De La Salle University

Term 1, A.Y. 2022 - 2023

In partial fulfillment

of the course

Data Structures and Algorithms

MCO2 Documentation Report: Search Algorithms on Hash Tables and BSTs as a Method of Generating k-Mer Distributions

Submitted By:

Dichoso, Aaron Gabrielle C.

Donato, Adriel Joseph

Natividad, Josh Austin

Razon, Luis Miguel Antonio

Submitted To:

Christine Gendrano

December 09, 2022

Documentation Report - MCO2

I. Description of the algorithm in Task 1 and Task 2

```
main(HashTable, DNASequence, n, k)
{
    kMers = insertkMersHT(HashTable, DNASequence, n, k);
    createHTDistribution(kMers, HashTable, n, k);
}
```

```
insertkMersHT (HashTable, DNASequence, n, k)
{
    //Let kMers be an array of Strings with size k^n
    x = 1;
    for i = 1 to n - k + 2:
        //Let kMer be an array of characters with size k
        for j = i to k + i:
            kMer[j - i + 1] = DNASequence[j];
            kMers[x] = kMer;
            x++;
        ht_insert(HashTable, i, kMer);
}
```

```
ht_insert(HashTable, key, value)
{
    // Create the bucket
    Bucket* bucket = create_bucket(key, element);

    //Let index be the hashing value obtained from either using the
crc32
    //OR the murmurhash algorithm
    //Let head be the bucket located at index in the hashtable

    if head == NIL:
        head = new Bucket(key, element);
        head.next = NIL;
    else
```

```
        chain(head, new Bucket(key, element));
}
```

```
chain(head, bucket)
{
    while head.next != NIL:
        head = head.next;

    new_node = bucket;
    new_node.next = NIL;
    head.next = new_node;
}
```

```
createHTDistribution (kMers, HashTable, n, k)
{
    //Let kMersCount be an array of integers with the same length as
kMers
    //Initialize kMersCount with all zeroes.

    removed_bucket = ht_remove(table);
    while removed_bucket != NIL:
        for i = 1 to n - k + 2:
            if removed_bucket.element == kMers[i].data:
                kMersCount[i]++;
                break;
        removed_bucket = ht_remove(table);
}
```

```
ht_remove (HashTable)
{
    // Get the linked list with the key
    index = ht_bucket_list_head(HashTable);

    if(index != -1):
        //Let head be the bucket located at index in the hashtable

        old_node = head;
        head = old_node.next;
        old_node.next = NIL;
        return old_node;
    else:
        return NIL;
}
```

```
ht_bucket_list_head(HashTable)
{
    int i;
    For i = 1 to HashTable.size:
        //Let curr be the bucket located at i in the hashtable
        if(curr != NIL):
            return i;

    return -1;
}
```

Figure 1. Hash Table Implementation Algorithm

The Hashtable algorithm implementation is broken down into many different functions. The first function is the main function where it first calls insertkMersHT that is assigned to an array called kMers.

insertkMersHT is responsible for generating a k-mer element from the DNA sequence array which then appends the current k-mer to the array kMers and then inserts the generated k-mer element to the hashtable by calling ht_insert that determines the index where the k-mer will be stored through the use of a hash function. If at the specific index there is no element, then it simply assigns the values to that index. If a collision occurs, however, it will be resolved via chaining, at which point the chain() function is called, linking elements with the same hashed index together in a linked list.

After all the k-mer elements have been assigned to the hashtable, it then goes back to main then proceeds to call createHTDistribution, which calculates the k-mer distribution by taking each bucket in the hashtable and incrementing the number of times a specific kMer was spotted in the Hash Table. To do so, the ht_remove function is called, which takes the bucket spotted at the very start of the hash table. The function then removes that bucket from the hash table, until there are no more elements left.

```
main(BST, DNASequence, n, k)
{
    kMers = insertkMersBST(&BST, bigString, n, k);
    createBSTDistribution(kMers, &BST);
}
```

```
insertkMersBST(Node,DNASequence,n,k){
    //Let kMers be an array of Strings with size k^n
    x = 1;
    for i = 1 to n - k + 2:
        //Let temp be an array of characters with size k
        for j = i to k+i:
            temp[j-i]= DNASequence[j];

        kMers[x]= temp;
        i++;
        Node = insert(Node, temp);
}
```

```
createBSTDistribution(kMers, Node){
    for i = 1 to 16401:
        if kMers[i] != "":
            while search(Node, kMers[i].data) != NULL:
                Node = delete(Node, kMers[i].data);
                kMers[i].count ++;
}
```

```
insert(root, x)
{
    if root == NIL:
        root.key = x;
        root.left = NIL;
        root.right = NIL;
    else if x <= root->data:
        root.left = insert(root.left, x);
    else if x > root->data:
        root.right = insert(root.right, x)

    return root;
}
```

```
search(root, x)
{
    if root == NIL:
        return NIL;
    else if x <= root->data:
        root.left = search(root.left, x);
    else if x > root->data:
        root.right = search(root.right, x);

    return root;
}
```

```
delete(root, x)
{
    if root == NIL:
        return NIL;
    else if x <= root->data:
        root.left = delete(root.left, x);
    else if x > root->data:
        root.right = delete(root.right, x);
    else if root.left != NIL && root.right != NIL:
    //Get the successor of X (case 1: leftmost node at right subtree)
        right = findMin(root.right);
        root.key = right.key;
        root.right = delete(root.right, root.key);
    else:
    //Get the successor of X (case 2: lowest ancestor of X at which x is
a
        left children of the ancestor)
        if root->left == NIL:
            root = root.right;
        else if root->right == NIL:
            root = root.left;

    return root;
}
```

```
findMin(root)
{
    if root == NIL:
        return NIL;
    else if root.left == NIL:
        return root;
    else:
        return findMin(root.left);
}
```

Figure 2. Hash Table Implementation Algorithm

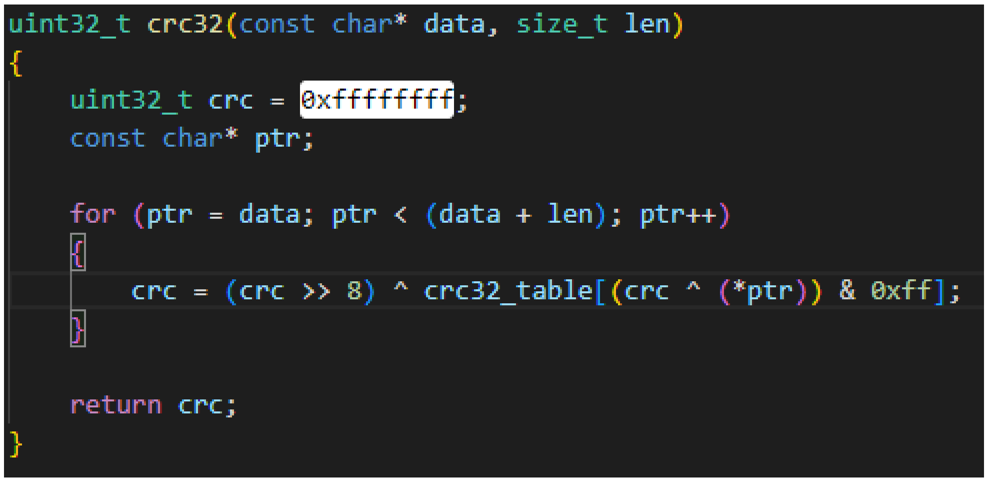
The Binary Search Tree implementation for computing the k-mer distribution is broken down into two algorithms, insertkMersBST, and createBSTDistribution. The first algorithm, insertkMersBST, is in charge of creating the elements that are inserted into the BST by an implemented insert algorithm for BST. The loop iterates through the given DNA sequence and stores each generated k-mer element into a temp string which will then be added to the BST passed through the algorithm as a parameter using the insert function.

Once all k-mer elements have been inserted into the BST, the following algorithm, createBSTDistribution, is used to generate the k-mer distribution. This algorithm accepts a list of non-identical k-mers that also stores its own frequency count. When this algorithm is called, the algorithm iterates through the entire list of k-mers. Inside the loop, another while loop is executed that repeatedly checks if the current k-mer of index i exists in the BST using an implemented search algorithm for BST, and calls an implemented delete algorithm for BST until the search algorithm cannot find the given k-mer. The approach of the group's BST-based implementation is to count the number of deletions of a k-mer in a BST to count the number of occurrences of a k-mer; therefore, whenever the delete function is called, the frequency count of each k-mer gets incremented.

II. Description of the two hashing functions you chose and the reason why we chose them

Both of our chosen hashing functions were adopted from Harvey (2016) during one of his talks regarding non-cryptographic hash functions. He noted how they were useful, even if they are non-cryptographic, as it allows for rapid deployment of hashing. From this, we decided to go for 2 of his mentioned functions, the crc32 and murmurhash3 functions.

CRC32



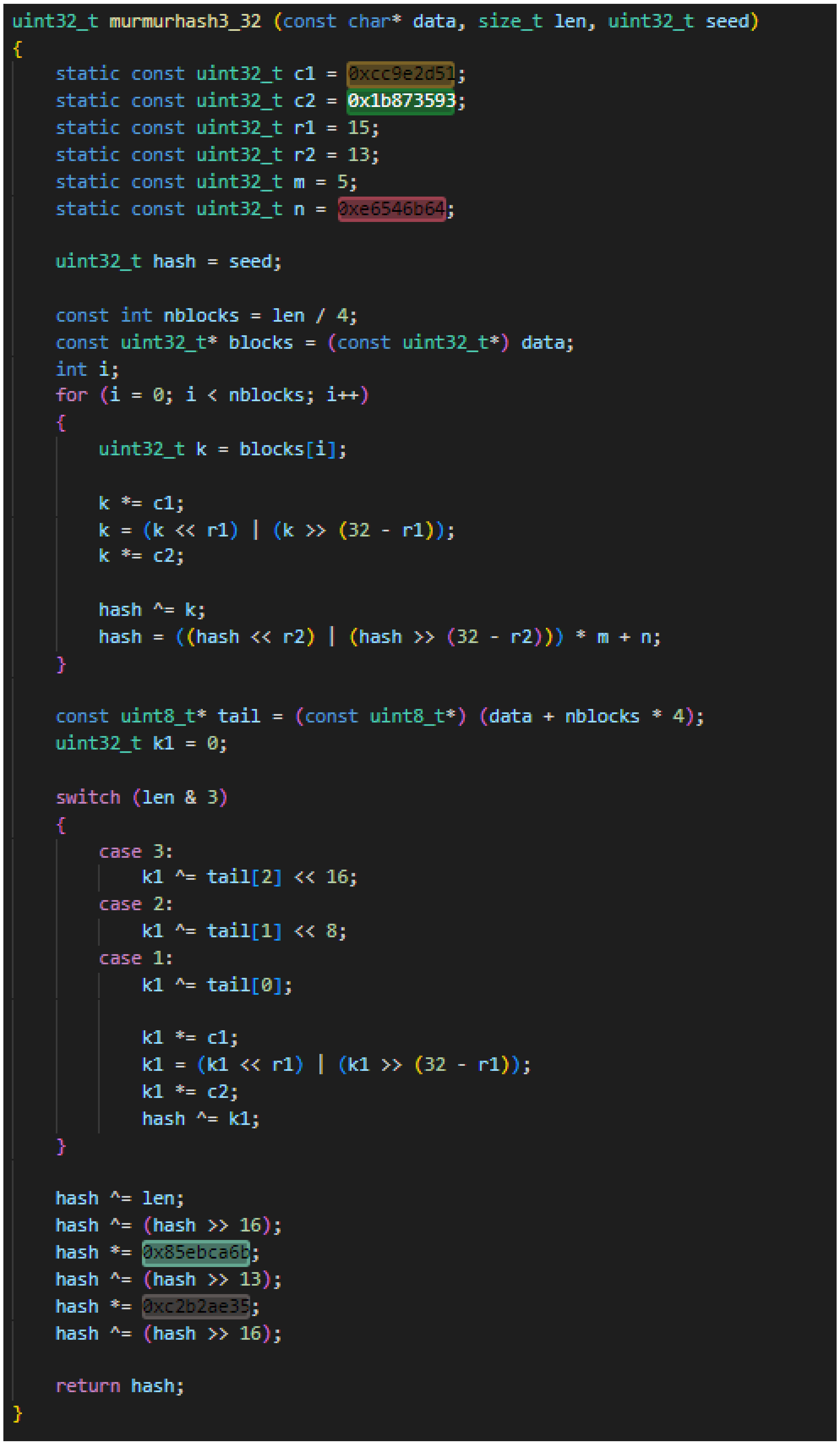
```
uint32_t crc32(const char* data, size_t len)
{
    uint32_t crc = 0xffffffff;
    const char* ptr;

    for (ptr = data; ptr < (data + len); ptr++)
    {
        crc = (crc >> 8) ^ crc32_table[(crc ^ (*ptr)) & 0xff];
    }

    return crc;
}
```

Figure 3. crc32 Hashing Implementation

CRC32 is a simple hashing function that can be used for checking file integrity since the same data input will always return the same hash value (Chung, n.d.). In the context of the program, the function will be used to determine the location of data in the hashtable.

The hash function operates by first taking the *crc* variable and shifts the bits to the right 8 times, dividing its integer equivalent by 2^8. Next, it gets a value from an array of predetermined values of unsigned integers. The value is determined by performing bitwise XOR operations between the initial *crc* value and the integer value of the character currently being referenced in *ptr.* The result of the previous operation is then compared to the integer 255 (0xff) through the bitwise AND operator where the result is then used as the index for the array. Then, a bitwise XOR operation is performed between the adjusted *crc* value and the unsigned integer taken from the array. The whole calculation is repeatedly performed until the *ptr* variable points to the address of the last character in string *data*. Finally, the final value of *crc* is returned as an unsigned 32-bit integer.

Murmurhash3



```c
uint32_t murmurhash3_32 (const char* data, size_t len, uint32_t seed)
{
    static const uint32_t c1 = 0xcc9e2d51;
    static const uint32_t c2 = 0x1b873593;
    static const uint32_t r1 = 15;
    static const uint32_t r2 = 13;
    static const uint32_t m = 5;
    static const uint32_t n = 0xe6546b64;

    uint32_t hash = seed;

    const int nblocks = len / 4;
    const uint32_t* blocks = (const uint32_t*) data;
    int i;
    for (i = 0; i < nblocks; i++)
    {
        uint32_t k = blocks[i];

        k *= c1;
        k = (k << r1) | (k >> (32 - r1));
        k *= c2;

        hash ^= k;
        hash = ((hash << r2) | (hash >> (32 - r2))) * m + n;
    }

    const uint8_t* tail = (const uint8_t*) (data + nblocks * 4);
    uint32_t k1 = 0;

    switch (len & 3)
    {
        case 3:
            k1 ^= tail[2] << 16;
        case 2:
            k1 ^= tail[1] << 8;
        case 1:
            k1 ^= tail[0];

            k1 *= c1;
            k1 = (k1 << r1) | (k1 >> (32 - r1));
            k1 *= c2;
            hash ^= k1;
    }

    hash ^= len;
    hash ^= (hash >> 16);
    hash *= 0x85ebca6b;
    hash ^= (hash >> 13);
    hash *= 0xc2b2ae35;
    hash ^= (hash >> 16);

    return hash;
}
```

Figure 4. murmurhash3 Hashing Implementation

Murmurhash3 is a hashing function generally used for hash-based lookup and relies on two main operations to get its hash value, namely multiplication(MU) and rotation(R) which is where the name murmurhash comes from. (The Apache Software Foundation, n.d.; Harvey, 2016).

