Trees
Traversals / Removal / Balancing
Traversals / Printing
public void printPostOrder() {
    printPostOrderUtil(root, 0);
    System.out.println();
}

private void printPostOrderUtil(Node treePart, int depth) {
    if (treePart == null) return;
    System.out.print("["+treePart.payload+","+depth+"]");
    printPostOrderUtil(treePart.left, depth+1);
    printPostOrderUtil(treePart.right, depth+1);
}
Breadth First traversal

0 [7]
1 [4 12]
2 [2 6 9 19]
3 [3 5 8 11 15 20]
Remove

• boolean remove(E element);

• returns true if element existed and was removed and false otherwise

• Cases
  ▫ element not in tree
  ▫ element is a leaf
  ▫ element has one child
  ▫ element has two children
Leaf

- Just delete
One child

- Replace with child – skip over like in linked list
Two Children

- Replace with in-order predecessor or in-order successor
  - in-order predecessor
    - rightmost child in left subtree
    - max-value child in left subtree
  - in-order successor
    - leftmost child in right subtree
    - min-value child in right subtree
2 child replacement

Suc, pred for 10, 15, 19?
remove pseudocode

boolean remove(element)
    return removeUtil(element, root, null);

boolean removeUtil(element, node, parent)
    if (node==null) return false;
    if (node.payload>element)
        removeUtil(element, node.left, node);
    else if (node.payload<element)
        removeUtil(element, node.right, node);
    else
remove pseudocode 2

// found the node to delete
if (node.right == null && node.left == null)
    // at a leaf
    parent.remove(node)
    return true
if (node.right == null)
    // one descendent on left
    attach node.left to parent
    return true;
if (node.left == null)
    // one descendent on right
    attach node.right to parent
    return true;
remove pseudocode 3

// two children
successorNode = inorderSuccessor(node.right)
node.payload = successorNode.payload
removeUtil(successorNode.payload, node.right, node);
return true;
Balanced Search Trees

• A variety of algorithms augment a standard BST with occasional operations to reshape, reduce height and maintain balance.

• General approach: Rotation — moves a child to be above its parent,
  • ideally $O(1)$
  • certainly $O(lgn)$
AVL Trees

• Height-balance property
  □ For every internal node, the avlHeight of the two children differ by at most 1
    □ avlHeight = max distance from null endpoint

• Any binary tree satisfying the height-balance property is an AVL tree

• A height-balanced tree has height $O(\lg n)$
  • max height is provably $1.44^*\lg(n)$
AVL Tree Example
Insertion

- Maintain with each node the avlHeight.
- On insertion, first recur down through tree to insert.
- Then as you unwind recursion, update the avlHeight of each node.
- If height changes, check the height of other child
  - if not in balance then fix
private class Node {
    Comparable<E> element;
    int avlHight;
    Node right;
    Node left;

    public Node(Comparable<E> e) {
        avlHight = 1;
        element = e;
        right = null;
        left = null;
    }
}
More insertion (pseudo)code

```java
int insertUtil(node, element):
    if element==node.payload
        return -1;

    avlD=2; //!!
    if node.payload > element:
        if node.left==null
            node.left=new Node(payload)
        else
            avlD = 1+insertUtil(node.left,element);
    else
        // same but for right

    node.avlHeight = greater of avlD and
    node.avlHeight

    return node.avlHeight
```
Fixing height imbalances
Rotation!!

• Two types of rotation
• Single
  • left subtree of left node causes imbalance
  • right subtree of right node causes imbalance
• Double
  • right subtree of left node causes imbalance
  • left subtree of right node causes imbalance
  • The first rotation of a double puts the tree into position for a single rotation!
Single Rotation

Rotate across parent at the lowest imbalance

right-right => counter-clockwise rotation

left-left => cw rotation

M₄
R₂
P₃
E₁
S₁
Double Rotation

First rotate across the child imbalance, This shifts from R-L to R-R

Then do a single rotation, on the parent

r,l => cw around child, then ccw around parent
Lab Part 1

- Given the following data show the tree after each operation, while keeping the tree balanced using AVL
- for deletions, always delete using inorder predecessor.
- iXXX == insert XXX into tree
- dXXX == delete XXX from tree

- i1024, i512, i256, i128, i64, i32, i16, i750, i875, d128, d32

- solutions:
  - https://cs.brynmawr.edu/cs151/L22/balance1.jpg
  - https://cs.brynmawr.edu/cs151/L22/balance2.jpg
Lab  Part 2

Show the tree after each insertion / deletion
Before and after AVL rotation,
Send photo to gtowell151@cs.brynmawr.edu

<table>
<thead>
<tr>
<th>insert</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>200</td>
</tr>
<tr>
<td>insert</td>
<td>300</td>
</tr>
<tr>
<td>insert</td>
<td>400</td>
</tr>
<tr>
<td>insert</td>
<td>500</td>
</tr>
<tr>
<td>insert</td>
<td>600</td>
</tr>
<tr>
<td>insert</td>
<td>700</td>
</tr>
<tr>
<td>insert</td>
<td>800</td>
</tr>
<tr>
<td>insert</td>
<td>900</td>
</tr>
<tr>
<td>insert</td>
<td>750</td>
</tr>
<tr>
<td>insert</td>
<td>1000</td>
</tr>
<tr>
<td>insert</td>
<td>850</td>
</tr>
<tr>
<td>delete</td>
<td>400</td>
</tr>
<tr>
<td>delete</td>
<td>300</td>
</tr>
<tr>
<td>delete</td>
<td>200</td>
</tr>
<tr>
<td>delete</td>
<td>700</td>
</tr>
<tr>
<td>delete</td>
<td>500</td>
</tr>
</tbody>
</table>