Cuckoo Hashing Analysis

Joseph Garro (jmg289@uakron.edu)

Abstract:

Cuckoo hashing is a method of resolving hash collisions implemented in dictionary based data structures. With cuckoo hashing, two hash tables with unique hash functions are used to store data. When a value is added to the dictionary, it is hashed to the first hash table at an index determined by that tables' hash function. If an element already exists in that index in the hash table, it is displaced and hashed into the second hash table. Since a given element can only be in one hash table, direct element access is possible using a tables' unique hash function. This allows constant time deletions and searches are achieved. Insertions have an amortized constant time complexity but are sneakily costly due requiring the dictionary to be rehashed and its size potentially expanded if it cannot successfully store all values. As shown by this research, cuckoo hashing is plagued by wasted space despite its appealing advertised time complexities. It is particularly inefficient at using the space allocated to it, creating a large and highly favored primary hash table and an equally sized secondary hash table with little to no utilization.

Introduction:

The purpose of this research is to determine the effectiveness of cuckoo hashing as a method of resolving hash collisions. To conduct this research, a dictionary data structure implementing cuckoo hashing was created and compared with other data structures. Cuckoo hashing is believed to result in constant time complexities for search and deletion operations and an amortized constant time complexity for insertions. However, an issue with hash based data structures is that they present a wide range of tradeoffs. For instance, hash tables are not guaranteed to be completely full, so more space is allocated than required. If hash tables store buckets, the implementation of the buckets is important as a hardcoded bucket size has the same potential for wasted space or not enough space to store all elements. Additionally, buckets implemented as a linked list experience a performance versus space tradeoff as they can grow and shrink as needed, but there is overhead incurred in implementing it. Furthermore, a linear time complexity exists when verifying that a given value resides in the bucket and accessing it if it does. Evidently, hash based structures are not a perfect solution due to the battle between performance and space to implement. Hence, the goal of this research is to determine whether a dictionary using cuckoo hashing behaves as expected, achieving a constant time deletion and search as well as an amortized constant time insertion. The space complexity associated with a dictionary utilizing cuckoo hashing is also determined, as well as the amount of space wasted by this implementation. These results are compared to the AVL tree created for Project 2 and the STL vector as they offer comparable time complexities to cuckoo hashing for some operations, allowing the best structure to be determined based qualitative measurements. To test the data structures, driver code was created to measure the run time of each operation when performed on each data structure relative to a set problem size N. Additionally, the cuckoo hashing implementation had one of its key behaviors varied, the method of increasing the size of the hash tables during a rehash, to see if any observable impact existed in the time to insert elements into the dictionary and the amount of space wasted as a result. Finally, sufficient information was gathered about the performance cuckoo hashing to determine under what situations it is an effective solution to solve a problem relative to size, time, and performance constraints, as well as the potential drawbacks associated with such a solution.

Discussion:
Hypothesis: A dictionary that implements cuckoo hashing to handle hash collisions is an ideal data structure for performing fast searches and deletions. Insertion performance sufferers due to the potential of cycles requiring rehashes of both hash tables to resolve, furthered by the potential of requiring multiple rehashes to resolve. Additionally, the potential for wasted space is high due to the constraint that each hash table must be at least 50% empty, so a minimum of half the space allocated for the data structure will be unused at any given time. Therefore, there exists a point at which cuckoo hashing is an unfit solution to a given problem due to the massive amount of space wasted. Performance wise, constant time searches and deletions are expected. Insertions should be worse performing, although an amortized constant time complexity should be observed in the results.

Cuckoo hashing is a method of handling hash collisions in a dictionary based data structure developed by Rasmus Pagh and Flemming Rodler (Pagh 2001). Compared to traditional hash collision resolution policies such as open hashing and bucket hashing, cuckoo hashing utilizes 2 independent hash tables (arrays) t_1 and t_2, and 2 hash functions, $h_1(x)$ and $h_2(x)$. Both hash tables are of size s and each hash function is unique to t_1 or t_2, respectively. A given value, q, is first inserted into t_1 according to an index generated by $h_1(q)$. Thus, $t_1[h_1(q)] = q$. Due to the possibility of hash collisions, the index $t_1[h_1(q)]$ can already be occupied with a value, such as $t_1[h_1(q)] = r$. In this event, the value r is evicted from its location in t_1, q is inserted in its place, and r is hashed into t_2 at index $h_2(r)$. This is shown below in Figures 1 through 3:

Suppose t_1 and t_2 are of size 4 and initially empty.

$h_1(x) = x\%4$, $h_2(x) = (x/4)\%4$; x = 4

$h_1(4) = 0$: $t_1[0] = 4$

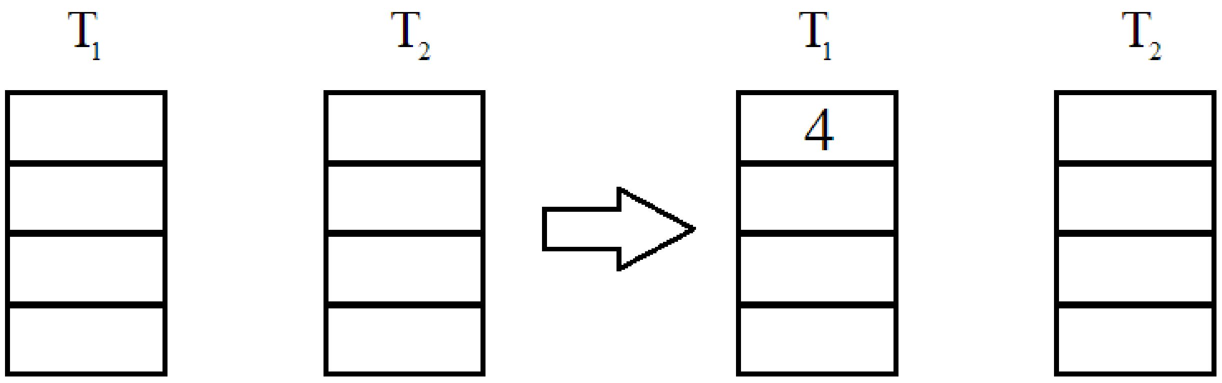


Figure 1: Inserting a value of x = 4 into the hash tables where the table size s = 4, $h_1(x) = x\%4$, $h_2(x) = (x/4)\%4$

Now suppose x = 16 is inserted into the same dictionary:

$h_1(16) = 0$, but $t_1[0]$ is occupied : Solution: evict $t_1[0]$ and hash into t_2

$t_1[0] = 4$, $h_2(4) = 0$: $t_1[0] = 16$, $t_2[0] = 4$

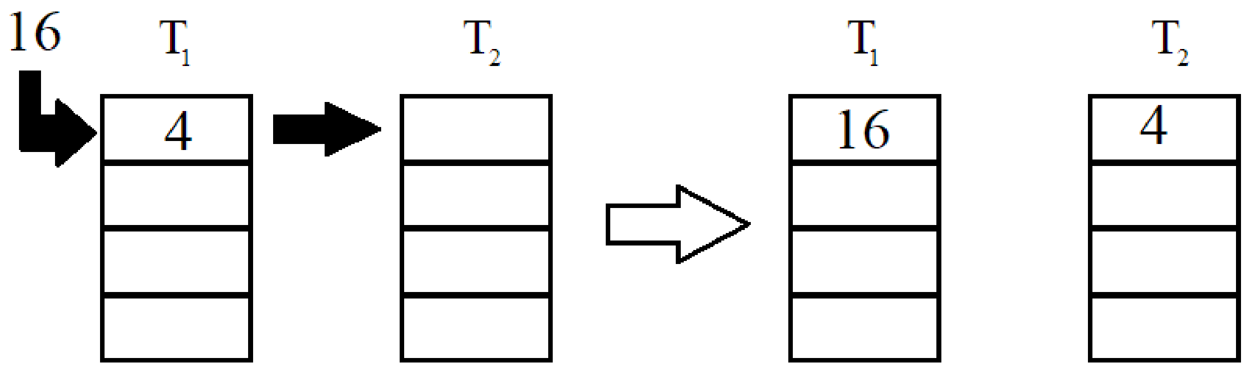


Figure 2: Inserting a value of x = 16 into the hash tables shown in Figure 1

Finally, suppose x = 0 is inserted into the same dictionary:

$h_1(0) = 0$, but $t_1[0]$ is occupied with 16	:	Solution: evict $t_1[0]$ and hash into t_2
$t_1[0] = 16$, $h_2(16) = 0$; $t_2[0]$ is occupied with 4	:	Solution: evict $t_2[0]$ and hash into t_1
$t_2[0] = 4$, $h_1(4) = 0$; $t_1[0]$ is occupied with 0	:	Solution: evict $t_1[0]$ and hash into t_2
$t_1[0] = 0$, $h_2(0) = 0$; $t_2[0]$ is occupied with 16	:	Solution: evict $t_2[0]$ and hash into t_1
$t_2[0] = 16$, $h_1(16) = 0$; $t_1[0]$ is occupied with 4	:	Solution: evict $t_1[0]$ and hash into t_2
$t_1[0] = 4$, $h_2(4) = 0$; $t_2[0]$ is occupied with 0	:	Solution: evict $t_1[0]$ and hash into t_2
⋮		⋮

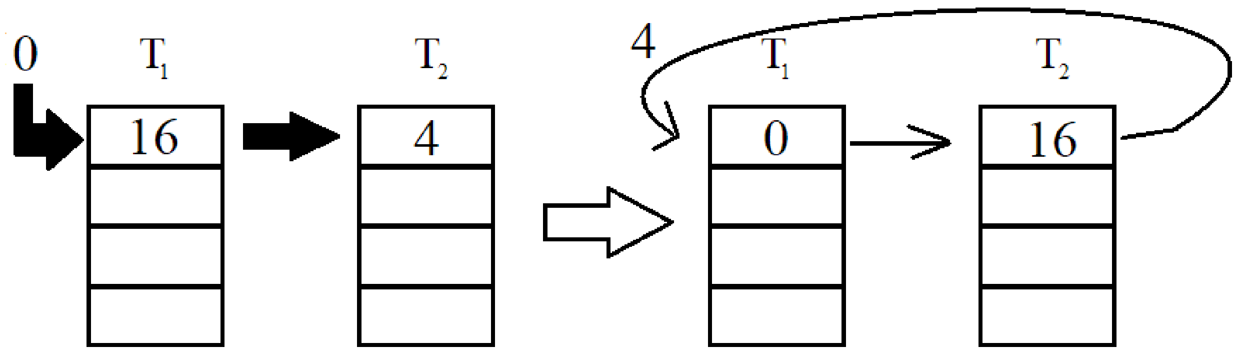


Figure 3: Inserting a value of x = 0 into the hash tables shown in Figure 2, resulting in an infinite loop of evictions

As shown in Figure 3, it is possible to enter an infinite loop of evictions, known as a cycle, provided enough input hashes to the same set of indices. It is important to note than a given insertion may not cause a cycle, instead resulting in so many evictions occurring that the two instances are synonymous. Hence, a cycle is said to occur at a predetermined number of evictions, defined as $log_2(s)$ from lecture. The cuckoo hashing implementation defined by Pagh and Rodler includes two methods to escape a cycle, which are rehashing both hash tables with new hash functions and increasing the size of both tables to reduce the probability of future collisions.

An additional constraint imposed by Pagh and Rodler is that each hash table utilized in the dictionary must remain at least half empty or have a load of less than 50%. Some variants of cuckoo hashing seek to increase the load by adding more hash tables, such as one study discovering that the use of 3 hash tables allowed for a load of up to 92% (Mitzenmacher, 2009) and another discovering that converting each index to a bucket capable of storing 2 values resulted in an acceptable load of 80% (Dietzfelbinger, 2017). However, these variations are not discussed, and the implementation of cuckoo hashing described by Pagh and Rodler is the focus of this research, so the load of each hash table was restricted to a maximum of 50%.

The appeal of using a dictionary data structure with cuckoo hashing is that as the two hash tables store individual keys and not buckets, a value in the dictionary is either in the first hash table or the second hash table, at an index specified by that hash tables corresponding hash function. This requires only two locations to be checked to determine if the value is in the dictionary. To check either location, the hash function used by the hash table is applied to the value that is being searched for and the result is used as in index into the appropriate hash table. Checking the value of an array at a specific index is direct element access - a constant time operation. Furthermore, the worst case for a search is that the value is not in either hash table, requiring two constant time operations to determine. Therefore, the worst case scenario is that two direct element accesses need to be performed, resulting in time complexity of O(1) for searches. Deletions perform similarly, as the value is searched for at its two possible locations and removed if it was found, resulting in the same time complexity of O(1).

Insertions with cuckoo hashing are more complicated to analyze due to the possibility of multiple evictions occurring as a result of the insertion, the potential for cycles, and the overhead associated with rehashing the data structure to handle them. As stated in a lecture by 2014 Data Structures course at Stanford University, the goal is to cause insertions to result in multiple cycles and rehashes so infrequently that the cost associated with them is spread out to obtain an amortized time complexity of O(1) as well (Schwarz, 2014).

Code Analysis:
As discussed above, cuckoo hashing is a method of handling hash collisions in a dictionary based data structure. Its defining features according to Pagh and Rodler are the use of 2 independent hash tables that must be at least 50% empty, their unique hash functions, elements being evicted from one table to another whenever a hash collision occurs, and the ability to rehash both tables with the elements contained in the dictionary at that instance in time. Beyond these key features, cuckoo hashing is largely implementation defined and the programmer can implement it in a way that suits their needs. Thus, cuckoo hashing is a general problem solving strategy that can be implemented in a variety of ways rather than a specific solution with a fixed approach.

The implementation of cuckoo hashing for this research closely follows Pagh and Rodler's approach. The dictionary based data structure was designed to hold integer values and contains two separate hash tables. Each hash table has a unique hash function, and insertions always occur into the first hash table. If the index in the first hash table is occupied, the existing value is replaced with the incoming value and rehashed into the second hash table. When inserting values, a cycle occurs if there are more evictions than $log_2(s)$, where s is the size of either table. When a cycle occurs, the dictionary is rehashed until the cycle is resolved.

