SE274 Data Structure

Lecture 8: Search Trees

(textbook: Chapter 11)

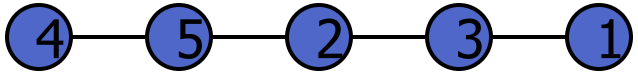
Apr 20, 2020

Instructor: Sunjun Kim

Information&Communication Engineering, DGIST

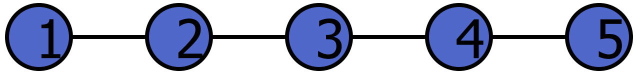
Recap: Insertion sort, Selection Sort

- **Selection sort**: Priority Queue Implementation with an unsorted list



- Performance:
  - add takes $O(1)$ time since we can insert the item at the beginning or end of the sequence
  - Remove_min and min take $O(n)$ time since we have to traverse the entire sequence to find the smallest key

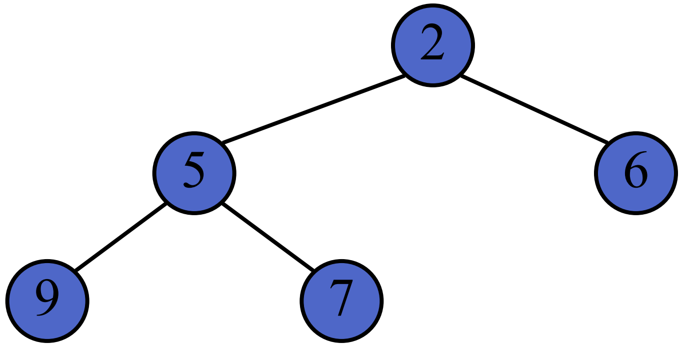
- **Insertion sort**: Priority Queue Implementation with a sorted list



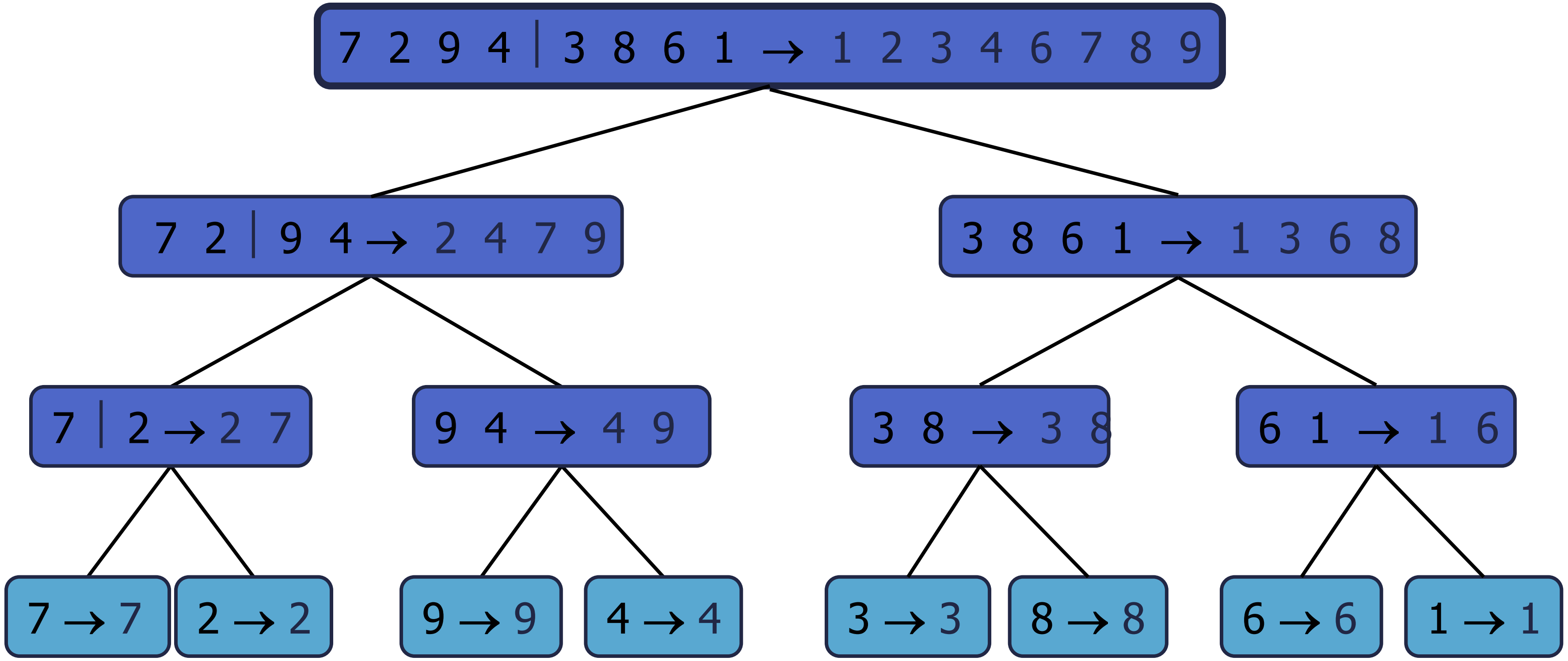
- Performance:
  - add takes $O(n)$ time since we have to find the place where to insert the item
  - remove_min and min take $O(1)$ time, since the smallest key is at the beginning

