CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2020
Doc 16 Assignment 2 Comments
Oct 27, 2020

Copyright ©, All rights reserved. 2020 SDSU & Roger Whitney,
5500 Campanile Drive, San Diego, CA 92182-7700 USA.
OpenContent (http://www.opencontent.org/opl.shtml) license
defines the copyright on this document.

Null Object

if (leftChild != null) {
so something}

vs

if (!leftChild.isNull()) {
so something}

Client does not have to explicitly check for null or some other special value

Null Object

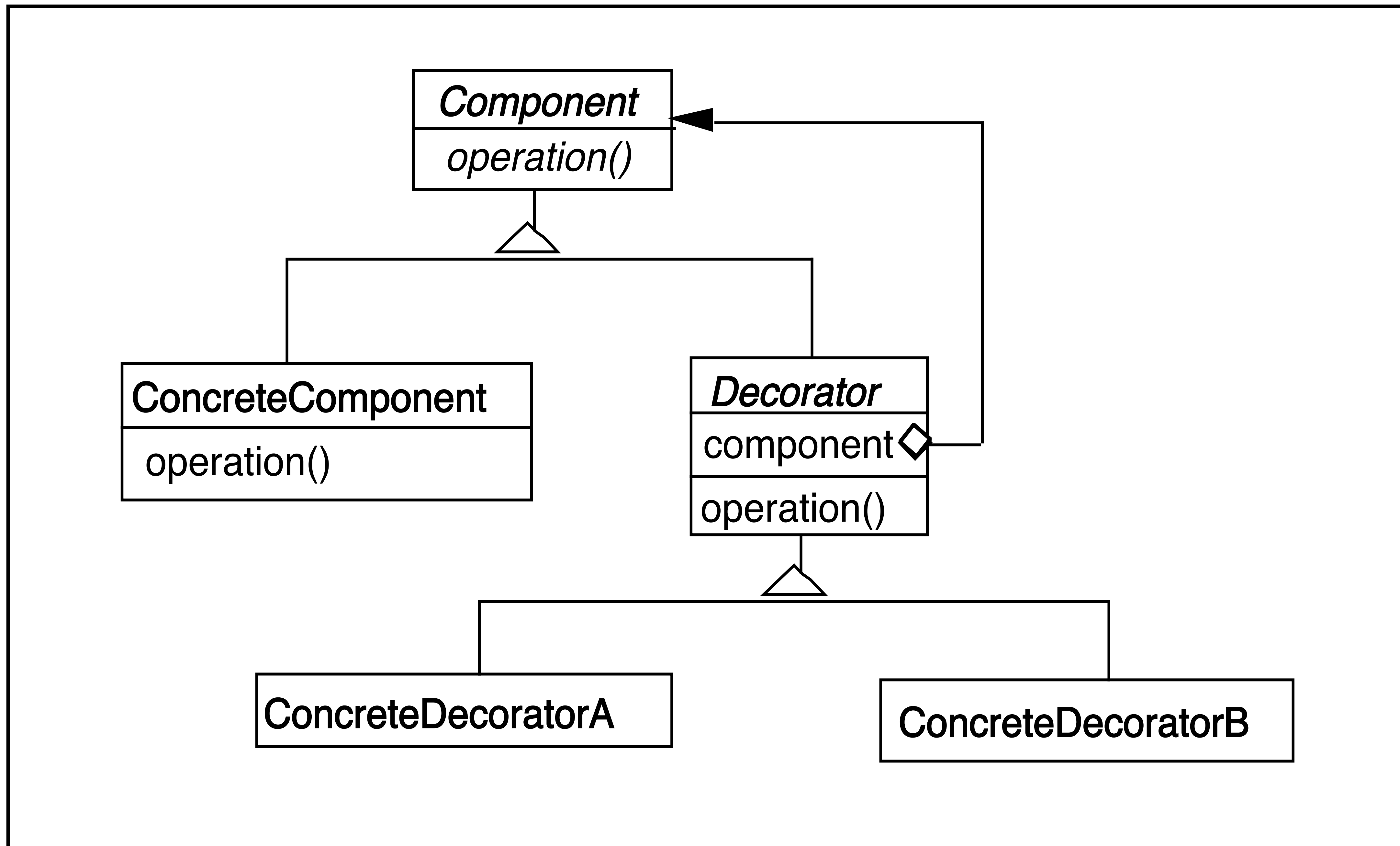
```
class InnerNode ... {
    public String toString() {
        String result = '';
        if (leftChild != null)
            result += leftChild.toString()
        result += value
        if (rightChild != null)
            result +=rightChild.toString();
        return result;
        }
}
```

```
class InnerNode ... {
    public String toString() {
        return left.toString() + value + right.toString();
    }
}

class NullNode ... {
    public String toString() {
        return "";
    }
}
```

