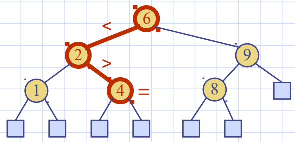
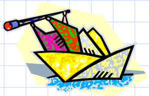
Binary Search Trees



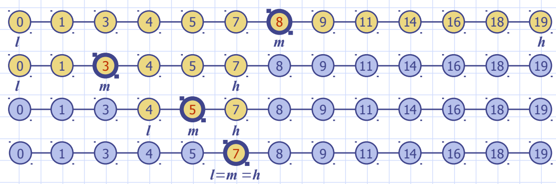
Ordered Maps

- 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)



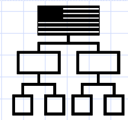
Search Tables

- 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)

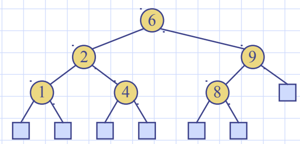
Sorted Map Operations

- 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



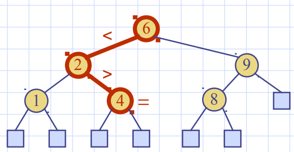
- A binary search tree is a binary tree storing keys (or key-value items) at its nodes and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not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



Search

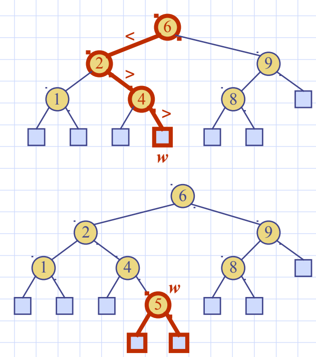
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf, the key is not found
- Example: find(4):
  - Call TreeSearch(4,root)
- The algorithms for nearest neighbor queries are similar

```
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

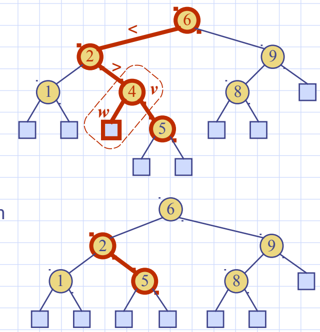


Insertion Pseudo-code

```
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
```

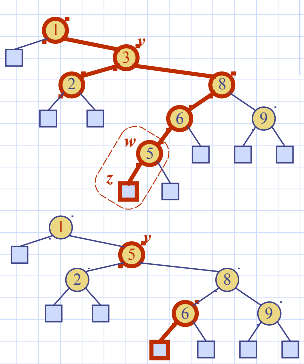

Deletion

- To perform operation remove(k), we search for key k
- Assume key k is in the tree, and let 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



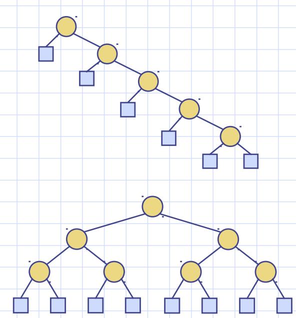
Deletion (cont.)

- 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

```
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

  #------------------------------- nonpublic utilities -------------------------------
  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, Part 2

```
  def first(self):
    """Return the first Position in the tree (or None if empty)."""
    return self._subtree_first_position(self.root()) if len(self) > 0 else None

  def last(self):
    """Return the last Position in the tree (or None if empty)."""
    return self._subtree_last_position(self.root()) if len(self) > 0 else None

  def before(self, p):
    """Return the Position just before p in the natural order.

    Return None if p is the first position.
    """
    self._validate(p)                          # inherited from LinkedBinaryTree
    if self.left(p):
      return self._subtree_last_position(self.left(p))
    else:
      # walk upward
      walk = p
      above = self.parent(walk)
      while above is not None and walk == self.left(above):
        walk = above
        above = self.parent(walk)
      return above

  def after(self, p):
    """Return the Position just after p in the natural order.

    Return None if p is the last position.
    """
    # symmetric to before(p)

  def find_position(self, k):
    """Return position with key k, or else neighbor (or None if empty)."""
    if self.is_empty():
      return None
    else:
      p = self._subtree_search(self.root(), k)
      self._rebalance_access(p)                # hook for balanced tree subclasses
      return p
```


Python Implementation, Part 3

```
  def find_min(self):
    """Return (key,value) pair with minimum key (or None if empty)."""
    if self.is_empty():
      return None
    else:
      p = self.first()
      return (p.key(), p.value())

  def find_ge(self, k):
    """Return (key,value) pair with least key greater than or equal to k.

    Return None if there does not exist such a key.
    """
    if self.is_empty():
      return None
    else:
      p = self.find_position(k)                # may not find exact match
      if p.key() < k:                          # p's key is too small
        p = self.after(p)
      return (p.key(), p.value()) if p is not None else None

  def find_range(self, start, stop):
    """Iterate all (key,value) pairs such that start <= key < stop.

    If start is None, iteration begins with minimum key of map.
    If stop is None, iteration continues through the maximum key of map.
    """
    if not self.is_empty():
      if start is None:
        p = self.first()
      else:
        # we initialize p with logic similar to find_ge
        p = self.find_position(start)
        if p.key() < start:
          p = self.after(p)
      while p is not None and (stop is None or p.key() < stop):
        yield (p.key(), p.value())
        p = self.after(p)
```


Python Implementation, Part 4

```
  def __getitem__(self, k):
    """Return value associated with key k (raise KeyError if not found)."""
    if self.is_empty():
      raise KeyError('Key Error: ' + repr(k))
    else:
      p = self._subtree_search(self.root(), k)
      self._rebalance_access(p)                # hook for balanced tree subclasses
      if k != p.key():
        raise KeyError('Key Error: ' + repr(k))
      return p.value()

  def __setitem__(self, k, v):
    """Assign value v to key k, overwriting existing value if present."""
    if self.is_empty():
      leaf = self._add_root(self._Item(k,v))           # from LinkedBinaryTree
    else:
      p = self._subtree_search(self.root(), k)
      if p.key() == k:
        p.element()._value = v                 # replace existing item's value
        self._rebalance_access(p)              # hook for balanced tree subclasses
        return
      else:
        item = self._Item(k,v)
        if p.key() < k:
          leaf = self._add_right(p, item)      # inherited from LinkedBinaryTree
        else:
          leaf = self._add_left(p, item)       # inherited from LinkedBinaryTree
    self._rebalance_insert(leaf)               # hook for balanced tree subclasses

  def __iter__(self):
    """Generate an iteration of all keys in the map in order."""
    p = self.first()
    while p is not None:
      yield p.key()
      p = self.after(p)
```

Python Implementation, end

```
def delete(self, p):
  """Remove the item at given Position."""
  self._validate(p)                            # inherited from LinkedBinaryTree
  if self.left(p) and self.right(p):           # p has two children
    replacement = self._subtree_last_position(self.left(p))
    self._replace(p, replacement.element())    # from LinkedBinaryTree
    p = replacement
  # now p has at most one child
  parent = self.parent(p)
  self._delete(p)                              # inherited from LinkedBinaryTree
  self._rebalance_delete(parent)               # if root deleted, parent is None

def __delitem__(self, k):
  """Remove item associated with key k (raise KeyError if not found)."""
  if not self.is_empty():
    p = self._subtree_search(self.root(), k)
    if k == p.key():
      self.delete(p)                           # rely on positional version
      return                                   # successful deletion complete
    self._rebalance_access(p)                  # hook for balanced tree subclasses
  raise KeyError('Key Error: ' + repr(k))
```