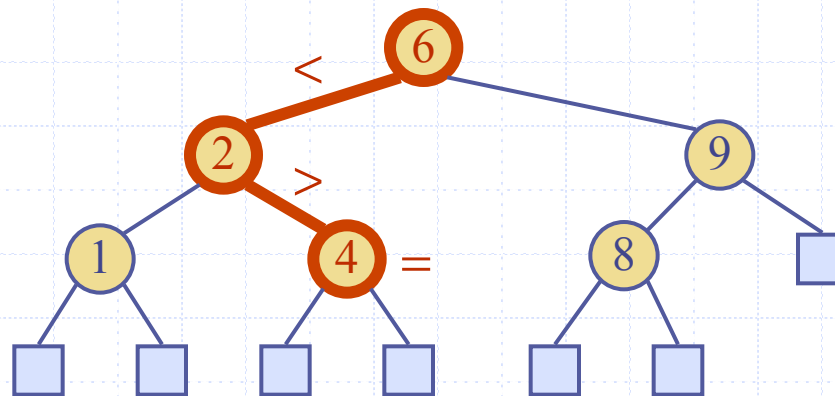
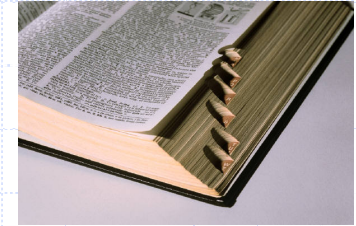


