Лабораторная работа №3

Хеширование

Процесс поиска данных в больших объемах информации сопряжен с временными затратами, которые обусловлены необходимостью просмотра и сравнения с ключом поиска значительного числа элементов. Сокращение поиска возможно осуществить путем локализации области просмотра. Например, отсортировать данные по ключу поиска, разбить на непересекающиеся блоки по некоторому групповому признаку или поставить в соответствие реальным данным некий код, который упростит процедуру поиска.

В настоящее время используется широко распространенный метод обеспечения быстрого доступа к информации, хранящейся во внешней памяти – хеширование.

Хеширование – это преобразование входного массива данных определенного типа и произвольной длины в выходную битовую строку фиксированной длины. Такие преобразования также называются хеш-функциями, а их результаты называют хешем, хеш-кодом, хеш-таблицей.

Хеш-таблица – это структура данных, реализующая интерфейс ассоциативного массива, то есть она позволяет хранить пары вида "ключ- значение" и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу. Хеш-таблица является массивом, формируемым в определенном порядке хеш-функцией.

Хорошей, с точки зрения практического применения, является такая хеш-функция, которая удовлетворяет следующим условиям:

- функция должна быть простой с вычислительной точки зрения;
- функция должна распределять ключи в хеш-таблице наиболее равномерно;
- функция не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов;
- функция должна минимизировать число коллизий – ситуаций, когда разным ключам соответствует одно значение хеш-функции (ключи в этом случае называются синонимами).

При этом первое свойство хорошей хеш-функции зависит от характеристик компьютера, а второе – от значений данных.

Если бы все данные были случайными, то хеш-функции были бы очень простые. Однако на практике случайные данные встречаются достаточно редко, и приходится создавать функцию, которая зависела бы от всего ключа. Если хеш-функция распределяет совокупность возможных ключей равномерно по множеству индексов, то хеширование эффективно разбивает множество ключей. Наихудший случай – когда все ключи хешируются в один индекс.

При возникновении коллизий необходимо найти новое место для хранения ключей, претендующих на одну и ту же ячейку хеш-таблицы. Причем, если коллизии допускаются, то их количество необходимо минимизировать. В некоторых случаях удается избежать коллизий вообще. Например, если все ключи элементов известны заранее (или очень редко меняются), то для них можно найти некоторую инъективную хеш-функцию, которая распределит их по ячейкам хеш-таблицы без коллизий. Хеш-таблицы, использующие подобные хеш-функции, не нуждаются в механизме разрешения коллизий, и называются хеш-таблицами с прямой адресацией.

Хеш-таблицы должны соответствовать следующим свойствам:

- Выполнение операции в хеш-таблице начинается с вычисления хеш-функции от ключа. Получающееся хеш-значение является индексом в исходном массиве.
- Количество хранимых элементов массива, деленное на число возможных значений хеш-функции, называется коэффициентом заполнения хеш-таблицы (load factor) и является важным параметром, от которого зависит среднее время выполнения операций.
- Операции поиска, вставки и удаления выполняются в среднем за время $O(1)$. Однако при такой оценке не учитываются аппаратные затраты на перестройку индекса хеш-таблицы, связанную с увеличением размера массива и добавлением в хеш-таблицу новой пары.
- Механизм разрешения коллизий является важной составляющей любой хеш-таблицы.

Хеширование полезно, когда широкий диапазон возможных значений должен быть сохранен в малом объеме памяти, и нужен способ быстрого, практически произвольного доступа. Хэш-таблицы часто применяются в базах данных, и, особенно, в языковых процессорах типа компиляторов и ассемблеров, где они повышают скорость обработки таблицы идентификаторов. В качестве использования хеширования в повседневной жизни можно привести примеры распределение книг в библиотеке по тематическим каталогам, упорядочивание в словарях по первым буквам слов, шифрование специальностей в вузах и т.д.

Методы разрешения коллизий

Коллизии осложняют использование хеш-таблиц, так как нарушают однозначность соответствия между хеш-кодами и данными. Способы преодоления возникающих сложностей:

- метод цепочек (внешнее или открытое хеширование);
- метод открытой адресации (закрытое хеширование).

