**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра АПУ**

отчет

**по лабораторной работе № 9**

**по дисциплине «Программирование»**

**Тема: Хеш-таблица с использованием бинарных деревьев поиска для разрешения коллизий**

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

Санкт-Петербург

2024

## Постановка задачи.

Необходимо реализовать структуру данных хеш-множества, которое хранит деревья бинарного поиска HashTree

Данная структура должна обладать следующим набором функций:

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

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

## Ход решения.

Данная программа реализует структуру данных под названием HashTree, которая представляет собой хеш-таблицу, где каждое значение хранится в виде дерева. Эта структура данных сочетает преимущества хеш-таблиц и деревьев для эффективного хранения и поиска данных.

В данном блоке кода (рис. 1.1) строка #include <iostream> включает стандартную библиотеку ввода-вывода C++, которая предоставляет функциональность для работы с потоками ввода и вывода, такими как cin, cout, cerr и clog. Это необходимо для того, чтобы программа могла выводить информацию на экран и считывать данные с клавиатуры. Строка #include "Tree.h" подключает пользовательский заголовочный файл "Tree.h". Этот заголовочный файл, вероятно, содержит объявления классов, функций и других элементов, необходимых для работы с деревьями данных в программе. Включение этого файла позволяет программе использовать все объявления, содержащиеся в "Tree.h". Наконец, строка using namespace std; указывает на использование стандартного пространства имен. Это позволяет использовать элементы стандартной библиотеки C++ (такие как cout, cin, vector и другие) без необходимости явного указания, что они принадлежат пространству имен std. В результате, код становится более читабельным и менее загроможденным.

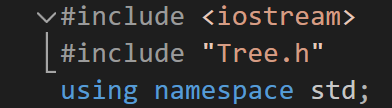


Рисунок 1.1 – Подключение и настройка заголовочных файлов

В данном фрагменте кода (рис. 1.2) определяется, класс HashTree включает приватные члены, которые недоступны извне класса, обеспечивая инкапсуляцию и защиту данных. Одним из таких членов является статическая константа TABLE\_SIZE, которая определяет размер хеш-таблицы и равна 256. Статическая константа означает, что это значение является общим для всех экземпляров класса и не изменяется.

Другим важным членом класса является массив table, состоящий из TABLE\_SIZE элементов типа Tree. Каждый элемент этого массива представляет собой дерево, что позволяет организовать структуру данных в виде хеш-таблицы, где в каждой ячейке таблицы находится дерево. Это может быть полезно для различных операций, таких как вставка, поиск и удаление элементов, предоставляя эффективный доступ к данным.

Метод hash представляет собой приватную функцию класса, которая вычисляет хеш для заданного целочисленного значения value. Внутри метода используется алгоритм хеширования, основанный на формуле: hash = 5381 \* 33 + value. Такой подход к хешированию позволяет получить равномерное распределение значений хеша. В конце метод возвращает значение хеша, взятое по модулю TABLE\_SIZE, что гарантирует, что индекс будет находиться в пределах допустимого диапазона массива.

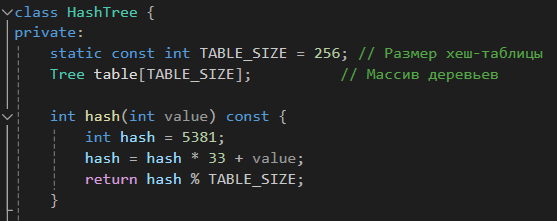


Рисунок 1.2 – Объявления класса HashTree

Данный фрагмент кода (рис. 1.3) представляет из себя конструктор по умолчанию для класса HashTree, который создает объект класса, инициализируя его без дополнительных действий. Деструктор класса HashTree, вызываемый при уничтожении объекта, вызывает метод clear(), который проходит по всем элементам массива table и очищает каждое дерево. Это гарантирует корректное освобождение всех выделенных ресурсов и предотвращает утечки памяти при удалении объекта класса HashTree.

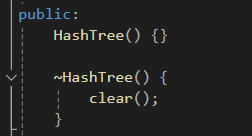


Рисунок 1.3 – Конструктор и деструктор класса HashTree

Метод add (рис. 1.4) класса HashTree предназначен для добавления нового значения в хеш-таблицу. Вначале вычисляется хеш индекса для данного значения с использованием функции hash, которая возвращает индекс, по которому будет храниться значение. Затем проверяется, содержится ли это значение в дереве по этому индексу массива table с помощью метода contains. Если значение отсутствует, оно добавляется в дерево с помощью метода add. Этот процесс обеспечивает, что каждое значение хранится только один раз в соответствующем дереве, предотвращая дублирование данных в хеш-таблице.

