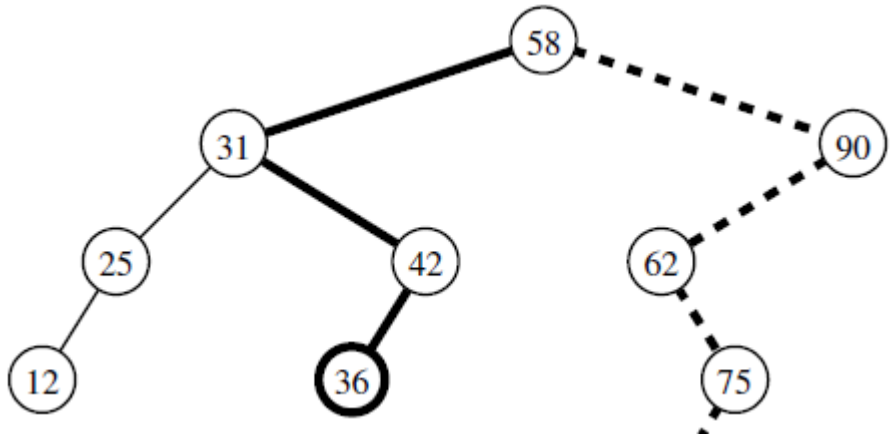
SEARCH TREES

School of Artificial Intelligence

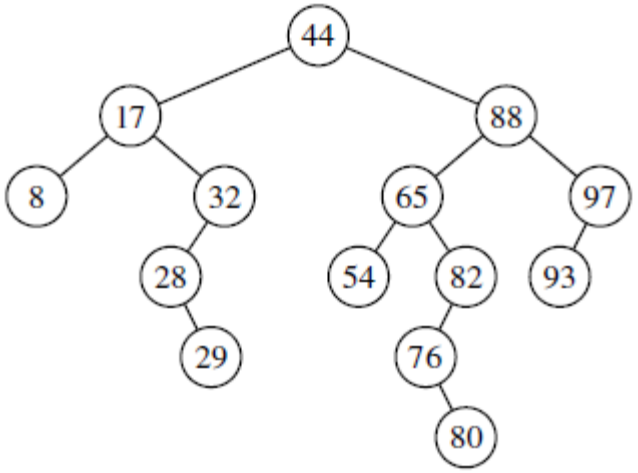
PREVIOUSLY ON BINARY SEARCH TREES

- Binary search tree: using tree to store an ordered sequence of elements in a binary tree
- Let S be a set whose unique elements have an order relation
- A binary search tree for S is a binary tree T such that
  - Position p stores an element of S, denoted as e(p)
  - Elements stored in the left subtree of p are less than e(p)
  - Elements stored in the right subtree of p are greater than e(p)



BINARY SEARCH TREES

- We'd use BSTs to implement a sorted map
  - M[k]
  - M[k] = v
  - del M[k]
- (Updated) definitions for BST
- Binary Search Tree property: a binary tree T with each position p storing a key-value pair (k, v) such that
  - Keys stored in the left subtree of p are less than k
  - Keys stored in the right subtree of p are greater than k



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

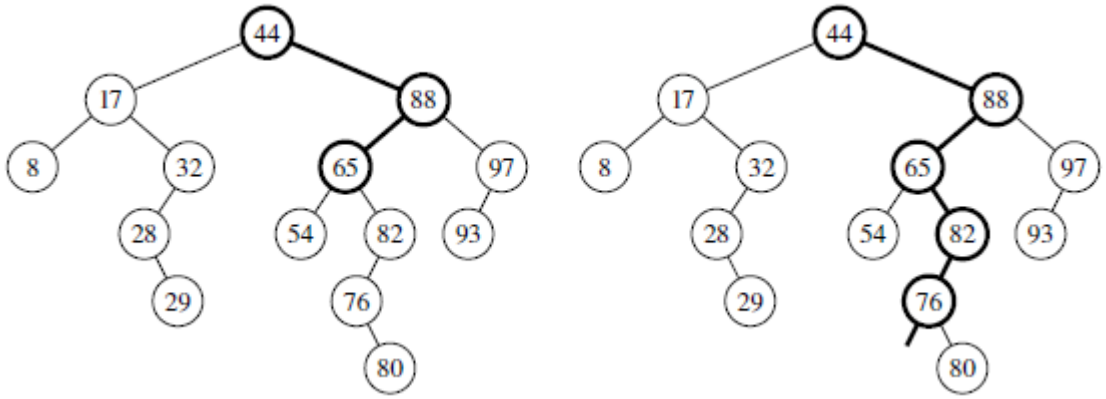
BINARY SEARCH TREE ADT

- after() and before() algorithms
- Complexity?
  - Bounded by the height h of the full tree
  - O(h)
  - O(1) amortised -> n calls to after(p) at the first position executes in a total of O(n) time

```
Algorithm after(p):
    if right(p) is not None then {successor is leftmost position in p's right subtree}
      walk = right(p)
      while left(walk) is not None do
        walk = left(walk)
      return walk
    else {successor is nearest ancestor having p in its left subtree}
      walk = p
      ancestor = parent(walk)
      while ancestor is not None and walk == right(ancestor) do
        walk = ancestor
        ancestor = parent(walk)
      return ancestor
```

SEARCHES

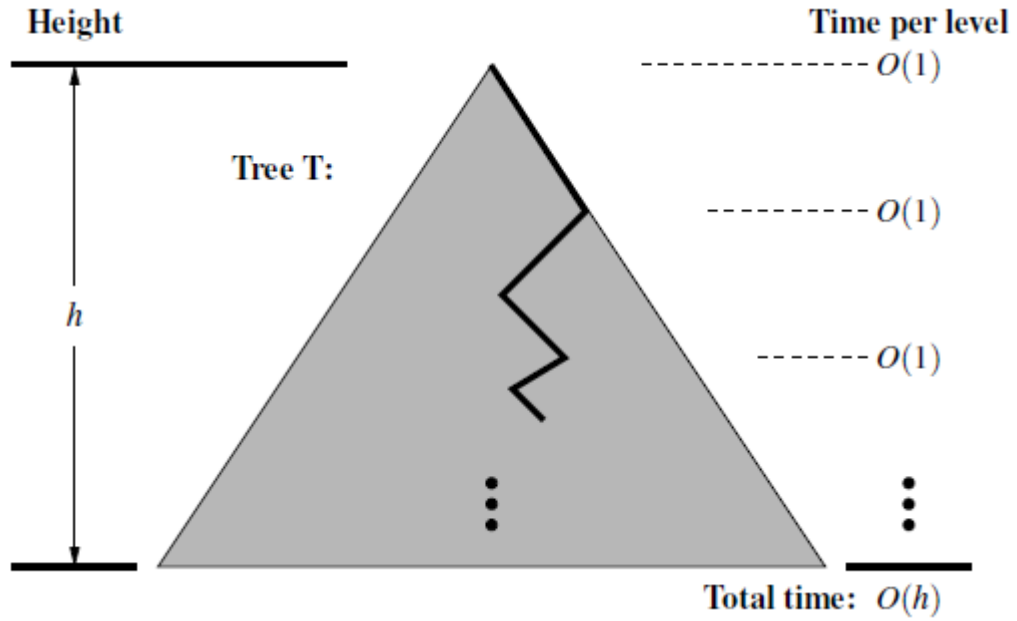
- Locate a particular key by viewing it as a decision tree
- At each position p: is the desired k less than, equal to, or greater than the key stored at position p?




```
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
```

TREE SEARCHING: ANALYSIS

- TreeSearch: recursive
- Each recursive call of TreeSearch(): made on a child of the previous position
- A Path p: starts at the root and goes down one level at a time.
- Length of path p: bounded by h+1, h is the height of T
- TreeSearch: O(h)

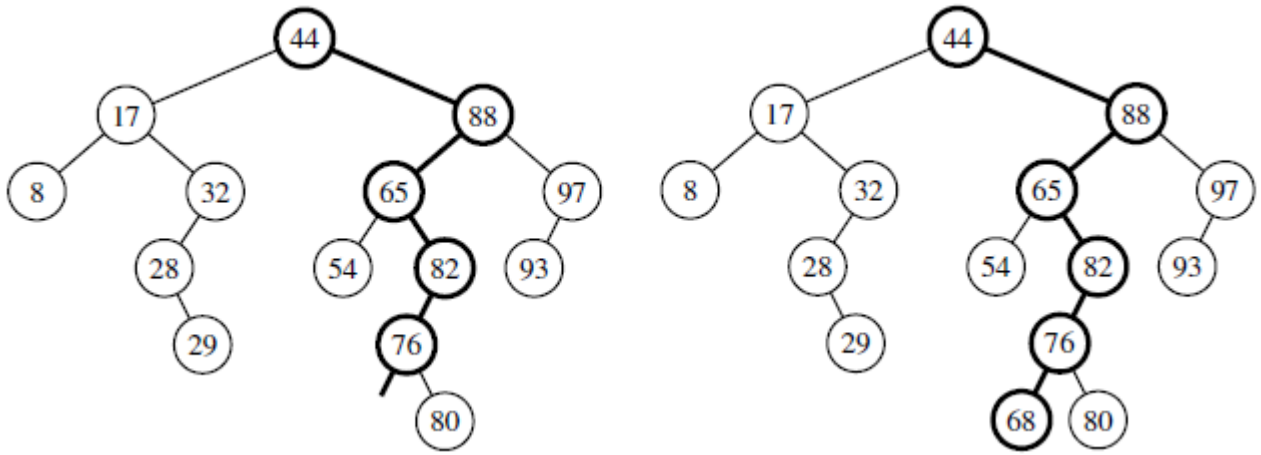



```
Algorithm TreeSearch(T, p, k):
    if k == p.key() then
        return p
    else if k < p.key() and T.left(p) is not None then
        return TreeSearch(T, T.left(p), k)
    else if k > p.key() and T.right(p) is not None then
        return TreeSearch(T, T.right(p), k)
    return p
```

