

Государственное образовательное учреждение высшего профессионального образования

"Московский государственный технический университет имени Н.Э.Баумана"

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

**Отчет по лабораторной работе №8
курса Анализ алгоритмов
на тему "Поиск по словарю"**

Студент: Денисенко А. А., группа ИУ7-646

Преподаватели: Волкова Л.Л., Строганов Ю.В.

Москва, 2020 г.

Содержание

Введение	2
1 Описание алгоритмов	3
1.1 Линейный поиск	3
1.2 Частотный анализ	3
1.3 Бинарный поиск	4
1.4 Поиск по бинарному дереву	6
2 Примеры работы программы	8
Заключение	10

Введение

Одна из наиболее часто встречающихся в программировании задач – поиск. Существует несколько основных вариантов поиска, и для них создано много различных алгоритмов. Целью данной работы является получение навыка реализации алгоритмов эффективного поиска по словарю. Для достижения поставленной цели необходимо решить следующие задачи:

1. описать алгоритмы частотного анализа и поиска по дереву;
2. реализовать указанные алгоритмы;
3. привести примеры работы.

1 Описание алгоритмов

В работе будет рассмотрена фиксированная группа данных, в которой необходимо найти заданный элемент. В общем случае множество a из N элементов задано в виде массива типа, который описывает запись с некоторым полем key , играющим роль ключа. Задача заключается в поиске элемента, ключ которого равен заданному аргументу поиска x . Полученный в результате индекс i , удовлетворяющий условию $a[i].key = x$, обеспечивает доступ к другим полям обнаруженного элемента. Так как целью исследования является, прежде всего, сам процесс поиска, то будем считать, что тип массива включает в себя только ключ.

1.1 Линейный поиск

Алгоритм линейного поиска представляет собой простой последовательный просмотр массива. Условия окончания поиска таковы: либо элемент найден, т. е. $a[i] = x$, либо весь массив просмотрен и совпадения не обнаружено. В листинге 1 представлен псевдокод алгоритма линейного поиска.

Листинг 1

```
i:=0;
while (i<N) and (a[i]<>x) do i:=i+1
```

В связи с малой эффективностью по сравнению с другими алгоритмами линейный поиск обычно используют только если отрезок поиска содержит очень мало элементов, тем не менее линейный поиск не требует дополнительной памяти или обработки/анализа функции, так что 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.

1.2 Частотный анализ

При работе с естественным языком известно, что у разных букв разные частоты. Опираясь на этот факт, можно реорганизовать словарь таким образом, чтобы при поиске при обходе дерева попасть в более частотный сегмент быстрее. Расположим часть словаря, начинающуюся с более частотной буквы, ближе к началу, тем самым попытаемся снизить трудоемкость поиска для случая с высокой вероятностью - таким образом вклад в трудоемкость в среднем снизится. Для формирования вектора признаков текста, будем считать количество повторов в тексте каждой буквы, стоящей в первой позиции.

Схема частотного анализа выглядит следующим образом.

1. построить словарь V всех букв текста T , стоящих в первой позиции
2. подсчитать количество вхождений каждой из букв
3. отсортировать словарь V по убыванию значений
4. реорганизовать исходный словарь в соответствии со словарем V

Для ускорения поиска для каждой фиксированной первой буквы слова решено было использовать бинарный поиск.

В листинге 2 представлена реализация частотного анализа.

Листинг 2

```
alph = [0]*33
for i in range(len(_dict)):
    alph[ord(_dict[i][0][0]) - 1040] += 1

keys = []

for i in range(len(alph)):
    tmp = max(zip(alph, count()))[1]
    if max(alph) != 0:
        alph[tmp] = 0
        keys.append(tmp)

index = 0

for i in range(len(_dict)):
    _dict[i] = [_dict[i], i]

for i in range(len(keys)):
    for j in range(len(_dict)):
        if ord(_dict[j][0][0][0]) - 1040 == keys[i]:
            tmp = _dict.pop(j)
            _dict.insert(index, tmp)
            index += 1
```

1.3 Бинарный поиск

Поиск можно организовать значительно более эффективно, если данные будут предварительно упорядочены.

Основная идея алгоритма заключается в выборе случайно некоторого элемента, предположим a_m , и сравнении его с аргументом поиска x . Если он равен x , то поиск заканчивается, если он меньше x , то делается вывод, что все элементы с индексами, меньшими или равными m , можно исключить из дальнейшего поиска; если же он больше x , то исключаются индексы больше и равные m . Выбор m совершенно не влияет на корректность алгоритма, но влияет на его эффективность. Очевидно, что чем большее количество элементов исключается на каждом шаге алгоритма, тем этот алгоритм эффективнее. Оптимальным решением будет выбор среднего элемента, так как при этом в любом случае будет исключаться половина массива.

