CSE 326
Lecture 15

Spring Quarter 1995

Summary of AVL Trees

- The worst cast height of an AVL tree is approximately 1.44 log n
- Complexity of Operations

  Insert - O(log n)
  Delete - O(log n)
  Find - O(log n)

- Features

  - At most one rotation needed to rebalance in insertion
  - Multiple rotations may be needed to rebalance on deletion
  - Compatible with lazy deletion.

- Alternatives to AVL (height balanced) trees

  - weight balanced trees - not nearly as nice as height balanced trees
  - red-black trees - a constant number of rotations needed to rebalance on both insertion and deletion
  - splay trees - self adjusting binary trees with several good properties but does not have worst case O(log n) per operation.

Batch Deletion in AVL Trees

- Recall that we simple marked nodes as deleted.
- There are two steps

  1. Do an inorder traversal putting all the unmarked nodes in an array.

  2. Do a divide and conquer construction of a balanced tree, which will have the AVL tree property.

- Suppose we have m deletions in a tree of size n.

  Ordinary AVL deletion time is O(m log n + n)

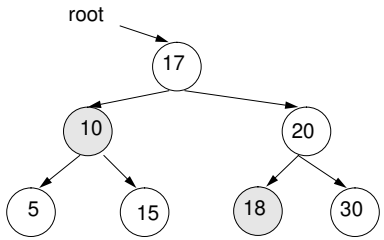
  Batch deletion is O(n)

Listing the Unmarked Items

- List(n,i) puts the items in the tree rooted at n in order in the array A starting at index i.
- List(n, ref i) {

```
if not(n=null) do {
  List(n.left, i);
  if not(n.mark) then {A[i] := n.data; i := i+1};
  List(n.right,i) }}
```

- Call List(root,1)



Creating the AVL Tree

- Assume A[1..N] is a sorted array. The procedure Build(i,j) will return a balanced tree whose nodes have data A[i],A[i+1],...,A[j].
- Build(i,j) {

```
if j < i then return(null)
else {
  k := (i+j)/2;
  return(New_Node(A[k],Build(i,k-1),Build(k+1,j)))  }}
```

- Call Build(1,N) to create the entire tree.
- Fill in the height fields correctly with a postorder traversal.
- Running time is O(n) which is better than inserting items individually, which would be O(n log n).

Example of Creating An AVL Tree

- 5 , 8, 10, 15, 17, 20, 30, 35, 40

Time Complexity of Building a Balanced Tree

$$T(0) \le b$$

$$T(n) \le T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + c$$

- We can prove by induction that T(n) < (b+c)n + b.

$$T(0) \le b = (b+c)\,0 + b$$

$$T(n) \le (b+c)\left\lceil \frac{n-1}{2} \right\rceil + (b+c)\left\lfloor \frac{n-1}{2} \right\rfloor + c = (b+c)\,(n-1) + c = (b+c)\,n + b$$

Splay Trees

- Splay trees are self adjusting binary search trees invented by Sleator and Tarjan around 1985.
- Basic principle of splaying: When accessing a node move it to the root by a sequence of rotations.
- Properties of splay trees - assume m $\geq$ n accesses in a tree of size n
  - No additional fields to ordinary binary search trees are needed.
  - Good time complexity: O(m log n), that is, O(log n) per access
    Worst case time complexity for an access could be bigger than log n.
  - Good locality: Suppose only k $\geq$ 2 distinct items are accessed in the m accessess. Time is O(m log k + n log n).
  - Insertion of n items in order into an empty tree is O(n) time.
  - Deletion of n items in order from a tree of size n is also O(n) time.

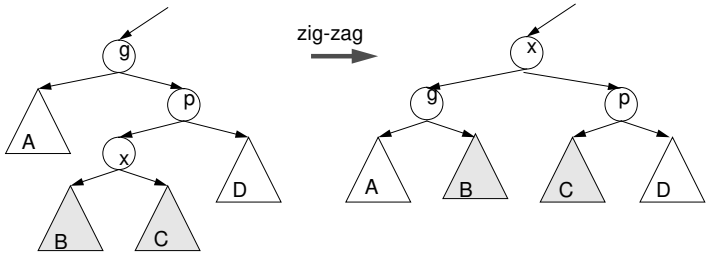
Find in a Splay Tree

- Find(x):

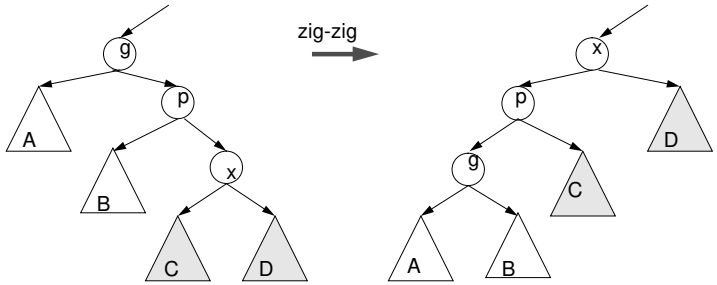
  1. do a normal search to find the node n such that n.data = x.

