Ответы на вопросы по инфопроге

Никитенко Яна

June 2024

1 Понятие Θ, Ω -нотации. Оценка сложности алгоритма.

Для оценки эффективности в середине 50-ых годов 20 века было введено понятие RAM-машины. Это некий гипотетический компьютер (Random Access Machine), т. е. машина с произвольным доступом к памяти. Обладает следующими свойствами:

1. Для исполнения простой операции ($+$, '$-$', $*$, $=$, if) требуется ровно один временной шаг.

2. Каждое обращение к памяти занимает один временной шаг. При этом неограничен объем оперативной памяти.

3. Цикл и подпрограмма состоят из нескольких простых операций. Время исполнения зависит от количества итераций или характеристик подпрограммы.

Естественно, это очень упрощенная абстракция работы компьютера, однако она позволяет оценивать основное время работы алгоритма. Время работы алгоритма оценивается в шагах, необходимых для решения задачи. Для того, чтобы говорить о том, плохим или хорошим является алгоритм, нужно знать как он работает для всех экземпляров задачи. Разделяют наилучший, наихудший и средний случай сложности алгоритмов. Проще всего объяснить на примере алгоритма сортировки.

Сложность алгоритма в наихудшем случае — функция, определяемая максимальным количеством шагов, необходимым для обработки любого экземпляра задачи. Для сортировок — количество шагов, необходимое для сортировки отсортированной в обратном порядке последовательности.

Сложность алгоритма в наилучшем случае — функция, определяемая минимальным количеством шагов, необходимым для обработки любого экземпляра задачи. Для сортировок — количество шагов, необходимое для сортировки уже отсортированной последовательности.

Сложность алгоритма в среднем случае — функция, определяемая средним количеством шагов, необходимым для обработки всех экземпляров задачи. Для сортировок — количество шагов, необходимое для сортировки случайной последовательности.

Другие обозначения сложности алгоритмов. Кроме обозначения "Big O существуют другие обозначения для оценки сложности алгоритмов.

Вот несколько наиболее распространенных обозначений:

Big Theta (Θ): Big Theta также оценивает верхнюю и нижнюю границы временной сложности алгоритма, но описывает точную сложность, а не только наихудший случай, как Big O. Θ (f(n)) обозначает, что время выполнения алгоритма ограничено функцией f(n) как сверху, так и снизу.

Big Omega (Ω): Big Omega оценивает нижнюю границу временной сложности алгоритма. Θ(f(n)) говорит о том, что алгоритм выполнится не быстрее, чем функция f(n).

Little O (θ): Little O представляет собой верхнюю границу, которая строже, чем Big O. Если f(n) является θ(g(n)), это означает, что время выполнения алгоритма ограничивается функцией g(n), но алгоритм работает быстрее, чем g(n).

Little Omega (ω): Little Omega представляет собой нижнюю границу, которая строже, чем Big Omega. Если f(n) является ω(g(n)), это означает, что алгоритм работает медленнее, чем g(n), но не медленнее, чем f(n).

2 Сортировка. Примеры устойчивой и неустойчивой сортировок.

Сортировка заключается в переупорядочении элементов таким образом, чтобы их ключи следовали в соответствии с четко определенными правилами (например, по возрастанию или в алфавитном порядке). Основной характеристикой алгоритма сортировки является время, затрачиваемое на его выполнение. В принципе, все алгоритмы сортировки можно разделить на три класса:

Алгоритмы, выполняющиеся за время, пропорциональное N^2

Алгоритмы, выполняющиеся за время, пропорциональное N log N.

Алгоритмы, выполняющиеся за линейное время. Но это алгоритмы не основанные на сравнениях и заменах.

Дополнительная память, используемая алгоритмом сортировки, является вторым по важности фактором. Все методы сортировки по этому фактору можно разделить на три класса: Алгоритмы, не требующие дополнительной памяти (кроме, возможно памяти для хранения одного элемента).

Алгоритмы, использующие для доступа к данным указатели или индексы и требующие дополнительной памяти для размещения N указателей или индексов.

Алгоритмы, требующие дополнительной памяти для размещения еще одной копии массива. В случае наличия элементов с несколькими ключами возникает одна из важных характеристик алгоритмов сортировки — устойчивость.

Алгоритм сортировки является устойчивым, если при наличии дублированных ключей, сохраняется относительный порядок размещения. Напри-

мер, есть список студентов, отсортированный по фамилии. В случае сортировки этого списка по годам рождения алгоритмы устойчивой сортировки оставят списки студентов, родившихся в один год, в алфавитном порядке. Еще иногда говорят об эффективной сортировке, т. е. прекращающей свою работу в случае отсортированной последовательности.

3 Сортировки за $O(n^2)$: вставка, пузырьком, выбор.

Сортировка вставками

Можно сказать, что это модификация гномьей сортировки. Гном идет вперед, пока не найдет неправильно стоящие горшки, запоминает на каком горшке он остановился, меняет горшки местами и идет назад, пока не поставит горшок на место. Однако дальше идет не на шаг вперед, а сразу начинает с горшка, который он запомнил. В примере показаны только итерации с обменами. Разными цветами показаны элементы, которые обменяли.

Алгоритм 9: Сортировка вставками
Вход: A - массив размерности N
Выход: Отсортированный массив
начало алгоритма
цикл для i = 1 до N - 1 выполнять
· j = i;
цикл пока j $>$ 0 и a[j] $<$ a[j - 1] выполнять
· меняем их местами;
· уменьшаем j;
конец алгоритма

8	3	9	6	1	2	5
$i = 1$						
3	8	9	6	1	2	5
$i = 2$						
$i = 3$						
3	8	6	9	1	2	5
3	6	8	9	1	2	5
$i = 4$						
3	6	8	1	9	2	5
3	6	1	8	9	2	5
3	1	6	8	9	2	5
1	3	6	8	9	2	5
$i = 5$						
1	3	6	8	2	9	5
1	3	6	2	8	9	5
1	3	2	6	8	9	5
1	2	3	6	8	9	5
$i = 6$						
1	2	3	6	8	5	9
1	2	3	6	5	8	9
1	2	3	5	6	8	9

Оценим сложность. Число обменов и сравнений одинаково.

