

МИНОБРНАУКИ РОССИИ

*Федеральное государственное бюджетное образовательное учреждение высшего образования*

***«МИРЭА – Российский технологический университет»***

Отчет

Практическая работа №9

Дисциплина Структуры и алгоритмы обработки данных

Тема. Алгоритмы поиска в таблице (массиве)

|  |  |
| --- | --- |
| Выполнил студент | Смольников А.Б. |
|  | Фамилия И.О. |
| Группа | ИКБО-13-21 |
|  | Номер группы |

**Москва 2022**

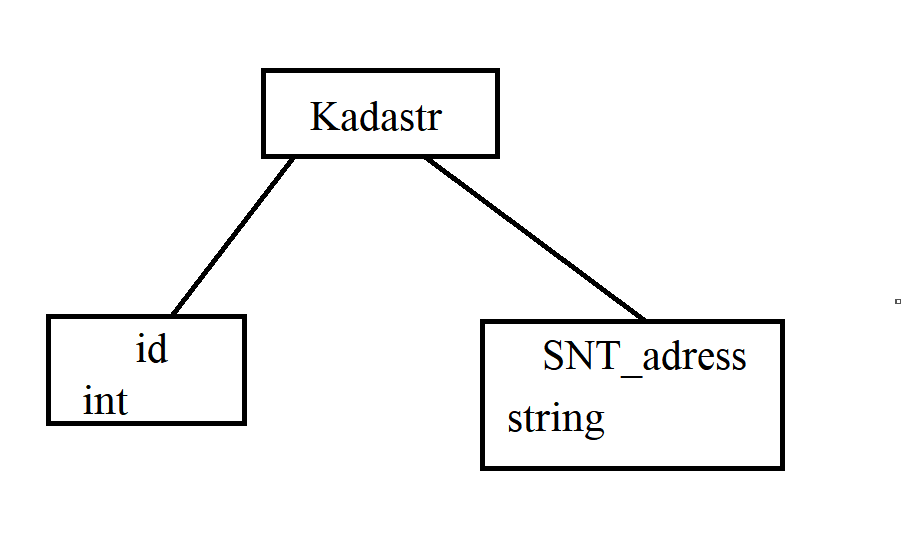
**Задание 1**

**Вариант №20**

**Разработать программу поиска записи по ключу в таблице записей с применение двух алгоритмов линейного поиска.**

1. Таблица содержит записи, структура которых определена вариантом. *(шестизначное число – номер участка, строка – название участка)* Ключи уникальны в пределах таблицы.
2. Разработать функцию линейного поиска (метод грубой силы).
3. Разработать функцию поиска с барьером.
4. Провести практическую оценку времени выполнения алгоритмов на таблицах объемом 100, 1000, 10 000 записей.
5. Составить таблицу с указанием: времени выполнения алгоритма, его фактическую и теоретическую вычислительную сложность.
6. Сделать выводы об эффективности алгоритмов.

**Структура записи**



Память, занимаемая элементом структуры – 28+4 = 32 байта

**Разработка алгоритма и его реализация**

Необходимо разработать два алгоритма линейного поиска в таблице по ключу.

Алгоритм: Для начала создадим таблицу из ячеек записи(ячейка содержит целое число – ключ и строку – название СНТ):

1. struct Kadastr{
2. int id;
3. string SNT\_adress;
4. };
5. struct KadastrTable{
6. Kadastr \*kadastr;
7. int size;
8. int append(Kadastr note);
9. void print();
11. Kadastr LinearSearchBarrier(int id);
12. Kadastr BruteForce(int id);
13. ~KadastrTable();
14. };

Метод грубой силы основан на грубом переборе всех значений ячеек пока не будет найдена нужная запись.

