Сириус
IT-Колледж

Лекция 11. Деревья

Бинарные деревья. Деревья поиска. Красно-чёрные деревья. AVL-деревья. B деревья. B+ деревья. B* деревья.

Сириус IT-Колледж

Деревья

Дерево — это связный ациклический граф. Связность означает наличие маршрута между любой парой вершин, ацикличность — отсутствие циклов. Отсюда, в частности, следует, что число рёбер в дереве на единицу меньше числа вершин, а между любыми парами вершин имеется один и только один путь.

Ориентированное (направленное) дерево — ацикличный орграф, в котором только одна вершина имеет нулевую степень захода (в неё не ведут дуги), а все остальные вершины имеют степень захода 1 (в них ведёт ровно по одной дуге). Вершина с нулевой степенью захода называется корнем дерева, вершины с нулевой степенью исхода (из которых не исходит ни одна дуга) называются концевыми вершинами или листьями.

Сириус
IT-Колледж

Бинарные деревья

Двоичное дерево (Бинарное дерево) — иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками. Двоичное дерево является упорядоченным ориентированным деревом.

Для практических целей обычно используют два подвида двоичных деревьев — двоичное дерево поиска и двоичная куча.

Сириус IT-Колледж

Бинарные деревья. Представление в виде списка

```
tree = [None] * 10

def root(key):
    if tree[0] != None:
        print("Дерево уже содержит корень")
    else:
        tree[0] = key

def set_left(key, parent):
    if tree[parent] == None:
        print("Не возможно установить потомка в", (parent * 2) + 1, ", родитель не найден")
    else:
        tree[(parent * 2) + 1] = key

def set_right(key, parent):
    if tree[parent] == None:
        print("Не возможно установить потомка в", (parent * 2) + 2, ", родитель не найден")
    else:
        tree[(parent * 2) + 2] = key
```

Сириус
IT-Колледж

Бинарные деревья. Динамическое представление узлов

Каждый узел дерева содержит следующую информацию:

- Данные
- Указатель на левого потомка
- Указатель на правого потомка

```
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.data = key
```

Сириус
IT-Колледж

Основные операции с бинарным деревом:

- Вставка элемента
- Удаление элемента
- Поиск элемента
- Удаление элемента по значению
- Обход дерева

Дополнительные операции с бинарным деревом:

- Нахождение высоты дерева
- Нахождение слоя дерева
- Нахождение размера дерева

Сириус
IT-Колледж

Бинарные деревья. Применение

- В компиляторах, в частности для выполнения арифметических выражений
- Деревья кодирования Хаффмана в алгоритмах сжатия
- Очереди с приоритетом
- Представления иерархических данных
- В табличных редакторах
- Для индексирования баз данных и кэша
- Для быстрого поиска
- Выделения памяти в компьютерах
- Операций кодирования и декодирования
- Для получения и организации информации из больших объемов данных
- В моделях принятия решений
- В алгоритмах сортировки

Сириус IT-Колледж

Бинарные деревья. Обход

Обход деревьев осуществляется на основе двух основных алгоритмов:

- DFS
- BFS

Обход дерева с использованием поиска в глубину (DFS) в свою очередь может быть разбит на 3 вида:

- Прямой обход (NLR)
  i. Проверяем, не является ли текущий узел пустым или None.
  ii. Показываем поле данных корня (или текущего узла).
  iii. Обходим левое поддерево рекурсивно, вызвав функцию прямого обхода.
  iv. Обходим правое поддерево рекурсивно, вызвав функцию прямого обхода.

Сириус
IT-Колледж

Бинарные деревья. Обход

- Центрированный обход (LNR)
    i. Проверяем, не является ли текущий узел пустым или None.
    ii. Обходим левое поддерево рекурсивно, вызвав функцию центрированного обхода.
    iii. Показываем поле данных корня (или текущего узла).
    iv. Обходим правое поддерево рекурсивно, вызвав функцию центрированного обхода.
- Обратный обход (LRN)
    i. Проверяем, не является ли текущий узел пустым или None.
    ii. Обходим левое поддерево рекурсивно, вызвав функцию обратного обхода.
    iii. Обходим правое поддерево рекурсивно, вызвав функцию обратного обхода.
    iv. Показываем поле данных корня (или текущего узла).

