ALDÜ – Handout Gruppe1

Alle 13 Beispiele findet Ihr im GitHub-Repository.
https://github.com/fhc02pw/Ald_Uebungen_Grupp1.git

Die Logik, also wie eine bestimmte Aufgabenstellung gelöst wurde, ist im Code mittels Kommentaren für die zu lösenden Beispiele 4, 6, 10 und 13 beschrieben.

Beispiel 4

Angabe

[A04] Baum traversieren

Implementieren Sie die zwei nachfolgenden Methoden für das Wörterbuch. Die dazu benötigten Klassen finden Sie im Projekt. Dort finden Sie auch JUnit-Tests, damit Sie Ihre Implementierung überprüfen können.

- Public int countWordsInSubTree(Wort w)
- Public Set<String> getWordsWithPrefix(String prefix)

Lösung

```
/**
 * Zählt alle Wörter des Teilbaums ab einem bestimmten Wort
 * @param w Wort
 * @return Zahl der Wörter (=Anzahl der Elemente)
 */
public int countWordsInSubTree(Wort w) {
    if(w == null) //Wort gibt es im Baum nicht.
        return 0;

    int currentWordCount = 1;                           //übergebenes Element selbst.
    currentWordCount += countWordsInSubTree(w.getLeft());   //Rekursiver Aufruf, damit die Anzahl unter dem linken Teilbaum, berechnet wird
    currentWordCount += countWordsInSubTree(w.getRight());  //Rekursiver Aufruf, damit die Anzahl unter dem rechten Teilbaum, berechnet wird
    return currentWordCount;
}

/**
 * Liefert die Menge aller Wörter retour, die ein spezifisches Präfix haben.
 * @param prefix Wörter müssen diesen Präfix haben
 * @return Menge aller zutreffenden Wörter
 */
public Set<String> getWordsWithPrefix(String prefix) {

    Set<String> foundWords = new HashSet<>();
    addWordsWithPrefix(getRoot(), prefix, foundWords); //Startet die rekursive Suche im Baum für den gegebenen Prefix.
    return foundWords;

}

private void addWordsWithPrefix(Wort w, String prefix, Set<String> set)
{
    if(w == null) //Brich ab, wenn das Wort null ist
        return;

    //Aus aktuellem Wort muss der Prefix extrahiert werden, damit der Vergleich funktioniert.
    String prefixOfCurrentWord = w.getWort().substring(0, prefix.length());
    int cmpVal = prefix.compareTo(prefixOfCurrentWord);

    //Wenn die Prefixes übereinstimmen, wird es dem Set hinzugefügt.
    if(cmpVal == 0)
        set.add(w.getWort());

    //Rekursiver Aufruf, damit die linken und rechten Teilbäume ebenfalls durchsucht werden.
    addWordsWithPrefix(w.getLeft(), prefix, set);
    addWordsWithPrefix(w.getRight(), prefix, set);
}
```

Beispiel 6

Angabe

[A06] Tiefensuche

Im Package finden Sie die notwendigen Basisklassen, um eine Tiefensuche in einem Baum zu implementieren. Auch passende JUnit-Tests, um Ihre Implementierung zu überprüfen, sind dort vorhanden.

Eingefügt in den Baum werden Film-Objekte, als Sortierkriterium soll deren Länge dienen. Sie müssen also die compare()-Methode entsprechend implementieren.

- public List<String> getNodesInOrder(Node<Film> node)
- public List<String> getMinMaxPreOrder(double min, double max)

