COMP9024: Data Structures and Algorithms

Self-Balancing Search Trees

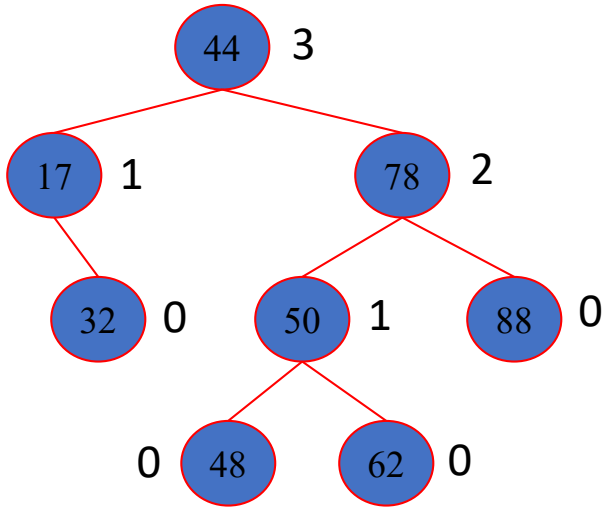
Contents

- AVL trees
- Splay trees
- 2-4 trees
- Red-black trees

AVL Trees

AVL Tree Definition

- **AVL trees are balanced.**
- An AVL Tree is a ***binary search tree*** such that for every internal node v of T, the *heights of the children of v can differ by at most 1*.
- We call it height difference constraint.



An example of an AVL tree where the heights are shown next to the nodes:

Height of an AVL Tree

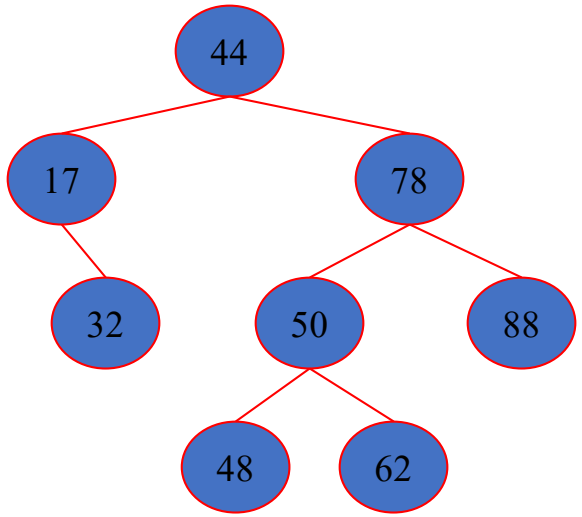
- **Fact**: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of nodes of an AVL tree of height h.
- We easily see that n(0) = 1 and n(1) = 2
- For n > 1, an AVL tree of height h contains the root node, one AVL subtree of height h-1 and another of height h-2.
- That is, n(h) = 1 + n(h-1) + n(h-2)
- Knowing n(h-1) > n(h-2), we get n(h) > 2n(h-2). So

  n(h) > 2n(h-2), n(h) > 4n(h-4), n(h) > 8n(n-6), … (by induction),

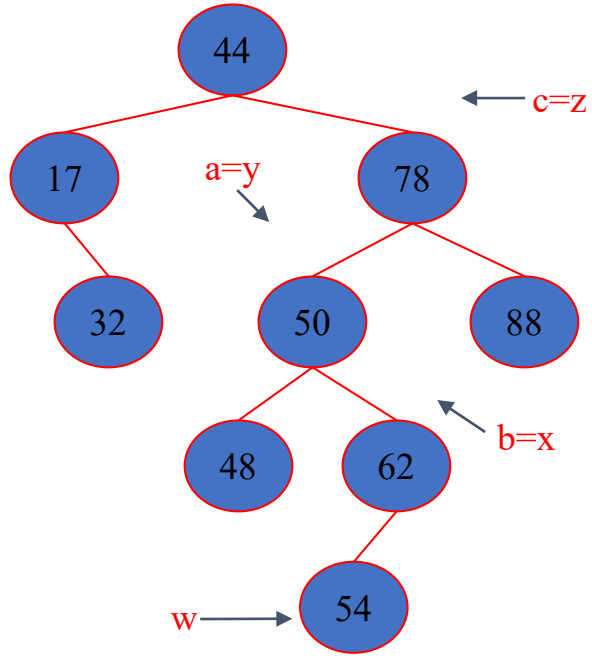
  $n(h) > 2^i n(h-2i)$
- Let h-2i=0. We have i=h/2 and $n(h) > 2^{h/2}$
- Taking logarithms: h < 2log n(h)
- Thus the height of an AVL tree is O(log n)

Insertion in an AVL Tree

- Insertion is as in a binary search tree
- Example: insert 54



before insertion



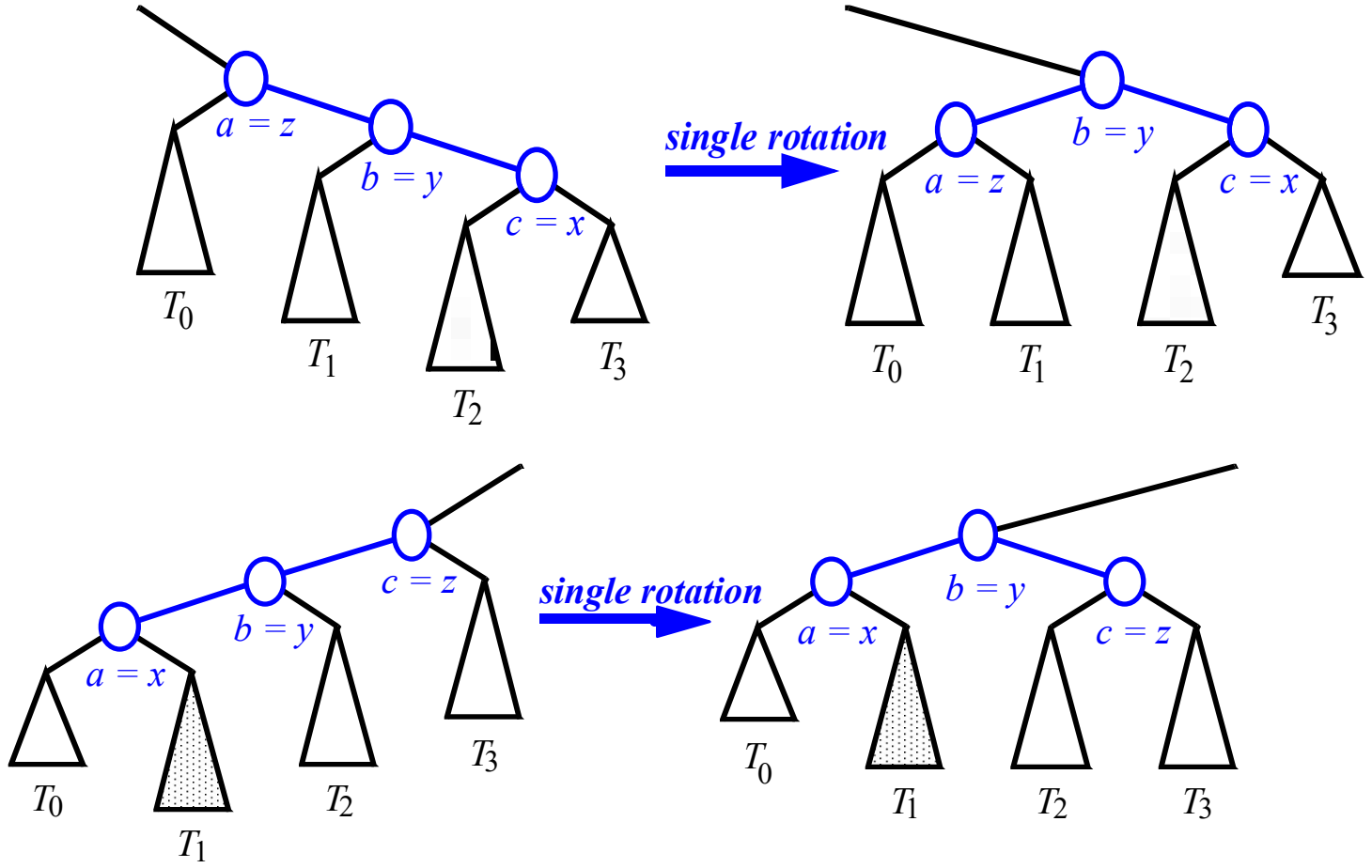
after insertion

Trinode Restructuring (1/4)

- After inserting a new entry into an AVL tree, the height difference property may be violated.
- In order to restore the height difference property, we perform the trinode restructuring:
  - Find the first ancestor *z* of the new node that violates the height difference constraint along the path from the new node to the root.
    - Find a child *y* of *z* with the larger height, and find a child *x* of *y* with the larger height .
    - Rename *x*, *y* and *z* as *a*, *b* and c, respectively, in in-order traversal order.
    - Perform trinode restructuring on *a*, *b* and *c* so that *b* becomes of the new root of the subtree previously rooted at *z*.

Trinode Restructuring (Single Rotations) (2/4)

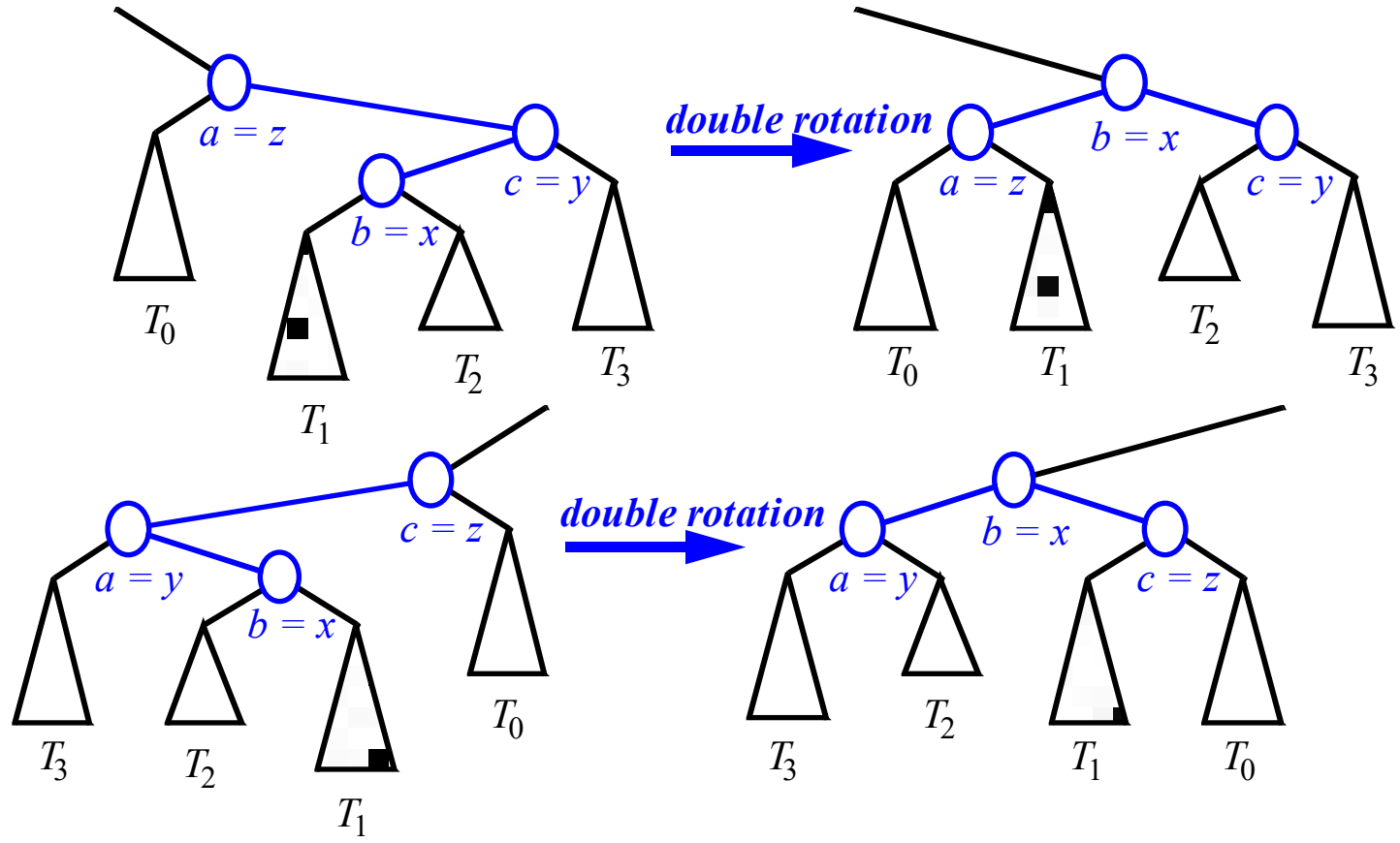
- Single Rotations:



8

Trinode Restructuring (Double Rotations) (3/4)

- double rotations:

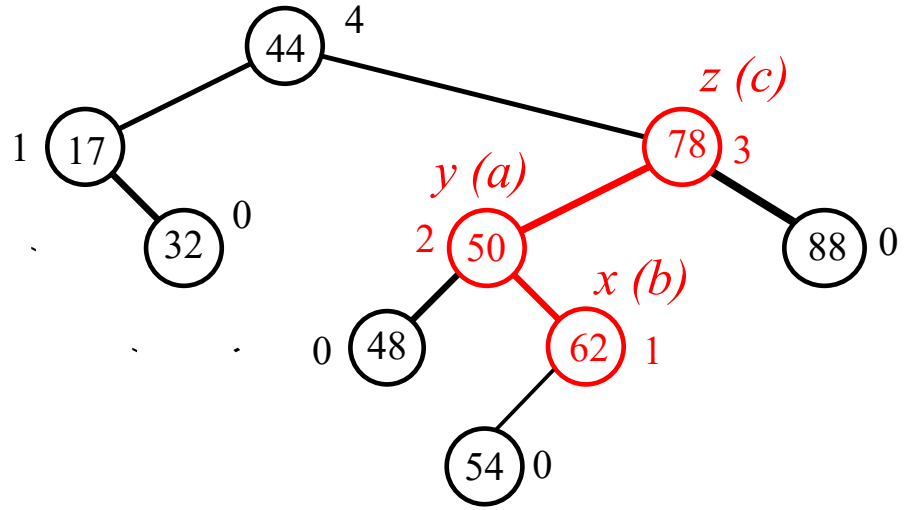


Trinode Restructuring (Double Rotations) (4/4)

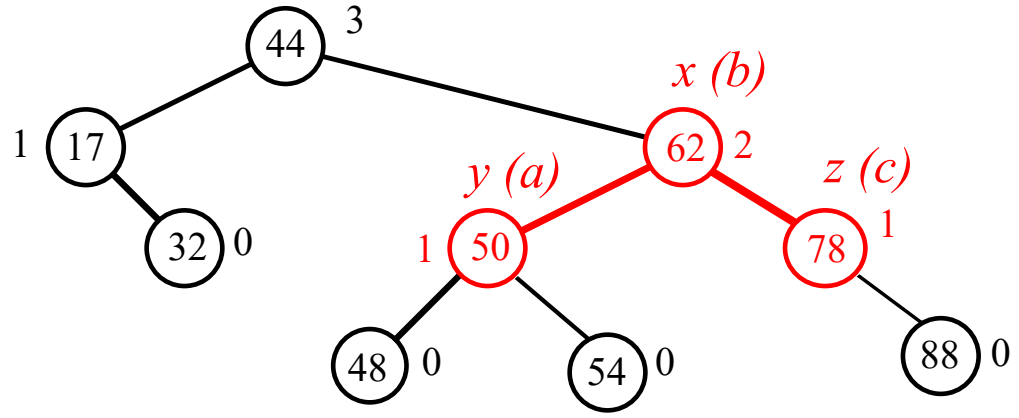
- Inserting a new node into an AVL tree may increase the height of the tree by one.
- The objective of a trinode restructuring is to reduce the height of the subtree rooted at node z by one while maintaining the binary tree property.

