CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2009
Doc 7 Assignment 2 Comments
Feb 17, 2011

Design Issues

Heap abstraction & Printing

Heap abstraction & odd

When to comment, what to comment

Comments

Duh Comments

```
//If Root is Null
if (rootNode == null)
```

```
// ---------------
// Construction
// ---------------
/**
 * Creates a new {@code MaxHeap} with a null root.
 */
public MaxHeap() {
    this.root = null;
}
```

```
/**
 * Create MaxHeap (this is the only public constructor)
 */
public MaxHeap(){
      this(null, 0);
}
```

```
#Setup MaxHeap with no initial elements for each test
def setup
 @max_heap = MaxHeap.new([])
end


 def teardown
  ## Nothing really
 end
```

```
/**
 * Get how many items are on the heap.
 * @return number of items on this heap
 */
public int size(){
      // Not strictly part of the requirements
      // but this is useful for testing - clients would probably find it
      // useful too.
      return cnt;
}
```

```
public void insert(int newValue) {
    // If this is root, and it's not set, then set it and return
    if (value == null) {
        value = newValue;
        return;
    }
etc.
```

```
public void insert(int newValue) {
    if (isRoot()) {
        value = newValue;
        return;
    }
```

```
public interface HeapADT<E> extends Iterable<E> {

//  Adds the Object obj to the heap
    public void add(E obj);

//  Returns an Iterator of the values in the heap, presented in
//  the preorder.
    public Iterator<E> iterator();

//  Returns array containing all the odd values in the heap in
//  preorder, using the iterator.
    public Object[] oddValuesInPreOrder();

//  Returns array containing all the values in the heap in
//  preorder, using the iterator.
    public Object[] allValuesInPreOrder();

}
```

```
        /
*==================================================================
         * FUNCTION:    isEmptyNode
         *
         * DESCRIPTION: Check if the node of the heap(HeapNode) is
         *              null or not.
         *
         * @param     : node is the MaxHeap's HeapNode which will be
         *                       checked.
         *
         * @return    : true if null, false otherwise.

*==================================================================*/
        private boolean isEmptyNode(HeapNode node)
        {
            return ( null == node );
        }
```

```
// if the root is empty then the child becomes the root
if (parent == null) {
    return child;
}

// the root is not empty so we have to compare the child
// to the root for who has the bigger value. If the child
// is bigger then the parent swap their values.
if (parent.getValue() < child.getValue()) {
    int childValue = child.getValue();
    child.setValue(parent.getValue());
    parent.setValue(childValue);
}
```

```
#It is usually not a good idea to manipulate the internal datastructure
#Be careful when using this method
def __get_root()
  return @root
end



#private
def __get_root()
  return @root
end
```

```
#Validate input then call recursive internal insert function
def insert(value)
  raise unless value.is_a?(Numeric)
  @heap_size = @heap_size + 1
  node = Node.new(value)
  if @root.value.nil?
    @root = node
  else
    __insert(node)
  end
end
```

API verses Implementation comment

```
/**
 * The add method takes the node you want to add, and checks to see first if the maxHeap is empty.
 * If so, it makes that node the TopNode. If not, it checks to see if the newNode's value is greater than the current Node's.
 * Otherwise it moves on to check the depth of the left and right subtrees.
 * If the two are equal in height it always picks the left subtree.
 * Otherwise it always adds to the smallest subtree.
 * At the end we update the depths.
 *
 * @param newNode The node you wish to be adding to the MaxHeap.
 */
public void add(Node newNode)
```

```
    /**
     * This is the basic MaxHeap constructor. It makes the top node null, because I'm not
good at Null Element design pattern.
     */
    public MaxHeap()
    {
        topNode = null;
    }
```

```
public class CNode {

    private int value;
    private CNode left;         //Left child
    private CNode right;   //Right child
```

```
/*===================================================================
 * MaxHeap implementation.
 *===================================================================*/
public class MaxHeap
{
       /* Number of nodes in the MaxHeap.*/ //Duh
         private int iMaxHeapSize;
```

```
cnt++; // increment the number of nodes on the heap
```

Names

```
private int cnt = 0;
```

```
public Boolean hasNoChild() //isLeaf() better name
```

Info Hiding

```
/**
 * Overload Constructor
 * @param node
 */
public Heap(Node node)
{
      this.rootNode = node;
}
```

Info Hiding Conceptual

```
public void insertNode(int number)
```

Static

```
public class MaxHeap extends Node{
    static Node root;
```

Static

```
public static HeapNode addNode(HeapNode parent, HeapNode child) {
    if (parent == null) {
        return child;
    }

        if (parent.getValue() < child.getValue()) {
        int childValue = child.getValue();
        child.setValue(parent.getValue());
        parent.setValue(childValue);
    }

    if (parent.getLeftNode() == null) {
        parent.setLeftNode(child);
        return parent;
    } else if (parent.rightNode == null) {
        parent.setRightNode(child);
        return parent;
    }
```

Java Standard Name & Polymorphism

```
//REW -1 name
public void addValueToHeap(int value) {
```

Struct

```
class Node
  attr_accessor :left, :right, :value

  def initialize(value)
    @left = nil
    @right = nil
    @value = value
  end
end
```

Abstraction

```
// REW -2 how is this part of Heap abstraction
public void printOddInPreorder(){
      this.printOddInPreorder(head);
}
```

System.out.Println

```
    public Vector printLevelOrder(){
        levelOrder = new Vector();
        index = 0;
        printLevelOrder(head);
        System.out.println();
        return levelOrder;
}
```

