Министерство образования Республики Беларусь
Белорусский государственный университет
Факультет прикладной математики и информатики
Кафедра дискретной математики и алгоритмики

СОГЛАСОВАНО
Заведующий кафедрой
_____ Котов В.М.
«10» января 2025 г.

СОГЛАСОВАНО
Декан факультета
_____ Орлович Ю.Л.
«14» января 2025 г.

Алгоритмы и структуры данных

Электронный учебно-методический комплекс для специальностей:
6-05-0533-09 «Прикладная математика»
6-05-0533-10 «Информатика»
6-05-0533-11 «Прикладная информатика»
6-05-0533-12 «Кибербезопасность»

В 3 частях. Часть 2

Регистрационный № 2.4.2-24 /562

Авторы:
Соболевская Е.П., кандидат физико-математических наук, доцент;
Котов В.М., доктор физико-математических наук, профессор;
Буславский А.А., старший преподаватель.

Рассмотрено и утверждено на заседании Научно-методического совета БГУ 16.01.2025 г., протокол № 6.

Минск, 2025

УДК 519.712(075.8)+004.422.63(075.8)
С 545

Утверждено на заседании Научно-методического совета БГУ.
Протокол № 6 от 16.01.2025 г.

Решение о депонировании вынес
Совет факультета прикладной математики и информатики
Протокол № 5 от 14.01.2025 г.

А в т о р ы:

Соболевская Елена Павловна, кандидат физико-математических наук, доцент кафедры дискретной математики и алгоритмики факультета прикладной математики и информатики БГУ;

Котов Владимир Михайлович, доктор физико-математических наук, заведующий кафедрой дискретной математики и алгоритмики факультета прикладной математики и информатики БГУ;

Буславский Александр Андреевич, старший преподаватель кафедры дискретной математики и алгоритмики факультета прикладной математики и информатики БГУ.

Рецензенты:

кафедра программного обеспечения информационных технологий, УО «Белорусский государственный университет информатики и радиоэлектроники» (заведующий кафедрой Лапицкая Н.В., кандидат технических наук, доцент),

Шлыков В.В., профессор кафедры математики и методики преподавания математики УО «Белорусский государственный педагогический университет имени Максима Танка», кандидат физ.-мат. наук, доктор пед. наук, доцент.

Соболевская, Е.П. Алгоритмы и структуры данных : электронный учебно-методический комплекс для специальностей 6-05-0533-09 «Прикладная математика», 6-05-0533-10 «Информатика», 6-05-0533-11 «Прикладная информатика», 6-05-0533-12 «Кибербезопасность». В 3 ч. Ч. 2 / Е.П. Соболевская, В.М. Котов, А.А. Буславский ; БГУ, Фак. прикладной математики и информатики, Каф. дискретной математики и алгоритмики. – Минск : БГУ, 2025. – 153 с. : ил. – Библиогр.: с. 147–148.

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Алгоритмы и структуры данных. Часть 2» предназначен для студентов специальностей 6-05-0533-09 «Прикладная математика», 6-05-0533-10 «Информатика», 6-05-0533-11 «Прикладная информатика», 6-05-0533-12 «Кибербезопасность».

В ЭУМК содержится материал по разделу: «Структуры данных для организации поиска элемента. Поисковые деревья». Изложение соответствует программе учебной дисциплины.

ОГЛАВЛЕНИЕ

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА **4**

1. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ **7**

1.1. Организация поиска. Бинарные поисковые деревья 7

1.1.1. Основные определения 7

1.1.2. Бинарные поисковые деревья 15

1.1.2.1. Представление дерева в памяти компьютера 15

1.1.2.2. Поиск ключа в дереве. Псевдокод 17

1.1.2.3. Добавление ключа в дерево. Псевдокод 18

1.1.2.4. Удаление ключа из дерева. Псевдокод 20

1.1.2.5. Обходы вершин бинарного дерева. Псевдокод 25

1.1.2.6. Алгоритмы решения базовых задач 27

1.1.2.7. Определение длины наибольшего полупути 32

1.1.2.8. Подсчёт числа наибольших полупутей 33

1.1.2.9. Нахождение средней по значению вершины полупути 38

1.1.2.10. Проверка бинарного дерева на поисковость 42

1.1.3. Прошитые деревья 46

1.2. Организация поиска. Сбалансированные поисковые деревья 51

1.2.1. АВЛ-дерево 51

1.2.1.1. Основные определения 51

1.2.1.2. Базовые операции и их вычислительная сложность 55

1.2.1.3. Восстановление сбалансированности. Повороты 57

1.2.1.4. Пример построения АВЛ-дерева 62

1.2.2. Красно-чёрное дерево 66

1.2.2.1. Основные определения 66

1.2.2.2. Добавление ключа в дерево. Восстановление RB-свойств 70

1.2.2.3. Пример построения красно-чёрного дерева 74

1.2.2.4. Удаление ключа из дерева. Восстановление RB-свойств 77

2. ПРАКТИЧЕСКИЙ РАЗДЕЛ **83**

2.1. Общие задачи по теме 83

2.2. Индивидуальные задачи по теме 85

2.3. Указания к решению задач 132

3. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ **140**

3.1. Набор тестовых заданий для проверки теоретических знаний 140

3.2. Набор тестов для проверки работоспособности программ 144

3.3. Средства диагностики 145

4. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ **147**

4.1. Рекомендуемая литература 147

4.2. Электронные ресурсы 148

Приложение 1. Листинг кода 149

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

Электронный учебно-методический комплекс (ЭУМК) по учебной дисциплине «Алгоритмы и структуры данных. Часть 2» предназначен для студентов специальностей:

6-05-0533-09 «Прикладная математика»,

6-05-0533-10 «Информатика»,

6-05-0533-11 «Прикладная информатика»,

6-05-0533-12 «Кибербезопасность».

Комплекс подготовлен в соответствии с требованиями Положения об учебно-методическом комплексе на уровне высшего образования, утвержденного Постановлением министерства образования Республики Беларусь от 26.07.2011 № 167. Содержание разделов ЭУМК соответствует образовательным стандартам, структуре и тематике учебных программ по дисциплине «Алгоритмы и структуры данных» для указанных специальностей.

Вторая часть электронного учебно-методического комплекса разработана по разделу «Структуры данных для организации поиска элемента. Поисковые деревья» учебной дисциплины «Алгоритмы и структуры данных».

Главные цели ЭУМК: помощь студентам в организации самостоятельной работы, повышение качества подготовки специалистов и усиление практико-ориентированности учебного процесса по дисциплине «Алгоритмы и структуры данных».

Дисциплина «Алгоритмы и структуры данных» для специальности 6-05-0533-09 «Прикладная математика» входит в компонент учреждения образования модуля «Дискретная математика и алгоритмика», для специальности 6-05-0533-10 «Информатика» дисциплина входит в государственный компонент модуля «Дискретные структуры и алгоритмы», для специальности 6-05-0533-11 «Прикладная информатика» дисциплина входит в государственный компонент модуля «Дискретная математика и алгоритмы», для специальности 6-05-0533-12 «Кибербезопасность» дисциплина входит в компонент учреждения образования модуля «Дискретная математика и алгоритмы».

Основные задачи, решаемые при изучении учебной дисциплины «Алгоритмы и структуры данных»:

– сформировать такие фундаментальные понятий, как размерность задачи и трудоёмкость алгоритма;

– изучить подходы для определения трудоёмкости алгоритма посредством составления и решения рекуррентных уравнений;

– изучить современные структуры данных, уметь обосновывать выбор соответствующей структуры в зависимости от набора базовых операций, используемых в алгоритме;

– уметь строить графовые модели и применять алгоритмы на графах для решения прикладных задач.

Дисциплины по теории алгоритмов кроме изучения теоретического материала предусматривают решение практических задач по построению

математической модели для решаемой задачи, разработке эффективного (с точки зрения трудоёмкости) алгоритма с последующим высоким уровнем его реализации на некотором языке программирования.

В результате изучения учебной дисциплины, обучающийся должен

знать:

– понятие размерности задачи и трудоёмкости алгоритма,

– основные способы решения рекуррентных уравнений,

– основные подходы при разработке эффективных алгоритмов,

– способы организации структур данных и технологию их использования,

– виды поисковых деревьев,

– базовые алгоритмы на графах.

уметь:

– сводить решение исходной задачи к решению подзадач и определять трудоёмкость алгоритмов на основе рекуррентных соотношений,

– выбирать подходящие структуры данных при разработке эффективного алгоритма решения задачи,

– реализовывать поисковые деревья,

– строить графовые модели и применять базовые графовые алгоритмы.

владеть:

– основными подходами для разработки эффективных алгоритмов: метод «разделяй и властвуй» и динамическое программирование;

– навыками реализации и использования современных структур данных.

Освоение учебной дисциплины «Алгоритмы и структуры данных» должно обеспечить формирование следующих компетенций: специализированной компетенции для специальности 6-05-0533-09 «Прикладная математика» – реализовывать современные структуры данных, строить графовые модели и применять алгоритмы на графах для решения прикладных задач, обосновывать корректность алгоритма и оценивать его асимптотическую сложность; базовой профессиональной компетенции для специальности 6-05-0533-10 «Информатика» – выполнять построение математических моделей и проводить их анализ в типовых задачах дискретной математики, интерпретировать получаемые результаты анализа математических моделей и осуществлять выбор структур данных для разработки эффективных алгоритмов решения прикладных задач; базовой профессиональной компетенции для специальности 6-05-0533-11 «Прикладная информатика» – характеризовать предмет и объекты дискретной математики и математической логики, использовать основные приёмы разработки эффективных алгоритмов и знания об основных структурах данных при решении прикладных задач; базовой профессиональной компетенции для специальности 6-05-0533-12 «Кибербезопасность» – понимать предмет и объекты дискретной математики и математической логики, использовать основные приёмы разработки эффективных алгоритмов и знания об основных структурах данных для решения прикладных задач.

ЭУМК состоит из следующих разделов:

1) *Теоретический*. Включает аннотацию ЭУМК, написанного в соответствии с программами дисциплины. Материал данного комплекса, наряду с конспектом лекций, может быть использован для самостоятельной подготовки студентов к контрольным заданиям и промежуточной аттестации (экзамену/зачету).

2) *Практический*. Содержит набор задач. Данные материалы используются при проведении лабораторных занятий и для самостоятельной работы над учебной дисциплиной.

3) *Раздел контроля знаний* представлен наборами тестов для проверки теоретических знаний и наборами тестов для проверки работоспособности программ.

4) *Вспомогательный раздел* включает рекомендуемую литературу и ссылки на электронные ресурсы.

Для организации самостоятельной работы студентов и самоподготовки по учебной дисциплине рекомендуется размещение программы курса, списка необходимой основной и дополнительной литературы, презентации лекций, заданий, тестов, методических рекомендаций на следующих доступных сетевых ресурсах факультета и университета:

1) Образовательный портал БГУ.
2) Образовательная платформа InsightRunner.

В образовательную платформу InsightRunner интегрирована свободная система управления содержимым (англ. CMS, или content management system) под названием MediaWiki. Вики-документы хорошо адаптированы для просмотра на мобильных устройствах, фрагменты кода на языках программирования отображаются с подсветкой синтаксиса.

Эффективность самоподготовки студентов целесообразно проверять в виде текущего контроля знаний в форме компьютерного тестирования как по отдельным темам, так и по разделам курса на Образовательной платформе InsightRunner. Тесты, разработанные в InsightRunner для проверки теоретических знаний, покрывают все разделы учебной дисциплины и генерируются автоматически. При составлении общих и индивидуальных заданий по учебной дисциплине необходимо предусмотреть возрастание их сложности. Рекомендуется делить задания на три модуля: задания, формирующие достаточные знания по изученному учебному материалу на уровне узнавания; задания, формирующие компетенции на уровне воспроизведения; задания, формирующие компетенции на уровне применения полученных знаний.

В образовательную платформу InsightRunner интегрирован модуль проверки исходного кода на плагиат (Акт о практическом использовании результатов исследования от 27.05.2022 №2.4/148). Модуль активно используется в учебном процессе в рамках дисциплин по теории алгоритмова в качестве диагностического инструментария и позволяет проверять самостоятельность выполнения студентами лабораторных заданий с целью недопущения элементов списывания и использования несанкционированных ресурсов в интернете.

1. ТЕОРЕТИЧЕСКИЙ РАЗДЕЛ

1.1. Организация поиска. Бинарные поисковые деревья

1.1.1. Основные определения

Определение 1.1. *Корневое дерево* T – орграф, который удовлетворяет следующим условиям:

1) имеется в точности одна вершина, в которую не входит ни одна дуга (корень дерева T);

2) в каждую вершину, кроме корня, входит ровно одна дуга;

3) из корня дерева есть путь к каждой вершине дерева.

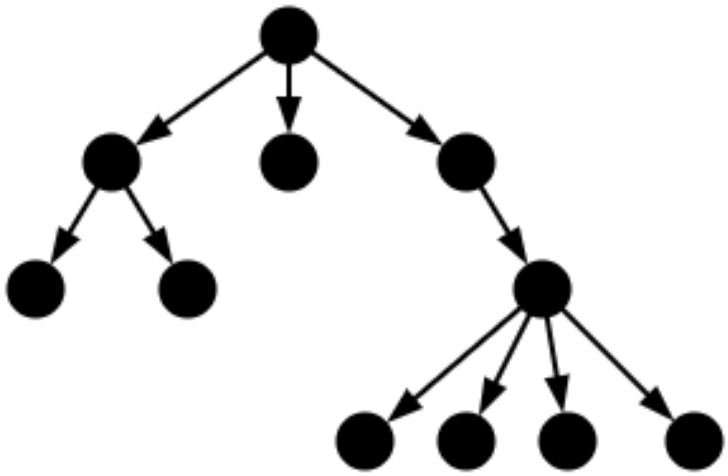


Рисунок 1.1 – Корневое дерево.

Пусть (u,v) – дуга корневого дерева, тогда вершину u называют *отцом* вершины v, а вершину v – *сыном* вершины u. Если у вершины u нет сыновей, то говорят, что вершина u – *лист* корневого дерева (в литературе листья часто называют *висячими* или *терминальными* вершинами дерева). Вершины корневого дерева, отличные от листьев, называют *внутренними* вершинами.

Пусть u – некоторая вершина корневого дерева, тогда все вершины v, которые достижимы из вершины u (т. е. существует путь из u в v), называют *потомками* вершины u (u – предок v). Подграф, состоящий из всех потомков вершины u (включая вершину u) и исходящих из них дуг, называется поддеревом с корнем в вершине u и обозначается через T^u.

Определение 1.2. *Высотой* вершины v в корневом дереве T называется длина наибольшего пути из вершины v до одного из его потомков (очевидно, что наибольший путь ведёт из вершины v в лист). Напомним, что длина пути определяется по количеству пройденных дуг.

Высота корневого дерева – высота корня. Высота дерева, состоящего из одной единственной вершины, полагается равной нулю.

На рисунке 1.2 возле каждой вершины корневого дерева указана его высота.

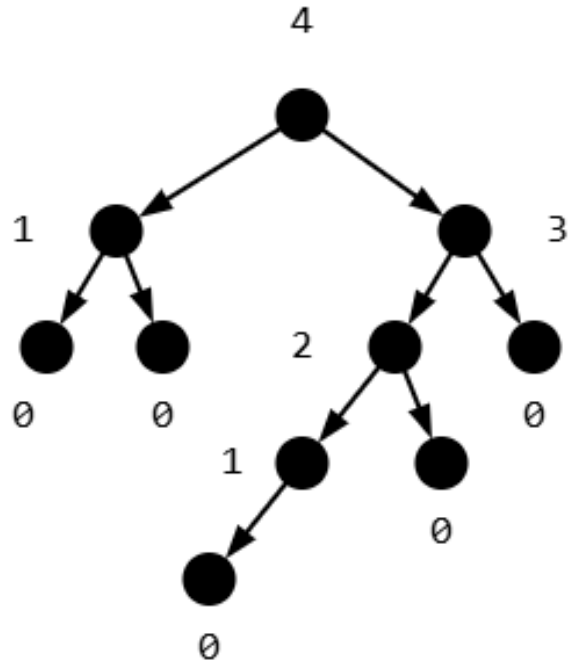


Рисунок 1.2 – Высоты вершин дерева.

Определение 1.3. *Глубиной* вершины *v* в корневом дереве *T* называется длина пути (в дугах) из корня дерева в вершину *v*. Нетрудно увидеть, что этот путь единственный, так как в каждую вершину дерева по определению входит только одна дуга.

На рисунке 1.3 возле каждой вершины корневого дерева указана её глубина.

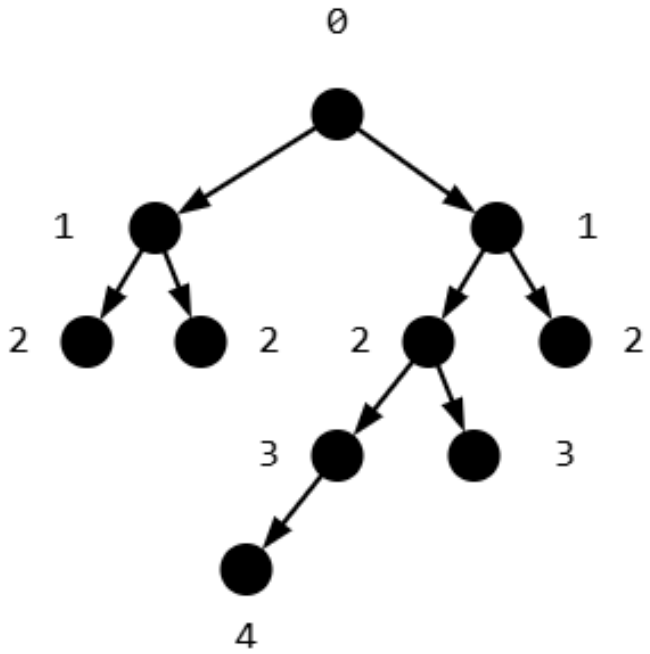


Рисунок 1.3 – Глубины вершин дерева.

Определение 1.4. *Уровнем* вершины *v* называется разность между высотой дерева *T* и глубиной вершины *v*.

На рисунке 1.4 вершины, которые находятся на одном уровне, выделены одним цветом. Номер уровня на рисунке 1.4 указан справа.

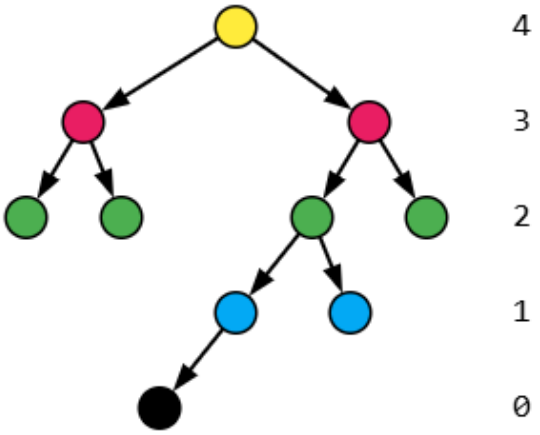


Рисунок 1.4 – Уровни вершин дерева.

Упорядоченное корневое дерево – это корневое дерево, у которого дуги, выходящие из каждой вершины, упорядочены (в дальнейшем будем считать, что они упорядочены слева направо).

Для корневых деревьев по разным критериям вводят инварианты сбалансированности, которые гарантируют, что все словарные операции (поиск элемента по заданному ключу, добавление элемента с заданным ключом, удаление элемента) выполняются за время $O(\log n)$, где n – число вершин в дереве, и при этом поддержка инвариантов сбалансированности также выполняется за время $O(\log n)$.

Определение 1.5. Корневое дерево называется *k-сбалансированным по высоте*, если для каждой вершины *v* высоты её максимального (по высоте) и минимального (по высоте) поддеревьев отличаются не более чем на *k*. Если $k = 1$, то будем говорить, что дерево сбалансировано.

На рисунке 1.5 приведён пример 3-сбалансированного по высоте дерева (возле каждой вершины дерева указана её высота).

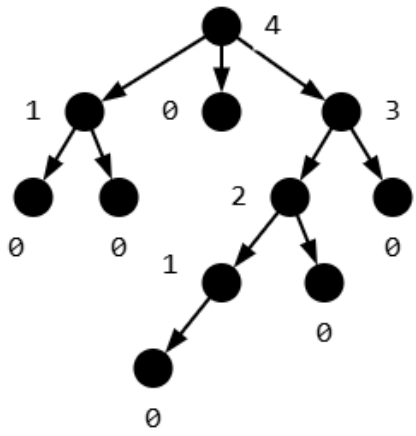


Рисунок 1.5 – Пример 3-сбалансированного по высоте дерева.

Определение 1.6. Корневое дерево называется *k-идеально сбалансированным по числу вершин*, если для каждой вершины v число вершин максимального (по числу вершин) и минимального (по числу вершин) поддеревьев отличаются не более чем на k. Если $k=1$, то будем говорить, что дерево идеально сбалансированно.

Каждое идеально сбалансированное дерево является сбалансированным, а вот обратное верно не всегда.

На рисунке 1.6 приведён пример 5-сбалансированного по числу вершин дерева (возле каждой вершины v указано число вершин в поддереве, корнем которого она является).

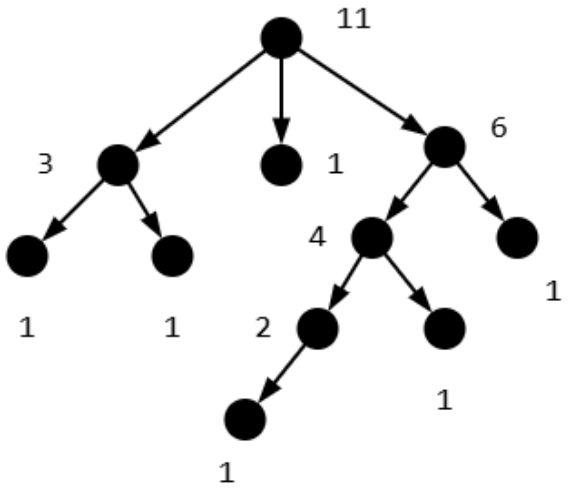


Рисунок 1.6 – Пример 5-сбалансированного по числу вершин дерева.

Определение 1.7. *Бинарное дерево* – это упорядоченное корневое дерево, у каждой вершины которого имеется не более двух сыновей. В бинарном дереве каждый сын произвольной вершины определяется как левый или правый.

Поддерево (если оно существует), корнем которого является левый сын вершины v, называется левым поддеревом вершины v. Аналогичным образом определяется правое поддерево для вершины v.

Предположим, что каждой вершине бинарного дерева соответствует некоторое ключевое значение (например, целое число).

Определение 1.8. Бинарное дерево называется *деревом поиска* (*бинарным поисковым деревом*), если оно организовано так, что для каждой вершины v справедливо утверждение, что все ключи в левом поддереве вершины v меньше ключа вершины v, а все ключи в правом поддереве – больше.

В дальнейшем будем предполагать, что в бинарном поисковом дереве нет двух вершин с одинаковыми ключевыми значениями, если не оговорено иное.

Высота h бинарного поискового дерева из n вершин равна $O(n)$ (максимальная высота достигается, когда вершины дерева вытянуты в линейный список). Высота h сбалансированного бинарного поискового дерева из n вершин равна $O(\log n)$.

Рассмотрим бинарное поисковое дерево T, приведённое на рисунке 1.7. Дерево T имеет высоту $h = 5$ и корень 10. Листьями дерева T являются вершины с ключами 3, 9, 14 и 50. Вершина с ключом 40 имеет высоту 1, глубину 3, уровень 2 (на втором уровне располагаются вершины с ключами 3, 7, 14 и 40).

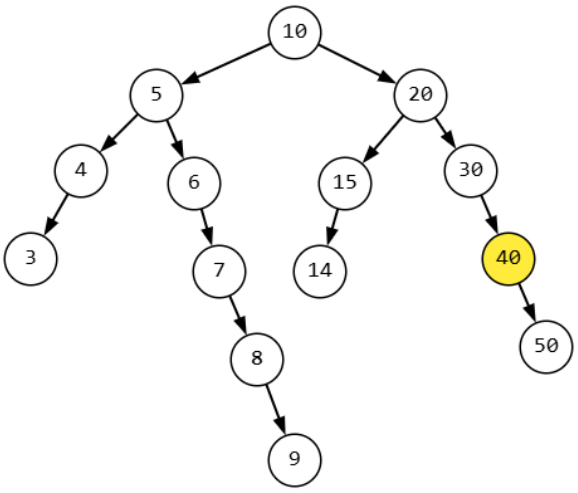


Рисунок 1.7 – Бинарное поисковое дерево.

Дерево T, приведённое на рисунке 1.7, не является сбалансированным, так как, например, для вершины с ключом 30 не выполняется инвариант сбалансированности по высотам (для простоты высоту несуществующего поддерева полагаем равной отрицательному числу -1). Не выполняется инвариант сбалансированности по высотам и для вершин с ключами 5, 6 и 7.

Определение 1.9. *Средней по значению из вершин* дерева является та вершина, которой соответствует такое ключевое значение x, что количество вершин, имеющих ключевое значение строго меньше x, равно количеству вершин, имеющих ключевые значения строго больше x.

Если количество вершин чётно, то будем считать, что средней по значению вершины не существует.

Если же дерево состоит из единственной вершины, то будем полагать, что эта вершина является средней по значению.

В дереве, приведённом на рисунке 1.7, средней по значению вершины нет, так как число вершин в дереве – чётно ($n = 14$). В дереве, приведённом на рисунке 1.8, число вершин $n = 11$, средняя по значению из вершин дерева – вершина с ключом 16 (вершины, ключи которых меньше и больше ключа 16, на рисунке 1.8 выделены разными цветами).

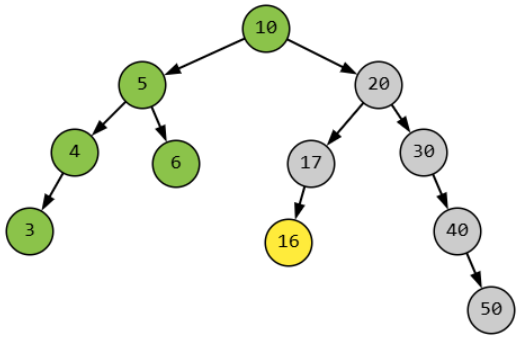


Рисунок 1.8 – Средняя по значению из вершин дерева (16).

Определение 1.10. *Путём* в дереве называется чередующаяся последовательность вершин и дуг $v_0, (v_0, v_1), v_1, (v_1, v_2), v_2, \ldots, v_k$. Известно, что в пути дуги не могут повторяться, а длина пути определяется как количество дуг в нём.

Для бинарного поискового дерева, приведённого на рисунке 1.9, последовательность

$$5, (5, 9), 9, (9, 14), 14, (14, 12), 12, (12, 13), 13$$

является путём длины 4 (вершины пути на рисунке 1.9 выделены). Будем говорить, что данный путь соединяет вершины 5 и 13.

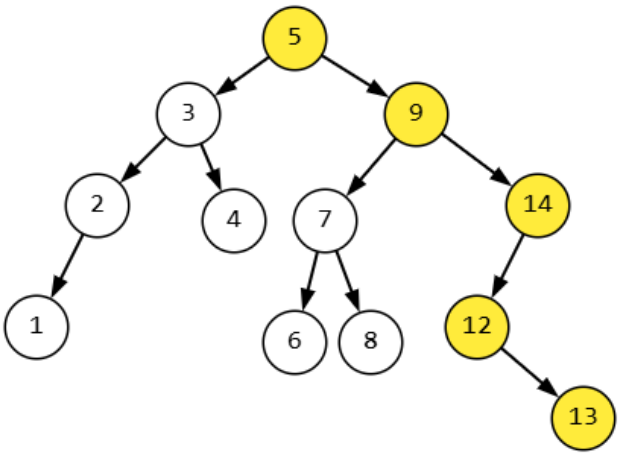


Рисунок 1.9 – Путь, соединяющий вершины с ключами 5 и 13.

Определение 1.11. Определим *центральную вершину некоторого пути P* как такую вершину *v*, что количество вершин в пути *P* до неё равно количеству вершин в пути *P* после неё (если количество вершин в пути *P* чётно, то будем говорить, что центральной вершины для заданного пути *P* не существует).

На рисунке 1.9 для пути, соединяющего вершины с ключами 5 и 13 центральной вершиной является вершина с ключом 14 (количество вершин в пути равно 5 – нечётное число, поэтому центральная вершина пути существует). В то же время, для пути, соединяющего вершины с ключами 5 и 8 центральной вершины не существует, так как этот путь содержит чётное число вершин.

Определение 1.12. Определим *среднюю вершину некоторого пути P* как такую вершину *v*, что количество вершин пути *P*, ключ которых меньше ключа вершины *v*, равно количеству вершин в пути *P*, ключ которых больше ключа вершины *v* (если количество вершин в пути *P* чётно, то будем говорить, что средней по значению вершины для пути *P* не существует).

На рисунке 1.9 для пути, соединяющего вершины 5 и 13, средней вершиной пути является вершина с ключом 12 (количество вершин в пути равно 5 – нечётное число, поэтому средняя вершина пути существует, если упорядочить ключи всех вершин заданного пути, то в упорядоченном массиве (5, 9, **12**, 13, 14) искомая вершина будет стоять по центру). В то же время, для пути, соединяющего вершины 5 и 8 средней вершины пути не существует, так как этот путь содержит чётное (4) число вершин.

Для *полупути* снимается ограничение на направление дуг. В полупути, как и в пути, дуги не могут повторяться.

Например, для дерева, приведённого на рисунке 1.10, последовательность

$$3,(3,5),5,(5,9),9,(9,7),7,(7,6),6$$

является полупутём длины 4. Будем говорить, что этот полупуть соединяет вершины с ключами 3 и 6.

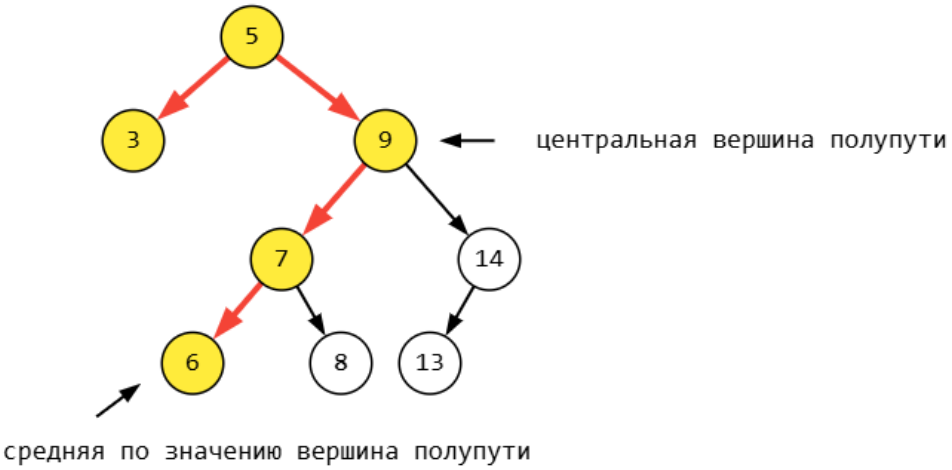


Рисунок 1.10 – Полупуть, соединяющий вершины с ключами 3 и 6.

Определение 1.13. *Корнем полупути* будем называть вершину этого полупути, которая находится на наибольшей высоте.

Например, для дерева, приведённого на рисунке 1.10, корнем полупути, соединяющего вершины с ключами 3 и 6, является вершина с ключом 5, а корнем полупути, соединяющего вершины с ключами 6 и 13, является вершина с ключом 9.

Центральная и *средняя по значению вершины полупути* определяются так же, как и для пути.

Например, для дерева, приведённого на рисунке 1.10, центральной вершиной полупути, который соединяет вершины с ключами 3 и 6, является вершина с ключом 9, а средней по значению вершиной этого полупути является вершина с ключом 6.

Определение 1.14. *Наибольшим полупутём в дереве* будем называть полупуть наибольшей длины.

Не стоит путать наибольший полупуть с максимальным – полупутём, который нельзя продолжить.

Наибольший полупуть всегда является максимальным, а вот обратное не всегда верно. Например, для дерева, изображённого на рисунке 1.10, полупуть, соединяющий вершины с ключами 6 и 8 $(6,(6,7),7,(7,8),8)$ является максимальным полупутём, но не является наибольшим, так как наибольший полупуть для этого дерева имеет длину 4 .

На рисунке 1.11 приведено бинарное поисковое дерево, в котором выделены пять наибольших полупутей (длина наибольшего полупути равна 4):

– первый наибольший полупуть соединяет вершины с ключами 3 и 6;
– второй наибольший полупуть соединяет вершины с ключами 3 и 8;
– третий наибольший полупуть соединяет вершины с ключами 3 и 15;
– четвёртый наибольший полупуть соединяет вершины с ключами 6 и 15;
– пятый наибольший полупуть соединяет вершины с ключами 8 и 15.

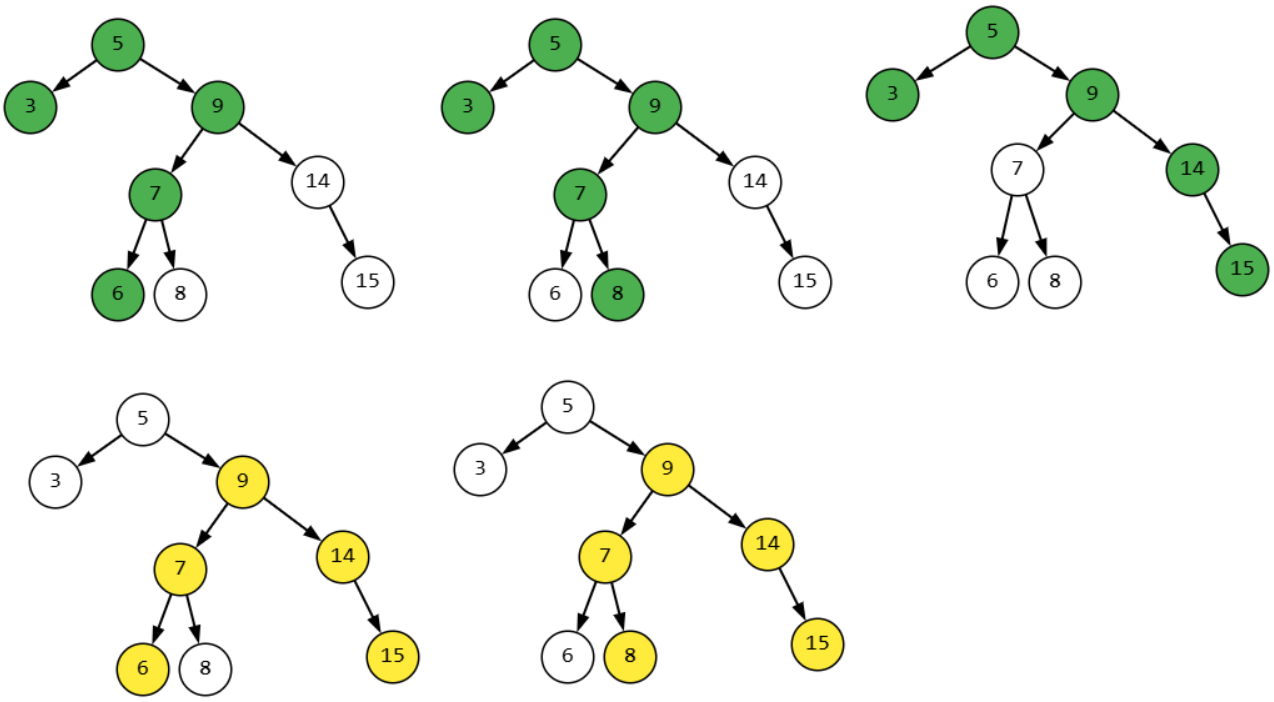


Рисунок 1.11 – Наибольшие полупути.

Все наибошьшие полупути для дерева, приведённого на рисунке 1.11, соединяли листья дерева. Следует отметить, что наибольший полупуть в дереве не обязательно соединяет листья. Например, если у корня дерева только одно поддерево, то наибольший полупуть может соединять корень дерева и лист, т. е. являться путём (см. рисунок 1.12).

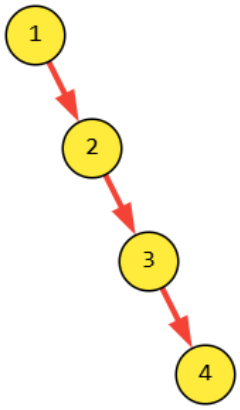


Рисунок 1.12 – Наибольший полупуть, соединяющий корень дерева и лист.

1.1.2. Бинарные поисковые деревья

1.1.2.1. Представление дерева в памяти компьютера

Как правило, каждая вершина дерева хранится в памяти отдельно в виде структуры некоторого типа. Каждый элемент содержит помимо информационной части (ключ вершины, дополнительные метки и т. п.), ссылку на другие вершины – на левого и правого сына и, возможно, ссылку на отца. Если поддерево отсутствует, то соответствующая ссылка имеет нулевое значение.

Заметим, что если известна ссылка на корень, то с помощью прохода по ссылкам можно достичь всех вершин дерева. Поэтому, чтобы говорить о представлении дерева в памяти, не нужно создавать список или массив всех вершин, а достаточно поддерживать только ссылку на корень. Остальные вершины доступны по связям от корня.

Для бинарного поискового дерева, приведённого на рисунке 1.13 слева, его представление в памяти компьютера в виде списковой структуры приведено на рисунке 1.13 справа.

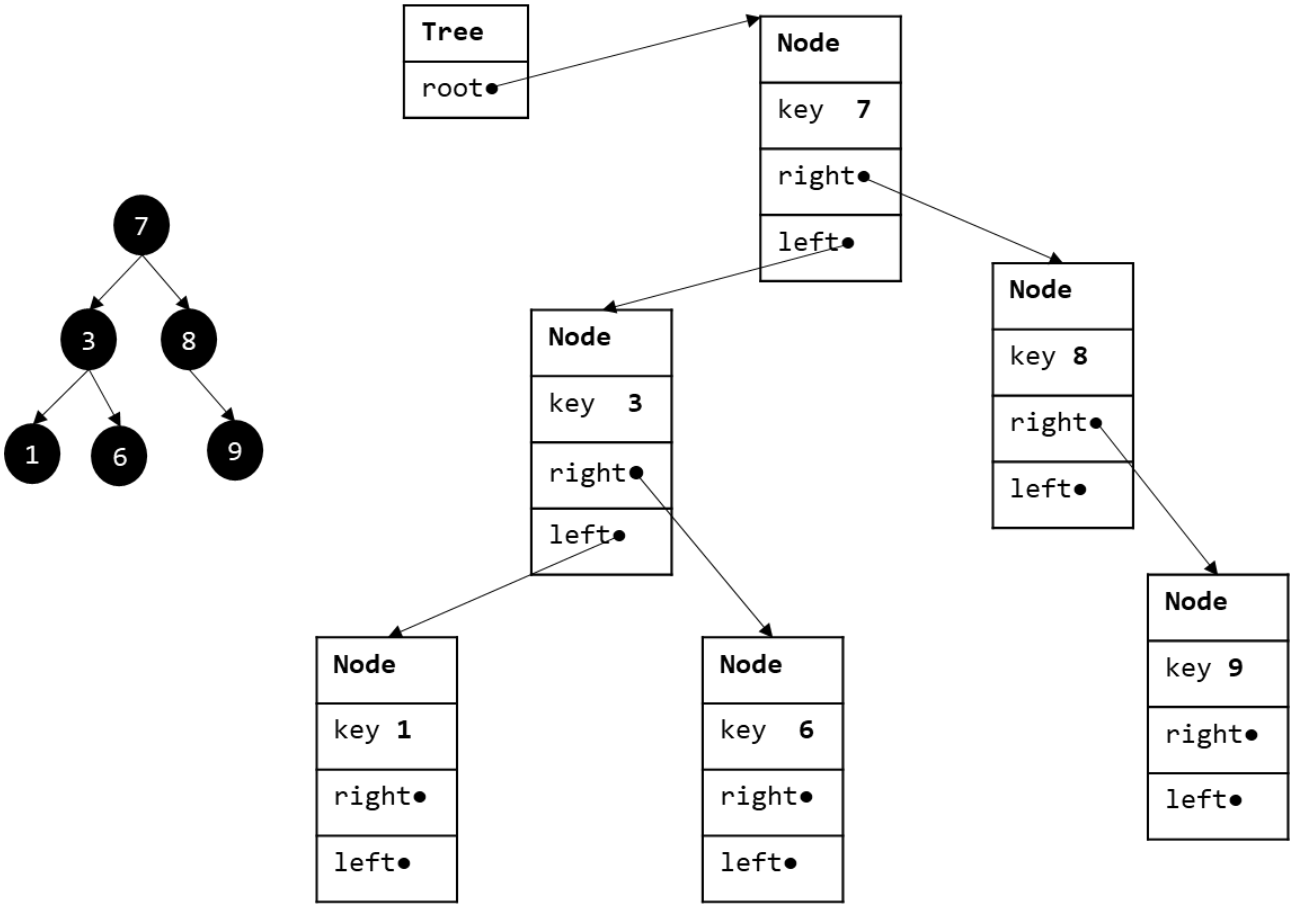


Рисунок 1.13 – Представление дерева в памяти компьютера.

В программе на объектно ориентированном языке программирования каждую вершину дерева удобно представлять в виде объекта определённого класса. Будем именовать класс словом **Node** (англ. – узел, вершина).

В простейшем случае объект вершины содержит три поля:

1) ключ (число, которое хранится в вершине);

2) ссылка на левого сына (возможно, нулевая, если левое поддерево отсутствует);

3) ссылка на правого сына (возможно, нулевая, если правое поддерево отсутствует).

Если у вершины отсутствует левое или правое поддерево, соответствующую ссылку обычно устанавливают в нулевое значение, принятое в конкретном языке программирования (`None, null, NULL, nullptr`).

В зависимости от задачи можно добавлять те или иные поля в класс вершины. Например, это может быть целочисленное поле для хранения высоты вершины или для подсчёта числа вершин в поддереве с корнем в текущей вершине. Важно не забывать поддерживать актуальность этих полей при изменении структуры дерева, если это требет решаемая задача (например, после выполнения удаления изменилась высота).

Кроме класса **`Node`** для вершины часто удобно создать класс **`Tree`**, представляющий всё дерево, который будет содержать единственное поле – ссылку на корень дерева. В зависимости от языка программирования класс вершины для удобства можно объявлять внутри класса дерева (вложенным классом), потому что вершина обычно не используется сама по себе.

С точки зрения теории объект, в котором нуль вершин, не может называться деревом. Однако на практике при программировании хорошо и удобно, когда структура данных «бинарное поисковое дерево» имеет состояние по умолчанию, когда в дереве нет ни одной вершины (дерево представляется пустым множеством ключей). Так, дерево можно создать изначально пустым, а затем по очереди добавить в него заданные ключи. Поддержка пустого состояния в классе дерева обычно требует некоторых усилий, что обусловлено удобством дальнейшей работы. Пустое дерево очевидным образом представляется объектом класса дерево, в котором поле вершины-корня нулевое.

Алгоритмы для работы с бинарными поисковыми деревьями можно реализовывать рекурсивно или нерекурсивно (итеративно). Организация дерева имеет рекурсивный характер: у любой вершины бинарного поискового дерева её левое и правое поддеревья по отдельности являются бинарными поисковыми деревьями. При использовании рекурсии код, вероятно, получится немного короче, но может работать на практике несколько медленнее. На теоретическую трудоёмкость способ реализации не влияет (один и тот же алгоритм записывается разными способами).

Для изложения алгоритмов мы будем использовать код на псевдоязыке, имеющем сходство с языком программирования Python. Действительно, исходный код программ на Python можно рассматривать как псевдокод: обычно он прост и понятен даже тем, кто с языком не сталкивался.

В псевдокоде

- оператор присваивания имеет вид `=`;
- проверка двух величин на равенство выполняется оператором `==`;
- ключевое слово **`def`** используется для объявления функций;
- проверка условия осуществляется при помощи оператора **`if`**;
- цикл с предусловием описывается через оператор **`while`**;
- для группировки операторов используются горизонтальные отступы.

Пусть переменная `x` в коде хранит целое число *x*. Создание вершины *v* с ключом *x* будем записывать так: `v = Node(x).`

В результате выполнения такого кода переменная `v` будет содержать ссылку на объект, который представляет вершину *v* в памяти компьютера.

Ключ вершины *v* будет доступен по имени `v.key`, левый сын будет обозначаться через `v.left`, правый сын – через `v.right`.

Нулевая ссылка имеет специальное значение `None`.

Корень дерева обозначим `tree.root`.

В Приложении 1 приведён листинг кода основных процедур работы с бинарными поисковыми деревьями.

1.1.2.2. Поиск ключа в дереве. Псевдокод

Отыскание в бинарном поисковом дереве заданного ключа *x* может быть реализовано так: выполняем спуск по дереву, при этом всякий раз направление дальнейшего движения (влево или вправо) выбирается исходя из того, меньше или больше искомый ключ текущего ключа в вершине. Начинаем с проверки корня дерева. Если дерево пустое, то ключ, который мы ищем, отсутствует. Иначе, если ключ совпадает с ключом корня, поиск завершился успешно, и мы возвращаем вершину. Если искомый ключ меньше ключа корня, продолжаем поиск в левом поддереве. Если ключ больше ключа в корне, ищем в правом поддереве. Процесс повторяется, пока ключ не будет найден или мы не придём к нулевому (отсутствующему) поддереву.

Для дерева, изображённого на рисунке 1.14, путь поиска вершины с ключом 9, выделен.

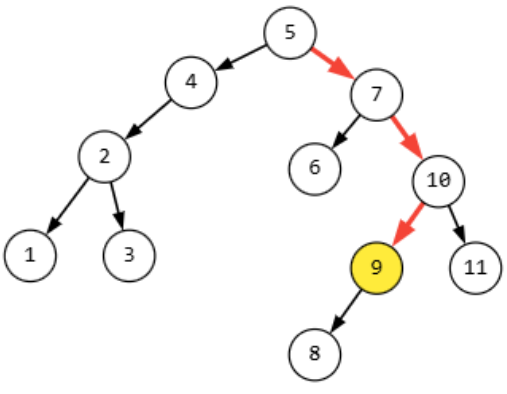


Рисунок 1.14 – Путь поиска вершины с ключом 9.

Рекурсивная реализация поиска ключа в дереве

Описанный алгоритм можно легко реализовать рекурсивно. Создаётся функция, принимающая на вход ссылку на вершину `v`, с которой будет начинаться поиск, и ключ `x`, который нужно найти. Поиск выполняется в поддереве, корнем которого является указанная стартовая вершина (возможно, поддерева не существует, если этот корень нулевой).

В качестве результата функция возвращает ссылку на найденную вершину или `None`, если ключа в поддереве нет.

```
def SearchRecursively(v, x):
    if v is None or v.key == x:
         return v
    elif x < v.key:
         return SearchRecursively(v.left, x)
    else:  # x > v.key
         return SearchRecursively(v.right, x)
```

Для поиска ключа во всём дереве при первом вызове указанной функции следует ей передавать в качестве вершины `v` корень дерева:

```
result = SearchRecursively(tree.root, x)
```

Нерекурсивная реализация поиска ключа в дереве

Тот же алгоритм может быть реализован нерекурсивно при помощи цикла.

```
def SearchIteratively(x):
    v = root
    while v is not None:
          if v.key == x:
             return v
          elif x < v.key:
             v = v.left
          else:  # x > v.key
             v = v.right
    return None
```

Отметим, что вершина с самым маленьким ключом – самая левая вершина в дереве, а с самым большим – самая правая.

1.1.2.3. Добавление ключа в дерево. Псевдокод

Рассмотрим, как можно реализовать добавление в дерево некоторого ключа *x*. Напомним, что по договорённости все ключи в дереве уникальны, поэтому при попытке добавить ключ, который уже есть, дерево не изменяется.

Как и прежде, выполняется спуск по дереву в поисках подходящего места для новой вершины. Спуск начинается с корня. Если новый ключ *x* меньше ключа текущей вершины, то спускаемся влево, если *x* больше, то вправо. В случае если *x* равен текущему ключу, то ничего делать не нужно, вставка не требуется.

Рекурсивная реализация добавления ключа в дерево

Используя рекурсию, операцию вставки часто реализуют так.

Создаётся функция, параметрами которой являются корень поддерева `v` (возможно нулевой) и ключ `x`, который нужно добавить.

Функция осуществляет вставку ключа в поддерево и возвращает корень поддерева после вставки (корень мог поменяться в единственном случае: поддерева изначально не существовало и была создана новая вершина, которая становится корнем; иначе функция просто возвращает тот же корень, который был ей передан).

```
def InsertRecursively(v, x):
    if v is None:
        return Node(x)
    if x < v.key:
        v.left = InsertRecursively(v.left, x)
    elif x > v.key:
        v.right = InsertRecursively(v.right, x)
    # v.key == x
    return v
```

Вызывать такую функцию нужно из корня дерева и при этом присваивать корню возвращаемое значение, чтобы был обработан случай изначально пустого дерева:

```
tree.node =  InsertRecursively(tree.root, x)
```

Нерекурсивная реализация добавления ключа в дерево

Если не использовать рекурсию, то код получается более сложным.

Спустимся посредством цикла по дереву, начиная с корня, в поисках места вставки. Чтобы, найдя пустое место, иметь возможность заменить его на новую вершину, в коде кроме ссылки на текущую вершину `v` поддержим ссылку на вершину-предка в переменной `parent`.

Если вставляемый ключ меньше текущего, пойдём влево; если больше, то двинемся вправо; если равен, то остановимся и выйдем: ключ уже есть в дереве и вставлять его не нужно.

Когда переменная `v` примет нулевое значение `None`, место для вставки найдено.

Создадим новую вершину `w` и в предке `parent` исправим ссылку `left` или `right` в зависимости от того, влево или вправо мы двигались из `parent` до того, как попали в `None`.

В отдельном случае, когда дерево изначально было пустое, нужно обновить переменную `root` (создан новый корень).

```
def InsertIteratively(x):
    parent = None
    v = tree.root
    while v is not None:
          parent = v
          if x < v.key:
             v = v.left
          elif x > v.key:
             v = v.right
          else: # x == v.key
             return
    w = Node(x)
    if parent is None:
          tree.root = w
    elif x < parent.key:
          parent.left = w
    elif x > parent.key:
          parent.right = w
```

1.1.2.4. Удаление ключа из дерева. Псевдокод

При удалении некоторой вершины v отдельно рассматривают три случая.

1) Вершина v не имеет поддеревьев, т. е. является листом дерева. В этом случае она просто уничтожается (вместе с дугой, ведущей в неё).

Пусть в программе переменная `v` – ссылка на вершину v, которую требуется удалить. Если у вершины `v` был предок `parent`, то соответствующая ссылка `parent.left` или `parent.right` обнуляется (рисунок 1.15); если же вершина `v` была корнем дерева, то обнуляется переменная `root`, которая хранит ссылку на корень.