Рассмотрим код функции поиска:

Предусловие – искомый ключ id

Постусловие – искомая запись

1. Kadastr KadastrTable::BruteForce(int id){
2. for(int i = 0; i < size; i++){
3. if(kadastr[i].id == id){
4. return kadastr[i];
5. }
6. }
7. return Kadastr{-1, "NULL"};
8. }

Метод линейного поиска с барьером основан на сокращении операций в цикле за счет установки барьера

Рассмотрим код функции:

Предусловие – искомый ключ id

Постусловие – искомая запись

1. Kadastr KadastrTable::LinearSearchBarrier(int id){
2. if(size == 0) {
3. return Kadastr{0, "NO NAME"};
4. }
5. Kadastr last = kadastr[size-1];
6. kadastr[size-1].id = id;
7. int i;
8. for(i=0; kadastr[i].id!= id;++i){}
9. kadastr[size-1] = last;
10. if(i != size-1 || id == last.id){
11. return kadastr[i];
12. }
13. return Kadastr{0, "NO NAME"};
14. }

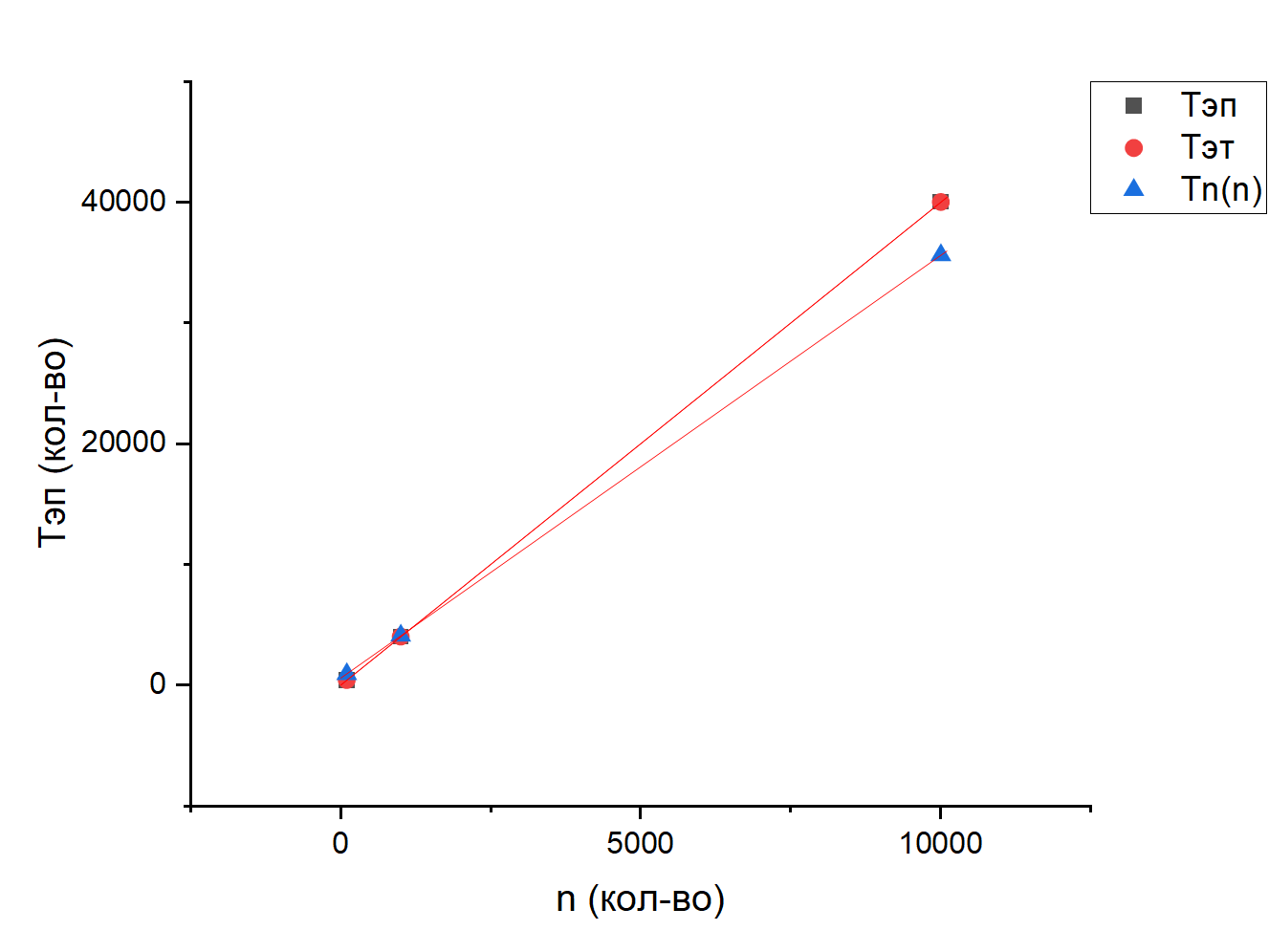
Оба этих поиска являются линейными, а значит их асимптотическая сложность:

Определим зависимость времени от количества входных данных эмпирически. В действительности эта цифра зависит от характеристик ВМ, типа операций, оптимизации самого компилятора.

Рассмотрим эмпирически количество выполняемых операций. В соответствии с кодом задачи, получим:

***Таблица 1. Сводная таблица результатов Brute force***

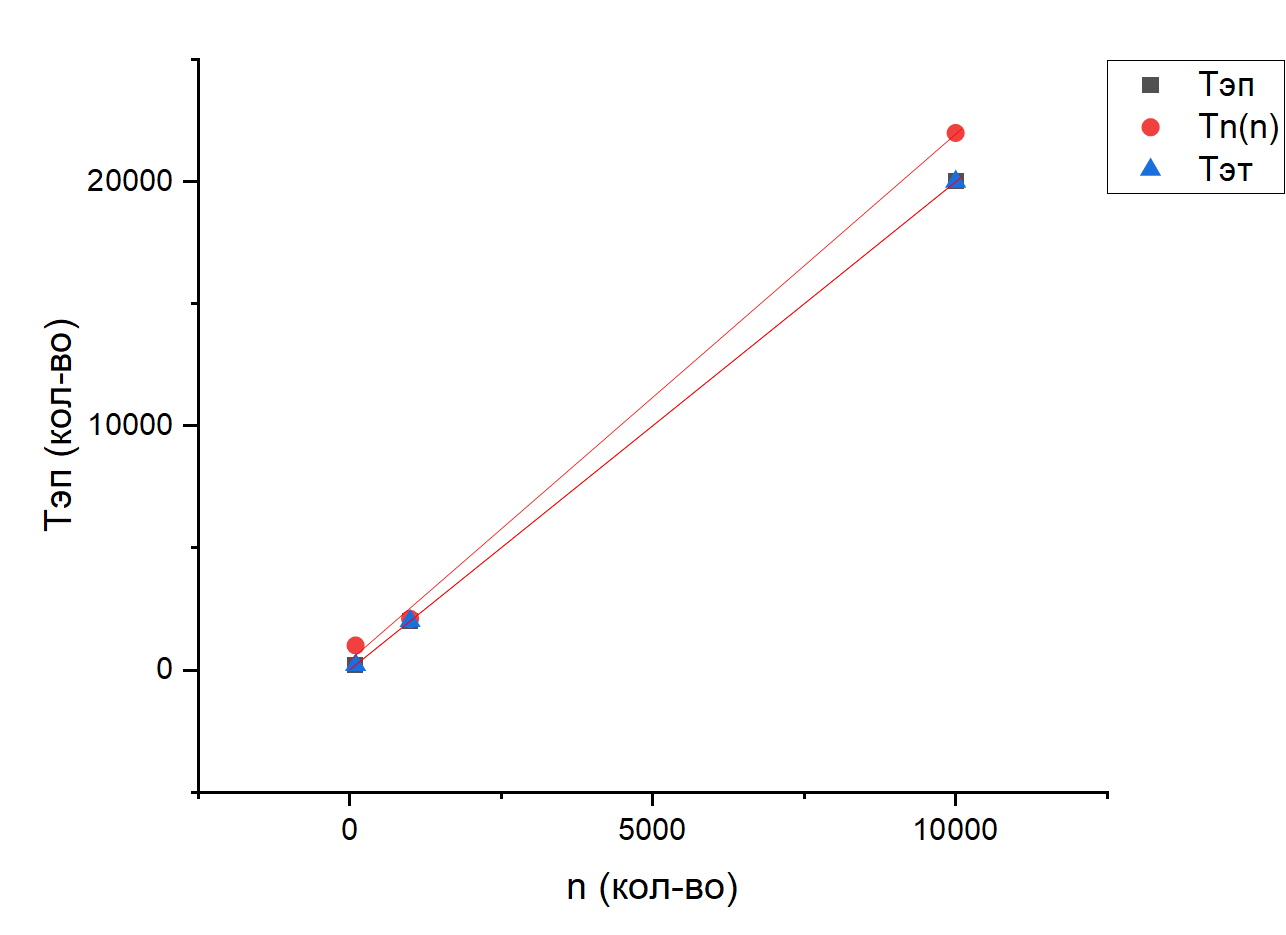
|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время,ns** | **Тэт=f(C+M)-функция, ns** | **Т**эп**=Cф+Mф- количество** |
| 100 | 900 | 399 | 399 |
| 1000 | 4100 | 3999 | 3999 |
| 10000 | 35600 | 39999 | 39999 |



