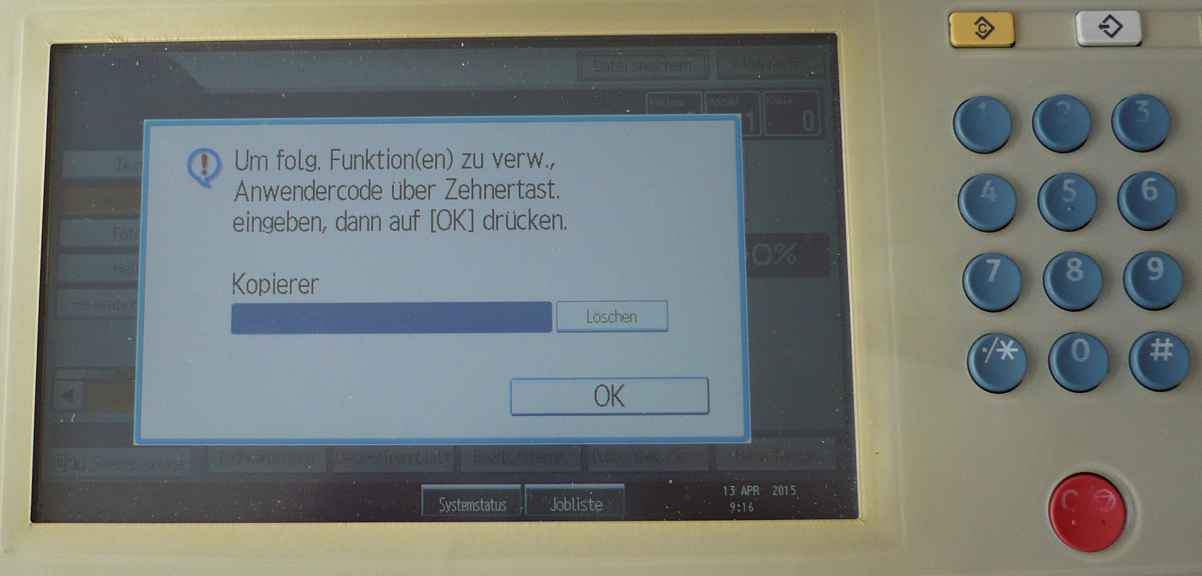
Sortieren auf linearen Datenstrukturen

Benutzerverwaltung für einen Fotokopierer



Die Softwareabteilung eines Herstellers von Fotokopierer stellt verschiedene Lösungen für die Benutzerverwaltung vor. Beim Einschalten des Kopierers werden die Benutzerdaten in einen Arbeitsspeicher geladen.

Benutzer des Kopierers geben eine vierstellige Ziffernkombination an, um sich zu legitimieren. Über diesen PIN-Code soll auch der Benutzer des Kopierers identifizierbar sein, also darf jede PIN nur einmal vergeben werden.

Gesucht wird eine Organisation der Benutzerdaten, die die Benutzerdaten nach verschiedenen Aspekten sortiert darstellt. Exemplarisch sollen hier auch drei verschiedene Sortierstrategien dargestellt werden: Sortieren durch Auswahl (Straight Selection), Sortieren durch Einfügen (Straight Insertion) und Quicksort. Die Sortierstrategien werden sowohl auf Arrays als auch auf Listenstrukturen angewandt.

Zur Bedeutung des Sortierens:

„Ein weiteres grundlegendes Problem in der Informatik neben dem Suchen ist das Sortieren. So entfällt ca. ¼ der kommerziell verbrauchten Rechenzeit auf Sortiervorgänge. Die Aufgabe besteht hierbei im Ordnen von Dateien mit Datensätzen, die Schlüssel enthalten. Die Datensätze sind derart umzuordnen, dass eine klar definierte Ordnung der Schlüssel – entweder numerisch oder auch lexigraphisch- entsteht.“[[1]](#footnote-1)

Entwurf einer Testumgebung der Benutzerverwaltung:

151009_Entwurfsdia_Sort

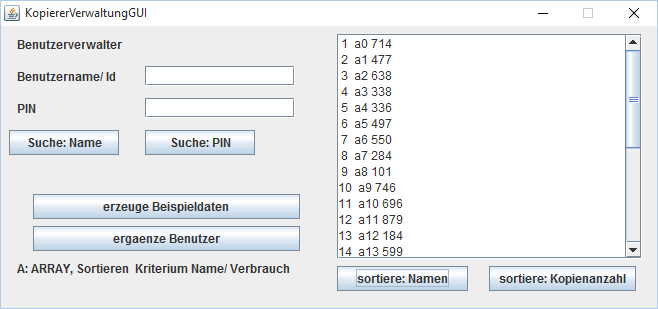
Dokumentation

Klasse **BenutzerVerwalter**

Ein Objekt der Klasse **BenutzerVerwalter** verwaltet die Daten aller Kopiererbenutzer.

