**Data structures and operating systems**

**Hashtable implementation and reflective report**

**Section One - Discussion of complexity**

A hashtable is an abstract data structure that associates or maps values and keys to, in our case an array of pair objects - using a hash function on the keys we can calculate an index and store them, these hashed keys essentially become an address for the original key and value pair which provides fast and efficient lookup in constant time *O(1)* rather than having to run a loop over the array which would have the time complexity of *O(n)* or linear time.

(Goodrich, M.T. and Tamassia, R. 1998)

My put method to insert pairs will have a time complexity of *θ(1)* amortized run time assuming that in our case the itemCount divided by the size of the array doesn’t exceed our maxLoad, this is because the vast majority of the time the resize method that will increase the time complexity will not be called upon and only the hash function, hasKey and findEmptyOrSameKey methods will be, a good hash function will reduce collisions, this is when two different keys when hashed return the same value or address, in our case we simply overwrite the value attributed to the pair, it will do this by evenly distributing the pairs around the array – the hash function itself has a time complexity of *O(k)* where *k* is equal to the length of the key string, in general key lengths are negligible so it wouldn’t affect the time complexity which is why hashtable put methods are still considered *O(1)* or constant time.

*(CS 312 lecture 20 hash tables and amortized analysis*, no date)

My hasKey method uses the get method which is *O(1)* at best i.e. when the hashed key returns an empty address and we can immediately store our key/value pair, otherwise the get method will have a time complexity of *O(n)* the same applies to the findEmptyOrSameKey method the majority of the time these hashed addresses will be empty and these two methods can run at *O(1)* rather than *O(n)* this is due to the hash function evenly dispersing the pair objects around the array and due to the loadFactor.

Every so often when the hashtable is being filled and itemCount increases the loadFactor which is the ratio of the itemCount to the max or measure of how full the table is exceeds the maxLoad in our case 60% full - this is when performance may start degrading and collisions start to increase, to maintain the constant time complexity we call the resize method to rehash all the key/value pair objects back into a new array double the size, we double the size to reduce the amount of times we have to call the resize method by reducing the loadFactor. The resize method itself runs at *O(n)* where *n* is the length of the array because it has a for loop to traverse the length of the array, it also has the put method nested in the for loop but as it has a time complexity of *θ(1)* it does not affect the overall complexity of the resize method.

Put Method

* Best case – *O(1)*
* Average case – *θ(1)*
* Worst case – *O(n)*

My get method is a simple if/else statement that calls the find method with the hashed key (startPos), key and stepNum passed as parameters, the find method will have a time complexity of *O(1)* when the call to find returns null or returns the optional.of(value) associated with the key, else it will call a recursive find with getNextLocation as the startPos parameter and increment stepNum by 1 which is in linear time and has the complexity of *O(n)* – again a good hash function should reduce the collisions and keep the get method at a constant time of *O(1)* as the recursive call should rarely be called upon.

Get Method

* Best case – *O(1)*
* Average case – *θ(1)*
* Worst case – *O(n)*

***Open Addressing vs Separate Chaining***

Another form of collision resolution is separate chaining which is where each address in the array of the hashtable is a linked list, so when a collision occurs rather than overwriting or searching for an empty address its simply added to the next node in the linked list - using another data structure for collision resolution.

The put or insert method using separate chaining will still have an average constant time of *θ(1)* as regardless of whether there is an object at the address or not it will be added to the linked list at *O(1)*.

The get or search method using separate chaining is directly proportional to the loadFactor giving it a time complexity of *θ(1 + α)* where *α* is the loadFactor, so if the loadFactor is *O(1)* so will the get method in general there is no difference in complexity between the two type of collision resolution.

(Cormen, T.H. *et al.* 2022)

***Applications For Hashtables***

We would want to use a hashtable in any situation where we would want to store, retrieve and potentially delete data efficiently in constant time based on association using their unique (hashed) key and value or attribute.

Common applications:

* Databases – in databases we use indexes to store and retrieve data, using a hashtable is very common with database indexing.
* File System – a hashtable can be used to associate file names with the actual path of the file and the location of the file on memory.
* Caches – a hashtable can be used to store recent states and if new data is added we can overwrite existing states.
* Password Verification – Log in credentials can be stored using a key/value where the value could represent a password on client side and be hashed and compared to the server-side table.

