2-15 Red-black Tree

Jun Ma

majun@nju.edu.cn

September 16, 2020

TC-13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

TC-13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Proof.

Assume x has are two different descendant leaves a and b, and $len(x, a) > 2len(x, b)$

TC-13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Proof.

Assume x has are two different descendant leaves a and b, and $len(x, a) > 2len(x, b)$

- $len(x, a) = \# \bigcirc$ in $(x, a) + \# \bigcirc$ in $(x, a) - 1$

TC-13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Proof.

Assume x has are two different descendant leaves a and b, and $len(x, a) > 2len(x, b)$

- $len(x, a) = \#\bigcirc$ in $(x, a) + \#{\color{red}\bigcirc}$ in $(x, a) - 1$
- $len(x, b) = \#\bigcirc$ in $(x, b) + \#{\color{red}\bigcirc}$ in $(x, b) - 1$

TC-13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Proof.

Assume x has are two different descendant leaves a and b, and $len(x,a) > 2len(x,b)$

- $len(x,a) = \#\bigcirc$ in $(x,a) + \#{\color{red}\bigcirc}$ in $(x,a) - 1$
- $len(x,b) = \#\bigcirc$ in $(x,b) + \#{\color{red}\bigcirc}$ in $(x,b) - 1$
- $\#\bigcirc$ in $(x,a) = len_{black}(x,a) = len_{black}(x,b) = \#\bigcirc$ in (x,b)

TC-13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Proof.

Assume x has are two different descendant leaves a and b, and $len(x,a) > 2len(x,b)$

- $len(x,a) = \# \bigcirc$ in $(x,a) + \#{\color{red}\bigcirc}$ in $(x,a) - 1$
- $len(x,b) = \# \bigcirc$ in $(x,b) + \#{\color{red}\bigcirc}$ in $(x,b) - 1$
- $\# \bigcirc$ in $(x,a) = len_{black}(x,a) = len_{black}(x,b) = \# \bigcirc$ in (x,b)
- $\#{\color{red}\bigcirc}$ in $(x,a) > \# \bigcirc$ in $(x,a) + 2\#{\color{red}\bigcirc}$ in $(x,b) - 1$

TC-13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Proof.

Assume x has are two different descendant leaves a and b, and $len(x,a) > 2len(x,b)$

- $len(x,a) = \#\bigcirc$ in $(x,a) + \#{\color{red}\bigcirc}$ in $(x,a) - 1$
- $len(x,b) = \#\bigcirc$ in $(x,b) + \#{\color{red}\bigcirc}$ in $(x,b) - 1$
- $\#\bigcirc$ in $(x,a) = len_{black}(x,a) = len_{black}(x,b) = \#\bigcirc$ in (x,b)
- $\#{\color{red}\bigcirc}$ in $(x,a) > \#\bigcirc$ in $(x,a) + 2\#{\color{red}\bigcirc}$ in $(x,b) - 1$
- $\#{\color{red}\bigcirc}$ in $(x,a) \geq \#\bigcirc$ in (x,a).

TC-13.1-5

Show that the longest simple path from a node x in a red-black tree to a descendant leaf has length at most twice that of the shortest simple path from node x to a descendant leaf.

Proof.

Assume x has are two different descendant leaves a and b, and $len(x, a) > 2len(x, b)$

- $len(x, a) = \# \bigcirc$ in $(x, a) + \# {\color{red}\bigcirc}$ in $(x, a) - 1$
- $len(x, b) = \# \bigcirc$ in $(x, b) + \# {\color{red}\bigcirc}$ in $(x, b) - 1$
- $\# \bigcirc$ in $(x, a) = len_{black}(x, a) = len_{black}(x, b) = \# \bigcirc$ in (x, b)
- $\# {\color{red}\bigcirc}$ in $(x, a) > \# \bigcirc$ in $(x, a) + 2\# {\color{red}\bigcirc}$ in $(x, b) - 1$
- $\# {\color{red}\bigcirc}$ in $(x, a) \geq \# \bigcirc$ in (x, a). Impossible!

□

TC-13.1-7

Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

TC-13.1-7

Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

Answer.

TC-13.1-7

Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

Answer.

- Largest: a tree with 3 three nodes and the root is the only black one. The ratio is 2.

