Тема 4. Методы поиска

4.2. Логарифмический поиск в статических таблицах

4.2.1. Бинарный поиск

Бинарный (*двоичный*) поиск предъявляет ряд требований к организации таблицы $T = \{x_1, x_2, \ldots, x_n\}$. Во-первых, имена в таблице должны храниться в их естественном порядке, т. е. таблица должна быть отсортирована. Во-вторых, должен обеспечиваться прямой доступ к именам таблицы, т. е. для статических таблиц наиболее удобно последовательное представление, когда имена распределены в памяти с произвольным доступом в последовательных ячейках. Другими словами, бинарный поиск требует отсортированной таблицы, организованной в виде массива.

Идея бинарного поиска заключается в том, что имя z ищется в интервале $[l, h]$, крайними точками которого являются указатели l (для нижней границы интервала) и h (для верхней границы интервала). Новый указатель m устанавливается в средней или близкой к ней точке интервала. Если $z = x_m$, искомое имя найдено. Если $z < x_m$, интервал поиска сводится к интервалу $[l, m - 1]$, если $z > x_m$ – к интервалу $[m + 1, h]$, и процесс повторяется. Если интервал становится пустым, поиск завершается безуспешно. Для получения логарифмического времени необходимо устанавливать указатель m за время, не зависящее от длины интервала, что обеспечивается возможностью прямого доступа к именам таблицы. Реализация рассмотренной идеи представлена алгоритмом 4.4.

$l \leftarrow 1$
$h \leftarrow n$

$$\textbf{while}\ \ l \le h\ \ \textbf{do}\begin{cases} \text{// в этот момент имеет место} \\ \text{// } 1 \le l \le h \le n,\ z \notin \{x_1, ..., x_{l-1}\},\ z \notin \{x_{h+1}, ..., x_n\} \\ m \leftarrow \lfloor (l+h)/2 \rfloor \\ \textbf{case}\begin{cases} z = x_m : \textbf{return}(m)\ \ \text{// найдено} \\ z < x_m : h \leftarrow m-1 \\ z > x_m : l \leftarrow m+1 \end{cases} \end{cases}$$

$\textbf{return}(0)$ // не найдено

Алгоритм 4.4. Бинарный поиск

Для анализа алгоритма последовательность сравнений ключа z с именем x_i (обозначим $z : x_i$) удобно представить расширенным бинарным деревом. На рис. 4.1 представлено расширенное бинарное дерево, соответствующее бинарному поиску в таблице из девяти имён. Первое сравнение – это корень дерева, его левое и правое поддеревья являются бинарными деревьями, структура которых определяется последующими сравнениями. Алгоритм останавливается, когда он находит $z = x_i$ или когда достигает листа (внешней вершины) в случае безуспешного поиска. Высота дерева $T(1, n)$, соответствующего бинарному поиску в таблице с n именами, равна $\lceil \log (n + 1) \rceil$. Поскольку высота дерева равна числу сравнений $z : x_i$, которые требуется сделать в худшем случае, можно сделать заключение, что для таблицы с n именами бинарный поиск требует не более $\lceil \log (n + 1) \rceil$ сравнений, т. е. временная сложность алгоритма бинарного поиска есть $O(\log n)$.

