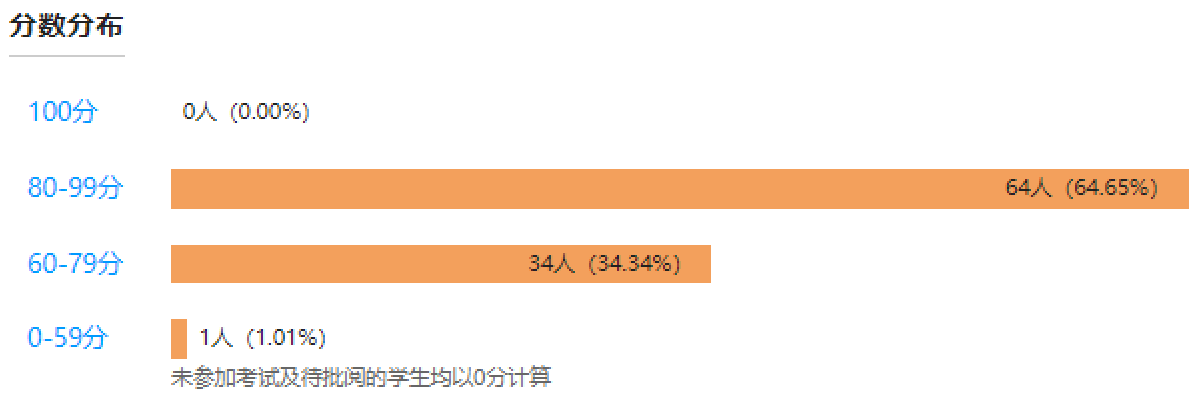
SEARCH TREES

School of Artificial Intelligence

ON EXAMS

- Mid-term
  - One page of key knowledge points
  - Difficulty: moderate
  - Average completion time: 60 minutes
- Final exam
  - Close book
  - Diverse types of questions
  - 50% of your final grade

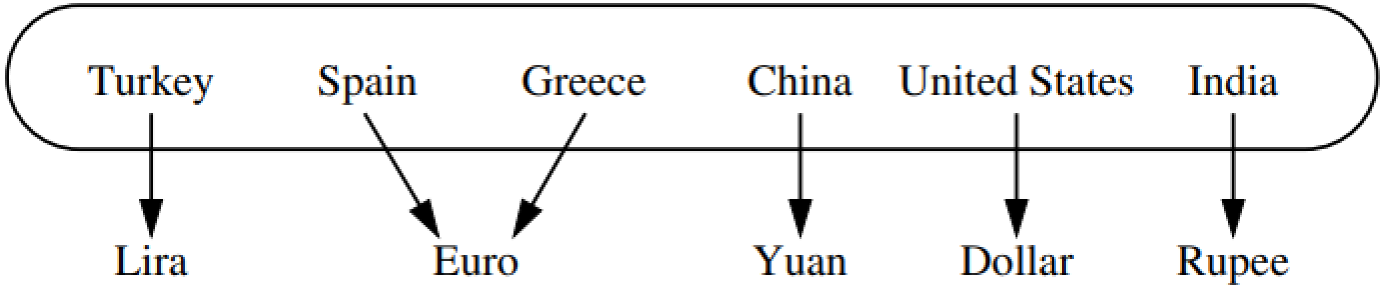


PREVIOUSLY ON DS&A

- Maps and Dictionaries
- Hash tables
- Hash functions
  - Hashing
  - Compressing
- Collision handling
  - Linear probing
  - Quadratic probing
  - Double hashing
- Skip Lists

MAPS AND DICTIONARIES

- Key->Value pairing
  - Unique association
- Most significant data structure in any programming language
- Often known as **associative arrays** or **maps**
- Keys are (assumed) to be unique, values are not necessarily unique
- Python: dict class
- Use key as ‘index’
- Indices need not be consecutive nor numeric



HASH TABLES (哈希表)

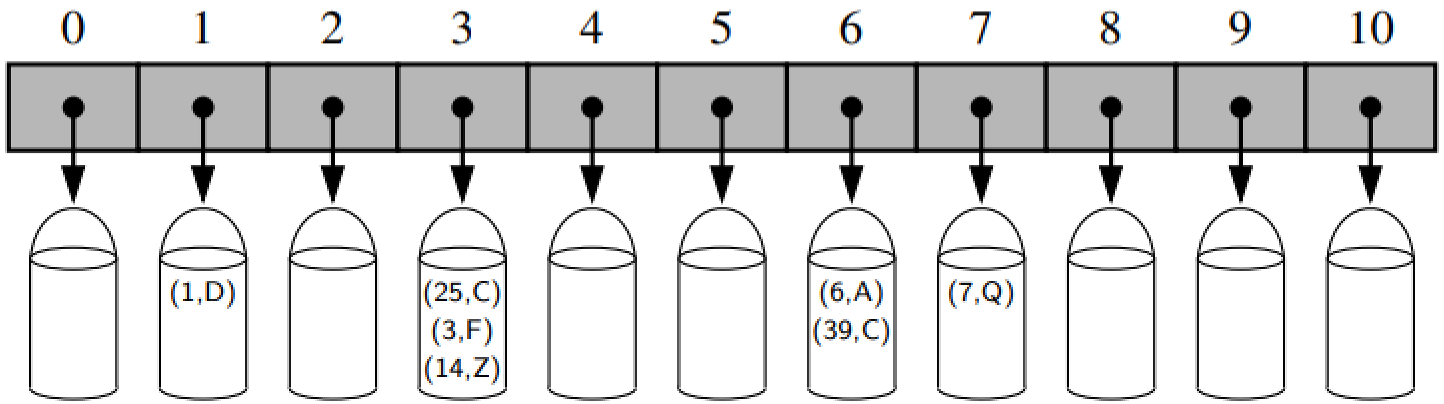
- Most practical data structures for implementing a map
- A map M supports the abstraction of using keys as indices with a syntax such as M[k].
- Assume a map with n items uses integer keys from 0 to N-1 for some N >= n
- We can represent the map like this:

0	1	2	3	4	5	6	7	8	9	10
	D		Z			C	Q			

- __getitem__, __setitem__ and __delitem__ become O(1)
- Problems?

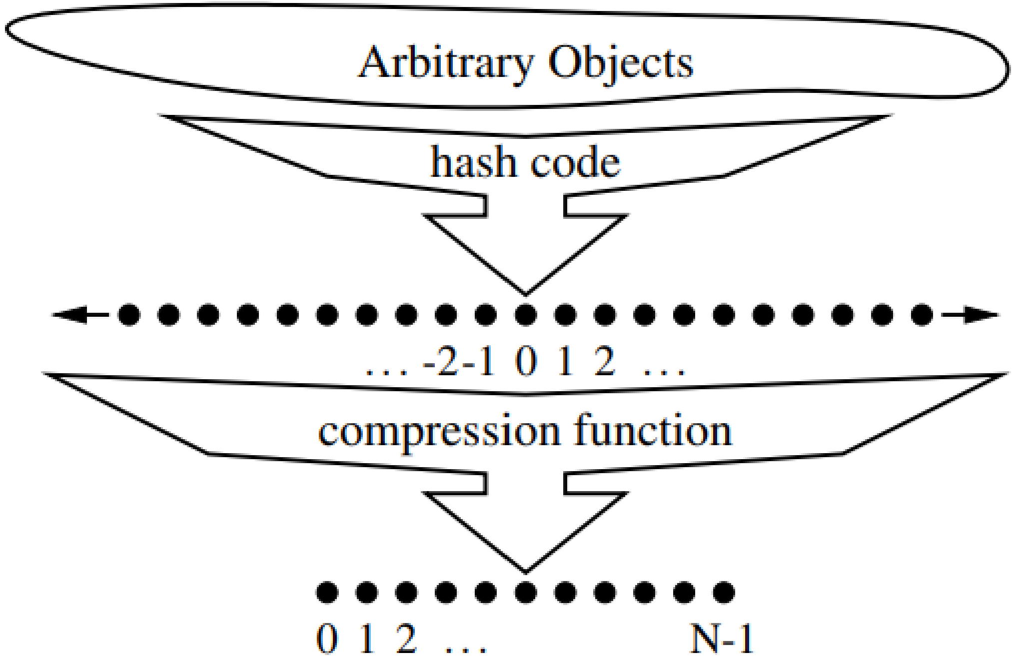
HASH TABLES (哈希表)

- Problems
  - Keys n may not be continuous, therefore an array for the map may have size N >> n
  - A map's key can be other data types, not just integers
- Solution: hash function to map keys to corresponding indices in a table
- Ideally, keys will be distributed in the range from 0 to N-1
- But, there may be two or more distinct keys that get mapped to the same index
- Bucket array



