
AuD - Zusammenfassung

Moritz Gerhardt

Contents

Was ist ein Algorithmus?	2
Sortieren	2
2.1 Sortierproblem	2
2.2 Insertion Sort	3
2.3	4
2.4 Quicksort	5
2.5 Radix Sort	6
Grundlegende Datenstrukturen	7
3.1 Stacks	7
3.2 Queues	8
3.3 Linked List	9
3.4 Binary Search Tree	10
3.5 Red-Black Tree	12

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 Voraussetzungen erfüllen:

1. Bestimmt:
 - Determiniert: Bei gleicher Eingabe liefert der Algorithmus 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. Anwendbar:
 - 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

```
1 class InsertionSort {
2     void insertionSort(int[] arr) {
3         for (int i = 1; i < arr.length; i++) {
4             // 1 to n - 1
5             int key = arr[i];
6             int j = i - 1;
7             while (j >= 0 && arr[j] > key) {
8                 // Loops backwards through the array starting at i - 1
9                 // until it finds an element that is greater than the key or the beginning of the array
10                arr[j + 1] = arr[j];
11                // Shifts the element to the right
12                j--;
13            }
14            arr[j + 1] = key;
15            // Assigns the key to the correct position
16        }
17    }
18 }
```

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

2.3 Merge Sort

```
1 class MergeSort {
2     void mergeSort(int[] arr, int left, int right) {
3         if (left < right) {
4             // left < right, otherwise the region has no elements
5             int mid = (left + right) / 2;
6             // Split the region into two halves and do the recursive calls
7             mergeSort(arr, left, mid);
8             mergeSort(arr, mid + 1, right);
9             // Merge the two (now sorted) halves
10            merge(arr, left, mid, right);
11        }
12    }
13
14    private void merge(int[] arr, int left, int mid, int right) {
15        int[] temp = new int[right - left + 1];
16        // Create a temporary array to store the merged elements
17
18        int p = left;
19        int q = mid + 1;
20        for (int i = 0; i < right - left + 1; i++) {
21            // Loops for each element in the region
22            if (q > right || (p <= mid && arr[p] <= arr[q])) {
23                // If p > mid the left half is finished, therefore the element needs to be in right half
24                // Otherwise p needs to be <= mid and the element at p needs to be <= the element at q
25                temp[i] = arr[p];
26                p++;
27                // Adds the element at p to the temporary array and increases p
28            }
29            else {
30                temp[i] = arr[q];
31                q++;
32                // Adds the element at q to the temporary array and increases q
33            }
34        }
35        // Copy the merged elements from the temporary array back to the original array
36        for (int i = 0; i < right - left + 1; i++) {
37            arr[left + i] = temp[i];
38            // left + 0 is the start of the region
39        }
40    }
41 }
```

2.4 Quicksort

```
1 class Quicksort {
2     void quickSort(int[] arr, int left, int right) {
3         if (left < right) {
4             // Region contains more than one element
5             int part = partition(arr, left, right);
6             quickSort(arr, left, part);
7             quickSort(arr, part + 1, right);
8         }
9     }
10
11     private int partition(int[] arr, int left, int right) {
12         int pivot = arr[left];
13
14         int p = left - 1;
15         int q = right + 1;
16         while (p < q) {
17             while (arr[p] < pivot) {
18                 p++;
19             }
20             while (arr[q] > pivot) {
21                 q--;
22             }
23             // Increase / decrease p and q until they are equal to pivot
24             if (p < q) {
25                 int temp = arr[p];
26                 arr[p] = arr[q];
27                 arr[q] = temp;
28                 // Swap arr[p] and arr[q]
29             }
30         }
31         return q;
32     }
33 }
```

2.5 Radix Sort

```
1 import java.util.ArrayList;
2
3 class RadixSort {
4     int D = 10; // possible unique digits
5     int d; // Max amount of digits
6     ArrayList<Integer>[] buckets = new ArrayList[D];
7
8     void radixSort(int[] arr) {
9         d = amountDigits(arr);
10        for (int i = 0; i < d; i++) {
11            // for each digit in the array, 0 least significant
12            for (int j = 0; j < arr.length; j++) {
13                putBucket(arr, i, j);
14            }
15            // Sorts the numbers into their buckets
16            int a = 0;
17            for (int k = 0; k < D; k++) {
18                for (int b = 0; b < buckets[k].size(); b++) {
19                    arr[a] = buckets[k].get(b);
20                    a++;
21                }
22                buckets[k].clear();
23            }
24            // Reads out the buckets in order
25        }
26    }
27
28    private void putBucket(int[] arr, int i, int j) {
29        int z = arr[j] / (int) Math.pow(D, i) % D;
30        // Gets the ith digit of the number
31        int b = buckets[z].size();
32        // size is next free index
33        buckets[z].add(b, arr[j]);
34        // puts the number in the bucket z at position b
35        // Depending on implementation might need to increase size manually
36    }
37
38    private int amountDigits(int[] arr) {
39        int max = arr[0];
40        for (int i = 1; i < arr.length; i++) {
41            if (arr[i] > max) {
42                max = arr[i];
43            }
44        }
45        // Get the biggest number
46        return (int) Math.log10(max) + 1;
47        // Get the amount of digits of the number
48    }
49 }
```

