Лекция 12. Декартово дерево по неявному ключу (Treap).

1. Введение и мотивация

Декартово дерево (Treap) – это гибридная структура данных, сочетающая в себе свойства **бинарного дерева поиска (BST)** и **бинарной кучи (Heap)**. Название «Treap» образовано от слов Tree (дерево) и Heap (куча).

Зачем нужно ещё одно дерево поиска?

1. **Разные стратегии балансировки.** AVL и Splay деревья балансируются строго по высоте или на основе операций доступа. Декартово дерево использует **вероятностную балансировку**, что часто приводит к более простой реализации.

2. **Поддержка операций с последовательностями.** В режиме **неявного ключа** декартово дерево позволяет эффективно работать с массивами, поддерживая операции вставки, удаления и разрезания/склеивания за логарифмическое время.

3. **Универсальность.** Может использоваться как для задач, решаемых деревом отрезков (сумма, минимум на отрезке), так и для задач, требующих работы с порядком элементов (циклические сдвиги, реверс отрезков).

Ключевые операции:

- merge(A, B) – объединяет два дерева A и B в одно (все элементы A должны быть меньше всех элементов B).
- split(T, k) – разделяет дерево T на два дерева L и R, где L содержит первые k элементов, а R – все остальные.
- insert(i, value) – вставляет элемент в позицию i.
- delete(i) – удаляет элемент из позиции i.

Сложность (в среднем):

- Все операции: **O(log n)**
- Память: **O(n)**

2. Структура декартова дерева

Каждый узел декартова дерева хранит:

- key – **основной ключ** (в неявном режиме это *индекс* элемента в массиве, но он не хранится явно, а вычисляется через размеры поддеревьев).
- value – значение, хранимое в узле.
- priority – **приоритет** (случайное значение, по которому строится куча).
- left, right – указатели на левого и правого потомка.
- size – размер поддерева с корнем в данном узле (критично для неявного ключа).

Инварианты:

1. **Инвариант BST (по ключу key).** Для любого узла все ключи в левом поддереве меньше, все ключи в правом поддереве больше. В неявном режиме ключи – это индексы в «виртуальном» массиве, поэтому порядок обхода в порядке возрастания соответствует порядку элементов в последовательности.

2. **Инвариант кучи (по приоритету priority).** Приоритет корня не меньше (для max-heap) приоритетов его потомков.

Пример узла:

```
from typing import Any, Optional


class TreapNode:
    def __init__(self, value: Any, priority: float):
        self.value = value
        self.priority = priority
        self.size = 1  # Изначально узел один
        self.left: Optional['TreapNode'] = None
        self.right: Optional['TreapNode'] = None
```

3. Базовые операции: merge и split

3.1. Операция merge

merge(A, B) объединяет два декартовых дерева A и B в одно, при условии, что **все ключи в A меньше всех ключей в B**.

Алгоритм:

1. Если A или B пусты, вернуть другое дерево.
2. Сравниваем приоритеты корней A и B.
3. Если приоритет корня A больше:
   - Корнем результата становится корень A.
   - Правым поддеревом A становится merge(A.right, B).
4. Если приоритет корня B больше (или равен):
   - Корнем результата становится корень B.
   - Левым поддеревом B становится merge(A, B.left).
5. Обновляем размер (size) нового корня.

```
def merge(A: Optional[TreapNode], B: Optional[TreapNode]) ->
Optional[TreapNode]:
    if A is None:
        return B
    if B is None:
        return A

    if A.priority > B.priority:
        A.right = merge(A.right, B)
        update_size(A)  # Пересчитываем размер после изменения детей
        return A
    else:
        B.left = merge(A, B.left)
```

```
        update_size(B)
        return B
```

3.2. Операция split

split(T, k) разделяет дерево T на два дерева L и R, где L содержит первые k элементов, R – остальные.

