More Trees

Dr. Diana Hintea

Lecturer in Computer Science

diana.hintea@coventry.ac.uk

210CT Week 6

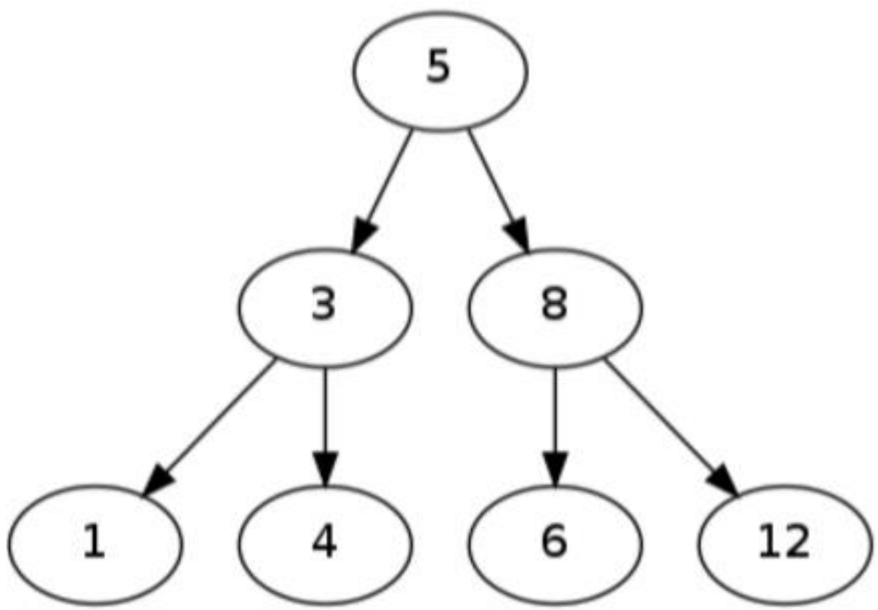
Learning Outcomes

1. A bit of recap
2. Additional BST Operations
3. Self Balancing Trees
4. AVL Trees
5. Rotation Operation

A bit of recap

Binary Search Trees

- Used to quickly find data.
- The key in each node must be greater than all keys stored in the left sub-tree, and smaller than all keys in right sub-tree, as below:



A bit of recap

Inserting a Node in A Binary Search Tree

```
BIN-TREE-INSERT(tree,item)

IF tree = Ø
   tree=Node(item)
ELSE
   IF tree.value > item
      IF tree.left = 0
         tree.left=Node(item)
      ELSE
         BIN-TREE-INSERT(tree.left,item)
   ELSE
      IF tree.right = 0
         tree.right=Node(item)
      ELSE
         BIN-TREE-INSERT(tree.right,item)
return r
```

A bit of recap

Finding an Item in a Binary Search Tree

```
BIN-TREE-FIND(tree,target)
IF tree = Ø
   r <- tree
WHILE r ≠ 0:
   IF r.value = trget
      RETURN r
   ELSE IF r.value > target
         r <- r.left
   ELSE
         r <- r.right
RETURN  0
```

A bit of recap

Recursively???

```
BIN-TREE-FIND(tree,target)
IF tree.value = target or tree = 0:
        RETURN  tree
ELIF target < tree.value:
        RETURN   BIN-TREE-FIND(tree.left, target)
ELSE
        RETURN   BIN-TREE-FIND(tree.right, target)
RETURN 0
```

A bit of recap

Binary Tree Traversal

- To traverse, or visit each node in a tree, there are a number of methods:
  1. **Pre order** - output item, then follow left tree, then right tree.
  2. **Post order** - follow the left child, follow the right child, output node value.
  3. **Breadth first** - start with the root and proceed in order of increasing depth/height (i.e root, second level items, third level items and so on).
  4. **In order** - for each node, display the left hand side, then the node itself, then the right. When displaying the left or right, follow the same instructions.