TC-13.1-7

Describe a red-black tree on n keys that realizes the largest possible ratio of red internal nodes to black internal nodes. What is this ratio? What tree has the smallest possible ratio, and what is the ratio?

Answer.

- Largest: a tree with 3 three nodes and the root is the only black one. The ratio is 2.
- Smallest: a tree with only a (black) root node. The ratio is 0

□

TC-13.3-1

In line 16 of RB-INSERT, we set the color of the newly inserted node z to red. Observe that if we had chosen to set z's color to black, then property 4 of a red-black tree would not be violated. Why didn't we choose to set z's color to black?

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

TC-13.3-1

In line 16 of RB-INSERT, we set the color of the newly inserted node z to red. Observe that if we had chosen to set z's color to black, then property 4 of a red-black tree would not be violated. Why didn't we choose to set z's color to black?

Answer.

P5 is violated!

> P5. For each node, all simple paths from the node to descendant leaves contain the same number of black nodes.

□

RB-INSERT(T, z)

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

TC-13.3-5

Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

```
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
```

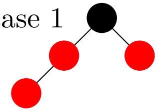
TC-13.3-5

Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

```
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
```

case 1



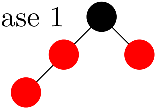
TC-13.3-5

Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

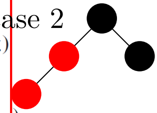
```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

```
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
```

case 1



case 2



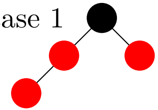
TC-13.3-5

Consider a red-black tree formed by inserting n nodes with RB-INSERT. Argue that if $n > 1$, the tree has at least one red node.

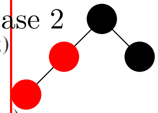
```
RB-INSERT(T, z)
y = T.nil
x = T.root
while x ≠ T.nil
    y = x
    if z.key < x.key
        x = x.left
    else x = x.right
z.p = y
if y == T.nil
    T.root = z
elseif z.key < y.key
    y.left = z
else y.right = z
z.left = T.nil
z.right = T.nil
z.color = RED
RB-INSERT-FIXUP(T, z)
```

```
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
```

case 1



case 2



TC-13.4-1

Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

