Министерство образования и науки РФ

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский технологический университет «МИСиС»

Институт ИТКН

Курсовая работа на тему:

«Декартово дерево»

Выполнила:

студентка группы БИВТ-23-2

Сорокина Маргарита Алексеевна

Москва 2025

Содержание

[1. Введение 3](#_Toc199362584)

[2. Теоретические основы 3](#_Toc199362585)

[2.1. Определение декартова дерева 3](#_Toc199362586)

[2.2. Свойства и принципы работы 4](#_Toc199362587)

[3. Базовые операции 4](#_Toc199362588)

[3.1. Разделение (split) 4](#_Toc199362589)

[3.2. Объединение (merge) 5](#_Toc199362590)

[3.3. Вставка узла 5](#_Toc199362591)

[3.4. Удаление узла 5](#_Toc199362592)

[3.5. Поиск 6](#_Toc199362593)

[4. Реализация на языке Python 6](#_Toc199362594)

[5. Графическая визуализация 9](#_Toc199362595)

[6. Сравнение с другими структурами данных 14](#_Toc199362596)

[7. Преимущества и недостатки 14](#_Toc199362597)

[8. Практическое применение 15](#_Toc199362598)

[9. Заключение 16](#_Toc199362599)

# 1. Введение

Современные алгоритмы и структуры данных являются фундаментом эффективной обработки и хранения информации. Среди них важную роль играют сбалансированные бинарные деревья, позволяющие обеспечивать логарифмическую сложность основных операций: вставки, удаления и поиска. Одной из таких структур является декартово дерево, или treap — уникальное сочетание бинарного дерева поиска и кучи, которое балансируется с помощью случайных приоритетов.

Treap интересен своей простотой реализации, математической элегантностью и практической эффективностью. Несмотря на то, что сбалансированность дерева достигается не строгим контролем высоты, как в AVL- или красно-черных деревьях, а благодаря случайным приоритетам, это обеспечивает стабильную работу в среднем случае, что делает treap удобным в задачах, где важна как эффективность, так и простота реализации.

Цель данной курсовой работы — изучить структуру данных «декартово дерево», её внутренние принципы, ключевые операции и сравнение с другими структурами. Также будет представлена реализация на языке программирования Python и рассмотрено возможное практическое применение treap.

# 2. Теоретические основы

## 2.1. Определение декартова дерева

Декартово дерево (или treap, от слов tree и heap) — это бинарное дерево, в котором каждый узел хранит два значения:

• Ключ — значение, по которому дерево упорядочено, как в обычном бинарном дереве поиска.

• Приоритет — случайное число, определяющее структуру кучи (heap).

Таким образом, декартово дерево удовлетворяет одновременно двум свойствам:

1. Свойство бинарного дерева поиска: для любого узла все элементы в левом поддереве имеют ключи меньше ключа узла, а в правом — больше.

2. Свойство кучи по приоритету: приоритет любого узла больше приоритета его потомков.

## 2.2. Свойства и принципы работы

Treap можно получить, если вставлять элементы в пустое дерево, используя алгоритм, который:

• ищет место для вставки по ключу, как в бинарном дереве поиска;

• при необходимости делает поворот (разделение и объединение), чтобы соблюсти свойство кучи.

Сбалансированность достигается за счёт случайных приоритетов. С вероятностью, близкой к 1, глубина дерева будет O(log n), если приоритеты выбраны случайно и независимо.

Такая реализация сочетает простоту и эффективность, что делает treap хорошим выбором в задачах, где важны быстрое выполнение операций и простота кода.

# 3. Базовые операции

Treap реализует пять основных операций, обеспечивая логарифмическую сложность в среднем случае. Рассмотрим каждую подробнее.

## 3.1. Разделение (split)

Операция split(T, x) разбивает дерево T на два дерева — L и R:

• Все элементы в L имеют ключи меньше x.

• Все элементы в R — ключи не меньше x.

Алгоритм работает рекурсивно, разбивая дерево по ключу и сохраняя при этом свойства treap.

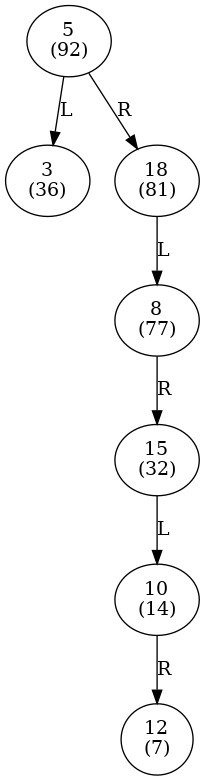


Рисунок 1- Графическая реализация декартова дерева с разбиением по ключу и сохранением при этом свойства treap.

