Поиск в таблице по ключу

В условиях роста объемов обрабатываемой информации эффективность алгоритмов поиска и сортировки выступает на первый план.
При нахождении элемента с заданным значением в массиве целых чисел (или массива символов), которые легко общепринятым образом сравниваются между собой, обычно различают постановки задачи поиска первого или последнего вхождения элемента в массив. Также может ставиться задача поиска всех вхождений заданного элемента в массив.
В данном разделе при дальнейшем рассмотрении делается принципиальное допущение: совокупность данных, в которой необходимо найти заданный элемент, фиксирована.
Будем считать, что множество из n элементов задано в виде массива a[n] Of Item. Обычно тип Item описывает запись с некоторым полем, выполняющим роль ключа. Задача заключается в поиске элемента, ключ которого равен заданному аргументу поиска a[i].inf=x. Полученный в результате поиска индекс i, удовлетворяющий условию, обеспечивает доступ к другим полям обнаруженного элемента.
Так как в данном случае исследуются алгоритмы поиска, и обработка найденных данных не рассматривается, принимается следующее упрощение –тип Item включает только ключ, то есть, запись (элемент массива a[i]) состоит из одного ключа. При описании алгоритмов поиска в массивах данных будем придерживаться классификации, приведенной на рис. 41.

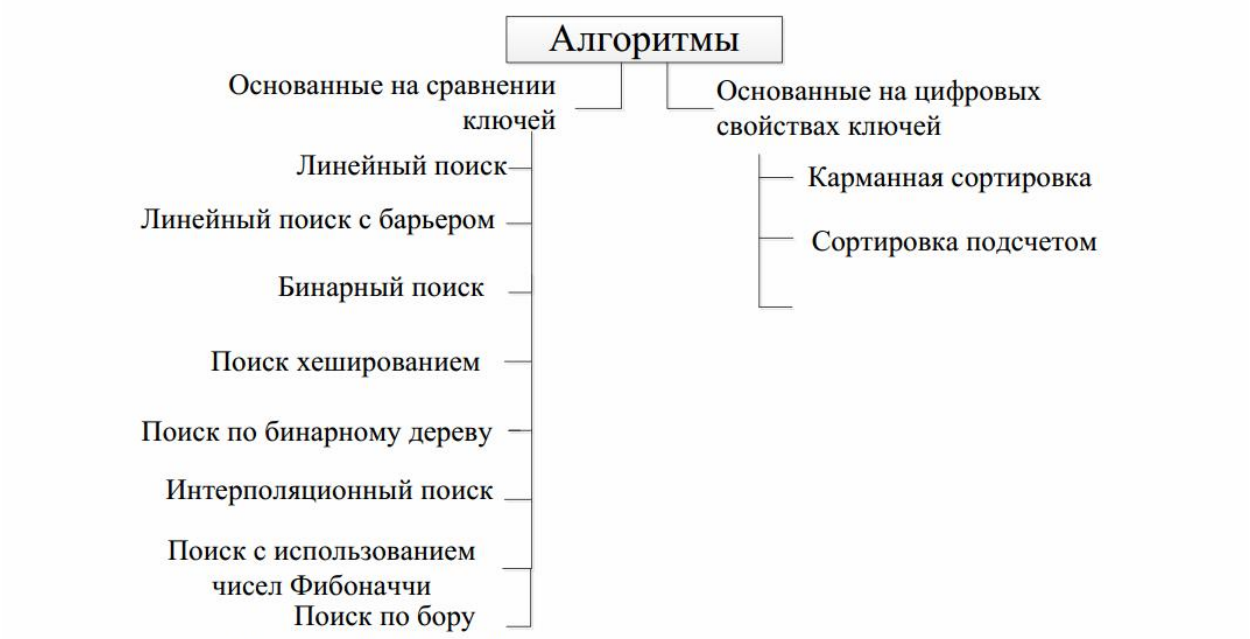

Рисунок 41. Классификация алгортмов сортировки массивов

1 Алгоритм линейного поиска

Линейный поиск (linear search) представляет тривиальный алгоритм, который используется, когда отсутствует какая-либо информация о

