

CSC 225

Algorithms and Data Structures: I

Rich Little

rlittle@uvic.ca

ECS 516

2-3 tree: implementation?

Direct implementation is complicated, because:

- Maintaining multiple node types is cumbersome.
- Need multiple compares to move down tree.
- Need to move back up the tree to split 4-nodes.
- Large number of cases for splitting.

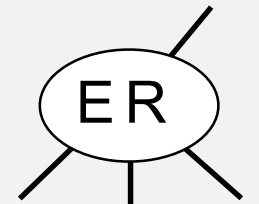
fantasy code

```
public void put(Key key, Value val)
{
    Node x = root;
    while (x.getTheCorrectChild(key) != null)
    {
        x = x.getTheCorrectChildKey();
        if (x.is4Node()) x.split();
    }
    if (x.is2Node()) x.make3Node(key, val);
    else if (x.is3Node()) x.make4Node(key, val);
}
```

Bottom line. Could do it, but there's a better way.

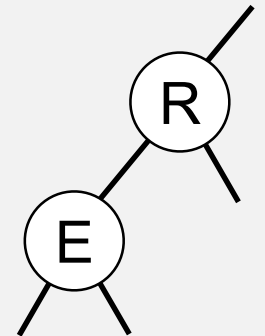
How to implement 2-3 trees with binary trees?

Challenge. How to represent a 3 node?



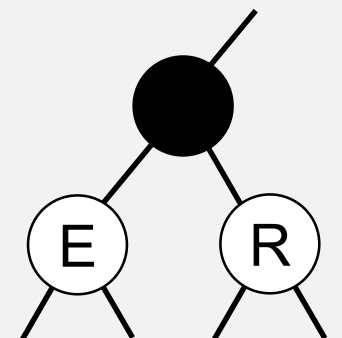
Approach 1: regular BST.

- No way to tell a 3-node from a 2-node.
- Cannot map from BST back to 2-3 tree.



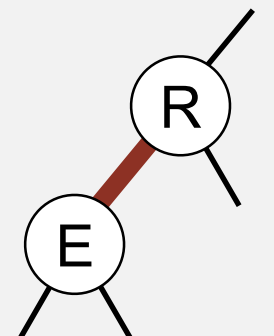
Approach 2: regular BST with "glue" nodes.

- Wastes space, wasted link.
- Code probably messy.



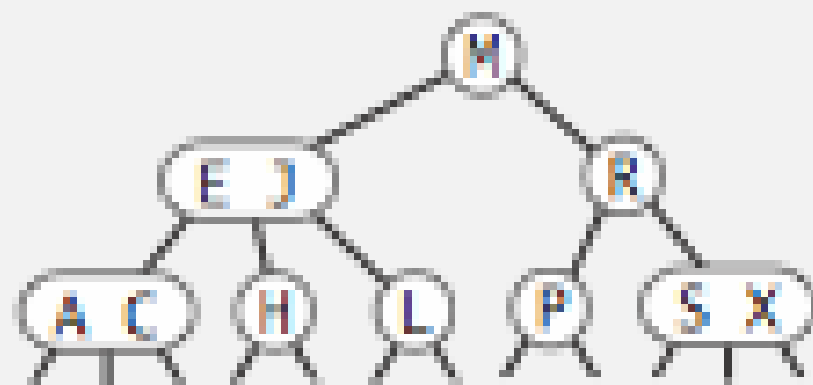
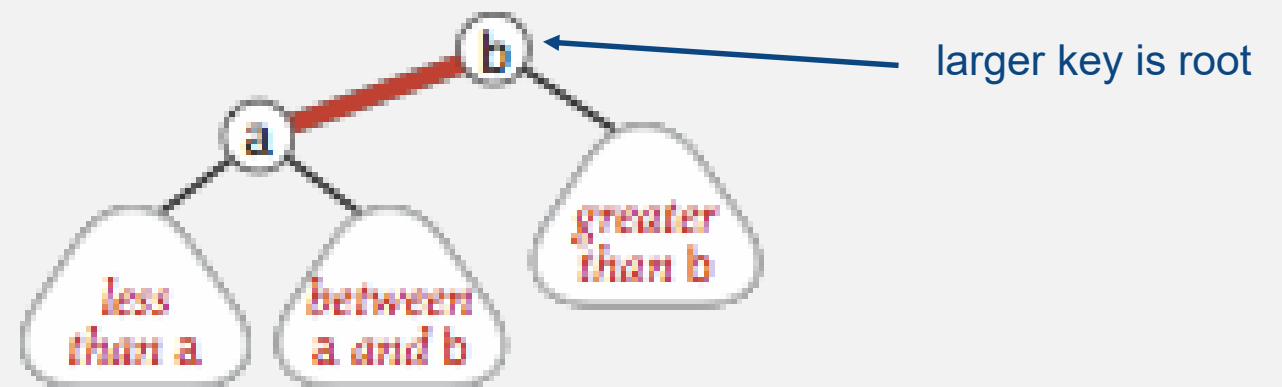
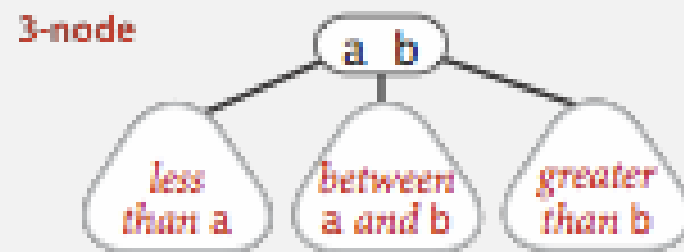
Approach 3: regular BST with red "glue" links.

- Widely used in practice.
- Arbitrary restriction: red links lean left.

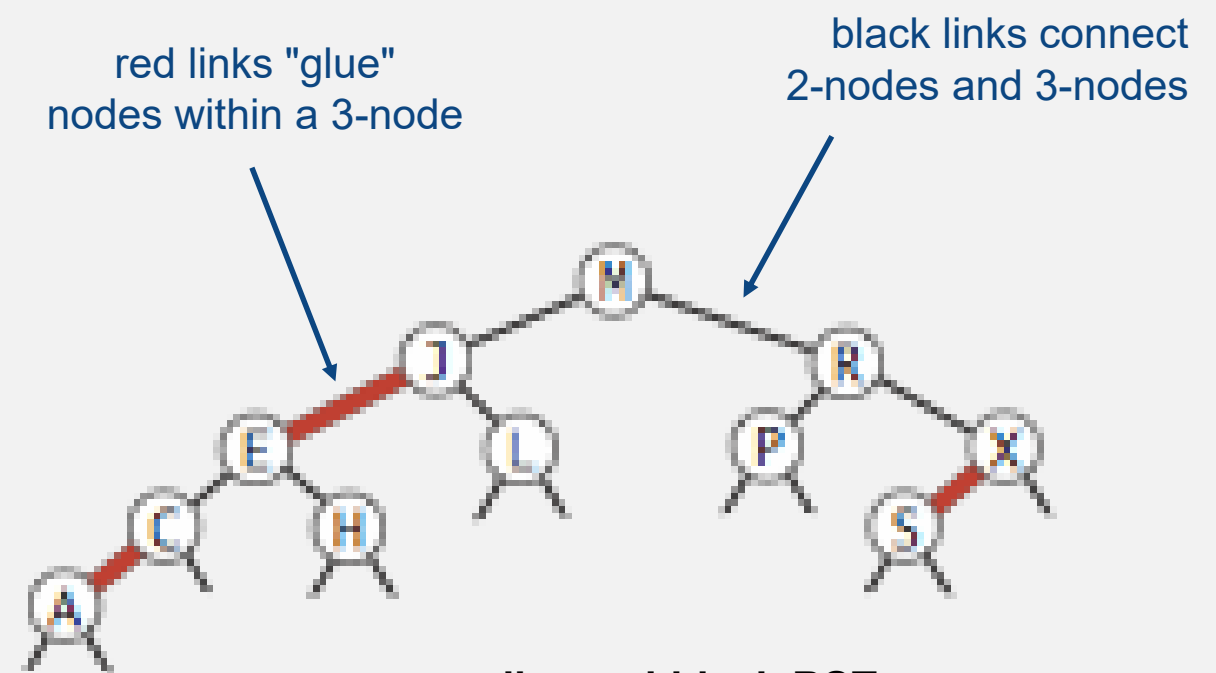


Left-leaning red-black BSTs (Guibas-Sedgwick 1979 and Sedgwick 2007)

1. Represent 2-3 tree as a BST.
2. Use "internal" left-leaning links as "glue" for 3-nodes.



2-3 tree

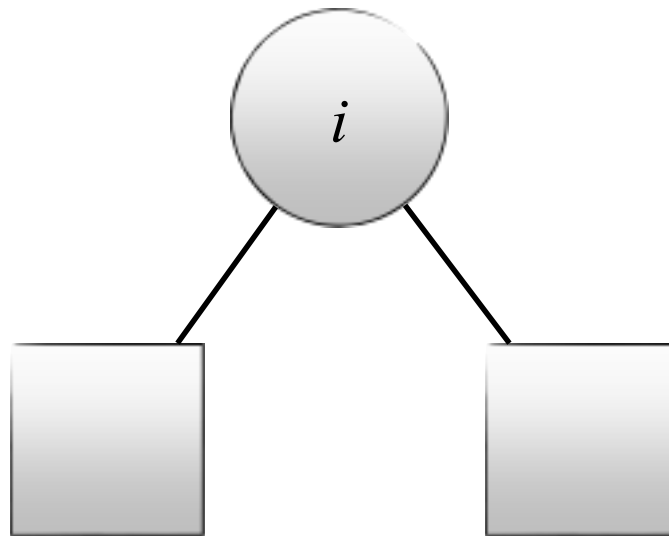


corresponding red-black BST

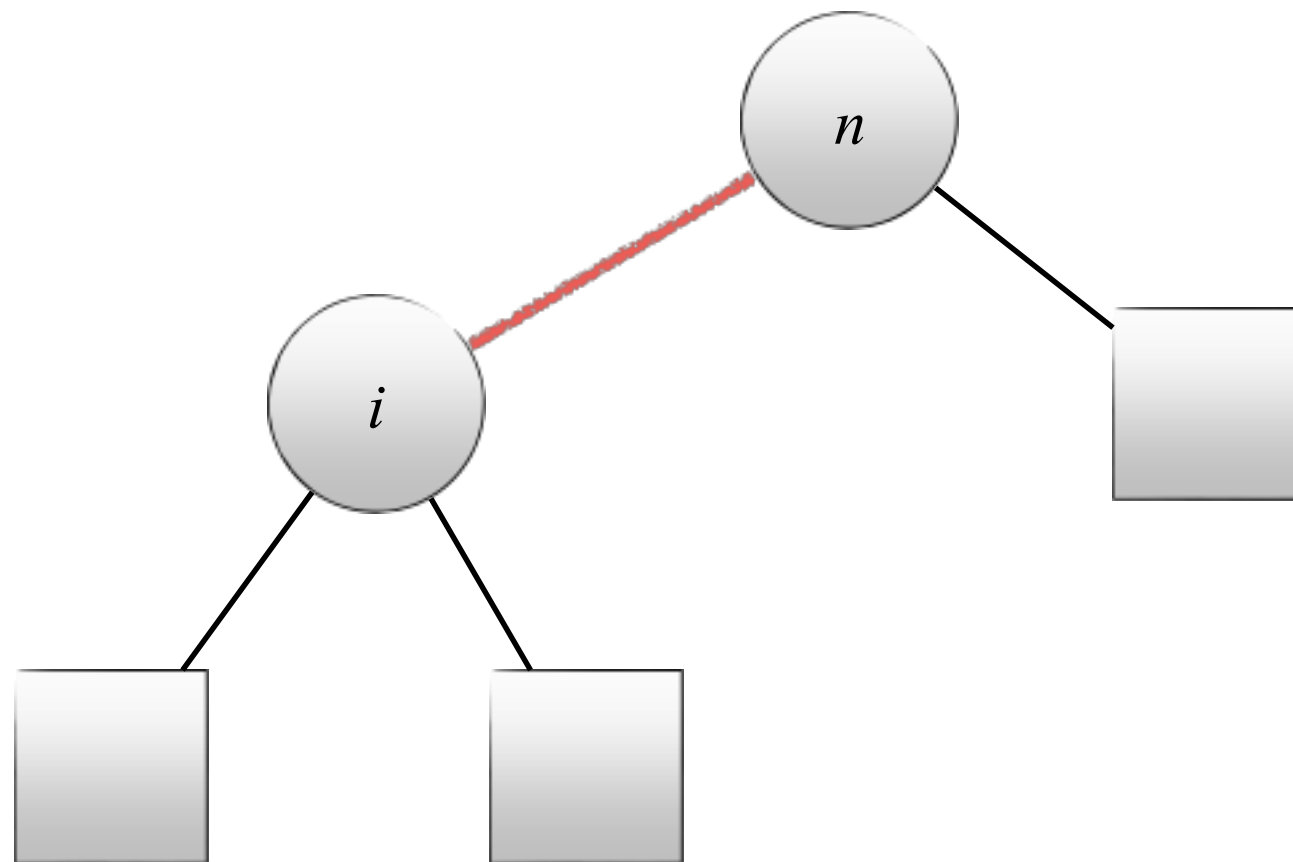
Definition: Red-Black Tree

- A red-black tree is a binary search tree where each link/edge is either red or black. Further
 - All red links lean left
 - No node has two red links connected to it
 - The tree has a balance: every path from the root to a leaf has the same number of black links
 - Links to leaves are black

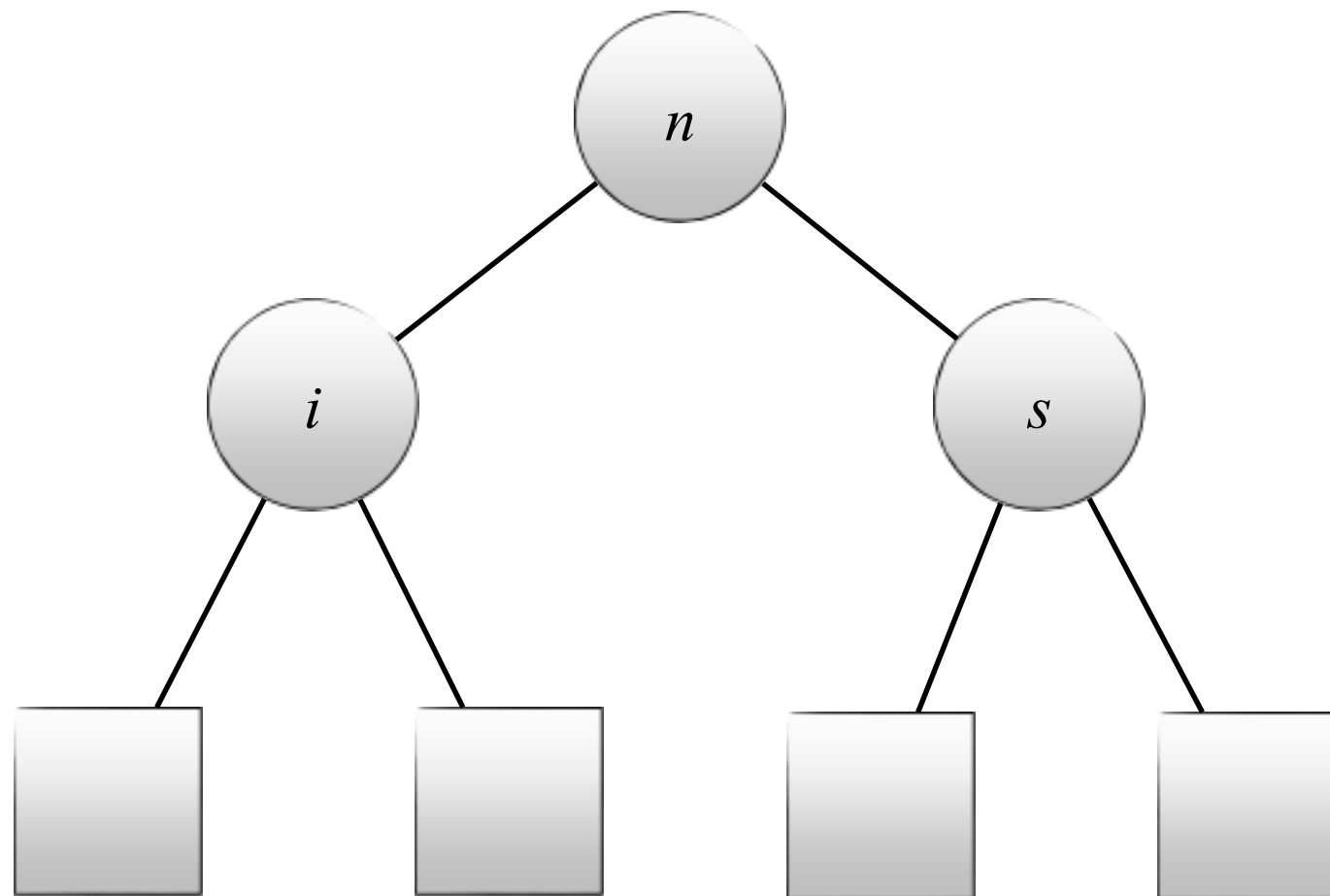
Example 1: red-black tree?