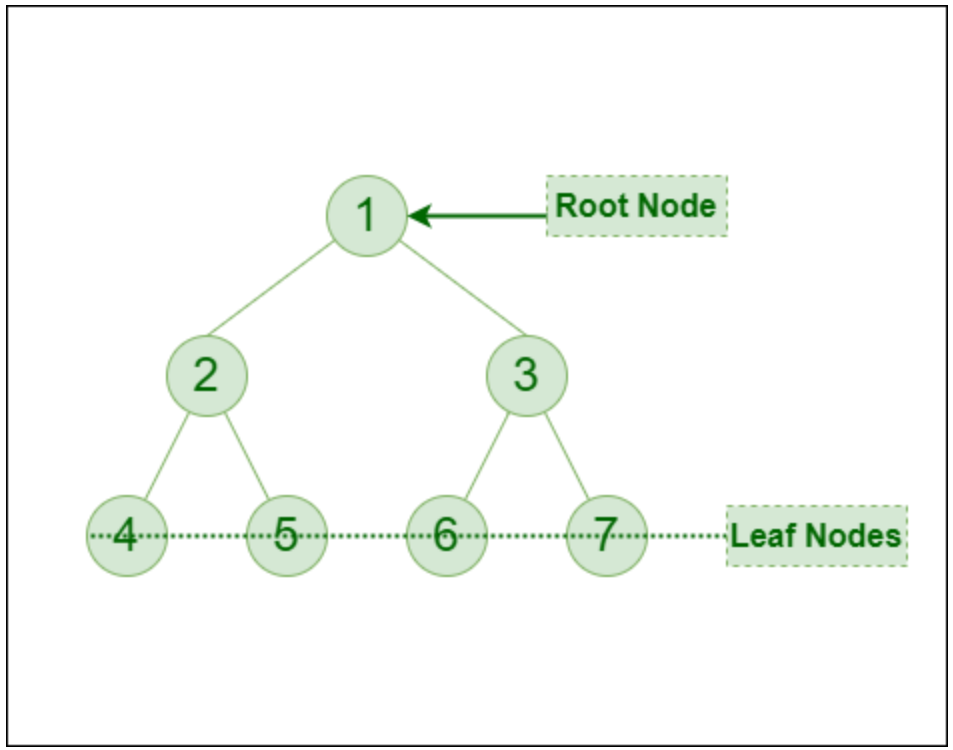
Сириус
IT-Колледж

Бинарные деревья. Обход

При обходе поиском в ширину (BFS) посещаются все узлы в порядке уровней, где сначала посещается каждый узел на уровне, прежде чем выполнить переход на следующий уровень.

Сириус
IT-Колледж

Бинарные деревья. Обход



Прямой обход: 1-2-4-5-3-6-7

Центрированный обход: 4-2-5-1-6-3-7

Обратный обход: 4-5-2-6-7-3-1

Уровневый порядок обхода: 1-2-3-4-5-6-7

Сириус IT-Колледж

Деревья поиска

Бинарное дерево поиска (binary search tree, BST) — структура данных для работы с упорядоченными множествами.

Бинарное дерево поиска обладает следующим свойством: если x — узел бинарного дерева с ключом k, то все узлы в левом поддереве должны иметь ключи, меньшие k, а в правом поддереве большие k .

Основным преимуществом двоичного дерева поиска перед другими структурами данных является возможная высокая эффективность реализации основанных на нём алгоритмов поиска и сортировки.

Сириус
IT-Колледж

Применение бинарных деревьев поиска

- Для индексации
- Для реализации алгоритмов поиска
- Для реализации других структур данных
- Как часть систем принятия решения, компьютерных симуляций для хранения и быстрого доступа к данным
- Для реализации систем автодополнения и проверки орфографии

Сириус IT-Колледж

Деревья поиска

Преимущества:

- Высокая скорость вставки и удаления на сбалансированном дереве ($O(logn)$)
- Высокая скорость поиска ($O(logn)$)
- Эффективное использование памяти
- Позволяют искать значения из диапазона
- Простая реализация
- Автоматическая сортировка элементов при добавлении

Недостатки:

- Всегда необходимо реализовывать сбалансированное бинарное дерево поиска, иначе дерево может выродиться в список, что увеличивает время выполнения операций
- Плохо подходят для случайного доступа к элементам
- Не поддерживают некоторые операции

Сириус
IT-Колледж

Красно-чёрные деревья

Красно-чёрное дерево (англ. red-black tree) — двоичное дерево поиска, в котором баланс осуществляется на основе "цвета" узла дерева, который принимает только два значения: "красный" (англ. red) и "чёрный" (англ. black).

При этом все листья дерева являются фиктивными и не содержат данных, но относятся к дереву и являются чёрными.

Для экономии памяти фиктивные листья можно сделать одним общим фиктивным листом.