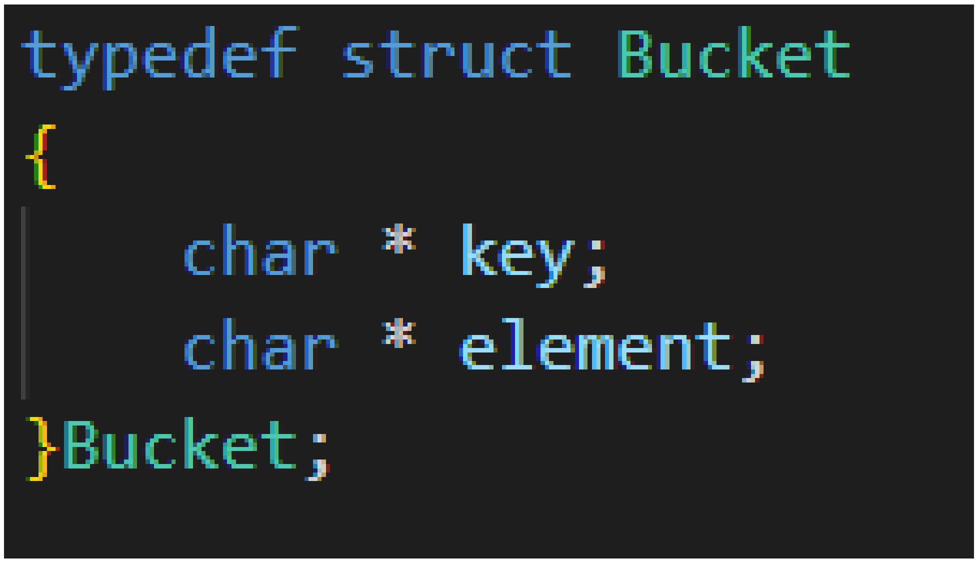
The hash is obtained by dividing the key into 4-byte chunks where multiplication and bitwise operations, such as shifting bits to the left or to the right and XOR, are performed on each chunk. The hash value is then adjusted by performing the XOR operation on each chunk as well as additional operations that involve shifting the hash, using bitwise OR,multiplication and addition. In the cases that the total number of bytes are not divisible by 4, the same operations are performed on the remaining bytes but with some differences where the individual bytes are shifted by (index * 8) bits to the left before being used to adjust the hash values by the same operations used before. The hash value is then further adjusted by more bitwise XOR operations, shifting to the left, and multiplication.

The main reason for the choice of hashing functions is due to their differences in complexity. Out of all the options available, CRC32 was one of the simplest hashing functions to implement and Murmurhash3 was one of the more complicated ones. With the huge difference in complexity, the group wanted to see if the additional complexities of generating a hash value in Murmurhash3 give it an advantage over CRC32 in terms of collision frequency, especially in larger data sets.

III. Implementation of HT-based algorithm

The implementation of the Hashtable-based algorithm starts with the definition of the data structures and functions necessary for the algorithm to work.

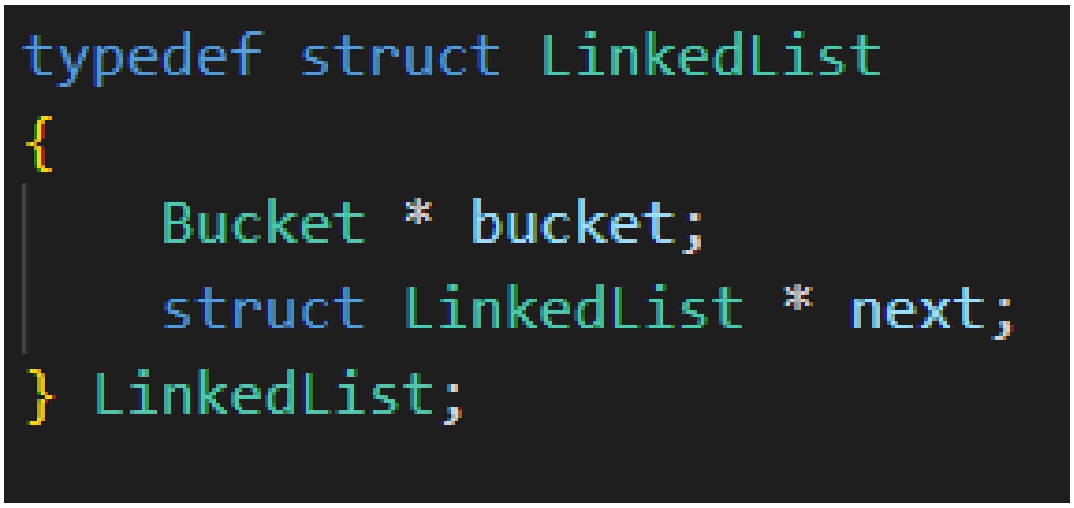
The Bucket data structure contains two character pointer data types that represent the key-element pairs of the data structure where the key is the representation of the index where the element would be stored in the bucket array.



```
typedef struct Bucket
{
    char * key;
    char * element;
}Bucket;
```

Figure 5. HashTable Buckets Structure Implementation

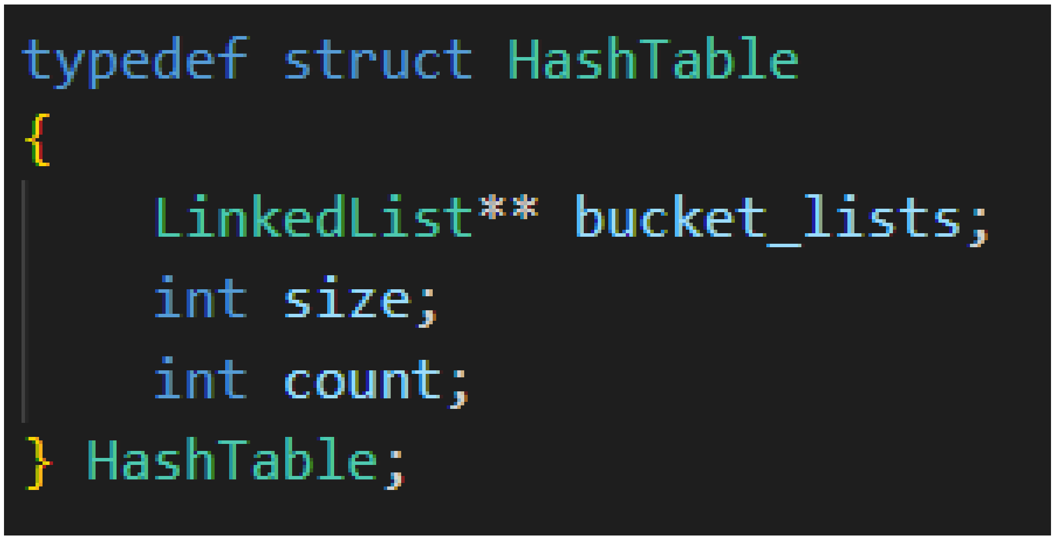
The LinkedList data structure is implemented based on the definition of Singly Linked Lists where it only contains the address of the element, the address of a Bucket data structure in this case, and the address of the next LinkedList data structure. The algorithm only implemented the use of a Singly Linked List since it is only necessary for the algorithm to be able to access the next LinkedList data structure to traverse the LinkedList since it starts from the very first LinkedList pointer in the Hashtable.



```
typedef struct LinkedList
{
    Bucket * bucket;
    struct LinkedList * next;
} LinkedList;
```

Figure 6. HashTable LinkedLists Structure Implementation

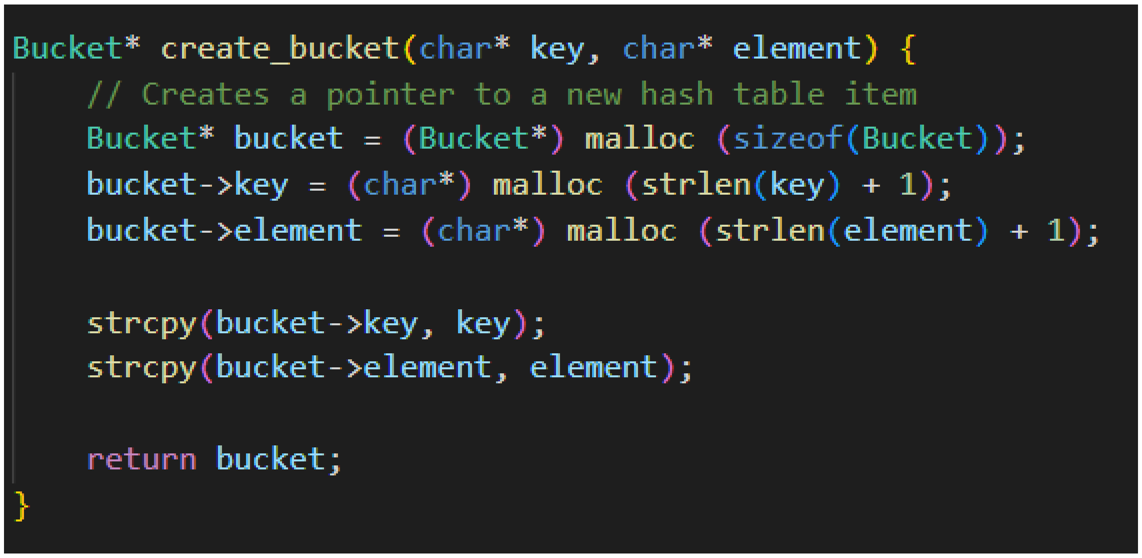
The HashTable data structure is implemented as a dynamically allocated array of bucket_lists where it stores the addresses of the LinkedList structures that are elements of the array. It also stores the size or the capacity of the HashTable data structure and the count or number of LinkedList data structures that are referenced in the HashTable, both in integer data types.



```c
typedef struct HashTable
{
    LinkedList** bucket_lists;
    int size;
    int count;
} HashTable;
```

Figure 7. HashTable Structure Implementation

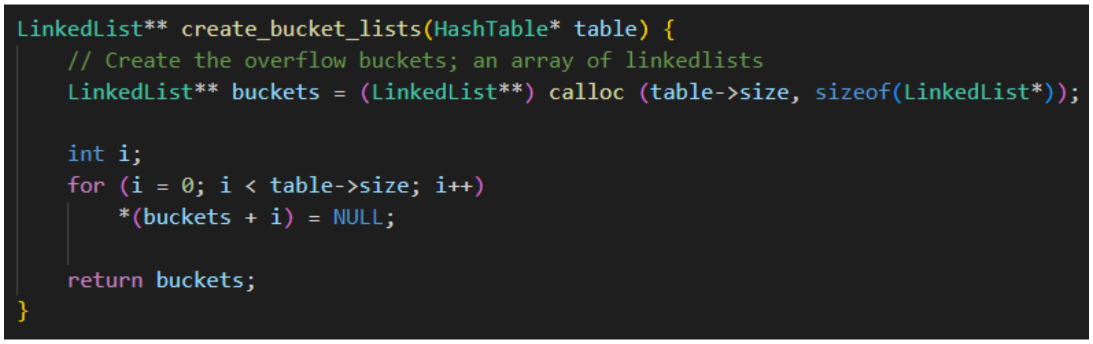
create_bucket() is responsible for creating a Bucket data structure. It starts off by initializing a Bucket and its contents through dynamically allocating memory based on the required size of the data types and structures. The function takes in two character pointer parameters, namely key and element, then copies the values of the parameters to the initialized key and element character pointers in the Bucket. The function ends by returning the memory address of the initialized Bucket.



```c
Bucket* create_bucket(char* key, char* element) {
    // Creates a pointer to a new hash table item
    Bucket* bucket = (Bucket*) malloc (sizeof(Bucket));
    bucket->key = (char*) malloc (strlen(key) + 1);
    bucket->element = (char*) malloc (strlen(element) + 1);

    strcpy(bucket->key, key);
    strcpy(bucket->element, element);

    return bucket;
}
```

Figure 8. Bucket Memory Allocation

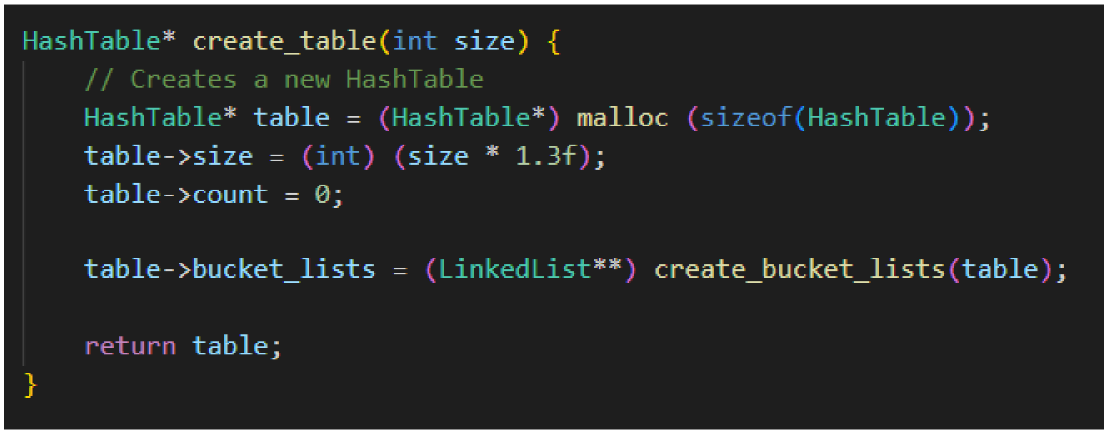
create_bucket_lists() is a function that creates an array of LinkedLists which is responsible for storing the Bucket pointers and represents the overflow buckets where chaining Buckets are stored as a result of collisions. The function initializes the LinkedList pointer array by allocating memory that can contain LinkedLists of the same amount as the size of the HashTable. Each LinkedList is then assigned to NULL to prevent the accessing of garbage values which can lead to potential errors.



```c
LinkedList** create_bucket_lists(HashTable* table) {
    // Create the overflow buckets; an array of linkedlists
    LinkedList** buckets = (LinkedList**) calloc (table->size, sizeof(LinkedList*));

    int i;
    for (i = 0; i < table->size; i++)
        *(buckets + i) = NULL;

    return buckets;
}
```

