Aufgabe 1: Drehfreudig?

Team-ID: 00540

Team: Eulers Fuß Fanatiker

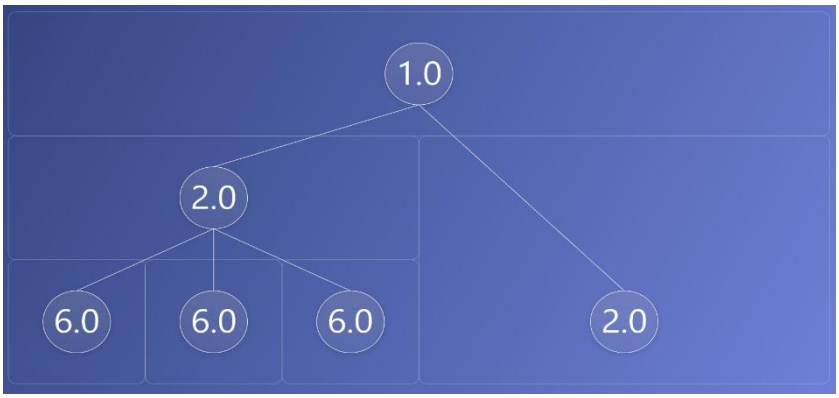
Bearbeiter/-innen dieser Aufgabe:
David Damaschin

1. November 2025

Inhaltsverzeichnis

Lösungsidee

Um zu überprüfen, ob ein Baum drehfreudig ist, müssen die Breiten aller Blattknoten gesammelt und verglichen werden. In diesem Programm wird jedem Knoten eine Zahl zugewiesen, die seine Breite repräsentiert. (Die tatsächliche Breite beim Zeichnen wird dann mit 1/Breite berechnet.) Diese Breite ergibt sich aus der Breite des Elternknotens multipliziert mit der Anzahl seiner Kindknoten. Beispiel: Ein Knoten, dessen Elternknoten eine Breite von 2 hat und der 3 Geschwister besitzt, erhält eine Breite von 6. Wichtig: Der Wurzelknoten hat immer die Breite 1.



Um zu überprüfen, ob der Baum gedreht werden kann, werden die Breiten aller Blattknoten in korrekter Reihenfolge (von links nach rechts) in einem Array gespeichert. Anschließend wird – ähnlich wie bei der Palindrom-Prüfung – verglichen, ob der erste Wert mit dem letzten übereinstimmt, der zweite mit dem vorletzten usw.

Hinweis: Im Menü des Programms gibt es die Option „Toggle Show Width". Nach dem Starten ist diese Option standardmäßig deaktiviert. Wenn Sie sie aktivieren, werden die Breiten als Zahlen innerhalb der Knoten angezeigt. Außerdem muss der Button am unteren Rand des Bildschirms gedrückt werden (nur sichtbar wenn ein Baum schon geladen ist oder ein neuer Baum erstellt

wurde) um zu überprüfen ob er drehbar ist. Sobald man den Knop drückt, sieht man einen gespiegelten Baum unter dem richtigen Baum und oben steht ein Text der entweder sagt: „Tree can be flipped“ oder „Tree cannot be flipped“.

Umsetzung

Der Baum wird aus einer Textdatei eingelesen, die nur aus öffnenden (und schließenden) Klammern besteht. Jedes Klammerpaar repräsentiert einen Knoten im Baum.

Die Methode loadTreeFromFile() validiert zunächst die Eingabe: Sie prüft, ob die Datei nicht leer ist, ob nur gültige Zeichen (Klammern) vorkommen und ob die Zeichenkette mit einer öffnenden Klammer beginnt. Ist dies der Fall, wird ein Wurzelknoten mit der Breite 1 erstellt und die eigentliche Konstruktion des Baums beginnt.

Die rekursive Methode parseTreeString() durchläuft die Zeichenkette Zeichen für Zeichen. Bei jeder öffnenden Klammer (wird ein neuer Kindknoten erstellt und dem aktuellen Knoten hinzugefügt. Anschließend wird die Methode rekursiv mit dem neuen Knoten als aktuellem Knoten aufgerufen. Bei einer schließenden Klammer) wird zum Elternknoten zurückgekehrt und die Rekursion dort fortgesetzt. Sobald die Wurzel wieder erreicht wird und keine Zeichen mehr übrig sind, ist der Baum vollständig aufgebaut.