Метод цепочек. Технология сцепления элементов состоит в том, что элементы множества, которым соответствует одно и то же хеш-значение, связываются в цепочку-список. В позиции номер i хранится указатель на голову списка тех элементов, у которых хеш-значение ключа равно i; если таких элементов в множестве нет, в позиции i записан NULL. На рисунке 1 демонстрируется реализация метода цепочек при разрешении коллизий. На ключ 002 претендуют два значения, которые организуются в линейный список.

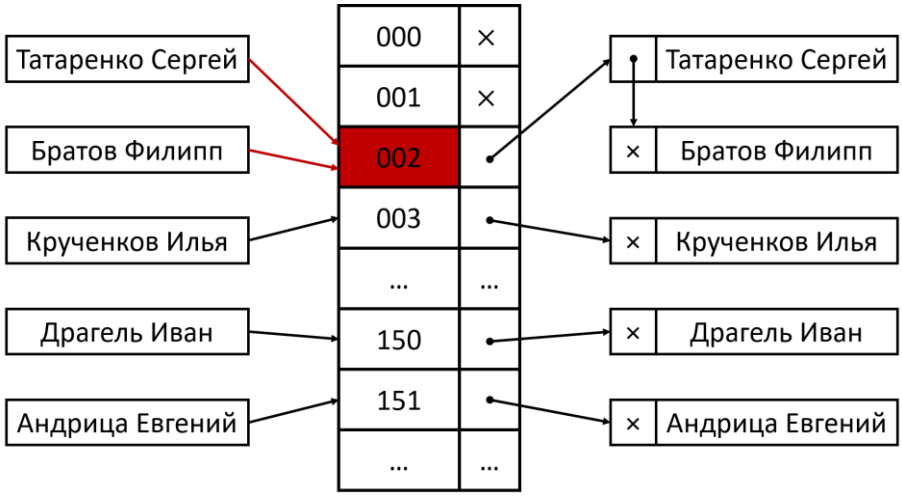

Рисунок 1 – Разрешение коллизий при помощи цепочек

Каждая ячейка массива является указателем на связный список (цепочку) пар ключ-значение, соответствующих одному и тому же хеш-значению ключа. Коллизии просто приводят к тому, что появляются цепочки длиной более одного элемента.

Операции поиска или удаления данных требуют просмотра всех элементов соответствующей ему цепочки, чтобы найти в ней элемент с заданным ключом. Для добавления данных нужно добавить элемент в конец или начало соответствующего списка, и, в случае если коэффициент заполнения станет очень велик, увеличить размер массива и перестроить таблицу.

При предположении, что каждый элемент может попасть в любую позицию таблицы с равной вероятностью и независимо от того, куда попал любой другой элемент, среднее время поиска элемента составляет $O(1 + k)$, где k – коэффициент заполнения таблицы.

Метод открытой адресации. В отличие от хеширования с цепочками, при открытой адресации никаких списков нет, а все записи хранятся в самой хеш-таблице. Каждая ячейка таблицы содержит либо элемент динамического множества, либо NULL.

В этом случае, если ячейка с вычисленным индексом занята, то можно просто просматривать следующие записи таблицы по порядку до тех пор, пока не будет найден ключ K или пустая позиция в таблице. Для вычисления шага можно также применить формулу, которая и определит способ изменения шага. На рисунке 2 разрешение коллизий осуществляется методом открытой адресации (для претендующего значения находится первое свободное (еще незанятое) место).

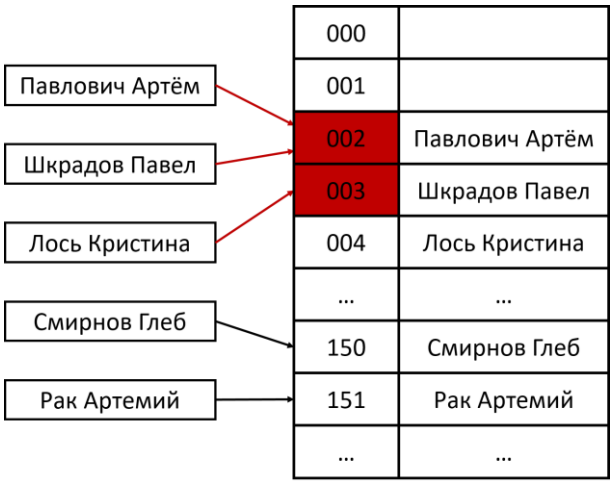

Рисунок 2 – Разрешение коллизий при помощи открытой адресации

При любом методе разрешения коллизий необходимо ограничить длину поиска элемента. Если для поиска элемента необходимо более 3-4 сравнений, то эффективность использования такой хеш-таблицы пропадает и следует найти другую хеш-функцию.

