

## Звіт

Автор: Момот Р. КІТ-119а

Дата: 02.05.2020

### ЛАБОРАТОРНА РОБОТА № 10. АЛГОРИТМИ ПОШУКУ З ВИКОРИСТАННЯМ ТАБЛИЦЬ

**Мета:** закріпити знання про алгоритми пошуку, що вимагають додаткової пам'яті; одержати навички виконання операцій пошуку із використанням таблиць прямого доступу, справочників та хешованих таблиць.

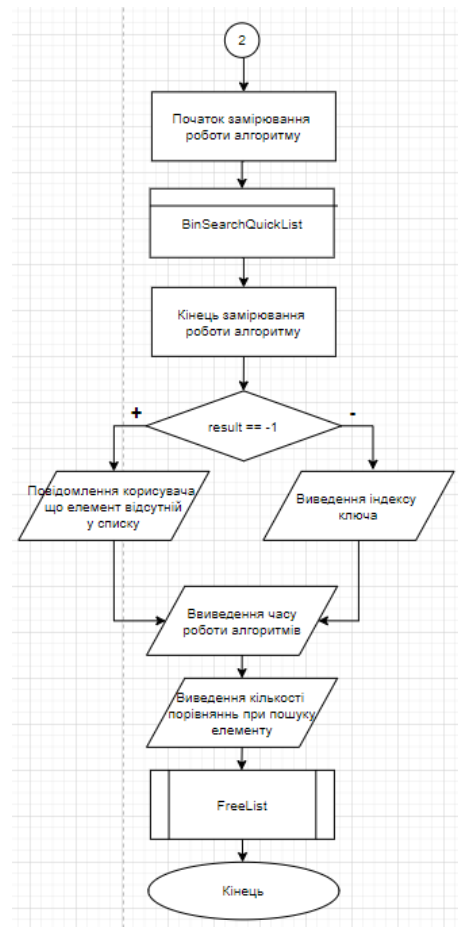
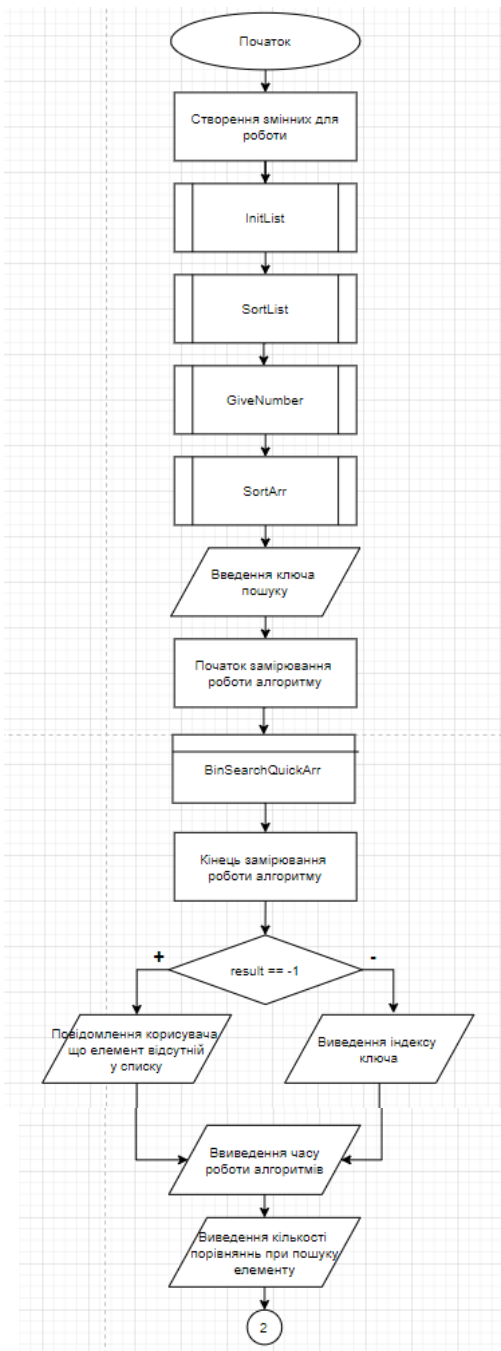
#### Індивідуальне завдання

