Лабораторная работа №7. Программирование с использованием хеширования

Теория.

Понятие хеширования

До сих пор среди данных большого объема остается проблемой поиск необходимого элемента. Если эти данные расположены беспорядочно в массиве (линейном списке, файле и т.п.), то осуществляют линейный поиск, эффективность которого 0(n/2). Если же данные в массиве упорядочены (например, сбалансированное двоичное дерево), то возможен двоичный поиск с эффективностью 0(log2n).

Однако при работе с двоичным деревом и упорядоченным массивом затруднены операции вставки и удаления элементов, так, например, дерево – разбалансируется, и вновь требуется его балансировка. Что можно придумать в данном случае более эффективное?

В данном случае был предложен алгоритм хеширования (hashing – перемешивание), при котором создаются ключи, определяющие данные массива и на их основании данные записываются в таблицу, названную хеш-таблицей. Ключи для записи определяются при помощи функции i = h(key), называемой хеш-функцией. Алгоритм хеширования определяет положение искомого элемента в хеш-таблице по значению его ключа, полученного хеш-функцией.

Хеширование – это способ сведения хранения большого множества к более меньшему.

Возьмем, например, словарь или энциклопедию. В этом случае буквы алфавита могут быть приняты за ключи поиска, т.е. основным элементом алгоритма хеширования является ключ (key)! В большинстве приложений ключ обеспечивает косвенную ссылку на данные.

Термин «хеширование» в литературе по программированию появился не так давно – в 1967 году, ввел его Хеллерман (Hellerman). Дословный перевод означает – рубить, крошить, но академик А.П.Ершов предложил довольно удачный эквивалент – «расстановка».

Фактически хеширование – это специальный метод адресации данных для быстрого поиска нужной информации по ключам.

Или хеширование – это разбиение общего (базового) набора уникальных ключей элементов данных на непересекающиеся наборы с определенным свойством.

Если базовый набор содержит N элементов, то его можно разбить на 2N различных подмножеств.

Хеш-функция и хеш-таблица

Функция, которая описывает определенное свойство подмножеств, т.е. преобразует ключ элемента данных в некоторый индекс в таблице (хеш-таблица), называется функцией хеширования или хеш-функцией:

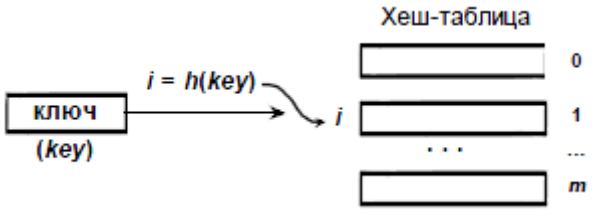
i = h(key);

где key – преобразуемый ключ, i – получаемый индекс таблицы, т.е. ключ отображается во множестве, например, целых чисел (хеш-адреса), которые впоследствии используются для доступа к данным.

Однако функция расстановки может для нескольких значений ключа давать одинаковое значение позиции i в таблице. Ситуация, при которой два или более ключа получают один и тот же индекс (хеш-адрес) называется коллизией при хешировании.

Хорошей хеш-функцией считается такая функция, которая минимизирует коллизии и распределяет данные равномерно по всей таблице.

Совершенная хеш-функция – это функция, которая не порождает коллизий:



Разрешить коллизии при хешировании можно двумя методами:

– методом открытой адресации с линейным опробыванием;

– методом цепочек.

Хеш-таблица

Хеш-таблица представляет собой обычный массив с необычной адресацией, задаваемой хеш-функцией.

Хеш-структуру считают обобщением массива, который обеспечивает быстрый прямой доступ к данным по индексу. С точки зрения хеширования, массив задает отображение A множества индексов I на множество элементов E, т.е. A:I→E и позволяет по индексу быстро найти нужный элемент.

Имеется множество схем хеширования, различающихся как выбором удачной функции h(key), так и алгоритма разрешения конфликтов. Эффективность решения реальной практической задачи будет существенно зависеть от выбираемой стратегии.

Примеры хеш-функций

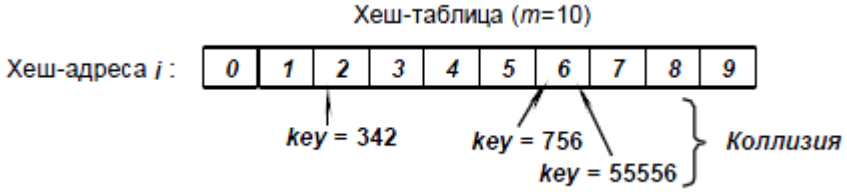
Функция хеширования является важной частью процесса хеширования. Она используется для преобразования ключей в адреса таблицы. Она должна легко вычисляться и преобразовывать ключи (обычно целочисленные или строковые значения) в целые числа в интервале до m – максимальный размер формируемой таблицы.

