

Data Structures and Algorithms

Werner Nutt

Werner.Nutt@unibz.it

<http://www.inf.unibz.it/~nutt>

Chapter 6

Academic Year 2012-2013

Acknowledgements & Copyright Notice

These slides are built on top of slides developed by [Michael Boehlen](#). Moreover, some material (text, figures, examples) displayed in these slides is courtesy of **Kurt Ranalter**. Some examples displayed in these slides are taken from [**Cormen, Leiserson, Rivest and Stein**, "Introduction to Algorithms", MIT Press], and their copyright is detained by the authors. All the other material is copyrighted by **Roberto Sebastiani**. Every commercial use of this material is strictly forbidden by the copyright laws without the authorization of the authors. No copy of these slides can be displayed in public or be publicly distributed without containing this copyright notice.

Data Structures and Algorithms

Chapter 6

- Binary Search Trees
 - Tree traversals
 - Searching
 - Insertion
 - Deletion
- Red-Black Trees
 - Properties
 - Rotations
 - Insertion
 - Deletion

Data Structures and Algorithms

Chapter 6

- **Binary Search Trees**
 - Tree traversals
 - Searching
 - Insertion
 - Deletion
- **Red-Black Trees**
 - Properties
 - Rotations
 - Insertion
 - Deletion



Dictionaries

- A *dictionary* D is a dynamic data structure with operations:
 - **Search**(D, k) – *returns a pointer x to an element such that $x.key = k$ (null otherwise)*
 - **Insert**(D, x) – *adds the element pointed to by x to D*
 - **Delete**(D, x) – *removes the element pointed to by x from D*
- An element has a *key* and *data* part.

Ordered Dictionaries

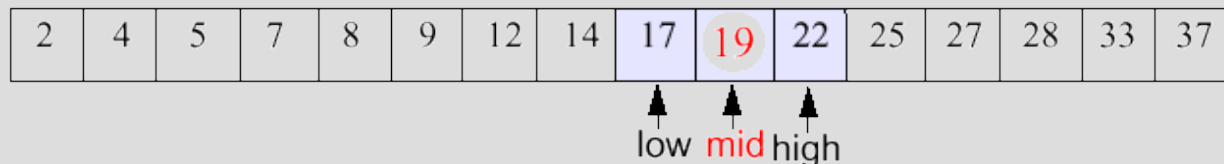
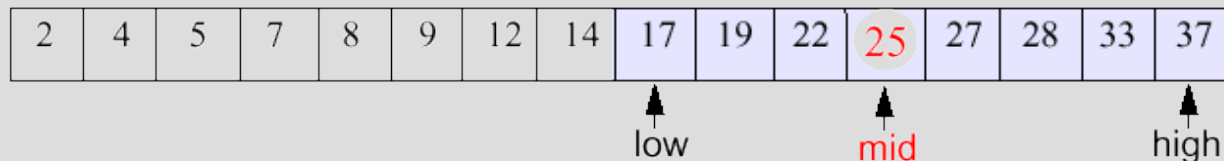
- In addition to dictionary functionality, we may want to support operations:
 - **Min(D)**
 - **Max(D)**
- and
 - **Predecessor(D, k)**
 - **Successor(D, k)**
- These operations require keys that are *comparable (ordered domain)*.

A List-Based Implementation

- Unordered list 
 - search, min, max, predecessor, successor: $O(n)$
 - insertion, deletion: $O(1)$
- Ordered list 
 - search, insertion: $O(n)$
 - min, max, predecessor, successor, deletion: $O(1)$

Refresher: Binary Search

- Narrow down the search range in stages
 - `findElement(22)`

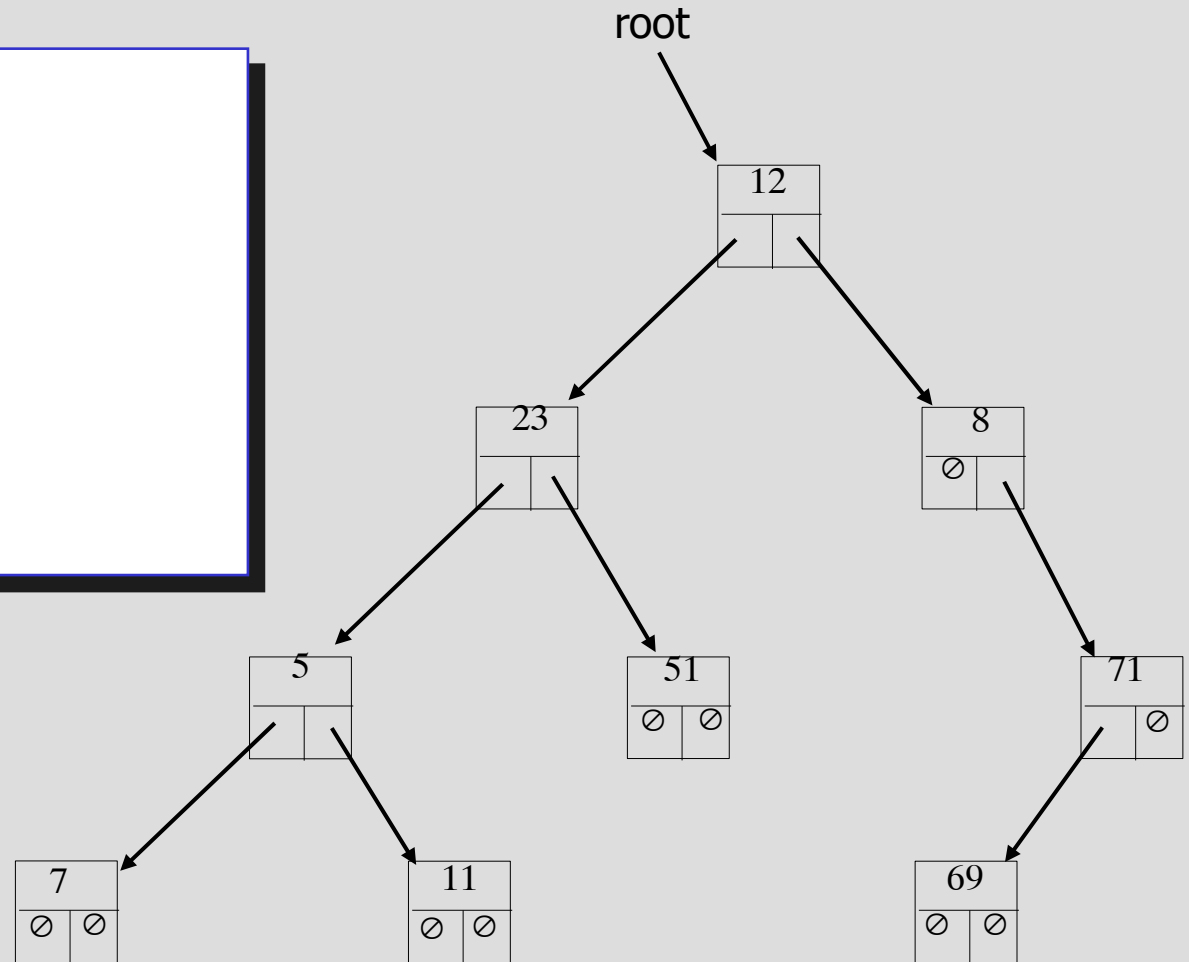


Run Time of Binary Search

- The range of candidate items to be searched is halved after comparing the key with the middle element.
- Binary search runs in $O(\log n)$ time.
- What about insertion and deletion?
 - search: $O(\log n)$
 - insert, delete: $O(n)$
 - min, max, predecessor, successor: $O(1)$
- The idea of a binary search can be extended to dynamic data structures → binary trees.

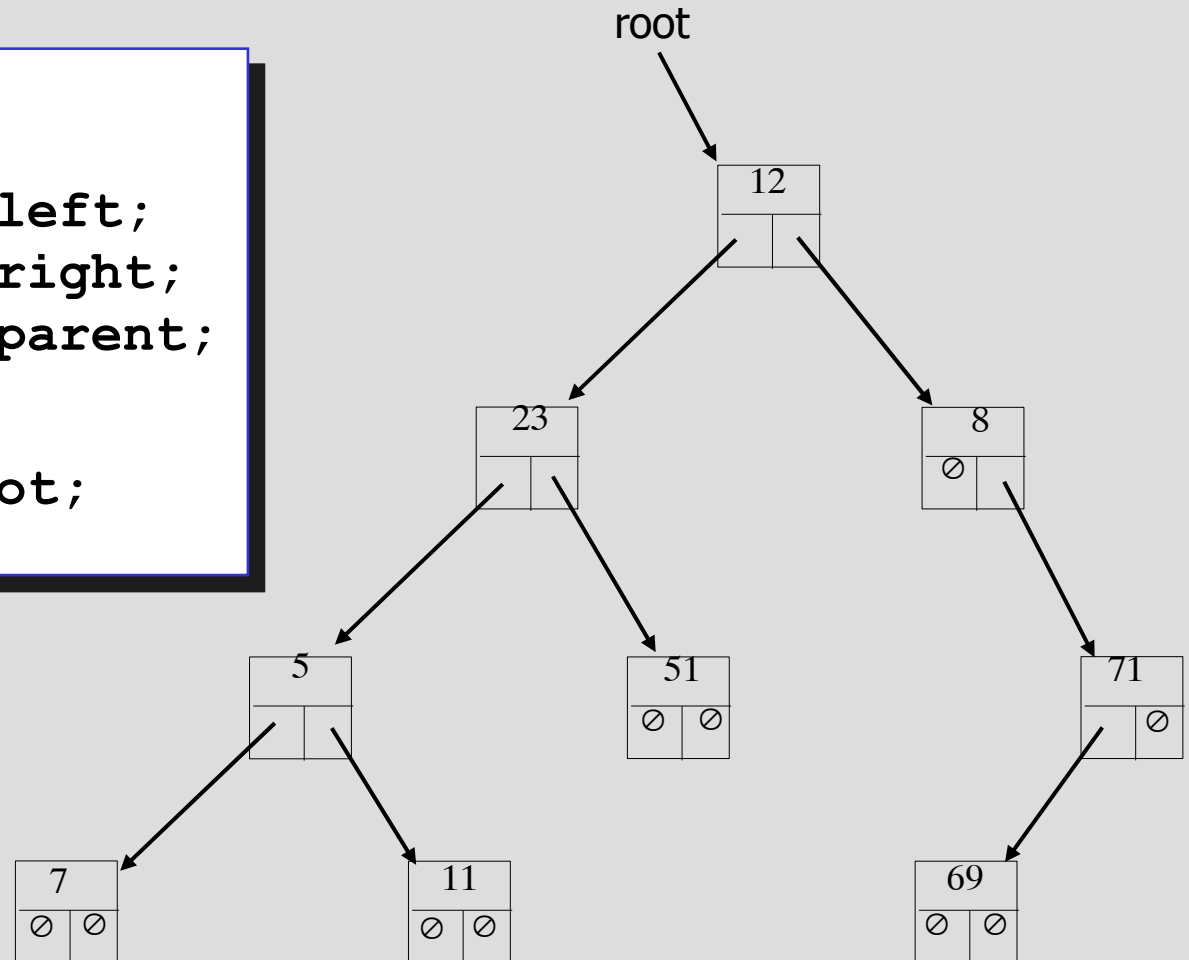
Binary Trees (Java)

```
class node {  
    int key;  
    node left;  
    node right;  
    node parent;  
}  
node root;
```



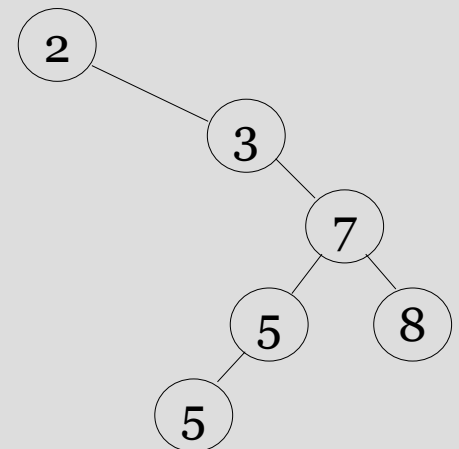
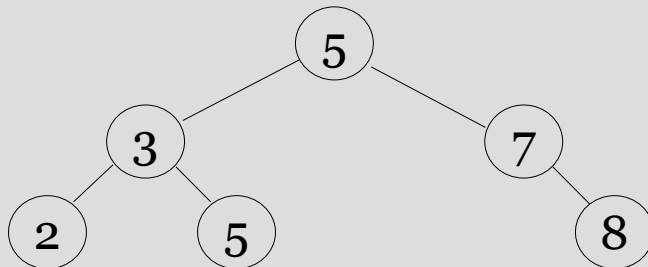
Binary Trees (C)

```
struct node {  
    int key;  
    struct node* left;  
    struct node* right;  
    struct node* parent;  
}  
  
struct node* root;
```



Binary Search Trees

- A **binary search tree** (BST) is a binary tree T with the following properties:
 - each internal node stores an item (k, e) of a dictionary
 - keys stored at nodes in the **left subtree** of v are **less than or equal to** k
 - keys stored at nodes in the **right subtree** of v are **greater than or equal to** k
- Example BSTs for 2, 3, 5, 5, 7, 8



Data Structures and Algorithms

Part 6

- **Binary Search Trees**
 - **Tree traversals**
 - Searching
 - Insertion
 - Deletion
- **Red-Black Trees**
 - Properties
 - Rotations
 - Insertion
 - Deletion

Tree Walks

- Keys in a BST can be printed using "tree walks"
- Keys of each node printed between keys in the left and right subtree – *inorder* tree traversal

```
InorderTreeWalk (x)
01   if x ≠ NIL then
02       InorderTreeWalk (x.left)
03       print x.key
04       InorderTreeWalk (x.right)
```

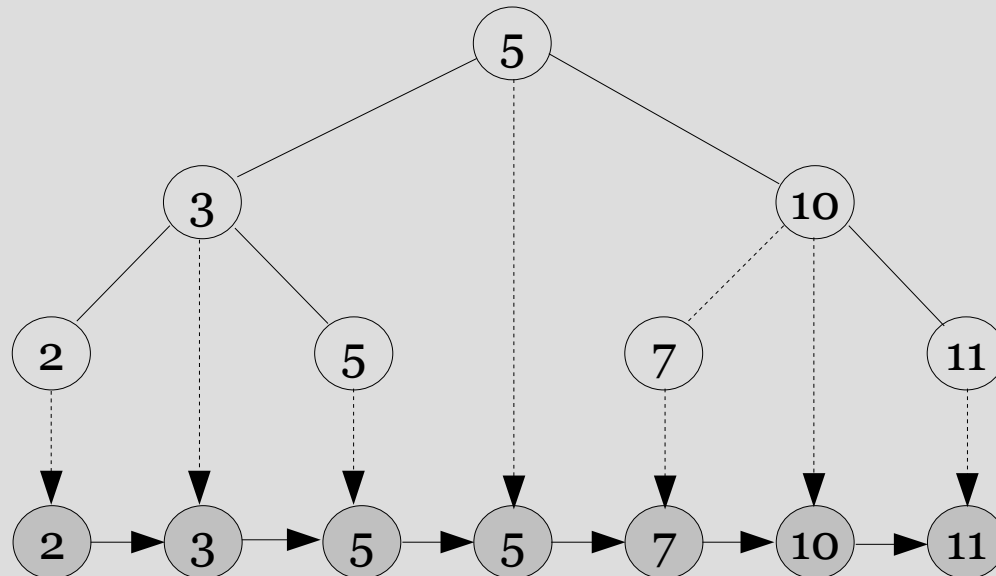
Tree Walks/2

- InorderTreeWalk is a divide-and-conquer algorithm.
- It prints all elements in monotonically increasing order.
- Running time $\Theta(n)$.