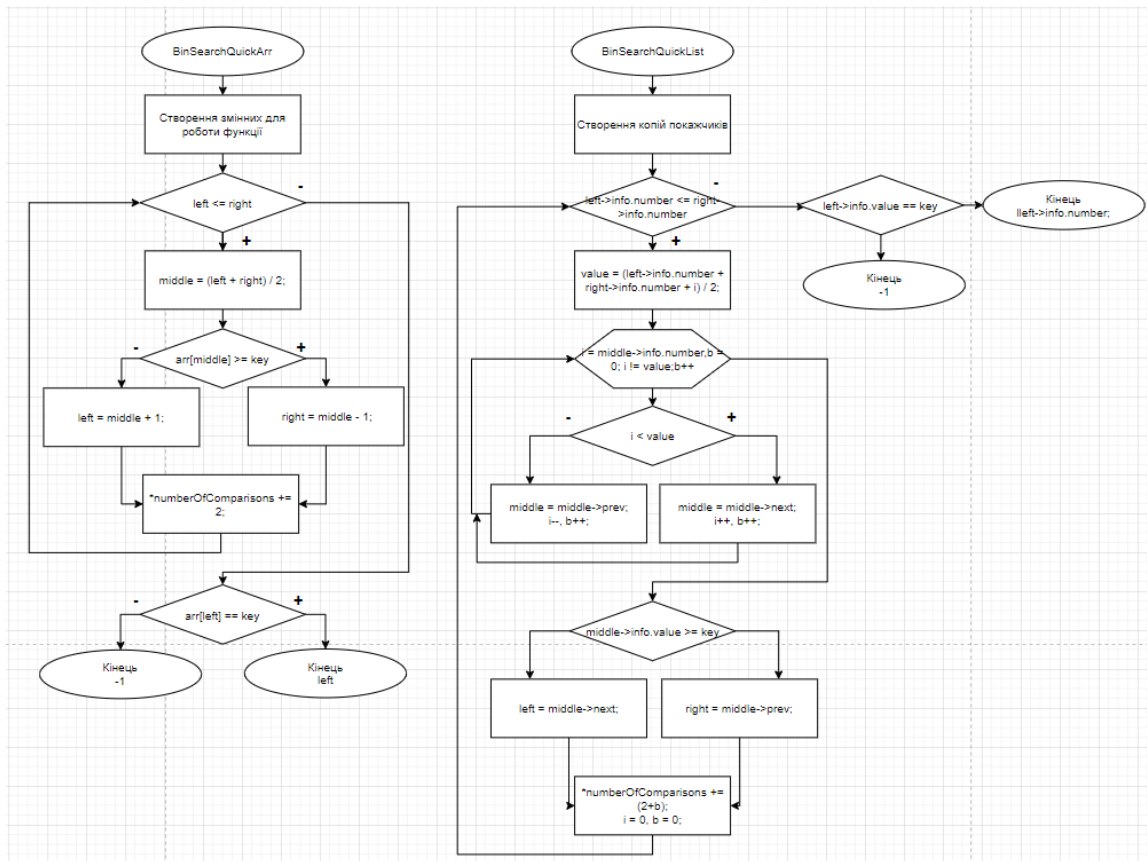
Для вмісту файла створити таблицю прямого доступу або хеш-таблицю у відповідності до індивідуального завдання. Перевірити працездатність створених таблиць на прикладі операцій пошуку.

Порівняти час пошуку із використанням створених таблиць та простих алгоритмів із лабораторної роботи №9. Для кожного з алгоритмів визначити кількість порівнянь у наборі даних з різною кількістю елементів (20, 1000, 5000, 10000, 50000) визначити час пошуку, заповнити таблицю по формі, побудувати графіки, зробити висновки.

N	Вміст вихідних даних	Таблиця	Хеш-функція	Ключ пошуку
13	Номер телефону, прізвище власника, адреса, час розмови	Хеш-таблиця з розподіленими ланцюжками переповнення	Функція середини квадрата	Номер телефону

