SEARCH TREES

School of Artificial Intelligence

LAST LECTURE:
(2,4) TREES AND RED-BLACK TREES

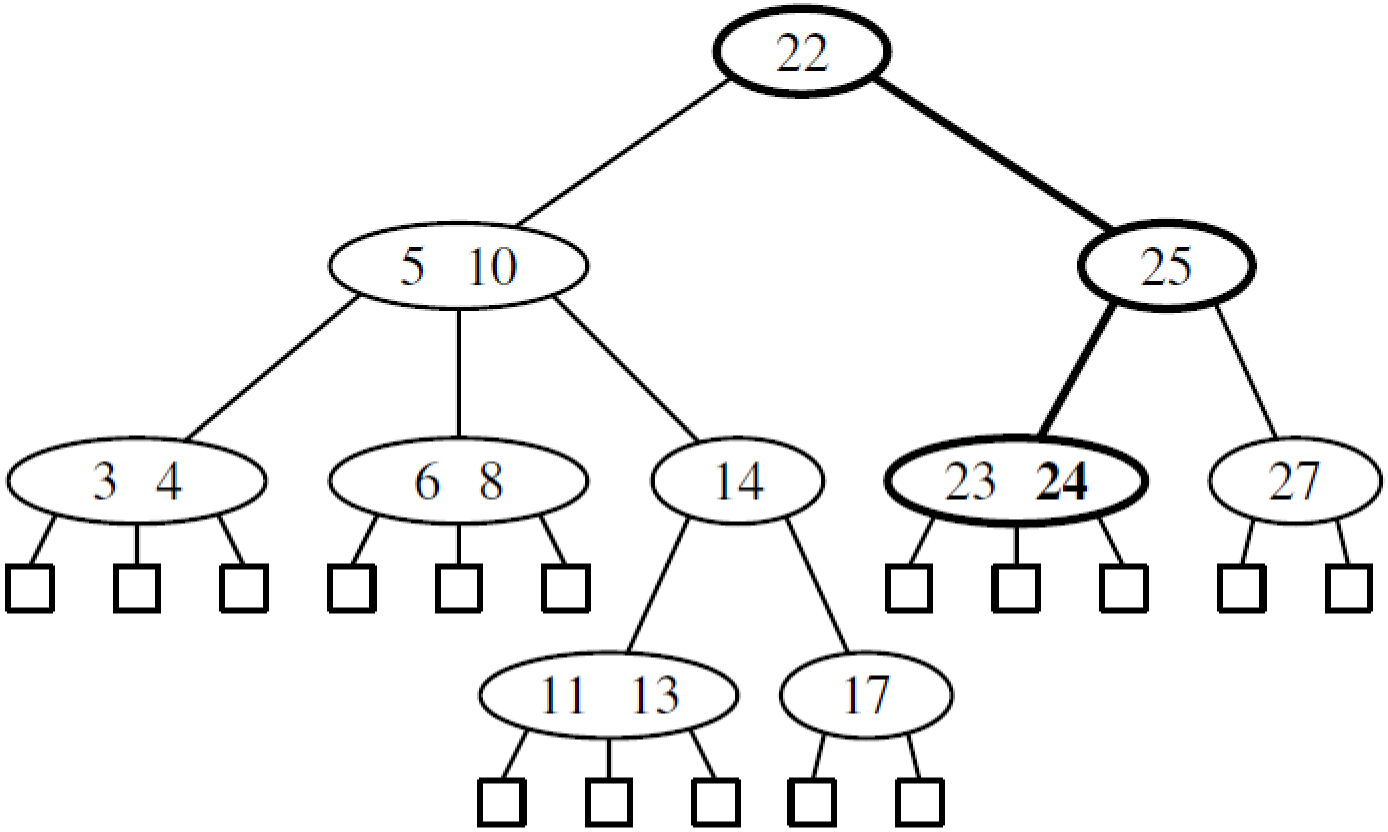
- Multi-way search tree: Internal nodes may have more than two children
  - let w be a node of an ordered tree
  - W is a d-node if w has d children

    - Each internal node of T has at least two children. That is, each internal node is a d-node such that $d \geq 2$.
    - Each internal d-node w of T with children $c_1, \ldots, c_d$ stores an ordered set of $d-1$ key-value pairs $(k_1, v_1), \ldots, (k_{d-1}, v_{d-1})$, where $k_1 \leq \cdots \leq k_{d-1}$.
    - Let us conventionally define $k_0 = -\infty$ and $k_d = +\infty$. For each item (k, v) stored at a node in the subtree of w rooted at c_i, $i = 1, \ldots, d$, we have that $k_{i-1} \leq k \leq k_i$.

  - A d-node stores d-1 regular keys
  - External nodes do not store any data and serve only as "placeholders"
    - Reference to None
- An n-item multiway search tree has n+1 external nodes

MULTIWAY SEARCH TREE

- Search:
  - Start from the root
  - Compare k with d-nodes
  - Search successful
    - Locate key
  - Search unsuccessful
    - Reach an external node

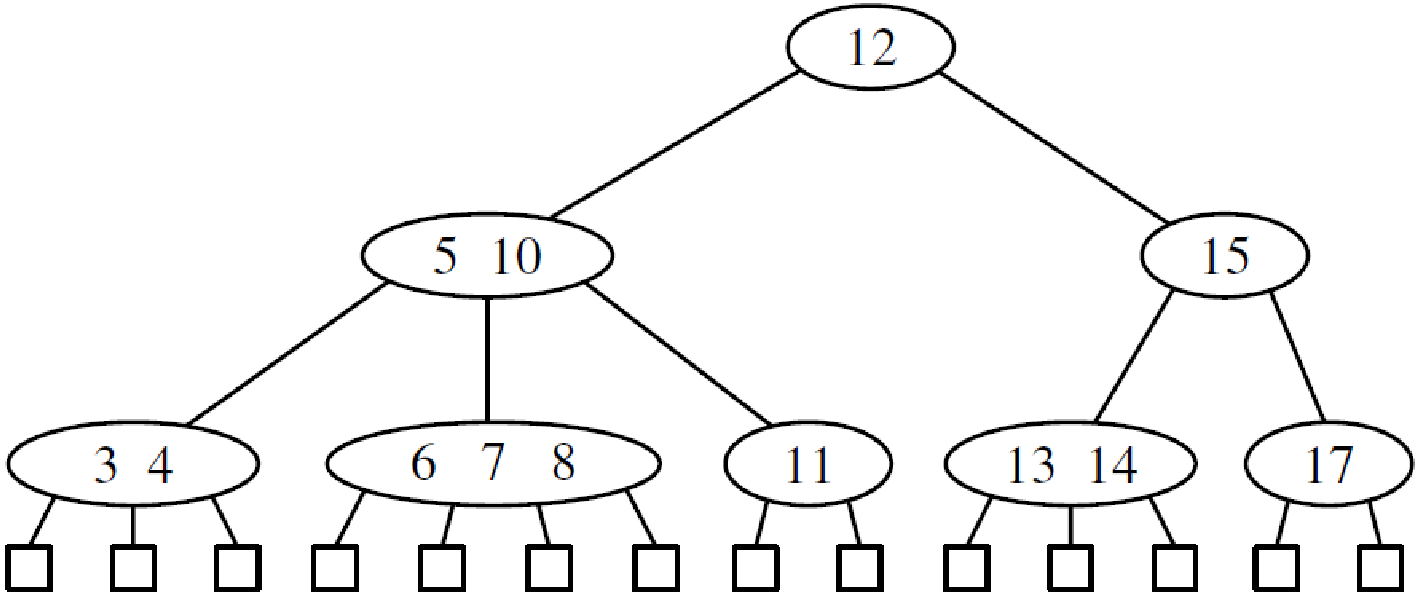


DATA STRUCTURE FOR MULTIWAY SEARCH TREES

- General tree: linked data structure
- Secondary container: finding the smallest key at the node that is greater than or equal to k
  - Sorted map: find_ge(k)
  - SortedTableMap from previous lecture
    - Associated value in case of a match for key k, or the child c_i such that $k_{i-1} < k < k_i$
    - k_i in the secondary structure to pair(v_i, c_i)
    - Process d-node when searching for an item of T with key k can be performed using binary search in O(log d), d = number of children
    - d_{max} = maximum number of children of any node of T, h = height of T, search time in a multiway search tree is $O(h \log d_{max})$
    - If d_{max} is a constant, the running time for performing a search is O(h)

(2,4) TREES

- Sometimes 2-4 tree or 2-3-4 tree
- **Size property**: every internal node has at most four children
- **Depth property**: all external nodes have the same depth
- The height of a 2-4 tree storing n items is O (log n)
  - Sufficient to keep the tree balanced
  - Search takes O(log n) time

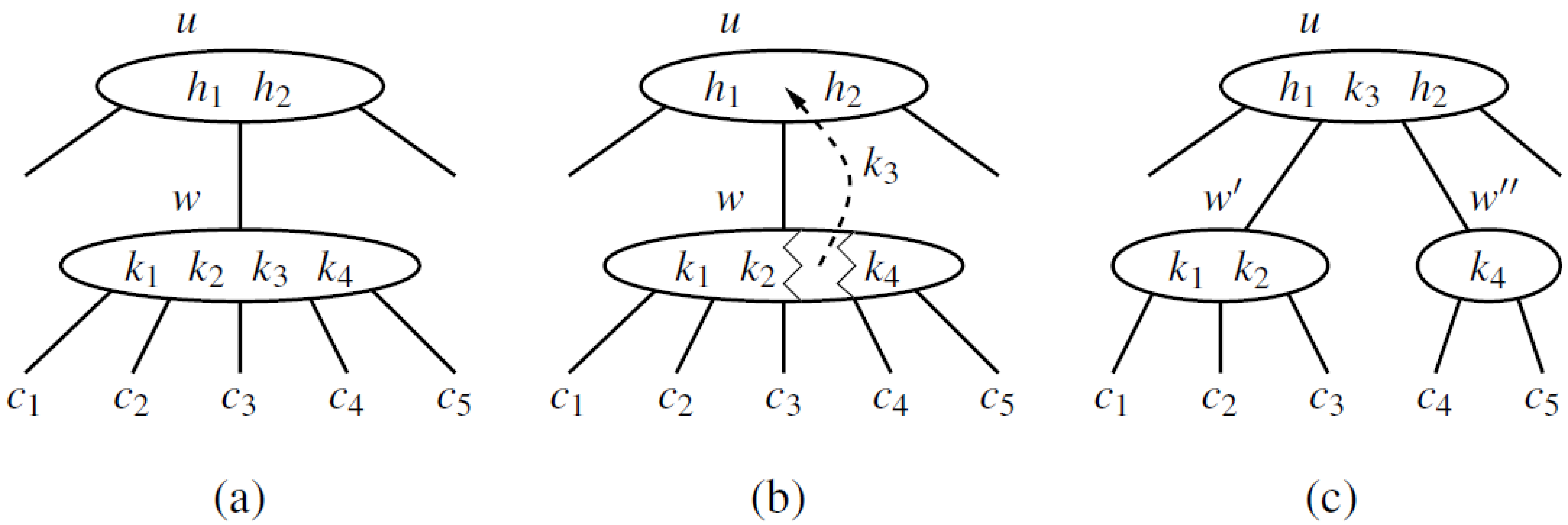


(2,4) TREES INSERTION

- Insert new item (k, v), search for k
  - k not in T: locate an external node z.
    - if w is the parent of z.
    - insert new item into node w and add a new child y to w on the left of z
  - may violate the size property
    - 4-node becomes 5-node after the insertion – **overflow**
    - Resolution: **split**
      - Replace w with two nodes w' and w'', where
        - w' is a 3-node with children c1, c2, c3 storing keys k1 and k2
        - w'' is a 2-node with children c4, c5 storing key k4
      - If w is the root of T, create a new root node u; else, let u be the parent of w
      - Insert key k3 into u and make w' and w'' children of u, so that if w was child I of u, then w' and w'' become children I and i+1 of u

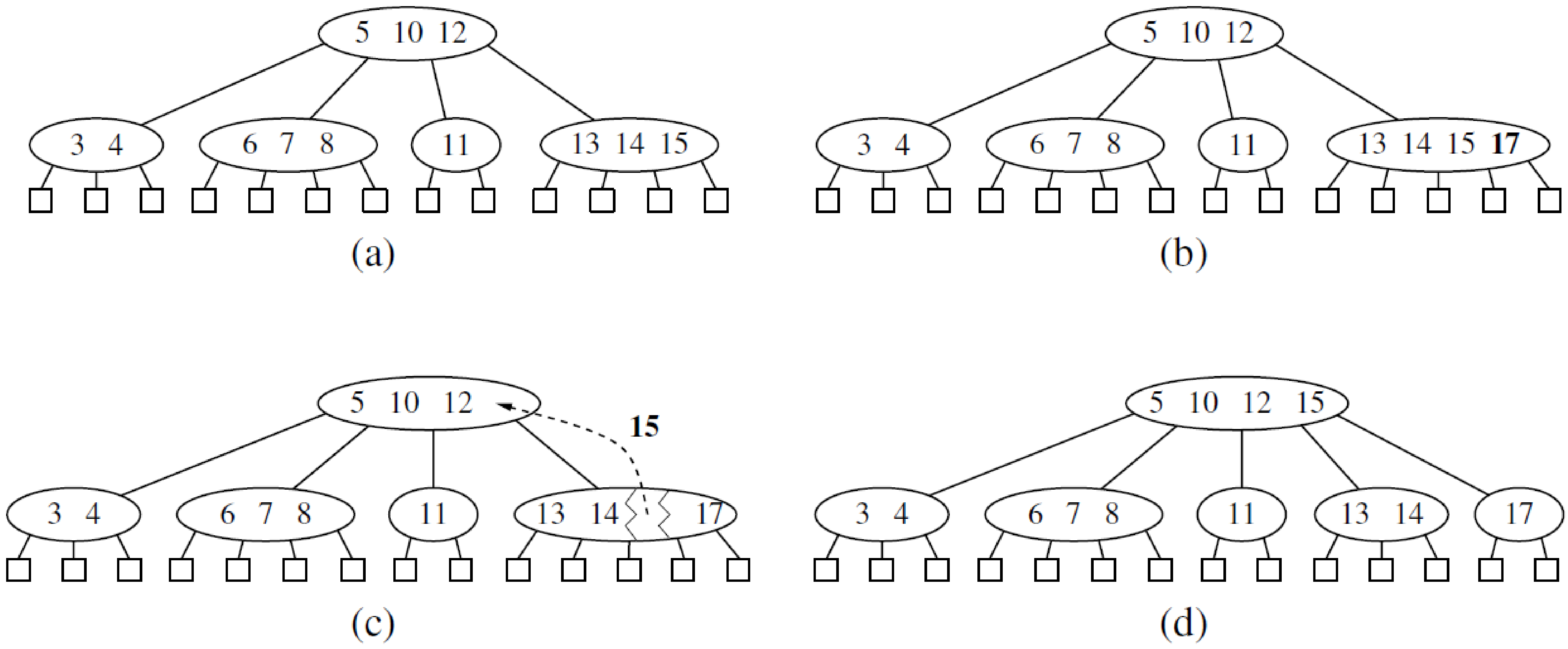
(2,4) TREES INSERTION

- Node split



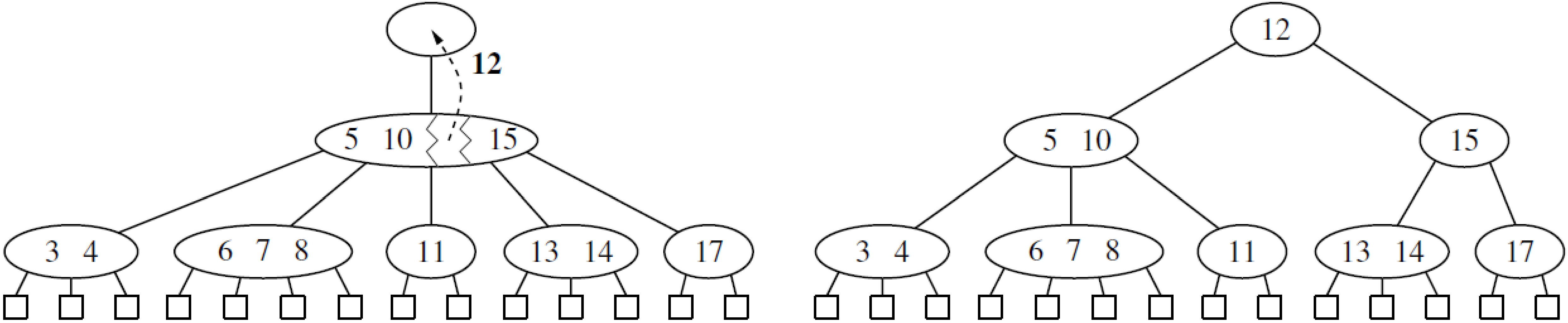
(2,4) TREES INSERTION

- Node split



(2,4) TREES INSERTION

- Node split



(e) (f)

(2,4) TREES INSERTION

- Analysis
  - d_{max} is at most 4, search for the placement of new key k uses O(1) time at each level, and O(log n) time overall
  - Split operations: bounded by the height of the tree
  - Insertion process runs in O(log n) time

