**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный университет**

**Факультет прикладной математики-процессов управления**

**Кафедра “фундаментальная информатика и информационные технологии”**

**отчет**

**по лабораторной работе №4**

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

**на тему «Деревья»**

|  |  |  |
| --- | --- | --- |
| Студент гр. 22Б15-пу |  | Добренкова Л.С. |
| Преподаватель |  | Дик А.Г. |

**Санкт-Петербург**

**2024 г**

Оглавление

[Цель работы 3](#__RefHeading___Toc9662_524586906)

[Задачи 3](#__RefHeading___Toc9664_524586906)

[Теоретическая часть 3](#__RefHeading___Toc9666_524586906)

[Описание алгоритма 4](#__RefHeading___Toc9668_524586906)

[Описание программы 6](#__RefHeading___Toc9670_524586906)

[Рекомендации для пользователя 8](#__RefHeading___Toc9672_524586906)

[Контрольный пример 9](#__RefHeading___Toc8166_812848710)

[Рекомендации для программиста 9](#__RefHeading___Toc9676_524586906)

[Анализ результатов работы алгоритма 9](#__RefHeading___Toc9680_524586906)

[Вывод 13](#__RefHeading___Toc9682_524586906)

[Листинг 13](#__RefHeading___Toc8168_812848710)

# ****Цель работы****

Разработать метод решения для решения неявно поставленной задачи: «В мире кибертронских данных, где информация представлена в виде матрицы связей и цифровых путей, отсортированный односвязный список становится основой для создания мощного сетевого массива. Как современный конструктор, мы преобразуем каждый элемент списка в узел сетевой матрицы, придавая ему силу и гибкость кибертронской технологии. Постепенно, словно строя конструкцию из сетевых компонентов, мы соединяем каждый узел с другим, создавая сложные цифровые траектории между данными. В конечном итоге, из фрагментов информации формируется сбалансированный по высоте сетевой массив, готовый стать фундаментом для воплощения мощи и знаний.»

# ****Задачи****

1. Необходимо формализовать задачу
2. Выбрать метод решения.
3. Реализовать программу.

# ****Теоретическая часть****

Сбалансированные деревья — это типы деревьев, которые поддерживают сбалансированное состояние, чтобы гарантировать эффективное выполнение основных операций, таких как вставка, удаление и поиск. Под сбалансированностью обычно понимается такое состояние дерева, при котором разница в высоте между левым и правым поддеревом каждого узла минимальна.

Преимущества сбалансированных деревьев

Быстрый поиск: Благодаря логарифмической высоте деревьев поиск элементов выполняется за O(log n).

Эффективные вставка и удаление: Эти операции также выполняются за O(log n), что делает сбалансированные деревья эффективными для динамических наборов данных.

Упорядоченность данных: Элементы в деревьях хранятся в отсортированном порядке, что облегчает операции диапазонного поиска.

Метод "разделяй и властвуй" (divide and conquer) — это парадигма разработки алгоритмов, которая включает разбиение задачи на более мелкие подзадачи, решение каждой из них отдельно и комбинирование решений для получения итогового результата.

Шаги метода "разделяй и властвуй"

Разделение (Divide):

Исходная задача разбивается на несколько подзадач, которые являются меньшими экземплярами той же задачи.

Властвуй (Conquer):

Каждая подзадача решается рекурсивно. Если подзадачи достаточно малы, они решаются непосредственно.

Объединение (Combine):

Решения подзадач комбинируются для получения решения исходной задачи.

