Red-Black Trees

1. Red-Black Trees
   - balancing binary search trees
   - relation with 2-3-4 trees
2. Insertion into a Red-Black Tree
   - algorithm for insertion
   - an elaborate example of an insert
   - inserting a sequence of numbers
3. Recursive Insert Function
   - pseudo code

MCS 360 Lecture 36
Introduction to Data Structures
Jan Verschelde, 20 April 2020

Red-Black Trees

1. Red-Black Trees
   - balancing binary search trees
   - relation with 2-3-4 trees
2. Insertion into a Red-Black Tree
   - algorithm for insertion
   - an elaborate example of an insert
   - inserting a sequence of numbers
3. Recursive Insert Function
   - pseudo code

Binary Search Trees

Binary search trees store ordered data.

Rules to insert x at node N:

- if N is empty, then put x in N
- if $x < N$, insert x to the left of N
- if $x \geq N$, insert x to the right of N

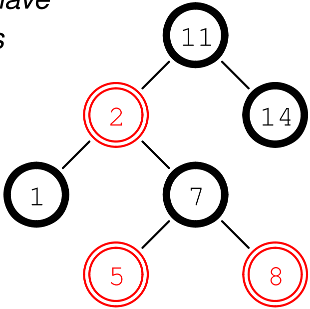
Balanced tree of n elements has depth is $\log_2(n)$
$\Rightarrow$ retrieval is $O(\log_2(n))$, almost constant.

With rotation we make a tree balanced.
Alternative to AVL tree: nodes with > 2 children.

Every node in a binary tree has at most 2 children.
A 2-3-4 tree has nodes with 2, 3, and 4 children.
A red-black tree is a binary tree equivalent to a 2-3-4 tree.

a red-black tree

red nodes have hollow rings



Four invariants:

1. A node is either red or black.
2. The root is always black.
3. A red node has always black children.
4. #black nodes in path from root to any leaf is the same.

verification of all four invariants

```
11 is black
  2 is red
    1 is black
    7 is black
      5 is red
      8 is red
  14 is black
The root is black.
Node 2 is red and has black left child.
Node 2 is red and has black right child.
All red nodes have black children.
The first path ends at 1 is black.
The number of black nodes is 2.
Reached end of path at 5 is red.
The number of black nodes is 2.
Reached end of path at 8 is red.
The number of black nodes is 2.
Reached end of path at 14 is black.
The number of black nodes is 2.
The tree respects all invariants.
```

a node in a red-black tree

```
template<typename Item_Type>
struct Node
{
   Item_Type data;  // numbers stored at node in tree
   bool isred;      // true if red, false if black
   Node *parent;    // pointer to the parent node
   Node *left;      // pointer to left branch of tree
   Node *right;     // pointer to right branch of tree

   Node(const Item_Type& item, bool redcolor = true,
        Node* parent_ptr = NULL,
        Node* left_ptr = NULL, Node* right_ptr = NULL) :
        data(item), isred(redcolor), parent(parent_ptr),
        left(left_ptr), right(right_ptr) {}
   // Makes a node with as data field the value of item,
   // the default color is red, and all pointers are set
   // to NULL by default.
```

a class to define a red-black tree

- The root of the tree is a pointer to a node, declared as private, as a hidden data attribute.
- The `bool isred` in `Node` respects the first invariant: all nodes are either red (`true`) or black (`false`).
- The second invariant: red nodes have black children is verified by a recursive tree traversal.
- The #black nodes in any path from root to leaf is the same is verified via backtracking.

performance

#black nodes in path from root to any leaf is the same
$\Rightarrow$ equivalent 2-3-4 tree is balanced.

Performance of a red-black tree:

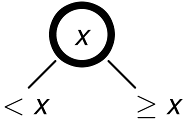
- upper limit of depth of red-black tree: $2\log_2(n) + 2$ for a search of tree with n elements.
- average cost of search is empirically: $1.002\log_2(n)$.