Рассмотрим эмпирически количество выполняемых операций. В соответствии с кодом задачи, получим:

***Таблица 2. Сводная таблица результатов Linear Search Barrier***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время,ns** | **Тэт=f(C+M)-функция, ns** | **Т**эп**=Cф+Mф- количество** |
| 100 | 1000 | 206 | 206 |
| 1000 | 2100 | 2006 | 2006 |
| 10000 | 22000 | 20006 | 20006 |



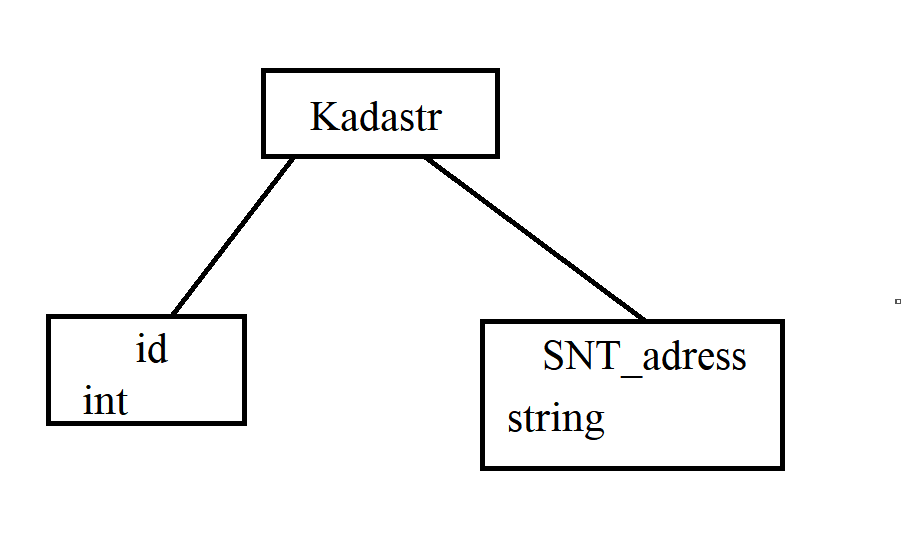
Графики построены в программе OriginPro 2022. Отчетливо видна линейная зависимость всех графиков. Расхождения с теоретическим расчетом времени выполнения пренебрежимо малы.

Данные алгоритмы подходят для небольшого объема данных и очень просты в написании. Однако их асимптотику нельзя назвать подходящей для больших данных

**Задание 2. Разработать программу поиска записи по ключу в таблице записей с применением алгоритма, определенного в задании варианта.**

1. Таблица содержит записи, структура которых определена вариантом. Ключи уникальны в пределах таблицы.
2. Разработать алгоритм поиска, определенный в варианте. Реализовать алгоритм функцией.
3. Провести практическую оценку времени выполнения алгоритмов на таблицах объемом 100, 1000, 10 000 записей на случайно заполненных таблицах (худший случай). На таблицах с лучшим временем и средним.
4. Составить таблицу с указанием: времени выполнения алгоритма, его фактическую и теоретическую вычислительную сложность.