Lösung

```
@Override
/**
 * Sortierkriterium im Baum: Länge des Films
 */
protected int compare(Film a, Film b) {

    return Double.compare(a.getLänge(), b.getLänge());
}

/**
 * Retourniert die Titelliste der Film-Knoten des Teilbaums in symmetrischer Folge (engl. in-order, d.h. links-Knoten-rechts)
 * @param node Wurzelknoten des Teilbaums
 * @return Liste der Titel in symmetrischer Reihenfolge
 */
public List<String> getNodesInOrder(Node<Film> node) {

    List<String> filmsInOrder = new LinkedList<>();
    addNodesInOrder(node, filmsInOrder); //starte rekursiven Aufruf um Filme aus Teilbaum des Übergabefilms zu retournieren
    return filmsInOrder;
}

public void addNodesInOrder(Node<Film> node, List<String> filmsInOrder) {

    if(node == null) //Brich ab, wenn kein Film gefunden
        return;

    //Die Reihenfolge der aufgerufenen Methoden entspricht 'in-order' (links - Node - rechts)
    addNodesInOrder(node.getLeft(), filmsInOrder);      //Suche (REKURSION) zuerst im linken Teilbaum und füge dort alle Elemente ein
    filmsInOrder.add(node.getValue().getTitel());       //füge aktuelles Element ein
    addNodesInOrder(node.getRight(), filmsInOrder);     //Suche (REKURSION) im rechten Teilbaum und füge von dort alle Elemente ein
}

/**
 * Retourniert Titelliste jener Filme, deren Länge zwischen min und max liegt, in Hauptreihenfolge (engl. pre-order, d.h. Knoten-links-rechts)
 * @param min Minimale Länge des Spielfilms
 * @param max Maximale Länge des Spielfilms
 * @return Liste der Filmtitel in Hauptreihenfolge
 */
public List<String> getMinMaxPreOrder(double min, double max) {

    List<String> minMaxPreOrder = new LinkedList<>();
    addMinMaxPreOrder(this.getRoot(), min, max, minMaxPreOrder); //Starte rekursiven Aufruf
    return minMaxPreOrder;
}
```

```java
private void addMinMaxPreOrder(Node<Film> node, double min, double max, List<String> minMaxPreOrder) {

    if(node == null) //Brich ab, wenn kein Node
        return;

    //Hole die Länge des Filmtitels (Es werden ja nur Filme retouniert, welche innerhalb von min und max liegen)
    double laenge = node.getValue().getLänge();

    //Ist der aktuelle Filmtitel innerhalb von min und max, so füge diesen dem Titel hinzu
    if(laenge <= max && laenge >= min)
        minMaxPreOrder.add(node.getValue().getTitel());

    //Wenn die aktuelle Länge noch größer ist, als min, dann Suche im linken Teilbaum weiter, bis min unterschritten wurde
    if(min <= laenge)
        addMinMaxPreOrder(node.getLeft(), min, max, minMaxPreOrder);

    //Wenn die aktuelle Länge noch kleiner ist, als max, dann Suche im rechten Teilbaum weiter, bis max überschritten wurde
    if(max >= laenge)
        addMinMaxPreOrder(node.getRight(), min, max, minMaxPreOrder);
}
```

Beispiel 10

Angabe

[A10] DijkstraPQShortestPath

Implementieren Sie den Dijkstra-Algorithmus mit Heap (DijkstraPQShortestPath) anhand der Folien nach. Setzen Sie die beiden Arrays pred[](Vorgänger) und dist[](kumulierte Entfernung) entsprechend und verwenden Sie für den Heap die Klasse VertexHeap.

Lösung

```java
/**
 * Berechnet *alle* kürzesten Wege ausgehend vom Startknoten Setzt dist[]-
 * und pred[]-Arrays, kein Rückgabewert
 *
 * @param from
 *            Startknoten
 */
protected boolean calculatePath(int from, int to) {

    VertexHeap heap = new VertexHeap(graph.numVertices());

    //initialisiert die Vorgänger und den Heap
    initPredecessorAndHeap(heap);

    //from ist unser startknoten und hat keine Kosten (Entfernung = 0).
    heap.setCost(from, 0); //innerhalb von setCost wird ein swim durchgeführt, wodurch der Startknoten ganz nach oben wandert
    dist[from] = 0;

    while(!heap.isEmpty()) //Algorithmus so lange wiederholen, bis der Heap leer ist (und somit alle Knoten abgearbeitet worden sind).
    {
        //holt nächsten Knoten aus dem Heap.
        Vertex currentV = heap.remove();

        //Für jede Kante aus dem aktuellen Knoten muss für den Zielknoten die Entfernung vom Start aktualisiert werden (in dist und heap),
        //wenn der neue Weg kürzer ist als der alte.
        List<WeightedEdge> edges = graph.getEdges(currentV.vertex);
        for(WeightedEdge we : edges)
        {
            int totalDistance = dist[currentV.vertex] + we.weight;

            //Weg nur aktualisieren, wenn der neue kürzer ist, als der alte.
            if(totalDistance < dist[we.to_vertex])
            {
                dist[we.to_vertex] = totalDistance;
                pred[we.to_vertex] = currentV.vertex;
                heap.setCost(we.to_vertex, totalDistance);
            }
        }
    }

    return true;

}
```

```java
/*
 * Initialisiert die Vorgänger pred[] auf -1 (Wir wissen aktuell nicht, von welchem Knoten wir zu welchem Knoten gelangen). Das
 * berechnet Dijkstra.
 * Weiters werden alle Knoten in den Heap eingefügt (Im Heap befinden sich alle Knoten mit der aktuellen Distanz vom Start entfernt).
 * */
private void initPredecessorAndHeap(VertexHeap heap) {
    pred = new int[graph.numVertices()];
    for(int i = 0; i < graph.numVertices(); i++)
    {
        pred[i] = -1;
        heap.insert(new Vertex(i, dist[i]));
    }
}
```

Beispiel 13

Angabe

[A13] MergeSort

Im Package finden Sie die Klasse Person. Erweitern Sie die Klasse um die Funktion public int compareTo(Person p)Die Funktion soll anhand des Nachnamens, bei gleichen Nachnamen anhand des Vornamens, eine alphabetische Ordnung der beiden Objekte ermöglichen. Ähnlich der compareTo()-Funktion der String-Klasse soll eine negative Zahl retourniert werden, wenn die bergebene Person (Nachname+Vorname) später im Alphabet kommt, eine positive Zahl, wenn die übergebene Person früher im Alphabet kommt und Null, wenn Nachname und Vorname identisch sind.