Сириус
IT-Колледж

Красно-чёрные деревья

Красно-чёрным называется бинарное поисковое дерево, у которого каждому узлу сопоставлен дополнительный атрибут — цвет и для которого выполняются следующие свойства:

1. Каждый узел промаркирован красным или чёрным цветом
2. Корень и конечные узлы (листья) дерева — чёрные
3. У красного узла родительский узел — чёрный
4. Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов
5. Чёрный узел может иметь чёрного родителя

Сириус
IT-Колледж

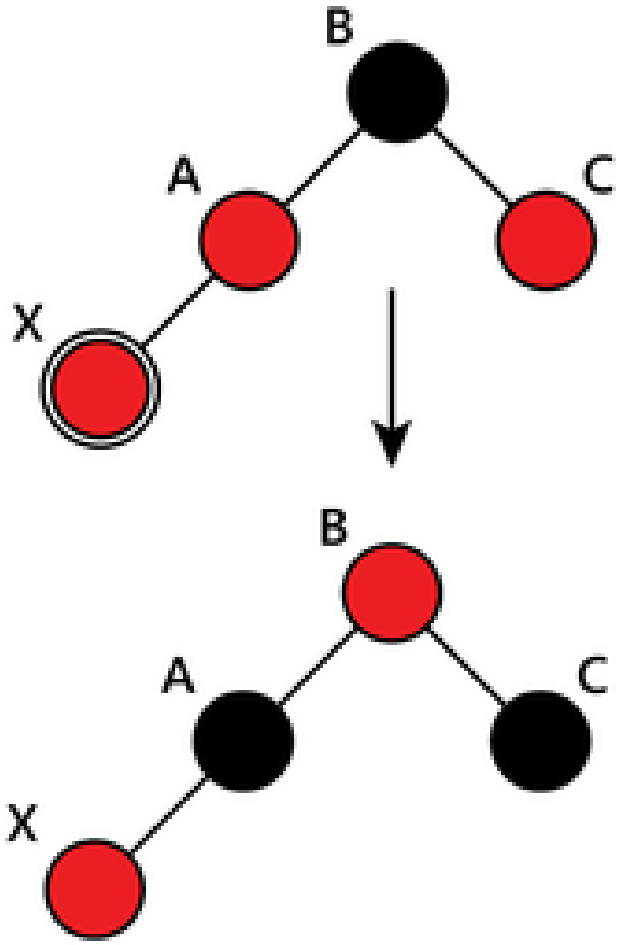
Красно-чёрные деревья. Вставка элемента

Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего сына не станет `None` (то есть этот сын — лист). Вставляем вместо него новый элемент с нулевыми потомками и красным цветом. Теперь проверяем балансировку. Если отец нового элемента черный, то никакое из свойств дерева не нарушено. Если же он красный, то нарушается свойство 3, для исправления достаточно рассмотреть два случая:

Сириус IT-Колледж

Красно-чёрные деревья. Вставка элемента

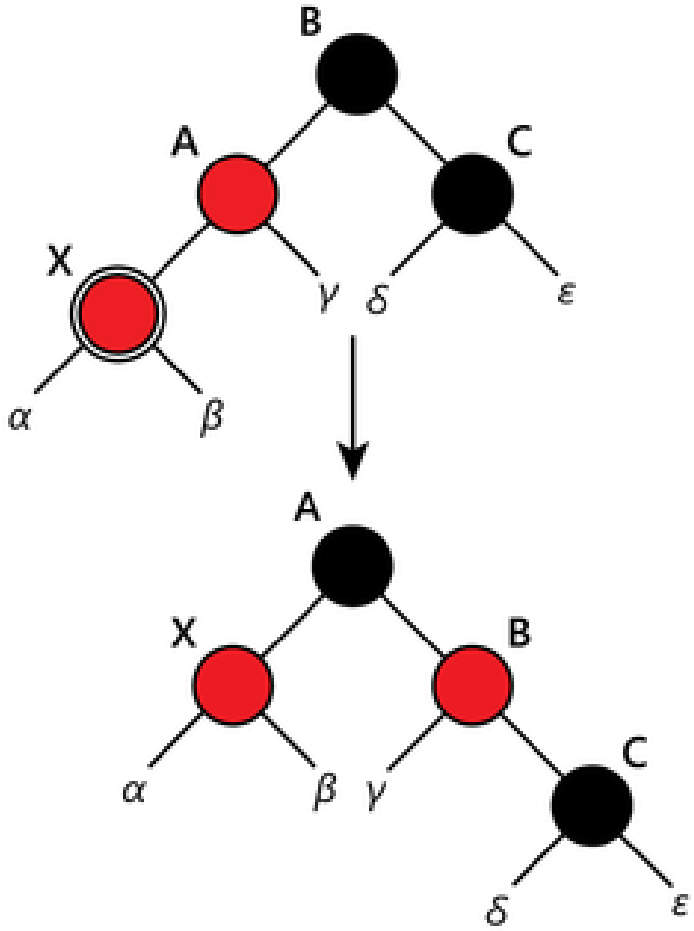
1. "Дядя" этого узла тоже красный. Тогда, чтобы сохранить свойства 3 и 4, просто перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" — в красный. В таком случае черная высота в этом поддереве одинакова для всех листьев и у всех красных вершин "отцы" черные. Проверяем, не нарушена ли балансировка. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет, чтобы дерево удовлетворяло свойству 2



Сириус IT-Колледж

Красно-чёрные деревья. Вставка элемента

2. "Дядя" чёрный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком. Таким образом, свойство 3 и постоянство черной высоты сохраняются.