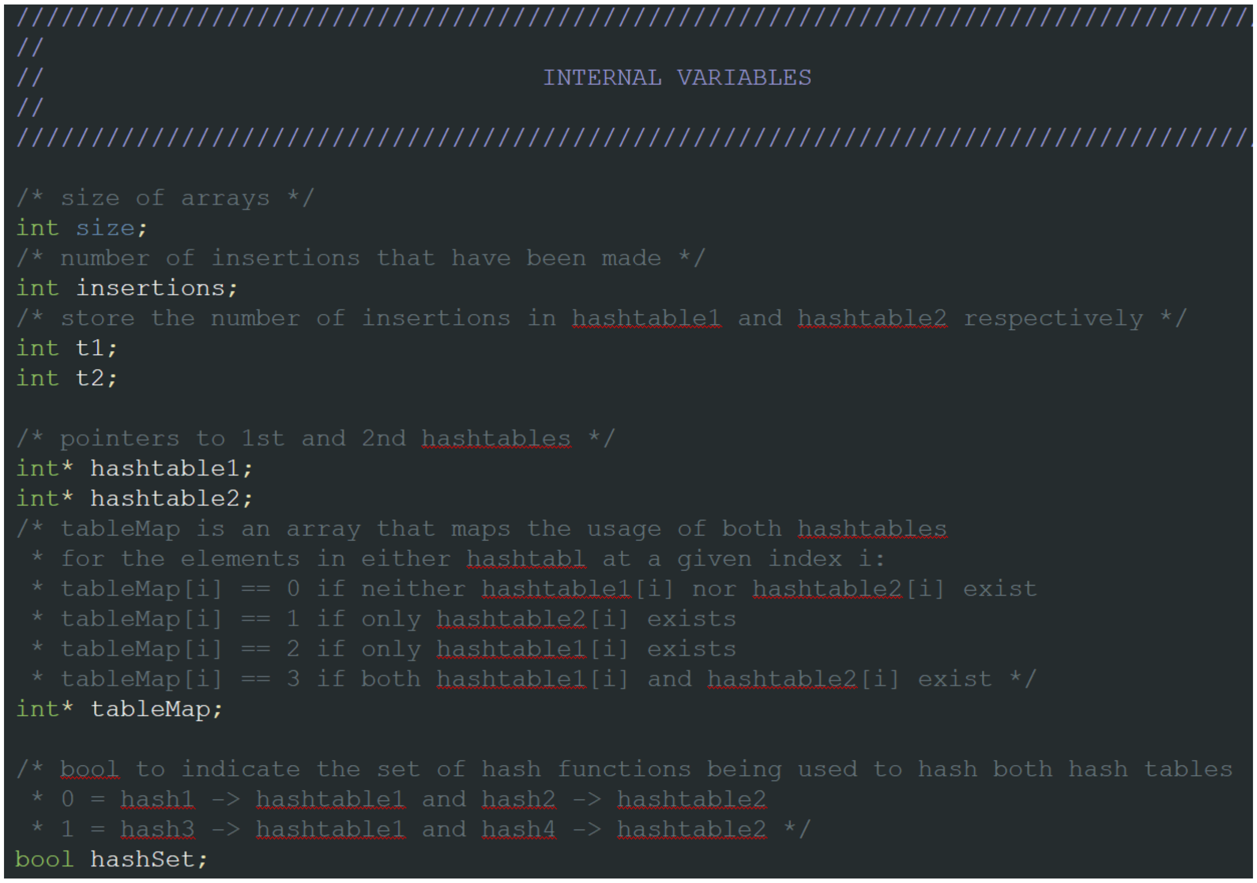
To escape a cycle, Pagh and Rodler required their cuckoo hashing implementation to rehash the dictionary using an alternate pair of hash functions and have the capability to increase the size of both hash tables. The cuckoo hashing implementation created for this research does this in two steps that are repeated until the cycle is resolved:

1. Rehash both hash tables using their alternate hash functions
    a. If no cycle is detected during reconstruction, a valid dictionary was created
2. If a cycle is detected, increase the size of the hash tables and rehash with the original hash functions
    a. If a cycle is detected, repeat the process starting with step 1

The idea is that when a cycle is experienced, the dictionary avoids the unnecessary allocation of space by creating new hash tables of a larger size only if both pairs of hash functions result in a cycle. This produces a "second chance algorithm" where each table has two opportunities to create a valid table at its current size, being hashed with the original hash function and then the alternate hash function. If both pairs of hash functions result in a cycle, the hash table is reallocated with a larger size and the data is attempted to be reinserted successfully. This works as if the tables were hashed with using their alternate hash functions and a cycle resulted, the original hash functions must result in a cycle on the same set of data otherwise the alternate hash functions would not be in use. Therefore, the only way to proceed is to create larger hash tables as both sets of hash functions currently fail to produce a valid dictionary, providing each hash table with 2 chances to succeed before it is recreated. This method of rehashing was chosen to avoid repeatedly allocating new space for each hash table and hashing values into them, which is an expensive set of operations. Instead, the creation of larger hash tables is reserved for when it is necessary to do so.

The class that was created to implement a dictionary using the interpretation of cuckoo hashing defined above is called "cuckooHashtable". As mentioned previously, this class was primarily based on the research paper by Pagh and Rodler with additional sources of information such as the lecture "Hashing Part 4" (Crissey Renwald) and the lecture notes from Stanford consulted as well. The class is configured to accept unique positive integer values as input, making it a simple example of cuckoo hashing. The class checks to ensure that any input meets these 2 criteria, and any invalid input is discarded and not entered into the dictionary.

The cuckooHashtable class contains 8 variables that enable it to function, shown in Figure 4.



```
////////////////////////////////////////////////////////////////////////////////
//
//                                   INTERNAL VARIABLES
//
////////////////////////////////////////////////////////////////////////////////

/* size of arrays */
int size;
/* number of insertions that have been made */
int insertions;
/* store the number of insertions in hashtable1 and hashtable2 respectively */
int t1;
int t2;

/* pointers to 1st and 2nd hashtables */
int* hashtable1;
int* hashtable2;
/* tableMap is an array that maps the usage of both hashtables
 * for the elements in either hashtabl at a given index i:
 * tableMap[i] == 0 if neither hashtable1[i] nor hashtable2[i] exist
 * tableMap[i] == 1 if only hashtable2[i] exists
 * tableMap[i] == 2 if only hashtable1[i] exists
 * tableMap[i] == 3 if both hashtable1[i] and hashtable2[i] exist */
int* tableMap;

/* bool to indicate the set of hash functions being used to hash both hash tables
 * 0 = hash1 -> hashtable1 and hash2 -> hashtable2
 * 1 = hash3 -> hashtable1 and hash4 -> hashtable2 */
bool hashSet;
```

Figure 4: Internal variables composing cuckooHashtable class

The first 2 variables are integers used to store the size of each hash table and total number of insertions made into the instance of the object. The integer variables t1 and t2 correspond to the total number of insertions made into the first hash table and the second hash table, so their sum is equal to the total number of insertions. The next 3 variables are pointers to integer arrays which compose the 2 hash tables used by the dictionary to store elements and a supporting array that stores information about the insertions. All arrays were implemented as pointers so that their size is never hardcoded in the class definition, allowing space for each to grow dynamically as more elements are added.

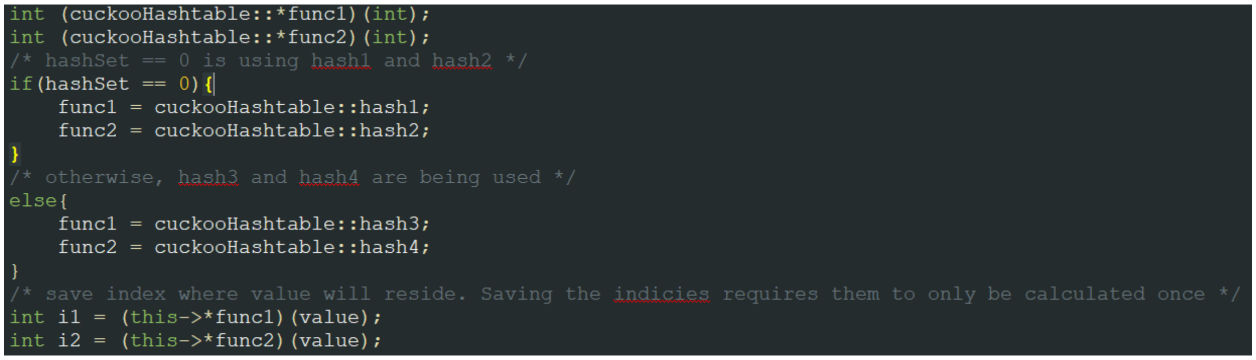
The supporting array, tableMap, is the same size as the two hash tables and uses a single integer variable that represents the state of each corresponding hash table index. In tableMap, each index stores a value from 0 to 3 that reflects whether a value is stored in the same index in hashtable1 or hashtable2. The values have the following meaning:

0 = neither hashtable1 nor hashtable2 is storing a value at this index
1 = only hashtable2 is storing a value at this index
2 = only hashtable1 is storing a value at this index
3 = both hashtable1 and hashtable2 are storing values at this index

TableMap is essential to the operation of the cuckooHashtable class as it allows values to be easily tracked across both hash tables, providing a mechanism to differentiate genuine input from the default initialization of both hash tables, to track the number of values in each hash table, and to look ahead and determine if an insertion will result in an eviction. The use of an array to track this was necessary as it is possible for a value to be inserted that is the same as the default initialization of each hash table, requiring a way to distinguish the two. The ability to detect evictions before they occurred was beneficial as a function could be created to handle insertions into a given hash table and change its behavior based on whether eviction will occur. An additional benefit was that simplified the rehashing process as it created a convenient way to parse both hash tables and temporarily store all genuine input when rehashing them by copying values from either hash table based on the corresponding tableMap entry. The inclusion of tableMap results in the cuckooHashtable class needed more space due to having a third array that is the same size as the hash tables, but the performance benefits and simplification of essential behavior that it provides outweigh the cost.

The final variable is a bool called hashSet which is used to indicate the pair of hash functions that are currently being used to hash input. A value of 0 indicates that the original hash functions are being used while a value of 1 indicates that the pair of alternate hash functions are being used. The hash functions used by each member function in the cuckooHashtable class are not hardcoded to avoid instances where the pair of hash functions in use change, so this variable allows each member function to dynamically change the hash functions it uses accordingly. For example, a search member function that was programmed to use the original pair of hash functions would not return the proper output on a cuckooHashtable object currently using the alternate pair of hash functions. This variable ensures that each function can modify the hashing function it uses to generate the expected output. As a result, applicable member functions were programmed once as the hashSet variable could be checked to change the hash functions used inside a given member function. This was achieved using function pointers to member functions, allowing the functions to use a pointer that points to the desired hashing function.

A guide to create function pointers to class member functions was found on a Stack Overflow post (Heto, 2017) and adapted to fit the cuckooHashtable class. The use of the member function pointers is shown in the sample code blow, taken from the “find” member function.

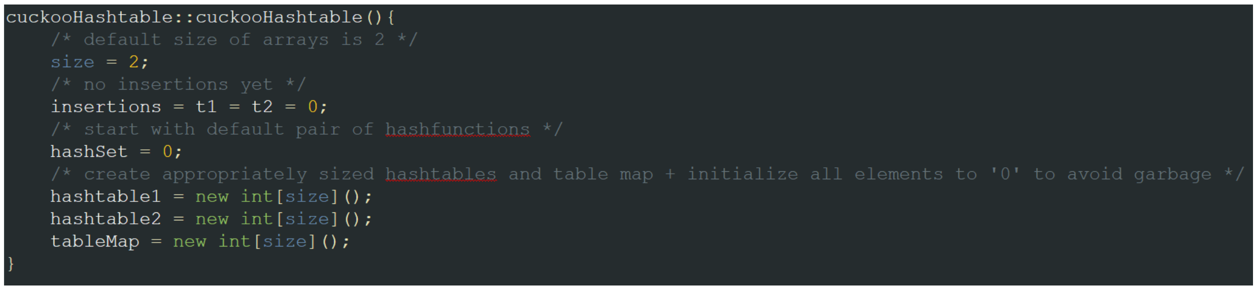


```
int (cuckooHashtable::*func1)(int);
int (cuckooHashtable::*func2)(int);
/* hashSet == 0 is using hash1 and hash2 */
if(hashSet == 0){
    func1 = cuckooHashtable::hash1;
    func2 = cuckooHashtable::hash2;
}
/* otherwise, hash3 and hash4 are being used */
else{
    func1 = cuckooHashtable::hash3;
    func2 = cuckooHashtable::hash4;
}
/* save index where value will reside. Saving the indicies requires them to only be calculated once */
int i1 = (this->*func1)(value);
int i2 = (this->*func2)(value);
```

Figure 5: Sample code from the find member function demonstrating how the hashSet variable and function pointer to member functions allowed a member function to dynamically change the hash functions used

As shown in Figure 5, two pointers to member functions of the cuckooHashtable class were created. Based on the value of hashSet, the member function pointers were loaded with the appropriate hash functions from the cuckooHashtable class. As this code is from the find member function, two indices were generated using the function pointers to determine the possible location of a value in each hash table. This resulted in a general find function with a lesser cost than a find function using all 4 hash functions to generate 4 indices that must be checked. As demonstrated, the use of the hashSet variable and the ability to make function pointers to member functions allowed the code for each function to become a “template” in which the proper hash function(s) could be loaded into at run time. This simplified the creation of the cuckooHashtable class by combining multiple case specific member functions into single general purpose functions, reducing the amount of code to implement the class and the number of member functions that needed to be kept track of.

To use the cuckooHashtable class, a cuckooHashtable object is created with the default constructor shown in Figure 6.



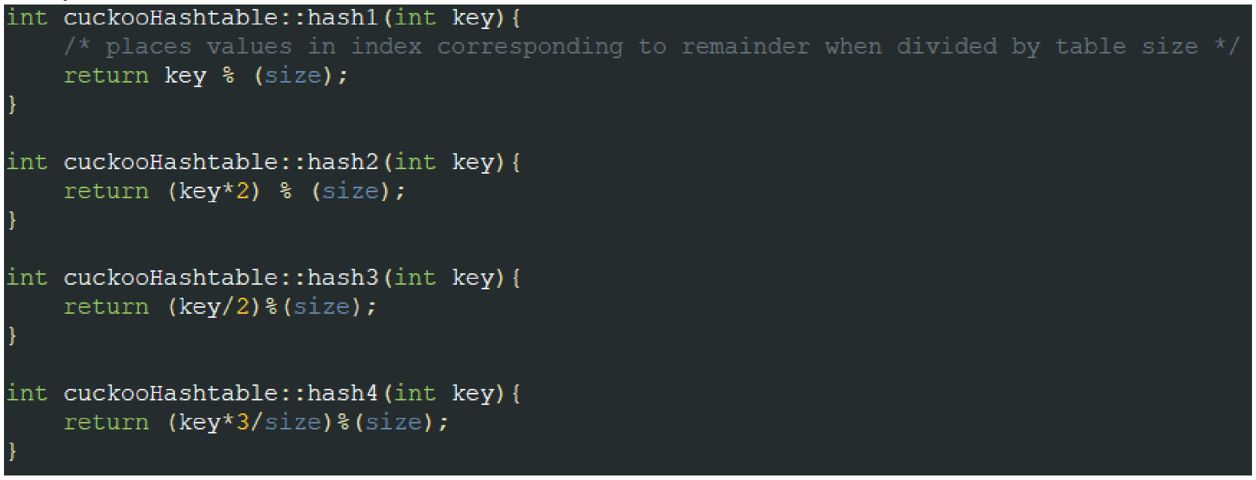
```
cuckooHashtable::cuckooHashtable(){
    /* default size of arrays is 2 */
    size = 2;
    /* no insertions yet */
    insertions = t1 = t2 = 0;
    /* start with default pair of hashfunctions */
    hashSet = 0;
    /* create appropriately sized hashtables and table map + initialize all elements to '0' to avoid garbage */
    hashtable1 = new int[size]();
    hashtable2 = new int[size]();
    tableMap = new int[size]();
}
```

Figure 6: Default constructor code

The default constructor sets the size of the object to ‘2’ and creates three arrays to represent the two hash tables and the tableMap. The three arrays are dynamically created and the use of parenthesis following “new int[size]” is used to initialize the contents of each array to ‘0’. As values are added to the dictionary, the size of the hash tables increases due to evictions causing rehashes. This allows the dictionary to function more like a vector as its size is always expanding and the user does not need to set a limit on the number of values that can be input. The reasoning for using the default constructor is because each hash table must remain at least 50% empty, there is no way to know the size required to implement the hash tables without advance knowledge of the input and the hash functions. Even if these are known, it is not guaranteed that setting the size to two times the number of input values will prevent needing to allocate more space to each table, as cycles can occur that are only fixed through the allocation of larger hash tables. Therefore, it was decided that it was better to absorb this cost and allow the dictionary to grow without an upper limit as allocating more space is likely unavoidable and this scheme provides greater functionality to users of the class.

Concerning the other key member functions in the cuckooHashtable class, the following section outlines them and determines their time complexity with respect to the problem size n.