Tree Walks/2

4

- **Inorder tree walk** can be thought of as a projection of the BST nodes onto a one dimensional interval.



Tree Walks/3

Other forms of tree walk:

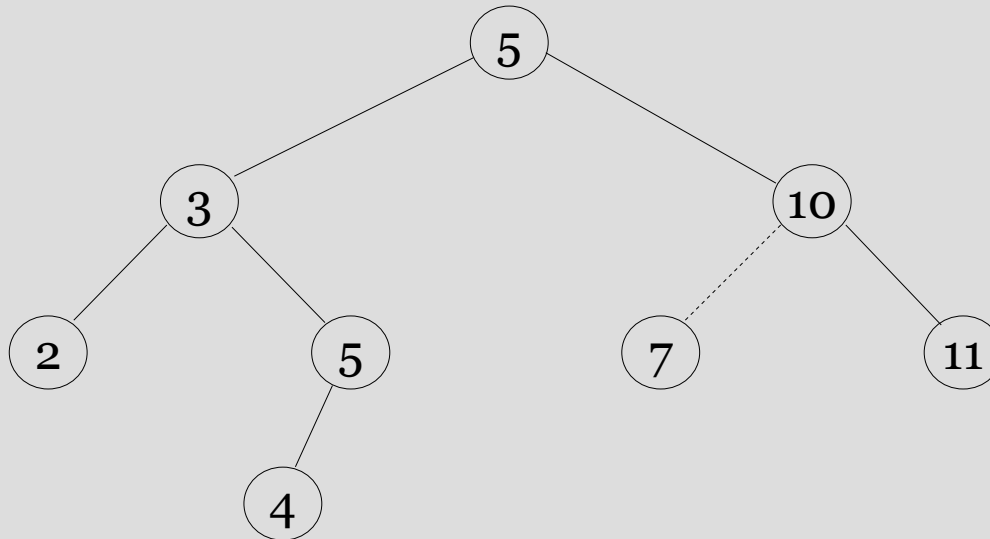
- A **preorder tree walk** processes each node before processing its children.
- A **postorder tree walk** processes each node after processing its children.

Data Structures and Algorithms

Part 6

- **Binary Search Trees**
 - Tree traversals
 - **Searching**
 - Insertion
 - Deletion
- **Red-Black Trees**
 - Properties
 - Rotations
 - Insertion
 - Deletion

Searching a BST



- To find an element with key k in a tree T
 - compare k with $T.key$
 - if $k < T.key$, search for k in $T.left$
 - otherwise, search for k in $T.right$

Pseudocode for BST Search

5

- Recursive version: divide-and-conquer

Search (T, k)

```
01 if T = NIL then return NIL
02 if k = T.key then return T
03 if k < T.key
04     then return Search(T.left, k)
05     else return Search(T.right, k)
```

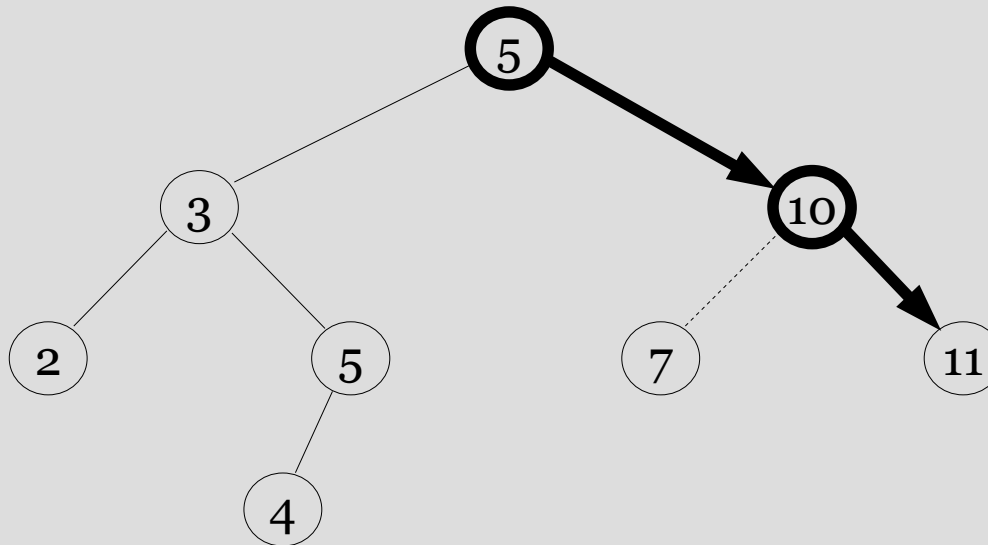
- Iterative version

Search (T, k)

```
01 x := T
02 while x ≠ NIL and k ≠ x.key do
03     if k < x.key
04         then x := x.left
05         else x := x.right
06 return x
```

Search Examples

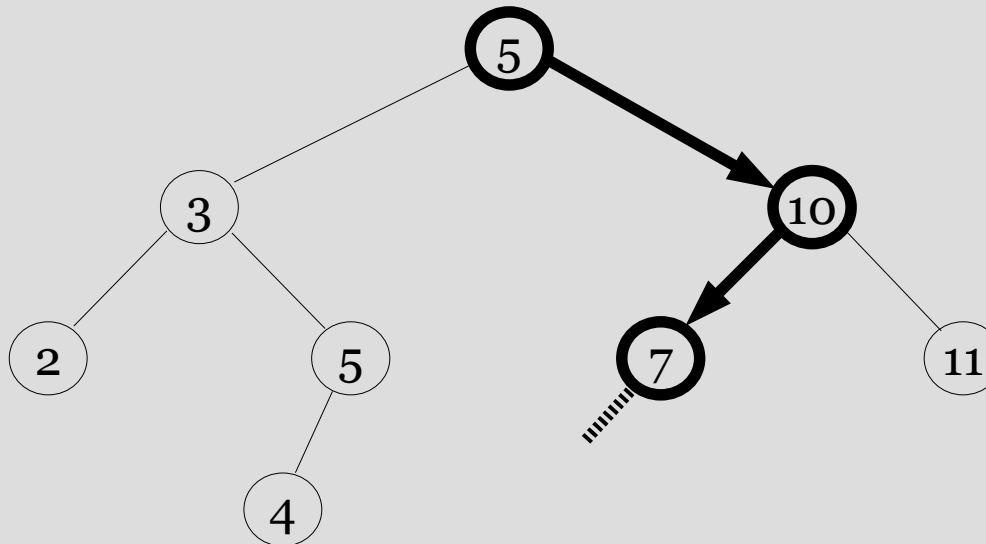
- $\text{Search}(T, 11)$



Search Examples/2

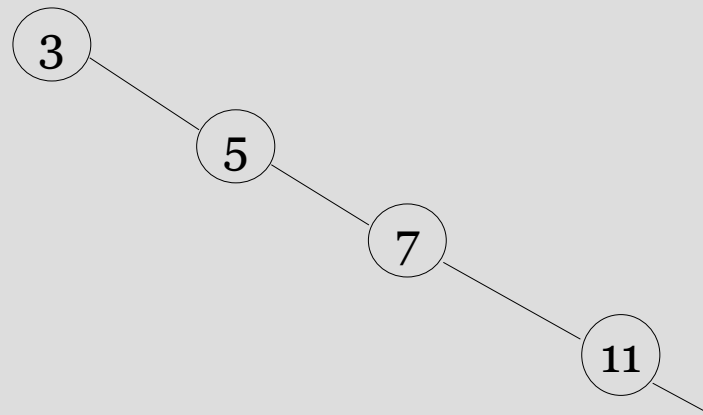
2

- $\text{Search}(T, 6)$



Analysis of Search

- Running time on tree of height h is $O(h)$
- After the insertion of n keys, the worst-case running time of searching is $O(n)$



BST Minimum (Maximum)

- Find the minimum key in a tree rooted at x .

```
TreeMinimum( $x$ )
```

```
01 while  $x$ .left  $\neq$  NIL do
```

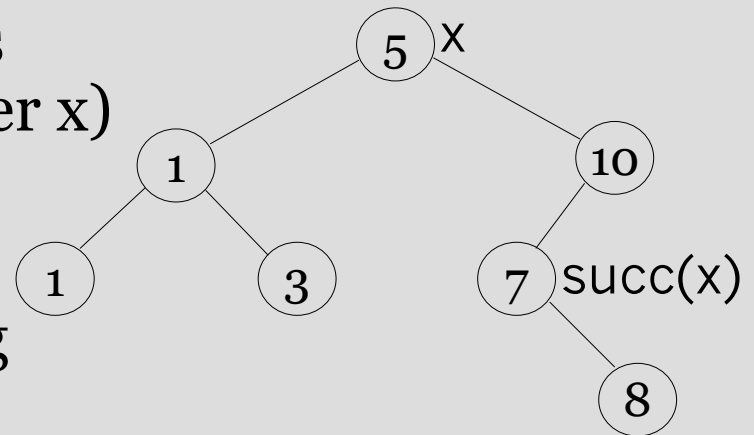
```
02    $x := x$ .left
```

```
03 return  $x$ 
```

- Maximum: same, x .right instead of x .left
- Running time $O(h)$, i.e., it is proportional to the height of the tree.

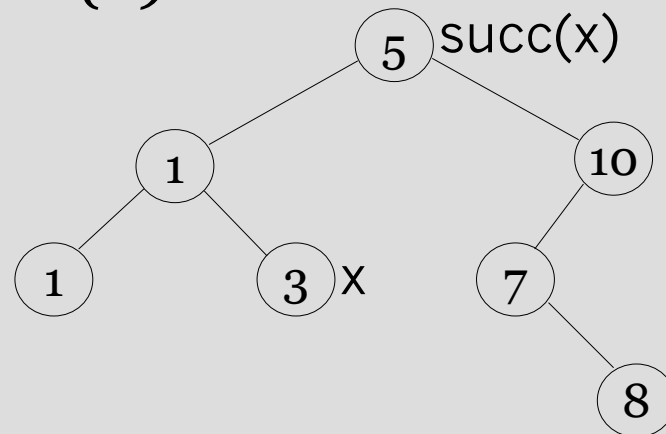
Successor

- Given x , find the node with the smallest key greater than $x.key$.
- We can distinguish two cases, depending on the right subtree of x
- Case 1: The right subtree of x is non-empty ($\text{succ}(x)$ inserted after x)
 - successor is the leftmost node in the right subtree.
 - this can be done by returning $\text{TreeMinimum}(x.\text{right})$.



Successor/2

- Case 2: the right subtree of x is empty ($\text{succ}(x)$, if any, was inserted before x).
 - The successor (if any) is the lowest ancestor of x whose left subtree contains x .
 - *Note: if x had a right child, then it would be smaller than $\text{succ}(x)$*



Successor Pseudocode

TreeSuccessor (x)

```
01 if x.right ≠ NIL
02     then return TreeMinimum(x.right)
03 y := x
04 while y.parent ≠ NIL and
05     y = y.parent.right
06     y := y.parent
07 return y
```

- For a tree of height h , the running time is $O(h)$.
- *Note: no comparison among keys needed!*

Successor with Trailing Pointer

Idea: Introduce `yp` to avoid dereferencing `y.parent`

```
TreeSuccessor (x)
01 if x.right ≠ NIL
02   then return TreeMinimum(x.right)
03   y := x
04   yp := y.parent
04   while yp ≠ NIL and y = yp.right do
05     y := yp
06     yp := y.parent
03 return yp
```

Data Structures and Algorithms

Chapter 6

- **Binary Search Trees**
 - Tree traversals
 - Searching
 - **Insertion**
 - Deletion
- **Red-Black Trees**
 - Properties
 - Rotations
 - Insertion
 - Deletion

BST Insertion

- The basic idea derives from searching:
 - construct an element p whose left and right children are NULL and insert it into T
 - find location in T where p belongs to (as if searching for $p.key$),
 - add p there
- The running time on a tree of height h is $O(h)$.

