MCO2:

Machine Project Searching

CCDSALG S15 GROUP 1

GUILLERMO, Eugene Siy

SIMBAHON, Joolz Ryane Chavez

VILLANUEVA, Nathaniel Frigillana

Ms. Ma. Christine Gendrano

**Algorithm Design**

* Hash Table

**Description**

Through the observation of hash table implementations online, hash items are popularly represented as a key : value struct pair. This struct idea was adapted and modified to produce kMer distributions of DNA strings:

**struct item**

string kMer

int nCount // kMer counter

struct item\* next // points to next item (for separate chaining)

Originally, the struct member had no struct pointer member as we were planning to implement linear probing as our method of collision resolution. However, when testing our first implementation with large DNA strings (i.e n = 10^k), we found that open addressing filled the hash tables almost instantly. Collisions were high as well. To solve this issue, we implemented external chaining, using linked lists. This notably reduced the likeliness of the tables running out of space as well as relatively reduced collisions significantly.

**INSERT-TO-TABLE** is the primary algorithm that produces the kMer distribution – it utilizes **SEARCH** and **INSERT-ITEM** functions. The algorithm first uses **SEARCH** to check if the key exists in the hash table. If it exists, the counter of the kMer is incremented and return is called. The unique index is then calculated, using **murmurhash3** or **superfasthash.** The item at the index is checked: if it is NULL, the item is inserted at that index, while if it is not NULL, a collision is identified. In the case where there is a collision, the index is passed into **INSERT-ITEM**, as a head of a linked list. From there, the linked list is traversed until the tail is found – inserting the new item, at that location. Below is our final algorithm:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CREATE-TABLE(int nSize)**

item\*\* hashTable = (item\*\*) calloc (sizeof(item\*),nSize); // allocates memory for table of nSize

**for** i = 1 to nSize

hashTable[i] = NULL; // initializes all items to NULL

**GET-MURMUR-HASH-INDEX(char\* key, int nSize)** // murmur3 hash function

**GET-SUPER-FAST-HASH-INDEX(char\* key, int nSize) /**/ superfasthash function

**SEARCH (item\*\* hashTable, char \*key, int nSize, int index)**

**if** tempItem == NULL

return NULL; // key does not exist

**while** tempItem != NULL // traverses linked list

**if** tempItem->kMer = key

return tempItem; // key is found and returned

tempItem = tempItem->next;

**INSERT-ITEM (item\*\* head, char\* key, int nSize)**

item\* newItem = malloc(sizeof(item)) // allocates memory for item

newItem->kMer = malloc(strlen(key) + 1) // allocates memory for kMer

newItem->kMer = key

newItem->nCount = 1 // initializes kMer count

newItem->next = NULL

if \*head == NULL

\*head = newItem

else

item\* currItem = \*head

while currItem->next != NULL // traverses linked list

currItem= currItem->next

currItem->next = newItem // new item is stored in next of the last item in linked list

**INSERT-TO-TABLE (item\*\* hashTable, char\* key, int nSize)**

int index = **HashFunction**(key,nSize) // murmurhash3 or superfasthash

item\* head;

head = **SEARCH**(hashTable,key,nSize,index); // points to location of key in search

**if** head != NULL

head->nCount++ // if key exists, its counter is incremented

**return**

**if** hashTable[index] == NULL

head = NULL

**INSERT-ITEM**(&head,key,nSize) // if index is empty, new item is created and stored in head

**else**

head = hashTable[index] // if index is full, it is stored as head of linked list

**INSERT-ITEM**(&head,key,nSize)

nCollisions++; // collision is recorded

hashTable[index] = head; // item in head is stored in table

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Hash Functions**

The two hash functions we chose were **murmurhash3** and **superfasthash.**

* murmurhash3

The name of the hash function, “murmur”, encapsulates what occurs in the function: cyclic shifting and multiplication operations (Derosiaux, 2017). It is well recognized for its speed and avalanche effect, thus being chosen for our implementation. Another unique feature is that it takes in a seed parameter, which can alter the hashes of a key (Harvey, 2016). In the hash function, if one bit of the key is altered, it triggers half of the bits to change. This is beneficial as it increases the randomization of hashes and reduces collision occurrences.

Implementation used: <https://github.com/PeterScott/murmur3>

* superfasthash

In contrast to the murmur3, we chose another hash function of an opposite nature. Unlike murmur3, the collision rate of superfasthash is higher. Harvey (2016) states that over 10% of input will collide into the same hash values. The likeliness of the increases, especially if inputs are reoccurring. As kMer inputs are repetitive, we theorize a significant difference in speed and collision rates of the two functions.