Languages and Compilers – Different languages incorporate hashtables e.g. Python dictionaries and Java HashMaps and compilers will store keywords in hashtables for quick compilation. (AfterAcademy, A. 2020)

**Section Two - Data structures and operating systems**

**Stack in Assembly Language**

The assembly language is known as a low-level language used to communicate with the CPU this is needed on many occasions including when a high-level language needs to be compiled.

The stack itself is a LIFO (Last In First Out) data structure with the only access to the stack being the top element - elements are always removed in reverse order to how they were put in.

In assembly language the stack is a data structure of parameters, variables and function calls but with a stack pointer, this pointer is used to point to the address in memory that is being used, adding to the stack is called a “push” and removing data is known as a “pop”.

Every time a function is called the address in memory of the next instruction for execution is pushed to the stack and when the function has finished executing its code it is then popped off the stack and the next address or function moves to the top.

(Bolton, D. 2019)

**Hash Functions in File Systems and Directory Listings**

Modern file systems will not limit you on the number of files you can store in a directory (storage permitting) but as the directory increases in size the way to maintain decent performance is to use file name hashing.

As discussed earlier the hash function will create a unique hashCode from a key which can be used to associate file names with the actual path of the file and the location of the file on memory – although a hashtable can be used other data structures can be used as well producing fast insert, search and remove functionality within the file system and directory listings.

(Andrews, M. 2019)

**Use of a Binary Algorithm in the Buddy System**

The Buddy System is a memory management and allocation technique where lists are created to store free memory blocks in sizes of 1, 2, 4, 8 all the way to *n* where *n* is the actual size of memory in bytes, e.g. one megabyte memory requires 21 lists.

Lists are held as powers of two meaning if we had a 112K process needed to be put into memory we would split the one megabyte into two 512K memory blocks hence the name “Buddy System”, then split one of the 512K into two 256K blocks and then one of the 256K blocks will be split into two 128K block, one being left free the other being allocated to the process – in this scenario we would have three lists with one entry and a wasted 16K which is known as *internal fragmentation*.

When a process ends it simply releases its block of memory and if its buddy, or adjacent block is free as well they will be merged, the merging process is fast as you only check the buddy block next to you and adjoining address above so if the returned block was 64K bytes in size only the 64K list has to be searched. (Kendall, G. no date)

# **References**

AfterAcademy, A. (2020) *Applications of hash table*, *AfterAcademy*. Available at: https://afteracademy.com/blog/applications-of-hash-table/.

Andrews, M. (2019) *File name Hashing: Creating a hashed directory structure*, *Medium*. Eonian Technologies. Available at: https://medium.com/eonian-technologies/file-name-hashing-creating-a-hashed-directory-structure-eabb03aa4091.

Bolton, D. (2019) *How to understand a stack in computer programming*, *ThoughtCo*. ThoughtCo. Available at: https://www.thoughtco.com/definition-of-stack-in-programming-958162#:~:text=A%20stack%20is%20an%20array,first%20out%E2%80%9D%20or%20LIFO%20order.

Cormen, T.H. *et al.* (2022) “Hash Tables,” in *Introduction to algorithms*. Cambridge, Massachusett: The MIT Press, pp. 276–282.

*CS 312 lecture 20 hash tables and amortized analysis* (no date) *Lecture 20: Hash tables and amortized analysis*. Available at: https://www.cs.cornell.edu/courses/cs312/2008sp/lectures/lec20.html#:~:text=Each%20resizing%20operation%20takes%20O,performance%20is%20O(n).

Goodrich, M.T. and Tamassia, R. (1998) “Hash Tables,” in *Data Structures and algorithms in Java*. New York: Wiley, pp. 522–552.

Kendall, G. (no date) *G53OPS - operating systems*, *G53OPS : Memory Management*. Available at: https://www.cs.nott.ac.uk/~pszgxk/courses/g53ops/Memory%20Management/MM08-buddysystem.html.