МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ЛЬВІВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ІМЕНІ ІВАНА ФРАНКА

Факультет електроніки та комп’ютерних технологій

**Звіт**

Про виконання лабораторної роботи №1

з дисципліни «Алгоритми та структури даних»

На тему «Нелінійні списки(дерева) та бінарні дерева»

**Виконав**:

Малярчук О.В. ФЕП-24

**Перевірив**:

**Львів 2022**

**Мета роботи**: ознайомитись з структурою даних дерева та бінарні дерева, зрозуміти їх реалізацію, та навчитись реалізовувати їх в довільній мові програмування.

**Програмне забезпечення**: PyCharm.

**Теоретичні відомості:**

**Нелінійні списки (дерева):**

*Деревом* називається зв’язний граф без простих циклів (це одне з багатьох означень) .

*Дерево* можна також означити як рекурсивну структуру вузлів, яка є або порожньою, або складається з базового вузла – *кореня* (див. далі), з яким зв’язана скінчена кількість дерев, які ще називають *піддеревами* (див. далі).

Якщо дві вершини у дереві суміжні, то одну з них називають *батьком*, а іншу *сином*. За необхідності, ребро від батька до сина можна вважати орієнтованим. Вершина, що не має батька називається *коренем* дерева. Вершини, що не мають синів називаються *листками*. Вершини, які мають синів називаються *внутрішніми* (корінь також належить до внутрішніх вершин). Якщо існує шлях (орієнтований) з однієї вершини дерева в іншу, то одна з цих вершин є *предком*, а інша – *нащадком* (батько і син є частковим випадком). Нехай вершина не є листком. Тоді підграф, що містить цю вершину і всіх її нащадків називається *піддеревом* із коренем в цій вершині.

Кількість синів вершини  називають її *степенем* (позн. deg(*v*) ). Дерево називається *m-арним*, якщо жодна вершина має не більше ніж *m* синів ( deg(*v*)  *m*). Якщо кожна вершина має точно *m* синів ( deg(*v*)  *m* ), то

дерево називається *повним m-арним*.

При *m* = 2 розглядають *бінарне* та *повне бінарне* дерева. У бінарному дереві для вершини, що має двох синів розрізняють *лівого сина* та *правого сина*. Якщо вершина має тільки одного сина, то його вважають лівим. У бінарному дереві для вершин, що не є листками аналогічно вводиться поняття *лівого піддерева* та *правого піддерева*.

*Рівнем* вершини *L* називається довжина шляху від кореня до цієї вершини (рівень кореня рівний нулеві). *Висотою* дерева *H* називають довжину найдовшого шляху між вершинами в ньому (тобто максимальний рівень). Дерево називається *збалансованим*, якщо всі його листки знаходяться на одному, або сусідніх рівнях (тобто рівнях *H* та *H* – 1).

В даній роботі будемо розглядати тільки бінарні дерева.

Найпоширенішими операціями для роботи з деревами є: створення дерева, додавання вузла у дерево, видалення вузла з дерева, обхід дерева та пошук у ньому.

Створення бінарного дерева.

Розглянемо рекурсивний алгоритм створення бінарного дерева з наперед

заданою кількістю вузлів *n* . Даний алгоритм дозволяє створити симетричне

бінарне дерево з мінімальною, для заданої кількості вузлів, глибиною. На

кожному кроці рекурсії кількість вузлів у лівому і правом визначаються за допомогою двох формул: *nleft*  *n* / 2 ,*nright*  *n*  *nleft*  1.

Відображення бінарного дерева.

Розглянемо тепер рекурсивний алгоритм відображення щойно створеного бінарного дерева. Піддерево рівня *L* виводиться так: спочатку рекурсивно відображається ліве піддерево, потім корінь піддерева рівня *L*, перед яким потрібно вивести *L* пробілів, потім відображається праве піддерево.

Обхід дерева.

Задача обходу дерева полягає у одноразовому відвідуванні всіх вузлів дерева та виконання з ними певних дій (в нашому випадку – вивід на екран значення інформаційного поля (ключа) вузла). З іншого боку цю задачу можна розглядати як задання порядку (впорядкування) на множині вершин дерева.

Існує три рекурсивних алгоритми обходу дерева:

* обхід у прямому порядку (зверху вниз);
* обхід у зворотному порядку (знизу вверх);
* обхід у внутрішньому порядку (зліва направо).

