Group 7 – LA95:

1. Leony Suhendryck Mao – 2702242253
2. Arzeta Putri Arsandhi – 2702322544
3. Liauw, Jeremy Marvelle – 2702337376
4. Farrel Hakim Amran – 2702315671

1. **Background consists of team's planning and the reason of choosing the algorithm related with the suitability between case and algorithm.**

Background

In computer science, hash tables are widely used data structures that provide efficient data retrieval based on key-value pairs. Hash tables use a hash function to compute an index into an array of buckets or slots, from which the desired value can be found. This method allows for average-case constant time complexity, O(1), for both insertion and search operations, making hash tables an excellent choice for applications requiring fast data access.

Team's Planning and Algorithm Choice

Our team has chosen to implement a hash table for this project due to its suitability for scenarios requiring rapid insertion, deletion, and lookup/searching operations. The specific case we are addressing involves managing a collection of strings, which necessitates efficient and quick access to ensure smooth performance.

Suitability Between Case and Algorithm

1) **Efficiency:** Hash tables provide O(1) average time complexity for insert, delete, and search operations. This efficiency is crucial for applications that handle large datasets, where performance is a primary concern.
2) **Scalability:** The hash table can dynamically adjust to varying sizes of datasets. Our implementation allows for easy modification of the table size, providing flexibility as the dataset grows or shrinks.
3) **Simplicity:** The implementation of a hash table is straightforward and does not require complex data structures. This simplicity reduces the likelihood of errors and makes the codebase easier to maintain and extend.

4) **Collision Handling:** Our implementation uses separate chaining with linked lists to handle collisions. This method is effective and simple, ensuring that even when multiple keys hash to the same index, all values can still be stored and retrieved efficiently.

Detailed Explanation of the Code

The provided C++ code defines a hash table class with the following functionalities:

1) **Hash Function:** The Hash function computes the hash value for a given string using the DJB2 algorithm, known for its simplicity and good distribution.
2) **Constructor:** The constructor initializes the hash table with a specified size and allocates memory for the table array.
3) **Insert:** The Insert function adds a new string to the hash table. It computes the hash value to determine the correct index and handles collisions by inserting new nodes at the head of the linked list at that index.
4) **Delete:** The Delete function removes a specific string from the hash table. It traverses the linked list at the computed index to find and delete the target node.
5) **Search:** The Search function checks for the existence of a specific string in the hash table. It traverses the linked list at the computed index to find the target node.
6) **View:** The View function prints the contents of the entire hash table, showing the data stored at each index.
7) **Destructor:** The destructor deallocates memory used by the hash table, ensuring no memory leaks.

2. Literature Review consists of theory of related/chosen data structure and general theory (theory that align with the case of the program).

A. Hash Table

Hash Table is a data structure which organizes data using hash functions in order to support quick insertion and search. There are two different kinds of hash tables: hash set and hash map.

A. The hash set is one of the implementations of a set data structure to store no repeated values.
B. The hash map is one of the implementations of a map data structure to store (key, value) pairs. Here, the key is index, while the value is the key of each data.

But here, whenever we mention hash table, we refer to hash map.

In hash table, the indices are extracted from the key of the data using a hash function. For example, we will use a hash function that extracts the first character of a key string value and map it into some numbers. Therefore, we map the key to array indices. This process of mapping the key to appropriate indices in a hash table is called hashing.

Another example, supposed we want to store 5 key string: define, float, exp, char, and atan into a hash table with size 26. The hash function we will use is "transform the first character of each string into a number between 0..25" (a will be 0, b will be 1, c will be 2, …, z will be 25).

So, the **hash table** of those string:

h[ ]	key
0	atan
1	
2	char
3	define
4	exp
5	float
6	
...	
25	

atan is stored in h[0] because a is 0, char is stored in h[2] because c is 2, define is stored in h[3] because d is 3, exp is stored in h[3] because e is 3, float is stored in h[3] because f is 3, and so on.. We only consider the first character of each string.

B. Hash Function

There are many ways to map a key into an index. The following are some of the important methods for constructing hash functions.

1. **Mid-square**

Square the key and then use an appropriate number of bits from the middle of the square to obtain the index (hash-key). If the key is a string, it is converted to a number. Steps:

1. Square the key. (k^2)
2. Extract the middle r bits of the result obtained in step 1.

> Function : h(k) = s
> k = key
> s = the hash key obtained by selecting r bits from k^2

For example,

KEY	SQUARED KEY	MIDDLE PART
3121	9740641	406
3122	9746884	408

2. Division (most common)

Divide the key by using modulus operator. The simplest method of hashing an integer.

