МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования

«БРЕСТСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Интеллектуальные информационные технологии»

Лабораторная работа №1

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

Тема: «***Бинарные деревья поиска и хеш-таблицы***»

**Выполнил:** Антонюк Никита

**Группа:** ПО-11

**Проверила:** Глущенко Т.А.

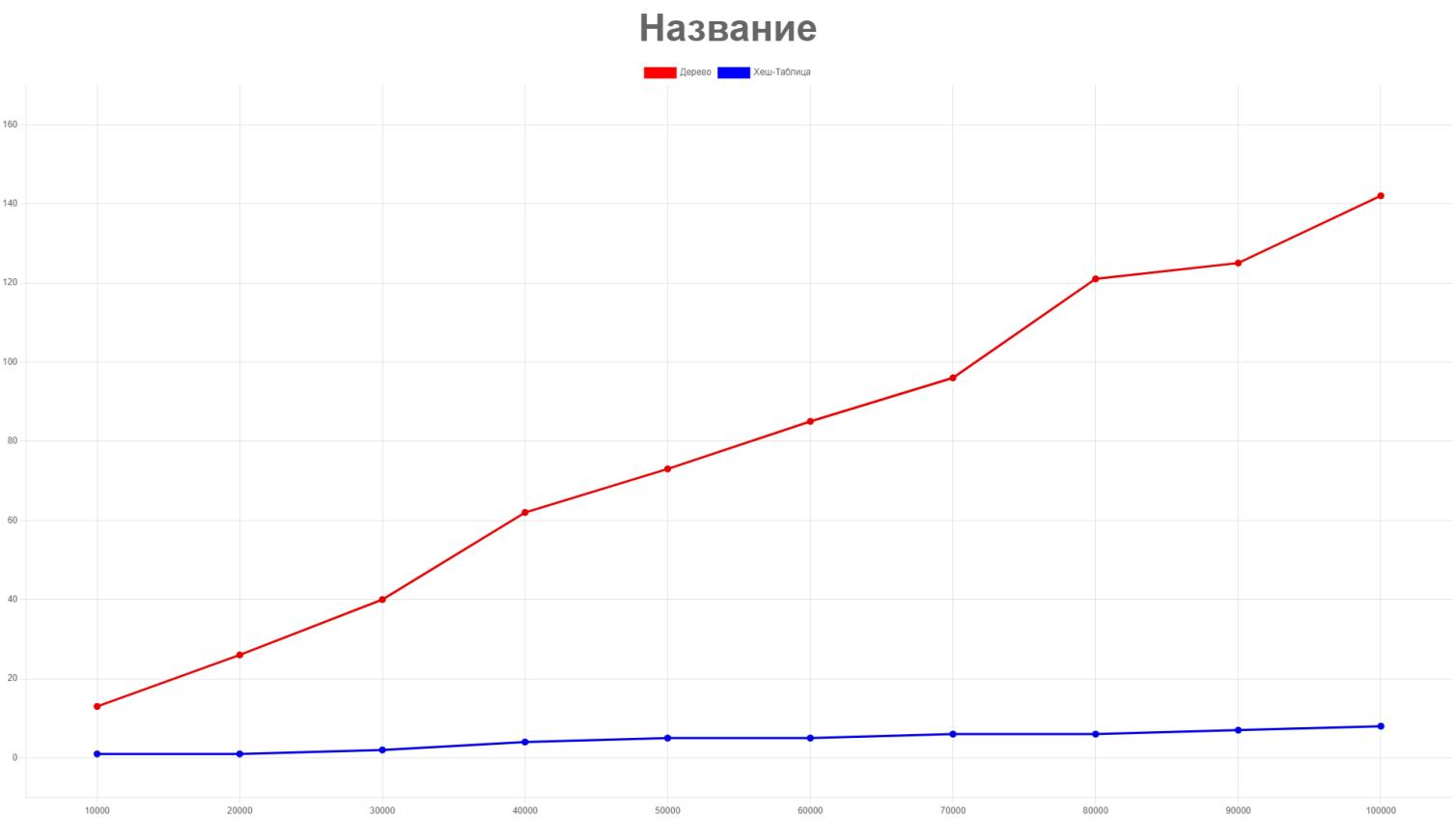
Брест 2023

**Постановка задачи.**

Требуется реализовать две библиотеки для работы с бинарным деревом поиска и хеш таблицей