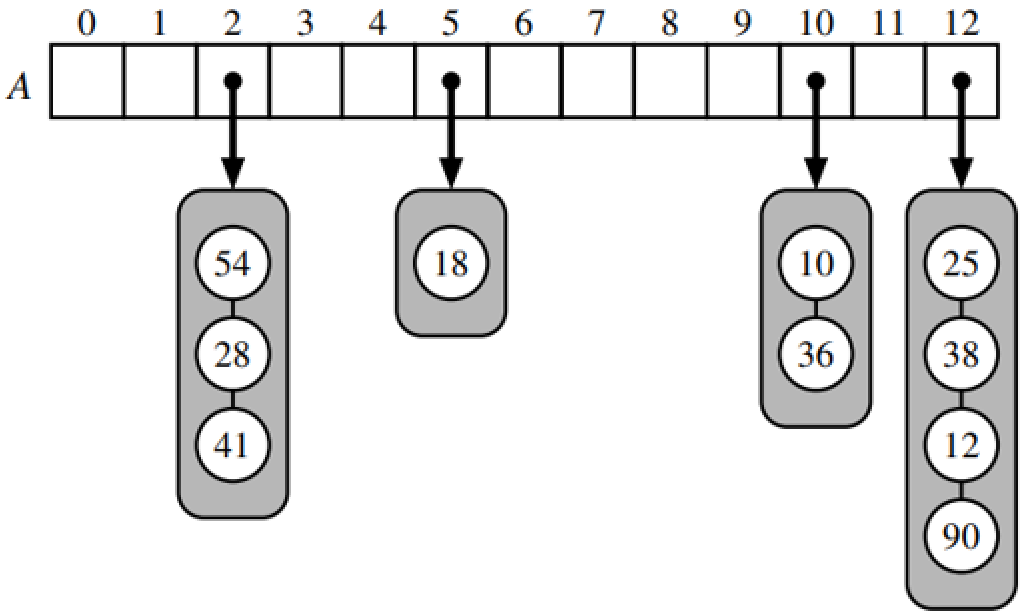
HASH FUNCTION（哈希函数）

- Hash function: h(k)
- **Hashing**: produce a *hash code* that maps a key k to an integer
- **Compressing**: maps the hash code to an integer within a range of indices [0, N-1]
- Why the separation?
  - Independence: hashing is independent of a specific hash table size
  - OO design: hash functions can be overridden



COLLISION HANDLING

- Separate chaining
- Operations on an individual index (bucket) take time proportional to the size of the bucket
- "Good" hash function:
  - expected size of a bucket: n/N
  - n = number of items in the map
  - N = capacity of the bucket array
  - Core map operations run in O(ceiling(n/N))
- Load factor（负载因子）: λ = n/N
- When λ is O(1), operations on the hash table run in O(1)

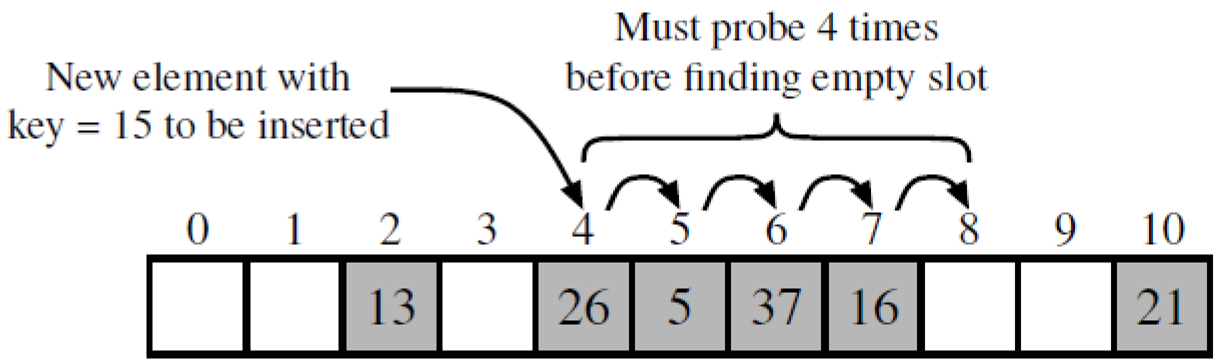


COLLISION HANDLING

- Separate chaining （分离链表）：
  - Advantage: simple implementation
  - Disadvantage: relies on auxiliary data structure - list to hold items with colliding keys
- Alternative approach: open addressing （开放寻址）
  - Always store each item directly in a table slot
  - No auxiliary structures are used
  - Load factor is ways at most 1 and items are stored directly in the cells of the bucket array

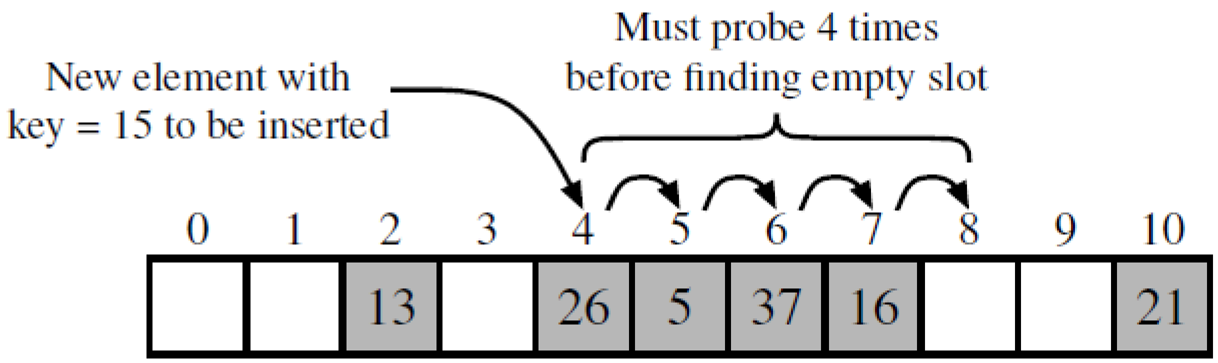
LINEAR PROBING （线性探索）

- Linear probing
- Insert an item (k, v) into a bucket A[j]
- If a[j] is occupied, j = h(k), then try A[(j+1) mod N]
- If A[(j+1) mod N] is occupied, try A[(j+2) mod N], so on
- Need to change the implementation of funcgtions such as __getitem__, __setitem__, __delitem__



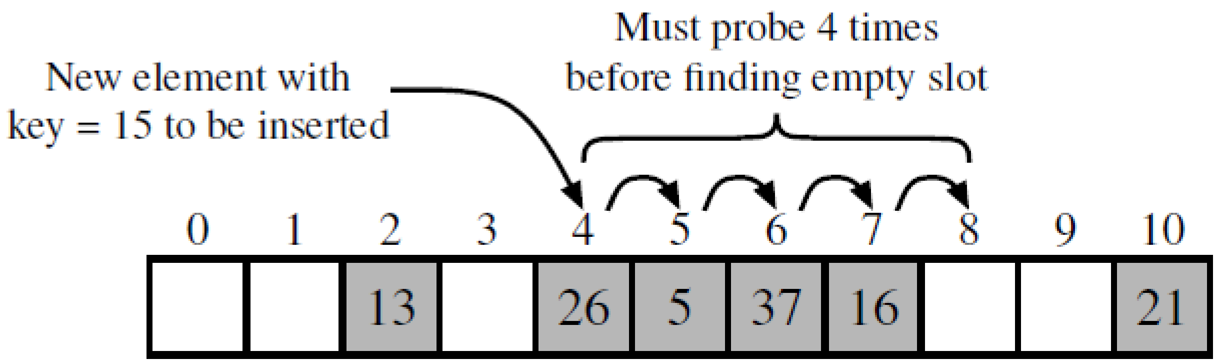
QUADRATIC PROBING （二次探索）

- Iteratively tries A[h(k) + f(i) mod N] for I = 0, 1, 2, …, f(i) = i^2
- Spreads the probing distance over the length N
- Deletion – same strategy as linear probing
- Problems again: secondary clustering （二次聚集）
- When the bucket array is half full, or N is not a prime, quadratic probing does not guarantee to find an empty slot



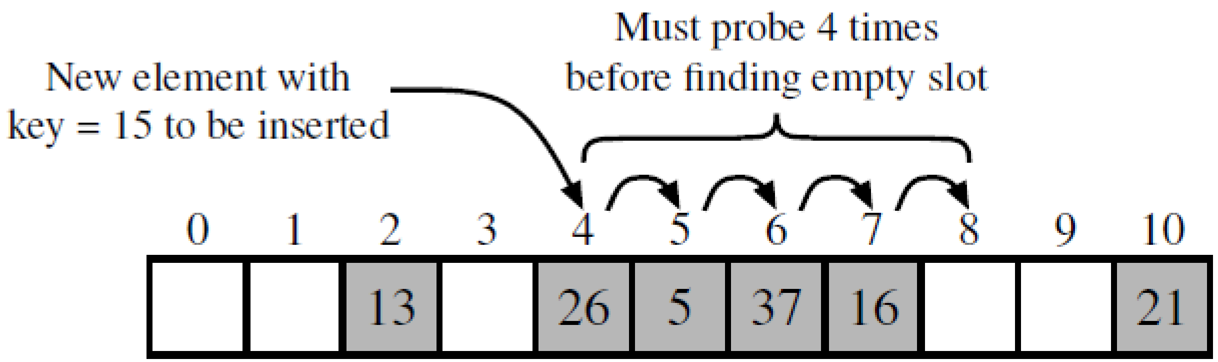
DOUBLE HASHING （二次哈希）

- Secondary hash function h'
- If h maps some key k to a bucket A[h(k)] that is occupied, try A[h(k) + f(i) mod N], for i = 1,2,3, ...
- f(i) = i*h'(k)
- h'(k) cannot be 0
- h'(k) = q – (k mod q), q, N are prime numbers and q < N



ADDITIONAL OPEN ADDRESSING

- A[h(k) + f(i) mod N] where f(i) produces a pseudo-random number
- Repeatable random number



