МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ
Московский государственный технический университет им. Н.Э. Баумана
(МГТУ им. Н.Э. Баумана)

Кафедра «Информационная безопасность»
(ИУ8)

Домашнее задание
По дисциплине: «Алгоритмы и структуры данных»

Вариант 1

Тема: «Сравнить 2-3-дерево и красно-черное дерево»

Выполнила: Захарова Ольга,
студентка группы ИУ8-53

Проверил: Чесноков В.О.,
Преподаватель каф. ИУ8

г. Москва
2017 г.

Теоретическая часть

2-3-дерево

2-3-дерево - структура данных, представляющая собой сбалансированное дерево поиска, такое что из каждого узла может выходить две или три ветви и глубина всех листьев одинакова. Является частным случаем B+ дерева.

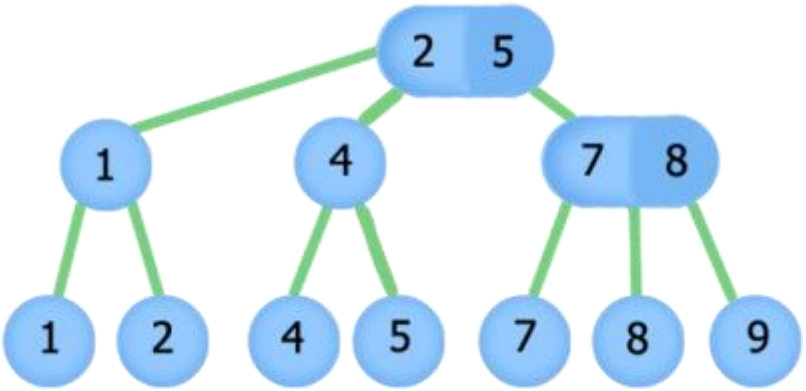


Рис. 1: Пример 2-3 дерева (взято с просторов интернета)

Свойства:

- ✓ нелистовые вершины имеют либо 2, либо 3 сына,
- ✓ нелистовая вершина, имеющая двух сыновей, хранит максимум левого поддерева. Нелистовая вершина, имеющая трех сыновей, хранит два значения. Первое значение хранит максимум левого поддерева, второе максимум центрального поддерева,
- ✓ сыновья упорядочены по значению максимума поддерева сына,
- ✓ все листья лежат на одной глубине,
- ✓ высота 2-3 дерева $O(\log n)$, где n — количество элементов в дереве.

Нелистовые вершины содержат одно или два поля, указывающие на диапазон значений в их поддеревьях. Значение первого поля строго больше наибольшего значения в левом поддереве и меньше или равно наименьшему значению в правом поддереве (или в центральном поддереве, если это 3-вершина); аналогично, значение второго поля (если оно есть) строго больше наибольшего значения в центральном поддереве и меньше или равно, чем наименьшее значение в правом поддереве. Эти нелистовые вершины используются для направления функции поиска к нужному поддереву и, в конечном итоге, к нужному листу.

Вставка (в дерево элемента с ключом key):

1. Если дерево пусто, то создать новую вершину, вставить ключ и вернуть в качестве корня эту вершину, иначе
2. Если вершина является листом, то вставляем ключ в эту вершину и если получили 3 ключа в вершине, то разделяем её, иначе
3. Сравниваем ключ key с первым ключом в вершине, и если key меньше данного ключа, то идем в первое поддерево и переходим к пункту 2, иначе
4. Смотрим, если вершина содержит только 1 ключ (является 2-вершиной), то идем в правое поддерево и переходим к пункту 2, иначе
5. Сравниваем ключ key со вторым ключом в вершине, и если key меньше второго ключа, то идем в среднее поддерево и переходим к пункту 2, иначе
6. Идем в правое поддерево и переходим к пункту 2.

Поиск

1. Ищем искомый ключ key в текущей вершине, если нашли, то возвращаем вершину, иначе
2. Если key меньше первого ключа вершины, то идем в левое поддерево и переходим к пункту 1, иначе
3. Если в дереве 1 ключ, то идем в правое поддерево (среднее, если руководствоваться нашим классом) и переходим к пункту 1, иначе
4. Если key меньше второго ключа вершины, то идем в среднее поддерево и переходим к пункту 1, иначе
5. Идем в правое поддерево и переходим к пункту 1.

