Praktische Informatik II
Tutor:in: Tobias Diehl

SoSe 2024
Bearbeiter:in: Altug Uyanik, Öykü Koç

Übungsblatt 5

Lösungsvorschlag

Aufgabe 1 Qual der Wahl

Methoden- und Datenstrukturanalysen

1. `void insert(E e)`

Besonders effiziente Datenstruktur: Verkettete Liste (Linked List)

Warum: Eine verkettete Liste ermöglicht das schnelle und effiziente Einfügen von Elementen an jeder Position. Der Einfügevorgang hat eine Zeitkomplexität von O(1), da nur die Verbindungen an der Einfügestelle aktualisiert werden müssen.

Weniger geeignete Datenstruktur: Array

Warum: Das Einfügen eines Elements in ein Array dauert in der Regel O(n), da alle nachfolgenden Elemente verschoben werden müssen. Dies ist besonders ineffizient, wenn das Array voll ist.

2. `E accessMin()`

Besonders effiziente Datenstruktur: Heap

Warum: Eine Min-Heap-Datenstruktur ermöglicht den Zugriff auf das minimale Element in O(1)-Zeit. Das Einfügen und Entfernen von Elementen erfolgt in O(log n)-Zeit, was das Auffinden des minimalen Elements effizient macht.

Weniger geeignete Datenstruktur: Verkettete Liste (Linked List)

Warum: In einer verketteten Liste dauert das Finden des minimalen Elements O(n), da alle Elemente einzeln überprüft werden müssen. Dies ist bei großen Datensätzen ineffizient.

3. `boolean contains(E e)`

Besonders effiziente Datenstruktur: HashSet

Warum: Ein HashSet ermöglicht die Überprüfung der Existenz von Elementen in durchschnittlich O(1)-Zeit. Durch Hashing kann schnell überprüft werden, ob ein Element vorhanden ist.

Weniger geeignete Datenstruktur: Array

Warum: In einem Array dauert die Überprüfung, ob ein Element vorhanden ist, O(n), da das gesamte Array durchsucht werden muss.

4. `E successor(E e)`

Besonders effiziente Datenstruktur: Ausgeglichener binärer Suchbaum (Balanced Binary Search Tree, z.B. AVL-Baum oder Rot-Schwarz-Baum)

Warum: Ausgeglichene binäre Suchbäume ermöglichen das Auffinden des Nachfolgeelements (successor) in O(log n)-Zeit. Aufgrund der ausgeglichenen Baumstruktur kann das Nachfolgeelement schnell gefunden werden.

Weniger geeignete Datenstruktur: Verkettete Liste (Linked List)

Warum: In einer verketteten Liste dauert das Finden des Nachfolgeelements O(n), da die Liste von Anfang bis Ende durchsucht werden muss.

5. `E kthElement(int k)`

Besonders effiziente Datenstruktur: Ausgeglichener binärer Suchbaum (Balanced Binary Search Tree, z.B. AVL-Baum oder Rot-Schwarz-Baum)

Warum: Ausgeglichene binäre Suchbäume ermöglichen das Auffinden des k-ten Elements in O(log n)-Zeit. Die ausgeglichene Struktur des Baumes ermöglicht es, das k-te Element schnell zu finden.

Weniger geeignete Datenstruktur: Array

Warum: In einem Array dauert das Finden des k-ten Elements O(n), da das Array sequentiell durchsucht werden muss, abhängig von der Größe des Arrays.

Die obigen Analysen zeigen für jede Methode, welche Datenstruktur besonders effizient und welche weniger geeignet ist. Diese Informationen helfen uns, durch die Wahl der richtigen Datenstruktur die Leistung unserer Algorithmen zu optimieren.

Aufgabe 2 Regionales Bildungsprogramm

Das Ziel dieser Methode ist es, alle zusammenhängenden weißen Regionen in einem Schwarz-Weiß-Bild mit verschiedenen zufälligen Farben zu markieren. Dies soll dazu beitragen, jede Region visuell voneinander zu unterscheiden und die Verständlichkeit des Bildes zu verbessern.