данных, среди которых необходимо провести поиск. В данном случае процесс поиска требует простого последовательного просмотра элемент за элементом всего массива.
Он начинается с просмотра первого элемента и завершается, либо, когда искомый элемент найден, либо, когда будет просмотрен весь массив, и не было обнаружено совпадения.
Для этой цели реализовать алгоритм удобнее всего оператором цикла While (или Repeat-Until) с двойным условием. Первое условие контролирует принадлежность массиву (i < n). Второе условие - условие продолжения поиска
(a[i] = x). В теле цикла обычно записывается только один оператор, изменяющий индекс (параметр) массива a.
После выхода из цикла необходимо проверить, по какому из условий цикл завершился.
Следует обратить внимание на то, что если элемент найден, то он найден вместе с минимально возможным индексом, то есть, это первый из таких элементов в массиве. Равенство i = n свидетельствует, что элемента с искомым значением ключа в массиве не обнаружено.
Окончание цикла гарантировано, поскольку на каждом шаге значение i увеличивается, и, следовательно, за конечное число шагов оно обязательно достигнет n; фактически же, если совпадения не было, это произойдет после nшагов.

LINEAR_SEARCH
i ← 1 // инициализация параметра цикла
While (i<n) And (a[i]<>x)
Do i ← i + 1
If (i = n) Then <Элемент не найден>
Else <Элемент найден>

Примечания.

Порядок следования условий в логическом выражении заголовка оператора цикла имеет принципиальное значение, так как условие a[i]=x выполняется всего один раз, в то время как условие i<n в общем случае выполняется многократно;

При прямом последовательном поиске в среднем проверяются n/2 элементов. В лучшем случае будет проверяться только один элемент, а в худшем случае будут проверяться n элементов. Если информация размещается на диске, то поиск может оказаться чрезвычайно долгим. Однако, не следует забывать, что для не сортированных данных, последовательный поиск является единственным возможным алгоритмом поиска.

Линейный поиск представляет собой превосходную иллюстрацию стратегии «грубой силы», с её характерными сильными (простота) и слабыми (низкая эффективность) сторонами.

Легко заметить, что вычислительная сложность алгоритма порядка O(n).

2 Алгоритм линейного поиска с барьером

На каждом шаге только что рассмотренного алгоритма линейного поиска, помимо необходимого увеличения индекса, производится вычисление двойного логического выражения, что является непозволительной роскошью. Естественно желание упростить логическое выражение путем упразднения одного из условий, связанного с контролем границ массива. Однако при этом необходимо гарантировать, что совпадение с аргументом поиска произойдет всегда. С этойцелью достаточно в конец массива поместить дополнительный элемент со значением x. Такой вспомогательный элемент называется барьером (sentinel), ведь он будет «ограждать» алгоритм от выхода за границу массива. В этом случае потребуется размер массива увеличить на единицу. Алгоритм линейного поиска с барьером можно записать следующим образом

LINEAR_SEARCH_SENTINEL (a,n)
a[n+1] ←= x
i ←1 // инициализация параметра цикла
While a[i]<>x
Do i ←i + 1
If (i = n+1) Then < элемент не найден >
Else < элемент найден >

Примечания.

В случае алгоритма поиска с барьером после выхода из цикла не следует забывать проводить проверку, был ли найден элемент массива или сравнение произошло с «барьером»;

Имеется модификация этого алгоритма, когда размер исходного массива не изменяется. В этом случае барьер устанавливается на место последнего элемента массива, а последний элемент сначала сохраняется (y←a[n]), а затем восстанавливается (a[n]←y). Однако эта модификация потребует дополнительной проверки значения последнего элемента.

Несмотря на то, что данный алгоритм линейного поиска с барьером работает быстрее простого линейного поиска, его вычислительная сложность порядка O(n), как в наихудшем, так и в среднем случаях.

3 Алгоритм двоичного поиска

Очевидно, что можно значительно повысить эффективность поиска, если данные в исходном массиве будут упорядочены. Одним из таких алгоритмов

как раз и является алгоритм двоичного поиска (binary search) или поиска делением пополам (или алгоритм дихотомии), который может применяться исключительно для массивов, удовлетворяющих условию a[k-1]<=a[k], где 1<=k<=n.

