Hashing and Collision

Hash Tables

Hash table is a data structure in which keys are mapped to array positions by a has function. A value stored in a hash table can be searched in $O(1)$ time by using a hash function which generates an address from the key.

Wheb the set K of keys that are actually used is smaller than the universe of keys (U), a hash table consumes less storage space. The storage requirement for a hash table is $O(k)$, where k is the number of keys actually used.

In a hash table, an element with key k is stored at index $h(k)$ and not k. It means a hash function h is used to calculate the index at which the element key k will be stored. The process of mapping the keys to appropriate locations (or indices) in a hash table is called **hashing**.

A **collision**, is when two or more keys map to the same memory location. The main goal of using a hash function is to reduce the range of array indices that habe to be handled. Thus, instead of having U values, we just need K values, thereby reuding the amount of storage space required.

Hash Functions

A hash function is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table. The main goal of a hash function is that elements should be relatively, random, and uniformly distributed. It produced a unique set of integers within some suitable range in order to reduce the number of collisions. In practive, there is no has function that eliminates collisions completely. A good hash function can only minimize the number of collisions by spreading the elements uniformly throught the array.

Properties of a Good Hash Function

Low cost: The cost of executing a hash function must be small, so that using the hashing technique becomes preferable over other approaches.

Determinism: A hash procedure must be deterministic. This means that the same hash value must be generated for a given input value. This criteria excludes hash functions that depend on external variable parameters and on the memory address of the object being hashed.

Uniformity: A good hash function must map the keys as evenly as possible over its output range. This means that the probability of generating every hash value in the out range should roughly be the same. The property of uniformity also minimizes the number of collisions.

Different Hash Functions

Division Method

It is the most simple method if hashinan integer x. Thus divides x by M and then uses the remainder obtained. In this case the hash function can be given as

$$h(x) = x \texttt{ mod } 2^k$$

then the function will simply extract the lowest k bits of the binary representation of x.

A potential drawback of the division method is that while using the method, consecutive keys map to consecutive hash values. This ensures that consecutive keys do not collide, but it also means that consecutive array locations will be occupied. This may lead to degradation in performance.

Multiplication Method

The steps involved in the multiplication method are as follows:

Step 1: Choose a constant A such that $0 < A < 1$.

Step 2: Multiply the key k by A.

Step 3: Extract the fractional part of kA.

Step 4: Multiply the result of Steo 3 by the size of hash table (m).

hence, the hash function can be given as:

$$h(k) - \lfloor m(kA \texttt{ mod } 1) \rfloor$$

where (kA `mod` 1) gives the fractional part of kA and m is the total number of indicies in the hash table.

The greatest advantage of this method is that it works practaclly with any value of A. Although the algorithm works bettwe with some values, the optimal choice depends on the characteristics of the data being hashed. Knuth has suggested that the best choice of A is

$$\frac{(\sqrt{5} - 1)}{2} \approx 0.6180339887$$

Mid-Square Method

The mid-square method is a good hash function which works in two steps:

Step 1: Square the value of the key. That is, find k^2.

Step 2: Extract the middle r digits of the result obtained in Step 1.

The algorithm works well because most or all of the key value contribute to the result. This is because all the digits in the original key value contribute to produce the middle digits of the squared value. Therefore, the result is not dominated by the distribution of th bottom digit or the top digit of the original value.

In the mid-square method, the same r digits must be chosen from all the keys. Therefore, the hash function can be given as:

$$h(k) = s$$

where s is obtained by selecing r digits from k^2.

Folding Method

The folding method works in the following two steps:

Step 1: Divide the key value into a number of parts. That is, divide k into parts $k_1, k_2, \ldots, k_n$, where each part has the same number of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. That is, obtain the sum of $k_1 + k_2 + \cdots + k_n$. The hash value is produced by ignoring the last carry, if any.

Note that the number of digits in each part of the key will vary depending upon the size of the hash table.

Collisions

Collisions occur when the hash function maps two different keys to the same location. A method used to solve the problem of collision, also called **collision resolution technique**, is applied. The two most popular methods of resolving collisions are:

1. Open addressing
2. Chaining

Collision Resolution by Open Addressing

Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position. In this technique, all the values are stored in the hash table. The has table contains two types of values: **sentinel values** and **data values**. The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.

