МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования

Дальневосточный федеральный университет

ИНСТИТУТ МАТЕМАТИКИ И КОМПЬЮТЕРНЫХ ТЕХНОЛОГИЙ

Департамент математического и компьютерного моделирования

ДОКЛАД о практическом задании по дисциплине АИСД

Сбалансированные деревья: Splay и AA

Курпас Артём Викторович, гр. Б9121-09.03.03пикд

г. Владивосток, 2022

Содержание

Аннотация

В данном докладе рассматриваются формы бинарного дерева - Splay-деревья и AA-

деревья, описывается принцип работы и реализация этих структур данных. Исследованию подлежит их производительность относительно друг друга.

1. Введение

По мере развития компьютерной сферы возникает необходимость в быстром и удобном хранилище данных, которое, в идеале, должно отвечать на все запросы быстро и хранить данные эффективно (с минимальными затратами памяти). Быстрее, чем в обычном массиве, данные можно хранить, например, используя бинарные деревья, сохраняя элементы по ключу.
Далее будут рассматриваться формы бинарного дерева поиска - Splay-деревья и AA-деревья, которые принадлежат к классу так называемых сбалансированных деревьев.

Splay-дерево было придумано Робертом Тарьяном и Даниелем Слейтером в 1983 году.
AA-дерево было придумано Арне Андерссоном в 1993 году.

1. 1. Глоссарий

1. Сбалансированные структуры данных - структуры данных, которые так или иначе изменяют структуру (например, используя высоту или цвет), для достижения эффективного взаимодействия с ними.

1.2. Неформальная постановка задачи

1. Необходимо осуществить реализацию алгоритмов как отдельные заголовочные файлы (.h)
2. Осуществить методы взаимодействия с ними - команды *access*, *insert*, *erase*.
3. Необходимо установить асимптотическую оценку для данных алгоритмов и определить, на каких задачах они будут максимально эффективными, в том числе, используя тесты с различным набором данных.

1.3. Математические методы

При оценке эффективности Splay-дерева используется метод так называемого *амортизационного анализа*. Источник - Статья на Вики ИТМО: *https://neerc.ifmo.ru/wiki/index.php?title=Амортизационный_анализ*.

Определение:
Амортизационный анализ (англ. *amortized analysis*) — метод подсчёта времени, требуемого для выполнения последовательности операций над структурой данных. При этом время усредняется по всем выполняемым операциям, и анализируется средняя производительность операций в худшем случае.

Определение:
Средняя амортизационная стоимость операций — величина a, находящаяся по формуле: $a = \frac{\sum_{i=1}^{n} t_i}{n}$ где t — время выполнения операций 1, 2… n, совершённых над структурой данных.

Конкретно, для доказательства эффективности операций *splay*, *access*, *insert*, *erase* используется метод амортизационного анализа - *метод потенциалов*.

Теорема (О методе потенциалов):
Введём для каждого состояния структуры данных величину Φ — потенциал. Изначально потенциал равен Φ_0, а после выполнения i-й операции — Φ_i. Стоимость i-й операции обозначим $a_i = t_i + \Phi_i - \Phi_i - 1$. Пусть n — количество операций, m — размер структуры данных. Тогда средняя амортизационная стоимость операций $a = O(f(n, m))$, если выполнены два условия:

- Для любого i : $a_i = O(f(n, m))$
- Для любого i : $\Phi_i = O(n \cdot f(n, m))$

$$a = \frac{\sum_{i=1}^{n} t_i}{n} = \frac{\sum_{i=1}^{n} a_i \sum_{i=0}^{n-1} \Phi_i - \sum_{i=1}^{n} \Phi_i}{n} = \frac{n \cdot O(f(n, m)) + \Phi_0 - \Phi_n}{n} = O(f(n, m))$$

1.4. Обзор существующих методов решения

Существуют готовые реализации рассматриваемых структур данных.

