МІНІСТРЕСТВО ОСВІТИ Й НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

“Харківський Політехнічний Інститут”

Кафедра управління проєктами в інформаційних технологіях

Звіт з лабораторної роботи №4

“Хеш-таблиці та робота з ними”

з дисципліни

“Алгоритми та структури даних”

Варіант №5

Перевірив: ст. викл. каф. УПІТ Мошко Є.О.

Виконав: ст. гр. КН-1223г Шинкаренко О.В.

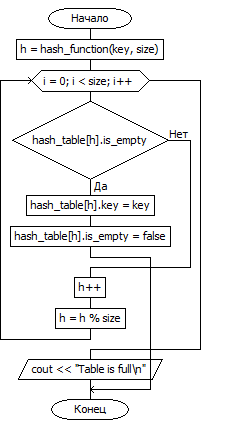
Харків – 2024

**Мета:** Вивчити роботу алгоритмів: прямої адресації, хеш-таблиці і відкритої адресації. Реалізувати перелічені вище алгоритми і провести їх порівняльний аналіз.

**Завдання:**

* 1. Нехай розмір хеш-таблиці дорівнює m = 20, а хеш-функція має вигляд: h(k) = [m(kA mod 1)], где А = ( 5 - l)/2. В які позиції потраплять ключі (див. таблицю, стовпець 3)?
* 2. Як буде виглядати хеш-таблиця з ланцюжками після того, як в неї послідовно помістили елементи з ключами (див. таблицю, стовпець 2) (в зазначеному порядку)? Число позицій в таблиці дорівнює 9, хеш-функція має вигляд h(k) = k mod 9.
* 3. Виконайте додавання ключів (в зазначеному порядку, див. таблицю стовпець 2) в хеш-таблицю з відкритою адресацією розміру m = 11. Для обчислення послідовності проб використовується лінійний метод з h'(k) = k mod m. Виконайте те ж завдання, якщо використовується квадратичний метод з тієї ж h', c1 = 1, c2 = 3, а також для подвійного хешування з h1 = h' и h2(k) = 1 + (k mod (m - 1)) . Використовувати готові реалізації структур даних (наприклад, STL) заборонено, але можна використати реалізацію рядків (наприклад, std::string у C++).

Додавання елементів



Блок-схема 1 Додавання елементів

Вхідні дані: посилання на таблицю, розмір таблиці, ключ

Код функції:

void add(hash\_node\* hash\_table, int key, int size) {

int h = hash\_function(key, size);

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

{

if (hash\_table[h].is\_empty) {

hash\_table[h].key = key;

hash\_table[h].is\_empty = false;

return;

}

h++;

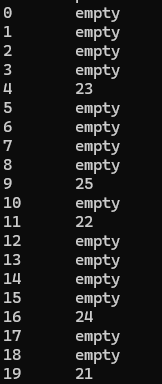
h = h % size;

}

cout << "Table is full\n";

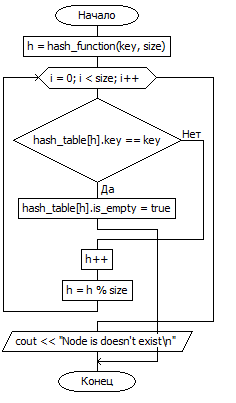
}

Результат у консолі:



Виведення у консоль 1 Додавання елементів

Видалення елементів



Блок-схема 2 Видалення елементів

Вхідні дані: посилання на таблицю, розмір таблиці, ключ

Код функції:

void delete\_node(hash\_node\* hash\_table, int key, int size) {

int h = hash\_function(key, size);

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

{

if (hash\_table[h].key == key) {

hash\_table[h].is\_empty = true;

return;

}

h++;

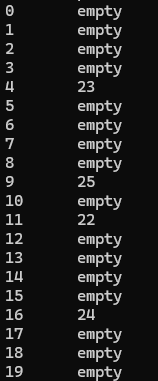
h = h % size;

}

cout << "Node is doesn't exist\n";

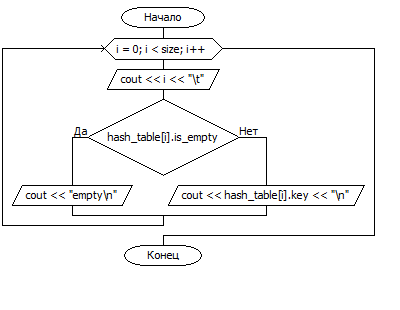
}

Результат у консолі:



Виведення у консоль 2 Видалення елементів

Роздрукування функції



Блок-схема 3 Роздрукування функції

Вхідні дані: посилання на таблицю, розмір таблиці

Код функції:

void print\_hash\_table(hash\_node\* hash\_table, int size) {

for (int i = 0; i < size; i++) {

cout << i << "\t";

if (hash\_table[i].is\_empty) {

cout << "empty\n";