Для успешной работы алгоритмов поиска, последовательность проб должна быть такой, чтобы все ячейки хеш-таблицы оказались просмотренными ровно по одному разу.

Удаление элементов в такой схеме несколько затруднено. Обычно поступают так: заводят логический флаг для каждой ячейки, помечающий, удален ли в ней элемент. Тогда удаление элемента состоит в установке этого флага для соответствующей ячейки хеш-таблицы, но при этом необходимо модифицировать процедуру поиска существующего элемента так, чтобы она считала удалённые ячейки занятыми, а при добавлении – свободными и сбрасывала значение флага.

Алгоритмы хеширования

Существует несколько типов функций хеширования, каждая из которых имеет свои преимущества и недостатки и основана на представлении данных. Приведем обзор и анализ некоторых наиболее простых из применяемых на практике хеш-функций.

Таблица прямого доступа. Простой организацией таблицы, обеспечивающей быстрый поиск, является таблица прямого доступа. В такой таблице ключ является адресом записи в таблице или может быть преобразован в адрес, причём таким образом, что никакие два разных ключа не преобразуются в один и тот же адрес. При создании таблицы выделяется память для хранения таблицы и заполняется пустыми записями. Затем записи вносятся в таблицу – каждая на свое место, определяемое ее ключом. При поиске ключ используется как адрес и по этому адресу выбирается запись. Если выбранная запись пуста, то записи с таким ключом нет в таблице. Таблицы прямого доступа эффективны в использовании, но область их применения ограничена.

Назовем пространством ключей множество всех теоретически возможных значений ключей записи. Назовем пространством записей множество тех ячеек памяти, которые выделяются для хранения таблицы. Таблицы прямого доступа применимы только для задач, в которых размер пространства записей может быть равен размеру пространства ключей. В большинстве задач размер пространства записей много меньше, чем пространства ключей. Так, если в качестве ключа используется фамилия, то, даже ограничив длину ключа десятью символами кириллицы, получаем 3310 возможных значений ключей. Если ресурсы вычислительной системы и позволят выделить пространство записей такого размера, то значительная часть этого пространства будет заполнена пустыми записями, т.к. в каждом конкретном заполнении таблицы фактическое множество ключей не будет полностью покрывать пространство ключей.

В целях экономии памяти можно назначать размер пространства записей равным размеру фактического множества записей или превосходящим его незначительно. В этом случае необходимо иметь некоторую функцию, обеспечивающую отображение точки из пространства ключей в точку в пространстве записей, то есть, преобразование ключа в адрес записи:

$$a = h(k),$$

где a – адрес, а k – ключ.

Идеальной хеш-функцией является инъективная функция, которая для любых двух неодинаковых ключей дает неодинаковые адреса.

Метод остатков от деления. Простейшей хеш-функцией является деление по модулю числового значения ключа `Key` на размер пространства записи `HashTableSize`. Результат интерпретируется как адрес записи. Следует иметь в виду, что такая функция хорошо соответствует первому, но плохо – последним трем требованиям к хеш-функции и сама по себе может быть применена лишь в очень ограниченном диапазоне реальных задач. Однако операция деления по модулю обычно применяется как последний шаг в более сложных функциях хеширования, обеспечивая приведение результата к размеру пространства записей.

Если ключей меньше, чем элементов массива, то в качестве хеш-функции можно использовать деление по модулю, то есть остаток от деления целочисленного ключа `Key` на размерность массива `HashTableSize`: `Key % HashTableSize`.

Данная функция очень проста, хотя и не относится к хорошим. Вообще, можно использовать любую размерность массива, но она должна быть такой, чтобы минимизировать число коллизий. Для этого в качестве размерности лучше использовать простое число. В большинстве случаев подобный выбор вполне удовлетворителен. Для символьной строки ключом может являться остаток от деления, например, суммы кодов символов строки на HashTableSize.

