SE274 Data Structure

Lecture 8: Search Trees, Part 3

(textbook: Chapter 11, Splay Tree, 2-4 Tree)

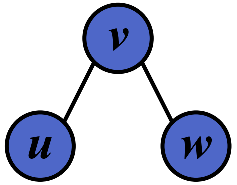
Apr 20, 2020

Instructor: Sunjun Kim

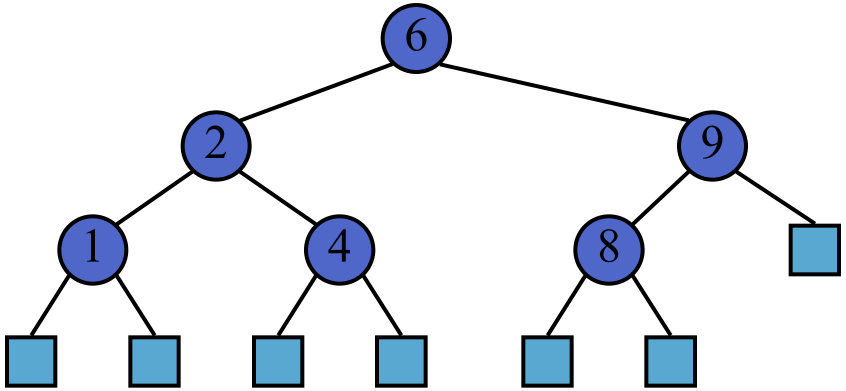
Information&Communication Engineering, DGIST

Recap: Binary Search Trees

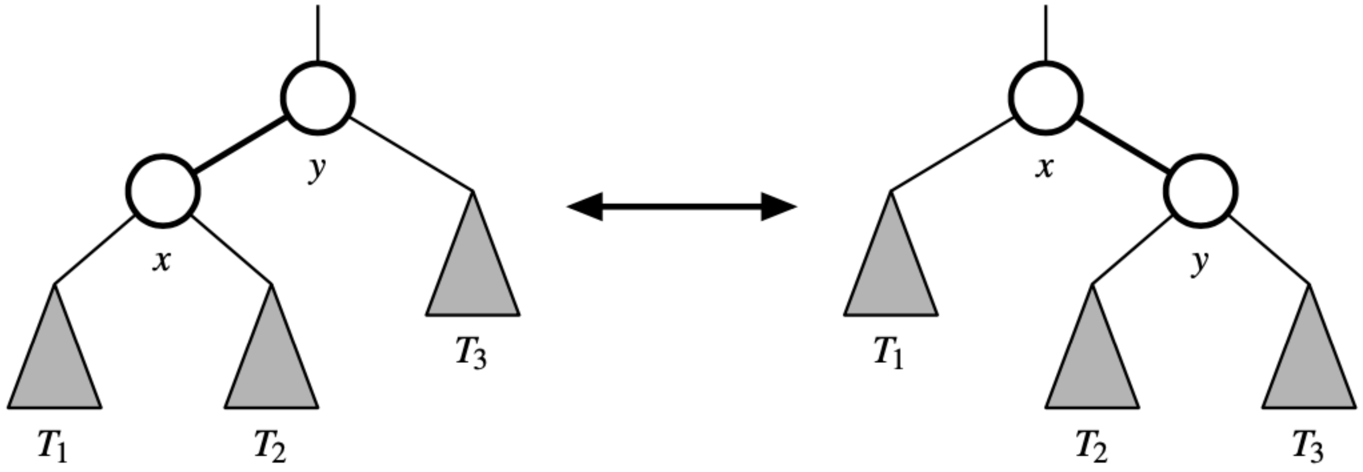
- A binary search tree is a binary tree storing keys (or key-value items) at its nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items, instead we consider them as None



- An inorder traversal of a binary search trees visits the keys in increasing order

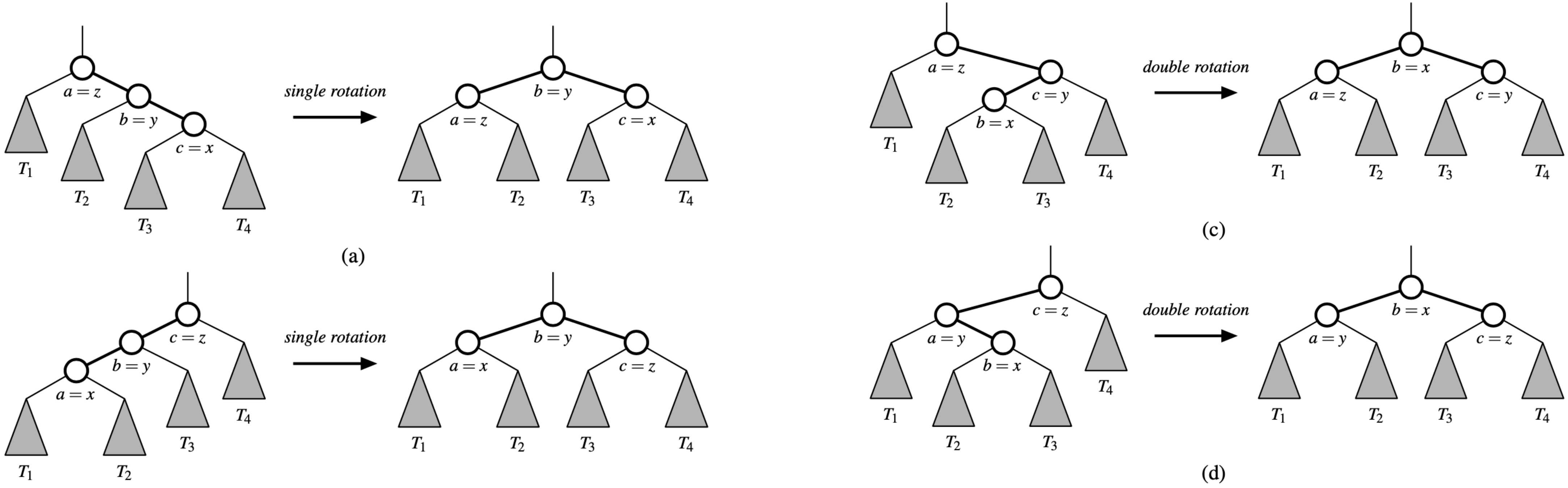


Recap: Tree Rotation Operation



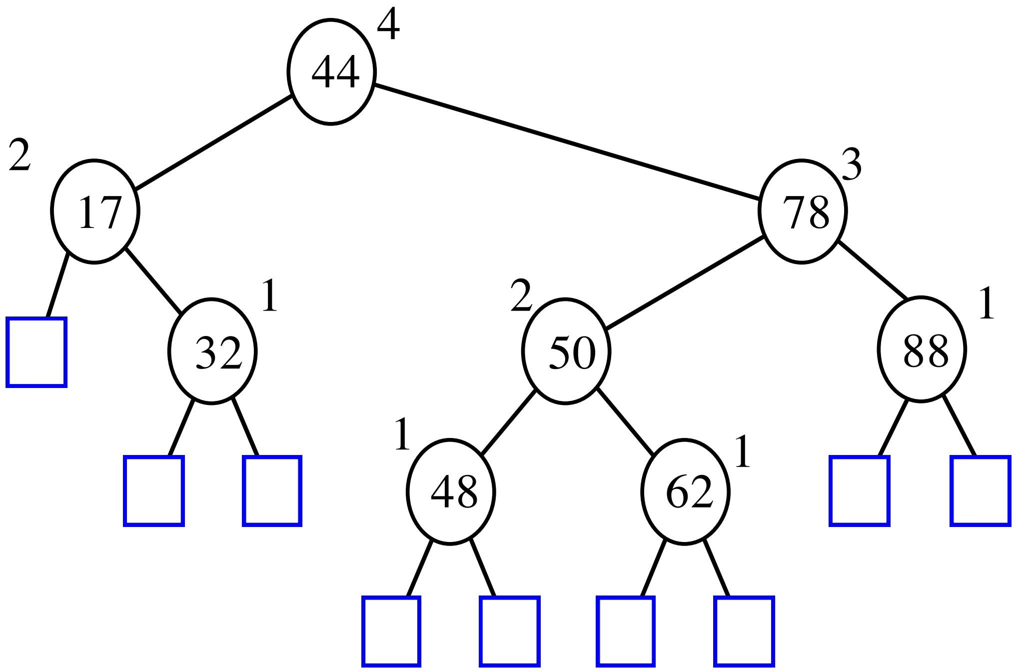
Recap: Trinode reconstruction

- Nodes: (a,b,c) -> inorder listing of the three positions x, y, z
- Sub-trees: (T_1, T_2, T_3, T_4) -> inorder listing of the four subtrees



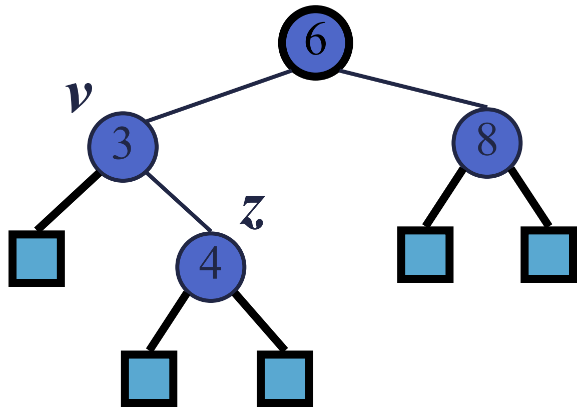
Recap: AVL Tree

- 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:

Splay Trees



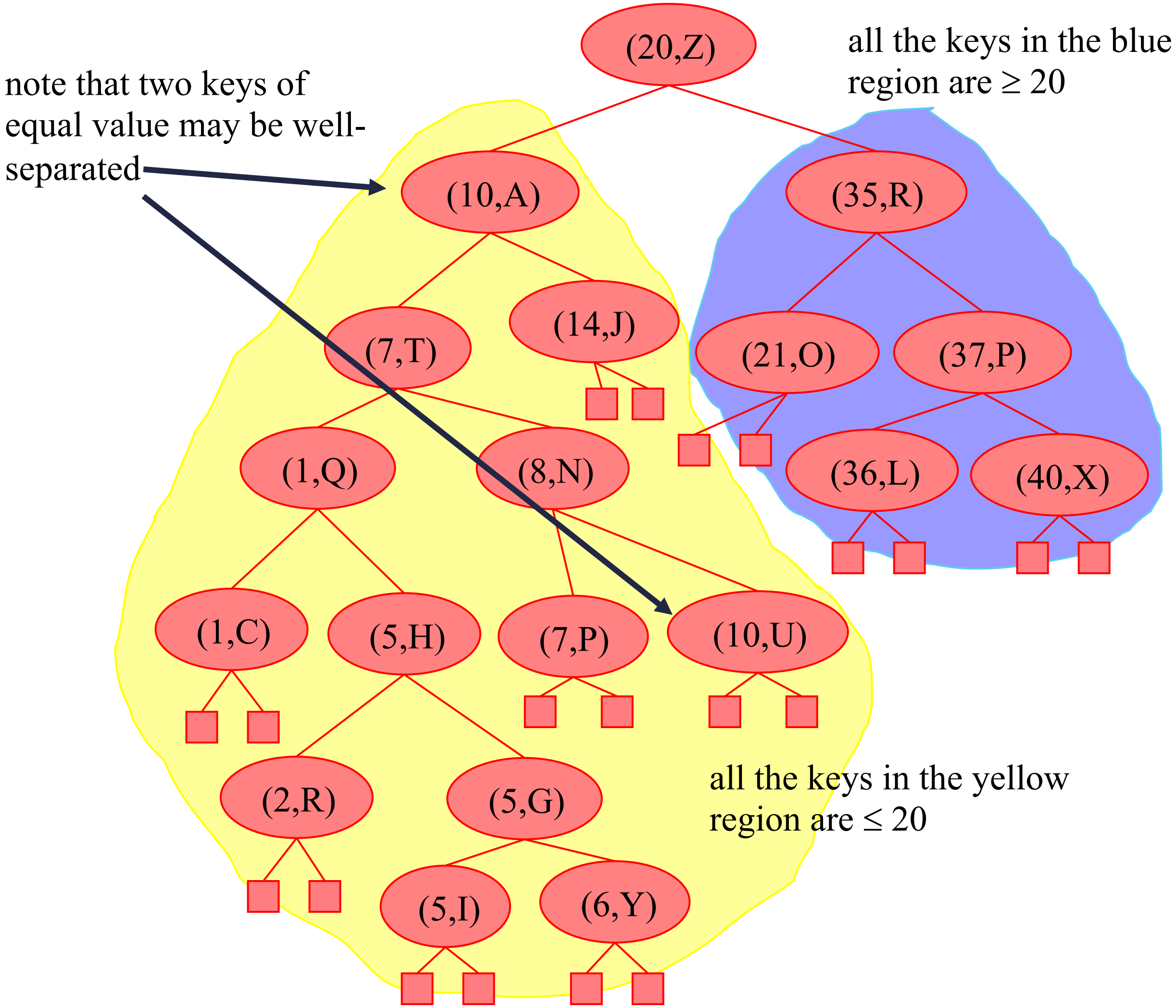
Splay tree principle

Reorder the recenctly accessed item to the ***root***

- In non-random access, O(1) is often expected.
- ***Best data structure for a cache application.***