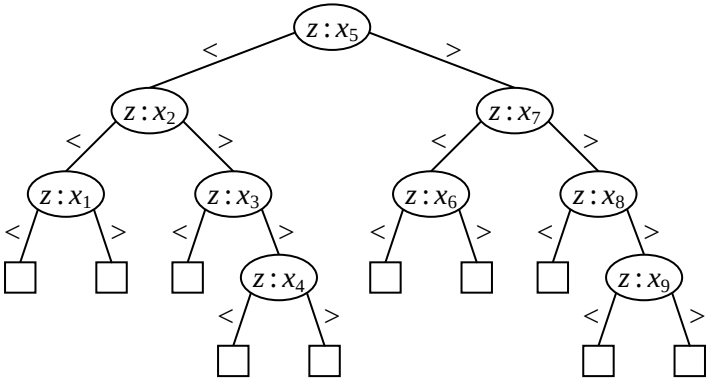


Рис. 4.1. Дерево, соответствующее бинарному поиску в таблице из девяти имён

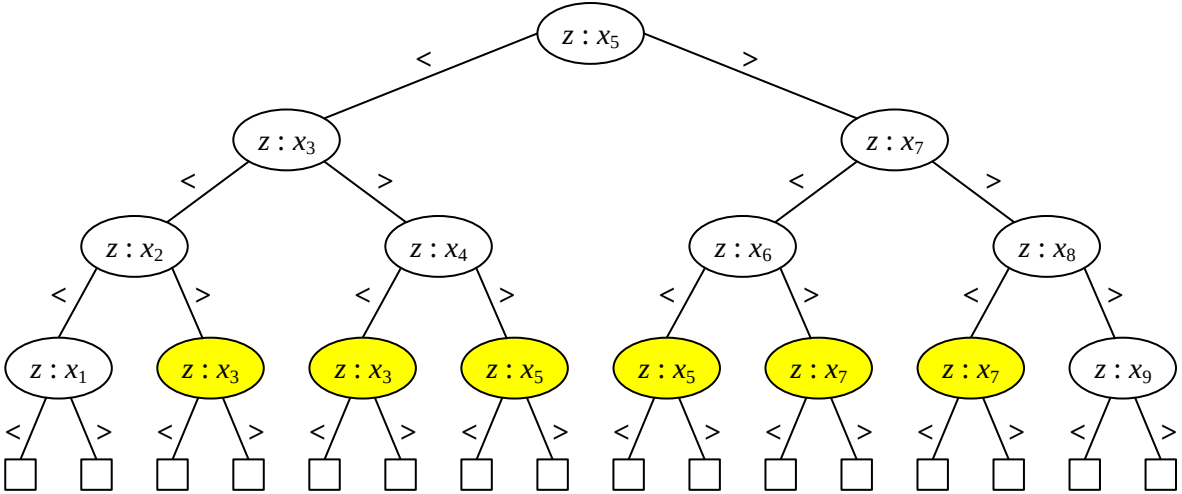
4.2.2. Однородный бинарный поиск

Однородный (*равномерный*) бинарный поиск является модификацией обычного бинарного поиска. Эта модификация заключается в следующем. Вместо трех указателей – l (нижняя граница интервала), h (верхняя граница интервала) и m (середина интервала) – используется только два: текущая позиция m и величина его изменения δ. После каждого сравнения, не давшего равенства, можно установить $m \leftarrow m \pm \delta$ и $\delta \leftarrow \delta / 2$.

Практическая реализация однородного бинарного поиска требует исключительного внимания к мельчайшим деталям, например, округлять результат деления к ближайшему большему или ближайшему меньшему целому, в каких случаях требуются имена-сторожа и т. п.

Один из вариантов реализации заключается в следующем. Сначала устанавливаются $m = \lceil n / 2 \rceil$ и $\delta = \lfloor n / 2 \rfloor$. Затем сравниваются ключ поиска z и имя x_m. Если $z = x_m$, поиск успешно завершен. В противном случае происходит либо увеличение (если $z > x_m$), либо уменьшение (если $z < x_m$) m на $\lceil \delta / 2 \rceil$. В обоих случаях устанавливается новое значение $\delta = \lfloor \delta / 2 \rfloor$. Снова производится сравнение z с x_m и т. д. Процесс продолжается до тех пор, пока на каком-то шаге не будет $z = x_m$ (успешный поиск) либо значение δ не станет равным нулю (безуспешный поиск). При такой реализации в случае четного n необходимо установить имя-сторож $x_0 = -\infty$. Детали реализации алгоритма предлагаются в качестве упражнения.

Если процесс поиска представить в виде расширенного бинарного дерева, то легко обнаружить, что разность между индексами имен вершин на уровне l и ее вершины-предшественника на уровне $l-1$ представляет собой константу δ для всех вершин на уровне l. Поэтому такой поиск и называется однородным. По дереву можно также увидеть, что при безуспешном поиске перед окончанием работы алгоритма могут производиться лишние сравнения имен (выделены желтым цветом).



Теорию, лежащую в основе однородного бинарного поиска, можно пояснить следующим образом. Пусть на начальном этапе имеется интервал для поиска длиной $n-1$ (длина интервала [a, b] есть $b-a$). Сравнение со средним элементом (для четного n) или одним из двух средних элементов (для нечетного n) дает два интервала длиной $\lfloor n/2 \rfloor - 1$ и $\lceil n/2 \rceil - 1$. После повторения этого процесса k раз получается 2^k интервалов, наименьший из которых имеет длину $\lfloor n/2^k \rfloor - 1$, а наибольший – $\lceil n/2^k \rceil - 1$. Таким образом, длины двух интервалов на одном уровне отличаются не более чем на единицу. Это делает возможным выбор среднего элемента без запоминания последовательности точных значений длин интервалов.

