CI583 - Data Structures and Operating Systems, 2022-23
Coursework report
Joseph Perry

Contents

---

1 Hash Tables

---

[1] Hash tables are a data structure that control access to their elements using a mapping function called a hash function, where the hash function is responsible for mapping keys to indices in an underlying array. The time complexity of accessing records in the hash table is constant time because the time complexity of both hashing the key and accessing the corresponding record is constant.

1.1 Hash functions

[2] To map a key to an index, hash functions first map a key to a very large number. Then using the modulus function, this large number is mapped into the range of possible indices for an underlying array. The first mapping is a perfect hash, meaning the mapped values are unique for each key. However since the first mapping produces number that are very large, it cannot be used directly in a hash table implementation because of hardware limitations. This is why the modulus function is used to map this value into an arbitrary size. (the size of the array)

1.2 Collisions and collision handling

[3] Using the modulus function to wrap these mappings results in a problem called collisions, which is where two distinct keys are mapped into the same index by the hash function. There are two main approaches to deal with this issue, open addressing and separate chaining. In open addressing, collisions are handled by deterministic functions called probing functions, which return a new index, probe length away from the initial index to try an insert. Probe length is either a function of the number of insert attempts or a different (different to the hash function) function of the key.

The main probe types in open addressing are linear probing, quadratic probing and hash chaining.
In a linear probe the probe length is: **insert attempts**.
In a quadratic probe the probe length is **(insert attempts)^2**.
In hash chaining, the probe length is **(hash(key) % k) + 1**, where **k** is a constant that determines the maximum probe length. **[4]** Here, the size of the underlying array must be a prime number so that no value of **k** divides it evenly.

In separate chaining, the underlying array contains lists of collided keys which can be searched in linear time for the correct record after hashing. The best and worst case time complexities of data retrieval in separate chaining are constant time **O(1)** and linear time **O(n)** respectively, where **n** is the number of keys when all keys have hashed to the same index. The best and worst case time complexities of inserting in separate chaining however are both constant time **O(1)** because hashing the key, accessing the array element and adding to a list are all constant time regardless of list length.

In general open addressing is more efficient than separate chaining because with a good hash function and probe strategy, the average time complexity of both data retrieval and access is constant time **O(p)**, where **p** is a constant representing the average length of the probe sequence. The best time complexity occurs when the probe sequence length is **1**, when there are no collisions, in this case the best case time complexity is constant time **O(1)**.

1.3 Types of clustering with different probe types

In practice open addressing often suffers from a common failure mode called clustering, which itself consists of two sub-problems called primary clustering and secondary clustering. All clustering issues reduce the efficiency of data access and retrieval from constant time **O(1)** to linear time **O(n)** in their worst case by increasing the average probe length **p** to **n**.

Primary clustering occurs most frequently when using a linear probe, where when colliding records start a cluster, further colliding records increase its size by moving to the next index after the cluster. When the cluster increases in size, the likelihood of further collisions increases too, causing the cluster to grow and the performance of the hash table to decrease as the average probe length **p** increases. Quadratic probing and hash chaining methods are less susceptible to primary clustering because they do not follow linear step sequences.

Secondary clustering is an extension of primary clustering that occurs for all probe types but mainly for quadratic probes when colliding keys follow the same probe sequence every time, causing the performance of data access to drop as the probe is forced to follow increasingly long sequences to find a valid position. Hash chaining is less susceptible to secondary clustering because the probe sequence is key dependent, meaning colliding keys usually follow different step sequences.

Since clustering cannot be completely avoided, it is necessary to manage what is called the load factor of the hash table. This is the ratio of the number of records to the size of the underlying array, or rather the fullness of the table. In practice the table should not exceed about two thirds **2/3** full, so that the likelihood of clustering remains low. If the load factor does exceed this value then the table should be resized and the records rehashed in order to reduce the average probe length **p**.