When a key is mapped to a particular memory location, then the value it holds is checked. If it contains a sentinel value, then the location is free and the data value can be stored in it. However, if the location already has some data value stored in it, then other slots are examined systematically in the forward direction to find a free slot. If even a single free location is not found, then we have an `OVERFLOW` condition.

The process of examining memory locations in the hash table is called **probing**. Open addressing technique can be implemented using linear probing, quadratic probing, double hashing, and rehashing.

Linear Probing

The simplest approach to resolve a collision is linear probing. In this technique, if a value is already stored at a location generated by $h(k)$, then the following function is used to resolve the collision:

$$h(k, i) = [h'(k) + i] \texttt{ mod } m$$

where m is the size of the hash table, $h'(k) = (k \texttt{ mod } m)$ and i is the probe number that varies from 0 to $m - 1$.

Therefore, for a given key k, first the location generated by $[h'(k) \texttt{ mod } m]$ is probed because for the first time $i = 0$. If the location is free, the value is stored in it, else the second probe generates the address of the location given by $[h'(k) + 1] \texttt{ mod } m$. If the location is occupied then subsequent probes generate the address as $[h'(k) + 2] \texttt{ mod } m$, $[h'(k) + 3] \texttt{ mod } m$, $[h'(k) + 4] \texttt{ mod } m$,$[h'(k) + 5] \texttt{ mod } m$, and so on until a free location is found.

Searching Values using Linear Probing

The procedure for searching a value in a hash table is the same as for storing a value in a hash table. While searching for a value in a hash table, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If a match is found, then the search operation is successful. The search time in this case is given as $O(1)$. If the key does not match, then the search function begins a sequential search of the array that continues until:

- the value is found, or
- the search function encounters a vacant location in the array, indicating that the value is not present, or
- the search function terminates because it reaches the end of the table and the value is not present.

In the worst case, the search operation may have to make $n - 1$ comparisons, and the running time of the search algorithm take $O(n)$ time. The worst case will be encountered when after scanning all the $n - 1$ elements, the value is either present at the last location or not present in the table.

Thus, we see that with the increase in the number of collisions, the distance between the array index computed by the hash function and the actual location of the element increases, thereby increasing the search time.

Pros and Cons

Linear probing finds an empty location by doing a linear search in the array beginning from poisition $h(k)$. Although the algorithm provides good memory caching through locality of reference, the drawback of this algorithm is that it results in clustering, and thus there is a higher risk of more collisions where one collision has already taken place. aThe performance of linear probing is sensitive to the distribution of input values.

As the hash table fills, cluster of consecutive cells are formed and the time required for a search increases with the size of the cluster. In addition to this, when a new value has to be inserted into the table at a position which is already occupied, that value is inserted at the end of the cluster, which again increases the length of the cluster. Generally, an insertion is made between two clusters that are separated by one vacant space. With linear probing, there are more chances that subsequent insertions will also end up in one of the clusters, thereby potentially increasing the cluster length by an amount much greater than one. The more number of collisions, higher ther probes that are required to find a free location and the performance is degraded. This is called **primary clustering**. To aviod primary clustering, other techniques such as quadratic probing and double hashing are used.

Quadratic Probing

In this technique, if a value is already stored at a location generated by $h(k)$, then the following hash function is used to resolve the collision:

$$h(k,i) = [h'(k) + c_1 i + c_2 i^2] \texttt{ mod } m$$

where m is the size of the hash table, $h'(k) = (k \texttt{ mod } m)$, i is the probe number that varies from 0 to $m-1$, and c_1 and c_2 are constants such that c_1 and $c_2 \neq 0$.

Quadratic probing eliminates the primary clustering phenomenon of linear probing because instead of doing a linear search, it does a quadratic search. For a given key k, first the location generated by $h'(k) \texttt{ mod } m$ is probed. If the location is free, the value is stored in it, else subsequent locations probed are offset by factors that depend in a quadratic manner on the probe number i. Although quadratic probing performs better than linear probing, in order to maximize the utilization of the hash table, the values of c_1, c_2 and m need to be constrained.

