Министерство образования Республики Беларусь

Учреждение образования

«Брестский государственный технический университет»

Кафедра ИИТ

Лабораторная работа №1

По дисциплине « ***Алгоритмы и структуры данных*** »

Тема: ***«Бинарное дерево поиска и хеш-таблица»***

Выполнил:

Студент 2 курса

Группы ПО-11(2)

Сымоник И.А

Проверила:

Глущенко Т.А

**Цель работы**: изучить структуры данных бинарное дерево поиска, хеш-таблица и алгоритм Рабина-Карпа.

**Ход работы**

**Эксперимент 1** Сравнение эффективности поиска элементов в бинарном дереве поиска и хеш-таблице в среднем случае (average case)

|  |  |  |  |
| --- | --- | --- | --- |
| # | Количество элементов в словаре | Время выполнения функции bstree\_lookup, с | Время выполнения функции hashtab\_lookup, с |
| 1 | 10 000 | 0,000007 | 0,000012 |
| 2 | 20 000 | 0,000008 | 0,000010 |
| 3 | 30 000 | 0,000007 | 0,000016 |
| 4 | 40 000 | 0,000008 | 0,000018 |
| 5 | 50 000 | 0,000015 | 0,000038 |
| 6 | 60 000 | 0,000010 | 0,000031 |
| 7 | 70 000 | 0,000008 | 0,000061 |
| 8 | 80 000 | 0,000009 | 0,000022 |
| 9 | 90 000 | 0,000009 | 0,000043 |
| 10 | 100 000 | 0,000010 | 0,000112 |
| 11 | 110 000 | 0,000009 | 0,000026 |
| 12 | 120 000 | 0,000012 | 0,000074 |
| 13 | 130 000 | 0,000008 | 0,000050 |
| 14 | 140 000 | 0,000012 | 0,000054 |
| 15 | 150 000 | 0,000015 | 0,000062 |
| 16 | 160 000 | 0,000009 | 0,000041 |
| 17 | 170 000 | 0,000008 | 0,000034 |
| 18 | 180 000 | 0,000007 | 0,000043 |
| 19 | 190 000 | 0,000012 | 0,000176 |
| 20 | 200 000 | 0,000010 | 0,000280 |

**Эксперимент 2** Сравнение эффективности добавления элементов в бинарное дерево поиска и хеш-таблицу

|  |  |  |  |
| --- | --- | --- | --- |
| # | Количество элементов в словаре | Время выполнения функции bstree\_add, с | Время выполнения функции hashtab\_add, с |
| 1 | 10 000 | 0,131 | 0,138 |
| 2 | 20 000 | 0,284 | 0,321 |
| 3 | 30 000 | 0,404 | 0,561 |
| 4 | 40 000 | 0,558 | 0,831 |
| 5 | 50 000 | 0,707 | 1,246 |
| 6 | 60 000 | 0,843 | 1,657 |
| 7 | 70 000 | 0,991 | 2,145 |
| 8 | 80 000 | 1,130 | 2,643 |
| 9 | 90 000 | 1,339 | 3,146 |
| 10 | 100 000 | 1,444 | 4,265 |
| 11 | 110 000 | 1,625 | 4,600 |
| 12 | 120 000 | 1,785 | 5,870 |
| 13 | 130 000 | 1,920 | 6,648 |
| 14 | 140 000 | 2,067 | 7,731 |
| 15 | 150 000 | 2,243 | 8,998 |
| 16 | 160 000 | 2,410 | 10,771 |
| 17 | 170 000 | 2,666 | 12,675 |
| 18 | 180 000 | 2,787 | 13,860 |
| 19 | 190 000 | 2,891 | 16,056 |
| 20 | 200 000 | 3,091 | 17,594 |