Рис. 1: Caption

Сортировка Шелла(Усовершенственная сориртировка вставками)

Также как и в случае сортировки расческой Шелл предложил сравнивать не соседние элементы, а стоящие друг от друга на каком-то расстоянии. Это ускоряет достаточно ускоряет сортировку. Сам Шелл предложил следующую последовательность шагов: $\frac{N}{2}$, $\frac{N}{4}$ ..., 1. Оценить сложность достаточно сложно. В худшем случае сортировка дает сложность $O(N^2)$. Впоследствии были получены последовательности шагов, которые позволяют довести сложность до $O(N^{\frac{4}{3}})$

Алгоритм 10: Сортировка Шелла

Вход: A — массив размерности N

Выход: Отсортированный массив

начало алгоритма

· Определяем шаг;

цикл пока step $>$ 1 выполнять

цикл для i = 0 до N - step выполнять

· j = i;

цикл пока j $>=$ 0 и a[j] $>$ a[j + step] выполнять

· меняем их местами;

· уменьшаем j на шаг;

· переход к следующему шагу;

конец алгоритма

В качестве шага выбрано $\frac{N}{2}$. Для лучшего примера рассмотрим массив из 8 элементов: $\{8, 3, 9, 6, 1, 2, 5, 7.\}$:

8	3	9	6	1	2	5	7
$step = 4$							
8	3	9	6	1	2	5	7
1	3	9	6	8	2	5	7
1	2	9	6	8	3	5	7
1	2	5	6	8	3	9	7
$step = 2$							
1	2	5	6	8	3	9	7
1	2	5	3	8	6	9	7
$step = 1$							
1	2	5	3	8	6	9	7
1	2	3	5	8	6	9	7
1	2	3	5	6	8	9	7
1	2	3	5	6	8	7	9
1	2	3	5	6	7	8	9

Рис. 2: Caption

Сортировка пузырьком

Сделаем простейшее улучшение алгоритма: поменяв местами два элемента не возвращаемся назад, а идем дальше. Тогда при одном проходе минимальный элемент окажется на своем месте. Очевидно, что алгоритм будет быстрее. Такая сортировка называется сортировкой пузырьком, (под пузырьком понимается «самый легкий» элемент, который «поднимается» наверх).

Алгоритм 2: Сортировка пузырьком

Вход: A — массив размерности N

Выход: Отсортированный массив

начало алгоритма

цикл для i = 0 до N - 1 выполнять
цикл для j = N - 1 до i выполнять
если $a[j-1] > a[j]$ то · меняем их местами; конец алгоритма

.....

8	3	9	6	1	2	5
$i = 0$						
8	3	9	1	6	2	5
8	3	1	9	6	2	5
8	1	3	9	6	2	5
1	8	3	9	6	2	5
$i = 1$						
1	8	3	9	2	6	5
1	8	3	2	9	6	5
1	8	2	3	9	6	5
1	2	8	3	9	6	5
$i = 2$						
1	**2**	8	3	9	5	6
1	**2**	8	3	5	9	6
1	**2**	3	8	5	9	6
$i = 3$						
1	**2**	**3**	8	5	6	9
1	**2**	**3**	5	8	6	9
$i = 4$						
1	**2**	**3**	**5**	6	8	9
Результат						
1	**2**	**3**	**5**	**6**	**8**	9

Определяем сложность. Оценим отдельно сравнения и обмены.

Рис. 3: Caption

Сортировка выбором

Другой тип сортировки состоит в том, что в подмассиве [i + 1, N] находится минимальный элемент и ставится на i-тую позицию

Бывает как устойчивая, так и неустойчивая сортировка.

Рассмотрим сначала алгоритм неустойчивой сортировки.

Алгоритм 11: Неустойчивая сортировка выбором
Вход: A — массив размерности N
Выход: Отсортированный массив
начало алгоритма
цикл для i = 0 до N выполнять
· находим минимум из элементов массива от [i + 1] до [N - 1];
· меняем местами i-тый и минимальный элементы;
конец алгоритма

В примере различными цветами показаны только обмены. Полужирным шрифтом показаны элементы, которые уже отсортированы.

8	3	9	6	1	2	5
1	3	9	6	8	2	5
1	2	9	6	8	3	5
1	**2**	3	6	8	9	5
1	**2**	**3**	5	8	9	6
1	**2**	**3**	**5**	6	9	8
1	**2**	**3**	**5**	**6**	8	9

Рис. 4: Caption

4 Быстрая сортировка и сортировка слиянием

Рассмотрим теперь идею модификации, которая была предложена Хоаром в 1960 году. Алгоритм, предложенный им, превратил одну из самых медленных сортировок в одну из самых быстрых. Единственное, что она стала неустойчивой, так как происходит обмен не между соседними элементами, а это приводит к неустойчивости. Общая идея алгоритма такова:

1. Выбираем опорный элемент.

2. Меняем местами элементы так, чтобы слева были меньшие опорного, справа — большие.

3. Рекурсивно вызываем функцию для подмассивов.

Рассмотрим данный алгоритм подробнее:

Алгоритм 7: Быстрая сортировка

Вход: A — массив, L — левая граница массива, R— правая граница массива Выход: Отсортированный массив

начало алгоритма

· i = L;

· j = R;

· выбирается опорный элемент;

цикл пока i <= j выполнять

цикл пока a[i] < опорного выполнять

· увеличиваем i;

цикл пока a[j] > опорного выполнять

· уменьшаем j;

если i <= j то

· меняем местами a[i] и a[j];

· уменьшаем j;

· увеличиваем i;

если левый подмассив содержит больше одного элемента то

· вызываем функцию для [L, j];

если правый подмассив содержит больше одного элемента то

· вызываем функцию для [i, R];

конец алгоритма

Быстрая сортировка легко применяется к двусвязным спискам и другим структурам, допускающим проход как в прямом, так и в обратном порядке. Также быстрая сортировка удобна для параллельных алгоритмов.

Идея слияния (англ. merge) используется очень часто. Если есть две отсортированные последовательности, то результатом слияния должна быть отсортированная последовательность. Алгоритм слияния можно объяснить на примере двух колод карт. Обе колоды отсортированы, лежат рубашками вниз. Берется верхние карты из каждой колоды, сравниваются, меньшая откладывается, берется следующая карта из этой колоды. Следовательно, требуется N шагов для алгоритма слияния. В алгоритме используется один массив, и сливаются отсортированные подмассивы [l, m] и [m + 1, r], где l и r - левая и правая границы массива.

Алгоритм 15: Слияние двух отсортированных подмассивов

Вход: A - массив, l - левая граница массива, r - правая граница массива, m - место разбиения на подмассивы.

Подмассивы [l, m] и [m + 1, r] являются отсортированными.