Die Funktionsweise des Codes ist wie folgt: Zunächst wird ein ‘BufferedImage‘ namens ‘segmented‘ erstellt, das das segmentierte Bild repräsentiert, das am Ende erzeugt werden soll. Falls das ‘segmented‘ Bild noch nicht erstellt wurde (Zeile 156), werden die folgenden Schritte ausgeführt.

Dabei werden mit Hilfe der Klasse ‘UnionFind‘ alle horizontalen weißen Segmente (‘Run‘-Objekte) aus dem Bild extrahiert (Zeile 160). Für jedes ‘Run‘-Objekt wird seine zugehörige Wurzel (‘root‘) ermittelt (Zeile 163). Die Wurzel repräsentiert dabei das Hauptsegment, das die Farbe für das gesamte Segment bestimmt.

Für jedes ‘Run‘-Objekt wird eine Farbe festgelegt: - Wenn das ‘run‘-Objekt mit seiner Wurzel (‘root‘) übereinstimmt (Zeile 166), wird eine neue zufällige Farbe generiert (‘getRandomColor()‘). Diese Farbe wird der ‘root‘ zugewiesen und als die Farbe für dieses Segment festgelegt (Zeile 168). - Wenn das ‘run‘-Objekt nicht mit seiner Wurzel übereinstimmt (Zeile 169-171), wird die Farbe des ‘root‘ verwendet, um die Farbe des Segments festzulegen.

Abschließend werden die Pixel jedes ‘Run‘-Objekts mit der entsprechenden Farbe in das ‘segmented‘ Bild eingetragen, indem die Methode ‘uf.drawRunSegments()‘ verwendet wird (Zeile 173).

Am Ende wird das fertige ‘segmented‘ Bild zurückgegeben (Zeile 176). Diese Methode verwandelt ein ursprünglich Schwarz-Weiß-Bild in ein Bild, in dem jede Region mit einer einzigartigen Farbe markiert ist. Dies erleichtert die Analyse und Interpretation des Bildes erheblich.

```
148     /**
149      * Liefert ein segmentiertes Bild, bei dem alle zusammenhängenden, weißen Regionen
150      * des Schwarzweißbildes mit zufälligen Farben markiert sind. Das Bild wird erst bei
```

```
151      * Bedarf berechnet.
152      * @return Das segmentierte Bild.
153      */
154     BufferedImage getSegmented()
155     {
156         if (segmented == null) {
157                     final BufferedImage image = getBlackAndWhite();
158                     segmented = new BufferedImage(image.getWidth(), image.getHeight(),
                                BufferedImage.TYPE_INT_RGB);
159
160                     UnionFind uf = new UnionFind(image, WHITE);
161
162                     for (Run run : uf.getRuns()) {
163                         Run root = run.getRoot();
164
165                         int color;
166                         if (root == run) {
167                             color = getRandomColor();
168                             root.setColor(color);
169                         } else {
170                             color = root.getColor();
171                         }
172
173                         uf.drawRunSegments(run, segmented, color);
174                     }
175                 }
176                 return segmented;
177     }
```

