Алгоритмы поиска

Задача алгоритмов поиска

Дано множество из N элементов.

Элементы представлены записями $R_1\, R_2\, R_3 \ldots\ldots\ldots\, R_N$, которые содержат специальное поле – *ключ.*

Ключи в записях уникальны, т.е. ключ однозначно определяет запись.

Совокупность всех записей называют таблицей или файлом.

Под таблицей можно понимать массив записей или небольшой по объему файл.

В алгоритмах поиска присутствует *аргумент поиска – К (ключ)* и

задача сводится к отысканию записи, имеющей К своим ключом.

Пример таблицы

```
struct student{
  char num[8]; //номер зачетки
 char fio[50];
 unsigned int date;
char group;
};
student group[30];          // таблица
```

num	fio	date	group	num	fio	date	group		num	fio	date	group
123	AA	1.12	IKBO1	100	BB	1.03	IKBO2	……	124	CC	5.05	IKBO1
Запись 1				Запись 2					Запись 3			

Два случая завершения поиска

- *удачный* – запись с ключом К найдена (позволяет определить положение записи, содержащей аргумент К);
- *неудачный* – показывает, что элемент с аргументом К не может быть найден ни в одной из записей.

Поиск со вставкой

Некоторые алгоритмы при неудачном поиске предусматривают вставку новой записи, содержащей ключ К в таблицу, такие алгоритмы называют алгоритмами *«поиск со вставкой».*

Классификация методов поиска

- Последовательный поиск
- Поиск посредством сравнения ключей
- Цифровой поиск
- Поиск с применением хеш функции - хеширование

Последовательный поиск

Предусматривает поиск записи с ключом К в таблице (массиве или файле) записей.

При последовательном поиске допустим один из двух исходов:

$K \neq K_i$ Нет записи с ключом К

$K = K_i$ Запись с ключом К найдена в позиции i

Результат:

при $K = K_i$ – ключ найден (удачно),

при $K \neq K_i$ значение, сообщающее о неудаче, например -1.

Алгоритмы последовательного поиска

Задача. Имеется таблица записей $R_1 R_2 R_3 \ldots\ldots R_N$, имеющими соответственно ключи $К_1 К_2 К_3 \ldots\ldots К_N$. *Требуется найти запись с ключом К. Предполагается, что N≥1.*

Шаг 1. Начальная установка: К и i ← 1.

Шаг 2. Сравнение ключей: К= $К_i$ вернуть i // удачное завершение алгоритма

Шаг 3. Продвижение: i=i+1

Шаг 4. Если i≤ N перейти к шаг 2

Иначе алгоритм завершается неудачно

Эффективность алгоритма оценивается количеством выполненных операций сравнения

Анализ алгоритмов поиска

Критерием оценки сложности алгоритмов поиска является **количество операций *сравнения***

Анализ алгоритма

Время работы алгоритма зависит от C – количество сравнений и S – исхода поиска: S=1 запись найдена; S=0 запись не найдена;

Начальная установка: K *i=1.*	1 1
While(i<=N)do	C
if (K = K_i) *return i;*	C C-S
else i=i+1; *endif*	C-S
return -1; od	1

Алгоритм требует 4N – 2S + 3 единиц времени.

При удачном исходе, т.е. нашли $K = K_i$, то C=i, а S=1, значит полное время равно (4i+2) t

При неудачном исходе C=N, а S=0, значит полное время равно (4N+3) t.

Если все ключи поступают на вход с равной вероятностью, то среднее значение C при удачном поиске $\frac{1+2+3+\cdots\ldots+N}{N} = \frac{N+1}{2}$

Быстрый последовательный поиск (использование барьерного элемента)

Барьер – фиктивный элемент, добавленный в конец таблицы/массива R_{N+1} и содержащий значение аргумента поиска т.е. К.

Например, есть массив ключей 1 2 3 4 5, ищем К=6. Тогда массив с барьером: 1 2 3 4 5 6.

$K_{N+1} \leftarrow K$

For i $\leftarrow$1 to N+1 do

if К= $К_i$ goto 1 ***//найдем и уйдем***

i=i+1

od

1: Если i≤ N алгоритм завершается удачно;

Иначе алгоритм завершается не удачно (i=N+1)

Анализ алгоритма

Ввод K $i \leftarrow 0$ $K_{N+1} \leftarrow K$	1 1 1
While $(K \neq K_i)$ *do*	C+1-S
$i=i+1$ *od*	C+1-S
if($i \leq N$*)* *return i;* *else* *return -1;* *endif*	1 1-S
Можно заключить, что время работы программы уменьшилось до (2C -3S+7) t и дает улучшение при C>=4 (при удачном поиске) и при N>=6 при неудачном поиске	Уменьшается количество сравнений
Улучшение за счет сокращения операций сравнения в теле цикла. В первом алгоритме было два сравнения в цикле (в заголовке и if), а в этом цикле одно в заголовке. Улучшение на 30% по сравнению с предыдущим алгоритмом.	

Анализ алгоритма

Ввод K *i ← 0* *K_{N+1} ←K*	1 1 1
While ($K \neq K_i$) do	C+1-S
i=i+1 *od*	C+1-S
if(i≤ N) *return i;* *else* *return -1;* *endif*	1 1-S
Можно заключить, что время работы программы уменьшилось до указанной величины (2C -3S+7) t и дает улучшение при C>=4 (при удачном поиске) и при N>=6 при неудачном поиске	Уменьшается количество сравнений
Улучшение за счет сокращения операций сравнения в теле цикла. В первом алгоритме было два сравнения в цикле (в заголовке и if), а в этом цикле одно в заголовке. Улучшение на 30% по сравнению с предыдущим алгоритмом.	

