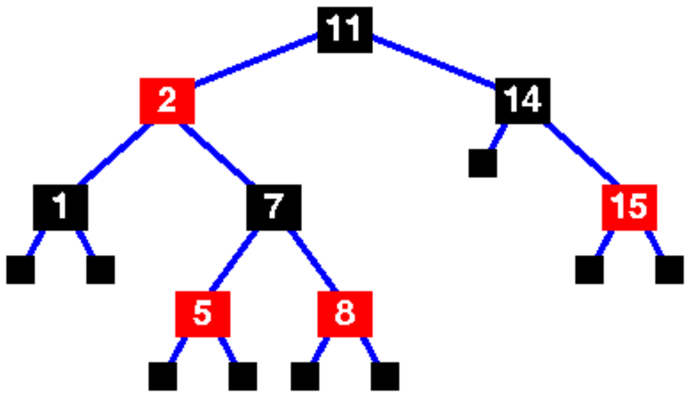
Red-black trees (Cormen et. al. Chapter 13)

A red-black tree is a binary search tree that has the following *red-black properties*:

1. Every node is either red or black
2. Every leaf (NULL) is black
3. If a node is red, then both its children are black
4. Every simple path from a node to a descendant leaf contains the same number of black nodes



Basic red-black tree with the **sentinel** nodes added. Implementations of the red-black tree algorithms will usually include the sentinel nodes as a convenient means of flagging that you have reached a leaf node. They are the NULL black nodes of property 2.

The number of black nodes on any path from, but not including, a node x to a leaf is called the *black-height* of a node, denoted **bh(x).**

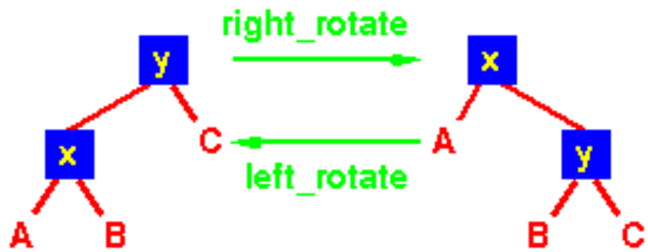
Lemma. A red-black tree with n internal nodes has height at most 2**log(*n*+1)**.
Proof: see textbook.

This demonstrates why the red-black tree is a good search tree: it can always be searched in **O(log n)** time.

Additions and deletions from red-black trees destroy the red-black property. To re-balance it, we need to look at some operations on red-black trees. Re-balancing takes O(log n) time

<u>Rotation</u>

A rotation is a local operation in a search tree that preserves *in-order* traversal key ordering.

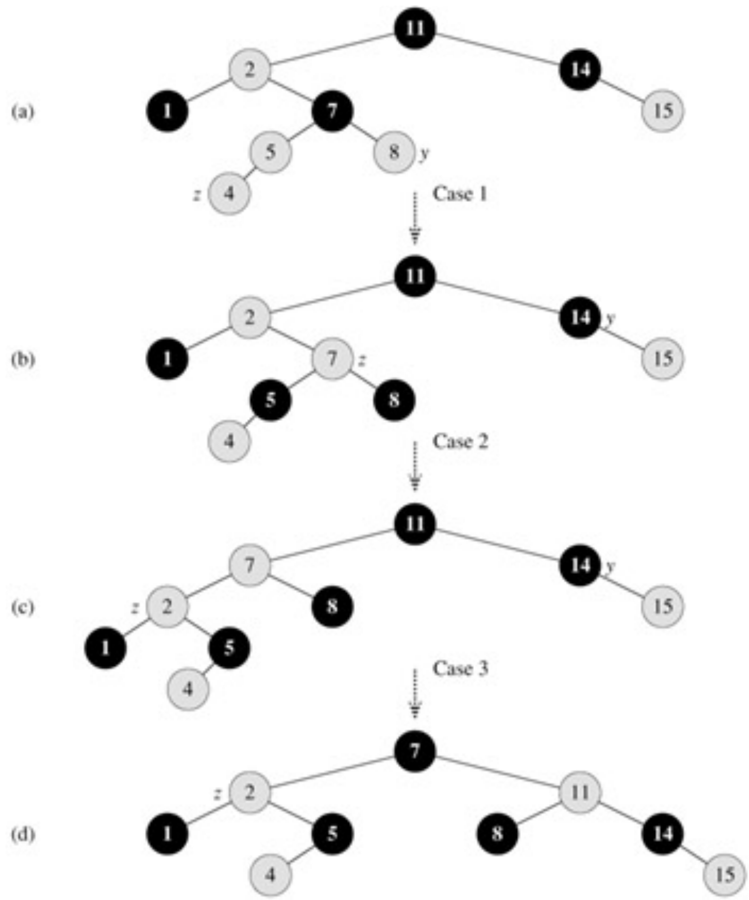


After rotation, the in-order traversal preserves the same order: A x B y C.

