Санкт-Петербугский политехнический университет Петра Великого

Институт машиностроения, материалов и транспорта

Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: объектно-ориентированное программирование

Тема: АВЛ Дерево

Студенты гр. 3331506 / 00401 Богдашевская А. Д.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2023

Оглавление

Введение

АВЛ Дерево – сбалансированное бинарное дерево поиска. Бинарное дерево поиска – иерархическая структура данных в которой каждый узел имеет не более чем два потомка, причем значение ключа любого узла левого поддерева произвольного узла X меньше, чем значение самого узла X, а значение любого узла правого поддерева узла X больше либо равно значению ключа узла X. Сбалансированность АВЛ Дерева заключается в том, что разница высот поддеревьев любого узла не превышает 1.

Этот тип данных был впервые введен в 1962 году Г. М. Адельсоном-Вельским и Е. М. Ландисом [1], первые буквы фамилий которых стали названием их изобретения.

У АВЛ Дерева разница высот правого и левого поддерева любого узла лежит в диапазоне $\{-1, 0, 1\}$. Ввиду этого высоту дерева с n элементами можно представить как:

$$h = O(\log n).$$

Для сохранения сбалансированности дерева после каждой операции добавления или удаления вершины нужно производить балансировку. Есть четыре типа балансировки: малое левое вращение, малое правое вращение, большое левое вращение и большое правое вращение. Подробнее они описаны в основной части работы. Балансировка требует $O(1)$.

Так как в процессе добавления, удаления или поиска вершины мы рассматриваем не более, чем $O(h)$. вершин дерева, и для каждой запускаем балансировку не более одного раза, то суммарное количество операций при включении новой вершины в дерево составляет $O(\log n)$ операций. Зависимость времени операций от количества вершин представлена на рисунке 1.