Client does not have to explicitly check for null or some other special value

Strategy

```
public class MinHeapStrategy implements HeapStrategy {

    @Override
    public int compare(int currentElement, int newElement) {
        return currentElement < newElement;
    }
}
```



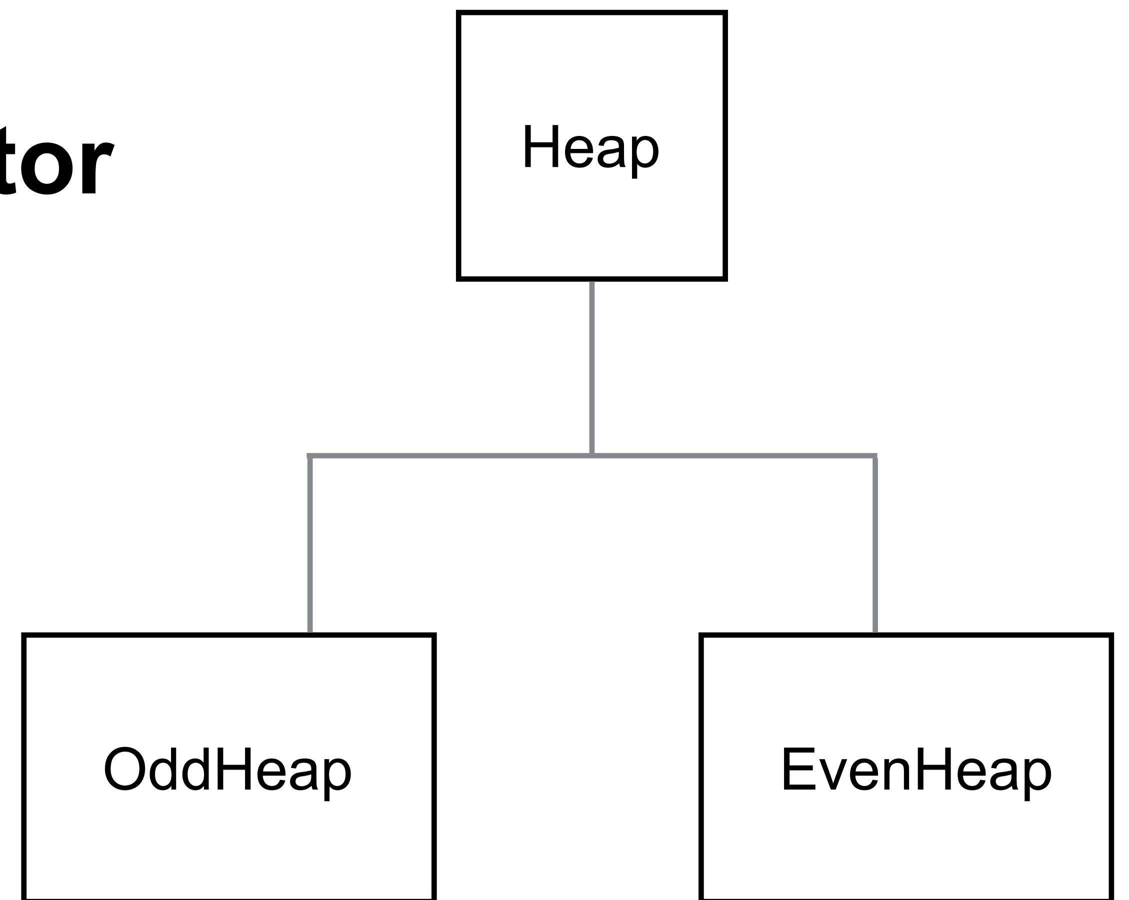
Adds responsibilities to individual objects