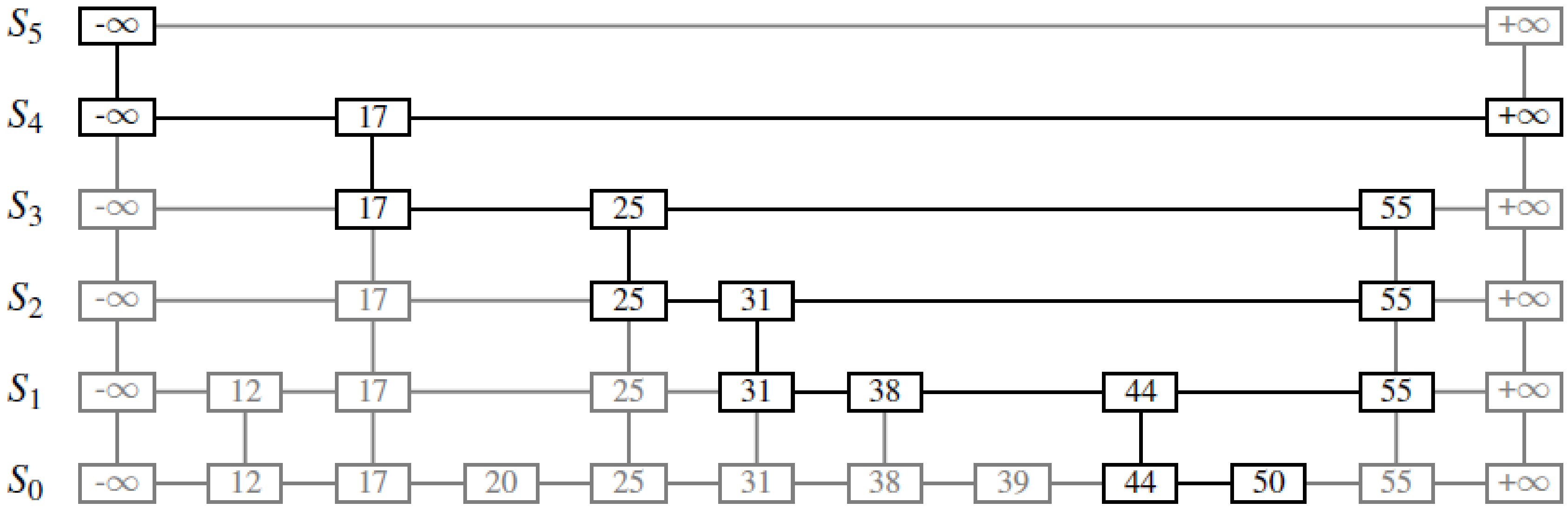
LOAD FACTORS AND REHASING

- Load factor（负载因子）: $\lambda = n/N$ should be kept below 1
- Separate chaining: $\lambda \to 1$ the probability of a collision increases greatly
  - $\lambda < 0.9$ for separate chaining
- Open addressing
  - When λ grows beyond 0.5 and approaches 1, clusters start to show
  - Linear probing: $\lambda < 0.5$
  - Other open addressing: $\lambda < 2/3$
- What happens if an insertion causes the load factor to go beyond 0.5 for linear probing and 2/3 for other open addressing means?
- **Rehashing**: resize the table + reinsert all objects into new table
  - Resize: how?
  - new compression function -> why?

SKIP LISTS

- Search: SkipSearch(k)
- Key k
- Position p to point to the top left position – **start position**

1. If S.below(p) is None, search is done, obtained key at position p <= k. Otherwise p = S.below(p)
2. Move p forward until it is at the rightmost position, such that k(p) <= k
3. Return to step 1



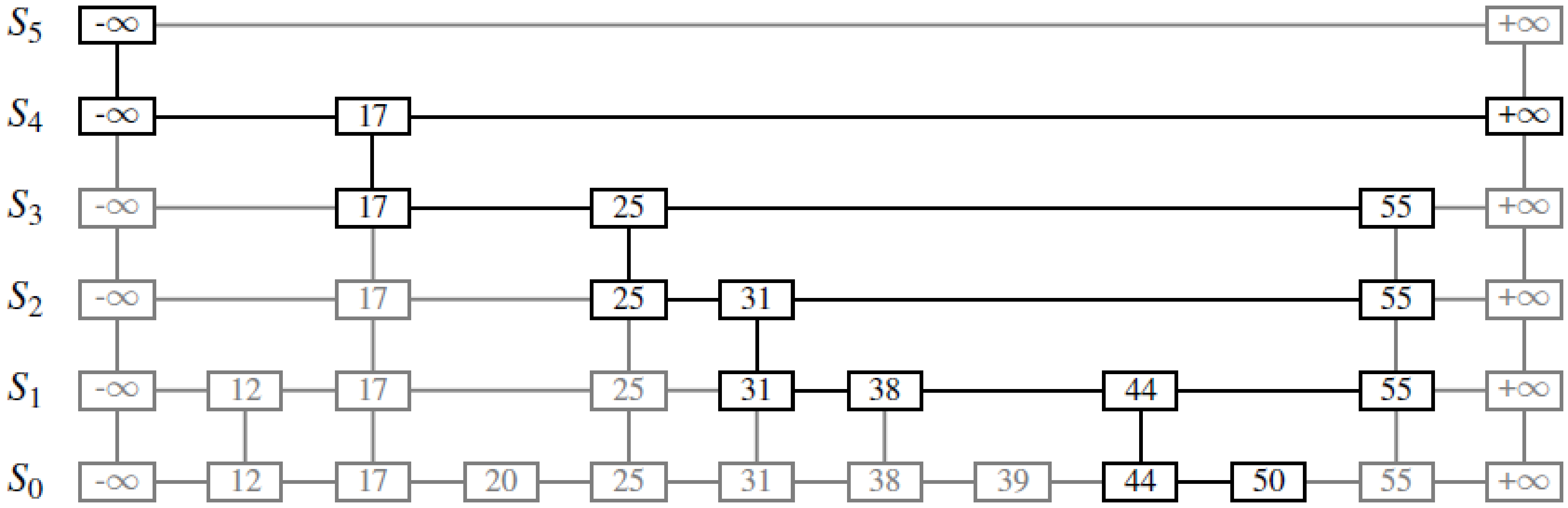
SKIP LISTS

Algorithm SkipSearch(k):

Input: A search key k

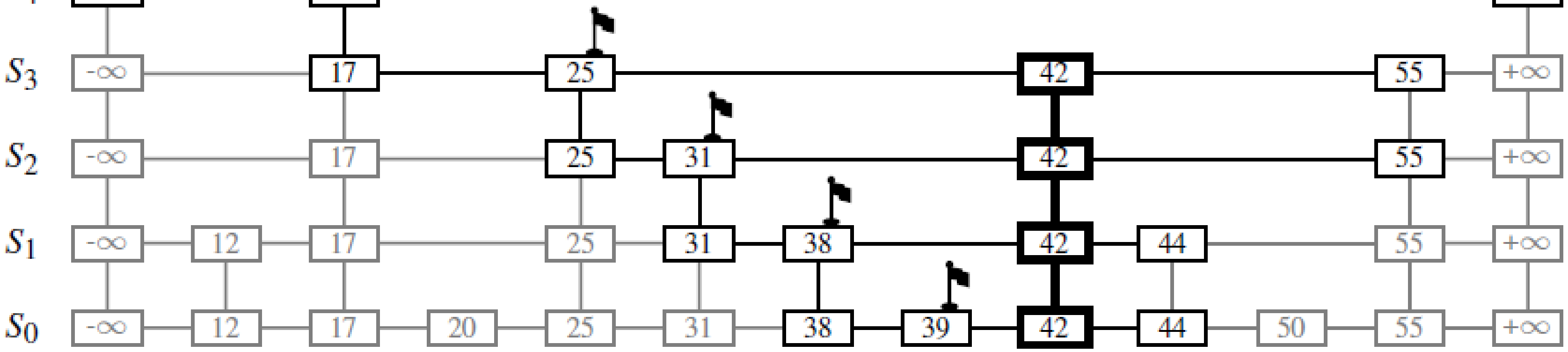
Output: Position p in the bottom list S_0 with the largest key such that $\text{key}(p) \leq k$

```
p = start                              {begin at start position}
while below(p) ≠ None do
    p = below(p)                       {drop down}
    while k ≥ key(next(p)) do
        p = next(p)                    {scan forward}
return p.
```



SKIP LISTS

- Insertion
- Begins with SkipSearch(k) to find a position p
- If key(p) == k, update (k, v)
- Otherwise, need to create a new **tower**
- Insert (k, v) immediately after position p within S_0
- Randomise to decide the height of the tower