```
RB-DELETE(T, z)
y = z
y-original-color = y.color
if z.left == T.nil
    x = z.right
    RB-TRANSPLANT(T, z, z.right)
elseif z.right == T.nil
    x = z.left
    RB-TRANSPLANT(T, z, z.left)
else y = TREE-MINIMUM(z.right)
    y-original-color = y.color
    x = y.right
    if y.p == z
        x.p = y
    else RB-TRANSPLANT(T, y, y.right)
        y.right = z.right
        y.right.p = y
    RB-TRANSPLANT(T, z, y)
    y.left = z.left
    y.left.p = y
    y.color = z.color
if y-original-color == BLACK
    RB-DELETE-FIXUP(T, x)
```

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                  // case 1
            x.p.color = RED                                  // case 1
            LEFT-ROTATE(T, x.p)                              // case 1
            w = x.p.right                                    // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                    // case 2
            x = x.p                                          // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                         // case 3
                w.color = RED                                // case 3
                RIGHT-ROTATE(T, w)                           // case 3
                w = x.p.right                                // case 3
            w.color = x.p.color                              // case 4
            x.p.color = BLACK                                // case 4
            w.right.color = BLACK                            // case 4
            LEFT-ROTATE(T, x.p)                              // case 4
            x = T.root                                       // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

TC-13.4-1

Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                  // case 1
            x.p.color = RED                                  // case 1
            LEFT-ROTATE(T, x.p)                              // case 1
            w = x.p.right                                    // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                    // case 2
            x = x.p                                          // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                         // case 3
                w.color = RED                                // case 3
                RIGHT-ROTATE(T, w)                           // case 3
                w = x.p.right                                // case 3
            w.color = x.p.color                              // case 4
            x.p.color = BLACK                                // case 4
            w.right.color = BLACK                            // case 4
            LEFT-ROTATE(T, x.p)                              // case 4
            x = T.root                                       // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Case 1:
x's sibling w is red

Case 2:
x's sibling w is black, and both of w's children are black

Case 3:
x's sibling w is black, w's left child is red, and w's right child is black

Case 4:
x's sibling w is black, and w's right child is red

TC-13.4-1

Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                   // case 1
            x.p.color = RED                                   // case 1
            LEFT-ROTATE(T, x.p)                               // case 1
            w = x.p.right                                     // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                     // case 2
            x = x.p                                           // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                          // case 3
                w.color = RED                                 // case 3
                RIGHT-ROTATE(T, w)                            // case 3
                w = x.p.right                                 // case 3
            w.color = x.p.color                               // case 4
            x.p.color = BLACK                                 // case 4
            w.right.color = BLACK                             // case 4
            LEFT-ROTATE(T, x.p)                               // case 4
            x = T.root                                        // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Case 1:
x's sibling w is red

case 1 is converted into case 2,3, or 4.

Case 2:
x's sibling w is black, and both of w's children are black

Case 3:
x's sibling w is black, w's left child is red, and w's right child is black

Case 4:
x's sibling w is black, and w's right child is red

TC-13.4-1

Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                  // case 1
            x.p.color = RED                                  // case 1
            LEFT-ROTATE(T, x.p)                              // case 1
            w = x.p.right                                    // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                    // case 2
            x = x.p                                          // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                         // case 3
                w.color = RED                                // case 3
                RIGHT-ROTATE(T, w)                           // case 3
                w = x.p.right                                // case 3
            w.color = x.p.color                              // case 4
            x.p.color = BLACK                                // case 4
            w.right.color = BLACK                            // case 4
            LEFT-ROTATE(T, x.p)                              // case 4
            x = T.root                                       // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Case 1:
x's sibling w is red

case 1 is converted into case 2,3, or 4.

Case 2:
x's sibling w is black, and both of w's children are black

if terminates, the root of the subtree (the new x) is set to black.

Case 3:
x's sibling w is black, w's left child is red, and w's right child is black

Case 4:
x's sibling w is black, and w's right child is red

TC-13.4-1

Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                   // case 1
            x.p.color = RED                                   // case 1
            LEFT-ROTATE(T, x.p)                               // case 1
            w = x.p.right                                     // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                     // case 2
            x = x.p                                           // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                          // case 3
                w.color = RED                                 // case 3
                RIGHT-ROTATE(T, w)                            // case 3
                w = x.p.right                                 // case 3
            w.color = x.p.color                               // case 4
            x.p.color = BLACK                                 // case 4
            w.right.color = BLACK                             // case 4
            LEFT-ROTATE(T, x.p)                               // case 4
            x = T.root                                        // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Case 1:
x's sibling w is red

case 1 is converted into case 2,3, or 4.

Case 2:
x's sibling w is black, and both of w's children are black

if terminates, the root of the subtree (the new x) is set to black.

Case 3:
x's sibling w is black, w's left child is red, and w's right child is black

transform to case 4.

Case 4:
x's sibling w is black, and w's right child is red

TC-13.4-1

Argue that after executing RB-DELETE-FIXUP, the root of the tree must be black.