Splay-дерево.

https://www.cs.usfca.edu/~galles/visualization/SplayTree.html

Возможности:

- Визуализация данных
- Добавление, удаление, поиск элемента
- Вывод дерева с использованием симметричного обхода
- Выбор скорости анимации, можно приостановить / воспроизвести анимацию

AA-дерево.

https://tjkendev.github.io/bst-visualization/red-black-tree/aa-tree.htm

Возможности:

- Визуализация данных
- Добавление, удаление, поиск элемента
- Добавление/удаление случайного элемента
- Перемотка анимации

2. Требования к окружению

2.1. Требования к аппаратному обеспечению

- Персональный компьютер, поддерживающий компилирование C++
- Тестирующая система CATS

2.2. Требования к программному обеспечению

- ОС, позволяющая компилировать файлы .cpp и испольнять .exe
- Любой компилятор C++ (например, g++)
- Git

2.3. Требования к пользователям

- Умение работать в командой строке, выполнять базовые команды Git
- Умение пользоваться компилятором

3. Спецификация данных

В рассматриваемых деревьях используется структура данных Node (далее - узел). Её использование обосновано тем, что не нужно будет иметь несколько различных массивов для хранения различного рода. Узел содержит несколько полей:

- Поле с данными, ключ
- Поле, содержащее ссылки на другие узлы

Сравнивание ключей и переход по нужной ссылке, позволит передвигаться по памяти, отведённой по дерево, для выполнения необходимых операций (см. пункт 4).

Опционально, в зависимости от реализации, имеются также поля:

- Поле с числом, содержащее текущую высоту узла, отрицательная высота не предполагается
- Поле, содержащее ссылку на родителя

4. Функциональные требования

Разработанная структура данных должна:

- представлять собой библиотеку-класс (.h) с определёнными методами (далее - операции с деревом):
    - $insert(key)$ - добавить данные по ключу key, если найден узел с таким же ключом, заверить операцию.
    - $erase(key)$ - удалить данные по ключу key, если не найдено ключа *key*, то завершить операцию.
    - $access(key)$ - получить данные по ключу key, если не найдено

ключа key, то вернуть *нулевой указатель*.

- хранить данные
- иметь отдельный заголовочный файл, который будет управлять ей для считывания команд из файлов (в основном, для облегчения тестирования).

Тесты должны:

- полностью покрывать функционал программы, состоять из операций с деревом
- иметь автоматическую генерацию (для больших тестов)
- покрывать производительность, затраты на время и/или память

5. Формальное описание алгоритмов

Splay и AA деревья являются модификациями бинарного дерева поиска. Они принадлежат классу так называемых балансирующихся структур данных. Балансировка в них происходит за счёт применения определённого набора правил, применяемых во время построения.

Обращаясь к деревьям, мы подразумеваем обращение к их корневому узлу. Дерево, корневой узел которого пустой указатель, является пустым.

Основные операции

- $access(i, t)$ – найти элемент i и вернуть указатель на него или нулевой указатель – в противном случае.
- $insert(i, t)$ – вставить элемент i, если его ещё не существет во множестве.
- $delete(i, t)$ – удалить элемент i, если он есть во множестве.

> Стоит отметить, что элемент x есть в данном множестве, если применение операции *access* вернёт указатель на этот элемент. Элемента нет во множестве, если операция возвратила пустой указатель (*nullptr*).

- $join(t_1, t_2)$ – объединить деревья t_1 и t_2 в одно, которое содержит все элементы из обоих деревьев. Операция предполагает, что все элементы из t_1 должны быть меньше, чем минимальный из t_2. Возвращает указатель на новое дерево, удаляет t_1 и t_2.

- $split(t)$ – вернуть два дерева t_1 и t_2, где t_1 будет содержать элементы из t, которые меньше или равны i, и t_2 будет содержать элементы, которые больше или равны i. Операция удаляет t.

Splay-деревья

Splay-дерево является «самобалансирующейся» структурой данных. Самобалансирующиеся деревья способы проводить балансировку без использования каких-либо дополнительных полей в узлах дерева (хранение ссылки на родителя не является таковой). Например, в узлах Красно-чёрных деревьев или АВЛ-деревьев хранится, соответственно, цвет вершины или глубина поддерева. Splay-деревья всегда находятся в произовольном состоянии, т. е. они не стремятся к конкретному балансу. Наример, форма AVL-деревьев практически всегда есть форма идеально сбалансированного бинарного дерева.

Выделяют следующие преимущества самобалансирующихся структур данных:

- Требуется меньше места в памяти, так как мы не храним информацию о высоте, цвете и балансе
- Их принцип работы достаточно прост, и поэтому алгоритм реализуется быстрее

Splay-деревья призваны уменьшать время операции для наихудшего случая. Это достигается путём «выталкивания» элемента x в корень дерева. Мы договариемся, что, если, применять эту *эвристику*, ответы на запросы станут быстрее. Таким образом, недавно использовавшийся элемент будет ближе к корню (откуда всегда и начинается поиск). Это позволит быстрее получить к нему доступ и совершить необходимые *операции*.

