Lab 11, Red Black Tree

Contents

- Review algs4 red-black tree.
- Writing non-recursive red-black tree.

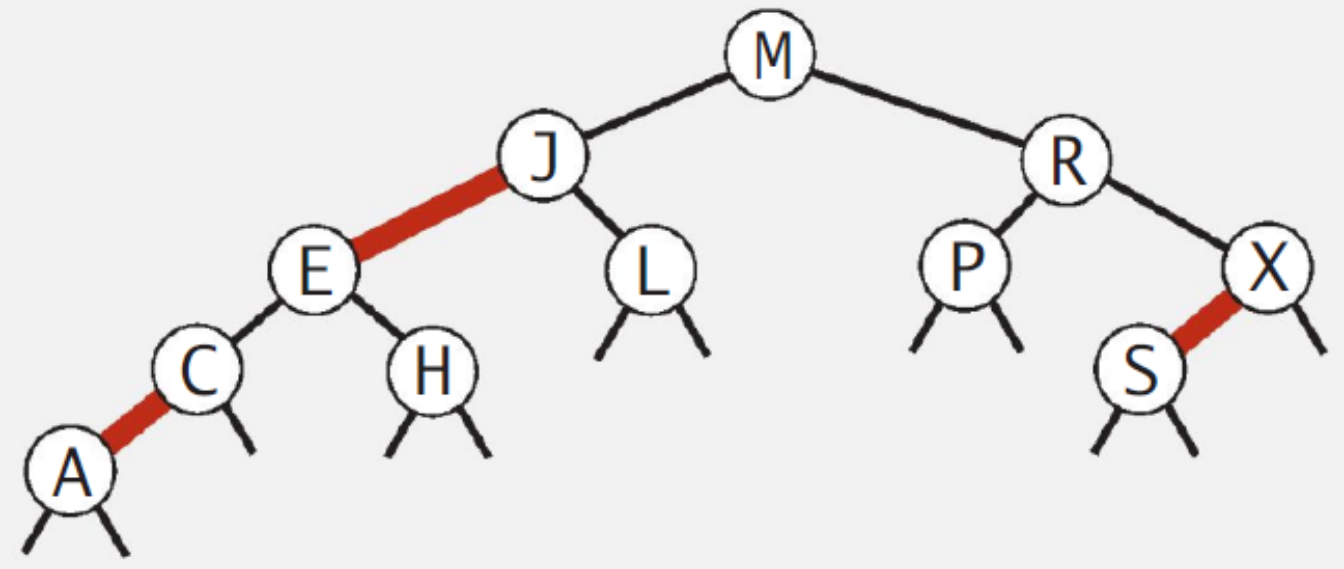
Red-black tree

On the algs4 red-black tree:

https://algs4.cs.princeton.edu/code/edu/princeton/cs/algs4/RedBlackBST.java.html

Red-black tree definition

A red black tree is something like below:



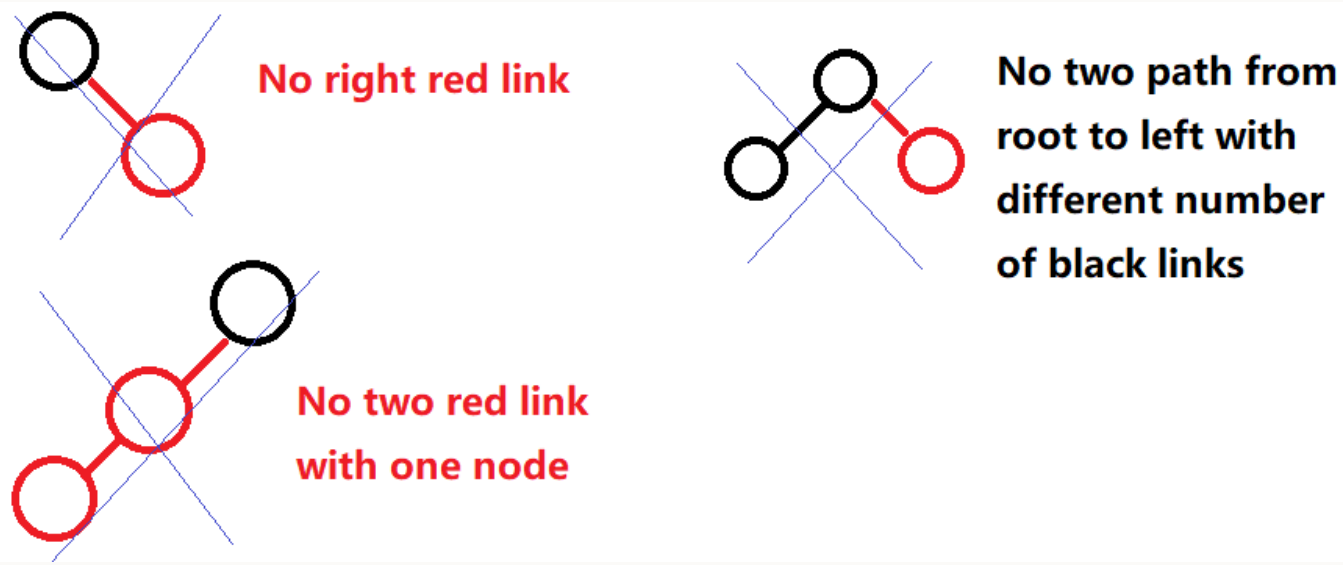
Red-black tree definition

The red-black tree follows the following rules:

- Red links lean left.
- No node has two red links connected to it.
- The tree has perfect black balance : every path from the root to a null link has the same number of black links.

Red-black tree definition

See three restrictions on graph:



Algs4 definition of red-black tree

Below is the data stored in a rb tree in algs4 implementation.

```
private static final boolean RED   = true;
private static final boolean BLACK = false;
private class Node {
    private Key key;           // key
    private Value val;         // associated data
    private Node left, right;  // to left and right subtrees
    private boolean color;     // color of parent link
    private int size;          // subtree count

    public Node(Key key, Value val, boolean color, int size)
      {
        this.key = key;
        this.val = val;
        this.color = color;
        this.size = size;
    }
}
private Node root;     // root of the BST
```

Algs4 definition of red-black tree

From the above definition we see that there is no "link" or "edge" stored in the tree structure. It only stores nodes with colors.

This is because the color of the links is actually stored in nodes.

Red-black tree definition

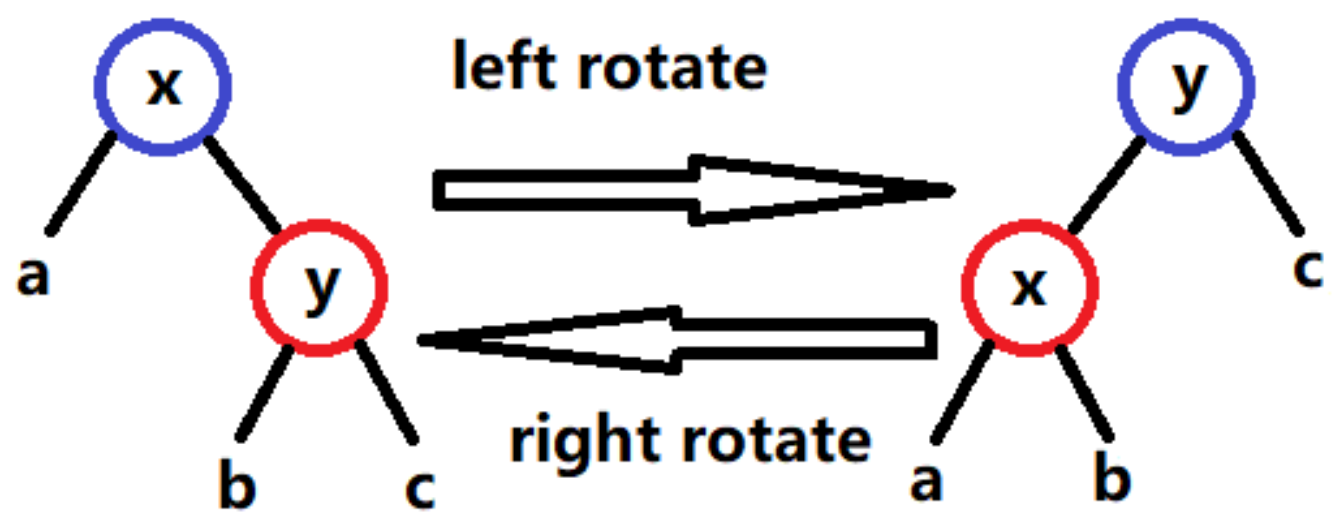
The new rules for red-black tree after we store color in nodes:

- If a node has red child, it is a left child.
- If a node is red, it has no red child.
- The tree has perfect black balance: every path from the root to a null reference has the same number of black node.

Note that different versions of red-black tree may have different definition. Same implementations of red-black tree does not require the first rule.

Left rotate and right rotate

Left rotate and right rotate are very import operations for maintaining the tree structure.



Left rotate and right rotate

Let's review how the are written in algs4 code:

```
private Node rotateLeft(Node h) {
  assert (h != null) && isRed(h.right);
  Node x = h.right;
  h.right = x.left;
  x.left = h;
  x.color = x.left.color;
  x.left.color = RED;
  x.size = h.size;
  h.size = size(h.left) + size(h.right) + 1;
  return x;
}
```

Note that rotate left operation requires that h.right is red.

Left rotate and right rotate

Right rotation is symmetric.

```
private Node rotateRight(Node h) {
  assert (h != null) && isRed(h.left);
  Node x = h.left;
  h.left = x.right;
  x.right = h;
  x.color = x.right.color;
  x.right.color = RED;
  x.size = h.size;
  h.size = size(h.left) + size(h.right) + 1;
  return x;
}
```

Flip color operation

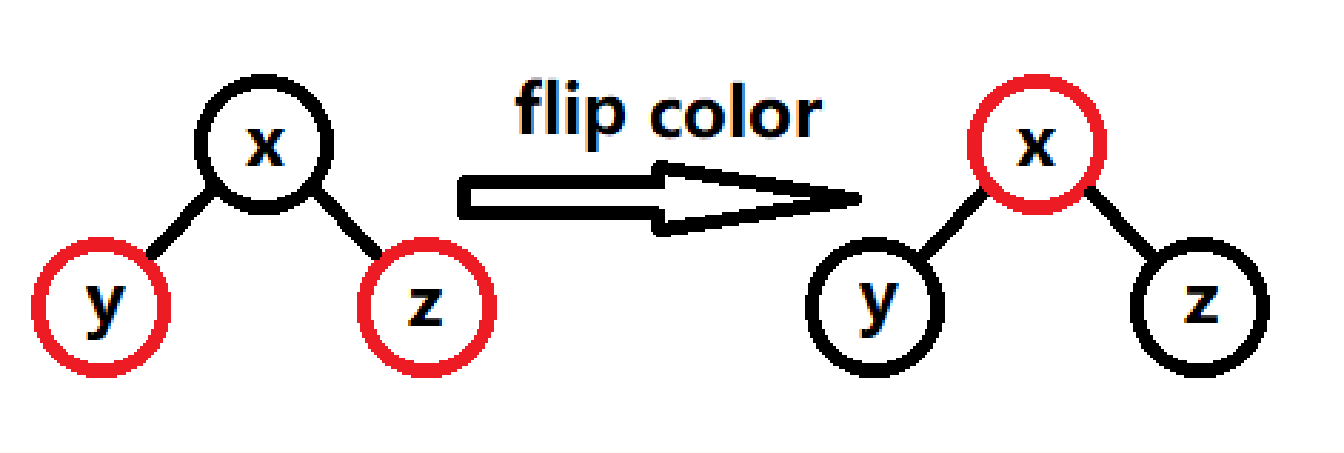
Flip color operation is much simplier.

```
private void flipColors(Node h) {
  h.color = !h.color;
  h.left.color = !h.left.color;
  h.right.color = !h.right.color;
}
```

You should make sure that node h has two children and it has opposite color with them.