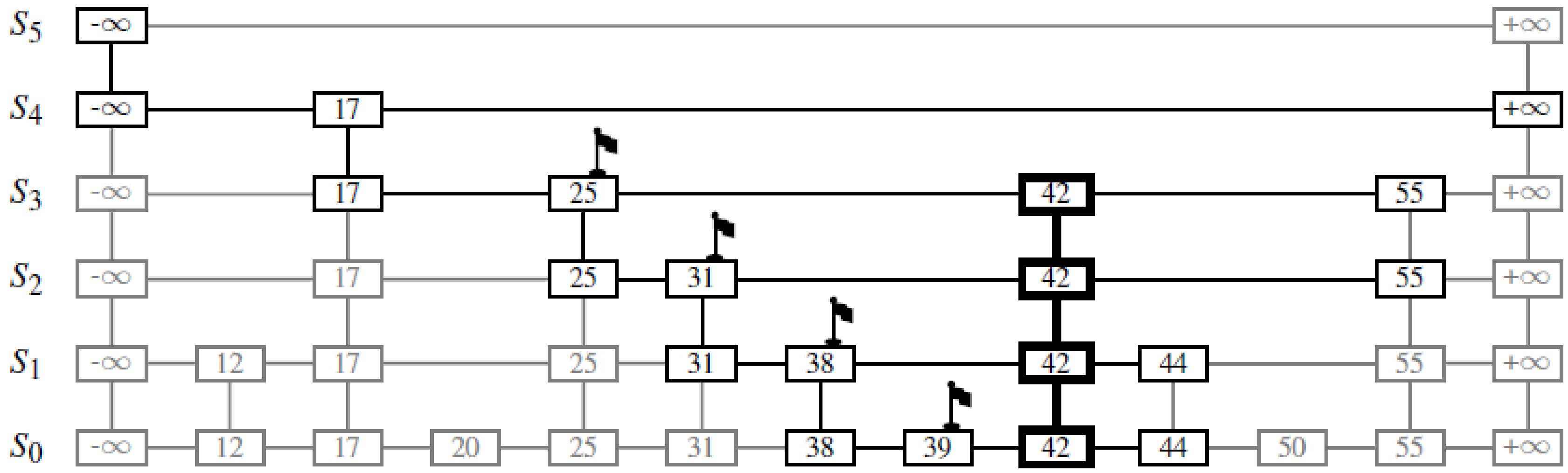
SKIP LISTS

Algorithm SkipInsert(k,v):

Input: Key k and value v

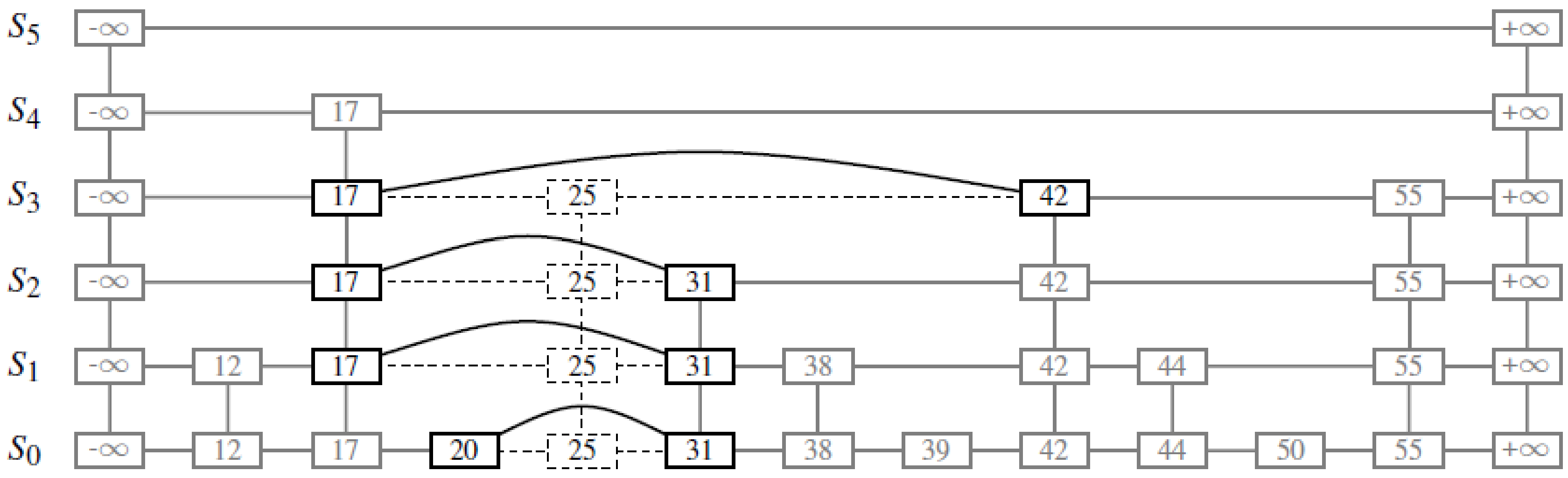
Output: Topmost position of the item inserted in the skip list

```
p = SkipSearch(k)
q = None                          {q will represent top node in n
i = −1
repeat
   i = i + 1
   if i ≥ h then
      h = h + 1                                {add a new leve
      t = next(s)
      s = insertAfterAbove(None, s, (−∞, None))       {grov
      insertAfterAbove(s, t, (+∞, None))             {grow
   while above(p) is None do
      p = prev(p)
   p = above(p)
   q = insertAfterAbove(p, q, (k, v))
until coinFlip() == tails
n = n + 1
return q
```



SKIP LISTS

- Removal
- Perform SkipSearch(k) to obtain p
- If p is illegal, raise error
- I p is legal, remove the entire tower
- Re-establish links between the horizontal neighbours of each removed position

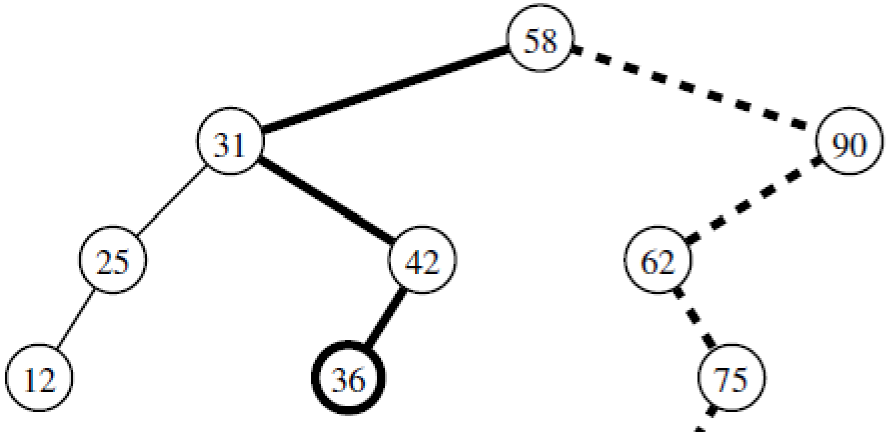


THIS LECTURE

- Search Trees
  - Binary Search Trees
  - Balanced Search Trees

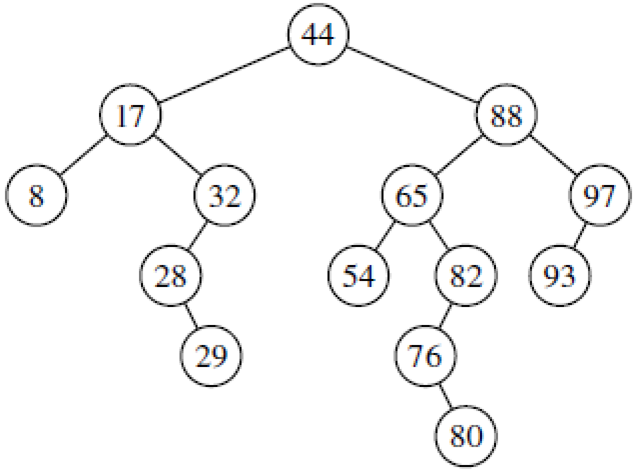
PREVIOUSLY ON BINARY SEARCH TREES

- 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, dentoed 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)
- Running time for finding an occurrence?
  - Proportional to the height of T
  - How you organize the tree really matters
  - Best case?
  - Worst case?



