AIE2001: 자료구조

#9 – Search Tree

인공지능공학과
김 병 형

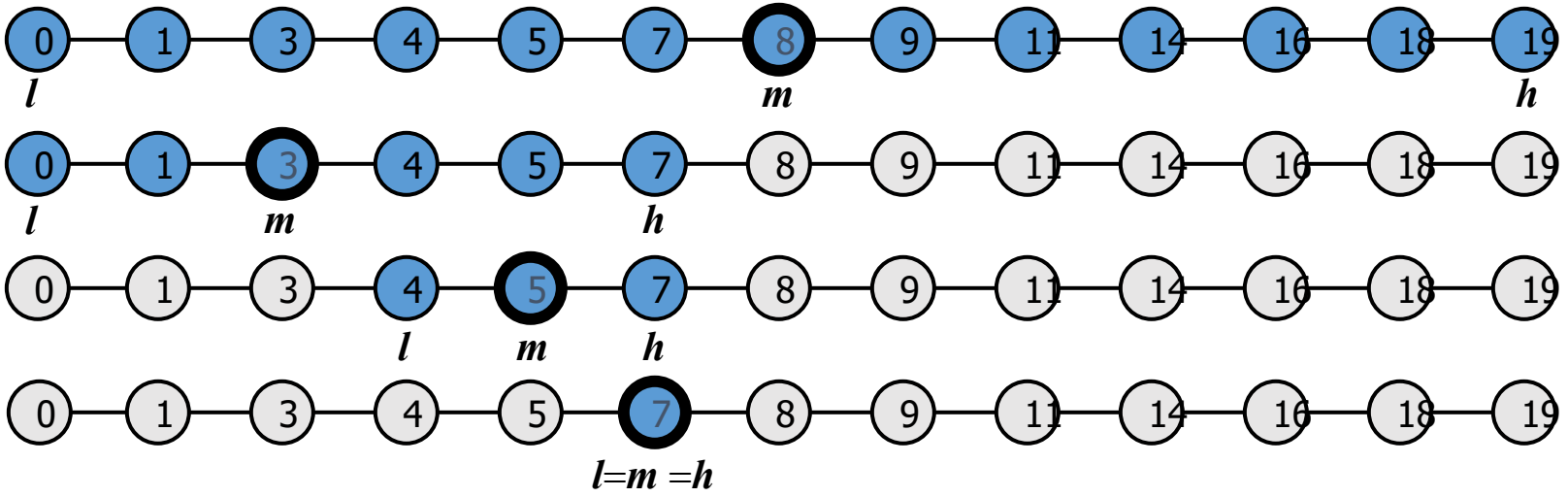
인하대학교
INHA UNIVERSITY

Ordered Maps

- Keys are assumed to come from a total order.
- Items are stored in order by their keys
- This allows us to support nearest neighbor queries:
  - Item with largest key less than or equal to k
  - Item with smallest key greater than or equal to k

Binary Search

- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - similar to the high-low children's game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)



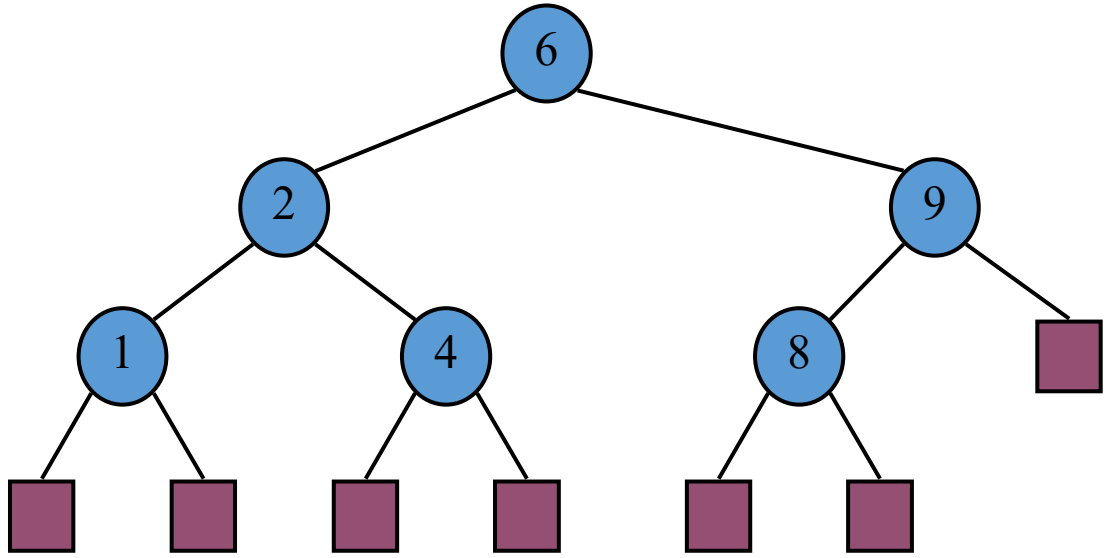
Search Tables

- A search table is an ordered map implemented by means of a sorted sequence
  - We store the items in an array-based sequence, sorted by key
  - We use an external comparator for the keys
- Performance:
  - Searches take $O(\log n)$ time, using binary search
  - Inserting a new item takes $O(n)$ time, since in the worst case we have to shift n items to make room for the new item
  - Removing an item takes $O(n)$ time, since in the worst case we have to shift n items to compact the items after the removal
- The lookup table is effective only for ordered maps of small size or for maps 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 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

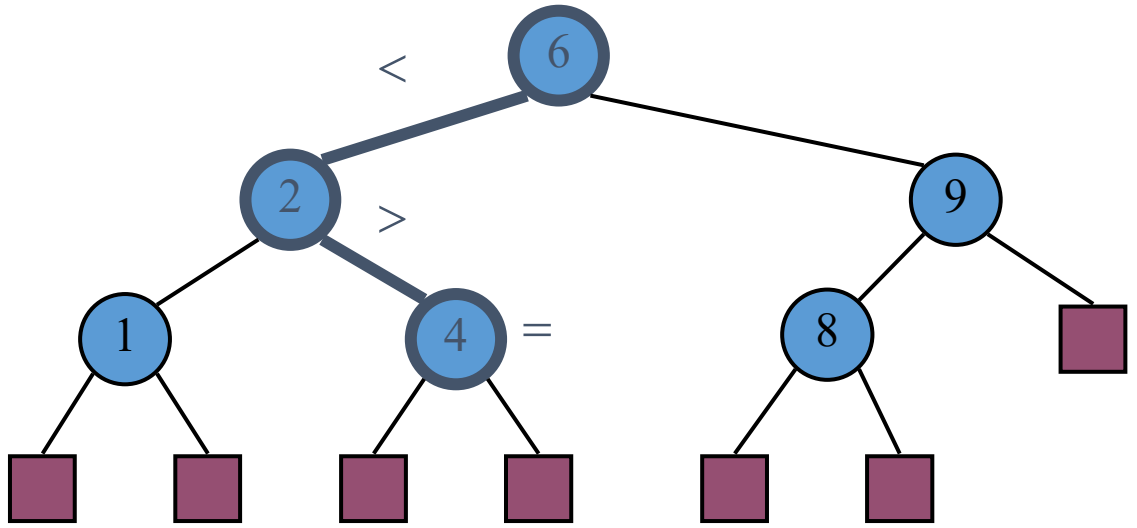
- An inorder traversal of a binary search trees visits the keys in increasing order



Search

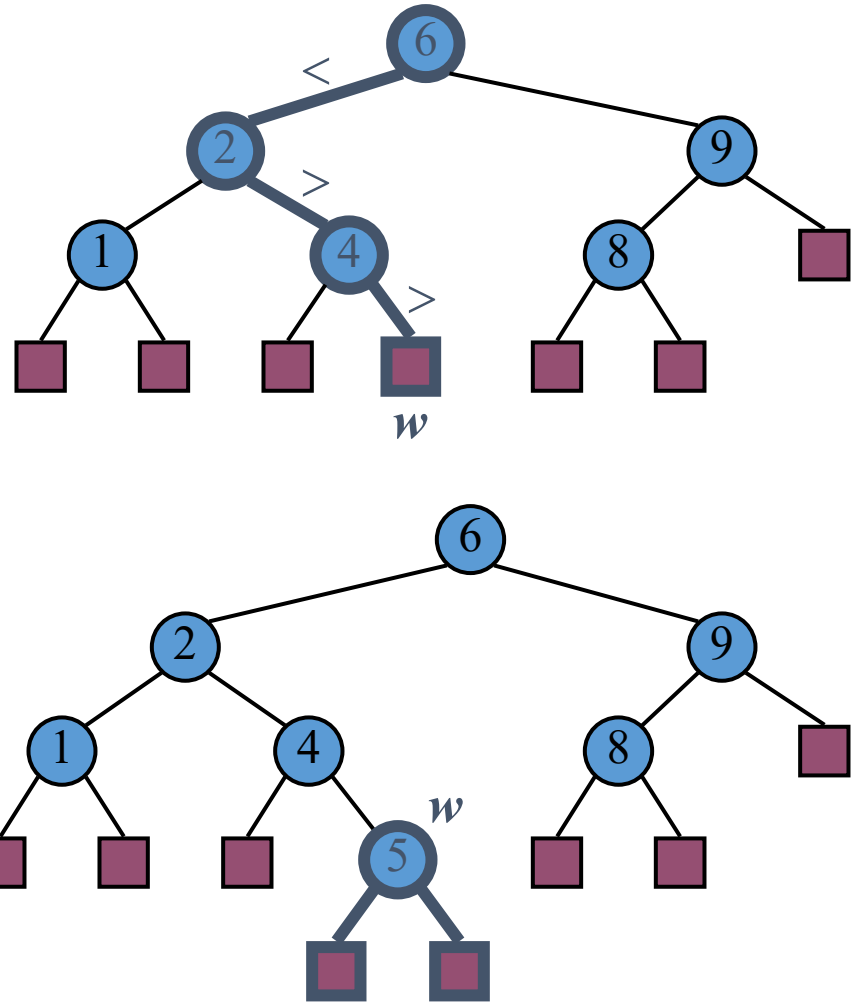
- To search for a key ***k***, we trace a downward path starting at the root
- The next node visited depends on the comparison of ***k*** with the key of the current node
- If we reach a leaf, the key is not found
- Example: find(4):
  - Call TreeSearch(4,root)
- The algorithms for nearest neighbor queries are similar

```
Algorithm TreeSearch(T, p, k):
    if k == p.key() then
        return p                                        {successful search}
    else if k < p.key() and T.left(p) is not None then
        return TreeSearch(T, T.left(p), k)              {recur on left subtree}
    else if k > p.key() and T.right(p) is not None then
        return TreeSearch(T, T.right(p), k)             {recur on right subtree}
    return p                                            {unsuccessful search}
```



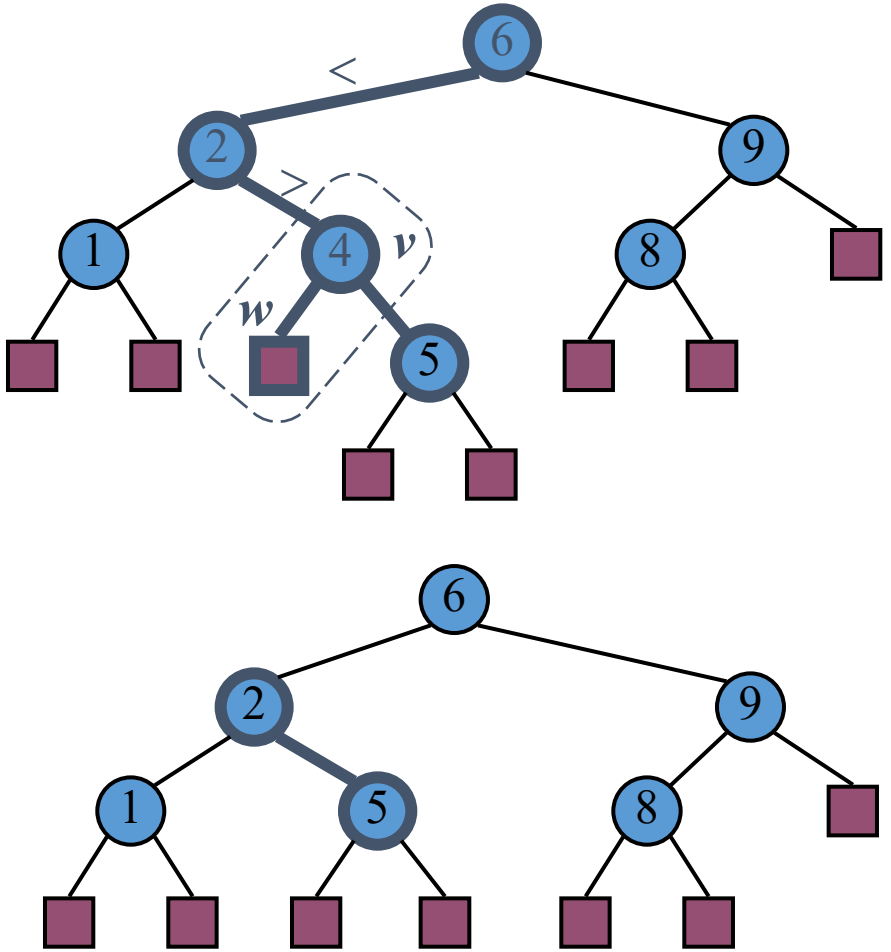
Insertion

- To perform operation put(k, o), we search for key k (using TreeSearch)
- Assume k is not already in the tree, and let w be the (None) leaf reached by the search
- We insert k at node w and expand w into an internal node
- Example: insert 5



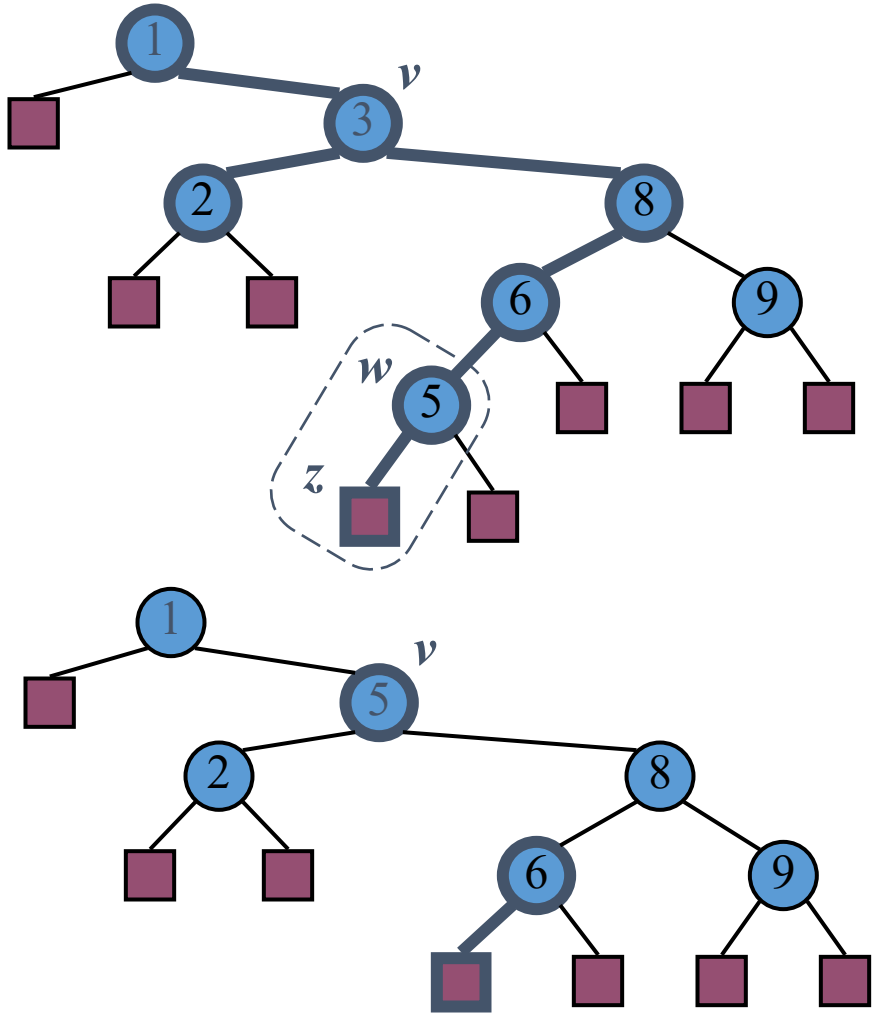
Deletion

- To perform operation remove(***k***), we search for key ***k***
- Assume key ***k*** is in the tree, and let ***v*** be the node storing ***k***
- If node ***v*** has a (None) leaf child ***w***, we remove ***v*** and ***w*** from the tree with operation removeExternal(***w***), which removes ***w*** and its parent
- Example: remove 4



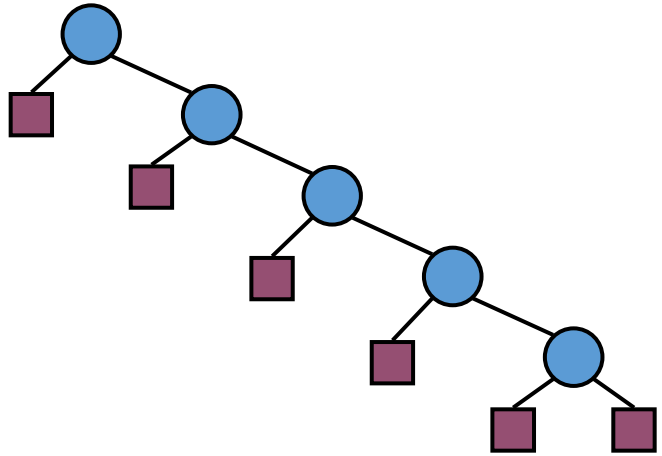
Deletion (cont.)