Implementation used: <https://github.com/kmike/c-hat-trie/blob/master/src/superfasthash.c>

* Binary Search Tree

**Description**

A Binary Search Tree (BST) is a data structure that involves the use of nodes (maximum of 2 nodes). It is usually implemented by utilizing a struct that has a key and value, together with the pointers to the left and right nodes. In our implementation, the string kMer served as the node key, while the nCount would be the value.

**struct node**

char \*kMer // stores the pointer to the string

int nCount // kMer counter

struct node \*pLeft // pointer of left node

struct node \*pRight // pointer of right node

Using this struct, we were able to implement an algorithm that works as a BST. In computing the k-mer distribution of a DNA sequence, we utilized the BST functions **CREATE**, **INSERT**, and **DELETE**. The **CREATE** function creates an empty binary tree by initializing the passed node to NULL. The **INSERT** function is then used to create and add new nodes to the BST. If the current node is unoccupied, it will proceed to the allocation of memory for the new node and string key. The string key of the node will then copy the key that was passed in the function. However, if the current node is occupied, it will check if the key passed in the function belongs to the left or right subtree by using the strcmp() function. Upon checking, the insert function is recursively called to traverse from either the left or right subtree until it reaches an empty node. It is also possible that the new node has a key that already exists in the BST. In this case, it will simply add the nCount of that same node in the BST. Lastly, the **DELETE** function is used to destroy a BST by continuously checking if the node is occupied and recursively calling the DELETE function. After the deletion, the memory allocated from the nodes and keys should be released. We did not use the SEARCH function anymore since we realized that it is simpler and faster if we just made use of the nCount variable. Below is our final algorithm:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**CREATE (node\*\* node)**

(\*node) = NULL // initializes the pointer to NULL

**SEARCH (char\* key, node\*\* node)**

**if** (\*node) == NULL or strcmp((\*node)->kMer, key) == 0 //checks if the key is found at the current node

**return (\*node)** // returns the location of the node

**else if** strcmp((\*node)->kMer, key) > 0) // checks if the key is in the left subtree

**return search(key, &(\*node)->pLeft)** // perform search on the left subtree

**else //** checks if the key is in the right subtree

**return search(key, &(\*node)->pRight)** // perform search on the left subtree

**INSERT (char\* key, node\*\* node)**

int nRes // stores the result value for strcmp

**if**  (\*node) == NULL // if the current node is not occupied

(\*node) = (struct node\*)malloc(sizeof(struct node) + (strlen(key) + 1)) // allocates memory for the node

(\*node)->kMer = malloc(strlen(key) + 1) // allocates memory for the string

strcpy((\*node)->kMer, key) // copies the key to the node

(\*node)->pLeft = NULL

(\*node)->pRight = NULL

(\*node)->nCount = 1 // initializes nCount

**else** // if the current node is occupied

nRes = strcmp(key, (\*node)->kMer)