Searching a Value using Quadratic Probing

While searching a value using the quadratic probing technique, the array index is re-computed and the key of the element stored at that location is compared with the value that has to be searched. If the desired key value matches with the key value at that location, then the element is present in the hash table and the search is said to be successful. In this case, the search time is given as $O(1)$. However, if the value does not match, then the search function begins a sequential search of the array that continues until:

- the value is found, or
- the search function encounters a vacant location in the array, indicating that the value is not present, or
- the search function terminates because it reaches the end of the table and the value is not present.

In the worst case, the search operation may take $n-1$ comparisons, and the running time of the search algorithm $O(n)$. The worst case will be encountered when after scanning all the $n-1$ elements, the value is either present at the last location or not present in the table.

Pros and Cons

Quadratic probing resolves the primary clustering problem that exists in the linear probing technique. Quadratic probing good memory caching because it preserves locality of reference. But linear probing does this task better and gives a better cache performance.

One of the major drawbacks probing is that that a sequence of successive probes may only explore a fracion of the table, and this fraction may be quite small. If this happens, then we will not be able to find an empty location in the table despite the fact that the table is by no means full.

Although quadratic probing is free from primary clustering, it is still liable to what is known as **secondary clustering**. It means that if there is a collision between two keys, then the same probe sequence will be followed for both. With quadratic probing, the probability for multiple collisions increases as the table becomes full. This situation is usually encountered when the hash table is more than full.

Quadratic probing is widely applied in the Berkeley Fast File System to allocate free blocks.

Double Hashing

To start with, double hashing uses one hash value and then repeatedly steps forward an interval until an empty location is reached. The interval is decided using a second, independent hash function. In double hashing, we use two hash functions rather than single function. The hash function in the case of double hashing can be given as:

$$h(k,i) = [h_1(k) + i h_2(k)] \texttt{ mod } m$$

where m is the size of the hash table, $h_1(k)$ and $h_2(k)$ are two hash functions given as $h_1(k) = k$ `mod` , $h_2(k) = k$ `mod` m', i is the probe number that varies from 0 to $m-1$, and m' is chosen to be less than m. We can choose $m' - m - 1$ or $m - 2$.

When we have to insert a key k in the hash table, we first probe the location given by applyinh $[h_1(k)$ `mod`$m]$ because during the first probe, $i = 0$. If the location is vacant, the key is inserted into it, else subsequent probes generate locations that are at an offset of $[h_2(k)$ `mod` $m]$ from the previous location. Since the offset may vary with every probe depending on the value generated by the second hash function, the performance of double hashing is very close to the performance of the ideal scheme of uniform hashing.

Pros and Cons

Double hashing minimizes repeated collisions and the effects of clustering. That is, double hashing is free from problems associated with primary clustering as well as secondary clustering.

Rehashing

When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table.

All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table.

Though rehashing seems to be simple process, it is quite expensive and must therefore not be done frequently.

Collision Resolution by Chaining

In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. That is, location 1 in the hash table points to the head of the linked list of all the key values that hashed to 1. However, if no key value hashes to 1, then location 1 in the hash table contains `NULL`.

Operations on a Chained Hash Table

Searching for a value in a chained hash table is as simple as scanning a linked list for an entry with the given key. Insertion operation appends the key to the end of the linked list pointed by the hashed location. Deleting a key requires searching the list and removing the element.

Chained hash tables with linked lists are widely used to the simplicity of the algorithms to insert, delete, and search a key. The code for these algorithms is exactlythe same as that for inserting, deleting and searching a value in a single linked list.

While the cost of inserting a key in a chained hash table is $O(1)$, the cost of deleting and searching a value is given as $O(m)$ where m is the number of elements in the list of that location. Searching and deleting takes more time because these operations scan the entries of the selected location for the desired key.

In the worst case, searching a value may take a running time of $O(n)$, where n is the number of key values stored in the chained hash table. This case arises when all the key values are inserted into the linked list of the same location (of the hash table). In this case, the hash table is ineffective.