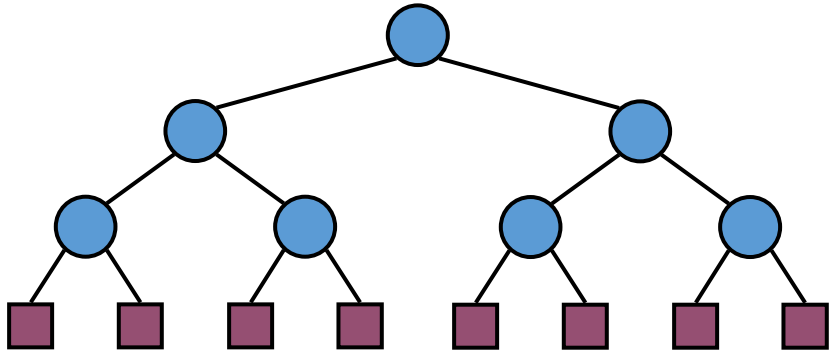
- We consider the case where the key ***k*** to be removed is stored at a node ***v*** whose children are both internal
  - we find the internal node ***w*** that follows ***v*** in an inorder traversal
  - we copy ***key***(***w***) into node ***v***
  - we remove node ***w*** and its left child ***z***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



Performance

- Consider an ordered map with ***n*** items implemented by means of a binary search tree of height ***h***
  - the space used is ***O***(***n***)
  - Search and update methods take ***O***(***h***) time
- The height ***h*** is ***O***(***n***) in the worst case and ***O***(log ***n***) in the best case

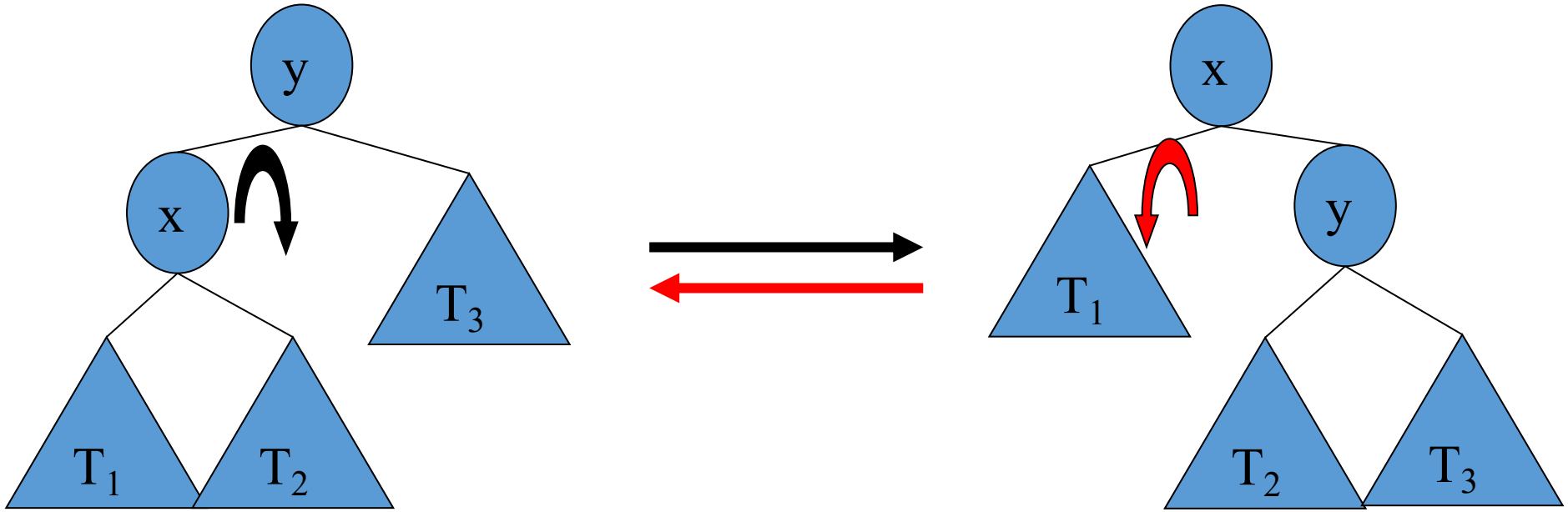




Rotations

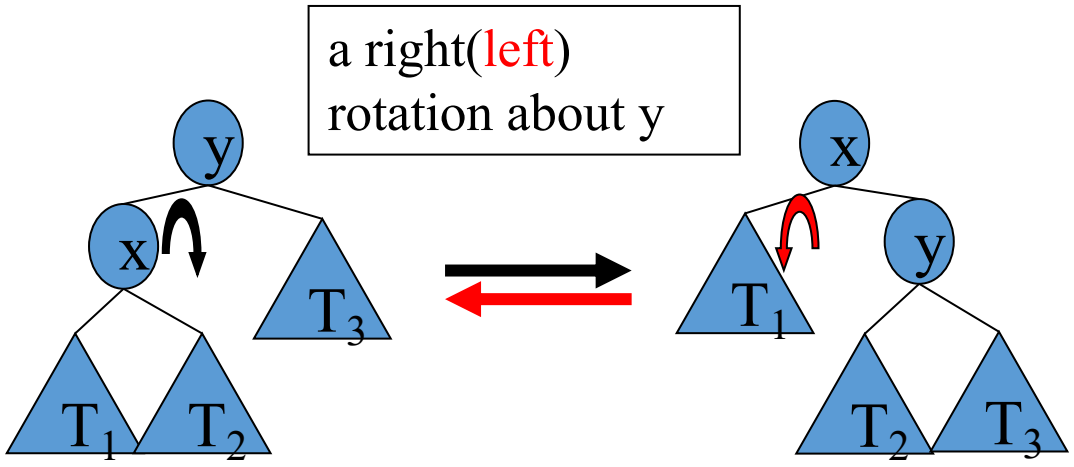
- right (left) rotation
  - makes the left child *x* of a node *y* into *y*'s parent; *y* becomes the right child of *x*

a right(left) rotation about y



Rotations

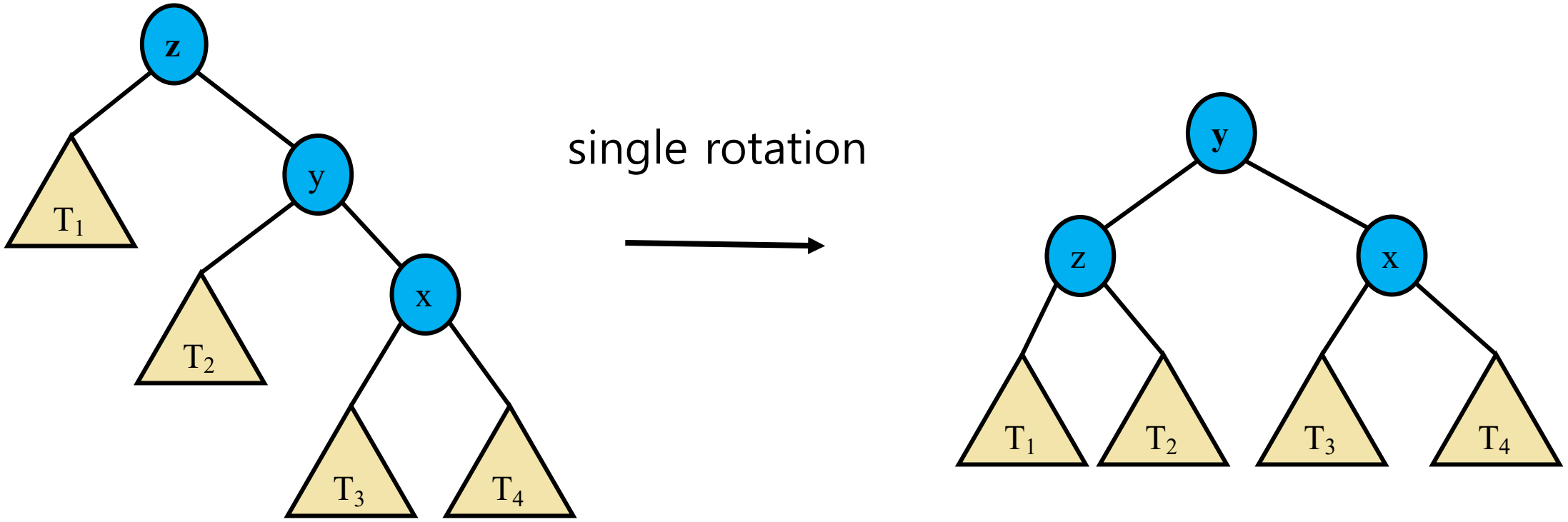
- right (left) rotation
  - makes the left child *x* of a node *y* into *y*'s parent; *y* becomes the right child of *x*



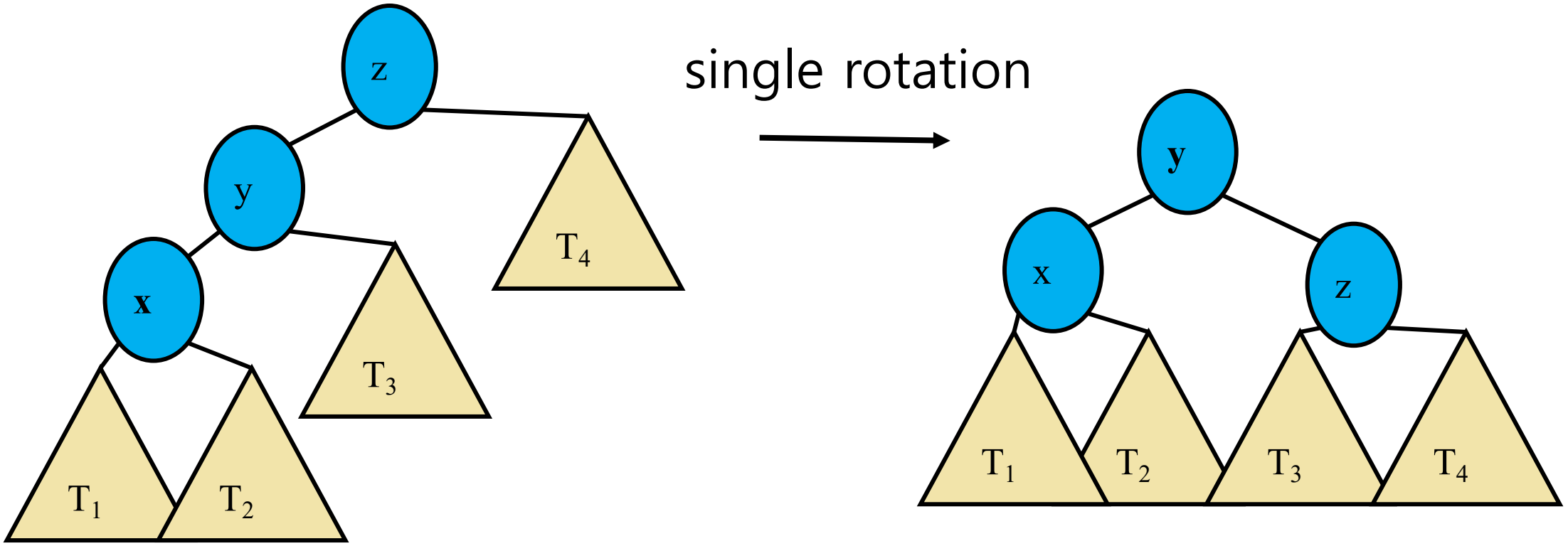

```
def _relink(self, parent, child, make_left_child):
  """Relink parent node with child node (we allow child to be None)."""
  if make_left_child:                          # make it a left child
    parent._left = child
  else:                                        # make it a right child
    parent._right = child
  if child is not None:                        # make child point to parent
    child._parent = parent
```

```
def _rotate(self, p):
  """Rotate Position p above its parent."""
  x = p._node
  y = x._parent                                # we assume this exists
  z = y._parent                                # grandparent (possibly None)
  if z is None:
    self._root = x                             # x becomes root
    x._parent = None
  else:
    self._relink(z, x, y == z._left)           # x becomes a direct child of z
```