|  |  |
| --- | --- |
| Auftrag | **void neuerBenutzer(pName: Text)**  Ein neuer Benutzer wird der Datensammlung hinzugefügt. Die PIN wird zufällig ermittelt. |
| Anfrage | **sortiereVerbrauch()**  Die Objekte der Klasse Benutzer werden in der Datensammlung nach dem Verbrauch an Kopien absteigend sortiert, d.d. der Benutzer mit dem größten Kopienverbrauch steht an erster Stelle. |
| Anfrage | **sortiereNamen()**  Die Objekte der Klasse Benutzer werden in der Datensammlung nach den Namen lexiographisch sortiert. |
| Anfrage | **sortierePIN()**  Die Objekte der Klasse Benutzer werden in der Datensammlung nach der PIN aufsteigend sortiert. |

Die Testumgebung hat die folgende Benutzungsoberfläche.



Benutzerverwaltung in einem ARRAY

Variante A

In einem ARRAY befinden sich die vergebenen zufälligen PIN-Nummern und die Benutzerdaten, die aufsteigend nach den PIN-Nummern sortiert sind.

Benutzerverwaltung in einer Liste

Variante L

In einer Liste befinden sich die vergebenen zufälligen PIN-Nummern und die Benutzerdaten, die aufsteigend nach den PIN-Nummern sortiert sind.

Benutzer

In einem Objekt der Klasse **Benutzer** werden folgende Daten des Benutzers verwaltet.

|  |  |
| --- | --- |
| **name** | Name oder ID des Benutzers. Der Name ist eindeutig. |
| **pin** | Die vierstellige Ziffernkombination, über die der Benutzer identifiziert werden kann und mit der sich der Benutzer am Kopierer anmelden kann. |
| **limit** | Die Zahl beschreibt die Anzahl der Kopien, die der Benutzer erstellen darf. |
| **verbrauch** | Die Zahl beschreibt die Anzahl der Kopien, die der Benutzer bereits erstellt hat. |

Dokumentation

Klasse **Benutzer**

Ein Objekt der Klasse **Benutzer** verwaltet die Daten eines Kopererbenutzers.

|  |  |
| --- | --- |
| Konstruktor | **Benutzer(String pName, String pPIN)**  Ein Benutzer wird angelegt. Als Parameter werden der eindeutige Benutzername und die zugehörige PIN festgelegt. |
| Auftrag | **void erhoeheVerbrauch(int pZahl)**  Der Verbrauch des Benutzers wird um **pZahl** erhöht. |

Es gibt keinen informatischen Grund, dass die Benutzerdatensätze nach unterschiedlichen Strategien sortiert werden. Die Sortierung in den folgenden Fällen könnte auch immer nach derselben Strategie erfolgen.

Straight Selection

Man bestimme diejenige Position, an der ein Benutzerobjekt mit dem höchsten Verbrauch an Kopien auftritt. Dieses Objekt wird dann mit dem ersten Objekt getauscht. Dann bestimme man das Benutzerobjekt ab der zweiten Position mit dem höchsten Verbrauch. Anschließend wird dieses Objekt mit dem zweiten Benutzerobjekt getauscht. Dieses Verfahren wird dann für alle folgenden Anfangspositionen fortgesetzt.

Darstellung der Sortierstrategie für ein Zahlenbeispiel. Dargestellt wird eine absteigende Sortierung.

151011_ASort1

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13** | **public void sortiereVerbrauch() {**  **// Straight Selection**  **for (int i = 0; i < aktBenutzerzahl; i++) {**  **int max = i;**  **for (int j = i+1; j < aktBenutzerzahl; j++){**  **if (benutzerDaten[j].gibAnzahl() >**  **benutzerDaten[max].gibAnzahl()) {**  **max = j;**  **}**  **Benutzer temp = benutzerDaten[i];**  **benutzerDaten[i] = benutzerDaten[max];**  **benutzerDaten[max] = temp;**  **}**  **}**  **}** |

Straight Insertion

Die Vorgehensweise kann wie folgt angegeben werden.

Starte mit dem ersten Element der sortierten Folge.

Nimm jeweils das nächste Element der originalfolge und füge es an die richtige Stelle in der sortierten Folge ein.