# ****Описание алгоритма****

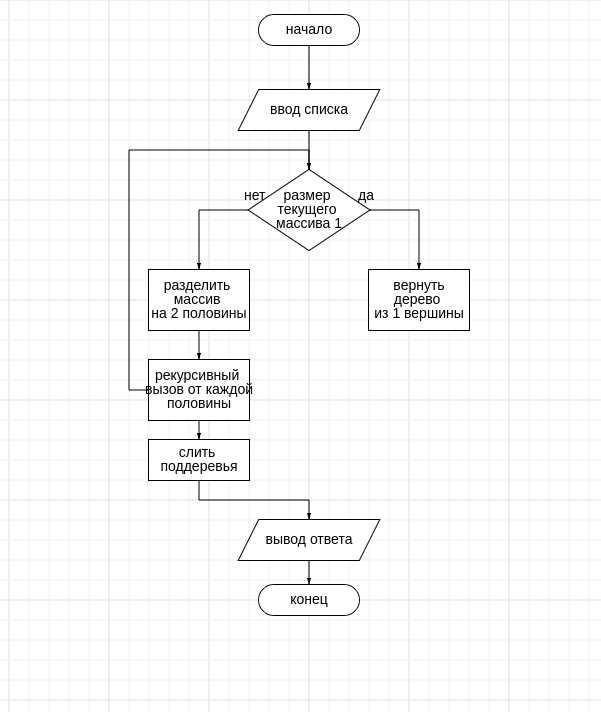
1. Инициализация отсортированного списка
2. Решение с помощью метода разделяй и властвуй

2.1 — Рекурсивное разделение отсортированного массива на 2 части

2.2 — Построение сбалансированного дерева для текущей части

2.3 — Соединение левого и правого поддеревьев

1. Вывод ответа с построением графа

Рисунок 4.1. Основной алгоритм программы

# ****Описание программы****

Программа реализована при помощи языка python 3.8.

Использованные модули: tkinter, math.

В таблицах 5.1, 5.2 представлены описания функций и методов программы.

Таблица 5.1. Описание методов класса App

|  |  |  |
| --- | --- | --- |
| **Метод** | **описание** | **Параметры** |
| add\_vertex | Добавление новой вершины | event |
| draw\_vertex | Добавление вершины на холст | vertex: dict |
| add\_edge | Добавление нового ребра | - |
| draw\_edge | Добавление ребра на холст | edge: dict, isDark: bool |
| draw | Рендеринг | - |
| display\_tree | Вычисление координат вершин | cur: TreeNode | None,  l: int, r: int, h: int |
| run | Запуск | - |

Таблица 5.2. Описание методов lib

|  |  |  |
| --- | --- | --- |
| **Метод** | **Описание** | **Параметры** |
| bst\_from\_arr | Построение дерева из массива | arr: list,  l: int,  r: int |
| bst\_from\_list | Построение дерева из списка | head: ListNode |

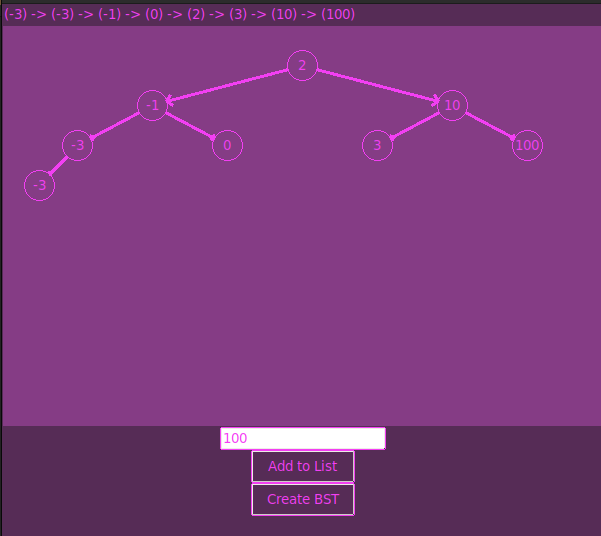
# ****Рекомендации для пользователя****

Для запуска программы необходим интерпретатор python 3.8, пакетный менеджер pip. Необходимо установить библиотеку tkinter.

Системные требования 200мб ОЗУ, 1мб ПЗУ.

