LECTURE 3

Advanced data structures for dynamic sets

Red-black trees

Dynamic sets
Elements of a dynamic set

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 $p \to x = k$, 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. DELETE(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.

Dynamic sets
Implementation issues

- 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 time 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
    - ...

Binary search trees
Remember that ...

Binary search tree = implementation of dynamic set as a binary tree that supports well all its operations:

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

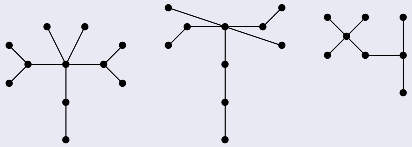
Question: Can we improve the worst case time complexity of binary search trees without many changes?

Answer: Red-black trees

- A free tree is a connected, acyclic, undirected graph. There is no fixed root node.

Examples

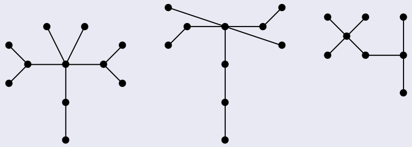
The following graphs are free trees:



- A free tree is a connected, acyclic, undirected graph. There is no fixed root node.

Examples

The following graphs are free trees:



A rooted tree is a free tree with a distinguished node called the root of the tree.

- Rooted trees are usually depicted with the root at top, and branches growing downward.
- A rooted tree can be ordered or unordered.

Trees
Basic notions

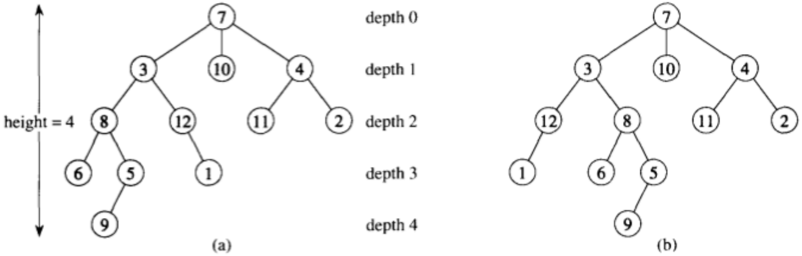


Figure 5.6 Rooted and ordered trees. **(a)** A rooted tree with height 4. The tree is drawn in a standard way: the root (node 7) is at the top, its children (nodes with depth 1) are beneath it, their children (nodes with depth 2) are beneath them, and so forth. If the tree is ordered, the relative left-to-right order of the children of a node matters; otherwise it doesn't. **(b)** Another rooted tree. As a rooted tree, it is identical to the tree in (a), but as an ordered tree it is different, since the children of node 3 appear in a different order.

- $deg(x)$:= number of children of a node x (a.k.a. degree of x).
- $depth(x)$:= length of path from root to node x.
- $height(T)$:= largest depth of any node in tree T.

A binary tree is a structure defined on a set of nodes. It is either

- the tree without nodes, also known as empty tree or null tree, and denoted by NIL,
- or, a structure consisting of 3 items:
    1. a root node,
    2. a binary tree, called its left subtree,
    3. a binary tree, called its right subtree.

Binary trees and full binary trees

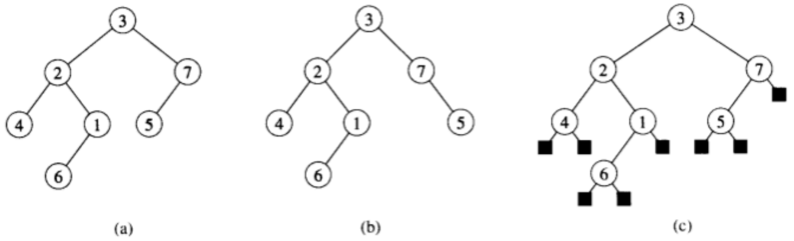


Figure 5.7 Binary trees. **(a)** A binary tree drawn in a standard way. The left child of a node is drawn beneath the node and to the left. The right child is drawn beneath and to the right. **(b)** A binary tree different from the one in (a). In (a), the left child of node 7 is 5 and the right child is absent. In (b), the left child of node 7 is absent and the right child is 5. As ordered trees, these trees are the same, but as binary trees, they are distinct. **(c)** The binary tree in (a) represented by the internal nodes of a full binary tree: an ordered tree in which each internal node has degree 2. The leaves in the tree are shown as squares.