Figure 9. Bucket Lists Memory Allocation

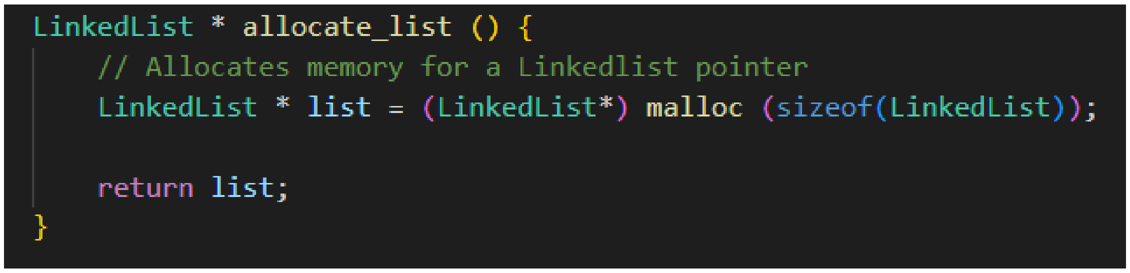
create_table() is responsible for initializing a HashTable and its necessary properties. The function takes in an integer parameter and then assigns that value to the size variable in the Hashtable with a 30% allowance to prevent potential errors (UCSD, n.d.). It then assigns 0 to the count variable, indicating that the HashTable only contains NULL LinkedLists. Finally, the LinkedList array is initialized and the pointer is returned and assigned to the bucket_lists variable.



```c
HashTable* create_table(int size) {
    // Creates a new HashTable
    HashTable* table = (HashTable*) malloc (sizeof(HashTable));
    table->size = (int) (size * 1.3f);
    table->count = 0;

    table->bucket_lists = (LinkedList**) create_bucket_lists(table);

    return table;
}
```

Figure 10. Hash Table Memory Allocation

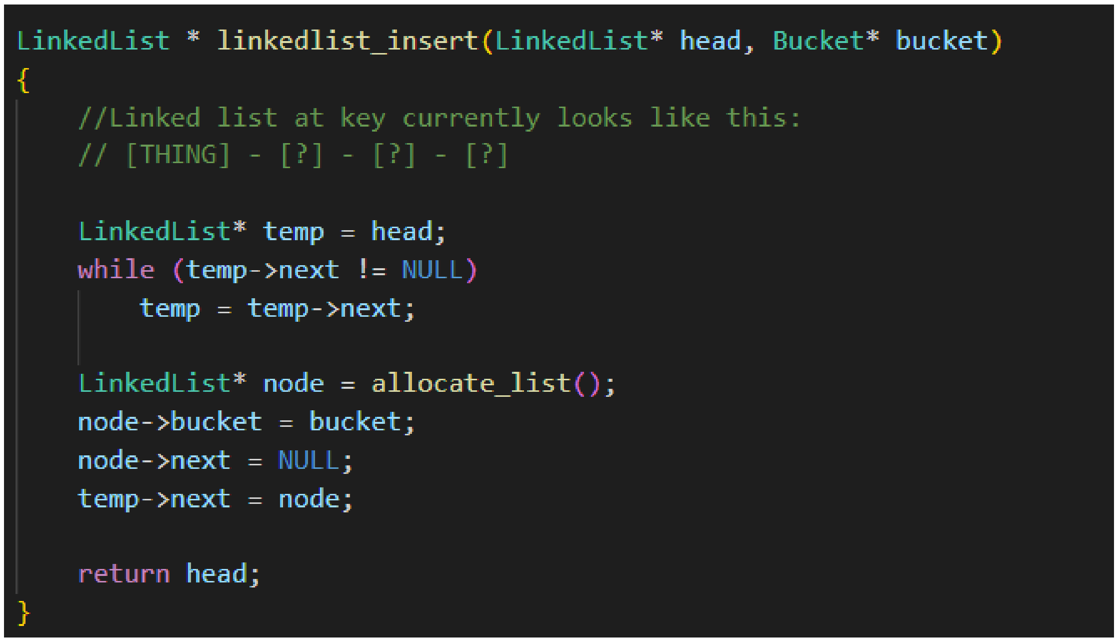
allocate_list() is responsible for allocating memory for storing a LinkedList and returning the pointer of the address where the LinkedList is located.



```c
LinkedList * allocate_list () {
    // Allocates memory for a Linkedlist pointer
    LinkedList * list = (LinkedList*) malloc (sizeof(LinkedList));

    return list;
}
```

Figure 11. Linked Lists Memory Allocation

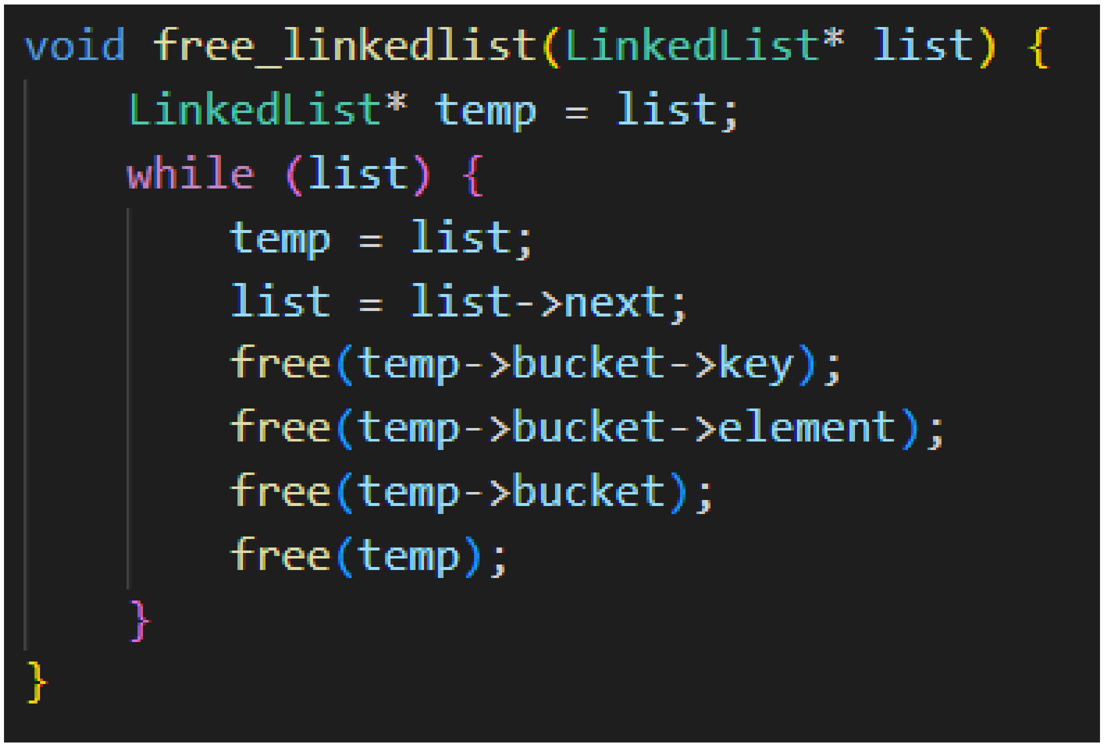
linkedlist_insert() takes in two parameters, a LinkedList pointer called head and a Bucket pointer called bucket. The function assigns the pointer in head to another variable to find the next NULL or empty LinkedList in the LinkedList array before initializing a new LinkedList that would contain the bucket variable and returning the pointer variable referencing to the first LinkedList, or the head.



```c
LinkedList * linkedlist_insert(LinkedList* head, Bucket* bucket)
{
    //Linked list at key currently looks like this:
    // [THING] - [?] - [?] - [?]

    LinkedList* temp = head;
    while (temp->next != NULL)
        temp = temp->next;

    LinkedList* node = allocate_list();
    node->bucket = bucket;
    node->next = NULL;
    temp->next = node;

    return head;
}
```

Figure 12. Linked Lists Insert Function

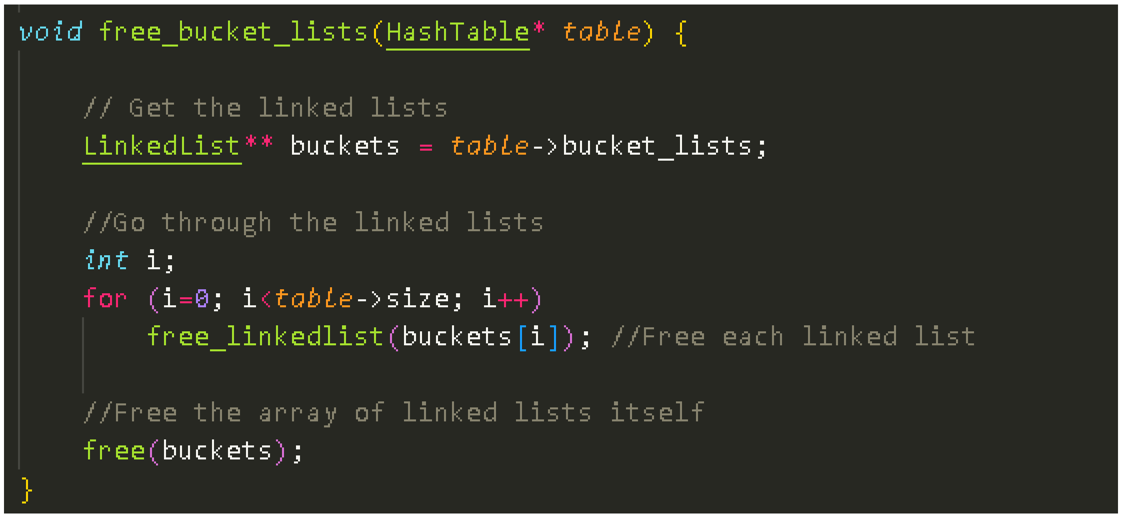
free_linkedlist() takes the pointer of LinkedList as its parameter and deletes the LinkedList and its succeeding LinkedLists by deallocating the memory where it is located.



```c
void free_linkedlist(LinkedList* list) {
    LinkedList* temp = list;
    while (list) {
        temp = list;
        list = list->next;
        free(temp->bucket->key);
        free(temp->bucket->element);
        free(temp->bucket);
        free(temp);
    }
}
```

Figure 13. Linked Lists Free Function

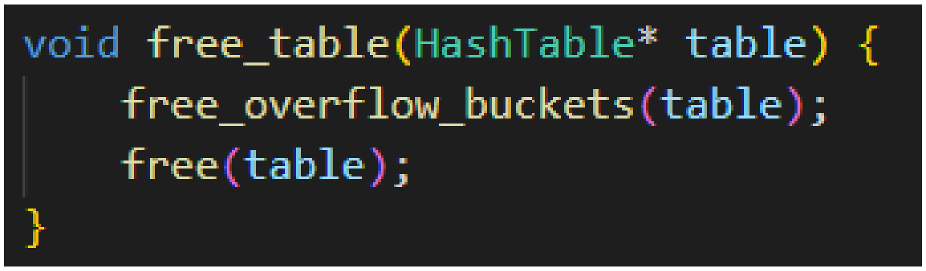
free_bucket_lists() takes a HashTable pointer parameter and deletes all the bucket lists from the HashTable by using free_linkedlist() for the individual buckets in the bucket list and then deallocating the memory for the bucket list afterwards.



```c
void free_bucket_lists(HashTable* table) {

    // Get the linked lists
    LinkedList** buckets = table->bucket_lists;

    //Go through the linked lists
    int i;
    for (i=0; i<table->size; i++)
        free_linkedlist(buckets[i]); //Free each linked list

    //Free the array of linked lists itself
    free(buckets);
}
```

Figure 14. Free Bucket Lists Memory Function

free_table() takes a HashTable pointer parameter and deletes the HashTable as well as all the overflow buckets the HashTable has.

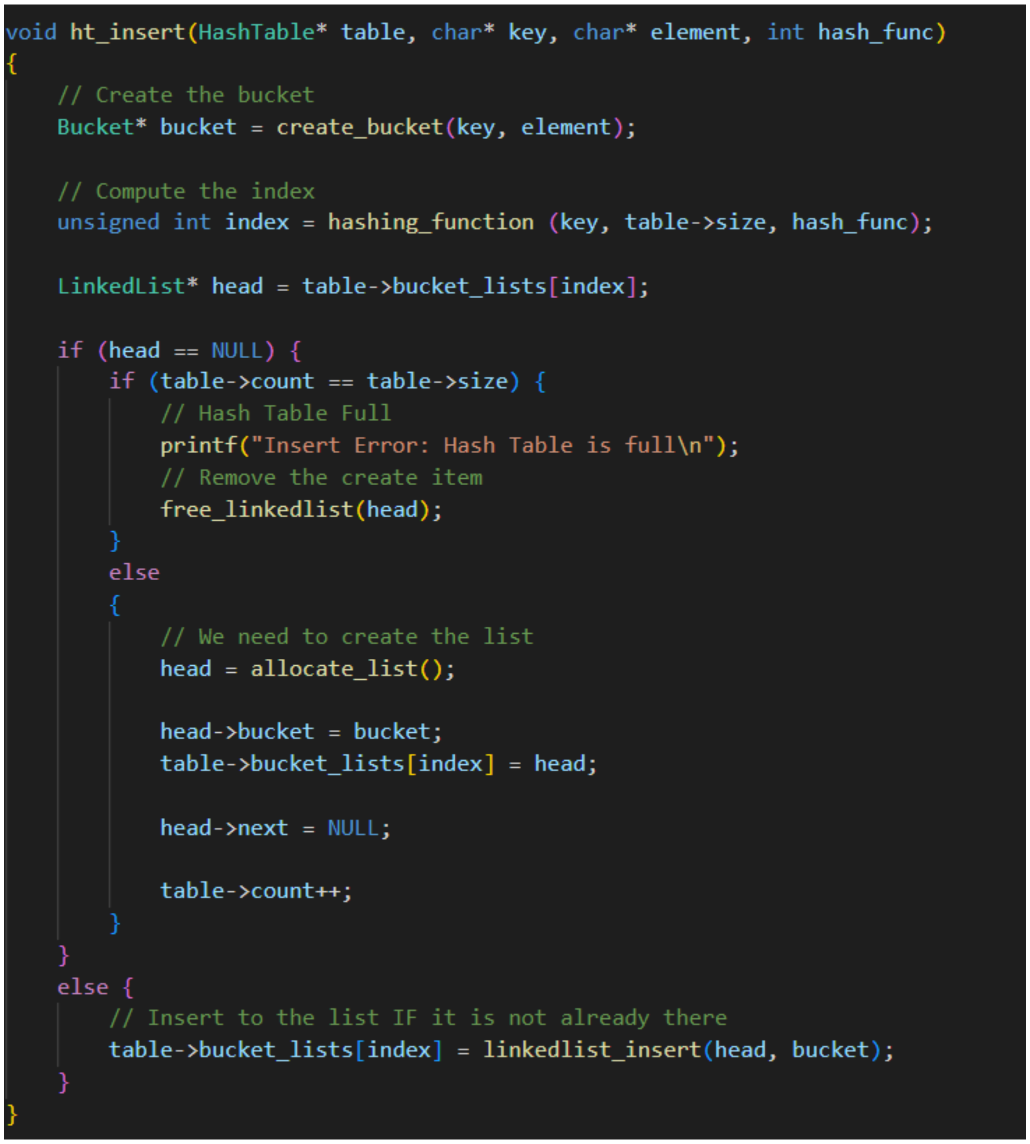


```c
void free_table(HashTable* table) {
    free_overflow_buckets(table);
    free(table);
}
```

Figure 15. Free Table Memory Function

ht_insert() inserts the Bucket into a table where its index is calculated by the hash function. It requires a HashTable pointer variable where the Bucket will be inserted, a character pointer for the key which will determine the index, another character pointer for the element which will be the data that is stored in the index, and an integer hash_func that can only be 1 or 2 and will determine the hashing function that would be used to get the index. It starts off by creating the bucket with the key and element variables, then obtaining the index through a hash function.

