Lecture 8: Binary Search Trees
Outline

■ Return to the ordered dictionary ADT
  ■ will spend the next 2-3 weeks on several different ways of implementing an ordered dictionary ADT
  ■ Ordered dictionary is a very useful ADT and many interesting data structures were developed for it

■ Binary search tree (BST) data structure for Ordered Dictionary ADT implementation
  ■ simple but not the most efficient
  ■ forms the basis for other tree-based ordered dictionary implementations
Ordered Dictionaries

- Keys are ordered
- Basic operations
  - `find(k)`: return an entry with key `k`, if exists
  - `findAll(k)`: return an iterator of all entries with keys equal to `k`
  - `insert(k,v)`: insert an entry with key `k` and value `v`
  - `remove(e)`: remove an entry `e` and return it
Ordered Dictionaries

- Operations specific to Ordered Dictionary
  - `first()`: first entry in the dictionary ordering
  - `last()`: last entry in the dictionary ordering
  - `successors(k)`: iterator of entries with keys greater than or equal to k; increasing order
  - `predecessors(k)`: iterator of entries with keys less than or equal to k; decreasing order
Ordered dictionary via Search Table

- Performance:
  - **find** is $O(\log n)$ with using binary search
  - **insert** is $O(n)$
  - **remove** is $O(n)$

- Would like to have ordered dictionary where find, insert, remove are $O(\log n)$
- Start with binary search tree
- Modify it to eventually get ADT with required efficiency
A binary search tree is a binary tree storing entries \((k,v)\) at internal nodes and satisfying

\[\text{key}(u) \leq \text{key}(z) \leq \text{key}(w)\]

whenever if \(u\) is in the left subtree of \(z\) and \(w\) is in the right subtree of \(z\).

- Use proper binary trees
- Do not store entries in external nodes
  - simplifies algorithms
- inorder traversal visits keys in non-decreasing order
To search for key $k$, trace a downward path starting at root, turning left or right, based on comparison.

If reach a leaf, the key is not present, return this leaf.

- entry with key $k$, if existed, would belong at this leaf.

Example: find(4):
- Call TreeSearch(4,root)

Algorithm TreeSearch($k,w$)

Input: key $k$ and a tree node $w$

Output: node with key $k$ or leaf where search stopped

if $T$.isExternal($w$)
    return $w$ // no node with key $k$

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

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

else // we know here $k > \text{key}(w)$
    return TreeSearch($k,T$.right($w$))
Insertion Case 1

- First search for key $k$ using TreeSearch
- There are 2 cases
- Case 1: $k$ is not already in the tree
  - insert entry $(k,v)$ at leaf node $w$ be the leaf returned by the search
  - expand $w$ into an internal node
  - example: $\text{insert}(5,v)$

```
       6
      /   \
  2       9
 /     /   |
1     4     8
 |     |
 w  
```

```
       6
      /   \
  2       9
 /     /   |
1     4     8
 |     |
  
```
Insertion Case 2

- Case 2: an entry with key $k$ is already in the tree
  - $w$ returned by the search is internal node s.t. $\text{key}(w) = k$
  - can insert entry $(k,v)$ into either the left or the right subtree of $w$
  - we choose the left subtree
  - use recursive call $\text{insertTree}(k,v,T.\text{left}(w))$

- Example insert$(2,v)$
Algorithm TreeInsert(k,v,u)
Input: A search key k, value v, and node u of tree T
Output: node z storing entry(k,v)
  z = TreeSearch(k, u)
  if !(T.isExternal(z)) // key k is already in the tree
      return TreeInsert(k, v, T.left(z))
  T.insertAtExternal(z, (k,v)) // insert (k,v) at node z
  return z

insertAtExternal(z,(k,v)) adds 2 leaf children to z and stores (k,v) at z
Deletion Case 1