Алгоритм:

1. Если T пусто, вернуть два пустых дерева.
2. Вычисляем размер левого поддерева текущего корня: left_size = size(T.left).
3. Если k <= left_size:
   - Рекурсивно вызываем split(T.left, k), получая L1 и R1.
   - T.left = R1
   - Обновляем размер T.
   - Возвращаем (L1, T).
4. Если k > left_size:
   - Рекурсивно вызываем split(T.right, k - left_size - 1), получая L2 и R2.
   - T.right = L2
   - Обновляем размер T.
   - Возвращаем (T, R2).

```
def split(T: Optional[TreapNode], k: int) -> tuple[Optional[TreapNode],
Optional[TreapNode]]:
    if T is None:
        return None, None

    left_size = size(T.left)  # Функция size безопасно обрабатывает None
    if k <= left_size:
        L, R = split(T.left, k)
        T.left = R
        update_size(T)
        return L, T
    else:
        L, R = split(T.right, k - left_size - 1)
        T.right = L
        update_size(T)
        return T, R


def size(node: Optional[TreapNode]) -> int:
    return node.size if node else 0


def update_size(node: TreapNode) -> None:
    node.size = 1 + size(node.left) + size(node.right)
```

4. Реализация операций с последовательностью

Используя merge и split, можно легко реализовать вставку и удаление по индексу.

4.1. Вставка элемента

Чтобы вставить элемент value с приоритетом priority на позицию pos (индексация с 0), мы:

1. Разделяем исходное дерево T на две части L и R, где L содержит первые pos элементов.
2. Создаём новый узел new_node для вставляемого элемента.
3. Объединяем L с new_node, получая L_new.
4. Объединяем L_new с R, получая новое дерево.

```
def insert(T: Optional[TreapNode], pos: int, value: Any, priority: float =
None) -> TreapNode:
    if priority is None:
        priority = random.random()  # Генерируем случайный приоритет
    L, R = split(T, pos)
    new_node = TreapNode(value, priority)
    return merge(merge(L, new_node), R)
```

4.2. Удаление элемента

Чтобы удалить элемент на позиции pos:

1. Разделяем T на L и R, где L содержит первые pos элементов.
2. Разделяем R на M и R2, где M содержит 1 элемент (тот, который нужно удалить).
3. Объединяем L и R2, игнорируя M.

```
def delete(T: Optional[TreapNode], pos: int) -> Optional[TreapNode]:
    L, R = split(T, pos)
    M, R2 = split(R, 1)  # M - узел, который нужно удалить
    return merge(L, R2)
```

4.3. Получение элемента по индексу

```
def get_at(T: Optional[TreapNode], index: int) -> Any:
    if T is None:
        raise IndexError("Index out of range")
    left_size = size(T.left)
    if index < left_size:
        return get_at(T.left, index)
    elif index == left_size:
        return T.value
    else:
        return get_at(T.right, index - left_size - 1)
```

5. Полная реализация класса ImplicitTreap

```
import random
from typing import Any, Optional, List, Tuple


class ImplicitTreap:
    class Node:
        __slots__ = ('value', 'priority', 'size', 'left', 'right')

        def __init__(self, value: Any, priority: float):
            self.value = value
            self.priority = priority
            self.size = 1
            self.left = None  # type: Optional[ImplicitTreap.Node]
            self.right = None  # type: Optional[ImplicitTreap.Node]

    def __init__(self, data: Optional[List[Any]] = None):
        self.root = None  # type: Optional[ImplicitTreap.Node]
        if data:
            for item in data:
                self.insert(len(self), item)  # Вставляем в конец

    def _size(self, node: Optional[Node]) -> int:
        return node.size if node else 0

    def _update_size(self, node: Node) -> None:
        node.size = 1 + self._size(node.left) + self._size(node.right)

    def _merge(self, A: Optional[Node], B: Optional[Node]) -> Optional[Node]:
        if A is None: return B
        if B is None: return A
        if A.priority > B.priority:
            A.right = self._merge(A.right, B)
            self._update_size(A)
            return A
        else:
            B.left = self._merge(A, B.left)
            self._update_size(B)
            return B

    def _split(self, T: Optional[Node], k: int) -> Tuple[Optional[Node],
Optional[Node]]:
        if T is None:
            return None, None
        left_size = self._size(T.left)
        if k <= left_size:
            L, R = self._split(T.left, k)
            T.left = R
            self._update_size(T)
            return L, T
        else:
            L, R = self._split(T.right, k - left_size - 1)
            T.right = L
```

```
            self._update_size(T)
            return T, R

    def insert(self, pos: int, value: Any) -> None:
        L, R = self._split(self.root, pos)
        new_node = self.Node(value, random.random())
        self.root = self._merge(self._merge(L, new_node), R)

    def delete(self, pos: int) -> None:
        L, R = self._split(self.root, pos)
        M, R = self._split(R, 1)
        self.root = self._merge(L, R)

    def __getitem__(self, index: int) -> Any:
        return self._get_at(self.root, index)

    def _get_at(self, node: Optional[Node], index: int) -> Any:
        if node is None:
            raise IndexError("Treap index out of range")
        left_size = self._size(node.left)
        if index < left_size:
            return self._get_at(node.left, index)
        elif index == left_size:
            return node.value
        else:
            return self._get_at(node.right, index - left_size - 1)

    def __len__(self) -> int:
        return self._size(self.root)
```

