Introduction to Algorithms

Chapter 12 & 13: Binary, AVL and Red-Black Trees

Revisiting Trees
- Revisiting BST
- Introducing AVL trees
- Introducing Red-Black Trees

Types of Trees

- Binary Search Trees (BSTs) are an important data structure for dynamic sets
- In addition to satellite data, elements have:
  - key: an identifying field inducing a total ordering
  - left: pointer to a left child (may be NULL)
  - right: pointer to a right child (may be NULL)
  - p: pointer to a parent node (NULL for root)
Binary Search Trees

- BST property: \( \text{key}[\text{left}(x)] \leq \text{key}[x] \leq \text{key}[\text{right}(x)] \)
- Example:

```
A   F     H
B   D   K
```

Inorder Tree Walk

- **What does the following code do?**
  
  ```
  TreeWalk(x)
  TreeWalk(left[x]);
  print(x);
  TreeWalk(right[x]);
  ```

- A: prints elements in sorted (increasing) order
- This is called an *inorder tree walk*
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root

Operations on BSTs: Search

- Given a key and a pointer to a node, returns an element with that key or NULL:

  ```
  TreeSearch(x, k)
  if (x = NULL or k = key[x])
    return x;
  if (k < key[x])
    return TreeSearch(left[x], k);
  else
    return TreeSearch(right[x], k);
  ```

- Here’s another function that does the same:

  ```
  TreeSearch(x, k)
  while (x != NULL and k != key[x])
    if (k < key[x])
      x = left[x];
    else
      x = right[x];
  return x;
  ```
Operations of BSTs: Insert

- Adds an element x to the tree so that the binary search tree property continues to hold
- The basic algorithm
  - Like the search procedure above
  - Insert x in place of NULL
  - Use a “trailing pointer” to keep track of where you came from (like inserting into singly linked list)

BST Insert: Example

- Example: Insert C

BST Insert: Example

BST Search/Insert: Running Time

- What is the running time of TreeSearch() or TreeInsert()?
- A: $O(h)$, where $h$ = height of tree
- What is the height of a binary search tree?
- A: worst case: $h = O(n)$ when tree is just a linear string of left or right children
  - We’ll keep all analysis in terms of $h$ for now
  - Later we’ll see how to maintain $h = O(\lg n)$

Sorting With Binary Search Trees

- Informal code for sorting array A of length $n$:
  
  ```
  BSTSort(A)
  for i=1 to n
    TreeInsert(A[i]);
  InorderTreeWalk(root);
  ```

  This is $\Omega(n \lg n)$
  - What will be the running time in the
    - Worst case?
    - Average case?
Sorting With BSTs

- Average case analysis
  - It’s a form of quicksort!

```java
for i=1 to n
    TreeInsert(A[i]);
InorderTreeWalk(root);
```

3 1 8 2 6 7 5
1 2 8 6 7 5
2 6 7 5
5

Sorting with BSTs

- Same partitions are done as with quicksort, but in a different order
  - In previous example
    - Everything was compared to 3 once
    - Then those items < 3 were compared to 1 once
    - Etc.
  - Same comparisons as quicksort, different order!
    - Example: consider inserting 5
      - We need 3 comparisons, this is: $\log(n) = 3$ = tree height

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: $O(n \log n)$
  - *Which do you think is better, quicksort or BSTSort? Why?*

Sorting with BSTs

- Since run time is proportional to the number of comparisons, same time as quicksort: $O(n \log n)$
  - *Which do you think is better, quicksort or BSTSort? Why?*
  - A: quicksort
    - Better constants
    - Sorts in place
    - Doesn’t need to build data structure
More BST Operations

- BSTs are good for more than sorting. For example, can implement a priority queue

What operations must a priority queue have?
- Insert
- Minimum
- Extract-Min
  - For deletion, we will need a Successor() operation

BST Operations: Delete

- Deletion is a bit tricky
- 3 cases:
  - x has no children:
    - Remove x
  - x has one child:
    - Splice out x
  - x has two children:
    - Swap x with successor
    - Perform case 1 or 2 to delete it

AVL Trees

- Balanced binary search tree offer a O(log n) insert and delete.