2 Open addressing hash table implementation

2.1 Time complexity of put method

[5] The worst case time complexity of the put method in the open addressing hash table implementation is linear time **O(n)** when either the table is resized or the probe sequence length is close to **n**. The time complexity of put in the resize function is worst case constant time **O(1)** rather than linear time **O(n)** because after a resize, the resize method would not be immediately called again and the find* methods would return to constant time as probe sequence length returns to close to **1**.

```
O(put) = O(
  O(invalidKey) +
  O(getLoadFactor) +
  O(resize) +
  O(hasKey) +
  O(findemptyOrSameKey) +
  O(hash)
) = O(
  O(1) +
  O(1) +
  O(resize) +
  O(hasKey) +
  O(findemptyOrSameKey) +
  O(1) +
) = O(
  O(resize) +
  O(hasKey) +
  O(findemptyOrSameKey)
) = O(
  O(n) +
  O(n) +
  O(n)
) = O(n)
```

```
O(resize) = O(
  O(for) +
  O(nextPrime) +
  O(for)
) = O(
  O(n) +
  O(1) +
  O(n) * (
    O(1) +
    O(put)
  )
) = O(
  O(n) + (O(n) * O(put))
) = O(
  O(n) * (O(1) + O(put))
) = O(
  O(n) * O(1)
) = O(n)
```

```
O(haskey) = O(
  O(hash) +
  O(find)
) = O(
  O(1) +
  O(n)
) = O(n)

O(find) = O(
  probe sequence length
) = O(n)

O(findEmptyOrSameKey) = O(
  probe sequence length
) = O(n)
```

In the best case, the put method does not result in a table resize (highlighted in red above) or long probe lengths (highlighted in yellow above) so would be constant time **O(1)**. In the average case, the time complexity would be constant time **O(p)** where **p** is an average probe sequence length less than **n** and greater than **1** if the table is not resized.

2.2 Time complexity of get method

```
O(get) = O(
  O(find) +
  O(hash)
) = O(
  O(n) +
  O(1)
) = O(n)
```

The worst case time complexity of the get method in the open addressing hash table implementation is linear time **O(n)** when the probe sequence length is close to **n**.

The best case time complexity of the get method is constant time **O(1)** when the probe sequence length is close to **1**, whilst the average case time complexity would be constant time **O(p)** where p is an average probe sequence length less than **n** and greater than **1**.

2.3 Practical applications

Hash tables are used in a variety of applications including:

- **[6]** Dictionaries are a generalisation of hash tables that ultimately define the same functionality of storing (key, value) records. It is worth mentioning hash tables in this context because hash tables are not the only way to implement dictionaries, where implementations often use different types of search trees.

- **[7]** Hash tables can also be useful for efficiently mapping directory names to physical memory blocks, however the extra handling required for deleting hash table records in open addressing makes it impractical for this task. Separate chaining might be more appropriate since data access sequences are not dependent on occupied records as is the case with open addressing.

3 Neural network hash table implementation

Neural networks are a type of mapping function, their main strength is their flexibility in modelling many different kinds of associations. I considered using a neural network trained to map keys to indices as a hash function and implemented; a neural-network-based hash table allowing deletions, unit tests for the hash table, a neural network and a matrix arithmetic class.

The neural network is implemented as a fully connected (dense) network with sigmoid activation functions and cross entropy cost function. Parameters are initialised with random values sampled from the standard normal distribution and optimised by gradient descent. The network learns the associations between normalised key values and one hot encoding labels in order to map keys to indices. The mapped index is determined by the index of the output node with the maximum activation.

The key normalisation works by calculating the min-max normalisation **[8]** of each character in a key using the minimum and maximum character range in the table. The normalised values are returned in a zeros column vector, with a size preset by a maximum key length.

3.1 Time complexity of put method

For every unique key inserted to the table, the network mapping must be updated with gradient descent. This makes the worst case time complexity of the put method the time complexity of gradient descent, which can be expressed as **O(kn)**, where **k** is the maximum number of training steps needed to converge and **n** is the number of records. **[9]** In the average case, the time complexity expression is the same, except **k** is the average number of training steps needed to converge. Since **k** is somewhat related to **n** these are both roughly polynomial time.