Сириус IT-Колледж

Красно-чёрные деревья. Удаление вершины

При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

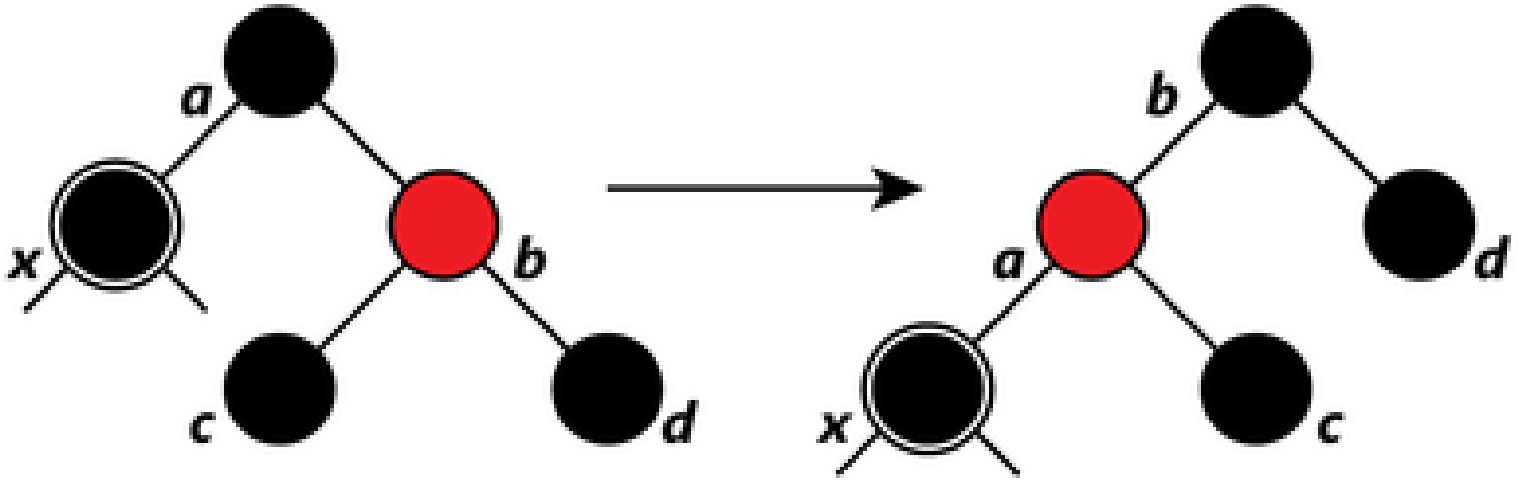
1. Если у вершины нет детей, то изменяем указатель на неё у родителя на `None`.
2. Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.
3. Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка (так как такая вершина находится в правом поддереве исходной вершины и она самая левая в нем, иначе бы мы взяли ее левого ребенка. Иными словами сначала мы переходим в правое поддерево, а после спускаемся вниз в левое до тех пор, пока у вершины есть левый ребенок). Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

