Search Trees

- Explain Binary Search Trees
- Implement insert/delete operations
- Analyze the performance of binary search trees
- Explain balanced search trees

Maps and Hashtables - Review

- A map is a a collection of key/value entries – keys are unique
- Map ADT: get(k), put(k, v), remove(k), size(), isEmpty(), entrySet(), keySet(), values()
- We can use a doubly linked list to implement an unsorted map:
  - put takes $O(1)$ time
  - get and remove take $O(n)$ time
- Can we do better?
  - Use hash tables

Maps and Hashtables - Review

- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$
- A hash function composed of two functions:

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

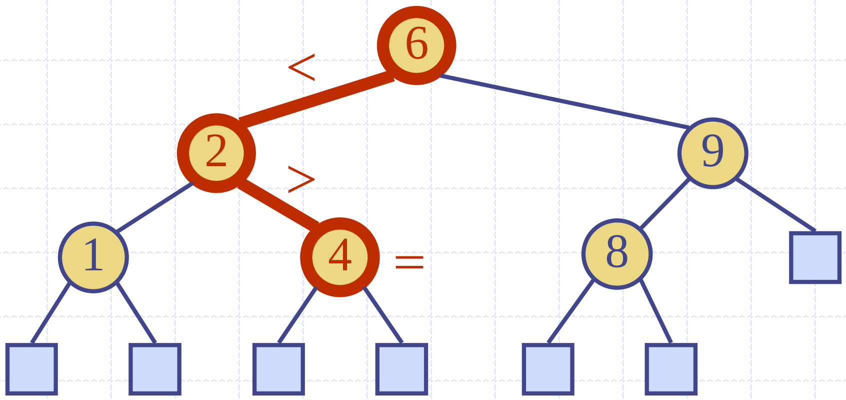
  h_2: integers $\rightarrow [0, N-1]$

Maps and Hashtables - Review

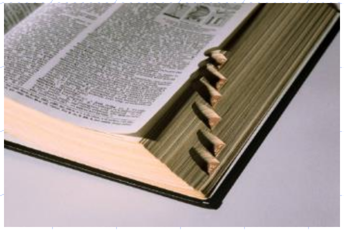
- Hash code implementation:
  - Memory address of the key
  - Integer cast - reinterpret the bits of the key as an integer
  - Component sum - partition the bits of the key into components of fixed length (e.g., 16 or 32 bits) and we sum the components
  - Polynomial accumulation
- Compression functions:
  - Division: h2 (y) = y mod N
  - Multiply, Add and Divide (MAD): h2 (y) = (ay + b) mod N
- Collision Handling
  - Chaining
  - Linear and quadratic probing

Presentation for use with the textbook Data Structures and Algorithms in Java, 6th edition, by M. T. Goodrich, R. Tamassia, and M. H. Goldwasser, Wiley, 2014