Aufbauend auf dieser Klasse implementieren Sie im Package die Klasse MergeSort die Methode

public void sort(Person[] personen)

Im Package finden Sie auch passende JUnit-Tests. Diese leiten sich von der Klasse PersonenSortTest ab. Für die ausgeführten Tests müssen Sie also in dieser Klasse nachsehen.

Lösung

```java
/**
 * Vergleicht zwei Personen miteinander
 * @return <0, wenn a<b || =0, wenn a=b || >0, wenn a>b
 */
public int compareTo(Person p) {
    //Sortiert zuerst nach Nachnamen und dann nach Vornamen.

    int cmpVal = this.getNachname().compareTo(p.getNachname());

    if(cmpVal != 0)
        return cmpVal;

    return this.getVorname().compareTo(p.getVorname());
}
```

```java
/**
 * Sortier-Funktion
 */
public void sort(Person[] personen) {
    // Start des rekursiven Aufrufs für das gesamte Array
    sort(personen, 0, personen.length-1);
}

/**
 * Rekursive Funktion zum Sortieren eines Teils des Arrays
 * @param personen Zu sortierendes Array
 * @param start    Startpunkt im Array
 * @param end      Endpunkt im Array
 */
public void sort(Person[] personen, int start, int end)
{
    //Nur mehr 1 Element bei diesem Aufruf. Teile das Array nicht weiter sondern brich ab, damit im übergeordneten Aufruf
    //die Elemente in der richtigen Reihenfolge getauscht werden können.

    if((end - start) < 1)
        return;


    int mitte = start + (end - start)/2;

    //Array immer in der Mitte teilen, bis nur noch 1 Element übrig ist.
    sort(personen, start, mitte);
    sort(personen, mitte + 1, end);

    // Für Merge: Hälften in eigene Arrays kopieren
    // Hinweis: bei copyOfRange ist Obergrenze exklusiv, deshalb "+ 1"
    Person[] teil1 = Arrays.copyOfRange(personen, start, mitte + 1);
    Person[] teil2 = Arrays.copyOfRange(personen, mitte + 1, end+1);

    // Die beiden extrahierten Hälften sind für sich schon sortiert. Nun müssen
    //aus den beiden Teilen die einzelnen Elemente in der richtigen Reihenfolge im gesamten Array eingefügt werden.
    merge(teil1, teil2, personen, start);
}
```

```java
/**
 * Merge zweier Arrays in ein Ergebnis-Array
 * @param pers1 Erstes Array
 * @param pers2 Zweites Array
 * @param result Ergebnisarray
 * @param start  Position für Speicherung in Ergebnisarray
 */
public void merge(Person[] pers1, Person[] pers2, Person[] result, int start) {

    int indexP1 = 0;
    int indexP2 = 0;

    int i = 0;

    while(indexP1 < pers1.length || indexP2 < pers2.length) //so lange ausführen, bis beide Teile einsortiert sind.
    {
        Person p1 = null;

        //Bestimme Person aus dem Teil 1, bis dieses vollständig abgearbeitet wurde
        if(indexP1 < pers1.length)
            p1 = pers1[indexP1];

        Person p2 = null;

        //Bestimme Person aus dem Teil 2, bis dieses vollständig abgearbeitet wurde
        if(indexP2 < pers2.length)
            p2 = pers2[indexP2];

        //Schon alle Personen aus Teil 1 abgearbeitet. Füge nur noch Personen aus Teil 2 ins Gesamtarray.
        if(p1 == null)
        {
            result[start + i] = p2;
            indexP2++;
        }
        //Schon alle Personen aus Teil 2 abgearbeitet. Füge nur noch Personen aus Teil 1 ins Gesamtarray.
        else if(p2 == null)
        {
            result[start + i] = p1;
            indexP1++;
        }

        //Personen aus beiden Teilen müssen noch eingefügt werden. Füge jene Person ein, die im Ergebnisarray vorher vorhanden sein muss.
        else //compare the 2 persons.
        {
            int cmpVal = p1.compareTo(p2);

            if(cmpVal < 0) //Negativer Wert. P1 ist vor P2 -> Füge P1 ein.
            {
                result[start + i] = p1;
                indexP1++;
            }
            else if(cmpVal > 0) //Positiver Wert. P1 ist nach P2 -> Füge P2 ein.
            {
                result[start + i] = p2;
                indexP2++;
            }
        }

        i++;
    }
}
```