Union-Find Class

Diese Klasse stellt die Union-Find-Datenstruktur für die Verarbeitung von Run-Segmenten dar. Sie ist so beschrieben, dass sie Aufgaben enthält.

```
1  package de.uni_bremen.pi2;
2
3  import java.awt.image.BufferedImage;
4  import java.util.ArrayList;
5
6  /**
7   * Die Union-Find-Datenstruktur zur Verarbeitung von Run-Segmenten in einem Bild.
8   */
9  public class UnionFind {
10
11     private final ArrayList<Run> runs = new ArrayList<>();
12
13     /**
14      * Gibt die Liste der Runs zurück.
15      *
16      * @return die Liste der Runs
17      */
18     public ArrayList<Run> getRuns() {
19         return runs;
20     }
21
22     /**
23      * Konstruktor, der Run-Segmente aus einem Bild erstellt und sie miteinander vereint.
24      * Erstellt zunächst horizontale Segmente, indem es weiße Pixel im Bild verfolgt und
25      * anschließend werden benachbarte Segmente miteinander vereint.
26      *
27      * @param image das zu verarbeitende Bild
28      * @param white der Farbwert für Weiß
29      */
30     public UnionFind(BufferedImage image, int white) {
31         createRunSegments(image, white);
32         unionRunSegments(this.runs);
33     }
34
35     /**
36      * Fügt einen Run zur Union-Find-Datenstruktur hinzu.
```

```
37       *
38       * @param run der hinzuzufügende Run
39       */
40      public void insert(Run run) {
41          this.runs.add(run);
42      }
43
44      /**
45       * Gibt den Run mit der kleinsten y-Koordinate zurück.
46       * Diese Methode durchläuft alle gespeicherten Runs und findet denjenigen mit der
                kleinsten y-Koordinate.
47       *
48       * @return der Run mit der kleinsten y-Koordinate
49       */
50      public Run accessMin() {
51          Run minRun = runs.get(0);
52          for (Run run : runs) {
53              if (run.getY() < minRun.getY()) {
54                  minRun = run;
55              }
56          }
57          return minRun;
58      }
59
60      /**
61       * Überprüft, ob ein Run in der Union-Find-Datenstruktur enthalten ist.
62       * Diese Methode durchsucht die Liste der gespeicherten Runs, um zu prüfen, ob der
                gegebene Run vorhanden ist.
63       *
64       * @param run der zu überprüfende Run
65       * @return true, wenn der Run enthalten ist, andernfalls false
66       */
67      public boolean contains(Run run) {
68          return runs.contains(run);
69      }
70
71      /**
72       * Findet den Nachfolger eines Runs in der Union-Find-Datenstruktur.
73       * Diese Methode sucht in der Liste der Runs nach einem Run, der eine höhere y-
                Koordinate hat als der gegebene Run.
74       *
75       * @param run der Run, dessen Nachfolger gefunden werden soll
76       * @return der Nachfolger des Runs, oder null wenn kein Nachfolger gefunden wird
77       */
78      public Run successor(Run run) {
79          for (Run nextRun : runs) {
80              if (nextRun.getY() > run.getY()) {
81                  return nextRun;
82              }
83          }
84          return null;
85      }
86
87      /**
88       * Gibt den k-ten Run in der Union-Find-Datenstruktur zurück.
89       * Diese Methode gibt den Run am Index k in der Liste der gespeicherten Runs zurück.
90       *
91       * @param k der Index des zurückzugebenden Runs
92       * @return der k-te Run
93       * @throws IndexOutOfBoundsException wenn der Index außerhalb des Bereichs liegt
94       */
95      public Run kthElement(int k) {
96          if (k < 0 || k >= runs.size()) {
97              throw new IndexOutOfBoundsException();
98          }
99          return runs.get(k);
100     }
```

Aufgabe 2.1 Horizontal zusammenfassen

Diese Methode erstellt horizontale Segmente, indem sie weiße Pixel im Bild verfolgt. Sie durchläuft das Bild zeilenweise und erstellt für zusammenhängende weiße Pixel Run-Objekte.

Die äußere Schleife (Zeile 110) iteriert über jede Zeile des Bildes (‘image.getHeight()‘), während die innere Schleife (Zeile 115) jede Spalte durchläuft (‘image.getWidth()‘).

Zu Beginn jeder Zeile werden die Variablen ‘xStart‘, ‘xEnd‘ und ‘isContinue‘ initialisiert. ‘xStart‘ und ‘xEnd‘ markieren die Anfangs- und Endpunkte eines horizontalen Segments, während ‘isContinue‘ den Zustand speichert, ob gerade ein Segment verfolgt wird.

