**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

КУРСОВАЯ РАБОТА

**по дисциплине «Алгоритмы и структуры данных»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 1302 |  | Новиков Г.В. |
| Преподаватель |  | Родионова Е.А. |

Санкт-Петербург

2022

**ЗАДАНИЕ**

**на курсовую работу**

|  |  |  |
| --- | --- | --- |
| Студент Новиков Г.В. | | |
| Группа 1302 | | |
| Тема работы: бинарные деревья | | |
| Задание:  Реализовать вычисление арифметических выражений с  бинарными операциями {+, -, \*, /, ^} и числовыми операндами,  записанных в префиксной/инфиксной/постфиксной записи. | | |
| Содержание пояснительной записки:  Аннотация, содержание, введение, теория, оценка сложности, пример работы, листинг, ссылка на код, заключение, список используемых источников. | | |
| Предполагаемый объем пояснительной записки:  Не менее 13 страниц. | | |
| Дата выдачи задания: 14.11.2022 | | |
| Дата сдачи реферата: 11.12.2022 | | |
| Дата защиты реферата: 11.12.2022 | | |
| Студент гр. 1302 |  | Новиков Г.В. |
| Преподаватель |  | Родионова Е.А. |

**Аннотация**

В основе программы лежит бинарное дерево, в узлах которого находятся либо один из следующих символов: +, -, \*, /, ^, либо число типа int или float. Программа состоит из двух файлов – node.py, содержащего класс Node - узел бинарного дерева, и bin\_tree.py, содержащего класс BinTree – бинарное дерево, состоящее из узлов типа Node. Node содержит числовое значение и ссылки на 2 узла. BinTree содержит поле root – корень дерева, с помощью которого осуществляется доступ ко всем узлам.

В результате реализован класс, объекты которого могут считывать выражение, записанное в префиксной, инфиксной или постфиксиной записи, вычислять выражение, хранящееся в дереве, а также выводить дерево в консоль и вычислять его высоту.

**Summary**

The program is based on a binary tree, the nodes of which contain either one of the following symbols: +, -, \*, /, ^, or a number of type int or float. The program consists of two files - node.py, containing the Node class - a binary tree node, and bin\_tree.py, containing the BinTree class - a binary tree consisting of nodes of the Node type. Node contains a numeric value and links to 2 nodes. BinTree contains the root field - the root of the tree, through which all nodes are accessed.

As a result, a class has been implemented whose objects can read an expression written in prefix, infix, or postfix notation, evaluate an expression stored in a tree, and also print the tree to the console and calculate its height.

**СОДЕРЖАНИЕ**

Оглавление

[Введение 5](#_Toc121697540)

[Теория 6](#_Toc121697541)

[Оценка сложности 6](#_Toc121697542)

[Пример работы 6](#_Toc121697543)

[Листинг 7](#_Toc121697544)

[Ссылка на код 11](#_Toc121697545)

[Заключение 12](#_Toc121697546)

[Список использованных источников 13](#_Toc121697547)

## Введение

Цель работы - реализовать вычисление арифметических выражений с

бинарными операциями {+, -, \*, /, ^} и числовыми операндами,

записанных в префиксной/инфиксной/постфиксной записи.

Средства разработки: Python3.9.1

Метод решения: создать класс для узла и класс для бинарного дерева, состоящего из узлов. В узлах хранить либо оператор, либо число. При создании дерева разделять выражение на 2 части относительно оператора (в зависимости от вида записи) и передавать выражения рекурсивно. Для вычисления выражения использовать рекурсивный обход дерева с вычислением значений слева и справа.

## Теория

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 (оба потомка которых равны NULL) называются листьями.