Проверим балансировку дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

Сириус
IT-Колледж

Красно-чёрные деревья. Удаление вершины

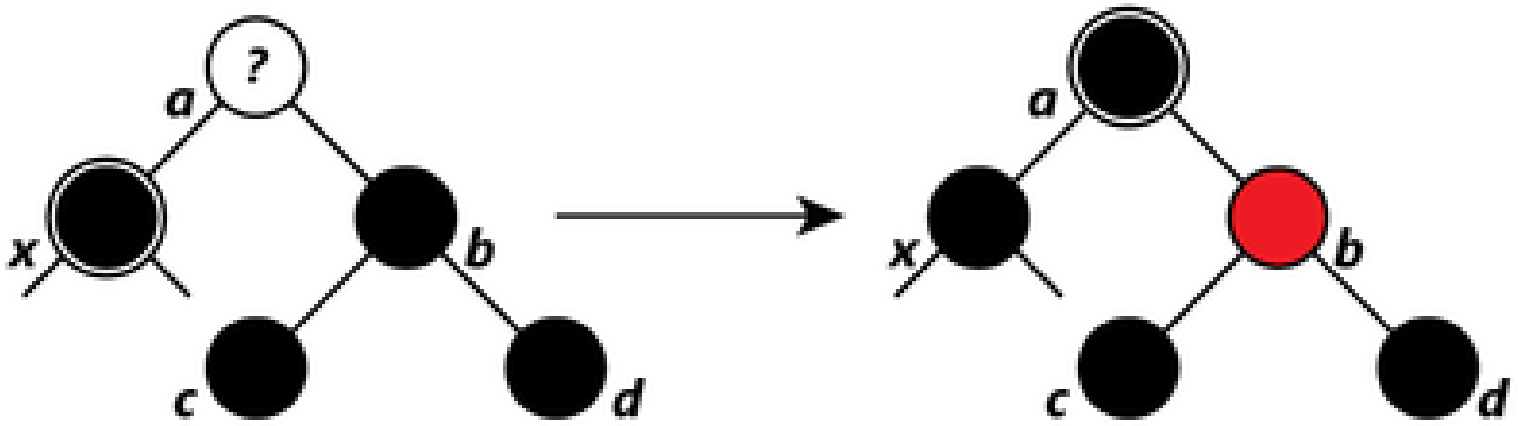
Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца — в красный цвет, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество чёрных узлов, сейчас x имеет чёрного брата и красного отца. Таким образом, мы можем перейти к следующему шагу.



Сириус
IT-Колледж

Красно-чёрные деревья. Удаление вершины

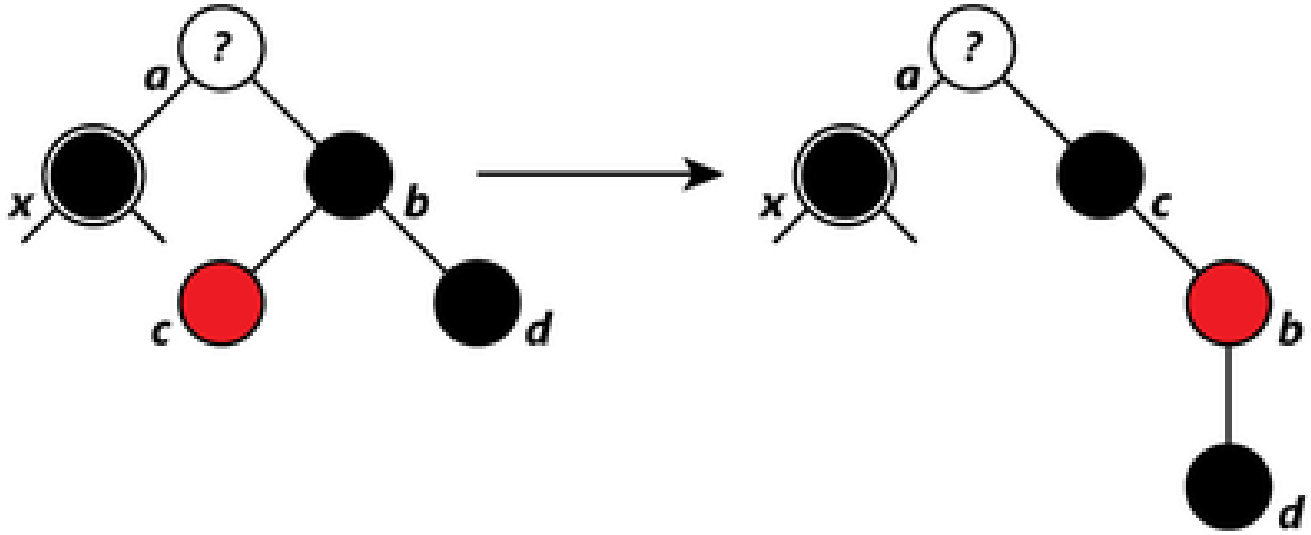
Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество чёрных узлов на путях, проходящих через b, но добавит один к числу чёрных узлов на путях, проходящих через x, восстанавливая тем самым влиянние удаленного чёрного узла. Таким образом, после удаления вершины черная глубина от отца этой вершины до всех листьев в этом поддереве будет одинаковой.