На практике, метод деления – самый распространенный.

```
// функция создания хеш-таблицы метод деления по модулю
int Hash(int Key, int HashTableSize) {

      return Key % HashTableSize;

}
```

Метод функции середины квадрата. Следующей хеш-функцией является функция середины квадрата. Значение ключа преобразуется в число, это число затем возводится в квадрат, из него выбираются несколько средних цифр и интерпретируются как адрес записи.

Метод свертки. Еще одной хеш-функцией можно назвать функцию свертки. Цифровое представление ключа разбивается на части, каждая из которых имеет длину, равную длине требуемого адреса. Над частями производятся определенные арифметические или поразрядные логические операции, результат которых интерпретируется как адрес. Например, для сравнительно небольших таблиц с ключами – символьными строками неплохие результаты дает функция хеширования, в которой адрес записи получается в результате сложения кодов символов, составляющих строку-ключ.

В качестве хеш-функции также применяют функцию преобразования системы счисления. Ключ, записанный как число в некоторой системе счисления P, интерпретируется как число в системе счисления $Q > P$. Обычно выбирают $Q = P + 1$. Это число переводится из системы Q обратно в систему P, приводится к размеру пространства записей и интерпретируется как адрес.

Открытое хеширование. Основная идея базовой структуры при открытом (внешнем) хешировании заключается в том, что потенциальное множество (возможно, бесконечное) разбивается на конечное число классов. Для B классов, пронумерованных от 0 до $B - 1$, строится хеш-функция $h(x)$ такая, что для любого элемента x исходного множества функция $h(x)$ принимает целочисленное значение из интервала $0, 1, \ldots, B - 1$, соответствующее классу, которому принадлежит элемент x. Часто классы называют сегментами, поэтому будем говорить, что элемент x принадлежит сегменту $h(x)$. Массив, называемый таблицей сегментов и проиндексированный номерами сегментов $0, 1, \ldots, B - 1$, содержит заголовки для B списков. Элемент x, относящийся к i-му списку – это элемент исходного множества, для которого $h(x) = i$.

Если сегменты примерно одинаковы по размеру, то списки всех сегментов должны быть наиболее короткими при данном числе сегментов. Если исходное множество состоит из N элементов, тогда средняя длина списков будет N/B. Если можно оценить величину N и выбрать B как можно ближе к этой величине, то в каждом списке будет один или два элемента. Тогда время выполнения операторов словарей будет малой постоянной величиной, не зависящей от N.

Пример 1. Программная реализация открытого хеширования.

```
#include <iostream>
#include <fstream>

using namespace std;

typedef int T;                      // тип элементов
typedef int hashTableIndex;         // индекс в хеш-таблице

#define compEQ(a,b) (a == b)

typedef struct Node_ {
	T data;              // данные, хранящиеся в вершине
	struct Node_ *next;  // следующая вершина
} Node;


// хеш-функция размещения вершины
hashTableIndex myhash(T data) {
	return (data % hashTableSize);
}

// функция поиска местоположения и вставки вершины в таблицу
Node *insertNode(T data) {
	Node *p, *p0;
	hashTableIndex bucket;

	// вставка вершины в начало списка
	bucket = myhash(data);

	if ((p = new Node) == 0) {
		fprintf(stderr, "Нехватка памяти (insertNode)\n");
		exit(1);
	}

	p0 = hashTable[bucket];
	hashTable[bucket] = p;
	p->next = p0;
	p->data = data;

	return p;
}

//функция удаления вершины из таблицы
void deleteNode(T data) {
	Node *p0, *p;
	hashTableIndex bucket;

	p0 = 0;
	bucket = myhash(data);
	p = hashTable[bucket];

	while (p && !compEQ(p->data, data)) {
		p0 = p;
		p = p->next;
	}

	if (!p) return;

	if (p0) p0->next = p->next;
	else hashTable[bucket] = p->next;

	free(p);
}
```

```
// функция поиска вершины со значением data
Node *findNode(T data) {
	Node *p;

	p = hashTable[myhash(data)];

	while (p && !compEQ(p->data, data)) p = p->next;

	return p;
}

int main() {
	int i, *a, maxnum;

	cout << "Введите количество элементов maxnum : ";
	cin >> maxnum;

	cout << "Введите размер хеш-таблицы HashTableSize : ";
	cin >> hashTableSize;

	a = new int[maxnum];
	hashTable = new Node*[hashTableSize];

	for (i = 0; i < hashTableSize; i++) hashTable[i] = NULL;

	// генерация массива
	for (i = 0; i < maxnum; i++) a[i] = rand();

	// заполнение хеш-таблицы элементами массива
	for (i = 0; i < maxnum; i++) insertNode(a[i]);

	// поиск элементов массива по хеш-таблице
	for (i = maxnum - 1; i >= 0; i--) findNode(a[i]);

	// вывод элементов массива в файл List.txt
	ofstream out("List.txt");
	for (i = 0; i < maxnum; i++) {
		out << a[i];
		if (i < maxnum - 1) out << "\t";
	}

	out.close();

	// сохранение хеш-таблицы в файл HashTable.txt
	out.open("HashTable.txt");
	for (i = 0; i < hashTableSize; i++) {
		out << i << "  :  ";

		Node *Temp = hashTable[i];

		while (Temp) {
			out << Temp->data << " -> ";
			Temp = Temp->next;
		}
		out << endl;
	}
	out.close();

	// очистка хеш-таблицы
	for (i = maxnum - 1; i >= 0; i--) deleteNode(a[i]);

	return 0;
}
```

