Lecture Notes on Red/Black Trees

15-122: Principles of Imperative Computation
Frank Pfenning

Lecture 17
October 21, 2010

1 Introduction

In this lecture we discuss an ingenious way to maintain the balance invariant for binary search trees. The resulting data structure of *red/black trees* is used in a number of standard library implementations in C, C++, and Java.

2 Three Invariants

A red/black tree is a binary search tree in which each node is colored either red or black. At the interface, we maintain three invariants:

Ordering Invariant This is the same as for binary search trees: all the keys to left of a node are smaller, and all the keys to the right of a node are larger than the key at the node itself.

Height Invariant The number of *black* nodes on every path from the root to each leaf is the same. We call this the *black height* of the tree.

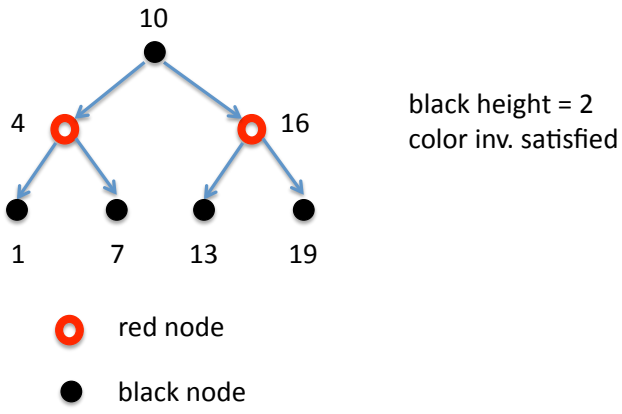
Color Invariant No two consecutive nodes are red.

The balance and color invariants together imply that the longest path from the root to a leaf is at most twice as long as the shortest path. Since insert and search in a binary search tree have time proportional to the length of the path from the root to the leaf, this guarantees $O(\log(n))$ times for these operations, even if the tree is not perfectly balanced. We therefore refer to the height and color invariants collectively as the *balance invariant*.

3 Insertion

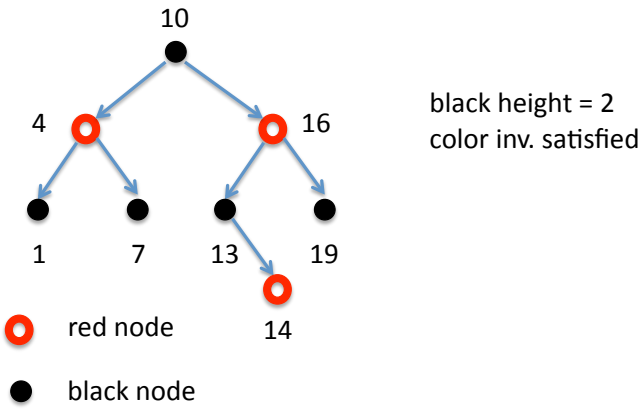
The trick is, of course, to maintain all three invariants while sticking to the logarithmic time bound for each insert and search operation. Search is easy, since the search algorithm does not require the node colors: it works exactly as for the general case of balanced binary trees.

Insertion, however, is not obvious. Let's try just following the usual algorithm for insertion. We compare the key of the new element with the key at the root. If it is equal, we replace the current element. If it is less we recursively insert into the left subtree. If it is greater, we recursively insert into the right subtree. Below is a concrete example.



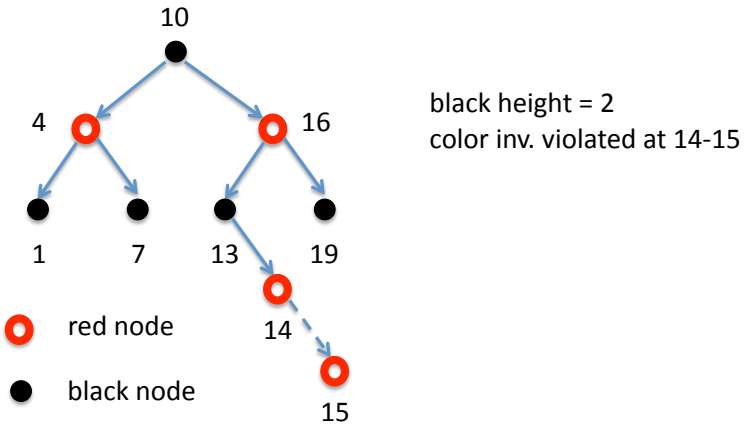
If we want to insert a new element with a key of 14, the insertion algorithm sketched above would put it to the right of 13. In order to preverse the