When are we done?

```
while(doneFlag != true)
```

```
while(!doneFlag)
```

Oh when we add the node we are done

```
while(nodeNotAdded)
```

One Method

```
public void add(Node newNode)
{
    Node currentNode = this.topNode;
    Node previous = null;
    int lastMove = -1;
    //REW -1 name
    boolean doneFlag = false;
    int currentPlace = 1;
    while(doneFlag != true)
    {
        /*
         * This big if block deals with if the current given node is null.
         * If it is, we check to see if previous is null. If so, we are at the top node.
         * If not, we are at some random node, and Previous is available to be made the parent.
         * We also have another item, lastMove, which tells us if our current node is the left or right child
         * of its parent.
         */
        if(currentNode == null)
        {
            if(previous == null)
            {
                this.topNode = newNode;
                this.topNode.setDepth(0);
                //REW -1 dont use system.out in code
                System.out.println("Adding a node to the top.");

                doneFlag = true;
            }
            else
            {
                currentNode = newNode;
                if(lastMove == 0)
                {
                    previous.setlChild(currentNode);
                    currentNode.setParent(previous);
                    currentNode.setDepth(currentPlace);
                    System.out.println("Adding a node.");
                    doneFlag = true;
                }
                else if(lastMove == 1)
                {
                    previous.setrChild(currentNode);
                    currentNode.setParent(previous);
                    currentNode.setDepth(currentPlace);
                    System.out.println("Adding a node.");
                    doneFlag = true;
                }
                else
                {
                    System.out.println("What the F are you doing?");
                    System.out.println("Exiting without doing anything.");
                    doneFlag = true;
                }
            }
        }

        */
        else
        {
            if(newNode.getValue() > currentNode.getValue())
            {
                Node tempNode = currentNode;
                currentNode = newNode;
                currentNode.setlChild(tempNode.getlChild());
                currentNode.setrChild(tempNode.getrChild());
                tempNode.setlChild(null);
                tempNode.setrChild(null);
                tempNode.setParent(null);
                currentNode.setDepth(currentPlace);
                System.out.println("Switching two items.");
                this.add(tempNode);
                doneFlag = true;
            }
            else
            {
                Node lChild = currentNode.getlChild();
                Node rChild = currentNode.getrChild();
                int ldepth;
                int rdepth;
                if(lChild == null)
                    ldepth = 0;
                else
                    ldepth = lChild.getDepth();
                if(rChild == null)
                    rdepth = 0;
                else
                    rdepth = rChild.getDepth();
                if(ldepth > rdepth)
                {
                    previous = currentNode;
                    currentNode = currentNode.getrChild();
                    currentPlace ++;
                    lastMove = 1;
                    System.out.println("Reiterating.");
                }
                else if(ldepth <= rdepth)
                {
                    previous = currentNode;
                    currentNode = currentNode.getlChild();
                    currentPlace ++;
                    lastMove = 0;
                    System.out.println("Reiterating.");
                }
                else
                {
                    System.out.println("What in the F is wrong with this.");
                    System.out.println("Bug alert bug alert.");
                    doneFlag = true;
                }
            }
        }
    }
}
```

Formatting

```
private static int leftNodeHeight = 0;
private static int rightNodeHeight = 0;
public Node(int key)
{
    this.value = key;
    leftNode = null;
    rightNode = null;
}
                                              }
                                                return java.lang.Math.max(leftNodeHeight, rightNodeHeight);
                                            }
                                        }
                                        public Boolean hasNoChild()
                                        {
                                            if (this.leftNode == null && this.rightNode == null)
                                            {
                                                return true;
                                            }
```

Local Variables as Field, Name

```
boolean nodeExists = false;
private boolean Exists(Node rootNode, Node newNode) {
    if(rootNode == null) { nodeExists = false;
    } else {
        if(rootNode.value == newNode.value) {
            nodeExists = true;
            return nodeExists;
        }
        if(rootNode.leftNode != null) {
            nodeExists = Exists(rootNode.leftNode,newNode);
        }
        if(rootNode.rightNode != null) {
            nodeExists = Exists(rootNode.rightNode,newNode);
        } else {
            nodeExists = false;
        }
    }
    return nodeExists;
}
```

```
public boolean isEmpty() {
    if (rootValue == null) {
        return true;
    } else {
        return false;
    }
}
```

→

```
public boolean isEmpty() {
    return (rootValue == null)
}
```

```
public int getHeight() {
    int height = 0;
    if (!isEmpty()) {
        height++;
    }
    int leftHeight = 0;
    if (leftChild != null && !leftChild.isEmpty()) {
        leftHeight += leftChild.getHeight();
    }
    int rightHeight = 0;
    if (rightChild != null && !rightChild.isEmpty()) {
        rightHeight += rightChild.getHeight();
    }

    return height + Math.max(leftHeight, rightHeight);
}
```

```
 public int getHeight() {
     if (isEmpty()) {
         return 0;
     }
etc..
```

Tabs & Spaces

```
private class Node<E> {
    private E value;
    private Node<E> leftChild;
     private Node<E> rightChild;




    public IteratorHelper() {
          iterIndex = 0;
            counter = 0;
            sequenceChecker = modificationCounter;
            itterArray = (E[]) new Object[currentSize+1];
            preOrder(root);
    }
```

Formatting & temp

```
public void adjustHeap(Node node)
{

            int temp;
            parent = node.parentNode;

            if (parent != null)
            {
                    if(parent.key < node.key)
                     {
                      temp = parent.key;
                       parent.key = node.key;
                       node.key =temp;
                    }
                adjustHeap(parent);
            }
 }
```