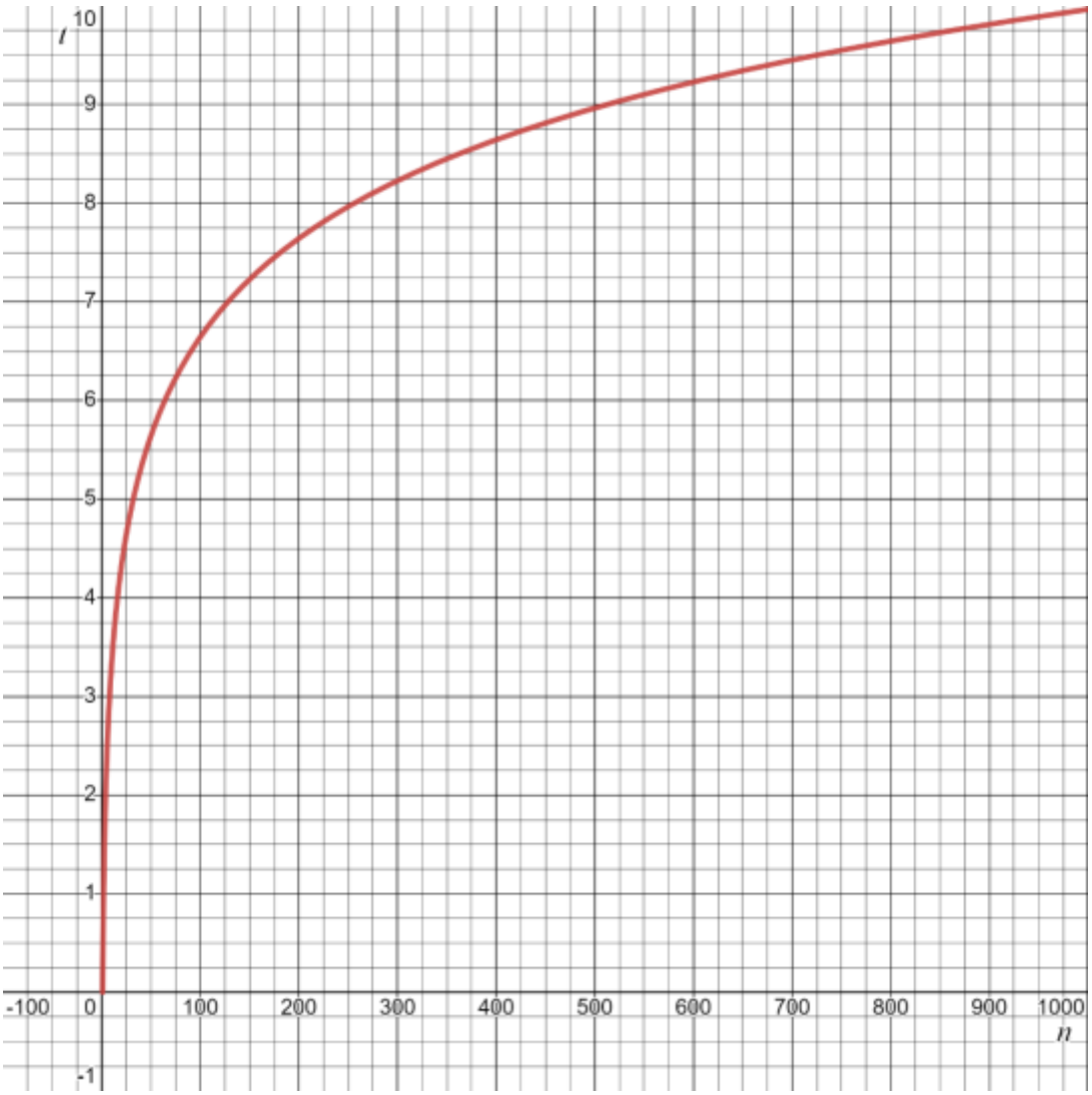

Рисунок 1

АВЛ Деревья широко применяются для хранения, поиска и сортировки данных ввиду эффективности организации.

Основная часть

Организация дерева

Для узлов дерева создадим отдельный класс *NodeTree*. Главные элементы этого класса:

- *data* – данные, хранимые в узле
- *key* – ключ, по которому осуществляется сортировка
- *left_child* – указатель на левого ребенка
- *right_child* – указатель на правого ребенка
- *height* – высота наибольшего из поддеревьев
- *bf* – коэффициент сбалансированности (разность высот левого и правого поддеревьев)

Само АВЛ Дерево реализовано классом *Tree*. Дерево определяется корнем *root* – указателем на узел, являющийся корнем.

Основные операции

1. Малое левое вращение

Этот метод балансировки применяется в случае, если коэффициент сбалансированности узла меньше -1, а коэффициент сбалансированности его правого ребенка неположительный. Схема вращения представлена на рисунке 2.