(2,4) TREES DELETION

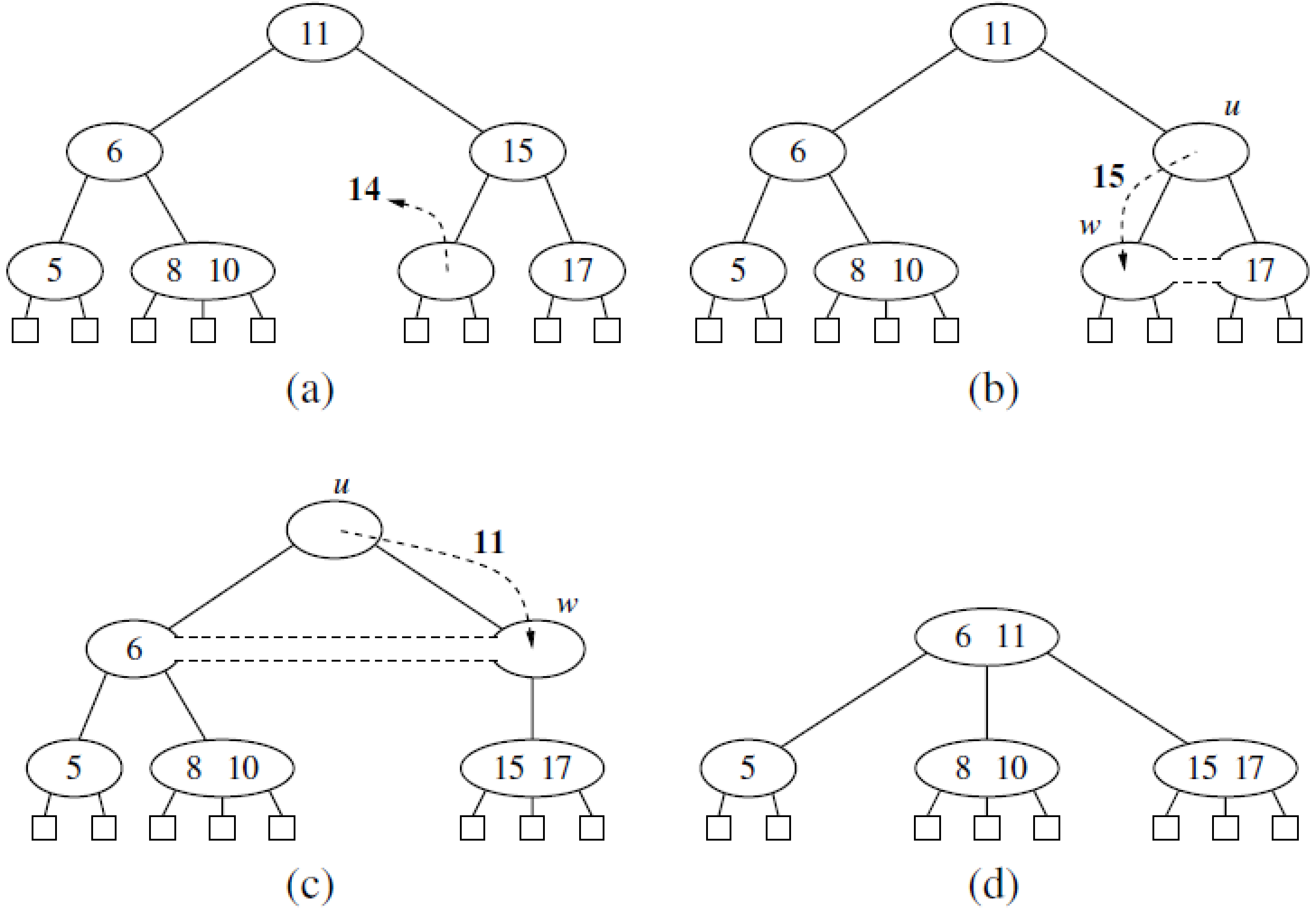
- Removal of an item
  - Search for k in T
  - Can always be reduced to - the item to be removed is stored at a node w whose children are external nodes
  - Item with key k to be removed is stored in the i^{th} item (k_i, v_i) at a node z that has only internal-node children,
  - swap item (k_i, v_i) with an appropriate item that is stored at a node w with external-node children
    - Find the rightmost internal node w in the subtree rooted at the i^{th} child of z
    - Swap the item (ki, vi) at z with the last item of w

(2,4) TREES DELETION

- Removal preserves the depth property, may violate the size property at w
  - 2-node becomes a 1-node with no items at all – **underflow**
  - Check if an immediate sibling s of w is a 3-node or a 4-node, then perform a **transfer:** move a child of s to w, a key of s to the parent u of w and s, and a key of u to w
  - If w has only one sibling, or if both immediate siblings of w are 2-nodes, then perform a **fusion:** merge w with a sibling to a new node w' and move a key from the parent u of w to w'

(2,4) TREES DELETION

- Fusion at node w may cause a new underflow to occur at the parent u of w, which triggers a transfer or fusion at u
- Number of fusion operations is bounded by the height of the tree – O(log n)



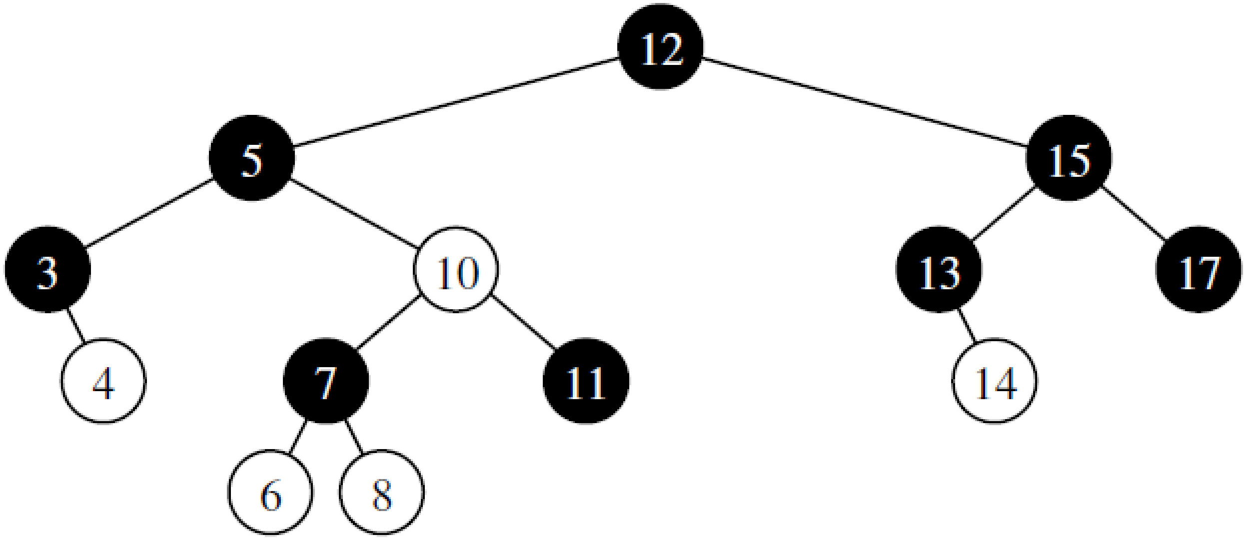
(2,4) TREES PERFORMANCE

- Identical to AVL tree
  - Height of a 2-4 tree with n items is O(log n)
  - Split, transfer, fusion: O(1)
  - Search, insertion, removal: O(log n)

Operation	Running Time
k in T	$O(\log n)$
T[k] = v	$O(\log n)$
T.delete(p), del T[k]	$O(\log n)$
T.find_position(k)	$O(\log n)$
T.first( ), T.last( ), T.find_min( ), T.find_max( )	$O(\log n)$
T.before(p), T.after(p)	$O(\log n)$
T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k)	$O(\log n)$
T.find_range(start, stop)	$O(s+\log n)$
iter(T), reversed(T)	$O(n)$

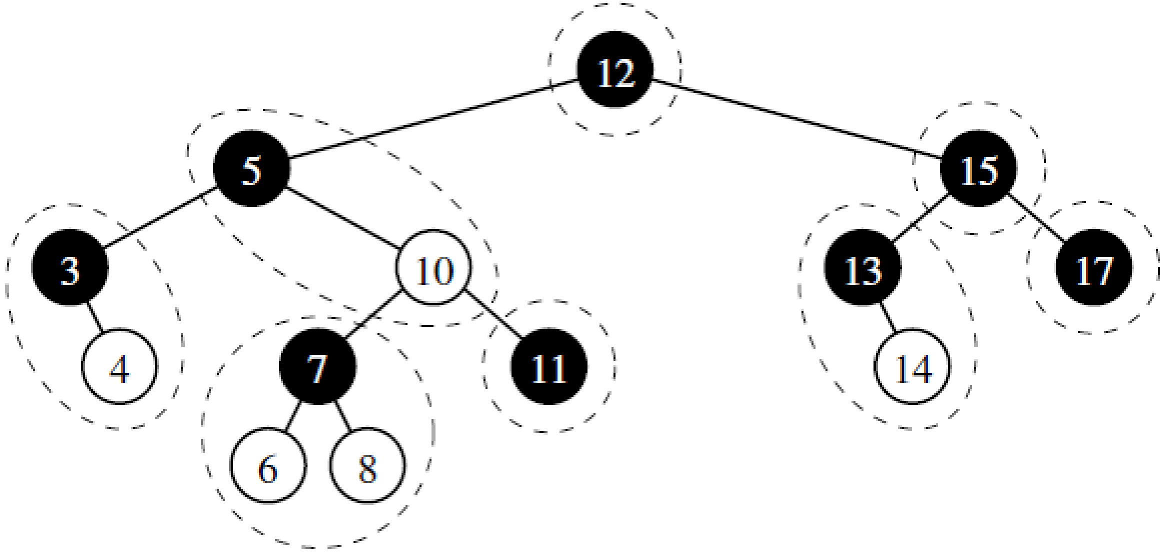
RED-BLACK TREES

- AVL trees – need to perform rotations
- 2-4 trees – need to perform split and fusion operations
- Red-black trees: O(1) structural changes after an update to stay balanced
- Red-black tree
  - Binary search tree, nodes coloured
  - **Root property**: root is black
  - **Red property**: the children of a red node are black
  - **Depth property**: all nodes with zero or one children have the same black depth
  - **Black depth**: number of black ancestors



RED-BLACK TREES

- Red-black trees and 2-4 trees
  - Given a red-black tree, a 2-4 tree can be constructed by merging every red node w into its parent,
  - storing the entry from w at its parent,
  - and with the children of w becoming ordered children of the parent
- Height of a red-black tree – O(log n)

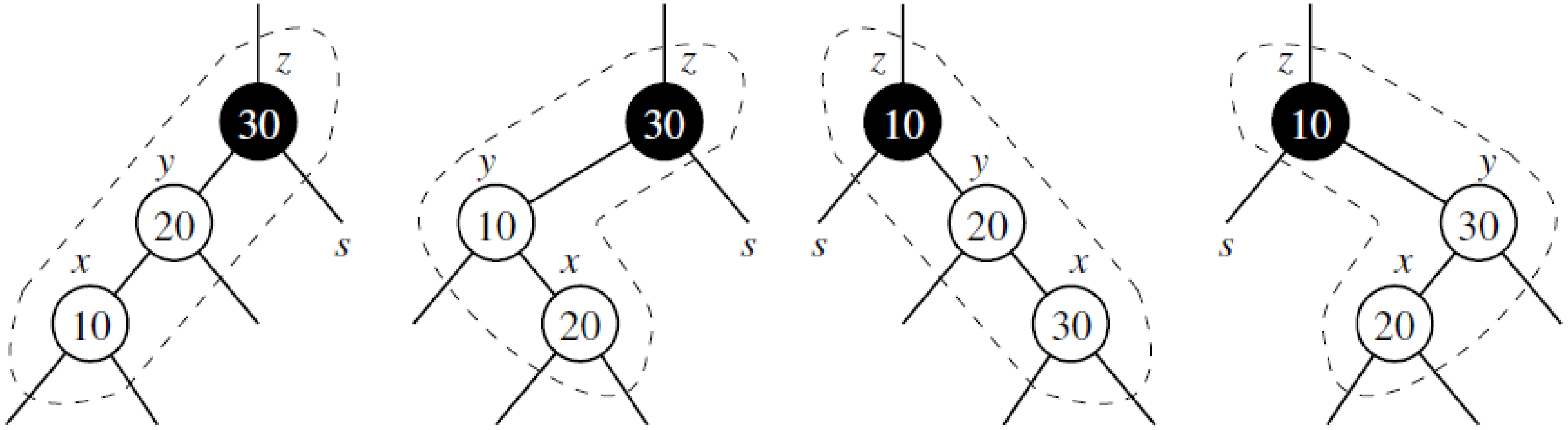


RED-BLACK TREES INSERTION

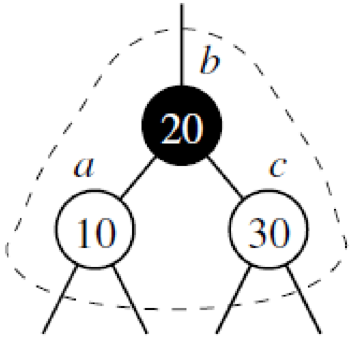
- Search: similar to binary search tree - O (log n)
  - returns position x
- If x is the root, colour it black
- Other cases, colour x red
- Insertion preserves the root and depth properties, but may violate the red property
- if x is not the root of T and parent y of x is red – double red situation

RED-BLACK TREES INSERTION

- Case 1: The sibling s of y is black (or None)
- Trinode restructuring
  - Node x, y, z
  - Label them a, b, and c
  - Replace z with the node labeled b and make nodes a and c the children of b
  - Colour b black and a and c red



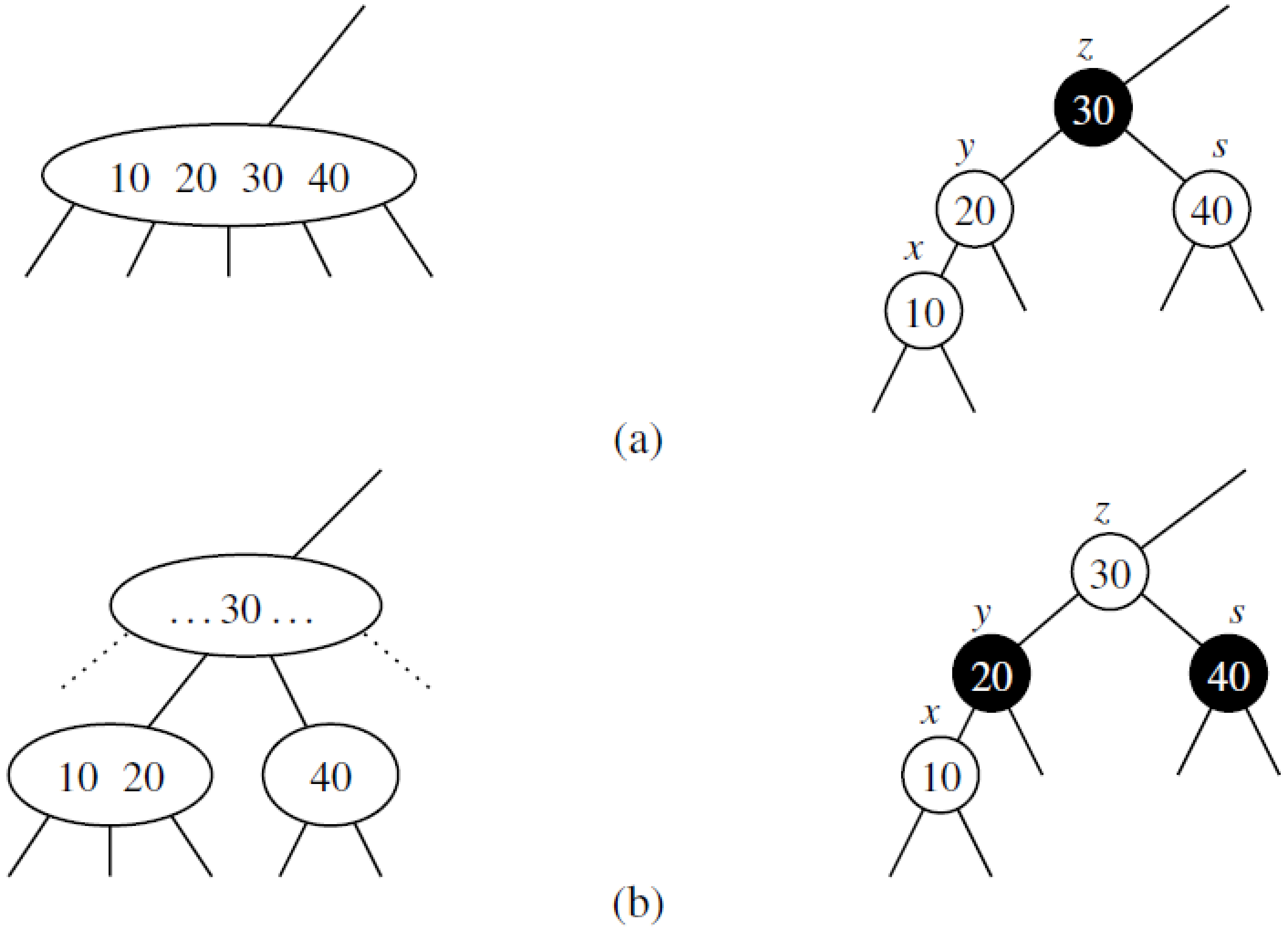
(a)