BST Insertion: Pseudocode

- Notice:
trailing
pointer
technique

```
TreeInsert(n, root)
  front:=root; rear:=NIL;
  while front ≠ NIL do
    rear:=front;
    if n.key < front.key
      then front:=front.left
      else front:=front.right
  if rear = NIL
    then n.parent:=NIL; return n;
  elsif n.key < rear.key
    then rear.left:=n;
    else rear.right:=n;
  n.parent:=rear;
  return root;
```

BST Insertion Code (java)

- Have a "one step delayed" pointer.

```
node insert(node p, node r) { //insert p in r
  node y = NULL; node x = r;
  while (x != NULL) {
    y := x;
    if (x.key < p.key) x = x.right;
    else x = x.left;
  }
  if (y == NULL) {r = p; p.parent=null;} // r is empty
  else if (y.key < p.key) y.right = p;
  else y.left = p;
  p.parent =y;
  return r;
}
```

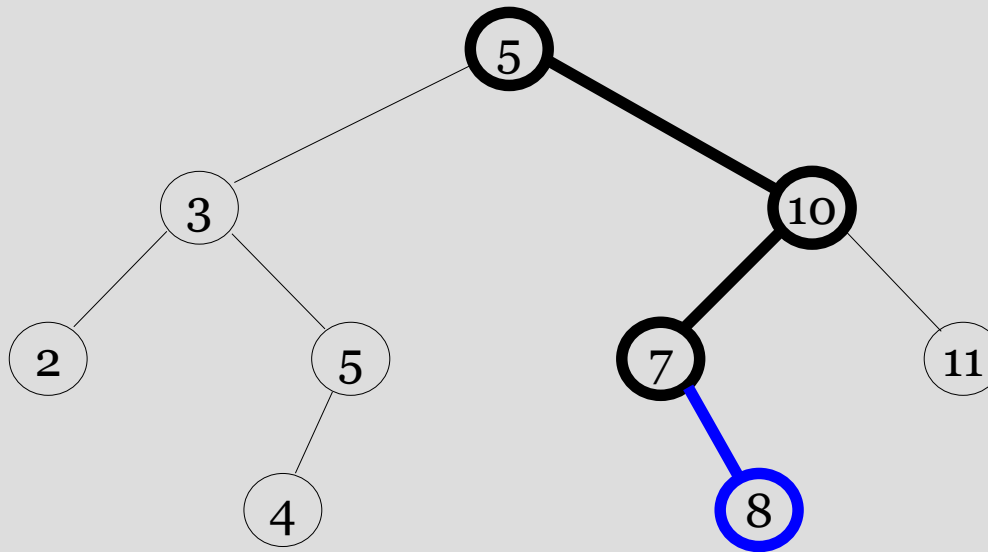

BST Insertion Code (C)

- Have a "one step delayed" pointer.

```
struct node* insert(struct node* p, struct node* r) {
    struct node* y = NULL; struct node* x = r;
    while (x != NULL) {
        y := x;
        if (x->key < p->key) x = x->right;
        else x = x->left;
    }
    if (y == NULL) {r = p;p->parent=NULL}
    else if (y->key < p->key) y->right = p;
    else y->left = p;
    p->parent = u;
    return r;
}
```

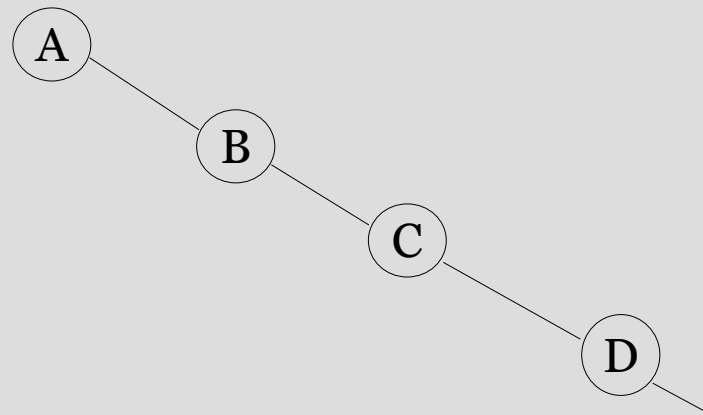
BST Insertion Example

- Insert 8



BST Insertion: Worst Case

- In what kind of sequence should the insertions be made to produce a BST of height n ?



BST Sorting

- Use `TreeInsert` and `InorderTreeWalk` to sort a list of n elements, A

TreeSort (A)

01 $T := \text{NIL}$

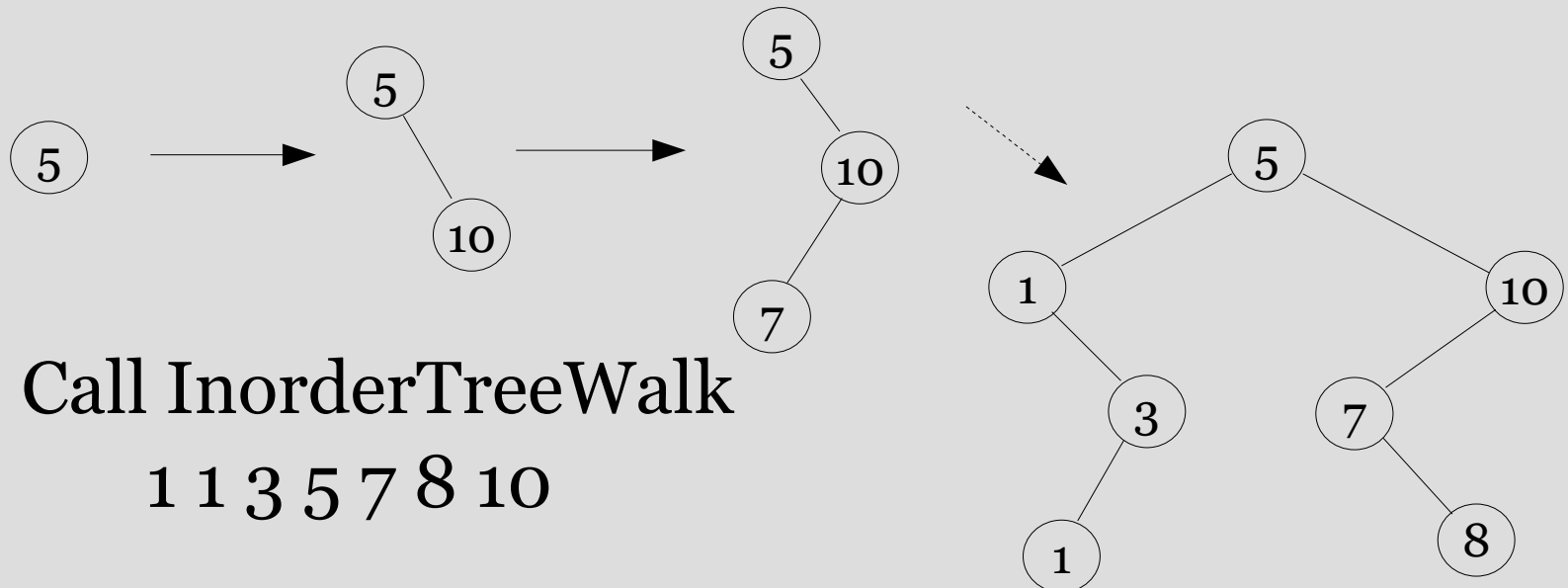
02 **for** $i := 1$ **to** n

03 `TreeInsert`(T , `BinTree`($A[i]$))

04 `InorderTreeWalk`(T)

BST Sorting/2

- Sort the following numbers
5 10 7 1 3 1 8
- Build a binary search tree



- Call InorderTreeWalk
1 1 3 5 7 8 10

Data Structures and Algorithms

Part 6

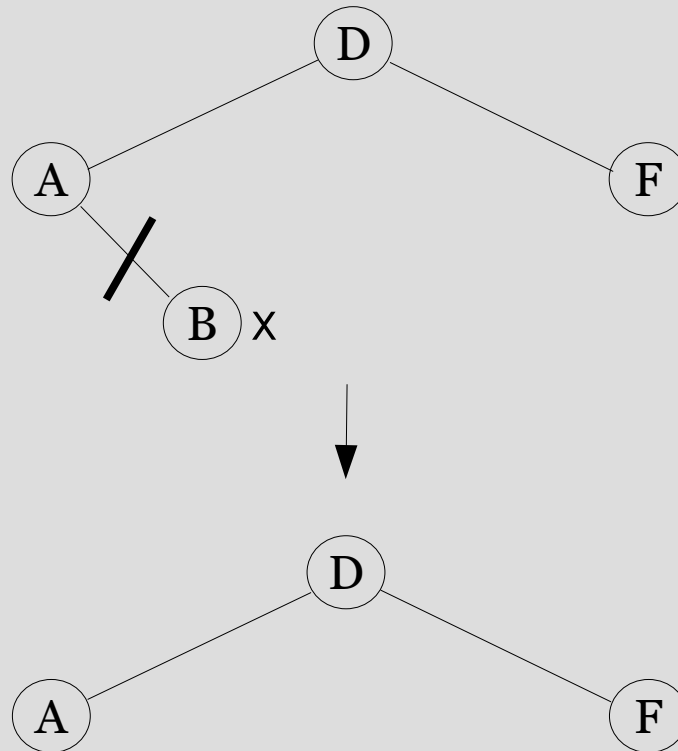
- **Binary Search Trees**
 - Tree traversals
 - Searching
 - Insertion
 - **Deletion**
- **Red-Black Trees**
 - Properties
 - Rotations
 - Insertion
 - Deletion

Deletion

- Delete node x from a tree T
- We can distinguish three cases
 - x has no child
 - x has one child
 - x has two children

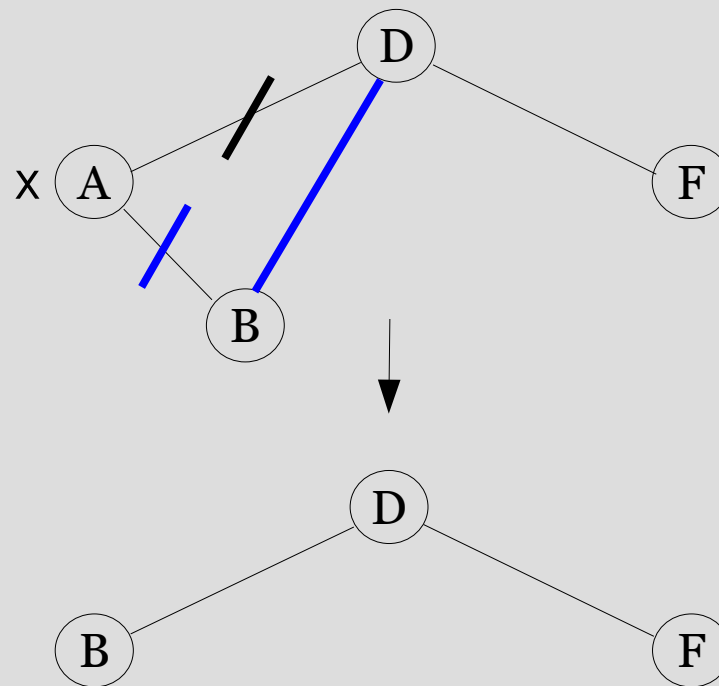
Deletion Case 1

- If x has no children: simply remove x



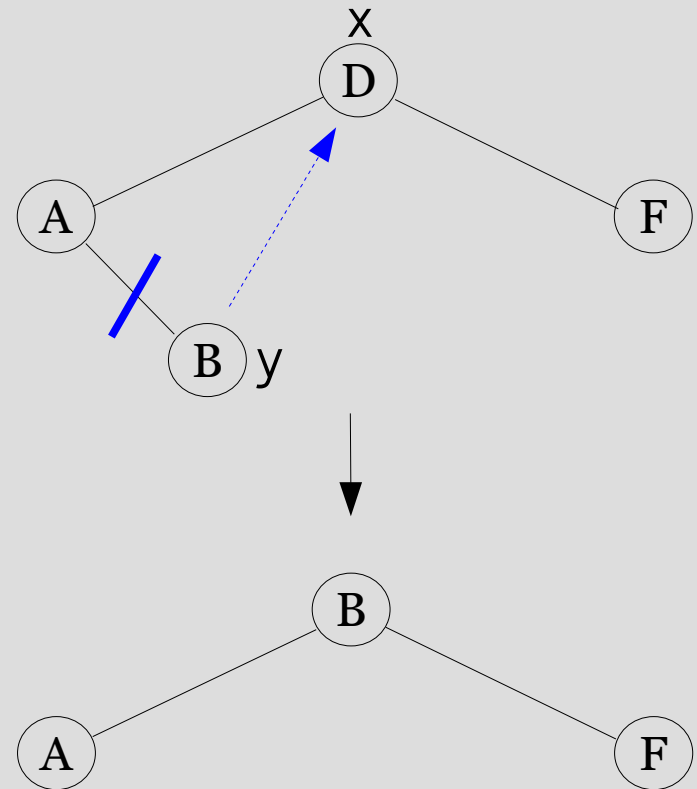
Deletion Case 2

- If x has exactly one child, make parent of x point to that child and delete x .



Deletion Case 3

- If x has two children:
 - find the largest child y in the left subtree of x (i.e. y is predecessor(x))
 - Recursively remove y (note that y has at most one child), and
 - replace x with y .
- “Mirror” version with successor(x) (CLRS)



Deletion Pseudocode

Delete (T, x)

```
if x.left = nil or x.right = nil
  then drop := x
  else drop := Succ(x)
if drop.left ≠ nil
  then keep := drop.left
  else keep := drop.right
if keep ≠ nil
  then keep.parent := drop.parent
if drop.parent = nil
  then T.root := keep
  else if drop = drop.parent.left
    then drop.parent.left := keep
  else drop.parent.right := keep
if drop ≠ x
  then x.key := drop.key
  % x.info := drop.info
```

Version with
parent pointer

BST Deletion Code (java)

- Version without “parent” field
- Note again the trailing pointer technique

```
node delete(node root, node x) {  
  
    front = root; rear = NULL;  
    while (front != x) {  
        rear := front;  
        if (x.key < front.key) front := front.left;  
        else front := front.right;  
    } // rear points to a parent of x (if any)  
  
    ...  
}
```

BST Deletion Code (java)/2

- x has less than 2 children
- Fix pointer of parent of x

```
...
    if (x.right == NULL) {
        if (rear == NULL) root = x.left;
        else if (rear.left == x) rear.left = x.left;
        else rear.right = x.left;}
    else if (x.left == NULL) {
        if (rear == NULL) root = x.right;
        else if (rear.left == x) rear.left = x.right;
        else rear.right = x.right;
    else {
...

```

BST Deletion Code (java)/3

- x has 2 children

```
succ = x.right; srear = succ;
while (succ.left != NULL)
    { srear:=succ; succ:=succ.left; }

if (rear == NULL) root = succ;
else if (rear.left == x) rear.left = succ;
else rear.right = succ;

succ.left = x.left;
if (srear != succ) {
    srear.left = succ.right;
    succ.right = x.right;
}
return root
```

BST Deletion Code (C)

- Version without “parent” field

```
struct node* delete(struct node* root,
                   struct node* x) {

    u = root; v = NULL;
    while (u != x) {
        v := u;
        if (x->key < u->key) u := u->left;
        else u := u->right;
    } // v points to a parent of x (if any)

    ...
}
```

BST Deletion Code (C)/2

- x has less than 2 children
- Fix pointer of parent of x

```
...
if (u->right == NULL) {
    if (v == NULL) root = u->left;
    else if (v->left == u) v->left = u->left;
    else v->right = u->left;
else if (u->left == NULL) {
    if (v == NULL) root = u->right;
    else if (v->left == u) v->left = u->right;
    else v->right = u->right;
else {
...

