**Data structure and operating system**

By Joseph Batchelor

**Section 1: Hashtable and complexity**

A hashtable is an unordered data structure that implements a simple notion of an array. This structure maps keys to values by using a hash function to calculate an index which creates a hash code, this is used in an array to find or store a desired value. This unique and effective structure provides direct addressing, which is the ability to inspect a determined position within the array. This means elements are found in constant time, O(1). A key can be broken down into a unique index value that is used to store a key-value item. Every key will have a value that usually has an association to it, this can be in the form of a string, a value and even a shape. The idea of hash tables is the ability to perform quick look up for these values.

An important part of hash tables is the hash function, with comparison to my implementation, a hash function will take in a string and convert it into the form of an integer, this will then remap that integer into an index in the array, which is where we can find that given key and value. An important consideration is that the hash code isnt the actual index in the array, as we map from the key to the hash code and then over to the index. The reason behind this is that the array stores data from the hashtable which might be much smaller than the potential hash codes, which is why we remap it into something smaller.

An interesting possibility is that two different strings could have the same hash code this is due to there being an infinite number of strings and a fine amount of hash codes. This can mean multiple strings can be mapped to the same index, when this happens this is known as a collision. Collisions causes conflict in linear probing; a resolution is the approach of introducing chaining. Chaining is the idea of placing all the elements that hash to the same key into a linked list. The run time of a hash table are generally constant time O(1). However, hash tables with a faulty hash function can be linear time O(n), this due to collisions which effects the time.

With my implementation the time complexity of my hashtable is constant time O (1) this is since elements can be directly addressed as well as my hash function is not faulty and that collisions do not occur. My code can be broken down into various methods where each perform important functions, below I will be explaining method put and get as well as any methods called and used within them. Additionally, I will be discussing the potential occurrence of certain scenarios and any additional scenarios that are important, such as when the hashtable is empty, reaching its max load factor.

Starting with put, this method is designed to insert values into the table. This method plays an important part as it must determine certain conditions before inserting a value. The conditions that put must consider are what to do if the key is present within the table, where to store it if the key is not in the table and finally what to do if the key is null. A crucial feature that method put does is the ability to resize the array, this will occur when the load factor is greater than the max load. Resizing will double the array size while making sure it is the smallest possible prime number, it uses methods resize and load factor to determine when to perform this action. Method put checks if a key is present within the table by using method haskey to decide whether to override the key if it is present or to create a new key in an empty location. Finding a position for the new key is done by using methods findempty and hash essentially the key will be hashed to calculate an index which is then used in findempty to check if that location is null, if the location is not then it will search through the hashtable to find the next available null position.

Finally, If a key was null then put will throw an “IllgealArgumentException”. A possible scenario that can occur within method put is the occurrence of existing data being stored within the table, thus creating duplicates. To avoid any duplicates any matching key-value items will be overwritten replacing only the value so that only one item exists per value. This prevents collisions occurring as well as an increased amount of unnecessary data. The worst time complexity of method put has been calculated to run at quadratic time O(n2). This occurs when the hashtable needs to be resized, within the first if statement it checks if the load factor is greater than the max load if this occurs then method resize is called which has a performance of quadratic time. A quadratic performance means that the number of steps taken to complete a task is square of n. This happens in the last part of method resize where two iterations take place giving a worst time of O(n2). Additionally, the best possible time that method put can run at is constant time O (1), this can occur if the array does not resize and the key found is null as this will throw an “IllegalArgumentException” which is done in constant time O(1). This means the time is not affect by the value of N or in this case the size of the table.

The next method is get which is designed to find and access keys within the table. It is done by hashing the key given and using the calculated hash code within method find as the starting position to search the table, allowing us to utilise direct addressing. The time complexity of method get performs at linear time O(n). The reason behind this is that method hash is called which performs at an overall of linear time O(n), additionally method find is called and used within the return statement this means when method find is called and executed it performs in linear time. This means in terms of complexity method get will increase the time taken to search for a key as the table grows because more positions will need to be searched and that the longer the key is the more iterations hash must perform.