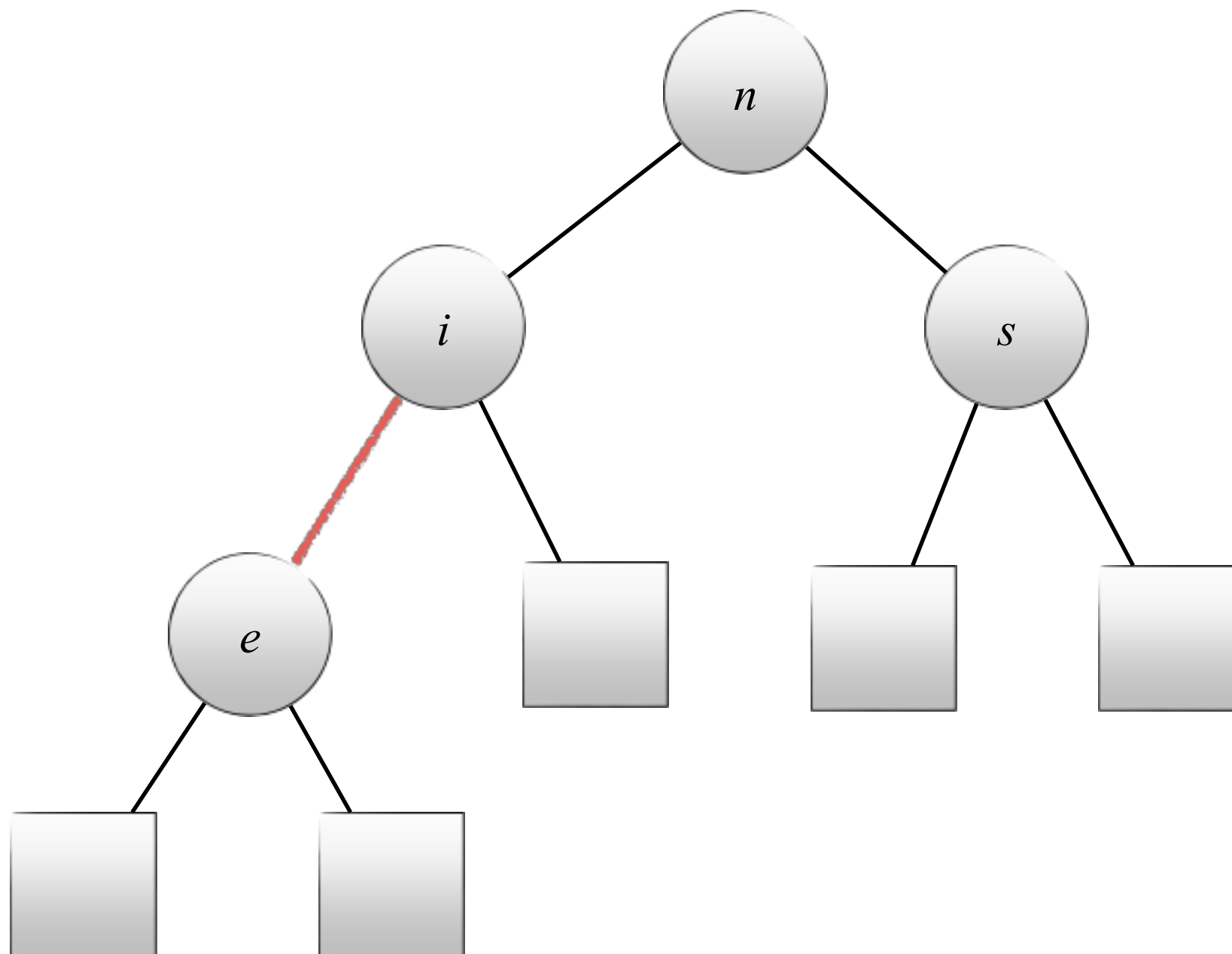
Example 2: red-black tree?



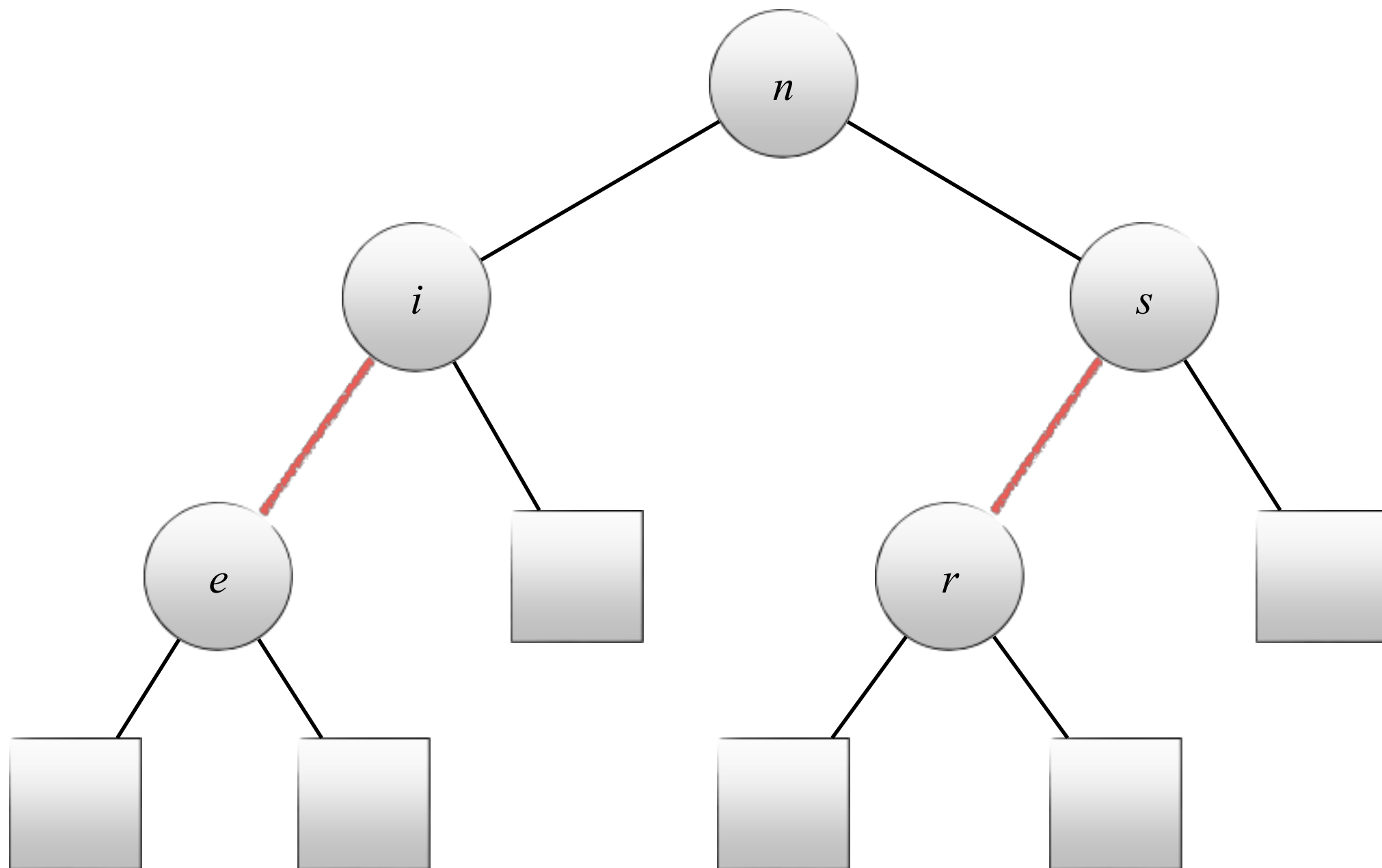
Example 3: red-black tree?



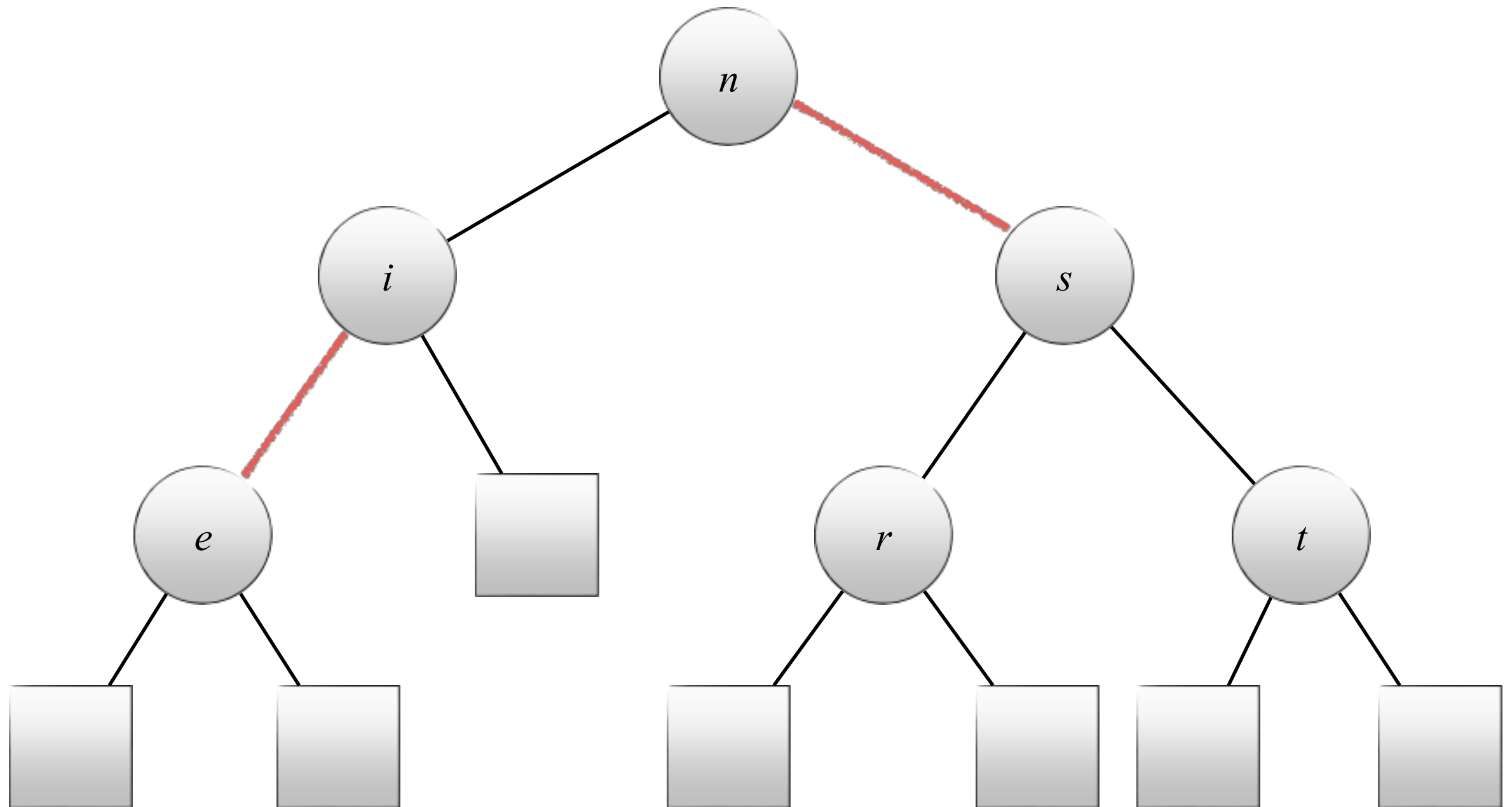
Example 4: red-black tree?



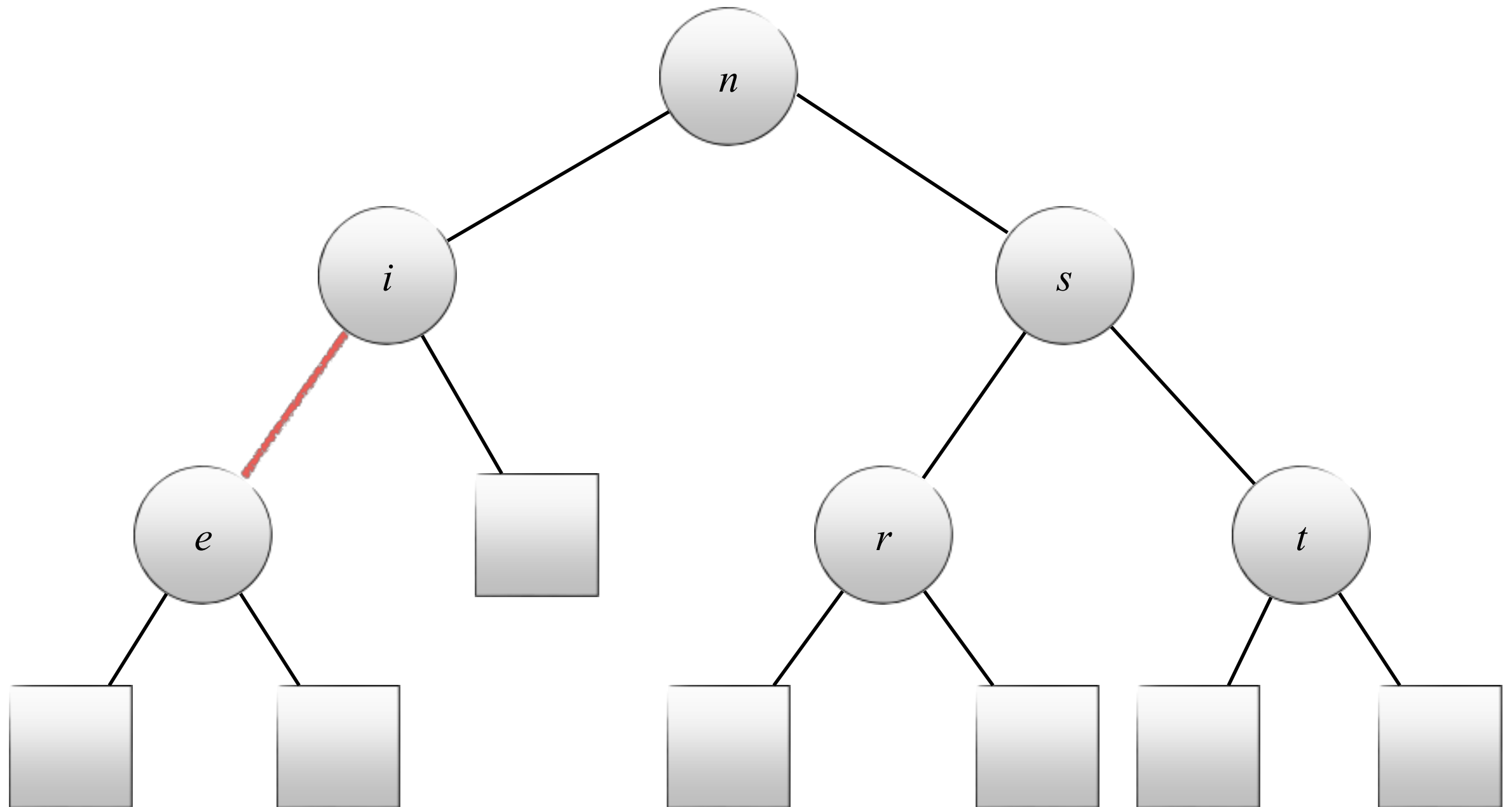
Example 5: red-black tree?



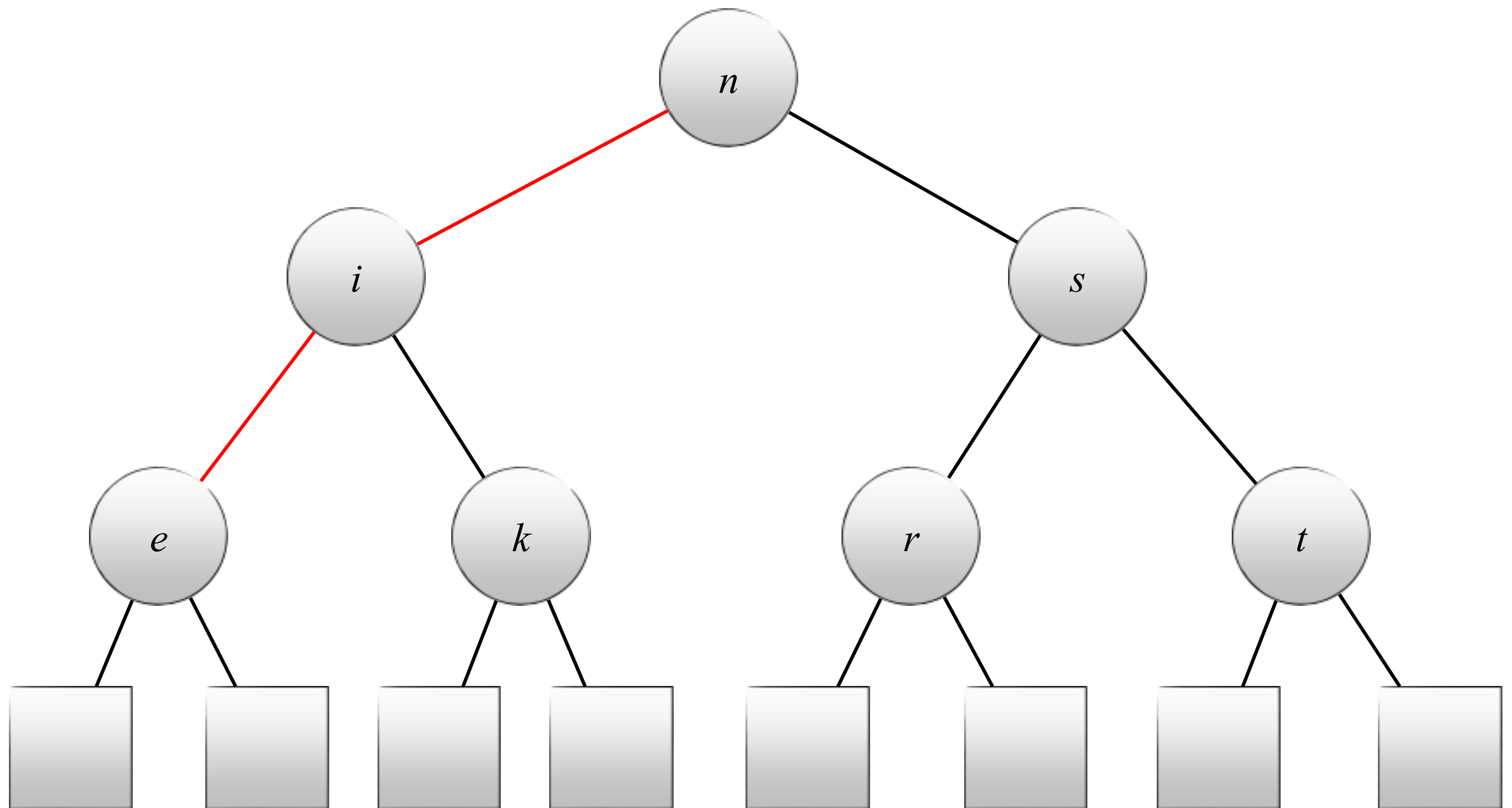
Example 6: red-black tree?