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

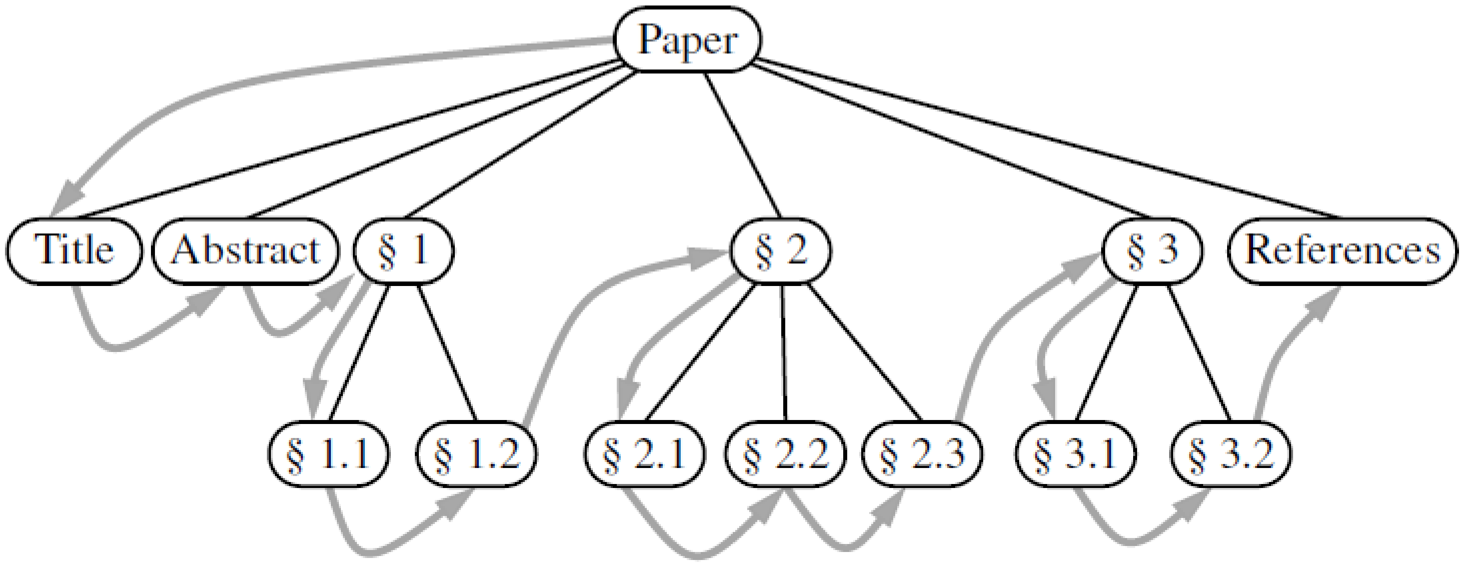


NAVIGATING A BINARY SEARCH TREE

- 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

TREE TRAVERSAL （遍历） ALGORITHMS

- Traversal: a systematic way of accessing or 'visiting' all the positions of T
- Preorder traversal （正序遍历）：
  - visit root of T first and then visit the sub-trees recursively
  - If tree is ordered, then the subtrees are traversed according to the order of the children



Algorithm preorder(T, p):
    perform the "visit" action for position p
    for each child c in T.children(p) **do**
        preorder(T, c)   {recu

TREE ADT

- T.children(p): returns a collection of the children of position p
- T.is_leaf(p): returns true if position p does not have any children (i.e. it is a leaf node)
- len(T): returns the number of elements in T
- T.is_empty(): returns true if T does not contain any elements
- T.positions(): returns a collection of all positions of tree T
- Iter(T): returns a iterator of all elements stored in T

BINARY TREE ADT

- T.left(p): returns the position that represents the left child of p
- T.right(p): returns the position of the right child of p
- T.sibling(p): returns the sibling of p

- For Linked Structure
  - T.add_root(e): create a root for an empty tree, store e as the element, and return the position of the root.
  - T.add_left(p, e): create a node storing element e, link the node as the left child or position p, and return the position
  - T.replace(p, e): replace the element at position p with element e, and return the previous value stored at p
  - T.delete(p): remove the node at position p, replace it with its child, if any, and return the element stored previously at p
  - T. attach(p, T1, T2): attach the internal structure of trees T1 and T2, as left and right subtrees of leaf position p; reset T1 and T2 to empty trees

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

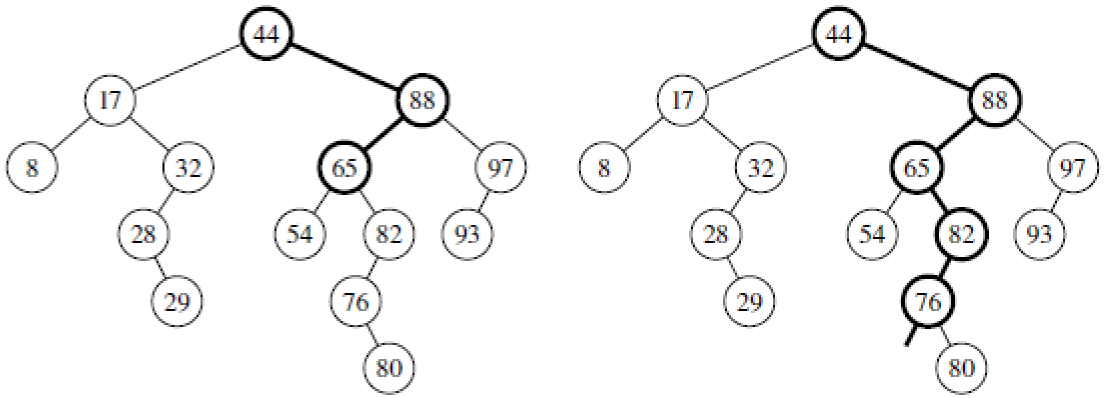
BINARY SEARCH TREE ADT

- 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

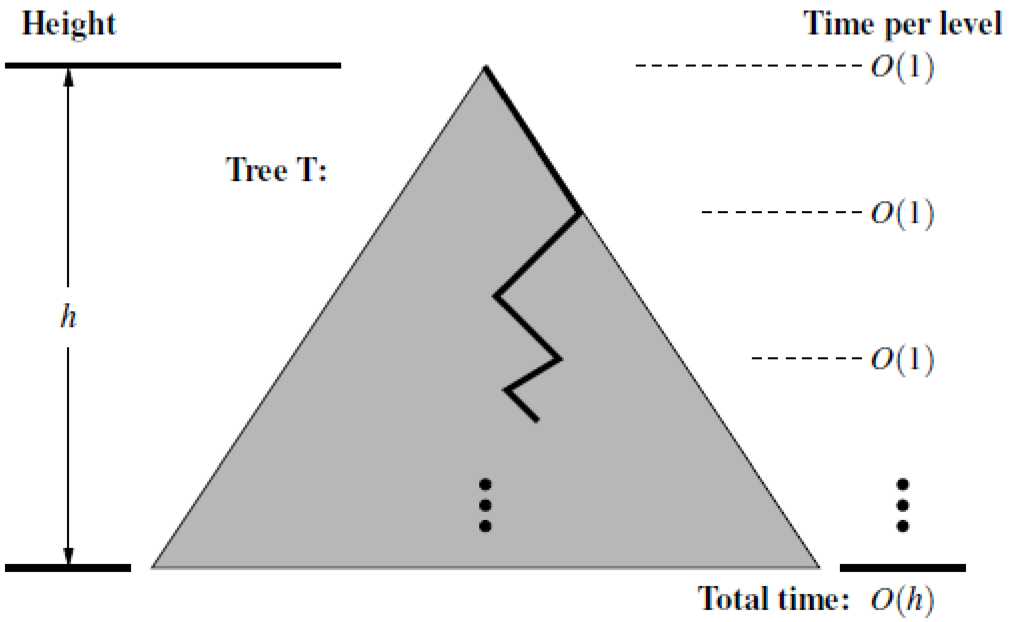
- Locate a particular key by viewing it as a decision tree
- At each position p: is the desired k less than, equal to, or greater than the key stored at position p?




```
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)

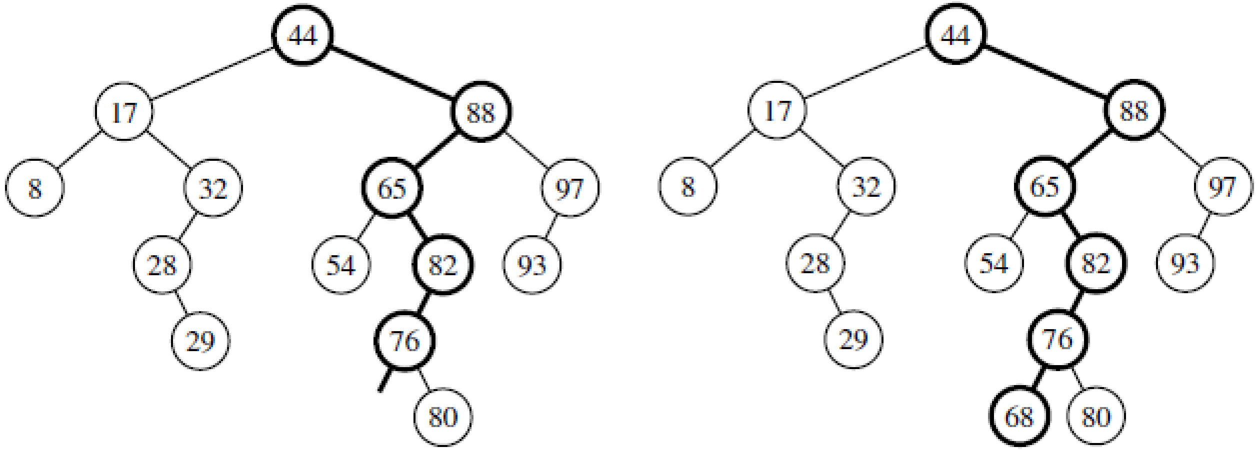