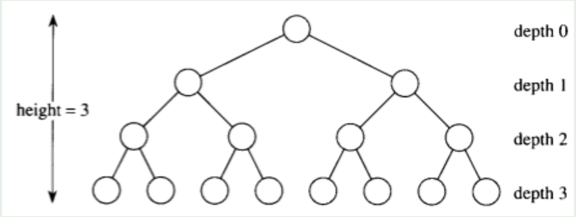
The position of a node in a binary tree can be clarified if we replace every missing child in the tree with a node without children (drawn as ■ in Fig. 5.7(c)). The tree that results is a full binary tree: each node is either leaf or has degree exactly 2. There are no degree-1 nodes.

k-ary and complete *k*-ary trees

- A *k*-ary tree is an ordered tree in which no node has more than k children. A binary tree is a 2-ary tree.
- A complete *k*-ary tree is a k-ary tree in which all leaves have the same depth and all internal nodes have degree k.

Example

A complete binary tree with 8 leaves and 7 internal nodes



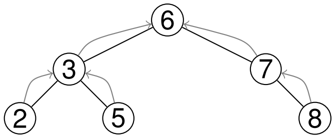
Binary Search trees and Red-Black trees
Similarities

- Both are advanced data structures that support the following operations on dynamic sets: SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT and DELETE.
- Both have similar node structure:

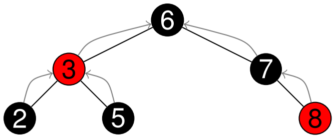
Node structure	
Binary search tree fields	Red-black tree fields
key	*key*
left, *right*: pointers to children	*left*, *right*: pointers to children
p: pointer to parent	*p*: pointer to parent
	color: RED or BLACK

Binary Search trees and Red-Black Trees

Graphical representation



(a) Binary 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.
- Gray arrows are pointers from node to parent. They are usually not drawn.

Main property: For every node *x*

- Keys in left subtree of *x* are smaller than *x*−>*key*
- Keys in right subtree of *x* are larger than *x*−>*key*

Tree representations of a dynamic set *S*

Alternatives in C++

As a binary search tree:

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

```
class BTree {
public:
   BNode* root; // pointer to root
   BTree() { 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; }
   ...
}
```

Binary Search trees and Red-Black Trees
Basic operations: SEARCH, MINIMUM, MAXIMUM

SEARCH(S, k)=SEARCH$(S.root, k)$, where

```
SEARCH(x, k)
1 if $x = Nil$ or $x{-}{>}key = k$
2     return $x$
3 if $k < x{-}{>}key$
4     return SEARCH($x{-}{>}left, k$)
5 else return SEARCH($x{-}{>}right, k$)
```

MINIMUM(S)=MINIMUM$(S.root)$, where

```
MINIMUM(x)
1 while $x{-}{>}left \neq Nil$
2         $x := x{-}{>}left$
3 return $x$
```

MAXIMUM(S)=MAXIMUM$(S.root)$, where

```
MAXIMUM(x)
1 while $x{-}{>}right \neq Nil$
2         $x := x{-}{>}right$
3 return $x$
```

Binary Search trees and Red-Black Trees
Basic operation: SUCCESSOR

Given a node x in a binary search tree S, SUCCESSOR(x) returns the node immediately after x in an inorder tree walk. This node is called the successor of x in the tree.

SUCCESSOR(S, x) = SUCCESSOR(x) where

```
SUCCESSOR($x$)
1 **if** $x->right \neq Nil$
2     **return** MINIMUM($x->right$)
3 $y := x->p$
4 **while** $y \neq Nil$ and $x = y->right$
5     $x := y$
6     $y := y->p$
7 **return** $y$
```

Binary Search trees and Red-Black Trees

Basic operation: PREDECESSOR

Given a node x in a binary search tree, PREDECESSOR(x) returns the node immediately before x in an in order tree walk. This node is called the predecessor of x in the tree.

PREDECESSOR(S, x) = PREDECESSOR(x)

```
PREDECESSOR($x$)
1 if $x->left \neq Nil$
2     return MAXIMUM($x->left$)
3 $y := x->p$
4 while $y \neq Nil$ and $x = y->left$
5     $x := y$
6     $y := y->p$
7 return $y$
```

Binary Search trees

Basic operation: INSERT

INSERT(T, z) where

T : binary search tree

z : 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($T, z$)
 1 $y = Nil$
 2 $x = T.root$
 3 while $x \neq Nil$
 4       $y := x$
 5       if $z{-}{>}key < x{-}{>}key$
 6          $x = x{-}{>}left$
 7       else $x = x{-}{>}right$
 8 $z{-}{>}p = y$
 9 if $y = Nil$
10       $T.root = z$
11 else if $(z{-}{>}key < y{-}{>}key)$
12       $y{-}{>}left = z$
13 else $y{-}{>}right = z$
```

Binary Search trees

Basic operation: DELETE

DELETE(T, z) deletes node z from the binary search tree T