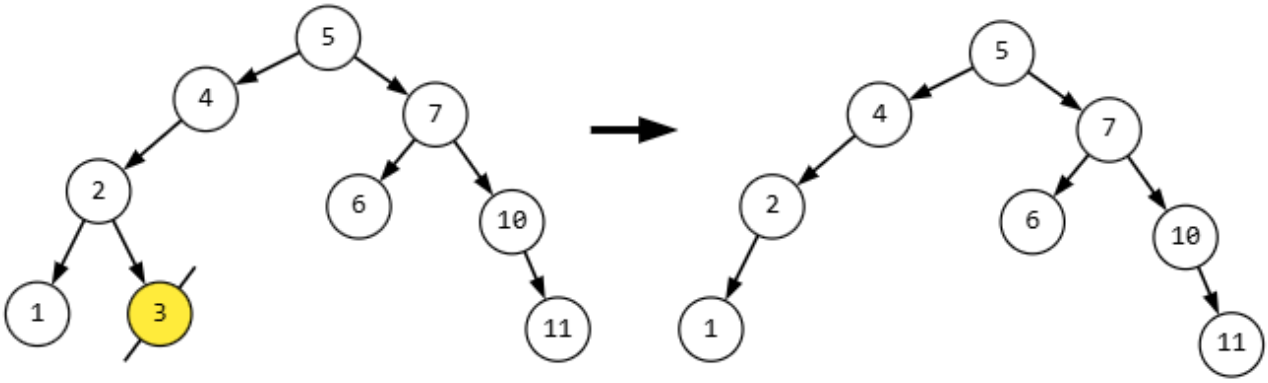


Рисунок 1.15 – Удаление вершины с ключом 3.

Заметим, что для выполнения такой операции необходимо иметь ссылку на предка вершины. Как правило, хранить для этого в каждой вершине дополнительное поле `parent` со ссылкой на родителя неразумно. Как можно обойти эту трудность, будет рассмотрено позже.

2) У вершины v есть только левое или только правое поддерево. В этом случае вершина v также убирается без перестроения дерева. Если у вершины v есть вершина-предок, то дуга, ведущая в вершину v из предка, перенаправляется на того сына вершины v, который существует (рисунок 1.16).

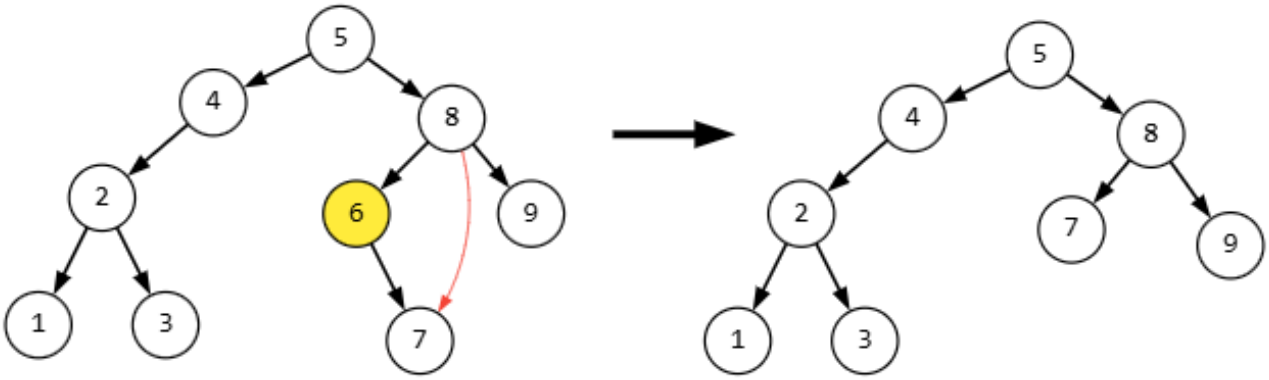


Рисунок 1.16 – Удаление вершины с ключом 6.

На рисунке 1.17 продемонстрирован отдельно случай удаления, когда удаляемая вершина v является корнем дерева и у неё есть только одно поддерево.

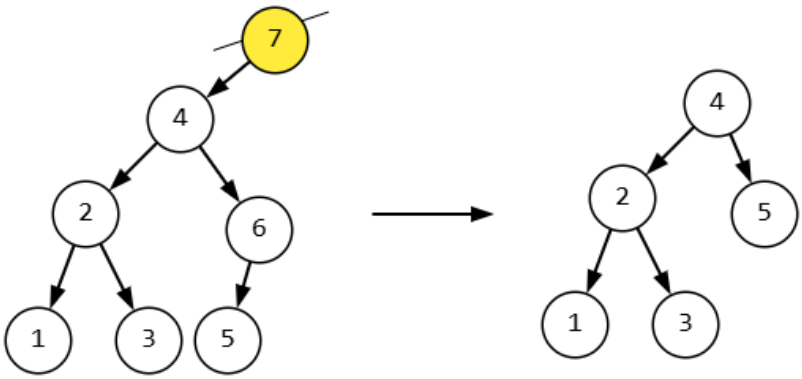


Рисунок 1.17 – Удаление корня дерева.

3) У вершины v есть оба поддерева. Это самый сложный случай, но он сводится к предыдущим. Здесь различают *левое* и *правое* удаление.

Вначале отыскивается или в левом поддереве вершины v самая правая вершина (*левое удаление*), или в правом поддереве – самая левая (*правое удаление*). Пусть это вершина w. Нетрудно понять, что вершина w будет иметь не более одного поддерева, поэтому она может быть удалена указанным выше методом (случай 1 или случай 2). Затем ключ вершины v заменяется на ключ вершины w.

На рисунке 1.18 продемонстрировано *левое* удаление вершины с ключом 7.

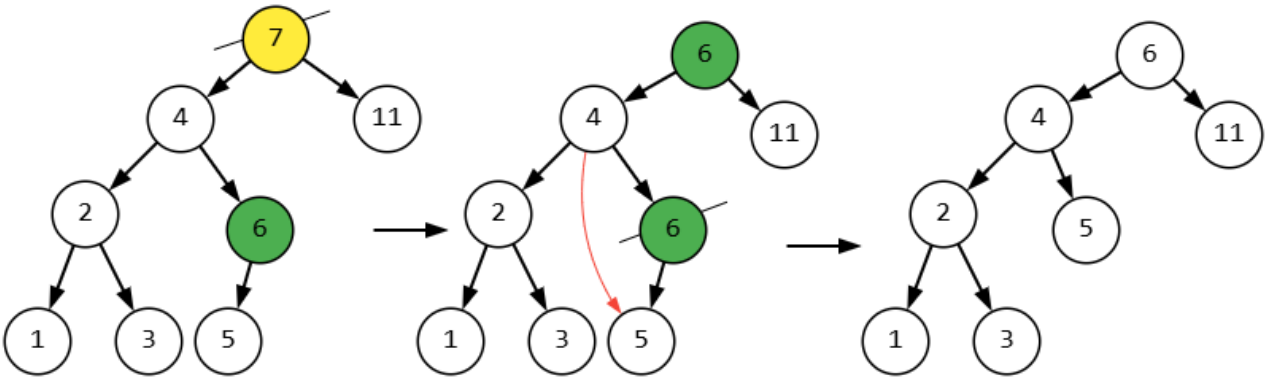


Рисунок 1.18 – Левое удаление вершины с ключом 7.

На рисунке 1.19 продемонстрировано *правое* удаление вершины с ключом 2.

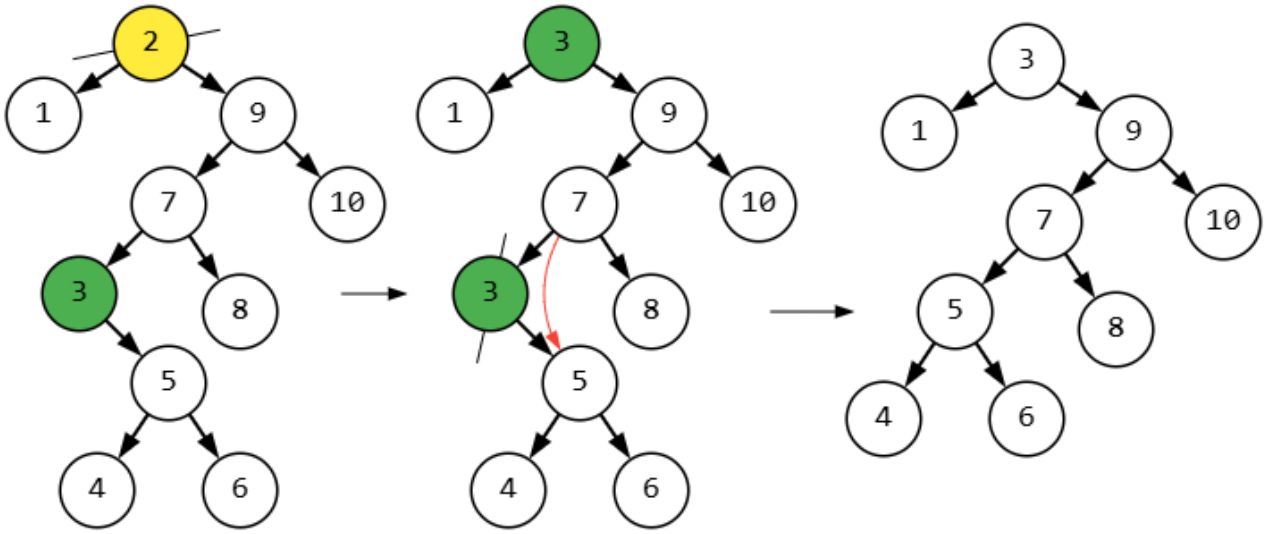


Рисунок 1.19 – Правое удаление вершины с ключом 2.

Рекурсивная реализация удаления вершины из дерева

При помощи рекурсии удаление вершины из дерева с заданным ключом обычно осуществляется так.

Реализуется функция, будем называть её `DeleteRecursively`, которая принимает на вход ссылку на вершину `v` и ключ `x`. Функция выполняет удаление вершины с ключом `x` в поддереве с корнем `v`, если такой ключ там есть. Функция возвращает новый корень поддерева после удаления (корень `v` может поменяться, если он сам окажется удалённым).

Функцию нужно вызывать из корня дерева и при этом присваивать корню возвращаемое значение:

```
tree.node = DeleteRecursively(tree.root, x)
```

В коде функции вначале делаются проверки ключа и выполняется спуск по дереву в поисках `x` (если меньше – идём влево, если больше – идём вправо). Когда функция `DeleteRecursively` оказывается вызванной для вершины `v` с ключом `x`, переходим непосредственно к удалению.

Сначала проверяется число потомков вершины `v`. Если левое и/или правое поддерево отсутствует, это простой случай. Иначе, когда есть оба поддерева, действия отличаются в зависимости от того, выполняется удаление левое или правое.

Рассмотрим случай *правого* удаления. Вызывается вспомогательная функция `FindMin`, которая ищет вершину `min_key` с наименьшим ключом в правом поддереве вершины `v`. Затем ключ в вершине `v` заменяется на найденный наименьший ключ `min_key`, а исходная вершина с ключом `min_key` удаляется через новый вызов функции `DeleteRecursively`.

```
def DeleteRecursively(v, x):
    if v is None:
        return None
    if x < v.key:
        v.left = DeleteRecursively(v.left, x)
        return v
    elif x > v.key:
        v.right = DeleteRecursively(v.right, x)
        return v
    # v.key == x
    if v.left is None:
        return v.right
    elif v.right is None:
        return v.left
    else:
        # both subtrees are present
        min_key = FindMin(v.right).key
        v.key = min_key
        v.right = DeleteRecursively(v.right, min_key)
        return v
```

Вспомогательная функция `FindMin` находит вершину с наименьшим ключом в поддереве с корнем `v`. Выполняется рекурсивный спуск по дереву всё время влево (там ключи меньше):

```
def FindMin(v):
    if v.left is not None:
        return FindMin(v.left)
    else:
        return v
```

Нерекурсивная реализация удаления вершины из дерева

Реализация удаления вершины с заданным ключом без использования рекурсии происходит намного сложнее. Для начала определим вспомогательную функцию, которая у вершины `parent` заменит вершину-сына `old` на вершину `new` (если в качестве `parent` будет передано значение `None`, то заменён будет корень всего дерева):

```
def ReplaceChild(parent, old, new):
    if parent is None:
         tree.root = new
    elif parent.left == old:
         parent.left = new
    elif parent.right == old:
         parent.right = new
```

Для поиска вершины с ключом `x` можно было бы воспользоваться ранее разработанной функцией поиска, но она потребовала бы доработки: для удаления нужно знать не только ссылку на саму вершину `v` с ключом `x`, но и ссылку на её предка `parent`.

```
def DeleteIteratively(x):
  parent = None
  v = tree.node
  while True
     if v is None:
       return
     if x < v.key:
       parent = v
       v = v.left
     elif x > v.key:
       parent = v
       v = v.right
     else:  # x == v.key
       break
  result = None
  if v.left is None:
     result = v.right
  elif v.right is None:
     result = v.left
  else:
     min_node_parent = v
     min_node = v.right
     while min_node_left is not None:
         min_node_parent = min_node
         min_node = min_node.left
     result = v
     v.key = min_node.key
     ReplaceChild(min_node_parent, min_node, min_node.right)
  ReplaceChild(parent, v, result)
```

В целом логика удаления здесь та же, как и в рекурсивной реализации. Когда ищем вершину с наименьшим ключом в правом поддереве, мы также вынуждены поддерживать при спуске ссылку на предка, чтобы потом иметь возможность эту вершину удобно удалить (вершина `min_node_parent` является предком вершины `min_node`).

1.1.2.5. Обходы вершин бинарного дерева. Псевдокод

Многие алгоритмы, работая с бинарными корневыми деревьями, посещают в определённом порядке каждую вершину дерева.

Существуют три наиболее распространённых способа обхода вершин бинарного дерева:

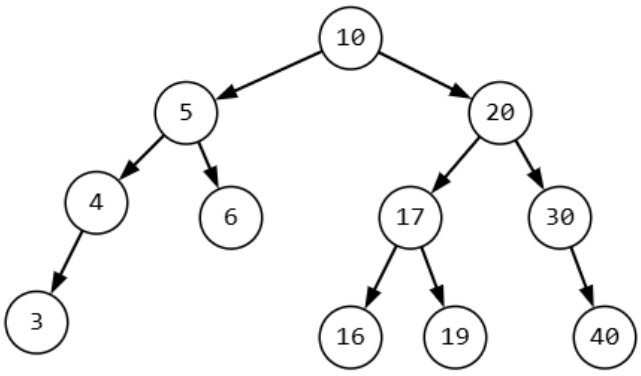
1) *прямой* (префиксный);
2) *обратный* (постфиксный);
3) *внутренний* (инфиксный).

Каждый обход выполняется за время $O(n)$, где n – число вершин в дереве.

Прямой порядок обхода заключается в том, что корень некоторого дерева посещается раньше, чем его поддеревья. Если при прямом обходе после обработки корня осуществляется переход в его левое (правое) поддерево, то обход называется прямым левым (правым). Далее приведена функция `PreOrderTraversal` прямого левого обхода. В коде использована вспомогательная функция `Action`, в которой могут выполняться некоторые действия с вершиной `v`, например, пересчёт некоторой характеристики вершины на основании данных, полученных от её отца (например, глубина вершины).

```
def PreOrderTraversal(v):
    if v is not None:
        Action(v)
        PreOrderTraversal(v.left)
        PreOrderTraversal(v.right)
```

Для дерева, приведённого на рисунке 1.20, указан порядок посещения вершин при выполнении прямого (левого и правого) обхода.



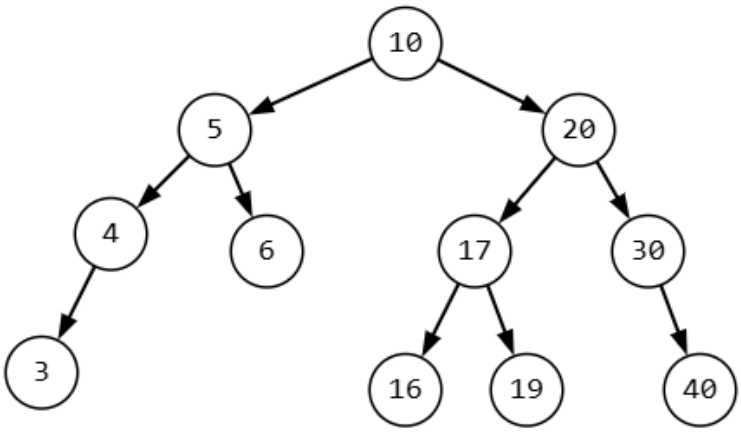

```
Прямой левый обход: 10, 5, 4, 3, 6, 20, 17, 16, 19, 30, 40

Прямой правый обход: 10, 20, 30, 40, 17, 19, 16, 5, 6, 4, 3
```

Рисунок 1.20 – Прямой порядок обхода.

Обратный порядок обхода заключается в том, что корень дерева посещается после его поддеревьев. Если при обратном обходе сначала осуществляется переход в левое (правое) поддерево корня, то обход называется обратным левым (правым). Далее приведена функция `PostOrderTraversal` обратного левого обхода, где вызывается вспомогательная функция `Action`, в которой могут выполняться некоторые действия с вершиной `v`, например подсчёт меток вершины исходя из меток её сыновей.

```
def PostOrderTraversal(v):
    if v is not None:
        PostOrderTraversal(v.left)
        PostOrderTraversal(v.right)
        Action(v)
```

Для дерева, приведённого на рисунке 1.21, указан порядок посещения вершин при выполнении обратного (левого и правого) обхода.



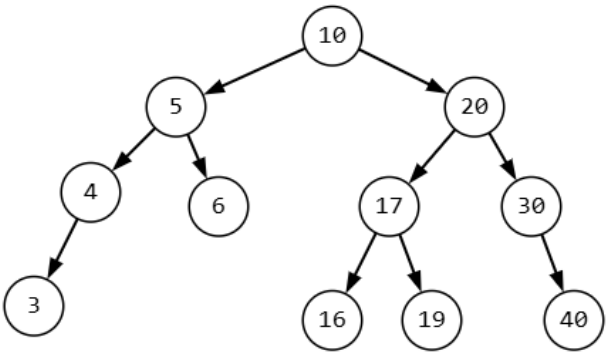

```
Обратный левый обход: 3, 4, 6, 5, 16, 19, 17, 40, 30, 20, 10

Обратный правый обход: 40, 30, 19, 16, 17, 20, 6, 3, 4, 5, 10
```

Рисунок 1.21 – Обратный порядок обхода.

Внутренний порядок обхода заключается в том, что корень дерева посещается после одного из его поддеревьев. Если корень посещается после обхода левого (правого) поддерева, то обход называется внутренним левым (правым). Ниже приведена функция `InOrderTraversal` внутреннего левого обхода. Во вспомогательной функции `Action` могут выполняться необходимые действия с вершиной `v`, например печать ключа вершины.

```
def InOrderTraversal(v):
    if v is not None:
        InOrderTraversal(v.left)
        Action(v)
        InOrderTraversal(v.right)
```

Для дерева, приведённого на рисунке 1.22, указан порядок посещения вершин при выаполнении внутреннего (левого и правого) обхода.




```
Внутренний левый обход: 3, 4, 5, 6, 10, 16, 17, 19, 20, 30, 40

Внутренний правый обход: 40, 30, 20, 19, 17, 16, 10, 6, 5, 4, 3
```

Рисунок 1.22 – Внутренний порядок обхода.

Заметим, что внутренний левый (правый) обход проходит все вершины дерева в порядке возрастания (убывания) ключей вершин. Таким образом, если нам необходимо отсортировать последовательность чисел, то мы можем сначала по заданной последовательности построить сбалансированное бинарное поисковое дерево T $\left(h(T)=\mathrm{O}\left(\log n\right)\right)$, а затем выполнить внутренний обход дерева.

Вычислительная сложность описанного выше алгоритма сортировки:

$$\mathrm{O}\left(n\cdot\log n\right)+\mathrm{O}\left(n\right)=\mathrm{O}\left(n\cdot\log n\right).$$

1.1.2.6. Алгоритмы решения базовых задач

1) **Подсчёт числа вершин в дереве**.

Для того, чтобы подсчитать число вершин в дереве достаточно выполнить обход дерева (прямой, обратный или внутренний), а в функции `Action(v)` выполнять увеличение счётчика числа вершин на 1.

Время работы содержательной части алгоритма (не учитывается построение дерева) – $\mathrm{O}\left(n\right)$, где n – число вершин в дереве.

2) **Определение высоты дерева. Проверка на сбалансированность.**

Сначала сформируем для каждой вершины метку высоты. Для этого достаточно выполнить обратный обход дерева.

Во время обратного обхода сформируем метки высот по следующим правилам:

– если вершина `v` – лист, то полагаем её высоту равной 0;

– если у вершины `v` только одно поддерево, то полагаем её высоту равной высоте единственного поддерева плюс 1;

– если у вершины `v` есть оба поддерева, то полагаем её высоту равной максимуму из меток высот её поддеревьев плюс 1.

Пример 1.1. Найти высоту дерева, изображённого на рисунке 1.23. Является ли это дерево сбалансированным по высоте?

Решение.

Для решения задачи сначала обратным обходом расставим метки высот (на рисунке 1.23 метки высот указаны возле каждой вершины).

Метка, которую получит корень, и есть высота дерева:

$$h(T) = 4.$$

Во время обратного обхода в функции `Action(v)` необходимо для вершины v находить модуль разности меток высот её сыновей, если хотя бы у одной вершины v это значение больше 1, то дерево не является сбалансированным по высоте.

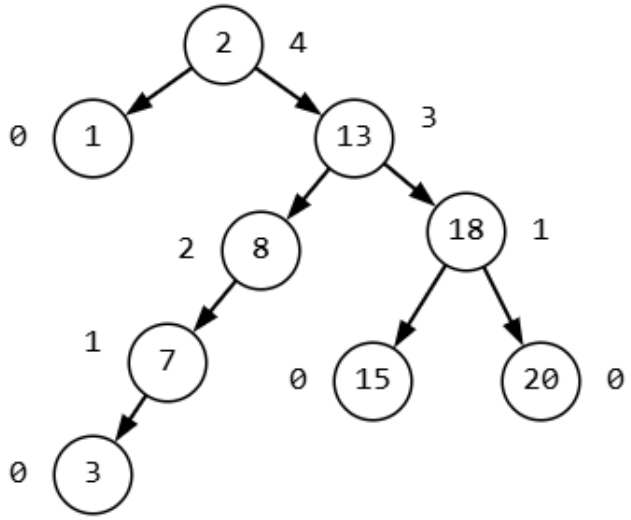


Рисунок 1.23 – Метки высот вершин дерева.

Для дерева, приведённого на рисунке 1.23, свойства сбалансированности по высотам нарушено у вершин с ключами 2 и 8 (для вершины с ключом 2 высота её левого поддерева равна 0, а правого поддерева равна 3; для вершины с ключом 8 высота её левого поддерева равна 1, а правого поддерева у вершины с ключом 8 нет, т. е. его высота полагается равной минус 1).

Дерево, изображённое на рисунке 1.23, не является сбалансированным по высоте, но является 3-сбалансированным.

Время работы содержательной части алгоритма (не учитывается построение дерева) – $O(n)$, где n – число вершин в дереве.

3) **Проверка на идеальную сбалансированность**.

Для решения задачи сначала обратным обходом каждой вершине v поставим метку, равную числу вершин в поддереве с корнем в вершине v.

Во время обратного обхода сформируем метки по следующим правилам:

– если вершина v – лист, то полагаем метку равной 1;

– если у вершины v только одно поддерево, то полагаем метку равной метке этого поддерева плюс 1;

– если у вершины v есть оба поддерева, то полагаем её метку равной сумме меток её поддеревьев плюс 1.

Пример 1.2. Является ли дерево, изображённое на рисунке 1.24, идеально сбалансированным по числу вершин?

Решение. Для решения задачи сначала обратным обходом расставим метки. Метка вершины `v` – число вершин в поддереве с корнем в данной вершине. На рисунке 1.24 метки указаны возле каждой вершины.

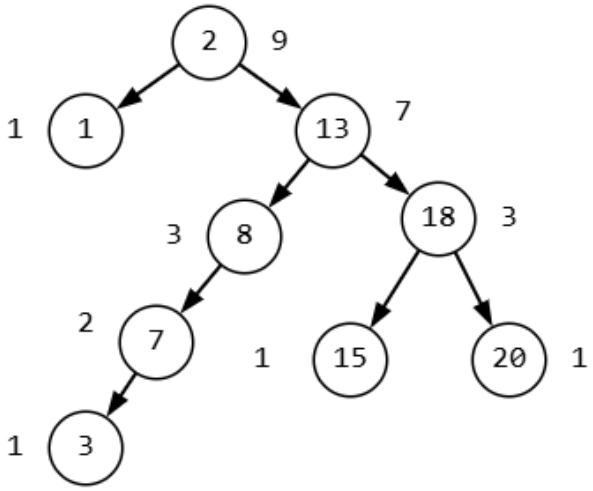


Рисунок 1.24 – Метки числа вершин в поддеревьях.

Во время этого же обратного обхода в функции `Action(v)` необходимо для каждой вершины `v` находить модуль разности меток её поддеревьев и, если хотя бы у одной вершины `v` это значение больше 1, то дерево не является идеально сбалансированным по числу вершин.

Для дерева, приведённого на рисунке 1.24, свойства сбалансированности по числу вершин нарушено у вершин с ключами 2 и 8 (для вершины с ключом 2 число вершин в её левом поддереве равно 1, а в правом – 7; для вершины с ключом 8 число вершин в её левом поддереве равно 2, а правое поддерево отсутствует, т. е. число вершин в нём равно 0).

Дерево, изображённое на рисунке 1.24, не является идеально сбалансированным по числу вершин, но является 6-идеально сбалансированным.

Время работы содержательной части алгоритма (не учитывается построение дерева) – $O(n)$, где n – число вершин в дереве.

4) **Нахождение средней по значению вершины среди вершин, удовлетворяющих заданным свойствам.**

Сначала нужно подсчитать число вершин, которые удовлетворяют заданным свойствам. Предположим, что это число m.

Если число m – чётно, то будем считать, что средней по значению вершины среди вершин, удовлятворяющих заданным свойствам не существует.

Если m – нечётно, то, если упорядочить ключи вершин, удовлетворяющих заданным свойствам, то в упорядоченном массиве искомая вершина будет стоять по счёту:

$$m' = \frac{m-1}{2} + 1.$$

Предположим, что число m – нечётно. Выполним внутренний обход бинарного поискового дерева, анализируя в функции `Action(v)` текущую вершину `v`, и, если она удовлетворяет заданным свойствам, то уменьшаем значение счётчика m' на 1. Если на некотором шаге текущее значение счётчика $m'=0$, то текущая вершина `v` является средней по значению среди вершин, удовлетворяющих заданным свойствам.

Если подсчёт числа вершин, удовлетворяющих свойству, выполнялся за время $O(n)$, то время работы содержательной части алгоритма (не учитывается построение дерева) – $O(n)$, где n – число вершин в дереве.

Пример 1.3. Для дерева, изображённого на рисунке 1.25, найти среднюю по значению вершину из листьев дерева.

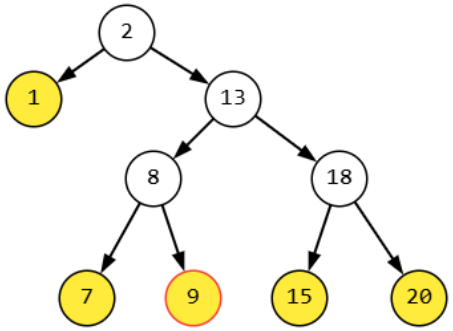


Рисунок 1.25 – Средняя по значению вершина из листьев дерева.

Решение. Для решения задачи сначала любым обходом подсчитаем число листьев в дереве (в функции `Action(v)` анализируем текущую вершину `v`, и, если она является листом, то увеличиваем значение счётчика m на 1).

Если число m – чётно, то будем считать, что средней по значению вершины среди листьев дерева нет.

Если число m – нечётно, то полагаем $m'=\frac{m-1}{2}+1$, выполняем внутренний обход дерева: во время внутреннего обхода, если текущая вершина является листом, то уменьшаем значение m' на 1 и, как только значение m' становится равным числу 0, средний среди листьев найден – это текущая вершина `v`.

Для дерева, приведённого на рисунке 1.25, число листьев $m=5$. Полагаем

$$m'=\frac{m-1}{2}+1=\frac{5-1}{2}+1=3.$$

Выполняем внутренний обход дерева:

✓ вершина с ключом 1 – лист, $m'=m'-1=3-1=2\neq 0$, продолжаем обход;

✓ вершина с ключом 2 – не лист, продолжаем обход;

✓ вершина с ключом 7 – лист, $m'=m'-1=2-1=1\neq 0$, продолжаем обход;

✓ вершина с ключом 8 – не лист, продолжаем обход;

✓ вершина с ключом 9 – лист, $m'=m'-1=1-1=0$ – вершина с ключом 9 является средней по значению вершиной из листьев дерева.

Время работы содержательной части алгоритма (не учитывается построение дерева) – $O(n)$, где n – число вершин в дереве.

Пример 1.4. Для дерева, изображённого на рисунке 1.26, найти среднюю по значению из вершин, у которых высота левого поддерева равна высоте правого.

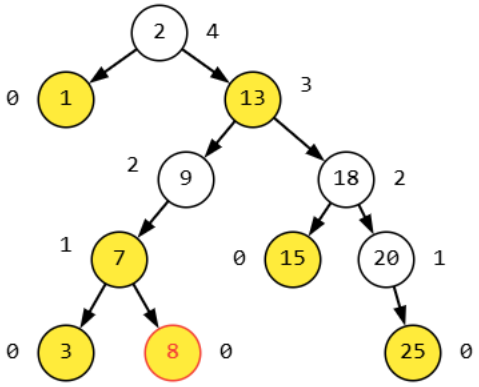


Рисунок 1.26 – Средняя по значению из вершин с равными высотами поддеревьев.

Решение. Для решения задачи сначала обратным обходом расставим метки высот и подсчитаем число вершин, у которых высоты поддеревьев равны. Если число m – чётно, то полагаем, что средней вершины, среди вершин, удовлетворяющих условию задачи, нет. Если число m – нечётно, то полагаем $m' = \frac{m-1}{2} + 1$, выполняем внутренний обход дерева: во время внутреннего обхода, если у текущей вершины метки сыновей совпадают, то уменьшаем значение m' на 1 и, как только значение m' становится равным 0, средняя вершина, среди вершин, у которых высоты поддеревьев равны, найдена.

На рисунке 1.26, вершины, у которых высоты поддеревьев равны, выделены. Так как число таких вершин $m = 7$ – нечётное число, то средняя по значению среди них существует. Полагаем $m' = \frac{m-1}{2} + 1 = \frac{7-1}{2} + 1 = 4$. Выполняем внутренний обход дерева:

✓ вершина с ключом 1 – удовлетворяет условию, $m' = m' - 1 = 4 - 1 = 3 \neq 0$, продолжаем обход;

✓ вершина с ключом 2 – не удовлетворяет условию, продолжаем обход;

✓ вершина с ключом 3 – удовлетворяет условию, $m' = m' - 1 = 3 - 1 = 2 \neq 0$, продолжаем обход;

✓ вершина с ключом 7 – удовлетворяет условию, $m' = m' - 1 = 2 - 1 = 1 \neq 0$, продолжаем обход;

✓ вершина с ключом 8 – удовлетворяет условию и $m' = m' - 1 = 1 - 1 = 0$, поэтому вершина 8 – средняя по значению из вершин, у которых высота левого поддерева равна высоте правого.

Если бы в задаче требовалось найти среднюю по значению из вершин дерева, у которых высота левого поддерева не равна высоте правого, то ответ был бы отрицательным (число вершин, удовлетворяющих этому свойству, равно числу 4 – чётное число, поэтому средней по значению вершины в заданном множестве вершин нет).

Время работы содержательной части алгоритма (не учитывается построение дерева) – $O(n)$, где n – число вершин в дереве.

1.1.2.7. Определение длины наибольшего полупути

Напомним, что наибольшим полупутём в дереве называется полупуть наибольшей длины. Для нахождения наибольшего полупути с корнем в вершине *v* достаточно знать для каждой вершины дерева её высоту. Для расстановки меток высот обойдём дерево обратным обходом (во время обратного обхода для каждой вершины *v* сначала посещаются её поддеревья). Во время обратного обхода сформируем метки высот по следующим правилам:

– если вершина *v* – лист, то полагаем её высоту равной 0;

– если у вершины *v* только одно поддерево, то полагаем её высоту равной высоте этого поддерева плюс 1;

– если у вершины *v* есть оба поддерева, то полагаем её высоту равной максимуму из меток высот её поддеревьев плюс 1.

Для определения длины наибольшего полупути с корнем в вершине *v*, которую обозначим $\mathrm{MSL}(v)$ (англ. maximum semipath length), необходимо найти сумму меток высот сыновей вершины *v* и увеличить это число на 1, если у вершины *v* только один сын, и на число 2, если у вершины *v* есть оба сына (для дерева, приведённого на рисунке 1.27, возле каждой вершины указаны метки).

Для определения длины *наибольшего полупути для всего дерева*, которую будем обозначать $\mathrm{MSL}(T)$, нужно выбрать вершины, для которых величина $\mathrm{MSL}(v)$ является максимальной. Заметим, что в дереве может быть несколько корней полупутей длины $\mathrm{MSL}(T)$, а через один и тот же корень наибольшего полупути может проходить несколько попарно различных полупутей наибольшей длины.

Для дерева *T*, приведённого на рисунке 1.27, длина наибольшего полупути $\mathrm{MSL}(T)=6$, а выделенные вершины с ключами 3 и 8 являются корнями полупутей наибольшей длины.

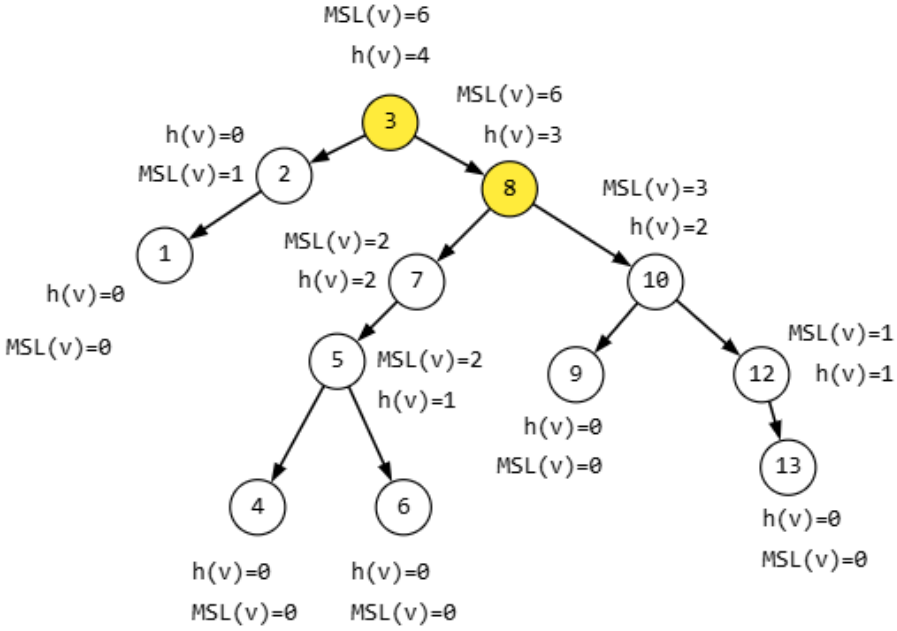


Рисунок 1.27 – Наибольший полупуть в дереве.

Следующий пример иллюстрирует, что корень дерева не всегда является корнем наибольшего полупути. Для дерева T, приведённого на рисунке 1.28, длина наибольшего полупути $\mathrm{MSL}(T)=4$. Выделенная вершина с ключом 8 – корень наибольшего полупути, через корень дерева (вершина дерева с ключом 3) ни один наибольший полупуть не проходит.

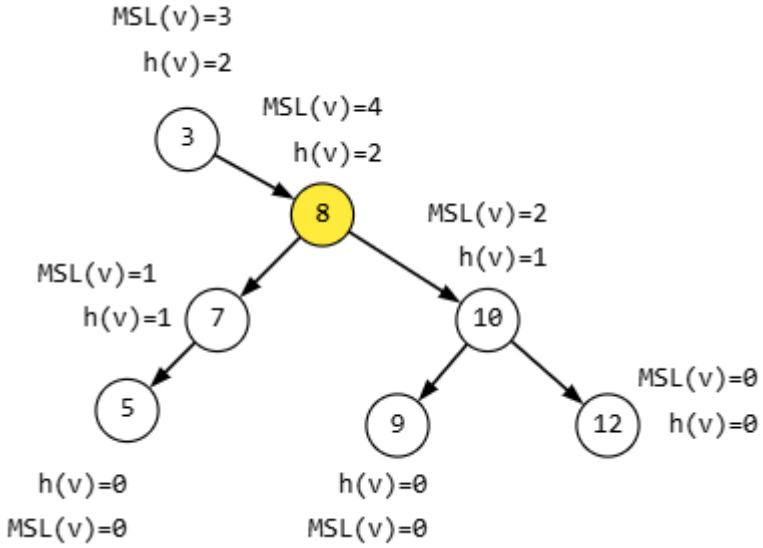


Рисунок 1.28 – Наибольший полупуть в дереве.

1.1.2.8. Подсчёт числа наибольших полупутей

Для подсчёта числа попарно различных полупутей наибольшей длины, которые проходят через вершину v (данную величину будем обозначать через $c(v)$), можно использовать алгоритм, приведённый ниже.

Пусть u и w – сыновья вершины v, f – отец (если v не корень дерева), v' – брат (если есть) (рисунок 1.29).

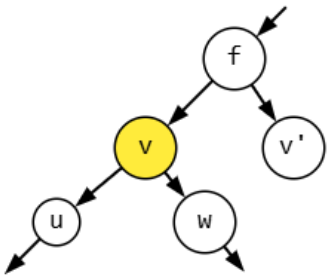


Рисунок 1.29

На первом шаге алгоритма выполним обратный обход дерева, формируя для каждой вершины v две метки: высоту $h(v)$ и число листьев $l_h(v)$, лежащих на пути из v на расстоянии $h(v)$. Во время этого же обхода найдём длину наибольшего полупути $\mathrm{MSL}(T)$ и корни полупутей наибольшей длины.

Правило формирования метки $h(v)$ приведено в п. 1.5.2.7, а для вычисления метки $l_h(v)$ поступим следующим образом:

– если вершина v – лист, то $l_h(v)=1$;

– если у вершины v есть только одно поддерево с корнем в вершине w, то $l_h(v)=l_h(w)$;

– если у вершины v есть оба поддерева с корнями u и w и $h(u)=h(w)$, то $l_h(v)=l_h(u)+l_h(w)$;

– если у вершины v есть оба поддерева и, например, $h(u)>h(w)$, то $l_h(v)=l_h(u)$.

Теперь для каждой вершины v определим число $b(v)$ попарно различных полупутей длины $\mathrm{MSL}(T)$, для каждого из которых данная вершина – корень наибольшего полупути (рисунок 1.30, а).

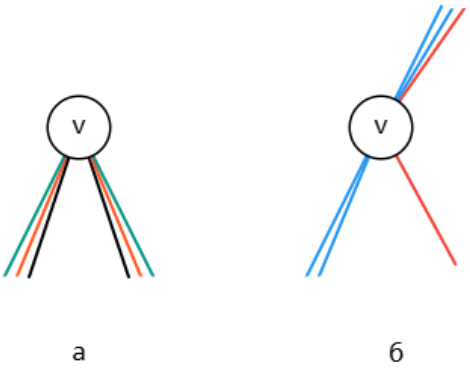


Рисунок 1.30 – Наибольшие полупути.

Для вычисления $b(v)$ поступим следующим образом:

– если вершина v – не корень наибольшего полупути, то $b(v)=0$;

– если вершина v – корень наибольшего полупути, то полагаем $h(v)=l_h(u)\cdot l_h(w)$, при этом если у вершины v нет некоторого сына, то в формуле соответствующее значение полагается равным единице.

На втором шаге алгоритма для каждой вершины v дерева вычислим число $a(v)$ попарно различных полупутей длины $\mathrm{MSL}(T)$, которые проходят через v и при этом приходят в неё сверху от отца (рисунок 1.30, б). Рассмотрим два алгоритма решения этой подзадачи.

Алгоритм 1.

Для вычисления $a(v)$ поступим следующим образом: выполним прямой обход дерева, формируя у каждой вершины v две метки: $h'(v)$ – длина наибольшего полупути из v в некоторую вершину, не принадлежащую поддеревьям вершины v; $l'(v)$ – число таких наибольших полупутей.

Если f – корень дерева, то положим $h'(f)=0$ и для простоты дальнейших вычислений полагаем $l'(f)=1$.

Пусть мы находимся в вершине f и для неё подсчитаны $h'(f)$ и $l'(f)$, тогда вычислим эти величины для сыновей v и v' вершины f (рисунок 1.31).

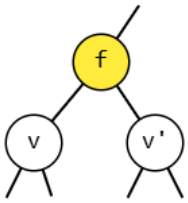


Рисунок 1.31.

Сначала предположим, что у вершины f есть только один сын, например, v. Тогда справедливы следующие соотношения:

$$h'(v)=h'(f)+1,$$
$$l'(v)=l'(f).$$

Теперь предположим, что у вершины f есть оба сына. Покажем, как вычислить величины h' и l' для одного из сыновей, например v (для другого сына вычисления выполняются аналогично):

– если $h'(f)=h(v')+1$, тогда

$$h'(v)=h(v')+2,$$
$$l'(v)=l'(f)+l_h(v');$$

– если $h'(f)<h(v')+1$, тогда

$$h'(v)=h(v')+2,$$
$$l'(v)=l_h(v');$$

– если $h'(f)>h(v')+1$, тогда

$$h'(v)=h'(f)+1,$$
$$l'(v)=l'(f);$$

Если f – корень дерева, то положим $a(f)=0$.

Пусть v – некоторая некорневая вершины и выполняется равенство

$$h'(v)+h(v)=\mathrm{MSL}(T),$$

тогда положим

$$a(v)=l'(v)\cdot l_h(v), \qquad (1.1)$$

в противном случае – $a(v)=0$.

После того как для каждой вершины v сформированы две метки $a(v)$ и $b(v)$, число полупутей длины $\mathrm{MSL}(T)$, проходящих через эту вершину, складывается из двух величин: $c(v)=a(v)+b(v)$.

Алгоритм 2.

Данный подход не предполагает поддержку меток $h'(v)$ и $l'(v)$, а величина $a(v)$ для вершины v пересчитывается во время прямого обхода дерева на основании аналогичной метки её отца.

Если v – корень дерева, то положим $a(v)=0$.

Если у вершины v есть только один сын, например u (см. рисунок 1.32), то положим $a(u)=a(v)+b(v)$.

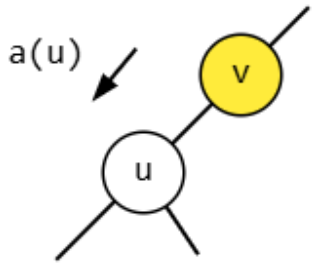


Рисунок 1.32.

Теперь предположим, что у вершины v есть оба сына (рисунок 1.33).

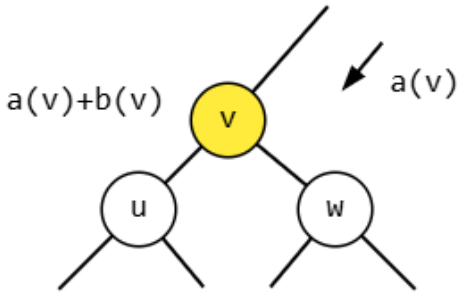


Рисунок 1.33.

Пусть в вершину v сверху пришло $a(v)$ полупутей длины, тогда поступим следующим образом:

– если $h(u)>h(w)$, то

$$a(u)=a(v)+b(v),$$
$$a(w)=b(v);$$

– если $h(u)<h(w)$, то

$$a(w)=a(v)+b(v),$$
$$a(u)=b(v);$$

– если $h(u)=h(w)$, то с учётом формулы (1.1) справедливы равенства:

$$a(u)=b(v)+l_h(u)\cdot l'(v)=b(v)+l_h(u)\cdot\frac{a(v)}{l_h(v)},$$

$$a(w)=b(v)+l_h(w)\cdot l'(v)=b(v)+l_h(w)\cdot\frac{a(v)}{l_h(v)}.$$

Пример 1.5. Для дерева, изображённого на рисунке 1.34, найти длину наибольшего полупути и для каждой вершины подсчитать число наибольших полупутей, которые через неё проходят.

Решение.

Продемонстрируем описанный выше алгоритм для дерева, приведённого на рисунке 1.34. Возле каждой вершины на рисунке 1.34 указаны вычисленные метки: $h(v), l_h(v), b(v), a(v)$.

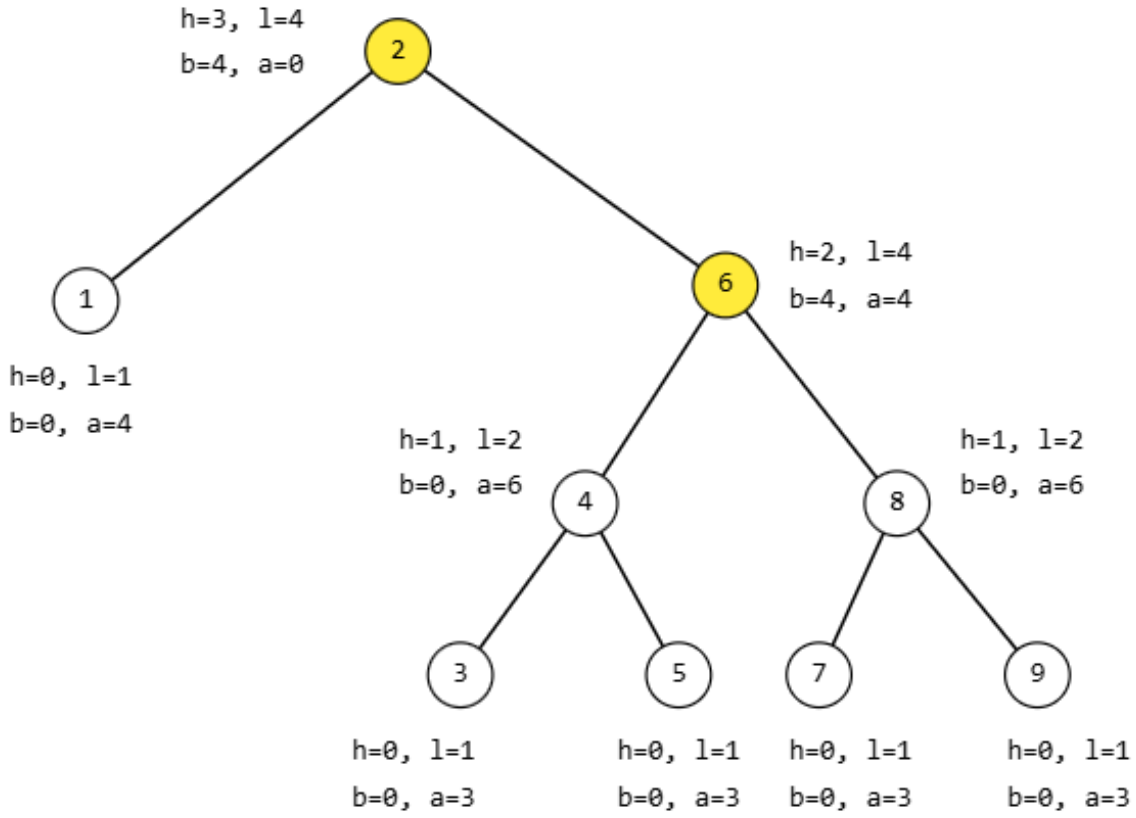


Рисунок 1.34 – Подсчёт числа наибольших полупутей.

Для данного примера длина наибольшего полупути $\mathrm{MSL}(T)=4$.

Вершины с ключами 2 и 6 являются корнями полупутей наибольшей длины.

Через вершину дерева с ключом 6 проходит наибольшее число полупутей наибольшей длины (8), из которых четыре наибольших полупути приходят в вершину 6 от её отца (это полупути, которые соединяют вершины 1 и 3, 1 и 5, 1 и 7, 1 и 9), а для четырёх полупутей наибольшей длины вершина 6 является корневой (это полупути, которые соединяют вершины 3 и 7, 3 и 9, 5 и 7, 5 и 9).

1.1.2.9. Нахождение средней по значению вершины полупути

Ранее были рассмотрены задачи, в которых необходимо было найти среднюю по значению вершину среди вершин, удовлетворяющих некоторым свойствам. В таких задачах сначала для вершины вычисляются характеристики, которые позволят сделать вывод о том, удовлетворяет или не удовлетворяет вершина заданному свойству, а затем, если число нужных вершин нечётно, то внутренним обходом осуществляется поиск среди них средней по значению.

В некоторых задачах требуется найти среднюю по значению вершину среди вершин некоторого пути (полупути). Как и ранее будем считать, что средней по значению вершины не существует, если количество вершин пути – чётное число.

Для пути, изображённого на рисунке 1.35, который соединяет вершины с ключами 17 и 21:

$17,(17,18),18,(18,30),30,(30,20),20,(20,23),23,(23,22),22,(22,21),21$

вершина с ключом 21 является средней по значению.

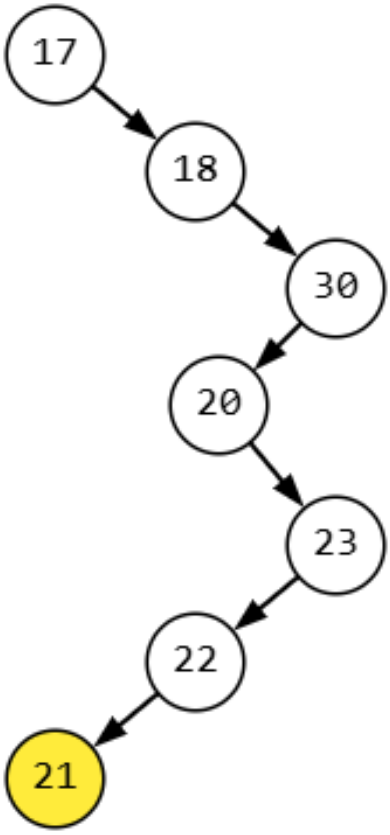


Рисунок 1.35 – Средняя по значению вершина пути.

Действительно, так как число вершин пути равно 7 – нечётное число, то средняя по значению вершина пути существует. Если упорядочить ключи вершин пути, то по центру в упорядоченном массиве будет стоять вершина с ключом 21.

	1	2	3	4	5	6	7
v	17	18	20	21	22	23	30

Для решения задачи можно применить алгоритм, который основывается на том факте, что дерево является поисковым, а поиск средней по значению вершины осуществляется среди вершин заданного пути.

Рассмотрим, как располагаются относительно друг друга ключи вершин, если последовательно двигаться вдоль пути из его корня. По мере движения будем определять номер ячейки таблицы, в которую надо занести элемент так, чтобы после прохождения всего пути элементы таблицы оказались упорядоченными по возрастанию.