**Структура записи**



**Реализация алгоритма**

Таблица в задании 2 аналогична прошлой задаче, состоит из строк, содержащих номер участка и название СНТ. В соответствии с персональным вариантом нужно использовать хеш-таблицу с открытой адресацией и линейным смещением. Открытая адресация позволяет разрешить коллизии в хеш таблице, однако объем данных ключей (шестизначное число) позволяет использовать идеальную хеш функцию – остаток от деления на миллион, а сама хеш таблица будет начального размера миллион. Данная реализация расходует много памяти, однако хеш функцию и изначальный размер массива можно изменить, реализовано динамическое расширение хеш-таблицы при коэффициенте заполнения больше, чем 0.7. Такая реализация позволяет добиться линейного времени на любых данных(по условию ключ уникален, а значит коллизии могут быть вызваны только хеш функцией)

Рассмотрим простой пример. Получаем на вход два ключа – 123456 и 245677. Оба передадим в хеш функцию, которая выдает хеш-номер в качестве остатка от деления на 1000000. Возможности пк позволяют создать массив arr длинной в 1000000 ячеек. В таком случае запись 123456 поместится в ячейку arr[123456], а запись 245677 в ячейку arr[245677]. Также в хеш таблице реализовано разрешение коллизий и динамическое расширение.

**Алгоритм поиска**

1. Передадим значение в хеш-функцию
2. Если значение arr[key] является искомым, то вернем его
3. Иначе продвигаемся по хеш таблице далее пока не найдется искомый элемент(линейный поиск)
4. Если элемент не найден, вернуть пустую запись

Данная реализация является поиском в хеш-таблице с открытой адресацией и линейным сдвигом

**Код метода поиска**

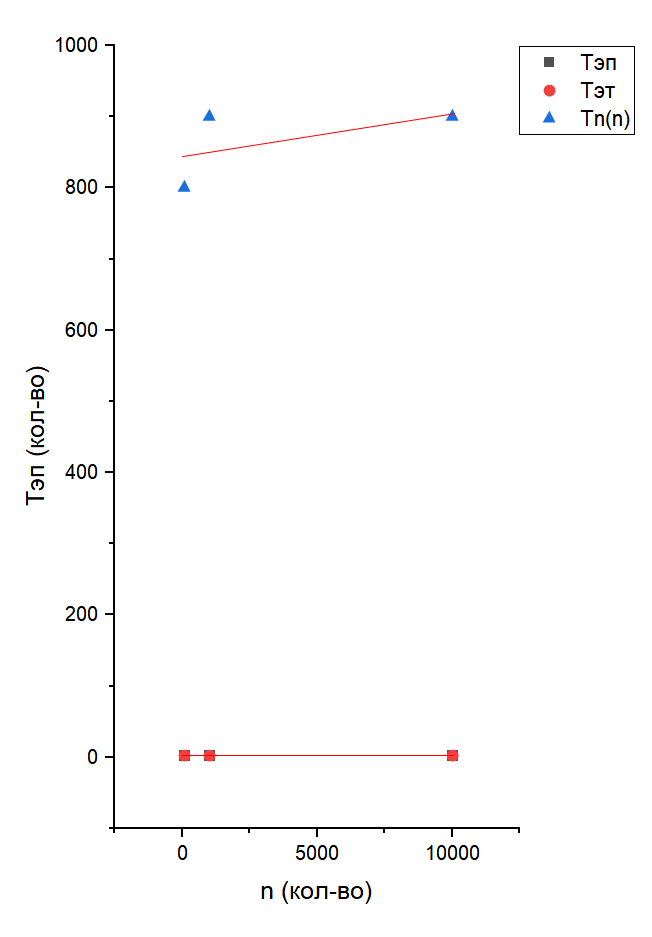
Предусловие – int id – номер искомого значения, хеш-таблица(описана как поле класса хеш-таблицы)

Постусловие – искомая запись типа Kadastr

1. Kadastr HashTable::get(int id) {
2. int index = HashFunction(id);
3. if(kadastr[index].id == id) {
4. return kadastr[index];
5. }
6. else {
7. int i = index;
8. while(kadastr[i].id != id) {
9. i++;
10. if(i == size) {
11. i = 0;
12. }
13. if(i == index) {
14. return Kadastr{0, "NO NAME"};
15. }
16. }
17. return kadastr[i];
18. }
19. }