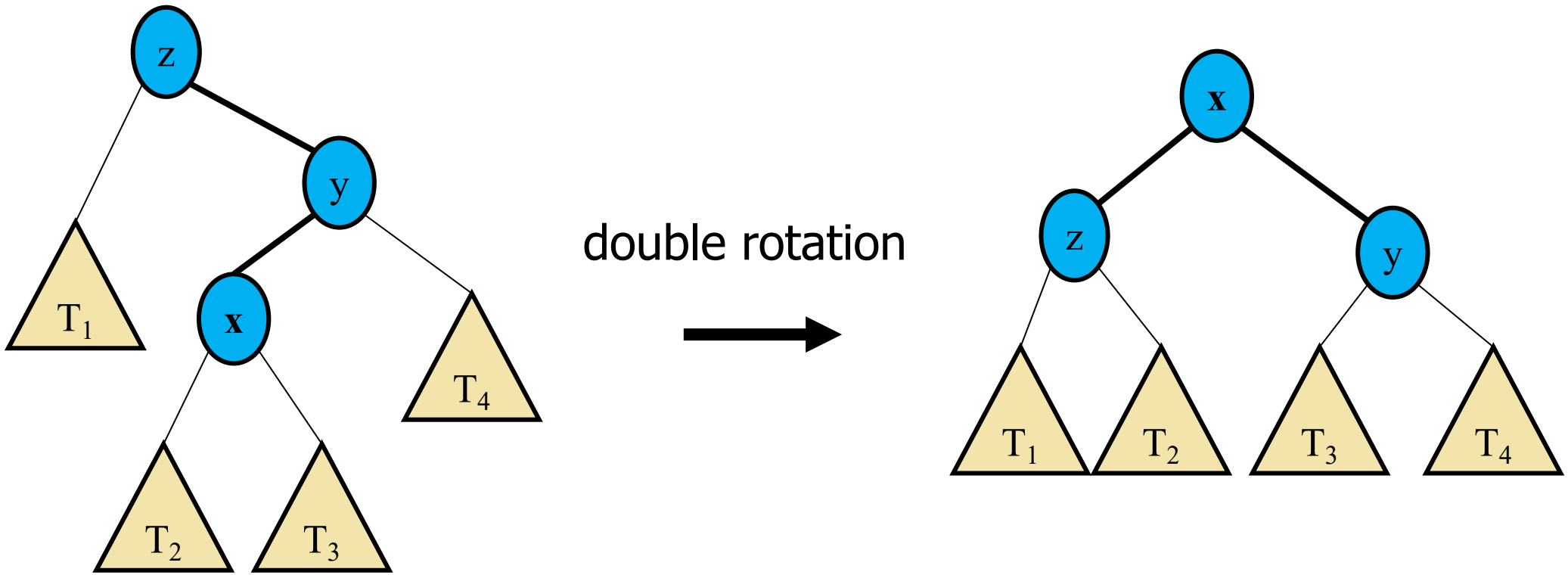
Trinode Restructuring



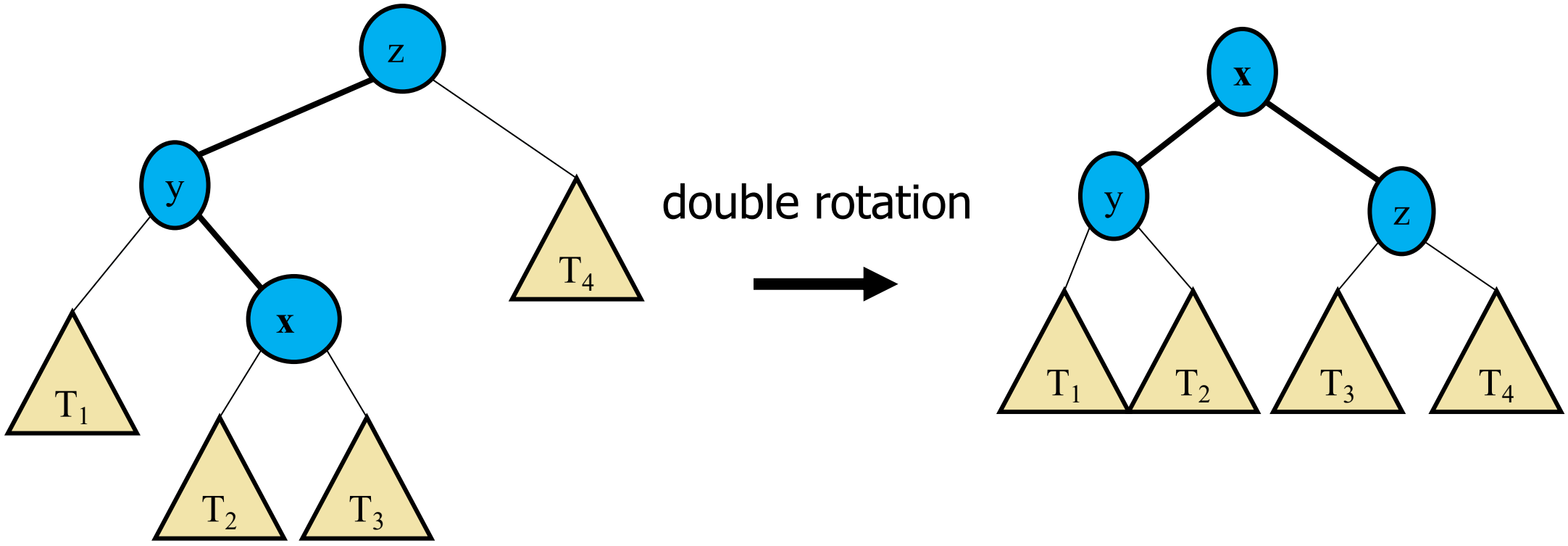
Trinode Restructuring



Trinode Restructuring



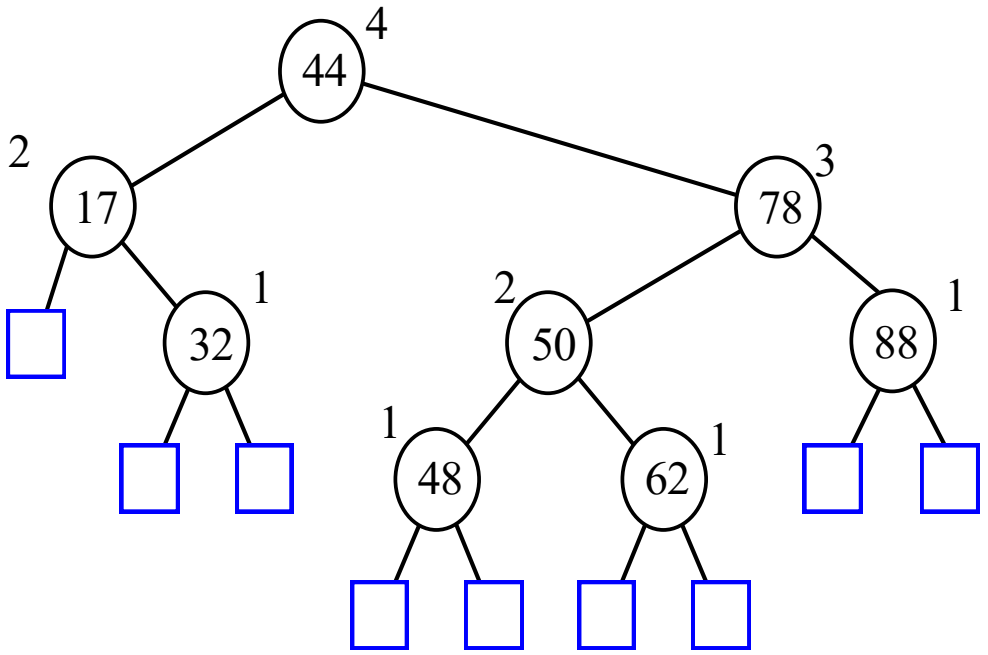
Trinode Restructuring



AVL Trees

AVL Tree Definition

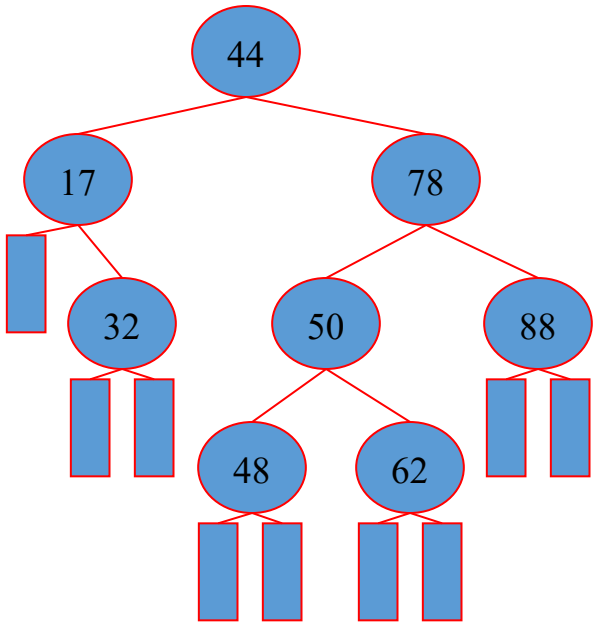
- AVL trees are balanced
- An AVL Tree is a binary search tree such that for every internal node v of T, the heights of the children of v can differ by at most 1



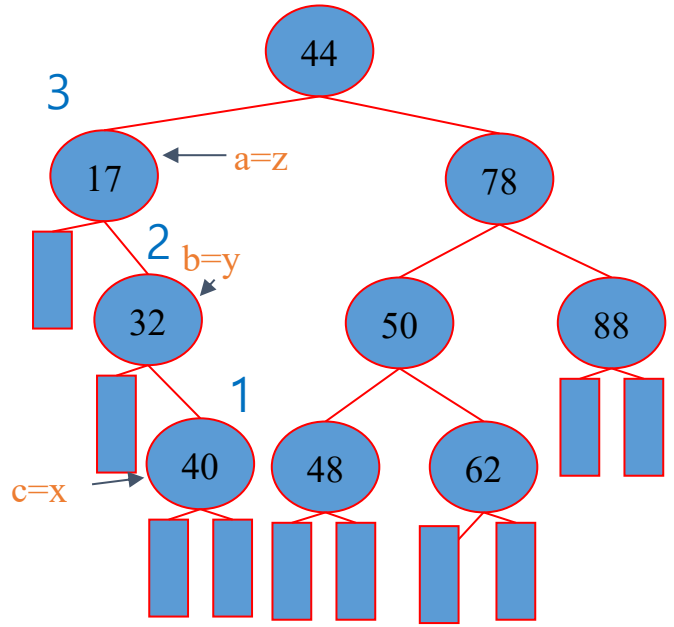
An example of an AVL tree where the heights are shown next to the nodes:

Insertion

- Insertion is as in a binary search tree
- Always done by expanding an external node.
- Example: insert 40

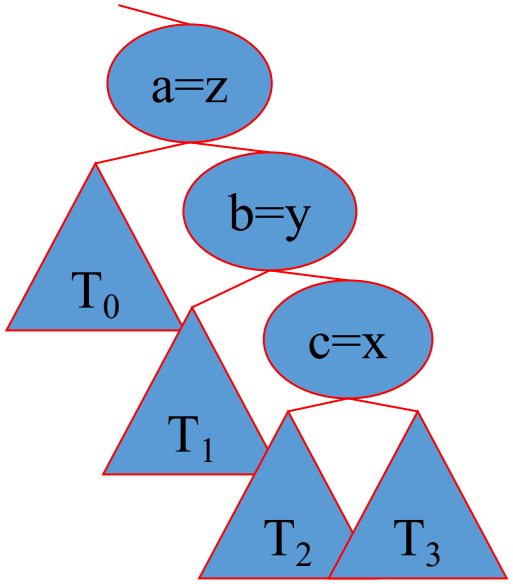


before insertion

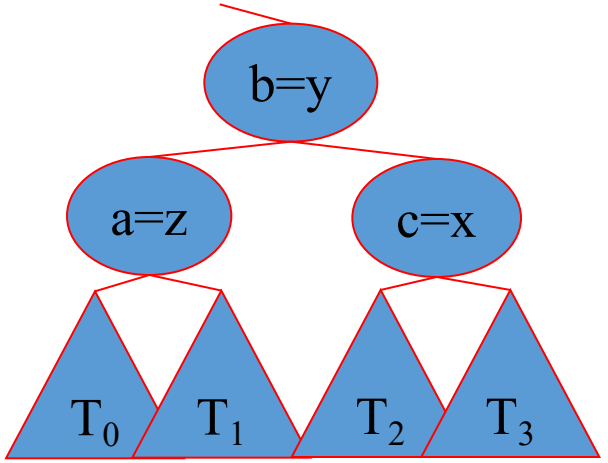


after insertion

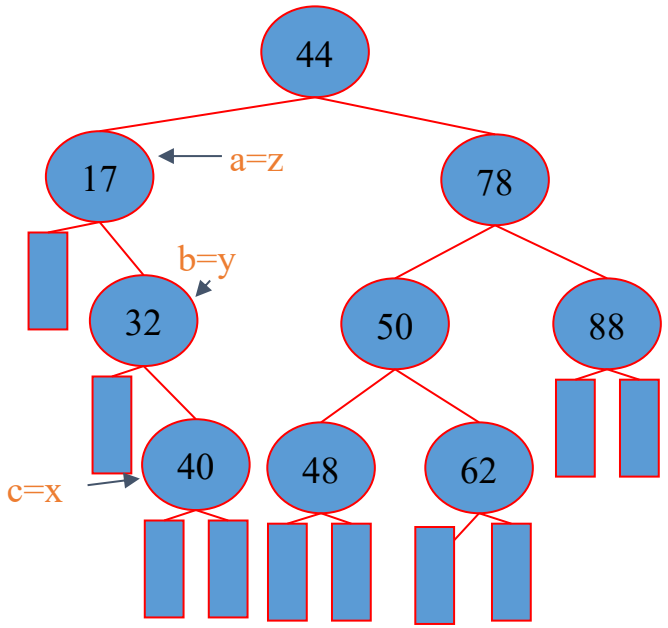
Trinode Restructuring



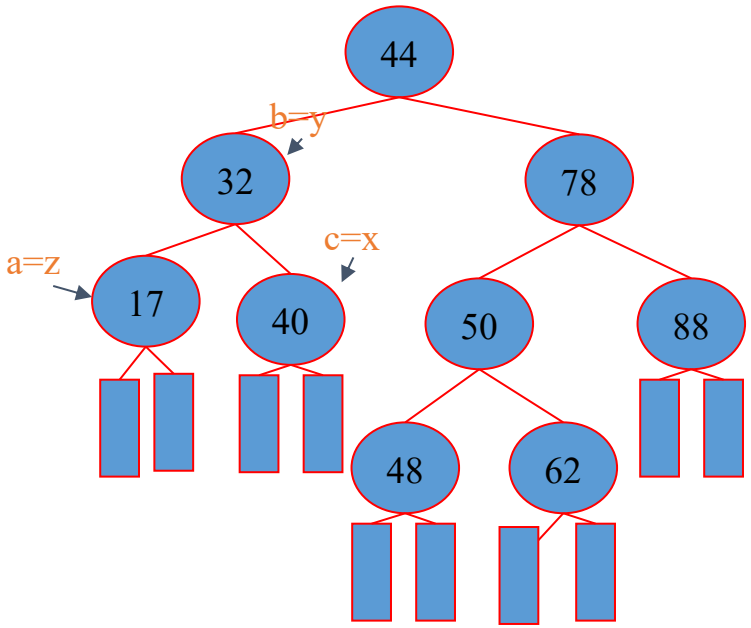
case 1: single rotation
(a left rotation about *a*)