Recap: heapsort

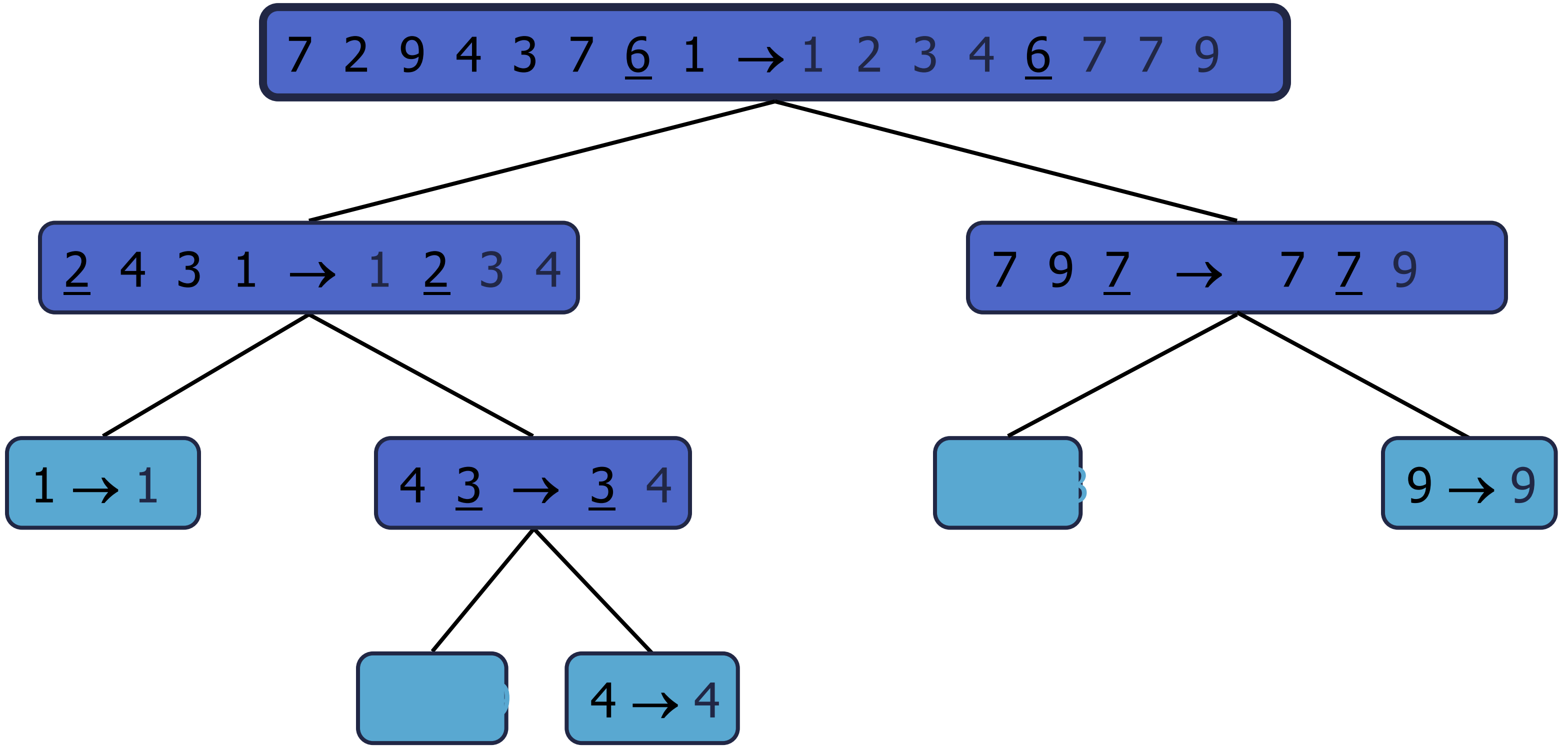
- ***Add*** all the items to a heap.
- Repeat ***Remove_min*** until the heap is empty.
- ***Add*** takes $\boldsymbol{O}(log\ \boldsymbol{n})$ time, remove_min take $\boldsymbol{O}(log\ \boldsymbol{n})$ time.
- Heapsort takes $\boldsymbol{O}(\boldsymbol{n}\ log\ \boldsymbol{n})$