Analysis of hash functions:

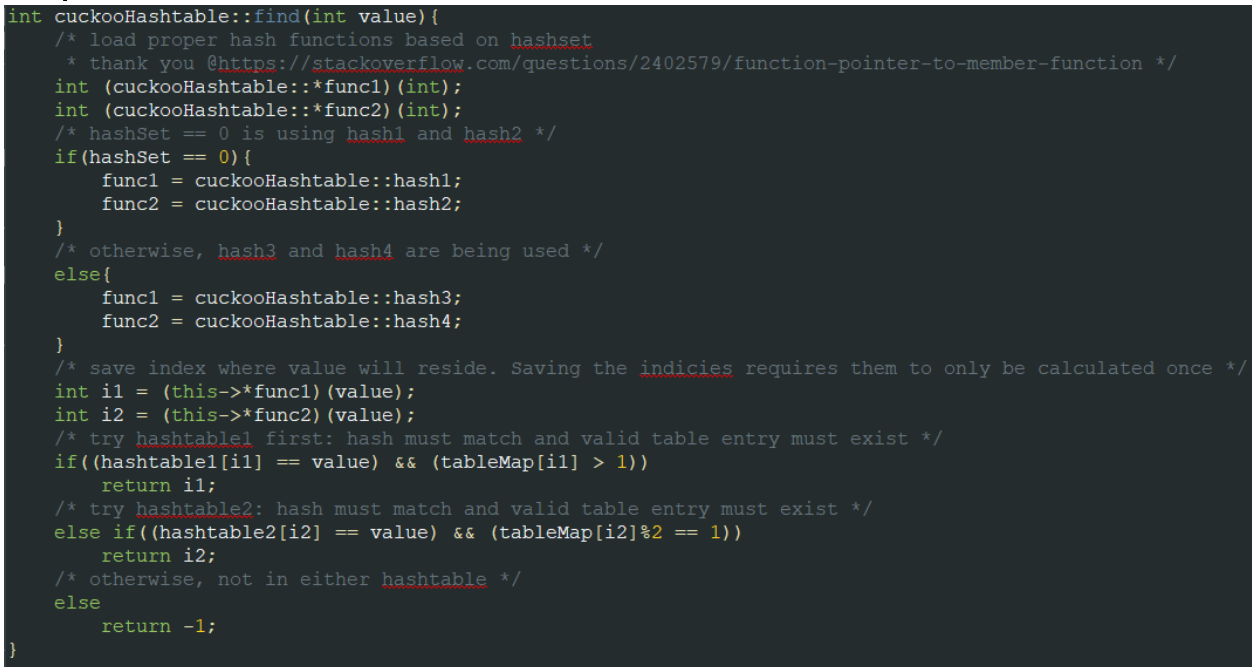


```
int cuckooHashtable::hash1(int key){
    /* places values in index corresponding to remainder when divided by table size */
    return key % (size);
}

int cuckooHashtable::hash2(int key){
    return (key*2) % (size);
}

int cuckooHashtable::hash3(int key){
    return (key/2)%(size);
}

int cuckooHashtable::hash4(int key){
    return (key*3/size)%(size);
}
```

Figure 7: Code to implement the 4 hash functions in the cuckooHashtable class

The cuckooHashtable class has 4 hash functions that allow the dictionary to work. As stated previously, hash1 and hash2 are the original hash functions and correspond to a hashSet value of 0, while a hashSet value of 1 corresponds to the alternate hash functions, hash3 and hash4. Each hash function is performed in constant time => O(1).

Analysis of the find function:

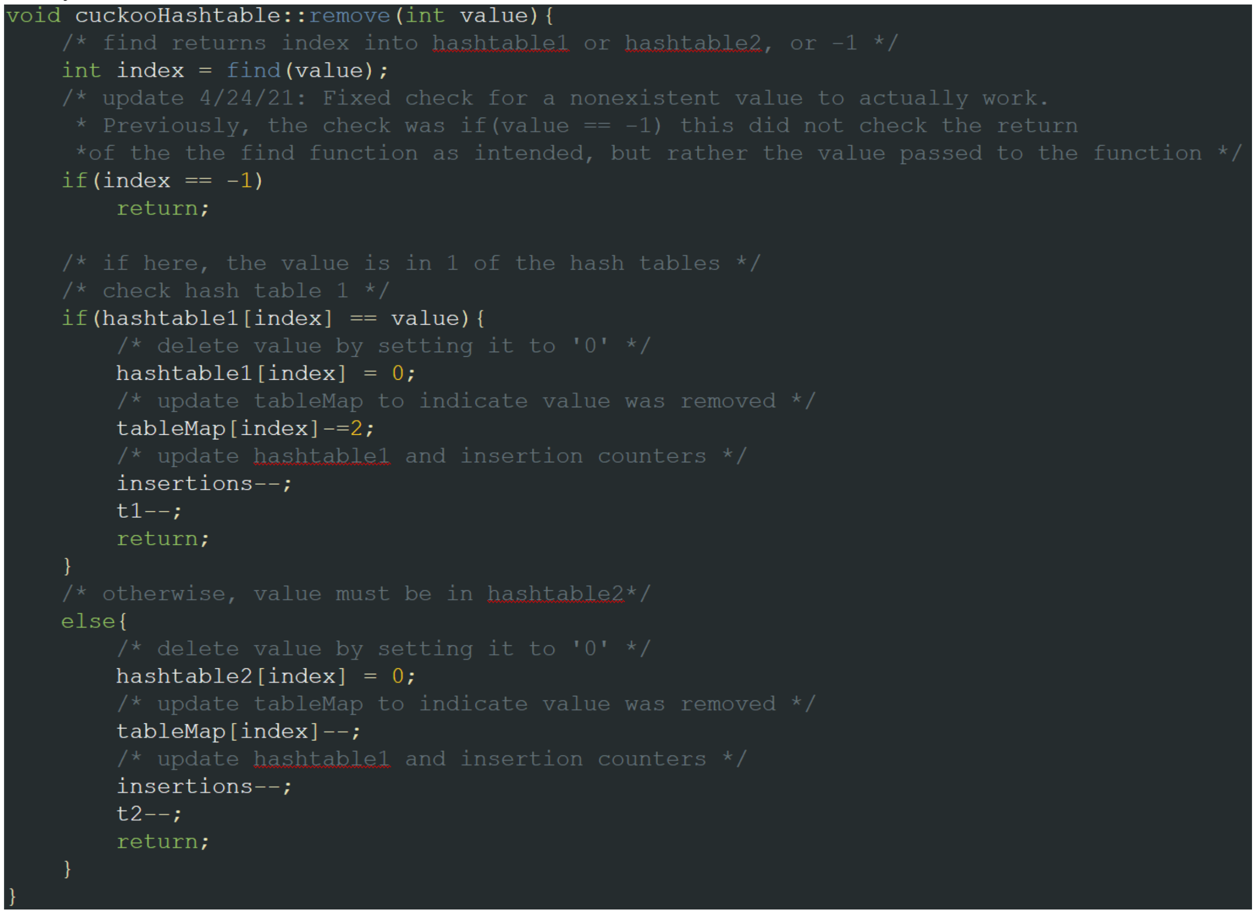


```
int cuckooHashtable::find(int value){
    /* load proper hash functions based on hashset
     * thank you @https://stackoverflow.com/questions/2402579/function-pointer-to-member-function */
    int (cuckooHashtable::*func1)(int);
    int (cuckooHashtable::*func2)(int);
    /* hashSet == 0 is using hash1 and hash2 */
    if(hashSet == 0){
        func1 = cuckooHashtable::hash1;
        func2 = cuckooHashtable::hash2;
    }
    /* otherwise, hash3 and hash4 are being used */
    else{
        func1 = cuckooHashtable::hash3;
        func2 = cuckooHashtable::hash4;
    }
    /* save index where value will reside. Saving the indicies requires them to only be calculated once */
    int i1 = (this->*func1)(value);
    int i2 = (this->*func2)(value);
    /* try hashtable1 first: hash must match and valid table entry must exist */
    if((hashtable1[i1] == value) && (tableMap[i1] > 1))
        return i1;
    /* try hashtable2: hash must match and valid table entry must exist */
    else if((hashtable2[i2] == value) && (tableMap[i2]%2 == 1))
        return i2;
    /* otherwise, not in either hashtable */
    else
        return -1;
}
```

Figure 8: Code to implement the find function in the cuckooHashtable class

The find function takes an integer argument that it attempts to locate in the cuckooHashtable object. Based on the value of hashSet, the appropriate hash functions are loaded into member function pointers and the potential location of the value in both hash tables is computed. If the value is found, the index in the hash table is returned, but the hash table itself is not specified. If no value is found, -1 is returned instead. In the worst case scenario, the value is either in hashtable2 or not in the hash table at all, requiring 2 indices be generated based off the appropriate hash functions and the corresponding locations in each hash table checked. This can be accomplished in constant time -> O(1).

Analysis of the remove function:



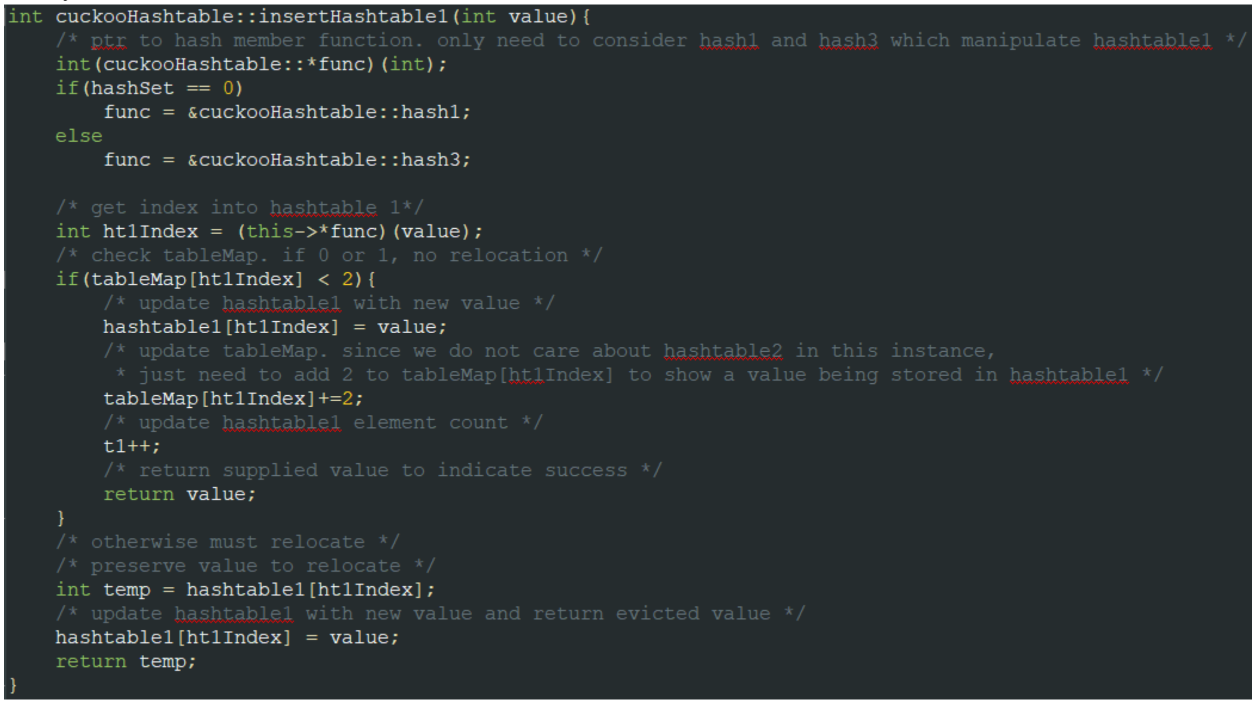
```
void cuckooHashtable::remove(int value){
    /* find returns index into hashtable1 or hashtable2, or -1 */
    int index = find(value);
    /* update 4/24/21: Fixed check for a nonexistent value to actually work.
     * Previously, the check was if(value == -1) this did not check the return
     *of the the find function as intended, but rather the value passed to the function */
    if(index == -1)
        return;

    /* if here, the value is in 1 of the hash tables */
    /* check hash table 1 */
    if(hashtable1[index] == value){
        /* delete value by setting it to '0' */
        hashtable1[index] = 0;
        /* update tableMap to indicate value was removed */
        tableMap[index]-=2;
        /* update hashtable1 and insertion counters */
        insertions--;
        t1--;
        return;
    }
    /* otherwise, value must be in hashtable2*/
    else{
        /* delete value by setting it to '0' */
        hashtable2[index] = 0;
        /* update tableMap to indicate value was removed */
        tableMap[index]--;
        /* update hashtable1 and insertion counters */
        insertions--;
        t2--;
        return;
    }
}
```

Figure 9: Code to implement the remove function in the cuckooHashtable class

The remove function accepts an integer that it removes from the dictionary if it exists. To check if the value exists, the find member function is used[1]. Depending on its return value, the remove function stops or checks both hash tables at the returned index. Once found, the value is deleted by being set to ‘0’, the corresponding index in the tableMap updates, and the total number of insertions and the insertions made into the specific hash table decremented. The worst case scenario is that the value is in the second hash table, which requires 3 constant time operations to determine. Once determined, 4 additional constant time operations take place, but as all operations occur in constant time, the remove function takes constant time => O(1).

[1] This function initially deleted elements that did not exist in the dictionary as “value” was accidentally checked instead of “index”, which stores the result of the “find” function. As all tests were generated using unique input, this bug remained unnoticed until discovered 4/24/21 by Dimitry Melnikov. Updating the code does not change the results of the data structure documented in this research paper, as the tests were created to ensure unique input. This change is just to make the cucukoHashtable class operate as intended.

Analysis of the insertHashtable function:

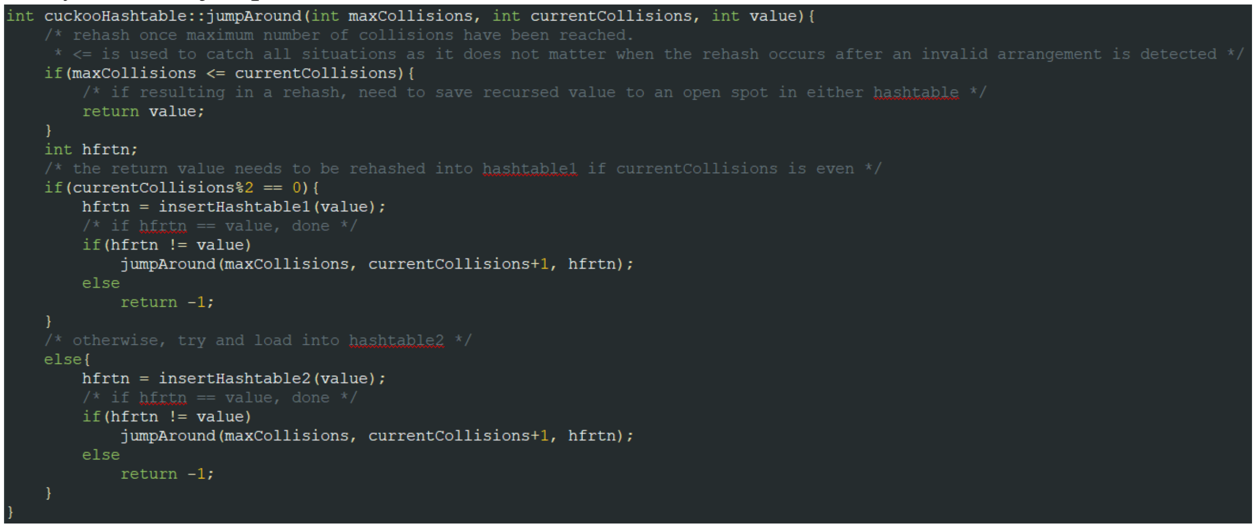


```
int cuckooHashtable::insertHashtable1(int value){
    /* ptr to hash member function. only need to consider hash1 and hash3 which manipulate hashtable1 */
    int(cuckooHashtable::*func)(int);
    if(hashSet == 0)
        func = &cuckooHashtable::hash1;
    else
        func = &cuckooHashtable::hash3;

    /* get index into hashtable 1*/
    int ht1Index = (this->*func)(value);
    /* check tableMap. if 0 or 1, no relocation */
    if(tableMap[ht1Index] < 2){
        /* update hashtable1 with new value */
        hashtable1[ht1Index] = value;
        /* update tableMap. since we do not care about hashtable2 in this instance,
         * just need to add 2 to tableMap[ht1Index] to show a value being stored in hashtable1 */
        tableMap[ht1Index]+=2;
        /* update hashtable1 element count */
        t1++;
        /* return supplied value to indicate success */
        return value;
    }
    /* otherwise must relocate */
    /* preserve value to relocate */
    int temp = hashtable1[ht1Index];
    /* update hashtable1 with new value and return evicted value */
    hashtable1[ht1Index] = value;
    return temp;
}
```

Figure 10: Code to implement the insertHashTable1 function in the cuckooHashtable class

Two insertHashTable functions exist, specific to each hash table. The code above is for the function specific to hashtable1, but the two functions are identical besides the table they modify. Each function accepts an integer parameter and the appropriate hash functions are determined based on the value of hashSet. The tableMap is checked to determine if an eviction will result from the insertion, returning the evicted value or the value that was just inserted if not. As no duplicate values can be inserted into the dictionary, the return of this function can be checked to determine if the insertion caused an eviction or not. As this function only inserts the input, updates the tableMap, and returns the input or the evicted value, it takes constant time => O(1).