- But balancing itself costs O(n) in the average case.

- Is there any way to have a O(log n) insert too?

- Yes, by almost but not fully balancing the tree: AVL (Adelson Velskii and Landis) balancing
**AVL Property**

- If N is a node in a binary tree, node N has AVL property if the heights of the left sub-tree and right sub-tree are equal or if they differ by 1.

**In other words: AVL trees**

AVL Trees: Balanced search tree

- Balance ensures that the search tree height will always be $\log n$
- AVL for Adelson-Velskii and Landis
- Definition: If T is a nonempty binary tree with $T_L$ and $T_R$ as its left and right subtrees, then T is an AVL tree iff
  - $T_L$ and $T_R$ are AVL trees
  - $|h_L - h_R| \leq 1$

**AVL Tree: Example**

```
  k
 / \
 g   i
 /   /
 e   h
     / \
    i  a
     \
      e
```
AVL Trees: Balanced Search tree

- Insertion: $O(\log n)$
- Deletion: $O(\log n)$
- Normally represented by a linked list
- Balance Factor ($bf$): The $bf(x)$ of node $x$ is defined to be as:
  
  $\text{Height of left subtree of } x - \text{height of right subtree of } x$

Red-Black Trees

- The red-black properties:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     - Note: this means every “real” node has 2 children
  3. If a node is red, both children are black
     - Note: can’t have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

In other words: Red-Black Tree

- A red-black tree can also be defined as a binary search tree that satisfies the following properties:
  - Root Property: the root is black
  - External Property: every leaf is black
  - Internal Property: the children of a red node are black
  - Depth Property: all the leaves have the same black depth

RB Trees: Worst-Case Time

- The red-black tree has $O(\lg n)$ height

- Corollary: These operations take $O(\lg n)$ time:
  - Minimum(), Maximum()
  - Successor(), Predecessor()
  - Search()

- Insert() and Delete():
  - Will also take $O(\lg n)$ time
  - But will need special care since they modify tree
Red-Black Trees: An Example

- Color this tree:

Red-black properties:
1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

Red-Black Trees: The Problem With Insertion

- Insert 8
  - Where does it go?
  - What color should it be?

Red-Black Trees: The Problem With Insertion

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

- Insert 11
  - Where does it go?
  - What color?

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black
Red-Black Trees: The Problem With Insertion

- Insert 11
  - Where does it go?
  - What color?
  - Solution: recolor the tree

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

- Insert 10
  - Where does it go?
  - What color?

1. Every node is either red or black
2. Every leaf (NULL pointer) is black
3. If a node is red, both children are black
4. Every path from node to descendent leaf contains the same number of black nodes
5. The root is always black

- Insert 10
  - Where does it go?
  - What color?
  - Goal: restructure tree in $O(\log n)$ time

- Insert 10
  - Where does it go?
Operations on RB Trees

- All operations can be performed in $O(\lg n)$ time.
- The query operations, which don’t modify the tree, are performed in exactly the same way as they are in BSTs.
- Insertion and Deletion are not straightforward.

Rotations

- Rotations are the basic tree-restructuring operation for almost all balanced search trees.
- Rotation takes a red-black-tree and a node.
- Changes pointers to change the local structure, and
- Won’t violate the binary-search-tree property.
- Left rotation and right rotation are inverses.

Left Rotation – Pseudo-code

```
Left-Rotate (T, x)
1. y ← right[x] // Set y.
2. right[x] ← left[y] // Turn y’s left subtree into x’s right subtree.
3. if left[y] ≠ nil[T ]
4. then p[left[y]] ← x
5. p[y] ← p[x] // Link x’s parent to y.
6. if p[x] = nil[T ]
7. then root[T ] ← y
8. else if x = left[p[x]]
9. then left[p[x]] ← y
10. else right[p[x]] ← y
11. left[y] ← x // Put x on y’s left.
12. p[x] ← y
```
Rotation

- The pseudo-code for Left-Rotate assumes that
  - right[x] ≠ nil[T], and
  - root’s parent is nil[T].

