Лабораторна робота №2

Тема: Використання хеш-функцій

Мета: познайомитися з хеш-функціями та хеш-таблицями та отримати навички програмування алгоритмів, що їх обробляють.

ТЕОРЕТИЧНІ ВІДОМОСТІ

Хеш-таблиця - це структура даних, яка реалізовує інтерфейс - *асоціативний масив*, а саме, вона дозволяє зберігати пари (ключ, значення) і здійснювати три операції: операцію додавання нової пари, операцію пошуку і операцію видалення за ключем.

Існує два типи хеш таблиць:

- З ланцюжками;
- З відкритою адресацією.

Різницею між цими типами хеш-таблиць є те, що масив *H*, що використовується для збереження даних, містить пари значень(з відкритою адресацією), або ж списки пар(з ланцюжками).

Виконання операцій в хеш-таблиці починається з обчислення хеш-функції від ключа. Отримане хеш-значення *i = hash(key)* відіграє роль індексу в масиві *H*. Після цього операція (додавання, видалення, пошук) перенаправляється об'єктові, який зберігається у відповідній комірці масиву *H[i]*. Ситуація, коли для різних ключів отримується одне й те саме хеш-значення, називається колізією. Такі події непоодинокі — наприклад, при додаванні в хеш-таблицю розміром 365 комірок, усього лише 23-х елементів, ймовірність колізії вже перевищує 50 відсотків (якщо кожний елемент може з однаковою ймовірністю потрапити в будь-яку комірку). Через це механізм розв'язання колізій — важлива складова будь-якої хеш-таблиці.

В деяких особливих випадках вдається взагалі уникнути колізій. Наприклад, якщо всі ключі елементів відомі заздалегідь (або дуже рідко змінюються), тоді для них можна знайти деяку досконалу хеш-функцію, яка розподілить їх за комірками хеш-таблиці без колізій. Хеш-таблиці, які використовують подібні хеш-функції, не потребують механізму розв'язання колізій, і називаються хеш-таблицями з прямою адресацією.

Властивості хеш-таблиці. Важлива властивість хеш-таблиць полягає в тому, що при деяких розумних припущеннях, всі три операції (пошук, вставлення, видалення елементів) зазвичай виконується за час *O(1)*. Але при цьому не гарантується, що час виконання окремої операції малий. Це пов'язано з тим, що при досягненні деякого значення коефіцієнта заповнення необхідно здійснювати перебудову індексу хеш-таблиці: збільшити розміри масиву H і заново додати в порожню хеш-таблицю всі пари.

Метод ланцюжків. Кожна комірка масиву H є вказівником на зв'язаний список (ланцюжок) пар ключ-значення, відповідних одному і тому самому хеш-значенню ключа.

Колізії просто призводять до того, що з'являються ланцюжки довжиною більше одного елемента. Операції пошуку або видалення елемента вимагають

перегляду всіх елементів відповідного ланцюжка, щоб знайти в ньому елемент з заданим ключем. Для додавання нового елемента необхідно додати елемент в кінець або початок відповідного списку, і, у випадку якщо коефіцієнт заповнення стане занадто великим, збільшити розмір масиву H і перебудувати таблицю. При припущенні, що кожний елемент може потрапити в будь-яку позицію таблиці H з однаковою ймовірністю і незалежно від того, куди потрапив будь-який елемент, пересічний час роботи операції пошуку елемента складає Θ(1 + α), де α — коефіцієнт заповнення таблиці.

Відкрита адресація. В стратегії, названій відкритою адресацією, всі записи зберігаються в самому масиві H. Коли додається новий запис, масив перевіряється в певному порядку доки не буде знайдена вільна комірка куди й вставиться елемент. У випадку пошуку елемента, масив сканується в тій самій послідовності доки потрібний запис або порожня комірка не буде знайдена, друге означає відсутність елемента. Назва «відкрита адресація» показує, що розташування («адреса») елемента не визначається його хеш-значенням. Цей метод також називають закритим хешуванням. В загальному випадку, послідовність в якій переглядаються комірки хеш-таблиці залежить тільки від ключа елемента, тобто це послідовність h0(x), h1(x), …, hn-1(x), де x — ключ елемента, а hi(x) — довільна функція, яка зіставляє кожен ключ комірки з хеш-таблицею. Перший елемент послідовності, зазвичай, дорівнює значенню деякої хеш-функції від ключа, а інші обчислюються від нього одним з наведених нижче способів. Для успішної роботи алгоритму пошуку, послідовність перебору має бути має бути такою, щоб всі комірки хеш-таблиці виявились переглянутими рівно по одному разу.

ЗАВДАННЯ 1

Розробити програму, яка читає з клавіатури цілі числа N, M (1 (ключ — ціле, дійсне число або рядок в залежності від варіанту завдання; значення — рядок; усі рядки до 255 символів), жодний з яких не повторюється та ще M ключів. Всі рядки розділяються пробілом або новим рядком. Програма зберігає пар рядків до хеш-таблиці та видає на екран значення, що відповідають переліченим ключам. Приклад входу для ключів-рядків.

