Advanced data structures

Lecture 1

Introduction. Binary search trees and red-black trees. Efficiency issues

Mircea Marin
mircea.marin@e-uvt.ro

Organizatorial items

- Lecturer and Teaching Assistant: Mircea Marin
  - email: `mircea.marin@e-uvt.ro`
- Course objectives:
  1. Become familiar with some of the advanced data structures (ADS) and algorithms which underpin much of today's computer programming
  2. Recognize the data structure and algorithms that are best suited to model and solve your problem
  3. Be able to perform a qualitative analysis of your algorithms – time complexity analysis
- Course webpage: http://web.info.uvt.ro/˜mmarin/lectures/ADS
- Handouts: will be posted on the webpage of the lecture
- Grading: 40% final exam (written), 60% labwork (mini-projects)
- Attendance: required

Organizatorial items

- Lab work: implementation in C++ of applications, using data structures and algorithms presented in this lecture
- Requirements:
  - ▷ Be up-to-date with the presented material
  - ▷ Prepare the programming assignments for the stated deadlines
- Prerequisites:
  1. A good understanding of the data structures and related algorithms, taught in the lecture DATA STRUCTURES.
  2. Familiarity with the C++ programming language.
  3. Recommended IDEs: Eclipse or Code::Blocks
- Recommended textbook:
  - Cormen, Leiserson, Rivest. *Introduction to Algorithms*. MIT Press.

A dynamic set is a collection of objects that may grow, shrink, or otherwise change over time.

- Objects have fields.
- A key is a field that uniquely identifies an object.
    - In all implementations of dynamic sets presented in this lecture, keys are assumed to be totally ordered.
- Operations on a dynamic set are grouped into two categories:
    1. **Queries:** they simply return information about the set. Typical examples: SEARCH, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR
    2. **Modifying operations:** INSERT, DELETE

Operations on a dynamic set S

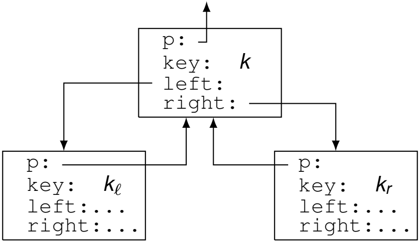
1. search(S, k): returns a pointer p to an element in S such that $p \rightarrow key = k$, or Nil if no such element exists.
2. minimum(S): returns a pointer to the element of S whose key is minimum, or Nil is S has no elements.
3. maximum(S): returns a pointer to the element of S whose key is maximum, or Nil is S has no elements.
4. successor(S, x): returns the next element larger than x in S, or Nil if x is the maximum element.
5. predecessor(S, x): returns the next element smaller than x in S, or Nil if x is the minimum element.
6. insert(S, p): augment S with the element pointed to by p. If p is Nil, do nothing.
7. del(S, p): remove the element pointed to by p from S. If p is Nil, do nothing.

Operations 1-5 are queries; operations 6-7 are modifying operations.

- The complexity of operations of a dynamic set is measured in terms of its size n = number of elements.
- Different implementations of dynamic sets vary by the runtime complexity of their operations
  - The choice of an implementation depends on the operations we perform most often.
  - Typical examples:
    - Binary search trees
    - Red-black trees
    - B-trees
    - Binomial heaps
    - Fibonacci heaps
    - ...

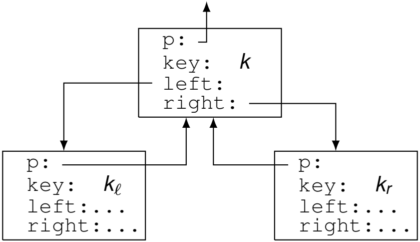
- Relevant information is stored in the nodes of a binary tree
  - every non-leaf node has a left and a right child.
- Every node has a unique key, that is used to identify a node:
- The finding of a node with a particular key must be fast $\Rightarrow$ the keys are distributed in a special way:
  - For every node, its key is larger then the keys of the nodes to its left, and smaller than the keys of the nodes to its right.

```
struct Node {
    int key;       // key
    Node *p;       // pointer to parent
    Node *left;    // pointer to left child
    Node *right;   // pointer to right child
    ...            // satellite data
};
struct BSTree {
   Node *root;     // pointer to node at root position
   ...             // operations on trees
};
```