Red-Black Trees

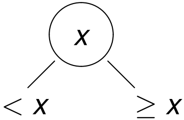
1. Red-Black Trees
   - balancing binary search trees
   - relation with 2-3-4 trees
2. Insertion into a Red-Black Tree
   - algorithm for insertion
   - an elaborate example of an insert
   - inserting a sequence of numbers
3. Recursive Insert Function
   - pseudo code

relation with 2-3-4 trees: nodes with 2 and 4 children

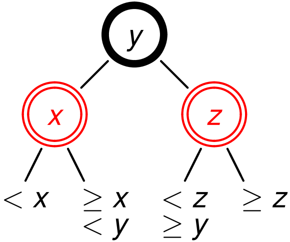
No red children:



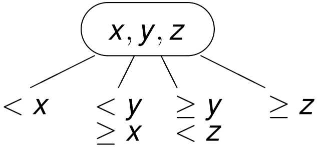
a 2-node



Two red children:

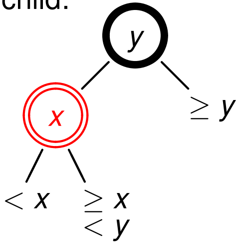


a 4-node

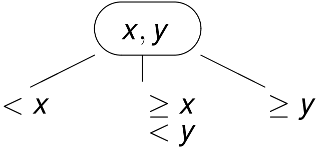


relation with 2-3-4 trees: nodes with 3 children

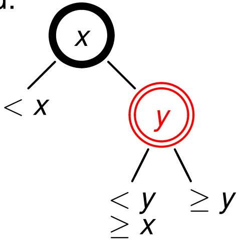
One red child:



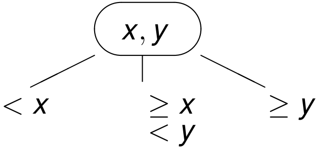
a 3-node



One red child:



a 3-node



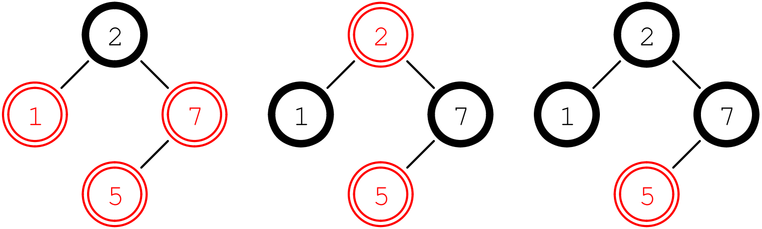
Red-Black Trees

1. Red-Black Trees
   - balancing binary search trees
   - relation with 2-3-4 trees
2. Insertion into a Red-Black Tree
   - algorithm for insertion
   - an elaborate example of an insert
   - inserting a sequence of numbers
3. Recursive Insert Function
   - pseudo code

Changing Colors

Case 0: The color of a new leaf is always red.
If parent of new leaf is black, then done.

Case 1: We just change colors, e.g.:

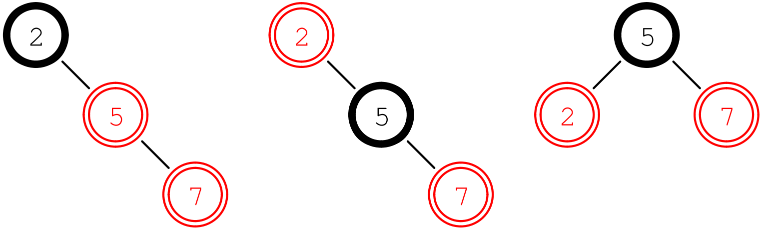


1. If sibling of parent also red, change color of parent and its sibling to black, and change color of the grandparent.
2. Ensure color of the root is black.

Tree Rotation

What if parent does not have a red sibling?