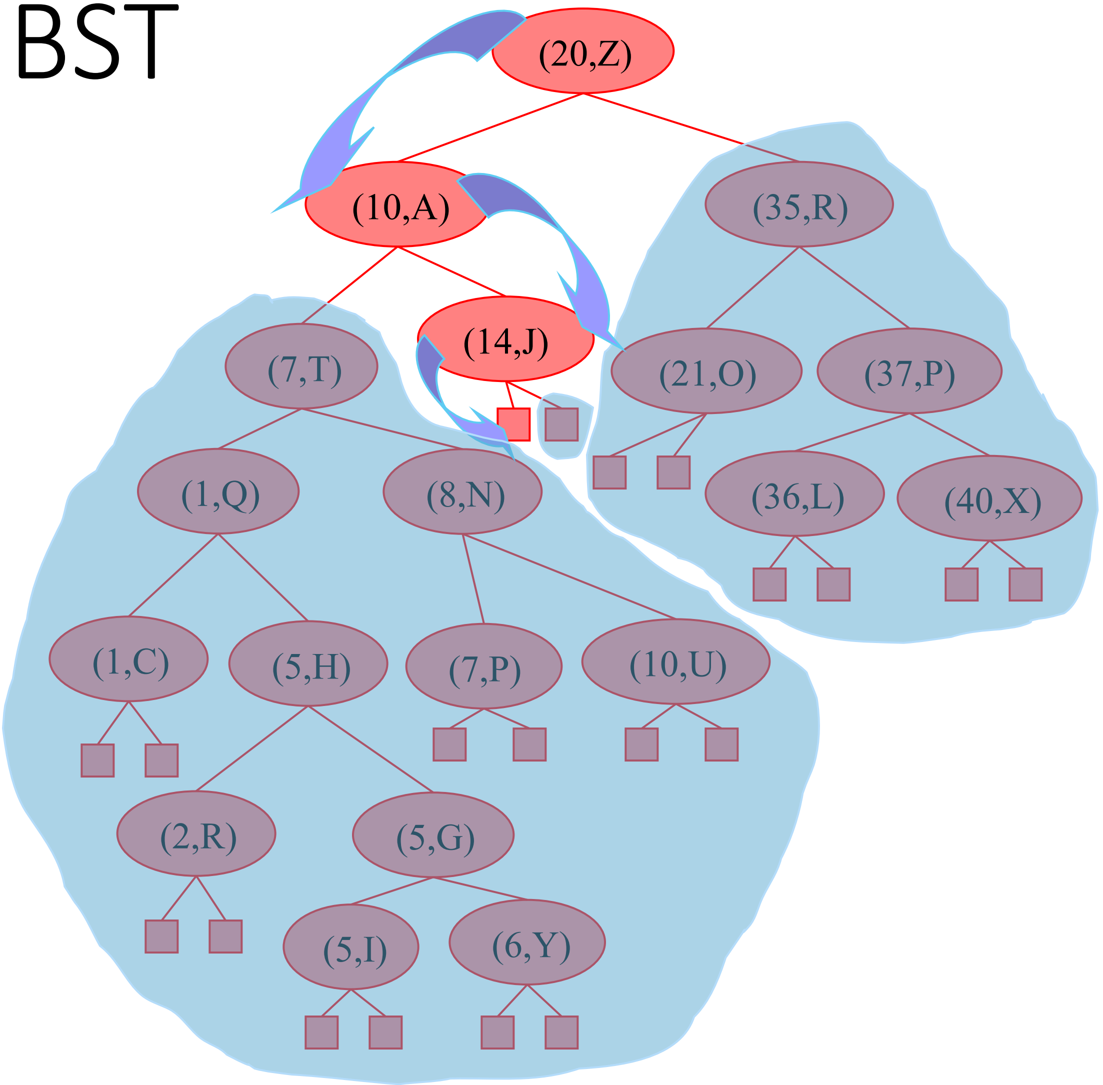
Splay Trees are Binary Search Trees

- 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



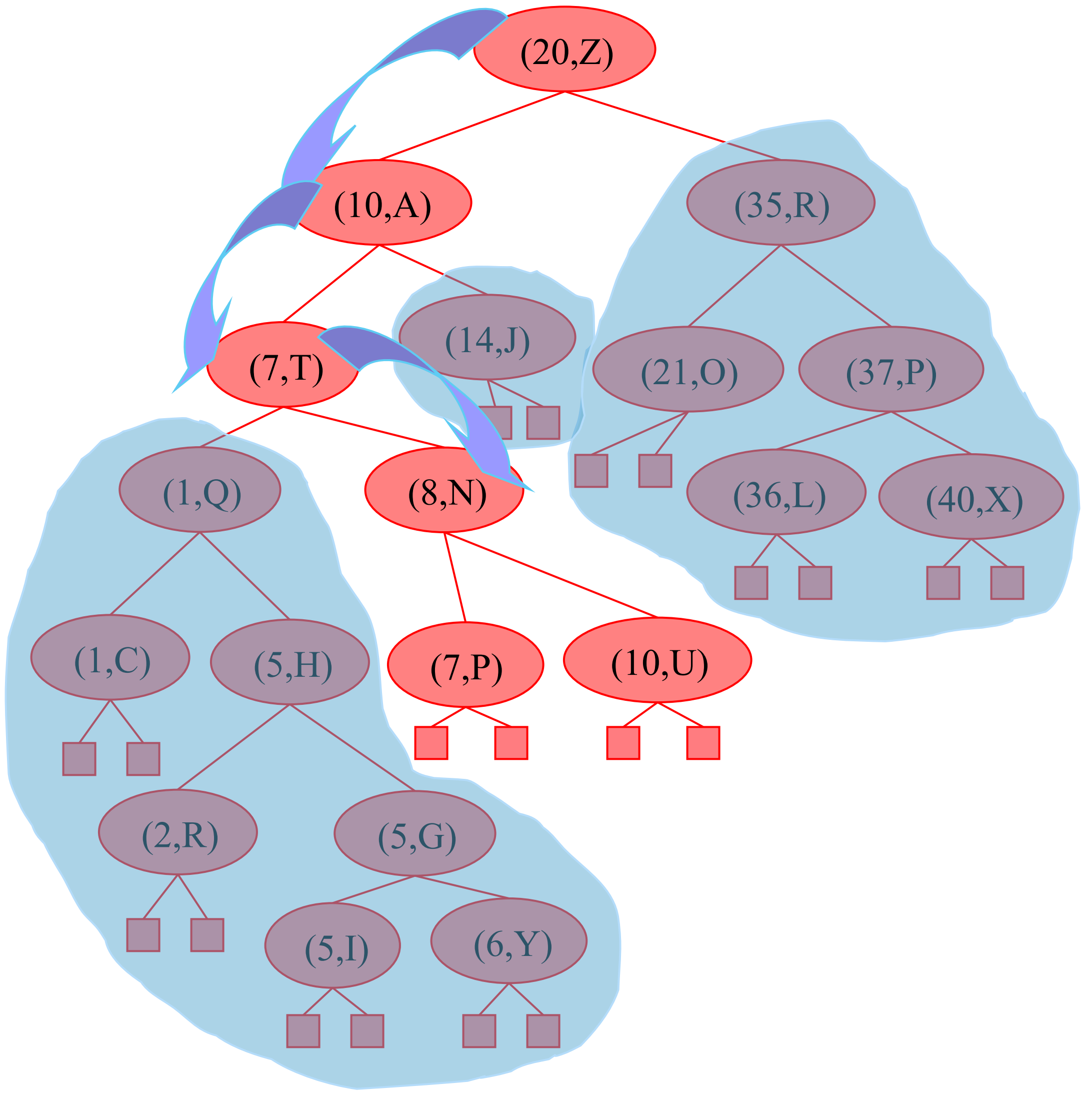
Searching in a Splay Tree: Starts the Same as in a BST

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



Example Searching in a BST, continued

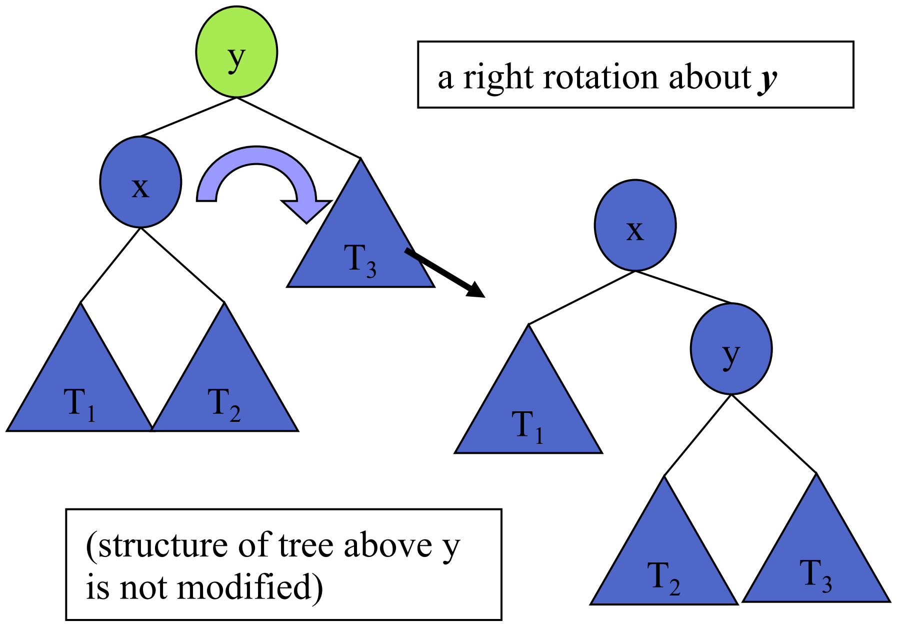
- search for key 8, ends at an internal node.



