ФЕДЕРАЛЬНОЕ АГЕНТСТВО СВЯЗИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

**ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**

к курсовому проекту по дисциплине

«Структуры и алгоритмы обработки данных»

на тему

СПИСОК С ПРОСПУСКАМИ

|  |  |
| --- | --- |
| Выполнил студент | Любицкий М. Е. |
|  | Ф.И.О. |

|  |  |
| --- | --- |
| Группы | ИС242 |
|  |  |

|  |  |  |
| --- | --- | --- |
| Работу приняла |  | ассистент Кафедры ВС Насонова А.О. |
|  | подпись |  |

|  |  |  |  |
| --- | --- | --- | --- |
| Защищена |  | Оценка |  |
|  |  |  |  |

Новосибирск – 2023

Оглавление

[ВВЕДЕНИЕ 3](#_Toc153502477)

[ОПИСАНИЕ СТРУКТУРЫ 4](#_Toc153502478)

[ОПИСАНИЕ ОСНОВНЫХ ОПЕРАЦИЙ 5](#_Toc153502479)

[1. Вставка и удаление ключа 5](#_Toc153502480)

[2. Поиск ключа: 7](#_Toc153502481)

[АНАЛИЗ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ 8](#_Toc153502482)

[1. Поиск ключа 9](#_Toc153502483)

[2. Добавление ключей 10](#_Toc153502484)

[3. Удаление ключа 11](#_Toc153502485)

[ЗАКЛЮЧЕНИЕ 12](#_Toc153502486)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 13](#_Toc153502487)

[ПРИЛОЖЕНИЕ 14](#_Toc153502488)

# ВВЕДЕНИЕ

На сегодняшний день разработано и исследовано множество различных структур данных, каждая из которых обладает своими особенностями и может эффективно применяться в подходящих задачах.

В рамках данной курсовой работы мною была изучена и реализована такая структура данных как Skip list.

Изобрел список с пропусками американский ученый Уильям Уортингтон Пью-младший

Исследуемая структура может обладать множеством различных полезных применений:

* QMap (до Qt 4) класс шаблона Qt , который предоставляет словарь.
* SkiMap использует списки пропуска как базовая структура данных для создания более сложной трехмерной разреженной сетки для систем отображения роботов.
* Сервер Cyrus IMAP.
* Lucene использует списки пропуска для поиска списков сообщений с дельта-кодированием в логарифмическом времени.

# ОПИСАНИЕ СТРУКТУРЫ

Skip list -  вероятностная структура данных, основанная на нескольких параллельных отсортированных связных списках с эффективностью, сравнимой с двоичным деревом.

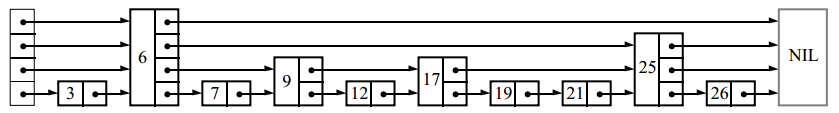


Рисунок 1. Пример Skip list.

В основе списка с пропусками лежит расширение отсортированного связного списка дополнительными связями, добавленными в случайных путях с геометрическим/негативным биномиальным распределением, таким образом, чтобы поиск по списку мог быстро пропускать части этого списка.

Несмотря на то, что у списков с пропусками плохая производительность в худшем случае, не существует такой последовательности операций, при которой бы это происходило постоянно (примерно, как в алгоритме быстрой сортировки со случайным выбором опорного элемента). Очень маловероятно, что эта структура данных значительно разбалансируется (например, для словаря размером более 250 элементов вероятность того, что поиск займёт в три раза больше ожидаемого времени, меньше одной миллионной).

Свойства списка с пропусками:

* Каждый элемент в списке с пропусками представлен узлом
* У каждого узла есть высота или уровень, который соответствует количеству указателей на следующие уровни
* i-ый указатель узла указывает на следующий узел, находящийся на уровне i или выше
* При вставке нового элемента в список, узел вставляется на уровень со случайным номером
* Уровни со случайными номерами генерируются по шаблону. Например: 50% - уровень 1, 25% - уровень 2, 12.5% - уровень 3 и т.д.

