Dictionary ADT

- The dictionary ADT models a searchable collection of key-element items.
- The main operations of a dictionary are searching, inserting, and deleting items.
- Multiple items with the same key are allowed.
- Applications:
  - address book
  - credit card authorization
  - mapping host names (e.g., cs16.net) to internet addresses (e.g., 128.148.34.101)

Dictionary ADT methods:
- findElement(k): if the dictionary has an item with key k, returns its element; else, returns the special element NO_SUCH_KEY.
- insertItem(k, o): inserts item (k, o) into the dictionary.
- removeElement(k): if the dictionary has an item with key k, removes it from the dictionary and returns its element; else, returns the special element NO_SUCH_KEY.
- size(), isEmpty()
- keys(), Elements()
Binary Search Tree

A binary search tree is a binary tree storing keys (or key-element pairs) 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 \( \text{key}(u) \leq \text{key}(v) \leq \text{key}(w) \)
- 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 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 \text{NO_SUCH_KEY}

Example: findElement(4)

Algorithm findElement(k, v)

\begin{align*}
\text{if } & T.\text{isExternal}(v) \\
\text{return } & \text{NO_SUCH_KEY} \\
\text{if } & k < \text{key}(v) \\
\text{return } & \text{findElement}(k, T.\text{leftChild}(v)) \\
\text{else if } & k = \text{key}(v) \\
\text{return } & \text{element}(v) \\
\text{else } & k > \text{key}(v) \\
\text{return } & \text{findElement}(k, T.\text{rightChild}(v))
\end{align*}

Insertion

To perform operation insertItem(k, o), we search for key \( k \)
- 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 removeElement(\( k \)), 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 removeAboveExternal(\( w \))
- 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 \( \text{key}(w) \) into node \( v \)
- we remove node \( w \) and its left child \( z \) (which must be a leaf) by means of operation removeAboveExternal(\( v \))
- 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 findElement, insertItem and removeElement take \( O(h) \) time
- The height \( h \) is \( O(n) \) in the worst case and \( O(\log n) \) in the best case