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.
     $\implies$ Ausgabe nur von Eingabe abhängig, keine äußeren Faktoren.
   - Determinismus: Bei gleicher Eingabe läuft der Algorithmus immer gleich durch die Eingabe.
     $\implies$ 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: $\forall x, y, z \in M : x \leq y \land y \leq z \implies x \leq z$
  (Wenn x in Relation zu y steht und y in Relation zu z steht, so folgt, dass x in Relation zu z steht)
- Antisymmetrisch: $\forall x, y \in M : x \leq y \land y \leq x \implies x = y$
  (Wenn x in Relation zu y steht und y in Relation zu x steht, so folgt, dass x = y)
- Totalität: $\forall x, y \in M : x \leq y \lor y \leq x$
  (Alle Elemente müssen in einer Relation zueinander stehen)

2.2 Insertion Sort

```
Function insertion_sort(A):
    for i = 1 to A.length - 1 do
        key = A[i]                      ▷ Element zum Sortieren
        j = i - 1                       ▷ Einfügepunkt wird von hinten gesucht
        while j >= 0 and A[j] > key do
            A[j + 1] = A[j]             ▷ Elemente nach Rechts verschieben
            j = j - 1
        end
        A[j + 1] = key                  ▷ Element wird in den Einfügepunkt geschoben
    end
```

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

2.3 Merge Sort

```
Function mergeSort(A, left, right):
    if left < right then
        ▷ Wenn Bereich ist nicht leer mid = floor((left + right) / 2)    ▷ Nach unten gerundet
        mergeSort(A, left, mid)    ▷ Sortiert von left zu mid
        mergeSort(A,mid + 1, right)    ▷ Sortiert von mid + 1 zu right
        mergeA, left, mid, right    ▷ Fügt Hälften zusammen
    end
```

```
Function merge(A, left, mid, right):
    B = new Array[right - left + 1]    ▷ Temp array
    p = left
    q = mid + 1
    ▷ Array A wird mithilfe von zwei Pointern durchgegangen, von links und von mid + 1
    for i = 0 to right - left do
        ▷ Läuft für jedes Element im Zielbereich
        if q > right or (p =< mid and A[p] =< A[q]) then
            ▷ Wenn q > right, dann ist der rechte Teil durchlaufen. Wenn p =< mid ist, so ist der linke Teil
              noch nicht durchlaufen. Wenn das linke Element =< dem rechten ist, dann wird das Element
              dem temp array B hinzugefügt.
            B[i] = A[p]
            p = p + 1
        end
        else
            ▷ Wenn oben nicht zutrifft:
            B[i] = A[q]
            q = q + 1
        end
    end
    for i = 0 to right - left do
        ▷ Läuft wieder für jedes Element im Zielbereich
        ▷ Kopiert den jetzt sortierten temp array B zurück in den eigentlichen Array an der richtigen Stelle
        A[i + left] = B[i]    ▷ i + left, damit nicht am Anfang sondern an dem richtigen Teilbereich eingefügt
          wird.
    end
```

2.4 Quicksort

```
Function quicksort(A, left right):
    if left < right then
        q = partition(A, left, right)
        quicksort(A, left, q)
        quicksort(A, q + 1, right)
    end
```

```
Function partition(A, left, right):
    pivot = A[left]
    p = left - 1
    q = right + 1
    while p < q do
        while A[p] < pivot do
            p = p + 1
        end
        while A[q] > pivot do
            q = q - 1
        end
        if p < q then
            temp = A[q]
            A[q] = A[p]
            A[p] = temp
        end
    end
    return q
```

2.5 Radix Sort

```
keys = digits d in range [0, D-1]                 ▷ possible digits
B = new Array[0]                                  ▷ Buckets, initially empty
Function radixSort(A):
    for i = 0 to d - 1 do
        ▷ From least to most significant for j = 0 to n - 1 do
            putBucket(A, B, i, j)
        end
        a = 0
        for k = 0 to D - 1 do
            for b = 0 to B[k].size - 1 do
                A[a] = B[k][b]                    ▷ Read bucket in order
                a = a + 1
            end
            B[k].size = 0                         ▷ Clear Bucket
        end
    end
    return A
```

```
Function putBucket(A, B, i, j):
    z = A[j].digit[i]                             ▷ i-th digit of A[j]
    b = B[z].size                                 ▷ Size corresponds to next free spot
    B[z][b] = A[j]
    B[z].size = B[z].size + 1
```

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

```
Class Stack:
    arr = null
    top = -1
    Function new(n):
        arr = new Array[n]
        return this
    Function isEmpty:
        return top == -1
    Function pop:
        return arr[top–]
    Function push(k):
        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:

```
Class LinkedElement:
    key = null
    next = null
    Function new(k):
        key = k
        return this

Class LinkedList:
    head = null
    Function insert(k):
        elem = new(k)
        if head == null then
            head = elem
        end
        else
            elem.next = head
            head = elem
        end
    Function delete(k):
        prev = null
        current = head
        while current != null and current.key != k do
            prev = current
            current = current.next
        end
        if current == null then
            error Element not found
        end
        if prev != null then
            prev.next = current.next
        end
        else
            head = current.next
        end
    Function search(k):
        current = head
        while current != null and current.key != k do
            current = current.next
        end
        return current
```