Leaf nodes are empty: they contain no data

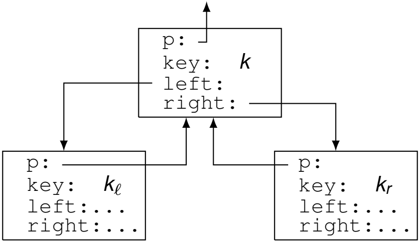
```
struct Node {
    int key;       // key
    Node *p;       // pointer to parent
    Node *left;    // pointer to left child
    Node *right;   // pointer to right child
    ...            // satellite data
};
struct BSTree {
   Node *root;     // pointer to node at root position
   ...             // operations on trees
};
```



Leaf nodes are empty: they contain no data

Requirement: $k_\ell \leq k \leq k_r$

```
struct Node {
    int key;       // key
    Node *p;       // pointer to parent
    Node *left;    // pointer to left child
    Node *right;   // pointer to right child
    ...            // satellite data
};
struct BSTree {
   Node *root;     // pointer to node at root position
   ...             // operations on trees
};
```



Leaf nodes are empty: they contain no data

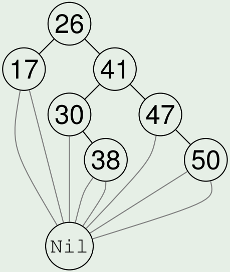
Requirement: $k_\ell \leq k \leq k_r$

If n is the root node then $n.\mathrm{p} == 0$.

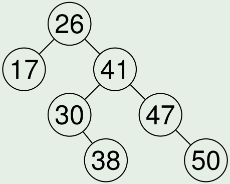
In OOP (e.g., C++), there are two ways to represent leaf nodes:

1. with the null pointer
2. with a single sentinel node NIL, which is also the root's parent.

Example (Diagrammatic representations)



or



ASSUMPTION: S is an instance of `BSTree`, which represents a binary search tree with n nodes.

operation	average case	worst case
S.`search`(k)	$O(\log_2 n)$	$O(n)$
S.`minimum`$()$	$O(\log_2 n)$	$O(n)$
S.`maximum`$()$	$O(\log_2 n)$	$O(n)$
S.`successor`(x)	$O(\log_2 n)$	$O(n)$
S.`predecessor`(x)	$O(\log_2 n)$	$O(n)$
S.`insert`(x)	$O(\log_2 n)$	$O(n)$
S.`del`(x)	$O(\log_2 n)$	$O(n)$

- Average case time complexity is $O(\log n)$ for all operations.
- Worst case time complexity is $O(n)$ for all operations.

Question: Can we improve the worst case time complexity of binary search trees without many changes?

Answer: Yes; Red-black trees

Binary Search trees versus Red-Black trees

Node structures

Binary search tree fields	Red-black tree fields
`key`	`key`
`left`,`right`: pointers to children	`left`,`right`: pointers to children
`p`: pointer to parent	`p`: pointer to parent
	`color`: RED or BLACK

- Similarities: both kinds of trees satisfy the binary search tree property:
  - if x is a node with left child y then $y.\mathtt{key} \leq x.\mathtt{key}$
  - if x is a node with right child y then $x.\mathtt{key} \leq y.\mathtt{key}$
- Properties specific to red-black trees (the **red-black properties**)
  1. Every node is either red or black
  2. The root is black
  3. Every leaf NIL is black.
  4. The children of a red node are black.
  5. Every path from a node to a descendant leaf contains the same number of black nodes.

Leaf nodes

- For binary-search tress, no memory needs to be allocated for leaf nodes. All pointers to a leaf node are assumed to be `Nil`, which is 0.
- For Red-black trees, it is convenient to represent all leaves with a black sentinel node NIL:
  - The left child, right child, and parent of NIL are NIL.
  - Also, the parent of the root node of a red-black tree is assumed to be NIL.

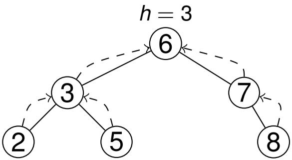
NOTIONS:

Height $h(n)$ of a node n = number of edges in a longest path from n to a leaf.