- Left Rotation on x, makes x the left child of y, and the left subtree of y into the right subtree of x.

- Pseudocode for Right-Rotate is symmetric: exchange left and right everywhere.

- **Time:** $O(1)$ for both Left-Rotate and Right-Rotate, since a constant number of pointers are modified.

Red-Black Trees: Insertion

- Insertion: the basic idea
  - Insert x into tree, color x red
  - Only r-b property 3 might be violated (if p[x] red)
    - If so, move violation up tree until a place is found where it can be fixed
  - Total time will be $O(\lg n)$
**Red-Black Insert**

- Insert node \( z \)
- \( z.\text{Color} = \text{red} \)
  - This can cause two possible problems:
    - Root must be black (initial insert violates this)
    - Two red nodes cannot be adjacent (this is violated if parent of \( z \) is red)
  - Cleanup

**Red-Black Cleanup**

- \( y = z \text{‘}s \text{“uncle”} \)
- Three cases:
  - \( y \) is red
  - \( y \) is black and \( z \) is a right child
  - \( y \) is black and \( z \) is a left child

**Case 1 – \( z \text{’}s \text{Uncle is red} \)**

- \( y.\text{Color} = \text{black} \)
- \( z.\text{Parent.\text{Color}} = \text{black} \)
- \( z.\text{Parent.\text{Parent.\text{Color}}} = \text{red} \)
- \( z = z.\text{Parent.\text{Parent}} \)
- Repeat cleanup

**Case 1 Visualized**
Case 1 Visualized

```
y.Color = black
z.Parent.Color = black
z = z.Parent.Parent
repeat cleanup

New Node
```

Case 1 Visualized

```
y.Color = black
z.Parent.Color = black
z = z.Parent.Parent
repeat cleanup

New Node
```

Case 1 Visualized

```
y.Color = black
z.Parent.Color = black
z = z.Parent.Parent
repeat cleanup

New Node
```

Case 1 Visualized

```
y.Color = black
z.Parent.Color = black
z = z.Parent.Parent
repeat cleanup

New Node
```
Case 2 – z’s Uncle is black & z is a right child
- \( z = z.\text{Parent} \)
- Left-Rotate(T, z)
- Do Case 3
- Note that Case 2 is a subset of Case 3
Case 3 – z’s Uncle is black & z is a left child
- z.Parent.Color = black
- z.Parent.Parent.Color = red
- Right-Rotate(T, z.Parent.Parent)

Case 3 Visualized
Case 3 Visualized

Trees - Red-Black Trees

- **Data structure**
  - As we'll see, nodes in red-black trees need to know their parents,
  - so we need this data structure

```c
struct t_red_black_node {
    enum { red, black } color;
    void *item;
    struct t_red_black_node *left,
    *right,
    *parent;
}
```