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:

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

Eine mögliche Implementation auf Grundlage eines Arrays wäre:

```
1 class Stack {
2     private int[] arr;
3     private int top;
4
5     Stack(int size) {
6         arr = new int[size];
7         top = -1;
8         // Creates a new array with size
9     }
10
11     boolean isEmpty() {
12         return top < 0;
13         // Returns true if empty
14     }
15
16     int pop() {
17         return arr[top--];
18         // Removes and returns the top element
19     }
20
21     void push(int k) {
22         arr[++top] = k;
23         // Adds an element
24     }
25 }
```

Push und Pop schmeißen Fehlermeldung wenn Stack leer bzw. voll ist. Oft als Stack underflow und Stack overflow benannt. Hier wäre es automatisch IndexOutOfBoundsException.

Oft werden Stacks auch mit variabler Größe 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 Größe kopiert wird.

3.2 Queues

Queues werden normalerweise mit den folgenden Funktionen erstellt:

- **newn**: Erstellt einen neuen Queue.
- **isEmpty**: gibt an ob der Queue leer ist.
- **enqueue(k)**: Fügt k auf den Queue hinzu
- **dequeue**: gibt das erste Element des Queues zurück und entfernt es vom Queue.

Hier ist die Implementation für Queues wie folgt:

```
1 class Queue {
2
3     private int[] arr;
4     private int front;
5     private int back;
6
7     Queue(int size) {
8         arr = new int[size];
9         front = -1;
10        back = -1;
11    }
12
13    boolean isEmpty() {
14        return back == -1;
15    }
16
17    boolean isFull() {
18        return (front + 1) % arr.length == back;
19        // If front + 1 is equal to back, the queue is full
20        // Modulo makes this usable for cyclic arrays
21    }
22
23    void enqueue(int k) {
24        if (isFull()) {
25            throw new RuntimeException("Queue is full");
26        }
27        else{
28            if (isEmpty()) {
29                front = 0;
30            }
31            back = (back + 1) % arr.length;
32            // Modulo so that cyclic arrays work
33            arr[back] = k;
34        }
35    }
36
37    int dequeue() {
38        if (isEmpty()) {
39            throw new RuntimeException("Queue is empty");
40        }
41        else {
42            int temp = arr[front];
43            front = (front + 1) % arr.length;
44            // Modulo so that cyclic arrays work
45            if (front == back) {
46                front = -1;
47                back = -1;
48            }
49            // If front and back are equal, the queue is empty -> reset
50            return temp;
51        }
52    }
53 }
```


3.3 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     Integer key = null;
3     LinkedElement next = null;
4
5     public LinkedElement(Integer key) {
6         this.key = key;
7     }
8 }
9
10
11 class LinkedList {
12     LinkedElement head = null;
13
14     void insert(int k) {
15         LinkedElement elem = new LinkedElement(k);
16         if (head == null) {
17             head = elem;
18         }
19         else {
20             head.next = elem;
21             head = elem;
22         }
23     }
24
25     void delete(int k) {
26         LinkedElement prev = null;
27         LinkedElement curr = head;
28         while (curr != null && curr.key != k) {
29             prev = curr;
30             curr = curr.next;
31         }
32         if (curr == null) {
33             throw new UException("Element not found");
34         }
35         if (prev != null) {
36             prev.next = curr.next;
37         }
38         else {
39             head = curr.next;
40         }
41     }
42
43     LinkedElement search(int k) {
44         LinkedElement curr = head;
45         while (curr != null && curr.key != k) {
46             curr = curr.next;
47         }
48         if (curr == null) {
49             throw new UException("Element not found");
50         }
51         return curr;
52     }
53 }
```

3.4 Binary Search Tree

```
1 class BSTNode {
2     Integer key;
3     BSTNode left;
4     BSTNode right;
5     BSTNode parent;
6
7     BSTNode(Integer k) {
8         key = k;
9         left = null;
10        right = null;
11        parent = null;
12    }
13 }

1 class BSTree {
2
3     BSTNode root = null;
4
5     void insert(BSTNode k) {
6         BSTNode node = root;
7         BSTNode nParent = null;
8
9         while(node != null) {
10            nParent = node;
11            if (k.key < node.key) {
12                node = node.left;
13            } else {
14                node = node.right;
15            }
16        }
17
18        k.parent = nParent;
19        if (nParent == null) {
20            root = k;
21        } else if (k.key < nParent.key) {
22            nParent.left = k;
23        } else {
24            nParent.right = k;
25        }
26    }
27
28    void delete(int k) {
29        BSTNode delNode = iterativeSearch(k);
30        // Gets the node with the corresponding key
31
32        if (delNode.left == null) {
33            transplant(delNode, delNode.right);
34        } else if (delNode.right == null) {
35            transplant(delNode, delNode.left);
36        } else {
37            BSTNode y = delNode.right;
38            while (y.left != null) {
39                y = y.left;
40            }
41            if (y.parent != delNode) {
42                transplant(y, y.right);
43                y.right = delNode.right;
44                y.right.parent = y;
45            }
46            transplant(delNode, y);
47            y.left = delNode.left;
48            y.left.parent = y;
49        }
50    }
51 }
```