Insertion Example, continued

unbalanced

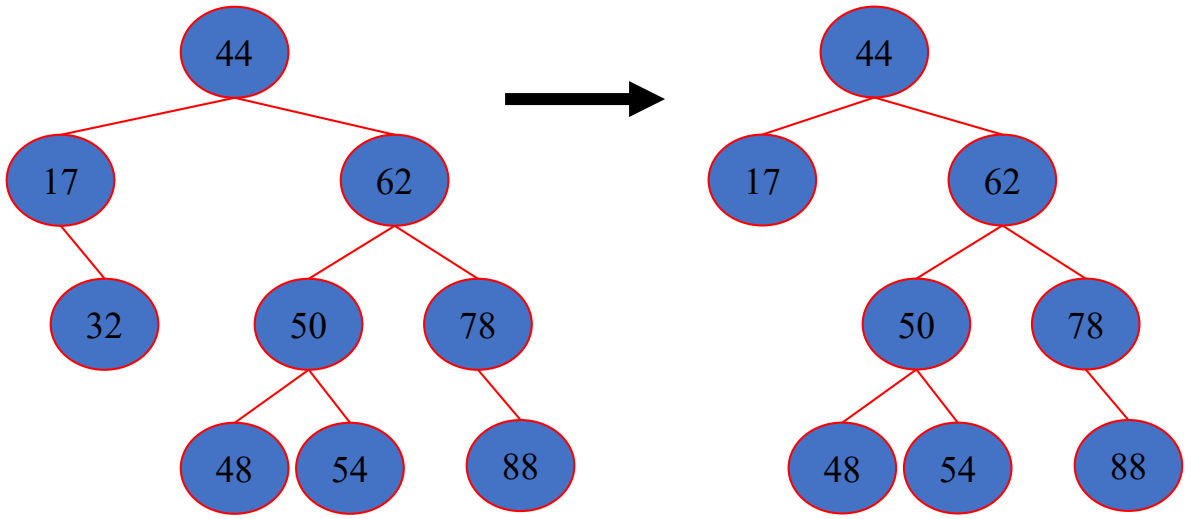


balanced



Removal in an AVL Tree

- Removal begins as in a binary search tree, which means the node removed will become an empty external node. Its parent, w, may cause an imbalance.
- Example: delete 32

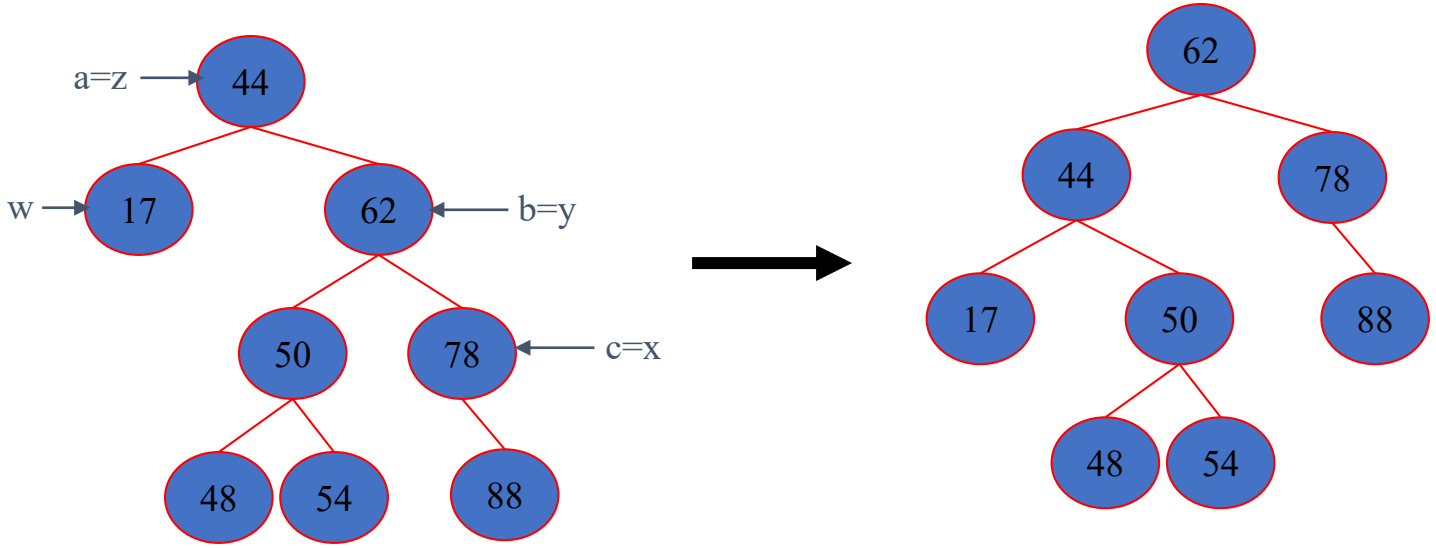


before deletion of 32

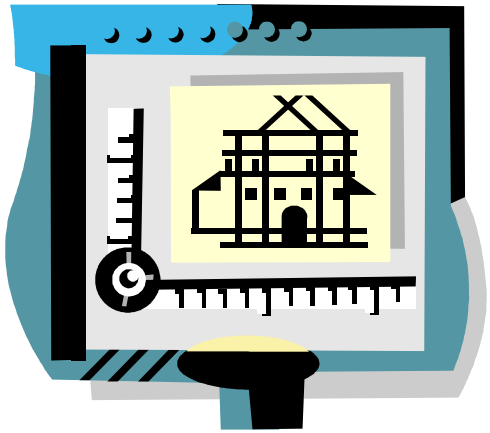
after deletion

Rebalancing after a Removal

- Let *z* be the first unbalanced node encountered while travelling up the tree from w. Also, let y be the child of z with the larger height, and let x be the child of y with the larger height.
- Perform trinode restructing on x, y and z to restore balance at z.
- As this restructuring may upset the balance of another node higher in the tree, we must continue checking for balance until the root of T is reached.



Running Times for AVL Trees

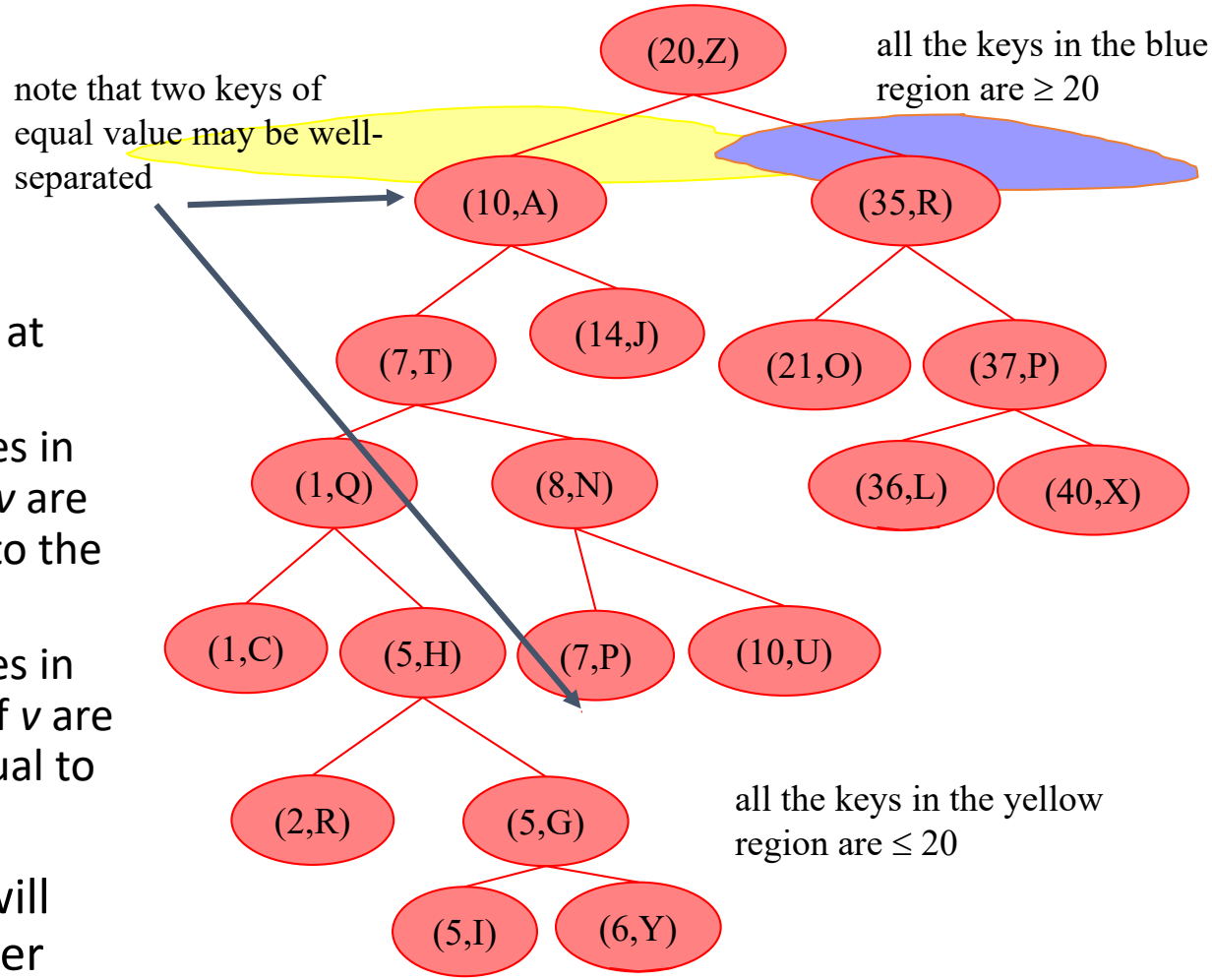


- a single restructure is O(1)
  - using a linked-structure binary tree
- find is O(log n)
  - height of tree is O(log n), no restructures needed
- insert is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- delete is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)

Splay Trees

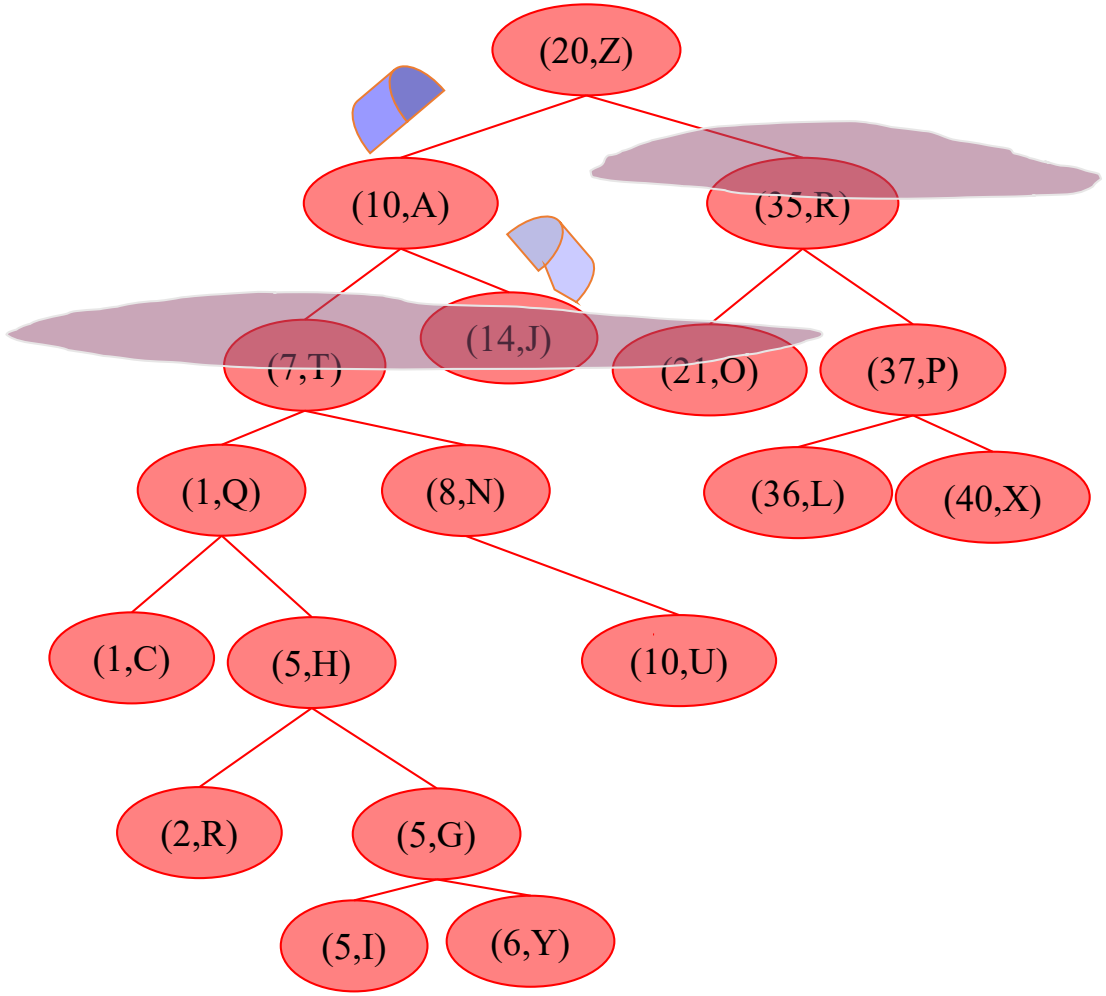
Splay Trees are Binary Search Trees

- BST Rules:
  - entries stored only at internal nodes
  - keys stored at nodes in the left subtree of *v* are less than or equal to the key stored at *v*
  - keys stored at nodes in the right subtree of *v* are greater than or equal to the key stored at *v*
- An inorder traversal will return the keys in order



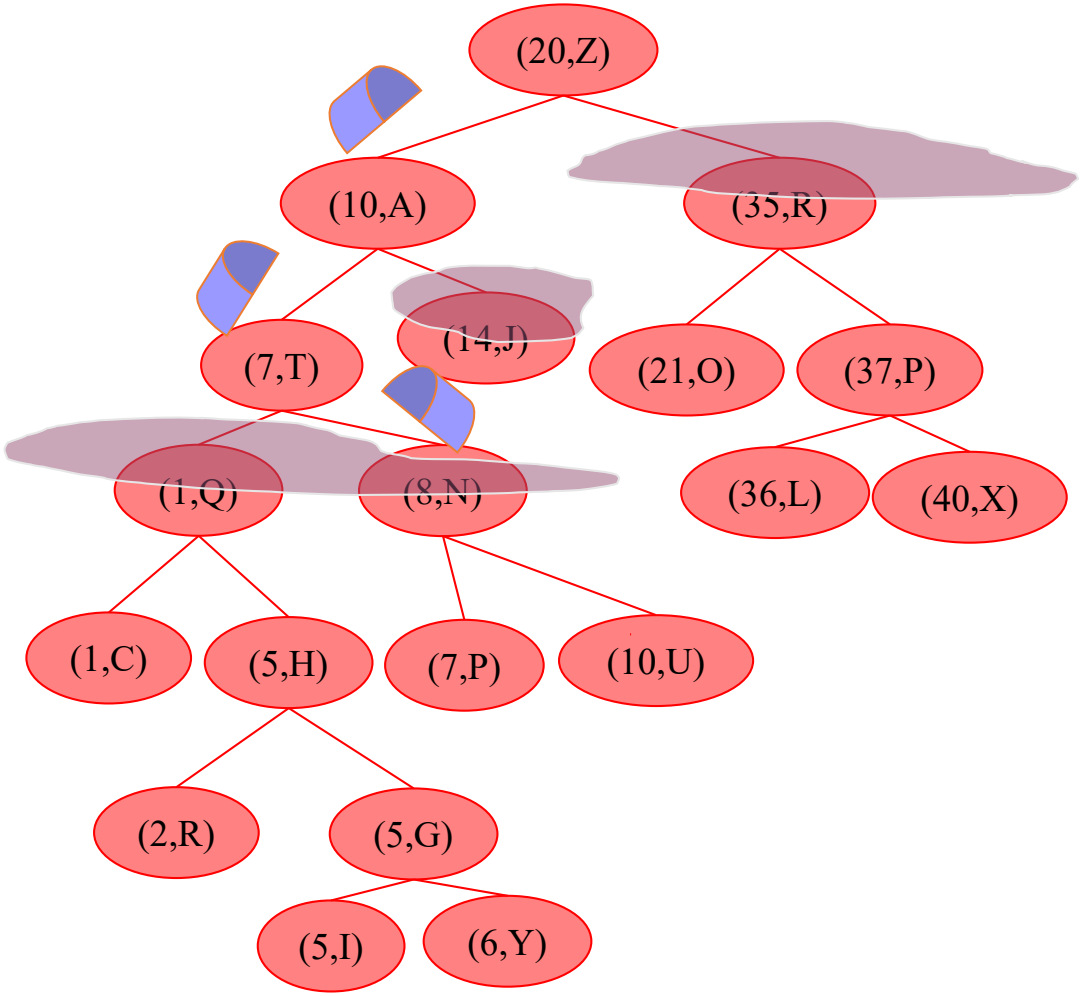
Searching in a Splay Tree: Starts the Same as in a BST

- Search proceeds down the tree to find the item or an empty node.
- Example: Search for an item with key 11.



