CSE 326 Autumn 2006
Section 4 Notes

1 Binary Search Tree Summary

- All search trees below are binary in the sense that each node has at most two children, so we will call them collectively *binary search trees* (BSTs). However, the basic, unbalanced binary search tree is also called the BST. It will be clear from the context which one is meant.
- In the table below, n represents the number of nodes in a tree.
- All BSTs satisfy the constraint of *BST order*, that is, $key(leftchild) < key(parent) < key(rightchild)$.
- The BST order does not use $\leq$ inequalities because we can handle duplicates by keeping a counter at each node. This is just a performance hack, but note that the path between two copies with the same key will only pass between more copies of that key.
- The running time analysis is always worst-case, but amortized worst-case is averaged over any sequence of operations.
- The space requirements are those beyond what is required to store the key and child pointer at each node, which is the same $O(2n)$ for all BSTs.
- Constant factors are given for comparison but are actually implementation-dependent.
- Logarithms are base 2 unless otherwise specified, but all logarithmic bases are asymptotically the same.
- Insert is a special-case of Contains which adds the new node at the first null child reached in a traversal.
- Remove is a special-case of Contains on the left and right subtrees in which the desired key is first found, then removed, and finally the remaining tree is rearranged to fill the "hole."

BST Type	Insert	Remove	Contains	Space	Analysis
Basic BST	$O(n)$	$O(n)$	$O(n)$	$O(1)$	Worst-case
AVL Trees	$O(2\log_2 n)$	$O(2\log_2 n)$	$O(2\log_2 n)$	$O(n)$	Worst-case
Splay Trees	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(1)$	Amortized

Basic BST: In the worst-case, the tree can be completely unbalanced with all nodes in a linear chain. However, no extra information needs to be maintained at each node. This is analogous to a basic binary heap.

AVL Trees: We can force the trees to be balanced by keeping some extra information (the height) at each node, trading off increased space for decreased running time. This is analogous to leftist heaps versus the basic binary heap.

The height information is used to maintain the *AVL property*, which states that the heights of the left and right subtrees never differ by more than 1. In symbols: $|height(left) - height(right)| \leq 1$. We can show that this bounds the height of the entire tree to $O(\log n)$, which improves the running times of Contains and by extension all other operations.

The AVL property is maintained by single and double rotations.

Splay Trees: Similar to AVL trees, but the last accessed item is always moved to the top, using *zig-zag* rotations (which are the same as AVL double rotations) or *zig-zig* rotations. There are two symmetric cases for both, or four total: zig-zag to the left then right, zig-zag to the right then left, zig-zig to the left, zig-zig to the right. This gives an amortized worst-case running time of $O(\log n)$.

2 Pointers, predecessors, successors, FindMin, FindMax

Because of the BST order, you know that you can sort all the nodes using an in-order traversal:

- Output the left subtree recursively
- Output the parent
- Output the right-subtree recursively

When you remove an item from a search tree, you must replace it with either the previous item or the next item in the sorted order. The previous sorted item is also the rightmost child of the left subtree and is called the *predecessor*. The next sorted item is also the leftmost child of the right subtree and is called the *successor*.

There are many operations which we could do in $O(1)$ time if we kept extra pointers with every BST. Two important examples are

1. finding the predecessor or successor of a node when we remove it
2. finding the minimum or maximum element

Updating these pointers takes $O(\log n)$ time because the tree must be re-traversed to find the new pointer target if new items are added or old items removed. To find the minimum (maximum) element in a tree, we only have to keep traversing the left (right) child until we reach a null and then return the last non-null node. To find the predecessor (successor), we have to traverse the left child and then go right until we reach null (or traverse the right child and then go left until we reach null). These operations can be "absorbed" into the running times of Insert and Remove which are already $\Omega(\log n)$ for any BST.

If we had doubly-linked lists, we could update the pointers in $O(1)$ also, but this would increase the storage needed at each node (one extra child pointer). For the minimum or maximum element, we could keep $O(1)$ new pointers with the tree, but for predecessors and successors, we would need to keep two extra pointers *at each node*. This is a total $O(3n)$ of extra space. Keep this time-space tradeoff in mind if you need to optimize the above operations and have lots of memory to spare.