CSE 326: Data Structures
Lecture #8
Balanced Dendrology

Bart Niswonger

Summer Quarter 2001

Today's Outline

- Clear up build tree analysis
- Deletion from BSTs
- Binary Search Trees

Analysis of BuildTree

- Worst case is $O(n^2)$

  $1 + 2 + 3 + \ldots + n = O(n^2)$

- Average case assuming all orderings equally likely is $O(n \log n)$
  - *not averaging over all binary trees, rather averaging over all input sequences (inserts)*
  - equivalently: average depth of a node is $\log n$
  - proof: see *Introduction to Algorithms*, Cormen, Leiserson, & Rivest

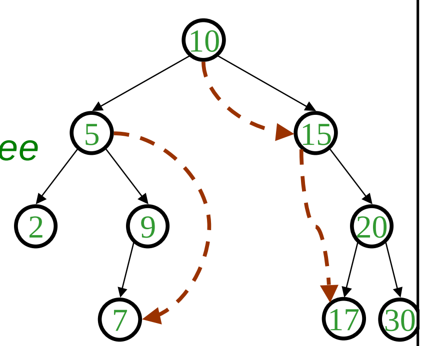
---

Digression

Finding the Successor

Find the next larger node in this node's subtree.

– *not next larger in entire tree*



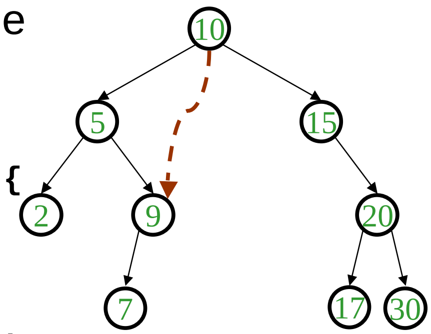

```
Node * succ(Node * root) {
  if (root->right == NULL)
    return NULL;
  else
    return min(root->right);
}
```

How many children can the successor of a node have?

Predecessor

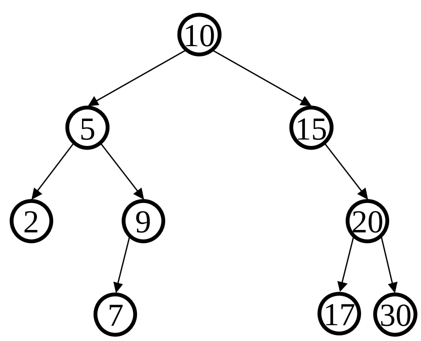
Digression

Find the next smaller node in this node's subtree.




```
Node * pred(Node * root) {
  if (root->left == NULL)
    return NULL;
  else
    return max(root->left);
}
```

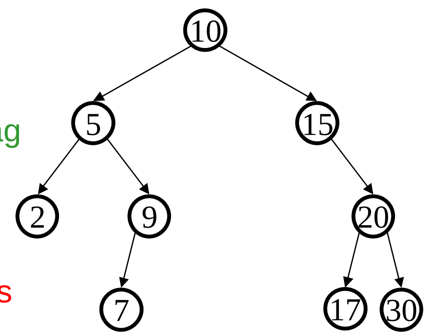
Deletion



Why might deletion be harder than insertion?

Lazy Deletion

- Instead of physically deleting nodes, just mark them as deleted
  - + Simpler
  - + some adds just flip deleted flag
  - + physical deletions done in batches
  - + extra memory for deleted flag
  - – many lazy deletions slow finds
  - – some operations may have to be modified (e.g., min and max)



Lazy Deletion

Delete(17)

Delete(15)

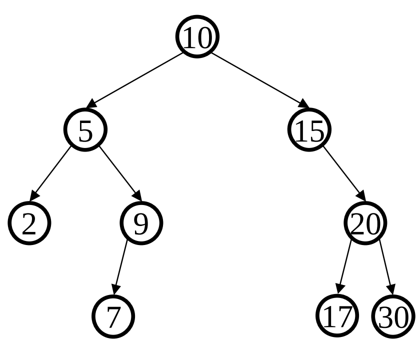
Delete(5)

Find(9)

Find(16)

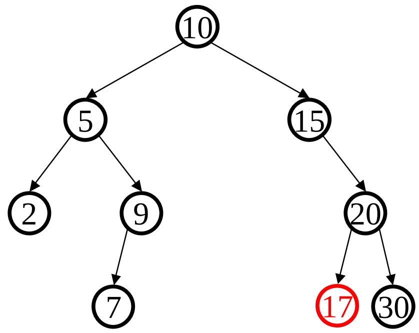
Insert(5)