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
```



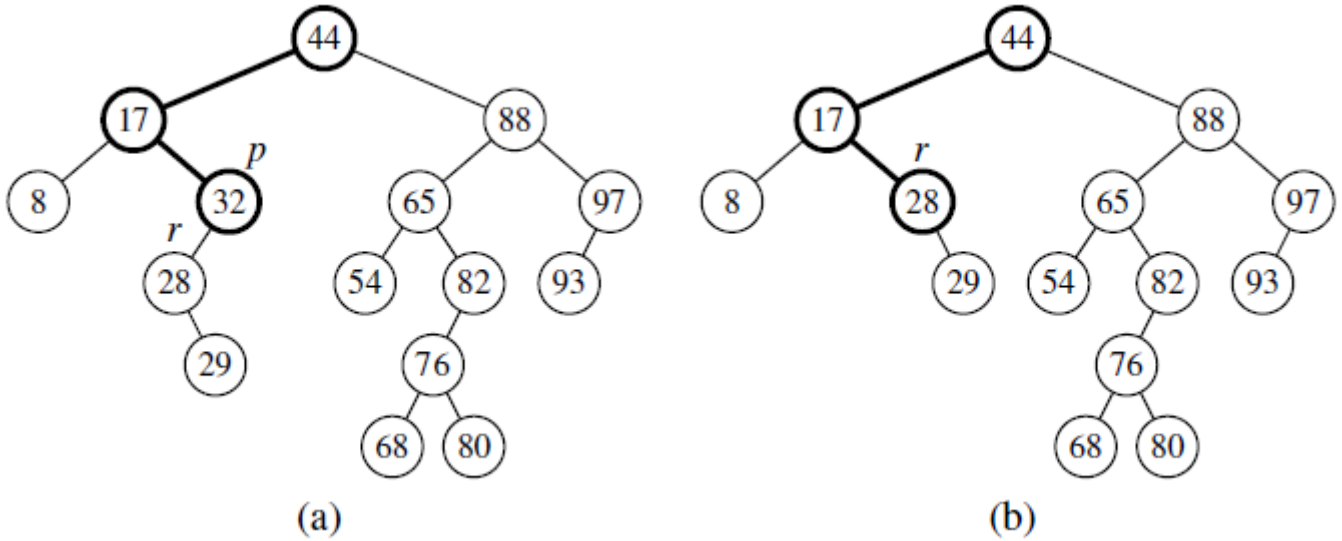
DELETION

- 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

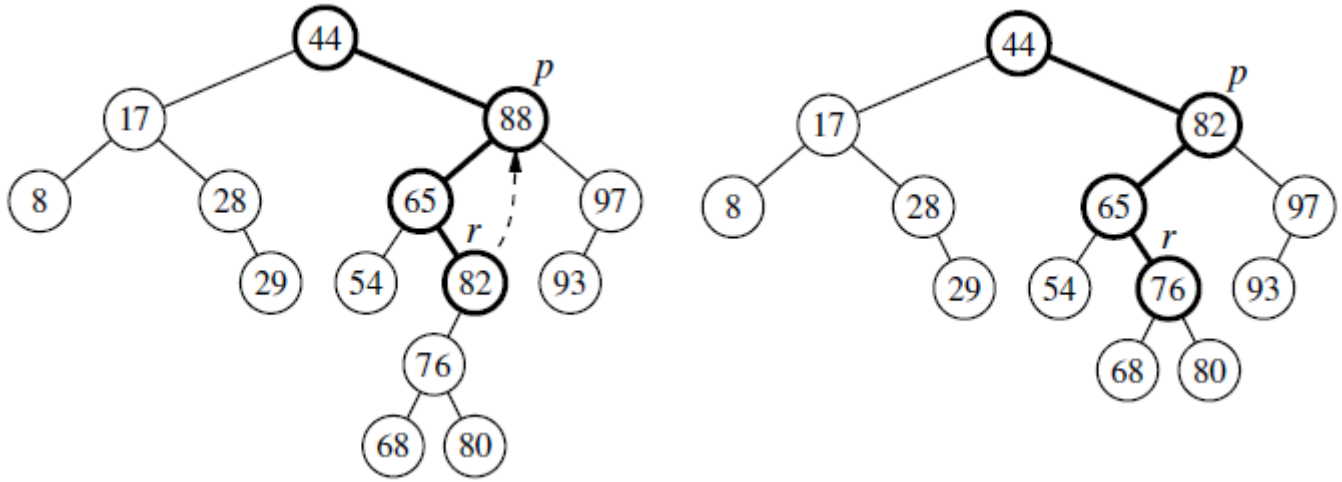
DELETION

- Delete item with k=32 with one child r



DELETION

- Delete item with k==88



PYTHON IMPLEMENTATION

- Will look at it in a practical

```
class TreeMap(LinkedBinaryTree, MapBase):
  """Sorted map implementation using a binary search tree."""

  #---------------------- override Position class ----------
  class Position(LinkedBinaryTree.Position):
    def key(self):
      """Return key of map's key-value pair."""
      return self.element()._key

    def value(self):
      """Return value of map's key-value pair."""
      return self.element()._value
```

```
  def _subtree_search(self, p, k):
    """Return Position of p's subtree having key k, or last node searched."""
    if k == p.key():                                  # found match
      return p
    elif k < p.key():                                 # search left subtree
      if self.left(p) is not None:
        return self._subtree_search(self.left(p), k)
    else:                                             # search right subtree
      if self.right(p) is not None:
        return self._subtree_search(self.right(p), k)
    return p                                          # unsucessful search

  def _subtree_first_position(self, p):
    """Return Position of first item in subtree rooted at p."""
    walk = p
    while self.left(walk) is not None:                # keep walking left
      walk = self.left(walk)
    return walk

  def _subtree_last_position(self, p):
    """Return Position of last item in subtree rooted at p."""
    walk = p
    while self.right(walk) is not None:               # keep walking right
      walk = self.right(walk)
    return walk
```

PYTHON IMPLEMENTATION

```
40    def first(self):
41      """Return the first Position in the tree (or None if empty)."""
42      return self._subtree_first_position(self.root( )) if len(self) > 0 else None
43
44    def last(self):
45      """Return the last Position in the tree (or None if empty)."""
46      return self._subtree_last_position(self.root( )) if len(self) > 0 else None
47
48    def before(self, p):
49      """Return the Position just before p in the natural order.
50
51      Return None if p is the first position.
52      """
53      self._validate(p)                       # inherited from LinkedBinaryTree
54      if self.left(p):
55        return self._subtree_last_position(self.left(p))
56      else:
57        # walk upward
58        walk = p
59        above = self.parent(walk)
60        while above is not None and walk == self.left(above):
61          walk = above
62          above = self.parent(walk)
63        return above
```

```
65    def after(self, p):
66      """Return the Position just after p in the
67
68      Return None if p is the last position.
69      """
70      # symmetric to before(p)
71
72    def find_position(self, k):
73      """Return position with key k, or else neig
74      if self.is_empty( ):
75        return None
76      else:
77        p = self._subtree_search(self.root( ), k)
78        self._rebalance_access(p)               # h
79        return p
```

PYTHON IMPLEMENTATION

```
80    def find_min(self):
81      """Return (key,value) pair with minimum key (or None if e
82      if self.is_empty( ):
83        return None
84      else:
85        p = self.first( )
86        return (p.key( ), p.value( ))
87
88    def find_ge(self, k):
89      """Return (key,value) pair with least key greater than or ec
90
91      Return None if there does not exist such a key.
92      """
93      if self.is_empty( ):
94        return None
95      else:
96        p = self.find_position(k)                    # may not fi
97        if p.key( ) < k:                             # p's key is t
98          p = self.after(p)
99        return (p.key( ), p.value( )) if p is not None else None
```

```
101   def find_range(self, start, stop):
102     """Iterate all (key,value) pairs such that start <= key < stop.
103
104     If start is None, iteration begins with minimum key of map.
105     If stop is None, iteration continues through the maximum key of
106     """
107     if not self.is_empty( ):
108       if start is None:
109         p = self.first( )
110       else:
111         # we initialize p with logic similar to find_ge
112         p = self.find_position(start)
113         if p.key( ) < start:
114           p = self.after(p)
115       while p is not None and (stop is None or p.key( ) < stop):
116         yield (p.key( ), p.value( ))
117         p = self.after(p)
```

PYTHON I ATION

```
118   def __getitem__(self, k):
119     """Return value associated with key k (raise Key
120     if self.is_empty( ):
121       raise KeyError('Key Error: ' + repr(k))
122     else:
123       p = self._subtree_search(self.root( ), k)
124       self._rebalance_access(p)              # hook fo
125       if k != p.key( ):
126         raise KeyError('Key Error: ' + repr(k))
127       return p.value( )
128
129   def __setitem__(self, k, v):
130     """Assign value v to key k, overwriting existing
131     if self.is_empty( ):
132       leaf = self._add_root(self._Item(k,v))
133     else:
134       p = self._subtree_search(self.root( ), k)
135       if p.key( ) == k:
136         p.element( )._value = v              # replace
137         self._rebalance_access(p)            # hook fo
138         return
139       else:
140         item = self._Item(k,v)
141         if p.key( ) < k:
142           leaf = self._add_right(p, item)  # inherite
143         else:
144           leaf = self._add_left(p, item)   # inherite
145     self._rebalance_insert(leaf)             # hook fo
```