Insertion

We first insert node z into the tree T as if it were an ordinary binary search tree, and then we color z red. To guarantee that the red-black properties are pre- served, we then use an auxiliary procedure to recolor nodes and perform rotations.

```
RB-INSERT-FIXUP(T , z )
    while color [p [z ]] = RED
        if p [z ] = left [p[p[z]]]
            y ← right [p[p[z]]]
            if color [y] = RED
                color [p[z]] ← BLACK      //  Case 1
                color [y] ← BLACK         // Case 1
                color [p[p[z]]] ← RED     // Case 1
                z ← p [p[z]]              // Case 1
            else if z  = right [p[z ]]
                z ← p [z ]                //  Case 2
                LEFT-ROTATE(T , z )       // Case 2
            else
                color[p[z]] ← BLACK       // Case 3
                color[p[p[z]]] ← RED      //  Case 3
                RIGHT-ROTATE(T, p[p[z]])       // case 3
        else
…
```



Deletion

We first splice out the node y from the tree T as if it were an ordinary binary search tree, and call an auxiliary procedure RB-DELETE-FIXUP to restore the RB property of the tree. If y is red, the redblack properties still hold when y is spliced out, for the following reasons: 1) no black-heights in the tree have changed, 2) no red nodes have been made adjacent, and 3) since y could not have been the root if it was red, the root remains black.

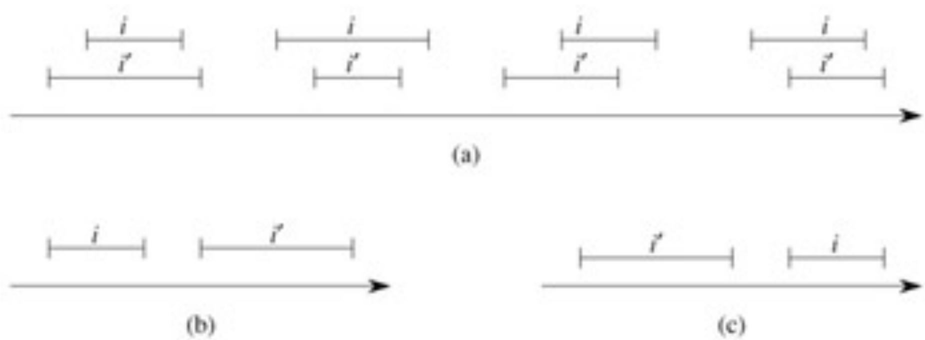
Augmenting Red-black tree

The process of augmenting a basic data structure to support additional functionality occurs quite frequently in algorithm design. Augmenting a data structure can be broken into four steps:

1. choosing an underlying data structure,
2. determining additional information to be maintained in the underlying data structure,
3. verifying that the additional information can be maintained for the basic modifying operations on the underlying data structure, and
4. developing new operations.

Example 1. Given a set of **n** elements, where **i** $\in$ {1, 2,..., **n**}, the *selection problem* is to select the element in the set with the **i**th smallest key (i.e. the *order statistic*). We saw that any order statistic could be retrieved in **O**(**n**) time from an unordered set. Red-black trees can be modified so that any order statistic can be determined in **O**(lg **n**) time. We store in each node (as the field of *rank*) a count of how many descendants it has, and use this to determine which path to follow: if the rank of the left child (denote as r) ≥ i, go to the left child; if r= i-1, return the node; otherwise (if r < i-1), go to the right node and decrease i to i-r. The rank can be updated efficiently since adding a node only affects the counts of its O(log *n*) ancestors, and tree rotations only affect the counts of the nodes involved in the rotation (O(log n)).

Example 2. (Interval tree)
We say that intervals i and i' overlap if i ∩ i' ≠ ø, that is, if low[i] ≤ high[i'] and low[i'] ≤ high[i]. Any two intervals i and i' satisfy the interval *trichotomy*; that is, exactly one of the following three properties holds: a) i and i' overlap, b) i is to the left of i' (i.e., high[i]< low[i']), or c) i is to the right of i' (i.e., high[i']< low[i]).



An *interval tree* is a red-black tree that maintains a dynamic set of elements, with each element x containing an interval. Interval trees support the following operations.
- INTERVAL-INSERT(T, x): adds the element x;
- INTERVAL-DELETE(T, x): removes the element x from the interval tree T.
- INTERVAL-SEARCH(T, i): returns a pointer to an element x in the interval tree T such that int[x] overlaps interval i, or the sentinel nil[T] if no such element is in the set.

We choose a red-black tree in which each node x contains an interval int[x] and the key of x is the low endpoint, low[int[x]], of the interval. Thus, an inorder tree walk of the data structure lists the intervals in sorted order by low endpoint.

In addition to the intervals themselves, each node x contains a value max[x], which is the maximum value of any interval endpoint stored in the subtree rooted at x.

We can determine **max**[**x**] given interval **int**[**x**] and the **max** values of node **x**'s children: **max**[**x**] = max(**high**[**int**[**x**]], **max**[**left**[**x**]], **max**[**right**[**x**]]). Thus, insertion and deletion maintaining the max values in the tree can run in **O**(lg **n**) time. In fact, updating the max fields after a rotation can be accomplished in O(1) time.

The only new operation we need is INTERVAL-SEARCH(T, i), which finds a node in tree T whose interval overlaps interval i.

```
INTERVAL-SEARCH(T , i )
    x ← root [T]
    while x ≠ nil [T ] and i does not overlap int [x]
        if left [x] ≠ nil [T] and max [left[x]] ≥ low [i]
            x ← left [x]
        else
            x ← right [x]
    return x
```

The search terminates when either an overlapping interval is found or x points to the sentinel nil[T]. Since each iteration of the basic loop takes O(1) time, and since the height of an n-node red-black tree is O(lg n), the INTERVAL-SEARCH procedure takes O(lg n) time.