Поиск посредством сравнения ключей

- Последовательный поиск в упорядоченной таблице
- Бинарный поиск
- Фибоначчиев поиск
- Интерполяционный поиск
- Поиск по бинарному дереву

Алгоритмы поиска посредством сравнения ключей

Применяются к таблицам со случайным доступом и упорядоченными ключами $К_1 < К_2 < К_3 < \ldots\ldots\ldots < К_N$

При сравнения К и $К_i$ имеем:

- Если $К < К_i$ то из рассмотрения исключаются записи

$R_i R_{i+1} \ldots\ldots\ldots R_N$

- Если $К > К_i$ то из рассмотрения исключаются записи

$R_1 R_2 \ldots\ldots\ldots R_i$

- Если $К = К_i$ то поиск закончен

Отсюда: экономия времени при применении методов, за исключением случаев, когда i близка к границам таблицы.

Т.е. упорядочение позволяет создавать эффективные алгоритмы

Поиск посредством сравнения ключей

1. Т.е. таблица должна быть отсортирована по значениям ключей.

2. После сравнения аргумента К с ключом K_i, поиск продолжается одним из трех способов в зависимости от того, какое из сравнений верно:

$K < K_i$, $K = K_i$, $K > K_i$

Примечание

Если поиск надо произвести только один раз, то быстрее произвести его последовательно без предварительной сортировки.

Но если поиск в таблице выполняется часто, то эффективнее упорядочить таблицу и применить один из рассматриваемых далее методов.

Последовательный поиск в ***упорядоченной*** таблице

Постановка задачи

Дано. Имеется таблица записей $R_1\ R_2\ R_3 \ldots\ldots\ldots R_N$, причем ключи удовлетворяют условию $К_1 < К_2 < К_3 < \ldots\ldots\ldots < К_N$.

Результат. Требуется найти запись с ключом К.

Ограничения. Предполагается, что $N \geq 1$ и в конце таблицы находится фиктивная запись – барьер со значением большим К.

Установка $i \leftarrow 0$, $K_{N+1} \leftarrow \infty$ *т.е. значение* $> K$

While $(K < K_i)$*do*

$i = i + 1$

Od

if$(K = K_i)$

return i;

else

return -1;

Преимущество перед методом линейного поиска со случайным расположением ключей: отсутствие нужной записи обнаруживается быстрее примерно в два раза.

Бинарный поиск (логарифмический или деления пополам)

Постановка задачи

Дано. Имеется таблица записей $R_1\ R_2\ R_3 \ldots\ldots\ R_N$, причем ключи удовлетворяют условию $К_1 < К_2 < К_3 < \ldots\ldots < К_N$.

Результат. Требуется найти запись с ключом К.

Применяется метод разработки алгоритма «Разделяй и властвуй».

Общая схема алгоритма

1. Выбирается элемент в середине мссива - $К_{ср}$.
2. Его значение сравнивается с аргументом К.
3. Если $К = К_{ср}$ то элемент найден и возвращается индекс найденного элемента.
4. Если $К < К_{ср,}$ то поиск продолжается в части таблицы

$R_1\ R_2\ R_3 \ldots\ldots\ R_{ср-1}$ этим же методом.

5. Если $К > К_{ср,}$ то поиск продолжается в части таблицы $R_{ср+1} \ldots\ldots\ R_N$ этим же методом .

Пример. Массив ключей 2 5 7 9 11 13 45 77. Ищем 2.

Binsearch(x, n,K) *l* ← 1 r ← n if K=x[l] do вернуть *l* od *l* ← *l*+1 while(l<r) do mid=(l+r)/2 if K=x[mid] do вернуть mid od else do if K>x[mid] *l* ← mid+1 od else do r ← mid-1 od od	1 1 C1 1 C1-S C+1-S C+1-S C+1-S 1 C+1-S C+1-S C2 C2 Т.к. C=C1+C2 (сколько раз выполнялись сравнения по > <. Тогда Время работы программы (7C – 6S+9) При S=1 C=i время 7i+3, а при C=N и S=0 7N+9

Рекурсивное дерево бинарного поиска

Работу алгоритма бинарного поиска можно изобразить в виде дерева. Рассмотрим алгоритм на бинарном дереве для N=16

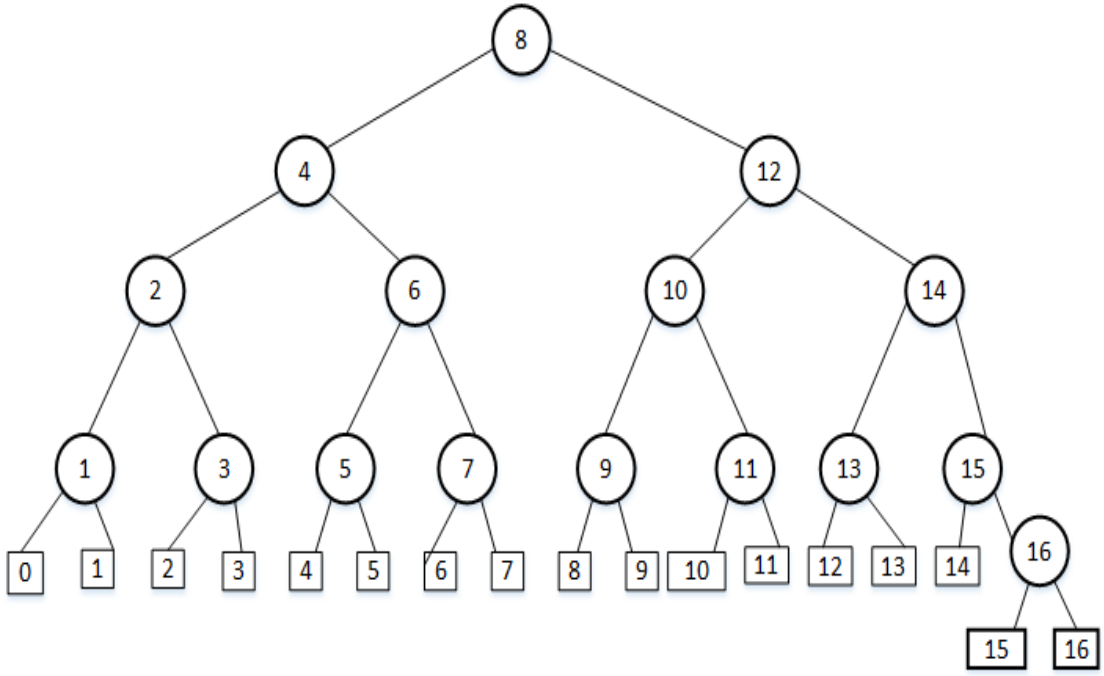
Корень дерева – это первое деление диапазона поиска на две части.

