

ADS Fragenkatalog

Karim Ibrahim

January 2021

1. Schreiben Sie einen Algorithmus in Pseudo-Code, der überprüft, ob eine Zahl eine Primzahl ist. Tipp: Verwenden Sie die Modulo-Funktion.

Solution:

Listing 1: Java example

```
Boolean isPrime(int x) {  
    if(x < 1) return false;  
    for(int i = 2; i*i < x; i++) {  
        if(x % i == 0) return false;  
    }  
    return true;  
}
```

2. Was ist der Unterschied zwischen einer Queue und einem Stack?

Solution:

Stack: LIFO (Last in First out) Das Element welches als letztes eingefügt wurde ist das erste Element welches entfernt wird. Operationen: push(), pop(), isEmpty() (optional peek()).

Queue: FIFO (First in First out) Das Element welches als erstes eingefügt wurde ist das erste Element welches entfernt wird. Operationen: enqueue(), dequeue() oder add(), remove().

3. Was ist ein Stack; welches sind die grundlegenden Operationen; wie kann er implementiert werden ?

Solution:

Ein Stack ist eine Datenstruktur welche einen "Stapel" repräsentiert. LIFO (Last in First out) Das Element welches als letztes eingefügt wurde ist das erste Element welches entfernt wird.

Operationen: push(), pop(), isEmpty() (optional peek()).

Implementation: Ein Stack kann über eine Liste oder ein Array implementiert werden.

4. Angenommen, während der Ausführung eines Java-Programms kommt es zur Ausführung der Anweisung $x = x + 1$. Angenommen weiter, x sei eine int-Variable und diese Anweisung stehe unter der Vorbedingung $x \geq 0$. Nennen Sie eine möglichst genaue Nachbedingung.

Solution:

Hoare-Tripel: $\{x \geq 0\}x = x + 1\{x \geq 1x \leq x^{32} - 1\}$

5. Was ist eine Queue; welches sind die grundlegenden Operationen; wie kann sie implementiert werden; was ist eine PriorityQueue und wie funktioniert sie?

Solution:

Queue: Eine Queue ist eine Datenstruktur welche eine "Warteschlange" repräsentiert. FIFO (First in First out) Das Element welches als erstes eingefügt wurde ist das erste Element welches entfernt wird.

Operationen: enqueue(), dequeue() oder add(), remove(), isEmpty()

Implementation: Eine Queue kann über eine Liste oder ein Array implementiert werden.

Solution:

PriorityQueue: Eine Queue ist eine Datenstruktur welche eine "Warteschlange" repräsentiert. Die PriorityQueue setzt im Unterschied zur Queue eine Natürliche Ordnung der beinhalteten Elemente voraus.

Operationen: enqueue(), dequeue() oder add(), remove(), isEmpty()

Implementation: Eine PriorityQueue kann über eine Liste oder ein Array implementiert werden.

6. Angenommen, während der Ausführung eines Java-Programms kommt es zur Ausführung der Anweisung $x = x + 1$. Angenommen weiter, x sei eine int-Variable und diese Anweisung stehe unter der Vorbedingung $x = 5$. Nennen Sie eine möglichst genaue Nachbedingung.

Solution:

Hoare-Tripel: $\{x = 5\}x = x + 1\{x = 6\}$

7. Wie ist die Worst-Case-Laufzeit von binärer Suche? Wieso?

Solution:

Worst Case: $\mathcal{O}(\log n) \rightarrow$ Element ist nicht im Array vorhanden $\lfloor \log(n) + 1 \rfloor$ Schritte. Alternativ: Falls sich das gesuchte Element auf Stufe mit Tiefe $\log(n)$ befindet.

Best Case: $\mathcal{O}(1) \rightarrow$ Element ist in der Mitte des Arrays.

8. Erklären Sie wie binäre Suche funktioniert.

Solution:

Binäre Suche: Die Binäre Suche funktioniert nach dem "Divide-and-Conquer"-Prinzip, d.h. Das Problem wird in kleinere Teilprobleme aufgeteilt. Zuerst wird das mittlere Element des Arrays überprüft. Es kann kleiner, größer oder gleich dem gesuchten Element sein. Ist es kleiner als das gesuchte Element, muss das gesuchte Element in der hinteren Hälfte stecken, falls es sich dort überhaupt befindet. Ist es hingegen größer, muss nur in der vorderen Hälfte weitergesucht werden. Die jeweils andere Hälfte muss nicht mehr betrachtet werden. Ist es gleich dem gesuchten Element, ist die Suche beendet. Diese Prozedur wird jeweils auf dem zu durchsuchenden Array durchgeführt bis das gesuchte Element gefunden wurde oder kein Ergebnis vorliegt. Zu beachten ist dass die binäre Suche eine Totale Ordnung sowie ein sortiertes Array der zu durchsuchenden Elemente voraussetzt. (Wikipedia)

Laufzeit:

Worst Case: $\mathcal{O}(\log n) \rightarrow$ Element ist nicht im Array vorhanden $\lfloor \log(n) + 1 \rfloor$ Schritte. Alternativ: Falls sich das gesuchte Element auf Stufe mit Tiefe $\log(n)$ befindet.

Best Case: $\mathcal{O}(1) \rightarrow$ Element ist in der Mitte des Arrays.

