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:

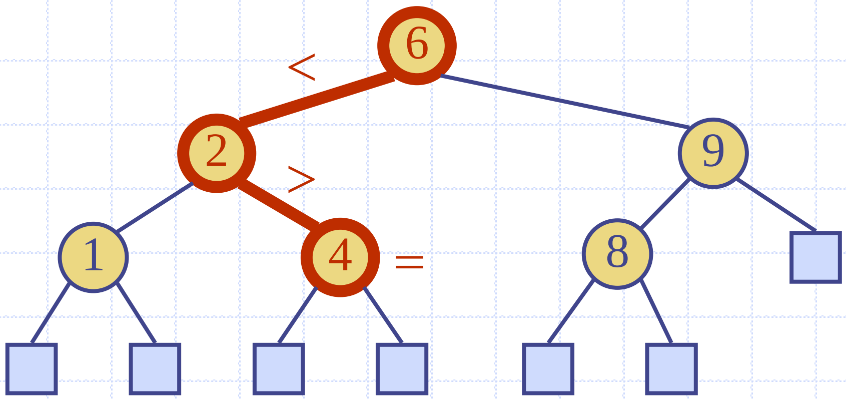
  h_2: integers $\rightarrow [0, N-1]$

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
  - Separate **Chaining**
  - Linear and quadratic **probing**
- The **expected running time** of operations in a hash table is $O(1)$

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

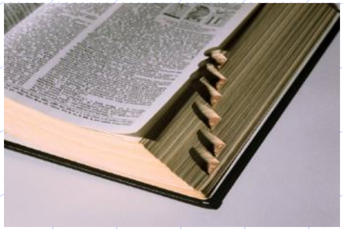
Binary Search Trees



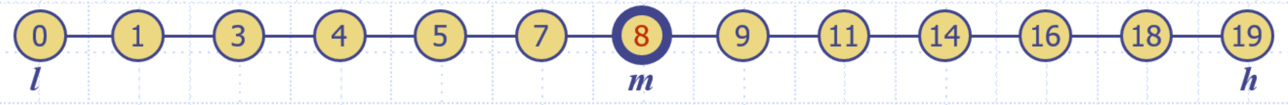
Binary Search Trees

- In this lecture we use a **search-tree structure** to efficiently implement a ***sorted map***.
- Recall that three most fundamental methods of a map are:
  - **get(*k*)**: Returns the value *v* associated with key *k*, if such an entry exists; otherwise returns null.
  - **put(*k*, *v*)**: Associates value *v* with key *k*, replacing and returning any existing value if the map already contains an entry with key equal to *k*.
  - **remove(*k*)**: Removes the entry with key equal to *k*, if one exists, and returns its value; otherwise returns null.

Ordered Maps

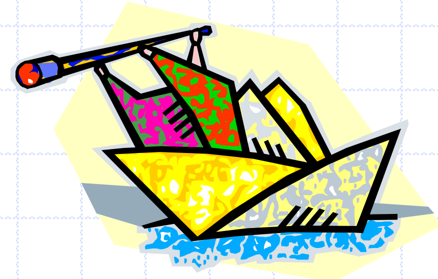


- In a **sorted map**, entries are stored in order by their keys:

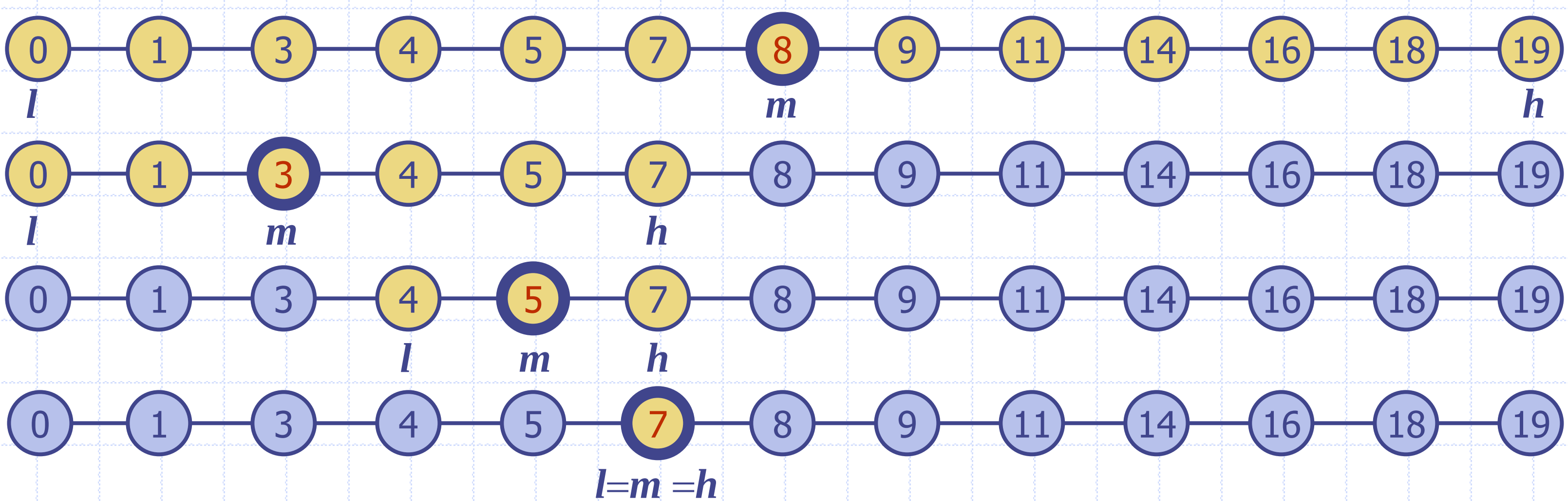


- This allows us to **support nearest neighbor queries** (finding the item in a data structure that is closest to a given item, or key closest to lookup key):
  - Item with **largest key less** than or equal to k
  - Item with **smallest key greater** than or equal to k

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 (**l**-low, **m**-middle, **h**-high)
  - at each step, the number of candidate items is halved
  - terminates after **O(log n)** steps
- Example: find(**7**)



Recall Binary Trees

- A **full binary tree** (sometimes **proper binary tree**) is a **tree** in which every node other than the leaves has two children.

```
          O
        /   \
      O       O
     / \     / \
    O   O   O   O
```

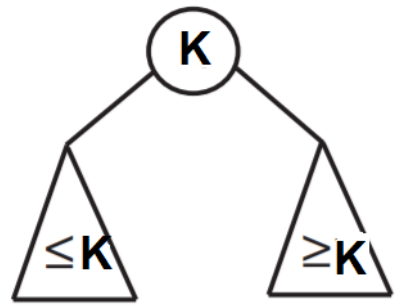
Recall Binary Trees

- A complete **binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are **as far left as possible**.

```
            O
          /   \
        O       O
       / \     / \
      O   O   O   O
     / \  /
    O   O O
```

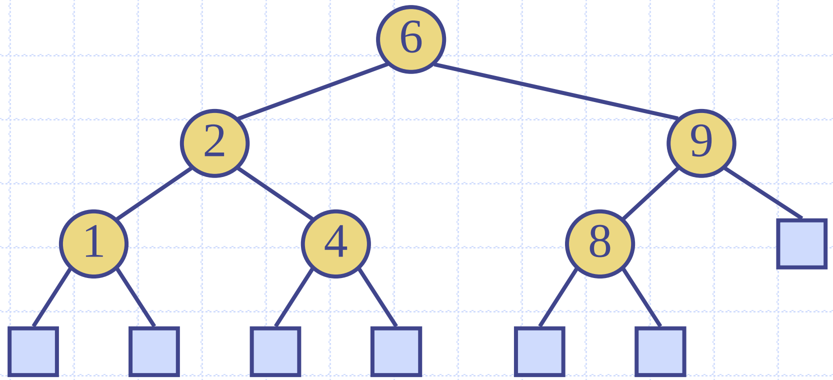
Binary Search Trees

- 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*.



Traversing Binary Trees

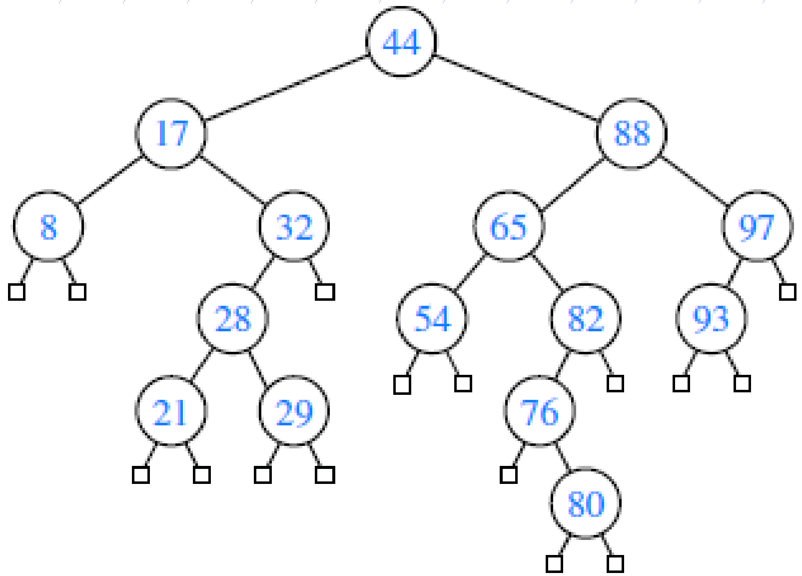
- Traversal strategy specifies the order in which the current node, the left subtree, and the right subtree are visited.
  - **Inorder** traversal: the **left subtree** is visited first, then the **current node** followed by the **right subtree**
  - We assume the left subtree always comes before the right subtree



  - The **Inorder** traversal gives: **1, 2, 4, 6, 8, 9**

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** (number of leaves in a full binary tree with n nodes is equal to $(n+1)/2$)