```
RB-DELETE-FIXUP(T, x)
while x ≠ T.root and x.color == BLACK
    if x == x.p.left
        w = x.p.right
        if w.color == RED
            w.color = BLACK                                  // case 1
            x.p.color = RED                                  // case 1
            LEFT-ROTATE(T, x.p)                              // case 1
            w = x.p.right                                    // case 1
        if w.left.color == BLACK and w.right.color == BLACK
            w.color = RED                                    // case 2
            x = x.p                                          // case 2
        else if w.right.color == BLACK
                w.left.color = BLACK                         // case 3
                w.color = RED                                // case 3
                RIGHT-ROTATE(T, w)                           // case 3
                w = x.p.right                                // case 3
            w.color = x.p.color                              // case 4
            x.p.color = BLACK                                // case 4
            w.right.color = BLACK                            // case 4
            LEFT-ROTATE(T, x.p)                              // case 4
            x = T.root                                       // case 4
    else (same as then clause with "right" and "left" exchanged)
x.color = BLACK
```

Case 1:
x's sibling w is red

case 1 is converted into case 2,3, or 4.

Case 2:
x's sibling w is black, and both of w's children are black

if terminates, the root of the subtree (the new x) is set to black.

Case 3:
x's sibling w is black, w's left child is red, and w's right child is black

transform to case 4.

Case 4:
x's sibling w is black, and w's right child is red

the root (the new x) is set to black.

TC-13.4-7

Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

TC-13.4-7

Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

Answer.

- NO!

TC-13.4-7

Suppose that a node x is inserted into a red-black tree with RB-INSERT and then is immediately deleted with RB-DELETE. Is the resulting red-black tree the same as the initial red-black tree? Justify your answer.

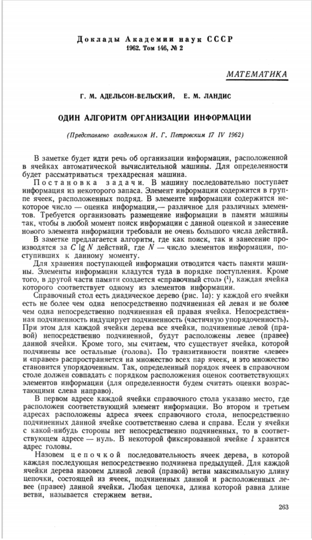
Answer.

- NO!

RedBlack Tree Visualization

TC Problem 13-3 (AVL Tree)

- An algorithm for the organization of information (1962).
- Named after 2 Russian mathematicians:
  - Georgii **A**delson-**V**elsky
  - Evgenii Mikhailovich **L**andis



Доклады Академии наук СССР
1962. Том 146, № 2

МАТЕМАТИКА

Г. М. АДЕЛЬСОН-ВЕЛЬСКИЙ, Е. М. ЛАНДИС

ОДИН АЛГОРИТМ ОРГАНИЗАЦИИ ИНФОРМАЦИИ

(Представлено академиком И. Г. Петровским 17 IV 1962)

В заметке будет идти речь об организации информации, расположенной в ячейках автоматической вычислительной машины. Для определенности будет рассматриваться трехадресная машина.

П о с т а н о в к а з а д а ч и. В машину последовательно поступает информация из некоторого запаса. Элемент информации содержится в группе ячеек, расположенных подряд. В элементе информации содержится некоторое число — оценка информации,— различное для различных элементов. Требуется организовать размещение информации в памяти машины так, чтобы в любой момент поиск информации с данной оценкой и занесение нового элемента информации требовали не очень большого числа действий.

В заметке предлагается алгоритм, где как поиск, так и занесение производятся за $C \lg N$ действий, где N — число элементов информации, поступивших к данному моменту.

Для хранения поступающей информации отводится часть памяти машины. Элементы информации кладутся туда в порядке поступления. Кроме того, в другой части памяти создается «справочный стол» [1], каждая ячейка которого соответствует одному из элементов информации.

Справочный стол есть диадическое дерево (рис. 1*a*): у каждой его ячейки есть не более чем одна непосредственно подчиненная ей левая и не более чем одна непосредственно подчиненная ей правая ячейка. Непосредственная подчиненность индуцирует подчиненность (частичную упорядоченность). При этом для каждой ячейки дерева все ячейки, подчиненные левой (правой) непосредственно подчиненной, будут расположены левее (правее) данной ячейки. Кроме того, мы считаем, что существует ячейка, которой подчинены все остальные (голова). По транзитивности понятие «левее» и «правее» распространяется на множество всех пар ячеек, и это множество становится упорядоченным. Так, определенный порядок ячеек в справочном столе должен совпадать с порядком расположения оценок соответствующих элементов информации (для определенности будем считать оценки возрастающими слева направо).