```
DELETE(T, z)
 1 if $z->left = Nil$ or $z->right = Nil$
 2    $y := z$
 3 else $y =$ SUCCESSOR($z$)
 4 if $y->left \neq Nil$
 5    $x = y->left$
 6 else $x = y->right$
 7 if $x \neq Nil$
 8    $x->p = y->p$
 9    if $y->p = Nil$
10    $T.root = x$
11 else if $y = y->p->left$
12    $y->p->left = x$
13 else $y->p->right = z$
14 if $y \neq z$
15    $z->key = y->key$
16    if $y$ has other fields, copy them to $z$ too
17 return $y$
```

Binary Search trees and Red-Black trees
Runtime complexity

- All basic operations take time proportional to the height of the tree.
- If the tree is complete binary, the operations run in $\Theta(\log_2(n))$ worst-case time.
- The height of a randomly built binary search tree is $O(\log_2(n)) \Rightarrow$ basic operations take $\Theta(\log_2(n))$ average-case time.
- The binary search tree can degenerate into a tree of depth n
  - $\Rightarrow$ worst-case runtime complexity of Binary Search tree operations is $\Theta(n)$.
- Balanced red-black trees reduce the worst-case runtime complexity of all basic operations to $\Theta(\log_2(n))$.

Red-Black trees

- **Red-black tree** = binary search tree where nodes have also a color: RED or BLACK.
- **Balanced RB-tree** = RB-tree where no branch is more than twice as long as any other.
- The following **red-black properties** must be satisfied:
  1. Every node is either red or black
  2. Every leaf NIL is black.
  3. The children of a red node are black.
  4. Every simple path from a node to a descendant leaf contains the same number of black nodes.
     - The **black-height** $bh(x)$ of a node x is the number of black nodes on any path from, but not including, x to a leaf node.
     - The **black-height** of an RB-tree is the number of nodes on any simple path from root to a leaf.

Red-Black trees
Example

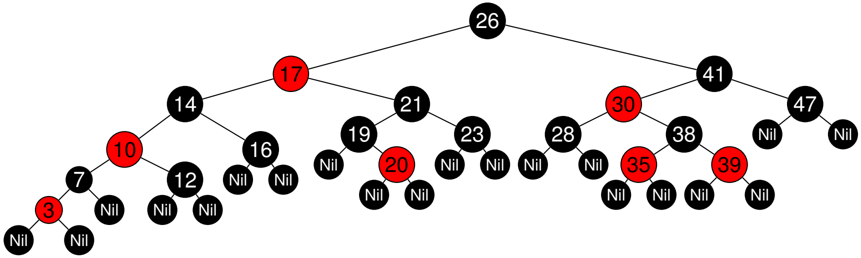


Figure : A balanced RB-tree with black-height 3

Balanced RB-trees
Important properties

- The number of red nodes on a branch of the tree is $\leq$ the number of black nodes.
  - $\Rightarrow$ All branches have length $\leq 2 \times N$ where N is the black height of the tree.
  - $\Rightarrow$ A red-black tree with n internal nodes has height at most $2\log_2(n+1)$.

Balanced RB-trees
Important properties

- The number of red nodes on a branch of the tree is $\leq$ the number of black nodes.
  - $\Rightarrow$ All branches have length $\leq 2 \times N$ where N is the black height of the tree.
  - $\Rightarrow$ A red-black tree with n internal nodes has height at most $2\log_2(n+1)$.

Corollary

The operations SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR on red-black trees do not modify the structure of the tree, and can be run on $O(\log_2(n))$ time.

Balanced RB-trees
Important properties

- The number of red nodes on a branch of the tree is $\leq$ the number of black nodes.
  - $\Rightarrow$ All branches have length $\leq 2 \times N$ where N is the black height of the tree.
  - $\Rightarrow$ A red-black tree with n internal nodes has height at most $2\log_2(n+1)$.

Corollary

The operations SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR on red-black trees do not modify the structure of the tree, and can be run on $O(\log_2(n))$ time.

- INSERT and DELETE should preserve the red-black properties of the balanced RB-tree.

Balanced RB-trees
Important properties

- The number of red nodes on a branch of the tree is $\leq$ the number of black nodes.
  - $\Rightarrow$ All branches have length $\leq 2 \times N$ where N is the black height of the tree.
  - $\Rightarrow$ A red-black tree with n internal nodes has height at most $2\log_2(n+1)$.

Corollary

The operations SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, and SUCCESSOR on red-black trees do not modify the structure of the tree, and can be run on $O(\log_2(n))$ time.

- INSERT and DELETE should preserve the red-black properties of the balanced RB-tree.
- We shall see how to perform INSERT and DELETE in $O(\log_2(n))$

Preliminary remarks

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

$\Rightarrow$ red-black properties must be restored:

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

LEFTROTATE and RIGHTROTATE

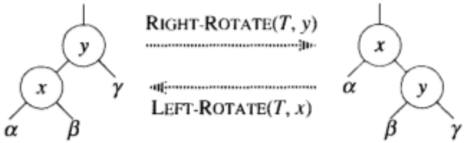


Figure 14.2 The rotation operations on a binary search tree. The operation RIGHT-ROTATE(T, x) transforms the configuration of the two nodes on the left into the configuration on the right by changing a constant number of pointers. The configuration on the right can be transformed into the configuration on the left by the inverse operation LEFT-ROTATE(T, y). The two nodes might occur anywhere in a binary search tree. The letters α, β, and γ represent arbitrary subtrees. A rotation operation preserves the inorder ordering of keys: the keys in α precede *key*$[x]$, which precedes the keys in β, which precede *key*$[y]$, which precedes the keys in γ.

LeftRotate and RightRotate

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