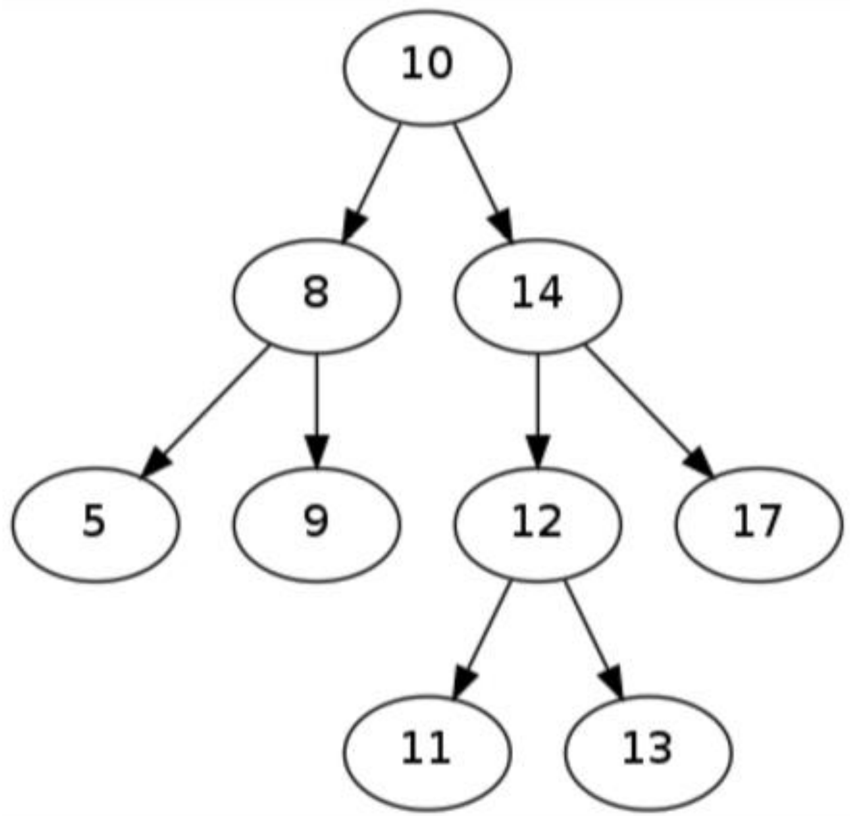
A bit of recap

Binary Search Tree Traversal

Pre order - output item, then follow left tree, then right tree.

10, 8, 5, 9, 14, 12, 11, 13, 17

NLR



A bit of recap

Binary Search Tree Traversal

Pre order - output item, then follow left tree, then right tree.

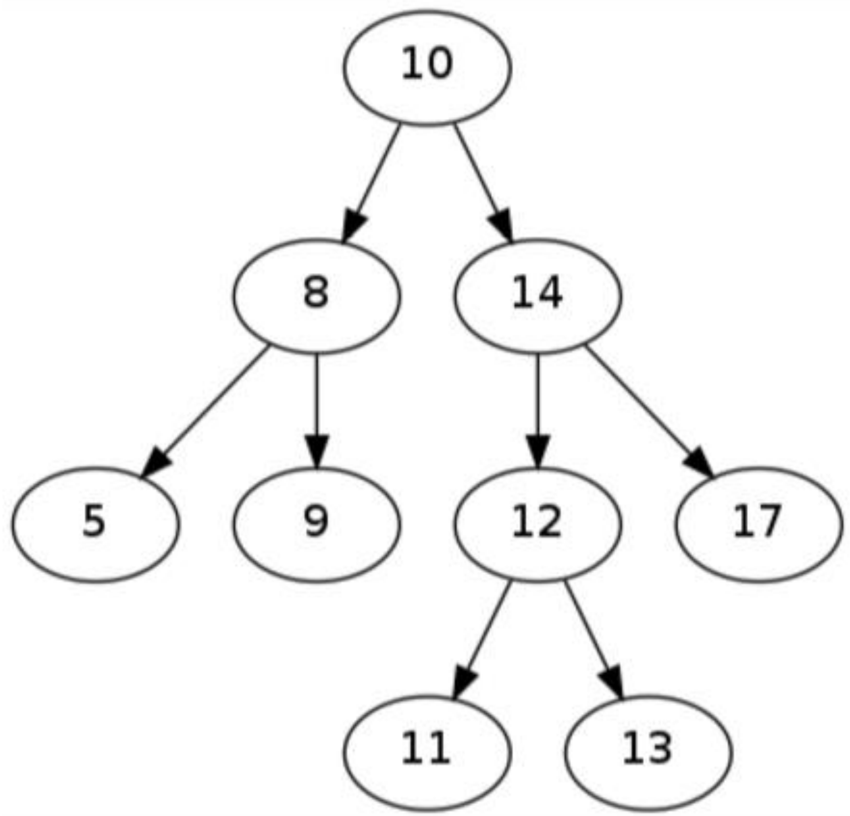
10, 8, 5, 9, 14, 12, 11, 13, 17

NLR

```
PREORDER(tree):
    print tree.value
    if tree.left! ≠0:
        PREORDER (tree.left)
    if tree.right≠0:
        PREORDER (tree.right)
```

A bit of recap

Binary Search Tree Traversal



In order - for each node, display the left hand side, then the node itself, then the right. When displaying the left or right, follow the same instruction.