Find(17)



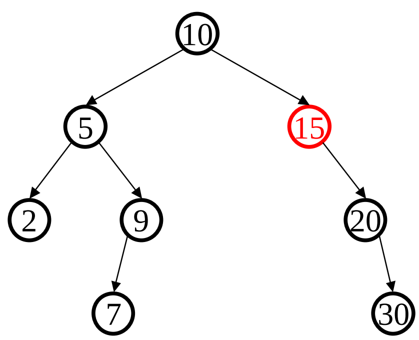
Deletion - Leaf Case

Delete(17)



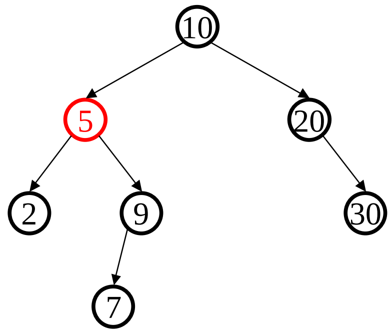
Deletion - One Child Case

Delete(15)



Deletion - Two Child Case

Delete(5)

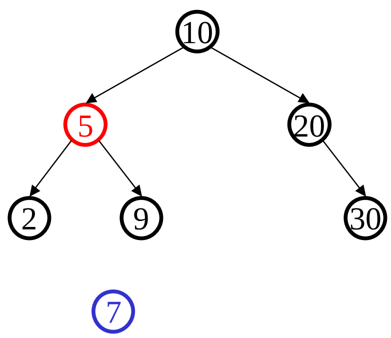


replace node with value guaranteed to be between the left and right subtrees: the successor

Could we have used the predecessor instead?

Deletion - Two Child Case

Delete(5)



always easy to delete the successor – always has either 0 or 1 children!

Delete Code

```
void delete(Comparable x, Node *& p) {
  Node * q;
  if (p != NULL) {
      if (p->key < x) delete(x, p->right);
      else if (p->key > x) delete(x, p->left);
      else { /* p->key == x */
         if (p->left == NULL) p = p->right;
         else if (p->right == NULL) p = p->left;
         else {
            q = successor(p);
            p->key = q->key;
            delete(q->key, p->right);
         }
      } } }
```

Dictionary Implementations

	unsorted array	sorted array	linked list	BST
insert	find + O(n)	O(n)	find + O(1)	O(Depth)
find	O(n)	O(log n)	O(n)	O(Depth)
delete	find + O(1)	O(n)	find + O(1)	O(Depth)

BST's looking good for shallow trees, *i.e.* the depth D is small (log n), otherwise as bad as a linked list!

Beauty is Only Θ(log n) Deep

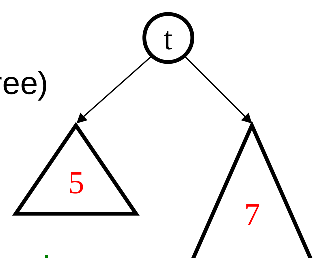
- Binary Search Trees are fast if they're shallow:
  - *e.g.:* perfectly complete
  - *e.g.:* perfectly complete except the "fringe" (leafs)
  - any other good cases?

What *matters* here?

Problems occur when one branch is much longer than the other!

Balance

- Balance:
  height(left subtree) - height(right subtree)



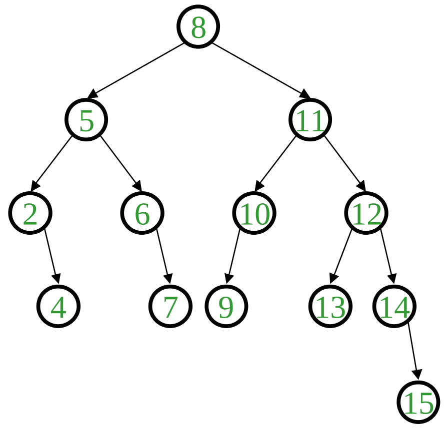
- zero everywhere ⇒ perfectly balanced
- small everywhere ⇒ balanced enough