Кроме того, учитывая то, что в большинстве реальных практических задач (напр., в базах данных) последовательности операций преобладают над отдельными операциями, поэтому по время замеров производительности будем считать, что важно не время одной кокретной операции, а общее время выполнения последовательности (так называемое *амортизированное время*).

Splaying

Предположим, мы хотим выполнить несколько вышеперечисленных операций в дереве. Тогда, для того чтобы минимизировать общее время выполнения операций, мы должны убедиться в том, что наиболее частые по запросам элементы находятся наиболее близко к корню, так как время доступа до элемента напрямую зависит от его глубины - $O(h)$, где h - глубина узла.

Как уже говорилось ранее, задача *splay*-дерева состоит в том, чтобы «перестроиться», тем самым подняв элемент x выше, чтобы быстрее получить к нему доступ во время следующей операции. Это производится за счёт вращения (англ. *rotation*).

Одиночное вращение

Выполняется за $O(1)$ (опускаем константное количество операций, так как общее число узлов в дереве не влияет на скорость этой операции), сохраняет все свойства дерева, т. е. ключ левого сына x остаётся меньше чем x, а ключ правого - соответсвенно больше. Вращение может быть как правым, так и левым.

Одиночное вращение от x есть поворот относительно ребра, соединяющего x со своим предком.

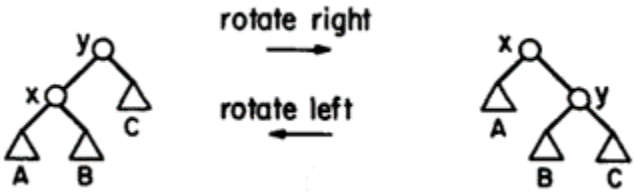

Рисунок 1. Вращение относительно x. Треугольники обозначают поддеревья.

Move to root

Совершаем одиночные вращения от x, пока не убедимся, что x - корень.

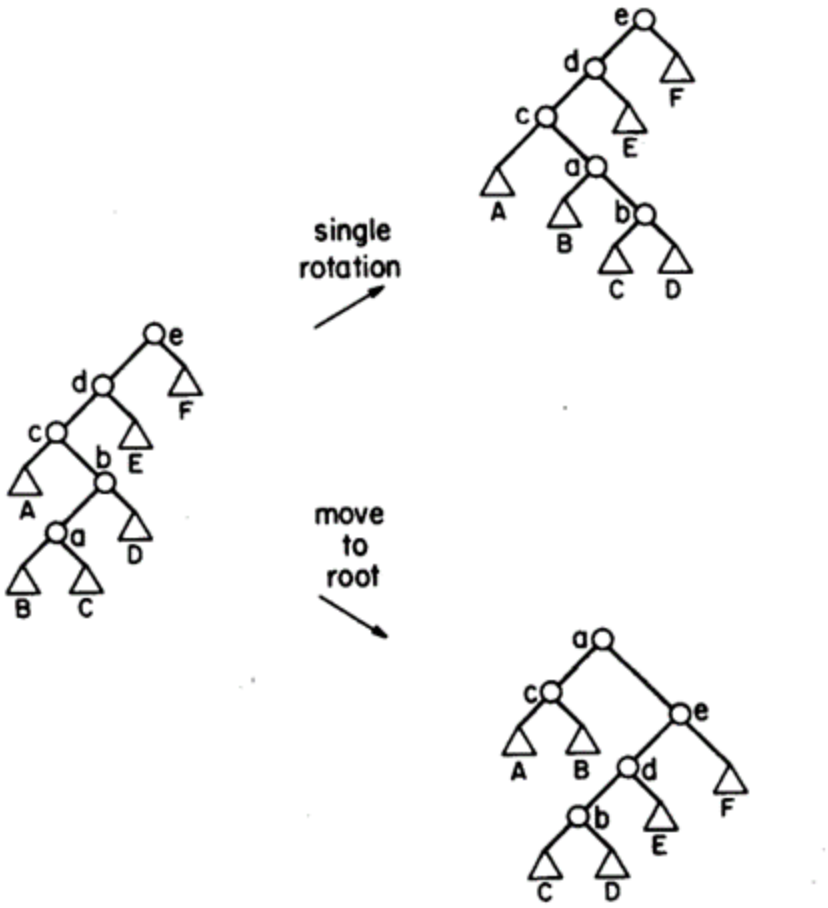

Рисунок 2. Применение операций single_rotation и move_to_root к узлу a.