Идея алгоритма заключается в следующем. Сначала производится проверка среднего элемента массива. Если его ключ больше ключа искомого элемента, то делается проверка среднего элемента из первой половины. В противном случае делается проверка среднего элемента из второй половины. Этот процесс повторяется до тех пор, пока не будет найден требуемый элемент или не будет

проверен весь массив.

В приведенном ниже псевдокоде две переменные left и right обозначают соответственно левый и правый концы фрагментов исходного массива, в которых может быть обнаружен аргумент поиска x.

BINARY_SEARCH (a,n)
left ←1 right ← n flag ←0 // инициализация переменных алгоритма
While (left <= right) And (flag =0)
Do
d ← Div ((left+right) /2)
If a[d] = x Then flag←1
Else
Do
If x >a[d] Then left←d +1
Else right← d - 1
If flag =1Then < элемент найден >
Else < элемент не найден >

Например, для поиска в массиве 4 26 31 41 41 56 58 77 числа 58 сначала делается проверка среднего элемента, которым является число 41. Поскольку
этот элемент меньше 58, первая половина массива больше рассматриваться не
будет и поиск будет продолжен во второй половине массива, то есть, среди чисел 41 56 58 77. Здесь средним элементом является 56, значит, поиск будет продолжен среди чисел 58 77. На следующем шаге искомый элемент будет найден и алгоритм завершит свою работу.

Следует отметить, что порядок следования условий в логическом выражении заголовка оператора цикла имеет принципиальное значение, так как проверка на равенство flag=0 встречается всего один раз и приводит к завершению цикла и к окончанию работы.

Однако, более важным является упрощение логического выражения за счет полного отказа от одного из условий (подобно тому, как это было сделано в линейном поиске с барьером). И, как ни странно упразднению в данном случае
подлежит проверка на совпадение.

С учетом сказанного можно переписать алгоритм в следующем виде

```
BINARY_SEARCH_ MODIF (a,n)
left ←1  right ← n  // инициализация переменных алгоритма
While left < right
Do
d ←Div ((left+right) /2)
If x > a[d] Then left←d +1
Else right ←d
If a[l] = x Then < элемент найден >
Else < элемент не найден >
```

В этом случае цикл закончится, как только выполниться условие left>=right. Точнее при завершении цикла будет выполняться двойное равенство left=right=d. Но не следует забывать о том, что окончание цикла ничего не
говорит о наличие или отсутствии аргумента поиска в исходном массиве. Поэтому далее в алгоритме предусмотрена дополнительная проверка a[left]=x.
Как видно из предложенной процедуры алгоритм на каждом шаге отбрасывает половину данных в массиве, поэтому максимальное количество шагов
пропорционально тому, сколько раз можно поделить число n на два, таким образом, вычислительная сложность алгоритма двоичного поиска в худшем случае пропорциональна log2n, что довольно неплохо.
В среднем в алгоритме проверяются n/2 элементов, в лучшем случае сравнение произойдет на первом шаге.
Примечания.
Поиск в линейных связных списках подобен рассмотренным выше алгоритмам поиска в массивах. Отличие заключается лишь в том, что работа с индексами заменяется работой с адресами (естественно, посредством указателей);
Алгоритм двоичного поиска использует стратегию "разделяй и властвуй".
Вычислительная сложность алгоритма двоичного поиска порядка O(log(n)).

4 Алгоритм поиска с использованием бинарного дерева

Настоящий алгоритм поиска основан на предварительном построении бинарного дерева поиска (БДП) из исходного массива. Под БДП понимается отсортированное сбалансированное бинарное дерево, в котором слева от любого
вышестоящего элемента находятся элементы с меньшими значениями, а справа
от любого вышестоящего элемента находятся элементы с большими значениями (предполагается, что тип элементов дерева допускает применение операций
сравнения). Например, для массива из 7 чисел: 4 26 31 41 56 58 77 может быть построено следующее БДП, изображение которого представлено на рис. 42.