```
LEFTROTATE($T, x$)
1 $y := x$->$right$
2 $x$->$right := y$->$left$
3 if $y$->$left \neq$ NIL
4     $y$->$left$->$p := x$
5 $y$->$p := x$->$p$
6 if $x$->$p =$ NIL
7     $T.root := y$
8 else if $x = x$->$p$->$left$
9     $x$->$p$->$left := y$
10 else $x$->$p$->$right := y$
11 $y$->$left := x$
12 $x$->$p := y$
```

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

LeftRotate illustrated

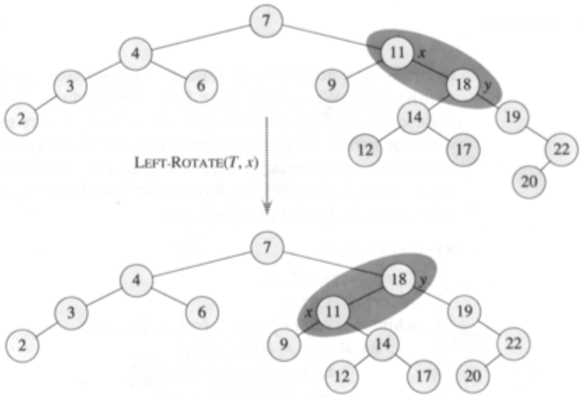


Figure 14.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. The NIL leaves are omitted. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

Insertion
Main idea

To insert a node x in a RB-tree T with n nodes, 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 `RB-Insert` procedure.

RB-trees
Insertion

