

Algorithms

Search Trees

Overview

- Binary Search Trees (Chapter 12 except 12.4)
- Balanced Binary Trees (Red-Black Trees) (Chapter 13)

Search Trees

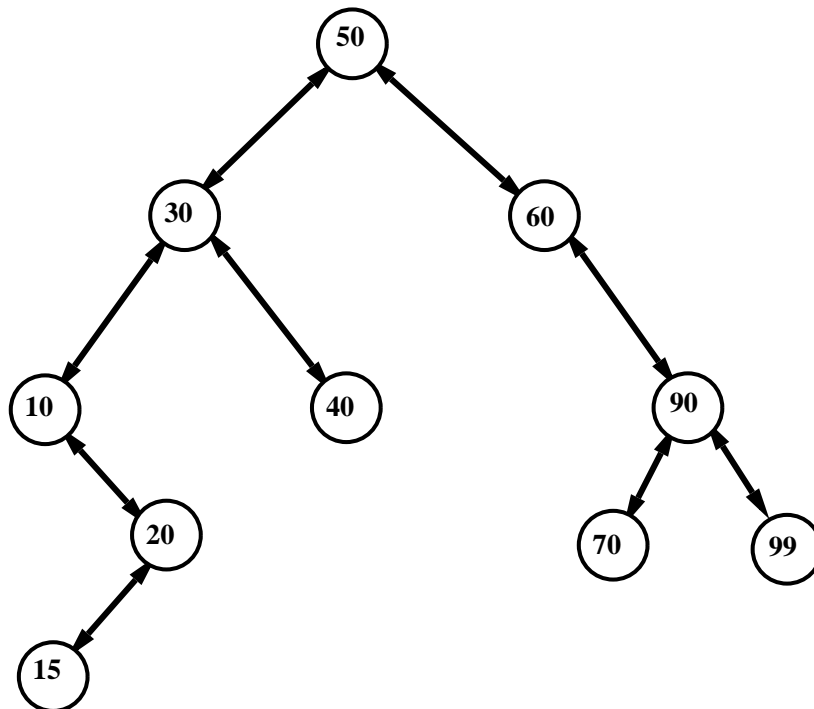
A data structure which can be used to represent disjoint sets of items with keys is called a *search tree* if the following operations are available.

- **access**(k, S): returns item with the key k from the set S ; if $k \notin S$, then returns **null**.
- **insert**(i, S): inserts item i into the search tree S , i not already in S .
- **delete**(i, S): deletes item i from S .
- **make**: returns new empty search tree.
- **join**(S, i, T): returns the search tree set $S \cup \{i\} \cup T$. S and T are destroyed. Every item in S has a key less than the key of i . Every item in T has a key greater than the key of i .
- **split**(i, SiT): splits the search tree SiT containing i into three search trees: s containing all items with keys less than i , $\{i\}$, and t containing all items with keys greater than i . The pair of search trees $[S, T]$ is returned. SiT is destroyed.

Each item has a unique key. Items other than roots cannot be accessed in $O(1)$ time.

Full Binary Search Trees

- Each node represents one item.
- Nodes in the left subtree of any node have keys less than the node itself.
- Nodes in the right subtree of any node have keys greater than the node itself.
- Pointers at any node x :
 - $\text{left}(x)$
 - $\text{right}(x)$
 - $\text{p}(x)$



Accessing Items

 $\text{access}(k, S)$

STEP 1: $x = \text{root of } S$.

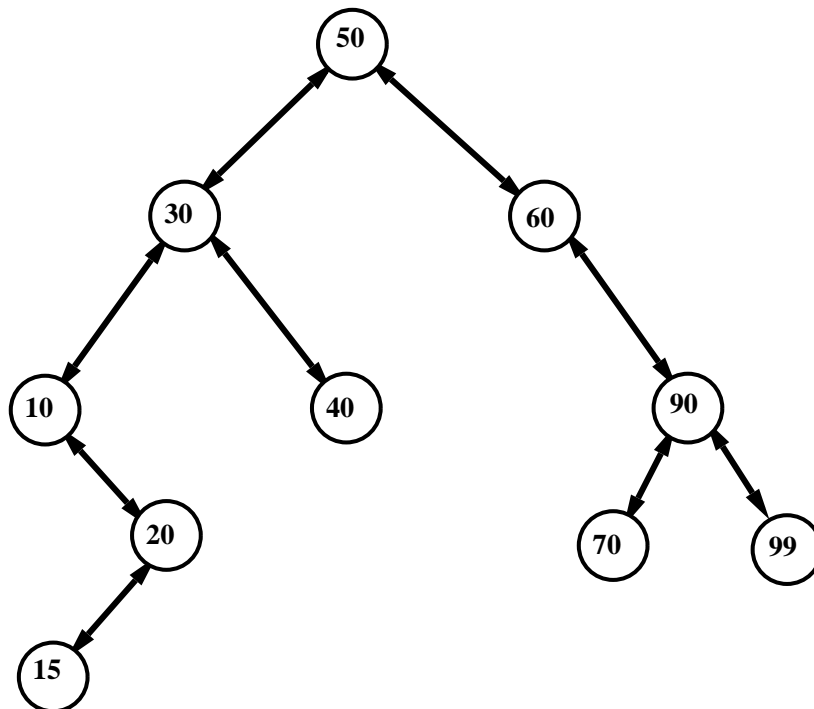
STEP 2: If $\text{key}(x) < k$, then $x = \text{right}(x)$. Go to **STEP 2**.

STEP 3: If $\text{key}(x) > k$, then $x = \text{left}(x)$. Go to **STEP 2**.

STEP 4: If $\text{key}(x) = k$, return x . **STOP**.

STEP 5: If $\text{key}(x) = \text{NIL}$, return NIL . **STOP**.

- $\text{access}(70, S)$



- Accessing item i takes time proportional to the depth of i in the search tree: $O(n)$ since a binary tree with n items can have depth $n - 1$.

Inserting Items

$$\text{insert}(i, S)$$

STEP 1: $k = \text{key}(i)$. $x = \text{root of } S$.

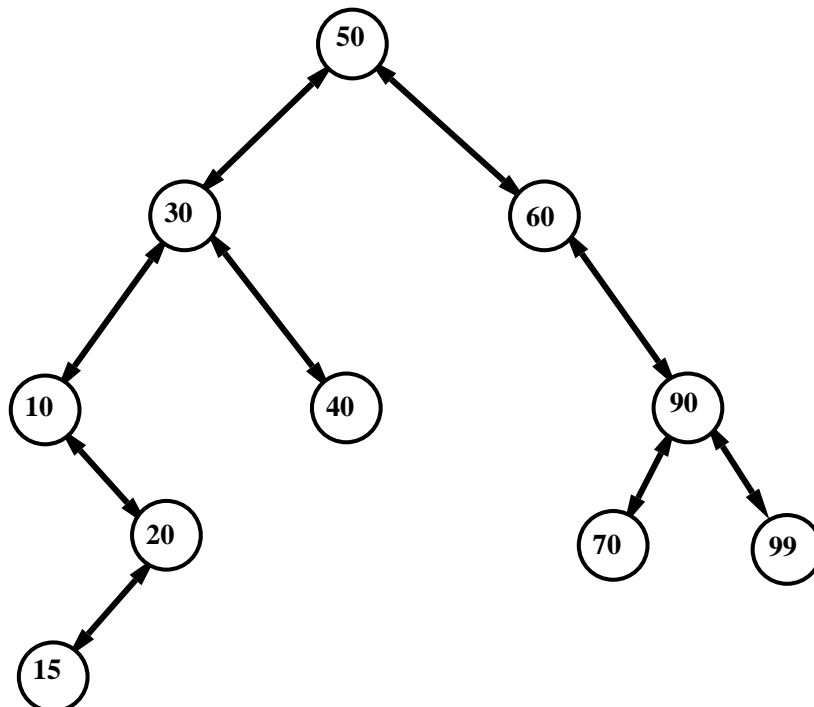
STEP 2: $\text{key}(x) < k$ and $\text{right}(x) = \text{NIL}$: let i be the right son of x . **STOP**.

STEP 3: $\text{key}(x) < k$ and $\text{right}(x) \neq \text{NIL}$: $x = \text{right}(x)$. Go to **STEP 2**.

STEP 4: $\text{key}(x) > k$ and $\text{left}(x) = \text{NIL}$: let i be the left son of x . **STOP**.

STEP 5: $\text{key}(x) > k$ and $\text{left}(x) \neq \text{NIL}$: $x = \text{left}(x)$. Go to **STEP 2**.

- $\text{insert}(80, S)$



- inserting i takes time proportional to the depth of i after the insertion: $O(n)$

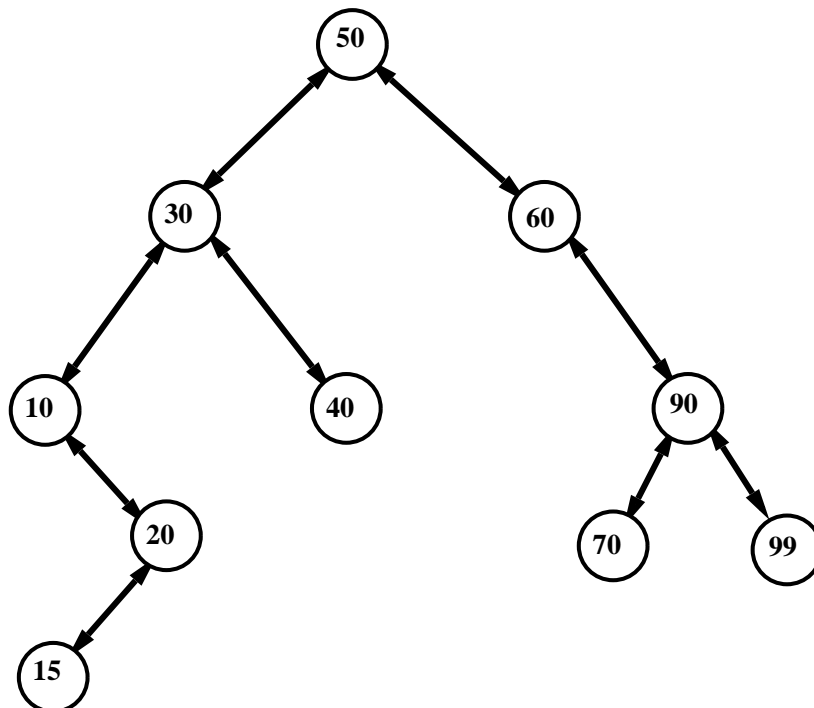
Deleting Items

STEP 1: Begin at i .

STEP 2: If i has a NIL child, replace i by the other child (which can be NIL). **STOP**

STEP 3: Follow right sons of $\text{left}(i)$ until reaching a node j with NIL as right child. Swap i and j . Replace i by its left child (can be NIL). **STOP**.

- `delete(20)`
- `delete(30)`



- proportional to the depth of i when i has a NIL child (i has to be accessed).
- proportional to the depth of i + depth of i 's left son.

In both cases $O(n)$ time is required.

Joining Trees

`join(S, i, T)`

- Make the roots of S and T the left and right children of i , respectively.
- Joining requires $O(1)$ time.

Splitting Trees`split(i, SiT)`

STEP 1: Access i .

STEP 2: Let S be the left subtree of i , and let T be the right subtree of i .

STEP 3: Stop if i is the root of the original tree SiT .

STEP 4: If i is a left child, then $T = \text{join}(T, p(i)i, r(p(i)i))$.

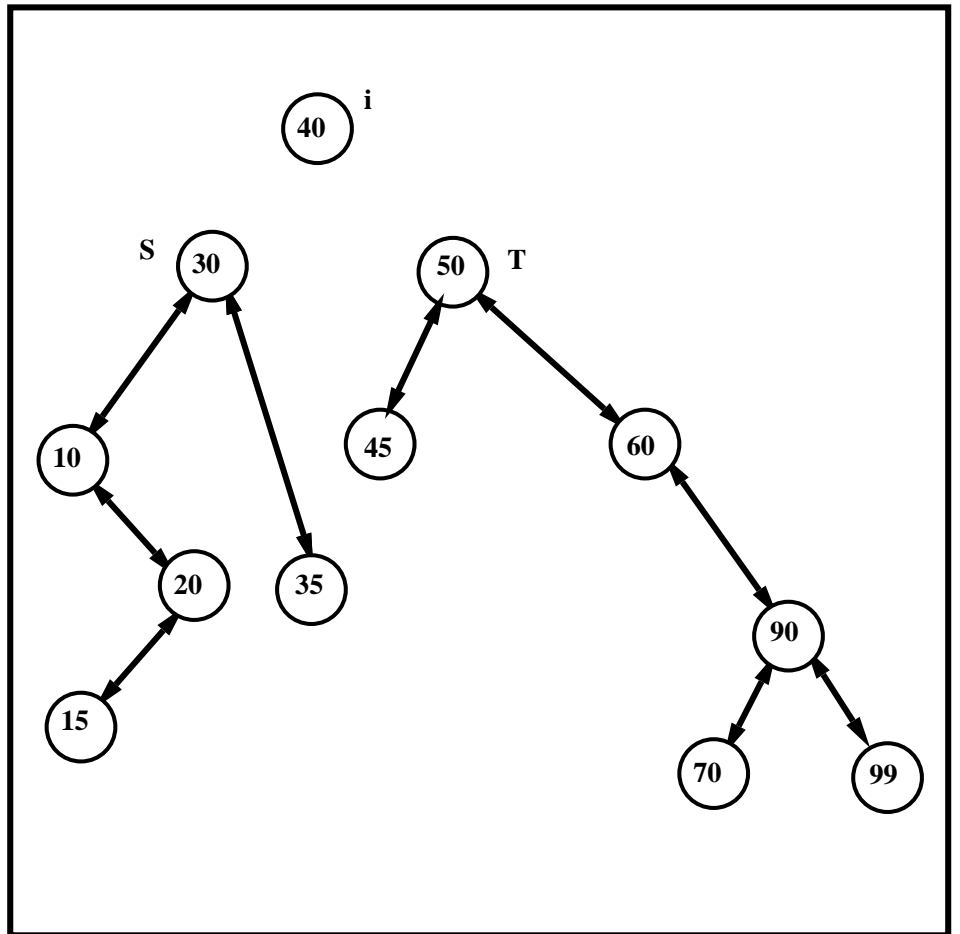
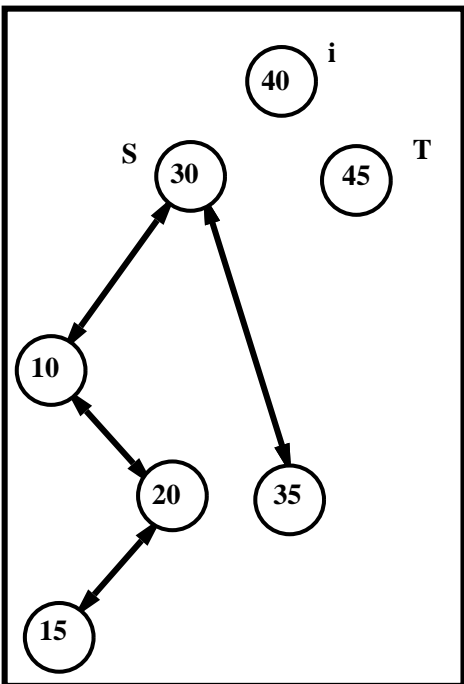
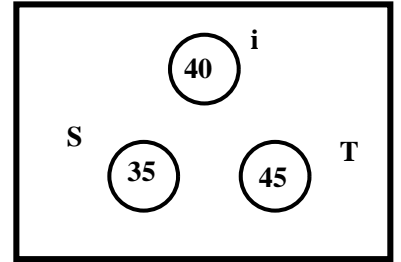
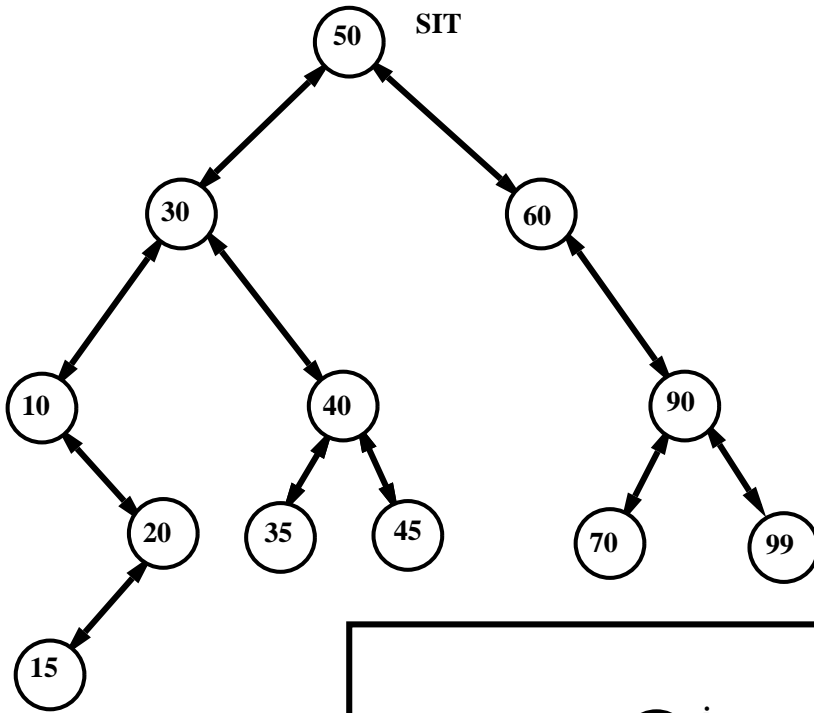
STEP 5: If i is a right child, then $S = \text{join}(l(p(i)), p(i)i, S)$.

STEP 6: Let $p(i)$ act as i and go to **STEP 3**.

- Splitting is proportional to the depth of i : $O(n)$

Splitting Trees

split(40,SIT)

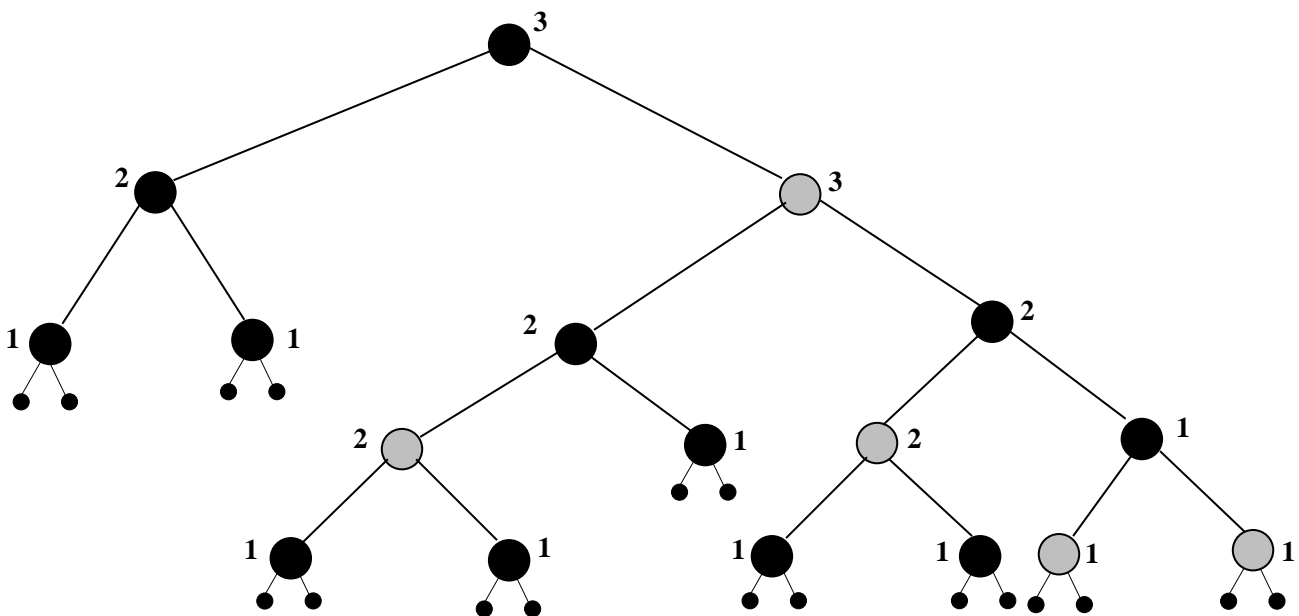


Balanced Binary Search Trees

- How to carry out tree operations while keeping the depth of the tree small?
- Is it possible to reduce maximal depth from $O(n)$ to for example $O(\log n)$?

Balanced Binary Search Trees

- Every node is either black or red, the root is black.
- All leaves (NIL nodes) are black.
- If a node is red, then both its children are black.
- Every path from a given node to each of the leaves in its subtree has the same number of black nodes.



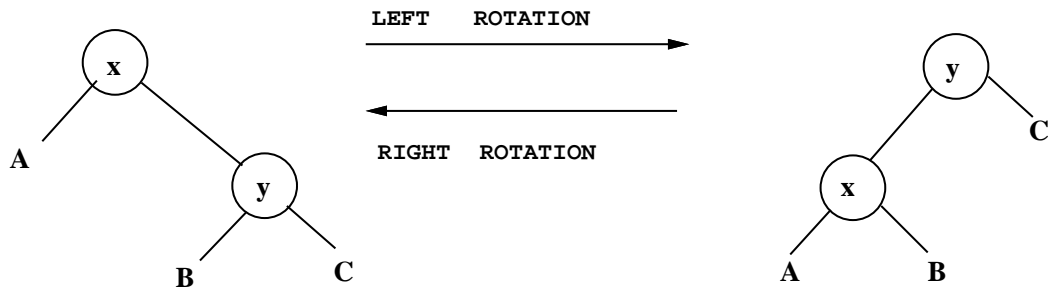
Height of Balanced Binary Search Trees

- A red-black tree with n internal (non-leaf) nodes has height at most $2 \log(n + 1)$.
- A subtree rooted at a node x with black height $bh(x)$ has at least $2^{bh(x)} - 1$ internal nodes.
- Proof by induction on the height of x .
- If $h(x) = 0$ then x is an (external) leaf, and the number of internal nodes is 0. And $2^{bh(x)} - 1 = 2^0 - 1 = 0$.
- Assume that x has a positive height. Each of the two children of x has height one less. Furthermore, their black height is at least $bh(x) - 1$. Therefore, by the inductive hypothesis, the number of internal nodes in the subtree rooted at x is at least

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$$

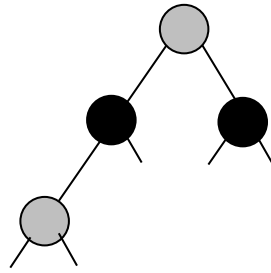
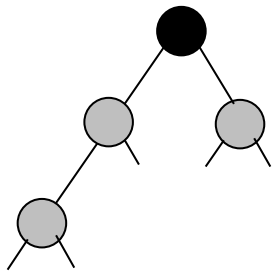
- Let h be the height of the tree.
- Every red node has two black sons. Number of red nodes on any path from the root r cannot be more than $h/2$. Therefore the number of black nodes must be at least $h/2$. Therefore $bh(r) \geq h/2$. Therefore $n \geq 2^{h/2} - 1$.
- Or $n+1 \geq 2^{h/2}$, or $\log(n+1) \geq h/2$, or $2 \log(n+1) \geq h$.

Rebalancing Balanced Binary Search Trees

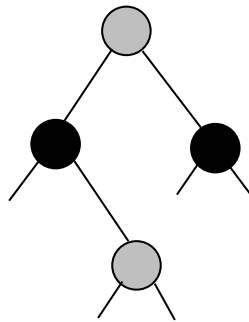
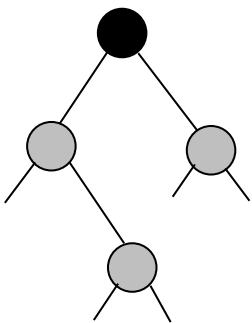


- Each of these operations can be done in $O(1)$ time.

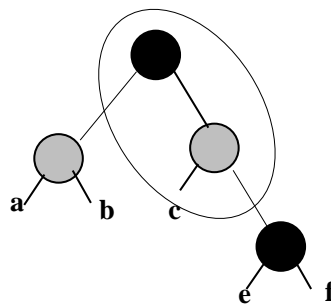
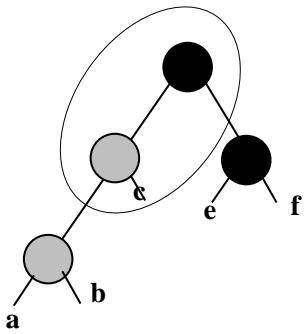
Rebalancing after Insertion



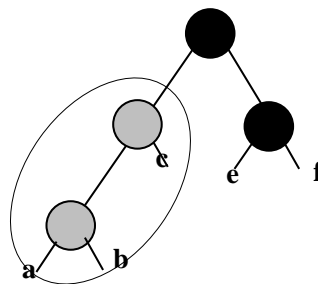
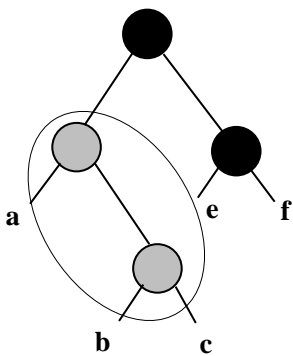
problem propagates up the tree
if the root is reached it is colored black



problem propagates up the tree
if the root is reached it is colored black



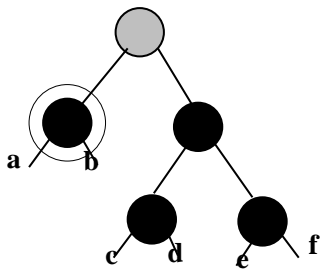
rotation and some color changes
termination



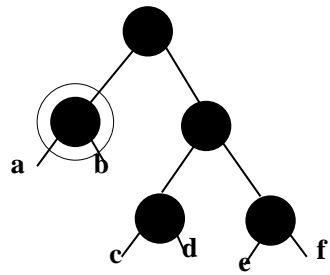
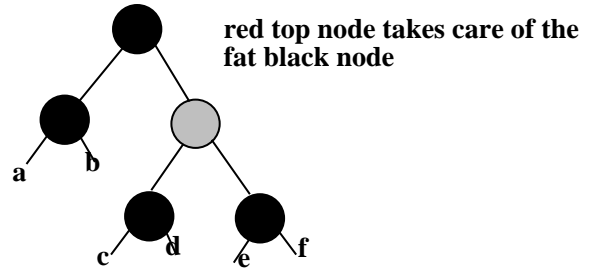
rotation and case above

Rebalancing after Deletion

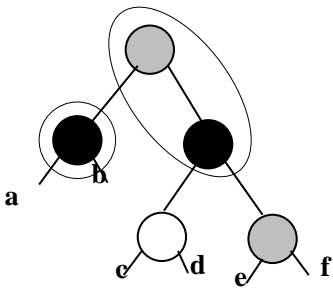
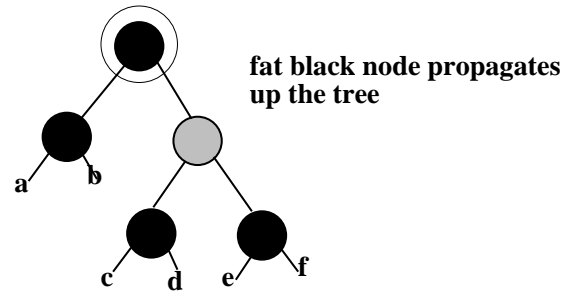
- If the removed node was red, no problems.
- If the removed node was black while its left son x was red, make x black.
- If the removed node was black while its left son x was black?



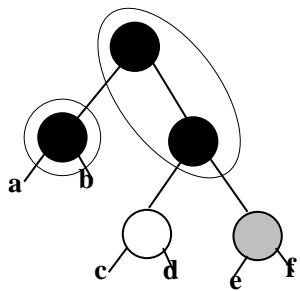
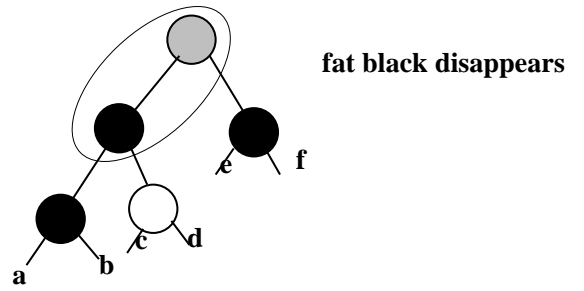
case 2a



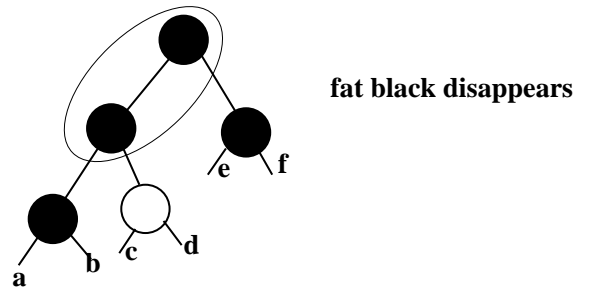
case 2b



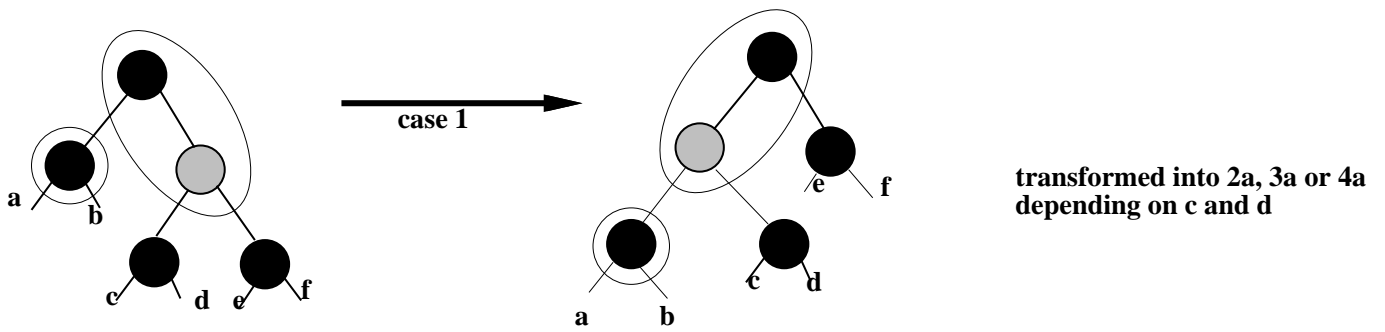
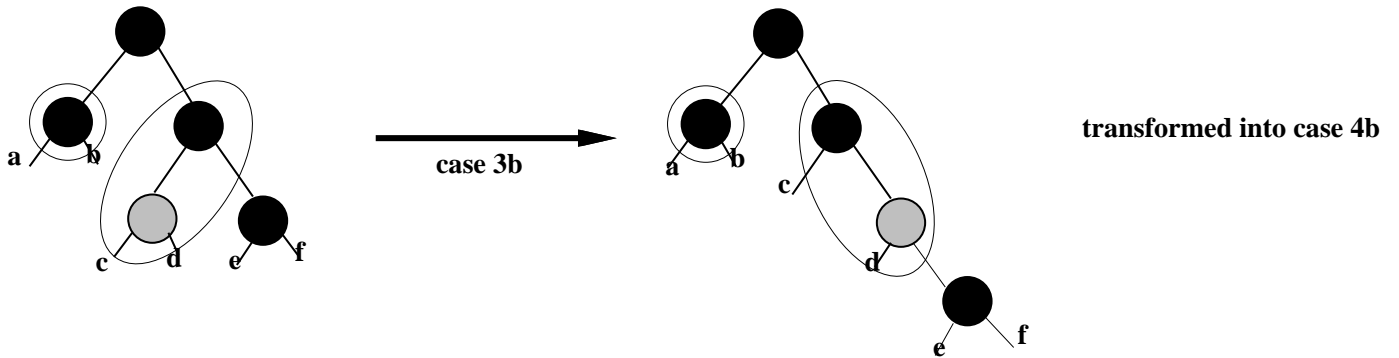
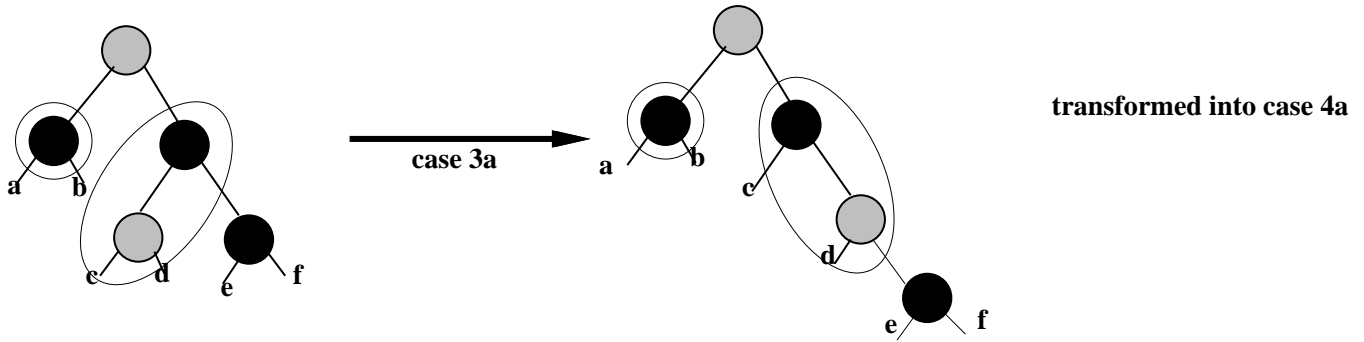
case 4a



case 4b

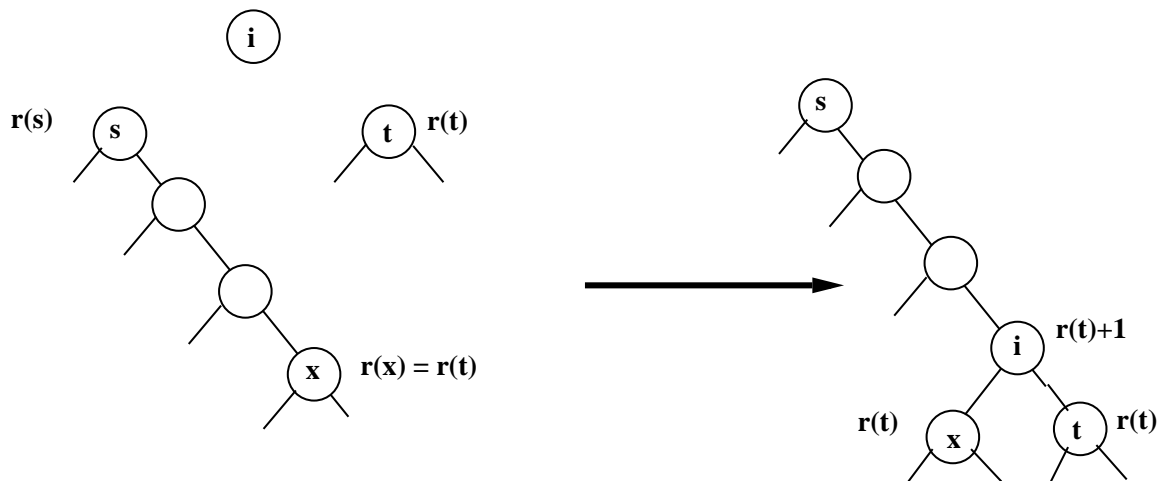


Rebalancing after Deletion -cont.



Joining Balanced Trees

- In order to join a balanced tree s , item i , and a balanced tree t , $bh(s)$ and $bh(t)$ is compared.
- If $bh(s) \geq bh(t)$, right pointers in s are followed until a black node x with $bh(x) = bh(t)$ is reached.
- x and its subtree is replaced by i , x (and its descendants) is made left subtree of i , and t is made the right subtree of i
- i is set to be red.
- rebalancing beginning from i is then carried out as if i was inserted.
- The case $bh(s) < bh(t)$ is symmetric.
- Each join is proportional to the sum of black heights of trees. Hence join takes at most $O(\log n)$ time.



Splitting Balanced Trees

- As splitting in binary search trees. But joins as in balanced binary trees.
- Split requires $O(\log n)$ time.