Trees - Insertion

- Insertion of a new node
  - Requires a re-balance of the tree

```c
rb_insert( Tree T, node x ) {
    /* Insert in the tree in the usual way */
    tree_insert( T, x );
    /* Now restore the red-black property */
    x->color = red;
    while ( (x != T->root) && (x->parent->color == red) ) {
        if ( x->parent == x->parent->parent->left ) {
            y = x->parent->parent->right;if ( y->color == red ) {
                /* case 1 - change the colors */
                x->parent->color = red;
                y->color = black;
                x->parent->parent->color = red;
                /* Move x up the tree */
                x = x->parent->parent;
            } else {
                /* case 2 - a single rotation */
                Right-Rotate(T, z, Parent.Parent);
            }
        } else if ( x->parent == x->parent->parent->right ) {
            z = x->parent->parent->left;
            if ( z->color == red ) {
                /* case 3 - a double rotation */
                Right-Rotate(T, z, Parent.Parent);
                z = z->parent;
                Right-Rotate(T, z, Parent.Parent);
            }
        }
    }
}
```
rb_insert(T, x) {
    /* Insert in the tree in the usual way */
    tree_insert(T, x);
    /* Now restore the red-black property */
    x->color = red;
    while ((x != T->root) && (x->parent->color == red)) {
        if (x->parent == x->parent->parent->left) {
            /* If x's parent is a left, y is x's right 'uncle' */
            y = x->parent->parent->right;
            if (y->color == red) {
                /* case 1 - change the colors */
                x->parent->color = black;
                y->color = black;
                x->parent->parent->color = red;
                /* Move x up the tree */
                x = x->parent->parent;
            } else {
                /* If x is to the left of its grandparent */
                x->parent->parent->color = black;
                x->parent->color = black;
                x->color = red;
                /* Move x up the tree */
                x = x->parent->parent;
            }
        } else {
            /* If x's parent is a right, y is x's left 'uncle' */
            y = x->parent->parent->left;
            if (y->color == red) {
                /* case 1 - change the colors */
                x->parent->color = black;
                y->color = black;
                y->parent->color = red;
                /* Move x up the tree */
                x = x->parent;
            } else {
                /* If the uncle is red, change the colors of y, the grand-parent and the parent */
                y->color = black;
                x->parent->color = black;
                x->color = red;
                /* Move x up the tree */
                x = x->parent;
            }
        }
    }
}

/* Now restore the red-black property */
while ((x != T->root) && (x->parent->color == red)) {
    if (x->parent == x->parent->parent->left) {
        /* If x's parent is a left, y is x's right 'uncle' */
        y = x->parent->parent->right;
        if (y->color == red) {
            /* case 1 - change the colors */
            x->parent->color = black;
            y->color = black;
            x->parent->parent->color = red;
            /* Move x up the tree */
            x = x->parent->parent;
        } else {
            /* If x is to the left of its grandparent */
            x->parent->parent->color = black;
            x->parent->color = black;
            x->color = red;
            /* Move x up the tree */
            x = x->parent->parent;
        }
    } else {
        /* If x's parent is a right, y is x's left 'uncle' */
        y = x->parent->parent->left;
        if (y->color == red) {
            /* case 1 - change the colors */
            x->parent->color = black;
            y->color = black;
            y->parent->color = red;
            /* Move x up the tree */
            x = x->parent;
        } else {
            /* If the uncle is red, change the colors of y, the grand-parent and the parent */
            y->color = black;
            x->parent->color = black;
            x->color = red;
            /* Move x up the tree */
            x = x->parent;
        }
    }
}

/* Now restore the red-black property */
while ((x != T->root) && (x->parent->color == red)) {
    if (x->parent == x->parent->parent->left) {
        /* If x's parent is a left, y is x's right 'uncle' */
        y = x->parent->parent->right;
        if (y->color == red) {
            /* case 1 - change the colors */
            x->parent->color = black;
            y->color = black;
            x->parent->parent->color = red;
            /* Move x up the tree */
            x = x->parent->parent;
        } else {
            /* If x is to the left of its grandparent */
            x->parent->parent->color = black;
            x->parent->color = black;
            x->color = red;
            /* Move x up the tree */
            x = x->parent->parent;
        }
    } else {
        /* If x's parent is a right, y is x's left 'uncle' */
        y = x->parent->parent->left;
        if (y->color == red) {
            /* case 1 - change the colors */
            x->parent->color = black;
            y->color = black;
            y->parent->color = red;
            /* Move x up the tree */
            x = x->parent;
        } else {
            /* If the uncle is red, change the colors of y, the grand-parent and the parent */
            y->color = black;
            x->parent->color = black;
            x->color = red;
            /* Move x up the tree */
            x = x->parent;
        }
    }
}