В первом адресе каждой ячейки справочного стола указано место, где расположен соответствующий элемент информации. Во втором и третьем адресах расположены адреса ячеек справочного стола, непосредственно подчиненных данной ячейке соответственно слева и справа. Если у ячейки с какой-нибудь стороны нет непосредственно подчиненных, то в соответствующем адресе — нуль. В некоторой фиксированной ячейке l хранится адрес головы.

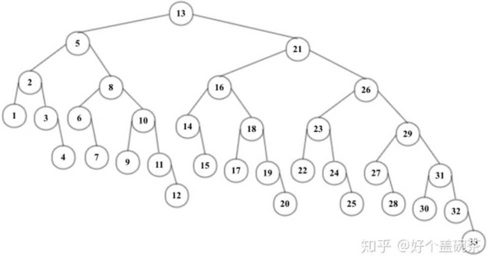
Назовем ц е п о ч к о й последовательность ячеек дерева, в которой каждая последующая непосредственно подчинена предыдущей. Для каждой ячейки дерева назовем длиной левой (правой) ветви максимальную длину цепочки, состоящей из ячеек, подчиненных данной и расположенных левее (правее) данной ячейки. Любая цепочка, длина которой равна длине ветви, называется стержнем ветви.

http://www.mathnet.ru/links/e8bbcf127a4f573d00aa42a7e5cdaaba/dan26964.pdf

TC Problem 13-3 (AVL Tree)

An AVL tree is a binary search tree that is **height balanced**: for each node x, the heights of the left and right subtrees of x differ by at most 1.

- To implement an AVL tree, we maintain an extra attribute in each node: $x.h$ is the height of node x.
- As for any other binary search tree T , we assume that $T.root$ points to the root node.



TC Problem 13-3 (a) I

(a) Prove that an AVL tree with n nodes has height $O(\lg n)$. (Hint: Prove that an AVL tree of height h has at least F_h nodes, where F_h is the h-th Fibonacci number.)

TC Problem 13-3 (a) II

Proof.

$p(h)$: an AVL tree of height h has at least F_h nodes.

- **(B)** $p(1)$ is obvious true.
- **(H)** Assume $p(k)$ is true for all $k < h$
- **(I)** Let r be the root, $r.left$ and $r.right$ be the left and right subtrees of r accordingly. $|r|$: number of nodes in r.
  - Assume $h - 2 \leq r.left.h \leq r.right.h = h - 1$, then $|r.left| \geq F_{h-2}$ and $|r.right| \geq F_{h-1}$
  - So,
    $n = |r| = |r.left| + |r.right| + 1 \geq F_{h-2} + F_{h-1} + 1 \geq F_h = \lfloor \frac{\phi^h}{2} + \frac{1}{2} \rfloor$
  - $h = O(\lg n)$

□

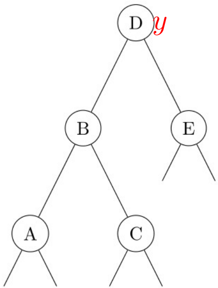
TC Problem 13-3 (b) I

(b) To insert into an AVL tree, we first place a node into the appropriate place in binary search tree order. Afterward, the tree might no longer be height balanced. Specifically, the heights of the left and right children of some node might differ by 2. Describe a procedure BALANCE(x), which takes a subtree rooted at x whose left and right children are height balanced and have heights that differ by at most 2, i.e., $|x.right.h - x.left.h| \leq 2$, and alters the subtree rooted at x to be height balanced. (Hint: Use rotations.)

TC Problem 13-3 (b) II

Answer

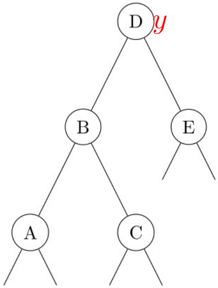
- Given a node x we define the balancing factor of x as $bf(x) = r.left.h - r.right.h$.
- After insertion, the height balance property (i.e., $|bf(x)| = |r.left.h - r.right.h| \leq 1$) might be broken.
- We have to maintain the property along the path from the inserted node to root.
- After insertion, $|bf(x)| = |r.left.h - r.right.h| \leq 2$
- Assuming $x.left.h \geq r.right.h = 2$



$B.h - E.h = 2$

TC Problem 13-3 (b) III

- Two subcases based on the difference between $A.h$ and $C.h$:
  - Case 1: $A.h \geq C.h$
  - case 2: $A.h < C.h$



$$B.h - E.h = 2$$

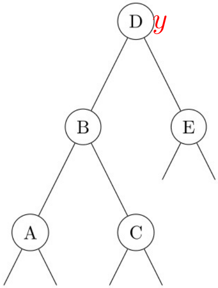
TC Problem 13-3 (b) IV

Case 1: $A.h \geq C.h$

- Assume $E.h = y$, then we have

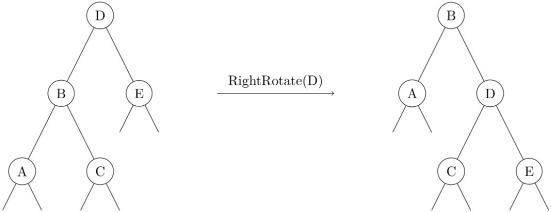
$$\begin{cases} B.h = y + 2 \\ A.h = y + 1 \\ y \leq C.h \leq y + 1 \end{cases}$$

- Right-Rotate at D



$$B.h - E.h = 2$$

TC Problem 13-3 (b) V



- $A.h, C.h, E.h$ keep unchanged, and

$$\begin{cases} 0 \le C.h - E.h \le 1 \\ y + 1 \le D.h = \max(C.h, E.h) + 1 = C.h + 1 \le y + 2 \\ 0 \le D.h - A.h \le 1 \end{cases}$$

- So, $|bf(B)| \le 1$ and $|bf(D)| \le 1$

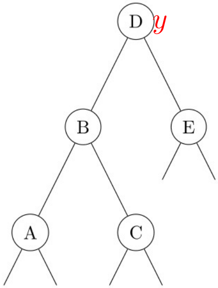
TC Problem 13-3 (b) VI

Case 2: $A.h < C.h$

- Assume $E.h = y$, then we have

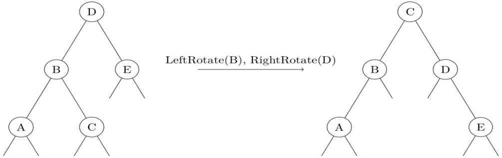
$$\begin{cases} B.h = y + 2 \\ C.h = y + 1 \\ A.h = y \end{cases}$$

- Left-Rotate at B
- Rigth-Rotate at D



$$B.h - E.h = 2$$

TC Problem 13-3 (b) VII



- $A.h, E.h$ keep unchanged, and

$$
\begin{cases}
y - 1 \le B.right.h, D.left.h \le y \\
0 \le A.h - B.right.h \le 1 \\
0 \le E.h - D.left.h \le 1 \\
B.h = \max(A.h, B.right.h) + 1 = y + 1 \\
D.h = \max(E.h, D.left.h) + 1 = y + 1 \\
B.h = D.h
\end{cases}
$$

- So, $|bf(B)| \le 1$, $|bf(C)| \le 1$ and $|bf(D)| \le 1$

TC Problem 13-3 (b) VIII

Proof.

BALANCE(x)