Innerhalb der inneren Schleife wird überprüft, ob das Pixel weiß ist (‘image.getRGB(x, y) == WHITE‘) und ob gerade kein Segment verfolgt wird (‘!isContinue‘). Wenn dies der Fall ist, wird ‘xStart‘ auf die aktuelle x-Position gesetzt und ‘isContinue‘ auf ‘true‘ gesetzt, um anzugeben, dass ein neues Segment begonnen hat.

Wenn das Pixel nicht weiß ist (‘image.getRGB(x, y) != WHITE‘) und ‘isContinue‘ gleichzeitig ‘true‘ ist, bedeutet dies, dass das Ende eines Segments erreicht wurde. In diesem Fall wird ‘xEnd‘ auf die aktuelle x-Position gesetzt, ‘isContinue‘ auf ‘false‘ gesetzt, und ein neues Run-Objekt (‘Run(xStart, xEnd, y)‘) wird durch den Aufruf der ‘insert‘-Methode erstellt und hinzugefügt.

Nachdem die innere Schleife die gesamte Zeile durchlaufen hat, wird überprüft, ob das Segment bis zum Ende der Zeile (‘image.getWidth()‘) fortgesetzt wurde (‘isContinue == true‘). Falls ja, wird ein letztes Run-Objekt erstellt, das bis zum Ende der Zeile reicht.

Zusammengefasst dient diese Methode dazu, alle horizontalen weißen Segmente im Bild zu identifizieren und jeweils ein Run-Objekt für jedes dieser Segmente zu erstellen.

```
102     /**
103      * Erstellt horizontale Segmente, indem es weiße Pixel im Bild verfolgt.
104      * Diese Methode durchläuft das Bild zeilenweise und erstellt Run-Objekte für
                zusammenhängende weiße Pixel.
105      *
106      * @param image das zu verarbeitende Bild
107      * @param WHITE der Farbwert für Weiß
108      */
109     void createRunSegments(BufferedImage image, int WHITE) {
110         for (int y = 0; y < image.getHeight(); y++) {
111             int xStart = -1;
112             int xEnd = -1;
113             boolean isContinue = false;
114 
115             for (int x = 0; x < image.getWidth(); x++) {
116                 if (image.getRGB(x, y) == WHITE && !isContinue) {
117                     xStart = x;
118                     isContinue = true;
119                 } else if (image.getRGB(x, y) != WHITE && isContinue) {
120                     xEnd = x;
121                     isContinue = false;
122                     insert(new Run(xStart, xEnd, y));
123                 }
124             }
125             if (isContinue)
126                 insert(new Run(xStart, image.getWidth(), y));
127         }
128     }
```

Aufgabe 2.2 Regionen repräsentieren

Dieser Code definiert die Klasse ‘Run‘, die ein Segment in einer Union-Find-Datenstruktur darstellt. Die Klasse enthält die Start- (‘xStart‘) und End- (‘xEnd‘) x-Koordinaten, die y-Koordinate (‘y‘), das Wurzel-Segment (‘root‘) und die Farbe des Segments (‘color‘). Der Konstruktor der