height invariant, we must color the new node red.



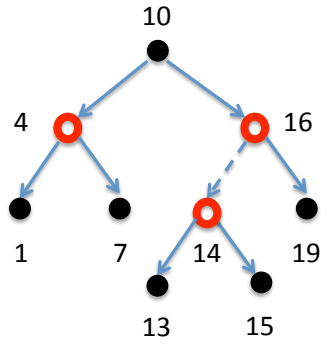
Both color and height invariants are still satisfied, as is the ordering invariant (which we always preserve).

Now consider another insertion, this time of an element with key 15. This is inserted to the right of the node with key 14. Again, to preserve the height invariant we must color it red, but this violates the color invariant since we have to adjacent red nodes.



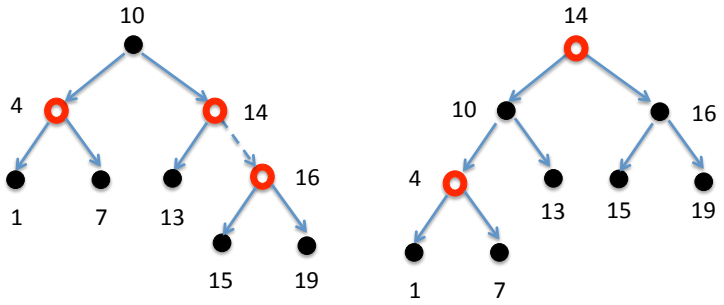
In order to restore the color invariant between 14 and 15 we can apply a *left rotation*, moving 14 up in the tree and 13 to the left. At the same time we

recolor 15 to be black, so as to remove the color conflict while preserving the black height invariant. However, we introduce a new color conflict between 14 and its parent.



black height = 2
color inv. restored at 14-15
color inv. violated at 16-14

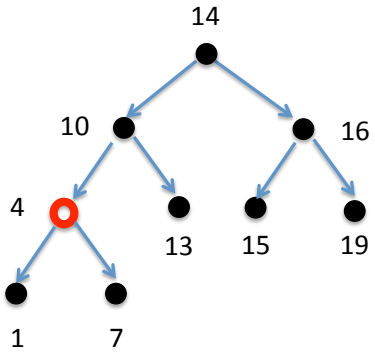
Now we can apply a right rotation, moving 14 up one level, immediately followed by a left rotation, moving 14 up to the root.



Such a sequence of two rotations is called a *double rotation*. The result now satisfies the height invariant (still with height 2) and the color invariant.

We can apply one further simple step, which is to recolor the root to black. This increases the black height on every path from the root by one, so it preserves the height invariant. Generally speaking, the fewer red nodes

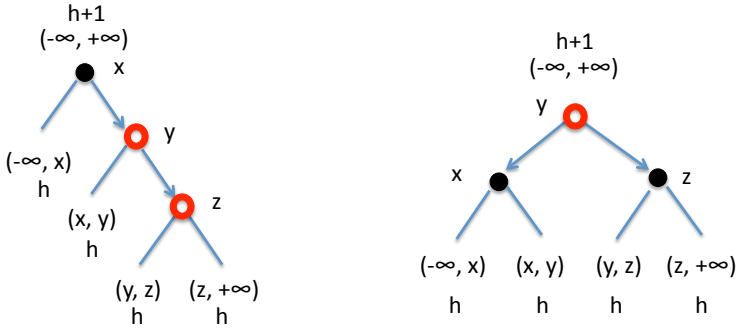
in the tree the fewer rotations will be forced, which is why we perform this recoloring. The final result is