Example Searching in a BST, continued

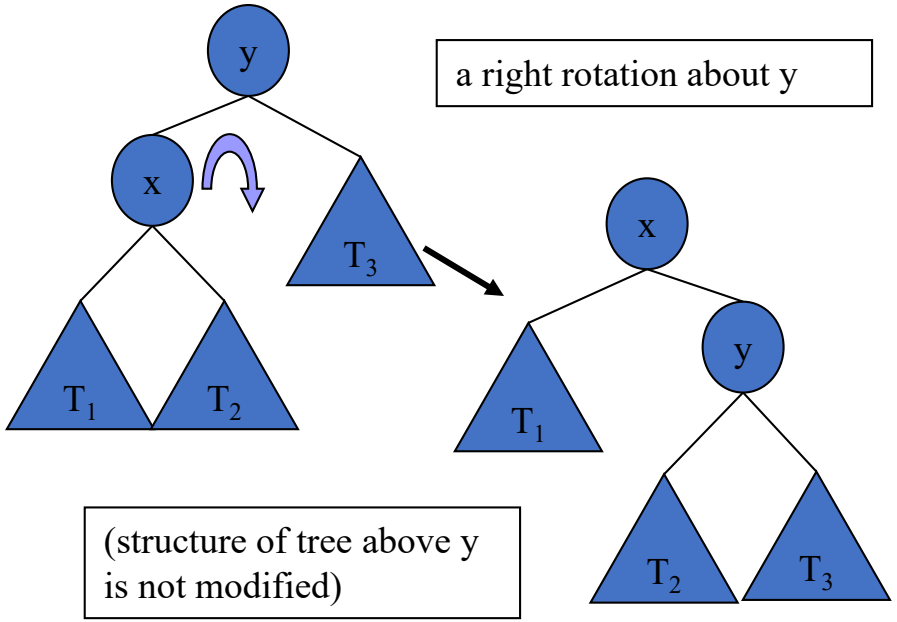
- search for key 8, ends at a non-empty node.



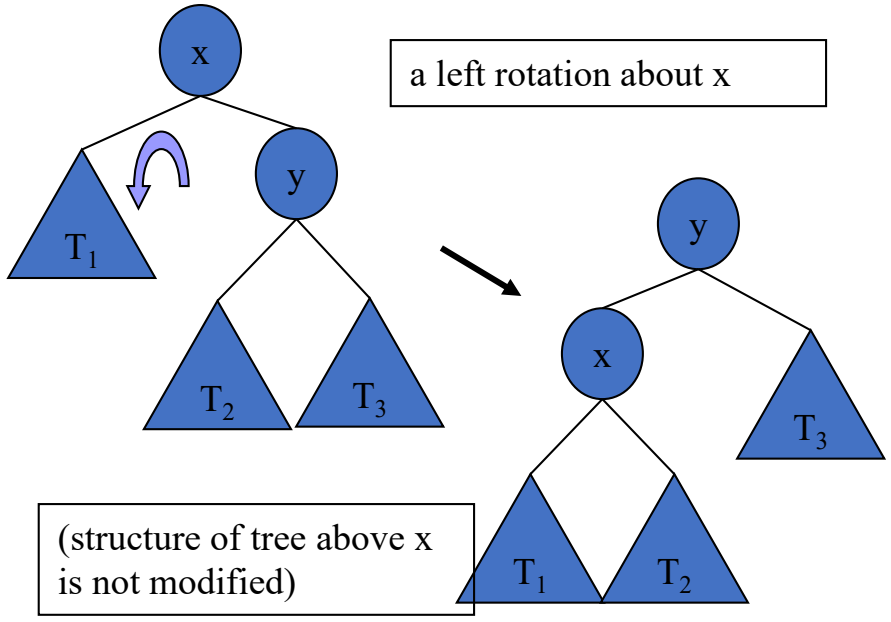
Splay Trees do Rotations after Every Operation (Even Search)

- new operation: ***splay***
  - splaying moves a node to the root using rotations

- right rotation
  - makes the left child x of a node y into y's parent; y becomes the right child of x

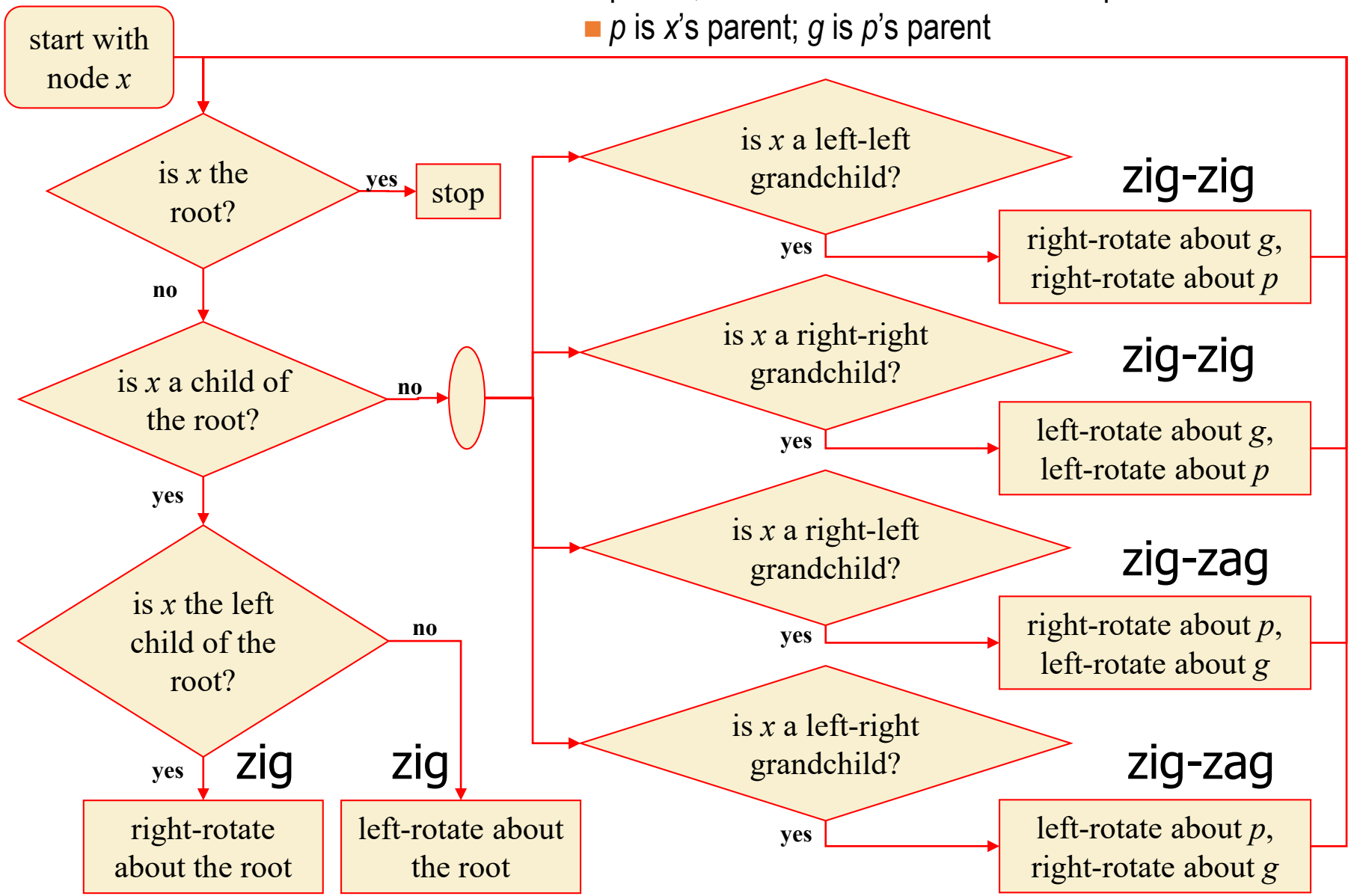


- left rotation
  - makes the right child y of a node x into x's parent; x becomes the left child of y

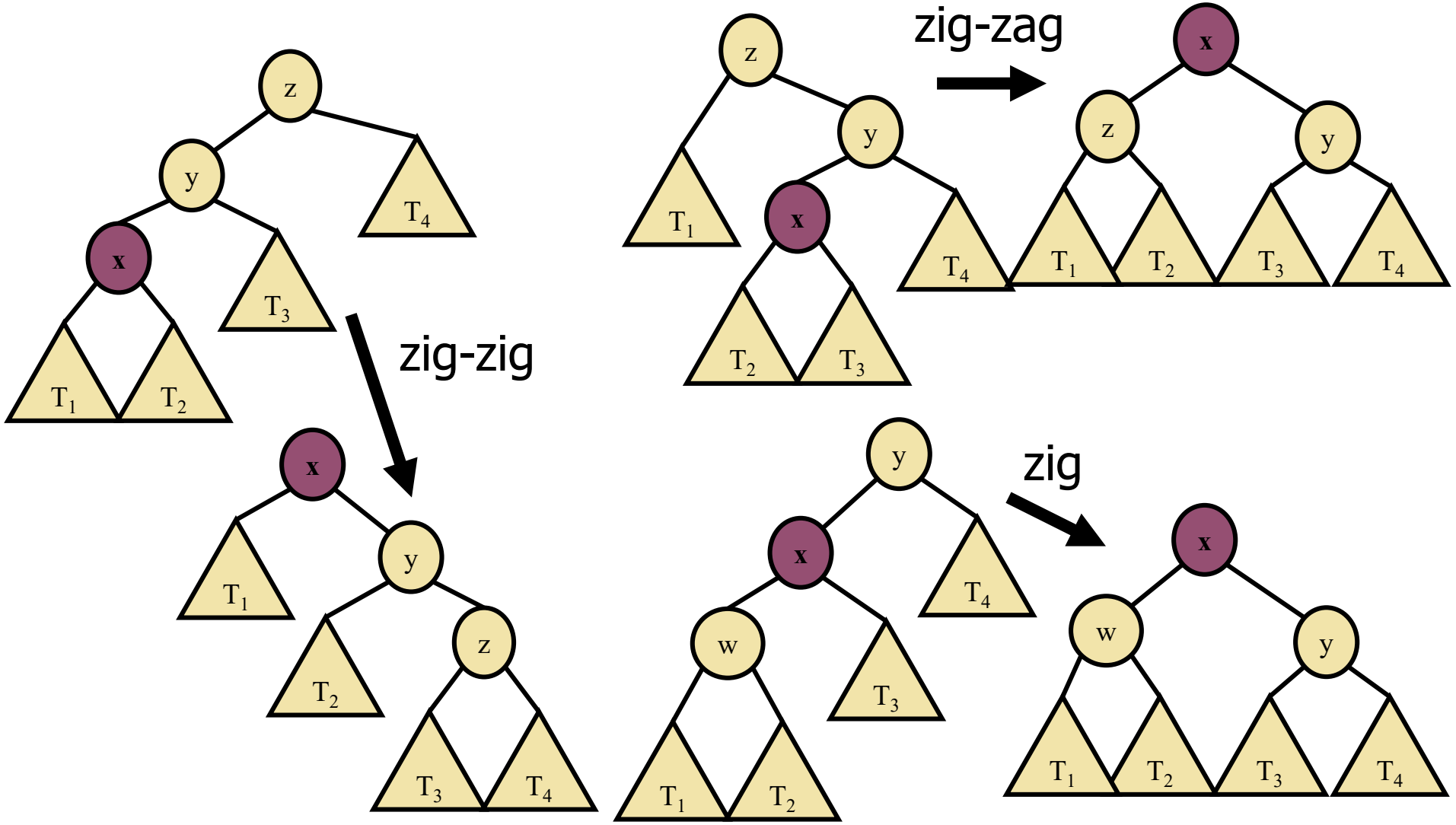


Splaying:

- "*x* is a left-left grandchild" means *x* is a left child of its parent, which is itself a left child of its parent
- *p* is *x*'s parent; *g* is *p*'s parent