Выбираемая функция должна создавать как можно меньше коллизий, т.е. должна равномерно распределять ключи на имеющиеся индексы в таблице. Конечно, нельзя определить, будет ли некоторая конкретная хеш-функция распределять ключи правильно, если эти ключи заранее не известны. Однако, хотя до выбора хеш-функции редко известны сами ключи, некоторые свойства этих ключей, которые влияют на их распределение, обычно известны. Рассмотрим наиболее распространенные методы задания хеш-функции.

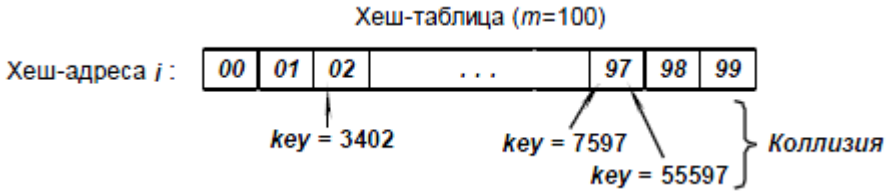
Метод деления. Исходными данными являются – некоторый целый ключ key и размер таблицы m. Результатом данной функции является остаток от деления этого ключа на размер таблицы. Общий вид функции:

```
int h(int key, int m) {
        return key % m; // Значения
}
```

Для m = 10 хеш-функция возвращает младшую цифру ключа.



Для m = 100 хеш-функция возвращает две младших цифры ключа.



Аддитивный метод, в котором ключом является символьная строка C++. В хеш-функции строка преобразуется в целое, суммированием всех символов и возвращается остаток от деления на m (обычно размер таблицы m=256).

```
int h(char *key, int m) {
    int s = 0;
    while(*key)
    s += *key++;
    return s % m;
    }
```

Коллизии возникают в строках, состоящих из одинакового набора символов, например, abc и cab.

Данный метод можно несколько модифицировать, получая результат, суммируя только первый и последний символы строки-ключа.

```
int h(char *key, int m) {
    int len = strlen(key), s = 0;
    if(len < 2) // Если длина ключа равна 0 или 1,
    s = key[0]; // возвратить key[0]
    else
    s = key[0] + key[len–1];
    return s % m;
    }
```

В этом случае коллизии будут возникать только в строках, например, abc и amc.

Метод середины квадрата, в котором ключ возводится в квадрат (умножается сам на себя) и в качестве индекса используются несколько средних цифр полученного значения.

Например, ключом является целое 32-битное число, а хеш-функция возвращает средние 10 бит его квадрата:

```
int h(int key) {
    key *= key;
    key >>= 11; // Отбрасываем 11 младших бит
    return key % 1024; // Возвращаем 10 младших бит
    }
```

Метод исключающего ИЛИ для ключей-строк (обычно размер таблицы m=256). Этот метод аналогичен аддитивному, но в нем различаются схожие слова.

Метод заключается в том, что к элементам строки последовательно применяется операция «исключающее ИЛИ».

В мультипликативном методе дополнительно используется случайное действительное число r из интервала [0,1[, тогда дробная часть произведения r*key будет находиться в интервале [0,1]. Если это произведение умножить на размер таблицы m, то целая часть полученного произведения даст значение в диапазоне от 0 до m-1.

```
int h(int key, int m) {
    double r = key * rnd();
    r = r - (int)r; // Выделили дробную часть
    return r * m;
    }
```

В общем случае при больших значениях m индексы, формируемые хеш-функцией, имеют большой разброс. Более того, математическая теория утверждает, что распределение получается более равномерным, если m является простым числом.

В рассмотренных примерах хеш-функция i = h(key) только определяет позицию, начиная с которой нужно искать (или первоначально – поместить в таблицу) запись с ключом key. Поэтому схема хеширования должна включать алгоритм решения конфликтов, определяющий порядок действий, если позиция i = h(key) оказывается уже занятой записью с другим ключом.

Схемы хеширования

В большинстве задач два и более ключей хешируются одинаково, но они не могут занимать в хеш-таблице одну и ту же ячейку. Существуют два возможных варианта: либо найти для нового ключа другую позицию, либо создать для каждого индекса хеш-таблицы отдельный список, в который помещаются все ключи, отображающиеся в этот индекс.