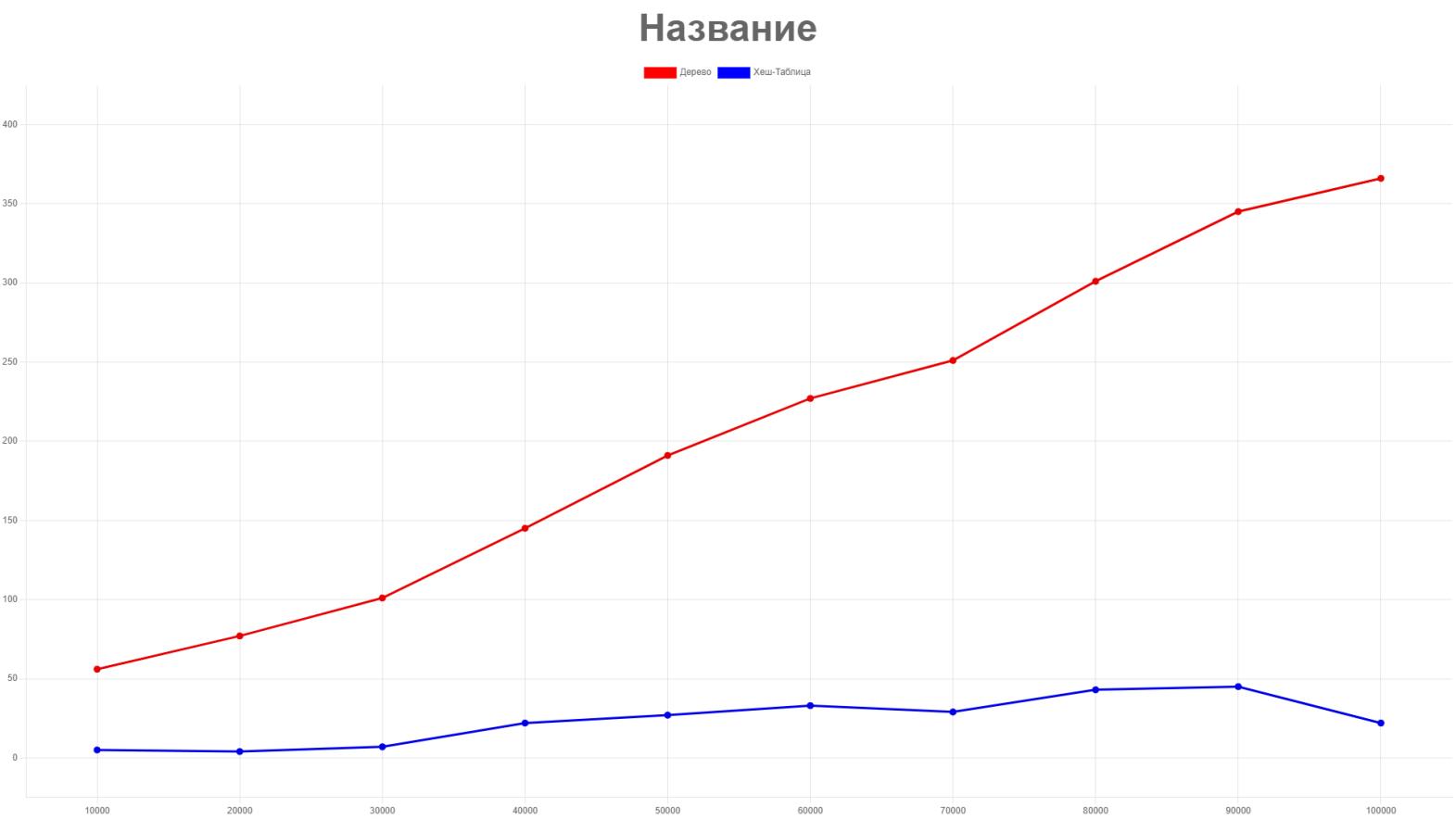
**Эксперимент 1**. Сравнение эффективности поиска элементов в бинарном дереве поиска и хеш-таблице в среднем случае

|  |  |  |  |
| --- | --- | --- | --- |
| # | Количество элементов в словаре | Время выполнения функции bstree\_lookup, с | Время выполнения функции hashtab\_lookup, с |
| 1 | 10 000 | 13 | 1 |
| 2 | 20 000 | 26 | 1 |
| 3 | 30 000 | 40 | 2 |
| 4 | 40 000 | 62 | 4 |
| 5 | 50 000 | 73 | 5 |
| 6 | 60 000 | 85 | 5 |
| 7 | 70 000 | 96 | 6 |
| 8 | 80 000 | 121 | 6 |
| 9 | 90 000 | 125 | 7 |
| 10 | 100 000 | 142 | 8 |