К сожалению в *splay*-дереве невозможно использование операции *move_to_root*, так как, для неё найдётся такая произвольно длинная последовательность операциq, такая, что время выполнения окажется близким к $O(n)$.

Основная эвристика, используемая для перебалансировки дерева в *splay*-дереве называется *splaying*. Она преследует ту же цель, что и *move_to_root*, однако подъём в корень осуществляется при помощи чередующихся операций поворотов. Чтобы осуществить *splaying* в узле x, мы будем повторять следующие операции, пока x не станет корнем.

Операции splay

> *Обозначение*. p – родитель (англ. *parent*) узла x;

> *Обозначение*. g – прародитель (англ. *grandfather*) узла x, отец p.

Операция 1 (*zig*). Если p – родитель x, то совершаем поворот относительно ребра,

соединяющего p и x. (Совершается один раз и только в конце).

Операция 2 (*zig-zig*). Если p – не корень и родитель x, p – ребенок g и x, и p – только левые дети или только правые дети, то совершаем сначала поворот относительно ребра, соединяющего p и g, после чего совершаем поворот относительно ребра, соединяющего x и p.

Операция 3 (*zig-zag*). Если p – не корень и x - левый ребёнок p, а p – правый ребёнок g ИЛИ x - правый ребёнок p, а p – левый ребёнок g, то совершаем сначала поворот относительно ребра, соединяющего x и p, после этого совершаем поворот относительно ребра, соединяющего x и g.

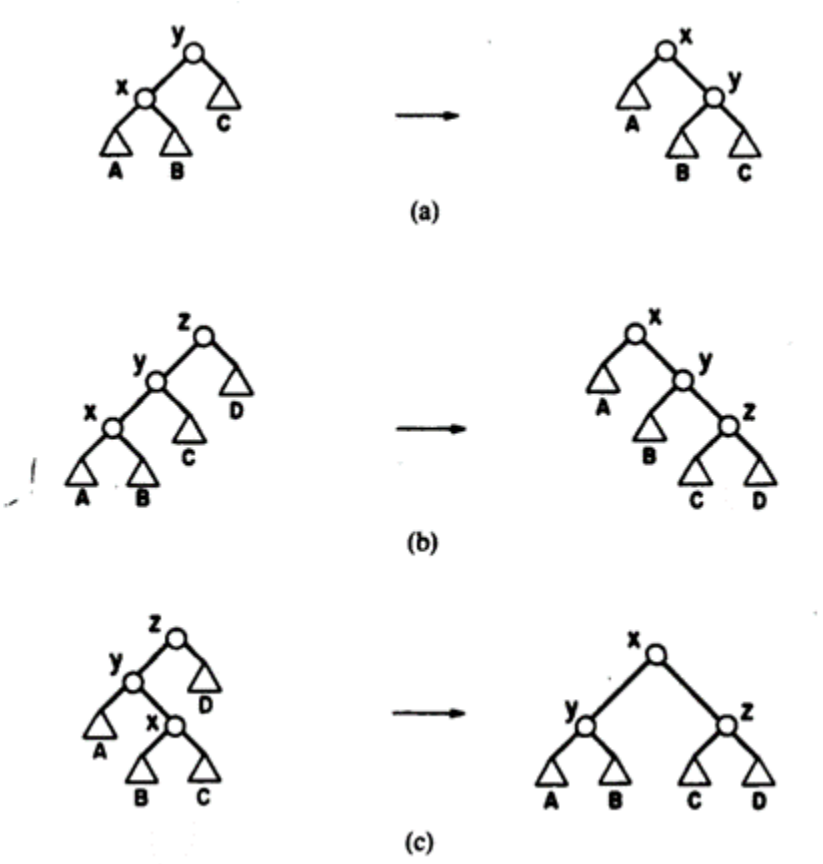

Рисунок 3. Применение splay к узлу x. Каждый вариант поворота имеет свой зеркальный вариант. (a) Zig: окончательное одиночное вращение. (b) Zig-zig: два одиночных вращения. (c) Zig-zag: двойное вращение.