Search Tables vs Search Trees

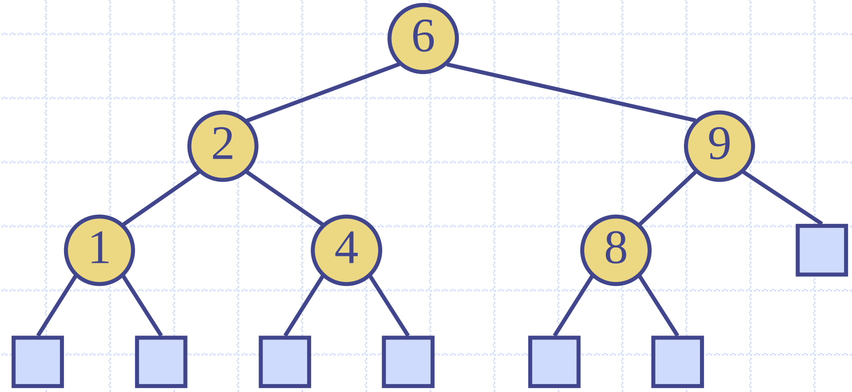
- **A search table is an ordered map** implemented by means of a sorted sequence
  - We store the items in **an array-based sequence**, **sorted by key**
  - We use an **external comparator for the keys**
- Performance:
  - **Searches** take $O(\log n)$ time, **using binary search**
  - **Inserting** a new item takes $O(n)$ time, since in the worst case we have to shift n items to make room for the new item
  - **Removing** an item takes $O(n)$ time, since in the worst case we have to shift $n-1$ items to compact the items after the removal

Search Tables vs Search Trees

- 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)
- We can use a **search-tree structure** to efficiently implement a ***sorted map***
  - *Achieve* **better performance for insertions and removals**

Binary Search Trees

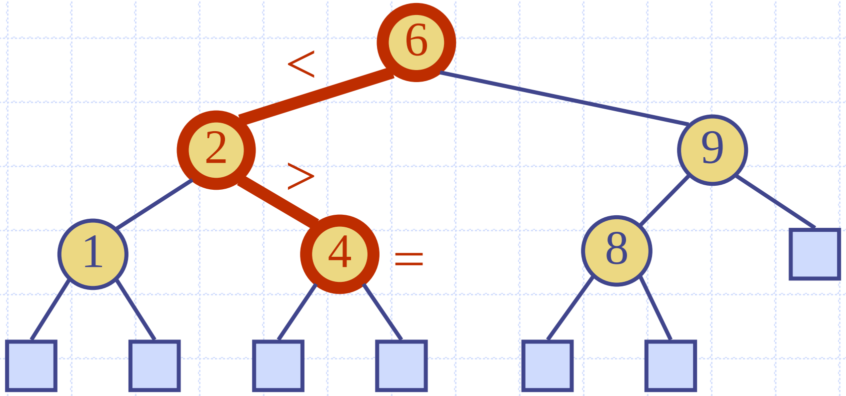
- A binary search tree is a binary tree **storing** keys (or key-value entries) **at its internal nodes** and satisfying the following property:
  - Let u, v, and w be three nodes such that u is in the left subtree of v and w is in the right subtree of v. We have $key(u) \leq key(v) \leq key(w)$
- External nodes do not store items
- An **inorder** traversal of a binary search trees **visits the keys in increasing order**
  - The left-most child has the smallest key
  - The right-most child has the largest key



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

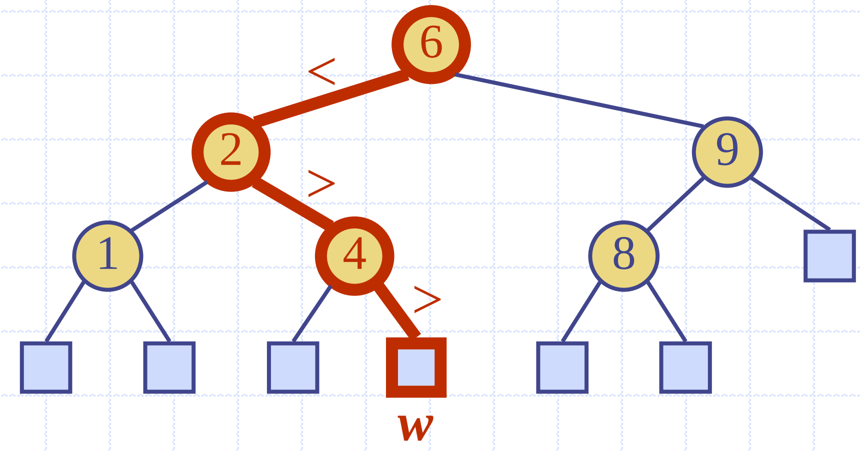
- The map operation **put**(k, v) begins with a search for an entry with key ***k***.
  - If found, that entry's existing value is reassigned.
  - Otherwise, the **new entry can be inserted into the underlying tree** by **expanding the leaf that was reached at the end of the failed search into an internal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