}

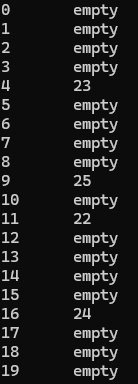
else {

cout << hash\_table[i].key << "\n";

}

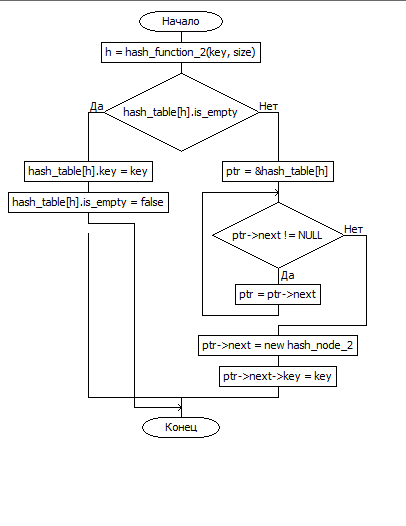
}

}



Виведення у консоль 3 Роздрукування функції

Додавання елементів методом ланцюжків



Блок-схема 4 Додавання елементів методом ланцюжків

Вхідні дані: посилання на таблицю, розмір таблиці, ключ

Код функції:

void add\_2(hash\_node\_2\* hash\_table, int key, int size) {

int h = hash\_function\_2(key, size);

if (hash\_table[h].is\_empty) {

hash\_table[h].key = key;

hash\_table[h].is\_empty = false;

return;

}

else {

hash\_node\_2\* ptr;

ptr = &hash\_table[h];

while (ptr->next != NULL) {

ptr = ptr->next;

}

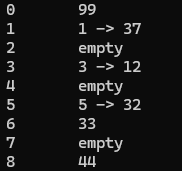
ptr->next = new hash\_node\_2;

ptr->next->key = key;

}

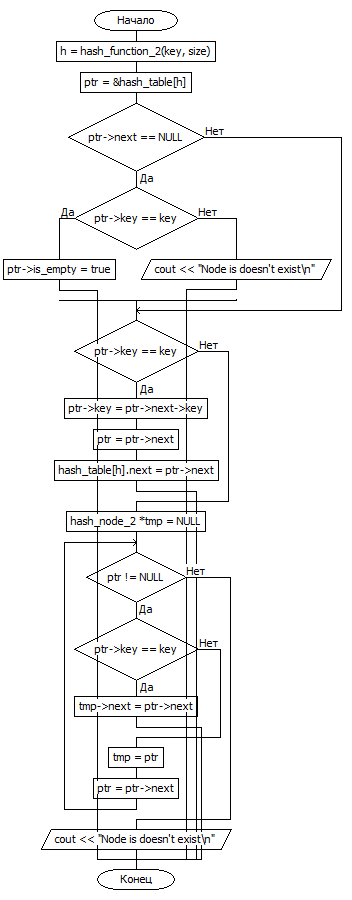
}

Результат у консолі:



Виведення у консоль 4 Додавання елементів методом ланцюжків

Видалення елементів методом ланцюжків



Блок-схема 5 Видалення елементів методом ланцюжків

Вхідні дані: посилання на таблицю, розмір таблиці, ключ

Код функції:

void delete\_node\_2(hash\_node\_2\* hash\_table, int key, int size) {

int h = hash\_function\_2(key, size);

hash\_node\_2\* ptr;

ptr = &hash\_table[h];

if (ptr->next == NULL) {

if (ptr->key == key) {

ptr->is\_empty = true;

return;

}

else {

cout << "Node is doesn't exist\n";

return;

}

}

if (ptr->key == key) {

ptr->key = ptr->next->key;

ptr = ptr->next;

hash\_table[h].next = ptr->next;

delete ptr;

return;

}

hash\_node\_2\* tmp = NULL;

while(ptr != NULL)

{

if (ptr->key == key) {

tmp->next = ptr->next;

delete ptr;

return;

}

tmp = ptr;

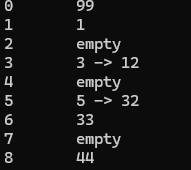
ptr = ptr->next;

}

cout << "Node is doesn't exist\n";

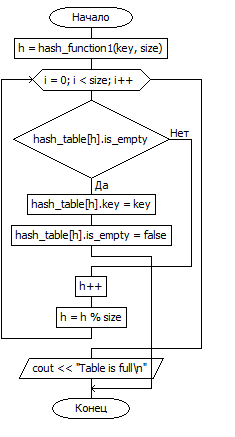
}

Результат у консолі:



Виведення у консоль 5 Видалення елементів методом ланцюжків

Додавання елементів методом лінійної адресації



Блок-схема 4 Додавання елементів методом лінійної адресації

Вхідні дані: посилання на таблицю, розмір таблиці, ключ

Код функції:

void add\_linar(hash\_node\* hash\_table, int key, int size) {

int h = hash\_function1(key, size);