black height = 3
color inv. satisfied

4 Rotations

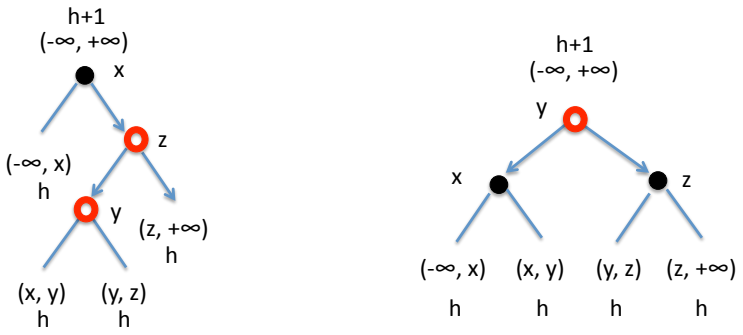
The general shape of a *left rotation* is shown in the following diagram. With each potential subtree and its interval we also show the black height of the the tree. The whole tree has black height $h + 1$ which means each elided subtree must have height h.



We see that all invariants are preserved, and the color invariant is restored. Of course, if this is a subtree below a red node, the tree on the left would satisfy the color invariant at the connection to its parent, while the tree on the right would not. We also saw this in the example.

The right rotation is entirely symmetric, so we don't show it here.

The double rotation is best thought of as a single operation, so we can examine its shape directly to verify that it preserves the invariants.



Observe that all invariants are preserved and, in fact, the tree on the right does observe the color invariant. As with a single rotation, however, we may still violate the color invariant at the interface of the node y at the top and its parent.

5 Abstract Data Types Revisited

We revisit some of the ideas underlying abstract types, as they are exhibited in this implementation. One of them is the fact that sometimes the client has to provide some operations to the implementation of an abstract type. Exactly which operations have to be supplied varies from case to case. Here, the client needs to provide the type of `elem` of elements (which must be a pointer type so it can be null), the function `elem_key` to extract keys from elements, and `compare` which compares two keys and returns their order (< 0 for *less*, $= 0$ for *equal* and > 0 for *greater*). First, the type definitions and the functions themselves.

```
/* elements */
struct elem {
  string word; /* key */
  int count;   /* information */
};

/* key comparison */
int compare(string s1, string s2) {
```

```
  return string_compare(s1,s2);
}

/* extracting keys from elements */
string elem_key(struct elem* e)
//@requires e != NULL;
{
  return e->word;
}
```

Next the interface, using the functions above.

```
/* interface definitions, implementation by client */
typedef string key;
typedef struct elem* elem;  /* NULL must be an elem */
key elem_key(elem e);
int compare(key k1, key k2);
```

Next comes the interface to the data structure itself.

```
typedef struct bst* bst;
bst bst_new();
void bst_insert(bst B, elem x);
elem bst_search(bst B, key k);  /* return NULL if not in tree */
```

This is uniform across all binary search tree implementations, so that we can experiment with different implementations without having to change the client code.

6 Defining Invariants

First, the implementation of the trees themselves. We add a boolean field `red` to the trees from the last two lectures. If it is `true`, the node is red, otherwise it should be considered black. As usual, we also have a header.

```
typedef struct tree* tree;
struct tree {
  elem data;                    /* data element */
  bool red;                     /* is node red? */
  tree left;                    /* left subtree */
  tree right;                   /* right subtree */
};
```

```
struct bst {
  tree root;                    /* root of the tree */
};

typedef bst rbt;
```

The last definition allows us to write `rbt` internally to refer to the particular implementation of binary search trees that are red/black trees.

