Алгоритмы поиска
Лекция 5

Алгоритмы поиска

Поиск – процесс нахождения конкретной информации в ранее созданном множестве данных. Обычно данные представляют собой записи, каждая из которых имеет хотя бы один ключ.

Ключ поиска – это поле записи, по значению которого происходит поиск. Ключи используются для отличия одних записей от других. Целью поиска может быть **нахождение всех записей** если они есть) с данным значением ключа (**фильтрация**).

Поиск является одним из наиболее часто встречаемых действий в программировании.

Существует множество различных алгоритмов поиска, которые принципиально **зависят от способа организации данных**.

У каждого алгоритма поиска есть свои **преимущества и недостатки**. Поэтому важно выбрать тот алгоритм, который лучше всего подходит для решения конкретной задачи.

Алгоритмы поиска

Исчерпывающий поиск (exhaustive search), полный перебор — алгоритм нахождения заданного значения произвольной функции на некотором отрезке.

Полный перебор (или **метод «грубой силы»** от *brute force* — метод решения задачи путем перебора всех возможных вариантов. Сложность полного перебора зависит от размерности пространства всех возможных решений задачи.

В криптографии на сложности полного перебора основывается, например, **оценка криптостойкости шифров**.

В частности, **шифр** считается **криптостойким**, если не существует метода взлома существенно более быстрого, чем полный перебор всех ключей.

Алгоритмы поиска

Поиск значения функции осуществляется простым сравнением очередного рассматриваемого значения (как правило поиск **происходит слева направо**, то есть от меньших значений аргумента к большим) и, **если значения совпадают (с той или иной точностью**), то поиск **считается завершённым**.

- В связи с малой эффективностью по сравнению с другими алгоритмами линейный поиск обычно **используют,** только **если отрезок поиска содержит очень мало элементов**.
- Но так как линейный поиск не требует дополнительной памяти или обработки/анализа функции, то **может работать в потоковом режиме при непосредственном получении данных из любого источника**.
- Так же, **линейный поиск часто используется в виде линейных алгоритмов поиска максимума/минимума**.

Алгоритмы поиска

Поиск в линейных структурах

Требуется проверить, входит ли заданный ключ в массив. Если входит, то найдем номер этого элемента массива, то есть, определим **первое вхождение** заданного ключа (элемента) в исходном массиве.

1. **Линейный, последовательный поиск.** Данный алгоритм является простейшим алгоритмом поиска и в отличие, например, от двоичного поиска, не накладывает никаких ограничений на функцию и имеет простейшую реализацию.

Наихудший случай для этого алгоритма возникает, **если элемент находится в конце списка или вообще не присутствует в нем.** В этих случаях, алгоритм проверяет все элементы в списке, поэтому время его выполнения (сложность) **в наихудшем случае порядка O(N).**

Если элемент находится в списке, то **в среднем алгоритм проверяет N/2 элементов** до того, как обнаружит искомый. Поэтому в усредненном случае **время выполнения** алгоритма также **порядка O(N).**

Алгоритмы поиска

2. Поиск с барьером — модификации алгоритма последовательного поиска**.** Идея поиска с барьером состоит в том, чтобы **не проверять каждый раз в цикле условие, связанное с границами множества.** Это можно обеспечить, **установив в данном множестве так называемый барьер.**

Барьер это любой элемент, который удовлетворяет условию поиска. Тем самым будет ограничено изменение индекса.

Выход из цикла, может произойти либо на найденном элементе, либо на барьере. В качестве барьера используется дополнительный элемент, устанавливаемый в конце массива.

Поиск с барьером работает быстрее, но **временная сложность** алгоритма остается такой же **— O(n).**

Алгоритмы поиска

3. Поиск в связанных списках

Поиск методом полного перебора — это единственный способ поиска в связанных списках. Так как доступ к элементам возможен только при помощи указателей на следующий элемент, то необходимо проверить по очереди все элементы с начала списка, чтобы найти искомый.

Если хранить указатель на конец списка, то **можно добавить в конец списка ячейку, которая будет содержать искомый элемент. Этот элемент называется сигнальной меткой** (sentinel). Это позволяет обрабатывать особый случай конца списка так же, как и все остальные.

Сравните с поиском с барьером.

Добавление метки в конец списка гарантирует, что в конце концов искомый элемент будет найден.