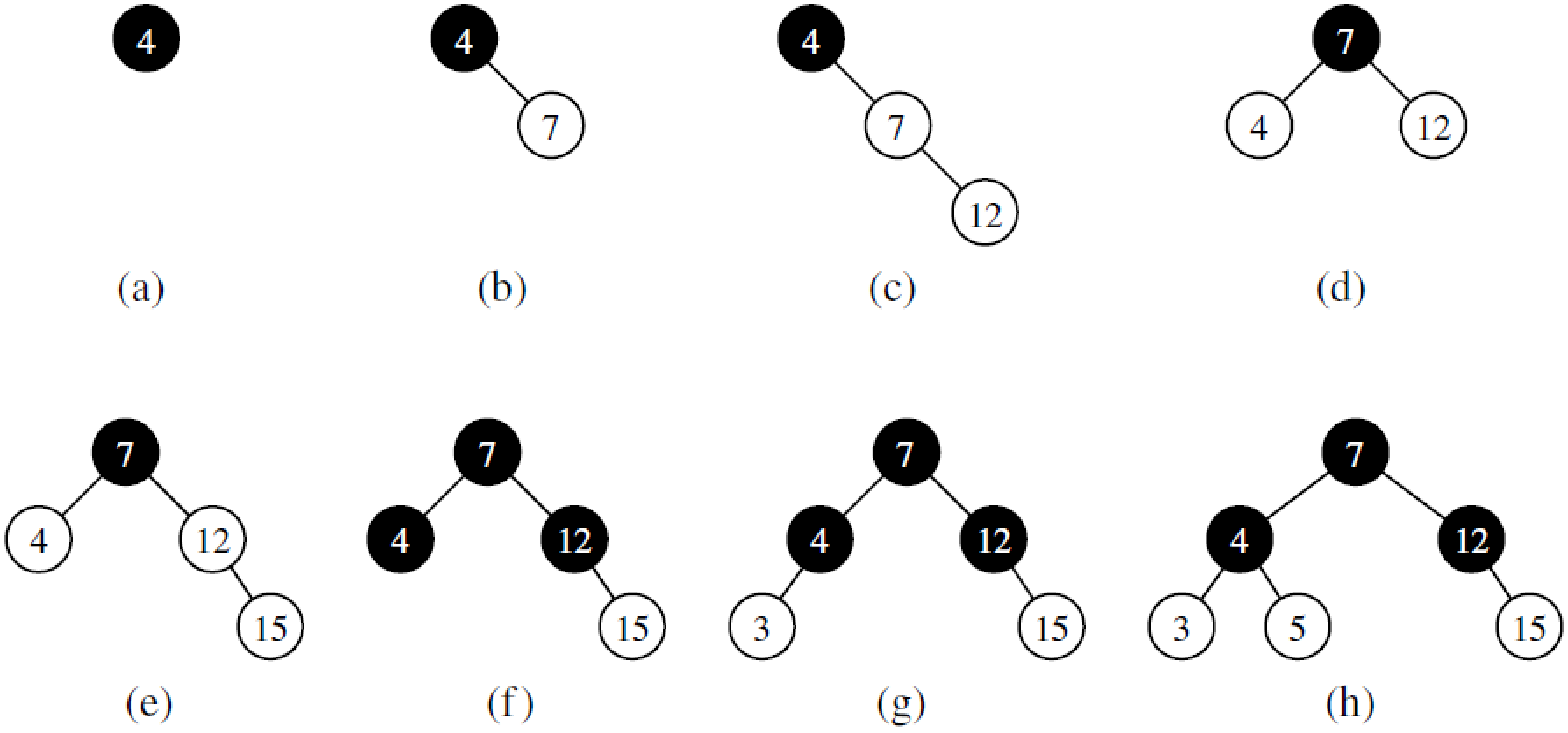
(b)

RED-BLACK TREES INSERTION

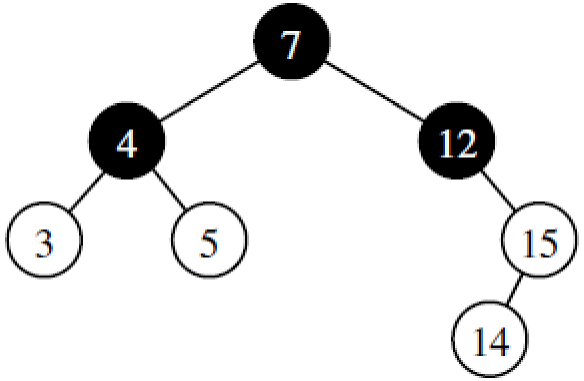
- Case 2: sibling s of y is red
- Overflow in its equivalent 2-4 tree
- Fix: **split/recolouring**
- Colour y and s black and their parent z red
- If z is root, it remains black
  - Unless z is the root, the portion of any path through the affected part of the tree is incident to one black node



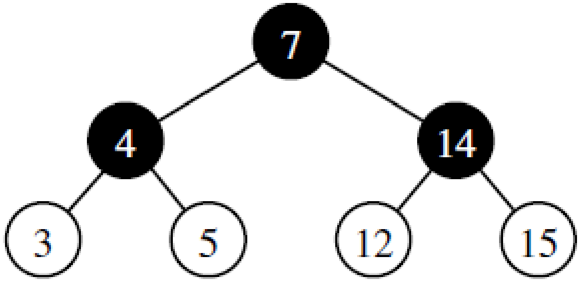
RED-BLACK TREES
INSERTION



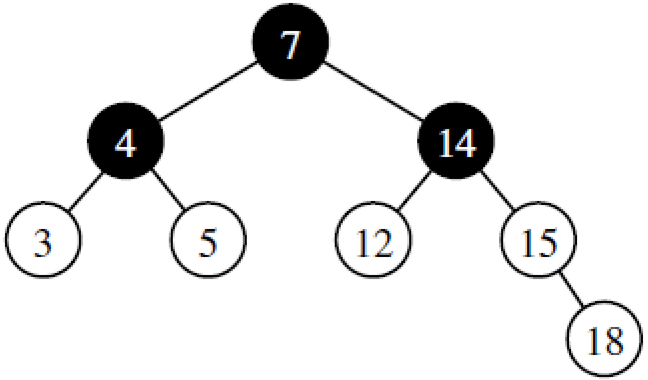
RED-BLACK TREES
INSERTION



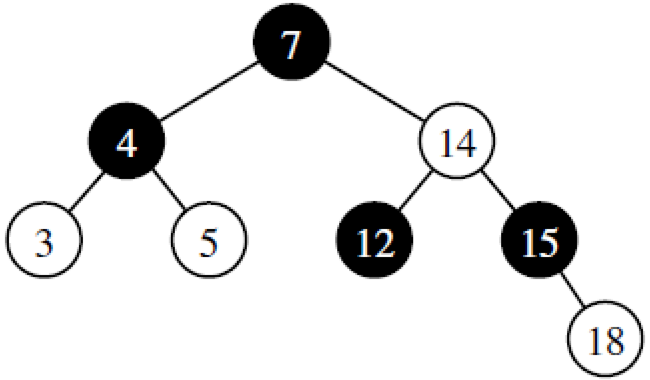
(i)



(j)

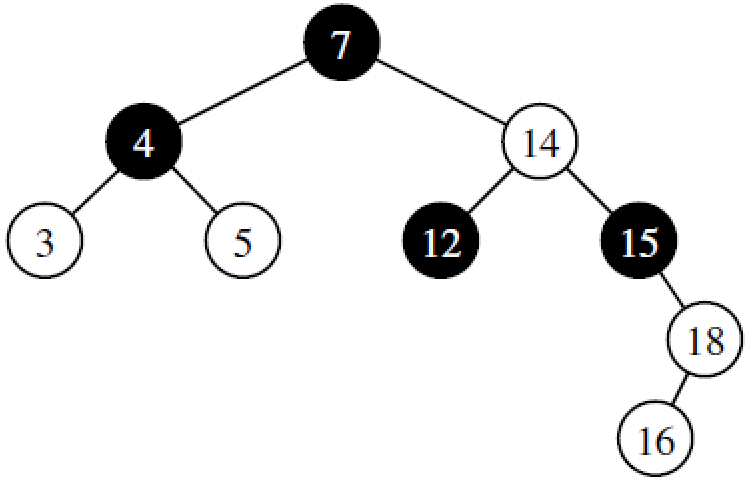


(k)

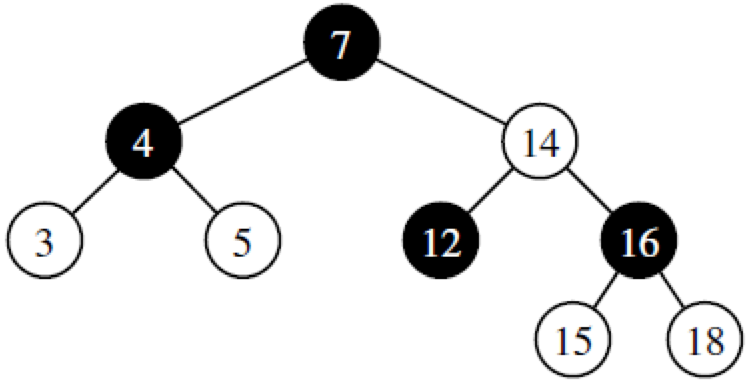


(l)

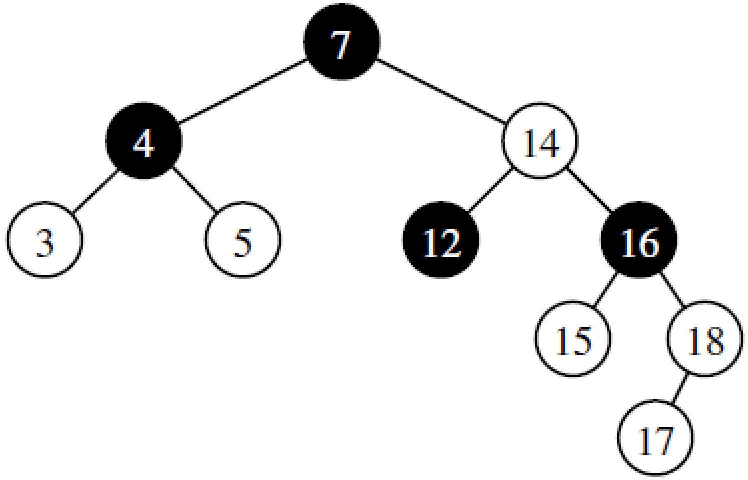
RED-BLACK TREES INSERTION



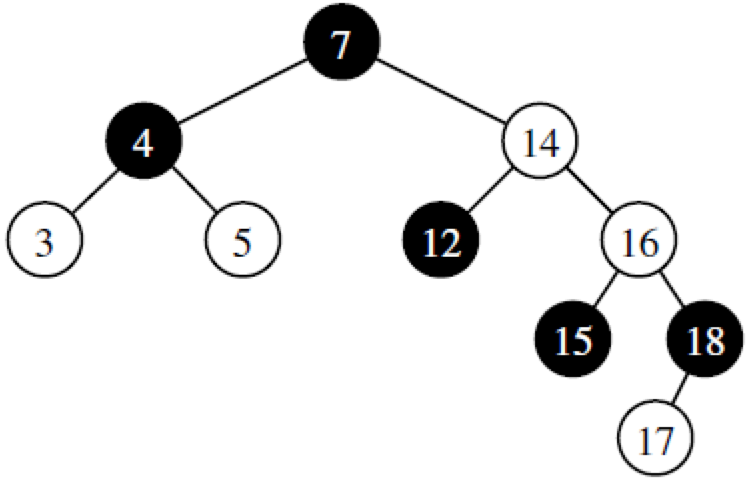
(m)



(n)

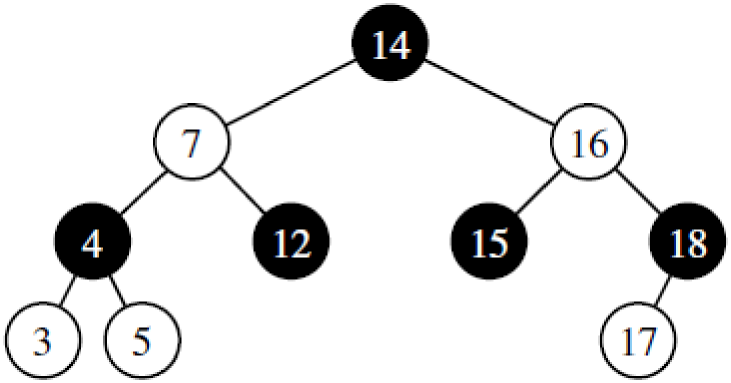


(o)



(p)

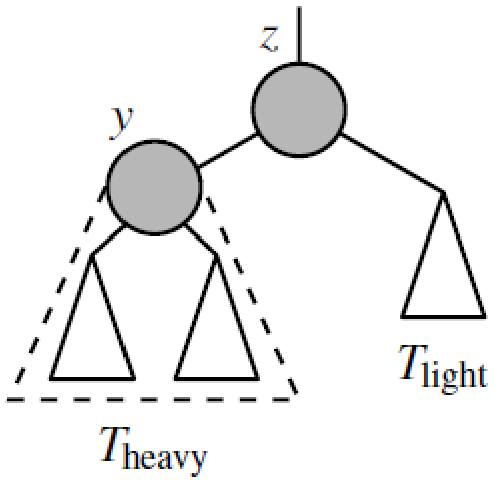
RED-BLACK TREES INSERTION



(q)

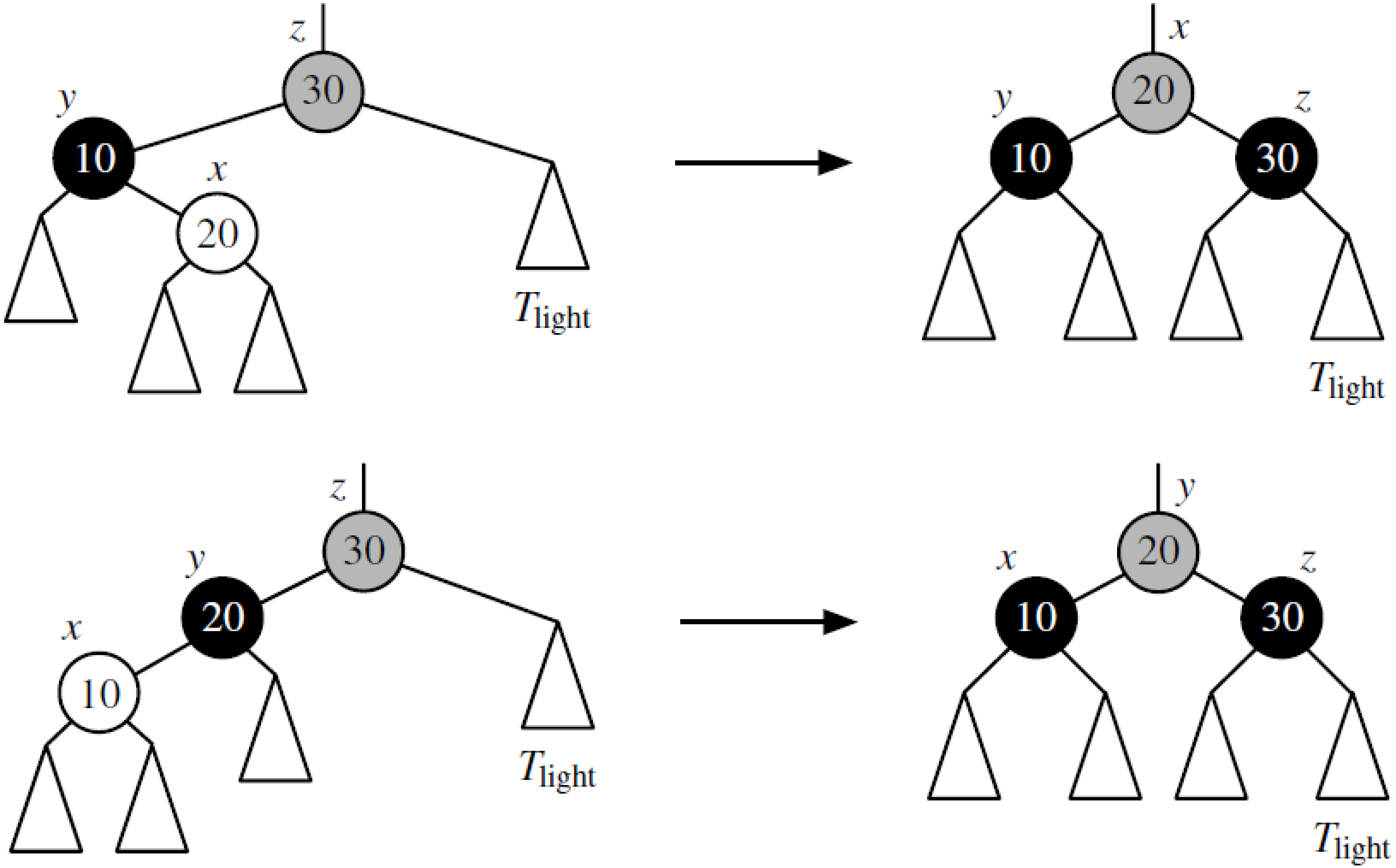
RED-BLACK TREES DELETION

- Search – O(log n)
- If removed node is red – no affect on the black depth property, or red violations
- If removed node is black and has one child that was a red leaf
    - Recolour solves the problem
- If removed node is a black leaf
    - Black deficit of 1
    - Removed node must have a sibling whose subtree has black height 1
    - More general setting with a node z with two subtrees: T heavy and T light. Black depth of T heavy is one more than T light
    - Z: parent of removed leaf
    - Y: root of T heavy