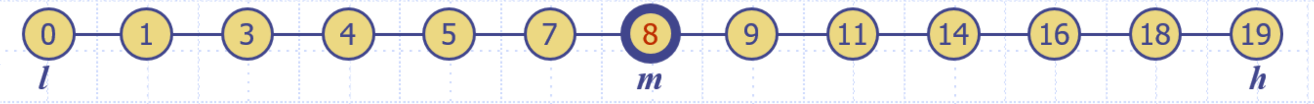
Binary Search Trees



Ordered Maps

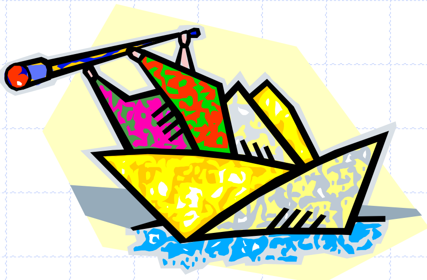


- In a sorted map, entries are stored in order by their keys

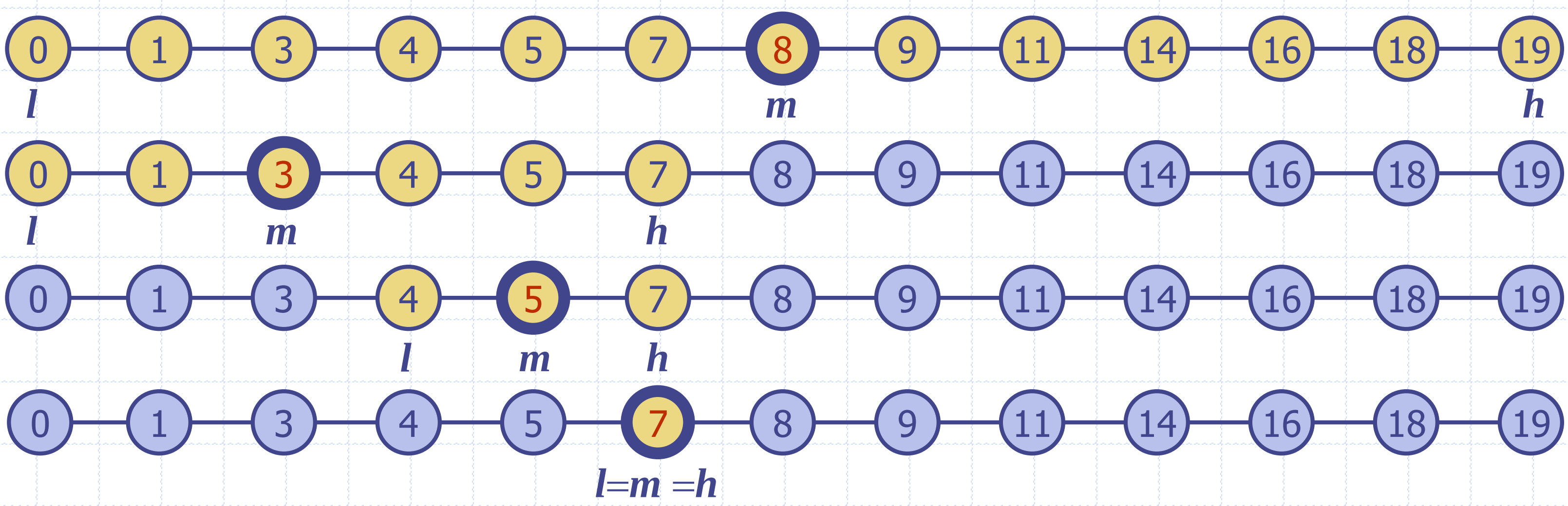


- This allows us to support nearest neighbor queries:
  - Item with largest key less than or equal to k
  - Item with smallest key greater than or equal to k

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)

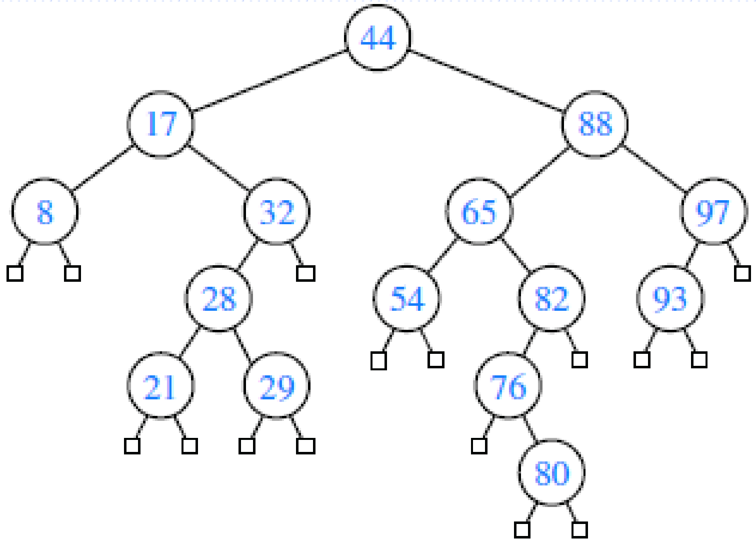


Binary Search Trees

- We can use a **search-tree structure** to efficiently implement a ***sorted map***
- Binary trees are an excellent data structure for storing entries of a map, assuming we have an order relation defined on the keys.
- *A **binary search tree*** is a *proper binary tree:* each internal position *p* stores 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*.