```


BST Deletion Code (C)/3

- x has 2 children

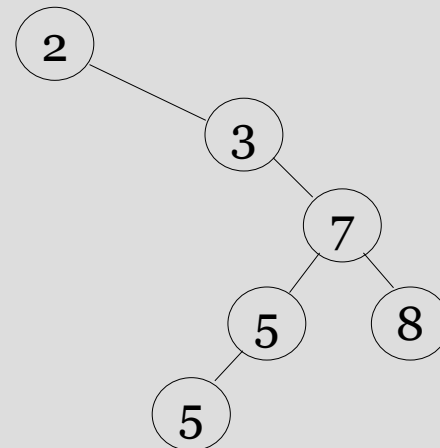
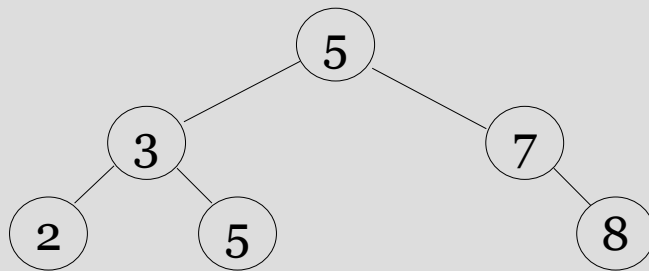
```
p = x->left; q = p;
while (p->right != NULL) { q:=p; p:=p->right; }

if (v == NULL) root = p;
else if (v->left == u) v->left = p;
else v->right = p;

p->right = u->right;
if (q != p) {
    q->right = p->left;
    p->left = u->left;
}
return root
```

Balanced Binary Search Trees

- Problem: execution time for tree operations is $\Theta(h)$, which in worst case is $\Theta(n)$.
- Solution: balanced search trees *guarantee* small height $h = O(\log n)$.



Suggested exercises

- Implement a binary search tree with the following functionalities:
 - init, max, min, successor, predecessor, search (iterative & recursive), insert, delete (both swap with succ and pred), print, print in reverse order
 - TreeSort

Suggested exercises/2

Using paper & pencil:

- draw the trees after each of the following operations, starting from an empty tree:
 1. Insert 9,5,3,7,2,4,6,8,13,11,15,10,12,16,14
 2. Delete 16, 15, 5, 7, 9 (both with succ and pred strategies)
- simulate the following operations after 1:
 - Find the max and minimum
 - Find the successor of 9, 8, 6

Data Structures and Algorithms

Chapter 6

- Binary Search Trees
 - Tree traversals
 - Searching
 - Insertion
 - Deletion
- **Red-Black Trees**
 - Properties
 - Rotations
 - Insertion
 - Deletion

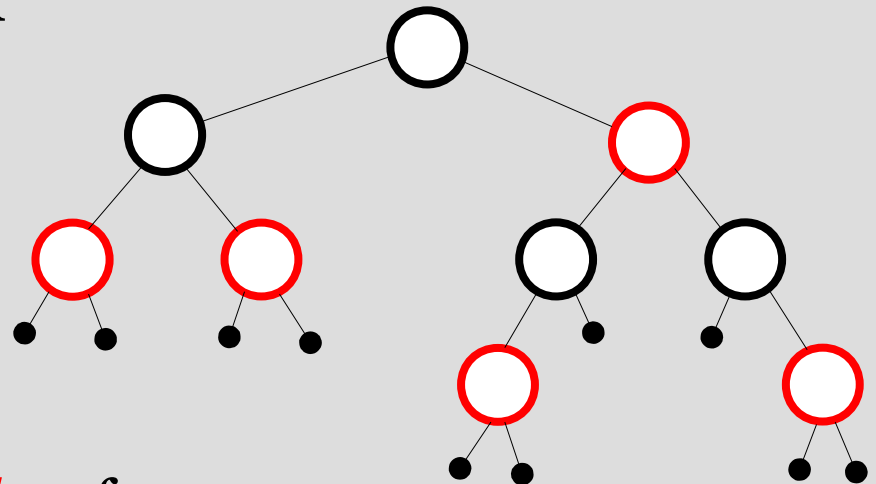
Data Structures and Algorithms

Chapter 6

- Binary Search Trees
 - Tree traversals
 - Searching
 - Insertion
 - Deletion
- Red-Black Trees
 - Properties
 - Rotations
 - Insertion
 - Deletion

Red/Black Trees

- A **red-black** tree is a binary search tree with the following properties:
 1. Nodes are colored **red** or **black**
 2. NULL leaves are **black**
 3. The root is **black**
 4. No two consecutive **red nodes** on any root-leaf path.
 5. Same number of black nodes on any root-leaf path (called *black height* of the tree).



Java's TreeMap

[Overview](#) [Package](#) **Class** [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

Java™ Platform
Standard Ed. 6

[PREV CLASS](#) [NEXT CLASS](#)

[FRAMES](#) [NO FRAMES](#) [All Classes](#)

SUMMARY: [NESTED](#) | [FIELD](#) | [CONSTR](#) | [METHOD](#)

DETAIL: [FIELD](#) | [CONSTR](#) | [METHOD](#)

java.util

Class TreeMap<K,V>

[java.lang.Object](#)

└ [java.util.AbstractMap<K,V>](#)

└ [java.util.TreeMap<K,V>](#)

Type Parameters:

k - the type of keys maintained by this map

v - the type of mapped values

All Implemented Interfaces:

[Serializable](#), [Cloneable](#), [Map<K,V>](#), [NavigableMap<K,V>](#), [SortedMap<K,V>](#)

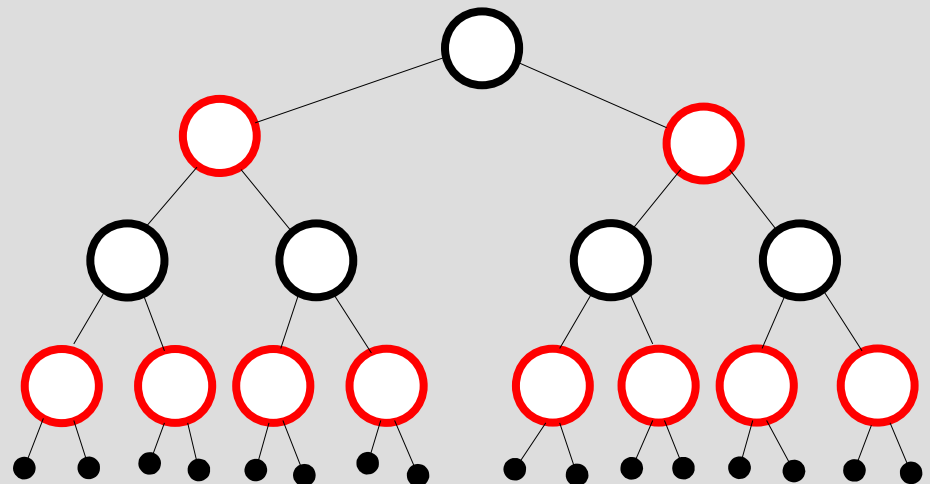
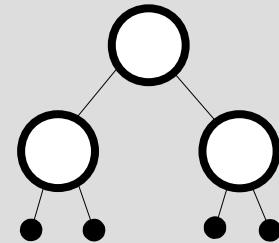
```
public class TreeMap<K,V>  
extends AbstractMap<K,V>  
implements NavigableMap<K,V>, Cloneable, Serializable
```

A Red-Black tree based [NavigableMap](#) implementation. The map is sorted according to the [natural ordering](#) of its keys, or by a [Comparator](#) provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

RB-Tree Properties

- Some measures
 - n – # of internal nodes
 - h – height
 - bh – black height
- $2^{bh} - 1 \leq n$
- $h/2 \leq bh$
- $2^{h/2} \leq n + 1$
- $h \leq 2 \log(n + 1)$
- **BALANCED!**



RB-Tree Properties/2

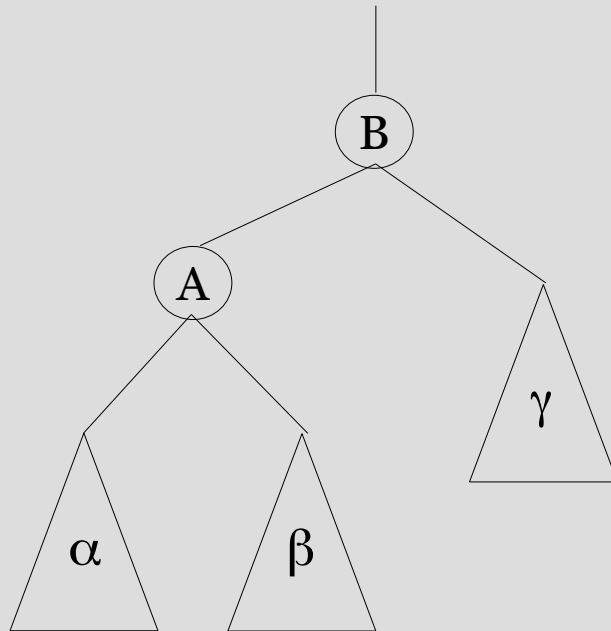
- Operations on a **binary-search tree** (search, insert, delete, ...) can be accomplished in $O(h)$ time.
- The **RB-tree** is a binary search tree, whose **height** is **bounded by $2 \log(n + 1)$** , thus the operations run in $O(\log n)$.
 - Provided that we can **maintain** red-black tree properties spending no more than $O(h)$ time on each insertion or deletion.

Data Structures and Algorithms

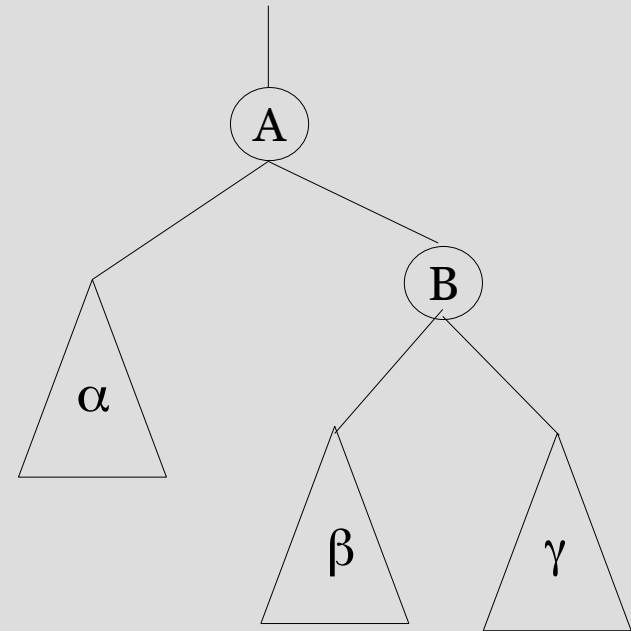
Chapter 6

- Binary Search Trees
 - Tree traversals
 - Searching
 - Insertion
 - Deletion
- Red-Black Trees
 - Properties
 - Rotations
 - Insertion
 - Deletion

Rotation



right rotation of B



left rotation of A

Right Rotation

RightRotate (B)

01 **A := B.left**

02 **B.left := A.right**

03 **B.left.parent := B**

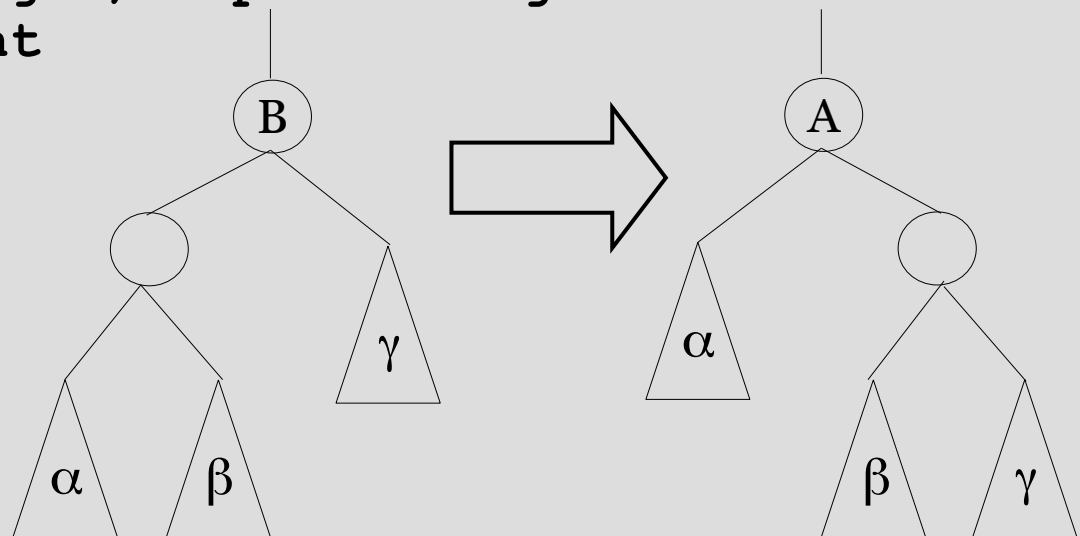
04 **if (B = B.parent.left) B.parent.left := A**

05 **if (B = B.parent.right) B.parent.right := A**

06 **A.parent := B.parent**

07 **A.right := B**

08 **B.parent := A**



The Effect of a Rotation

- Maintains inorder key ordering
 - $\forall a \in \alpha, b \in \beta, c \in \gamma$
we can state the invariant
 - $a \leq A \leq b \leq B \leq c$
- After right rotation
 - Depth(α) decreases by 1
 - Depth(β) stays the same
 - Depth(γ) increases by 1
- Left rotation: symmetric
- Rotation takes $O(1)$ time