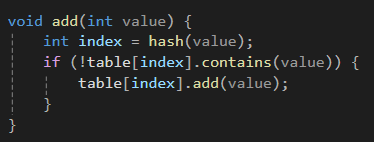


Рисунок 1.4 – Метод add для добавления нового значения в хеш-таблицу

Метод remove (рис. 1.5) класса HashTree отвечает за удаление значения из хеш-таблицы. Вначале с помощью функции hash вычисляется индекс для данного значения. Далее проверяется, содержится ли это значение в дереве, находящемся по вычисленному индексу в массиве table, с помощью метода contains. Если значение присутствует, вызывается метод remove, который удаляет это значение из дерева. Таким образом, метод обеспечивает корректное удаление значения из соответствующего дерева в хеш-таблице, если оно там существует.

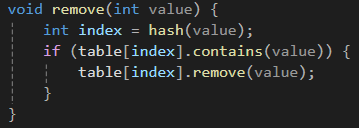


Рисунок 1.5 – Метод remove для удаления значения из хеш-таблицы

Метод clear (рис. 1.6) класса HashTree предназначен для очистки всей хеш-таблицы. Внутри метода используется цикл for, который проходит по всем элементам массива table, размер которого определяется константой TABLE\_SIZE. Для каждого элемента массива вызывается метод clear, который очищает соответствующее дерево, находящееся в данной ячейке массива. Таким образом, метод clear гарантирует, что все деревья в хеш-таблице будут полностью очищены, удаляя все значения и освобождая ресурсы, ассоциированные с каждым деревом.

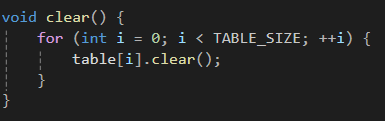


Рисунок 1.6 – Метод clear для очистки всей хеш-таблицы

Метод contains (рис. 1.7) класса HashTree проверяет, содержится ли заданное значение в хеш-таблице. Сначала вычисляется хеш-индекс для данного значения с использованием функции hash, что позволяет определить, в каком дереве массива table искать это значение. Затем вызывается метод contains для соответствующего дерева по вычисленному индексу. Метод возвращает true, если значение найдено в дереве, и false в противном случае. Таким образом, contains эффективно определяет наличие значения в хеш-таблице, используя комбинацию хеширования и деревьев для быстрого доступа.

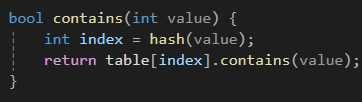


Рисунок 1.7 – Метод contains, который проверяет, содержится ли заданное значение в хеш-таблице

Метод count (рис. 1.8) класса HashTree вычисляет общее количество элементов, хранящихся в хеш-таблице. Инициализируется переменная totalCount, в которой будет накапливаться количество элементов. С помощью цикла for происходит итерация по всем элементам массива table, размер которого определяется константой TABLE\_SIZE. Для каждого дерева вызывается метод count, который возвращает количество элементов в данном дереве, и это значение добавляется к totalCount. По завершении цикла метод возвращает общее количество элементов, накопленное в totalCount. Таким образом, метод count суммирует количество элементов во всех деревьях, обеспечивая полный подсчет всех значений в хеш-таблице.

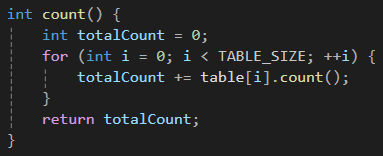


Рисунок 1.8 – Метод count, который вычисляет общее количество элементов, хранящихся в хеш-таблице

Метод ToArray (рис. 1.9) класса HashTree предназначен для преобразования всех элементов хеш-таблицы в массив. Сначала определяется общее количество элементов в хеш-таблице, вызывая метод count, результат которого сохраняется в переменной totalSize. Затем выделяется память для массива array размера totalSize для хранения всех элементов. Инициализируется переменная index, которая будет использоваться для отслеживания текущей позиции в массиве. С помощью цикла for, который проходит по всем элементам массива table, вызывается метод fillArray для каждого дерева, который заполняет массив array элементами дерева и обновляет значение index. По завершении цикла метод возвращает указатель на заполненный массив array. Таким образом, метод ToArray собирает все элементы из хеш-таблицы и возвращает их в виде одномерного массива.

