QUADRATIC PROBING

A hash table with ***quadratic probing*** handles a collision by starting at the key's mapped bucket, and then quadratically searches subsequent buckets until an empty bucket is found. If an item's mapped bucket is H, the formula (H+c1∗i+c2∗i2)mod(tablesize) is used to determine the item's index in the hash table. c1 and c2 are programmer-defined constants for quadratic probing. Inserting a key uses the formula, starting with i = 0, to repeatedly search the hash table until an empty bucket is found. Each time an empty bucket is not found, i is incremented by 1. Iterating through sequential i values to obtain the desired table index is called the ***probing sequence***.

HashInsert with quadratic probing.

HashInsert(hashTable, item) {

i = 0

bucketsProbed = 0

// Hash function determines initial bucket

bucket = Hash(item⇢key) % N

while (bucketsProbed < N) {

// Insert item in next empty bucket

if (hashTable[bucket] is Empty) {

hashTable[bucket] = item

return true

}

// Increment i and recompute bucket index

// c1 and c2 are programmer-defined constants for quadratic probing

i = i + 1

bucket = (Hash(item⇢key) + c1 \* i + c2 \* i \* i) % N

// Increment number of buckets probed

bucketsProbed = bucketsProbed + 1

}

return false

}

The search algorithm uses the probing sequence until the key being searched for is found or an empty-since-start bucket is found. The removal algorithm searches for the key to remove and, if found, marks the bucket as empty-after-removal.

HashRemove(hashTable, key) {

i = 0

bucketsProbed = 0

// Hash function determines initial bucket

bucket = Hash(key) % N

while ((hashTable[bucket] is not EmptySinceStart) and (bucketsProbed < N)) {

if ((hashTable[bucket] is Occupied) and (hashTable[bucket]⇢key == key)) {

hashTable[bucket] = EmptyAfterRemoval

return true

}

// Increment i and recompute bucket index

// c1 and c2 are programmer-defined constants for quadratic probing

i = i + 1

bucket = (Hash(key) + c1 \* i + c2 \* i \* i) % N

// Increment number of buckets probed

bucketsProbed = bucketsProbed + 1

}

return false // key not found

}

HashSearch(hashTable, key) {

i = 0

bucketsProbed = 0

// Hash function determines initial bucket

bucket = Hash(key) % N

while ((hashTable[bucket] is not EmptySinceStart) and (bucketsProbed < N)) {

if ((hashTable[bucket] is Occupied) and (hashTable[bucket]⇢key == key)) {

return hashTable[bucket]

}

// Increment i and recompute bucket index

// c1 and c2 are programmer-defined constants for quadratic probing

i = i + 1

bucket = (Hash(key) + c1 \* i + c2 \* i \* i) % N

// Increment number of buckets probed

bucketsProbed = bucketsProbed + 1

}

return null // key not found

}

DOUBLE HASHING

***Double hashing*** is an open-addressing collision resolution technique that uses 2 different hash functions to compute bucket indices. Using hash functions h1 and h2, a key's index in the table is computed with the formula (h1(key)+i∗h2(key))mod(tablesize). Inserting a key uses the formula, starting with i = 0, to repeatedly search hash table buckets until an empty bucket is found. Each time an empty bucket is not found, i is incremented by 1. Iterating through sequential i values to obtain the desired table index is called the ***probing sequence***.

Using double hashing, a hash table search algorithm probes (or checks) each bucket using the probing sequence defined by the two hash functions. The search continues until either the matching item is found (returning the item), an empty-since-start bucket is found (returning null), or all buckets are probed without a match (returning null).

A hash table insert algorithm probes each bucket using the probing sequence, and inserts the item in the next empty bucket (the empty kind doesn't matter).

A hash table removal algorithm first searches for the item's key. If the item is found, the item is removed, and the bucket is marked empty-after-removal.

HASH TABLE RESIZING

A hash table ***resize*** operation increases the number of buckets, while preserving all existing items. A hash table with N buckets is commonly resized to the next prime number ≥ N \* 2. A new array is allocated, and all items from the old array are re-inserted into the new array, making the resize operation's time complexity O(N).

Hash table resize operation.

HashResize(hashTable, currentSize) {

newSize = nextPrime(currentSize \* 2)

newArray = Allocate new array of size newSize

Set all entries in newArray to EmptySinceStart

bucket = 0

while (bucket < currentSize) {

if (hashTable[bucket] is not Empty) {

key = hashTable[bucket]

HashInsert(newArray, key)

}

bucket = bucket + 1

}

return newArray

}

Hash(key, tableSize) {

return key % tableSize

}

A hash table's ***load factor*** is the number of items in the hash table divided by the number of buckets. Ex: A hash table with 18 items and 31 buckets has a load factor of 18/31=0.58. The load factor may be used to decide when to resize the hash table.

An implementation may choose to resize the hash table when one or more of the following values exceeds a certain threshold:

* Load factor
* When using open-addressing, the number of collisions during an insertion
* When using chaining, the size of a bucket's linked-list

COMMON HASH FUNCTIONS

**A good hash function minimizes collisions**

A hash table is fast if the hash function minimizes collisions.