Angewendet auf eine Folge von Elementen der Länge n bedeutet dieses, dass ein Element zu „merken“ ist (hier aktuell). Dadurch wird eine Position in der Folge frei, die genutzt werden kann, um alle Elemente, die größer als das gemerkte Element sind, eine Position nach rechts zu verschieben. Das Verschieben erfolgt ausgehend von der aktuellen Position rückwärts bis zum ersten Element. Durch die Verschiebung nach rechts wird die Position in der Folge frei, an der das gemerkte Element eingefügt werden muss.

Darstellung der Sortierstrategie für ein Zahlenbeispiel. Dargestellt wird eine aufsteigende Sortierung.

151011_ASort2

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11** | **public void sortiereNamen() {**  **for (int i = 1; i < aktBenutzerzahl; i++) {**  **Benutzer aktuell = benutzerDaten[i];**  **int j = i;**  **while (j > 0**  **&& benutzerDaten[j-1].gibName().compareTo(aktuell.gibName()) > 0) {**  **benutzerDaten[j] = benutzerDaten[j - 1];**  **j--;**  **}**  **benutzerDaten[j] = aktuell;**  **}**  **}** |

Quicksort

Die Grundidee von Quicksort ist die Zerlegung der Folge in zwei Teile, wobei alle Elemente der einen Teilfolge kleiner als ein Referenzelement (das so genannte Pivot-Element) und alle Elemente der anderen Teilfolge größer als das Referenzelement sind. Das Pivot-Element kann beliebig gewählt werden. Typischerweise wird das mittlere Element der Folge verwendet, es kann aber auch das erste oder das letzte Element genutzt werden. Die beiden Teilfolgen werden durch den rekursiven Aufruf sortiert. Das Verfahren bricht für eine Teilfolge ab, wenn diese die Länge 0 hat.

151012_ASort3

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24** | **public void sortierePIN() {**  **// Quicksort**  **int n = aktBenutzerzahl;**  **quicksort(0, n-1);**  **}**  **private void quicksort (int lo, int hi) {**  **int i = lo, j = hi;**  **// Vergleichselement pivot (PIN)**  **String pivot = benutzerDaten[lo+(hi-lo)/2].gibPIN();**  **// Aufteilung**  **while (i <= j) {**  **while (benutzerDaten[i].gibPIN().compareTo(pivot) < 0) { i++; }**  **while (benutzerDaten[j].gibPIN().compareTo(pivot) > 0) { j--; }**  **if (i <= j) {**  **tausche(i, j);**  **i++; j--;**  **}**  **}**  **// Rekursion**  **if (lo < j) { quicksort(lo, j); }**  **if (i < hi) { quicksort(i, hi); }**  **}**  **private void tausche(int i, int j) {**  **Benutzer b = benutzerDaten[i];**  **benutzerDaten[i] = benutzerDaten[j];**  **benutzerDaten[j] = b;**  **}** |

Straight Selection

Durchlaufe die Liste und bestimme das Objekt mit dem größten Schlüsselwert. Schreibe das Objekt in eine Hilfsliste und lösche es aus der Originalliste. Durchlaufe anschließend die verkleinerte Originalliste und bestimme das Objekt mit dem größten Schlüsselwert. Füge das Objekt in die Hilfsliste hinten ein. Wende dieses Verfahren solange an, bis die verkleinerte Originalliste leer ist. Nun steht in der Hilfsliste die sortierte Liste. Diese sortierte Hilfsliste kann nun der Originalliste zugewiesen werden.

151012_LSort1

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24** | **public void sortiereVerbrauch() {**  **// Straight Selection, Anzahl der Kopien**  **List<Benutzer> hilfsliste = new List<Benutzer>();**  **while (!benutzerDaten.isEmpty()) {**  **benutzerDaten.toFirst();**  **Benutzer max = benutzerDaten.getContent();**  **while (benutzerDaten.hasAccess()) {**  **Benutzer aktueller = benutzerDaten.getContent();**  **if (aktueller.gibAnzahl() > max.gibAnzahl()) {**  **max = aktueller;**  **}**  **benutzerDaten.next();**  **}**  **benutzerDaten.toFirst();**  **while (benutzerDaten.hasAccess()) {**  **if (max == benutzerDaten.getContent()) {**  **benutzerDaten.remove();**  **}**  **benutzerDaten.next();**  **}**  **hilfsliste.append(max);**  **}**  **benutzerDaten.concat(hilfsliste);**  **}** |