> Function: h(z) = z mod M
> z = key
> M = the value using to divide the key, usually using a prime number, the table size or the size of memory used.

For example, suppose the table is to store strings. A very simple hash function would be to add up ASCII values of all the characters in the string and take modulo of table size, say 97.

"COBB" would be stored at the location:

(67 + 79 + 66 + 66) % 97 = **84**

"HIKE" would be stored at the location:

(72 + 73 + 75 + 69) % 97 = **95**

"ABCD" would be stored at the location:

(65 + 66 + 67 + 68) % 97 = **72**

3. Folding

Partition the key into several parts, then add the parts together to obtain the hash key. Steps:

a. Divide the key into a number of parts.

b. Add the individual part (usually the last carry is ignored).

For example, given a hash table of 100 locations, calculate the hash value of 5678, 321 and 34567, since there are 100 locations to address, so we will break the key into parts where each part will contain two digits.

Key	5678	321	34567
Parts	56 and 78	32 and 1	34, 56 and 7
Sum	134	33	97
Hash-key (index)	34 (ignore the last carry)	33	97

4. **Digit Extraction**

   A predefined digit of the given number is considered as the hash address.

   Example:

   A. Consider x = 14,568

   B. If we extract the 1st, 3rd, and 5th digit, we will get a hash code of: 158.

5. **Rotating Hash**

   Use any hash method (such as division or mid-square method). After getting the hash code/address from the hash method, do rotation. Rotation is performed by shifting the digits to get a new hash address. Example:

   a. Given hash address = 20021

   b. Rotation result: 12002 (fold the digits)

C. Collision

What will happen if we want to store these strings using the previous hash function (use the first character of each string)? For example: define, **float**, exp, **char**, **atan**, **acos ceil**, **floor**. There are several strings which have the same hash-key, it's float and floor (hash-key: 5), atan and acos (hash-key: 0), char and ceil (hash-key: 2). It's called a collision. How can we handle this? There are two general ways to handle collisions:

1. **Open Addressing**

Computes new positions using probe sequences and the next record is stored in that position. Techniques:

A. Linear probing

This is the hash table of these string (**define, float, exp, char, atan, ceil, floor, acos**):

h[ ]	Key
0	atan
1	acos
2	char
3	define
4	exp
5	float
6	ceil
7	floor
...	

Note that *ceil* is stored in h[6], *acos* is stored in h[1] and *floor* is stored in h[7]. When we want to store "ceil", there is already "char" stored in h[2], so we search the next empty slot which is h[6]. If a collision occurs, the next slot (index + 1) is checked, and this process continues until an empty slot is found.

B. Quadratic probing

Similar to linear probing, but it uses a quadratic function to determine the next slot. For example, the sequence might be index + 1^2, index + 2^2, and so on.

C. Rehashing

When the hash table becomes nearly full, the number of collisions increases, there by degrading the performance of insertion and search operations. In such cases, a better option is to create a new hash table with size double of the original hash table. All the entries in the original hash table will then have to be moved to the new hash table. This is done by taking each entry, computing its new hash value, and then inserting it in the new hash table.

Consider the hash table of size 5 with hash function h(x) = x % 5.

0	1	2	3	4
	26	31	43	17

Now, we rehash the key from the old hash table into new hash table of size 10 using hash function h(x) = x % 10.

0	1	2	3	4	5	6	7	8	9
	31		43			26	17		

2. Chaining

In chaining, we store each string in a chain (linked list). So, if there is collision, we only need to iterate on that chain.

h[ ]	Value
0	atan → acos
1	NULL
2	char → ceil
3	define
4	exp
5	float → floor
6	NULL
7	NULL
...	

D. Theory align case program

1. Data Management with Hash Tables

In many applications, managing data efficiently is crucial. Hash tables are particularly useful for cases requiring fast insertion, deletion, and searching operations. The time complexity for these operations is typically O(1) on average, making hash tables an optimal choice for applications with high-frequency access patterns.

2. **Application to Storing Strings**

In the context of managing a collection of strings, a hash table provides an efficient way to store, search, and manage these strings. Each string (key) maps to a specific slot in the hash table, allowing for quick retrieval. Given the nature of operations involving strings, such

as frequent lookups or updates, the O(1) average time complexity of hash tables makes them well-suited for this task.

3. **Search Efficiency**

   Searching for a string in a collection involves checking whether the string exists in the data structure. Hash tables, due to their direct access nature, allow for constant time complexity searches, significantly outperforming other data structures like lists or trees for this purpose.