Сириус
IT-Колледж

Красно-чёрные деревья. Удаление вершины

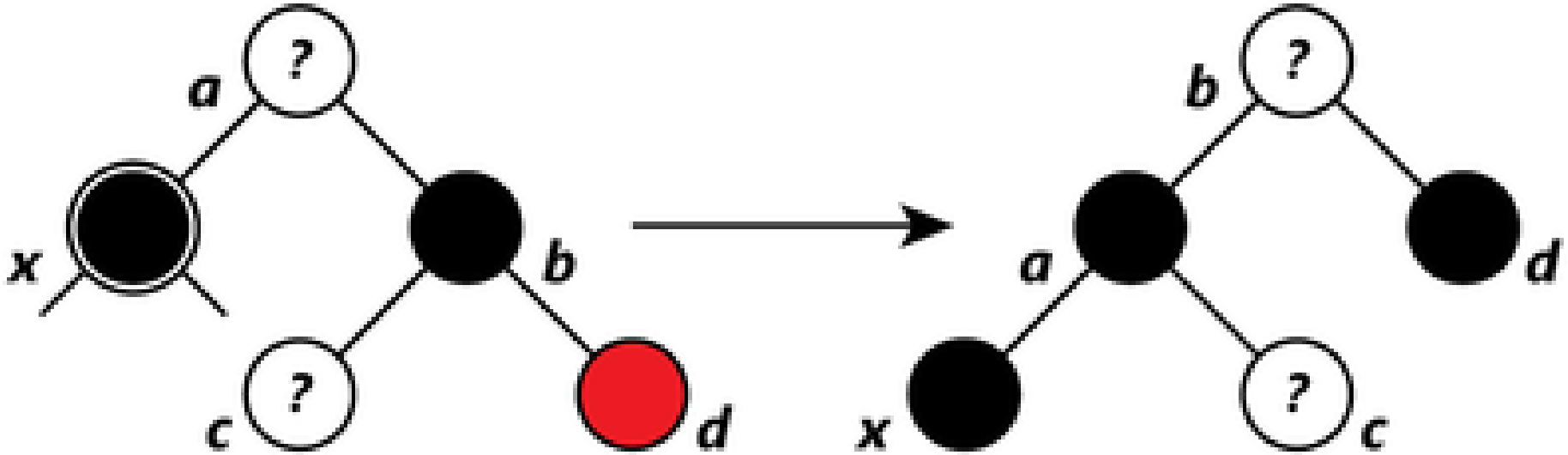
Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество чёрных узлов, но теперь у x есть чёрный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x, ни его отец не влияют на эту трансформацию.



Сириус
IT-Колледж

Красно-чёрные деревья. Удаление вершины

Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца — в чёрный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 3 и 4 не нарушаются. Но у x теперь появился дополнительный чёрный предок: либо a стал чёрным, или он и был чёрным и b был добавлен в качестве чёрного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный чёрный узел. Выходим из алгоритма.



Сириус IT-Колледж

Красно-чёрные деревья. Преимущества

1. Самое главное преимущество красно-черных деревьев в том, что при вставке выполняется не более $O(1)$ вращений.
2. Процедуру балансировки практически всегда можно выполнять параллельно с процедурами поиска, так как алгоритм поиска не зависит от атрибута цвета узлов.
3. Сбалансированность этих деревьев хуже, чем у АВЛ, но работа по поддержанию сбалансированности в красно-чёрных деревьях обычно эффективнее. Для балансировки красно-чёрного дерева производится минимальная работа по сравнению с АВЛ-деревьями.
4. Использует всего 1 бит дополнительной памяти для хранения цвета вершины.

Красно-чёрные деревья являются наиболее активно используемыми на практике самобалансирующимися деревьями поиска. В частности, ассоциативные контейнеры библиотеки STL(map, set, multiset, multimap) основаны на красно-чёрных деревьях. TreeMap в Java тоже реализован на основе красно-чёрных деревьев.

Сириус IT-Колледж

AVL-деревья

АВЛ-дерево (англ. AVL-Tree) — сбалансированное двоичное дерево поиска, в котором поддерживается следующее свойство: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

АВЛ-деревья названы по первым буквам фамилий их изобретателей, Г. М. Адельсона-Вельского и Е. М. Ландиса, которые впервые предложили использовать АВЛ-деревья в 1962 году.

```
class Node:
    def __init__(self, val):
        self.data = val
        self.left = None
        self.right = None
        self.height = 1
```

Сириус
IT-Колледж

AVL-деревья. Балансировка

Балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев $|h(L) - h(R)| = 2$, изменяет связи предок-потомок в поддереве данной вершины так, чтобы восстановилось свойство дерева $|h(L) - h(R)| \leqslant 1$, иначе ничего не меняет. Для балансировки будем хранить для каждой вершины разницу между высотой её левого и правого поддерева `diff[i]=h(L)-h(R)`

Для балансировки вершины используются один из 4 типов вращений:

- Малое левое вращение
- Большое левое вращение (Правое-левое вращение)
- Малое правое вращение
- Большое правое вращение (Левое правое)