Главным достоинством однородного бинарного поиска является то, что имеется возможность исключить из процесса поиска все операции деления, связанные с величиной δ. Для этого используется дополнительная вспомогательная таблица (конечно, если нет ограничений по объему памяти) с вычисленными заранее значениями $\delta_j = \left\lfloor \frac{n+2^{j-1}}{2^j} \right\rfloor$ для каждого j-го шага поиска, где $1 \le j \le \lfloor \log n \rfloor + 2$. Тогда первое сравниваемое имя таблицы имеет индекс $m = \delta_1$; после каждого сравнения, не давшего равенства, устанавливаются $m = m \pm \delta_j$ и $j = j + 1$. Процесс завершается безуспешно, если достигается $\delta_j = 0$. Ясно, что для четного n необходимо имя-сторож $x_0 = -\infty$. Такая модификация позволяет осуществлять поиск быстрее обычного бинарного поиска. Однородный поиск требует времени $O(\log n)$.

4.2.3. Поиск Фибоначчи

Последовательность чисел Фибоначчи определяется рекуррентным соотношением

$$F_0 = 0,\ F_1 = 1,\ F_k = F_{k-1} + F_{k-2} \text{ при } k \geq 2.$$

Другими словами, в последовательности Фибоначчи

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

каждое число равно сумме двух предыдущих. Причем числа F_k экспоненциально растут с ростом k.

Числа Фибоначчи позволяют разработать альтернативу бинарному поиску. Технология поиска Фибоначчи заключается в следующем. Интервал поиска длины F_k разбивается на два интервала с длинами F_{k-1} и F_{k-2} соответственно, приводя к делению на две части в соотношении $\alpha = \left(\sqrt{5}-1\right)\!\!\diagup 2 \approx 0{,}62$ (в отличие от бинарного поиска, где $\alpha \approx 0{,}5$).

Пусть имена представляют собой натуральные числа 1, 2, ..., n, т. е. $x_i = i$, $1 \le i \le n$. Это упрощает изложение без потери общности, так как в упорядоченной таблице имеет место однозначное соответствие между именем и его порядковым номером (индексом). Расширенное бинарное дерево (*дерево Фибоначчи*), соответствующее поиску Фибоначчи, показано на рис. 4.2. Это дерево Фибоначчи порядка 6, т. е. корню дерева сопоставлено имя $i = F_6$.

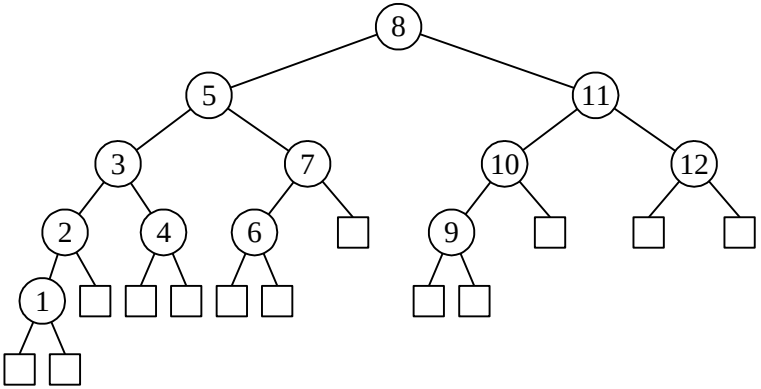


Рис. 4.2. Дерево Фибоначчи порядка 6

В общем случае дерево Фибоначчи порядка k имеет $F_{k+1} - 1$ внутренних и F_{k+1} внешних вершин (соответствуют безуспешному поиску). Строится оно следующим образом.

Если $k = 0$ или $k = 1$, дерево состоит из единственной вершины, представляющей собой внешнюю вершину.

Если $k > 1$, дерево состоит из корня, которому сопоставлено число (имя) F_k; левое поддерево представляет собой дерево Фибоначчи порядка $k - 1$; правое поддерево – дерево Фибоначчи порядка $k - 2$ с числами, увеличенными на величину F_k.

Следует отметить, что за исключением внешних вершин имена двух сыновей каждой внутренней вершины отличаются от имени отца, являющегося корнем поддерева порядка k, на одну и ту же величину – на число Фибоначчи F_{k-2} (например, для вершины 8 с сыновьями 5 и 11 имеет место $5 = 8 - F_4$ и $11 = 8 + F_4$). Если такая разница на каком-либо уровне составляет F_j, то на следующем уровне она будет равна F_{j-1} для левой ветви дерева и F_{j-2} – для правой. Например, $3 = 5 - F_3$, а $10 = 11 - F_2$.