5, 8, 9, 10, 11, 12, 13, 14, 17

LNR

A bit of recap

Binary Search Tree Traversal

In order - for each node, display the left hand side, then the node itself, then the right. When displaying the left or right, follow the same instruction.

5, 8, 9, 10, 11, 12, 13, 14, 17

LNR

```
INORDER(tree):
  if tree.left! ≠0:
    INORDER(tree.left)
  print tree.value
  if tree.right≠0:
    INORDER(tree.right)
```

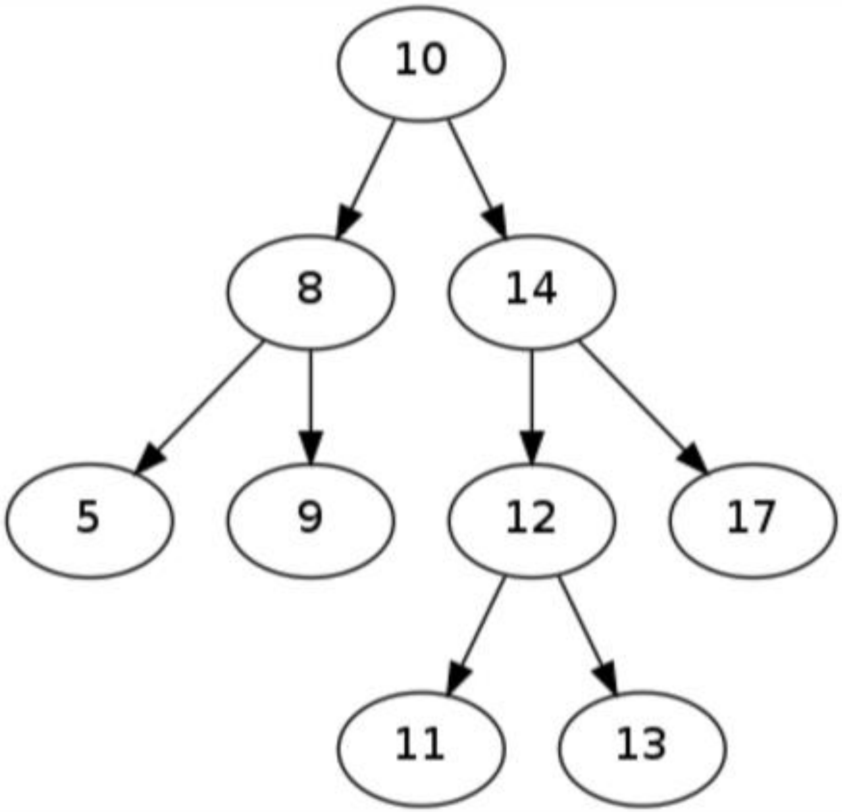
A bit of recap

Binary Search Tree Traversal

Post order - follow the left child, follow the right child, output node value.

5, 9, 8, 11, 13, 12, 17, 14, 10

LRN



A bit of recap

Binary Search Tree Traversal

Post order - follow the left child, follow the right child, output node value.