Recap: Merge Sort

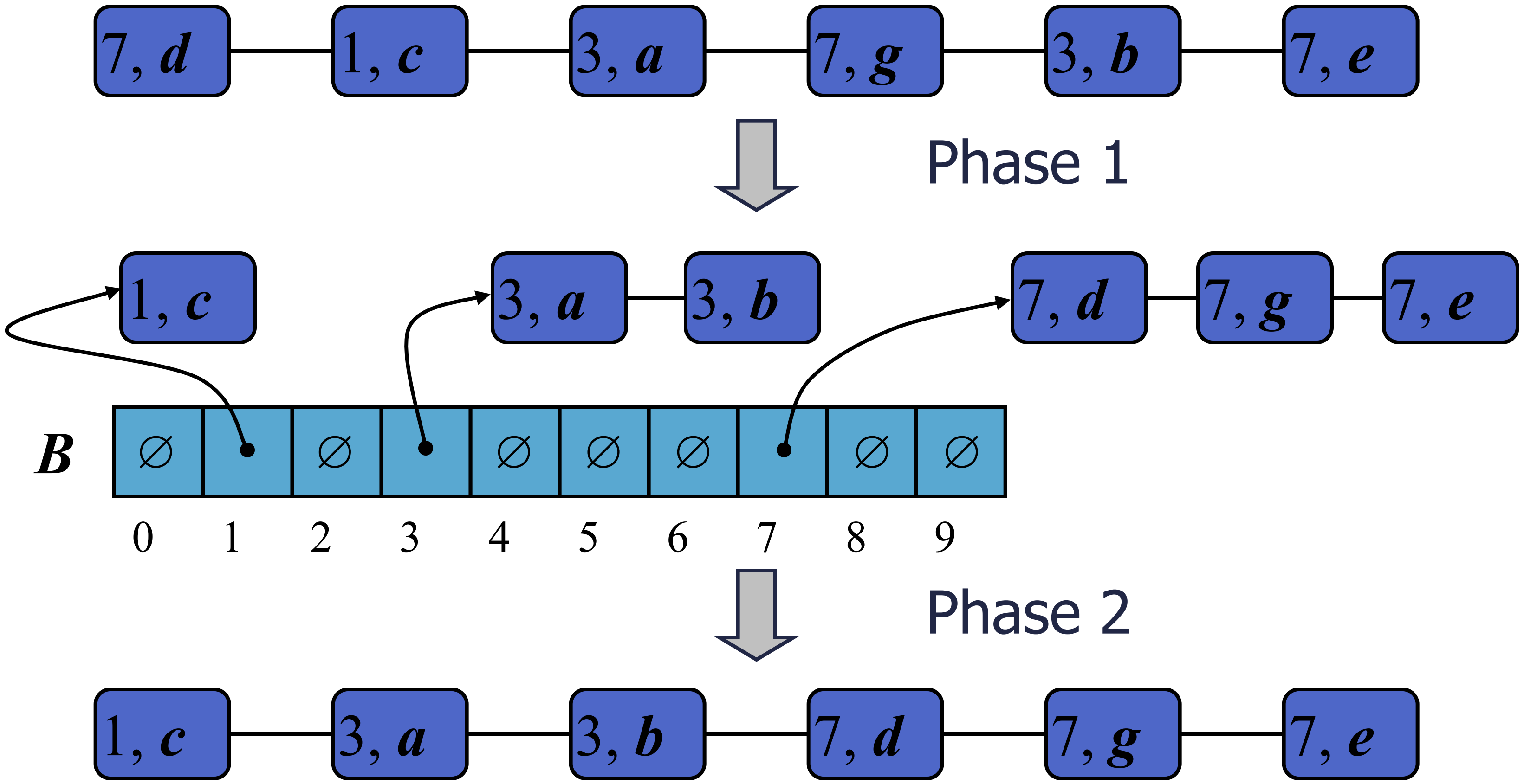


Recap: Quick Sort



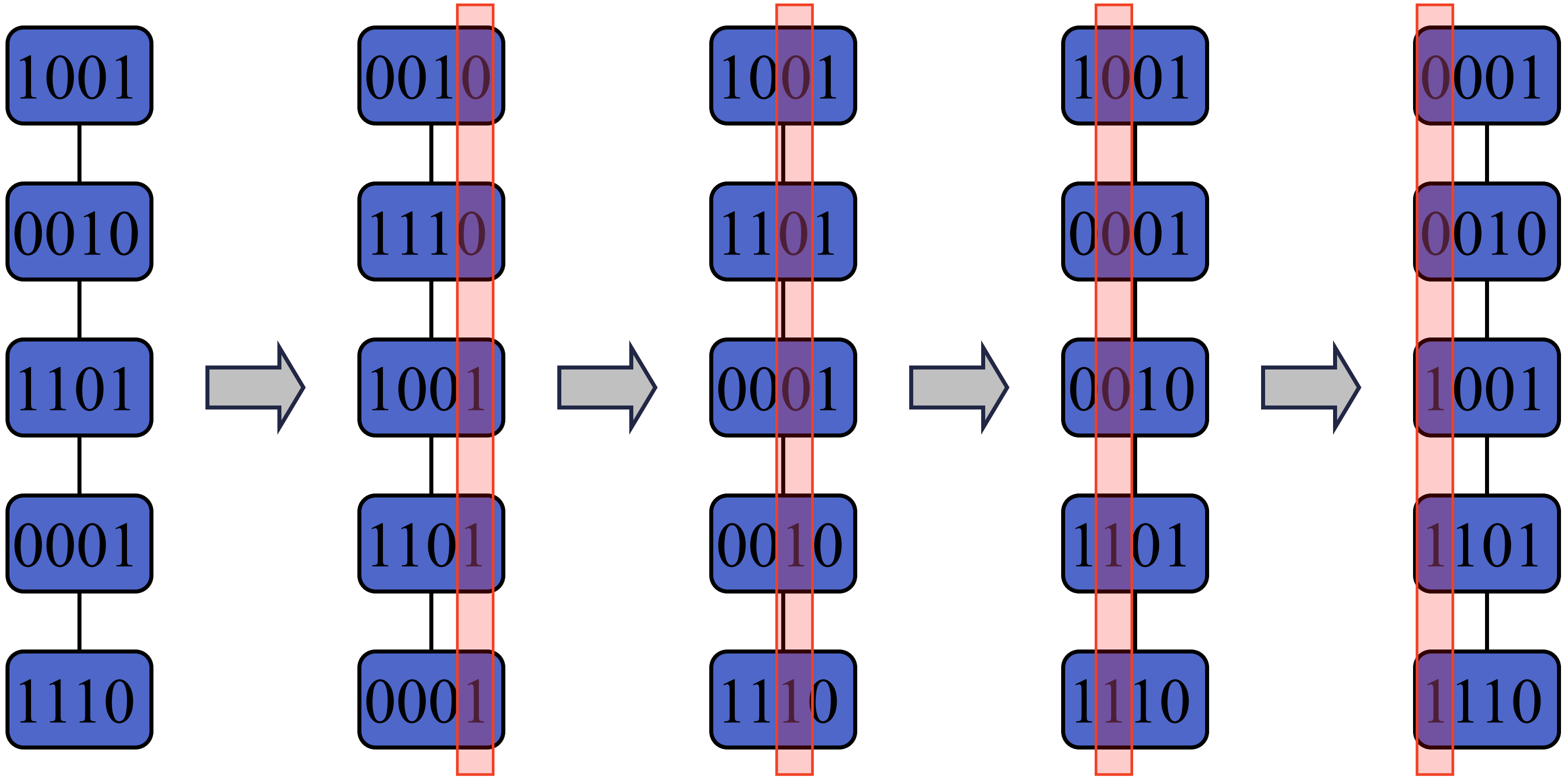
Recap: Bucket Sort

- Key range $[0, 9]$



Recap: Radix Sort

- Sorting a sequence of 4-bit integers



Recap: Algorithm time complexity

Algorithm	Time	Notes
selection-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
insertion-sort	$O(n^2)$	▪ in-place ▪ slow (good for small inputs)
quick-sort	$O(n \log n)$	▪ in-place, randomized ▪ fastest (good for large inputs)
heap-sort	$O(n \log n)$	▪ in-place ▪ fast (good for large inputs)
merge-sort	$O(n \log n)$	▪ sequential data access ▪ fast (good for huge inputs)
bucket-sort radix-sort	$O(n)$	▪ N (# of possible values) should be small ▪ applicable when N $\approx$ n (or N < n)

Selection

The Selection Problem

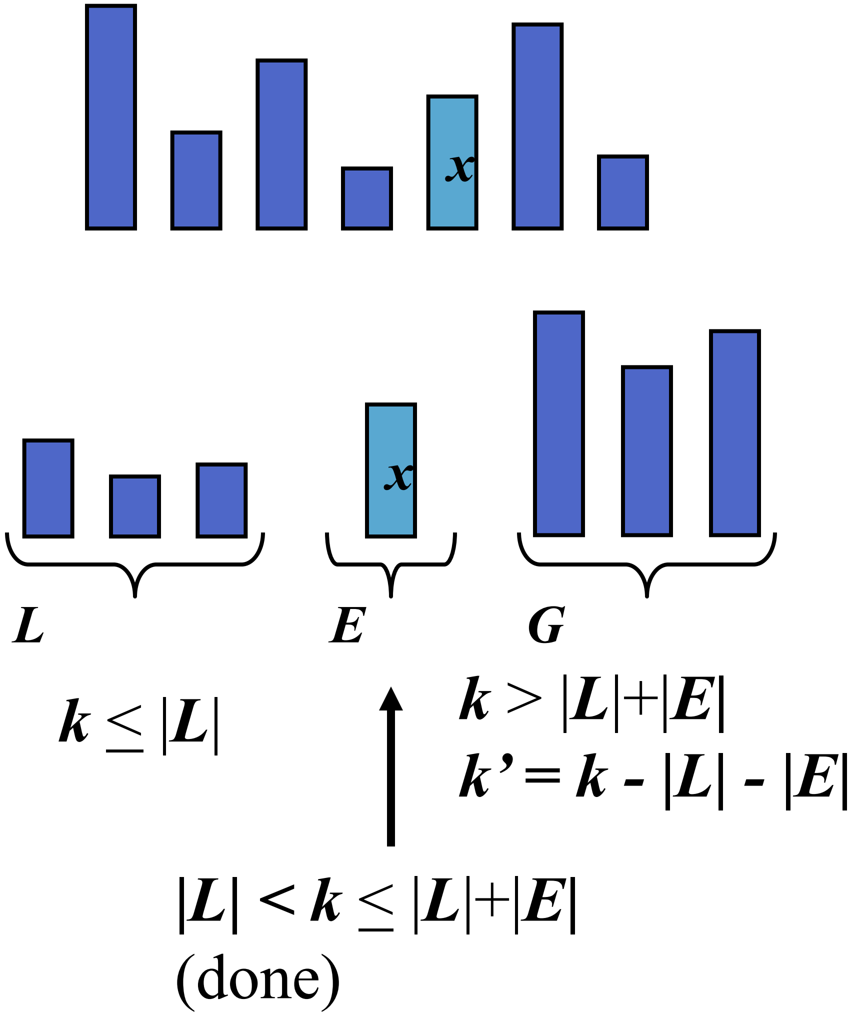
- Given an integer k and n elements $x_1, x_2, \ldots, x_n$, taken from a total order, find the k-th smallest element in this set.
- Of course, we can sort the set in O(n log n) time and then index the k-th element.

k=3 7 4 9 <u>6</u> 2 → 2 4 <u>6</u> 7 9

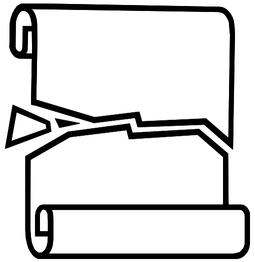
- Can we solve the selection problem faster?