Nach der Konstruktion wird calculateWidth() aufgerufen, um jedem Knoten seine Breite zuzuweisen. Diese Methode arbeitet ebenfalls rekursiv: Für jeden Knoten wird die Breite als Breite des Elternknotens × Anzahl der Geschwister berechnet. Dies geschieht von oben nach unten vom Wurzelknoten ausgehend, sodass die Breiten aller Knoten korrekt berechnet werden.

Die Überprüfung der Drehfreudigkeit wird in der Methode checkIfTreeCanBeFlipped() implementiert. Diese Methode ruft zunächst getLastLayerWidths() auf, welche alle Breiten der Blattknoten sammelt.

Die Methode getLastLayerWidths() durchläuft den Baum rekursiv mithilfe von collectLastLayerWidths(). Diese prüft für jeden Knoten, ob er ein Blattknoten ist (keine Kinder hat). Falls ja, wird seine Breite in eine ArrayList eingefügt. Falls nein, werden alle Kindknoten rekursiv durchlaufen. Da der Baum von links nach rechts durchlaufen wird, werden die Breiten automatisch in der korrekten Reihenfolge gespeichert.

Anschließend wird die ArrayList in ein int-Array umgewandelt und an checkIfTreeCanBeFlipped() zurückgegeben. Dort erfolgt der eigentliche Vergleich: Eine Schleife läuft nur bis zur Hälfte des Arrays (i < widths.length/2). In jedem Durchlauf wird der Wert an Position i mit dem gespiegelten Wert an Position widths.length-1-i verglichen. Sobald ein Paar nicht übereinstimmt, gibt die Methode false zurück – der Baum ist nicht drehfreudig. Stimmen alle Paare überein, wird true zurückgegeben.

Da die Breiten bereits beim Laden des Baumes in der Methode calculateWidth() festgelegt wurden, welche rekursiv jedem Knoten seine Breite zuweist, muss diese Methode nicht nochmal aufgerufen

werden. Außerdem wird bei jedem Modifizieren die Methode calculateWidth() noch einmal aufgerufen.

Der Baum kann nach dem Laden oder Erstellen durch Mausklicks modifiziert werden. Ein Linksklick fügt dem angeklickten Knoten ein neues Kind hinzu, während ein Rechtsklick das letzte Kind des angeklickten Knotens entfernt.

Die Klick-Verarbeitung erfolgt in der Methode processClick() im Controller. Zunächst wird geprüft, ob ein Baum geladen ist und ob dieser bearbeitet werden darf. Anschließend wird überprüft, ob die Mausposition innerhalb der Zeichenfläche des Baums liegt. Falls ja, wird je nach Klickart entweder addChildAt() oder removeChildAt() aufgerufen.

Die Methode findNodeAndAddChild() durchläuft den Baum rekursiv und sucht den Knoten, dessen Bereich die Klickkoordinaten enthält. Dabei wird für jeden Knoten geprüft, ob die X-Koordinate innerhalb seiner horizontalen Breite liegt und ob die Y-Koordinate in seiner Ebene ist. Wird der richtige Knoten gefunden, wird ihm ein neues Kind hinzugefügt. Die Methode berücksichtigt dabei die hierarchische Struktur: Liegt der Klick nicht in der aktuellen Ebene, werden die Kindknoten rekursiv durchsucht.

Die Methode findNodeAndRemoveChild() funktioniert analog, entfernt jedoch das letzte Kind des gefundenen Knotens mithilfe von removeLast().

Nach jeder Modifikation wird calculateWidth() erneut aufgerufen, um die Breiten aller Knoten neu zu berechnen, da sich durch das Hinzufügen oder Entfernen von Knoten die Baumstruktur und damit auch die Breiten ändern. Anschließend wird der Baum mit redraw() neu gezeichnet, um die Änderungen visuell darzustellen.