Balance between -1 and 1 everywhere ⇒
maximum height of 1.44 log n

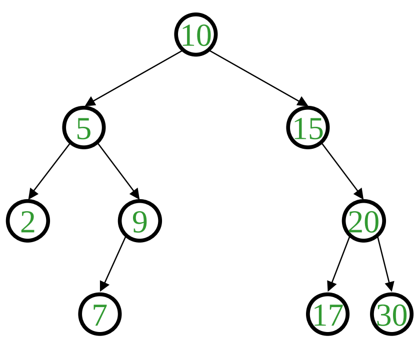
AVL Tree Dictionary Data Structure

- Binary search tree properties
  - binary tree property
  - search tree property
- Balance property
  - balance of every node is:

    $-1 \leq b \leq 1$
  - result:
    - depth is $\Theta(\log n)$

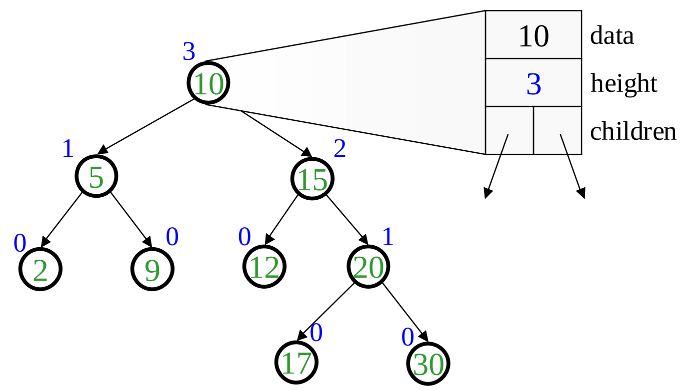


Testing the Balance Property

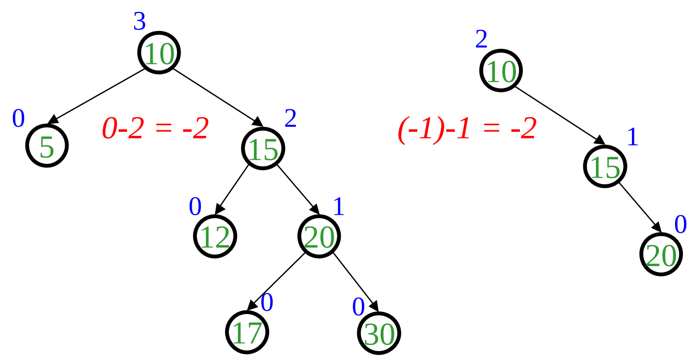


NULLs have height **-1**

An AVL Tree



Not AVL Trees

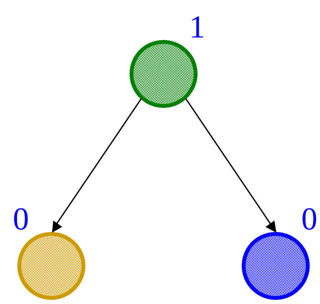


But, How Do We Stay Balanced?

- I need:
  – the smallest person in the class
  – the tallest person in the class
  – the averagest (?) person in the class

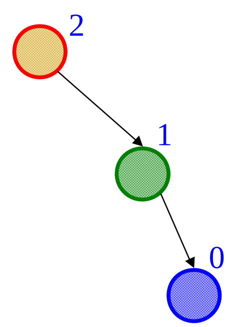
Beautiful Balance

Insert(middle)
Insert(small)
Insert(tall)