- Dynamically
- Transparently

Subclassing is Not Decorator

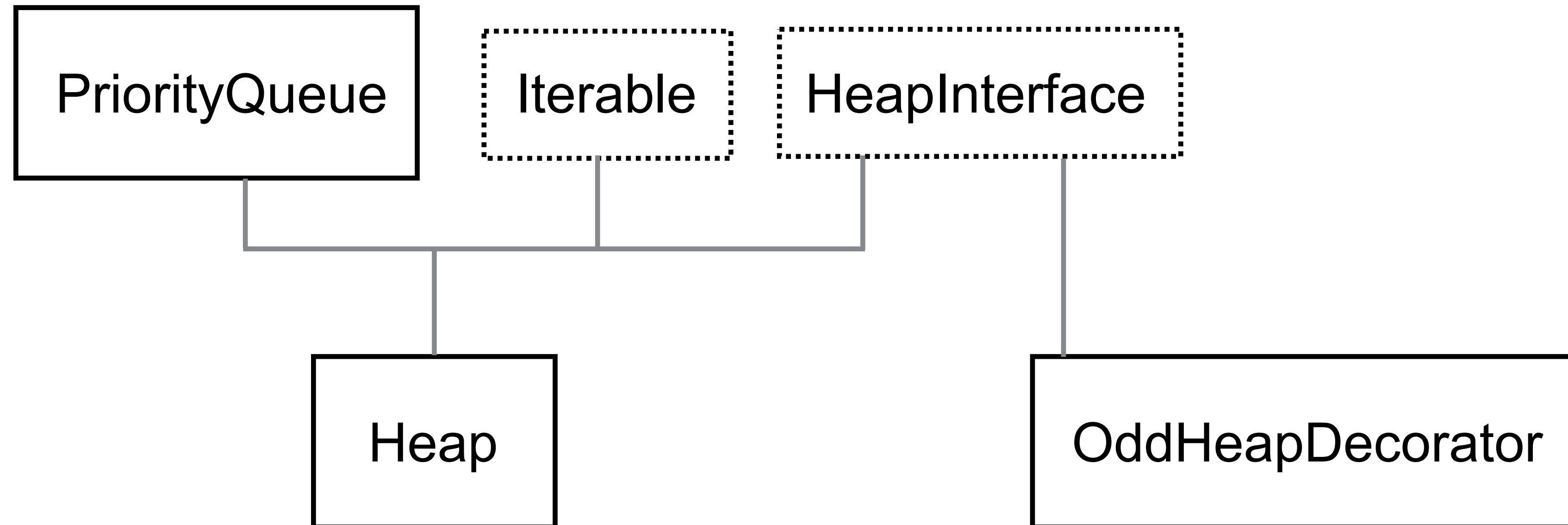
Decorators can be added at run time

```
foo = new Heap();
...
foo = new OddHeap(foo);
```