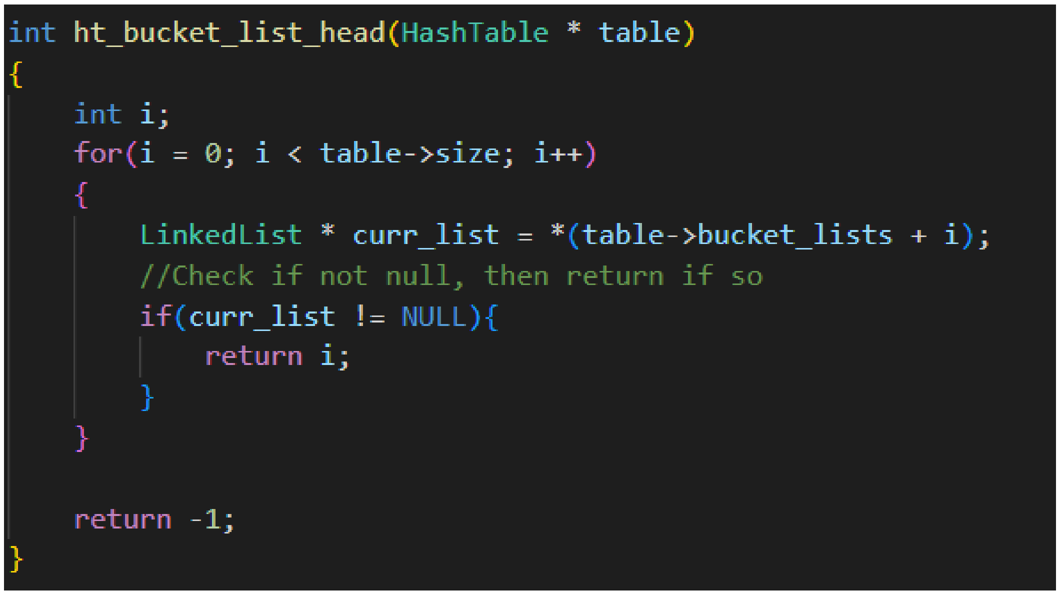
It tries to insert the bucket into the HashTable by checking if the head, or the first LinkedList in the index, is NULL. If the head is NULL, it creates the list first by using allocate_list() before assigning the bucket to the head and the head to the index and updating the count. If head is not NULL, then linkedlist_insert is called to handle the collision.



```c
void ht_insert(HashTable* table, char* key, char* element, int hash_func)
{
    // Create the bucket
    Bucket* bucket = create_bucket(key, element);

    // Compute the index
    unsigned int index = hashing_function (key, table->size, hash_func);

    LinkedList* head = table->bucket_lists[index];

    if (head == NULL) {
        if (table->count == table->size) {
            // Hash Table Full
            printf("Insert Error: Hash Table is full\n");
            // Remove the create item
            free_linkedlist(head);
        }
        else
        {
            // We need to create the list
            head = allocate_list();

            head->bucket = bucket;
            table->bucket_lists[index] = head;

            head->next = NULL;

            table->count++;
        }
    }
    else {
        // Insert to the list IF it is not already there
        table->bucket_lists[index] = linkedlist_insert(head, bucket);
    }
}
```

Figure 16. Hash Table Insert Elements Function

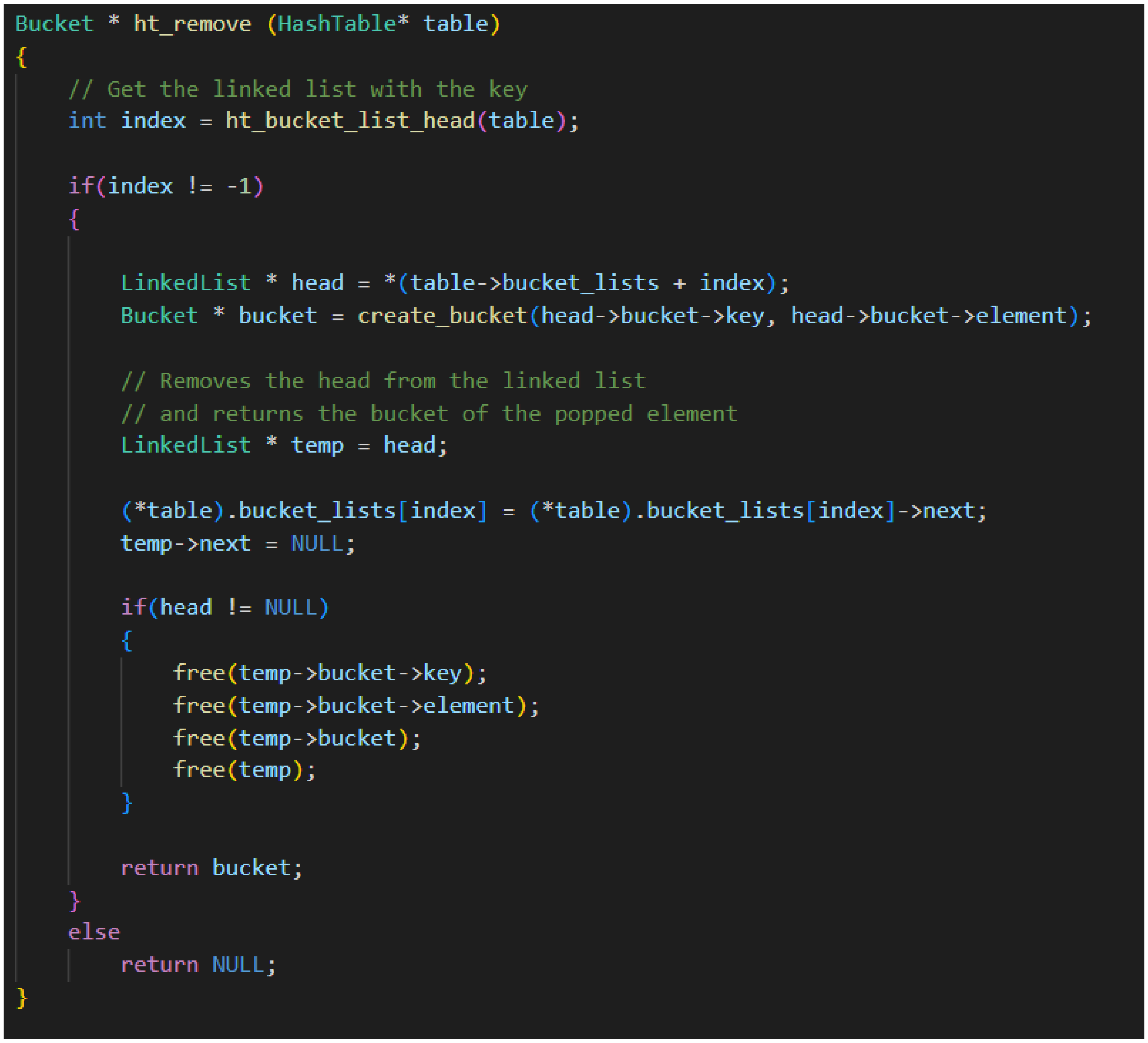
ht_bucket_list_head() checks the hashtable for the linked list with an element that is closest to the start of the hash table and returns the index of the linked list. This is used in conjunction with the ht_remove() function to be able to remove and pop out buckets that can be read later on by the user. Additionally, this function is used to generate the kMer distribution for the user by checking the element inside of the popped out bucket.



```c
int ht_bucket_list_head(HashTable * table)
{
    int i;
    for(i = 0; i < table->size; i++)
    {
        LinkedList * curr_list = *(table->bucket_lists + i);
        //Check if not null, then return if so
        if(curr_list != NULL){
            return i;
        }
    }

    return -1;
}
```

Figure 17. Hash Table List Head Function

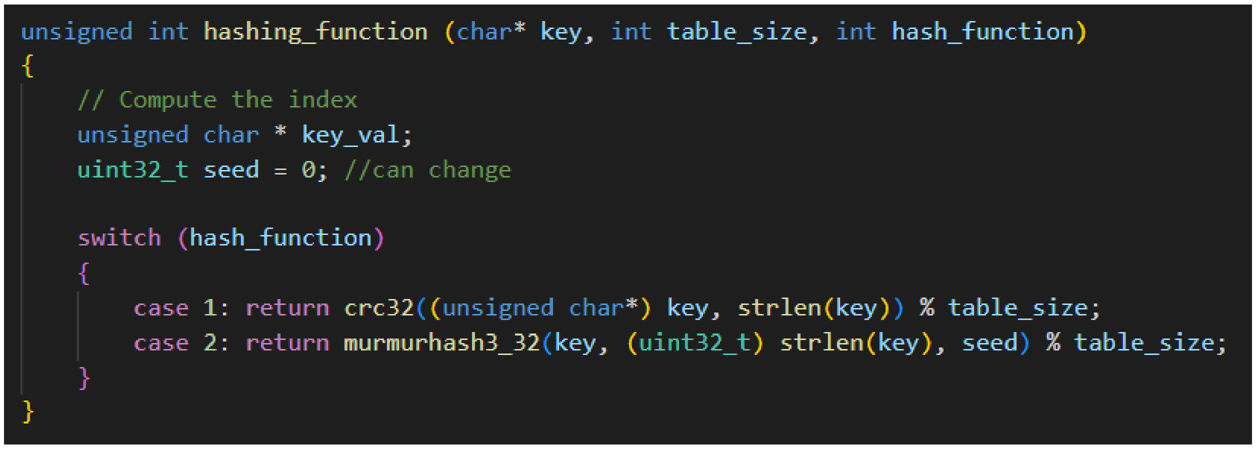
Going more in depth, the ht_remove() function performs its action by first checking if the currently accessed linked list has values. If so, it will then create a new bucket using the values of the currently accessed bucket, and then proceed to remove the currently accessed bucket from its connection to the linked list. This essentially separates the contents of the bucket from the linked list, allowing it to be read and removed from the Hash Table.



```c
Bucket * ht_remove (HashTable* table)
{
    // Get the linked list with the key
    int index = ht_bucket_list_head(table);

    if(index != -1)
    {

        LinkedList * head = *(table->bucket_lists + index);
        Bucket * bucket = create_bucket(head->bucket->key, head->bucket->element);

        // Removes the head from the linked list
        // and returns the bucket of the popped element
        LinkedList * temp = head;

        (*table).bucket_lists[index] = (*table).bucket_lists[index]->next;
        temp->next = NULL;

        if(head != NULL)
        {
            free(temp->bucket->key);
            free(temp->bucket->element);
            free(temp->bucket);
            free(temp);
        }

        return bucket;
    }
    else
        return NULL;
}
```

Figure 18. Hash Table Remove function

The hashing for the hash table is performed by hashing_function(), which returns the hashed index. It uses the key and table size parameters to determine the index and the hash function integer determines if the hashtable uses a CRC32 or a Murmurhash3 hash function. Afterwards, it will then take the result from the chosen hashing function and proceed to take its modulo by the table size, ensuring that it will land inside the hash table.



```c
unsigned int hashing_function (char* key, int table_size, int hash_function)
{
    // Compute the index
    unsigned char * key_val;
    uint32_t seed = 0; //can change

    switch (hash_function)
    {
        case 1: return crc32((unsigned char*) key, strlen(key)) % table_size;
        case 2: return murmurhash3_32(key, (uint32_t) strlen(key), seed) % table_size;
    }
}
```

Figure 19. Hashing Function

IV. Comparison of the collision frequency for the two hashing functions

In the graph below, the HT-based implementation (crc32) and HT-based implementation (murmurhash3) were compared based on their collision frequency when the program was run. It is also indicated in the graph that it only looks at when k = 5 meaning a 5-mer distribution was used in running the program. Now, it can be observed that when n = 10,000 the collision frequency of the two implementations are roughly similar where they are both in the range of 3000. The same can also be said for n = 100,000 where both have a similar collision frequency of around 30,000. Also, when n = 1,000,000 both of their collision frequency are in the same range of 300,000. Lastly, it can also be seen that as the number of elements (n) increases their collision frequency also increases (See Figure 20).

