

Balanced Trees

- in an ideal world we would like to always ensure that our binary trees are perfectly balanced
- keeping a tree perfectly balanced after each deletion and insertion is computationally expensive

AVL Trees

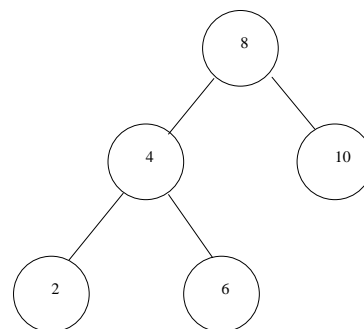
- Adelson-Velskii and Landis postulated a more relaxed definition for balancing a tree
 - a tree is balanced if and only if for every node the heights of its two subtrees differ by at most 1
- this provides a compromise between efficient tree access and the need to rebalance

AVL Trees

- consider insertion of a new int on the left branch, causing the left height to increase by one
- here we have three cases
 - $\text{height}(\text{left}) == \text{height}(\text{right})$: left and right become of unequal height, but the balance criterion is not violated
 - $\text{height}(\text{left}) < \text{height}(\text{right})$: left and right become of equal height, balance criterion was improved
 - $\text{height}(\text{left}) > \text{height}(\text{right})$: left and right differ by 2, balance criterion was violated, the tree must be balanced

AVL Trees

- consider adding 1, 3, 5, 7, 9 and 11



- which numbers require the tree to be rebalanced?

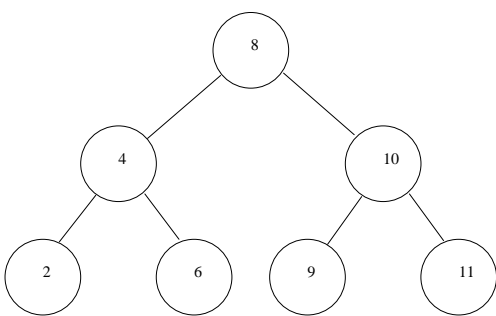
AVL Trees

- in summary we need to rebalance the tree if the left and right height differ by two or more

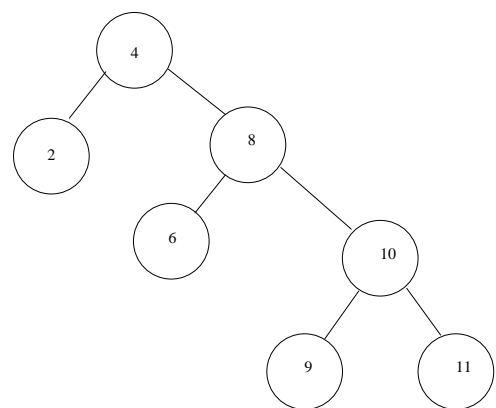
Rebalancing a tree

- rebalancing a tree is achieved through two operations which might be applied a number of times
 - rotate right
 - rotate left

Rotate right



Rotate right



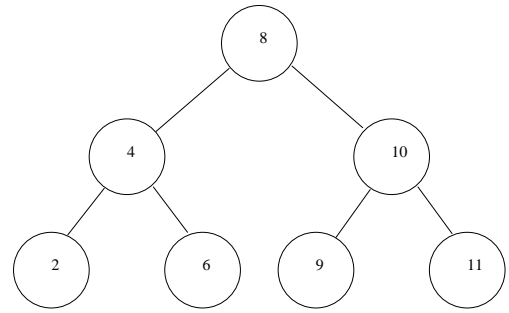
Rotate right axioms

```

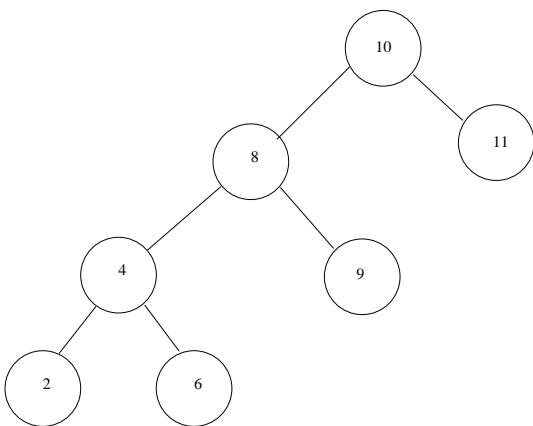
rotate_right (empty ())      -> empty ()
rotate_right (cons (i, l, r) -> cons (root (l),
                                     left (l),
                                     cons (i, right (l),

```

Rotate left



Rotate left



Rotate left axioms

```

rotate_left (empty ())      -> empty ()
rotate_left (cons (i, l, r) -> cons (root (r),
                                     cons (i, l, left (r),
                                     right (r))

```

rotate_right

examples/c++/trees/avl/int/tree.cc

```

/*
 * rotate_right - pre-condition : left branch is not empty.
 *                post-condition: a new tree is return which
 *                is the result of rotating the
 *                current tree.
 */

tree tree::rotate_right (void)
{
  if (is_empty ())
    return *this;
  else
    return cons (left ().root (),
                left ().left (),
                cons (root (),
                    left ().right (),
                    right ());
}

```

rotate_left

examples/c++/trees/avl/int/tree.cc

```

/*
 * rotate_left - pre-condition : right branch is not empty.
 *                post-condition: a new tree is return which
 *                is the result of rotating the
 *                current tree.
 */

tree tree::rotate_left (void)
{
  if (is_empty ())
    return *this;
  else
    return cons (right ().root (),
                cons (root (),
                    left (),
                    right ().left ()),
                right ().right ());
}

```

insert

examples/c++/trees/avl/int/tree.cc

```

tree tree::insert (int i)
{
  if (is_empty ())
    return cons (i, empty (), empty ());
  else
    return insert_non_empty (i).balance ();
}

```

balance

examples/c++/trees/avl/int/tree.cc

```

tree tree::balance (void)
{
  tree r = right ();
  tree l = left ();
  int d = l.height () - r.height ();
  tree t = *this;

  if ((d == 0) || (d == 1) || (d == -1))
    return t;
}

```

balance

■ `examples/c++/trees/avl/int/tree.cc`

```
/*  
 * now rotate  
 */  
  
if (d < 0)  
    return t.rotate_left ();  
else if (d > 0)  
    return t.rotate_right ();  
  
return cons (t.root (),  
            r.balance (),  
            l.balance ());  
}
```

Conclusion

- a functional approach can be useful in describing rotate right and left
- a functional approach allows a balance function to be implemented reasonably simply
- at the expense of lost memory, hence the requirement for a garbage collector