Klasse initialisiert diese Werte und setzt die Wurzel auf sich selbst. Methoden wie ‘getXStart‘, ‘getXEnd‘, ‘getY‘, ‘getRoot‘, ‘getLength‘, ‘getColor‘ und ‘setColor‘ werden verwendet, um die Eigenschaften des Segments abzurufen oder zu setzen. Die Methode ‘toString‘ gibt eine Zeichenkette zurück, die die Koordinaten des Segments und seines Wurzel-Segments enthält.

```java
package de.uni_bremen.pi2;

/**
 * Repräsentiert einen Lauf in der Union-Find-Datenstruktur.
 */
public class Run {

    /**
     * Die Start-x-Koordinate des Laufs.
     */
    private final int xStart;

    /**
     * Die End-x-Koordinate des Laufs.
     */
    private final int xEnd;

    /**
     * Die y-Koordinate des Laufs.
     */
    private final int y;

    /**
     * Der Oberlauf dieses Laufs.
     */
    private Run root;

    /**
     * Erstellt einen neuen Lauf mit den angegebenen x-Start-, x-End- und y-Koordinaten.
     *
     * @param xStart die Start-x-Koordinate des Laufs
     * @param xEnd   die End-x-Koordinate des Laufs
     * @param y      die y-Koordinate des Laufs
     */
    public Run(int xStart, int xEnd, int y) {
        this.xStart = xStart;
        this.xEnd = xEnd;
        this.y = y;
        this.root = this;
    }

    /**
     * Gibt die Start-x-Koordinate des Laufs zurück.
     *
     * @return die Start-x-Koordinate des Laufs
     */
    public int getXStart() {
        return xStart;
    }

    /**
     * Gibt die End-x-Koordinate des Laufs zurück.
     *
     * @return die End-x-Koordinate des Laufs
     */
    public int getXEnd() {
        return xEnd;
    }

    /**
     * Gibt die y-Koordinate des Laufs zurück.
     *
     * @return die y-Koordinate des Laufs
     */
    public int getY() {
        return y;
    }

```

```
69      /**
70       * Gibt den Wurzel-Lauf dieses Laufs zurück.
71       *
72       * @return der Wurzel-Lauf dieses Laufs
73       */
74      public Run getRoot() {
75          if (root != this) {
76              root = root.getRoot(); // Pfadkomprimierung
77          }
78          return root;
79      }
80
81      /**
82       * Setzt den Oberlauf dieses Laufs.
83       *
84       * @param root der neue Oberlauf
85       */
86      public void setRoot(Run root) {
87          this.root = root;
88      }
89
90      @Override
91      public String toString() {
92          return "(" + "Lauf: " + xStart + "-" + xEnd + "-" + y + " --> Wurzel: " + root.
                    xStart + "-" + root.xEnd + "-" + root.y + ")";
93      }
94
95      /**
96       * Gibt die Länge des Laufs zurück.
97       *
98       * @return die Länge des Laufs
99       */
100     public int getLength() {
101         return xEnd - xStart;
102     }
103
104     private int color;
105     /**
106      * Gibt die Farbe des Laufs zurück.
107      *
108      * @return die Farbe des Laufs
109      */
110     public int getColor() {
111         return color;
112     }
113
114     /**
115      * Setzt die Farbe des Laufs.
116      *
117      * @param color die neue Farbe des Laufs
118      */
119     public void setColor(int color) {
120         this.color = color;
121     }
122 }
```

Aufgabe 2.3 Regionen vereinigen

Dieser Code enthält Methoden zur Überprüfung, ob zwei Segmente Nachbarn sind, und zum Vereinen von zwei Segmenten. Die Methode ‘isNeighbors‘ überprüft, ob die y-Koordinaten von zwei Segmenten aufeinander folgen und ob sich ihre x-Koordinaten überschneiden. Die Methode ‘union‘ vereinigt zwei Segmente in der Union-Find-Datenstruktur. Wenn zwei Segmente bereits denselben Wurzel-Segment haben, wird keine Vereinigung durchgeführt. Andernfalls wird das Wurzel-Segment basierend auf der y-Koordinate festgelegt, und das Wurzel-Segment eines Segments wird als Wurzel des anderen Segments gesetzt.

```
153     /**
154      * Überprüft, ob zwei Segmente Nachbarn sind.
```

```
155      * Diese Methode prüft, ob zwei Segmente benachbarte y-Koordinaten haben und ob sich
               ihre x-Koordinaten überschneiden.
156      *
157      * @param runi das erste Segment
158      * @param runj das zweite Segment
159      * @return true, wenn die beiden Segmente Nachbarn sind, andernfalls false
160      */
161     public boolean isNeighbors(Run runj, Run runi) {
162         return runj.getY() + 1 == runi.getY() && runj.getXStart() < runi.getXEnd() &&
                    runi.getXStart() < runj.getXEnd();
163     }
164
165     /**
166      * Vereint zwei Segmente.
167      * Diese Methode führt die Union-Operation durch, um zwei Segmente in der Union-Find-
               Datenstruktur zu vereinen.
168      * Dabei wird das Wurzel-Element des einen Segments als Wurzel des anderen Segments
               gesetzt.
169      *
170      * @param runi das erste Segment
171      * @param runj das zweite Segment
172      */
173     public void union(Run runi, Run runj) {
174         Run rooti = runi.getRoot();
175         Run rootj = runj.getRoot();
176
177         if (rooti == rootj) {
178             return;
179         }
180
181         if (rooti.getY() <= rootj.getY()) {
182             rootj.setRoot(rooti);
183         } else {
184             rooti.setRoot(rootj);
185         }
186     }
```

