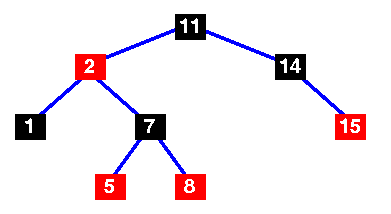
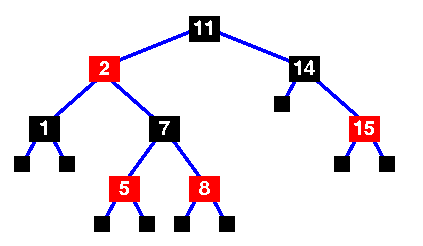
# Основные Концепции

1. Определение и Свойства:
   * Красно-чёрное дерево - это самобалансирующееся двоичное поисковое дерево.
   * Каждый узел окрашен в красный или чёрный цвет.
   * Свойства:
     1. Каждый узел либо красный, либо чёрный.
     2. Корень всегда чёрный.
     3. Все листья (NULL) чёрные.
     4. Красный узел не может иметь красного потомка.
     5. Любой путь от узла до его потомков содержит одинаковое количество чёрных узлов.
2. Вставка и Балансировка:
   * Вставка узлов может нарушить свойства красно-чёрного дерева, поэтому после каждой вставки необходимо выполнять балансировку.
   * Балансировка включает в себя повороты и перекраску узлов.
3. Удаление и Балансировка:
   * Удаление также может нарушить баланс, поэтому требуется последующая балансировка.



красно-черное дерево



красно-черное дерево с добавленными сторожевыми узлами. Реализации алгоритмов красно-черного дерева обычно включают сторожевые узлы в качестве удобного средства маркировки того, что вы достигли конечного узла.

Это черные NULL-узлы свойства 3.

Операция левого поворота:

left\_rotate( Tree T, node x ) {

node y;

y = x->right;

/\* Turn y's left sub-tree into x's right sub-tree \*/

x->right = y->left;

if ( y->left != NULL )

y->left->parent = x;

/\* y's new parent was x's parent \*/

y->parent = x->parent;

/\* Set the parent to point to y instead of x \*/

/\* First see whether we're at the root \*/

if ( x->parent == NULL ) T->root = y;

else

if ( x == (x->parent)->left )

/\* x was on the left of its parent \*/

x->parent->left = y;

else

/\* x must have been on the right \*/

x->parent->right = y;

/\* Finally, put x on y's left \*/

y->left = x;

x->parent = y;

}

Операция вставки:

Вставка довольно сложна и включает в себя ряд случаев. Обратите внимание, что мы начинаем со вставки нового узла x в дерево так же, как и для любого другого двоичного дерева, используя функцию Tree\_insert. Этот новый узел помечен красным и, возможно, разрушает свойство «красный-черный». Основной цикл перемещается вверх по дереву, восстанавливая свойство красного-черного.

rb\_insert( Tree T, node x ) {

/\* Insert in the tree in the usual way \*/

tree\_insert( T, x );

/\* Now restore the red-black property \*/

x->colour = red;

while ( (x != T->root) && (x->parent->colour == red) ) {

if ( x->parent == x->parent->parent->left ) {

/\* If x's parent is a left, y is x's right 'uncle' \*/

y = x->parent->parent->right;

if ( y->colour == red ) {

/\* case 1 - change the colours \*/

x->parent->colour = black;

y->colour = black;

x->parent->parent->colour = red;

/\* Move x up the tree \*/

x = x->parent->parent;

}

else {

/\* y is a black node \*/

if ( x == x->parent->right ) {

/\* and x is to the right \*/

/\* case 2 - move x up and rotate \*/

x = x->parent;

left\_rotate( T, x );

}

/\* case 3 \*/

x->parent->colour = black;

x->parent->parent->colour = red;

right\_rotate( T, x->parent->parent );

}

}

else {

/\* repeat the "if" part with right and left

exchanged \*/

}

}

/\* Colour the root black \*/

T->root->colour = black;

}

|  |  |
| --- | --- |
|  | Вот оригинальное дерево..  Обратите внимание, что на следующих диаграммах черные сторожевые узлы опущены для простоты диаграмм. |
|  | Только что была вызвана процедура вставки дерева для вставки узла «4» в дерево.  Это уже не красно-черное дерево — на пути есть два последовательных красных узла.  11 – 2 – 7 – 5 – 4  Отметьте новый узел x и его наследника y.  y красный, поэтому у нас случай 1... |
|  | Измените цвета узлов 5, 7 и 8. |
|  | Переместите x вверх к его предку, 7.  Родитель x (2) по-прежнему красный, так что это еще не красно-черное дерево.  Отметьте наследника, y.  В данном случае наследника черный, поэтому имеем случай 2... |
|  | Переместите x вверх и поверните влево. |
|  | До сих пор не красно-черное дерево.. наследник черный, а родитель х слева.. |
|  | Поменяйте цвета 7 и 11 и поверните вправо.. |
|  | Теперь это красно-черное дерево, так что мы закончили!  O(logn) время! |

Красно-чёрные деревья применяются во многих областях, где требуется эффективное упорядоченное хранение и быстрый доступ к данным:

1. **Базы данных**: Красно-чёрные деревья используются во многих типах баз данных для управления индексами. Благодаря своей способности поддерживать сбалансированность, они обеспечивают быструю вставку, удаление и поиск данных, что является критически важным для производительности баз данных.
2. **Файловые системы**: В файловых системах, таких как NTFS, используются структуры, похожие на красно-чёрные деревья, для управления свободным пространством на диске, а также для организации файлов и каталогов.
3. **Контейнерные библиотеки**: В стандартной библиотеке шаблонов C++ (STL) и во многих других языковых библиотеках ассоциативные контейнеры, такие как **map**, **multimap**, **set**, и **multiset** часто реализуются с использованием красно-чёрных деревьев, предоставляя баланс между временем доступа и эффективностью изменений.
4. **Планировщики задач**: В некоторых системах планирования задач в операционных системах используются красно-чёрные деревья для управления приоритетами задач.
5. **Компьютерная графика**: Алгоритмы, такие как алгоритмы поиска столкновений и разбиение пространства, могут использовать красно-чёрные деревья для управления динамическими наборами объектов.

**Красно-чёрные деревья**

1. **Менее строгая балансировка**: Красно-чёрные деревья не требуют такой строгой балансировки, как AVL-деревья. Это означает, что они могут быть немного "выше" (иметь больше уровней), но всё равно остаются достаточно сбалансированными для эффективной работы.
2. **Проще в изменении**: При вставке или удалении элементов, красно-чёрные деревья часто требуют меньше изменений (перебалансировок). Это делает их более эффективными в ситуациях, где часто происходят вставки и удаления.
3. **Хороший компромисс**: Они представляют собой хороший компромисс между необходимостью поддержания баланса и затратами на перебалансировку при изменении дерева.

**AVL-деревья**

1. **Строгая балансировка**: AVL-деревья поддерживают более строгую балансировку, что делает их "ниже" (с меньшим количеством уровней) и более быстрыми для операций поиска.
2. **Больше перебалансировок**: Эта строгость означает, что при каждом добавлении или удалении элемента часто требуются сложные операции для поддержания баланса, что может быть менее эффективно при частых изменениях.
3. **Лучше для чтения**: Они идеально подходят для ситуаций, где операции чтения (поиска) происходят гораздо чаще, чем операции вставки или удаления.

источник: https://www.cs.auckland.ac.nz/software/AlgAnim/red\_black.html