Analysis of the jumpAround function:

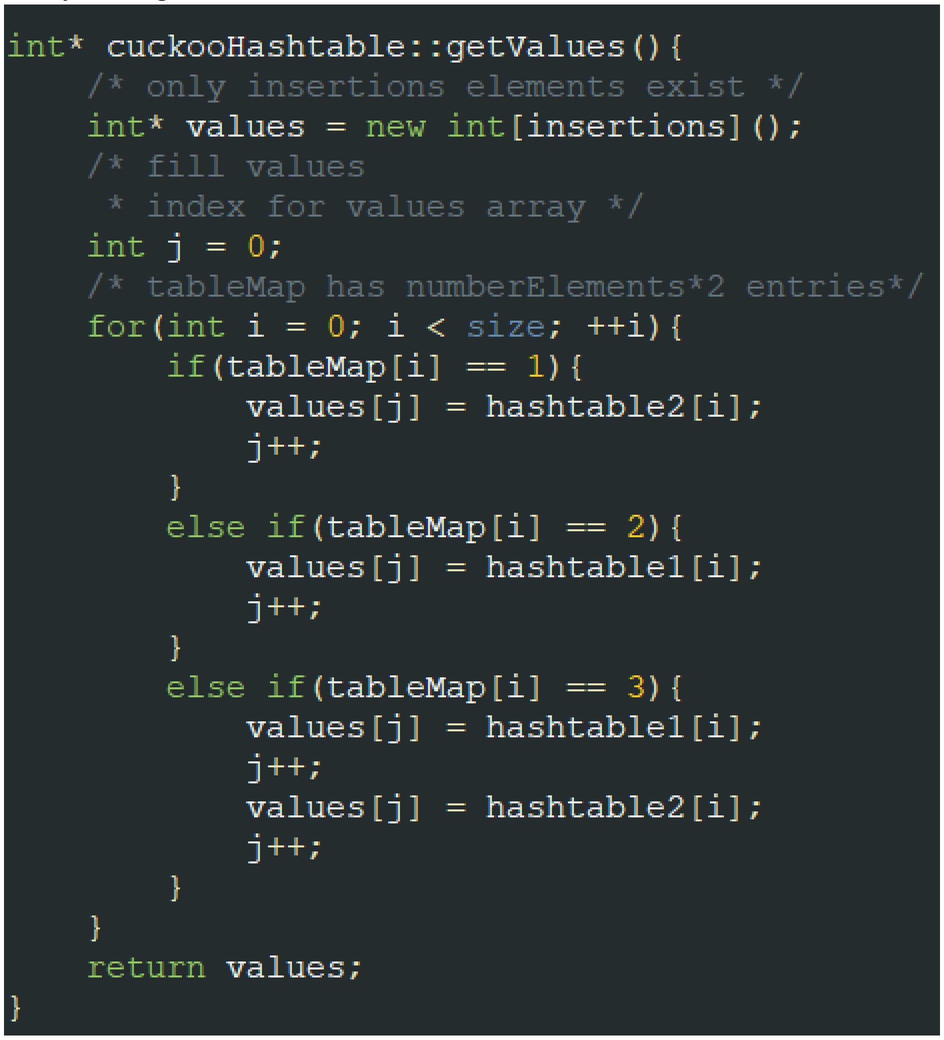


```
int cuckooHashtable::jumpAround(int maxCollisions, int currentCollisions, int value){
    /* rehash once maximum number of collisions have been reached.
     * <= is used to catch all situations as it does not matter when the rehash occurs after an invalid arrangement is detected */
    if(maxCollisions <= currentCollisions){
        /* if resulting in a rehash, need to save recursed value to an open spot in either hashtable */
        return value;
    }
    int hfrtn;
    /* the return value needs to be rehashed into hashtable1 if currentCollisions is even */
    if(currentCollisions%2 == 0){
        hfrtn = insertHashtable1(value);
        /* if hfrtn == value, done */
        if(hfrtn != value)
            jumpAround(maxCollisions, currentCollisions+1, hfrtn);
        else
            return -1;
    }
    /* otherwise, try and load into hashtable2 */
    else{
        hfrtn = insertHashtable2(value);
        /* if hfrtn == value, done */
        if(hfrtn != value)
            jumpAround(maxCollisions, currentCollisions+1, hfrtn);
        else
            return -1;
    }
}
```

Figure 11: Code to implement the jumpAround function in the cuckooHashtable class

The jumpAround function is responsible for handling insertions, the evictions that can arise, and signaling the need to rehash the dictionary. The jumpAround function accepts 3 integer arguments. The first is the maximum number of collisions that can occur, the second is the current number of collisions, and the third is the value to insert into the dictionary. The jumpAround function is recursively called until a value is successfully inserted, or the number of collisions exceeds the maximum number of collisions allowed and causes a cycle. Based on the value of current collisions, the function knows whether the value argument is to be inserted into hashtable1 or hashtable2. This is as hashtable1 is the location of the first insertion, so an eviction that results of an insertion here would be the first eviction. If the eviction from hashtable1 causes another eviction when hashed into hashtable2, a second eviction occurs. As this cycle repeats itself until the evictions stop or the threshold is reached, even values of collisions correspond to insertions into hashtable1 and odd values to hashtable2. Thus, the evicted value is attempted to be inserted into the appropriate hash table. A return value of 1 indicates that the eviction was rehashed without causing another eviction while a return value of anything else indicates that a cycle occurred, and the return value is the value that was evicted last. Here, the worst case scenario is that the value is returned due to the maximum number of evictions being reached, which occurs at $log_2(s)$, where s is the size of the hash table. Therefore, this function has a worst case logarithmic time complexity relative to the size of the hash tables, => O($log_2(s)$).

Analysis of getValues function:

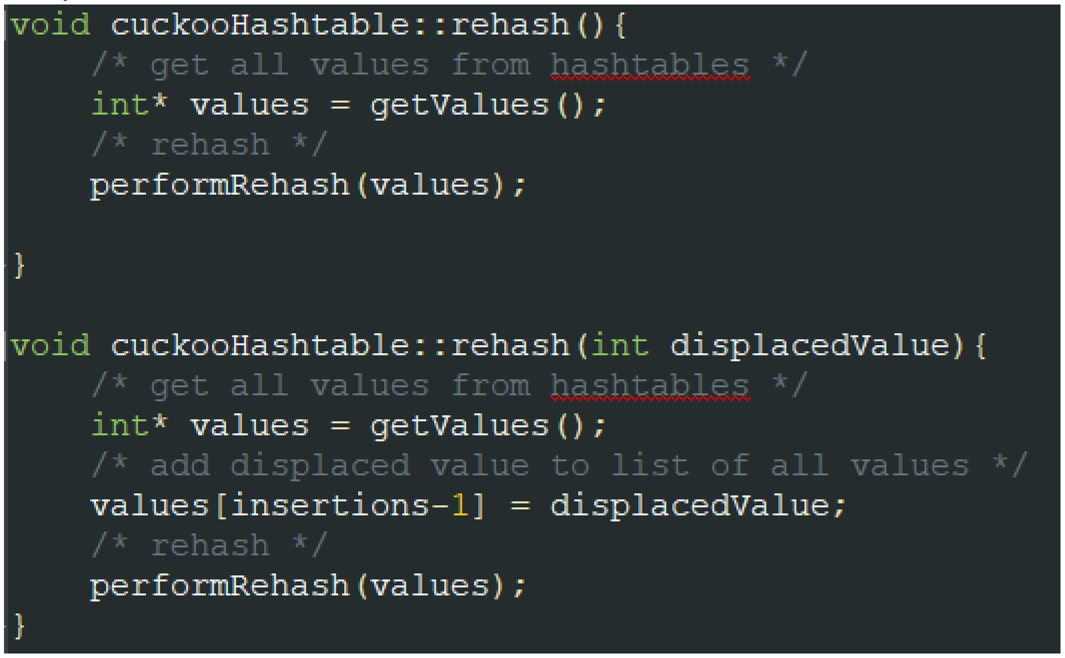


```
int* cuckooHashtable::getValues(){
    /* only insertions elements exist */
    int* values = new int[insertions]();
    /* fill values
     * index for values array */
    int j = 0;
    /* tableMap has numberElements*2 entries*/
    for(int i = 0; i < size; ++i){
        if(tableMap[i] == 1){
            values[j] = hashtable2[i];
            j++;
        }
        else if(tableMap[i] == 2){
            values[j] = hashtable1[i];
            j++;
        }
        else if(tableMap[i] == 3){
            values[j] = hashtable1[i];
            j++;
            values[j] = hashtable2[i];
            j++;
        }
    }
    return values;
}
```

Figure 12: Code to implement the getValues function in the cuckooHashtable class

The getValues function is a helper function called when the dictionary is rehashed. The getValues function dynamically creates an array that it fills with values from the hash tables, based on the value in the corresponding tableMap index. Once done, this function returns a pointer to the array containing all the values that were in both hash tables. this function iterates through the entire tableMap array, its time complexity is based on the size of tableMap, which is the size of the hash tables. Therefore, the getValues function has a linear time complexity with respect to the size of all arrays => O(s).

Analysis of rehash functions:

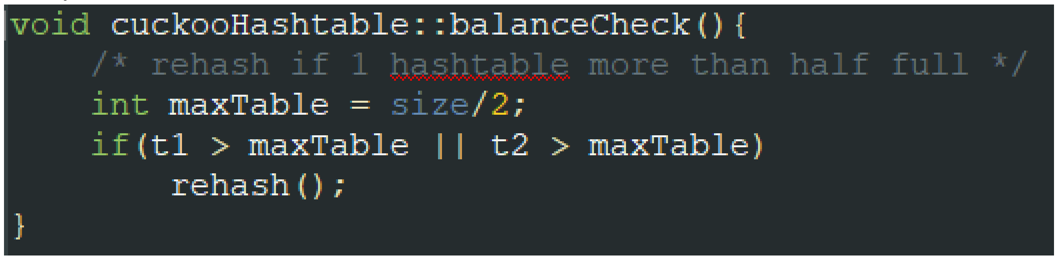


```
void cuckooHashtable::rehash(){
    /* get all values from hashtables */
    int* values = getValues();
    /* rehash */
    performRehash(values);

}

void cuckooHashtable::rehash(int displacedValue){
    /* get all values from hashtables */
    int* values = getValues();
    /* add displaced value to list of all values */
    values[insertions-1] = displacedValue;
    /* rehash */
    performRehash(values);
}
```

Figure 13: Code to implement the 2 rehash functions in the cuckooHashtable class

The rehash function is a helper function that aids in rehashing the hash tables. 2 versions of the rehash function exist, a default version and an overloaded version that accepts an integer argument. This function is responsible for obtaining an array of all the values in the dictionary and passing that array to the function that rehashes it. The default version of the rehash function accepts no arguments and is called for rebalances that occur due to tables being more than 50% full. The version overloaded to accept an integer argument is called when a rehash occurs due to an eviction, with the passed argument being the return value from the jumpAround function as it is no longer in any hash table due to being the final eviction. Each rehash function uses the linear getValues function to obtain all the values in the dictionary, which is passed to the performRehash function. The most complicated function in each instance is the performRehash function, whose time complexity determines the time complexity of both rehash functions.

Analysis of balanceCheck function:

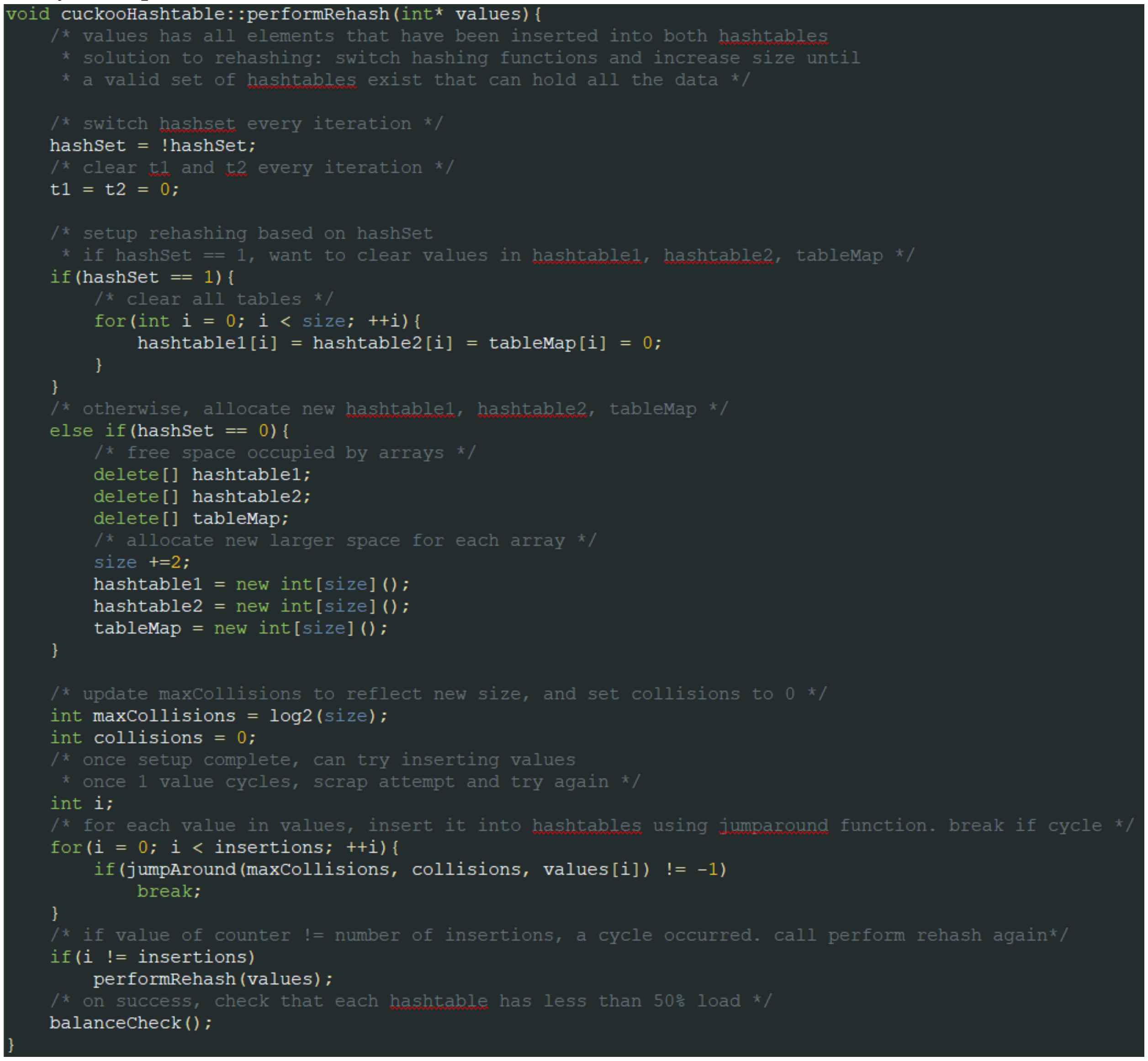


```
void cuckooHashtable::balanceCheck(){
    /* rehash if 1 hashtable more than half full */
    int maxTable = size/2;
    if(t1 > maxTable || t2 > maxTable)
        rehash();
}
```

Figure 14: Code to implement the balanceCheck function in the cuckooHashtable class

The balanceCheck function is a member function that checks the load of each hash table to determine if the dictionary needs to be rehashed. As this function calls the rehash function, it has the rehash function’s time complexity, which is that of the performRehash function.