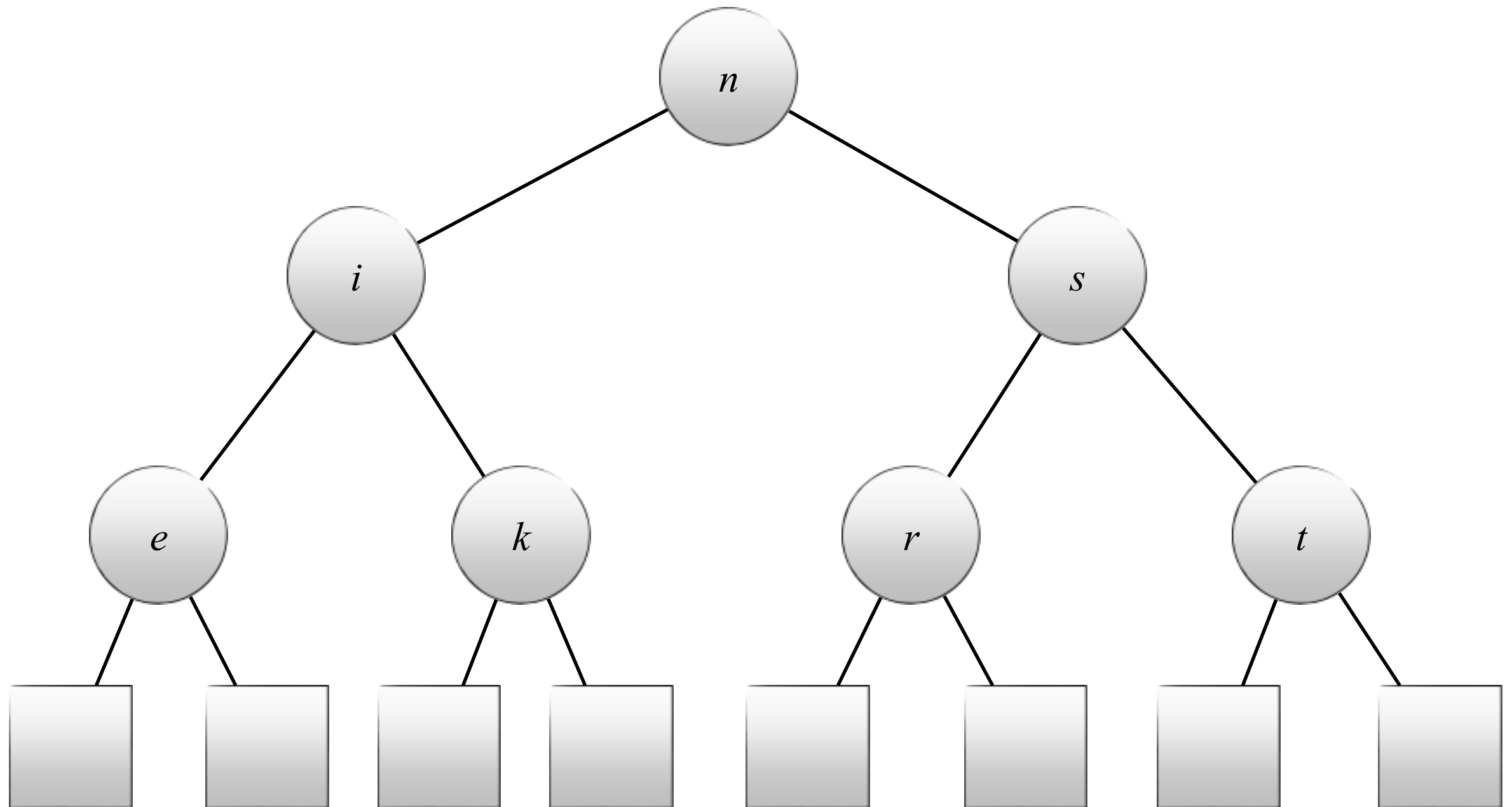
Example 7: red-black tree?



Example 8: red-black tree?



Example 9: red-black tree?



Red-black BST representation

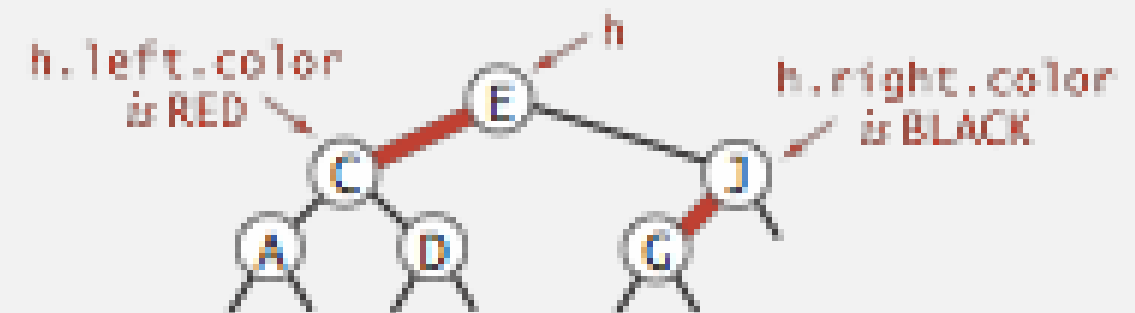
Each node is pointed to by precisely one link (from its parent)
can encode color of links in nodes.

```
private static final boolean RED  = true;
private static final boolean BLACK = false;
```

```
private class Node
{
    Key key;
    Value val;
    Node left, right;
    boolean color; // color of parent link
}
```

```
private boolean isRed(Node x)
{
    if (x == null) return false;
    return x.color == RED;
}
```

← null links are black

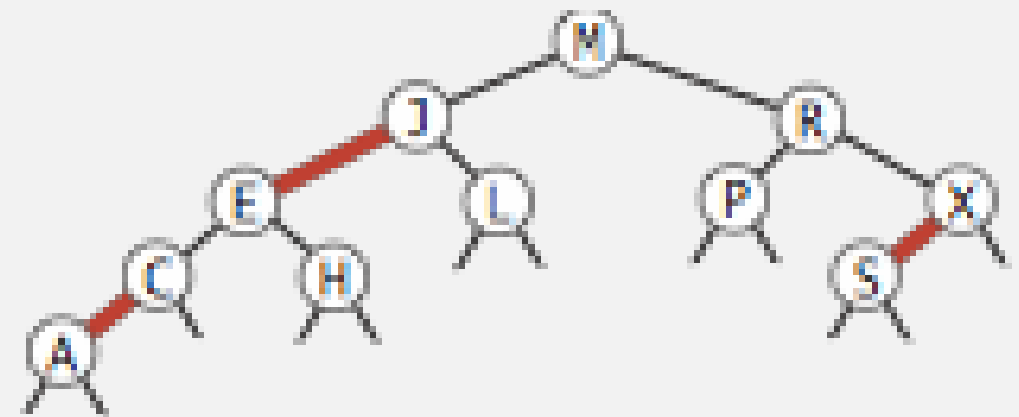


Search implementation for red-black BSTs

Observation. Search is the same as for elementary BST (ignore color).

but runs faster
because of better balance

```
public Val get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if (cmp < 0)
            x = x.left;
        else if (cmp > 0)
            x = x.right;
        else
            return x.val;
    }
    return null;
}
```



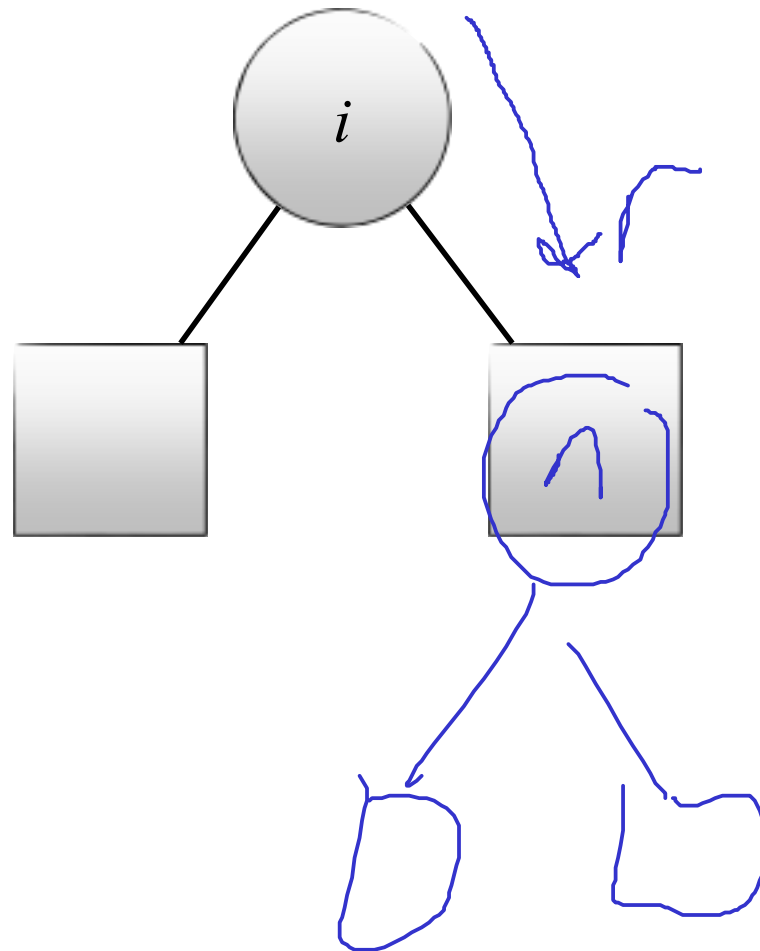
Inserting into a red-black tree

- Insert just as in BST
- but: link/edge to new node is red
- rotations and color flipping (depending on case)

