**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра АПУ**

отчет

**по лабораторной работе № 5**

**по дисциплине «Программирование»**

**Тема: Хеш-множества**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3371 |  | Манешов Д. В. |
| Преподаватель |  | Писарев А.С. |

Санкт-Петербург

2024

## Постановка задачи.

Необходимо реализовать структуру данных хеш-множества HashSet (для хеш-множества характерны некоторые особенности математических множеств. Например, последовательность элементов не имеет значение или элементы в множестве не повторяются)

Данная структура должна обладать следующим набором функций:

* void add(string) - добавление элемента в множество
* void remove(string) - удаляет из множество элемент (если такого элемента нет ничего не делает)
* clear() - удаление всех элементов коллекции
* bool contains(string) - Определяет, содержит ли объект HashSet указанный элемент.
* int count() - возвращает количество элементов в коллекции
* string ToArray()\* - конвертирует хеш-множество в динамический массив строк

Данная структура данных будет подвергнута тестированию автоматическими unit-тестами, поэтому наименование функций (и названия структуры) и сигнатуру изменять не следует.

Размерность и хеш-функцию определите самостоятельно (собственная реализация). Коллизии разрешайте, используя вспомогательные списки «конфликтующих» слов.

## Ход решения.

Данная программа реализует структуру данных хеш-множества HashSet, с внутренним управлением списками для разрешения коллизий. Это достигается благодаря шаблонному классу List, который управляет списком элементов типа string.

В данном блоке кода (рис. 1.1) происходит подключение заголовочного файла List.h, который, содержит определение и реализацию шаблонного класса List. Данный класс используется для управления списками объектов в программе. Также подключаются стандартные библиотеки ввода-вывода iostream, что позволяет выполнять операции ввода и вывода, например, чтение данных с клавиатуры и вывод информации на экран. Директива using namespace std; позволяет использовать стандартные классы и функции C++ без необходимости явного указания пространства имен std.

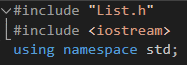


Рисунок 1.1 – Подключение библиотек и пространство имён

В данном фрагменте кода (рис. 1.2) представлен класс HashSet, который служит для реализации хеш-таблицы, способной хранить строки. Эта хеш-таблица структурирована с использованием массива указателей на списки для разрешения коллизий хешей методом цепочек. Количество различных "корзин" или "слотов" в хеш-таблице установлено равным 256, что определено в статической константе TABLE\_SIZE.

Хеш-функция в классе HashSet начинает работу с инициализации переменной hash значением 5381, используя популярный алгоритм хеширования "djb2", разработанный Дэниелом Дж. Бернштейном. Этот алгоритм использует алгоритм умножения на 33 для каждого символа в ключе, представляющем собой строку. Для каждого символа ключа выполняются следующие шаги:

* Инициализация значения hash. Функция начинается с установки начального значения hash равным 5381.
* Перебор символов строки. В цикле for, переменная c последовательно принимает значения каждого символа строки key.
* Вычисление хеша для каждого символа. Выражение ((hash << 5) + hash) + c представляет собой вычисление нового значения hash с учетом текущего символа. Подробно, это:
  + hash << 5: сдвиг значения хеша влево на 5 бит, что эквивалентно умножению на 32.
  + (hash << 5) + hash: после сдвига добавляется исходное значение hash, что в сумме дает умножение на 33.
  + ((hash << 5) + hash) + c: к результату добавляется ASCII-значение символа c.
* Остаток от деления. Завершается функция вычислением остатка от деления значения хеша на TABLE\_SIZE, что обеспечивает попадание результата в допустимые пределы индексов массива списков.