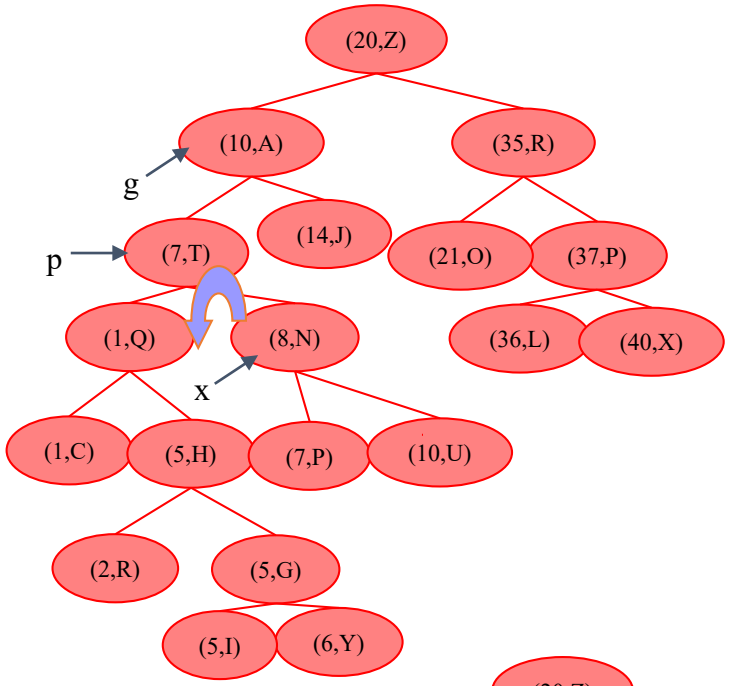


Visualizing the Splaying Cases

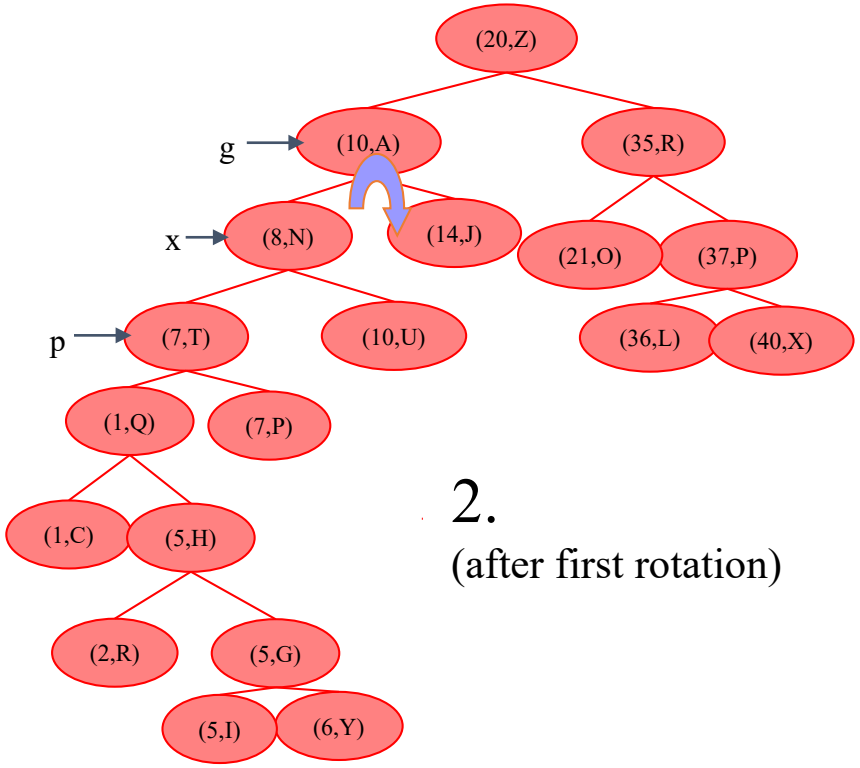


Splaying Example

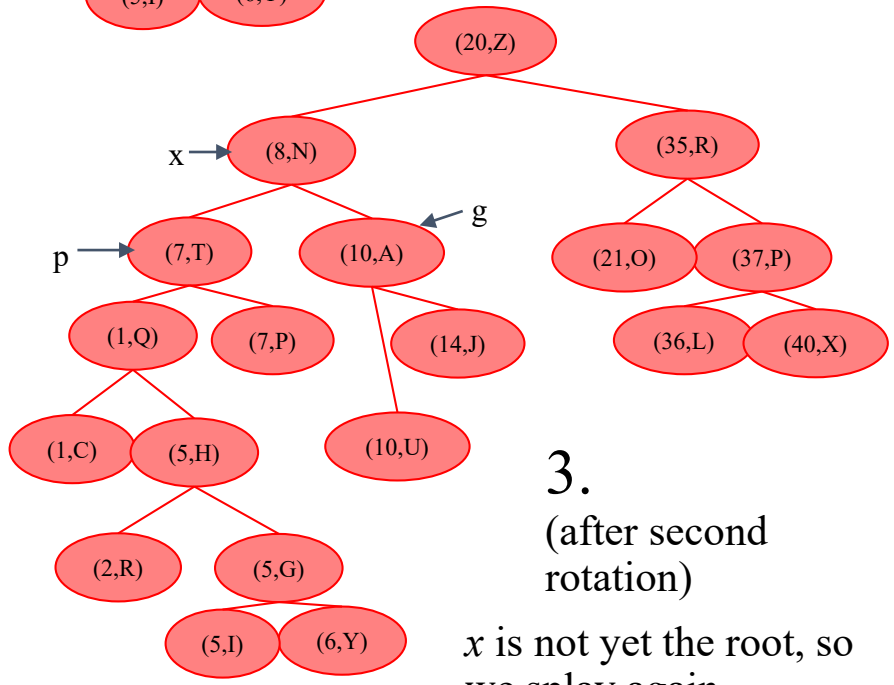
- let x = (8,N)
  - x is the right child of its parent, which is the left child of the grandparent
  - left-rotate around p, then right-rotate around g



1.
(before rotating)



2.
(after first rotation)

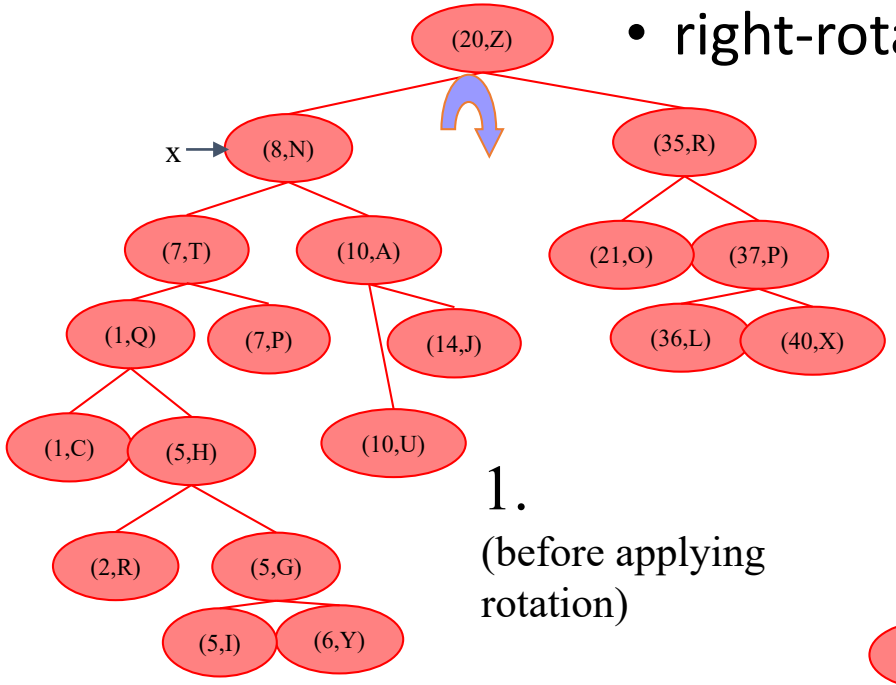


3.
(after second rotation)

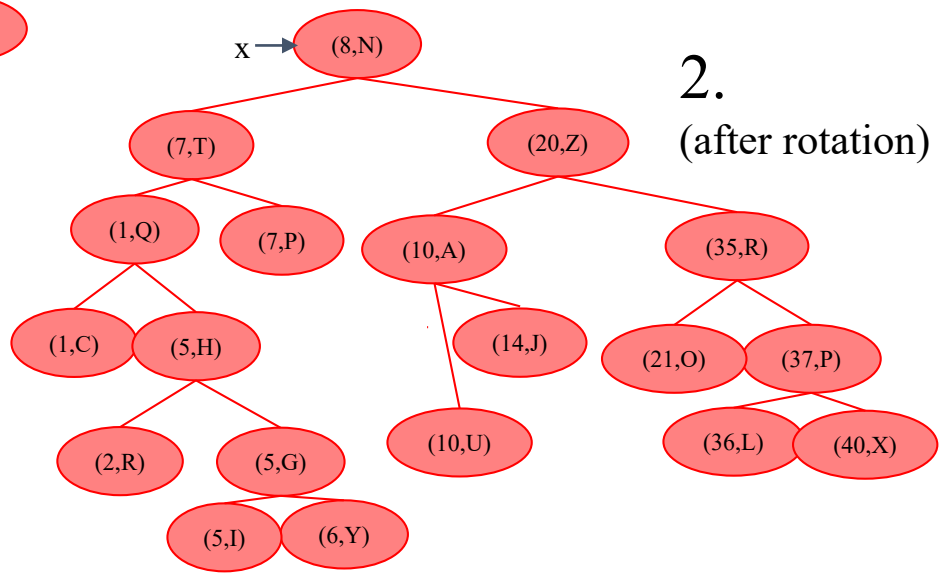
x is not yet the root, so we splay again

Splaying Example, Continued

- now *x* is the left child of the root
  - right-rotate around root



1.
(before applying rotation)

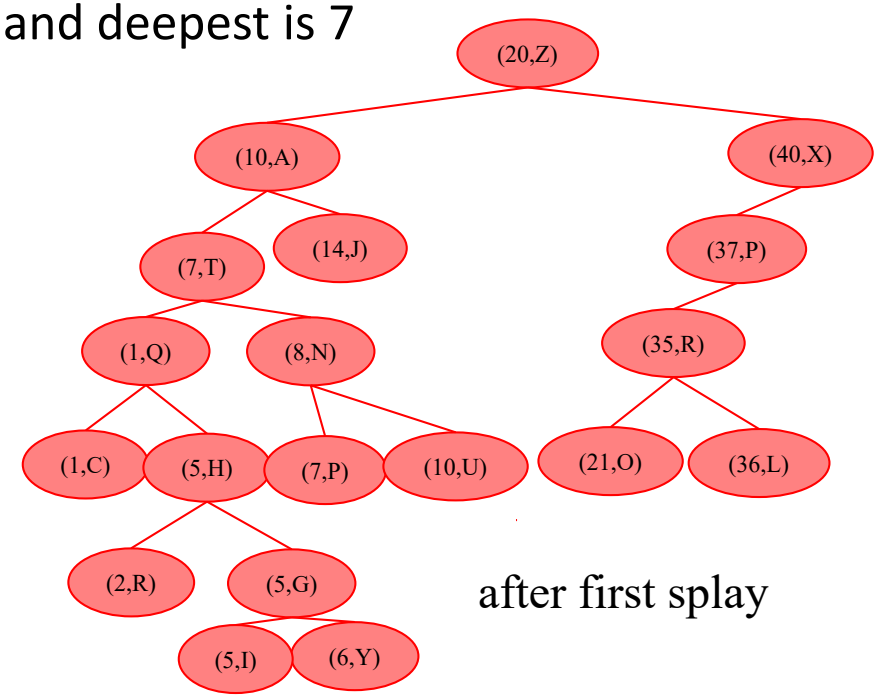


2.
(after rotation)

x is the root, so stop

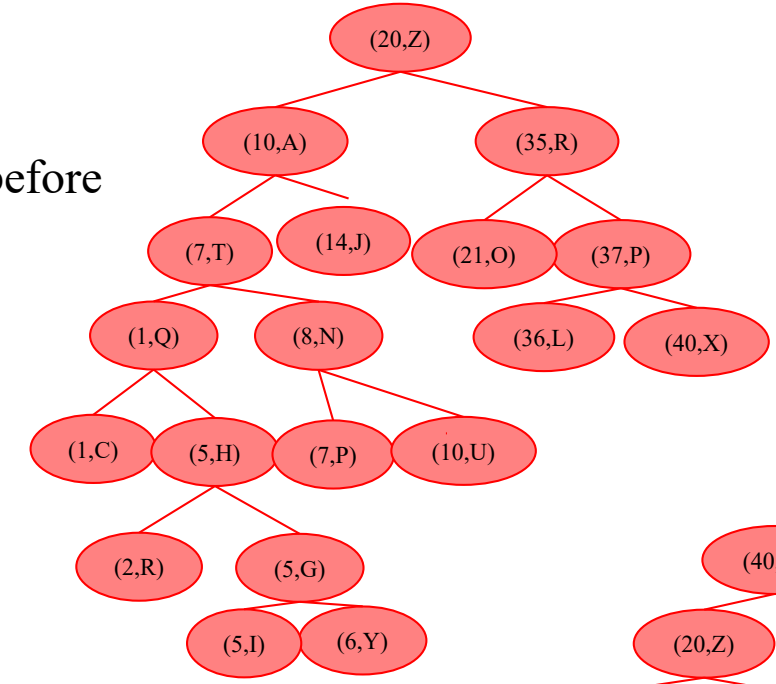
Example Result of Splaying

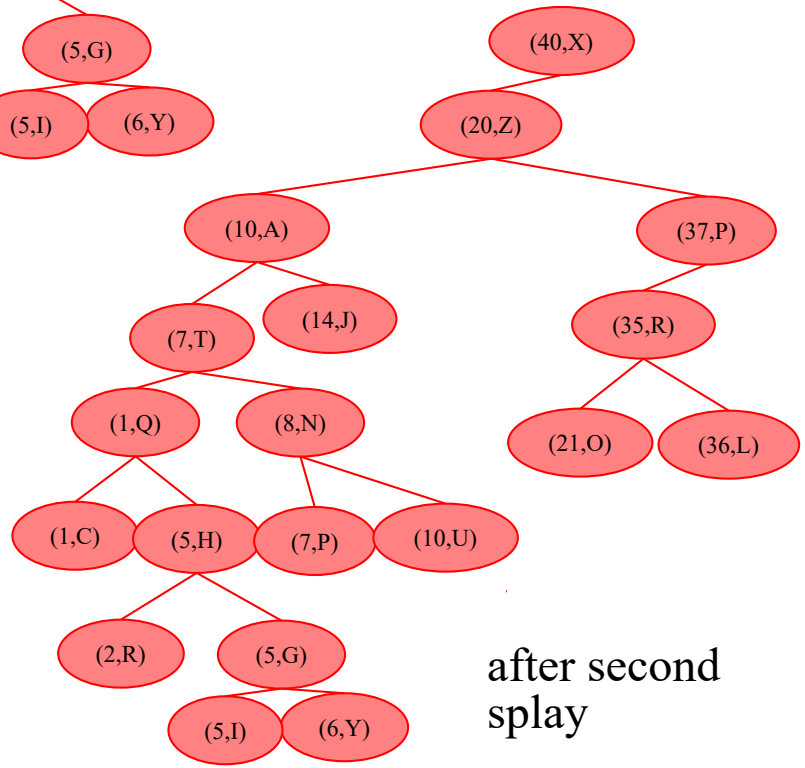
- tree might not be more balanced
- e.g. splay (40,X)
  - before, the depth of the shallowest leaf is 2 and the deepest is 6
  - after, the depth of shallowest leaf is 3 and deepest is 7



after first splay

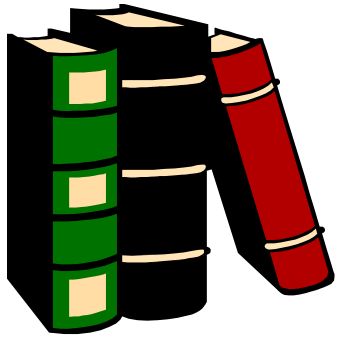
before





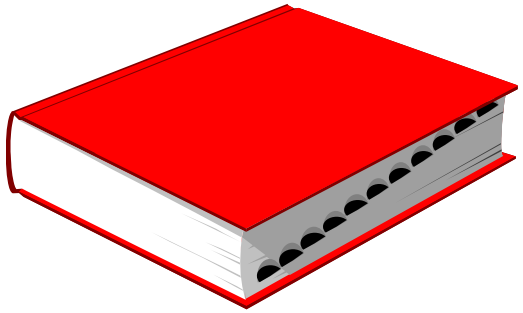
after second splay

Splay Tree Definition



- a *splay tree* is a binary search tree where a node is splayed after it is accessed (for a search or update)
  - deepest node accessed is splayed
  - splaying costs O(h),where h is height of the tree – which is still O(n) worst-case **worst case is when tree is a list**
    - O(h) rotations, each of which is O(1)

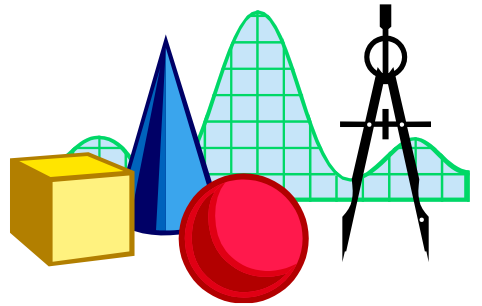
Splay Trees & Ordered Dictionaries



- which nodes are splayed after each operation?