## 3.2. Объединение (merge)

Операция merge(L, R) объединяет два treap’а L и R, в которых все ключи в L меньше, чем в R. Объединение выполняется по приоритетам: корнем нового дерева становится тот из корней L или R, у которого выше приоритет.

## 3.3. Вставка узла

Операция insert(T, x) вставляет узел с ключом x в treap T. Выполняется сначала как обычная вставка в бинарное дерево поиска, после чего, при необходимости, дерево восстанавливается с помощью операций split и merge для соблюдения свойства кучи.

## 3.4. Удаление узла

Операция erase(T, x) удаляет узел с ключом x. Если найденный узел имеет двух потомков, он заменяется результатом merge его левого и правого поддеревьев.

## 3.5. Поиск

Поиск в treap аналогичен бинарному дереву поиска и требует O(log n) операций в среднем случае.

# 4. Реализация на языке Python

Пример реализации treap с основными операциями:

import random

class Node:

def \_\_init\_\_(self, key, priority=None):

self.key = key

self.priority = priority if priority else random.randint(1, 1000000)

self.left = None

self.right = None

def split(root, key):

if not root:

return (None, None)

if key < root.key:

left, root.left = split(root.left, key)

return (left, root)

else:

root.right, right = split(root.right, key)

return (root, right)

def merge(left, right):

if not left or not right:

return left or right

if left.priority > right.priority:

left.right = merge(left.right, right)

return left

else:

right.left = merge(left, right.left)

return right

def insert(root, node):

if not root:

return node

if node.priority > root.priority:

left, right = split(root, node.key)

node.left, node.right = left, right

return node

elif node.key < root.key:

root.left = insert(root.left, node)

else:

root.right = insert(root.right, node)

return root

def delete(root, key):

if not root:

return None

if root.key == key:

return merge(root.left, root.right)

elif key < root.key:

root.left = delete(root.left, key)

else:

root.right = delete(root.right, key)

return root

def search(root, key):

if not root:

return False

if root.key == key:

return True

elif key < root.key:

return search(root.left, key)

else:

return search(root.right, key)

Пример использования:

root = None

for key in [10, 20, 5, 15]:

root = insert(root, Node(key))

print(search(root, 15)) # True

root = delete(root, 10)

print(search(root, 10)) # False

# 5. Графическая визуализация

Для графической визуализации были использованы библиотеки Python: import random,

import networkx, import matplotlib.pyplot.

Пример кода:

import random

import networkx as nx

import matplotlib.pyplot as plt

class Node:

def \_\_init\_\_(self, key, priority=None):

self.key = key

self.priority = priority if priority else random.randint(1, 1000000)

self.left = None

self.right = None

def split(root, key):

if not root:

return (None, None)

if key < root.key:

left, root.left = split(root.left, key)

return (left, root)

else:

root.right, right = split(root.right, key)

return (root, right)

def merge(left, right):

if not left or not right:

return left or right

if left.priority > right.priority:

left.right = merge(left.right, right)

return left

else:

right.left = merge(left, right.left)

return right

def insert(root, node):

if not root:

return node

if node.priority > root.priority:

left, right = split(root, node.key)

node.left, node.right = left, right

return node

elif node.key < root.key:

root.left = insert(root.left, node)

else:

root.right = insert(root.right, node)

return root

def delete(root, key):

if not root:

return None

if root.key == key:

return merge(root.left, root.right)

elif key < root.key:

root.left = delete(root.left, key)

else:

root.right = delete(root.right, key)

return root

def search(root, key):

if not root:

return False

if root.key == key:

return True

elif key < root.key:

return search(root.left, key)

else:

return search(root.right, key)

def visualize\_treap(root):

graph = nx.DiGraph()

pos = {}

def add\_edges(node, x=0, y=0, layer=1):

if node is not None:

pos[f"{node.key}({node.priority})"] = (x, y)

if node.left:

graph.add\_edge(f"{node.key}({node.priority})", f"{node.left.key}({node.left.priority})")

add\_edges(node.left, x - 1 / layer, y - 1, layer \* 2)

if node.right:

graph.add\_edge(f"{node.key}({node.priority})", f"{node.right.key}({node.right.priority})")

add\_edges(node.right, x + 1 / layer, y - 1, layer \* 2)

if root:

add\_edges(root)

plt.figure(figsize=(10, 7))

nx.draw(graph, pos, with\_labels=True, node\_size=3000, node\_color="skyblue",

font\_size=10, font\_weight="bold", arrowsize=20)

plt.title("Treap Visualization")

plt.show()

