PROBLEM SOLVING 4

Trees

CS340

Sort with binary search tree?

- What is the difference between the **binary-search-tree property** and the **min-heap property**?
- Can the min-heap property be used to print out the keys of an n-node tree in sorted order in O(n) time?

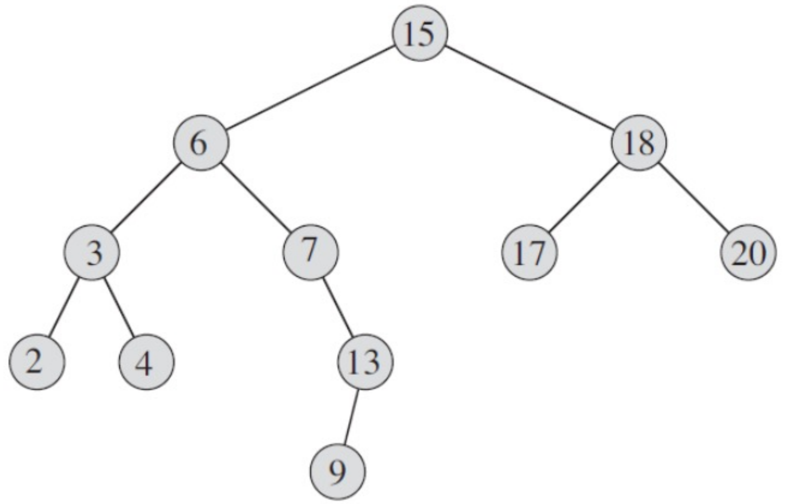
Binary Search Tree

- Suppose that we have numbers between 1 and 1000 in a binary search tree, and we want to search for the number 363. Which of the following sequences could *not* be the sequence of nodes examined?
- ***a.*** 2, 252, 401, 398, 330, 344, 397, 363.
- ***b.*** 924, 220, 911, 244, 898, 258, 362, 363.
- ***c.*** 925, 202, 911, 240, 912, 245, 363.
- ***d.*** 2, 399, 387, 219, 266, 382, 381, 278, 363.
- ***e.*** 935, 278, 347, 621, 299, 392, 358, 363.

continued →

Sort with binary search tree?

- An alternative method of performing an inorder tree walk of an n-node binary search tree finds the minimum element in the tree by calling TREE-MINIMUM and then makes n-1 calls to TREE-SUCCESSOR.
  - What is the omega for this walk?
  - How many times is each edge traversed?
  - What is the theta for this walk?



Sorting with binary search tree?

- We can sort a given set of n numbers by first building a binary search tree containing these numbers (using TREE-INSERT repeatedly to insert the numbers one by one) and then printing the numbers by an inorder tree walk. What are the worst case and best-case running times for this sorting algorithm?

BST Heights

- For the set of {1, 4, 5, 10, 16, 17, 21} of keys, draw binary search trees of heights 2, 3, 4, 5, and 6.

Red-Black Tree Properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf is NIL and black.
4. If a node is red then both its children are black.
5. For each node, all simple paths to its descendant leaves have the same number of black nodes.
    - This number defines the black height of a node x, bh(x).
    - By the previous property, $bh(x) \geq h(x)/2$

Red-Black Trees

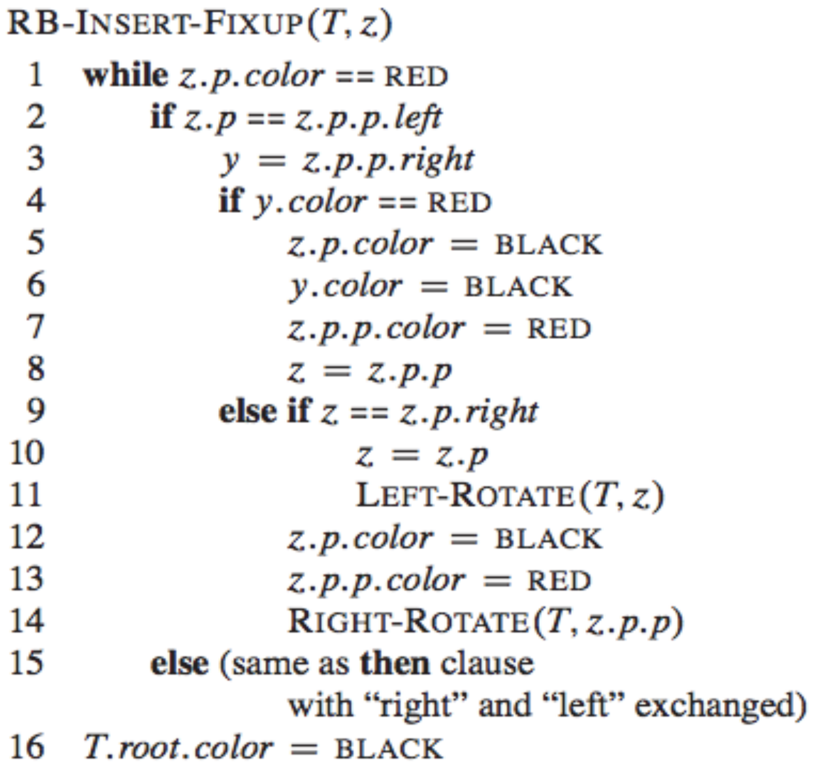
- What is the largest possible number of internal nodes in a red-black tree with black-height *k*? What is the smallest possible number?

Rotations

- Argue that in every n-node binary search tree, there are exactly n - 1 possible rotations.

RB-FIXUP

- It's a {while} loop
- What's our guarantee it works?