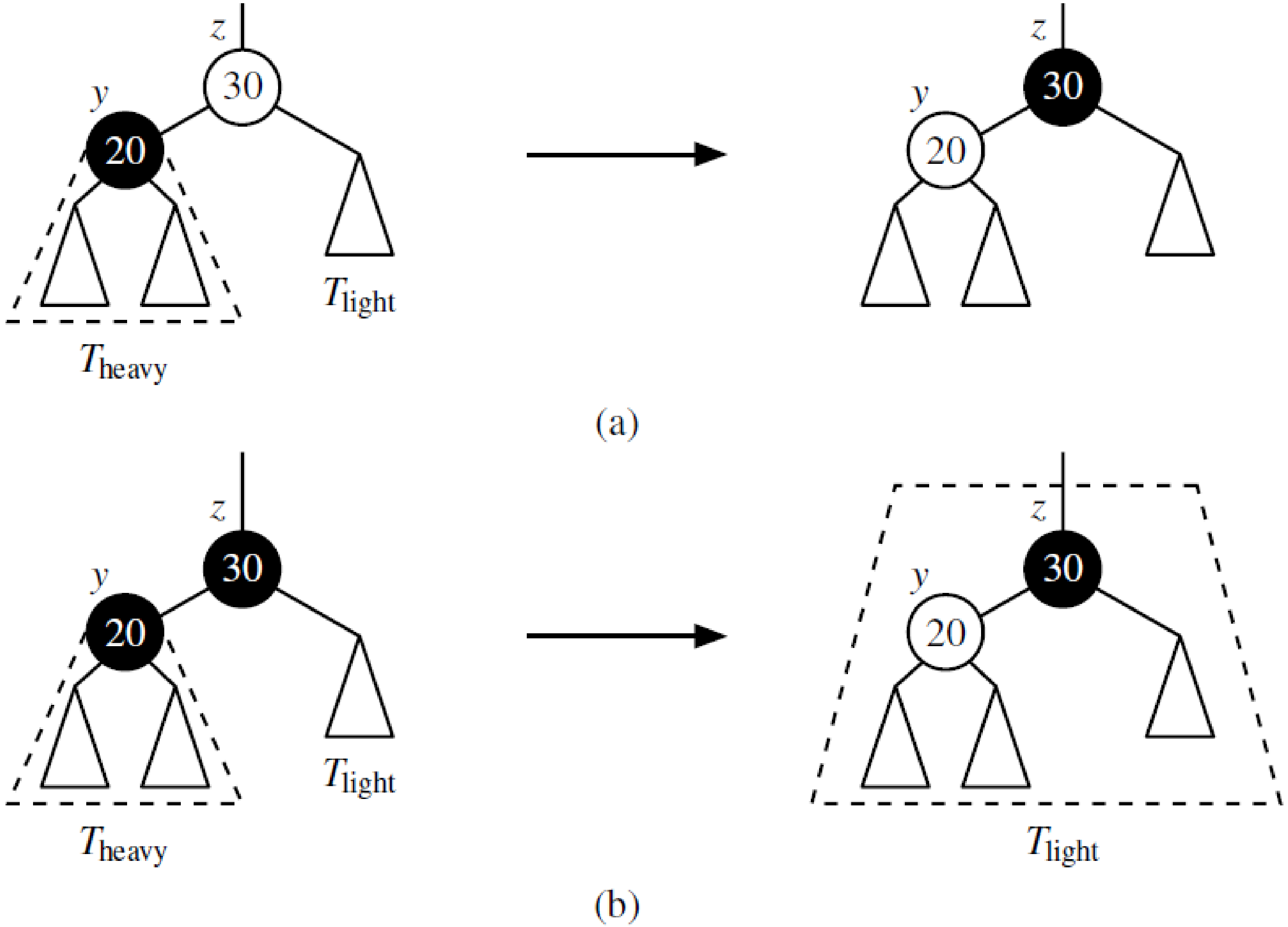
RED-BLACK TREES DELETION

- Case 1: node y is black and has a red child x
- Trinode restructuring:
- x, y and z
- a, b and c
- Make b the parent of the other two
- Colour a and c black
- Give b the previous colour of z



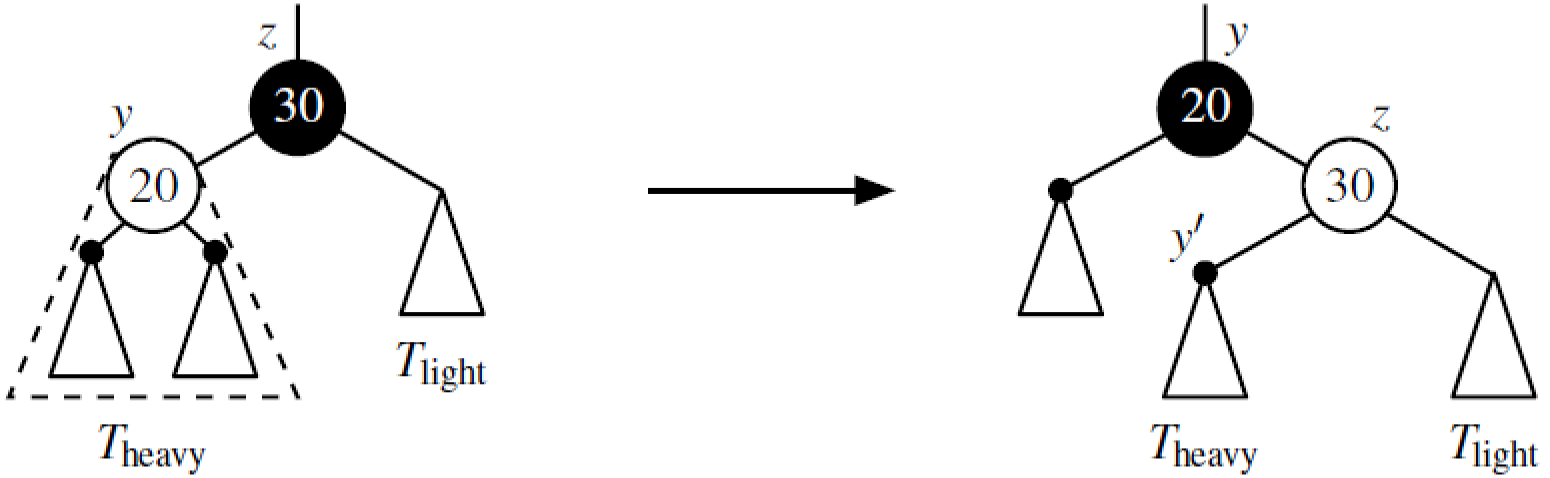
RED-BLACK TREES DELETION

- Case 2: node y is black and both children of y are black(or None)
- Recolouring: colour y red and if z is red, colour it black
- Z becomes deficient, repeat consideration of all three cases at the parent of z as a remedy

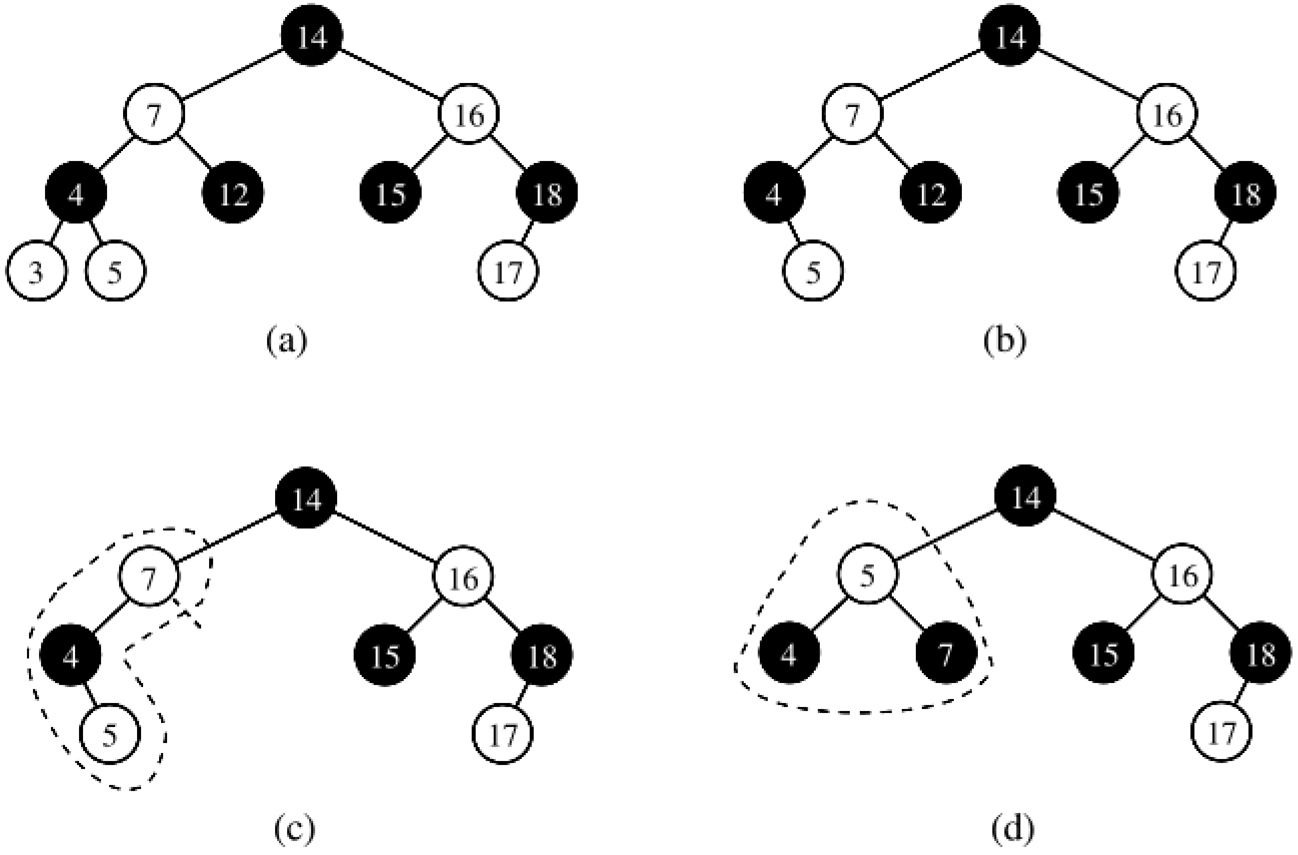


RED-BLACK TREES DELETION

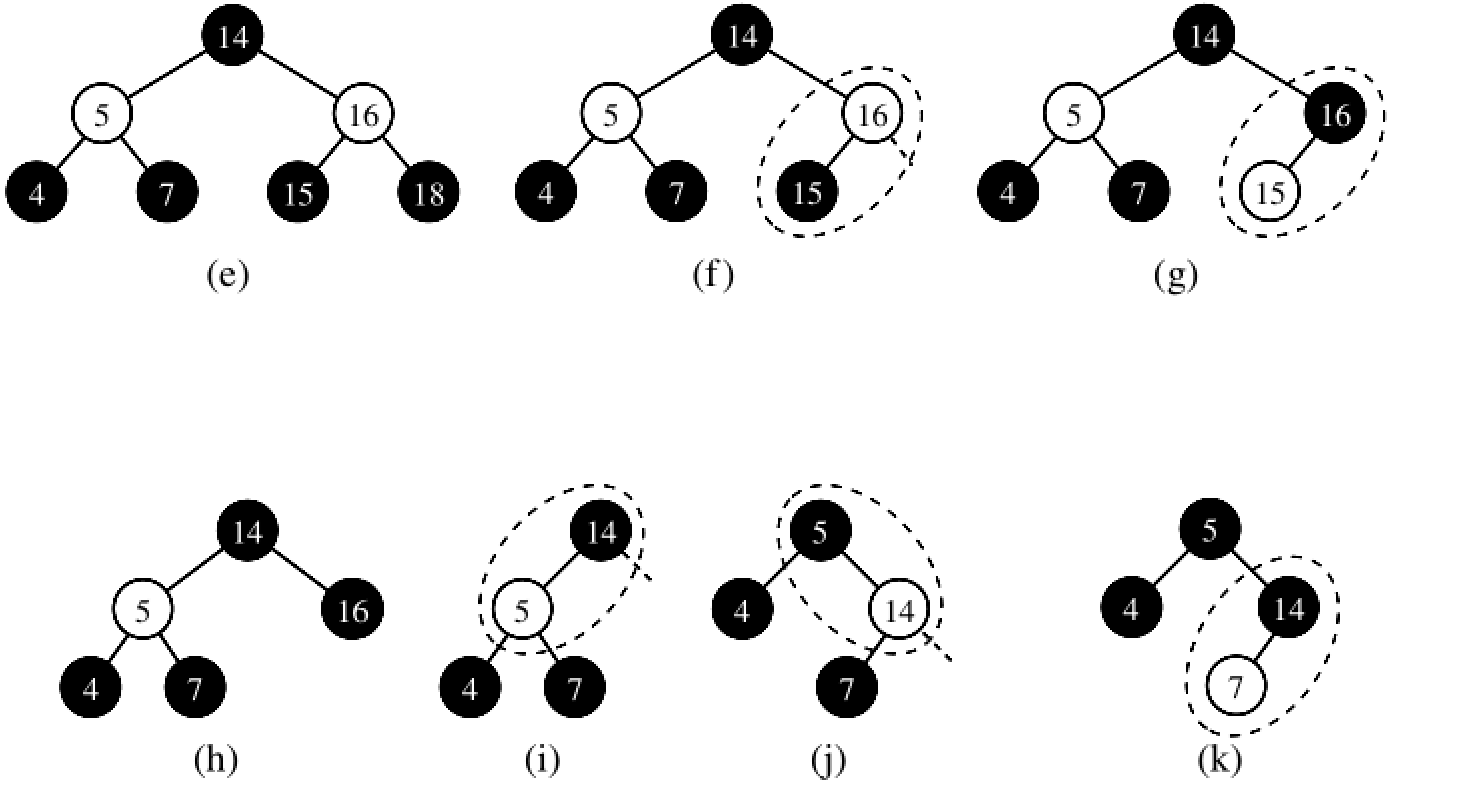
- Case 3: node y is red
- Rotation about y and z
- Recolor y black and z red
- Repeat step 1, 2 and 3 if necessary



RED-BLACK TREES DELETION



RED-BLACK TREES DELETION



RED BLACK TREES PERFORMANCE

- Identical to AVL tree
  - Height of a red-black tree is O (log n)
  - Insertion requires O(log n) recolourings and at most one trinode restructuring
  - Deletion requires O(log n) reclourings and at most two restructuring operations

Operation	Running Time
k in T	$O(\log n)$
T[k] = v	$O(\log n)$
T.delete(p), del T[k]	$O(\log n)$
T.find_position(k)	$O(\log n)$
T.first( ), T.last( ), T.find_min( ), T.find_max( )	$O(\log n)$
T.before(p), T.after(p)	$O(\log n)$
T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k)	$O(\log n)$
T.find_range(start, stop)	$O(s+\log n)$
iter(T), reversed(T)	$O(n)$

PYTHON IMPLEMENTATION

```
class RedBlackTreeMap(TreeMap):
  """Sorted map implementation using a red-black tree."""
  class _Node(TreeMap._Node):
    """Node class for red-black tree maintains bit that denotes color."""
    __slots__ = '_red'      # add additional data member to the Node class

    def __init__(self, element, parent=None, left=None, right=None):
      super().__init__(element, parent, left, right)
      self._red = True      # new node red by default
```

```
  #------------------------ positional-based utility methods ------------
  # we consider a nonexistent child to be trivially black
  def _set_red(self, p): p._node._red = True
  def _set_black(self, p): p._node._red = False
  def _set_color(self, p, make_red): p._node._red = make_red
  def _is_red(self, p): return p is not None and p._node._red
  def _is_red_leaf(self, p): return self._is_red(p) and self.is_leaf(p

  def _get_red_child(self, p):
    """Return a red child of p (or None if no such child)."""
    for child in (self.left(p), self.right(p)):
      if self._is_red(child):
        return child
    return None
```

PYTHON IMPLEMENTATION

```
def _rebalance_insert(self, p):
  self._resolve_red(p)                          # new node is always red

def _resolve_red(self, p):
  if self.is_root(p):
    self._set_black(p)                          # make root black
  else:
    parent = self.parent(p)
    if self._is_red(parent):                    # double red problem
      uncle = self.sibling(parent)
      if not self._is_red(uncle):               # Case 1: misshapen 4-node
        middle = self._restructure(p)           # do trinode restructuring
        self._set_black(middle)                 # and then fix colors
        self._set_red(self.left(middle))
        self._set_red(self.right(middle))
      else:                                     # Case 2: overfull 5-node
        grand = self.parent(parent)
        self._set_red(grand)                    # grandparent becomes red
        self._set_black(self.left(grand))       # its children become black
        self._set_black(self.right(grand))
        self._resolve_red(grand)                # recur at red grandparent
```

```
def _rebalance_delete(self, p):
  if len(self) == 1:
    self._set_black(self.root())                # special case: ensure that root
  elif p is not None:
    n = self.num_children(p)
    if n == 1:                                  # deficit exists unless child is a
      c = next(self.children(p))
      if not self._is_red_leaf(c):
        self._fix_deficit(p, c)
    elif n == 2:                                # removed black node with red
      if self._is_red_leaf(self.left(p)):
        self._set_black(self.left(p))
      else:
        self._set_black(self.right(p))
```

PYTHON IMPLEMENTATION

```python
def _fix_deficit(self, z, y):
    """Resolve black deficit at z, where y is the root of z's heavier subtree."""
    if not self._is_red(y): # y is black; will apply Case 1 or 2
        x = self._get_red_child(y)
        if x is not None: # Case 1: y is black and has red child x; do "transfer"
            old_color = self._is_red(z)
            middle = self._restructure(x)
            self._set_color(middle, old_color)          # middle gets old color of z
            self._set_black(self.left(middle))          # children become black
            self._set_black(self.right(middle))
        else: # Case 2: y is black, but no red children; recolor as "fusion"
            self._set_red(y)
            if self._is_red(z):
                self._set_black(z)                      # this resolves the problem
            elif not self.is_root(z):
                self._fix_deficit(self.parent(z), self.sibling(z)) # recur upward
    else: # Case 3: y is red; rotate misaligned 3-node and repeat
        self._rotate(y)
        self._set_black(y)
        self._set_red(z)
        if z == self.right(y):
            self._fix_deficit(z, self.left(z))
        else:
            self._fix_deficit(z, self.right(z))
```

PREVIOUSLY ON BINARY SEARCH TREES

- Binary Search Tree
  - Performance: O(h)
- Balancing search tree
  - Rotation
    - X-Y rotation
    - Trinode rotation
- AVL tree
  - Height of AVL tree: number of nodes in a path
  - Height balance property
- Splay tree
  - Splay operations: search, add, remove

NAVIGATING A BINARY SEARCH TREE

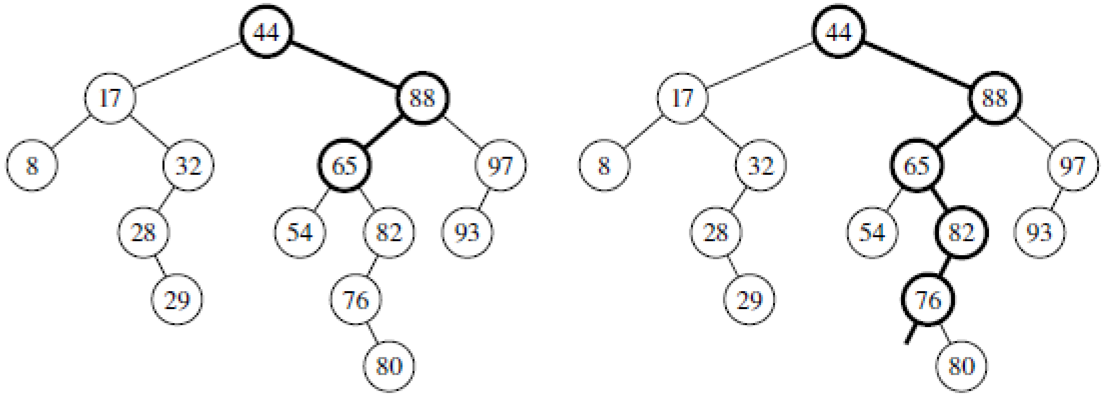
- In order traversal of a binary search tree visits positions in increasing order of their keys
- Proof by induction
  - Base: tree has one item
  - Inductive: recursive inorder traversal: left child(ren) -> node -> right child(ren), by binary search tree property, inorder traversal visits positions in increasing order
- Inorder traversal: O(n) => sorted iteration of the keys of a map in O(n), provided that the map is represented as a binary search tree

BINARY SEARCH TREE ADT

- first(): returns the position containing the least key, or None if the tree is empty
- last(): returns the position containing the greatest key, or None if empty tree
- before(p): returns the position containing the greatest key that is less than that of position p, or None if p is the first position
- after(p): returns position containing the least key that is greater than that of the position p, or None if p is the last position

SEARCHES

- Locate a particular key by viewing it as a decision tree
- At each position p: is the desired k less than, equal to, or greater than the key stored at position p?