Выход: Отсортированный подмассив [l, r]

начало алгоритма

если l > r или m < l или m > r то

· прекращаем работу алгоритма;

если в массиве 2 элемента и они не отсортированы то

· меняем их местами;

· прекращаем работу алгоритма;

· Создаем дополнительный массив buf размерностью r - l + 1;

· cur = 0;

· i = l, j = m + 1;

цикл пока r - l + 1 6= cur выполнять

если i > m то

· дописываем в buf оставшиеся элементы от j до r ;

иначе если j > r то

· дописываем в buf оставшиеся элементы от i до m ;

иначе если a[i] > a[j] то

· записываем в buf a[j];

· увеличиваем j;

иначе

· записываем в buf a[i];

· увеличиваем i;

· записываем массив buf в начальный массив A, начиная с индекса l;

конец алгоритма

2	5	9		1	3	4		1					
2	5	9		1	3	4		1	2				
2	5	9		1	3	4		1	2	3			
2	5	9		1	3	4		1	2	3	4		
2	5	9		1	3	4		1	2	3	4	5	9

Рис. 5: Caption

Алгоритм 16: Сортировка слиянием

Вход: A — массив, l — левая граница массива, r — правая граница массива

Выход: Отсортированный массив

начало алгоритма

если l > r то

· прекращаем работу алгоритма;

· m $=\frac{l+r}{2}$

· вызываем данную функцию от (A, l, m);

· вызываем данную функцию от (A, m + 1, r);

· вызываем функцию слияния от (A, l, r, m);

конец алгоритма

5 Понятие кучи. Пирамидальная сортировка.

При использовании пирамидальной сортировки представляем массив в виде специальной структуры, называемой кучей (англ. heap).

Куча, или основанная на ее основе структура, называемая очередь с приоритетами, используется во многих алгоритмах.

Куча - это специальное бинарное дерево, обладающее следующими свойствами: 1. Дерево почти полное, то есть, заполнено на всех уровнях, кроме последнего. На последнем уровне элементы заполняются слева направо, пока не закончатся элементы.

2. Значение узла всегда больше значений его потомков. Следовательно, корнем дерева будет будет максимальный элемент.

Как уже говорилось при описании быстрой сортировки, высота почти полного бинарного дерева равна $\log_2 N + 1$. Куча можно описать с помощью массива. Потомками i-того узла являются элементы с индексами 2i + 1 и 2i + 2 (при условии нумерации с нуля).

Алгоритм : «Просеивание» i-ого элемента в куче

Вход: A — массив, i — номер просеиваемого элемента, N — размер массива

Выход: Куча

начало алгоритма

· полагаем, что i-тый элемент является максимальным; цикл пока true выполнять

если ∃ левый ребенок для i-ого элемента то

· находим максимум между ним и родителем;

если ∃ правый ребенок для i-ого элемента то

· находим максимум между ним и родителем;

если максимум — это родитель то

· прекращаем цикл;

иначе

· меняем местами максимум и родителя;

· i = max;

конец алгоритма

Пирамидальная сортировка

На вершине кучи всегда находится максимальный элемент. Следовательно, алгоритм пирамидальной сортировки заключается в следующем: меняем местами первый и последний элемент и перестраиваем пирамиду, уменьшая количество элементов на 1. Алгоритм 14: Пирамидальная сортировка

Вход: A — массив, N — размер массива
Выход: Отсортированный массив
начало алгоритма
цикл для i $=\frac{N}{2}$
до 0 выполнять
· строим пирамиду, вызывая функцию просеивания от (A, i, N);
цикл для i = N - 1 до 1 выполнять
· меняем местами a[0] и a[i];
· перестраиваем пирамиду, вызывая функцию просеивания от (A, 0, i);
конец алгоритма

6 Динамические структуры данных. Примеры работы со стеком.

Все обрабатываемые данные, в конце концов компьютером разбиваются на отдельные биты. Но писать программы для работы с битами достаточно трудоемкое занятие. Поэтому, типы позволяют указать, как будут использоваться определенные наборы битов, функции — задавать операции, определяемые над данными, структуры используются для группировки разнородных частей информации, указатели для непрямой ссылки на информацию.

Динамические структуры данных создаются с помощью операции new, следовательно, располагаются в динамической памяти. Каждый элемент, помимо информационного поля, содержит адрес следующего элемента (ссылку). Элементы не имеют собственного наименования, имеется только указатель head, который расположен в статической памяти, и содержит адрес первого элемента структуры. Ко всем остальным элементам можно обратиться только пройдя последовательно по всем связям.

Стек — частный случай списка. Стек удовлетворяет принципу LIFO — «Last In First Out» (последний зашел — первый вышел). На основе стеков устроены большинство компьютерных операций, в частности, рекурсивные функции основаны на стеках. По своему устройству стек напоминает детскую игрушку — пирамидку. На примере пирамидки понятно, что для того, чтобы добраться до одетого первым элемента, необходимо снять все верхние элементы.

Точно также для стека определены только две операции: добавить элемент в начало стека и извлечь элемент из начала стека. Других операций для стека НЕ ОПРЕДЕЛЕНО

Стек описывается как struct следующим образом:

Листинг 1.1.

```
1 struct stack{
2 int inf;
3
3 stack *next;
4 };
}
```

Рассмотрим функцию добавления элемента в стек. По сложившейся традиции эта функция называется push(). Листинг 1.2.

```
1 void push(stack *&h, int x){
2 stack *r = new stack; //создаем новый элемент
3 r->inf = x; //поле inf = x
4 r->next = h; //следующим элементов является h
5 h = r; //теперь r является головой
6 }
```

Рассмотрим теперь функцию удаления элемента из стека. По сложившийся традиции эта функция называется pop. Для простоты будем не только удалять элемент, но и возвращать его значение. Листинг 1.3.

```
1 int pop (stack *&h){
2 int i = h->inf; //значение первого элемента
3 stack *r = h; //указатель на голову стека
4 h = h->next; //переносим указатель на следующий элемент
5 delete r; //удаляем первый элемент
6 return i; //возвращаем значение
7 }
```

7 Динамические структуры данных. Пример работы с очередью.