Binary Search Trees

- In the example below, the leaves of the tree serve only as "placeholders."



- They can be represented as null references in practice, thereby reducing the number of nodes in half

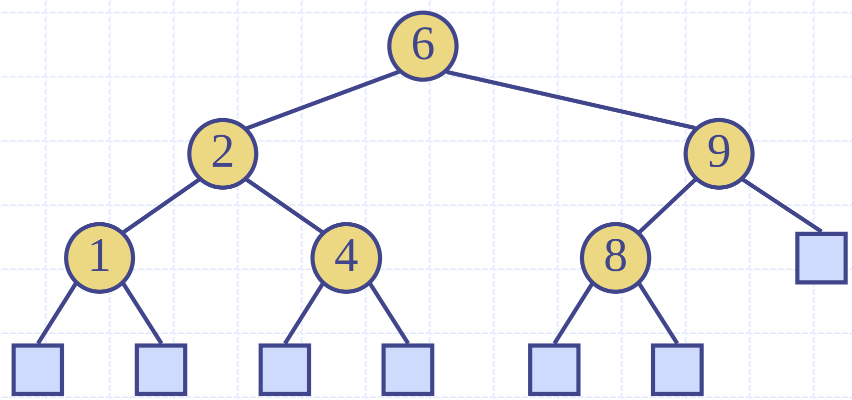
Search Tables

- **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)

Binary Search Trees

- A binary search tree is a binary tree storing keys (or key-value entries) at its internal 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

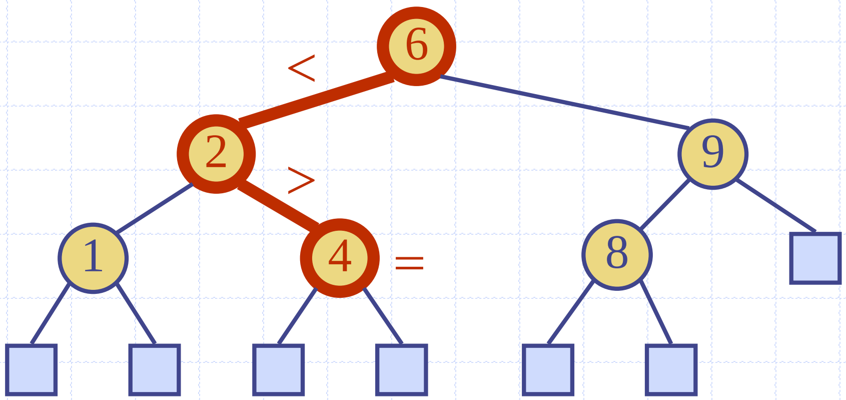
- An **inorder** traversal of a binary search trees **visits the keys in increasing order**



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: get(4):
  - Call TreeSearch(4,root)
- The algorithms for nearest neighbor queries are similar

Algorithm *TreeSearch*(k, v)
    if *T.isExternal* (v)
        return v
    if $k < key(v)$
        return *TreeSearch*(k, *left*(v))
    else if $k = key(v)$
        return v
    else { $k > key(v)$ }
        return *TreeSearch*(k, *right*(v))