```
147   def __iter__(self):
148     """Generate an iteration
149     p = self.first( )
150     while p is not None:
151       yield p.key( )
152       p = self.after(p)
153   def delete(self, p):
154     """Remove the item at given Position."""
155     self._validate(p)                        # inhe
156     if self.left(p) and self.right(p):       # p ha
157       replacement = self._subtree_last_position(s
158       self._replace(p, replacement.element( ))
159       p =  replacement
160     # now p has at most one child
161     parent = self.parent(p)
162     self._delete(p)                          # inhe
163     self._rebalance_delete(parent)           # if ro
164
165   def __delitem__(self, k):
166     """Remove item associated with key k (raise
167     if not self.is_empty( ):
168       p = self._subtree_search(self.root( ), k)
169       if k == p.key( ):
170         self.delete(p)                       # rely
171         return                               # succ
172       self._rebalance_access(p)              # hool
173     raise KeyError('Key Error: ' + repr(k))
```

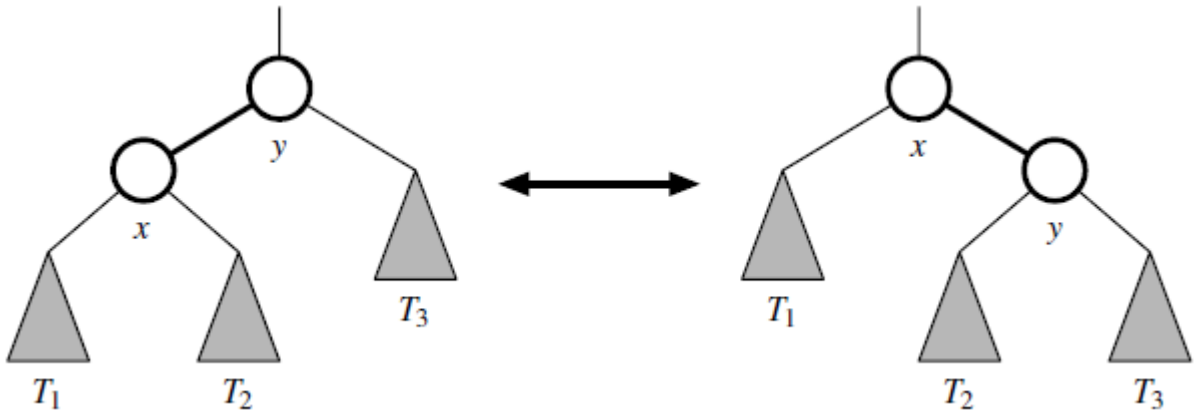
PERFORMANCE OF A BINARY SEARCH TREE

- 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)	$O(n)$

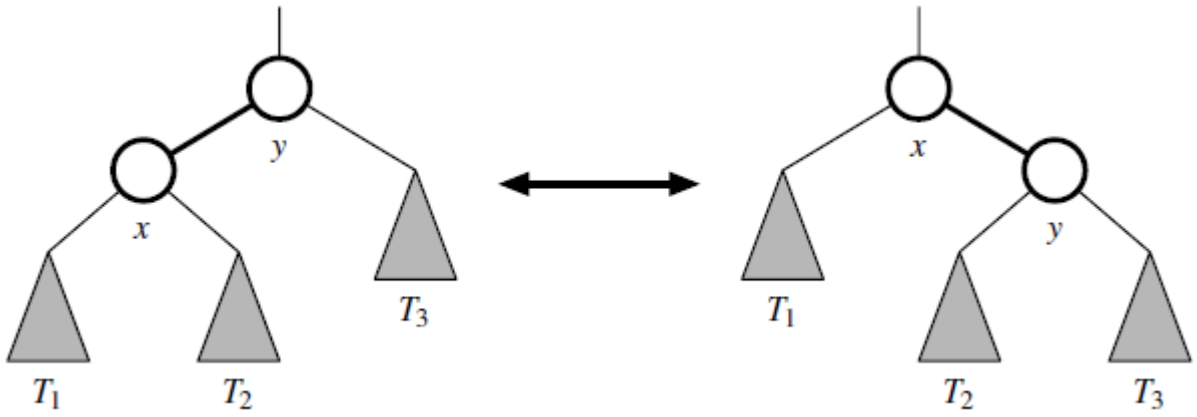
BALANCED SEARCH TREES

- 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



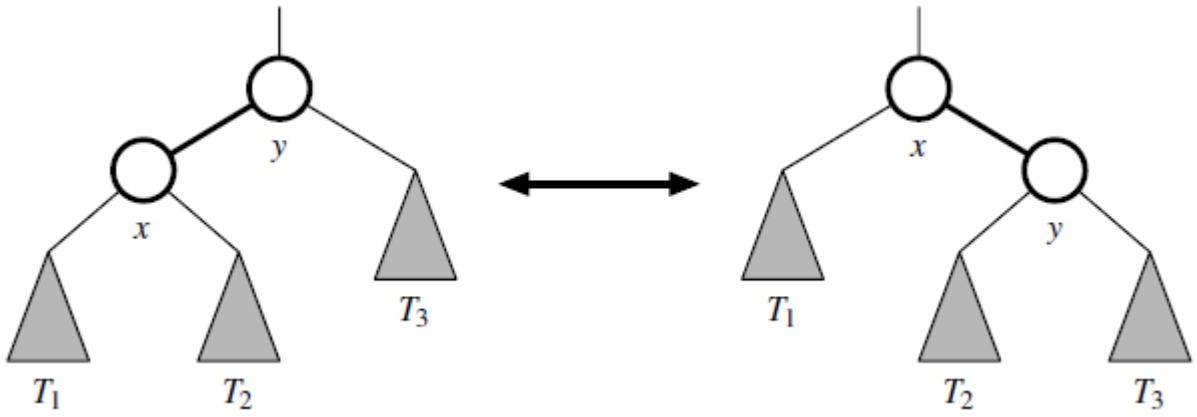
BALANCED SEARCH TREES

- Before Rotation
  - Position x: x.key < y.key
  - For each node in T1, their keys are less than x.key
  - For each node in T2, their keys are greater than x.key, but less than y.key
- After rotation
  - Position x: x.key < y.key
  - For each node in T1, their keys are less than x.key
  - For each node in T2, their keys are greater than x.key, but less than y.key



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

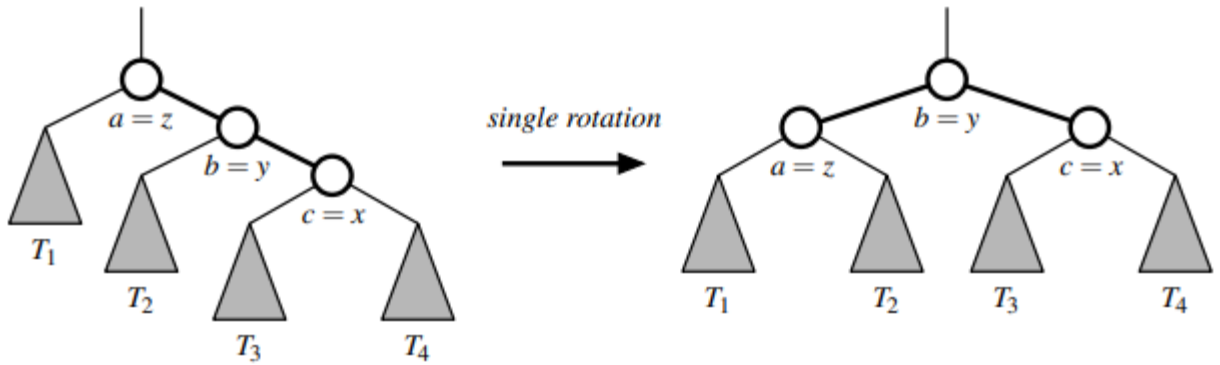
1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

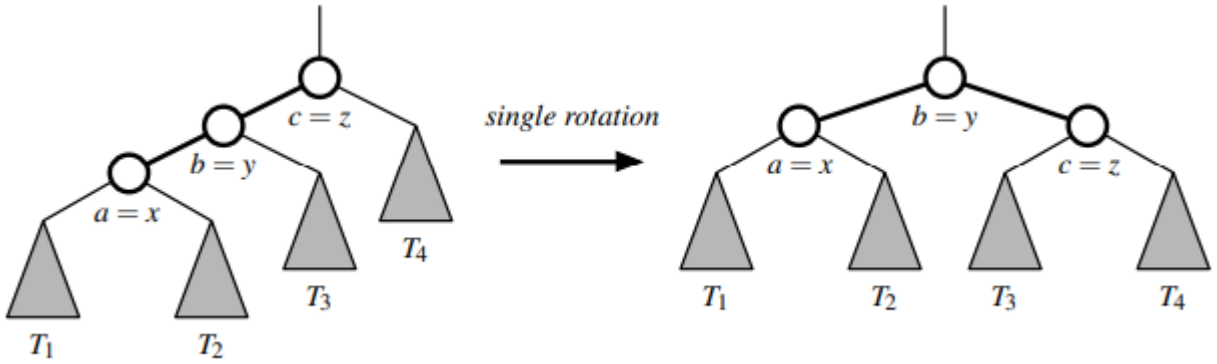
2: Replace the subtree rooted at z with a new subtree rooted at b.

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

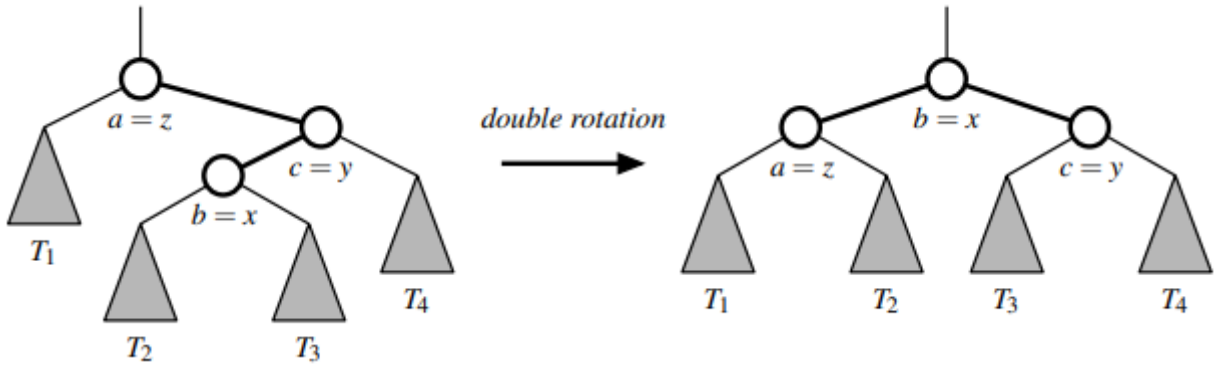
1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

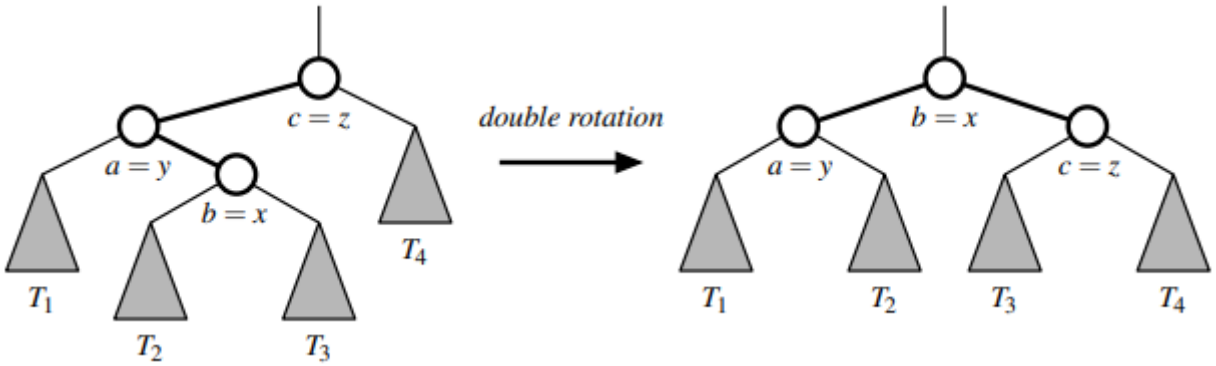
2: Replace the subtree rooted at z with a new subtree rooted at b.

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.

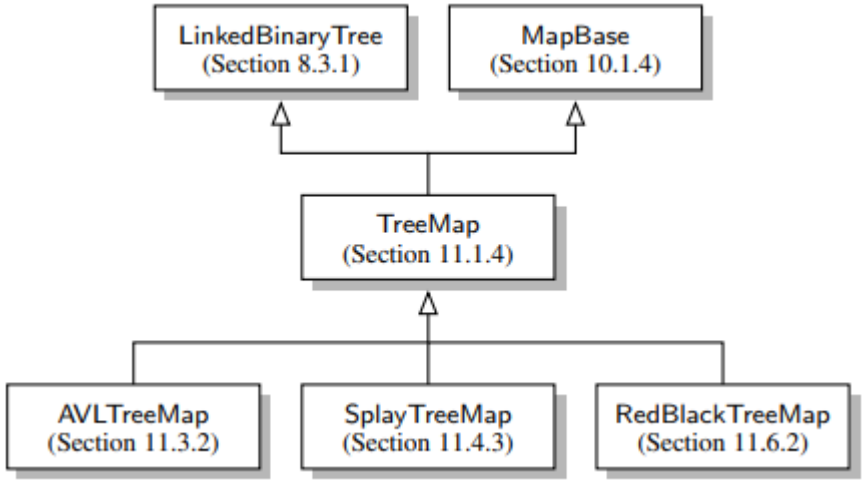
2: Replace the subtree rooted at z with a new subtree rooted at b.

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.

PYTHON FRAMEWORK FOR BALANCING SEARCH TREES

- _rebalance_insert(p): called by __setitem__
- _rebalance_delete(p): called by __delitem__
- _rebalance_access(p): called by __getitem__



PYTHON FRAMEWORK FOR BALANCING SEARCH TREES

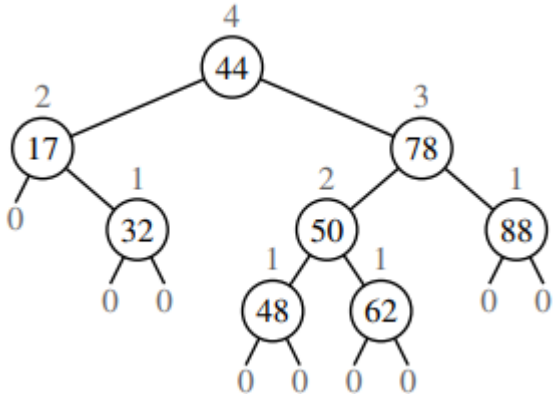
```
177   def _relink(self, parent, child, make_left_child):
178     """Relink parent node with child node (we all
179     if make_left_child:                              # ma
180       parent._left = child
181     else:                                            # ma
182       parent._right = child
183     if child is not None:                            # ma
184       child._parent = parent
```

