




КРАСНО – ЧЕРНЫЕ ДЕРЕВЬЯ

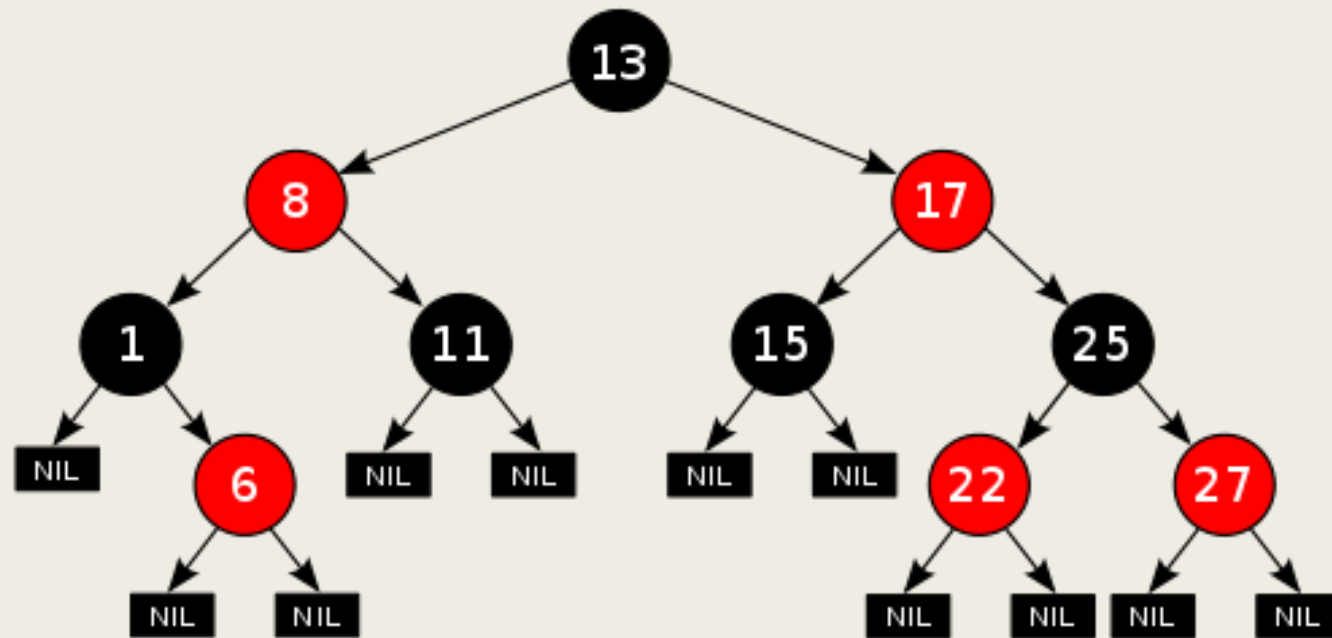
Один из видов самобалансирующихся двоичных деревьев поиска, гарантирующий логарифмический рост высоты дерева от числа узлов и позволяющий быстро выполнять основные операции дерева поиска: добавление, удаление и поиск узла.

Выполнено студентами 141 гр. ВМК: Безвершенко А., Вохмин М.,
Ганжин З., Назаров З., Смирнова А.
2020 г.



Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева — «цвета». Этот атрибут может принимать одно из двух возможных значений — «чёрный» или «красный». Кроме того, выполняются следующие свойства:

- Каждый узел промаркирован красным или чёрным цветом
- Корень и конечные узлы (листья) дерева — чёрные
- У красного узла родительский узел — чёрный
- Все простые пути из любого узла x до листьев содержат одинаковое количество чёрных узлов (так называемая черная высота)
- Чёрный узел может иметь чёрного родителя

