Balancing Trees

Tricks to amaze your friends
Background

- BSTs were introduced because in theory they give nice fast search time.
- We have seen that depending on how the data arrives the tree can degrade into a linked list.
- So what is a good programmer to do.
- Of course, they are to balance the tree.
One idea would be to get all of the data first, and store it in an array.
Then sort the array and then insert it in a tree.
Of course this does have some drawbacks.
Ok, we need another idea.
DSW Trees

- Named for Colin Day and then for Quentin F. Stout and Bette L. Warren, hence DSW.
- The main idea is a rotation
- `rotateRight( Gr, Par, Ch )`
  - If Par is not the root of the tree
    - Grandparent Gr of child Ch, becomes Ch’s parent by replacing Par;
  - Right subtree of Ch becomes left subtree of Ch’s parent Par;
  - Node Ch acquires Par as its right child
Maybe a picture will help

(a)

(b)
More of the DSW

- So the idea is to take a tree and perform some rotations to it to make it balanced.
- First you create a backbone or a vine
- Then you transform the backbone into a nicely balanced tree
Algorithms

- createBackbone(root, n)
  - Tmp = root
  - While ( Tmp != 0 )
    - If Tmp has a left child
      - Rotate this child about Tmp
      - Set Tmp to the child which just became parent
    - Else set Tmp to its right child

- createPerfectTree(n)
  - M = $2^{\lfloor \log(n+1) \rfloor-1}$
  - Make n-M rotations starting from the top of the backbone;
  - While ( M > 1 )
    - M = M/2;
    - Make M rotations starting from the top of the backbone;
Maybe some more pictures
Wrap-up

- The DSW algorithm is good if you can take the time to get all the nodes and then create the tree.
- What if you want to balance the tree as you go?
- You use an AVL Tree.
AVL Trees

- Named for Adel’son-Vel’skii and Landis, hence AVL
- The heights of any subtree can only differ by at most one.
- Each node will indicate balance factors.
- Worst case for an AVL tree is 44% worst, then a perfect tree.
- In practice, it is closer to a perfect tree.
What does an AVL do?

- Each time the tree structure is changed, the balance factors are checked and if an imbalance is recognized, then the tree is restructured.
- For insertion there are four cases to be concerned with.
- Deletion is a little trickier.
AVL Insertion

Case 1: Insertion into a right subtree of a right child.
  - Requires a left rotation about the child

Case 2: Insertion into a left subtree of a right child.
  - Requires two rotations
    - First a right rotation about the root of the subtree
    - Second a left rotation about the subtree’s parent
Some more pictures

Figure 6.42  Balancing a tree after insertion of a node in the left subtree of node Q.
Deletion

- Deletion is a bit trickier.
- With insertion after the rotation we were done.
- Not so with deletion.
- We need to continue checking balance factors as we travel up the tree
Deletion Specifics

- Go ahead and delete the node just like in a BST.
- There are 4 cases after the deletion:
Cases

- Case 1: Deletion from a left subtree from a tree with a right high root and a right high right subtree.
  - Requires one left rotation about the root

- Case 2: Deletion from a left subtree from a tree with a right high root and a balanced right subtree.
  - Requires one left rotation about the root
Case 3: Deletion from a left subtree from a tree with a right high root and a left high right subtree with a left high left subtree.

- Requires a right rotation around the right subtree root and then a left rotation about the root.

Case 4: Deletion from a left subtree from a tree with a right high root and a left high right subtree with a right high left subtree.

- Requires a right rotation around the right subtree root and then a left rotation about the root.
Definitely some pictures
Self-adjusting Trees

- The previous sections discussed ways to balance the tree after the tree was changed due to an insert or a delete.
- There is another option.
- You can alter the structure of the tree after you access an element
  - Think of this as a self-organizing tree
Splay Trees

- You have some options
- When you access a node, you can move it to the root
- You can also swap it with its parent
- When you modify the move to root strategy with a pair of swaps you get a splay tree
Splay Cases

- Depending on the configuration of the tree you get three cases
- Case 1: Node R’s parent is the root
- Case 2: Node R is the left child of its parent Q and Q is the left child of its parent R
- Case 3: Node R is the right child of its parent Q and Q is the left child of its parent R
Splay Algorithm

- **Splaying( P, Q, R )**
  - While R is not the root
    - If R’s parent is the root
      - Perform a singular splay, rotate R about its parent
    - If R is in a homogenous configuration
      - Perform a homogenous splay, first rotate Q about P and then R about Q
    - Else
      - Perform a heterogeneous splay, first rotate R about Q and then about P
Semisplaying

- You can modify the traditional splay techniques for homogenous splays.
- When a homogenous splay is made instead of the second rotation taking place with R, you continue to splay with the node that was previously splayed.
Example