Insertion

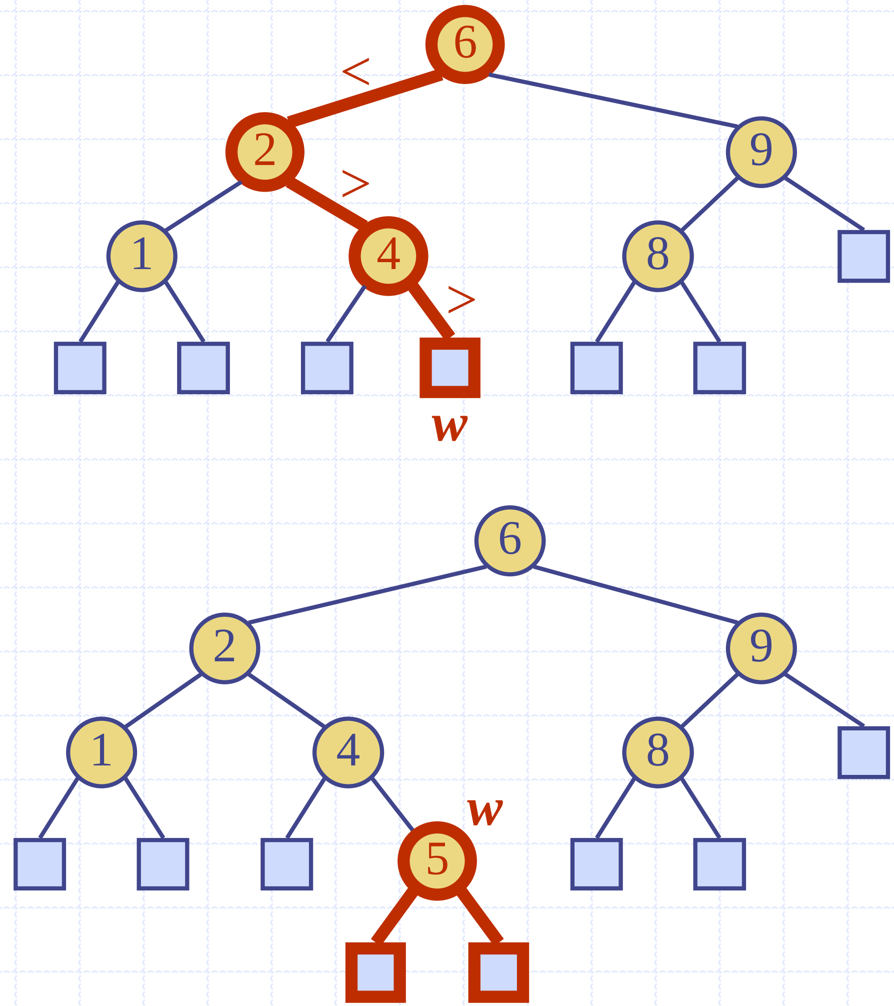
- The map operation put(k, v) begins with a search for an entry with key k.
  - If found, that entry's existing value is reassigned.
  - Otherwise, the **new entry can be inserted into the underlying tree** by expanding the leaf that was reached at the end of the failed search into an internal node.

Algorithm TreeInsert(k, v):
  Input: A search key k to be associated with value v
  p = TreeSearch(root(), k)
  if k == key(p) **then**
    Change p's value to (v)
  else
    expandExternal(p, (k, v))

- expandExternal(p, e): Stores entry e at the external position p, and expands p to be internal, having two new leaves as children.

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 leaf reached by the search
- We insert **k** at node **w** and expand **w** into an internal node
- Example: insert 5