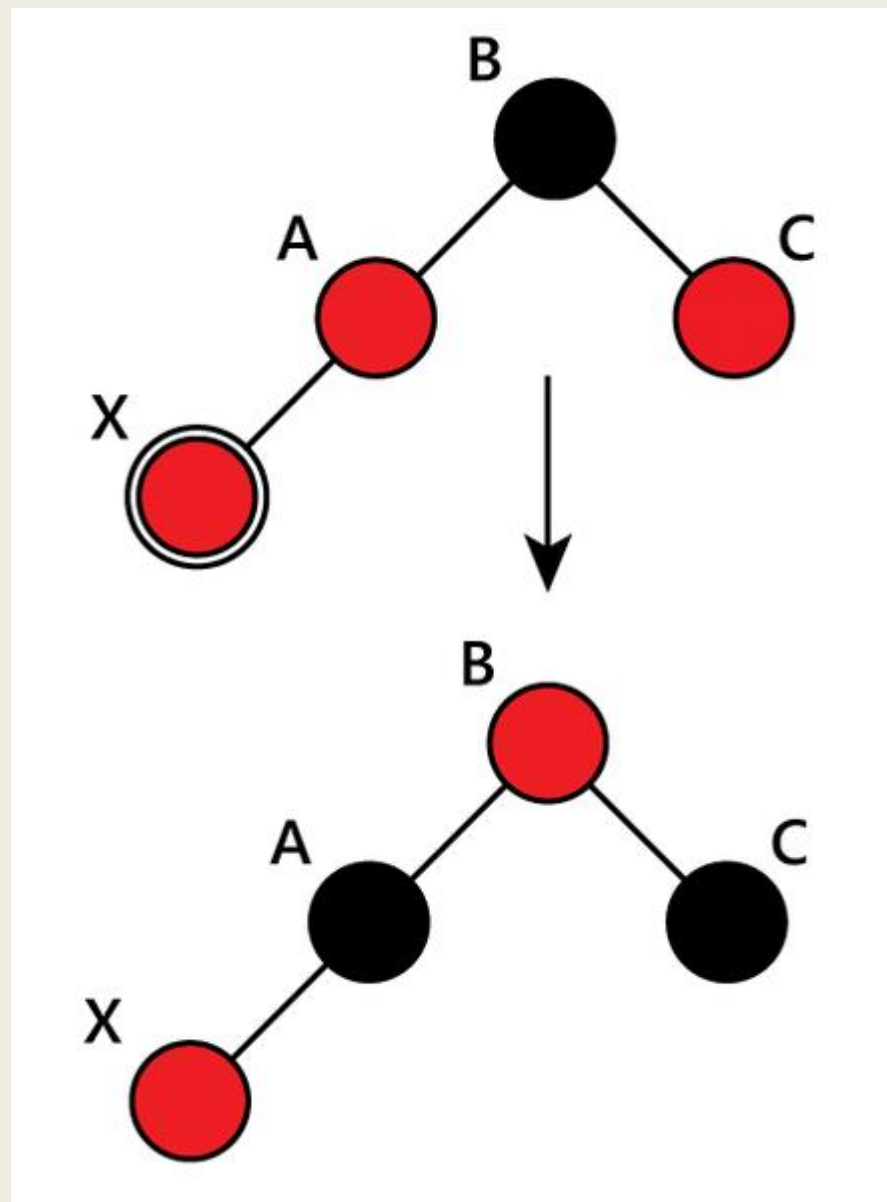


Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идем от корня до тех пор, пока указатель на следующего сына не станет nil (то есть этот сын – лист). Вставляем вместо него новый элемент с нулевыми потомками и красным цветом. Теперь проверяем балансировку. Если отец нового элемента черный, то никакое из свойств дерева не нарушено. Если же он красный, то нарушается свойство 3 (у красного узла родительский узел – черный), для исправления достаточно рассмотреть два случая.