```
Algorithm TreeSearch(T, p, k):
  if k == p.key() then
    return p
  else if k < p.key() and T.left(p) is not None then
    return TreeSearch(T, T.left(p), k)
  else if k > p.key() and T.right(p) is not None then
    return TreeSearch(T, T.right(p), k)
  return p
```

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



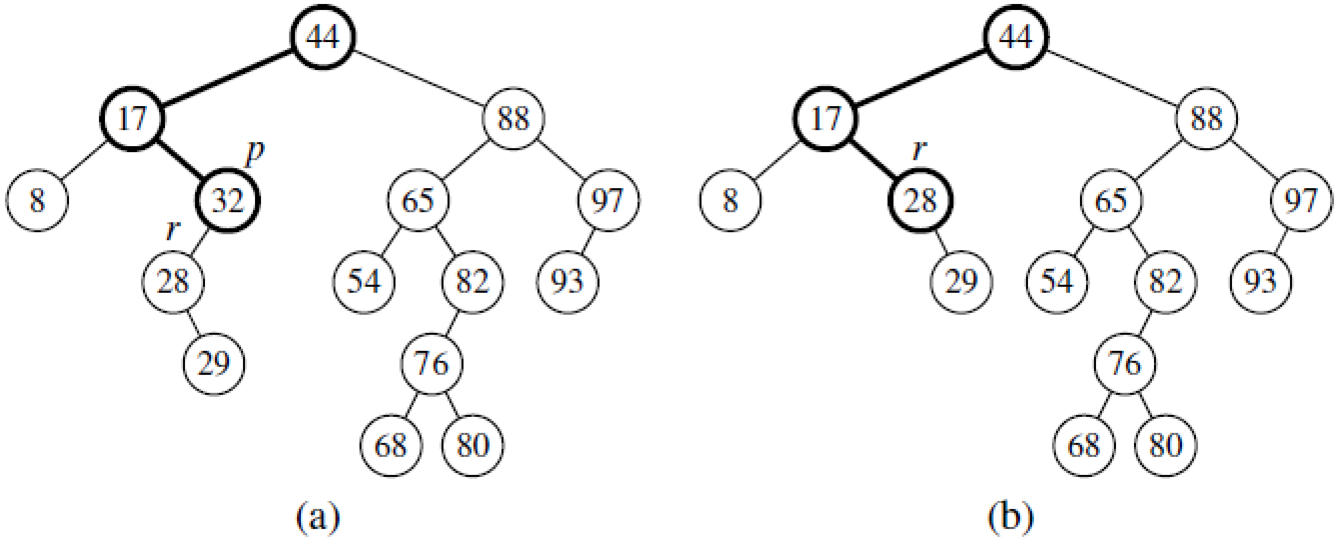
DELETION

- 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 having the greatest key that is less than that of p, 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

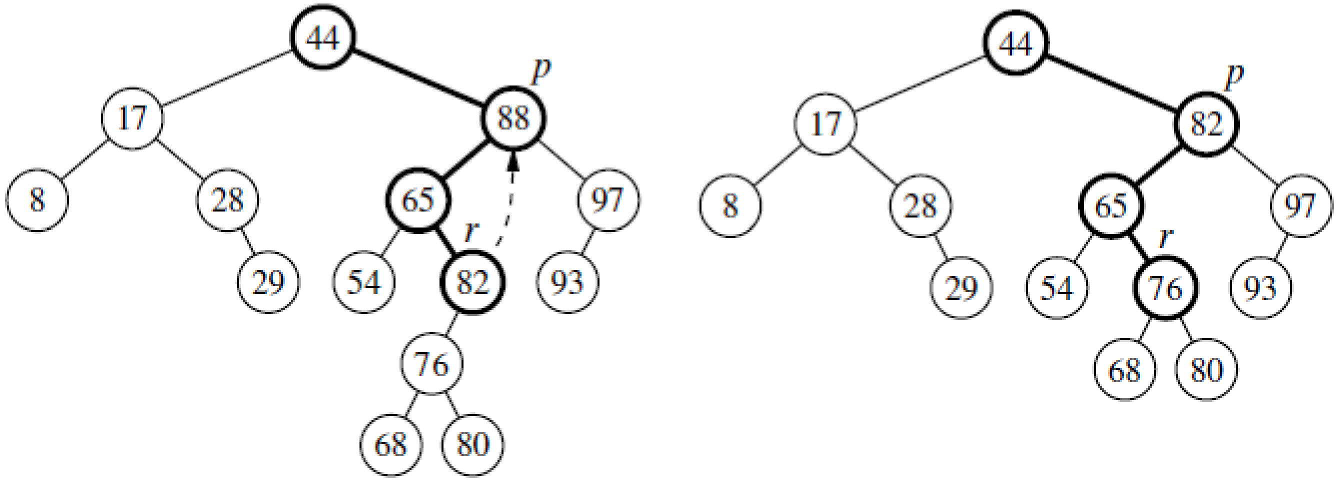
DELETION

- 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 subtre
      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

```
  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

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

  def find_position(self, k):
    """Return position with key k, or else neig
    if self.is_empty():
      return None
    else:
      p = self._subtree_search(self.root(), k)
      self._rebalance_access(p)                 # h
      return p
```

PYTHON IMPLEMENTATION

```
def find_min(self):
  """Return (key,value) pair with minimum key (or None if e
  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 ec

  Return None if there does not exist such a key.
  """
  if self.is_empty():
    return None
  else:
    p = self.find_position(k)              # may not fi
    if p.key() < k:                        # p's key is t
      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
  """
  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 I TION

```
def __getitem__(self, k):
  """Return value associated with key k (raise Key
  if self.is_empty():
    raise KeyError('Key Error: ' + repr(k))
  else:
    p = self._subtree_search(self.root(), k)
    self._rebalance_access(p)              # hook fc
    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
  if self.is_empty():
    leaf = self._add_root(self._Item(k,v))
  else:
    p = self._subtree_search(self.root(), k)
    if p.key() == k:
      p.element()._value = v               # replace
      self._rebalance_access(p)            # hook fc
      return
    else:
      item = self._Item(k,v)
      if p.key() < k:
        leaf = self._add_right(p, item)    # inherite
      else:
        leaf = self._add_left(p, item)     # inherite
  self._rebalance_insert(leaf)             # hook fc
```

```
def __iter__(self):
  """Generate an iteration
  p = self.first()
  while p is not None:
    yield p.key()
    p = self.after(p)

def delete(self, p):
  """Remove the item at given Position."""
  self._validate(p)                                # inhe
  if self.left(p) and self.right(p):               # p ha
    replacement = self._subtree_last_position(s
    self._replace(p, replacement.element())
    p = replacement
  # now p has at most one child
  parent = self.parent(p)
  self._delete(p)                                  # inhe
  self._rebalance_delete(parent)                   # if ro

def __delitem__(self, k):
  """Remove item associated with key k (raise
  if not self.is_empty():
    p = self._subtree_search(self.root(), k)
    if k == p.key():
      self.delete(p)                               # rely
      return                                       # succ
    self._rebalance_access(p)                      # hool
  raise KeyError('Key Error: ' + repr(k))
```