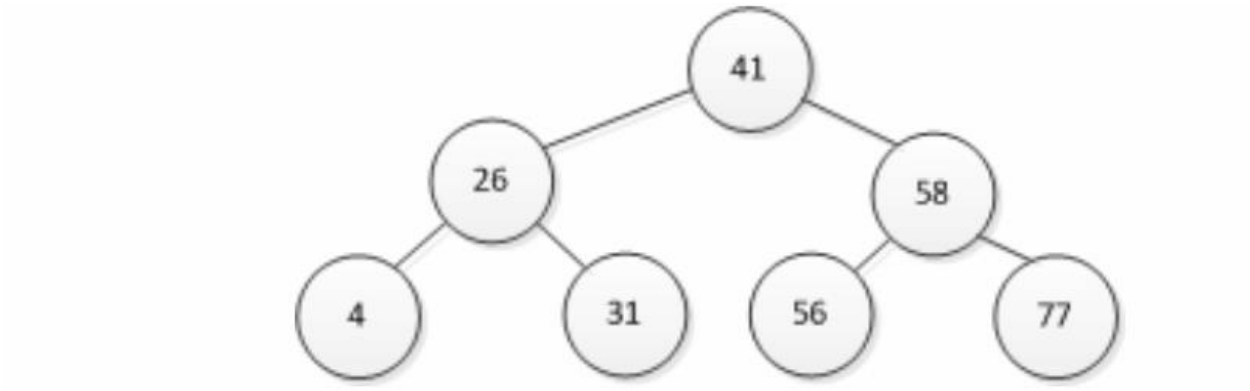

Рисунок 42. Бинарное дерево поиска для исходного массива

Принцип построения БДП достаточно прост. При n = 0 дерево сводится к «нулевому» узлу, В противном случае корневым узлом является элемент массива с индексом [n/2]+1, левое поддерево соответствует бинарному дереву, построенному из первых [n/2] элементов массива, а правое поддерево является
бинарным деревом, построенным из последующих n-[n/2]-1 элементов массива.

Алгоритм поиска по БДП также достаточно прост. Сначала поисковый элемент сравнивается с ключом корневого элемента БДП. Если поисковый элемент совпадает с ключом, поиск закончен. В противном случае поиск продолжается либо в левом поддереве, если поисковый элемент меньше ключа, либо –
в правом поддереве, если поисковый элемент больше ключа. Этот процесс
сравнения рекурсивно повторяется для следующих уровней БДП, в которых текущий узел становиться корнем текущего поддерева.

Например, поиск в рассматриваемом методе БДП элемента со значением ключа 56 будет следующим:

- 56>41 – поиск переходит в правое поддерево,
- 56<58 – поиск переходит в левое поддерево,
- 56=56 – элемент найден, поиск завершен.

Примечание.

Рассмотренный алгоритм поиска разработан Джоном Мочни в 1946 году. Нетрудно увидеть, что вычислительная сложность алгоритма поиска с использованием бинарного дерева порядка O(log(n)).

5 Алгоритм поиска с использованием чисел Фибоначчи

Поиск с использованием чисел Фибоначчи основан на последовательном сравнении отыскиваемого ключа key с элементами исходного массива, расположенными в позициях, равных числам Фибоначчи: 1, 1, 2, 3, 5, 8, 13, 21
(f1= 1, f2 = 1, fk = fk-2 + fk-1).