4. **Viewing and Deletion Operations**

   - Viewing Specific Data: The hash function quickly maps the string to its index, and the corresponding value can be retrieved with minimal computation.
   - Viewing All Data: Iterating over the entire hash table allows for displaying all stored strings.
   - Deletion: Hash tables allow for efficient deletion by locating the string through the hash function and removing it from the table.

5. **Hash Function Used: DJB2**

   In our program, we use the DJB2 hash function, which was created by Daniel J. Bernstein. DJB2 is a popular and effective hash function for strings. It works by iterating over each character in the string and updating the hash value using a combination of bit shifting and addition.

   Explanation of DJB2

   The DJB2 algorithm initializes a hash value to 5381. For each character in the input string, it shifts the current hash value left by 5 bits (equivalent to multiplying by 32), adds the current hash value, and then adds the ASCII value of the character. This process ensures that the hash value is influenced by each character in the string.

   However, it effectively multiplies the hash value by 33 because it shifts the hash left by 5 bits (multiplying by 32) and then adds the current hash value.

   Example with String "cat":

   Initialization:

   Start with an initial hash value of 5381.

Process Each Character:

Character 'c':

ASCII value of 'c' is 99.

Update hash: hash = (5381 << 5) + 5381 + 99

This is equivalent to: hash = 5381 * 33 + 99

Calculate: hash = 177672

Character 'a':

ASCII value of 'a' is 97.

Update hash: hash = (177672 << 5) + 177672 + 97

This is equivalent to: hash = 177672 * 33 + 97

Calculate: hash = 5863273

Character 't':

ASCII value of 't' is 116.

Update hash: hash = (5863273 << 5) + 5863273 + 116

This is equivalent to: hash = 5863273 * 33 + 116

Calculate: hash = 193488125

Final Hash Value:

The final hash value for the string "cat" is 193488125.

6. **Collision Resolution Technique: Chaining**

   To handle collisions, we use chaining. In chaining, each bucket in the hash table points to a linked list of entries that hash to the same index. When a collision occurs, the new entry is simply added to the list at that index. This method is effective and relatively easy to implement.

Conclusion

Hashing algorithms, particularly the DJB2 method combined with chaining for collision resolution, provide an efficient solution for managing a collection of strings. They offer quick search, insertion, and deletion capabilities, making them suitable for applications requiring fast and frequent data

access. By handling collisions effectively, the hash table maintains its performance even as the dataset grows.

3. **Benefits consists of benefits of choosing the algorithm.**
    - Hash provides better synchronization than other data structures.
    - Hash tables are more efficient than search trees or other data structures.
    - Hash provides constant time for searching, insertion and deletion operations on average.
    - Hash tables are space-efficient.
    - Most Hash table implementation can automatically resize itself.
    - Hash tables are easy to use.
    - Hash tables offer a high-speed data retrieval and manipulation.
    - Fast lookup: Hashes provide fast lookup times for elements, often in constant time O(1), because they use a hash function to map keys to array indices. This makes them ideal for applications that require quick access to data.
    - Efficient insertion and deletion: Hashes are efficient at inserting and deleting elements because they only need to update one array index for each operation. In contrast, linked lists or arrays require shifting elements around when inserting or deleting elements.
    - Space efficiency: Hashes use space efficiently because they only store the key-value pairs and the array to hold them. This can be more efficient than other data structures such as trees, which require additional memory to store pointers.
    - Flexibility: Hashes can be used to store any type of data, including strings, numbers, and objects. They can also be used for a wide variety of applications, from simple lookups to complex data structures such as databases and caches.
    - Collision resolution: Hashes have built-in collision resolution mechanisms to handle cases where two or more keys map to the same array index. This ensures that all elements are stored and retrieved correctly.

4. **Result consists of all feature screenshot, the program code (is presented in written form, not screenshot form) and give the coding documentation of the program (the code/function with human language).**