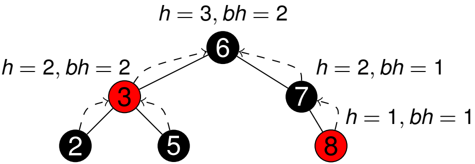
Black height $bh(n)$ of a node n = number of black nodes (including NIL) on the path from n to a leaf, not counting n.

DERIVED PROPERTIES:

1. $bh(n) \geq h(n)/2$ for every node n.
2. The subtree rooted at any node x contains $\geq 2^{bh(x)} - 1$ internal nodes.
3. A red-black tree with n internal nodes has height $\leq 2\log_2(n+1)$.



(a) Binary search tree



(b) Red-Black tree

- Nodes are represented by circles with the key value written inside.
  - Nodes may contain more data, not shown in the graphical representation.
  - For Red-Black trees, the circles are colored with the node colour.
- Thick lines between nodes are the pointers from parent to children.
- Dashed arrows are pointers from node to parent. They are usually not drawn.

As a binary search tree:

```
class Node {
public:
   int key;
   Node* left;
   Node* right;
   Node* p;
   // constructors
   ...
}
```

```
class BSTree {
 public:
    Node* root; // pointer to root
    BSTree() { root = 0; }
    ...
}
```

As a red-black tree:

```
class RBNode {
public:
   int key;
   RBNode* left;
   RBNode* right;
   RBNode* p;
   enum Color {RED, BLACK};
   Color color;
   // constructors
   ...
}
```

```
class RBTree {
public:
   RBNode* root; // pointer to root
   RBTree() { root = 0; }
   ...
}
```

Operations on Red-black trees

ASSUMPTION: S is an instance of `RBTree`, which represents a red-black search tree with n nodes.

operation	average case	worst case
S.`search`(k)	$O(\log_2 n)$	$O(\log_2 n)$
S.`minimum`()	$O(\log_2 n)$	$O(\log_2 n)$
S.`maximum`()	$O(\log_2 n)$	$O(\log_2 n)$
S.`successor`(x)	$O(\log_2 n)$	$O(\log_2 n)$
S.`predecessor`(x)	$O(\log_2 n)$	$O(\log_2 n)$

- These are query operations, implemented the same way as for binary search trees.
- The modifying operations `insert` and `del` must be redesigned carefully, to guarantee that the newly produced tree has the red-black properties.

Insertion in binary search trees

T.`insert`(z) where

> T : binary search tree
> z : pointer to node with z`->key` $= v$,
> z`->left` $= z$`->right` $= Nil$.

Effect: T and z are modified such that z is inserted at the right position in T.

```
insert(Node* z) // method of class BSTree
y = Nil
x = root
while x ≠ Nil
    y = x
    if z->key < x->key
        x = x->left
    else x = x->right
z->p = y
if y == Nil
    root = z
else if (z->key < y->key)
    y->left = z
else y->right = z
```

Deletion from binary search trees

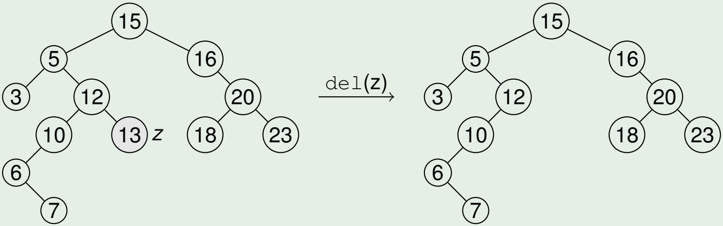
T.del(z) deletes node z from the binary search tree T

```
del(Node* z) // method of class BSTree
 if z->left == 0 or z->right == 0
   y := z
 else y = successor(z)
 if y->left ≠ 0
   x = y->left
 else x = y->right
 if x ≠ 0
   x->p = y->p
 if y->p == 0
   root = x
 else if y == y->p->left
         y->p->left = x
      else y->p->right = x
 if y ≠ z
      z->key = y->key
      if y has other fields, copy them to z too
 return y
```