Путь проходит через семь вершин, поэтому, как только в ячейку с номером четыре будет добавлен элемент, то он и будет средним по значению.

Движение начинаем из вершины с ключом 17 (корень пути).

1) Проходим вершину с ключом 17. Все оставшиеся вершины пути лежат в её правом поддереве, значит среди всех вершин пути вершина с ключом 17 имеет наменьшее значение. Заносим $v = 17$ в ячейку таблицы с номером $i = 1$.

i	1	2	3	4	5	6	7
v	17						

2) Проходим вершину с ключом 18. Все оставшиеся вершины пути лежат в её правом поддереве, значит среди всех оставшихся вершин пути вершина с ключом 18 имеет наименьшее значение. Заносим $v = 18$ в ячейку таблицы $i = 2$.

i	1	2	3	4	5	6	7
v	17	18					

3) Проходим вершину с ключом 30. Все оставшиеся вершины пути лежат в её левом поддереве, значит среди всех оставшихся вершин пути вершина с ключом 30 имеет наибольшее значение. Заносим $v = 30$ в ячейку таблицы $i = 7$.

i	1	2	3	4	5	6	7
v	17	18					30

4) Проходим вершину с ключом 20. Все оставшиеся вершины пути лежат в её правом поддереве, значит среди всех оставшихся вершин пути вершина с ключом 20 имеет наименьшее значение. Заносим $v = 20$ в ячейку таблицы $i = 3$.

i	1	2	3	4	5	6	7
v	17	18	20				30

5) Проходим вершину с ключом 23. Все оставшиеся вершины пути лежат в её левом поддереве, значит среди всех оставшихся вершин пути вершина с ключом 23 имеет наибольшее значение. Заносим $v = 23$ в ячейку таблицы $i = 6$.

i	1	2	3	4	5	6	7
v	17	18	20			23	30

6) Проходим вершину с ключом 22. Все оставшиеся вершины пути лежат в её левом поддереве, значит среди всех оставшихся вершин пути вершина с ключом 22 имеет наибольшее значение. Заносим $v = 22$ в ячейку таблицы $i = 5$.

i	1	2	3	4	5	6	7
v	17	18	20		22	23	30

7) Проходим вершину с ключом 21. Вершина 21 последняя вершина пути и её место – в единственной свободной ячейке таблицы $i = 4$. В ячейке с таким номером и должна стоять средняя по значению вершина пути.

i	1	2	3	4	5	6	7
v	17	18	20	21	22	23	30

Таким образом, в задаче можно ввести два счётчика: $turn_{left}$ – число поворотов налево и $turn_{right}$ – число поворотов направо (изначально оба счётчика равны 0).

Если из текущей вершины v путь идёт направо, то

$$turn_{right} = turn_{right} + 1.$$

Если из текущей вершины v путь идёт налево, то

$$turn_{left} = turn_{left} + 1.$$

Значения счётчиков $turn_{left}$ и $turn_{right}$ у отца вершины v указывают на то, сколько вершин пути, пройденных до v имели ключ больше, чем ключ вершины v ($turn_{left}$), а сколько – меньше ($turn_{right}$).

Пусть m – число вершин пути. Тогда, как только для вершины v актуализирован один из счётчиков и его значение стало равным

$$(m+1)/2,$$

то текущая вершина v является средней по значению и движение вдоль пути останавливается (или дошли до крайней вершины пути, которая и является средней по значению вершиной пути).

Пример 1.6.

Найти среднюю по значению вершину пути, приведённого на рисунке 1.36.

Решение. Применим описанный выше алгоритм для пути, приведённого на рисунке 1.36. Возле каждой вершины на рисунке 1.36 указаны значения счётчиков $turn_{left}$ и $turn_{right}$ после прохождения этой вершины.

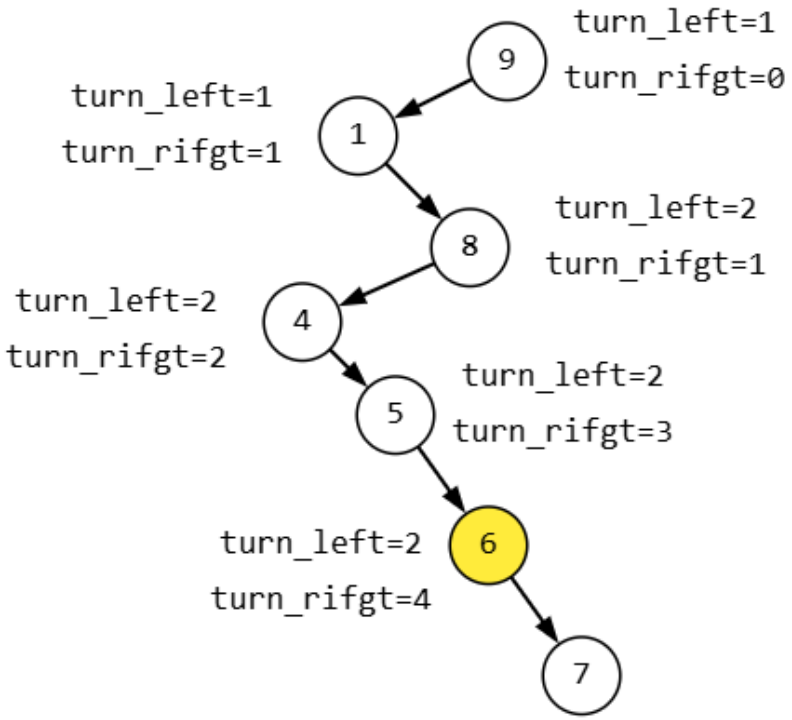


Рисунок 1.36 – Поиск средней по значению вершины пути.

Так как $(m+1)/2=(7+1)/2=4$, то движение вдоль пути было завершено при прохождении вершины с ключом 6, так как для этой вершины счётчик числа поворотов направо $turn_{right}$ стал равным 4. Вершина 6 – средняя по значению вершина пути.

Замечание 1.1. Если требуется найти среднюю по значению вершину некоторого полупути, то задача сводится к предыдущему случаю. Инициализация одного из счётчиков на начальном этапе выполняется в соответствии с тем, какая из веток полупути левая или правая содержит меньшее число вершин. Спуск начинается из коня полупути по большей ветке. Предположим, что у корня полупути *root* слева на полупути – x вершин, а справа – y вершин.

Пусть $x<y$, тогда начальная инициализация счётчиков: $turn_{left}=0,\ turn_{right}=x.$ Спуск начинается из корня полупути в правое поддерево, идёт по вершинам полупути и завершается, как только один из счётчиков после обработки текущей вершины получит значение, равное

$$\frac{(x+y+1)+1}{2}.$$

1.1.2.10. Проверка бинарного дерева на поисковость

Задано бинарное дерево. Необходимо проверить, является ли оно поисковым. Будем предполагать, что в бинарном поисковом дереве могут быть вершины с одинаковыми ключами. Тогда для того, чтобы дерево было поисковым, должно выполняться следующее требование: для каждой вершины *v* все ключи в левом поддереве вершины *v* меньше ключа вершины *v*, а все ключи в правом поддереве больше либо равны ключу вершины *v*.

Бинарное дерево подготовлено во входном файле в следующем виде.

Первая строка входного файла содержит единственное целое число n $\left(1 \le n \le 8 \cdot 10^5\right)$ – количество вершин в дереве.

Следующая строка содержит одно целое число m $\left(-2^{31} \le m \le 2^{31}-1\right)$ – значение в корневой вершине дерева.

В каждой из последующих $n-1$ строк через пробелы перечисляются три параметра m, p и c, которые задают какую-либо вершину дерева:

m – целое число $\left(-2^{31} \le m \le 2^{31}-1\right)$, значение, записанное в вершине;

p – целое число $\left(1 \le p \le n-1\right)$, номер строки входного файла, в которой был задан родитель текущей вершины (нумерация строк с нуля). Гарантируется, что p меньше, чем номер текущей строки.

c – может принимать одно из двух значений `L` или `R`. Значение `L` указывает на то, что текущая вершина присоединена к родительской слева, `R` – справа.

Гарантируется, что совокупность всех строк задаёт корректное бинарное дерево.

Гарантируется, что бинарное дерево задано корректно и для каждого элемента информация об отце поступила раньше.

На рисунке 1.37 приведено бинарное дерево, построенное по данным входного файла, приведённого ниже.

0-я строка файла	8
1-я строка файла	10
2-я строка файла	5 L 1
3-я строка файла	40 R 1
4-я строка файла	3 L 2
5-я строка файла	6 R 2
6-я строка файла	30 L 3
7-я строка файла	40 R 3
8-я строка файла	45 R 7

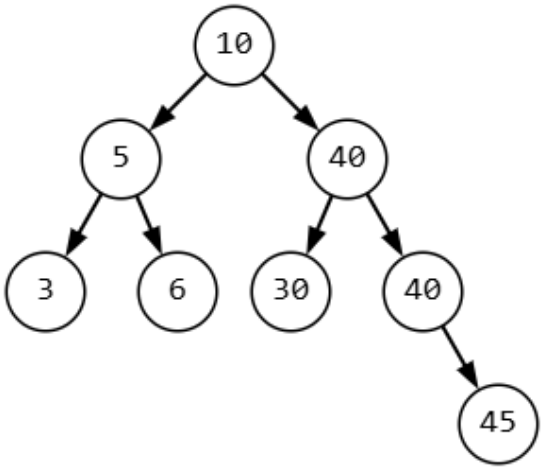


Рисунок 1.37 – Бинарное дерево.

Несложно видеть, что бинарное дерево, приведённое на рисунке 1.37, является поисковым (для каждой вершины *v* ключи всех вершин в её левом поддереве меньше, чем ключ *v*, а в правом поддереве – не меньше, чем ключ *v*).

На рисунке 1.38 приведены бинарные деревья, которые не являются поисковыми (выделены вершины, которые добавлены «не на свои позиции»).

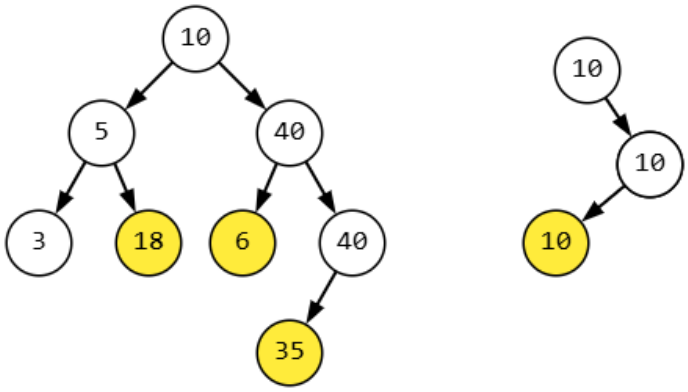


Рисунок 1.38 – Деревья не являются поисковыми.

Для проверки дерева на поисковость не верным является утверждение о том, что, если в дереве для каждой вершины *v* ключ её левого сына меньше ключа вершины *v*, а ключ её правого сына – не меньше, то дерево является поисковым. Действительно, для дерева, приведённого на рисунке 1.38 слева, это свойство выполняется, но дерево не является поисковым.

Для проверки дерева на поисковость не верным является утверждение о том, что, если в результате выполнения внутреннего обхода дерева получим упорядоченную последовательность ключей, то дерево является поисковым. Действительно, для дерева, приведённого на рисунке 1.38 справа, это свойство выполняется, но дерево не является поисковым.

Один из алгоритмов решения поставленной задачи заключается в формировании для вершины *v* промежутка допустимых значений.

1. Для корня дерева промежуток допустимых значений равен $(-\infty;+\infty)$.
2. Пусть *v* – текущая вершина и для неё промежуток допустимых значений равен $[L;R)$, тогда для левого сына вершины *v* промежуток допустимых значений равен $[L;v)$. Если ключ левого сына вершины *v* не попадает в указанный промежуток, то дерево не является поисковым.
3. Пусть *v* – текущая вершина и для неё промежуток допустимых значений равен $[L;R)$, тогда для правого сына вершины *v* промежуток допустимых значений равен $[v;R)$. Если ключ правого сынаПродемо вершины *v* не попадает в указанный промежуток, то дерево не является поисковым.

Так как в задаче гарантируется, что бинарное дерево задано корректно и для каждого элемента информация об отце поступила раньше, то проверка на поисковость будет корректно осуществляться при последовательном считывании данных из входного файла, а вычислительная сложность алгоритма – $O(n)$.

Применим описанный выше алгоритм для дерева, приведённого на рисунке 1.39. На рисунке 1.39 возле каждой вершины указан интервал допустимых значений. Дерево, приведённое на рисунке 1.39, не является поисковым, так как вершины с ключами 15, 3 и 35 не попадают в промежуток допустимых для них значений.

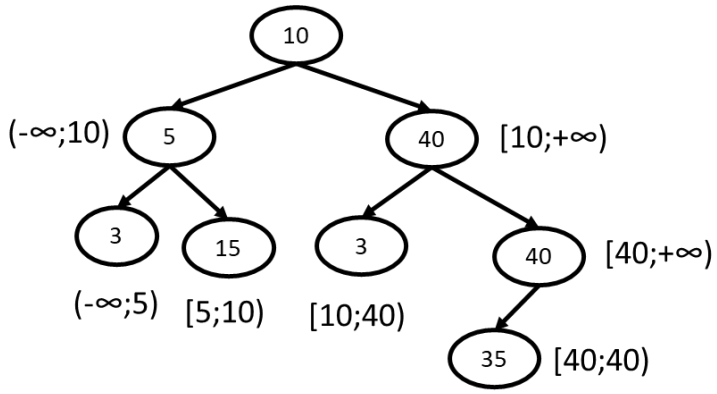


Рисунок 1.39 – Проверка бинарного дерева на поисковость.

Для программной реализации алгоритма достаточно ввести пять массивов и, последовательно считывая строки входного файла, осуществлять формирование диапазона допустимых значений для вершины *v* (формируется за константное время на основании ключа её отца и диапазона значений, который был сформирован для него) и проверять ключ вершины *v* на принадлежность сформированному диапазону.

Пример 1.7. Проверить, является ли бинарное дерево поисковым?

0-я строка файла	8
1-я строка файла	10
2-я строка файла	5 L 1
3-я строка файла	40 R 1
4-я строка файла	3 L 2
5-я строка файла	15 R 2
6-я строка файла	3 L 3
7-я строка файла	40 R 3
8-я строка файла	35 L 7

Решение.

В приведённой далее таблице показан сформированный диапазон значений для каждой вершины дерева. Формирование таблицы осуществляется по столбцам (можно задать таблицу пятью одномерными массивами). После считывания очередной строки входного файла для текущей вершины формируется диапазон допустимых значений на основании данных её отца и осуществляется проверка принадлежности ключа вершины данному диапазону.

Если ключ вершины не попадает в указанный диапазон, то дерево не является поисковым и работу программы можно завершить досрочно, не считывая оставшуюся в файле информацию до конца. Если ключ вершины попадает в указанный диапазон, то происходит считывание следующей строки входного файла.

1-я строка файла	2-я строка файла	3-я строка файла	4-я строка файла	5-я строка файла	6-я строка файла	7-я строка файла	8-я строка файла
10	**5**	**40**	**3**	**15**	**3**	**40**	**35**
–	L	R	L	R	L	R	L
–	1	1	2	2	3	3	7
$(-\infty;+\infty)$	$(-\infty;10)$	$[10;+\infty)$	$(-\infty;5)$	$[5;10)$	$[10;40)$	$[40;+\infty)$	$[40;40)$

В приведённом примере вершины с ключами 15, 3 и 35 не попадают в диапазон допустимых для них значений. Дерево не является поисковым (данным из входного файла соответствует дерево, изображёное на рисунке 1.39).

Пример 1.8. Проверить, является ли дерево поисковым?

0-я строка файла	8
1-я строка файла	10
2-я строка файла	5 L 1
3-я строка файла	40 R 1
4-я строка файла	3 L 2
5-я строка файла	6 R 2
6-я строка файла	30 L 3
7-я строка файла	40 R 3
8-я строка файла	45 R 7

Решение.

Дерево является поисковым так как ключи всех вершин попадают в сформированный диапазон допустимых для них значений (данным из входного файла соответствует дерево, изображённое на рисунке 1.37).

1-я строка файла	2-я строка файла	3-я строка файла	4-я строка файла	5-я строка файла	6-я строка файла	7-я строка файла	8-я строка файла
10	**5**	**40**	**3**	**6**	**30**	**40**	**45**
–	L	R	L	R	L	R	R
–	1	1	2	2	3	3	7
$(-\infty;+\infty)$	$(-\infty;10)$	$[10;+\infty)$	$(-\infty;5)$	$[5;10)$	$[10;40)$	$[40;+\infty)$	$[40;+\infty)$

1.1.3. Прошитые деревья

Предположим, что бинарное поисковое дерево содержит n вершин и хранится в памяти компьютера в виде структуры некоторого типа. Каждый элемент содержит помимо информационной части, ссылку на другие вершины – на левого и правого сына. Если поддерево отсутствует, то соответствующая ссылка имеет нулевое значение. Таким образом при задании дерева в памяти компьютера, имеется ровно $n-1$ ссылка на реально существующие вершины (количество ссылок равно числу дуг дерева) и $n+1$ ссылка будет иметь нулевое значение. Действительно, так как каждая вершина дерева имеет две ссылки (на левого и правого сына), то

$$2 \cdot n - (n-1) = n+1$$

ссылка будет содержать поле `None` (пустой указатель).

Таким образом количество полей связи, имеющих значение `None`, всегда больше количества связей, указывающих на реальные вершины дерева. Если размер информационного поля вершины сопоставим с размером указателя, то такой способ хранения оказывается неэффективным с точки зрения использования памяти.

Ещё одним недостатком при работе с бинарными поисковыми деревьями является использование системного стека (стека) при рекурсивной (не рекурсивной) реализации обходов. При этом пространство стека, которое использует алгоритм обхода, пропорционально высоте дерева ($h = \mathrm{O}(n)$, если дерево не является сбалансированным и $h = \mathrm{O}(\log n)$ для сбалансированных деревьев).

Джеймс Хирам Моррис (1941, США)

В 1979 году Джеймс Х. Моррис предложил одно из решений проблем, связанных с использованием памяти, которое называется *прошивка дерева*.

Прошитые деревья – один из способов использования памяти, занимаемой в дереве пустыми указателями. Прошивка дерева позволяет выполнять обход дерева без использования рекурсии или стека, с помощью обычного цикла (как связный список), что значительно ускоряет процесс обхода деревьев с большим числом узлов.

Предположим, что выполняется внутренний левый обход дерева. Обход задаёт упорядочение вершин. Для примера рассмотрим внутренний левый обход дерева, приведённого на рисунке 1. 40. В процессе внутреннего левого обхода:

(1, 2, 3, 10, 20, 24, 25, 30)

у каждой вершины есть предыдущая по порядку обхода вершина (если такая существует) и следующая по порядку обхода вершина (если такая существует). Например, для вершины с ключом 10 предыдущей в порядке обхода является вершина с ключом 3, а следующей по порядку обхода является вершина с

ключом 20. Для вершины с ключом 1 предыдущей по порядку обхода вершины не существует, а следующая по порядку обхода – вершина с ключом 2. Для вершины с ключом 30 предыдущей по порядку обхода вершина – вершина с ключом 25, а последующей по порядку обхода вершины не существует.

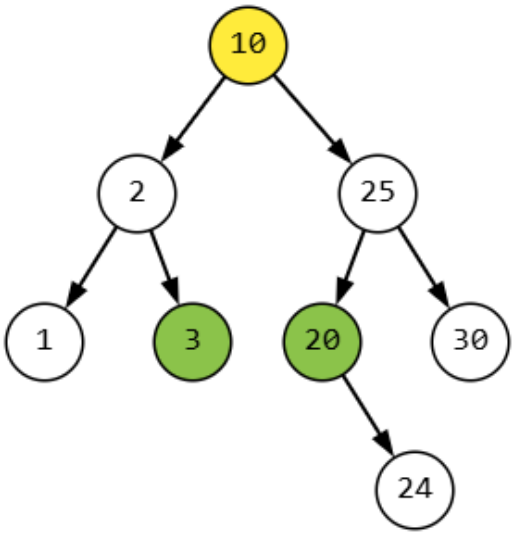


Рисунок 1.40 – Предыдущая и последующая вершины обхода.

Двоичное дерево прошивается добавлением *нитей*:

– если у некоторой вершины *v* дерева левый указатель нулевой, то этому указателю присваивается ссылка на вершину, которая по порядку обхода предшествует вершине *v*;

– если у некоторой вершины *v* дерева правый указатель нулевой, то этому указателю присваивается ссылка на вершину, которая по порядку обхода следующая за вершиной *v*.

Если дерево прошивается добавлением *нити* либо на предыдущую по порядку вершину, либо на последующую, то такое прошивание называют *одиночное прошивание*.

Если дерево прошивается добавлением *нити* как на предыдущую по порядку обхода вершину, так и на последующую, то такое прошивание называют *двойное прошивание*.

На рисунке 1.41 продемонстрировано двойное прошивание нитями бинарного поискового дерева, приведённого на рисунке 1.40, выполненное для внутреннего левого обхода. Нити на рисунке 1.41 выделены красной заливкой.

Для того, чтобы значения самой левой и самой правой *нити* были определены, вводят головную вершину `head` (на рисунке 1.41 головная вершина выделена синей заливкой). У головной вершины `head` её правая ссылка указывает на саму эту вершину, а её левая ссылка указывает на корень дерева. Теперь левая ссылка для первого элемента в порядке обхода (на рисунке 1.41. это левая ссылка для вершины с ключом 1) и правая ссылка для последнего элемента в порядке обхода (на рисунке 1.41. это правая ссылка для вершины с ключом 30), указывают на головную вершину.

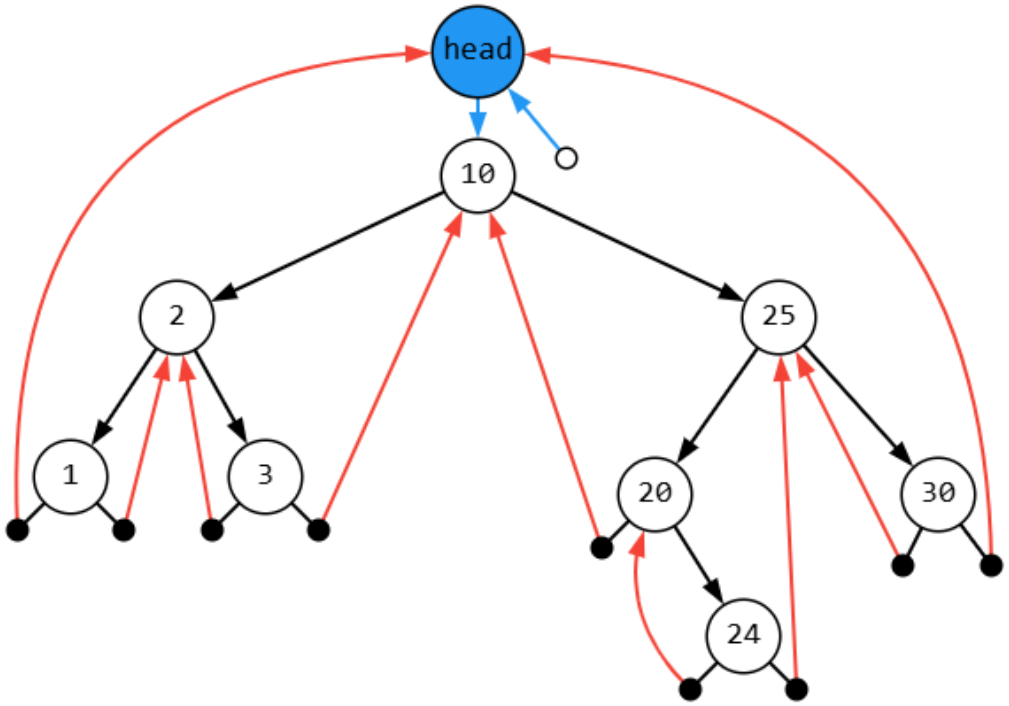


Рисунок 1.41 – Прошитое для внутреннего левого обхода дерево.

Для того, чтобы отличать обычную связь от *нити*, для каждой вершины вводят два дополнительных битовых поля `ltag` и `rtag`. Если значение тэга равно `True`, то имеем обычную связь, если же значение тэга равно `False`, то имеем прошитую связь, т. е. *нить*.

Алгоритм
двойного прошивания дерева
для внутреннего обхода

1. Для прошивки дерева сначала строится бинарное поисковое дерево (не прошитое), в котором левая и правая ссылки вершины указывают на левое и правое поддерево либо равны `None`.

Поля `ltag` и `rtag` пока остаются неопределёнными.

2. На корень дерева указывает переменная `root`.

3. На текущую вершину указывает переменная `p`.

4. Создаётся фиктивная головная вершина `head`, левая ссылка которой указывает на корень дерева, а правая – на саму головную вершину (поля тегов и информационное поле головной вершины можно оставить неопределёнными).

5. Поддерживается дополнительный глобальный указатель `p_old`, который указывает для текущей вершины указатель на предшествующую ей вершину в порядке обхода (указатель `p_old` устанавливается на головную вершину, а в качестве текущей вершины выступает корень дерева):

```
p_old = head
p = head.left
```

6. Прошивка *левых связей*: выполняем внутренний левый обход дерева, а при обработке текущей вершины в функции `Action(p)` выполняем непосредственную прошивку:

```
def InOrderTraversal_l(p):

    global p_old
    if p is not None:
         InOrderTraversal_l(p.left)
         if p.left is not None:
               p.ltag = True
         else:
             p.ltag = False
             p.left = p_old
        p_old = p
        InOrderTraversal_l(p.right)
```

7. Прошивка *правых связей* (снова указатель `p_old` устанавливается на головную вершину, а в качестве текущей вершины выступает корень дерева): выполняем внутренний правый обход дерева, а при обработке текущей вершины в функции `Action(p)` выполняем непосредственную прошивку, учитывая при этом, что левые связи уже прошиты и для левых поддеревьев никогда не получим указатель, равный `None`:

```
def InOrderTraversal_r(p):

    global p_old
    if p is not None:
         InOrderTraversal_r(p.right)
         if p.right is not None:
               p.rtag = True
         else:
             p.rtag = False
             p.right = p_old
        p_old = p
        if p.ltag == True:
            InOrderTraversal_r(p.left)
```

Если в последующем будет осуществляться только левый внутренний обход, то достаточно прошить только правые связи.

Если в последующем будет осуществляться только правый внутренний обход, то достаточно прошить только левые связи.

Алгоритм
для внутреннего левого обхода
прошитого дерева

Пусть `head` – указатель на головную вершину прошитого дерева, а `p` – указатель на текущую вершину.

Алгоритм заканчивает свою работу, когда `p` станет равно `head`.

1. Переходим к корню дерева:

```
p = head.left.
```

2. **Пока** `p.ltag == True` **повторять**

```
p = p.left
```

(дошли по левой ветке до самой левой вершины дерева).

3. Обрабатываем узел `p`

(например, печатаем его информационную часть `p.key`).

4. Если

```
p.rtag == False
```

(правого поддерева у текущей вершины нет, поэтому будет осуществлён переход по *нити* прошитого дерева к вершине, которая в порядке обхода следует за текущей вершиной),

то

```
p = p.right
```

и возвращаемся **к шагу 3** алгоритма.

Иначе (`p.rtag == True`)

(переходим в правое поддерево текущей вершины)

```
p = p.right
```

и возвращаемся **к шагу 2** алгоритма.

Следует отметить, что необходимо выполнять прошивку дерева заново, если происходит модификация дерева (добавление, удаление элемента), а для каждого обхода (прямого, обратного, внутреннего) выполняется своя прошивка.

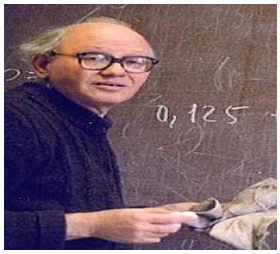
Поэтому прошитые деревья целесообразно использовать в тех задачах, где изменения в деревьях происходят редко, а обход выполняется часто.

В Приложении 1 приведён листинг кода основных процедур работы с прошитыми деревьями.

1.2. Организация поиска. Сбалансированные поисковые деревья

1.2.1. АВЛ-дерево

В 1962 году советские учёные Г. М. **А**дельсон-**В**ельский и Е. М. **Л**андис предложили структуру данных сбалансированного поискового дерева. **АВЛ**-дерево – аббревиатура, образованная первыми буквами фамилий создателей.



Г.М. Адельсон-Вельский

Е.М. Ландис

1.2.1.1. Основные определения.

Определение 1.15. АВЛ-дерево это бинарное поисковое дерево, которое является сбалансированным по высоте.

Таким образом, если для каждой вершины бинарного поискового дерева высоты её поддеревьев отличаются не более, чем на 1, то такое дерево будет АВЛ-деревом.

На рисунке 1.42 приведён пример АВЛ-дерева: бинарное дерево является поисковым так как для каждой вершины *v* ключи вершин в её левом поддереве меньше ключа вершины *v*, а в правом – больше, а также для каждой вершины дерева высоты её поддеревьев отличаются не более, чем на 1:

– у вершины с ключом 40 высота её левого поддерева равна 1, а высота её правого поддерева равна 0;

– у вершины с ключом 80 высота её левого поддерева равна 1, а высота её правого поддерева равна 0;

– у вершины с ключом 71 высота её левого поддерева равна –1 (для простоты вычислений высоту несуществующего поддрева полагают равной минус единице), а высота её правого поддерева равна 0;

– у всех остальных вершин дерева высоты их поддеревьев равны.

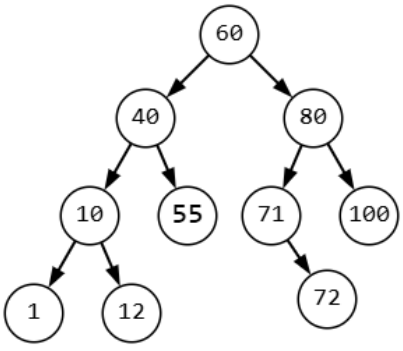


Рисунок 1.42 – АВЛ-дерево.

На рисунке 1.43 приведены примеры деревьев, которые не являются АВЛ.

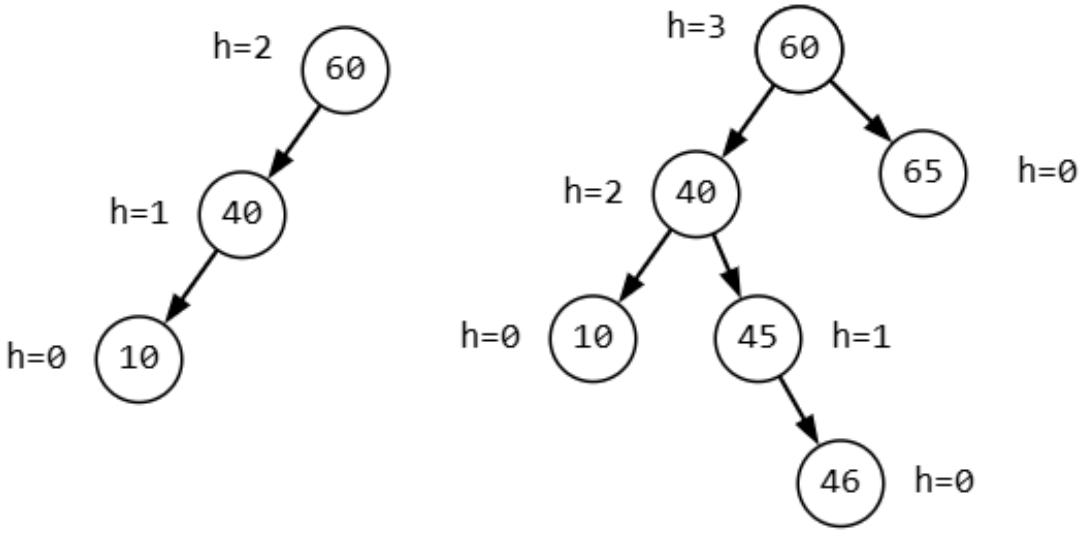


Рисунок 1.43 – Деревья, которые не являются АВЛ.

Для бинарного поискового дерева, приведённого на рисунке 1.43 слева, для вершины с ключом 60 высота её правого поддерева равна -1 (минус единица), а высота её левого поддерева равна $1\left(\left|-1-1\right|=2\right)$, нарушается свойство сбалансированности по высотам $\left(2>1\right)$, дерево не является АВЛ-деревом.

Для бинарного поискового дерева, приведённого на рисунке 1.43 справа, для вершины с ключом 60 высота её правого поддерева равна 0, а высота её левого поддерева равна 2 $\left(\left|0-2\right|=2\right)$, нарушается свойство сбалансированности по высотам $\left(2>1\right)$, дерево не является АВЛ-деревом.

Теорема 1.1. Пусть n – число внутренних вершин АВЛ-дерева, h – его высота. Тогда справедливы следующие неравенства:

$$\log_2\left(n+1\right)\le h<1,4404\cdot\log_2\left(n+2\right)-0,328.$$

Доказательство.

Для доказательства утверждения оценивают максимальное и минимальное число внутренних вершин дерева (внутренние вершины дерева – все вершины дерева, за исключением листьев; если дерево состоит из единственной вершины, то полагаем число внутренних вершин равным 0).

Максимальное число внутренних вершин АВЛ-дерева получается достаточно просто. Подсчитаем число внутренних вершин у бинарного дерева высоты h, у которого полностью заполнены вершинами все уровни:

$$n_h^{\max}=2^0+2^1+2^2+\cdots+2^{h-1}=\frac{2^h-1}{2-1}=2^h-1.$$

Получаем оценку снизу для высоты АВЛ-дерева:

$$n \le n_h^{\max} = 2^h - 1,$$

$$n \le 2^h - 1,$$

$$h \ge \log_2(n+1).$$

Для оценки минимального числа внутренних вершин АВЛ-дерева используются свойства чисел Фибоначчи.

Пусть N_h – число внутренних вершин АВЛ-дерева высоты h с минимальным числом внутренних вершин (обозначим такое дерево через T_h). Деревья T_0, T_1 определяются однозначно (рисунок 1.44). Предположим, что мы умеем строить АВЛ-деревья высоты $< h$ с минимальным числом внутренних вершин. Если мы хотим построить АВЛ-дерево T_h, то у корня этого дерева одно из поддеревьев должно иметь высоту $(h-1)$ (в качестве такого поддерева возьмём АВЛ-дерево высоты $(h-1)$ с минимальным числом внутренних вершин), а второе поддерево должно иметь высоту $(h-1)$ или $(h-2)$ (так как дерево T_h должно быть сбалансированным). Так как мы минимизируем число внутренних вершин, то в качестве второго поддерева возьмём АВЛ-дерево высоты $(h-2)$ с минимальным числом внутренних вершин. Поскольку принцип построения деревьев напоминает построение чисел Фибоначчи, то такие деревья обычно называют *деревьями Фибоначчи* (рисунок 1.44).

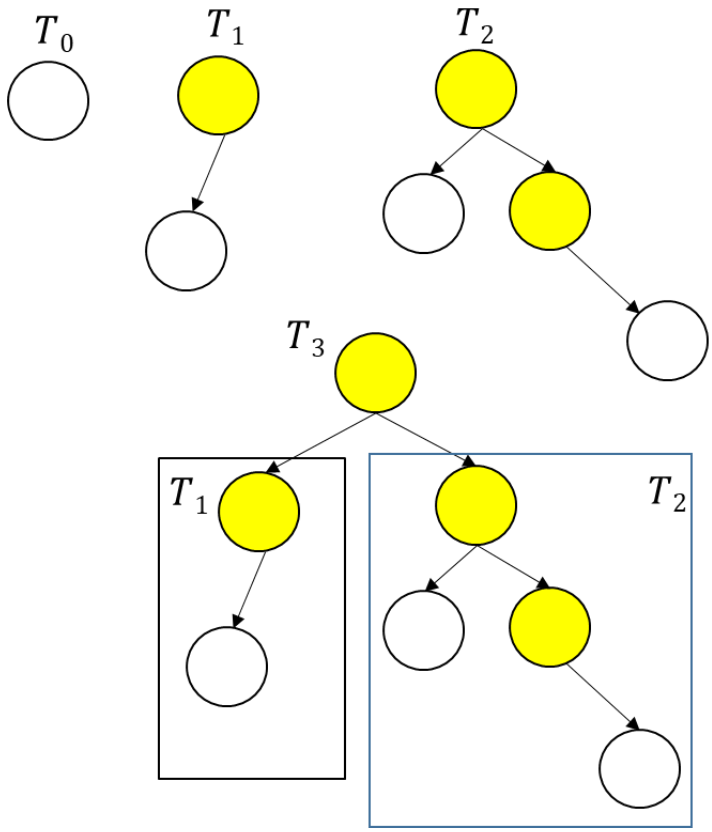


Рисунок 1.44 – АВЛ-деревья с минимальным числом внутренних вершин.

Справедливы следующие равенства:

$$N_{h+1} = N_h + N_{h-1} + 1,$$

$$(N_{h+1} + 1) = (N_h + 1) + (N_{h-1} + 1).$$

Выполняя замену переменной, $F'_i = N_i + 1$, получим следующее равенство: $F'_{h+1} = F'_h + F'_{h-1}$. Какая связь между F'_h и числом Фибоначчи F_h?

Сравним значения N_h, F'_h и F_h.

	N_h – число внутренних вершин АВЛ-дерева высоты h	$F'_h = N_h + 1$	F_h – h-е число Фибоначчи
$h = 0$	0	1	0
$h = 1$	1	2	1
$h = 2$	2	3	1
$h = 3$	4	5	2
$h = 4$	7	8	3
$h = 5$	12	13	5
$h = 6$	…	…	8
$h = 7$	…	…	13
…	…	…	…

Несложно увидеть, что

$$F_{(h+2)} = F'_h = N_h + 1,$$

откуда получаем, что

$$N_h = F_{(h+2)} - 1.$$

Учитывая формулу общего члена последовательности Фибоначчи

$$F_h = \frac{\Phi^h - \Phi^h}{\sqrt[2]{5}}, \text{ где } \Phi = \frac{1+\sqrt[2]{5}}{2}, \Phi = \frac{1-\sqrt[2]{5}}{2},$$

(число $\Phi = \frac{1+\sqrt[2]{5}}{2}$, которое фигурирует в формуле, – «золотое сечение») и тот факт, что

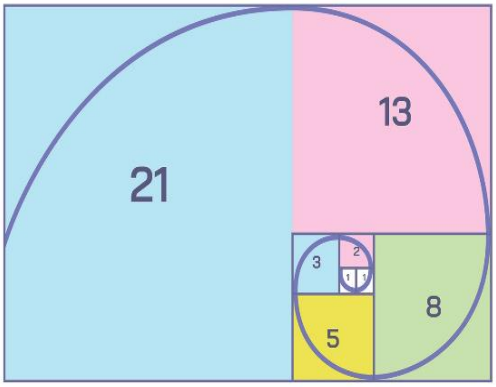
$$\left|\frac{1-\sqrt[2]{5}}{2}\right| < 1,$$

получим следующие неравенства

$$n \geq N_h = \frac{\Phi^{(h+2)} - \Phi^{(h+2)}}{\sqrt[2]{5}} - 1 \geq \frac{\left(\frac{1+\sqrt[2]{5}}{2}\right)^{(h+2)}}{\sqrt[2]{5}} - 2.$$

Логарифмируя полученное неравенство, получим для высоты АВЛ-дерева оценку сверху: $h < 1{,}4404 \cdot \log_2(n+2) - 0{,}328$. Доказательство теоремы завершено.

Из истории математики



Золотое сечение (*золотая пропорциям*) – пропорция, которую заметили еще древние египтяне. Чтобы её получить, нужно разделить линию на две части так, чтобы длинная (a) относилась с короткой (b) в такой же пропорции, как линия ($a+b$) соотносится с длинной (a). Эта пропорция всегда равняется 1,618 (это число обозначают прописной греческой буквой Ф («фи»), в честь древнегреческого скульптора и архитектора Фидия

$$\frac{a}{b}=\frac{a+b}{a}=1,6180339887...=...=\Phi.$$

Если взять длину линии равной 1, то $a+b=1$, $\frac{a}{b}=\frac{1}{a}$, откуда, решая квадратное уравнение $x=1+\frac{1}{x}\Leftrightarrow x^2-x-1=0$, где $x=\frac{a}{b}$, получим и число $\Phi=\frac{1+\sqrt[2]{5}}{2}$.

Для последовательности чисел Фибоначчи отношение последующего члена последовательности к предыдущему в пределе стремится к «золотому сечению» (числу Φ).

1.2.1.2. Базовые операции и их вычислительная сложность

Операции поиска, добавления и удаления элементов для АВЛ-дерева осуществляются точно также, как и для бинарных поисковых деревьев.

1) Для поиска элемента с ключом x спуск начинается от корня дерева. Пусть v – текущая вершина. Если ключ x меньше ключа текущей вершины, то продолжаем поиск в левом поддереве вершины v, если x больше, то продолжаем поиск в правом поддереве вершины v. Движение осуществляем до тех пор, пока не найдём элемент x или не придём к нулевому (отсутствующему поддереву).

Вычислительная сложность алгоритма поиска элемента с ключом x в АВЛ-дереве – $O(\log n)$.

2) Для добавления элемента с ключом x выполняется спуск по дереву в поисках подходящего места для новой вершины. Спуск начинается с корня. Если новый ключ x меньше ключа текущей вершины, то спускаемся влево, если x больше, то вправо. В случае если x равен текущему ключу, то ничего делать не нужно, вставка не требуется.

Вычислительная сложность данного этапа добавления элемента с ключом x в АВЛ-дерево – $O(\log n)$.

Однако, после добавления элемента может нарушится свойство сбалансированности по высотам (при этом оно может нарушиться сразу у нескольких вершин) и его нужно восстановить (в последующем мы покажем, что для восстановления свойства сбалансированности достаточно будет сбалансировать только одну вершину и у всех вершин свойство сбалансированности по высотам будет выполнено).

На рисунке 1.45 в АВЛ-дерево была добавлена вершина с ключом 6, в результате вершины с ключами 2 и 4 стали разбалансированны (у вершины с ключом 2 высота левого поддерева равна 0, а высота правого поддерева равна 2; у вершины с ключом 4 левого поддерева нет (высота такого поддерева полагается равной минус 1), а высота правого поддерева равна 1).

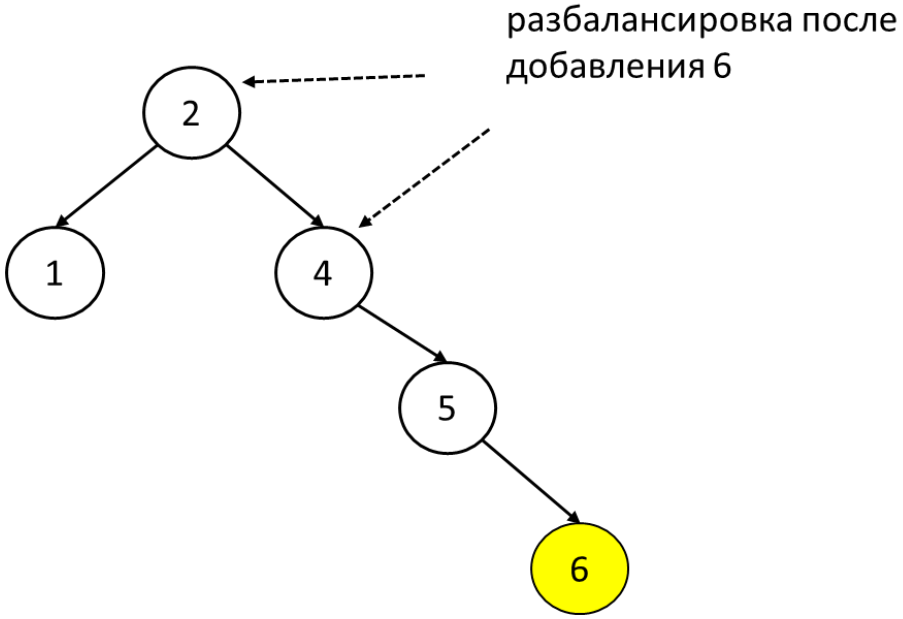


Рисунок 1.45 – Разбалансировка после добавления вершины с ключом 6.

3) При удалении вершины v из АВЛ-дерева, как и при удалении из бинарного поискового дерева, возможны три случая: 1) вершина v является листом, 2) у вершины v есть только одно левое/правое поддерево, 3) у вершины v есть оба поддерева. Удаляем вершину. Вычислительная сложность этого этапа удаления – $O(\log n)$.

После выполнения операции удаления элемента может нарушится свойство сбалансированности по высотам (только у одной вершины) и его нужно восстановить. Однако после балансировки могут потребоваться повторные балансировки. Так как каждый раз разбалансированная вершина будет находиться на меньшей глубине, то количество повторных балансировок ограничено высотой дерева, т. е. $O(\log n)$.

На рисунке 1.46 из АВЛ-дерева была удалена вершина с ключом 3. В результате для вершины с ключом 4 нарушено свойство сбалансированности по высотам (высота её левого поддерева равна -1 (минус единица), а высота правого поддерева равна 1).

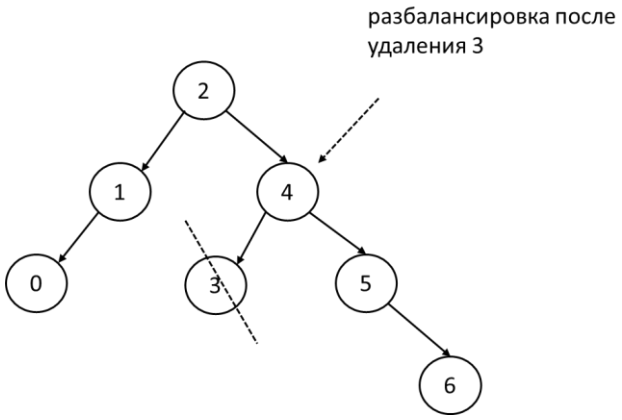


Рисунок 1.46 – Разбалансировка после удаления вершины.

Восстановление свойств сбалансированности выполняют каждый раз, как только происходит разбалансировка.

1.2.1.3. Восстановление сбалансированности. Повороты

Для восстановления свойств сбалансированности выполняют один из четырёх поворотов:

1) **LL**-поворот (малое правое вращение, одинарный правый поворот);
2) **RR**-поворот (малое левое вращение, одинарный левый поворот);
3) **LR**-поворот (большое правое вращение, двойной правый поворот);
4) **RL**-поворот (большое левое вращение, двойной левый поворот).

Каждый из четырёх поворотов выполняется за константное время, поэтому вычислительная сложность базовых операций добавления/удаления элемента для АВЛ-дерева выполняется за время $O(\log n)$.

На рисунке 1.47 рассмотрен случай, когда в дерево *A* была добавлена новая вершина, что привело к увеличению высоты дерева *A* на единицу (предположим, что для всех вершин дерева *A* свойство сбалансированности по-прежнему выполнено). В результате добавления новой вершины высота вершины k_2 увеличилась на единицу, а это, в свою очередь, привело к разбалансировке вершины k_1 (высота её левого поддерева равна $(h-1)$, а высота её правого поддерева равна $(h-3)$). Могла произойти разбалансировка и у других вершин дерева, которые лежат на пути от корня дерева к вершине k_2.

Покажем, что достаточно сбалансировать только ту из разбалансированных вершин, которая наиболее удалена от корня дерева, в результате чего свойство сбалансированности будет выполнено для всех вершин дерева.

На рисунке 1.47 продемонстрировано выполнение балансировки для вершины k_1, где k_1 – вершина на максимальной глубине, для которой произошла разбалансировка. После балансировки высота вершины, которая является сыном вершины *z* (после балансировки сыном *z* вместо k_1 стала вершина k_2), снова равна $(h-1)$, как и до разбалансировки. Следовательно, после поворота свойство сбалансированности у всех вершин дерева выполняется.

LL-поворот

Пусть k_1 – вершина на максимальной глубине, для которой произошла разбалансировка и высота её левого поддерева больше высоты правого поддерева на 2. Пусть k_2 – левый сын вершины k_1 и высота его левого поддерева (A) больше (или равна) высоты его правого поддерева (B).

На рисунке 1.47 продемонстрирован LL-поворот для восстановления свойства сбалансированности. Поворот выполняется за время $O(1)$.

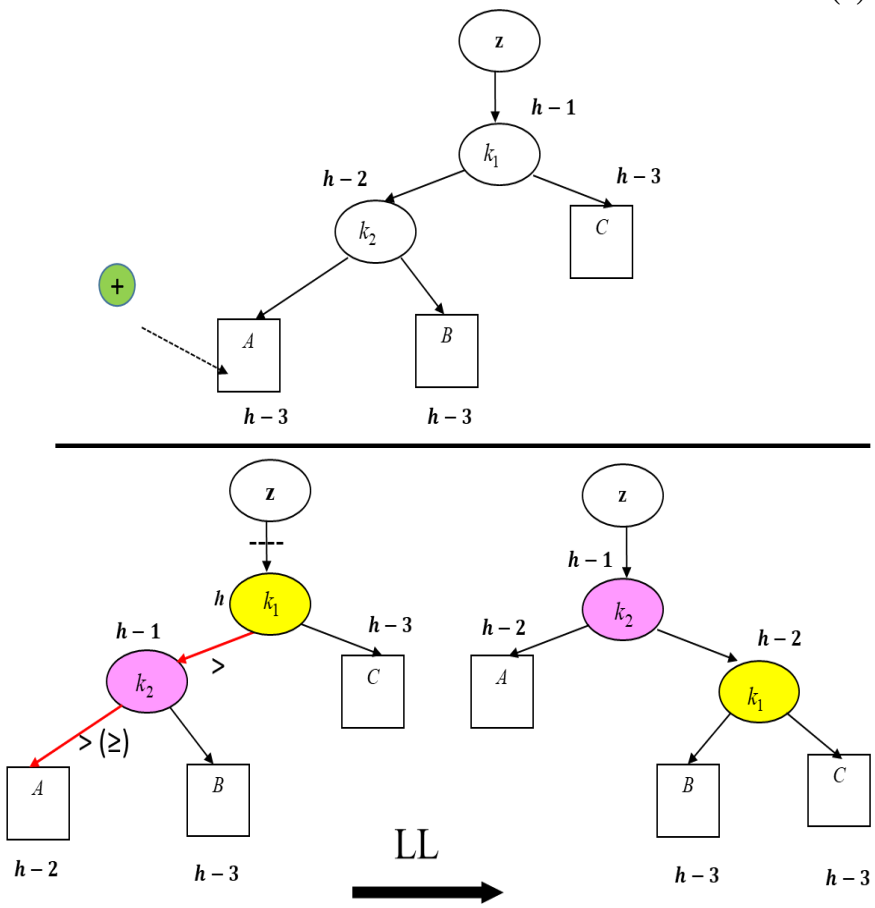


Рисунок 1.47 – LL-поворот.

На рисунке 1.48 выполнен LL-поворот для разбалансированной вершины с ключом 70 (высоты её поддеревьев отличаются на 2).