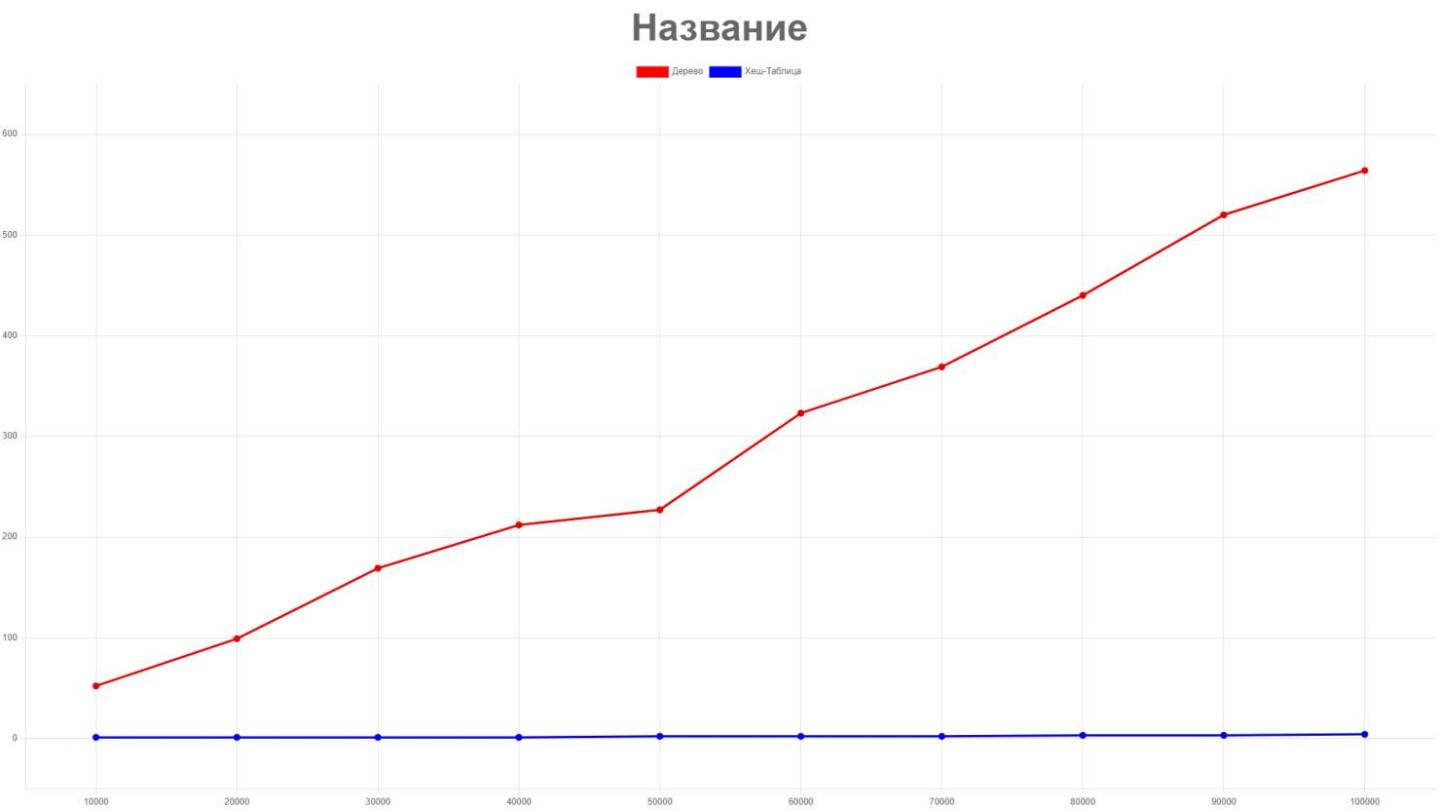
**Эксперимент 2**. Сравнение эффективности добавления элементов в бинарное дерево поиска и хеш таблицу