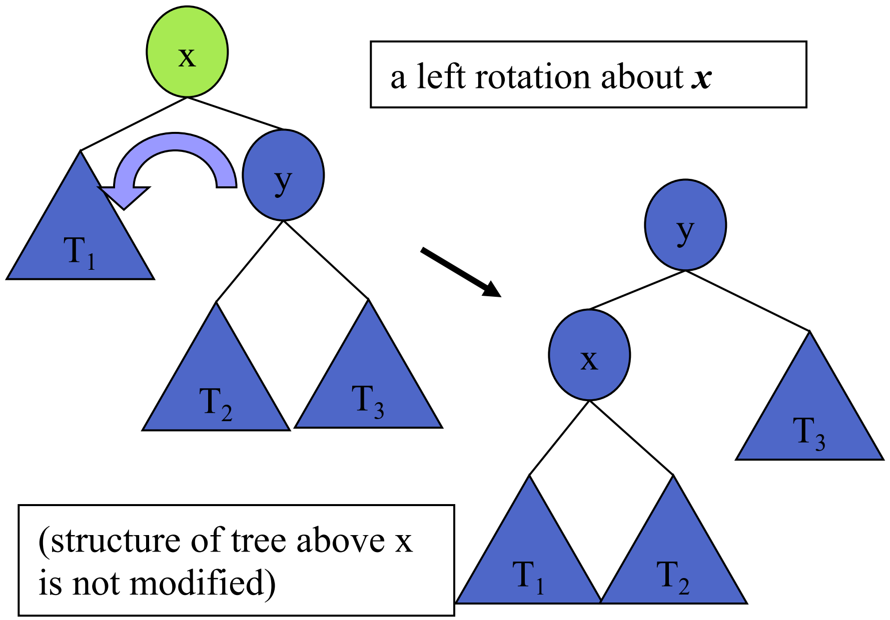
Splay Trees do Rotations after Every Operation (Even Search)

- 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

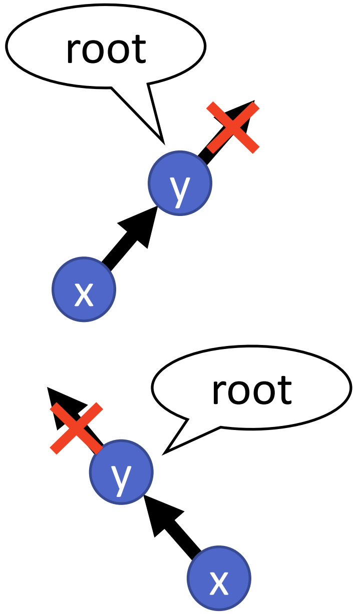


- left rotation
  - makes the right child y of a node x into x' s parent; x becomes the left child of y

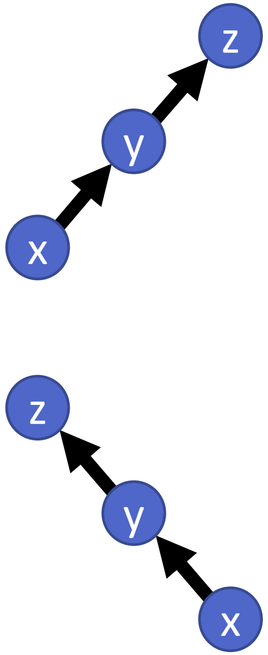


Visualizing the Splaying cases

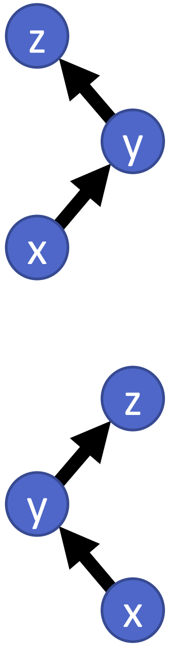
Zig



Zig-zig

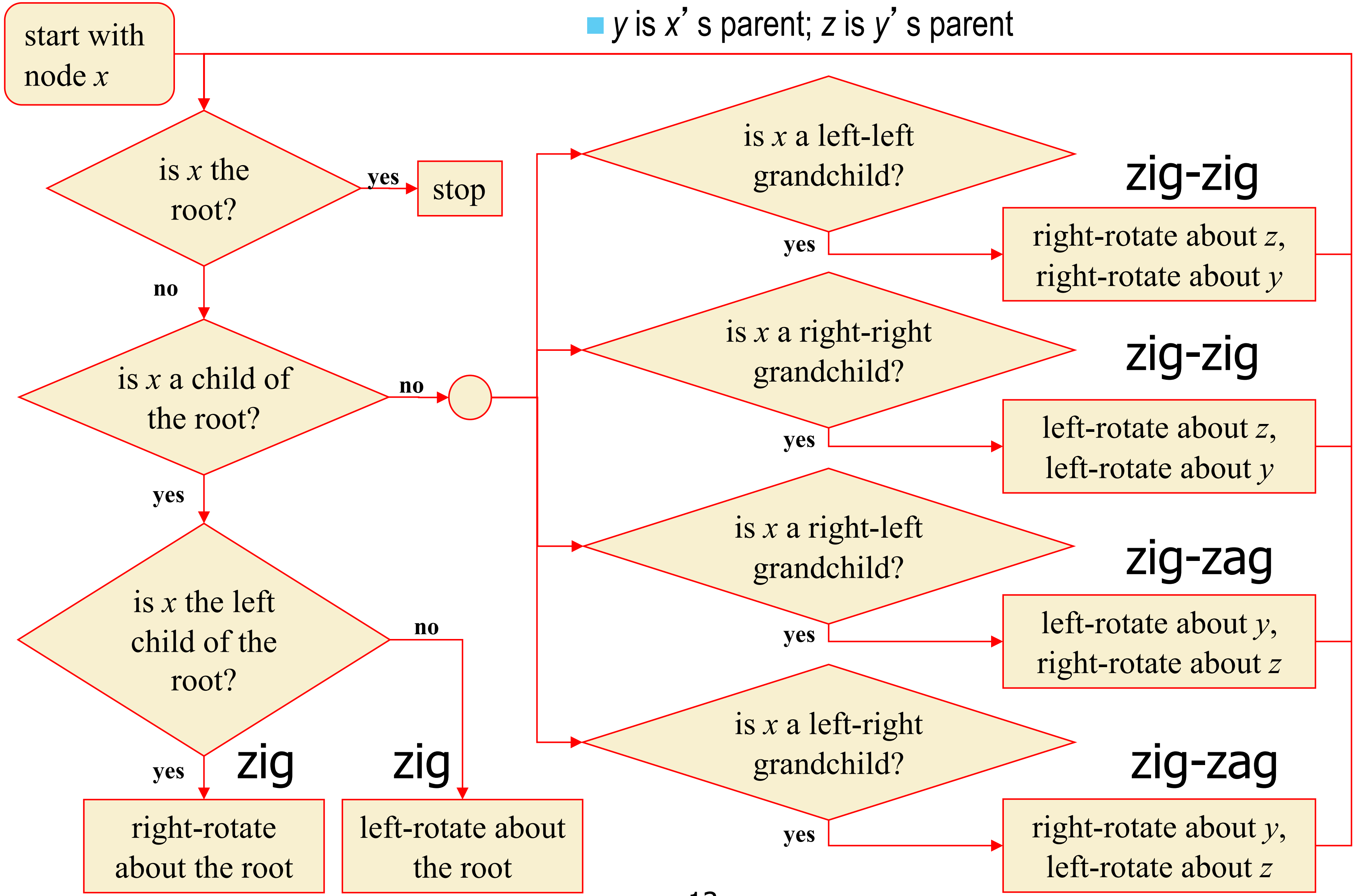


Zig-zag

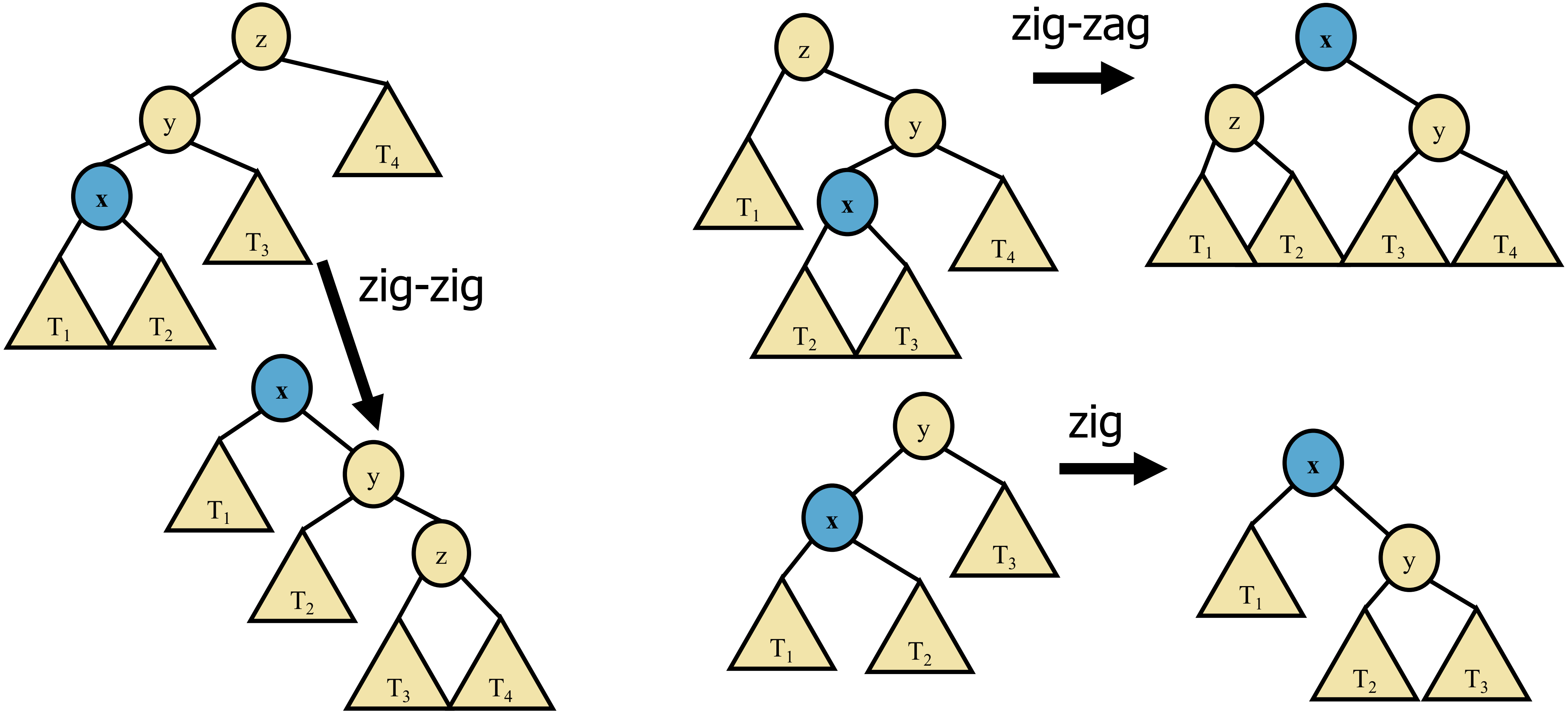


Splaying:

- "x is a left-left grandchild" means x is a left child of its parent, which is itself a left child of its parent
- y is x' s parent; z is y' s parent