# Пример

root = None

for key in [10, 20, 5, 15]:

root = insert(root, Node(key))

print("Поиск 15:", search(root, 15))

print("Поиск 10 до удаления:", search(root, 10))

# Визуализация до удаления

print("Визуализация Treap до удаления:")

visualize\_treap(root)

# Удаление узла

root = delete(root, 10)

print("Поиск 10 после удаления:", search(root, 10))

# Визуализация после удаления

print("Визуализация Treap после удаления:")

visualize\_treap(root)

Что мы получили:

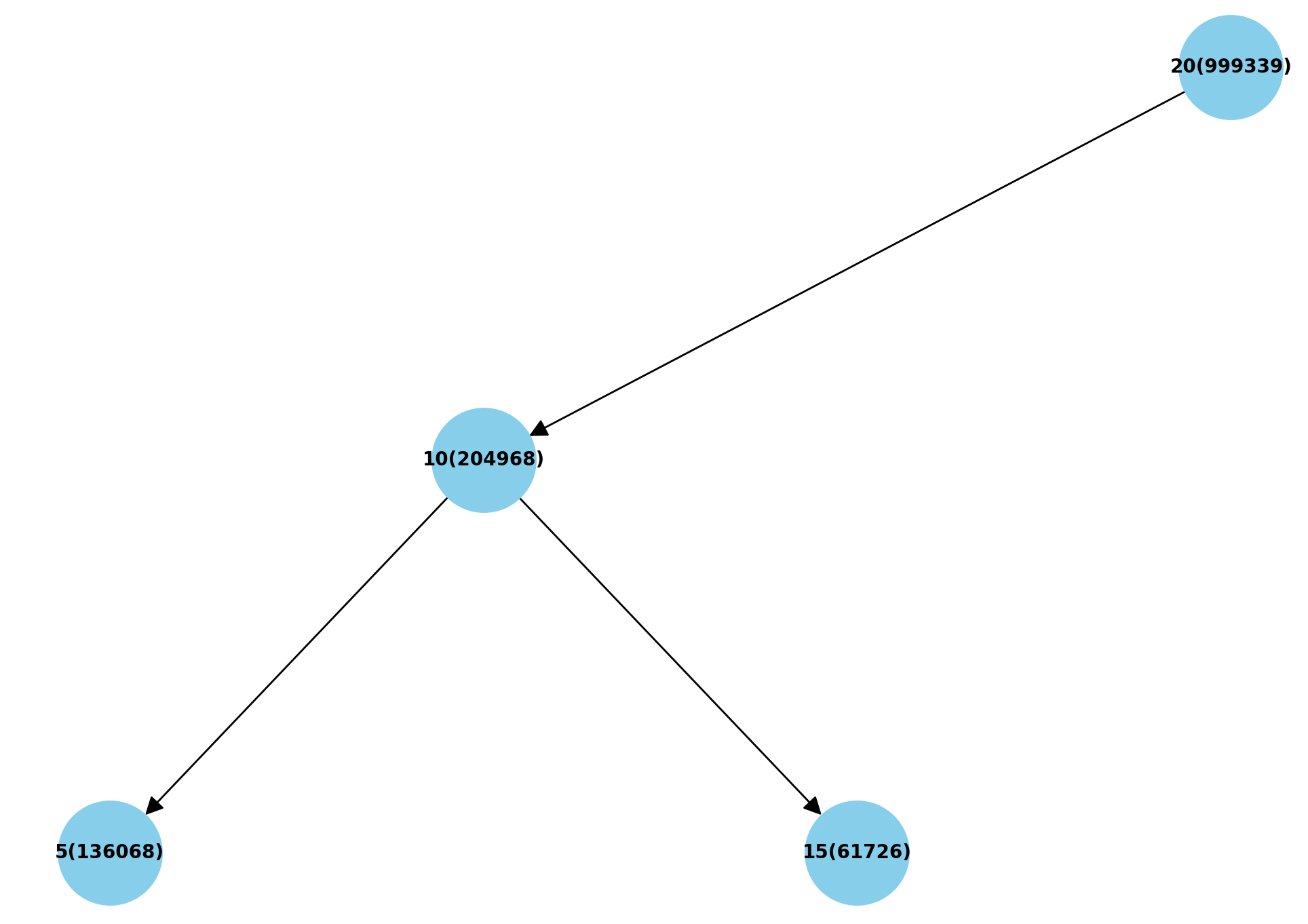


Рис.2 - Визуализация до удаления

На Рис.2 изображено **декартово дерево до удаления одного из узлов**, а именно:

* Центральный узел: **10(204968)** — это корень дерева. В скобках указан приоритет.
* Слева от него — узел **5(136068).**
* Справа — узел **15(61726).**
* Вверху — **20(999339),** он оказался выше, чем 10, потому что у него **приоритет выше**, и дерево "подняло" его.

Рис.2 показывает, как **структура treap** подчиняется одновременно: **правилам бинарного дерева поиска** (лево < корень < право) и **правилам кучи по приоритету** (родитель > потомки по приоритету).