Splay в узле x на глубине h требует времени $O(h)$, что пропорционально времени, затраченному на нахождение узла x. Операция *splay*, помимо перемещения узла x, на каждой итерации грубо вдвое уменьшает глубину каждого узла (перемещает их ближе к

корню) (см. Рисунок 4 и 5). Это преимущество не даёт применение более простых методов, таких как *move-to-root*, так как в них не используются такие подходы как *zig-zig* и *zig-zag*.

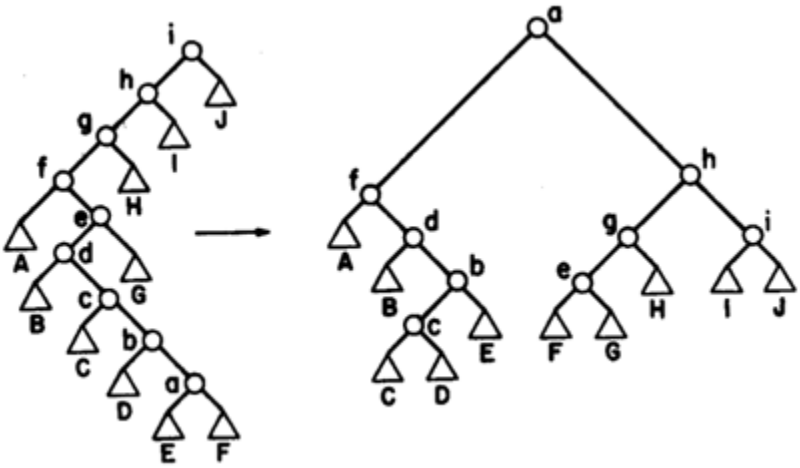

Рисунок 4. Применение splay к узлу a.

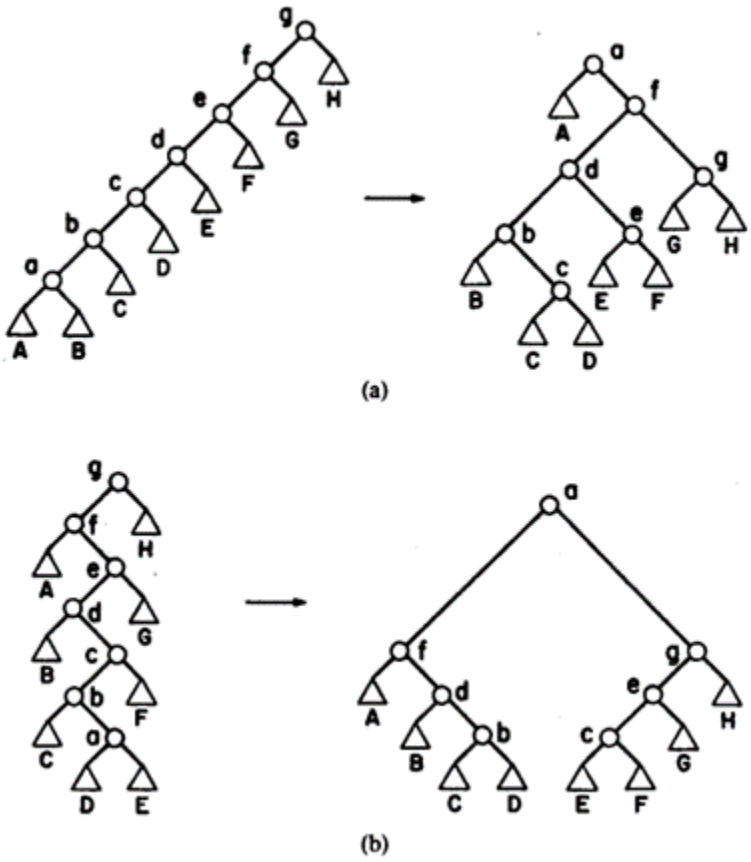

Рисунок 5. Наихудшие случаи для splay. (a) Использовано исключительно zig-zig. (b) Использовано исключительно zig-zag.

Работа с деревом

Вышеперечисленные операции выполняются следующим образом.