/* Now restore the red-black property */
while ((x != T->root) && (x->parent->color == red)) {
    if (x->parent == x->parent->parent->left) {
        /* If x's parent is a left, y is x's right 'uncle' */
        y = x->parent->parent->right;
        if (y->color == red) {
            /* case 1 - change the colors */
            x->parent->color = black;
            y->color = black;
            x->parent->parent->color = red;
            /* Move x up the tree */
            x = x->parent->parent;
        } else {
            /* If x is to the left of its grandparent */
            x->parent->parent->color = black;
            x->parent->color = black;
            x->color = red;
            /* Move x up the tree */
            x = x->parent->parent;
        }
    } else {
        /* If x's parent is a right, y is x's left 'uncle' */
        y = x->parent->parent->left;
        if (y->color == red) {
            /* case 1 - change the colors */
            x->parent->color = black;
            y->color = black;
            y->parent->color = red;
            /* Move x up the tree */
            x = x->parent;
        } else {
            /* If the uncle is red, change the colors of y, the grand-parent and the parent */
            y->color = black;
            x->parent->color = black;
            x->color = red;
            /* Move x up the tree */
            x = x->parent;
        }
    }
}
while ( (x != T->root) && (x->parent->color == red) ) {
    if ( x->parent == x->parent->parent->left ) {
        /* If x's parent is a left, y is x's right 'uncle' */
        y = x->parent->parent->right;
        if ( y->color == red ) {
            /* case 1 - change the colors */
            x->parent->color = black;
            y->color = black;
            x->parent->parent->color = red;
            x = x->parent->parent;
        } else {
            /* y is a black node */
            if ( x == x->parent->right ) {
                /* and x is to the right */
                /* case 2 - move x up and rotate */
                x = x->parent;
                left_rotate( T, x );
            }
        }
    } else {
        /* y is a black node */
        if ( x == x->parent->right ) {
            /* and x is to the right */
            /* case 2 - move x up and rotate */
            x = x->parent;
            left_rotate( T, x );
        }
    }
}
while ( (x != T->root) && (x->parent->color == red) ) {
  if ( x->parent == x->parent->parent->left ) {
    y = x->parent->parent->right;
    if ( y->color == red ) {
      /* case 1 - change the colors */
      x->parent->color = black;
      y->color = black;
      x->parent->parent->color = red;
      /* Move x up the tree */
      x = x->parent->parent;
    } else {
      /* y is a black node */
      if ( x == x->parent->right ) {
        /* and x is to the right */
        /* case 2 - move x up and rotate */
        left_rotate( T, x );
      } else {
        /* case 3 */
        x->parent->color = black;
        x->parent->parent->color = red;
        right_rotate( T, x->parent->parent );
      }
    }
  } else {
    /* y is a black node */
    if ( x == x->parent->left ) {
      /* and x is to the left */
      /* case 3 */
      right_rotate( T, x );
    } else {
      /* case 3 */
      x->parent->color = black;
      x->parent->parent->color = red;
      right_rotate( T, x->parent->parent );
    }
  }
}

Trees - Insertion

while ( (x != T->root) && (x->parent->color == red) ) {
    if (x->parent == x->parent->parent->left) {
        /* If x's parent is a left, y is x's right 'uncle' */
        y = x->parent->parent->right;
        if (y->color == red) {
            /* case 1 - change the colors */
            x->parent->color = black;
            y->color = black;
            x->parent->parent->color = red;
            /* Move x up the tree */
            x = x->parent->parent;
        } else { /* y is a black node */
            if (x == x->parent->right) {
                /* and x is to the right */
                /* case 2 - move x up and rotate */
                x = x->parent;
                left_rotate(T, x);
            } else { /* case 3 */
                x->parent->color = black;
                x->parent->parent->color = red;
                right_rotate(T, x->parent->parent);
            }
        }
    } else { /* case 3 */
        x->parent->color = black;
        x->parent->parent->color = red;
        right_rotate(T, x->parent->parent);
    }
}