Сириус
IT-Колледж

AVL-деревья. Добавление вершины

Пусть нам надо добавить ключ t. Будем спускаться по дереву, как при поиске ключа t. Если мы стоим в вершине a и нам надо идти в поддерево, которого нет, то делаем ключ t листом, а вершину a его корнем. Дальше поднимаемся вверх по пути поиска и пересчитываем баланс у вершин. Если мы поднялись в вершину i из левого поддерева, то `diff[i]` увеличивается на единицу, если из правого, то уменьшается на единицу. Если пришли в вершину и её баланс стал равным нулю, то это значит высота поддерева не изменилась и подъём останавливается. Если пришли в вершину и её баланс стал равным 1 или -1, то это значит высота поддерева изменилась и подъём продолжается. Если пришли в вершину и её баланс стал равным 2 или -2, то делаем одно из четырёх вращений и, если после вращения баланс стал равным нулю, то останавливаемся, иначе продолжаем подъём.

Так как в процессе добавления вершины мы рассматриваем не более, чем $O(h)$ вершин дерева, и для каждой запускаем балансировку не более одного раза, то суммарное количество операций при включении новой вершины в дерево составляет $O(\log n)$ операций.

Сириус
IT-Колледж

AVL-деревья. Удаление вершины

Для простоты опишем рекурсивный алгоритм удаления. Если вершина — лист, то удалим её, иначе найдём самую близкую по значению вершину a, переместим её на место удаляемой вершины и удалим вершину a. От удалённой вершины будем подниматься вверх к корню и пересчитывать баланс у вершин. Если мы поднялись в вершину i из левого поддерева, то `diff[i]` уменьшается на единицу, если из правого, то увеличивается на единицу. Если пришли в вершину и её баланс стал равным 1 или -1, то это значит, что высота этого поддерева не изменилась и подъём можно остановить. Если баланс вершины стал равным нулю, то высота поддерева уменьшилась и подъём нужно продолжить. Если баланс стал равным 2 или -2, следует выполнить одно из четырёх вращений и, если после вращений баланс вершины стал равным нулю, то подъём продолжается, иначе останавливается.

В результате указанных действий на удаление вершины и балансировку суммарно тратится, как и ранее, $O(h)$ операций. Таким образом, требуемое количество действий — $O(\log n)$.

Сириус
IT-Колледж

B деревья

B-дерево — структура данных, дерево поиска. С точки зрения внешнего логического представления — сбалансированное, сильно ветвистое дерево. Часто используется для хранения данных во внешней памяти.

Использование B-деревьев впервые было предложено Р. Бэйером и Э. МакКрейтом в 1970 году.

С точки зрения физической организации B-дерево представляется как мультисписочная структура страниц памяти, то есть каждому узлу дерева соответствует блок памяти (страница). Внутренние и листовые страницы обычно имеют разную структуру.
Структура B-дерева применяется для организации индексов во многих современных СУБД.

Сириус IT-Колледж

B деревья

B-деревом называется дерево, удовлетворяющее следующим свойствам:

1. Ключи в каждом узле обычно упорядочены для быстрого доступа к ним. Корень содержит от 1 до $2t-1$ ключей. Любой другой узел содержит от $t-1$ до $2t-1$ ключей. Листья не являются исключением из этого правила. Здесь t — параметр дерева, не меньший 2 (и обычно принимающий значения от 50 до 2000).
2. У листьев потомков нет. Любой другой узел, содержащий ключи $K_1, \ldots, K_n$, содержит $n+1$ потомков. При этом
   i. Первый потомок и все его потомки содержат ключи из интервала $(-\infty, K_1)$
   ii. Для $2 \leq i \leq n$, i-й потомок и все его потомки содержат ключи из интервала (K_{i-1}, K_i)
   iii. $(n+1)$-й потомок и все его потомки содержат ключи из интервала (K_n, ∞)
3. Глубина всех листьев одинакова.

Свойство 2 можно сформулировать иначе: каждый узел B-дерева, кроме листьев, можно рассматривать как упорядоченный список, в котором чередуются ключи и указатели на потомков.

Сириус IT-Колледж

B деревья

- Во всех случаях полезное использование пространства вторичной памяти составляет свыше 50 %. С ростом степени полезного использования памяти не происходит снижения качества обслуживания.
- Произвольный доступ к записи реализуется посредством малого количества подопераций (обращения к физическим блокам).
- В среднем достаточно эффективно реализуются операции включения и удаления записей; при этом сохраняется естественный порядок ключей с целью последовательной обработки, а также соответствующий баланс дерева для обеспечения быстрой произвольной выборки.
- Неизменная упорядоченность по ключу обеспечивает возможность эффективной пакетной обработки.