Вставка
элемента
($O \log N$)

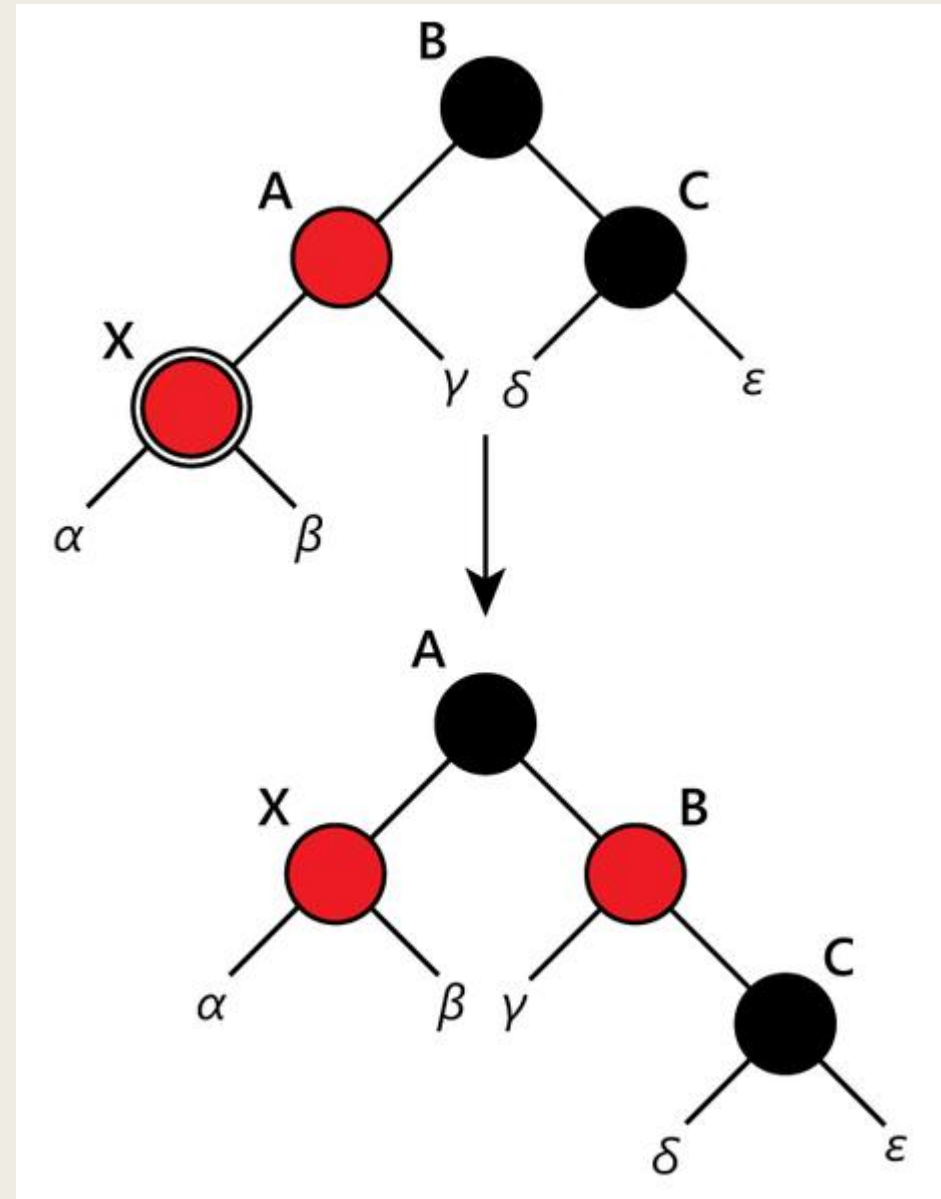
1 случай:

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



2 случай:

«Дядя» черный. Если выполнить только перекрашивание, то может нарушиться постоянство черной 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.