Analysis of performRehash function:

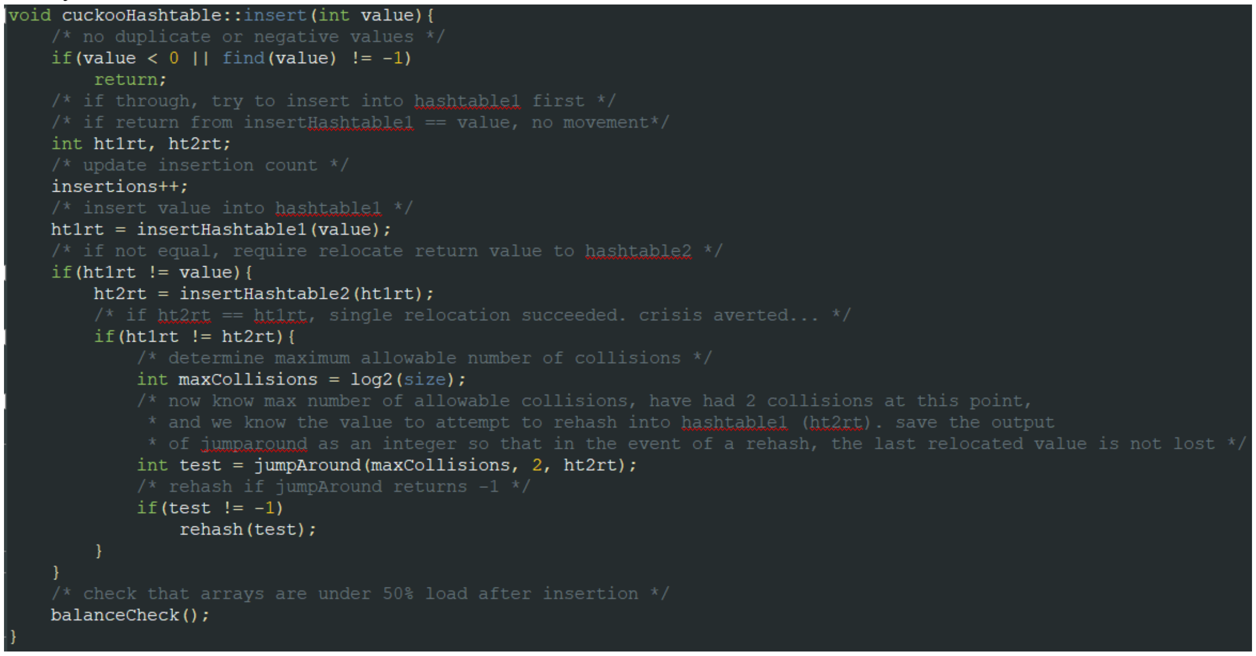


```
void cuckooHashtable::performRehash(int* values){
    /* values has all elements that have been inserted into both hashtables
     * solution to rehashing: switch hashing functions and increase size until
     * a valid set of hashtables exist that can hold all the data */

    /* switch hashset every iteration */
    hashSet = !hashSet;
    /* clear t1 and t2 every iteration */
    t1 = t2 = 0;

    /* setup rehashing based on hashSet
     * if hashSet == 1, want to clear values in hashtable1, hashtable2, tableMap */
    if(hashSet == 1){
        /* clear all tables */
        for(int i = 0; i < size; ++i){
            hashtable1[i] = hashtable2[i] = tableMap[i] = 0;
        }
    }
    /* otherwise, allocate new hashtable1, hashtable2, tableMap */
    else if(hashSet == 0){
        /* free space occupied by arrays */
        delete[] hashtable1;
        delete[] hashtable2;
        delete[] tableMap;
        /* allocate new larger space for each array */
        size +=2;
        hashtable1 = new int[size]();
        hashtable2 = new int[size]();
        tableMap = new int[size]();
    }

    /* update maxCollisions to reflect new size, and set collisions to 0 */
    int maxCollisions = log2(size);
    int collisions = 0;
    /* once setup complete, can try inserting values
     * once 1 value cycles, scrap attempt and try again */
    int i;
    /* for each value in values, insert it into hashtables using jumparound function. break if cycle */
    for(i = 0; i < insertions; ++i){
        if(jumpAround(maxCollisions, collisions, values[i]) != -1)
            break;
    }
    /* if value of counter != number of insertions, a cycle occurred. call perform rehash again*/
    if(i != insertions)
        performRehash(values);
    /* on success, check that each hashtable has less than 50% load */
    balanceCheck();
}
```

Figure 15: Code to implement the performRehash function in the cuckooHashtable class

The performRehash function accepts a pointer to an array of values that contain the contents of the dictionary prior to rehashing. When called, the value of hashSet is flipped and the counter of inserts into each hash table is reset. If the new value of hashSet is 1, the hash tables and tableMap are zeroed out. If the new value of hashSet is 0, the hash tables and tableMap are deleted and recreated at a larger size. The values are then attempted to be reinserted into each hash table. If the insertion fails, the performRehash function is called again until it is successful. Once successful, the balanceCheck function is called to ensure that each hash table does not exceed a load of 50%. If either hash table does, the rehash function is called which invokes performRehash once again. In terms of time complexity, this is an extremely expensive function to call. Each path depending on hashSet includes multiple expensive operations in the form of a linear time complexity zeroing out of all arrays or the invocation of 3 delete and new commands. Then, there are up to as many calls to the jumpAround function as there are insertions and the potential for another recursive call if the entire array of values is not successfully inserted or the balanceCheck function detects one table having over 50% load. As a result, the worst case time complexity for this function cannot be computed, but the best case time complexity can be. The best case occurs when the hashSet variable is equal to 1, causing all the arrays to be zeroed out. consists of one iteration of the function which includes a linear segment proportional the size of the hash tables s and a logarithmic section proportional to the number of insertions into the dictionary i. Assuming this is successful and the balanceCheck passes (as it is expected to in the best case scenario), this function has a time complexity of $\Omega(s+i log_2(s))$. In the best case scenario, the number of insertions i is equal to half of the hash table size s. This turns the time complexity into $\Omega(s+(s/2) log_2(s))$. Finally, $(s/2) log_2(s)$ is the dominant term so this function has a best case time complexity of $\Omega((s/2) log_2(s))$.

Analysis of insert function:



```
void cuckooHashtable::insert(int value){
    /* no duplicate or negative values */
    if(value < 0 || find(value) != -1)
        return;
    /* if through, try to insert into hashtable1 first */
    /* if return from insertHashtable1 == value, no movement*/
    int ht1rt, ht2rt;
    /* update insertion count */
    insertions++;
    /* insert value into hashtable1 */
    ht1rt = insertHashtable1(value);
    /* if not equal, require relocate return value to hashtable2 */
    if(ht1rt != value){
        ht2rt = insertHashtable2(ht1rt);
        /* if ht2rt == ht1rt, single relocation succeeded. crisis averted... */
        if(ht1rt != ht2rt){
            /* determine maximum allowable number of collisions */
            int maxCollisions = log2(size);
            /* now know max number of allowable collisions, have had 2 collisions at this point,
             * and we know the value to attempt to rehash into hashtable1 (ht2rt). save the output
             * of jumparound as an integer so that in the event of a rehash, the last relocated value is not lost */
            int test = jumpAround(maxCollisions, 2, ht2rt);
            /* rehash if jumpAround returns -1 */
            if(test != -1)
                rehash(test);
        }
    }
    /* check that arrays are under 50% load after insertion */
    balanceCheck();
}
```

Figure 16: Code to implement the insert function in the cuckooHashtable class

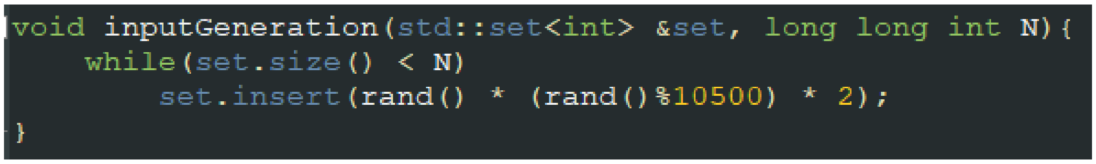
The insert function takes an integer argument that it attempts to insert into the dictionary. First, the value is checked to ensure that it is not already in the dictionary and that it is positive. If this check passes, the number of insertions is incremented, and the value is attempted to be inserted into the first hash table. If this operation does not return the inserted value, an eviction occurred, and the returned value is attempted to be inserted into the second hash table. If the insertion into the second hash table does not return the inserted value, the jumpAround function is called to try and find a place for the value evicted from the second hash table. If the jumpAround function fails to find a place for the value within the allowed number of evictions, the rehash function is called to setup for the rehash of the dictionary. Once the rehash function finishes its setup and the performRehash function finishes, the balanceCheck function is called to ensure that each hash table remains under 50% load. Interestingly, the balanceCheck function will only ever rebalance the dictionary if an insertion occurred with requiring a rehash, as the balanceCheck in the performRehash function will eliminate the potential of a rehash at this point. Concerning the time complexity of this function, it has the potential to call the performRehash function, so it takes on its time complexity. However, the odds that an insertion reaches the point of requiring a rehash become low once the table is sufficiently large and kept over 50% empty according to Pagh and Rodler and the proof from the above Stanford Data Structures Lecture, so insertions are concluded to have an amortized constant time complexity => O(1).

Testing Methodology:

To test this implementation of cuckoo hashing, driver code was created from the sample project test code posted on Brightspace called "Complexity_Runtime_DS_Example". In the timeframe which this project needed to be completed, no student I reached out to responded with their hash based data structure. Consequently, the AVL tree created for project 2 was used instead. The AVL Tree is a good substitute even though it is based on a different data structure as it theoretically offers comparable time complexities for deletion, insertion, and search operations to cuckoo hashing at a radically different space requirement. The driver code recorded the time that it took the cuckoo hashing dictionary, the AVL Tree, and the STL Vector to perform each test using the supporting Complexity_Recorder.hpp" and "Complexity_Timer.hpp" files. The three data structures were subject to the same three tests, which were the time to be filled with random unique data, the time to delete all the values in the data structure, and the time to find the largest value in the data structure. The dictionary using cuckoo hashing was also subjected to a special test which varied how much the size of each hash table was increased when the dictionary was rehashed, allowing the impact on performance and the amount of wasted space to be found to determine if an optimal method of increasing the hash table size exists.

When performing the tests, the problem size N was defined in terms of a lower bound, 100, and an upper bound, 25600. Once an iteration of a test completed, the lower bound was increased to scale the size of the problem. For the insertion tests, the lower bound was increased by 100 each iteration, while it was multiplied by 2 for the search and deletion tests. This was as the time to insert the data into the cuckoo hashing dictionary could vary wildly depending on how many rehashes would be required to store the randomly generated data, so a smaller step size provided more data points from which a conclusion regarding performance could be derived. Additionally, with each iteration of each test, the lower bound was multiplied by a constant factor = 10 to determine the final problem size for each iteration of the test. Each test was then repeated on the data structure 4 times, defined by the constant "REPETITIONS" at the top of the driver source code.

For each test, input was generated randomly and placed into a STL set to prevent duplicates. This ensured that the dictionary with cuckoo hashing would always have a number of elements equal to the problem size, and that each data structure was fed with the same randomized data for each iteration of each test. The function that did this is called inputGeneration and is shown below in Figure 17.

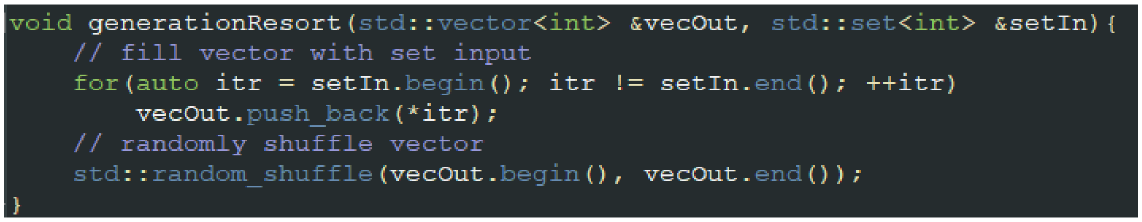


```
void inputGeneration(std::set<int> &set, long long int N){
    while(set.size() < N)
        set.insert(rand() * (rand()%10500) * 2);
}
```

Figure 17: Code to implement the random input generation function inputGeneration in the driver code

The inputGeneration function uses 2 rand function calls to generate values greater than the largest problem size so that a random subset of values can be generated but takes care to not exceeded the maximum value than an integer can store: 2147483647. As RAND_MAX is set to 32767 by default, this approach can produce values in the range ((0 - 32767) * (0 - 10500) * 2), resulting in a minimum of 0 and a maximum of 688107000, which is well between the largest problem size and the maximum value than an integer can store.

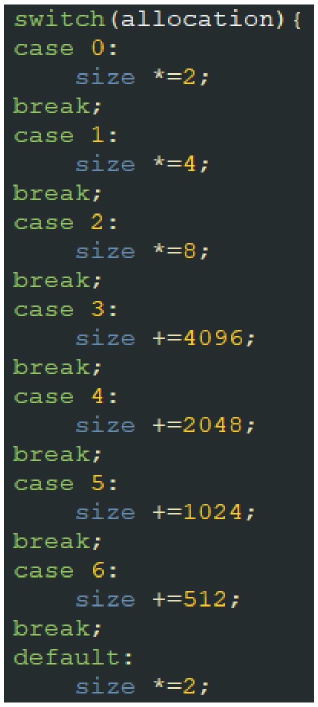
An additional function called generationResort was created to take the values that were generated using inputGeneration, place them in a vector, and then randomly sort them. This produced a random order of values that was used as the order of deletion for the deletion test. This was beneficial as it moved values from the set, which does not support direct element access with "[ ]" to a data structure that does. The code to implement this function is shown below in Figure 18.



```
void generationResort(std::vector<int> &vecOut, std::set<int> &setIn){
    // fill vector with set input
    for(auto itr = setIn.begin(); itr != setIn.end(); ++itr)
        vecOut.push_back(*itr);
    // randomly shuffle vector
    std::random_shuffle(vecOut.begin(), vecOut.end());
}
```

Figure 18: Code to implement the random sorting of input function generationResort in the driver code

For the test to vary the method to increase the hash table size, a copy of the cuckooHashtable class was created called "cuckooHashtable_modAlloc". The modified class includes a constructor that is passed an integer that specifies one of 7 ways the size of the hash tables will be increased during a rehash. To support this, an additional integer variable called allocation was included in the cuckooHashtable_modAlloc class definition and logic to check this value and increase the size appropriately was inserted into the performRehash function. The code added to the performRehash function is shown below in Figure 19.



```
switch(allocation){
case 0:
    size *=2;
break;
case 1:
    size *=4;
break;
case 2:
    size *=8;
break;
case 3:
    size +=4096;
break;
case 4:
    size +=2048;
break;
case 5:
    size +=1024;
break;
case 6:
    size +=512;
break;
default:
    size *=2;
```

Figure 19: Switch statement added to the performRehash member function in the cuckooHashtable_modAlloc class, allowing the method of increasing the hash table size to be specified

For this test, input was randomly generated with the inputGeneration function and fed into an original cuckooHashtable object and 6 instances of cuckooHashtable_modAlloc objects that each varied how the table size would be increased. An additional debug member function was written to output the size of each object, the total number of insertions, and the number of insertions into the first and second hash tables to determine how the space used by each object varied. The upper bound was also changed from 25600 to 12800 as 25600 resulted in tests that took too long to complete and could occasionally crash.

