AuD - Zusammenfassung

Moritz Gerhardt

Contents

Sektion 1 Was ist ein Algorithmus?

Ein Algorithmus beschreibt eine Handlungsvorschrift zur Umwandlung von Eingaben in eine Ausgabe.
Dabei sollte ein Algorithmus im allgemeinen folgende Vorraussetzungen erfüllen:

1. Bestimmt:
   - Determiniert: Bei gleicher Eingabe liefert der Algortihmus gleiche Ausgabe.
     $\Longrightarrow$ Ausgabe nur von Eingabe abhängig, keine äußeren Faktoren.
   - Determinismus: Bei gleicher Eingabe läuft der Algorithmus immer gleich durch die Eingabe.
     $\Longrightarrow$ Gleiche Schritte, Gleiche Zwischenstände.
2. Berechenbar:
   - Finit: Der Algorithmus ist als endlich definiert. (Theoretisch)
   - Terminierbar: Der Algorithmus stoppt in endlicher Zeit. (Praktisch)
   - Effektiv: Der Algorithmus ist auf Maschine ausführbar.
3. Andwendbar:
   - Allgemein: Der Algorithmus ist für alle Eingaben einer Klasse anwendbar, nicht nur für speziellen Fall.
   - Korrekt: Wenn der Algorithmus ohne Fehler terminiert, ist die Ausgabe korrekt.

Sektion 2 Sortieren

2.1 Sortierproblem

Sortieralgorithmen sind die wohl am häufigsten verwendeten Algorithmen. Hierbei wird als Eingabe eine Folge von Objekten gegeben, die nach einer bestimmten Eigenschaft sortiert werden. Der Algorithmus soll die Eingabe in der richtigen Reihenfolge (nach einer bestimmten Eigenschaft) zur Ausgabe umwandeln. Es wird hierbei meist von einer total geordneten Menge ausgegangen. (Alle Elemente sind miteinander vergleichbar).
Eine Totale Ordnung wie folgt definiert:

Eine Relation $\leq$ auf M ist eine totale Ordnung, wenn:

- Reflexiv: $\forall x \in M : x \leq x$
  (x steht in Relation zu x)
- Transitiv: $\forall x, y, z \in M : x \leq y \wedge y \leq z \implies x \leq z$
  (Wenn x in Relation zu y steht und y in Relation zu z steht, so folgt, dass x in Relation zu z steht)
- Antisymmetrisch: $\forall x, y \in M : x \leq y \wedge y \leq x \implies x = y$
  (Wenn x in Relation zu y steht und y in Relation zu x steht, so folgt, dass x = y)
- Totalität: $\forall x, y \in M : x \leq y \vee y \leq x$
  (Alle Elemente müssen in einer Relation zueinander stehen)

2.2 Insertion Sort

```java
class InsertionSort {
    void insertionSort(int[] arr) {
        for (int i = 1; i < arr.length; i++) {
            // 1 to n - 1
            int key = arr[i];
            int j = i - 1;
            while (j >= 0 && arr[j] > key) {
                // Loops backwards through the array starting at i - 1
                // until it finds an element that is greater than the key or the beginning of the array
                arr[j + 1] = arr[j];
                // Shifts the element to the right
                j--;
            }
            arr[j + 1] = key;
            // Assigns the key to the correct position
        }
    }
}
```

Prinzip: Die Eingabe wird von links nach rechts durchlaufen. Dafür wird für jedes Element

2.3 Merge Sort

```java
class MergeSort {
    void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            // left < right, otherwise the region has no elements
            int mid = (left + right) / 2;
            // Split the region into two halves and do the recursive calls
            mergeSort(arr, left, mid);
            mergeSort(arr, mid + 1, right);
            // Merge the two (now sorted) halves
            merge(arr, left, mid, right);
        }
    }

    private void merge(int[] arr, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
        // Create a temporary array to store the merged elements

        int p = left;
        int q = mid + 1;
        for (int i = 0; i < right - left + 1; i++) {
            // Loops for each element in the region
            if (q > right || (p <= mid && arr[p] <= arr[q])) {
                // If p > mid the left half is finished, therefore the element needs to be in right half
                // Otherwise p needs to be <= mid and the element at p needs to be <= the element at q
                temp[i] = arr[p];
                p++;
                // Adds the element at p to the temporary array and increases p
            }
            else {
                temp[i] = arr[q];
                q++;
                // Adds the element at q to the temporary array and increases q
            }
        }
        // Copy the merged elements from the temporary array back to the original array
        for (int i = 0; i < right - left + 1; i++) {
            arr[left + i] = temp[i];
            // left + 0 is the start of the region
        }
    }
}
```