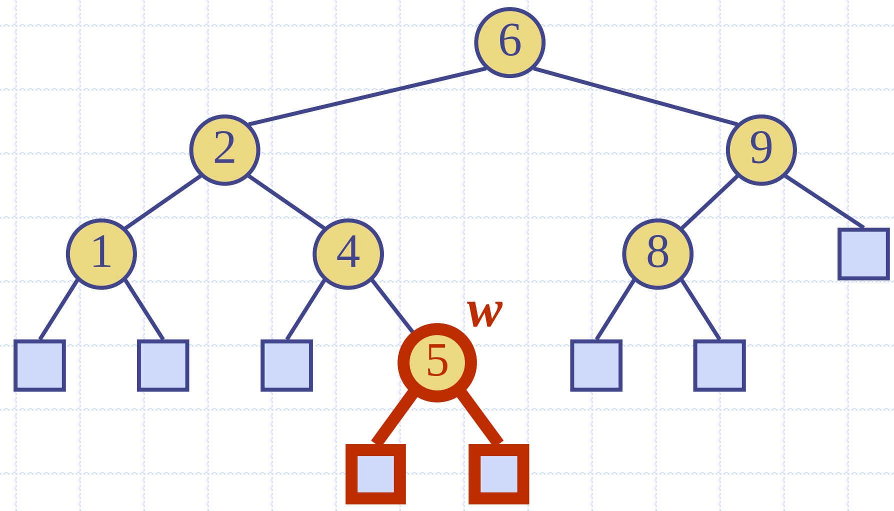
Algorithm TreeInsert(k, v):
    Input: A search key k to be associated with value v
    p = TreeSearch(root(), k)
    if k == key(p) **then**
        Change p's value to (v)
    else
        expandExternal(p, (k, v))

- 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**



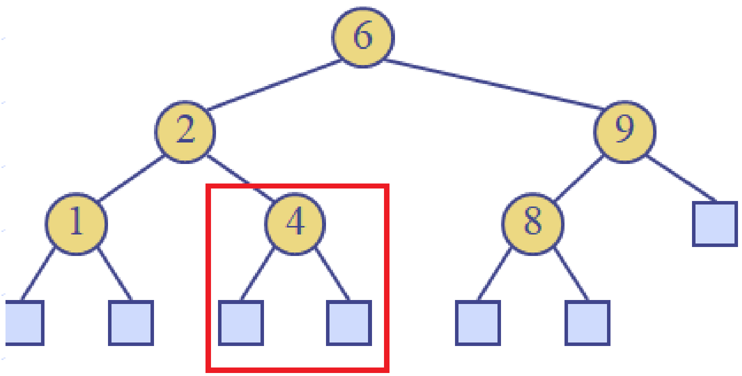


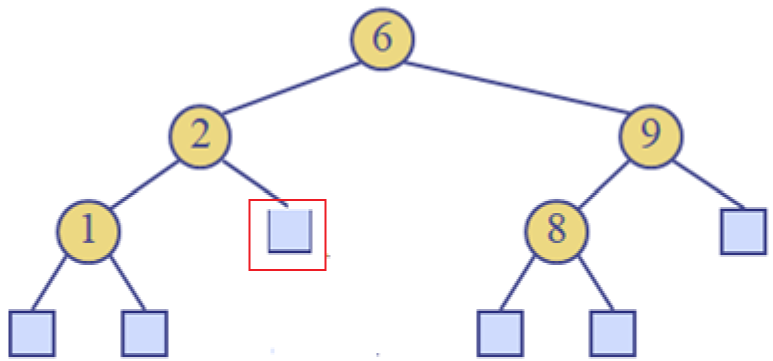
Deletion

- 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**).
    - node has **no children**
    - node has **one child**
    - node is **internal**
- 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 – node has no children

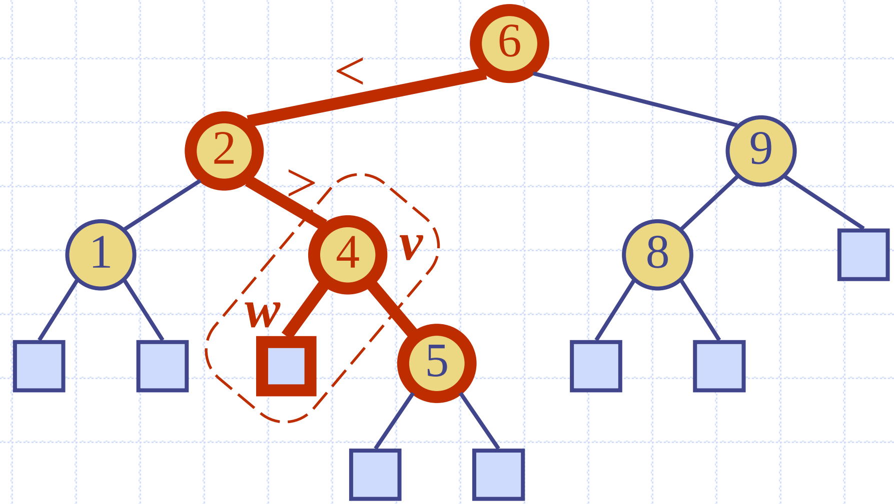
- Assume key ***k*** is in the tree, and let ***v*** be the node storing ***k***
  - *Remove v and its leafs (placeholders)*
  - *Replace v with a leaf (placeholder)*
- Example: **remove 4**