```
RB-INSERT($T, x$)
Insert($T, x$)
$x{-}{>}color = $ RED
while $x \neq T.root$ and $x{-}{>}p{-}{>}color = $ RED
        if $x{-}{>}p = x{-}{>}p{-}{>}p{-}{>}left$
            $y = x{-}{>}p{-}{>}p{-}{>}right$
            if $y{-}{>}color = $ RED
                $x{-}{>}p{-}{>}color = $ BLACK          Case 1
                $y{-}{>}color = $ BLACK                 Case 1
                $x{-}{>}p{-}{>}p{-}{>}color = $ RED     Case 1
                $x = x{-}{>}p{-}{>}p$                   Case 1
            else if $x = x{-}{>}p{-}{>}right$
                $x = x{-}{>}p$                          Case 2
                LEFT-ROTATE($T, x$)                     Case 2
            $x{-}{>}p{-}{>}color = $ BLACK              Case 3
            $x{-}{>}p{-}{>}p{-}{>}color = $ RED         Case 3
            RIGHT-ROTATE($T, x{-}{>}p{-}{>}p$)          Case 3
        else (same as then clause with “right” and “left” exchanged)
$T.root{-}{>}color = $ BLACK
```

Implementation analysis (1)

1. What violations of the red-black properties do happen in lines 1-2?
2. What is the overall goal of lines 3-17 in the **while loop**?
3. Explore how the three cases of the **while** loop accomplish their goal.

Implementation analysis (1)

1. What violations of the red-black properties do happen in lines 1-2?
2. What is the overall goal of lines 3-17 in the **while loop**?
3. Explore how the three cases of the **while** loop accomplish their goal.

- ▷ Red-black property 1 holds after lines 1-2.

Implementation analysis (1)

1. What violations of the red-black properties do happen in lines 1-2?
2. What is the overall goal of lines 3-17 in the **while loop**?
3. Explore how the three cases of the **while** loop accomplish their goal.

- ▷ Red-black property 1 holds after lines 1-2.
- ▷ Red-black property 2 holds after lines 1-2, because the newly inserted new node has NIL's for children.

Implementation analysis (1)

1. What violations of the red-black properties do happen in lines 1-2?
2. What is the overall goal of lines 3-17 in the **while loop**?
3. Explore how the three cases of the **while** loop accomplish their goal.

- ▷ Red-black property 1 holds after lines 1-2.
- ▷ Red-black property 2 holds after lines 1-2, because the newly inserted new node has NIL's for children.
- ▷ Red-black property 4 holds after lines 1-2, because red node x replaces black node NIL, and x is with NIL children.

Implementation analysis (1)

1. What violations of the red-black properties do happen in lines 1-2?
2. What is the overall goal of lines 3-17 in the **while loop**?
3. Explore how the three cases of the **while** loop accomplish their goal.

- ▷ Red-black property 1 holds after lines 1-2.
- ▷ Red-black property 2 holds after lines 1-2, because the newly inserted new node has NIL's for children.
- ▷ Red-black property 4 holds after lines 1-2, because red node x replaces black node NIL, and x is with NIL children.

$\Rightarrow$ The only properly that might be violated after lines 1-2 is 3.

Implementation analysis (1)

1. What violations of the red-black properties do happen in lines 1-2?
2. What is the overall goal of lines 3-17 in the **while loop**?
3. Explore how the three cases of the **while** loop accomplish their goal.

- ▷ Red-black property 1 holds after lines 1-2.
- ▷ Red-black property 2 holds after lines 1-2, because the newly inserted new node has NIL's for children.
- ▷ Red-black property 4 holds after lines 1-2, because red node x replaces black node NIL, and x is with NIL children.

⇒ The only properly that might be violated after lines 1-2 is 3.

- ▷ This happens if the parent of x is red.

Implementation analysis (1)

1. What violations of the red-black properties do happen in lines 1-2?
2. What is the overall goal of lines 3-17 in the **while loop**?
3. Explore how the three cases of the **while** loop accomplish their goal.

- ▷ Red-black property 1 holds after lines 1-2.
- ▷ Red-black property 2 holds after lines 1-2, because the newly inserted new node has NIL's for children.
- ▷ Red-black property 4 holds after lines 1-2, because red node *x* replaces black node NIL, and *x* is with NIL children.

⇒ The only properly that might be violated after lines 1-2 is 3.

  - ▷ This happens if the parent of *x* is red.

- ▷ The goal of the **while** loop is to move this violation up the tree, while maintaining property 4.

Implementation analysis (1)

1. What violations of the red-black properties do happen in lines 1-2?
2. What is the overall goal of lines 3-17 in the **while loop**?
3. Explore how the three cases of the **while** loop accomplish their goal.

- ▷ Red-black property 1 holds after lines 1-2.
- ▷ Red-black property 2 holds after lines 1-2, because the newly inserted new node has NIL's for children.
- ▷ Red-black property 4 holds after lines 1-2, because red node *x* replaces black node NIL, and *x* is with NIL children.

⇒ The only properly that might be violated after lines 1-2 is 3.

  - ▷ This happens if the parent of *x* is red.

- ▷ The goal of the **while** loop is to move this violation up the tree, while maintaining property 4.