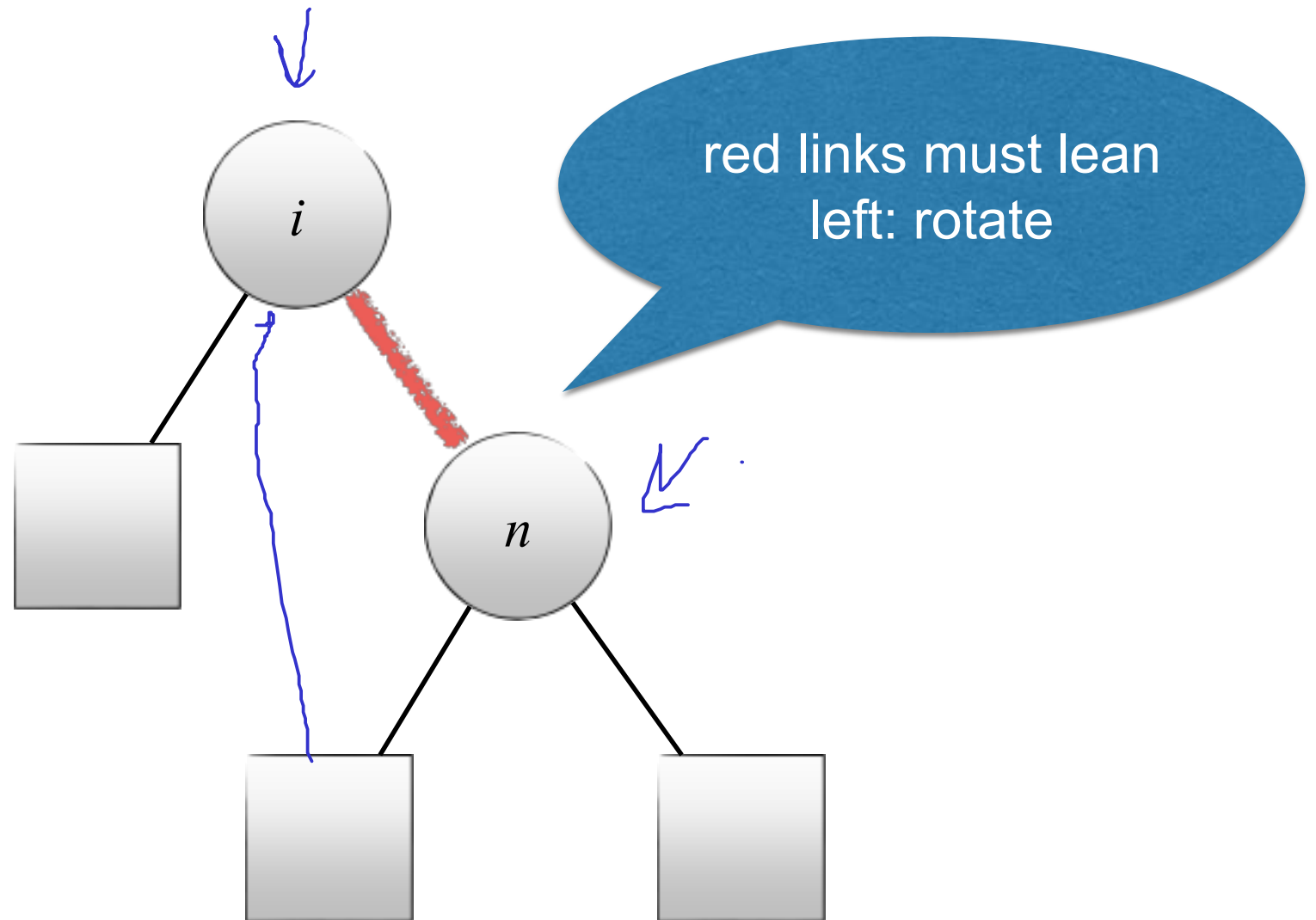
Insert key i into empty tree



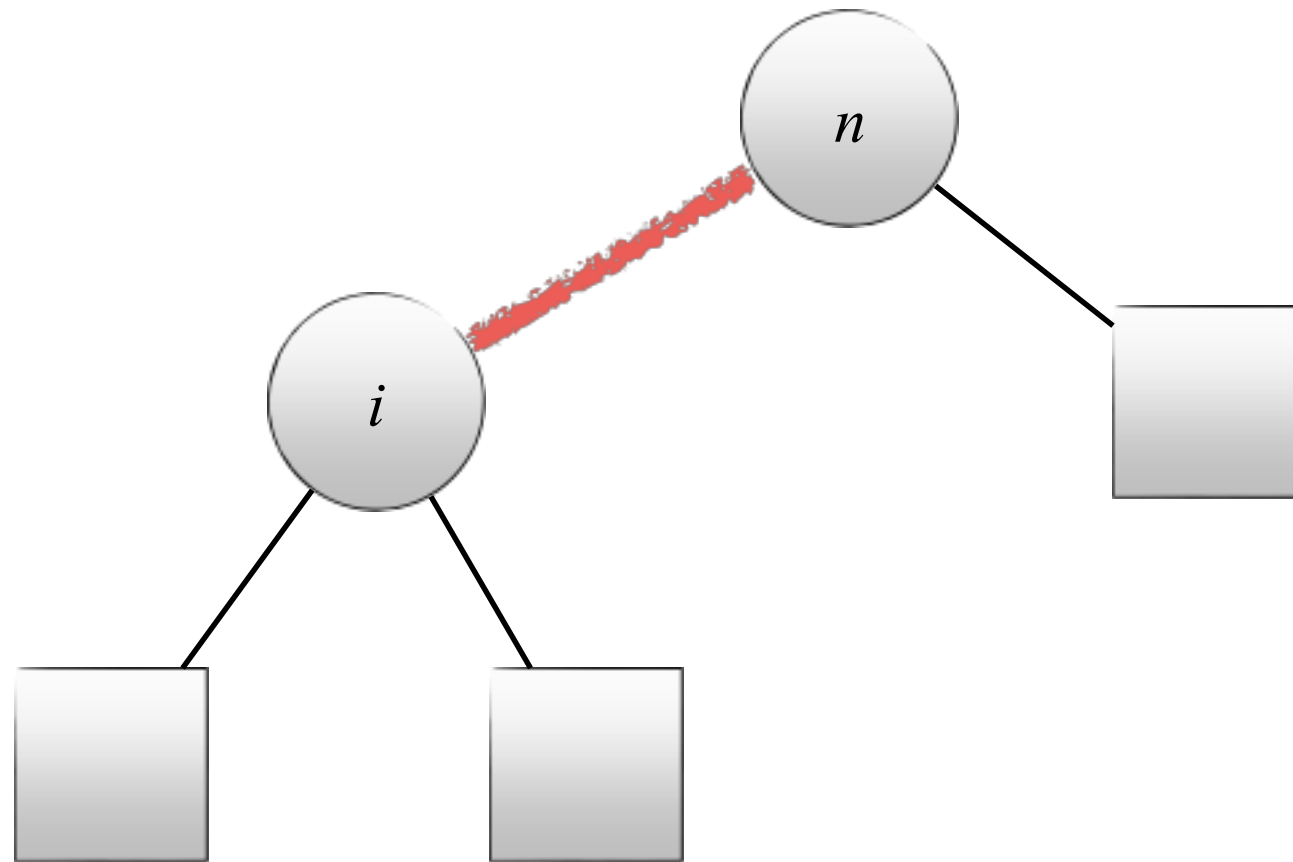
Insert key n



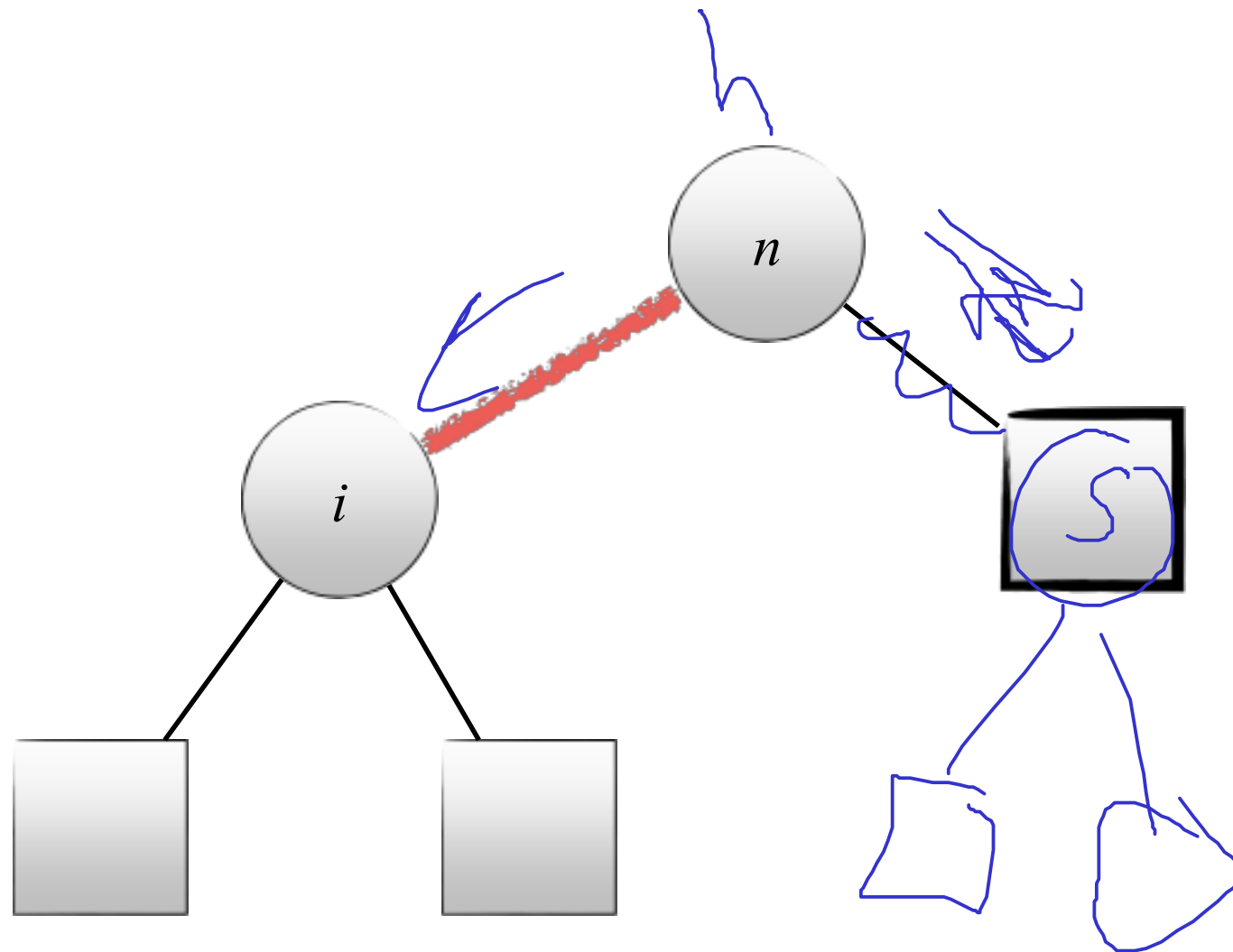
Link to new node is red



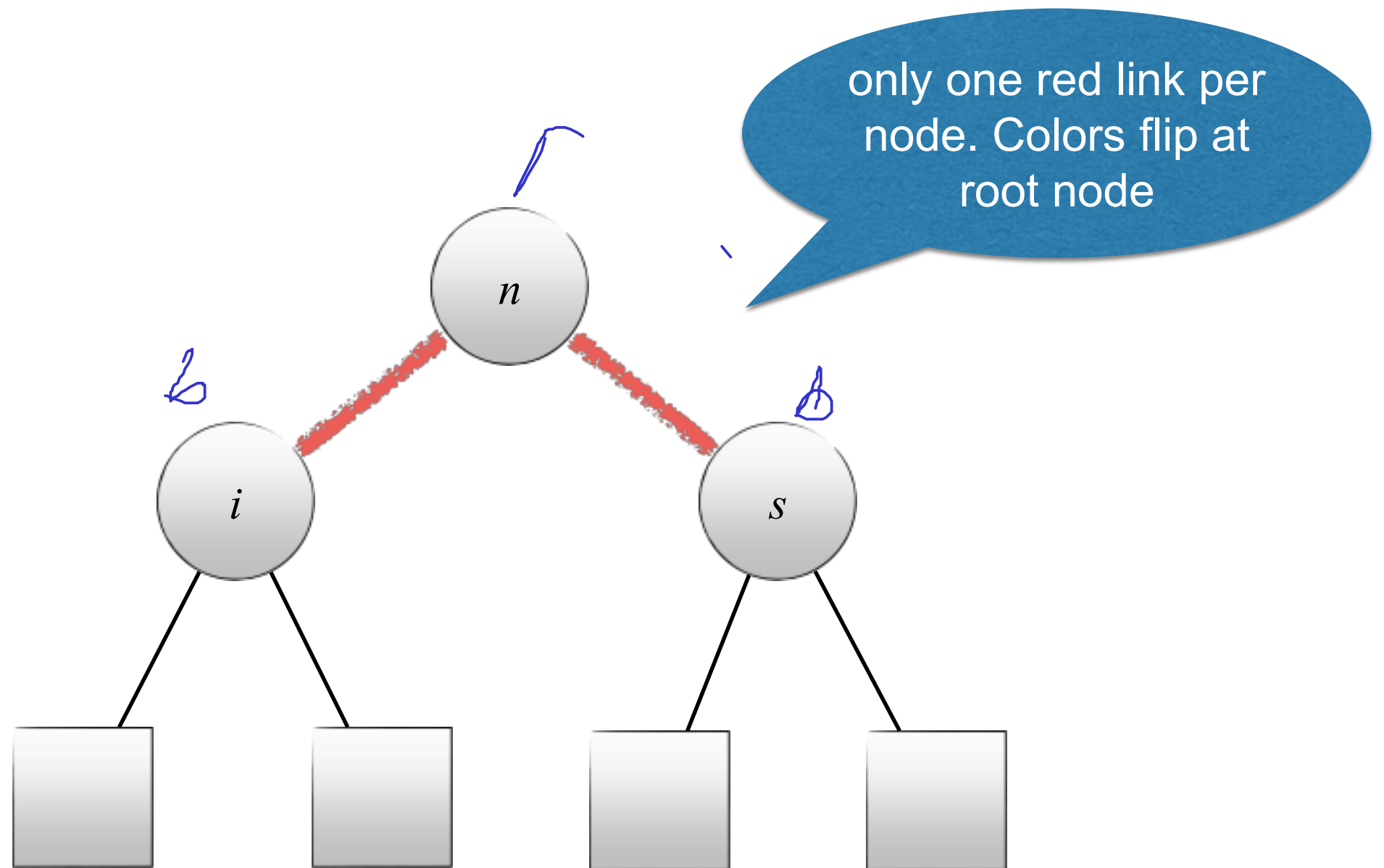
After rotation



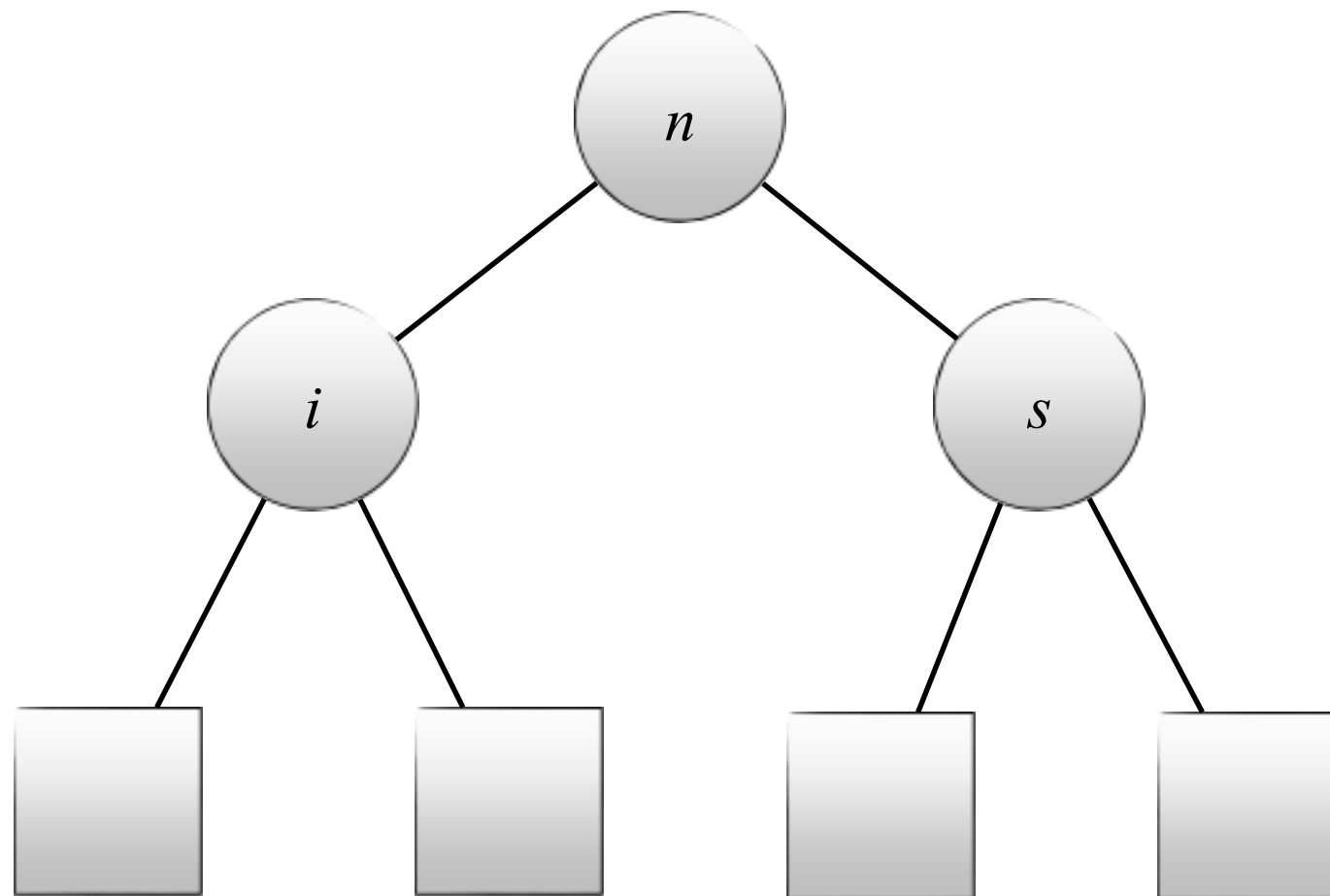
Insert key s



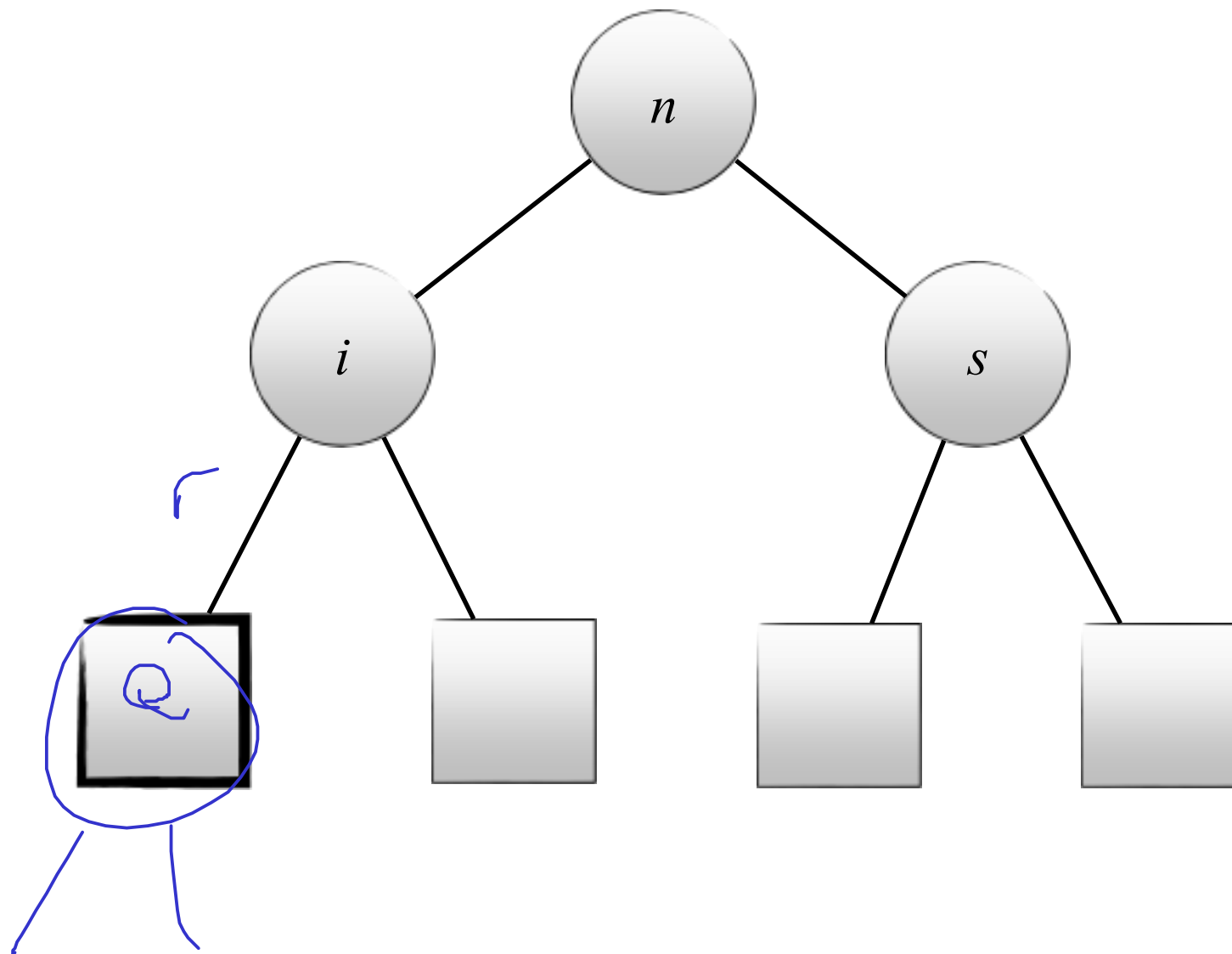
Insert key s , new link is red



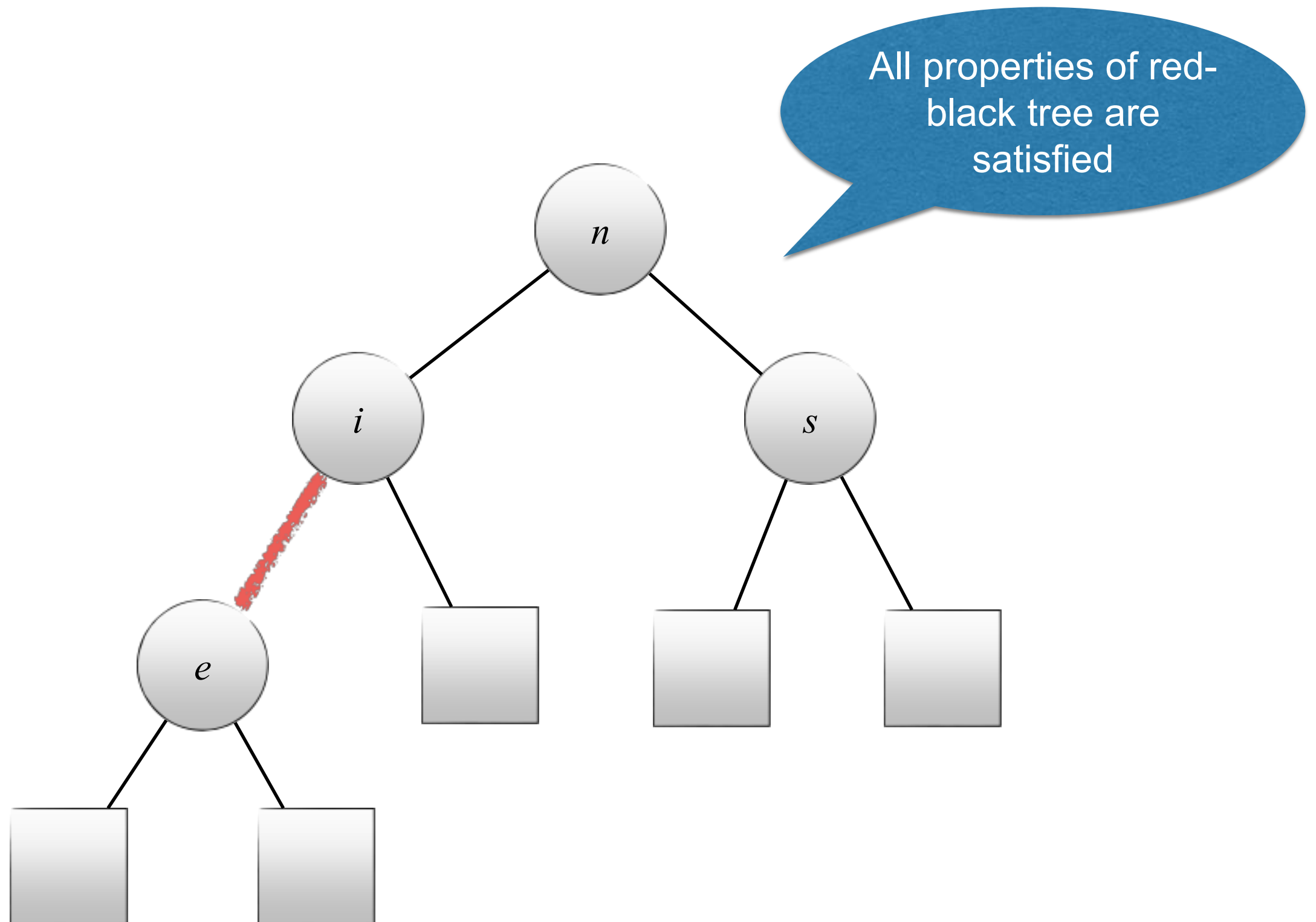
Colors flipped to black