Trinode Restructuring



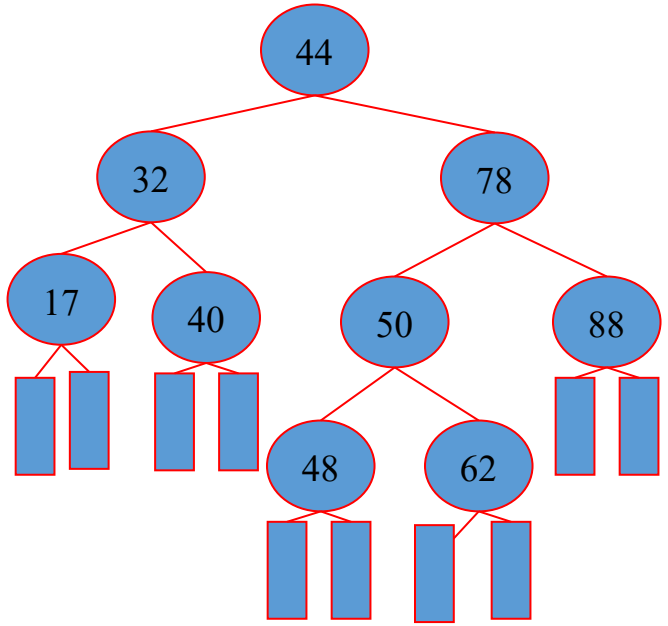
before rotation



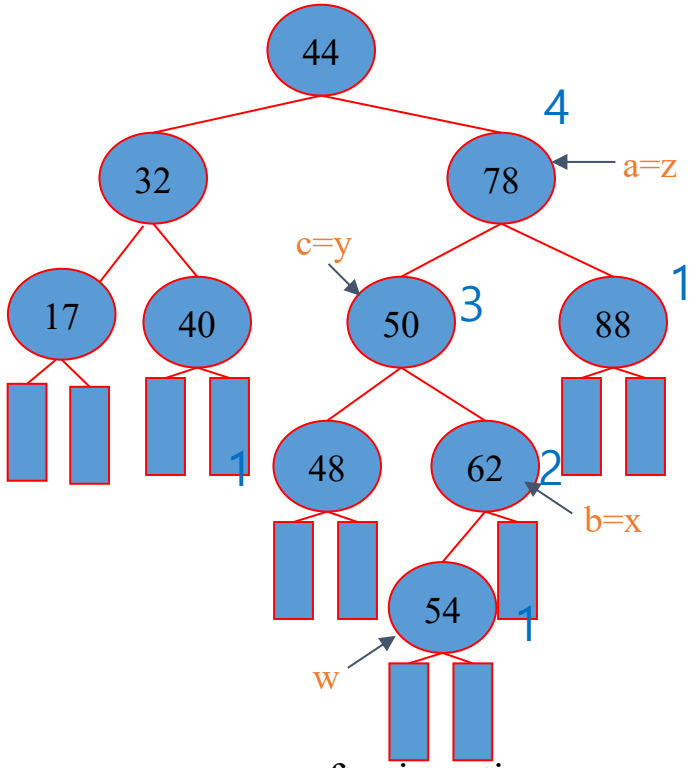
after rotation

Trinode Restructuring

- Example: insert 54

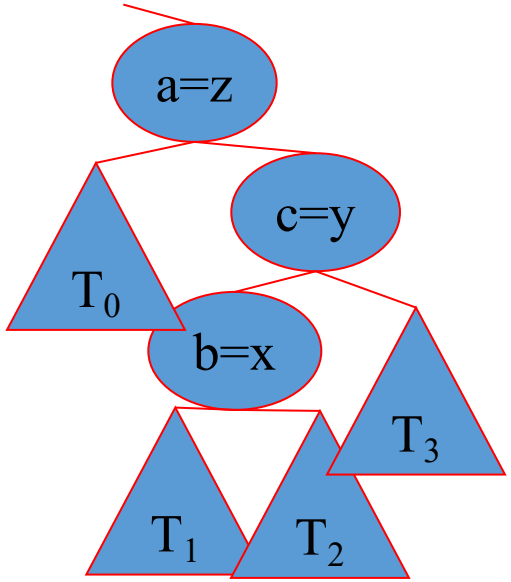


before insertion

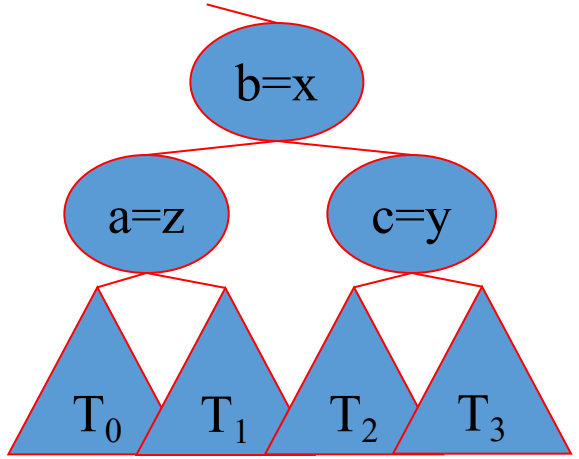


after insertion

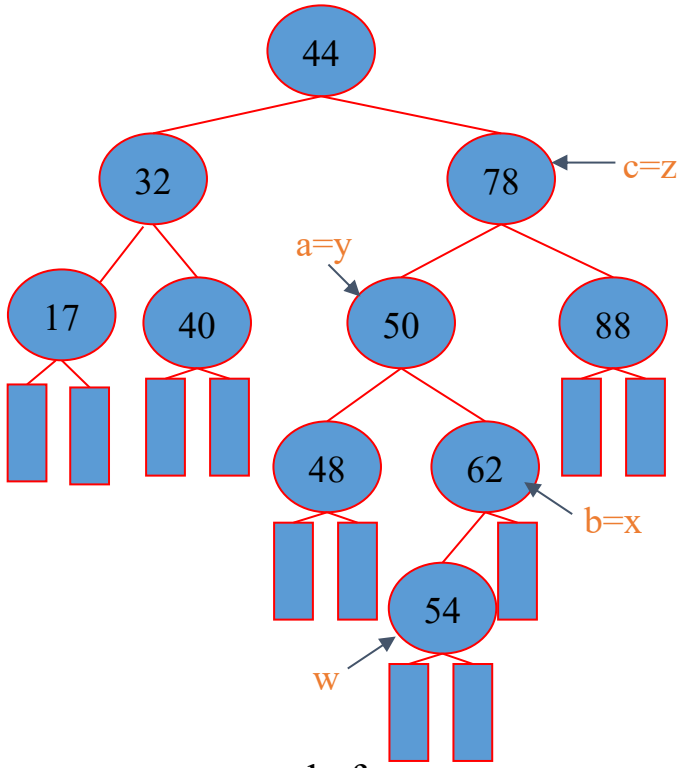
Trinode Restructuring



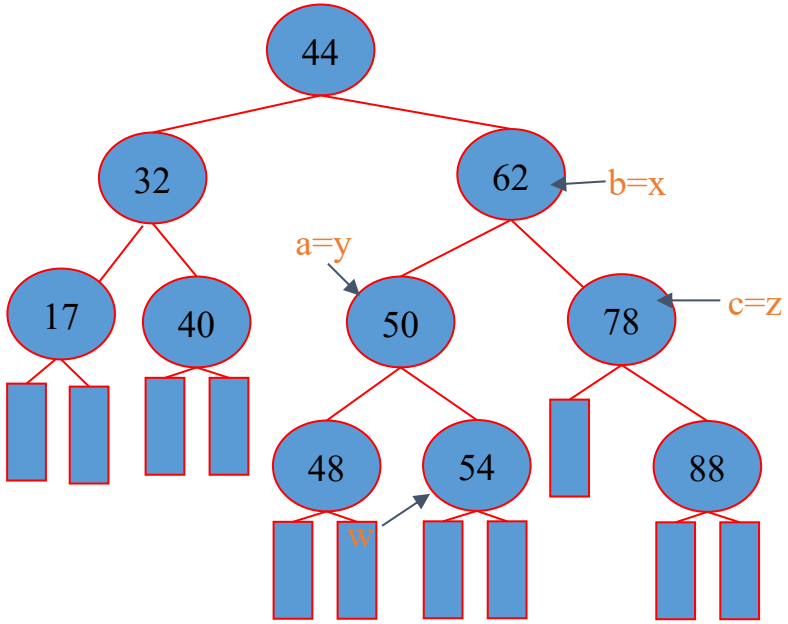
case 2: double rotation
(a right rotation about *c*,
then a left rotation about *a*)



Trinode Restructuring



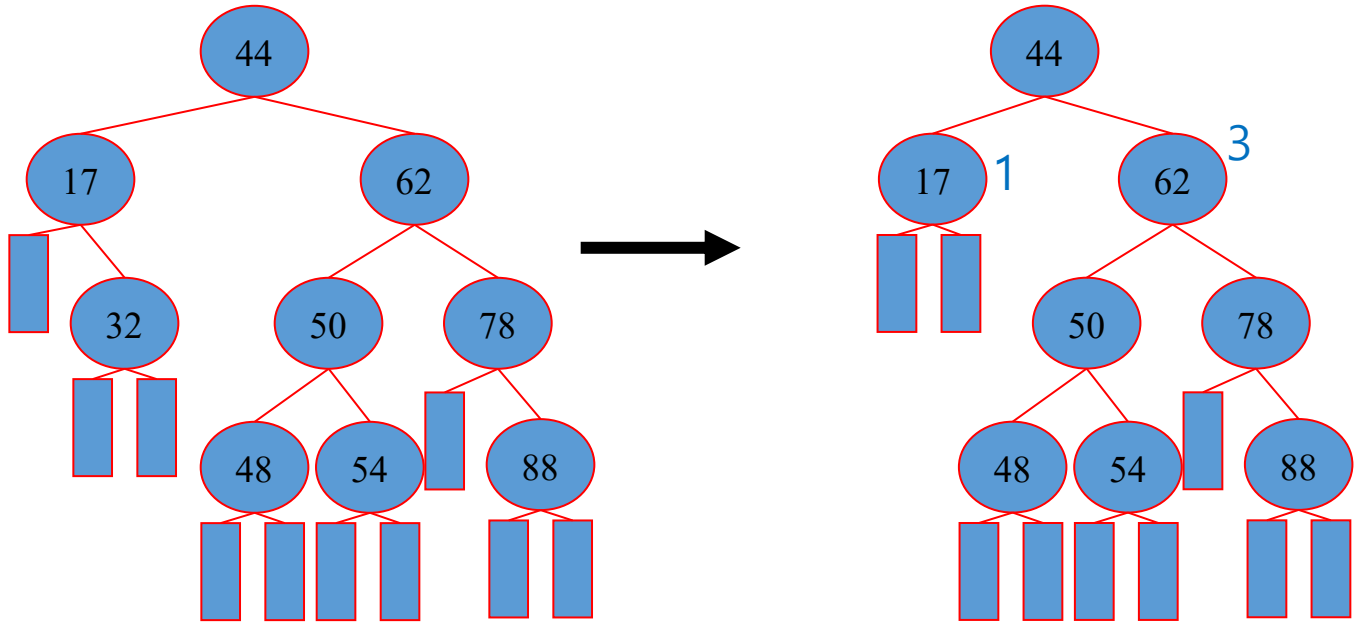
before



after

Removal

- 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: 32

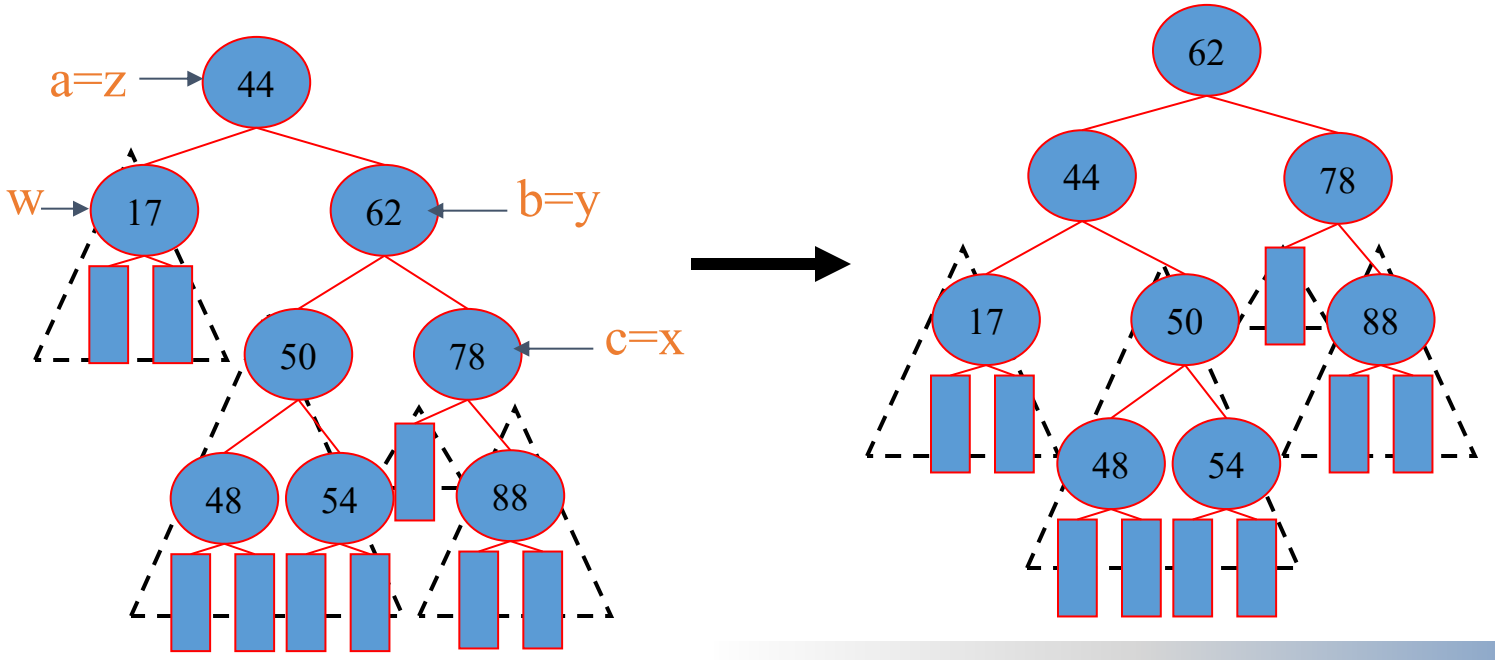


before deletion of 32

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
- 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