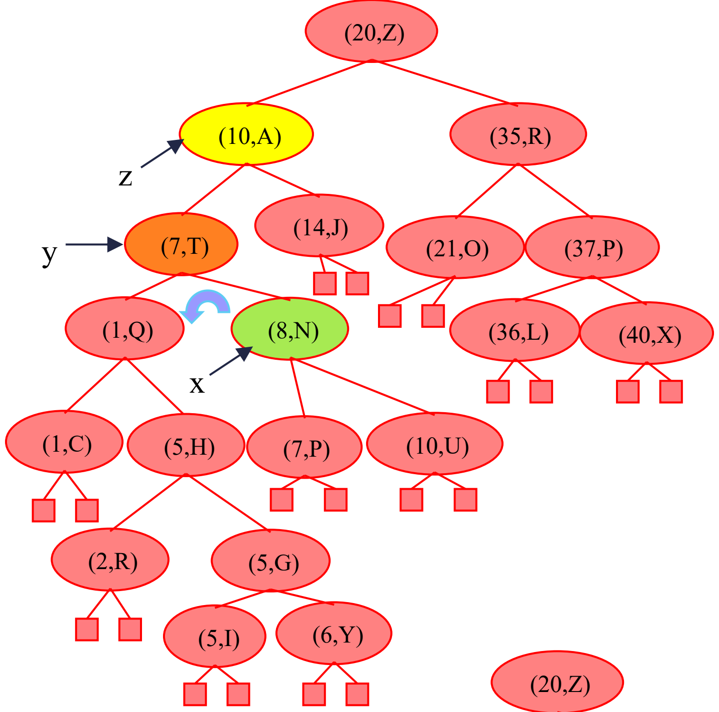


Visualizing the Splaying cases

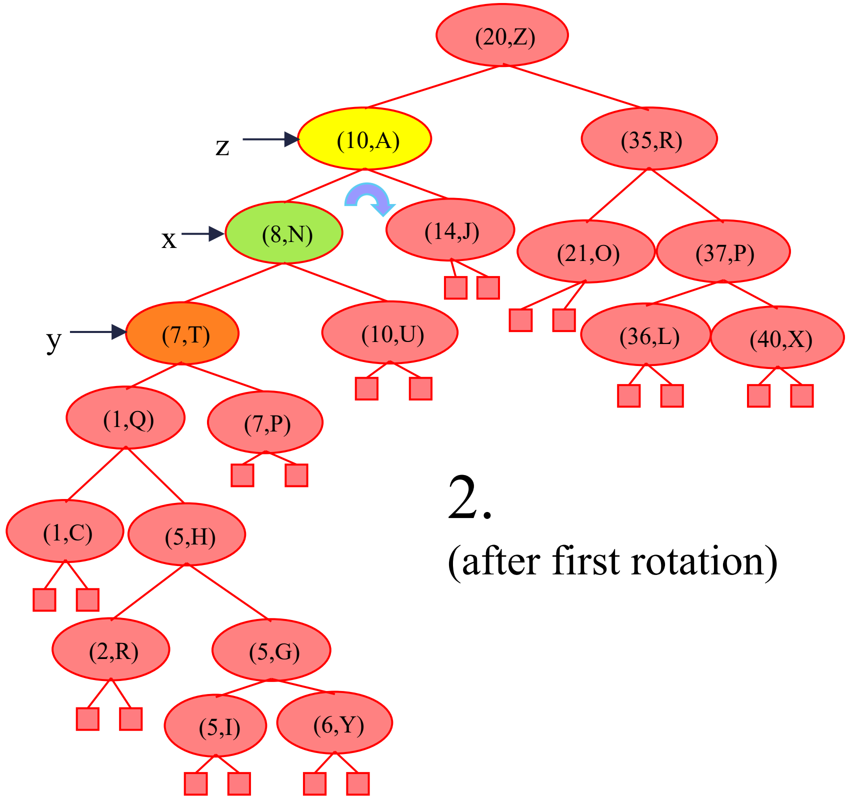


Splaying Example

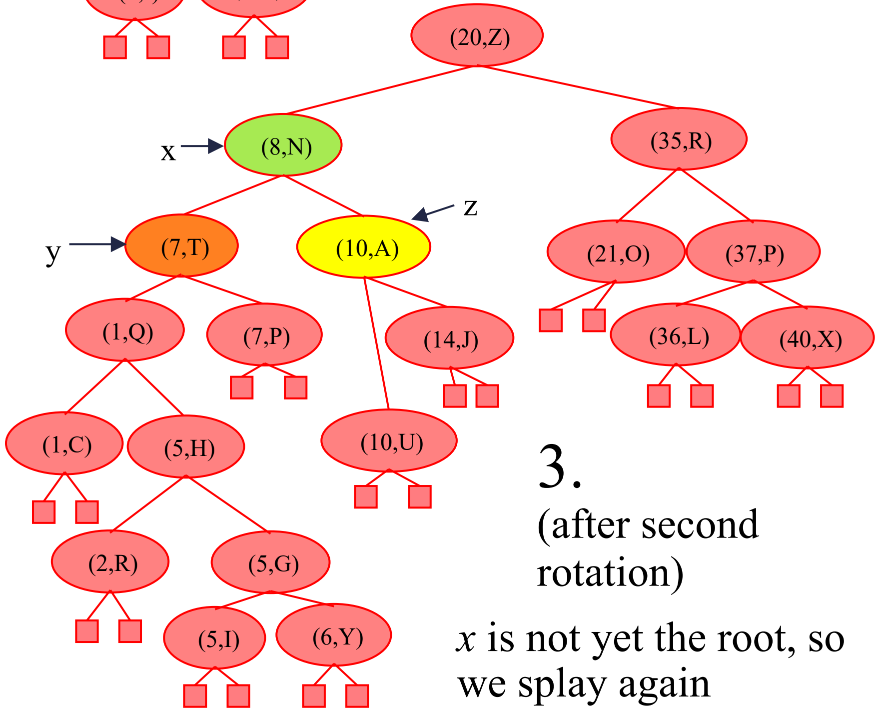
- let x = (8,N)
  - x is the right child of its parent, which is the left child of the grandparent
  - left-rotate about y, then right-rotate around z



1.
(before rotating)
- zig-zag case



2.
(after first rotation)

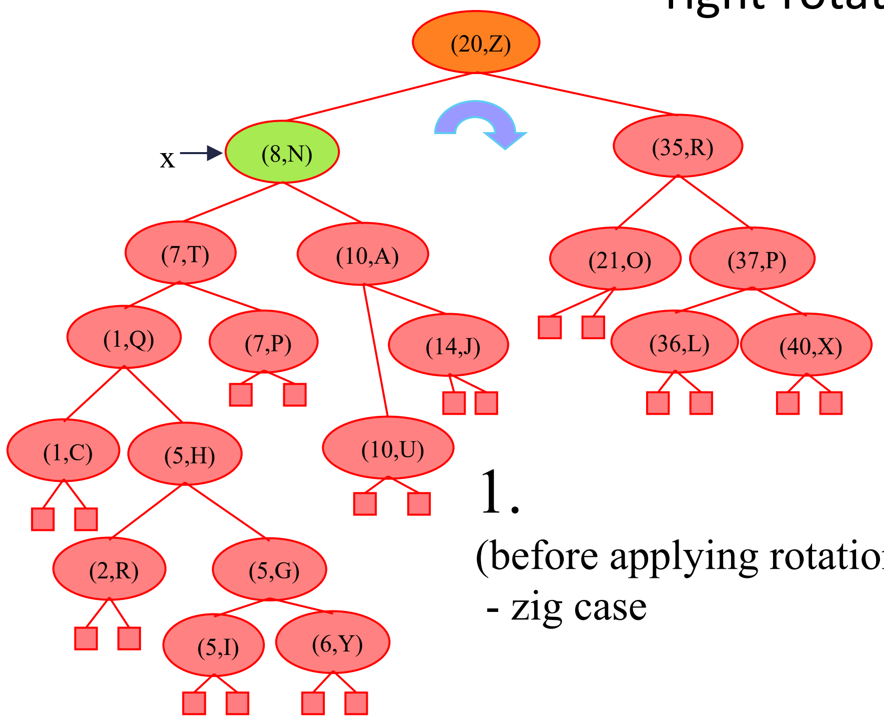


3.
(after second rotation)

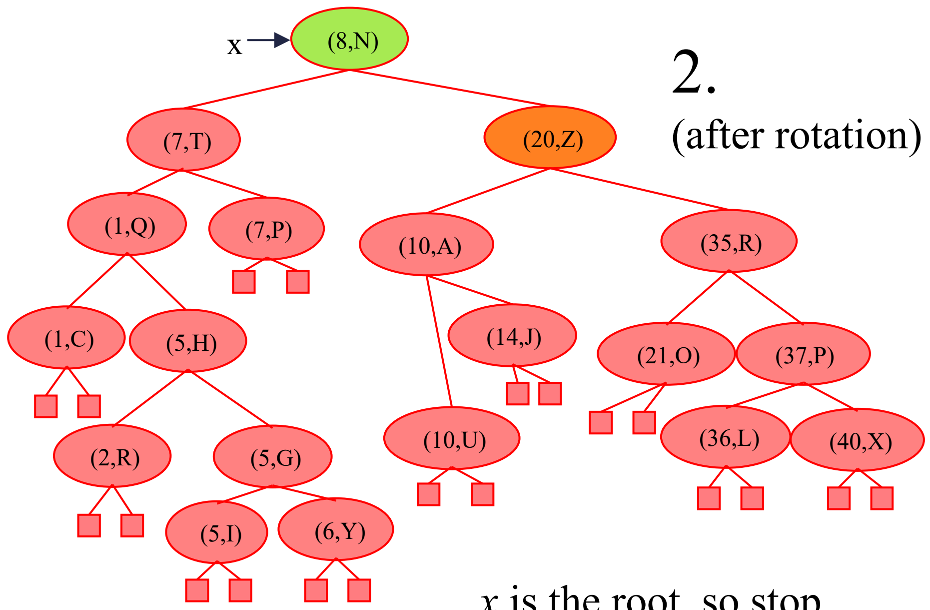
x is not yet the root, so we splay again

Splaying Example, Continued

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



1.
(before applying rotation)
- zig case

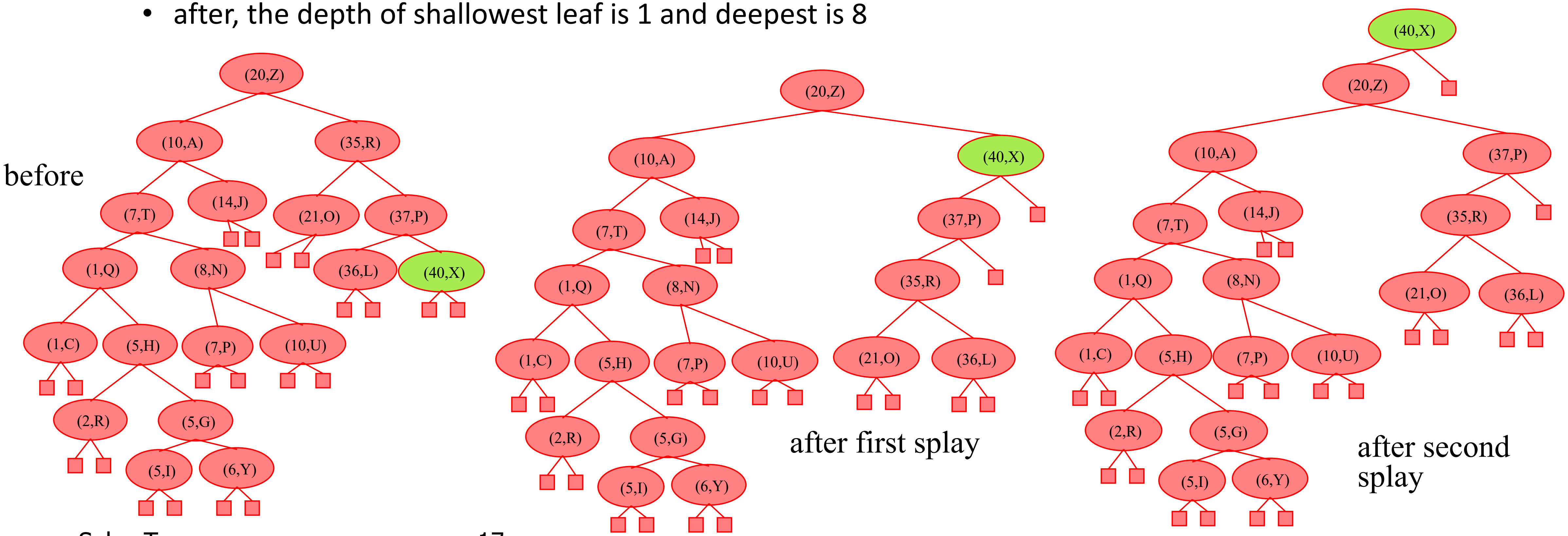


2.
(after rotation)

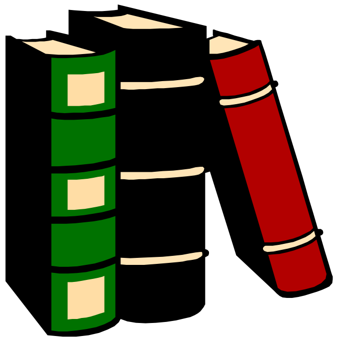
x is the root, so stop

Example Result of Splaying

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



Splay Tree Definition



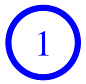
- a splay tree is a binary search tree where a node is splayed after it is accessed (for a search or update)
  - deepest internal node accessed is splayed
  - splaying costs O(h), where h is height of the tree
    – which is still O(n) worst-case
    - O(h) rotations, each of which is O(1)

Splay tree search/insertion/deletion

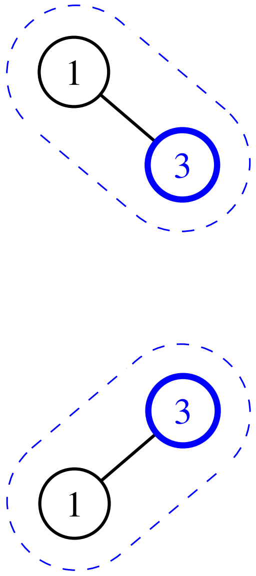
- **Search for a key *k***
  - If *k* is found at position *p*, we splay *p.*
  - Else, we splay the leaf position which the search was terminated.
- **Insert a key *k***
  - Do a regular BST insertion.
  - The newly created internal node for *k* is then splayed.
- **Delete a key *k***
  - Do a regular BST deletion.
  - Splay the parent of the actually removed node.