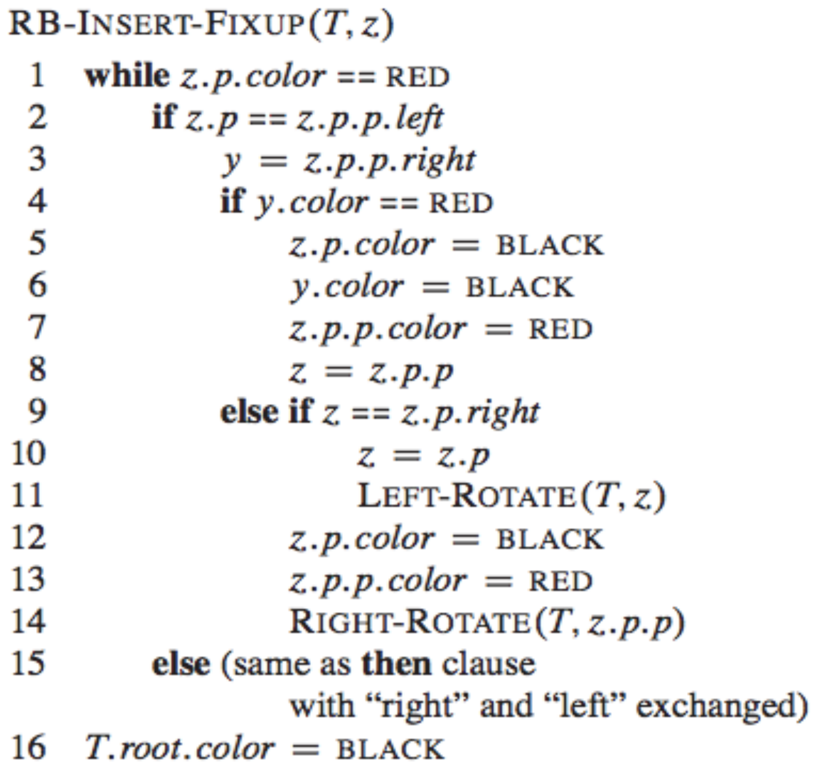
```
RB-INSERT-FIXUP($T, z$)
while $z.p.color$ == RED
    if $z.p == z.p.p.left$
        $y = z.p.p.right$
        if $y.color$ == RED
            $z.p.color$ = BLACK                    // case 1
            $y.color$ = BLACK                      // case 1
            $z.p.p.color$ = RED                    // case 1
            $z = z.p.p$                            // case 1
        else if $z == z.p.right$
                $z = z.p$                          // case 2
                LEFT-ROTATE($T, z$)                // case 2
            $z.p.color$ = BLACK                    // case 3
            $z.p.p.color$ = RED                    // case 3
            RIGHT-ROTATE($T, z.p.p$)               // case 3
    else (same as then clause
            with "right" and "left" exchanged)
$T.root.color$ = BLACK
```

```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.color = BLACK
6              y.color = BLACK
7              z.p.p.color = RED
8              z = z.p.p
9          else if z == z.p.right
10                 z = z.p
11                 LEFT-ROTATE(T, z)
12             z.p.color = BLACK
13             z.p.p.color = RED
14             RIGHT-ROTATE(T, z.p.p)
15     else (same as then clause
               with "right" and "left" exchanged)
16 T.root.color = BLACK
```

```
if (uncle.color == red)
{
   # Handle case
}
else
{
    if (z == z.p.right)
    {
         # Handle case 2
    }
    # Handle case 3
}
```

RB-FIXUP

- What is the loop invariant?
  a. Node z is red
  b. If z.p is the root, then z.p is black
  c. If the tree violates any of the red-black properties, they must be:
     - 2) z is the root and is red
     - 4) z and its parent are both red

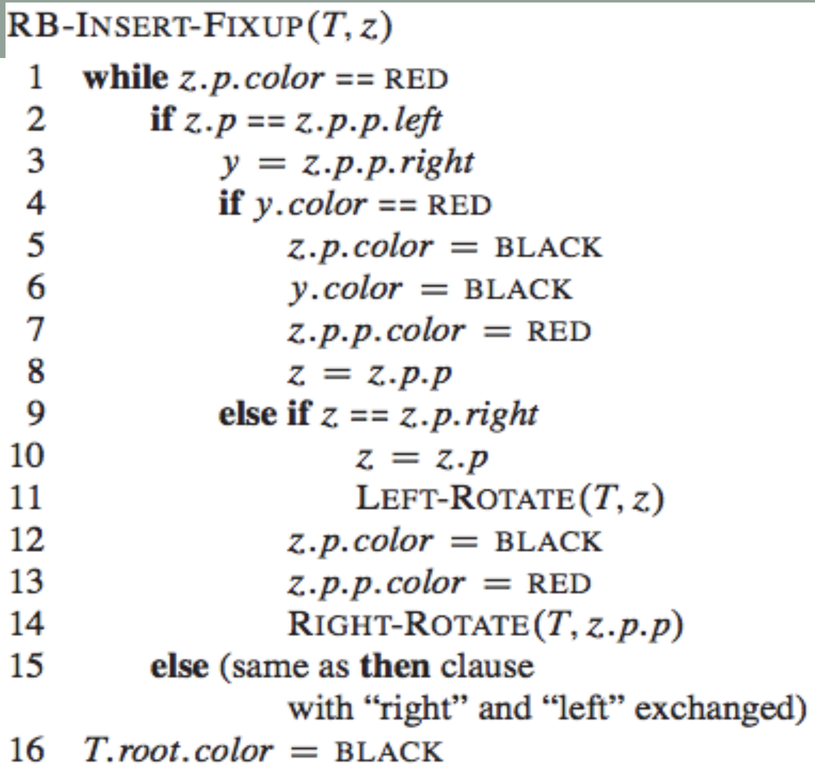


```
RB-INSERT-FIXUP(T, z)
while $z.p.color$ == RED
    if $z.p == z.p.p.left$
        $y = z.p.p.right$
        if $y.color$ == RED
            $z.p.color$ = BLACK
            $y.color$ = BLACK
            $z.p.p.color$ = RED
            $z = z.p.p$
        else if $z == z.p.right$
                $z = z.p$
                LEFT-ROTATE(T, z)
            $z.p.color$ = BLACK
            $z.p.p.color$ = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then clause
            with "right" and "left" exchanged)
$T.root.color$ = BLACK
```