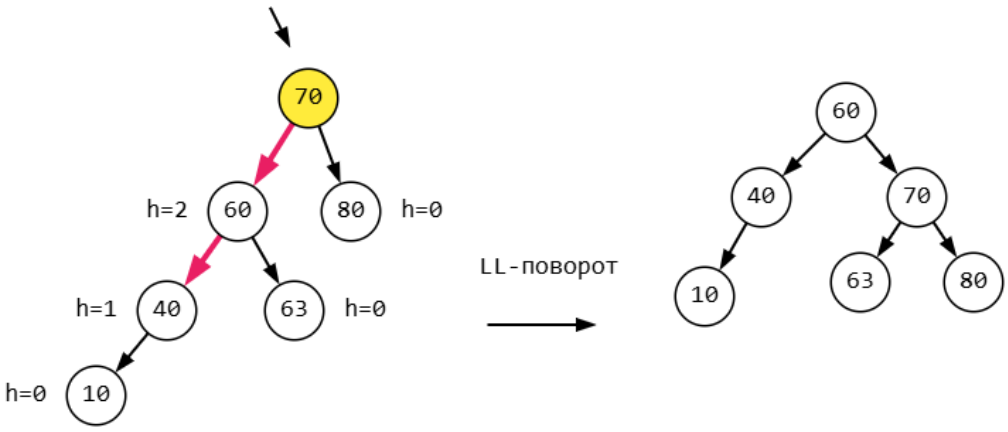


Рисунок 1.48– LL-поворот вершины с ключом 70.

RR-поворот

Пусть k_1 – вершина на максимальной глубине, для которой произошла разбалансировка и высота её правого поддерева больше высоты левого поддерева на 2.

Пусть k_2 – правый сын вершины k_1 и высота его правого поддерева (C) больше (или равна) высоты его левого поддерева (B).

На рисунке 1.49 продемонстрирован RR-поворот для восстановления свойства сбалансированности. Поворот выполняется за время $O(1)$.

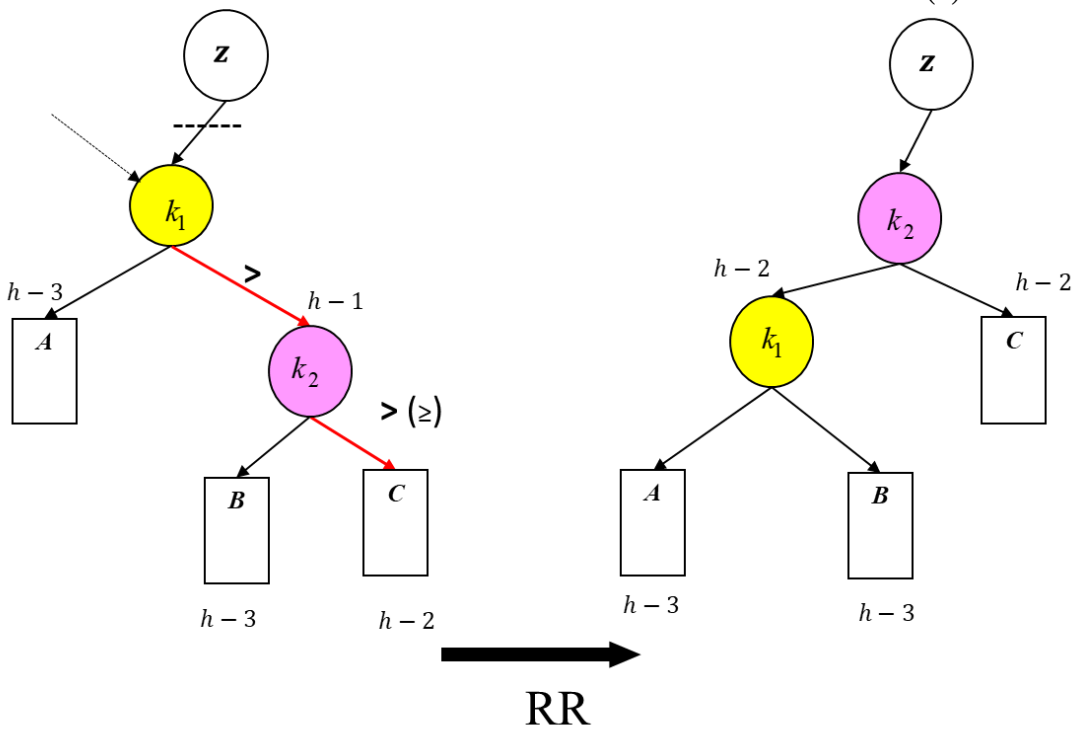


Рисунок 1.49 – RR-поворот.

На рисунке 1.50 выполнен RR-поворот для единственной разбалансированной вершины с ключом 60 (высоты её поддеревьев отличаются на 2).

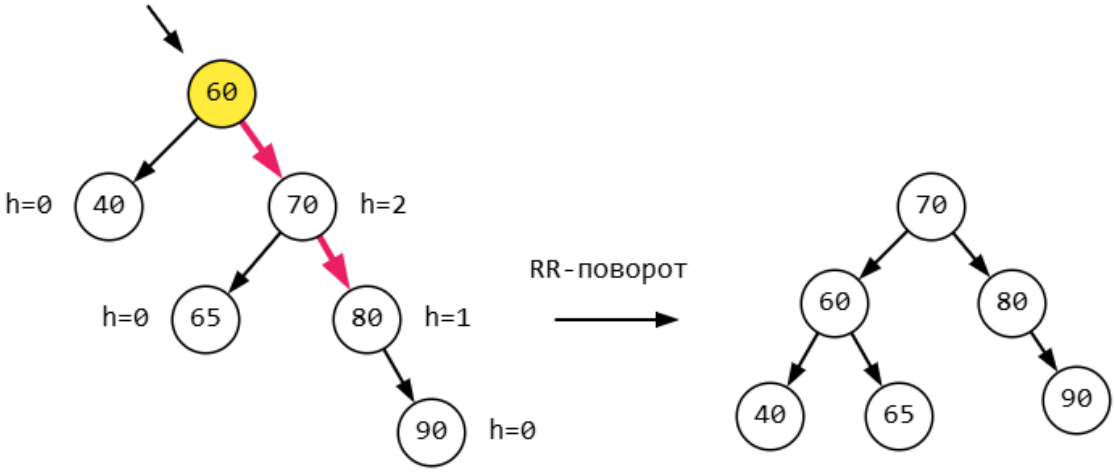


Рисунок 1.50 – RR-поворот вершины с ключом 60.

RL-поворот

Пусть k_1 – вершина на максимальной глубине, для которой произошла разбалансировка и высота её правого поддерева больше высоты левого поддерева на 2.

Пусть k_2 – правый сын вершины k_1 и высота его левого поддерева (поддерево с корнем в вершине k_3) больше высоты его правого поддерева (D).

На рисунке 1.51 продемонстрирован RL-поворот для восстановления свойства сбалансированности. Поворот выполняется за время $O(1)$.

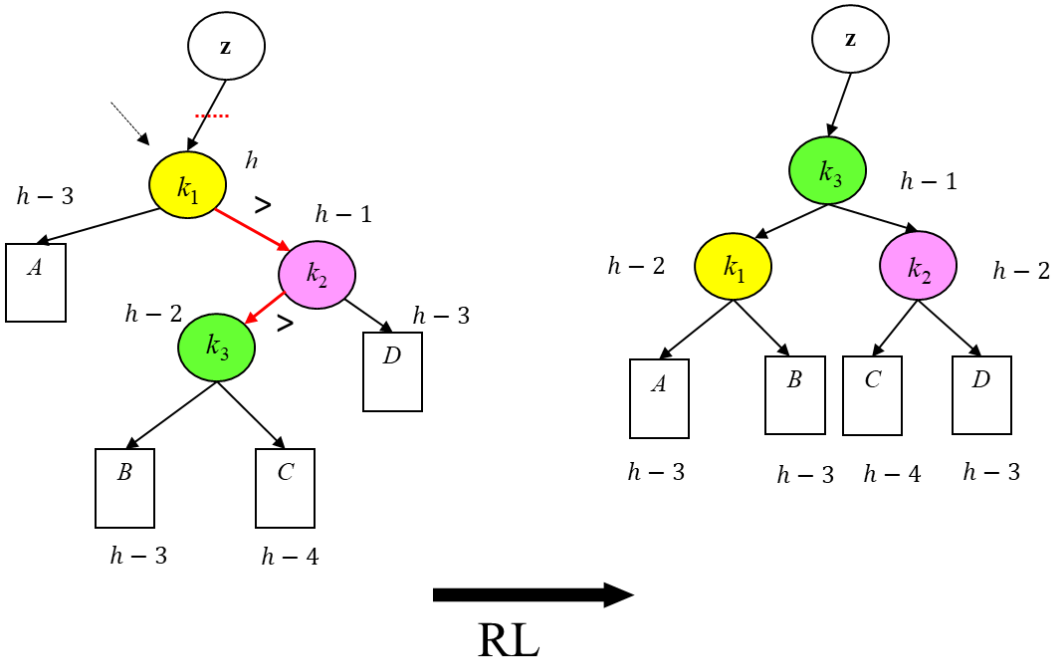


Рисунок 1.51 – RL-поворот.

На рисунке 1.52 выполнен RL-поворот для единственной разбалансированной вершины с ключом 60 (высоты её поддеревьев отличаются на 2).

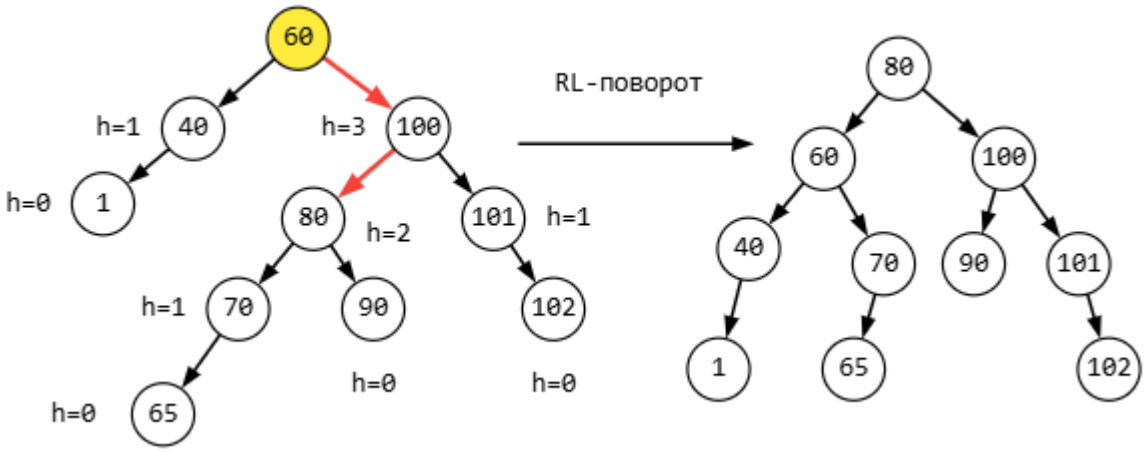


Рисунок 1.52 – RL-поворот вершины с ключом 60.

LR-поворот

Пусть k_1 – вершина на максимальной глубине, для которой произошла разбалансировка и высота её левого поддерева больше высоты правого поддерева на 2.

Пусть k_2 – левый сын вершины k_1 и высота его правого поддерева (поддерево с корнем в вершине k_3) больше высоты его левого поддерева (A).

На рисунке 1.53 продемонстрирован LR-поворот для восстановления свойства сбалансированности. Поворот выполняется за время $O(1)$.

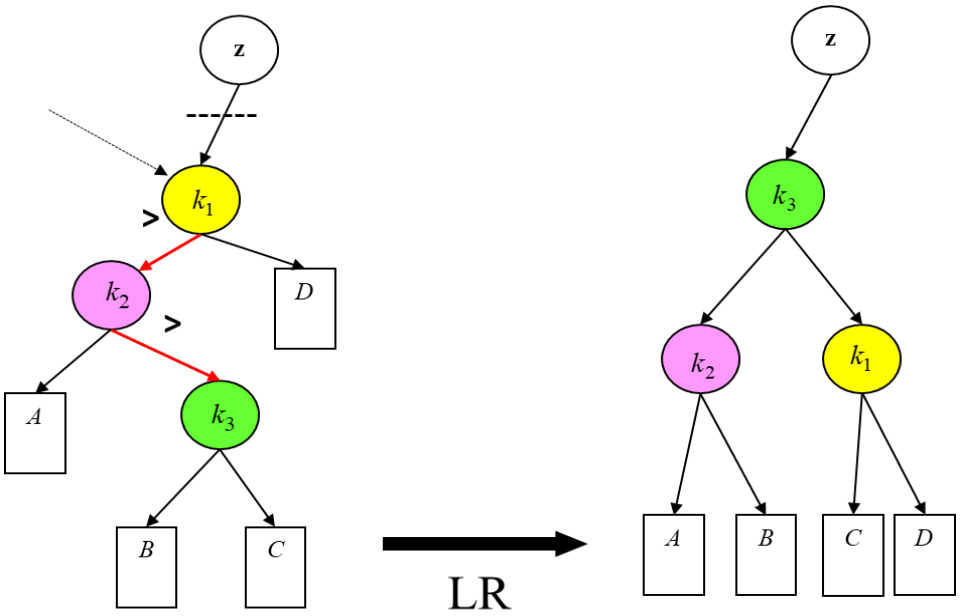


Рисунок 1.53 – LR-поворот.

На рисунке 1.54 выполнен LR-поворот для единственной разбалансированной вершины с ключом 8 (высоты её поддеревьев отличаются на 2).

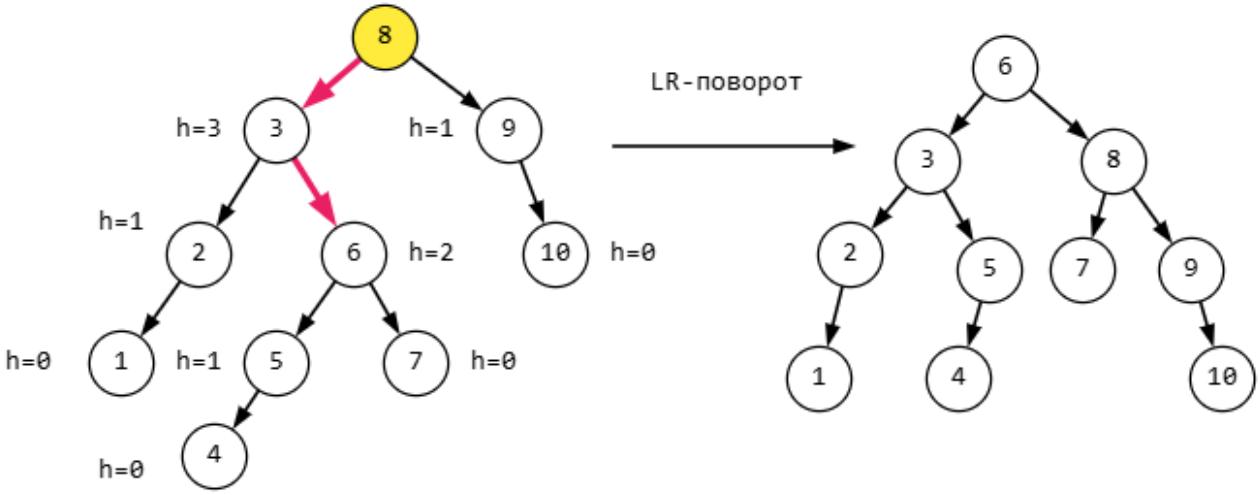


Рисунок 1.54 – LR-поворот вершины с ключом 60.

Таким образом для определения буквенного названия поворота, который выполняется для балансировки, поступаем следующим образом.

1. Берём среди разбалансированных вершин ту, которая находится на наибольшей глубине (предположим, что это вершина k_1) и определяем, какое по высоте поддерево у неё больше:

– если левое поддерево больше по высоте, то пишем в названии поворота первую букву **L** (англ. **L**eft);

– если правое поддерево больше по высоте, то пишем в названии поворота первую букву **R** (англ. **R**ight).

2. Спускаемся из разбалансированной вершины (k_1) в её поддерево с большей высотой. Предположим, что из k_1 спустились в вершину k_2. Определяем, какое у вершины k_2 поддерево больше по высоте:

– если левое поддерево больше по высоте, то пишем в названии поворота вторую букву **L** (англ. **L**eft);

– если правое поддерево больше по высоте, то пишем в названии поворота вторую букву **R** (англ. **R**ight);

– если у вершины k_2 высоты подеревьев равны (эта ситуация возможна при выполнении операции удаления), то пишем вторую букву в названии поворота такую же, как и первую (получаем название поворота **LL** или **RR**).

1.2.1.4. Пример построения АВЛ-дерева

Пример 1.9. Построить АВЛ-дерево для последовательности чисел:

$$7, 8, 2, 3, 4, 6, 1, 9, 10, 11, 5.$$

Решение. На рисунке 1.55 продемонстрированы шаги, выполняемые при последовательном добавлении вершин с ключами 7, 8, 2 и 3 в АВЛ-дерево. При добавлении вершин сохранялось свойство сбалансированности по высотам и балансировки не выполнялись.

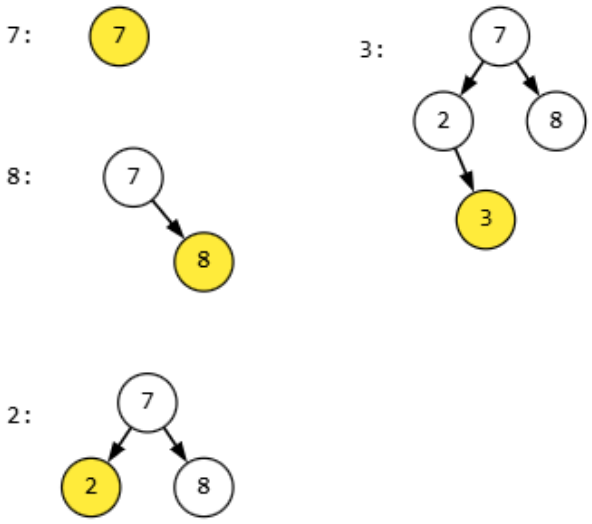


Рисунок 1.55 – Последовательное добавление вершин с ключами 7, 8, 2, 3.

При добавлении вершины с ключом 4 (рисунок 1.56) для двух вершин с ключами 2 и 7 произоизошла разбалансировка.

Среди разбалансированных вершин выбираем ту, которая имеет наибольшую глубину (наиболее удалена от корня дерева). В примере будет выбрана вершина с ключом 2 (у вершины с ключом 2 глубина равна 1, а у вершины с ключом 7 глубина равна 0).

Для вершины с ключом 2 выполняем RR-поворот.

На рисунке 1.56 справа показано АВЛ-дерево после выполнения поворота.

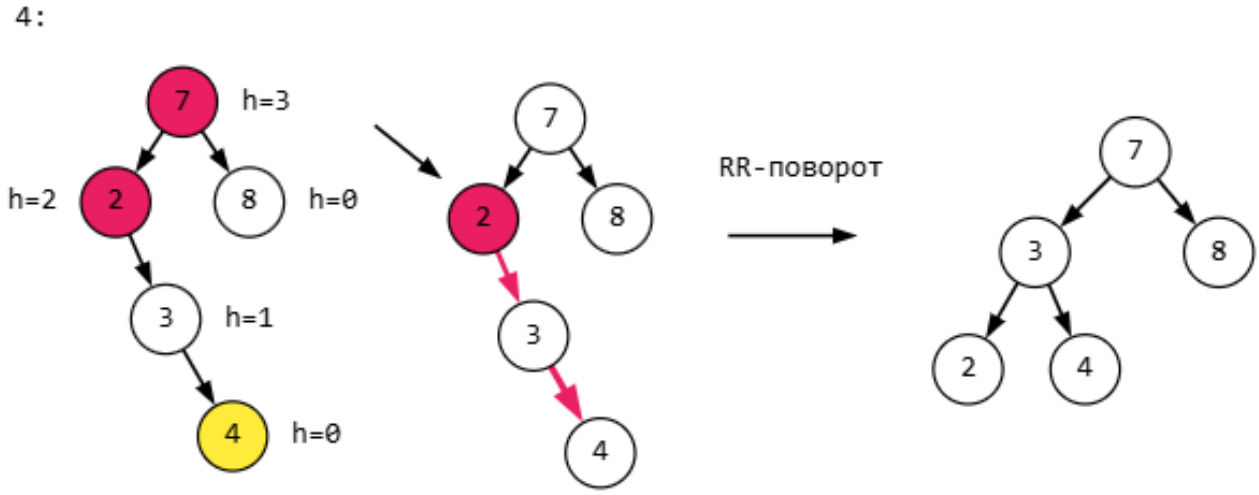


Рисунок 1.56 – Добавление вершины с ключом 4.

При добавлении вершины с ключом 6 (рисунок 1.57) у вершины с ключом 7 произойдёт разбалансировка.

Для разбалансированной вершины с ключом 7 выполняем LR-поворот.

На рисунке 1.57 показано АВЛ-дерево после выполнения поворота.

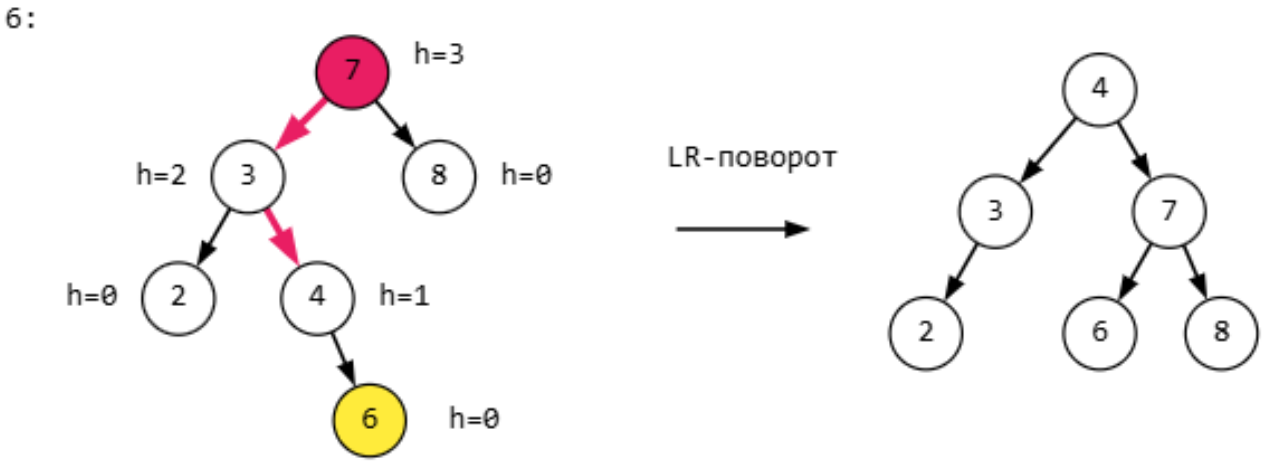


Рисунок 1.57 – Добавление вершины с ключом 6.

При добавлении вершины с ключом 1 (рисунок 1.58) только у одной вершины с ключом 3 произойдёт разбалансировка.

Для разбалансированной вершины с ключом 3 выполняем LL-поворот. На рисунке 1.58 показано АВЛ-дерево после выполнения поворота.

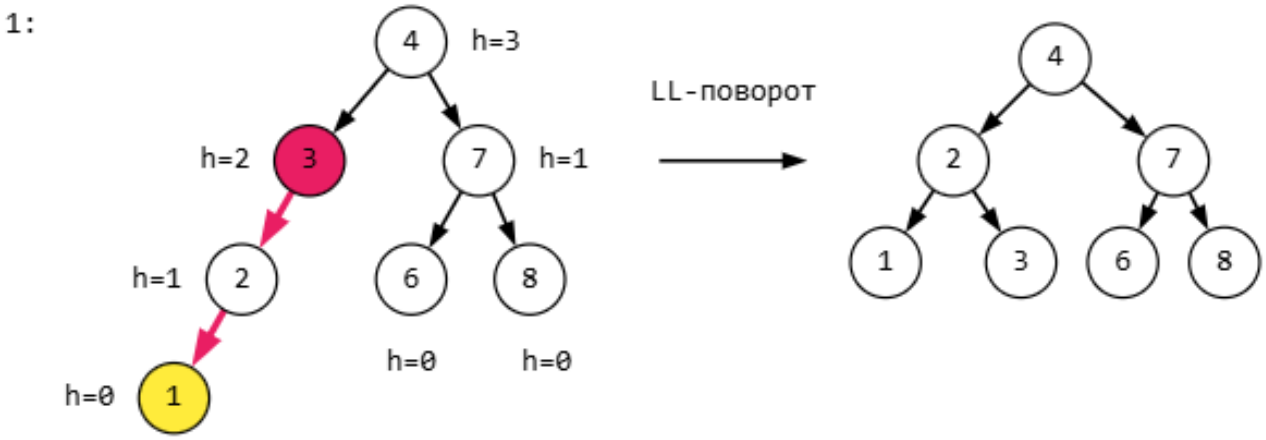


Рисунок 1.58 – Добавление вершины с ключом 1.

При добавлении вершины с ключом 9 разбалансировки нет, а после добавления вершины с ключом 10 разбалансировка произойдёт сразу у нескольких вершин: 4, 7 и 8. Выбираем среди разбалансированных вершин ту, которая имеет наибольшую глубину, это – вершина с ключом 8 (рисунок 1.59). Для разбалансированной вершины с ключом 8 выполняем RR-поворот.

На рисунке 1.59 показано АВЛ-дерево после выполнения поворота.

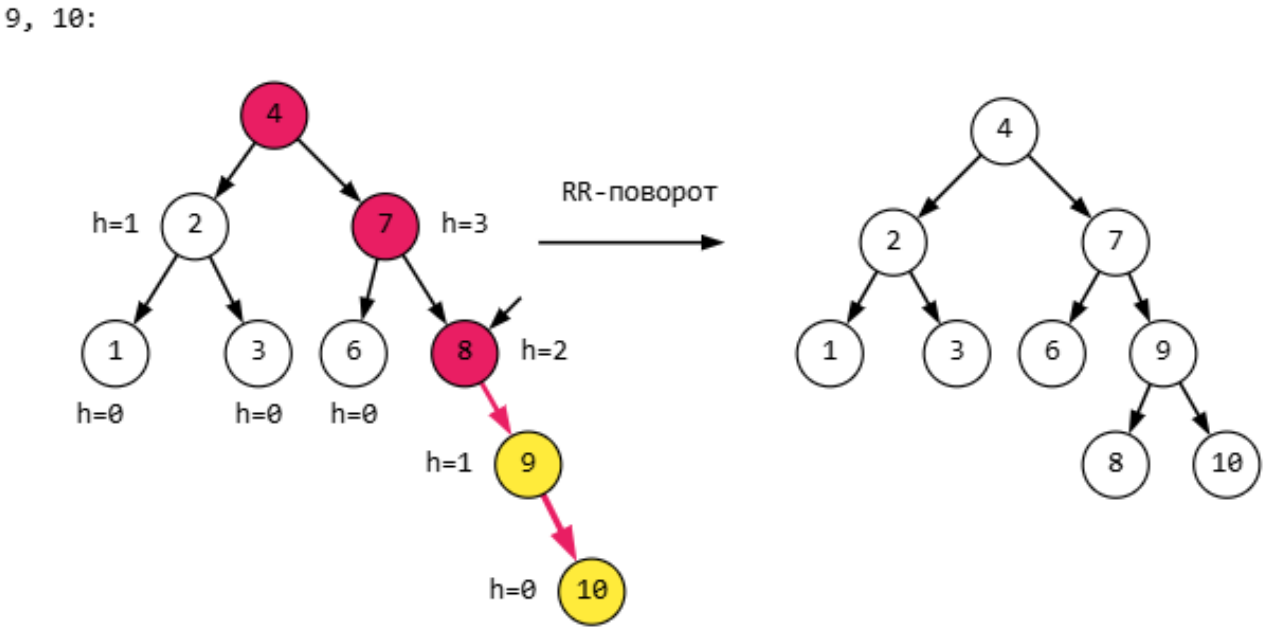


Рисунок 1.59 – Добавление вершин с ключами 9 и 10.

При добавлении вершины с ключом 11 (рисунок 1.60) у двух вершин с ключами 4 и 7 произойдёт разбалансировка. Среди разбалансированных вершин выбираем ту, которая имеет наибольшую глубину (наиболее удалена от корня дерева). В нашем примере это вершина с ключом 7 (у вершины с ключом 4 глубина равна 0, а у вершины с ключом 7 глубина равна 1). Для разбалансированной вершины с ключом 7 выполняем RR-поворот. На рисунке 1.60 показано АВЛ-дерево после выполнения поворота.

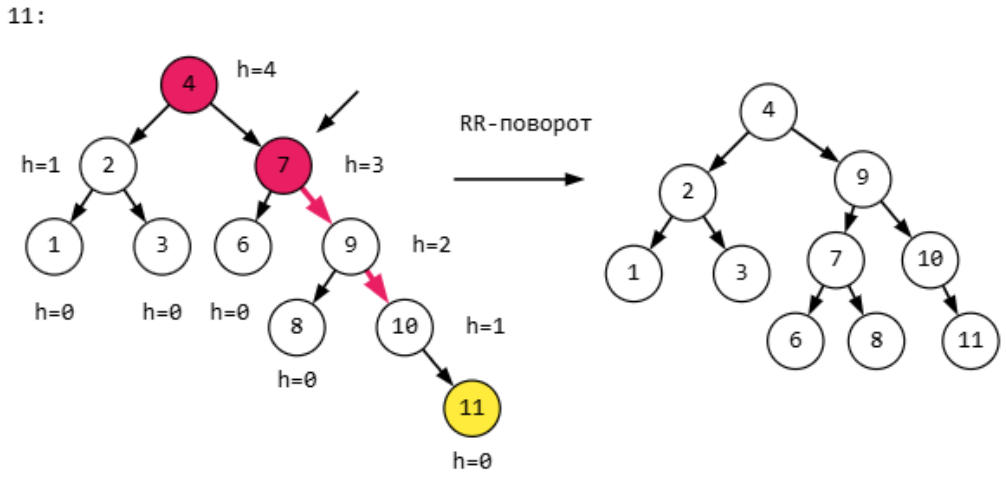


Рисунок 1.60 – Добавление вершины с ключом 11.

При добавлении вершины с ключом 5 произойдёт разбалансировка только у одной вершины с ключом 4 (рисунок 1.61). Для разбалансированной вершины с ключом 4 выполняем RL-поворот. На рисунке 1.61 показано АВЛ-дерево после выполнения поворота.

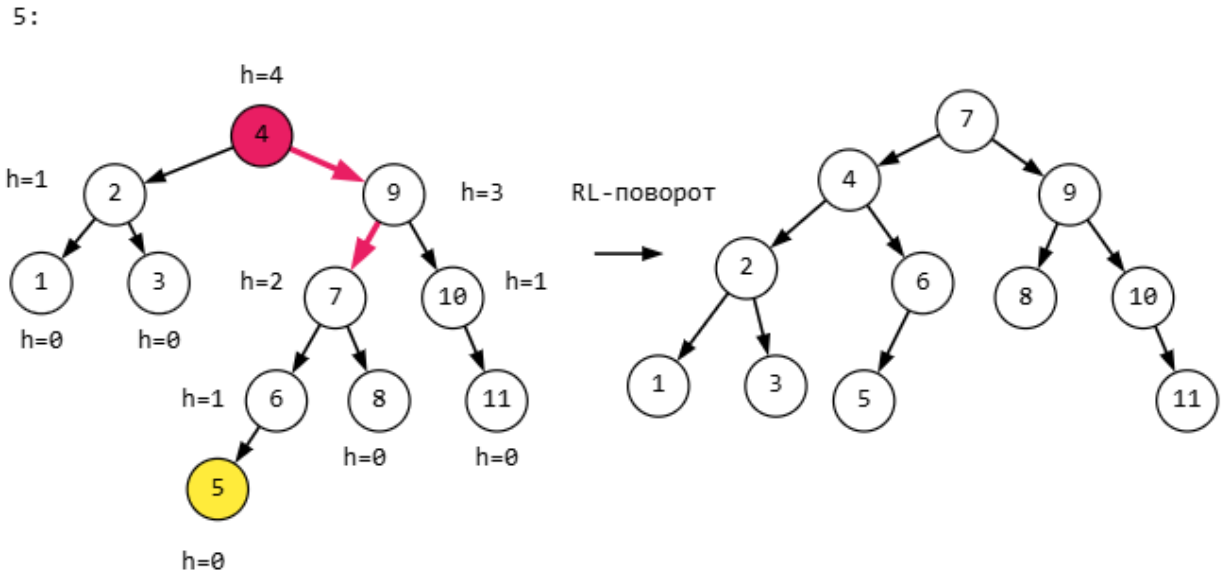
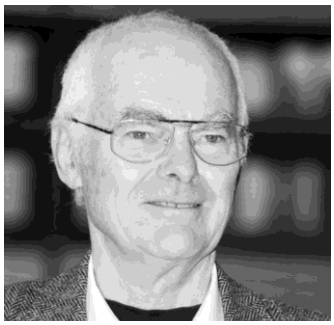


Рисунок 1.61 – Добавление вершины с ключом 5.

Замечание 1.2. Балансировку (один из четырёх поворотов: LL, RR, RL, LR) необходимо выполнять каждый раз, как только происходит разбалансировка, при этом балансировка выполняется только для одной разбалансированной вершины, которая находится на наибольшей глубине (т. е. наиболее удалена от корня дерева).

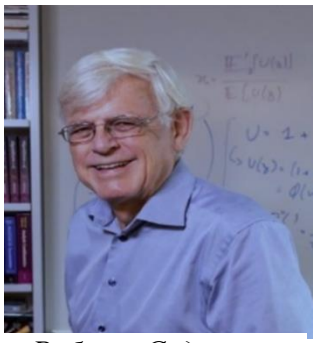
1.2.2. Красно-чёрное дерево

Изобретателем красно-чёрного дерева (RB-дерева) считают немецкого учёного Р. Байера (Германия, Мюнхенский технический университет, научная сфера – информатика, изобретатель RB-дерева, UB-дерева, B-дерева).



Рудольф Байер

Леонидас Джон Гибас



Роберт Седжвик

В 1978 году в статье Л. Гибаса и Р. Седжвика структура данных получила название «красно-чёрное дерево». По словам Л. Гибаса, они использовали ручки двух цветов, а по словам Р. Седжвика, красный и чёрный цвет лучше всех смотрелись на лазерном принтере Xerox.

Красно-чёрное дерево является примером бинарного поискового дерева, для которого специальные операции балансировки гарантируют, что высота дерева не превзойдёт $O(\log n)$, при этом время выполнения процедур балансировки (перекрашивания вершин, вращения) также ограничено этой величиной.

1.2.2.1. Основные определения

Определение 1.16. Красно-чёрное дерево– это бинарное поисковое дерево, для которого выполняются RB-свойства:

1) каждая вершина является либо красной, либо чёрной;

2) каждый лист – чёрный;

3) у красной вершины оба сына – чёрные (нет двух подряд идущих красных вершин);

4) любой путь из корня в листья содержит одинаковое количество чёрных вершин;

5) корень – чёрный (это требование не обязательно, так как корень всегда можно перекрасить в чёрный цвет и это не нарушит ни одно из RB-свойств).

Если к каждой вершине, которая не имеет сыновей, добавить фиктивный чёрный лист, то мы получим, что каждая внутренняя вершина дерева содержит ключевое значения, а все листья – фиктивные. На рисунке 1.62 приведён пример красно-чёрного дерева.

Не сложно видеть, что красно-чёрное дерево может не являться сбалансированным по высоте (красно-чёрное дерево, приведённое на рисунке 1.62, не является сбалансированным по высоте, например, для вершины с ключом 19 высота её правого поддерева равна 1, а левого – 3).

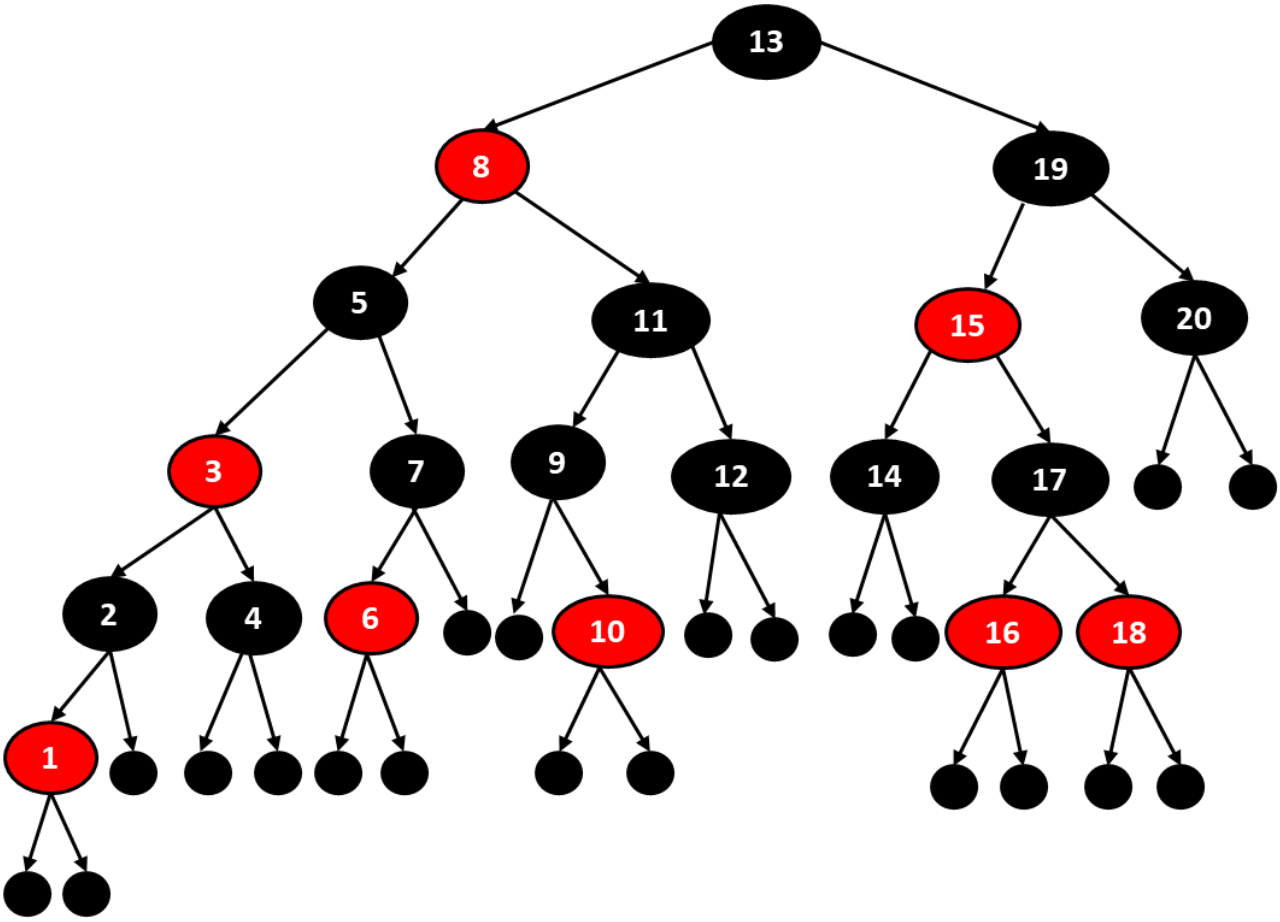


Рисунок 1.62 – Красно-чёрное дерево.

Для каждой вершины v определим её «чёрную высоту» $bh(v)$ (англ. black-height).

Определение 1.17. Чёрная высота вершины v – количество чёрных вершин (без учёта самой вершины v) на пути из вершины v в лист.

Чёрную высоту дерева будем полагать равной чёрной высоте корня дерева (чёрная высота дерева, приведённого на рисунке 1.62, равна 3).

Нетрудно доказать, что для каждой вершины v любой путь из вершины v к листу содержит одинаковое количество чёрных вершин (в основе доказательства лежит тот факт, что путь в дереве в любую вершину единственный, и для RB-дерева выполняется свойство 4).

Для любой вершины v красно-чёрного дерева справедливы следующие неравенства:

$$bh(v) \le LengthPath(v, leaf) \le 2 \cdot bh(v),$$

$$bh(v) \le h(v) \le 2 \cdot bh(v),$$

где $LengthPath(v, leaf)$ – длина некоторого пути от вершины v до листа $leaf$, $h(v)$ – высота вершины v, $bh(v)$ – чёрная высота вершины v. Действительно, самый короткий путь от вершины v до листа $leaf$ состоит только из чёрных вершин, а в самом длинном пути после каждой чёрной вершины (кроме листа) следует красная вершина.

Теорема 1.2. Красно-чёрное дерево, содержащее n внутренних вершин, имеет высоту

$$h(T) \le 2 \cdot \log(n+1).$$

Доказательство.

1) Сначала, используя метод математической индукции (по высоте h), докажем, что поддерево с корнем в вершине v содержит как минимум $\left(2^{bh(v)} - 1\right)$ внутренних вершин.

База индукции. Если вершина v – лист, то

$$bh(v) = 0 \text{ и } 2^{bh(v)} - 1 = 2^0 - 1 = 0.$$

Шаг индукции. Предположим, что для всех вершин дерева, для которых высота меньше или равна h, свойство выполняется. Рассмотрим вершину v, у которой высота равна $h+1 > 0$. Вершина v – внутренняя вершина дерева, значит у неё есть оба поддерева. Если у вершины v чёрная высота равна $bh(v)$, то её сыновья могут иметь чёрную высоту либо $\left(bh(v) - 1\right)$ (если сын – чёрный), либо $bh(v)$ (если сын – красный). С учётом индукционного предположения для сыновей вершины v получим, что для числа внутренних вершин поддерева с корнем в вершине v справедливы неравенства:

$$\ge 2 \cdot \left(2^{bh(v)-1} - 1\right) + 1 = 2^{bh(v)} - 1.$$

Таким образом доказано, что поддерево с корнем в вершине v содержит как минимум $\left(2^{bh(v)} - 1\right)$ внутренних вершин.

2) Теперь рассмотрим произвольный путь из корня дерева в лист. По крайней мере половина вершин на пути, не считая сам корень, должны быть чёрными.

Следовательно,

$$bh(T) \le h(T) \le 2 \cdot bh(T),$$

откуда

$$\frac{h(T)}{2} \le bh(T) \le h(T),$$

т. е. чёрная высота корня $bh(T)$ должна составлять как минимум $h(T)/2$.

Тогда, по доказанному в пункте 1) свойству, для числа n внутренних вершин дерева T справедливы неравенства:

$$n \ge 2^{bh(T)} - 1 \ge 2^{\frac{h(T)}{2}} - 1. \qquad (1.2)$$

Логарифмируем неравенство (1.2), получаем утверждение теоремы 1.2. $h(T) \le 2 \cdot \log_2(n+1)$. Доказательство теоремы завершено.

Для того, чтобы описать процедуры балансировки (перекрашивания и вращения), введём следующие обозначения (рисунок 1.63).

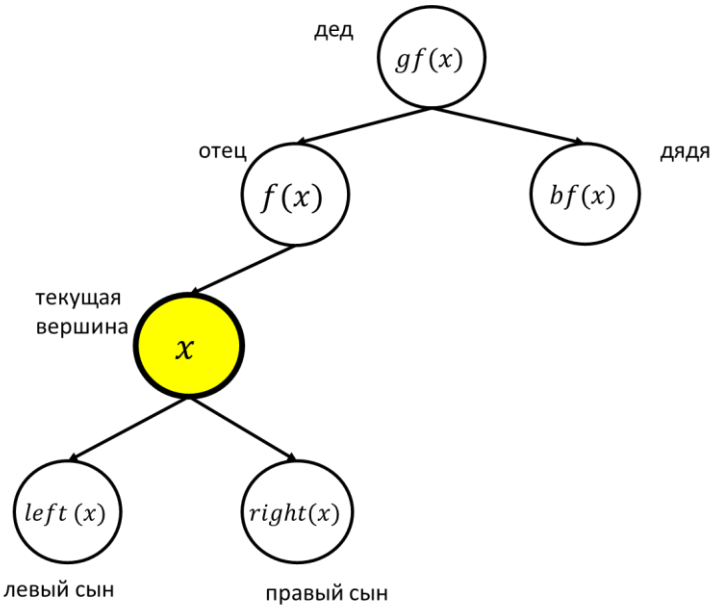


Рисунок 1.63 – Обозначения.

Для поддержки свойств красно-чёрных деревьев выполняются вращения, каждое из которых выполняется за время $O(1)$.

На рисунке 1.64 приведено левое вращение (англ. **L**eft rotate).

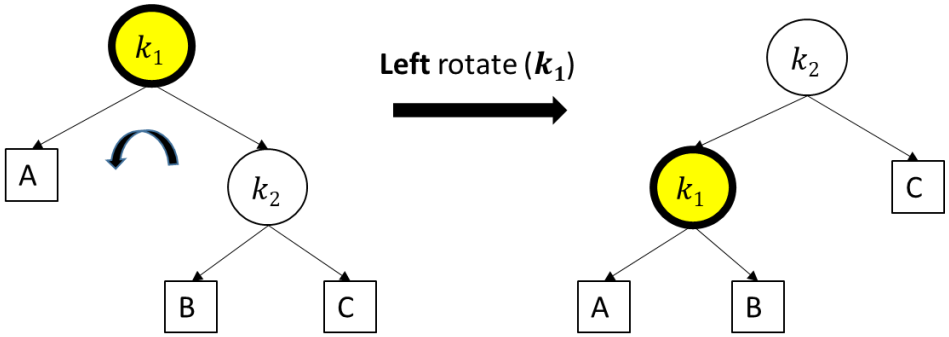


Рисунок 1.64 – Левое вращение.

На рисунке 1.65 приведено правое вращение (англ. **R**ight rotate).

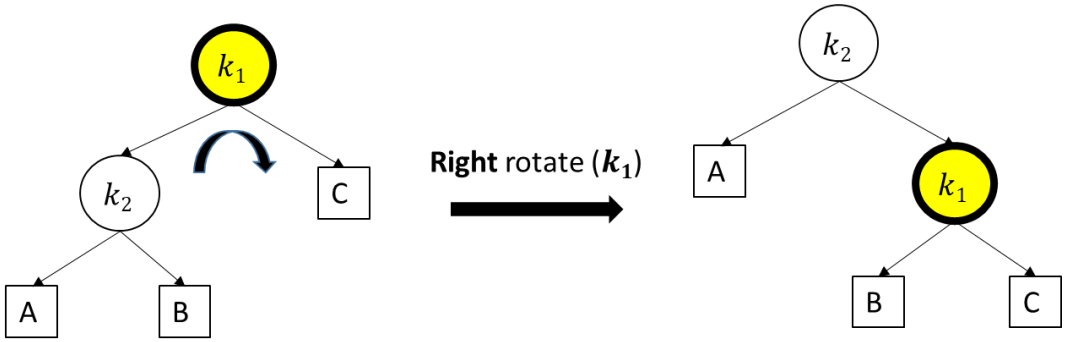


Рисунок 1.65 – Правое вращение.

1.2.2.2. Добавление ключа в дерево. Восстановление RB-свойств

Операция добавления вершины с ключом ***x*** в красно-чёрное дерево осуществляется по следующему алгоритму.

Алгоритм добавления вершины в RB-дерево

1) Осуществляем поиск места для добавляемой вершины с ключом ***x*** по аналогии с тем, как это делали для бинарного поискового дерева.

2) Добавляем вершину с ключом ***x*** на место чёрного фиктивного листа и добавляем ей в качестве сыновей две фиктивные чёрные вершины.

3) Красим вершину с ключом ***x*** в красный цвет.

4) Если после добавления произошло нарушение RB-свойства (может нарушиться только одно свойство: появились две подряд идущие красные вершины), то выполняем процедуру, восстанавливающую RB-свойства: перекраска вершин и, возможно, не более двух вращений.

На рисунке 1.66 проиллюстрированы этапы последовательного добавления вершин с ключами 1, 2 и 7 в красно-чёрное дерево.

При добавлении вершин с ключами 1 и 2 все свойства красно-чёрного дерева выполнялись.

При добавлении вершины с ключом 7 свойство 3) красно-чёрного дерева нарушено, так как две подряд идущие вершины с ключами 2 и 7 имеют красный цвет.

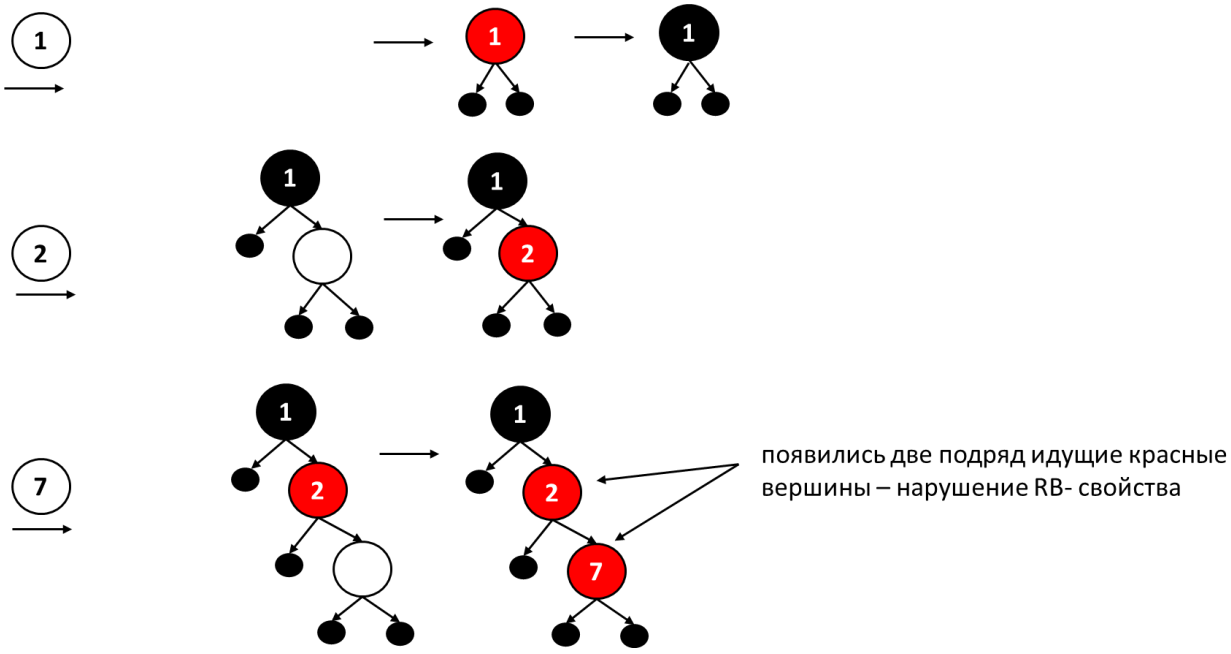


Рисунок 1.66. Добавление вершин с ключами 1,2 и 7.

Рассмотрим процедуры, восстанавливающие RB-свойства: перекраски и вращения.

Случай 1. Отец и дядя вершины ***x*** – красные.