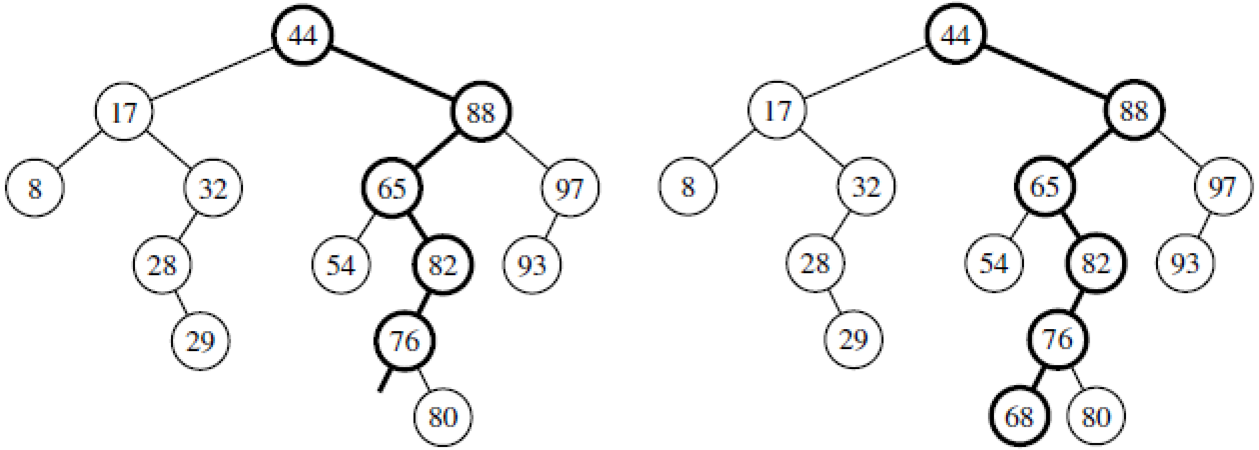



```
Algorithm TreeSearch(T, p, k):
    if k == p.key() then
        return p
    else if k < p.key() and T.left(p) is not None then
        return TreeSearch(T, T.left(p), k)
    else if k > p.key() and T.right(p) is not None then
        return TreeSearch(T, T.right(p), k)
    return p
```

INSERTIONS

- Map backed by a binary search tree
- M[k] = v
  - Search for key k
  - If found, update value
  - Otherwise, create a new node and insert it into the binary search tree
- E.g. insert 68 into tree

Algorithm TreeInsert(T, k, v):
    Input: A search key k to be associated with value v
    p = TreeSearch(T, T.root(), k)
    if k == p.key() **then**
        Set p's value to v
    else if k < p.key() **then**
        add node with item (k,v) as left child of p
    else
        add node with item (k,v) as right child of p



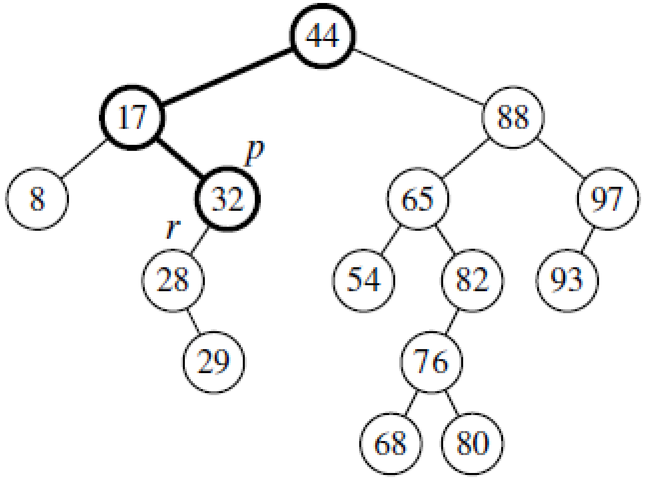
DELETION

- Find the position p of T storing an item with key equal to k, if the search is successful:

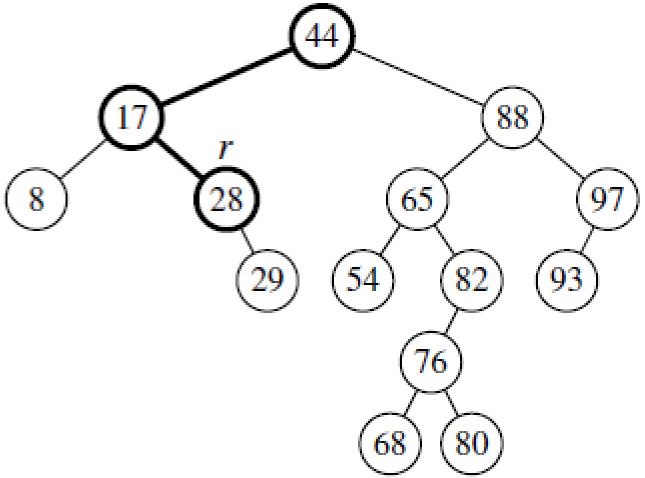
1. If p has at most one child, then delete p, replace it with the child
2. If p has two children
   - Locate position r, where r = before(p). r is the rightmost position of the left subtree of p
   - Use r's item as a replacement for position p
   - Delete node at r, since r has at most 1 child, repeat step 1 for r

DELETION

- Delete item with k=32 with one child r



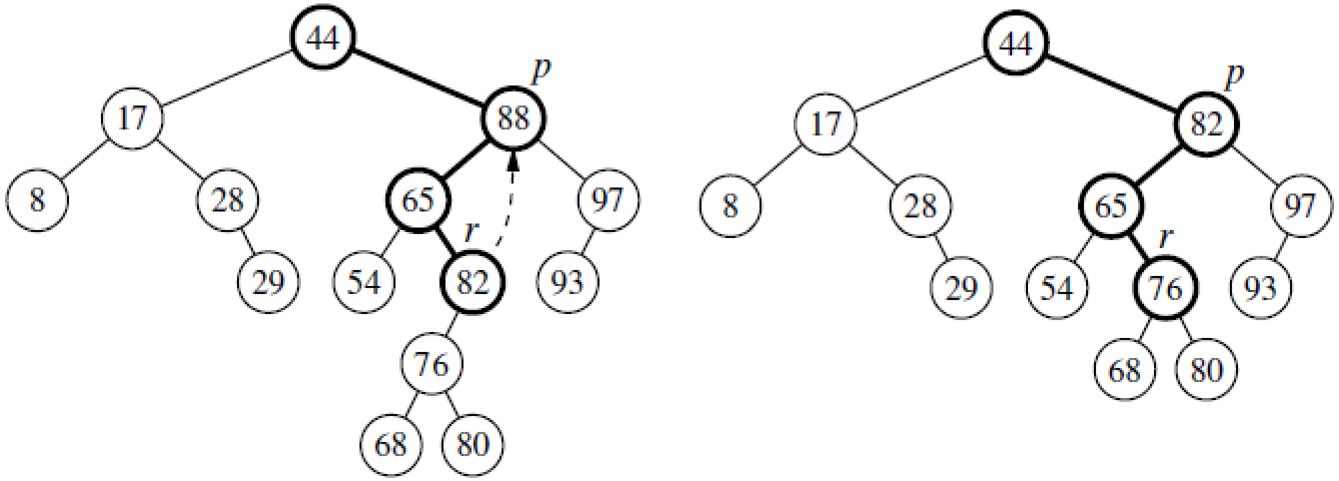
(a)



(b)

DELETION

- Delete item with k==88



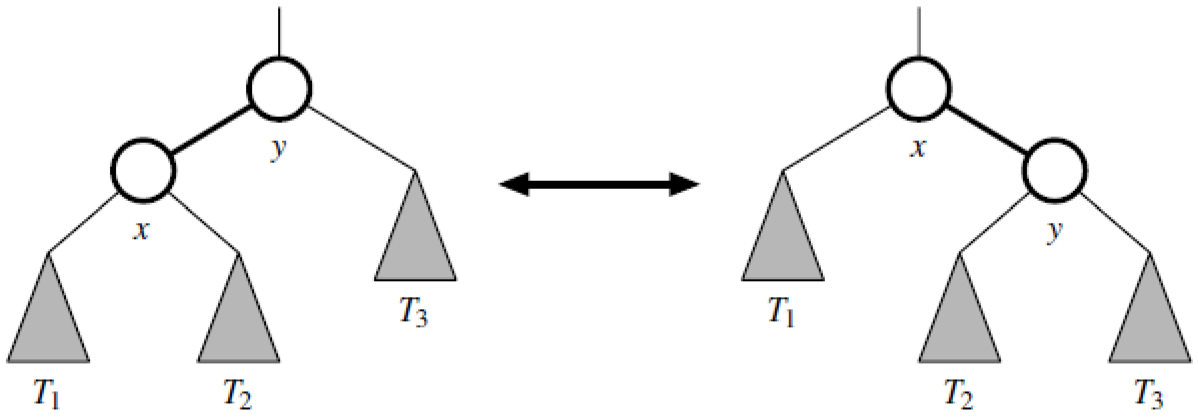
PERFORMANCE OF A BINARY SEARCH TREE

- Almost all operations have a worst-case running time of O(h)
- Single call to after() is worst case O(h), n successive calls made during a call to __iter__ require a total of O(n) time" each edge is traced at most twice
- O(1) amortised time bounds
- Is O(h) same as O(log n)?
- No, BST can be unbalanced

Operation	Running Time
k in T	$O(h)$
T[k], T[k] = v	$O(h)$
T.delete(p), del T[k]	$O(h)$
T.find_position(k)	$O(h)$
T.first(), T.last(), T.find_min(), T.find_max()	$O(h)$
T.before(p), T.after(p)	$O(h)$
T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k)	$O(h)$
T.find_range(start, stop)	$O(s+h)$
iter(T), reversed(T)	$O(n)$

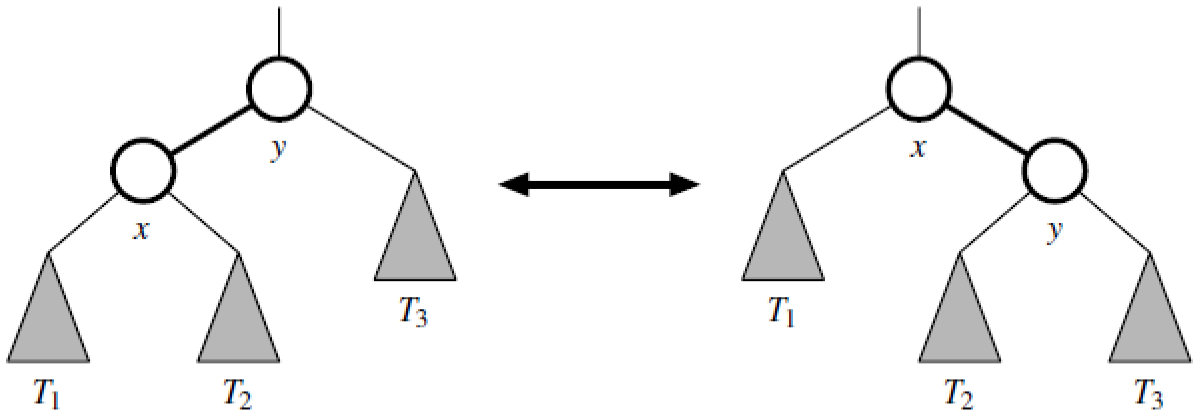
BALANCED SEARCH TREES

- Balanced binary search tree: O(log n) time for basic map operations
- What about the running time of operations after some sequence of operations?
  - O(n)
  - Why?
- Balanced Search Trees: stronger performance guarantees
- Main idea: rotation



BALANCED SEARCH TREES

- Single rotation: a constant number of parent-child relationships are modified
  - O(1) for linked binary with a linked binary tree representation
- Rotations allow the shape of a tree to be modified while maintaining the search tree property
  - Rightward rotation: depth of each node in T1 reduced by 1, depth of each node in T3 increased by 1
- One or more rotation: trinode restructuring



BALANCED SEARCH TREES

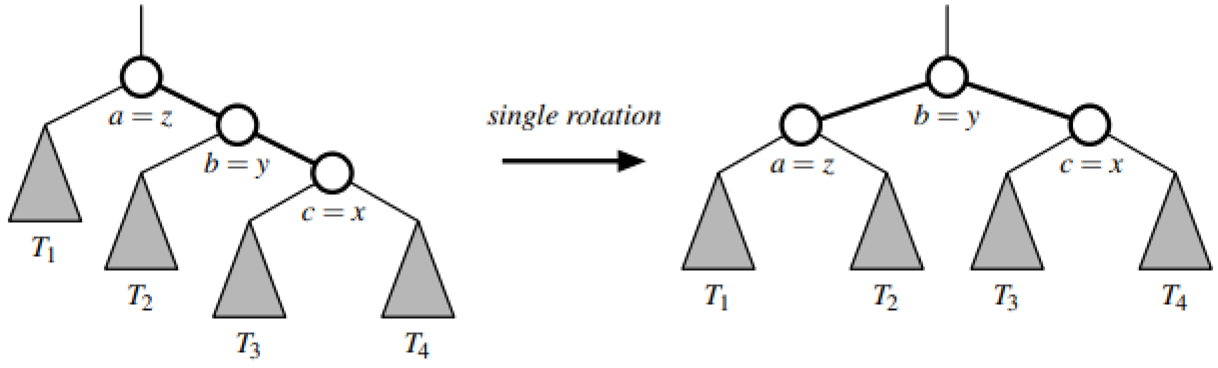
Algorithm restructure(x):

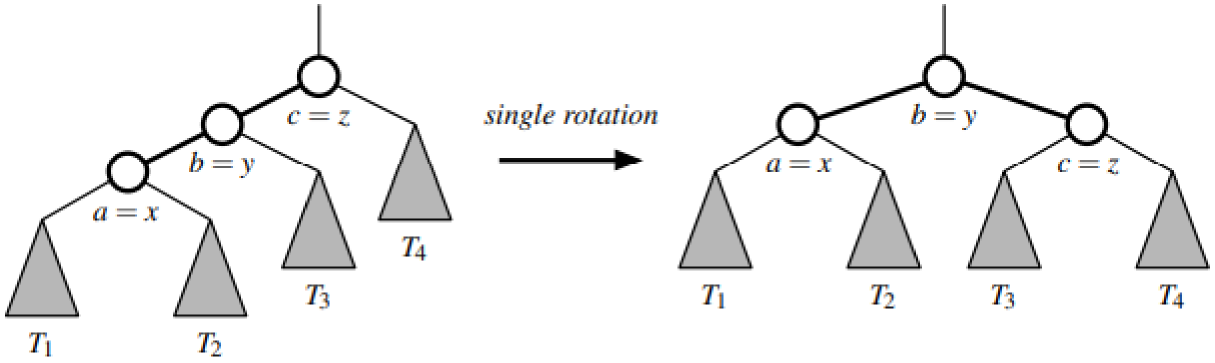
Input: A position x of a binary search tree T that has both a parent y and a grandparent z

Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving positions x, y, and z

1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.
2: Replace the subtree rooted at z with a new subtree rooted at b.
3: Let a be the left child of b and let T_1 and T_2 be the left and right subtrees of a, respectively.
4: Let c be the right child of b and let T_3 and T_4 be the left and right subtrees of c, respectively.