Flip color operation

Flip color operation is much simplier.



Implement put operation in red-black tree

The put operation is almost the same as an ordinary binary search tree. But how to maintain the three restrictions mentioned before?

- If a node has red child, it is a left child.
- If a node is red, it has no red child.
- The tree has perfect black balance: every path from the root to a null reference has the same number of black node.

Implement put operation in red-black tree

How to maintain the three restrictions mentioned before?

How to make sure "if a node has red child, it is a left child"?

Use a left rotation.

Implement put operation in red-black tree

How to maintain the three restrictions mentioned before?

How to make sure "if a node is red, it has no red child"?

Use a right rotation followed by flip color.

Implement put operation in red-black tree

How to maintain the three restrictions mentioned before?

How to make sure "every path from the root to a null reference has the same number of black node."?

Insert red nodes.

Implement put operation in red-black tree

A recursive put implementation:

```
// insert the key-value pair in the subtree rooted at h
private Node put(Node h, Key key, Value val) {
    if (h == null) return new Node(key, val, RED, 1);

    int cmp = key.compareTo(h.key);
    if      (cmp < 0) h.left  = put(h.left,  key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else              h.val   = val;

    // fix-up any right-leaning links
    if (isRed(h.right) && !isRed(h.left))      h = rotateLeft(h);
    if (isRed(h.left)  &&  isRed(h.left.left)) h = rotateRight(h);
    if (isRed(h.left)  &&  isRed(h.right))     flipColors(h);
    h.size = size(h.left) + size(h.right) + 1;

    return h;
}
```

Implement put operation in red-black tree

If you still remember the binary search tree we have written last time, we can implement a non-recursive put method based on that.

Implement delete operation in red-black tree

A delete operation could be much harder to implement in red-black tree.

The hard part is that once you delete a black node, the tree is no longer balanced again.

One solution is that you always make sure the node "h" is red or one of its child is red.

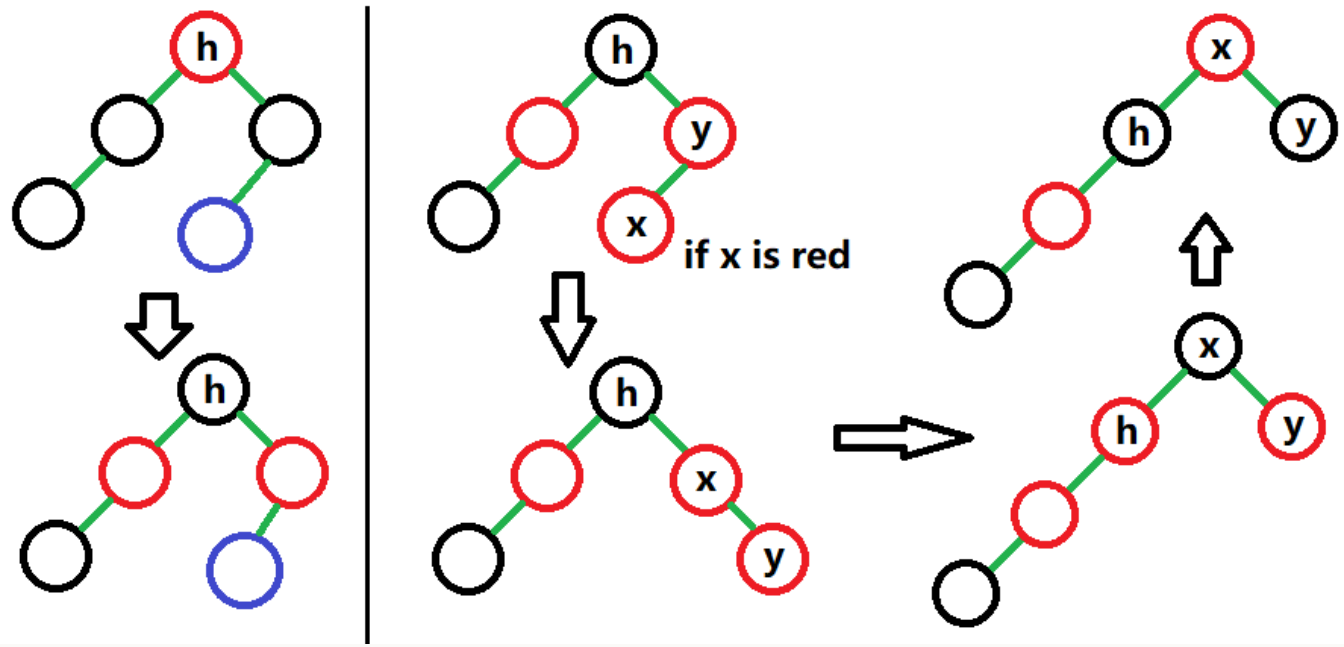
Implement delete operation in red-black tree