Quick-Select

- Quick-select is a randomized selection algorithm based on the prune-and-search (or decrease-and-conquer) paradigm:
  - Prune: pick a random element x (called pivot) and partition S into
    - L: elements less than x
    - E: elements equal x
    - G: elements greater than x
  - Search: depending on k, either answer is in E, or we need to recur in either L or G



Partition

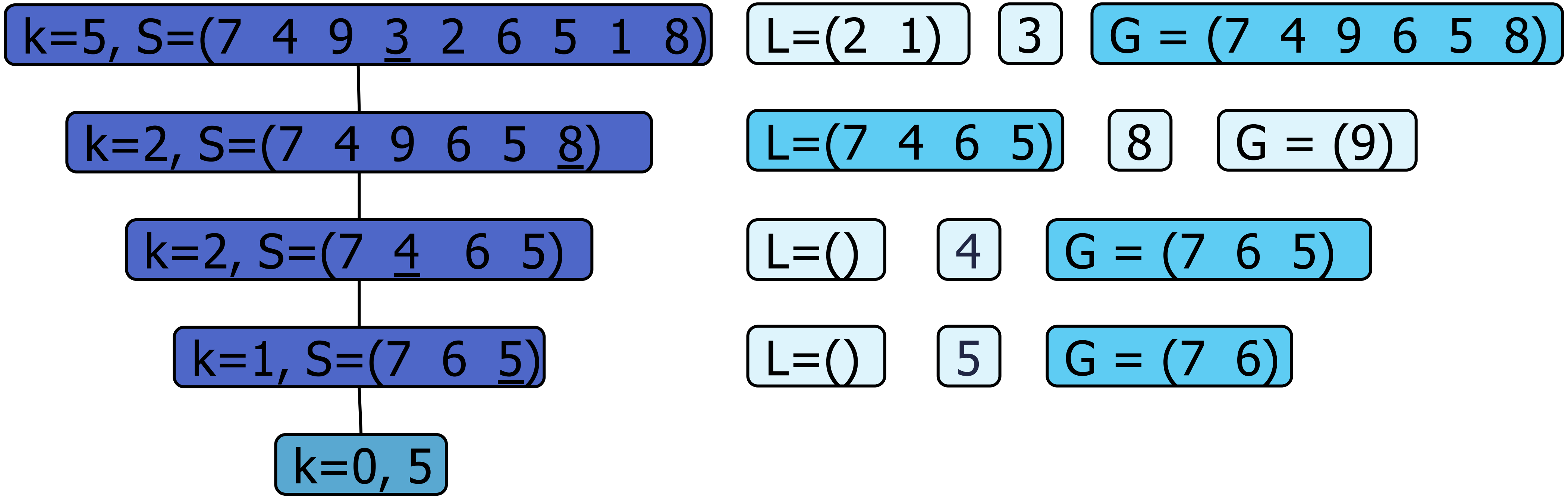


- We partition an input sequence as in the quick-sort algorithm:
  - We remove, in turn, each element y from S and
  - We insert y into L, E or G, depending on the result of the comparison with the pivot x
- Each insertion and removal is at the beginning or at the end of a sequence, and hence takes $O(1)$ time
- Thus, the partition step of quick-select takes $O(n)$ time

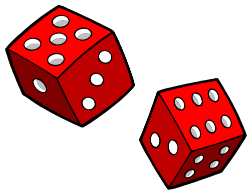
Algorithm *partition*(S, p)
    Input sequence S, position p of pivot
    Output subsequences L, E, G of the elements of S less than, equal to, or greater than the pivot, resp.
    $L, E, G \leftarrow$ empty sequences
    $x \leftarrow S.remove(p)$
    while $\neg S.isEmpty()$
        $y \leftarrow S.remove(S.first())$
        if $y < x$
            $L.addLast(y)$
        else if $y = x$
            $E.addLast(y)$
        else $\{ y > x \}$
            $G.addLast(y)$
    return L, E, G

Quick-Select Visualization

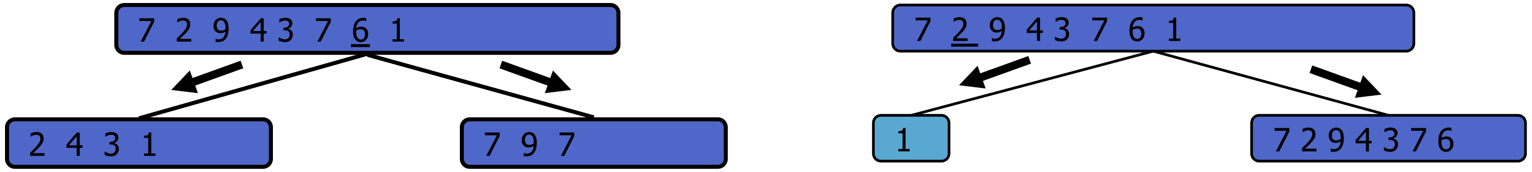
- An execution of quick-select can be visualized by a recursion path
  - Each node represents a recursive call of quick-select, and stores k and the remaining sequence



Expected Running Time

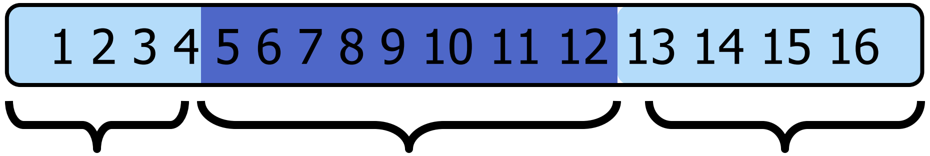


- Consider a recursive call of quick-select on a sequence of size s
  - Good call: the sizes of L and G are each less than $3s/4$
  - Bad call: one of L and G has size greater than $3s/4$



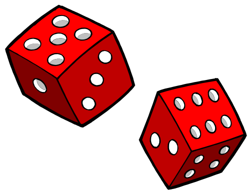
Good call **Bad call**

- A call is good with probability $1/2$
  - 1/2 of the possible pivots cause good calls:



Bad pivots **Good pivots** **Bad pivots**

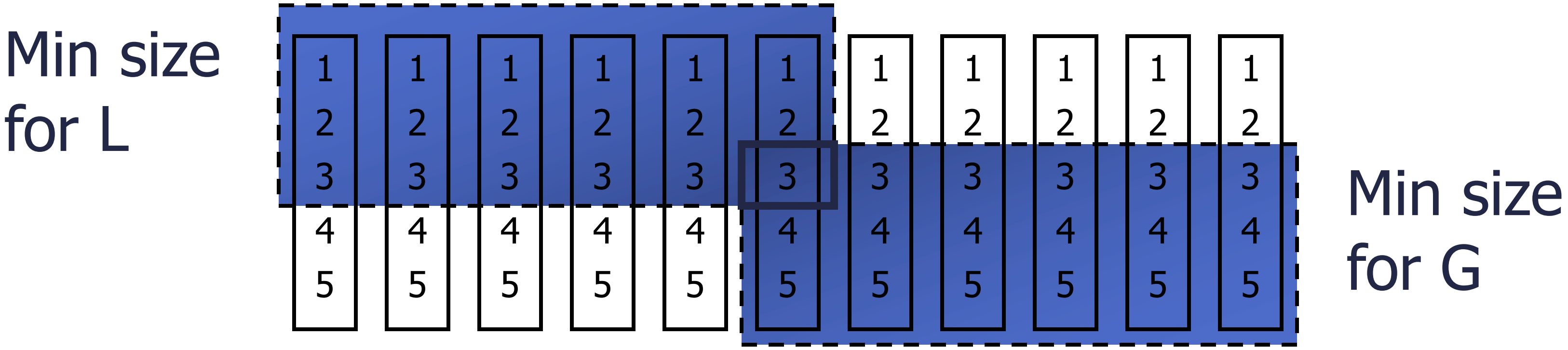
Expected Running Time, Part 2



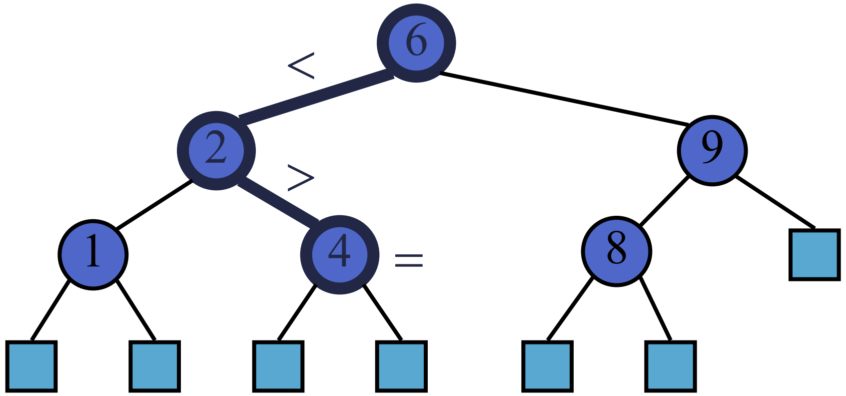
- Probabilistic Fact #1: The expected number of coin tosses required in order to get one head is two
- Probabilistic Fact #2: Expectation is a linear function:
  - $E(X+Y) = E(X) + E(Y)$
  - $E(cX) = cE(X)$
- Let T(n) denote the expected running time of quick-select.
- By Fact #2,
  - $T(n) \le T(3n/4) + bn$*(expected # of calls before a good call)
- By Fact #1,
  - $T(n) \le T(3n/4) + 2bn$
- That is, T(n) is a geometric series:
  - $T(n) \le 2bn + 2b(3/4)n + 2b(3/4)^2 n + 2b(3/4)^3 n + \ldots$
- So T(n) is O(n).
- We can solve the selection problem in O(n) expected time.

Deterministic Selection

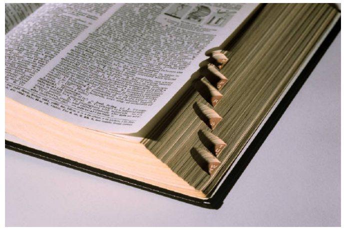
- We can do selection in O(n) worst-case time.
- Main idea: recursively use the selection algorithm itself to find a good pivot for quick-select:
  - Divide S into n/5 sets of 5 each
  - Find a median in each set
  - Recursively find the median of the "baby" medians.



Binary Search Trees



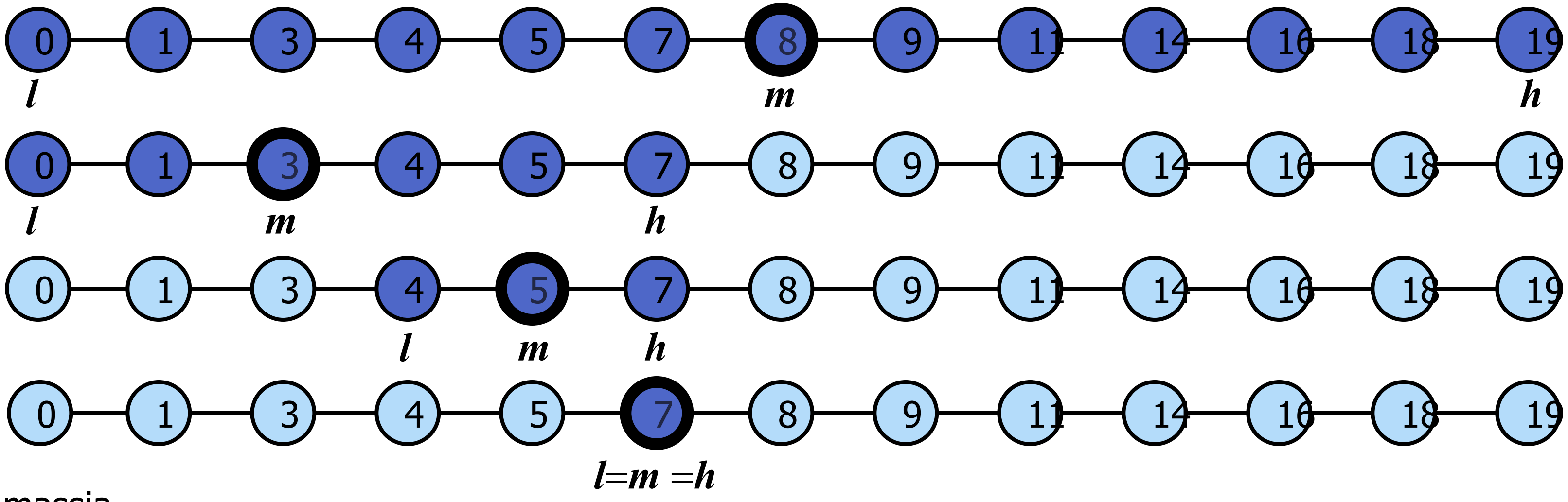
Ordered Maps



- 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

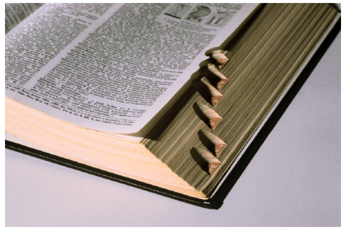
- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - similar to the high-low children's game
  - at each step, the number of candidate items is halved
  - terminates after O(log n) steps
- Example: find(7)



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.

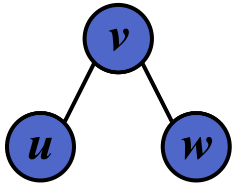
M[k] = v: Associate value v with key k in map M, replacing the existing value if the map already contains an item with key equal to k; implemented with __setitem__ method.

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.