Эти варианты и представляют собой две классические схемы хеширования:

– хеширование методом цепочек (со списками), или так называемое, многомерное хеширование – chaining with separate lists;

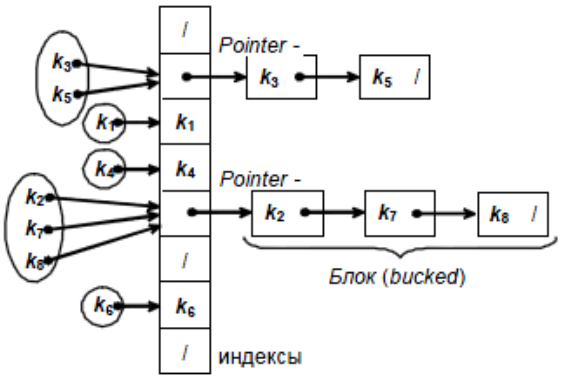
– хеширование методом открытой адресацией с линейным опробыванием – linear probe open addressing.

Метод открытой адресацией с линейным опробыванием. Изначально все ячейки хеш-таблицы, которая является обычным одномерным массивом, помечены как не занятые. Поэтому при добавлении нового ключа проверяется, занята ли данная ячейка. Если ячейка занята, то алгоритм осуществляет осмотр по кругу до тех пор, пока не найдется свободное место («открытый адрес»).

Т.е. либо элементы с однородными ключами размещают вблизи полученного индекса, либо осуществляют двойное хеширование, используя для этого разные, но взаимосвязанные хеш-функции.

В дальнейшем, осуществляя поиск, сначала находят по ключу позицию i в таблице, и, если ключ не совпадает, то последующий поиск осуществляется в соответствии с алгоритмом разрешения конфликтов, начиная с позиции i по списку.

Метод цепочек является доминирующей стратегией. В этом случае i, полученной из выбранной хеш-функцией h(key)=i, трактуется как индекс в хеш-таблице списков, т.е. сначала ключ key очередной записи отображается на позицию i = h(key) таблицы. Если позиция свободна, то в нее размещается элемент с ключом key, если же она занята, то отрабатывается алгоритм разрешения конфликтов, в результате которого такие ключи помещаются в список, начинающийся в i-той ячейке хеш-таблицы. Например



В итоге имеем таблицу массива связных списков или деревьев.

Процесс заполнения (считывания) хеш-таблицы прост, но доступ к элементам требует выполнения следующих операций:

– вычисление индекса i;

– поиск в соответствующей цепочке.

Для улучшения поиска при добавлении нового элемента можно использовать алгоритма вставки не в конец списка, а – с упорядочиванием, т.е. добавлять элемент в нужное место.

При решении задач на практике необходимо подобрать хеш-функцию i = h(key), которая по возможности равномерно отображает значения ключа key на интервал [0, m-1], m – размер хеш-таблицы. И чаще всего, если нет информации о вероятности распределения ключей по записям, используя метод деления, берут хеш-функцию i = h(key) = key%m.

При решении обратной задачи – доступ (поиск) к определенному подмножеству возможен из хеш-таблицы (хеш-структуры), которая обеспечивает по хеш-адресу (индексу) быстрый доступ к нужному элементу.

Пример реализации метода прямой адресации с линейным опробыванием. Исходными данными являются 7 записей (для простоты информационная часть состоит только из целочисленных данных), объявленного структурного типа:

```
struct zap {
    int key; // Ключ
    int info; // Информация
    } data;
```

{59,1}, {70,3}, {96,5}, {81,7}, {13,8}, {41,2}, {79,9}; размер хеш-таблицы m=10.

Хеш-функция i = h(data) = data.key%10; т.е. остаток от деления на 10 – i∈[0,9].

На основании исходных данных последовательно заполняем хеш-таблицу.

Хеш-таблица (*m*=10)

Хеш-адреса *i* :	0	1	2	3	4	5	6	7	8	9
key :	*70*	*81*	*41*	*13*	*79*		*96*			*59*
info :	*3*	*7*	*2*	*8*	*9*		*5*			*1*
проба :	1	1	2	1	6		1			1

Хеширование первых пяти ключей дает различные индексы (хеш-адреса):

i = 59 % 10 = 9;

i = 70 % 10 = 0;

i = 96 % 10 = 6;

i = 81 % 10 = 1;

i = 13 % 10 = 3.

Первая коллизия возникает между ключами 81 и 41 – место с индексом 1 занято. Поэтому просматриваем хеш-таблицу с целью поиска ближайшего свободного места, в данном случае – это i = 2.