Алгоритм имеет ограниченное применение – его

можно использовать только для упорядоченных массивов.
Например, если требуется найти элемент с ключом 58 в массиве 4 26 31 41 41 56 58 77, последовательность сравнений данного алгоритма будет следующей:
1-ый шаг 58>a[1]=4 – поиск продолжается
2-ой шаг 58>a[2]=26 – поиск продолжается
3-ий шаг 58>a[3]=31 – поиск продолжается
4-ый шаг 58>a[5]=41 – поиск продолжается
5-ый шаг 58<a[8]=77 – найден интервал, в котором может находиться отыскиваемый ключ, а именно, между элементами с индексами 5 и 8 (41 56 58
77).
Поиск продолжается с использованием следующих чисел Фибоначчи, до тех пор, пока не будет найден интервал между двумя ключами, в котором может располагаться отыскиваемый ключ.
В найденном интервале поиск продолжается также в позициях, равным числам Фибоначчи:
6-ой шаг 58>a[5]=41 – поиск продолжается
7-ой шаг 58>a[6]=56 – поиск продолжается
8-ой шаг 58=a[7]=58 – сравнение произошло, алгоритм завершен.
Примечания.
Алгоритм поиска с использованием чисел Фибоначчи разработал Д.Фергюсон в 1960 году;
Данный алгоритм содержит только элементарные арифметические операции, а именно, сложение и вычитание, у него нет необходимости в делении на
2, как, например, в алгоритме поиска с использованием бинарного дерева. Тем
самым значительно экономится процессорное время!
Вычислительная сложность алгоритма поиска с использованием чисел Фибоначчи порядка O(log(n)). Однако, в силу вышесказанного, данный алгоритм по эффективности превосходит алгоритм поиска с использованием бинарного дерева.

6 Алгоритм интерполяционного поиска

Представьте, что Вам срочно понадобилось узнать, как переводится на русский язык какое-нибудь английское слово, то есть, необходимо найти это
слово в словаре.
По своей сути, Ваши действия в данном случае будут ничем иным, как реализацией некоторого алгоритма поиска. Аргумент поиска, искомое слово - известно. Исходный массив – это Русско-английский словарь, все значения в котором отсортированы по алфавиту. Таким образом, поиск в данном случае будет производиться в отсортированном массиве.
Метод последовательного поиска для этой цели явно не подходит и вряд

ли кто-нибудь возьмется применить поиск с использованием чисел Фибоначчи.
Конечно можно, воспользоваться методом двоичного поиска, хотя, скорее всего, разумный человек будет действовать следующим образом. Проанализируем эти действия.
Если, например, искомое слово начинается на букву T, то целесообразно сразу открыть словарь немного дальше, чем на середине. Если открылась страница со словами, начинающимися с буквы R, ясно, что теперь искать надо во второй части словаря. А на сколько продвинуться? На половину? «Ну, зачем же на половину - это больше, чем надо», скажете Вы и будете правы. Ведь в данном поиске не только ясно, в какой части массива находится искомое значение, но и, хотя приблизительно, но известно на какую величину необходимо «шагнуть» в очередной раз. И величина этого шага переменная, что существенно.
Именно это и является сутью интерполяционного поиска. Итак, существенным отличием рассматриваемого алгоритма является то, что он не просто определяет область нового поиска, но и оценивает величину нового шага.
Иначе говоря, интерполяционный поиск (interpolar search) использует способ нахождения промежуточных значений величины по имеющемуся дискретному набору значений.
Алгоритм интерполяционного поиска может быть применим только для отсортированных исходных массивов данных (например, упорядоченных по возрастанию). Более того, алгоритм базируется на предположении равномерного распределения величин. Именно поэтому, зная величину аргумента поиска x, можно предсказать более точное положение искомого элемента массива, чем просто в середине некоторого отрезка right–left (как это имело место в алгоритме двоичного поиска). Формула определения места положения следующего элемента для сравнения следует из деления длины отрезка (right-left) пропорционально величинам разностей ключей (x–aleft) и (aright –aleft).
Более подробно, идея алгоритма заключается в следующем. Первоначальное сравнение осуществляется на расстоянии шага d, которое вычисляется по формуле

d = DIV((right – left)*(x – aleft) / (aright – aleft))

где left – номер первого рассматриваемого элемента;
right – номер последнего рассматриваемого элемента;
x – аргумент поиска (отыскиваемый ключ);
aright, aleft – значения ключей элементов массива a в позициях right и left.
Затем шаг d меняется после каждой итерации по указанной формуле.
Алгоритм заканчивает работу при выполнении условия left = right. После чего принимается окончательное решение о результатах поиска.