Присваиваем значения границ: *l* = 1, r = 16.

Корень дерева соответствует mid = (l + r) / 2 = 8.

Из корня дерева выходят два ребра – одно (левое) соответствует ветке «да», другое (правое) – ветке «нет» условия if (a[mid] > K), где K – аргумент поиска.

Две дочерние вершины корня – это следующие переходы по границе mid.



Первым производится сравнение K_8, что на дереве соответствуе
Если К<K_8, то алгоритм обрабатывает левое поддерево, сравнивая К и K_4.
Если К> K_8, то алгоритм обрабатывает правое поддерево, сравнивая К и K_{12}.

Неудачный поиск ведет к одному из внешних узлов (в квадратах), занумерованных от 0 до N.
Например, будет достигнут узел только тогда, когда K_6<К< K_7

Любой алгоритм поиска на основе сравнений ключей в упорядоченной таблице длины N можно представить бинарным деревом с узлами, помеченными числами от 1 до N.

Обратно, любое бинарное дерево соответствует некоторому методу поиска в упорядоченной таблице.

При поиске аргумента равного K_{10} выполняются сравнения K>K_8, $K < K_{12}, K = K_{10}$, что соответствует пути в дереве от корня, до узла 10 .

С помощью метода построения бинарных деревьев, соответствующих алгоритму бинарного поиска и индукции по N легко доказывается теорема:

при $2^{k-1} \leq N < 2^k$ удачный поиск алгоритма бинарного поиска требует min 1, max k сравнений. Неудачный поиск требует k сравнений при N=2^k-1 либо k-1 сравнений или k сравнений при $2^{k-1} \leq N < 2^k$-1.

Отсюда видно, что алгоритм требует максимально $\lfloor log_2 \mathrm{N} \rfloor$+1 сравнений. Среднее число сравнений при удачном поиске $\approx \lfloor log_2 \mathrm{N} \rfloor$-1.

Ни один из методов, основанных на сравнении ключей не может дать лучшего результата.

Среднее время работы программы Binsearch(x, n,K) составляет примерно ($7log_2$N+3) при удачном поиске и 7 log_2N+9 при неудачном поиске.

Алгоритм однородного бинарного поиска

Постановка задачи

Дано. Имеется таблица записей $R_1 R_2 R_3 \ldots\ldots R_N$, причем ключи удовлетворяют условию $К_1 < К_2 < К_3 < \ldots\ldots < К_N$.

Результат. Требуется найти запись с ключом К.

Особенность. При четном N в алгоритме бинарного поиска иногда происходит обращение к фиктивному ключу K_0, который необходимо установить равным $-\infty$ или любому значению меньшему аргумента К. Предполагается, что $N \geq 1$.

Идея алгоритма: использовать вместо трех переменных l, r, mid две переменные :

- *i* (вместо mid) – текущее положение элемента,

- delta – величина изменения *i*.

При выполнении текущего сравнения не приведшего к удачному исходу можно установить i ← i$\pm$ delta и

delta ← delta/2.

Т.е. в отличии от бинарного поиска, этот алгоритм определяет ***интервал*** для поиска К.

Поэтому алгоритм использует округления сверху и снизу.

Шаг 1. i$\leftarrow \lceil N/2 \rceil$, m $\leftarrow \lfloor N/2 \rfloor$

Шаг 2. Если K<K_i то *шаг 3.*

Если K>K_i то *шаг 4*.

Если K=K_iто вернуть *i* .

Шаг 3. (Уменьшение *i) (здесь определен интервал, где нужно продолжать поиск. Он содержит m или m-1 записей;*

i указывает на первый элемент справа от интервала).

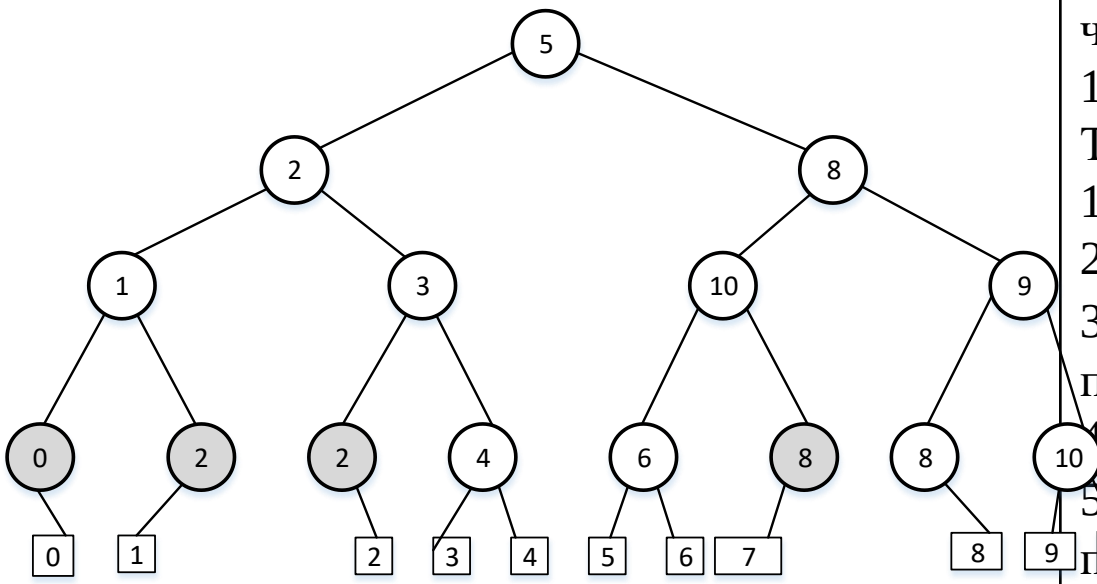
Если m=0, то алгоритм завершается неудачно.