Очередь также представляет собой частный случай списка. Элементы очереди устроены по принципу FIFO — First In First Out (Первый зашел — первый вышел). Очередь можно представить как сосуд без дна. С одной стороны заполняется, с другой извлекается:

Очередь описывается как struct следующим образом: Листинг 1.7.

```
1 struct queue {
2 int inf;
3 queue *next;
4 };
```

Рассмотрим функцию добавления элемента в очередь. По сложившейся традиции эта функция называется push(). Листинг 1.8.

```
1  1 void push (queue *&h, *&t, int x){ //вставка элемента в очередь
2  2 queue *r = new queue; //создаем новый элемент
3  3 r->inf = x;
4  11
5  4 r->next = NULL; //всегда последний
6  5 if (!h && !t){ //если очередь пуста
7  6 h = t = r; //это и голова и хвост
8  7 }
9  8 else {
10 9 t->next = r; //r - следующий для хвоста
11 10 t = r; //теперь r - хвост
12 11 }
13 12 }
```

Рассмотрим теперь функцию удаления элемента из головы очереди. По сложившийся традиции эта функция называется pop. Для простоты будем не только удалять элемент, но и возвращать его значение. Листинг 1.9.

```
1  1 int pop (queue *&h, *&t){ //удаление элемента из очереди
2  2 queue *r = h; //создаем указатель на голову
3  3 int i = h->inf; //сохраняем значение головы
4  4 h = h->next; //сдвигаем указатель на следующий элемент
5  5 if (!h) //если удаляем последний элемент из очереди
6  6 t = NULL;
7  7 delete r; //удаляем первый элемент
8  13
9  8 return i;
10 9 }
11 10 }
```

8 Динамические структуры данных. Двусвязный список. Поиск элемента. Вставка элемента в двусвязный список.

Наиболее общий случай связных списков. Каждый элемент списка состоит из трех полей: информационного и двух ссылочных на следующий и предыдущий элементы:

С элементами списка можно выполнять любые действия: добавлять, удалять, просматривать и т. д. Вставка и удаление элемента выполняется за время O(1), так как надо всего лишь поменять значения в ссылочных полях. Список относится к элементам с последовательным доступом, т. е., чтобы просмотреть пятый элемент, необходимо последовательно просмотреть первые четыре элемента. Для перехода к следующему элементу указателю p

присваивается значение p->next. Так как список двусвязный, можно просматривать элементы как в прямом (p = p->next), так и в обратном порядке (p = p->prev).

Список описывается как struct следующим образом: Листинг 1.10.

```
1 struct list {
2 int inf;
3 list *next;
4 list *prev;
5 };
```

Рассмотрим функцию добавления элемента в конец списка. По сложившейся традиции эта функция называется push(). Листинг 1.11.

```
1 void push (list *&h, list *&t, int x){ //вставка элемента в конец
  ↪  списка
2 list *r = new list; //создаем новый элемент
3 r->inf = x;
4 r->next = NULL; //всегда последний
5 if (!h && !t){ //если список пуст
6 r->prev = NULL; //первый элемент
7 h = r; //это голова
8 }
9 else{
10 t->next = r; //r - следующий для хвоста
11 r->prev = t; //хвост - предыдущий для r
12 }
13 t = r; //r теперь хвост
14 }
```

Для добавления элементов в начало списка, нужно в функции push() поменять местами prev и next, а также h и t. Для вывода данных на экран необходимо создать указатель на голову, и с его помощью пройти все элементы: Листинг 1.12.

```
1 void print ( list *h, list *t){ //печать элементов списка
2 list *p = h; //указатель на голову
3 while (p){ //пока не дошли до конца списка
4 cout << p->inf << " ";
5 p = p->next; //переход к следующему элементу
6 }
7 }
```

Для поиска элемента в списке пользуемся тем же алгоритмом: создаем указатель на голову, и с его помощью проходим по всему списку. Если встретили искомый элемент, прекращаем работу. Результат работы функции: либо адрес искомого элемента, либо NULL , если искомого элемента в списке нет:

```
1 list *find ( list *h, list *t, int x){ //печать элементов списка
2 list *p = h; //указатель на голову
```

```
3 while (p){ //пока не дошли до конца списка
4 if (p->inf == x) break // если нашли, прекращаем цикл
5 p = p->next; //переход к следующему элементу
6 }
7 return p; //возвращаем указатель
8 }
```

9 Динамические структуры данных. Двусвязный список. Поиск элемента. Удаление элемента из двусвязного списка

Рассмотрим теперь удаление элемента из списка.

Листинг 1.15.

```
1 void del_node (list *&h, list *&t, list *r){ //удаляем после r
2 if (r == h && r == t) //единственный элемент списка
3 h = t = NULL;
4 else if (r == h){ //удаляем голову списка
5 h = h->next; //сдвигаем голову
6 h->prev = NULL;
23
7 }
8 else if (r == t){ //удаляем хвост списка
9 t = t->prev; //сдвигаем хвост
10 t->next = NULL;
11 }
12 else{
13 r->next->prev = r->prev; //для следующего от r предыдущим
становится r->prev
14 r->prev->next = r->next; //для предыдущего от r следующим
становится r->next
15 }
16 delete r; //удаляем r
17 }
```

Для поиска элемента в списке пользуемся тем же алгоритмом: создаем указатель на голову, и с его помощью проходим по всему списку. Если встретили искомый элемент, прекращаем работу. Результат работы функции: либо адрес искомого элемента, либо NULL , если искомого элемента в списке нет:

```
1 list *find ( list *h, list *t, int x){ //печать элементов списка
2 list *p = h; //укзатель на голову
3 while (p){ //пока не дошли до конца списка
4 if (p->inf == x) break // если нашли, прекращаем цикл
5 p = p->next; //переход к следующему элементу
6 }
7 return p; //возвращаем указатель
8 }
```

10 Деревья. Основные понятия. Бинарные деревья. Обходы бинарных деревьев.

Дерево — иерархическая абстрактная структура данных. Используется в различных областях. Формально, Определение 1. Дерево — набор элементов, связанных отношениями «родитель – ребенок», удовлетворяющих следующим условиям:

1. Если дерево непустое, то существует вершина, называемая корнем дерева, не имеющая родителя.

2. Каждый узел v дерева имеет одного родителя ω. Тогда v является ребенком ω. Дерево можно также определить рекурсивно:

Определение 2. Дерево — набор элементов, удовлетворяющих следующим условиям:

1. Если дерево непустое, то существует вершина, называемая корнем дерева, не имеющая родителя.

2. Каждый узел v можно трактовать как корень своего дерева. Такие деревья называют поддеревом.

Визуально, дерево представляется в виде узлов и ребер, соединяющих родителя и его детей. Не бывает изолированных узлов. Если два узла имеют одного родителя, то эти узлы не могут быть соединены ребром (тогда это будет граф.) В английских источниках узлы, имеющие одного родителя, называются siblings. В русском языке аналога этого слова нет (только в биологии встречается сибс.)