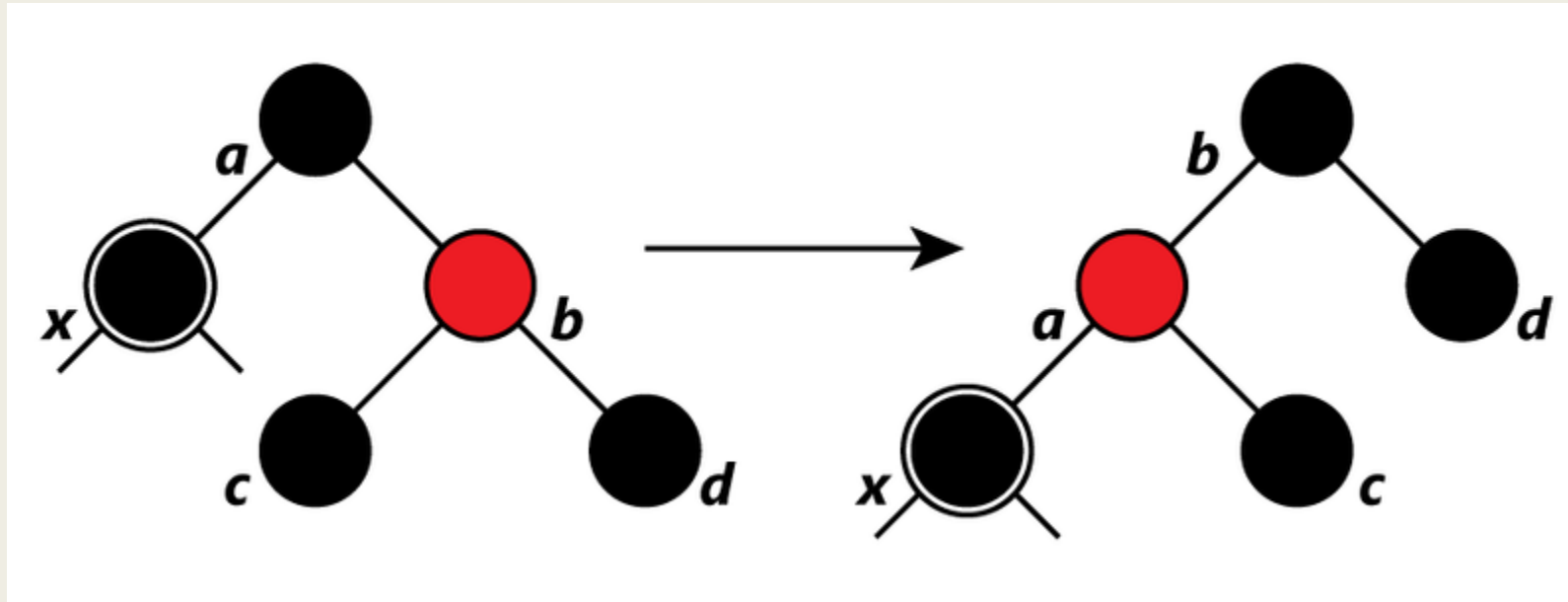
При удалении вершины могут возникнуть три случая в зависимости от количества ее детей:

- Если у вершины нет детей, то изменяем указатель на нее у родителя на `nil`.
- Если у нее только один ребенок, то делаем у родителя ссылку на него вместо этой вершины.
- Если же имеются оба ребенка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребенка (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 ее ключ в изначальную вершину.

Проверим балансировку дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении черной. Рассмотрим ребенка удаленной вершины.

Удаление вершины ($O \log N$)