Results:

This research yielded 3 results, which were the time complexities of the deletion, insertion, and search operations for cuckoo hashing, the affect the method of increasing hash table size has on insertion runtime and efficient space use, and the space complexity of the dictionary using cuckoo hashing (as the space complexity of the other 2 data structures are already known).

First, the insertion execution time as a function of the problem size is shown below in Figure 20. For this test, all data structures were rehashed from the lower bound of the problem size to the upper bound in steps of 1000.

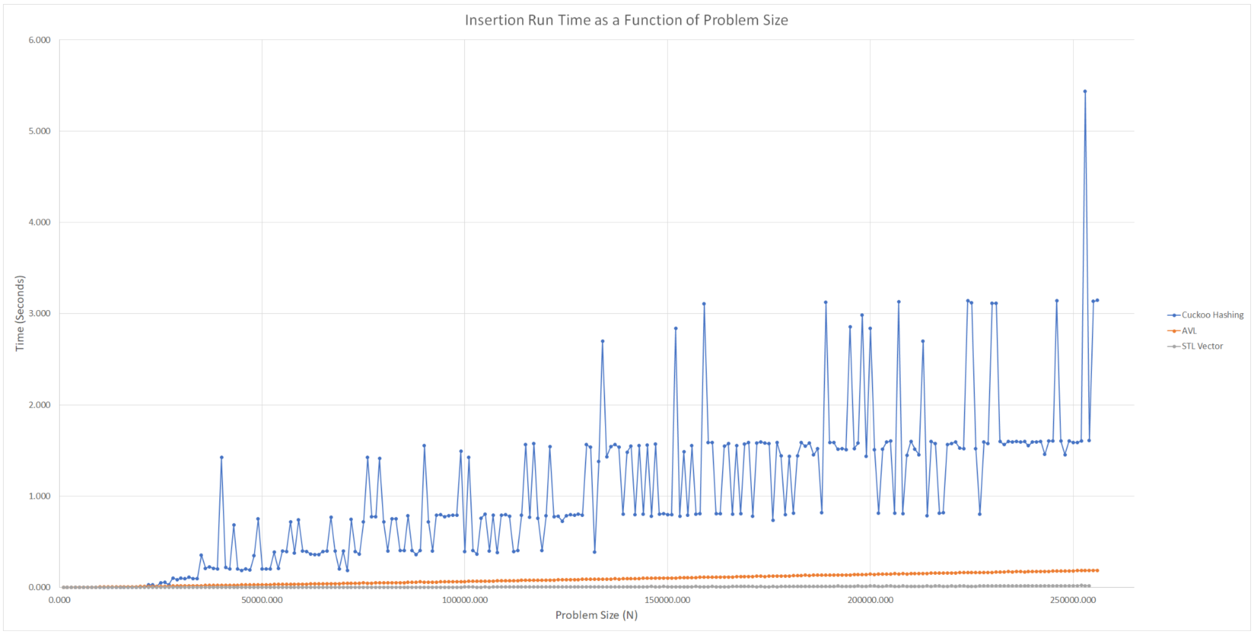


Figure 19: Insertion run time as a function of problem size, incremented in steps of 1000 elements

As shown, the AVL Tree and the STL Vector performed considerably faster than cuckoo hashing for problem sizes roughly greater than 20000 elements. The AVL tree performed worse due to its self balancing nature, while insertions into the STL Vector were easily done with the "push_back" method. With cuckoo hashing, the runtime was shown to generally increase as the problem size increased. However, for certain groupings of problem sizes, the runtimes clustered around a median time with outliers occurring both above and below. As input was randomly generated, this variance was due to some sets of data requiring greater or fewer rehashes than others. This does support the claim of an amortized constant time run time, as the groupings of similar runtimes indicate that when similar numbers of rehashes are required, similar sized ranges of elements can be inserted in near equal amounts of time. These differences in time show the strength and weakness of cuckoo hashing, as when it works efficiently, the outliers from the median are much faster than the median time to insert. However, when cuckoo hashing works poorly on the set of data, the penalty in insertion time can be very large as shown.

The results for the deletion and search tests are shown below.

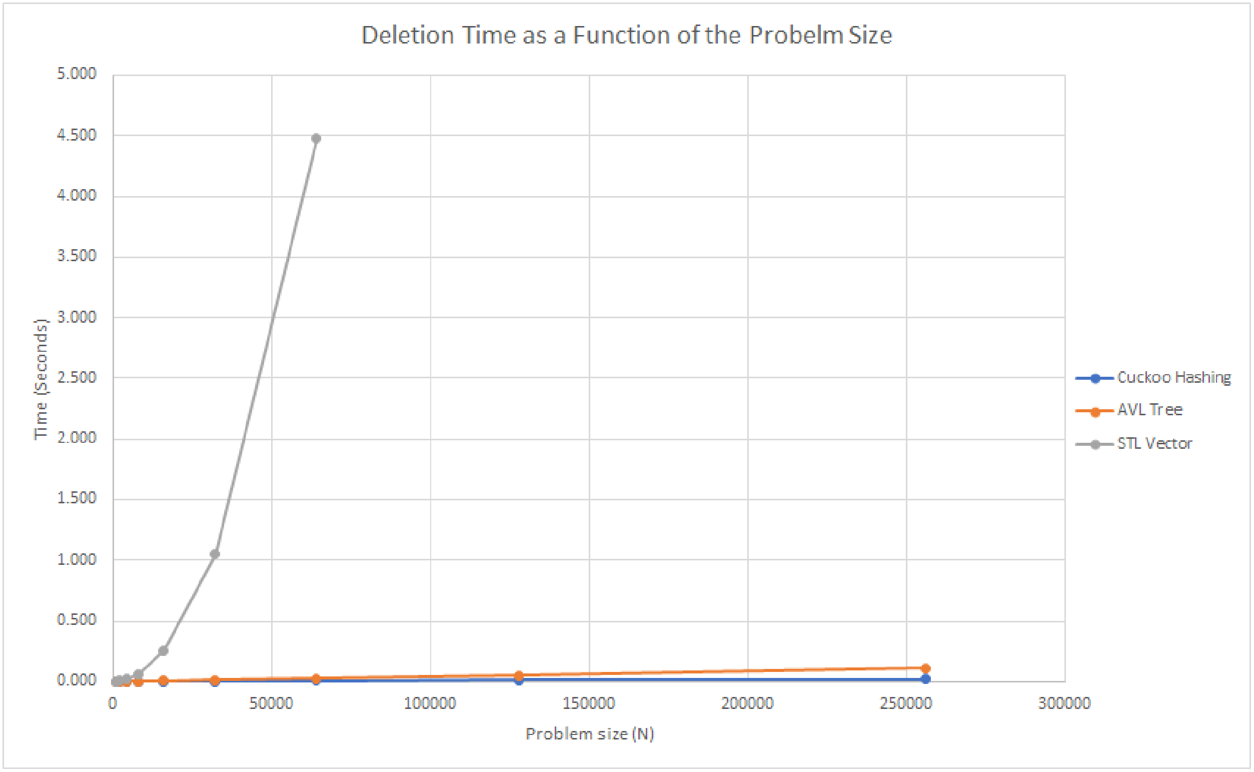


Figure 20: Time to delete the entire structure as a function of problem size

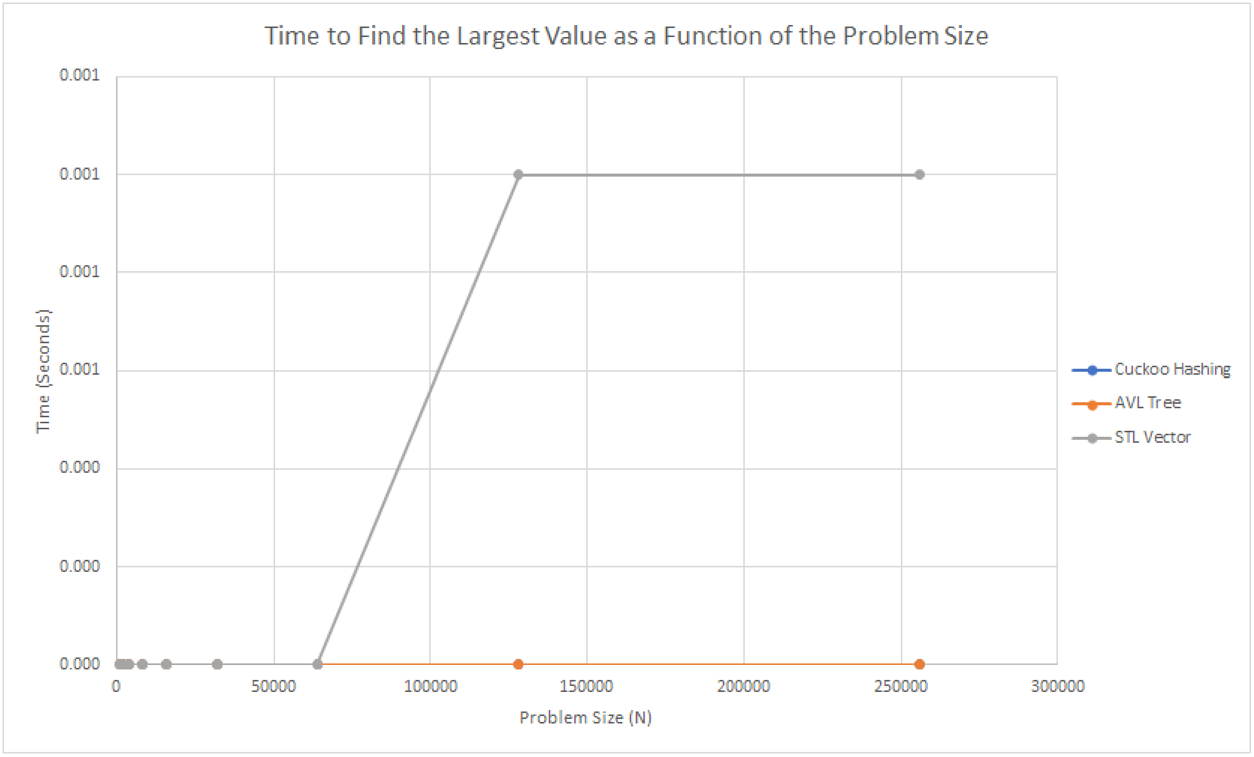


Figure 21:Time to find the largest value as a function of the problem size

As shown in figure 20, cuckoo hashing provided a much faster deletion time than both the AVL tree and the STL vector. This was caused by the dictionary having to check 2 locations, while the AVL tree had to navigate its branches to find the indicated node, remove it, and then rebalance if necessary. Likewise, the value had to be found in the STL vector prior to deletion, which was done using iterators. To implement this, the stl::find method was used to generate an iterator pointing to the specific value to delete that was passed to the vector's erase method. This produced the quadratic runtime shown below, as the value had to be found using stl::find first and then stl:erase caused all the values after the deleted value to be shifted towards the beginning of the vector.

Figure 21 shows the time to find the largest inserted value in each data structure. Searches are easy in the dictionary as only 2 indices need to be checked. Within the AVL tree, searching is a little more complex, but the largest value is always in the rightmost leaf of the AVL tree. This requires $log_2(N)$ traversals to find, which is a small value even for large data sets. For the vector, even though data is stored in a linear fashion, the time to find a single value is short. Multiple searches would drastically increase the time required to find all values.

The next results detail the impact that the method of increasing the hash table size during a rehash has on the runtime of the insertion operation, as well as the usage of allocated space in the dictionary. These results are biased, as the performance of cuckoo hashing ultimately depends on the data inserted into it. However, the input was kept constant when alternating between methods of increasing table size to demonstrate the potential impact on the same set of data. The indicated results are not guaranteed to hold exactly for different data sets, no matter they be larger, smaller or the same size, but they demonstrate the relationship between insertion time, the method if increasing the table size, and space utilization in the dictionary. As a final note, remember the upper bound of the problem size was reduced to 12800 for the following tests.

First, the relationship between the insertion time and the method of increasing table size was established.

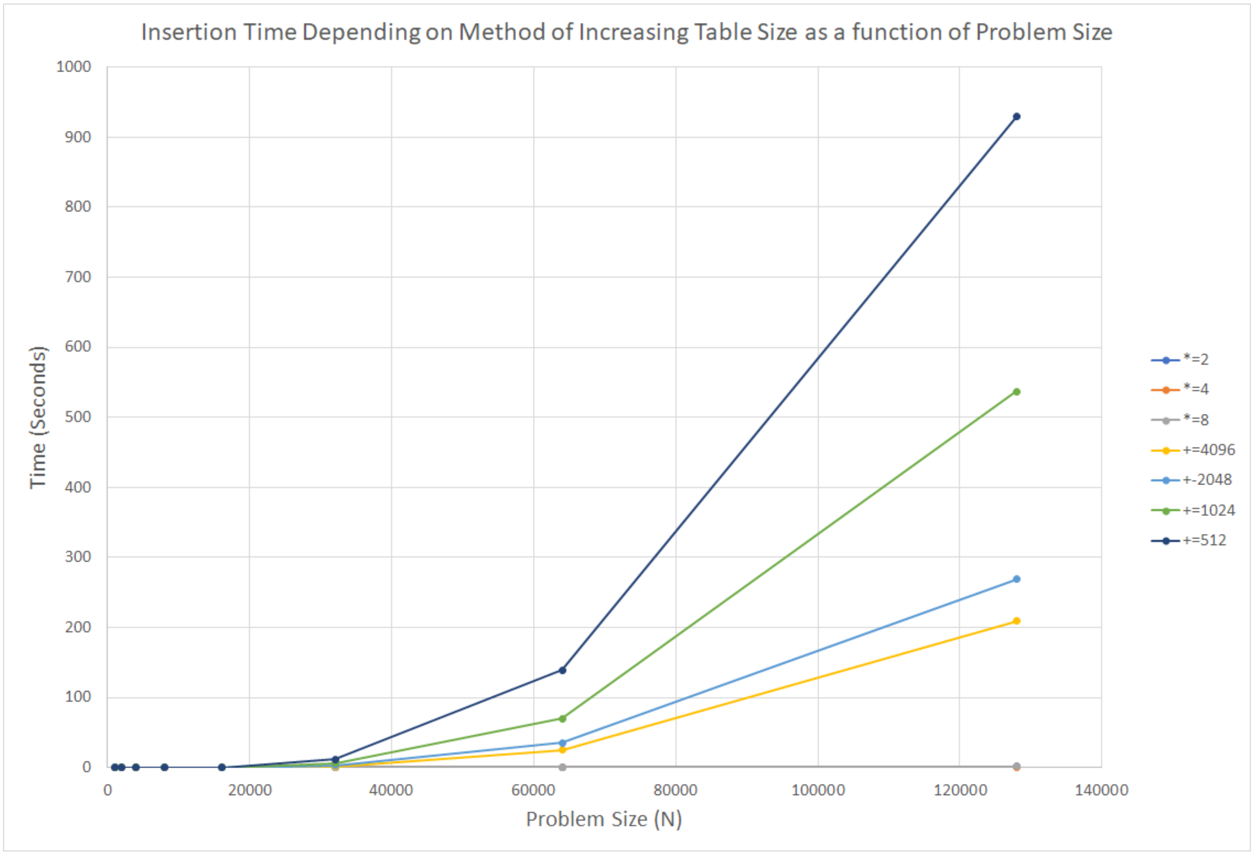


Figure 22: Time to insert values relative to the problem size and dependent on the method of increasing table size

As shown in Figure 22, insertions that used multiplication to increase the table size experienced far faster runtimes than addition. Increasing the table size by multiplying the current size by 4 was the fastest, followed by multiplication by 8 and 2 respectively. Switching from multiplication to addition produced a large increase in the time to insert all the values, with the addition of smaller values resulting in drastically longer execution times compared to larger values. This was expected as multiplication allows the largest increase in table size at once, reducing the likelihood that another rehash will be needed in the future.