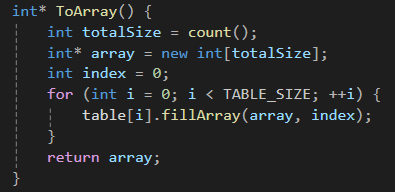


Рисунок 1.9 – Метод ToArray для преобразования всех элементов хеш-таблицы в массив

Функция main (рис. 1.10) является точкой входа в программу и обеспечивает интерактивное взаимодействие пользователя с объектом класса HashTree. Вначале создается объект ht класса HashTree и объявляются переменные choice для хранения команды пользователя и data для ввода данных. В бесконечном цикле while (true) программа ожидает ввода команды от пользователя.

При вводе команды add считывается значение data, которое добавляется в хеш-таблицу с помощью метода add. Для команды remove также считывается значение data и вызывается метод remove, удаляющий это значение из таблицы. Команда clear вызывает метод clear, очищающий хеш-таблицу.

Команда count вызывает метод count, который возвращает и выводит общее количество элементов в хеш-таблице. Для команды contains вводится значение data, и метод contains проверяет наличие этого значения в таблице, выводя "Yes" или "No" в зависимости от результата.

Команда toArray вызывает метод ToArray, который преобразует элементы хеш-таблицы в массив. Затем массив выводится на экран, и выделенная память под массив освобождается с помощью delete[] array.

Команда stop прерывает выполнение цикла и завершает программу. Если введена неизвестная команда, программа выводит сообщение "Unknown command". Таким образом, функция main обеспечивает полный цикл интерактивного управления хеш-таблицей, поддерживая добавление, удаление, очистку, подсчет элементов, проверку наличия и преобразование в массив.

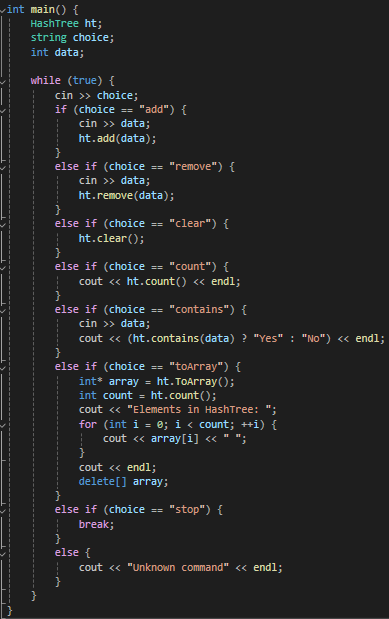


Рисунок 1.8 – Функция main

**Выводы.**

В ходе выполнения лабораторной работы была разработана программа, реализующая структуру данных хеш-множества, которая хранит деревья бинарного поиска HashTree. Эта структура обеспечивает эффективное хранение и управление элементами, предоставляя произвольный доступ к данным. Включен следующий набор функций для управления элементами множества: добавление элемента (void add(int)), удаление элемента (void remove(int)), очистка всей коллекции (clear()), проверка наличия элемента (bool contains(int)), подсчет количества элементов в коллекции (int count()) и преобразование хеш-множества в динамический массив чисел (int\* ToArray()). Программа также включает пользовательский интерфейс, позволяющий взаимодействовать с хеш-таблицей через консольные команды, что обеспечивает удобный доступ ко всем функциям структуры.

# 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 «Lab9.cpp»:

#include <iostream>

#include "Tree.h"

using namespace std;

class HashTree {

private:

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

Tree table[TABLE\_SIZE]; // Массив деревьев

int hash(int value) const {

int hash = 5381;

hash = hash \* 33 + value;

return hash % TABLE\_SIZE;

}

public:

HashTree() {}

~HashTree() {

clear();

}

void add(int value) {

int index = hash(value);

if (!table[index].contains(value)) {

table[index].add(value);

}

}

void remove(int value) {

int index = hash(value);

if (table[index].contains(value)) {

table[index].remove(value);

}

}

void clear() {

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

table[i].clear();

}

}

bool contains(int value) {

int index = hash(value);

return table[index].contains(value);

}

int count() {

int totalCount = 0;

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

totalCount += table[i].count();

}

return totalCount;

}

int\* ToArray() {

int totalSize = count();

int\* array = new int[totalSize];

int index = 0;

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

table[i].fillArray(array, index);