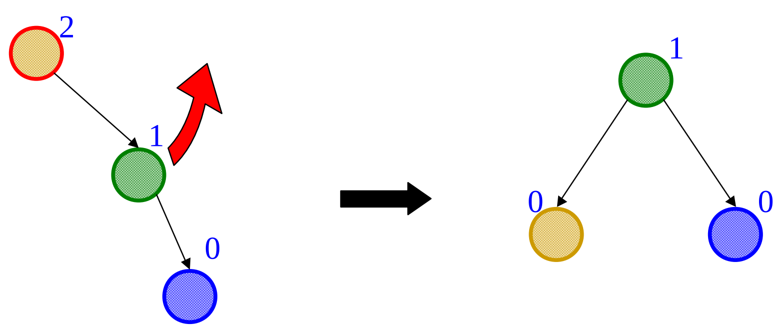


Bad Case #1

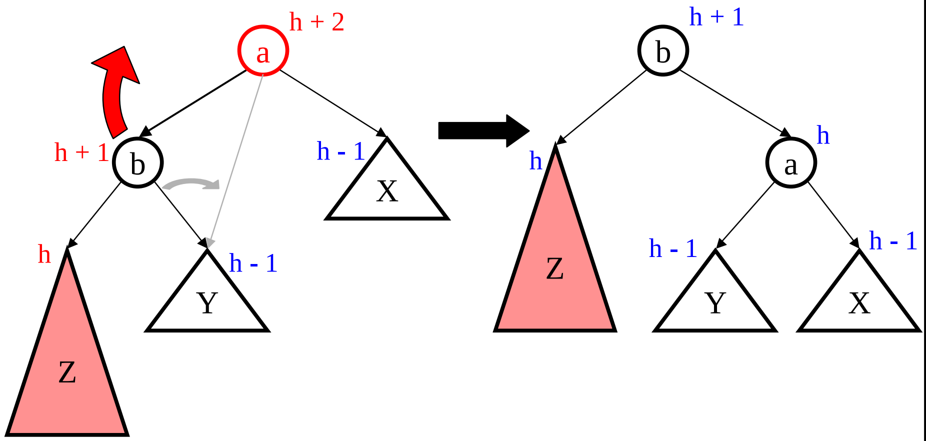
Insert(small)
Insert(middle)
Insert(tall)



Single Rotation



General Single Rotation



- Height of subtree same as it was before insert!
- Height of all ancestors unchanged.

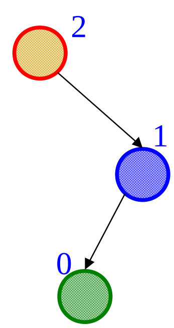
So?

Bad Case #2

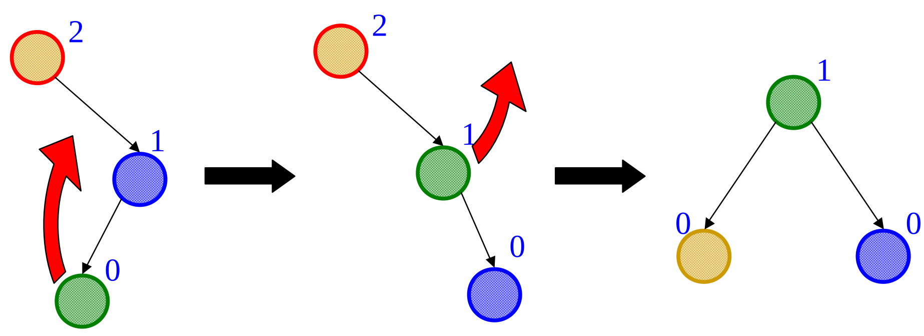
Insert(small)

Insert(tall)

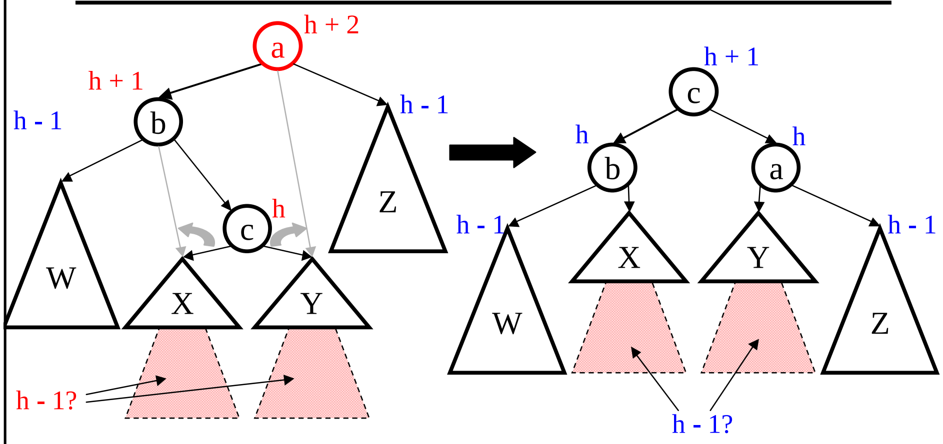
Insert(middle)



Double Rotation



General Double Rotation



- Height of subtree **still** the same as it was before insert!
- Height of all ancestors unchanged.

To Do

- Project II-A
- Read through section 4.6 in the book

Coming Up

- Project II – the complete version!
- More balancing acts
- A **Huge** Search Tree Data Structure