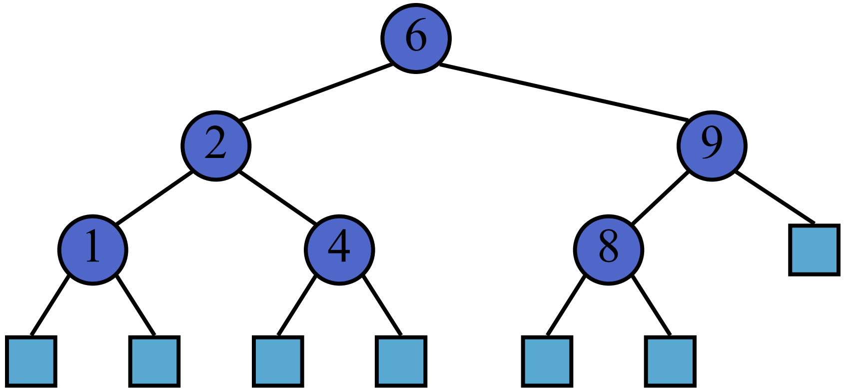
- The sorted map ADT includes additional functionality, guaranteeing that an iteration reports keys in sorted order, and supporting additional searches such as find_gt(k) and find_range(start, stop).

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



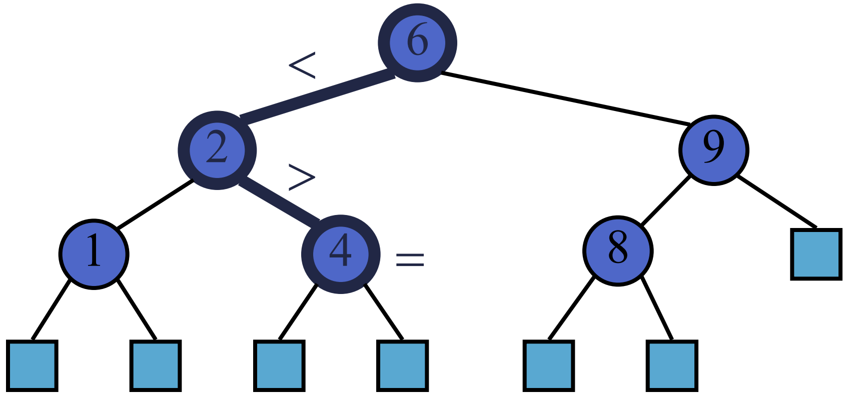
- An inorder traversal of a binary search trees visits the keys in increasing order