Родитель и ребенок — это пара смежных узлов. Если говорить о более дальних связях, то вводится понятие пути. Путь из узла A до узла B — это набор узлов A,a_1 $\dots a_n$, B таких, что узел a_+1 является ребенком узла a. Длина пути — число таких узлов минус единица.

Если существует путь из узла A в узел B, тогда узел A является предком узла B, а узел A — потомком узла B. Корень дерева является предком всех остальных узлов.

Узлы разделяются на внутренние и внешние. Внешние узлы не имеют детей, очень часто называются также листьями.

Высоты узла — максимальная длина пути от узла до листа. (Находим длины путей от узла до всех листьев и выбираем максимальную из них). Соответственно, высота дерева — максимальная длина пути от корня до листа. Глубина узла — длина пути от корня до узла. Глубина узла находится однозначно, так как у каждого узла только один родитель.

Деревья бывают упорядоченными и неупорядоченными. В случае упорядоченного дерева является важным порядок следования узлов на уровне, обычно следование идет слева направо.

Бинарное дерево, т. е.,дерево, каждый узел которого имеет не более двух детей.На уровне d бинарное дерево содержит максимум 2_d узлов (на нулевом — корень,на первом — два, на втором — 4 и т. д.)

Построенное дерево необходимо вывести на экран. Для этого используются обходы деревьев. **Обход** — способ вывода всех узлов дерева ровно

один раз. Поскольку бинарное дерево состоит из левого поддерева (L), правого поддерева (R), корня (N). Понятно, что существует шесть вариантов обходов: LRV, LVR, VLR, RLV, RVL, VRL. Последние три являются симметричными первым трем, поэтому обычно рассматриваются три обхода:

11 Работа с деревом бинарного поиска.

В этом случае добавляется дополнительное условие на узлы: для любого узла левый ребенок меньше своего родителя, правый — больше. В случае случайного распределения данных такое дерево подходит для поиска данных, так как необходимо пройти только по одной ветке дерева. Но, можно подобрать данные таким образом, что дерево будет представлять собой одну ветку, что увеличивает поиск до O(n), где n — это количество элементов в дереве. Для обхода дерева удобно использовать симметричный обход, так как в этом случае на экран будет выведена отсортированная последовательность. Поскольку неравенства строгие, то дерево не содержит повторяющихся элементов, при вставке в дерево они просто игнорируются. Простейшая реализация дерева бинарного поиска состоит в следующем:

1. Первый элемент всегда является корнем;

2. Если вставляемый элемент меньше корня, ищем подходящее место на левой ветке;

3. Если вставляемый элемент больше корня — на правой.

Для дерева бинарного поиска элементарными функциями являются поиск элемента, нахождение минимального и максимального элементов, а также поиск предшествующего и следующего элеметнов (в смысле симметричного обхода).

Поиск элемента

Если указатель равен NULL, значит элемента в дереве нет, возвращаем NULL;

Если значение текущего элемента равно искомому значению, возвращаем указатель на этот элемент;

Если значение текущего элемента больше искомого, рекурсивно вызываем поиск по левой ветке;

Иначе рекурсивно вызываем поиск по правой ветке.

Поиск минимального элемента

Если нет левого ребенка, элемент минимальный и возвращаем указатель на данный элемент.

Иначе рекурсивно вызываем функцию по левой ветке.

Поиск максимального элемента

Если нет правого ребенка, элемент максимальный и возвращаем указатель на данный элемент.

Иначе рекурсивно вызываем функцию по правой ветке. Поиск следующего элемента

Если существует правый ребенок, то ищем минимальный по правой ветке.

Иначе, идем вверх по дереву, до тех пока не дойдем до корня или пока текущий элемент остается правым ребенком. Возвращаем указатель на родителя.

Поиск предыдущего элемента

Если существует левый ребенок, то ищем максимальный по левой ветке.

Иначе, идем вверх по дереву, до тех пока не дойдем до корня или пока текущий элемент остается левым ребенком. Возвращаем указатель на родителя.

Удалить элемент из дерева надо таким образом, чтобы дерево по-прежнему оставалось деревом бинарного поиска. Возможно три случая:

1. Удаление листа. Просто заменяем у родителя указатель на этот лист на NULL.

2. Удаление узла с одним ребенком. Для ребенка родителем становится «дед» (родитель удаляемого узла). Для «деда» ребенком становится «внук».

3. Удаление узла с двумя детьми. Находим следующий за удаляемым узел. У него гарантировано не будет левого ребенка. Меняем значения удаляемого и найденного узлов. Если у найденного узла нет детей, выполняем пункт 1, если есть правый ребенок — выполняем пункт 2.

12 Работа с идеально сбалансированным деревом.

Идеально сбалансированное дерево — дерево, для каждого узла которого число потомков левого узла отличается от числа потомков правого узла не более чем на единицу.

Для построения идеально сбалансированного дерева необходимо заранее знать общее количество элементов. Тогда дерево строится по следующему алгоритму:

Пусть дано n элементов. Первый элемент списка является корнем.

В левом поддереве будет $\frac{N}{2}$ элементов, в правом $\frac{N}{2} - 1$ элемент (общее число элементов минус левое поддерево минус корень.)

Сначала рекурсивно заполняем левую ветку, потом правую.

Такой способ построения позволяет гарантировать, что высота будет минимально возможной $\lfloor log_2 \text{ N} \rfloor$. Сначала заполняются левые ветки, потом правые.

Можно попробовать добавить один узел в идеально сбалансированное дерево. Самый простой вариант добавить узел либо в крайнюю правую ветку, если h_i не совпадает с h_r . Если у крайнего правого узла нет детей, новый узел становится левым ребенком этого узла, если левый ребенок есть — вставляемый узел становится правым ребенком. Если «высоты» совпадают, то дерево полностью заполнено и новый узел становится левым ребенком крайне левого листа. Но таким образом можно добавить только один

узел. Если необходимо добавить несколько узлов, лучше просто перестроить дерево.

Аналогично можно удалить один узел. Если «высоты» одинаковые — заменяем значение удаляемого листа и крайнего правого и удаляем крайний правый лист, если разные — заменяем значение крайнего левого листа и удаляемого узла и удаляем крайний левый лист. Если надо удалить несколько узлов, проще каждый раз перестраивать дерево.

Поиск элемента в дереве можно проводить с помощью любого обхода. Обходим дерево до тех пор, пока не встретим элемент или пока не закончатся узлы. В примере, приведенном ниже, поиск происходит с помощью прямого обхода.

Вывод элементов на экран тоже происходит с помощью прямого обхода, так как в этом случае, порядок следования элементов совпадает с порядком ввода элементов из файла или с клавиатуры.