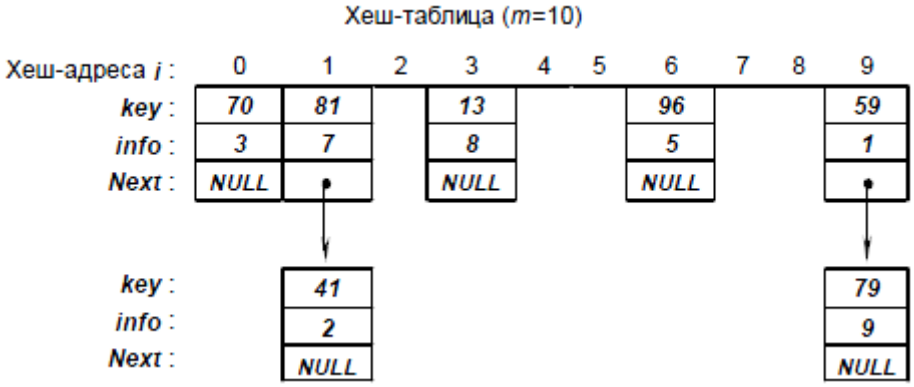
Следующий ключ 79 также порождает коллизию: позиция 9 уже занята. Эффективность алгоритма резко падает, т.к. для поиска свободного места понадобилось 6 проб (сравнений), свободным оказался индекс i = 4. Общее число проб – 1,9 пробы на элемент.

Реализация метода цепочек для предыдущего примера. Объявляем структурный тип для элемента списка (однонаправленного):

```
struct zap {
    int key; // Ключ
    int info; // Информация
```

zap *Next; // Указатель на следующий элемент в списке
} data;

На основании исходных данных последовательно заполняем хеш-таблицу, добавляя новый элемент в конец списка, если место уже занято.



Хеширование первых пяти ключей, как и в предыдущем случае, дает различные индексы (хеш-адреса): 9, 0, 6, 1, и 3.

При возникновении коллизии, новый элемент добавляется в конец списка. Поэтому элемент с ключом 41, помещается после элемента с ключом 81, а элемент с ключом 79 – после элемента с ключом 59.

Задание.

Разработать приложение, в котором содержатся следующие классы:

Родительский класс, реализующий методы работы с хеш-таблицей на основе массива стеков **(не использовать шаблоны STL)**.

Производный класс, созданный на базе родительского и реализующий метод решения своего варианта.

В приложении продемонстрировать работу всех методов работы с хеш-таблицей. Результат формирования и преобразования хеш-таблицы показывать в компоненте TMemo методом Print(TMemo) в виде строк, отображающих стеки.

Написать обработчик события, реализующий вызов метода решения своего варианта.

Индивидуальные задания

1. Создать хеш-таблицу со случайными целыми ключами в диапазоне от–50 до +50 и преобразовать ее в две таблицы. Первая должна содержать только положительные ключи, а вторая – отрицательные.
2. Создать хеш-таблицу со случайными целыми ключами и удалить из него записи с четными ключами.
3. Создать хеш-таблицу со случайными целыми ключами в диапазоне от –10 до 10 и удалить из него записи с отрицательными ключами.
4. Создать хеш-таблицу со случайными целыми ключами и найти запись с минимальным ключом.
5. Создать хеш-таблицу со случайными целыми ключами и найти запись с максимальным ключом.
6. Подсчитать, сколько элементов хеш-таблицы со случайными ключами превышает среднее значение от всех ключей.
7. Создать хеш-таблицу из случайных целых чисел и найти в ней номер стека, содержащего минимальное значение ключа.
8. Создать хеш-таблицу из случайных целых чисел и найти в ней номер стека, содержащего максимальное значение ключа.

9. Создать хеш-таблицу со случайными ключами и распечатать все элементы в порядке возрастания ключа.

10. Создать хеш-таблицу со случайными ключами и распечатать все элементы в порядке убывания ключа

11. Создать хеш-таблицу со случайными ключами и определить, сколько элементов находится в каждом стеке.

12. Подсчитать, сколько элементов хеш-таблицы со случайными ключами не превышает среднее значение от всех ключей.

13. Создать хеш-таблицу из случайных целых чисел и из нее сделать еще две. В первую поместить записи с ключами большими K, а во второй – с меньшими K.

14. Создать хеш-таблицу со случайными ключами в диапазоне от 1 до 10 и определить наиболее часто повторяющийся ключ.

15. Создать хеш-таблицу со случайными ключами и удалить из нее записи с ключами из диапазона $K1 < key < K2$.