На рисунке 1.67 приведены возможные случаи взаимного расположения: вершины ***x***, отца $f(x)$, деда $gf(x)$ и дяди $bf(x)$, а также действия, которые нужно выполнить, чтобы восстановить RB-свойства.

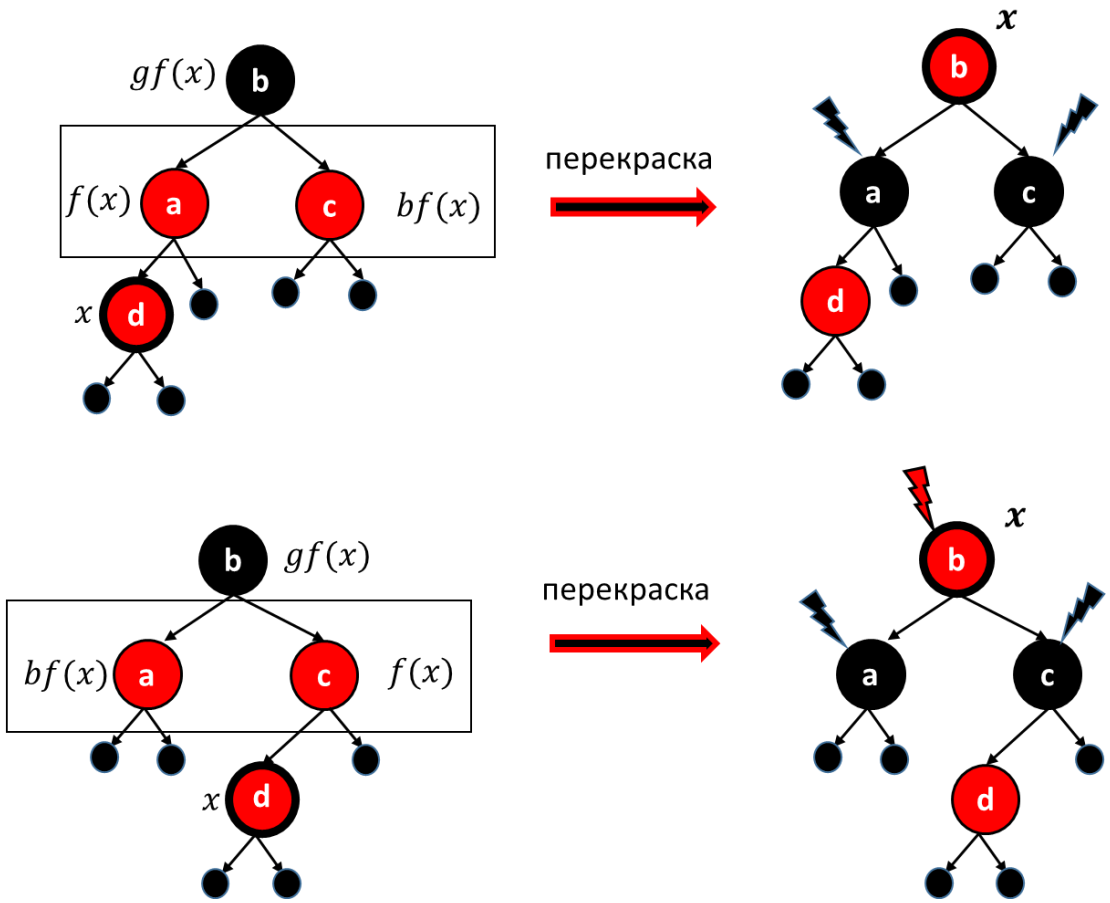


Рисунок 1.67.

Если отец у вершины ***x*** – чёрный, то операция добавления завершена (рисунок 1.68 слева). Если отец у вершины ***x*** – красный, то продолжаем рекурсивно восстанавливать RB-свойства, рассматривая в качестве вершины ***x*** вершину b (рисунок 1.68 справа).

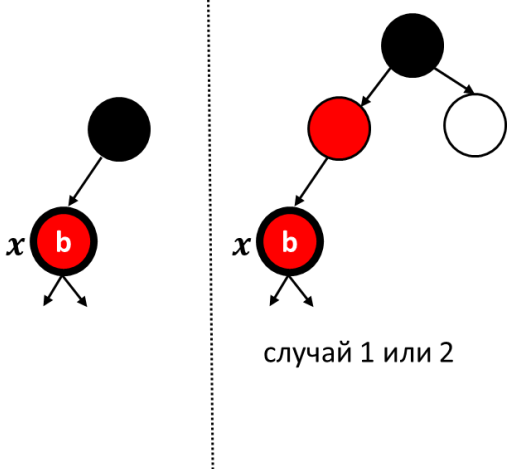


Рисунок 1.68.

Случай 2. Отец и дядя вершины x – разного цвета: отец – красный, дядя – чёрный.

На рисунке 1.69 приведены четыре случая взаимного расположения вершины **x**, отца $f(x)$, деда $gf(x)$ и дяди $bf(x)$.

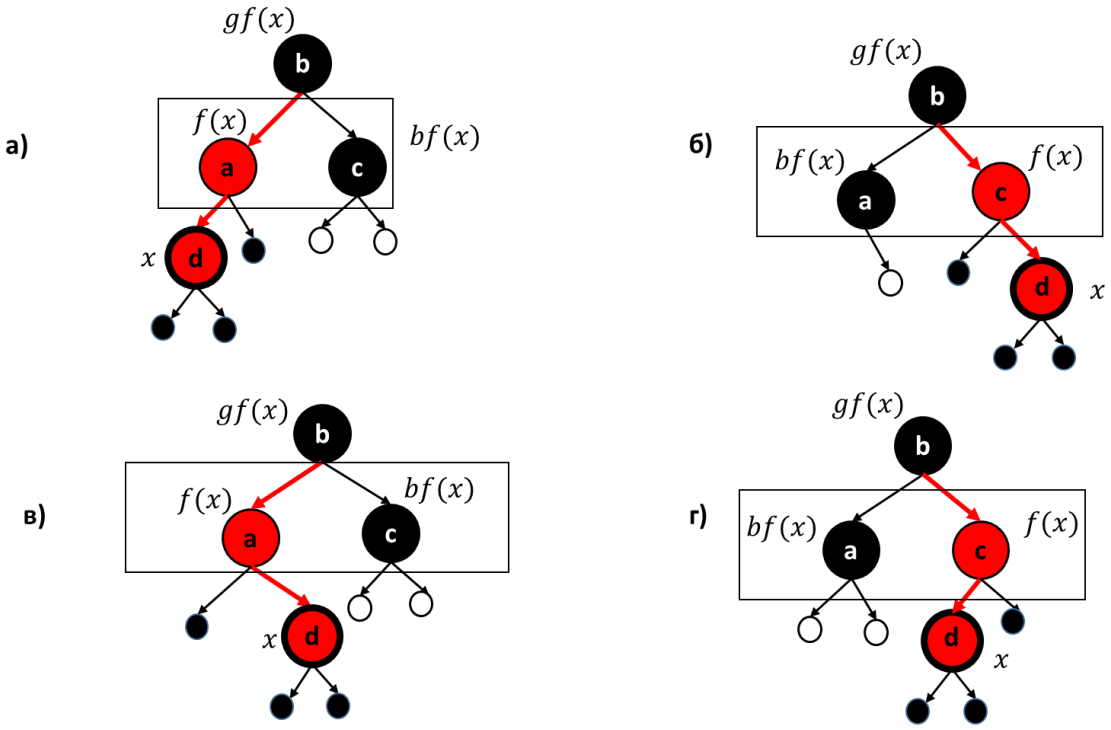


Рисунок 1.69.

Сначала рассмотрим ситуацию, когда вершина **x**, отец $f(x)$ и дед $gf(x)$ располагаются на одной линии (рисунки 1.70 и 1.71).

Для восстановления RB-свойств выполняют два перекрашивания и одно вращение (RightRotate$(gf(x))$ или LeftRotate$(gf(x))$.

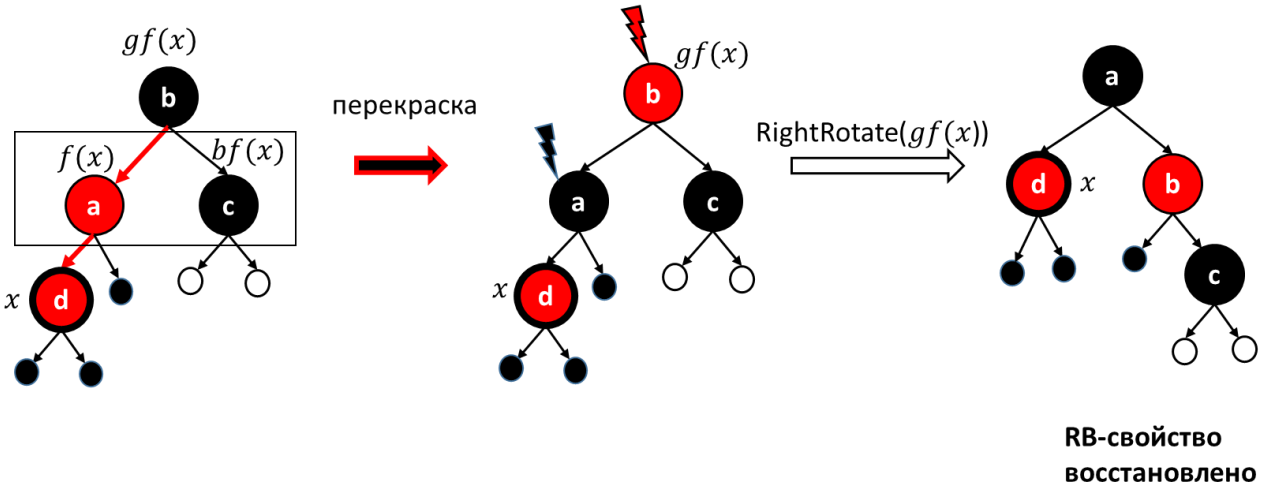


Рисунок 1.70.

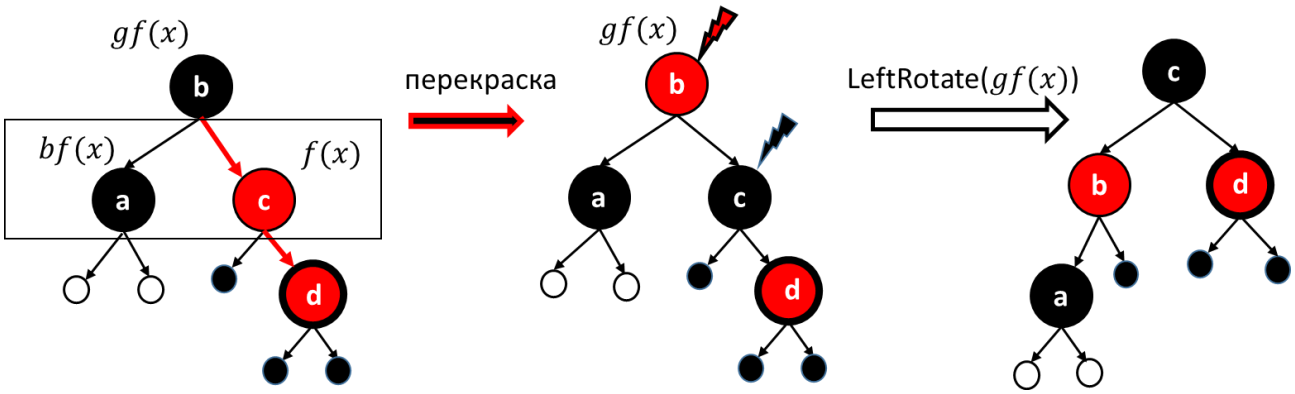


Рисунок 1.71.

Теперь рассмотрим ситуацию, когда расположение вершин **x**, отец $f(x)$ и дед $gf(x)$ образует угол (на рисунке 1.72 рассмотрен один из таких случаев, а остальные рассматриваются аналогично).

Для восстановления RB-свойства сначала вершины **x**, $f(x)$ и $gf(x)$ выстраивают левым вращением LeftRotate$(f(x))$ на одну линию, сводя этот случай к предыдущему. Затем в качестве вершины ***x*** берут вершину a и выполняют два перекрашивания и одно вращение (см. рисунок 1.72).

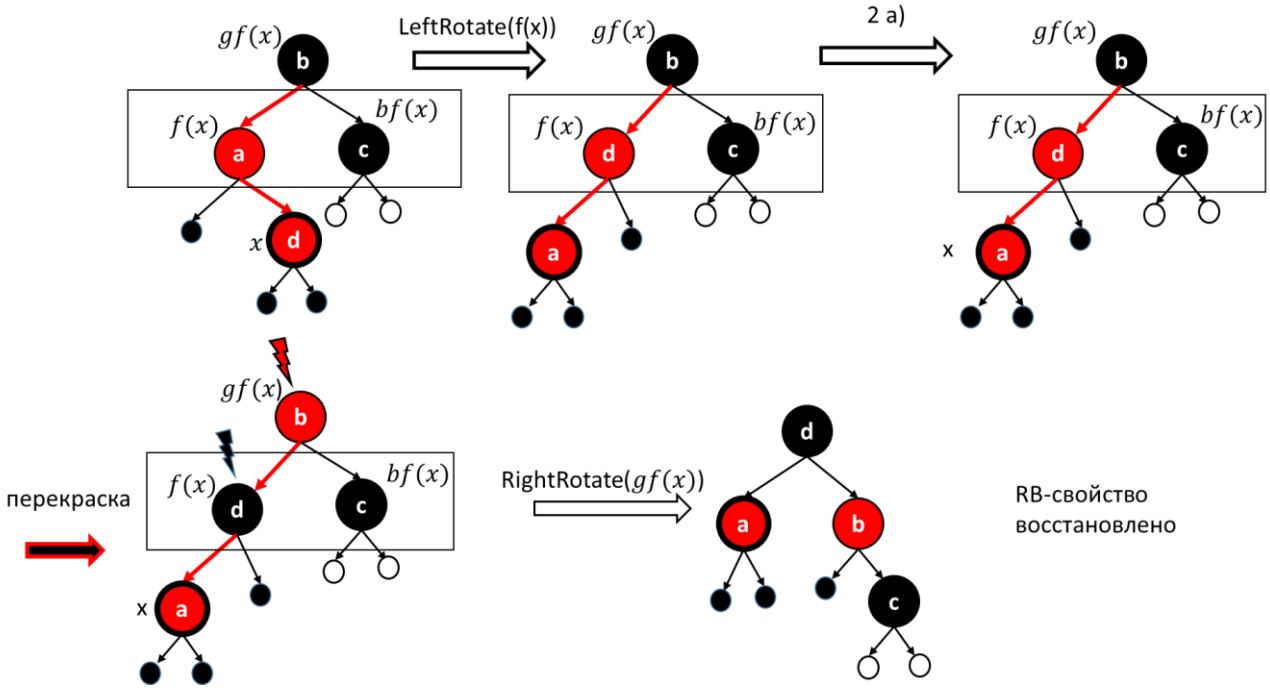


Рисунок 1.72.

После выполнения указанных действий (перекрашивания и два вращения) все RB-свойства выполнены.

Два шага, выполненные последовательно на рисунке 1.72, на рисунке 1.73 объединены в один.

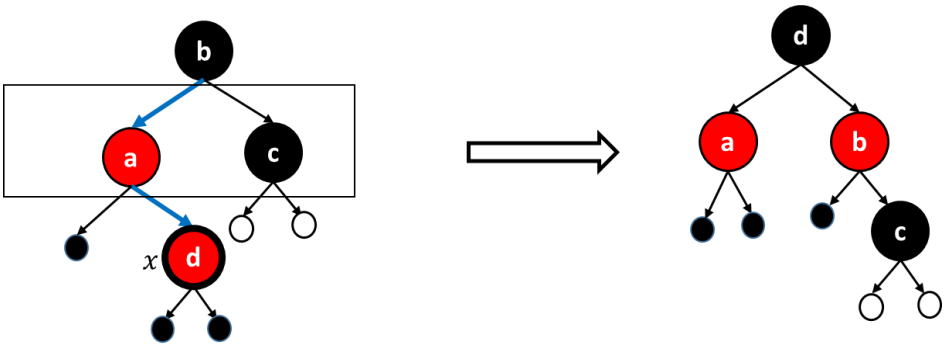


Рисунок 1.73.

Несложно видеть, что эти действия аналогичны LR-повороту для АВЛ-дерева (для случая, приведённого на рисунке 1.69, г) действия будут аналогичны RL-повороту для АВЛ-дерева).

1.2.2.3. Пример построения красно-чёрного дерева

Пример 1.10. Построить красно-чёрное дерево для последовательности чисел:

1, 2, 7, 3, 8, 14, 9.

Решение. На рисунке 1.74 проиллюстрированы этапы добавления вершин с ключами 1, 2 и 7.

При добавлении вершин с ключами 1 и 2 все свойства красно-чёрного дерева выполнялись.

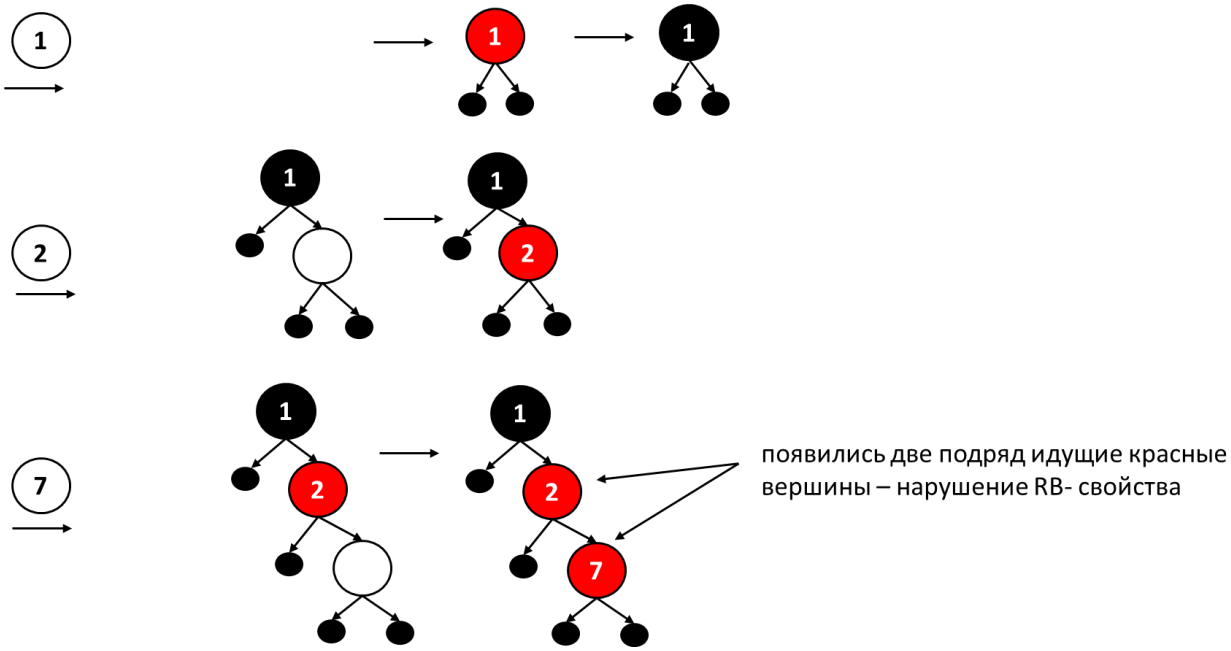


Рисунок 1.74. Добавление вершин с ключами 1, 2 и 7.

После добавления вершины ***x*** с ключом 7 появились две подряд идущие вершины красного цвета (это вершины с ключами 2 и 7). У вершины с ключом 7 отец и дядя разного цвета. Так как вершина ***x*** (вершина с ключом 7), её отец (вершина с ключом 2) и её дед (вершина с ключом 1) располагаются на одной линии, то выполним перекрашивание и одно вращение. В результате выполненных действий RB-свойства восстановлены, операция добавления вершины ***x*** с ключом 7 выполнена успешно (рисунок 1.75).

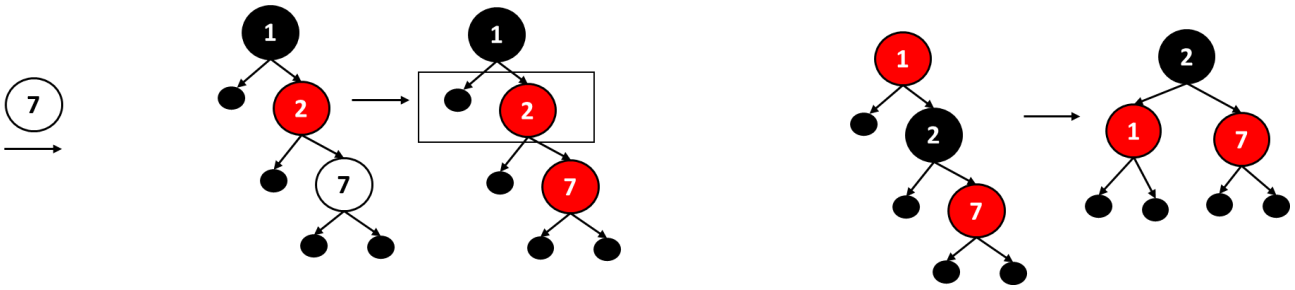


Рисунок 1.75. Добавление вершины с ключом 7.

После добавления вершины ***x*** с ключом 3 появились две подряд идущие вершины красного цвета (это вершины с ключами 3 и 7). Так как отец и дядя вершины с ключом 3 одного цвета (это вершины с ключами 7 и 1), то выполним перекрашивания и рассмотрим в качестве текущей вершины ***x*** вершину с ключом 2. Так как вершина с ключом 2 становится корнем дерева, то перекрасим её в чёрный цвет. RB-свойство восстановлены, операция добавления вершины с ключом 3 выполнена успешно (рисунок 1.76).

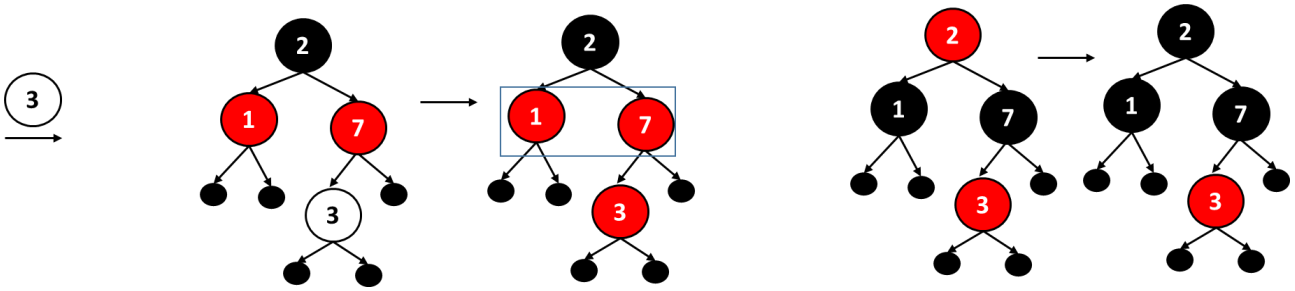


Рисунок 1.76. Добавление вершины с ключом 3.

После добавления вершины ***x*** с ключом 8 все RB-свойства выполнены, так как отец вершины (вершина с ключом 7) имеет чёрный цвет (рисунок 1.77). Операция добавления вершины ***x*** с ключом 8 выполнена успешно.

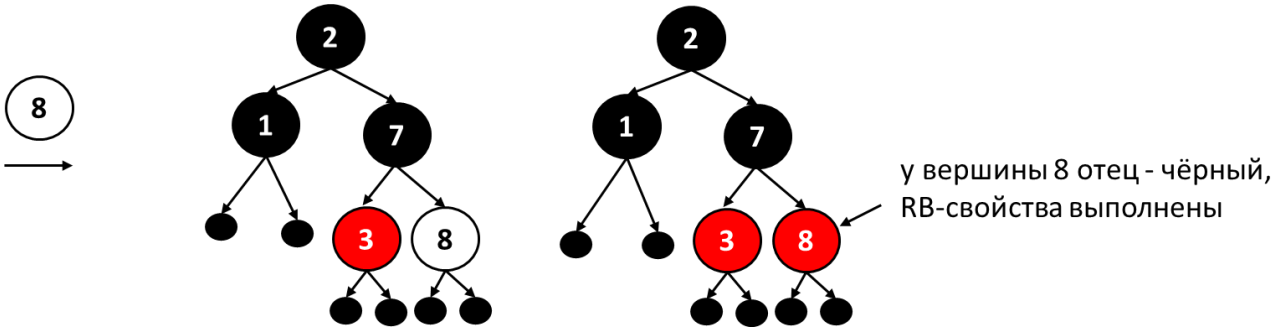


Рисунок 1.77. Добавление вершины с ключом 8.

После добавления вершины ***x*** с ключом 14 появились две подряд идущие вершины красного цвета (это вершины с ключами 8 и 14). Так как отец и дядя вершины с ключом 14 одного цвета (это вершины с ключами 8 и 3), то выполним перекрашивания и рассмотрим в качестве текущей вершины ***x*** вершину с ключом 7. Отец вершины с ключом 7 (вершина с ключом 2) имеет чёрный цвет, поэтому все RB-свойства выполнены и процедура добавления вершины с ключом 14 выполнена успешно (рисунок 1.78).

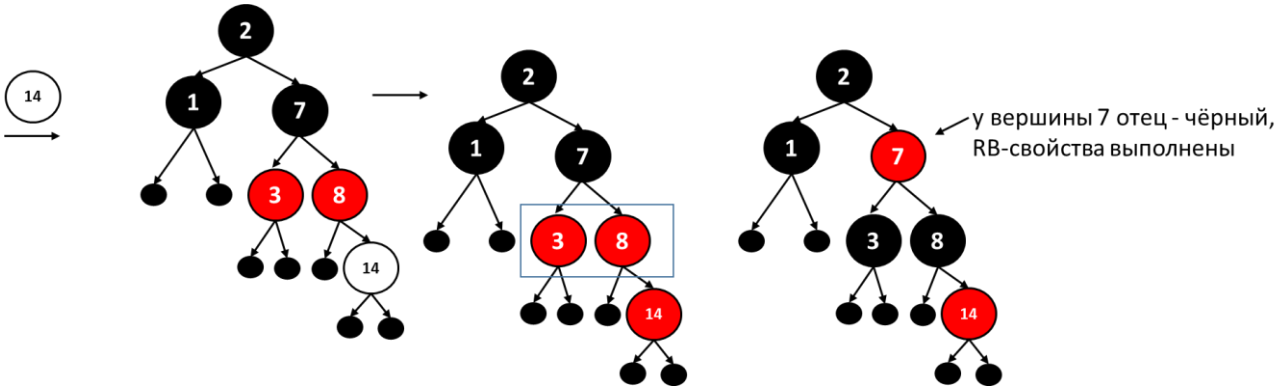


Рисунок 1.78. Добавление вершины с ключом 14.

После добавления вершины ***x*** с ключом 9 появились две подряд идущие вершины красного цвета (это вершины с ключами 9 и 14). Так так как у вершины с ключом 14 отец и дядя разного цвета, то выполним перекрашивания и два вращения. Операция добавления вершины ***x*** с ключом 9 выполнена успешно (рисунок 1.79).

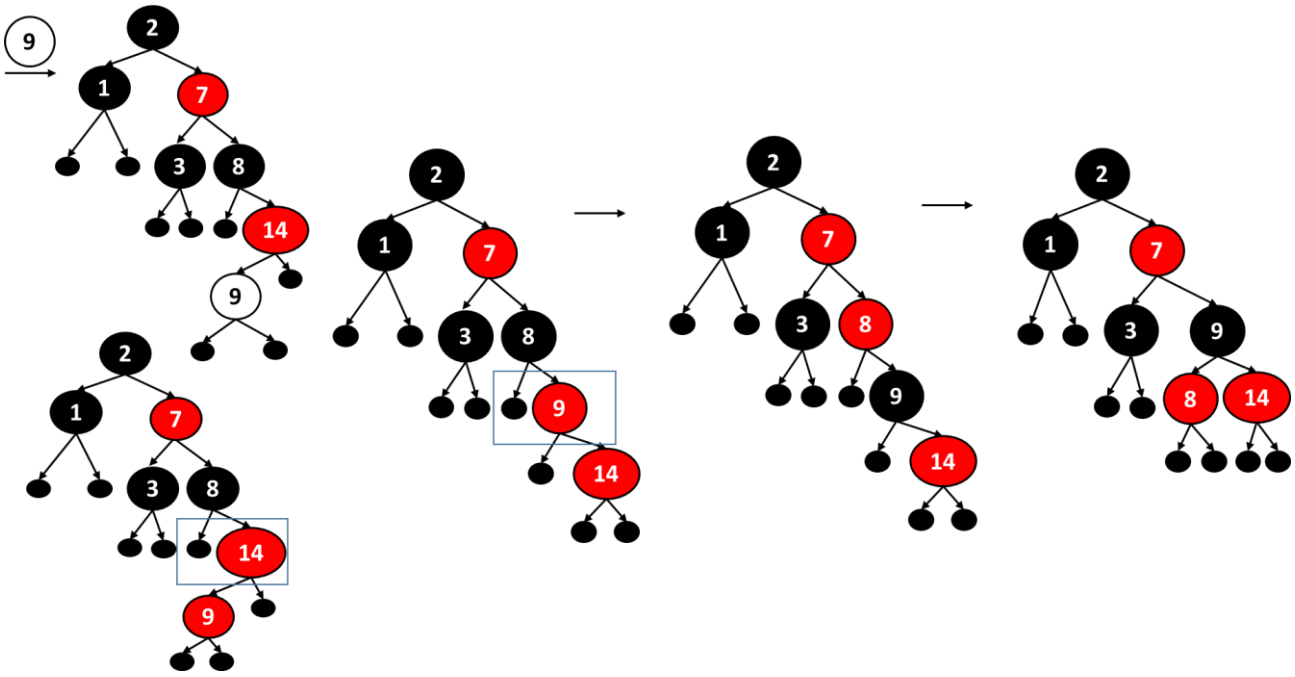


Рисунок 1.79. Добавление вершины с ключом 9.

Построение RB-дерева для последовательности чисел 1, 2, 7, 3, 8, 14, 9 завершено.

Вычислительная сложность операции добавления элемента с ключом ***x*** в красно-чёрное дерево есть $O(\log n)$, так как:

1. Поиск отца для добавляемой вершины – $O(\log n)$.
2. Добавление вершины – $O(1)$.
3. Перекрашивания – $O(\log n)$.
4. Повороты (не более 2-х) – $O(\log n)$.

1.2.2.4. Удаление ключа из дерева. Восстановление RB-свойств

Для удаления ключа из красно-чёрного дерева будем использовать следующий алгоритм.

Алгоритм удаления ключа из RB-дерева

1. Удаляем вершину из дерева по аналогии с тем, как это делали в бинарном поисковом дереве. Пусть *y* – фактически удалённая вершина.

2. Если удалённая вершина *y* имела красный цвет, то все RB-свойства будут выполняться и операция удаления элемента завершена.

3. Если удалённая вершина *y* имела чёрный цвет, то любой путь, через неё проходивший, теперь содержит на одну чёрную вершину меньше. Нарушается RB-свойство 4), которое требует, чтобы любой путь из корня в листья содержал одинаковое количество чёрных вершин. Восстановим RB-свойство.

Введём следующие обозначения.

y – фактически удалённая вершина;

x – единственный ребёнок вершины *y* (если детей у вершины *y* не было, то $x = NULL$;

$f(x) = f(y)$;

вершину, которая может быть окрашена, как в красный, так и в чёрный цвет, будем обозначать на рисунках *r / b*.

Если фактически удалённая вершина ***y*** имела чёрный цвет, то любой путь, через неё проходивший, теперь содержит на одну чёрную вершину меньше, поэтому поступим следующим образом:

1) если вершина ***x*** – красная, то сделаем её чёрной, теперь все RB-свойства выполнены;

2) если ***x*** – чёрная, то, будем при подсчёте числа чёрных вершин на пути от корня к листьям считать её за две чёрные вершины; однако дерево не предполагает такие «двойные чёрные вершины», поэтому нужно выполнить процедуру, которая превращает полученное дерево в настоящее красно-чёрное дерево.

Случай 1 (рисунок 1.80).

- ✓ ***x*** – чёрная и является левым сыном своего отца (ситуация правого сына выполняется симметрично),
- ✓ брат ***w*** – красный.

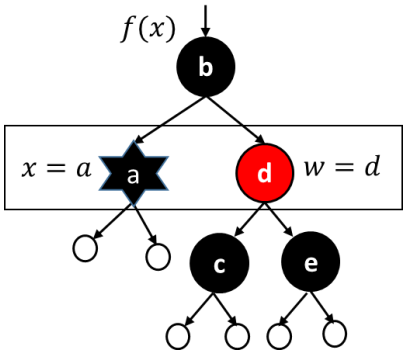


Рисунок 1.80.

На рисунке 1.81 показаны перекрашивания и вращение, которые нужно выполнить на первом этапе. После указанных действий: ***w*** – «новый брат» вершины ***x*** имеет чёрный цвет, вершину ***x*** считаем «дважды чёрной», а затем рассматриваются различные случаи (случаи 2, 3 и 4 будут рассмотрены позже) в зависимости от того, какого цвета дети у вершины ***w***. «Дважды чёрные» вершины на рисунках будем изображать в виде звезды

.

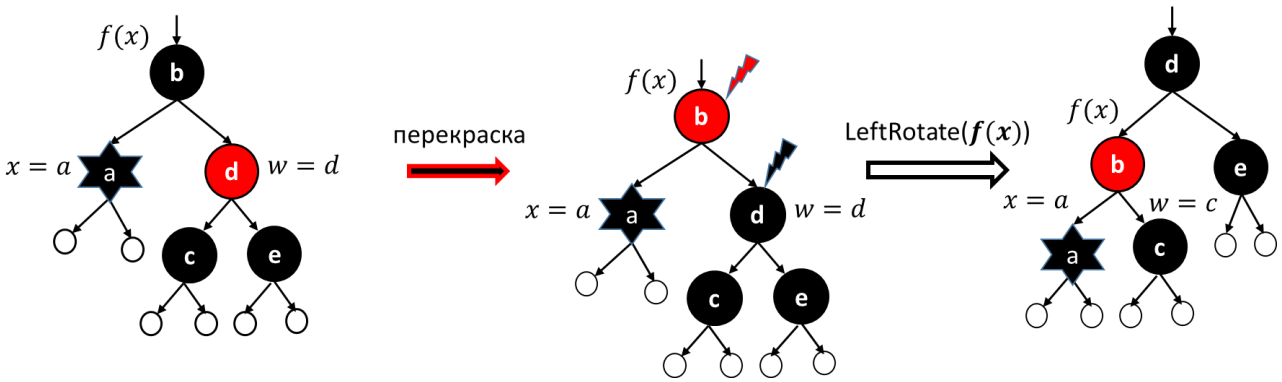


Рисунок 1.81.

Случай 2 (рисунок 1.82).

- ✓ ***x*** – «дважды чёрная» и является левым сыном своего отца,
- ✓ брат ***w***, левый и правый сын вершины *w*: ***Left*(*w*)**, ***Right*(*w*)** – чёрные.

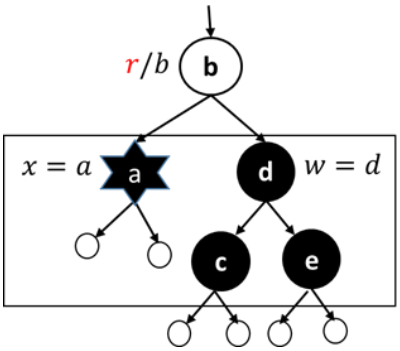


Рисунок 1.82.

На рисунках 1.83 и 1.84 продемонстрированы действия, которые будут выполнены: перекрашивания, снятие/добавление лишней чёрной окраски. Если вершина ***b*** была раньше чёрной, то после выполненных шагов она становится «дважды чёрной», поэтому полагаем $x = b$ и рекурсивно продолжаем восстановление RB-свойств для вершины x (рисунок 1.83).

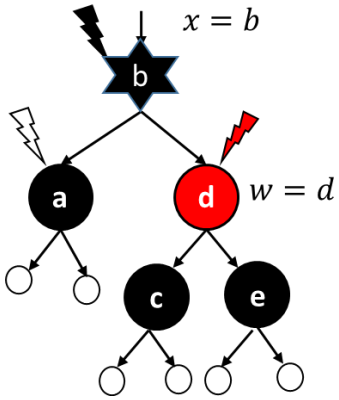


Рисунок 1.83.

Если вершина ***b*** была раньше красной, то она становится чёрной и процедура восстановления RB-свойств завершена (рисунок 1.84).

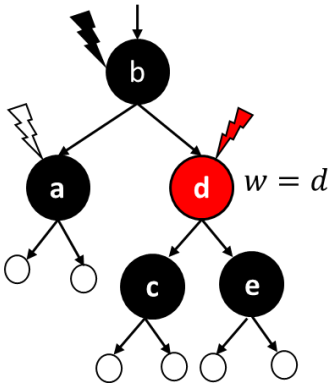


Рисунок 1.84.

Случай 3 (рисунок 1.85).

- ✓ ***x*** – «дважды чёрная» и является левым сыном своего отца,
- ✓ брат ***w*** и его правый сын ***Right***(***w***) – чёрные,
- ✓ левый сын вершины ***w: Left***(***w***) – красный.

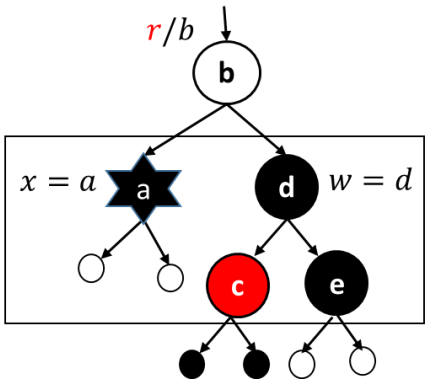


Рисунок 1.85.

На рисунке 1.86 продемонстрированы действия, которые будут выполнены для случая 3: перекрашивания и вращение.

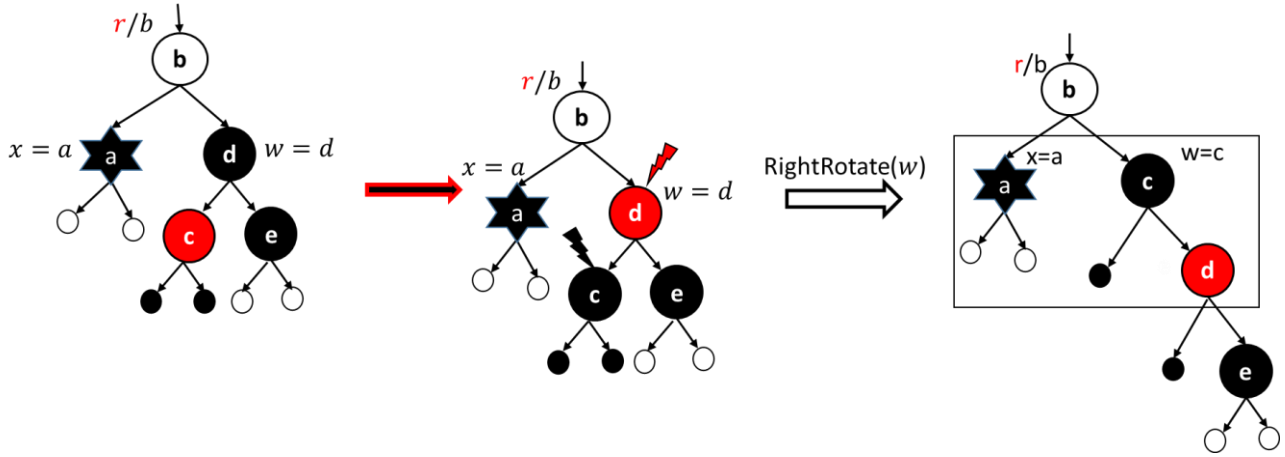


Рисунок 1.86.

В результате этих действий переходим к случаю 4: ***x*** – «дважды чёрная» и является левым сыном своего отца, брат ***w = c*** и его левый сын ***Left*(*w*)** – чёрные, правый сын вершины ***w:*** ***Right*(*w*)** – красный, который, как будет показано дальше, предполагает перекрашивание и одно вращение в результате чего RB-свойства будут восстановлены.

Случай 4 (рисунок 1.87).

- ✓ ***x*** – «дважды чёрная» и является левым сыном своего отца,
- ✓ брат ***w = c*** и его левый сын ***Left*(*w*)** – чёрные,
- ✓ правый сын вершины ***w:*** ***Right*(*w*)** – красный.

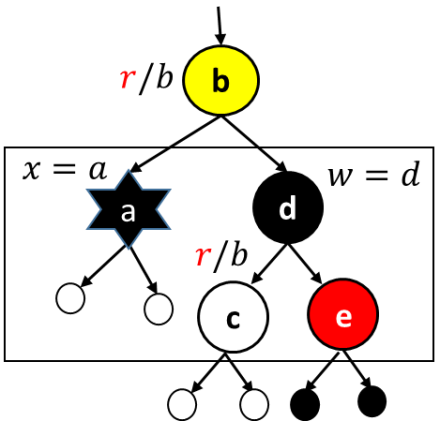


Рисунок 1.87.

На рисунке 1.88 продемонстрированы действия, которые будут выполнены для случая 4: перекрашивания и вращение в результате чего RB-свойства будут восстановлены (на рисунке «жёлтая молния» означает, что вершин *d* красится в тот же цвет, который был изначально у вершины *f*(*x*) (вершины *b*)).

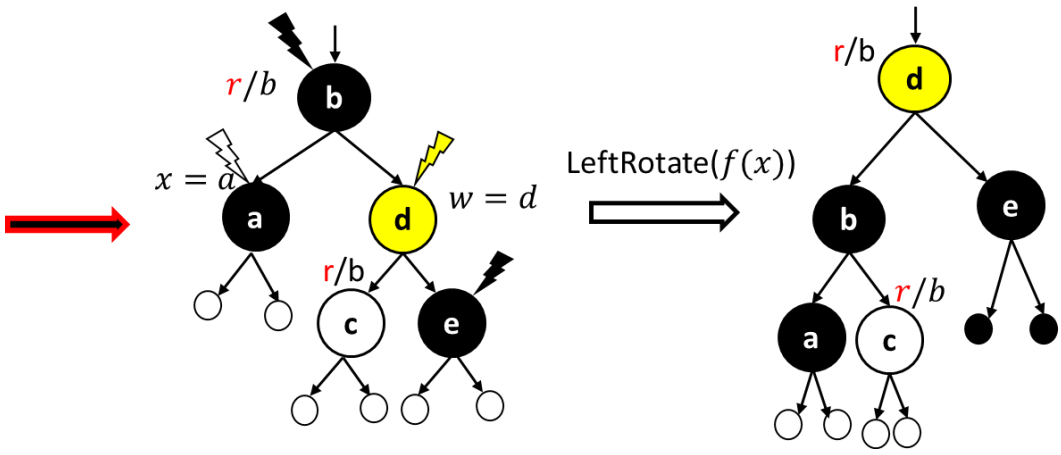


Рисунок 1.88.

Таким образом

– случай 4 заключается в перекраске, вращении и RB-свойства будут восстановлены;

– случай 3 сводится перекраской и вращением к случаю 4;

– случай 2 заключается в перекраске, а затем либо RB-свойства будут восстановлены, либо рекурсивно продолжаем их восстановление для вершины, которая находится на меньшей глубине;

– случай 1 сводится к случаям 2, 3 и 4; покажем, что переход к случаю 2 гарантирует, что после перекрашиваний RB-свойства будут восстановлены.

Случай 1 (продолжение, сведение к случаю 2 и завершение, рисунок 1.89):

✓ если у нового брата вершины ***x*** оба сына – чёрные.

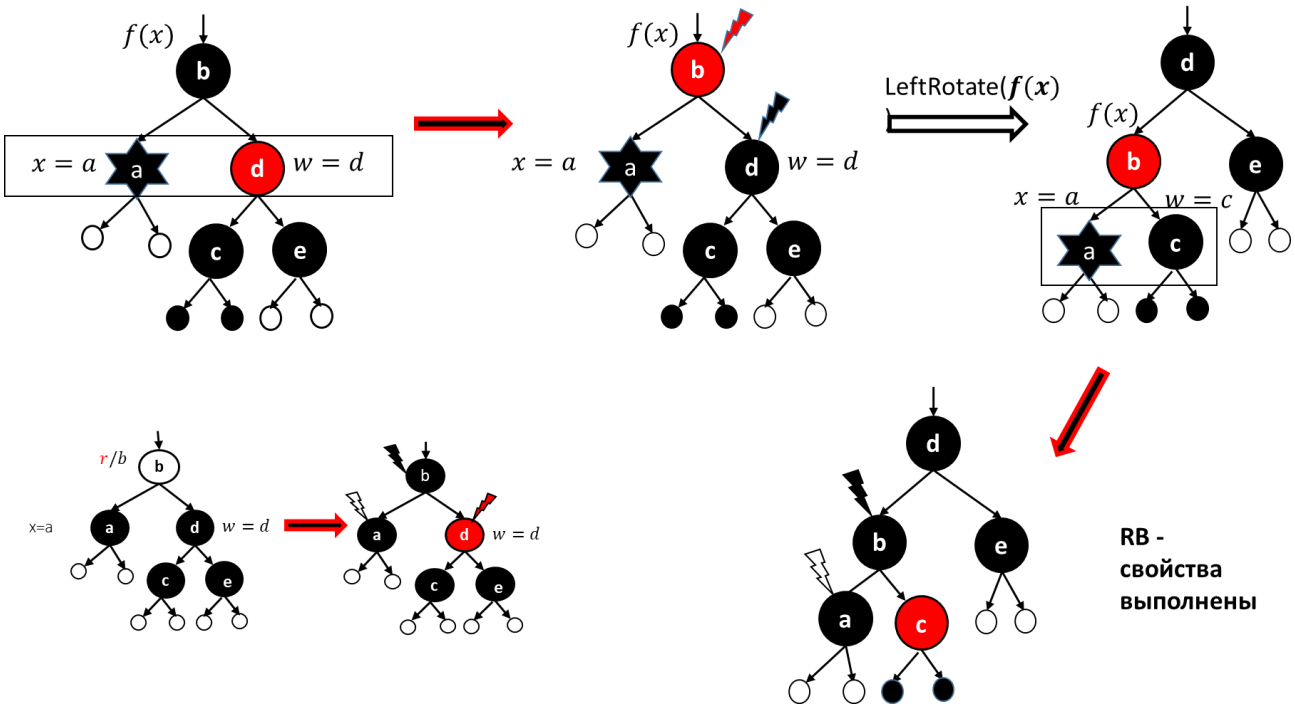


Рисунок 1.89.

Оценим вычислительную сложность операции восстановления RB-свойств после удаления вершины из красно-чёрного дерева.

Случаи 1, 3 и 4. Выполняются за время $O(1)$ (выполняется самое большое 3 вращения).

Случай 2. При каждом выполнении этого случая цикл, возможно, продолжит свою работу. Одна итерация цикла выполняется за время $O(1)$, но при каждом повторении указатель на вершину ***x*** перемещается вверх по дереву, никакие вращения при этом не происходят, поэтому количество повторений случая 2 ограничено высотой дерева $h = O(\log n)$.

Таким образом время на восстановление RB-свойств после выполнения операции удаления элемента равно $O(\log n)$.

Вычислительная сложность операции удаления элемента – $O(\log n)$:

1. Поиск удаляемой вершины – $O(\log n)$.
2. Непосредственное удаление вершины – $O(\log n)$.
3. Все перекрашивания – $O(\log n)$.
4. Вращения (не более 3-х) – $O(1)$.

2. ПРАКТИЧЕСКИЙ РАЗДЕЛ

2.1. Общие задачи по теме

Задача 0.1. Построить дерево

По набору ключей постройте бинарное поисковое дерево и выполните его прямой левый обход.

Формат входных данных

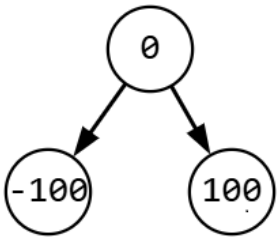
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Ключи задаются в формате по одному в строке.

В поисковом дереве все ключи по определению уникальны, поэтому при попытке добавить в дерево ключ, который там уже есть, он игнорируется.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом дерева.

входной файл	*выходной файл*
0 100 -100	0 -100 100



Задача 0. 2. Удалить из дерева

По набору ключей постройте бинарное поисковое дерево. Удалите из него ключ (правым удалением), если он есть в дереве. Выполните прямой левый обход полученного дерева.

Формат входных данных

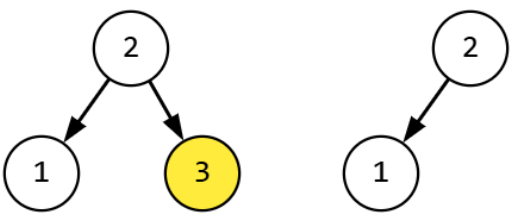
В первой строке записано целое число – ключ, который нужно удалить из дерева. Вторая строка пустая. Последующие строки содержат последовательность чисел – ключи вершин в порядке добавления в дерево. Ключи задаются в формате по одному в строке.

Дерево содержит хотя бы две вершины. Напомним, что в поисковом дереве все ключи по определению уникальны, поэтому при попытке добавить в дерево ключ, который там уже есть, он игнорируется.

Формат выходных данных

Выведите последовательность ключей вершин, полученную прямым левым обходом дерева.

входной файл	*выходной файл*
3 2 3 1	2 1



Задача 0. 3. Является ли бинарное дерево поисковым?

Задано бинарное дерево. Необходимо проверить, является ли оно поисковым. Будем предполагать, что в бинарном поисковом дереве могут быть вершины с одинаковыми ключами. Тогда для того, чтобы дерево было поисковым, должно выполняться следующее требование: для каждой вершины x все ключи в левом поддереве вершины x меньше ключа вершины x, а все ключи в правом поддереве больше либо равны ключу вершины x.

Формат входных данных

Первая строка входного файла содержит единственное целое число n $\left(1 \le n \le 8 \cdot 10^5\right)$ – количество вершин в дереве.

Следующая строка содержит одно целое число m $\left(-2^{31} \le m \le 2^{31} - 1\right)$ – значение в корневой вершине дерева.

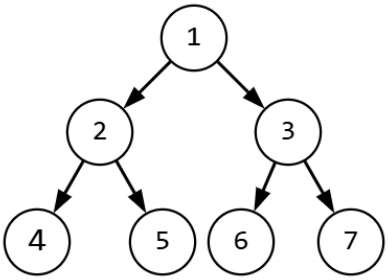
В каждой из последующих $n-1$ строк через пробелы перечисляются три параметра m, p и c, которые задают какую-либо вершину дерева: m – целое число $\left(-2^{31} \le m \le 2^{31} - 1\right)$, значение, записанное в вершине; p – целое число $\left(1 \le p \le n-1\right)$, номер строки входного файла, в которой был задан родитель текущей вершины (нумерация строк с нуля). Гарантируется, что p меньше, чем номер текущей строки. c – может принимать одно из двух значений `L` или `R`. Значение `L` указывает на то, что текущая вершина присоединена к родительской слева, `R` – справа.