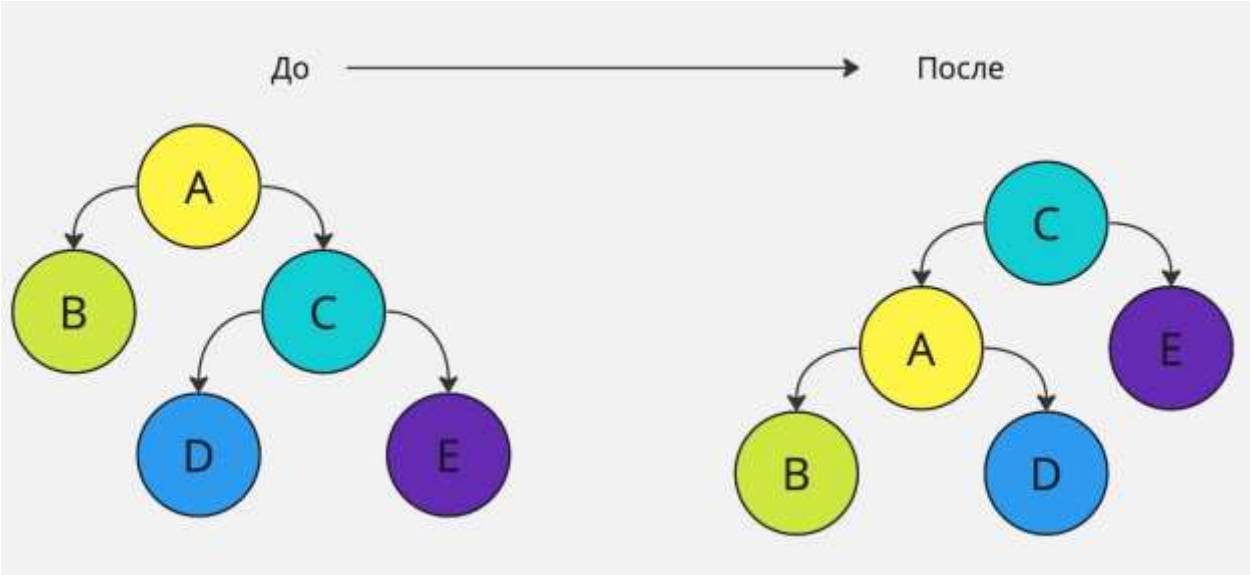

Рисунок 2

Малое левое вращение реализовано в методе класса *Tree l_rotate*.

2. Малое правое вращение

Этот метод балансировки применяется в случае, если коэффициент сбалансированности узла больше 1, а коэффициент сбалансированности его правого ребенка неотрицательный. Схема вращения представлена на рисунке 3.

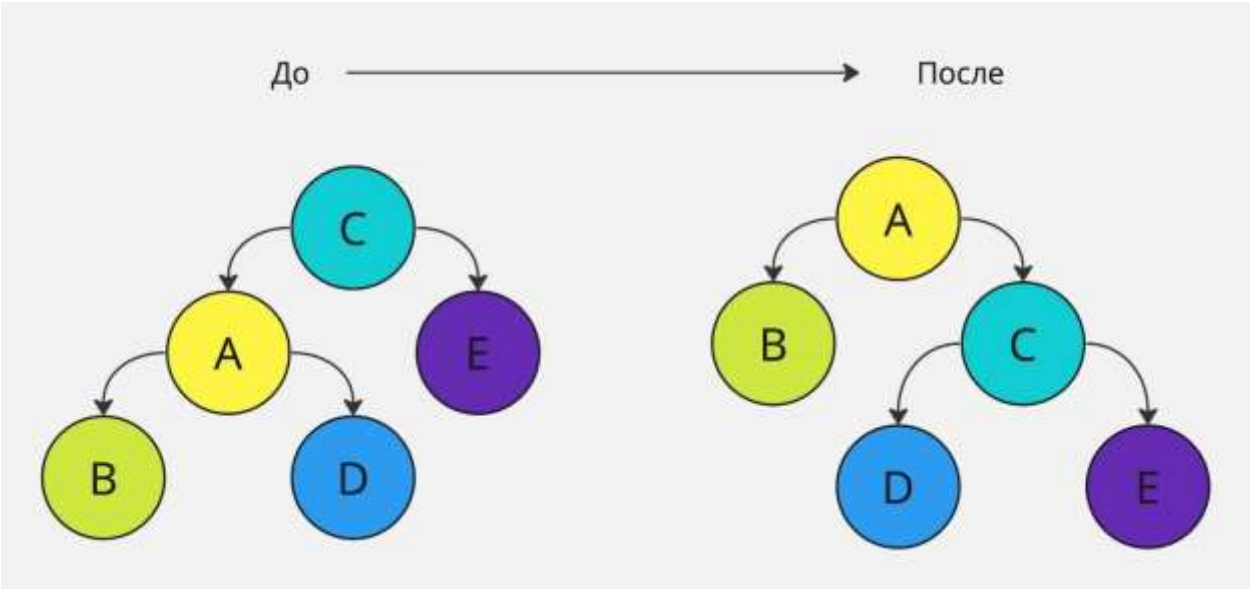

Рисунок 3

Малое правое вращение реализовано в методе класса *Tree r_rotate*.