Binary search trees

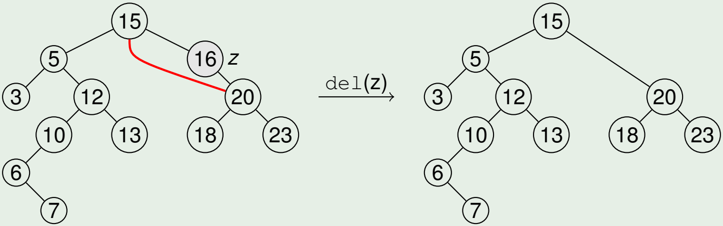
Case 1: Deletion of a node z without children

Example



- Node with key 13 has no children
  $\Rightarrow$ we simply remove it from the binary search tree.

Example

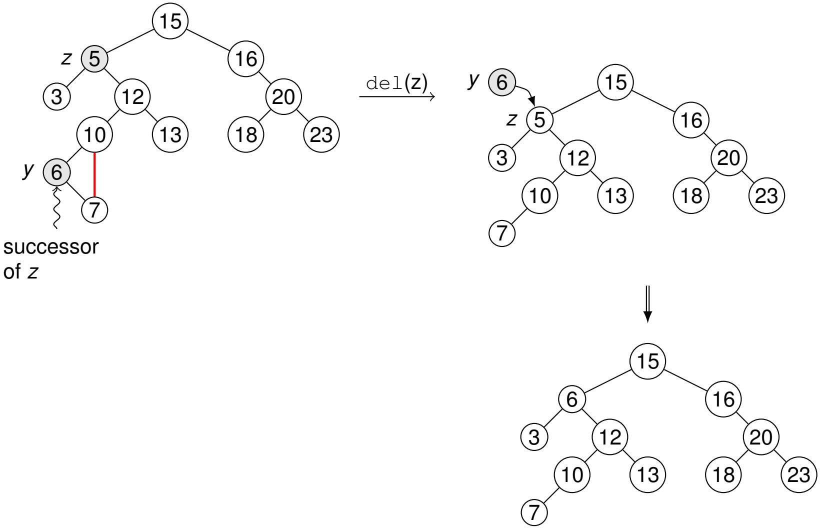


Node *z* with key 16 has only one child

$\Rightarrow$ the child of *z* becomes the child of the parent of *z*

Binary search trees

Case 2: Deletion of a node z with two children



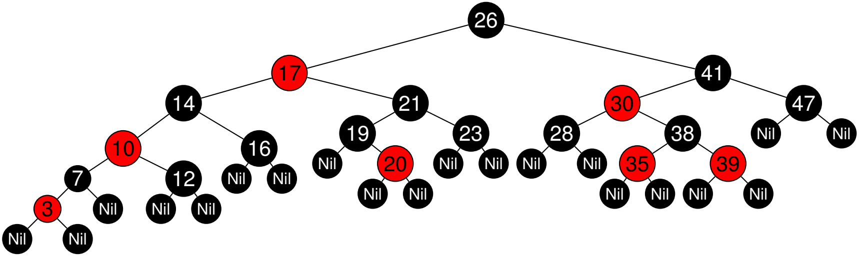


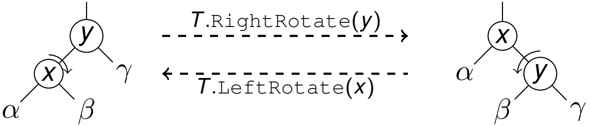
Figure: A balanced RB-tree with black-height 3

- The operations `insert` and `del` for binary search trees can be run on red-black trees with n keys
  - ⇒ they take $O(\log_2(n))$ time.
  - ⇒ they may destroy the red-black properties of the tree:

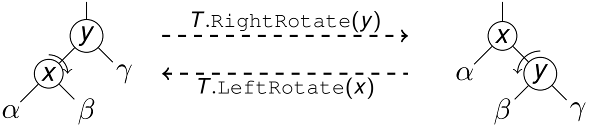
`insert`	of RED node might violate property 4; of BLACK node might violate property 5.
`del`	of RED node: no property violations; of BLACK node might violate properties 2, 4, 5.