- ▷ At the beginning of each **while** loop, *x* points to a red node with a red parent. (This is the only violation of a red-black property in the tree!)

Implementation analysis (1)

1. What violations of the red-black properties do happen in lines 1-2?
2. What is the overall goal of lines 3-17 in the **while loop**?
3. Explore how the three cases of the **while** loop accomplish their goal.

- ▷ Red-black property 1 holds after lines 1-2.
- ▷ Red-black property 2 holds after lines 1-2, because the newly inserted new node has NIL's for children.
- ▷ Red-black property 4 holds after lines 1-2, because red node *x* replaces black node NIL, and *x* is with NIL children.

⇒ The only properly that might be violated after lines 1-2 is 3.

  - ▷ This happens if the parent of *x* is red.

- ▷ The goal of the **while** loop is to move this violation up the tree, while maintaining property 4.
- ▷ At the beginning of each **while** loop, *x* points to a red node with a red parent. (This is the only violation of a red-black property in the tree!)

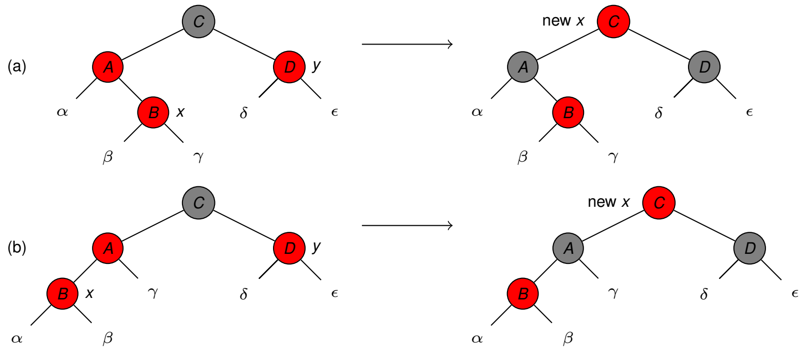
4. There are 2 possible outcomes of a loop iteration:
   1. The pointer *x* moves up the tree.
   2. Some rotations are performed and the loop terminates.

Implementation analysis (2)

- The **while** loop has 6 distinct cases to consider. Three of them are symmetric to the other three, depending on whether $x{-}{>}p$ is a left child or right child of $x{-}{>}p{-}{>}p$. (This situation is determined in line 4.) $\Rightarrow$ 3 cases to consider.
- Case 1 analyzes the color of $x{-}{>}p{-}{>}p{-}{>}right$. Line 5 makes y point to $x{-}{>}p{-}{>}p{-}{>}right$, and a test is made in line 6. If y is red, then case 1 is executed, Otherwise, one of cases 2 or 3 is executed. In all cases, $x{-}{>}p{-}{>}p$ is black (because the $x{-}{>}p$ is red), and property 3 is violated only between x and $x{-}{>}p$.
- In cases 2-3, the color of x's uncle y is black. The two cases are distinguished by whether x is a right or left child of $x{-}{>}p$. Lines 12-13 constitute case 2. In this case, we immediately perform a left-rotation to transform the situation into case 3 (lines 14-16), where x is a left child. Regardless of how we enter case 2, x's uncle y is black, since otherwise we would have executed case 1. We perform some color changes and a right-rotation, which preserve property 4, and stop executing another loop because $x{-}{>}p$ is now black.

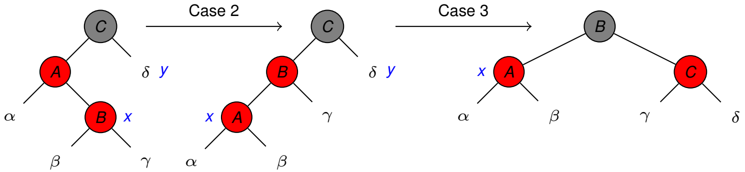
It can be argued easily that the running time of RB-INSERT(T, x) is $O(\log_2(n))$, where n is the number of nodes in the RB-tree T.

Illustration: Case 1 of RB-INSERT(T, x)



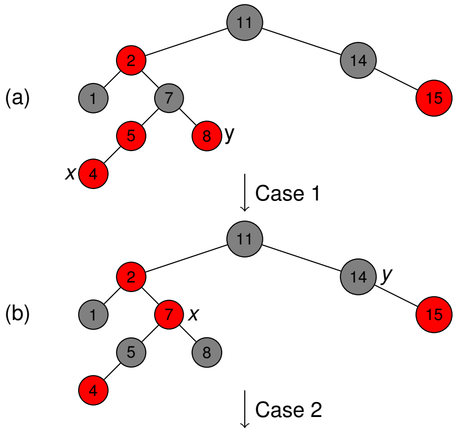
Property 3 is violated, since both x and its parent are red. The same action takes place whether (a) x is a right child or (b) x is a left child. Each of the subtrees α, β, γ, δ, and ϵ has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 4: all downward paths from a node to a leaf have the same number of blacks. The **while** loop continues with node x's grandparent $x->p->p$ as the new x. Any violations of property 3 can now occur only between the new x, which is red, and its parent, if it is red as well.

Illustration: Cases 2 and 3 of RB-INSERT(T, x)



As in case 1, property 3 is violated either in case 2 or case 3 because x and its parent x−>p are both red. Each of the subtrees α, β, γ, and δ has a black root, and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 4: all downward paths from a node to a leaf node have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 4. The **while** loop then terminates, because property 3 is satisfied.

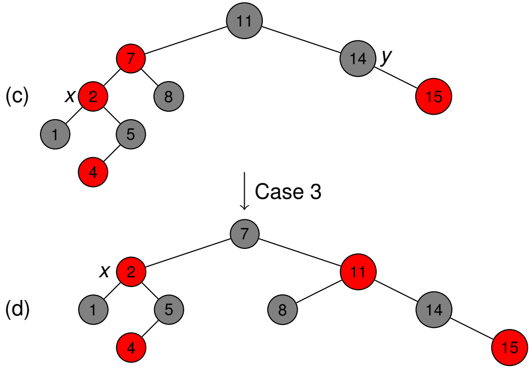
Illustration: RB-INSERT(T, x)



(a) A node x after insertion. Both x and $x{-}{>}p$ are red, therefore red-black property is violated. Since x's uncle y is red, we apply case 1: nodes are recolored and the pointer x is moved up the tree, resulting in the tree shown in (b).

(b) x and $x{-}{>}p$ are both red again, but x's uncle y is now black. Since x is a right child, case 2 can be applied by applying a left rotation, which yields the tree shown in (c). (See next slide).

Illustration: RB-INSERT(T, x)



(c) x is a left child $\Rightarrow$ execute case 3.
A right rotation yields the tree in (d), which is a legal red-black tree.

Deletion

- We will present a procedure RB-DELETE(T, x) that performs deletion of node x from RB-tree T in $O(\log_2(n))$ time.
- To simplify the implementation, we replace NIL with a sentinel NIL$[T]$ which is an ordinary BLACK node whose value of the other fields – p, *left*, *right*, and *key* — will be set to arbitrary values. Benefits of this change are:
  - There will be only one sentinel node NIL$[T]$.
  - When we wish to manipulate a child (possibly NIL$[T]$) of a node x, we can set NIL$[T].p$ to x, so that NIL$[T]$ assumes, during that manipulation, that x is its parent.
- RB-DELETE(T, x) is a subtle adjustment of the deletion procedure for binary search trees. After splicing out a node, it calls an auxiliary procedure RB-DELETE-FIXUP(T, x) that changes colors and performs rotations to restore the red-black properties.

RB-DELETE versus DELETE

