Algorithmic Design / Algorithmic Data Mining

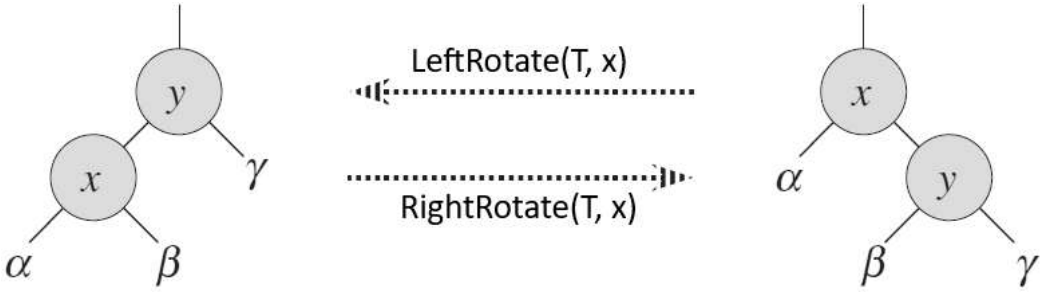
Data Mining - Lecture 2

Tommaso Padoan <tommaso.padoan@units.it>

Red-Black trees operations

Since a RBT is a BST, to search a key in it we can do in the same way as we would in a BST. Therefore, such operation takes time linear in the height of the tree, which for RBTs is guaranteed to grow logarithmically in the total number of nodes.

On the other hand, the result of insertion and deletion operations of BSTs, when run on a RBT, may violate the red-black properties, and so they cannot be used directly as they are. To restore the balance properties, the tree structure may need to be changed and also the color of some nodes. The former is adjusted through local operations called *rotations*, as depicted in the figure below. Both kinds of rotation assume that the nodes x and y are inner nodes (not *NIL* leaves).



Consider, for example, the left rotation. Since the input tree is assumed to be a BST, we know that (the keys of) $x \leq \beta \leq y$, thus the subtree β can become the right child of x and the latter the left child of y. Indeed, rotations always preserve the BST property. However, they only change the pointers structure, leaving unchanged all other attributes. So, to ensure the red-black properties as well, colors may still need to be adjusted. Both rotation procedures, which are symmetrical, take only constant $O(1)$ time, since they just locally update a few pointers, but they may still be used multiple times during a single insertion or deletion operation.

```
def LeftRotate(T, x):
  y = x.right
  x.right = y.left
  if y.left != T.nil:
    y.left.p = x
  y.p = x.p
  if x.p == T.nil:
    T.root = y
  elif x == x.p.left:
    x.p.left = y
  else:
    x.p.right = y
  y.left = x
  x.p = y

def RightRotate(T, x):
  y = x.left
  x.left = y.right
  if y.right != T.nil:
    y.right.p = x
  y.p = x.p
  if x.p == T.nil:
    T.root = y
  elif x == x.p.right:
    x.p.right = y
  else:
    x.p.left = y
  y.right = x
  x.p = y
```

Insertion

The insertion procedure basically works as for BSTs, with the additional final steps required to restore balance and adjust nodes' color. So, the input node z is initially added in an empty spot, i.e. a *NIL* leaf, found by traversing the tree depending on the value of its key $z.key$ (going left if $z.key$ is less than the one of the current node, otherwise right). Proceeding in this way, the same of BSTs, ensures that the

tree we obtain is still a BST. However, the newly added node is initially colored red and its children are both black *NIL* leaves. This may violate the red-black properties of the tree, and so a subroutine, called RB-Insert-Fixup, is called to restore them.

```
def Insert(T, z):
  y = T.nil
  x = T.root
  while x != T.nil:
    y = x
    if z.key < x.key:
      x = x.left
    else:
      x = x.right
  z.p = y
  if y == T.nil:
    T.root = z
  elif z.key < y.key:
    y.left = z
  else:
    y.right = z
  z.left = T.nil
  z.right = T.nil
  z.color = RED        # additional instruction
  RBInsertFixup(T, z)  # additional instruction
```

The fixup subroutine works by iteratively checking for violations and solving them, until no more violations are found. In particular, it first checks rule 4 (children of red nodes must be black), and possibly fixes violations by changing some colors. This may lead to increase the number of black nodes in the tree, with consequent violations of rule 5 (same amount of black nodes along downwards paths from each node to leaves). Thus, the subroutine solves the latter problem by performing some rotation, which may lead to new color violations, and so on. To be precise, also rule 2 (the root is black) may be violated, but it is immediate to restore, just coloring it black, as done by the last instruction of the subroutine.

Violations are not searched in the whole tree, just locally. Indeed, the subroutine is first called on to the newly added red node z. Assuming the tree was a RBT before the insertion, any possible violation must be related to z and its parent. Whenever some fixing is performed on a node z (and its close relatives), then any new violation introduced must be related to the parent or grandparent of z, therefore the search continues locally on to them.