Straight Insertion

151012_LSort2

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23** | **public void sortiereNamen() {**  **List<Benutzer> hilfsliste = new List<Benutzer>();**  **benutzerDaten.toFirst();**  **while (!benutzerDaten.isEmpty()) {**  **Benutzer aktueller = benutzerDaten.getContent();**  **benutzerDaten.remove();**  **hilfsliste.toFirst();**  **boolean gefunden = false;**  **while (!gefunden && hilfsliste.hasAccess()) {**  **if (aktueller.gibName().**  **compareTo(hilfsliste.getContent().gibName()) < 0) {**  **gefunden = true;**  **} else {**  **hilfsliste.next();**  **}**  **}**  **if (hilfsliste.hasAccess()) {**  **hilfsliste.insert(aktueller);**  **} else {**  **hilfsliste.append(aktueller);**  **}**  **}**  **benutzerDaten.concat(hilfsliste);**  **}** |

Quicksort

Die Grundidee von Quicksort ist die Zerlegung der Folge in zwei Teile, wobei alle Elemente der einen Teilfolge kleiner als ein Referenzelement (das so genannte Pivot-Element) und alle Elemente der anderen Teilfolge größer als das Referenzelement sind. Das Pivot-Element kann beliebig gewählt werden. Hier wird das erste Listenobjekt als Pivot-Element gewählt.

151012_LSort3

|  |  |
| --- | --- |
| **1**  **2**  **3**  **4**  **5**  **6**  **7**  **8**  **9**  **10**  **11**  **12**  **13**  **14**  **15**  **16**  **17**  **18**  **19**  **20**  **21**  **22**  **23**  **24**  **25**  **26**  **27**  **28** | **public void sortierePIN() {**  **benutzerDaten = quicksort(benutzerDaten);**  **}**  **private List<Benutzer> quicksort(List<Benutzer> pListe) {**  **List<Benutzer> ergebnis = new List<Benutzer>();**  **if (!pListe.isEmpty()) {**  **pListe.toFirst();**  **Benutzer pivot = pListe.getContent();**  **pListe.remove();**  **List<Benutzer> kleiner = new List<Benutzer>();**  **List<Benutzer> groesser = new List<Benutzer>();**  **while (!pListe.isEmpty()) {**  **Benutzer aktuelles = pListe.getContent();**  **if (aktuelles.gibPIN().compareTo(pivot.gibPIN()) < 0) {**  **kleiner.append(aktuelles);**  **} else {**  **groesser.append(aktuelles);**  **}**  **pListe.remove();**  **}**  **kleiner = quicksort(kleiner);**  **groesser = quicksort(groesser);**  **ergebnis = kleiner;**  **ergebnis.append(pivot);**  **ergebnis.concat(groesser);**  **}**  **return ergebnis;**  **}** |

Vergleich der Sortierverfahren

Der Laufzeitaufwand eines Sortierverfahrens wird im Wesentlichen durch die Anzahl der Vergleiche und die Anzahl der Vertauschungen bestimmt.

Übung 1:

Ergänzen Sie die Quelltexte durch Zähler für Vergleiche und Zähler für Implementationen. Führen Sie Testreihen durch, bei denen Sie die Anzahl der zu sortierenden Elemente variieren. Halten Sie die Ergebnisse in einem Tabellenkalkulationsprogramm fest und stellen Sie die Ergebnisse graphisch dar.

Ein weiteres Merkmal von Sortierverfahren ist die Stabilität. Ein Verfahren heißt stabil, wenn es die relative Reihenfolge gleicher Schlüssel in der Datei beibehält.[[2]](#footnote-2)

Übung 2:

Überprüfen Sie, ob es sich bei dem Sortierverfahren um ein stabiles Verfahren handelt.

Eigenschaften von Sortierverfahren[[3]](#footnote-3)

|  |  |  |
| --- | --- | --- |
| *Verfahren* | *Stabilität* | *Vergleiche (im Mittel)* |
| Selection Sort | instabil | **≈ n2/2** |
| Insertion Sort | stabil | **≈ n2/4** |
| QuickSort | instabil | **≈ n log2n** |

Quellen

Ottmann, Thomas; Widmayer, Peter

Algorithmen und Datenstrukturen

Heidelberg 2002 (4. Auflage): Spektrum Akademischer Verlag

Saake, Gunter; Sattler, Kai-Uwe

Algorithmen und Datenstrukturen

Heidelberg 2010 (4. Überarbeitete Auflage): dpunkt.verlag

1. Saake/ Sattler S. 124 [↑](#footnote-ref-1)
2. Saake / Sattler S. 124 [↑](#footnote-ref-2)
3. Saake / Sattler S. 139 [↑](#footnote-ref-3)