Let's see moveRedLeft, it does the following thing:

Assuming that h is red and both h.left and h.left.left are black, make h.left or one of its children red.

Implement delete operation in red-black tree

moveRedLeft:



Implement delete operation in red-black tree

The code looks simple, however:

```
private Node moveRedLeft(Node h) {
  flipColors(h);
  if (isRed(h.right.left)) {
      h.right = rotateRight(h.right);
      h = rotateLeft(h);
      flipColors(h);
  }
  return h;
}
```

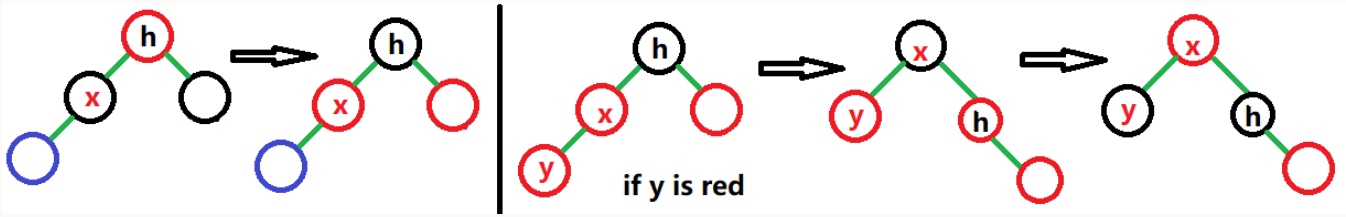
Implement delete operation in red-black tree

Since we have moveRedLeft, we also have moveRedRight:

Assuming that h is red and both h.right and h.right.left are black, make h.right or one of its children red.

Implement delete operation in red-black tree

moveRedLeft and moveRedRight are not symmetric.



Implement delete operation in red-black tree

The right code is also simple.

```
private Node moveRedRight(Node h) {
  flipColors(h);
  if (isRed(h.left.left)) {
      h = rotateRight(h);
      flipColors(h);
  }
  return h;
}
```

Implement delete operation in red-black tree

This time it also needs a delete min operation.

```
private Node deleteMin(Node h) {
  if (h.left == null)
      return null;

  if (!isRed(h.left) && !isRed(h.left.left))
      h = moveRedLeft(h);

  h.left = deleteMin(h.left);
  return balance(h);
}
```

Implement delete operation in red-black tree

Now we have all the tools to implement a delete operation.

```
private Node delete(Node h, Key key) {
  if (key.compareTo(h.key) < 0)  {
      if (!isRed(h.left) && !isRed(h.left.left))
          h = moveRedLeft(h);
      h.left = delete(h.left, key);
  }
  else {
      if (isRed(h.left))
          h = rotateRight(h);
      if (key.compareTo(h.key) == 0 && (h.right == null))
          return null;
      if (!isRed(h.right) && !isRed(h.right.left))
          h = moveRedRight(h);
      if (key.compareTo(h.key) == 0) {
          Node x = min(h.right);
          h.key = x.key;
          h.val = x.val;
          // h.val = get(h.right, min(h.right).key);
          // h.key = min(h.right).key;
```

```
            h.right = deleteMin(h.right);
        }
        else h.right = delete(h.right, key);
    }
    return balance(h);
}
```

Exercise: non-recursive delete

Read the code of the recursive delete in algs4 library.

Write a non-recursive delete.