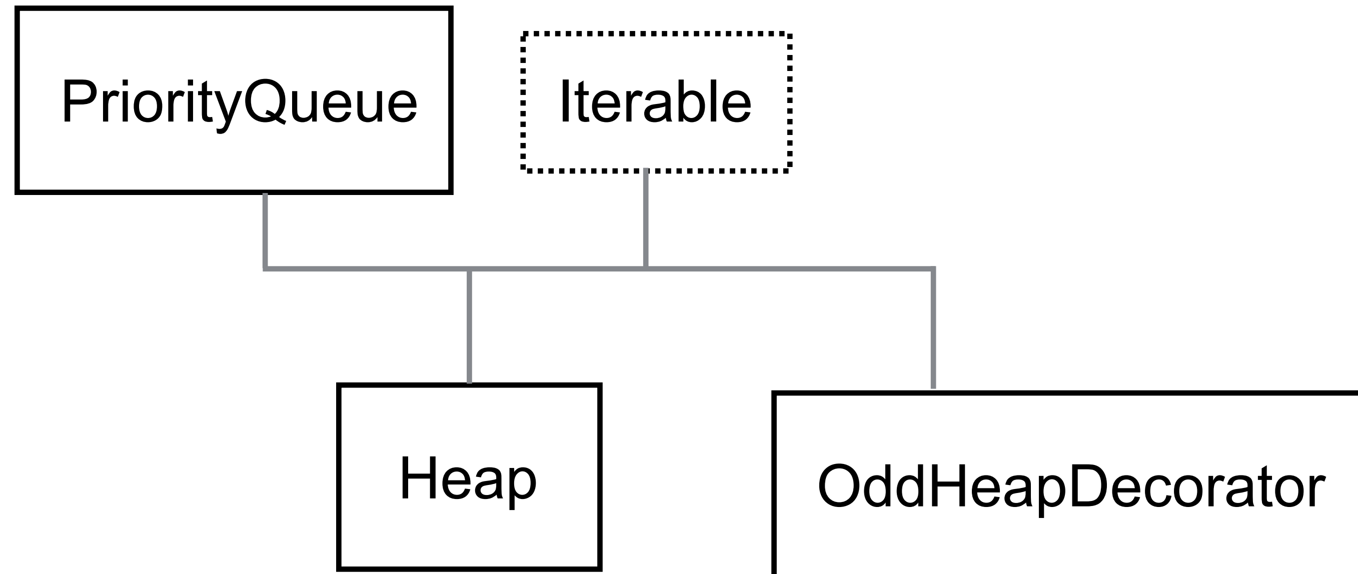


Not Dynamic

Not Transparent



Now Transparent



Internal Iterator

```
class Heap ... {
    public void forEach(Consumer<? super Integer> action) {
        if (root == null) return;
        root.forEach(action);
    }

class InternalNode .... {
    public void forEach(Consumer<? super Integer> action) {
        leftChild.forEach(action);
        action.accept(data);
        rightChild.forEach(action);
    }

class NullNode ... {
    public void forEach(Consumer<? super Integer> action)
    {
        return;
    }
```

toArray

```
class Heap ... {
    public Object[] toArray() {
        ArrayList<Integer> heapElements = new ArrayList<>();
        this.forEach( (value) -> { heapElements.append(value)};
        return heapElements;
    }
```

for Each

```
class Heap ... {
    public void forEach(Consumer<? super Integer> action) {
        if (root == null) return;
        Iterator elements = this.iterator();
        while (elements.hasNext()) {
            action.accept(elements.next());
    }
```