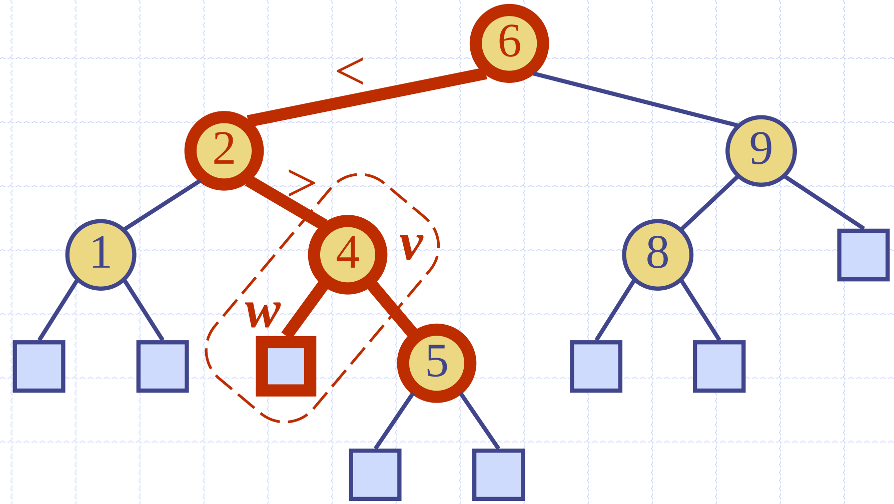


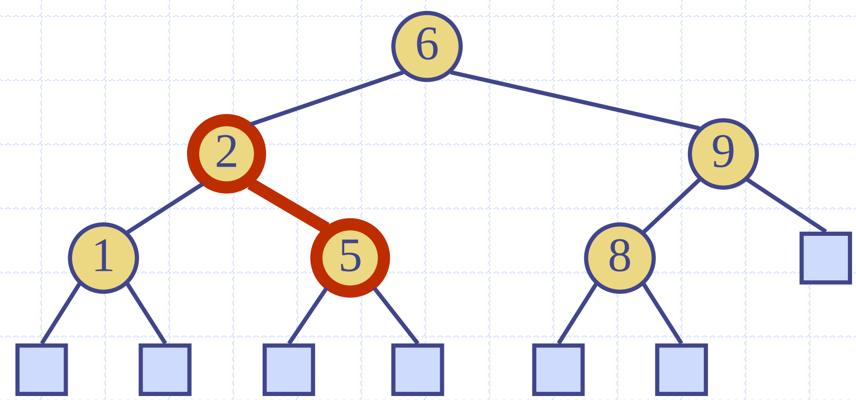
Deletion

- Deleting an entry from a binary search tree is a bit more complex than inserting a new entry
  - the position of an entry to be deleted might be anywhere in the tree (as opposed to insertions, which always occur at a leaf).
- To delete an entry with key *k*, we begin by calling TreeSearch(root( ), *k*) to find the position *p* storing an entry with key equal to *k* (if any).

Deletion

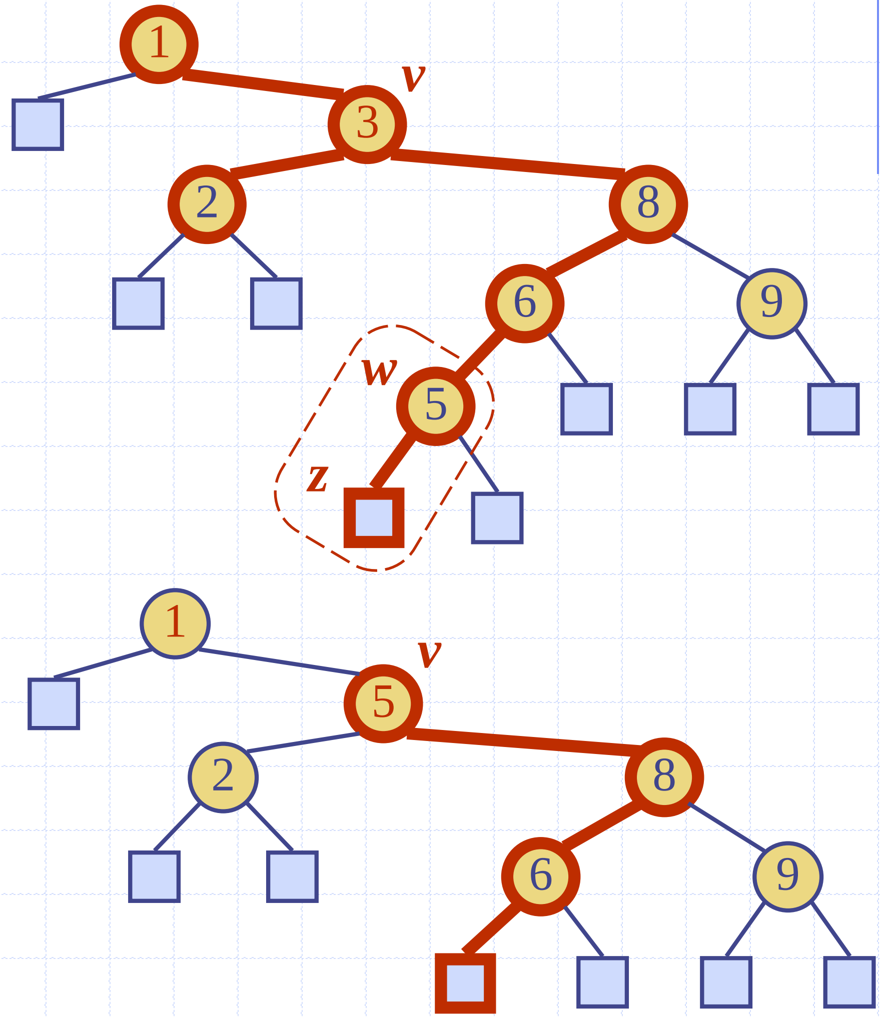
- To perform operation remove(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If node v has a leaf child w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: remove 4