Red-black trees - Analysis

- Addition
  - Insertion Comparisons $O(\log n)$
- Fix-up
  - At every stage, x moves up the tree at least one level $O(\log n)$
- Overall $O(\log n)$
- Deletion
  - Also $O(\log n)$
- More complex
  - ... but gives $O(\log n)$ behavior
Another version of insertion

```
RB-INSERT(T, z)
1  y = nil(T)
2  x ← root[T]
3  while x ≠ nil(T]
4    do y ← x
5      if key[y] < key[x]
6        then x ← left[x]
7      else x ← right[x]
8      p[z] ← y
9      if y = nil(T]
10     then root[T] ← z
11    else if key[z] < key[y]
12     then left[y] ← z
13     else right[y] ← z
14     left[z] ← nil(T]
15     right[z] ← nil(T]
16     color[z] ← red
17  RB-INSERT-FIXUP(T, z)
```

Start by doing regular binary-search-tree insertion:

- **RB-INSERT** ends by coloring the new node z red.
- Then it calls RB-INSERT-FIXUP because we could have violated a red-black property.

**Which property might be violated?**
1. OK.
2. If z is the root, then there's a violation. Otherwise, OK.
3. OK.
4. If p[z] is red, there's a violation: both z and p[z] are red.
5. OK.

- Remove the violation by calling RB-INSERT-FIXUP:

### Loop Invariant

At the start of each iteration of the while loop,

- **z** is red.
- There is at most one red-black violation:
  - Property 2: z is a red root, or
  - Property 4: z and p[z] are both red.

#### Initialization:
We've already seen why the loop invariant holds initially.

#### Termination:
The loop terminates because p[z] is black. Hence, property 4 is OK.

Only property 2 might be violated and the last line fixes it.

#### Maintenance:
We drop out when z is the root (since then p[z] is the sentinel nil[T], which is black). When we start the loop body, the only violation is of property 4. There are 6 cases
- 3 of which are symmetric to the other 3.
- The cases are not mutually exclusive.

Consider cases in which p[z] is a left child.

Let y be z's uncle (p[z]'s sibling).

#### Case 1: y is red
- p[z] (z's grandparent) must be black, since z and p[z] are both red and there are no other violations of property 4.
- Make p[z] and y black ⇒ now z and p[z] are not both red. But property 5 might now be violated.
- Make p[z] red ⇒ restores property 5.
- The next iteration has p[z] as the new z (i.e., z moves up 2 levels).

#### Case 2: y is black, z is a right child
- Left rotate around p[z] ⇒ now z is a left child, and both z and p[z] are red.
- Takes us immediately to case 3.

#### Case 3: y is black, z is a left child
- Make p[z] black and p[z] red.
- Then right rotate on p[z].
- No longer have 2 reds in a row.
- p[z] is now black ⇒ no more iterations.

### Analysis
- **O(lg n)** time to get through RB-INSERT up to the call of RB-INSERT-FIXUP.

#### Within RB-INSERT-FIXUP:
- Each iteration takes **O(1)** time.
- Each iteration is either the last one or it moves z up 2 levels.
- **O(lg n)** levels ⇒ **O(lg n)** time.

Thus, insertion into a red-black tree takes **O(lg n)** time.
RB-INSERT-FIXUP(T, z)
1 while color[p(z)] = RED
2    do if p[z] = left[p[p[z]]]
3       then y ← right[p[p[z]]]
4       if color[y] = RED
5          then color[p[z]] ← BLACK  ▷ Case 1
6            color[y] ← BLACK      ▷ Case 1
7            color[p[z]] ← RED     ▷ Case 1
8            z ← p[p[z]]          ▷ Case 1
9       else if z = right[p[z]]
10          then z ← p[z]          ▷ Case 2
11                  LEFT-ROTATE(T, z)  ▷ Case 2
12          color[p[z]] ← BLACK    ▷ Case 3
13          color[p[z]] ← RED      ▷ Case 3
14          RIGHT-ROTATE(T, p[p[z]]) ▷ Case 3
15 else (same as then clause
16       with “right” and “left” exchanged)
17 color[root(T)] ← BLACK

