**Hash Tables**

The goal of hash table is to be able to access an entry based on its key value, not its location

We want to be able to access an entry directly through its key value, rather than by having to determine its location first by searching for the key value in an array

Using a hash table enables us to retrieve an entry in constant time (on average, O(1))

Hash Codes and Index Calculation

The basis of hashing is to transform the item’s key value into an integer value (its hash code) which is then transformed into a table index

Diagram

Description automatically generated

A picture containing text, sky

Description automatically generatedConsider the Huffman code problem.

If a text contains only ASCII values, which are the first 128 Unicode values we could use a table of size 128 and let its Unicode value be its location in the table. Because Unicode is too big, number of symbols are around 65000. We don’t need all of them. We need, for example, 100 of them. You can put only 100 symbols in a table and access them faster.

However, what if all 65536 Unicode characters were allowed?

If you assume that on average 100 characters were used, you could use a table of 200 characters and compute the index by :

int index = unicode % 200

If a text contains this snippet:

* . . . mañana (tomorrow), I’ll finish my program . . .

Given the following Unicode values:

A picture containing table

Description automatically generated

The indices for letters ‘ñ’ and ‘)’ are both 41:

* 41 % 200 = 41 and 241 % 200 = 41

This is called a collision; we will discuss how to deal with collisions shortly, but it cannot be completely avoided. Collision possibility can be reduced.

Methods for Generating Hash Codes

In most applications, a key will consist of strings of letters or digits (such as a social security number, an email address, or a partial ID) rather than a single character

* If key is just integer, “int % tableSize” would be good hash function
  + Performing mod operations based on prime number is better for the hash code
    - … % 7 is better than … % 10

The number of possible key values is much larger than the table size

Generating good hash codes typically is an experimental process

* Hash methods have to distribute the keys to indices in a uniformly random fashion

The goal is a random distribution of values

Simple algorithms sometimes generate lots of collisions

Java HashCode Method

For strings, summing the int values of all characters returns the same hash code for “sign” and “sing”

The Java API algorithm accounts for position of the characters as well

String.hashCode() returns the integer calculated by the formula:

Text

Description automatically generated

“Cat”’s hash code 🡪 ‘C’ x + ‘a’ x 31 + ‘t’ = 67510

After converting to integer, we have to perform mod operation based on the table size to get the index

31 is a prime number, and prime numbers generate relatively few collisions

Because there are too many possible strings, integer value returned by String.hashCode cant be unique

However, because the String.hashCode method distributes the hash code values fairly evenly throughout the range, the probability of two strings having the same hash code is low

The probability of a collision with:

s.hashCode() % table.length

is proportional to how full the table is. If table is very full, collision probability is larger; if table is more empty, collision probability is smaller.

If the collision probability gets larger, you have to use larger table.

A good hash function should be relatively simple and efficient to compute

It doesn’t make sense to use an O(n) hash function to avoid doing an O(n) search

We now consider 2 ways to organize hash tables (handle collision):

* open addressing
* chaining

Open Addressing

In open addressing, linear probing (increment the index until find an empty place) can be used to access an item in a hash table

* If the index calculated for an item’s key is occupied by an item with that key, we have found the item
* If that element contains an item with a different key, increment the index by one
* Keep incrementing until you find the key or a null entry (assuming the table is not full)

Graphical user interface, text, application

Description automatically generated

**----------> key is not available in the table**

Table Wraparound and Search Termination

As you increment the table index, your table should wrap around as in a circular array

This enables you to search the part of the table before the hash code value in addition to the part of the table after the hash code value

But it could lead an infinite loop

How do you know when to stop searching if the table is full and you have not found the correct value?

* Stop when the index value for the next probe is the same as the hash code value for the object
* Ensure that the table is never full by increasing its size after an insertion when its load factor exceeds a specified threshold

Hash Code Insertion Example:

A picture containing table

Description automatically generated

Table

Description automatically generatedSince 4 is empty, we are sure that “Tom” is not inserted before. We insert “Tom” at 4.

We check 4 for “Dick”. It is not empty and value is not “Dick” (it is “Tom”). We cannot make sure “Dick” is not available in the table or we cannot insert it at 4. We increment the index value by 1 (wraparound) and index becomes 0. Since index 0 is empty, we are sure that “Dick” is not in the table. 0 is empty so we insert “Dick” at 0.

We insert “Harry” at 3.

We insert “Sam” at 1.

We insert “Pete” at 2.

In the case of this hash table if we have to search for “Dick”, its hashCode is 4 so we get index 4. It is not “Dick”, but we have to keep searching for next position until an empty space is found. In this case, we found “Dick” at index 0.

This is called open addressing and linear probing. We probe the next position to find the empty space during insertion and to continue search for the value if it is not found yet.

A value cannot be inserted twice, it implements the set. During insertion, we check whether it is available or not, if it is available we don’t insert it again.

While inserting “Pete”, we traverse all the array. So worst case depends on the table size. But in hashing, we would like to keep the collision probability low while increasing the table size when we hit some threshold. So this kind of position (having the table almost full) will not happen in hashing. So having linear time is very rare. So average case will be constant time.