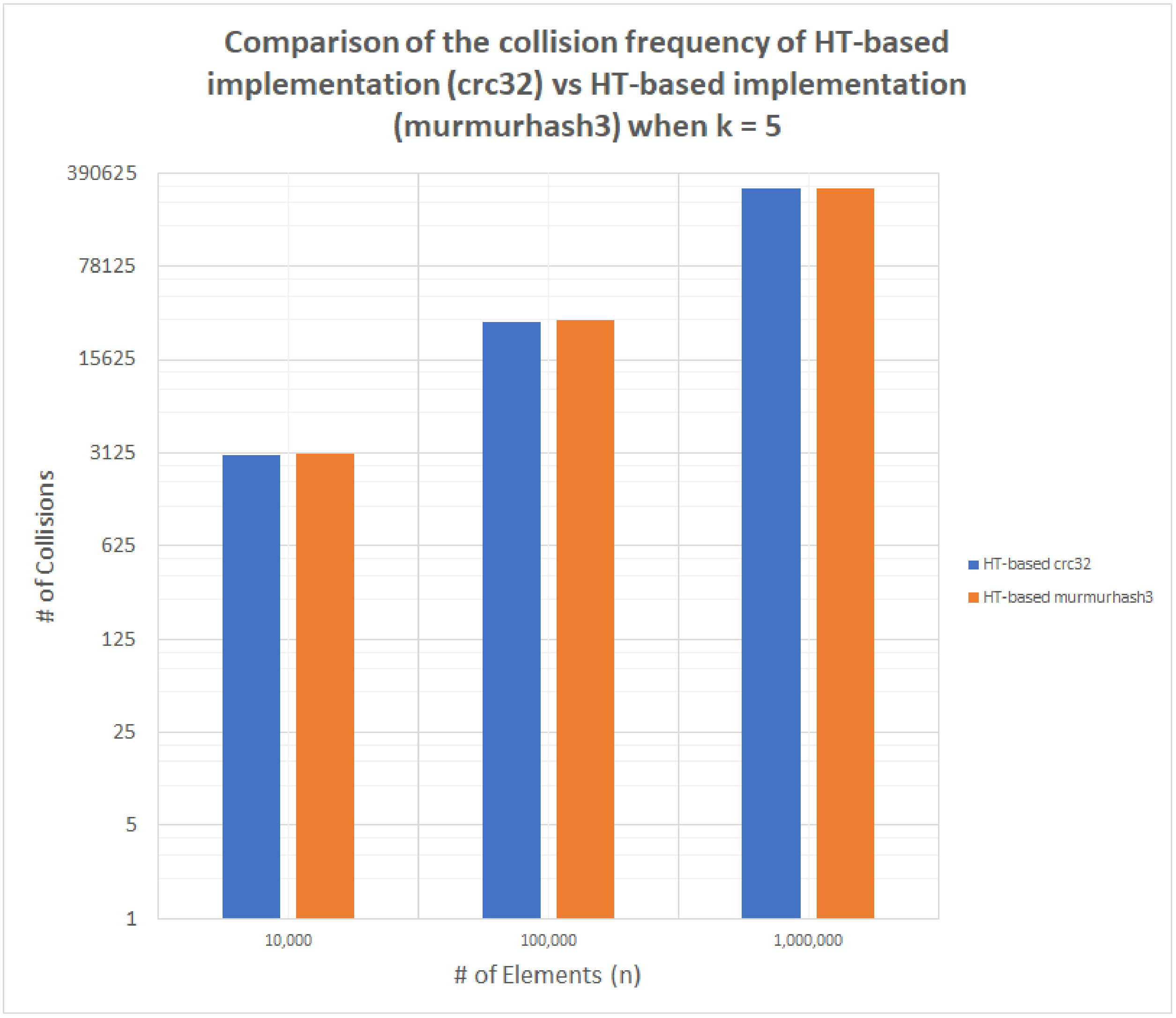


Figure 20. Collision frequency of the two HT-based implementations when k = 5

Moving on, the graph below is also similar to the graph above whereas the only difference was that a different value of k was used. Here it uses k = 6 meaning a 6-mer distribution was used in running the program. Now, comparing the two implementations again it can be seen that when n = 10,000 then their collision frequencies are also similar to that of each other. The same characteristic is also observed whenever n = 100,000 or 1,000,000. Also, both of their collision frequencies again increase when n increases (See Figure 21).

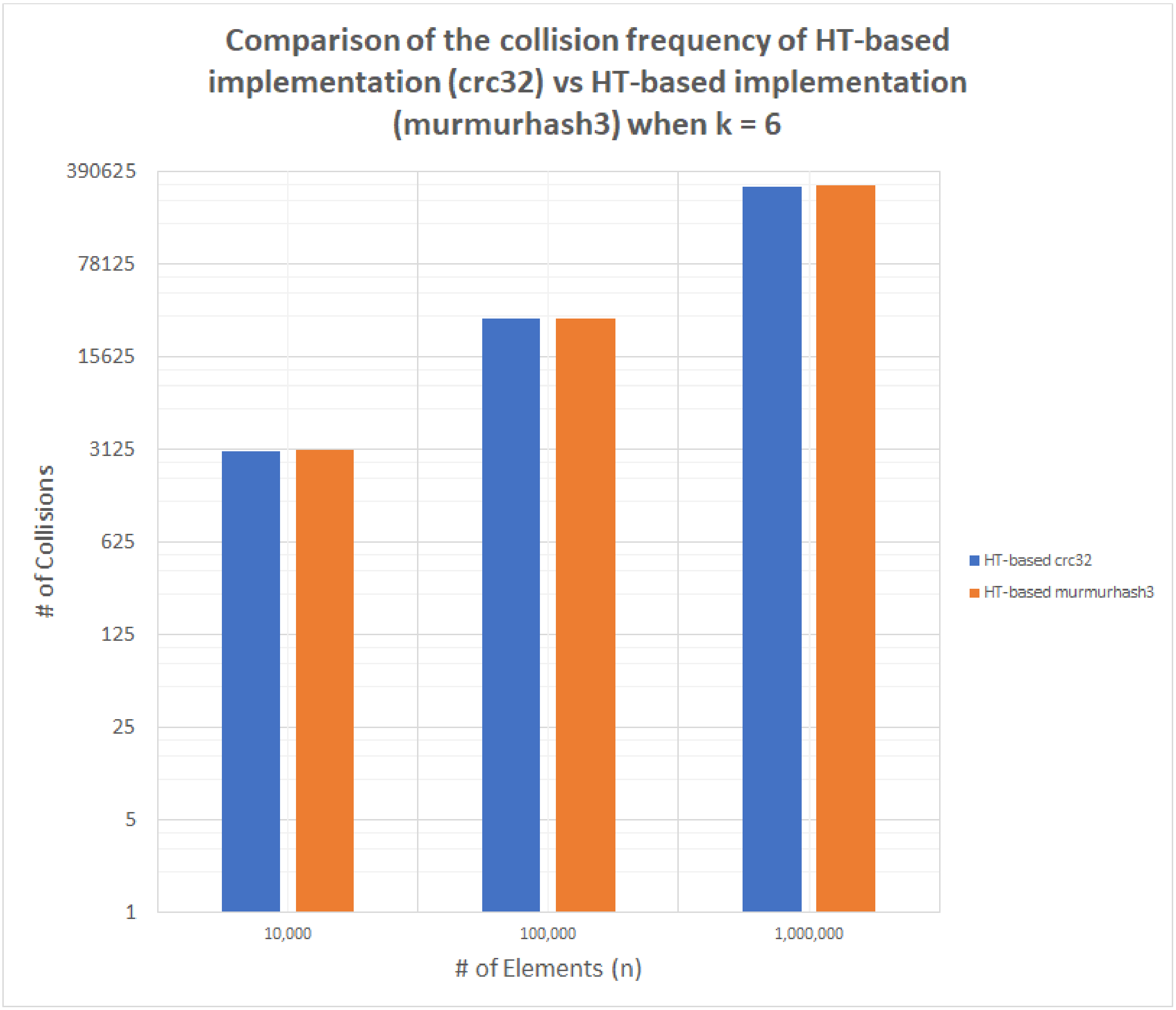


Figure 21. Collision frequency of the two HT-based implementations when k = 6

Lastly, the graph below is also similar to the discussed graphs where only the value of k differs. In this graph, k = 7 meaning a 7-mer distribution was used in running the program. Again, it is observed that when we compare the two implementations their collision frequencies are similar to each other whenever n = 10,000, 100,000 or 1,000,000. Also, it is again observed that as the number of elements (n) increases their collision frequencies also increase even if k = 7 (See Figure 22).

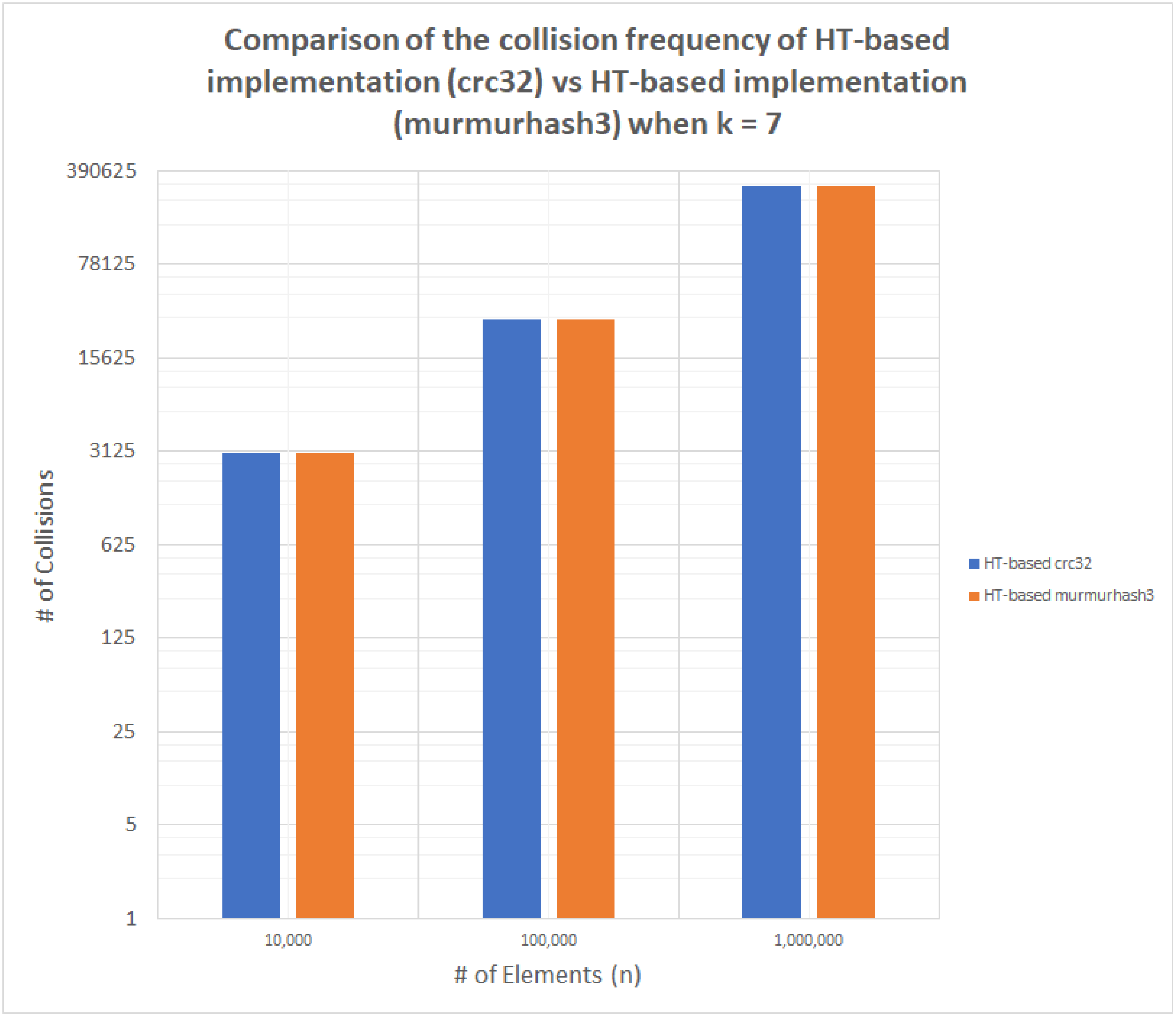
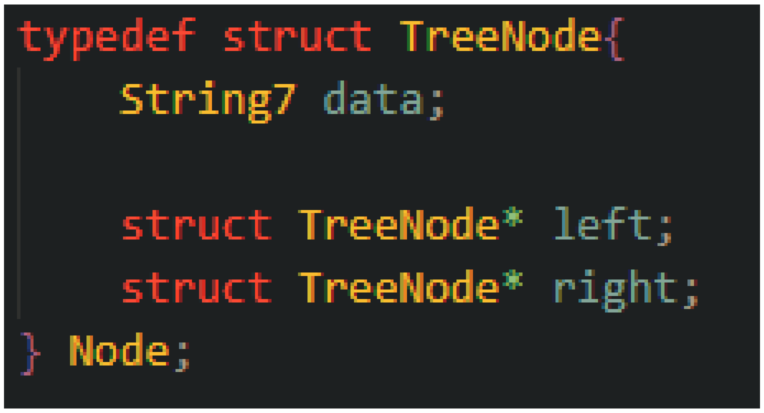


Figure 22. Collision frequency of the two HT-based implementations when k = 7

V. Implementation of BST data structure

To implement a Binary Search Tree (BST)-based k-mer distribution algorithm, we first need to implement the BST data structure from scratch. For the nodes of the BST, a C Structure defined as TreeNode was created that contains the data of the node, which for this implementation, is a character array with a length of 8 defined as String7. The structure as well contains two pointers that point to the addresses of its left and right nodes that are initialized as NULL using the next function to be discussed.

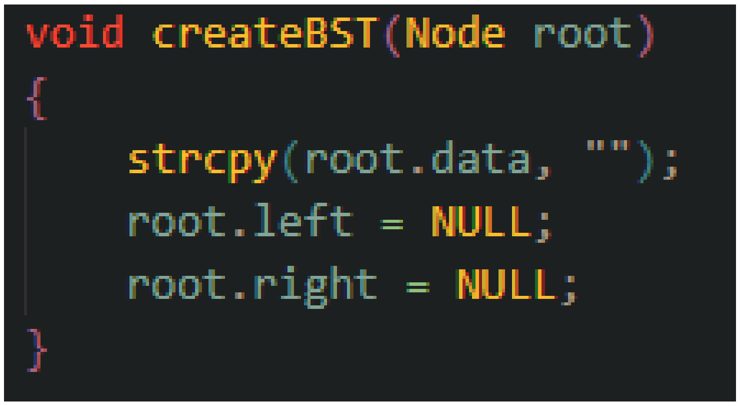


```
typedef struct TreeNode{
    String7 data;

    struct TreeNode* left;
    struct TreeNode* right;
} Node;
```

Figure 23. TreeNode C Structure

In order for the Structure to work properly as a BST, the following operations must be implemented in the program: create(), insert(), search(), and destroy(). Aside from the following functions, the group also implemented delete() and findMin() functions.