The check that the tree is ordered does not change from the last two lectures, so we do not replicate the code here. To check that the color invariant is satisfied we just recurse over the tree, passing down the information if the parent is red. This is a common pattern of recursion.

```
/* sat_colorinv(T, red_parent) iff T satisfies the color invariant */
bool sat_colorinv(tree T, bool red_parent)
{ if (T == NULL) return true;
  if (red_parent && T->red) return false;
  return sat_colorinv(T->left, T->red)
      && sat_colorinv(T->right, T->red);
}
```

For the height invariant we calculate the black height of the two subtrees and compare. If the height invariant is violated, we return -1, otherwise the valid height invariant (which should be zero or positive).

```
/* black_height(T) >= 0 is the black height of T if it exists,
 * -1 otherwise
 */
int black_height(tree T) {
  if (T == NULL) return 0;
  {
    int hleft = black_height(T->left);
    int hright = black_height(T->right);
    if (hleft < 0 || hright < 0 || hleft != hright) return -1;
    //@assert hleft == hright;
    if (T->red) return hleft;
    else return hleft+1;
  }
}
```

A balanced tree satisfies the height invariant and the color invariant.

```
bool is_baltree(tree T) {
  int h = black_height(T);
  return h >= 0 && sat_colorinv(T, false);
}
```

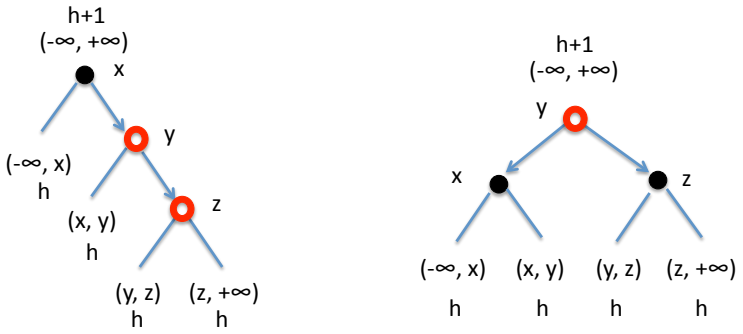
For insertion, we need to be able to check if we have a valid red/black tree with all invariants *except* that the color invariant might be violated between the root and its left child or the root and its right child. Care should be taken to make sure that we don't dereference a pointer to a tree that is potentially null.

```
bool is_rbtree_except_root(tree T) {
  bool ok = is_ordtree(T);
  ok = ok && black_height(T) >= 0;
  if (T->red
      && T->left != NULL && T->left->red
      && (T->right == NULL || !T->right->red))
    ok = ok && sat_colorinv(T->left, false)
            && sat_colorinv(T->right, false);
  else if (T->red
           && T->right != NULL && T->right->red
           && (T->left == NULL || !T->left->red))
    ok = ok && sat_colorinv(T->left, false)
            && sat_colorinv(T->right, false);
  else
    ok = ok && sat_colorinv(T, false);
  return ok;
}
```

In the first four calls to `sat_colorinv` we pass `false` as the second argument, pretending that the parent is *not* red, in order to allow the exception.

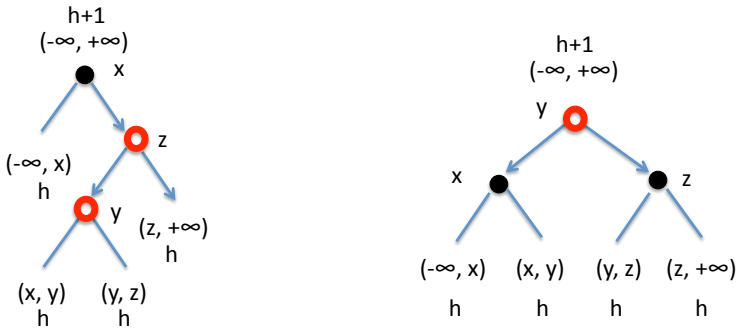
7 Implementing Insertion

Recall the picture for, say, the left rotation.



The right subtree, here labeled with key y is the result of a recursive insertion, which is allowed to violate the color invariant at its root. However, the tree we generate by adding the black node x now would violate the color invariant below x: we need to apply the rotation.

