Lecture 7: Binary Search Trees

- Motivation
- Definition
- Binary search tree for ordered dictionary implementation

Courtesy to Goodrich, Tamassia and Olga Veksler

Instructor: Yuzhen Xie
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 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

- Recall basic operations
  - \texttt{find(k)}: return an entry with key \texttt{k}, if exists
  - \texttt{findAll(k)}: return an iterator of all entries with keys equal to \texttt{k}
  - \texttt{insert(k,v)}: insert an entry with key \texttt{k} and value \texttt{v}
  - \texttt{remove(e)}: remove an entry \texttt{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
Recall Search Tables

- A search table is a dictionary implemented with a sorted sequence
  - Store dictionary entries in an array-based sequence, sorted by key

- Performance:
  - find takes $O(\log n)$ time, using binary search
    - good efficiency
  - insert takes $O(n)$ time, in the worst case have to shift $n/2$ items to make room for the new item
    - not great efficiency
  - remove take $O(n)$ time, in the worst case have to shift $n/2$ items to compact the items after removal
    - not great efficiency

- lookup table is effective only for dictionaries of small size or for dictionaries on which searches are most common operations

- Can we implement a dictionary so that each find, insert, and remove operations take not more than $O(\log n)$ time?
  - YES! With special types of data structures based on trees
  - We’ll start today and spend quite a bit of time on this
A binary search tree is a binary tree storing entries \((k,v)\) at its internal nodes and satisfying the following property:

- Let \(u\), \(z\), and \(w\) be three nodes such that \(u\) is in the left subtree of \(z\) and \(w\) is in the right subtree of \(z\). Then
  \[\text{key}(u) \leq \text{key}(z) \leq \text{key}(w)\]

- In other words, for any node \(z\), all nodes in its left subtree have smaller (or equal) keys, all nodes in its right subtree have larger (or equal) keys.

- Will use proper binary trees
- Will not store entries in external nodes
  - this is done purely to simplify algorithms
- An \textit{inorder traversal} of a binary search tree visits the keys in non-decreasing order.
Search

- Search is similar to binary search in ordered array
- To search for a key \( k \), we trace a downward path starting at the root
  - 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 the leaf node where entry with key \( k \) would belong if it existed
- Example: \texttt{find}(4):
  - Call \texttt{TreeSearch}(4, root)

Algorithm \texttt{TreeSearch}(k, w)

Input: key \( k \) and a tree node \( w \)
Output: node with key \( k \) or leaf where search stopped

\begin{align*}
\text{if } & \text{T.isExternal}(w) \\
& \quad \text{return } w \quad \text{// no node with key } k \\
\text{if } k < \text{key}(w) \\
& \quad \text{return } \text{TreeSearch}(k, \text{T.left}(w)) \\
\text{else if } k = \text{key}(w) \\
& \quad \text{return } w \\
\text{else } & \text{// we know here that } k > \text{key}(w) \\
& \quad \text{return } \text{TreeSearch}(k, \text{T.right}(w))
\end{align*}
Insertion: Case 1

- To perform operation `insert(k,v)`, search for key `k` using TreeSearch
- There are 2 cases
  - Case 1: `k` is not already in the tree
    - Let `w` be the leaf reached by the search. Then `w` is the correct place to insert entry with key `k`, inserting at `w` preserves the binary tree order
    - Insert `k` at node `w` and expand `w` into an internal node
    - Example: `insert(5,v)`
Insertion Case 2:

- Case 2: an entry with key $k$ is already in the tree
  - Then $w$ returned by the search is an **internal** node s.t. $\text{key}(w) = k$
  - To preserve binary tree order, can insert the entry $(k, v)$ into either the left or the right subtree of node $w$
  - Let’s choose left subtree
    Recursively call $\text{insertTree}(k, v, \text{left}(w))$

- Example insert($2, v$)
Algorithm $\text{TreeInsert}(k, v, u)$

Input: A search key $k$, value $v$, and node $u$ of tree $T$

Output: node $z$ in the subtree rooted at $u$ that stores entry $(k, v)$

$z = \text{TreeSearch}(k, u)$

if !(T.isExternal($z$)) // key $k$ is already in the tree
    return $\text{TreeInsert}(k, v, T.left(z))$

$T.insertAtExternal(z, (k, v))$ // insert $(k, v)$ at external node $z$

return $z$

$\text{insertAtExternal}(z, (k, v))$ adds 2 leaf children to node $z$ and stores $(k, v)$ at node $z$. 
Deletion Case 1

- To perform operation `remove(k)`, we search for key `k`.
- Assume key `k` is in the tree, and let `v` be the node storing `k`.
- There are 2 cases:
  - Case 1
    - if node `v` has a leaf child `w`, we remove `v` and `w` from the tree with operation `removeAtExternal(w)`, which removes `w` and its parent.
    - Example: remove 4
Algorithm RemoveAtExternal(w)
Input: leaf node w. Removes w and its parent node

```
p = w.parent();  //also need to remove p
if p.leftChild() == w
    nodeToRelink = p.right
else  nodeToRelink = p.left
if p == root   //node we removing is a root, set new root
    root = nodeToRelink
    root.parent = null  // set new root to null parent
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
```

RemoveAtExternal(w) is a member method of the Tree class
Deletion Case 2

- Case 2: key \( k \) to be removed is stored at a node \( v \) whose children are both internal
  - need replace entry at \( v \) with an entry whose key \( k \) preserves binary tree order
    - either largest key entry in left subtree of \( v \) or smallest entry in right subtree of \( v \)
  - find internal node \( w \) that follows \( v \) in an inorder traversal (entry with smallest key in the right subtree of node \( v \))
  - Copy entry at node \( w \) into node \( v \). The tree order is preserved
  - remove node \( w \) and its left child \( z \) (which must be a leaf) by means of operation \( \text{removeAtExternal}(z) \)

- Example: remove 3
Performance

- Consider a dictionary with $n$ items implemented by means of a binary search tree of height $h$
- The analysis of find, insert and remove is similar
  - We spend $O(1)$ time at each node visited
  - In the worst case, visit $O(h)$ nodes
- Thus
  - 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
Discussion about Binary Search Trees

- Binary search trees do not have a better worst case performance than the ordered search table
  - In fact, the worst case complexity of findKey() is worse for binary trees than the ordered search table
- Basic operations on binary search trees are $O(h)$
  - $h$ is the height of the tree
- Thus, if we find a way to make sure the height $h$ of a binary search tree is always $O(\log n)$, then search tree will be much more efficient, the worst case complexity for find, insert, remove will be $O(\log n)$, significantly better than insert and remove in the search table
- To make sure height $h$ of a tree is always $O(\log n)$, the tree must be “balanced”, that is for any node, the number of descendants in its left subtree should not be too different from the number of descendants in its right subtree
- How to implement “balanced” trees which allows the operations that we need?
- One way is with AVL trees