Werkzeuge

-Java (Programmiersprache)

-Intellij Idea (um den Code zu schreiben)

-JavaFX (für die Visualisierung)

-Stift und Papier (um die Idee zu entwickeln und aufzuschreiben)

Beispiele

Drehfreudig-01: kann gedreht werden „((((()())(()()))((()())(()()))((()()())(()()())))“

Drehfreudig-02: kann nicht gedreht werden „((()())(()()()))“

Drehfreudig-03: kann nicht gedreht werden „((()())((()())()))“

Drehfreudig-04: kann gedreht werden „((()()())(()()()))“

Drehfreudig-05: kann nicht gedreht werden „((()()())(()())(()()()))“

Drehfreudig-06: kann gedreht werden „((()()()()())((()())(()())(()())))“

Drehfreudig-07: kann nicht gedreht werden „((()(()((()())()))(((()())()()))))“

Drehfreudig-08: kann gedreht werden „((()()))(()()()())(((())(()())(())))(((())(()))(()()))“

Drehfreudig-09: kann nicht gedreht werden „((((()))(()))(()))(((()())(()))(()))(((()())(())))“

Drehfreudig-10: kann gedreht werden „((()))(()))(()))(()))(()))(()))((()()()))(()))(()()))(()))(()()()))((()()))(()()))(()()))(()()))“

Drehfreudig-11: kann nicht gedreht werden „(((((()))()))(()))((()))(()))(()))))(((()))(()))))“

Drehfreudig-12: kann nicht gedreht werden „((()))(()))(()))(()))(()))(()))(((()))(()))(()))(()()))(()))(()))(((()))(()))(()))(()))(()))“

Drehfreudig-13: kann gedreht werden „(((((()))(()))(()))((()))(()))(()))((()))(()))(()))((()))(()))(()))))((()()()()))(()))))))(()))))))(()))))))(()))))))(()))))))(()))))))“

Drehfreudig-14: kann nicht gedreht werden „((()))(()))((()()()))(()))(()))((()))(()()))(()))))“