Similarly, if we have the situation on the left, with the black root



then we can apply the double rotation and restore the invariant.

So far, we get the following specification for a function that is called on the tree *after* insertion into the right subtree.

```
tree balance_right(tree T)
//@requires T != NULL;
//@requires !T->red ? is_rbtree_except_root(T->right) : ??;
//@ensures \old(!T->red) ? is_rbtree(\result) : ??;
  ;
```

In words: If the root of the tree we have to rebalance is black, then the right subtree must be a valid red/black tree except that the color invariant might be violated at its root. In turn, we guarantee that the result will be a valid red/black tree, as the diagrams above show.

But what happens if the root is red? Turns out in this case we can arrange that the right subtree is valid, without exception. This in turns means that we can immediately return, but we may violate the invariant at the root (which is permitted).

```
tree balance_right(tree T)
//@requires T != NULL;
/*@requires !T->red ? is_rbtree_except_root(T->right)
                    : is_rbtree(T->left); @*/
/*@ensures \old(!T->red) ? is_rbtree(\result)
                         : is_rbtree_except_root(\result); @*/
  ;
```

How can this work? It works because we can assert the following refined postcondition for `tree_insert`.

```
tree tree_insert(tree T, elem e)
//@requires is_rbtree(T);
/*@ensures \old(T != NULL && T->red)
           ? is_rbtree_except_root(\result)
           : is_rbtree(\result); @*/
  ;
```

If we insert into a tree with a red root, then the result may violate the invariant at the root. However, if we insert into a tree with a black or null root, then the result will be valid without exception.

Here is the complete code for insert.

```
tree tree_insert(tree T, elem e)
//@requires is_rbtree(T);
/*@ensures \old(T != NULL && T->red)
           ? is_rbtree_except_root(\result)
           : is_rbtree(\result); @*/
//@ensures tree_search(\result, elem_key(e)) == e;
{
  if (T == NULL) {  /* create new node */
    T = alloc(struct tree);
    T->data = e; T->red = true; /* new nodes are red */
```

```
    T->left = NULL; T->right = NULL;
  } else {
    key kt = elem_key(T->data);
    key k = elem_key(e);
    if (compare(k, kt) == 0) {
      T->data = e;
    } else if (compare(k, kt) < 0) {
      //@assert T->red ? (T->left == NULL || !T->left->red) : true;
      T->left = tree_insert(T->left, e);
      /*@assert T->red ? is_rbtree(T->left)
                       : is_rbtree_except_root(T->left); @*/
      T = balance_left(T);
    } else {
      //@assert compare(k, kt) > 0;
      //@assert T->red ? (T->right == NULL || !T->right->red) : true;
      T->right = tree_insert(T->right, e);
      /*@assert T->red ? is_rbtree(T->right)
                       : is_rbtree_except_root(T->right); @*/
      T = balance_right(T);
    }
  }
  return T;
}
```

Let's reason through the crucial section, when we insert into the right subtree. Inserting into the left subtree is symmetric.

```
//@assert T->red ? (T->right == NULL || !T->right->red) : true;
T->right = tree_insert(T->right, e);    /* (2) */
/*@assert T->red ? is_rbtree(T->right)
                 : is_rbtree_except_root(T->right); @*/
T = balance_right(T);                   /* (5) */
...
return T;
```

We have the following cases.

T **is red:** T is a valid red/black tree, so when its root is red, its right subtree must be null or black. Therefore, the postcondition for the recursive call in line (2) tells us that `is_rbtree(T->right)` after the assignment in line (2). This establishes the precondition for `balance_right(T)`.

From the postcondition of the call to `balance_right(T)` in line (5) we conclude that `is_rbtree_except_root(T)` after the assignment in line (5). If the original `T` is red, this is what we needed to establish.