Data Structures and Algorithms

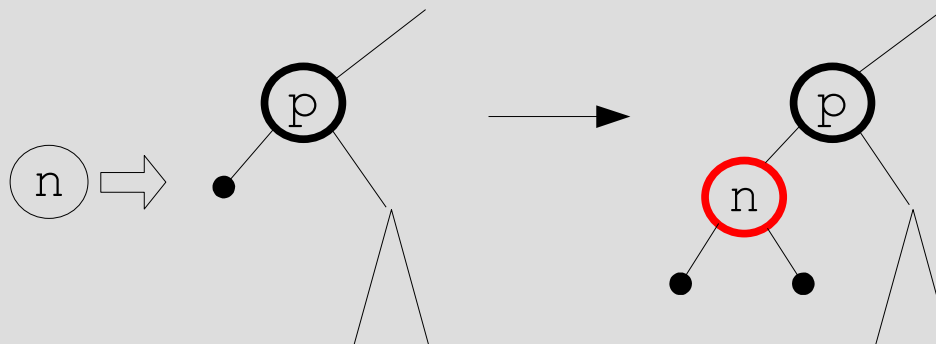
Chapter 6

- Binary Search Trees
 - Tree traversals
 - Searching
 - Insertion
 - Deletion
- **Red-Black Trees**
 - Properties
 - Rotations
 - **Insertion**
 - Deletion

Insertion in the RB-Trees

RBInsert (T, n)

- 01 *Insert n into T using the binary search tree insertion procedure*
- 02 $n.\text{left} := \text{NIL}$
- 03 $n.\text{right} := \text{NIL}$
- 04 $n.\text{color} := \text{red}$
- 05 **RBInsertFixup** (n)

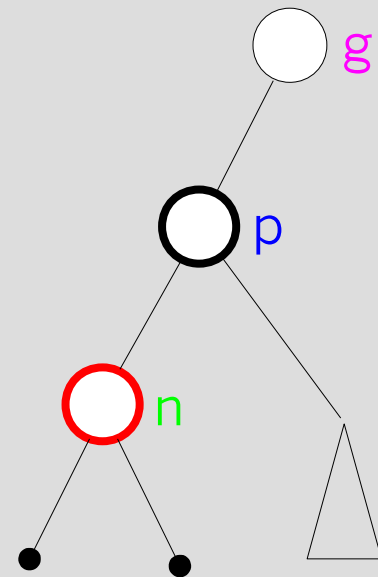


Fixing Up a Node: Intuition

- Case 0: parent is black
=> *ok*
- Case 1: both parent and uncle are red
=> change colour of parent/uncle to black
=> change colour of grandparent to red
=> *fix up the grandparent*
Exception: grandparent is root => then keep it black
- Case 2: parent is red and uncle is black, and
node and parent are in a straight line
=> *rotate at grandparent*
- Case 3: parent is red and uncle is black, and
node and parent are **not** in a straight line
=> *rotate at parent* (leads to Case 2)

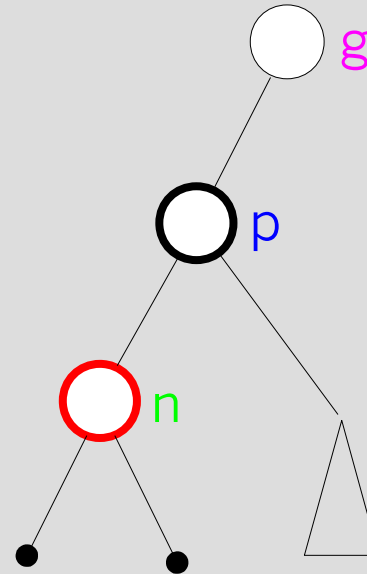
Insertion

- Let
 - n = the new node
 - p = n .parent
 - g = p .parent
- In the following assume:
 - p = g .left



Insertion: Case 0

- **p.color = black**
 - No properties of the tree are violated
 - we are done.



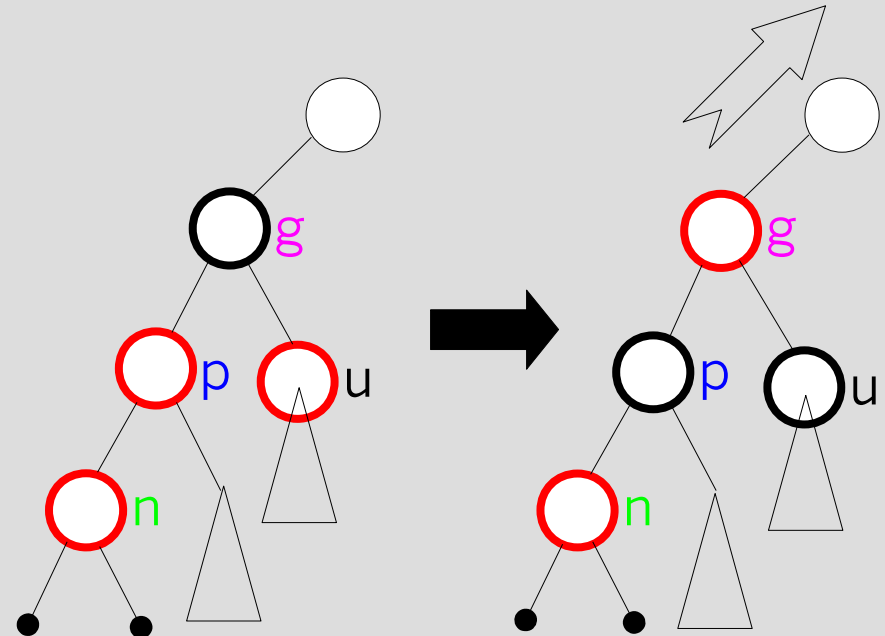
Insertion: Case 1

6

- Case 1
 - n 's uncle u is red

- Action

- $p.\text{color} := \text{black}$
- $u.\text{color} := \text{black}$
- $g.\text{color} := \text{red}$
- $n := g$



- Note: the tree rooted at g is balanced enough (black depth of all descendants remains unchanged).

Insertion: Case 2

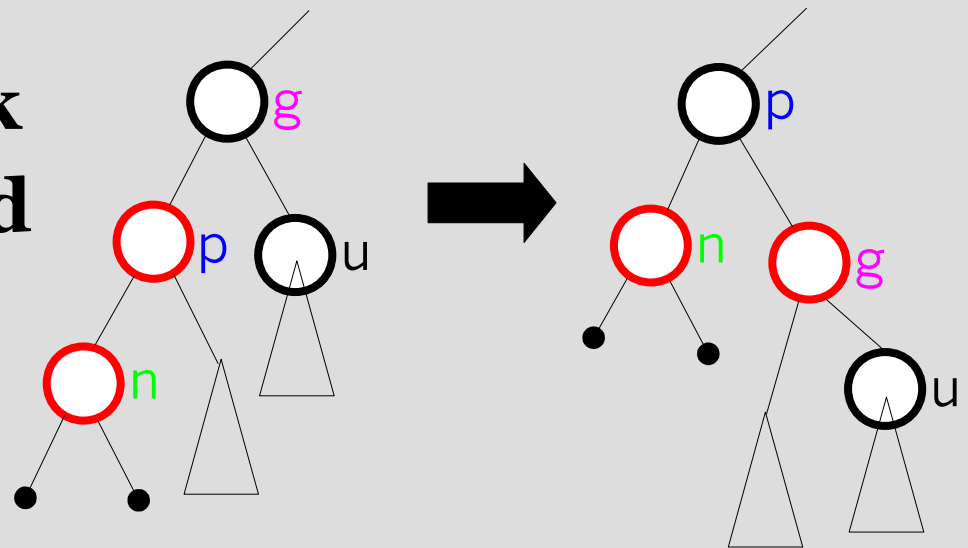
■ Case 2

- n 's uncle u is black and n is a left child

■ Action

- $p.\text{color} := \text{black}$
- $g.\text{color} := \text{red}$
- $\text{RightRotate}(g)$

- Note: the tree rooted at g is balanced enough (black depth of all descendants remains unchanged).



Insertion: Case 3

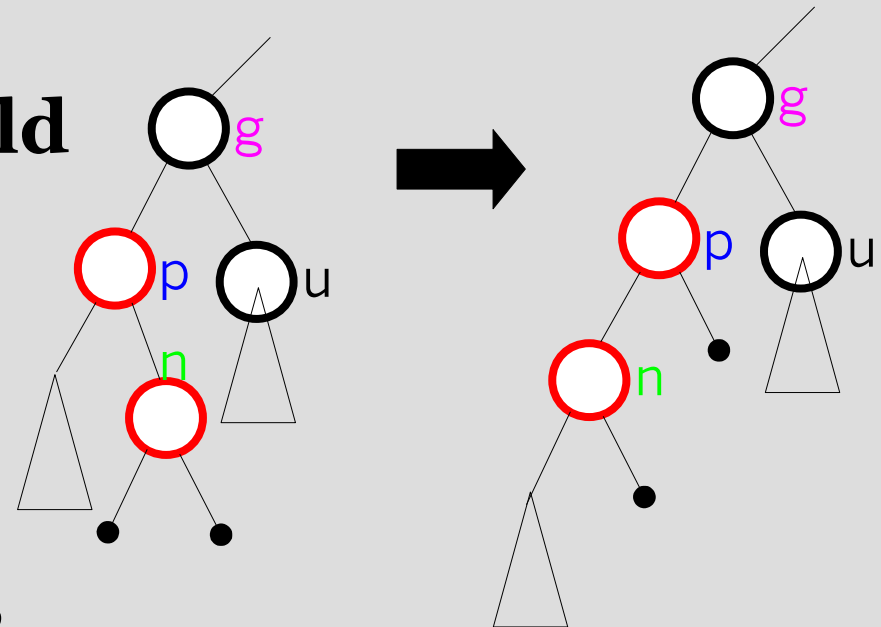
- Case 3
 - n 's uncle u is black and n is a right child

- Action

- `LeftRotate(p)`
- $n := p$

- Note

- The result is a case 2.



Insertion: Mirror cases

- All three cases are handled analogously if **p** is a right child.
- Exchange *left* and *right* in all three cases.

Insertion: Case 2 and 3 mirrored

■ Case 2m

- **n 's uncle u is black and n is a *right* child**
- Action
- `p.color := black`
- `g.color := red`
- `LeftRotate(g)`

■ Case 3m

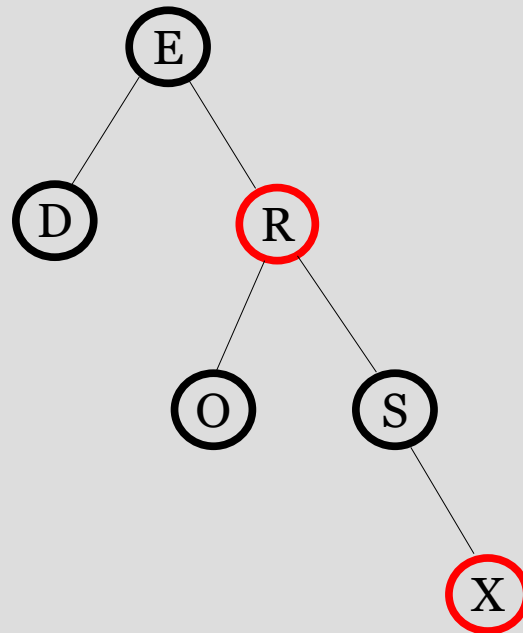
- **n 's uncle u is black and n is a *left* child**
- Action
 - `RightRotate(p)`
 - `n := p`

Insertion Summary

- If two **red** nodes are adjacent, we do either
 - a **restructuring** (with one or two rotations) and **stop** (cases 2 and 3), or
 - recursively **propagate** red upwards (case 1)
- A **restructuring** takes constant time and is performed at most once. It reorganizes an off-balanced section of the tree
- **Propagations** may continue up the tree and are executed $O(\log n)$ times (height of the tree)
- The running time of an insertion is $O(\log n)$.