Splay tree insertion example

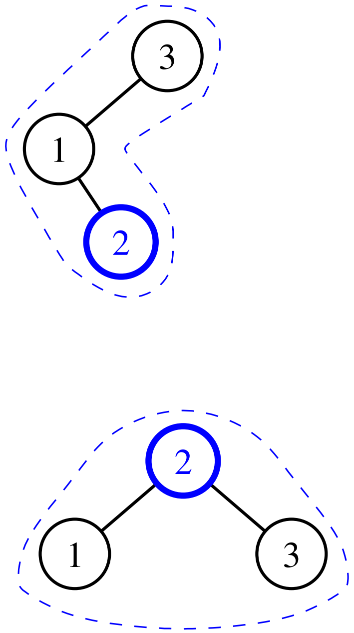
Insert (1)



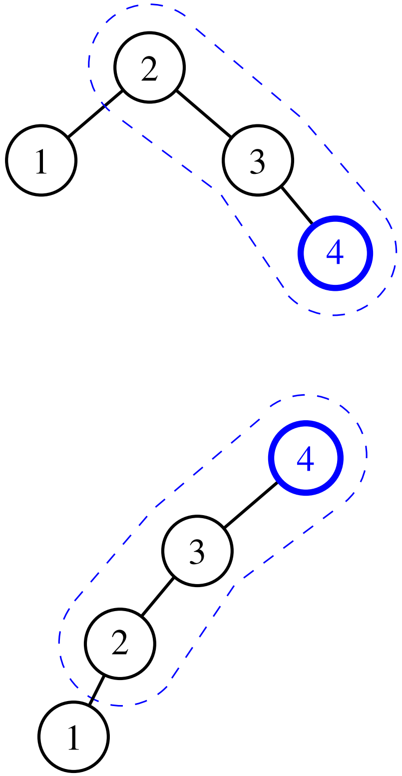
Insert (3)



Insert (2)

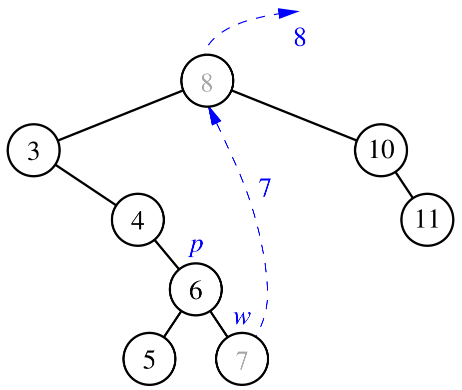


Insert (4)

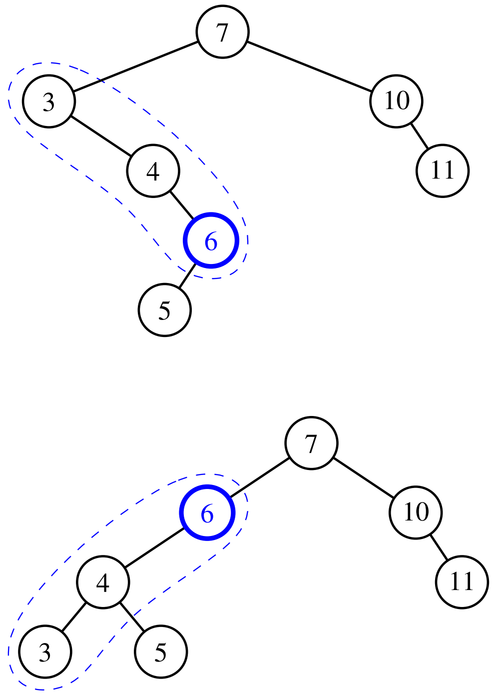


Splay tree deletion example

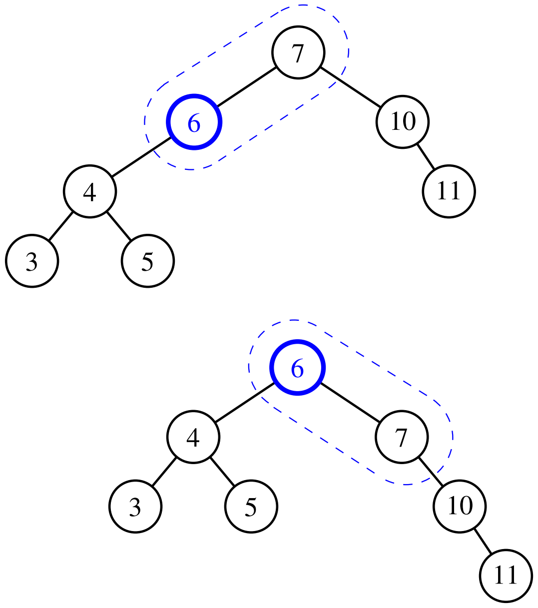
Delete (8)



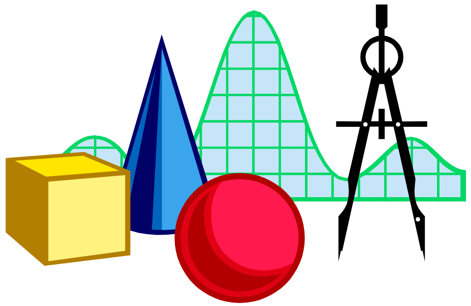
First splay



Second splay



Amortized Analysis of Splay Trees

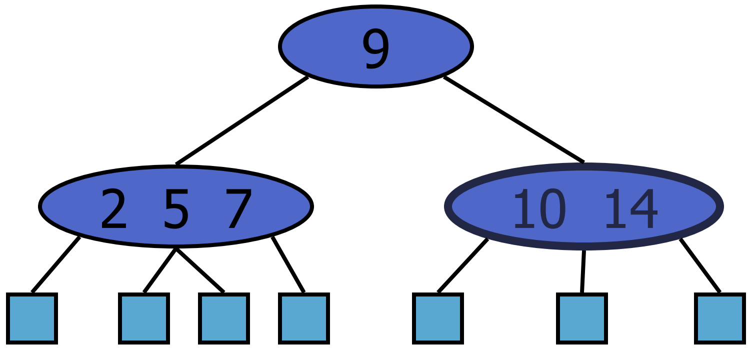


- Running time of each operation is proportional to time for splaying.
  => with a tree height *h*, it's O(*h*).
  - In the worst-case, *h* = *n.*
- Amortized performance of Splay tree operations are done in O(log n).
  - Refer the textbook Chapter 11.4.4 for the detailed analysis
    (the analysis is out of the scope of this course).

Python Implementation

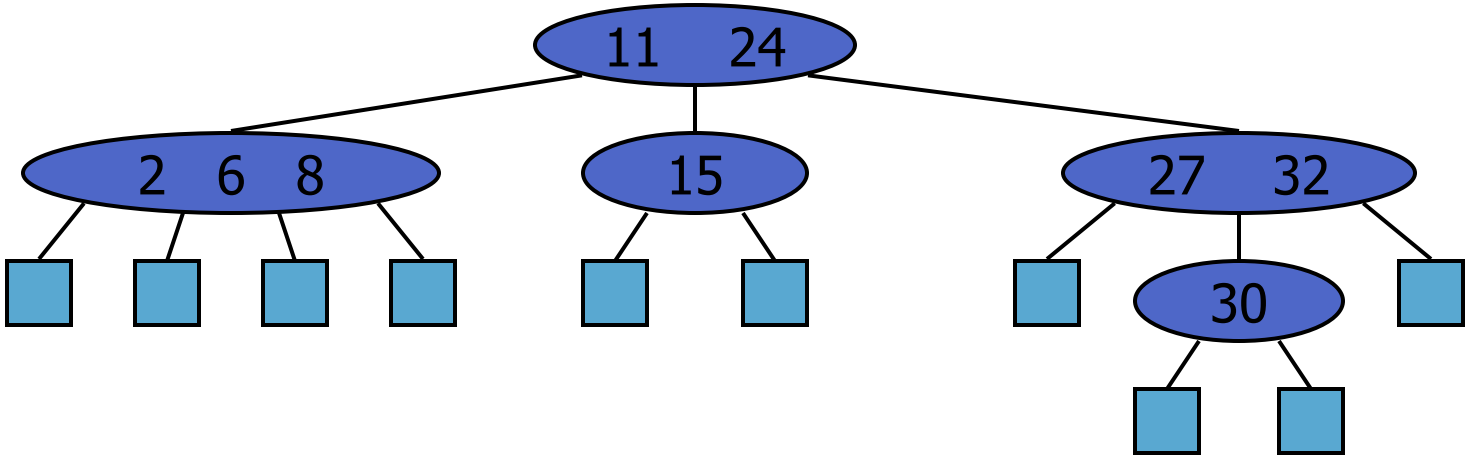
```
class SplayTreeMap(TreeMap):
  """Sorted map implementation using a splay tree."""
  #-------------------------------- splay operation --------------------------------
  def _splay(self, p):
    while p != self.root():
      parent = self.parent(p)
      grand = self.parent(parent)
      if grand is None:
        # zig case
        self._rotate(p)
      elif (parent == self.left(grand)) == (p == self.left(parent)):
        # zig-zig case
        self._rotate(parent)                          # move PARENT up
        self._rotate(p)                               # then move p up
      else:
        # zig-zag case
        self._rotate(p)                               # move p up
        self._rotate(p)                               # move p up again

  #--------------------------- override balancing hooks ---------------------------
  def _rebalance_insert(self, p):
    self._splay(p)

  def _rebalance_delete(self, p):
    if p is not None:
      self._splay(p)

  def _rebalance_access(self, p):
    self._splay(p)
```