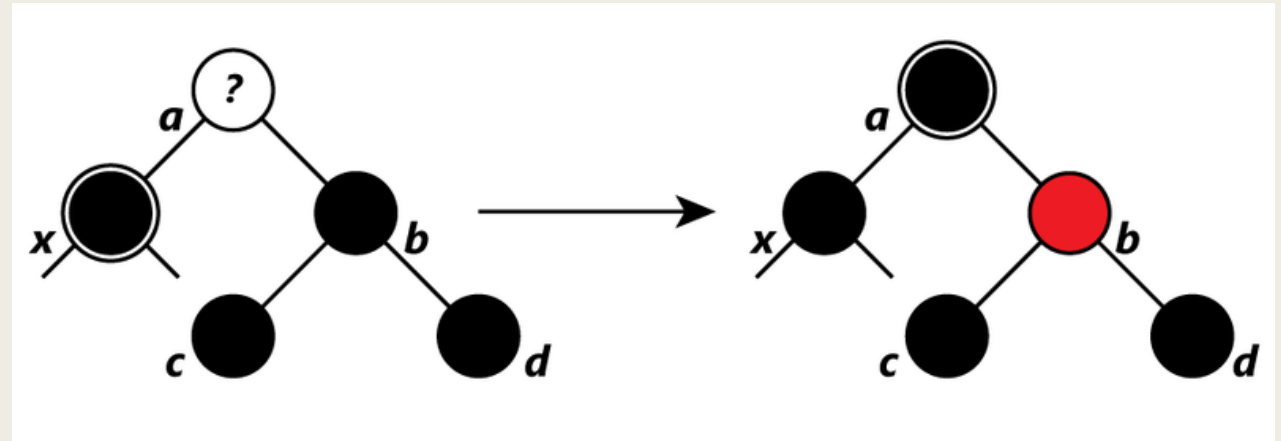
Если брат этого ребенка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в черный, а отца – в красный, сохраняя таким образом черную высоту дерева. Хотя все пути по-прежнему содержат одинаковое количество черных узлов, сейчас x имеет черного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



Если же брат текущей вершины был черным, то получаем три случая.