RB-FIXUP

- What is the loop invariant?
  a. Node z is red
  b. If z.p is the root, then z.p is black
  c. If the tree violates any of the red-black properties, it only violates one, and it must be:
    - 2) z is the root and is red
    - 4) z and its parent are both red

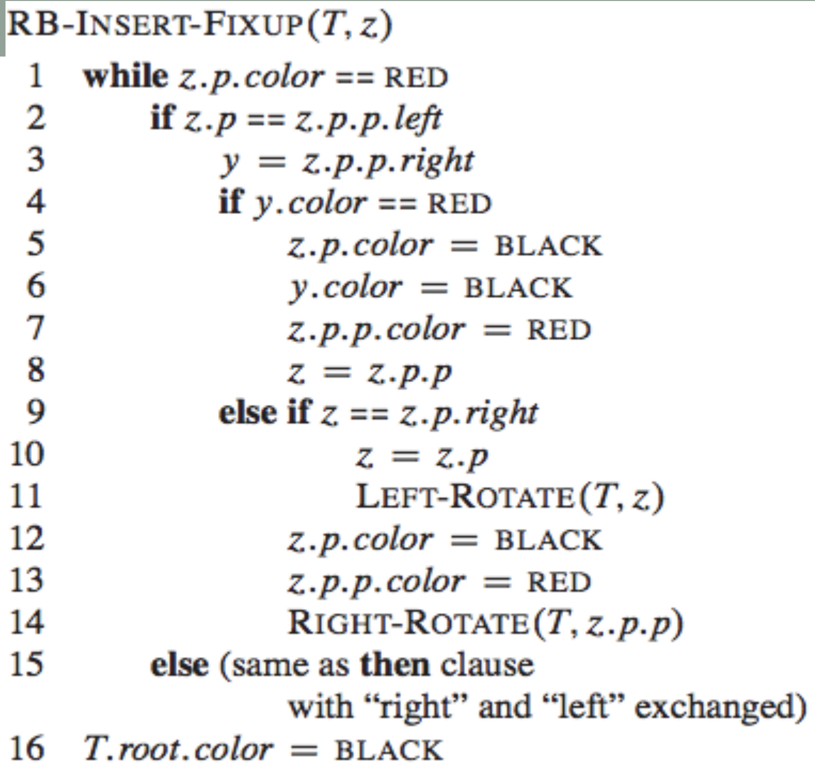
- INITIALIZATION



```
RB-INSERT-FIXUP($T, z$)
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 $z == z.p.right$
                $z = z.p$
                LEFT-ROTATE($T, z$)
            $z.p.color$ = BLACK
            $z.p.p.color$ = RED
            RIGHT-ROTATE($T, z.p.p$)
    else (same as then clause
            with "right" and "left" exchanged)
$T.root.color$ = BLACK
```

RB-FIXUP

- What is the loop invariant?
  a. Node z is red
  b. If z.p is the root, then z.p is black
  c. If the tree violates any of the red-black properties, it only violates one, and it must be:
     - 2) z is the root and is red
     - 4) z and its parent are both red



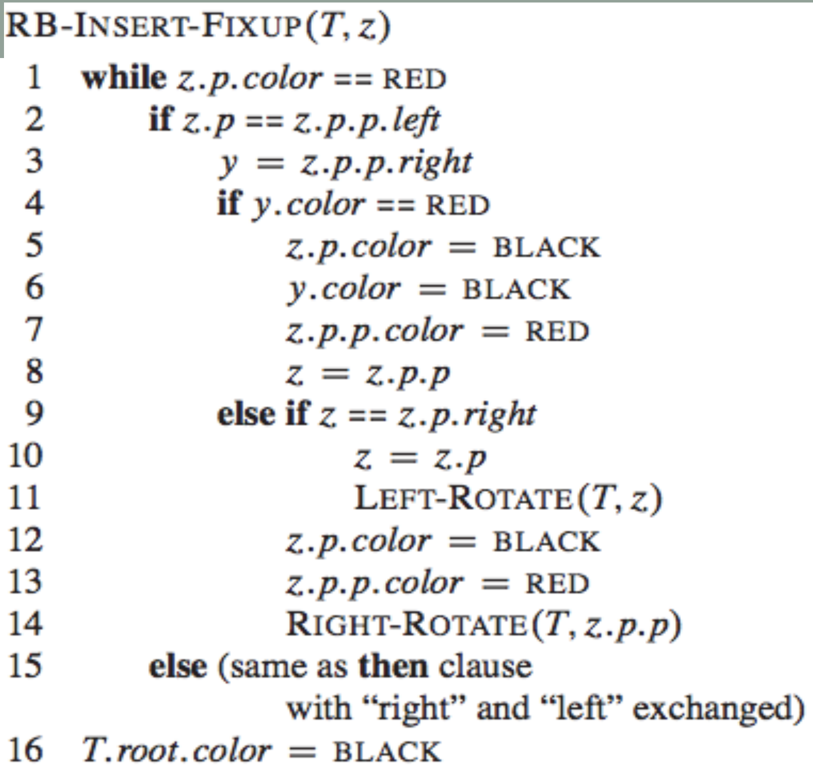
```
RB-INSERT-FIXUP(T, z)
while $z.p.color$ == RED
    if $z.p == z.p.p.left$
        $y = z.p.p.right$
        if $y.color$ == RED
            $z.p.color$ = BLACK
            $y.color$ = BLACK
            $z.p.p.color$ = RED
            $z = z.p.p$
        else if $z == z.p.right$
                $z = z.p$
                LEFT-ROTATE(T, z)
            $z.p.color$ = BLACK
            $z.p.p.color$ = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then clause
            with "right" and "left" exchanged)
$T.root.color$ = BLACK
```

- INITIALIZATION
  - The tree was previously valid
  - A. Node z must be red because we just inserted a red node.
  - B. The root was previously valid (black) and nothing happened to change it.
  - C. If it is the first node, it is a red root. Otherwise, the only possible violation is red-red.