(2,4) Trees



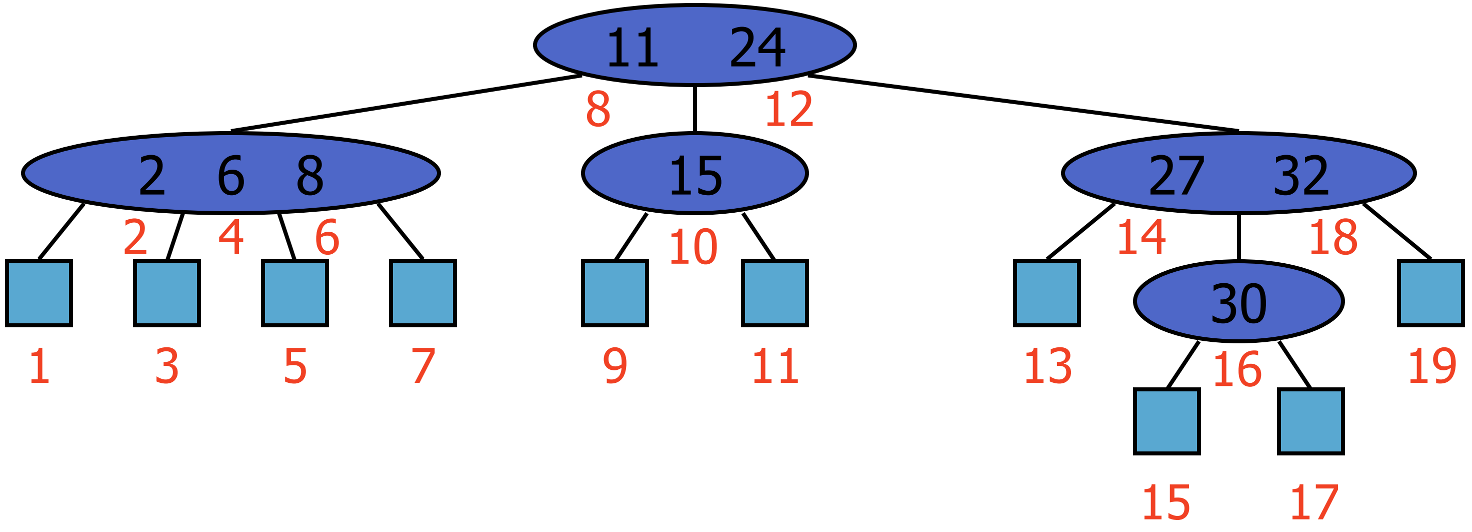
Multi-Way Search Tree

- A multi-way search tree is an ordered tree such that
  - Each internal node has at least two children
    - When d is the number of children, the node stores $d-1$ key-element items (k_i, o_i), where…
  - 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



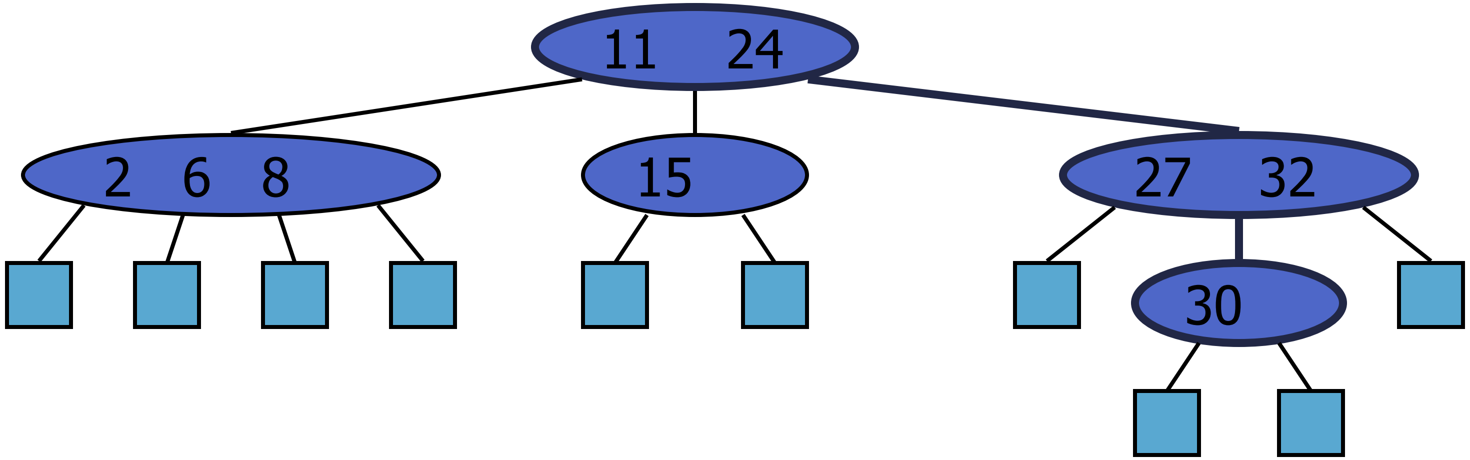
Multi-Way Inorder Traversal

- 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



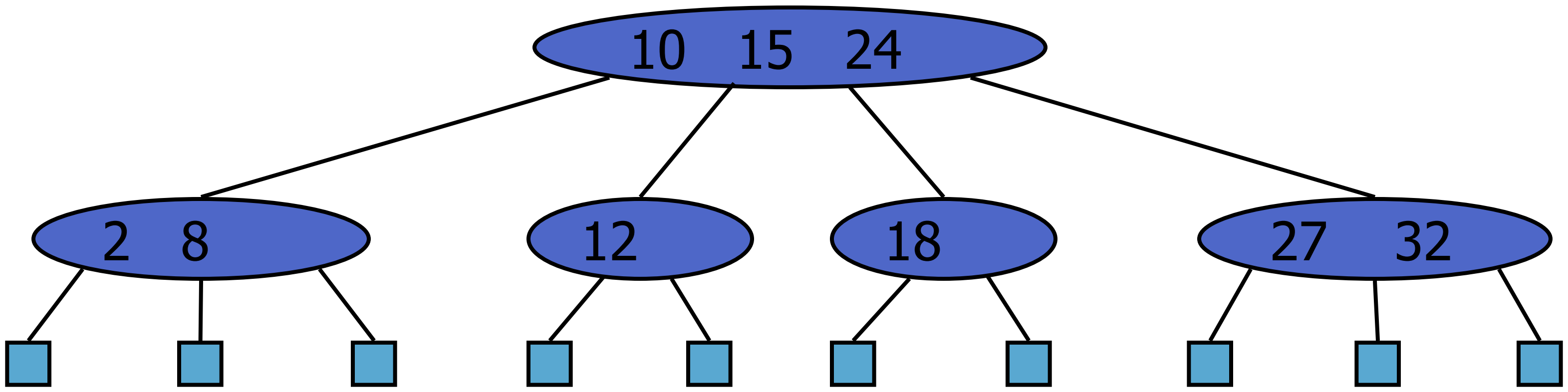
Multi-Way Searching

- 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

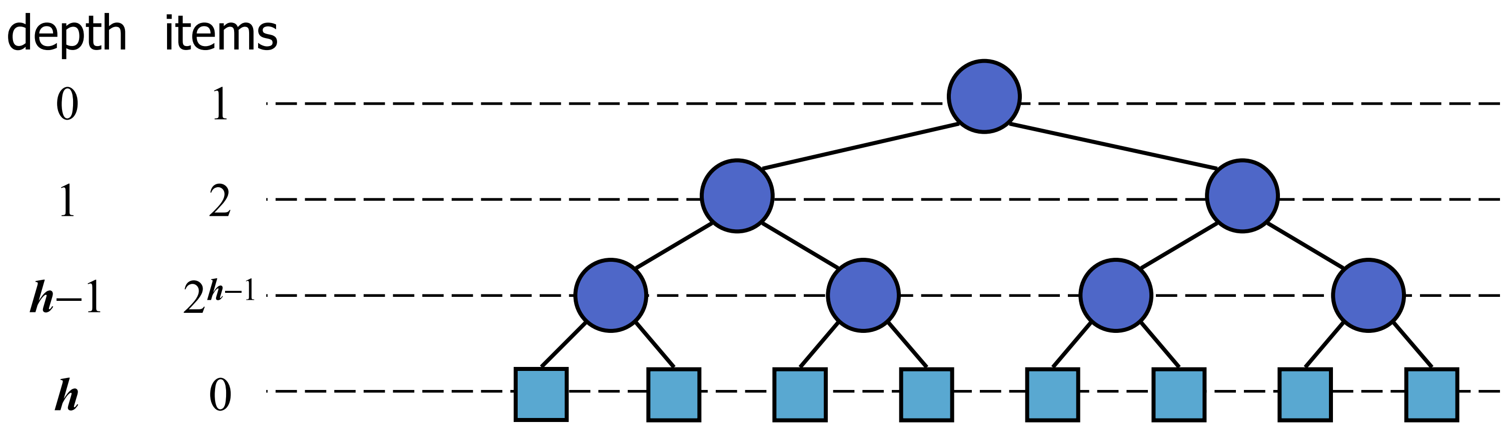
- A (2,4) tree (also called 2-4 tree or 2-3-4 tree) is a multi-way search with the following properties
  - Node-Size Property: every internal node has at most four children
  - Depth Property: all the external nodes have the same depth
- Depending on the number of children, an internal node of a (2,4)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



Height of a (2,4) Tree

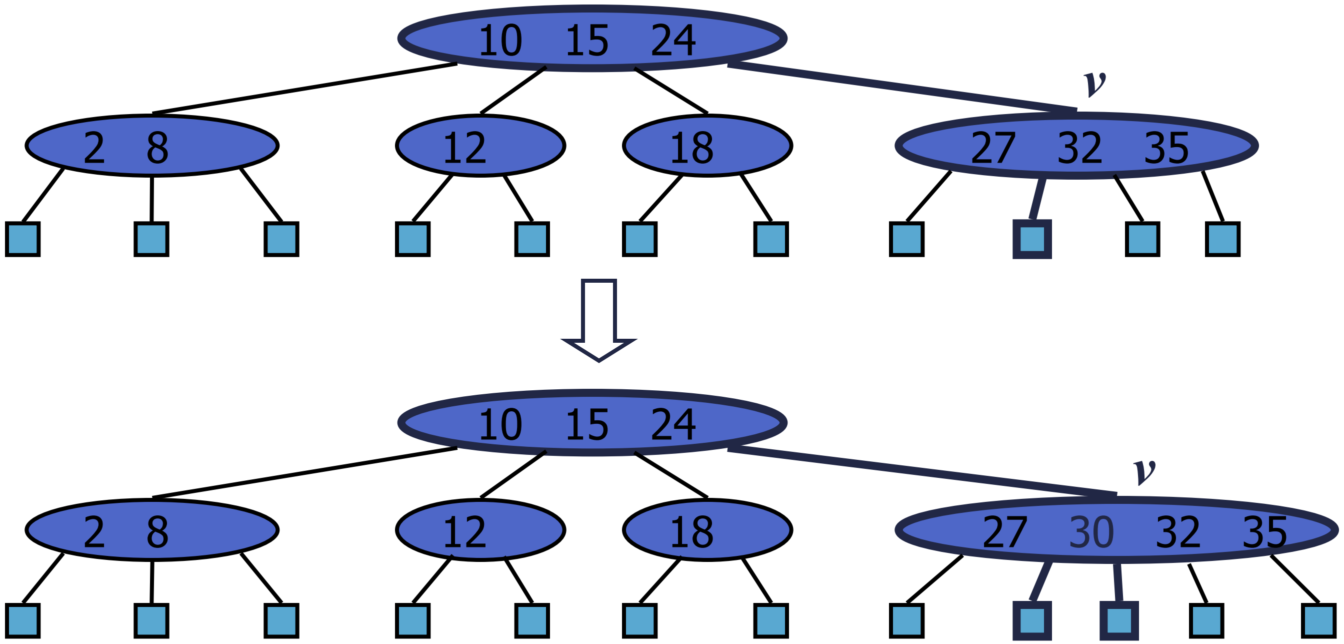
- Theorem: A (2,4) tree storing n items has height $O(\log n)$

  Proof:
  - Let h be the height of a (2,4) tree with n items
  - Since there are at least 2^i items at depth $i = 0, \dots, h-1$ and no items at depth h, we have
    $$n \geq 1 + 2 + 4 + \dots + 2^{h-1} = 2^h - 1$$
  - Thus, $h \leq \log(n+1)$
- Searching in a (2,4) tree with n items takes $O(\log n)$ time



