# Министерство образования и науки Российской Федерации

# Федеральное государственное бюджетное образовательное учреждение высшего образования «Новосибирский государственный технический университет»



## Кафедра параллельных вычислительных технологий

## Домашнее задание № 3 по дисциплине «Методы проектирования и анализа алгоритмов»

**Алгоритмы поиска**

Вариант 12

|  |  |  |
| --- | --- | --- |
|  | Факультет: | ПМИ |
| Группа: | ПМ-53 |
| Студент: | Тябин Е.А. |
| Преподаватель: | Щукин Г.А. |

Новосибирск

2016

# Условие задачи

Дано множество записей о людях. Каждая запись содержит поля: имя, фамилия, день месяца, месяц и год рождения.

Требуется реализовать указанную в варианте структуру данных (сбалансированное дерево поиска или хэш-таблицу) и операции для работы над ней (вставка и поиск элементов; элемент - запись о человеке). Если совпадающие ключи не допускаются, при вставке нового элемента с существующим ключом он замещает старый элемент; при поиске возвращается один элемент. Если совпадающие ключи допускаются, при вставке нового элемента с существующим ключом он не замещает старый элемент, а “добавляется” к нему; при поиске возвращается вся группа элементов для указанного ключа.  Правильность работы алгоритмов проверить с помощью юнит-тестов.

Для реализованной структуры данных замерить время поиска на данных разного размера (использовать генерацию входных данных из предыдущей работы). Сначала все входные данные добавляются в структуру по указанным ключам, далее производится поиск  заданного числа ключей (например, 10000 или 100000), время которого замеряется. Ключи для поиска также сгенерировать самостоятельно. Время поиска для тех же данных и ключей сравнить со временем линейного поиска в обычном массиве/списке.

Для деревьев поиска также найти высоту получившегося дерева, для хэш-таблицы со списками - минимальную, максимальную и среднюю длину списка, для хэш-таблицы с прямой адресацией - процент заполненности таблицы и минимальное, максимальное и среднее число проверок при поиске элементов.

12. АВЛ-дерево. Ключ: фамилия. Совпадающие ключи допускаются.

# -Алгоритм решения

**Алг** search(key){

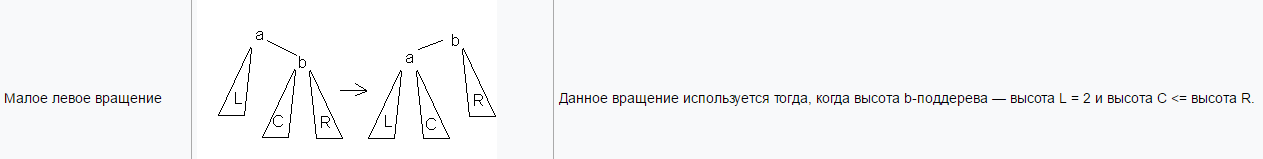
Если корень есть, то  
 Если ключ корня = ключу:  
 Возвращение элемента  
 Иначе, если ключ<ключа корня:  
 search(key) в левом поддереве  
 Иначе, если ключ>ключа корня:  
 search(key) в правом поддереве  
Иначе:  
 Такого человека нету  
}

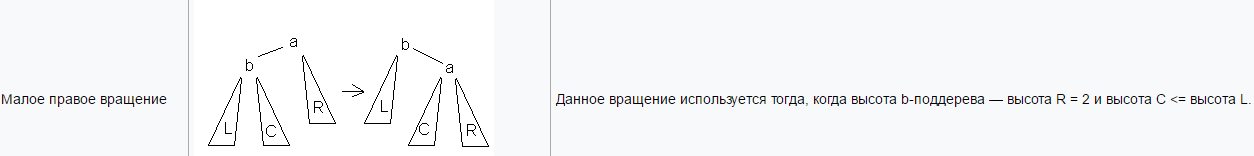
**Алг** insert(key){

Если корня нет, создаём вместе с поддеревьями  
Иначе, если Фамилия(поле-ключ)<Фамилии корня:  
 insert(key) в левое поддерево  
Иначе, если Фамилия>Фамилии корня:  
 insert(key) в правое поддерево  
Иначе:  
 Соединяем однофамильцев в одном элементе  
Балансировка дерева

}

Краткая иллюстрация алгоритма балансировки:





# Текст программы

**/\* Файл DZ.py \*/**

**from** bstree **import** BSTree  
**from** timer **import** time\_us  
**from** generate **import** read\_file, write\_file  
**from** util **import** shuffled\_ints, random\_ints, powers\_of  
**import** random, math  
  
outputdebug = **False  
  
def** debug(msg):  
 **if** outputdebug:  
 print (msg)  
  