RB-FIXUP

- What is the loop invariant?
    a. Node z is red
    b. If z.p is the root, then z.p is black
    c. If the tree violates any of the red-black properties, it only violates one, and it must be:
        - 2) z is the root and is red
        - 4) z and its parent are both red

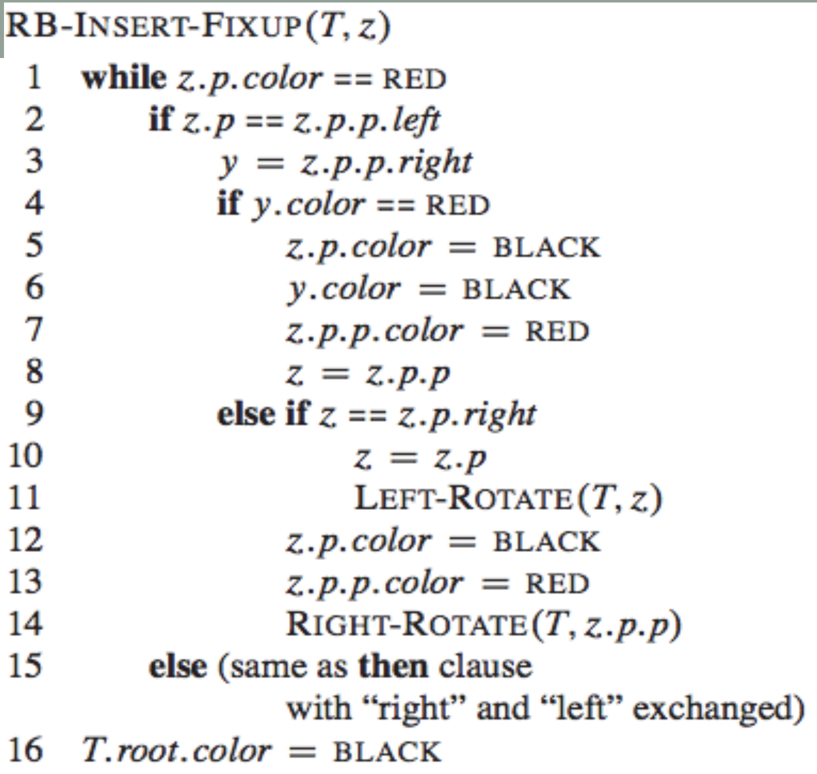
- TERMINATION



```
RB-INSERT-FIXUP(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
                LEFT-ROTATE(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then clause
            with "right" and "left" exchanged)
T.root.color = BLACK
```

RB-FIXUP

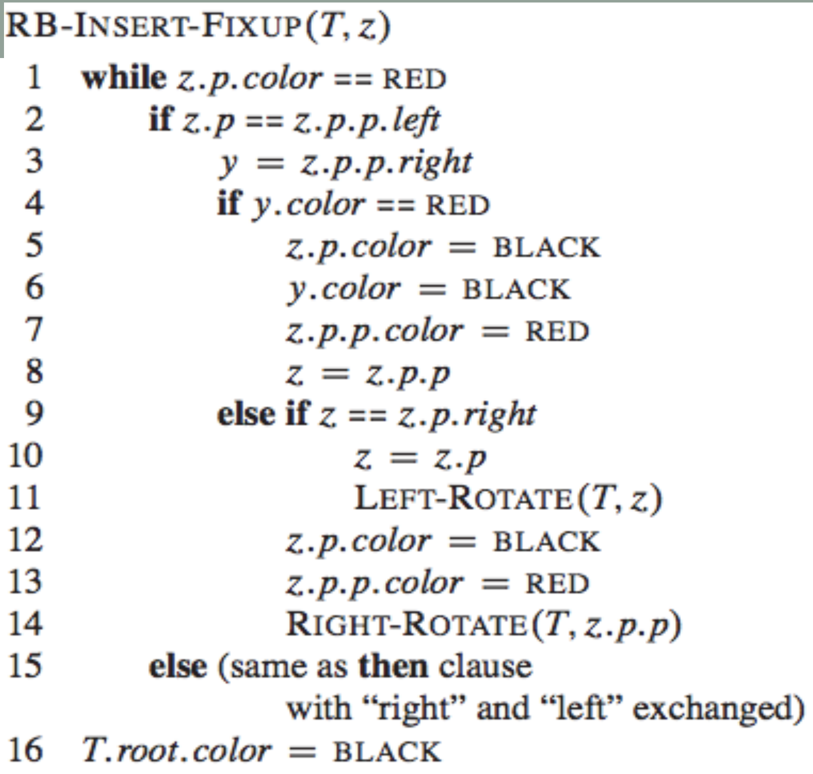
- What is the loop invariant?
  a. Node z is red
  b. If z.p is the root, then z.p is black
  c. If the tree violates any of the red-black properties, it only violates one, and it must be:
     - 2) z is the root and is red
     - 4) z and its parent are both red
- TERMINATION
- The loop ends when z.p is black (see line 12)
  - A) node z is still red; it's always red
  - B) z.p is black when the loop ends, so this must be true
  - C) At termination property 4 is not violated.
  - It's possible that property 2 is violated, but line 16 fixes it



```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK
 6              y.color = BLACK
 7              z.p.p.color = RED
 8              z = z.p.p
 9          else if z == z.p.right
10                  z = z.p
11                  LEFT-ROTATE(T, z)
12              z.p.color = BLACK
13              z.p.p.color = RED
14              RIGHT-ROTATE(T, z.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

RB-FIXUP

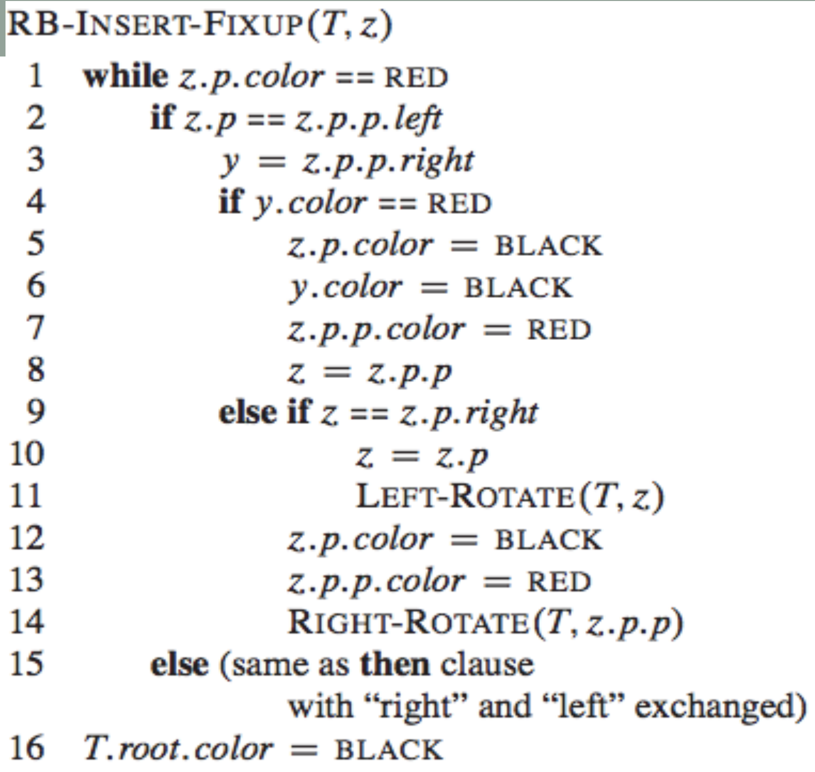
- What is the loop invariant?
  a. Node z is red
  b. If z.p is the root, then z.p is black
  c. If the tree violates any of the red-black properties, it only violates one, and it must be:
     - 2) z is the root and is red
     - 4) z and its parent are both red
- MAINTENANCE?
  - Must consider case1, case2, case3
  - Case1, recolor and move problem to grandparent:
    - A) lines 7 and 8 recolor gp to red and make it the new z, so property (A) is maintained
    - B) z.p moves to z.p.p.p (great-grandparent), which remains black if it was already
    - C) Property (4) could still be violated, if the great-grandparent is red.