|  |  |  |  |
| --- | --- | --- | --- |
| # | Количество элементов в словаре | Время выполнения функции bstree\_add, с | Время выполнения функции hashtab\_add, с |
| 1 | 10 000 | 56 | 5 !!!!!!!!!!!!!! |
| 2 | 20 000 | 77 | 4 |
| 3 | 30 000 | 101 | 7 |
| 4 | 40 000 | 145 | 22 |
| 5 | 50 000 | 191 | 27 |
| 6 | 60 000 | 227 | 33 |
| 7 | 70 000 | 251 | 29 |
| 8 | 80 000 | 301 | 43 |
| 9 | 90 000 | 345 | 45 |
| 10 | 100 000 | 366 | 22 |



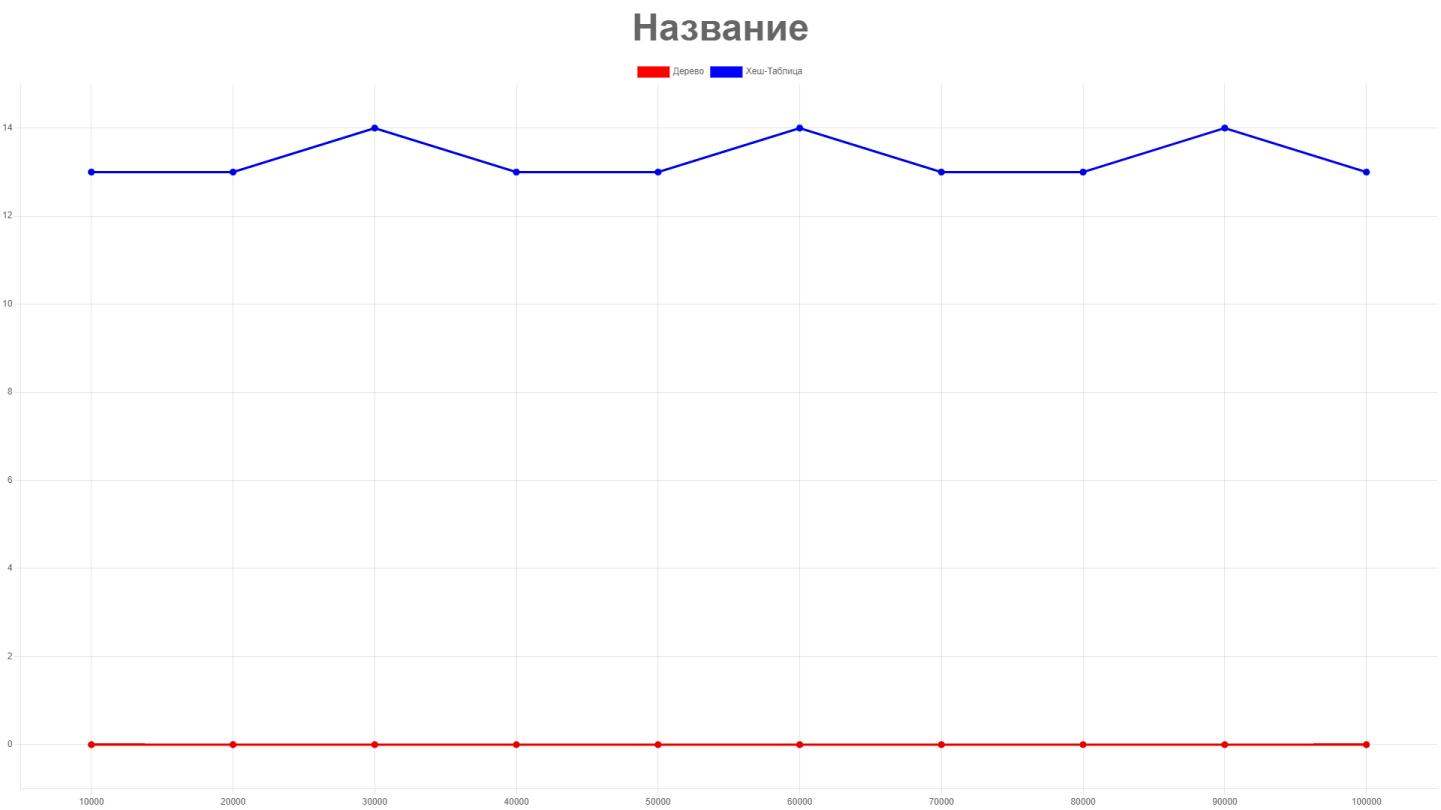
**Эксперимент 3**. Сравнение эффективности поиска элементов в бинарном дереве поиска и хеш таблице в худшем случае