Приклад входу для ключів-рядків.

```
3 2
abc x
gh yq
io qw
gh
io
Вихід.
yq
qw
```

Використовувати готові реалізації структур даних (наприклад, STL)

заборонено, але можна використати реалізацію рядків (наприклад, std::string у C++).
(Варіант 3) Ключ — ціле число; мультиплікативне хешування.

ОПИСАННЯ РОЗРОБЛЕНОГО ЗАСТОСУНКУ

Весь код програми реалізований у одному файлі реалізації lab2. У цьому файлі розв'язуються всі необхідні задачі відповідно до завдання.

Програмний код:

```
#include

#include
using namespace std;
struct List {

int data = NULL;

bool isFull = false;
List* next = nullptr;

};
void print(List* a) {

for (List* i = a; i != nullptr; i = i->next)

{

cout << i->data << " ";

}

}
class HashTable {
private:

static const int size= 8;

List* arr[size];

const double A = 0.6180339;
int HashFunction(int key) {

return abs(size * fmod(key * A, 1));
```

```
}

public:
HashTable(){

for (int i = 0; i < size; i++) {
arr[i] = new List;

}

}
void insert(int key, int value)

{

int hashKey = HashFunction(key);

if (!arr[hashKey]->isFull)

{

arr[hashKey]->data = value;

arr[hashKey]->isFull = true;

}

else

{

for (List* i = arr[hashKey]; ; i = i->next)

{

if (i->next == nullptr)
{

i ->next= new List();

i->next->data = value;

i->next->isFull = true;
```

```
i->next->next = nullptr;

arr[hashKey];

break;

}

}

}

}
List* search(int key) {

int hashkey = HashFunction(key);

return arr[hashkey];

}

};
int main() {

HashTable a;
int k, d;

char s;

while (true) {

cout << "Input key and data: ";

cin >> k >> d;

if (k > 256) {

cout << "Incorect key" << endl;

continue;

}
```

```
a.insert(k, d);
cout << "Countine ?(y/n)";

cin >> s;

if (s == 'n') break;

}

while (true) {

cout << "Iput key ";

cin >> k;
print(a.search(k));

cout << "Countine ?(y/n)";

cin >> s;

if (s == 'n') break;

cout << endl;

}

return 0;

}
```

РЕЗУЛЬТАТИ

.1 Розглянемо ситуацію, коли всі елементи вводяться коректно згідно з умовою завдання (рис. 3.1).

Рисунок 3.1 – Вводяться коректні дані

.2 При введенні кількості елементів N, що не задовольняє інтервал , програма попереджає про це й пропонує перезаписати цю змінну.(рис. 3.2).

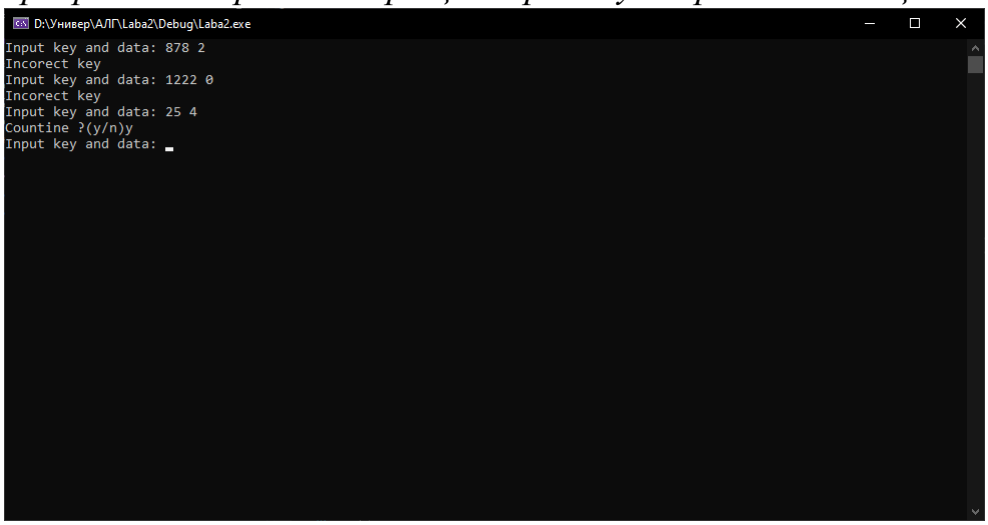

Рисунок 3.2 – N вводиться некоректно

.3 При виникнення колізії ключів, елемент додається до однобічного списку, що вирішує цю проблему (рис. 3.3).

