**Министерство образования и науки Украины**

**Национальный технический университет Украины "Киевский политехнический институт имени Игоря Сикорского"**

**Факультет информатики и вычислительной техники**

**Кафедра автоматизированных систем обработки**

**информации и управления**

**ОТЧЕТ**

по лабораторной работе № 3 по дисциплине

«Проектирование и анализ вычислительных алгоритмов»

„ **Проектирование структур данных**”

**Выполнил**

(шифр, фамилия, имя, отчество)

*ІП-61 Кушка М.О.*

**Проверил**

(фамилия, имя, отчество )

*Головченко М.Н.*

Киев 2018

Содержание

[1 Цель лабораторной работы 3](#_Toc513996284)

[2 Задание 4](#_Toc513996285)

[3 Выполнение 7](#_Toc513996286)

[3.1 Псевдокод алгоритмов 7](#_Toc513996287)

[3.2 Временная сложность поиска 9](#_Toc513996288)

[3.3 Программная реализация 9](#_Toc513996289)

[3.3.1 Примеры работы 13](#_Toc513996290)

[3.4 Испытания алгоритма 15](#_Toc513996291)

[3.4.1 Временные оценочные характеристики 15](#_Toc513996292)

[Выводы 16](#_Toc513996293)

# Цель лабораторной работы

Цель работы – изучить основные подходы к проектированию и обработке сложных структур данных.

# Задание

Согласно варианту (таблица 2.1), записать алгоритм поиска, добавления, удаления и редактирования записи в структуре данных при помощи псевдокода (или другого способа по выбору).

Записать временную сложность поиска в ней в асимптотических оценках.

Выполнить программную реализацию небольшой СУБД, с функциями поиска (алгоритм поиска в узле структуры согласно варианту таблица 2.1), добавления, удаления и редактирования записей (запись состоит из ключа и данных, ключи уникальные и целочисленные, данных можно несколько полей для одного ключа). Для хранения ключей использовать структуру данных согласно варианту (таблица 2.1).

Заполнить базу случайными значениями до 10000 и зафиксировать среднее (из 10-15 поисков) число сравнений для нахождения записи по ключу.

Сделать обобщенный вывод по лабораторной работе.

Таблица 2.1 – Варианты алгоритмов

|  |  |
| --- | --- |
| **№** | **Структура данных** |
| 1 | Файлы с плотным индексом с перестройкой индексной области, бинарный поиск |
| 2 | Файлы с плотным индексом с областью переполнения, бинарный поиск |
| 3 | Файлы с неплотным индексом с перестройкой индексной области, бинарный поиск |
| 4 | Файлы с неплотным индексом с областью переполнения, бинарный поиск |
| 5 | B-дерево t=50, бинарный поиск |
| 6 | B-дерево t=250, бинарный поиск |
| 7 | B-дерево t=500, бинарный поиск |
| 8 | B-дерево t=1000, бинарный поиск |
| 9 | Файлы с плотным индексом с перестройкой индексной области, однородный бинарный поиск |
| 10 | Файлы с плотным индексом с областью переполнения, однородный бинарный поиск |
| 11 | Файлы с неплотным индексом с перестройкой индексной области, однородный бинарный поиск |
| 12 | Файлы с неплотным индексом с областью переполнения, однородный бинарный поиск |
| 13 | B-дерево t=50, однородный бинарный поиск |
| 14 | B-дерево t=250, однородный бинарный поиск |
| 15 | B-дерево t=500, однородный бинарный поиск |
| 16 | B-дерево t=1000, однородный бинарный поиск |
| 17 | Файлы с плотным индексом с перестройкой индексной области, метод Шарра |
| 18 | Файлы с плотным индексом с областью переполнения, метод Шарра |
| 19 | Файлы с неплотным индексом с перестройкой индексной области, метод Шарра |
| 20 | Файлы с неплотным индексом с областью переполнения, метод Шарра |
| 21 | B-дерево t=50, метод Шарра |
| 22 | B-дерево t=250, метод Шарра |
| 23 | B-дерево t=500, метод Шарра |
| 24 | B-дерево t=1000, метод Шарра |
| 25 | Файлы с плотным индексом с перестройкой индексной области, интерполяционный поиск |
| 26 | B-дерево t=1000, интерполяционный поиск |
| 27 | Файлы с неплотным индексом с перестройкой индексной области, интерполяционный поиск |
| 28 | B-дерево t=500, интерполяционный поиск |
| 29 | B-дерево t=50, интерполяционный поиск |
| 30 | B-дерево t=250, интерполяционный поиск |