# ОПИСАНИЕ ОСНОВНЫХ ОПЕРАЦИЙ

## Вставка и удаление ключа

**Insert**(list, searchKey, newValue)  
    **local** update[1..MaxLevel]  
    x := list→header  
    **for** i := list→level **downto** 1 **do**  
        **while** x→forward[i]→key < searchKey **do**  
            x := x→forward[i]  
        *# x→key < searchKey ≤ x→forward[i]→key*  
        update[i] := x  
    x := x→forward[1]  
    **if** x→key = searchKey **then** x→value := newValue  
    **else**  
        lvl := randomLevel()  
        **if** lvl > list→level **then**  
            **for** i := list→level + 1 **to** lvl **do**  
                update[i] := list→header  
            list→level := lvl  
        x := makeNode(lvl, searchKey, value)  
        **for** i := 1 **to** level **do**  
            x→forward[i] := update[i]→forward[i]  
            update[i]→forward[i] := x

**Delete**(list, searchKey)  
    **local** update[1..MaxLevel]  
    x := list→header  
        **for** i := list→level **downto** 1 **do**  
            **while** x→forward[i]→key < searchKey **do**  
                x := x→forward[i]  
        update[i] := x  
    x := x→forward[1]  
    **if** x→key = searchKey **then**  
        **for** i := 1 **to** list→level **do**  
            **if** update[i]→forward[i] ≠ x **then** **break**  
            update[i]→forward[i] := x→forward[i]  
        free(x)  
        **while** list→level > 1 **and** list→header→forward[list→level] = NIL **do**  
            list→level := list→level – 1

Для вставки или удаления узла применяем алгоритм поиска для нахождения всех элементов перед вставляемым (или удаляемым), затем обновляем соответствующие указатели

Начиная с указателя наивысшего уровня, двигаемся вперед по указателям до тех пор, пока они ссылаются на элемент, не превосходящий искомый. Затем спускаемся на один уровень ниже и снова двигаемся по тому же правилу. Если мы достигли уровня 1 и не можем идти дальше, то мы находимся как раз перед элементом, который ищем (если он там есть).

На рисунке 2 можно наглядно увидеть работу алгоритма добавления ключей в дерево на примере добавления ключа 17.

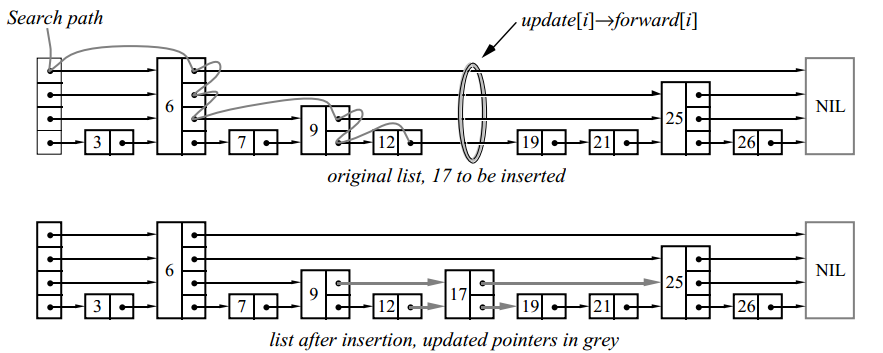


Рисунок 2. Вставка и удаление ключа.

## 2. Поиск ключа:

**Search**(list, searchKey)  
    x := list→header  
    *x→key < searchKey*  
    **for** i := list→level **downto** 1 **do**  
        **while** x→forward[i]→key < searchKey **do**  
            x := x→forward[i]  
     *x→key < searchKey ≤ x→forward[1]→key*  
    x := x→forward[1]  
    **if** x→key = searchKey **then** **return** x→value  
    **else** **return** failure

Начиная с указателя наивысшего уровня, двигаемся вперед по указателям до тех пор, пока они ссылаются на элемент, не превосходящий искомый. Затем спускаемся на один уровень ниже и снова двигаемся по тому же правилу. Если мы достигли уровня 1 и не можем идти дальше, то мы находимся как раз перед элементом, который ищем (если он там есть).