На псевдокоде алгоритм интерполяционного поиска можно записать следующим образом

INTERPOLAR_SEARCH (a, n, x)
left ←1
right←n
While left<>right
Do
d ←left + Div ((right–left)*(x–a[left])/(a[right]–a[left]))
If x > a[d] Then left ← d + 1
Else right ←d
If a[left] = x Then < элемент найден >
Else < элемент не найден >

Пример, пусть требуется найти элемент с ключом 56 в массиве 4 26 31 41 41 56 58 77.
1) шаг 1. Представлен на рис. 43. (left = 1; right = 8);
d = 1 + DIV ((8 – 1)*(56 – 4) / (77 – 4)) = 5

1	2	3	4	5	6	7	8
4	26	31	41	41	56	58	77
left							right
				d			

Рисунок 43. 1-ый шаг интерполяционного поиска

Сравниваем (рис. 43) аргумент поиска с пятым элементом исходного массива: 56>41, следовательно, область поиска сужается:
2) шаг 2 (left = 6; right = 8);
d = 6 + DIV ((8 – 6)*(56 – 56) / (77 – 56)) = 6

1	2	3	4	5	6	7	8
4	26	31	41	41	56	58	77
					left		right
					d		

Рисунок 44. 2-ой шаг интерполяционного поиска

Аргумент поиска равен элементу массива a[6], значит right = d = 6. Сравниваем аргумент поиска с шестым элементом исходного массива: 56 = 56 - сравнение произошло после двух итераций! Результат поиска положителен, искомое число обнаружено на шестом месте.
Примечание.

Алгоритм интерполяционного поиска прекрасно работает при соблюдении гипотезы о равномерном распределении величин ключей в исходном массиве.
В этом случае данный алгоритм по эффективности превосходит алгоритм поиска с использованием бинарного дерева. Действительно, если при поиске по бинарному дереву за каждый шаг массив поиска уменьшался с n значений до n/2,
то в этом алгоритме в каждом шаге область поиска уменьшается с n значений
до корня квадратного из n.
Вычислительная сложность алгоритма интерполяционного поиска порядка O(log(log(n))).

7 Алгоритм поиска хешированием

Идея алгоритма заключается в замене поиска в исходном массиве данных на поиск адреса (индекса) в хеш-таблице (hash tables). На практике применяются различные методы хеширования (hashing) - определения хеш-функций h(k),
наиболее простым и распространенным из них является метод деления, использующий для вычисления адреса остаток от деления по модулю m:
h(k) = k mod m
где k – значение (ключ) элемента исходного множества,
m - целое число,
mod – остаток от целочисленного деления (модуль от числа k по основанию m).
Значение хэш-функции (hash value) указывает адрес, по которому следует отыскивать искомый ключ. Заметьте, что для разных ключей хеш-функция может принимать одно и тоже значение h(ki) = h(kj). Такая ситуация называется коллизией. В этом случае, необходимым дополнительным действием поиска является, так называемое, разрешение коллизий. Чаще всего используются следующие методы разрешения коллизий:
- метод цепочек;
- метод открытой адресации;
- линейная адресация;
- квадратичная и произвольная адресация;
- адресация с двойным хешированием.

Метод цепочек - наиболее очевидный путь реализации механизма разрешения коллизий, в случае, когда для нескольких различных ключей вычисляется одинаковое значение хеш-функции (один и тот же адрес), то поэтому адрес находится указатель на связанный список, который содержит все значения ключей. Поиск в этом списке осуществляется простым перебором. При грамотном выборе хеш-функций любой из списков оказывается достаточно коротким.
Наглядным примером коллизий является телефонный справочник, в котором по одному номеру телефона числиться несколько абонентов.

Естественными требованиями, предъявляемыми к хеш-функциям, являются:
- время вычисление хеш-функции должно быть минимальным;
- хеш-функция должна минимизировать число коллизий.

Рассмотрим алгоритм поиска с использованием хеш-функций и метода цепочек для разрешения коллизий на примере исходного массива:
7 13 6 3 9 1 8 5.