  2. Move n to the root by a series of zig-zag, zig-zig rotations, followed by a final zig if needed.

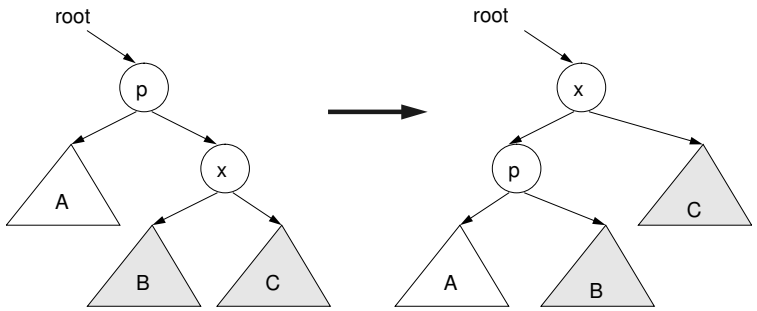
- Zig-zag is just our old friend double rotation.



Zig - Zig



Plane Ole Zig



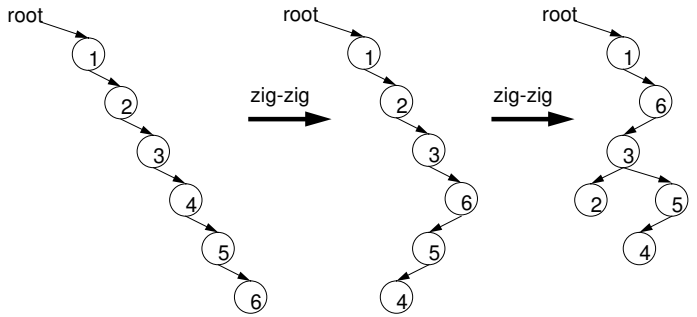
Why Does Splaying Help

- If a node n on the access path is at depth d before splaying then it is at depth approximately d/2 after splaying.

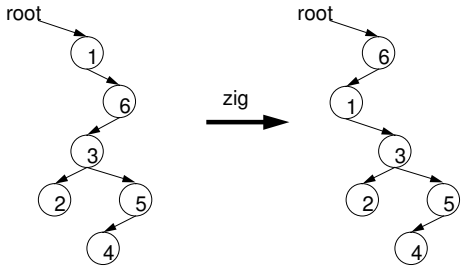
  Exceptions are the root, the child of the root and the node splayed.

- Overall, nodes which are below nodes on the access path tend to move closer to the root.

