**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра САПР**

отчет

**по лабораторной работе №4**

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

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

Санкт-Петербург

2022

Постановка задачи

Реализовать структуру данных двоичное дерево поиска и следующие методы:

нахождение минимума, нахождение максимума, прямой (preorder),

центрированный (inorder) и обратный обход (postorder) по дереву, поиск

элемента, нахождение следующего и предыдущего элемента, удаление

элемента, обход в ширину. Реализовать визуализацию дерева. Указать

теоретическую временную сложность для всех операций.

С помощью реализованной структуры данных написать программу,

позволяющую преобразовать запись из префиксной/инфиксной/постфиксной

нотации в префиксную/инфиксную/постфиксную нотацию.

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.

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 каждой операции вставки нового или удаления существующего узла отсортированный порядок дерева сохраняется. 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 и поиск прекращается.

Сбалансированное бинарное дерево поиска — это бинарное дерево поиска с логарифмической высотой. Данное определение скорее идейное, чем строгое. Строгое определение оперирует разницей глубины самого глубокого и самого неглубокого листа (в AVL-деревьях) или отношением глубины самого глубокого и самого неглубокого листа (в красно-черных деревьях). В сбалансированном бинарном дереве поиска операции поиска, вставки и удаления выполняются за логарифмическое время (так как путь к любому листу от корня не более логарифма). В вырожденном случае несбалансированного бинарного дерева поиска, например, когда в пустое дерево вставлялась отсортированная последовательность, дерево превратится в линейный список, и операции поиска, вставки и удаления будут выполняться за линейное время. Поэтому балансировка дерева крайне важна. Технически балансировка осуществляется поворотами частей дерева при вставке нового элемента, если вставка данного элемента нарушила условие сбалансированности.

Оценка сложности

n – кол-во узлов

h – высота дерева

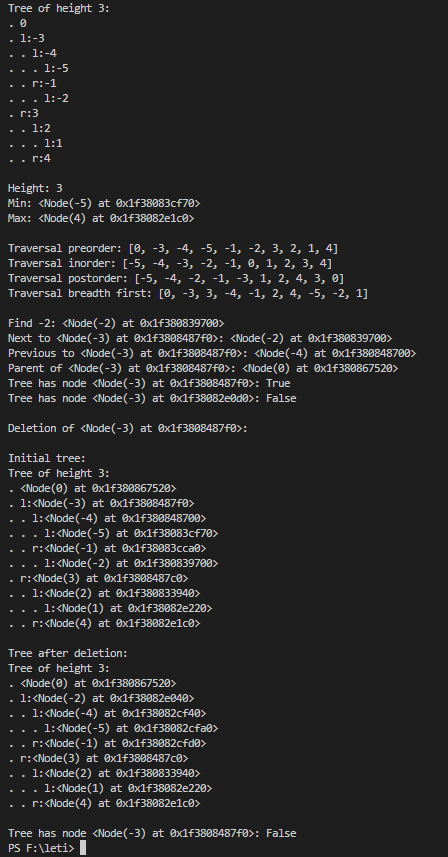
**BinSearchTree:**

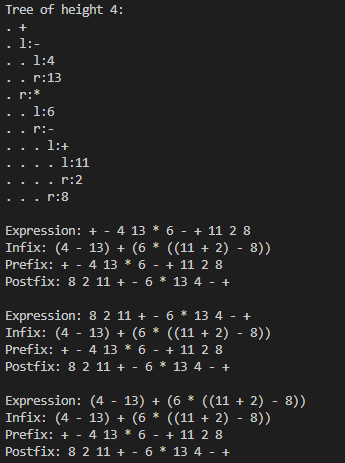
|  |  |
| --- | --- |
| **Метод** | **Сложность** |
| delete | O(n) |
| find | O(h) |
| get\_nodes\_breadth\_first | O(n) |
| get\_nodes\_inorder | O(n) |
| get\_nodes\_postorder | O(n) |
| get\_nodes\_preorder | O(n) |
| get\_parent | O(h) |
| has\_node | O(h) |
| height | O(n) |
| max | O(h) |
| min | O(h) |
| next\_el | O(h) |
| prev\_el | O(h) |
| print\_tree | O(n) |

**BinTree:**

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

Пример работы





Листинг

**bin\_search\_tree.py**

from queue import Queue

class Node:

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

self.value = value

self.left = None

self.right = None

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

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

class BinSearchTree:

def \_\_init\_\_(self, elements: list):

elements.sort()

self.root = self.\_\_tree\_from\_list(elements, len(elements))

# public:

def height(self) -> int:

# O(n)

return self.\_\_get\_height(self.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 min(self) -> Node:

# O(h)

return self.\_\_min(self.root)

def max(self) -> Node:

# O(h)

return self.\_\_max(self.root)

def get\_nodes\_inorder(self) -> list:

# O(n)

return self.\_\_get\_nodes\_inorder(self.root)

def get\_nodes\_preorder(self) -> list:

# O(n)

return self.\_\_get\_nodes\_preorder(self.root)

def get\_nodes\_postorder(self) -> list:

# O(n)

return self.\_\_get\_nodes\_postorder(self.root)

def get\_nodes\_breadth\_first(self) -> list:

# O(n)

if self.root is None:

return []

nodes = []

queue = Queue()

current = self.root

queue.put(current)

while not queue.empty():

current = queue.get()

nodes.append(current)

if current.left:

queue.put(current.left)

if current.right:

queue.put(current.right)

return nodes

def find(self, value: int) -> Node:

# O(h)

return self.\_\_find(self.root, value)

def next\_el(self, node: Node) -> Node:

# O(h)

if node is None or not self.has\_node(node):

return None

if node.right is not None:

return self.\_\_min(node.right)

return self.\_\_get\_next\_in\_ancestors(self.root, node)

def prev\_el(self, node: Node) -> Node:

# O(h)

if node is None or not self.has\_node(node):

return None

if node.left is not None:

return self.\_\_max(node.left)

return self.\_\_get\_prev\_in\_ancestors(self.root, node)

def get\_parent(self, node: Node) -> Node:

# O(h)

if node is None:

return None

return self.\_\_get\_parent(node, self.root)

def has\_node(self, node: Node) -> bool:

# O(h)

return self.\_\_has\_node(self.root, node)

def delete(self, node: Node):

# O(n)

if node is None:

return self

if not self.has\_node(node): # O(h)

return self

parent = self.get\_parent(node) # O(h)

successors = self.\_\_get\_nodes\_inorder(node.left) + self.\_\_get\_nodes\_inorder(node.right) # O(n)

if parent is None:

self.root = self.\_\_tree\_from\_list([el.value for el in successors], len(successors)) # O(n)

elif parent.left and parent.left == node:

parent.left = self.\_\_tree\_from\_list([el.value for el in successors], len(successors)) # O(n)

elif parent.right and parent.right == node:

parent.right = self.\_\_tree\_from\_list([el.value for el in successors], len(successors)) # O(n)

return self

# private:

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 \_\_tree\_from\_list(self, elements: list, n: int) -> Node:

# elements must be sorted

# O(n)

if n == 1:

return Node(elements[0])

if n == 0:

return None

root = Node(elements[n // 2])

left = self.\_\_tree\_from\_list(elements[:n // 2], n // 2)

right = self.\_\_tree\_from\_list(elements[n // 2 + 1:], n // 2 - 1 + n % 2)

root.left = left

root.right = right

return root

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 \_\_min(self, root: Node) -> Node:

# O(h)

if root is None or root.left is None:

return root

return self.\_\_min(root.left)

def \_\_max(self, root: Node) -> Node:

# O(h)

if root is None or root.right is None:

return root

return self.\_\_max(root.right)

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

# O(n)

if root is None:

return []

left = self.\_\_get\_nodes\_inorder(root.left)

right = self.\_\_get\_nodes\_inorder(root.right)

return left + [root] + right

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

# O(n)

if root is None:

return []

left = self.\_\_get\_nodes\_preorder(root.left)

right = self.\_\_get\_nodes\_preorder(root.right)

return [root] + left + right

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

# O(n)

if root is None:

return []

left = self.\_\_get\_nodes\_postorder(root.left)

right = self.\_\_get\_nodes\_postorder(root.right)

return left + right + [root]

def \_\_find(self, root: Node, value: int) -> Node:

# O(h)

if root is None:

return None

if value == root.value:

return root

if value < root.value:

return self.\_\_find(root.left, value)

if value > root.value:

return self.\_\_find(root.right, value)

def \_\_get\_next\_in\_ancestors(self, root: Node, node: Node) -> Node:

# O(h)

next\_el = None

while root and root.value != node.value:

if root.value > node.value:

next\_el = root

root = root.left

elif root.value < node.value:

root = root.right

return next\_el

def \_\_get\_prev\_in\_ancestors(self, root: Node, node: Node) -> Node:

# O(h)

next\_el = None

while root and root.value != node.value:

if root.value < node.value:

next\_el = root

root = root.right

elif root.value > node.value:

root = root.left

return next\_el

def \_\_get\_parent(self, node: Node, root: Node) -> Node:

# O(h)

if root == None:

return None

if root.right == node or root.left == node:

return root

if root.value < node.value:

return self.\_\_get\_parent(node, root.right)

elif root.value > node.value:

return self.\_\_get\_parent(node, root.left)

else:

left = self.\_\_get\_parent(node, root.left)

right = self.\_\_get\_parent(node, root.right)

if left:

return left

if right:

return right

def \_\_has\_node(self, root: Node, node: Node) -> bool:

# O(h)

if root is None:

return False

if node.value == root.value:

if node == root:

return True

return self.\_\_has\_node(root.left, node) or self.\_\_has\_node(root.right, node)

if node.value < root.value:

return self.\_\_has\_node(root.left, node)

if node.value > root.value:

return self.\_\_has\_node(root.right, node)