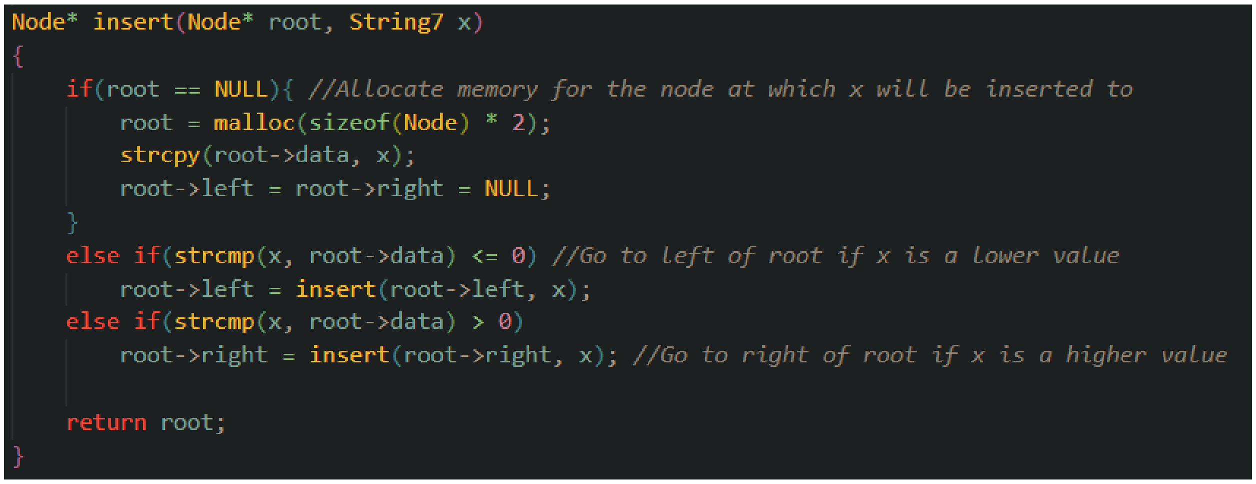
Before doing other BST operations, a BST is initialized through the createBST() function. This function accepts a Node Structure as a parameter, initializes its data to an empty string, and its left and right pointers to NULL.



```
void createBST(Node root)
{
    strcpy(root.data, "");
    root.left = NULL;
    root.right = NULL;
}
```

Figure 24. BST create() function

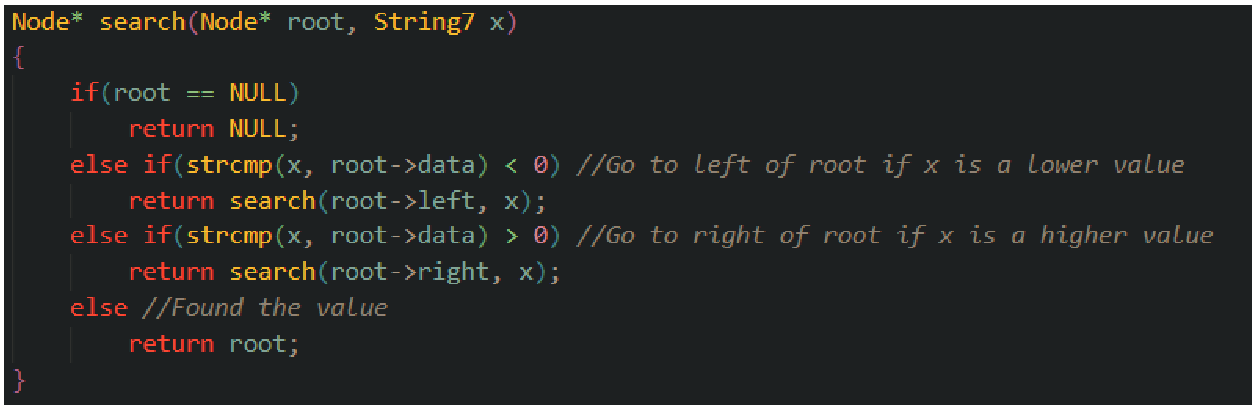
To add more nodes to the BST, an insert() function was created. This function accepts a Node pointer as the root of the BST, and the data that is stored in the node x. The algorithm first checks if the current node pointer points to NULL and if it does, it allocates memory for the size of the TreeNode structure, sets the data of the node to the given data x, and sets its left and right node pointers to NULL. If the pointer passed as a parameter is not equal to NULL, the algorithm will start to search for the proper location of the node that will be created, by following the conditions of a BST, where the value of the left node is less than or equal to the current node, and the value of the right node is greater than the current node. In this case, since the algorithm is comparing strings, the strcmp() function defined in the string.h header file is used to determine the inequality between the two compared strings. The algorithm recursively calls insert() again on either the left or right TreeNode pointer until a NULL pointer is found.



```
Node* insert(Node* root, String7 x)
{
    if(root == NULL){ //Allocate memory for the node at which x will be inserted to
        root = malloc(sizeof(Node) * 2);
        strcpy(root->data, x);
        root->left = root->right = NULL;
    }
    else if(strcmp(x, root->data) <= 0) //Go to left of root if x is a lower value
        root->left = insert(root->left, x);
    else if(strcmp(x, root->data) > 0)
        root->right = insert(root->right, x); //Go to right of root if x is a higher value

    return root;
}
```

Figure 25. BST insert() function

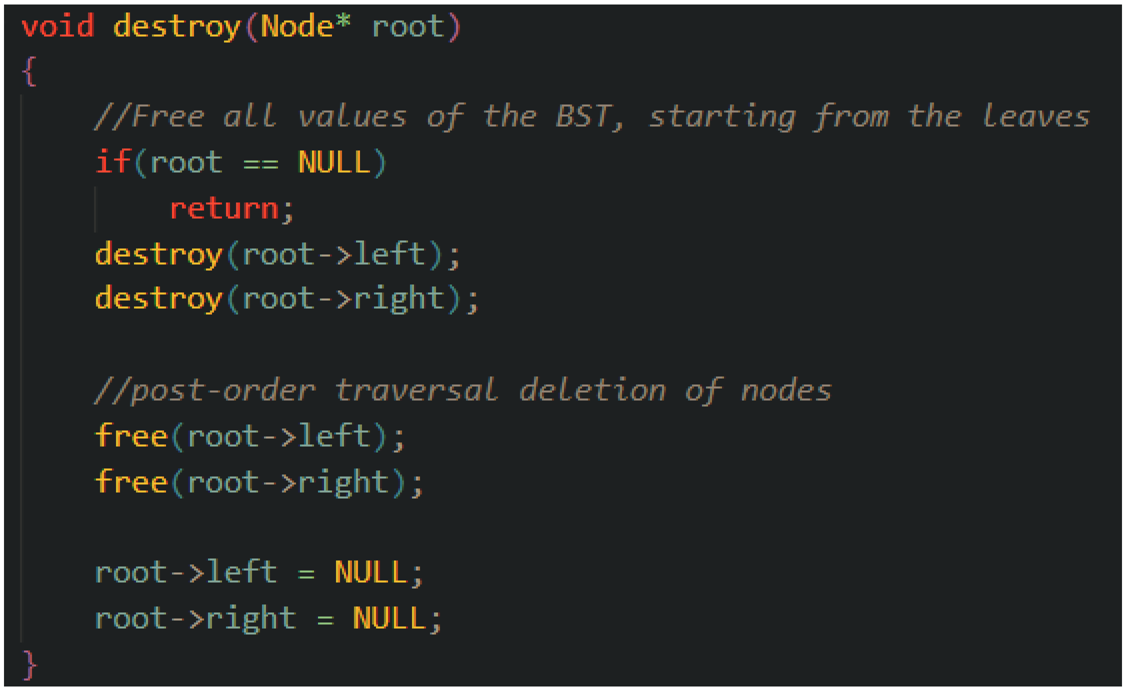
To search for a given string, and check if it exists in the BST, a search() function was created. The algorithm of this function first checks if the TreeNode pointer passed is a NULL pointer. This function will return null if the data does not exist in the BST, or if the BST or subtree is an empty tree. The algorithm traverses through the BST by comparing the strings through strcmp() and recursively calls the function passing either its left or right pointer depending on whether the given string x is less than or greater than the data stored in the current Node.



```
Node* search(Node* root, String7 x)
{
    if(root == NULL)
        return NULL;
    else if(strcmp(x, root->data) < 0) //Go to left of root if x is a lower value
        return search(root->left, x);
    else if(strcmp(x, root->data) > 0) //Go to right of root if x is a higher value
        return search(root->right, x);
    else //Found the value
        return root;
}
```

Figure 26. BST search() function

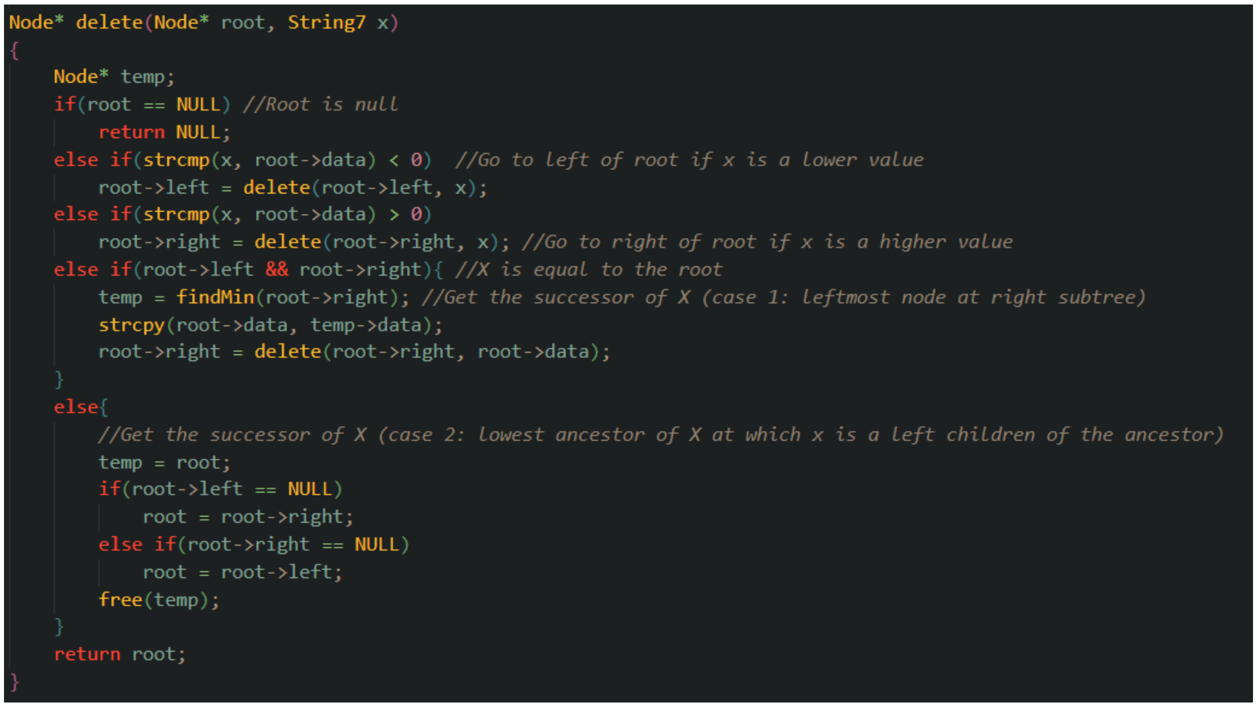
After all changes or actions done with a BST, the destroy() function can be called to remove all nodes linked to a BST, thus freeing memory space. This function deletes nodes in a post-order traversal to delete all the subtrees of a root before deleting itself.



```c
void destroy(Node* root)
{
    //Free all values of the BST, starting from the leaves
    if(root == NULL)
        return;
    destroy(root->left);
    destroy(root->right);

    //post-order traversal deletion of nodes
    free(root->left);
    free(root->right);

    root->left = NULL;
    root->right = NULL;
}
```

Figure 27. BST destroy() function

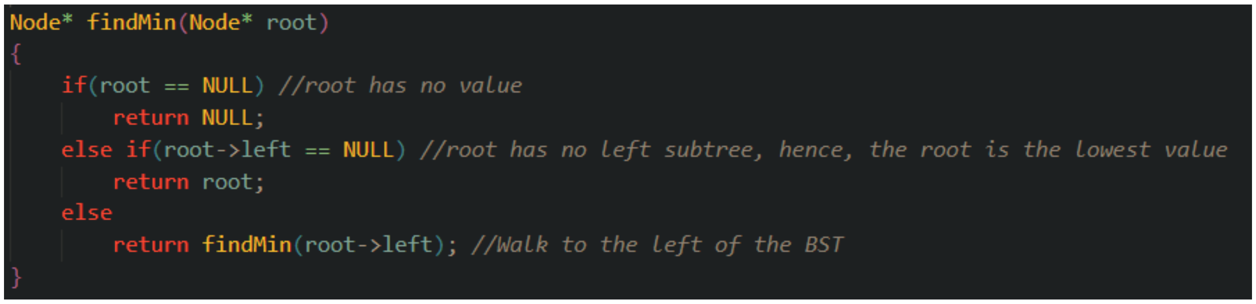
In addition to the given operations, we implemented a delete() function that deletes a node that contains the string equal to the given string passed as a parameter. This is used for an algorithm in the k-mer distribution algorithm that searches and deletes a node that stores a string equal to the given string. This algorithm considers the three cases of deletion wherein the node does not have children, the node has one child, and the node has children. If the node has no children, the function calls findMin() to find its successor, but since the node has no children it will copy a null node stored in temp. If the node has 1 child from either the left or right branch, its left or right child will be stored in the node itself. Lastly, if the node has a child on both sides, the function will calls findMin() similar to the first case, but in this case, find an actual successor that will replace the current node and call delete() recursively to delete the successor that replaced the current node.



```
Node* delete(Node* root, String7 x)
{
    Node* temp;
    if(root == NULL) //Root is null
        return NULL;
    else if(strcmp(x, root->data) < 0)  //Go to left of root if x is a lower value
        root->left = delete(root->left, x);
    else if(strcmp(x, root->data) > 0)
        root->right = delete(root->right, x); //Go to right of root if x is a higher value
    else if(root->left && root->right){ //X is equal to the root
        temp = findMin(root->right); //Get the successor of X (case 1: leftmost node at right subtree)
        strcpy(root->data, temp->data);
        root->right = delete(root->right, root->data);
    }
    else{
        //Get the successor of X (case 2: lowest ancestor of X at which x is a left children of the ancestor)
        temp = root;
        if(root->left == NULL)
            root = root->right;
        else if(root->right == NULL)
            root = root->left;
        free(temp);
    }
    return root;
}
```

Figure 28. BST delete() function

This function uses a utility function defined as findMin() to retrieve the leftmost node from the given tree/subtree. It is used for obtaining the successor that will replace the node to be deleted by using findMin() on the right subtree of the given node.