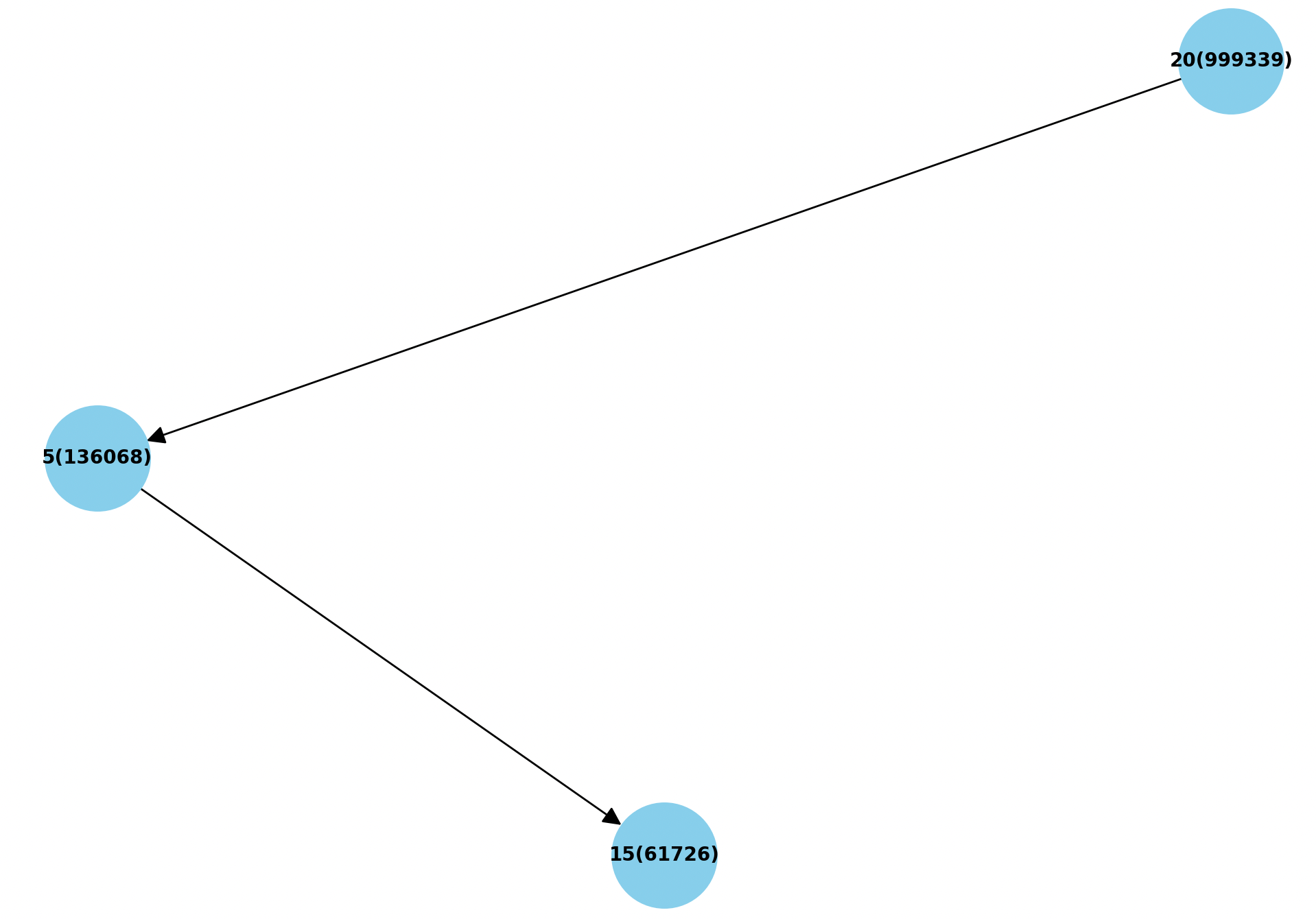


Рис.3 - Визуализация после удаления

На Рис.3 мы удалили узел с ключом **10**, и теперь его **нет на дереве**.

Удаление устроено так:

* Если нужно удалить узел, то его левое и правое поддерево **сливаются** функцией merge.
* При этом сохраняется и структура бинарного дерева, и "куча по приоритетам".

В итоге:

* Узел **10(204968)** исчез.
* Его дети **5(136068)** и **15(61726)** теперь напрямую соединены с другим узлом (с 20)
* Структура дерева изменилась, и теперь узлы пересобрались в новую конфигурацию.

Данная реализация демонстрирует работу **декартового дерева (Treap)**— структуры данных, сочетающей свойства **бинарного дерева поиска** и **двоичной кучи по приоритету**.

# 6. Сравнение с другими структурами данных

Treap относится к классу самобалансирующихся деревьев и может быть альтернативой таким структурам, как AVL-деревья и красно-черные деревья (RBT). Ниже приведены основные отличия.

|  |  |  |  |
| --- | --- | --- | --- |
| **Характеристика** | **Treap** | **AVL-дерево** | **Красно-чёрное дерево** |
| **Балансировка** | Случайная (по приоритету) | Строгая (по высоте) | По цветам узлов |
| **Сложность операций** | O(log n) в среднем | O(log n) в худшем случае | O(log n) в худшем случае |
| **Устойчивость к худшему случаю** | Нет (возможно вырождение) | Да | Да |
| **Реализация** | Простая | Сложнее | Сложная |

Treap выигрывает простотой реализации и эффективностью в среднем случае. Однако в отличие от строгих структур (например, AVL), он не гарантирует логарифмическую глубину в худшем случае — возможно вырождение при неудачном распределении приоритетов.

# 7. Преимущества и недостатки

Преимущества:

• Простота кода — минимальное количество условий и поворотов.

• Эффективность в среднем случае — логарифмическая сложность операций.