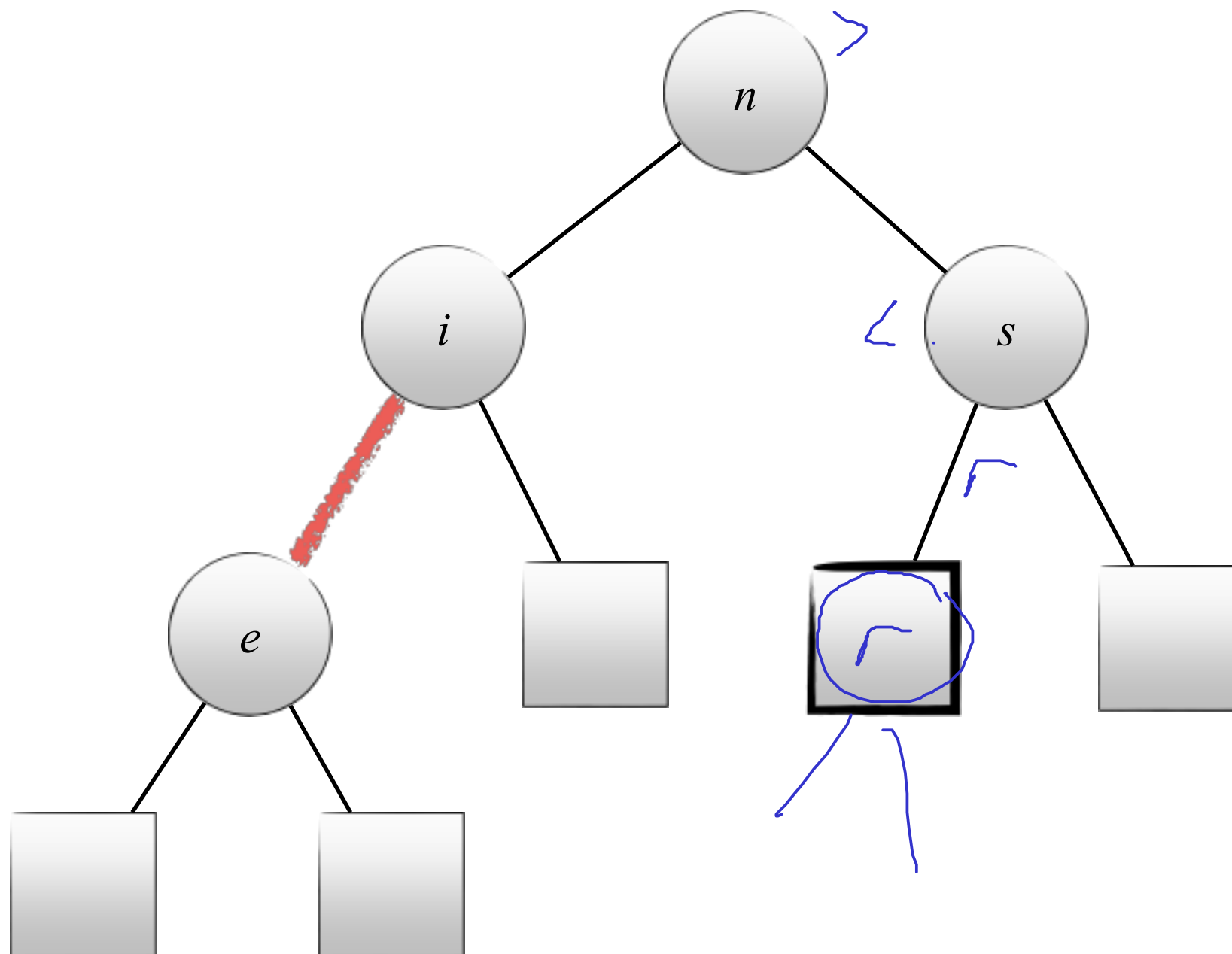
Insert key e



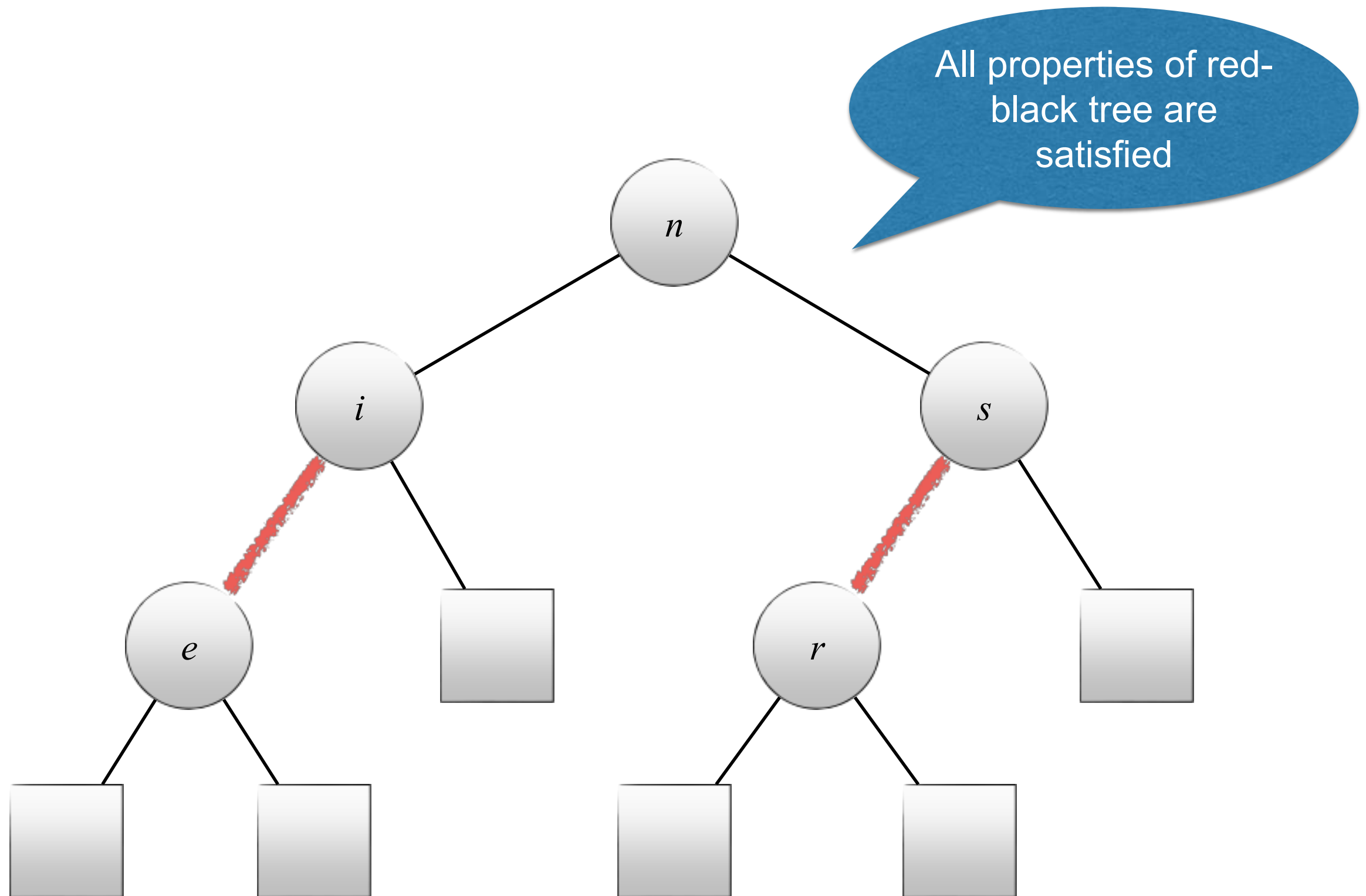
Insert key e , new link is red



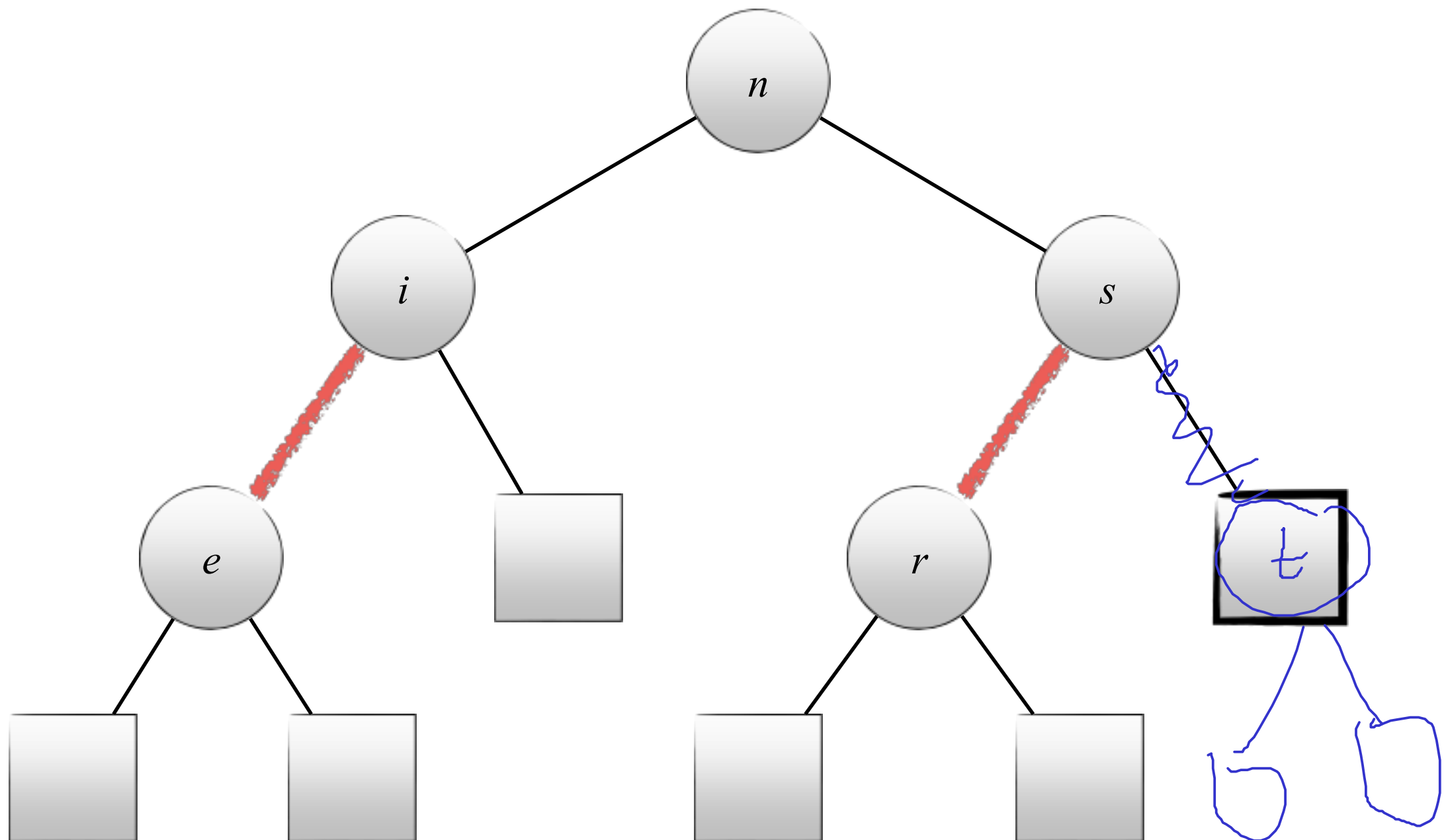
Insert key r



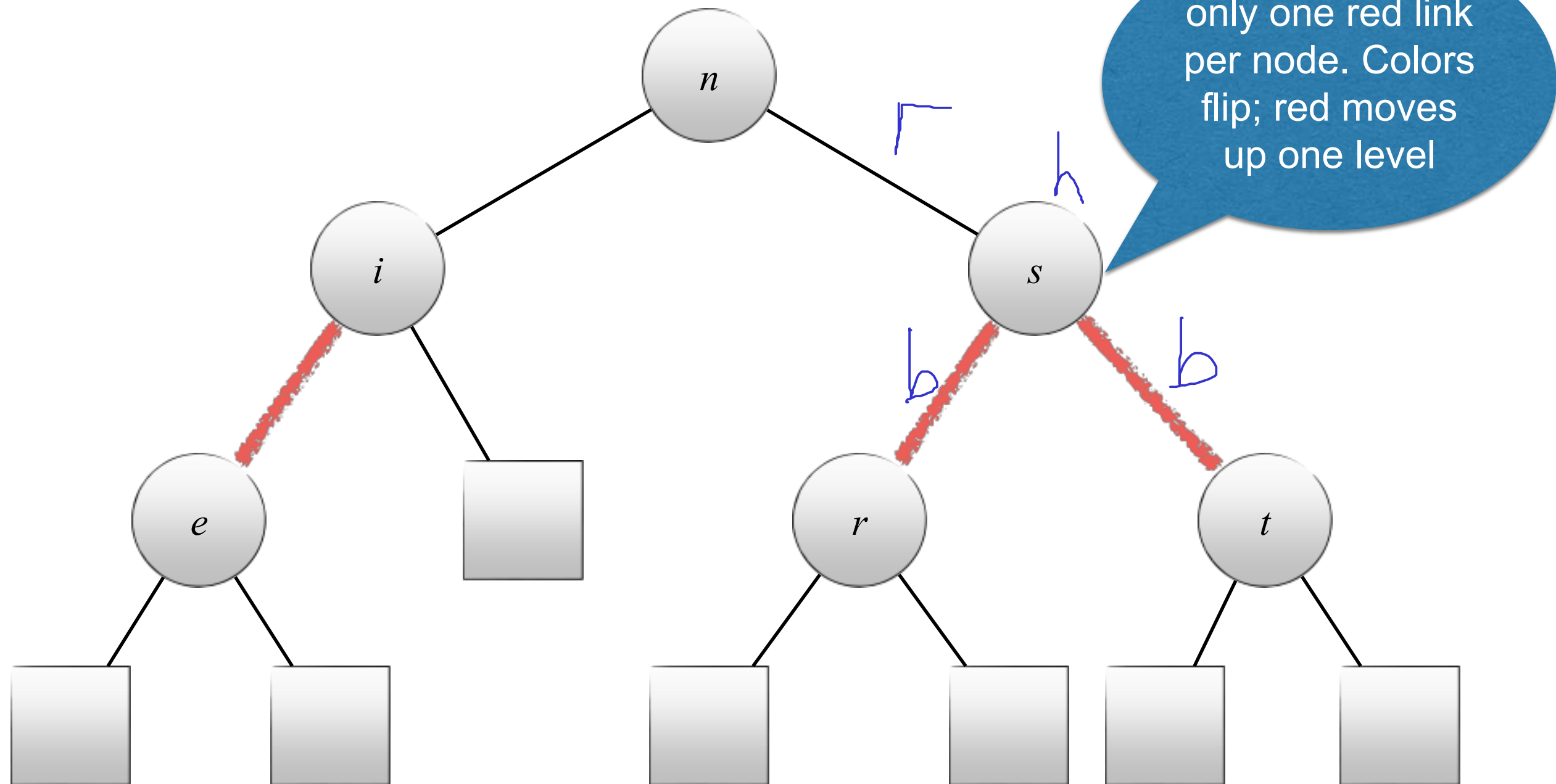
Insert key r , new link is red



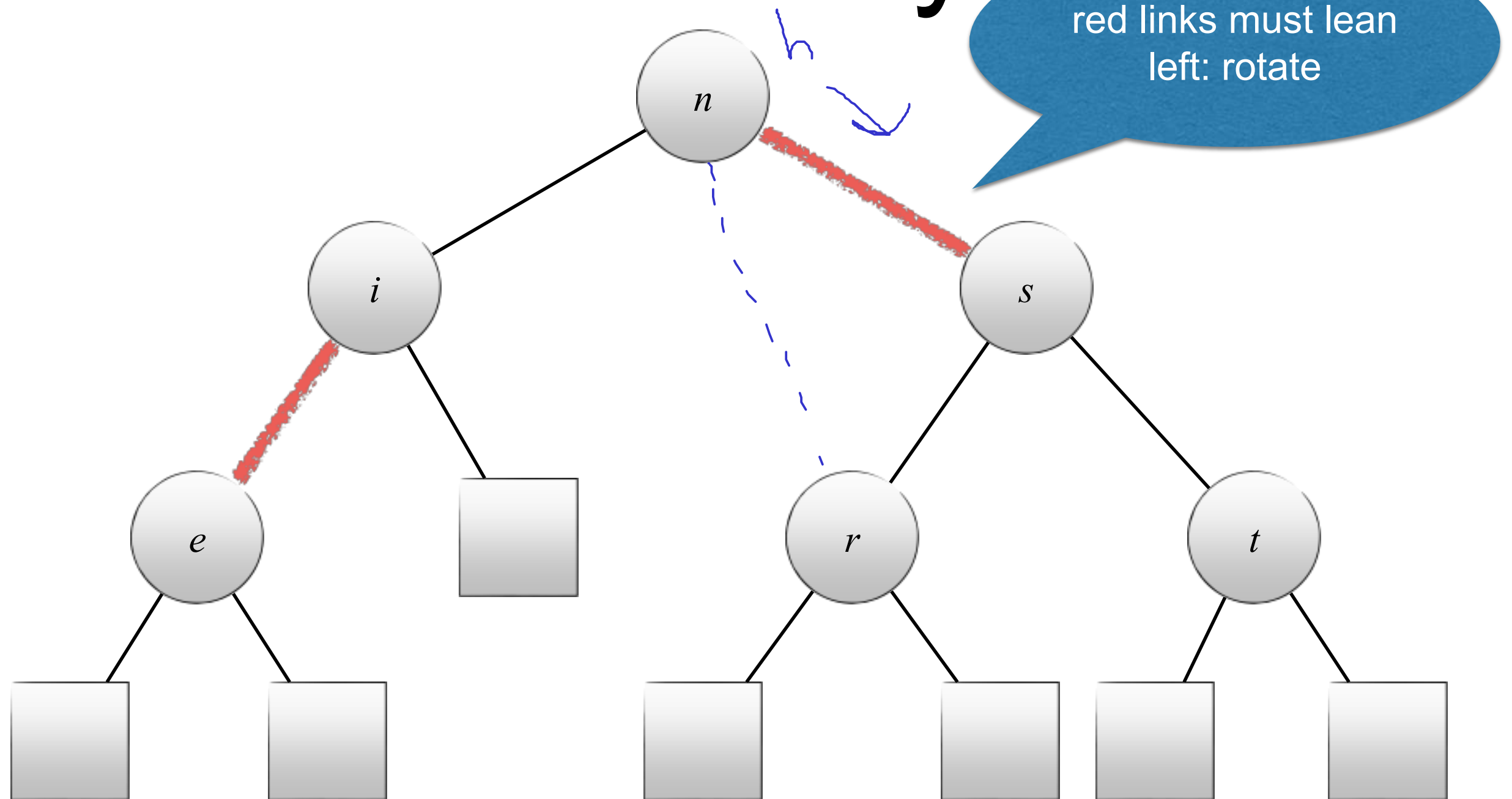
Insert key t



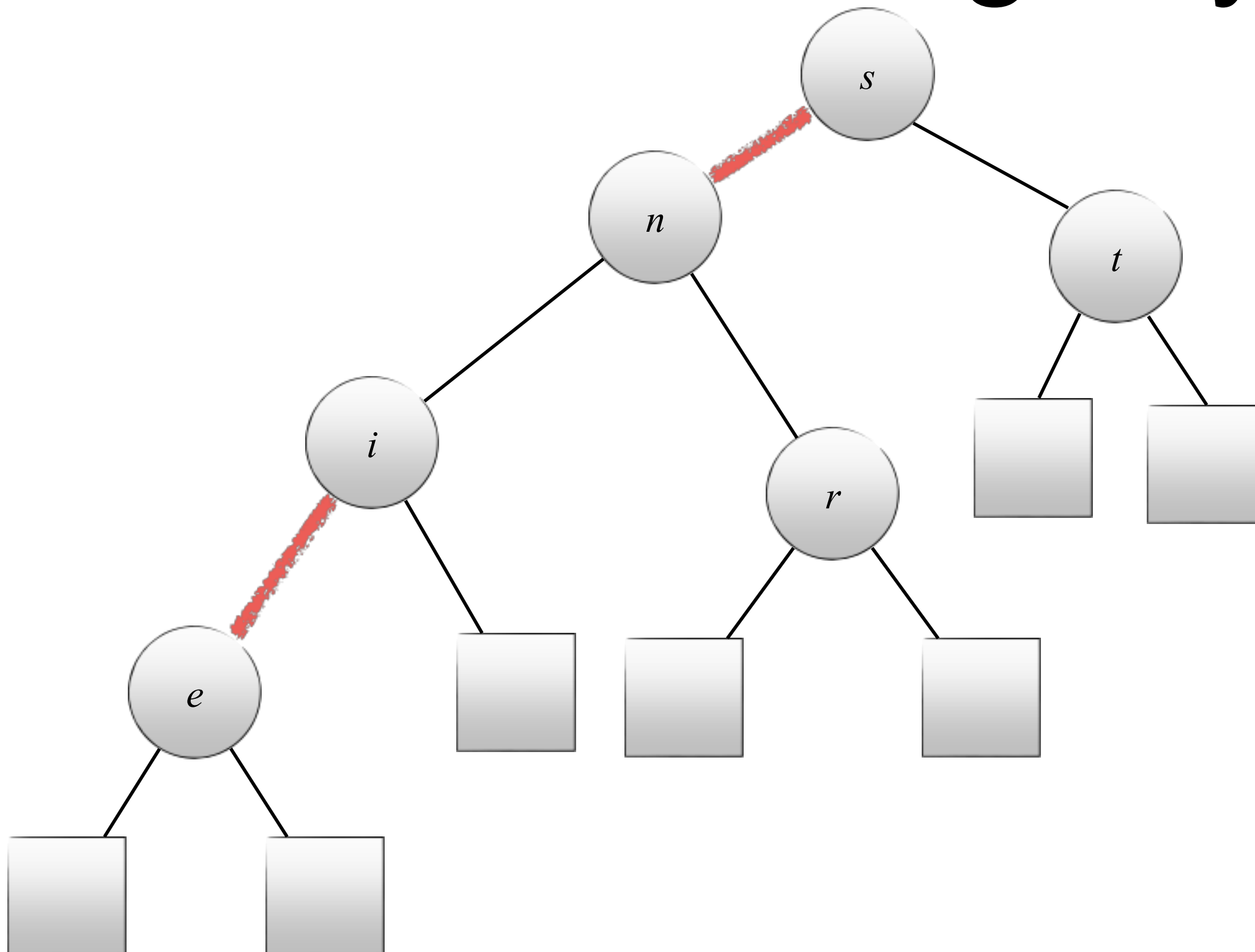
Insert key t , new link is red



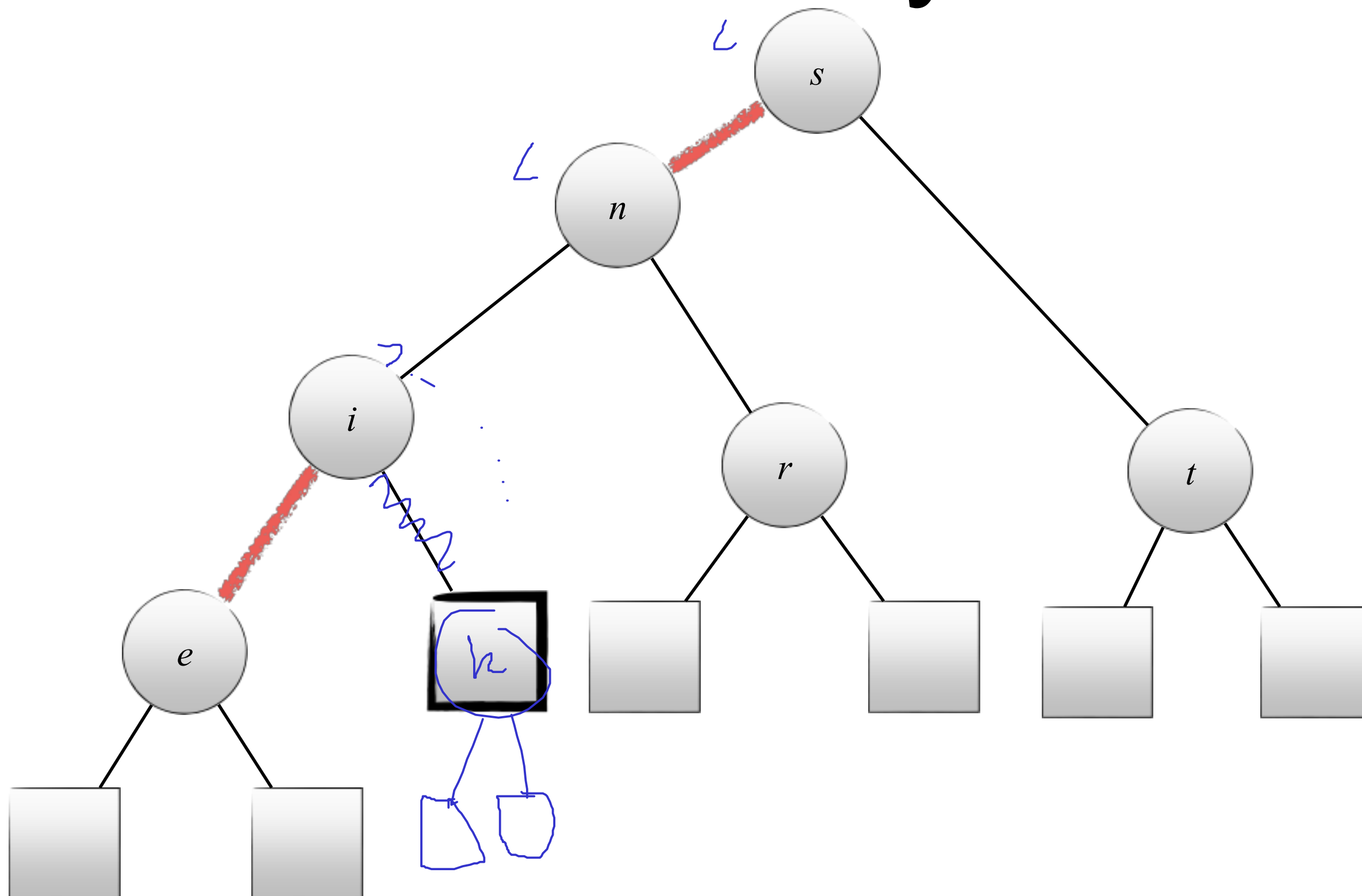
Insert key t



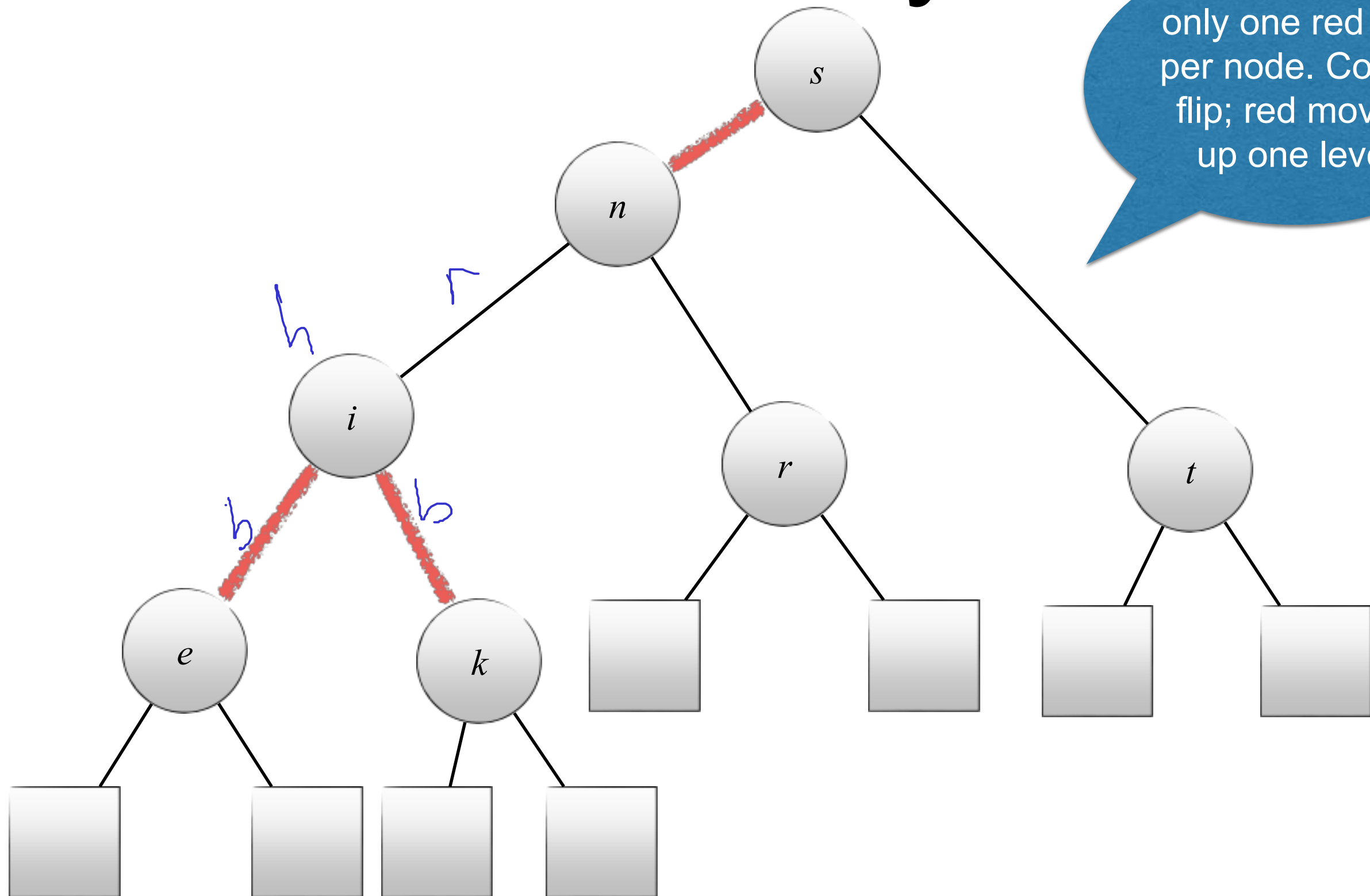
