Chapter 16 Hash Table

Contents

Syllabus Learning Outcomes

1.2 Fundamental Algorithms

1.2.3 Implement search algorithms.
- Linear search
- Binary search
- Hash table search

1.2.4 Use examples to explain search algorithms.

1.2.5 Compare and describe the efficiencies of the search algorithms using Big-O notation for time complexity (worst case).
Exclude space complexity

2.3 Implementing Algorithms and Data Structures

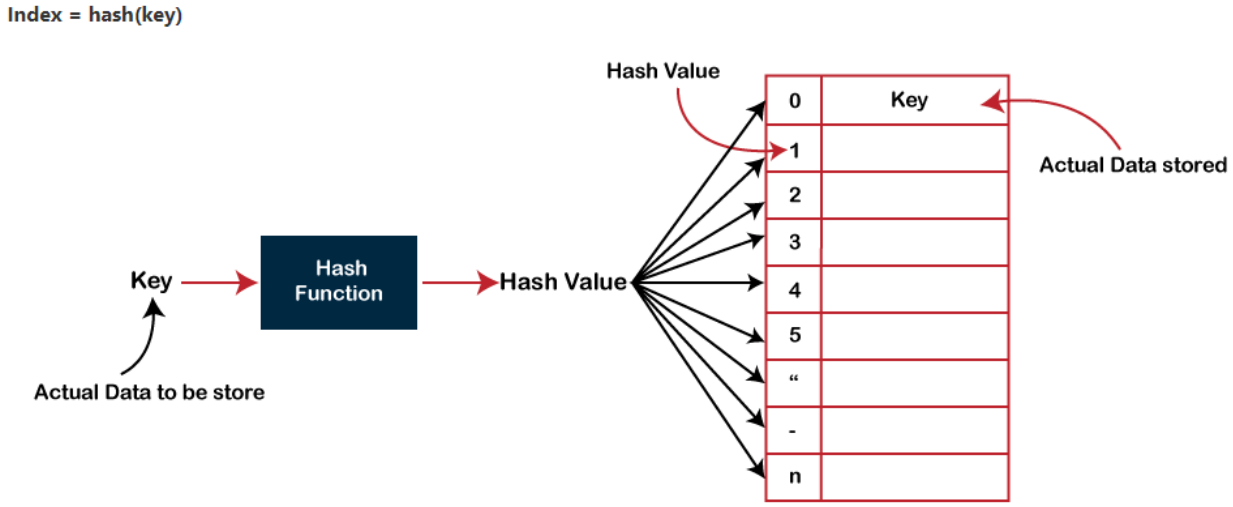
2.3.2 Implement search programs.
- Linear search
- Binary search
- Hash table search

1 Introduction to Hash table

A hash table is a data structure that store data in an array and have direct access to the records. An address (the array index) is calculated from the key value of the data and the data is then stored at this address in the array. To search for a data, the address is calculated from the key and we can look up the array using the calculated address to find the data. Calculating an address from a key is called hashing.

A key, which could be part of the data item or the entire data item, is parsed into a hash function to generate an address/index/hash value, which points to a location in a table (array) where the data is stored. This enables the data to be accessed directly.

The positions where data can be stored are sometimes known as buckets.



2 Components of a hash table

There are 2 components that we will need:

1. Hash table

   The hash table holds all the data entries in an array (implemented using a python list). The size of the array should be set according to the amount of data expected.

2. Hash function

   The hash function is an algorithm that converts a key to a value which is the address/index. Choosing an efficient hash function is a crucial part of creating a good hash table.

   The main requirements of a hash function are that it must:

   - always produce the same hash value for the same key.
   - provide a uniform distribution of hash values. This means that every value has an equal probability of being generated.
   - minimise clustering; this will arise when many different keys produce the same hash value. Where two or more different keys produce the same hash value, we say a ‘collision’ has occurred.
   - be very fast to compute.

3 Implementing the hash table

Like what we have learnt under data structures, we need to be able to perform CRUD -Create, Read, Update and Delete operations.

3.1 Defining a hash function

A Hash Function is a function that converts a given numeric or alphanumeric key to a small practical integer value such that the value can be used as the index to access the hash table directly.

