Сравнение линейного, бинарного и интерполяционного поисков на массиве и бинарном дереве поиска

Косяков Матвей 253505

Введение

При работе с большими объемами данных возникает необходимость в эффективных алгоритмах поиска. Линейный поиск, бинарный поиск и интерполяционный поиск являются основными методами поиска, которые могут быть применены для решения этой задачи. В данном докладе мы сравним эти три метода поиска на массиве и бинарном дереве поиска, чтобы выяснить их преимущества и недостатки, а также определить в каких ситуациях каждый метод является наиболее эффективным.

Массив - это линейная и однородная элементарная структура данных, представляющая собой непрерывную упорядоченную последовательность ячеек памяти и предназначенная для хранения данных обычно одного и того же типа.

Асимптотики:

- Чтение $O(1)$
- Вставка $O(n)$
- Удаление $O(n)$

Линейный поиск

Линейный поиск является простейшим методом поиска. Он последовательно перебирает все элементы массива, сравнивая искомый элемент с каждым элементом массива до тех пор, пока не будет найдено совпадение или пока не будет пройден весь массив. Преимущества линейного поиска включают его простоту реализации и работу на неупорядоченных данных. Однако его основной недостаток заключается в том, что время выполнения линейного поиска пропорционально размеру массива, что делает его неэффективным для больших массивов данных.
Асимптотика: O(n)

```python
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return None
```

$$T(n) = T(n-1) + c,\ n \geq 2,$$
$$T(1) = 1$$

Бинарный поиск

Бинарный поиск является эффективным методом поиска в отсортированном массиве. Он работает путем деления массива пополам и сравнения искомого элемента с элементом в середине массива. Если элемент равен искомому, то поиск завершается. Если искомый элемент меньше, чем элемент в середине массива, то поиск продолжается только в левой половине массива. В случае, если искомый элемент больше, чем элемент в середине массива, поиск продолжается только в правой половине массива. Процесс повторяется до тех пор, пока не будет найден искомый элемент или массив не будет исчерпан.
Асимптотика: O(log(n))

```python
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return None
```

$$T(n) = T(n/2) + c, n \geq 2,$$
$$T(1) = 1$$

Интерполяционный поиск

Поиск происходит подобно двоичному поиску, но вместо деления области поиска на две части, интерполяционный поиск производит оценку новой области поиска по расстоянию между ключом и текущим значением элемента. Использование интерполяционного поиска оправдано в том случае, когда данные в массиве, в котором происходит поиск, распределены достаточно равномерно.
Асимптотика: O(log(log(n)))

```python
def interpolation_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low ⩽ high and target ⩾ arr[low] and target ⩽ arr[high]:
        pos = low + ((high - low) // (arr[high] - arr[low])) * (target - arr[low])

        if arr[pos] == target:
            return pos
        elif arr[pos] < target:
            low = pos + 1
        else:
            high = pos - 1

    return None
```

Бинарное дерево поиска представляет собой структуру данных, которая обеспечивает эффективный поиск, вставку и удаление элементов. Оно состоит из узлов, каждый из которых содержит значение и два потомка - левого и правого. Бинарное дерево поиска упорядочено таким образом, что значение в левом поддереве меньше значения в корне, а значение в правом поддереве больше значения в корне.

Асимптотики:

- Чтение O(log(n))
- Вставка O(log(n))
- Удаление O(log(n))

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


class BinarySearchTree:
    def __init__(self):
        self.root = None

    def insert(self, data):
        if self.root is None:
            self.root = Node(data)
        else:
            node = self.root
            while True:
                if data < node.data:
                    if node.left is None:
                        node.left = Node(data)
                        break
                    else:
                        node = node.left
                else:
                    if node.right is None:
                        node.right = Node(data)
                        break
                    else:
                        node = node.right
```

Линейный поиск в дереве

```python
def tree_linear_search(root, data):
    if root is None:
        return None

    stack = [root]

    while stack:
        node = stack.pop()
        if node.data == data:
            return node
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)

    return None
```

Этот метод выполняет поиск в ширину (breadth-first search) путем обхода дерева по уровням с использованием стека.
Начиная с корневого узла, проверяется значение данных узла. Если оно совпадает с искомым значением, возвращается найденный узел.
Если значение данных не совпадает, происходит добавление правого и левого потомка узла в стек (если они существуют).
Алгоритм продолжает проверку узлов до тех пор, пока стек не опустеет или не будет найден узел с искомым значением.
Асимптотика: O(n)

Бинарный поиск в дереве

Этот метод выполняет бинарный поиск в отсортированном двоичном дереве поиска.
Начиная с корневого узла, проверяется значение данных узла. Если оно совпадает с искомым значением, возвращается найденный узел.
Если значение данных меньше искомого значения, поиск продолжается в правом поддереве, иначе - в левом поддереве.
Алгоритм повторяется на соответствующем поддереве до тех пор, пока узел с искомым значением не будет найден.
Асимптотика: O(log(n))