All feature screenshot:

```
void Insert(string New) {
  unsigned long Key = Hash(New);
  Node *NewNode = new Node(New);
  if (this->Table[Key])
    NewNode->Next = this->Table[Key];
  this->Table[Key] = NewNode;
}
void Delete(string Target) {
  unsigned long Key = Hash(Target);
  if (!this->Table[Key]) return;
  Node *Prev = NULL;
  Node *Current = this->Table[Key];
  while (Current && Current->Data != Target) Prev = Current,
    Current = Current->Next;
  if (Current) {
    if (Prev) {
      Prev->Next = Current->Next;
    } else {
      this->Table[Key] = Current->Next;
    }
    delete Current;
  }
}
void Search(string Target) {
  unsigned long Key = Hash(Target);
  Node *Current = this->Table[Key];
  while (Current && Current->Data != Target) Current = Current->Next;
  if (Current) {
    cout << Target << " exists at index: " << Key << '\n';
  } else {
    cout << Target << " wasn't found!" << '\n';
  }
}
void View() {
  for (size_t i = 0;i < this->Size;i++) {
    cout << i << ": ";
    Node *Current = this->Table[i];
    while (Current && Current->Next) {
      cout << Current->Data << " | ";
      Current = Current->Next;
    }
    if (Current) cout << Current->Data;
    cout << '\n';
  }
}
```

The program code (is presented in written form, not screenshot form) :

```
#include <bits/stdc++.h>
using namespace std;

class HashTable {
      private:
      struct Node {
        string Data;
        Node *Next;
        Node(string Data): Data(Data), Next(NULL) {}
```

```
};
Node **Table;
size_t Size;
unsigned long Hash(string str) {
  unsigned long hash = 5381;
  for (char c: str)
    hash = ((hash << 5) + hash) + c;
  return hash%Size;
}
public:
HashTable(size_t Size) {
  this->Size = Size;
  this->Table = new Node*[Size];
  memset(Table, NULL, sizeof(Table));
}
void Insert(string New) {
  unsigned long Key = Hash(New);
  Node *NewNode = new Node(New);
  if (this->Table[Key])
    NewNode->Next = this->Table[Key];
  this->Table[Key] = NewNode;
}
void Delete(string Target) {
  unsigned long Key = Hash(Target);
  if (!this->Table[Key]) return;
  Node *Prev = NULL;
  Node *Current = this->Table[Key];
  while (Current && Current->Data != Target) Prev = Current,
    Current = Current->Next;
  if (Current) {
    if (Prev) {
      Prev->Next = Current->Next;
    } else {
      this->Table[Key] = Current->Next;
    }
    delete Current;
  }
}
void Search(string Target) {
  unsigned long Key = Hash(Target);
  Node *Current = this->Table[Key];
  while (Current && Current->Data != Target) Current = Current->Next;
  if (Current) {
    cout << Target << " exists at index: " << Key << '\n';
  } else {
    cout << Target << " wasn't found!" << '\n';
  }
}
void View() {
  for (size_t i = 0;i < this->Size;i++) {
    cout << i << ": ";
    Node *Current = this->Table[i];
    while (Current && Current->Next) {
      cout << Current->Data << " | ";
      Current = Current->Next;
    }
    if (Current) cout << Current->Data;
    cout << '\n';
  }
}
~HashTable() {
  for (size_t i = 0;i < this->Size;i++) {
    Node *Current = this->Table[i];
```

```
          while (Current) {
            Node *Temp = Current;
            Current = Current->Next;
            delete Temp;
          }
        }
        delete[] Table;
      }
};
```

DOCUMENTATION OF "HashTable"

Declared with the syntax "HashTable [Identifier]([Size]);"

Change [Identifier] to the name you wish the hash table to have.

Change [Size] to the size you wish the hash table to have.

void Insert(string New): Insert a new string into the hash table.

void Delete(string Target): Delete one instance of the target from the hash table.

void Search(string Target): Search for the target value and inform about its existence in the hash table.

void View(): View every index of the hash table.

DOWNLOAD FULL FILE PROGRAM:

https://drive.google.com/file/d/1NEKtfPBo1qd3LpRUtOPsRfMbPc6S-r2B/view?usp=sharing

Notes: The program was created with version C++17.

REFERENCE:

- S. Sridhar. 2015. Design and Analysis of Algorithms. Oxford University Press. New Delhi. ISBN: 9780198093695. Chapter 5
- Reema Thareja. 2014. Data structures using C. Oxford University Press. New Delhi. ISBN:9780198099307. Chapter 15

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, & Clifford Stein. (2022). Introduction to Algorithms. 04. The MIT Press. London. ISBN: 9780262046305. Chapter 11.
- Hash Table, https://visualgo.net/en/hashtable?slide=1
- Hash Table, https://leetcode.com/explore/learn/card/hash-table/
- benefits of chosing hash algorithm : https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-hash-data-structure/amp/
- https://theartincode.stanis.me/008-djb2/
- GeeksforGeeks: Hashing Data Structure, https://www.geeksforgeeks.org/hashing-data-structure/