}

return array;

}

};

int main() {

HashTree ht;

string choice;

int data;

while (true) {

cin >> choice;

if (choice == "add") {

cin >> data;

ht.add(data);

}

else if (choice == "remove") {

cin >> data;

ht.remove(data);

}

else if (choice == "clear") {

ht.clear();

}

else if (choice == "count") {

cout << ht.count() << endl;

}

else if (choice == "contains") {

cin >> data;

cout << (ht.contains(data) ? "Yes" : "No") << endl;

}

else if (choice == "toArray") {

int\* array = ht.ToArray();

int count = ht.count();

cout << "Elements in HashTree: ";

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

cout << array[i] << " ";

}

cout << endl;

delete[] array;

}

else if (choice == "stop") {

break;

}

else {

cout << "Unknown command" << endl;

}

}

}

Файл «Tree.h»:

#pragma once

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = nullptr;

right = nullptr;

}

};

class Tree {

private:

int size;

Node\* root;

void fillArrayHelper(Node\* node, int\*& array, int& index) {

if (node == nullptr) return;

fillArrayHelper(node->left, array, index);

array[index++] = node->data;

fillArrayHelper(node->right, array, index);

}

public:

Tree() {

size = 0;

root = nullptr;

}

~Tree() {

clear();

}

void add(int value) {

Node\* newNode = new Node(value);

if (!root) {

root = newNode;

size++;

return;

}

Node\* curr = root;

while (true) {

if (value < curr->data) {

if (!curr->left) {

curr->left = newNode;

size++;

break;

}

else {

curr = curr->left;

}

}

else if (value > curr->data) {

if (!curr->right) {

curr->right = newNode;

size++;

break;

}

else {

curr = curr->right;

}

}

else {

delete newNode;

break;

}

}

}

bool contains(int value) {

Node\* curr = root;

while (curr) {

if (value < curr->data) {

curr = curr->left;

}

else if (value > curr->data) {

curr = curr->right;

}

else {

return true;

}

}

return false;

}

void removeHelper(Node\*& node, int value) {

if (!node) return; // Узел не найден

if (value < node->data) {

removeHelper(node->left, value);

}

else if (value > node->data) {

removeHelper(node->right, value);

}

else { // Найден узел, который нужно удалить

if (!node->left && !node->right) { // Узел является листом

delete node;

node = nullptr;

size--;

}

else if (node->left && !node->right) { // Узел имеет только левого ребенка

Node\* temp = node;

node = node->left;

delete temp;

size--;

}

else if (!node->left && node->right) { // Узел имеет только правого ребенка

Node\* temp = node;

node = node->right;

delete temp;

size--;

}

else { // Узел имеет двух детей

Node\* minNode = node->right;

Node\* parentOfMinNode = node;

if (!minNode->left) { // Минимальный узел находится непосредственно справа

node->data = minNode->data;

node->right = minNode->right; // Подвесим правое поддерево минимального узла к текущему узлу

delete minNode;

}

else {

while (minNode->left) { // Находим минимальный узел в правом поддереве

parentOfMinNode = minNode;

minNode = minNode->left;

}

node->data = minNode->data; // Заменяем данные узла на минимальные из правого поддерева

removeHelper(parentOfMinNode->left, minNode->data); // Удаляем минимальный узел

}

}

}

}

void remove(int value) {

removeHelper(root, value);

}

int count() {

return size;

}

void clearNode(Node\* node) {

if (!node) return;

clearNode(node->left);

clearNode(node->right);

delete node;

}

void clear() {

clearNode(root);

root = nullptr;

size = 0;

}

void fillArray(int\* array, int& index) {

fillArrayHelper(root, array, index);

}

};

Файл data1-1.txt:

add 5

add 15

add 25

contains 15

contains 10

count

remove 15

contains 15

count

stop

Файл report1-1.txt:

Yes

No

3

No

2

Файл data2-1.txt:

add 100

add 200

add 300

add 400

count

remove 100

remove 200

count

toArray

clear

count

stop

Файл report2-1.txt:

4

2

Elements in HashTree: 300 400

0

Файл data3-1.txt:

add 7

add 14

add 21

add 28

contains 14

contains 22

count

remove 7

remove 28

count

stop

Файл report3-1.txt:

Yes

No

4

2

Файл data4-1.txt:

add 8

add 16

add 24

add 32

add 40

count

clear

count

stop

Файл report4-1.txt:

5

0