Search

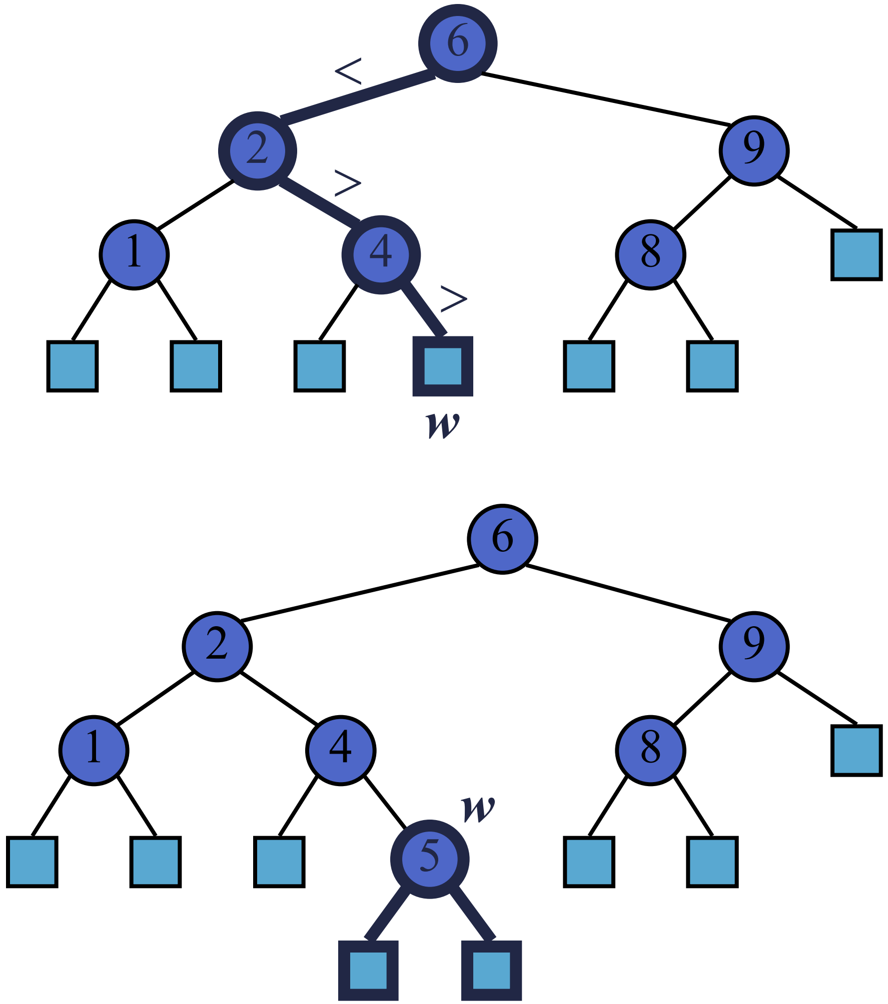
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the comparison of k with the key of the current node
- If we reach a leaf, the key is not found
- Example: find(4):
  - Call TreeSearch(4,root)
- The algorithms for nearest neighbor queries are similar

```
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

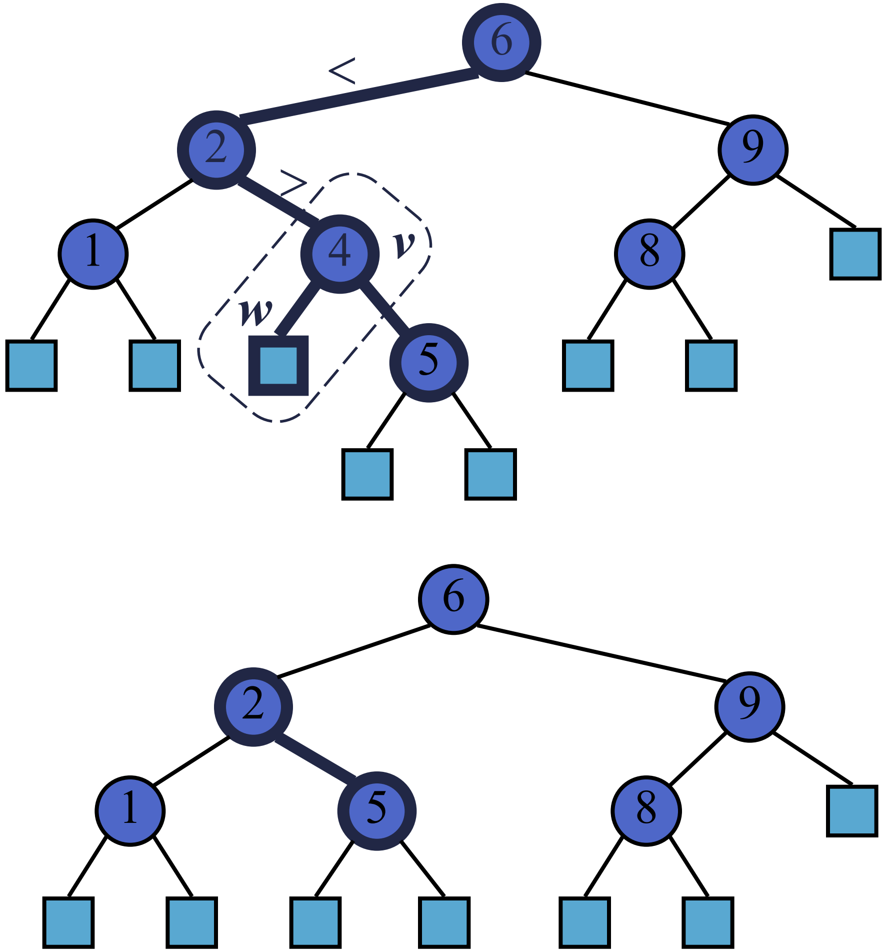
Algorithm TreeInsert(T, k, v):

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

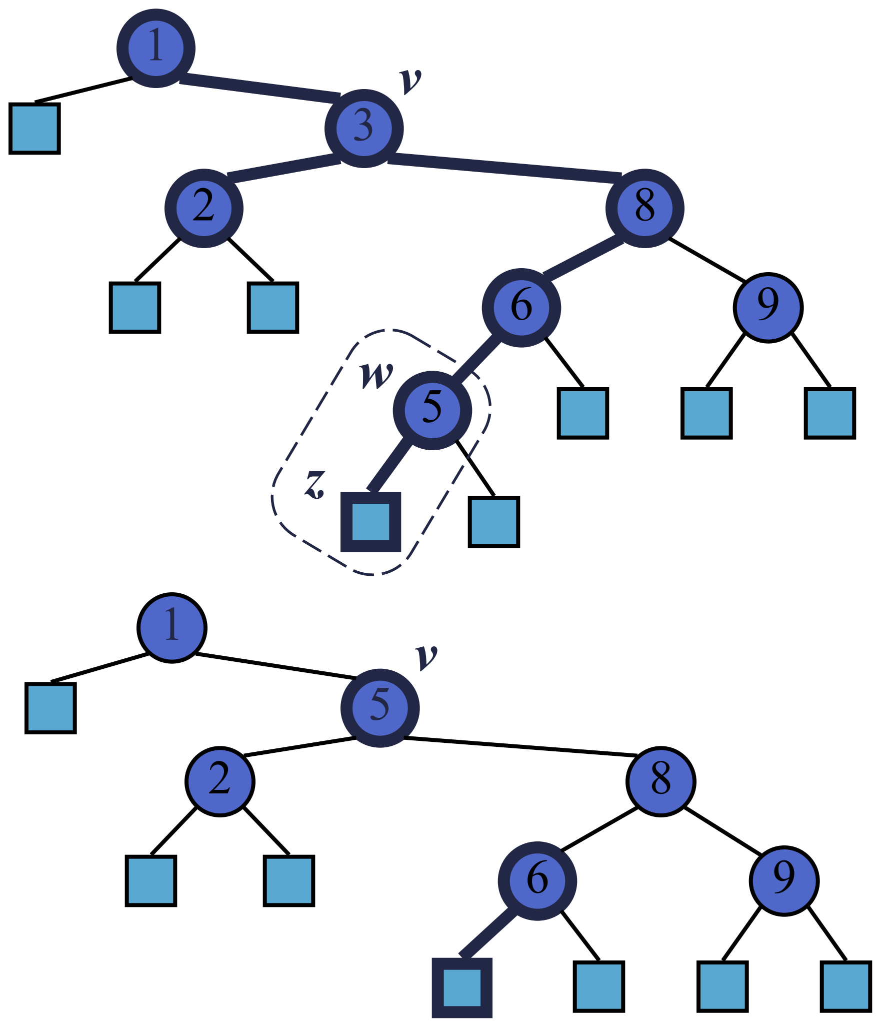
Deletion

- 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



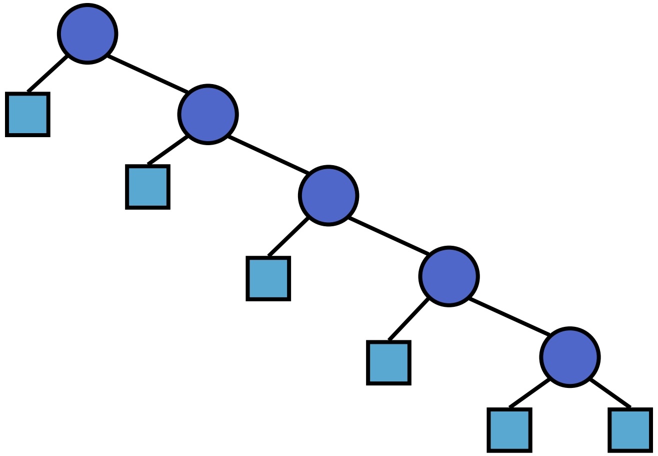
Deletion (cont.)

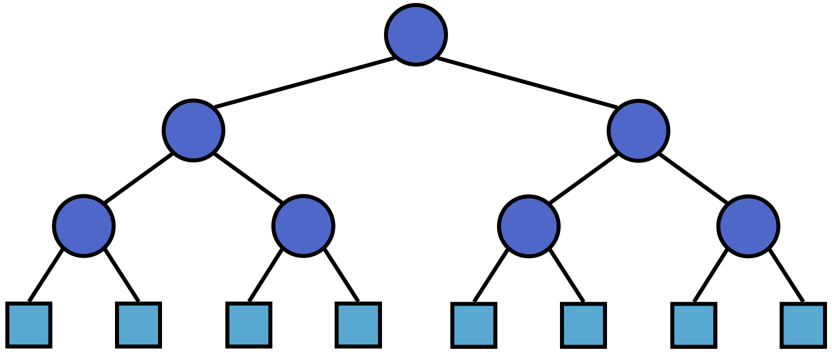
- We consider the case where the key k to be removed is stored at a node v whose children are both internal
    - we find the internal node w that follows v in an inorder traversal
    - we copy $key(w)$ into node v
    - we remove node w and its left child z (which must be a leaf) by means of operation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



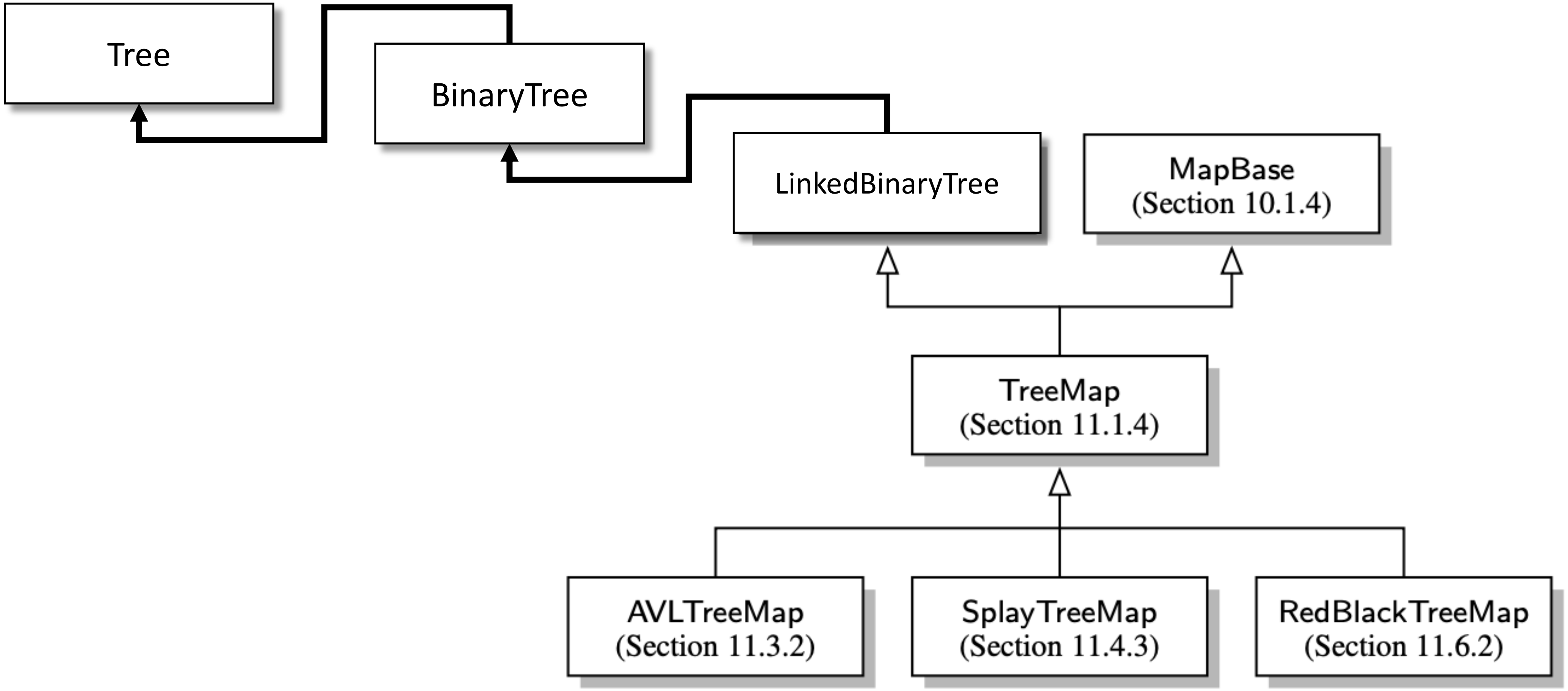


Performance (worst-case running time)

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)	$O(n)$

- h = current height of the tree
- s = # of the requested range

Python framework for Trees



Python Implementation

