Binary Search Trees and Ordered Maps
Recall the Map ADT

- **get(k)**: if the map M has an entry with key k, return its associated value; else, return null
- **put(k, v)**: insert entry (k, v) into the map M; if key k is not already in M, then return null; else, return old value associated with k
- **remove(k)**: if the map M has an entry with key k, remove it from M and return its associated value; else, return null
- **size()**, **isEmpty()**
- **entrySet()**: return an iterable collection of the entries in M
- **keySet()**: return an iterable collection of the keys in M
- **values()**: return an iterator of the values in M
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 $k$
Ordered Map ADT

- `firstEntry()`: entry with smallest key
- `lastEntry()`: entry with largest key
- `ceilingEntry(k)`: entry with the least key value greater than or equal to k
- `floorEntry(k)`: entry with greatest key value less than or equal to k
Ordered Map ADT

- lowerEntry(k): entry with greatest key value strict less than k
- higherEntry(k): entry with least key value strictly higher than k
- subMap(k1, k2): Returns an Iterable of all entries with key k such that $k_1 \leq k < k_2$
Binary Search

- Binary search can perform nearest neighbor queries on an ordered map that is implemented with an array, sorted by key
  - 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 $n$ 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

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

- \( \text{key}(u) < \text{key}(v) < \text{key}(w) \)

External nodes do not store items

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

© 2014 Goodrich, Tamassia, Goldwasser Binary Search Trees
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: \texttt{get(4)}:
- Call \texttt{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

- To perform operation \( \text{put}(k, v) \), 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

To perform operation \texttt{remove}(k), we search for key \( k \).

Assume key \( k \) is in the tree, and 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 \texttt{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 $\text{removeExternal}(z)$
- 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.