Haskey is a method within my code that is tasked to check whether a key is present within the table. It works with method get to check if the key it is looking for is currently in the table. It will return true if the key is present within the table and false if not, this means it will then be added which is done within method put. The overall time complexity of haskey is linear time O(n) for all case scenarios, this is because method get is called within the condition of the if statement meaning that either return value will run at linear time as it must call and execute method get first to get the value to compare and then decide which cannot be avoided, thus giving the overall method a run time of O(n).

Method getLoadFactor is designed to calculate the load factor of the hashtable. The calculation for the load factor is the number of items within the array divided by the size of the array. The load factor is always changing, this happens when method put adds elements into the hash table, as the number of items in the array increases so does the load factor. The load factor is an important feature that is designed to determine how full the dataset is. It is used in method put to determine if a resize is needed. The reason for this is that the load factor will change if a resize occurs. If the array was full this simply means that no more elements can be added which is then why we resize so more elements can be added if needed.

The way the load factor is used within method put is to compare the current load factor to the max load which has been set to 0.6, the max load is the point at which it determines when the array will need to resize. The reason the value has been set to 0.6 is if the value was too low it will require the array to be resized too frequently, creating a larger percentage of null positions compared to stored key-values. If the value is set to high it may mean that the array might not resize at all, this is because there is a possibility that the last element can’t be added because its greater than the size of the hashtable while the load factor still remains under the max load. The overall time complexity for all case scenarios for method load factor is constant O(1), this is because the size of N does not affect the time as it’s just a simple return statement.

A scenario that will occur is the load factor reaching its maximum load factor. In my implementation the max load will be reached when the hashtable is 60% full of items, once it exceeds that amount it will have to perform a resize to reduce the load factor. When we resize the table with the new size this will be double the original which means that the load factor will drop down to half its value meaning more items can be stored.

A consideration I made regarding the relationship between time complexity and the load factor in terms of performance within linear probing is that. We measure the fullness of a hashtable with the use of the load factor and measure the run time execution of code with the use of time complexity. The relation between the two measurements is proportional which means they increase at the same rate. In this case If the load factor was low, this means that not enough items are present in the hashtable and therefore the chances of directly addressing the key-value pair at their right index is high, this means it will perform at constant time O(1). However, when the load factor is high the chances of directly addressing the key-value pair into their exact position is low and therefore we will need to use method find to search through the table to find the key-value pair where the complexity will rise to be linear time O(n). Therefore, if the load factor were to be 1.0 then the time complexity would perform at linear time O(n). Additionally, the time complexity of a hashtable would be affected as collisions would be more common if the load factor was high, therefore the overall time of the hashtable would perform at linear time O(n).

Method find is designed to search the array and give the location of where the desired key is stored. Method find is called and used within method get, the way this works is that method get converts the given key to a hash and within the return statement method find is called where the parameter for startpos is set to be the hashed value which is where direct addressing is introduced. Method find has three parameters which are startpos which is where the method will start its search for the key, key the non-hashed key we are looking for within the array and stepnum this is the number of steps to take when to move on. The way this method starts is by creating an object that pairs the key – value of the given startpos within the table. It then checks if the newly created object has a key in that position within the array and checks if it is null. This should test whether the key is present within the array, if it is not then the search will stop. If this is the case, then this can be considered the best scenario in terms of time complexity with a performance of constant time O(1) as the method will only preform one action before returning null.

However, if the key were to be present within the array then it would test if the objects key matches the key we are looking for, if it is a match the find method will return its value. However, if the keys do not match, method find will perform a recursive call with a new startpos and repeat until a value or null is returned. The new startpos is created using the method getNextLocation which finds a new location to start searching depending on the probe type, this can be considered the worst scenario of method find with a complexity of linear time O(n). The reason its linear time is because variable startpos is set to the return value from method getNextLocation which has an overall time of O(n). Additionally, when performing the recursive call this performs as O(n) because its only called once and acts the same as a loop.

Find Empty method is like the find method except instead of looking for stored keys within the array it looks for empty locations where keys could be stored. This method is called within method put to store new keys within the array. With the same concept as method find three parameters are used to fill method getNextLocation. Method findempty iterates through the array looking for a null location for startpos. If startpos doesn’t equal null the method will then call method getNextLocation to find a new location to check, this will keep on calling until a null position is found.

