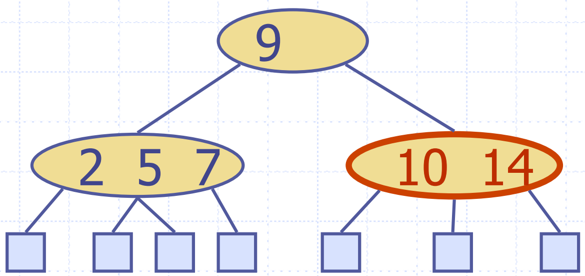
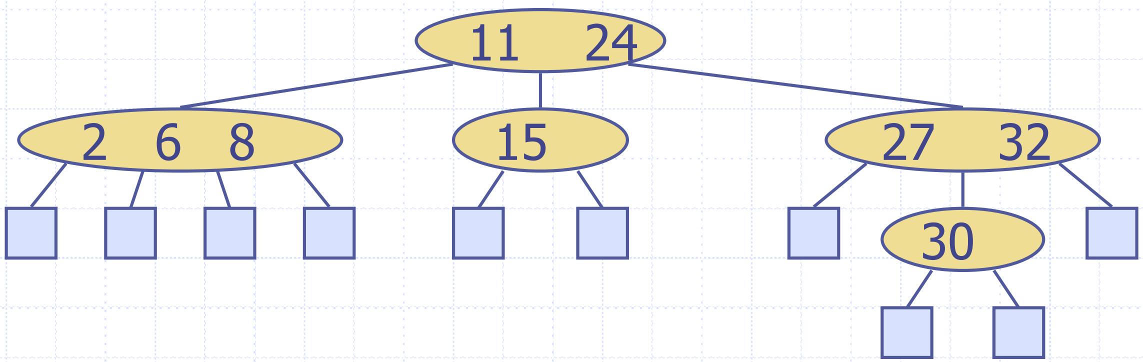
(2,4) Trees



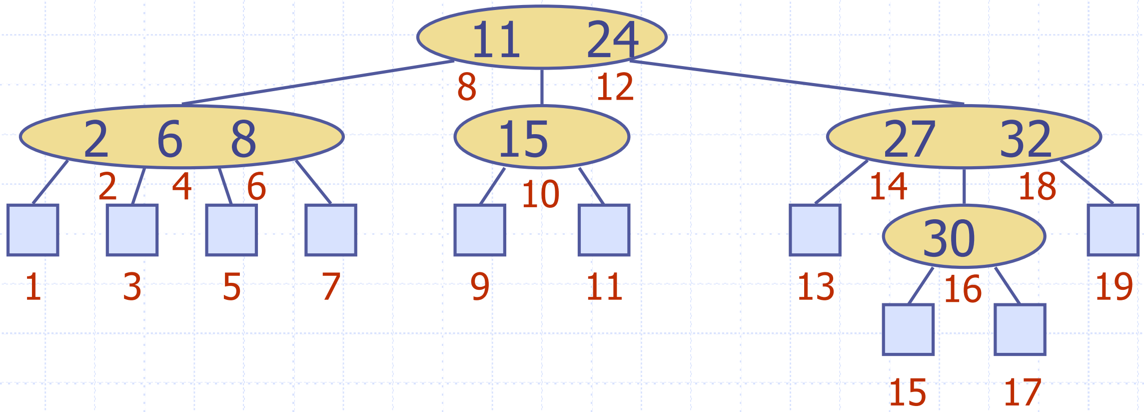
Multi-Way Search Tree

- 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}
  - The leaves store no items and serve as placeholders



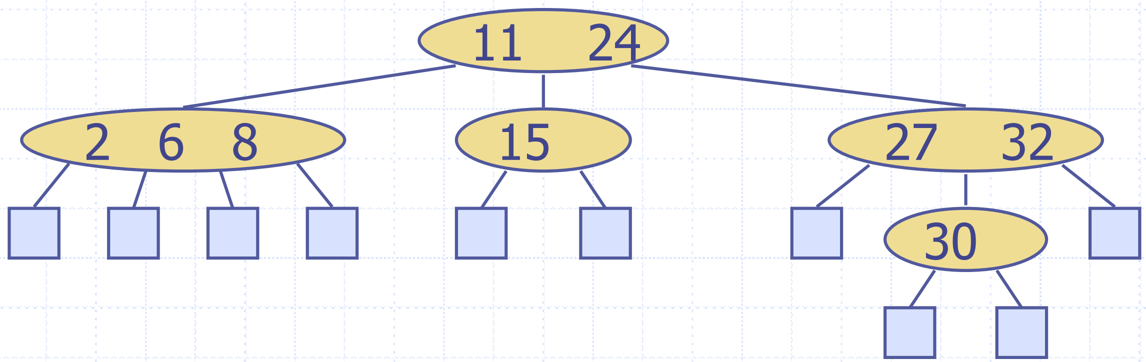
Multi-Way Inorder Traversal

- 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



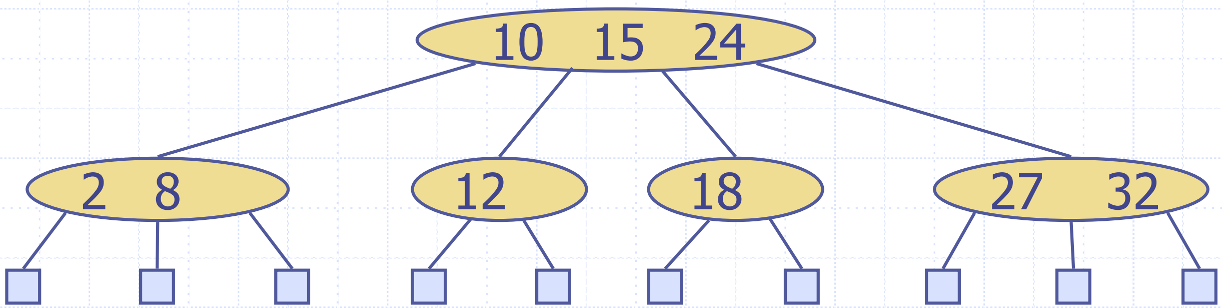
Multi-Way Searching

- Similar to search in a binary search tree
- For 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
- Reaching an external node terminates the search unsuccessfully
- Example: search for 30



(2,4) Trees

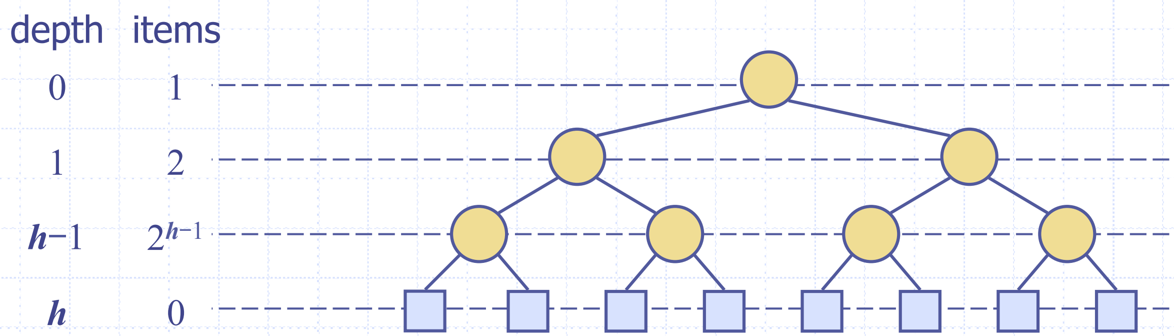
- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4) tree is called a 2-node, 3-node or 4-node



Height of a (2,4) Tree

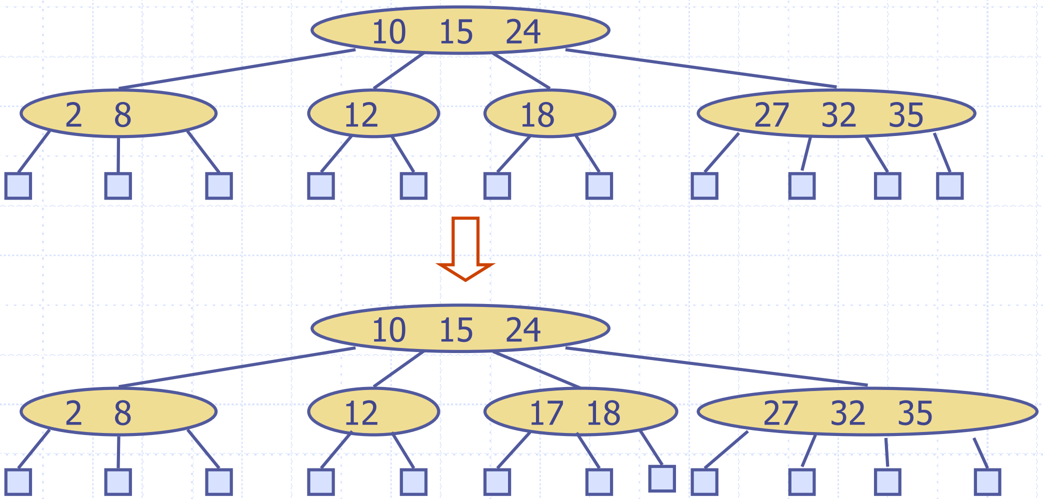
- Theorem: A (2,4) tree storing n items has height $O(\log n)$

  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



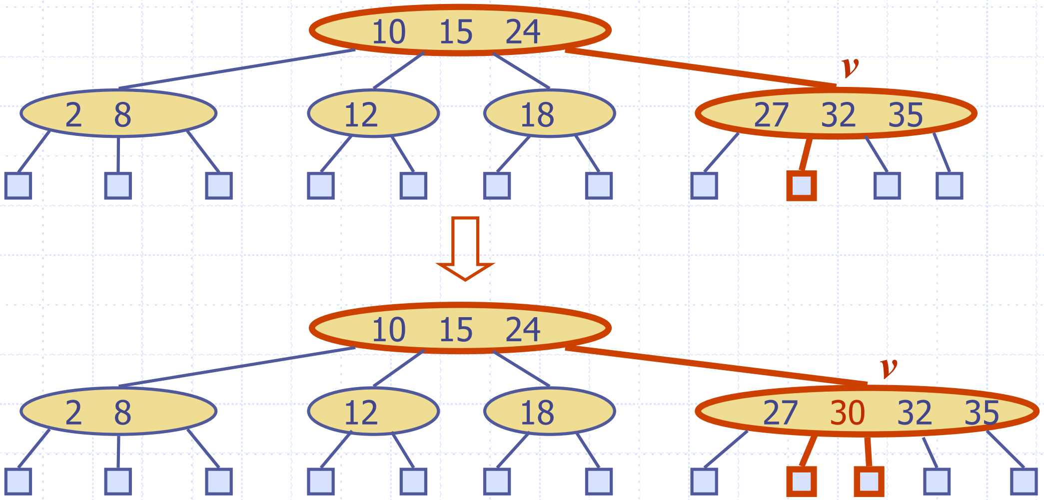
Insertion

- We insert a new item (k, o) at the parent v of the leaf 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 17



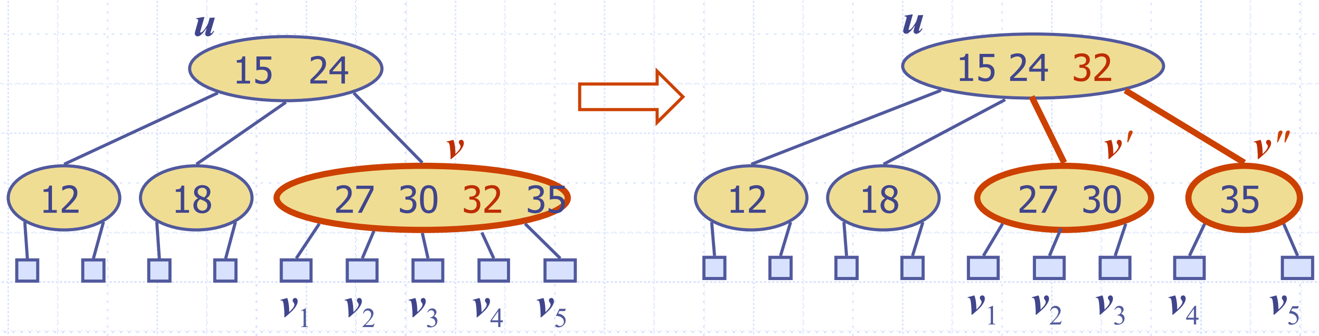
Insertion

- We insert a new item (k, o) at the parent v of the leaf 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



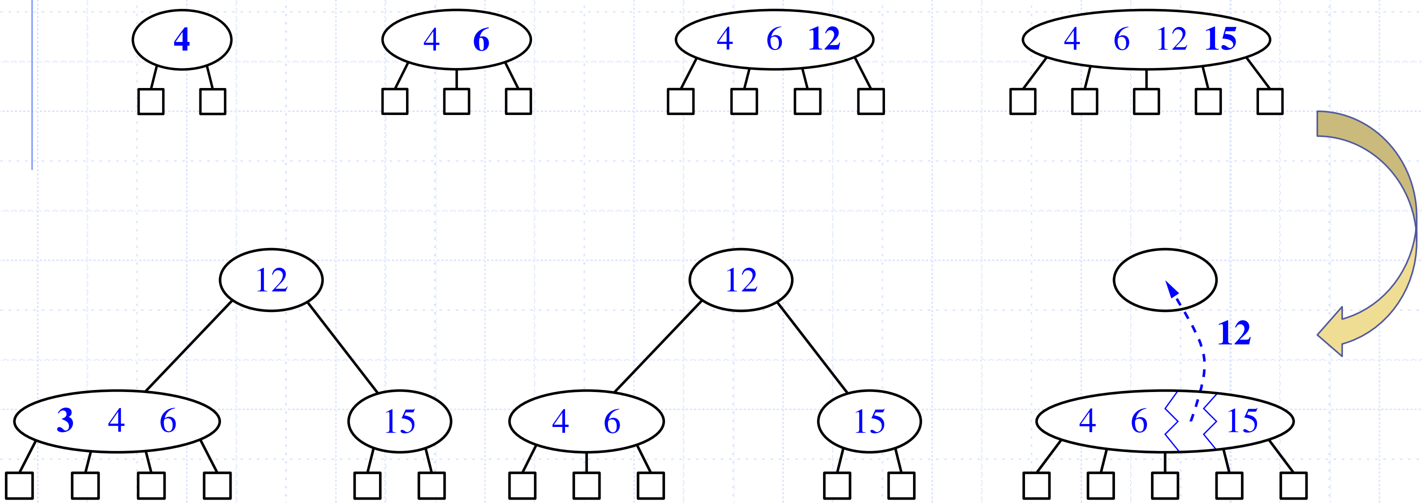
Overflow and Split

- 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 nodes v' and v''
    - v' is a 3-node with keys k_1 k_2 and children v_1 v_2 v_3
    - v'' is a 2-node with key k_4 and children v_4 v_5
  - 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



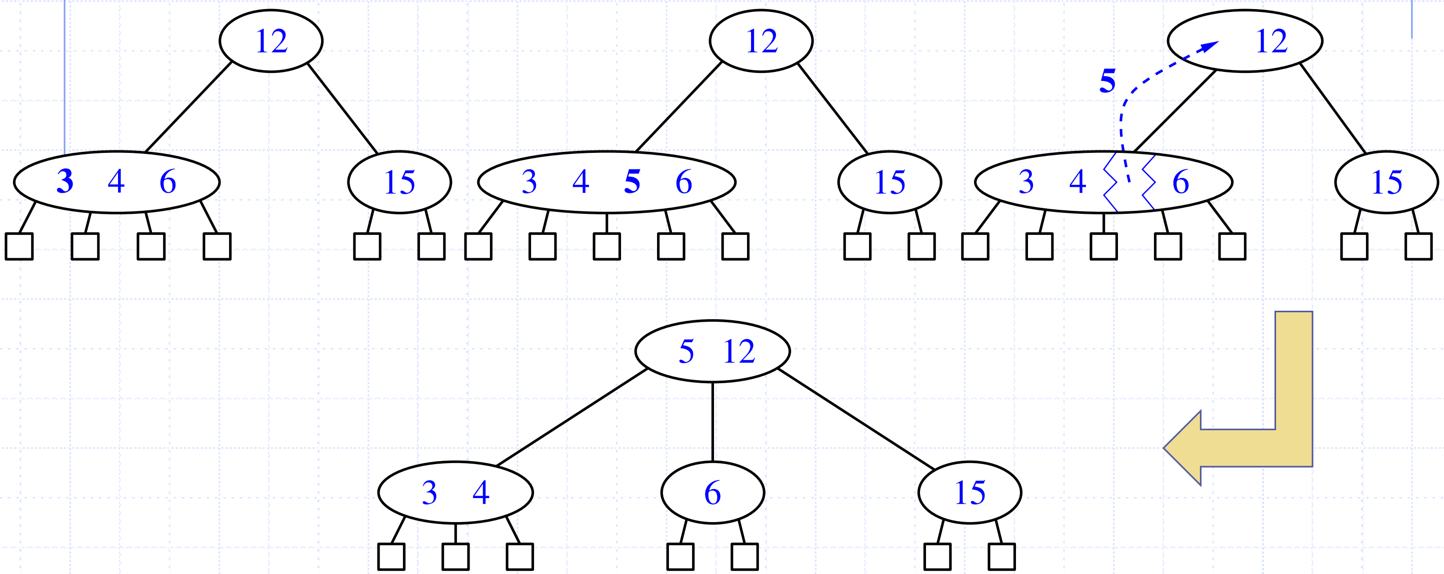
Insertion

4, 6, 12, 15, 3, 5, 10, 8, 11, 7, 13, 14, 17



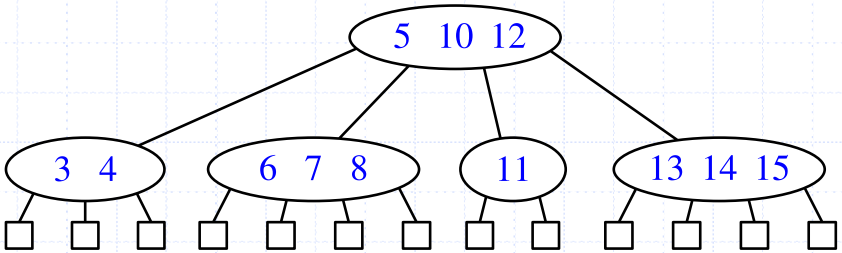
Insertion

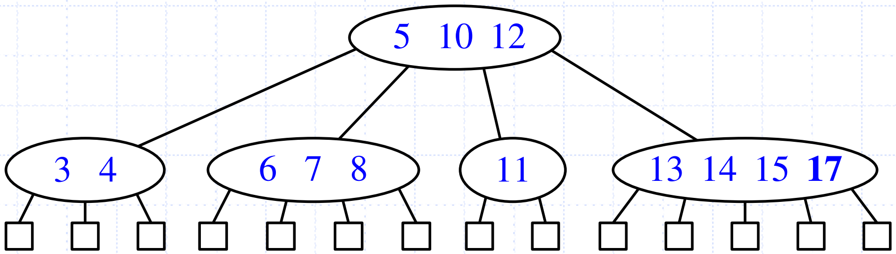
4, 6, 12, 15, 3, 5, 10, 8, 11, 7, 13, 14, 17



Insertion

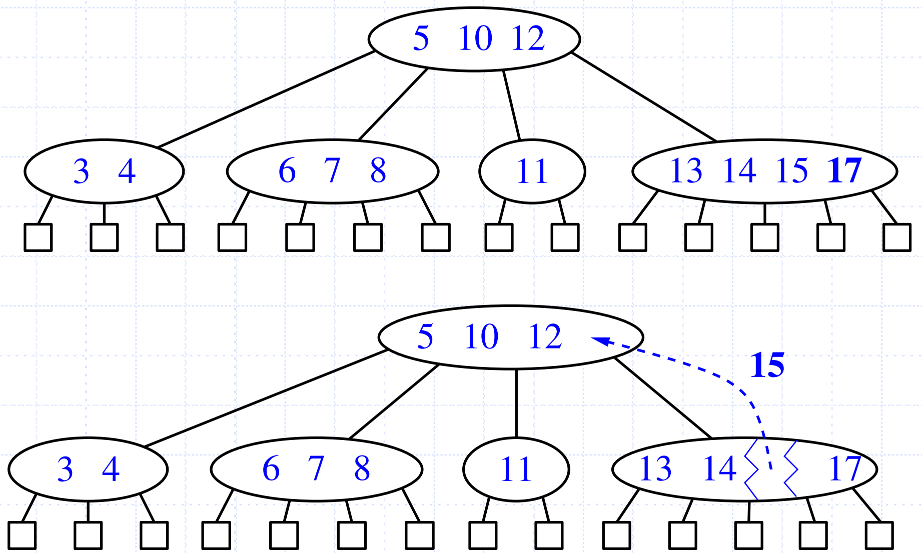
4, 6, 12, 15, 3, 5, 10, 8, 11, 7, 13, 14, **17**





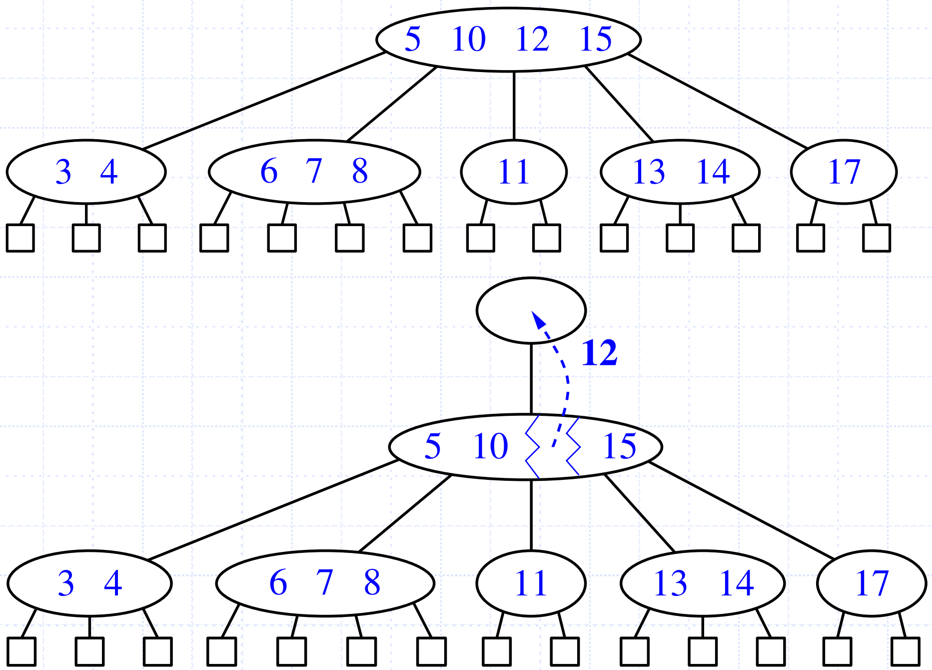
Insertion

4, 6, 12, 15, 3, 5, 10, 8, 11, 7, 13, 14, 17



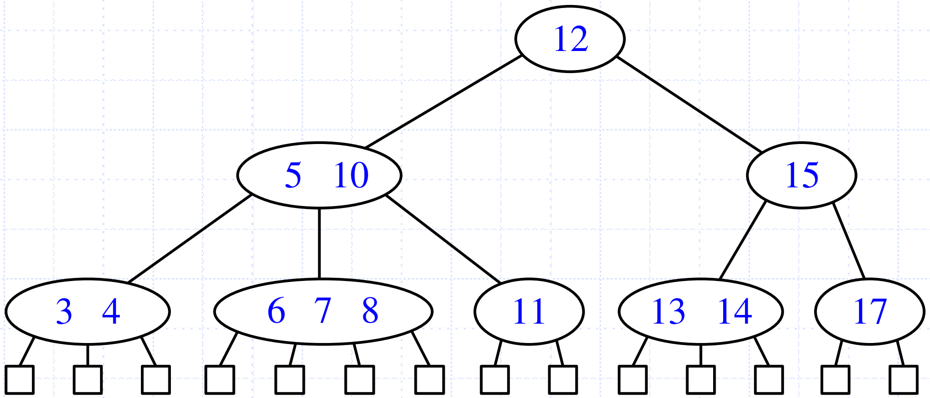
Insertion

4, 6, 12, 15, 3, 5, 10, 8, 11, 7, 13, 14, 17



Insertion

4, 6, 12, 15, 3, 5, 10, 8, 11, 7, 13, 14, 17



Analysis of Insertion

Algorithm *put*(k, o)

1. We search for key k to locate the insertion node v
2. We add the new entry (k, o) at node v
3. **while** *overflow*(v)

    if *isRoot*(v)

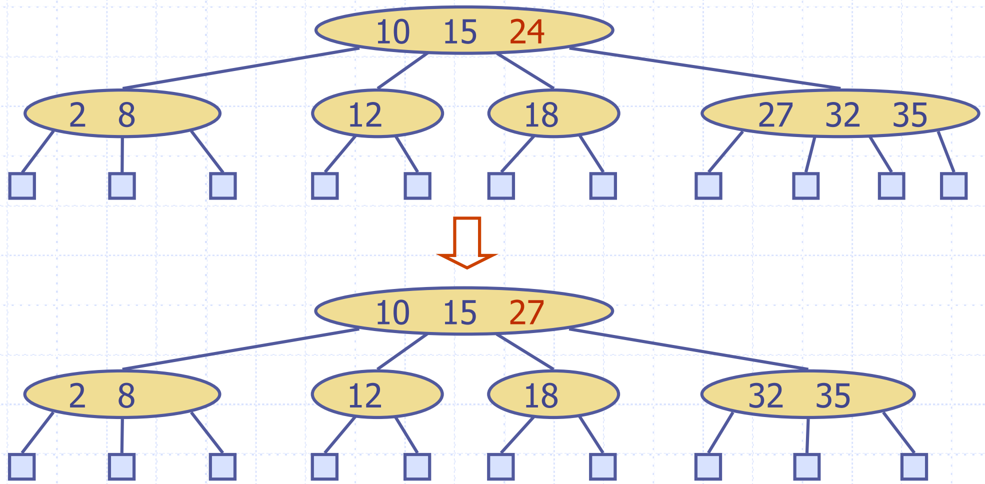
        create a new empty root above v

    $v \leftarrow$ *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

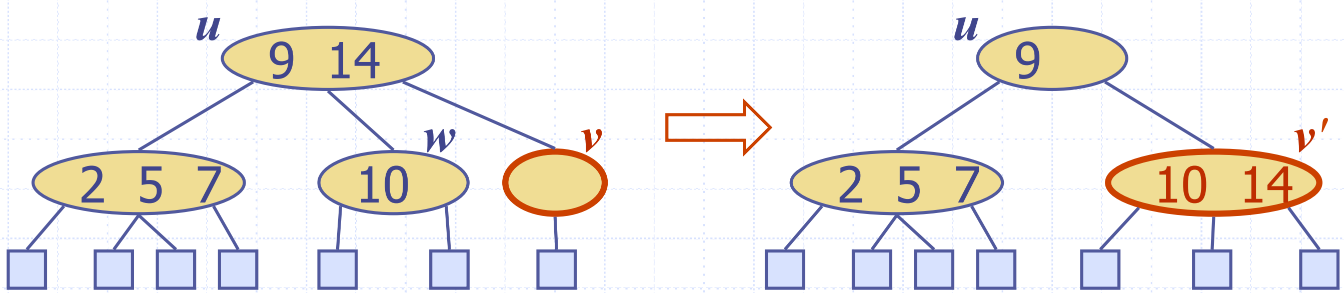
Deletion

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



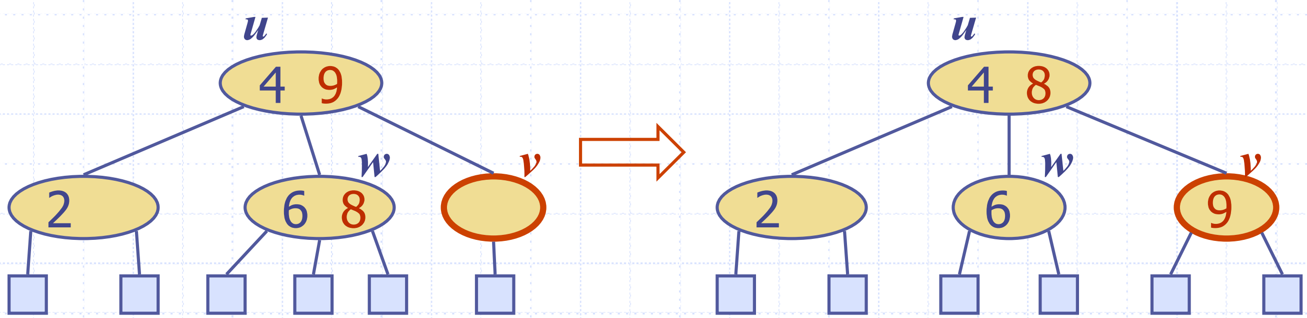
Underflow and Fusion

- 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

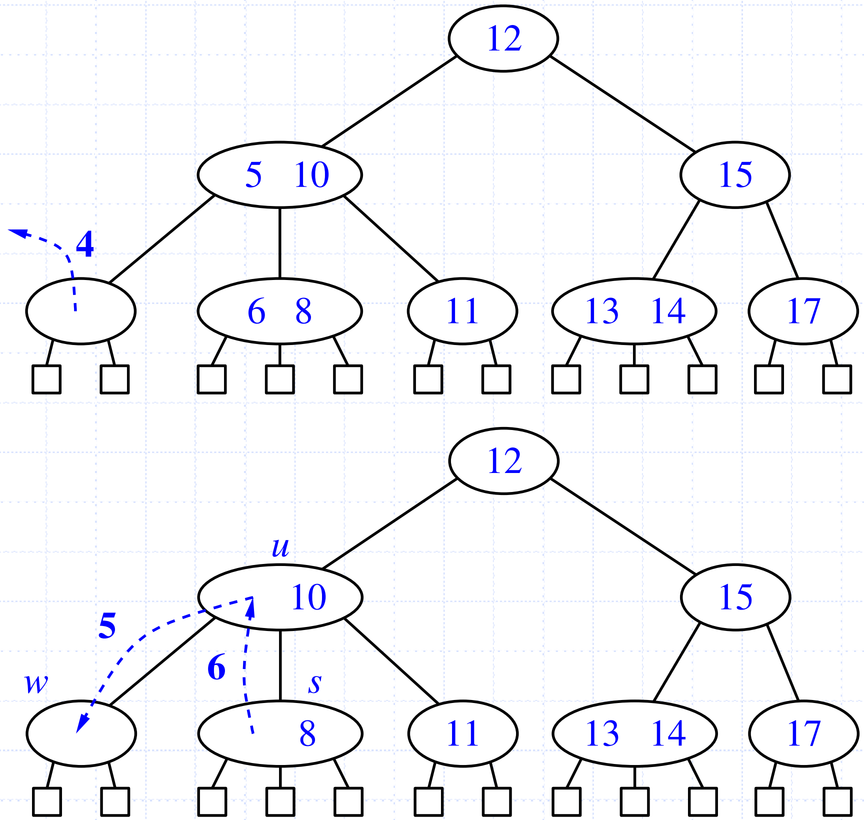


Underflow and Transfer

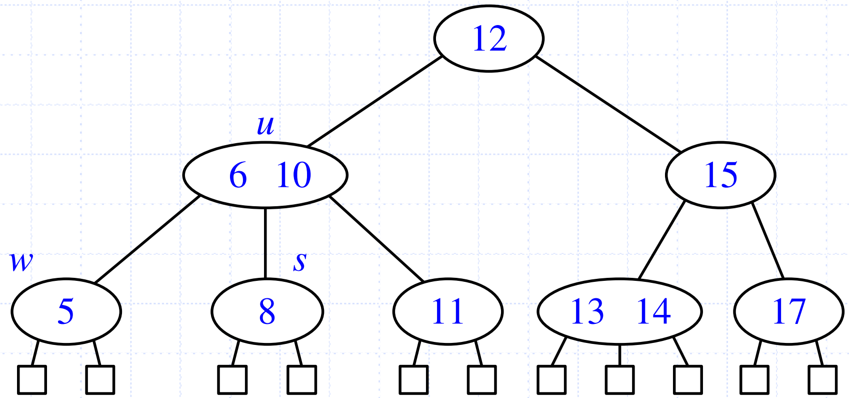
- To handle an underflow at node v with parent u, we consider two cases
- Case 2: an adjacent sibling w of v is a 3-node or a 4-node
  - Transfer operation:
    1. we move a child of w to v
    2. we move an item from u to v
    3. we move an item from w to u
  - After a transfer, no underflow occurs



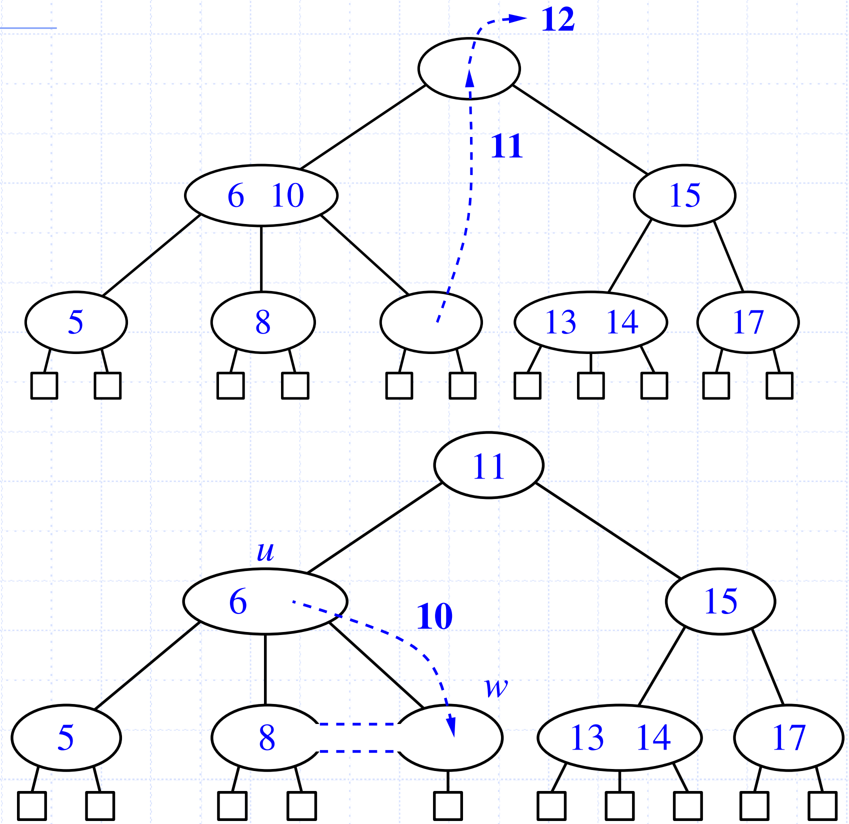
Deletion



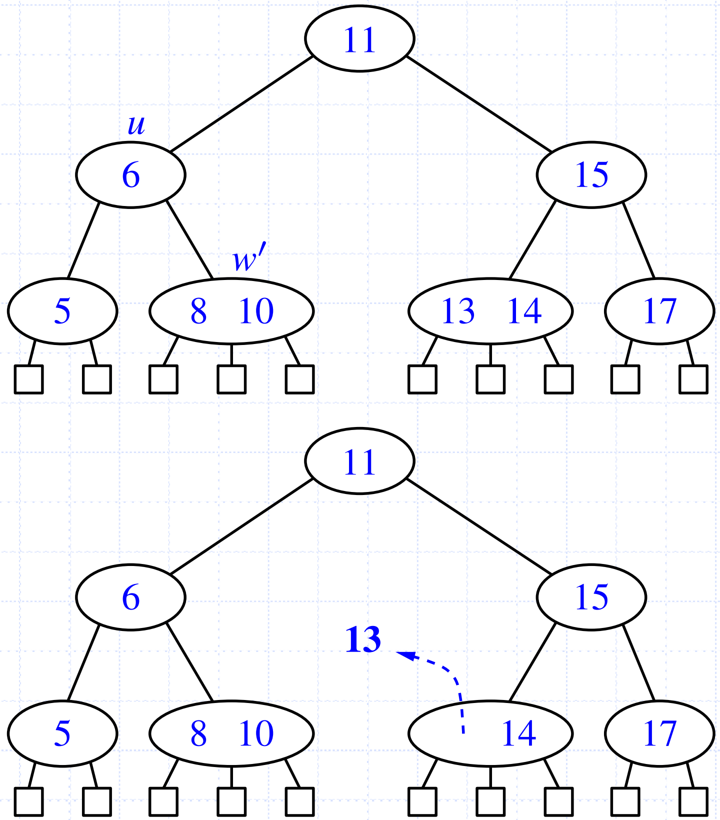
Deletion



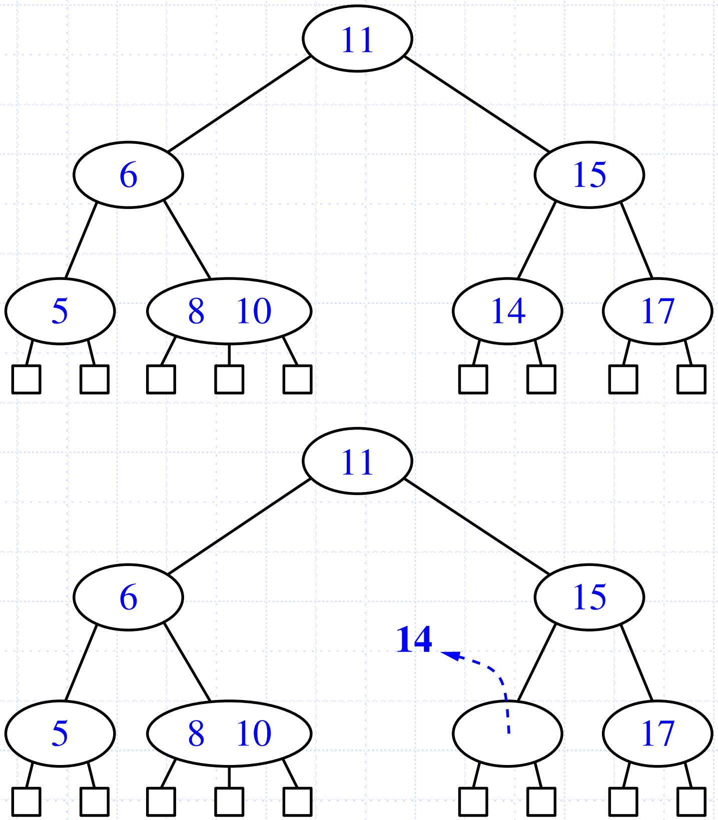
Deletion



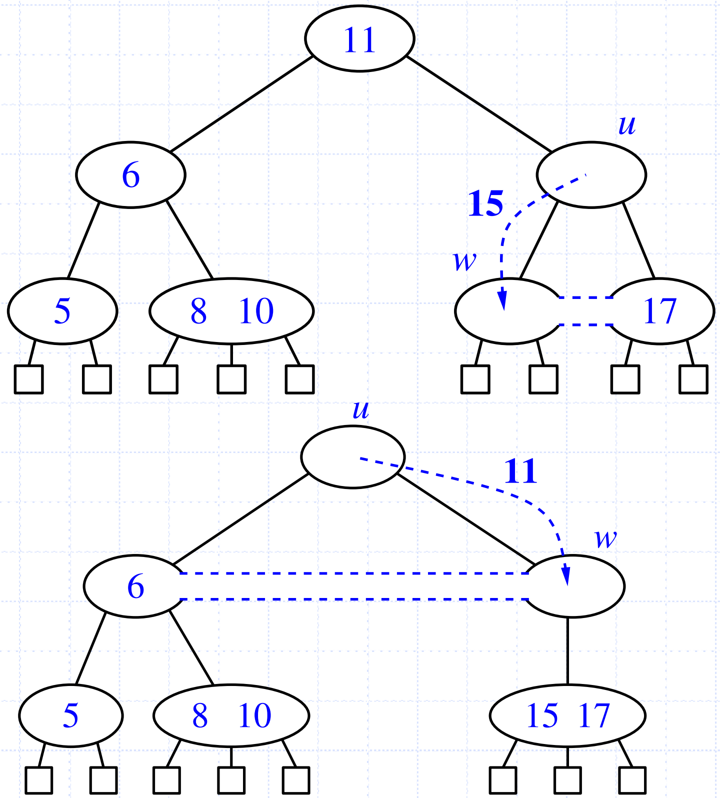
Deletion



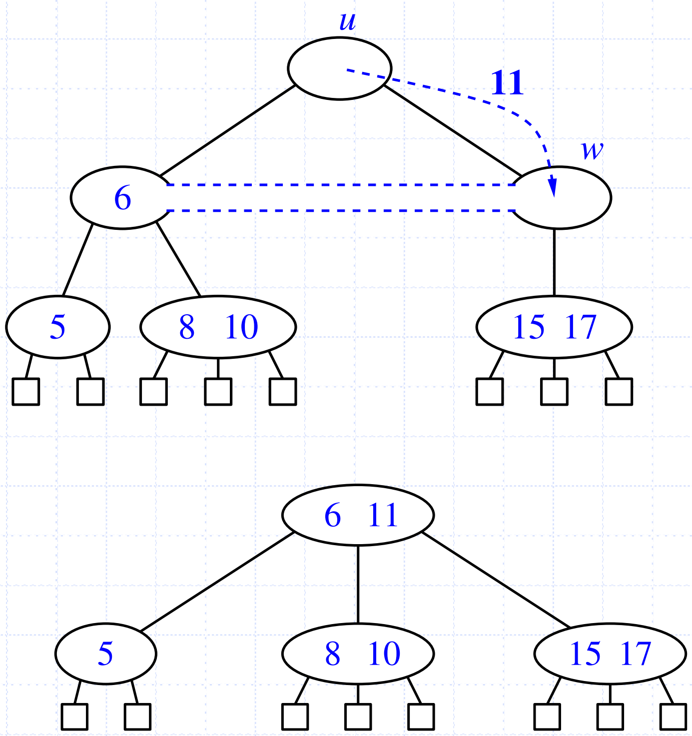
Deletion



Deletion



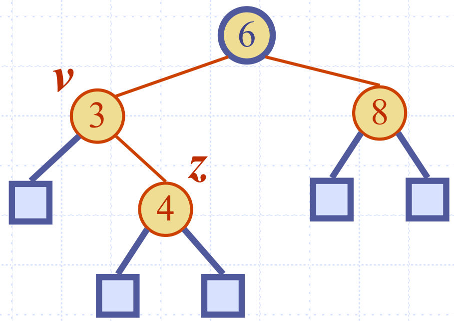
Deletion



Analysis of Deletion

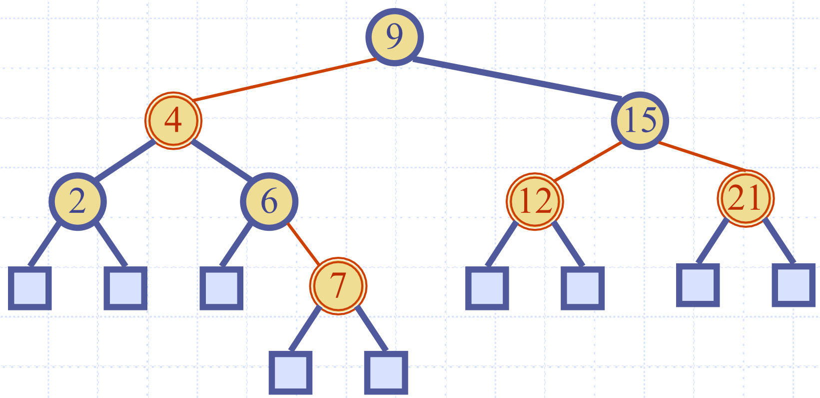
- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
- In a deletion operation
  - We visit $O(\log n)$ nodes to locate the node from which to delete the entry
  - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - Each fusion and transfer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

Red-Black Trees

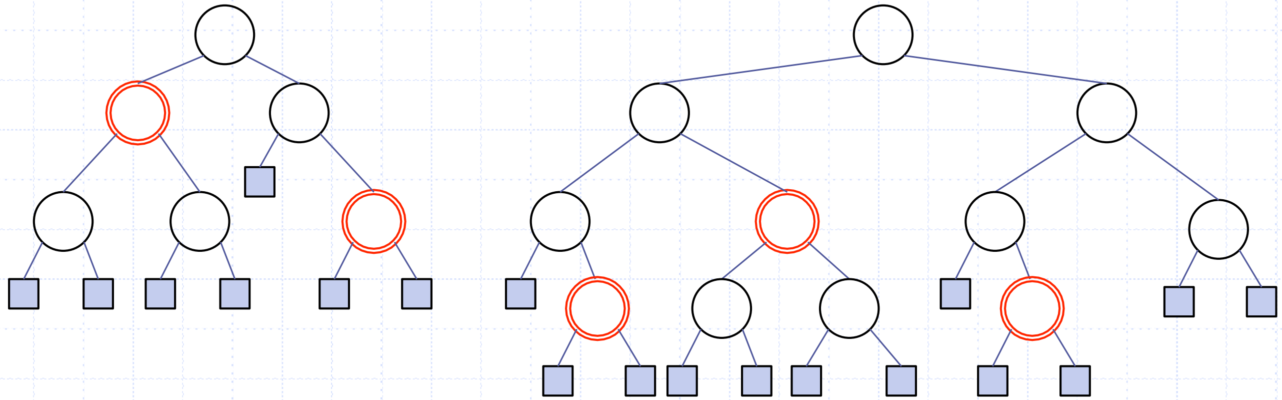


Red-Black Trees

- A red-black tree can also be defined as a binary search tree that satisfies the following properties:
  - Root Property: the root is black
  - External Property: every leaf is black
  - Internal Property: the children of a red node are black
  - Depth Property: all the leaves have the same black depth



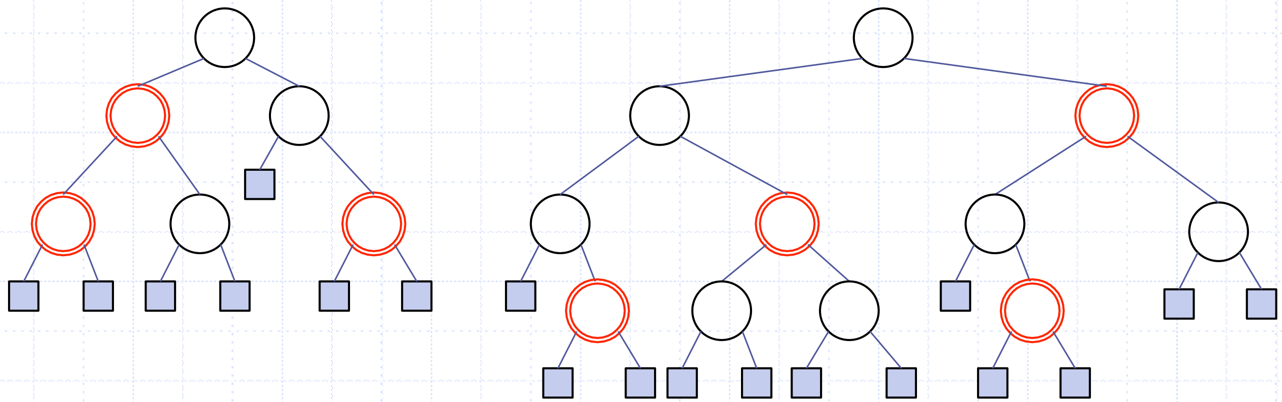
Examples of Red-Black Trees



Black Depth - 2

Black Depth - 3

Examples of Not Red-Black Trees



Double Red

Black Depth not uniform

Height of a Red-Black Tree

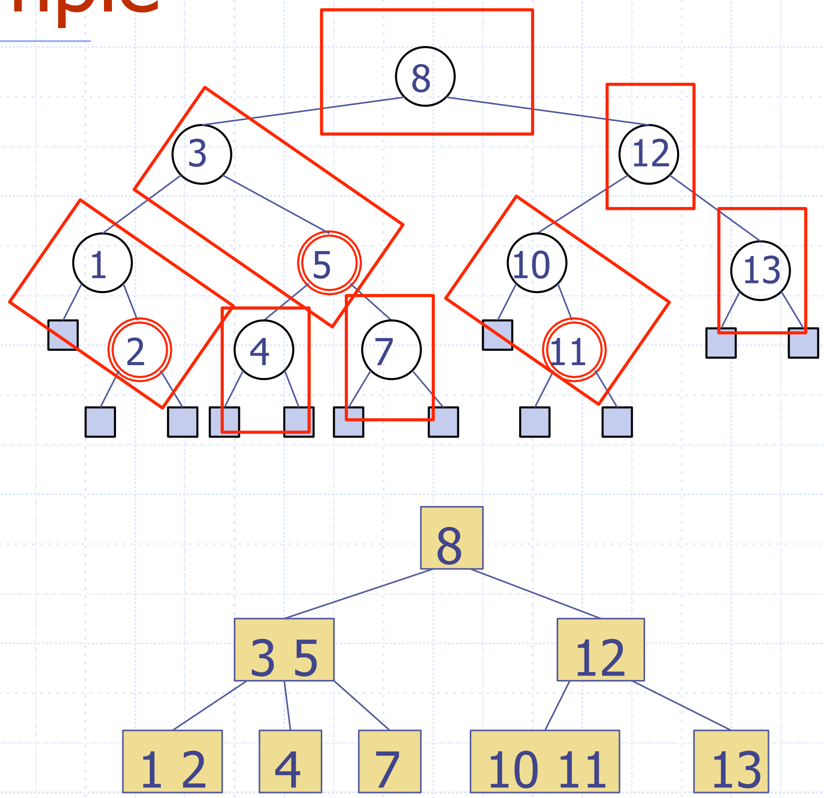
- Theorem: A red-black tree storing n items has height $O(\log n)$

  Proof:
  - Let h be the black depth of a red-black tree on n nodes
  - n is smallest when all the nodes are black.
    - complete binary tree of height h and $n=2^h-1$
  - n is largest the tree alternates between red and black nodes at every level.
    - height of the tree – $2h$ and $n=2^{2h}-1$
  - Hence $\log_4 n < h < 1+\log n$
- The search algorithm for a red-black 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

Red-Black Trees to 2-4 Trees

- Any red-black tree can be converted into a 2-4 tree
- Take a black node and its red children (at most 2) and combine them into one node of a 2-4 tree.
- Each node thus formed has at least 1 and at most 3 keys
- Since black depth of all external nodes is the same, in the resulting 2-4 tree all the external nodes will be at the same level.

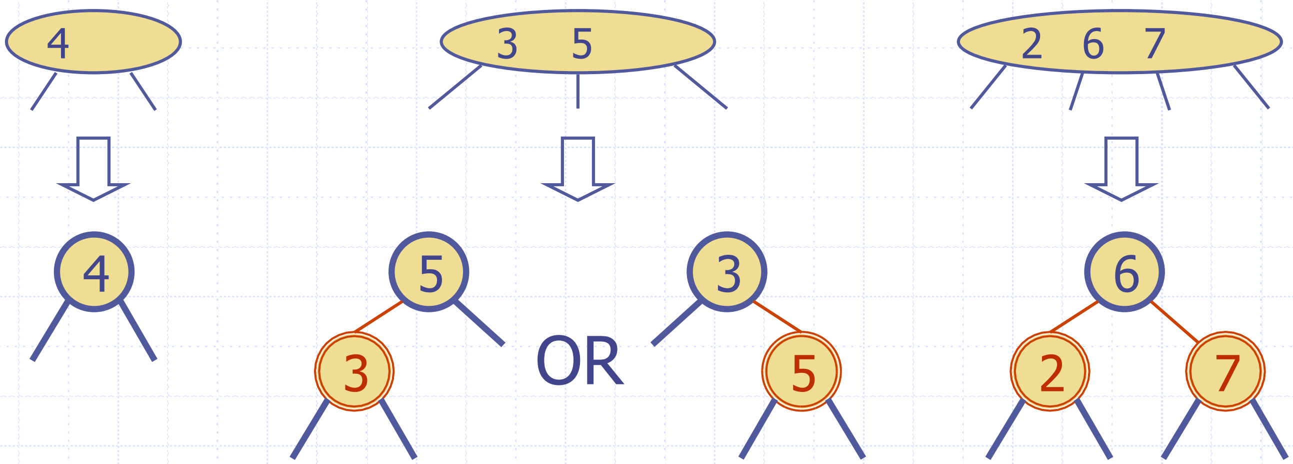
Red-Black Tree to 2-4 Tree - Example



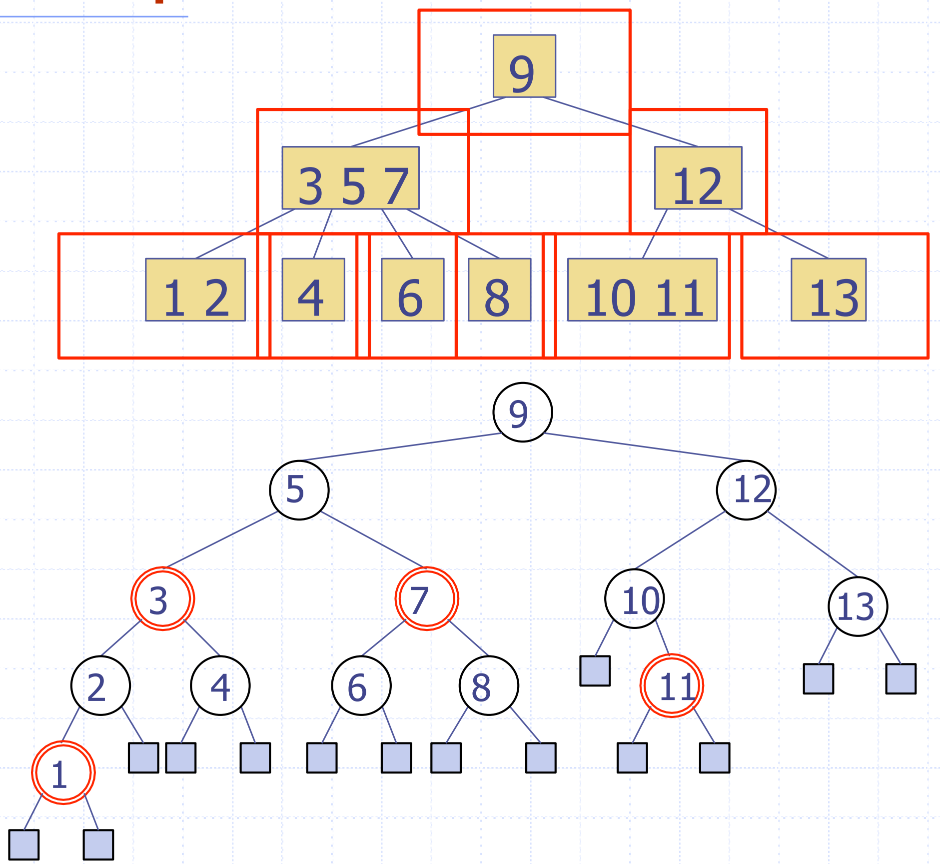
2-4 Trees to Red-Black Trees

- Any 2-4 tree can be converted into a red-black tree
- We replace a node of the 2-4 tree with one black node and 0/1/2 red nodes which are children of the black node.
- The height of 2-4 tree is the black depth of the red-black tree created.
- Every red node has a black child.

2-4 Tree to Red-Black Trees



2-4 Trees to Red-Black Trees - Example

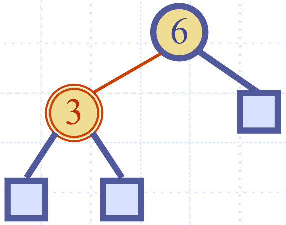


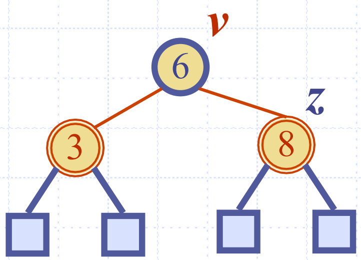
Insertion in Red-Black Trees

- To insert (k, o), we execute the insertion algorithm for binary search trees.
  - search for k; this gives us the place where we have to insert k.
- We create a new node with key k and insert it at this place.
- The new node is colored red (unless it is the root).

Insertion

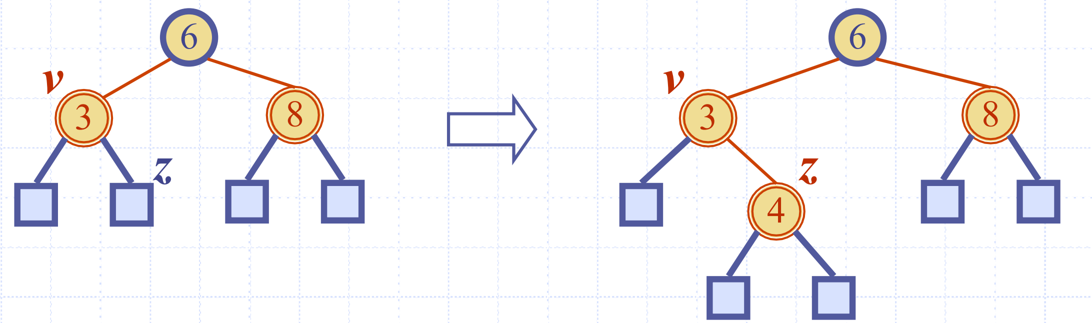
- We preserve the root, external, and depth properties
- If the parent v of z is black, we also preserve the internal property and we are done.
- Example – insert 8





Insertion

- We preserve the root, external, and depth properties
- If the parent v of z is black, we also preserve the internal property and we are done
- Else (v is red) we have a double red (i.e., a violation of the internal property), which requires a reorganization of the tree
- Example where the insertion of 4 causes a double red:

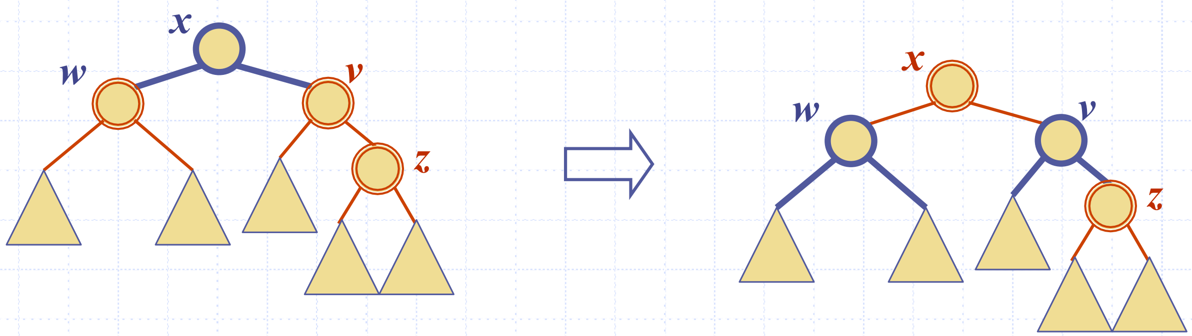


Remedying a Double Red

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

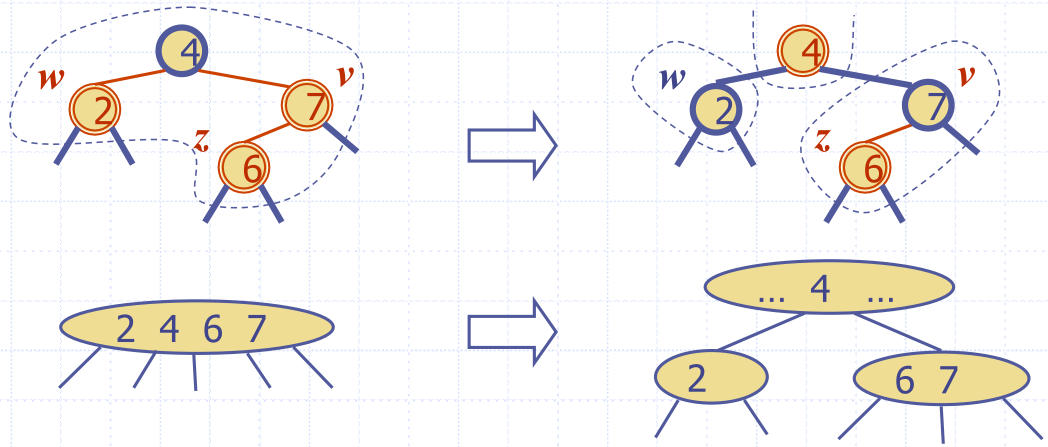
Case 1: w is red

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



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

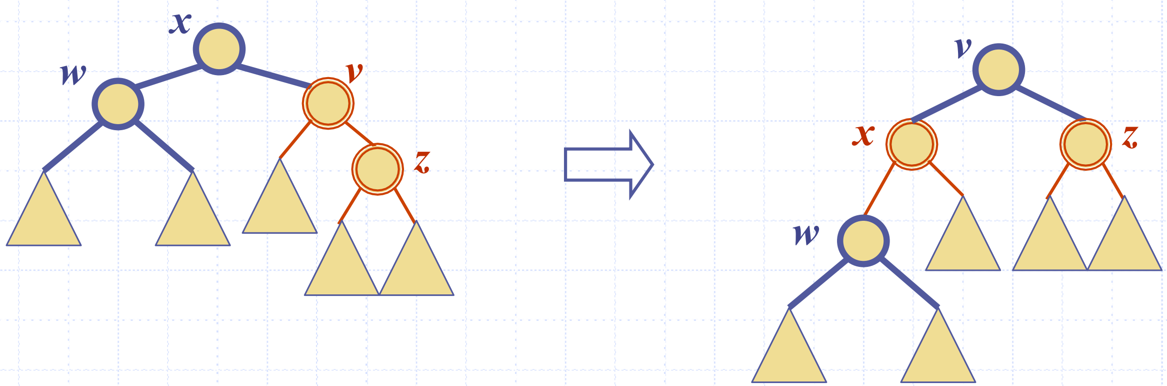


Remedying a Double Red

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

Case 2: w is black

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

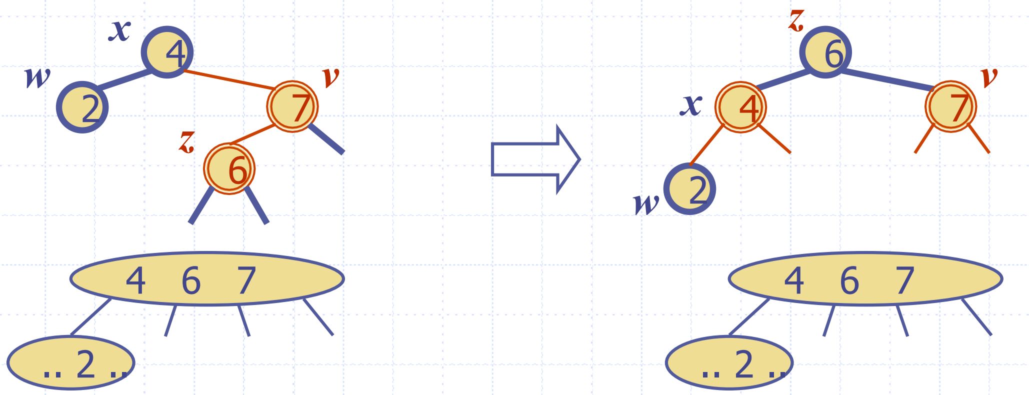


Remedying a Double Red

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

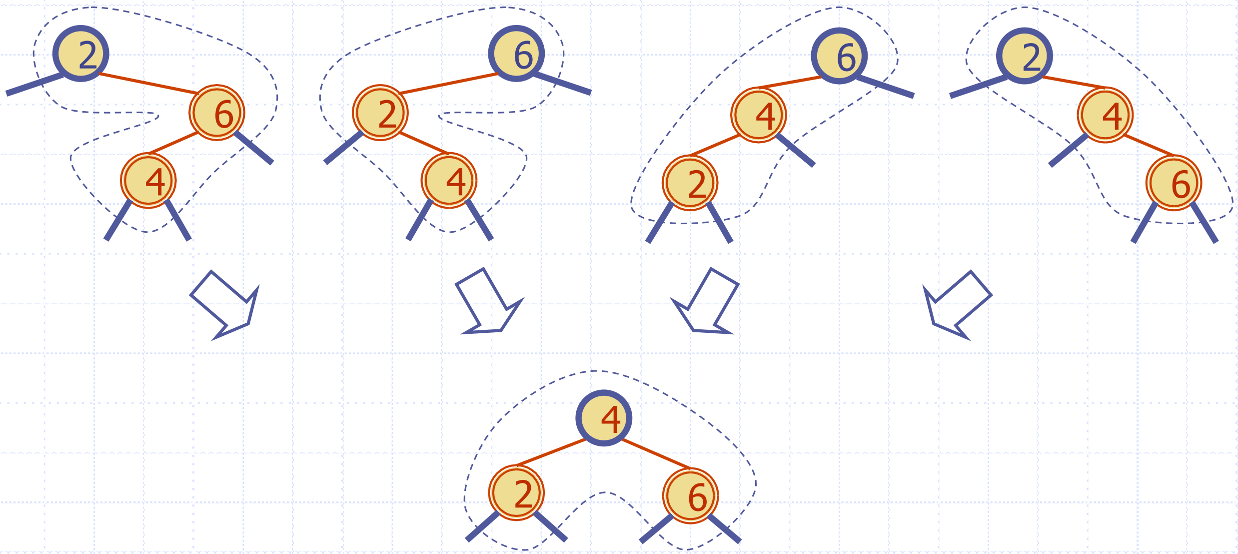
Case 1: w is black

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

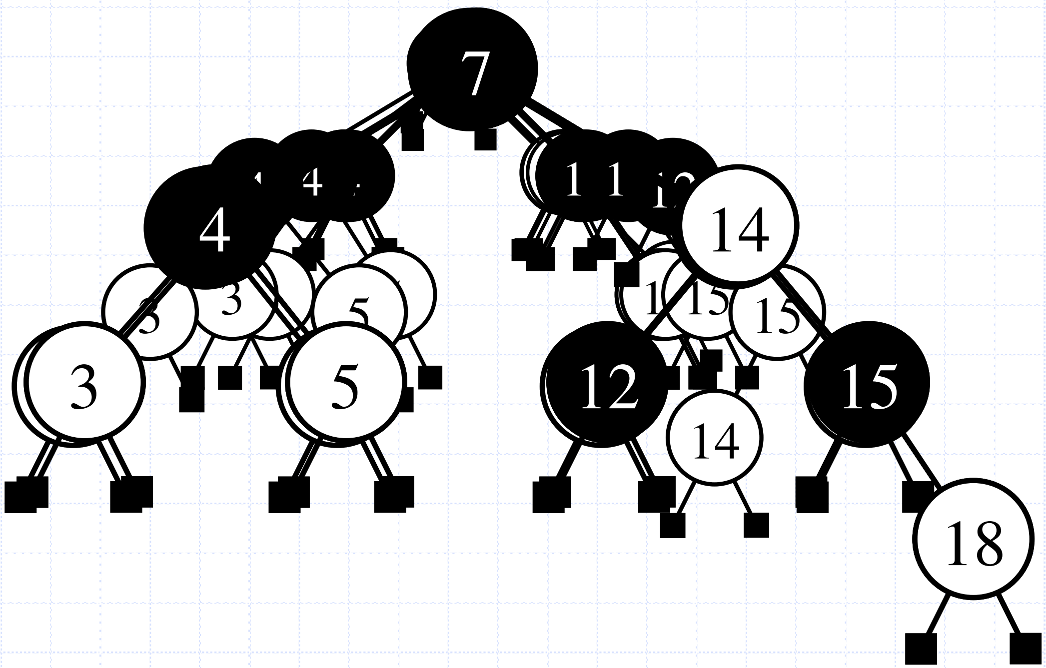


Restructuring (cont.)

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



Insertion - Example



Analysis of Insertion

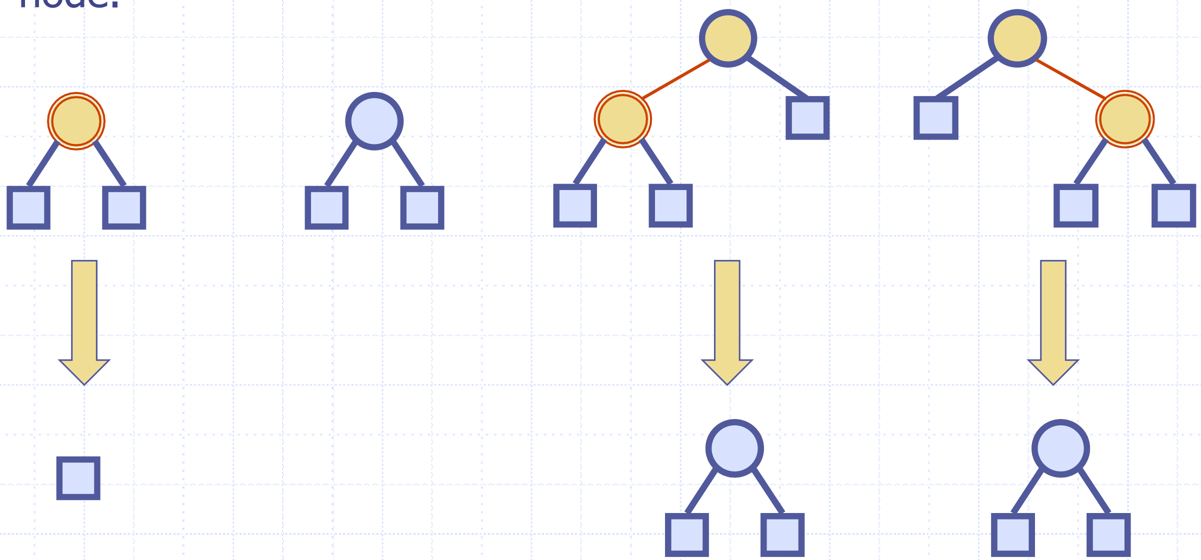
- Recall that a red-black tree has $O(\log n)$ height
- Step 1 takes $O(\log n)$ time because we visit $O(\log n)$ nodes
- Step 2 takes $O(1)$ time
- Step 3 takes $O(\log n)$ time because we perform
  - $O(\log n)$ recolorings, each taking $O(1)$ time, and
  - at most one restructuring taking $O(1)$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

Algorithm *insert*(k, o)

1. We search for key k to locate the insertion node z
2. We add the new entry (k, o) at node z and color z red
3. **while** *doubleRed*(z)
   if *isBlack*(*sibling*(*parent*(z)))
   $z \leftarrow$ *restructure*(z)
   return
   else { *sibling*(*parent*(z) is red }
   $z \leftarrow$ *recolor*(z)

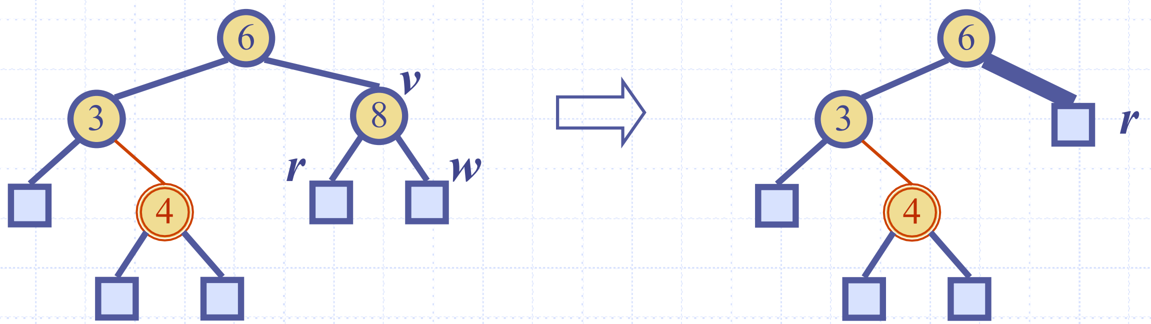
Deletion

- To perform operation remove(k), we first execute the deletion algorithm for binary search trees
- Thus the node which is deleted is the parent of an external node.

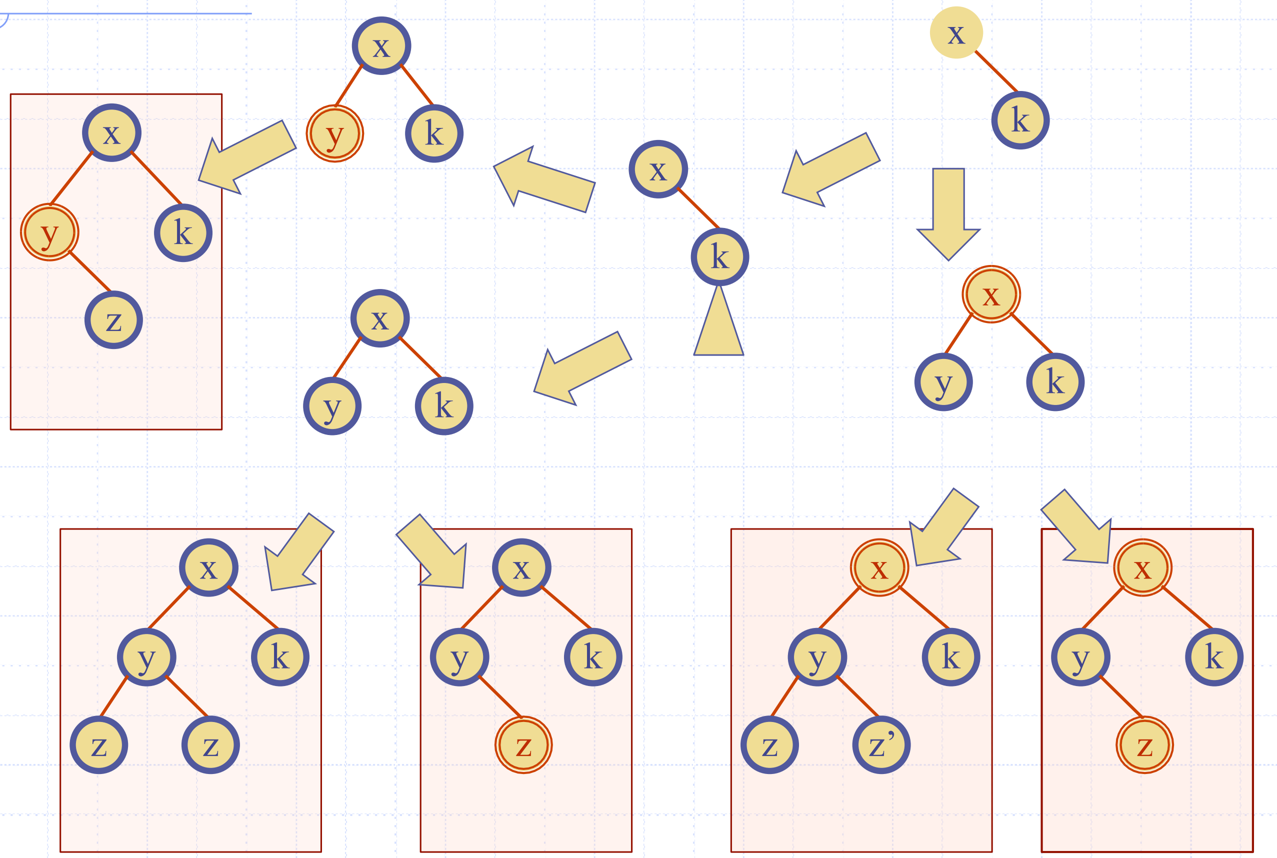


Deletion

- Consider the case when the node to be deleted is black and has two external children – **double black**
- Removing the node reduces the black depth of an external node by 1.
- Hence, in a general step, we consider how to reorganize the tree when the height (black depth) of a subtree reduces by 1.
- Example where the deletion of 8 causes a **double black**:

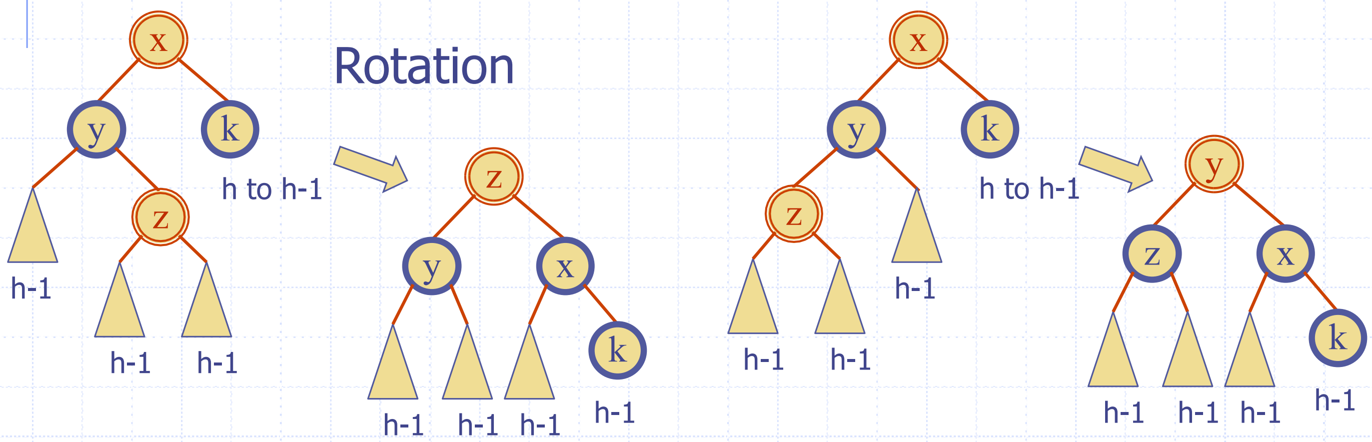


Deletion - cases



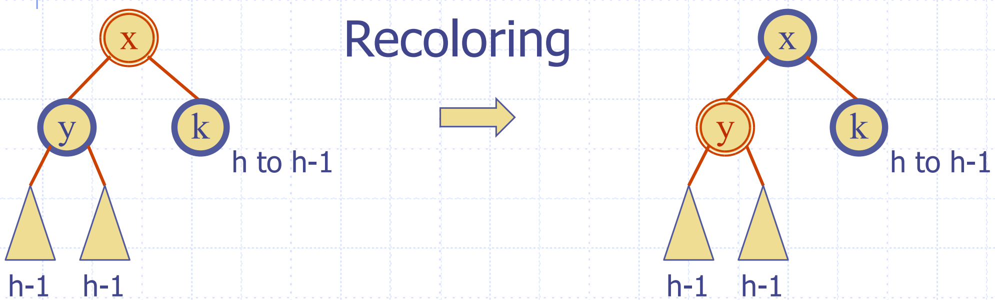
Deletion – case 1

- The parent of k, x is red.
- The sibling of k, y is black.
- At least one child of y, z is red.



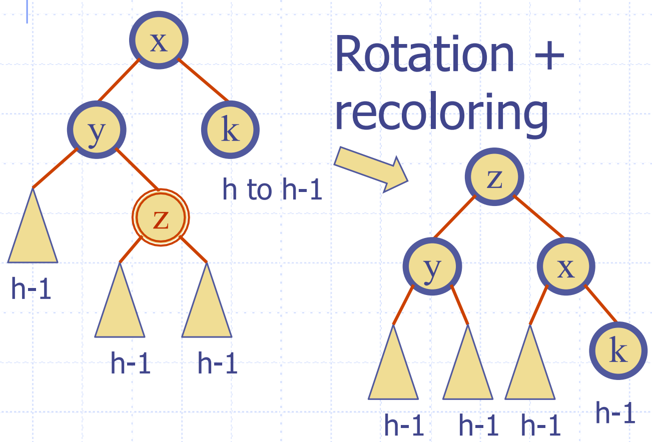
Deletion – case 2

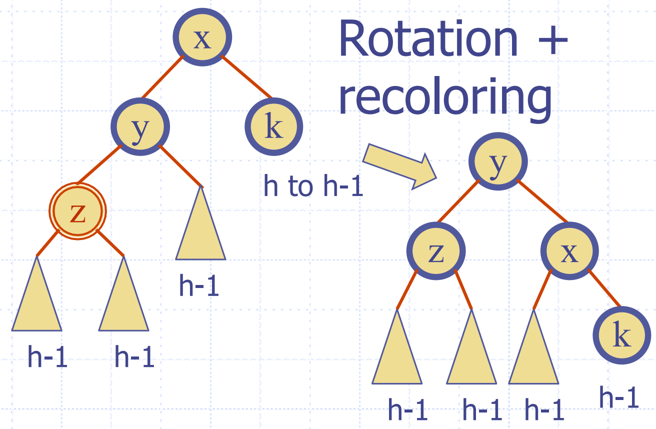
- The parent of k, x is red.
- The sibling of k, y is black.
- Both the children of y are black.



Deletion – case 3

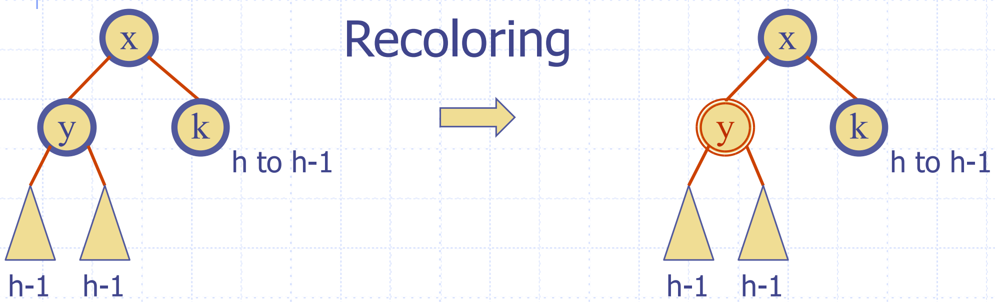
- The parent of k, x is black.
- The sibling of k, y is black.
- At least one child of y, z is red.





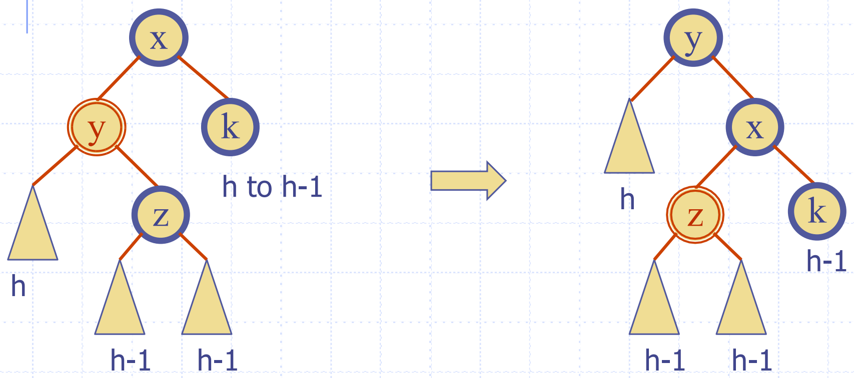
Deletion – case 4

- The parent of k, x is black.
- The sibling of k, y is black.
- Both the children of y are black.



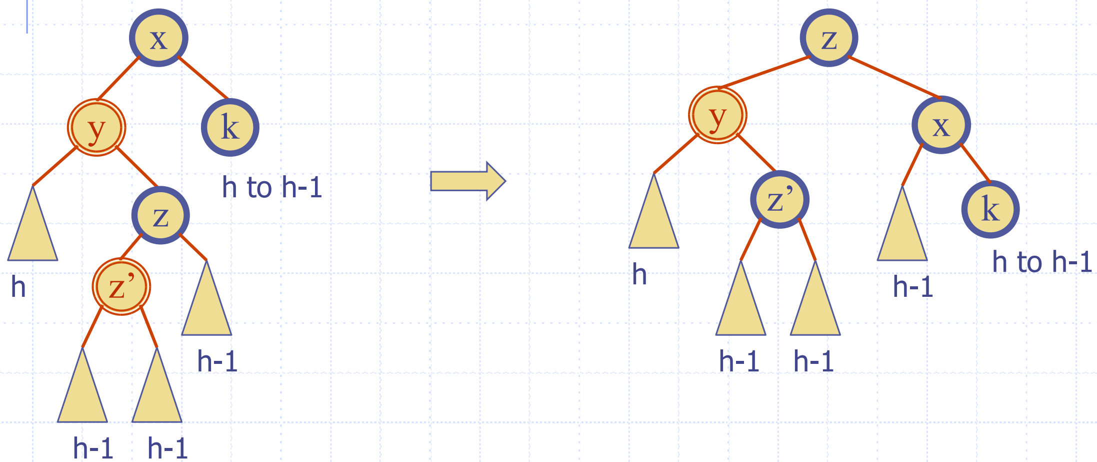
Deletion – case 5.1

- The parent of k, x is black.
- The sibling of k, y is red.
- Both the children of z are black.



Deletion – case 5.2

- The parent of k, x is black.
- The sibling of k, y is red.
- At least one child of z is red.



Deletion – Summary

- In all cases, except 4, deletion can be completed by a simple rotation/ recoloring.
- In case 4, the height of the subtree reduces and so we need to proceed up the tree
  - If we proceed up the tree, we only need to recolor/rotate.
- Complexity- O(log n)

Problems

- Write a recursive procedure to convert a binary search tree to a doubly linked list in place.

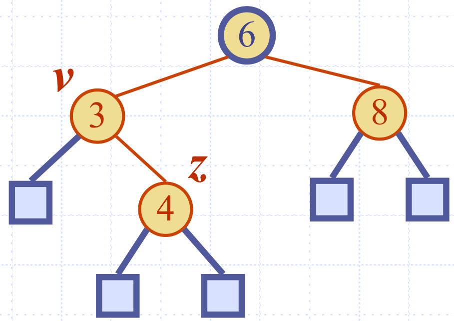
Problems

- Given two binary trees (not necessarily balanced), write an algorithm that merges the two given trees into a balanced search tree in linear time.

Problems

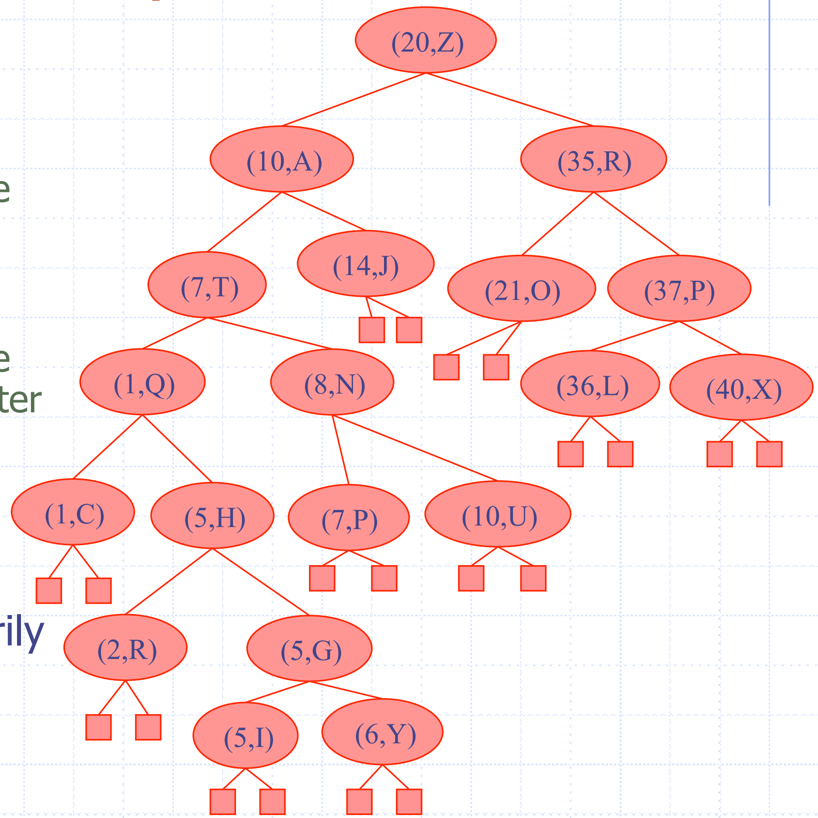
- Given a balanced binary search tree of n nodes and a target sum, write a function that returns true, if there is a pair that adds up to the sum, otherwise returns false.

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
- A Splay tree is not necessarily balanced.

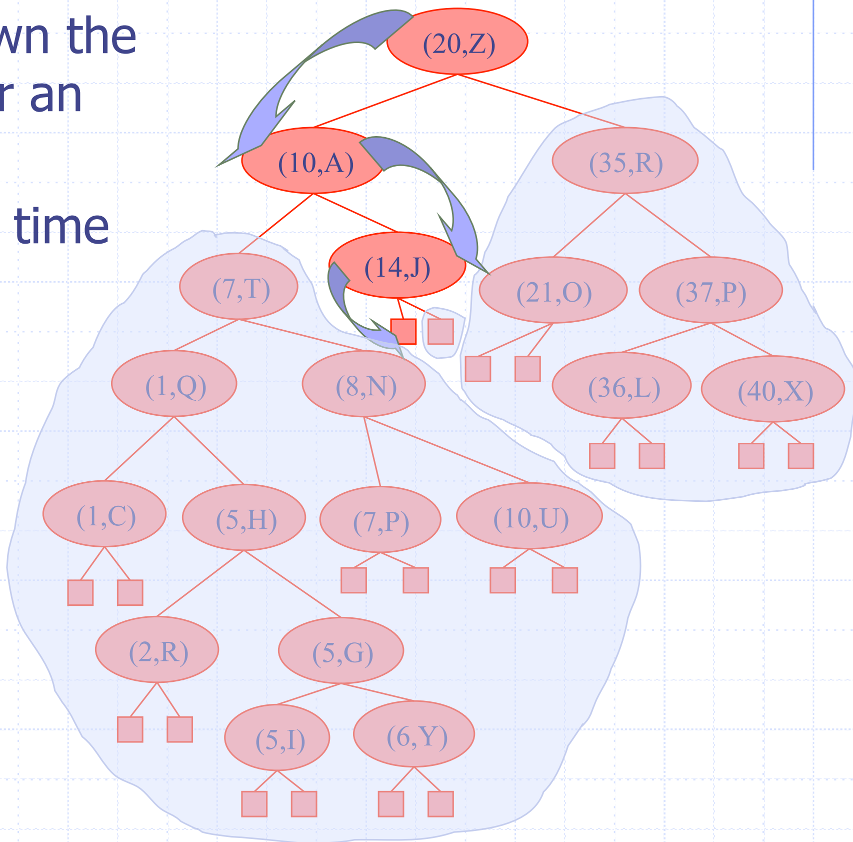


Slide by Matt Dickerson

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

Slide by Matt Dickerson

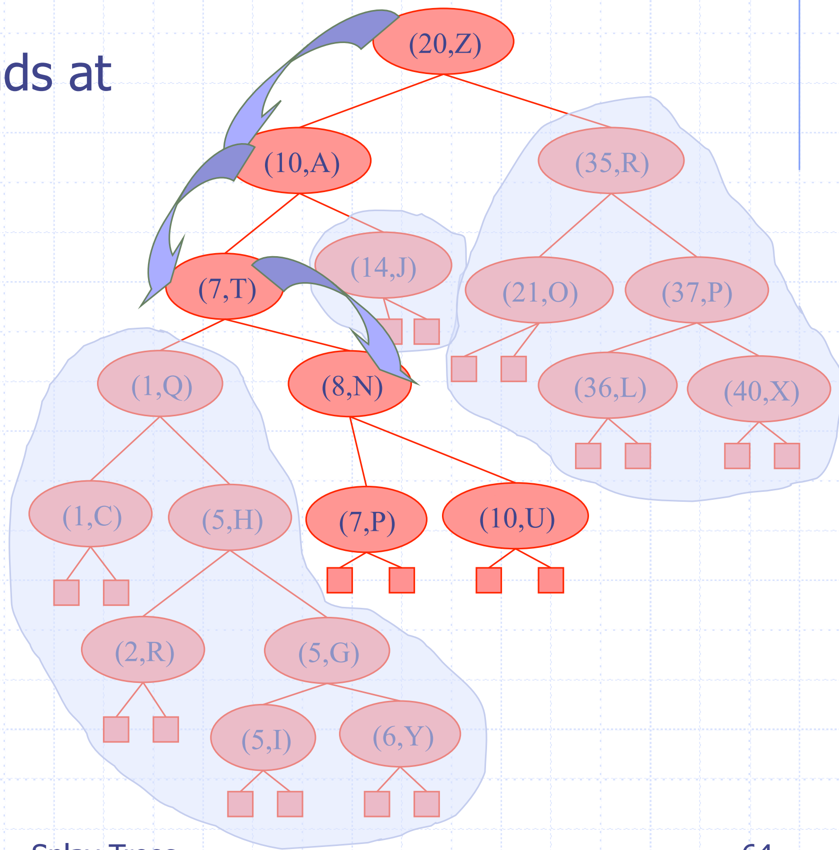
- Search proceeds down the tree to found item or an external node.
- Example: Search for time with key 11.



Example Searching in a BST, continued

Slide by Matt Dickerson

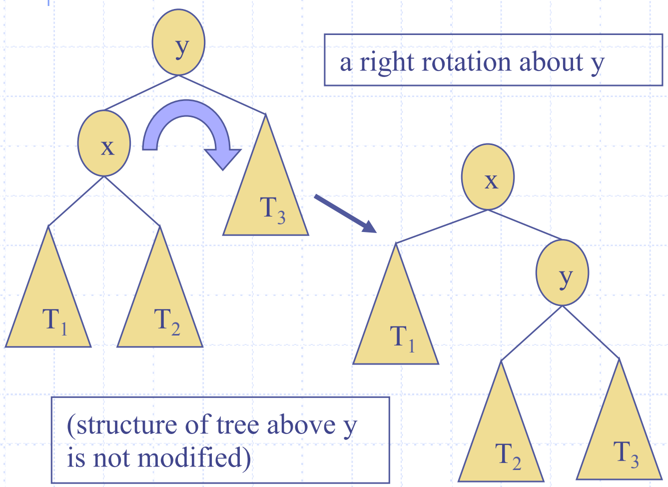
- search for key 8, ends at an internal node.



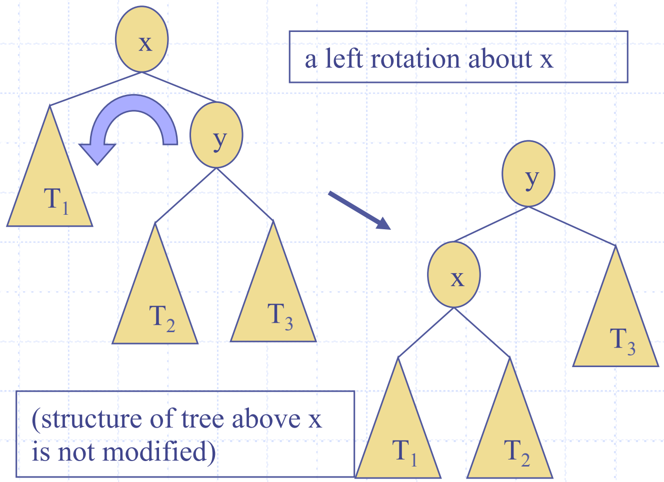
Splay Trees do Rotations after Every Operation (Even Search)

- 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*

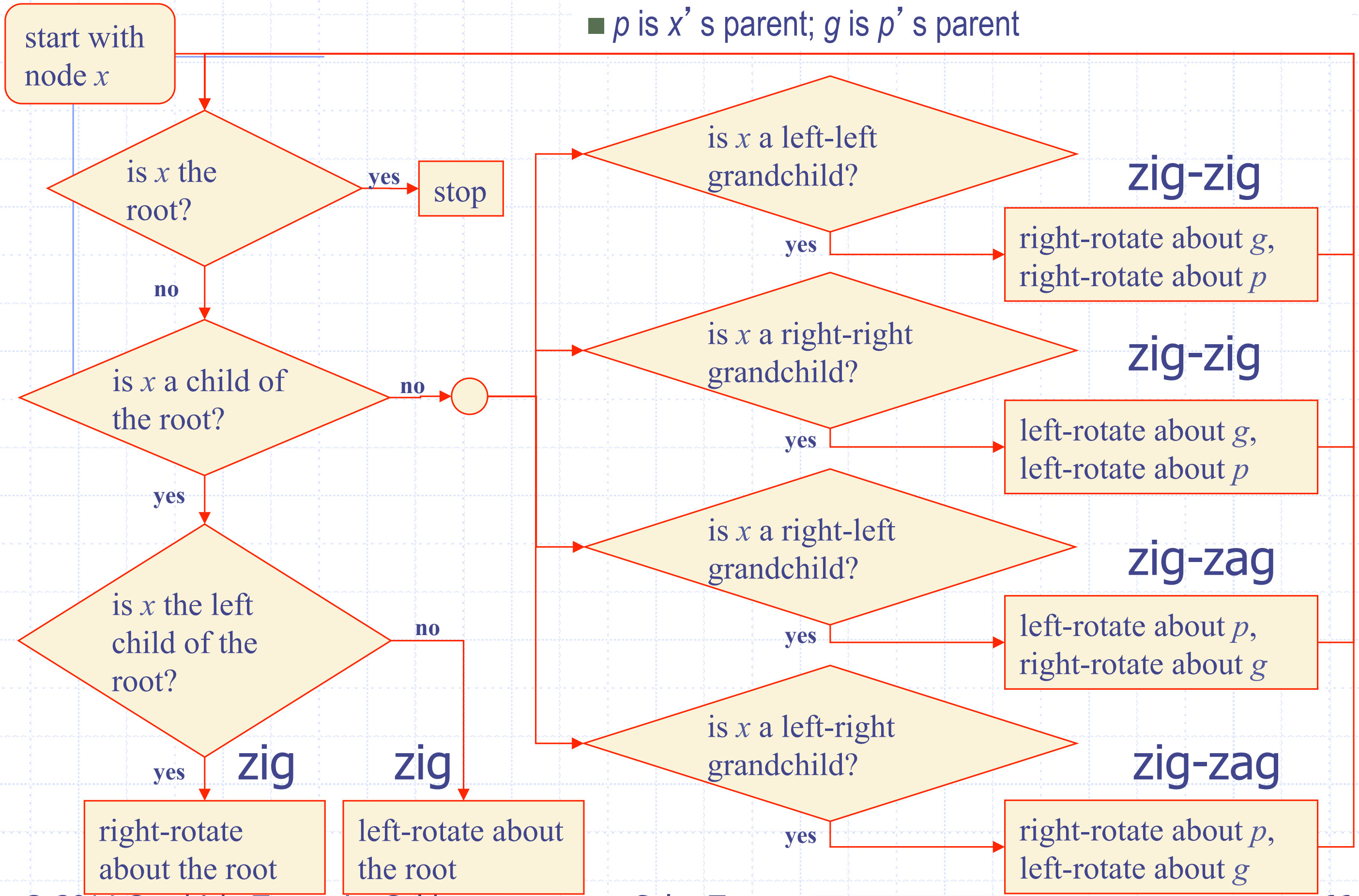


- left rotation
  - makes the right child *y* of a node *x* into *x*' s parent; *x* becomes the left child of *y*

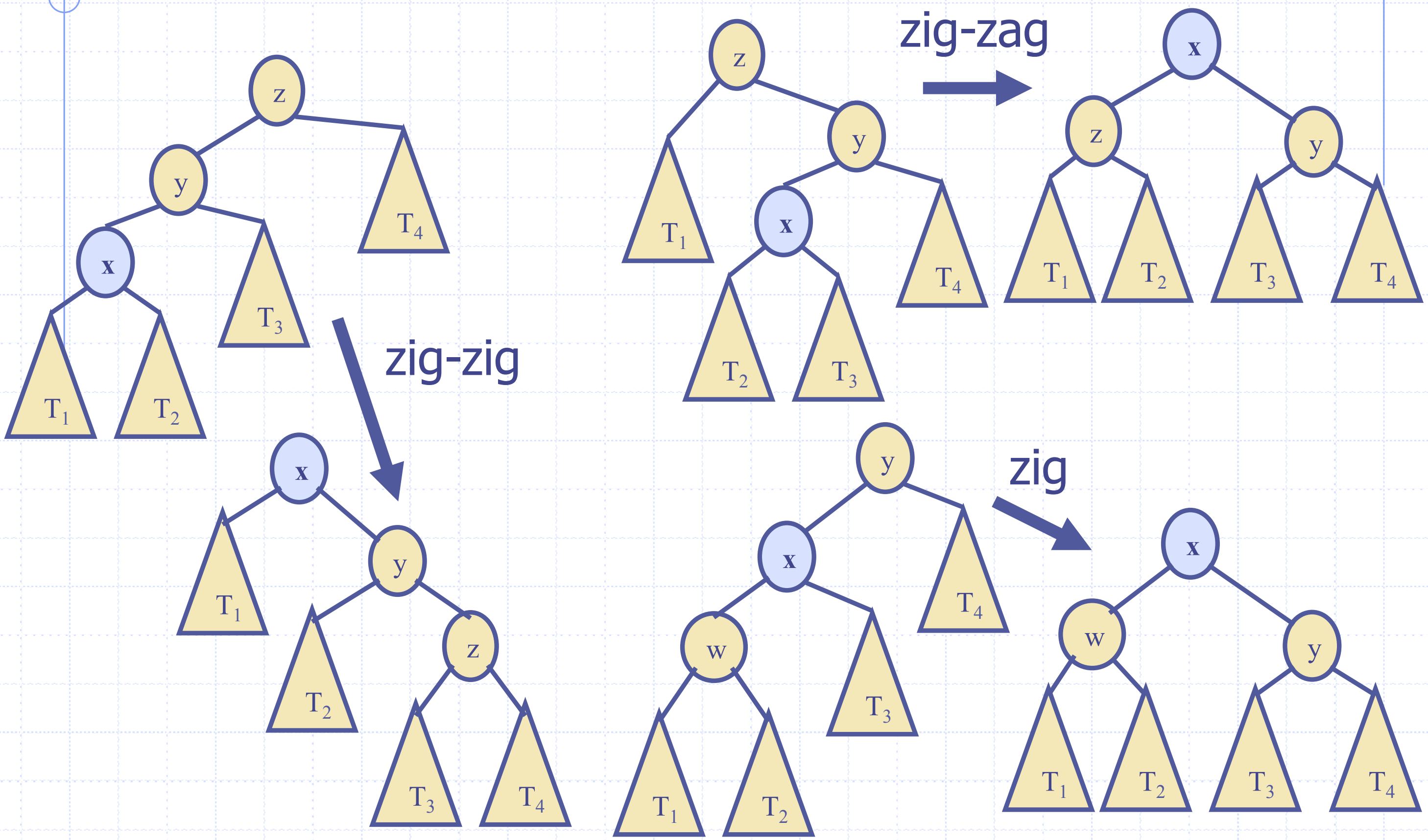


Splaying:

- "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

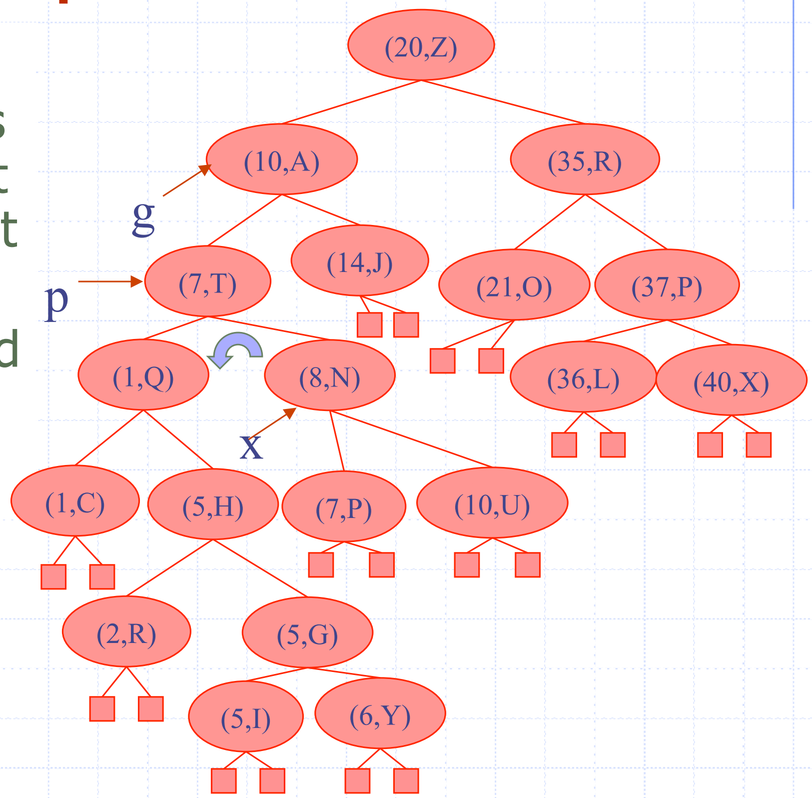


Visualizing the Splaying Cases



Splaying Example

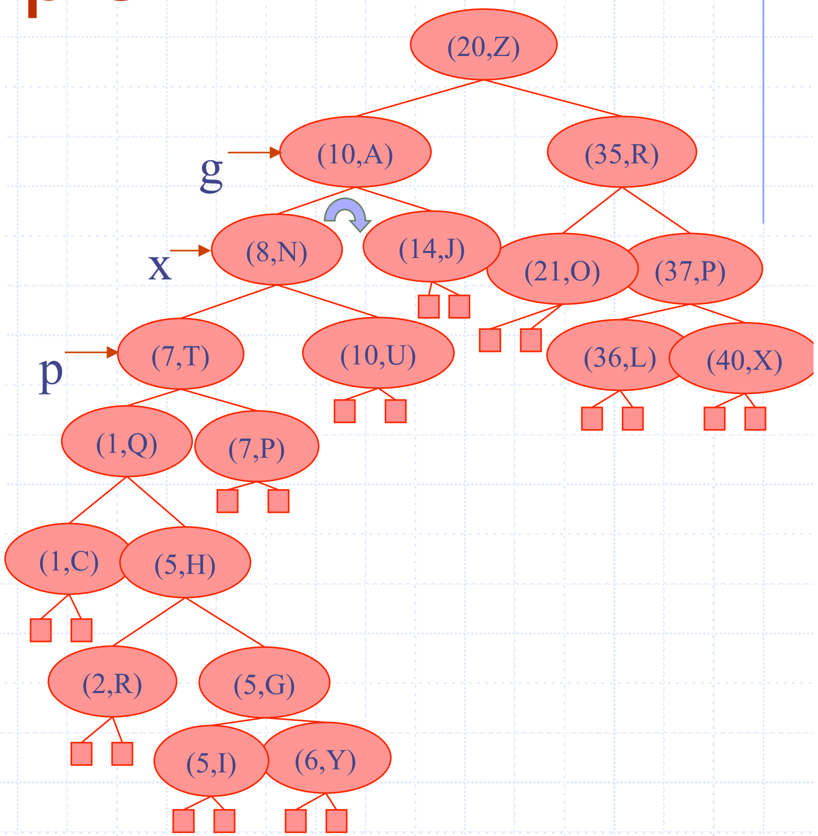
- let x = (8,N)
  - x is the right child of its parent, which is the left child of the grandparent
  - left-rotate around p, then right-rotate around g



 Slide by Matt Dickerson

Splaying Example

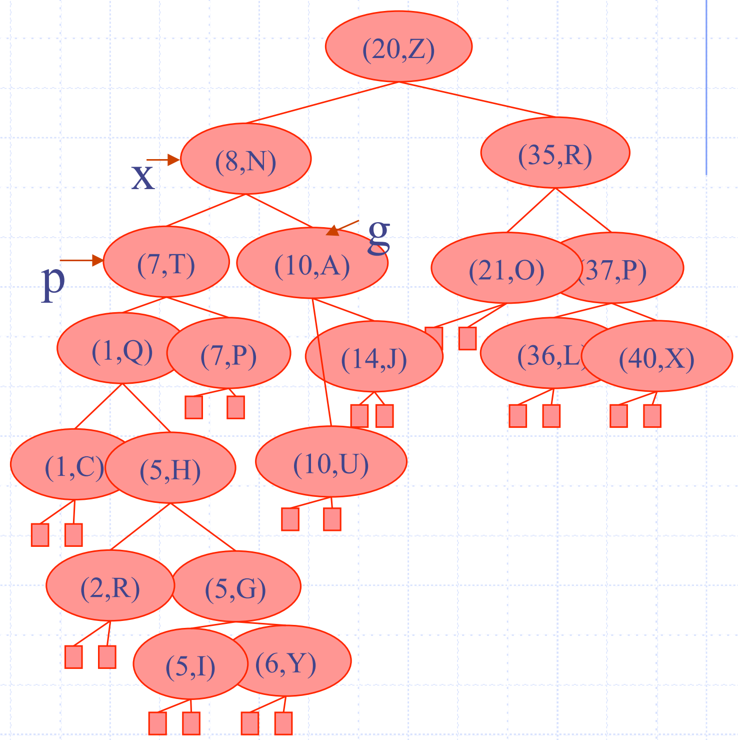
- let x = (8,N)
  - x is the right child of its parent, which is the left child of the grandparent
  - left-rotate around p, then right-rotate around g



Slide by Matt Dickerson

Splaying Example

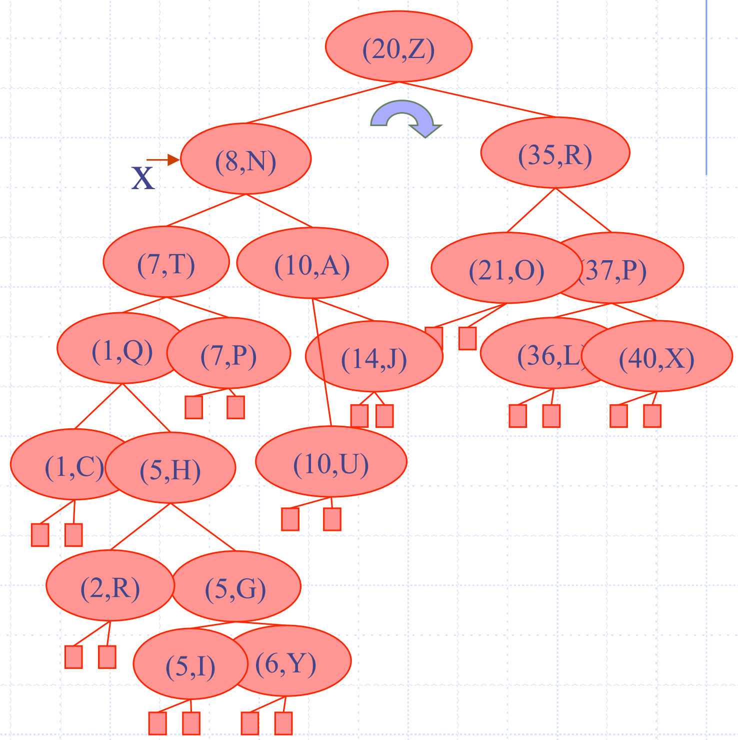
- let x = (8,N)
  - x is the right child of its parent, which is the left child of the grandparent
  - left-rotate around p, then right-rotate around g



Slide by Matt Dickerson

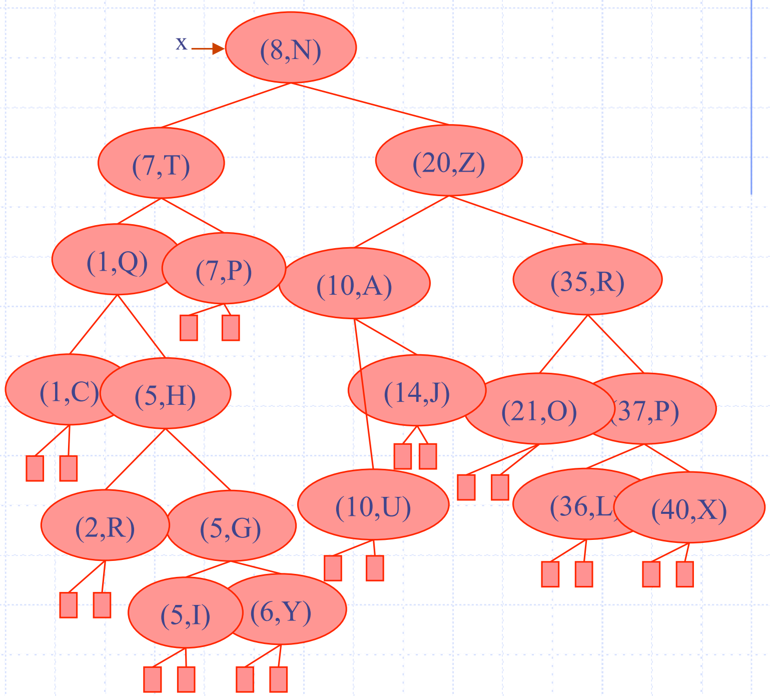
Splaying Example

- let *x* = (8,N)
  - *x* is the left child of the root
  - right-rotate around *root.*



Splaying Example

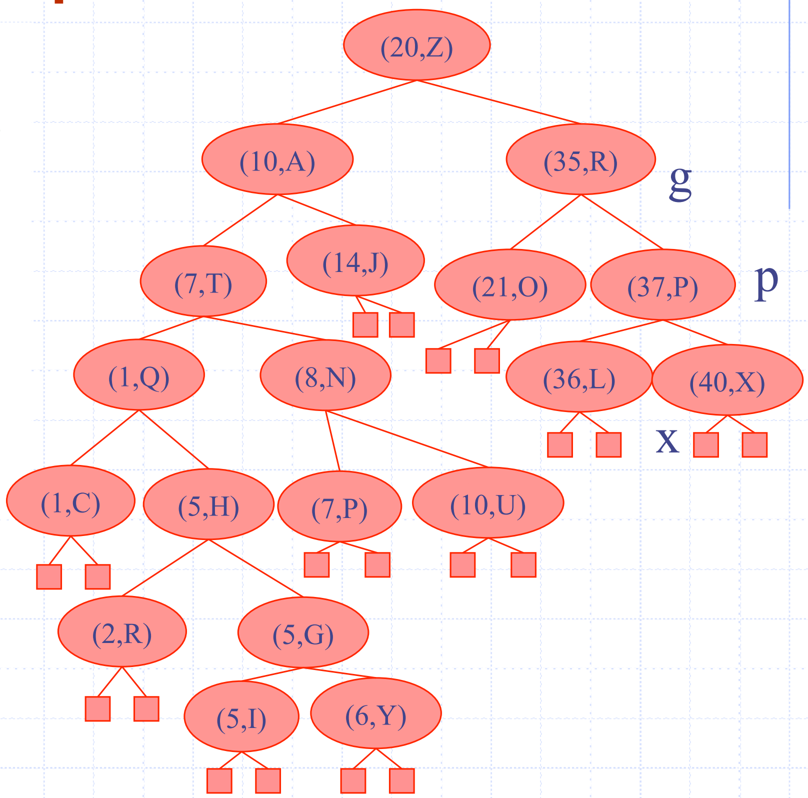
- let *x* = (8,N)
  - *x* is the left child of the root
  - right-rotate around *root.*



Slide by Matt Dickerson

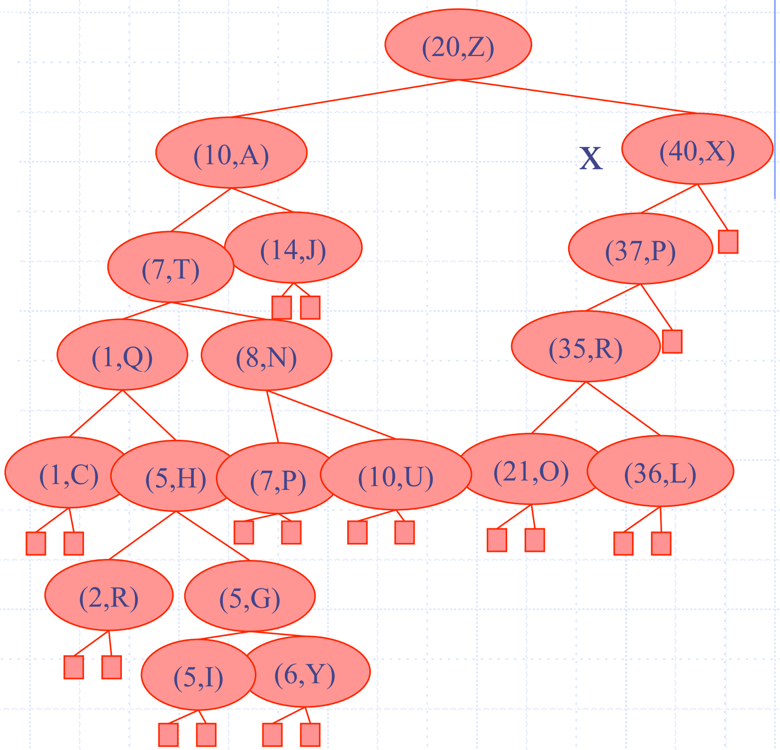
Splaying Example

- let x = (40,X)
  - x is the right child of its parent, which is the right child of its parent
  - zig-zig operation.



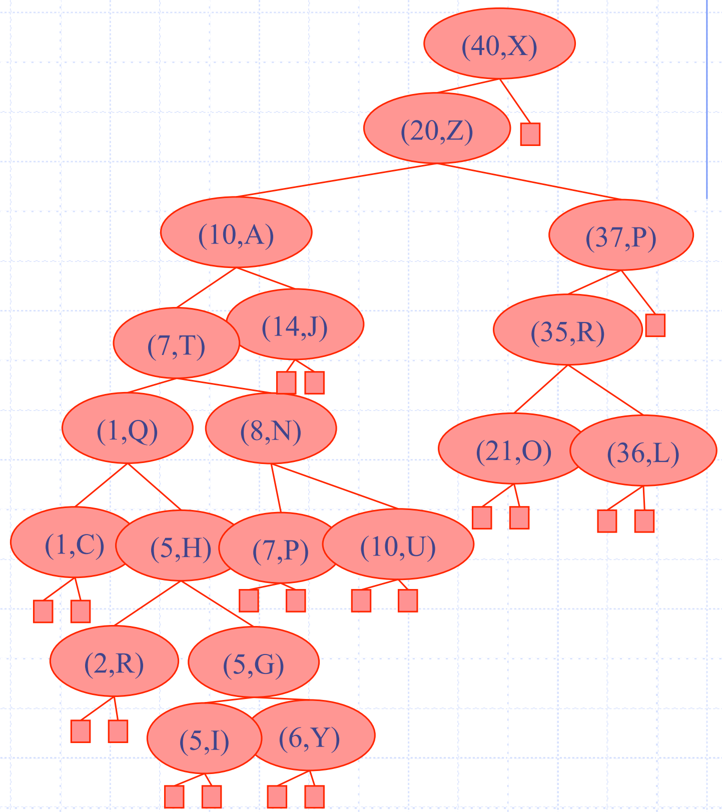
Splaying Example

- let *x* = (40,X)
  - *x* is the right child of its parent, which is the right child of its parent
  - zig-zig
  - *x* is the right child of the root
  - left rotate around the root



Splaying Example

- let x = (40,X)
  - x is the right child of its parent, which is the right child of its parent
  - zig-zig
  - x is the right child of the root
  - left rotate around the root



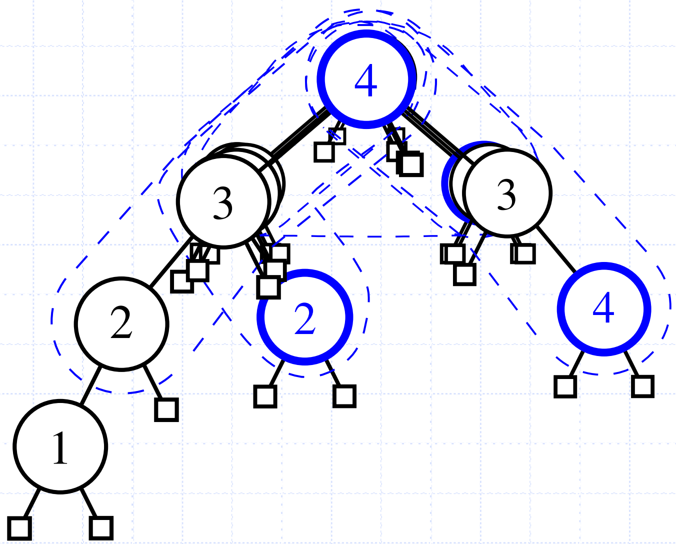
Splay Tree Definition

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

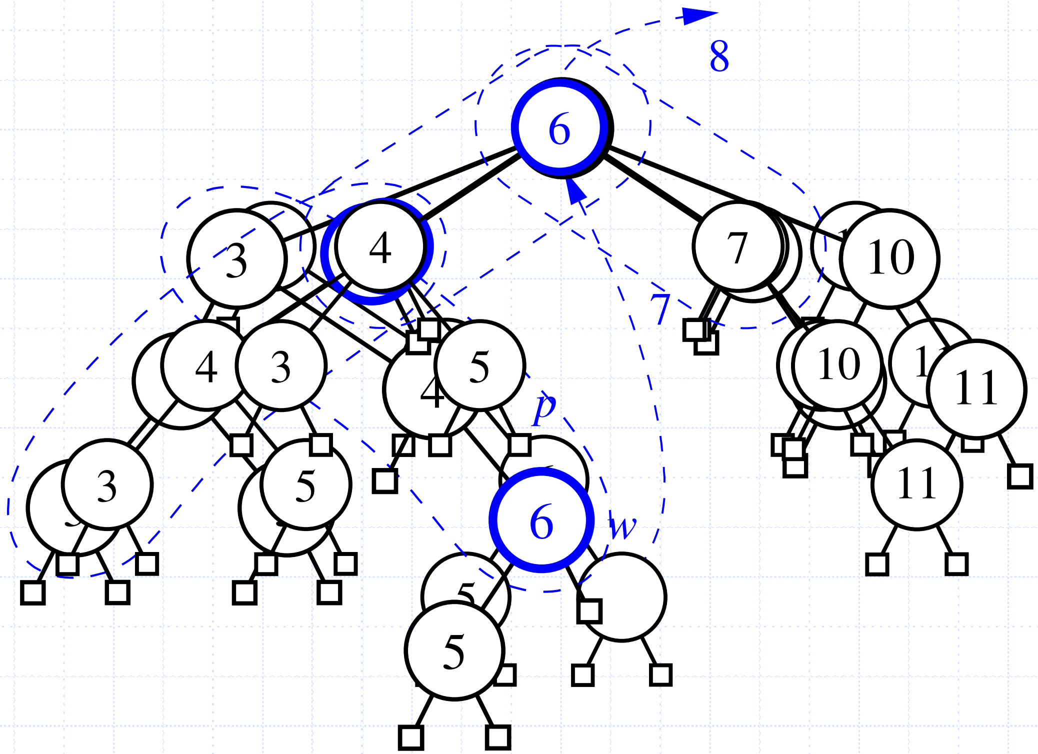
Splayed Nodes after Each Operation

method	**splay node**
Search for k	if key found, use that node if key not found, use parent of ending external node
Insert (k,v)	use the new node containing the entry inserted
Remove item with key k	use the parent of the internal node that was actually removed from the tree (the parent of the node that the removed item was swapped with)

Insertion



Deletion



Performance of Splay Trees

- Amortized cost of any splay operation is O(log n)
- Splay trees can actually adapt to perform searches on frequently-requested items much faster than O(log n) in some cases.