```
186   def _rotate(self, p):
187     """Rotate Position p above its pa
188     x = p._node
189     y = x._parent
190     z = y._parent
191     if z is None:
192       self._root = x
193       x._parent = None
194     else:
195       self._relink(z, x, y == z._left)
196     # now rotate x and y, including t
197     if x == y._left:
198       self._relink(y, x._right, True)
199       self._relink(x, y, False)
200     else:
201       self._relink(y, x._left, False)
202       self._relink(x, y, True)
```

```
204   def _restructure(self, x):
205     """Perform trinode restructure of Position x with
206     y = self.parent(x)
207     z = self.parent(y)
208     if (x == self.right(y)) == (y == self.right(z)):
209       self._rotate(y)
210       return y
211     else:
212       self._rotate(x)
213       self._rotate(x)
214       return x
```

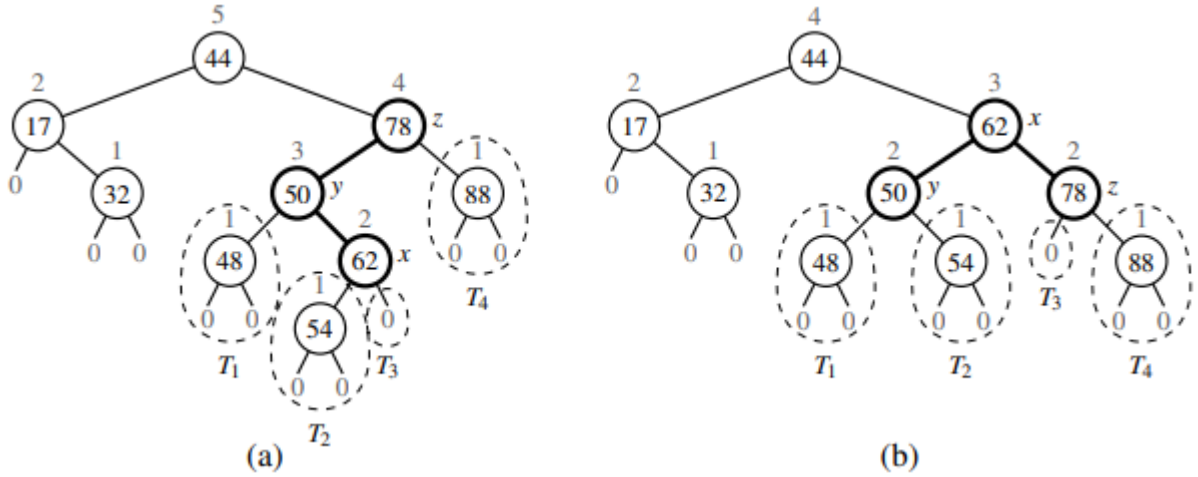
