#Hecho por Armin Posadas Flores

Documentación Tabla HASH

```
#ifndef TABLASHASH_HASHTABLE_H
#define TABLASHASH_HASHTABLE_H
```

Se encargan de que el header sea incluido una sola vez para evitar copias de este

```
enum CollisionHandling {
    SEPARATE_CHAINING,
    LINEAR_PROBING,
    QUADRATIC_PROBING,
    DOUBLE_HASHING,
    ROBIN_HOOD,
    COALESCED
};
```

Se encarga de definir diferentes metodos de manejo de colisiones

```
class HashTable {
public:
    explicit HashTable(int size, CollisionHandling method);
    int hashFunction(int value) const;
    int secondHashFunction(int value) const;
    void insertItem(int value);
    void deleteItem(int value);
    int get(int value);
    void displayHash() const;
private:
    int tableSize;
    int itemCount;
    CollisionHandling collisionMethod;
    std::vector<int> table;
    std::vector<bool> occupied;
    std::vector<std::list<int>> chains;
    std::vector<int> next;
    void rehash();
    int nextIndex(int currentIndex, int value, int i) const;
    void insertSeparateChaining(int value);
    void insertLinearProbing(int value);
    void insertQuadraticProbing(int value);
    void insertDoubleHashing(int value);
    void insertRobinHood(int value);
    void insertCoalesced(int value);
};
```

Se encarga de definir varios miembros para la clase, tanto publicos como privados, la mayoria solo poseen una integral de tipo valor, mientras que otros como explicit HasTable utilizan integrales de tamaño y metodo de colision.

```
explicit HashTable(int size, CollisionHandling method)
    : tableSize(size), itemCount(0), collisionMethod(method) {
    table.resize(tableSize, -1);
    occupied.resize(tableSize, false);
    if (method == SEPARATE_CHAINING) {
        chains.resize(tableSize);
    } else if (method == COALESCED) {
        next.resize(tableSize, -1);
    }
}
```

Se inicializa una tabla hash con un valor dado para tamaño, un conteo de items y un metodo de colision, en caso de que el metodo sea por medio de "encadenamiento separado" el tamaño de la tabla sera renombrado a chain.resize, mientras que si se utiliza el metodo "coalesced" es renombrado a "next.resize".

```
int hashFunction(int value) const {
        return value % tableSize;
    }
```

Este se encarga de computarizar el valor primario de la tabla hash

```
int secondHashFunction(int value) const {
        return 7 - (value % 7);
    }
```

Mientras que este se encarga de computarizar un valor secundario, en este caso se encarga de dar un numero primo para que halla mejor distribucion.

```
void insertItem(int value) {
        if (itemCount / static_cast<double>(tableSize) >= 0.7) {
            rehash();
        }
        switch (collisionMethod) {
            case SEPARATE_CHAINING:
                insertSeparateChaining(value);
                break;
            case LINEAR_PROBING:
                insertLinearProbing(value);
                break;
            case QUADRATIC_PROBING:
                insertQuadraticProbing(value);
                break;
            case DOUBLE_HASHING:
                insertDoubleHashing(value);
                break;
            case ROBIN_HOOD:
                insertRobinHood(value);
                break;
            case COALESCED:
                insertCoalesced(value);
                break;
        }
```

```
    }
```

La primera parte se encarga de hacer rehash en caso de que el factor sea mayor a 0.7, la otra parte se encarga de realizar un caso dependiendo del metodo de colisiones.

```
void deleteItem(int value) {
        int index = hashFunction(value);
        switch (collisionMethod) {
            case SEPARATE_CHAINING:
                chains[index].remove(value);
                itemCount--;
                break;
            case LINEAR_PROBING:
            case QUADRATIC_PROBING:
            case DOUBLE_HASHING:
            case ROBIN_HOOD:
            case COALESCED:
                while (occupied[index]) {
                    if (table[index] == value) {
                        table[index] = -1;
                        occupied[index] = false;
                        itemCount--;
                        return;
                    }
                    index = nextIndex(index, value, 0); // Reset i to 0 for quadratic probing
                }
                break;
        }
    }
```

En este caso elimina items dependiendo del metodo que se utilize, para encadenamiento separado se elimina el valor index del encadenamiento y elimina el contador de items, para "coalesced" se busca un index ocupado luego si hay un valor en el index de la tabla se elimina, marca como falso y elimina el contador.

```
int get(int value) {
        int index = hashFunction(value);
        switch (collisionMethod) {
            case SEPARATE_CHAINING:
                for (const int& val : chains[index]) {
                    if (val == value) {
                        return val;
                    }
                }
                break;
            case LINEAR_PROBING:
            case QUADRATIC_PROBING:
            case DOUBLE_HASHING:
            case ROBIN_HOOD:
            case COALESCED:
                int i = 0; // Initialize i for quadratic probing
                while (occupied[index]) {
                    if (table[index] == value) {
                        return table[index];
                    }
                    index = nextIndex(index, value, i++);
```

```
                }
                break;
            }
            return -1; // Return -1 if value is not found
        }
```