5, 9, 8, 11, 13, 12, 17, 14, 10

LRN

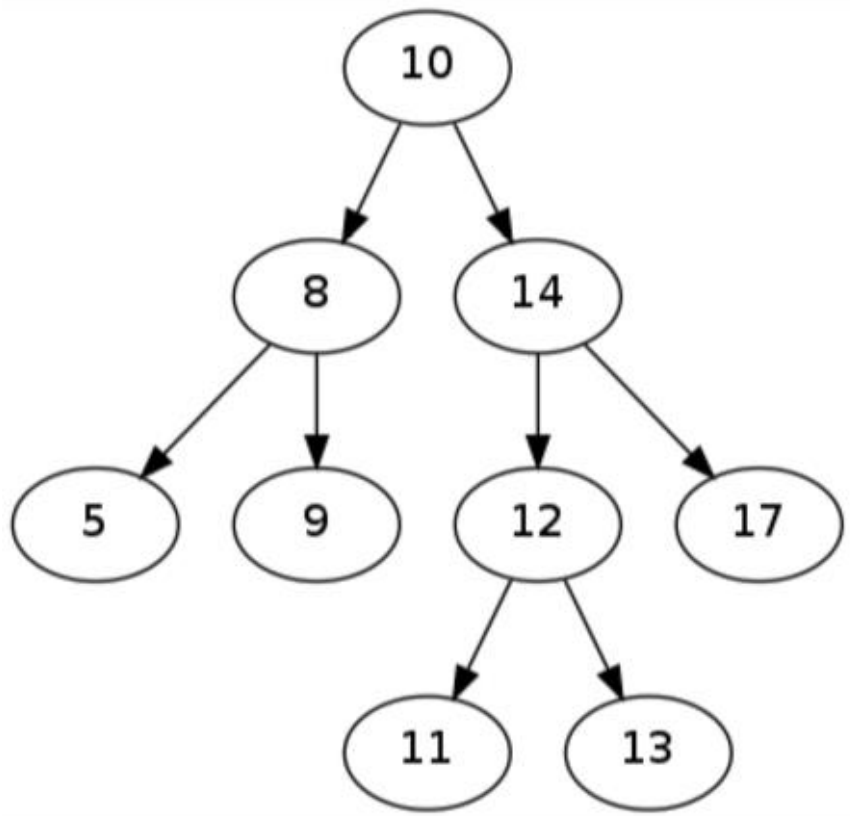
```
POSTORDER(tree):
   if tree.left! ≠0:
      POSTORDER (tree.left)
   if tree.right≠0:
      POSTORDER (tree.right)
   print tree.value
```

A bit of recap

Binary Search Tree Traversal

Breadth first - start with the root and proceed in order of increasing depth/height (i.e root, second level items, third level items and so on).

10, 8, 14, 5, 9, 12, 17, 11, 13



A bit of recap

Removing a Node

- More complex than search or insertion as the process depends on the number of child nodes.
- If the node to be deleted is a leaf i.e. no children, this is easy, just delete the node.
- If the node has only one child we set the child's parent variable to the node above the node to be deleted.
- If the node has two children this is more complex.

A bit of recap

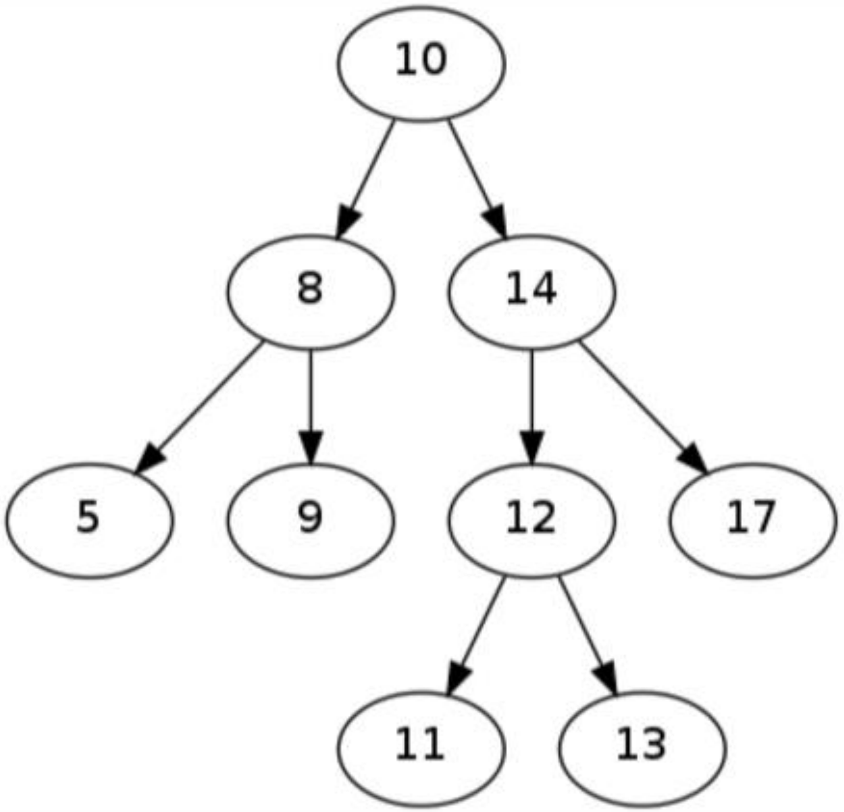
Removing a Node

- Find either the minimum value node from the right subtree, or maximum value node from the left subtree.
- Replace the deleted node with that value.
- Remove the leaf node whose value has been copied.