**Бінарні дерева пошуку:**

*Бінарним деревом пошуку* (англ. *binary search tree* (BST)) називається бінарне дерево, що задовольняє таким умовам:

* поля даних (ключі) у різних вузлах не можуть бути однакові;
* для будь-якого вузла поля даних (ключі) вузлів його лівого піддерева є меншими, а правого – більшими за ключ цього вузла.

Подібна організація даних є зручною з точки зору їх пошуку. Алгоритмічна складність процедури пошуку у бінарному дереві пошуку становить *O*(log(*n*)) , де *n* – кількість вузлів дерева.

Розглянемо алгоритми для створення бінарного дерева пошуку, пошуку даних у ньому, додавання та видалення вузлів.

Пошук даних за ключем у бінарному дереві пошуку.

Спочатку реалізуємо процедуру пошуку за ключем, оскільки вона використовується як при додаванні вузлів у дерево, так і при їх видалені.

Створення (додавання нових вузлів) бінарного дерева пошуку.

Процедура створення бінарного дерева пошуку складається з трьох етапів: створення першого вузла (кореня), пошуку місця вставки і безпосередньо самої вставки нового вузла.

**Зміст роботи**

* Створив файл BTS.py та Tree.py, де буду реалізувати весь функціонал, який вимагається в лабораторній роботі.
* Реалізував клас Node, який буде використувуватись як нода дерева
* У файлі Tree.py реалізував клас Tree, в якому буде реалізоване 2-арне дерево. В цьому класі є наступні функції:
* \_\_init\_\_(self, n: int, data=None)
* \_ChangeData(self, n, data)
* CreateTree(self, n: int, parent: Node, data)
* Depth(self, root: Node)
* InOrderTraversal(self, root: Node)
* PreOrderTraversal(self, root: Node)
* PostOrderTraversal(self, root: Node)
* ShowTree(self, root: Node, indent: int = 0)
* У файлі BTS.py реалізував клас BTS, який репрезентує структуру даних бінарне дерево, в ньому є наступний функціонал:
* \_\_init\_\_(self)
* CreateBTS(self, data)
* \_SearchNodeBST(self, data, root: Node)
* SearchNodeBST(self, data)
* \_ShowTree(self, root: Node, indent: int = 0)
* ShowTree(self, indent: int = 0)
* InsertNodeBST(self, data)
* \_InsertNode(self, data)
* DelNode(self, data)
* SuccessorNodeBST(self, root: Node)
* PredecessorNodeBST(self, root: Node)
* \_maximum(root: Node)
* \_minimum(root: Node)

1. В файлі main.py прописав імпорти класів Tree з файлу Tree.py та BTS з файлу BTS.py.

**Висновок:** на даній лабораторній роботі я ознайомився з найпоширенішими операціями для роботи з 2-арними та бінарними деревами, розглянув алгоритми для створення бінарного дерева пошуку. Також, навчився реалізовувати функції для них .

Додаток 1.

INDENTATON = 6

class Node:

def \_\_init\_\_(self, data=None, parent=None, left=None, right=None):

self.data = data

self.parent = parent

self.left = left

self.right = right

class Tree:

def \_\_init\_\_(self, n: int, data=None):

if isinstance(data, list):

self.arr = data

self.index = 0

self.\_ChangeData(n=n, data=data)

self.root = Node(data=self.data)

self.\_ChangeData(n=n-1, data=self.arr)

self.CreateTree(n=n // 2, parent=self.root, data=self.arr)

self.CreateTree(n=n - n // 2 - 1, parent=self.root, data=self.arr)

else:

self.data = n

self.root = Node(data=self.data)

self.\_ChangeData(n=n, data=self.data)

self.CreateTree(n=n // 2, parent=self.root, data=self.data)

self.CreateTree(n=n - n // 2, parent=self.root, data=self.data)

def \_ChangeData(self, n, data):

if isinstance(data, int):

self.data = self.data - 1

elif isinstance(data, list):

try:

self.data = self.arr[self.index]

except IndexError:

return None

else:

self.index += 1

# print(self.data, end=' -!-')

def CreateTree(self, n: int, parent: Node, data):

if n == 0:

return None

else:

new\_node = Node(data=self.data, parent=parent)

if parent.left is None:

parent.left = new\_node

else:

parent.right = new\_node

self.\_ChangeData(n=n, data=data)

self.CreateTree(n=n // 2, parent=new\_node, data=data)

self.CreateTree(n=n - n // 2 - 1, parent=new\_node, data=data)

def Depth(self, root: Node):

if root.left is None and root.right is None:

return 1

elif root.left is None:

return self.Depth(root=root.right) + 1

elif root.right is None:

return self.Depth(root=root.left) + 1

else:

return max(self.Depth(root=root.left), self.Depth(root=root.right)) + 1

def InOrderTraversal(self, root: Node):

if root is None:

return

self.InOrderTraversal(root.left)

print(root.data)

self.InOrderTraversal(root.right)

def PreOrderTraversal(self, root: Node):

if root is None:

return

print(root.data)

self.PreOrderTraversal(root.left)

self.PreOrderTraversal(root.right)

def PostOrderTraversal(self, root: Node):

if root is None:

return

self.PostOrderTraversal(root.left)

self.PostOrderTraversal(root.right)

print(root.data)

def ShowTree(self, root: Node, indent: int = 0):

if root is None:

return

indent += INDENTATON

self.ShowTree(root.right, indent=indent)

print()

for i in range(INDENTATON, indent):

print(end=" ")

print(root.data)

self.ShowTree(root.left, indent=indent)

(base) mihajlofemak@MacBook-Pro Lab\_2 % cat BTS.py

INDENTATON = 3

class Node:

def \_\_init\_\_(self, data=None, parent=None, left=None, right=None):

self.data = data

self.parent = parent

self.left = left

self.right = right

class BTS:

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

self.root = None

def CreateBTS(self, data):

self.root = Node(data=data)

def \_SearchNodeBST(self, data, root: Node):

if root is None:

return None

if root.data == data:

return root

if root.data > data:

return self.\_SearchNodeBST(data=data, root=root.left)

else:

return self.\_SearchNodeBST(data=data, root=root.right)

def SearchNodeBST(self, data):

return self.\_SearchNodeBST(data=data, root=self.root)

def \_ShowTree(self, root: Node, indent: int = 0):

if root is None:

return

indent += INDENTATON

self.\_ShowTree(root.right, indent=indent)

print()

for i in range(INDENTATON, indent):

print(end=" ")

print(root.data)

self.\_ShowTree(root.left, indent=indent)

def ShowTree(self, indent: int = 0):

print('-'\*20)

self.\_ShowTree(root=self.root, indent=indent)

print('-'\*20)

def InsertNodeBST(self, data):

if self.SearchNodeBST(data=data) is not None:

return False

self.\_InsertNode(data=data)

return True

def \_InsertNode(self, data):

prev\_node = None

node: Node = self.root

while node is not None:

if node.data > data:

prev\_node = node

node = node.left

else:

prev\_node = node

node = node.right

new\_node = Node(data=data, parent=prev\_node)

if prev\_node.data > data:

prev\_node.left = new\_node

elif prev\_node.data < data:

prev\_node.right = new\_node

def DelNode(self, data):

node: Node = self.SearchNodeBST(data=data)

if node is None:

return False

if node.left is None and node.right is None:

parent: Node = node.parent

if parent.left == node:

parent.left = None

else:

parent.right = None

del node

elif node.left is None or node.right is None:

parent: Node = node.parent

if node.left is not None:

child: Node = node.left

elif node.right is not None:

child: Node = node.right

if parent.left == node:

parent.left = child

else:

parent.right = child

del node

else:

sucessor = self.SuccessorNodeBST(root=node)

sucessor\_data= sucessor.data

self.DelNode(sucessor.data)

node.data = sucessor\_data

def SuccessorNodeBST(self, root: Node):

if root is None:

return None

if root.right is not None:

return self.\_minimum(root=root.right)

y: Node = root.parent

while y is not None and root is y.right:

root = y

y = root.parent

return y

def PredecessorNodeBST(self, root: Node):

if root is None:

return None

if root.left is not None:

return self.\_maximum(root=root.left)

y: Node = root.parent

while y is not None and root is y.left:

root = y

y = root.parent

return y

@staticmethod

def \_minimum(root: Node):

while root.left is not None:

root = root.left

return root

@staticmethod

def \_maximum(root: Node):

while root.right is not None:

root = root.right

return root%