AVL TREES

- 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

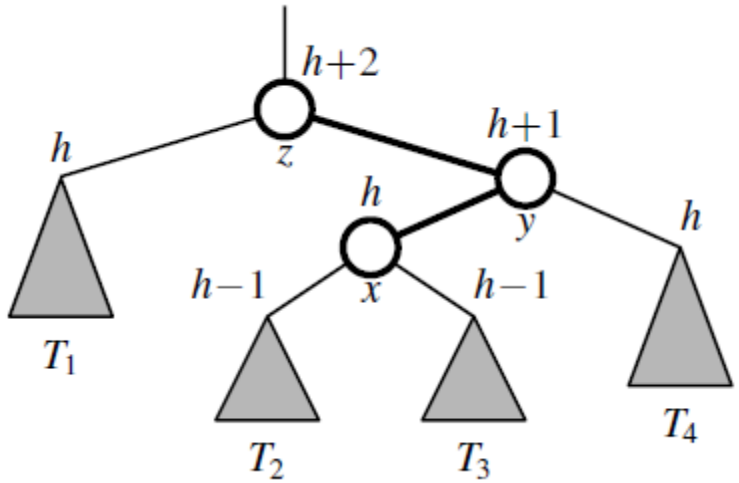


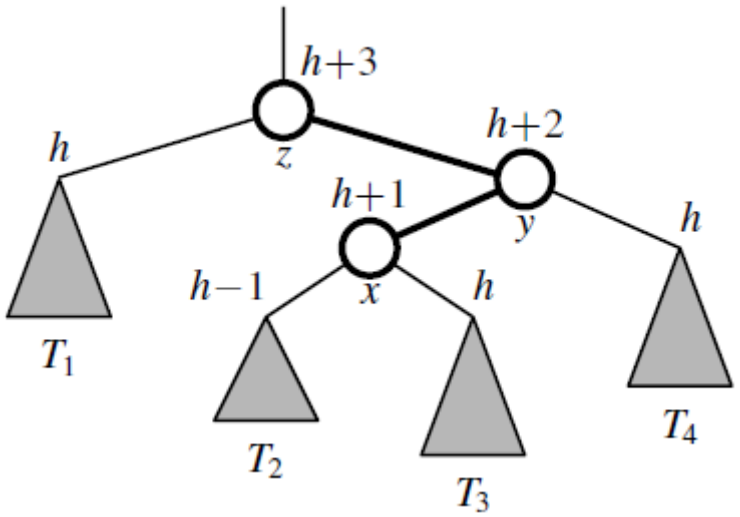
AVL TREES

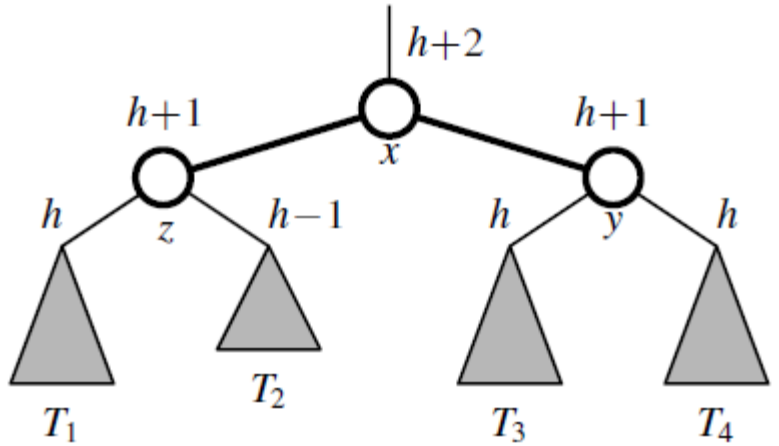
- Insertion: insert item with key 54
- "search and repair": going up from p to the root of T
  - z: first unbalanced position
  - y: child of z with higher height, y must be an ancestor of p
  - x: child of y with higher height (no tie, x must be an ancestor of p)
  - Call the trinode restructuring method,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



AVL TREES

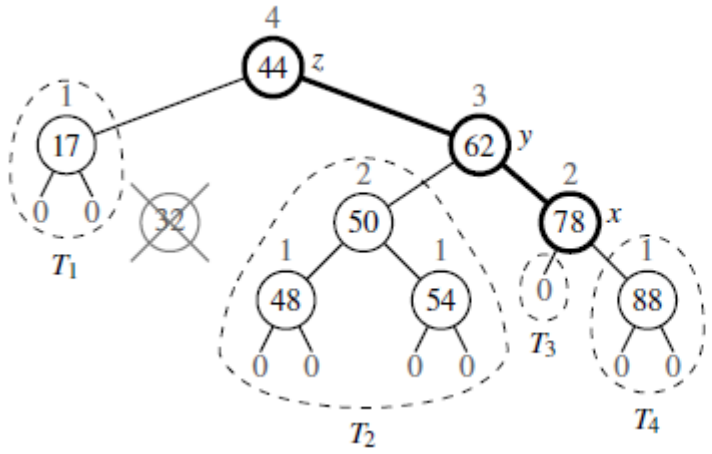

h+2
z
h
h+1
y
h
h
x
h−1
h−1
T1
T2
T3
T4

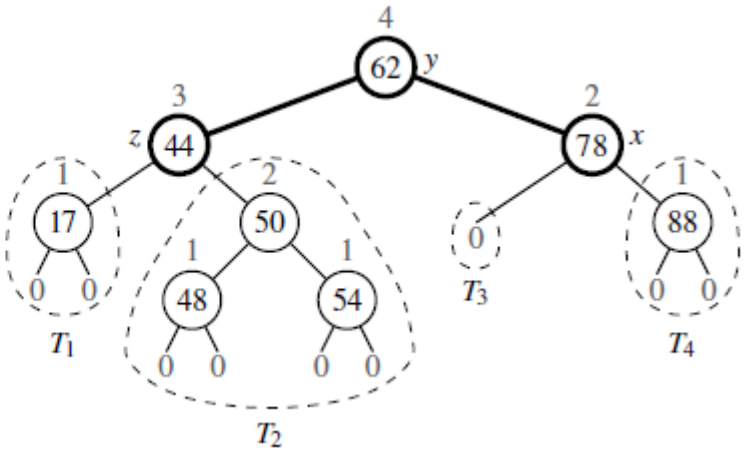

h+3
z
h
h+2
y
h+1
h
x
h−1
h
T1
T2
T3
T4


h+2
x
h+1
h+1
z
h
h−1
y
h
h
T1
T2
T3
T4

AVL TREES

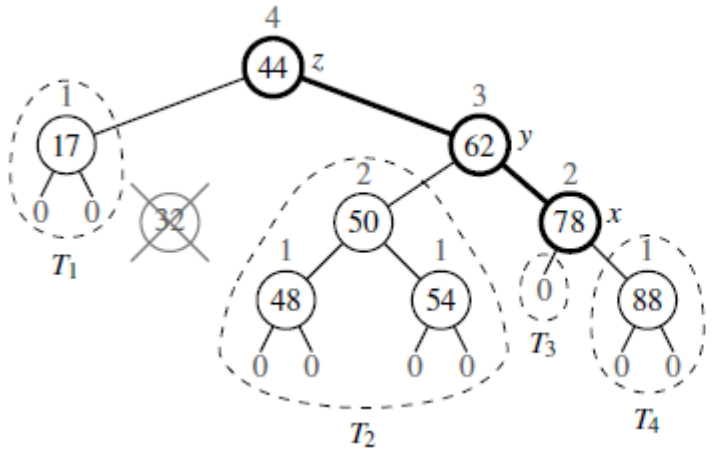
- Deletion: delete item with key 32
- Trinode restructuring:
  - z: first unbalanced position
  - y: child of z with larger height
  - x: child of y, such that if one the children of y is taller than the other, let x be the taller child of y. Else let x be the child of y on the same side as y
  - Perform restructure(x)

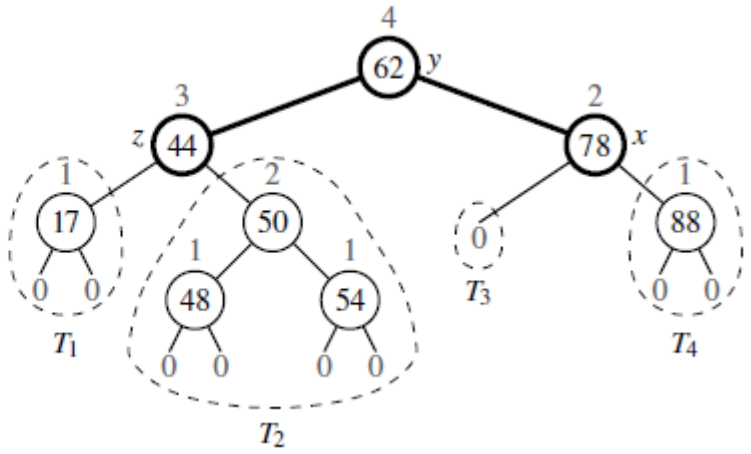




AVL TREES

- 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

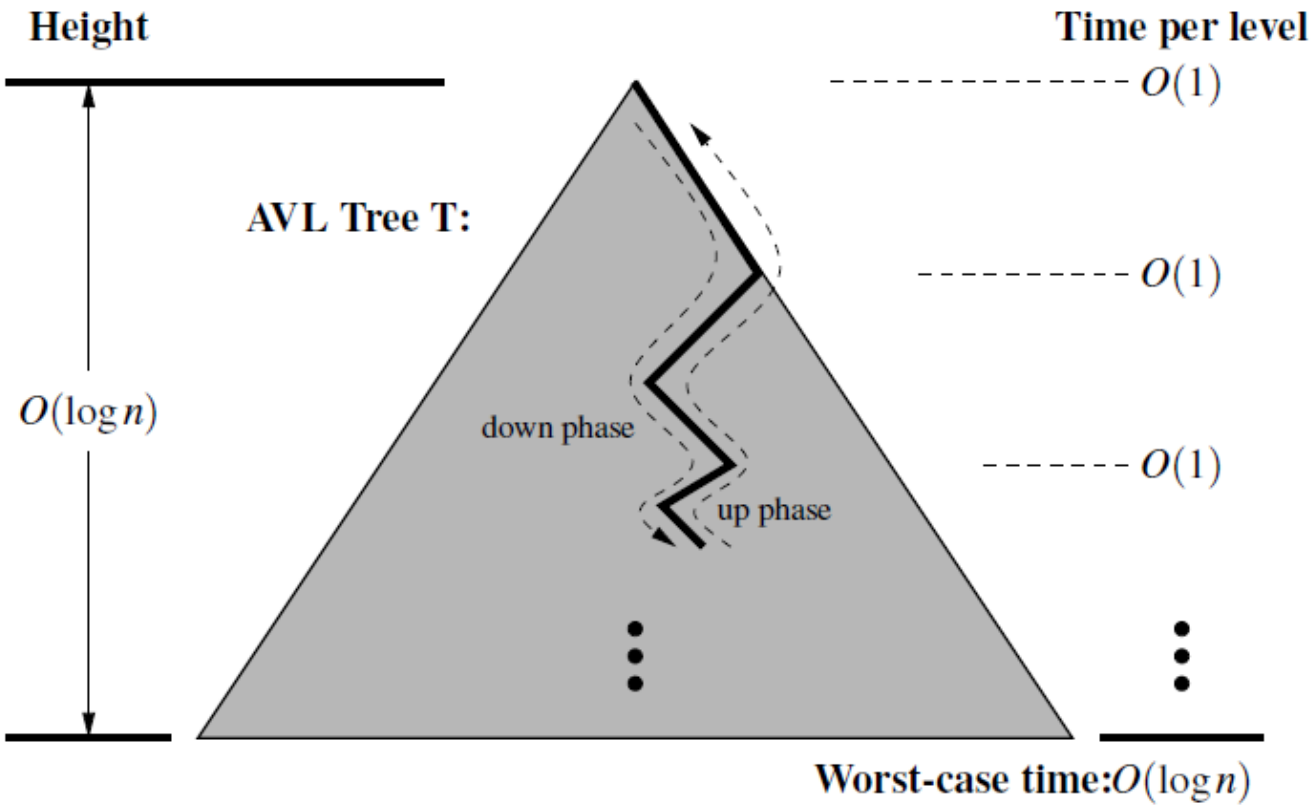




PERFORMANCE OF AVL TREES

- 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)	$O(n)$

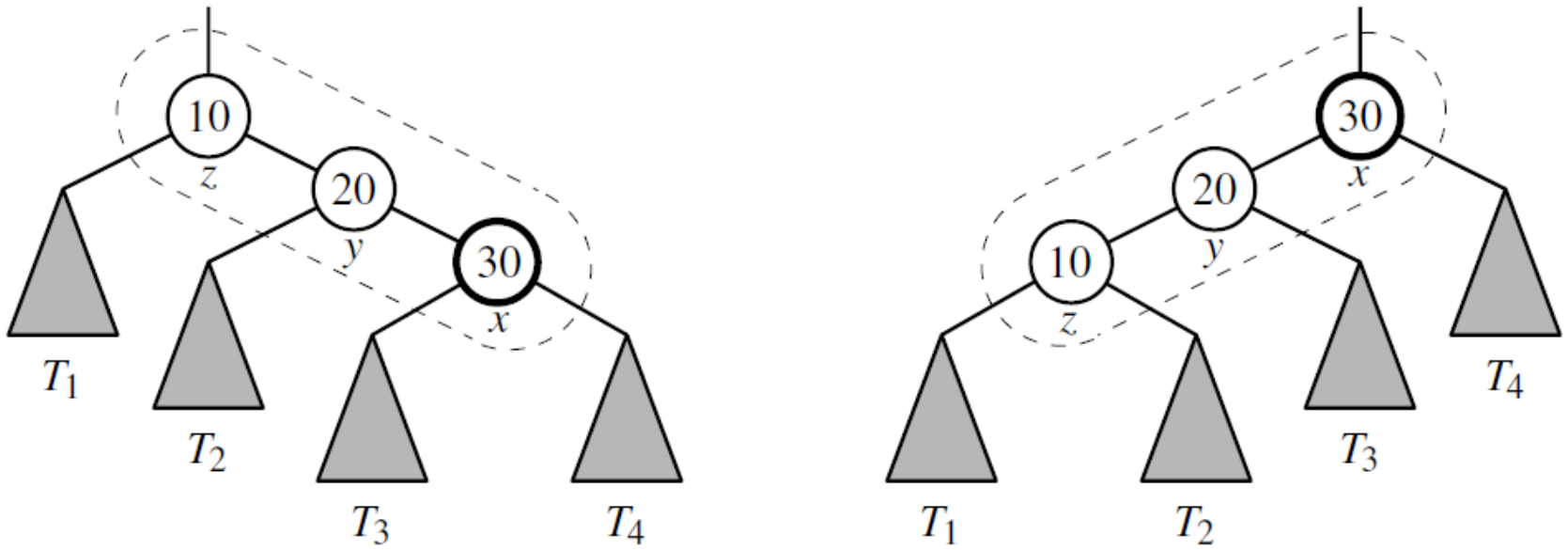


SPLAY TREES(伸展树)

- 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

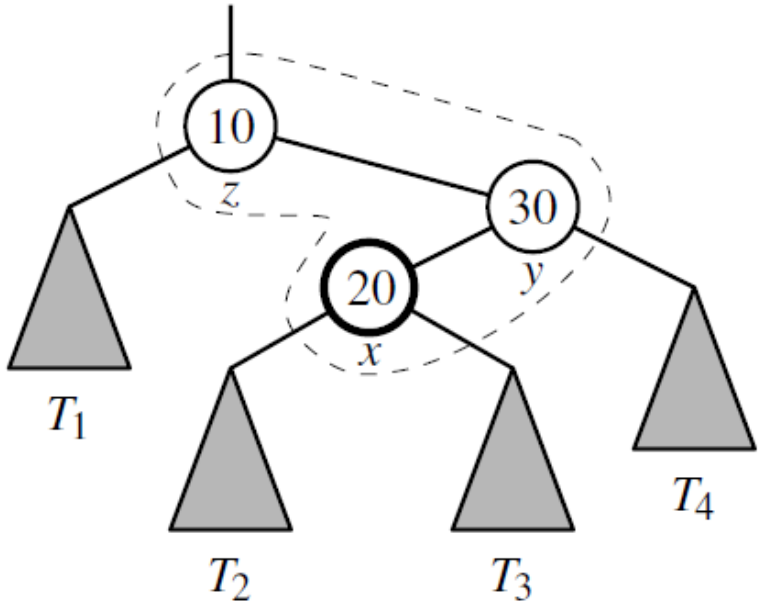
SPLAY TREES(伸展树)

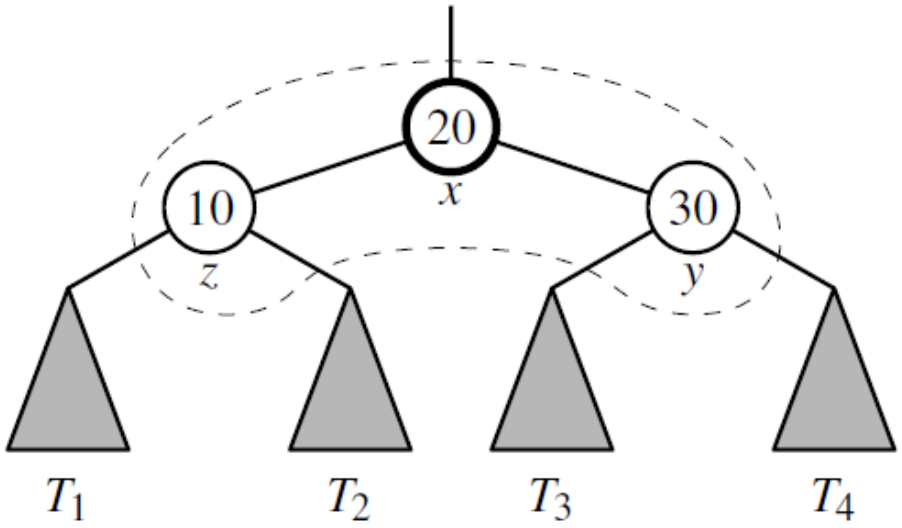
- Splay operations
- Given a node x of a binary search tree T, splay x – moving x to the root of T through a sequence of restructurings