## Блок-схема алгоритму програми





## Текст програми

```
#include <iostream>
#include <iomanip>
#include <chrono>
#include <sstream>
#include <fstream>
#include <cmath>
using namespace std;
const int SIZE = 50000;           //количество элементов в списке

struct HashTable
{
    long long PhoneNumber;
    string Surname;
    int TalkTime;
    string Address;

    HashTable* next;
} *Htable[SIZE];

long long Hash(long long);
void TableInitialization();
HashTable* Search(int);
void Insert(long long, int, string, string);
void Task();
void DeleteTable();

void main()
{
    setlocale(LC_ALL, "Rus");
    srand(time(NULL));

    Task();
}

void Task()
{
    string* SurnameList = new string[SIZE];
    long long* PhoneNumbersList = new long long[SIZE];
    int* TalkTimeList = new int[SIZE];
    string* AddressList = new string[SIZE];
    long long key;
    int SurnameLength;
    int AddressLength;
    long long value, value2;

    TableInitialization();

    for (size_t i = 0; i < SIZE; i++)
    {
        SurnameLength = rand() % 10 + 5;

        SurnameList[i] = rand() % 31 + 192;
        for (int j = 0; j < SurnameLength - 1; j++)
            SurnameList[i] += rand() % 31 + 224;
    }

    for (size_t i = 0; i < SIZE; i++)
    {
        PhoneNumbersList[i] = rand() % 9 + 0;
        for (size_t j = 1; j <= 5; j++)
        {
            key = rand() % 9 + 0;
            value = pow(10, j);
```

```

        PhoneNumbersList[i] += (key*value);
    }
}
key = 0;

for (size_t i = 0; i < SIZE; i++)
    TalkTimeList[i] = rand() % 5000 + 10;

for (int i = 0; i < SIZE; i++)
{
    AddressLength = rand() % 15 + 5;

    AddressList[i] = rand() % 31 + 192;
    for (int j = 0; j < AddressLength - 1; j++)
        AddressList[i] += rand() % 31 + 224;
}

for (size_t i = 0; i < SIZE; i++)
    Insert(PhoneNumbersList[i], TalkTimeList[i], SurnameList[i], AddressList[i]);

auto time = chrono::steady_clock::now();
auto timeCp = time;
for (size_t i = 0; i < 10; i++)
{
    for (size_t i = 0; i < SIZE; i++)
    {
        key = rand() % 9 + 0;
        for (size_t j = 1; j <= 5; j++)
        {
            key = rand() % 9 + 0;
            value = pow(10, j);

            PhoneNumbersList[i] += (key * value);
        }
    }

    auto begin = chrono::steady_clock::now();
    Search(key);
    auto end = chrono::steady_clock::now();
    auto elapsed_ms = chrono::duration_cast<chrono::nanoseconds>(end - begin);
    time += elapsed_ms;
}
auto result = (time - timeCp) / 10;
cout << "Количество элементов: " << SIZE << "\nСреднее время поиска: " << result.count()
<< endl;

delete[] SurnameList;
delete[] PhoneNumbersList;
delete[] TalkTimeList;
delete[] AddressList;
DeleteTable();
}

void TableInitialization()
{
    for (int i = 0; i < SIZE; i++)
        Htable[i] = NULL;
}

void Insert(long long phone, int talkTime, string surname, string address)
{
    HashTable* temp = new HashTable;
    long long key = Hash(phone);

    temp->next = NULL;

```

```

temp->PhoneNumber = phone;
temp->TalkTime = talkTime;
temp->Surname = surname;
temp->Address = address;

//cout << setw(3) << key << endl;
//cout << temp->PhoneNumber << "\t" << temp->TalkTime << "\t" << temp->Surname << "\t"
<< temp->Address << endl;

if (!Htable[key]) Htable[key] = temp;
else
{
    temp->next = Htable[key];
    Htable[key] = temp;
}
}

long long Hash(long long key)
{
    key *= key;                      // возвести ключ в квадрат
    key >>= 11;
    return key % 2048;               // вернуть 10 младших бит
}

HashTable* Search(int x)
{
    int key = Hash(x);
    HashTable* temp = Htable[key];
    if (!temp) return NULL;
    while ((temp->next) && (temp->PhoneNumber != x)) temp = temp->next;
    if (temp->PhoneNumber != x) return NULL;
    return temp;
}

void DeleteTable()
{
    struct HashTable* p, * pCopy;

    for (size_t i = 0; i < SIZE; i++)
    {
        p = Htable[i];
        if (Htable[i] != NULL)
            while (p)
            {
                pCopy = p;
                p = p->next;
                delete pCopy;
            }
    }
}

```

## Результати роботи програми

```

Числа в списке:
41 491 2995 4827 5436 5705 6334 9961 11478 11942 15724 16827 18467 19169 23281 24464 26500 26962 28145 29358
Введите число, которое надо найти в массиве: 19169

Индекс элемента 19169: 13
Время поиска элемента: 500ns
Количество сравнений: 10

Индекс элемента 19169: 13
Время поиска элемента: 1200ns
Количество сравнений: 48

Утечка памяти отсутствует.
_

```

### Результати тестування алгоритма пошуку для масива

Кількість елементів	20	100	1000	10000
Кількість порівнянь	11	16	22	29
Час пошуку*	533	667	933	933

### Результати тестування алгоритма пошуку для списку

Кількість елементів	20	100	1000	10000
Кількість порівнянь	148	217	2021	20026
Час пошуку*	867	2267	22167	316767

\*Час пошуку вказан у наносекундах.

\*\*У таблиці вказані середні значення після триразових перевірок даних.