Удаление

Удаление в 2-3-дереве происходит только из листа. Поэтому, когда мы нашли ключ, который нужно удалить, сначала надо проверить, находится ли этот ключ в листовой или нелистовой вершине. Если ключ находится в нелистовой вершине, то нужно найти эквивалентный ключ для удаляемого ключа из листовой вершины и поменять их местами. Для нахождения эквивалентного ключа есть два варианта: либо найти максимальный элемент в левом поддереве, либо найти минимальный элемент в правом поддереве.

Использование 2-3-дерева значительно более оптимально, чем использование, например, АВЛ-дерева, потому что персистентая СД, основанная на 2-3-дереве, позволяет откатиться на шаг назад (сохраняет предыдущие версии). Соответственно, в задачах, в которых это требуется

(или может потребоваться), его и нужно использовать. Также его можно использовать для реализации абстрактной таблицы (оно подойдет больше, чем просто бинарное дерево поиска).

Красно-черное дерево

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

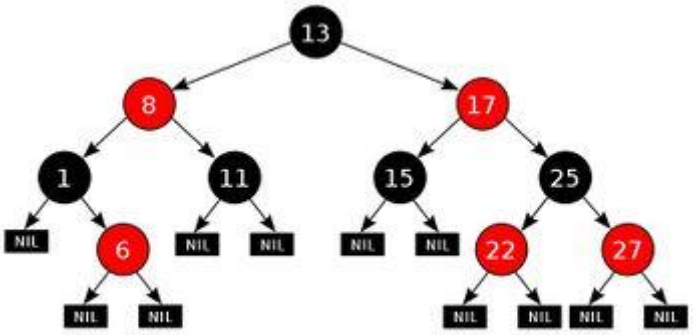


Рис. 2: Пример красно-черного дерево (взято с просторов интернета)

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.

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

Вставка

Чтобы вставить узел, мы сначала ищем в дереве место, куда его следует добавить. Новый узел всегда добавляется как лист, поэтому оба его потомка являются NIL-узлами и предполагаются черными. После вставки красим узел в красный цвет. После этого смотрим на предка и проверяем, не нарушается

ли красно-черное свойство. Если необходимо, мы перекрашиваем узел и производим поворот, чтобы сбалансировать дерево.

Вставив красный узел с двумя NIL-потомками, мы сохраняем свойство черной высоты (свойство 4). Однако, при этом может оказаться нарушенным свойство 3, согласно которому оба потомка красного узла обязательно черны. В нашем случае оба потомка нового узла черны по определению (поскольку они являются NIL-узлами), так что рассмотрим ситуацию, когда предок нового узла красный: при этом будет нарушено свойство 3. Достаточно рассмотреть следующие два случая:

Красный предок, красный "дядя": Ситуацию красный-красный иллюстрирует рис. 3. У нового узла X предок и "дядя" оказались красными. Простое перекрашивание избавляет нас от красно-красного нарушения. После перекраски нужно проверить "дедушку" нового узла (узел B), поскольку он может оказаться красным. Обратите внимание на распространение влияния красного узла на верхние узлы дерева. В самом конце корень мы красим в черный цвет корень дерева. Если он был красным, то при этом увеличивается черная высота дерева.