- Zig-zig: node x and its parent y are both left children or both right children



SPLAY TREES(伸展树)

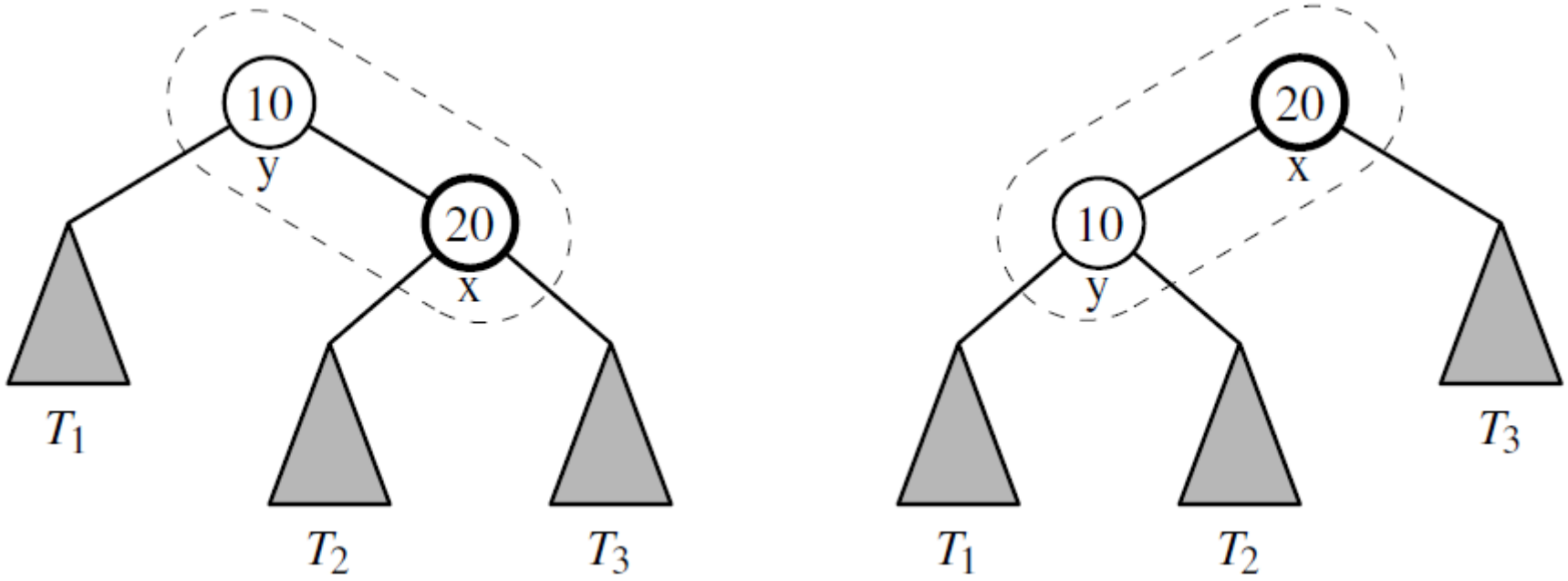
- Zig-zag: one of x and y is a left child and the other is a right child.
- Promote x by making x have y and z as its children, while maintaining the inorder relationships of the nodes in T



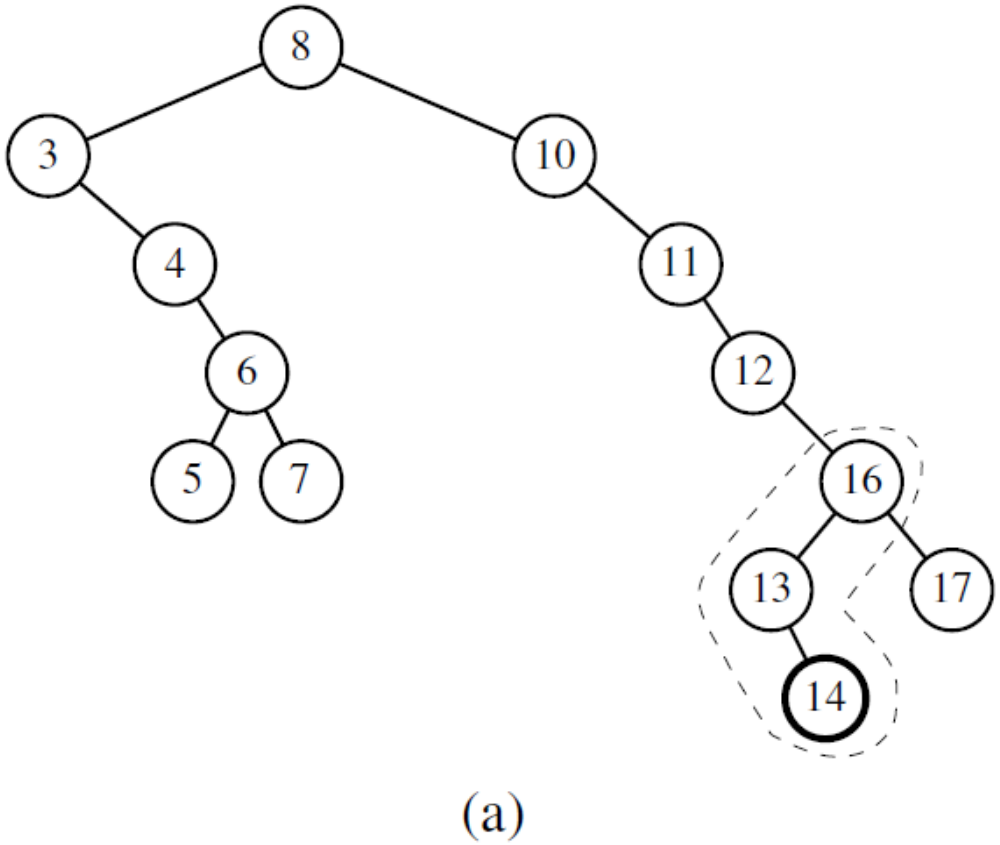


SPLAY TREES(伸展树)

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

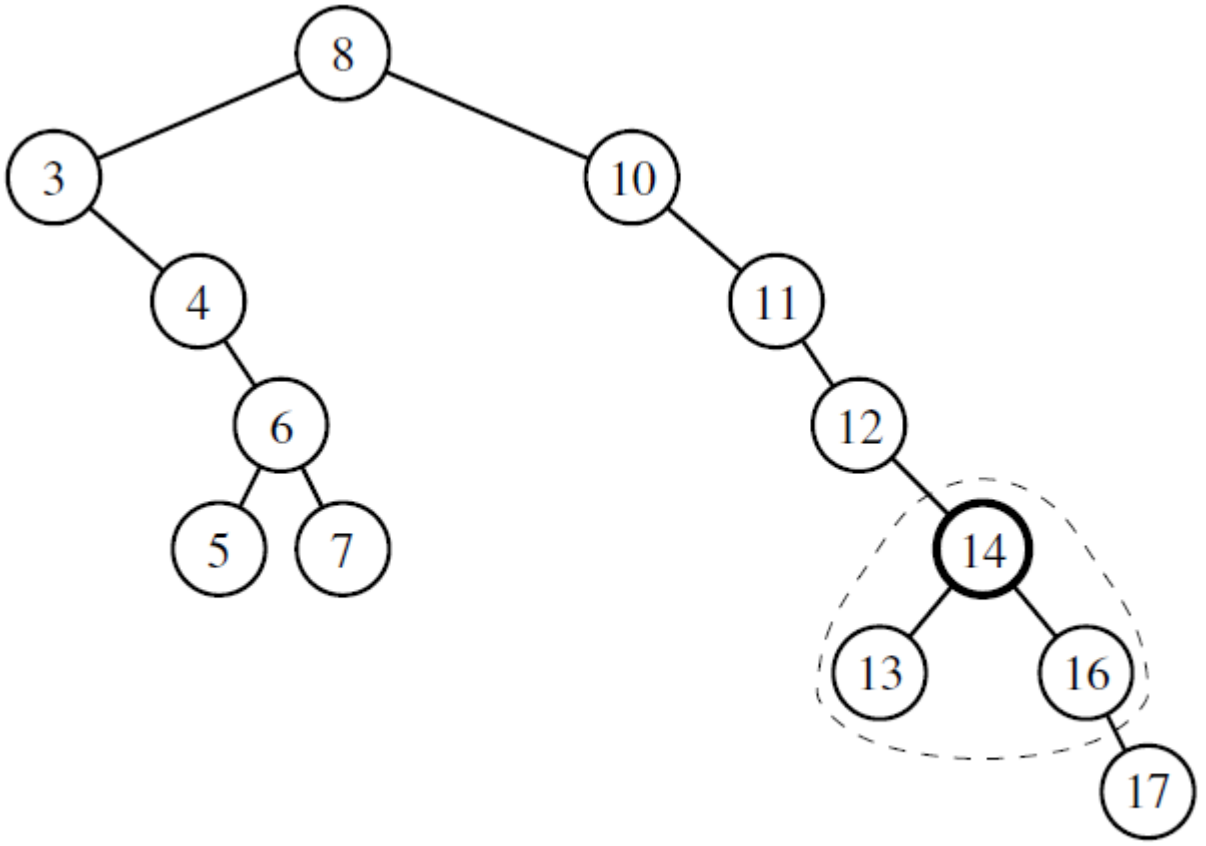


SPLAY TREES(伸展树)



(a)

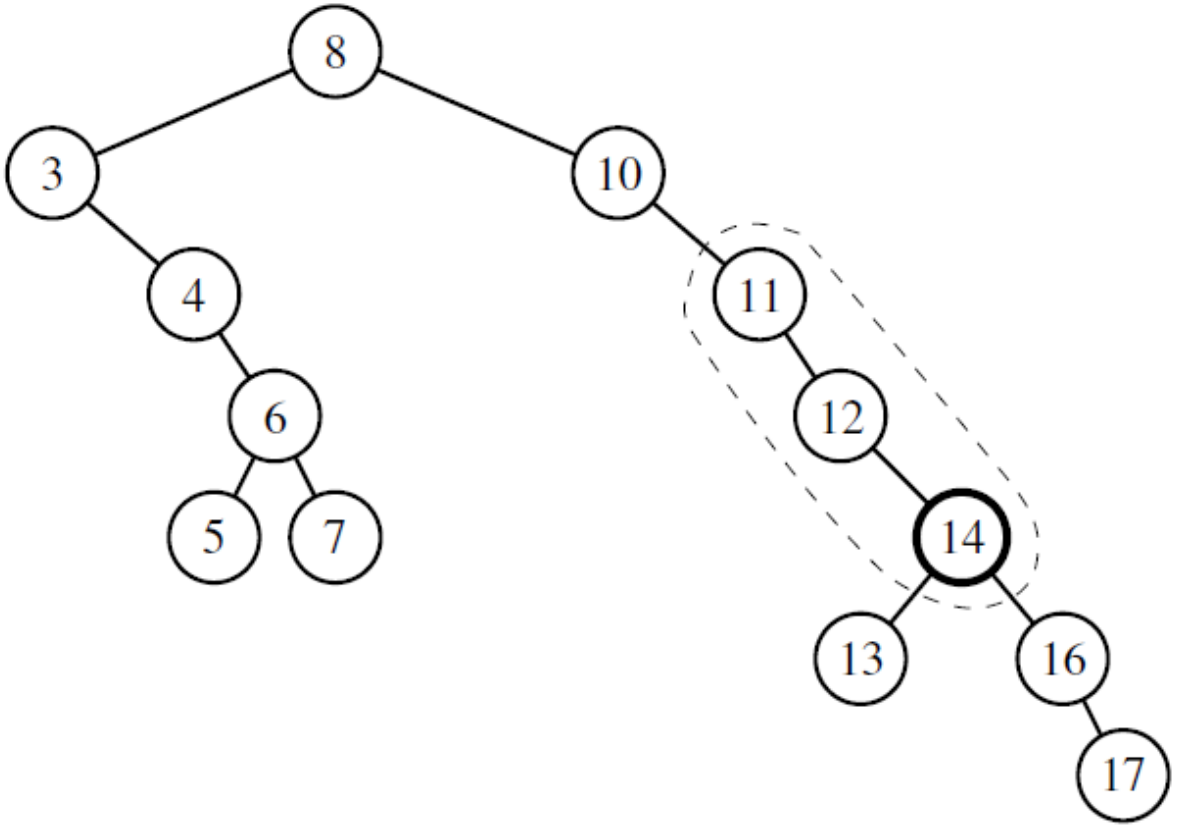
Splaying the node storing 14 with a zig-zag



(b)

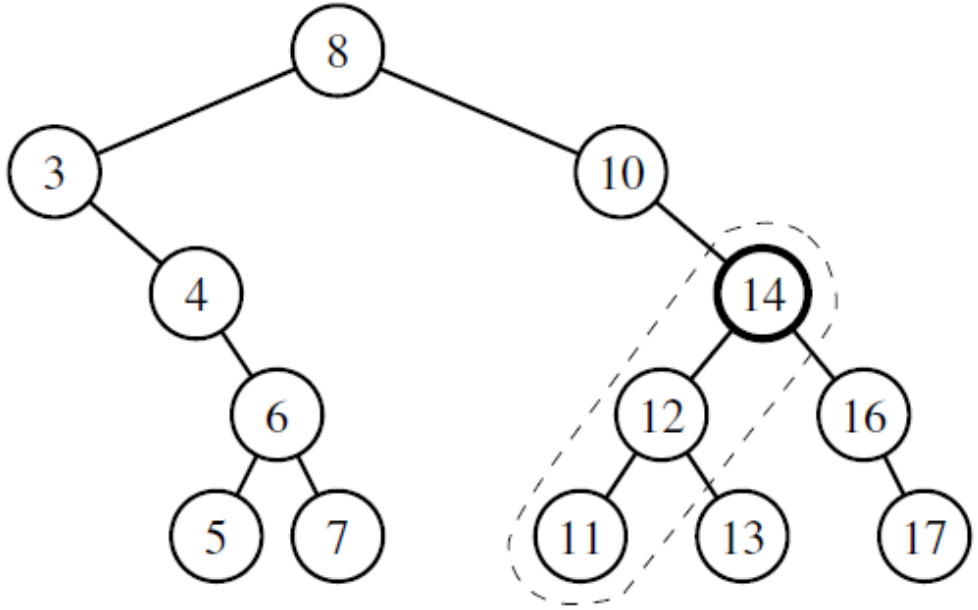
After the zig-zag

SPLAY TREES(伸展树)



(c)

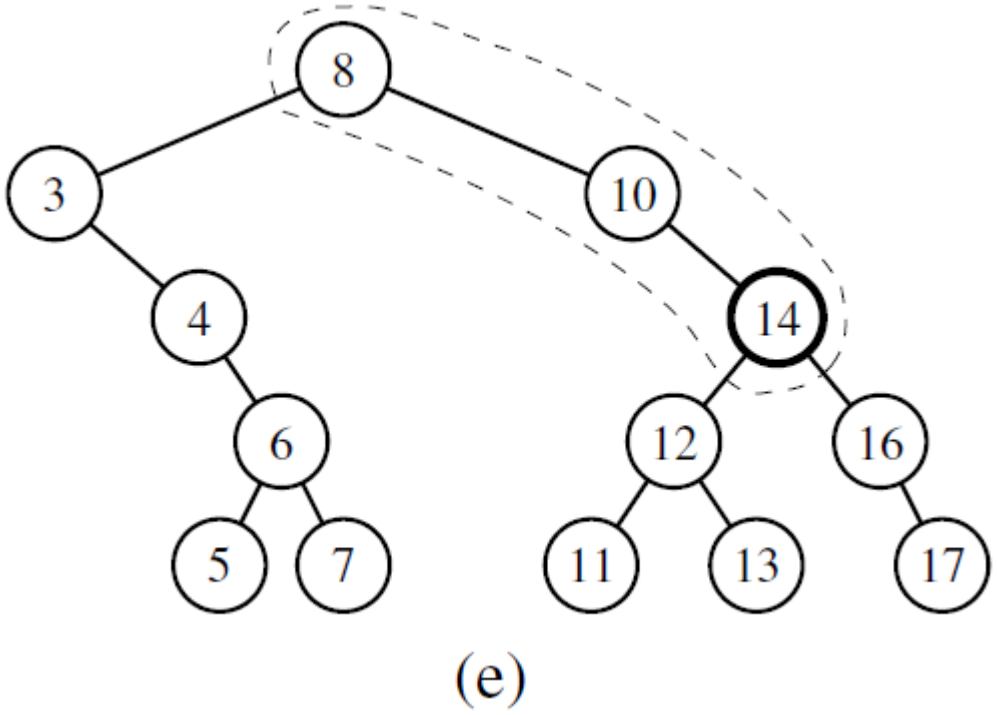
Zig-zig



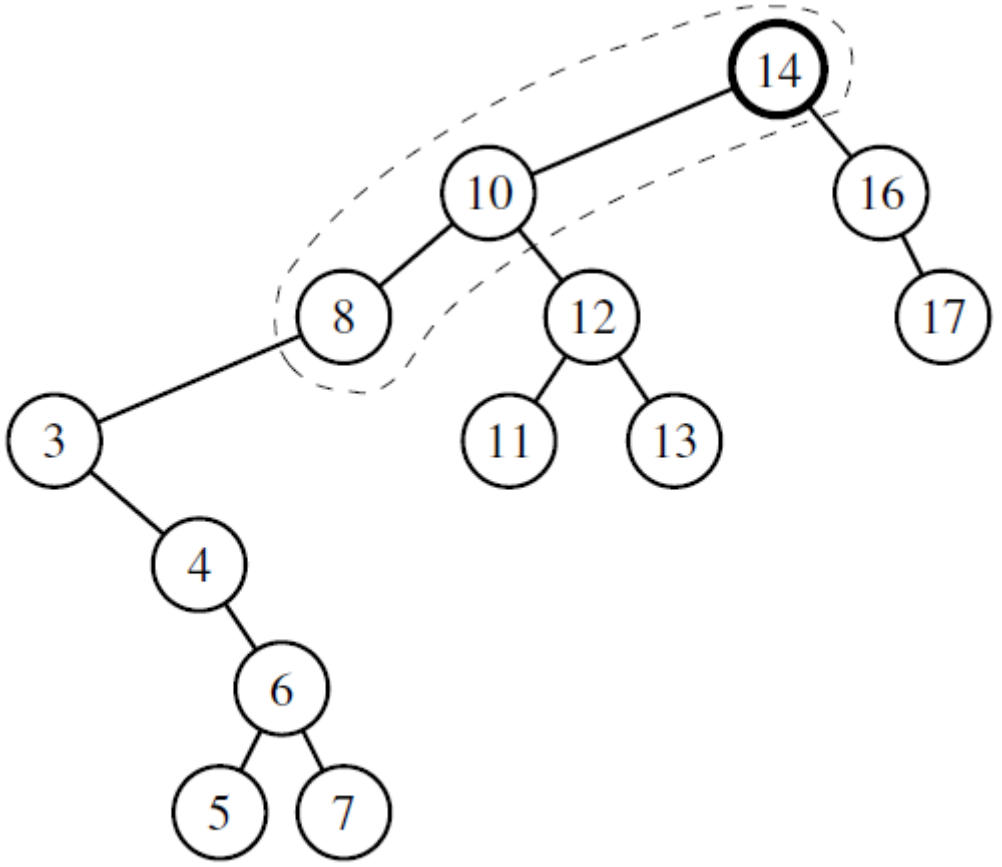
(d)

After the zig-zig

SPLAY TREES(伸展树)



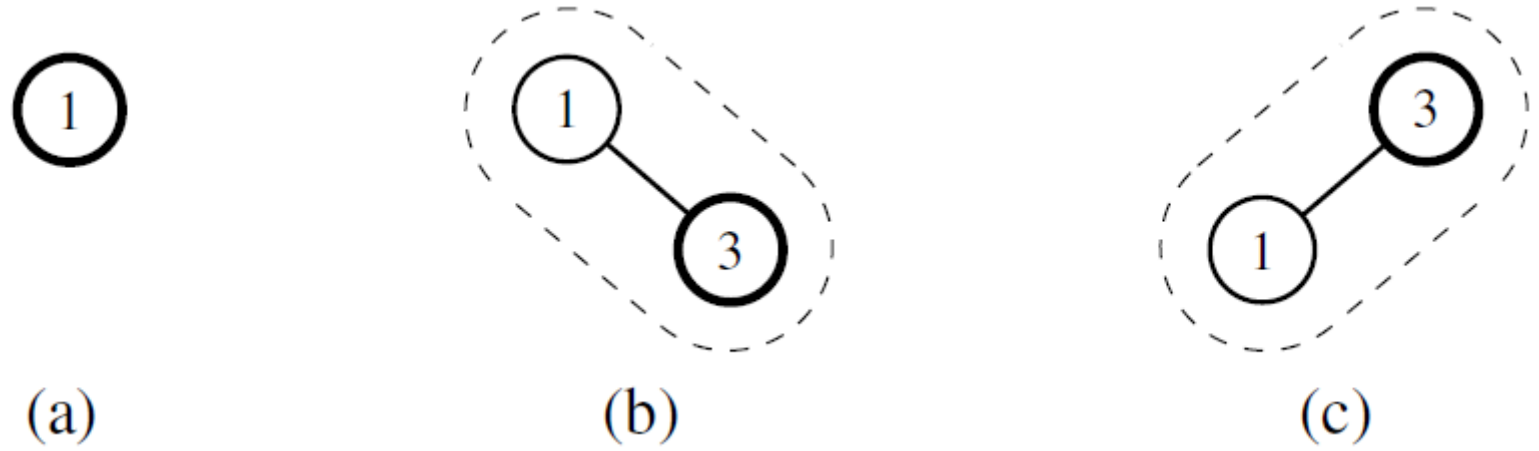
Zig-zig



After zig-zig