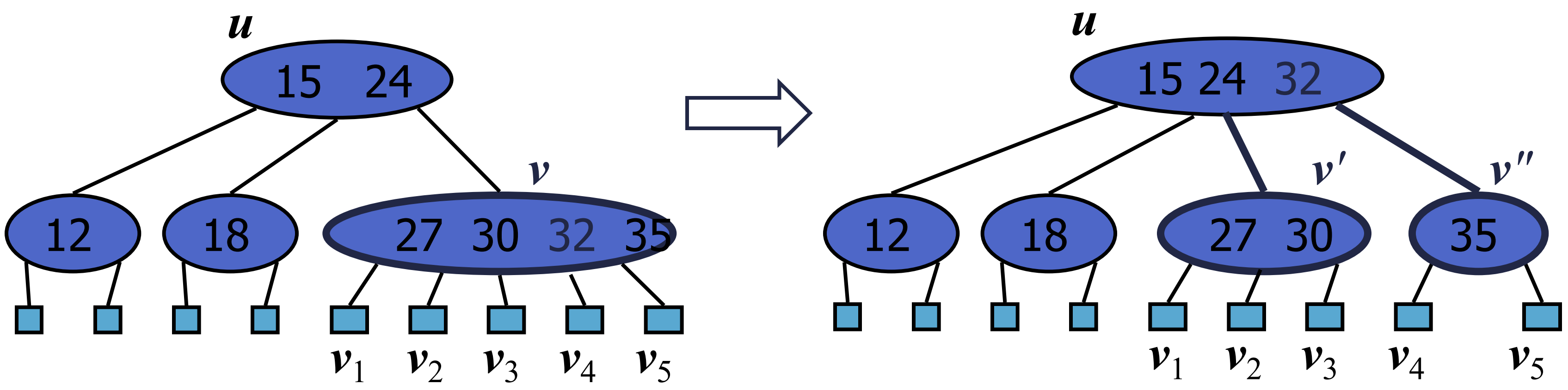
Insertion

- 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



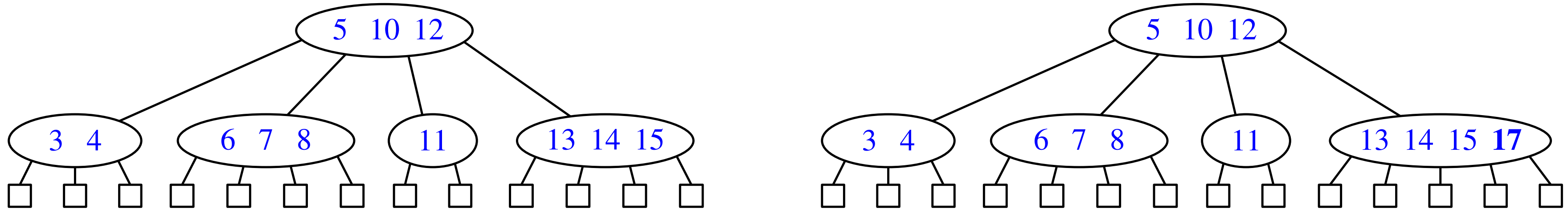
Overflow and Split

- We handle an overflow at a 5-node $\boldsymbol{v}$ with a split operation:
  - let $\boldsymbol{v}_1 \ldots \boldsymbol{v}_5$ be the children of $\boldsymbol{v}$ and $\boldsymbol{k}_1 \ldots \boldsymbol{k}_4$ be the keys of $\boldsymbol{v}$
  - node $\boldsymbol{v}$ is replaced nodes $\boldsymbol{v}'$ and $\boldsymbol{v}''$
    - $\boldsymbol{v}'$ is a 3-node with keys $\boldsymbol{k}_1\ \boldsymbol{k}_2$ and children $\boldsymbol{v}_1\ \boldsymbol{v}_2\ \boldsymbol{v}_3$
    - $\boldsymbol{v}''$ is a 2-node with key $\boldsymbol{k}_4$ and children $\boldsymbol{v}_4\ \boldsymbol{v}_5$
  - key $\boldsymbol{k}_3$ is inserted into the parent $\boldsymbol{u}$ of $\boldsymbol{v}$ (a new root may be created)
- The overflow may propagate to the parent node $\boldsymbol{u}$

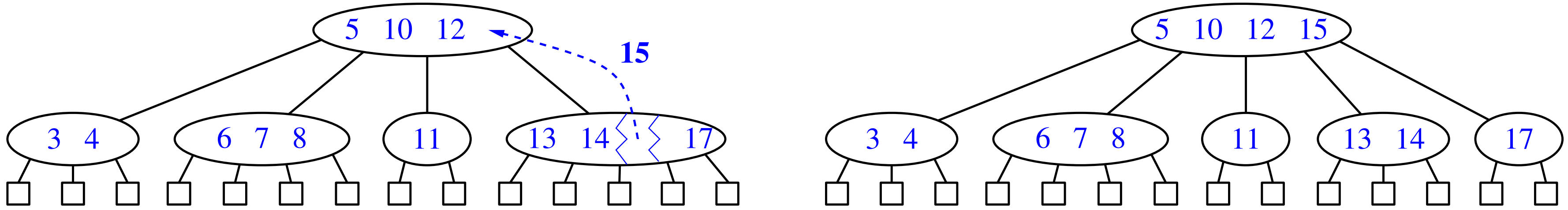


Insetion Example

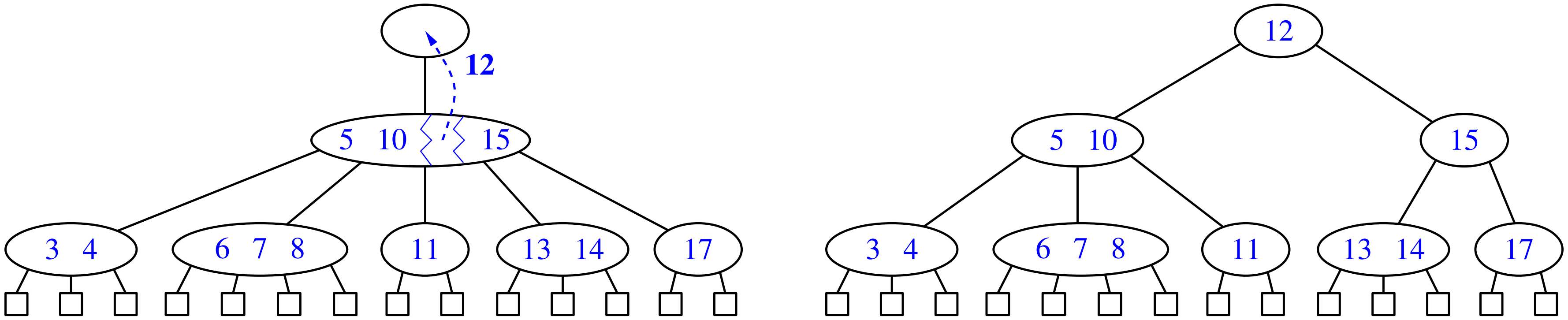
Insert
- overflown



First split
- overflown

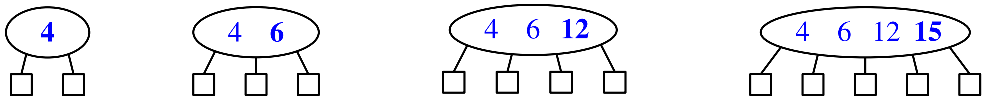


Second split

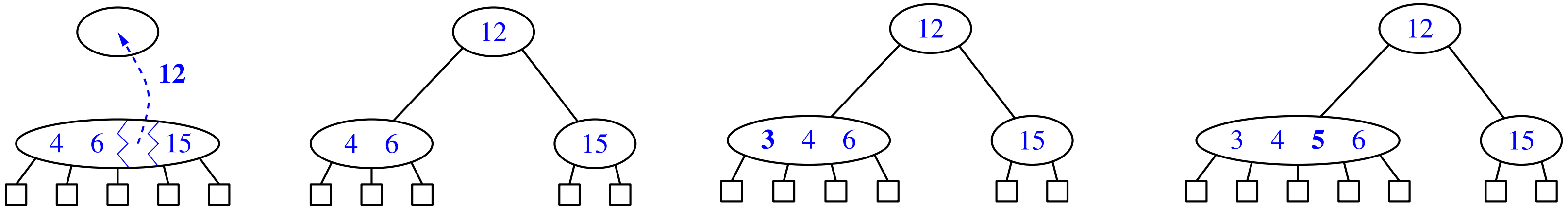


Insertion – more examples

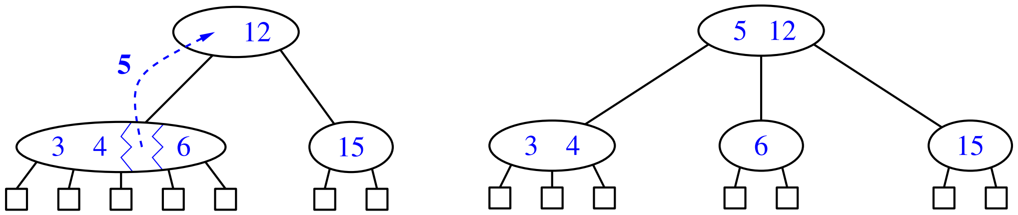
4, 6, 12, 15
overflow



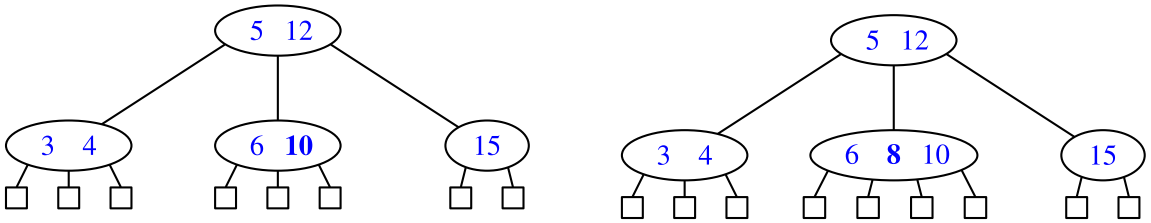
split
3, 5
overflow



split



10, 8



Analysis of Insertion

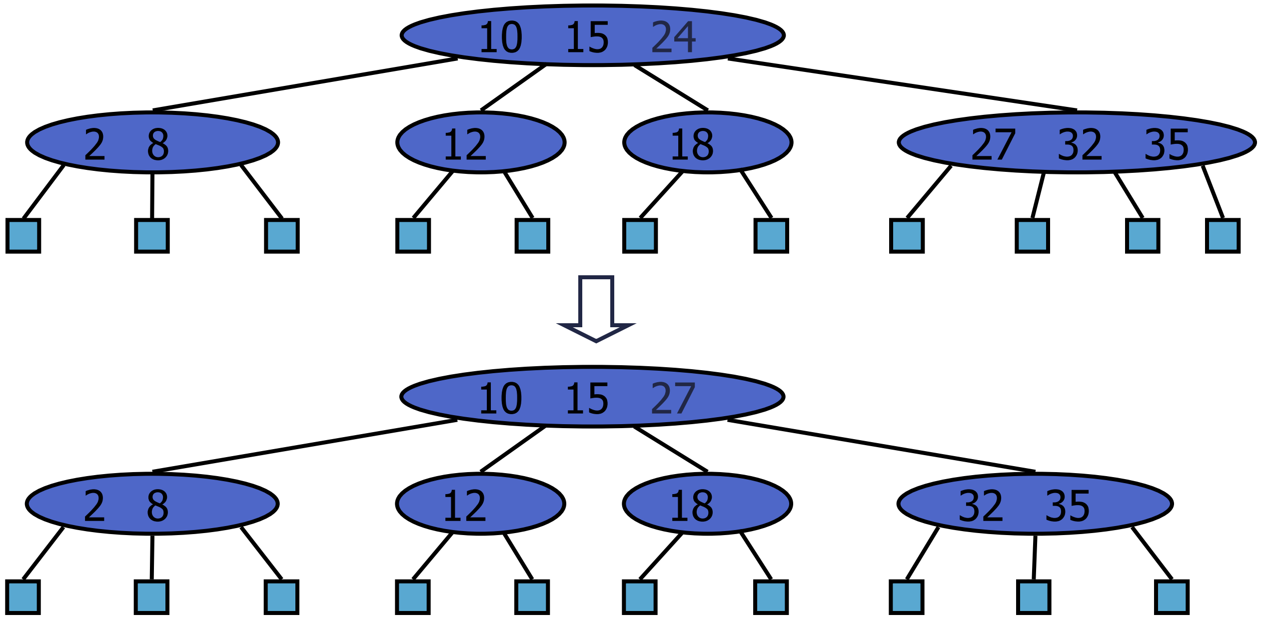
Algorithm ***put***(k, o)