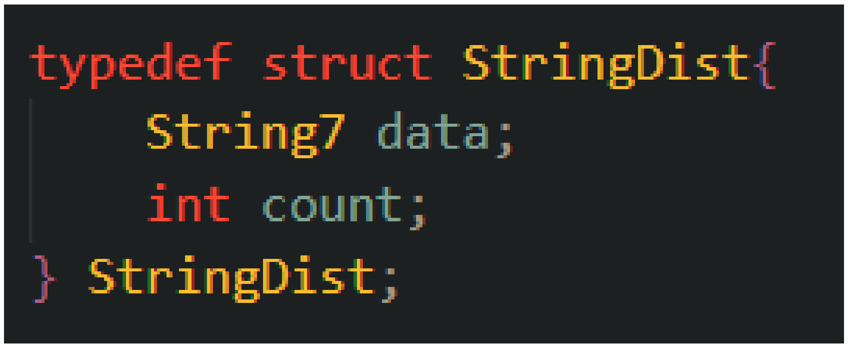


```
Node* findMin(Node* root)
{
    if(root == NULL) //root has no value
        return NULL;
    else if(root->left == NULL) //root has no left subtree, hence, the root is the lowest value
        return root;
    else
        return findMin(root->left); //Walk to the left of the BST
}
```

Figure 29. BST implementation of findMin()

VI. Implementation of BST-based k-mer distribution algorithm

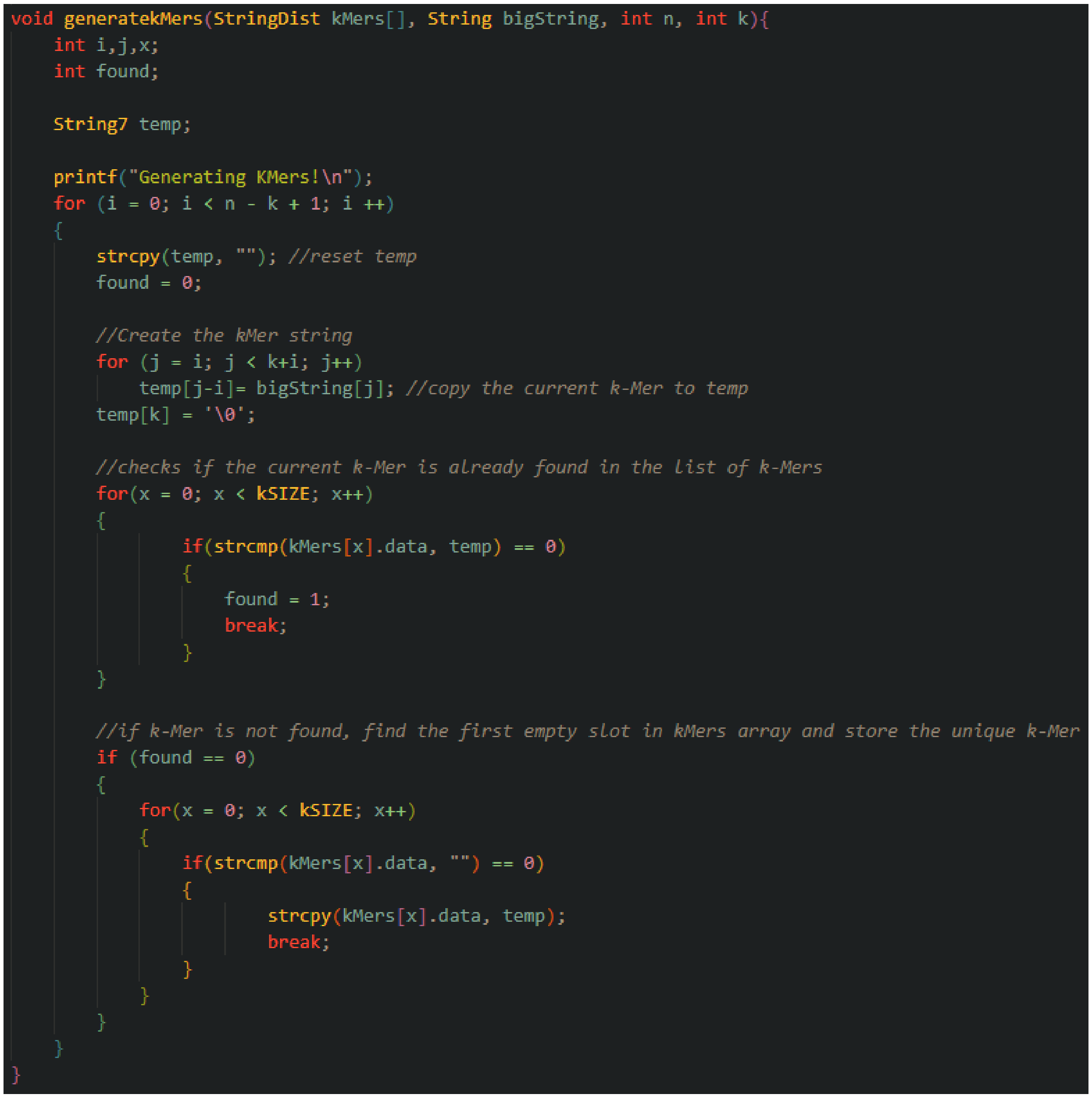
To create a BST-based k-mer distribution algorithm, the aforementioned BST implementation and its functions were used. Firstly, the k-mers will be stored in another C Structure defined as StringDist. This structure contains a k-mer and a counter that stores the number of occurrences in the generated DNA sequence.



```c
typedef struct StringDist{
    String7 data;
    int count;
} StringDist;
```

Figure 30. StringDist C Structure for storing k-mer occurrences

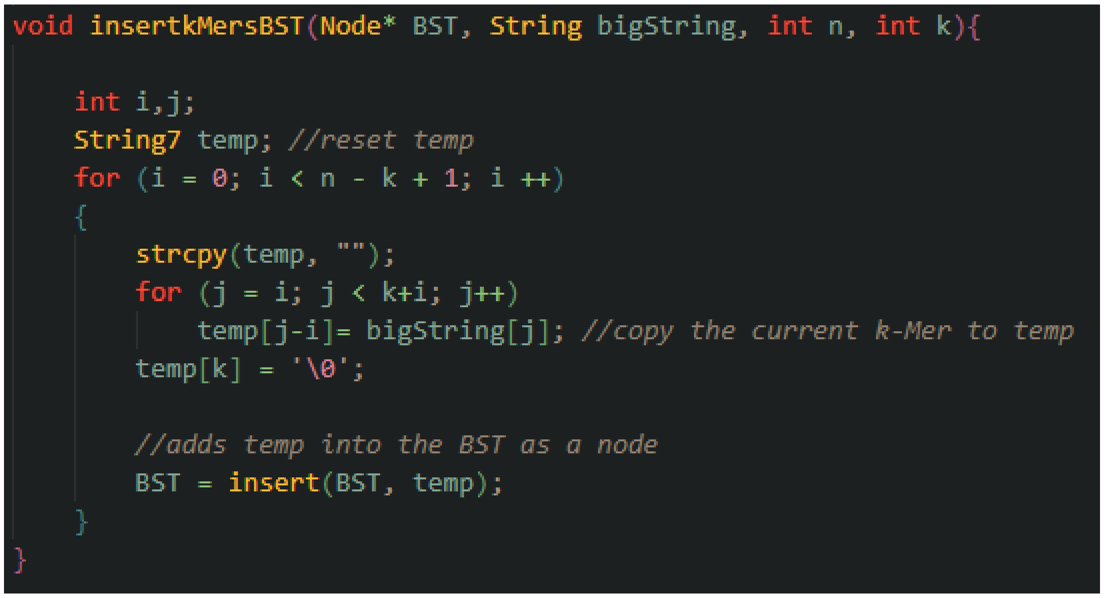
At the beginning of the program's execution, an array of StringDist structures with a length of 16400 is declared. This length is computed by solving for the maximum number of 7-mers that can be generated with the given set of characters S = {a, c, g, t}. To create a list of k-mers without repetition, a utility function called generatekMers() was implemented to add the k-mers generated from the given DNA sequence into the array of StringDist structures.



```c
void generatekMers(StringDist kMers[], String bigString, int n, int k){
    int i,j,x;
    int found;

    String7 temp;

    printf("Generating KMers!\n");
    for (i = 0; i < n - k + 1; i ++)
    {
        strcpy(temp, ""); //reset temp
        found = 0;

        //Create the kMer string
        for (j = i; j < k+i; j++)
            temp[j-i]= bigString[j]; //copy the current k-Mer to temp
        temp[k] = '\0';

        //checks if the current k-Mer is already found in the list of k-Mers
        for(x = 0; x < kSIZE; x++)
        {
                if(strcmp(kMers[x].data, temp) == 0)
                {
                    found = 1;
                    break;
                }
        }

        //if k-Mer is not found, find the first empty slot in kMers array and store the unique k-Mer
        if (found == 0)
        {
            for(x = 0; x < kSIZE; x++)
            {
                if(strcmp(kMers[x].data, "") == 0)
                {
                        strcpy(kMers[x].data, temp);
                        break;
                }
            }
        }
    }
}
```

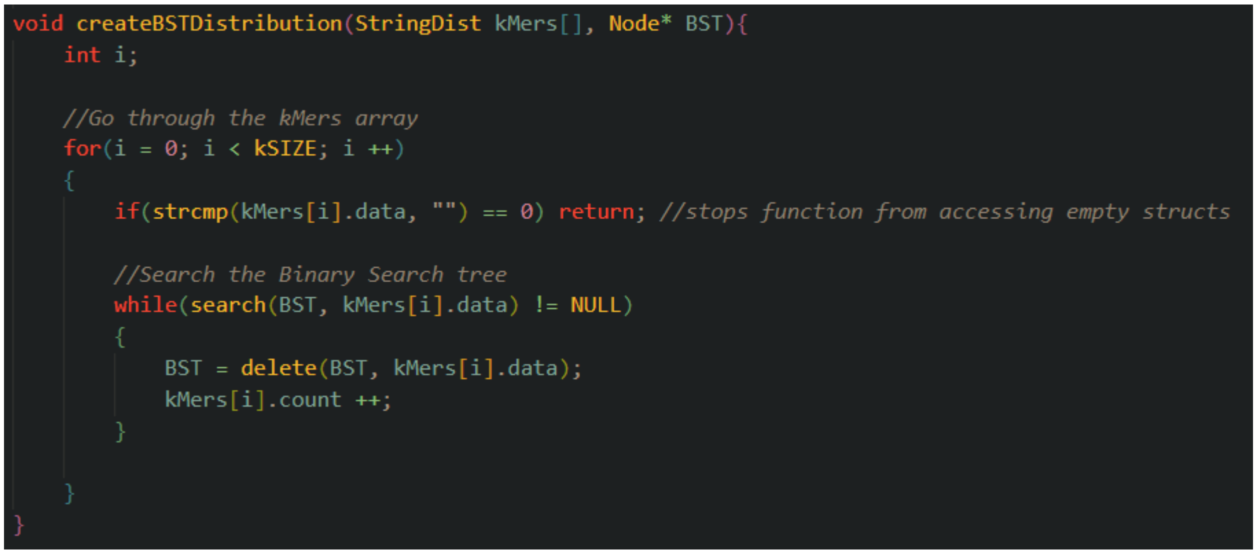
Figure 31. generatekMers() function to create a list of unique k-mers for counting occurrences

Once the StringDist array is initialized, the program will then start creating the BST by inserting all of the generated k-mer substrings using the created insert() function. To count the number of occurrences each unique k-mer has, a for loop will iterate through each StringDist and repeatedly uses the search() and delete() function while incrementing the count variable of each structure, until the search() function returns a NULL pointer, indicating that there are no more occurrences of the currently iterated k-mer in the BST. The algorithm will continue to loop until all unique k-mers have been tested.



```c
void insertkMersBST(Node* BST, String bigString, int n, int k){

    int i,j;
    String7 temp; //reset temp
    for (i = 0; i < n - k + 1; i ++)
    {
        strcpy(temp, "");
        for (j = i; j < k+i; j++)
            temp[j-i]= bigString[j]; //copy the current k-Mer to temp
        temp[k] = '\0';

        //adds temp into the BST as a node
        BST = insert(BST, temp);
    }
}
```

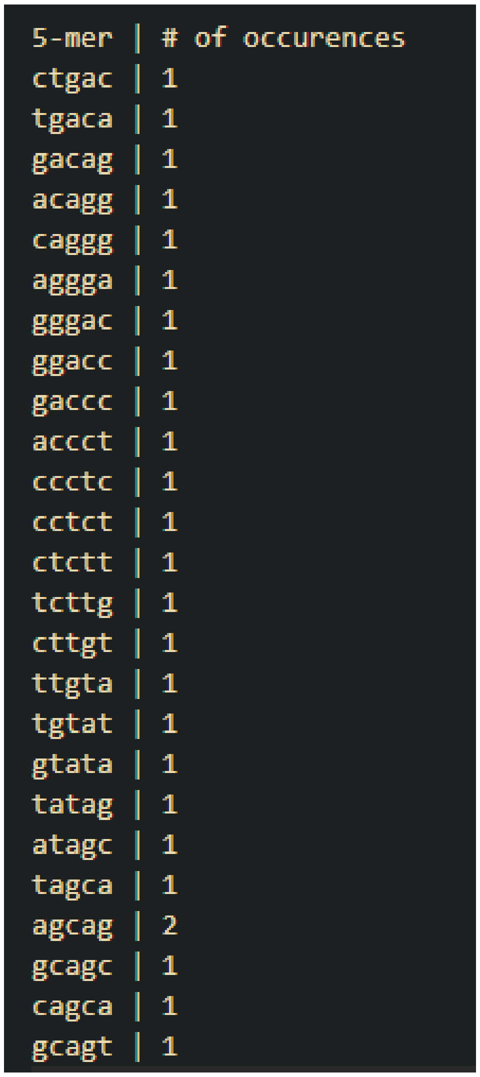
Figure 32. insertkMersBST() function for inserting all k-mer substrings into a BST



```c
void createBSTDistribution(StringDist kMers[], Node* BST){
    int i;

    //Go through the kMers array
    for(i = 0; i < kSIZE; i ++)
    {
        if(strcmp(kMers[i].data, "") == 0) return; //stops function from accessing empty structs

        //Search the Binary Search tree
        while(search(BST, kMers[i].data) != NULL)
        {
            BST = delete(BST, kMers[i].data);
            kMers[i].count ++;
        }

    }
}
```

Figure 33. createBSTDistribution() function for generating the number of occurrences per unique k-mer

Lastly, the output is written into a text file that shows the entire k-mer distribution. This method of output is chosen to store the entire distribution because using printf() to display the entire distribution on the terminal cannot support a large data set. An example of the text file is shown below (refer to Figure 34).



```
5-mer | # of occurences
ctgac | 1
tgaca | 1
gacag | 1
acagg | 1
caggg | 1
aggga | 1
gggac | 1
ggacc | 1
gaccc | 1
accct | 1
ccctc | 1
cctct | 1
ctctt | 1
tcttg | 1
cttgt | 1
ttgta | 1
tgtat | 1
gtata | 1
tatag | 1
atagc | 1
tagca | 1
agcag | 2
gcagc | 1
cagca | 1
gcagt | 1
```

Figure 34. Example of k-mer distribution on a simple test data set n = 30, k =5

VII. Comparison of running time of the two HT-based implementation and the BST-based implementation

As seen in the graph below, the BST-based implementation, HT-based implementation(crc32), and HT-based implementation(murmurhash3) are all compared to each other based on their running time of the program. It is also important to take note that the graph illustrates their run times based on the value of k stated. So, since k = 5 means that a 5-mer distribution was used. Now, it can be observed that the BST implementation runs/executes the fastest because it has the lowest amount of seconds taken by 0.005528s when n = 10,000. It also has the lowest times whenever n = 100,000 or 1,000,000 with the values of 0.190892s and 43.147534s respectively. For the two HT-based implementations, it can be seen that both have similar run times as n increases. This is seen when n = 10,000 where both have a record of around 0.17s - 0.20s and continue to be close to each other as n increases. It is also observed that as n reaches 1,000,000 then the HT implementation (murmurhash3) had a slightly longer running time than its other HT implementation (crc32) making it the longest to run amongst the three. Lastly, it is evident that as the number of elements (n) increase then their running time also increases (See Figure 35).