# АНАЛИЗ ЭФФЕКТИВНОСТИ АЛГОРИТМОВ

В данной главе будет проводится асимптотический анализ описанных выше алгоритмов вместе с их экспериментальным исследованием.

В операциях поиска, вставки и удаления больше всего времени уходит на поиск подходящего элемента. Для вставки и удаления дополнительно нужно время, пропорциональное уровню вставляемого или удаляемого узла. Время поиска элемента пропорционально количеству пройденных в процессе поиска узлов, которое, в свою очередь, зависит от распределения их уровней.

Так как Skip list является вероятностной структурой данных, очень большую роль играет максимальное количество уровней и вероятность с которой увеличивается уровень элемента.

Далее будут описываться эксперименты при которых добавляется по 100 000 ключей в списки из 8 уровней и вероятностью p = 0.5, 0.35, 0.25. Измерение времени будет производиться 10 раз и высчитываться среднее арифметическое.

## Поиск ключа

Длительность выполнения операции поиска напрямую зависит от того, сколько узлов понадобится пройти от начала дерева до самого последнего узла, который будем. Таким образом, в худшем случае, сложность операции поиска составляет O(N), где N – количество элементов в списке. Но такой случай маловероятен, поэтому худший случай рассматривать не будем. В среднем случае, сложность будет где N – количество элементов в списке.

Экспериментальное исследование времени поиска изображено на рисунке 3.



Рисунок 3. Время поиска ключа.

## Добавление ключей

При выполнении каждой операции добавления каждый раз происходит поиск того места, куда добавим новый элемент. Этот поиск практически ничем не отличается от самой операции поиска, поэтому время в среднем случае, будет где N – количество элементов в списке.

Экспериментальное исследование времени добавления изображено на рисунке 4.