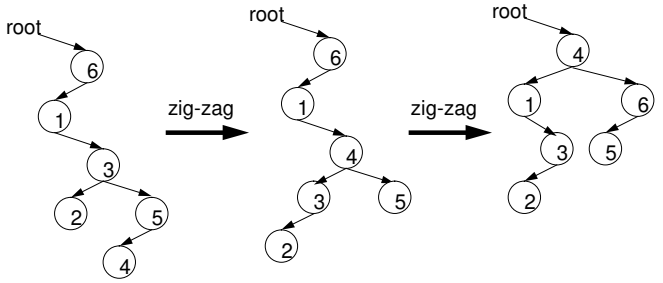
Example - Find 6



Find 6 Continued



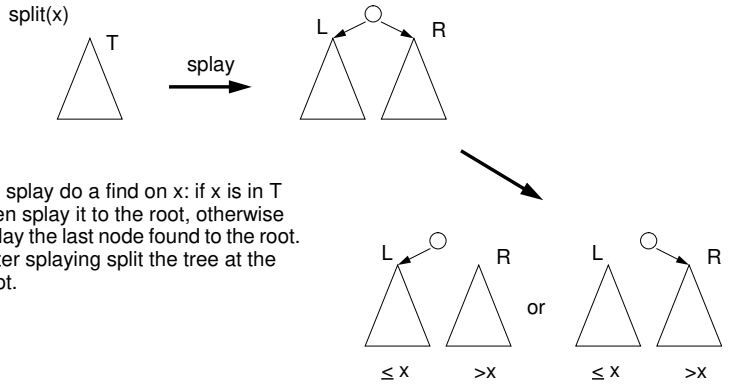
Find 4



Insertion and Deletion

- In order to define insert and delete in splay trees we introduce the operation so split and join.
- Split(x) returns two binary search trees L and R where L contains all nodes with keys $\leq$ x and R contains all nodes > x
- Assuming the keys in L are all smaller thant the keys in R, Join(L,R) returns a single tree with the same data L and R combined.

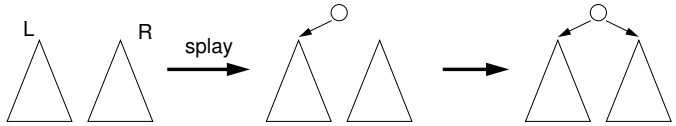
Split



To splay do a find on x: if x is in T then splay it to the root, otherwise splay the last node found to the root. After splaying split the tree at the root.

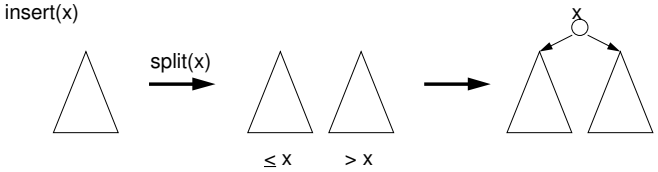
Joining Two Trees

- Join(L,R) is only done if the keys in L are smaller than the keys in R.

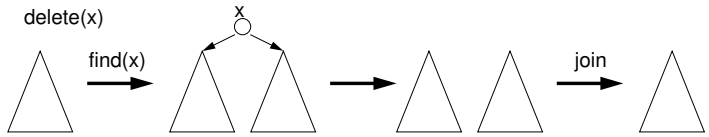


Splay on the maximum element in L then attach R as a right child.

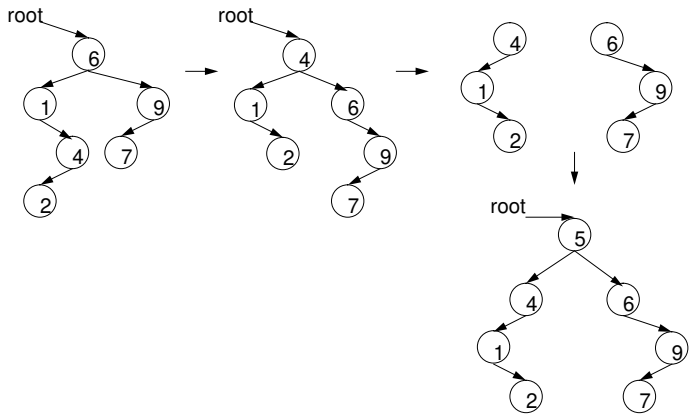
Insert



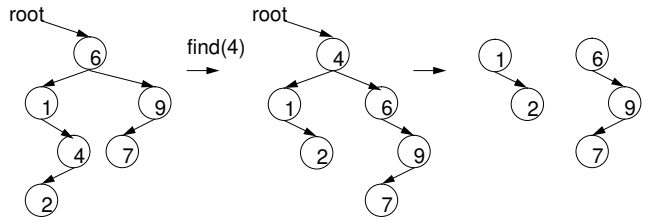
Delete



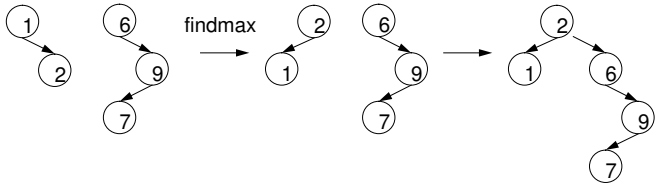
Example of Insert 5



Delete 4



Delete Continued



Splay Tree Notes

- Splaying can be done top down, which is probably the right way to do it.
- There are alternatives to the standard split-join implementation of insert and delete.

  For example, for insertion, just do a normal binary search tree insertion then splay the node to the root.
- Splay tree are currently the most popular binary search tree technique.

  1. It is relatively simple.

  2. No extra fields are needed.

  3. It has excellent temporal locality properties. Keys that are accessed often end up near the root.