Codes to initialize, insert, delete, and search a value in a chained hash table

Strucutre of the node

```
typedef struct node_HT {
        int value;
        struct node *next;
} node;
```

Code to initialize a chained hash table

// Initializes m location in the chained hash table.

// The operation takes a running time of $O(m)$ */

```
void  initializeHashTable(node *hash_table[], int m) {
     int i;
     for(i = 0; i ≤ m; i++)
        hash_table[i] = NULL;
}
```

Code to insert a value

// The element is inserted at the beginning of the linked list whose pointer to its head is

// stored in the location given by $h(k)$. The running time of the insertion operation is $O(1)$,

// as the new key is alwaysadded as the first element of the list irrespective of the

// size of the linked list as well as that of the chained hash table. *

```
node  *insert_value(node *hash_table[], int val) {
     node *new_node;
     new_node = (node *)malloc(sizeof(node));
     new_node value = val;
new_node next = hash_table[h(x)];
hash_table[h(x)] = new_node;
}
```

Code to search a value

// The element is searched in the linked list whose pointer to its head is stored in the

// location given by $h(k)$. If search is successful, the function returns a pointer to the node

// in the linked list; otherwise it returns `NULL`. The wost case running time of the

// search operation is given as order of size of the linked list. */

```
node *search_value(node *hash_table[], int val) {
    node *ptr;
    ptr = hash_table[h(x)];
    while((ptr ≠ NULL) && (ptr -> value ≠ val))
         ptr = ptr -> next;
    if(ptr -> value == val)
         return ptr;
    else
         return NULL
}
```

Code to delete a value

// To delete a node from the linked list whose head is stored at the location given by $h(k)$

// in the hash table, we need to know the address of the node's predecessor. We do this

// using a pointer save. The running time complexity of the delete operation is same as that

// of the search operation because we need to search the predecessor of the node so that the

// node can be removed without affecting other nodes in the list. */

```
void  delete_value(node *hash_table[], int val) {
     node *save, *ptr;
     save = NULL;
     ptr = hash_table[h(x)];
     while((ptr ≠ NULL) && (ptr value ≠ val)) {
          save = ptr;
          ptr = ptr next;
     }
     if(ptr ≠ NULL) {
          save next = ptr next;
          free(ptr);
     }
     else
          printf("\n VALUE NOT FOUND"):
}
```

Pros and Cons

The main advantage of using a chained hash table is that is remains effective even when the number of key values to be stored is much higher than the number of locations in the hash table. However, with the increase in the number of keys to be stored, the performance of a chained hash table does degrade gradually (linearly).

The other advantage of using chaining for collision resolution is that its preformance, unlike quadratic probing, does not degrade when the table is more than half full. This technique is absolutely free from clustering problems and thus provides an efficient mechanism to handle collisions.

Chained hash tables inherit the disadvantages of linked list. First, to store a key value, the space overhead of the next pointer in each entry can be signigicant. Second, traversing a linked list has poor cache performance, making the processor cache ineffective.

Bucket Hashing

In closed hashing, all the records are directly stored in the hash table. Each record with a key value k is stored in a location called its home position. The home position is calculated by applying some hash functions.

In case the home position of the record with key k is already occupied by another record then the record will be stored in some other location in the hash table. This location will be determined by technique that is used for resolving collisions. Once the records are inserted, the same algorithm is again applied to search for a specified record.

One implementation of closed hashing groups the hash table into buckets where M slots of the hash table are divided into B buckets. Therefore, each bucket contains M/B. Now when a new record has to be inserted, the hash function computes the home position. If the slot is fre, the record is inserted. Otherwise, the bucket's slots are sequentially searched until an open slot is found. In case, the entire bucket is full, the record is inserted into an overflow bucket. The overflow bucket has infinite capacity at the end of the table and is shared by all the buckets.

An efficient implementation of bucket hashing will be to use a hash function that evenly distributes the records amongst the buckets so that very few records have to be inserted in the overflow bucket.