Se encarga de obtener un objeto de la tabla hash de igual forma hay varias maneras de obtenerlas dependiendo del metodo. para el metodo de encadenamiento separado este ira en loop por toda la lista hasta encontrar el valor index, y para el metodo coalesced primero se inicializa un contador de sondeo cuadratico y seguida contando en caso de que el index actual este ocupado, luego checa el valor del slot actual y una vez encontrado lo regresa.

```
    void displayHash() const {
            for (int i = 0; i < tableSize; i++) {
                std::cout << i << " --> ";
                if (collisionMethod == SEPARATE_CHAINING) {
                    if (!chains[i].empty()) {
                        for (const auto& val : chains[i]) {
                            std::cout << val << " ";
                        }
                    }
                } else {
                    if (occupied[i]) {
                        std::cout << table[i];
                    }
                }
                std::cout << std::endl;
            }
        }
```

Se encarga de mostrar la tabla hash mostrando el index y su valor correspondiente.

```
    void rehash() {
            std::vector<int> oldTable = table;
            std::vector<bool> oldOccupied = occupied;
            std::vector<std::list<int>> oldChains = chains; // Store the old chains
            tableSize *= 2; // Double the table size
            table = std::vector<int>(tableSize, -1);
            occupied = std::vector<bool>(tableSize, false);
            if (collisionMethod == SEPARATE_CHAINING) {
                chains = std::vector<std::list<int>>(tableSize);
            } else if (collisionMethod == COALESCED) {
                next = std::vector<int>(tableSize, -1);
            }
            itemCount = 0; // Reset itemCount
            for (int i = 0; i < oldTable.size(); i++) {
                if (oldOccupied[i]) {
                    insertItem(oldTable[i]);
                }
            }
            // Rehash the elements from the old chains
            if (collisionMethod == SEPARATE_CHAINING) {
                for (int i = 0; i < oldChains.size(); i++) {
                    for (const auto& value : oldChains[i]) {
                        insertItem(value);
                    }
```

```
            }
        }
    }
```

Se encarga de multiplicar el tamaño de la tabla por 2 y le hace rehash a todos los items en caso de que el valor del factor sea mayor a 0.7

```
int nextIndex(int currentIndex, int value, int i) const {
        switch (collisionMethod) {
            case LINEAR_PROBING:
                return (currentIndex + 1) % tableSize;
            case QUADRATIC_PROBING:
                return (currentIndex + i * i) % tableSize;
            case DOUBLE_HASHING:
                return (currentIndex + secondHashFunction(value)) % tableSize;
            case ROBIN_HOOD:
                return (currentIndex + 1) % tableSize;
            case COALESCED:
                return next[currentIndex];
            default:
                return -1;
        }
    }
```

Se encarga de calcular el siguiente índice para sondear según el método de manejo de colisiones.

```
void insertSeparateChaining(int value) {
        int index = hashFunction(value);
        chains[index].push_back(value);
        itemCount++;
    }
```

Metodo de encadenamiento separado

Para este metodo primero se saca el index de la funcion hash, luego el valor se agrega a la lista en el índex, finalmente se aumenta el contado de los items.

```
void insertLinearProbing(int value) {
        int index = hashFunction(value);
        while (occupied[index]) {
            index = (index + 1) % tableSize;
        }
        table[index] = value;
        occupied[index] = true;
        itemCount++;
    }
```

Metodo de sondeo lineal

Para este metodo se saca el index de la funcion hash, luego checa si el index esta ocupado, en caso de que lo este se le suma al index y agrega al tamaño de la tabla, luego pasa a convertir el index a un valor, el index ocupado a true y se

aumenta el contado de los items.

```
void insertQuadraticProbing(int value) {
        int index = hashFunction(value);
        int i = 0;
        while (occupied[(index + i * i) % tableSize]) {
            i++;
        }
        table[(index + i * i) % tableSize] = value;
        occupied[(index + i * i) % tableSize] = true;
        itemCount++;
    }
```

Metodo de sondeo cuadratico

Para este metodo se saca el index de la funcion hash, luego se utiliza una fórmula de sondeo cuadrático para encontrar la siguiente ranura disponible, incrementando el contador de sonda hasta que se encuentra una ranura desocupada. para que al final el valor se inserta en el índice calculado, la ranura se marca como ocupada y se incrementa el recuento de elementos.

```
void insertDoubleHashing(int value) {
        int index = hashFunction(value);
        int stepSize = secondHashFunction(value);
        while (occupied[index]) {
            index = (index + stepSize) % tableSize;
        }
        table[index] = value;
        occupied[index] = true;
        itemCount++;
    }
```

Metodo de doble hash

Para este metodo la función hash principal calcula el índice inicial, luego el hash primario La función hash secundaria calcula el tamaño del paso para el hash doble. La función calcula el índice inicial, el siguiente espacio disponible se encuentra agregando repetidamente el tamaño del paso al índice actual hasta que se encuentre un espacio desocupado, el valor se inserta en el índice calculado, la ranura se marca como ocupada y se incrementa el recuento de elementos.