# Выполнение

## Псевдокод алгоритмов

class Node is

constructor Node(t) is

keys = <empty>

children = <empty>

leaf = True

\_t = t

function split(parent, payload) is

new\_node = Node(t)

mid\_point = size//2

split\_value = keys[mid\_point]

parent.add\_key(split\_value)

new\_node.children = children[mid\_point + 1:<to the end>]

children = children[0:mid\_point + 1]

new\_node.keys = keys[mid\_point+1:<to the end>]

keys = keys[0:mid\_point]

if len(new\_node.children) > 0 then

new\_node.leaf = False

parent.children = parent.add\_child(new\_node)

if payload < split\_value then

return this

else

return new\_node

function \_is\_full() is

return size == 2 \* \_t - 1

function size() is

return length(keys)

function add\_key(value) is

keys.append(value)

keys.sort()

function add\_child(new\_node) is

i = length(self.children) - 1

while i >= 0 and children[i].keys[0] > new\_node.keys[0] do

i -= 1

return children[0:i + 1]+ [new\_node] + children[i + 1:<to the end>]

class BTree is

constructor BTree(t) is

\_t = t

if \_t <= 1 then

raise Error("B-Tree must have a degree of 2 or more.")

root = Node(t)

function insert(payload) is

node = root

if node.\_is\_full is

new\_root = Node(\_t)

new\_root.children.append(root)

new\_root.leaf = False

node = node.split(new\_root, payload)

root = new\_root

while not node.leaf do

i = node.size - 1

while i > 0 and payload < node.keys[i] do

i -= 1

if payload > node.keys[i] then

i += 1

next = node.children[i]

if next.\_is\_full then

node = next.split(node, payload)

else

node = next

node.add\_key(payload)

function search(value, node=NULL) is

if node is NULL then

node = root

if value in node.keys then

return True

else if node.leaf then

return False

else

i = 0

while i < node.size and value > node.keys[i] do

i += 1

return search(value, node.children[i])

function print\_order() is

this\_level = [self.root]

while this\_level do

next\_level = []

output = ""

for node in this\_level do

if node.children then

next\_level.extend(node.children)

output += string(node.keys) + " "

print(output)

this\_level = next\_level

## Временная сложность поиска

Количество узлов в дереве: 10,000

Searching for -184. Time: 0.000029 sec. Found: True

Searching for -388. Time: 0.000016 sec. Found: True

Searching for 367. Time: 0.000018 sec. Found: True

Searching for -330. Time: 0.000026 sec. Found: True

Searching for 426. Time: 0.000024 sec. Found: True

Searching for -25. Time: 0.000014 sec. Found: True

Searching for 101. Time: 0.000014 sec. Found: True

Searching for 299. Time: 0.000009 sec. Found: True

Searching for -251. Time: 0.000035 sec. Found: True

Searching for -133. Time: 0.000049 sec. Found: True

Searching for 90. Time: 0.000031 sec. Found: True

Searching for 461. Time: 0.000030 sec. Found: True

Searching for -213. Time: 0.000007 sec. Found: True

Searching for 375. Time: 0.000038 sec. Found: True

Среднее время поиска: 0.000024 sec

## Программная реализация

from \_\_future\_\_ import (nested\_scopes, generators, division, absolute\_import,

with\_statement, print\_function, unicode\_literals)

import random

import time

class BTree:

"""

A BTree implementation with search and insert functions. Capable of any

order t.

"""

class Node:

"""A simple B-Tree Node."""

def \_\_init\_\_(self, t):

self.keys = []

self.children = []

self.leaf = True

# t is the order of the parent B-Tree. Nodes need this value to

# define max size and splitting.

self.\_t = t

def split(self, parent, payload):

"""Split a node and reassign keys/children."""

new\_node = self.\_\_class\_\_(self.\_t)

mid\_point = self.size//2

split\_value = self.keys[mid\_point]

parent.add\_key(split\_value)

# Add keys and children to appropriate nodes

new\_node.children = self.children[mid\_point + 1:]

self.children = self.children[:mid\_point + 1]

new\_node.keys = self.keys[mid\_point+1:]

self.keys = self.keys[:mid\_point]

# If the new\_node has children, set it as internal node

if len(new\_node.children) > 0:

new\_node.leaf = False

parent.children = parent.add\_child(new\_node)

if payload < split\_value:

return self

else:

return new\_node

@property

def \_is\_full(self):

return self.size == 2 \* self.\_t - 1

@property

def size(self):

return len(self.keys)

def add\_key(self, value):

"""

Add a key to a node. The node will have room for the key by

definition.

"""

self.keys.append(value)

self.keys.sort()

def add\_child(self, new\_node):

"""

Add a child to a node. This will sort the node's children, allowing

for children to be ordered even after middle nodes are split.

returns: an order list of child nodes

"""

i = len(self.children) - 1

while i >= 0 and self.children[i].keys[0] > new\_node.keys[0]:

i -= 1

return self.children[:i + 1] + [new\_node] + self.children[i + 1:]

def \_\_init\_\_(self, t):

"""

Create the B-tree. t is the order of the tree. Tree has no keys when

created. This implementation allows duplicate key values, although that

hasn't been checked strenuously.