When searching a record, first the hash function is used to determine the bucket in which the record can be present. Then the bucket is sequentially searched to find the desired record. If the record is not found and the bucket still has some empty slots, then it means that the search is complete and the desired record is not present in the hash table.

If the bucket is full and the record has not been found, then the overflow bucket is searched until the record is found or all the records in the overflow bucket have been checked. Searching the overflow bucket can be expensive if it has too many records.

Pros and Cons of Hashing

One advantage of hashing is that no extra space is required to store the index as in the case of other data structures. In addition, a hash table provides fast data access and an added advantage of rapid updates.

The primary drawback of using the hashing technique for inserting and retrieving data values is that is usually lacks locality and sequntial retrieval by key. This makes insertion and retrivel of data values even more random.

Applications of Hashing

Hash tables are widey used in situations where enormous amounts of data have to be accessed to quickly search and retrieve information.

Hashing is used for database indexing.

• Hash table is a data structure in which keys are mapped to array positions by a hash function. A value stored in a hash table can be searched in $O(1)$ time using a hash function which generates an address from the key.

• The storage requirement for a hash table is $O(k)$, where k is the number of keys actually used. In a hash table, an element with key k is stored at index $h(k)$, not k. This means that a hash function h is used to calculate the index at which the element with key k will be stored. Thus, the process of mapping keys to appropriate locations (or indices) in a hash table is called hashing.

• Popular hash functions which use numeric keys are division method, multiplication method, mid square method, and folding method.

• Division method divides x by M and then uses the remainder obtained. A potential drawback of this method is that consecutive keys map to consecutive hash values.

• Multiplication method applies the hash function given as $h(x) = \lfloor m(kA \texttt{mod } 1) \rfloor$

• Mid square method works in two steps. First, it find k^2 and then extracts the middle r digits of the result.

• Folding method works by first dividing the key value k into parts $k_1, k_2, \ldots, k_n$, where each part has the same number of digits except the last part which may have lesser digits except the other parts, then obtaining the sum of $k_1 + k_2 + \ldots + k_n$. The hash value is produced by ignoring the last carry, if any.

• Collisions occur when a hash function maps two different keys to the same location. Therefore, a method used to solve the problem of collisions, also called collision resolution technique, is applied.The two most popular methods of resolving collisions are: (a) open addressing and (b) chaining.

• Once a collision takes place, open addressing computes new positions using a probe sequence anf the next record is stored in that position. In this technique of collision resolution, all the values are stored in the hash table. The hash table will contain two types of values—either sentinel value or a data value.

• Open addressing technique can be implemented using linear probing, quadratic probing, double hashing, and rehashing.

• In linear probing if a value is already stored at a location generated by $h(k)$, then the following hash function is used to resolve the collision:

$$h(k, i) = [h'(k) + i] \texttt{ mod } m$$

Though linear probing enables good memory caching, the drawback of this algorithm is that it results in primary clustering.

• In quadratic probing, if a value is already stored at a location generated by $h(k)$, if a value is already stored at a location generated by $h(k)$, then the following hash function is used to resolve the collision:

$$h(k, i) = [h'(k) + c_1 i + c_2 i^2] \texttt{ mod} m$$

Quadratic probing eliminates primary and provides good memory caching. But it is still liable to secondary clustering.

• In double hashing, we use two hash functions rather than a single function. The hash function rather than a single function. The hash function in the case of double hashing can be given as

$$h(k, i) = [h_1(j) + ih_2(k)] \bmod m$$

The performance of double hashing is very close to the performance of the ideal scheme of uniform hashing. It minimizes repeated collisions and the effects of clustering.

• When the hash table becomes nearly full, the number of collisions increases, thereby degrading the performance of insertion and search operations. So in rehashing, all the entries in the original hash table are moved to the new hash table which is double the size of the original hash table.

• In chaining, each location in a hash table stores a pointer to a linked list that contains all the key values that were hashed to that location. While the cost of inserting a key in a chained hash table is $O(1)$, the cost for deleting and searching a value is given as $O(m)$, where m is the number of elements in the list of that location. However, in the worst case, searching for a value may take a running time of $O(n)$.