⇒ **the red-black properties must be restored:**

- Some nodes must change color
- Some pointers must be changed



The rotation operations on a binary search tree.
T.RightRotate(x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers.
The configuration on the right can be transformed into the configuration on the left by the inverse operation T.LeftRotate(y).
The two nodes may appear anywhere in a binary search tree T. The letters α, β, and γ represent binary subtrees.



The rotation operations on a binary search tree.
T.RightRotate(x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers.
The configuration on the right can be transformed into the configuration on the left by the inverse operation T.LeftRotate(y).
The two nodes may appear anywhere in a binary search tree T. The letters α, β, and γ represent binary subtrees.

A rotation operation preserves the inorder ordering of keys: the keys in α precede x->key, which precedes the keys in β, which precede y->key, which precedes the keys in γ.

LeftRotate and RightRotate

Assumption: x->right $\neq$NIL.

```
LeftRotate(Node* x) // method of class BSTree
y = x->right
x->right = y->left
if y->left ≠ Nil
    y->left->p = x
y->p = x->p
if x->p == Nil
    root = y
else if x == x->p->left
    x->p->left = y
else x->p->right = y
y->left = x
x->p = y
```

- The code for RightRotate is similar.
- Both LeftRotate and RightRotate run in $O(1)$ time.

LEFTROTATE illustrated

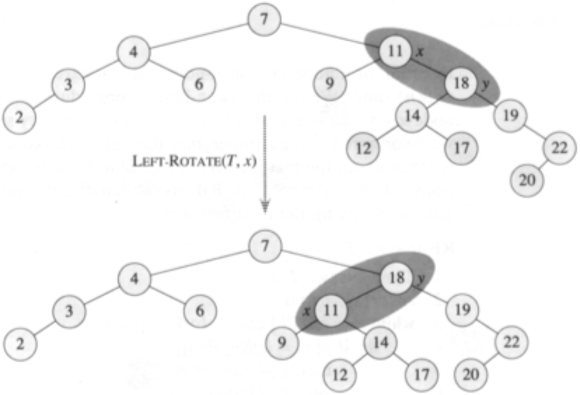


Figure 14.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. The NIL leaves are omitted. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

To insert a node x in a red-black tree T, proceed as follows:

- Perform insertion of x in T, as if T were a binary search tree.
- Color x to be RED.
- Adjust the color of the modified tree, by recoloring nodes and performing rotations.
- These ideas are implemented in the `RBInsert` procedure.

RB-trees

Insertion

```
RBInsert(Node∗ z) // method of class RBNode
y = NIL
x = root
while x ≠ Nil
        y = x
        if z->key < x->key
          then x = x->left
          else x = x->right
z->p = y
if y == NIL
          then root = z
          else if z->key < y->key
                  then y->left = z
                  else y->right = z
z->left = NIL
z->right = NIL
z->color = RED
RBInsertFixup(z)
```

Analysis of RBInsert

- RBInsert ends by coloring the new node z red.
- Then it calls RBInsertFixup because we could have violated some red-black properties:
  - The red-black properties 1, 3, and 5 are not violated.
  - Property 2 is violated if z is the root.
  - Property 4 is violated if z->p is RED, because both z and z->p are RED.

Analysis of `RBInsert`

- `RBInsert` ends by coloring the new node *z* red.
- Then it calls `RBInsertFixup` because we could have violated some red-black properties:
  - The red-black properties 1, 3, and 5 are not violated.
  - Property 2 is violated if *z* is the root.
  - Property 4 is violated if *z*`->p` is RED, because both *z* and *z*`->p` are RED.

  These violations are removed by calling `RBInsertFixup`.