T **is black:** In this case the postcondition of the recursive insertion only guarantees that `is_rbtree_except_root(T->right)` after the assignment in line (2). This is establishes the precondition for `balance_right(T)` in line (5). The postcondition established `is_rbtree(T)` after the assignment in line (5). If the original `T` was black, this is what we neede to establish.

T **is null:** In this case we need to establish `is_rbtree(T)`, where T is the value returned by the insertion. But this is evident from its construction: T consists of a single red node with black height 0.

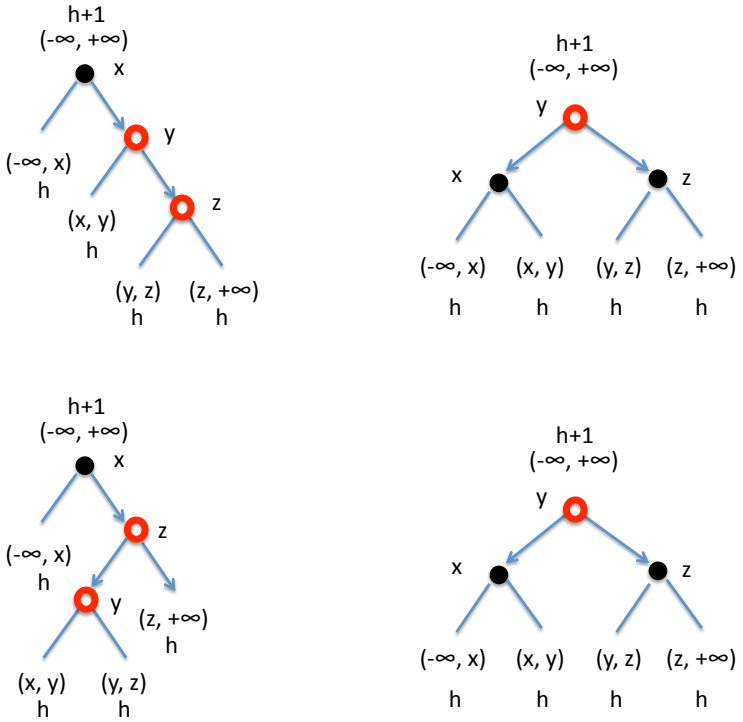
Finally, we have a wrapper function which might need to restore the invariant at the root, because the recursive insertion may leave with with the color invariant violated at the root. Fortunately, recoloring the root black restores the color invariant and maintains the black height invariant by uniformly increasing the number of black nodes on all paths to the leaves by one.

```
void bst_insert(rbt RBT, elem e)
//@requires is_rbt(RBT);
//@ensures is_rbt(RBT);
{
  // wrapper function to start the process at root
  RBT->root = tree_insert(RBT->root, e);
  //@assert is_rbtree_except_root(RBT->root);
  RBT->root->red = false; /* color root black; might already be black */
  return;
}
```

8 Implementing Rebalancing

Now that we have a precise specification, the rebalancing code practically implements itself. Here is the code for rebalancing after an insertion into

the right subtree; the other one is completely symmetric.



The picture should help understand the code.

```
tree balance_right(tree T)
//@requires T != NULL;
/*@requires T->red ? is_rbtree(T->right)
                   : is_rbtree_except_root(T->right); @*/
/*@ensures \old(T->red) ? is_rbtree_except_root(\result)
                        : is_rbtree(\result); @*/
{ if (T->red) return T;
  //@assert !T->red;
  if (T->right->red
     && T->right->right != NULL && T->right->right->red)
    // rotate left
    { tree root = T->right; T->right = root->left; root->left = T;
      root->right->red = false;
      //@assert root->red == true;
      //@assert root->left->red == false;
```

```
      return root;
    } else if (T->right->red
       && T->right->left != NULL && T->right->left->red)
    // double rotate left
    { tree root = T->right->left; T->right->left = root->right;
      root->right = T->right; T->right = root->left; root->left = T;
      root->right->red = false;
      //@assert root->red == true;
      //@assert root->left->red == false;
      return root;
    } else return T;
}
```