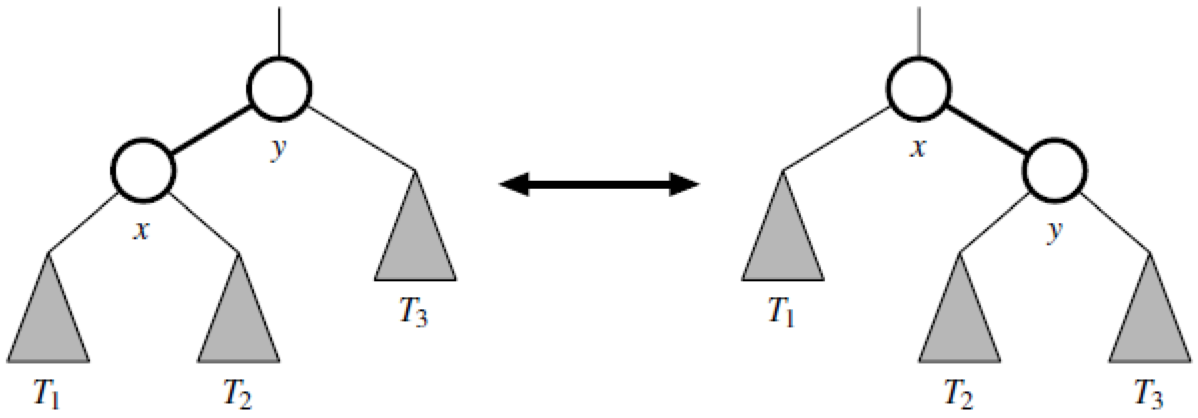
PERFORMANCE OF A BINARY SEARCH TREE

- 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)	$O(n)$

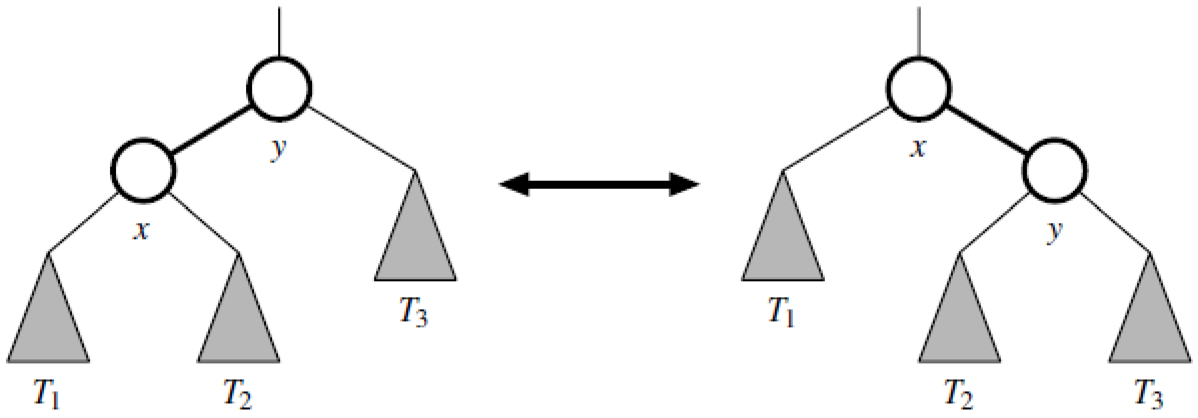
BALANCED SEARCH TREES

- 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



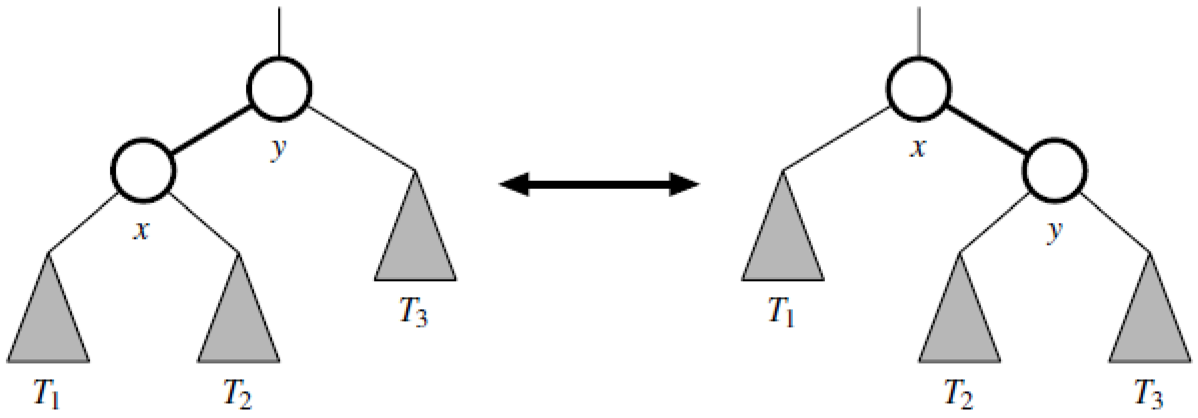
BALANCED SEARCH TREES

- 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

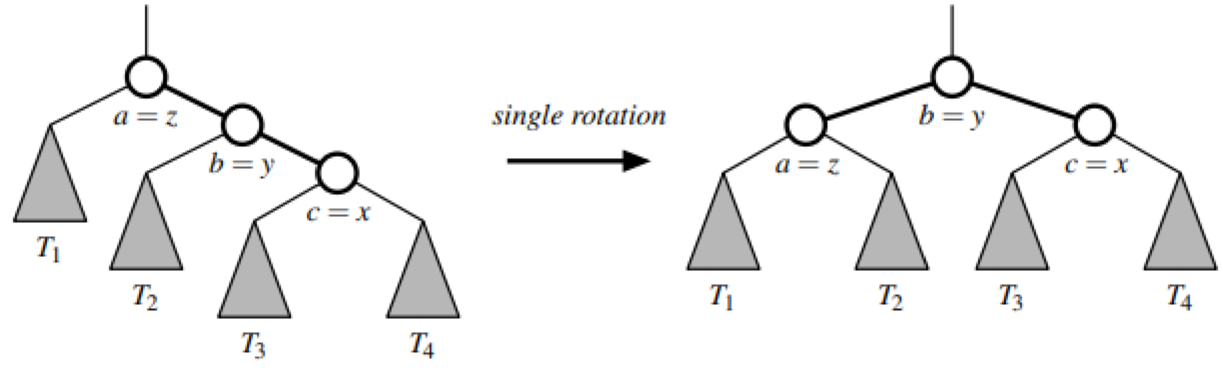
1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

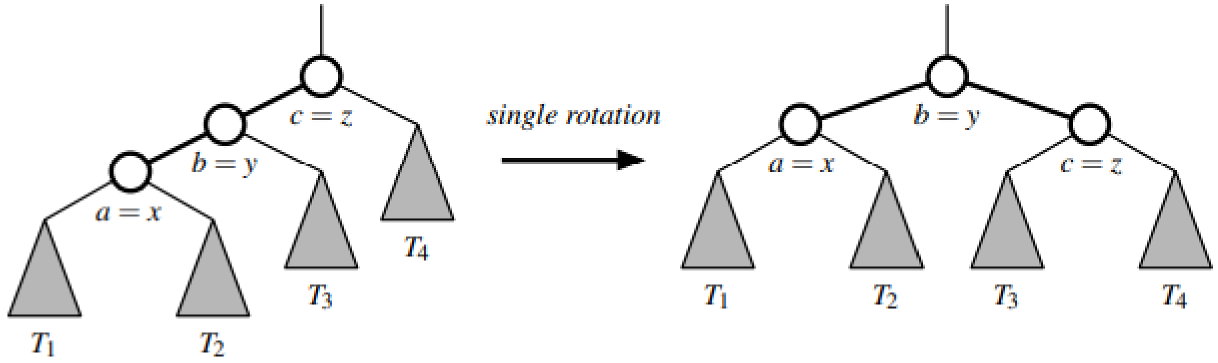
2: Replace the subtree rooted at z with a new subtree rooted at b.

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

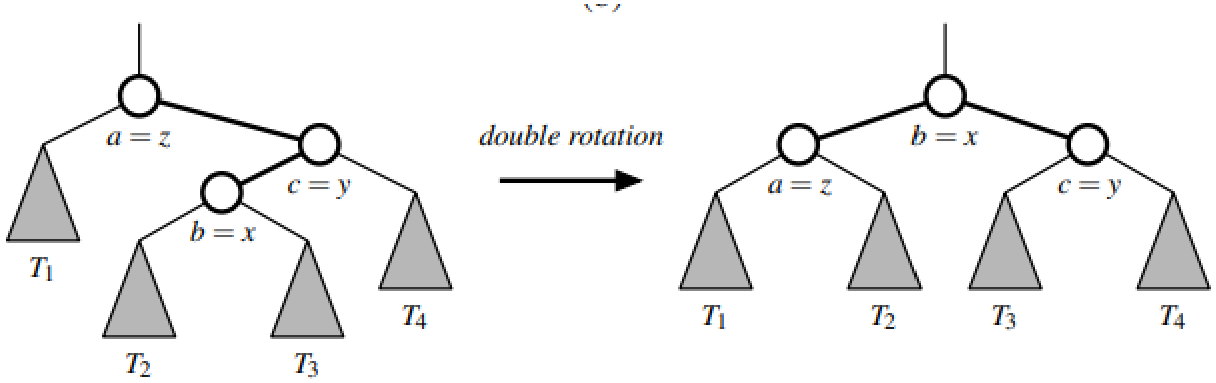
1: Let (a, b, c) be a left-to-right (inorder) listing of the positions x, y, and z, and let (T_1, T_2, T_3, T_4) be a left-to-right (inorder) listing of the four subtrees of x, y, and z not rooted at x, y, or z.

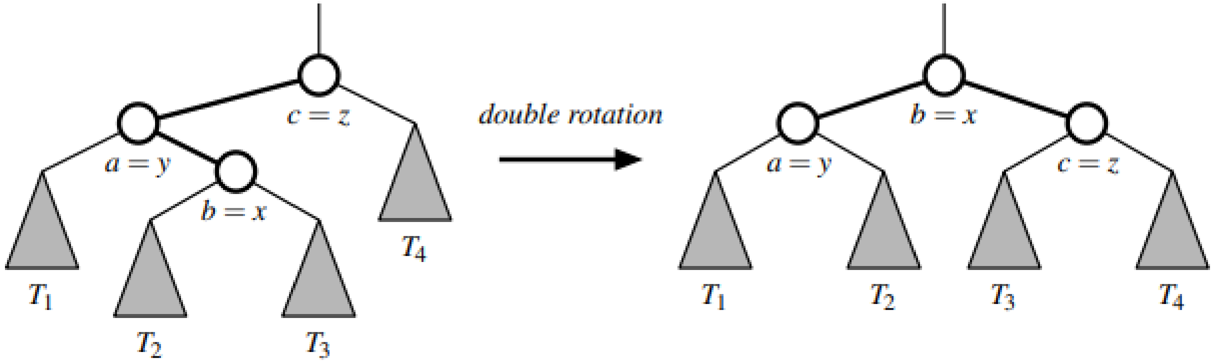
2: Replace the subtree rooted at z with a new subtree rooted at b.