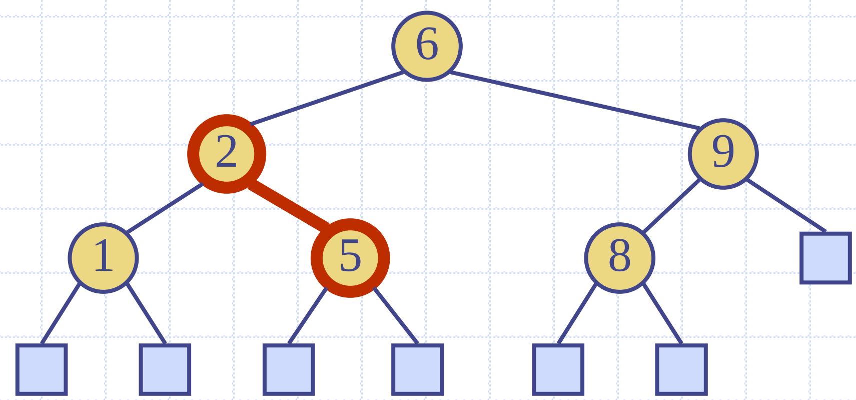




Deletion – node has one child

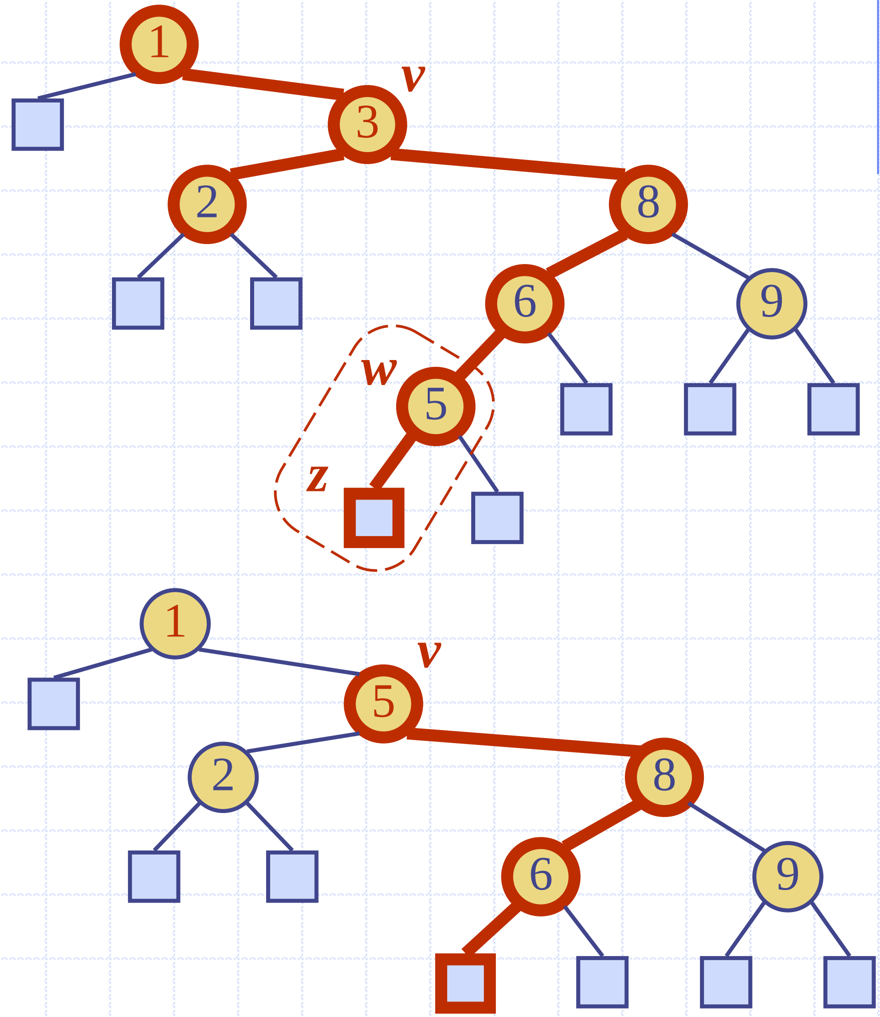
- To perform operation remove(k), we search for key k
- Assume key k is in the tree, and let let v be the node storing k
- If **node v has a leaf child** w, we remove v and w from the tree with operation removeExternal(w), which removes w and its parent
- Example: **remove 4**