After inserting key t



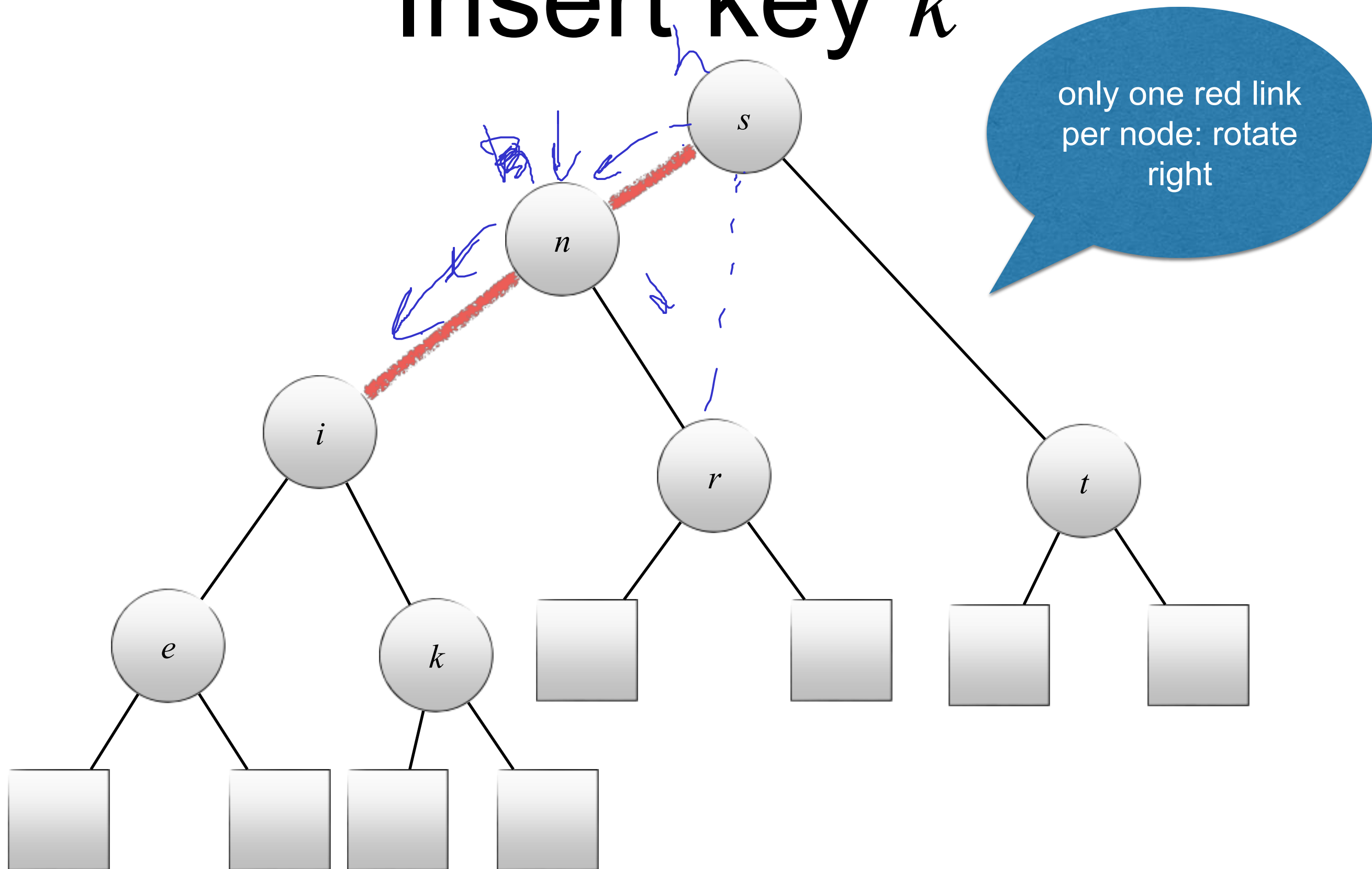
Insert key k



Insert key k

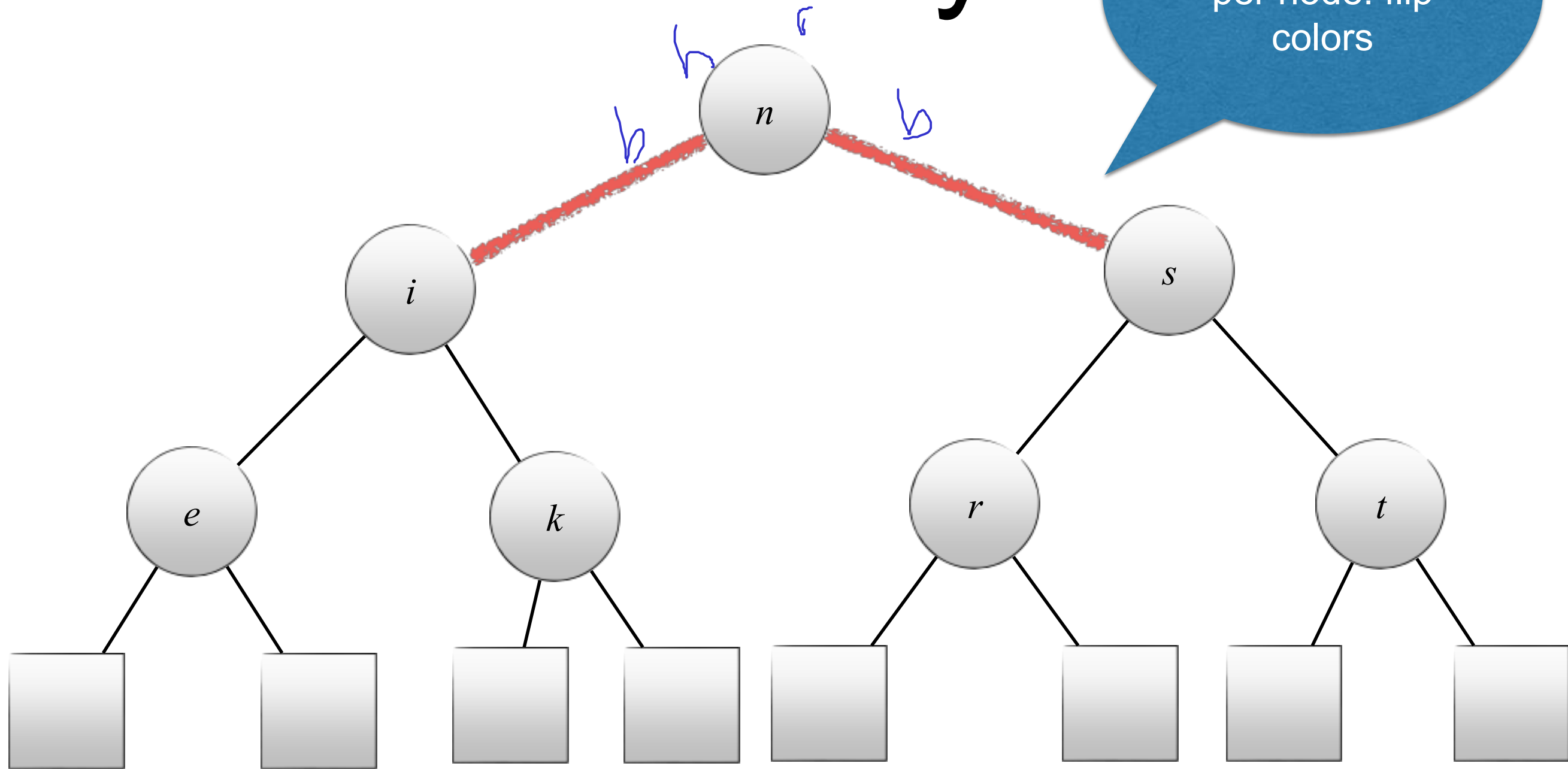


Insert key k

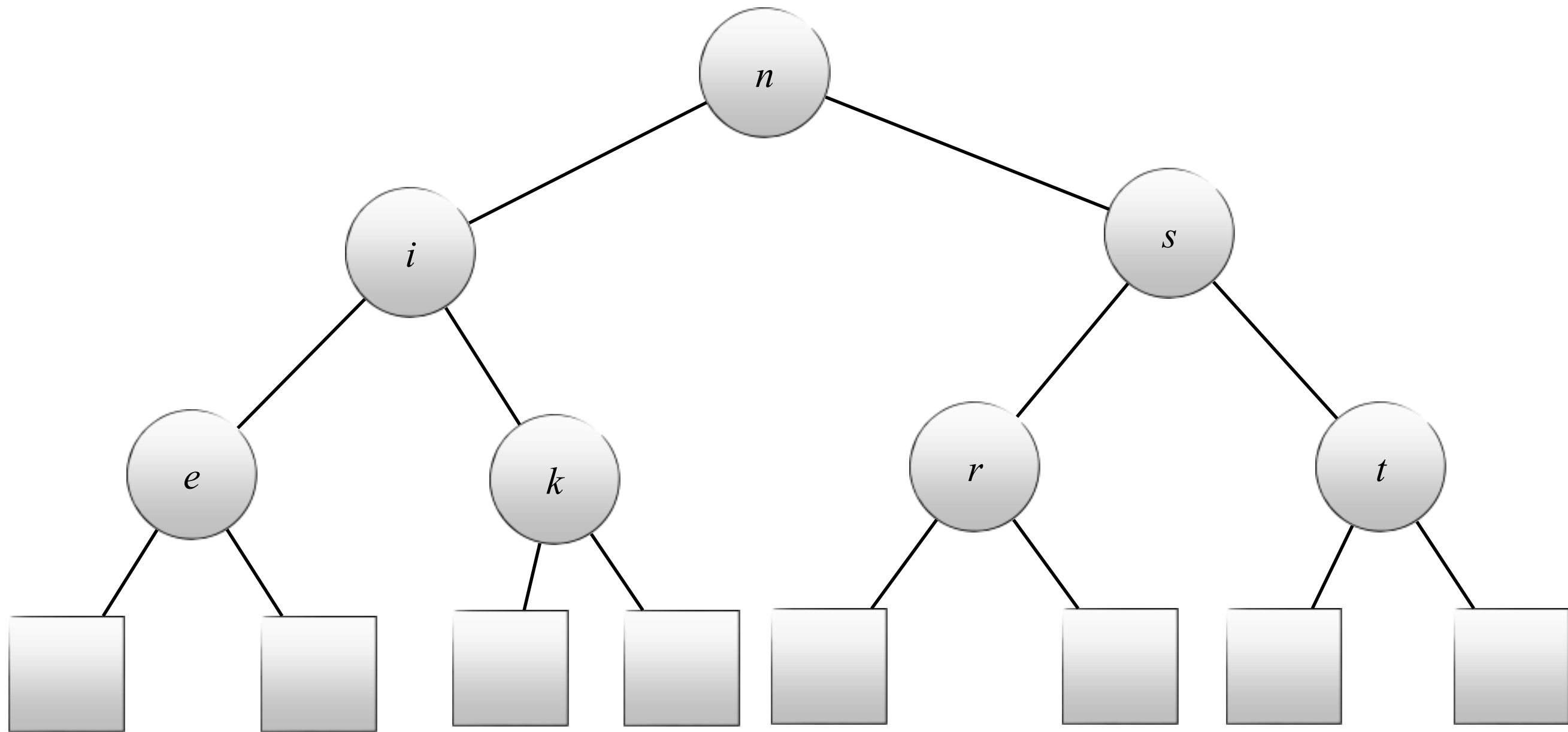


Insert key k

only one red link
per node: flip
colors



After inserting key k



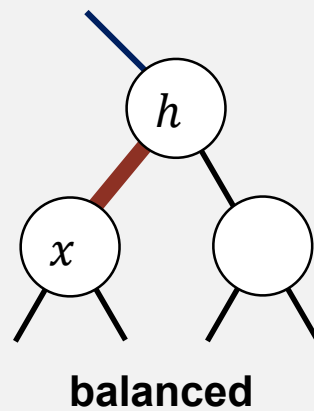
Insertion in a LLRB tree: overview

Basic strategy. Maintain 1-1 correspondence with 2-3 trees.

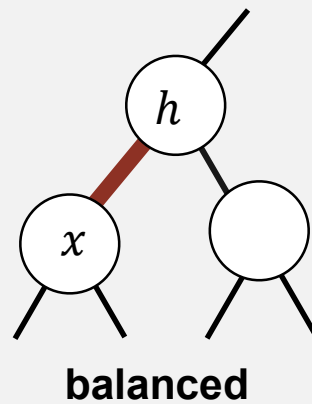