6. Примеры использования

Пример 1. Базовые операции с массивом.

```
# Создаём пустое дерево
arr = ImplicitTreap()

# Вставляем элементы
arr.insert(0, 'A')
arr.insert(1, 'C')
arr.insert(1, 'B')  # Вставляем 'B' между 'A' и 'C'

print([arr[i] for i in range(len(arr))])  # ['A', 'B', 'C']

# Удаляем элемент на позиции 1
arr.delete(1)
print([arr[i] for i in range(len(arr))])  # ['A', 'C']

# Получаем элемент по индексу
print(arr[1])  # 'C'
```

Пример 2. Разрезание и склеивание.

```
# Создаём дерево из списка
data = ['x', 'y', 'z']
treap = ImplicitTreap(data)
```

```
# Разрезаем после первого элемента
L, R = treap._split(treap.root, 1)

# L содержит ['x'], R содержит ['y', 'z']
# Склеиваем R и L, получая ['y', 'z', 'x']
new_root = treap._merge(R, L)

# Восстанавливаем корень в классе (для демонстрации)
treap.root = new_root
print([treap[i] for i in range(len(treap))])  # ['y', 'z', 'x']
```

7. Расширение функциональности: агрегатные операции

Как и в дереве отрезков, в декартово дерево можно добавить поддержку операций на отрезке (сумма, минимум, массовые обновления). Для этого в каждом узле нужно хранить дополнительную информацию и поддерживать её при операциях merge и split.

Пример. Подсчёт суммы на поддереве.

```
class ImplicitTreapWithSum(ImplicitTreap):
    class Node(ImplicitTreap.Node):
        def __init__(self, value: int, priority: float):
            super().__init__(value, priority)
            self.sum = value  # Сумма значений в поддереве

    def _update_node(self, node: Node) -> None:
        super()._update_size(node)
        left_sum = node.left.sum if node.left else 0
        right_sum = node.right.sum if node.right else 0
        node.sum = node.value + left_sum + right_sum

    # Переопределяем _merge и _split для обновления суммы
    def _merge(self, A: Optional[Node], B: Optional[Node]) -> Optional[Node]:
        if A is None: return B
        if B is None: return A
        if A.priority > B.priority:
            A.right = self._merge(A.right, B)
            self._update_node(A)
            return A
        else:
            B.left = self._merge(A, B.left)
            self._update_node(B)
            return B

    # Аналогично переопределяем _split
    # ... (код аналогичен базовому, но с вызовом _update_node)

    def range_sum(self, l: int, r: int) -> int:
        """Возвращает сумму на полуинтервале [l, r)."""
        L, M = self._split(self.root, l)
        M, R = self._split(M, r - l)
        result = M.sum if M else 0
        self.root = self._merge(L, self._merge(M, R))
        return result
```

8. Сравнение с другими структурами

Параметр	**Декартово дерево (неявный ключ)**	**Дерево отрезков**	**Список (list)**
Вставка/удаление	**O(log n)** (в среднем)	O(log n)	O(n)
Индексный доступ	**O(log n)**	O(log n) (только с доп. данными)	**O(1)**
Разрезание/склейка	**O(log n)**	Сложно	O(n)
Память	O(n)	O(n)	O(n)
Реализация	Средняя сложность (рекурсия)	Относительно простая	Очень простая
Лучшее применение	Динамические последовательности, склейки	Статические/обновляемые запросы на отрезке	Статические данные