При этом программа не может выйти за конец списка, и **нет необходимости проверять условие окончания цикла в цикле While**. Для больших списков это условие проверяется множество раз, и выигрыш времени суммируется.

Алгоритмы поиска

Поиск в связанных списках

```
function SentinelSearch(target : Longint; top,BottomCell
                                        : PCell) : PCell;
var
   bottom_sentinel : TCell;
begin
   // Добавление метки.
   BottomCell^.NextCell := @bottom_sentinel;
   bottom_sentinel.Value := target;
   // Обычный поиск
   top := top^.NextCell;
   while (top^.Value < target) do
      top := top^.NextCell;
   if ((top^.Value<>target) or (top = @bottom_sentinel)) then
      Result := nil
   else
      Result := top;

   // Удаление метки.
   BottomCell^.NextCell := nil;
end;
```

Алгоритмы поиска

Если список упорядочен, то можно **прекратить поиск, если найдется элемент со значением, большим, чем значение искомого элемента.**

Некоторые алгоритмы **используют потоки для ускорения поиска** в связанных списках.

4. Поиск элемента с использованием двоичного дерева поиска

При помощи указателей в ячейках списка можно организовать **список в виде двоичного дерева поиска**. **Поиск элемента** с использованием этого дерева **займет время порядка O(log(N)), если дерево сбалансировано, и O(N), в противном случае**.

Алгоритмы поиска

5. Поиск в упорядоченных массивах

Линейный поиск

Если во время выполнения поиска алгоритм **находит элемент со значением, большим, чем значение искомого элемента, то он завершает свою работу**. При этом искомый элемент не находится в списке, так как иначе он бы встретился раньше.

Бинарный (двоичный, дихотомический) поиск

Алгоритм двоичного поиска (binary search) сравнивает элемент в середине массива с искомым. Если искомый элемент меньше, чем серединный, то алгоритм продолжает поиск в первой половине массива, если больше — во второй половине.

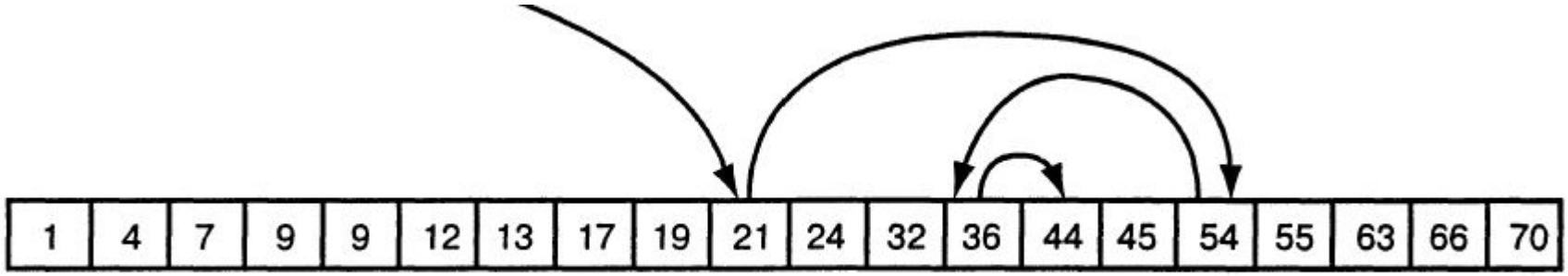
Хотя по своей природе алгоритм является рекурсивным, его достаточно просто и лучше записать без применения рекурсии.

На каждом шаге число элементов, которые еще могут иметь искомое значение, уменьшается вдвое. **Для массива размера N, алгоритму может потребоваться максимум O(log(N)) шагов**, чтобы найти любой элемент или определить, что его нет в списке. Это намного быстрее, чем в случае применения алгоритма полного перебора.

Алгоритмы поиска

Двоичный поиск в упорядоченном массиве

```
//описание функции бинарного поиска
int BinarySearch(int *x, int k, int key){
  bool found = false;
  int high = k - 1, low = 0;
  int middle = (high + low) / 2;
  while ( !found && high >= low ){
    if (key == x[middle])
      found = true;
    else if (key < x[middle])
      high = middle - 1;
    else
      low = middle + 1;
    middle = (high + low) / 2;
  }
  return found ? middle : -1 ;
}
```



Двоичный поиск элемента со значением 44

Алгоритмы поиска

6. Интерполяционный поиск (interpolation search).