A bit of recap

Node Removal Pseudocode

```
COUNT_CHILDREN(n):
    count <- 0
    if n.left!=0:
        count <- count + 1
    if n.right!=0:
        count <- count 1
    return count
```



A bit of recap

REMOVE_NODE(value):

Find the node corresponding to value using BIN_TREE_FIND

Find the node's number of children using COUNT_CHILDREN

Keep track of the node's parent (integrate in node constructor)

If the node has no children then remove it (make the parent connections equal to 0)

Else if the node has 1 child then interchange the child node with the parent node and remove the current child node.

Else

Find max value from left subtree or min value from right subtree

Swap the node value with the successor node value

Remove the leaf node with the duplicate value

A bit of recap

Binary Search Trees

	Average case	Worst case
Search	O(log n)	O(n)
Insert	O(log n)	O(n)
Delete	O(log n)	O(n)

Additional BST Operations

Comparing two trees

Additional BST Operations

Comparing two trees

```
COMPARE_TREES(tree1, tree2)
    if tree1 = 0 or tree2 = 0 return  False
    if tree1. value ≠ tree2.value return  False
    result = True
    if tree1.left = 0:
        if tree2.left ≠0 return False
    else result = COMPARE_TREES (tree1.left, tree2.left)
    if tree1.right = 0:
        if tree2.right ≠ 0 return False
    else result = COMPARE_TREES (tree1.right, tree2.right)
    return result
```

Additional BST Operations

Comparing two trees – function within class

```
COMPARE_TREES(tree)
    if tree = 0 return  False
    if self. value ≠ tree.value return  False
    result = True
    if self.left = 0:
        if tree.left ≠0 return False
    else result = COMPARE_TREES (tree.left)
    if self.right = 0:
        if tree.right ≠ 0 return False
    else result = COMPARE_TREES (tree.right)
    return result
```

Tree Sort

- A tree sort is a sort algorithm.
- Builds a binary search tree from the values to be sorted, and then traverses the tree so that the keys come out in sorted order.
- How exactly?
- What do I mean by building the tree?
- What kind of traversal?

Additional BST Operations

Tree Sort

```
INSERT(Tree, item)
INORDER(Tree)

TREE_SORT(A)
FOR each item i IN A
    INSERT(Tree, i)
INORDER(Tree)
```

Additional BST Operations

Homework 1

- Implement TREE_SORT algorithm in a language of your choice, but make sure either the INSERT or the INORDER function is implemented iteratively.

Additional BST Operations

Homework 2

- For C++ people:
  - Create a class called Comparator that has three functions:

    isEqual(N)

    isGreater(N)

    isLessThan(N)

  In each case, N is a comparator object. All functions should return 0.

Additional BST Operations

Homework 2

- Now create a class called Hyperbole, which extends the Comparator class. It should have a data item which is a string that may contain one of the following:

  Massive, Super, Mega, Ultra, Ultimate
- The isEqual, isGreater and isLessThan functions should be reimplemented to allow Hyperbole objects to be compared
- Their order in the list above shows their relative value
- For example: Mega is more than Super, but less than Ultra.

Additional BST Operations

Homework 2

- For Python:
  - As above, but you don't need the superclass. As long as the Hyperbole class implements the functions, it will work in all the places the C++ equivalent does.

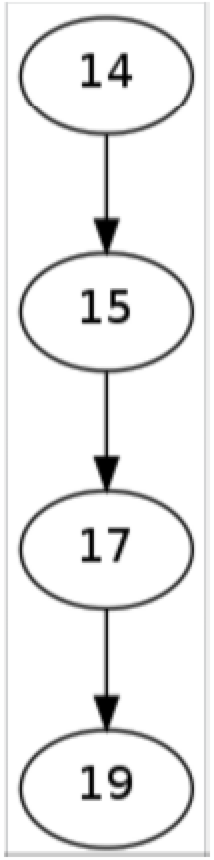
BREAK!!!

Exam

Self Balancing Trees

Insertion order effect

- As we discussed last week, the order that elements are inserted into the tree have an effect on the structure of the tree.
- Elements inserted in order will unbalance the tree, e.g. inserting 14, 15, 17 and 19 in order will create the following tree