**if** nRes < 0 // if the key is less than the string on the current node

insert(key, &(\*node)->pLeft) // perform insert to the left subtree

**else if** nRes > 0 // if the key is greater than the string on the current node

insert(key, &(\*node)->pRight) // perform insert to the right subtree

**else //** if key is the same as the string on the current node

(\*node)->nCount++ // increments 1 to nCount

**DELETETREE (node\*\* node)**

**if** (\*node) != NULL // if the current node is occupied

**if** (\*node)->pLeft != NULL // if the left node is occupied

deleteTree(&(\*node)->pLeft) // perform deleteTree on the left subtree

**if** (\*node)->pRight != NULL // if the right node is occupied

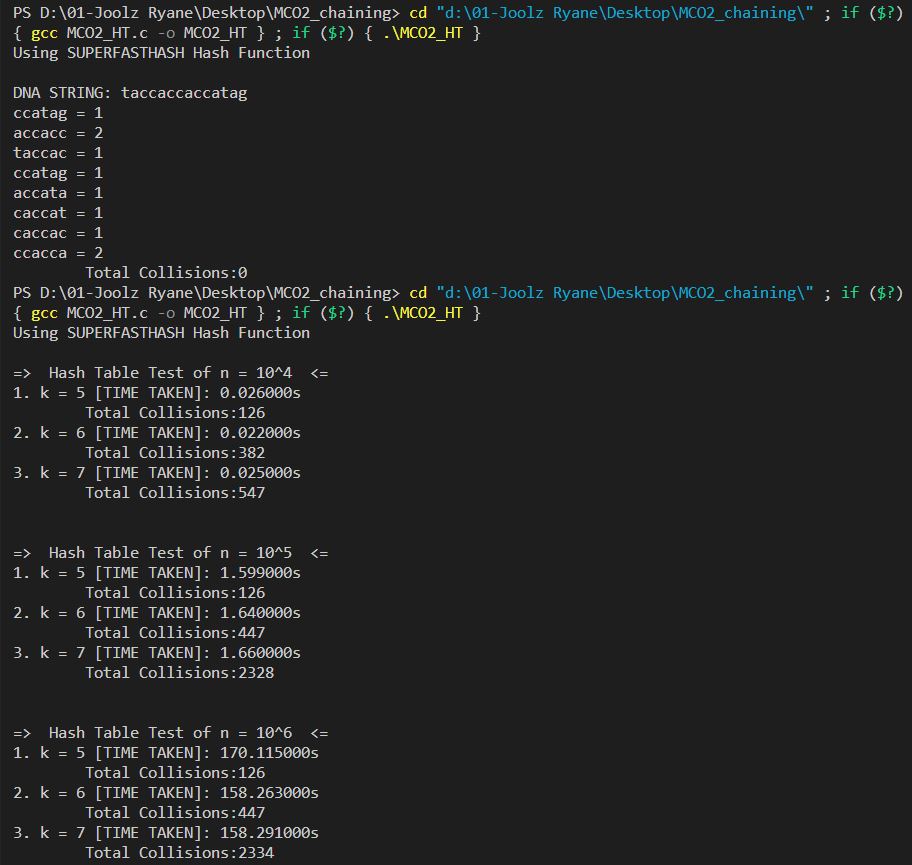
deleteTree(&(\*node)->pRight) // perform deleteTree on the right subtree

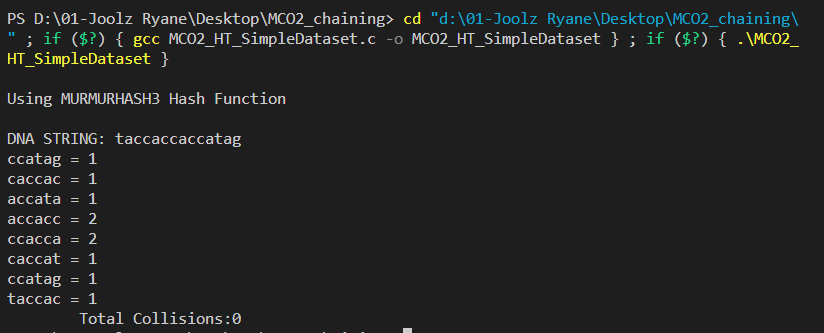
free((\*node)->kMer) // releases the memory allocation for the string

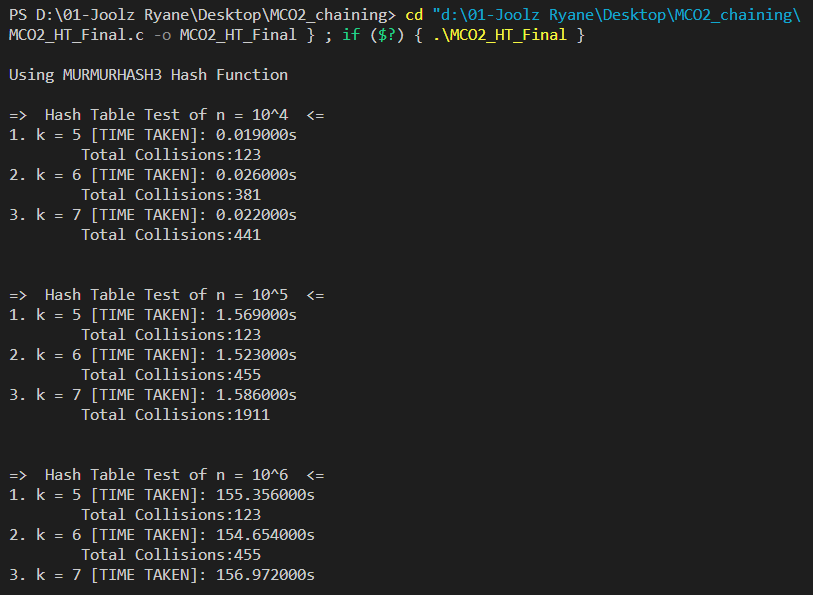
free((\*node)) // releases the memory allocation for the node