A ***perfect hash function*** maps items to buckets with no collisions. A perfect hash function can be created if the number of items and all possible item keys are known beforehand. The runtime for insert, search, and remove is O(1) with a perfect hash function.

A good hash function should uniformly distribute items into buckets. With chaining, a good hash function results in short bucket lists and thus fast inserts, searches, and removes. With linear probing, a good hash function will avoid hashing multiple items to consecutive buckets and thus minimize the average linear probing length to achieve fast inserts, searches, and removes. On average, a good hash function will achieve O(1) inserts, searches, and removes, but in the worst-case may require O(N).

A hash function's performance depends on the hash table size and knowledge of the expected keys. Ex: The hash function key % 10 will perform poorly if the expected keys are all multiples of 10, because inserting 10, 20, 30, ..., 90, and 100 will all collide at bucket 0.

**Modulo hash function**

A ***modulo hash*** uses the remainder from division of the key by hash table size N.

Modulo hash function.

HashRemainder(int key) {

return key % N

}

A ***mid-square hash*** squares the key, extracts R digits from the result's middle, and returns the remainder of the middle digits divided by hash table size N. Ex: For a hash table with 100 entries and a key of 453, the decimal (base 10) mid-square hash function computes 453 \* 453 = 205209, and returns the middle two digits 52. For N buckets, R must be greater than or equal to ⌈log10N⌉ to index all buckets. The process of squaring and extracting middle digits reduces the likelihood of keys mapping to just a few buckets.

The mid-square hash function is typically implemented using binary (base 2), and not decimal, because a binary implementation is faster. A decimal implementation requires converting the square of the key to a string, extracting a substring for the middle digits, and converting that substring to an integer. A binary implementation only requires a few shift and bitwise AND operations.

A binary mid-square hash function extracts the middle R bits, and returns the remainder of the middle bits divided by hash table size N, where R is greater than or equal to ⌈log2N⌉. Ex: For a hash table size of 200, R = 8, then 8 bits are needed for indices 0 to 199.

Mid-square hash function (base 2).

HashMidSquare(int key) {

squaredKey = key \* key

lowBitsToRemove = (32 - R) / 2

extractedBits = squaredKey >> lowBitsToRemove

extractedBits = extractedBits & (0xFFFFFFFF >> (32 - R))

return extractedBits % N

}

A ***multiplicative string hash*** repeatedly multiplies the hash value and adds the ASCII (or Unicode) value of each character in the string. A multiplicative hash function for strings starts with a large initial value. For each character, the hash function multiplies the current hash value by a multiplier (often prime) and adds the character's value. Finally, the function returns the remainder of the sum divided by the hash table size N.

DIRECT HASHING

A ***direct hash function*** uses the item's key as the bucket index. Ex: If the key is 937, the index is 937. A hash table with a direct hash function is called a ***direct access table***. Given a key, a direct access table ***search*** algorithm returns the item at index key if the bucket is not empty, and returns null (indicating item not found) if empty.

Direct Hash Function

HashSearch(hashTable, key) {

if (hashTable[key] is not Empty) {

return hashTable[key]

}

else {

return null

}

}

Direct Hashing: Insert, Remove, and search operations use items key as bucket index

HashInsert(hashTable, item) {

hashTable[item.key] = item

}

HashRemove(hashTable, item) {

hashTable[item.key] = Empty

}

HashSearch(hashTable, key) {

if (hashTable[key] is not Empty) {

return hashTable[key]

}

else {

return null

}

}

A direct access table has the advantage of no collisions: Each key is unique (by definition of a key), and each gets a unique bucket, so no collisions can occur. However, a direct access table has two main limitations.

1. All keys must be non-negative integers, but for some applications keys may be negative.
2. The hash table's size equals the largest key value plus 1, which may be very large.

HASHING ALGORITHMS: CRYPTOGRAPHY, PASSWORD HASHING

***Cryptography*** is a field of study focused on transmitting data securely. Secure data transmission commonly starts with ***encryption***: alteration of data to hide the original meaning. The counterpart to encryption is ***decryption***: reconstruction of original data from encrypted data.

A hash function can be used to produce a hash value for data in contexts other than inserting the data into a hash table. Such a function is commonly used for the purpose of verifying data integrity. Ex: A hashing algorithm called MD5 produces a 128-bit hash value for any input data. The hash value cannot be used to reconstruct the original data, but can be used to help verify that data isn't corrupt and hasn't been altered.

### Cryptographic hashing

A ***cryptographic hash function*** is a hash function designed specifically for cryptography. Such a function is commonly used for encrypting and decrypting data.

A ***password hashing function*** is a cryptographic hashing function that produces a hash value for a password. Databases for online services commonly store a user's password hash as opposed to the actual password. When the user attempts a login, the supplied password is hashed, and the hash is compared against the database's hash value. Because the passwords are not stored, if a database with password hashes is breached, attackers may still have a difficult time determining a user's password.