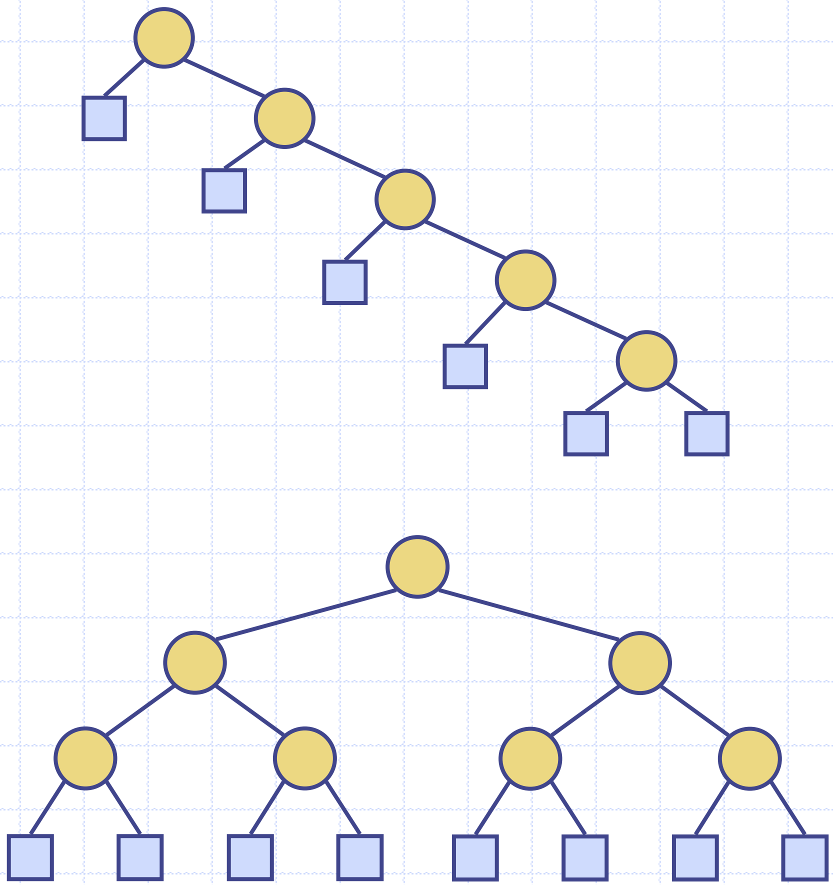
Deletion – node has two children

- 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, why?) 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 (**null**), and map entries stored only at internal nodes.

Java Implementation

```
/** An implementation of a sorted map using a binary search tree. */
public class TreeMap<K,V> extends AbstractSortedMap<K,V> {
  // To represent the underlying tree structure, we use a specialized subclass of the
  // LinkedBinaryTree class that we name BalanceableBinaryTree (see Section 11.2).
  protected BalanceableBinaryTree<K,V> tree = new BalanceableBinaryTree<>( );

  /** Constructs an empty map using the natural ordering of keys. */
  public TreeMap( ) {
    super( );                           // the AbstractSortedMap constructor
    tree.addRoot(null);                 // create a sentinel leaf as root
  }
  /** Constructs an empty map using the given comparator to order keys. */
  public TreeMap(Comparator<K> comp) {
    super(comp);                        // the AbstractSortedMap constructor
    tree.addRoot(null);                 // create a sentinel leaf as root
  }
  /** Returns the number of entries in the map. */
  public int size( ) {
    return (tree.size( ) − 1) / 2;      // only internal nodes have entries
  }
  /** Utility used when inserting a new entry at a leaf of the tree */
  private void expandExternal(Position<Entry<K,V>> p, Entry<K,V> entry) {
    tree.set(p, entry);                 // store new entry at p
    tree.addLeft(p, null);              // add new sentinel leaves as children
    tree.addRight(p, null);
  }

```

Java Implementation

```
// Omitted from this code fragment, but included in the online version of the code,
// are a series of protected methods that provide notational shorthands to wrap
// operations on the underlying linked binary tree. For example, we support the
// protected syntax root() as shorthand for tree.root() with the following utility:
protected Position<Entry<K,V>> root( ) { return tree.root( ); }

/** Returns the position in p's subtree having given key (or else the terminal leaf).*/
private Position<Entry<K,V>> treeSearch(Position<Entry<K,V>> p, K key) {
  if (isExternal(p))
    return p;                              // key not found; return the final leaf
  int comp = compare(key, p.getElement( ));
  if (comp == 0)
    return p;                              // key found; return its position
  else if (comp < 0)
    return treeSearch(left(p), key);       // search left subtree
  else
    return treeSearch(right(p), key);      // search right subtree
}
```

Code Fragment 11.3: Beginning of a TreeMap class based on a binary search tree.

Java Implementation

```
/** Returns the value associated with the specified key (or else null). */
public V get(K key) throws IllegalArgumentException {
  checkKey(key);                                    // may throw IllegalArgumentException
  Position<Entry<K,V>> p = treeSearch(root( ), key);
  rebalanceAccess(p);                               // hook for balanced tree subclasses
  if (isExternal(p)) return null;                   // unsuccessful search
  return p.getElement( ).getValue( );               // match found
}
/** Associates the given value with the given key, returning any overridden value.*/
public V put(K key, V value) throws IllegalArgumentException {
  checkKey(key);                                    // may throw IllegalArgumentException
  Entry<K,V> newEntry = new MapEntry<>(key, value);
  Position<Entry<K,V>> p = treeSearch(root( ), key);
  if (isExternal(p)) {                              // key is new
    expandExternal(p, newEntry);
    rebalanceInsert(p);                             // hook for balanced tree subclasses
    return null;
  } else {                                          // replacing existing key
    V old = p.getElement( ).getValue( );
    set(p, newEntry);
    rebalanceAccess(p);                             // hook for balanced tree subclasses
    return old;
  }
}
```