Операция $access(i, t)$ производится от корня. Если поиск достигает узла x, содержащего i, мы завершаем операцию, начиная $splay(x)$, и возвращаем указатель на x. Если поиск достигает нулевого указателя, то мы завершаем операцию, вызывая $splay$ от предыдущего узла, к которому был получен доступ. Если дерево пусто, мы возвращаем пустой указатель. (см. Рисунок 6.)

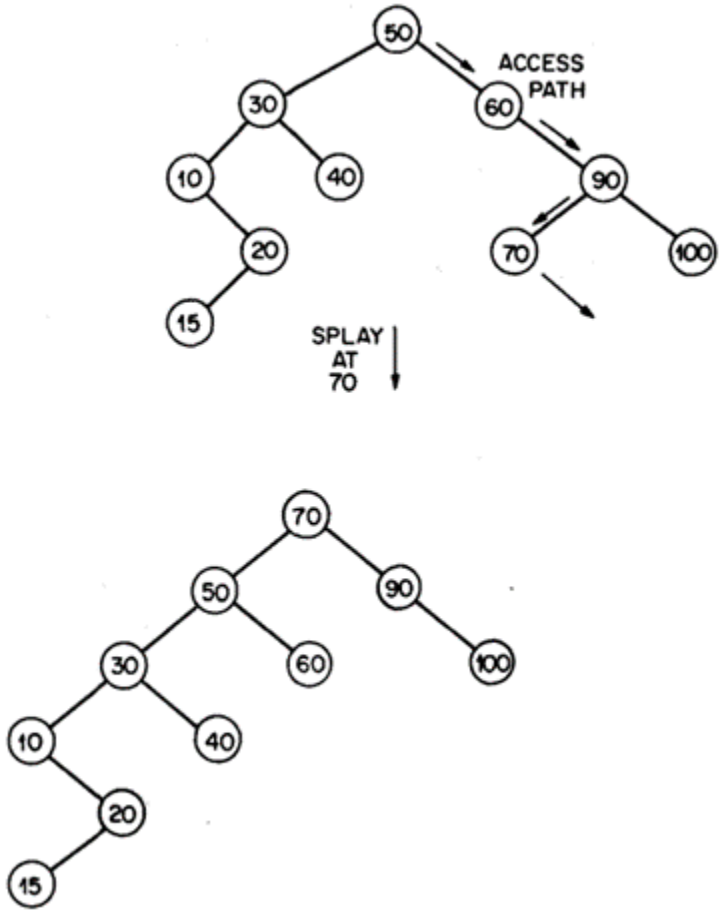

Рисунок 6. Попытка найти узел с ключом 80.

Для реализации функций $insert$ и $delete$ можно использовать функции $join$ и $split$.

Чтобы выполнить $join(t_1, t_2)$, мы ищем максимальный элемент i в t_1. Известно, что, элемент i будет иметь нулевой указатель в качестве правого сына. В качестве результата возвращаем новый корень t, с правым сыном t_2.

Чтобы выполнить $split(i, t)$, сначала производится $access(i)$, а затем в качестве

результата возвращаем два дерева, образованные левым и правым сыном t.

Чтобы выполнить $insert(i, t)$, мы производим поиск пустого узла, и, если достигли пустого указателя, заменяем его новым узлом, содержащим i, после чего производим $splay(i)$.

Чтобы выполнить $delete(i, t)$, мы производим $access(i, t)$, после чего заменяем t на $join(t1, t2)$, где t_1 – левое поддерево t, а t_2 – правое поддерево t.