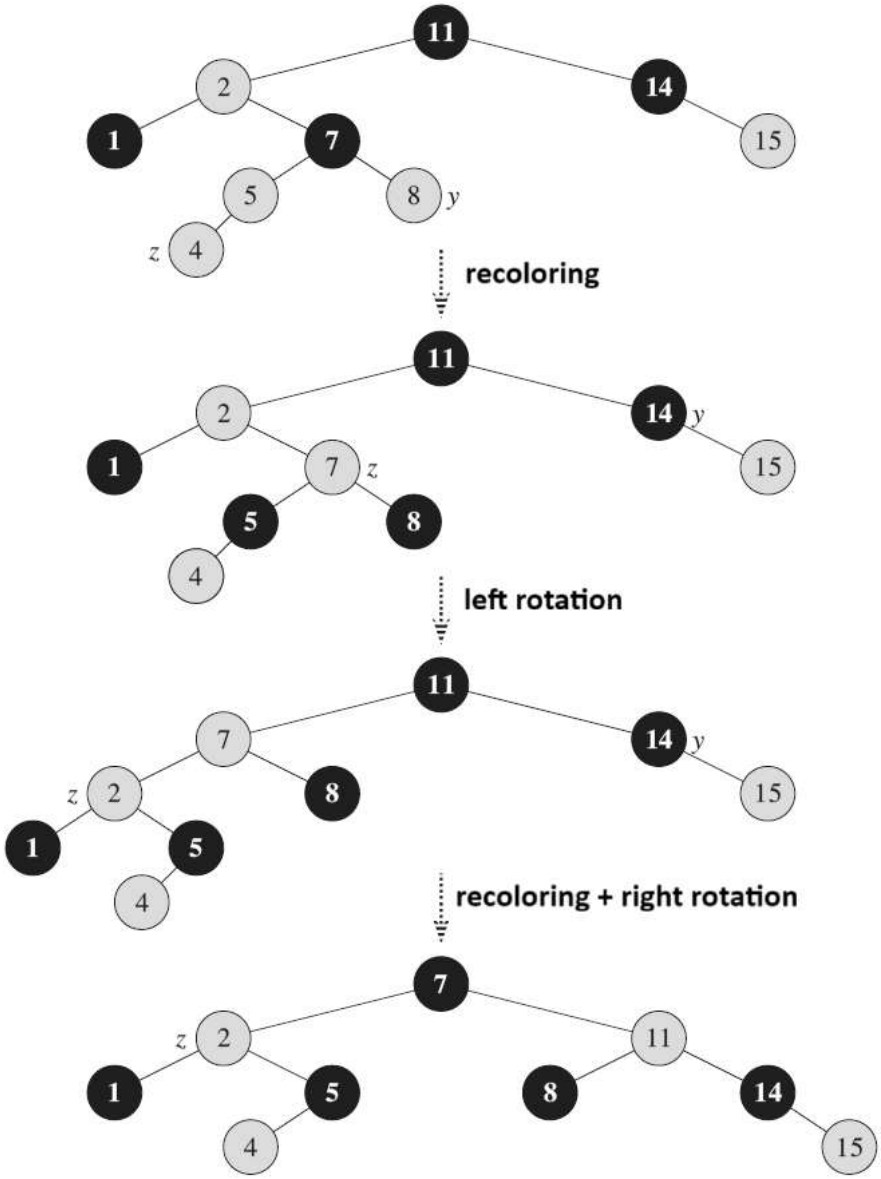
```
def RBInsertFixup(T, z):
  while z.p.color == RED:

    if z.p == z.p.p.left:
      y = z.p.p.right
      if y.color == RED:
        z.p.color = BLACK
        y.color = BLACK
        z.p.p.color = RED
        z = z.p.p
      else:
        if z == z.p.right:
          z = z.p
          LeftRotate(T, z)
        z.p.color = BLACK
        z.p.p.color = RED
        RightRotate(T, z.p.p)

    else:  # same as above with "right" and "left" exchanged
      y = z.p.p.left
      if y.color == RED:
        z.p.color = BLACK
        y.color = BLACK
        z.p.p.color = RED
        z = z.p.p
      else:
        if z == z.p.left:
          z = z.p
          RightRotate(T, z)
        z.p.color = BLACK
        z.p.p.color = RED
        LeftRotate(T, z.p.p)

  T.root.color = BLACK
```

Example



Correctness

The correctness of the fixup subroutine essentially corresponds to proving that, after its execution, the resulting RBT does not contain violations. The proof is based on the following loop invariant for the *while* loop in the procedure:

node z is red; if $z.p$ is the root, then $z.p$ is black; if the tree violates any red-black property, then it violates at most one and it is either property 2 (because z is the root) or 4 (because z.p is also red).

At the end of the loop, $z.p$ is black, so the invariant tells us that the only possible remaining violation could be that z is the root but it is red. So ensuring that the root is black solves this possible case as well.