Aufgabe 2.4 Regionenbildung

Dieser Code vereinigt benachbarte Segmente in einer gegebenen Liste von Run-Objekten. Die Methode ‘unionRunSegments‘ überprüft, ob zwei Segmente Nachbarn sind (mittels der Methode ‘isNeighbors‘), und wenn ja, führt sie eine Union-Operation durch, um diese Segmente zu vereinigen (mittels der Methode ‘union‘). Die äußere Schleife (Index ‘j‘) durchläuft alle Segmente, und die innere Schleife (Index ‘i‘) überprüft die Nachbarn jedes Segments. Wenn zwei Segmente keine Nachbarn sind, wird die Schleife fortgesetzt und ‘j‘ geht zum nächsten Segment über. Wenn benachbarte Segmente vereinigt werden, wird der Index ‘i‘ erhöht.

```
130     /**
131      * Vereint benachbarte Segmente in der gegebenen Liste von Run-Objekten.
132      * Diese Methode überprüft, ob zwei Segmente benachbart sind (d.h. ihre y-Koordinaten
               sind benachbart und ihre x-Koordinaten überschneiden sich),
133      * und führt eine Union-Operation durch, um diese Segmente zu vereinen.
134      *
135      * @param runs die Liste der zu vereinigenden Run-Objekte
136      */
137     public void unionRunSegments(ArrayList<Run> runs) {
138         int j = 0;
139         int i = 1;
140         while (j < runs.size()) {
141             if (i < runs.size() && isNeighbors(runs.get(j), runs.get(i))) {
142                 union(runs.get(j), runs.get(i));
143                 i++;
144             } else if (i < runs.size() && (runs.get(j).getY() <= runs.get(i).getY() ||
                        runs.get(j).getXEnd() < runs.get(i).getXEnd())) {
145                 i++;
146             } else {
147                 j++;
148                 i = j;
```

```
149             }
150         }
151     }
```

Aufgabe 2.5 Einfärben

Dieser Code zeichnet ein Segment auf ein Bild. Die Methode ‘drawRunSegments‘ zeichnet die x-Koordinaten des gegebenen Segments an der angegebenen y-Koordinate in der angegebenen Farbe auf das Bild. Die Methode nimmt die Start- und End-x-Koordinaten des Segments und malt alle Pixelpunkte zwischen diesen Koordinaten in der festgelegten Farbe.

```
188     /**
189      * Zeichnet ein Segment auf ein Bild.
190      * Diese Methode zeichnet die x-Koordinaten des gegebenen Segments auf der
               angegebenen y-Koordinate in der angegebenen Farbe auf das Bild.
191      *
192      * @param run       das zu zeichnende Segment
193      * @param segmented das Bild, auf das gezeichnet wird
194      * @param color     die Farbe zum Zeichnen
195      */
196     public void drawRunSegments(Run run, BufferedImage segmented, int color) {
197         int xstart = run.getXStart();
198         int xend = run.getXEnd();
199         int y = run.getY();
200 
201         for (int x = xstart; x < xend; x++) {
202             segmented.setRGB(x, y, color);
203         }
204     }
205 }
```