```
def tree_binary_search(root, data):
    while root is not None:
        if root.data == data:
            return root
        elif data < root.data:
            root = root.left
        else:
            root = root.right
    return None
```

Интерполяционный поиск в дереве

Этот метод использует интерполяцию для приближенного определения местоположения искомого значения в дереве.
Начиная с корневого узла, проверяется значение данных узла. Если оно совпадает с искомым значением, возвращается найденный узел.
Если значение данных меньше искомого значения, поиск продолжается в правом поддереве, иначе - в левом поддереве.
Алгоритм повторяется на соответствующем поддереве, и выбирается новый узел для проверки на основе интерполяции значения.
Интерполяция выполняется путем оценки пропорционального расстояния между значениями данных в узлах.
Алгоритм продолжает поиск, используя интерполяцию, пока узел с искомым значением не будет найден.
Асимптотика: O(log(n))

```python
def tree_interpolation_search(root, data):
    if root is None:
        return None

    node = root
    while node is not None:
        if node.data == data:
            return node
        elif data < node.data:
            if node.left is None:
                return None
            node_low = node.left
            node_high = node
        else:
            if node.right is None:
                return None
            node_low = node
            node_high = node.right

        if node_high.data == node_low.data:
            return None

        mid = node_low.data + (
            (data - node_low.data) * (node_high.data - node_low.data)
        ) // (node_high.data - node_low.data)
        mid_node = root
        while mid_node.data != mid:
            if mid < mid_node.data:
                mid_node = mid_node.left
            else:
                mid_node = mid_node.right

        if mid_node.data == data:
            return mid_node
        elif mid_node.data < data:
            node = mid_node.right
        else:
            node = mid_node.left

    return None
```

Программное обеспечение

Данные для графиков

	Number of Elements	Linear Search Time	Binary Search Time	Interpolation Search Time	Tree Linear Search Time	Tree Binary Search Time	Tree Interpolation Search Time
0	200.000000000	0.000014067	0.000004053	0.000001907	0.000001907	0.000001192	0.000000954
1	300.000000000	0.000021935	0.000003815	0.000000954	0.000002146	0.000000000	0.000001192
2	400.000000000	0.000029802	0.000005007	0.000001192	0.000002146	0.000000954	0.000000715
3	500.000000000	0.000038147	0.000005245	0.000002146	0.000002146	0.000000954	0.000000954
4	600.000000000	0.000046015	0.000004768	0.000000954	0.000000954	0.000001192	0.000000954
5	700.000000000	0.000054598	0.000005960	0.000000954	0.000001907	0.000000954	0.000000954
6	800.000000000	0.000062227	0.000005007	0.000001192	0.000000954	0.000000000	0.000000954
7	900.000000000	0.000071764	0.000005245	0.000000954	0.000000954	0.000000954	0.000000954
8	1000.000000000	0.000077009	0.000004768	0.000000954	0.000002146	0.000000954	0.000000954
9	1100.000000000	0.000085831	0.000009060	0.000000715	0.000000715	0.000000000	0.000000954
10	1200.000000000	0.000093937	0.000005007	0.000000954	0.000000954	0.000000954	0.000000954
11	1300.000000000	0.000101805	0.000005960	0.000001192	0.000001192	0.000000954	0.000000954
12	1400.000000000	0.000108957	0.000005245	0.000000954	0.000000715	0.000000954	0.000000000
13	1500.000000000	0.000122070	0.000006199	0.000001669	0.000001907	0.000000715	0.000000954
14	1600.000000000	0.000121832	0.000005245	0.000000954	0.000002146	0.000000954	0.000000954
15	1700.000000000	0.000134230	0.000005007	0.000001907	0.000001907	0.000000000	0.000001192
16	1800.000000000	0.000138998	0.000004768	0.000000954	0.000002146	0.000000954	0.000001192
17	1900.000000000	0.000149965	0.000005007	0.000000954	0.000001907	0.000001907	0.000002146
18	2000.000000000	0.000159979	0.000005007	0.000001669	0.000000954	0.000000954	0.000000954
19	2100.000000000	0.000161886	0.000005722	0.000001907	0.000000715	0.000000954	0.000000715

Выводы

В зависимости от характеристик данных и требуемых операций, выбор наиболее эффективного метода поиска может различаться. Линейный поиск подходит для небольших массивов или неотсортированных данных, бинарный поиск обеспечивает высокую эффективность на отсортированных данных, интерполяционный поиск может быть эффективным в случае равномерно распределенных данных. Для бинарного дерева поиска эффективен алгоритм двоичного поиска.

Спасибо за внимание!