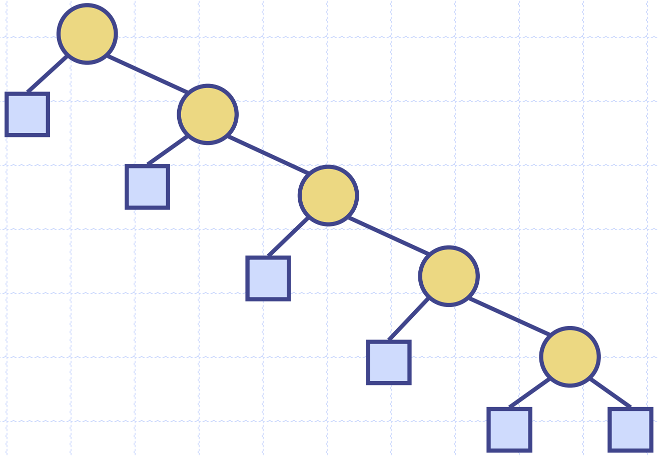
Deletion (cont.)

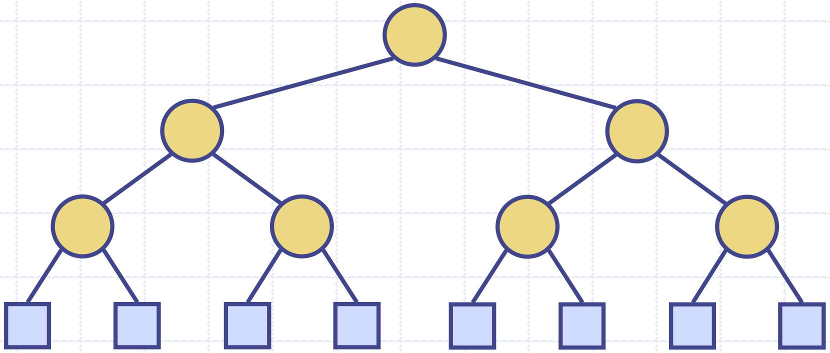
- We consider the case where the key k to be removed is stored at a node v whose children are both internal
  - we find the internal node w that follows v in an inorder traversal
  - we copy $key(w)$ into node v
  - we remove node w and its left child z (which must be a leaf) by means of operation removeExternal(z)
- 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 $O(n)$
  - methods get, put and remove take $O(h)$ time
- The height h is $O(n)$ in the worst case and $O(\log n)$ in the best case





Performance

- on average, a binary search tree with n keys generated from a random series of insertions and removals of keys has expected height $O(\log n)$