**Эксперимент 3** Сравнение эффективности поиска элементов в бинарном

дереве поиска и хеш-таблице в худшем случае (worst case)

|  |  |  |  |
| --- | --- | --- | --- |
| # | Количество элементов в словаре | Время выполнения функции bstree\_lookup, с | Время выполнения функции hashtab\_lookup, с |
| 1 | 10 000 | 0,000512 | 0,000007 |
| 2 | 20 000 | 0,000665 | 0,000052 |
| 3 | 30 000 | 0,000744 | 0,000064 |
| 4 | 40 000 | 0,000843 | 0,000066 |
| 5 | 50 000 | 0,000926 | 0,000073 |
| 6 | 60 000 | 0,001078 | 0,000095 |
| 7 | 70 000 | 0,001192 | 0,000127 |
| 8 | 80 000 | 0,001326 | 0,000142 |
| 9 | 90 000 | 0,001551 | 0,000167 |
| 10 | 100 000 | 0,001728 | 0,000170 |
| 11 | 110 000 | 0,001942 | 0,000183 |
| 12 | 120 000 | 0,002001 | 0,000158 |
| 13 | 130 000 | 0,002336 | 0,000190 |
| 14 | 140 000 | 0,002692 | 0,000214 |
| 15 | 150 000 | 0,002789 | 0,000221 |
| 16 | 160 000 | 0,002985 | 0,000236 |
| 17 | 170 000 | 0,003291 | 0,000234 |
| 18 | 180 000 | 0,003341 | 0,000257 |
| 19 | 190 000 | 0,003671 | 0,000280 |
| 20 | 200 000 | 0,003811 | 0,000282 |

**Эксперимент 4** Исследование эффективности поиска минимального элементов бинарном дереве поиска в худшем и среднем случаях

|  |  |  |  |
| --- | --- | --- | --- |
| # | Количество элементов в словаре | Время выполнения функции bstree\_min(worst case), нс | Время выполнения функции hashtab\_min, нс |
| 1 | 10 000 | 500 | 400 |
| 2 | 20 000 | 800 | 900 |
| 3 | 30 000 | 900 | 600 |
| 4 | 40 000 | 900 | 1300 |
| 5 | 50 000 | 1400 | 1100 |
| 6 | 60 000 | 1500 | 800 |
| 7 | 70 000 | 1700 | 1200 |
| 8 | 80 000 | 1600 | 600 |
| 9 | 90 000 | 500 | 900 |
| 10 | 100 000 | 2100 | 1000 |
| 11 | 110 000 | 1400 | 600 |
| 12 | 120 000 | 2000 | 1300 |
| 13 | 130 000 | 800 | 1700 |
| 14 | 140 000 | 1000 | 1400 |
| 15 | 150 000 | 900 | 900 |
| 16 | 160 000 | 900 | 800 |
| 17 | 170 000 | 1000 | 1100 |
| 18 | 180 000 | 700 | 600 |
| 19 | 190 000 | 800 | 800 |
| 20 | 200 000 | 900 | 500 |

**Эксперимент 5** Исследование эффективности поиска максимального

элемента в бинарном дереве поиска в худшем и среднем случаях

|  |  |  |  |
| --- | --- | --- | --- |
| # | Количество элементов в словаре | Время выполнения функции bstree\_max(worst case), нс | Время выполнения функции hashtab\_max, нс |
| 1 | 10 000 | 32000 | 300 |
| 2 | 20 000 | 64400 | 400 |
| 3 | 30 000 | 23000 | 400 |
| 4 | 40 000 | 19900 | 500 |
| 5 | 50 000 | 20800 | 400 |
| 6 | 60 000 | 19800 | 300 |
| 7 | 70 000 | 20200 | 500 |
| 8 | 80 000 | 35500 | 400 |
| 9 | 90 000 | 52400 | 300 |
| 10 | 100 000 | 100000 | 400 |
| 11 | 110 000 | 104100 | 300 |
| 12 | 120 000 | 95400 | 300 |
| 13 | 130 000 | 71200 | 400 |
| 14 | 140 000 | 98500 | 300 |
| 15 | 150 000 | 103200 | 400 |
| 16 | 160 000 | 107700 | 400 |
| 17 | 170 000 | 116000 | 400 |
| 18 | 180 000 | 129100 | 400 |
| 19 | 190 000 | 132500 | 400 |
| 20 | 200 000 | 138200 | 400 |

1. Объясните, почему временная сложность операции поиска в *сбалансированном* дереве поиска есть *O(log n).*

С каждой итерацией поиска текущий узел опускается на один уровень вниз. Поскольку каждый уровень дерева делит количество возможных элементов пополам, время поиска в сбалансированном дереве растет логарифмически относительно количества элементов

1. Укажите двойное неравенство для высоты бинарного дерева поиска.

log2 N < h ≤ 1+ log2 N.