Case 2: One rotation suffices, e.g.:

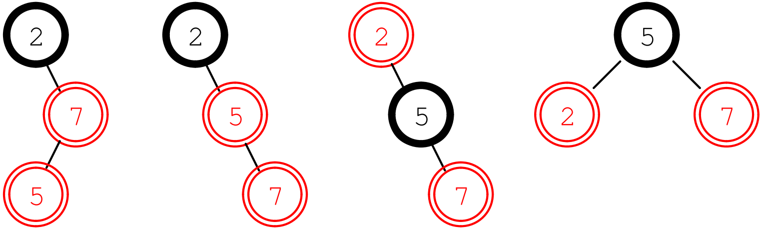


1. Change the color of the parent to black, change the color of the grandparent to red.
2. Rotate to restore the 4th invariant.

Double Rotation

One rotation works for right-right or left-left tree.

Case 3: Tree is right-left (or left-right), e.g.:



1. Rotate right-left tree to right-right tree.
2. Change color of parent and grandparent.
3. Rotate to restore the 4th invariant.

Red-Black Trees

1. Red-Black Trees
   - balancing binary search trees
   - relation with 2-3-4 trees
2. Insertion into a Red-Black Tree
   - algorithm for insertion
   - an elaborate example of an insert
   - inserting a sequence of numbers
3. Recursive Insert Function
   - pseudo code

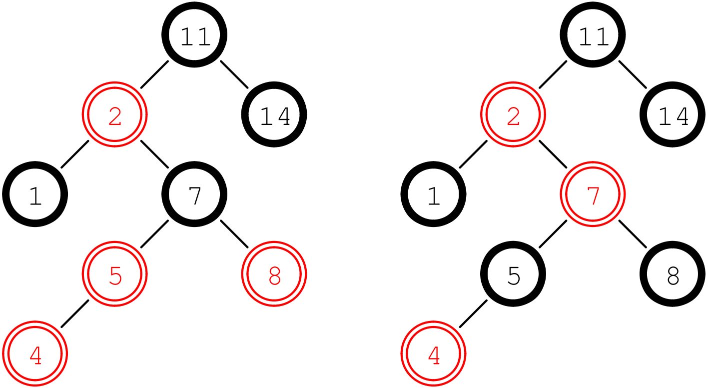
inserting into a red-black tree

During insertion, many cases may occur:

1. Locate the new leaf and color it red.
2. If in case 0, then done.
3. Apply cases 1, 2, and 3, restoring invariants.

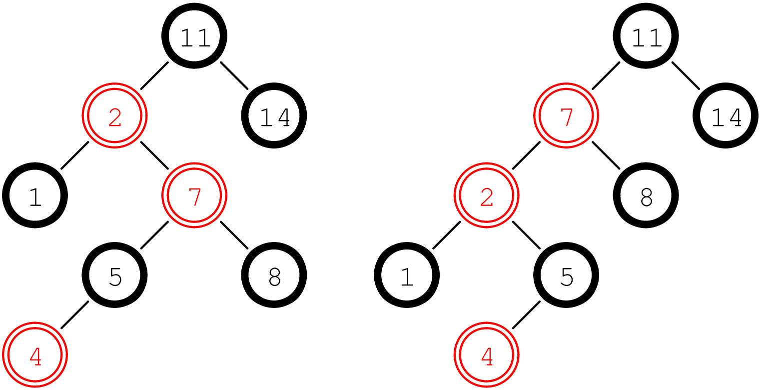
    1. case 1: parent has red sibling
       $\Rightarrow$ change colors
    2. case 2: left-left or right-right tree
       $\Rightarrow$ change colors and rotate
    3. case 3: right-left or left-right tree
       $\Rightarrow$ rotate, change colors, rotate