Once found it will then return the empty location to be used. However, it is possible that the given startpos is already null this means that it will return straight away, if this were to happen this can be considered the best scenario in terms of time complexity with a performance of constant time O(1) as it would not need to loop as it will simply return what was given. However, the worst performance is at linear time O(n), this means it will have to perform the while loop and search for a null location which will take longer when the array increases in size, additionally method getNextLocation will run at linear time when executed in method find empty.

As mentioned earlier an important feature of a hashtable is the ability to hash keys, this is used to create an index also known as a hash code which will be used to store key-value items in the table. This is where method hash is utilised, in my implementation we are hashing string keys which are essentially words being converted into a hash code. There are hundreds of different variations of hash functions each performing different calculations depending on the type of key. My version starts off execution to test if the key is null and will throw a NullPointerException if it is, this will finish the execution of the method hash at constant time O(1) which is the best-case scenario.

However, if the key is not null then it will be able to calculate a hash code. It does this by creating two varibles hash and prime, hash will be used to return our hash code and starts off at 0. Where variable prime is a small prime number, the reason prime numbers are used is that there is a higher chance of creating a unique hash code which ultimately reduces the chances of collisions to occur and the benefit of using a small one is that it will store values early on within the table. The next step is iterating through the key which is the word. I do this by looping through the key and at every letter within the word we perform a calculation which is hash = (prime number x hash + letter index) % the length of the array, this will be done for each letter, notice how the variable hash will be equalled to the previous letter incrementing every letter. within the word until the iteration is complete, you should be left with a final hash code to be used within the array. This means the worst case would be linear time O(n).

An important method is Resize; this will be used constantly within method put to increase the size of the array, this will occur when a certain number of items are in the hashtable. Its work coincides with isPrime and nextPrime to make sure that the next size of the array is a prime number. The way this method works is once the load factor exceeds the max load - which is done determined in method put - this will call method resize. Method resize will doubles the current length of the table which will halve the current load factor. However, before setting the new length we must check whether n is a prime number this is done by using both isPrime and nextPrime to test if n is a prime number and if it needs to find the next smallest possible prime number.

Once the new size has been established, we than use a temporary array which will hold our current array meaning the unaltered size previously and items contained are used. We then create a new array with the new size, at this point the array will be empty and will need to be filled with the previous items. This is done by iterating through the temporary array created earlier and comparing every index to see if they contain a key-value item this will essentially copy over the previous items making it ready to start storing new items. This process can be done numerous times gradually increasing the size of the table until elements are no longer being stored and resizing is not needed anymore. As mentioned previously a scenario that briefly occurs is the possibility that the hashtable becomes empty. This scenario happens in method resize when we create a new hashtable with the increased size as soon as it is created it is automatically empty however, it will start to copy over the previous items from the temporary array. Every time method resize is used the hashtable will always be temporarily empty.

Chart, line chart

Description automatically generatedMethod resize performs at a time complexity of quadratic time O(n2) for all case scenarios. This happens when the for loop iterates through the hashtable to copy over the items from the previous table to the newly created one. While the for loop is iterating method put is called within the if statement to store the previous key-value items, because method put calls method find empty where this performs a loop to find an empty location at linear time. This means that overall, with both the for loop and the find empty loop within method put, this will perform at quadratic time O(n2). A quadratic performance means that the number of steps taken to complete a task is square of n, in terms of method resize this means that for every iteration the for loop performs in resize another complete iteration is done by method put.