Figure 23 shows the final size of each hash table required to store the problem size depending on the method of increasing table size.

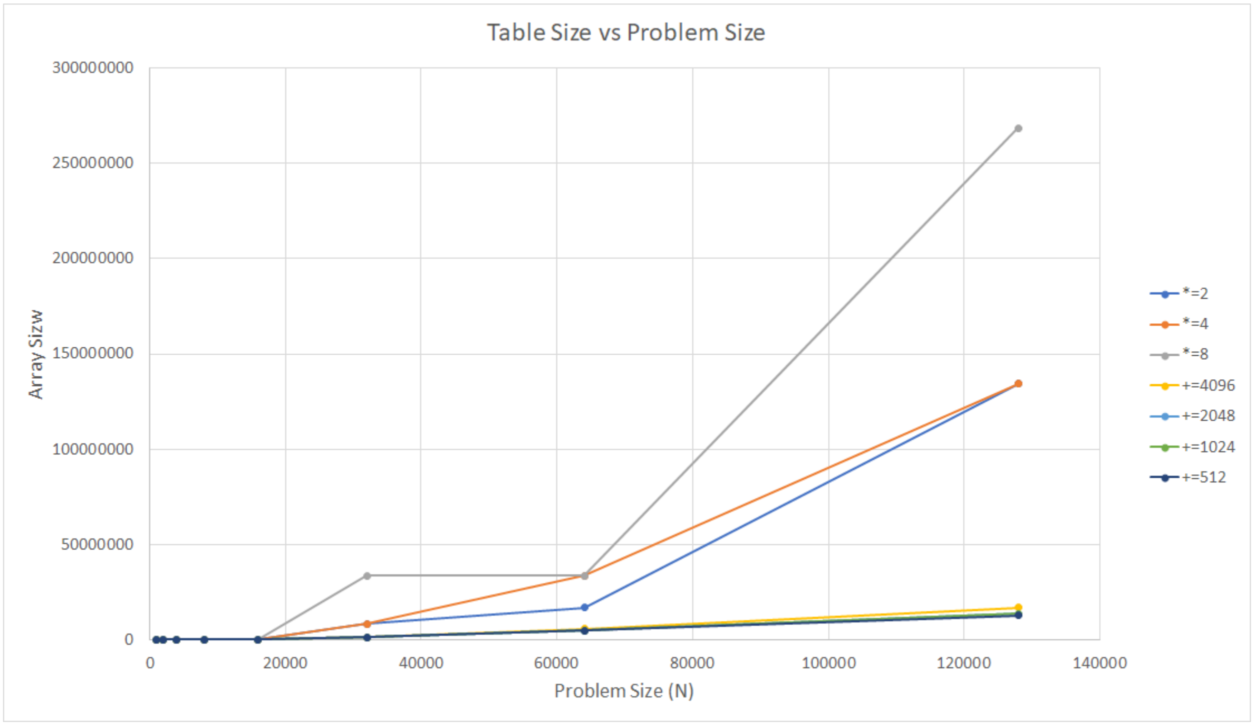


Figure 23: Hash table size required to store the problem size and dependant on the method used to increase table size

As shown, larger hash table sizes resulted in far faster execution times for the insertion test. However, using addition to increase table size resulted in hash tables much smaller than those created using multiplication. For all methods of increasing table size, the size of each hash table greatly exceeded the problem size for all instances except small problem sizes (N < 32000), resulting in lots of allocated space and low utilization.

To determine the space utilization associated with each method of increasing table size, the number of insertions were subtracted from the final hash table size to determine the total amount of wasted space. Here, the final hash table size is 2 times the table size as both hashtable1 and hashtable2 need to be considered. Next, 100% bar graphs were created in Excel to illustrate how each method of increasing table size changed the utilization of allocated space. Each graph depicts the total space allocated between both the hash tables, the percentage of space used by each hash table, and the total amount of space wasted with respect to each problem size.

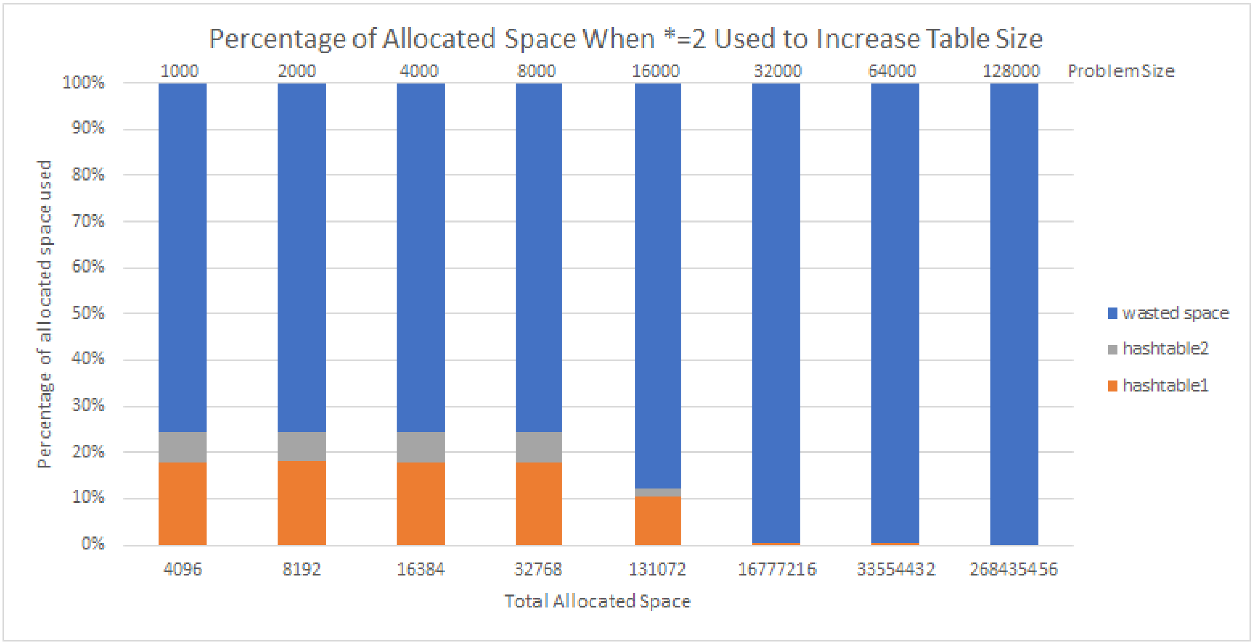


Figure 24: Percentages of Allocated Space When *=2 Used to Increase Table Size

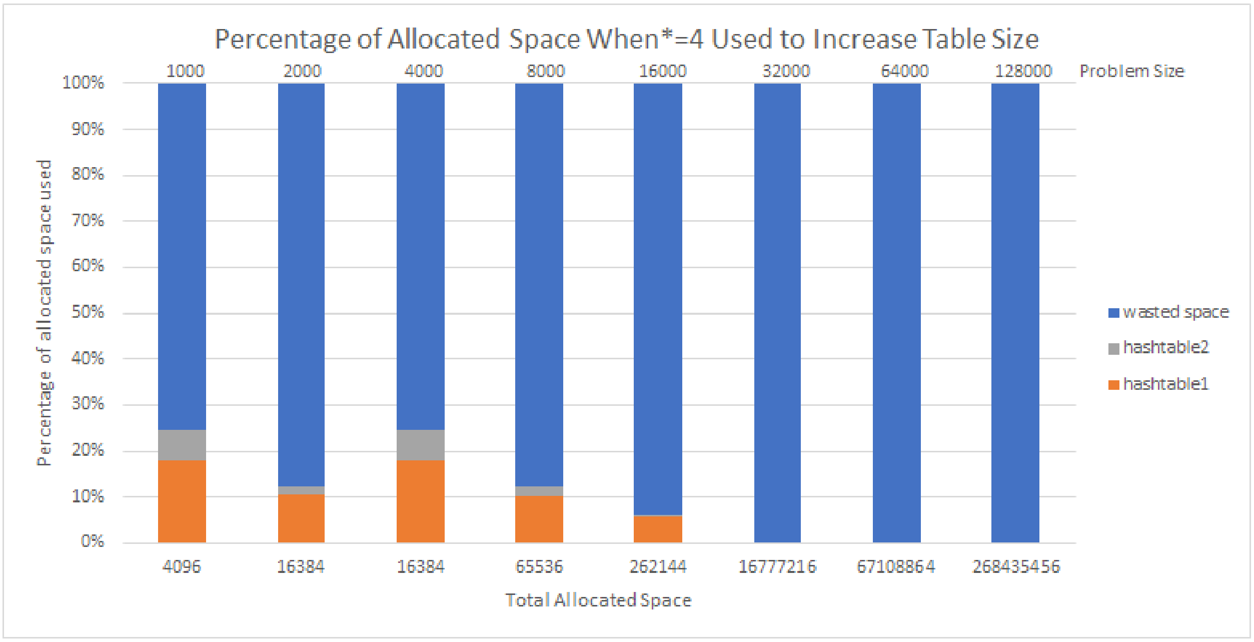


Figure 25: Percentages of Allocated Space When *=4 Used to Increase Table Size

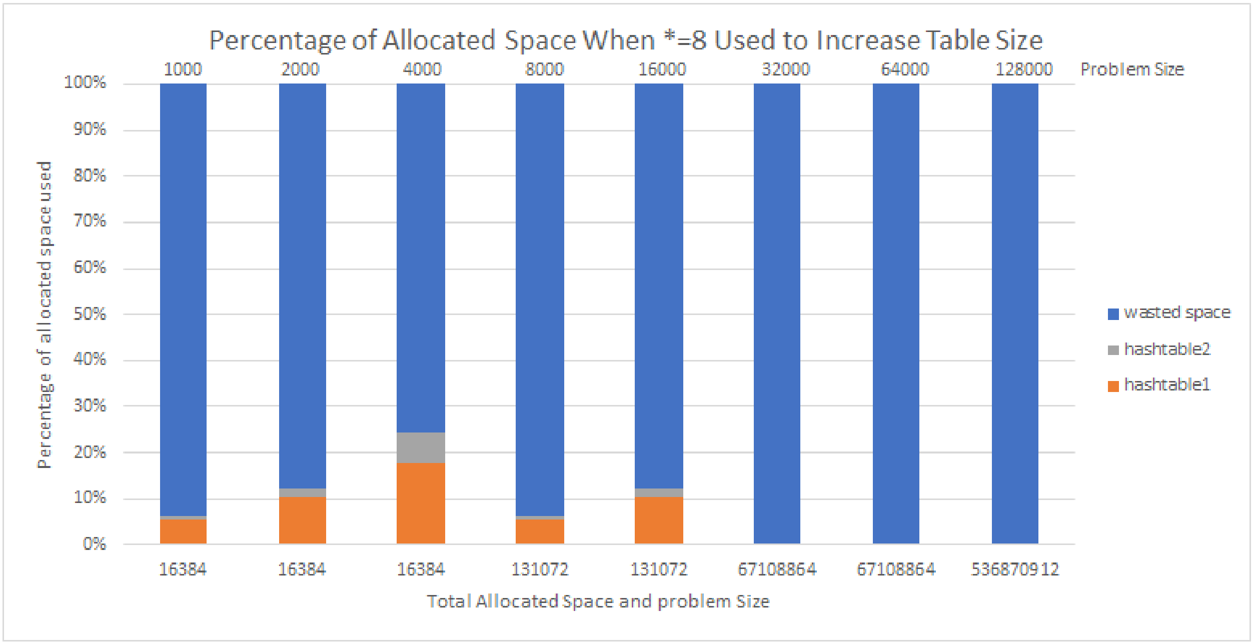


Figure 26: Percentages of Allocated Space When *=8 Used to Increase Table Size

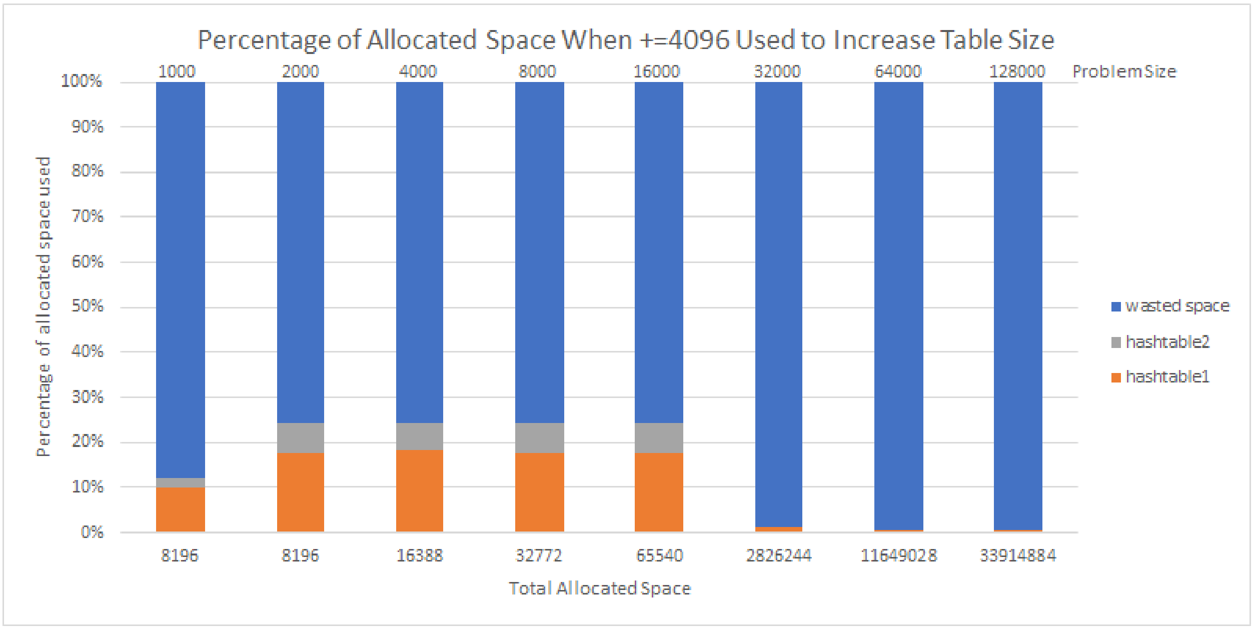


Figure 27: Percentages of Allocated Space When +=4096 Used to Increase Table Size

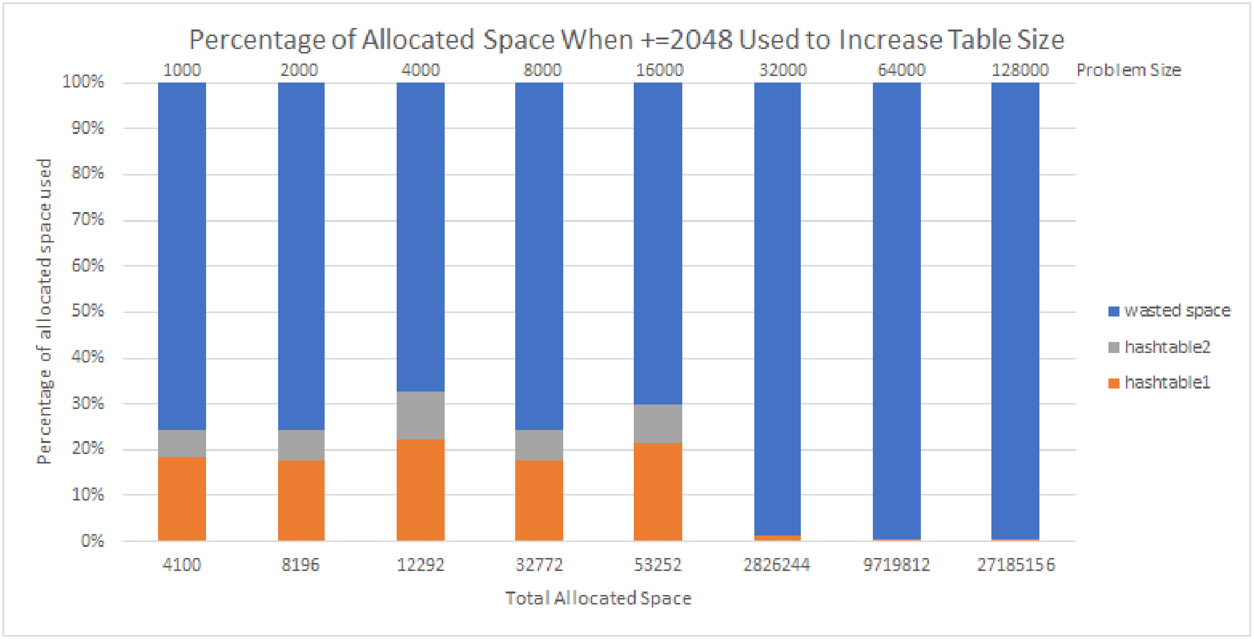


Figure 28: Percentages of Allocated Space When +=2048 Used to Increase Table Size