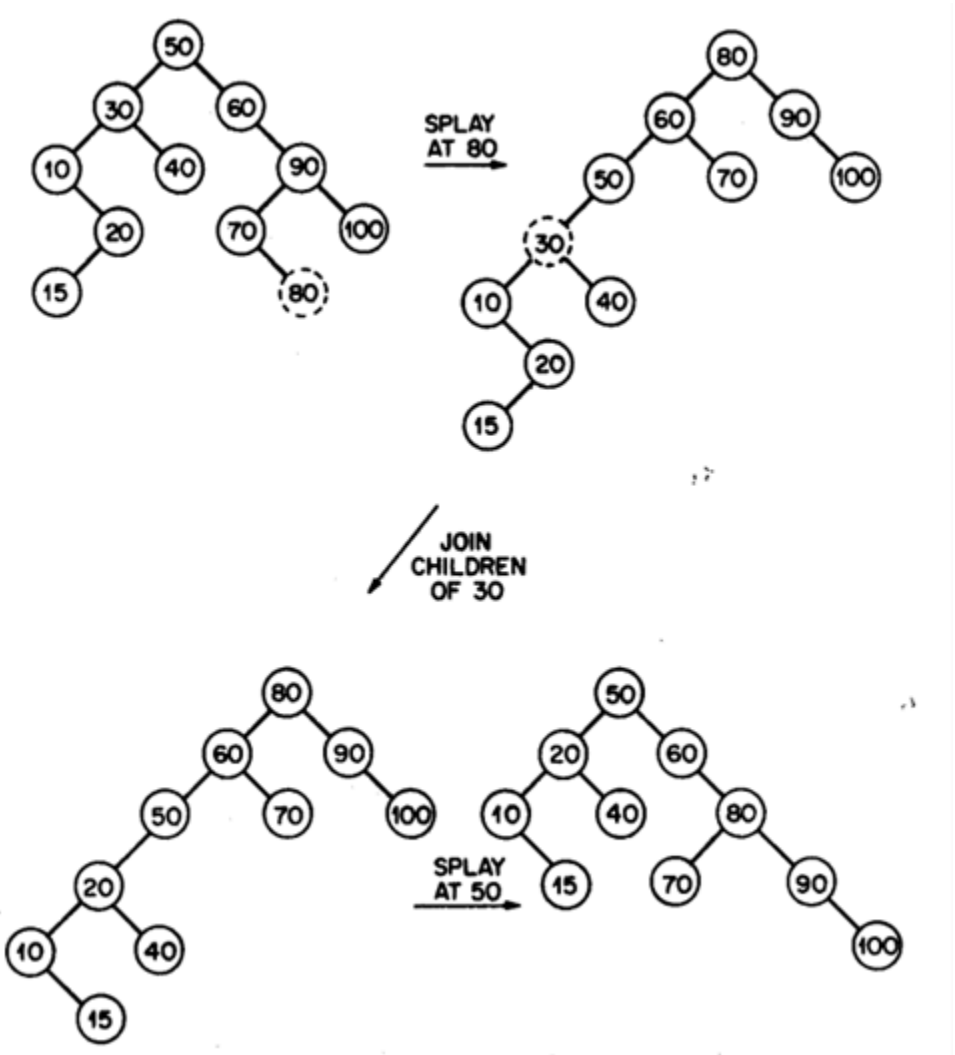

Рисунок 7. Реализации вставки и удаления. За вставкой ключа 80 последовало удаление узла с ключом 30.

AA-деревья

AA-дерево (англ. *AA-Tree*) — структура данных, представляющая собой

сбалансированное двоичное дерево поиска, которое является разновидностью *красно-черного дерева* с дополнительными ограничениями.

Наблюдение за другими структурами данных позволило понять, что можно избавиться от некоторого присущего им недостатка: большое количество рассматриваемых случаев во время балансировки можно заменить всего двумя операциями: *skew* и *split*. Во время работы бинарных деревьев их узлы могут принимать различного рода формы, которые должны быть обязательно рассмотрены для правильной балансировки. Это и есть причина, по которой они становятся сложными. Например, добавление ребра к узлу или может привести к пяти различным случаям. Таким образом, для балансировки, нужно рассматривать все эти случаи.

АА-дерево решает эту проблему следующим образом: к одной вершине можно присоединить вершину *только того же уровня, только одну и только справа*.

Свойства АА-дерева:

1. Уровень каждого листа равен 1 (в различных реализациях может быть как 0, так и 1).
2. Уровень каждого левого ребенка ровно на один меньше, чем у его родителя.
3. Уровень каждого правого ребенка равен или на один меньше, чем у его родителя.
4. Уровень каждого правого внука строго меньше, чем у его прародителя.
5. Каждая вершина с уровнем больше 1 имеет двоих детей.

Операции балансировки

> *Определение*. Горизонтальное ребро (англ. *Horizontal edges*) — ребро, соединяющее вершины с одинаковым уровнем.

$Skew(t)$ – устранение левого горизонтального ребра (придерживаемся правила 2). Делаем правое вращение (см.пункт Одиночное вращение), чтобы заменить поддерево, содержащее левую горизонтальную связь, на поддерево, содержащее разрешенную правую горизонтальную связь.