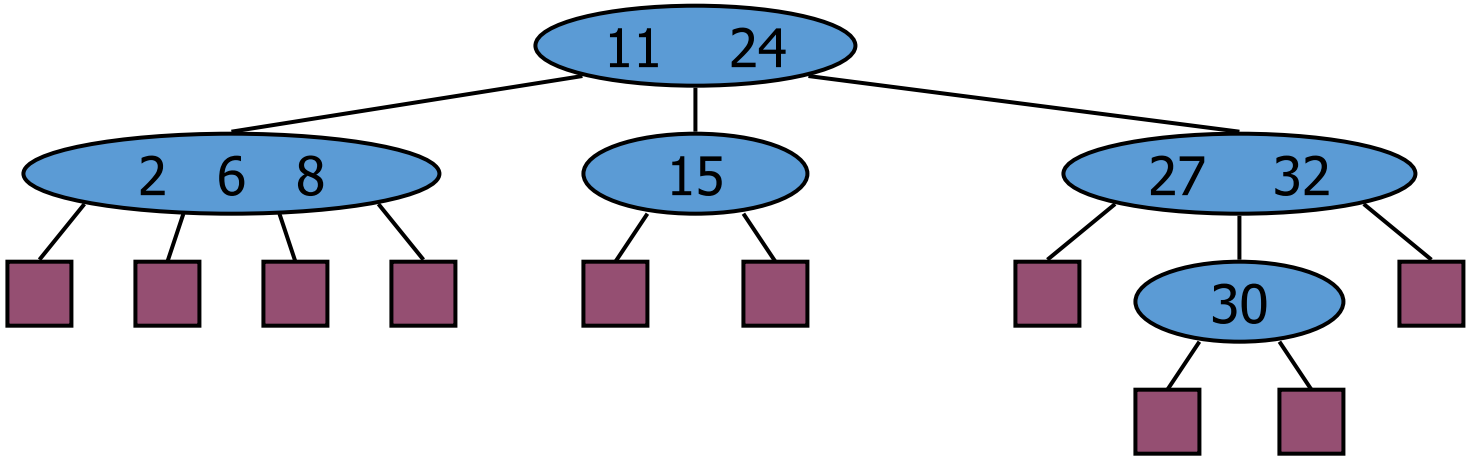
AVL Tree Performance

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

(2, 4) Trees

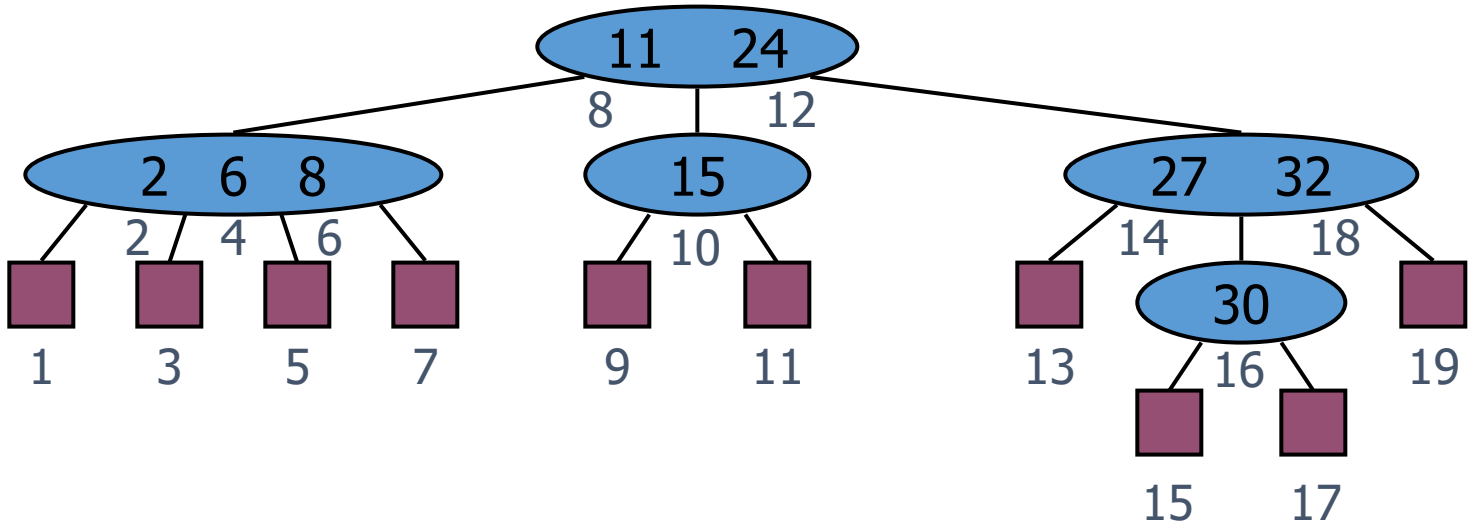
Multi-Way Search Tree

- 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 … v_d storing keys k_1 k_2 … 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, …, d - 1)
    - keys in the subtree of v_d are greater than k_{d-1}
  - The leaves store no items and serve as placeholders



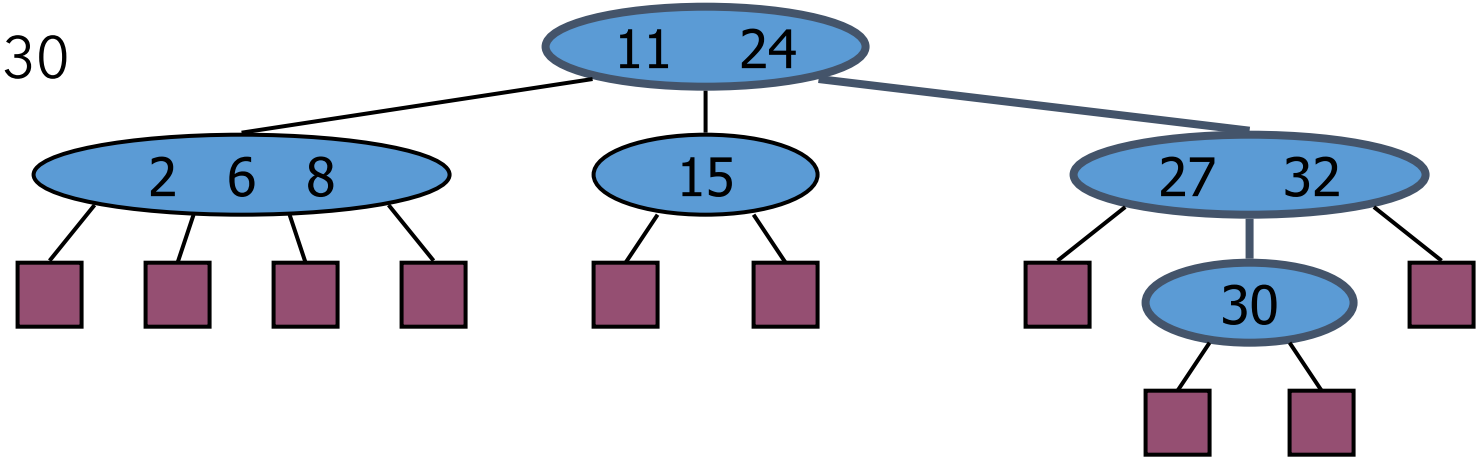
Multi-Way Inorder Traversal

- 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



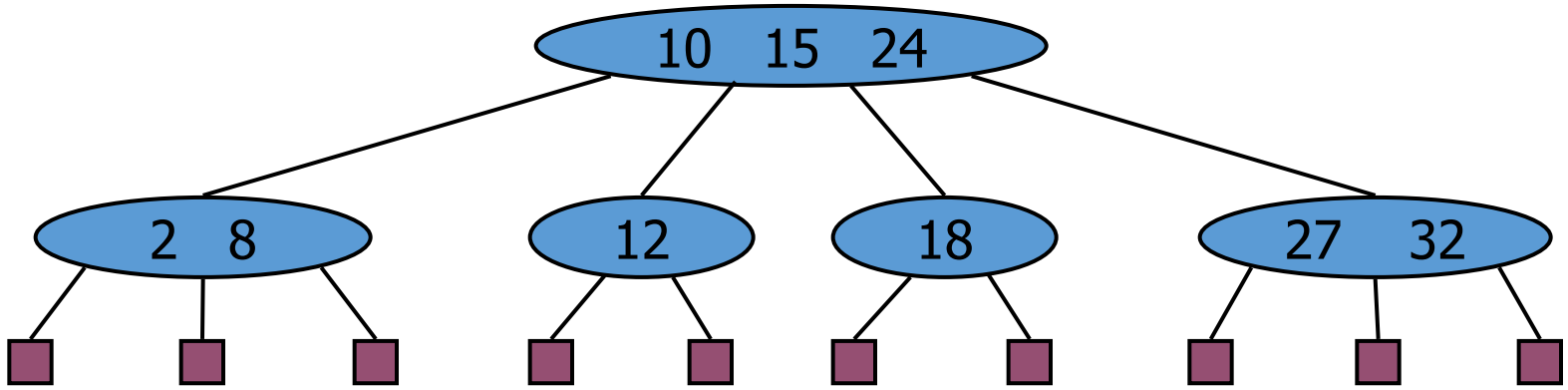
Multi-Way Searching

- Similar to search in a binary search tree
- A each internal node with children $\boldsymbol{v}_1\ \boldsymbol{v}_2\ \ldots\ \boldsymbol{v}_{\boldsymbol{d}}$ and keys $\boldsymbol{k}_1\ \boldsymbol{k}_2\ \ldots\ \boldsymbol{k}_{\boldsymbol{d}-1}$
  - $\boldsymbol{k} = \boldsymbol{k}_{\boldsymbol{i}}$ ($\boldsymbol{i} = 1, \ldots, \boldsymbol{d} - 1$): the search terminates successfully
  - $\boldsymbol{k} < \boldsymbol{k}_1$: we continue the search in child $\boldsymbol{v}_1$
  - $\boldsymbol{k}_{\boldsymbol{i}-1} < \boldsymbol{k} < \boldsymbol{k}_{\boldsymbol{i}}$ ($\boldsymbol{i} = 2, \ldots, \boldsymbol{d} - 1$): we continue the search in child $\boldsymbol{v}_{\boldsymbol{i}}$
  - $\boldsymbol{k} > \boldsymbol{k}_{\boldsymbol{d}-1}$: we continue the search in child $\boldsymbol{v}_{\boldsymbol{d}}$
- 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 $O(\log n)$

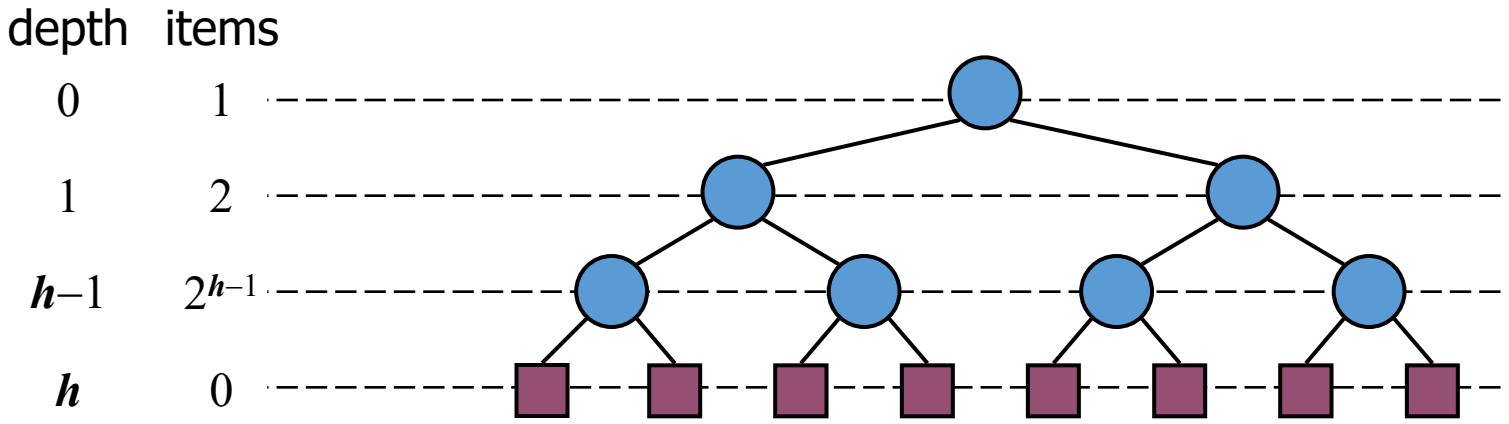
  Proof:

  - 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

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

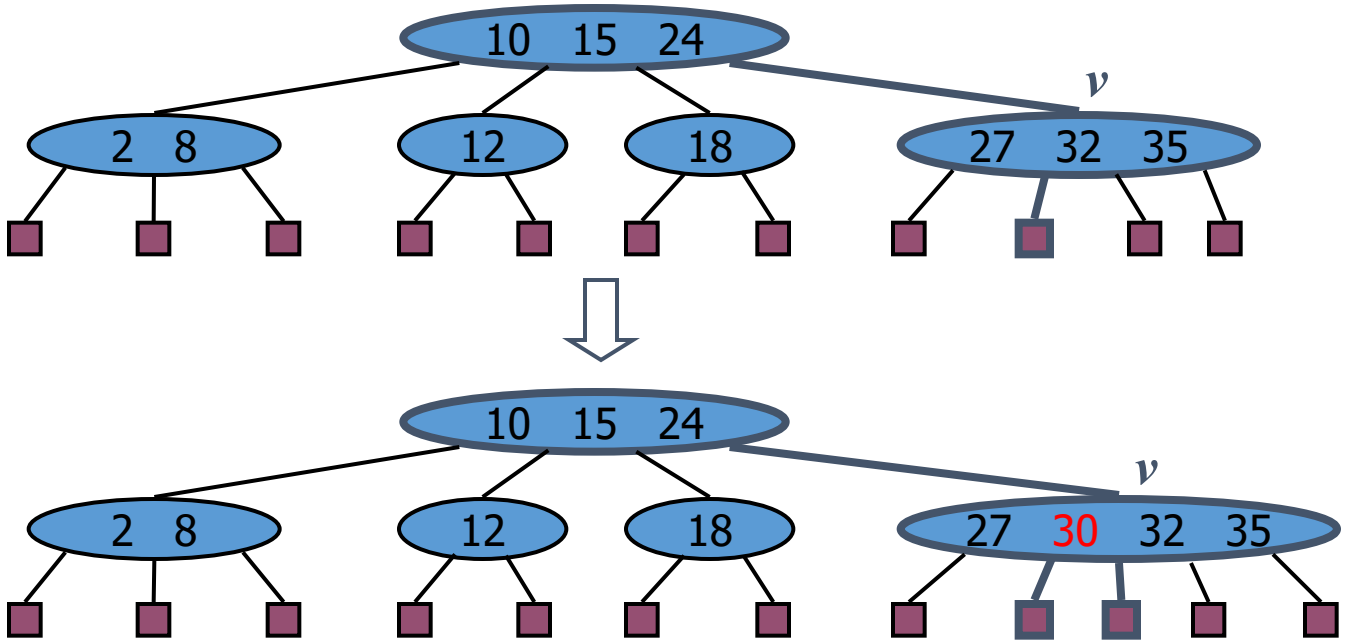
  - Thus, $h \leq \log (n + 1)$

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



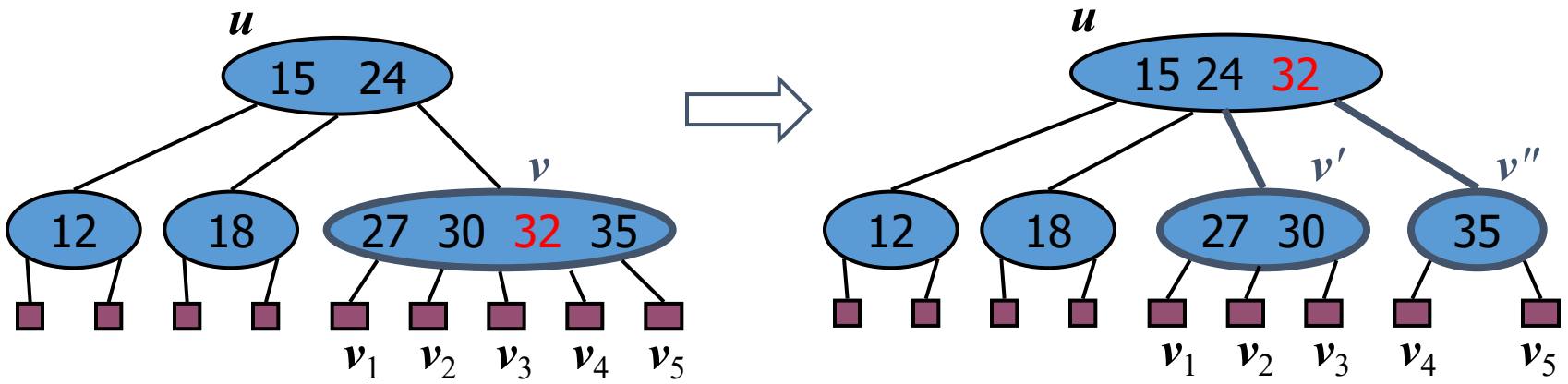
Insertion

- 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



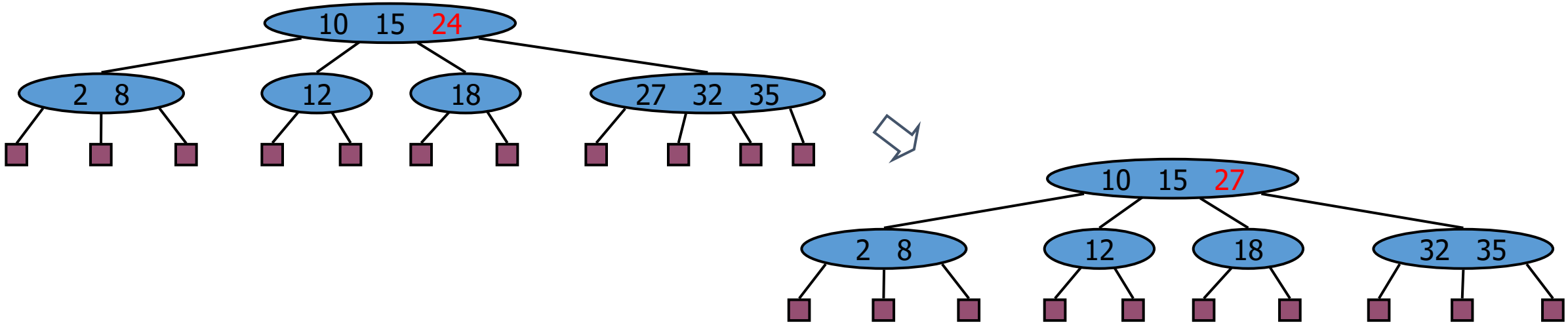
Overflow and Split

- 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 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