Binary Search Trees

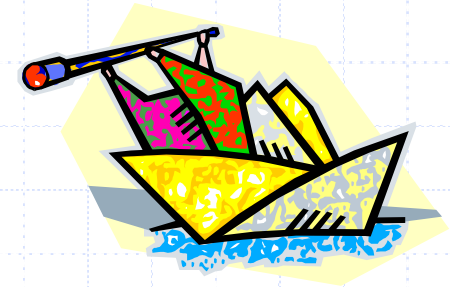


Ordered Maps

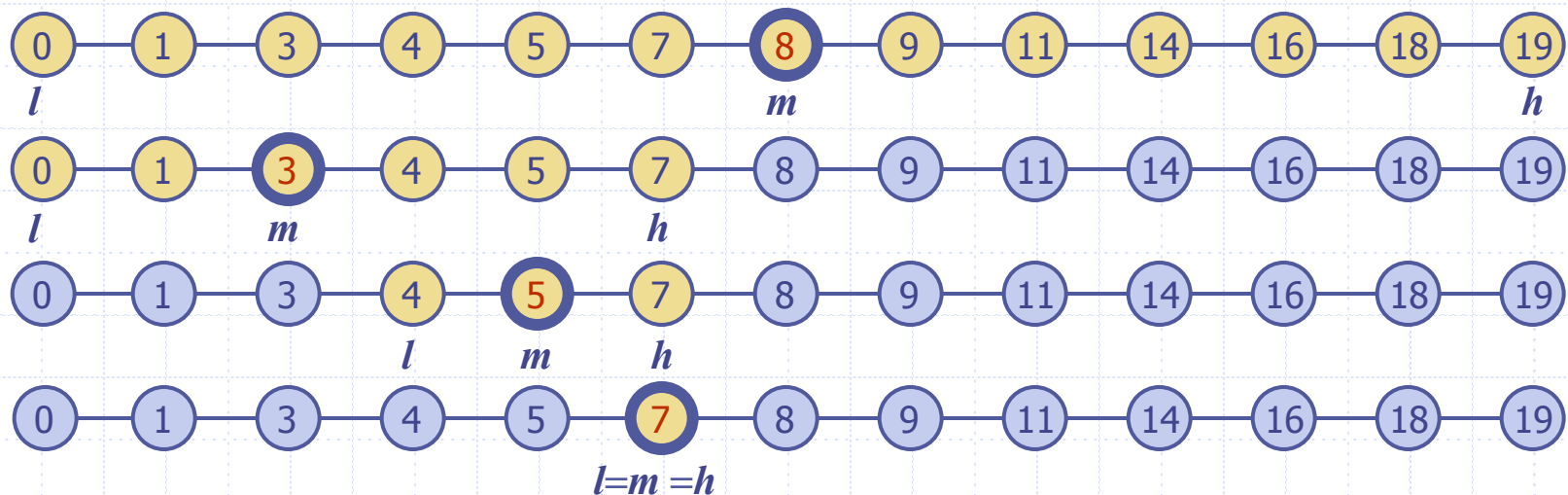


- ◆ Keys are assumed to come from a total order.
- ◆ Items are stored in order by their keys
- ◆ This allows us to support nearest neighbor queries:
 - ◆ Item with largest key less than or equal to k
 - ◆ Item with smallest key greater than or equal to k

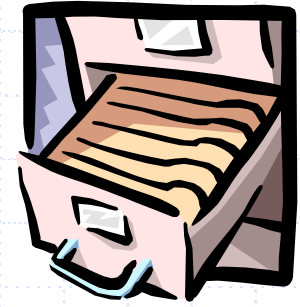
Binary Search



- ◆ 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)

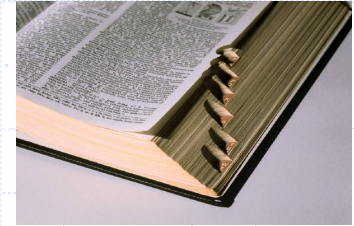


Search Tables



- ◆ 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 $n/2$ items to make room for the new item
 - Removing an item takes $O(n)$ time, since in the worst case we have to shift $n/2$ 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)

Sorted Map Operations



◆ Standard Map methods:

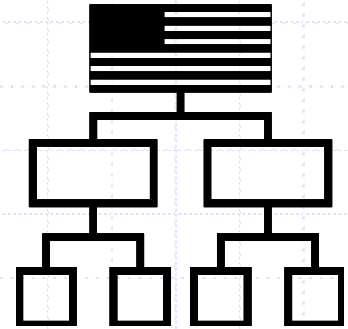
$M[k]$: Return the value v associated with key k in map M , if one exists; otherwise raise a `KeyError`; implemented with `__getitem__` method.

$M[k] = v$: Associate value v with key k in map M , replacing the existing value if the map already contains an item with key equal to k ; implemented with `__setitem__` method.

`del M[k]`: Remove from map M the item with key equal to k ; if M has no such item, then raise a `KeyError`; implemented with `__delitem__` method.

◆ The sorted map ADT includes additional functionality, guaranteeing that an iteration reports keys in sorted order, and supporting additional searches such as `find_gt(k)` and `find_range(start, stop)`.

Binary Search Trees

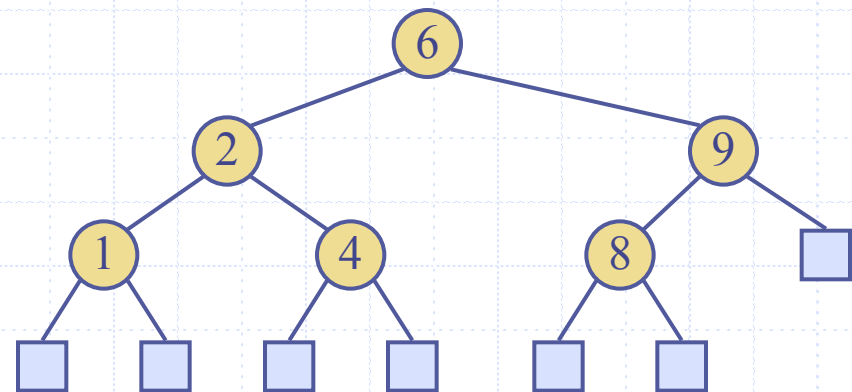


- ◆ A binary search tree is a binary tree storing keys (or key-value items) at its nodes and satisfying the following property:

- 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 tree visits the keys in increasing order