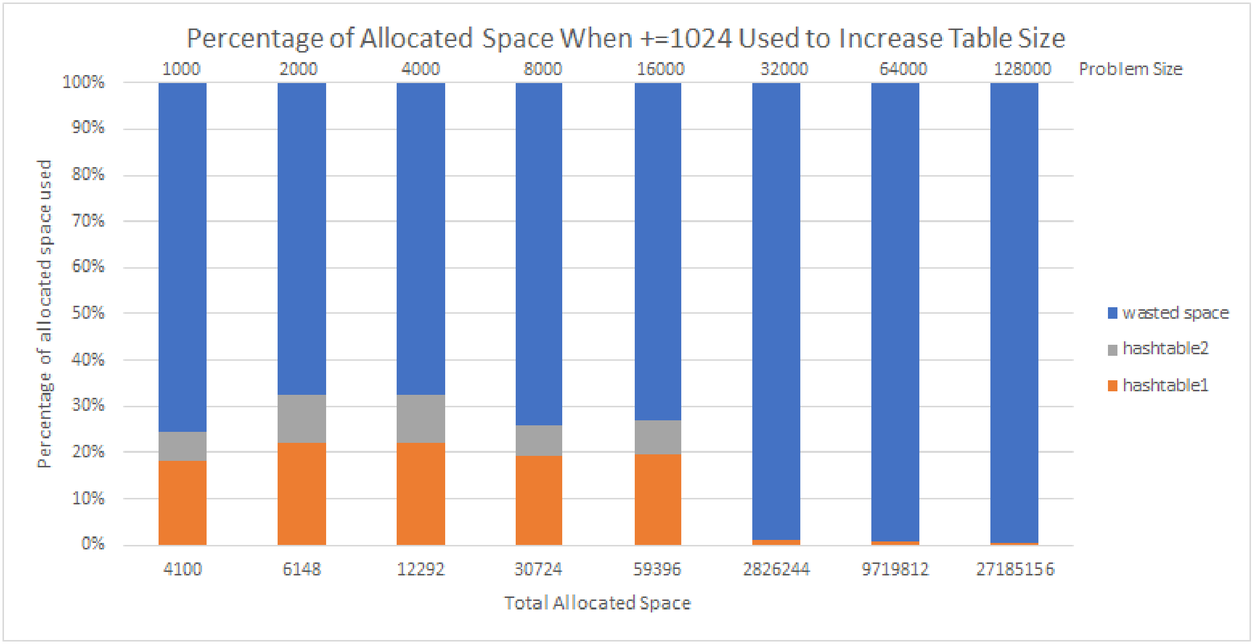


Figure 29: Percentages of Allocated Space When +=1024 Used to Increase Table Size

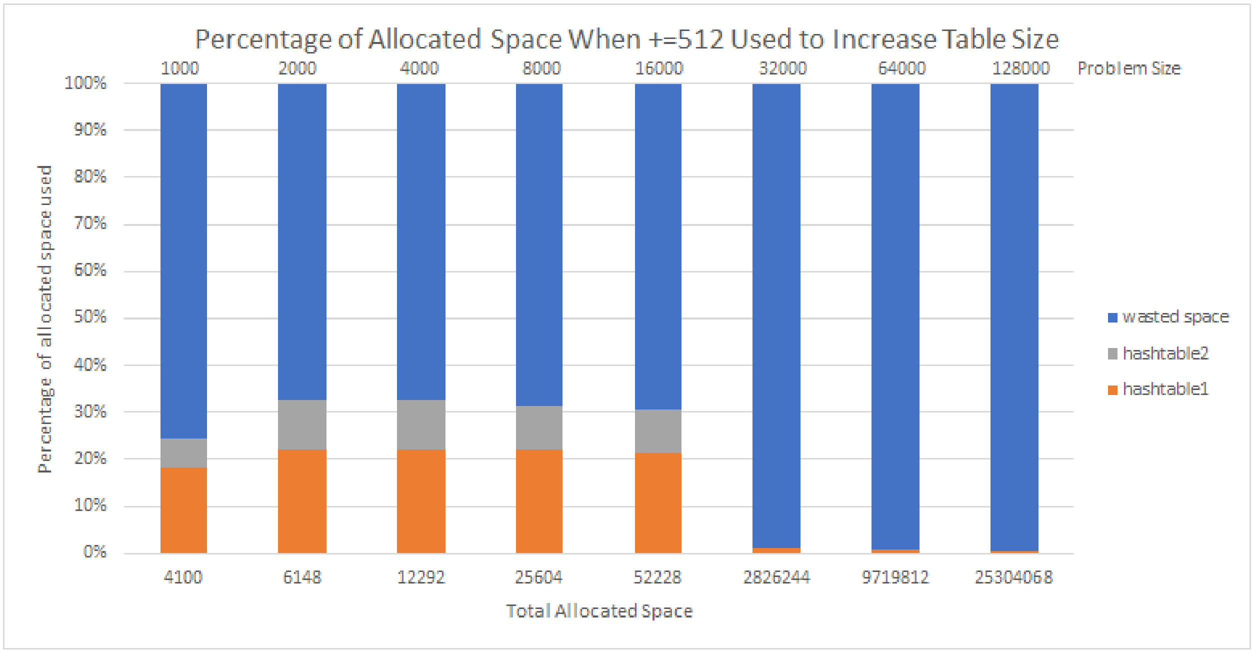


Figure 30: Percentages of Allocated Space When +=512 Used to Increase Table Size

At the top of Figures 24 through 30 is the problem size associated with each bar and total amount of space allocated at the bottom. The orange portion is the space utilized by hashtable1, the grey hashtable2, and the blue unused space. As shown, each method of increasing the table size resulted in the generation of nearly 100% wasted space for the problem sizes of 32000, 64000, and 128000. The smaller problem sizes performed much better compared to those, but they still resulted in large amounts of wasted space. For reference, according the Pagh and Rodler, each hash table must be at least 50% empty. The ideal cuckoo hashing dictionary will use space equal to 2 times the number of elements to store, with each hash table being allocated half the space and the elements split evenly between the two hash tables. Therefore, the ideal size for each hash table to store N elements is 2N bytes, allocating a total of 4N bytes. Alternatively, each hash table should have as close to 50% of its space utilized as possible, each being 25% of the total allocated space. As shown in the above figures, this was not achieved. The most space efficient dictionary was shown in Figure 30 and had a total space utilization of slightly over 30%. The least efficient (bar the problem sizes mentioned above), was less than 10% in Figures 25 and 26. Evidently, a large amount of space is wasted by this implementation of cuckoo hashing.

Another interesting result shown in Figures 24 to 30 is the bias towards hashtable1. Most elements in each insertion test case occurred in hashtable1, despite the large amount of free space present in hashtable2. As each hash table must be 50% full to have a maximally space efficient dictionary, it is even more evident that this implementation performs poorly.

Looking at the space complexity of the cuckooHashtable class, it uses 3 arrays of variable size, which must be at least 2 times the number of elements to hold. Additionally, 1 bool, 4 integers, and 3 pointers to the arrays are required to implement each instance of the class. The space to implement an instance of the cuckooHashtable is 5V+3(P+S), where V is the size of the bool and integer variables, P is the size of the pointers, and S is the size of each hash table. Generally, V = 4 and P = 8, so the equation becomes 44+3S. In the best case scenario, the elements are split evenly between the hash tables, so all tables have size N. In the worst case, the first hash table holds all the values and has a size of at least 2*N, which is the minimum size to be at least 50% full. This requires a minimum of 2*N*Z bytes, where Z is the size of the data being stored. Thus, the best size that can be obtained is 44+3ZN bytes and the "best" worst case scenario with the first hash table being exactly large enough to hold all the elements has a size of 44+6ZN bytes. Assuming integers are being stored, the best case scenario becomes 44+12N bytes and the "best" worst case scenario 44+24N. Here, it is important to note that the space required in a true worst case scenario is impossible to determine as it depends on the input data and the method of increasing table size when rehashing. If data requires no rehashes and the first hash table is exactly twice as large as the amount of input, a size 40+24N is guaranteed. As shown the results, this is unlikely to be achieved as the size of the hash tables are never remotely close to the size of the problem, so the actual space used will be greater.

The AVL Tree implementation used here has an overhead fraction of 2/3 and requires n(2P+2D) bytes to store n elements. Supposing 1000 elements need to be stored, D = 4 bytes and P = 8 bytes, the AVL Tree requires 1000(2*8+2*4) = 24000 bytes. Attempting this with the cuckoo hashing dictionary requires at best 44+12(1000) = 12044 bytes. If the assumption is that all the values will be in the first hash table, 44+24(1000) = 24044 bytes are required. Evidently, this is more space efficient than the AVL Tree if all elements are not stored in the first hash table and the table size is kept under 2N. This becomes untrue once both those requirements are broken. The reason behind this result is that the AVL Tree's size is incurred due to overhead: only ⅓ of its size is spent storing the actual data = 8000 bytes. Comparing the space to store 1000 elements in the vector, D*N bytes are required = 4000 bytes. The cuckoo hashing implementation is much less efficient than the vector, requiring over 3 times as much space in the absolute best case scenario and over 8 times as much space in the best worst case scenario.

Conclusion:
In conclusion, this experiment demonstrated the benefits and shortcomings associated with dictionaries that use cuckoo hashing. Worst case constant time searches and deletions are promised, but the performance of insertions is variable based on the data that the dictionary will store. It is possible for an amortized worst case constant time complexity to be achieved for insertions, but this is not guaranteed. Provided unique input, insertions tended to cluster around a time relative to a range of problem sizes with some outlying times both above and below the median time. Cycles are the biggest drawback to cuckoo hashing, requiring the table to be repeatedly rehashed and its size increased until all elements can be successfully stored. This is a time consuming operation, as it is unknown when a cycle will occur so all input must be processed linearly. For instance, during a rehash, 90% of the input could be rehashed and then a cycle detected, requiring the hash functions to change and possibly increase the table size before starting the entire rehashing process over.

One objective of this research was to determine how the method of increasing table size impacted the runtime of the insertion operation and how space was used in the dictionary. Due to the problem mentioned above, increasing the table size in too small of increments caused cycles to repeatedly occur. As recursion was used to implement insertions and rehashing, it was possible for the number of recursive calls to be too large and crash the program due to stack overflows, segmentation faults, and other memory leaks. This occured when a large problem size attempted to solve a cycle by increase the table size in a small amount relative to the problem size, such as a problem size of 12800 elements increasing the table size by +=2, +=4. +=8, etc. These allocation methods were originally included in this test but removed due to their inability to function for the larger problem sizes. This itself is an interesting result, as increasing the size of each hash table in increments of 2 worked flawlessly for smaller problem sizes but broke with the larger ones. It can be hypothesized that increasing the size of the tables in smaller amounts would be more efficient as it was demonstrated that large increased to the table size resulted in an extreme waste of space. Furthermore, smaller problem sizes that can take advantage of these more efficient methods of increasing table size will likely experience an increase in insertion time just like the larger problem sizes with "smaller" methods of increasing the table size.

Also observed was that despite the variation in how table size was increased, the second hash table was always underutilized. This demonstrates another fatal flaw of cuckoo hashing: the predominant use of one hash table and not the other. As values are always first hashed to the first hash table and the hash function are dependent on the hash table size, it is probable that most values in the second hash table will be mapped to first hash table during a rehash as the probability of collisions in the original table decreased. This is especially true for values (or in a more sophisticated implementation, values that generate keys) much smaller than the size of the table, as they will be indexed uniquely unless the hashing function utilizes a smaller table size (such as half of the actual table size) to produce more evictions. This effect was exaggerated by the fact that the input was generated uniquely in attempt to generate unique indices into each hash table, but ultimately resulted in the first hash table obtaining most of the values as every other rehash increased the portion of the input that could be stored in it. This resulted in an unacceptable amount of wasted space, as the hash tables have to be equal size so overuse of one causes underuse of the other. Expanding the first hash table so it can hold more elements results in the second increasing to hold fewer.

Increasing the size of the hash tables in larger increments did have the benefit of decreasing the time to perform all the insertions. This result makes sense as if the hash functions are relative to the size of the tables, a greater increase of their size results in the creation of a greater range of possible indices. Here, increasing the table size in larger amounts decreases the likelihood of future collisions and allows elements to be inserted in near constant time. However, increasing the size of the hash tables in larger chunks also results in more space being wasted, as most values will be hashed to first hash table and the size will keep being increased by the larger amount until it holds most of the values and is less than 50% full. Hence, a tradeoff between the time to complete and the space to implement the dictionary is present.

As shown, this implementation of a dictionary with cuckoo hashing was able to achieve constant time searches and deletions, along with an amortized constant time insertion. The method in increasing the hash table size by doubling it resulted in a good balance between execution time and space utilization. Other modifications could be made to the cuckooHashtable class to increase its performance, such as each index in the hash table becoming a bucket capable of storing a finite number of values. As displacements were found to be rare, the buckets could be implemented as short linked lists to ensure all elements can be inserted into the proper bucket and prevent the tradeoffs associated with linked lists due to their short length. To solve the underutilization of the second hash table, the implementation of the cuckoo hashing algorithm could be modified to alternate insertions between the first and second hash tables to grow them at the same rate compared to only performing insertions into the first hash table. Given a set of input that would produce unique indices in either hash table, each hash table would need to be the size of the input to store all the data compared to all hash tables being 2 times the size of the input if only stored in the first hash table. The final improvement to this structure could be changing the hash functions to be better suited to the data that will be input, with the goal becoming to cause more displacements to ensure better utilization of the second hash table. This could be achieved by the hash functions using a value that is a subset of the size of each hash table, such as half the hash table size, to restrict the locations that the data could be placed in each hash table and increase the probability of displacements.

References:

Crissey Renwald, W. (n.d.). *Hashing Part 4 - perfect and cuckoo* [VIDEO]. Akron: University of Akron Department of Computer Science.

Heto. (2017, May 23). Function pointer to member function [web log]. https://stackoverflow.com/questions/2402579/function-pointer-to-member-function.

Martin Dietzfelbinger, & Christoph Weidling (2007). Balanced allocation and dictionaries with tightly packed constant size bins. *Theoretical Computer Science, 380(1), 47-68.*

Mitzenmacher, M. (2009). (rep.). *Some Open Questions Related to Cuckoo Hashing*. Retrieved from http://www.eecs.harvard.edu/~michaelm/postscripts/esa2009.pdf

Rasmus Pagh, & Flemming Friche Rodler (2001). Cuckoo hashing. In *JOURNAL OF ALGORITHMS* (pp. 2004).

Schwarz, K. (2014, May). *Cuckoo Hashing*. *CS166 Data Structures*. Stanford, CA; Stanford University.
https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/13/Small13.pdf.