**class** Node():  
 **def** \_\_init\_\_(self, key):  
 self.key = key[1]  
 self.people=**' '**.join(key)  
 self.left = **None** self.right = **None  
  
  
class** AVLTree():  
 **def** \_\_init\_\_(self, \*args):  
 self.node = **None** self.height = -1  
 self.balance = 0  
  
 **if** len(args) == 1:  
 **for** i **in** args[0]:  
 self.insert(i)  
  
 **def** height(self):  
 **if** self.node:  
 **return** self.node.height  
 **else**:  
 **return** 0  
  
 **def** is\_leaf(self):  
 **return** (self.height == 0)  
  
 **def** insert(self, key):  
 tree = self.node  
 newnode = Node(key)  
  
 **if** tree == **None**:  
 self.node = newnode  
 self.node.left = AVLTree()  
 self.node.right = AVLTree()  
 debug(**"Inserted key ["** + str(key) + **"]"**)  
  
 **elif** key[1] < tree.key:  
 self.node.left.insert(key)  
  
 **elif** key[1] > tree.key:  
 self.node.right.insert(key)  
  
 **else**:  
 self.node.people=self.node.people+**' '**+**' '**.join(key)  
  
 self.rebalance()  
  
 **def** rebalance(self):  
 *'''  
 Rebalance a particular (sub)tree  
 '''  
 # key inserted. Let's check if we're balanced* self.update\_heights(**False**)  
 self.update\_balances(**False**)  
 **while** self.balance < -1 **or** self.balance > 1:  
 **if** self.balance > 1:  
 **if** self.node.left.balance < 0:  
 self.node.left.lrotate() *# we're in case II* self.update\_heights()  
 self.update\_balances()  
 self.rrotate()  
 self.update\_heights()  
 self.update\_balances()  
  
 **if** self.balance < -1:  
 **if** self.node.right.balance > 0:  
 self.node.right.rrotate() *# we're in case III* self.update\_heights()  
 self.update\_balances()  
 self.lrotate()  
 self.update\_heights()  
 self.update\_balances()  
  
  
  
 **def** rrotate(self):  
 *# Rotate left pivoting on self* debug (**'Rotating '** + str(self.node.key) + **' right'**)  
 A = self.node  
 B = self.node.left.node  
 T = B.right.node  
  
 self.node = B  
 B.right.node = A  
 A.left.node = T  
  
  
 **def** lrotate(self):  
 *# Rotate left pivoting on self* debug (**'Rotating '** + str(self.node.key) + **' left'**)  
 A = self.node  
 B = self.node.right.node  
 T = B.left.node  
  
 self.node = B  
 B.left.node = A  
 A.right.node = T  
  
  
 **def** update\_heights(self, recurse=**True**):  
 **if not** self.node == **None**:  
 **if** recurse:  
 **if** self.node.left != **None**:  
 self.node.left.update\_heights()  
 **if** self.node.right != **None**:  
 self.node.right.update\_heights()  
  
 self.height = max(self.node.left.height,  
 self.node.right.height) + 1  
 **else**:  
 self.height = -1  
  
 **def** update\_balances(self, recurse=**True**):  
 **if not** self.node == **None**:  
 **if** recurse:  
 **if** self.node.left != **None**:  
 self.node.left.update\_balances()  
 **if** self.node.right != **None**:  
 self.node.right.update\_balances()  
  
 self.balance = self.node.left.height - self.node.right.height  
 **else**:  
 self.balance = 0  
  
 **def** search(self, key):  
 **if** self.node != **None**:  
 **if** self.node.key == key:  
 **return** self.node.people  
 **elif** key < self.node.key:  
 **return** self.node.left.search(key)  
 **elif** key > self.node.key:  
 **return** self.node.right.search(key)  
 **else**:  
 **return 'There is no such person'  
  
 def** TreeSearch(self, keys):  
 **for** key **in** keys:  
 self.search(key)  
  
  
  
 **def** check\_balanced(self):  
 **if** self == **None or** self.node == **None**:  
 **return True** *# We always need to make sure we are balanced* self.update\_heights()  
 self.update\_balances()  
 **return** ((abs(self.balance) < 2) **and** self.node.left.check\_balanced() **and** self.node.right.check\_balanced())  
  
  
 **def** display(self, level=0, pref=**''**):  
 *'''  
 Display the whole tree. Uses recursive def.  
 TODO: create a better display using breadth-first search  
 '''* self.update\_heights() *# Must update heights before balances* self.update\_balances()  
 **if**(self.node != **None**):  
 print( **'-'** \* level \* 2, pref, self.node.people, **"["** + str(self.height) + **":"** + str(self.balance) + **"]"**, **'L' if** self.is\_leaf() **else ' '**)  
 **if** self.node.left != **None**:  
 self.node.left.display(level + 1, **'<'**)  
 **if** self.node.right != **None**:  
 self.node.right.display(level + 1, **'>'**)  
  