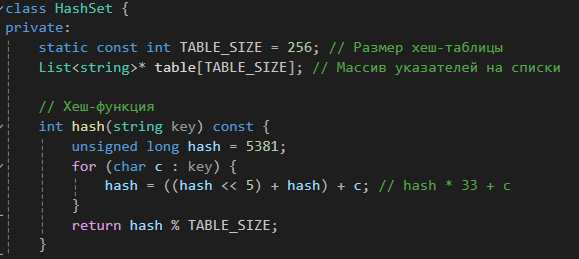


Рисунок 1.2 – Определение класса HashSet и его хеш-функции

В следующем фрагменте кода (рис. 1.3) в конструкторе класса HashSet происходит инициализация хеш-таблицы, которая включает в себя создание массива указателей на списки. Для каждого элемента массива (всего их TABLE\_SIZE, равное 256), создается новый экземпляр шаблонного класса List<string>. Это обеспечивает, что у каждой "корзины" хеш-таблицы есть собственный список для хранения строк, которые коллизионно попадают в одну и ту же корзину. Эти списки инициализируются как пустые, готовые к добавлению элементов.

В деструкторе класса HashSet происходит очистка ресурсов, занятых хеш-таблицей. Прежде всего, вызывается метод clear(), который очищает все списки, содержащиеся в массиве table. Метод clear() у каждого списка вызывает удаление всех его элементов, освобождая память, занятую строками. После полной очистки каждого списка происходит удаление самих объектов списков. Для каждого указателя в массиве table выполняется оператор delete, что приводит к деаллокации памяти, занятой объектом списка и его внутренними узлами. Это гарантирует, что вся динамически выделенная память корректно освобождена, предотвращая утечки памяти после удаления экземпляра HashSet.

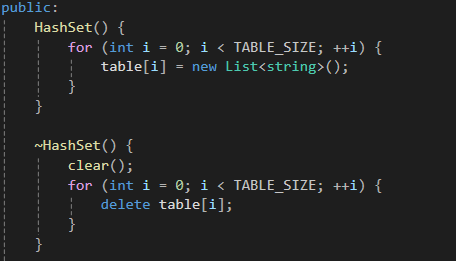


Рисунок 1.3 – Конструктор и деструктор класса HashSet

Следующий фрагмент кода (рис. 1.4) содержит реализацию метода add в классе HashSet, который предназначен для добавления строки data в хеш-таблицу. Он выполняет следующие шаги:

1. Вычисление хеш-индекса. Сначала для предоставленной строки data вычисляется хеш-значение с помощью вызова метода hash(data). Этот метод преобразует строку в числовое значение, которое затем используется для определения соответствующего индекса в массиве table. Остаток от деления хеша на размер таблицы (TABLE\_SIZE) дает индекс, который обеспечивает равномерное распределение строк по "корзинам" или слотам таблицы.
2. Проверка на наличие данных. Перед добавлением новой строки в таблицу выполняется проверка, не содержится ли уже такая строка в соответствующем списке. Для этого вызывается метод contains(data), который итеративно проверяет каждый элемент списка в слоте с индексом index на совпадение с data. Если строка уже присутствует в списке, то добавление не происходит, что предотвращает дублирование данных.
3. Добавление данных. Если проверка показывает, что строка data отсутствует в таблице, метод add(data) класса List<string> вызывается для списка, расположенного в выбранном слоте массива table[index]. Это добавляет новую строку в конец списка, связанного с этим индексом.

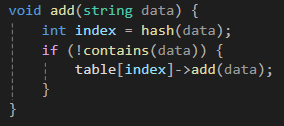


Рисунок 1.4 – Метод для добавления строки data в хеш-таблицу

Метод remove (рис. 1.5) в классе HashSet предназначен для удаления строки data из хеш-таблицы. Этот метод включает в себя несколько ключевых шагов для эффективного управления данными:

1. Вычисление хеш-индекса. Сначала для предоставленной строки data вычисляется хеш-значение, используя метод hash(data). Этот метод преобразует строку в числовой индекс, который указывает на соответствующий слот в массиве table. Хеш-значение позволяет быстро локализовать, в каком списке внутри массива следует искать строку.
2. Определение размера списка. Полученный индекс используется для доступа к соответствующему списку в массиве table, и для этого списка вызывается метод count(), который возвращает количество элементов в списке. Это количество используется для определения количества итераций в последующем цикле.
3. Поиск и удаление элемента. Затем выполняется итерация по элементам списка с начала до конца. В каждой итерации элемент, на который указывает индекс i, извлекается с помощью метода elementAt(i). Если значение этого элемента совпадает с data, то строка найдена, и выполняется её удаление из списка с помощью метода removeAt(i). После удаления элемента цикл прерывается оператором break для предотвращения дальнейшего перебора и возможных ошибок индексации после изменения размера списка.