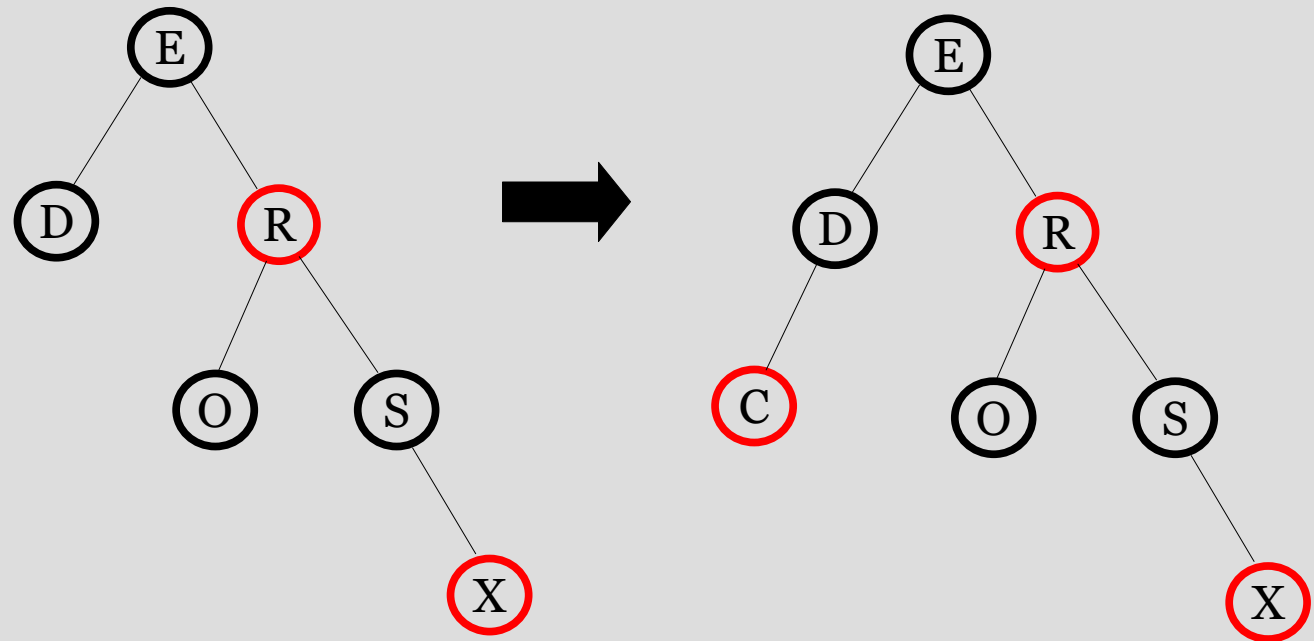
An Insertion Example

- Inserting "REDSOX" into an empty tree

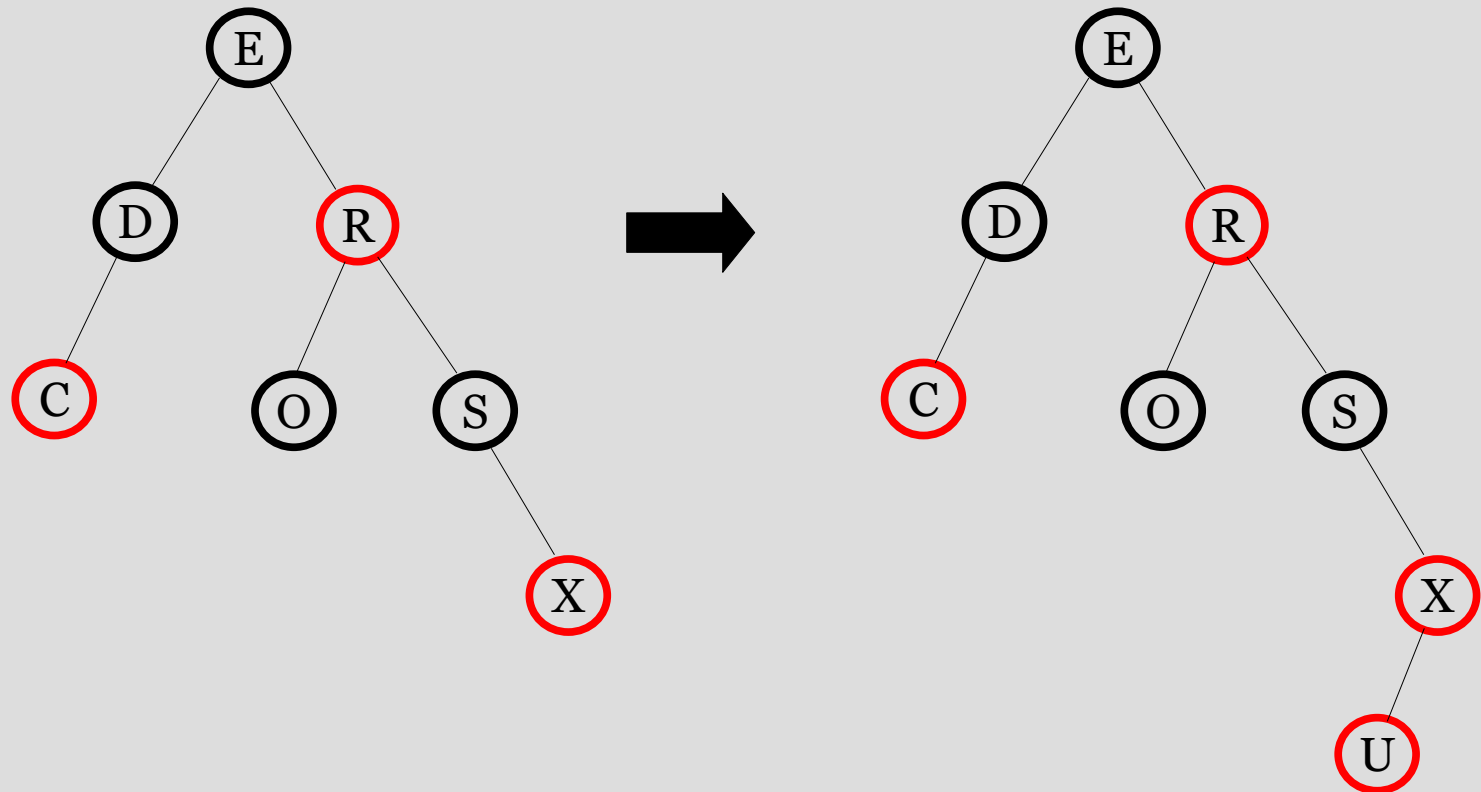


- Now, let us insert "CUBS"

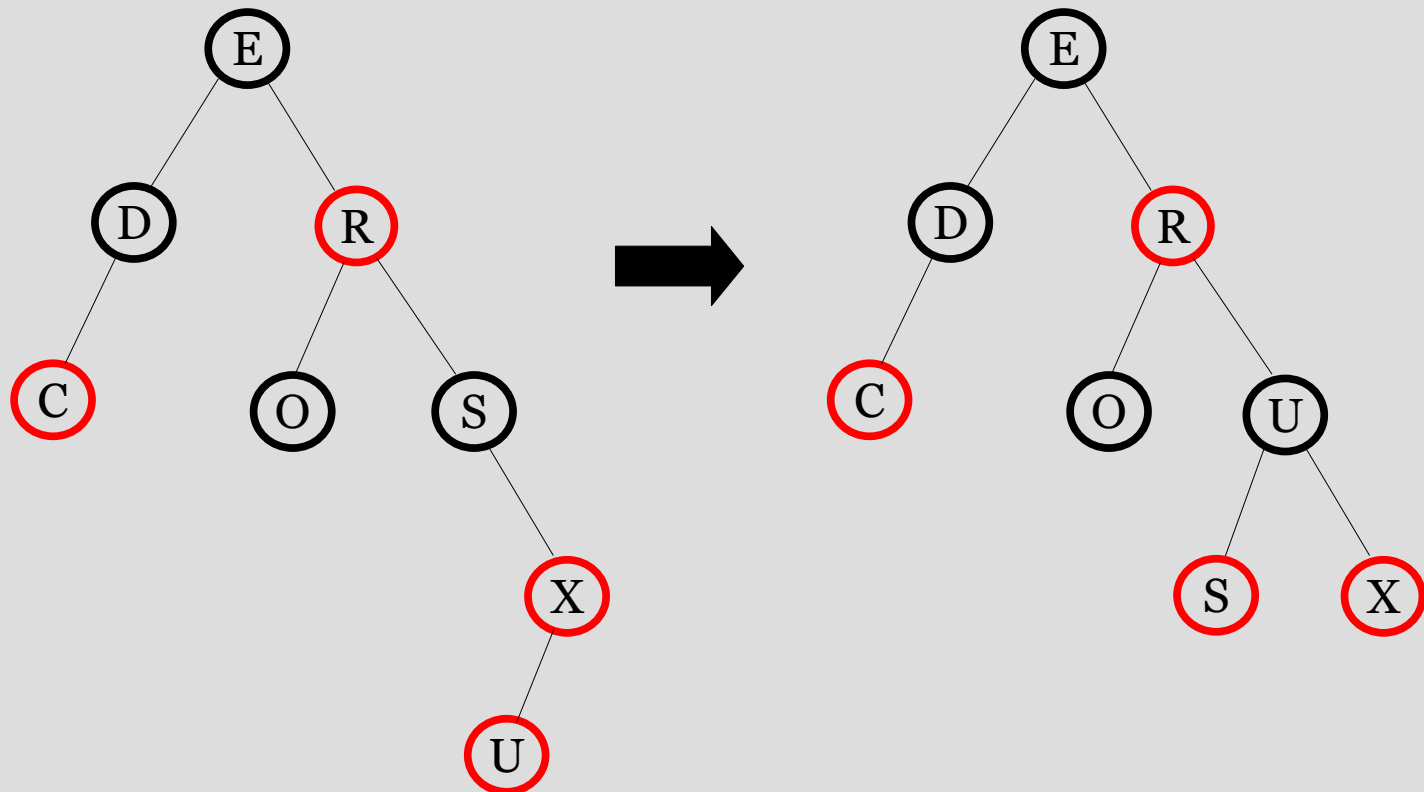
Insert C (case 0)



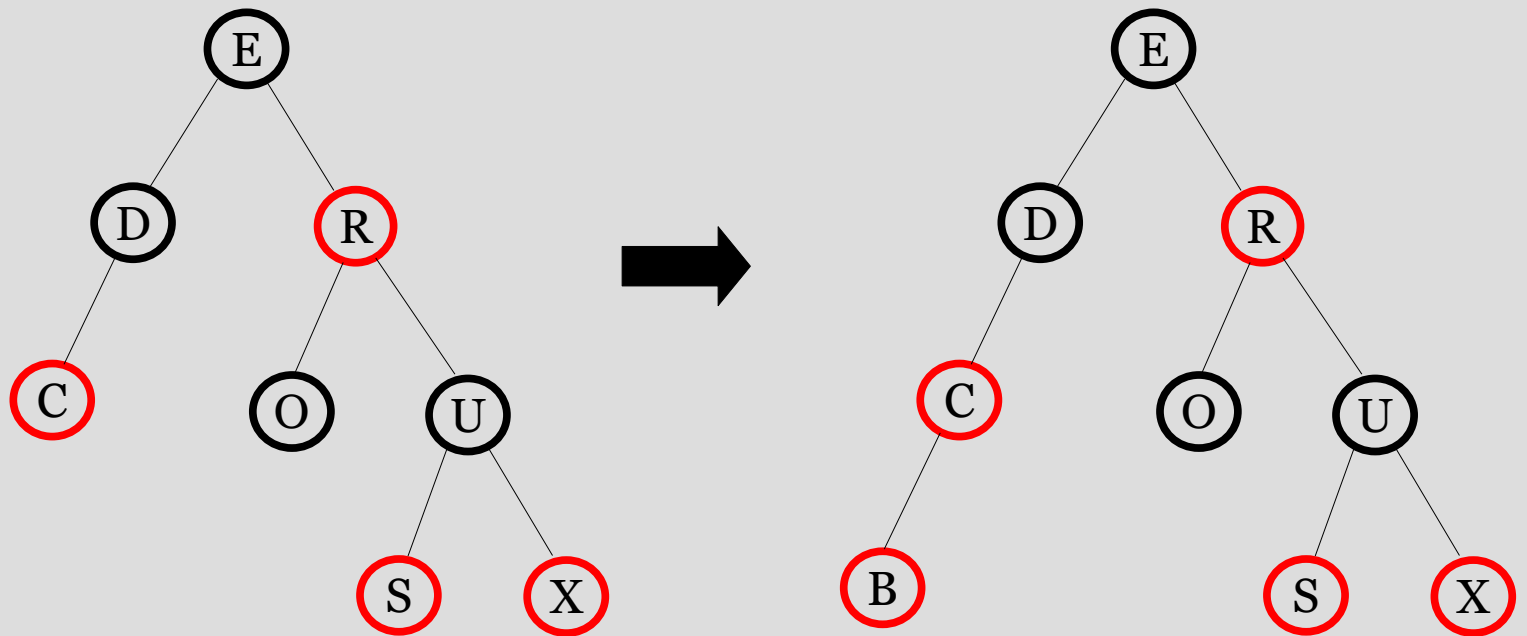
Insert U (case 3, mirror)