Figure 13.4 The operation of RB-INSERT-FIXUP. (a) A node z after insertion. Since z and its parent p(z) are both red, a violation of property 4 occurs. Since z’s sister y is red, case 1 in the code can be applied. Nodes are colored red and the process is continued until the resulting tree is shown in (b). Once again, z and its parent are both red, but z’s sister y is black. Since z’s the right child of p(z), case 2 can be applied. A left rotation is performed, and the new tree is shown in (c). Now z is the left child of its parent, and case 3 can be applied. A right rotation yields the tree in (d), which is a legal red-black tree.

Figure 13.5 Case 1 of the procedure RB-INSERT. Property 4 is violated, since z and its parent p(z) are both red. The same action is taken whether (a) z is a right child or (b) z is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 5: all downward paths from a node to a leaf have the same number of blacks. The while loop continues with node z’s grandparent p[p(z)] as the new z. Any violation of property 4 can now occur only between the new z, which is red, and its parent, if it is red as well.

Figure 13.6 Cases 2 and 3 of the procedure RB-INSERT. As in case 1, property 4 is violated in either case 2 or case 3 because z and its parent p[z] are both red. Each of the subtrees α, β, γ, δ, and ε has a black root (α, β, and γ from property 4, and δ because otherwise we would be in case 1), and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 5: all downward paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 5. The while loop then terminates, because property 4 is satisfied: there are no longer two red nodes in a row.
Red-Black Trees: Deletion

- Deletion, like insertion, should preserve all the RB properties.
- The properties that may be violated depends on the color of the deleted node.
  - Red – OK. Why?
  - Black? Why?

Steps:
- Do regular BST deletion.
- Fix any violations of RB properties that may result.

Deletion

$\text{RB-Delete}(T, z)$ (Contd.)

1. if $p[y] = \text{nil}[T]$ then $x \leftarrow x$
2. else if $y = \text{left}[p[y]]$ then $x \leftarrow x$
3. else right[$p[y]$] \leftarrow $x$
4. if $y \neq z$ then $\text{key}[z] \leftarrow \text{key}[y]$
5. copy $y$’s satellite data into $z$
6. if color[$y$] = BLACK then $\text{RB-Delete-Fixup}(T, x)$
7. return $y$

Deletion

$\text{RB-Delete}(T, z)$

1. if $\text{left}[z] = \text{nil}[T]$ or $\text{right}[z] = \text{nil}[T]$
2. then $y \leftarrow z$
3. else $y \leftarrow \text{TREE-SUCCESSOR}(z)$
4. if $\text{left}[y] \neq \text{nil}[T]$
5. then $x \leftarrow \text{left}[y]$
6. else $x \leftarrow \text{right}[y]$
7. $p[x] \leftarrow p[y]$ // Do this, even if $x$ is $\text{nil}[T]$

Tree Successor

$\text{TREE-SUCCESSOR}(x)$

1. if $\text{right}[x] \neq \text{NIL}$
2. then return $\text{TREE-MINIMUM}(\text{right}[x])$
3. $y \leftarrow p[x]$
4. while $y \neq \text{NIL}$ and $x = \text{right}[y]$
5. do $x \leftarrow y$
6. $y \leftarrow p[y]$
7. return $y$
**RB Properties Violation**

- If $y$ is black, we could have violations of red-black properties:
  - Prop. 1. OK.
  - Prop. 2. If $y$ is the root and $x$ is red, then the root has become red.
  - Prop. 3. OK.
  - Prop. 4. Violation if $p[y]$ and $x$ are both red.
  - Prop. 5. Any path containing $y$ now has 1 fewer black node.