Гарантируется, что совокупность всех строк задаёт корректное бинарное дерево.

Формат выходных данных

В единственной строке выведите `YES`, если заданное дерево является бинарным деревом поиска, и `No` в противном случае.

входной файл	*выходной файл*
7 1 2 1 L 4 2 L 3 1 R 5 2 R 6 4 L 7 4 R	No



2.2. Индивидуальные задачи по теме

Задача 1.

Найти среднюю по значению вершину из вершин дерева, у которых число потомков в левом поддереве не равно числу потомков в правом поддереве. Удалить её (правым удалением), если такая вершина существует. Выполнить прямой левый обход полученного дерева. Если у вершины отсутствует некоторое поддерево, то число вершин этого поддерева полагаем равным нулю.

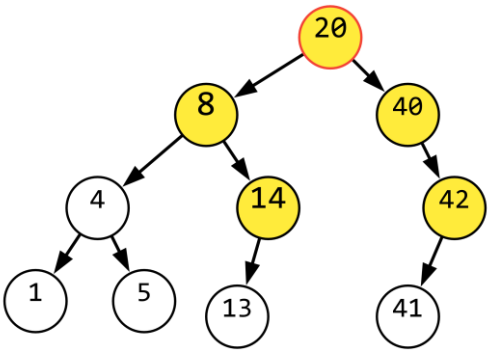
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
20 8 40 42 14 4 13 41 5 1	40 8 4 1 5 14 13 42 41



Для дерева, приведённого в примере, условию удовлетворяют пять вершин: 8, 14, 20, 40 и 42. Вершина 20 будет средней по значению среди них.

В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту. Пусть m – чётно. Тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих свойству нет, а в задаче ничего из дерева удалять не нужно

Задача 2.

Найти среднюю по значению вершину из вершин дерева, у которых высота левого поддерева не равна высоте правого поддерева. Удалить её (правым удалением), если такая вершина существует.

Выполнить прямой левый обход полученного дерева.

Если у вершины отсутствует некоторое поддерево, то его высоту полагаем равной -1.

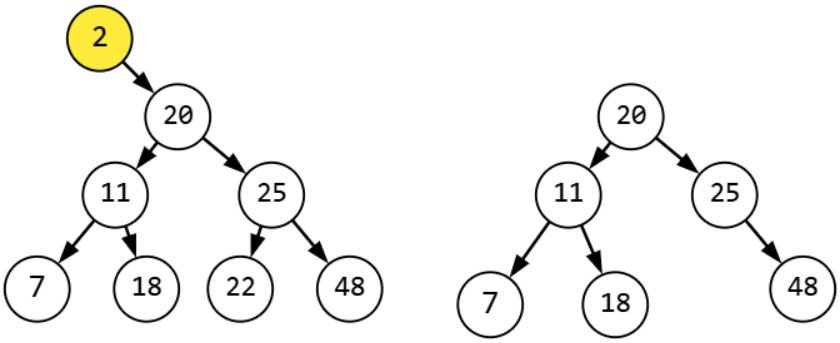
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
2 20 11 18 25 22 7 48	20 11 7 18 25 22 48



Для дерева, приведённого в примере, условию удовлетворяет только одна вершина 2. У всех остальных вершин высоты их поддеревьев равны. Вершина с ключом 2 будет средней по значению.

В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m.

Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту.

Пусть m – чётно. Тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих свойству нет, а в задаче ничего из дерева удалять не нужно.

Задача 3.

Найти среднюю по значению из вершин дерева, у которых число потомков в левом поддереве отличается от числа потомков в правом поддереве на единицу. Удалить её (правым удалением), если такая вершина существует. Выполнить прямой левый обход полученного дерева. Если у вершины нет некоторого поддерева, то число вершин этого поддерева полагаем равным нулю.

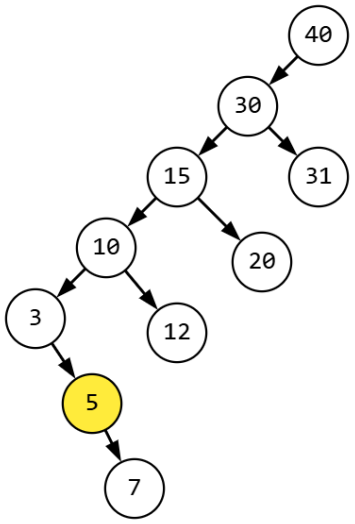
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
40 30 15 10 20 31 3 12 5 7	40 30 15 10 3 7 12 20 31



Для дерева, приведённого в примере, условию удовлетворяет только одна вершина с ключом 5. У всех остальных вершин число потомков в левом поддереве отличается от числа потомков в правом поддереве не на единицу. Вершина 5 будет средней по значению. В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту. Пусть m – чётно. Тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих свойству нет, а в задаче ничего из дерева удалять не нужно.

Задача 4.

Найти высоту дерева H и удалить (правым удалением) среднюю по значению из вершин на уровне $\lfloor H/2 \rfloor$, у которых число потомков в левом поддереве больше числа потомков в правом поддереве. Выполнить прямой левый обход полученного дерева. Если у вершины нет некоторого поддерева, то число вершин этого поддерева полагаем равным 0.

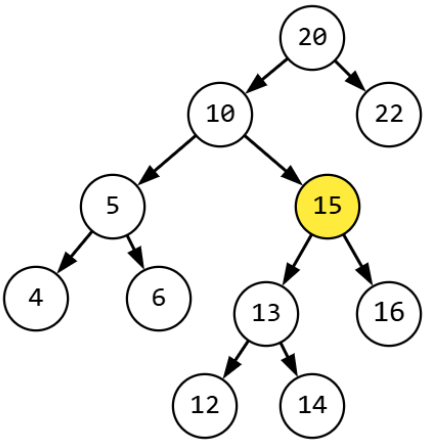
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
20 10 22 15 13 12 14 5 6 16 4	20 10 5 4 6 16 13 12 14 22



Для дерева, приведённого в примере, высота $H=4$. На уровне $\lfloor H/2 \rfloor=2$ находятся две вершины 5 и 15, но только 15 удовлетворяет условию задачи: в левом поддереве у неё 3 потомка, а в правом – 1. Вершина 15 и будет средней по значению.

В общем случае, предположим, что число вершин на нужном уровне $\lfloor H/2 \rfloor$, удовлетворяющих свойству, равно m. Пусть m – нечётно. Тогда, если на уровне упорядочить ключи всех вершин, которые удовлетворяют свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil+1$ по счёту. Пусть m – чётно. Тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих свойству нет, а в задаче ничего из дерева удалять не нужно.

Задача 5.

Среди полупутей минимальной положительной длины между листьями выбрать тот, у которого сумма ключей вершин минимальна.

Если такой полупуть существует, то удалить (правым удалением) центральную вершину этого полупути.

Если решение неоднозначно, то выбирать такой минимальный полупуть между листьями, корневая вершина которого имеет минимальное ключевое значение.

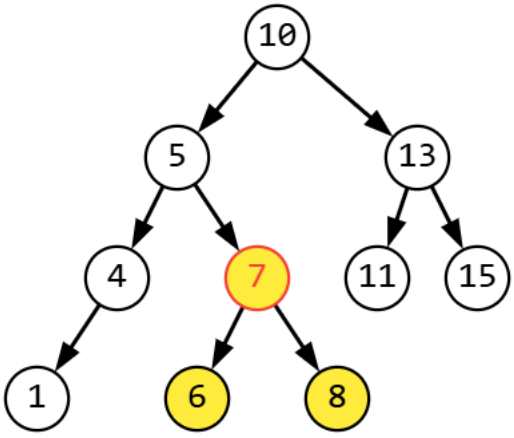
Формат входных данных

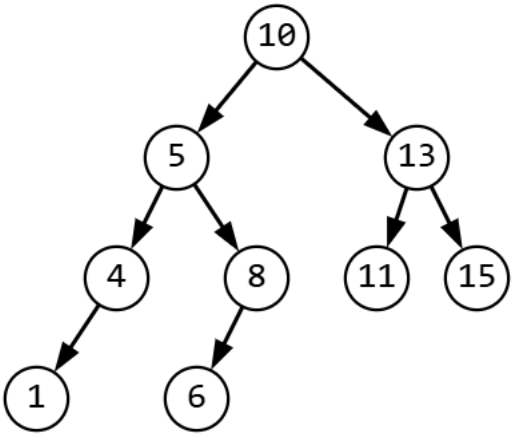
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 5 13 7 4 11 6 8 15 1	10 5 4 1 8 6 13 11 15





Для дерева, приведённого в примере, минимальная положительная длина полупути между листьями равна 2. Один такой полупуть соединяет вершины 6 и 8, а второй – 11 и 15. Среди этих двух полупутей длины 2 минимальная сумма ключей будет у полупути, соединяющего вершины 6 и 8 (действительно, $6+7+8<11+13+15$). Центральная вершина искомого полупути – это вершина с ключом 7 (на рисунке выделена). Справа на рисунке продемонстрировано дерево после удаления вершины с ключом 7 (правое удаление).

Задача 6.

Найти наибольший полупуть между вершинами дерева, у которого сумма ключей вершин максимальна, и удалить (правым удалением) сначала центральную вершину этого полупути, если она существует, потом корень этого же полупути, если он не являлся одновременно и центральной вершиной этого полупути.

Выполнить прямой левый обход полученного дерева.

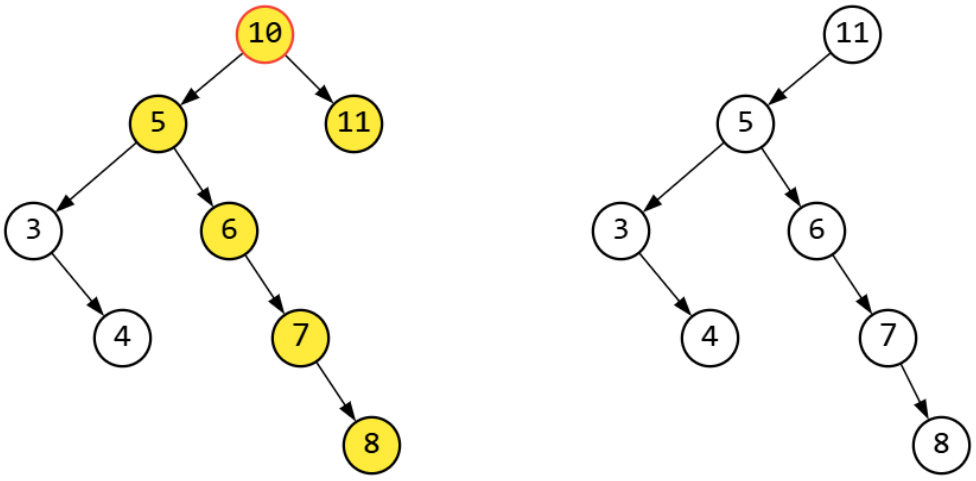
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 11 5 3 6 7 8 4	11 5 3 4 6 7 8



Для дерева, приведённого в примере, длина наибольшего полупути равна 5. В дереве два корня полупутей наибольшей длины: 5 и 10. Среди этих полупутей наибольшую сумму ключей имеет полупуть с корнем в вершине 10 (действительно, $11+10+5+6+7+8>4+3+5+6+7+8$).

Так как число вершин искомого полупути чётно ($=6$), то считаем, что центральной вершины у этого полупути нет. Поэтому в качестве центральной вершины ничего не удаляется.

Удаляем только вершину с ключом 10, которая является корнем искомого наибольшего полупути с максимальной суммой ключей вершин (правое удаление). На рисунке справа продемонстрировано дерево после удаления.

Задача 7.

Найти высоту дерева H и удалить (правым удалением) среднюю по значению из вершин на уровне $\lfloor H/2 \rfloor$, у которых высота левого поддерева равна высоте правого.

Выполнить прямой левый обход полученного дерева.

Если у вершины отсутствует некоторое поддерево, то его высоту полагаем равной -1.

Формат входных данных

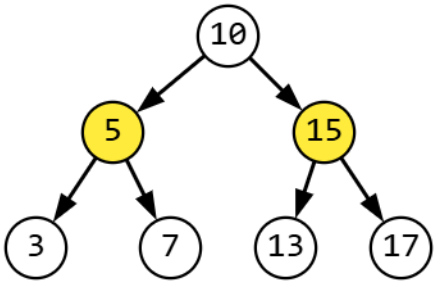
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 5 15 7 13 17 3	10 5 3 7 15 13 17



Для дерева, приведённого в примере, высота равна 2. На уровне $\lfloor H/2 \rfloor = 1$ находятся две вершины, у которых которых высота левого поддерева равна высоте правого. Так как число вершин на 1-ом уровне, удовлетворяющих требуемому свойству чётно $(=2)$, то средней по значению вершины, удовлетворяющей заданным свойствам нет. Ничего из дерева не удаляется.

В общем случае, предположим, что число вершин на нужном уровне $\lfloor H/2 \rfloor$, удовлетворяющих свойству, равно m.

Пусть m – нечётно. Тогда, если на уровне упорядочить ключи всех вершин, которые удовлетворяют свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту.

Пусть m – чётно. Тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих свойству нет, а в задаче ничего из дерева удалять не нужно.

Задача 8.

Найти такой наибольший полупуть между вершинами дерева, у которого корневая вершина находится на наименьшей глубине. Удалить (правым удалением) эту корневую вершину.

Формат входных данных

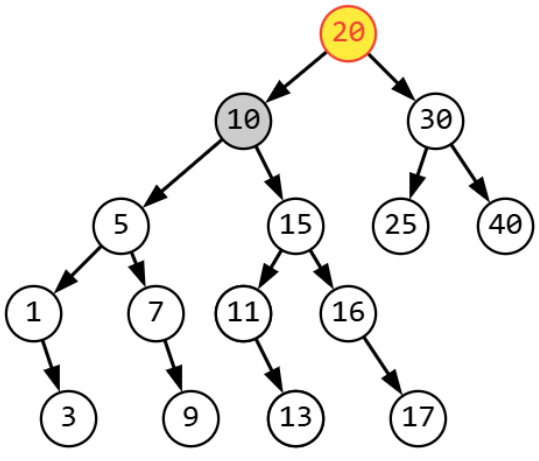
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

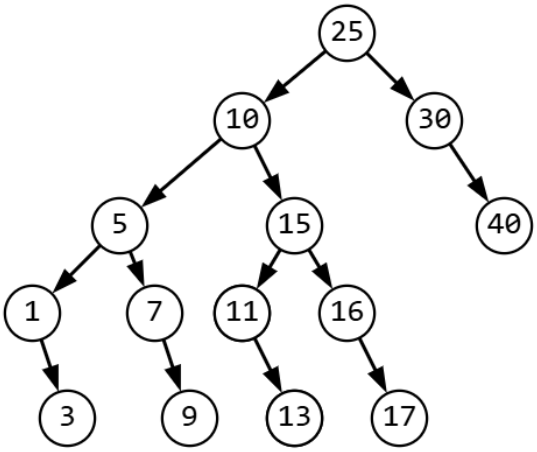
Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
20 10 30 5 1 7 3 9 25 40 15 11 13 16 17	25 10 5 1 3 7 9 15 11 13 16 17 30 40





Для дерева, приведённого в примере, длина наибольшего полупути равна 6. Вершины с ключами 10 и 20 являются корнями полупутей наибольшей длины. Наименьшую глубину среди вершины с ключами 10 и 20 имеет вершина $20\,(=0)$.

Удаляем вершину с ключом 20 (правое удаление). На рисунке справа продемонстрировано дерево после удаления вершины.

Задача 9.

Найти вершины, через которые проходят наибольшие полупути, и удалить (правым удалением) ту из них, ключ которой является вторым по значению.

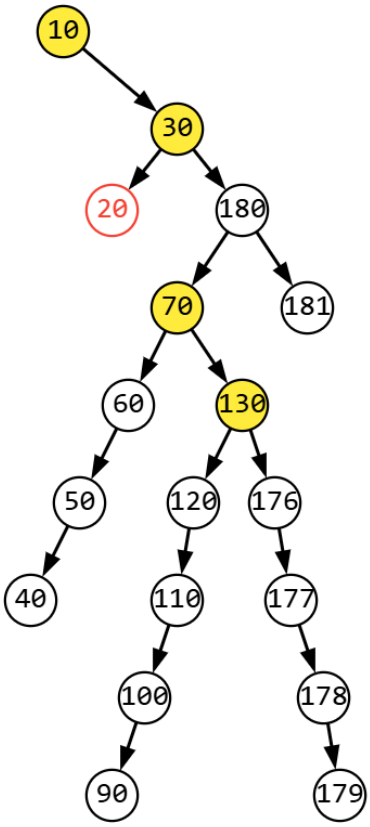
Формат входных данных

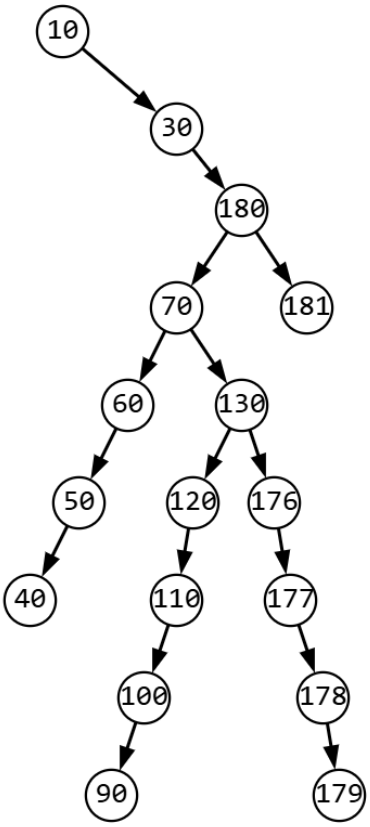
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 30 180 181 20 70 60 50 40 130 176 177 178 179 120 110 100 90	10 30 180 70 60 50 40 130 120 110 100 90 176 177 178 179 181





Для дерева, приведённого в примере, длина наибольшего полупути равна 8. В дереве вершины с ключами 10, 30, 70, 130 являются корнями полупутей наибольшей длины.

Полупути наибольшей длины проходят через все вершины дерева, за исключением вершины с ключом 181.

Удаляется из дерева вершина с ключом 20. На рисунке справа продемонстрировано дерево после удаления вершины с ключом 20.

Задача 10.

Найти вершины, через которые проходит наибольшее число наибольших (по длине) полупутей, и удалить их (правым удалением) обратным левым обходом.

Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

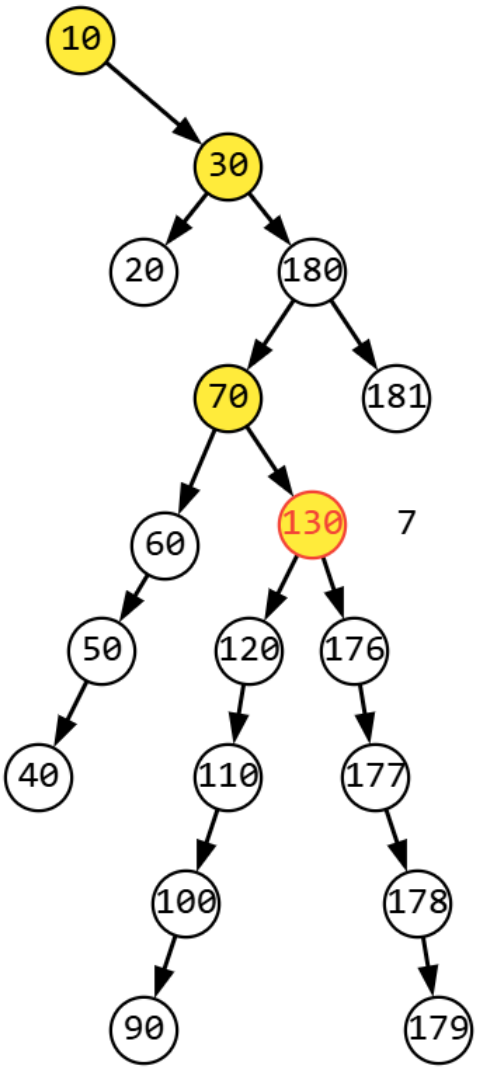
Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

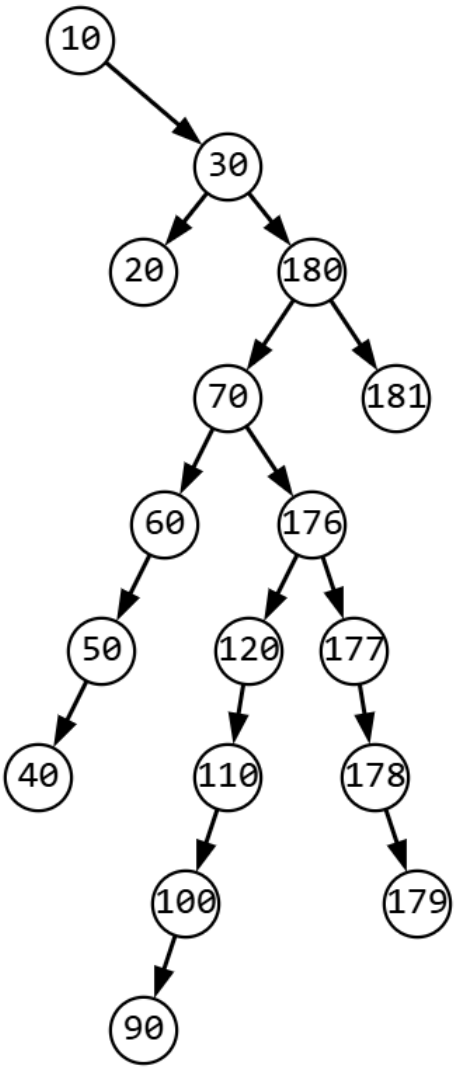
Гарантируется, что хотя бы одна вершина не подлежит удалению.

входной файл	*выходной файл*
10 30 180 181 20 70 60 50 40 130 176 177 178 179 120 110 100 90	10 30 20 180 70 60 50 40 176 120 110 100 90 177 178 179 181

Замечание 1.

Для удаления всех вершин, удовлетворяющих заданным требованиям, необходимо разработать алгоритм, работающий за время $O(n)$, где n – число вершин в дереве.





Для дерева, приведённого в примере, длина наибольшего полупути равна 8. В дереве вершины 10, 30, 70 и 130 являются корнями полупутей наибольшей длины.

Через вершину с ключом 130 проходит наибольшее число полупутей наибольшей длины: проходит 7 наибольших полупутей.

Для сравнения, укажем число наибольших полупутей, проходящих через остальные вершины дерева: 181 – ноль, 10 – два, 20 – два, 60 – два, 50 – два, 40 – два, 180 – четыре, 120 – четыре, 110 – четыре, 100 – четыре, 90 – четыре, 176 – четыре, 177 – четыре, 178 – четыре, 179 – четыре, 70 – шесть.

Удаляется из дерева вершина с ключом 130. На рисунке справа продемонстрировано дерево после удаления вершины 130 правым удалением.

Замечание 2.

В задаче нет необходимости для каждой вершины дерева считать точное количество наибольших полупутей, проходящих через неё. Достаточно будет проанализировать относительное расположение наибольших полупутей, что позволит сделать вывод о том, через какие вершины пройдет наибольшее число наибольших (по длине) полупутей.

Задача 11.

Найти такой наибольший полупуть, у которого сумма ключей крайних вершин минимальна.

Удалить (правым удалением) корневую вершину этого полупути.

Формат входных данных

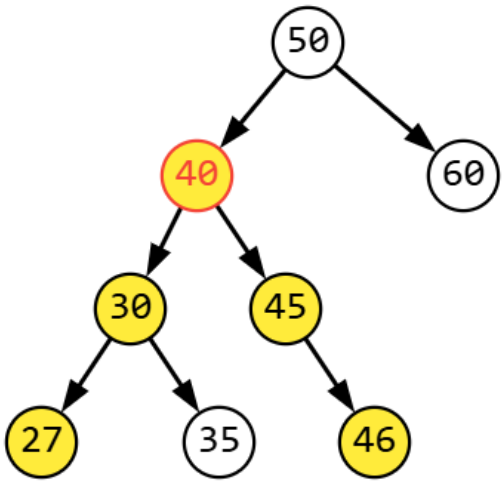
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

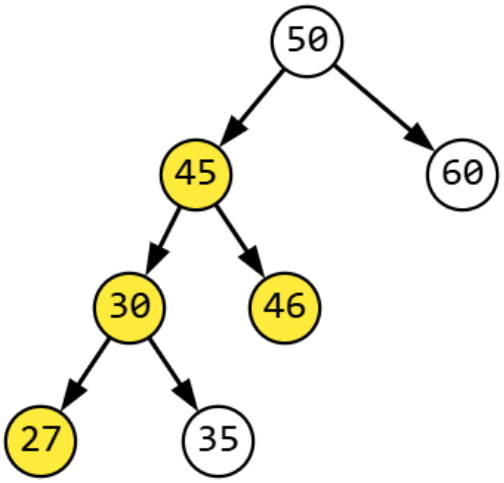
Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
50 40 60 30 45 27 35 46	50 45 30 27 35 46 60





Для дерева, приведённого в примере, длина наибольшего полупути равна 4. В дереве вершины 50 и 40 являются корнями полупутей наибольшей длины.

Наибольший полупуть, у которого сумма ключей крайних вершин минимальна, даёт корень наибольшего полупути с ключом 40 (искомый полупуть соединяет вершины с ключами 27 и 46).

Удаляется из дерева вершина с ключом 40.

На рисунке справа продемонстрировано дерево после удаления вершины с ключом 40 правым удалением.

Задача 12.

Найти наибольший полупуть между вершинами с разным числом потомков с минимальной суммой ключей крайних вершин (длина искомого полупути должна быть больше 0).

Если таких полупутей несколько, то выбрать из них тот, у которого корневая вершина имеет минимальное ключевое значение.

Удалить (правым удалением), если существует, среднюю по значению вершину этого полупути.

Если у вершины отсутствует некоторое поддерево, то число вершин этого поддерева полагаем равным 0.

Замечание.

В случае неоднозначности выбора удаляемой вершины (например, несколько полупутей максимальной длины между вершинами с разным числом потомков и с минимальной суммой ключей крайних вершин имеют один и тот же корень, но средние по значению вершины этих полупутей не совпадают) ничего из дерева удалять не нужно.

Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

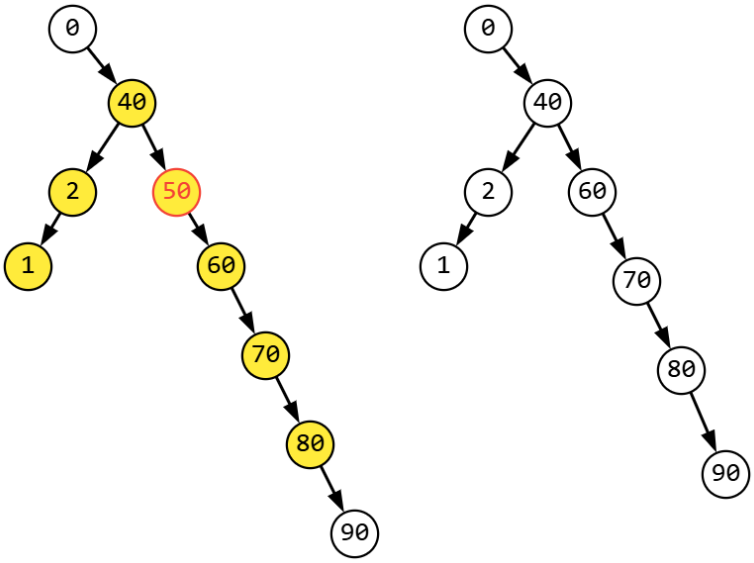
Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

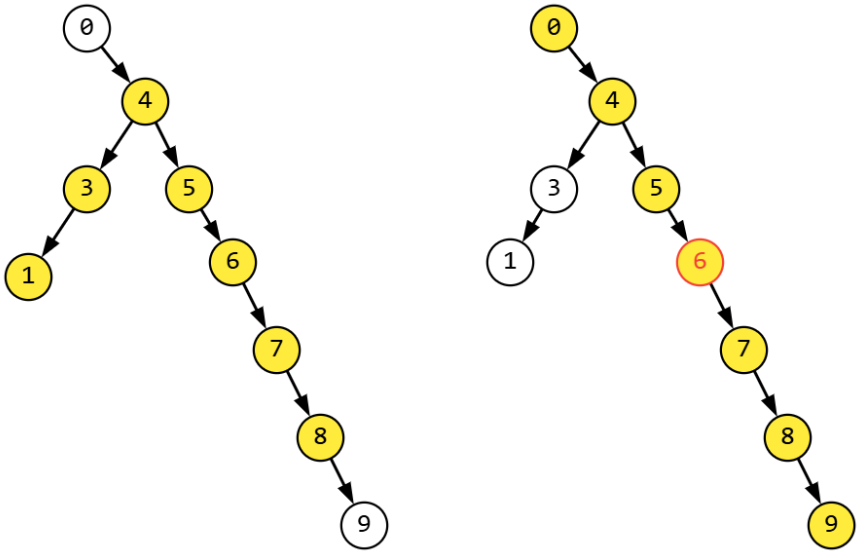
входной файл	*выходной файл*
0 40 50 60 70 80 90 2 1	0 40 2 1 60 70 80 90
0 4 5 6 7 8 9 3 1	0 4 3 1 5 7 8 9

Если число вершин искомого полупути чётно, то считаем, что средней по значению вершины этого полупути не существует, а значит, ничего из дерева не удаляем.

Для дерева, приведённого в примере 1, длина искомого полупути равна 6. Искомый полупуть соединяет вершину с ключом 1 (вершина является листом и у неё 0 потомков) и вершину с ключом 80 (у вершины 1 потомок). Существуют ещё два полупути длины 6, которые соединяют вершины с разным числом потомков (первый – соединяет вершины с ключами 2 и 90, второй – соединяет вершины с ключами 0 и 90), но для этих двух полупутей сумма ключей крайних вершин больше 81 ($2+90>81$ и $0+90>81$).



Для дерева, приведённого в примере 2, длина искомого полупути равна 6. Существуют три полупути длины 6, соединяющие вершины с разным числом потомков (первый – соединяет вершины с ключами 1 и 8, второй – соединяет вершины с ключами 3 и 9, третий – соединяет вершины с ключами 0 и 9).



Выбираем из них тот, у которого сумма крайних вершин минимальна (при неоднозначности выбираем тот, у которого корневая вершина имеет минимальное ключевое значение). Искомый полупуть изображен на рисунке справа, он соединяет вершину 0 (корень дерева) и вершину 9 (лист). Число вершин искомого полупути 7 – нечётно, средняя по значению вершина этого полупути, которая будет удалена – вершина с ключом 6.

Задача 13.

Необходимо найти наибольший полупуть между вершинами разной высоты с минимальной суммой ключей крайних вершин. Если таких полупутей несколько, то выбрать из них тот, у которого корневая вершина имеет наименьшее ключевое значение. Удалить (правым удалением), если существует, среднюю по значению вершину этого полупути. Если у вершины отсутствует некоторое поддерево, то его высоту полагаем равной –1.

Замечание. В случае неоднозначности выбора удаляемой вершины (например, несколько наибольших полупутей между вершинами разной высоты и с минимальной суммой ключей крайних вершин имеют один и тот же корень, но средние по значению вершины этих полупутей не совпадают) ничего из дерева удалять не нужно.

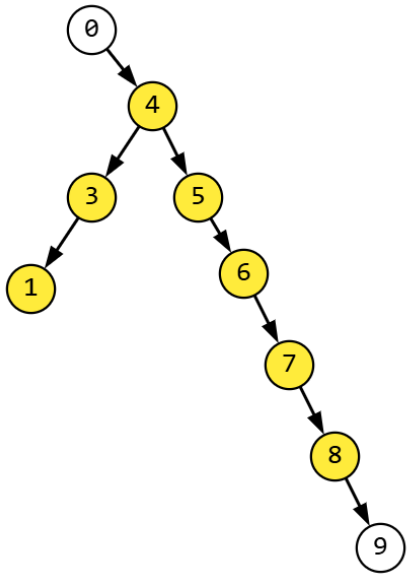
Формат входных данных

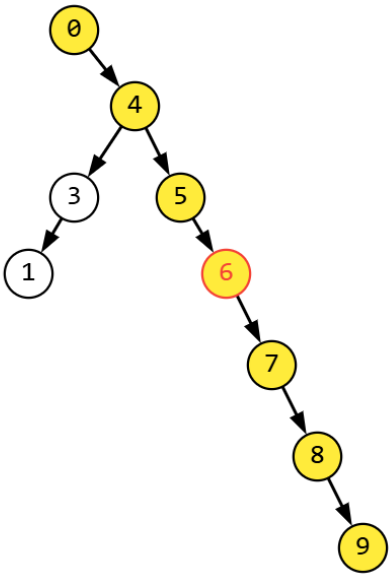
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
0 4 5 6 7 8 9 3 1	0 4 3 1 5 7 8 9





Искомый полупуть изображен на рисунке справа, он соединяет вершину с ключом 0 (высота 6) и вершину с ключом 9 (высота 0). Число вершин искомого полупути 7 – нечётно, средняя по значению вершина этого полупути, которая будет удалена – 6.

Задача 14.

Необходимо найти пути минимальной длины между корнем и листьями и удалить (левым удалением) в порядке возрастания ключей средние по значению вершины этих путей, если они существуют.

Замечание. Если некоторая вершина является средней по значению для нескольких путей минимальной длины, то удаляется она только один раз.

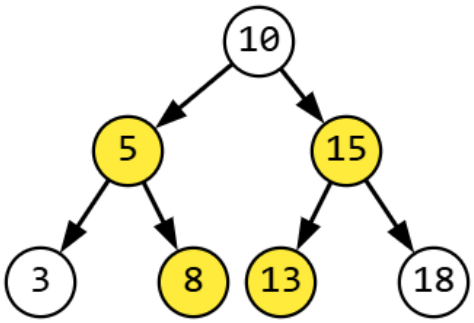
Формат входных данных

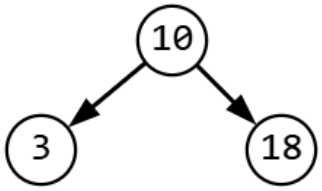
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 15 13 18 5 8 3	10 3 18





Для дерева, приведённого в примере, длина минимального пути между конем и листьями равна 2.

1) Первый путь соединяет вершину с ключом 10 и вершину с ключом 3 (средняя по значению вершина этого пути – 5).

2) Второй путь соединяет вершину с ключом 10 и вершину ключом 8 (средняя по значению вершина этого пути – 8).

3) Третий путь соединяет вершину ключом 10 и вершину ключом 13 (средняя по значению вершина этого пути – 13).

4) Четвёртый путь соединяет вершину ключом 10 и вершину ключом 18 (средняя по значению вершина этого пути 15)

Удаляем вершины в порядке возрастания ключей: 5, 8, 13 и 15.

На рисунке справа показано дерево после выполнения процедуры удаления.

Задача 15.

Определить, являются ли два дерева зеркальным отражением друг друга по структуре, если сначала удалить (правым удалением) корень каждого из деревьев.

Замечание.

1) Если деревья были тривиальными (одна вершина в каждом дереве), то в результате удаления корней получаются два не графа (*нуль-граф*), поэтому будем считать, что тривиальные деревья зеркально совпадают.

2) Если в результате удаления корней получается два тривиальных дерева, то будем считать, что они зеркально совпадают.

Формат входных данных

Входной файл содержит последовательности чисел – ключи вершин в порядке добавления в дерево.

Деревья разделяются строкой с единственным символом «*».

Формат выходных данных

Выходной файл должен содержать в первой строке сообщение YES или NO.

Если сообщение YES, то во второй строке вывести последовательность ключей вершин, полученную прямым левым обходом первого дерева, а в третьей строке – последовательность ключей вершин, полученную прямым левым обходом второго дерева.

Строки не должны заканчиваться пробелом.

входной файл	*выходной файл*
2 1 3 * 2 1 3	NO
96 90 97 80 95 85 * 20 30 40 50 33 45	YES 97 90 80 85 95 30 40 33 50 45

Иллюстрация для примера 1. После удаления корневых вершин (выполнено правое удаление) деревья не являются зеркальным отражением друг друга.

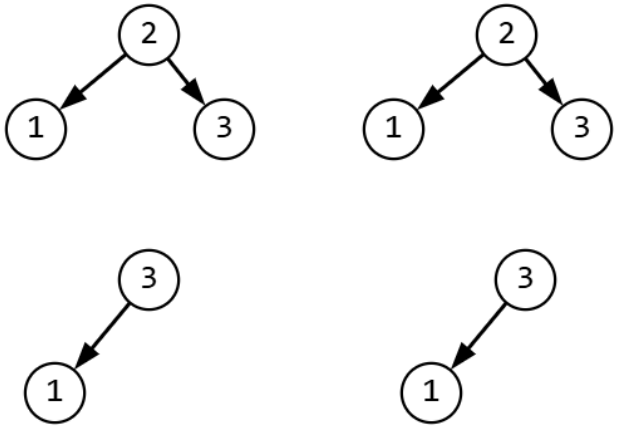
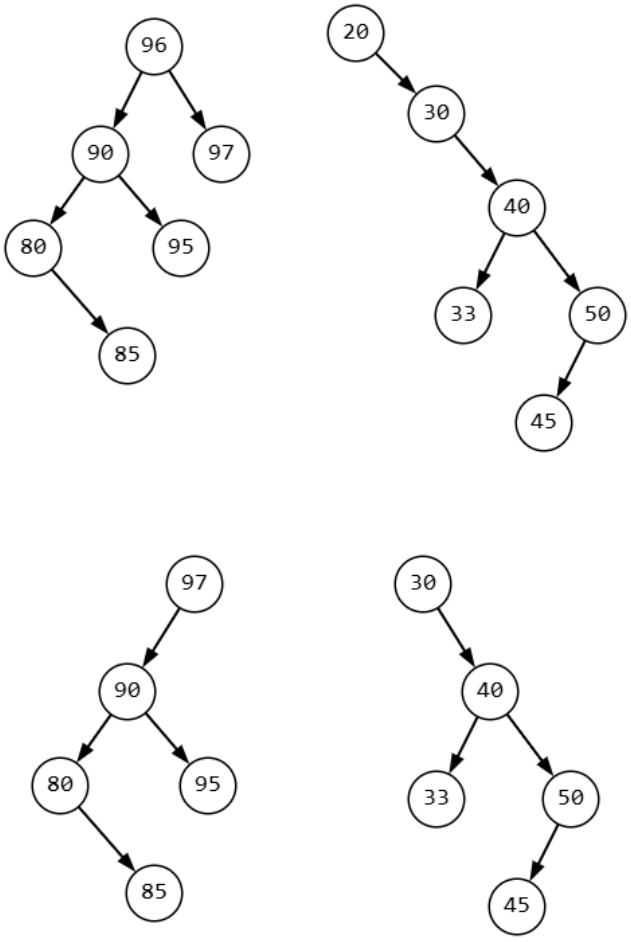


Иллюстрация для примера 2. Для первого дерева корневая вершина с ключом 96 удаляется правым удалением. Корень второго дерева – вершина с ключом 20 имеет только одно поддерево с корнем в вершине 30, поэтому после удаления вершины с ключом 20 корнем второго дерева становится вершина с ключом 30.



После удаления корневых вершин деревья являются зеркальным отражением друг друга.

Задача 16.

Необходимо удалить (правым удалением) корень дерева, а затем для каждой вершины определить, сколько наибольших полупутей проходят через неё.

Замечание. Если после удаления в дереве остается одна вершина, то считаем, что имеется один наибольший полупуть длины 0.

Формат входных данных

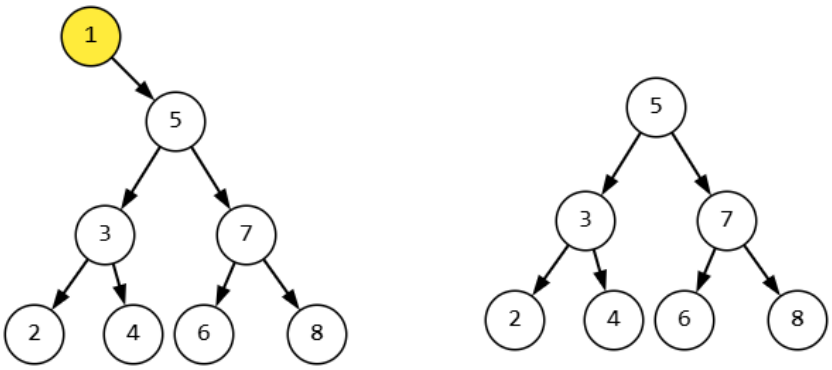
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

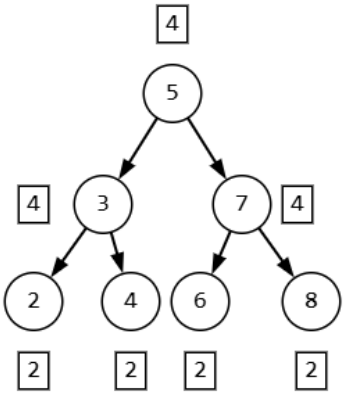
Выполните прямой левый обход итогового дерева. Для каждой вершины в отдельной строке выведите через пробел её ключ и количество полупутей наибольшей длины, проходящих через неё.

входной файл	*выходной файл*
1 5 3 7 2 4 6 8	5 4 3 4 2 2 4 2 7 4 6 2 8 2

На рисунке сначала приведено исходное дерево, а затем – дерево после удаления корневой вершины с ключом 1.



Длина наибольшего полупути равна 4. В дереве единственный корень наибольшего полупути – вершина с ключом 5. На рисунке возле каждой вершины указано число полупутей наибольшей длины, которые через неё проходят.



Задача 17.

Найти и удалить (правым удалением), если существует, среднюю по значению из вершин дерева, у которых высота левого поддерева отличается от высоты правого наибольшим образом. Выполнить прямой левый обход полученного дерева.

Если у вершины отсутствует некоторое поддерево, то его высоту полагаем равной -1.

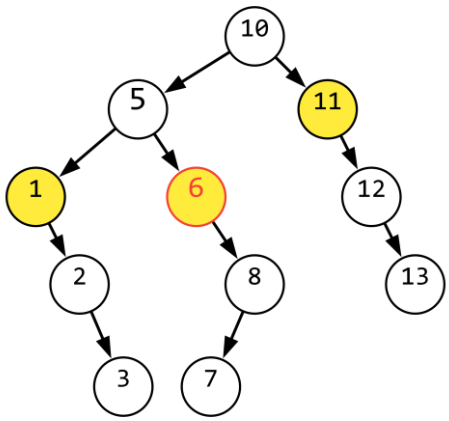
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 5 6 8 7 11 1 2 12 13 3	10 5 1 2 3 8 7 11 12 13



Для дерева, приведённого в примере, наибольшее значение, на которое отличаются у вершин высоты поддеревьев равно 2. Это значение 2 достигается для вершин с ключами 1, 6 и 11. Средней по значению среди них будет вершина с ключом 6.

В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту. Пусть m – чётно. Тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих свойству нет, а в задаче ничего из дерева удалять не нужно.

Задача 18.

Найти вершины, через которые проходит чётное *ненулевое* количество наибольших полупутей (наибольших среди *всех* возможных полупутей в дереве, а не только среди проходящих через конкретную вершину), и удалить (правым удалением) ту из них, ключ которой наименьший.

Если нет вершин, удовлетворяющих нужному свойству, то ничего из дерева удалять не нужно.

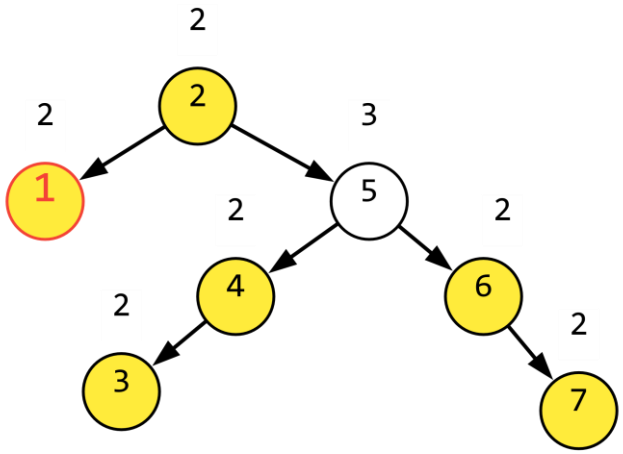
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
2 1 5 4 6 3 7	2 5 4 3 6 7



Для дерева, приведённого в примере, длина наибольшего полупути равна 4.

В дереве вершины с ключами 2 и 5 являются корнями полупутей наибольшей длины.

На рисунке для каждой вершины дерева указано количество наибольших полупутей, которые через неё проходят. Жёлтой заливкой на рисунке выделены вершины, через которые проходит чётное ненулевое количество наибольших полупутей. Наименьший ключ, среди выделенных вершин имеет вершина с ключом 1, которая и будет удалена.

Задача 19.

Найти вершины, через которые проходят наибольшие полупути, и удалить (правым удалением) самую высокую из них.

Формат входных данных

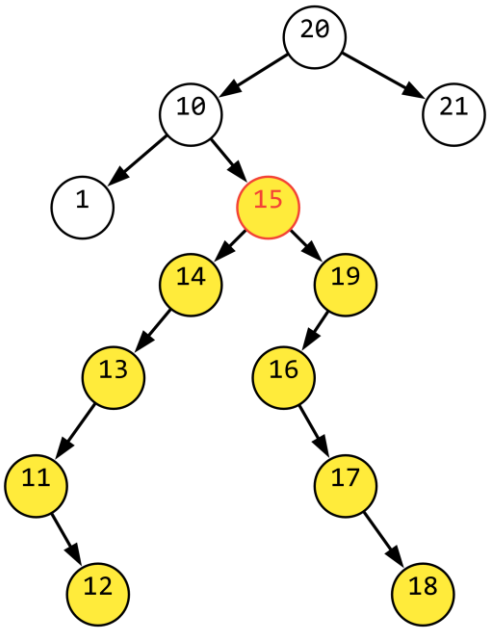
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
20 21 10 1 15 14 13 19 16 17 11 12 18	20 10 1 16 14 13 11 12 19 17 18 21

Для дерева, приведённого в примере, длина наибольшего полупути равна 8. На рисунке жёлтой заливкой выделены вершины, через которые проходят полупути наибольшей длины.



Среди выделенных вершин наибольшую высоту имеет вершина с ключом 15 ($h(15)=4$), которая и будет удалена из дерева (правым удалением).

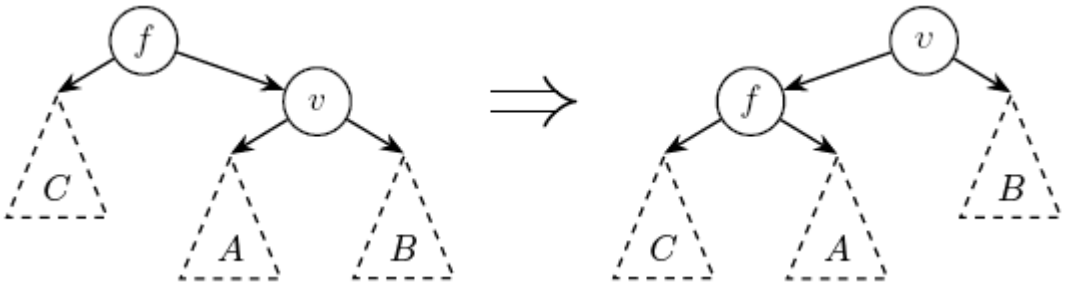
Задача 20.

Найти наибольший полупуть между вершинами разного уровня с минимальной суммой ключей крайних вершин и сделать центральную вершину этого полупути корнем дерева (методом поворотов).

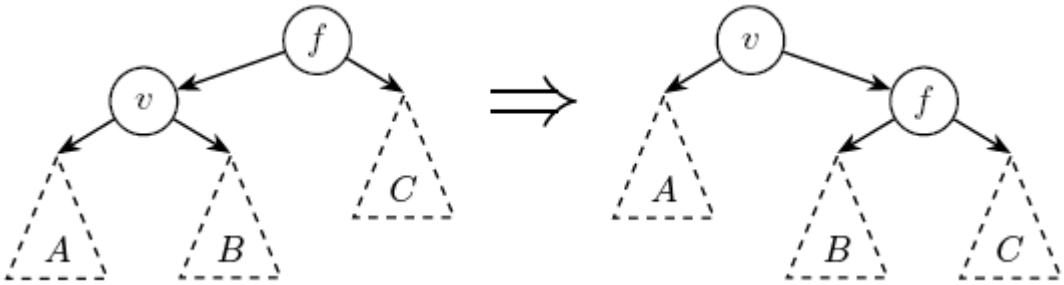
Замечания.

1. Исходное дерево содержит не менее двух вершин.
2. Если центральной вершины нет, то метод поворотов не выполняется.
3. Выходной файл должен содержать сначала минимальную сумму ключей крайних вершин, а затем массив вершин, полученный прямым левым обходом исходного дерева.
4. Для выполнения метода поворотов нужно спуститься из корня дерева к центральной вершине и поднять её (после выполнения очередного поворота центральная вершина оказывается на меньшей глубине).

Малый левый поворот



Малый правый поворот



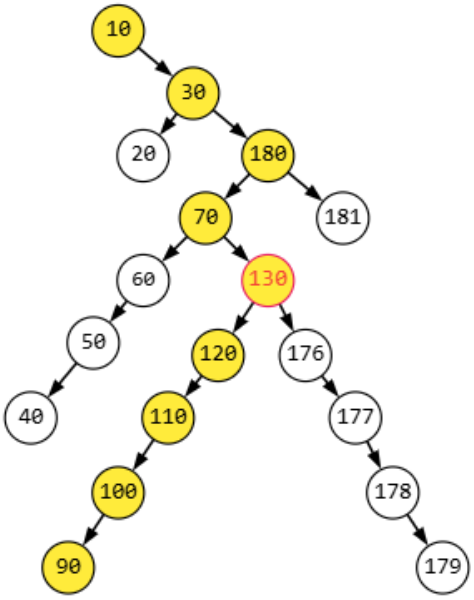
Формат входных данных

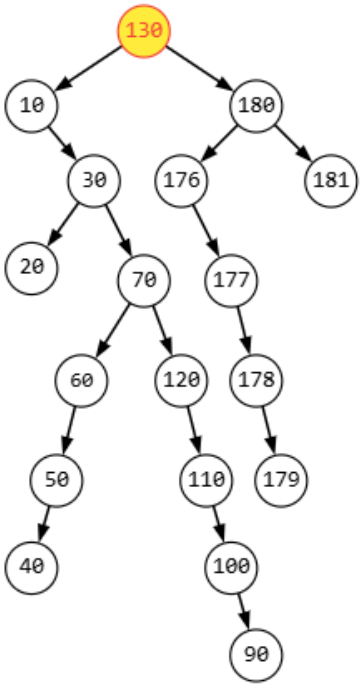
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать сначала сумму ключей крайних вершин найденного полупути, а затем последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 30 180 181 20 70 60 50 40 130 176 177 178 179 120 110 100 90	100 130 10 30 20 70 60 50 40 120 110 100 90 180 176 177 178 179 181





Для дерева, приведённого в примере, длина наибольшего полупути, крайние вершины которого расположены на разных уровнях, равна 8. Вершины с ключами 10, 30 и 70 являются корнями полупутей наибольшей длины, которые удовлетворяют заданным требованиям. Вершина с ключом 10 – корень искомого полупути наибольшей длины с минимальной суммой $(10+90=100)$ ключей крайних вершин (крайняя вершина с ключом 10 находится на 8-ом уровне, а крайняя вершина с ключом 90 находится на 0-ом уровне). Вершина с ключом 130 является центральной для искомого полупути.