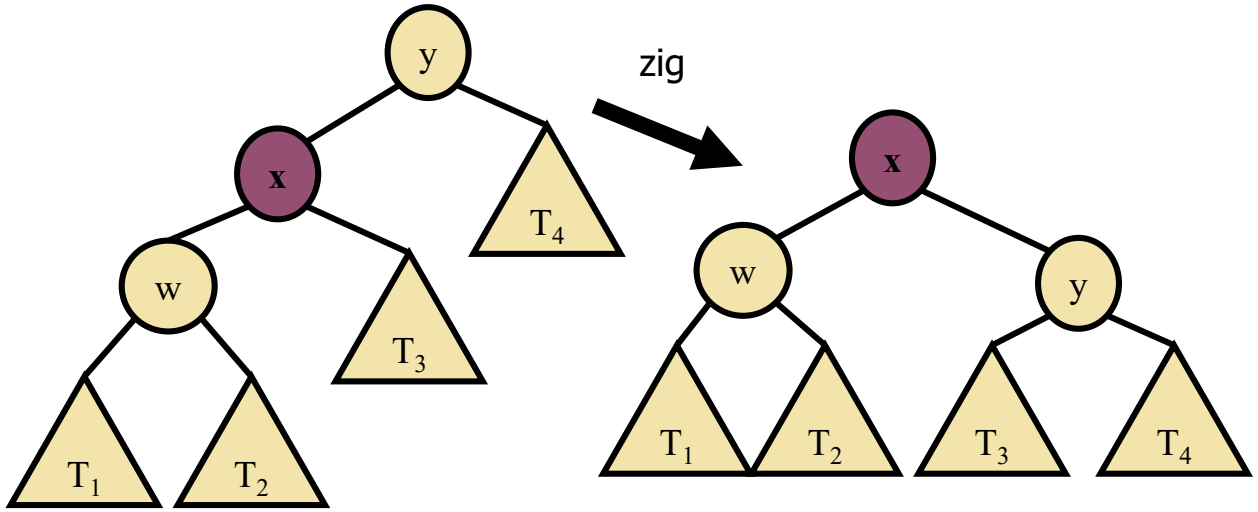
method	splay node
find(k)	if key found, use that node if key not found, use the node where the search stops
insert(k,v)	use the new node containing the entry inserted
delete(k)	use the parent of the node that was actually removed from the tree (the parent of the node that the removed item was swapped with)

Amortized Analysis of Splay Trees



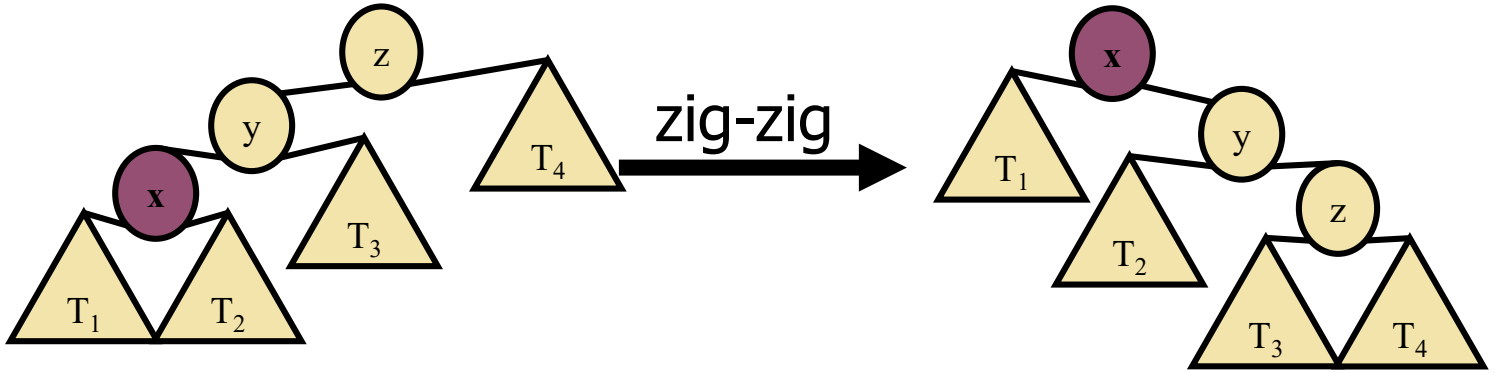
- Running time of each operation is proportional to time for splaying.
- Define rank(v) as the logarithm (base 2) of the number of nodes in subtree rooted at v.
- Costs: zig = $1, zig-zig = $2, zig-zag = $2.
- Thus, cost for splaying a node at depth d = $d.
- Imagine that we store rank(v) & cyber-dollars at each node v of the splay tree (just for the sake of analysis).

Cost per zig

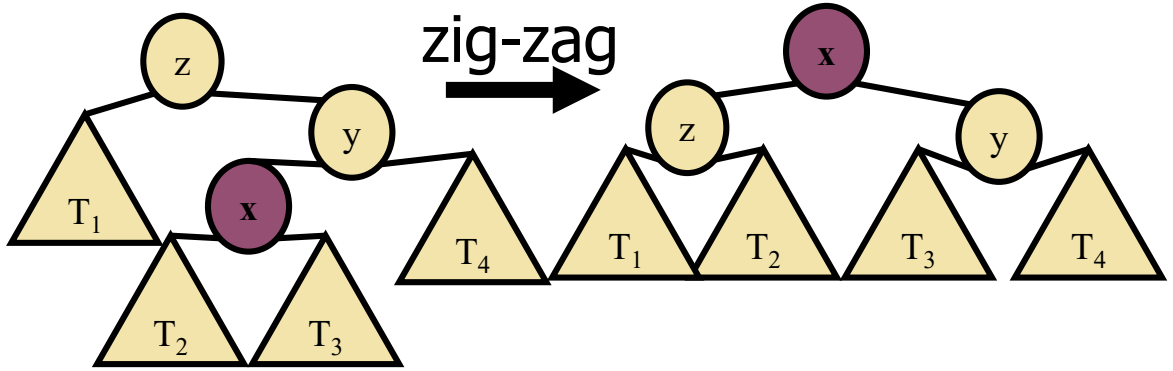


- Doing a zig at x costs at most rank'(x) - rank(x):
  - cost = rank'(x) + rank'(y) - rank(y) - rank(x)
    ≤ rank'(x) - rank(x).

Cost per zig-zig and zig-zag



- Doing a zig-zig or zig-zag at x costs at most
  3(rank'(x) - rank(x)) - 2.
  - Proof: See Proposition 9.2, Page 440.



Cost of Splaying

- Cost of splaying a node x at depth d of a tree rooted at r:
  - at most 3(rank(r) - rank(x)) - d + 2:
  - Proof: Splaying x takes d/2 splaying substeps:

$$\begin{aligned}
\text{cost} &\le \sum_{i=1}^{d/2} \text{cost}_i \\
&\le \sum_{i=1}^{d/2} (3(\text{rank}_i(x) - \text{rank}_{i-1}(x)) - 2) + 2 \\
&= 3(\text{rank}(r) - \text{rank}_0(x)) - 2(d/d) + 2 \\
&\le 3(\text{rank}(r) - \text{rank}(x)) - d + 2.
\end{aligned}$$

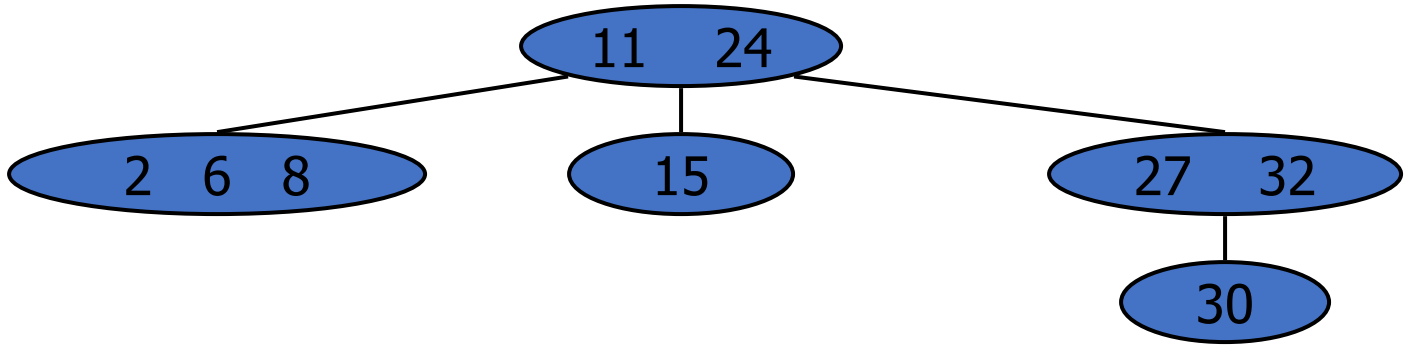
Performance of Splay Trees

- Recall: rank of a node is logarithm of its size.
- Thus, amortized cost of any splay operation is **O(log n).**
- In fact, the analysis goes through for any reasonable definition of rank(x).
- This implies that splay trees can actually adapt to perform searches on frequently-requested items much faster than O(log n) in some cases.

(2,4) Trees

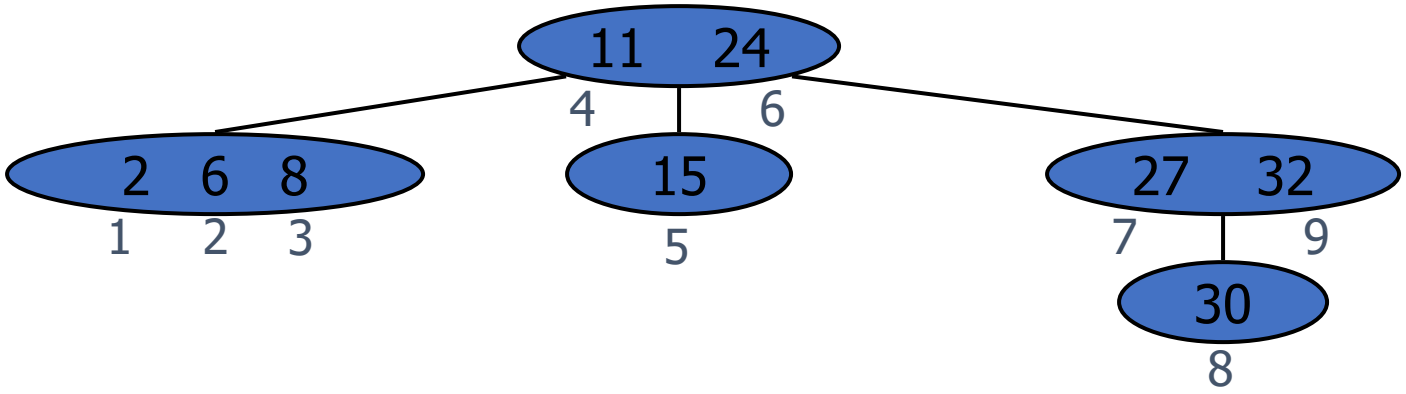
Multi-Way Search Tree

- A multi-way search tree is an ordered tree such that
  - Each internal node has at least two children and stores $d - 1$ key-element items (k_i, o_i), where d is the number of children
  - For a node with children $v_1\ v_2\ \ldots\ v_d$ storing keys $k_1\ k_2\ \ldots\ k_{d-1}$
    - keys in the subtree of v_1 are less than k_1
    - keys in the subtree of v_i are between k_{i-1} and k_i $(i = 2, \ldots, d - 1)$
    - keys in the subtree of v_d are greater than k_{d-1}



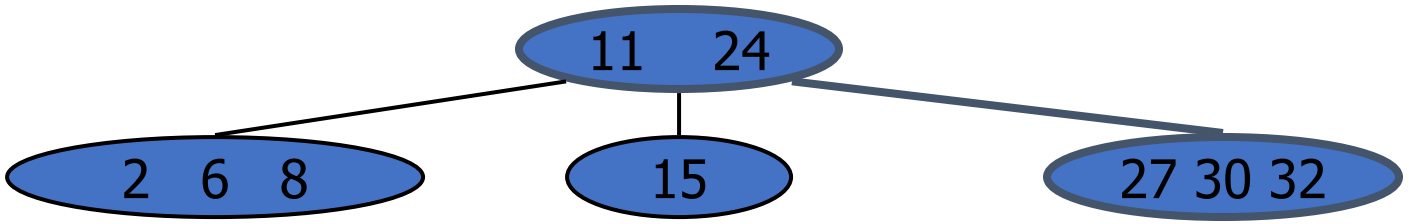
Multi-Way Inorder Traversal

- We can extend the notion of inorder traversal from binary trees to multi-way search trees
- Namely, we visit item (k_i, o_i) of node v between the recursive traversals of the subtrees of v rooted at children v_i and v_{i+1}
- An inorder traversal of a multi-way search tree visits the keys in increasing order



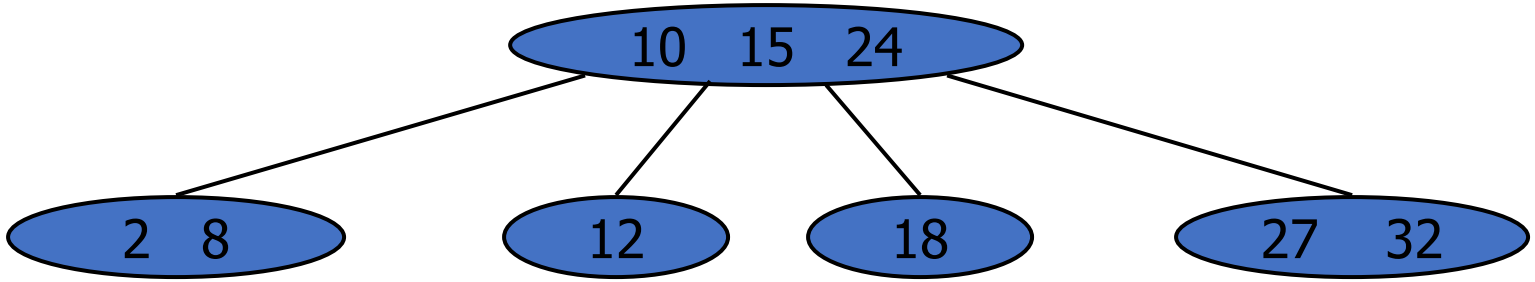
Multi-Way Searching

- Similar to search in a binary search tree
- A each internal node with children $v_1\ v_2\ \ldots\ v_d$ and keys $k_1\ k_2\ \ldots\ k_{d-1}$
  - $k = k_i\ (i = 1, \ldots, d-1)$: the search terminates successfully
  - $k < k_1$: we continue the search in child v_1
  - $k_{i-1} < k < k_i\ (i = 2, \ldots, d-1)$: we continue the search in child v_i
  - $k > k_{d-1}$: we continue the search in child v_d
- Example: search for 30



(2,4) Trees

- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
  - Node-Size Property: every node is either a 2-node, or a 3-node, or a 4-node, and holds one element, or two elements, or three elements, respectively.
  - Depth Property: all the external (leaf) nodes have the same depth.