Method	Running Time
size, isEmpty	$O(1)$
get, put, remove	$O(h)$
firstEntry, lastEntry	$O(h)$
ceilingEntry, floorEntry, lowerEntry, higherEntry	$O(h)$
subMap	$O(s+h)$
entrySet, keySet, values	$O(n)$

Java Implementation

- We define a **TreeMap** class that implements the sorted map ADT while using a binary search tree for storage.
- The TreeMap class is declared as a child of the **AbstractSortedMap** base class, thereby inheriting support for performing comparisons based upon a given (or default) **Comparator**, a nested **MapEntry** class for storing key-value pairs, and concrete implementations of methods **keySet** and values based upon the **entrySet** method, which we will provide.

Java Implementation

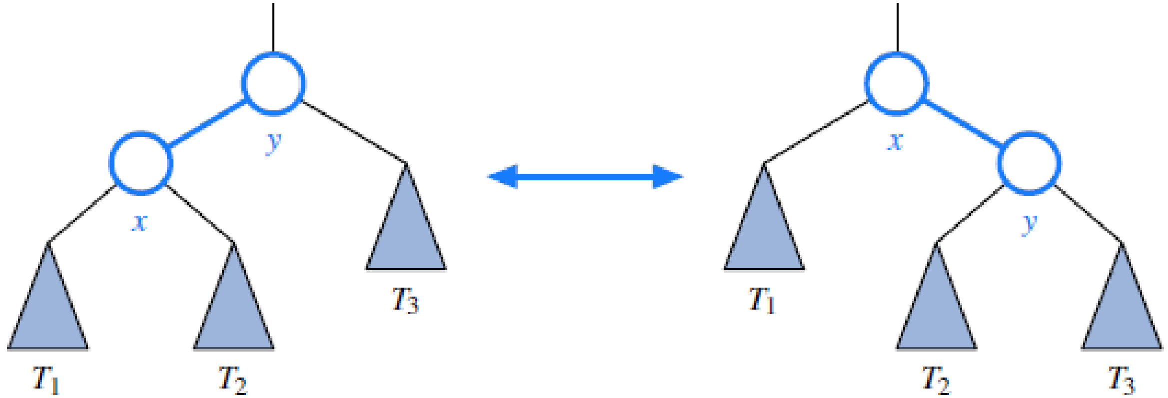
- For representing the tree structure, our TreeMap class maintains an instance of a subclass of the **LinkedBinaryTree** class from Section 8.3.1.
- In this implementation, we choose to represent the search tree as a *proper* binary tree, with explicit leaf nodes in the binary tree as sentinels, and map entries stored only at internal nodes.

Balanced Search Trees

- if we could assume a random series of insertions and removals, the standard binary search tree supports $O(\log n)$ expected running times for the basic map operations.
- However, we may only claim $O(n)$ worst-case time, because **some sequences of operations may lead to an unbalanced tree with height proportional to *n*.**
- A **balanced binary search tree** is a tree that automatically **keeps its height small** (guaranteed to be logarithmic) for a sequence of insertions and deletions.

Balanced Search Trees

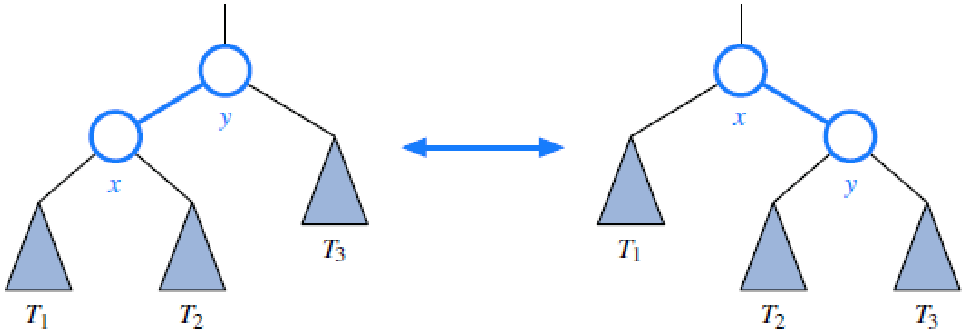
- The primary operation to rebalance a binary search tree is known as a ***rotation***.
- During a rotation, we “rotate” a child to be above its parent.