где:

- h - высота бинарного дерева поиска,

- n - количество элементов в дереве

1. Укажите алгоритмы построения различных *hash-функций*.

Есть несколько алгоритмов для построения различных hash-функций. Некоторые из них включают:

1. Метод деления (Division method): Этот метод вычисляет хэш-значение путем деления ключа на размер хэш-таблицы и использования остатка от деления. Формула выглядит так: hash(key) = key % table\_size.

2. Метод умножения (Multiplication method): В этом методе ключ умножается на некоторую константу, извлекается дробная часть результата и умножается на размер хэш-таблицы. Формула выглядит так: hash(key) = floor(table\_size \* (key \* A mod 1)), где A - константа, 0 < A < 1.

3. Метод сложения (Additive method): В этом методе каждый символ ключа преобразуется в числовое значение, а затем все числа суммируются. Формула выглядит так: hash(key) = (key[0] + key[1] + ... + key[n-1]) % table\_size, где key[i] - числовое значение i-го символа ключа.

4. Метод смешивания (Mixing method): Этот метод использует операции смешивания, такие как побитовое исключающее ИЛИ (XOR), сдвиги и побитовые операции. Он обычно применяется к каждому символу ключа и комбинирует их значения для генерации хэш-значения.

5. Криптографические hash-функции: Криптографические hash-функции, такие как MD5, SHA-1 и SHA-256, используются для обеспечения безопасности и имеют свойства, такие как стойкость к коллизиям и уникальность. Они основаны на сложных математических алгоритмах.

1. Что такое коллизии и какие существуют методы борьбы с коллизиями.

Коллизия – это совпадение значения хэш-функции для 2 разных ключей.

Методы борьбы:

1. Цепочки (Chaining): В этом методе каждая ячейка хэш-таблицы содержит связанный список элементов с одинаковым хэшем. При коллизии элементы добавляются в список. Поиск элемента осуществляется путем прохода по списку.

2. Открытая адресация (Open addressing): В этом методе при коллизии происходит последовательное перебирание ячеек до поиска первой свободной ячейки. Существуют различные методы открытой адресации, такие как линейное открытая адресация, квадратичная открытая адресация и открытая адресация с двойным хэширование.

1. Укажите временную сложность операции поиска для *hash* таблицы для *лучшего* и *худшего* случая. Объясните приведенные оценки.

Лучший случай:

В лучшем случае, когда хэш-функция равномерно распределяет элементы по всей таблице без коллизий, время поиска будет константным O(1). Это происходит потому, что поиск элемента по хэш-значению требует только одну операцию доступа к ячейке таблицы.

Худший случай:

В худшем случае, когда все ключи имеют одно и то же хэш-значение или коллизии происходят очень часто, время поиска может стать линейным относительно количества элементов в таблице. В этом случае, если используется метод цепочек (chaining), время поиска будет O(n), где n - количество элементов в таблице. Это происходит потому, что для поиска элемента необходимо пройти по всем элементам в связанном списке, который соответствует хэш-значению.

6.Какова временная сложность алгоритма Рабина-Карпа?

Временная сложность алгоритма Рабина-Карпа составляет O(n + m), где n - длина строки, а m – длина образца.

7. Какова *пространственная* (*емкостная*) сложность алгоритма?

Пространственная сложность алгоритма Рабина-Карпа составляет O(m), где m - длина образца.

8. Что такое *коллизии* и какие методы позволяют их избежать при данной реализации алгоритма?