13 Графы. Основные понятия.

Формально, можно сказать, что граф — это совокупность множества X, элементы которого называются *вершинами* и множества упорядоченных пар вершин, элементы которого называются *дугами*. Граф обычно обозначается (X, A).

Довольно часто каждому ребру графа ставят в соответствие какую-нибудь метку, например расстояние между двумя точками. Такую метку называют *весом*, а граф — *взвешенным* или помеченным.

В некоторых задачах имеет значение какая из вершин дуги является начальной, а какая конечной, а в некоторых нет. В первом случае граф называется *ориентированным* или орграфом. Во втором случае — *неориентированным*. Дуги в этом случае часто называются *ребрами*. В качестве примера ориентированного и неориентированного графов можно привести поиск правильного маршрута от точки A до точки B в городе, например, в центре Саратова с его односторонним движением на улицах. Пешеходу не важно разрешенное направление движения — маршрут будет проложен по неориентированному графу, т. е., если существует путь от точки A до точки B, то существует путь от точки B до точки A. Для водителя необходимо учитывать разрешенное направление движения и из того, что существует путь от точки A до точки B, необязательно следует, что существует путь от точки B до точки A. Графически ребра ориентированного графа представляются со стрелкой, показывающей направление от начальной точки к конечной. На рисунке 1 слева представлен неориентированный граф, справа — ориентированный.

Рис. 6: Caption

Два графа называются *изоморфными*, если можно поменять метки вершин одного графа таким образом, чтобы набор ребер в итоге для обоих графов стал идентичным.

Граф называется *планарным*, если на чертеже его ребра не пересекаются.

В примерах выше и дальше, будут рассматриваться простые графы. *Простой граф* — это граф, не имеющий кратных ребер (две вершины могут быть соединены только одним ребром) и *петель* (ребро, начинающееся и заканчивающиеся в одной вершине).

Рис. 7: Caption

Граф является полным, если он содержит все возможные ребра.

Если имеется ребро, соединяющее две вершины графа, то такие вершины называются *смежными*. А ребро — *инцидентным* этим вершинам. *Степень вершины* неориентированного графа — это количество инцидентных ей ребер. Для ориентированного графа можно говорить о *полустепени исхода* и *полустепени захода*. Это, соответственно, число ребер исходящих и заходящих в данную вершину.

Если все вершины имеют одинаковую степень, то говорят о *регулярных графах*. Например, K_5 граф — полный граф с пятью вершинами, имеющий степень вершин 4.

Рис. 8: Caption

Подграф — множество ребер и вершин, которые сами представляют из себя граф.

Путь в графе — это последовательность вершин a_0, a_1, a_2, ..., a_n, таких что для любых $i > 0$ вершина a_i смежна с вершиной a_{i-1}.

Простой путь — путь, все вершины и ребра которого различны. Если существует простой путь, начинающийся и заканчивающийся в одной вершине, такой путь называется *циклом*. Например, на рисунке 2 циклами являются: $0- > 6- > 4- > 5- > 0$; $0- > 6- > 4- > 3- > 5- > 0$; $3- > 4- > 5- > 3$; $9- > 11- > 12- > 9$. Цикл должен содержать как минимум 3 различных вершины и три различных ребра.

Длина пути — количество ребер, составляющих путь (или количество вершин минус единица) для невзвешенного графа и сумма весов соответствующих ребер для взвешенного графа.

Неориентированный граф называется *связным*, если из любой вершины графа существует путь в любую другую вершину. *Несвязный граф* состоит из некоторого множества *связных компонент*, представляющих собой максимальные связные подграфы.

Например, представленный на рисунке 2 граф несвязный, содержит три связные компоненты. Первая состоит из вершин 0, 1, 2, 3, 4, 5, 6, вторая — из вершин 7, 8 и третья — из вершин 9, 10, 11, 12.

Мост — ребро, удаление которого увеличивает число компонент связности. Достаточно важное ребро, например, для сети дорог — наличие моста (единственная дорога) может привести к невозможности, например, добраться до какого-нибудь населенного пункта, если что-то случится с данной дорогой.

Рис. 9: Caption

Вершина, удаление которой приводит к увеличению числа компонент связности, называется *точкой сочленения*.

Рис. 10: Caption

Вершина, из которой существует путь во все остальные вершины, называется *истоком.*

Вершины, в которую существует путь из всех остальных вершин, называется *стоком.*

В большинстве задач граф содержит малое количество из возможных ребер. Введем понятие *насыщенность* — среднее значение степени вершин, $P = \frac{2E}{N}$, где E — количество ребер, N — количество вершин.

Граф является *насыщенным (плотным)*, если количество его ребер пропорционально N^2 и *разреженным* в противоположном случае.

В зависимости от насыщенности определяется какой из алгоритмов необходимо использовать. Пусть есть два алгоритма, решающих одну задачу. Один имеет сложность $O(N^2)$, другой — $O(E \log E)$.

В случае плотного графа с $N = 1000$ и $E = 5 \times 10^5$ для первого алгоритма сложность $O(N^2) = 10^6$, для второго — сложность $O(E \log E) \approx 10^7$. Очевидно, что первый алгоритм выгоднее.

В случае разреженного графа с $N = 10^6$ и $E = 10^6$ для первого алгоритма сложность $O(N^2) = 10^{12}$, для второго — сложность $O(E \log E) \approx 2 \times 10^7$. Очевидно, что второй алгоритм выгоднее.

Рис. 11: Caption

14 Графы. Обходы в глубину и ширину.

3 Обходы

Как и в случае деревьев, для работы с графами необходим алгоритм для работы с вершинами. Обход — посещение каждой вершины графа по определенному алгоритму. Существует два обхода — в глубину (обходим смежные вершины до тех пор, пока это возможно) и в ширину (записываем в очередь все смежные вершины).

В идеале работа с графами описывается внутри класса. В нашем случае все переменные, необходимые для работы с графами, делаем глобальными (список смежности (`Gr`), список посещенных вершин (`used`), количество вершин (`N`), список вершин, составляющих путь (`path`) и т. д.).

Все алгоритмы ниже описаны с использованием списка смежности в виде вектора векторов.

3.1 Обход в глубину

Общий смысл обхода: выбираем начальную вершину, помечаем ее как посещенную, ищем смежную не посещенную вершину и вызываем обход относительно ее.