- First search for key $k$
- Assume key $k$ is present, let $v$ be the node storing $k$
- There are 2 cases
- Case 1: node $v$ has a leaf child $w$
  - remove $v$ and $w$ from tree with method $\text{removeAtExternal}(w)$
  - Example: remove 4
Algorithm RemoveAtExternal(w)

Input: leaf node w, removes w and its parent

p = w.parent();  // also need to remove p

if p.leftChild() = w
    nodeToRelink = p.right
else nodeToRelink = p.left

if p = root  // we removing root node, set new root
    root = nodeToRelink
    root.parent = null
else  // p has a parent in this case
    if p = (p.parent()).leftChild()  // p is a left child
        p.parent().leftChild = nodeToRelink
    else  // p is a right child
        p.parent().rightChild = nodeToRelink
    nodeToRelink.parent = p.parent
Algorithm RemoveAtExternal(w)
Input: leaf node w, removes w and its parent

\[
p = w\.parent(); \quad //\text{also need to remove } p
\]

if \( p\.leftChild() = w \)
    nodeToRelink = p.right
else nodeToRelink = p.left

if \( p = \text{root} \quad //\text{we removing root node, set new root} \)
    root = nodeToRelink
    root\.parent = null
else \quad //\text{p has a parent in this case}
    if \( p = (p\.parent()).leftChild() \quad //\text{p is a left child} \)
        p\.parent().leftChild = nodeToRelink
    else \quad //\text{p is a right child}
        p\.parent().rightChild = nodeToRelink
nodeToRelink\.parent = p\.parent
Algorithm RemoveAtExternal(w)
Input: leaf node w, removes w and its parent

\[
p = w.\text{parent}(); \quad \text{// also need to remove p}
\]

if \( p.\text{leftChild}() = w \)
\[
\text{nodeToRelink} = p.\text{right}
\]
else \( \text{nodeToRelink} = p.\text{left} \)

if \( p = \text{root} \quad \text{// we removing root node, set new root} \)
\[
\text{root} = \text{nodeToRelink}
\]
\[
\text{root.}\text{parent} = \text{null}
\]
else \quad \text{// p has a parent in this case}

if \( p = (p.\text{parent}()).\text{leftChild}() \quad \text{// p is a left child} \)
\[
p.\text{parent}().\text{leftChild} = \text{nodeToRelink}
\]
else \quad \text{// p is a right child}
\[
p.\text{parent}().\text{rightChild} = \text{nodeToRelink}
\]
\[
\text{nodeToRelink.}\text{parent} = p.\text{parent}
\]
Deletion Case 2

Case 2: node $v$ storing entry to be removed has no leaf child

- replace entry at $v$ with entry whose key $k$ preserves binary tree order
  - either largest key entry in left subtree of $v$ or smallest key entry in right subtree of $v$
  - we choose to take the smallest key entry in the right subtree of $v$
- copy entry at node $w$ into node $v$
- remove node $w$ and its left child $z$ (if $z$ is a leaf) with $\text{removeExternal}(z)$

Example: remove 3
Performance

- For **find**, **insert** and **remove**
  - spend $O(1)$ time at each node visited
  - in the worst case, visit $O(h)$ nodes
  - methods **find**, **insert** and **remove** take $O(h)$ time
  - the space used is $O(n)$

- Height $h$ is $O(n)$ in the worst case and $O(\log n)$ in the best case

- Worst case performance is $O(n)$
Binary Search Trees

- Worst case performance of BST is the same or worse compared to Search Table
  - `find()` is worse
- `find`, `insert` and `remove` are $O(h)$ in BST
  - $h$ is the height of the tree
- If insure the height of BST is $O(\log n)$, BST will be much more efficient
  - the worst case complexity for `find`, `insert`, `remove` will be $O(\log n)$
- To insure height $h$ of BST is $O(\log n)$, the tree must be **balanced**
  - for any node, the number of descendants in its left subtree should be roughly the same as the number of descendants in its right subtree
- How to implement balanced trees?
- One way is with **AVL trees**