An example of a simple hashing function that gives addresses between 0 and n is shown below:

```
FUNCTION hash (key) RETURNS INTEGER
  // generate address using modulo operator
  address  key MOD (n + 1)
  RETURN address
ENDFUNCTION
```

Let's use customer records as an example to illustrate calculating addresses in a hash table.

Assume we want to store customer records into a 1D array hash table and each customer has a unique customer ID, an integer in the range 10001 to 99999.

For illustrative purposes, let n be 9.

The hashing function will be:

```
index  customerID MOD 10
```

Customer IDs to store are in this sequence: `45876`, `32390` and `95312`
Using the hashing function, `45876` will give an index `6`, `32390` will give an index `0` and `95312` will give an index `2`. The customer IDs are then stored in the hash table using the index calculated from the hashing function.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876			

3.2 Implementing insert, search and delete functions

An insertion into the table can be easily performed as follows:

```
PROCEDURE insert (data)
  // hash function returns the address to insert data
  index  hash (key of data)
  hashtable[index]  data
ENDPROCEDURE
```

One can search for the data in the hashtable using the following algorithm:

```
FUNCTION search (data) RETURNS INTEGER
  // Returns index of data if found
  // Else return -1 if not found
  index  hash (key of data)

  IF hashtable[index] = data THEN
      OUTPUT 'Data found at index ', index
      RETURN index
  ELSE
      OUTPUT 'Data not found in hashtable.'
      RETURN -1
  ENDIF
ENDFUNCTION
```

Similarly, the pseudocode for the deletion operation is as follows:

```
FUNCTION delete (data) RETURNS BOOLEAN
  // Deletes data and return True if data found in hashtable
  // Else return False if data not found
  Index  hash (key of data)

  IF hashtable[index] = data THEN
      Delete data from hashtable[index]
      RETURN TRUE
  ENDIF
  RETURN FALSE
ENDFUNCTION
```

The average complexity to search, insert, and delete data in a hash table is O(1) — a constant time. It means that, on average, a single hash table lookup is sufficient to find the desired data.

4 Handling collisions

Finding a hashing function that will give a unique address from a unique key value is very difficult.

Collisions happen when two (or more) different key values hash to the same address. There are two ways to handle collisions:

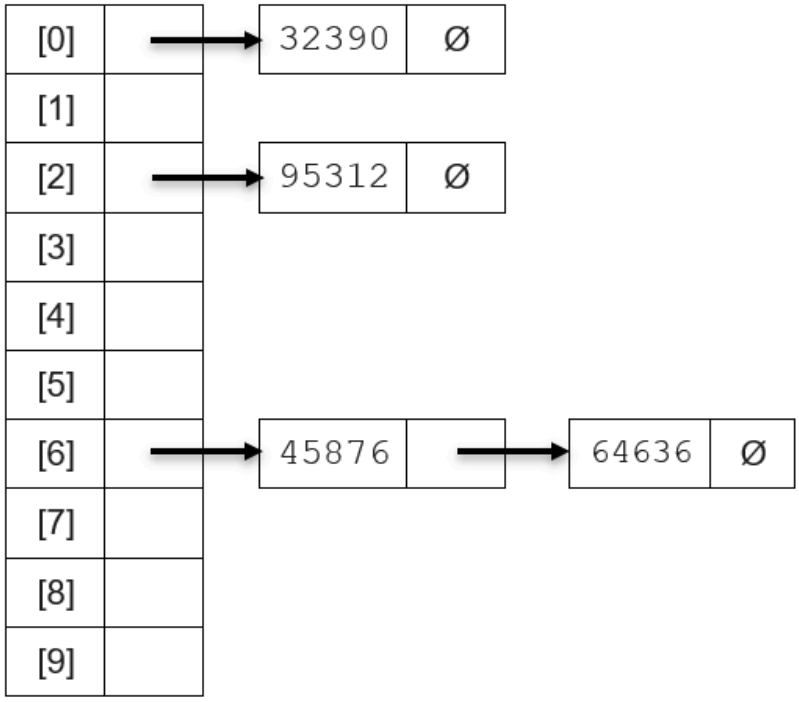
1) Closed addressing
2) Open addressing

4.1 Closed addressing

For Closed Addressing, the data is always stored in the same position it is hashed to. Collisions are dealt with using separate data structures to store all the data at the calculated address. One way to implement this is the use the idea of **separate chaining**. Separate chaining can be implemented using linked lists. When multiple elements are hashed to the same index of the hash table, these elements are inserted into a singly-linked list which is known as a chain.