Иначе устанавливаем: i $\leftarrow$i- $\lceil m/2 \rceil$ *и m*$\leftarrow$ $\lfloor m/2 \rfloor$ *и переходим к шаг 2*

Шаг 4. (Увеличение i) (здесь так же как и Шаг 3, только i указывает на первый элемент слева от интервала).

Если m=0, то алгоритм завершается неудачно.

Иначе устанавливаем: i $\leftarrow$i+$\lceil m/2 \rceil$ *и m*$\leftarrow$ $\lfloor m/2 \rfloor$ *и переходим к шаг 2*



Бинарное дерево для «Однородного» бинарного

Пример. Пусть есть массив из 10 чисел

10 12 15 18 20 32 45 60 70 81

Тест 1. Найти индекс числа K=70

1) i=5 m=5

2) $K_{\mathbf{5}} = \mathbf{20}$, тогда K>K_i

3) $\boldsymbol{i} \leftarrow \mathbf{i}+\lceil m/2 \rceil$ $\boldsymbol{u}$ $\boldsymbol{m} \leftarrow \lfloor m/2 \rfloor$,

получаем: i=8 и m=2 и на шаг 2

4) $K_{\mathbf{8}} = \mathbf{60}$, тогда K>$K_{\mathbf{8}}$

5) $\boldsymbol{i} \leftarrow \mathbf{i}+\lceil m/2 \rceil$ $\boldsymbol{u}$ $\boldsymbol{m} \leftarrow \lfloor m/2 \rfloor$,

получаем: i=9 и m=1 и на шаг 2

Успешное завершение i=9

При неудачном поиске при завершении работы алгоритма могут выполняться лишние сравнения. Узлы, отвечающие этим сравнениям закрашены.

Это процесс поиска называют однородным, так как разность между числами в узлах уровня i и числом в узле родителе (i – 1 уровень) есть постоянная величина delta для всех узлов уровня i .

На этом основана следующая модификация алгоритма

Оптимизация однородного бинарного поиска

Шаг 1. i←Delta[1] и j← 2

Шаг 2. Если K<K_i то *шаг 3.*

Если K>K_i то *шаг 4.*

Если K=K_iто вернуть *i* .

Шаг 3. (Уменьшение *i)*

Если Delta[j] =0*, то алгоритм завершается неудачно.*

Иначе устанавливаем: i ←i-Delta[j] *и j*← $j+1$ *и переходим к шаг 2*

Шаг 4. (Увеличение i)

Если Delta[j] =*0, то алгоритм завершается неудачно.*

Иначе устанавливаем: i ←i+Delta[j] *и j=j+1 и переходим к шаг 2*

Важным преимуществом этого однородного бинарного поиска является то, что не нужно хранить значение m. Нужно лишь ссылаться на таблицу значений delta для каждого уровня, так как они постоянны.

Delta[j]=$\left\lfloor -\frac{N+2^{j-1}}{2^j} \right\rfloor$

При удачном поиске этот алгоритм соответствует дереву бинарного поиска с той же длиной внутреннего пути. Поэтому и среднее время их соответствует $O(\log_2 N)$

Фибоначчиев поиск

Основан на таблицах, записи которых упорядочены в порядке возрастания ключей.

Алгоритм поиска Фибоначчи представляет собой альтернативу бинарному поиску.

Если разделять исходное множество ключей не пополам, а на подмножества, начальные и конечные индексы которых являются числами Фибоначчи, то при компьютерном исполнении можно *избежать операций деления.*

Числа Фибоначчи можно вычислять, используя сложение, которое выполняется быстрее, чем деление, поэтому в ряде случаев этот алгоритм предпочтительнее.

При использовании данного алгоритма предполагается, что множество ключей представляет собой возрастающую последовательность, т.е. выполняются соотношения

$$k_1 < k_2 < ... < k_{i-1} < k_i < k_{i+1} < ...k_{n-1} < k_n.$$

Данный метод позволяет выделить диапазон значений, в котором разместился аргумент К. Диапазон определяется индексами на основе последовательности чисел Фибоначчи.

Рассмотрим пример массива и поиска K=42

{ 3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}

Ниже приведены состояния множества *K*. Красным цветом указаны подмножества элементов, начальный и конечный индексы которых составляют два последовательных числа Фибоначчи.

{ 3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}

{ 3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}

{ 3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}

{ 3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}

{ 3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}

{ 3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52} д.б. 21 его нету

Состояние соответствует $m = 8$, для которого $F_{m-1} = 13$ и $F_m = 21$. Последнее значение превышает количество ключей $n = 18$, поэтому подмножеством, в котором находится искомый ключ будет {35, 37, 42, 45, 48, 52}. ?

Процесс выделения подмножеств продолжается до тех пор, пока не будет найдено совпадение ключей (или пока невозможно будет найти такие числа F_{m-1} и F_m, в этом случае проверяется совпадение искомого ключа и последней пары значений Fq, Fj).

Алгоритм состоит в определении такого минимального числа m, что для искомого ключа K выполнялось соотношение K_{Fm-1}<K<K_{Fm}. Иными словами, K расположен между ключами, индексы которых являются двумя последовательными числами Фибоначчи.

После того как эти числа будут определены, алгоритм поиска Фибоначчи применяется ко множеству F_{m-1} до $F_{m.}$

Последовательность первых 10 чисел Фибоначчи :

$$\{0, 1, 1, 2, 3, 5, 8, 13, 21, 34\}.$$

Когда не найден диапазон, в котором может находиться значение равное аргументу K, то для его поиска вновь применяем уже к этому диапазону этот же метод.