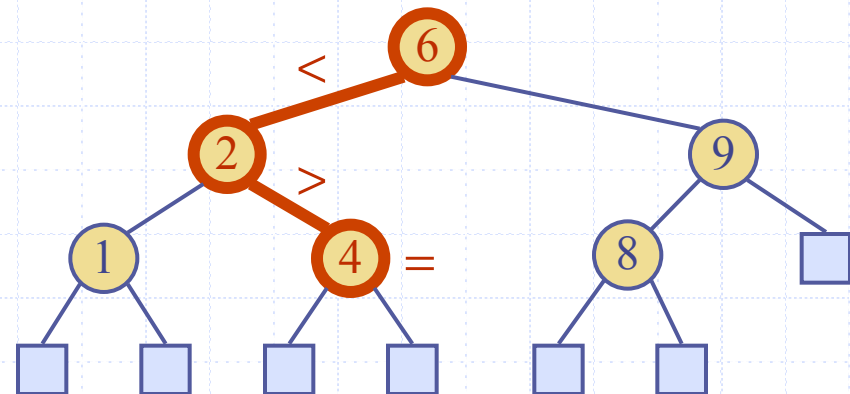


Search

- ◆ 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

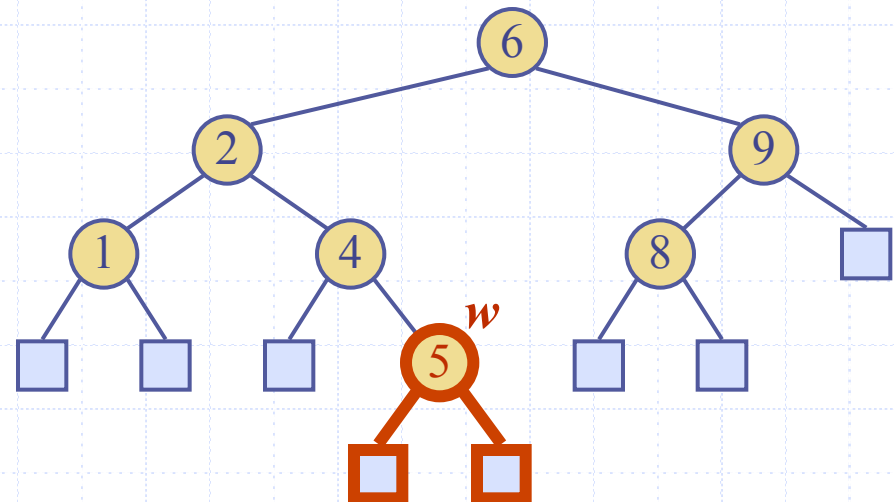
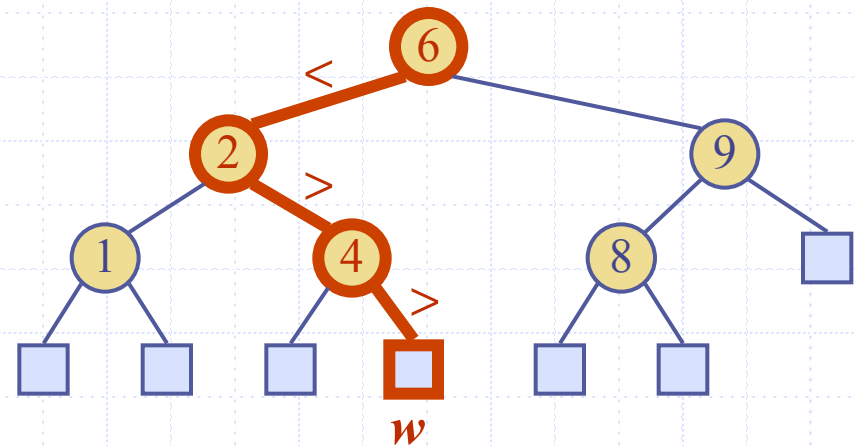
Algorithm `TreeSearch(T, p, k):`

```
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



Insertion Pseudo-code