Исходные параметры: `x` — текущая вершина
Результат: тип функции — `void`
начало алгоритма
  Помечаем вершину `x` как посещенную (`used[x] = 1`);
  Записываем ее в вектор `path`;
  цикл $i = 0$ *до* $i < Gr[x].size()$ **выполнять**
    если *(существует не посещенная вершина (`Gr[x][i]`))* **тогда**
      вызываем рекурсивно эту функцию относительно вершины `Gr[x][i]`;

Рис. 12: Caption

3.2 Обход в ширину

Алгоритм прост: создаем очередь, записываем в нее начальную вершину, помечая ее как посещенную. Извлекаем голову очереди и записываем в очередь все смежные не посещенные вершины головы. Повторяем процесс до тех пор, пока очередь не пуста.

Исходные параметры: `x` — текущая вершина
Результат: тип функции — `void`
начало алгоритма
- Помечаем вершину `x` как посещенную (`used[x] = 1`);
- Записываем ее в вектор `path`;
- Помещаем ее в очередь;
- **до тех пор, пока** *очередь не пуста* **выполнять**
  - Извлекаем из очереди голову (`y`);
  - **цикл** $(i = 0$ *до* $i < Gr[y].size())$ **выполнять**
    - **если** *(существует не посещенная вершина (`Gr[y][i]`))* **тогда**
      - Помечаем вершину `Gr[y][i]` как посещенную;
      - Записываем ее в вектор `path`;
      - Помещаем ее в очередь;

Рис. 13: Caption

15 Библиотека STL. Контейнеры.

Контейнеры делятся на две больших группы: последовательные и ассоциативные. Последовательные контейнеры представляют собой набор элементов, расположение которых в контейнеры зависит от порядка поступления в контейнер: добавление элемента происходит либо в конец контейнера, либо в начало. К последовательным контейнерам относятся vector, list, deque.

Ассоциативные контейнеры представляют собой отсортированную последовательность, расположение элемента зависит от его значения. К ассоциативным контейнерам относятся set, multiset, map, multimap.

Для контейнеров должны выполняться три основных требования:

1. Поддерживается семантика значений вместо ссылочной семантики. При вставке элемента контейнер создает его внутреннюю копию, а не сохраняет ссылку на объект. 2. Элементы в контейнере располагаются в определенном порядке. При повторном переборе порядок должен остаться прежним. Для этого определены возвращающие итераторы для каждого контейнера. 3. В общем случае операции с контейнерами небезопасны. Необходимо следить за выполнением операций.

Для использования контейнеров необходимо подключать соответствующие библиотеки. Например, для использования списка подключается библиотека include<list> и т. д.

Описание любого контейнера: cont<type> x;, где cont — наименование контейнера, type — тип элементов (может быть любым, включая контейнеры). Например, vector<int> c.

Основные методы, определенные для всех контейнеров:

x.size() — возвращает размер контейнера.

x.empty() — возвращает true, если контейнер пустой.

x.insert(pos, elem) — вставляет копию elem в позицию pos. Возвращаемое значение зависит от контейнера.

x.erase(beg,end) — удаляет все элементы из диапазона [beg,end].

x.clear() — удаляет из контейнера все элементы.

Для каждого контейнера определен свой итератор. Итератор — это «умный» указатель, который содержит данные о расположении элементов. Для каждого контейнера запись p++ означает переход к следующему элементу, который определен для каждого контейнера по-разному.

Описание итератора vector<int>::iterator iter. Обращение к элементу, на который указывает итератор: *iter.

Для каждого контейнера определены два итератора x.begin(), определяющий положение начального элемента контейнера и x.end(), определяющий адрес следующей ячейки после последнего элемента контейнера. Многие алгоритмы, возвращают x.end() в качестве отсутствующего результата (например, при поиске элемента в контейнере возвращается либо итератор на этот элемент, либо x.end(), если искомого элемента в контейнере не существует).

16 Библиотека STL. Итераторы.

Итератор — это объект, предназначенный для перебора элементов контейнера STL. Итератор представляет некоторую позицию в контейнере. Основные операторы:

* — получение значения элемента в текущей позиции итератора (*iter). Для map необходимо использовать >: iter->first, iter->second.

++ — перемещение итератора к следующему элементу контейнера. Для разных контейнеров имеет разный смысл:

Вектор, дек — переход к следующему элементу, увеличивая адрес на sizeof() байт.

Список — переход к следующему элементу по полю p->next

Множество и отображение — переход к следующему элементу, используя симметричный обход дерева.

== и ! = — проверка совпадений позиций, представленных двумя итераторами. Для вектора и дека определены операции сравнения < и > .

17 Библиотека STL. Алгоритмы.

Для работы с встроенными алгоритмами необходимо подключить библиотеку <algorithm>. Некоторые алгоритмы, необходимые для обработки чис-

ловых данных, определены в файла <numeric>. Подробно описание алгоритмов рассматривать не будем. Остановимся только на общих принципах.

1. Большая часть алгоритмов возвращает итераторы. Большая часть алгоритмов в качестве параметров использует итераторы. Например, vector<int>::iterator iter = minelement(x.begin(), x.end())в качестве параметров использует итераторы. Результатом будет итератор, указывающий на элемент с минимальным значением.

Очень опасно использовать, например, следующую запись: remove(x.begin(), x.end(), *iter). Предполагается, что с помощью этого алгоритма удаляются все минимальные элементы. Но на самом деле удаляется только элемент, на который указывает iter.

Для правильной работы необходимо выполнить:

vector<int>::iterator iter = minelement(x.begin(), x.end())

int Min = *iter

remove(x.begin(), x.end(), Min)

2. При работе с интервалами всегда подразумевается полуоткрытый интервал [beg,end] т. е., включая beg и не включая end. Естественно, что beg $\leq$ end. Если алгоритм использует несколько интервалов, то для первого задается начало и конец интервала, а для второго — только начало. Например, алгоритм equal(x.begin(), x.end(), y.end()) сравнивает поэлементно содержимое коллекции x с содержимым коллекции y.

ВАЖНОЕ ТРЕБОВАНИЕ для алгоритмов, осуществляющих запись в контейнеры или для алгоритмов, использующих несколько интервалов, необходимо заранее убедится, что размер контейнера достаточен для работы с алгоритмом.

3. Алгоритмы, предназначенные для «удаления» элементов (remove и unique), на самом деле просто переставляют элементы, «удаляя» нужные, но не изменяет размер контейнера. Однако эти алгоритмы возвращают итератор, указывающий на новый конец контейнера. И можно удалить все элементы, расположенные после этого итератора с помощью метода x.erase(): vector<int>::iterator iter = remove(x.begin(), x.end(), val)

x.erase(iter, x.end())

4. Модифицирующие алгоритмы (алгоритмы, удаляющие элементы, изменяющие порядок их следования или значения) не могут применяться для ассоциативных контейнеров. Некоторые алгоритмы имеют могут быть представлены в нескольких видах: обычном, с суффиксом `_if` и суффиксом `_copy`.