BALANCED SEARCH TREES





Algorithm restructure(x):

Input: A position x of a binary search tree T that has both a parent y and a grandparent z

Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving positions x, y, and z

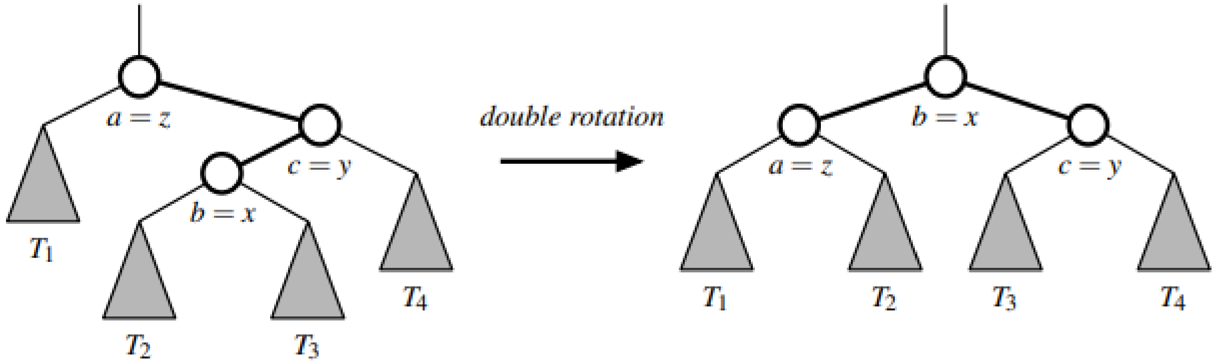
1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

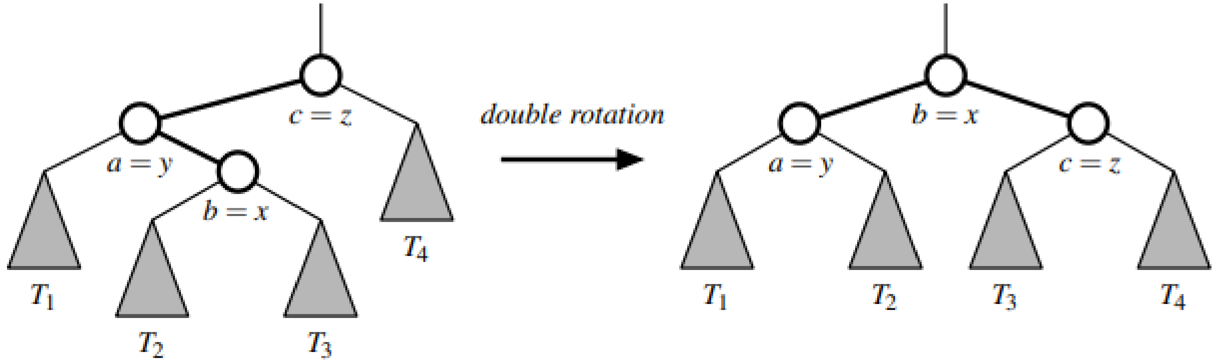
2: Replace the subtree rooted at z with a new subtree rooted at b.

3: Let a be the left child of b and let T_1 and T_2 be the left and right subtrees of a, respectively.

4: Let c be the right child of b and let T_3 and T_4 be the left and right subtrees of c, respectively.

BALANCED SEARCH TREES





Algorithm restructure(x):

Input: A position x of a binary search tree T that has both a parent y and a grandparent z

Output: Tree T after a trinode restructuring (which corresponds to a single or double rotation) involving positions x, y, and z

1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

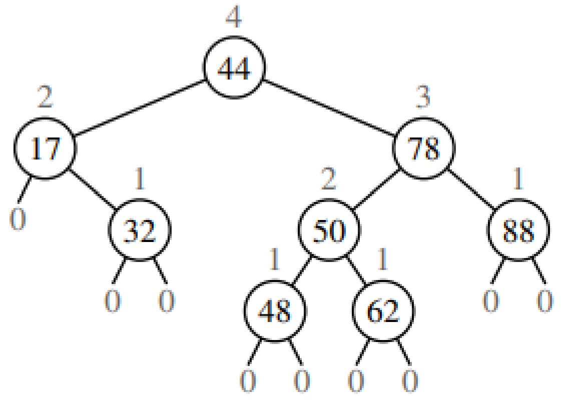
2: Replace the subtree rooted at z with a new subtree rooted at b.

3: Let a be the left child of b and let T_1 and T_2 be the left and right subtrees of a, respectively.

4: Let c be the right child of b and let T_3 and T_4 be the left and right subtrees of c, respectively.

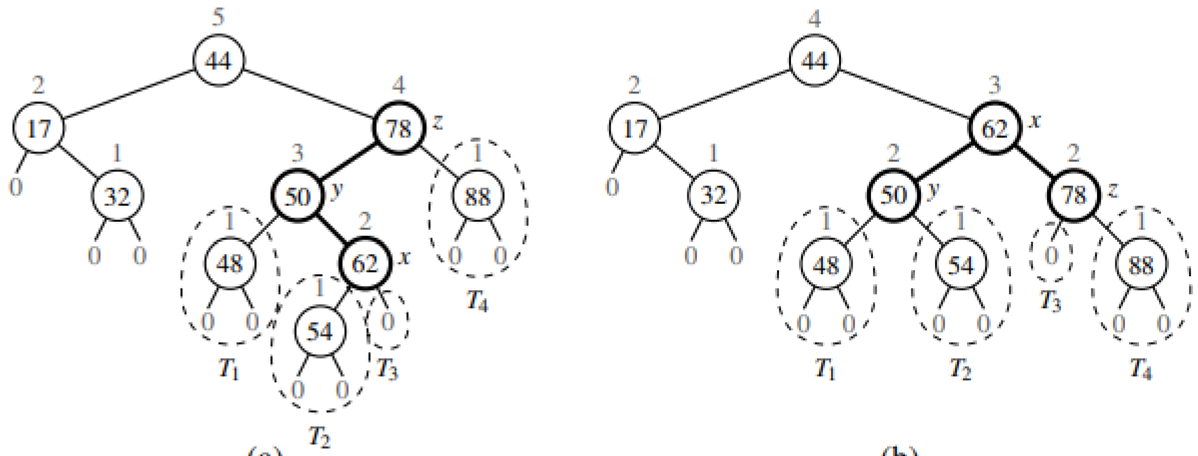
AVL TREES

- AVL: Adelson-Velsky and Landis
- Adds a rule to the binary search tree to maintain a logarithmic height for the tree
- Height: number of edges on the longest path vs **number of nodes on this longest path**
  - Leaf position has height 1
- **Height balance property**: for every position p of T, the heights of the children of p differ by at most 1
- A subtree of an AVL tree is itself an AVL tree
- The height of an AVL tree storing n entries is O(log n)
  - Proof in the text book



AVL TREES

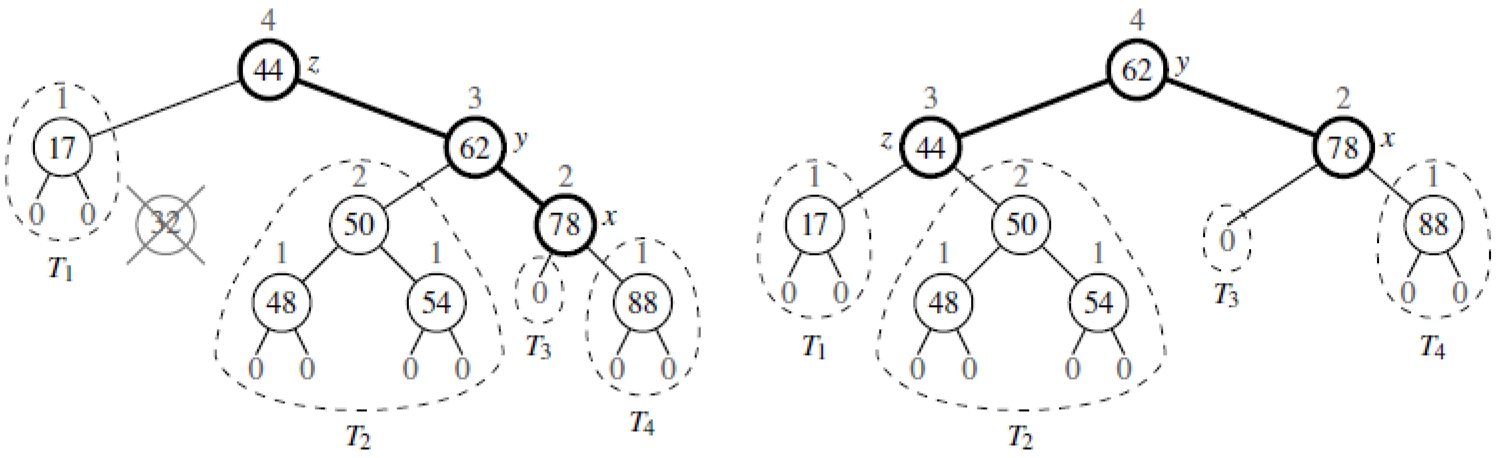
- Insertion: insert item with key 54
- "search and repair": going up from p to the root of T
    - z: first unbalanced position
    - y: child of z with higher height, y must be an ancestor of p
    - x: child of y with higher height (no tie, x must be an ancestor of p)
    - Call the trinode restructuring method, restructure(x)



(a) (b)

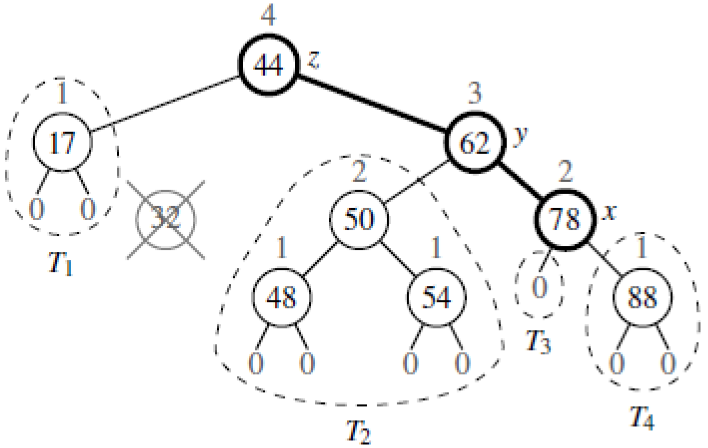
AVL TREES

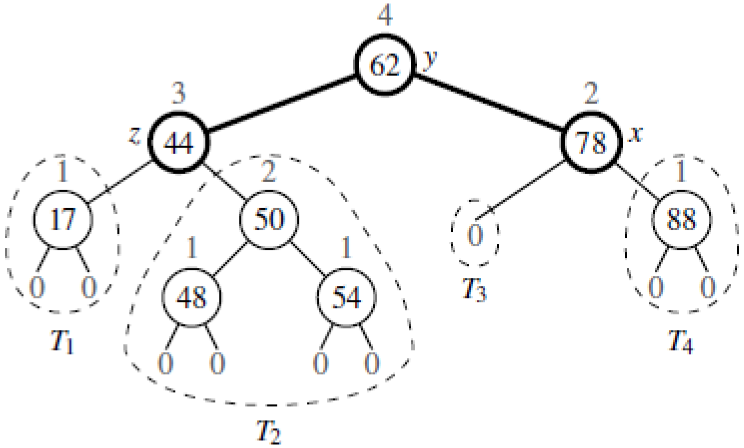
- Deletion: delete item with key 32
- Trinode restructuring:
  - z: first unbalanced position
  - y: child of z with larger height (y not ancestor of p)
  - x: child of y, such that if one the children of y is taller than the other, let x be the taller child of y. Else let x be the child of y on the same side as y
  - Perform restructure(x)
- Is this enough?



AVL TREES

- Deletion: delete item with key 32
- Trinode restructuring:
  - May reduce the height of the subtree rooted at b by 1
  - Causes an ancestor of b to become unbalanced
  - Walk up T looking for unbalanced positions
  - O(log n) trinode restructuring are sufficient

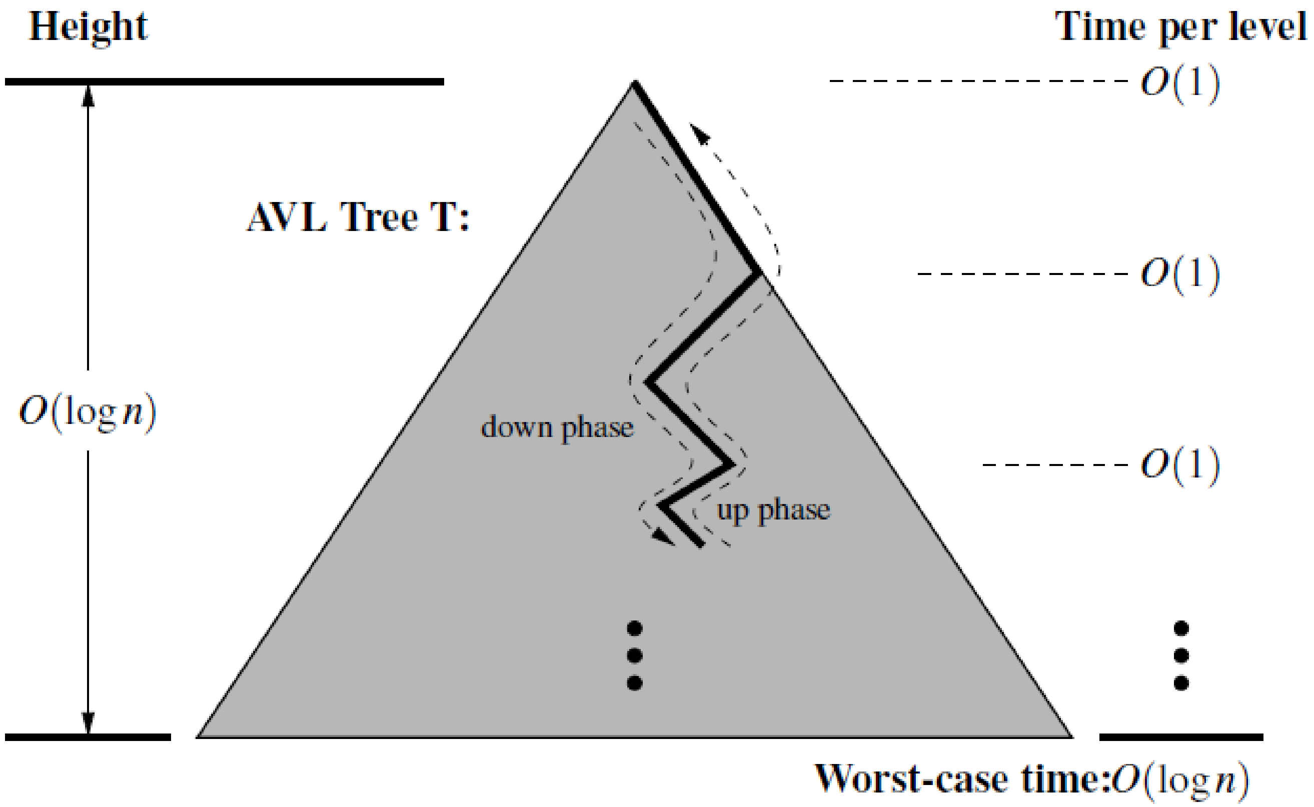




PERFORMANCE OF AVL TREES

- AVL tree with n items: height guaranteed to be O(log n)
- Standard BST operations: bounded by the height of tree
- AVL trees: O(log n) for most of the operations

Operation	Running Time
k in T	$O(\log n)$
T[k] = v	$O(\log n)$
T.delete(p), del T[k]	$O(\log n)$
T.find_position(k)	$O(\log n)$
T.first(), T.last(), T.find_min(), T.find_max()	$O(\log n)$
T.before(p), T.after(p)	$O(\log n)$
T.find_lt(k), T.find_le(k), T.find_gt(k), T.find_ge(k)	$O(\log n)$
T.find_range(start, stop)	$O(s + \log n)$
iter(T), reversed(T)	$O(n)$

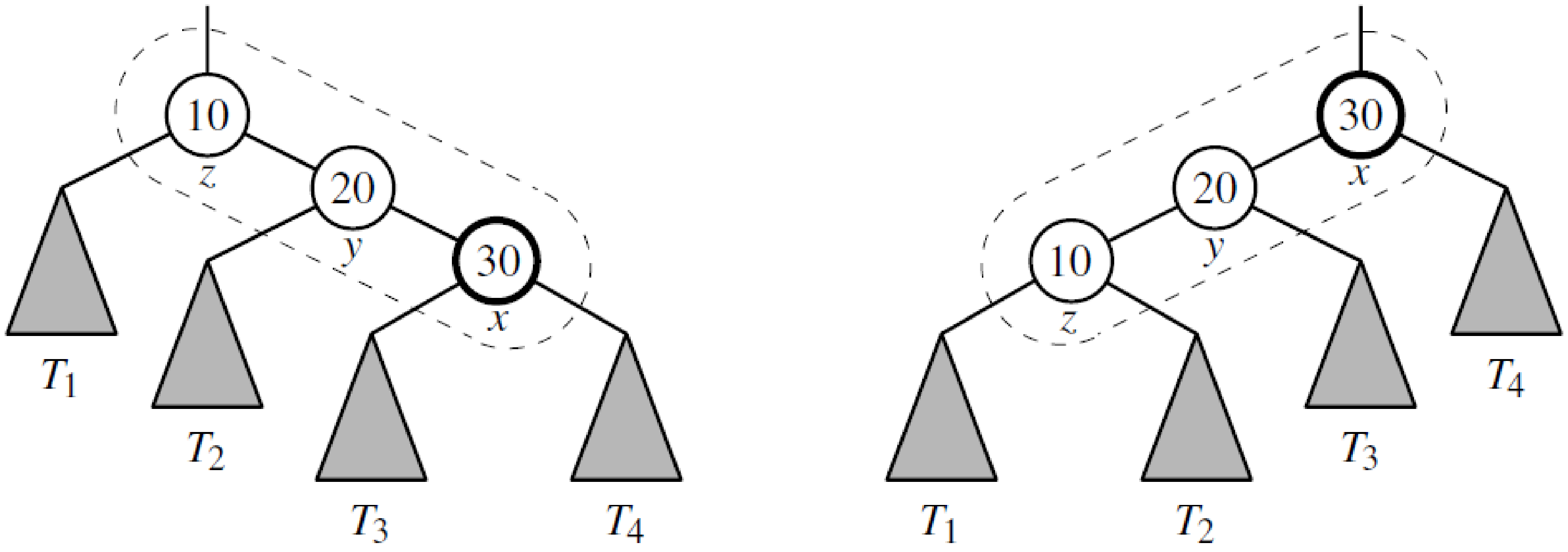


SPLAY TREES(伸展树)