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):

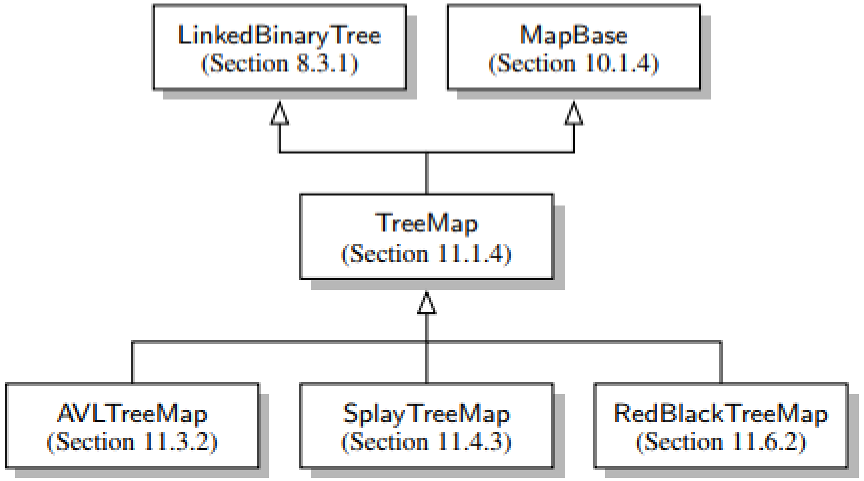
Input: A position x of a binary search tree T that has both a parent y and a grandparent z

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

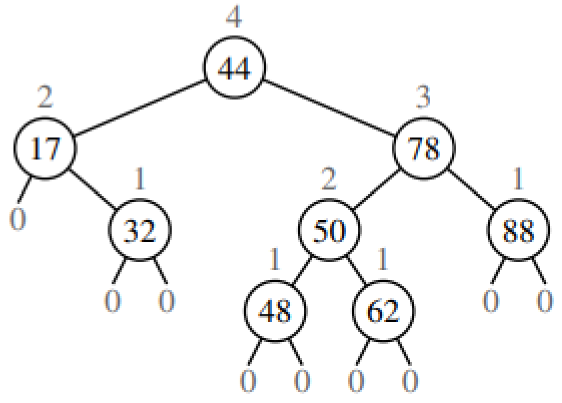
```
def _relink(self, parent, child, make_left_child):
  """Relink parent node with child node (we all
  if make_left_child:                           # ma
    parent._left = child
  else:                                         # ma
    parent._right = child
  if child is not None:                         # ma
    child._parent = parent
```

```
def _rotate(self, p):
  """Rotate Position p above its pa
  x = p._node
  y = x._parent
  z = y._parent
  if z is None:
    self._root = x
    x._parent = None
  else:
    self._relink(z, x, y == z._left)
  # now rotate x and y, including tı
  if x == y._left:
    self._relink(y, x._right, True)
    self._relink(x, y, False)
  else:
    self._relink(y, x._left, False)
    self._relink(x, y, True)
```

```
def _restructure(self, x):
  """Perform trinode restructure of Position x with
  y = self.parent(x)
  z = self.parent(y)
  if (x == self.right(y)) == (y == self.right(z)):
    self._rotate(y)
    return y
  else:
    self._rotate(x)
    self._rotate(x)
    return x
```

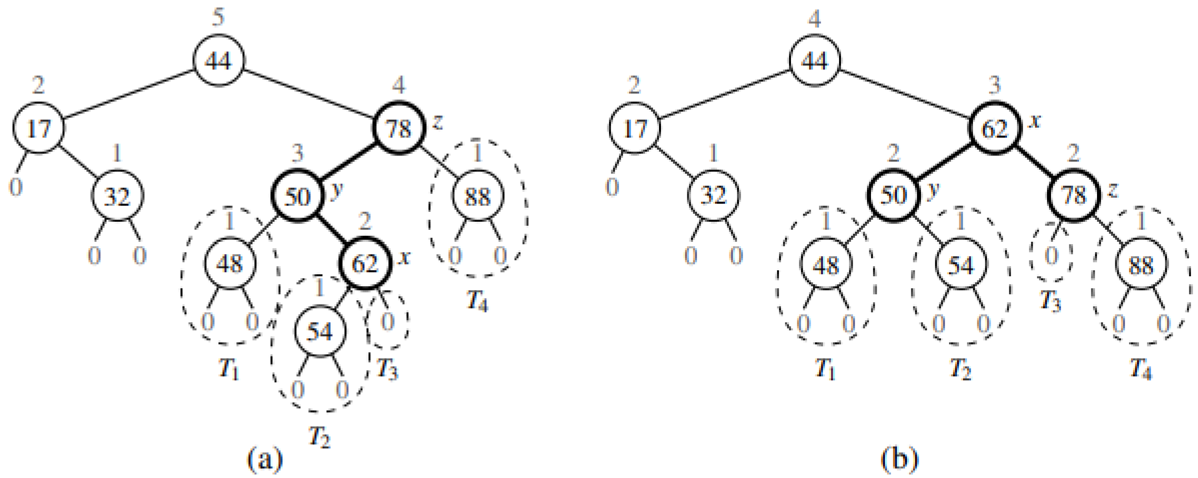
AVL TREES

- 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)

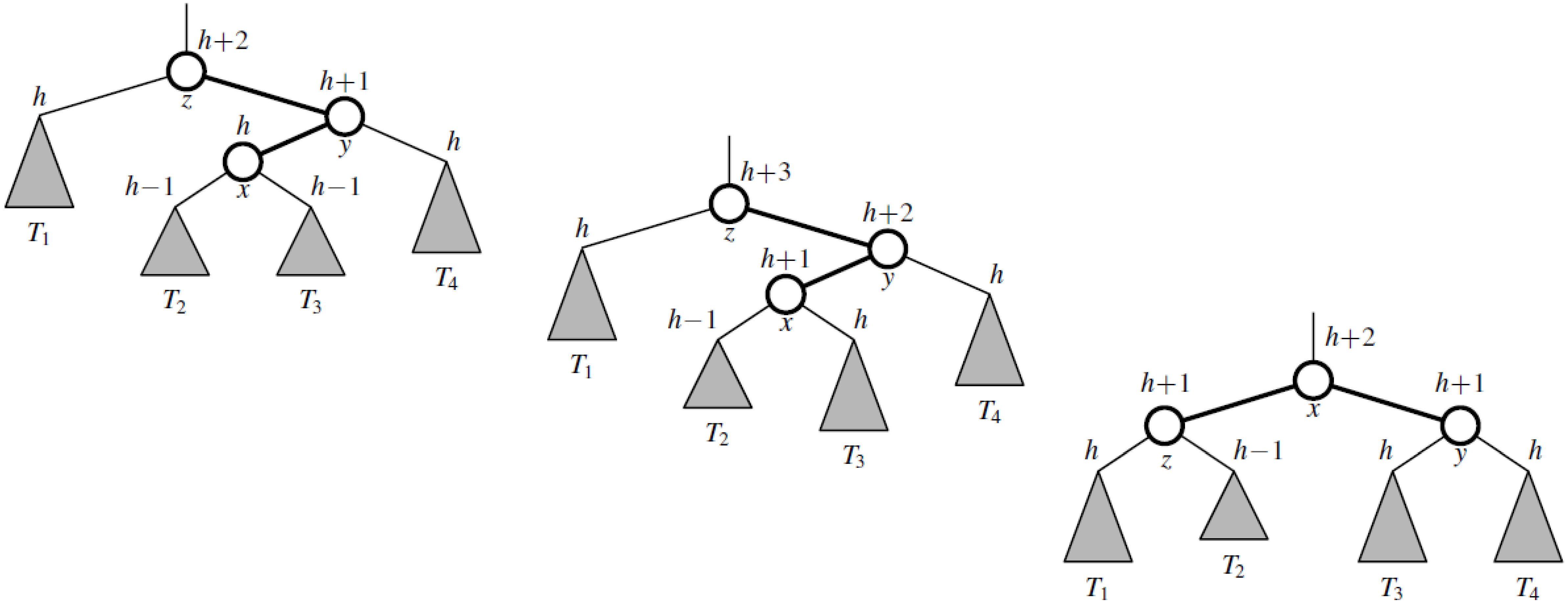


AVL TREES

- 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)

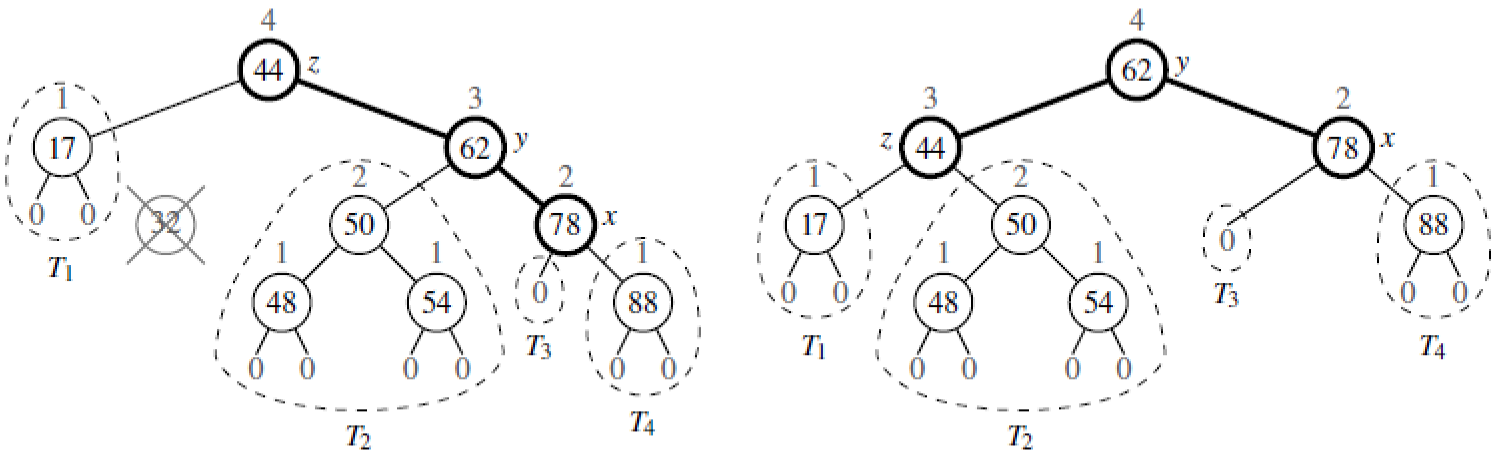


AVL TREES


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



PERFORMANCE OF AVL TREES

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)	$O(n)$

THANKS

See you in the next session!