RBInsertFixup

```
RBInsertFixup(Node* z) // method of class RBNode
while z->p->color == RED
    if z->p == z->p->p->left
    then y = z->p->p->right
        if y->color == RED
          then 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
                 then z = z->p               // Case 2
                      LeftRotate(z)          // Case 2
               z->p->color = BLACK           // Case 3
               z->p->p->color = RED          // Case 3
               RightRotate(z->p->p)          // Case 3
    else
        (same as then clause with right and left exchanged)
root->color = BLACK
```

RBInsertFixup

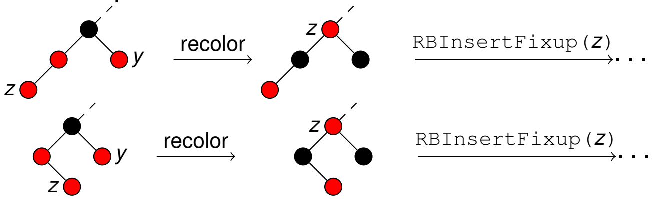
```
RBInsertFixup(Node* z) // method of class RBNode
while z->p->color == RED
      if z->p == z->p->p->left
      then y = z->p->p->right
            if y->color == RED
               then z->p->color = BLACK                   // Case 1
                    y->color = BLACK                      // Case 1
                    z->p->p->color = RED                  // Case 1
                    z = z->p->p                           // Case 1
               else if z == z->p->right
                       then z = z->p                      // Case 2
                            LeftRotate(z)                 // Case 2
                    z->p->color = BLACK                   // Case 3
                    z->p->p->color = RED                  // Case 3
                    RightRotate(z->p->p)                  // Case 3
      else
            (same as then clause with right and left exchanged)
root->color = BLACK
```

Remark: The following loop invariant holds at the start of each **while** loop:

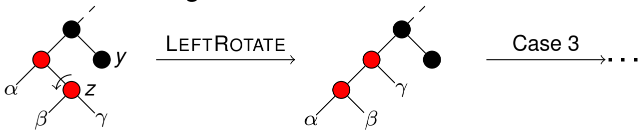
1. z is RED
2. There is at most one red-black violation: z is RED (property 2), or z and z->p are both RED

The new node z is RED, and inserted at the bottom of tree T. If the parent of z is red, we must fix T.

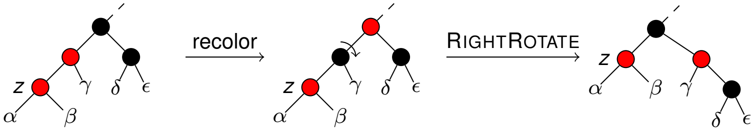
- Case 1: parent and uncle of z are RED



- Case 2: z is right child, and its uncle is BLACK



- Case 3: z is left child, and its uncle is BLACK



Deletion

- We will present a procedure $T.\texttt{RBDelete}(x)$ that performs deletion of node x from RB-tree T in $O(\log_2(n))$ time.
- $T.\texttt{RBDelete}(x)$ is a subtle adjustment of the deletion procedure for binary search trees. After splicing out a node, it calls an auxiliary procedure $\texttt{RBDeleteFixup}(x)$ that changes colors and performs rotations to restore the red-black properties.

RBDelete versus del

```
RBDelete(z)
if z->left == Nil or z->right == Nil
  y := z
else y = successor(z)
if y->left ≠ Nil
  x = y->left
else x = y->right
x->p = y->p
if y->p = Nil
  root = x
else if y == y->p->left
  y->p->left = x
else y->p->right = x
if y ≠ z
  z->key = y->key
  if y has other fields,
  copy them to z too
if y->color = BLACK
  RBDeleteFixup(x)
return y
```

```
DEL(z)
if z->left == Nil or z->right == Nil
  y := z
else y = successor(z)
if y->left ≠ Nil
  x = y->left
else x = y->right
if x ≠ Nil
  x->p = y->p
if y->p = Nil
  root = x
else if y == y->p->left
  y->p->left = x
else y->p->right = x
if y ≠ z
  z->key = y->key
  if y has other fields,
  copy them to z too
return y
```

T.RBDelete(z) returns node y which is spliced out from T.

- If y is red, the tree without y is red-black.
- Otherwise, y is black and the tree without y may violate the following red-black properties:
  - property 2: when y is the root, and y has only one child, which is red.
  - property 4: when x and y->p (which becomes x->p) are both red.
  - property 5: all paths that previously contained y have one fewer black nodes.