  
**def** CreateTree(inlist):  
 a=AVLTree()  
 **for** i **in** inlist:  
 i = i.split(**' '**)  
 a.insert(i)  
 **return** a  
  
  
*# Usage example***if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 a = AVLTree()  
 print (**"----- Inserting -------"**)  
 inlist = read\_file(**"records\_1e1.txt"**)  
 **for** i **in** inlist:  
 i=i.split(**' '**)  
 a.insert(i)  
 a.display()  
 print(a.search(**'Aarons'**))

**/\* Файл timer.py \*/**

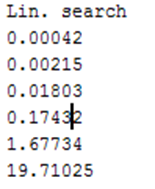
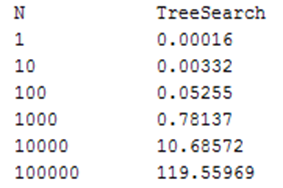
**import** timeit  
**import** DZ  
**from** generate **import** read\_file  
**from** generate **import** main  
**from** util **import** shuffled\_ints, random\_ints, powers\_of  
  
**def** make\_header(func\_names):  
 **return " "**.join([**"{0:<12}"**.format(s) **for** s **in** [**"N"**] + func\_names])  
  
  
**def** make\_line(n, times):  
 **return " "**.join([**"{0:<12}"**.format(n)] + [**"{0:<12.5f}"**.format(t) **for** t **in** times])  
  
  
**def** time\_us(ns, generator, repeats=int(1e6)):  
 *"""Prints time table for given functions and inputs.  
 functions - dictionary of {func name: func(input)} - functions to time,  
 ns - list of n for which generate input,  
 generator - func(n) - input generation function,  
 repeats - number of times to call functions for each given input."""* generator()  
 print(make\_header([**'TreeSearch'**]))  
 **for** n **in** ns:  
 data = read\_file(**"records\_1e{0}.txt"**.format(n))  
 data\_surnames= read\_file(**"records\_2e{0}.txt"**.format(n))  
 times = []  
 a=DZ.CreateTree(data)  
 timer = timeit.Timer(**lambda**: a.TreeSearch(data\_surnames))  
 times.append(timer.timeit(repeats))  
 print(make\_line(10\*\*(n-1), times))  
  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 time\_us(ns=powers\_of(1, 0, 5), generator=main, repeats=10)

# Юнит-тесты

**/\* Файл Test.py \*/**

**from** DZ **import** AVLTree  
**import** unittest  
  
  
**class** BSTreeTests(unittest.TestCase):  
 **def** setUp(self):  
 self.tree = AVLTree()  
  
 **def** test\_empty\_tree(self):  
 self.assertEqual(self.tree.search(**'Aleksandrova'**), **'There is no such person'**)  
  
 **def** test\_insert\_and\_search(self):  
 self.tree.insert([**'Dori'**, **'Kovalcheck'**, **'2'**, **'Mar'**, **'1952'**])  
 self.assertEqual(self.tree.search(**'Kovalcheck'**), **'Dori Kovalcheck 2 Mar 1952'**)  
  
 **def** test\_insert\_and\_search\_more(self):  
 self.tree.insert([**'Dori'**, **'Kovalcheck'**, **'2'**, **'Mar'**, **'1952'**])  
 self.tree.insert([**'Kasi'**, **'Hughett'**, **'14'**, **'Nov'**, **'1927'**])  
 self.tree.insert([**'Kimberlie'**, **'Loncaric'**, **'12'**, **'Sept'**, **'1954'**])  
 self.assertEqual(self.tree.search(**'Hughett'**), **'Kasi Hughett 14 Nov 1927'**)  
  
 **def** test\_balance(self):  
 self.tree.insert([**'Aleks'**, **'Akonechnikov'**, **'4'**, **'Jun'**, **'1997'**])  
 self.tree.insert([**'Babka'**, **'Bozhko'**, **'5'**, **'Mar'**, **'1996'**])  
 self.tree.insert([**'Vuasya'**, **'Durakov'**, **'27'**, **'Sep'**, **'1997'**])  
 self.assertEqual(self.tree.height, 1)  
  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 unittest.main()

# Результаты замера времени работы алгоритма



# Выводы

Проделав работу, мы успешно построили алгоритм для решения нашей задачи и реализовали его на языке программирования, после чего были сделаны юнит-тесты для проверки правильности работы алгоритма, а также сделаны замеры времени работы алгоритма. При анализе времени работы алгоритмов замечаем, что оба алгоритма возрастают по времени на порядок, однако алгоритм поиска в АВЛ возрастает наполовину от порядка больше, это обусловлено работой со строковыми типами данных. Программа выдала верное решение на всех тестах и, следовательно, правильно работает.