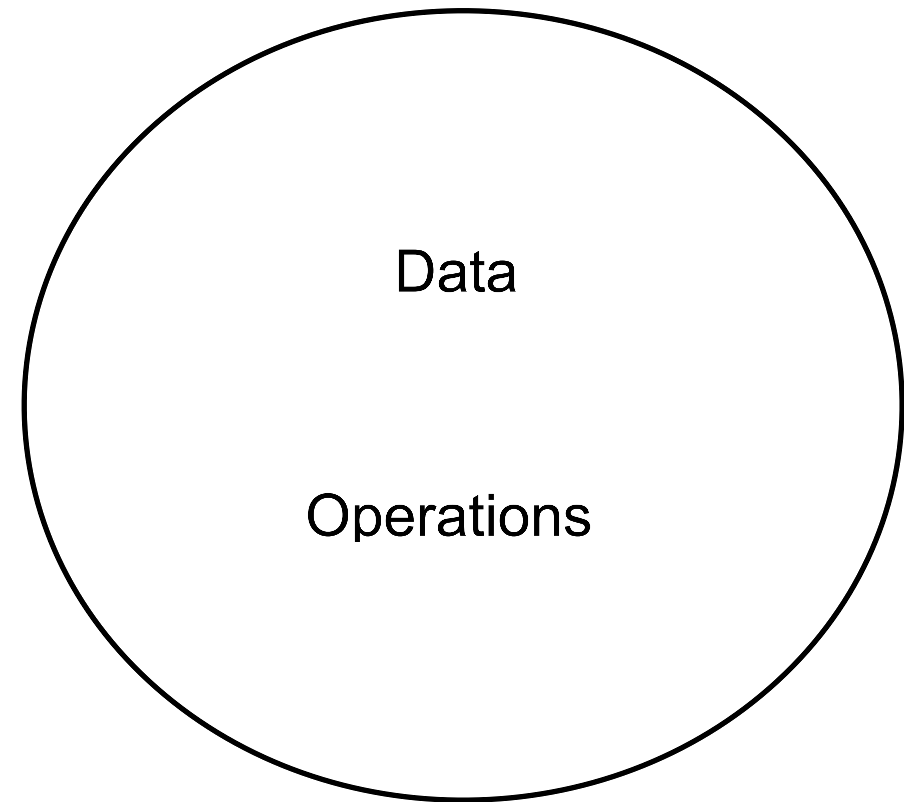
Helper Methods

```
class Heap ... {
    public String toString() {
        String string;
        StringBuilder str=new StringBuilder();
        string=toString(root,str);
        return string;
    }

    private String toString(Node root, StringBuilder str) {
        AbstractNode node=NodeFactory.getNode(root);
        if(node.isNull())
            return "";
        toString(root.left,str);
        str.append(root.value);
        str.append(" ");
        toString(root.right,str);
        return str.toString();
    }
```

```
public String toString() {
    StringBuilder inorderElements=new StringBuilder();
    return toString(root,inorderElements);
}
```

No Helper Methods

```
class Heap ... {
    public String toString() {
        if (root == null) return "";
        return root.toString();
    }
}

class InnerNode ... {
    public String toString() {
        return left.toString() + value + right.toString();
    }
}

class NullNode ... {
    public String toString() {
        return "";
    }
}
```



```
public Heap() {
        this.root = null;
        this.strategy = null;
}
```

Info Hiding -5

```
public HeapNode<T> getRoot() {
        return root;
}


// use strategy as parameter to construct the heap
public Heap(Strategy<T> strategy) {
        this.root = null;
        this.strategy = strategy;
}
```

postOrderList should be a local variable not a field

```
// list to store the nodes of the heap in post-order traversal
List<T> postOrderList = new ArrayList<T>();

// call the helper method to store heap values in pre-order traversal
public List<T> postOrder() {

    postOrderHelper(root,postOrderList);
    return postOrderList;
}
```

ForEach does not modify the structure

```
public void forEach(IntOperator operator) {
    Iterator<HeapNode> nodeIterator = iterateNodes();
    while (nodeIterator.hasNext()) {
        HeapNode currentNode = nodeIterator.next();
        currentNode.setValue(operator.op(currentNode.getValue()));
    }
}
```

More Helper Methods

```
public class MinHeapInsertion<E extends Comparable<E>> implements InsertionStrategy {
  @Override
  public void insert(Node current, Comparable newValue) {
    if(current.value.compareTo(newValue) > 0){
      E swap = (E) current.value;
      current.value = newValue;
      newValue = swap;
    }

    if(current.left.isNullNode()){
      current.left = new Node(newValue);
      return;
    } else if (current.right.isNullNode()){
      current.right = new Node(newValue);
      return;
    }

    if (current.left.height() > current.right.height()) {
      insert(current.right, newValue);
    } else {
      insert(current.left, newValue);
```