  To restore the red-black properties, we call RBDeleteFixup(x) where x is either
  - the sole child of y, before y was spliced out, or
  - the sentinel node *Nil*

  Note: In both cases, $x\text{->p} = y\text{->p}$

RBDeleteFixup(x)

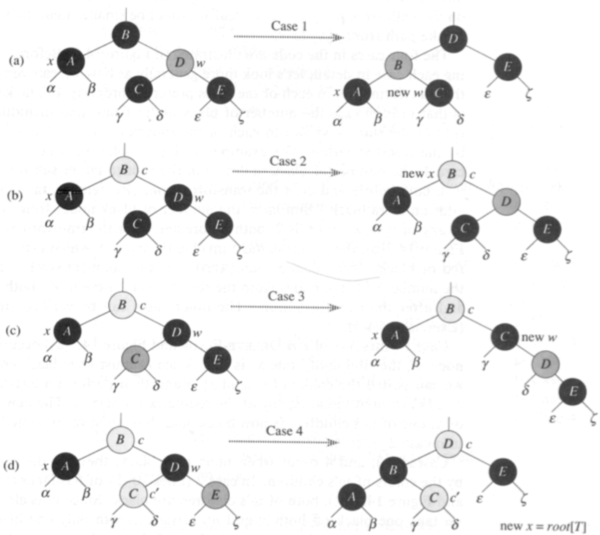
```
RBDeleteFixup(x)
while x ≠ root and x->color == BLACK
    if x == x->p->left
        w = x->p->right
        if w->color = RED
            w->color = BLACK                                   Case 1
            x->p->color = RED                                  Case 1
            LeftRotate(x->p)                                   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
                RightRotate(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
            LeftRotate(x->p)                                   Case 4
            x = root                                           Case 4
    else (same as then clause, with right and left exchanged)
x->color = BLACK
```

- Lines 1-22 are intended to move the extra black up the tree until either
  1. x points to a red node $\Rightarrow$ we will color the node black (line 23)
  2. x points to the root $\Rightarrow$ the extra black can be simply "removed"
  3. Suitable rotations and recolorings can be performed.

- ▷ Lines 1-22 are intended to move the extra black up the tree until either
  1. x points to a red node $\Rightarrow$ we will color the node black (line 23)
  2. x points to the root $\Rightarrow$ the extra black can be simply "removed"
  3. Suitable rotations and recolorings can be performed.
- ▷ In lines 1-22, x always points to a non-root black node that has the extra black, and w is set to point to the sibling of x. The **while** loop distinguishes 4 cases. See figure on next slide, where:
  - *Darkened nodes* are black, *heavily shaded nodes* are red, and *lightly shaded nodes* can be either red or black.
  - Small Greek letters represent arbitrary subtrees.
  - A node pointed to by x has an extra black.

RBDeleteFixup(x)

Implementation analysis



The four cases in the **while** loop:

- The only case that can cause the loop to repeat is case 2.

(a) Case 1 is transformed into case 2,3, or 4 by exchanging colors of nodes *B* and *D* and performing a left rotation.

(b) In case 2, the extra black represented by the pointer *x* is moved up the tree by coloring node *D* red and setting *x* to point to *B*. If we enter case 2 through case 1, the **while** loop terminates, since the color *c* is red.

(c) Case 3 is transformed to case 4 by exchanging the colors of nodes *C* and *D* and performing a right rotation.

(d) In case 4, the extra black represented by *x* can be removed by changing some colors and performing a left rotation.

- The height of the RB-tree is $O(\log_2(n)) \Rightarrow$ the total cost of the procedure without the call to `RBDeleteFixup` is $O(\log_2(n))$.
- Within the `RBDeleteFixup` call, cases 1, 3, 4 each terminate after a constant number of color changes and $\leq$ 3 rotations. Case 2 is the only case in which the **while** loop can be repeated, and then the pointer x moves upward at most $O(\log_2(n))$ times, and no rotations are performed. Thus
  - $\Rightarrow$ `RBDeleteFixup`(x) takes $O(\log_2(n))$ time and performs at most 3 rotations.

$\Rightarrow$ The overall time of `RBDelete`(x) is also $O(\log_2(n))$.

References

- Chapters 13 (Binary Search Trees), 14 (Red-Black trees), and Section 5.5 from the book
  - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, 2000.