Обычный алгоритм используется при передаче значения. Например, replace(x.begin(), x.end(), old, New)

заменяет все элементы со значением old значением New.

Алгоритм с суффиксом `_if` используется при передаче функции. Например, replace`_if`(x.begin(), x.end(), func, New)

заменяет все элементы, для которых func(elem) возвращает true, значением New.

Алгоритм с суффиксом `_copy`. используется в случае, когда результат записывается в новый контейнер (или интервал). Необходимо убедиться,

что памяти в приемном контейнере хватает для копирования результата. Например, replace_copy.(x.begin(), x.end(), y.begin(), old, New) заменяет все элементы со значением old значением New и копирует результат в контейнер y.

18 Хэширование. Методы разрешения коллизий

Рассмотрим другой способ построения базы для поиска — хэширование. Вообще, хэширование достаточно распространено. Например, во многих базах для аутентификации используется не пароль, а хэш этого пароля. Т. е., Вы вводите пароль, вычисляется хэш этого пароля и проверяется с хранящимися данными.

В данном разделе не будут рассмотрены сложные функции хэширования. Рассмотрим только простейшие варианты.

Основная цель хэш-таблиц — расположить элементы в таблице по K ячейкам в соответствии со значением хэш-функции h(x). Для каждого элемента x строится хэш-функция, такая, что значение h(x) находится в интервале [0, . . . , B 1]. Потом элемент x ставится в ячейку, соответствующую значению хэш-функции.

Элемент x часто называют ключом, h(x) — хэш-значением. Значение хэш-функции должно быть обязательно целочисленным. В дальнейшем будет предполагать, что элементы располагаются по ячейкам хэш-функции равномерно и независимо.

Для простых хэш-функций достаточно часто бывают ситуации, когда несколько ключей имеют одинаковые значения хэш-функции. Такие ситуации называют коллизиями.

Предполагаем, что ключ всегда является целочисленным значением. Если, например, ключом является строка, то можно представить ее в виде целого числа, написанного в определенной системе счисления.

Представление данных в таком случае напоминает поразрядную сортировку (можно считать хэштаблицей, где хэш-функция — цифра в определенном разряде). Пусть есть N данных. Размер таблицы — M, следовательно, хэш-функция принимает значения в диапазоне [0, . . . , M 1]. Хэш-таблица представляет собой массив списков. Список выбран как структура, позволяющая удалять данные за константное время. При хорошей хэш-функции в каждой ячейке таблицы будет находится в среднем $\alpha = \frac{N}{M}$ Назовем α коэффициентом заполнения хэш-таблицы.

Алгоритм 1: Создание хэш-таблицы

Вход: A — массив размерности N, M — размерность хэш-таблицы

Выход: Хэш-таблица

начало алгоритма

цикл пока не дошли до конца массива выполнять

· Определяем значение хэш-функции k = h(A[i]);

· Добавляем элемент массива в k-ый список хэш-таблицы;
конец алгоритма

Алгоритм 2: Поиск или удаление элемента хэш-таблицы Вход: A — хэш-функция размерности M, X — элемент для поиска или удаления Выход: Измененная хэш-таблица (при удалении) или указатель на найденный элемент (при поиске) начало алгоритма

· Определяем значение хэш-функции для элемента X;

цикл пока не дошли до конца списка соответствующей ячейки хэш-таблицы выполнять

· Ищем необходимый элемент;

· Удаляем найденный элемент;

конец алгоритма

Открытое хэширование

Основные достоинства открытого хэширования: 1. Неограниченный размер хэш-таблицы (элементы в списки можно добавлять без ограничений) 2. Поиск и удаление за время $O(1 + \alpha)$, что меньше поиска в сбалансированном дереве бинарного поиска.

Простейшие хэш-функции.

Самая простая хэш-функция — остаток от деления на M : h(x) = x mod M. Хэширование достаточно быстрое. Если правильно подобрать размер таблицы, то хэш-функция достаточно эффективна.

Метод умножения

1. Сначала x умножается на коэффициент $0 < A < 1$ и получаем дробную часть полученного выражения. 2. Результат умножается на M и берется целая часть. Таким образом хэш-функция имеет вид h(x) = [M(xA mod 1)], где xA mod 1 = (xA [xA]) получение дробной части, $[z]$ — целая часть числа z.

Закрытое хэширование

В случае закрытого хэширования все элементы располагаются непосредственно в таблице. Это позволяет избавится от указателей, но накладывает существенные ограничения на размер таблицы.

Каждая ячейка таблицы содержит либо ключ, либо значение NULL. В случае закрытого хэширования удаление элементов вызывает сложности, поэтому не стоит пользоваться этим способом. Также недостатком закрытого хэширования является возможность неудачного подбора хэш-функции, так что для вставки очередного элемента не найдется свободной ячейки.

вставки или поиска последовательно исследуются ячейки до тех пор, пока не встретится пустая ячейка. Хэш-функция имеет зависит от двух параметров: $h'(x, i)$

Линейное хэширование

Первой возможной ячейкой является та, которую дает вспомогательная хэш-функция, далее последовательно исследуются все ячейки, пока не встретится пустая. Возможно создание длинной последовательности занятых ячеек, ячеек, что удлиняет время поиска.

Квадратичное хэширование

Каждая следующая ячейка смещена относительно нулевой ячейки (значение$h'(x)$) на величину, характеризующуюся квадратичной зависимостью, что лучше линейной. На при неудачном выборе параметров c_1, c_2, m, может возникнуть ситуация, когда для элемента не окажется свободной ячейки, удовлетворяющей заданной хэш-функции.

Двойное хэширование

Начальная ячейка — это значение $h_1(x)$, а смещение — значение $h_2(x)$. Для того, чтобы хэш-функция могла охватить всю таблицу, значение h2 должно быть взаимно простым с размером хэш-таблицы. Вариантов выбора несколько: либо выбрать M степенью двойки, а h_2 сконструировать таким образом, чтобы она возвращала только нечетные значения; либо выбрать $h_1(x) = \text{x mod M}$, а $h_2(x) = 1 + (\text{x mod } M\prime)$, где M — простое число, а $M\prime = \text{M}$ 1. Данная функция реально зависит от двух параметров, поэтому является достаточно хорошей и содержит малое число цепочек занятых ячеек.