Двоичный поиск обеспечивает значительное увеличение скорости поиска по сравнению с полным перебором. Он **исключает большие части списка, не проверяя при этом значения исключаемых элементов**.

Если известно, что **значения элементов распределены достаточно *равномерно***, то **можно исключать на каждом шаге еще больше элементов**, используя интерполяционный поиск.

Интерполяция, интерполирование — в вычислительной математике способ нахождения промежуточных значений величины по имеющемуся дискретному набору известных значений.

При интерполяционном поиске **индексы известных значений в списке используются для определения возможного положения искомого элемента**.

Алгоритмы поиска

Интерполяционный поиск

Линейная интерполяция — интерполяция алгебраическим двучленом $P1(x) = a\,x + b$ функции f, заданной в двух точках $x0$ и $x1$ отрезка $[a, b]$. В случае, если заданы значения в нескольких точках, функция f заменяется кусочно-линейной функцией.

Геометрически это означает замену графика функции f прямой, проходящей через точки $(x0, f(x0))$ и $(x1, f(x1))$.

Уравнение такой прямой имеет вид: $$\frac{y - f(x_0)}{f(x_1) - f(x_0)} = \frac{x - x_0}{x_1 - x_0}$$

Отсюда для $x \in [x0, x1]$

$$f(x) \approx y = P_1(x) = f(x_0) + \frac{f(x_1) - f(x_0)}{x_1 - x_0}(x - x_0)$$

Эта соотношение называется формулой линейной интерполяции.

Алгоритмы поиска

Интерполяционный поиск

Пусть список (массив) содержит **20 элементов** со значениями **между 1** и **70**. Требуется **найти** элемент, имеющий значение **44**. **Считаем, что значения элементов распределены равномерно. Используя линейную интерполяцию, составим пропорцию:**

(44-1) / (70-1) = (ind-1) / (20-1), откуда ind≅13, элемент = 36.

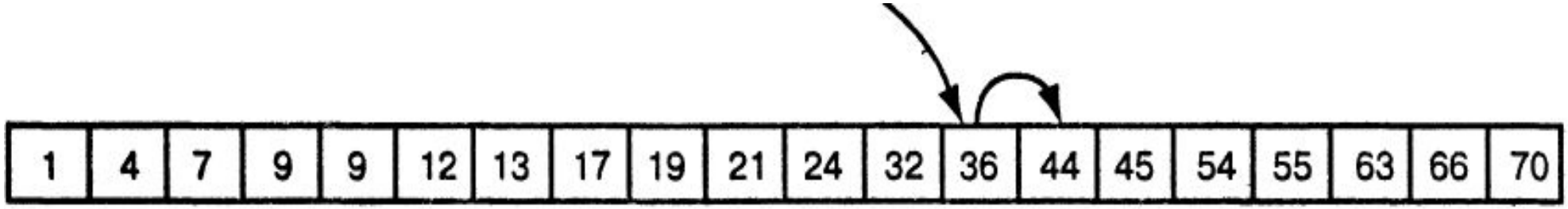
Если **позиция**, выбранная при помощи интерполяции, **оказывается неправильной**, то алгоритм **сравнивает искомое значение со значением элемента в выбранной позиции**.

Если искомое значение меньше, то поиск продолжается в первой части списка, если больше — во второй части.

В примере: 44>36, ищем во второй части,

(44-36) / (70-36) = (ind-13) / (20-13), откуда ind≅15, элемент =45.

44<45; 13< ind <15, следовательно ind=14, элемент =44.



Интерполяционный поиск значения 44

Алгоритмы поиска

Интерполяционный поиск

При двоичном поиске список последовательно разбивается посередине на две части. Интерполяционный поиск каждый раз разбивает список, пытаясь найти ближайший к искомому элемент в списке, при этом **точка разбиения определяется следующим образом:**

middle = round(minind + (target – List[minind]) *

((maxind - minind) / (List[maxind] – List[minind]))).

Замечания

1. **Перед выполнением операции деления необходимо проверить условие List(maxind) = List(minind).** Если это так, значит осталось только одно значение сравнить с искомым.

2. **Новое значение middle может выйти из диапазона между minind и maxind** только в том случае, если **искомое значение выходит за пределы диапазона от List(minind) до List(maxind).** При вычислении нового значения middle алгоритм вначале проверяет, находится ли новое значение между minind и maxind. Если нет, то искомого элемента нет в списке, и работа алгоритма завершена.