|  |  |  |  |
| --- | --- | --- | --- |
| # | Количество элементов в словаре | Время выполнения функции bstree\_lookup, с | Время выполнения функции hashtab\_lookup, с |
| 1 | 10 000 | 52 | 1 |
| 2 | 20 000 | 99 | 1 |
| 3 | 30 000 | 169 | 1 |
| 4 | 40 000 | 212 | 1 |
| 5 | 50 000 | 227 | 2 |
| 6 | 60 000 | 323 | 2 |
| 7 | 70 000 | 369 | 2 |
| 8 | 80 000 | 440 | 3 |
| 9 | 90 000 | 520 | 3 |
| 10 | 100 000 | 564 | 4 |



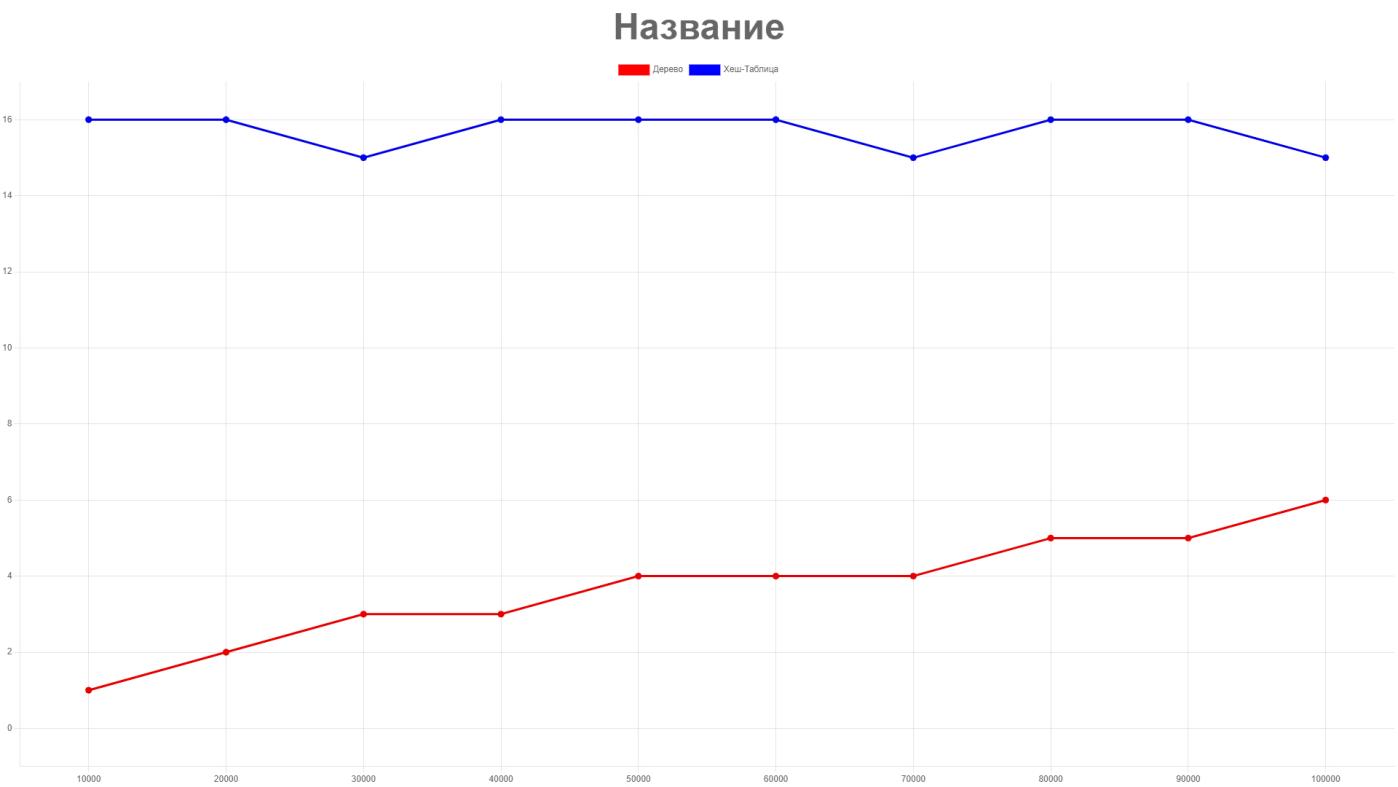
**Эксперимент 4**. Исследование эффективности поиска минимального элемента в бинарном дереве поиска в худшем и среднем случаях