На рисунке справа продемонстрировано дерево после того, как методом поворотов вершина с ключом 130 становится корнем дерева.

Задача 21.

Найти среднюю по значению из вершин дерева, у которых высоты поддеревьев равны, а число потомков в правом и левом поддеревьях отличается. Удалить её (правым удалением), если такая вершина существует.

Если у вершины отсутствует некоторое поддерево, то его высоту полагаем равной -1, а число вершин этого поддерева – 0.

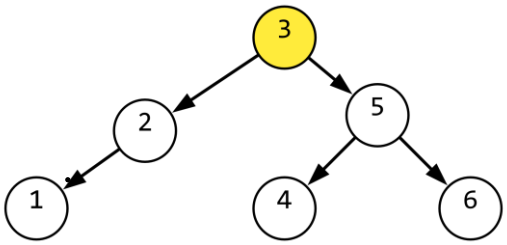
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
3 2 1 5 4 6	4 2 1 5 6



Для дерева, приведённого в примере, условию удовлетворяет только одна вершина с ключом 3 (у вершины с ключом 3 высота её левого поддрева равна 1, высота её правого поддерева равна 1, число вершин в её левом поддереве равно 2, а число вершин в её правом поддереве равно 3). Вершина с ключом 3 и будет средней по значению.

В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m.

Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту.

Пусть m – чётно, тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих свойству нет, а в задаче ничего из дерева удалять не нужно.

Задача 22.

Найти среднюю из вершин дерева, у которых высоты поддеревьев не равны, а число потомков в правом и левом поддеревьях совпадает. Удалить её (правым удалением), если такая вершина существует. Если у вершины отсутствует некоторое поддерево, то его высоту полагаем равной -1, а число вершин этого поддерева – 0.

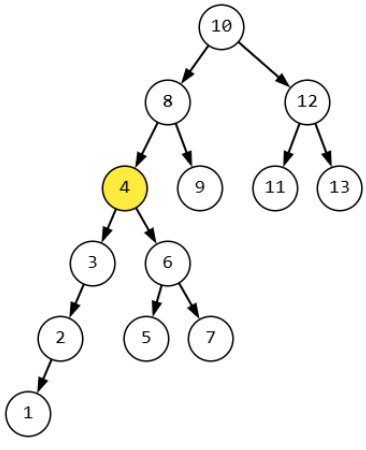
Формат входных данных

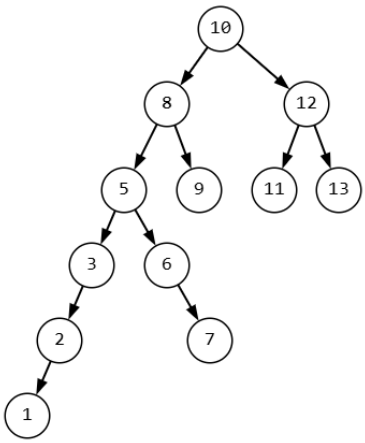
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 8 12 4 9 11 13 3 6 2 5 7 1	10 8 5 3 2 1 6 7 9 12 11 13





Для дерева, приведённого в примере, условию удовлетворяет только одна вершина с ключом 4 (у вершины с ключом 4 высота её левого поддрева не равна высоте её правого поддерева, а число вершин в её левом и правом поддеревьях совпадает (по 3 вершины). Вершина с ключом 4 и будет средней по значению.

В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту. Пусть m – чётно, тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих свойству нет.

Задача 23.

Найти все вершины, через которые проходят полупути длины K между висячими вершинами. Выбрать те, которые находятся на наименьшей глубине, и удалить (левым удалением) из них среднюю по значению, если она существует.

Формат входных данных

Входной файл содержит в первой строке число $K\left(2 \le K \le 2^{10}\right)$, а в последующих строках – последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

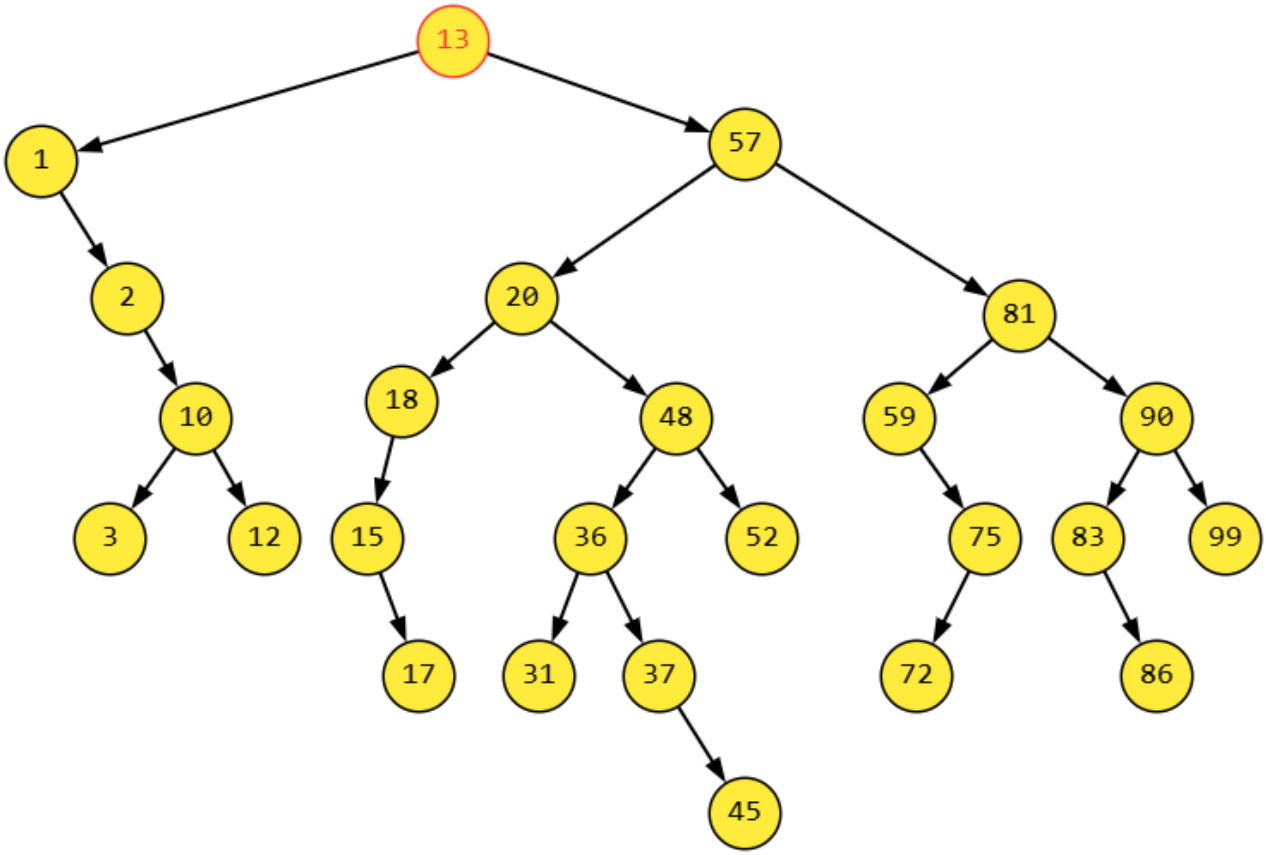
Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
8 13 1 57 20 81 59 48 36 90 83 75 18 86 72 52 31 2 10 37 15 17 99 45 12 3	12 1 2 10 3 57 20 18 15 17 48 36 31 37 45 52 81 59 75 72 90 83 86 99

Замечание.

Содержательная часть каждой задачи (т. е. всё, кроме построения дерева) должна решаться за линейное от числа вершин время. При решении данной задачи допускается линейная зависимость от параметра K в качестве мультипликативной добавки, т. е. общее время – $O\left(n \cdot K\right)$, где n – число вершин в дереве.

Для дерева, приведённого в примере, параметр $K = 8$. Через каждую вершину дерева проходит полупуть длины $K = 8$ между висячими вершинами (листьями дерева).



Вершины с ключами 13 и 57 являются корнями полупутей длины 8 между висячими вершинами.

Полупути длины 8 между висячими вершинами, для которых вершина с ключом 57 является корнем, соединяют вершины:

17 и 72,
17 и 86,
31 и 72,
31 и 86,
45 и 99.

Полупути длины 8 между висячими вершинами, для которых вершина с ключом 13 является корнем, соединяют вершины:

3 и 52,
3 и 99,
12 и 52,
12 и 99.

Наименьшую глубину среди вершин, через которые проходят требуемые полупути длины $K = 8$ между висячими вершинами, имеет вершина с ключом 13, которая и будет удалена (левым удалением).

Задача 24.

Найти все полупути длины H между висячими вершинами, где H – высота дерева.

Выбрать корни этих полупутей, расположенные на максимальной глубине, и удалить (правым удалением) тот из них, который является средним по значению.

Формат входных данных

Входной файл содержит последовательность чисел — ключи вершин в порядке добавления в дерево.

Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

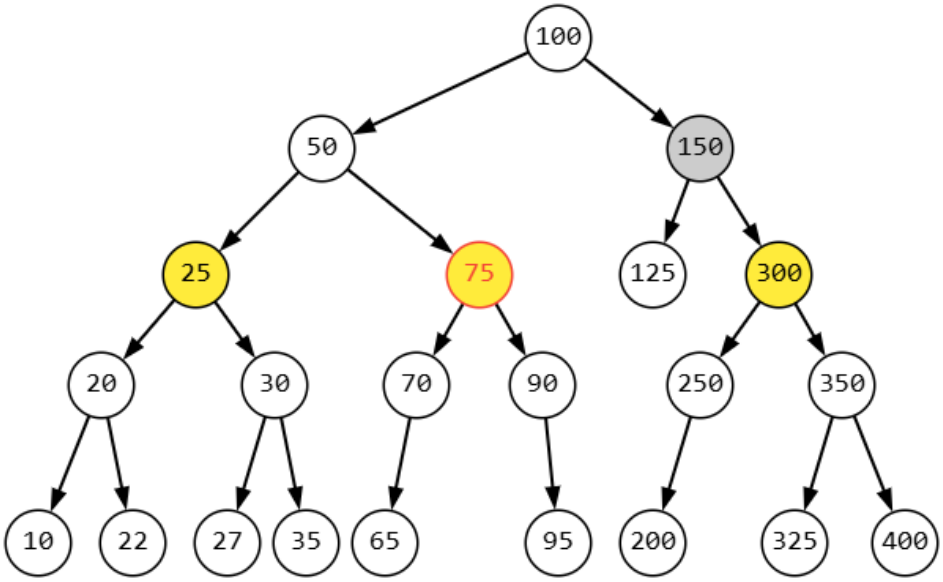
входной файл	*выходной файл*
100 50 150 25 75 125 300 20 30 70 90 250 350 10 22 27 35 65 95 200 325 400	100 50 25 20 10 22 30 27 35 90 70 65 95 150 125 300 250 200 350 325 400

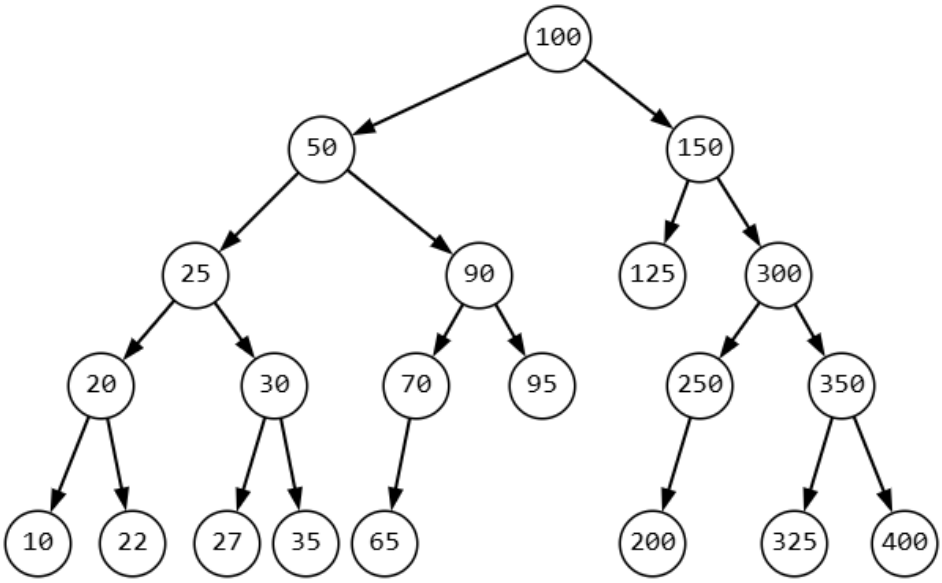
Замечание.

Содержательная часть каждой задачи (т. е. всё, кроме построения дерева) должна решаться за линейное от числа вершин время. При решении данной задачи допускается нелинейная зависимость от числа вершин.

Для дерева, приведённого в примере, высота дерева $H=4$. Вершины с ключами 25 (глубина 2), 75 (глубина 2), 150 (глубина 1) и 300 (глубина 2) являются корнями полупутей длины $H=4$ между висячими вершинами.

Среди вершин с ключами 25, 75, 150 и 300 наибольшую глубину ($=2$) имеют вершины с ключами 25, 75 и 300, а средней по значению среди них является вершина с ключом 75, которая и будет удалена (на рисунке снизу приведено дерево после удаления вершины с ключом 75 правым удалением).





В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту. Пусть m – чётно, тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих свойству нет.

Задача 25.

Найти и удалить (правым удалением), если существует, среднюю по значению из вершин дерева, у которых число потомков в левом поддереве отличается от числа потомков в правом наибольшим образом. Если у вершины отсутствует некоторое поддерево, то число вершин этого поддерева полагаем равным 0.

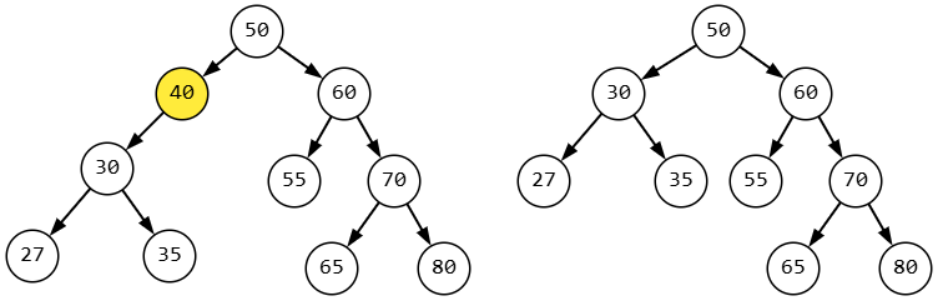
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
50 40 60 30 55 70 27 35 40 65 80 30	50 30 27 35 60 55 70 65 80



Для дерева, приведённого в примере, у вершины с ключом 40 число вершин в поддеревьях отличается наибольшим образом (у вершины 40 число вершин в её левом поддереве равно 3, а в правом – 0, т. е. отличается на $|3-0|=3$). У всех остальных вершин дерева это значение меньше, например, у вершины с ключом 50 число вершин в поддеревьях отличается на $|4-5|=1$, у вершины с ключом 60 число вершин в поддеревьях отличается на $|1-3|=2$. Таким образом имеем только одну вершину с ключом 40, у которой число вершин в поддеревьях отличается наибольшим образом, она будет средней по значению и её надо удалить. В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil+1$ по счёту. Пусть m – чётно, тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих свойству нет.

Задача 26.

Найти и удалить (правым удалением), вершину с наибольшим ключом из вершин дерева, у которых число потомков в правом и левом поддеревьях отличается наибольшим образом.

Если у вершины отсутствует некоторое поддерево, то число вершин этого поддерева полагаем равным 0.

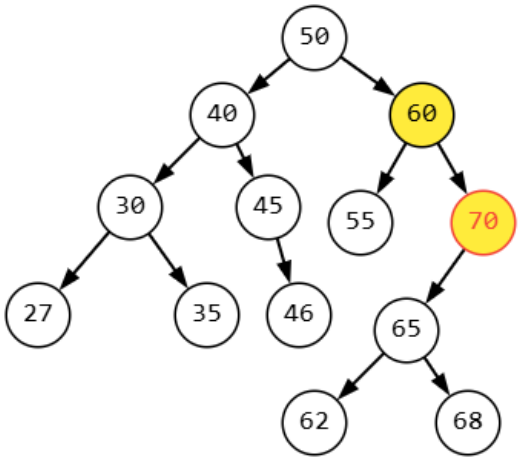
Формат входных данных

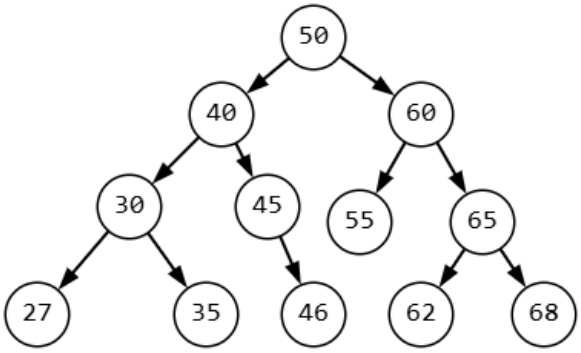
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
50 40 60 30 45 55 70 27 35 46 65 62 68 30	50 40 30 27 35 45 46 60 55 65 62 68





Для дерева, приведённого в примере, у вершин с ключами 60 и 70 число вершин в поддеревьях отличается наибольшим образом (на 3). У всех остальных вершин дерева это значение меньше. Наибольший ключ из вершин с ключами 60 и 70 имеет вершина с ключом 70, которая и будет удалена из дерева.

На рисунке справа изображено дерево после удаления вершины с ключом 70.

Задача 27.

Найти средний по значению, если существует, из листьев дерева и удалить (правым удалением) его отца.

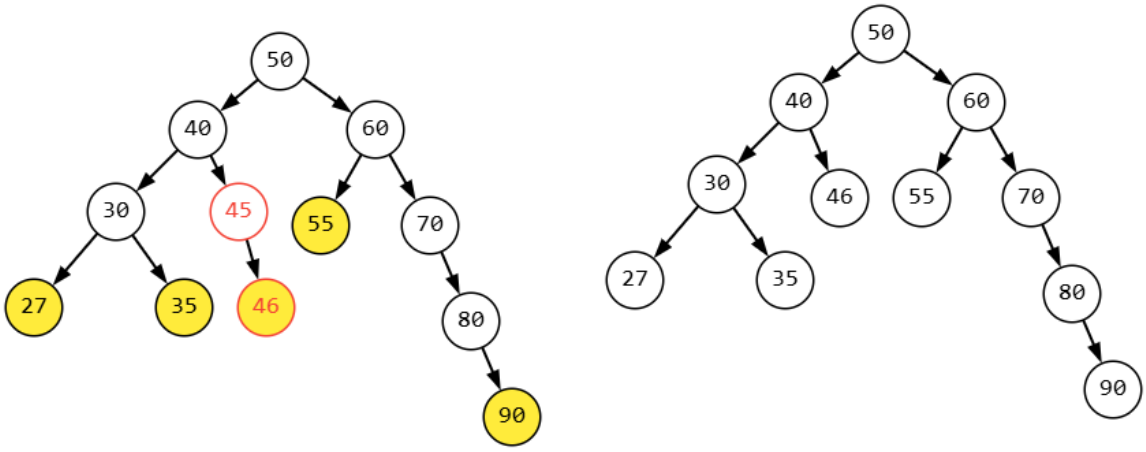
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
50 40 60 30 45 55 70 27 35 46 50 80 40 55 90	50 40 30 27 35 46 60 55 70 80 90



Для дерева, приведённого в примере, листьями являются вершины с ключами 27, 35, 46, 55 и 90 (на рисунке листья дерева выделены жёлтой заливкой). Средний из листьев – вершина с ключом 46. Удаляем отца вершины с ключом 46 – вершину с ключом 45 (на рисунке справа изображено дерево после удаления вершины с ключом 45). В общем случае, предположим, что число листьев равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех листьев, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту. Пусть m – чётно, тогда считаем, что средней по значению вершины среди листьев нет, ничего из дерева удалять не надо.

Задача 28.

Найти в дереве среднюю по значению из вершин, у которых высоты поддеревьев равны. Удалить (правым удалением) данную вершину, если она существует. Листья дерева также рассматриваются в качестве вершин, у которых высоты поддеревьев равны.

Если у вершины отсутствует некоторое поддерево, то его высоту полагаем равной -1.

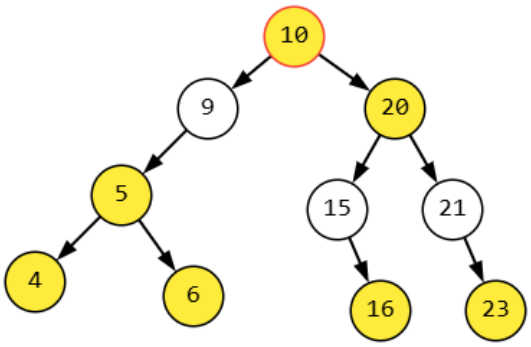
Формат входных данных

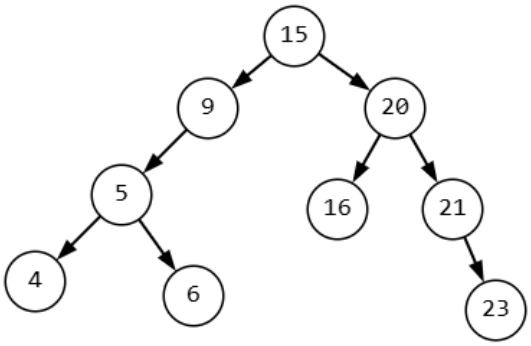
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 9 5 20 15 6 4 21 16 23	15 9 5 4 6 20 16 21 23





Для дерева, приведённого в примере, вершины, у которых высоты поддеревьев равны: 4, 5, 6, 10, 16, 20 и 23. Средняя из них – вершина с ключом 10, которая и будет удалена (правое удаление).

На рисунке справа изображено дерево после удаления вершины с ключом 10.

В общем случае, предположим, что число вершин, удовлетворяющих заданному свойству, равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих заданному свойству, то в упорядоченном массиве искомая вершина будет $\lfloor m/2 \rfloor + 1$ по счёту. Пусть m – чётно, тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих заданному свойству нет, ничего из дерева удалять не надо.

Задача 29.

Найти в дереве среднюю по значению из вершин, у которых число потомков в левом поддереве отличается от числа потомков в правом на два.

Удалить (левым удалением) данную вершину, если она существует.

Если у вершины отсутствует некоторое поддерево, то число вершин этого поддерева полагаем равным 0.

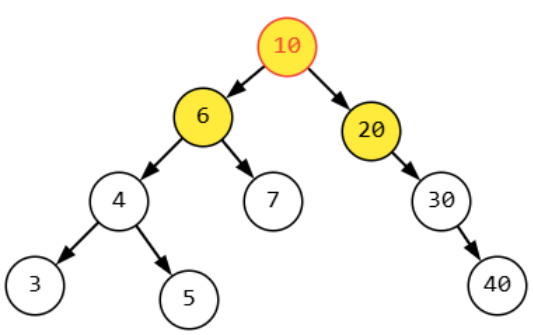
Формат входных данных

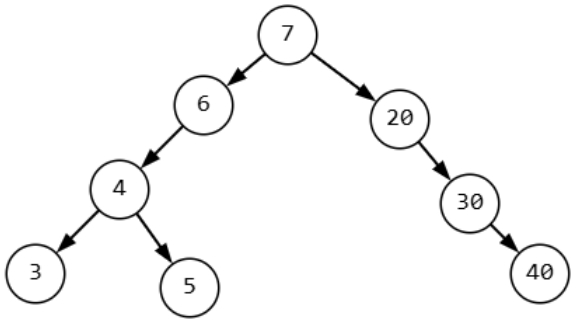
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 6 20 4 7 30 3 5 40	7 6 4 3 5 20 30 40





Для дерева, приведённого в примере, вершины, у которых число потомков в левом поддереве отличается от числа потомков в правом на два: 6, 10 и 20. Средняя из них – вершина с ключом 10, которая и будет удалена (левое удаление).

На рисунке справа изображено дерево после удаления вершины с ключом 10.

В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m.

Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих заданному свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту.

Пусть m – чётно, тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих заданому свойству нет, ничего из дерева удалять не надо.

Задача 30.

Найти среднюю по значению вершину в дереве, у которой высота левого поддерева отличается от высоты правого на два. Удалить (левым удалением) данную вершину, если она существует. Выполнить прямой (левый) обход полученного дерева.

Если у вершины отсутствует некоторое поддерево, то его высоту полагаем равной −1.

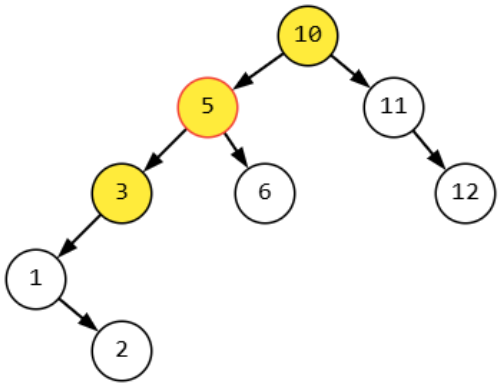
Формат входных данных

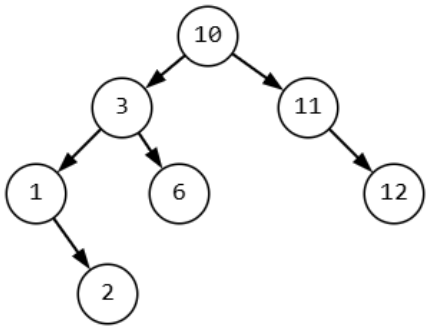
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
10 5 11 12 6 3 1 2	10 3 1 2 6 11 12





Для дерева, приведённого в примере, вершины, у которых высота левого поддерева отличается от высоты правого на два: 3, 5 и 10. Средняя из них – вершина с ключом 5, которая и будет удалена (левое удаление).

На рисунке справа изображено дерево после удаления вершины с ключом 5.

В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m.

Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих заданному свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту.

Пусть m – чётно, тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих заданому свойству нет, ничего из дерева удалять не надо.

Задача 31.

Найти вершины, через которые проходит нечётное число наибольших полупутей, и удалить (правым удалением) ту из них, ключ которой наибольший.

Если в дереве нет вершин, удовлетворяющих нужному свойству, то ничего удалять не требуется. Выполнить прямой (левый) обход полученного дерева.

Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

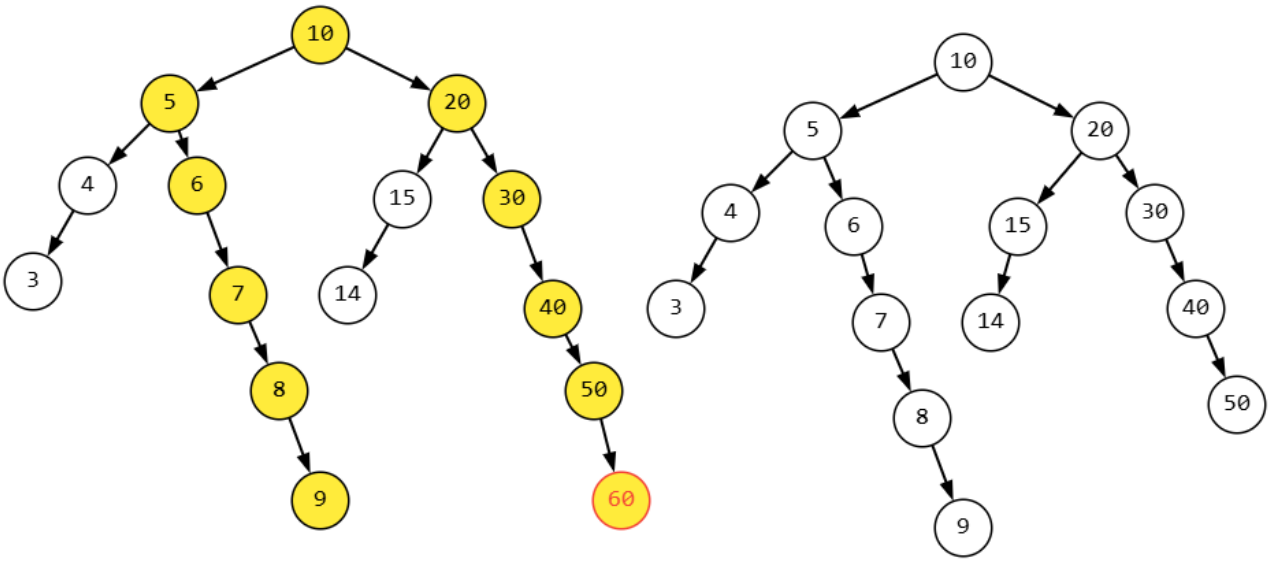
входной файл	*выходной файл*
10 5 20 4 6 15 30 3 7 14 40 8 50 9 60	10 5 4 3 6 7 8 9 20 15 14 30 40 50
11 7 12 3 9 2 5 8 10 1 4 6	11 7 3 2 1 5 4 6 9 8 10

Напомним, что 0 – чётное число.

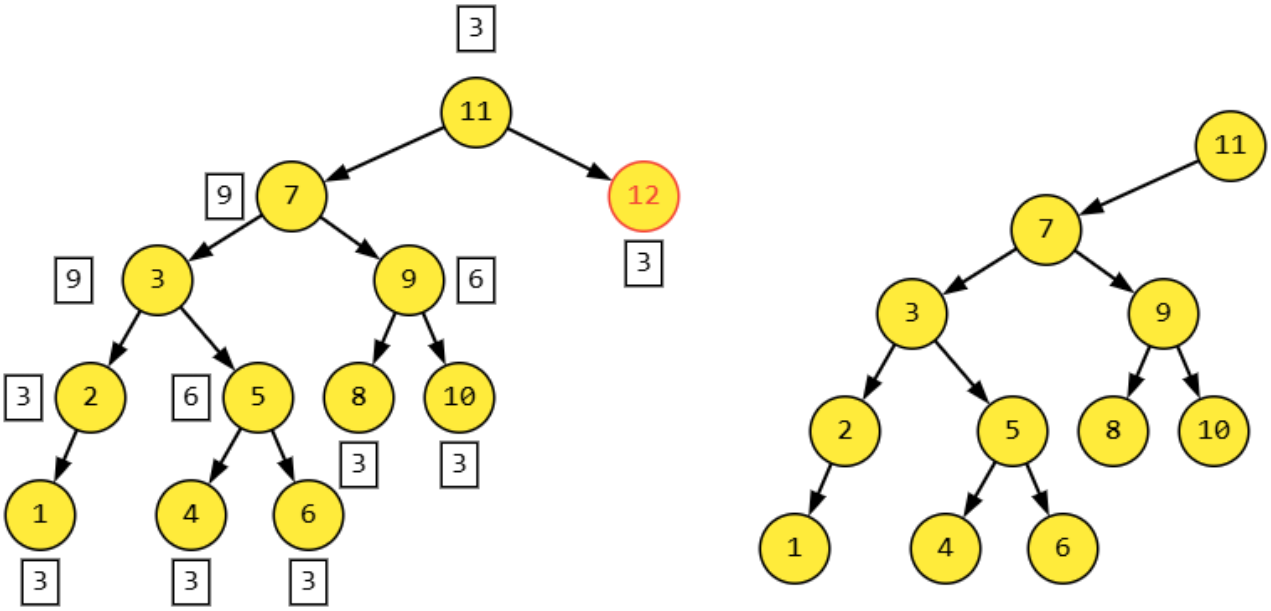
Для дерева, приведённого в примере 1, длина наибольшего полупути равна 10. Вершина с ключом 10 – корень наибольшего полупути длины 10.

На рисунке жёлтой заливкой выделены вершины, через которые проходит наибольший полупуть длины 10. Наибольший ключ среди выделенных вершин имеет вершина с ключом 60, которая и будет удалена.

На рисунке справа изображено дерево после удаления вершины с ключом 60.



Для дерева, приведённого в примере 2, длина наибольшего полупути равна 5. Вершины с ключами 11 и 7 являются корнями полупутей наибольшей длины. Через каждую вершину дерева проходит наибольший полупуть, а число наибольших полупутей, проходящих через каждую вершину, нечётно (на рисунке число наибольших полупутей, проходящих через каждую вершину, указано возле каждой вершины). Наибольший ключ среди выделенных вершин имеет вершина с ключом 12, которая и будет удалена.



На рисунке справа изображено дерево после удаления вершины с ключом 12.

Задача 32.

Найти среднюю по значению вершину дерева из вершин, через которые проходит чётное положительное число наибольших полупутей, и удалить её (правым удалением), если она существует.

Если в дереве нет вершин, удовлетворяющих заданному свойству, то ничего из дерева удалять не надо.

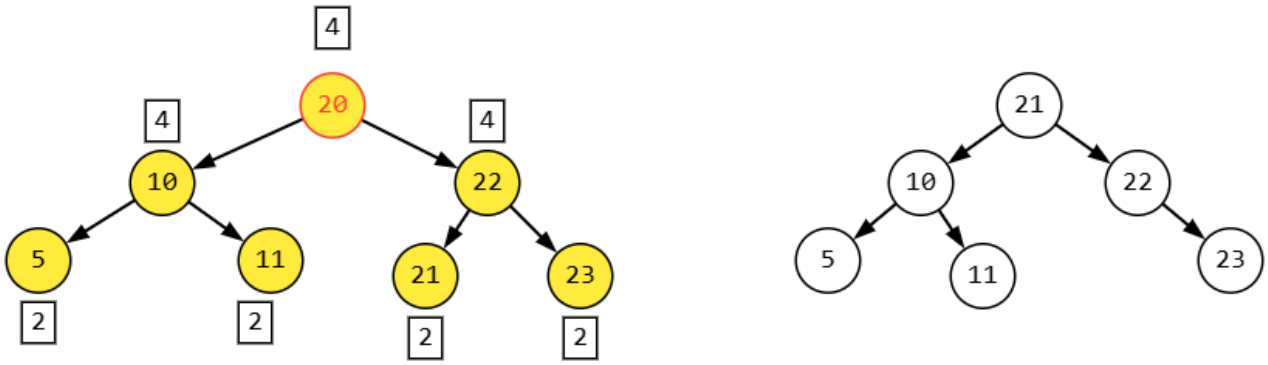
Формат входных данных

Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
20 10 22 5 11 21 23	21 10 5 11 22 23



Для дерева, приведённого в примере, длина наибольшего полупути равна 4. В дереве один корень наибольшего полупути – вершина с ключом 20. Через каждую вершину дерева проходит наибольший полупуть и число наибольших полупутей, проходящих через каждую вершину, чётно (на рисунке это число указано возле каждой вершины). Средняя среди вершин, через которые проходит чётное число полупутей наибольшей длины – вершина с ключом 20, которая и будет удалена правым удалением.

В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих заданному свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту. Пусть m – чётно, тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих заданому свойству нет, ничего из дерева удалять не надо.

Напомним, что 0 не является ни положительным, ни отрицательным числом.

Задача 33.

Найти наибольший полупуть между вершинами дерева, у которого сумма ключей крайних вершин минимальна. Удалить (правым удалением) среднюю по значению вершину этого полупути. Если средней по значению вершины не существует (т. е. число вершин искомого полупути чётно), то ничего из дерева удалять не надо.

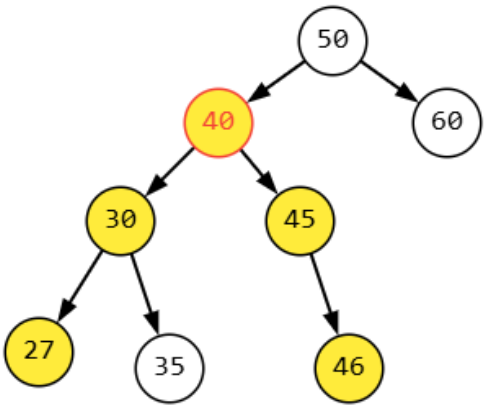
Формат входных данных

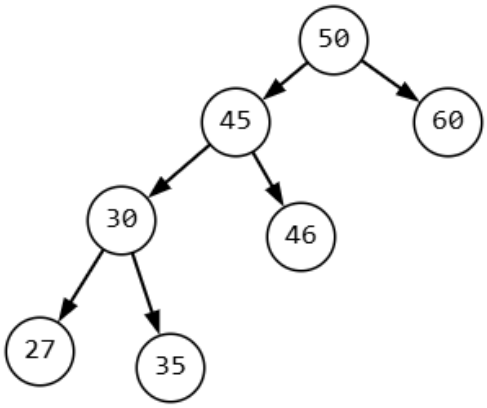
Входной файл содержит последовательность чисел – ключи вершин в порядке добавления в дерево. Гарантируется, что в дереве не менее двух вершин.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

входной файл	*выходной файл*
50 40 60 30 45 27 35 46	50 45 30 27 35 46 60





Для дерева, приведённого в примере, длина наибольшего полупути равна 4. Вершины с ключами 40 и 50 являются корнями полупутей наибольшей длины. Вершины наибольшего полупути с минимальной суммой ключей крайних вершин $(=27+46)$ на рисунке выделены жёлтой заливкой, а средняя из них – вершина с ключом 40, которая и будет удалена (правое удаление).

В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих заданному свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту. Пусть m – чётно, тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих заданому свойству нет, ничего из дерева удалять не надо.

Задача 34.

Найти и удалить (если такая вершина есть, правым удалением) максимальную по значению среди вершин, для которых расстояние до любой другой вершины дерева строго меньше k и через которые не проходят полупути длины k (длина полупути и расстояние между двумя вершинами измеряются числом дуг).

Формат входных данных

В первой строке находится целое число $k\left(0 \le k \le 10^9\right)$.

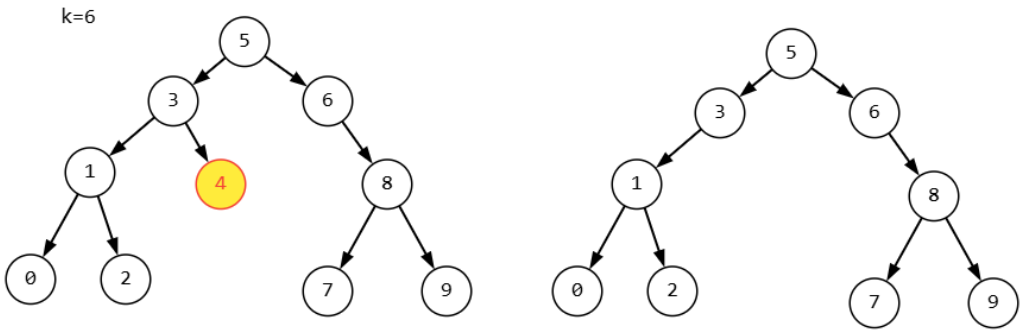
В последующих строках, в каждой по одному числу, находятся ключи вершин исходного дерева (от 0 до $2^{31}-1$) в порядке добавления в дерево.

Число вершин произвольное – от 0 до 1 000.

Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева. Если в результате дерево удалилось полностью, то необходимо вывести в выходной файл сообщение `Empty`.

входной файл	*выходной файл*
6 5 3 1 0 6 4 8 2 7 9	5 3 1 0 2 6 8 7 9
2 10	Empty



Единственная вершина дерева, через которые не проходит полупуть длины $k=6$, это вершина 4, которая и будет удалена. В общем случае, предположим, что число вершин, удовлетворяющих свойству, равно m. Пусть m – нечётно. Тогда, если упорядочить ключи всех вершин, удовлетворяющих свойству, то в упорядоченном массиве искомая вершина будет $\lceil m/2 \rceil + 1$ по счёту. Пусть m – чётно, тогда считаем, что средней по значению вершины среди вершин, удовлетворяющих заданому свойству нет, ничего из дерева удалять не надо.

Задача 35.

Найти максимальную по значению вершину, расстояние от которой до любой другой вершины дерева строго меньше k и через которую не проходят полупути длины k с висячими концевыми вершинами (длина полупути и расстояние между двумя вершинами измеряется в дугах). Если такая вершина есть, то удалить её правым удалением.

Формат входных данных

В первой строке находится целое число $k\left(0 \le k \le 10^9\right)$.

В последующих строках, в каждой по одному числу, находятся ключи вершин исходного дерева (от 0 до $2^{31}-1$) в порядке добавления в дерево.

Число вершин не превосходит 1 000.

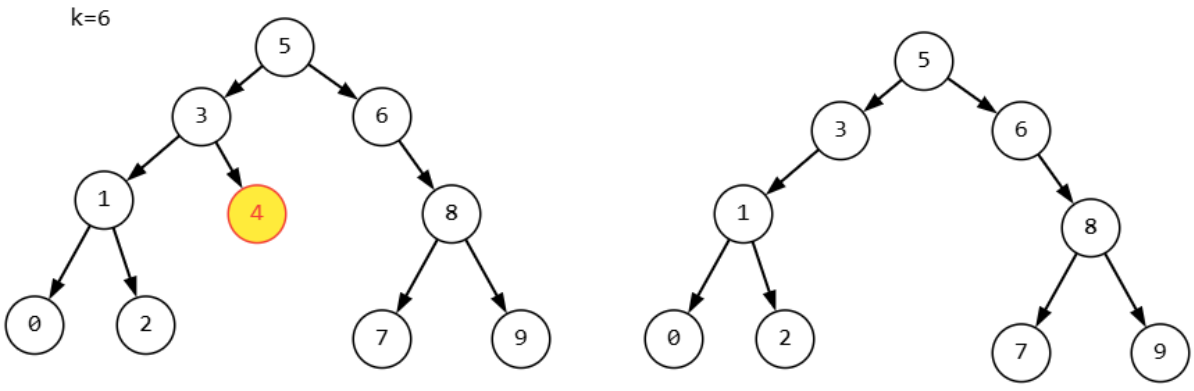
Формат выходных данных

Выходной файл должен содержать последовательность ключей вершин, полученную прямым левым обходом итогового дерева.

Если в результате дерево удалилось полностью, то необходимо вывести в выходной файл сообщение `Empty`.

входной файл	*выходной файл*
6 5 3 1 0 6 4 8 2 7 9	5 3 1 0 2 6 8 7 9
2 10	Empty

Иллюстрация, к примеру 1.



Задача 36.

Во входном файле задано бинарное дерево поиска в прямом левом порядке обхода, в котором для любой его вершины все ключи в её левом поддереве строго меньше её ключа, а все ключи в её правом поддереве не меньше её ключа.

В выходной файл требуется вывести данное дерево в обратном левом и внутреннем левом порядке обхода.

Требуется разработать алгоритм трудоемкости $O(n)$, где n – число вершин дерева.

Формат входных данных

В первой строке входного файла задано число n вершин дерева $(1 \le n \le 100\,000)$.

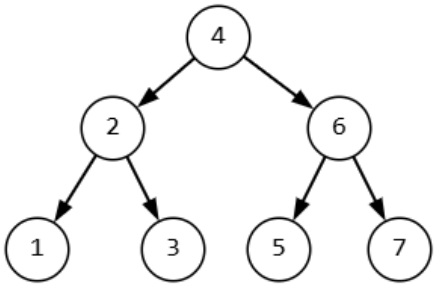
Во второй строке заданы ключи вершин дерева при прямом левом порядке обхода. Значения ключей – целые числа от 0 до 1 000 000 000.

Формат выходных данных

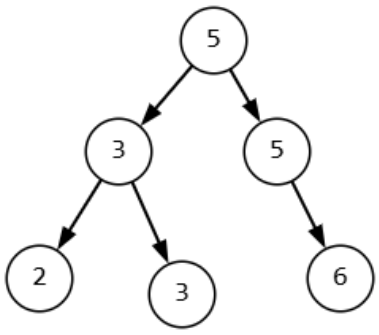
Выведите две строки: в первой выпишите ключи исходного дерева при обратном левом порядке обхода, во второй – при внутреннем левом.

входной файл	*выходной файл*
7 4 2 1 3 6 5 7	1 3 2 5 7 6 4 1 2 3 4 5 6 7
6 5 3 2 3 5 6	2 3 3 6 5 5 2 3 3 5 5 6

Иллюстрация, к примеру 1.



Иллюстрация, к примеру 2. Обратите внимание, что в данной задаче ключи вершин могут повторяться.



Задача 37.

Дано бинарное поисковое дерево, в котором выделены вершины, которые требуется удалить. Необходимо определить, можно ли получить два разных по структуре дерева при разном порядке удаления этих вершин.

Удаление вершины *v*, отличной от корня, происходит следующим образом. Если у вершины *v* нет сыновей, она удаляется вместе с инцидентной ей дугой. Если у вершины *v* ровно один сын, то вершина *v* удаляется, а пара инцидентных ей дуг заменяется дугой от её родителя к её сыну. Если у вершины *v* есть оба сына, она замещается вершиной *u* с наименьшим ключом, превосходящим ключ вершины *v*. Вершина *u*, в свою очередь, удаляется с исходной позиции в дереве по описанным выше правилам.

Два дерева называются *одинаковыми по структуре*, если для каждой пары вершин с одинаковыми ключами их поддеревья содержат одинаковое число вершин.

Формат входных данных

Входной файл содержит несколько тестов.

Для каждого теста в первой строке записано число n вершин дерева $(1 \le n \le 300\,000)$. В i-й из последующих n строк содержится информация об i-й вершине: разделённые пробелом флаг удаления (символ Y или N), ключ вершины (число от 1 до 10^9) и индекс вершины-родителя (для корневой вершины 0, для остальных – число от 1 до $i-1$). Последняя строка содержит число 0, этот случай не нужно обрабатывать. Сумма n по всем тестам в одном файле не превосходит 300 000.

Гарантируется, что ключи вершин попарно различны, корень дерева не удаляется и дерево является бинарным поисковым, т. е. каждая вершина имеет не более одного левого сына (с меньшим ключом) и не более одного правого (с бóльшим ключом), ключи всех вершин в левом поддереве меньше ключа самой вершины, а в правом поддереве – больше.

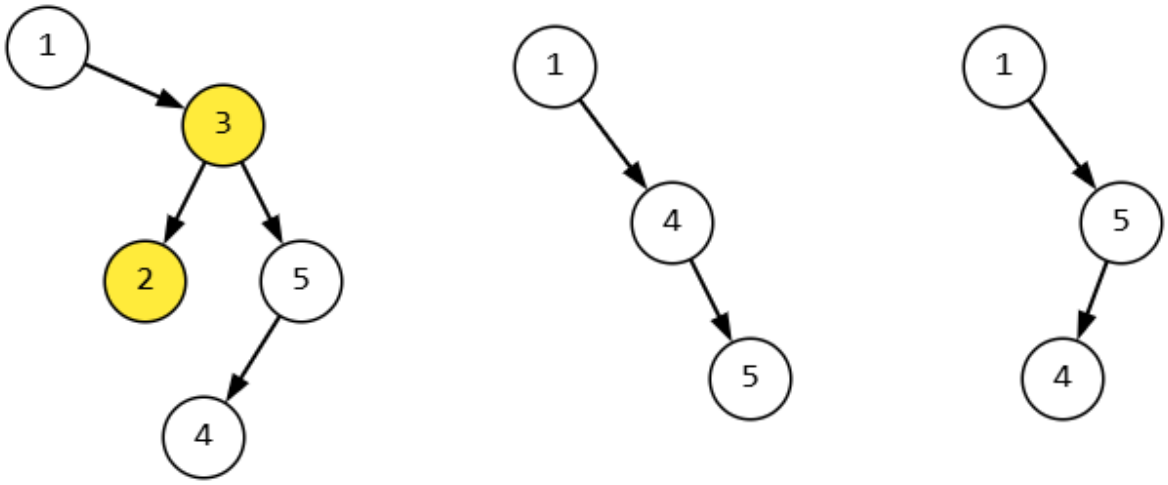
Формат выходных данных

Для каждого теста выведите YES, если структура дерева зависит от порядка удаления вершин, или NO в противном случае.

входной файл	*выходной файл*
5 N 1 0 Y 3 1 Y 2 2 N 5 2 N 4 4 5 N 1 0 Y 3 1 N 2 2 Y 5 2 N 4 4 0	YES NO

Рассмотрим дерево для первого теста (по условию, если у удаляемой вершины есть оба поддерева, то выполняется правое удаление):

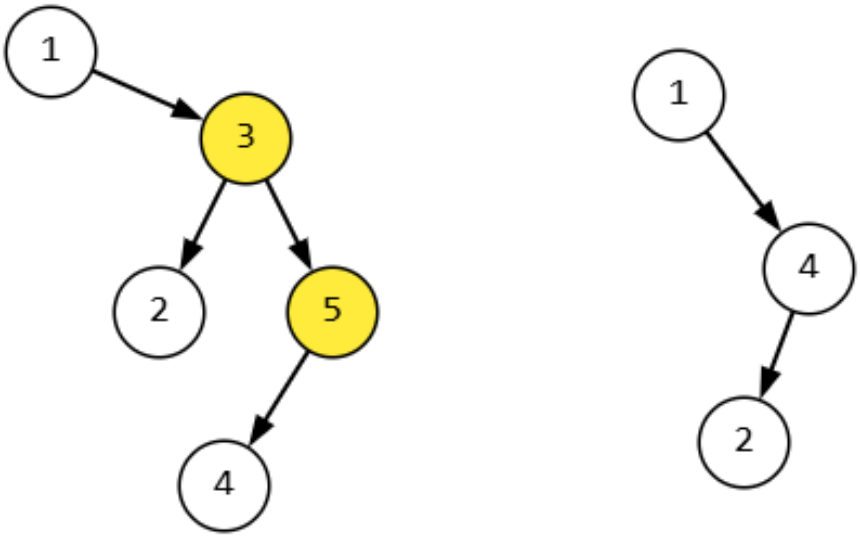
- удаление 3, затем 2;
- удаление 2, затем 3.



Мы видим, что в первом тесте порядок удаления вершин с ключами 3 и 2 влияет на структуру полученного после удаления дерева.

Рассмотрим дерево для второго теста:

- удаление 3, затем 5;
- удаление 5, затем 3.



Во втором тесте результат не зависит от порядка удаления вершин с ключами 3 и 5.

Задача 38.

Определить, можно ли одно дерево получить из второго удалением (правым либо левым) одной вершины (по структуре). Если можно, то указать ту из возможных для удаления вершин, которая имеет наибольший ключ.

Замечание.

Два дерева совпадают по структуре (без учёта ключей), если при внутреннем левом обходе деревьев для каждой пары вершин их поддеревья содержат одинаковое количество вершин. Возможны два случая: из первого дерева удаляется вершина, после чего оно по структуре совпадает со вторым деревом; из второго дерева удаляется вершина, после чего оно по структуре совпадает с первым деревом.