В этом некоторое сходство с бинарным поиском.

Алгоритм Фибоначчиева поиска

FibSearch(x,N,K)

$i \leftarrow F_m$

p $\leftarrow F_{m-1}$ p и q последовательные числа Фибоначчи

q$\leftarrow F_{m-2}$

Шаг 1. Если K<K_i то шаг 2

Если K>K_i то шаг 3

Если K=K_i вернуть i

Шаг 2. (Уменьшение i).Если $q=0$, неудачный поиск.

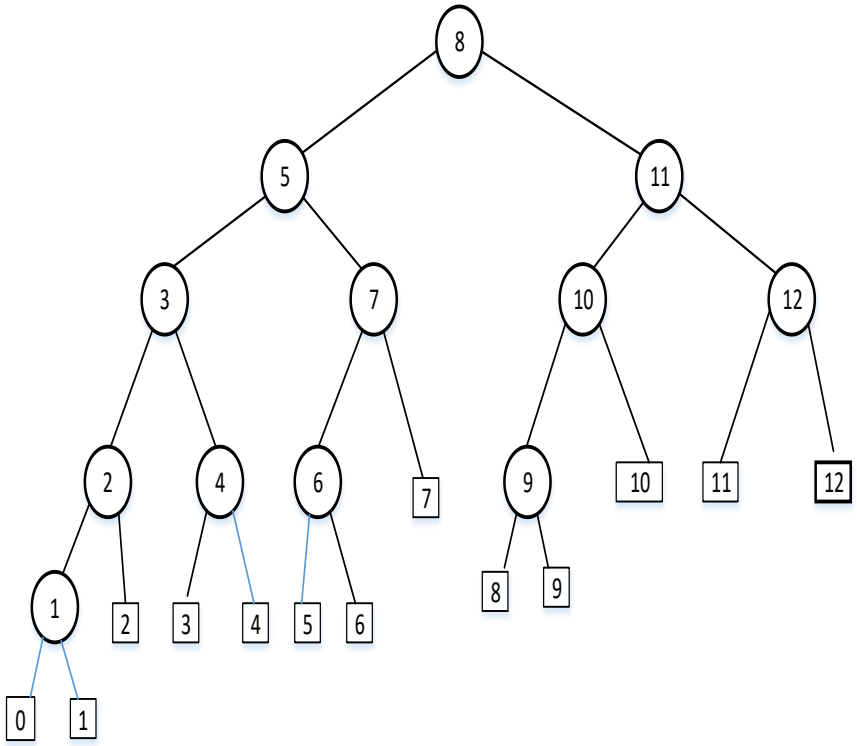
Иначе $i \leftarrow i - q$, p $\leftarrow q$, p $\leftarrow$p-q и на шаг 1

Шаг 3. (*Увеличение* i). *Если $p=1$, неудачный поиск.*

Иначе $i \leftarrow i + q$, p $\leftarrow$p-q, q $\leftarrow$q-p и на шаг 1

Этот алгоритм предполагает, что номер N+1 в массиве определяет F_m число Фибоначчи.

Дерево Фибоначчи уровня 6



Дерево порядка k имеет внутренних узлов $F_{k+1} - 1$ и F_{k+1} внешних узлов.

Алгоритм построения дерева

Если k=0 или k=1 дерево сводится к 0 внешнему узлу

Если k>=2, то корнем является F_k , левое поддерево – это дерево Фибоначчи порядка k-1, правое поддерево есть дерево Фибоначчи порядка k-2 с числами в узлах увеличенными на F_k.

Следует отметить: числа в сыновьях каждого внутреннего узла отличаются от чисел в этих узлах на одну и ту же величину – на число Фибоначчи. Например, 5=8-F_4, $11 = 8 + F_4$

Анализ Фибоначчиева поиска

Среднее время 8.6 log_2n , чуть медленнее чем в однородном бинарном поиске.

Интерполяционный поиск

Интерполяционный алгоритм поиска — алгоритм для поиска по заданным ключом в индексированном массиве, который упорядочен по значению ключей.

В математике **интерполяция - это** нахождение промежуточных значений между двумя (и более) известными величинами

Это подобно тому, как люди ищут определенное имя в телефонной книге.

Суть алгоритма:

На каждом шаге вычисляется, в каком поле поиска может быть элемент, основываясь на граничных значениях этого поля и ключе искомого элемента, обычно с помощью линейной интерполяции и выбирается некоторая позиция в массиве.

Ключ на выбранной позиции сравнивается с искомым ключом и, если они не равны, то в зависимости от результата сравнения, пространство поиска сводится к части до или после данного ключа.

Этот метод будет работать только в том случае, когда сравнение между ключами является разумным.

К примеру, двоичный поиск всегда выбирает середину в данном поле поиска и отвергает одну из половин, сравнивая значение середины с искомым.

А линейный поиск сравнивает все объекты по одному, игнорируя упорядоченность.

1. Исходное множество должно быть упорядочено по возрастанию ключей.

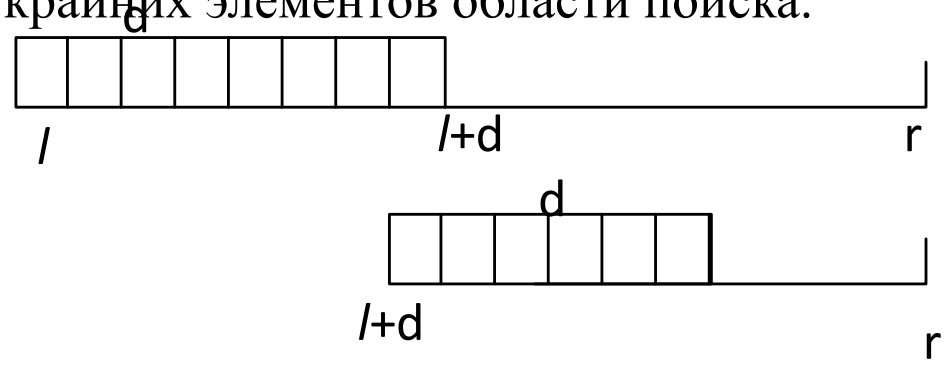
2. Если известно, что ключ К находится между ключами K_l и K_r (l и r – номера крайних элементов области поиска), то далее делаем пробу на расстоянии d, которое вычисляется по формуле

$d = (l - r)(K - K_l)/(K_r\text{-}K_l)$.