- In the context of a tree-balancing algorithm, a rotation allows the shape of a tree to be modified while maintaining the search-tree property.

Balanced Search Trees

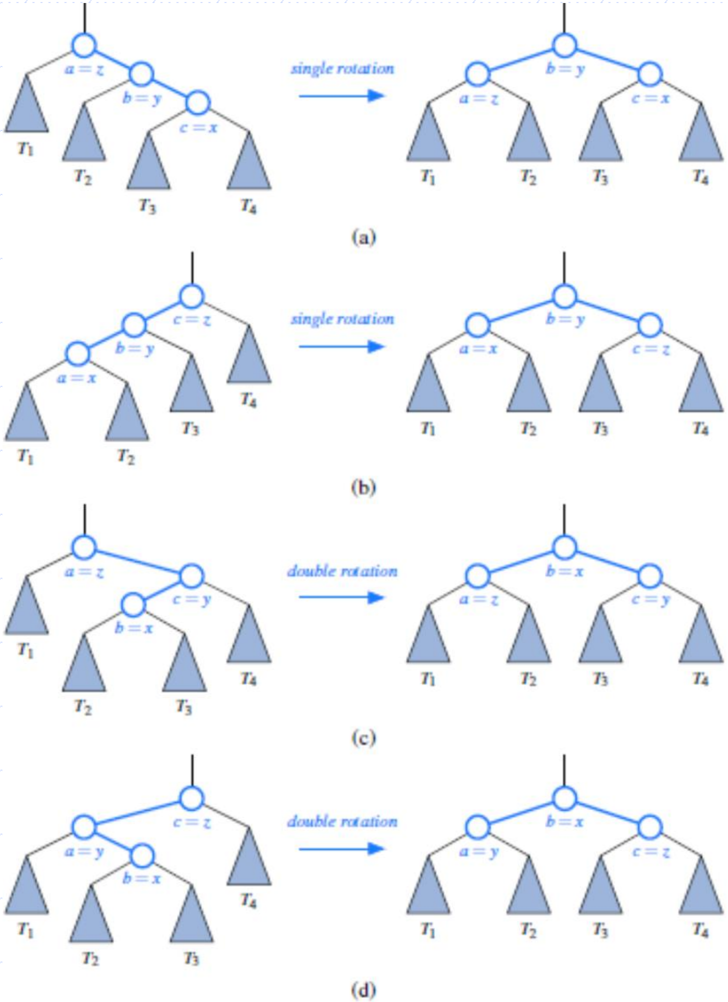
- A rotation can be performed to transform the left formation into the right, or the right formation into the left.
- Note that all keys in subtree *T1* have keys less than that of position *x*, all keys in subtree *T2* have keys that are between those of positions *x* and *y*, and all keys in subtree *T3* have keys that are greater than that of position *y*.



- In the first configuration, *T2* is the right subtree of position *x*; in the second configuration, it is the left subtree of position *y*.

Balanced Search Trees

- One or more rotations can be combined to provide broader rebalancing within a tree.
- One such compound operation we consider is a ***trinode restructuring***.



Java Framework for Balancing Search Trees

- The **TreeMap** class is designed in a way that allows it to be easily extended to provide more advanced tree-balancing strategies.
- Hooks for Rebalancing Operations
  - **rebalanceInsert(p)** is made from within the put method, after a new node is added to the tree at position p
  - **rebalanceDelete(p)** is made from within the remove method, after a node is deleted from the tree
  - **rebalanceAccess(p)** is made by any call to get, put, or remove that does not result in a structural change
- TreeMap class relies on storing the tree as an instance of a new nested class, **BalanceableBinaryTree**.
- That class is a specialization of the original **LinkedBinaryTree** class.