```
RB-INSERT-FIXUP(T, z)
1  while z.p.color == RED
2      if z.p == z.p.p.left
3          y = z.p.p.right
4          if y.color == RED
5              z.p.color = BLACK
6              y.color = BLACK
7              z.p.p.color = RED
8              z = z.p.p
9          else if z == z.p.right
10                 z = z.p
11                 LEFT-ROTATE(T, z)
12             z.p.color = BLACK
13             z.p.p.color = RED
14             RIGHT-ROTATE(T, z.p.p)
15     else (same as then clause
               with "right" and "left" exchanged)
16 T.root.color = BLACK
```

RB-FIXUP

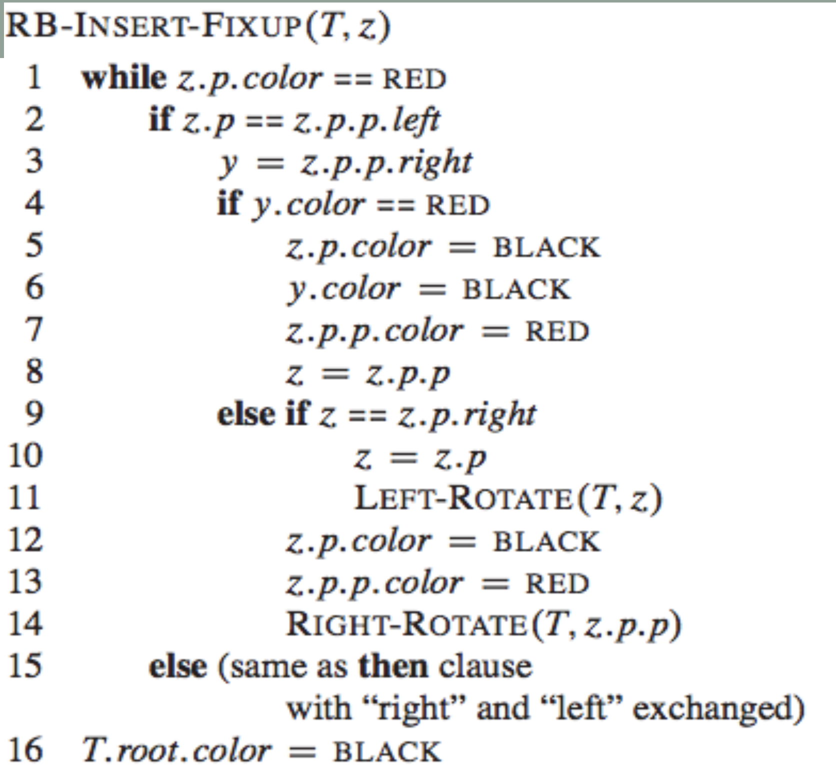
- What is the loop invariant?
  a. Node z is red
  b. If z.p is the root, then z.p is black
  c. If the tree violates any of the red-black properties, it only violates one, and it must be:
     - 2) z is the root and is red
     - 4) z and its parent are both red
- MAINTENANCE?
  - Case2, Black Uncle with Zig-Zag
  - A) Node z and its parent must be red
  - B) z.p can’t be the root or we don’t have a zig-zag
  - C) Rotating on z.p and z changes order but not color, we are still violating (4)