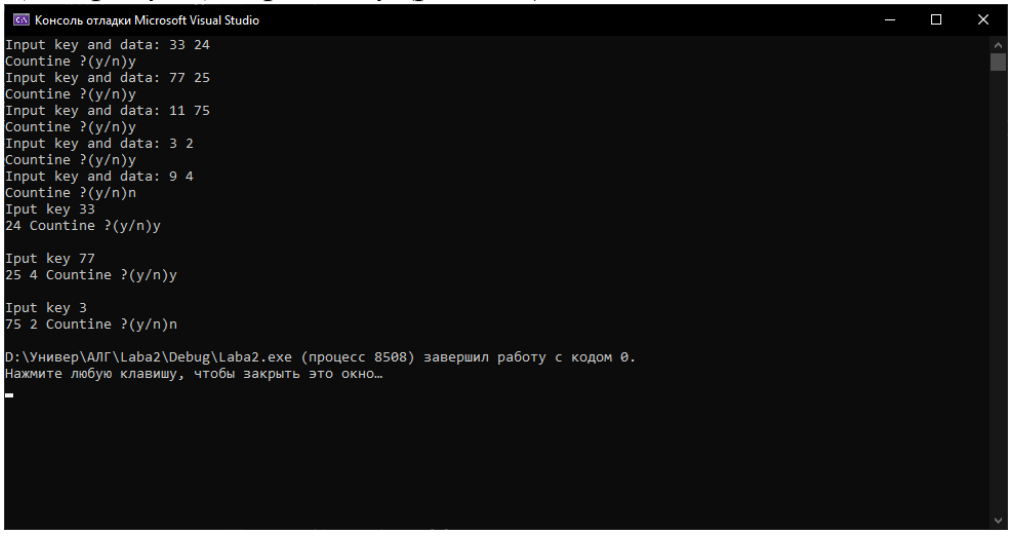

Рисунок 3.3 – Вводяться однакові елементи

Завдання 2

Реалізувати алгоритм пошуку ключа в масиві цілих чисел із використанням хеш-таблиці. Функція хешування – ділення по модулю. Алгоритм вирішення колізій – відкрите хешування з лінійним випробуванням.

Текст програми

```
#include <iostream>
using namespace std;
void Init(void);
int Insert(int key, int adr);
int Search(int key);
#define N 15 //число записів у таблиці
#define EMPTY -1
struct ElHashTabl
{ int key;
int adr;
};
ElHashTabl hashTabl[N]; //хеш - таблиця
int keys[N]={58,0,19,96,38,52,62,77,4,15,79,75,81,66,100};
void main(void)
{ int i, key, res;
Init();
cout <<"\nKeys -> ";
for (i=0;i<N;i++)
cout << keys[i] <<" ";
for (i=0;i<N;i++)
Insert(keys[i], i);
cout <<"\nHashed table (key-adr) -> ";
for (i=0;i<N;i++)
cout <<" " <<hashTabl[i].key <<"-" <<hashTabl[i].adr;
cout<<"\n";
for (i=0;i<N;i++)
{ cout <<" Input searched keys -> ";
cin >> key;
res=Search(key);
if (res==EMPTY) cout << " not search \n";
else cout << " " <<res <<"\n" ;
}
}
int Hash(int key)
{ return (key%N);
}
void Init(void)
{ for(int i=0;i<N;i++)
hashTabl[i].adr=EMPTY;
}
int Insert(int key, int adr)
{ int addr,a1;
```

```
addr=Hash(key);
if (hashTabl[addr].adr!=EMPTY)
{ a1=addr;
do
{ addr=Hash(addr+1);
}while(!((addr==a1)||(hashTabl[addr].adr==EMPTY)));
if (hashTabl[addr].adr != EMPTY)
return 0;
}
hashTabl[addr].key=key;
hashTabl[addr].adr=adr;
return 1;
}
int Search(int key)
{ int addr, a1;
addr=Hash(key);
if (hashTabl[addr].adr==EMPTY)
return EMPTY; //місце вільне – ключа немає в таблиці
else
if (hashTabl[addr].key==key)
return hashTabl[addr].adr; //ключ знайдений на своєму місці
else //місце зайняте іншим ключем
{ a1=Hash(addr+1);
//Пошук, поки не знайдений ключ чи не зроблений повний оборот
while((hashTabl[a1].key != key)&&(a1!=addr))
a1=Hash(a1+1);
if (hashTabl[a1].key != key)
return EMPTY;
else
return hashTabl[a1].adr;
```

Контрольні питання

1. Що записується в хеш-таблицю?
2. Чим визначається індекс запису в хеш-таблиці?
3. Які основні проблеми хешування і в чому вони полягають?
4. Скільки операцій порівняння виконується при пошуку по ключу із застосуванням таблиць прямого доступу?
5. Які ключи називають синонімами?
6. Які недоліки алгоритму пошуку по ключу з використанням таблиць прямого доступу?
7. Яке призначення хеш-функції?
8. Назвіть базові функції хешування.
9. Як створюється хеш-таблиця при реалізації метода відкритої адресації? Що зберігається в елементах такої хеш-таблиці?
10. Як створюється таблиця прямого доступу? Що зберігається в її елементах?
11. Назвіть алгоритми розв’язку колізій при відкритій адресації.