Time complexity

We already know that the first part of the insertion procedure takes at most linear time in the height of the tree, which is guaranteed to be logarithmic $O(\log_2 n)$ in the size for RBTs. The fixup subroutine iterates going upwards from the newly added node, until no more violations exist or the root is reached. Moreover, even though rotations may increase the length of the traversed path, by bringing down some of the nodes involved, it can be proved that no more than 2 rotations are ever required to fix all violations. Therefore, also the subroutine takes at most $O(\log_2 n)$ constant steps, leading to a total time complexity logarithmic in the size of the tree.

Deletion

As it is for BST, the deletion procedure is more complicated, but it still based on the BST one with the addition of another fixup subroutine. So, the procedure differentiates the cases when the node to be deleted has at most one non-*NIL* child or not. In the former case, the node is simply removed, replacing it with its child (possibly *NIL* if both children are *NIL* leaves). When, instead, the node to be delete has two non-*NIL* children, then it is replaced with the minimum of its right subtree, possibly rearranging some other descendent.

Similarly to the insertion procedure, after the deletion of a node, the tree is still guaranteed to be a BST, but may violate some red-black property. Then, the RB-Delete-Fixup subroutine is employed to restore them. However, it is called only when some violation might actually

arise, that is, when a black node has been deleted or moved. In such a case, the fixup subroutine starts from the node that took its place, and then proceed upwards from it.

```
def Delete(T, z):
  origin_color = z.color       # additional instruction
  if z.left == T.nil:
    x = z.right
    Transplant(T, z, z.right)
  elif z.right == T.nil:
    x = z.left
    Transplant(T, z, z.left)
  else:
    y = Minimum(z.right)
    origin_color = y.color     # additional instruction
    x = y.right                # additional instruction
    if y.p == z:
      x.p = y                  # additional instruction
    else:
      Transplant(T, y, y.right)
      y.right = z.right
      y.right.p = y
    Transplant(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color          # additional instruction
  if origin_color == BLACK:    # additional instruction
    RBDeleteFixup(T, x)        # additional instruction

def Transplant(T, u, v):
  if u.p == T.nil:
    T.root = v
  elif u == u.p.left:
    u.p.left = v
  else:
    u.p.right = v
  v.p = u.p                    # changed: executed even if v is NIL
```

After deleting or moving a black node, some red-black property may have been violated, in particular, rules 2, 4 or 5. The idea of the delete-fixup subroutine is based on the following observation: since any violation has been caused by the removal (or displacement) of a black node, whose position has been taken by the node x input to the fixup subroutine, all those violations could be solved by making x black (or *doubly black* if it were so already). Clearly a node cannot be doubly black, thus such idea must be employed in an other way. Indeed, the subroutine "moves" the extra black attribute up the tree, starting from the input node, until:

- the current node that should be made extra black is red, then it is simply colored black;
- the root has been reached, in which case the extra black color can be safely ignored;
- the rotations and recolorings performed while moving upwards solved the remaining violations.

```
def RBDeleteFixup(T, x):
  while x != T.root and x.color == BLACK:

    if x == x.p.left:
      w = x.p.right
      if w.color == RED:
        w.color = BLACK
        x.p.color = RED
        LeftRotate(T, x.p)
        w = x.p.right
      if w.left.color == BLACK and w.right.color == BLACK:
        w.color = RED
        x = x.p
      else:
        if w.right.color == BLACK:
          w.left.color = BLACK
          w.color = RED
          RightRotate(T, w)
          w = x.p.right
        w.color = x.p.color
        x.p.color = BLACK
        w.right.color = BLACK
        LeftRotate(T, x.p)
        x = T.root
```

```
    else:  # same as above with "right" and "left" exchanged
      w = x.p.left
      if w.color == RED:
        w.color = BLACK
        x.p.color = RED
        RightRotate(T, x.p)
        w = x.p.left
      if w.right.color == BLACK and w.left.color == BLACK:
        w.color = RED
        x = x.p
      else:
        if w.left.color == BLACK:
          w.right.color = BLACK
          w.color = RED
          LeftRotate(T, w)
          w = x.p.left
        w.color = x.p.color
        x.p.color = BLACK
        w.left.color = BLACK
        RightRotate(T, x.p)
        x = T.root

  x.color = BLACK
```

Time complexity

Even though several new instructions have been added to the original deletion procedure of BSTs, aside from the fixup call, they all take constant time. So the total time cost of the procedure without the fixup call is still linear in the height of the tree, that for RBTs is $O(\log_2 n)$ logarithmic in the number of nodes. Similarly to the insert-fixup subroutine, the delete-fixup iterates going upwards from the input node, at most up to the root. Moreover, it can be proved that no more than 3 rotations are ever performed. Therefore, the subroutine takes at most $O(\log_2 n)$ constant steps, leading again to a logarithmic total time complexity.