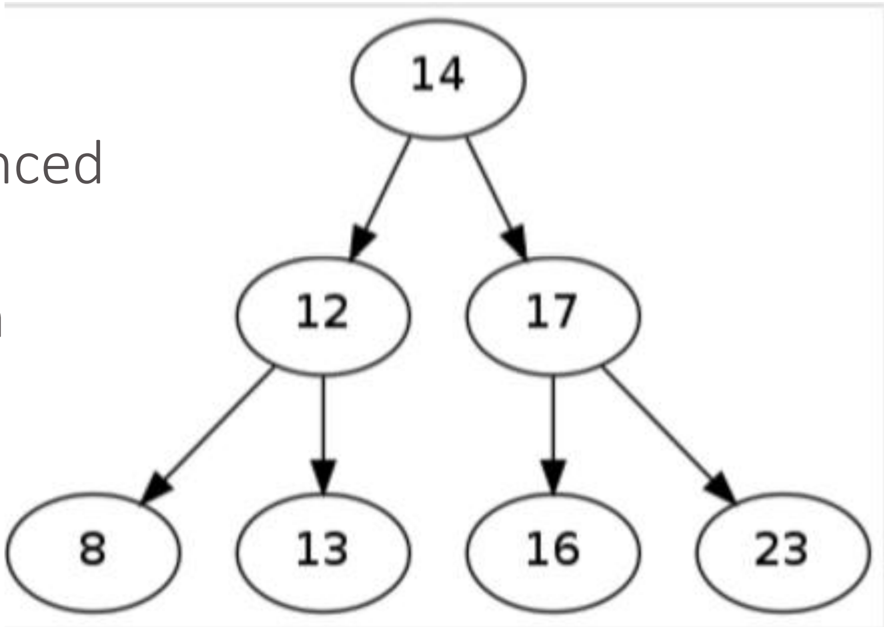


Imbalanced trees

- The tree in the previous example is called **imbalanced** because it's left subtree has a very different size to it's right subtree.
- The left subtree has a depth of 0, while the right subtree has a depth of 3.
- Searching this tree is much less efficient than searching a tree where the branches are equally deep.
- This has accidentally become a linked list and searching becomes a linear search operation with O(N)..
- This is the worst case scenario and the furthest from our ideal O(logn).

A well balanced tree

- A tree is considered well balanced when depth of its left subtree differs no more than one from the depth of its left subtree.



AVL trees

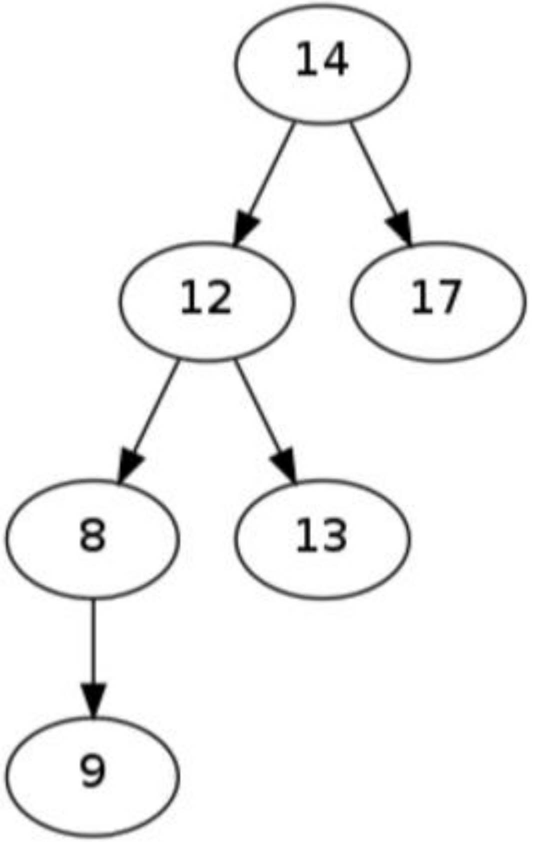
- Named after Adelson-Velskii and Landis, the inventors.
- The AVL tree automatically balances itself whenever a node is added or removed.
- The balancing process takes place in logarithmic time.
- Balancing process tries to ensure that the left and right subtrees do not differ by more than 1.
- Each node must keep track of the depth of each of its subtrees.
- Each node is an AVL tree itself, so upon insertion of a new item, the tree can be traversed in reverse to check the AVL constraint holds

Rotation

- The way that AVL trees are balanced is a process called rotation.
- Rotation works by moving the smaller subtree down the tree and moving the larger subtree up the tree.
- Rotation maintains the order of the elements in the tree.
- Rotation can be to the left or the right, one is the mirror of the other.