# tests for this are in tests.py in the same directory

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

li = list(range(-5, 5))

tree = BinSearchTree(li)

value\_to\_find = -2

el = tree.root.left

tree.print\_tree()

print("Height:", tree.height())

print("Min:", tree.min())

print("Max:", tree.max())

print()

print("Traversal preorder:", [i.value for i in tree.get\_nodes\_preorder()])

print("Traversal inorder:", [i.value for i in tree.get\_nodes\_inorder()])

print("Traversal postorder:", [i.value for i in tree.get\_nodes\_postorder()])

print("Traversal breadth first:", [i.value for i in tree.get\_nodes\_breadth\_first()])

print()

print(f"Find {value\_to\_find}:", tree.find(value\_to\_find))

print(f"Next to {el}:", tree.next\_el(el))

print(f"Previous to {el}:", tree.prev\_el(el))

print(f"Parent of {el}:", tree.get\_parent(el))

print(f"Tree has node {el}:", tree.has\_node(el))

new\_el = Node(el.value)

print(f"Tree has node {new\_el}:", tree.has\_node(new\_el))

print(f"\nDeletion of {el}:\n")

print("Initial tree:")

tree.print\_tree(detail=True)

tree.delete(el)

print("Tree after deletion:")

tree.print\_tree(detail=True)

print(f"Tree has node {el}:", tree.has\_node(el))

**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):

temp\_expr = ''

for sym in expr:

if sym in ['(', ')', '\*', '/', '+', '-']:

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

else:

temp\_expr = temp\_expr + sym

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

if expr[0] in ['+', '-', '\*', '/']:

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

elif expr[-1] in ['+', '-', '\*', '/']:

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 ['+', '-', '\*', '/']:

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

root = Node(expr.pop(0))

if is\_number(expr[0]):

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

elif expr[0] in ['+', '-', '\*', '/']:

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

if is\_number(expr[0]):

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

elif expr[0] in ['+', '-', '\*', '/']:

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 expression: {expr}")

# ( 5 - ( 3 - 1 ) \* 2 )

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 expression: {expr}")

elif len(expr) == 3 and is\_number(expr[0]) and expr[1] in ['+', '-', '\*', '/'] 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 ['+', '-', '\*', '/'] and expr[2] == '(':

if ')' not in expr:

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

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

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

raise ValueError(f"Invalid expression: {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 ['+', '-', '\*', '/']:

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 expression: {expr}")

elif expr[0] == '(':

if ')' not in expr:

raise ValueError(f"Invalid expression: {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 expression: {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 ['+', '-', '\*', '/']:

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 expression: {expr}")

else:

raise ValueError(f"Invalid expression: {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\_expression(self, notation: str):

# O(n)

if notation == "prefix":

expr = self.\_\_get\_prefix\_expression(self.root)

elif notation == "postfix":

expr = self.\_\_get\_prefix\_expression(self.root)

expr = [' ' + val.strip() for val in expr][::-1]

expr[0].strip()

elif notation == "infix":

expr = self.\_\_get\_infix\_expression(self.root)[1:-1]

else:

raise ValueError("Invalid notation")

return ''.join(expr).strip()

# 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\_infix\_expression(self, root: Node, expr: str = '') -> list:

# O(n)

if root is None:

return []

expr = []

left = self.\_\_get\_infix\_expression(root.left)

right = self.\_\_get\_infix\_expression(root.right)

l\_bracket = ''

r\_bracket = ''

if left != [] or right != []:

l\_bracket = '('

r\_bracket = ')'

if root.value in ['\*', '/', '+', '-']:

root.value = ' ' + root.value + ' '

expr = [l\_bracket] + left + [root.value] + right + [r\_bracket]

return expr

def \_\_get\_prefix\_expression(self, root: Node, expr: str = '') -> list:

# O(n)

if root is None:

return []

expr = []

left = self.\_\_get\_prefix\_expression(root.left)

right = self.\_\_get\_prefix\_expression(root.right)

for s in root.value:

if s in ['\*', '/', '+', '-']:

root.value = root.value.strip()

root.value = ' ' + root.value

expr = [root.value] + left + right

return expr

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 infix notation please use brackets for ordering

expression = "+ - 4 13 \* 6 - + 11 2 8"

tree = BinTree(expression)

tree.print\_tree()

print("Expression:", expression)

print("Infix:", tree.get\_expression(notation="infix"))

print("Prefix:", tree.get\_expression(notation="prefix"))

print("Postfix:", tree.get\_expression(notation="postfix"))

print()

expression = "8 2 11 + - 6 \* 13 4 - +"

print("Expression:", expression)

tree = BinTree(expression)

print("Infix:", tree.get\_expression(notation="infix"))

print("Prefix:", tree.get\_expression(notation="prefix"))

print("Postfix:", tree.get\_expression(notation="postfix"))

print()

expression = "(4 - 13) + (6 \* ((11 + 2) - 8))"

print("Expression:", expression)

tree = BinTree(expression)

print("Infix:", tree.get\_expression(notation="infix"))

print("Prefix:", tree.get\_expression(notation="prefix"))

print("Postfix:", tree.get\_expression(notation="postfix"))

print()

Ссылка на код

[GitHub](https://github.com/Gregory-hub/leti/blob/master/Algorithms%20and%20Data%20Structures/lab4)