Закрытое хеширование. При закрытом (внутреннем) хешировании в хеш-таблице хранятся непосредственно сами элементы, а не заголовки списков элементов. Поэтому в каждой записи (сегменте) может храниться только один элемент. При закрытом хешировании применяется методика повторного хеширования. Если осуществляется попытка поместить элемент x в сегмент с номером $h(x)$, который уже занят другим элементом (коллизия), то в соответствии с методикой повторного хеширования выбирается последовательность других номеров сегментов h1(x),h2(x),..., куда можно поместить элемент x. Каждое из этих местоположений последовательно проверяется, пока не будет найдено свободное. Если свободных сегментов нет, то, следовательно, таблица заполнена, и элемент x добавить нельзя.

При поиске элемента x необходимо просмотреть все местоположения h(x),h1(x),h2(x),..., пока не будет найден x или пока не встретится пустой сегмент. Чтобы объяснить, почему можно остановить поиск при достижении пустого сегмента, предположим, что в хеш-таблице не допускается удаление элементов. Пусть $h_3(x)$ – первый пустой сегмент. В такой ситуации невозможно нахождение элемента x в сегментах h4(x),h5(x) и далее, так как при вставке элемент x вставляется в первый пустой сегмент, следовательно, он находится где-то до сегмента h3(x). Но если в хеш-таблице допускается удаление элементов, то при достижении пустого сегмента, не найдя элемента x, нельзя быть уверенным в том, что его вообще нет в таблице, так как сегмент может стать пустым уже после вставки элемента x. Поэтому, чтобы увеличить эффективность данной реализации, необходимо в сегмент, который освободился после операции удаления элемента, поместить специальную константу, которую назовем, например, DEL. В качестве альтернативы специальной константе можно использовать дополнительное поле таблицы, которое показывает состояние элемента. Важно различать константы DEL и NULL – последняя находится в сегментах, которые никогда не содержали элементов. При таком подходе выполнение поиска элемента не требует просмотра всей хеш-таблицы. Кроме того, при вставке элементов сегменты, помеченные константой DEL, можно трактовать как свободные, таким образом, пространство, освобожденное после удаления элементов, можно рано или поздно использовать повторно. Но если невозможно непосредственно сразу после удаления элементов пометить освободившиеся сегменты, то следует предпочесть закрытому хешированию схему открытого хеширования.

Существует несколько методов повторного хеширования, то есть определения местоположений $h(x)$, $h1(x)$, $h2(x)$, …:

- линейное опробование;
- квадратичное опробование;
- двойное хеширование.

Линейное опробование сводится к последовательному перебору сегментов таблицы с некоторым фиксированным шагом:

$$\text{адрес} = h(x) + c \cdot i,$$

где i – номер попытки разрешить коллизию,

c – константа, определяющая шаг перебора.