Коллизия – это совпадение значения хэш-функции для 2 разных ключей.

В данной реализации алгоритма при совпадении хешов шаблона и подстроки для исключения коллизий, их сравнивают посимвольно.

9. Что такое схема *Горнера* и для чего она используется, есть ли она в данной реализации?

Схема Горнера (или метод Горнера) - это метод вычисления значения многочлена, использующий схему последовательного умножения и сложения.

Схема Горнера позволяет быстро находить неполное частное и остаток от деления произвольного многочлена на двучлен.

Схема Горнера присутствует в данной реализации.

10. В каких задачах применяется алгоритм *Рабина-Карпа*?

Поиск образца в тексте: Алгоритм Рабина-Карпа может быть использован для быстрого поиска всех вхождений заданного образца (подстроки) в тексте.

2. Проверка наличия дубликатов: Алгоритм Рабина-Карпа может быть применен для определения наличия дубликатов в тексте, путем поиска совпадающих подстрок.

3. Проверка схожести строк: Алгоритм Рабина-Карпа может использоваться для определения схожести или сравнения строк путем сравнения их хэш-значений.

4. Алгоритм Рабина-Карпа также может быть использован в задачах обработки текста и поиска паттернов, таких как поиск повторяющихся фраз, анализ ДНК и других строковых последовательностей.

Вывод: изучили бинарное дерево поиска, хеш-таблицу.

Файл bstree.hpp

#pragma once

#include <stdint.h>

#include "hash.hpp"

struct bstree

{

int val;

std::string key;

bstree \*left = nullptr;

bstree \*right = nullptr;

bstree(int val, std::string key)

: val(val), key(key)

{}

};

bstree \*bstree\_create(std::string key, int value);

void bstree\_add(struct bstree \*tree, std::string key, int value);

struct bstree \*bstree\_lookup(struct bstree \*tree, std::string key);

struct bstree \*bstree\_min(struct bstree \*tree);

struct bstree \*bstree\_max(struct bstree \*tree);

Файл hash.hpp

#pragma once

#include <stdint.h>

#include <iostream>

static constexpr auto HASH\_MUL = 31;

static constexpr auto HASH\_SIZE = 100000;

int Hash(std::string key);

Файл hashtab.hpp

#pragma once

#include <iostream>

#include <chrono>

#include "hash.hpp"

static constexpr auto HASHTAB\_MAX\_SIZE = 100000;

struct ListNode

{

int value;

std::string key;

ListNode\* next = nullptr;

ListNode(std::string key, int val)

:key(key), value(val)

{

this->key = key;

}

};

void hashtab\_init(ListNode\*\* hashtab);

void hashtab\_add(ListNode\*\* hashtab, std::string key, int value);

ListNode\* hashtab\_lookup(ListNode\*\* hashtab, std::string key);

void hashtab\_delete(ListNode\*\* hashtab, char\*

Файл bstree.cpp

#include "bstree.hpp"

bstree \*bstree\_create(std::string key, int value)

{

return new bstree(value,key);

}

void bstree\_add(struct bstree \*tree, std::string key, int value)

{

bstree\* parent = nullptr;

if (tree == nullptr)

return;

while(tree != nullptr)

{

if (tree->key.compare(key) > 0)

{

parent = tree;

tree = tree->right;

}

else

{

parent = tree;

tree = tree->left;

}

}

if(parent->key.compare(key) > 0)

parent->right = bstree\_create(key,value);

else

parent->left = bstree\_create(key,value);

}

struct bstree \*bstree\_lookup(struct bstree \*tree, std::string key)

{

while(tree != nullptr)

{

if(tree->key.compare(key) > 0)

tree = tree->right;

else if(tree->key.compare(key) == 0)

return tree;

else

tree = tree->left;

}

return nullptr;

}

bstree\* bstree\_min(bstree\* tree)

{

while (tree->left != nullptr)

{

tree = tree->left;

}

return tree;

}

bstree\* bstree\_max(bstree\* tree)

{

while (tree->right != nullptr)