Тогда поиск Фибоначчи ключа z в таблице $T = \{x_1, x_2, \ldots, x_n\}$ будет заключаться в следующем. Для удобства описания предполагается, что $n + 1$ представляет собой число Фибоначчи F_{k+1}. Первым исследуется имя x_i с $i = F_k$ (дерево порядка k). После каждого сравнения, не давшего равенства, если не выполнены критерии безуспешного поиска, устанавливается $i \leftarrow i \pm F_{k-2}$ и продолжается исследование поддеревьев меньшего порядка. Критериями безуспешного поиска являются: для левого поддерева – если $F_{k-2} = 0$, а для правого поддерева – если $F_{k-1} = 1$. Ясно, что нет необходимости хранить последовательность Фибоначчи, поскольку, зная два соседних числа Фибоначчи, можно однозначно восстановить всю последовательность. Поиск Фибоначчи для случая $n = F_{k+1} - 1$ представлен алгоритмом 4.5.

$i \leftarrow F_k$
$p \leftarrow F_{k-1}$
$q \leftarrow F_{k-2}$

$$\textbf{while}\ \ p \geq 1\ \textbf{and}\ q \geq 0\ \ \textbf{do case}\begin{cases} z = x_i : \textbf{return}\,(i)\ \ \text{// найдено} \\ z > x_i : \begin{cases} i \leftarrow i + q \\ p \leftarrow p - q \\ q \leftarrow q - p \end{cases} \\ z < x_i : \begin{cases} i \leftarrow i - q \\ t \leftarrow p - q \\ p \leftarrow q \\ q \leftarrow t \end{cases} \end{cases}$$

return$\,(0)$ // не найдено

Алгоритм 4.5. Поиск Фибоначчи для таблиц размера $n = F_{k+1} - 1$

Чтобы распространить процедуру поиска Фибоначчи для произвольного $n \geq 1$, можно использовать следующий прием. Необходимо найти наименьшее $m \geq 0$ такое, что $n + m = F_{k+1} - 1$. Тогда первым исследуется имя x_i с $i = F_k - m$.

$$
\begin{aligned}
&j \leftarrow 1\\
&\textbf{while} F_j < n+1 \textbf{ do } j \leftarrow j+1\\
&m \leftarrow F_j - 1 - n\\
&i \leftarrow F_{j-1} - m
\end{aligned}
$$

Перед сравнением имен дополнительно необходима проверка условия $i \leq 0$ для предотвращения выхода за пределы левых поддеревьев. Чтобы избежать дополнительных затрат времени в цикле на проверку условия $i \leq 0$, можно использовать вычисленное значение m по-другому: если результат самого первого сравнения $z > x_i$, где $i = F_k$, то устанавливается $i \leftarrow i - m$ и осуществляется переход к исследованию правого поддерева обычным поиском Фибоначчи.

Наихудшее время работы поиска Фибоначчи несколько больше обычного бинарного поиска, поскольку дерево Фибоначчи в общем случае не является полностью сбалансированным бинарным деревом. Среднее время поиска несколько меньше, чем у бинарного поиска [8]. Это связано с наличием в цикле только аддитивных операций и отсутствием мультипликативных операций. Временная сложность поиска Фибоначчи есть $O(\log n)$.

4.2.4. Интерполяционный поиск

Интерполяционный поиск предполагает равномерное распределение значений имен в некотором интервале от l до h, и исходная таблица $T = \{x_1, x_2, \ldots, x_n\}$ упорядочена по значениям имен. Поэтому, зная ключ поиска z, можно предсказать более точное положение искомого имени в таблице, чем просто в середине некоторого интервала. Положение следующего имени x_m для сравнения с z определяется делением интервала пропорционально разностям имен $x_h - x_l$ и $z - x_l$, т. е. $m = l + \lfloor (h-l)(z-x_l)/(x_h - x_l) \rfloor$. После каждого сравнения, не давшего равенства, интервалы поиска корректируются как в алгоритме обычного бинарного поиска. Таким образом, единственным отличием интерполяционного поиска от бинарного является метод определения m.

Асимптотически интерполяционный поиск превосходит бинарный и требует в среднем $O(\log \log n)$ операций. Однако пока таблица не очень велика, данное преимущество не компенсирует требуемое для дополнительных вычислений время.