|  |  |  |  |
| --- | --- | --- | --- |
| # | Количество элементов в словаре | Время выполнения функции bstree\_min, с | Время выполнения функции hashtab\_min, с |
| 1 | 10 000 | 0 | 13 |
| 2 | 20 000 | 0 | 13 |
| 3 | 30 000 | 0 | 14 |
| 4 | 40 000 | 0 | 13 |
| 5 | 50 000 | 0 | 13 |
| 6 | 60 000 | 0 | 14 |
| 7 | 70 000 | 0 | 13 |
| 8 | 80 000 | 0 | 13 |
| 9 | 90 000 | 0 | 14 |
| 10 | 100 000 | 0 | 13 |



**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, с | Время выполнения функции hashtab\_max, с |
| 1 | 10 000 | 1 | 16 |
| 2 | 20 000 | 2 | 16 |
| 3 | 30 000 | 3 | 15 |
| 4 | 40 000 | 3 | 16 |
| 5 | 50 000 | 4 | 16 |
| 6 | 60 000 | 4 | 16 |
| 7 | 70 000 | 4 | 15 |
| 8 | 80 000 | 5 | 16 |
| 9 | 90 000 | 5 | 16 |
| 10 | 100 000 | 6 | 15 |



**Вопросы к лабораторной работе:**

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

При поиске элемента в сбалансированном дереве поиска, каждое сравнение пропускает примерно половину оставшегося дерева, поэтому весь поиск занимает время, пропорциональное высоте дерева в худшем случае, которое составляет O(logN) времени в среднем случае по всем деревьям

1. **Укажите двойное неравенство для высоты бинарного дерева поиска?**

Левая часть неравенства - это минимальная высота бинарного дерева поиска с

N узлами. Это связано с тем, что бинарное дерево поиска с

N узлами может быть полностью заполнено до глубины ℎ, и в этом случае оно будет иметь 2h - 1 узлов. Поэтому, чтобы получить N узлов, нам нужно, чтобы 2h - 1 >= N, что эквивалентно левой части

Правая часть неравенства: N−1 - это максимальная высота бинарного дерева поиска с N узлами. Это связано с тем, что бинарное дерево поиска с N узлами может быть линейным, то есть все его узлы расположены в одной ветви. В этом случае высота дерева будет равна N−1.

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

*Существует множество алгоритмов построения хеш-функций, некоторые из них*:

* MD5 (Message Digest 5) - один из наиболее распространенных алгоритмов хеширования. Он принимает на вход сообщение произвольной длины и возвращает 128-битный хеш. (взломан)
* SHA (Secure Hash Algorithm) - семейство алгоритмов хеширования, разработанных Национальным институтом стандартов и технологий США (NIST). Наиболее распространенные версии - SHA-1, SHA-2 и SHA-3. SHA-1 в настоящее время считается устаревшим и не рекомендуется к использованию. SHA-2 широко используется
* MurmurHash - алгоритм, оптимизированный для быстрого вычисления хешей на больших объемах данных. Он используется в таких системах, как Hadoop и Cassandra.
* FNV (Fowler-Noll-Vo) Hash - алгоритм, основанный на простых операциях с целыми числами. Он быстро вычисляет хеши для больших объемов данных и используется в таких системах, как Redis и Memcached.