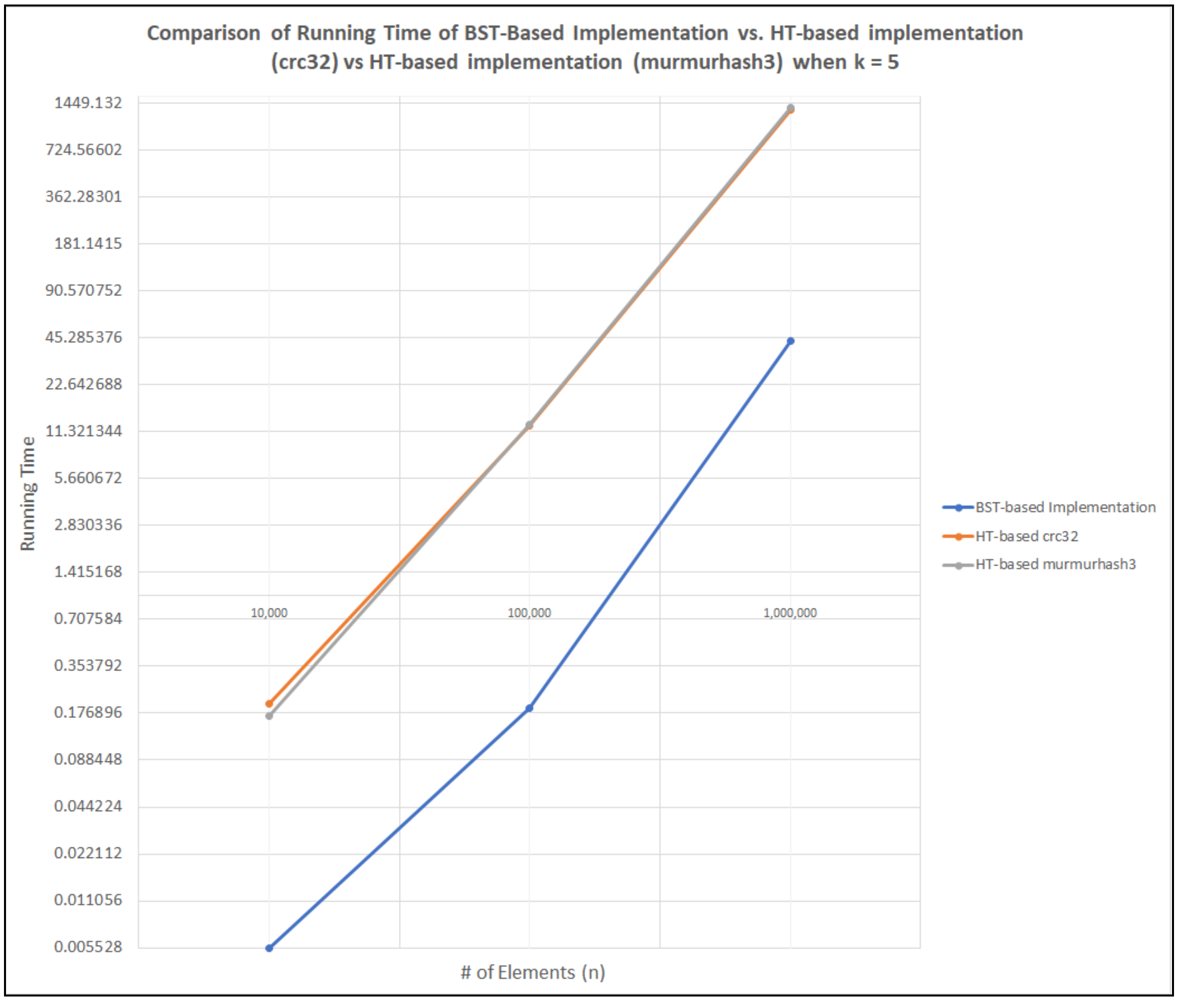


Figure 35. Running Time of the two HT-based implementation and the BST-based implementation when k = 5

The next graph below is also similar to the one discussed above where the only difference is the value of k. Here k = 6 meaning a 6-mer distribution was used in the program. So, since the graph below is closely similar with the previous graph then it also shares similar observations. It can be seen that the BST implementation is still by far the fastest to execute/run amongst the three with a record run time of 0.00769s when n = 10,000. It is also observed that when n = 1,000,000 then the running time of the BST implementation has gone down to 10.592747s which is lower than the value in its previous graph. Furthermore, it is again observed that the running time of the HT implementations are closely similar to each other. Also, the HT implementation (murmurhash3) has again a slightly longer running time (1476.91695s) than its HT counterpart which makes it the longest to run (See Figure 36).

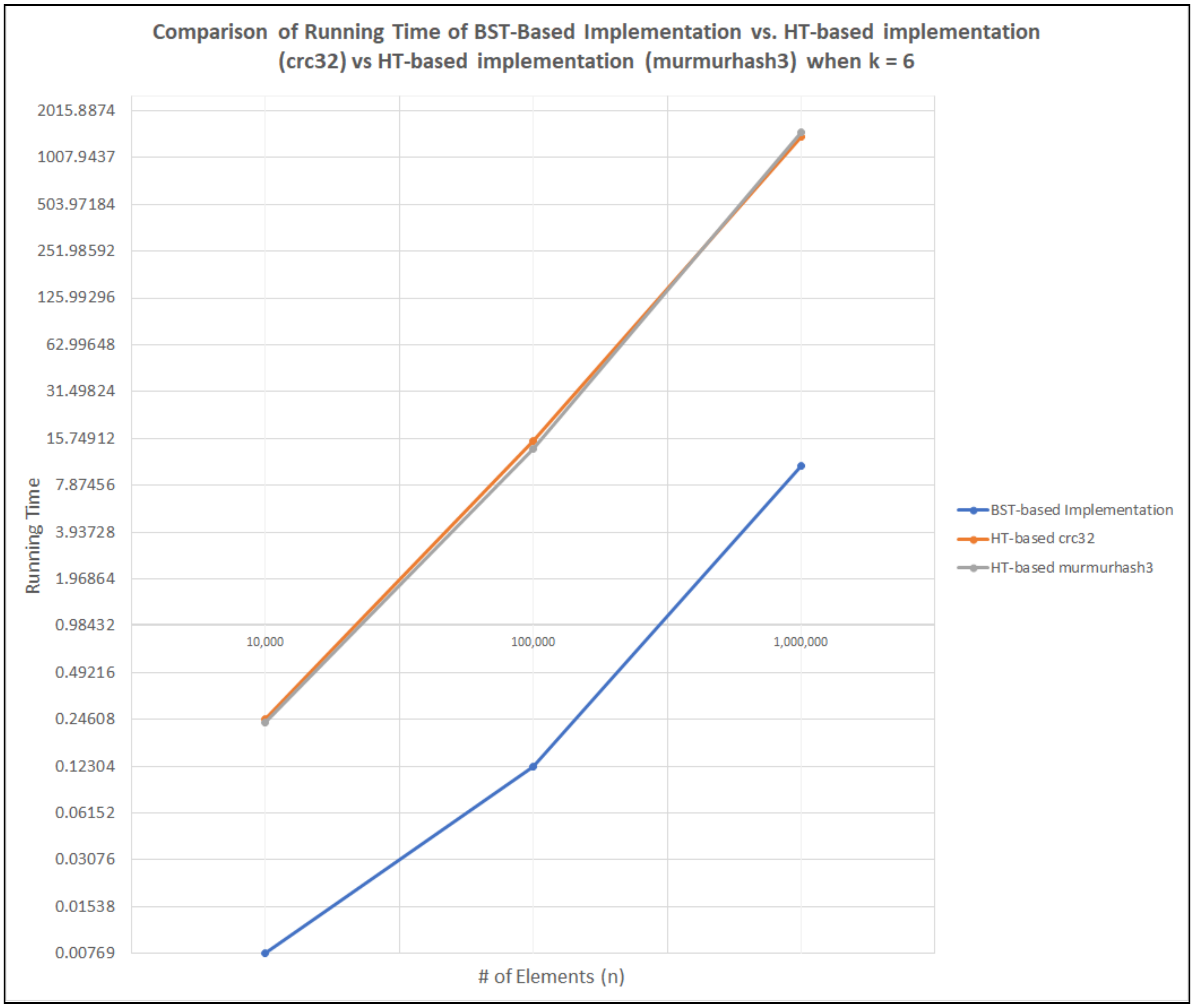


Figure 36. Running Time of the two HT-based implementation and the BST-based implementation when k = 6

Lastly, the graph below is again similar to the discussed graphs above where it only differs in the value of k. Here k = 7 meaning a 7-mer distribution was used. Now, even though the value of k is 7 its graph is still similar to that of the ones above making it have the same observations. So, it can still be seen that the BST implementation is the fastest with a record of 0.010919s when n = 10,000. The two HT implementations both again have similar run times as n increases. But as n reaches 1,000,000 the HT implementation (murmurhash3) has a slightly longer run time of 1472.206517s making it the longest to run amongst the three (See Figure 37).

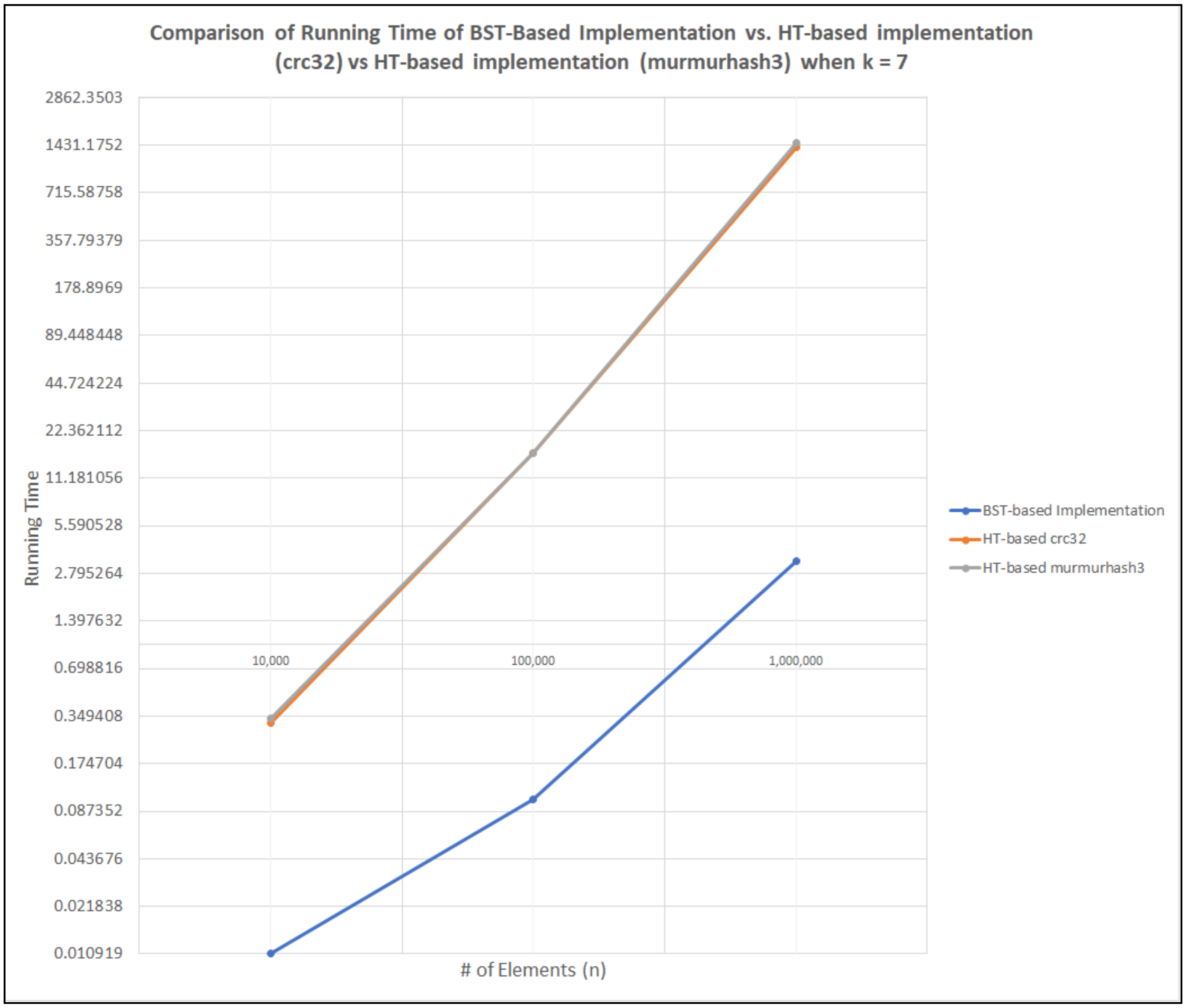


Figure 37. Running Time of the two HT-based implementation and the BST-based implementation when k = 7

VIII. References

Chung, C. (n.d.). *What is Hashing? Benefits, types and more*. Retrieved December 09, 2022, from https://www.2brightsparks.com/resources/articles/introduction-to-hashing-and-its-uses.html

harish-r. (n.d.). Binary Search Tree Implementation in C. *Gist*. Retrieved December 09, 2022, from https://gist.github.com/harish-r/da5fc1d0a2e63e0d60b0

Harvey, A. (2016, February 6). *Five* non-cryptographic hash functions enter. One hash function leaves.* Retrieved December 09, 2022, from https://www.youtube.com/watch?v=siV5pr44FAI.

Ram, V. (2022, August 4). *Hash Table in C/C++ - A Complete Implementation*. Retrieved December 09, 2022, from https://www.digitalocean.com/community/tutorials/hash-table-in-c-plus-plus

The Apache Software Foundation. (n.d.). *MurmurHash3 (Apache Commons Codec 1.15 API)*. Retrieved December 09, 2022, from https://commons.apache.org/proper/commons-codec/apidocs/org/apache/commons/codec/digest/MurmurHash3.html

IX. Work Distribution

The work distribution for MCO2 follows as such:

- Aaron Gabrielle C. Dichoso - Algorithm Design, Hashing Table Program Design
- Josh Austin Mikhail T. Natividad - Algorithm Design, Binary Search Tree Program Design, Running Time Testing and Analysis
- Luis Miguel Antonio B. Razon - Algorithm Design, Hashing Table Program Design
- Adriel Joseph Donato - Algorithm Design, Documentation, Running Time Testing and Analysis