1 Problems with BSTs

- As we remember, BSTs can quickly become unbalanced through a series of insertions
- An unbalanced search tree can cause insert, remove, and find operations to become slow, possibly as slow as a linked-list (or slower, due to constant factor overhead doing comparisons)

2 General Balanced Tree Goal

- We’d like to ensure some sort of balance in the search tree structure so that our operations are $O(\log n)$
- Typically, a balanced tree is one in which leaves are mostly at the same depth (or are all at the same depth)
- There are a few ways of doing this:
  - When tree operations are performed, check whether the tree has lost balance, and if it has, repair it to restore balance (AVL Trees)
  - Ensure that at all times the tree is balanced by performing operations that maintain leaf nodes at the same level at all times (B-Trees)
  - Perform a little bit of tree restructuring upon every operation so that the tree never goes out of balance (splay trees)
  - Probabilistically ensure that the tree remains in balance (skip lists – if you can call them trees)

3 Tree Rotations

- Given a tree, we’d like to rotate it left or right appropriately
- Rotations are symmetric – a left rotate followed by a right rotate yields the original tree

4 AVL Trees

- Binary Search Trees in which the following property is true: For all internal nodes, the heights of the child trees can differ by at most 1 (we saw this in Homework 3)
- When we find that the tree is out of balance, we perform rotations on it to restore balance
5 AVL Tree Insertion/Deletion

- We begin an AVL Tree insert operation as we did with Binary Search Trees - simply perform a find operation and place the key we are inserting at the desired location
- Insertion of an item may increase the height of subtrees by one, which may violate our balance requirement
- To decide what to rotate to rebalance the tree, we start at the node we just inserted and travel up the tree towards the root, looking for nodes that violate the balance requirement
- The same is possible when deleting keys from the tree – it may go out of balance
- We *restructure* out of balance nodes
- For diagrams, see Data Structures (Into Java) 9.3

6 Announcements

- Homework #5 posted
- Reading: Data Structures (Into Java) – 9.1, 9.3; Goodrich and Tamassia – 9.2, 9.4 (optional)

7 B-Trees

- Balanced tree structures that always maintain balance in which all empty children occur at the same level in the tree
- In a B-Tree of order \( m \), all nodes have \( m \) or fewer children (a special case of which are (2,4) trees, which are order 4 B-Trees, often called 2-3-4 trees because all nodes have 2, 3, or 4 children)
- All nodes other than the root node have at least \( m/2 \) children
- Each node may have up to \( m - 1 \) keys, on either side of which are children (thus allowing for the maximum of \( m \) children per node)
- Keys are ordered much as they are in a Binary Search Tree – keys to the left are less than keys to the right (and subsequently, keys in children to the left of a key \( k \) are less than \( k \) and those to the right of \( k \) are greater than \( k \))
- We’ll say that the *arity* of a node is the number of children it has
- For diagrams, see Data Structures (Into Java) 9.1

8 B-Tree Insertion

- To insert an item with key \( k \), we search for \( k \) in the B-Tree, finding a node at the bottom of the tree at which to insert
- We insert the item, ignoring the order of the B-Tree
- If the insertion caused the node to *overflow*, we *split* the node, moving one of the keys \( k_p \) up to the parent, and moving the keys to the left and right of \( k_p \) into new tree nodes placed appropriately as children to the left and right of \( k_p \) in the parent
- If a split causes another overflow, we split again, until all nodes satisfy our requirements
9 B-Tree Split

- To perform a split of a node $v$, we select the key at index $v.arity() / 2$, which we'll call $s$, essentially splitting the node in half.
- We create a new node $v_2$ and store all keys and associated children to the right of $s$ in it.
- We move $s$ into the parent node of $v$ and place $v_2$ as a child to the right of $s$ and $v$ to the left of $s$.
- If $v$ was the root node, we create a new root node $r$ and make $v$ and $v_2$ children of $r$.
- Finally, if the parent node of $v$ overflows due to the split operation, we split it as well.

10 B-Tree Deletion

- To delete an item with key $k$, we push a value downwards where deletion is easy, and remove it.
- Deletion only actually occurs at a node with all empty children.
- When doing this, we must merge child nodes, remove the value we want to delete, and then possibly split or merge nodes to restore the B-Tree.

11 B-Tree Merge

- To perform a merge given a key $k$ and its corresponding children, we merge the child immediately to the left of $k$ with the child immediately to the right of $k$, inserting $k$ in between.
- The merge operation will likely make the child temporarily too large, but after deletion has occurred, we split it as needed.
- We continue merging (regrouping) nodes up the B-Tree from the point of deletion.