Важно отметить, операция деления в выражении строго целочисленная, т. е. дробная часть, какая бы она ни была, отбрасывается.

3. Формула, определяющая алгоритм интерполяционного поиска :
$mid = l + d;$

Здесь mid – номер элемента, с которым сравнивается значение ключа, key – ключ (искомый элемент), A – массив упорядоченных элементов, l и r – номера крайних элементов области поиска.



Пример 2. Дано множество ключей.

$$\{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95, 98, 102\}.$$

Требуется отыскать ключ К = 90.

Шаг 1. Определяется шаг d для исходного множества ключей (l=1;.r= 17):

$$d = [(17-1)(90-4)/(102-4)] = 14.$$

mid=l+d=1+14=15

Сравнивается ключ, стоящий под 15-м порядковым номером, с отыскиваемым ключом: 95 > 90, следовательно, сужается область поиска

Шаг 2. Определяется шаг d для множества ключей (l= 1 ; r= 15):

$$d = [(15-1)(90-4)/(95-4)] = 13.$$

$$\{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95\}.$$

mid=l+d =1+13

Сравниваем ключ, стоящий под 14-м порядковым номером, с отыскиваемым ключом: 90 = 90 ключ найден.

Анализ интерполяционного поиска

Интерполяционный поиск предпочтительнее бинарного. Один шаг бинарного поиска уменьшает количество записей с n до ½ n.

А один шаг интерполяционного (если ключи в таблице распределены случайным образом) – с n до $\sqrt{n}$.

Интерполяционный поиск требует в среднем около $log_2 log_2$n шагов.

Сложность O($log_2 log_2$n)

Поиск по бинарному дереву

Применяется к ***динамическим*** таблицам данных, т.е. таким в которых при выполнении поиска требуется удалять записи по ключу и вставлять записи в определенном порядке.

В предыдущих алгоритмах мы использовали неявное бинарное дерево только для понимания алгоритмов бинарного и Фибоначчиева поиска.

Рассмотренные алгоритмы предназначены для поиска в структурах фиксированного размера, так как последовательное расположение записей делает операции вставки и удаления достаточно трудоемкими.

Если таблица динамически изменяется, то экономия от алгоритма бинарного поиска не покроет затрат на поддержание упорядоченного расположения ключей.

Явное использование структуры бинарного дерева позволяет оптимизировать выполнения операций вставки и удаления записей из таблицы и производить эффективный поиск записей по ключу.

Получаем метод удобный как для поиска в таблице, так и для сортировки.

Бинарное дерево поиска обладает свойствами:

В корневом узле любого поддерева находится ключ, значение которого:

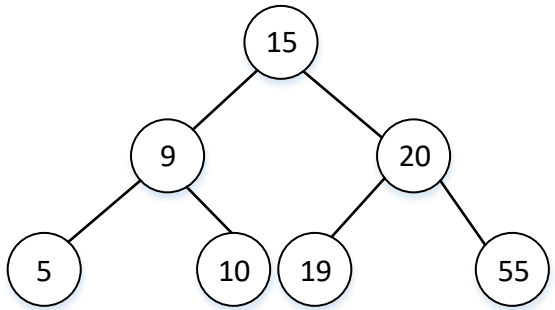
- больше значений ключей в его левом поддереве
- меньше значений ключей в его правом поддереве

Таким образом, элементы включаются в дерево и образуют упорядоченную последовательность.

Пример построения бинарного дерева поиска

Пусть имеется последовательность ключей: 15, 9, 20, 5, 10, 19, 55

Построим дерево поиска - вставка значения в дерево



структура узла дерева

NODE

KEY	
Ссылка на левое поддерево LLINK	Ссылка на правое поддерево RLINK

Применив симметричный обход дерева можно получить отсортированные ключи:

5 9 10 15 19 20 55

Алгоритм Поиск со вставкой по дереву

Дана таблица записей, образующих бинарное дерево.

Требуется найти аргумент K . Если такого ключа нету в таблице, то вставляем новый узел со значением K в подходящее место в дерева.

Пустые указатели будем обозначать NULL.

ROOT – корневой узел дерева.

Пусть ROOT≠NULL, т.е. дерево не пусто.

P - указатель на узел типа NODE для продвижения по дереву.

BinTreeSearch(ROOT, K)

Шаг 1. $P \leftarrow ROOT$

Шаг 2. Если K<KEY(P) то на шаг 3

Если K>KEY(P) то на шаг 4

Если K=KEY(P) то поиск завершен удачно

Шаг 3. (В левое поддерево)

Если LLINK(P)≠NULL то $P \leftarrow$LLINK(P) и Шаг 2

Иначе на шаг 5

Шаг 4. (В правое поддерево)

Если RLINK(P)≠NULL то $P \leftarrow$RLINK(P) и Шаг 2

Шаг 5. (Вставка узла в дерево)

создание узла на указателе Q

KEY(Q) ← K, LLINK(Q) ←NULL, RLINK(Q) ←NULL

На самом деле теперь надо включить узел в дерево: Если K было меньше KEY(P), LLINK(P) ←Q, иначе RLINK(P) ←Q,

Поиск по дереву требует около 2 log_2N сравнений. Т.е. временная сложность O(log_2N) и по памяти O(N).