• Гибкость — легко адаптируется под задачи с неявными ключами или диапазонными запросами.

• Малая зависимость от высоты дерева — не требует строгой балансировки.

Недостатки:

• Возможность вырождения — при неудачных приоритетах глубина может быть линейной.

• Зависимость от генератора случайных чисел — качество работы связано со случайностью.

• Нет гарантий на худший случай, в отличие от строго сбалансированных деревьев.

# 8. Практическое применение

Treap находит применение в следующих областях:

1. Контестное программирование — за счёт скорости и простоты, treap активно используется для реализации структур с быстрым доступом к элементам и поддержкой модификаций.

2. Текстовые редакторы — реализация rope-структур (строковых буферов) через неявные ключи.

3. Реализация деков и очередей с приоритетами, когда нужен быстрый доступ к минимуму/максимуму с возможностью удаления произвольных элементов.

4. Системы версионирования данных — благодаря возможности эффективных копий через split и merge.

# 9. Заключение

В ходе данной курсовой работы была рассмотрена структура данных декартово дерево (treap) — симбиоз бинарного дерева поиска и кучи. Несмотря на свою простоту, treap обеспечивает эффективную поддержку всех стандартных операций с логарифмической сложностью в среднем случае. Благодаря использованию случайных приоритетов, дерево самобалансируется без необходимости сложной логики.

Быстрая реализация, наглядная структура и гибкость делают treap особенно привлекательным в задачах соревновательного программирования и при построении эффективных хранилищ с произвольными доступами. В сравнении с другими структурами treap выигрывает в удобстве, хотя и уступает в гарантированной эффективности наихудшего случая.

Ссылка на репозиторий:

https://github.com/srknrita/-