inserting 4



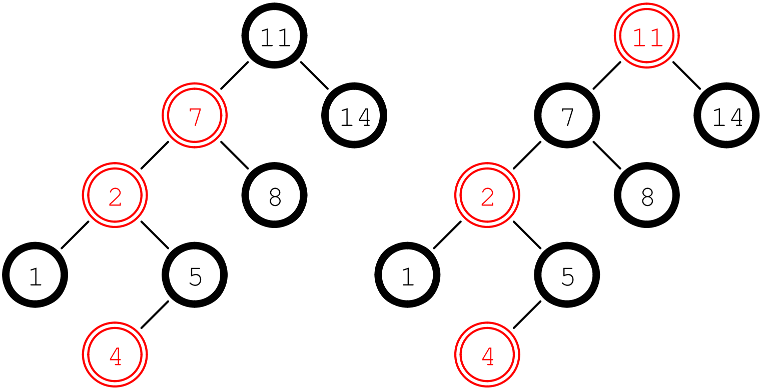
After changing colors of 5 and 8 to black,
and changing 7 to red, but then parent of 7 is red too...

rotating around 2



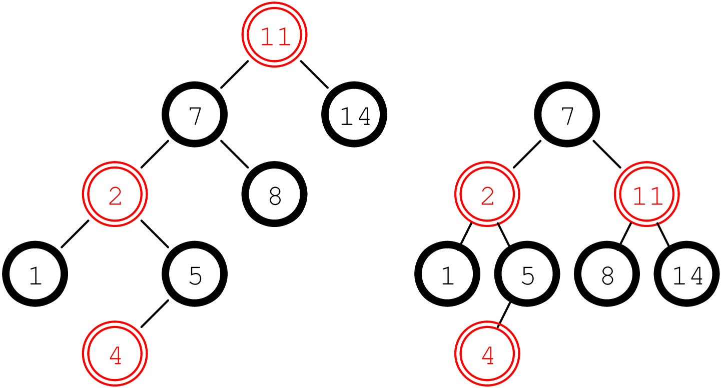
Case 3: rotate left-right tree (11-2-7) to left-left tree
is the first step...

changing colors



The second step in Case 3 is to change the color of 7 to black and the color of 11 to red.

rotate around 11



In the last step of Case 3, we restored the balance by a right rotation around 11.

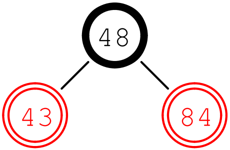
Red-Black Trees

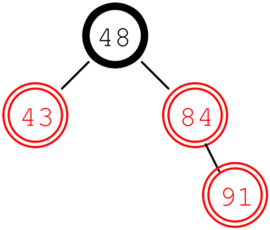
1. Red-Black Trees
   - balancing binary search trees
   - relation with 2-3-4 trees
2. Insertion into a Red-Black Tree
   - algorithm for insertion
   - an elaborate example of an insert
   - inserting a sequence of numbers
3. Recursive Insert Function
   - pseudo code

inserting a sequence of numbers

Insert 48, 84, 43, 91, 38, 79, 63, 59, 49, 98 into a red-black tree.

After inserting 48, 84, 43:

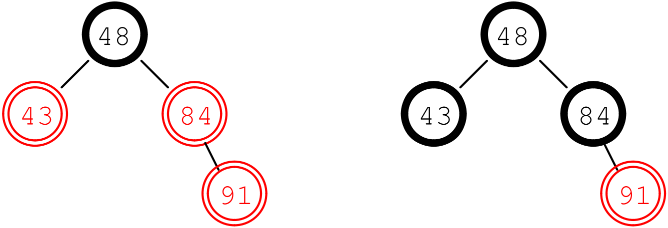




After inserting 91 to the binary search tree,
we verify the four invariants of a red-black tree.

a red node has always black children

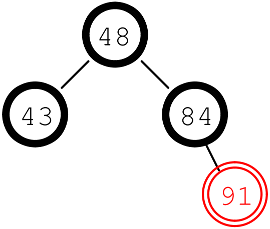
After inserting 91, we have to change colors:

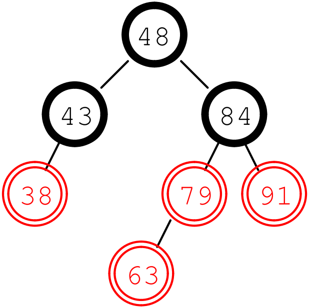


The numbers left to insert are 38, 79, 63, 59, 49, 98.

inserting more numbers

Inserting 38, 79, 63:

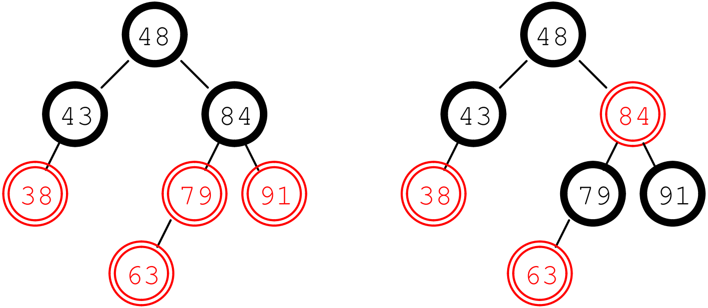




Are all four invariants of a red-black tree satisfied?

changing colors

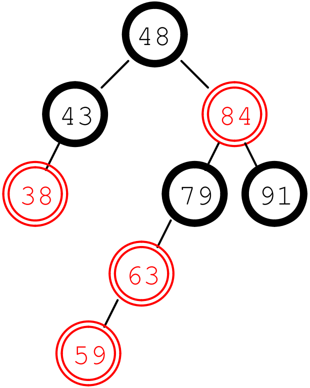
A red node must always have black children.

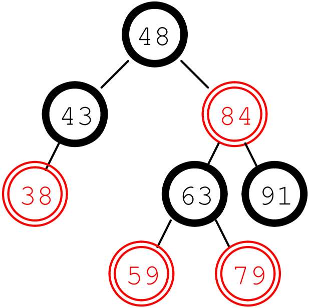


The numbers left to insert are 59, 49, 98.

rotate a left-left tree

After interting 59, we have a left-left tree.
Coloring 59 black will violate the fourth invariant, so we rotate.

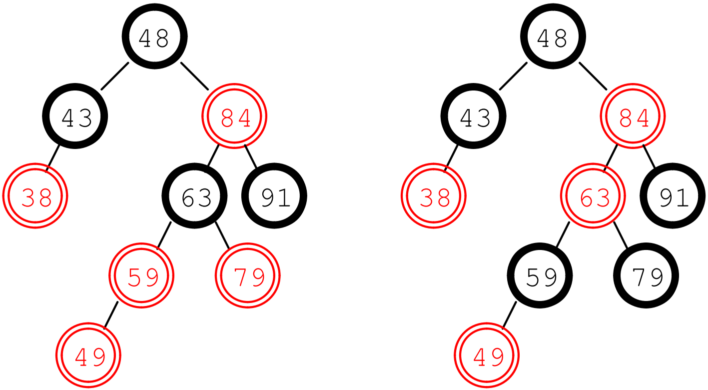




The numbers left to insert are 49 and 98.

inserting 49

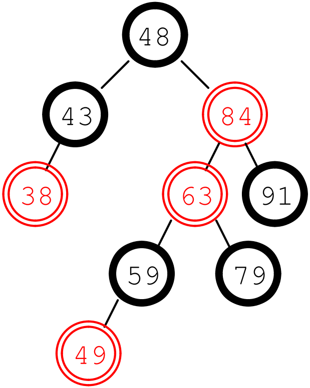
After inserting 49, we first change colors:

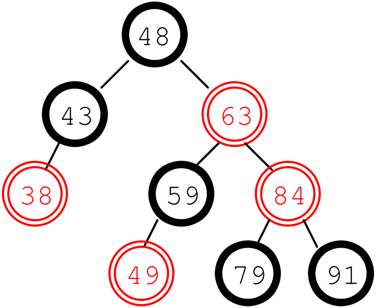


Not all invariants of the red-black tree are satisfied.

rotating once ...