- Splay tree: different from the other balanced search trees
- No strict enforcement on a logarithmic upper bound on the height of the tree
- Efficiency realized by **splaying** operations
  - Performed at the bottommost position p reached for insertion, deletion, and search.
  - Splay operation causes more frequently accessed elements to remain nearer to the root
    - To reduce search times
  - Logarithmic amortised running time

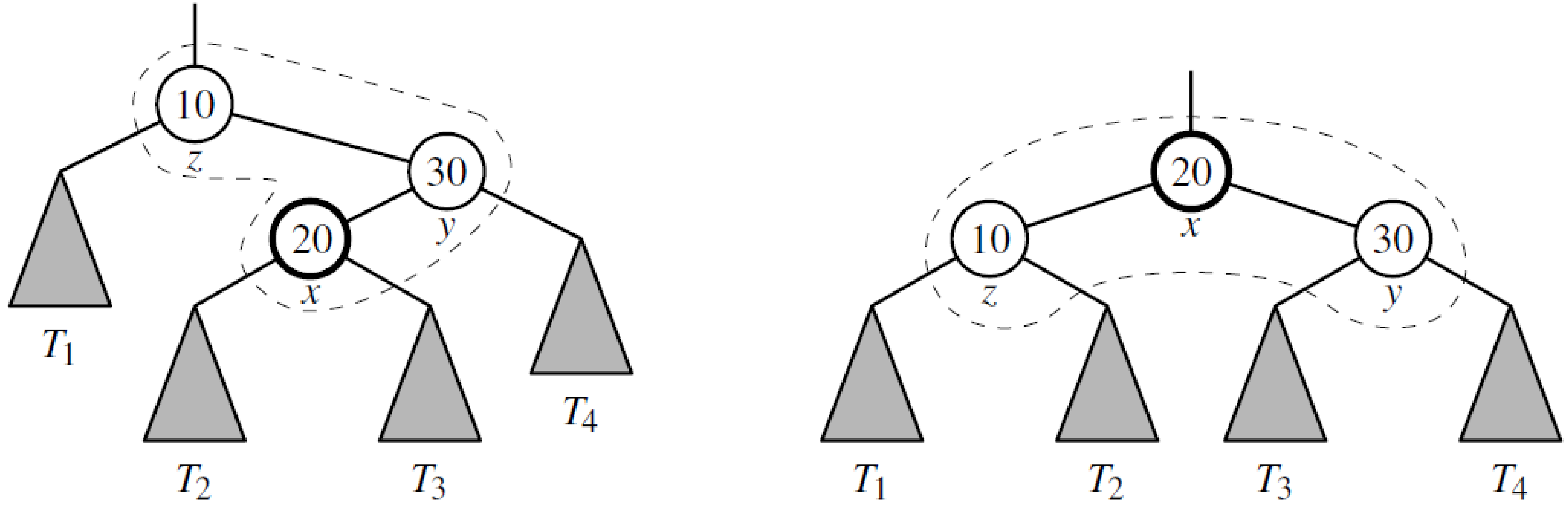
SPLAY TREES(伸展树)

- Splay operations
- Given a node x of a binary search tree T, splay x – moving x to the root of T through a sequence of restructurings
- Zig-zig: node x and its parent y are both left children or both right children



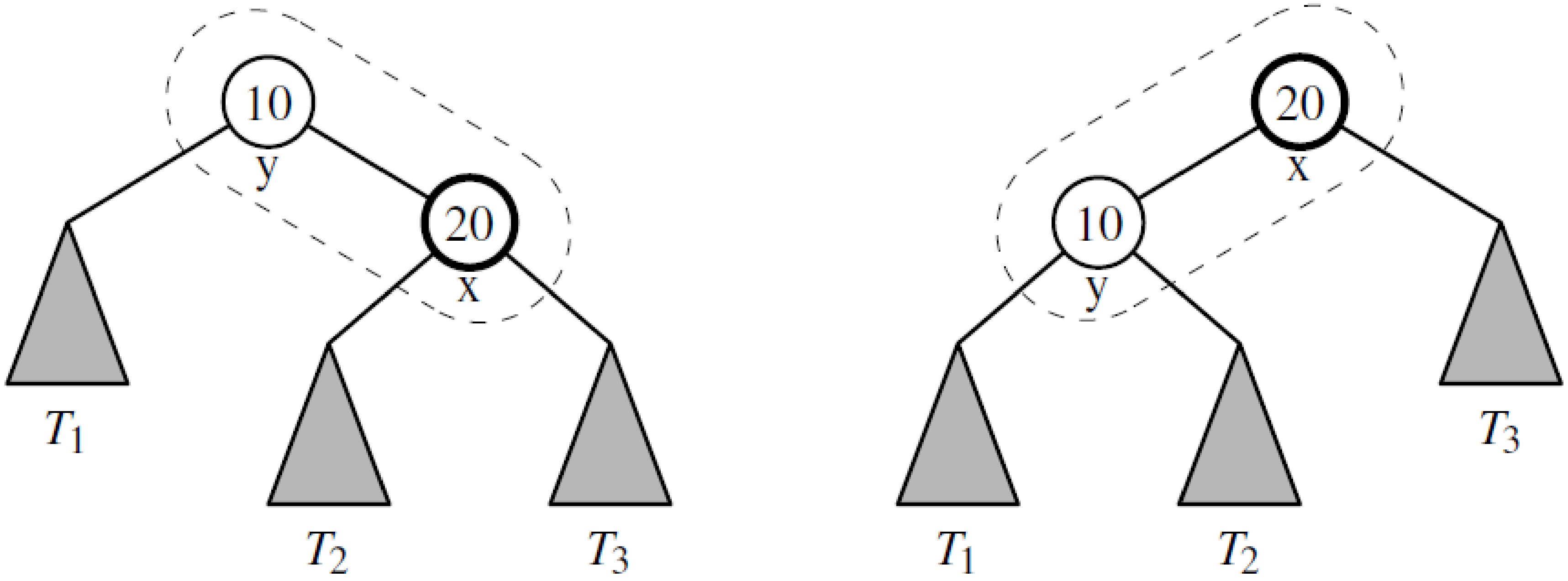
SPLAY TREES(伸展树)

- Zig-zag: one of x and y is a left child and the other is a right child.
- Promote x by making x have y and z as its children, while maintaining the inorder relationships of the nodes in T

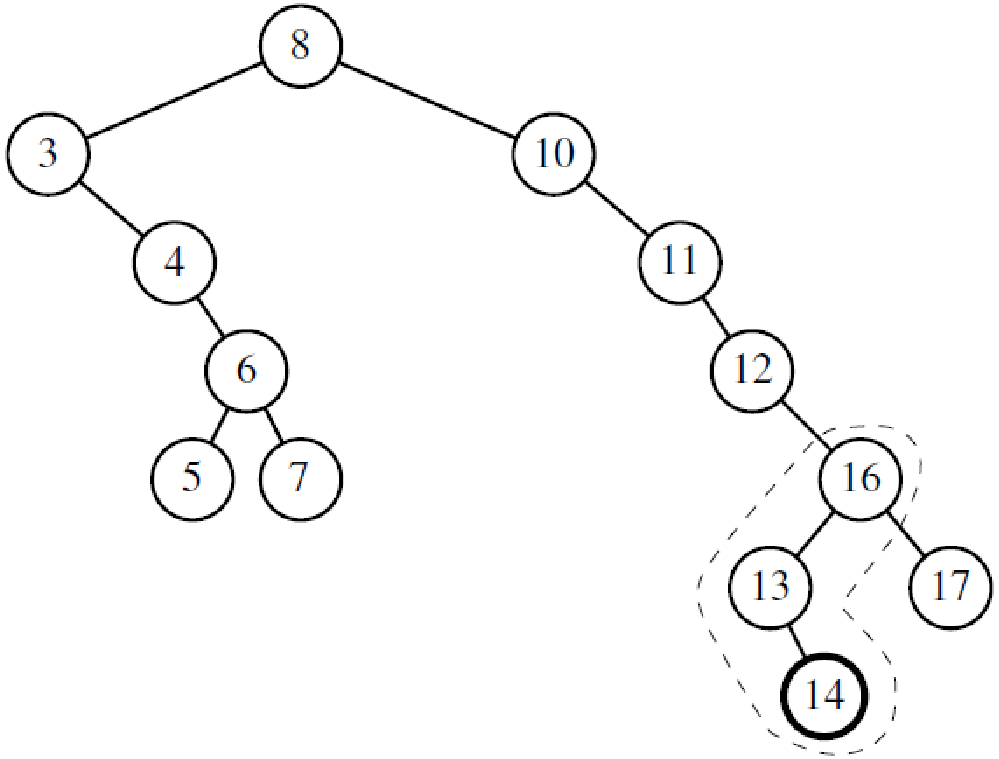


SPLAY TREES(伸展树)

- Zig: x does not have a grandparent
- Perform a single rotation to promote x over y making y a child of x

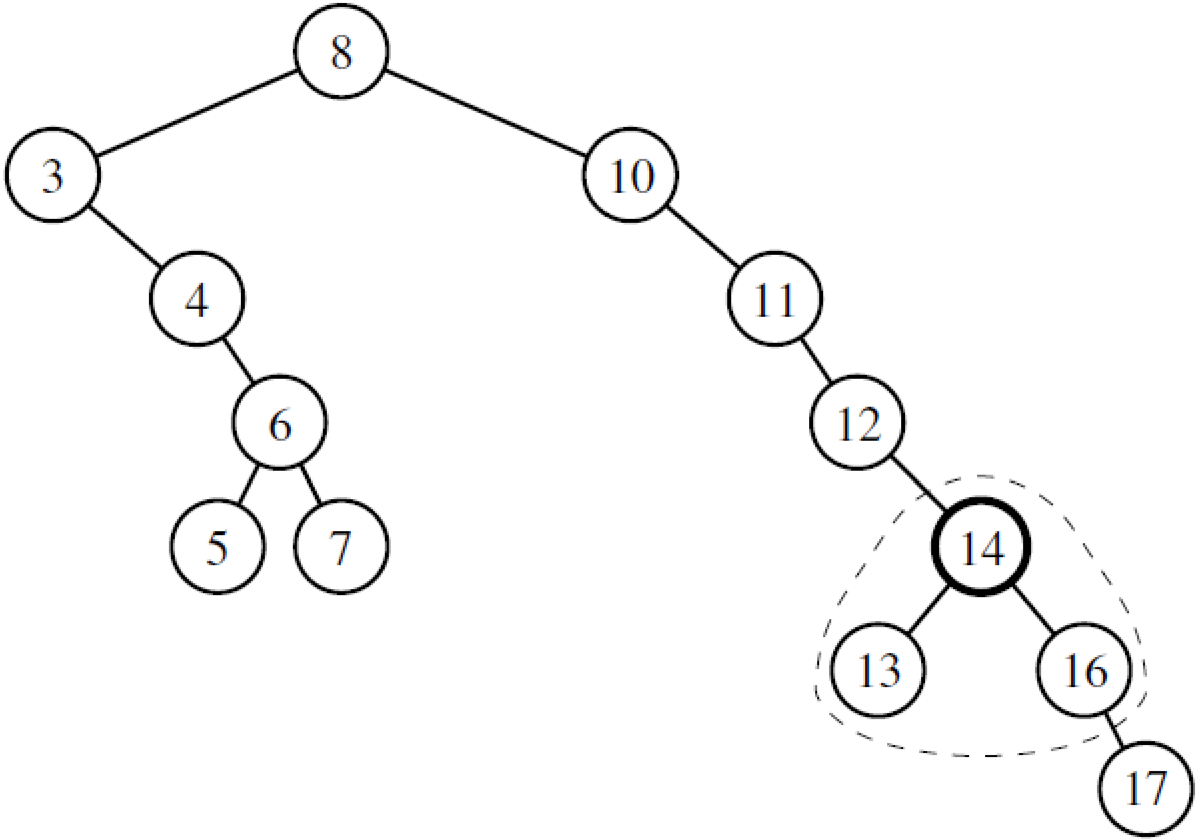


SPLAY TREES(伸展树)



(a)

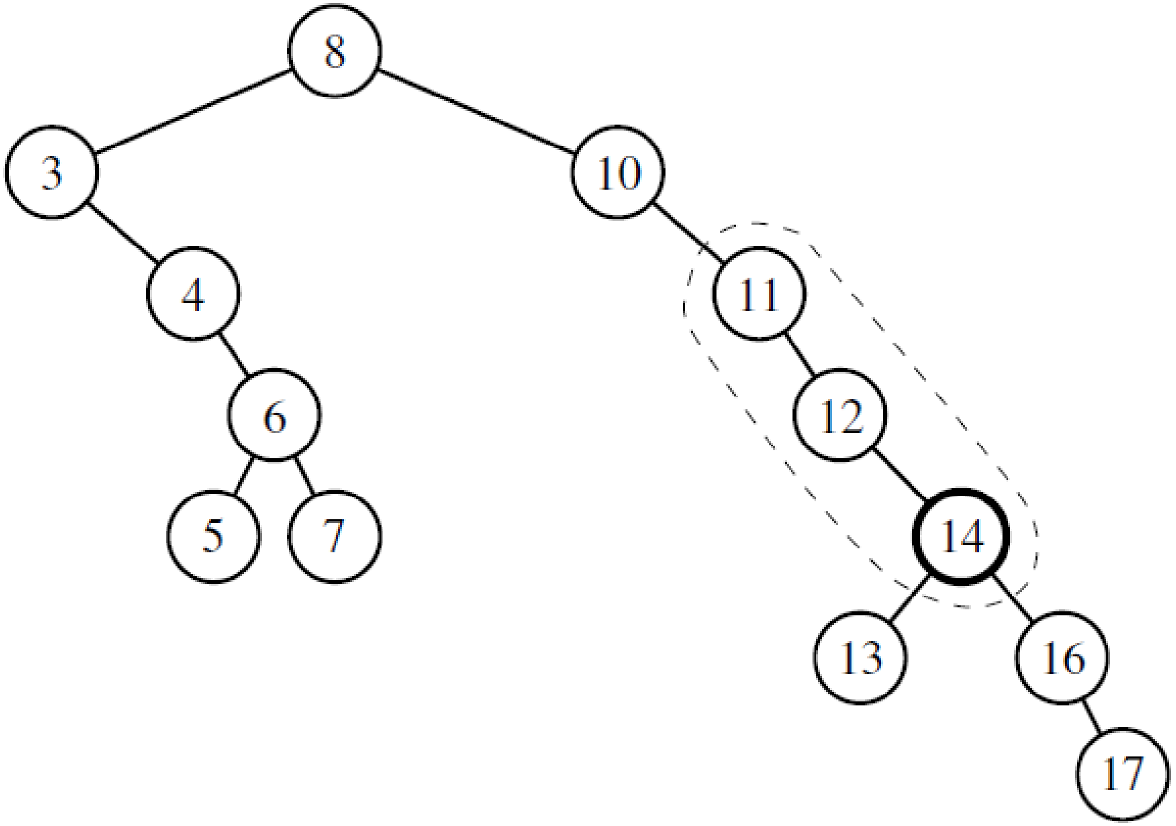
Splaying the node storing 14 with a zig-zag



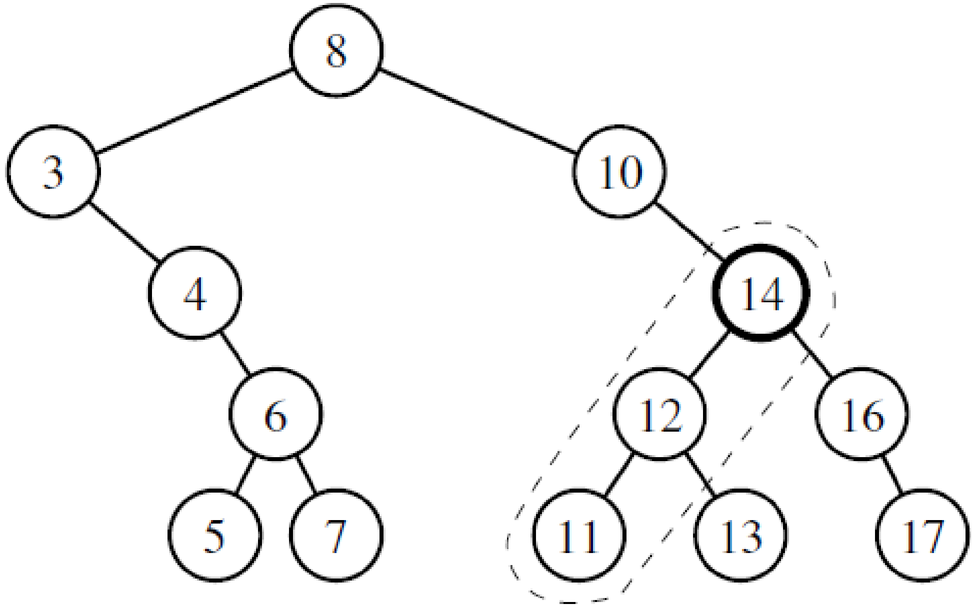
(b)

After the zig-zag

SPLAY TREES(伸展树)



(c)