Height of a (2,4) Tree

- Theorem: A (2,4) tree storing n items has height $O(\log n)$

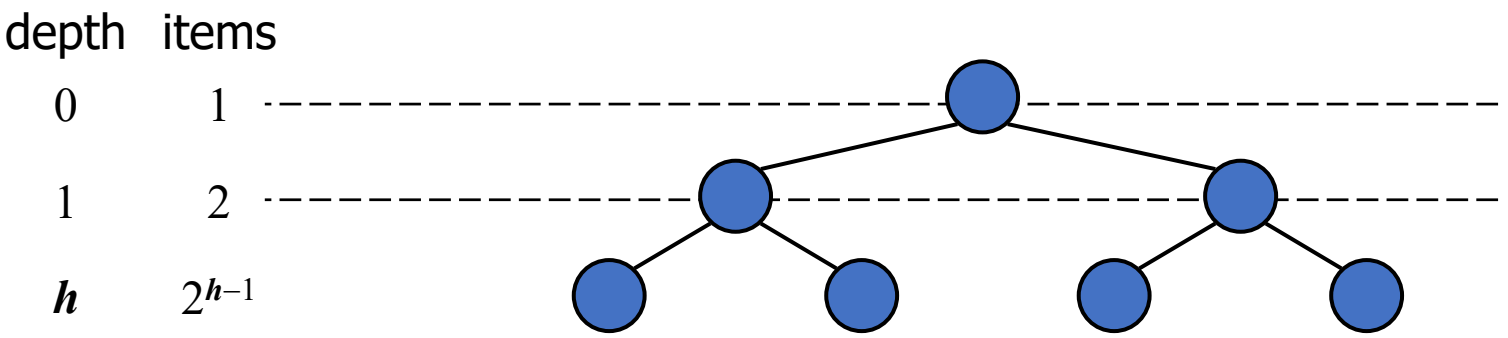
  Proof:

  - Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \ldots, h-1$ and no items at depth h, we have

    $$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$

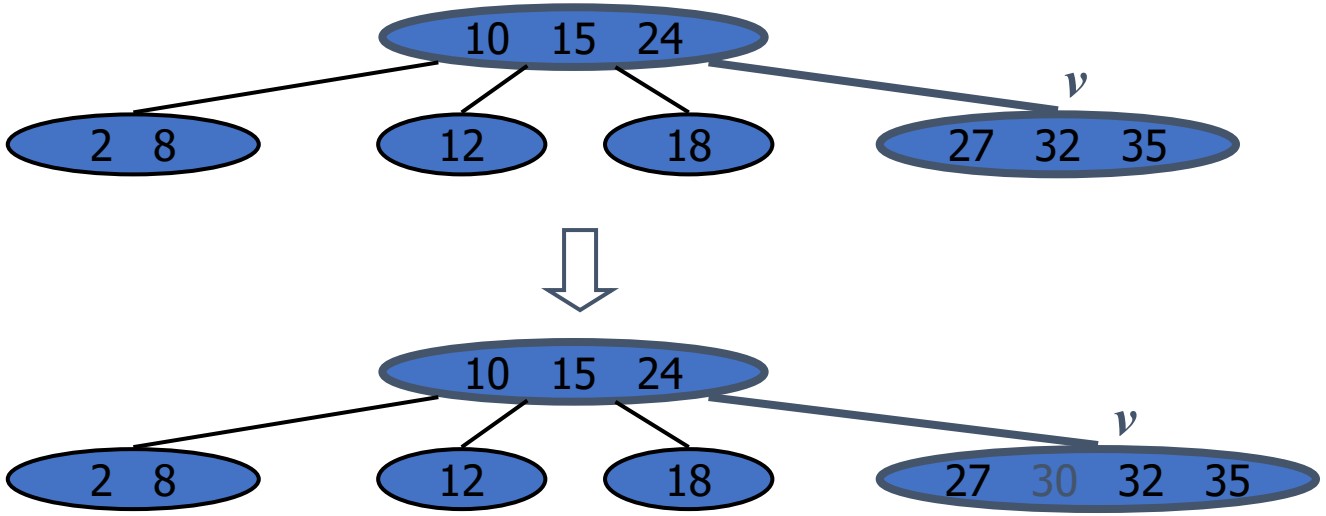
  - Thus, $h \leq \log (n+1)$

- Searching in a (2,4) tree with n items takes $O(\log n)$ time



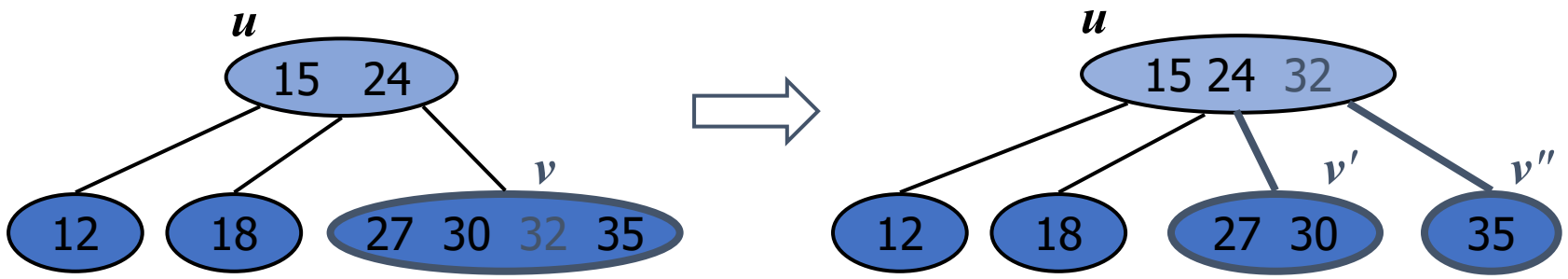
Insertion

- We insert a new item (k, o) at a leaf node reached by searching for k
  - We preserve the depth property but
  - We may cause an overflow (i.e., node v may become a 5-node)
- Example: inserting key 30 causes an overflow



Overflow and Split

- We handle an overflow at a 5-node v with a split operation:
  - let $v_1 \ldots v_5$ be the children of v and $k_1 \ldots k_4$ be the keys of v
  - node v is replaced by nodes v' and v''
    - v' is a 3-node with keys k_1 k_2
    - v'' is a 2-node with key k_4
  - key k_3 is inserted into the parent u of v (a new root may be created)
- The overflow may propagate to the parent node u



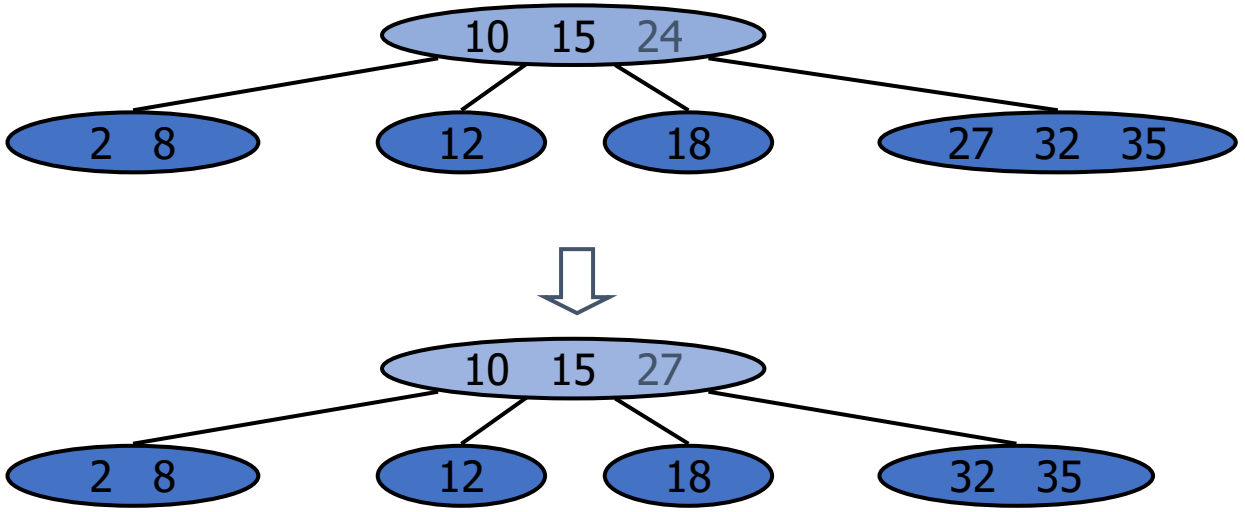
Analysis of Insertion

```
Algorithm insert(k, o)
  {
  search for key k to locate the insertion
  node v;

   add the new entry (k, o) at node v;

   while ( overflow(v) )
     { if  ( isRoot(v) )
           create a new empty root above v;
       v = split(v);
     }
  }
```

- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
  - Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
  - Step 2 takes $O(1)$ time
  - Step 3 takes $O(\log n)$ time because each split takes $O(1)$ time and we perform $O(\log n)$ splits
- Thus, an insertion in a (2,4) tree takes $O(\log n)$ time

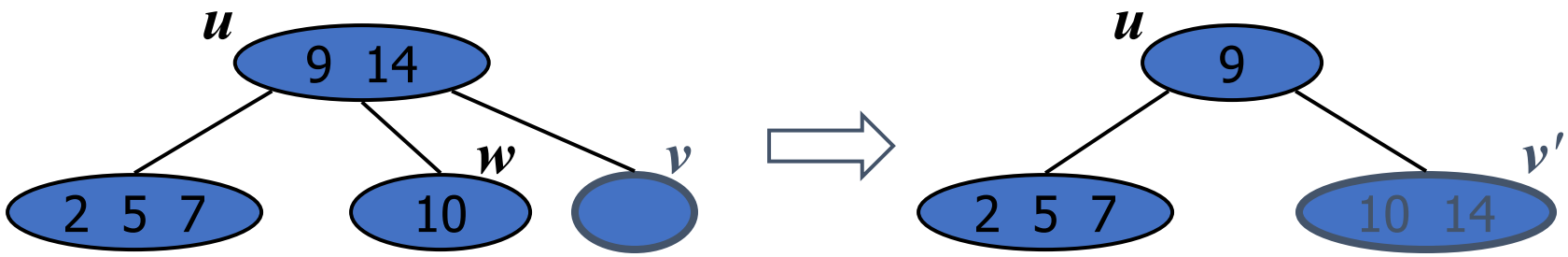
Deletion

- We reduce deletion of an entry to the case where the item is at a leaf node
- Otherwise, we replace the entry with its inorder successor (or, equivalently, with its inorder predecessor) and delete the latter entry
- Example: to delete key 24, we replace it with 27 (inorder successor)



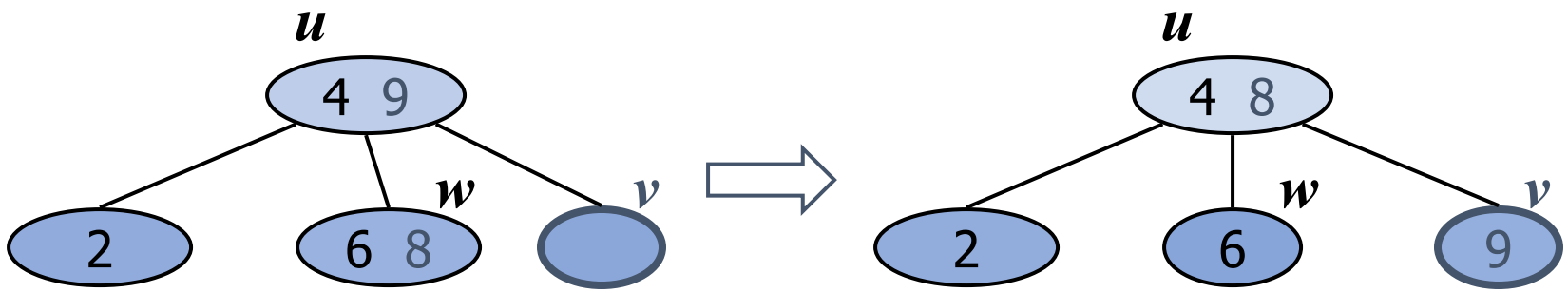
Underflow and Fusion

- Deleting an entry from a node v may cause an underflow, where node v becomes a 1-node with one child and no keys
- To handle an underflow at node v with parent u, we consider two cases
- Case 1: the adjacent siblings of v are 2-nodes
  - Fusion operation: we merge v with an adjacent sibling w and move an entry from u to the merged node v'
  - After a fusion, the underflow may propagate to the parent u



Underflow and Transfer

- To handle an underflow at node v with parent u, we consider two cases
- Case 2: an adjacent sibling w of v is a 3-node or a 4-node
  - Transfer operation:
    1. we move an item from u to v
    2. we move an item from w to u
  - After a transfer, no underflow occurs



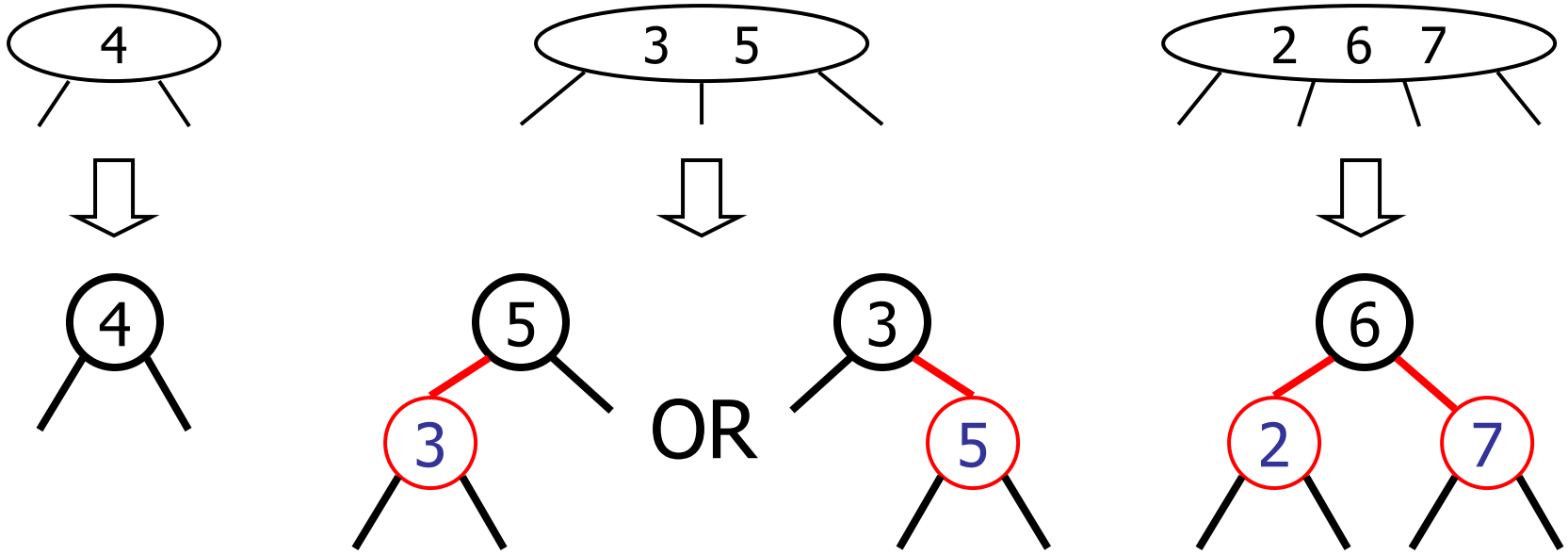
Analysis of Deletion

- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
- In a deletion operation
  - We visit $O(\log n)$ nodes to locate the node from which to delete the entry
  - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - Each fusion and transfer takes $O(1)$ time
- Thus, deleting an item from a (2,4) tree takes $O(\log n)$ time

Red-Black Trees

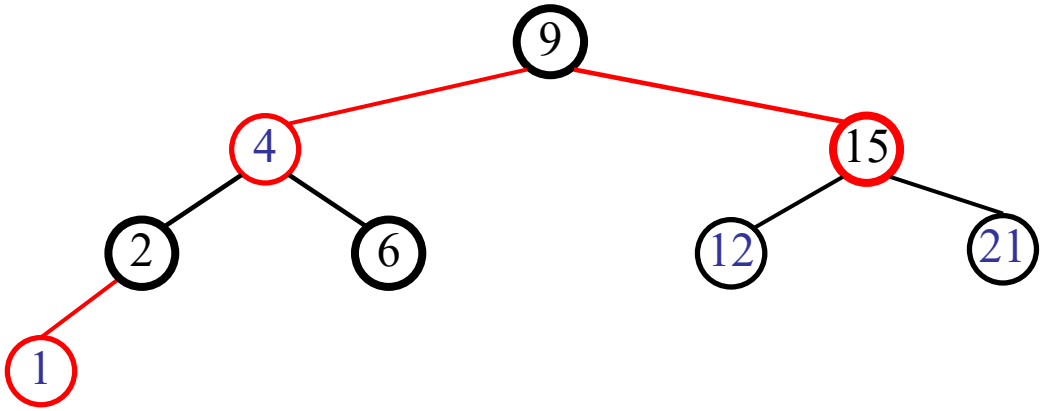
From (2,4) to Red-Black Trees

- A red-black tree is a representation of a (2,4) tree by means of a binary tree whose nodes are colored red or **black**
- In comparison with its associated (2,4) tree, a red-black tree has
  - same logarithmic time performance
  - simpler implementation with a single node type



Red-Black Trees

- A red-black tree is a binary search tree that satisfies the following properties:
  - Color Properties:
    - Each node is either black or red
    - The root is black
    - There are no two adjacent red nodes.
  - Depth Property: the number of black nodes on every path from the root to each leaf is the same



Height of a Red-Black Tree

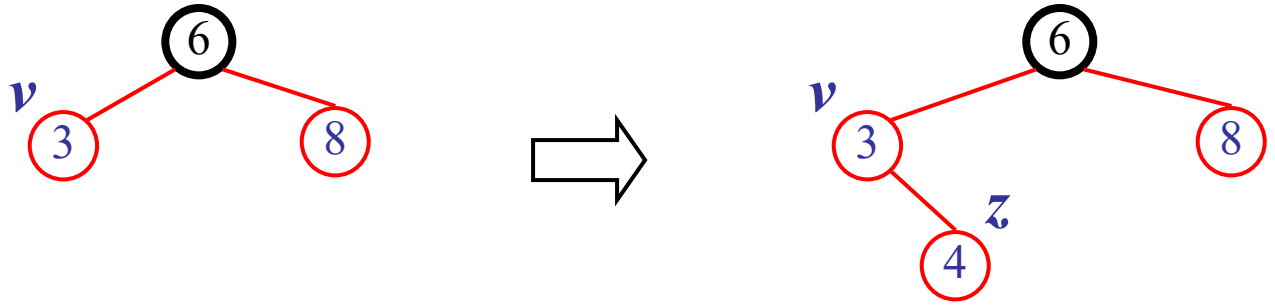
- Theorem: A red-black tree storing n entries has height $O(\log n)$

  Proof:

  - The height of a red-black tree is at most twice the height of its associated (2,4) tree, which is $O(\log n)$
- The search algorithm for a binary search tree is the same as that for a binary search tree
- By the above theorem, searching in a red-black tree takes $O(\log n)$ time

Insertion

- To perform operation insert(k, o), we execute the insertion algorithm for binary search trees and color red the newly inserted node z unless it is the root
  - We preserve the color and depth properties
  - If the parent v of z is black, we are done
  - Else (v is red) we have a double red (i.e., a violation of the color properties), which requires a restructuring of the tree
- Example where the insertion of 4 causes a double red:

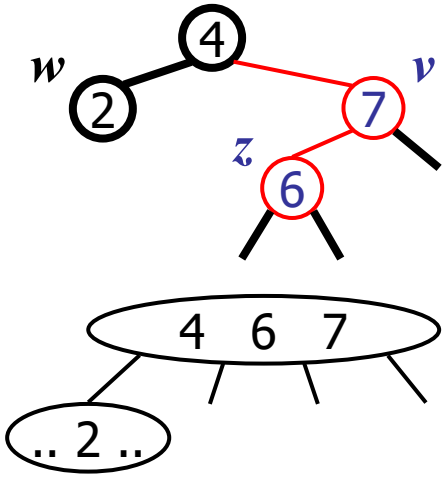


Remedying a Double Red

- Consider a double red with child z and parent v, and let w be the sibling of v

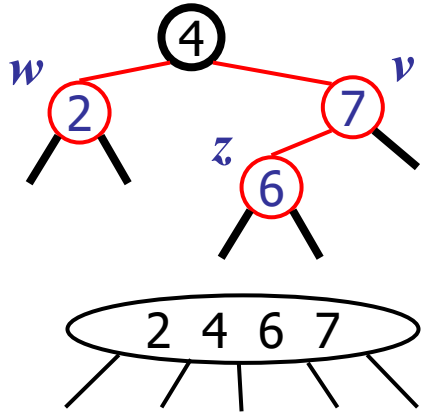
Case 1: w is black

- The double red is an incorrect replacement of a 4-node
- Restructuring: we change the 4-node replacement



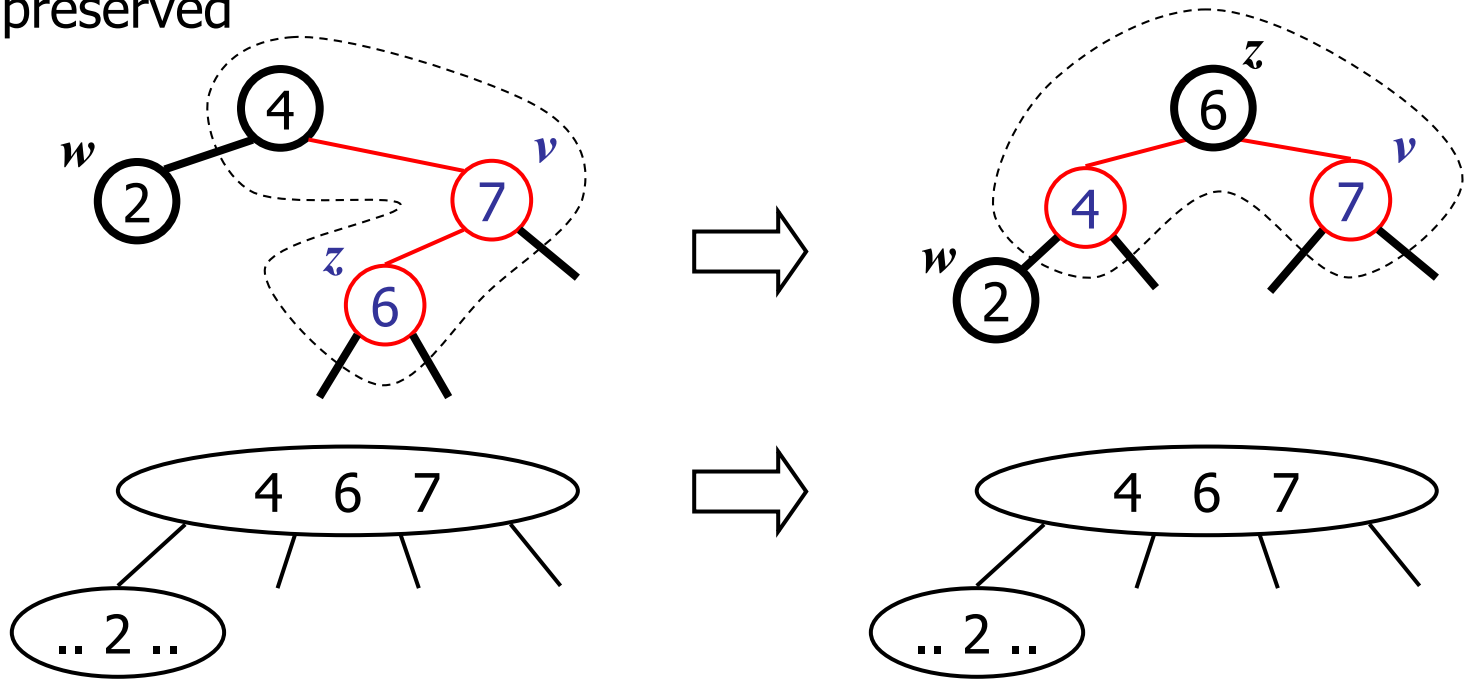
Case 2: w is red

- The double red corresponds to an overflow
- Recoloring: we perform the equivalent of a split



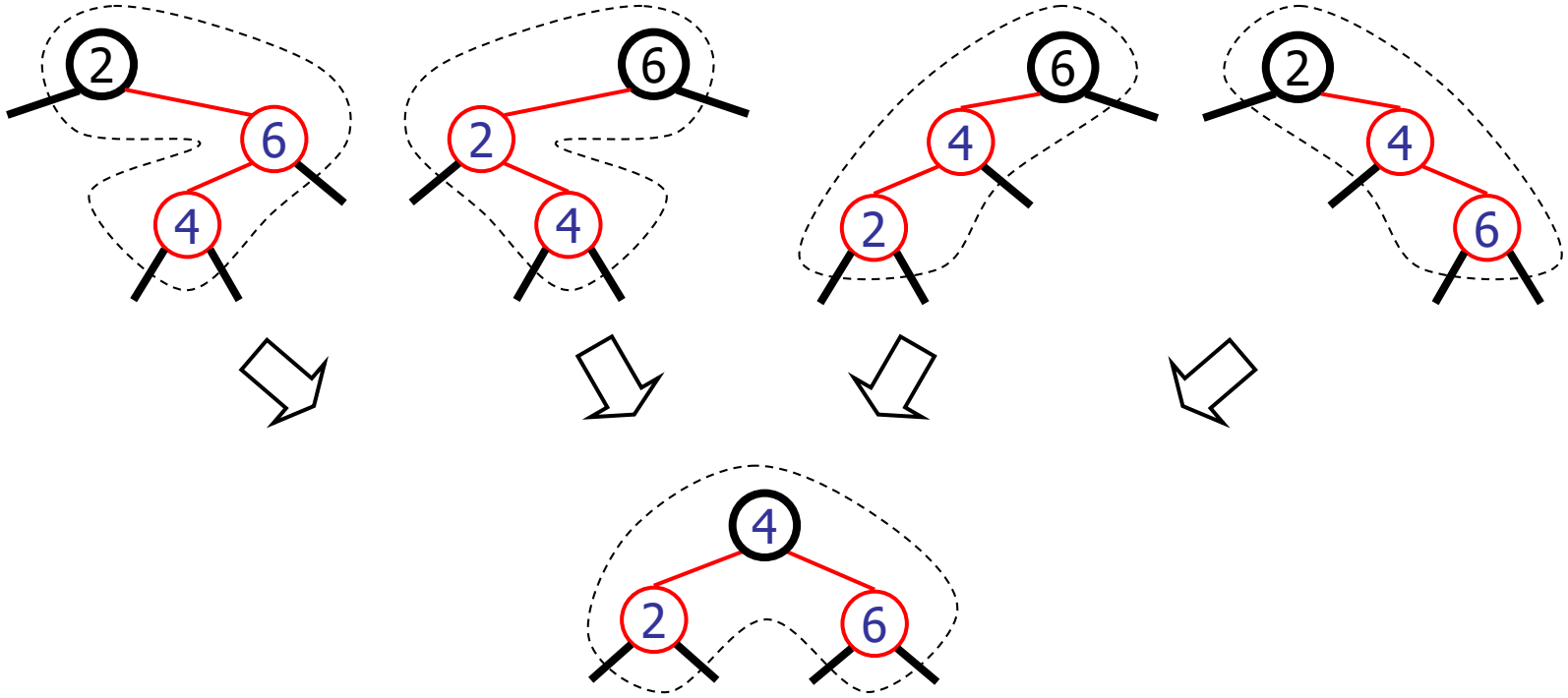
Restructuring (1/2)

- A restructuring remedies a child-parent double red when the parent red node has a black sibling
- It is equivalent to restoring the correct replacement of a 4-node
- The internal property is restored and the other properties are preserved



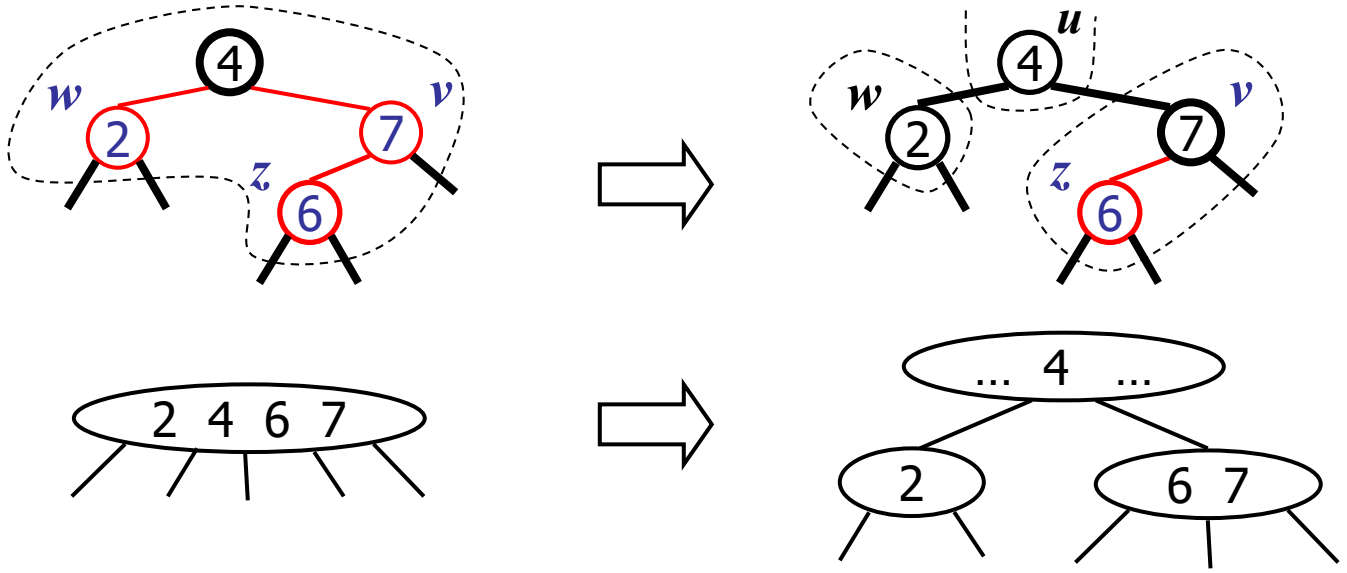
Restructuring (2/2)

- There are four restructuring configurations depending on whether the double red nodes are left or right children



Recoloring

- A recoloring remedies a child-parent double red when the parent red node has a red sibling
- The parent v and its sibling w become black and the grandparent u becomes red, unless it is the root
- It is equivalent to performing a split on a 5-node
- The double red violation may propagate to the grandparent u



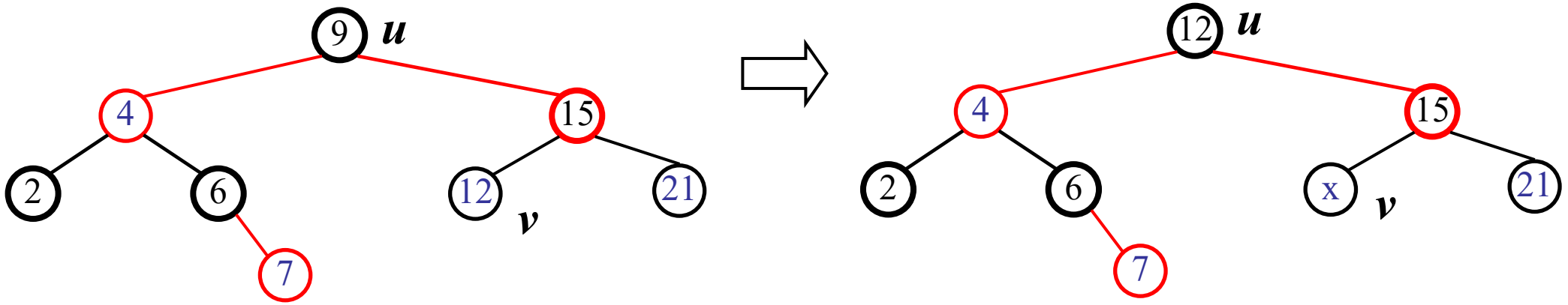
Analysis of Insertion

```
Algorithm insert(k, o)
 { search for key k to locate the
   insertion node z;

   add the new entry (k, o) at node z
   and color z red;

   while doubleRed(z)
     { if ( isBlack(sibling(parent(z))))
         {  z = restructure(z);
            return; }
       else // sibling(parent(z)) is red
          z = recolor(z);
 }
```

- Recall that a red-black tree has $O(\log n)$ height
- Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(\log n)$ time because we perform
  - $O(\log n)$ recolorings, each taking $O(1)$ time, and
  - at most one restructuring taking $O(1)$ time
- Thus, an insertion in a red-black tree takes $O(\log n)$ time

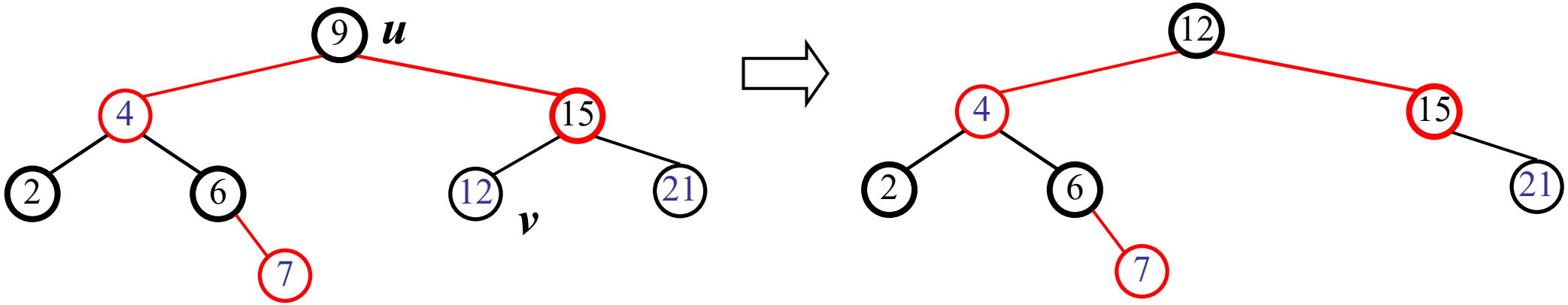
Deletion (1/9)

- To perform operation delete(k), we first execute the deletion algorithm for binary search tree. We search for a node u storing such an entry. If node ***u*** has two children, we find the node ***v*** following ***u*** or preceding ***u*** in the inorder traversal of T, move the entry at ***v*** to ***u***, and perform the removal at ***v***. Thus, we only consider the removal of an entry with key ***k*** stored at a node ***v*** with at most one child.
- Example: delete 9



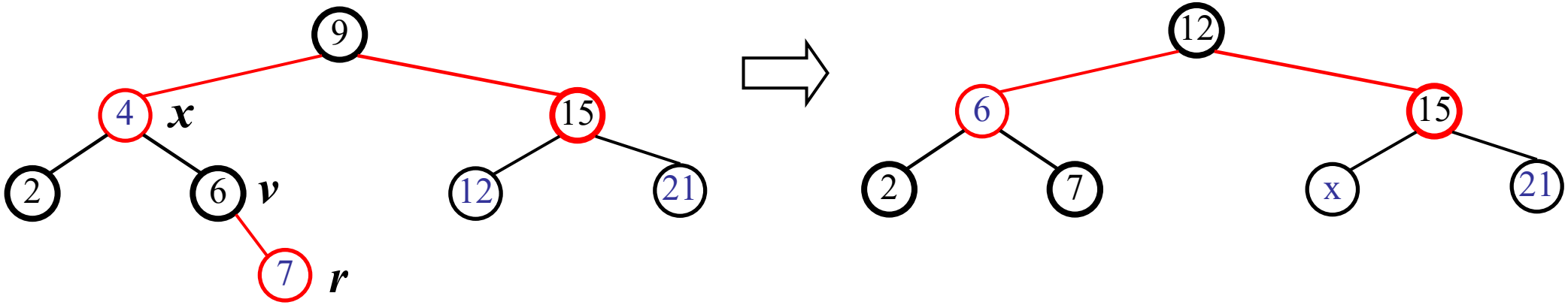
Deletion (2/9)

- If v has no child, we are done.



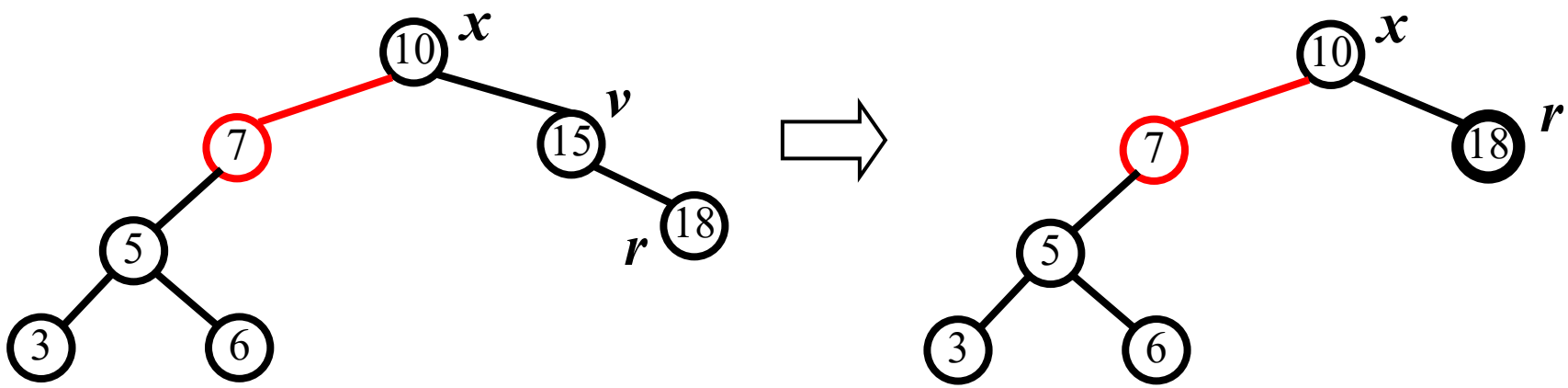
Deletion (3/9)

- Assume v has one child. Let r be the child of v and x be the parent of v. We remove nodes v, and make r a child of x. If v was red (hence r is black) or r is red (hence v was black), we color r black and we are done.
- Example: delete 4.



Deletion (4/9)

- If, instead, r is black and v was black, then, to preserve the depth property, we give r a fictitious double black color. We now have a color violation, called the double black problem. A double black in T denotes an underflow in the corresponding (2,4) tree T.



Deletion (5/9)

- Recall that x is the parent of the double black node r. Let y be the sibling of r. To remedy the double-black problem at r, we consider three cases:

  Case 1: y is black and has a red child
  - We perform a restructuring, equivalent to a transfer, and we are done

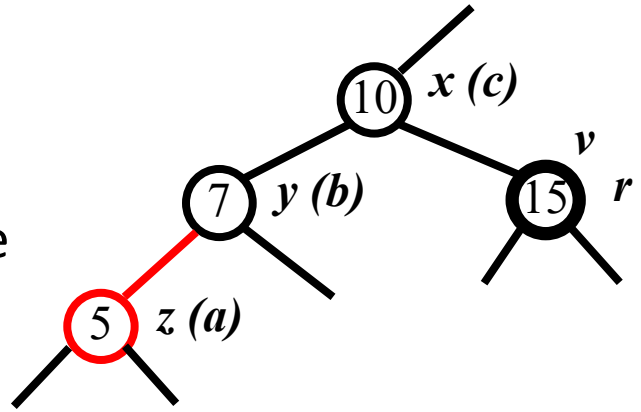
  Case 2: y is black and its children are both black
  - We perform a recoloring, equivalent to a fusion, which may propagate up the double black violation

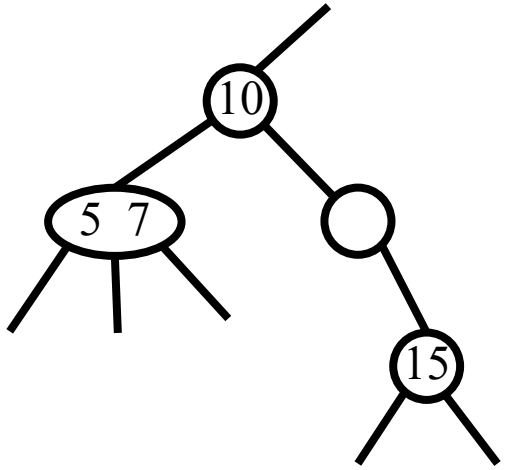
  Case 3: y is red
  - We perform an adjustment, equivalent to choosing a different representation of a 3-node, after which either Case 1 or Case 2 applies

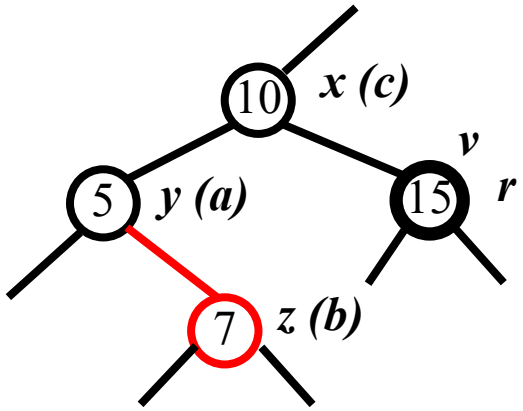
Deletion (6/9)

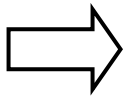
Case 1: y is black and has a red child

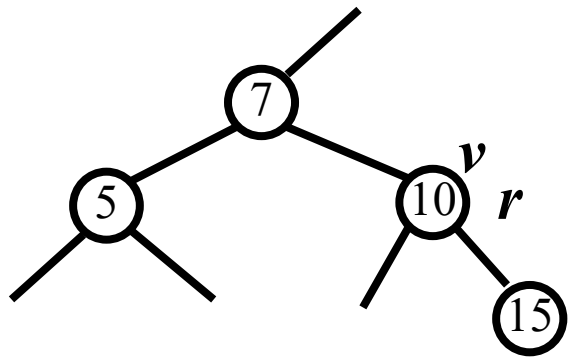
We perform tri-node restructuring. Color ***a*** and ***c*** black, give ***b*** the former color of ***x***, and color ***r*** black. This tri-node restructuring eliminates the double black problem.







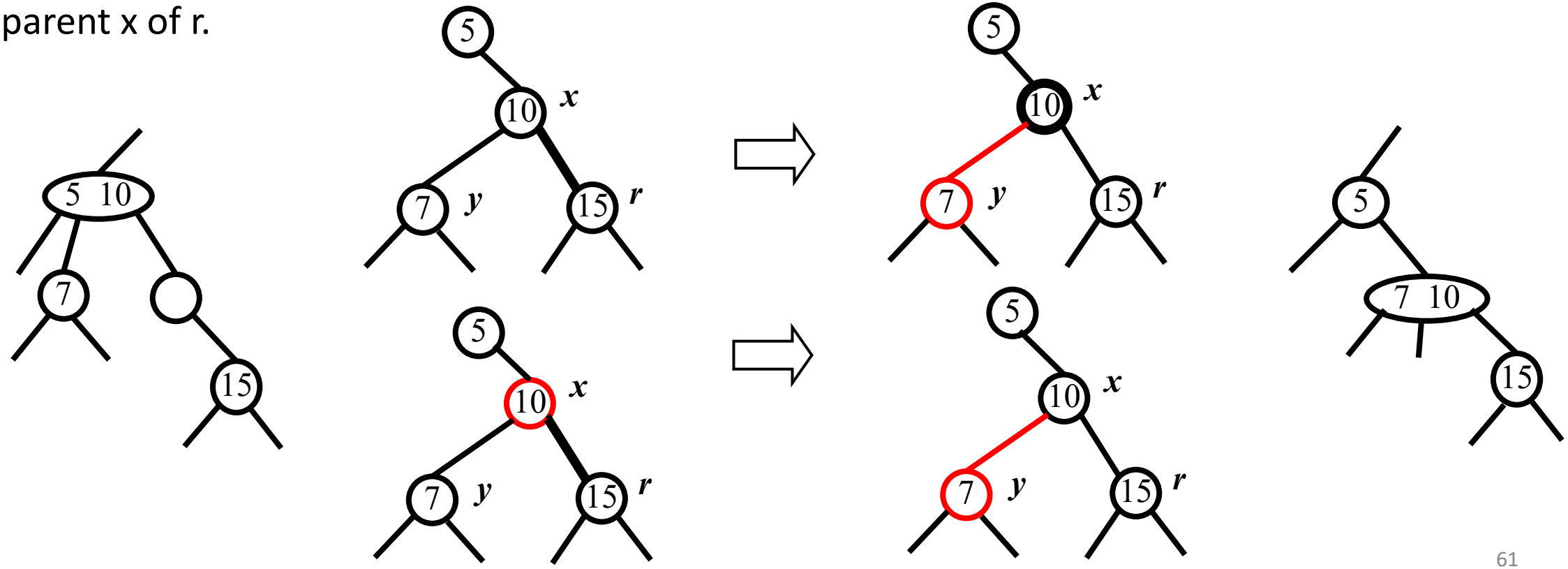




Deletion (7/9)

Case 2: The sibling ***y*** of ***r*** is black and both children of ***y*** are black.

We do a recoloring: color r black, color y red, and, if x is red, color it black; otherwise, color x double black. After this recoloring, the double black problem may reappear at the parent x of r.



Deletion (8/9)

Case 3: The sibling ***y*** of ***r*** is red.

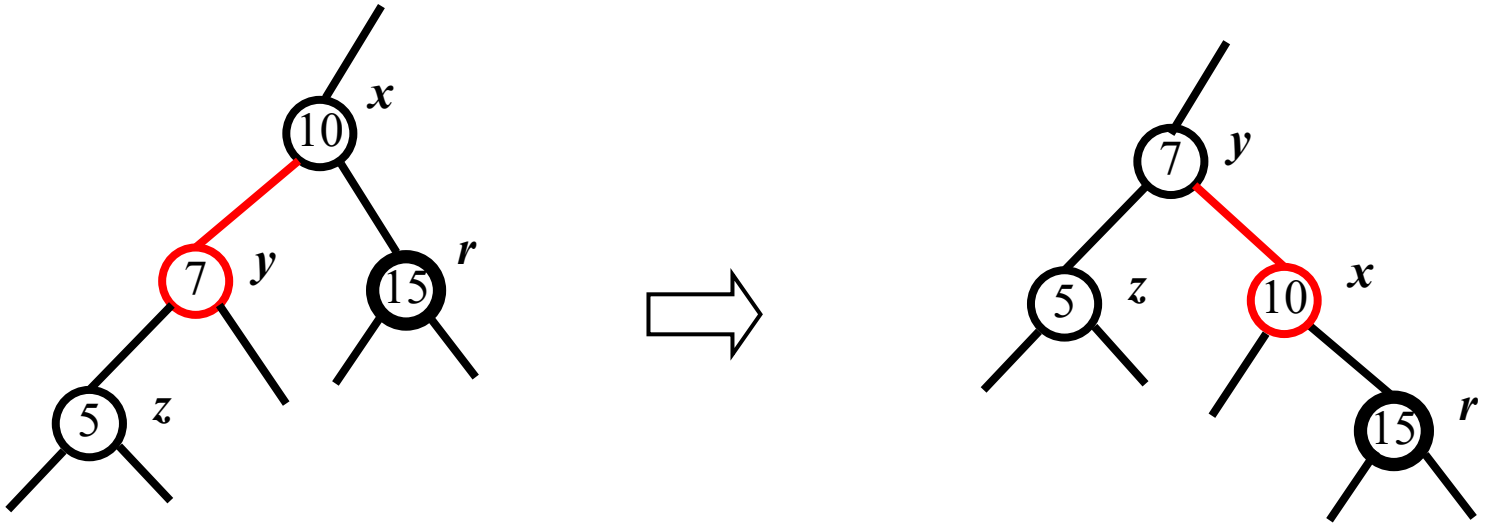
In this case, we perform an adjustment operation as follows:

1. If y is the right child of x, let z be the right child of y; otherwise, let z be the left child of y.
2. Execute the trinode restructuring operation restructure (z), which makes y the parent of x.
3. Color y black and x red.

An adjustment corresponds to choosing a different representation of a 3-node in the (2,4) tree. After the adjustment operation, the sibling of r is black, and either Case 1 or Case 2 applies, with a different meaning of x and y. Note that if Case 2 applies, the double-black problem cannot reappear. Thus, to complete Case 3 we make one more application of either Case 1 or Case 2 above and we are done. Therefore, at most one adjustment is performed in a removal operation.

Deletion (9/9)

Case 3: The sibling ***y*** of ***r*** is red.



Red-Black Tree Reorganization

Insertion	remedy double red	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	change of 4-node representation	double red removed
recoloring	split	double red removed or propagated up
Deletion	remedy double black	
Red-black tree action	**(2,4) tree action**	**result**
restructuring	transfer	double black removed
recoloring	fusion	double black removed or propagated up
adjustment	change of 3-node representation	restructuring or recoloring follows

Summary

- AVL Trees
- Splay Trees
- (2,4)-Trees
- Read-black Trees
- Suggested reading:
  - Sedgewick, Ch.13