We see that the right tree is left heavy.

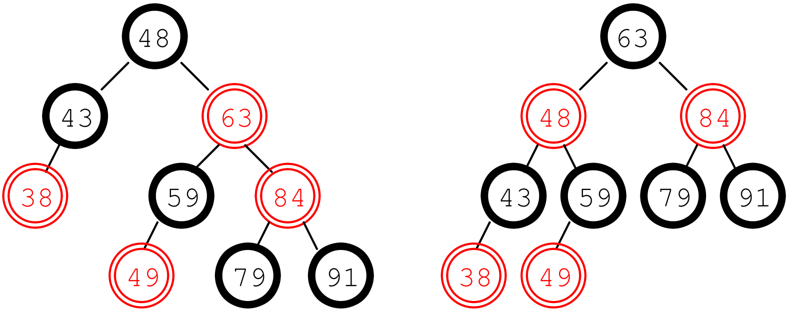




After rotation, the tree is right heavy.

rotating again

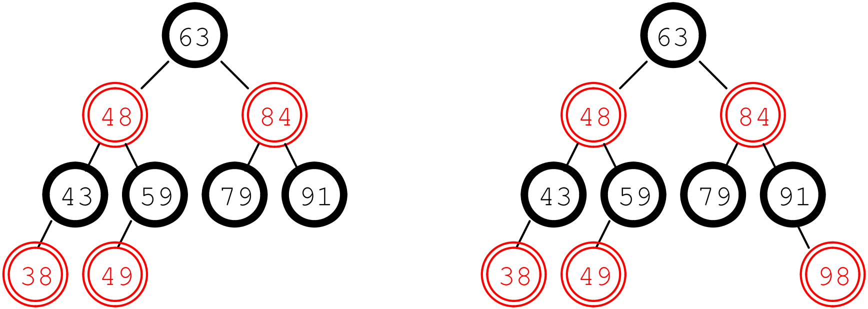
The new root will become the root of the right tree.



The left of 63 turned to the right of 48.

One number left to insert: 98.

inserting 98



Red-Black Trees

1. Red-Black Trees
   - balancing binary search trees
   - relation with 2-3-4 trees
2. Insertion into a Red-Black Tree
   - algorithm for insertion
   - an elaborate example of an insert
   - inserting a sequence of numbers
3. Recursive Insert Function
   - pseudo code

Recursive Insert Function

We extend the node type of a binary search tree:

1. a node contains data (often an integer key);
2. in addition, every node has a color: red or black;
3. two pointers to nodes lead to left and right children.

Prototype of a recursive insert function:

```
bool insert ( Tree &root, Item_Type item );

// returns false if item belonged to root

// returns true if item was added to the root
```

pseudo code

```
bool insert ( Tree &root, Item_Type item )
   if(root == NULL)
      root = new black node;
      return true;
   else if(item == root->data)
      return false;
   else if (item < root->data)
      if(left == NULL)
         left = new red node;
         return true;
      else if((left == red) && (right == red))
         change left and right to black
         and color the local root red;
```

pseudo code continued

```
    if(insert(left,item))
       if(left grandchild == red)
          change left to black
          and color the local root red;
          rotate the local root right;
       else if(right grandchild == red)
          rotate the left child left;
          change left to black
          and color the local root to red;
          rotate the local root right;

 else // item > root->data: exercise

 if(local root is root of the tree)
    color the root black.
```

Summary + Exercises

More on chapter 11 on balancing binary search trees.

Exercises:

1. Take the largest example of a red-black tree on these slides and draw the equivalent 2-3-4 tree.
2. Write the pseudo code for the case when the item is inserted to the right child.
3. Instead of inserting 4 in the elaborate example, insert 9 and illustrate all stages in the insertion.