Выбор оптимальной хеш-функции (или выбор m в качестве основания) может быть произведен различными способами. Пусть в данном случае m будет
равным половине максимального ключа исходного массива
m = [kmax/2] = 6
Тогда для данного m получим следующий массив hash value
h(k) = {1 1 0 3 3 1 2 5}

По парным сравнением элементам исходного массива и массива хеш-функций можно получить таблицу поиска, показанную на рис. 45. При этом
хэш-функция указывает адрес, по которому следует отыскивать ключ.

Например, для поиска ключа k=8 будет определена хэш-функция
h(k)=8 mod 6=2.

Это означает, что ключ k = 8 может находиться только в строке хэш-таблицы h(k)=2 и он там существует. Итак, в данном случае, после вычисления
хэш-функции поиск будет успешно завершен после одной операции сравнения.

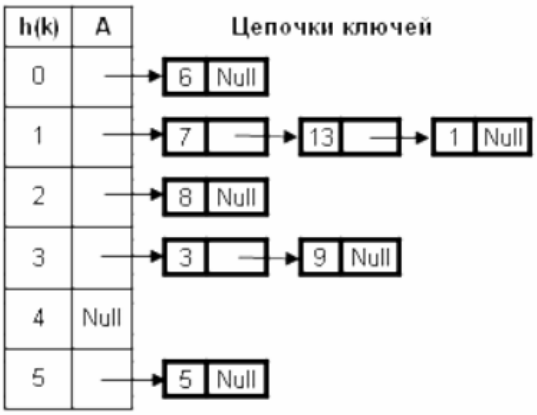

Рисунок 45. Иллюстрация метода поиска хешированием

Для поиска ключа k = 7 будет определена хеш-функция h(k) = 7 mod 6 = 1.
Это означает, что ключ k = 7 может находиться только в строке хэш-таблицы
h(k) = 1 и он там существует. В данном случае, после вычисления хэш-функции
поиск успешно завершится после прохода по цепочке из трех элементов.

Для поиска ключа k = 10 будет определена хэш-функция h(k) = 10 mod 6 =

4. Это означает, что ключ k = 10 может находиться только в строке хэш-таблицы h(k) = 4. Но строка 4 – пустая. В данном случае после вычисления хэш-функции поиск неудачно завершится после одной операции. Как уже было отмечено, очень важен правильный выбор хеш-функции. При удачном построении хеш-функции таблица заполняется более равномерно, уменьшается число коллизий и уменьшается время выполнения операций поиска, вставки и удаления. Для того чтобы предварительно оценить качество хеш-функции можно провести имитационное моделирование. Моделирование проводится следующим образом. Формируется целочисленный массив, длина которого совпадает с длиной хеш-таблицы. Случайно генерируется достаточно большое число ключей, для каждого ключа вычисляется хеш-функция. В элементах массива просчитывается число генераций данного адреса. По результатам такого моделирования можно построить график распределения значений хеш-функции. Для получения корректных оценок число генерируемых ключей должно в несколько раз превышать длину таблицы.

Примечания:

Хеш-таблицы - одно из величайших изобретений информатики. Сочетание массивов и списков с небольшой добавкой математики позволило создать эффективную структуру для организации динамических данных;

Поиск хешированием относится к алгоритмам поиска, основанным на цифровых свойствах ключей;

В настоящее время с хешированием приходится сталкиваться едва ли не на каждом шагу. Ваш любимый компилятор практически наверняка использует хеш-таблицу для управления информацией о переменных в Вашей программе.

Например, в используемом Вами web-браузере, метод хеширования применяется для хранения адресов страниц (web-ссылок), которые Вы недавно посещали.

При соединении компьютера с Интернетом, хеш-таблица применяется для оперативного хранения (cache — хеширования) недавно использованных доменных имен и их IP-адресов;

В идеале для задач поиска хеш-адрес должен быть уникальным, чтобы за одно обращение получить доступ к элементу с заданным ключом (идеальная хэш-функция). Однако на практике идеал приходиться заменять компромиссом, когда, как было рассмотрено выше, по одному хеш-адресу содержится более одного элемента и приходится мириться с разрешением коллизий.

Феноменально, вычислительная сложность алгоритма поиска с использованием хеширования порядка O(1).