***Таблица 3. Сводная таблица результатов HashTable***

|  |  |  |  |
| --- | --- | --- | --- |
| **n** | **T(n) время,ns** | **Тэт=f(C+M)-функция, ns** | **Т**эп**=Cф+Mф- количество** |
| 100 | 800 | 3 | 3 |
| 1000 | 900 | 3 | 3 |
| 10000 | 900 | 3 | 3 |



Рассчитанное время значительно отличается от фактического из-за сложности внутреннего устройства языка программирования: отсчет времени, возврат результата метода; а так же из-за малого отрезка времени, который в целом тяжело точно измерить

В представленной реализации средний случай является и худшим и лучшим, т. к. по условию ключи не повторяются, значит коллизий нет, поиск всегда выполняется за одинаковое время.

Данный алгоритм особенно эффективен, хотя и требует очень много памяти

**Выводы**

Очевидно, наиболее эффективным алгоритмом является хеш-таблица, однако она требует правильной реализации хеш-функции, алгоритма разрешения коллизий. Для небольших задач стоит выбрать линейный поиск

**Полный код программы**

1. #include <iostream>
2. #include <string>
3. #include <cstdint>
4. #include <cstring>
5. #include <chrono>
6. #include <random>
7. #include <ctime>
8. using namespace std;