В этом алгоритме используются две индексные переменные L и R, которые отмечают соответственно левый и правый конец секции массива a, где еще может быть обнаружен требуемый элемент. В листинге 3 представлен псевдокод алгоритма.

Листинг 3

```

L:=0; R:=N-1; Found:=false;
while(L<=R) and not Found do begin
    m:=(L+R) div 2;
    if a[m]=x then begin
        Found:=true
    end else begin
        if a[m]<x then L:=m+1 else R:=m-1
    end
end;
end;

```

Максимальное число сравнений для этого алгоритма равно $\log_2 n$, округленному до ближайшего целого. Таким образом, приведенный алгоритм существенно выигрывает по сравнению с линейным поиском, ведь там ожидаемое число сравнений – $N/2$.

В листинге 4 представлена реализация сортировки словаря по второй букве (для каждого "фрагмента" словаря, начинающегося с фиксированной буквы" и бинарный поиск.

Листинг 4

```

alph = copy.deepcopy(alph_copy)
alph_copy = sorted(alph_copy)
alph_copy.reverse()

ind = 0
for k in range(len(alph_copy)):
    for i in range(ind, ind+alph_copy[k]):
        for j in range(ind, ind+alph_copy[k]-1):
            if _dict[j][0][1] > _dict[j+1][0][1]:
                _dict[j], _dict[j+1] = _dict[j+1], _dict[j]
        ind += alph_copy[k]

.....

first = 0
while (word[0] != _dict[first][0][0]):
    first += 1

last = first + alph[ord(word[0]) - 1040] - 1

mid = (first + last) // 2

while _dict[mid][0] != word and first < last:
    if word > _dict[mid][0]:
        first = mid + 1
    else:
        last = mid - 1
    mid = (first + last) // 2

if first > last:

```

```

    print("Слово не найдено")
else:
    res = []
    tmp = mid-1
    while _dict[mid][0][1] == word[1]:
        if _dict[mid][0] == word:
            res.append(_dict[mid][1])
        mid += 1
    mid = tmp
    while _dict[mid][0][1] == word[1]:
        if _dict[mid][0] == word:
            res.append(_dict[mid][1])
        mid -= 1
    if len(res) == 0:
        print("Слово не найдено")
    else:
        print("Слово было найдено в позициях = ", res)

```

1.4 Поиск по бинарному дереву

Бинарное дерево — это иерархическая структура данных, в которой каждый узел имеет значение (оно же является в данном случае и ключом) и ссылки на левого и правого потомка. Узел, находящийся на самом верхнем уровне (не являющийся чьим либо потомком) называется корнем. Узлы, не имеющие потомков, называются листьями.

Бинарное дерево поиска — это бинарное дерево, обладающее дополнительными свойствами: значение левого потомка меньше значения родителя, а значение правого потомка больше значения родителя для каждого узла дерева. То есть, данные в бинарном дереве поиска хранятся в отсортированном виде. При поиске элемента сравнивается искомое значение с корнем. Если искомое больше корня, то поиск продолжается в правом потомке корня, если меньше, то в левом, если равно, то значение найдено. Пример бинарного дерева представлен на рисунке 1.