1. We search for key $\boldsymbol{k}$ to locate the insertion node $\boldsymbol{v}$
2. We add the new entry $(\boldsymbol{k}, \boldsymbol{o})$ at node $\boldsymbol{v}$
3. **while** ***overflow***$(\boldsymbol{v})$
   if ***isRoot***$(\boldsymbol{v})$
   create a new empty root above $\boldsymbol{v}$
   $\boldsymbol{v} \leftarrow$ ***split***$(\boldsymbol{v})$

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

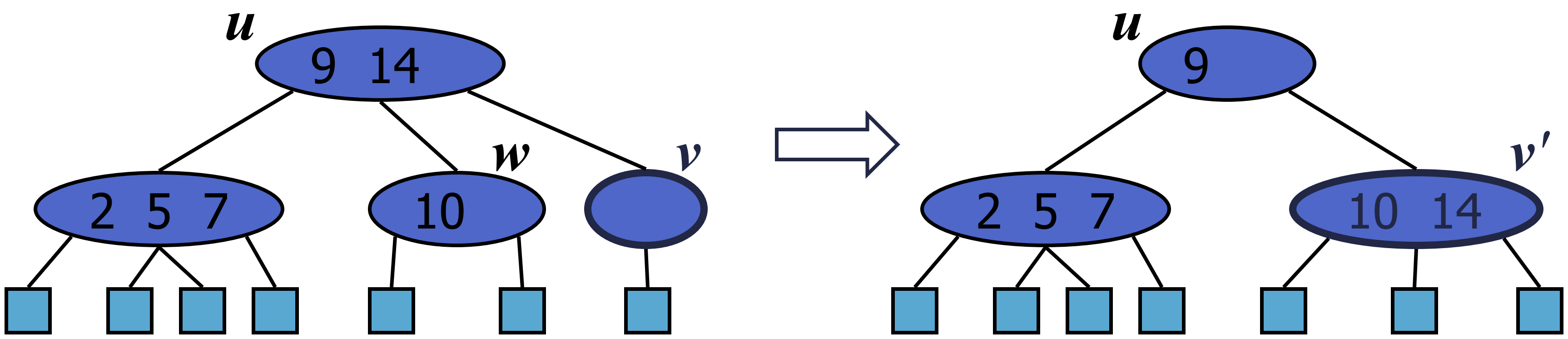
Deletion

- 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)



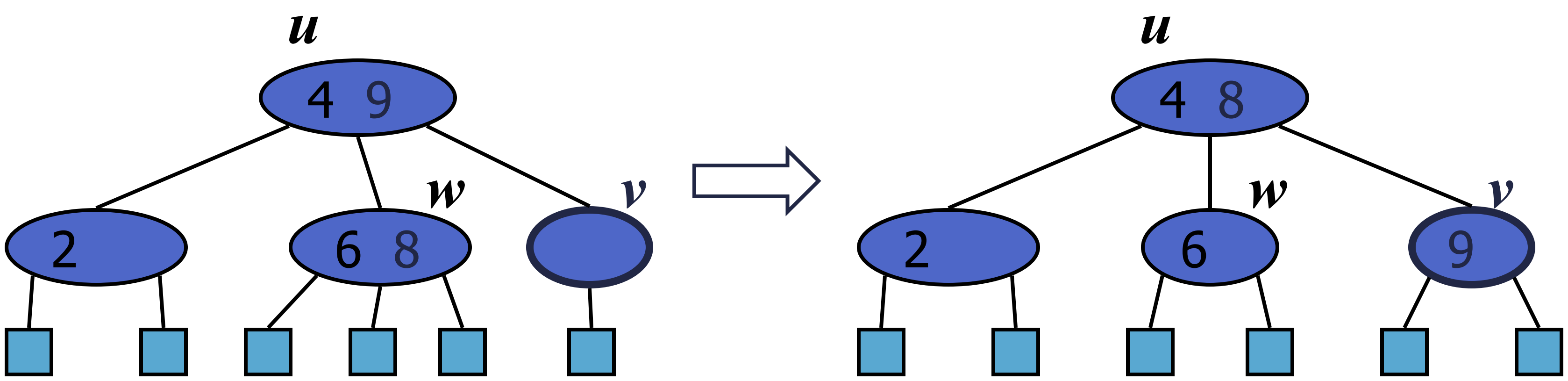
Underflow and Fusion

- 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



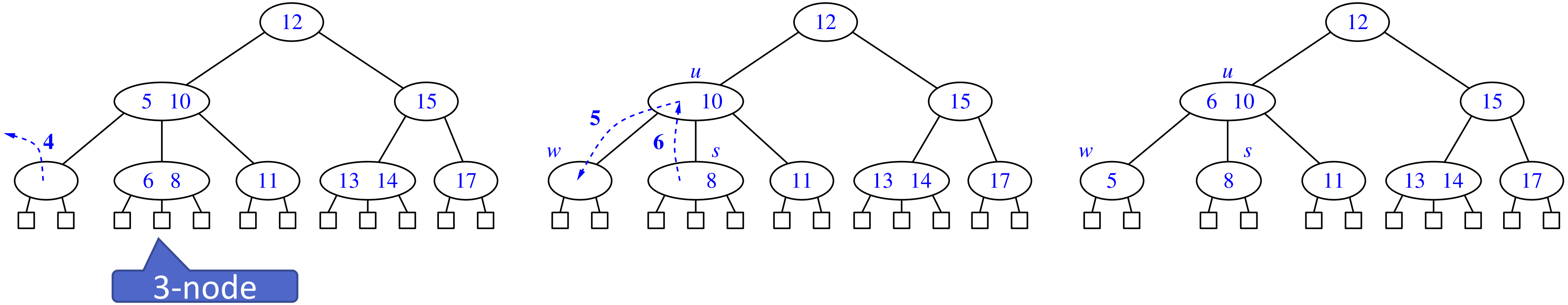
Underflow and Transfer

- 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

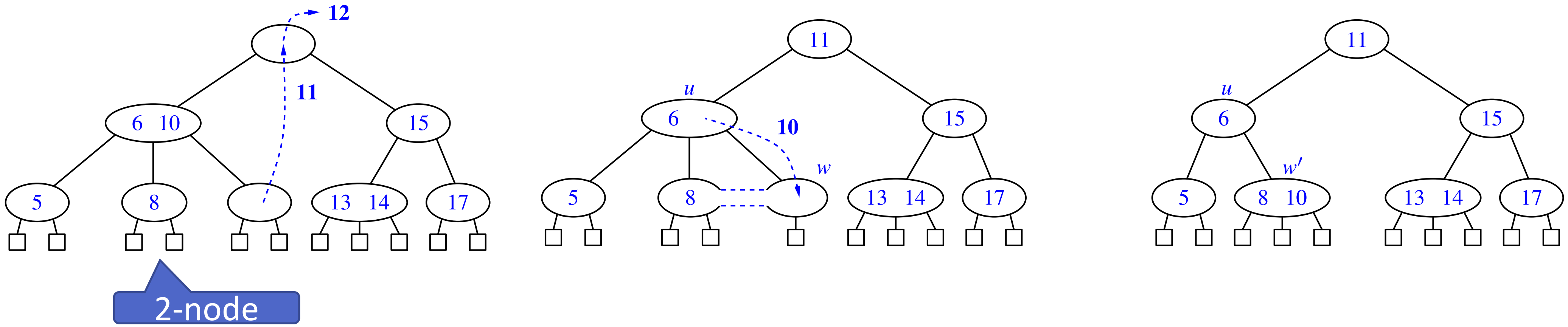


Deletion - Example

Remove 4
Underflow
Transfer

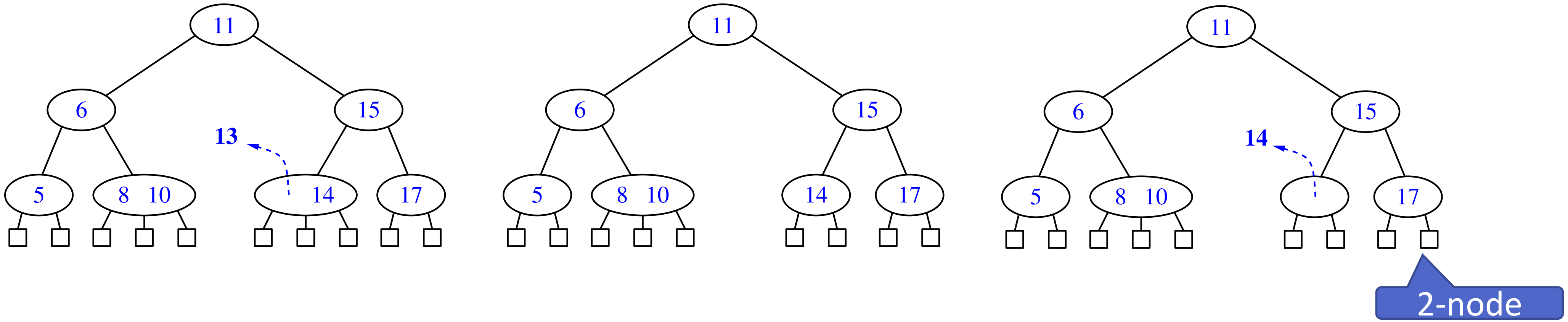


Remove 12
Underflow
Fusion

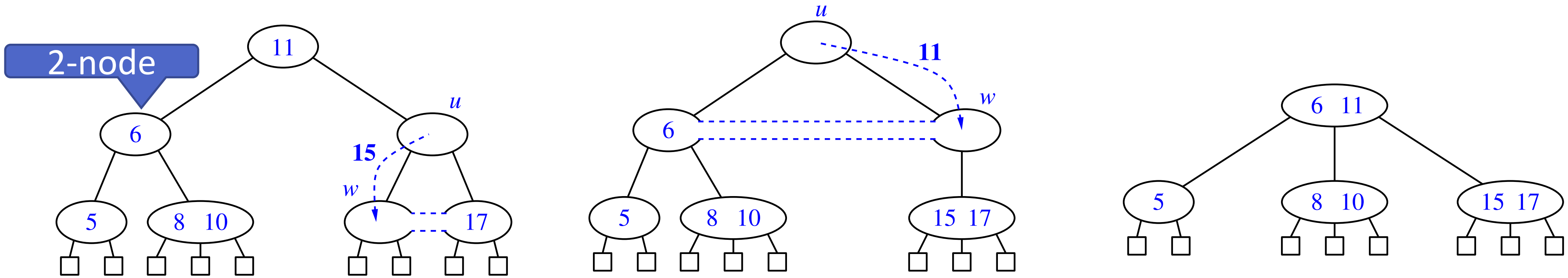


Deletion – Example (cont.)

Remove 13
Remove 14
Underflow



Transfer
Underflow
Merge



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

Comparison of Map Implementations

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