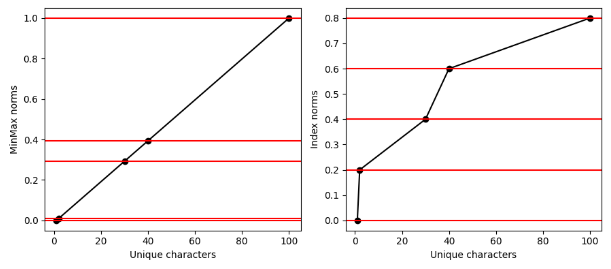
The best case time complexity is constant time **O(1)** when overwriting a key already in the table because an update is not called and the time complexity of both mapping the key (the time complexity of forward propagation) and writing to the array elements are not related to **n**.

3.2 Time complexity of get method

The get method is constant time **O(1)** in all cases because the complexity of both mapping the key (the time complexity of forward propagation) and accessing the array elements are both constant time regardless of **n**.

3.3 Limitations and possible improvements

The main failure mode for this method occurs when the network fails to converge on 100% classification accuracy before it reaches the maximum number of training iterations. In this case the network has formed an incorrect mapping, making some records inaccessible and some records at risk of being duplicated.



The min-max key normalisation (left) results in characters that are not easily distinguishable. The ideal key normalisation (right) could be calculated based on the sorted index of each unique character in the table and accessed with either a lookup table or a quadratic curve fitted to the ideal key normalisation.

In some cases this is the result of the min-max normalisation, (Left, left) where when the keys contain an outlier character the rest of the characters will be made much harder to distinguish. Keys with differences encoded in single or limited positions in the string are also harder to distinguish because dense networks rely on distinct node activations to form models effectively.

Data preprocessing methods that distribute the character normalisations evenly (Left, right) would solve the first issue described but to deal with indistinct input node activations generally, the network would need a better optimization strategy that used momentum or adaptive learning rates. **[10]** A better optimization strategy is probably the main improvement needed for the implementation as it would mean faster update times and fewer failures.

4 Data structures in operating systems

4.1 Linked lists in FAT file systems

[11] A FAT formatted drive is divided into a number of evenly sized blocks, where the first block contains the File Allocation Table or FAT, the second block contains the Root directory table and the remaining blocks are referred to as the Data region. The FAT is an array with an index for each of the blocks in the data Region and in each index is an index describing the next related block. In this way, the FAT describes a number of linked lists which are traversed for file access.

The Root directory table describes the files in the root directory of the file system. Each record in this table includes information about a file including its name, size and starting block. To access the whole file, the linked list described by the index in the FAT specified by the file's starting block is traversed until the whole of the file has been accessed. Directories in a FAT file system are special files that describe a number of sub-files in the same way as the Root directory table.

FAT is often too slow for modern use since traversing linked lists is linear time and so random access within files is **O(n)**, where **n** is the total number of blocks in a given file.

4.2 Binary trees in the Buddy System

[12] The Buddy System is a method for dynamically allocating memory, where all block sizes are a power of two. When inserting data, the requested storage size is rounded to the nearest power of two and inserted to the first available block of equal size. (initially the full size of memory) If no available block is equal in size then the next smallest block is split in two, forming two buddies. If the first of these buddies is equal in size then the data is inserted, otherwise the buddy is split again until a suitable block size is made available.

This process partitions the available space into a binary tree, making allocating and de-allocating data fast. The buddy system can result in high degrees of fragmentation when relatively small data is inserted and the space must be partitioned into many progressively smaller buddies to accommodate it. Data only slightly larger than a power of two also causes large degrees of fragmentation, where blocks almost twice as large as the data must be allocated.

4.3 Stacks in runtime environments

[13] In a runtime environment a stack called the control stack is used to hold information about unexecuted processes. Each process occupies a unit called a stack frame, which contains all the data needed for its execution such as local variables and memory addresses.

The stack itself is an array with two pointers, the stack pointer and the frame pointer. The stack pointer points to the beginning of the current stack frame and the frame pointer points to the end of the current stack frame. New stack frames are added to the stack at the position specified by the frame pointer and to remove completed stack frames, all that has to be done is move both pointers down the stack to the next stack frame, letting older stack frames get overwritten.