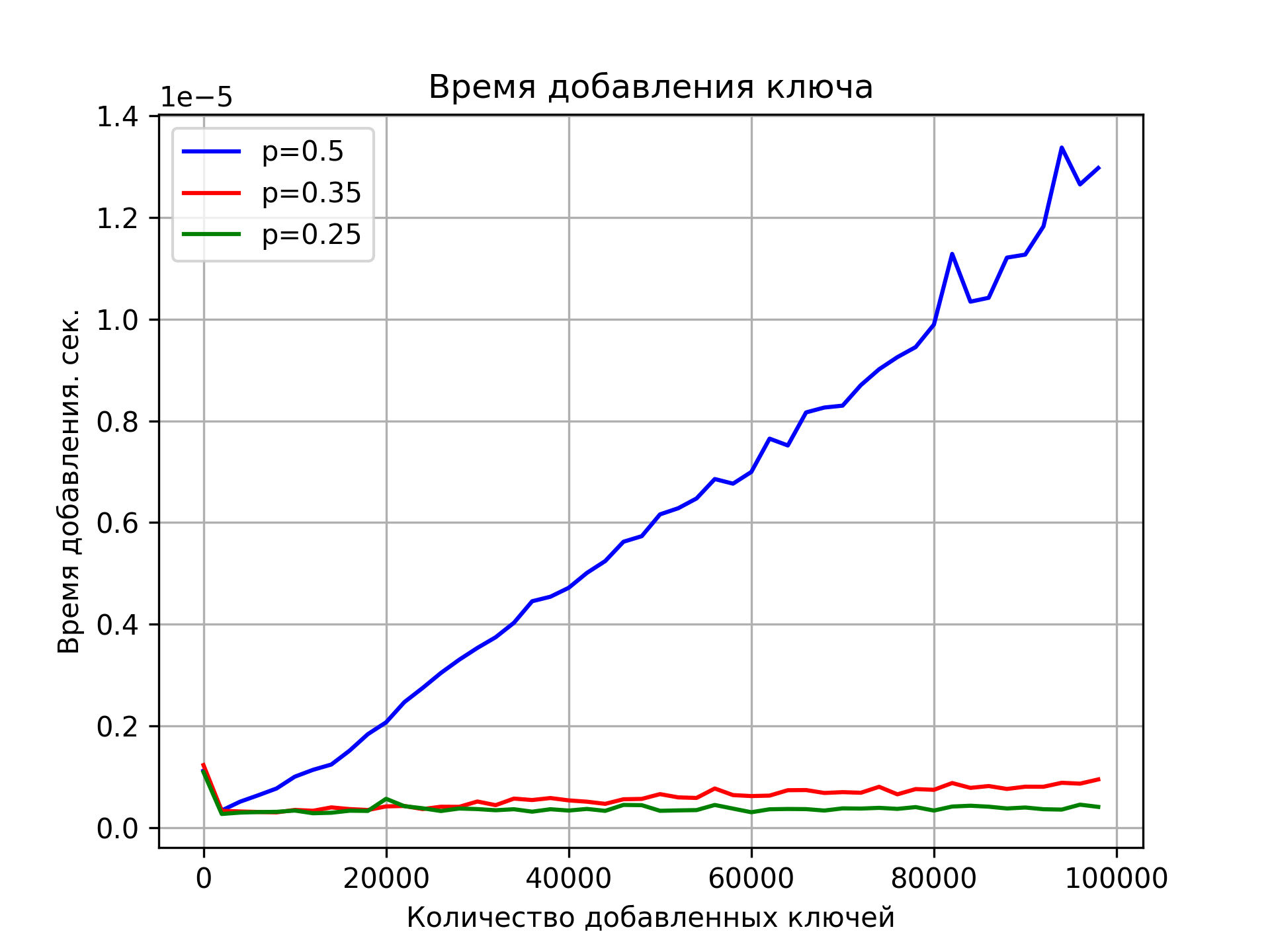


Рисунок 4 Время добавление ключа.

## Удаление ключа

Операция удаления является обратной к операции добавления. При выполнении этой операции также производится поиск, который практически ничем не отличается от обычного поиска. В среднем случае список формируется равномерно, и тогда время удаления будет где N – количество элементов в списке. Графики сложности алгоритма проиллюстрированы на рисунке 5.