Алгоритмы поиска

Интерполяционный поиск

Интерполяционный поиск выполняется вообще говоря быстрее двоичного.

Один шаг интерполяционного поиска уменьшает число рассматриваемых записей с n до n^(1/2), если ключи распределены в таблице случайным образом. В результате интерполяционный поиск требует в среднем ln (ln n) шагов для уменьшения диапазона проверки от n до 2 (Кнут Д. Искусство программирования. Сортировка и поиск).

Разница между значениями ln (ln n) и ln n становится значительной только при больших n.

Интерполяционный поиск обычно применяют на ранних стадиях поиска в больших внешних файлах. После того как диапазон поиска существенно уменьшится, переходят к бинарному поиску.

Алгоритмы поиска

Интерполяционный поиск

```
function InterpolationSearch(target : Longint; List : PLongArray;
                             min, max : Longint) : Longint;
var
   middle : Longint;
begin
   while (min <= max) do
   begin
       // Предотвращение деления на нуль.
       if (list^[min] = list^[max]) then
       begin
           // Это должен быть искомый элемент (если он есть в списке).
           if List[min] = target then
               Result := min
           else
               Result := 0;
               exit;
       end;

       // Вычисление точки деления.
       middle := Round(min + ((target - list^[min])*
                 ((max - min) / (list^[max] - list^[min]))));
```

Алгоритмы поиска

Интерполяционный поиск

```
    // Удостовериться, что мы не вышли за пределы диапазона.
    if ((middle < min) or (middle > max)) then
    begin
        // Элемента в списке нет.
        Result := 0;
        exit;
    end;

    if target = List[middle] then              // Элемент найден.
    begin
        Result := middle;
        exit;
    end else if target < List[middle] then
        // Перебор левой половины.
        max := middle - 1
    else
        // Перебор правой половины.
        min := middle + 1;
  end;                                  // Конец while (min <= max) do...

  // Если мы достигли этой точки, то элемента в списке нет.
  Result := 0;
end;
```

Алгоритмы поиска

Следящий поиск

Пусть **в списке требуется найти много различных элементов**, и известно, что **элементы будут близки друг другу**. Вместо того, чтобы начинать поиск, проверяя заново весь список, **можно использовать результаты предыдущего поиска** и начать поиск поблизости от искомой позиции.

Двоичное отслеживание и поиск

Чтобы начать **двоичный следящий поиск** (binary hunt and search), **сравним искомое значение из предыдущего поиска с новым искомым значением**.

Если новое значение меньше, начнем слежение влево, если больше — вправо.

Алгоритмы поиска

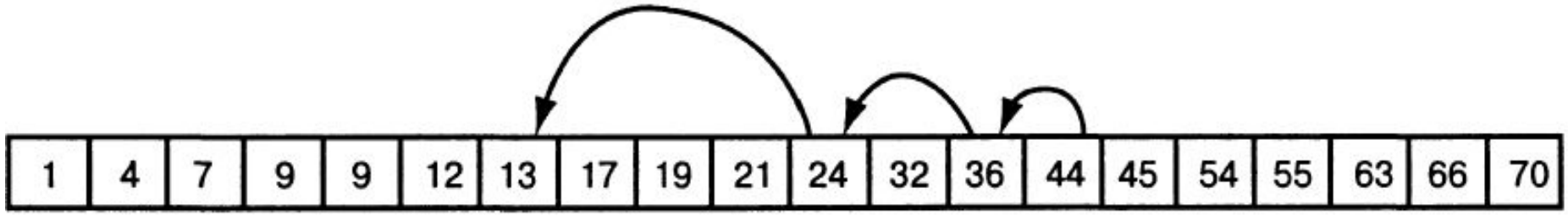
Двоичное отслеживание и поиск

Слежение влево. Установим значения переменных **min и max равными индексу, полученному во время предыдущего поиска (**max = min).

Затем **уменьшим значение min на единицу (**min = min – 1) **и сравним искомое значение со значением элемента List[min]**. Если они равны, элемент райден.

Если искомое значение меньше, чем значение List[min], установим max = min и min = min – 2, и сделаем еще одну проверку. Если искомое значение все еще меньше List[min], установим max = min и min = min – 4, и так далее.