Java Implementation

```java
/** Removes the entry having key k (if any) and returns its associated value. */
public V remove(K key) throws IllegalArgumentException {
  checkKey(key);                                  // may throw IllegalArgumentException
  Position<Entry<K,V>> p = treeSearch(root(), key);
  if (isExternal(p)) {                            // key not found
    rebalanceAccess(p);                           // hook for balanced tree subclasses
    return null;
  } else {
    V old = p.getElement().getValue();
    if (isInternal(left(p)) && isInternal(right(p))) { // both children are internal
      Position<Entry<K,V>> replacement = treeMax(left(p));
      set(p, replacement.getElement());
      p = replacement;
    } // now p has at most one child that is an internal node
    Position<Entry<K,V>> leaf = (isExternal(left(p)) ? left(p) : right(p));
    Position<Entry<K,V>> sib = sibling(leaf);
    remove(leaf);
    remove(p);                                    // sib is promoted in p's place
    rebalanceDelete(sib);                         // hook for balanced tree subclasses
    return old;
  }
}
```

Code Fragment 11.4: Primary map operations for the TreeMap class.

Java Implementation

```
/** Returns the position with the maximum key in subtree rooted at Position p. */
protected Position<Entry<K,V>> treeMax(Position<Entry<K,V>> p) {
  Position<Entry<K,V>> walk = p;
  while (isInternal(walk))
    walk = right(walk);
  return parent(walk);                    // we want the parent of the leaf
}
/** Returns the entry having the greatest key (or null if map is empty). */
public Entry<K,V> lastEntry() {
  if (isEmpty()) return null;
  return treeMax(root()).getElement();
}
/** Returns the entry with greatest key less than or equal to given key (if any). */
public Entry<K,V> floorEntry(K key) throws IllegalArgumentException {
  checkKey(key);                          // may throw IllegalArgumentException
  Position<Entry<K,V>> p = treeSearch(root(), key);
  if (isInternal(p)) return p.getElement();  // exact match
  while (!isRoot(p)) {
    if (p == right(parent(p)))
      return parent(p).getElement();      // parent has next lesser key
    else
      p = parent(p);
  }
  return null;                            // no such floor exists
}
```

Java Implementation

```
/** Returns the entry with greatest key strictly less than given key (if any). */
public Entry<K,V> lowerEntry(K key) throws IllegalArgumentException {
  checkKey(key);                              // may throw IllegalArgumentException
  Position<Entry<K,V>> p = treeSearch(root( ), key);
  if (isInternal(p) && isInternal(left(p)))
    return treeMax(left(p)).getElement( );   // this is the predecessor to p
  // otherwise, we had failed search, or match with no left child
  while (!isRoot(p)) {
    if (p == right(parent(p)))
      return parent(p).getElement( );         // parent has next lesser key
    else
      p = parent(p);
  }
  return null;                                // no such lesser key exists
}
```

Code Fragment 11.5: A sample of the sorted map operations for the TreeMap class. The symmetrical utility, treeMin, and public methods firstEntry, ceilingEntry, and higherEntry are available online.

Java Implementation

```
/** Returns an iterable collection of all key-value entries of the map. */
public Iterable<Entry<K,V>> entrySet() {
  ArrayList<Entry<K,V>> buffer = new ArrayList<>(size());
  for (Position<Entry<K,V>> p : tree.inorder())
    if (isInternal(p)) buffer.add(p.getElement());
  return buffer;
}
/** Returns an iterable of entries with keys in range [fromKey, toKey). */
public Iterable<Entry<K,V>> subMap(K fromKey, K toKey) {
  ArrayList<Entry<K,V>> buffer = new ArrayList<>(size());
  if (compare(fromKey, toKey) < 0)              // ensure that fromKey < toKey
    subMapRecurse(fromKey, toKey, root(), buffer);
  return buffer;
}
private void subMapRecurse(K fromKey, K toKey, Position<Entry<K,V>> p,
                                   ArrayList<Entry<K,V>> buffer) {
  if (isInternal(p))
    if (compare(p.getElement(), fromKey) < 0)
      // p's key is less than fromKey, so any relevant entries are to the right
      subMapRecurse(fromKey, toKey, right(p), buffer);
    else {
      subMapRecurse(fromKey, toKey, left(p), buffer); // first consider left subtree
      if (compare(p.getElement(), toKey) < 0) {          // p is within range
        buffer.add(p.getElement());                   // so add it to buffer, and consider
        subMapRecurse(fromKey, toKey, right(p), buffer); // right subtree as well
      }
    }
}
```