Image source: [Big o Cheatsheet - Data structures and Algorithms with thier complexities | HackerEarth](https://www.hackerearth.com/practice/notes/big-o-cheatsheet-series-data-structures-and-algorithms-with-thier-complexities-1/)

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Best case** | **Average case** | **Worst case** |
| put | O(1) | O(n2) | O(n2) |
| get | O(n) | O(n) | O(n) |
| Haskey | O(n) | O(n) | O(n) |
| getkeys | O() | O() | O() |
| getLoadFactor | O(1) | O(1) | O(1) |
| getCapacity | O(1) | O(1) | O(1) |
| Find | O(1) | O(n) | O(n) |
| findEmpty | O(1) | O(n) | O(n) |
| getNextLocation | O(n) | O(n) | O(n) |
| doubleHash | O(n) | O(n) | O(n) |
| Hash | O(n) | O(n) | O(n) |
| IsPrime | O(1) | O(n) | O(n) |
| nextPrime | O(√n) | O(√n) | O(√n) |
| resize | O(N2) | O(N2) | O(N2) |

**Section 2:** **Data structures and operating systems**

**Operating systems**

Developed in the mid 60’s operating systems are a collection of low-level software that is designed to support the functions of a computer. Comprised of four different levels which are User, Application, Operating system, and hardware this allows OS’s to process and access certain resources such as controlling peripherals, memory management and process management. Important components of an OS are files systems, schedulers, and device drivers. Below are some applications that utilise algorithms and data structures to perform computational functions for computers. Some of these concept utilizes the different stages of an operating system.

**Stack and stack frames in assembly code**

Assembly code is a low-level language designed to directly communicate to the CPU to make the computer perform actions. This can occur when high level language is being complied. A Stack is a data structure like an array that is designed to add and remove data in a last in first out system. You store elements onto the stack where the top item is always taken off first, elements can only be removed until it has risen to the top, meaning every element before must be removed first.

A programs stack is a stack that can be found in the memory usually with a fixed size that behaves as a last in first out buffer. Its job during execution of code is to store temporary variables created by a function such as parameters. The way stack memory works in assembly language is during execution parameters are pushed onto the stack when a function is called and varibles are popped off from the stack for the function. The concept is based off the stack data structure with some minor adjustments such as the use of the stack pointer, which is a component within the stack memory, it is used to point to the area of the stack that is being used. This introduces more flexibility than the last in first out system as it can adjust the area of what is being addressed by the stack.

A stack can be made up of many stack frames. Where a stack frame or frame is a divided part of data from the stack. A frame contains data such as the parameters for a function, the address of a function and its variables and their addresses. This frame of data is placed onto the stack, where each frame is its own function to be called. Essentially creating a stack full of different stack frames where each one contains a function and data regarding that function.

**Disk map data structure in the S5FS filesystem**

A file system is used by the operating system to perform different file operations. Specifically, System 5 file system (S5FS) is a filesystem designed for Unix systems especially OS version 5. This system allows files to be access efficiently while maintaining a simple process and design. In a S5FS formatted disk the layout is broken down into 1024-byte blocks, the first block always contains the superblock which describes the layout of the disk. This means it records the characteristics such as the filesystem type, size, and information about other metadata structures.

The next block contains an i-list this is like an array that is designed to hold inodes, an inode is a type of data structure which is used to hold information about a file or directory. Each inode stores data and attributes about a file, the number of inodes can indicate the number of files or directories present. The final block is the data region which contains the disk blocks holding or referring to file contents. Each inode describes a file this means that inodes transfer files internally, which is done through the inode’ s index in the i-list. Directories contain pairs of components such as names and inode numbers which is used to identify the inode of the file containing the next directory in the path. Reading the contents of a directory file which can be taken from the data region provides a path to the next component.

A useful section in inodes is the disk map this is a data structure that contains the disk blocks holding the files data. This allows for quick access, deletion, and compression of files, this is done by using logical block numbers to map to physical blocks. An inode stores 13 data block addresses which can be considered as pointers. The first 10 of these point directly to the first 10 data blocks of the file, this makes the concept efficient for small files. However, for the 11th data block this points to a block that contains a table of addresses that points to the next data blocks of the file. This allows us to increase the amount of data blocks a file can use the problem is this introduces a new level of indirection which means it reduces the time to access, as it is not directly addressing the block like previously. The 12th pointer creates two layers of indirections and then the 13th introduces three-layer indirection. The greater the indirection layer the more data blocks the file can have. The way this concept functions is that instead of mapping a file name to the first file block like FAT would do, it will map its file name to an inode which will be the path to access the file.

