**Hash Tables (C++)**

To understand the concept of hash tables, let’s first define the idea of a **hash function**:

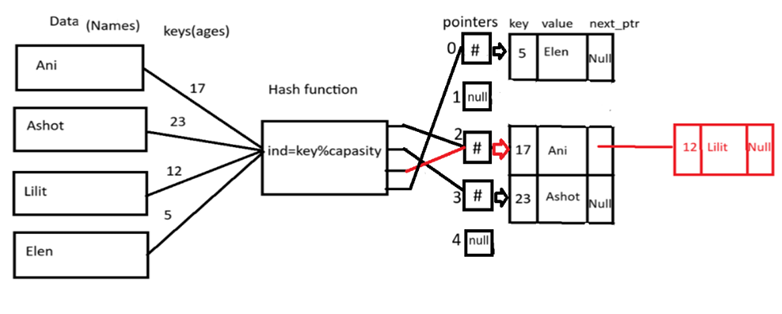
**Hash Functions**  
A hash function is a function that transforms a piece of data into a number, typically an integer, which can serve as an index in an array, making search operations in large tables faster and more efficient. The values returned by the hash function are called hash values, hash codes, or simply hashes. Hash functions are widely used to speed up search or comparison operations in tables, such as finding parts in a "database," searching for similar sequences in DNA structures, spell checking, and dictionary operations. These functions help achieve an average time complexity of O(1) for dictionary operations.  
A hash function must meet the following requirements:  
• It must return the same index for the same input.  
• Different inputs must correspond to different indices.  
• The hash function must return existing indices.

**Hash Tables**  
A hash table in C/C++ is a data structure that combines a hash function with an array. It represents an array of key-value pairs. A hash table allows for the establishment of a connection between a key and its corresponding value (e.g., the product name as the key and its price as the value). It uses a hash function, which, when provided with a key, returns the index in the array where the value is stored. As a result, searching for an element in the table can be performed in constant time, O(1), on average.

**Collision Handling**  
When a hash function returns the same index for two different keys, collisions occur. A good choice of hash function minimizes the likelihood of collisions, but it is theoretically impossible to eliminate collisions entirely. There are several algorithms for handling collisions. In the code presented below, it is suggested that all {key:value} pairs for which the hash function returns the same index should be stored at that index as follows:

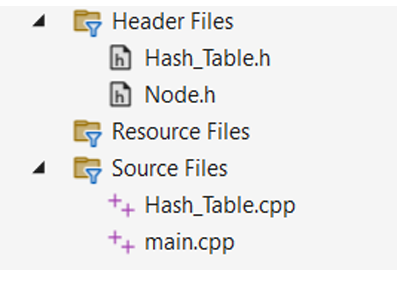
• We will have a **Node** object, which will represent a container of the {key:value} pair.  
• The hash table will represent an array of pointers to Node objects.  
• When a new element is added to an empty index of the table, the pointer at that index becomes a pointer to a node (subsequently forming a linked list of nodes).  
• The next time a new element is added to the same index, it is appended to the end of the linked list pointed to by the pointer at that index.

A characteristic of hash tables is the **load factor**, which shows the ratio of the table’s size to the number of actual elements in it. When the load factor exceeds 0.7, we increase the size of the array and rehash every element to place it in its new position in the larger table. This helps to reduce the number of collisions in the table.



**Implementation:**

* First, let's prepare the working environment.

****

* Let's start with the implementation of the Node (node) structure.

#pragma once

template<typename V, typename K>

struct Node

{

Node() {}//defoult constructor

Node(const V& v, const K& k)

: key(k)

, value(v)

, next(nullptr)

{

}

V value;

K key;

Node<V, K>\* next;

};

* Hash Table class interface

#pragma once

#include "Node.h"

#include <assert.h>

#include <iostream>

template<typename V, typename K>

class HashTable

{

private:

int TableSize; // Hash table size

Node<V, K>\*\* m\_table; // Array of pointers to Node type

int count\_of\_elements; // Number of elements in the table

float load\_factor; // Load factor metric

public:

HashTable(int size = 100); // Parameterized constructor for table size

~HashTable();

HashTable(const HashTable& copy); // Copy constructor

int HashFunction(K key); // Hash function for index generation

void Add(V value, K key); // To add an element

void Remove(K key); // To remove an element

void clear(); // To clear the table

V operator[] (K key); // Allows access to value by key

HashTable& operator= (const HashTable& copy); // Assignment operator

void print\_table() const; // Function to print the table

bool empty() const; // Check if the table is empty

int NumberOfItemsInKey(const K& key); // Number of collisions for a given key

private:

void resize(); // Function to resize the table (newSize = oldSize \* 2 + 1)

};