Продолжим устанавливать значение переменной max равным значению переменной min и **вычитать очередные степени двойки из значения переменной min** до тех пор, **пока не найдется значение min**, **для которого значение элемента List[min] будет меньше искомого значения.**



Двоичный следящий поиск значения 17 из значения 44

Алгоритмы поиска

Двоичное отслеживание и поиск

Если в какой-то момент min окажется меньше, чем нижняя граница массива, то min нужно присвоить значение нижней границы массива. Если при этом значение элемента List[min] все еще больше искомого, **значит искомого элемента нет в списке.**

Слежение вправо выполняется аналогично. Вначале значения переменных min и max устанавливаются равными значению индекса, полученного во время предыдущего поиска. Затем последовательно устанавливается min = max и max = max + 1, min = max и max = max + 2, min = max и max = max + 4, и так далее до тех пор, **пока в какой-то точке значение элемента массива List[max] не станет больше искомого**.

И снова необходимо следить за тем, чтобы не выйти за границу массива.

После завершения фазы слежения известно, что индекс искомого элемента находится между min и max. После этого **можно использовать обычный двоичный поиск** для нахождения точного положения искомого элемента.

Алгоритмы поиска

Двоичное отслеживание и поиск

Когда выгодно использовать двоичный следящий поиск?

Если новый и старый искомые элементы отстоят друг от друга на P позиций, то **потребуется порядка log(P) шагов для следящего поиска новых значений переменных min и max** (Sn = b1 (q^k – 1) / (q – 1), b1=1, q=2, Sn=P).

Предположим, что мы начали обычный двоичный поиск без фазы слежения. Тогда потребуется порядка log(N) – log(P) шагов для того, чтобы значения min и max были на расстоянии не больше, чем P позиций друг от друга. Это означает, что **следящий поиск будет быстрее обычного двоичного поиска, если log(P) < log(N) – log(P).** Или 2 * log(P) < log(N), откуда P^2 < N.

Вывод

Следящий поиск будет выполняться быстрее, если расстояние между последовательными искомыми элементами будет меньше, чем квадратный корень из числа элементов в списке.

Если следующие друг за другом искомые элементы расположены далеко друг от друга, то лучше использовать обычный двоичный поиск.

Алгоритмы поиска

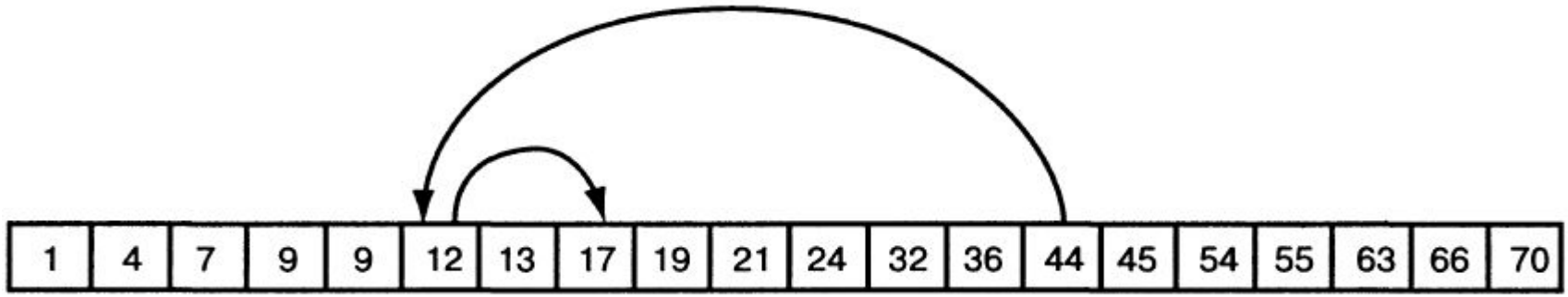
Интерполяционный следящий поиск (interpolative hunt and search)

Вначале **сравним искомое значение из предыдущего поиска с новым**. Если **новое искомое значение меньше**, начнем **слежение влево**, если **больше — вправо**.

Для **слежения влево** будем теперь использовать интерполяцию, чтобы предположить, где может находиться искомое значение **в диапазоне между значением элемента List[1] и предыдущим значением**. Это будет просто интерполяционный поиск, в котором min = 1 и max равен индексу, полученному во время предыдущего поиска.

После первого шага, фаза слежения заканчивается и дальше можно продолжить обычный интерполяционный поиск.

Слежение вправо выполняется аналогично. Устанавливаем min равным индексу, полученному во время предыдущего поиска, и max = N. Затем продолжаем обычный интерполяционный поиск.