Запуск файла main.py выдаст интуитивно понятный графический интерфейс. Ввод элементов списка осуществляется с помощью соответствующего поля и кнопки Add to list.

# Контрольный пример

Рисунок 7.1. Пример работы программы

# ****Рекомендации для программиста****

Для внесения изменений необходима ide для python. Код доступен по ссылке: https://github.com/v131v/alg\_labs\_3\_sem/tree/main/binary\_tree

# ****Вывод****

В рамках данной работы были исследованы сбалансированные деревья и некоторые методы их построения. Реализована программа, которая с помощью метода разделяй и властвуй строит сбалансированное дерево из отсортированного списка и отрисовывает результат на экране.

# Листинг

Файл main.py

import tkinter as tk

from tkinter import ttk

from tkinter import messagebox

from lib import \*

import math

class App(tk.Frame):

def \_\_init\_\_(self, master=None):

super().\_\_init\_\_(master)

self.master = master

self.pack(fill="both", expand=True)

self.create\_widgets()

self.head = None

self.current = None

self.\_vertex = list()

self.\_edge = list()

self.r = 15

def create\_widgets(self):

light = "#f33ff3"

dark = "#562c56"

self.light = light

self.dark = dark

styleBtn = ttk.Style()

styleBtn.theme\_use("clam")

styleBtn = styleBtn.configure(

"TButton", foreground=light, background=dark, bordercolor=light

)

styleEntry = ttk.Style()

styleEntry.theme\_use("clam")

styleEntry = styleEntry.configure(

"TEntry", foreground=light, background=dark, bordercolor=light

)

styleLabel = ttk.Style()

styleLabel.theme\_use("clam")

styleLabel = styleLabel.configure("TLabel", foreground=light, background=dark)

self.configure(bg=dark)

self.res\_label = ttk.Label(

self, text="There will be list...", foreground=light, background=dark

)

self.res\_label.pack(fill="x", expand=True)

self.canvas = tk.Canvas(

self, width=600, height=400, bg="#853c85", highlightbackground=dark

)

self.canvas.pack(side=tk.TOP)

self.entry = ttk.Entry(self)

self.entry.pack(side="top")

self.add\_button = ttk.Button(self, text="Add to List", command=self.add\_to\_list)

self.add\_button.pack(side="top")

self.bst\_button = ttk.Button(self, text="Create BST", command=self.create\_bst)

self.bst\_button.pack(side="top")

self.text = ttk.Label(self)

self.text.pack(side="bottom")

def add\_to\_list(self):

val = self.entry.get()

if (val.isdigit() or (val.startswith("-") and val[1:].isdigit())) and (

not self.current or self.current.val <= int(val)

):

val = int(val)

if not self.head:

self.head = ListNode(val)

self.current = self.head

self.res\_label.configure(text=f"({val})")

else:

self.current.next = ListNode(val)

self.current = self.current.next

self.res\_label.configure(text=self.res\_label["text"] + f" -> ({val})")

else:

messagebox.showerror(

"Error", "Please enter a valid integer more than previous"

)

def create\_bst(self):

if not self.head:

messagebox.showinfo("Info", "List is empty")

return

root = bst\_from\_list(self.head)

print(int(self.canvas["width"]))

self.\_edge = list()

self.\_vertex = list()

self.dfs(root, 0, int(self.canvas["width"]), 1)

self.draw()

self.head = None # Reset list after conversion

self.current = None

def dfs(self, cur: TreeNode | None, l: int, r: int, h: int) -> int:

if cur is None:

return None

m = (r - l) // 2 + l

cur\_id = len(self.\_vertex)

self.\_vertex.append({"x": m, "y": h \* 40, "id": cur\_id, "val": cur.val})

if cur.left is not None:

l\_id = self.dfs(cur.left, l, m, h + 1)

self.\_edge.append({"from": cur\_id, "to": l\_id})

if cur.right is not None:

r\_id = self.dfs(cur.right, m, r, h + 1)

self.\_edge.append({"from": cur\_id, "to": r\_id})

print(self.\_edge)

return cur\_id

def display\_tree(self, node, prefix="", pos="root"):

if node:

self.text.insert(

tk.END, f"{prefix[:-4]}{'+---' if prefix else ''}{pos}: {node.val}\n"