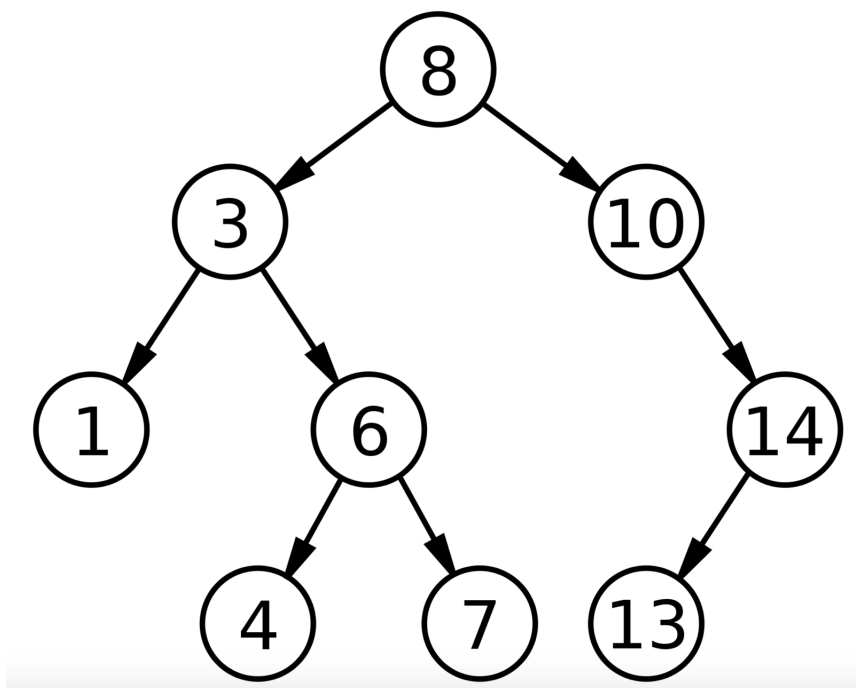


Рисунок 1 : Пример бинарного дерева поиска

Сбалансированное бинарное дерево поиска — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. Дерево считается сбалансированным, если для каждой вершины выполняется требование: высота левого и правого поддеревьев различаются не более, чем на 1. Алгоритм поиска элементов в таком дереве имеет сложность, пропорциональную $O(\log n)$.

В листинге 5 представлен листинг используемых в программе функций для работы с бинарным деревом.

Листинг 5

```

class Tree:
    def __init__(self, value, index, left = None, right = None):
        self.value = value
        self.index = index
        self.left = left
        self.right = right

    def __str__(self):
        return str(self.value)

def balance_search(tree, value):
    if tree == None: return False
    if tree.value == value:
        res = []
        res.append(tree.index)
        tmp = tree.left
        while tmp != None and tmp.value == value:
            res.append(tmp.index)
            tmp = tmp.left
  
```



```

        return res
    if value < tree.value:
        return balance_search(tree.left, value)
    else:
        return balance_search(tree.right, value)

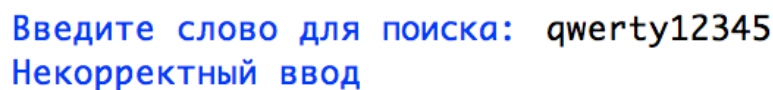
def build_tree(array):
    if len(array) == 1:
        return Tree(array[0][0], array[0][1])
    mid = len(array) // 2

    if len(array) != 2:
        return Tree(array[mid][0], array[mid][1], build_tree(array[:mid]), build_tree(array[mid+1:]))
    else:
        return Tree(array[mid][0], array[mid][1], build_tree(array[:mid]))

```

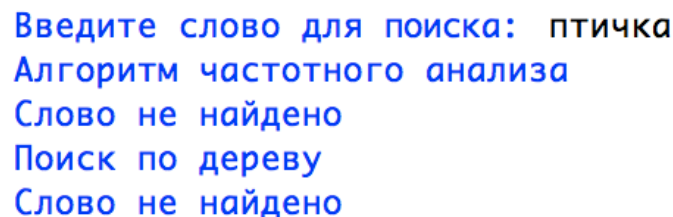
2 Примеры работы программы

Примеры работы программы приведены на рисунках 2-5.



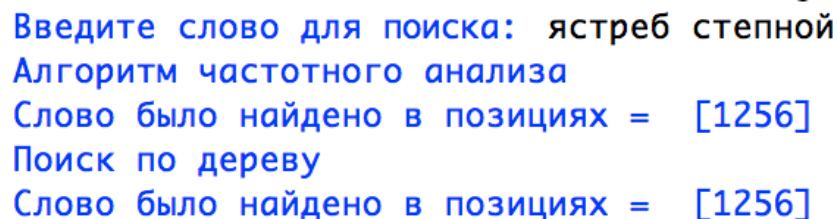
Введите слово для поиска: qwerty12345
Некорректный ввод

Рисунок 2 : Пример работы программы при некорректном вводе



Введите слово для поиска: птичка
Алгоритм частотного анализа
Слово не найдено
Поиск по дереву
Слово не найдено

Рисунок 3 : Пример работы программы со словом, отсутствующим в словаре



Введите слово для поиска: ястреб степной
Алгоритм частотного анализа
Слово было найдено в позициях = [1256]
Поиск по дереву
Слово было найдено в позициях = [1256]

Рисунок 4 : Пример работы программы со словом, которое встречается в словаре однажды

Введите слово для поиска: **авдотка**
Алгоритм частотного анализа
Слово было найдено в позициях = [199, 9]
Поиск по дереву
Слово было найдено в позициях = [199, 9]

Рисунок 5 : Пример работы программы программы со словом, которое встречается в словаре более одного раза

Заключение

В ходе данной работы были изучены алгоритмы поиска по бинарному сбалансированному дереву, алгоритм бинарного поиска и частотного анализа. Была теоретически проанализирована трудоемкость и эффективность указанных алгоритмов и осуществлен реализованы алгоритмы поиска с логарифмической сложностью.