Формат входных данных

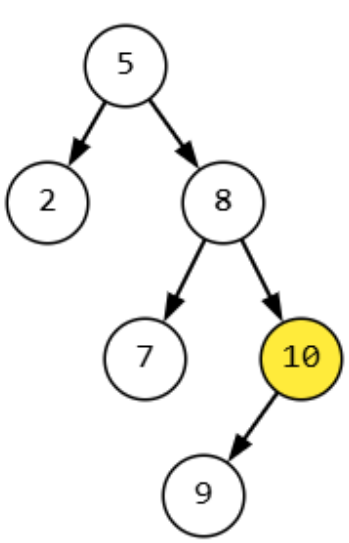
Входной файл содержит последовательность чисел – ключи вершин деревьев в порядке добавления в дерево (по одному числу в строке). Вершины первого дерева отделяются от вершин второго дерева символом *. Для обоих деревьев верно, что после построения все ключи попарно различны. Добавляемые ключи могут повторяться, но для каждого ключа новая вершина вставляется в дерево только один раз – при первом появлении во входных данных.

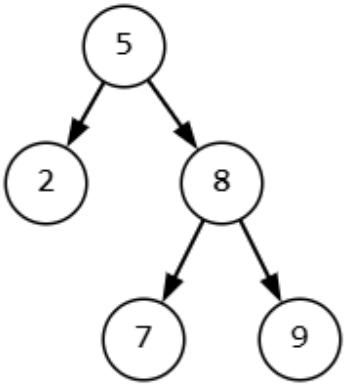
Формат выходных данных

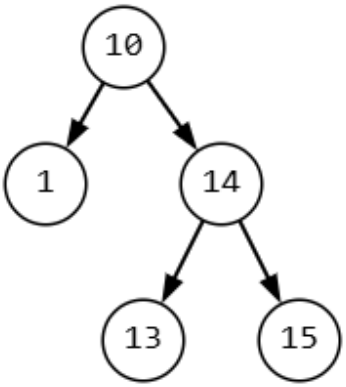
В первой строке выведите сообщение YES или NO (прописными буквами). Если сообщение YES, то во второй строке выведите ключ удаляемой вершины (указывается та из возможных для удаления вершин, которая имеет наибольший ключ).

входной файл	*выходной файл*
10 1 14 13 15 * 5 2 8 7 10 9	YES 10
10 11 9 8 * 10 7 9 8 6 11	NO

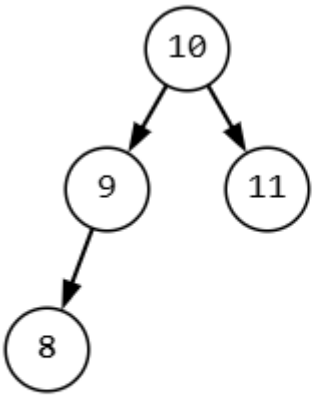
В примере 1 одно дерево можно получить из второго удалением одной вершины (по структуре). Действительно, если из второго дерева удалить вершину с ключом 10, то полученное дерево по структуре совпадет с первым (первое дерево на рисунке приведено снизу).

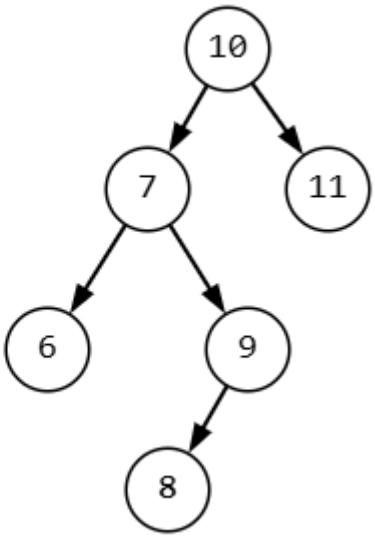






В примере 2 одно дерево нельзя получить из второго удалением одной вершины (по структуре) так как количество вершин в деревьях отличается на 2.





2.3. Указания к решению задач

Задача 1. С помощью обратного обхода для каждой вершины дерева v определим число вершин в дереве с корнем в v и подсчитаем количество вершин, удовлетворяющих нужным требованиям. Если число нужных вершин нечётно, то внутренним обходом найдём среднюю из них и удалим. Выведем дерево.

Задача 2. С помощью обратного обхода для каждой вершины дерева v определим её высоту и подсчитаем количество вершин, удовлетворяющих нужным требованиям. Если число нужных вершин нечётно, то внутренним обходом найдём среднюю из них и удалим. Выведем дерево.

Задача 3. С помощью обратного обхода для каждой вершины дерева v определим число вершин в дереве с корнем в v и подсчитаем количество вершин, удовлетворяющих нужным требованиям. Если число нужных вершин нечётно, то внутренним обходом найдём среднюю из них и удалим. Выведем дерево.

Задача 4. С помощью обратного обхода для каждой вершины дерева v определим её высоту и число потомков в поддереве с корнем в этой вершине. Для спуска на нужный уровень можно использовать поиск в ширину из корня дерева. Поскольку дерево поисковое, то на одном уровне вершины упорядочены слева направо по возрастанию ключа. Другой способ – прямым обходом определим дополнительно для каждой вершины её глубину, затем, зная высоту корня и глубины вершин, любым обходом вычислим уровни вершин по определению. Подсчитаем количество вершин, удовлетворяющих нужным требованиям на заданном уровне, и если число нужных вершин нечётно, то найдём среднюю из них и удалим. Выведем дерево.

Задача 5. С помощью обратного обхода для каждой вершины v определим длину $h'(v)$ кратчайшего пути от вершины v до листа и вес самого лёгкого такого пути (вес пути – сумма ключей вершин этого пути). Найдём корень полупути минимальной положительной длины между листьями с минимальным весом. Для этого рассмотрим каждую вершину v, имеющую двух потомков l и r. Длину искомого полупути с корнем в v можно вычислить по формуле $h'(l)+h'(r)+2$. Необходимо также учесть вес полупути и минимизировать его. В случае равенства длин и весов полупутей выберем корень с меньшим ключом. Если длина искомого полупути чётна, то найдём его центральную вершину и удалим. Выведем дерево.

Задача 6. С помощью обратного обхода для каждой вершины v определим её высоту $h(v)$ и вес самого тяжёлого пути (вес пути – сумма ключей вершин этого пути), выходящего из v и имеющего длину $h(v)$. Найдём наибольшие полупути, а среди них в качестве искомого выберем самый тяжёлый и определим его корень w. Если длина найденного полупути чётна, то найдём центральную вершину этого пути и удалим её. Если вершина w не совпала с центральной вершиной, то удалим и её. Выведем дерево.

Задача 7. С помощью обратного обхода для каждой вершины дерева v определим её высоту. Для спуска на нужный уровень можно использовать поиск в ширину из корня дерева. Поскольку дерево поисковое, то на одном уровне вершины упорядочены слева направо по возрастанию ключа. Другой способ – прямым обходом определим дополнительно для каждой вершины её глубину, затем, зная высоту корня и глубины вершин, любым обходом вычислим уровни вершин по определению. Подсчитаем количество вершин, удовлетворяющих нужным требованиям на заданном уровне, и если число нужных вершин нечётно, то найдём среднюю из них и удалим её. Выведем дерево.

Задача 8. С помощью обратного обхода для каждой вершины дерева определим её высоту. Найдём наибольшие полупути и выберем из них тот, у которого корневая вершина находится на наименьшей глубине (для этого нужно исследовать взаимное расположение корней полупутей наибольшей длины). Удалим корневую вершину искомого наибольшего полупути. Выведем дерево.

Задача 9. С помощью обратного обхода для каждой вершины дерева определим её высоту. Найдём корни наибольших полупутей и выберем из них тот, которому будет принадлежать вторая по значению вершина, через которую проходят наибольшие полупути. Осуществим спуск из найденного корня к нужной вершине и удалим её. Выведем дерево.

Задача 10. С помощью обратного обхода для каждой вершины дерева определим высоту. Найдём корни наибольших полупутей и выберем среди них тот, через который пройдёт наибольшее число полупутей наибольшей длины (для этого достаточно проанализировать взаимное расположение всех наибольших полупутей и выявить их общие фрагменты). Двигаясь из найденного корня, выделим все нужные вершины и удалим их (удаление одной вершины нужно проводить за константное время). Выведем дерево.

Задача 11. С помощью обратного обхода для каждой вершины дерева v определим её высоту $h(v)$. Вычислим корни наибольших полупутей и выберем среди них тот, у которого сумма ключей вершин, которые он соединяет, минимальна (для этого достаточно проанализировать взаимное расположение всех наибольших полупутей и выявить их общие фрагменты). Другим способом решения задачи является поддержка ещё одной метки у каждой вершины v: минимальное значение листа на расстоянии $h(v)$. Удалим корневую вершину найденного полупути. Выведем дерево.

Задача 12. С помощью обратного обхода для каждой вершины дерева v определим её высоту $h(v)$ и минимально возможные значения ключей на расстоянии $h(v)$ и $h(v)-1$, если двигаться из вершины v. Среди полупутей, у которых крайние вершины имеют разное число потомков (т. е. полупути, соединяющие лист – лист, нам не подходят), выберем наибольшие по длине, а среди них в качестве искомых оставим те, у которых сумма ключей крайних вершин минимальна. В случае неоднозначности выбора оставим из выбранных

полупутей те, у которых корневая вершина имеет минимальное ключевое значение. Если длина полупутей чётная и не возникает неоднозначности с выбором удаляемой вершины, то удалим среднюю по значению вершину. Выведем дерево.

Задача 13. С помощью обратного обхода для каждой вершины дерева v определим её высоту $h(v)$ и минимально возможные значения ключей на расстоянии $h(v)$ и $h(v)-1$, если двигаться из вершины v. Среди полупутей, у которых крайние вершины имеют разную высоту (т. е. полупути, соединяющие лист – лист, нам не подходят), выберем наибольшие по длине, а среди них оставим те, у которых сумма ключей крайних вершин минимальна (учтём, что искомый полупуть может являться путём, соединяющим корень дерева и лист). В случае неоднозначности выбора оставим те полупути, у которых корневая вершина имеет минимальное ключевое значение. Если длина полупутей чётная и не возникает неоднозначности с выбором удаляемой вершины, то удалим среднюю по значению вершину. Выведем дерево.

Задача 14. С помощью обратного обхода для каждой вершины дерева v определим её «минимальную» высоту: длину наименьшего пути из вершины v в лист. Найдём длину наименьшего пути между корнем и листьями. С помощью прямого обхода для каждой вершины дерева v определим число вершин, лежащих на пути от корня дерева к вершине v, ключи которых больше (меньше), чем ключ вершины v, и сделаем вывод о том, является ли вершина v средней по значению вершиной некоторого наименьшего пути. Удалим нужные вершины в порядке возрастания ключей. Выведем дерево.

Задача 15. Удалим корни деревьев. Выполним обход (любой) двух деревьев: если в первом поддереве двигаемся влево (вправо), то во втором поддереве будем двигаться вправо (влево). Если в некоторый момент окажется, что движение в одном поддереве удаётся сделать, а ассиметричное ему движение в другом поддереве не удаётся выполнить, то деревья не являются зеркальным отображением друг друга по структуре. Выведем соответствующие сообщения.

Задача 16. Удалим корень дерева. Затем с помощью обратного обхода для каждой вершины дерева v определим её высоту $h(v)$ и количество листьев $l_h(v)$, которые лежат на пути из v на расстоянии $h(v)$. Во время этого же обхода определим длину наибольшего полупути. Выполним прямой обход и определим для каждой вершины v число полупутей наибольшей длины, проходящих через данную вершину. Прямым левым обходом для каждой вершины выведем её ключ и количество полупутей наибольшей длины, проходящих через неё.

Задача 17. С помощью обратного обхода для каждой вершины дерева определим высоту вершины и разность между высотами её поддеревьев. Во время этого же обхода найдём вершины дерева, у которых высота левого поддерева отличается от высоты правого наибольшим образом. Если число

нужных вершин нечётно, то внутренним обходом найдём среднюю из них и удалим её. Выведем дерево.

Задача 18. С помощью обратного обхода для каждой вершины дерева v определим её высоту $h(v)$ и количество листьев $l_h(v)$, которые лежат на пути из v на расстоянии $h(v)$. Во время этого же обхода определим длину наибольшего полупути. Выполним прямой обход и определим для каждой вершины число полупутей наибольшей длины, проходящих через данную вершину. Запомним вершины, через которые проходит чётное число наибольших полупутей, и, если такие вершины есть, то удалим ту из них, ключ которой наименьший. Выведем дерево.

Задача 19. С помощью обратного обхода для каждой вершины дерева определим высоту. Найдём наибольшие полупути и выберем из них тот, у которого корневая вершина имеет наибольшую метку высоты. Выведем дерево.

Задача 20. С помощью обратного обхода для каждой вершины дерева v определим высоту $h(v)$ и минимально возможные значения ключей на расстоянии $h(v)$ и $h(v)-1$, если двигаться из вершины v. Среди полупутей, у которых крайние вершины имеют разный уровень, выберем наибольшие по длине, а среди них оставим те, у которых сумма ключей крайних вершин минимальна (учтём, что искомый полупуть может являться путём, соединяющим корень дерева и лист, а также претендентом может быть полупуть, не являющийся наибольшим). Если длина полупути чётная, то найдём центральную вершину этого полупути и по указанному правилу сделаем её корнем дерева: выполним серию поворотов, после каждого поворота найденная вершина поднимается на один уровень, пока не станет корнем дерева. Выведем сначала сумму ключей крайних вершин найденного полупути, а затем последовательность ключей полученного дерева.

Задача 21. С помощью обратного обхода для каждой вершины дерева v определим высоту и число вершин поддерева с корнем в v. Во время этого же обхода найдём вершины дерева, у которых высоты поддеревьев равны, а число потомков в правом и левом поддеревьях отличается. Если число нужных вершин нечётно, то внутренним обходом найдём среднюю из них и удалим её. Выведем дерево.

Задача 22. С помощью обратного обхода для каждой вершины дерева v определим высоту и число вершин поддерева с корнем в v. Определим число вершин дерева, у которых высоты поддеревьев не равны, а число потомков в правом и левом поддеревьях равны. Если число нужных вершин нечётно, то внутренним обходом найдём среднюю из них и удалим её. Выведем дерево.

Задача 23. С помощью обратного обхода для каждой вершины дерева v определим длины всевозможных путей из вершины v до листьев. Пометим вершины, через которые проходят полупути длины k между листьями. Выберем из помеченных вершин те, которые находятся на наименьшей глубине, и, если

их число нечётно, удалим из них среднюю по значению (для этого можно использовать поиск в ширину из корня дерева, спускаясь по уровням до тех пор, пока не встретим вершину, через которую проходят полупути длины k между листьями). Выведем дерево.

Задача 24. С помощью обратного обхода для каждой вершины дерева v определим высоту и длины всевозможных путей из вершины v до листьев, а с помощью прямого обхода – её глубину. Пусть H – высота дерева. Среди вершин, через которые проходят полупути длины H между листьями, выберем те, которые находятся на наибольшей глубине, и, если их число нечётно, удалим из них среднюю по значению. Выведем дерево.

Задача 25. С помощью обратного обхода для каждой вершины дерева v определим число вершин в поддереве с корнем в v. Во время этого же обхода найдём вершины дерева, у которых число потомков в правом и левом поддеревьях отличается наибольшим образом. Если число нужных вершин нечётно, то внутренним обходом найдём среднюю из них и удалим её. Выведем дерево.

Задача 26. С помощью обратного обхода для каждой вершины дерева v определим число вершин в поддереве с корнем в v. Во время этого же обхода найдём вершины дерева, у которых число потомков в правом и левом поддеревьях отличается наибольшим образом, а среди них выберем ту, ключ которой наибольший. Удалим найденную вершину. Выведем дерево.

Задача 27. Выполним любой обход, подсчитывая число листьев дерева. Если это число нечётно, то внутренним обходом найдём средний лист и удалим его отца. Выведем дерево.

Задача 28. С помощью обратного обхода для каждой вершины дерева определим её высоту и подсчитаем количество вершин, удовлетворяющих нужным требованиям. Если число нужных вершин нечётно, то внутренним обходом найдём среднюю из них и удалим её. Выведем дерево.

Задача 29. С помощью обратного обхода для каждой вершины дерева v определим число вершин дерева с корнем в v и подсчитаем количество вершин, удовлетворяющих нужным требованиям. Если число нужных вершин нечётно, то внутренним обходом найдём среднюю из них и удалим. Выведем дерево.

Задача 30. С помощью обратного обхода для каждой вершины дерева определим её высоту и подсчитаем количество вершин, удовлетворяющих нужным требованиям. Если число нужных вершин нечётно, то внутренним обходом найдём среднюю из них и удалим её. Выведем дерево.

Задача 31. С помощью обратного обхода для каждой вершины дерева v определим её высоту $h(v)$ и количество листьев $l_h(v)$, которые лежат на пути из v на расстоянии $h(v)$. Во время этого же обхода определим длину наибольшего полупути. Выполним прямой обход и определим для каждой вершины число полупутей наибольшей длины, проходящих через данную вершину. Запомним

вершины, через которые проходит нечётное число наибольших полупутей, и, если такие вершины есть, удалим ту из них, ключ которой наибольший. Выведем дерево.

Задача 32. С помощью обратного обхода для каждой вершины дерева v определим её высоту $h(v)$ и количество листьев $l_h(v)$, которые лежат на пути из v на расстоянии $h(v)$. Во время этого же обхода определим длину наибольшего полупути. Выполним прямой обход и определим для каждой вершины число полупутей наибольшей длины, проходящих через данную вершину. Запомним вершины, через которые проходит чётное положительное число наибольших полупутей, и, если такие вершины есть и их количество нечётно, удалим среднюю из них. Выведем дерево.

Задача 33. С помощью обратного обхода для каждой вершины дерева определим её высоту. Вычислим наибольшие полупути и выберем из них тот, у которого сумма ключей конечных вершин минимальна (для этого нужно исследовать взаимное расположение корней полупутей наибольшей длины и сделать вывод о том, какой из наибольших полупутей станет искомым). Удалим среднюю по значению вершину искомого наибольшего полупути (для этого достаточно спуститься из корня полупути, анализируя число поворотов налево и направо). Выведем дерево.

Задача 34. Для каждой вершины v необходимо сначала найти длину наибольшего полупути, который через неё проходит. Эту величину можно вычислить на основании трёх меток, поставленных в соответствие вершине v: длины наибольшего полупути, для которого вершина v является концевой (т. е. наибольшего полупути, идущего из вершины v в вершину, не принадлежащую поддеревьям вершины v), а также высоты левого и правого поддеревьев вершины v.

Предположим, что для вершины v длина наибольшего полупути, который через неё проходит, равна l и $l < k$. Тогда любой другой полупуть, проходящий через эту вершину, также имеет длину меньше, чем k, а значит, полупуть длины k через вершину v точно не пройдёт. Учитывая тот факт, что расстояние от v до любой вершины дерева строго меньше k, получим, что вершина v подходит нам в качестве претендента на удаление.

Предположим, что для вершины v длина наибольшего полупути, который через неё проходит, равна l и $l \geq k$. Значит, существует и полупуть длины k, проходящий через v (отметим, что концевые вершины этого полупути не обязательно листья), т. е. вершина v не подходит нам для удаления, так как одно из условий задачи нарушается.

Задача 35. Как и в предыдущей задаче, для каждой вершины v нужно найти длину l наибольшего полупути, который через неё проходит.

Если $l < k$, то вершина v является претендентом на удаление.

Если $l \geq k$, то мы не можем сразу исключить вершину v из рассмотрения, как это было сделано в предыдущей задаче. Необходимо проверить, существует

ли полупуть длины k с висячими концевыми вершинами, проходящий через вершину v, а также проанализировать расстояние от вершины v до всех остальных вершин дерева.

Рассмотрим, как понять, проходит ли через данную вершину v полупуть длины k с висячими концевыми вершинами.

Пусть L — множество длин путей от вершины v до листьев левого поддерева вершины v.

Аналогично определим множество R – множество расстояний до листьев правого поддерева.

Введём множество U – множество расстояний от вершины v до всех остальных листьев дерева, не лежащих в поддереве с корнем в вершине v (см. пример на рисунке 2.1: множество L включает расстояния до листьев 1, 3, 5, множество R – до листа 7, во множестве U хранятся расстояния до листьев 9 и 11).

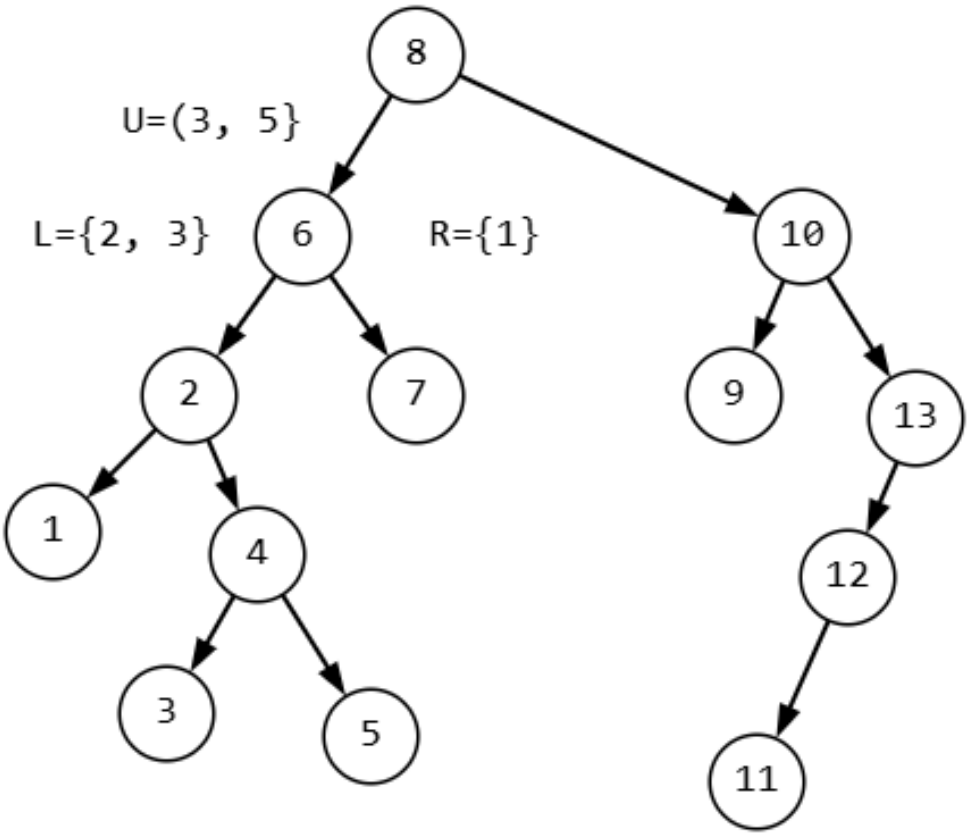


Рисунок 2.1 – Задача 35.

Пусть (X,Y)– одна из пар множеств (L,R), (L,U), (R,U).

Если существуют такие числа $x \in X$ и $y \in Y$, что

$$x + y = k,$$

то через вершину v проходит полупуть длины k, соединяющий два листа.

Множества L, R считаются одним обратным обходом, затем при следующем обходе множество U динамически пересчитывается на основании имеющейся информации.

Задача 36.

Пусть $a_1, a_2, \ldots, a_i, \ldots, a_n$ последовательность ключей дерева в прямом порядке обхода.

Тогда a_1 – корень дерева.

Пусть i – максимальный индекс такой, что $a_i < a_1$, тогда элементы:

$$a_1, a_2, \ldots, a_i$$

составляют левое поддерево дерева T, а элементы:

$$a_{i+1}, \ldots, a_n$$

составляют, соответственно, правое поддерево.

Причём, в каждом из списков, элементы также выписаны в прямом порядке обхода (это означает то, что элементы a_2 и a_{i+1} – являются корнями соответствующих поддеревьев, a_2 – левого, а a_{i+1} – правого).

Для реализации описанного процесса построения дерева можно использовать структуру данных *стек*.

1) Последовательно просматриваем элементы последовательности. Изначально в стек добавляем первый элемент последовательности a_1.

2) Предположим, что текущий элемент последовательности a_i, а на вершине стека расположен элемент a_j.

Тогда

– если $a_i < a_j$, то полагаем текущий элемент a_i левым сыном a_j и заносим текущий элемент a_i в стек;

– если $a_i \geq a_j$, то удаляем из стека все элементы, которые меньше либо равны текущего элемента a_i, полагаем текущий элемент a_i правым сыном последнего удалённого из стека элемента, а сам элемент a_i добавляем в стек.

Задача 38. Сначала для каждой вершины v обоих деревьев T^1 и T^2 подсчитаем обратным обходом, сколько вершин в поддереве с корнем v.

Если число вершин в деревьях T^1 и T^2 отличается более, чем на 1, то ответ No.

Предположим, что число вершин в деревьях T^1 и T^2 отличается ровно на 1. Пусть $c\left(T_1^1\right), c\left(T_2^1\right)$ – число вершин в поддеревьях дерева T^1, а $c\left(T_1^2\right), c\left(T_2^2\right)$ – дерева T^2. Тогда, для возможности получения положительного ответа нужно, чтобы, например, $c\left(T_1^1\right) = c\left(T_1^2\right)$, в этом случае мы переходим симметрично в поддеревья T_2^1 и T_2^2 (так как удалять вершину надо из дерева с большим числом вершин) и повторяем рассуждения.

Анализируем возможные варианты удаления и смотрим, равны ли после этого деревья.

3. РАЗДЕЛ КОНТРОЛЯ ЗНАНИЙ

3.1. Набор тестовых заданий для проверки теоретических знаний

По всем разделам учебной дисциплины Алгоритмы и структуры данных» для проверки теоретических знаний в InsightRunner разработаны тестовые задания. Тест может включать задания разных видов: вопросы, где нужно выбрать один правильный ответ из предложенных; вопросы, где правильных ответов несколько и требуется отметить множество вариантов; вопросы, где нужно ввести текстовый ответ. Отвечать на вопросы можно в любом порядке, все ответы автоматически сохраняются. Система ведёт учет времени.

Отличительной особенностью InsightRunner является использование автоматизированных генераторов. Это позволяет создать большую базу вопросов для тестов. Каждый студент получает уникальный набор заданий, тем самым исключается возможность списывания.

Студенты в Образовательной платформе InsightRunner на каждом практическом занятии выполняют тестовые задания, преподаватель проводит разбор допущенных ошибок. В рамках самоконтроля студенты имеют возможность выполнять тестовые задания в удобное для них время из дома/общежития, видеть правильно или нет они ответили, а также ознакомиться с правильными вариантами ответов.

Учебная дисциплина «Алгоритмы и структуры данных» завершается прохождением итогового теста, который, как правило, содержит 20 вопросов по основным разделам дисциплины и рассчитан на 30 минут. При прохождении итогового теста студентам не разрешено пользоваться вспомогательными материалами и интернет-источниками.

Примеры тестовых заданий.

Тестовое задание 1.

Задан массив $H = (2,4,2,0,4)$. Будем считать, что индексы начинаются с единицы. Строится помеченное корневое дерево T из пяти вершин (вершины дерева занумерованы целыми числами от 1 до 5, ключ вершины совпадает с её номером). Дерево строится таким образом, что массив H является каноническим представлением дерева T ($H[i]$ – отец вершины i; если $H[i] = 0$, то вершина i является корнем дерева).

Определить, какие из следующих утверждений являются верными?

☒ Можно восстановить дерево T так, чтобы оно было АВЛ-деревом.

☒ Можно восстановить дерево T так, чтобы оно было 2- сбалансированным (по высоте).

☐ Можно восстановить дерево T так, чтобы оно было идеально сбалансированным (по количеству вершин).

☒ Можно восстановить дерево T так, чтобы оно было полным бинарным.

☒ Можно восстановить дерево T так, чтобы оно было бинарным поисковым.

☐ Нет верных утверждений.

Решение.

Нарисуем один из возможный вариантов корневого дерева, у которого каноническое представление задаётся следующим массивом:

$$H = (2,4,2,0,4).$$

Для того, чтобы у дерева сохранить заданный канонический способ задания, можно менять местами у вершины *v* её левого и правого сына, но изменять отца у вершины *v* нельзя. На рисунке 3.1. приведён один из возможных вариантов дерева, для которого канонический способ задания: $H = (2,4,2,0,4)$.

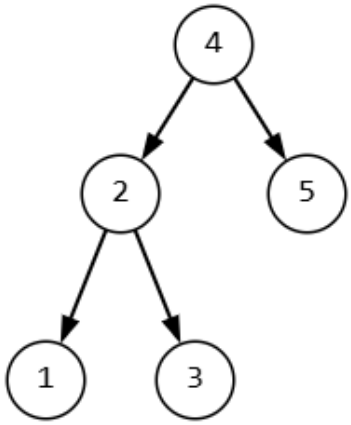


Рисунок 3.1 – Тестовое задание 1.

Дерево является:

– бинарным (у каждой вершины не более 2-х сыновей);

– поисковым (для каждой вершины *v* ключи вершин в её правом поддереве строго меньше ключа вершины *v*, а в правом – строго больше);

– сбалансированным по высоте (для каждой вершины *v* высоты её поддеревьев отличаются не более, чем на 1);

– АВЛ-деревом (бинарное, поисковое и сбалансированное);

– полным бинарным (бинарное дерево, которое заполнено вершинами по слоям сверху вниз, пока один слой не заполнен к следующему не переходят, на одном слое идёт заполение вершинами слева направо, последний слой заполнять полностью вершинами не требуется);

– так как дерево является сбалансированным по высоте $(k=1)$, то будет верным и утверждение о том, что дерево является $k \geq 1$ (2, 3, 4 и т. д.) сбалансированным по высоте;

– для вершины с ключом 4 количество вершин в поддеревьях отличается на 2, поэтому дерево не является идеально сбалансированным по количеству вершин (для идеальной сбалансированности требуется, чтобы для каждой вершины количество вершин в её поддеревьях отличалось не более, чем на 1);

– дерево является 2-идеально сбалансированным по числу вершин; верными будут являться и утверждения о том, что дерево является 3-идеально сбалансированным по количеству вершин, 4-идеально сбалансированным по количеству вершин, 5-идеально сбалансированным по количеству вершин и т. д.

Тестовое задание 2.

Если для последовательности элементов (5, 1, 3, 2, 4) построить АВЛ-дерево последовательным добавлением нового элемента с поддержкой инварианта сбалансированности после каждого добавления, то какая вершина будет являться корнем дерева и какие повороты (RR, LL, LR, RL) будут выполнены при этом? Если *x* – некоторая вершина дерева, то в её левом поддереве ключи вершин меньше, чем ключ вершины *x*, а в правом – больше (ключ вершины совпадает с её номером).

- ☐ 1
- ☐ 2
- ☒ 3
- ☐ 4
- ☐ 5
- ☐ RL
- ☐ RR
- ☐ LL
- ☒ LR
- ☐ повороты не выполнялись

Решение.

Построим АВЛ-дерево, последовательно добавляя элементы. Каждый раз, когда при добавлении элемента происходит разбалансировка, выполним соответствующий поворот (RR, LL, LR, RL).

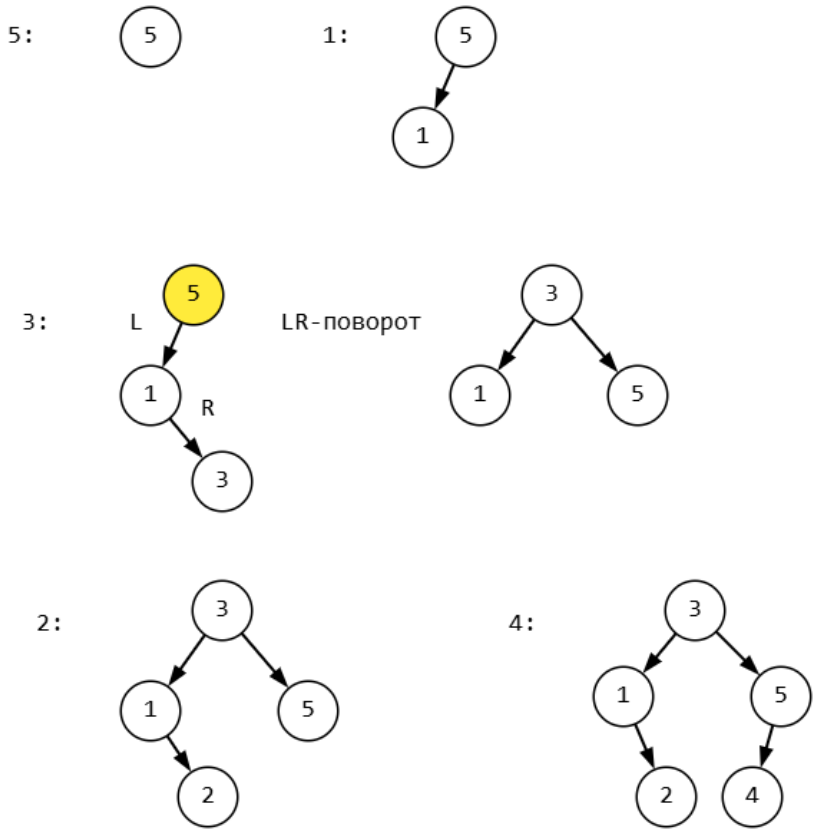


Рисунок 3.2 – Тестовое задание 2.

При добавлении вершины с ключом 3 у вершины с ключом 5 произошла разбалансировка. У вершины с ключом 5 высота её левого поддерева равна 1, а правого поддерева у вершины с ключом 5 нет (полагаем высоту несуществующего поддерева равной минус единице). Для вершины с ключом 5 выполняем LR-поворот для восстановления свойства сбалансированности.

После добавления всех вершин корнем дерева становится вершина с ключом 3, для восстановления балансировки был выполнен один LR-поворот (рисунок 3.2).

Тестовое задание 3.

В пустое бинарное поисковое дерево были последовательно добавлены вершины с ключами 3, 9, 19, 1, 12. В каком порядке будут посещены вершины при прямом правом обходе построенного дерева?

Определить, какие из следующих утверждений являются верными.

- ☐ 1, 12, 19, 9, 3
- ☐ 19, 12, 9, 3, 1
- ☐ 3, 1, 9, 19, 12
- ☐ 1, 3, 9, 12, 19
- ☐ 12, 19, 9, 1, 3
- ☒ 3, 9, 19, 12, 1

Решение.

Построим бинарное поисковое дерево и выполним требуемый обход.

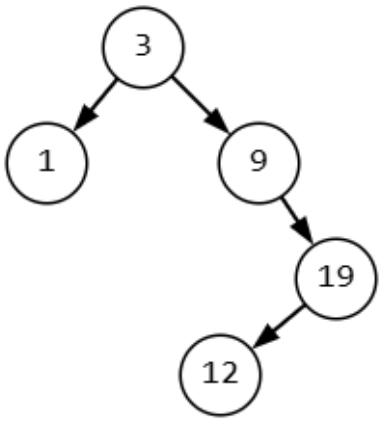


Рисунок 3.3 – Тестовое задание 3.

Обратный порядок обхода заключается в том, что корень дерева посещается после его поддеревьев. Если при обратном обходе сначала осуществляется переход в левое (правое) поддерево корня, то обход называется обратным левым (правым).

Порядок посещения вершин дерева, изображённого на рисунке 3.3, при прямом правом обходе:

3, 9, 19, 12, 1.

3.2. Набор тестов для проверки работоспособности программ

В рамках учебной дисциплины «Алгоритмы и структуры данных» студенты решают алгоритмические задачи. Решением задачи является программа, написанная на некотором высокоуровневом языке программирования.

Задачи подразбиты на четыре темы, которые покрывают все разделы учебной дисциплины:

1) рекуррентные соотношения;
2) деревья поиска;
3) структуры данных и графы;
4) алгоритмы на графах;
5) строковые алгоритмы.

Набор разнообразных задач, которые предлагаются для решения в InsightRunner, формируется преподавателями и постоянно расширяется. В настоящий момент в рамках учебной дисциплины «Алгоритмы и структуры данных» в системе InsightRunner разработано более 350 задач разного уровня сложности.

Задачи делятся на две категории: общие и индивидуальные.

Общие задачи выполняют все учащиеся (в рамках учебной дисциплины каждый учащийся должен выполнить не менее 30 общих задач). Алгоритмы решения общих задач подробно разбираются на лекциях, обосновывается корректность алгоритмов, приводятся псевдокоды.

Цель выполнения общих задач – отработать базовые алгоритмические навыки. Считается, что любой выпускник ФПМИ должен уметь реализовывать основные алгоритмы.

Индивидуальные задачи требуют творческого подхода, алгоритмических и программистских навыков, а также развивают креативное мышление.

Каждый студент в группе по каждой теме получает 1–2 индивидуальные задачи, уровень сложности которой от 4 до 10 баллов (студент может попросить преподавателя назначить ему дополнительные задачи, чтобы повысить свой уровень алгоритмической подготовки). Студенты могут по каждой теме заявить уровень сложности, на который они хотят решать задачи.

Все задачи хранятся в системе InsightRunner. Каждая задача состоит из двух важнейших частей: из текста условия и набора тестов.

Условия задач в системе по традиции имеют единую структуру и стиль оформления. В условие обычно входят следующие элементы:

– название задачи;

– уровень сложности по 10-бальной шкале;

– ограничения по времени и памяти, в которые должна уложиться программа;

– легенда (сказка) или формальная постановка;

– описание формата входных данных и ограничения на входные данные;

– описание формата выходных данных;

– примеры (примеры из условия задачи являются первыми тестами при тестировании и доступны студентам для просмотра в процессе тестирования).

Студенты могут читать условия всех задач, находящихся в системе.

Тесты к задаче. Под тестом для задачи понимается набор входных и выходных данных. Тесты могут выглядеть следующим образом:

Тест №1: вход 1, выход 1

Тест №2: вход 2, выход 2

Тест №3: вход 3, выход 5

…

Тест №16: вход 10 000 000, выход 640 540 120

Тесты составляются преподавателями. Студентам содержимое тестов недоступно (кроме примеров из условия). Чем больше тестов создано для задачи, тем больше неправильных решений этими тестами отсекается.

Наличие заготовленных тестов позволяет автоматически проверить правильность решения. Для этого нужно запускать программу, передавая ей входные данные, и сравнивать тот ответ, который выдает программа, с правильным ответом, который хранится в базе. Эту работу и делает Образовательная платформа InsightRunner.

Всего на сегодняшний день по всем задачам в системе хранится более 6,5 тыс. тестов.

3.3. Средства диагностики

Студент пишет программу, которая решает поставленную перед ним задачу. Затем он отправляет свой код в InsightRunner через специальную форму. На сервере решение проверяется на наборе тестов. Через некоторое время (обычно несколько секунд) студенту выдается вердикт. Например, «Неправильный ответ (5)» означает, что на первых четырех тестах программа отработала верно, а на пятом тесте выдала неверный ответ. Вердикт «Принято» значит, что решение прошло все тесты успешно. Студент может ознакомиться с протоколом тестирования.

Образовательная платформа InsightRunner при этом предоставляет дружественный интерфейс преподавателю курса для назначения студентам задач (индивидуальных, общих, дополнительных, штрафных), установки крайних сроков выполнения задания (для планирования равномерной работы студентов в семестре), отслеживания прогресса, расчета итоговых оценок по курсу (Журнал, Ведомость).

В системе InsightRunner есть и такие функционалы, как «Сообщения» (система обмена сообщениями между преподавателем и студентами), «Электронная очередь» (позволяет организовывать ход практического занятия: студенты по мере готовности записываются в очередь для беседы путём нажатия

кнопки, а преподаватель на паре беседует со студентами в порядке очереди; автоматически собирается статистика о том, когда студент работал с преподавателем).

Преподаватель также имеет возможность задать компиляторы, на которых студент может выполнять задания C# 8 (.NET Core 3.1), GNU C++17 7.3.0 (MinGW), GNU C++20 11.2.0 (MinGW) x64, Java 8, Kotlin 1.5, Microsoft Visual C++ 15.9 (2017), Microsoft Visual C++ 15.9 (2017) x64, PyPy3.6 v7.2.0, Python 3.8 + NumPy и др.

С целью предотвращения списывания в системе InsightRunner функционирует *модуль проверки решений на плагиат*, результаты работы которого доступны только преподавателю. Для каждого нового решения, которое прошло тесты, система ищет похожие решения в своей базе. Похожесть оценивается в процентах. Преподаватель затем окончательно решает, плагиат это или нет, и определяет способ наказания для студента (например, назначить штрафную задачу по данной теме).

Компьютерная система не заменяет преподавателя, а помогает ему: автоматизируется рутинная работа по проверке решений, тем самым у преподавателя появляется возможность уделять больше внимания вопросам алгоритмизации и проводить занятия в виде эвристических диалогов. Защита студентом алгоритма решения индивидуальной задачи проводится в вопросно-ответной форме: студенту даются наводящие, уточняющие вопросы, которые вытекают один из другого и, в совокупности, приводят учащегося к разработке эффективного алгоритма.

Система InsightRunner также используется для организации соревнований по программированию и для проведения тренировок. Знания, полученные в рамках учебных дисциплин по теории алгоритмов, играют ключевую роль при решении олимпиадных задач. Соревнуясь, студенты учатся придумывать сложные алгоритмы и писать программы без ошибок в условиях стресса и ограниченного времени.

Образовательная платформа InsightRunner функционирует на факультете прикладной математики и информатики Белорусского государственного университета с 2003 года и доказала свою эффективность на практике. Возможность круглосуточной самостоятельной работы студентов на базе образовательной платформы InsightRunner способствует получению высоких результатов на соревнованиях в области алгоритмизации и спортивного программирования и повышению позиции БГУ в мировых рейтингах.

Результатом внедрения платформы в учебный процесс стала повышенная заинтересованность в практических занятиях по учебной дисциплине «Алгоритмы и структуры данных» как со стороны учащихся, так и со стороны преподавателей [4, 9].

4. ВСПОМОГАТЕЛЬНЫЙ РАЗДЕЛ

4.1. Рекомендуемая литература

Основная

1. Алгоритмы и структуры данных : электронный учебно-методический комплекс для специальностей: 6-05-0533-09 «Прикладная математика», 6-05-0533-10 «Информатика», 6-05-0533-11 «Прикладная информатика», 1-31 03 04 «Информатика», направлений специальностей: 1-31 03 03-01 «Прикладная математика (научно-производственная деятельность)», 1-31 03 07-01 «Прикладная информатика (программное обеспечение компьютерных систем)». В 3 ч. Ч. 1 / Е. П. Соболевская [и др.] : БГУ, Фак. прикладной математики и информатики, Каф. дискретной математики и алгоритмики. – Минск : БГУ, 2023. – 183 с. : ил. – Библиогр.: с. 182–183. [Электронный ресурс]. – Режим доступа: https://elib.bsu.by/handle/123456789/309510. – Дата доступа: 06.01.2025.

2. Сборник задач по теории алгоритмов. Структуры данных: учеб. -метод. пособие / С. А. Соболь [и др.] – Минск: БГУ, 2020. – 159 с. [Электронный ресурс]. – Режим доступа: https://elib.bsu.by/handle/123456789/255033. – Дата доступа: 06.01.2025.

Дополнительная

1. Алгоритмы: построение и анализ / Т. Кормен [и др.] – М.: Вильямс, 2005. – 1296 с.

2. Котов, В. М. Алгоритмы и структуры данных : учеб. пособие / В. М. Котов, Е. П. Соболевская, А. А. Толстиков. – Минск : БГУ, 2011. – 267 с. – (Классическое университетское издание).

3. Котов, В. М. Теория алгоритмов. Организация перебора и приближенные алгоритмы: учеб. -метод. пособие / В. М. Котов, Е. П. Соболевская, Г. П. Волчкова. – Минск: БГУ, 2022. – 151 с.

4. Опыт использования образовательной платформы Insight Runner на факультете прикладной математики и информатики Белорусского государственного университета. Роль университетского образования и науки в современном обществе : материалы междунар. науч. конф., Минск, 26 – 27 февр. 2019 г. / редкол.: А. Д. Король (пред.) [и др.]. – Минск : БГУ, 2019. – С. 263 – 267. [Электронный ресурс]. – Режим доступа: https://elib.bsu.by/handle/123456789/231914. – Дата доступа: 06.01.2025.

5. Пападимитриу, Х. Комбинаторная оптимизация. Алгоритмы и сложность / Х. Пападимитриу, К. Стайглиц – М.: Мир, 1985. – 512 с.

6. Рейнтгольд, Э. Комбинаторные алгоритмы. Теория и практика / Э. Рейнтгольд., Ю. Нивергельт, Н. Део – М.: Мир, 1980. – 466 с.

7. Сборник задач по теории алгоритмов : учеб.-метод. пособие / В. М. Котов [и др.] – Минск : БГУ, 2017. – 183с. [Электронный ресурс]. – Режим доступа: http://elib.bsu.by/handle/123456789/181529. – Дата доступа: 02.01.2025.

8. Сборник задач по теории алгоритмов. Организация перебора и приближенные алгоритмы : электронный учебно-методический комплекс для специальности: 1-31 03 04 «Информатика» / В. М. Котов, Е П. Соболевская, Г. П. Волчкова ; БГУ, Фак. прикладной математики и информатики, Каф. дискретной математики и алгоритмики. – Минск : БГУ, 2021. – 144 с. : ил. – Библиогр.: с. 143–144. [Электронный ресурс]. – Режим доступа: https://elib.bsu.by/handle/123456789/272717. – Дата доступа: 06.01.2025.

9. Соболь, С. А. Методика преподавания дисциплин по теории алгоритмов с использованием образовательной платформы iRUNNER / С. А. Соболь, В. М. Котов, Е. П. Соболевская // Электронный науч.-методич. журнал «Педагогика информатики». – 2020 – № 2.

10. Теория алгоритмов: учеб. пособие / П. А. Иржавский [и др.] – Минск: БГУ, 2013. – 159 с. – [Электронный ресурс]. – Режим доступа: https://elib.bsu.by/handle/123456789/91612. – Дата доступа: 06.01.2025.

4.2. Электронные ресурсы

1. Образовательный портал БГУ [Электронный ресурс]. – Режим доступа: https://edufpmi.bsu.by/course/view.php?id=118. – Дата доступа: 06.01.2025.

2. Образовательная платформа Insight Runner [Электронный ресурс]. – Режим доступа: https://acm.bsu.by. – Дата доступа: 06.01.2025.

Приложение 1.

Листинг кода

```
class Node(object):
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.ltag = None
        self.rtag = None

class Tree(object):
    def __init__(self):
        self.root = None

def InsertRecursively(v, x):
    if v is None:
        return Node(x)
    if x < v.key:
        v.left = InsertRecursively(v.left, x)
    elif x > v.key:
        v.right = InsertRecursively(v.right, x)
    # v.key == x
    return v

def InsertIteratively(x):
    parent = None
    v = tree.root
    while v is not None:
        parent = v
        if x < v.key:
            v = v.left
        elif x > v.key:
            v = v.right
        else:  # x == v.key
            return
    w = Node(x)
    if parent is None:
        tree.root = w
    elif x < parent.key:
        parent.left = w
    elif x > parent.key:
        parent.right = w

def SearchRecursively(v, x):
    if v is None or v.key == x:
        return v
    elif x < v.key:
        return SearchRecursively(v.left, x)
    else: # x > v.key
```

```
        return SearchRecursively(v.right, x)

def SearchIteratively(x):
    v = tree.root
    while v is not None:
        if v.key == x:
            return v
        elif x < v.key:
            v = v.left
        else: # x > v.key
            v = v.right
    return None

def FindMin(v):
    if v.left is not None:
        return FindMin(v.left)
    else:
        return v

def DeleteRecursively(v, x):
    if v is None:
        return None
    if x < v.key:
        v.left = DeleteRecursively(v.left, x)
        return v
    elif x > v.key:
        v.right = DeleteRecursively(v.right, x)
        return v
    # v.key == x
    if v.left is None:
        return v.right
    elif v.right is None:
        return v.left
    else:
        # both subtrees are present
        min_key = FindMin(v.right).key
        v.key = min_key
        v.right = DeleteRecursively(v.right, min_key)
        return v

def ReplaceChild(parent, old, new):
    if parent is None:
        tree.root = new
    elif parent.left == old:
        parent.left = new
    elif parent.right == old:
        parent.right = new

def DeleteIteratively(x):
    parent = None
    v = tree.root
```

```
    while True:
        if v is None:
            return
        if x < v.key:
            parent = v
            v = v.left
        elif x > v.key:
            parent = v
            v = v.right
        else:  # x == v.key
            break
    result = None
    if v.left is None:
        result = v.right
    elif v.right is None:
        result = v.left
    else:
        min_node_parent = v
        min_node = v.right
        while min_node.left is not None:
            min_node_parent = min_node
            min_node = min_node.left
        result = v
        v.key = min_node.key
        ReplaceChild(min_node_parent, min_node, min_node.right)
    ReplaceChild(parent, v, result)

def Action(x):
    print(x.key)

def PreOrderTraversal(v):
    if v is not None:
        Action(v)
        PreOrderTraversal(v.left)
        PreOrderTraversal(v.right)

def PostOrderTraversal(v):
    if v is not None:
        PostOrderTraversal(v.left)
        PostOrderTraversal(v.right)
        Action(v)

def InOrderTraversal(v):
    if v is not None:
        InOrderTraversal(v.left)
        Action(v)
        InOrderTraversal(v.right)

tree = Tree()
# a = [10, 20, 5, 17, 4, 30, 6, 19, 3, 16, 40]
a = [5, 4, 7, 6, 2, 10, 1, 11, 3]
```

```
for x in a:
    tree.root =  InsertRecursively(tree.root, x)
#    InsertIteratively(x)

PreOrderTraversal(tree.root)
# PostOrderTraversal(tree.root)
# InOrderTraversal(tree.root)
print()

print(SearchRecursively(tree.root, 3) != None)
# print(SearchIteratively(1) != None)
print()

tree.node = DeleteRecursively(tree.root, 3)
# tree.node = DeleteIteratively(3)

PreOrderTraversal(tree.root)
print()
print(SearchRecursively(tree.root, 3) != None)

head = Tree()
head.left = tree.root
head.right = head

def InOrderTraversal_l(p):
    global p_old
    if p is not None:
        InOrderTraversal_l(p.left)
        if p.left is not None:
            p.ltag = True
        else:
            p.ltag = False
            p.left = p_old
        p_old = p
        InOrderTraversal_l(p.right)

p_old = head
p = head.left
InOrderTraversal_l(tree.root)

def InOrderTraversal_r(p):
    global p_old
    if p is not None:
        InOrderTraversal_r(p.right)
        if p.right is not None:
            p.rtag = True
        else:
            p.rtag = False
            p.right = p_old
        p_old = p
        if p.ltag == True:
```

```
            InOrderTraversal_r(p.left)

p_old = head
p = head.left
InOrderTraversal_r(tree.root)

def InOrderTraversal_n():
    p = head.left
    while p != head:
        while p.ltag == True:
            p = p.left
        while p != head:
            Action(p)
            if p.rtag == False:
                p = p.right
            else: # p.rtag == True
                p = p.right
                break

InOrderTraversal_n()
```