Typically, when mapping it should always start at the \ root directory this being the top level of a file system the way to access a simple text file is done like this. By starting at the \ root directory inode this allows you to follow the data pointer which finds the data of the \ root directory in the foo folder. The foo directory normally holds all stored files that the user saves on a UNIX system. Next, we find that \foo maps to a new inode by following the address which allows us to find the data block address for \foo, this is where we find the contents of \foo, this is where the text file can be found. From there it is going to map the text file to the correct inode, by using the structure explained earlier we can piece the data we need.

**Use of a binary algorithm in the Buddy System**

Buddy System is a dynamic storage allocation concept that is easy to implement, there are four different types of buddy systems the one we will be discussing is the binary buddy tree. The idea is that storage is held in large memory blocks where the sizes are powers of two. When a memory request is made, they are set to a power that is greater than the request size, if a block of that size is available, it will be filled. However, if one is not available then a block that is larger than the desired size while being the smallest possible available block is divided down into two equal pieces, this is done to satisfy the memory request, these two new blocks are called buddies. If the size of either buddy is set to what is needed, one of them is will be allocated while the other will remain free. Instead, one of the buddies is divided in half and will continue to be divided until the right size is found.

In detail if the size of the memory block is 28 or 256kb and the kernel requests 25kb of memory. The block is divided into two buddies of 27 or 128kb this will give you two blocks which we can label block A1 and block A2. Either block A1 or A2 will be split into two 26 or 64kb buddies giving us blocks B1 and B2. Because the request is 25kb the next highest power that 25kb will fit in is 25 or 32kb so, either B1 or B2 is divided into two 32kb buddies creating blocks C1 and C2 and finally one of these buddies is used to satisfy the 25kb request. Once a block has been divided it can only be combined with its buddy block, this will then create the previous larger block they were separated from, for example C1 and C2 can form B1.

The navigation and data layout share a similar style to a Binary tree algorithm. A Binary tree is a data structure that is made up of nodes, where each node contains a left and right pointer and a data element. A binary tree node can only have a maximum of two children. Typically, the buddy memory allocation concept is designed with the implementation of a binary tree the reason for this is that it can outline used or unused divided memory blocks. Due to the way the buddy system concept functions, this means that if a program were to request 65kb of memory this would be allocated within the 128kb block as the power before is 64kb which is just under the request size, However the problem with this is that it will waste of 63kb of memory.

**Research Material for report**

**Doeppner, T.W.**

**Book:** Operating systems in depth

Doeppner, T.W., 2011. *Operating Systems In Depth*. Hoboken, N.J.: Wiley, pp.107, 218, 226, 236,.

**ANDREW S. TANENBAUM., A. S.**

**Book:** Modern operating systems

Andrew S. Tanenbaum., A., 1992. *Modern Operating Systems*. England.

**B, H. AND MALAKHOV, A.**

**Website:** How does the stack work in assembly language?

B, H. and Malakhov, A., 2009. *How Does The Stack Work In Assembly Language?*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/556714/how-does-the-stack-work-in-assembly-language> [Accessed 8 December 2020].

**R, T., URA, A. AND KHAN, W.**

**Website:** Explain the concept of a stack frame in a nutshell

R, T., Ura, A. and Khan, W., 2012. *Explain The Concept Of A Stack Frame In A Nutshell*. [online] Stack Overflow. Available at: <https://stackoverflow.com/questions/10057443/explain-the-concept-of-a-stack-frame-in-a-nutshell> [Accessed 19 December 2020].

**GEEKSFORGEEKS**

**Website:** BUDDY SYSTEM - MEMORY ALLOCATION TECHNIQUE

GeeksforGeeks. 2019. *Buddy System - Memory Allocation Technique - Geeksforgeeks*. [online] Available at: <https://www.geeksforgeeks.org/buddy-system-memory-allocation-technique/> [Accessed 27 December 2020].

**Research material for code**

**S240: DATA STRUCTURES & ALGORITHMS I**

**Website:** Hashing

Cpp.edu. 2011. *CS240: Data Structures & Algorithms I*. [online] Available at: <https://www.cpp.edu/~ftang/courses/CS240/lectures/hashing.htm> [Accessed 29 December 2020].