```
RB-INSERT-FIXUP(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
                LEFT-ROTATE(T, z)
            z.p.color = BLACK
            z.p.p.color = RED
            RIGHT-ROTATE(T, z.p.p)
    else (same as then clause
            with “right” and “left” exchanged)
T.root.color = BLACK
```

RB-FIXUP

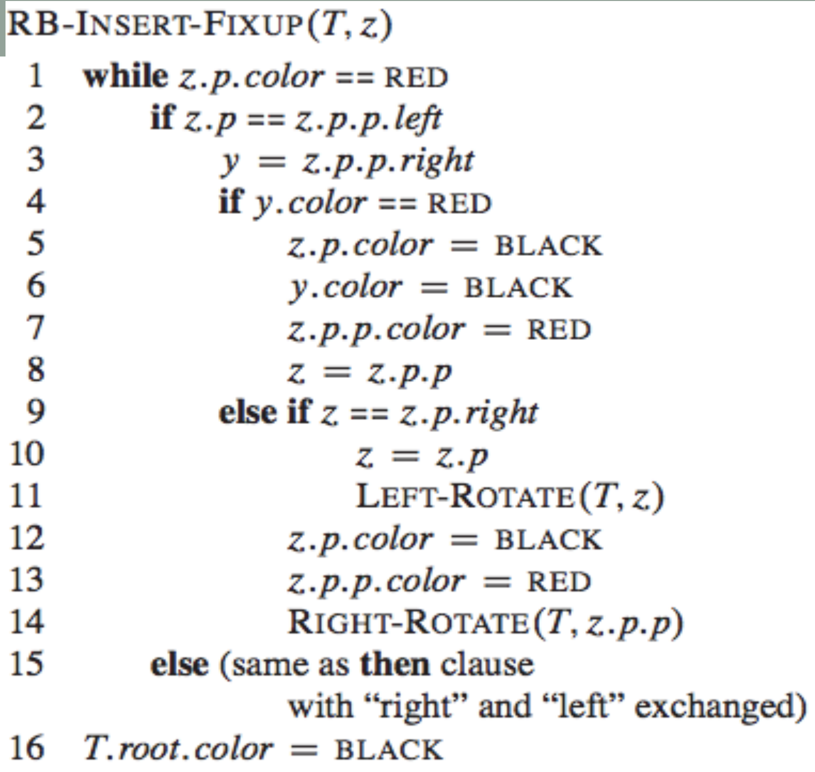
- What is the loop invariant?
  a. Node z is red
  b. If z.p is the root, then z.p is black
  c. If the tree violates any of the red-black properties, it only violates one, and it must be:
     - 2) z is the root and is red
     - 4) z and its parent are both red
- MAINTENANCE?
  - Case3, Black Uncle no Zig-Zag
  - A) if we make it to step 3, z.p is colored black, which ends the while loop; node z is never anything but red
  - B) z.p is black, so if it is the root, it is black
  - C) The violation of property 4 is corrected if we make it to case 3



```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK
 6              y.color = BLACK
 7              z.p.p.color = RED
 8              z = z.p.p
 9          else if z == z.p.right
10                  z = z.p
11                  LEFT-ROTATE(T, z)
12              z.p.color = BLACK
13              z.p.p.color = RED
14              RIGHT-ROTATE(T, z.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

RB-FIXUP

- How many times will the while loop execute?

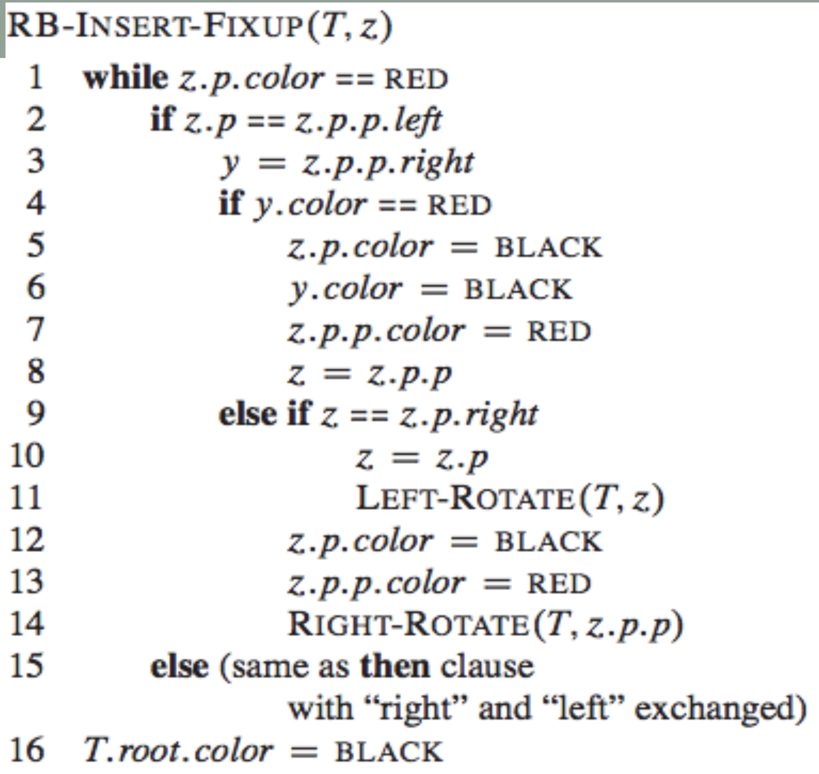


RB-INSERT-FIXUP(T, z)

```
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 $z == z.p.right$
                $z = z.p$
                LEFT-ROTATE$(T, z)$
            $z.p.color$ = BLACK
            $z.p.p.color$ = RED
            RIGHT-ROTATE$(T, z.p.p)$
    else (same as then clause
            with "right" and "left" exchanged)
$T.root.color$ = BLACK
```

RB-FIXUP

- How many times will the while loop execute?
  - If we make it to case2-3, the parent is recolored and the loop ends.
  - Case1 moves the problem up 2 levels.
  - So this could execute (lg n) / 2 times.
  - Insert is already O(lg n), so is fixup.
  - It does a max of 2 rotations.



```
RB-INSERT-FIXUP($T, z$)
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 $z == z.p.right$
                $z = z.p$
                LEFT-ROTATE($T, z$)
            $z.p.color$ = BLACK
            $z.p.p.color$ = RED
            RIGHT-ROTATE($T, z.p.p$)
    else (same as then clause
            with "right" and "left" exchanged)
$T.root.color$ = BLACK
```

Red-black tree insert

- Why is a new node set to red and not black?
- Show the red-black trees that result after successively inserting the keys 41, 38, 31, 12, 19, 8 into an initially empty red-black tree.