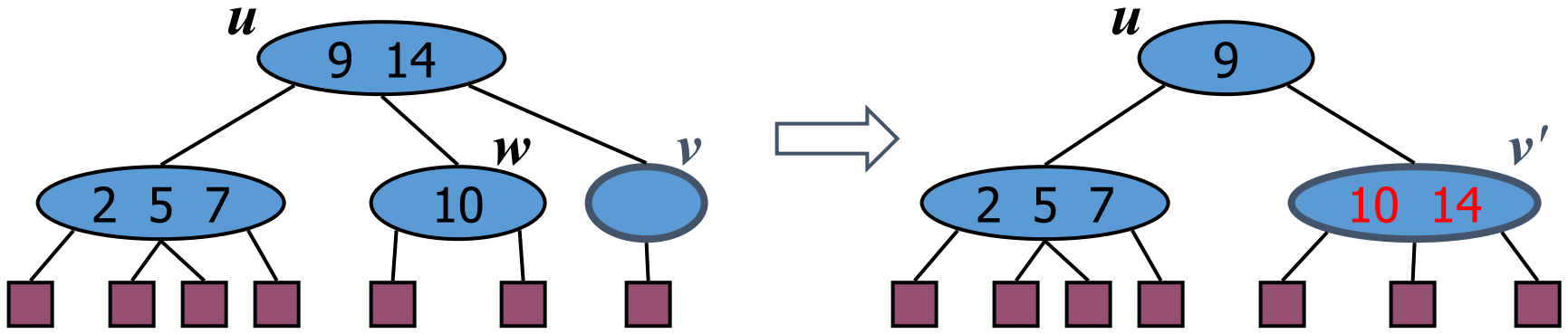
Deletion

- 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)



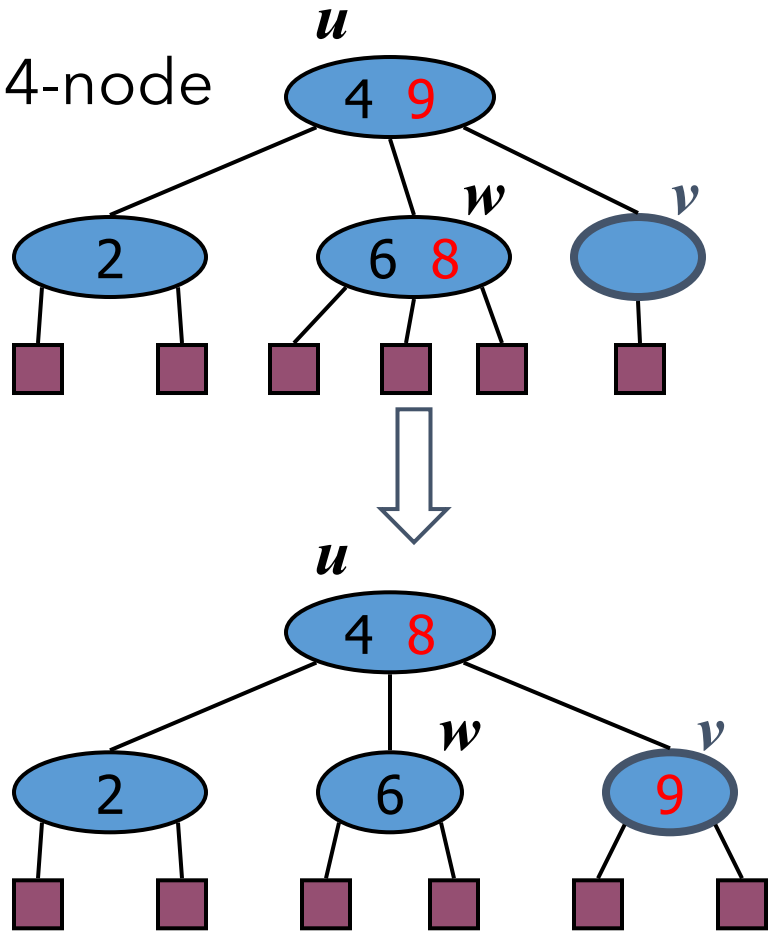
Underflow and Fusion

- 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



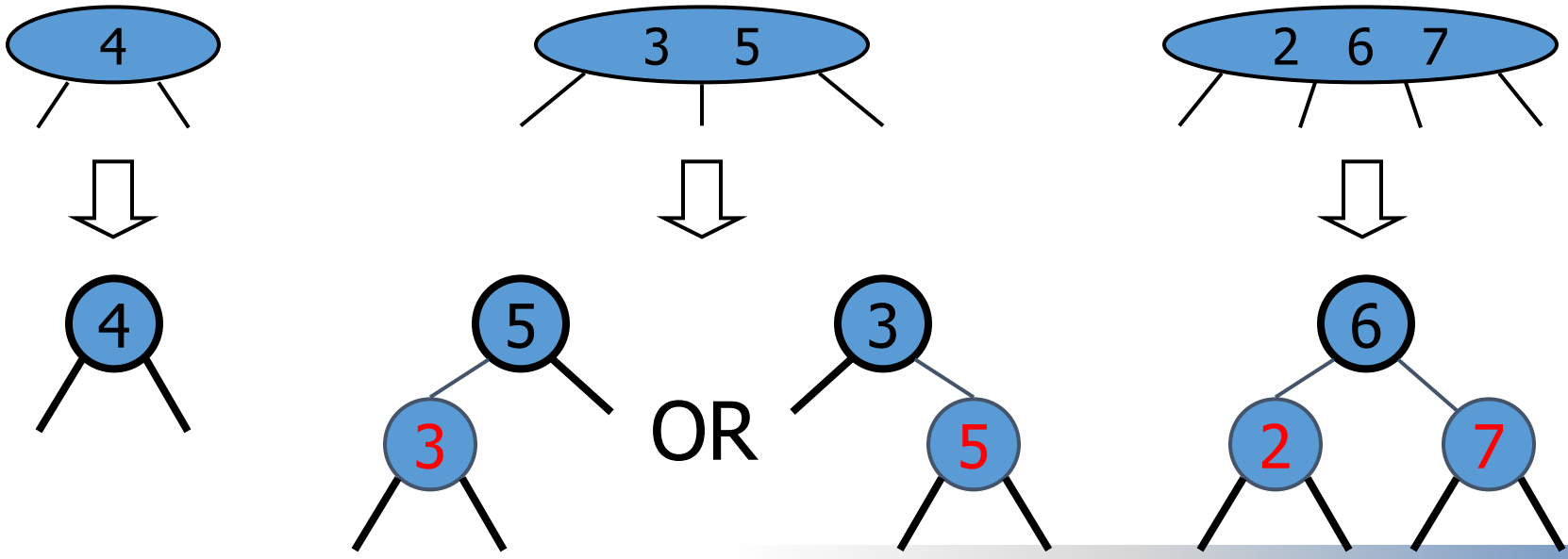
Analysis of Deletion

- 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

Red-Black Trees

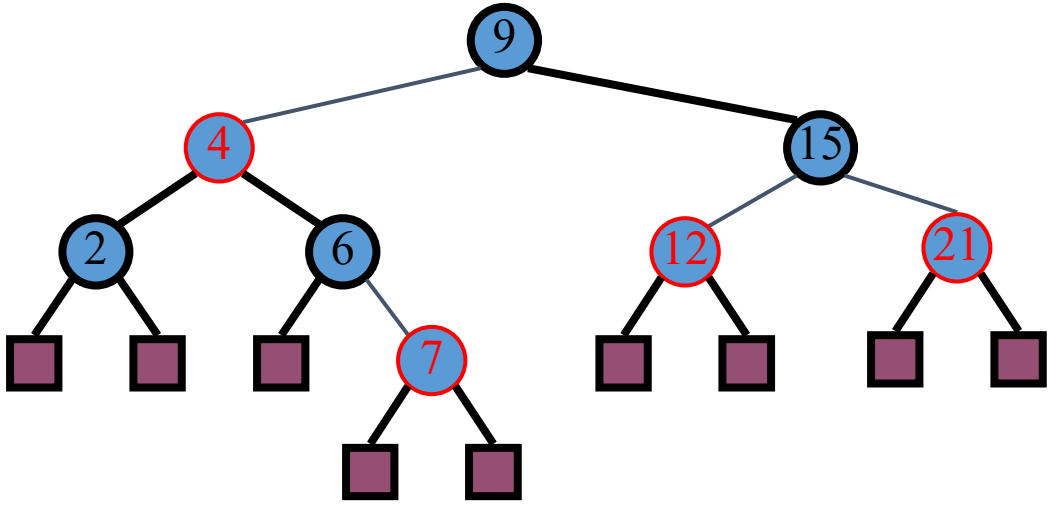
From (2,4) to Red-Black Trees

- 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 items has height $O(\log n)$

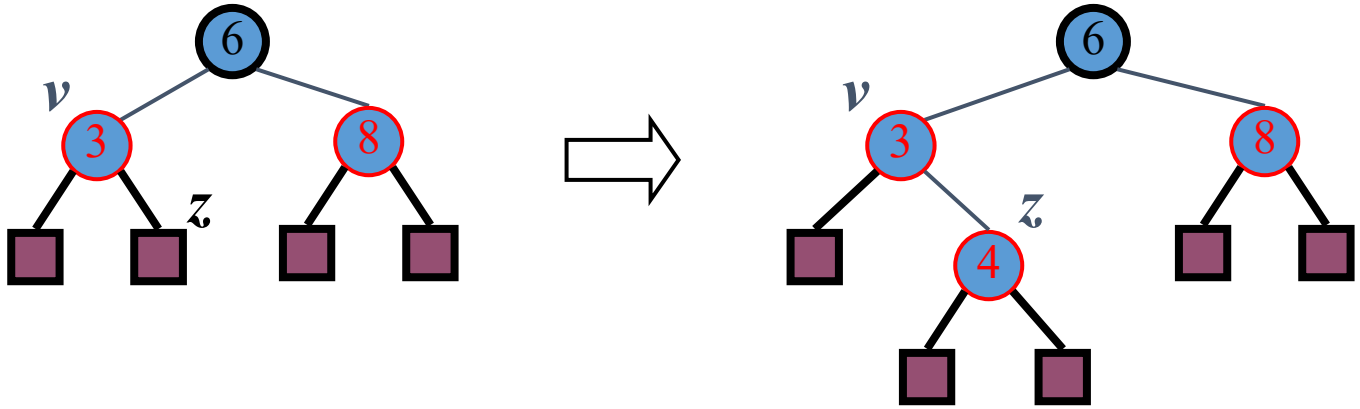
  Proof:

  - 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 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:

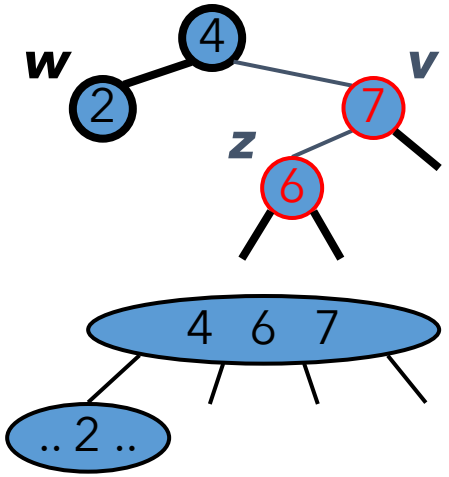


Remedying a Double Red

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

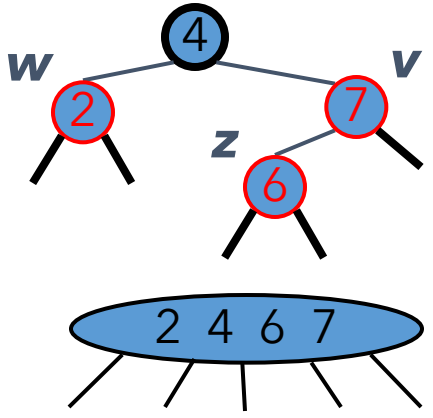
Case 1: ***w*** is black

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



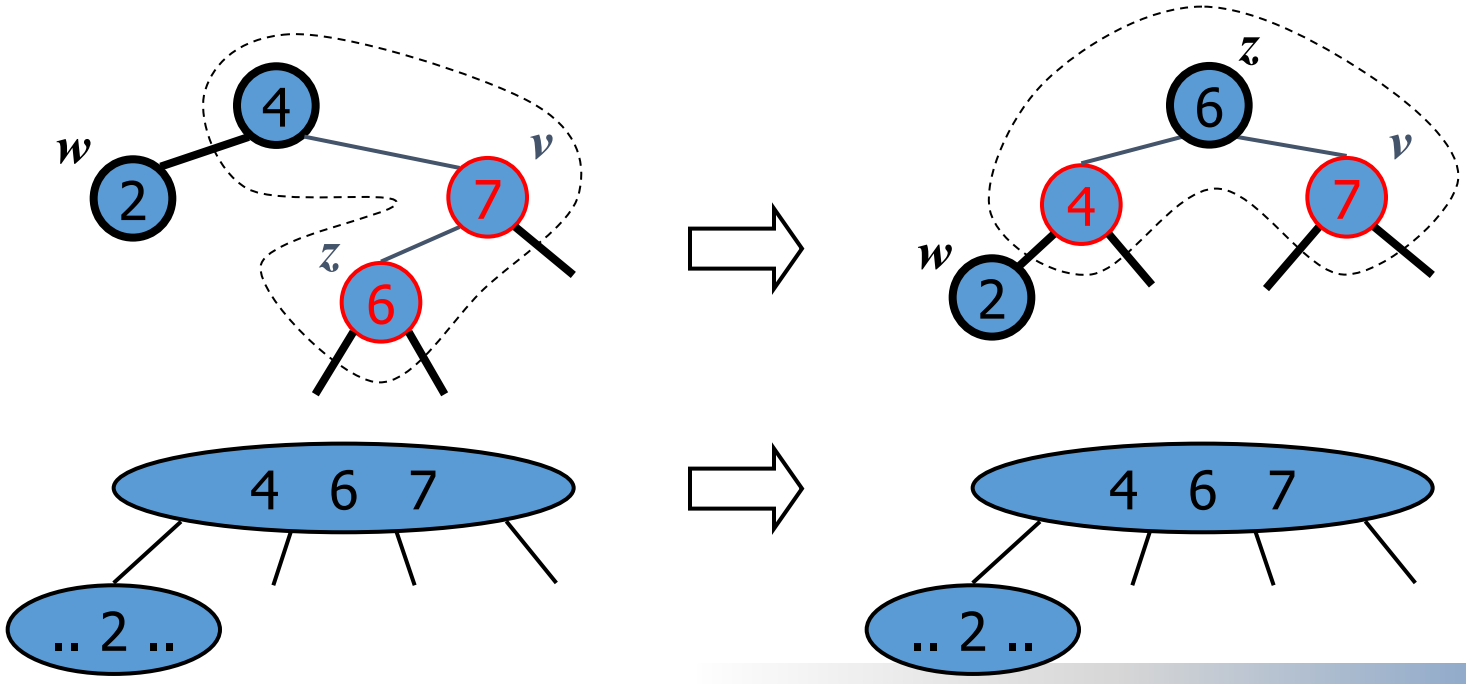
Case 2: ***w*** is red

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



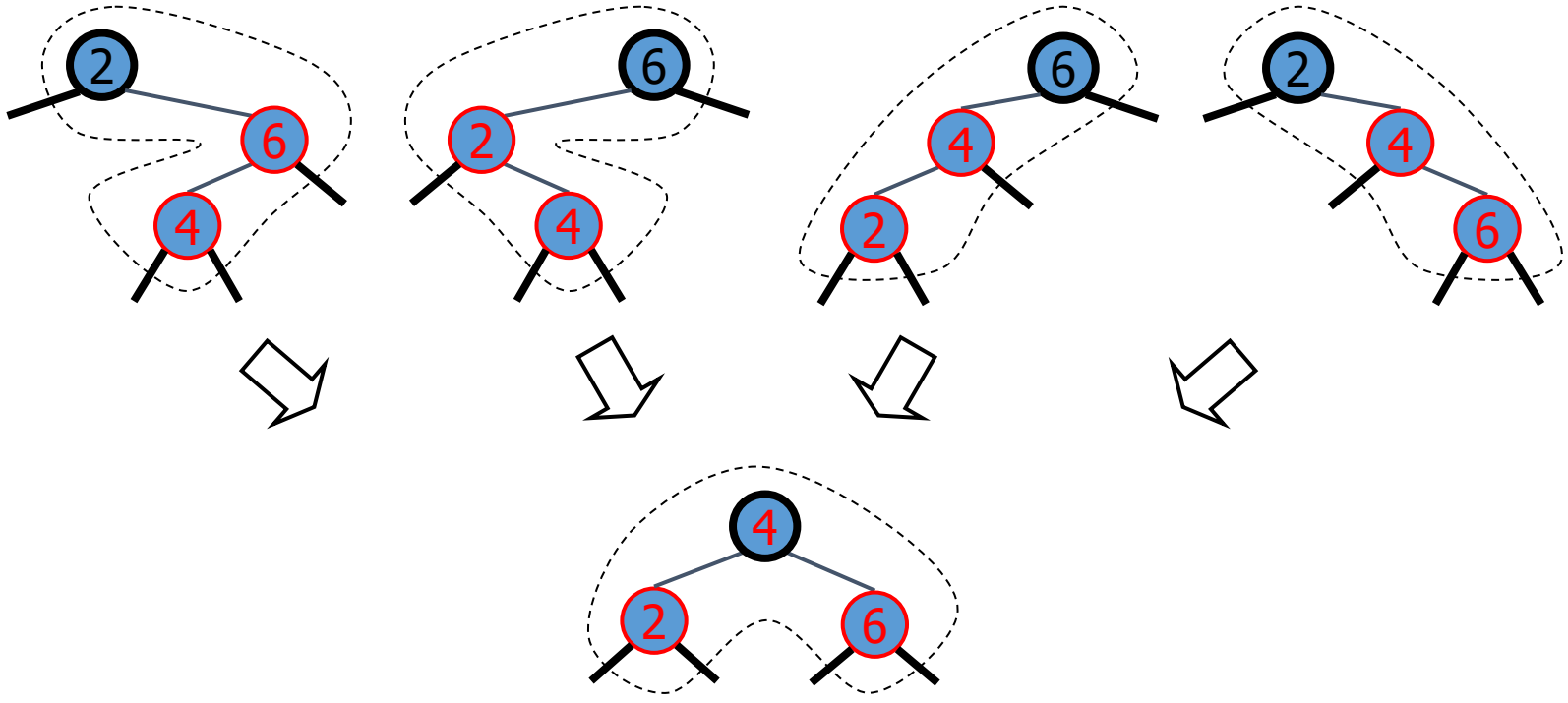
Restructuring

- 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



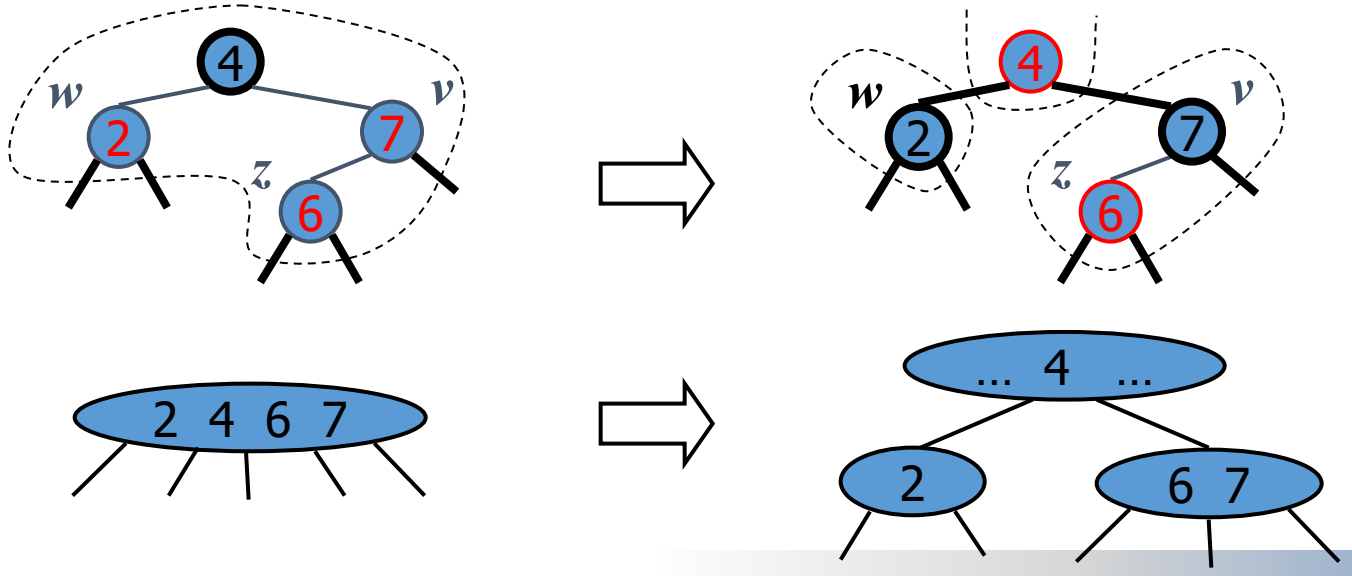
Restructuring (cont.)

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

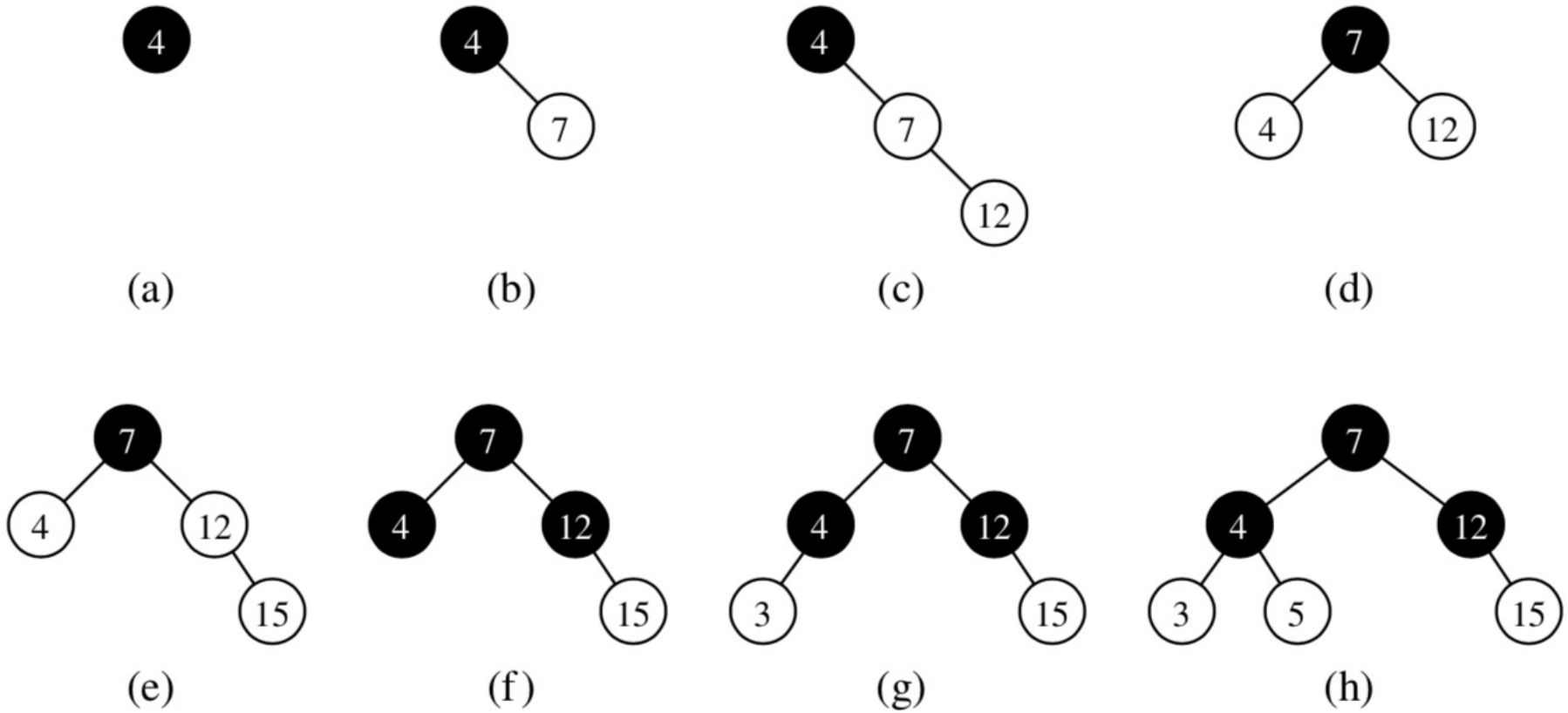


Recoloring

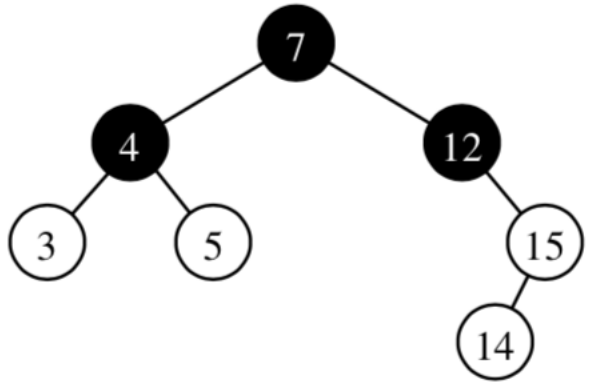
- A recoloring remedies a child-parent double red when the parent red node has a red sibling
- The parent ***v*** and its sibling ***w*** become black and the grandparent ***u*** becomes red, unless it is the root
- It is equivalent to performing a split on a 5-node
- The double red violation may propagate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



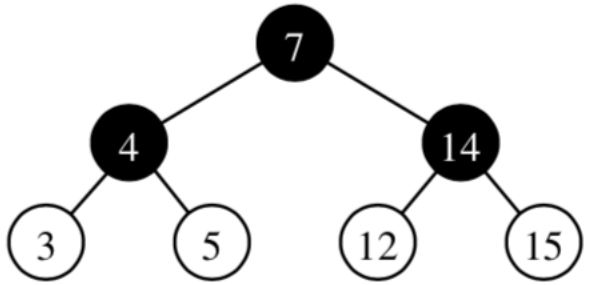
4, 7, 12, 15, 3, 5, 14, 18, 16, 17



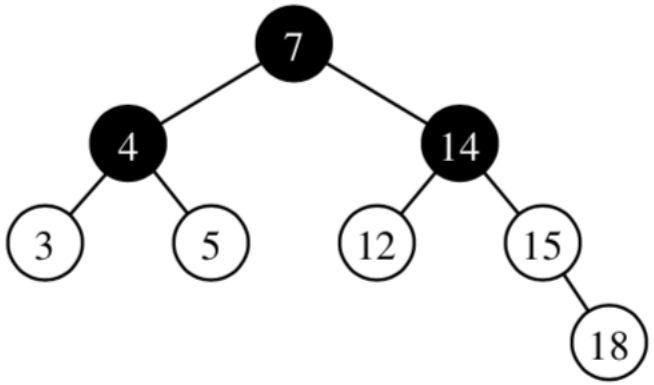
4, 7, 12, 15, 3, 5, 14, 18, 16, 17



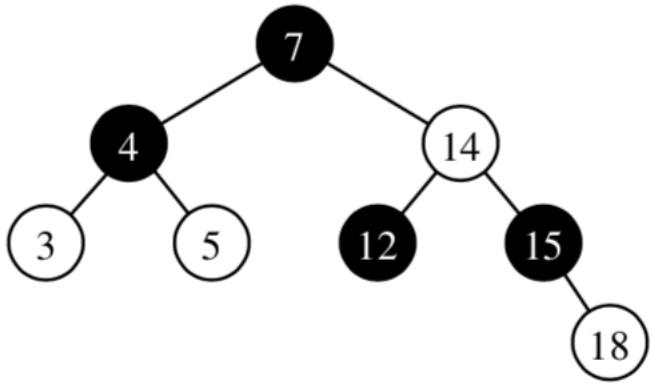
(i)



(j)

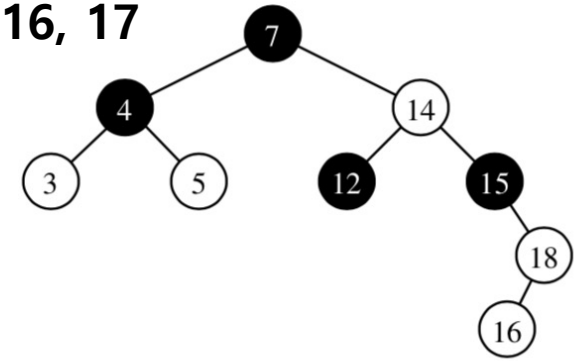


(k)

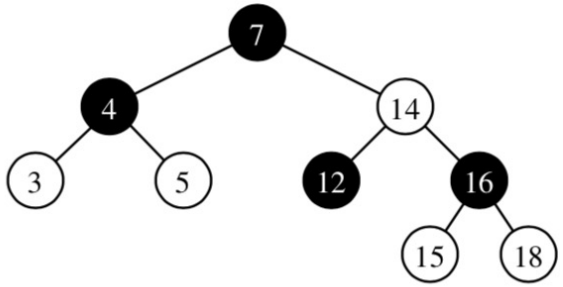


(l)

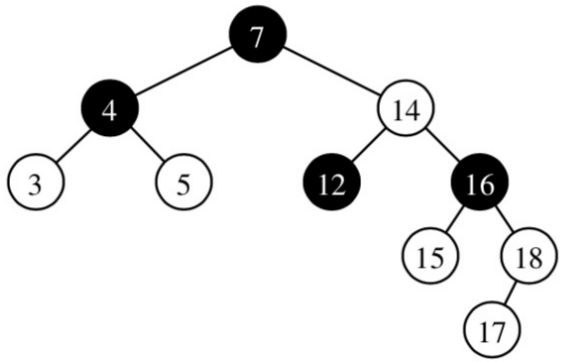
4, 7, 12, 15, 3, 5, 14, 18, 16, 17



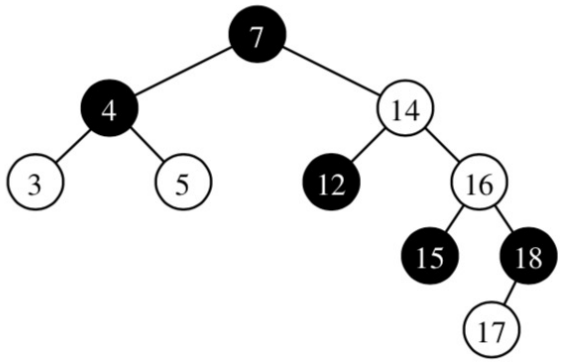
(m)