### Графік часу роботи алгоритмів пошуку



## Висновок

У результаті роботи програми було розроблено програму, яка створює масив елементів та список розміром, вказаним користувачем та заповнює його випадковими елементами. Після цього масив та список сортуються для роботи з алгоритмом пошуку. За результатами роботи алгоритмів з масивами різних розмірів ми бачимо, що двійковий алгоритм працює повільніше у списку, тому що багато часу витрачається на переміщення покажчиків у списку, в той час, як у звичайному масиві доступ до елементів відбувається за індексом

## Відповіді на питання

1. Що визначає складність алгоритму?

При визначенні складності алгоритмів пошуку за часом визначають насамперед кількість операцій порівнянь та присвоювань.

2. Яка умова повинна виконуватися при пошуку ключа цілого типу?

Ключ повинен бути цілого типу.

3. Яка умова має виконуватися при пошуку ключа дійсного типу?

Ключ повинен бути дійсного типу.

4. В алгоритмі лінійного пошуку з бар'єром, що є бар'єром?

В алгоритмі лінійного пошуку з бар'єром, бар'єром є останнє додаткове число в масиві.

5. Перерахуйте усі відомі прості алгоритми пошуку по числовому ключу в порядку зменшення їх середнього часу пошуку.

Інтерполяційний алгоритм, бінарний, лінійний, лінійний з бар'єром,

6. Які обмеження накладаються на набір даних при лінійному пошуку з бар'єром та без нього?

При роботі з алгоритмом лінійного пошуку з бар'єром останній елемент повинен бути ключом. Це означає, що масив має розмір  $N+1$ . У звичайному лінійному пошуку без бар'єра таких обмежень немає.



7. Поясніть, як виконується двійковий пошук?

Спочатку приблизно визначається запис у середині таблиці й аналізується значення ключа цього запису. Якщо воно занадто велике, то аналізується значення ключа, у середині першої половини таблиці. Зазначена процедура повторюється в цій половині доти, поки не буде знайдений необхідний запис. Якщо значення ключа занадто мале, випробується ключ, що відповідає запису в середині другої половини таблиці, і процедура повторюється в цій половині. Цей процес продовжується доти, поки не буде знайдений необхідний ключ або не стане порожнім інтервал, у якому здійснюється пошук.

8. Реалізація якого алгоритма наведена в наступному фрагменті програмного коду?

```
{ while(m[i]!= key && i<N) i++;  
  
  if( m[i]== key) return i; else return -1; }
```

**Це алгоритм лінійного пошуку без бар'єра.**

9. Які обмеження накладаються на дані при бінарному пошуку?

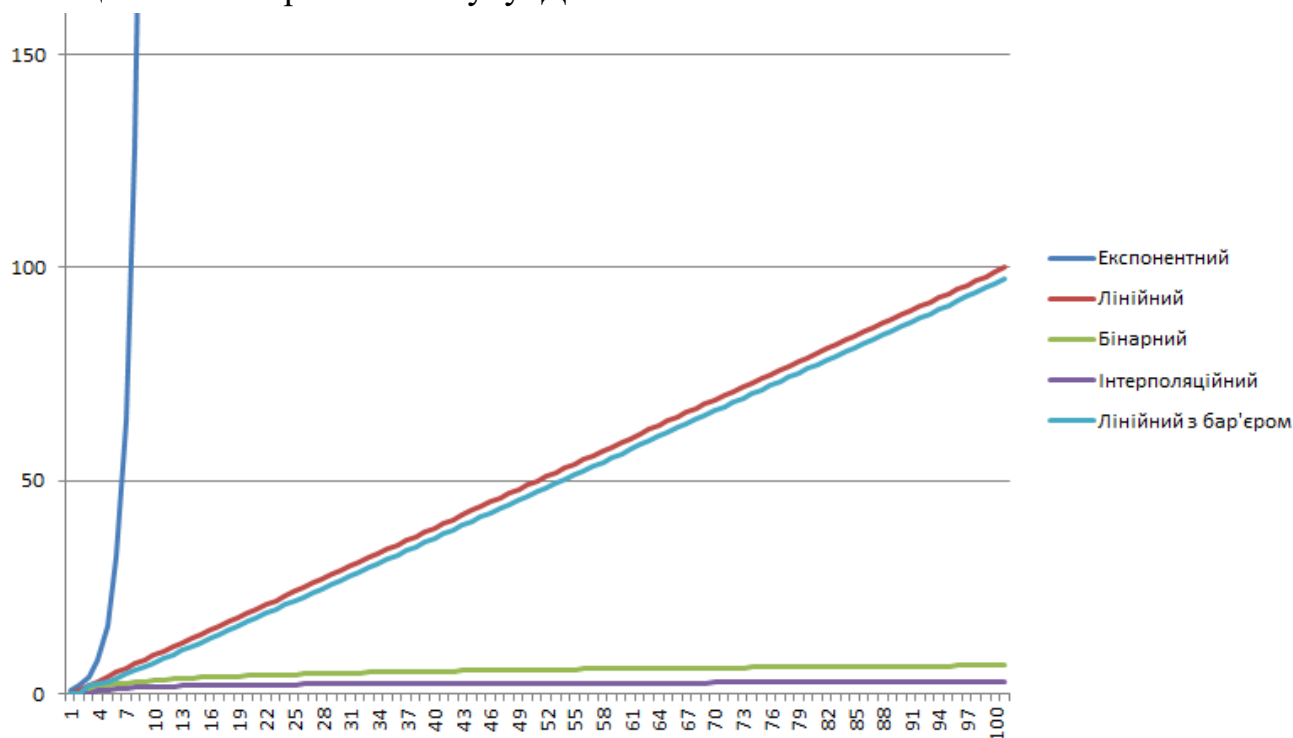
Дані для пошуку бінарним алгоритмом повинні бути упорядковані будь-яким методом сортування.