- Remember:
- Case0: parent is black
- Case1: Red uncle. Recolor parent, uncle and grandparent to move problem up tree.
- Case2: Black uncle with zig-zag: Rotate on child-parent to remove zig-zag.
- Case3: Black uncle with no zig-zag: Recolor parent and grandparent. Rotate on parent-grandparent.
- Case4: Red root: recolor root to back.

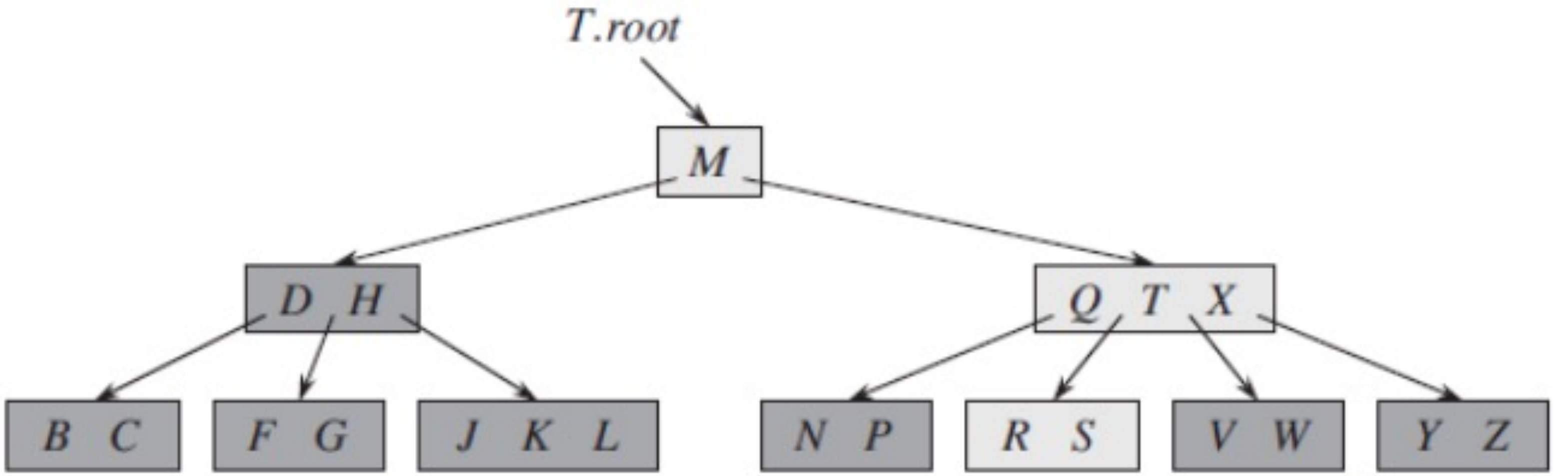
41, 38, 31, 12, 19, 8

Red Black Tree

- Insert the numbers 1-9 into a red-black tree

B-Trees

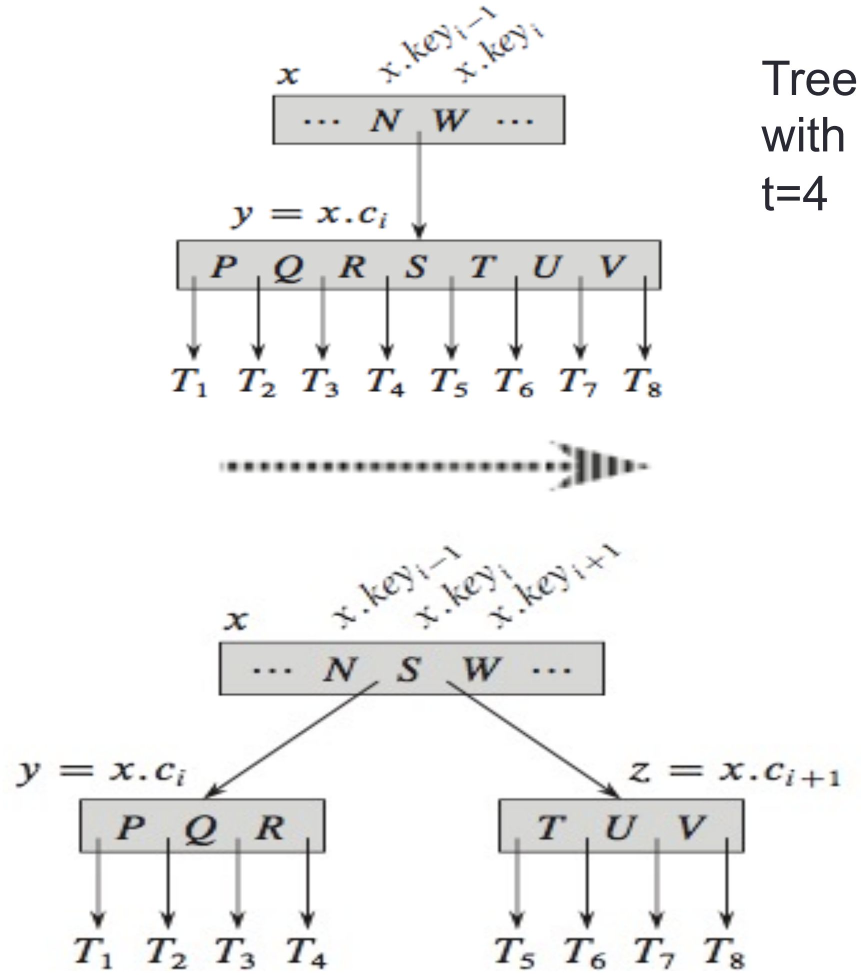
- a node containing n keys has n+1 children
- all leaves are at the same depth in the tree
- t=minimum degree of a node. Every node other than the root must have at least t children (and t-1 keys). They must also have at most 2t children.
- Every n-node B-tree has height $O(\log_t n)$. The branching factor is usually much larger than with binary trees.