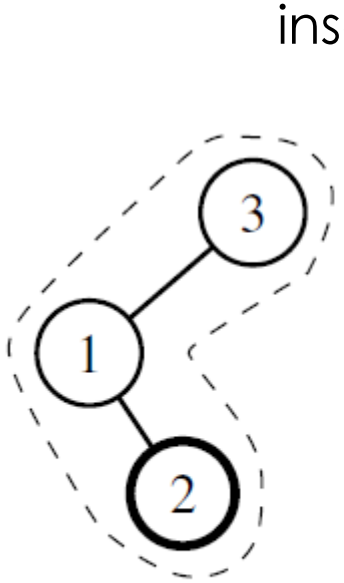
SPLAY TREES(伸展树)

- 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

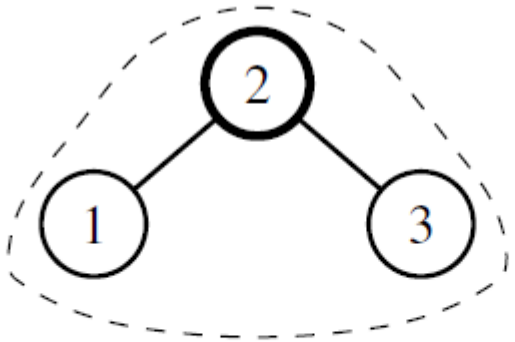


SPLAY TREES(伸展树)

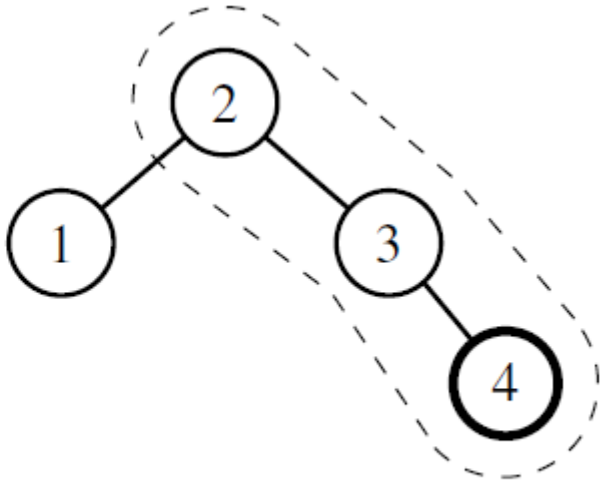
- When to splay
  - When searching for key k, if k is found at position p, splay p;
    - else splay the leaf position at which the search terminates unsuccessfully
  - When inserting key k, splay the newly created internal node where k gets inserted



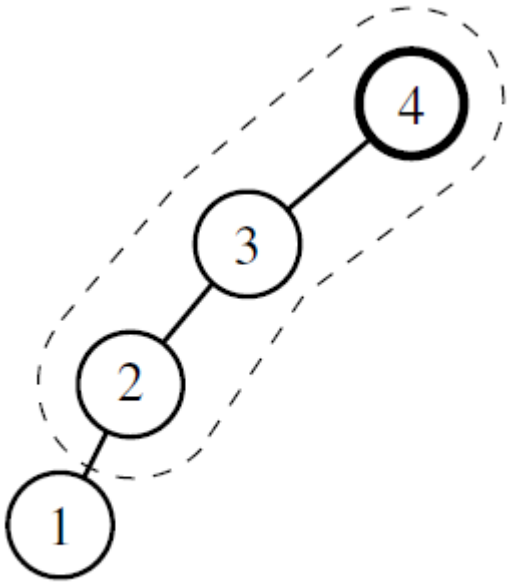
(d)



(e)



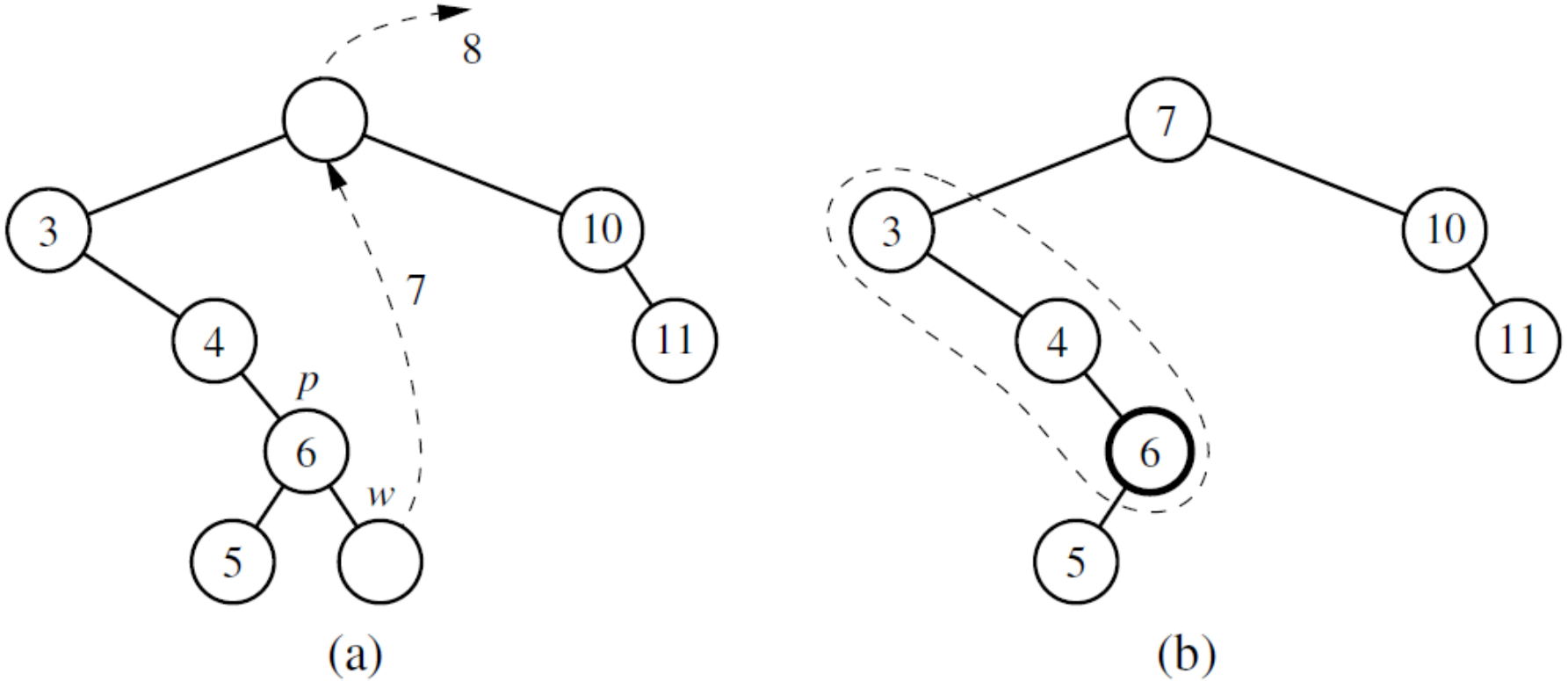
(f)



(g)

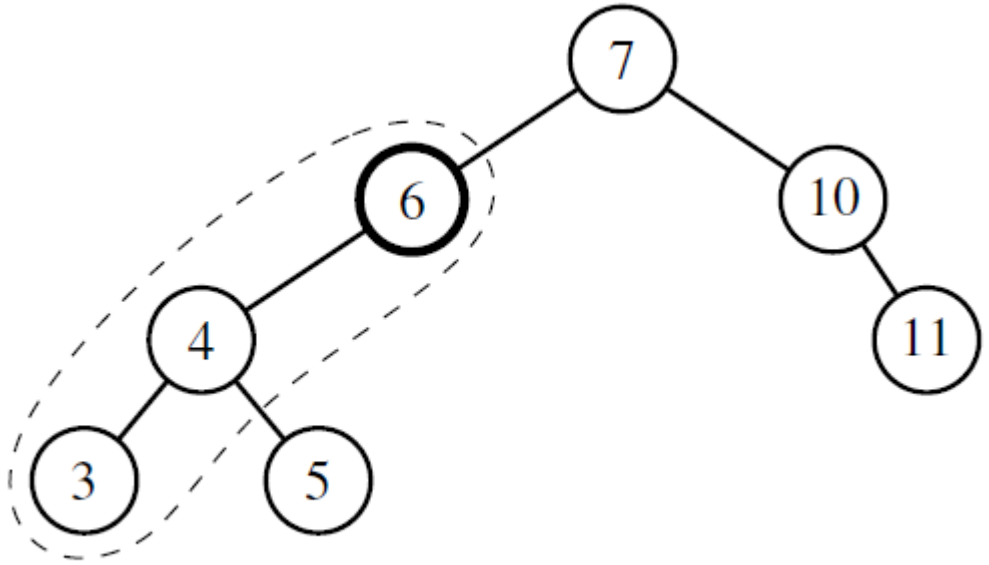
SPLAY TREES(伸展树)

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

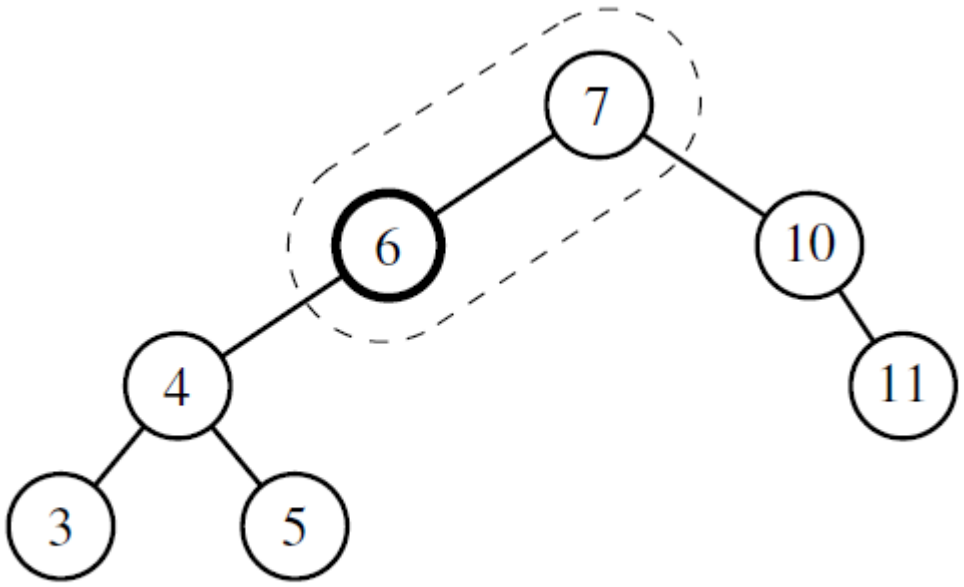


SPLAY TREES(伸展树)

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



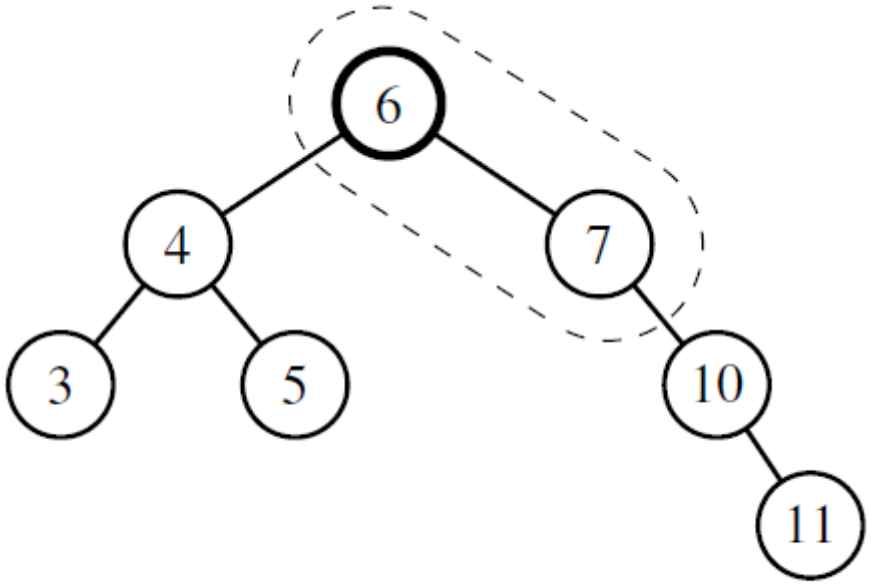
(c)



(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!