Хеширование и хеш-таблица

Хеширование - это механизм, позволяющий обеспечит доступ к записи в таблице за фиксированное время, т.е. O(1) в лучшем случае и O(n) в худшем случае.

Вопрос: Как можно достичь сложность O(1) ?

Ответ: Только если на прямую обратиться к нужному элементу, как в массиве.

Рассмотрим пример использования ключей в качестве индекса массива.

Пример 1. Пусть требуется создать систему для управления персоналом предприятия, численность сотрудников в котором 100 человек. Хранить надо анкетные данные этих сотрудников. Все сотрудники имеют ИНН. Поэтому ключом к данным в хранилище может быть ИНН.

ИНН состоит из 10 цифр. Для прямого доступа к данным по ключу можно создать массив из такого количества элементов, чтобы в качестве индекса массива можно было использовать ИНН. Это будет массив из 10^{10} элементов, а сотрудников всего 100, т.е. занятыми будут только 100 ячеек. Нерациональное представление данных в памяти.

Хороший вариант представления данных в рассматриваемом примере– массив из 100 элементов. Но как обеспечить время доступа к элементу за время O(1).

Хеширование прелагает, создать отображение (установить соответствие) множества ключей в множество целых чисел – множество индексов массива.

Отображение это функция, определенная на множестве элементов (области определения) одного типа и принимающая значения на множестве элементов (область значений) другого типа.

Т.е. отображение можно представить M(d)=r где d – ‘значение из области определения M, а r значение из области значений.

Хеширование - это процесс преобразования ключа в индекс массива, называемого хеш – таблицей.

Хеш-таблица - это еще одна структура для организации доступа к данным в структурах. Она может хранить сами элементы данных, или ключи со ссылками на данные.

Хеш функция

Хеш функция – это функция, осуществляющая процесс преобразования значения ключа в индекс массива, который называют хеш – таблицей.

Хеш – таблица – это массив определенного размера, в ячейках которого размещаются элементы данных. Место (индекс элемента в таблице) для размещения данных в таблице, вычисляет хеш-функция на основе значения ключа.

Пример 2. Заполнение хеш-таблицы записями с ключами из примера 1 и поиска значений в хеш таблице.

Пусть хеш-таблица имеет размер L=100 (по количеству сотрудников), хеш-функция h(key) должна вырабатывать индекс в пределах от 1 до 100, тогда алгоритм вычисления такого значения на основе ключа может быть совершенно простым: ***key % L.***

Итак, пусть ИНН сотрудников таковы: 1111112311, 1111112312, 1111112302.

Тогда хеш – функция h(1111112311) вернет индекс – 11, а это значит, что запись о сотруднике с ИНН равным 1111112311 разместиться в хеш-таблице под индексом 11. Соответственно запись с ключом 1111112312 разместиться по индексу 12, а 1111112302 по индексу 2.

После того как создана хеш-таблица, можно выполнять поиск, удаление записей на основе заданных ключей.

Например, мы хотим вывести сведения о сотруднике с ИНН равным 1111112312. Для этого ключ 1111112312 передадим той же хеш – функции, которая вычислит индекс равный 12, а следовательно мы получаем доступ к значению с заданным ключом.

Индексы	Ключ
1	1111112311
2	1111112302
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	1111112312
100	

Принцип однородного хеширования

Для успешного хеширования очень важно, чтобы функция создавала такие индексы, что размещаемые в таблице данные были равномерно (однородно) распределены по таблице. Т.е. созданный индекс не зависел от индексов с которым хешированы другие элементы.

Т.О. для применения хеширования необходимо:

- чтобы элементы размещаемые в хеш - таблице имели уникальный ключ
- подобрать хеш-функцию, обеспечивающую принцип однородности.

Коллизия – это ситуация, когда для двух разных ключей функция создает одинаковый индекс.

Поэтому процесс хеширования включает 2 этапа:

- Вызывается хеш-функция, которая, оперируя с ключом, возвращает индекс в хеш-таблице.
- Обработка возникающей коллизии.

Ключ	Индекс	
111001	1	
111205	5	
111312	12	
114101	1	коллизии
111312	12	коллизия
111313	13	
111305	5	коллизия

Пример 3. Возникновение коллизий. Создадим хеш-таблицу для рассмотренного ранее примера для сотрудников (100 человек) из L=100 элементов. При этом будем использовать хеш - функцию h(key)=key % L.

Так как в одной ячейке массива может храниться только данное с одним ключом, тогда появляется задача: каким – то образом запись с ключом, получившим коллизию разместить в хеш таблице. Этот процесс получил название - устранение коллизии.

УСТРАНЕНИЕ КОЛЛИЗИЙ

Два подхода к устранению коллизий

- ***Цепное хеширование*** (формирование цепочек из элементов, хешированных с одним индексом)
- ***Хеширование с открытым адресом*** (таблица большого объема в элементы которой записываются данные)