* Class implementation

template<typename V, typename K>

HashTable<V, K>::HashTable(int size)

: TableSize(size)

, m\_table(new Node<V, K>\* [TableSize]) // Memory allocation for the table

, count\_of\_elements(0) // Initial count of elements is 0

, load\_factor(0) // Load factor is 0 when the table is created

{

for (int i = 0; i < TableSize; ++i)

{

m\_table[i] = nullptr;

}

// Initially, assign nullptr to all pointers in the array

}

template<typename V, typename K>

HashTable<V, K>::~HashTable()

{

clear(); // Clears the table but doesn't free memory for the pointers in the array

delete[] m\_table; // Delete the array of table pointers

m\_table = nullptr; // Set the table pointer to nullptr

}

template<typename V, typename K>

HashTable<V, K>::HashTable(const HashTable& copy)

{

TableSize = copy.TableSize;

m\_table = new Node<V, K>\* [TableSize]; // Allocate memory for the table

for (int i = 0; i < TableSize; ++i)

{

m\_table[i] = nullptr; // Assign nullptr to the table pointers

}

for (int i = 0; i < copy.TableSize; ++i) // Iterate over the old table

{

Node<V, K>\* t = copy.m\_table[i]; // Get each table pointer

while (t) // Traverse the linked list if there is one

{

Add(t->value, t->key); // Add all old data to the new table

t = t->next;

}

}

}

template<typename V, typename K>

HashTable<V, K>& HashTable<V, K>::operator= (const HashTable& copy) // Perform the same steps as the copy constructor, but delete the current table content first

{

clear();

TableSize = copy.TableSize;

m\_table = new Node<V, K>\* [TableSize];

for (int i = 0; i < TableSize; ++i)

{

m\_table[i] = nullptr;

}

for (int i = 0; i < copy.TableSize; ++i)

{

Node<V, K>\* t = copy.m\_table[i];

while (t)

{

Add(t->value, t->key);

t = t->next;

}

}

return \*this; // Return a reference to the current object for multiple assignments

}

template<typename V, typename K>

void HashTable<V, K>::clear()

{

for (int i = 0; i < TableSize; i++) {

Node<V, K>\* tmp = m\_table[i]; // Temporary variable to traverse and delete the linked list

while (tmp != nullptr) {

Node<V, K>\* next = tmp->next;

delete tmp; // Keep the next node and delete the current one

tmp = next; // Move to the next node

}

m\_table[i] = nullptr; // Set the table pointer to nullptr

}

count\_of\_elements = 0;

load\_factor = 0;

}

template<typename V, typename K>

int HashTable<V, K>::HashFunction(K key) // Assumes the key is a text type, hash function can be replaced with a more appropriate one if necessary

{

int size = key.size();

int s = 0;

for (int i = 0; i < size; ++i)

{

s += (i + TableSize) \* int(key[i]);

}

// Traverse through the key and sum up the ASCII values of its characters, multiplied by a chosen pattern (i + TableSize) to ensure better dispersion in the table and avoid collisions

return s % TableSize; // Return the sum modulo the table size to get a valid index

}

template<typename V, typename K>

void HashTable<V, K>::Add(V v, K k)

{

int ind = HashFunction(k); // Get the index in the table where we need to store the new element

if (m\_table[ind] == nullptr) // If nothing is at this index, assign the pointer to a new node and create it with the passed data

{

m\_table[ind] = new Node<V, K>(v, k);

++count\_of\_elements;

}

else

{

// If the index is not empty (i.e., there's a linked list), traverse to the end and add the new element in the same way

Node<V, K>\* tmp = m\_table[ind];

Node<V, K>\* tmp\_prev = nullptr; // Temporary variable to hold the previous node

while (tmp)

{

if (tmp->key == k) // If the key already exists, replace the old value with the new one

{

tmp->value = v;

return;

}

tmp\_prev = tmp;

tmp = tmp->next;

}

// When tmp becomes nullptr, we've reached the end of the list

tmp\_prev->next = new Node<V, K>(v, k); // Link the new node to the previous one

++count\_of\_elements; // Increase the number of elements in the table

}

load\_factor = double(count\_of\_elements) / double(TableSize);

if (load\_factor > 0.7)

{

resize();

}

}

template<typename V, typename K>

void HashTable<V, K>::Remove(K k)

{

int ind = HashFunction(k);

if (m\_table[ind] == nullptr)

{

return; // Empty case

}

Node<V, K>\* tmp = m\_table[ind];

Node<V, K>\* tmp\_prev = nullptr;

if (tmp->key == k) // If the first node has the required element, delete it

{

m\_table[ind] = tmp->next;

delete tmp;

--count\_of\_elements;

load\_factor = double(count\_of\_elements) / double(TableSize); // Update the load factor

return;