**Deletion – Fixup**

```
RB-Delete-Fixup(T, x) (Contd.)
/* x is still left[p[x]] */
9. if color[left[w]] = BLACK and color[right[w]] = BLACK
   then color[w] ← BLACK // Case 2
   x ← p[x] // Case 2
10. else if color[right[w]] = BLACK
   then color[left[w]] ← BLACK // Case 3
   color[w] ← RED // Case 3
   RIGHT-ROTATE(T, w) // Case 3
   w ← right[p[x]] // Case 3
11. else (same as then clause with "right" and "left" exchanged)
    color[w] ← color[p[x]] // Case 4
   color[p[x]] ← BLACK // Case 4
   color[right[w]] ← BLACK // Case 4
   LEFT-ROTATE(T, p[x]) // Case 4
   x ← root[T] // Case 4
12. color[x] ← BLACK
```

**RB Properties Violation**

- Prop. 5. Any path containing $y$ now has 1 fewer black node.
  - Correct by giving $x$ an “extra black.”
  - Add 1 to count of black nodes on paths containing $x$.
  - Now property 5 is OK, but property 1 is not.
  - $x$ is either **doubly black** (if color$[x] = BLACK$) or **red & black** (if color$[x] = RED$).
  - The attribute color$[x]$ is still either RED or BLACK. No new values for color attribute.

Remove the violations by calling RB-Delete-Fixup.
Deletion – Fixup

- **Idea:** Move the extra black up the tree until \( x \) points to a red & black node ⇒ turn it into a black node,
- \( x \) points to the root ⇒ just remove the extra black, or
- We can do certain rotations and re-colorings and finish.
- Within the while loop:
  - \( x \) always points to a non-root doubly black node.
  - \( w \) is \( x \)'s sibling.
  - \( w \) cannot be \( \text{nil} \), since that would violate property 5 at \( p[x] \).
- 8 cases in all, 4 of which are symmetric to the other.

---

Case 1 – \( w \) is red

- \( w \) is red ⇒ it has black children
- switch colors of \( w, p[w] \) and left-rotate \( p[w] \)
- Result: one of the other cases

---

In other words :Case 1 – \( w \) is red

- \( w \) must have black children.
- Make \( w \) black and \( p[x] \) red (because \( w \) is red \( p[x] \) couldn’t have been red).
- Then left rotate on \( p[x] \).
- New sibling of \( x \) was a child of \( w \) before rotation ⇒ must be black.
- Go immediately to case 2, 3, or 4.

---

Case 2 – \( w \) is black, both \( w \)'s children are black

- \( w \) is black with black children
- Repeat the whole procedure for \( p[x] \) as the new \( x \)
**Case 3** – \( w \) is black, \( w \)'s left child is red, \( w \)'s right child is black

- Make \( w \) red and \( w \)'s left child black.
- Then right rotate on \( w \).
- New sibling \( w \) of \( x \) is black with a red right child \( \Rightarrow \) case 4.

**Case 4** – \( w \) is black, \( w \)'s right child is red

- Make \( w \) be \( \rho[x] \)'s color (\( c \)).
- Make \( \rho[x] \) black and \( w \)'s right child black.
- Then left rotate on \( \rho[x] \).
- Remove the extra black on \( x \) (\( \Rightarrow x \) is now singly black) without violating any red-black properties.
- All done. Setting \( x \) to root causes the loop to terminate.

---

**Analysis**

- \( \mathcal{O}(\lg n) \) time to get through RB-Delete up to the call of RB-Delete-Fixup.
- Within RB-Delete-Fixup:
  - Case 2 is the only case in which more iterations occur.
    - \( x \) moves up 1 level.
    - Hence, \( \mathcal{O}(\lg n) \) iterations.
  - Each of cases 1, 3, and 4 has 1 rotation \( \Rightarrow \leq 3 \) rotations in all.
  - Hence, \( \mathcal{O}(\lg n) \) time.