Binary Search Trees
Ordered Dictionaries

Keys are assumed to come from a total order.

New operations:

- **first()**: first entry in the dictionary ordering
- **last()**: last entry in the dictionary ordering
- **successor(k)**: first entry with key greater than or equal to k
- **predecessor(k)**: last entry with key less than or equal to k
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 $key(u) \leq key(v) < key(w)$

- External nodes do not store items

An inorder traversal of a binary search tree 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 outcome of the comparison of $k$ with the key of the current node.

If we reach a leaf, the key is not found and we return null.

Example: find(4):
- Call $\text{TreeSearch}(4, \text{root})$

Algorithm $\text{TreeSearch}(k, v)$

```java
if $T$.isExternal$(v)$
   return $v$

if $k < \text{key}(v)$
   return $\text{TreeSearch}(k, T.left(v))$

else if $k = \text{key}(v)$
   return $v$

else 
   return $\text{TreeSearch}(k, T.right(v))$
```
Insertion

- To perform operation `insert(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

To perform operation \( \text{remove}(k) \), 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 \( \text{removeExternal}(w) \), which removes \( w \) and its parent.

Example: remove 4
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 a dictionary with $n$ items implemented by means of a binary search tree of height $h$
- the space used is $O(n)$
- methods find, insert and remove take $O(h)$ time

The height $h$ is $O(n)$ in the worst case and $O(\log n)$ in the best case.