Retrieval of “Tom” or “Harry” takes one step, O(1).

Because of collisions, retrieval of the others requires a linear search.

We prevent infinity by checking if number of probes exceeds the number of elements. If so, we have to stop.

Line chart

Description automatically generated with medium confidence

by hash code

Number of possible keys for 11 digit student ID is but we need for one university. Size of the table is of scale, number of possible keys is scale.

So when we map all IDs into the table of , a collision cannot be avoided completely. Choosing a good hashcode and choosing the table size as big enough (not as big as but little bit bigger than students like ); we can decrease the probability of collision.

Good hashcode distribute possible keys into indexes almost randomly.

Load factor : percentage of the table that is used already. Load factor should be reduced to reduce the risk of the collision.

If the load factor gets too large, we increment the table size and copy all the elements in a proper way.

We can determine the load factor for our threshold. If we reach the threshold, inserting a new element is not possible. We usually increase the table size (x2) in that case, and copy all the elements. We cannot directly copy the elements because our table size got bigger, so we have to take mod differently to copy elements to correct indices.

Table

Description automatically generated

We multiply table size by 2. But we prefer prime number. So we choose prime number that is bigger than tableSize \* 2. It is 11.

Table

Description automatically generatedAll the elements (“Tom”, “Dick”, “Harry”, “Sam”, “Pete”) are searched and found in constant time.

For “Sam”, we performed extra probe but number of probes is quite low bc load factor is 5/11, it is less than 0.5, it is a lot smaller this makes running time close to constant time.

----------------------------------------------------------------------------------------

If we want to delete “Dick”, we can’t just delete it and make 5th index null bc if we search for “Sam”, we come to the 5th index and we say “Sam” doesn’t exist since its index is empty. Instead we make 5th entry deleted. Since it is deleted, when we search “Sam”, when we come to the deleted entry, we continue to search next item.

*Deleted positions are considered as occupied positions. They have te be considered when load factor is calculated.*

The best way to reduce the possibility of collision (and reduce linear search retrieval time because of collisions) is to increase the table size.

Traversing a Hash Table

You cannot traverse a hash table in a meaningful way since the sequence of stored values is arbitrary

Table

Description automatically generated with medium confidence

Deleting an Item Using Open Addressing

When an item is deleted, you cannot simply set its table entry to null

If we search for an item that may have collided with the deleted item, we may conclude incorrectly that it is not in the table

Instead, store a dummy value or mark the location as available, but previously occupied

Deleted items reduce search efficiency which is partially mitigated if they are marked as available

You cannot simply replace a deleted item with a new item until you verify that the new item is not in the table

Table

Description automatically generated

For example, if we delete “Harry” and try to add “Pete”, “Pete”’s index was 3 but we cannot simply add “Pete” to 3 because if we do it, than “Pete” is duplicated. So we have to search the array first.



So deleted items make insertion inefficient as well.

Table

Description automatically generatedIf you want to add “Ali” that belongs to index 3 after removing “Tom”, first you look to 3rd index. Since 3rd index is deleted, you continue to next index. Since next index is null, we are sure that “Ali” is not in the list and add “Ali” to index 3, its original position. Then searching “Ali” will be constant time. But if 4th index is deleted instead of null, then we have to look through index 8 which is null to make sure that “Ali” is not in the list.



Reducing Collisions by Expanding the Table Size

Use a prime number for the size of the table to reduce collisions

A fuller table results in more collisions, so, when a hash table becomes sufficiently full (load factor is above the threshold), a larger table should be allocated and the entries reinserted

You must reinsert (rehash) values into the new table; do not copy values as some search chains which were wrapped may break

Deleted items are not reinserted to new table, which saves space and reduces the length of some search chains

Reducing Collisions Using Quadratic Probing

Linear probing tends to form clusters of keys in the hash table, causing longer search chains

Quadratic probing can reduce the effect of clustering

* Increments form a quadratic series (1, , , …)

probeNum++;

index = (startIndex + probeNum \* probeNum) % table.length

//Computation time of index makes quadratic probing a bit slower

If an item has a hash code of 5, successive values of index will be 6 (5+1), 9 (5+4), 14 (5+9), …

Table

Description automatically generatedAs long as accumulation increases in the red area, chance of having a collision over that area increases.

If collision happens, size of this accumulation increases as well.

So the cluster increases rapidly in linear probing. This kind of cluster makes the more collisions and performance decreases because of these clusters.

Linear probing has primary clustering problem.

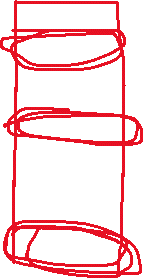
Table

Description automatically generatedIn quadratic probing, when a collision happens, we try to increment the original hash value by 1 first, if it is occupied we try to increment by , if it is occupied we try to increment by , …

We perform less probes in quadratic probing.