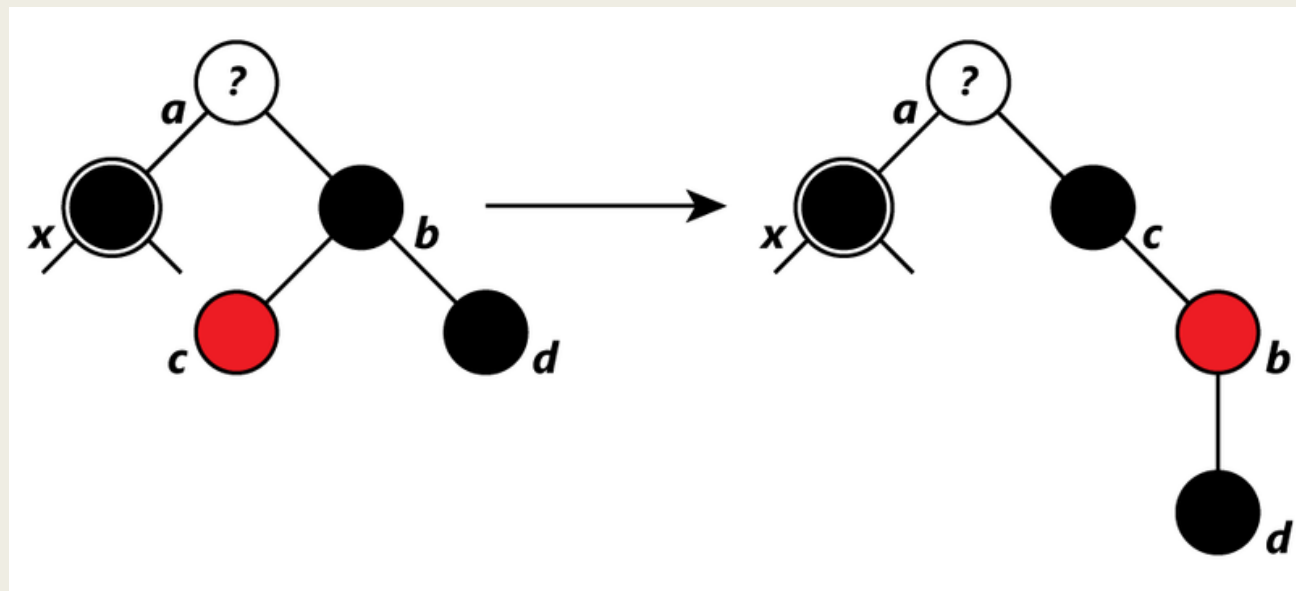
1 случай: оба ребенка у брата - черные

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



2 случай: у брата правый ребенок – черный, а левый – красный

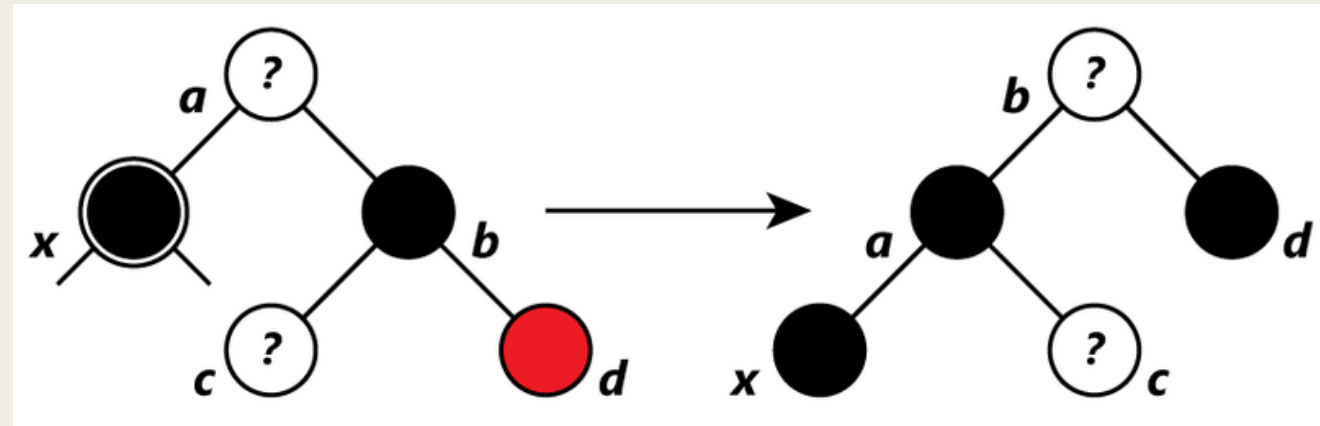
Перекрашиваем брата и левого сына и делаем вращение. Все пути по-прежнему содержат одинаковое количество черных узлов, но теперь у x есть черный брат с красным правым потомком, и мы переходим к следующему случаю. Ни x , ни его отец не влияют на эту трансформацию.



3 случай: у брата правый ребенок – красный

Перекрашиваем брата в цвет отца, его ребенка и отца – в черный, делаем вращение. Поддерево по-прежнему имеет тот же цвет корня, поэтому свойство 3 и 4 не нарушаются. Но у x теперь появился дополнительный черный предок: либо a стал черным, или он и был черным и b был добавлен в качестве черного дедушки. Таким образом, проходящие через x пути проходят через один дополнительный черный узел. Выходим из алгоритма.

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



Для каждого узла необходимо сравнить значение его ключа с искомым ключом. Если ключи одинаковы, то возвращаем указатель на текущий узел, в противном случае, если искомый ключ меньше текущего, то переходим в левое поддерево, иначе в правое.

Поиск
элемента
($O \log N$)

Преимущества красно-черных деревьев

1. При вставке выполняется не более $O(1)$ вращений. Кроме того, примерно половина вставок и удалений произойдут задаром.
2. Процедуру балансировки практически всегда можно выполнять параллельно с процедурой 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-черных деревьях обычно эффективнее. Для балансировки красно-черного дерева производится минимальная работа по сравнению с АВЛ – деревьями.
4. Используется всего 1 бит дополнительной памяти для хранения цвета вершины. Но, на самом деле, в современных вычислительных системах память выделяется кратно байтам, поэтому это не является преимуществом относительно, например, АВЛ – дерева, которое хранит 2 бита.

Красно-черные деревья являются наиболее активно используемыми на практике самобалансирующимися деревьями поиска. В частности, ассоциативные контейнерные библиотеки STL (map, set, multiset, multimap) основаны на красно-черных деревьях.

- https://ru.wikipedia.org/wiki/%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D1%91%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE
- <https://habr.com/ru/post/330644/>
- https://neerc.ifmo.ru/wiki/index.php?title=%D0%9A%D1%80%D0%B0%D1%81%D0%BD%D0%BE-%D1%87%D0%B5%D1%80%D0%BD%D0%BE%D0%B5_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE