COMP9024: Data Structures and Algorithms

Week Six: Search Trees

Hui Wu

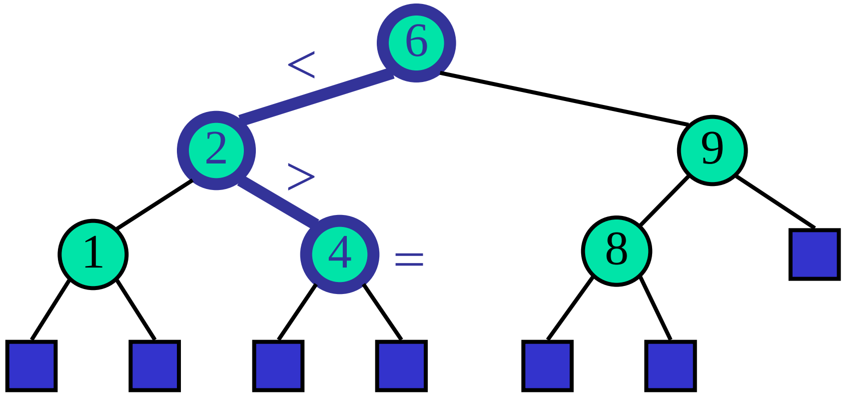
Session 1, 2015

http://www.cse.unsw.edu.au/~cs9024

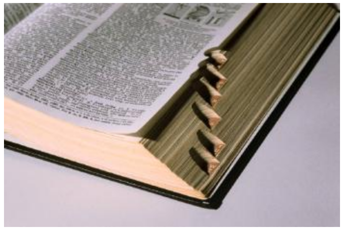
Outline

- Binary Search Trees
- AVL Trees
- Splay Trees
- (2,4) Trees
- Red-Black Trees

Binary Search Trees



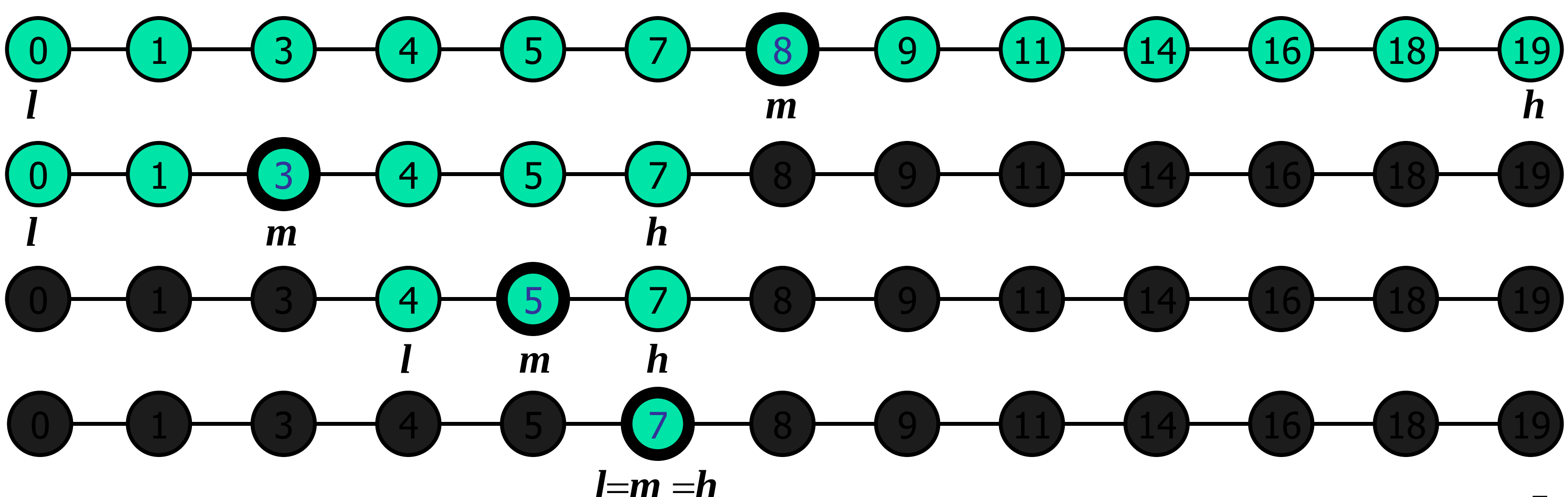
Ordered Dictionaries



- Keys are assumed to come from a total order.
- New operations:
  - first(): first entry in the dictionary ordering
  - last(): last entry in the dictionary ordering
  - successors(k): iterator of entries with keys greater than or equal to k; increasing order
  - predecessors(k): iterator of entries with keys less than or equal to k; decreasing order

Binary Search

- Binary search can perform operation find(k) on a dictionary implemented by means of an array-based sequence, sorted by key
  - similar to the high-low game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)

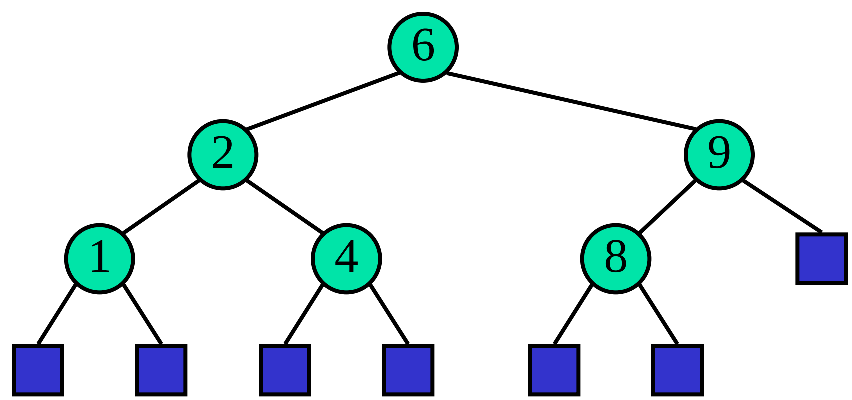


Search Tables

- A search table is a dictionary implemented by means of a sorted sequence
  - We store the items of the dictionary in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - find takes $O(\log n)$ time, using binary search
  - insert takes $O(n)$ time since in the worst case we have to shift $n/2$ items to make room for the new item
  - remove take $O(n)$ time since in the worst case we have to shift $n/2$ items to compact the items after the removal
- The lookup table is effective only for dictionaries of small size or for dictionaries on which searches are the most common operations, while insertions and removals are rarely performed (e.g., credit card authorizations)

Binary Search Trees

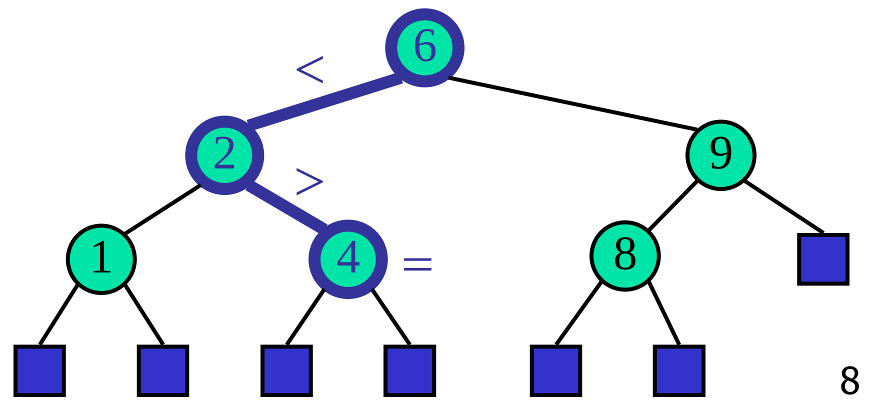
- A binary search tree is a binary tree storing keys (or key-value entries) at its internal nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items
- An inorder traversal of a binary search trees visits the keys in increasing order



Search

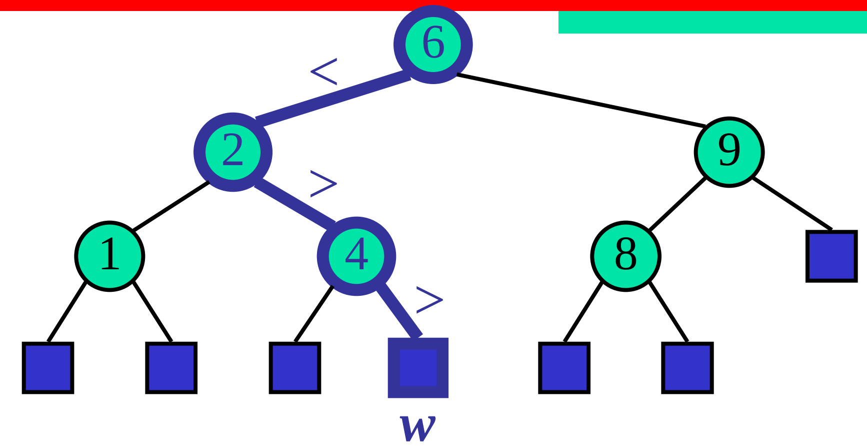
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node
- If we reach a leaf, the key is not found and we return null
- Example: find(4):
  - Call TreeSearch(4,root)

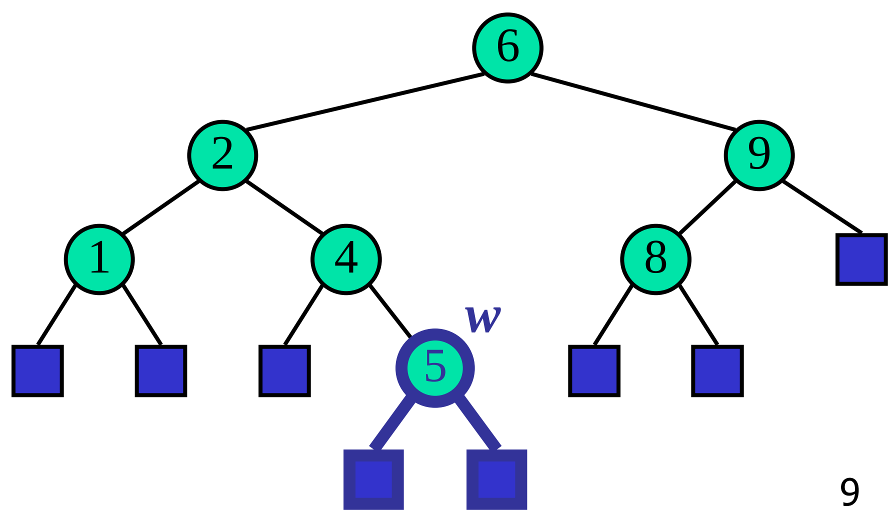
```
Algorithm TreeSearch(k, v)
  { if ( T.isExternal (v) )
        return v ;
    if ( k < key(v) )
        return TreeSearch(k, T.left(v));
    else if  ( k = key(v) )
        return v ;
    else  // k > key(v)
        return TreeSearch(k, T.right(v)) ; }
```



Insertion

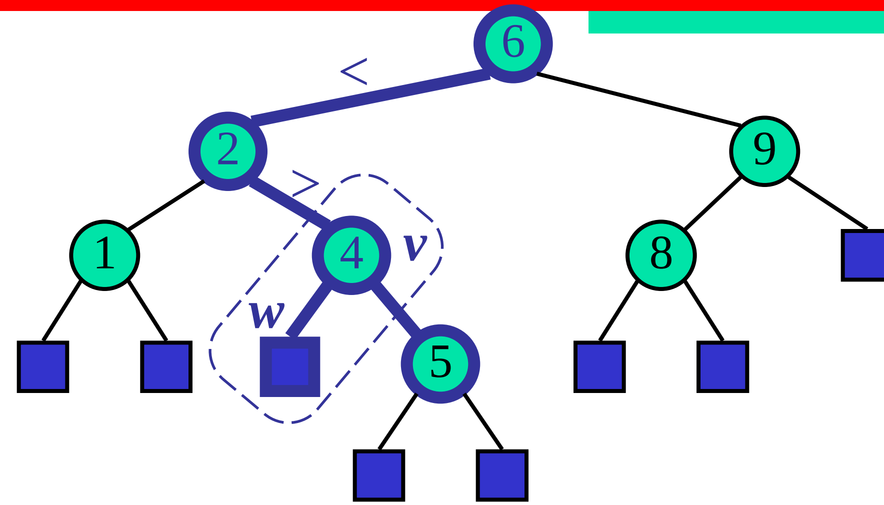
- To perform operation insert(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let let w be the leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5

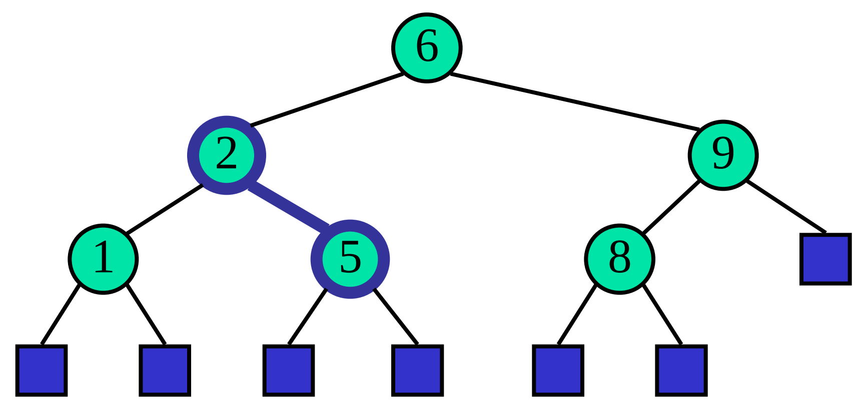




Deletion (1/2)

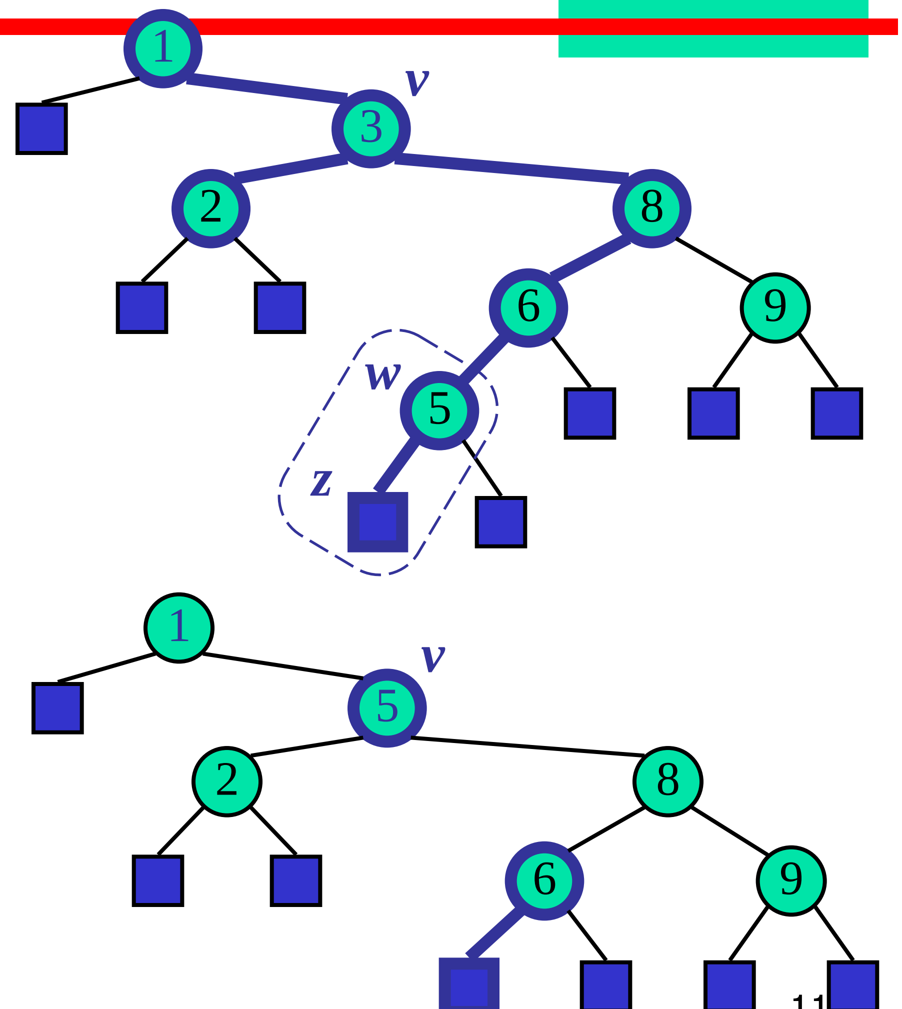
- To perform operation remove(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: remove 4





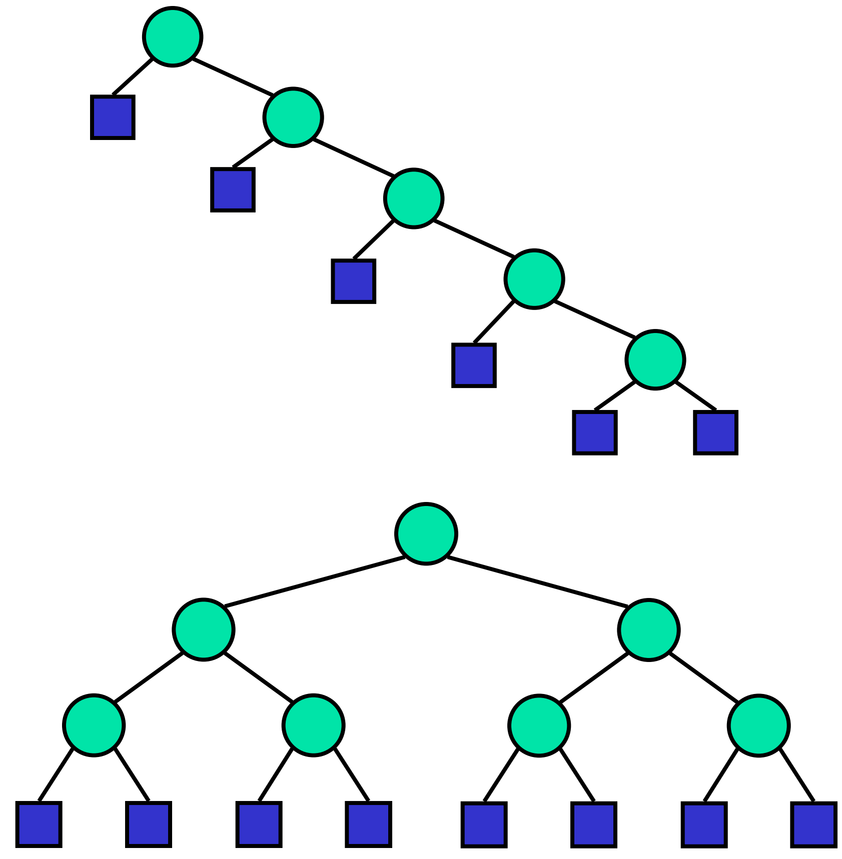
Deletion (2/2)

- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)
- Example: remove 3

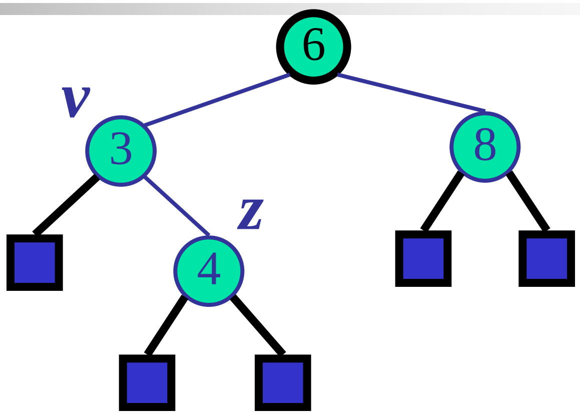


Performance

- Consider a dictionary with n items implemented by means of a binary search tree of height h
  - the space used is $O(n)$
  - methods find, insert and remove take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case

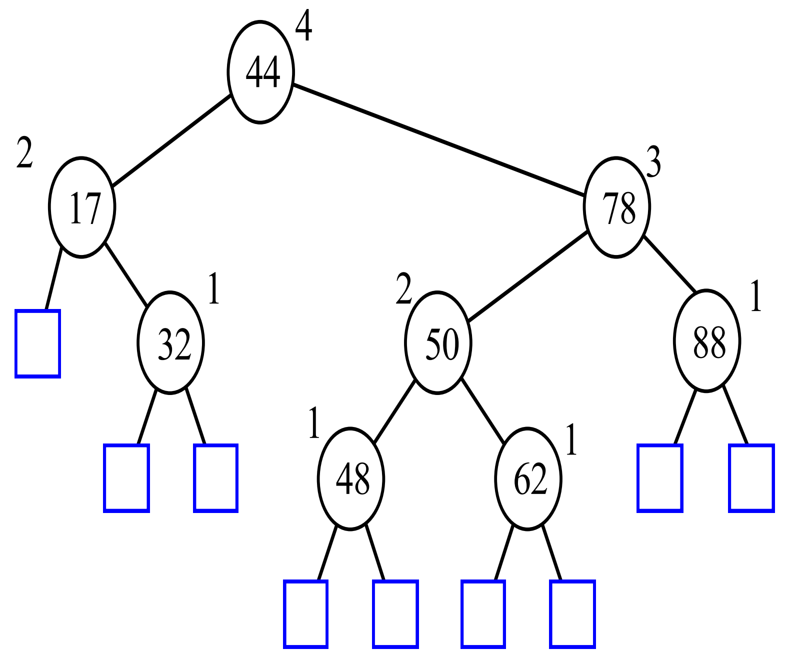


AVL Trees



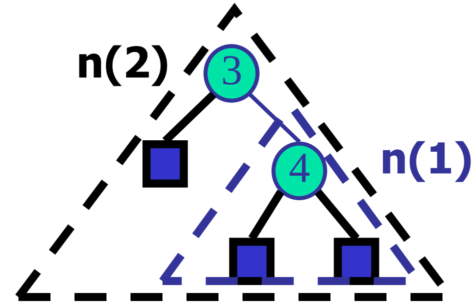
AVL Tree Definition

- **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*.



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

Height of an AVL Tree



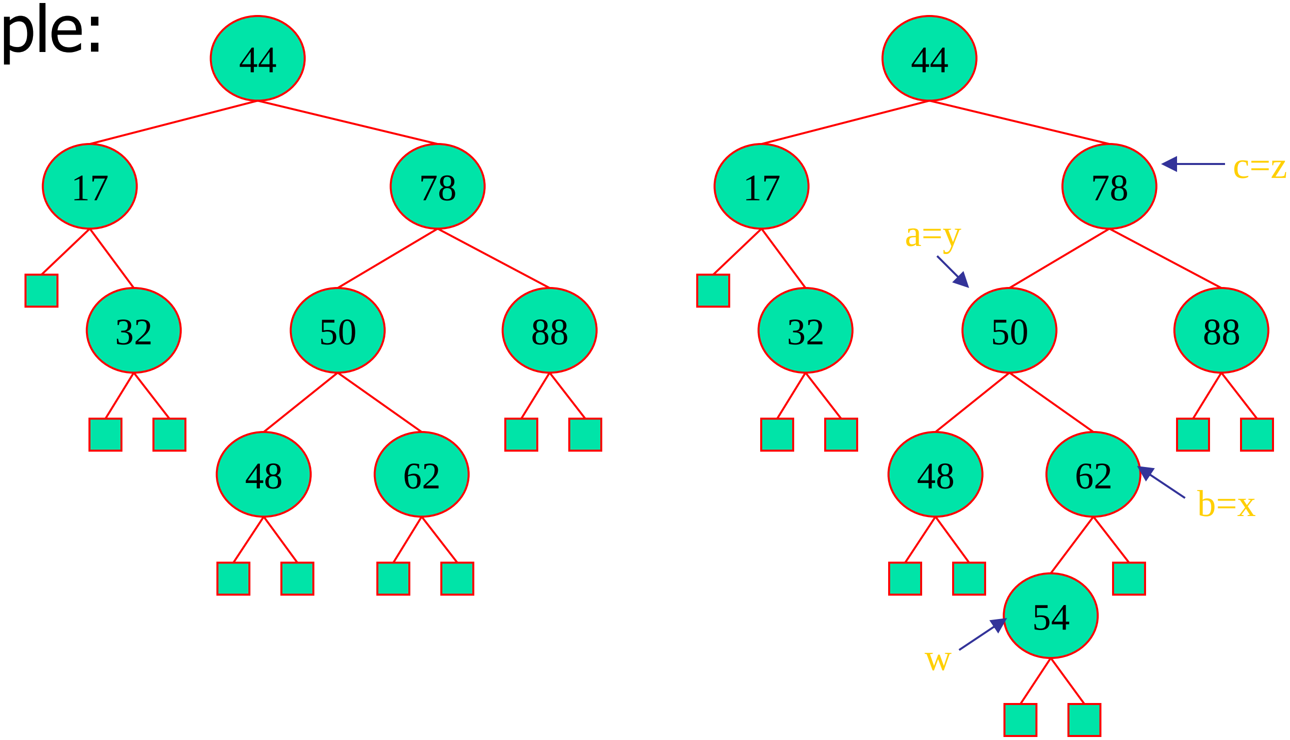
- **Fact**: The ***height*** of an AVL tree storing n keys is O(log n).
- **Proof**: Let us bound **n(h):** the minimum number of internal nodes of an AVL tree of height h.
- We easily see that n(1) = 1 and n(2) = 2
- For n > 2, an AVL tree of height h contains the root node, one AVL subtree of height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

  $n(h) > 2^i n(h-2i)$
- Solving the base case we get: $n(h) > 2^{h/2-1}$
- Taking logarithms: h < 2log n(h) +2
- Thus the height of an AVL tree is O(log n)

Insertion in an AVL Tree

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example:

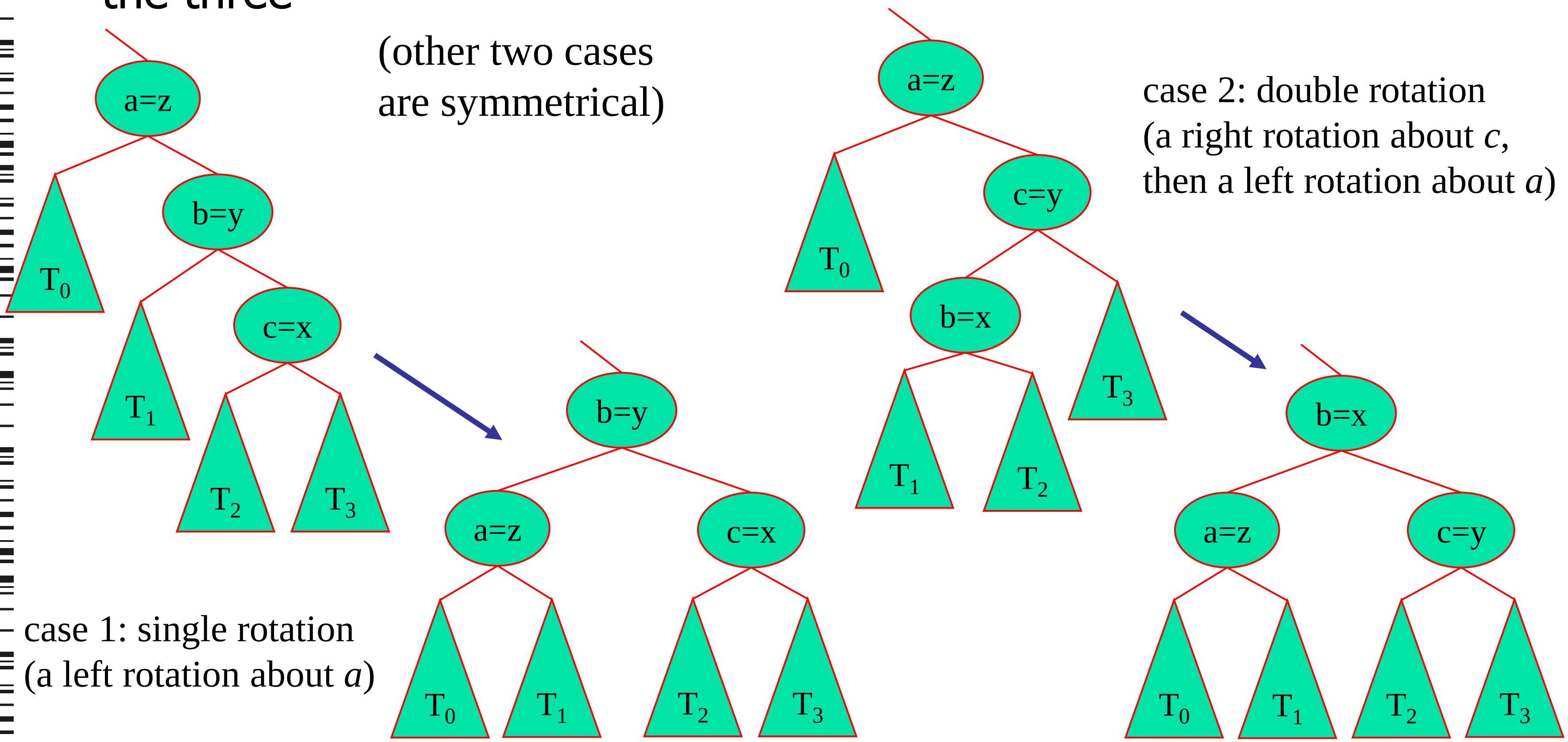


before insertion after insertion

Trinode Restructuring

- let (*a*,*b*,*c*) be an inorder listing of *x*, *y*, *z*
- perform the rotations needed to make *b* the topmost node of the three

(other two cases are symmetrical)

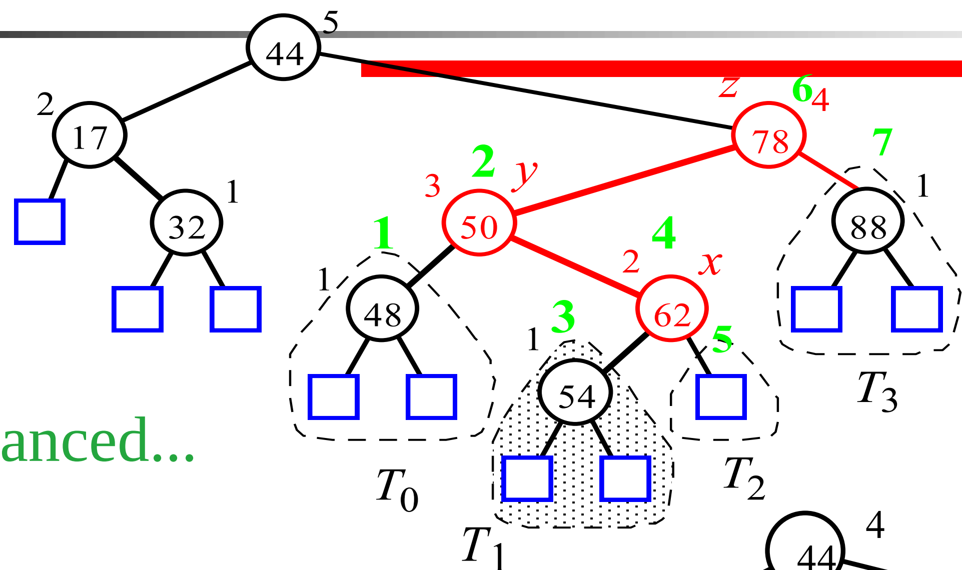


case 1: single rotation (a left rotation about *a*)

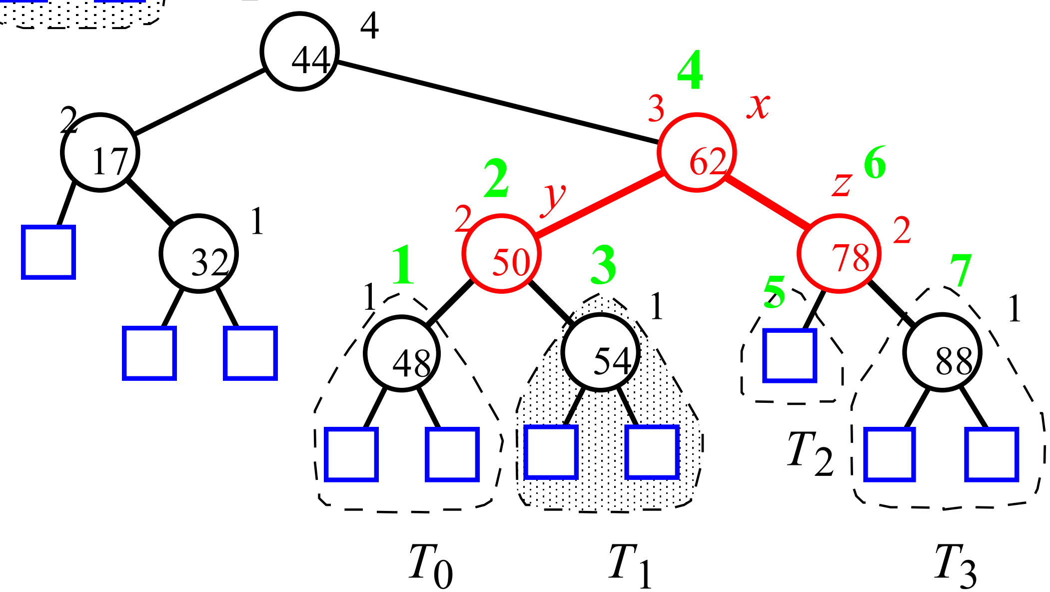
case 2: double rotation (a right rotation about *c*, then a left rotation about *a*)

Insertion Example, continued

unbalanced...

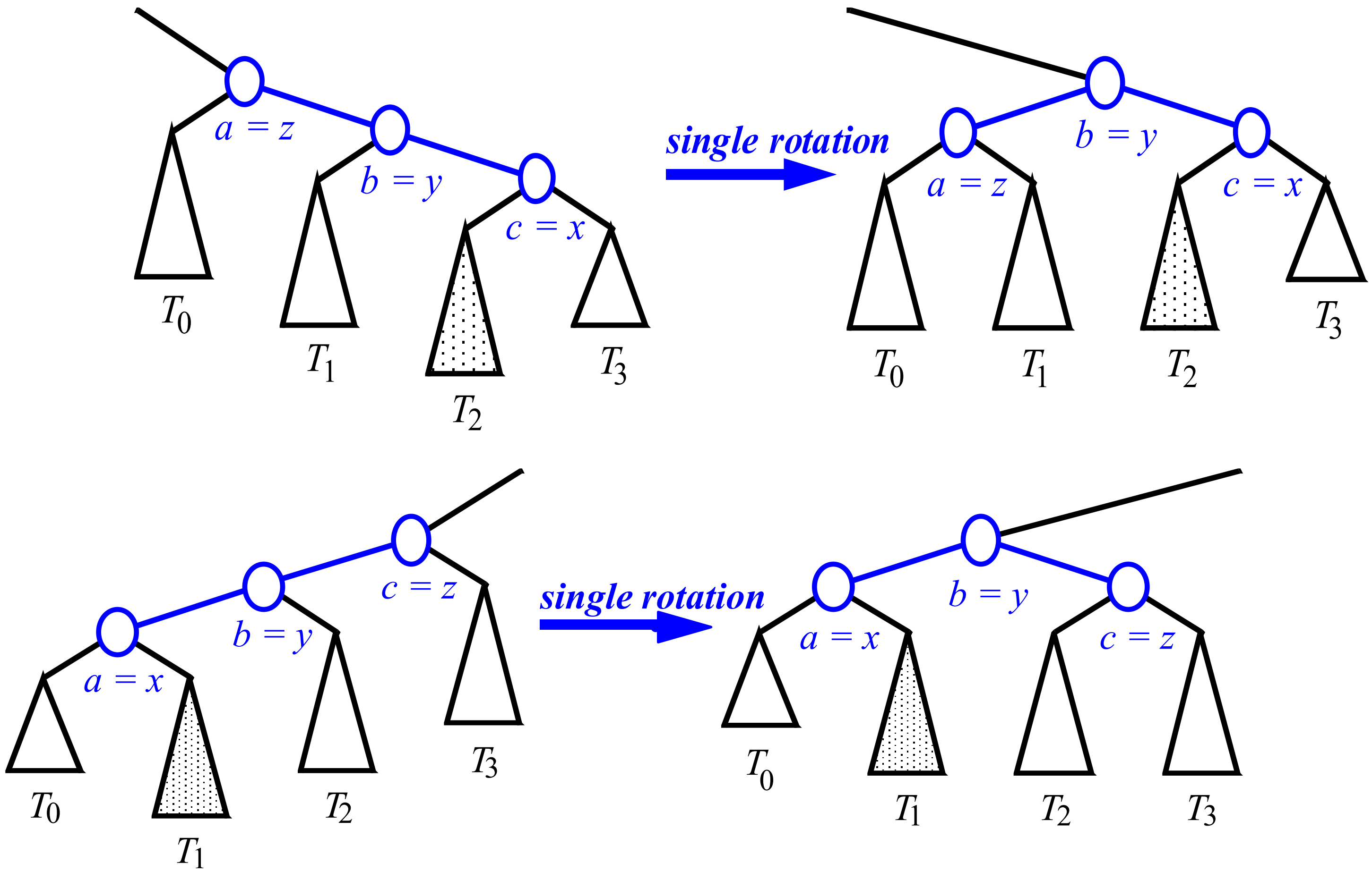


...balanced



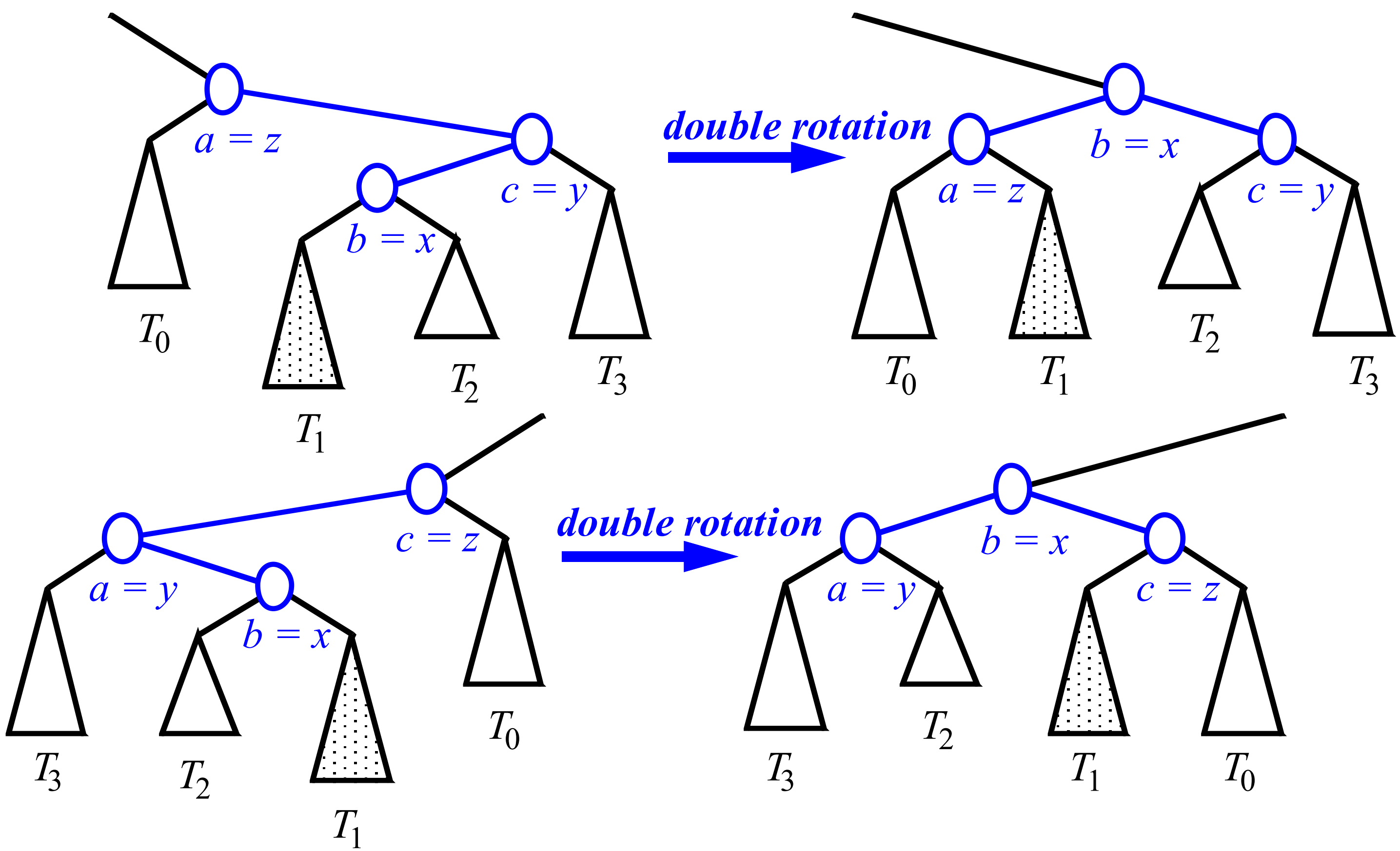
Restructuring(as Single Rotations)

- Single Rotations:



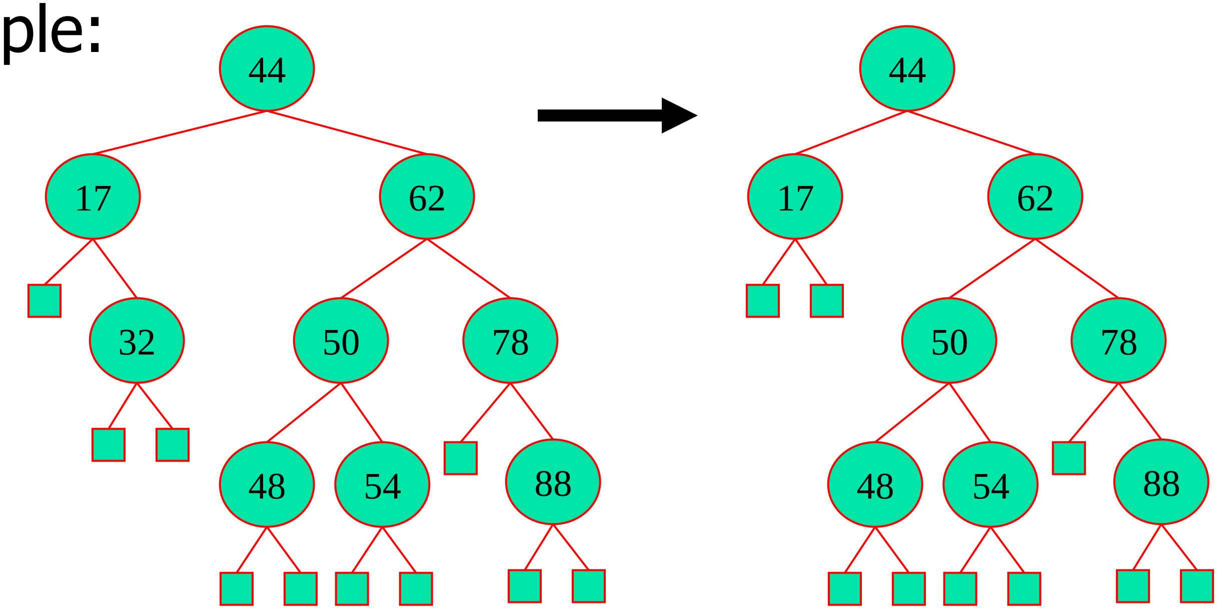
Restructuring(as Double Rotations)

- double rotations:



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:

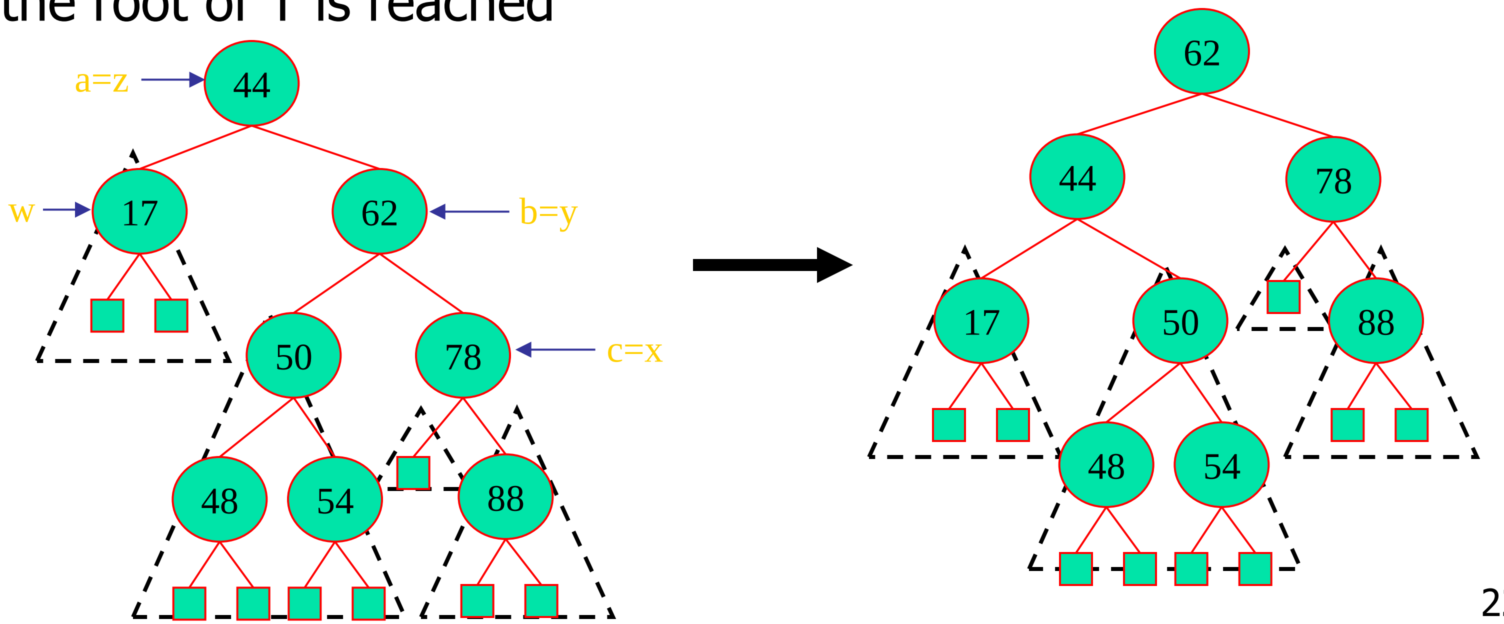


before deletion of 32

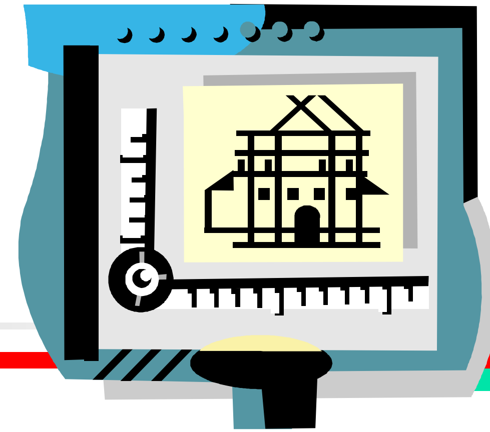
after deletion

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.
- We perform restructure(x) 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

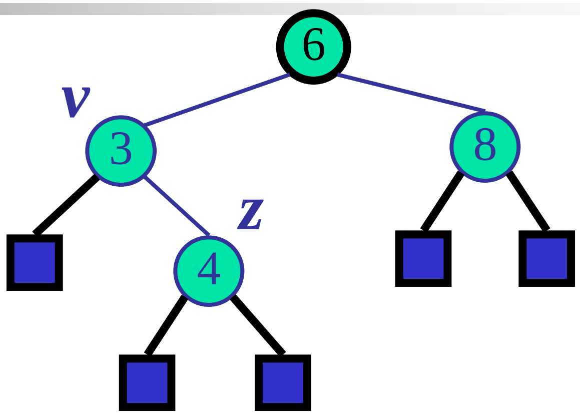


Running Times for AVL Trees



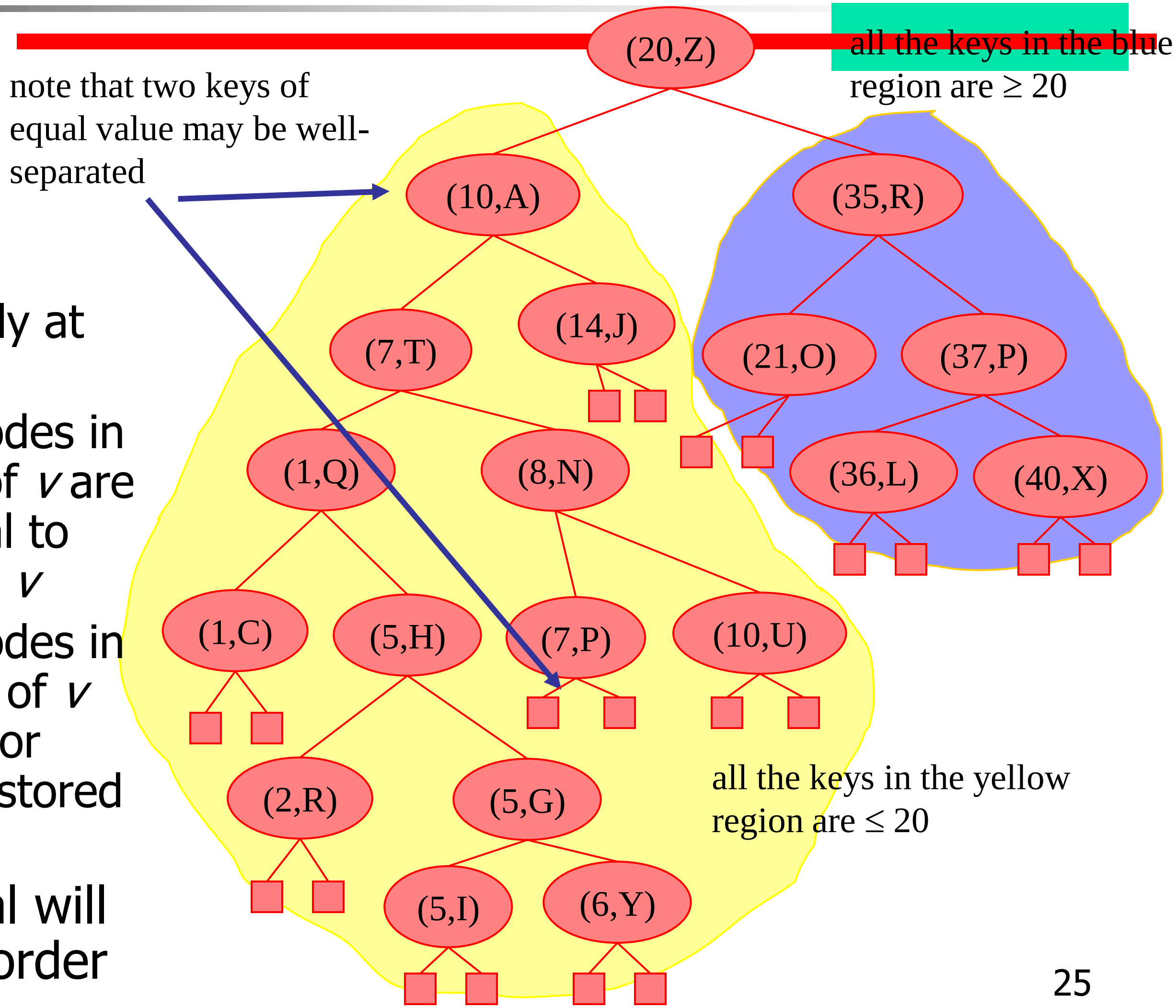
- a single restructure is O(1)
  - using a linked-structure binary tree
- find is O(log n)
  - height of tree is O(log n), no restructures needed
- insert is O(log n)
  - initial find is O(log n)
  - Restructuring up the tree, maintaining heights is O(log n)
- remove is O(log n)
  - initial find is O(log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

Splay Trees



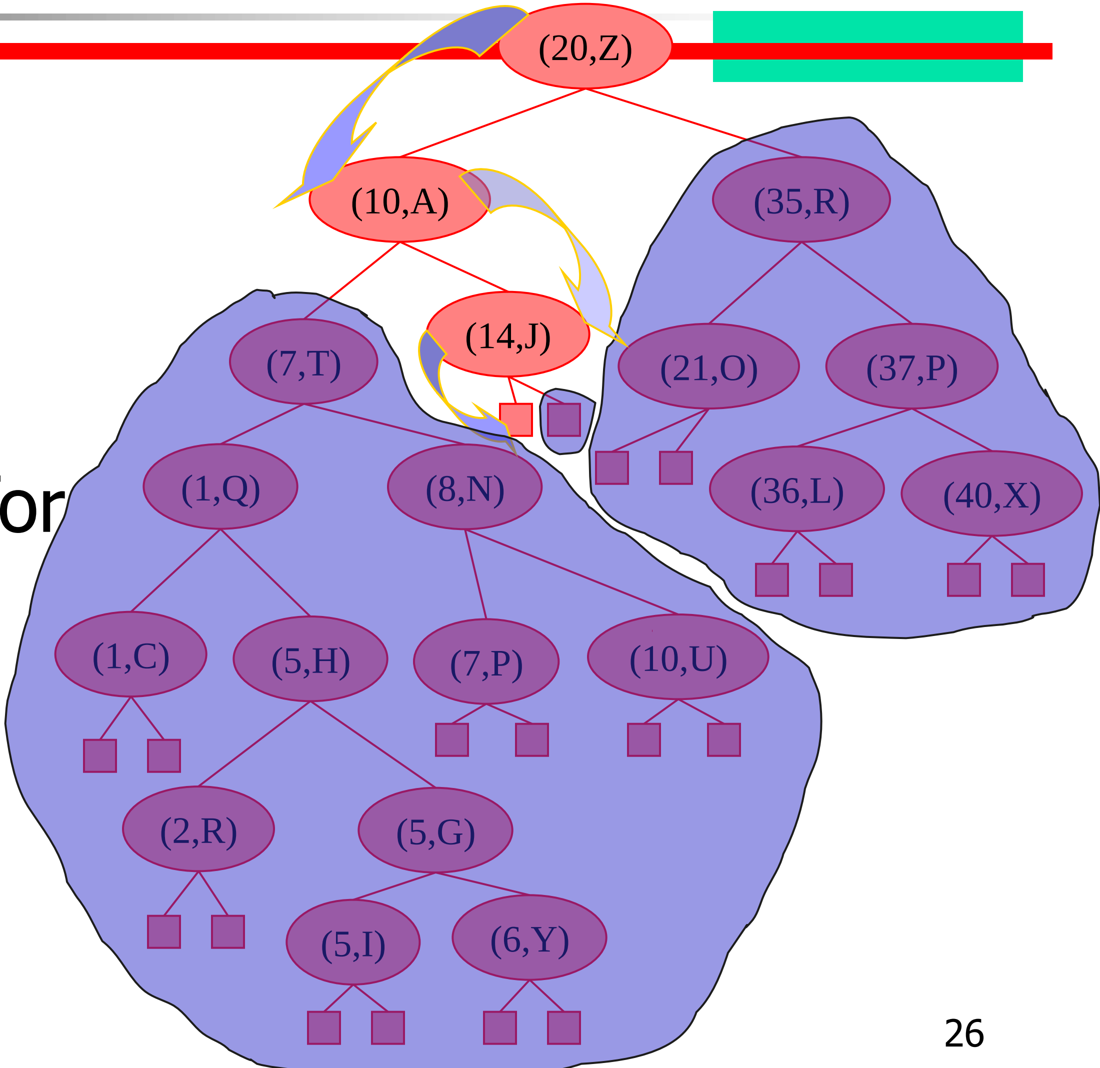
Splay Trees are Binary Search Trees

- 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



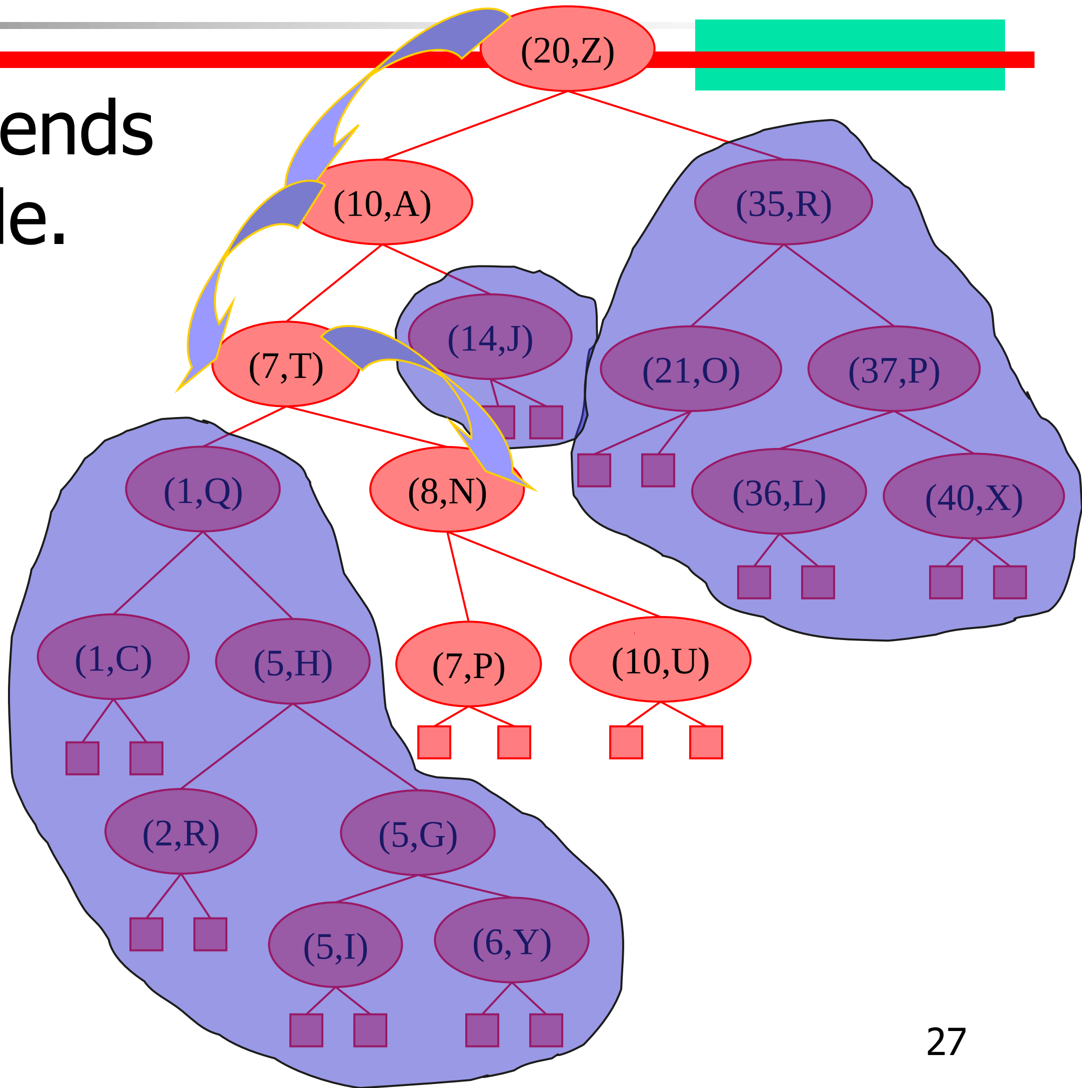
Searching in a Splay Tree: Starts the Same as in a BST

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



Example Searching in a BST, continued

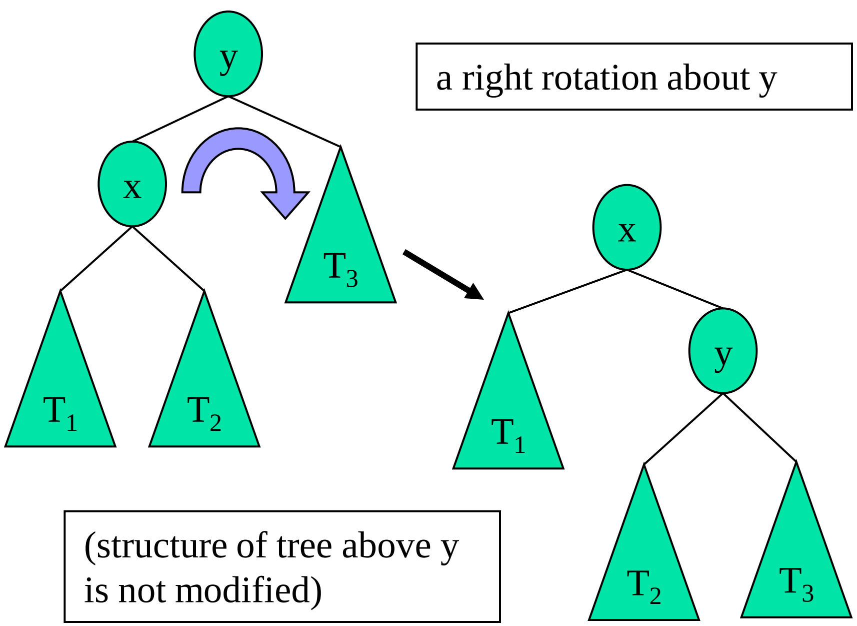
- search for key 8, ends at an internal node.



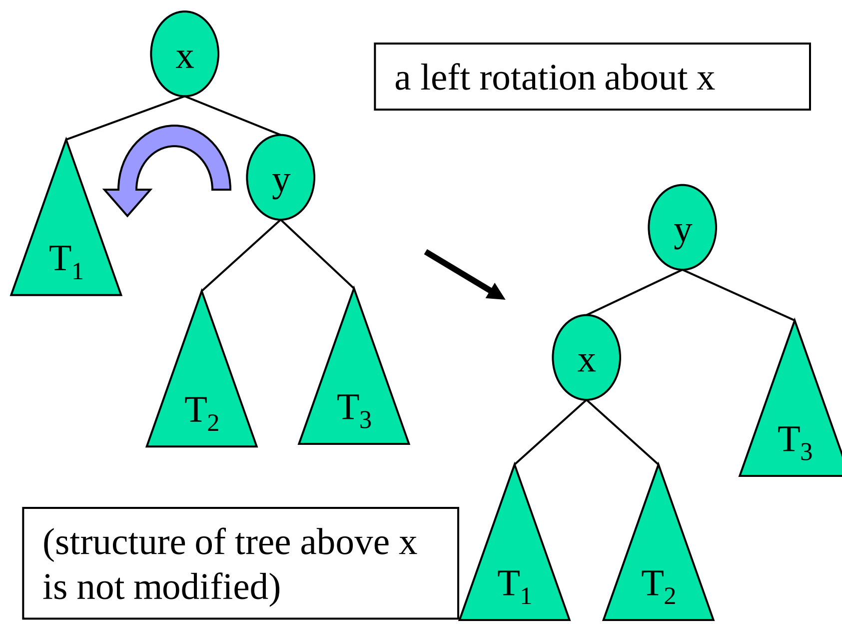
Splay Trees do Rotations after Every Operation (Even Search)

- 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*

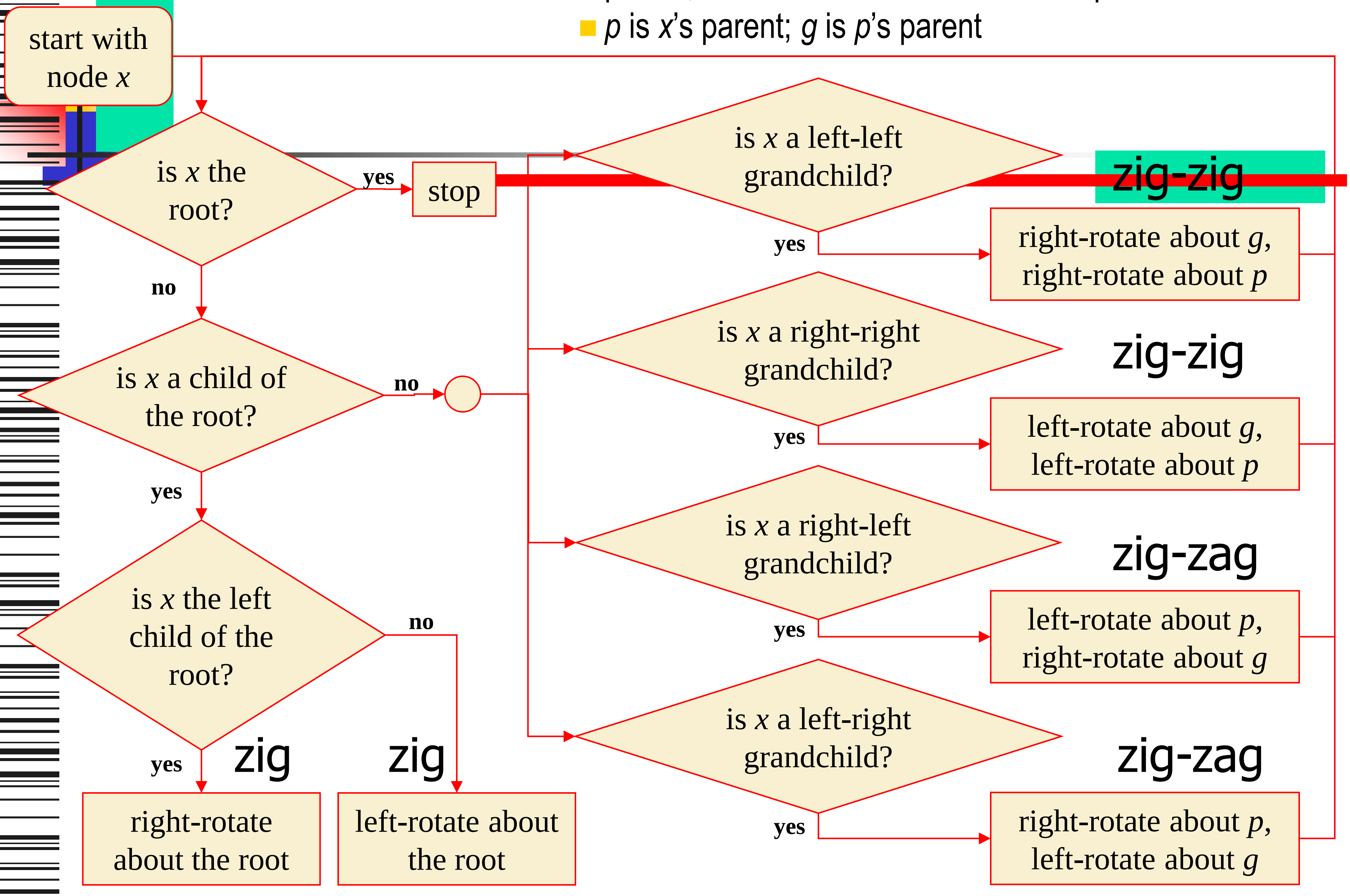


- left rotation
  - makes the right child *y* of a node *x* into *x*'s parent; *x* becomes the left child of *y*

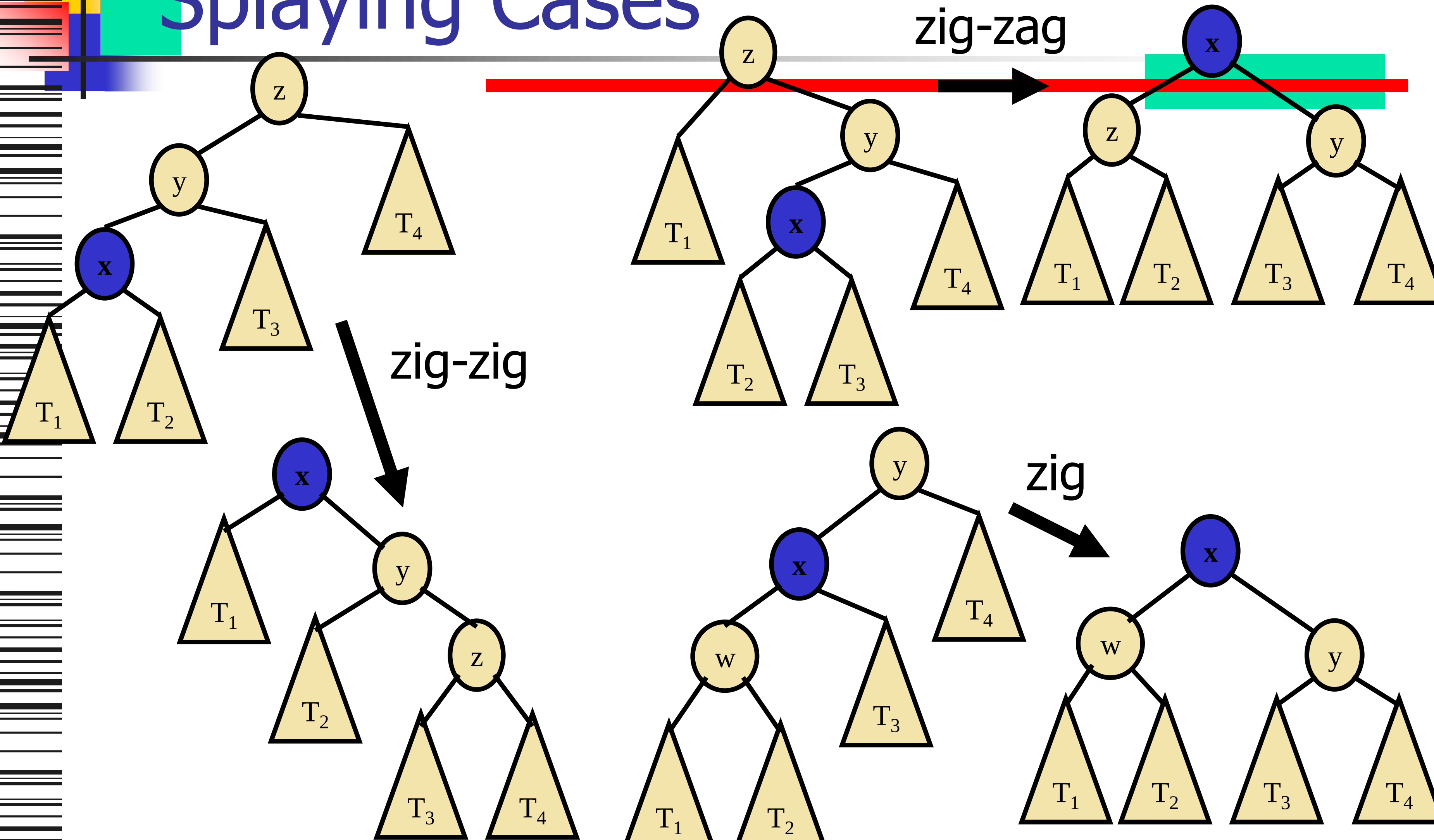


Splaying:

- "*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

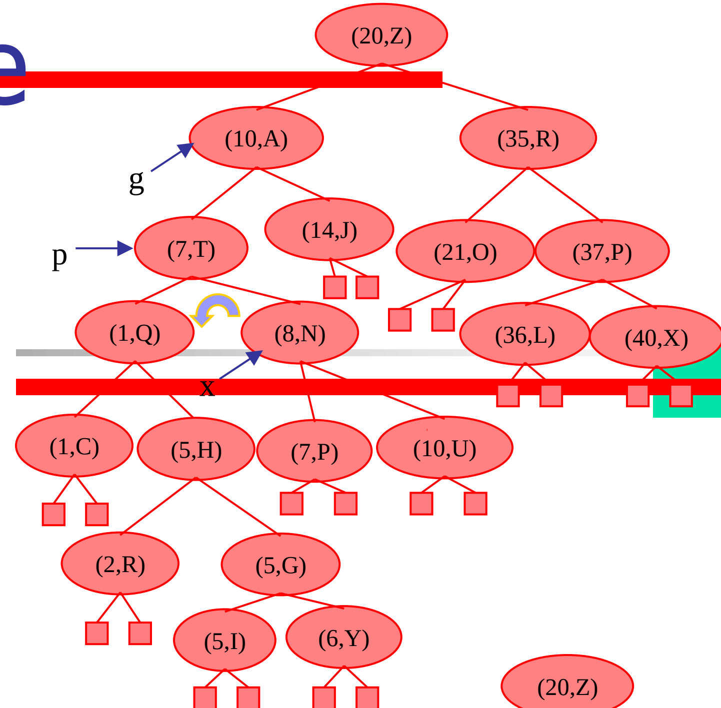


Visualizing the Splaying Cases

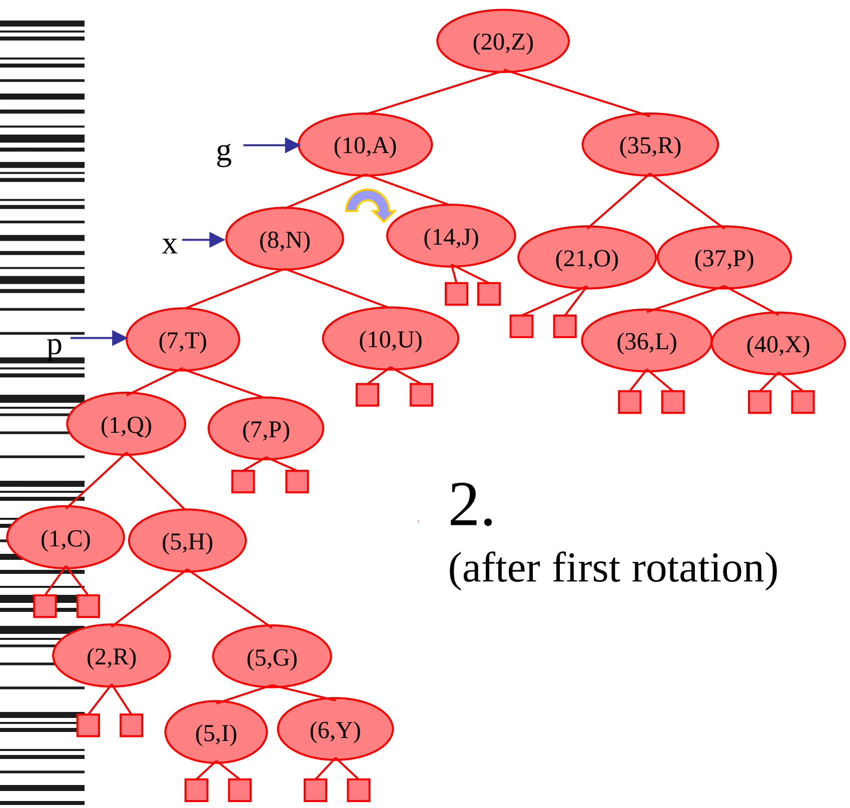


Splaying Example

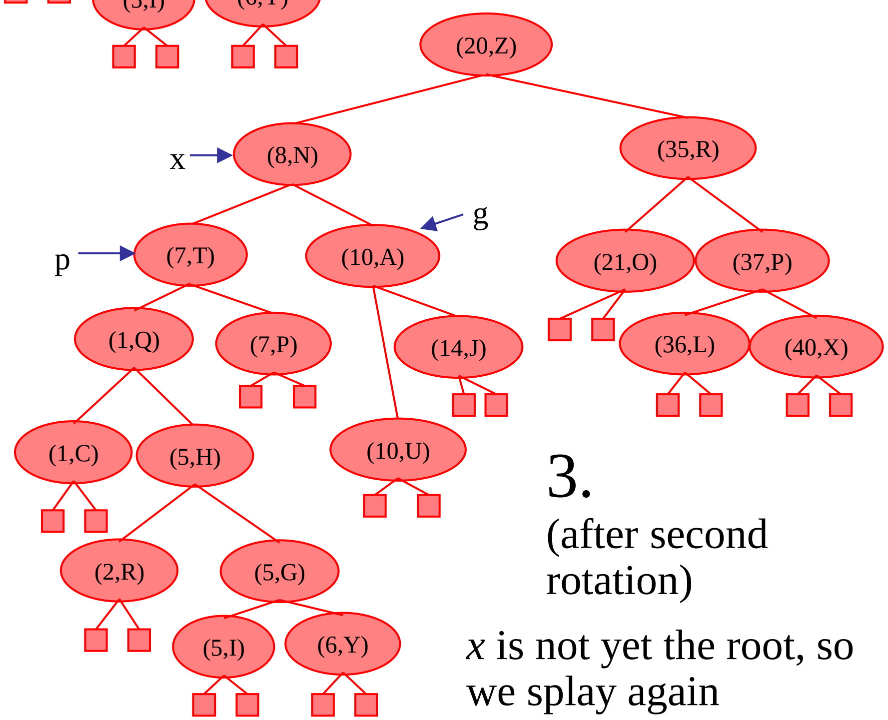
- let x = (8,N)
  - x is the right child of its parent, which is the left child of the grandparent
  - left-rotate around p, then right-rotate around g



1.
(before rotating)



2.
(after first rotation)

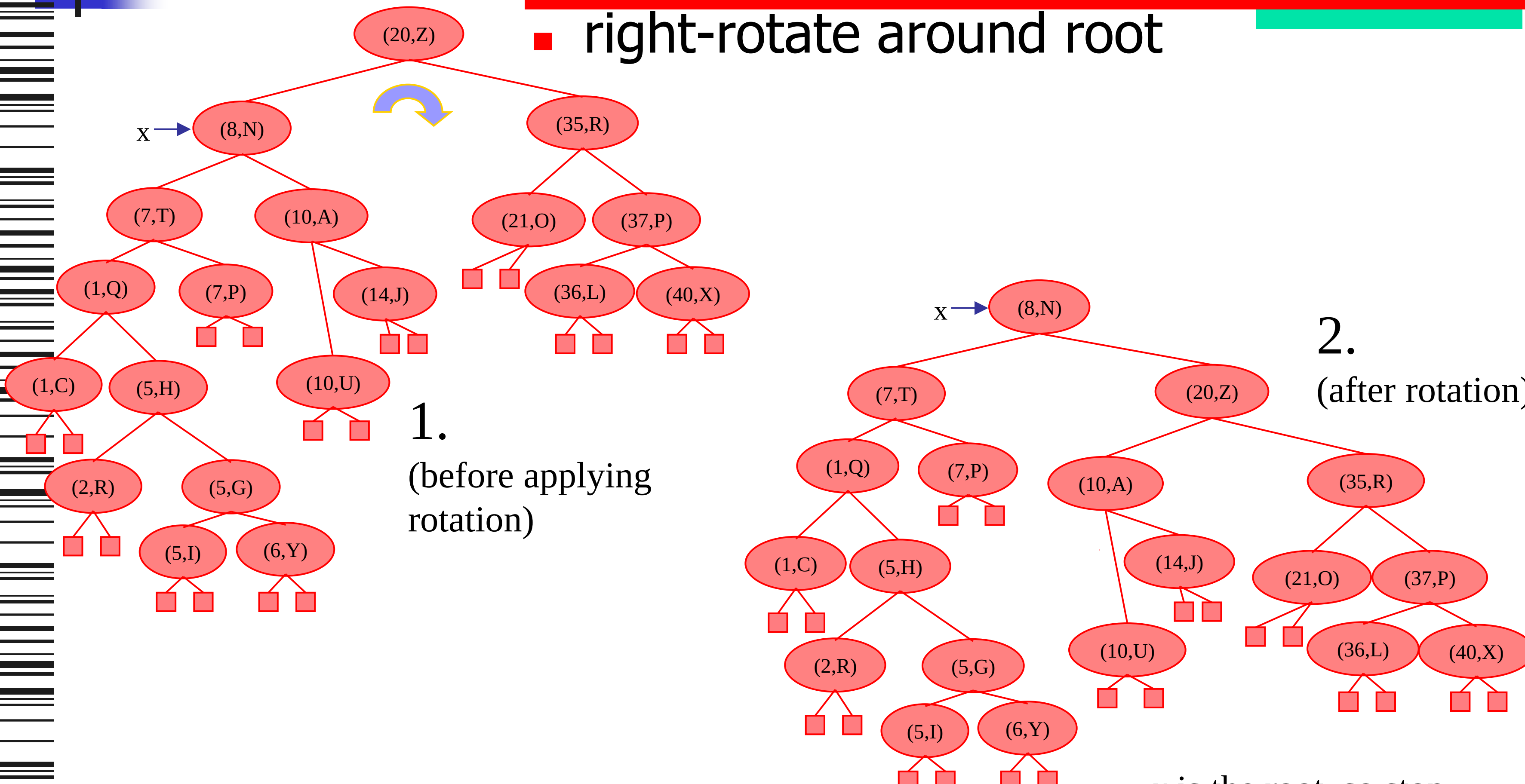


3.
(after second rotation)

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

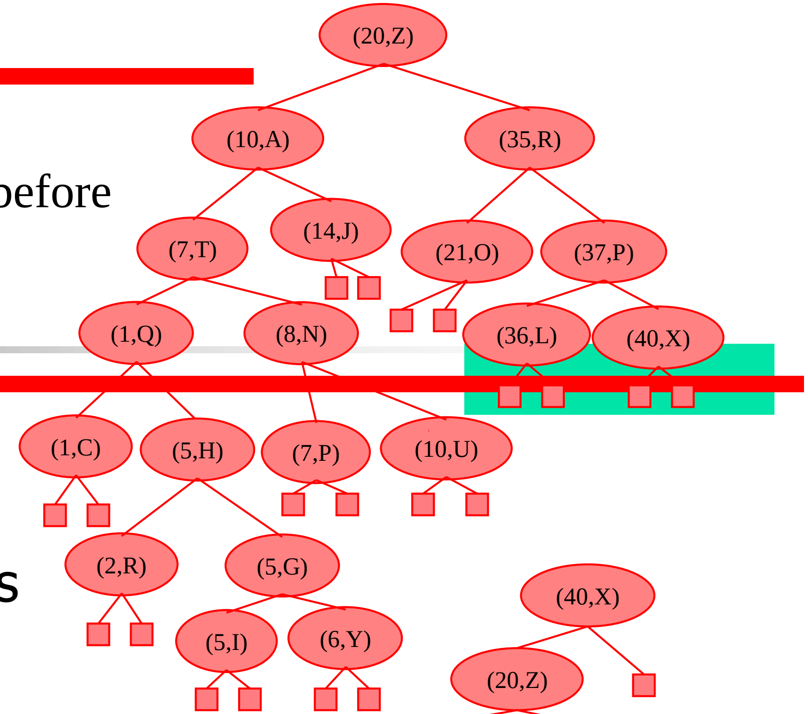


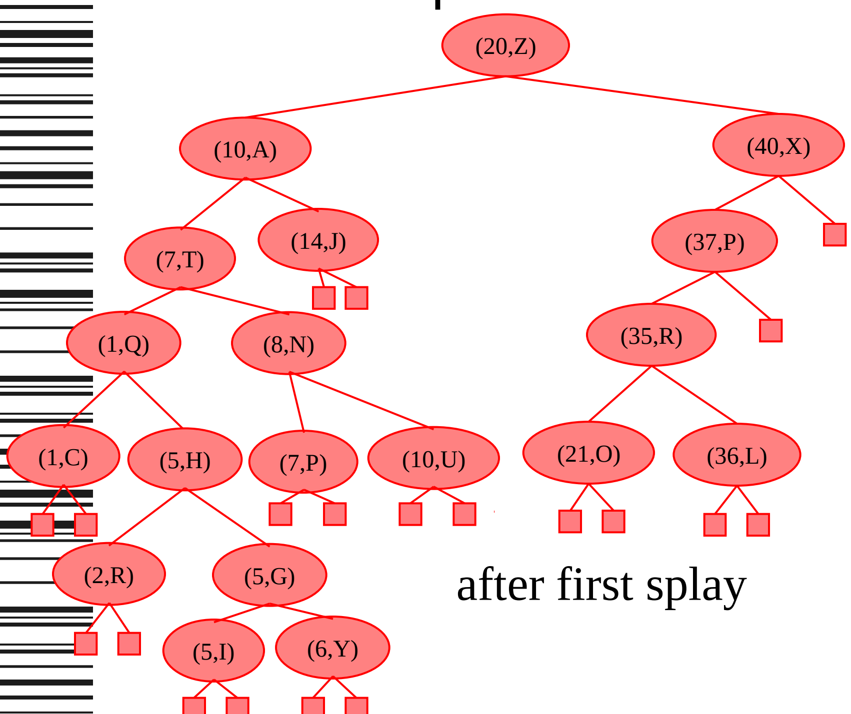
x is the root, so stop

Example Result of Splaying

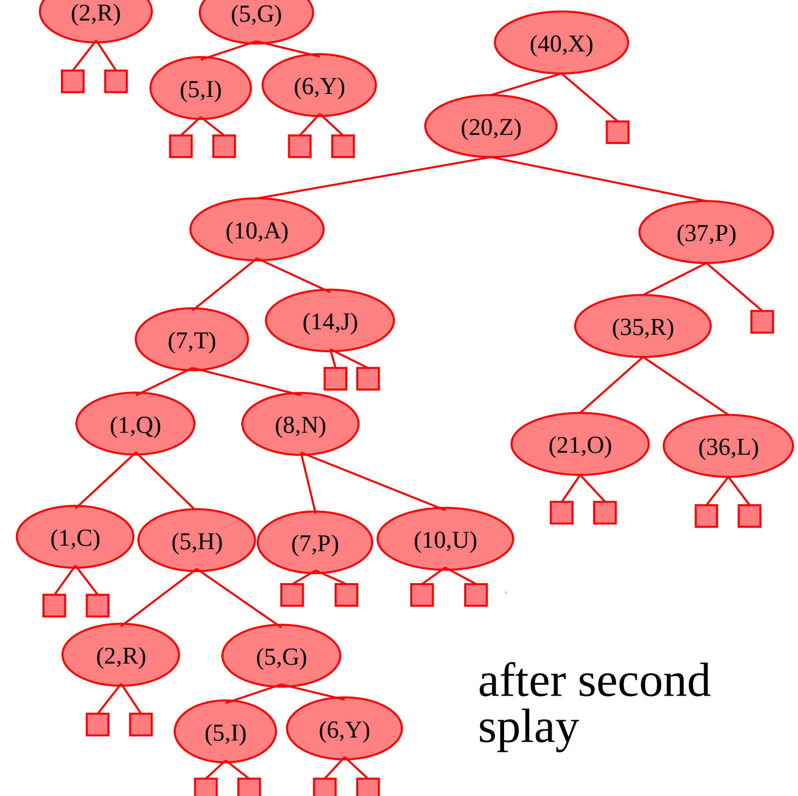
- tree might not be more balanced
- e.g. splay (40,X)
  - before, the depth of the shallowest leaf is 3 and the deepest is 7
  - after, the depth of shallowest leaf is 1 and deepest is 8

before





after first splay

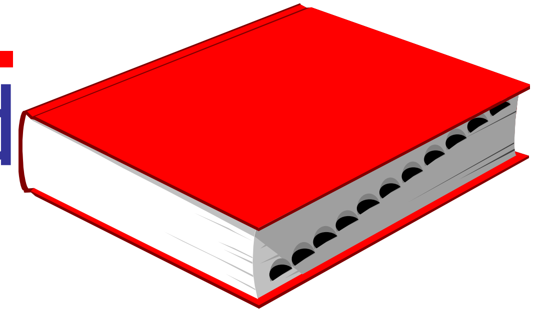


after second splay

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 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)

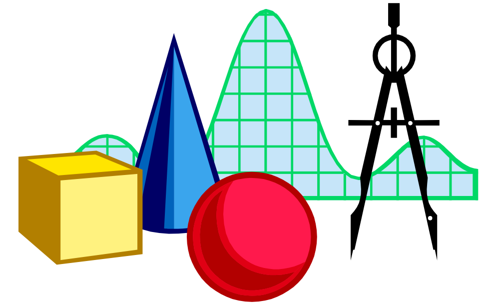
Splay Trees & Ordered Dictionaries



- which nodes are splayed after each operation?

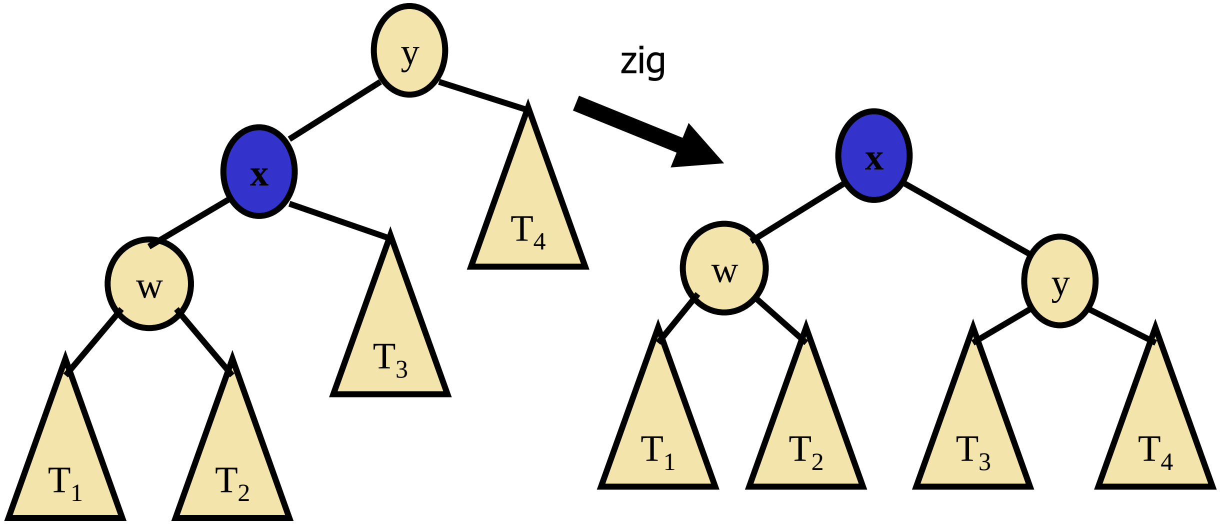
method	splay node
find(k)	if key found, use that node if key not found, use parent of ending external node
insert(k,v)	use the new node containing the entry inserted
remove(k)	use the parent of the internal node that was actually removed from the tree (the parent of the node that the removed item was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

Amortized Analysis of Splay Trees



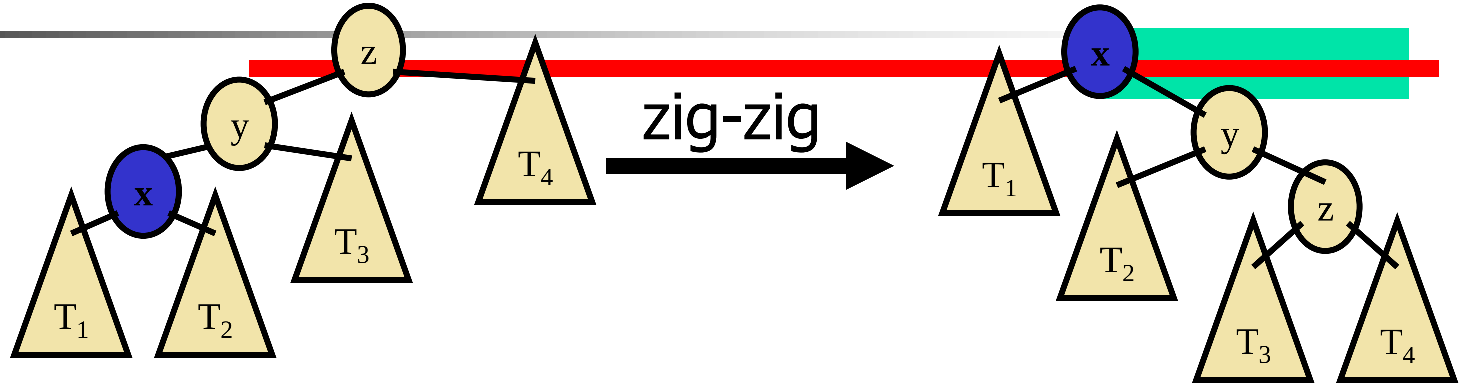
- Running time of each operation is proportional to time for splaying.
- Define rank(v) as the logarithm (base 2) of the number of nodes in subtree rooted at v.
- Costs: zig = \$1, zig-zig = \$2, zig-zag = \$2.
- Thus, cost for splaying a node at depth d = \$d.
- Imagine that we store rank(v) & cyber-dollars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

Cost per zig

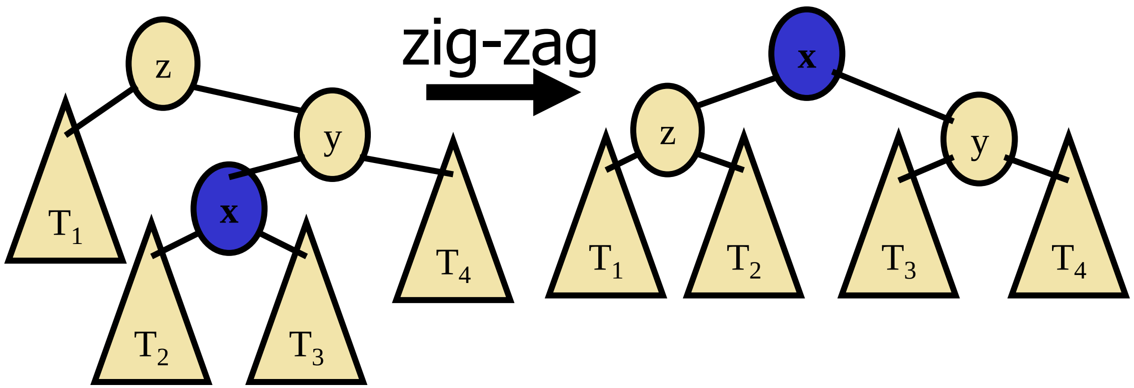


- Doing a zig at x costs at most rank'(x) - rank(x):
  - cost = rank'(x) + rank'(y) - rank(y) - rank(x) $\leq$ rank'(x) - rank(x).

Cost per zig-zig and zig-zag



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



Cost of Splaying

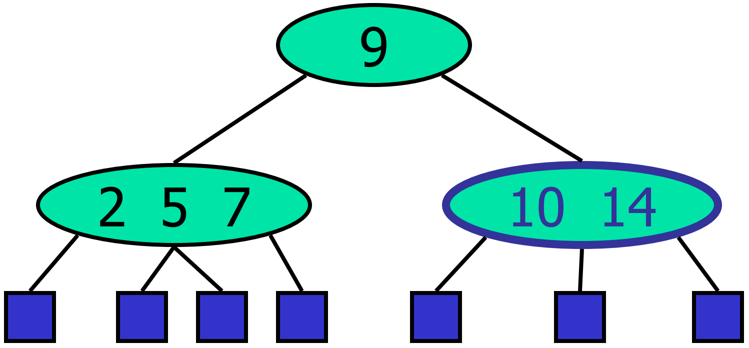
- Cost of splaying a node x at depth d of a tree rooted at r:
  - at most 3(rank(r) - rank(x)) - d + 2:
  - Proof: Splaying x takes d/2 splaying substeps:

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

Performance of Splay Trees

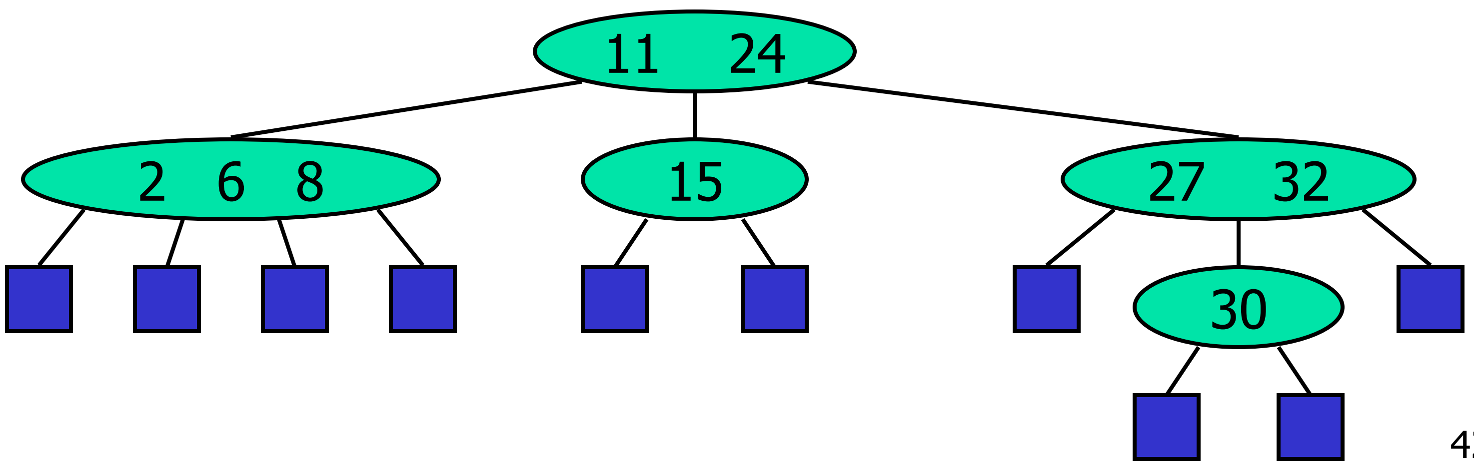
- Recall: rank of a node is logarithm of its size.
- Thus, amortized cost of any splay operation is **O(log n).**
- In fact, the analysis goes through for any reasonable definition of rank(x).
- This implies that splay trees can actually adapt to perform searches on frequently-requested items much faster than O(log n) in some cases. (See Proposition 10.6.)

(2,4) Trees



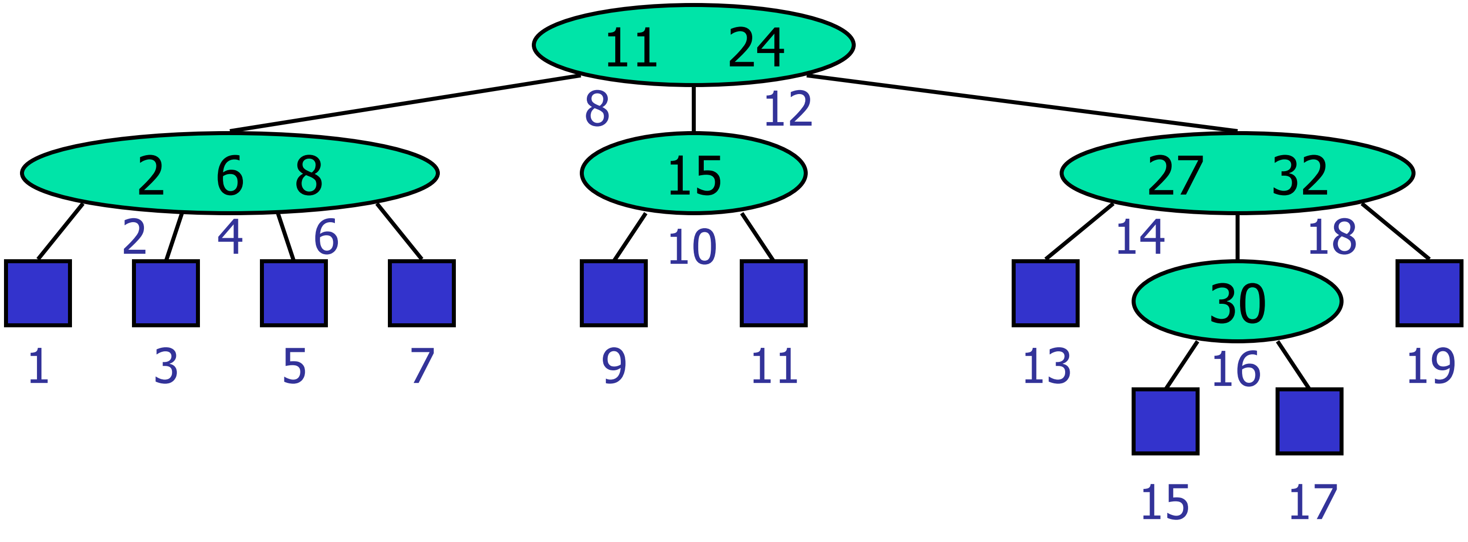
Multi-Way Search Tree

- 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



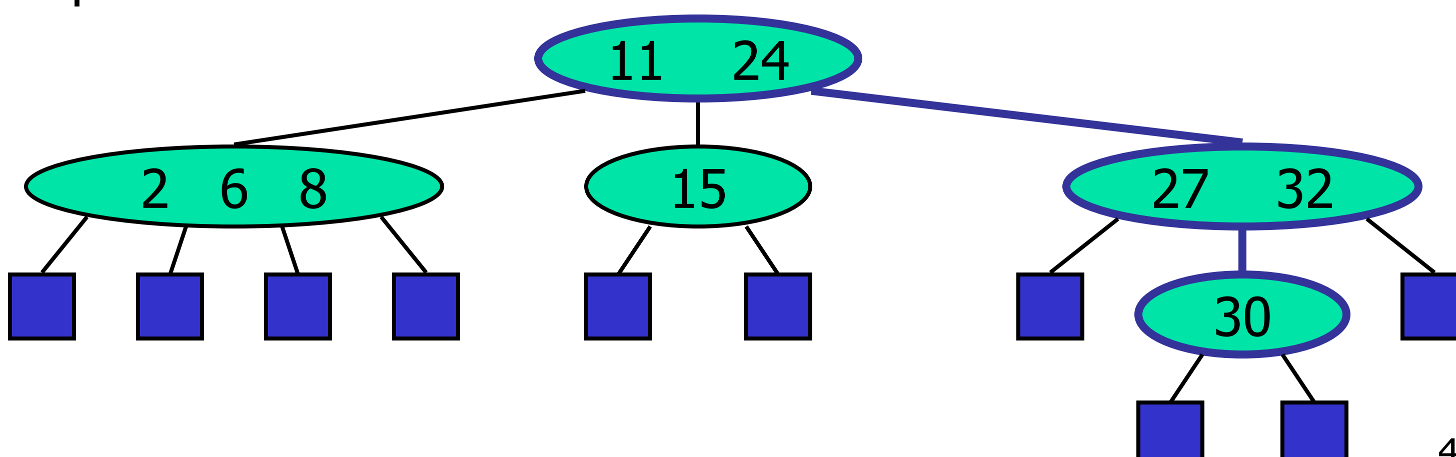
Multi-Way Inorder Traversal

- 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



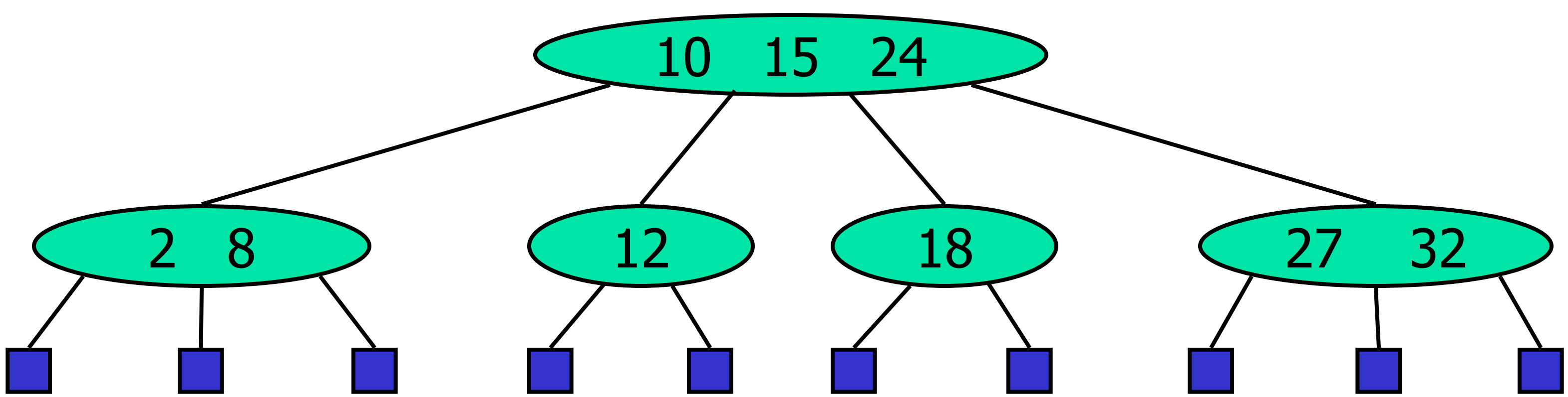
Multi-Way Searching

- Similar to search in a binary search tree
- 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
- 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

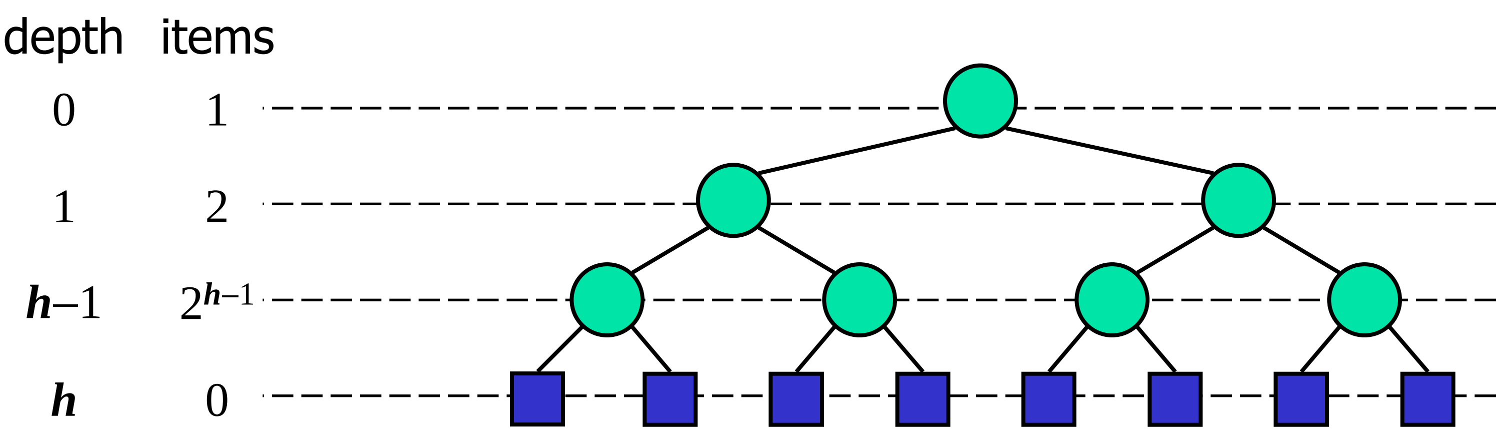


Height of a (2,4) Tree

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

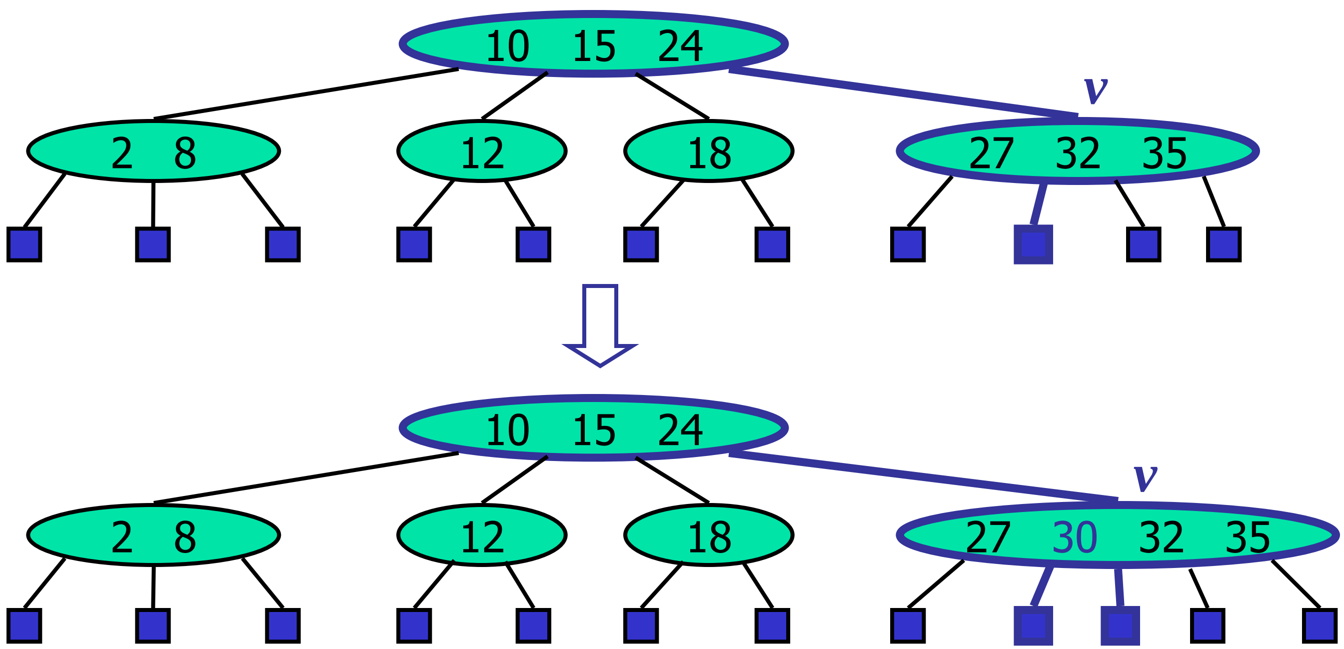
  Proof:
  - Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \ldots, h-1$ and no items at depth h, we have

    $$n \geq 1 + 2 + 4 + \ldots + 2^{h-1} = 2^h - 1$$
  - Thus, $h \leq \log(n + 1)$
- Searching in a (2,4) tree with n items takes $O(\log n)$ time



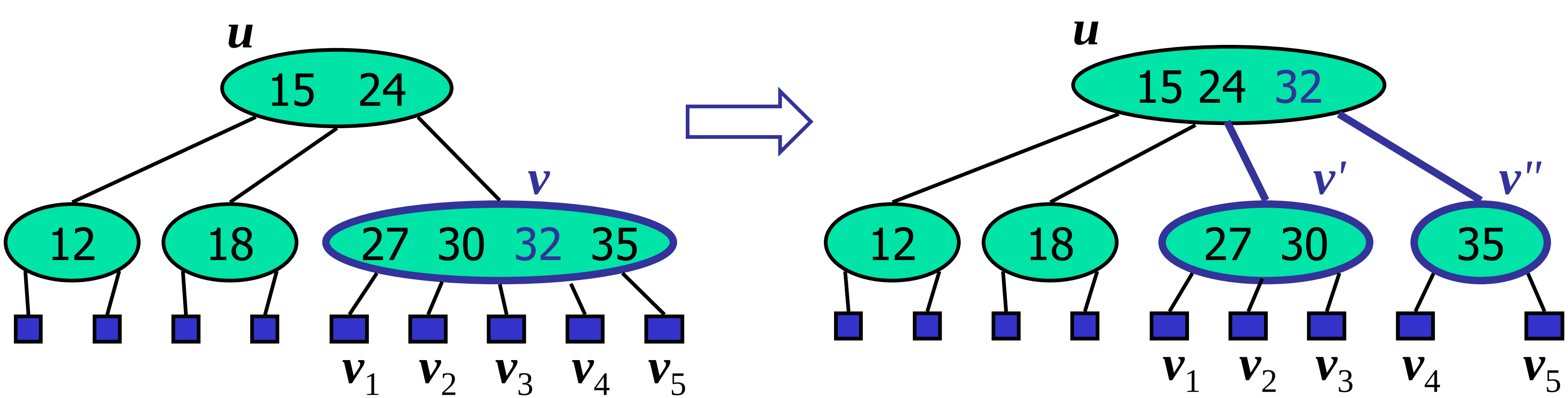
Insertion

- 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 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 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



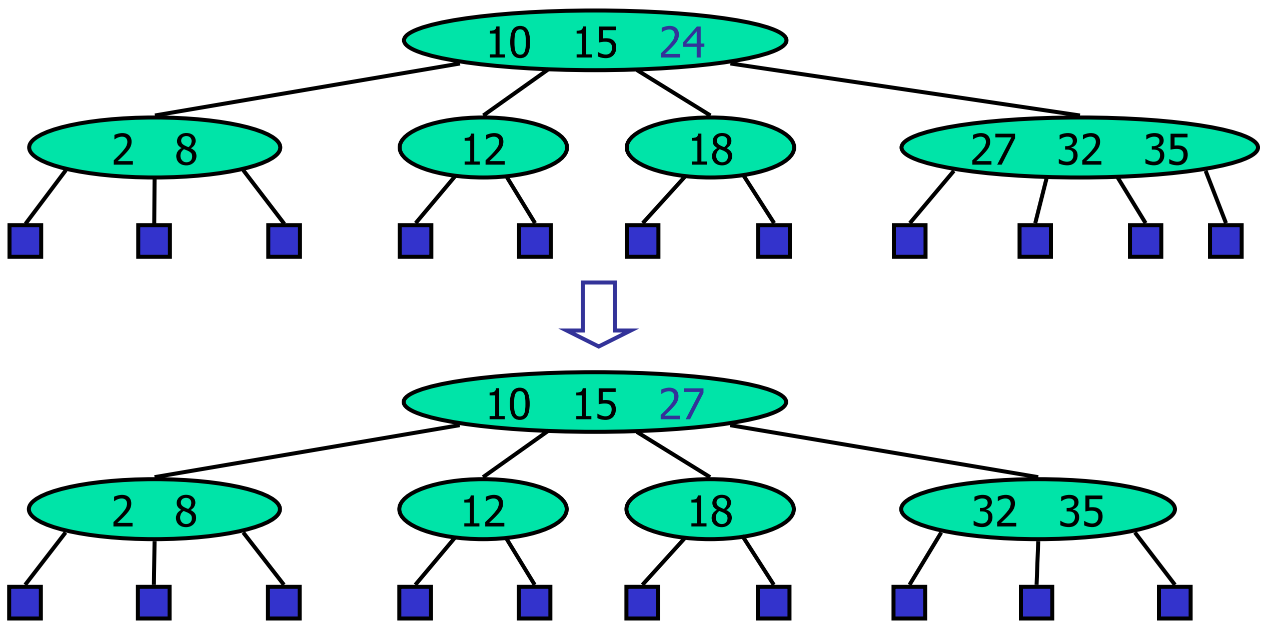
Analysis of Insertion

```
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

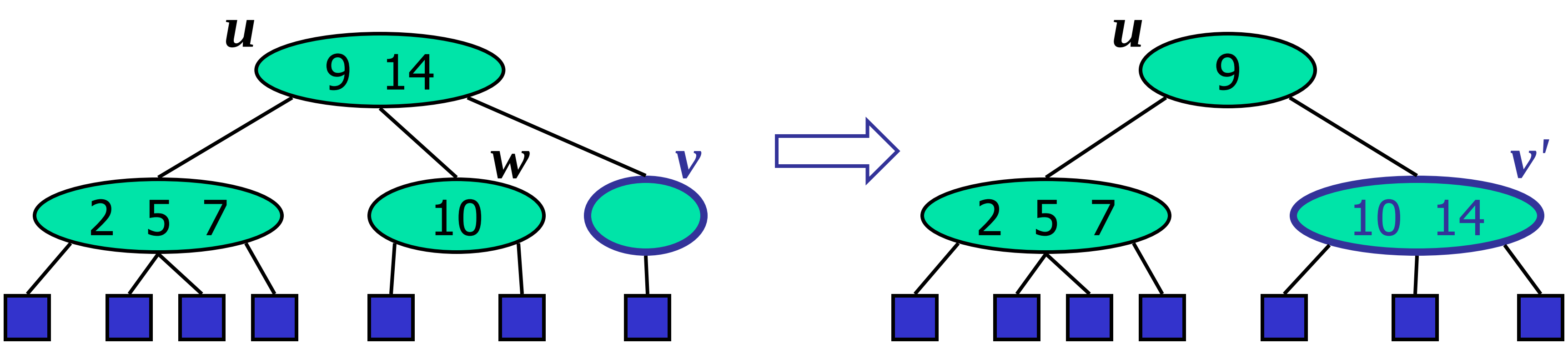
Deletion

- We reduce deletion of an entry to the case where the item is at 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: to delete key 24, we replace it with 27 (inorder successor)



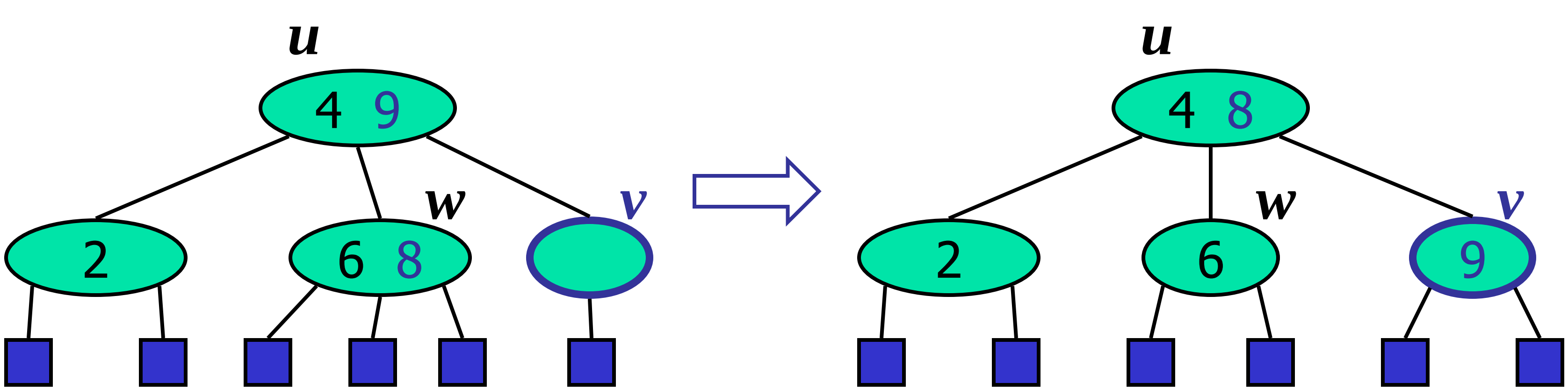
Underflow and Fusion

- 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 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



Analysis of Deletion

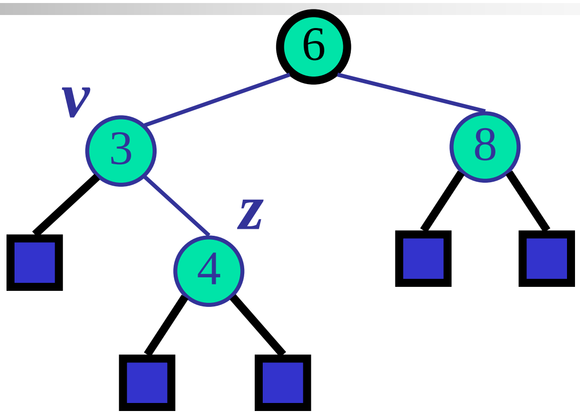
- Let T be a (2,4) tree with n items
  - Tree T has $O(\log n)$ height
- In a deletion operation
  - We visit $O(\log n)$ nodes to locate the node from which to delete the entry
  - We handle an underflow with a series of $O(\log n)$ fusions, followed by at most one transfer
  - Each fusion and transfer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

Implementing a Dictionary

- Comparison of efficient dictionary implementations

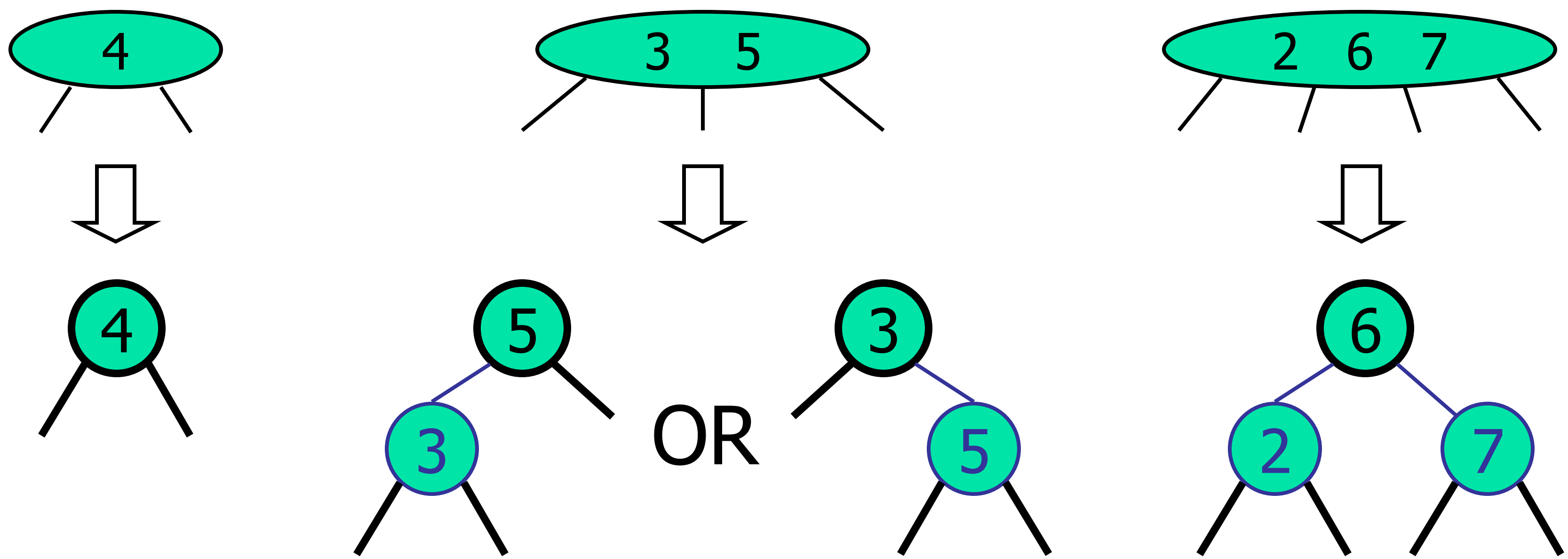
	Search	Insert	Delete	Notes
Hash Table	1 expected	1 expected	1 expected	▪ no ordered dictionary methods ▪ simple to implement
Skip List	log ***n*** high prob.	log ***n*** high prob.	log ***n*** high prob.	▪ randomized insertion ▪ simple to implement
(2,4) Tree	log ***n*** worst-case	log ***n*** worst-case	log ***n*** worst-case	complex to implement

Red-Black Trees



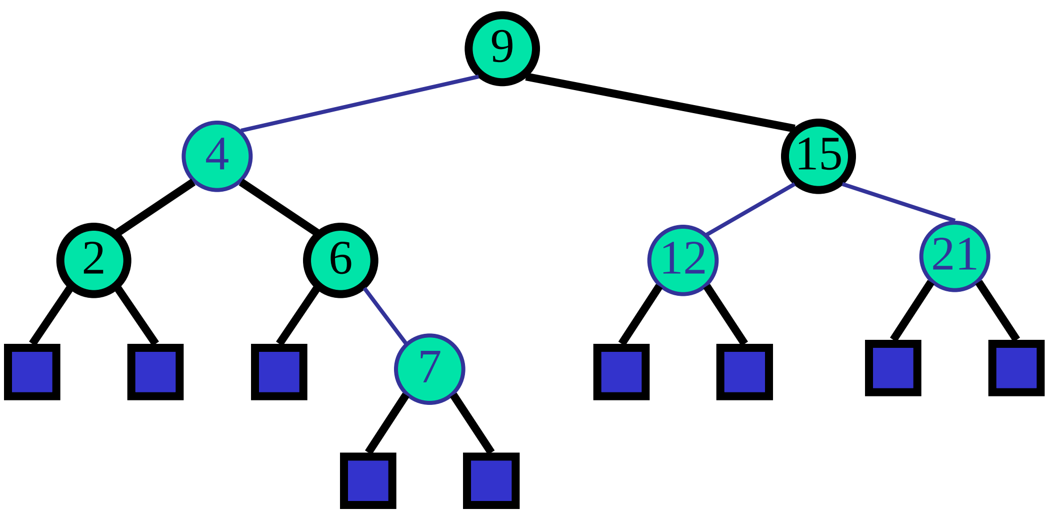
From (2,4) to Red-Black Trees

- 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 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



Height of a Red-Black Tree

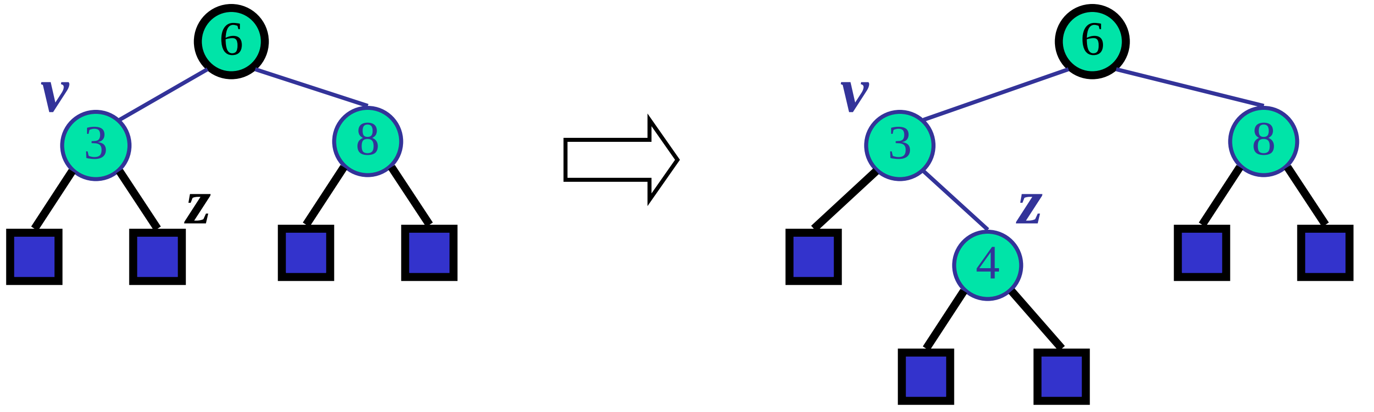
- Theorem: A red-black tree storing n entries has height $O(\log n)$

  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 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:

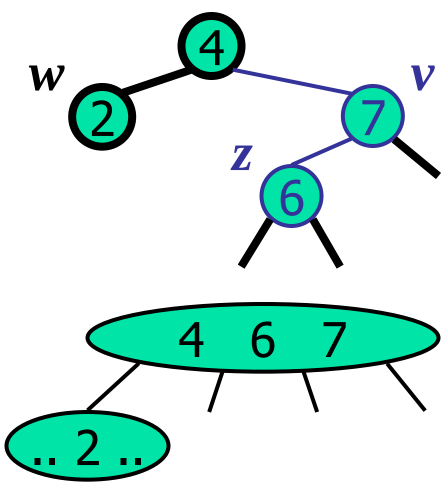


Remedying a Double Red

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

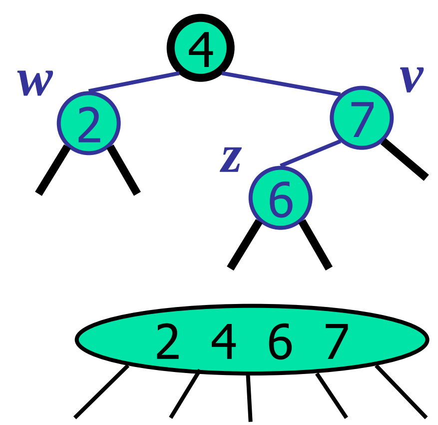
Case 1: w is black

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



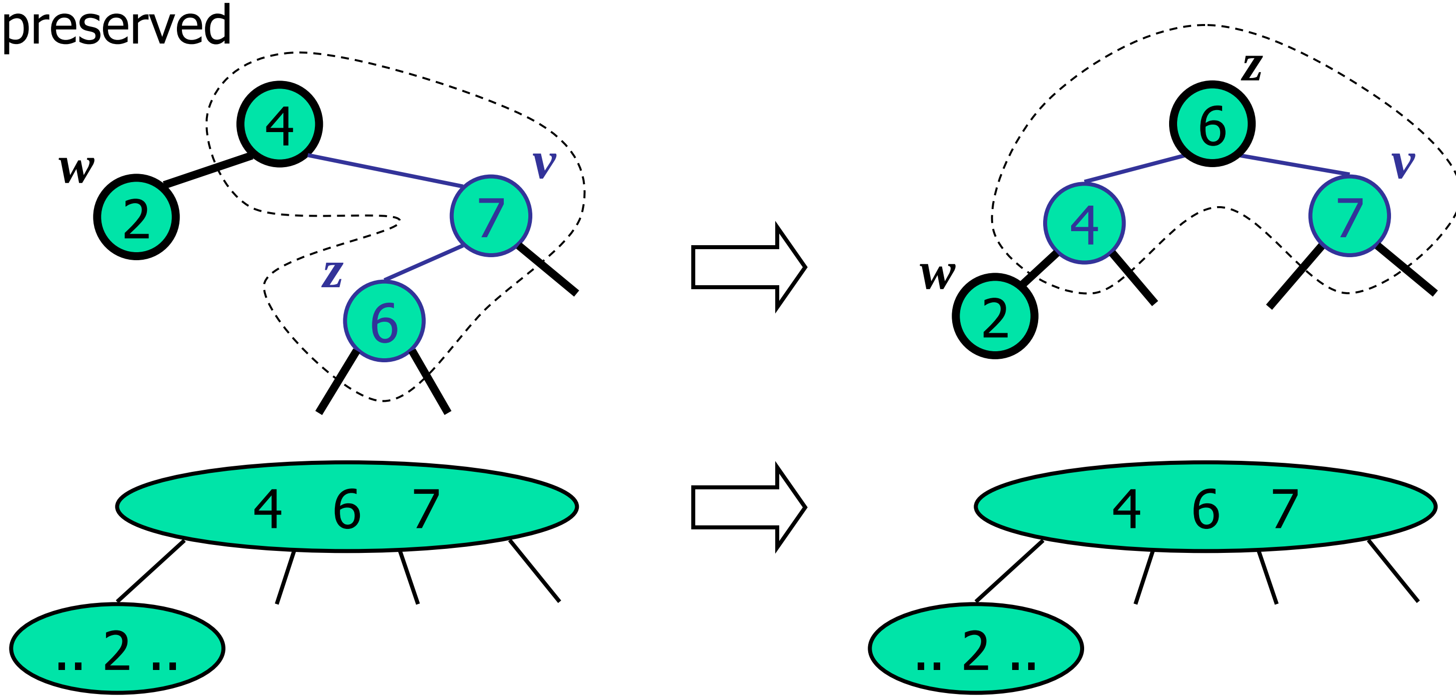
Case 2: w is red

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



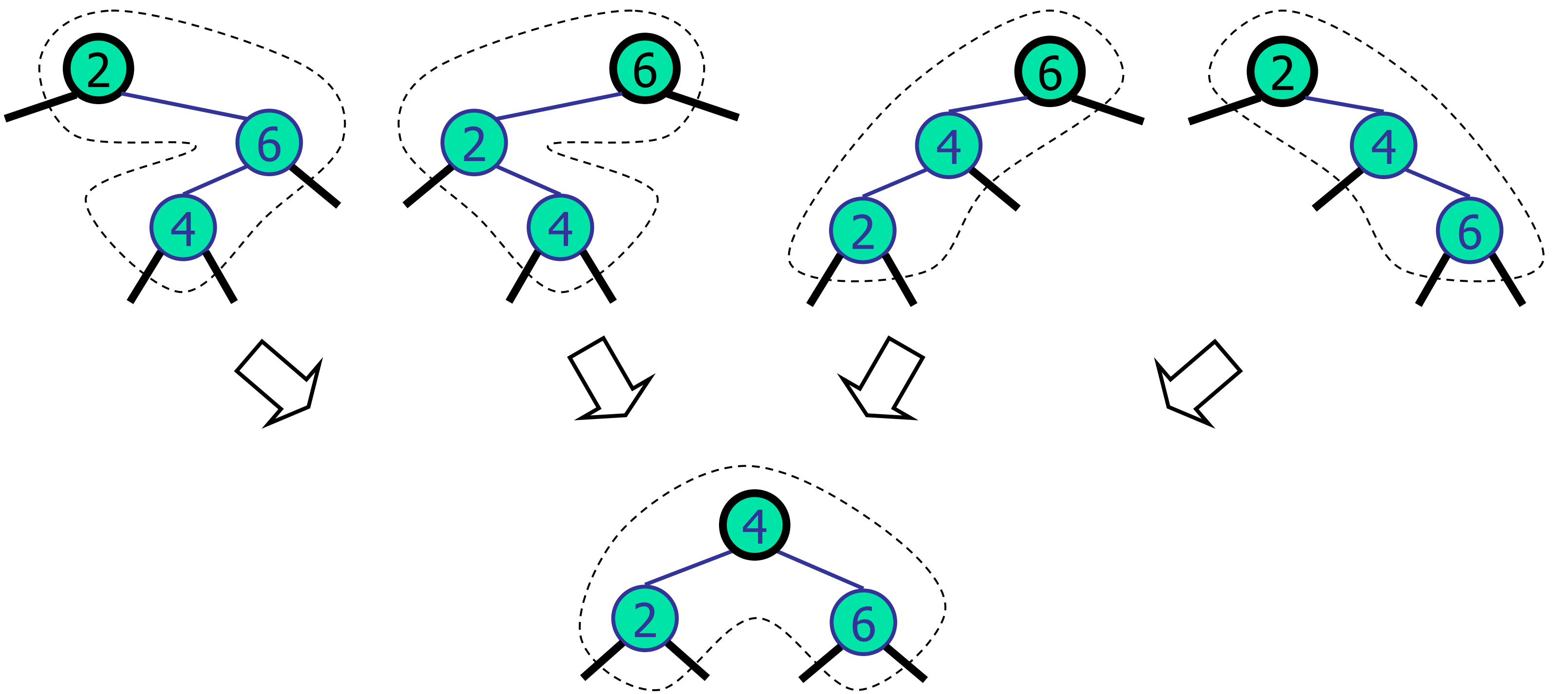
Restructuring (1/2)

- 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



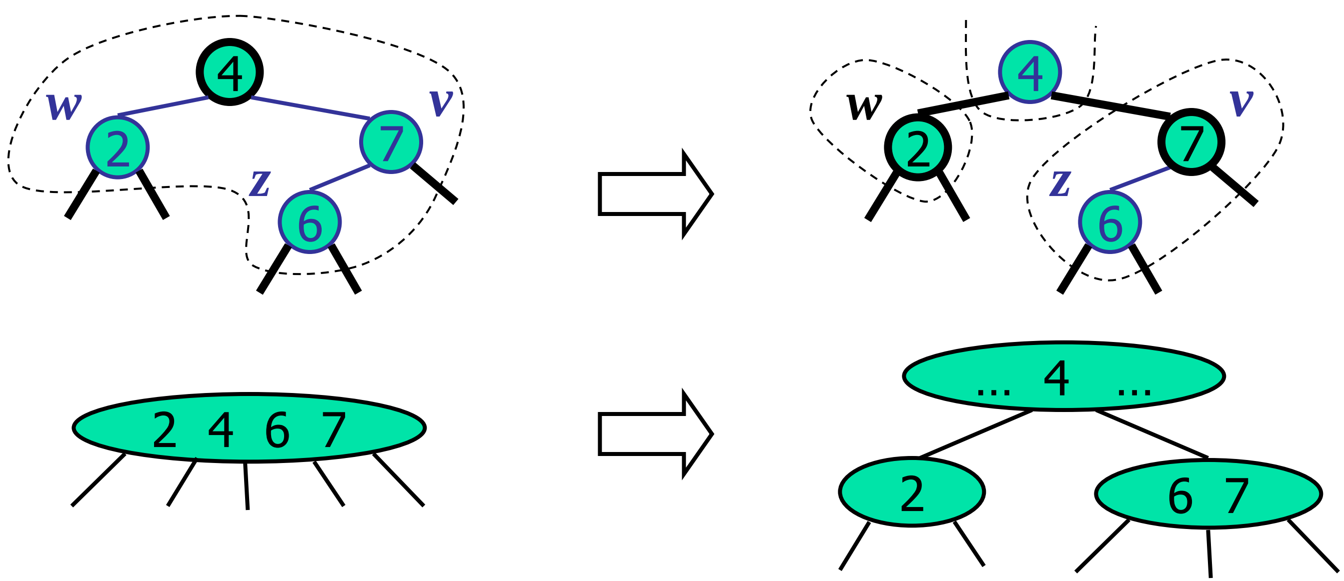
Restructuring (2/2)

- 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



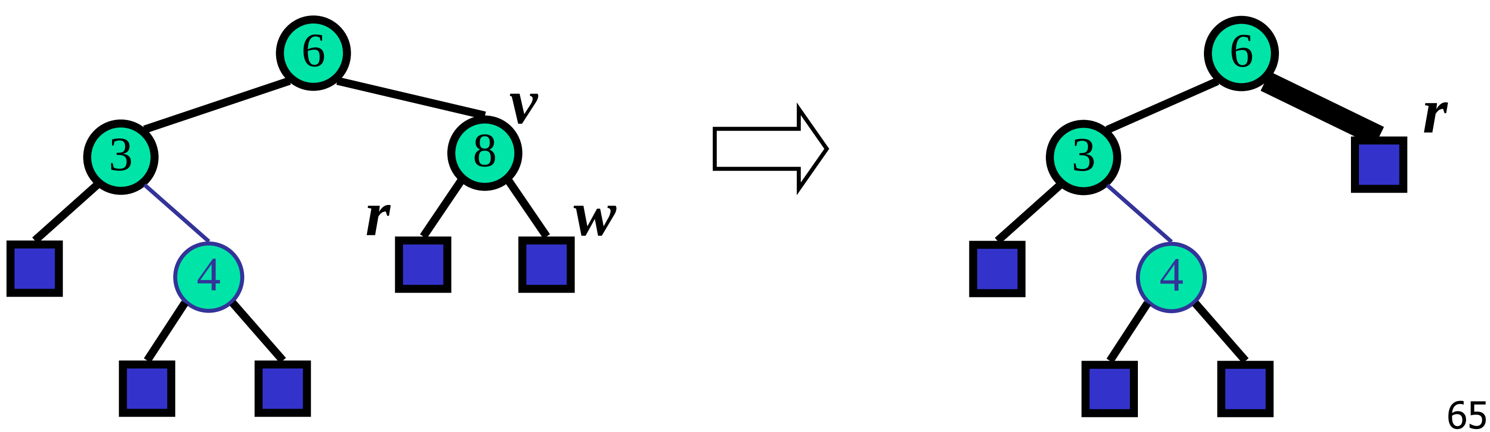
Analysis of Insertion

```
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

- To perform operation remove(k), we first execute the deletion algorithm for binary search trees
- Let v be the internal node removed, w the external node removed, and r the sibling of w
  - If either v of r was red, we color r black and we are done
  - Else (v and r were both black) we color r ***double black***, which is a violation of the internal property requiring a reorganization of the tree
- Example where the deletion of 8 causes a double black:



Remedying a Double Black

- The algorithm for remedying a double black node w with sibling y considers 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 in a red-black tree takes $O(\log n)$ time

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

References

1. Chapter 10, Data Structures and Algorithms by Goodrich and Tamassia.