{

tree = tree->right;

}

return tree;

}

Файл hash.cpp

#include "hash.hpp"

int Hash(std::string key)

{

uint32\_t hash = 0;

for (auto& i : key)

{

int x = static\_cast<int>(i - 'a' - 1);

hash = (hash \* HASH\_MUL + x);

}

return hash % HASH\_SIZE;

}

Файл hastab.cpp

#include "hashtab.hpp"

void hashtab\_init(ListNode\*\* hashtab)

{

for (int i = 0; i < HASHTAB\_MAX\_SIZE; i++)

{

hashtab[i] = nullptr;

}

}

void hashtab\_add(ListNode\*\* hashtab, std::string key, int value)

{

int hash = Hash(key);

if (hashtab[hash] == nullptr)

{

hashtab[hash] = new ListNode(key,value);

}

else

{

ListNode\* node = hashtab[hash];

while (node->next != nullptr && node->key != key)

node = node->next;

if (node->key == key)

node->value = value;

else

node->next = new ListNode(key, value);

}

}

ListNode\* hashtab\_lookup(ListNode\*\* hashtab, std::string key)

{

int index = Hash(key);

ListNode \*node = hashtab[index];

while (node->next != nullptr && node->key != key)

{

node = node->next;

}

return node;

}

void hashtab\_delete(ListNode\*\* hashtab, char\* key)

{

int index = Hash(key);

ListNode \*node = hashtab[index];

ListNode \*prev = nullptr;

while (node != nullptr && node->key != key)

{

prev = node;

node = node->next;

}

if (node == nullptr)

return;

if (prev == nullptr)

hashtab[index] = node->next;

else

prev->next = node->next;

}

Файл main.cpp

#include <iostream>

#include <fstream>

#include <chrono>

#include <windows.h>

#include <random>

#include "bstree.hpp"

#include "hashtab.hpp"

ListNode\* hashtab[HASHTAB\_MAX\_SIZE];

static std::string letters = "QWERTYUIOPLKJHGFDSAZXCVBNMqwertyuioplkjhgfdsazxcvbnm";

std::string GenerateWord()

{

std::string res;

std::random\_device rd; // obtain a random number from hardware

std::mt19937 gen(rd()); // seed the generator

std::uniform\_int\_distribution<> distr(0, letters.size()-1);

int size = distr(gen);

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

{

res += letters[distr(gen)];

}

return res;

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

int count;

std::cin >> count;

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> distr(0, count);

std::string lookWord;

int randomWord = distr(gen);

//hashtab\_init(hashtab);

bstree\* root = bstree\_create(GenerateWord(), -1);

std::string out;

int a = 0;

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

{

out = GenerateWord();

calls++;

if (i == count-1)

{

a = i;

lookWord = out;

}

//hashtab\_add(hashtab, out, i);

bstree\_add(root, out, i);

out.clear();

}

std::chrono::steady\_clock::time\_point begin = std::chrono::steady\_clock::now();

//auto res = hashtab\_lookup(hashtab, lookWord);

auto res = bstree\_max(root);

std::chrono::steady\_clock::time\_point end = std::chrono::steady\_clock::now();

std::cout << "Время работы алгоритма : "

<< std::chrono::duration\_cast<std::chrono::seconds>(end - begin).count()

<< " секунд "

<< std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin).count()

<< " миллисекунд "

<< std::chrono::duration\_cast<std::chrono::microseconds>(end - begin).count()

<< " микросекунд "

<< std::chrono::duration\_cast<std::chrono::nanoseconds>(end - begin).count()

<< " наносекунд "

<< std::endl;

}

Задача TwoSum: Runtime 7 ms Beats 90.1%

Memory 11.2 MB Beats 17.33%

class Solution {

public:

vector<int> twoSum(vector<int>& nums, int target) {

std::unordered\_map<int,int> map;

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

{

auto res = map.find(target - nums[i]);

if(res != map.end())

{

return {res->second,i};

}

map.insert({nums[i],i});

}

return {0,0};

}

};