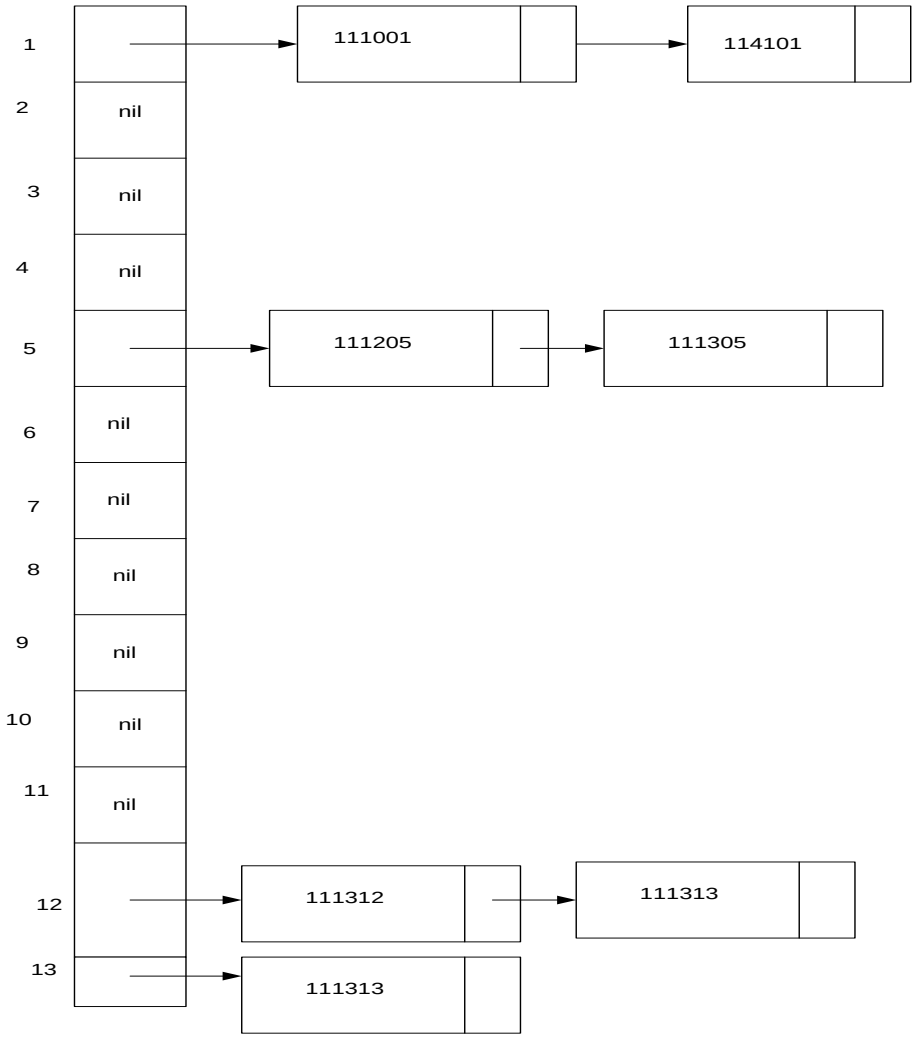
Устранение коллизии – связанно в некоторой степени с хеш-функцией т.е. какая часть ключа выбрана для хеширования. Но какую бы функцию бы не выбрали коллизии будут иметь место.

Цепное хеширование

Это один из способов разрешения коллизий, когда элементы, ключи которых получили один индекс, хранятся в одном однонаправленном списке, а вершина этого списка хранится в хеш-таблице. Тогда хеш-таблица – это массив указателей на такие списки.

Пример 4. Создание хеш – таблицы, использующей динамические списки для устранения коллизий.

Создадим хеш-таблицу для ключей из примера 3. При возникновении коллизии данные с ключом помещаются в начало списка, который в таблице связан с элементом, для которого хеш-функция создала индекс.


1
111001
114101
2
nil
3
nil
4
nil
5
111205
111305
6
nil
7
nil
8
nil
9
nil
10
nil
11
nil
12
111312
111313
13
111313

Хеширование с открытым адресом

Второй подход к разрешению коллизий – таблица с открытым адресом. Значение вставляется непосредственно в таблицу (массив из записей). Связанных списков нет.

Открытый адрес – это свободная ячейка таблицы, а закрытый адрес – это занятая ячейка.

Таблица – это массив, элементы которого могут содержать только ключ и ссылку на значение или само значение вместе с ключом.

Рассмотрим пример на создание хеш-таблиц с открытым адресом.

Пример 1. Пусть множество значений, в котором необходимо выполнять операции поиска представлен типизированным файлом. Записи файла имеют уникальный ключ – целое число. Для доступа к элементам файла создадим хеш-таблицу с открытым адресом, элемент хеш-таблицы будет хранить ключ и указатель на запись с этим ключом в файле. Тогда хеш-таблицу можно определить в программе так:

```
Typedef int Tkey;
struct Trec{
        Tkey Ke;                    //ключ
        void * ptr;        //указатель на запись в файле
};
struct TTable{
        int M;                      //длина таблицы
        int N;                      //количество закрытых адресов таблицы
        Trec *Keys;        //непосредственно таблица из элементов
};
TTable HeshTable;
```

Алгоритм операции вставки нового значения в таблицу будет состоять из двух блоков:

- получение индекса с помощью хеш-функции (обозначим индекс ind)
- если этот адрес открыт, то разместить в нем значение;
- если адрес закрыт, то надо подобрать первый открытый адрес.

Самый простой способ подбора – это линейный подбор со смещением равным единице. Суть линейного подбора: проверить ячейку таблицы следующую за ячейкой с индексом ind, т.е. ind=ind+1, если и этот адрес закрыт, то снова выполняем подбор со смещением единица и так до тех пор до тех пор, пока не будет найден открытый адрес.

Пусть в таблицу надо вставить следующие данные:

ключ	фамилия
111001	Иванов
111002	Петров
111007	Сидоров
112001	Волков
212001	Лисицын
303002	Жаворонков
304002	Медведев
305002	Рыбин
303010	Акулов

индекс	ключ	фамилия
1	111001	Иванов
2	111002	Петров
3		
4		
5		
6		
7	111007	Сидоров
8		
10		
11		
…….	……..	…….
100		

Создадим таблицу из 100 элементов. Первые три значения вставляются соответственно в ячейки с индексами 1, 2, 7. В результате таблица будет содержать следующие данные.

При вставке значения с ключом 112001 возникает коллизия, тогда подбираем линейно со смещением единица открытый адрес, это ячейка с индексом 2, но это закрытый адрес, тогда вновь применяем подбор и это ячейка с индексом 3, в которой размещаем значение.

Далее надо вставить значение с ключом 212001. Ячейки с индексами 1 и 2 и 3 закрыты, но открыта ячейка с индексом 4, производим в нее вставку значения.