Using the previous example of storing customer IDs, supposed we want to store one more customer ID `64636`, which will give an index of `6` using the hashing function, this will cause a collision with the previous customer ID `45876` already stored at index `6`.

The diagram below shows an illustration of close addressing using separate chaining implementation.



4.2 Open addressing

For open addressing, when a collision is detected, we will search for another available slot in the hash table. This will mean that we are unable to add any more data if the hash table is full (there are no more available slots)

One common method is **linear probing**. When inserting a data item and a collision is detected, we can search the next empty location in the array by looking at the next index until we find an empty location. The hash table is full when there is no more empty location.

Using the previous example of storing customer IDs in 3.1, supposed we want to store two more customer IDs `64636` and `23467`, which will give an index of `6` and `7` respectively using the hashing function, collisions will occur for both insertions.

Index `6` is already taken, hence there is a collision. If we store the new customer ID here, we will lose the previous customer ID. Using linear probing, we look up the next index to check if it is empty. We can then insert customer ID `64636` at index `7` since this is the next available space.

The next customer ID `23467`, which gives an index `7`, will not be able to be stored at index `7` as it is already taken up by `64636`. So we will need to look for a next available space to insert `23467`.

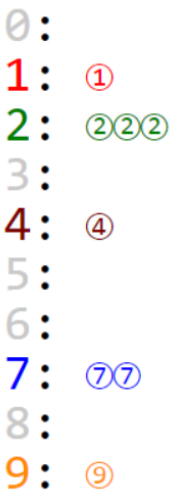
The diagram below shows an illustration of open addressing using linear probing.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
32390		95312				45876	64636	23467	

4.3 Comparisons Between Open and Closed Addressing

Characteristic structure (colors denote "home" bucket):

Characteristic structure (colors denote "home" bucket):



4.3.1 Advantages and Disadvantages of Closed Addressing

Advantages of using closed addressing technique is its easy implementation, as well as the surety that if the element is present in the hash table, it will only be found in the linked list at its key. Deletion is also quick and simple in chaining.

But chaining leads to inefficient use of memory as some addresses might never be used at all but have still been allocated space in the table. We also need extra memory allocation to store the elements as nodes in the linked list.

In the worst case, chaining can lead to linear time complexity O(n) for searching.

4.3.2 Advantages and Disadvantages of Open Addressing

Open Addressing techniques are highly efficient in memory usage as elements are stored in the space already allocated for the hash table. No extra space is r

equired.

But due to clustering, searching is slower. In the worst case, when the hash table is at full capacity, we would have to check every cell in the hash table to determine if the element exists in the hash table or not.

5 Good Hash Table

The requirements of a good hash table is to have a low chance of collision and to spread the data items evenly. There are 2 possible ways to address this.

5.1 Size of hash table

As we can see, if the size of the hash table is small, compared to the number of items to be stored, there will be a high chance of collision. In practice, the hash table is usually 1.5 times the maximum data size. We also sometimes use prime number for the size of the hash table to minimize clustering or cyclic allocation.

5.2 Good hash function

We can clearly see that our hash() function is overly simplistic. There is a high chance that it will map different key values to the same index. A good hash function should be fast to compute and able to distribute the keys uniformly, to avoid clustering and collision.

Theoretically, if we know all the possible key values, we should be able to create a hash function that has no collisions.

The following are some characteristics of a good hash function:

- Deterministic: For the same input, the hash function should always generate the same hash value.
- Efficient to compute: The hash function should be capable of returning the hash value quickly to ensure the overall efficiency of the hash table operations.
- Uniform distribution: The hash function should provide a uniform distribution across the hash table to minimise collisions. This means that every hash bucket should be equally likely for any input.
- Minimise collisions: While collisions are inevitable, a good hash function should minimise them. Two different inputs producing the same output (i.e. a collision) can slow down lookup time.
- Use all the input data: The hash function should use all the data in the input. Ignoring parts of the input can increase the likelihood of collisions.