Algorithm TreeInsert(T, k, v):

Input: A search key k to be associated with value v

$p = \text{TreeSearch}(T, T.\text{root}(), k)$

if $k == p.\text{key}()$ **then**

 Set p 's value to v

else if $k < p.\text{key}()$ **then**

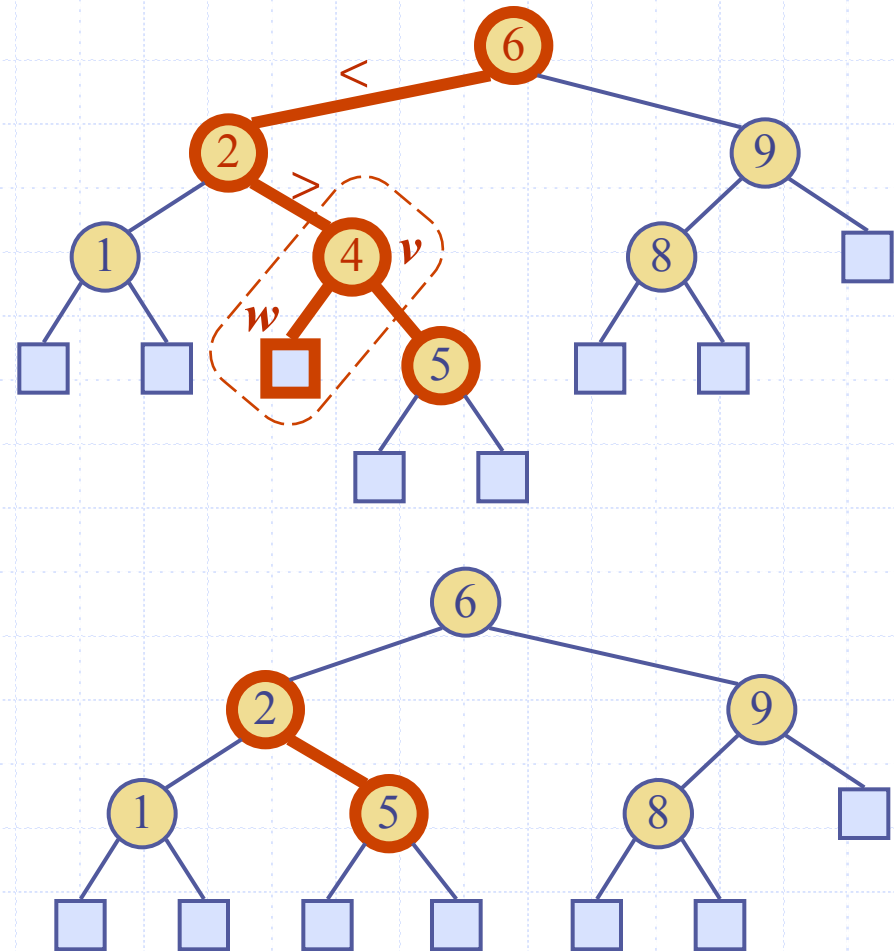
 add node with item (k, v) as left child of p