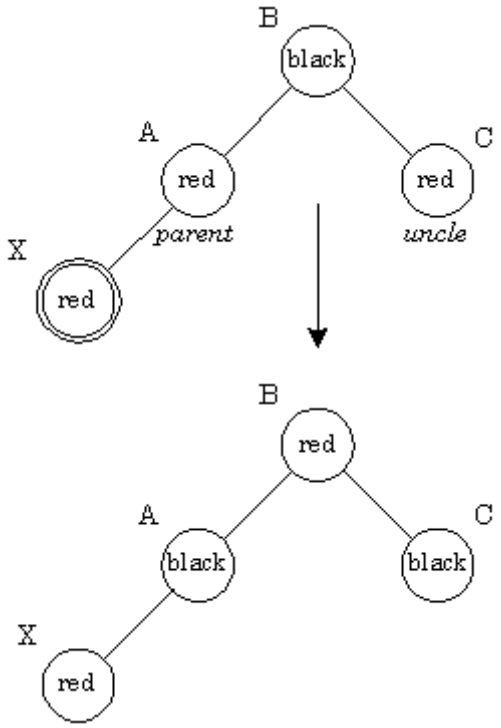


Рис. 3: Вставка - Красный предок, красный "дядя"

Красный предок, черный "дядя": На рис. 4 представлен другой вариант красно-красного нарушения - "дядя" нового узла оказался черным. Здесь

узлы может понадобиться вращать, чтобы скорректировать поддеревья. В этом месте алгоритм может остановиться из-за отсутствия красно-красных конфликтов и вершина дерева (узел A) окрашивается в черный цвет. Обратите внимание, что если узел X был в начале правым потомком, то первым применяется левое вращение, которое делает этот узел левым потомком.

Каждая корректировка, производимая при вставке узла, заставляет нас подняться в дереве на один шаг. В этом случае до остановки алгоритма будет сделано 1 вращение (2, если узел был правым потомком).

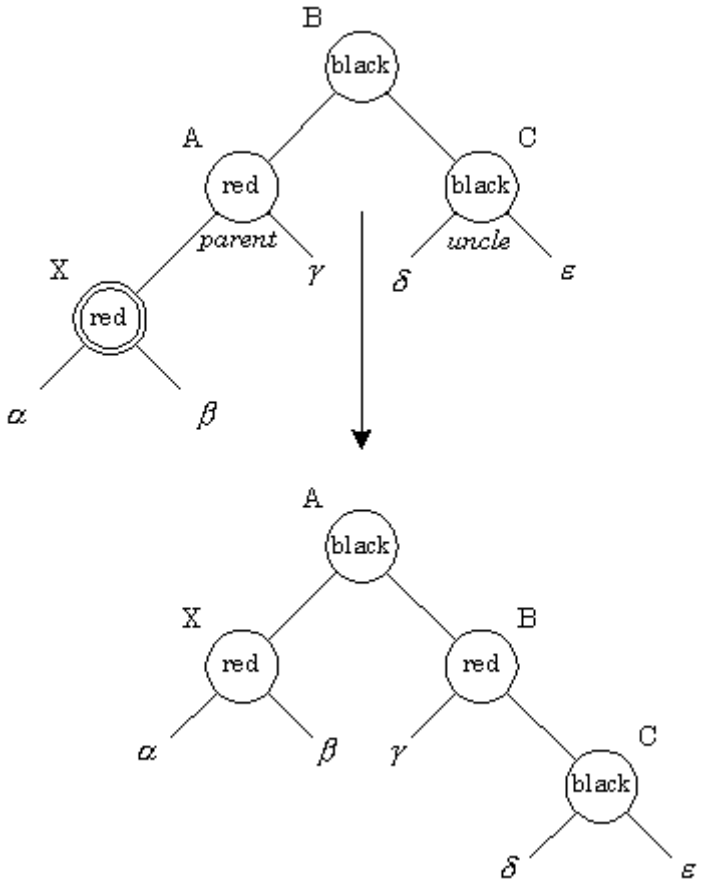


Рис. 4: Вставка - красный предок, черный "дядя"

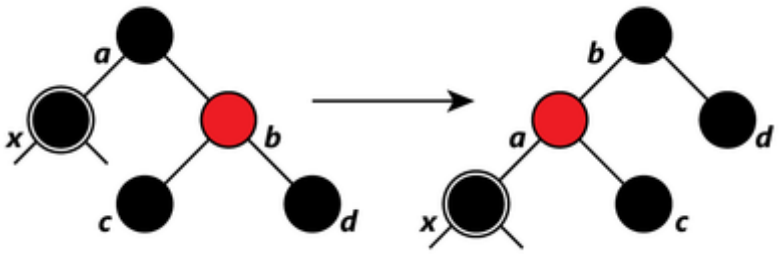
Удаление

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 nil.
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.