3. Большое левое вращение

Этот метод балансировки применяется в случае, если коэффициент сбалансированности узла меньше -1, а коэффициент сбалансированности его правого ребенка положительный. Схема вращения представлена на рисунке 4.

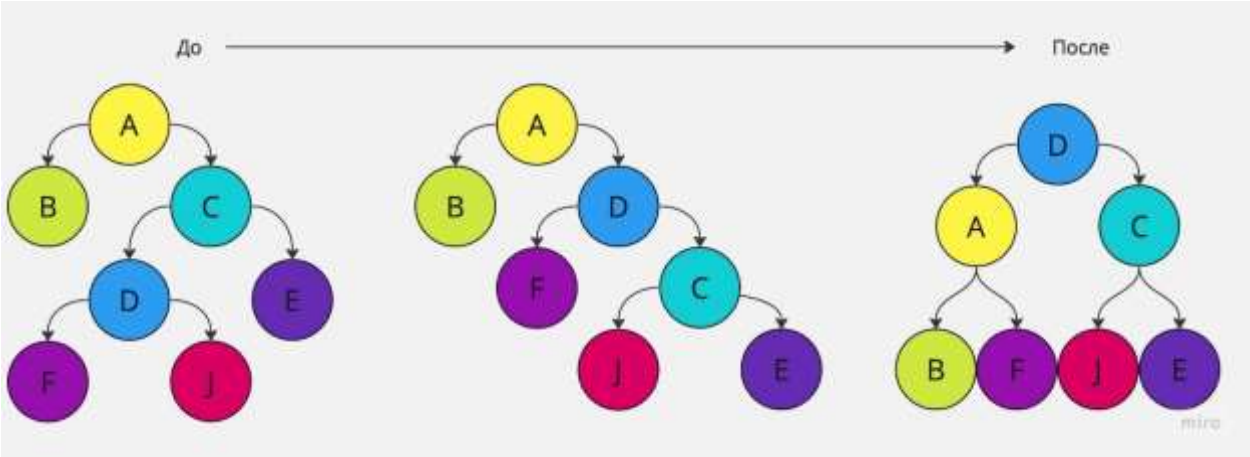

Рисунок 4

Большое левое вращение реализовано в методе класса *Tree rl_rotate*.

4. Большое правое вращение

Этот метод балансировки применяется в случае, если коэффициент сбалансированности узла меньше -1, а коэффициент сбалансированности его правого ребенка неположительный. Схема вращения представлена на рисунке 5.