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

{

if (hash\_table[h].is\_empty) {

hash\_table[h].key = key;

hash\_table[h].is\_empty = false;

return;

}

h++;

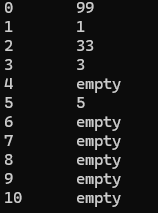
h = h % size;

}

cout << "Table is full\n";

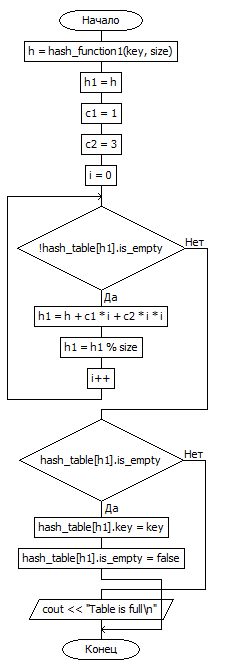
}

Результат у консолі:



Виведення у консоль 4 Додавання елементів методом лінійної адресації

Додавання елементів методом квадратичної адресації



Блок-схема 4 Додавання елементів методом квадратичної адресації

Вхідні дані: посилання на таблицю, розмір таблиці, ключ

Код функції:

void add\_square(hash\_node\* hash\_table, int key, int size) {

int h = hash\_function1(key, size);

int h1 = h;

int c1 = 1;

int c2 = 3;

int i = 0;

while (!hash\_table[h1].is\_empty)

{

h1 = h + c1 \* i + c2 \* i \* i;

h1 = h1 % size;

i++;

}

if (hash\_table[h1].is\_empty) {

hash\_table[h1].key = key;

hash\_table[h1].is\_empty = false;

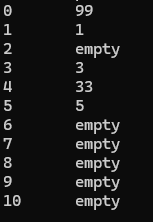
return;

}

cout << "Table is full\n";

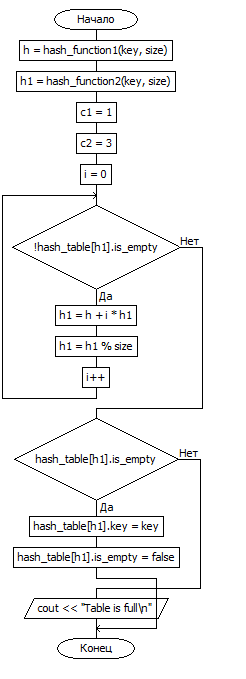
}

Результат у консолі:



Виведення у консоль 4 Додавання елементів методом квадратичної адресації

Додавання елементів методом двойного хешування



Блок-схема 4 Додавання елементів методом двойного хешування

Вхідні дані: посилання на таблицю, розмір таблиці, ключ

Код функції:

void add\_double(hash\_node\* hash\_table, int key, int size) {

int h = hash\_function1(key, size);

int h1 = hash\_function2(key, size);

int c1 = 1;

int c2 = 3;

int i = 0;

while (!hash\_table[h1].is\_empty)

{

h1 = h + i \* h1;

h1 = h1 % size;

i++;

}

if (hash\_table[h1].is\_empty) {

hash\_table[h1].key = key;

hash\_table[h1].is\_empty = false;

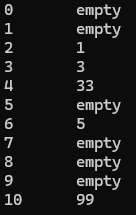
return;

}

cout << "Table is full\n";

}

Результат у консолі:



Виведення у консоль 4 Додавання елементів методом двойного хешування

**Висновки**

У процесі роботи над лабораторною, я вивчила алгоритми прямої адресації, хеш-таблиць і відкритої адресації. Це дозволило мені не лише зміцнити свої знання в теорії, але й отримати практичний досвід у реалізації цих алгоритмів. Я вивчила різні аспекти роботи хеш-таблиць, їхні методи хешування. Розробка та тестування алгоритмів дали мені можливість побачити, як вони працюють у дії, що значно покращило моє розуміння їхніх переваг і недоліків.

Хеш-таблиці мають кілька переваг, серед яких швидкість доступу до даних. Проте, хеш-таблиці також мають недоліки, зокрема, потребу в ефективному хешуванні для уникнення колізій і проблеми з динамічною зміною розміру. У випадку великої кількості колізій продуктивність може суттєво знижуватися, і від цього страждає ефективність роботи структури.

Порівнюючи хеш-таблиці з іншими структурами даних, такими як масиви або дерева, можна помітити, що кожна структура має свої переваги. Наприклад, масиви забезпечують швидкий доступ за індексом, але не підходять для динамічних наборів даних. Дерева, такі як бінарні дерева пошуку, забезпечують відсортованість даних і хорошу продуктивність при виконанні запитів, але їхня складність може бути вищою. Хеш-таблиці ж, завдяки своїй швидкості, ідеально підходять для сценаріїв, де потрібно здійснювати часті операції вставки та пошуку, але їхній потенціал залежить від правильно обраної функції хешування та обробки колізій.