4.4 Queues in scheduling systems

[14] The simplest form of process scheduling is in a Simple Batch System, (SBS) which use either a first-in-first-out (FIFO) queue or a priority queue to order the sequential execution of processes, or jobs, allowing each job to run to completion. In the case where a FIFO queue is used, jobs are executed in the order they were requested. In the case where a priority queue is used, an estimate of each job's execution time is used to assign priorities to the jobs, so that shorter jobs are executed first. These are called Shortest Job First (SJF) schedulers.

SJF based SBSs have a better average throughput (more jobs will be completed for an amount of time on average) than FIFO based SBSs but long jobs might be held at the back of the queue indefinitely if shorter jobs are continuously queued.

A Multiprogrammed Batch System (MBS) can partially solve this issue by running several SBS threads containing jobs grouped by relative priority simultaneously, passing control between them when any job exceeds an execution time limited by a value called a time quantum, where jobs exceeding the time quantum are reinserted to the back of the queue. This way, the SJF routine can be maintained without trapping long jobs at the back of the queue as certainly.

The priority routine can be improved on further by multilevel feedback queues, where the priority of jobs decrease whilst waiting and increase whilst running. This is achieved by moving jobs to a lower priority queue every time they exceed the time quantum, rather than to the back of the same queue.

References

1. Jim Burton 2022, *week5a.pdf*, lecture notes, CI583 Data Structures and Operating Systems, Brighton University, delivered Sept–Dec 2022

2. Rahul Kumar Yadav 2022, *Examples of Hash Functions*, OpenGenus, viewed 11 January 2023 <https://iq.opengenus.org/hash-functions-examples/>

3. Jim Burton 2022, *week5b.pdf*, lecture notes, CI583 Data Structures and Operating Systems, Brighton University, delivered Sept–Dec 2022

4. Jim Burton 2022, *week5c.pdf*, lecture notes, CI583 Data Structures and Operating Systems, Brighton University, delivered Sept–Dec 2022

5. Andrew Tomazos & JJJ 2017, *How can I find the time complexity of an algorithm?*, Stackoverflow, viewed 11 January 2023, <https://stackoverflow.com/questions/11032015/how-can-i-find-the-time-complexity-of-an-algorithm>

6. Christopher D. Manning, Prabhakar Raghavan and Hinrich Schütze 2008, “Search structures for dictionaries” in *Introduction to Information Retrieval*, Cambridge University Press, viewed 11 January 2023 <https://nlp.stanford.edu/IR-book/html/htmledition/search-structures-for-dictionaries-1.html>

7. Jim Burton 2022, *week9f.pdf*, lecture notes, CI583 Data Structures and Operating Systems, Brighton University, delivered Sept–Dec 2022

8. Jason Brownlee 2020, *How to Use StandardScaler and MinMaxScaler Transforms in Python*, Machine Learning Mastery, viewed 11 January 2023 <https://machinelearningmastery.com/standardscaler-and-minmaxscaler-transforms-in-python/>

9. K. Bergen, K. Chavez, A. Ioannidis, and S. Schmit 2015, *Lecture 11*, Lecture notes, CME 323: Distributed Algorithms and Optimization Spring 2015, Stanford University, delivered 5th April 2015, viewed 11 January 2023 <https://stanford.edu/~rezab/classes/cme323/S15/notes/lec11.pdf>

10. Sebastian Ruder 2016, *An overview of gradient descent optimization algorithms*, ruder.io, viewed 11 January 2023 <https://ruder.io/optimizing-gradient-descent/index.html>

11. Jim Burton 2022, *week9a.pdf*, lecture notes, CI583 Data Structures and Operating Systems, Brighton University, delivered Sept–Dec 2022

12. Jim Burton 2022, *week7c.pdf*, lecture notes, CI583 Data Structures and Operating Systems, Brighton University, delivered Sept–Dec 2022

13. Jim Burton 2022, *week7b.pdf*, lecture notes, CI583 Data Structures and Operating Systems, Brighton University, delivered Sept–Dec 2022

14. Jim Burton 2022, *week8c.pdf*, lecture notes, CI583 Data Structures and Operating Systems, Brighton University, delivered Sept–Dec 2022