```
procedure BALANCE(x)
    if |bf(x)| = 2 then
        if bf(x) > 0 then
            if x.left.left.h ≥ x.left.right.h then
                RIGHT-ROTATE(x)
            else
                LEFT-ROTATE(x.left)
                RIGHT-ROTATE(x)
        else
            if x.right.right.h ≥ x.right.left.h then
                LEFT-ROTATE(x)
            else
                RIGHT-ROTATE(x.right)
                LEFT-ROTATE(x)
```

TC Problem 13-3 (c) I

(c) Using part (b), describe a recursive procedure AVL-INSERT(x, z) that takes a node x within an AVL tree and a newly created node z (whose key has already been filled in), and adds z to the subtree rooted at x, maintaining the property that x is the root of an AVL tree.
Assume that $z.key$ has already been filled in and that $z.left$= NIL and $z.right$=NIL; also assume that $z.h = 0$. Thus, to insert the node z into the AVL tree T , we call AVL-INSERT$(T.root, z)$.

TC Problem 13-3 (c) II

AVL-INSERT(x, z)

```
procedure AVL-INSERT(x,z)
    if x.key > z.key then
        if x.left ≠Nil then
            AVL-INSERT(x.left,z)
        else
            x.left ← z
    else if x.key > z.key then
        if x.right ≠Nil then
            AVL-INSERT(x.right,z)
        else
            x.right ← z
    BALANCE(x)
    x.h ← max (x.left.h, x.right.h) + 1
```

TC Problem 13-3 (d) I

(d) Show that AVL-INSERT, run on an n-node AVL tree, takes $O(\lg n)$ time and performs $O(1)$ rotations.

TC Problem 13-3 (d) II

```
procedure AVL-INSERT(x,z)
    if x.key > z.key then
        if x.left ≠Nil then
            AVL-INSERT(x.left,z)
        else
            x.left ← z
    else if x.key > z.key then
        if x.right ≠Nil then
            AVL-INSERT(x.right,z)
        else
            x.right ← z
    BALANCE(x)
    x.h ← max (x.left.h, x.right.h) + 1
```

- AVL-INSERT is called recursively at most $h = \lg n$ times;
- Only one call to BALANCE actually involves rotation.

TC Problem 13-3 (?)

(1) How to implement AVL-DELETE with BALANCE?
(2) What is the time complexity of AVL-DELETE.


Thank
You!