```
RB-DELETE($T, z$)
 1 if $z\text{->}left = Nil$ or $z\text{->}right = Nil[T]$
 2   $y := z$
 3 else $y =$ SUCCESSOR($z$)
 4 if $y\text{->}left \neq Nil[T]$
 5   $x = y\text{->}left$
 6 else $x = y\text{->}right$
 7 $x\text{->}p = y\text{->}p$
 8 if $y\text{->}p = Nil[T]$
 9   $T.root = x$
10 else if $y = y\text{->}p\text{->}left$
11   $y\text{->}p\text{->}left = x$
12 else $y\text{->}p\text{->}right = z$
13 if $y \neq z$
14   $z\text{->}key = y\text{->}key$
15   if y has other fields,
     copy them to z too
16 if $y\text{->}color =$ BLACK
17   RB-DELETE-FIXUP($T, x$)
18 return $y$
```

```
DELETE($T, z$)
 1 if $z\text{->}left = Nil$ or $z\text{->}right = Nil$
 2   $y := z$
 3 else $y =$ SUCCESSOR($z$)
 4 if $y\text{->}left \neq Nil$
 5.  $x = y\text{->}left$
 6 else $x = y\text{->}right$
 7 if $x \neq Nil$
 8    $x\text{->}p = y\text{->}p$
 9 if $y\text{->}p = Nil$
10   $T.root = x$
11 else if $y = y\text{->}p\text{->}left$
12   $y\text{->}p\text{->}left = x$
13 else $y\text{->}p\text{->}right = z$
14 if $y \neq z$
15   $z\text{->}key = y\text{->}key$
16   if y has other fields,
     copy them to z too
17 return $y$
```

RB-DELETE versus TREEDELETE
Implementation analysis

- All references to NIL in DELETE have been replaced in RB-DELETE by references to the sentinel NIL$[T]$.
- The test if $x = $ NIL in line 7 disappeared; the assignment $x{-}{>}p := y{-}{>}p$ is performed unconditionally in line 7 of RB-DELETE:
  - If x is NIL$[T]$, it's parent pointer will point to the parent of the spliced-out node y.
- RB-DELETE-FIXUP(T, x) is called in lines 16-17 if y is black.
  - ▷ If y is red, the red-black properties are unaffected when y is spliced out, because no black heights in the tree have changed and no red nodes have been made adjacent.
  - ▷ The argument x passed to RB-DELETE-FIXUP is the sole child of y before y was spliced out if y had a non-NIL child, or the sentinel node NIL$[T]$ otherwise. In the latter case, line 7 guarantees that $p.x$ becomes the former parent of y, whether x is an ordinary node or the sentinel NIL$[T]$.

- If the spliced-out node y in RB-DELETE is black, its removal causes all paths that previously contained y to have one fewer black nodes $\Rightarrow$ red-black property 4 does not hold for the ancestors of y in the tree.

RB-DELETE
Implementation analysis

- ▷ If the spliced-out node y in RB-DELETE is black, its removal causes all paths that previously contained y to have one fewer black nodes $\Rightarrow$ red-black property 4 does not hold for the ancestors of y in the tree.

$\Rightarrow$ correction by: adding 1 to the count of all black nodes on any path that contains x, we obtain that

  - ▷ Property 4 holds.
  - ▷ When we splice out the black node y, we “push” its blackness onto its child.
  - ▷ Property 1 may get violated, when x may be “doubly black.”

RB-DELETE
Implementation analysis

- ▷ If the spliced-out node y in RB-DELETE is black, its removal causes all paths that previously contained y to have one fewer black nodes $\Rightarrow$ red-black property 4 does not hold for the ancestors of y in the tree.

$\Rightarrow$ correction by: adding 1 to the count of all black nodes on any path that contains x, we obtain that

  - ▷ Property 4 holds.
  - ▷ When we splice out the black node y, we “push” its blackness onto its child.
  - ▷ Property 1 may get violated, when x may be “doubly black.”

- ▷ Procedure call RB-DELETE-FIXUP(T, x) is intended to restore property 1.

RB-DELETE-FIXUP