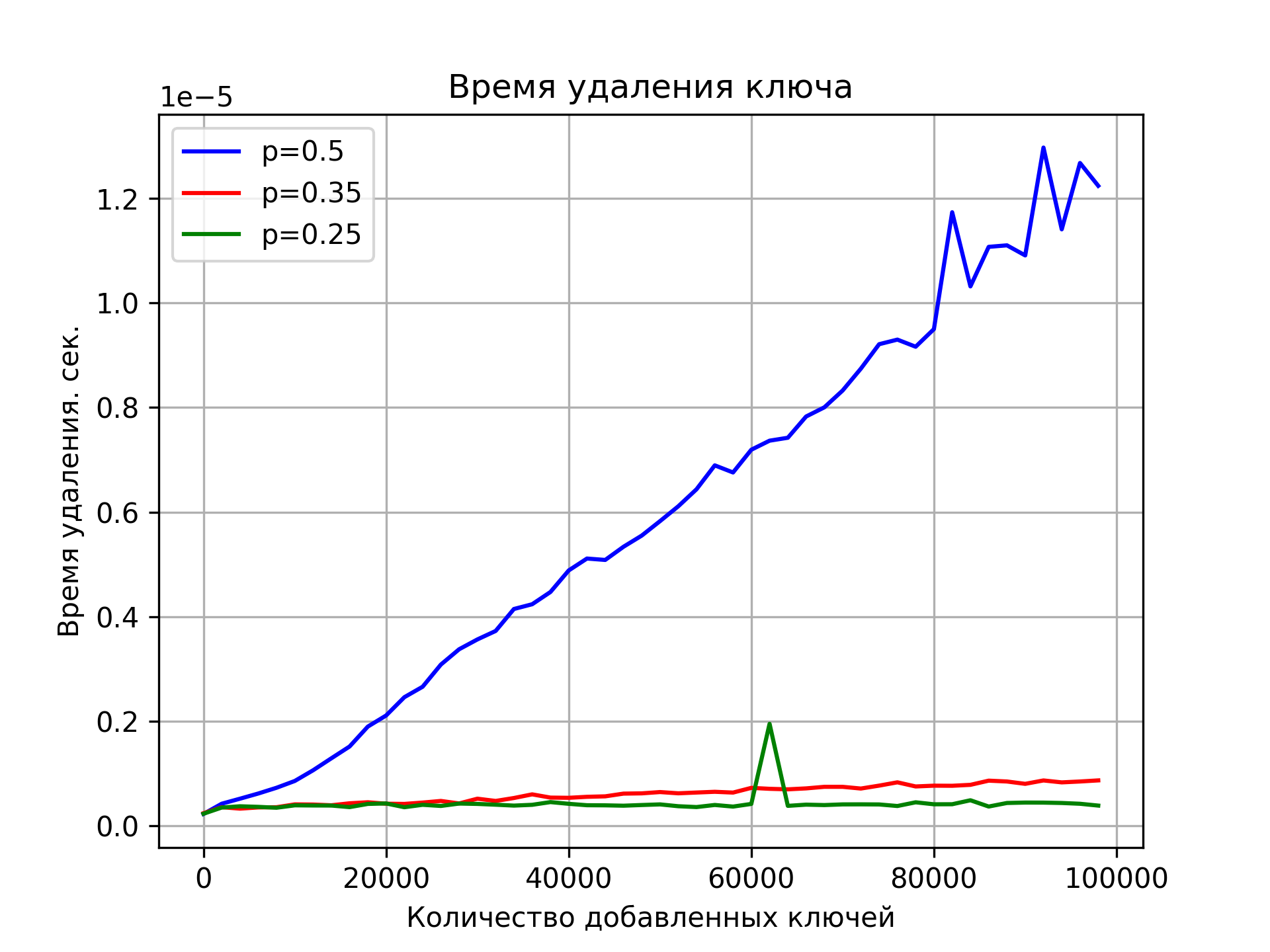


Рисунок 5. Время удаление ключа.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения данной курсовой работы была реализована и детально проанализирована такая структура данных как список с пропусками, которая способна выполнять операции поиска, вставки ключей, и удаления ключей за время log(N) в среднем случае. Данная структура данных не эффективно использует память, так как приходится хранить массив ссылок в каждом элементе. Несмотря на то, что эта структура данных может иметь множество различных применений, большого распространения она не получила.

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

- Wikipedia. Skip list. URL:

<https://en.wikipedia.org/wiki/Skip_list>

# ПРИЛОЖЕНИЕ

Исходный код программы

main.cpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175 | #include <iostream>  #include <iomanip>  #include <ctime>  #include <chrono>  #include <fstream>  #include <vector>  #include "Skiplist.hpp"  **int** **main**()  {  srand(time(NULL));    std::ofstream file("output.txt", std::ios\_base::out);  std::ofstream file1("output1.txt", std::ios\_base::out);  std::ofstream file2("output2.txt", std::ios\_base::out);  file << std::fixed << std::setprecision(**9**);  file1 << std::fixed << std::setprecision(**9**);  file2 << std::fixed << std::setprecision(**9**);  std::cout << std::fixed << std::setprecision(**9**);  file << "n," << "t**\n**";  file1 << "n," << "t**\n**";  file2 << "n," << "t**\n**";  **int** n = **100000**;  **int** sravn = **2000**;  **int** max = n / sravn;  std::vector<**double**> v(max, **0**);  **int** k = **0**;  Skiplist \*l = **new** Skiplist;  **for** (**int** i = **0**; i < **1000**; ++i)  {  l->insert(i, "");  }  **delete** l;  **auto** start = std::chrono::steady\_clock::now();  **for** (**int** j = **0**; j < **10**; ++j)  {  Skiplist list(**8**, **0.5**);  k = **0**;  **for** (**int** i = **0**; i < n; ++i)  {  **if** (i % sravn == **0**)  {  list.insert(i, "");  **auto** s = std::chrono::steady\_clock::now();  list.search(i);  **auto** e = std::chrono::steady\_clock::now();  **auto** sec = std::chrono::duration\_cast<std::chrono::nanoseconds>(e - s);  v[k] += sec.count() / **1.0e+9**;  ++k;  // file << i + 1 << ", " << sec.count() / 1.0e+9 << std::endl;  }  **else**  {  list.insert(i, "");  }  }  }  **for** (**int** i = **0**; i < v.size(); ++i)  {  v[i] /= **10**;  file << i \* sravn << ", " << v[i] << '\n';  }  v.clear();  v.resize(max, **0**);  **for** (**int** j = **0**; j < **10**; ++j)  {  Skiplist list25(**8**, **0.25**);  k = **0**;  **for** (**int** i = **0**; i < n; ++i)  {  **if** (i % sravn == **0**)  {  list25.insert(i, "");  **auto** s = std::chrono::steady\_clock::now();  list25.search(i);  **auto** e = std::chrono::steady\_clock::now();  **auto** sec = std::chrono::duration\_cast<std::chrono::nanoseconds>(e - s);  v[k] += sec.count() / **1.0e+9**;  ++k;  // file1 << i + 1 << ", " << sec.count() / 1.0e+9 << std::endl;  }  **else**  {  list25.insert(i, "");  }  }  }  **for** (**int** i = **0**; i < v.size(); ++i)  {  v[i] /= **10**;  file1 << i \* sravn << ", " << v[i] << '\n';  }  v.clear();  v.resize(max, **0**);  **for** (**int** j = **0**; j < **10**; ++j)  {  Skiplist list33(**8**, **0.35**);  k = **0**;  **for** (**int** i = **0**; i < n; ++i)  {  **if** (i % sravn == **0**)  {  list33.insert(i, "");  **auto** s = std::chrono::steady\_clock::now();  list33.search(i);  **auto** e = std::chrono::steady\_clock::now();  **auto** sec = std::chrono::duration\_cast<std::chrono::nanoseconds>(e - s);  v[k] += sec.count() / **1.0e+9**;  ++k;  // file1 << i + 1 << ", " << sec.count() / 1.0e+9 << std::endl;  }  **else**  {  list33.insert(i, "");  }  }  }  **for** (**int** i = **0**; i < v.size(); ++i)  {  v[i] /= **10**;  file2 << i \* sravn << ", " << v[i] << '\n';  }  **auto** end = std::chrono::steady\_clock::now();  **auto** elapsedtime = std::chrono::duration\_cast<std::chrono::nanoseconds>(end - start);  file.close();  file1.close();  file2.close();  // list.display();  std::cout << "elapsed time: " << elapsedtime.count() / **1.0e+9** << " seconds**\n**";  // std::cout << list.search(20) << '\n' << list.search(10) << '\n';  // list.display();  **return** **0**;  } |