Quadratic probing solves primary clustering problem but we still have secondary clustering problem but it is less serious.

secondary clustering problem ----------------------------->



Problems with Quadratic Probing

The disadvantage of quadratic probing is that the next index calculation is time-consuming, involving multiplication, addition, and modulo division

A more efficient way to calculate the next index is (k starts with 1):

k += 2;  
 index = (index + k) % table.length;

Examples:

5+1, 5+, 5+

If the initial value of k is -1, successive values of k will be 1, 3, 5, …

If the initial value of index is 5, successive value of index will be 6 (5+1), 9 (5+1+3), 14 (5+1+3+5)

The proof of the equality of these two calculation methods is based on the mathematical series:



A more serious problem is that not all table elements are examined when looking for an insertion index; this may mean that:

* an item can’t be inserted even when the table is not full
* the program will get stuck in an infinite loop searching for an empty slot

If the table size is a prime number and it is never more than half full, this won’t happen

However, requiring a half empty table wastes a lot of memory

**Chaining**

Chaining is an alternative to open addressing

Each table element references a linked list that contains all of the items that hash to same table index

* The linked list often is called a bucket
* The approach sometimes is called bucket hashing

Diagram

Description automatically generated

Advantages relative to open addressing:

* Only items that have the same value for their hash codes are examined when looking for an object
* You can store more elements in the table than the number of table slots (indices). Load factor can be more than 1 in chaining
* Once you determine an item is not present, you can insert it at the beginning or end of the list
* To remove an item, you simply delete it; you do not need to replace it with a dummy item or mark it as deleted

Performance of Hash Tables

Load factor has the greatest effect on hash table performance

The lower the load factor, the better the performance as there is a smaller chance of collision when a table is sparsely (seyrek olarak) populated

If there are no (or small number of) collisions, performance for search, retrieval, insertion, deletion is O(1) regardless of table size

Performance of Open Addressing vs. Chaining

A picture containing text, clock

Description automatically generatedDonald E. Knuth derived the following formula for the expected number of comparisons (number of probes), c, required for finding an item that is in a hash table using open addressing with linear probing and a load factor L

L (Load Factor) : probability of being occupied

1 - L : probability of being not occupied

1 / (1-L) : number of probes that should be done until an unoccupied (null) cell is found

1 + : for found case we have to find it

1/2 : 1 + 1/(1-L) is max number of probes that is required, we assumed we found our key in the half way

If search is unsuccessful, number of probes (c) =

A picture containing diagram

Description automatically generatedUsing chaining, if an item is in the table, on average we must examine the table element corresponding to the item’s hash code and then half of the items in each list

L (Load Factor) : average number of items in a linked list (the number of items divided by the table size)

c : number of comparisons

Table

Description automatically generatedIf search is unsuccessful, number of probes (c) = L

Performance of Hash Tables versus Sorted Array and Binary Search Tree

The number of comparisons required for a binary search of a sorted array is O(logn)

* A sorted array of size 128 requires up to 7 probes ( is 128) which is more than for a hash table of any size that is 90% full (5.50 probes needed with linear probing - check the table above)
* A binary search tree performs similar to sorted array

Insertion or removal:

Table

Description automatically generated

Storage Requirements for Hash Tables, Sorted Arrays, and Trees

The performance of hashing is superior to that of binary search of an array or a binary search tree (BST), particularly if the load factor is less than 0.75

However, the lower the load factor, the more empty storage cells

* There are no empty cells in a sorted array
* If you use ArrayList, almost half of it is empty

A BST requires 3 references per node (item, left subtree, right subtree), so more storage is required for a binary search tree than for a hash table with load factor 0.75

for n elements, we need 3n storage for bst,  
for n elements, we need (4/3)n storage for hash table with load factor 0.75

Storage Requirements for Open Addressing and Chaining

For open addressing, the number of references to items (key-value pairs) is n (the size of the table)

For chaining, the average number of nodes in a list is L (the load factor) and n is the number of table elements

* Using the Java API LinkedList, there will be 3 references in each node (item, next, previous)
* Using our own single linked list, we can reduce the references of 2 by eliminating the previous element reference
* Therefore, storage for n+2L\*n references is needed

Example:

Assume open addressing, 60000 items in the hash table, and a load factor of 0.75

This requires a table of size 80000 (80000 = (4/3) \* 60000) and results in an expected number of comparisons of 2.5

Calculating the table size n to get similar performance using chaining:

* 2.5 = 1 + L/2
* 5.0 = 2 + L
* 3.0 = 60000/n
* n = 20000

As far as the run-time performance is concerned, open addressing table with linear probing with 0.75 load factor and 60000 items is equivalent to chaining table with load factor of 3 and 20000 table size

Memory size for open addressing 🡪 80000  
Memory size for chaining 🡪 20000 + 2\*3\*20000 = 140000

A hash table of size 20000 provides storage space for 20000 references to lists

There are 60000 nodes in the table (one for each item)

This requires storage for 140000 references (2\*60000 + 20000), which is 175% of the storage needed for open addressing

For other cases, storage need may be better for chaining than open addressing