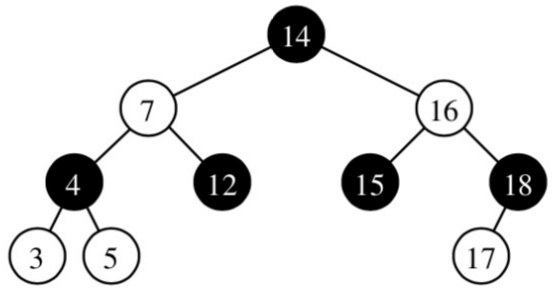
(n)



(o)



(p)



(q)

Analysis of Insertion

Algorithm *insert*(k, o)

1. We search for key k to locate the insertion node z

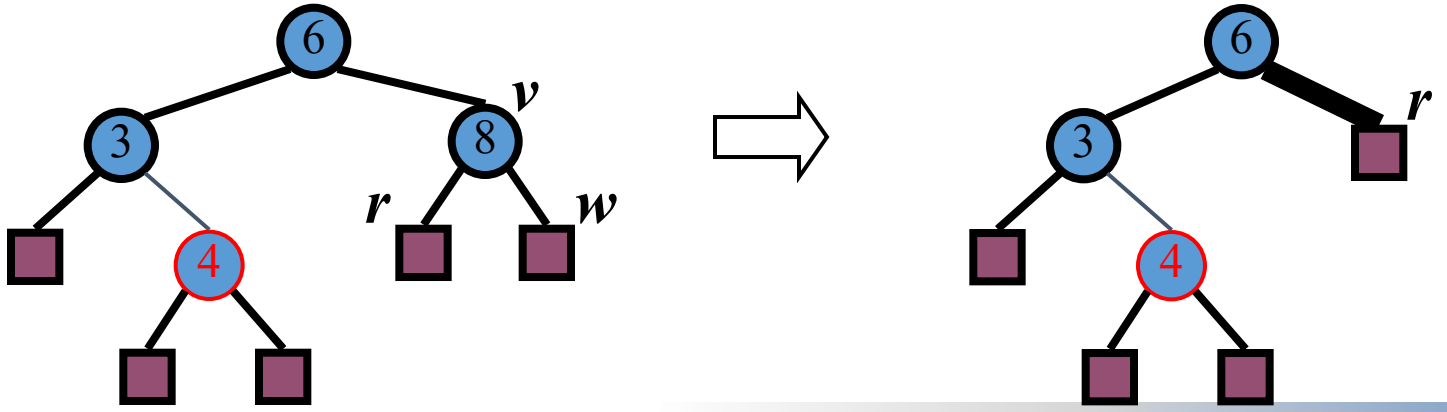
2. We add the new entry (k, o) at node z and color z red

3. **while** *doubleRed*(z)
   if *isBlack*(*sibling*(*parent*(z)))
      $z \leftarrow$ *restructure*(z)
      return
   else { *sibling*(*parent*(z)) is red }
      $z \leftarrow$ *recolor*(z)

- 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

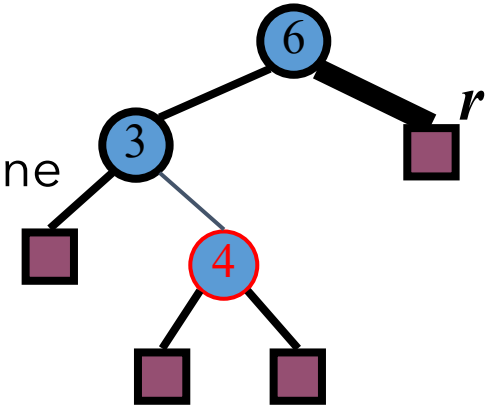
- To perform operation remove(***k***), we first execute the deletion algorithm for binary search trees
- Let ***v*** be the internal node removed, ***w*** the external node removed, and ***r*** the sibling of ***w***
  - If either ***v*** or ***r*** was red, we color ***r*** black and we are done
 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

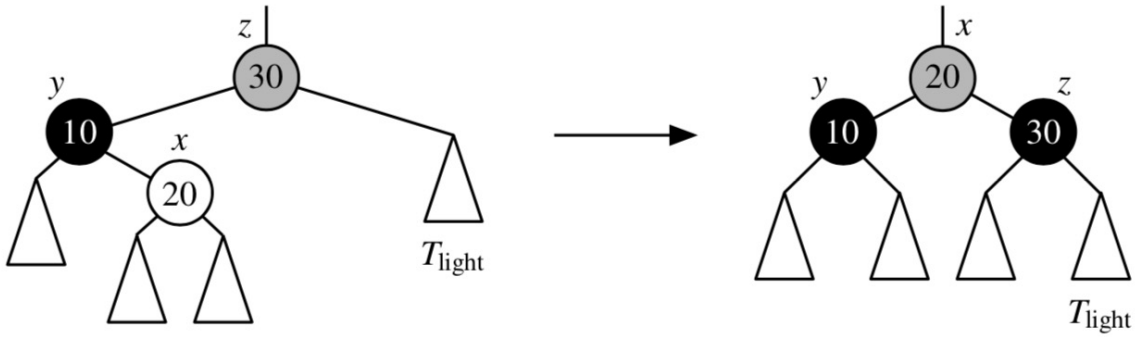


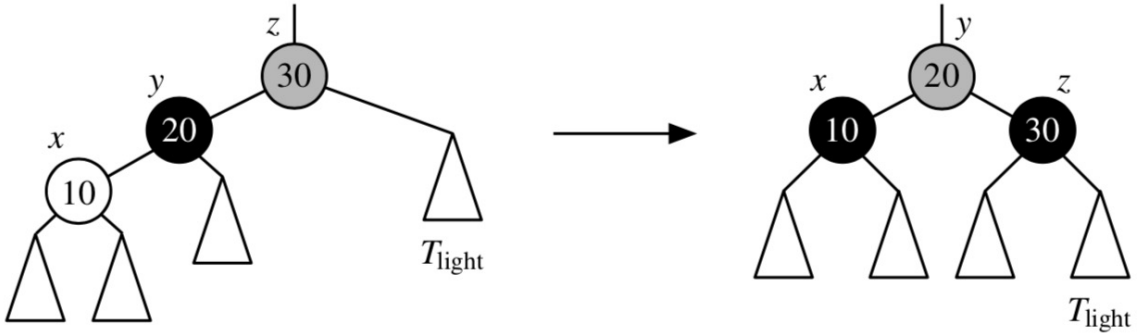
Remedying a Double Black

- 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





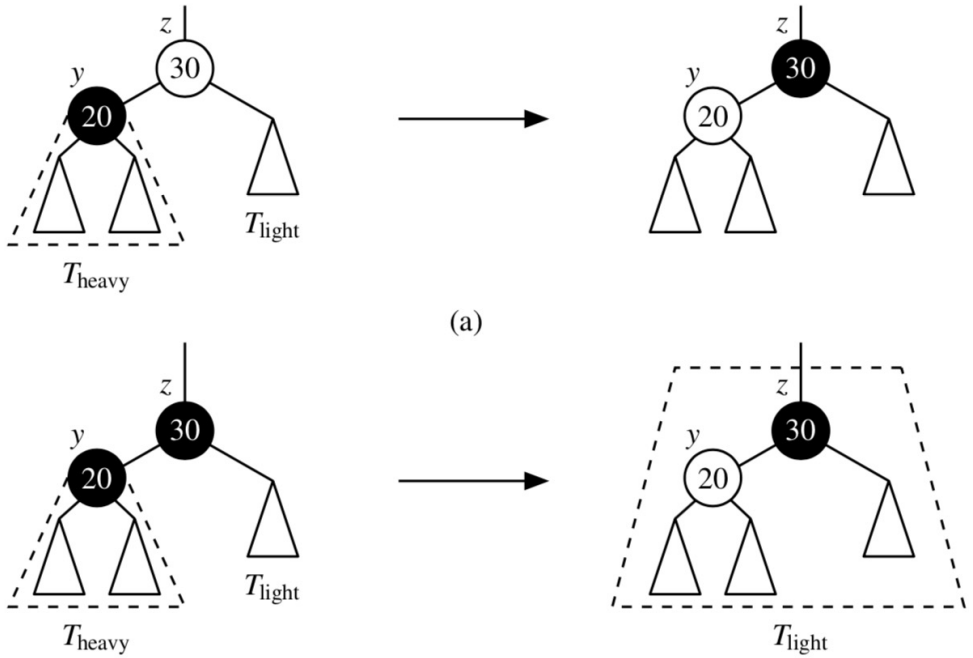


Remedying a Double Black

- The algorithm for remedying a double black node ***w*** with sibling ***y*** considers three cases

  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



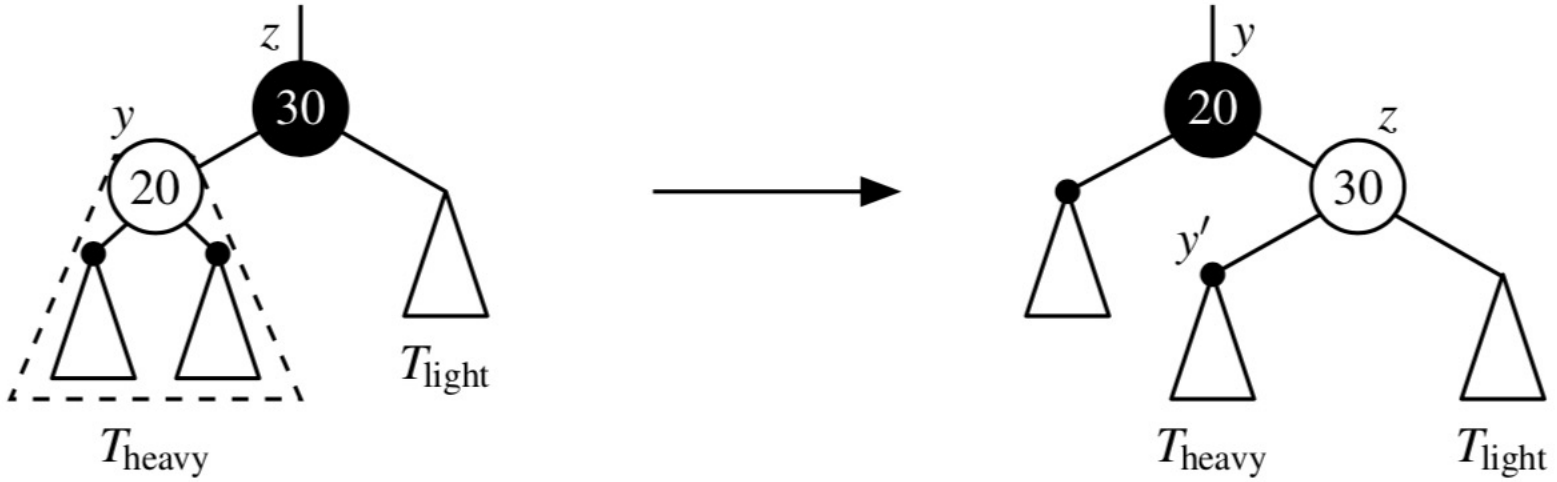
Remedying a Double Black

- The algorithm for remedying a double black node ***w*** with sibling ***y*** considers three cases

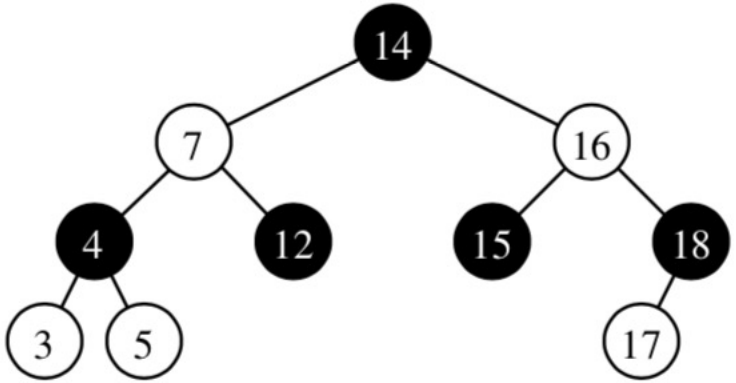
  Case 3: ***y*** is red

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

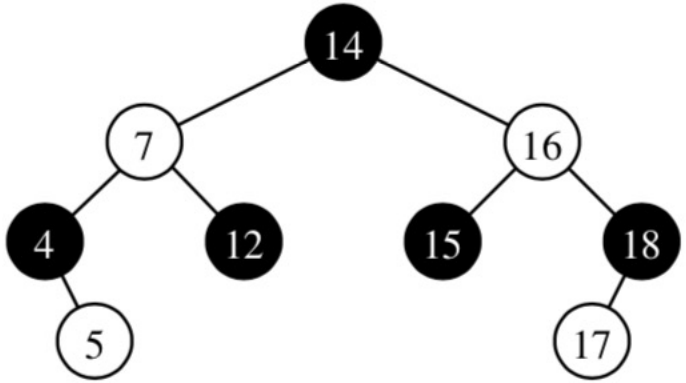
- Deletion in a red-black tree takes ***O***(log ***n***) time



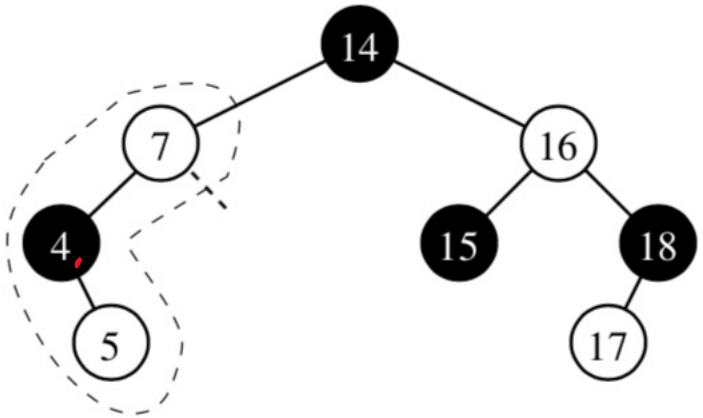
3, 12, 17, 18, 15, 16



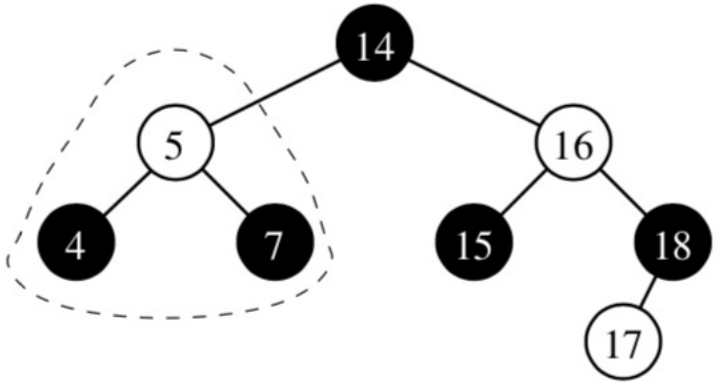
(a)



(b)



(c)



(d)

3, 12, 17, 18, 15, 16