else

 add node with item (k, v) as right child of p

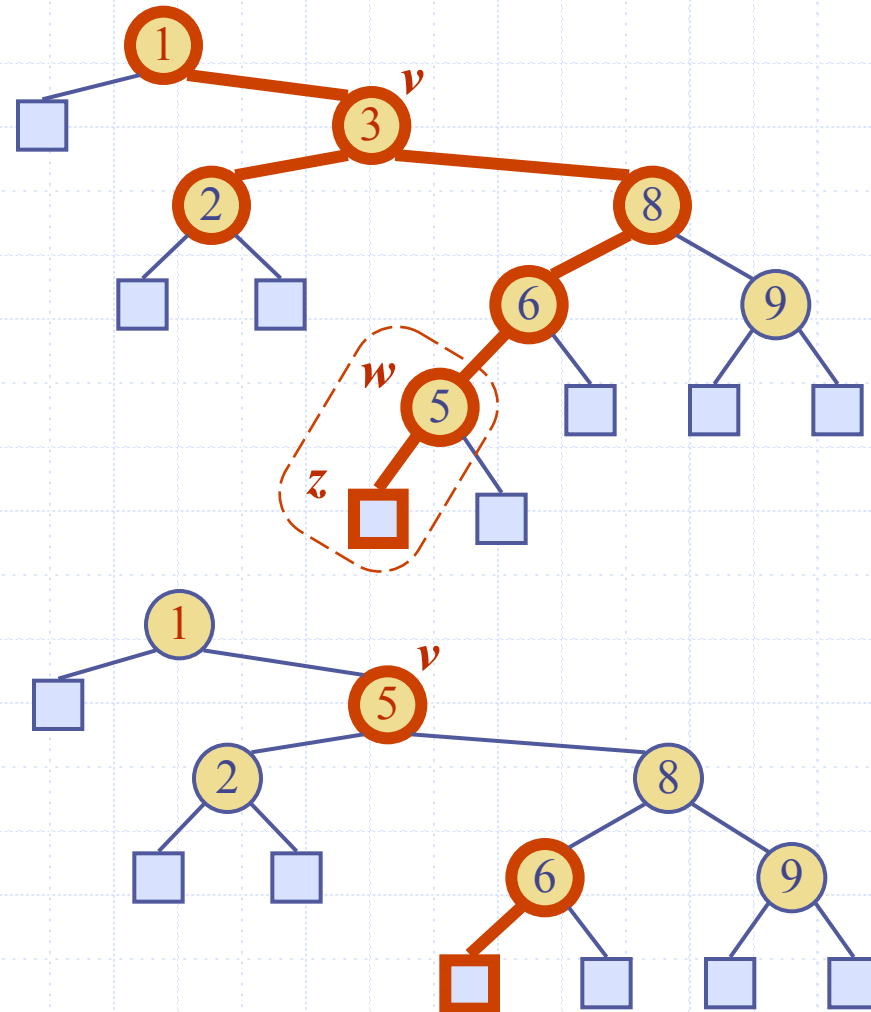
Deletion

- ◆ 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



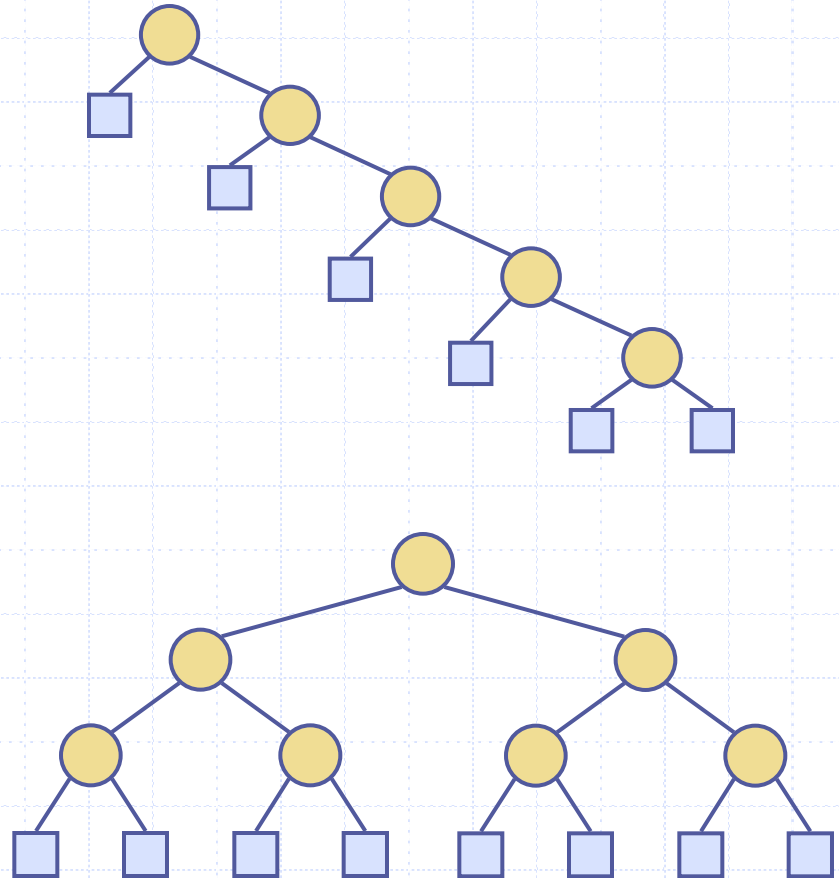
Deletion (cont.)

- ◆ 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



Python Implementation