При шаге, равном единице, происходит последовательный перебор всех сегментов после текущего. Квадратичное опробование отличается от линейного тем, что шаг перебора сегментов нелинейно зависит от номера попытки найти свободный сегмент:

$$\text{адрес} = h(x) + c \cdot i + d \cdot i^2,$$

где i – номер попытки разрешить коллизию, а c и d – константы.

Благодаря нелинейности такой адресации уменьшается число проб при большом числе ключей-синонимов. Однако даже относительно небольшое число проб может быстро привести к выходу за адресное пространство небольшой таблицы вследствие квадратичной зависимости адреса от номера попытки.

Еще одна разновидность метода открытой адресации, которая называется двойным хешированием, основана на нелинейной адресации, достигаемой за счет суммирования значений основной и дополнительной хеш-функций:

$$\text{адрес} = h(x) + i \cdot h2(x).$$

Очевидно, что по мере заполнения хеш-таблицы будут происходить коллизии, и в результате их разрешения очередной адрес может выйти за пределы адресного пространства таблицы. Чтобы это явление происходило реже, можно пойти на увеличение длины таблицы по сравнению с диапазоном адресов, выдаваемым хеш-функцией. С одной стороны, это приведет к сокращению числа коллизий и ускорению работы с хеш-таблицей, а с другой – к нерациональному расходованию памяти. Даже при увеличении длины таблицы в два раза по сравнению с областью значений хеш-функции нет гарантии того, что в результате коллизий адрес не превысит длину таблицы. При этом в начальной части таблицы может оставаться достаточно свободных сегментов. Поэтому на практике используют циклический переход к началу таблицы.

Однако в случае многократного превышения адресного пространства и, соответственно, многократного циклического перехода к началу будет происходить просмотр одних и тех же ранее занятых сегментов, тогда как между ними могут быть еще свободные сегменты. Более корректным будет использование сдвига адреса на 1 в случае каждого циклического перехода к началу таблицы. Это повышает вероятность нахождения свободных сегментов.

В случае применения схемы закрытого хеширования скорость выполнения вставки и других операций зависит не только от равномерности распределения элементов по сегментам хеш-функцией, но и от выбранной методики повторного хеширования (опробования) для разрешения коллизий, связанных с попытками вставки элементов в уже заполненные сегменты. Например, методика линейного опробования для разрешения коллизий – не лучший выбор.

Как только несколько последовательных сегментов будут заполнены, образуя группу, любой новый элемент при попытке вставки в эти сегменты будет вставлен в конец этой группы, увеличивая тем самым длину группы последовательно заполненных сегментов. Другими словами, для поиска пустого сегмента в случае непрерывного расположения заполненных сегментов необходимо просмотреть больше сегментов, чем при случайном распределении заполненных сегментов. Отсюда также следует очевидный вывод, что при непрерывном расположении заполненных сегментов увеличивается время выполнения вставки нового элемента и других операций.