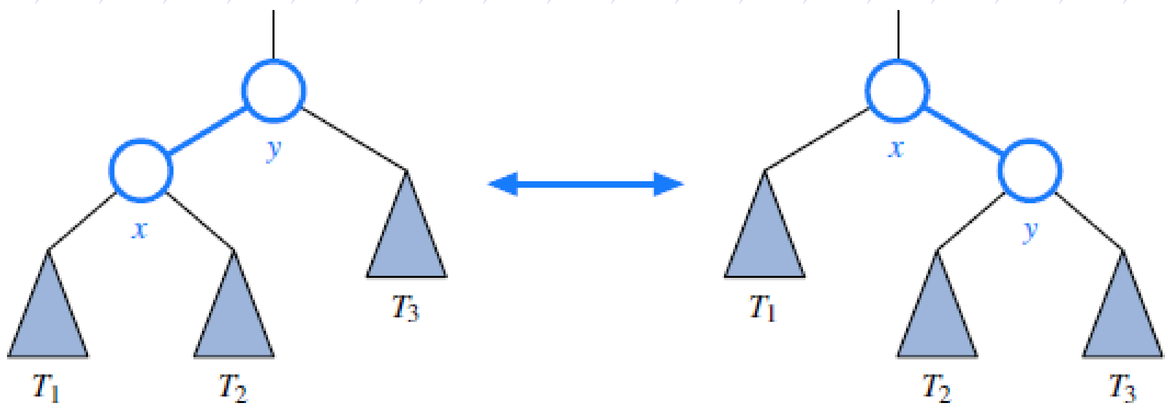
Code Fragment 11.6: TreeMap operations supporting iteration of the entire map, or a portion of the map with a given key range.

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

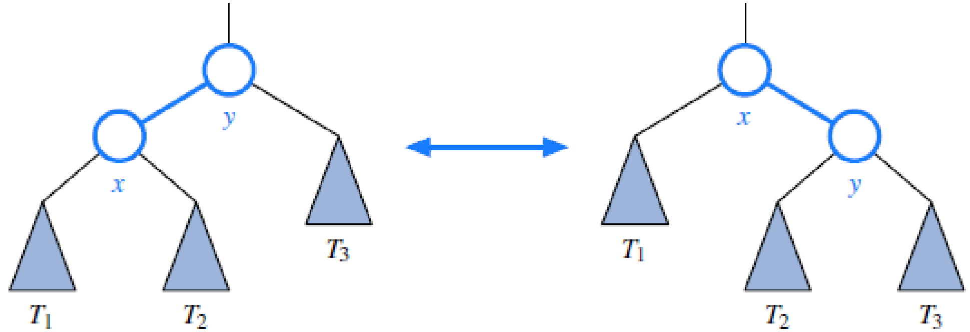
- The primary operation to rebalance a binary search tree is known as a ***rotation***.
- During a rotation, we "rotate" a child to be above its parent.



- 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

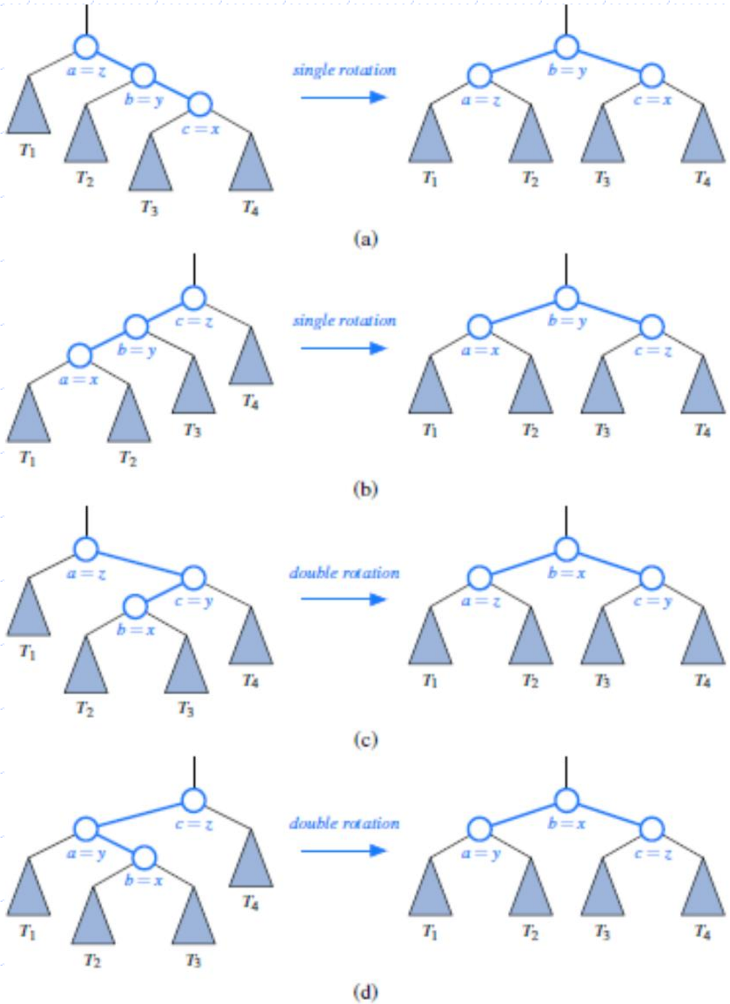
- A rotation can be performed to transform the left formation into the right, or the right formation into the left.
- Note that all keys in subtree *T1* have keys less than that of position *x*, all keys in subtree *T2* have keys that are between those of positions *x* and *y*, and all keys in subtree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