Каждый алгоритм имеет свои преимущества и недостатки, и выбор конкретного алгоритма зависит от требований к скорости, безопасности и размеру хеша

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

Коллизии - это ситуация, когда два или более ключей хеш-функции сопоставляются с одним и тем же индексом в хеш-таблице.

Методы борьбы:

* ***Метод цепочек(Chaining)***- каждый элемент хеш-таблицы содержит указатель на связанный список элементов, которые имеют одинаковый хеш-код. Этот метод является простым и эффективным, но может привести к увеличению размера хеш-таблицы.
* ***Открытая адресация(Open Addressing)*** - при этом методе все элементы хранятся непосредственно в хеш-таблице, а не в связанных списках. Если возникает коллизия, то производится поиск другого свободного слота в хеш-таблице для размещения элемента, создавшего проблему. Существуют различные методы открытой адресации, такие как линейное и квадратичное зондирование.
* ***Метод двойного хеширования(Double Hashing)*** - это метод, который использует две хеш-функции для разрешения коллизий. Если возникает коллизия, то производится поиск другого свободного слота в хеш-таблице с помощью второй хеш-функции. Этот метод может быть более эффективным, чем метод цепочек или открытая адресация.

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

В лучшем случае, когда все ключи хеш-функции хешируются в разные индексы хеш-таблицы, время поиска элемента будет константным и равным O(1).

В худшем случае, когда все ключи хеш-функции хешируются в один и тот же индекс хеш-таблицы, время поиска элемента будет линейным и равным O(N), где N - количество элементов в таблице. Это связано с тем, что при поиске элемента в хеш-таблице может потребоваться просмотреть все элементы в ячейке, что занимает O(N) времени.

1. **Ответить на вопросы к файлу *«Поиск подстроки в строке»***

**Вопросы по обработке алгоритма Рабина-Карпа:**

* **Какова временная сложность алгоритма?**

Алгоритм Рабина-Карпа имеет временную сложность O(n + m). При правильном выборе хеш-функции, алгоритм Рабина-Карпа может находить любую из заданных k строк одинаковой длины в среднем за время O(n) независимо от размера k

* **Какова пространственная сложность алгоритма?**

Пространственная сложность алгоритма Рабина-Карпа составляет O(m), где m - длина шаблона. При использовании оптимизаций, таких как схема Горнера для вычисления хеш-значения и хранение хеш-значений в виде циклического буфера, пространственная сложность может быть уменьшена

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

Схема Горнера - это алгоритм для деления многочлена на бином.

Схема Горнера используется, когда нужно отыскать целые корни уравнений высших степеней, имеющих целые коэффициенты. Она также может использоваться, если необходимо найти значение многочлена при заданном значении переменной.  
Да, схема Горнера используется в реализации алгоритма Рабина-Карпа. Она помогает избежать повторных вычислений хеш-функции для подстроки, что делает алгоритм более оптимальным. Схема Горнера используется для вычисления хеш-функции для каждой подстроки

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

1. Поиск подстроки в строке - использует хеширование для быстрого поиска подстроки в тексте
2. Поиск нескольких подстрок одновременно
3. Поиск целых чисел в больших массивах данных
4. Проверка схожести строк
5. Поиск паттернов (Повторяющиеся фразы, анализ ДНК)

Реализация на C#:

Файл main.cs:

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using HashTablee;

using BSTree;

using System.Diagnostics;

namespace LABA5\_AISD

{

class Class1

{

static void Main(string[] args)

{

/\*Tree tree = new Tree();

int iter = 100000;

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

{

tree.Add($"test{i}", i);

}

Console.WriteLine("ok");

Stopwatch stopwatch = Stopwatch.StartNew();

for (int i = 0; i < iter/2; i++)

{

tree.Lookup($"test{i}");

//Console.WriteLine(tree.Print($"test{i}"));

}

stopwatch.Stop();

Console.WriteLine(stopwatch.ElapsedMilliseconds);\*/

HashTable table = new HashTable();

Random rand = new Random();

string randomWord = "";

int iter = 20000;

string[] randWords = new string[iter];

string letters = "QWERTYUIOPLKJHGFDSAZXCVBNMqwertyuioplkjhgfdsazxcvbnm";

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

{

randomWord = "";

for(int j = 0; j < 5; j++)

{

int b = rand.Next(0, 52);

randomWord += letters[b];