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

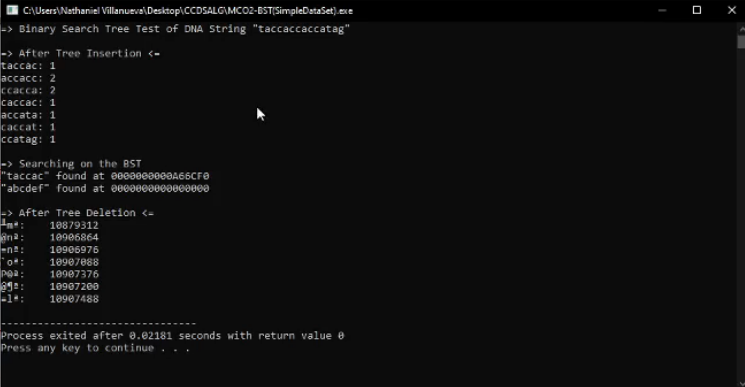
**Program Screenshots**

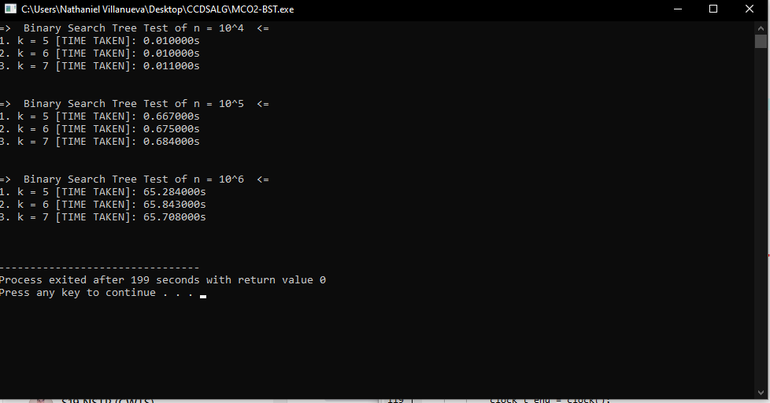
K-mer distribution and the runtime report of DNA string lengths n = 104, 105, 106 with k = 5, 6 ,7 using the Superfasthash





K-mer distribution and the runtime report of DNA string lengths n = 104, 105, 106 with k = 5, 6 ,7 using the Murmurhash3

K-mer distribution of a simple data set using a Binary Search Tree

Runtime reports of Binary Search Tree for DNA string lengths of n = 104, 105, 106 with k = 5, 6 ,7

**Empirical Analysis of Running Time**

* Hash Table (with **superfasthash**)
  + k = 5

| DNA string length (n) | Time Taken (s) | Collisions |
| --- | --- | --- |
| 10^4 | 0.026 | 126 |
| 10^5 | 1.599 | 126 |
| 10^6 | 170.115 | 126 |

* + k = 6

| DNA string length (n) | Time Taken (s) | Collisions |
| --- | --- | --- |
| 10^4 | 0.022 | 382 |
| 10^5 | 1.64 | 447 |
| 10^6 | 158.263 | 447 |

* + k = 7

| DNA string length (n) | Time Taken (s) | Collisions |
| --- | --- | --- |
| 10^4 | 0.025 | 547 |
| 10^5 | 1.66 | 2328 |
| 10^6 | 158.291 | 2334 |

* Hash Table (with **murmurhash3**)
  + k = 5

| DNA string length (n) | Time Taken (s) | Collisions |
| --- | --- | --- |
| 10^4 | 0.019 | 123 |
| 10^5 | 1.569 | 381 |
| 10^6 | 155.356 | 441 |

* + k = 6

| DNA string length (n) | Time Taken (s) | Collisions |
| --- | --- | --- |
| 10^4 | 0.026 | 123 |
| 10^5 | 1.523 | 455 |
| 10^6 | 154.654 | 1911 |

* + k = 7

| DNA string length (n) | Time Taken (s) | Collisions |
| --- | --- | --- |
| 10^4 | 0.022 | 123 |
| 10^5 | 1.586 | 455 |
| 10^6 | 156.972 | 1931 |

* Binary Search Tree
  + k = 5

| DNA string length (n) | Time Taken (s) |
| --- | --- |
| 10^4 | 0.01 |
| 10^5 | 0.667 |
| 10^6 | 65.284 |

* + k = 6

| DNA string length (n) | Time Taken (s) |
| --- | --- |
| 10^4 | 0.01 |
| 10^5 | 0.675 |
| 10^6 | 65.843 |

* + k = 7

| DNA string length (n) | Time Taken (s) |
| --- | --- |
| 10^4 | 0.011 |
| 10^5 | 0.684 |
| 10^6 | 65.708 |