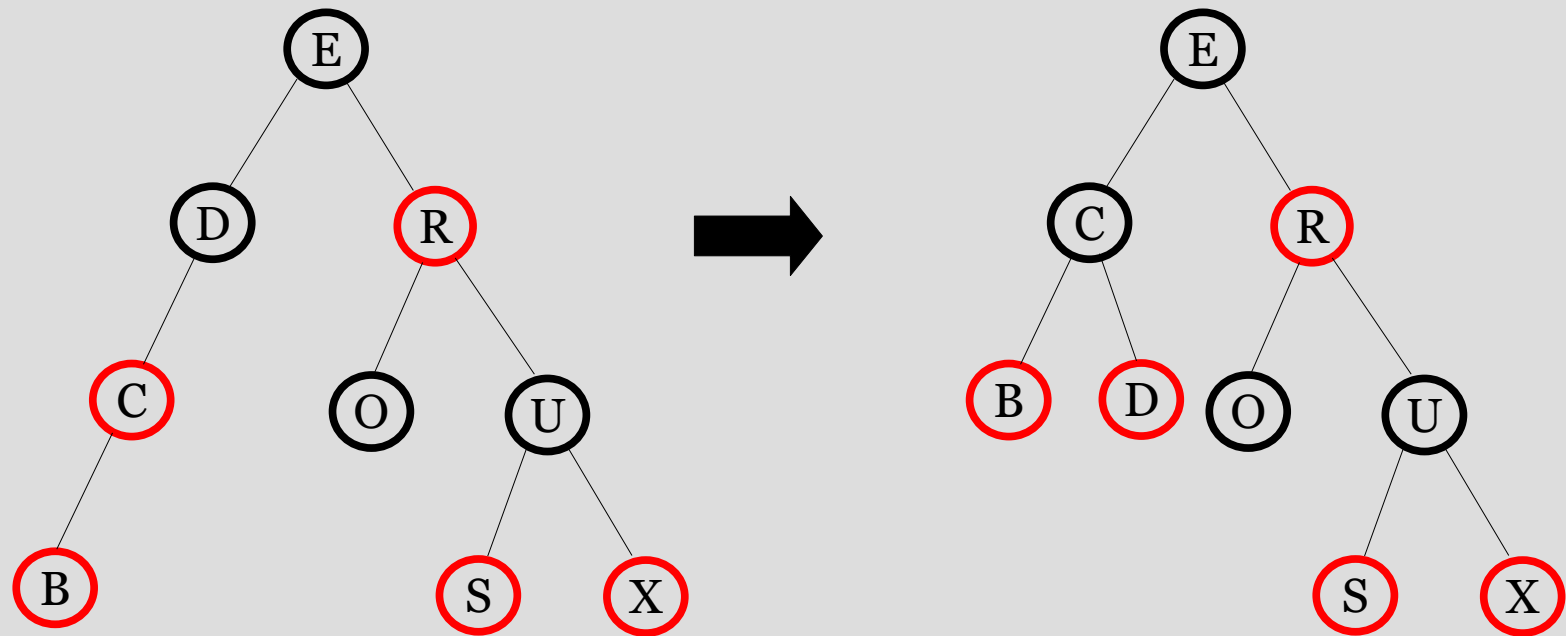
Insert U/2



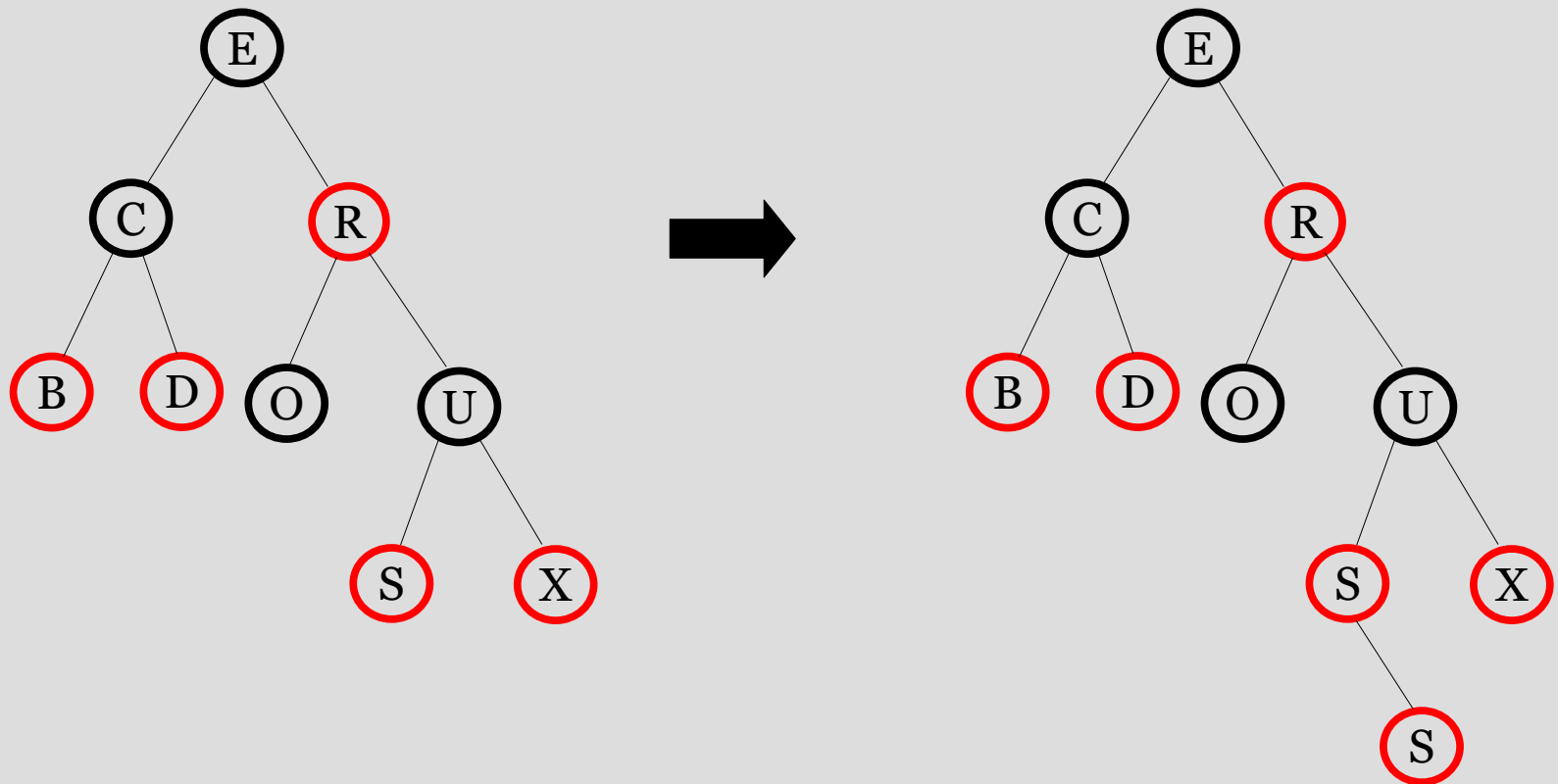
Insert B (case 2)



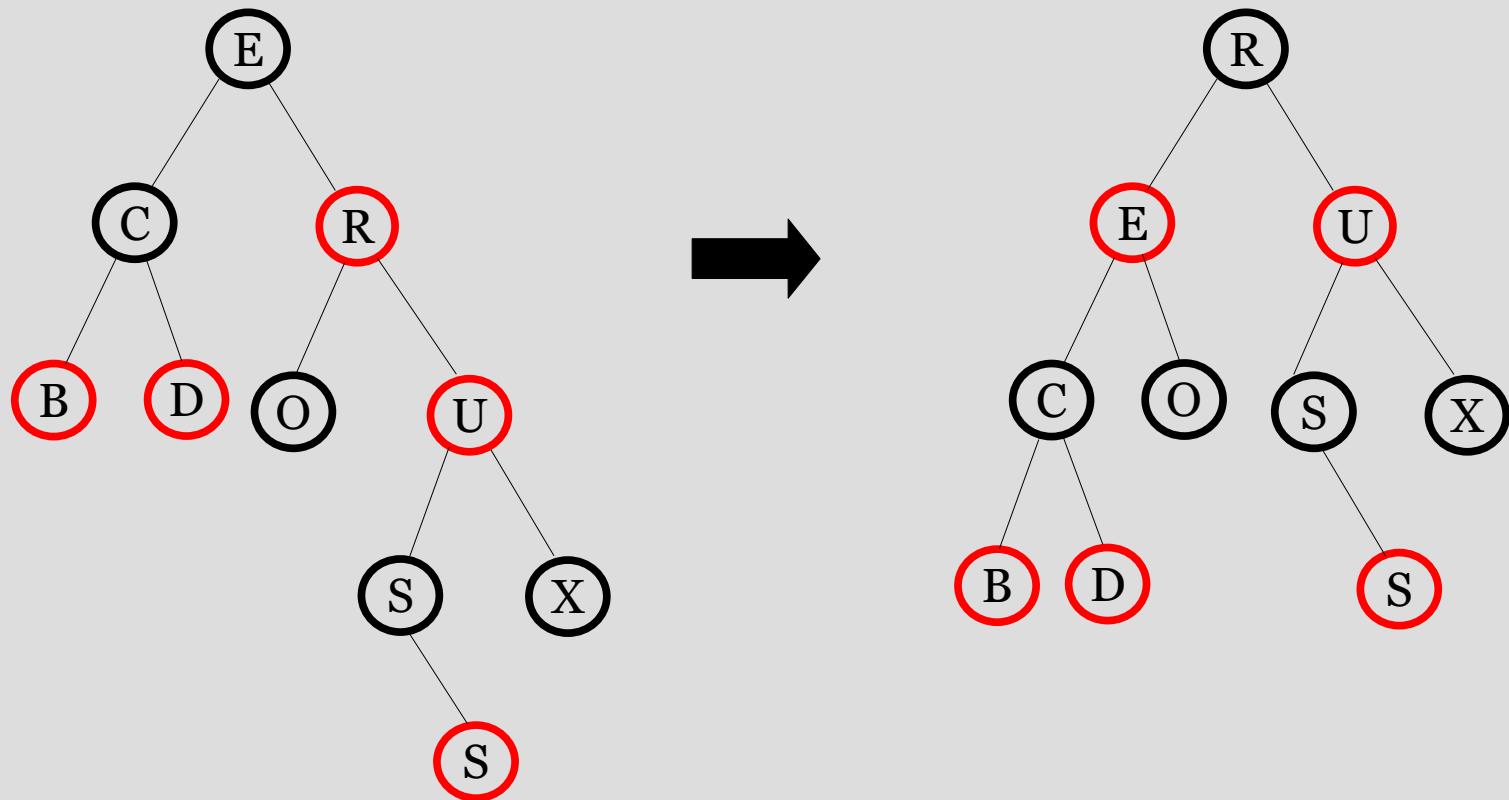
Insert B/2



Insert S (case 1)



Insert S/2 (case 2 mirror)



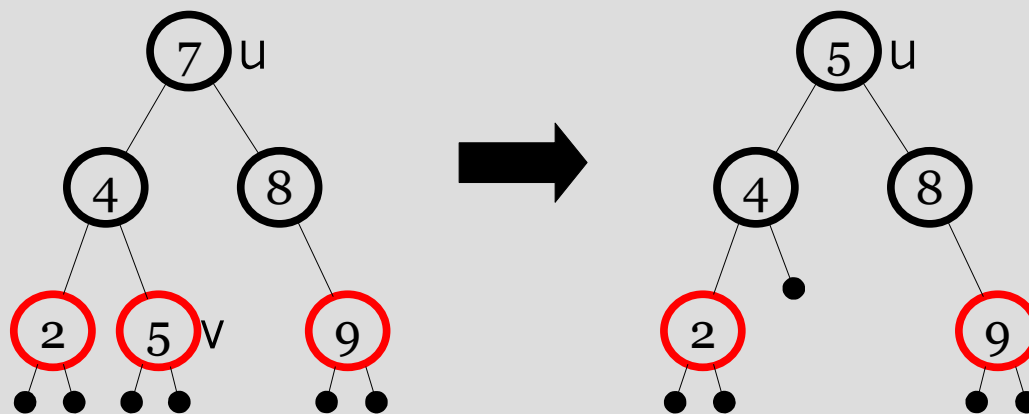
Data Structures and Algorithms

Chapter 6

- Binary Search Trees
 - Tree traversals
 - Searching
 - Insertion
 - Deletion
- **Red-Black Trees**
 - Properties
 - Rotations
 - Insertion
 - **Deletion**

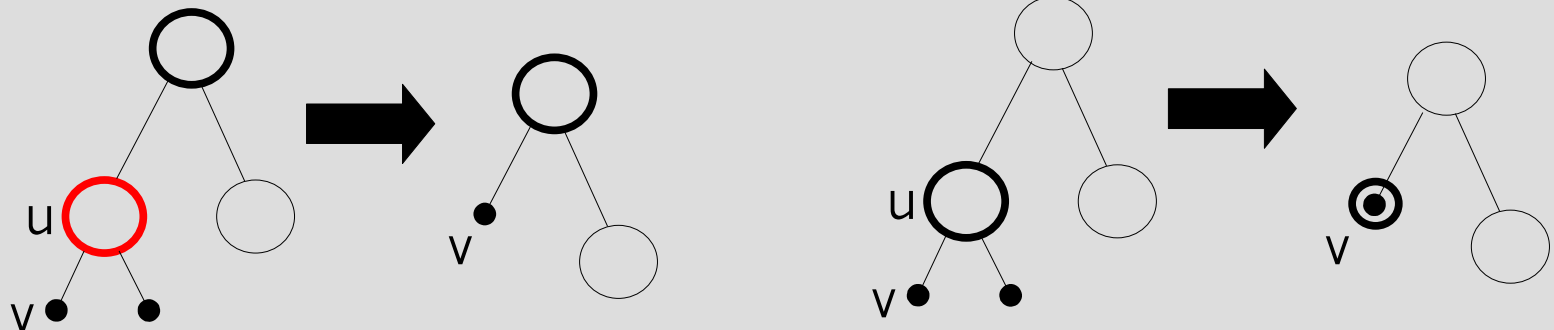
Deletion

- We first apply binary search tree deletion.
 - We can easily delete a node that has at least one *nil* child
 - If the key to be deleted is stored at a node u with two children, we replace its content with the content of the largest node v of the left subtree and delete v instead.



Deletion Algorithm

1. Remove u
2. If $u.\mathbf{color} = \mathbf{red}$, we are done. Else, assume that v (replacement of u) gets *additional black color*:
 - If $v.\mathbf{color} = \mathbf{red}$ then $v.\mathbf{color} := \mathbf{black}$ and we are done!
 - Else v 's color is “**double black**”.



Deletion Algorithm/2

- How to eliminate double black edges?
 - The intuitive idea is to perform a **color compensation**
 - Find a **red** edge nearby, and change the pair (**red, double black**) into (**black, black**)
 - Two cases: **restructuring** and **recoloring**
 - Restructuring resolves the problem locally, while recoloring may propagate it upward.
- Hereafter we assume v is a left child (swap right and left otherwise)

Deletion Case 1

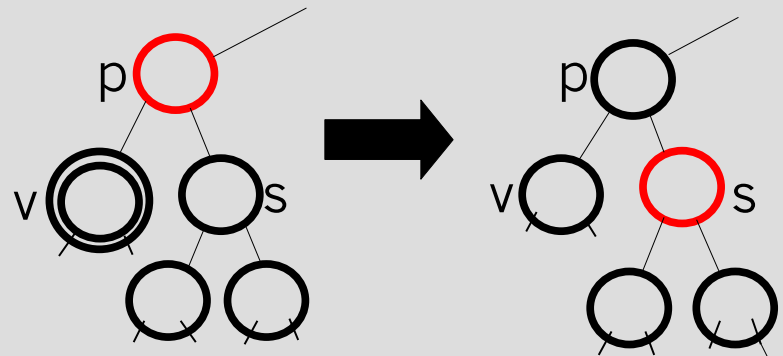
- Case 1
 - **v's sibling s is black and both children of s are black**

- Action

- `s.color := red`
- `v = p`

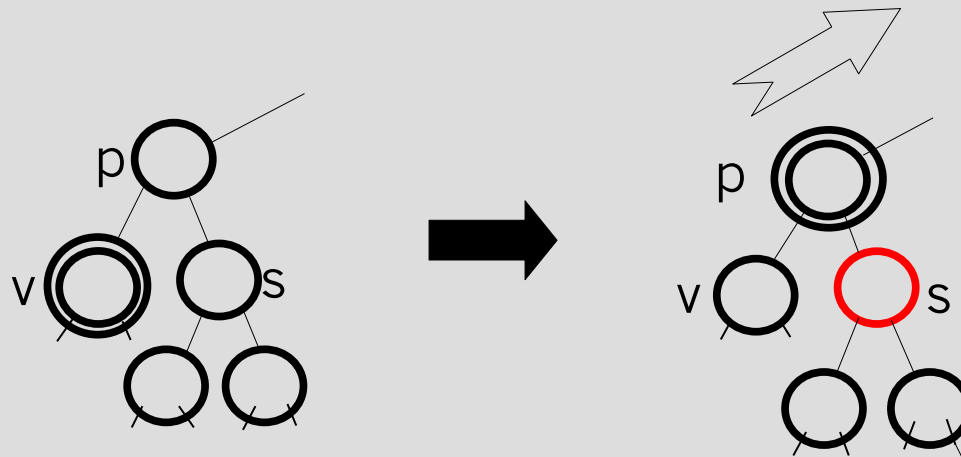
- Note

- We reduce the black depth of both subtrees of `p` by 1. Parent `p` becomes more black.



Deletion: Case 1

- If parent p becomes **double black**, continue upward.

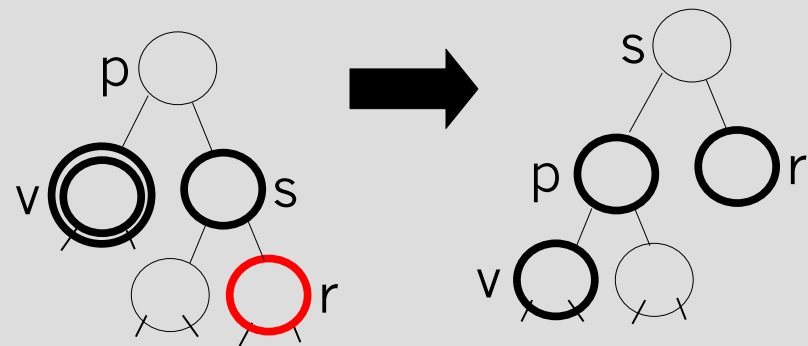


Deletion: Case 2

- Case 2
 - **v's sibling s is black and s's right child is red.**

- Action

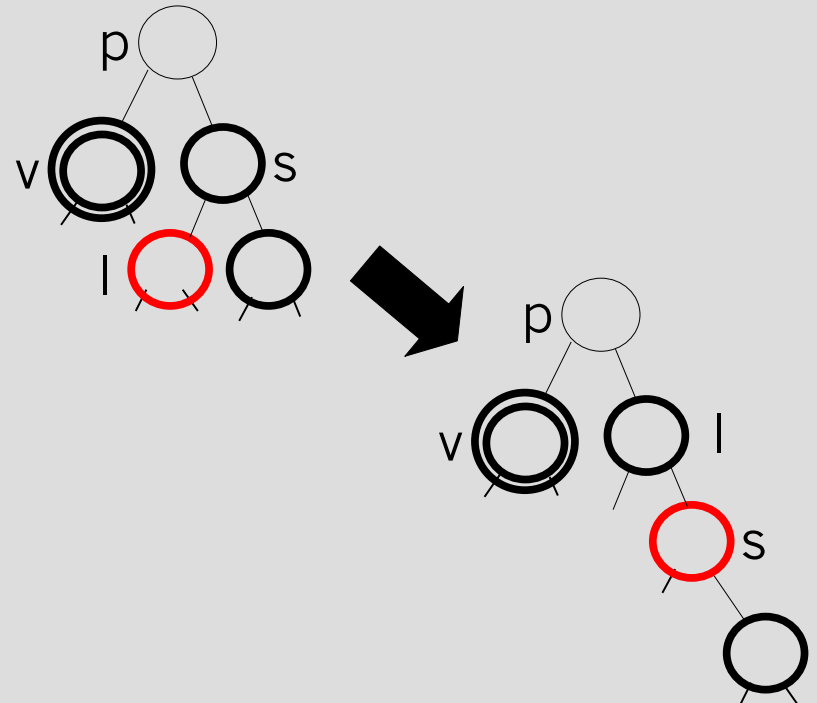
- `s.color = p.color`
- `p.color = black`
- `s.right.color = black`
- `LeftRotate(p)`



- Idea: Compensate the extra black ring of v by the red of r
- Note: Terminates after restructuring.

Deletion: Case 3

- Case 3
 - **v's sibling s is black, s's left child is red, and s's right child is black.**
- Idea: Reduce to case 2
- Action
 - `s.left.color = black`
 - `s.color = red`
 - `RightRotation(s)`
 - `s = p.right`
- Note:
 - This is now case 2

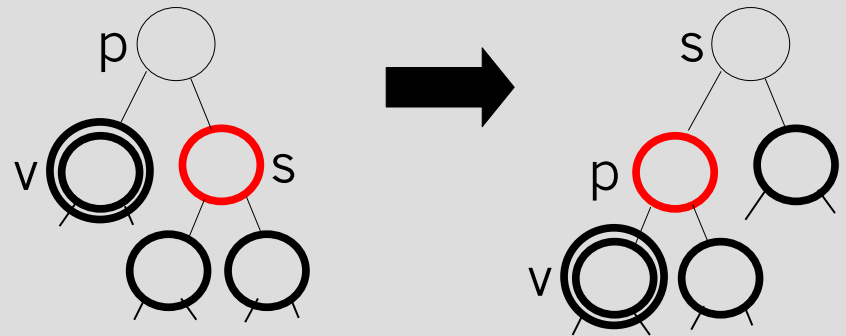


Deletion: Case 4

- Case 4
 - **v's sibling s is red**
- Idea: give v a black sibling

- Action

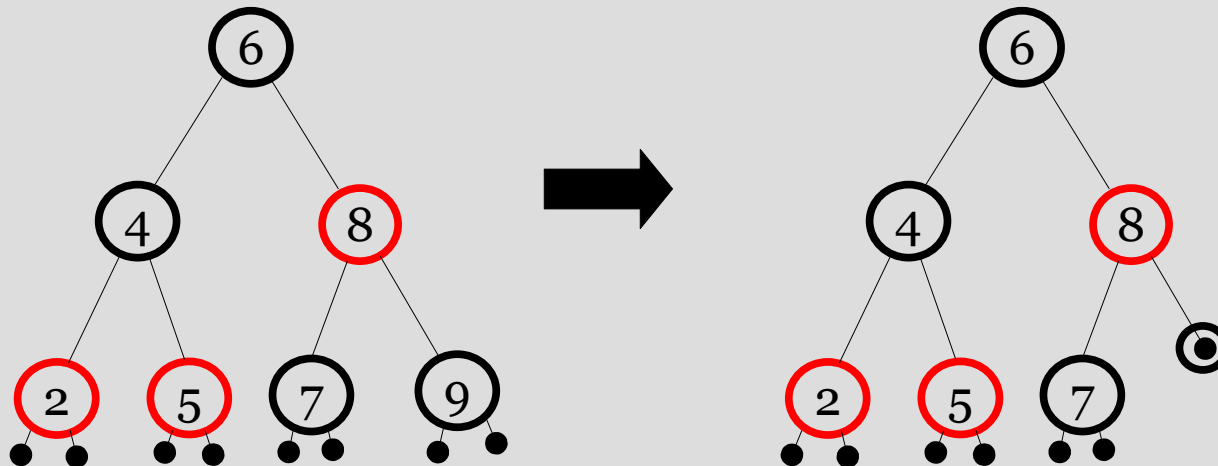
- `s.color = black`
- `p.color = red`
- `LeftRotation(p)`
- `s = p.right`



- Note

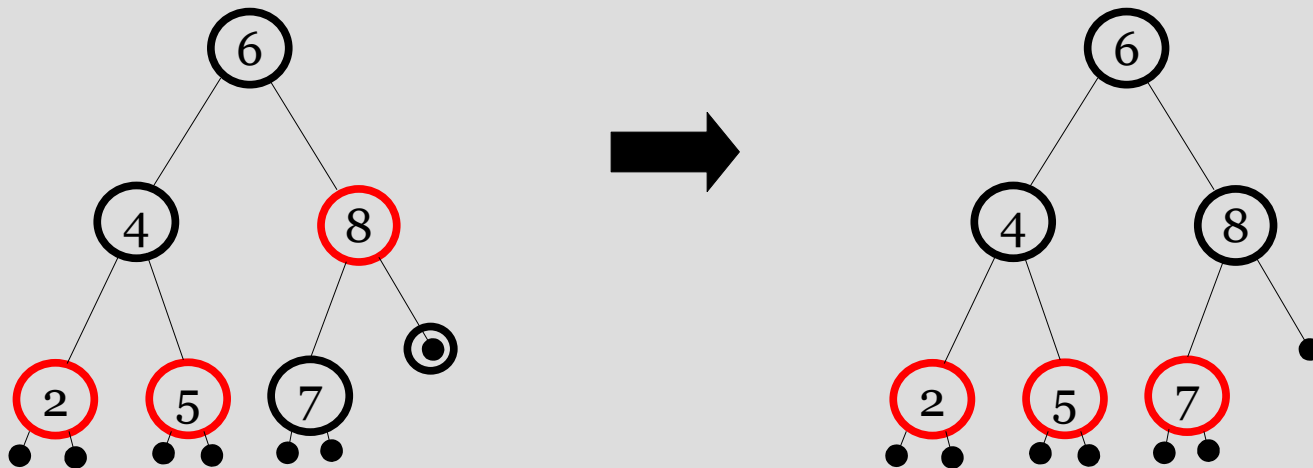
- This is now a case 1, 2, or 3

Delete 9

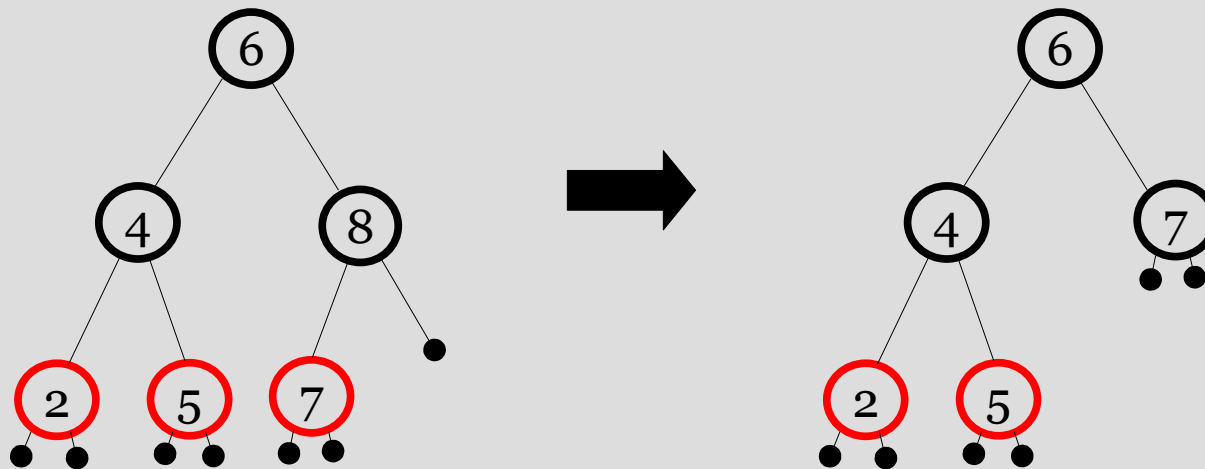


Delete 9/2

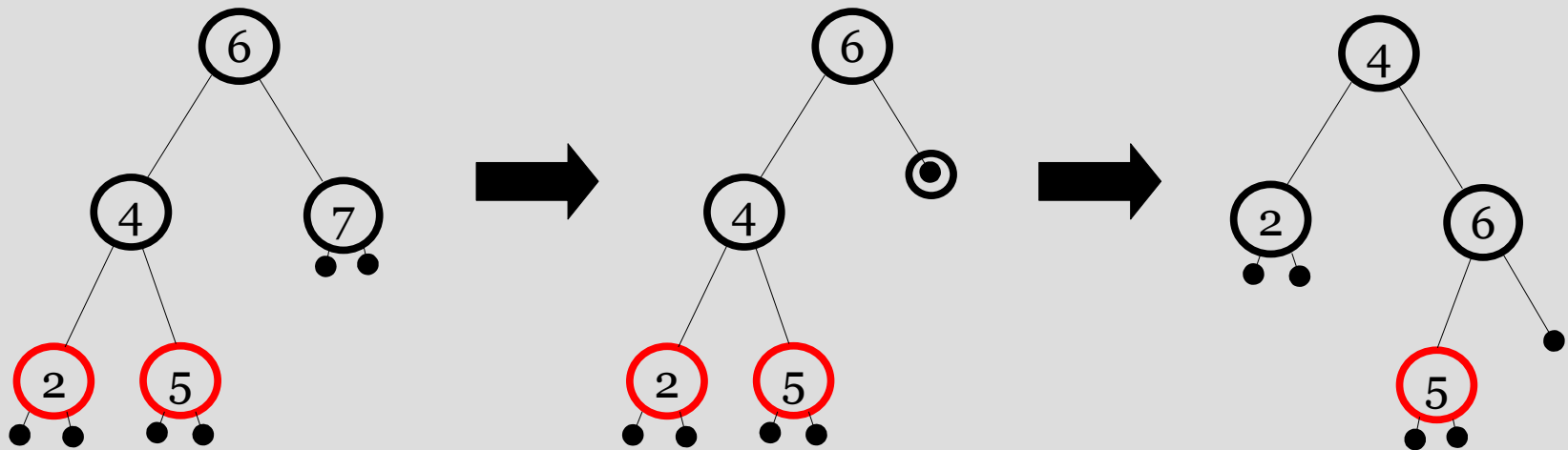
- Case 2 (sibling is black with black children) – recoloring



Delete 8



Delete 7: restructuring



How long does it take?

- Deletion in a RB-tree takes $O(\log n)$
 - Maximum three rotations and $O(\log n)$ recolorings

Suggested exercises

- Add left-rotate and right-rotate to the implementation of binary trees
- Implement a red-black search tree with the following functionalities:
 - (...), insert, delete

Suggested exercises/2

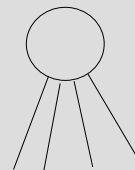
Using paper & pencil:

- draw the RB-trees after each of the following operations, starting from an empty tree:
 1. Insert 1,2,3,4,5,6,7,8,9,10,11,12
 2. Delete 12,11,10,9,8,7,6,5,4,3,2,1
- Try insertions and deletions at random

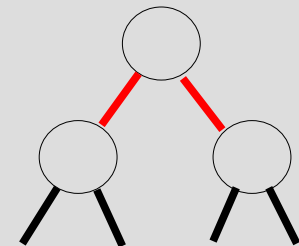
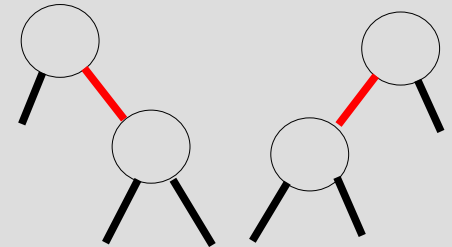
Other Balanced Trees

- Red-Black trees are related to 2-3-4 trees (non-binary)
- AVL-trees have simpler algorithms, but may perform a lot of rotations

2-3-4



Red-Black



Next Part

- Hashing