```
RB-DELETE-FIXUP($T, x$)
 1 while $x \neq T.root$ and $x\text{->}color = $ BLACK
 2     if $x = x\text{->}p\text{->}left$
 3         $w = x\text{->}p\text{->}right$
 4         if $w\text{->}color = $ RED
 5             $w\text{->}color = $ BLACK                                   Case 1
 6             $x\text{->}p\text{->}color = $ RED                              Case 1
 7             LEFT-ROTATE($T, x\text{->}p$)                                Case 1
 8             $w = x\text{->}p\text{->}right$                                 Case 1
 9         if $w\text{->}left\text{->}color = $ BLACK and $w\text{->}right\text{->}color = $ BLACK
10             $w\text{->}color = $ RED                                     Case 2
11             $x = x\text{->}p$                                            Case 2
12         else if $w\text{->}right\text{->}color = $ BLACK
13             $w\text{->}left\text{->}color = $ BLACK                            Case 3
14             $w\text{->}color = $ RED                                     Case 3
15             RIGHT-ROTATE($T, w$)                                  Case 3
16             $w = x\text{->}p\text{->}right$                                 Case 3
17         $w\text{->}color = x\text{->}p\text{->}color$                            Case 4
18         $x\text{->}p\text{->}color = $ BLACK                                Case 4
19         $w\text{->}right\text{->}color = $ BLACK                            Case 4
20         LEFT-ROTATE($T, x\text{->}p$)                                    Case 4
21         $x = T.root$                                                 Case 4
22     else (same as then clause, with “right” and “left” exchanged)
23 $x\text{->}color = $ BLACK
```

RB-DELETE-FIXUP

Implementation analysis

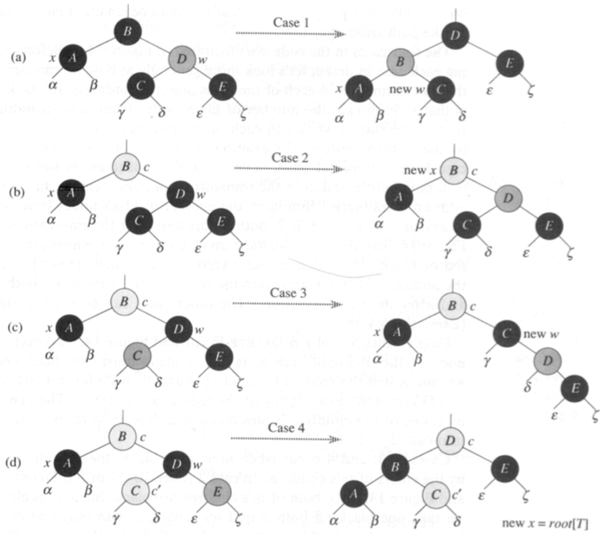
- Lines 1-22 are intended to move the extra black up the tree until either
    1. x points to a red node $\Rightarrow$ we will color the node black (line 23)
    2. x points to the root $\Rightarrow$ the extra black can be simply "removed"
    3. Suitable rotations and recolorings can be performed.

RB-DELETE-FIXUP
Implementation analysis

- 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.

RB-DELETE-FIXUP: Implementation Analysis

The four cases in the **while** loop



RB-DELETE-FIXUP: 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.

RB-DELETE
Time complexity analysis

- The height of the RB-tree is $O(\log_2(n)) \Rightarrow$ the total cost of the procedure without the call to RB-DELETE-FIXUP is $O(\log_2(n))$.
- Within the RB-DELETE-FIXUP call, cases 1, 3, 4 each terminate after a constant number of color changes and ≤ 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
  - $\Rightarrow$ RB-DELETE-FIXUP(T, x) takes $O(\log_2(n))$ time and performs at most 3 rotations.

$\Rightarrow$ The overall time of RB-DELETE(T, 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.