B-Trees

- Why don’t we allow a minimum degree of t=1?
- Show all legal B-trees of minimum degree t=3 that represent {1, 2, 3, 4, 5}.

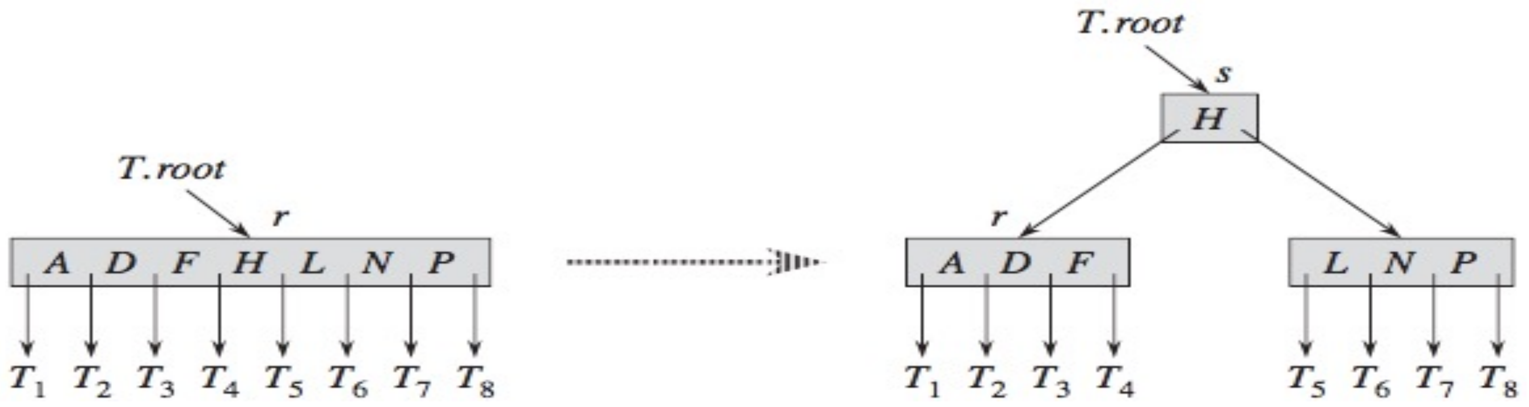
```
B-TREE-SPLIT-CHILD(x, i)
1   $z = \text{ALLOCATE-NODE}()$
2   $y = x.c_i$
3   $z.leaf = y.leaf$
4   $z.n = t - 1$
5   for $j = 1$ to $t - 1$
6       $z.key_j = y.key_{j+t}$
7   if not $y.leaf$
8       for $j = 1$ to $t$
9           $z.c_j = y.c_{j+t}$
10  $y.n = t - 1$
11  for $j = x.n + 1$ downto $i + 1$
12      $x.c_{j+1} = x.c_j$
13  $x.c_{i+1} = z$
14  for $j = x.n$ downto $i$
15      $x.key_{j+1} = x.key_j$
16  $x.key_i = y.key_t$
17  $x.n = x.n + 1$
18  DISK-WRITE(y)
19  DISK-WRITE(z)
20  DISK-WRITE(x)
```



Tree with t=4

In case of a full root

- Remember that the root is the only node that is allowed to have only two children as minimum.



B-Tree

- We are using a B-Tree to index a large amount of data stored on a hard drive. The hard drive is read in blocks of 4096 bits. The keys are 32-bit integers, and pointers to child nodes are 64-bits.
  - What is the maximum number of keys a node can hold?
  - What is the maximum number of pointers to child nodes a node can hold?
  - Each node in this B-Tree (except the root) must contain at least t-1 keys and at most 2t-1 keys. What is the value of t?
  - Leaf nodes contain key/pointer pairs, where the pointer points to the location of external data that corresponds to the key. If there are 1,000,000 items being indexed, what is the minimum number of leaf nodes the B-Tree must contain?
  - Assume we are using the B-Tree to store personal information about Americans, indexed by social security numbers, and there are 325.7 million Americans in the database. The first level of the tree is kept in memory, while successive levels require a disk access. How many disk accesses are required to obtain a piece of information and why?

Interview Questions

- Show the results of inserting the keys F,S,Q,K,C,L,H,T,V,W,M,R,N,P,A,B,X,Y,D,Z,E in order into an empty B-tree with minimum degree 3. Draw only the configurations of the tree just before some node must split, and also draw the final configuration.

- Explain how to find the minimum key stored in a B-tree and how to find the predecessor of a given key stored in a B-tree.

Interview Questions

- Suppose that we were to implement B-TREE-SEARCH to use binary search rather than linear search within each node. Show that this change changes the CPU time required from $O(t \log_t n)$ to $O(\lg n)$, independently of how t might be chosen as a function of n.
- You have eight balls all of the same size. 7 of them weigh the same, and one of them weighs slightly more. How can you find the ball that is heavier by using a balance and only two weighings?

Interview Questions

- The order of an internal node in a B tree index is the maximum number of children it can have. Suppose that a child pointer takes 6 bytes, the search field value takes 14 bytes, and the block size is 512 bytes. What is the maximum number of children of the internal node?

Interview Questions

- What's good about B-Trees? What's bad?