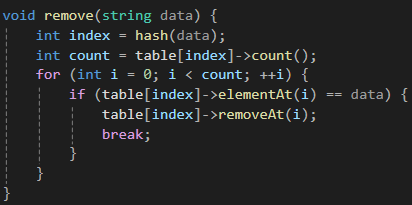


Рисунок 1.5 – Метод для удаления строки data из хеш-таблицы

Метод contains (рис. 1.6) в классе HashSet проверяет наличие заданной строки data в хеш-таблице. Этот процесс начинается с вычисления хеш-значения для строки data с использованием метода hash(data). Полученное хеш-значение преобразуется в индекс для массива table путем взятия остатка от деления на размер таблицы, TABLE\_SIZE. Этот индекс определяет, в каком из списков массива следует искать строку.

После определения соответствующего списка, метод вызывает функцию count() для этого списка, чтобы узнать, сколько элементов в нем содержится. Зная количество элементов, метод затем перебирает каждый элемент списка с помощью цикла. В каждой итерации цикла текущий элемент списка сравнивается со строкой data с использованием метода elementAt(i), который извлекает элемент списка по индексу i.

Если в какой-то момент цикла находится элемент, который совпадает со строкой data, метод немедленно возвращает true, что означает, что строка найдена в хеш-таблице. Если же после проверки всех элементов списка совпадение так и не обнаружено, метод возвращает false, указывая на отсутствие строки в таблице.

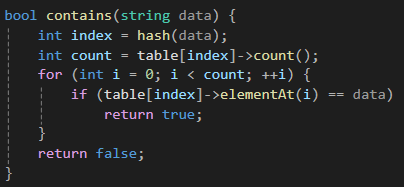


Рисунок 1.6 – Метод для проверки наличия заданной строки data в хеш-таблице

В данном фрагменте кода (рис. 1.7) представлены два метода clear и count.

Метод clear в классе HashSet занимается очисткой хеш-таблицы, удаляя все строки из всех списков, составляющих таблицу. Для этого метод проходит по каждому списку в массиве table, который содержит TABLE\_SIZE элементов, и вызывает метод clear() для каждого списка. Этот метод списка удаляет все его элементы, освобождая занятые ресурсы и оставляя каждый список пустым. Это обеспечивает полное удаление всех данных из хеш-таблицы, что является важной операцией для управления памятью и поддержания чистоты данных в приложении, особенно в ситуациях, когда необходимо освободить ресурсы или переинициализировать структуру данных.

Метод count выполняет подсчет общего количества элементов, содержащихся в хеш-таблице. Он начинает с инициализации переменной total значением ноль. Затем метод перебирает каждый список в массиве table и добавляет к total количество элементов в каждом списке, используя метод count() каждого списка. Этот метод возвращает количество элементов в списке, и таким образом, суммируя количество элементов всех списков, метод count определяет общее количество элементов во всей хеш-таблице. Это полезно для многих операций, включая мониторинг использования хеш-таблицы и оптимизацию производительности, так как позволяет оценить, насколько плотно или разреженно заполнена таблица.

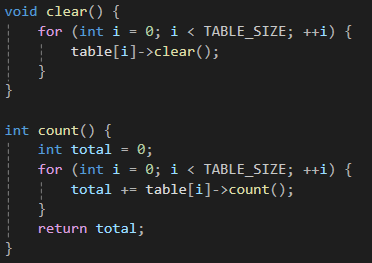


Рисунок 1.7 – Метод clear для очистки хеш-таблицы и метод count для подсчета общего количества элементов, содержащихся в хеш-таблице

Метод ToArray (рис. 1.8) в классе HashSet преобразует все элементы, хранящиеся в хеш-таблице, в массив строк. Этот метод начинается с вызова функции count() для определения общего количества элементов в таблице, что необходимо для инициализации массива строк с подходящим размером. Затем создается массив строк array, где каждая строка из хеш-таблицы будет сохранена.