$Split(t)$ – устранение двух последовательных правых горизонтальных ребер. Делаем левое вращение (см.пункт Одиночное вращение) и увеличиваем уровень, чтобы заменить поддерево, содержащее две или более последовательных правильных

горизонтальных связи, на вершину, содержащую два поддерева с меньшим уровнем.

6. Проект

6.1. Средства реализации

Для реализации поставленной задачи хорошо подойдёт язык программирования C++ (стандарт C++14). Он позволяет работать непосредсвенно с памятью: выделять и очищать отдельные её области. Хорошим преимуществом считается его быстрота, которая достигается благодаря тому, что компилятор может применять все микрооптимизации на этапе сборки.
В динамических языках программирования (Python, JavaScript) выполнение происходит построчно, без компиляции, что отрицательно влияет на производительность.

6.2. Структуры данных

В обеих деревьях на минимальная структурная единица используется узел. Узел, в зависимости от реализации, содержит разные поля. Основными являются `key` `left` `right`.

Структура (*класс*) Node, используемая в реализации Splay-дерева:

```
class Node {
public:
    T key;
    Node* left;
    Node* right;
    Node* parent;
}
```

- Так как реализация не является рекурсивной, исползуется ссылка на родителя `Node* parent`, что требует больше памяти.

Структура (*struct*) Node, используемая в реализации AA-дерева:

```
class Node {
public:
    T key;
    Node* left;
    Node* right;
    unsigned int level;
}
```

`T key` - ключ, по которому производятся сравнения. Так как используется Template (шаблон), то имеется возможность использовать произвольные данные для хранения, ввиду чего не имеет фиксированного размера, занимаемого в памяти.
`Node* left` - ссылка на левого родителя, представляет собой узел.
`Node* right` - ссылка на правого родителя, представляет собой узел.
Тип *Node* имеет размер 8 байт в 64-разрядной системе.
`unsigned int level` - уровень текущего узла. Позволяет хранить значения от 0 до 4294967295 включительно. Использование именно этого типа данных обосновано тем, что уровень не может быть отрицательным (в данной реализации). Максимально возможная высота дерева при этом составит $2^{4294967295}$, чего с запасом хватит для выполнения огромного спектра задач.
Например, мы создаём АА-дерево, расчитанное на работу с целочисленными данными (*int*) в 64-битной системе. Тогда поле:

- `key` будет занимать 4 байта
- `left` и `right` будут занимать 16 байт
- `level` будет занимать 4 байта
  Таким образом, на один узел будет приходиться 24 байта в памяти. Для 1000000 хранимых элементов затраты памяти составят $1000000 * 24 = 24000000 \approx 24$ МБайт.

Аналогичные расчёты можно провести и с Splay-деревом. Тогда, дерево из 1000000 элементов, расчитанное на работу с данными типа *int* в 64-битной системе будет занимать $1000000 * (4 + 8 * 3) = 32000000 \approx 32$ Мбайт с учётом битового выравнивания.

Список литературы

- Splay-деревья. Статья на Вики ИТМО. *https://neerc.ifmo.ru/wiki/index.php?title=Splay-дерево*
- Sleator, Daniel D.; Tarjan, Robert E."Self-Adjusting Binary Search Trees" *http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf*
- Uppsala University — Balanced searched trees made simple http://user.it.uu.se/~arnea/ps/simp.pdf
- AA-деревья. Статья на Вики ИТМО. *https://neerc.ifmo.ru/wiki/index.php?title=AA-дерево*
- Реализация AA-дерева. *https://www.geeksforgeeks.org/aa-trees-set-1-introduction*
- Реализация Splay-дерева. *https://www.geeksforgeeks.org/splay-tree-set-3-delete*
- Необработанные указатели (C++). *https://learn.microsoft.com/ru-ru/cpp/cpp/raw-pointers?view=msvc-170*
- Типы данных. Статья на Metanit. *https://metanit.com/cpp/tutorial/2.3.php*
- Расставим точки над структурами C/C++. *https://habr.com/ru/post/142662/*
- AVL-деревья. Статья на Вики ИТМО. *https://neerc.ifmo.ru/wiki/index.php?title=АВЛ-дерево*
- Красно-чёрные деревья. Статья на Вики ИТМО. *https://neerc.ifmo.ru/wiki/index.php?title=Красно-черное_дерево*