CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2011
Doc 19 Assignment 3 comments
May 5, 2011

instanceOf hards choices

```
if (root instanceof NullHeapNode) {
        root = MaxHeapNode.createHeapNode(intValue);
        return true;
    }
    return root.add(MaxHeapNode.createHeapNode(intValue));
```

```
if (root.isNullNode()) {
        root = MaxHeapNode.createHeapNode(intValue);
        return true;
    }
    return root.add(MaxHeapNode.createHeapNode(intValue));
```

```
return root.add(MaxHeapNode.createHeapNode(intValue));
```

Helper methods

```
private void generateArrayList(ArrayList<HeapNode> arrayList, HeapNode heapNode) {
    arrayList.add(heapNode);
    if (heapNode.getLeftNode() != null)
        generateArrayList(arrayList, heapNode.getLeftNode());

    if (heapNode.getRightNode() != null)
        generateArrayList(arrayList, heapNode.getRightNode());
}
```

```
private void generateArrayList(ArrayList<HeapNode> arrayList) {
    arrayList.add(this);
    if (this.getLeftNode() != null)
        getLeftNode().generateArrayList(arrayList);

    if (this.getRightNode() != null)
        getRightNode().generateArrayList(arrayList);
}
```

Why repeat logic?

```
private void updateTreeHeight(){
      treeHeight = (int)((Math.log(numNodes)/Math.log(2)));
}

private int findTreeHeight(int numberOfNodes){
      return (int)((Math.log(numberOfNodes)/Math.log(2)));
}
```

```
private void updateTreeHeight(){
      treeHeight = findTreeHeight(numNodes);
}

private int findTreeHeight(int numberOfNodes){
      return (int)((Math.log(numberOfNodes)/Math.log(2)));
}
```

```
private Vector<Node> levelOrder; // these are parameters to methods not fields -2
private Vector<Integer> preOrder;
private Vector<Integer> oddPreOrder;
private int index;

    //rew call this twice and you get wrong answer -1
    public Vector<Integer> printPreorder(){
          this.printPreorder(head);
    return preOrder;
    }

    //rew Helper method -2
    //rew using field instead of parameter -1
    private void printPreorder(Node head){
          if ( head != null ) {
                preOrder.add(head.data);  }
          if(head.left != null)
                this.printPreorder( head.left );
          if(head.right !=null)
                this.printPreorder( head.right );
    }
```

Why is it bad to use fields to replace paramters

Makes is harder to understand the class

- What do the fields have to do with class?

Easier to make errors

- Fields retain values between method calls
- Code on previous slide has error - does not rest value

Removing the field

```
//rew call this twice and you get wrong answer -1
public Vector<Integer> printPreorder(){
      Vector<Integer> preOrder = new Vector<Integer>();
      this.printPreorder(head, preOrder);
return preOrder;
}

//rew Helper method -2
private void printPreorder(Node head, preOrder){
      if ( head != null ) {
            preOrder.add(head.data);  }
      if(head.left != null)
            this.printPreorder( head.left, preOrder);
      if(head.right !=null)
            this.printPreorder( head.right, preOrder );
}
```

Removing the Helper method

```
//rew call this twice and you get wrong answer -1
public Vector<Integer> printPreorder(){
        Vector<Integer> preOrder = new Vector<Integer>();
        if ( head != null ) {
                head.printPreOrder(preOrder);
return preOrder;
}
```

```
class Node {
        private void printPreorder(preOrder){
                preOrder.add(head.data);
                if(left != null)
                        left.printPreorder( preOrder);
                if(right !=null)
                        right.printPreorder( preOrder );
        }
```

What is wrong with Helper methods

Code is harder to reuse

- Hidden in helper methods in unrelated classes

Class becomes too complex

- Logic ends up in same class

Why type so much

```
public boolean hasNext()
{
        if(super.hasNext())
        {
                return true;
        }
        else
                return false;
}
```

```
public boolean hasNext()
{
        return super.hasNext()
}
```

Dont repeat yourself

```
public Node(){
        this.setValue(0);
        this.setlChild(new NullNode());
        this.setrChild(new NullNode());
        this.setParent(new NullNode());
        this.visited = false;
        this.setlDepth(0);
        this.setrDepth(0);
}
```

```
public Node(){
        Node(0);
}

public Node(int x){
        this.setValue(x);
        this.setlChild(null);
        this.setrChild(null);
        this.setParent(null);
        this.visited = false;
        this.setlDepth(0);
        this.setrDepth(0);
}
```

```
public Node(int x){
        this.setValue(x);
        this.setlChild(null);
        this.setrChild(null);
        this.setParent(null);
        this.visited = false;
        this.setlDepth(0);
        this.setrDepth(0);
}
```

Information hiding

```
/*
 * Returns next Node from the Heap.
 */
 //rew don't return node -2
public Object next()
{
        if(hasNext())
                return list[index++];
        else
                return null;
}
```

Java and Interfaces

```
//rew what is the point of this?
public interface Decorator extends Heap
{
	/*
	 * Add additional methods
	 */
}
```

```
public abstract class AbstractHeap extends AbstractCollection implements Heap
{
	/*
	 * Default COnstructor
	 */
	protected AbstractHeap()
	{

	}
}
```

Decorator Pattern

```
//REW decorator must have same interface as what it decorates -4
public abstract class HeapDecorator {
```

Complex

```
void add(AbstractTreeNode<T> newNode)
{
      //REW this is complex
      if(this.isLessThan(newNode))
      {
        T oldValue = this.getValue();
        this.setValue(newNode.getValue());
        this.add(new MaxHeapNode<T>(oldValue));

      }else
    {
        if(this.getLeftNode().isNull())
        {
            this.setLeftNode(newNode);
        }else if(this.getRightNode().isNull())
        {
            this.setRightNode(newNode);
        }else if(getMaxNodeHeight(this.getLeftNode())
                <= getMaxNodeHeight(this.getRightNode()))
        {
            this.getLeftNode().add(newNode);
        }else
        {
            this.getRightNode().add(newNode);
        }
    }
}
```

Same code less complex

```
void add(AbstractTreeNode<T> newNode){
        if(this.isLessThan(newNode)) {
          T oldValue = this.getValue();
          this.setValue(newNode.getValue());
          this.add(new MaxHeapNode<T>(oldValue));
         return;
        }
    if(this.getLeftNode().isNull())  {
           this.setLeftNode(newNode);
           return;
     }
     if(this.getRightNode().isNull()) {
           this.setRightNode(newNode);
           return;
     }
     if(getMaxNodeHeight(this.getLeftNode())  <= getMaxNodeHeight(this.getRightNode())) {
           this.getLeftNode().add(newNode);
     }else {
           this.getRightNode().add(newNode);
     }
}
```

Names

public class IteratorHeap<E extends Comparable<E>> implements Iterator<E>

public class HeapDS extends AbstractCollection

public boolean add(Object obj,HeapNode n,HeapNode parent,boolean wasLeft)