CS206

Trees, continued
Shallow & deep copy (revisited)

```java
public class DeepShallow {
    private class Node {
        public int payload;
        public Node(int p) {
            this.payload = p;
        }
    }

    public void shde() {
        ArrayList<Node> arrOrig = new ArrayList<>();
        for (int i=0; i<10; i++) {
            arrOrig.add(new Node(i));
        }

        // Pointer Copy
        ArrayList arrpc = arrOrig;

        // Shallow Copy
        ArrayList<Node> arrShallow = new ArrayList<>();
        Collections.copy(arrShallow, arrOrig);

        // Deep Copy
        ArrayList<Node> arrDeep = new ArrayList<>();
        for (Node n : arrOrig) {
            arrDeep.add(new Node(n.payload));
        }

        public static void main(String[] args) {
            new DeepShallow().shde();
        }
    }
}
```
Again, use a recursive helper method

```java
@Override
public int maxDepth()
{
  return iMaxDepth(root, 1);
}
```

```java
int iMaxDepth(Node n, int depth)
{
  ...}
```

Recall: Depth == node property — distance from root
Height == tree property — max depth

From Tuesday

```java
private class Node
{
  Comparable<E> payload;
  Node right;
  Node left;

  public String toString()
  {
    return payload.toString();
  }
}
```

Terms

```
   7
 /  \
4    12
   /   \
2    9     19
  /     /    \
3    8    11    15
 /     /     \
5    11  15

```
Binary Tree Traversals

- Traversal visits all nodes in a tree in some order
- Inorder: left subtree, current, right subtree
- Preorder: current, left subtree, right subtree
- Postorder: left subtree, right subtree, current
Traversals
toString inorder

Yet another recursive helper method

```java
public String toString()
{
    return inorderString(root, 0);
}

private String inorderString(Node n, int depth)
```
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
• 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
Replace with Predecessor
Replace with Successor

Diagram:

Delete (20)
in-order successor

Tree before deletion:

```
  15
 /   \
10    20
|     |
8 12 18
```

Tree after deletion:

```
  15
 /   \
10    30
|     |
8 12 18 40
```

Successor:

```
  15
 /   \
10    30
|     |
8 12 18 40
```
Pseudo code

removeRec(sRoot, key):
    if sRoot == null:
        return null
    if sRoot.key > key:
        sRoot.left = removeRec(sRoot.left, key)
        return sRoot
    else if sRoot.key < key:
        sRoot.right = removeRec(sRoot.right, key)
        return sRoot
    else  // found the one to delete!!!!
        if sRoot.left == null:
            return sRoot.right
        else if sRoot.right == null:
            return sRoot.left
        else
            // either two children OR no children
            sRoot.key = minKey(sRoot.right)  //change value!
            sRoot.right = removeRec(sRoot.right, sRoot.key)
            return sRoot

minKey(sRoot):
    if sRoot.left == null:
        return sRoot.key
    else
        return minKey(sRoot.left)

10/31: Question in class about coverage of the no links case. I reviewed and this is correct. I added two return statements to the version shown in class
public boolean remove(E element) {
    int oSize=size;
    root = iRemoveRec(root, element);
    return oSize!=size;
}

private Comparable<E> iMinKey(Node sRoot) {
    if (sRoot.left==null)
        return sRoot.payload;
    else
        return iMinKey(sRoot.left);
}

private Node iRemoveRec(Node sRoot, Comparable<E> element) {
    if (sRoot==null) return null;
    ....
}
Array-based Binary Tree

- Number nodes level-by-level, left-to-right
- $f(root) = 0$
- $f(l) = 2f(p) + 1$
- $f(r) = 2f(p) + 2$
- Numbering is based on all positions, not just occupied positions
Level-numbering

• Number nodes level-by-level, left-to-right

- $f(root) = 0$
- $f(l) = 2f(p) + 1$
- $f(r) = 2f(p) + 2$
Array-based Binary Tree

- The numbering can then be used as indices for storing the nodes directly in an array.