2.4 Quicksort

```
class Quicksort {
    void quickSort(int[] arr, int left, int right) {
        if (left < right) {
            // Region contains more than one element
            int part = partition(arr, left, right);
            quickSort(arr, left, part);
            quickSort(arr, part + 1, right);
        }
    }

    private int partition(int[] arr, int left, int right) {
        int pivot = arr[left];

        int p = left - 1;
        int q = right + 1;
        while (p < q) {
            while (arr[p] < pivot) {
                p++;
            }
            while (arr[q] > pivot) {
                q--;
            }
            // Increase / decrease p and q until they are equal to pivot
            if (p < q) {
                int temp = arr[p];
                arr[p] = arr[q];
                arr[q] = temp;
                // Swap arr[p] and arr[q]
            }
        }
        return q;
    }
}
```

2.5 Radix Sort

```java
import java.util.ArrayList;

class RadixSort {
    int D = 10; // possible unique digits
    int d; // Max amount of digits
    ArrayList<Integer>[] buckets = new ArrayList[D];

    void radixSort(int[] arr) {
        d = amountDigits(arr);
        for (int i = 0; i < d; i++) {
            // for each digit in the array, 0 least significant
            for (int j = 0; j < arr.length; j++) {
                putBucket(arr, i, j);
            }
            // Sorts the numbers into their buckets
            int a = 0;
            for (int k = 0; k < D; k++) {
                for (int b = 0; b < buckets[k].size(); b++) {
                    arr[a] = buckets[k].get(b);
                    a++;
                }
                buckets[k].clear();
            }
            // Reads out the buckets in order
        }
    }

    private void putBucket(int[] arr, int i, int j) {
        int z = arr[j] / (int) Math.pow(D, i) % D;
        // Gets the ith digit of the number
        int b = buckets[z].size();
        // size is next free index
        buckets[z].add(b, arr[j]);
        // puts the number in the bucket z at position b
        // Depending on implementation might need to increase size manually
    }

    private int amountDigits(int[] arr) {
        int max = arr[0];
        for (int i = 1; i < arr.length; i++) {
            if (arr[i] > max) {
                max = arr[i];
            }
        }
        // Get the biggest number
        return (int) Math.log10(max) + 1;
        // Get the amount of digits of the number
    }
}
```

Sektion 3 Grundlegende Datenstrukturen

3.1 Stacks

Stacks operieren unter dem ”First in - Last out” (FILO) Prinzip. Ähnlich zu einem Kartendeck, wo die unterste (Erste Karte) die ist, die als letztes gezogen wird.
Stacks werden normalerweise mit den folgenden Funktionen erstellt:

- `new`: Erstellt einen neuen Stack.
- `isEmpty`: gibt an ob der Stack leer ist.
- `pop`: gibt das oberste Element des Stacks zurück und enfernt es vom Stack.
- `push(k)`: Fügt `k` auf den Stack hinzu

Eine mögliche Implementation auf Grundlage eines Arrays wäre: Push und Pop schmeißen Fehlermeldung wenn Stack

```
1  Class Stack:
2  |  arr = null
3  |  top = -1
4  |  Function new(n):
5  |  |  arr = new Array[n]
6  |  |  return this
7  |  Function isEmpty:
8  |  |  return top == -1
9  |  Function pop:
10 |  |  return arr[top−]
11 |  Function push(k):
12 |  |  arr[++top] = k
```

leer bzw. voll ist. Oft als Stack underflow und Stack overflow benannt. Hier wär es automatisch IndexOutOfBounds. Oft werden Stacks auch mit variabler Größer implementiert. Dies kann über verschiedene Wege passieren, zum Beispiel Kopieren des arrays in einen größeren Array oder implementation über mehrere Arrays (z.B. über Linked List). Häufig wird das erstere so implementiert, dass der Array in einen Array mit doppelter Größer kopiert wird.

3.2 Linked List

Eine einfache Linked List besteht aus mehreren Elementen, die jeweils immer einen Wert und eine Referenz auf das nächste Element in der Liste haben. Eine einfache Linked List kann wie folgt implementiert werden:

```
1  Class LinkedElement:
2      key = null
3      next = null
4      Function new(k):
5          key = k
6          return this
7
8  Class LinkedList:
9      head = null
10     Function insert(k):
11         elem = new(k)
12         if head == null then
13             head = elem
14         end
15         else
16             elem.next = head
17             head = elem
18         end
19     Function delete(k):
20         prev = null
21         current = head
22         while current != null and current.key != k do
23             prev = current
24             current = current.next
25         end
26         if current == null then
27             error Element not found
28         end
29         if prev != null then
30             prev.next = current.next
31         end
32         else
33             head = current.next
34         end
35     Function search(k):
36         current = head
37         while current != null and current.key != k do
38             current = current.next
39         end
40         return current
```