Основной недостаток B-деревьев состоит в отсутствии для них эффективных средств выборки данных (то есть метода обхода дерева), упорядоченных по свойству, отличному от выбранного ключа.

Сириус
IT-Колледж

B деревья. Вставка

1. Если дерево пустое, добавить корень и вставить значение.
2. Обновить количество ключей в узле.
3. Найти подходящий для вставки узел.
4. Если узел полон, то:
    i. Вставить элемент в порядке возрастания.
    ii. Так как количество элементов больше предела, разбить узел по медиане.
    iii. Сместить медианный ключ вверх и сделать ключи слева левым потомком, а ключи справа - правым.
5. Если узел не полон, то:
    i. Вставить элемент в порядке возрастания.

Сириус IT-Колледж

B+ деревья

B$^+$-дерево — структура данных на основе B-дерева, сбалансированное n-арное дерево поиска с переменным, но зачастую большим количеством потомков в узле. B$^+$-дерево состоит из корня, внутренних узлов и листьев, корень может быть либо листом, либо узлом с двумя и более потомками.

Изначально структура предназначалась для хранения данных в целях эффективного поиска в блочно-ориентированной среде хранения — в частности, для файловых систем; применение связано с тем, что в отличие от бинарных деревьев поиска, B$^+$-деревья имеют очень высокий коэффициент ветвления (число указателей из родительского узла на дочерние — обычно порядка 100 или более), что снижает количество операций ввода-вывода, требующих поиска элемента в дереве.

Вариант B$^+$-дерева, в котором все значения сохранялись в листовых узлах, систематически рассмотрен в 1979 году, притом отмечено, что такие структуры использовались IBM в технологии файлового доступа для мейнфреймов VSAM по крайней мере с 1973 года.

Сириус
IT-Колледж

B+ деревья

Структура широко применяется в файловых системах — NTFS, ReiserFS, NSS, XFS, JFS, ReFS и BFS используют этот тип дерева для индексирования метаданных; BeFS также использует B^+-деревья для хранения каталогов. Реляционные системы управления базами данных, такие как DB2, Informix, Microsoft SQL Server, Oracle Database (начиная с версии 8), Adaptive Server Enterprise и SQLite поддерживают этот тип деревьев для табличных индексов. Среди NoSQL-СУБД, работающих с моделью «ключ—значение», структура данных реализована для доступа к данным в CouchDB, MongoDB (при использовании подсистемы хранения WiredTiger) и Tokyo Cabinet.

Сириус IT-Колледж

B+ деревья

B+-деревом называется дерево, удовлетворяющее следующим свойствам:

1. Ключи в каждом узле обычно упорядочены для быстрого доступа к ним. Корень содержит от 1 до $2t-1$ ключей. Любой другой узел содержит от $t-1$ до $2t-1$ ключей. Листья не являются исключением из этого правила. Здесь t — параметр дерева, не меньший 2 (и обычно принимающий значения от 50 до 2000).
2. У листьев потомков нет. Любой другой узел, содержащий ключи $K_1, \ldots, K_n$, содержит $n+1$ потомков. При этом
   i. Первый потомок и все его потомки содержат ключи из интервала $(-\infty, K_1)$
   ii. Для $2 \leq i \leq n$, i-й потомок и все его потомки содержат ключи из интервала (K_{i-1}, K_i)
   iii. $(n+1)$-й потомок и все его потомки содержат ключи из интервала (K_n, ∞)
3. Глубина всех листьев одинакова.
4. Листья имеют ссылку на соседа, позволяющую быстро обходить дерево в порядке возрастания ключей, и ссылки на данные.

Сириус
IT-Колледж

B+ деревья

```
class Node:
    def __init__(self, order):
        self.order = order
        self.values = []
        self.keys = []
        self.nextKey = None
        self.parent = None
        self.check_leaf = False
```

Сириус IT-Колледж

B* деревья

B*-дерево — разновидность B-дерева, в которой каждый узел дерева заполнен не менее чем на $2/3$ (в отличие от B-дерева, где этот показатель составляет $1/2$).

B*-деревья предложили Рудольф Байер и Эдвард МакКрейт, изучавшие проблему компактности B-деревьев. B*-дерево относительно компактнее, так как каждый узел используется полнее. В остальном же этот вид деревьев не отличается от простого B-дерева.

Для выполнения требования «заполненность узла не менее $2/3$», приходится отказываться от простой процедуры разделения переполненного узла. Вместо этого происходит «переливание» в соседний узел. Если же и соседний узел заполнен, то ключи приблизительно поровну разделяются на 3 новых узла.

B+-дерево, удовлетворяющее таким требованиям, называется B^{*+}-деревом.