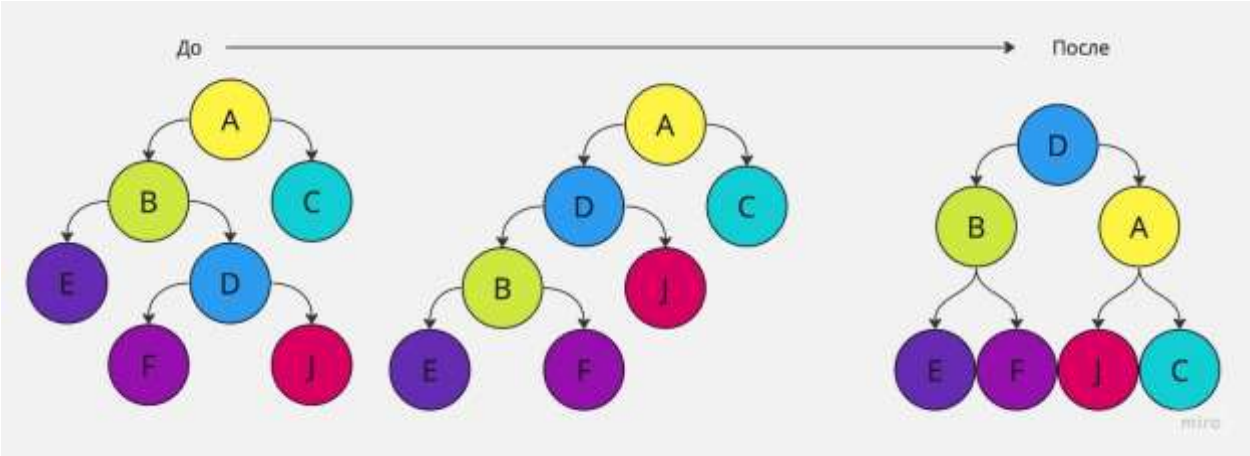

Рисунок 5

Большое правое вращение реализовано в методе класса *Tree lr_rotate*.

5. Балансировка

После каждого добавления или удаления узла стек со всеми узлами, которые находятся выше добавленного или удаленного, передается в функцию, которая обновляет значения *height* и *bf* и определяет, необходима ли балансировка и, если необходима, то какого типа.

Балансировка реализована в методе класса *Tree balance*.

6. Добавление узла

Новые элементы вставляются на место листа (отсутствующего ребенка). Для добавления нового узла мы сравниваем новый ключ с ключом текущего узла (начиная с корня) пока текущий узел не станет *nullptr*: если новый ключ больше текущего, текущим узлом становится правый ребенок, если меньше – левый, если новый ключ совпадает с текущим – выводим ошибку: “ Node with this key already exists”. Каждый текущий узел последовательно вносится в стек, который после добавления узла будет передан для балансировки.

Добавление узла реализовано в методе класса *Tree add*.

7. Удаление узла

Для удаления узла мы сравниваем удаляемый ключ с ключом текущего узла (начиная с корня) пока текущий узел не станет *nullptr*: если новый ключ больше текущего, текущим узлом становится правый ребенок, если меньше – левый, если новый ключ совпадает с текущим – начинаем процесс удаления и прерываем цикл. Если цикл завершился натуральным образом, выводим ошибку: “Node does not exist”. Каждый текущий узел последовательно вносится в стек, который после удаления узла будет передан для балансировки.

Процесс удаления делится на при типа:

- У удаляемого узла нет детей: заменяем удаляемый узел на *nullptr*.
- У удаляемого узла 1 ребенок: заменяем удаляемый узел на ребенка.
- У удаляемого узла 2 ребенка: заменяем удаляемый узел на крайнего правого потомка левого ребенка удаляемого узла.

Добавление узла реализовано в методе класса *Tree del_by_key*.

Также реализован метод *del_by_data*. Он вызываем последовательно функции поиска и удаления по ключу.

8. Поиск

С помощью очереди совершаем обход дерева в ширину, сравнивая искомое значение с текущими значениями узлов. Если находим совпадение, возвращаем ключ узла, если не находим – выводим ошибку: “ Node does not exist”.

Поиск узла реализован в методе класса *Tree get_key*.

Заключение

Реализовав алгоритм АВЛ Дерева и изучив его свойства, мы можем сделать выводы:

- использование их вместо простых двоичных деревьев поиска приведет к уменьшению времени операции поиска, вставки и удаления данных при одинаковом количестве элементов в дереве;
- однако это приведет к усложнению алгоритма взаимодействия с деревом ввиду того, что после выполнения каждой операции вставки и удаления придется выполнять операцию проверки дерева на сбалансированность, и при обнаружении разбалансировки выполнять операции поворота узлов дерева;
- АВЛ Деревья рационально использовать при работе с большим объемом данных

Список литературы

1. Адельсон-Вельский Г. М., Ландис Е. М. Один алгоритм организации информации // Доклады АН СССР. - 1962. - Т. 146, № 2. - С. 263-266.
2. Вирт Н. Алгоритмы и структуры данных. - М.: Мир, 1989. - С. 272-286
3. Lewis H. R., Denenberg L. Data Structures and Their Algorithms. Addison-Wesley. 1991

Приложение

Код программы:

https://github.com/soomrack/MR2022/blob/main/Bogdashevskaya_Alexandra/CourseWork/CourseWork.cpp