"""

self.\_t = t

if self.\_t <= 1:

raise ValueError("B-Tree must have a degree of 2 or more.")

self.root = self.Node(t)

def insert(self, payload):

"""Insert a new key of value payload into the B-Tree."""

node = self.root

# Root is handled explicitly since it requires creating 2 new nodes

# instead of the usual one.

if node.\_is\_full:

new\_root = self.Node(self.\_t)

new\_root.children.append(self.root)

new\_root.leaf = False

# Node is being set to the node containing the ranges we want for

# payload insertion.

node = node.split(new\_root, payload)

self.root = new\_root

while not node.leaf:

i = node.size - 1

while i > 0 and payload < node.keys[i]:

i -= 1

if payload > node.keys[i]:

i += 1

next = node.children[i]

if next.\_is\_full:

node = next.split(node, payload)

else:

node = next

# Since we split all full nodes on the way down, we can simply insert

# the payload in the leaf.

node.add\_key(payload)

def search(self, value, node=None):

"""Return True if the B-Tree contains a key that matches the value."""

if node is None:

node = self.root

if value in node.keys:

return True

elif node.leaf:

# If we are in a leaf, there is no more to check.

return False

else:

i = 0

while i < node.size and value > node.keys[i]:

i += 1

return self.search(value, node.children[i])

def print\_order(self):

"""Print an level-order representation."""

this\_level = [self.root]

while this\_level:

next\_level = []

output = ""

for node in this\_level:

if node.children:

next\_level.extend(node.children)

output += str(node.keys) + " "

print(output)

this\_level = next\_level

def main():

t = 1000

tree = BTree(t)

size = 10000

my\_range = [-500, 500]

for \_ in range(size):

tree.insert(random.randint(my\_range[0], my\_range[1]))

for \_ in range(15):

number = random.randint(my\_range[0], my\_range[1])

begin = time.time()

found = tree.search(number)

end = time.time()

print(

"Searching for {:4}. Time: {:6f} sec. Found: {}".format(

number, end - begin, found

)

)

if \_\_name\_\_ == "\_\_main\_\_":

main()

### Примеры работы

На рисунке 3.1 показан пример работы программы для добавления и поиска записей.

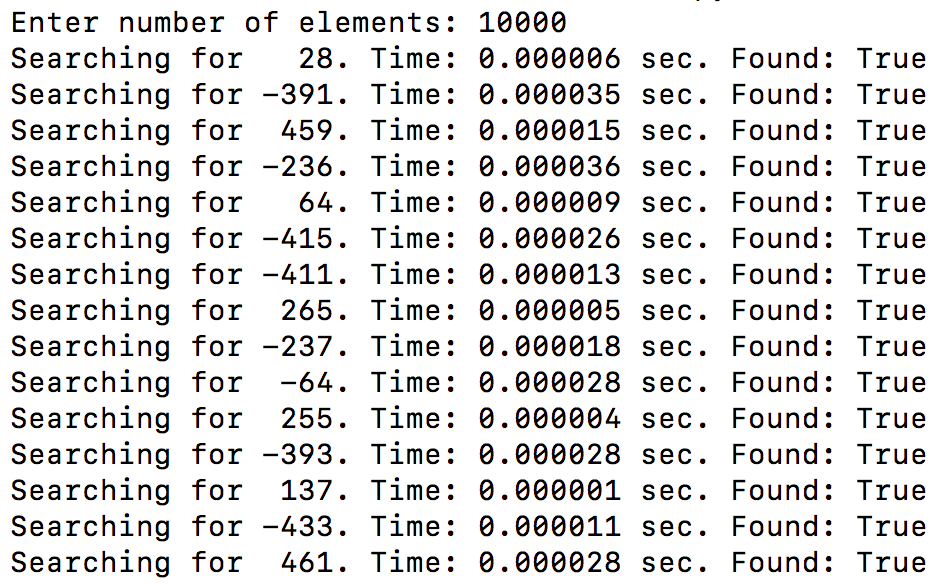


Рисунок 3.1 –Добавление и поиск записей

## Испытания алгоритма

### Временные оценочные характеристики

В таблице 3.1 приведено количество сравнений для 15 попыток поиска записи по ключу.

Таблица 3.1 – Число сравнений при попытке поиска записи по ключу

|  |  |
| --- | --- |
| Номер попытки поиска | Число сравнений |
| 1 | 10 |
| 2 | 12 |
| 3 | 2 |
| 4 | 14 |
| 5 | 6 |
| 6 | 4 |
| 7 | 14 |
| 8 | 8 |
| 9 | 14 |
| 10 | 13 |
| 11 | 2 |
| 12 | 12 |
| 13 | 12 |
| 14 | 12 |
| 15 | 4 |

Выводы

В рамках данной лабораторной работы я разобрался с B-деревом, его особенностями реализации при расширении и обработке. Реализация метода была осуществлена при помощи языка программирования Python3, поскольку на нем легко писать программный код, что позволяет сосредоточиться непосредственно на особенностях алгоритма.