Интерполяционный поиск значения 17 из значения 44

Алгоритмы поиска

Интерполяционный следящий поиск (interpolative hunt and search)

Использование предыдущего значения может помочь в случае, если данные распределены равномерно.

Если известно, что новое искомое значение находится близко к старому, **интерполяционный поиск, начинающийся с предыдущего значения**, обязательно найдет элемент, который находится рядом с предыдущим найденным.

Алгоритмы поиска

Индексно-последовательный поиск

Пусть исходный массив (файл) K отсортирован. Разобьем его на блоки, содержащие не более m элементов.

Построим дополнительный массив INDEX размером s=|(n-1) /m| +1, каждый элемент которого состоит из ключа kindex и указателя pindex на запись с ключом в исходном массиве. В массив INDEX помещаем максимального представителя каждого блока и указатель на него.

Сначала находим первый элемент в INDEX, ключ которого не меньше ключа элемента поиска key < INDEX.kindex[j].

Затем находим запись в массиве K с ключом, равным аргументу поиска на участке между K[INDEX. pindex[j] + 1] < key < K[INDEX. pindex[j+1]] .

В худшем случае (при неудачном поиске) число операций s+n/s.

Пример. Массив K: 2,5,6,9,12,|16,18,20,21,28,|32,39,44,46,51,|55,60

Массив INDEX: 12|28|51|60

4| 9 |14|16. Найти 44.

Алгоритмы поиска

Рекомендации по использованию алгоритмов поиска

1. Если элементы находятся **в связном списке**, используйте **поиск методом полного перебора**. По возможности используйте **сигнальную метку в конце списка** для ускорения поиска.

2. Если требуется время от времени проводить поиск **в списке, содержащем десятки элементов, то есть в небольшом списке**, также используйте **поиск методом полного перебора**.

3. Используя **двоичный или интерполяционный поиск**, можно очень быстро находить элементы даже **в очень больших упорядоченных списках**.

4. Если **значения данных в очень больших упорядоченных списках распределены достаточно равномерно**, то **интерполяционный поиск** обеспечит наилучшую производительность.

5. Если **список остается неизменным**, то **применение упорядоченного списка и использование метода интерполяционного поиска** даст прекрасные результаты.

Алгоритмы поиска

Рекомендации по использованию алгоритмов поиска

6. **Если список находится на диске или каком-либо другом медленном устройстве**, разница в скорости между интерполяционным поиском и другими методами поиска может быть достаточно велика.

7. Если используются **строковые данные**, можно попытаться **закодировать их числами в формате integer, long или double**, при этом для их поиска можно будет использовать **интерполяционный метод**.

8. Если **строки слишком длинные** и не помещаются даже в числа формата double, то проще всего может оказаться использовать **двоичный поиск**.

9. Если **требуется вставлять и удалять элементы из большого списка**, следует рассмотреть возможность замены его на другую структуру данных - **сбалансированные деревья**, вставка и добавление элемента в которые требует времени порядка **O(log(N)).**

В то время как вставка или удаление элемента из упорядоченного списка займет время порядка O(N).

Алгоритмы поиска

Рекомендации по использованию алгоритмов поиска

10. Если **требуется часто вставлять и удалять элементы из списка** и **при этом также нужно выводить элементы по порядку или перемещаться по списку в прямом или обратном направлении**, то оптимальную скорость и гибкость может обеспечить применение **сбалансированных деревьев**.

11. Если **требуется часто вставлять и удалять элементы из списка**, то можно рассмотреть возможность применения **хеш-таблицы**.

- Они обеспечивают высокую скорость выполнения этих операций, при этом **используется дополнительное пространство для хранения промежуточных данных**.
- **В хеш-таблицу можно вставлять,** из нее можно **удалять,** и в ней можно **находить элементы**, но сложно вывести элементы из таблицы по порядку.
- **Хеш-таблицы не хранят информацию о порядке расположения данных**.

Алгоритмы поиска

Преимущества и недостатки различных методов поиска

Метод	Преимущества	Недостатки
Полный перебор	Прост Высокая скорость для небольших списков	Низкая скорость для больших списков
Двоичный поиск	Высокая скорость для больших списков Не зависит от распределения данных Просто обрабатывает строковые данные	Более сложен, чем полный перебор
Интерполяционный поиск	Очень высокая скорость для больших списков	Очень сложен Данные должны быть распределены равномерно Сложно работать со строковыми данными