10. Реалізація якого алгоритма наведена в наступному фрагменті програмного коду?

```
{ while(m[i]!= key) i++;  
  
  if( i!= N) return i; else return -1; }
```

**Це алгоритм лінійного пошуку з бар'єром.**

11. Накресліть якісний графік залежності часу пошуку від кількості елементів в наборі даних для лінійного, лінійного з бар'єром, двійкового, інтерполяційного та експоненційного алгоритмів пошуку. Дайте пояснення.



Лінійний пошук та лінійний пошук з бар'єром мають однакові порядки алгоритму ( $O(N)$ ), але пошук з бар'єром швидше через те, що в циклі перевірюється більш простий вираз.

Різниця в швидкості роботи інтерполяційного метода помітна лише тоді, коли йде робота над масивами великих обсягів.

12. Поясніть, як виконується експоненційний пошук, в чому його відмінність від двійкового?

На першому етапі визначається діапазон елементів в масиві, де приблизно може перебувати ключ (програмний приклад 8.5). Для цього в циклі порівнюється ключ пошуку і елемент масиву. Поки ключ більше елемента масиву визначається значення індексу наступного елемента збільшенням в два рази поточного номера. Іншими словами випробовується елементи масиву з номерами 1, 2, 4, 8, 16 і т. Д. (В загальному вигляді  $2^i$ ). Звідси і назва алгоритму - експонентний.

Коли ключ пошуку виявляється менше чергового обраного елемента, настає другий етап. Тепер в знайденому інтервалі методом двійкового пошуку визначають положення ключа, якщо він в масиві є.

13. Поясніть алгоритм роботи БМ-пошука зразка в тексті.

На першому кроці будується таблиця зсувів для шуканого зразка. Далі поєднується початок рядка і зразка і починається перевірка з останнього символу зразка. Якщо останній символ зразка та відповідний йому при накладенні символ рядка не збігаються, зразок зрушується щодо рядка на величину, отриману з таблиці зміщень, і знову проводиться порівняння, починаючи з останнього символу зразка. Якщо ж символи збігаються, проводиться порівняння попереднього символу зразка і т. Д. Якщо всі символи зразка збіглися з накладеними символами рядки, значить знайшли входження і пошук закінчено. Якщо ж якийсь (не останній) символ зразка не збігається з відповідним символом рядка, зразок зсувається на один символ вправо і знову починається перевірка з останнього символу. Весь алгоритм виконується до тих пір, поки або не буде знайдено входження шуканого зразка, або не буде досягнутий кінець рядка.

14. Поясніть алгоритм роботи КМП-пошука зразка в тексті.

Цей алгоритм ґрунтується на тому, що починаючи кожен раз порівнювати зразок (підрядок) з самого початку, можна знищити цінну інформацію. Після часткового збігу початкової частини образу з відповідними символами тексту фактично вже відома, пройдена частина тексту, і можна обчислити деякі відомості (на основі самого образу), за допомогою яких потім виконується швидке переміщення по тексту.

15. В чому суттєва відмінність алгоритма прямого пошука зразка в тексті від алгоритмів КМП та БМ пошуків?

При прямому пошуку в циклі індекс збільшується до першого символу, що збігається з символом в шуканому тексті. Після чого починається цикл, який перевіряє текст на відповідність тексту, який щем.

КМП пошук виконує пошук переходами по тексту, а БМ робить це ще швидше.