)

self.display\_tree(node.left, prefix + "| ", "L")

self.display\_tree(node.right, prefix + " ", "R")

def draw(self):

self.canvas.delete("all")

for v in self.\_vertex:

self.draw\_vertex(v)

for e in self.\_edge:

self.draw\_edge(e)

def draw\_vertex(self, vertex: dict) -> None: # рисование вершины в canvas

x, y = vertex["x"], vertex["y"]

self.canvas.create\_oval(

x - self.r, y - self.r, x + self.r, y + self.r, outline=self.light

)

self.canvas.create\_text(x, y, text=str(vertex["val"]), fill=self.light)

def draw\_edge(self, edge: dict, dark: bool = False) -> None: # отображение ребра

color = self.dark if dark else self.light

from\_vertex, to\_vertex = edge["from"], edge["to"]

from\_x, from\_y = self.get\_vertex\_coordinates(

from\_vertex

) # получение координат исходящей вершины

to\_x, to\_y = self.get\_vertex\_coordinates(

to\_vertex

) # получение координат входящей вершины

# отображение ребраweight

dx = to\_x - from\_x

dy = to\_y - from\_y

l = (dx\*\*2 + dy\*\*2) \*\* 0.5

ang = math.atan2(dy, dx)

from\_x += math.cos(ang) \* self.r

from\_y += math.sin(ang) \* self.r

to\_x -= math.cos(ang) \* self.r

to\_y -= math.sin(ang) \* self.r

l2 = l / 20

x1 = l2 \* math.cos(ang + math.pi + math.pi / 4)

y1 = l2 \* math.sin(ang + math.pi + math.pi / 4)

self.canvas.create\_line(

to\_x, to\_y, to\_x + x1, to\_y + y1, width=3, fill=color, tag="line"

)

x1 = l2 \* math.cos(ang + math.pi - math.pi / 4)

y1 = l2 \* math.sin(ang + math.pi - math.pi / 4)

self.canvas.create\_line(

to\_x, to\_y, to\_x + x1, to\_y + y1, width=3, fill=color, tag="line"

)

self.canvas.create\_line(

from\_x, from\_y, to\_x, to\_y, width=3, fill=color, tag="line"

)

line = self.canvas.create\_line(

from\_x, from\_y, to\_x, to\_y, width=3, fill=color, tag="line"

)

self.canvas.tag\_lower(line)

def get\_vertex\_coordinates(self, vertex: int) -> tuple: # получение координат точки

return (self.\_vertex[vertex]["x"], self.\_vertex[vertex]["y"])

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

root = tk.Tk()

app = App(master=root)

app.mainloop()

Файл lib.py

class ListNode:

def \_\_init\_\_(self, val=0, next=None):

self.val = val

self.next = next

class TreeNode:

def \_\_init\_\_(self, val=0, left=None, right=None):

self.val = val

self.left = left

self.right = right

def bst\_from\_arr(arr: list, l: int, r: int):

if r - l < 1:

return None

if r - l == 1:

return TreeNode(arr[l].val)

m = (r - l) // 2 + l

return TreeNode(arr[m].val, bst\_from\_arr(arr, l, m), bst\_from\_arr(arr, m + 1, r))

def bst\_from\_list(head: ListNode):

arr = []

cur = head

while cur:

arr.append(cur)

cur = cur.next

return bst\_from\_arr(arr, 0, len(arr))

ls = ListNode(-3, ListNode(-1, ListNode(0, ListNode(5, ListNode(10, ListNode(11))))))

bst = bst\_from\_list(ls)