Процесс перебора элементов хеш-таблицы организован через двойной цикл, где внешний цикл проходит по каждой корзине хеш-таблицы, а внутренний цикл перебирает элементы в каждом списке корзины. Для каждого элемента списка метод elementAt(j) извлекает строку, которая затем добавляется в массив array на текущую позицию индекса idx. Индекс увеличивается после каждого добавления, чтобы следующий элемент помещался на следующую позицию в массиве.

После того как все элементы хеш-таблицы добавлены в массив, метод завершается и возвращает указатель на этот массив строк. Этот массив можно использовать для вывода данных на экран, их анализа или передачи в другие части программы.

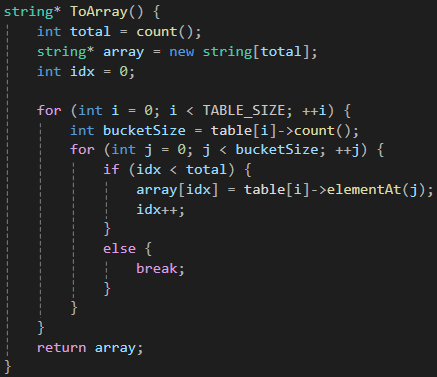


Рисунок 1.8 – Метод для преобразования всех элементов, хранящихся в хеш-таблице, в массив строк

**Выводы.**

В ходе выполнения лабораторной работы была разработана программа, реализующая структуру данных хеш-множества HashSet, обладающая особенностями математических множеств, такими как неупорядоченность и уникальность элементов. Программа включает методы для добавления, удаления, проверки наличия элементов, очистки множества, подсчёта элементов и конвертации множества в массив строк. Разрешение коллизий осуществляется с использованием вспомогательных списков, что обеспечивает эффективную работу с данными даже при совпадении хеш-значений. Эта структура данных идеально подходит для управления большими объемами уникальных данных, гарантируя высокую производительность всех операций.

# Список использованной литературы

1. Павловская Т.А., Чаевников В.В., Юрков Н.К. Программирование на языке С++. Электронное методическое пособие. СПб.: Издательство СПбГЭТУ «ЛЭТИ». 2014.
2. Практикум по процедурному программированию на языке C++ С. А. Ивановский, А. А. Лисс, В. П. Самойленко, О. М. Шолохова. Практикум по процедурному программированию на языке C++: учеб. пособие. СПб.: Издательство СПбГЭТУ«ЛЭТИ», 2016.
3. Справочник по стандартной библиотеке C++ (STL) // Справочный сайт learn.microsoft.com URL: https://learn.microsoft.com/ru-ru/cpp/standard-library/cpp-standard-library-reference?view=msvc-170 (дата обращения 09.03.2024).

Документация по языку C++ // Справочный сайт learn.microsoft.com URL: https://learn.microsoft.com/ru-ru/cpp/cpp/?view=msvc-170 (дата обращения 10.03.2024).

# Приложение

## Код написанной программы

Файл «Lab5.cpp»:

#pragma once

#include <string>

#include <iostream>

#include <windows.h>

#include <stdexcept>

template <typename T>

class List {

private:

struct Node {

T data;

Node\* prev;

Node\* next;

Node(T value) {

data = value;

prev = next = nullptr;

}

};

Node\* head;

Node\* tail;

int size;

Node\* currentNode;

int currentIndex;

Node\* getNode(int index) {

// Оптимизация: если запрашиваемый индекс близок к последнему

// использованному, идём от currentNode

if (currentNode != nullptr && abs(index - currentIndex) < size / 2) {

Node\* current = currentNode;

if (index > currentIndex) {

for (int i = currentIndex; i < index; i++) {

current = current->next;

}

}

else {

for (int i = currentIndex; i > index; i--) {

current = current->prev;

}

}

currentNode = current;

currentIndex = index;

return current;

}

// В остальных случаях идём от начала или конца списка

Node\* current;

if (index < size / 2) {

current = head;

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

current = current->next;

}

}

else {

current = tail;

for (int i = size - 1; i > index; i--) {

current = current->prev;