Helper Methods

```
public boolean insertIntoLeftChild(Node node, int data) {
    try {
        if (node.getLeftChild().isNull()) {
            node.setLeftChild(new Node(node.getData()));
            node.getLeftChild().setParent(node);
            node.setData(data);
            return true;
        } else {
            final int storeData = node.getData();
            node.setData(data);
            final Node newNode = new Node(storeData);
            newNode.setLeftChild(node.getLeftChild());
            node.setLeftChild(newNode);
            newNode.setParent(node);
            return true;
        }
    } catch (Exception ex) {
        return false;
    }
}
```

The Heap is the Context

```
public class Context {
    private Strategy strategy;

    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public void executeStrategy() {
        strategy.createHeap();
    }

}
```

```java
public NodeIterator(MinHeap minHeap) {
    next = minHeap.root;
    if (next.isNull()) {
        return;
    }
    while (!next.getLeftChild().isNull()) {
        next = next.getLeftChild();
    }
}
```

```java
public NodeIterator(Node root) {
    next = root;
    if (next.isNull()) {
        return;
    }
    while (!next.getLeftChild().isNull()) {
        next = next.getLeftChild();
    }
}
```

```
public Heap(Object heapType) {
    root = new NullNode();
    if (heapType instanceof MinHeap) {
        strategy = new MinHeap();
    } else {
        strategy = new MaxHeap();
    }
}
```

If we create another strategy we have to edit this code

```
public Heap(Strategy heapType) {
    root = new NullNode();
    strategy = heapType;
}
```

If we create another strategy
No change needed here

Send Messages instead of instanceof

```
private void toStrings(Node node, StringBuffer string) {

    if (node instanceof HeapNode) {
        string.append(node.getValue());
        if (node.leftNode instanceof HeapNode) {
            string.append(", ");
            toStrings(node.leftNode, string);
        }
        if (node.rightNode instanceof HeapNode) {
            string.append(", ");
            toStrings(node.rightNode, string);
        }
    }
}
```

Why 1

```
AbstractNode node1=NodeFactory.getNode(node)
```

Use Meaningful Names

```
public Object[] toArray()
{
    ArrayList<Integer> arrayList = new ArrayList<>();
    return toArray(root, arrayList);
}
```

```
public Object[] toArray()
{
    ArrayList<Integer> heapElements = new ArrayList<>();
    return toArray(root, heapElements);
}
```

Your strategies are identical except for 1 character -2

```
class MaxHeapStrategy(AbstractStrategy):
    def __init__(self):
        AbstractStrategy.__init__(self)

    # returns root of max heap
    def append(self, parent_node, new_child_node):
        if isinstance(parent_node, NullNode):
            return new_child_node
        if new_child_node.value > parent_node.value:
            parent_value_copy = parent_node.value
            parent_node.value = new_child_node.value
            new_child_node.value = parent_value_copy
            self.append(parent_node, new_child_node)
        else:
            # compare heights of sub-heaps and add to smaller sub-heap
            smaller_subheap_root_attr_name = 'left_child'
            if parent_node.right_child.get_height() < parent_node.left_child.get_height():
                smaller_subheap_root_attr_name = 'right_child'
            smaller_subheap_root = getattr(parent_node, smaller_subheap_root_attr_name)
            new_child_node = self.append(smaller_subheap_root, new_child_node)
            setattr(parent_node, smaller_subheap_root_attr_name, new_child_node)
        return parent_node
```

```
public class OddFilter implements Iterator<Object>{
    Iterator iterator;

    OddFilter(Iterator input) {
        this.iterator = input;
    }

    @Override
    public boolean hasNext() {
        return iterator.hasNext();  //Wrong, if all elements are even then none remain
```

Use Java Interface -2

```
public interface Iterator<Node> {
    boolean hasNext();
    int next();
}
```