```

1  void transplant(BSTNode u, BSTNode v) {
2      if (u.parent == null) {
3          root = v;
4      } else if (u == u.parent.left) {
5          u.parent.left = v;
6      } else {
7          u.parent.right = v;
8      }
9      if (v != null) {
10         v.parent = u.parent;
11     }
12 }
13 BSTNode iterativeSearch(int k) {
14     BSTNode curr = root;
15     while (curr != null && curr.key != k) {
16         if (k < curr.key) {
17             curr = curr.left;
18         } else {
19             curr = curr.right;
20         }
21     }
22     return curr;
23     // Returns null if element not found
24 }
25
26 BSTNode recursiveSearch(int k, BSTNode curr) {
27     if (curr == null) {
28         return null;
29     }
30     if (k < curr.key) {
31         return recursiveSearch(k, curr.left);
32     } else if (k > curr.key) {
33         return recursiveSearch(k, curr.right);
34     }
35
36     return curr;
37     // Returns null if element not found
38 }
39 }

```

3.5 Red-Black Tree

```
1 class RBNode extends BSTNode{
2     RBNode left;
3     RBNode right;
4     RBNode parent;
5     Color color;
6
7     RBNode(Integer k) {
8         super(k);
9         left = null;
10        right = null;
11        parent = null;
12        color = Color.BLACK;
13    }
14 }
```

```
1 class RBTree extends BSTree{
2     RBNode sent;
3     RBNode root = null;
4
5     RBTree() {
6         sent = new RBNode(null);
7         sent.color = Color.BLACK;
8         sent.left = sent;
9         sent.right = sent;
10        // Sentinel always points to itself ->
11        // node.parent.parent and its children will never result in null references
12    }
13    void insert(int k) {
14        RBNode z = new RBNode(k);
15        RBNode x = root;
16        RBNode px = sent;
17
18        while (x != null) {
19            px = x;
20            if (z.key < x.key) {
21                x = x.left;
22            } else {
23                x = x.right;
24            }
25        }
26
27        z.parent = px;
28        if (px == sent) {
29            root = z;
30        } else if (z.key < px.key) {
31            px.left = z;
32        } else {
33            px.right = z;
34        }
35
36        z.color = Color.RED;
37        fixColorsAfterInsertion(z);
38    }
```

```

1  void fixColorsAfterInsertion(RBNode z) {
2      while (z.parent.color == Color.RED) {
3          if (z.parent == z.parent.parent.left) {
4              RBNode y = z.parent.parent.right;
5              if (y != null && y.color == Color.RED) {
6                  z.parent.color = Color.BLACK;
7                  y.color = Color.BLACK;
8                  z.parent.parent.color = Color.RED;
9                  z = z.parent.parent;
10             } else {
11                 if (z == z.parent.right) {
12                     z = z.parent;
13                     rotateLeft(z);
14                 }
15                 z.parent.color = Color.BLACK;
16                 z.parent.parent.color = Color.RED;
17                 rotateRight(z.parent.parent);
18             }
19         } else {
20             // Same as above but with right and left exchanged
21             RBNode y = z.parent.parent.left;
22             if (y != null && y.color == Color.RED) {
23                 z.parent.color = Color.BLACK;
24                 y.color = Color.BLACK;
25                 z.parent.parent.color = Color.RED;
26                 z = z.parent.parent;
27             } else {
28                 if (z == z.parent.left) {
29                     z = z.parent;
30                     rotateRight(z);
31                 }
32                 z.parent.color = Color.BLACK;
33                 z.parent.parent.color = Color.RED;
34                 rotateLeft(z.parent.parent);
35             }
36         }
37     }
38     root.color = Color.BLACK;

```

```

1  }
2  void rotateLeft(RBNode x) {
3      RBNode y = x.right;
4
5      x.right = y.left;
6      if (y.left != null) {
7          y.left.parent = x;
8      }
9      y.parent = x.parent;
10     if (x.parent == sent) {
11         root = y;
12     } else if (x == x.parent.left) {
13         x.parent.left = y;
14     } else {
15         x.parent.right = y;
16     }
17     y.left = x;
18     x.parent = y;
19 }
20 void rotateRight(RBNode x) {
21     RBNode y = x.left;
22
23     x.left = y.right;
24     if (y.right != null) {
25         y.right.parent = x;
26     }
27     y.parent = x.parent;
28     if (x.parent == sent) {
29         root = y;
30     } else if (x == x.parent.right) {
31         x.parent.right = y;
32     } else {
33         x.parent.left = y;
34     }
35     y.right = x;
36     x.parent = y;
37 }
38 }

```