}

randWords[i] = randomWord;

}

Stopwatch stopwatch = new Stopwatch();

stopwatch.Start();

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

{

table.Add(randWords[i], i);

}

stopwatch.Stop();

Console.WriteLine(stopwatch.ElapsedMilliseconds);

}

}

}

Файл bstree.cs:

using System;

namespace BSTree

{

public class Node

{

public string key;

public int value;

public Node left;

public Node right;

public Node(string key, int value)

{

this.key = key;

this.value = value;

this.left = null;

this.right = null;

}

}

public class Tree

{

private Node root;

public Tree()

{

this.root = null;

}

public void Add(string key, int value)

{

if (this.root == null)

{

this.root = new Node(key, value);

}

else

{

AddNode(this.root, key, value);

}

}

private void AddNode(Node node, string key, int value)

{

if (string.Compare(key, node.key) < 0)

{

if (node.left != null)

{

AddNode(node.left, key, value);

}

else

{

node.left = new Node(key, value);

}

}

else if (string.Compare(key, node.key) > 0)

{

if (node.right != null)

{

AddNode(node.right, key, value);

}

else

{

node.right = new Node(key, value);

}

}

else

{

node.value = value;

}

}

public Node Lookup(string key)

{

return LookupNode(this.root, key);

}

private Node LookupNode(Node node, string key)

{

if (node == null)

{

return null;

}

if (string.Compare(key, node.key) < 0)

{

return LookupNode(node.left, key);

}

else if (string.Compare(key, node.key) > 0)

{

return LookupNode(node.right, key);

}

else

{

return node;

}

}

public int Print(string key)

{

return PrintNode(this.root, key);

}

private int PrintNode(Node node, string key)

{

if (node == null)

{

return 0;

}

if (string.Compare(key, node.key) < 0)

{

return PrintNode(node.left, key);

}

else if (string.Compare(key, node.key) > 0)

{

return PrintNode(node.right, key);

}

else

{

return node.value;

}

}

public Node Min()

{

return MinNode(this.root);

}

private Node MinNode(Node node)

{

if (node == null)

{

return null;

}

while (node.left != null)

{

node = node.left;

}

return node;

}

public Node Max()

{

return MaxNode(this.root);

}

private Node MaxNode(Node node)

{

if (node == null)

{

return null;

}

while (node.right != null)

{

node = node.right;

}

return node;

}

}

}

Файл hashtab.cs:

using System;

namespace HashTablee

{

public class ListNode

{

public string key;

public int value;

public ListNode next;

public ListNode(string key, int value)

{

this.key = key;

this.value = value;

this.next = null;

}

}

public class HashTable

{

private ListNode[] table;

public HashTable()

{

this.table = new ListNode[1000000];

}

private int Hash(string key)

{

int h = 0;

for (int i = 0; i < key.Length; i++)

{

h = 31 \* h + key[i];

}

return h % 1000000;

}

public void Add(string key, int value)

{

int index = Hash(key);

if (this.table[index] == null)

{

this.table[index] = new ListNode(key, value);

}

else

{

ListNode node = this.table[index];

while (node.next != null && node.key != key)

{

node = node.next;

}

if (node.key == key)

{

node.value = value;

}

else

{

node.next = new ListNode(key, value);

}

}

}

public ListNode Lookup(string key)

{

int index = Hash(key);

ListNode node = this.table[index];

while (node != null &&node.key != key)

{

node = node.next;

}

return node;

}

public int Print(string key)

{

int index = Hash(key);

ListNode node = this.table[index];

while (node != null && node.key != key)

{

node = node.next;

}

return node.value;

}

public void Delete(string key)

{

int index = Hash(key);

ListNode node = this.table[index];

ListNode prev = null;

while (node != null && node.key != key)

{

prev = node;

node = node.next;

}

if (node == null)

{

return;

}

if (prev == null)

{

this.table[index] = node.next;

}

else

{

prev.next = node.next;

}

}

}

}

**Задача 28**. *Find the Index of the First Occurrence in a String*

Реализация на С++:

public:

int strStr(string haystack, string needle) {

int m=haystack.size(),n=needle.size(),p=0;

while(p+n-1<m){

if(haystack.substr(p,n)==needle)

return p;

p++;

}

return -1;

}

};