**Comparison**

* + Collision Frequency Between the Two Hashing Functions

| n and k | superfasthash | murmurhash3 |
| --- | --- | --- |
| n = 10^4, k = 5 | 126 | 123 |
| n = 10^4, k = 6 | 382 | 381 |
| n = 10^4, k = 7 | 547 | 441 |
| n = 10^5, k = 5 | 126 | 123 |
| n = 10^5, k = 6 | 447 | 455 |
| n = 10^5, k = 7 | 2328 | 1911 |
| n = 10^6, k = 5 | 126 | 123 |
| n = 10^6, k = 6 | 447 | 455 |
| n = 10^6, k = 7 | 2334 | 1931 |

In reference to sources online, we hypothesized that there would be a huge difference between the collisions of the two functions. In a study by Harvey (2016), where the two hash functions were tested with string inputs, superfasthash had a higher number of collisions compared to murmurhash3. However, looking at our results, murmurhash3 seems slightly faster, only by a few seconds. Overall the collision rates are almost similar, where n=10^4 and n=10^5. For example, when n=10^4 and k=5, superfasthash had 126 collisions while the other had 123. Though it can be noted that superfasthash had twice as many collisions when n=10^6 and k=7.

* + Time Taken By All the Algorithms

| n and k | superfasthash | murmurhash3 | Binary Search Tree |
| --- | --- | --- | --- |
| n = 10^4, k = 5 | 0.026 | 0.019 | 0.01 |
| n = 10^4, k = 6 | 0.022 | 0.026 | 0.01 |
| n = 10^4, k = 7 | 0.025 | 0.022 | 0.011 |
| n = 10^5, k = 5 | 1.599 | 1.569 | 0.667 |
| n = 10^5, k = 6 | 1.64 | 1.523 | 0.675 |
| n = 10^5, k = 7 | 1.66 | 1.586 | 0.684 |
| n = 10^6, k = 5 | 170.115 | 155.356 | 65.284 |
| n = 10^6, k = 6 | 158.263 | 154.654 | 65.843 |
| n = 10^6, k = 7 | 158.291 | 156.972 | 65.708 |

Prior to these results, we hypothesized that superfasthash would be the fastest compared to the others due to its name being “super fast.” However, it can be seen that the Binary Search Tree is the fastest for all lengths of n and k-mers. The difference became prominent when it reached bigger n-length k-mers. At n-length 10^6, both hash functions had gone over 150 seconds which is much slower than the Binary Search Tree. Ultimately, the results showed that superfasthash was the slowest. Therefore, proving our assumptions wrong.

**Description of the Work Distribution**

The work was divided into two parts for the HT-based implementation and BST-based implementation. Eugene and Nathaniel worked together to implement the binary search tree, and Joolz implemented the hash tables. We then tested the created codes and checked if they were working correctly. Lastly, the writing and proofreading of the documentation were done collaboratively.

**References**

*Binary Search Tree*. (n.d.). https://www.programiz.com/dsa/binary-search-tree

*Binary tree for strings c*. (2014, November 11). Stack Overflow. https://stackoverflow.com/questions/26874194/binary-tree-for-strings-c

Derosiaux, S. (2017, August 26). *The murmur3 hash function: hashtables, bloom filters, hyperloglog*. sderosiaux.com. https://www.sderosiaux.com/articles/2017/08/26/the-murmur3-hash-function--hashtables-bloom-filters-hyperloglog/#:~:text=%E2%80%9Cmurmur%E2%80%9D%20stands%20for%20multiply%2C,distribution%2C%20good%20avalanche%20effect).

GeeksforGeeks. (2022, June 29). *How to handle duplicates in Binary Search Tree?* https://www.geeksforgeeks.org/how-to-handle-duplicates-in-binary-search-tree/

Jacob Sorber. (2020, January 20). *Understanding and implementing a Hash Table (in C)* [Video]. YouTube.<https://www.youtube.com/watch?v=2Ti5yvumFTU>

Korobov, M. (2011, November 21). *c-hat-trie/superfasthash.c at master · kmike/c-hat-trie*. GitHub. https://github.com/kmike/c-hat-trie/blob/master/src/superfasthash.c

Linux.conf.au 2016 -- Geelong, Australia. (2016, February 6). *Five\* non-cryptographic hash functions enter. One hash function leaves.* YouTube. https://www.youtube.com/watch?v=siV5pr44FAI

PeterScott, exabytes18, & macklin. (2015, May 2). *GitHub - PeterScott/murmur3: Murmur3 hash in C*. GitHub. https://github.com/PeterScott/murmur3

Ram, V. (2022, August 3). *Hash Table in C/C++ - A Complete Implementation*. DigitalOcean Community. https://www.digitalocean.com/community/tutorials/hash-table-in-c-plus-plus