions = 7**

1. **“Double hashing”**

The “double hashing” collision resolution method involves incrementing the hashedKey with the result of a secondary hashing function, and modulating the sum of the two with the length of the hash table. The secondary hashing function takes the hashedKey and performs modular arithmetic on it using a “double factor”, which is the greatest prime number that is less than the total length of the hash table.

The pseudo code for inserts looks just like the linear probing example above, except that instead of probing linearly, we set the hashedKey to the result of our double hash:

*// secondary hash function*

*Procedure secondaryHash(iterations, hashedKey):*

*return iterations \* (doubleFactor - (hashedKey % doubleFactor))*

*// probe function*

*If collision:*

*return (hashedKey + secondaryHash(hashedKey, iterations)) % length(self.\_table)*

Using my example input, **[7, 421, 12, 72, 46, 379, 58, 34, 42]**, this would look like the following in a tabular representation.

We have no collisions when hashing and inserting 7, 421, 12, 72, 46, and 379 into our table:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | **3** | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|  | 12 | 46 | **421** |  | 379 | 72 | 7 |  |  |  |

However, we encounter a collision when hashing and inserting 58, because hash(58) = 57 % 11 = 3, and the bucket in hashTable[3] is occupied.

Using double hashing, then, we increment using the following formulae (11 is table length, and 7 is our “double factor”, or, the greatest prime number less than our table length. Code is in Python):

>>> def primaryHash(key):

... return key % 11  
  
>>> def secondaryHash(iterations, key):

... return iterations \* (7 - (key % 7))

>>> def doubleHash(k, iterations):

... primary = primaryHash(k)

... secondary = secondaryHash(iterations, k)

... return (primary + secondary) % 11

// first iteration produces collision  
doubleHash(3, 1) = 7

// second iteration is successful, **total collisions = 2**

doubleHash(7, 2) = 10

Our bucket at hashTable[10] is empty, so we can go ahead and insert 58 into this position.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | **10** |
|  | 12 | 46 | 421 |  | 379 | 72 | 7 |  |  | **58** |

The next item we attempt to insert, 34, produces a collision, as 34 % 11 = 1, and hashedTable[1] is occupied.

Using double hashing:

// first iteration produces another collision

doubleHash(1, 1) = 7

// second iteration produces another collision

doubleHash(7, 2) = 10

// the third iteration is successful

doubleHash(10, 3) = 0

**Total collisions for inserting item 34 is 3**, which is better than quadratic probing in this circumstance, which produced 5. That brings our total collisions overall up to **5.**

The bucket at hashTable[0] is empty, so we can go ahead and insert 34 into hashTable[0]:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **0** | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| **34** | 12 | 46 | 421 |  | 379 | 72 | 7 |  |  | 58 |

The final item we attempt to insert, 42, does not produce a collision (42 % 11 = 9), as the bucket in hashTable[9] is available.

**Total collisions = 5**

***Conclusion***

Using my example input and a rudimentary primary hashing function, the collision resolution methods of linear probing produced 9 collisions, quadratic probing produced 7, and double hashing produced 5 collisions. **Double hashing is the clear winner in terms of reducing collisions.**

There are some situations that may make linear probing, or quadratic probing, better collision reduction methods, however.

Double hashing prevents primary clustering which leads to fewer probing operations. However, when memory space is at a premium and/or we are not using dynamic arrays, double hashing probes may have to “bounce around” for a while until they find the next available spot, and may have worse run-time performance than linear probes.

Linear probe access is typically more performant than quadratic or double hashing probes, because the memory location / machine code address of the key’s reference in memory is likely to be closer to the probe’s previous location (this is called the “locality of reference”).

So, in situations where memory is at a premium and / or dynamically sized arrays are not an option, linear probing implementations win over quadratic or double hashing.