During internal operations, maintain: Symmetric order, perfect black balance.

Cases.

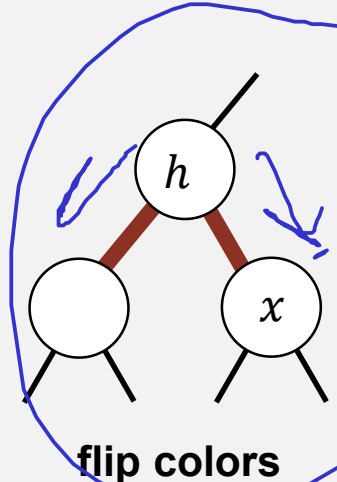
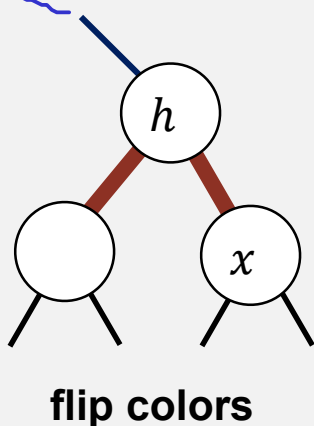
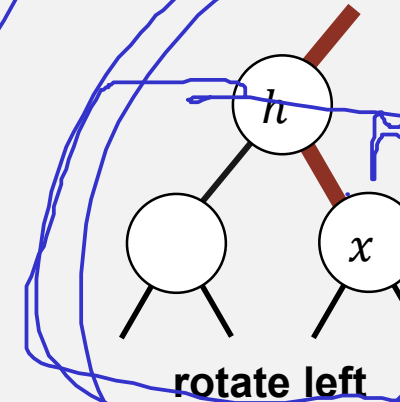
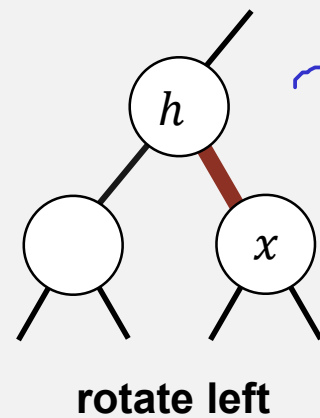
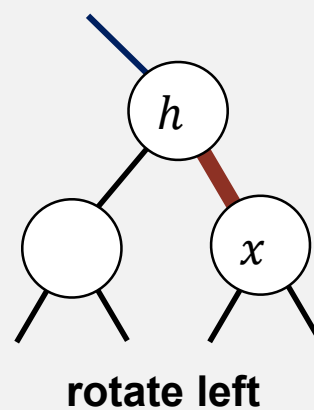
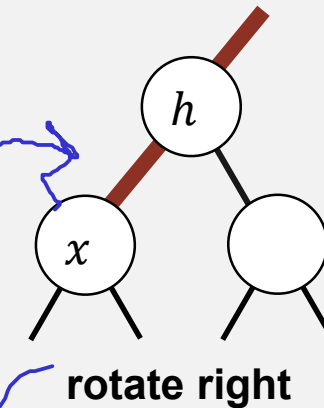
h right child



h left child - black

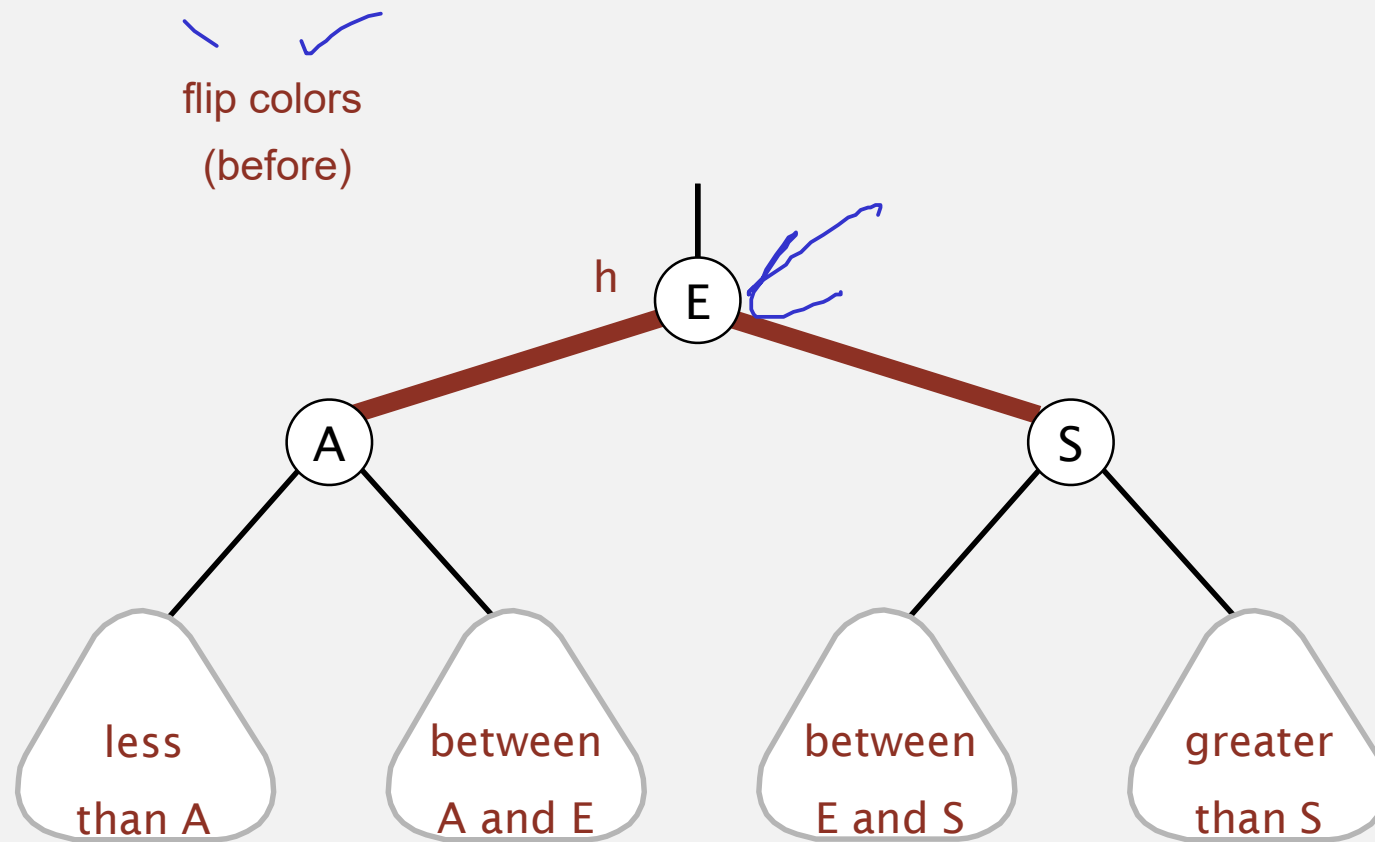


h left child - red



Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.