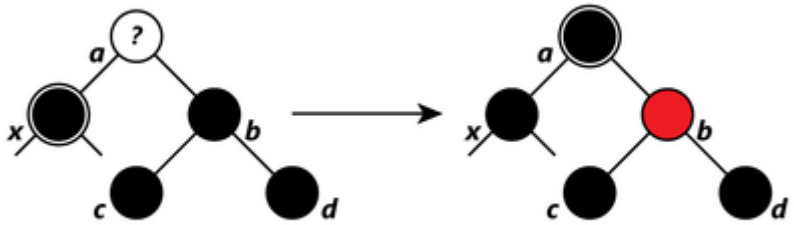
Проверим балансировку дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

1. 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.

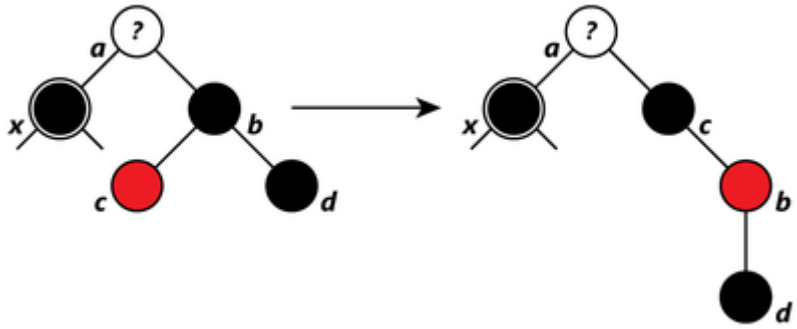


2. Если брат текущей вершины был чёрным, то получаем три случая:

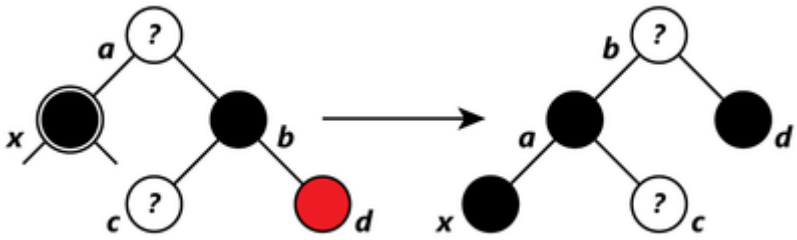
- Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины. Делаем его черным, это не повлияет на количество чёрных узлов на путях, проходящих через b, но добавит один к числу чёрных узлов на путях, проходящих через x, восстанавливая тем самым влияние удаленного чёрного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



- 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.



- 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 xтеперь 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.



Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева. Из рассмотренных случаев ясно, что при удалении выполняется не более трёх вращений.

Поиск в красно-черном аналогичен поиску в обычном двоичном дереве поиска.

Красно-черные деревья тратят меньше ресурсов на поддержание сбалансированности, чем, например, АВЛ-деревья, и их лучше использовать, когда вставка и чтение происходят примерно с одинаковой частотой. Именно на основе красно-черных деревьев основываются большинство реализаций set и map из STL.

План решения поставленной задачи:

1. Реализовать красно-черное дерево и 2-3-дерево (на языке C++), т.е. написать классы с виртуальными функциями вставки, поиска и т.д., реализовать эти функции в классах-наследниках.
2. Провести тесты написанного, подавая различные объемы данных на вход, которые еще и различно будут отсортированы. (скорее всего, напишу функцию для тестирования каждого из методов, причем, как параметр у нее будет количество итераций)
3. Посмотреть на поведение структур по памяти и времени на тестах. Не представляю, как описать это “литературно”, но по сути – гонять операции вставки, удаления и т.д. в цикле, измерять время, а потом делить и вычислять время. Память проще будет измерить на листочке, просто смотреть: сколько элемент занимает, сколько указателей хранится и так далее.