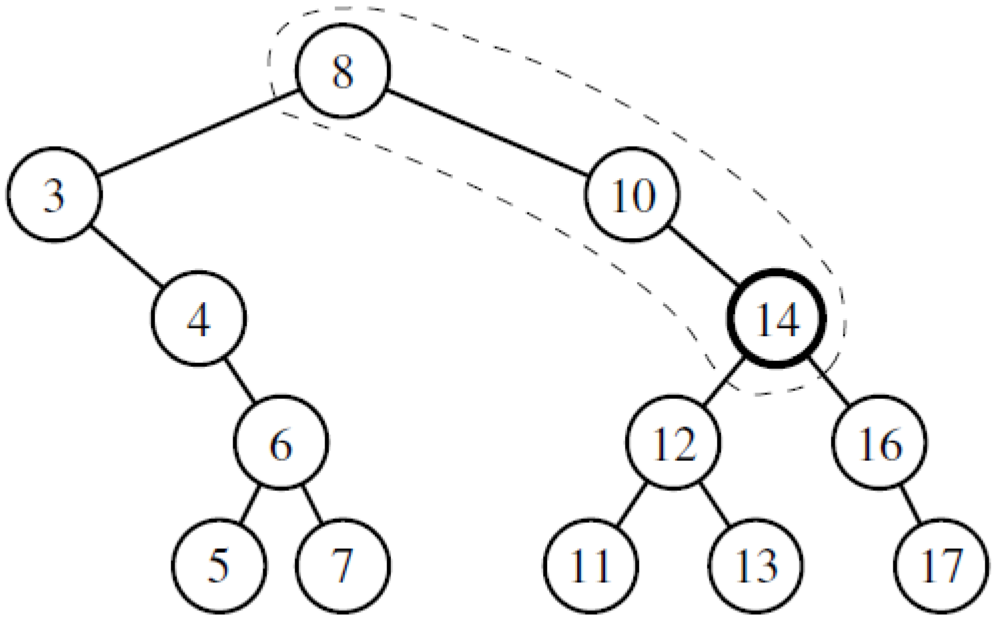


(d)

Zig-zig

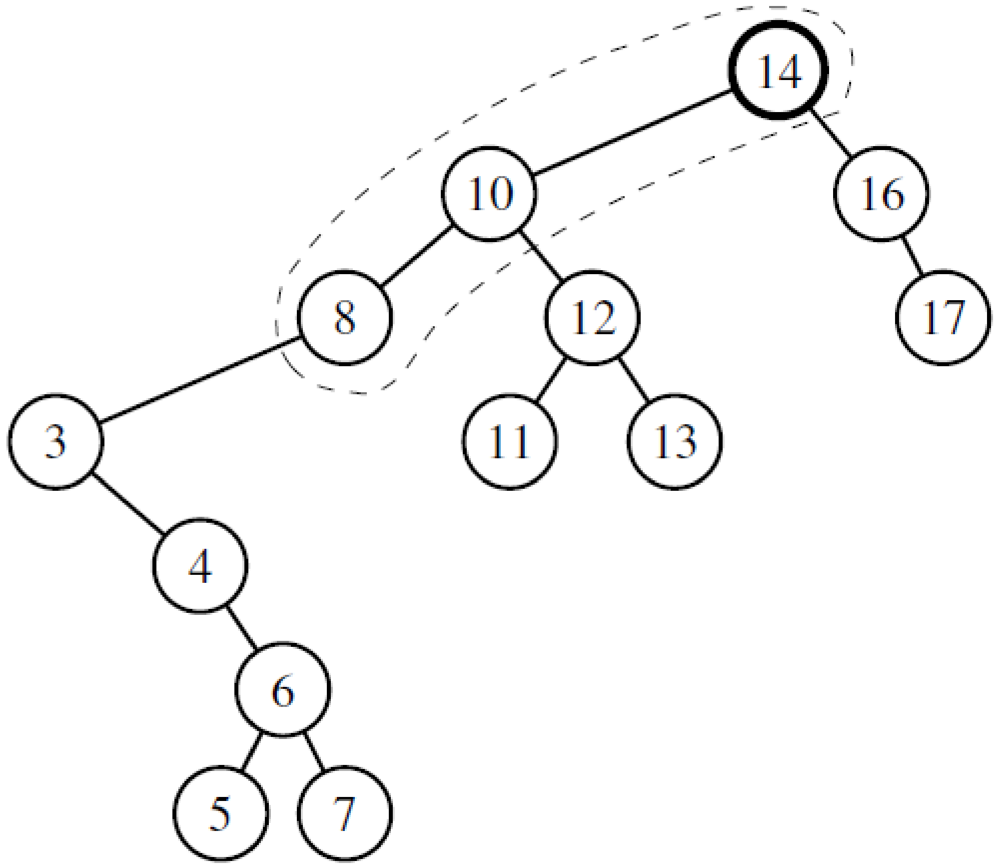
After the zig-zig

SPLAY TREES(伸展树)



(e)

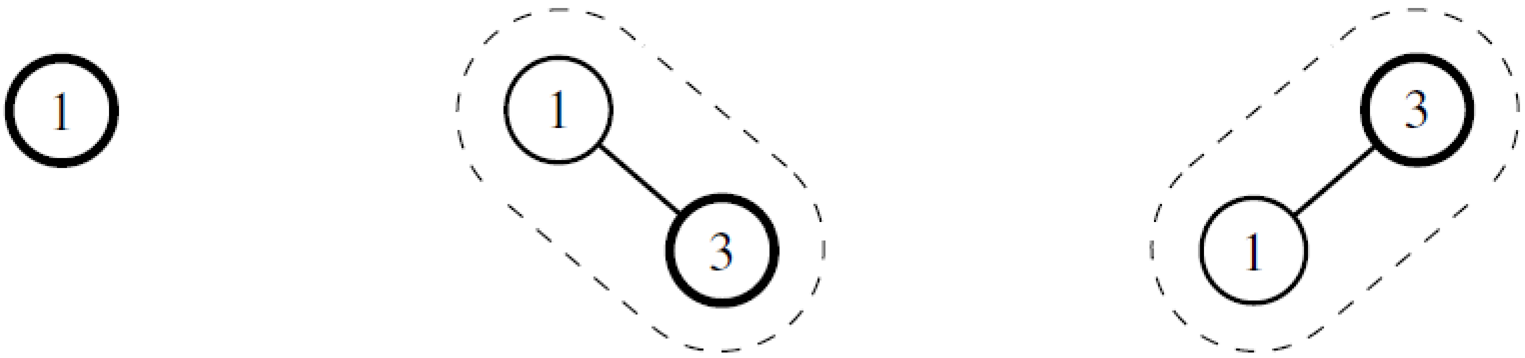
Zig-zig



After zig-zig (f)

SPLAY TREES(伸展树)

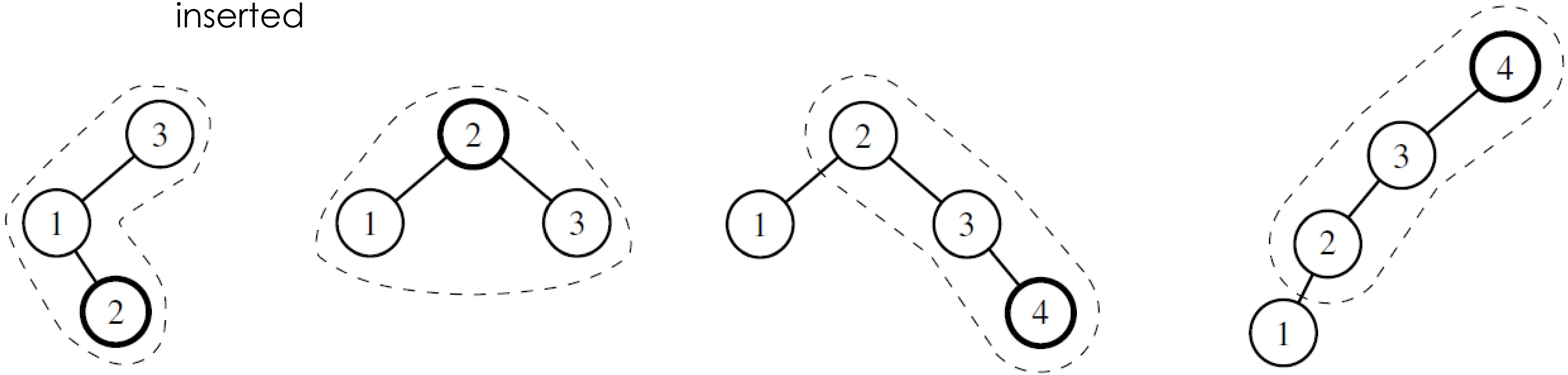
- When to splay
  - When searching for key k, if k is found at position p, splay p;
    - else splay the leaf position at which the search terminates unsuccessfully
  - When inserting key k, splay the newly created internal node where k gets inserted



(a) (b) (c)

SPLAY TREES(伸展树)

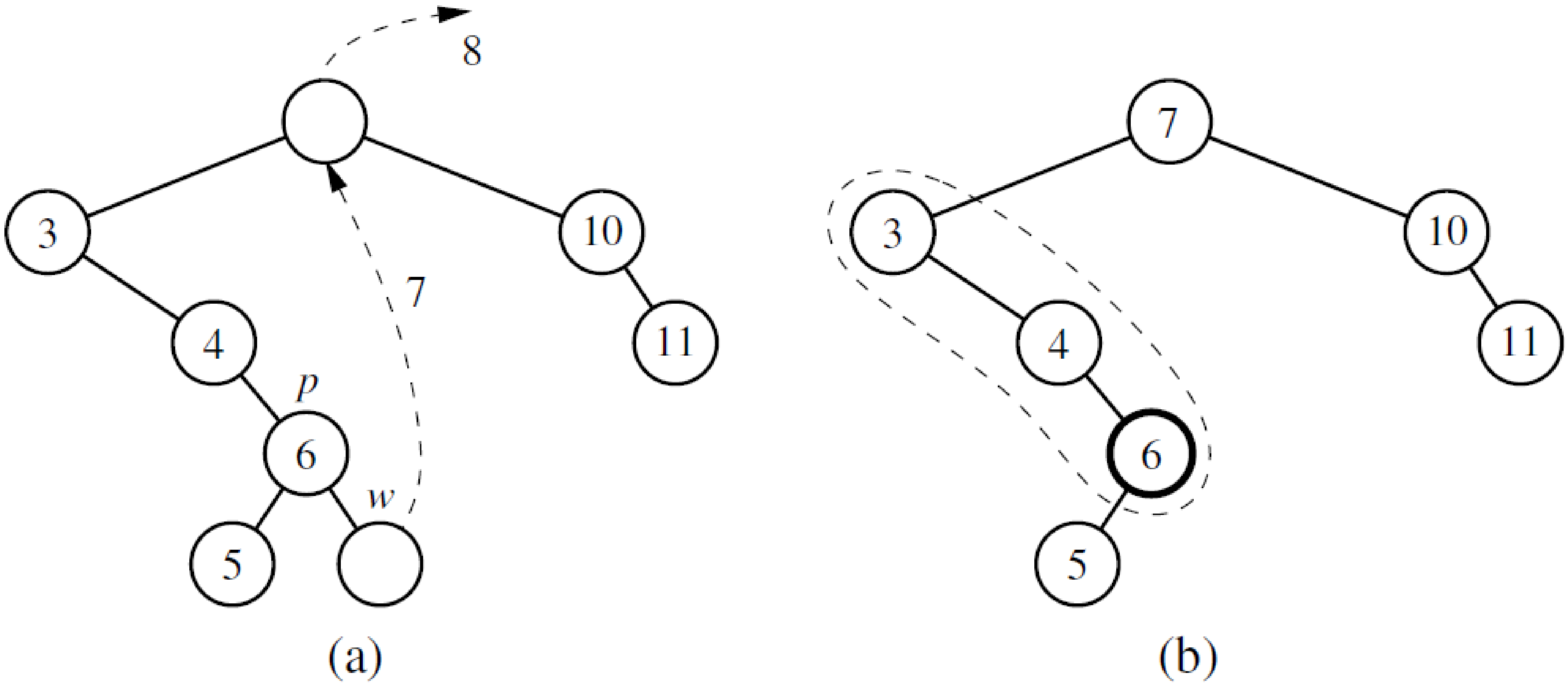
- When to splay
  - When searching for key k, if k is found at position p, splay p;
    - else splay the leaf position at which the search terminates unsuccessfully
  - When inserting key k, splay the newly created internal node where k gets inserted



(d) (e) (f) (g)

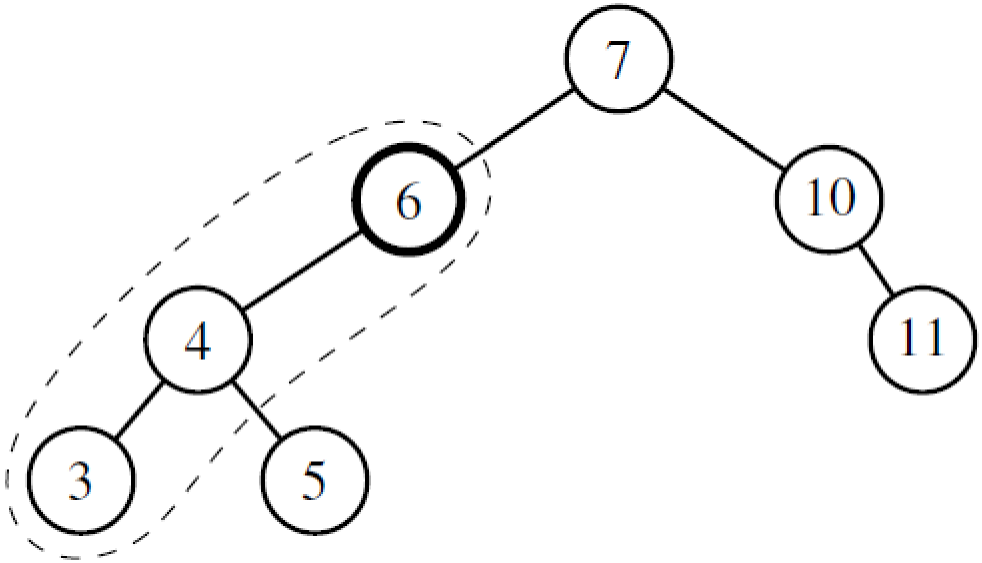
SPLAY TREES(伸展树)

- When to splay
  - When deleting a key, splay the position p that is the parent of the removed node

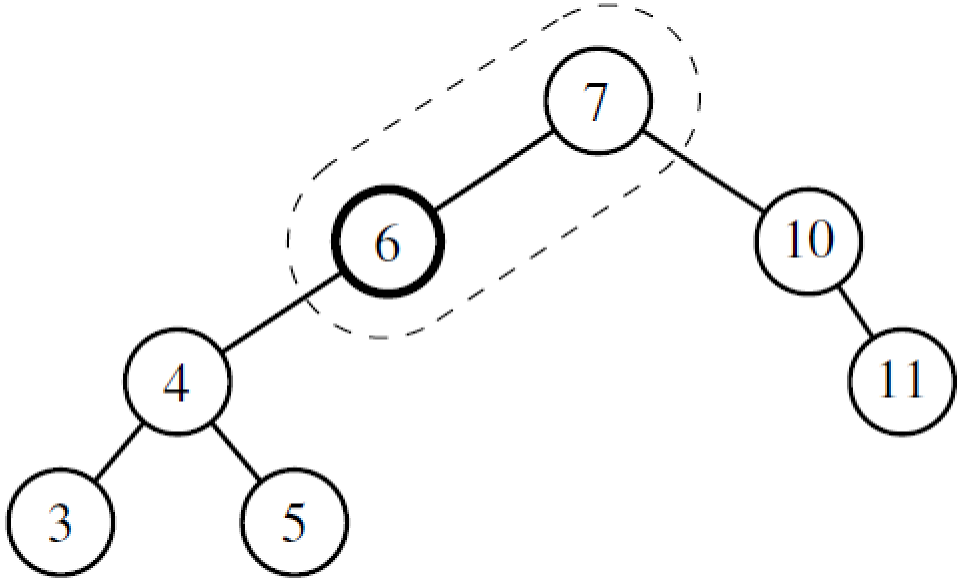


SPLAY TREES(伸展树)

- When to splay
  - When deleting a key, splay the position p that is the parent of the removed node



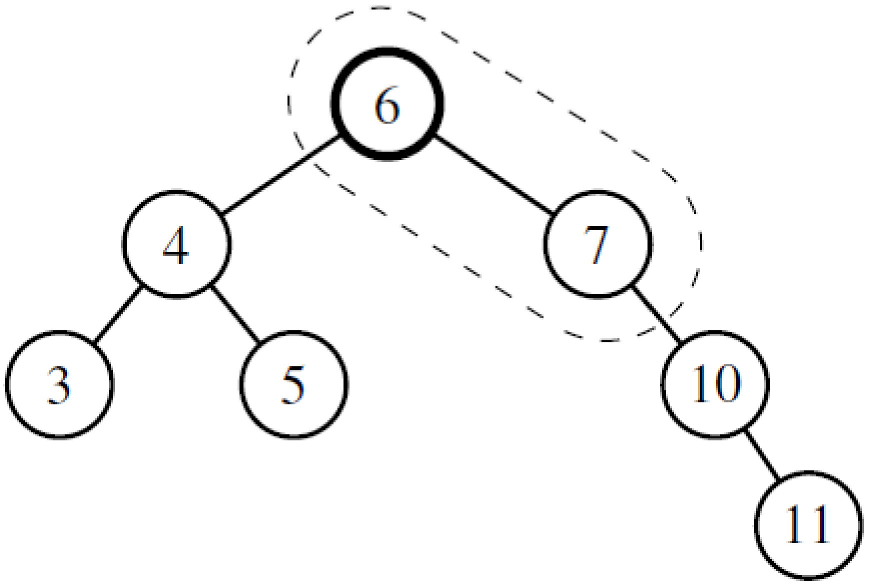
(c)



(d)

SPLAY TREES(伸展树)

- When to splay
  - When deleting a key, splay the position p that is the parent of the removed node



(e)

THANKS

See you in the next session!