## Оценка сложности

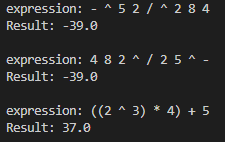
n – кол-во узлов

h – высота дерева

**BinTree:**

|  |  |
| --- | --- |
| **Метод** | **Сложность** |
| height | O(n) |
| print\_tree | O(n) |
| tree\_from\_infix | O(n) |
| tree\_from\_prefix | O(n) |
| get\_result | O(n) |

## Пример работы



## Листинг

**node.py:**

class Node:

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

self.value = value

self.left = None

self.right = None

def \_\_repr\_\_(self):

return f"<Node({self.value}) at {hex(id(self))}>"

**bin\_tree.py:**

from node import Node

def is\_number(val: str) -> bool:

if len(val) < 1:

return False

if val[0] == '-':

val = val[1:]

return val.replace('.', '', 1).isdigit()

class BinTree:

def \_\_init\_\_(self, expr: str):

self.SIGNS = ['+', '-', '\*', '/', '^']

temp\_expr = ''

for sym in expr:

if sym in ['(', ')'] + self.SIGNS:

temp\_expr = temp\_expr + ' ' + sym + ' '

else:

temp\_expr = temp\_expr + sym

expr = list(temp\_expr.split())

if expr[0] in self.SIGNS:

self.root = self.tree\_from\_prefix(expr)

elif expr[-1] in self.SIGNS:

self.root = self.tree\_from\_prefix(expr[::-1])

elif is\_number(expr[0]) or expr[0] in ['(', ')']:

self.root = self.tree\_from\_infix(expr)

else:

raise ValueError("Invalid notation")

# public:

def height(self) -> int:

# O(n)

return self.\_\_get\_height(self.root)

def tree\_from\_prefix(self, expr: list) -> Node:

# O(n)

# n - number of operands + number of numbers

if len(expr) < 3 or len(expr) == 1 and is\_number(expr[0]):

return None

if expr[0] not in self.SIGNS:

raise ValueError(f"Invalid result: {expr}")

root = Node(expr.pop(0))

if is\_number(expr[0]):

root.left = Node(expr.pop(0))

elif expr[0] in self.SIGNS:

root.left = self.tree\_from\_prefix(expr)

if is\_number(expr[0]):

root.right = Node(expr.pop(0))

elif expr[0] in self.SIGNS:

root.right = self.tree\_from\_prefix(expr)

return root

def tree\_from\_infix(self, expr: list) -> Node:

# O(n)

# n - number of operands + number of numbers

if len(expr) < 1:

return None

if not is\_number(expr[0]) and not expr[0] in ['(', ')']:

raise ValueError(f"Invalid result: {expr}")

if len(expr) == 1 and is\_number(expr[0]):

root = Node(expr[0])

elif len(expr) == 2 and is\_number(expr[0]):

raise ValueError(f"Invalid result: {expr}")

elif len(expr) == 3 and is\_number(expr[0]) and expr[1] in self.SIGNS and is\_number(expr[2]):

root = Node(expr.pop(1))

root.left = Node(expr.pop(0))

root.right = Node(expr.pop(0))

elif is\_number(expr[0]) and expr[1] in self.SIGNS and expr[2] == '(':

if ')' not in expr:

raise ValueError(f"Invalid result: {expr}")

closing\_bracket\_i = self.\_\_find\_closing\_bracket(expr, 2)

if closing\_bracket\_i < 4: # including -1

raise ValueError(f"Invalid result: {expr}")

inner\_expr = expr[3:closing\_bracket\_i]

further\_expr = []

if len(expr) > closing\_bracket\_i + 1:

further\_expr = expr[closing\_bracket\_i + 1:]

if len(further\_expr) == 0:

root = Node(expr.pop(1))

root.left = Node(expr.pop(0))

root.right = self.tree\_from\_infix(inner\_expr)

elif further\_expr[0] in self.SIGNS:

root = Node(further\_expr.pop(0))

root.left = self.tree\_from\_infix(expr[:closing\_bracket\_i + 1])

root.right = self.tree\_from\_infix(expr[closing\_bracket\_i + 1:])

else:

raise ValueError(f"Invalid result: {expr}")

elif expr[0] == '(':

if ')' not in expr:

raise ValueError(f"Invalid result: {expr}")

closing\_bracket\_i = self.\_\_find\_closing\_bracket(expr, 0)

if closing\_bracket\_i < 2 or closing\_bracket\_i == 3: # including -1

raise ValueError(f"Invalid result: {expr}")

inner\_expr = expr[1:closing\_bracket\_i]

further\_expr = []

if len(expr) > closing\_bracket\_i + 1:

further\_expr = expr[closing\_bracket\_i + 1:]

if len(further\_expr) == 0:

root = self.tree\_from\_infix(inner\_expr)

elif further\_expr[0] in self.SIGNS:

root = Node(further\_expr.pop(0))

root.left = self.tree\_from\_infix(inner\_expr)

root.right = self.tree\_from\_infix(expr[closing\_bracket\_i + 2:])

else:

raise ValueError(f"Invalid result: {expr}")

else:

raise ValueError(f"Invalid result: {expr}")

return root

def print\_tree(self, detail: bool = False) -> None:

# O(n)

print(f"Tree of height {self.height()}:")

self.\_\_print\_tree(self.root, detail=detail)

print()

def get\_result(self):

# O(n)

return self.\_\_get\_result(self.root)

# private

def \_\_print\_tree(self, root: Node, level: int = 0, child\_letter = '', detail: bool = False) -> None:

# O(n)

if root is None:

return

child\_str = child\_letter + ':' if child\_letter != '' else ''

if detail:

print(". ", ". " \* (level - 1), child\_str, root, sep='')

else:

print(". ", ". " \* (level - 1), child\_str, root.value, sep='')

self.\_\_print\_tree(root.left, level + 1, 'l', detail)

self.\_\_print\_tree(root.right, level + 1, 'r', detail)

def \_\_get\_result(self, root: Node) -> list:

# O(n)

if root is None:

return None

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

return float(root.value)

if root.left is not None and root.right is None:

raise ValueError(f"Invalid tree(root={root}, left={root.left}, right={root.right})")

if root.left is None and root.right is not None:

raise ValueError(f"Invalid tree(root={root}, left={root.left}, right={root.right})")

left = self.\_\_get\_result(root.left)

right = self.\_\_get\_result(root.right)

if left is not None and right is not None:

if root.value == '\*':

return left \* right

if root.value == '/':

return left / right

if root.value == '+':

return left + right

if root.value == '-':

return left - right

if root.value == '^':

return left\*\*(right)

else:

raise ValueError(f"Invalid tree(root={root}, left={left}, right={right})")

return None

def \_\_get\_height(self, root: Node) -> int:

# O(n)

if root is None:

return -1

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

return 0

height\_l = self.\_\_get\_height(root.left)

height\_r = self.\_\_get\_height(root.right)

return max(height\_l, height\_r) + 1

def \_\_find\_closing\_bracket(self, expr: list, opening\_br\_index: int) -> int:

# O(n)

if expr[opening\_br\_index] != '(':

raise ValueError("expr[opening\_br\_index] is not '('")

level = 1

i = 0

while i != opening\_br\_index:

if expr[i] == '(':

level += 1

i += 1

i += 1

while i < len(expr) and level != 0:

if expr[i] == '(':

level += 1

elif expr[i] == ')':

level -= 1

i += 1

if level != 0:

return -1

return i - 1

# for prefix and postfix nonation use spaces for number separation

expression = "- ^ 5 2 / ^ 2 8 4"

tree = BinTree(expression)

print("expression:", expression)

print("Result:", tree.get\_result())

print()

expression = "4 8 2 ^ / 2 5 ^ -"

print("expression:", expression)

tree = BinTree(expression)

print("Result:", tree.get\_result())

print()

# for infix notation use brackets for ordering

expression = "((2 ^ 3) \* 4) + 5"

print("expression:", expression)

tree = BinTree(expression)

print("Result:", tree.get\_result())

print()

## Ссылка на код

[GitHub](https://github.com/Gregory-hub/leti/tree/master/Algorithms%20and%20Data%20Structures/cursed/v-2)

## Заключение

При выполнении работы была написана программа для вычисления арифметических выражений, записанных в префиксной / инфиксной / постфиксной записи.

## Список использованных источников

1. Документация Python3.9.1 // Python 3.9.14 documentation. URL: [https://docs.python.org/3.9/](https://docs.python.org/3.9/%20) (дата обращения: 01.12.2022)
2. Шаблоны оформления студенческих работ СПбГЭТУ «ЛЭТИ» // Бланки заявлений и шаблоны для обучающихся // <https://etu.ru/ru/studentam/dokumenty-dlya-ucheby/blanki-zayavlenij-i-shablony-dlya-obuchayushhihsya> (дата обращения: 11.12.2022)