11. int random\_number(){
12. return ((double)rand() / RAND\_MAX) \* (999999 - 100000) + 100000;
13. }
15. struct Kadastr{
16. int id;
17. string SNT\_adress;
18. };
19. struct KadastrTable{
20. Kadastr \*kadastr;
21. int size;
22. int append(Kadastr note);
23. void print();
25. Kadastr LinearSearchBarrier(int id);
26. Kadastr BruteForce(int id);
27. ~KadastrTable();
28. };
30. Kadastr KadastrTable::LinearSearchBarrier(int id){
31. if(size == 0) {
32. return Kadastr{0, "NO NAME"};
33. }
34. Kadastr last = kadastr[size-1];
35. kadastr[size-1].id = id;
36. int i;
37. for(i=0; kadastr[i].id!= id;++i){}
38. kadastr[size-1] = last;
39. if(i != size-1 || id == last.id){
40. return kadastr[i];
41. }
42. return Kadastr{0, "NO NAME"};
43. }
44. Kadastr KadastrTable::BruteForce(int id){
45. for(int i = 0; i < size; i++){
46. if(kadastr[i].id == id){
47. return kadastr[i];
48. }
49. }
50. return Kadastr{-1, "NULL"};
51. }
53. int KadastrTable::append(Kadastr note) {
54. Kadastr \*tmp = new Kadastr[size+ 1];
55. //memcpy(tmp, kadastr, size \* sizeof(Kadastr));
56. for (int i = 0; i < size; i++) {
57. tmp[i] = kadastr[i];
58. }
59. size++;
60. tmp[size - 1] = note;
61. delete[] kadastr;
62. kadastr = tmp;
63. return 1;
64. }
65. void KadastrTable::print() {
66. for (int i = 0; i < size; i++) {
67. cout << kadastr[i].id << " " << kadastr[i].SNT\_adress << endl;
68. }
69. }
70. KadastrTable::~KadastrTable() {
71. delete[] kadastr;
72. }
74. //Kadastr brute\_force( const KadastrTable& kadastr\_table, int id) {
75. // for (int i = 0; i < kadastr\_table.size; i++) {
76. // if (kadastr\_table.kadastr[i].id == id) {
77. // return kadastr\_table.kadastr[i];
78. // }
79. // }
80. // return Kadastr{0, "NO NAME"};
81. //}
82. //
83. //Kadastr linear\_search\_barrier(const KadastrTable& kadastr\_table, int id) {
84. // if(kadastr\_table.size == 0) {
85. // return Kadastr{0, "NO NAME"};
86. // }
87. // Kadastr last = kadastr\_table.kadastr[kadastr\_table.size-1];
88. // kadastr\_table.kadastr[kadastr\_table.size-1].id = id;
89. // int i;
90. // for(i=0; kadastr\_table.kadastr[i].id!= id;++i){}
91. // kadastr\_table.kadastr[kadastr\_table.size-1] = last;
92. // if(i != kadastr\_table.size-1 || id == last.id){
93. // return kadastr\_table.kadastr[i];
94. // }
95. // return Kadastr{0, "NO NAME"};
96. //}
97. //==================HASH TABLE==========================
98. struct HashTable{
99. const int start\_size = 1000000;
100. HashTable();
101. Kadastr \*kadastr;
102. int load=0;
103. int size;
104. double calculate\_load\_factor();
105. int add(Kadastr note);
106. Kadastr get(int id);
107. ~HashTable();
108. };
109. HashTable::HashTable() {
110. kadastr = new Kadastr[start\_size];
111. size = start\_size;
112. }
113. int HashFunction(int id) {
114. return id % 1000000;
115. }
116. double HashTable::calculate\_load\_factor() {
117. return (double)load/size;
118. }
119. int HashTable::add(Kadastr note) {
121. if(calculate\_load\_factor() > 0.7) {
122. Kadastr \*tmp = new Kadastr[size\*2];
123. for (int i = 0; i < size; i++) {
124. tmp[i] = kadastr[i];
125. }
126. delete[] kadastr;
127. kadastr = tmp;
128. size \*= 2;
129. }
131. int index = HashFunction(note.id);
132. if(kadastr[index].id == 0) {
133. kadastr[index] = note;
134. load++;
135. return 1;
136. }
137. else {
138. int i = index;
139. while(kadastr[i].id != 0) {
140. i++;
141. if(i == size) {
142. i = 0;
143. }
144. if(i == index) {
145. return 0;
146. }
147. }
148. kadastr[i] = note;
149. load++;
150. return 1;
151. }
152. }
154. HashTable::~HashTable() {
155. delete[] kadastr;
156. }
158. Kadastr HashTable::get(int id) {
159. int index = HashFunction(id);
160. if(kadastr[index].id == id) {
161. return kadastr[index];
162. }
163. else {
164. int i = index;
165. while(kadastr[i].id != id) {
166. i++;
167. if(i == size) {
168. i = 0;
169. }
170. if(i == index) {
171. return Kadastr{0, "NO NAME"};
172. }
173. }
174. return kadastr[i];
175. }
176. }
178. int main() {
179. srand(time(NULL));
180. for(int i = 100, d=0; i<=10000;i = d%2==0 ? i\*=10 : i+=0){
181. d++;
182. KadastrTable kadastr\_table{};
183. HashTable hash\_table{};
184. for(int j = 0; j<i; j++){
185. int id = random\_number();
186. kadastr\_table.append(Kadastr{id, "SNT\_" + to\_string(id)});
187. hash\_table.add(Kadastr{id, "SNT\_" + to\_string(id)});
188. }
189. int random\_search =kadastr\_table.kadastr[rand()%i].id;
190. cout<<"Random note: "<<random\_search<<endl;
191. int search\_id = random\_search;
192. //cin>>search\_id;

195. auto start = chrono::steady\_clock::now();
196. kadastr\_table.BruteForce(search\_id);
197. auto end = chrono::steady\_clock::now();
198. auto diff = chrono::duration\_cast<chrono::nanoseconds>(end - start).count();
199. cout<<"Brute force(" << i << "): "<< diff <<" ns"<<endl;
201. start = chrono::steady\_clock::now();
202. kadastr\_table.LinearSearchBarrier(search\_id);
203. end = chrono::steady\_clock::now();
204. diff = chrono::duration\_cast<chrono::nanoseconds>(end - start).count();
205. cout<<"Linear search barrier(" << i << "): "<<diff<<" ns"<<endl;

208. Kadastr temp;
209. start = chrono::steady\_clock::now();
210. temp = hash\_table.get(search\_id);
211. end = chrono::steady\_clock::now();
212. diff = chrono::duration\_cast<chrono::nanoseconds>(end - start).count();
213. cout<<"Hash table(" << i << "): "<< diff <<" ns"<<endl;
214. }
215. return 0;
216. }