Drehfreudig-15: kann gedreht werden „((()))((()))(()))(()()))(()()))“

Quellcode

Hier werden die oben erwähnten Methoden dargestellt.

```java
public void loadTreeFromFile(String treeFile) { 1 usage  david +1 *
    isTreeLoaded = false;

    //check that File isnt empty
    if (treeFile == null || treeFile.isEmpty()) {
        System.out.println("Tree file is empty!");
        return;
    }

    //check if all tree characters are valid
    for(int i = 0; i < treeFile.length(); i++){
        if(treeFile.charAt(i) != '(' && treeFile.charAt(i) != ')'){
            System.out.println("Invalid character in file!");
            return; // stopping processing on invalid file
        }
    }
    //checking if tree string starts with ( and then stepping through the string character by character
    if(treeFile.charAt(0) == '('){
        TreeNode rootNode = new TreeNode();
        rootNode.setWidth(1);
        tree = new Tree(rootNode);
        int index = 1; // start after the first (
        parseTreeString(rootNode, treeFile, index);
        calculateWidth(rootNode); //calculating width of each node, this is needed for drawing the tree and checking if the tree can be flipped
        isTreeLoaded = true;
    }
    else{
        System.out.println("Invalid tree string!");
    }
}
```

```java
public void parseTreeString(TreeNode currentNode, String treeFile, int index){ 3 usages david +1
    //checking if current index is valid
    if(index >= treeFile.length()){
        System.out.println("Invalid tree string!");
        isTreeLoaded = false;
        return;
    }
    if(treeFile.charAt(index) == '('){
        //creating new node and adding it as a child to the current node, then stepping to the next character with the new node as current node
        TreeNode newNode = new TreeNode(currentNode);
        currentNode.addChild(newNode);
        parseTreeString(newNode, treeFile, index: index + 1);
    }
    //since we already checked for invalid characters, we can assume that if it is not '(', it must be ')'
    else{
        if(currentNode.getParent() != null){
            parseTreeString(currentNode.getParent(), treeFile, index: index + 1);
        } else {
            //we are back at the root node and the string has been fully processed
            isTreeLoaded = true;
        }
    }
}
```

```java
public void calculateWidth(TreeNode currentNode){ 4 usages david
    if(!currentNode.getChildren().isEmpty())
    {
        for(int i = 0; i < currentNode.getChildrenLength(); i++){
            currentNode.getChildren().get(i).setWidth(currentNode.getWidth() * currentNode.getChildrenLength());
            calculateWidth(currentNode.getChildren().get(i));
        }
    }
}
```

```java
public boolean checkIfTreeCanBeFlipped(){ 1 usage david
    if(!isTreeLoaded) return false;

    int[] widths = getLastLayerWidths();
    if(widths != null){
        for(int i = 0; i < widths.length/2; i++){
            if(widths[i] != widths[widths.length-1-i]){
                return false; // Tree cannot be flipped
            }
        }
    }
    else{
        System.out.println("Error calculating last layer widths.");
        return false;
    }
    return true;
}
```

```java
private int[] getLastLayerWidths(){ 1 usage  david
    if(tree == null || !isTreeLoaded) return null;

    java.util.ArrayList<Integer> widths = new java.util.ArrayList<>();

    collectLastLayerWidths(tree.getRootTreeNode(), widths);

    int[] widthsArray = new int[widths.size()];
    for(int i = 0; i < widths.size(); i++){
        widthsArray[i] = widths.get(i);
    }

    return widthsArray;
}
```

```java
private void collectLastLayerWidths(TreeNode node, java.util.ArrayList<Integer> widths){ 2 usages  david
    //checking if current node is a leaf (even if this node isn't in the last layer, leaf node will extend to last layer when drawn)
    boolean isLeaf = (node.getChildren() == null || node.getChildren().isEmpty());

    if(isLeaf){
        widths.add((int)node.getWidth());
    }
    else{
        for(TreeNode children: node.getChildren()){
            collectLastLayerWidths(children, widths);
        }
    }
}
```

```java
public void processClick(MouseEvent event, boolean rightClick){ 2 usages  david
    //processing mouse click, checking if a box was clicked
    if(treeManager.isTreeLoaded() && treeManager.canBeEdited()){
        int[] treeDimension = treeManager.getBoxDimensions();
        if(event.getX() >= treeDimension[1] && event.getX() <= treeDimension[1]+treeDimension[0] && event.getY() >= treeDimension[3] && event.getY() <= treeDimension[3]+treeDimension[2]){
            if(rightClick){
                treeManager.removeChildAt(event.getX(), event.getY());              }
            else{
                treeManager.addChildAt(event.getX(), event.getY());
            }
            redraw();
            if(stage.getTitle().charAt(stage.getTitle().length()-1) != '*')
                stage.setTitle(stage.getTitle() + "*");
        }
    }
}
```

```java
public void addChildAt(double x, double y){ 1 usage  david +1
    findNodeAndAddChild(tree.getRootTreeNode(), x, y, currentX: 0, layer: 1);
    calculateWidth(tree.getRootTreeNode());
}
```

```java
public void findNodeAndAddChild(TreeNode node, double x, double y, double currentX, int layer){  2 usages  david +1

    if(x >= boxOffsetX +  currentX && x <= boxOffsetX + currentX + boxWidth/node.getWidth()){
        if(y >= boxOffsetY + (layer-1) * (boxHeight/(double)tree.getLayers()) && y<= boxOffsetY + layer * (boxHeight/(double)tree.getLayers())){
            node.addChild(new TreeNode(node));
        }
        else {
            if (node.getChildren() == null || node.getChildren().isEmpty()) {
                node.addChild(new TreeNode(node));
            } else {
                double childX = currentX;
                for (TreeNode child : node.getChildren()) {
                    findNodeAndAddChild(child, x, y, childX,  layer: layer + 1);
                    double childPixelWidth = boxWidth / child.getWidth();
                    childX += childPixelWidth;
                }
            }
        }
    }
}
```

```java
public void removeChildAt(double x, double y){  1 usage  david +1
    findNodeAndRemoveChild(tree.getRootTreeNode(), x, y,  currentX: 0,  layer: 1);
    calculateWidth(tree.getRootTreeNode());
}
```

```java
public void findNodeAndRemoveChild(TreeNode node, double x, double y, double currentX, int layer){  2 usages  david +1

    if(x >= boxOffsetX +  currentX && x <= boxOffsetX + currentX + boxWidth/node.getWidth()){
        if(y >= boxOffsetY + (layer-1) * (boxHeight/(double)tree.getLayers()) && y<= boxOffsetY + layer * (boxHeight/(double)tree.getLayers())){
            if(node.getChildren() != null && !node.getChildren().isEmpty())
                node.getChildren().removeLast();
        }
        else {
            if (node.getChildren() != null || !node.getChildren().isEmpty()) {
                double childX = currentX;
                for (TreeNode child : node.getChildren()) {
                    findNodeAndRemoveChild(child, x, y, childX,  layer: layer + 1);
                    double childPixelWidth = boxWidth / child.getWidth();
                    childX += childPixelWidth;
                }
            }
        }
    }
}
```