```python
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)
```

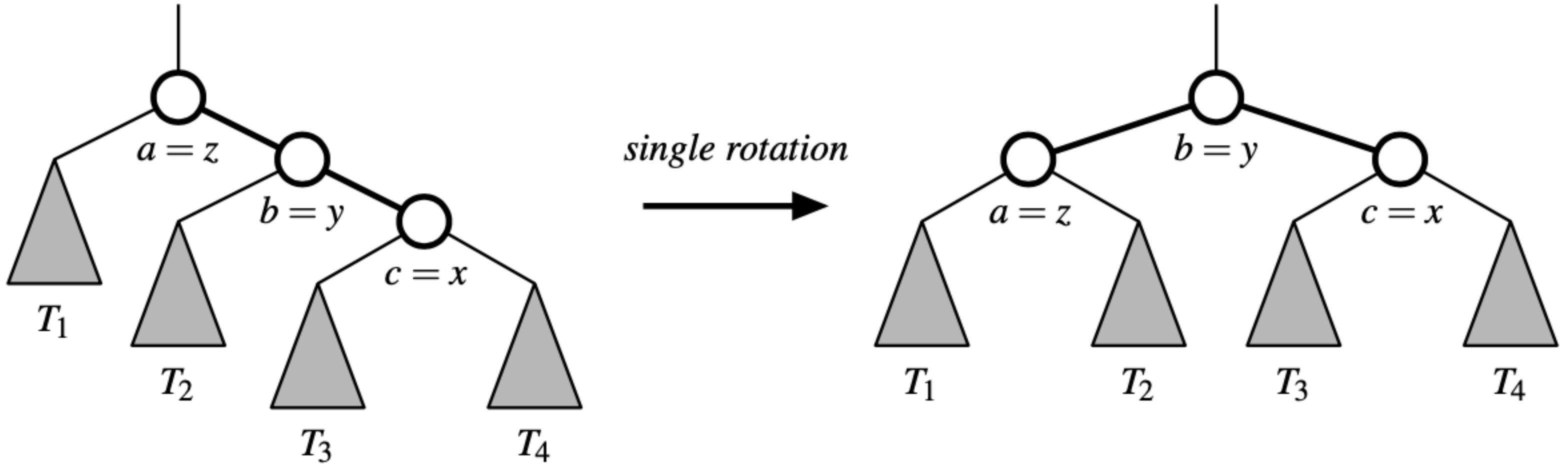
Python Implementation, Part 4

```python
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)
```

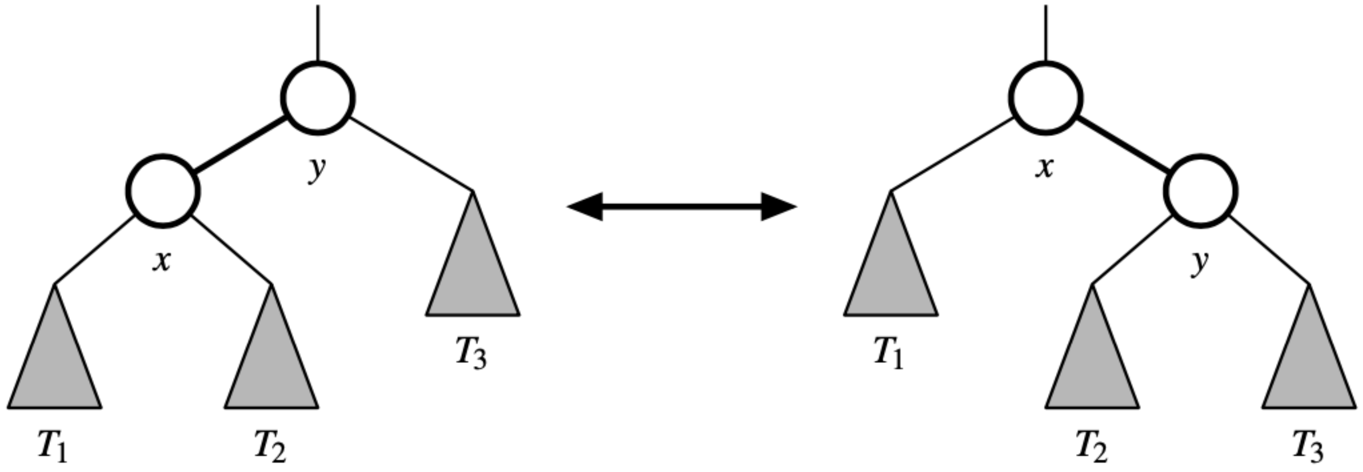
Python Implementation, end

```
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))
```

Balanced Search Tree



Tree Rotation Operation



Trinode reconstruction

- Nodes: (a,b,c) -> inorder listing of the three positions x, y, z
- Sub-trees: (T_1, T_2, T_3, T_4) -> inorder listing of the four subtrees