Rotation example

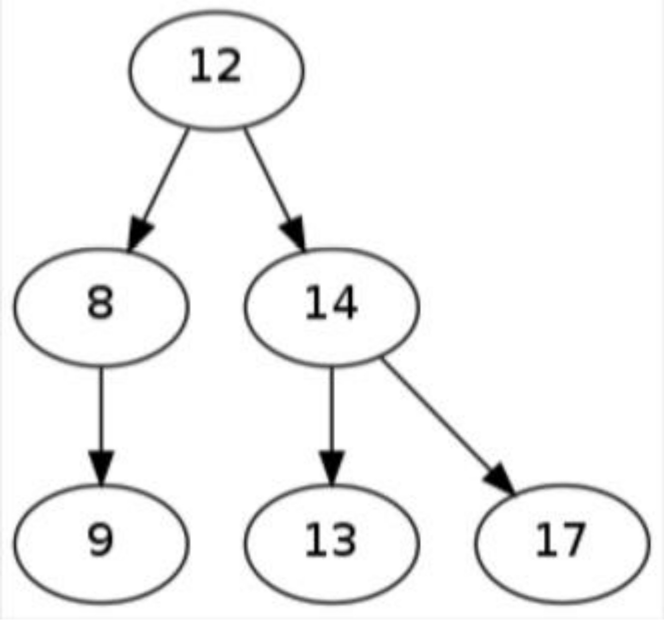
- The node ‘14’ has a left subtree greater than its right. We fix this by rotating to the right.
- To rotate:
  - Make the smaller subtree (13) the left child of the parent (14)
  - Make the previous parent (14) the right child of the new parent (12)



Rotation Operation

Rotation example

- This now results in a well balanced tree.



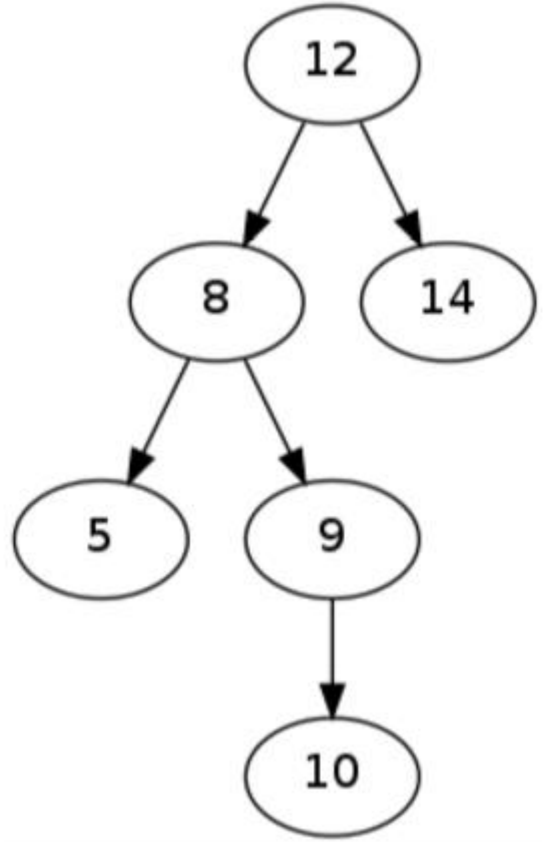
The left-right case

- There is also a case when rebalancing the tree where one rotation is insufficient.
- In the previous case the left subtree of the left tree is heavy, so it’s called the left-left case.
- If the right subtree of the left tree is heavy this is called the left-right case.
- In this case, 1 rotation is not sufficient.

Rotation Operation

Example

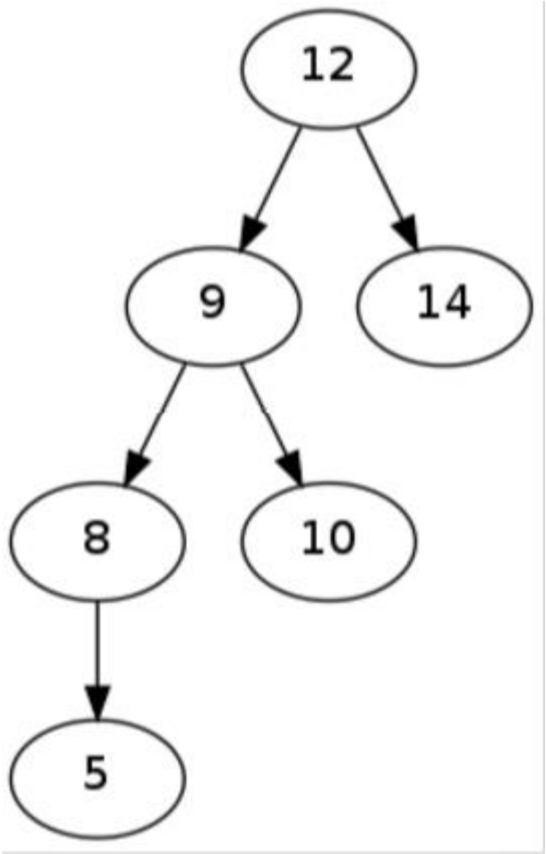
- In this case, the left subtree is unbalanced (8 down), but it is unbalanced on its own right.
- To fix this we rotate the right subtree to the left.