Skiplist.hpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99  100  101  102  103  104  105  106  107  108  109  110  111  112  113  114  115  116  117  118  119  120  121  122  123  124  125  126  127  128  129  130  131  132  133  134  135  136  137  138  139  140  141  142  143  144  145  146  147  148  149  150  151  152  153  154  155  156  157  158  159  160  161  162  163  164  165  166  167  168  169  170  171  172  173  174  175  176  177  178  179  180  181  182  183  184  185  186  187  188  189  190  191  192  193  194  195 | #pragma once  #include <iostream>  #include "Node.hpp"  **class** **Skiplist**  {  **private:**  Node \*header;  Node \*Nil;  **unsigned** maxLevel;  **float** probability;  **public:**  Skiplist();  Skiplist(**unsigned** maxLevel, **float** probability);  **void** **insert**(**const** **int** key, **const** std::string &value);  **void** **remove**(**const** **int** key);  **bool** **search**(**const** **int** key);  **void** **display**();  ~Skiplist();  };  Skiplist::Skiplist()  : maxLevel(**16**),  probability(**0.5**),  header(**new** Node(**16**)),  Nil(**new** Node(**16**))  {  **for** (**int** i = **0**; i < maxLevel; i++)  {  **this**->header->next[i] = **this**->Nil;  }  }  Skiplist::Skiplist(**unsigned** maxLevel, **float** probability)  : maxLevel(maxLevel),  probability(probability),  header(**new** Node(maxLevel)),  Nil(**new** Node(maxLevel))  {  **if** (**this**->probability >= **1**)  {  **this**->probability = **0.5**;  }  **for** (**int** i = **0**; i < maxLevel; i++)  {  **this**->header->next[i] = **this**->Nil;  }  }  Skiplist::~Skiplist()  {  Node\* node = **this**->header->next[**0**];  **while** (node != **this**->Nil)  {  Node\* tmp = node->next[**0**];  **delete** node;  node = tmp;  }    **delete** **this**->header;  **delete** **this**->Nil;  }  **void** Skiplist::insert(**const** **int** key, **const** std::string &value)  {  Node \*\*update = **new** Node \*[maxLevel];  Node \*node = **this**->header;  **for** (**int** i = node->level - **1**; i >= **0**; --i)  {  **while** (node->next[i] != **this**->Nil && node->next[i]->key < key)  {  node = node->next[i];  }  update[i] = node;  }  node = node->next[**0**];  **if** (node == **this**->Nil || node->key != key)  {  **int** newlvl = **0**;  **while** (newlvl < **this**->maxLevel - **1** && (**float**(rand()) / RAND\_MAX) < **this**->probability)  {  ++newlvl;  }  Node \*newnode = **new** Node(key, value, newlvl + **1**);  **for** (**int** i = **0**; i < newlvl + **1**; ++i)  {  newnode->next[i] = update[i]->next[i];  update[i]->next[i] = newnode;  }  // std::cout << key << ": inserted successfully\n";  }  **else**  {  // std::cout << key << ": already exist\n";  }  **delete** [] update;  }  **void** Skiplist::remove(**const** **int** key)  {  Node \*\*update = **new** Node \*[**this**->maxLevel];  Node \*node = **this**->header;  **for** (**int** i = node->level - **1**; i >= **0**; --i)  {  **while** (node->next[i] != **this**->Nil && node->next[i]->key < key)  {  node = node->next[i];  }  update[i] = node;  }  node = node->next[**0**];  **if** (node != **this**->Nil && node->key == key)  {  **for** (**int** i = **0**; i < **this**->maxLevel; ++i)  {  **if** (update[i]->next[i] != node)  {  **break**;  }  **else**  {  update[i]->next[i] = node->next[i];  }  }  **delete** node;  // std::cout << key << ": deleted successfully\n";  }  **else**  {  // std::cout << key << ": not found\n";  }  **delete** [] update;  }  **bool** Skiplist::search(**const** **int** key)  {  Node \*node = **this**->header;  **for** (**int** i = node->level - **1**; i >= **0**; --i)  {  **while** (node->next[i] != **this**->Nil && node->next[i]->key < key)  {  node = node->next[i];  }  }  node = node->next[**0**];  **if** (node != **this**->Nil && node->key == key)  {  **return** true;  }  **else**  {  **return** false;  }  }  **void** Skiplist::display()  {  std::cout << "Skiplist**\n**";  **for** (**int** i = **this**->maxLevel - **1**; i >= **0**; --i)  {  Node \*node = **this**->header->next[i];  std::cout << "Level " << i + **1** << ": ";  **while** (node != **this**->Nil)  {  std::cout << node->key << ' ';  node = node->next[i];  }  std::cout << '\n';  }  } |

Node.hpp

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43 | #pragma once  #include <string>  **struct** Node  {  **int** key;  **unsigned** level;  std::string value;  Node \*\*next;  Node(**const** **int** &key, **const** std::string &value, **int** level);  Node(**int** level);  ~Node();  };  Node::Node(**const** **int** &key, **const** std::string &value, **int** level)  : key(key),  value(value),  level(level),  next(**new** Node \*[level])  {  **for** (**int** i = **0**; i < level; ++i)  {  **this**->next[i] = nullptr;  }  }  Node::Node(**int** level)  : key(**0**),  value(""),  level(level),  next(**new** Node \*[level])  {  **for** (**int** i = **0**; i < level; ++i)  {  **this**->next[i] = nullptr;  }  }  Node::~Node()  {  **delete** [] **this**->next;  } |