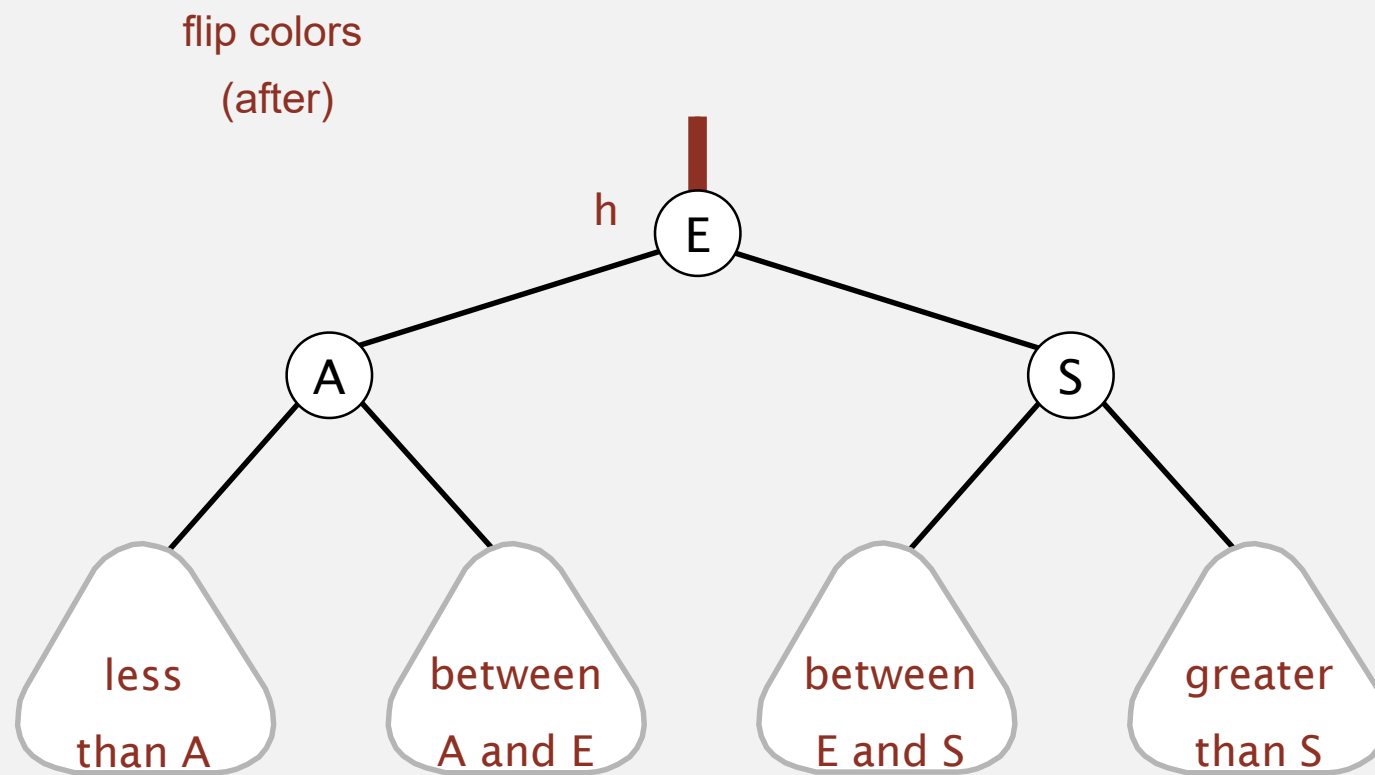


```
private void flipColors(Node h)
{
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Color flip. Recolor to split a (temporary) 4-node.



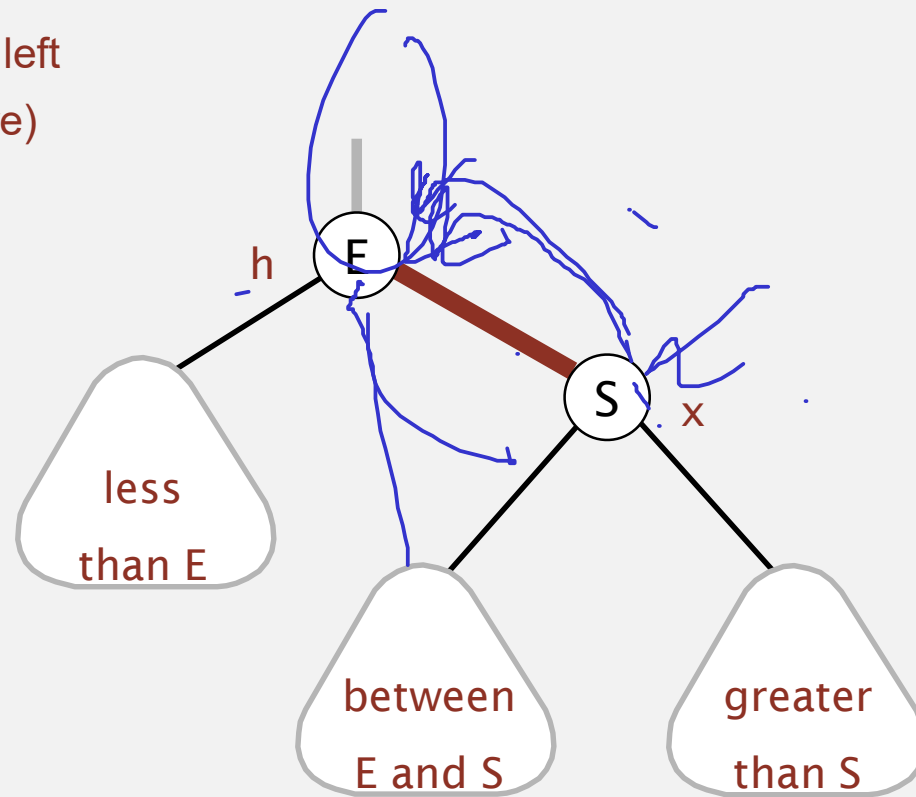
```
private void flipColors(Node h)
{
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

rotate E left
(before)

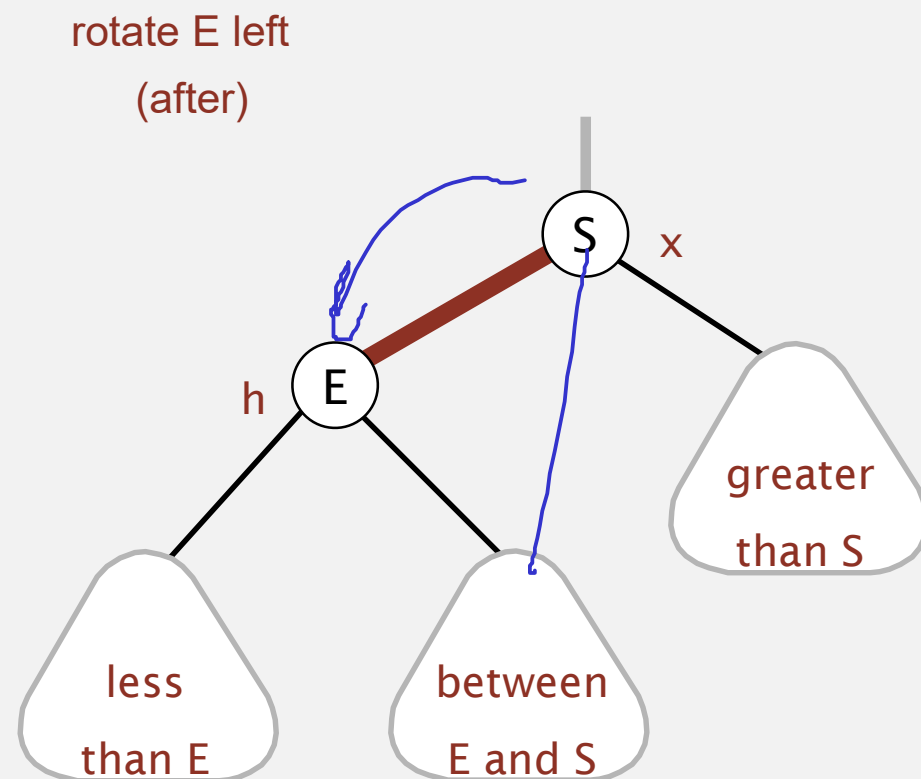


```
private Node rotateLeft(Node h)
{
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Left rotation. Orient a (temporarily) right-leaning red link to lean left.

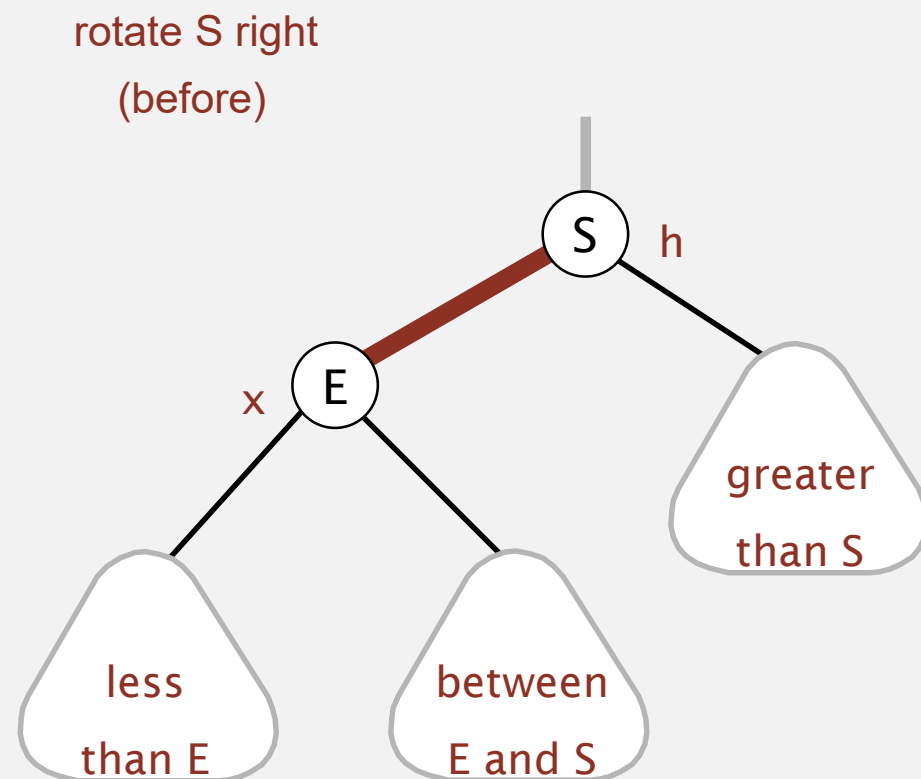


```
private Node rotateLeft(Node h)
{
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.



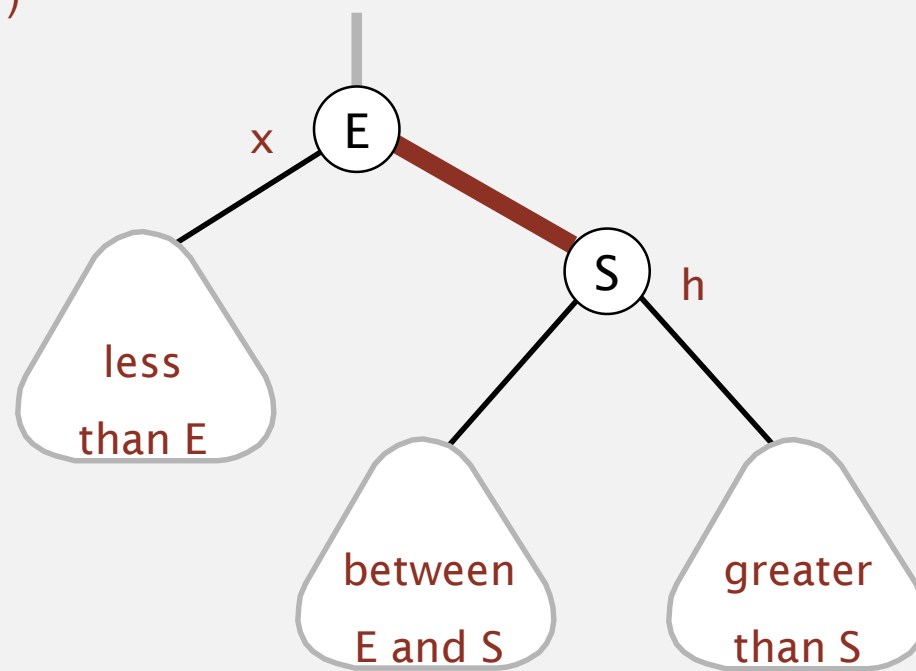
```
private Node rotateRight(Node h)
{
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Elementary red-black BST operations

Right rotation. Orient a left-leaning red link to (temporarily) lean right.

rotate S right
(after)



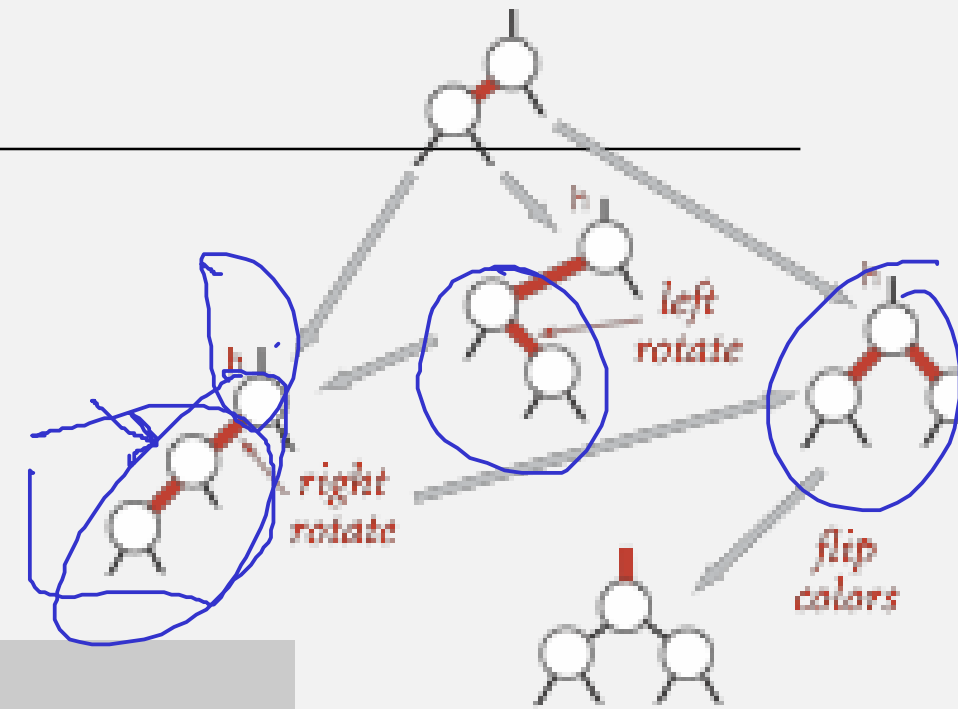
```
private Node rotateRight(Node h)
{
    Node x = h.left;
    h.left = x.right;
    x.right = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Invariants. Maintains symmetric order and perfect black balance.

Insertion in a LLRB tree: Java implementation

Same code for all cases.

- Right child red, left child black: **rotate left**.
- Left child, left-left grandchild red: **rotate right**.
- Both children red: **flip colors**.



```
private void put(Key key, Value val) {  
    root = put(root, key, val);  
    root.color = BLACK; }
```

```
private Node put(Node h, Key key, Value val) {  
    if (h == null) return new Node(key, val, RED);  
    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;
```

← insert at bottom
(and color it red)

```
    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);
```

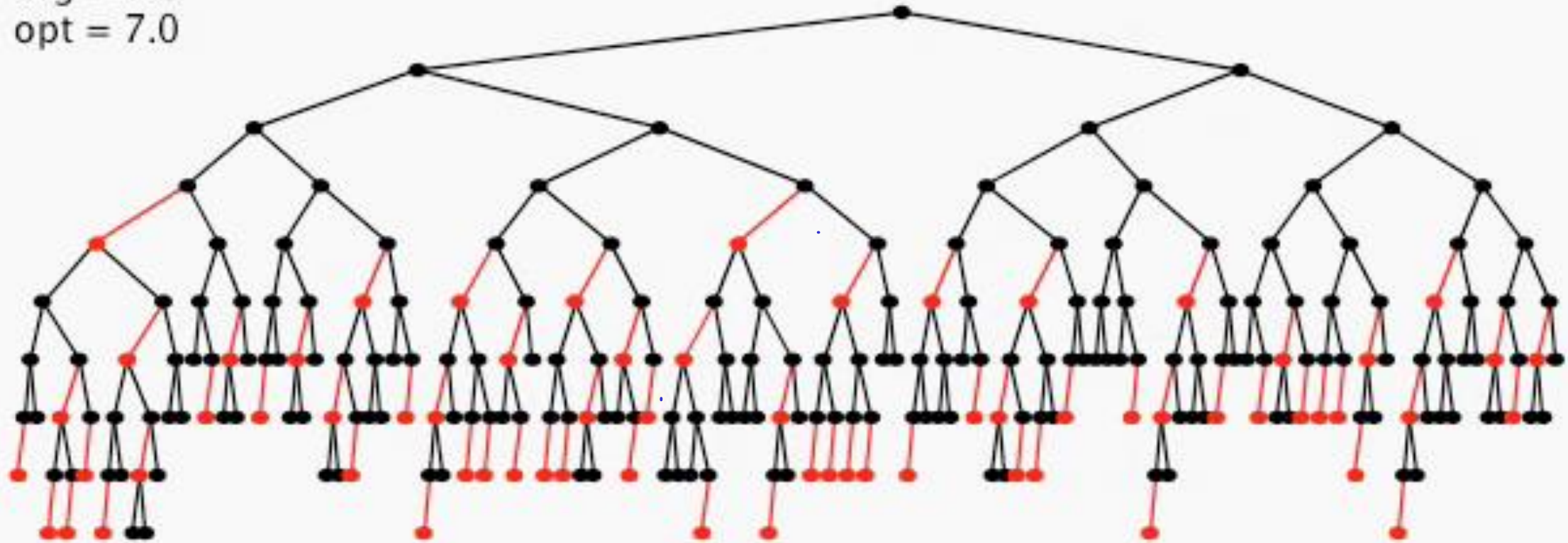
← lean left
← balance 4-node
← split 4-node

```
    return h; }
```

↑ only a few extra lines of code provides near-perfect balance

Insertion in a LLRB tree: visualization

N = 255
max = 10
avg = 7.3
opt = 7.0



255 random insertions