}

while (tmp->next) // Traverse until we find the element or reach the end

{

tmp\_prev = tmp;

tmp = tmp->next;

if (tmp->key == k) // If found, delete it

{

Node<V, K>\* t = tmp;

tmp\_prev->next = tmp->next; // Point the previous node to the next of the deleted one

delete t;

--count\_of\_elements;

load\_factor = double(count\_of\_elements) / double(TableSize); // Update the load factor

return;

}

}

std::cout << "Not found" << std::endl; // If the element was not found

}

template<typename V, typename K>

int HashTable<V, K>::NumberOfItemsInKey(const K& key)

{

int c = 1; // Start with 1

int ind = HashFunction(key); // Find the required index

if (m\_table[ind] == nullptr)

{

return 0; // If the index is empty, return 0

}

Node<V, K>\* tmp = m\_table[ind];

while (tmp->next)

{

tmp = tmp->next;

c += 1; // Increment the count for each new node

}

return c;

}

template<typename V, typename K>

V HashTable<V, K>::operator[] (K key)

{

int ind = HashFunction(key);

if (m\_table[ind]) // If the index is not empty

{

Node<V, K>\* tmp = m\_table[ind]; // Start traversing the list to find the key

while (tmp)

{

if (tmp->key == key)

{

return tmp->value; // Return the value for the found key

}

tmp = tmp->next;

}

}

std::cout << "Not found" << std::endl;

assert(0 > 1); // If not found, terminate the program with a message

}

template<typename V, typename K>

void HashTable<V, K>::print\_table() const // Traverse all table pointers and linked lists to print them

{

for (int i = 0; i < TableSize; ++i)

{

if (m\_table[i] == nullptr)

{

continue;

}

Node<V, K>\* tmp = m\_table[i];

std::cout << "[" << tmp->key << ":" << tmp->value << "]" << ' ';

while (tmp->next)

{

tmp = tmp->next;

std::cout << "[" << tmp->key << ":" << tmp->value << "]" << ' ';

}

std::cout << std::endl;

}

std::cout << std::endl;

}

template<typename V, typename K>

void HashTable<V, K>::resize()

{

std::cout << "Resizing" << std::endl;

int tmp\_size = TableSize \* 2 + 1; // Define the new size as 2 times the old size plus 1

HashTable<V, K> tmp(tmp\_size); // Create a new hash table with the new size

for (int i = 0; i < TableSize; ++i) // Iterate over the old table

{

Node<V, K>\* t = m\_table[i];

while (t) // If there is a linked list, traverse it

{

tmp.Add(t->value, t->key); // Add all old data to the new table

t = t->next;

}

}

\*this = tmp;

}

template<typename V, typename K>

bool HashTable<V, K>::empty()const

{

return count\_of\_elements == 0;

}

* Testing the code

#main.cpp

#include "Hash\_Table.h"

#include <iostream>

#include <string>

int main()

{

HashTable<int, std::string> a(5);

a.Add(19,"Anna");

a.Add(18, "Lilit");

a.Add(20, "Hayk");

a.Add(23, "Lilit");

a.Add(72, "Vazgen");

a.Add(7, "Petros");

a.Add(24, "Tatev");

a.Add(56, "Lusine");

a.Add(43, "Nare");

std::cout << "Table:" << std::endl;

a.print\_table();

a.Remove("Karine");

a.Remove("Vazgen");

std::cout << "Table after removing elements:" << std::endl;

a.print\_table();

std::cout <<"Number of items in key Anna are: "<<a.NumberOfItemsInKey("Anna") << std::endl;;

std::cout<<"Nare - " << a["Nare"] << std::endl;

std::cout <<"Lilit - "<< a["Lilit"] << std::endl;

std::cout <<"Lusine - "<< a["Lusine"] << std::endl;

HashTable<int, std::string> b;

b = a;

std::cout << std::endl;

std::cout << "assignment operator used:" << std::endl;

b.print\_table();

b.clear();

std::cout << "cleared table: " << std::endl;

b.print\_table();

std::cout << b.empty() << std::endl;

}

* Output

Resizing

Resizing

Table:

[Tatev:24] [Anna:19]

[Nare:43]

[Petros:7]

[Hayk:20]

[Vazgen:72]

[Lilit:23]

[Lusine:56]

Table after removing elements:

[Tatev:24] [Anna:19]

[Nare:43]

[Petros:7]

[Hayk:20]

[Lilit:23]

[Lusine:56]

Number of items in key Anna are: 2

Nare - 43

Lilit - 23

Lusine - 56

assignment operator used:

[Tatev:24] [Anna:19]

[Nare:43]

[Petros:7]

[Hayk:20]

[Lilit:23]

[Lusine:56]

cleared table:

1