}

}

currentNode = current;

currentIndex = index;

return current;

}

public:

List() {

head = tail = currentNode = nullptr;

size = currentIndex = 0;

}

~List() {

clear();

}

void add(T data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

}

else {

tail->next = newNode;

newNode->prev = tail;

}

tail = newNode;

size++;

}

void insert(int index, T data) {

if (index < 0 || index > size) {

return;

}

if (index == size) {

add(data);

return;

}

Node\* newNode = new Node(data);

if (index == 0) {

newNode->next = head;

head->prev = newNode;

head = newNode;

}

else {

Node\* curr = getNode(index);

newNode->prev = curr->prev;

newNode->next = curr;

curr->prev = newNode;

newNode->prev->next = newNode;

}

currentNode = newNode;

currentIndex = index;

size++;

}

void removeAt(int index) {

if (index < 0 || index >= size) {

return;

}

Node\* temp = nullptr;

if (index == 0) {

temp = head;

head = head->next;

if (head) {

head->prev = nullptr;

}

else {

tail = nullptr;

}

}

else if (index == size - 1) {

temp = tail;

tail = tail->prev;

tail->next = nullptr;

}

else {

temp = getNode(index);

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

}

if (temp == currentNode) {

currentNode = temp->prev;

currentIndex = max(0, index - 1);

}

else if (index < currentIndex) {

currentIndex--;

}

delete temp;

size--;

if (size == 0) {

head = tail = currentNode = nullptr;

currentIndex = 0;

}

}

T elementAt(int index) {

if (index < 0 || index >= size) {

throw ("");

}

if (index == currentIndex && currentNode != nullptr) {

return currentNode->data;

}

return getNode(index)->data;

}

int count() {

return size;

}

void clear() {

Node\* current = head;

while (current) {

Node\* next = current->next;

delete current;

current = next;

}

head = tail = nullptr;

size = 0;

}

};

Файл «List.h»:

#include "List.h"

#include <iostream>

using namespace std;

class HashSet {

private:

static const int TABLE\_SIZE = 256; // Размер хеш-таблицы

List<string>\* table[TABLE\_SIZE]; // Массив указателей на списки

// Хеш-функция

int hash(string key) const {

unsigned long hash = 5381;

for (char c : key) {

hash = ((hash << 5) + hash) + c; // hash \* 33 + c

}

return hash % TABLE\_SIZE;

}

public:

HashSet() {

for (int i = 0; i < TABLE\_SIZE; ++i) {

table[i] = new List<string>();

}

}

~HashSet() {

clear();

for (int i = 0; i < TABLE\_SIZE; ++i) {

delete table[i];

}

}

void add(string data) {

int index = hash(data);

if (!contains(data)) {

table[index]->add(data);

}

}

void remove(string data) {

int index = hash(data);

int count = table[index]->count();

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

if (table[index]->elementAt(i) == data) {

table[index]->removeAt(i);

break;

}

}

}

bool contains(string data) {

int index = hash(data);

int count = table[index]->count();

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

if (table[index]->elementAt(i) == data)

return true;

}

return false;

}

void clear() {

for (int i = 0; i < TABLE\_SIZE; ++i) {

table[i]->clear();

}

}

int count() {

int total = 0;

for (int i = 0; i < TABLE\_SIZE; ++i) {

total += table[i]->count();

}

return total;

}

string\* ToArray() {

int total = count();

string\* array = new string[total];

int idx = 0;

for (int i = 0; i < TABLE\_SIZE; ++i) {

int bucketSize = table[i]->count();

for (int j = 0; j < bucketSize; ++j) {

if (idx < total) {

array[idx] = table[i]->elementAt(j);

idx++;

}

else {

break;

}

}

}

return array;

}

};

int main() {

}

Файл «data1.txt»:

banana

apple

date

fig

cherry

Файл «result1.txt»:

Количество элементов после добавления: 5

Содержит 'banana': Да

Количество элементов после удаления 'banana': 4

Содержит 'banana' после удаления: Нет

Элементы в HashSet: apple date fig cherry

Количество элементов после очистки: 0