Пример 2. Программная реализация закрытого хеширования.

```
#include <iostream>
#include <fstream>

using namespace std;

typedef int T;                    // тип элементов
typedef int hashTableIndex;       // индекс в хеш-таблице
int hashTableSize;
T *hashTable;
bool *used;

// хеш-функция размещения величины
hashTableIndex myhash(T data) {

	return (data % hashTableSize);
}

// функция поиска местоположения и вставки величины в таблицу
void insertData(T data) {
	hashTableIndex bucket;

	bucket = myhash(data);

	while (used[bucket] && hashTable[bucket] != data)
		bucket = (bucket + 1) % hashTableSize;

	if (!used[bucket]) {
		used[bucket] = true;
		hashTable[bucket] = data;
	}
}

//функция удаления величины из таблицы
void deleteData(T data) {
	int bucket, gap;

	bucket = myhash(data);

	while (used[bucket] && hashTable[bucket] != data)
		bucket = (bucket + 1) % hashTableSize;

	if (used[bucket] && hashTable[bucket] == data) {

		used[bucket] = false;
		gap = bucket;
		bucket = (bucket + 1) % hashTableSize;

		while (used[bucket]) {
			if (bucket == myhash(hashTable[bucket])) {
				bucket = (bucket + 1) % hashTableSize;
			} else if (dist(myhash(hashTable[bucket]),bucket) < dist(gap,bucket)) {
				bucket = (bucket + 1) % hashTableSize;
			} else {
				used[gap] = true;
				hashTable[gap] = hashTable[bucket];
				used[bucket] = false;
				gap = bucket;
				bucket++;
			}
		}
	}
}
```

```
// функция поиска величины, равной data
bool findData(T data) {
	hashTableIndex bucket;

	bucket = myhash(data);

	while (used[bucket] && hashTable[bucket] != data)
		bucket = (bucket + 1) % hashTableSize;

	return used[bucket] && hashTable[bucket] == data;
}

// функция вычисления расстояние от a до b (по часовой стрелке, слева направо)
int dist(hashTableIndex a, hashTableIndex b) {

	return (b - a + hashTableSize) % hashTableSize;
}

int main() {
	int i, *a, maxnum;

	cout << "Введите количество элементов maxnum : ";
	cin >> maxnum;

	cout << "Введите размер хеш-таблицы hashTableSize : ";
	cin >> hashTableSize;

	a = new int[maxnum];
	hashTable = new T[hashTableSize];
	used = new bool[hashTableSize];

	for (i = 0; i < hashTableSize; i++) {
		hashTable[i] = 0;
		used[i] = false;
	}

	// генерация массива
	for (i = 0; i < maxnum; i++) a[i] = rand();

	// заполнение хеш-таблицы элементами массива
	for (i = 0; i < maxnum; i++) insertData(a[i]);

	// поиск элементов массива по хеш-таблице
	for (i = maxnum - 1; i >= 0; i--) findData(a[i]);

	// вывод элементов массива в файл List.txt
	ofstream out("List.txt");
	for (i = 0; i < maxnum; i++) {
		out << a[i];
		if (i < maxnum - 1) out << "\t";
	}
	out.close();

	// сохранение хеш-таблицы в файл HashTable.txt
	out.open("HashTable.txt");
	for (i = 0; i < hashTableSize; i++) {
		out << i << "  :  " << used[i] << " : " << hashTable[i] << endl;
	}
	out.close();

	// очистка хеш-таблицы
	for (i = maxnum - 1; i >= 0; i--) deleteData(a[i]);

	return 0;
}
```

До сих пор рассматривались способы поиска в таблице по ключам, позволяющим однозначно идентифицировать запись. Такие ключи называются первичными. Возможен вариант организации таблицы, при котором отдельный ключ не позволяет однозначно идентифицировать запись. Такая ситуация часто встречается в базах данных. Идентификация записи осуществляется по некоторой совокупности ключей. Ключи, не позволяющие однозначно идентифицировать запись в таблице, называются вторичными ключами. Даже при наличии первичного ключа, для поиска записи могут быть использованы вторичные.

Задания к лабораторной работе

Вариант	Задание
1	Задан набор записей следующей структуры: табельный номер, ФИО, заработная плата. По табельному номеру найти остальные сведения.
2	Задан набор записей следующей структуры: номер телефона, ФИО, адрес. По номеру телефона найти сведения о ФИО владельца и его адресе.
3	Задан набор записей следующей структуры: фамилия автора, название книги и год издания. По названию книги найти всю остальную информацию.
4	Задан набор записей следующей структуры: номер автомобиля, его марка и ФИО владельца. По номеру автомобиля найти остальные сведения.
5	Задан набор записей следующей структуры: фамилия и инициалы, дата рождения, месяц рождения, год рождения. По дате рождения найти фамилию и инициалы.
6	Задан набор записей следующей структуры: название команды, количество набранных очков. По количеству набранных очков определить название команды.
7	Задан набор записей следующей структуры: фамилия, улица, дом, квартира. По заданному адресу найти ФИО владельца.
8	Задан набор записей следующей структуры: город, улица, дом, квартира, кому, ценность посылки. По заданному адресу найти всю остальную информацию.
9	Задан набор записей следующей структуры: предмет, номер группы, дата экзамена. По дате проведения экзамена найти всю остальную информацию.
10	Задан набор записей следующей структуры: наименование товара, старая цена, новая цена. По новой цене найти всю остальную информацию.