Rotation Operation

Example

- This has the effect of giving us the case of the left-left as above.
- We can now rotate the left subtree to the right.



Rotation Operation

Scenarios

1) **left – left rotation:** Balance of node is 2 and the balance of the left child is > 0 , then single **right** rotation.
2) **left – right rotation:** Balance of node is 2 and the balance of the left child is < 0 , then **left** rotation followed by **right** rotation.
3) **right – left rotation:** Balance of node is -2 and the balance of the left child is > 0 , then **right** rotation followed by **left** rotation.
4) **right – right rotation:** Balance of node is -2 and the balance of the left child is < 0 , then single ‘left’ rotation.

Rotation Operation

Example

[AVLTree]

Rotation Operation

Pseudocode

```
IF tree is right heavy:
    IF tree's right subtree is left heavy:
        Perform Double Left rotation
    ELSE
        Perform Single Left rotation
ELSE IF tree is left heavy:
    IF tree's left subtree is right heavy
        Perform Double Right rotation
    ELSE
        Perform Single Right rotation
```

Rotation Operation

- What operations would have to ensure at the end that the balancing property is preserved?

Rotation Operation

- Inserting an element in the AVL tree.
- Removing an element from the AVL tree.

- If necessary, perform rotation.

Pseudocode for checking balance

```
CHECK_BALANCE(tree):
#Returns 0 for balanced tree, -1 for left imbalance by 1, +1
for right imbalance by 1, -2 for left imbalance by 2, etc.

lsize <- 0
rsize <- 0
IF tree.left ≠ 0:
        lsize <- tree.left.depth
IF tree.right ≠ 0:
        rsize <- tree.right.depth
RETURN  rsize-lsize
```

Pseudocode for rotation

```
ROTATE_RIGHT(tree)
root <- tree.left
tree.left <- root.right
root.right <- tree
IF root.right ≠ 0
    root.right.fixDepth()
IF root.left ≠ 0
    root.left.fixDepth()
root.fixDepth()
RETURN  root
```

```
FIX_DEPTH(tree) #After child nodes have been moved and their depths fixed, self is used to fix the depth of node
IF tree.left = 0
    lsize <- 0
ELSE  tree.left.depth
IF tree.right = 0
    rsize <- 0
ELSE  tree.right.depth
tree.depth=1+max(lsize, rsize)
```

Rotation Operation

Pseudocode for insertion

```
INSERT_FULL(tree, n)
IF n.data < tree.data:
    IF tree.left ≠  0:
        tree.left <- INSERT_FULL (tree.left, n)
    ELSE:
        tree.left <- n
ELSE:
    IF tree.right ≠ 0:
        tree.right <- INSERT_FULL (tree.right, n)
    ELSE:
         tree.right = n
```

Rotation Operation

```
IF  CHECK_BALANCE(tree)<-1:        #left-hand too heavy
    IF  CHECK_BALANCE(tree.left)<=0:      #single rotation to the right
        root <- ROTATE_RIGHT(tree)
    ELSE:     #Double rotation
        tree.left <- ROTATE_LEFT(tree.left)
        root <- ROTATE_RIGHT(tree)
ELSE IF CHECK_BALANCE(tree)>1:       #right-hand too heavy
    IF CHECK_BALANCE(tree.right)>=0:         #single rotation to the left
        root <- ROTATE_LEFT(tree)
    ELSE:     #Double rotation
        tree.right <- ROTATE_RIGHT(tree.right)
        root <- ROTATE_LEFT(tree)
root.fixDepth()
```

Rotation Operation

[AVL Tree C++]

Rotation Operation

[AVL Tree Python]

A bit of recap

AVL Trees

	Average case	Worst case
Search	O(log n)	?
Insert	O(log n)	?
Delete	O(log n)	?

A bit of recap

AVL Trees

	Average case	Worst case
Search	O(log n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)

Actual homework

- Research other balanced trees, such as the red/black tree and discuss the similarities and differences between them and AVL trees.
- Be sure to think about which applications each might be best suited for.

Rotation Operation

Alternative homework

- Using the code given in the lecture as a starting point, implement node deletion.
- Show pseudocode and implementation.