```
1 class TreeMap(LinkedBinaryTree, MapBase):
2     """Sorted map implementation using a binary search tree."""
3
4     #----- override Position class -----
5     class Position(LinkedBinaryTree.Position):
6         def key(self):
7             """Return key of map's key-value pair."""
8             return self.element()._key
9
10        def value(self):
11            """Return value of map's key-value pair."""
12            return self.element()._value
13
14        #----- nonpublic utilities -----
15        def _subtree_search(self, p, k):
16            """Return Position of p's subtree having key k, or last node searched."""
17            if k == p.key():
18                return p
19            elif k < p.key():
20                if self.left(p) is not None:
21                    return self._subtree_search(self.left(p), k)
22            else:
23                if self.right(p) is not None:
24                    return self._subtree_search(self.right(p), k)
25            return p
26
27        def _subtree_first_position(self, p):
28            """Return Position of first item in subtree rooted at p."""
29            walk = p
30            while self.left(walk) is not None:
31                walk = self.left(walk)
32            return walk
33
34        def _subtree_last_position(self, p):
35            """Return Position of last item in subtree rooted at p."""
36            walk = p
37            while self.right(walk) is not None:
38                walk = self.right(walk)
39            return walk
```

Python Implementation, Part 2

```
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
64
65 def after(self, p):
66     """Return the Position just after p in the natural order.
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 neighbor (or None if empty)."""
74     if self.is_empty():
75         return None
76     else:
77         p = self._subtree_search(self.root(), k)
78         self._rebalance_access(p) # hook for balanced tree subclasses
79         return p
```

Python Implementation, Part 3

```
80 def find_min(self):
81     """Return (key,value) pair with minimum key (or None if empty)."""
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 equal to k.
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 find exact match
97         if p.key() < k:                     # p's key is too small
98             p = self.after(p)
99         return (p.key(), p.value()) if p is not None else None
100
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 map.
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 Implementation, Part 4

```
118 def __getitem__(self, k):
119     """Return value associated with key k (raise KeyError if not found)."""
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 for balanced tree subclasses
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 value if present."""
131     if self.is_empty():
132         leaf = self._add_root(self._Item(k,v)) # from LinkedBinaryTree
133     else:
134         p = self._subtree_search(self.root(), k)
135         if p.key() == k:
136             p.element()._value = v # replace existing item's value
137             self._rebalance_access(p) # hook for balanced tree subclasses
138             return
139         else:
140             item = self._Item(k,v)
141             if p.key() < k:
142                 leaf = self._add_right(p, item) # inherited from LinkedBinaryTree
143             else:
144                 leaf = self._add_left(p, item) # inherited from LinkedBinaryTree
145             self._rebalance_insert(leaf) # hook for balanced tree subclasses
146
147 def __iter__(self):
148     """Generate an iteration of all keys in the map in order."""
149     p = self.first()
150     while p is not None:
151         yield p.key()
152         p = self.after(p)
```


Python Implementation, end

```
153 def delete(self, p):
154     """Remove the item at given Position."""
155     self._validate(p)                # inherited from LinkedBinaryTree
156     if self.left(p) and self.right(p): # p has two children
157         replacement = self._subtree_last_position(self.left(p))
158         self._replace(p, replacement.element()) # from LinkedBinaryTree
159         p = replacement
160         # now p has at most one child
161     parent = self.parent(p)
162     self._delete(p)                  # inherited from LinkedBinaryTree
163     self._rebalance_delete(parent)   # if root deleted, parent is None
164
165 def __delitem__(self, k):
166     """Remove item associated with key k (raise KeyError if not found)."""
167     if not self.is_empty():
168         p = self._subtree_search(self.root(), k)
169         if k == p.key():
170             self.delete(p)            # rely on positional version
171             return                    # successful deletion complete
172         self._rebalance_access(p)     # hook for balanced tree subclasses
173     raise KeyError('Key Error: ' + repr(k))
```