9. Welche Laufzeit hat folgender Code - grobe Abschätzung genügt?

Listing 2: Code example

```
sum = 0;
i = 1
while i < N
    i = i*2
    sum += log(i)
print(sum)
```

Solution:

Laufzeit: $\mathcal{O}(N)$ Bsp. Im Falle $N = 2$. Exakt wäre jedoch $\mathcal{O}(N)$

10. Nennen Sie einige typische Komplexitätsklassen

Solution:

Komplexitätsklassen: P, NP, 3SAT usw. Definiert wird eine Komplexitätsklasse durch eine obere Schranke für den Bedarf einer bestimmten Ressource unter Voraussetzung eines Berechnungsmodells. Die am häufigsten betrachteten Ressourcen sind die Anzahl der notwendigen Berechnungsschritte zur Lösung eines Problems (Zeitkomplexität) oder der Bedarf an Speicherplatz (Raum- oder Platzkomplexität). Gemessen wird der Ressourcenbedarf in der Regel durch sein asymptotisches Verhalten an der Obergrenze (dem worst case) in Abhängigkeit von der Länge der Eingabe (Problemgröße). (Wikipedia)

11. Was hält die Omega-Notation (Ω) fest?

Solution:

Die Omega-Notation (Ω) beschreibt die untere Schranke für das asymptotische Verhalten der Laufzeit eines Problems.

Formal: $\exists c > 0 \exists n_0 \forall n > n_0: f(n) \geq c \cdot g(n)$

12. 1. Was hält die Theta-Notation (Big- Θ) fest und 2. wie lautet die Definition der Theta-Notation?

Solution:

Die Theta-Notation (Θ) hält die untere als auch die obere Schranke der Laufzeit eines Problems fest.

Formal: $\exists k_1, k_2 > 0, \exists n_0 \forall n \geq 0 : k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n)$

Limit: $f(n) = \mathcal{O}(g(n))$ and $f(n) = \Omega(g(n))$ (Knuth)

13. Was ist der Unterschied zwischen Average und Worst Case Laufzeit?

Solution:

Average Case: Beschreibt den Durchschnitt der Komplexität über alle Inputs.

Worst Case: Beschreibt die Maximale Komplexität über alle Inputs.

14. Falls immer $f(x) < g(x)$ ist, ist dann $f = \mathcal{O}(g)$?

Solution:

Argument 1: Wahr: $f(x) < g(x) \Rightarrow f(x) \leq c \cdot g(x) \forall x \geq x_0 \Rightarrow f = \mathcal{O}(g)$

Argument 2: Falsch: Abuse of Notation (Es wäre besser \in anstelle von $=$ zu verwenden)
sei $f(x) = x$ und $g(x) = x^2$ dann ist $f = \mathcal{O}(x) \neq g(x) = \mathcal{O}(x^2)$

15. Hat folgender Algorithmus Laufzeit $\mathcal{O}(n^3)$?

Listing 3: Java example

```
sum = 0;
for i = 1 .. n
    for j = 1 to n
        sum += i*j
print(sum)
```

Solution:

Nein, der Algorithmus hat Laufzeit $\mathcal{O}(n^2)$

16. 1. Was hält die \mathcal{O} -Notation (Big- \mathcal{O}) fest und 2. wie lautet die Definition der \mathcal{O} -Notation?

Solution:

Die \mathcal{O} -Notation bezeichnet die asymptotisch obere Schranke der Komplexität eines Problems.

Formal: $f(x) : \exists C > 0 \forall n \geq n_0 : f(x) \leq C \cdot g(x)$

Limit: $\limsup_{x \rightarrow \infty} \left| \frac{f(x)}{g(x)} \right| < \infty$

17. Welche Laufzeit hat die Suche in einer unsortierten LinkedList? Was ist, wenn die Liste sortiert ist?

Solution:

Unsortiert

Best Case: $\mathcal{O}(1)$ falls das gesuchte Element das erste Element (HEAD) der Liste ist.

Worst Case: $\mathcal{O}(n)$ falls das gesuchte Element das letzte Element (TAIL) der Liste ist.

Sortiert Best Case: $\mathcal{O}(1)$ falls das gesuchte Element in der Mitte der Liste liegt (binäre Suche).

Worst Case: $\mathcal{O}(\log n)$ mit binärer Suche.

18. Welche Laufzeit hat folgender Code?

Listing 4: Java example

```
sum = 0;
for i = 1 .. N
    for j = 1 to i
        for k = 1 to j
            sum += i*j
print(sum)
```

Solution:

Der Algorithmus hat Laufzeit $\mathcal{O}(n^3)$

19. Welche Rechenregeln gelten bei der Landau-Notation? (z.B. $\mathcal{O}(n^2 + n^3) \rightarrow \mathcal{O}(n^3)$)

Solution:

Produkt

$f_1 = \mathcal{O}(g_1)$ und $f_2 = \mathcal{O}(g_2) \rightarrow f_1 f_2 = \mathcal{O}(g_1 g_2)$

$f \cdot \mathcal{O}(g) \rightarrow \mathcal{O}(fg)$

Summe

$f_1 = \mathcal{O}(g_1)$ und $f_2 = \mathcal{O}(g_2) \rightarrow f_1 + f_2 = \mathcal{O}(\max(f_1, f_2))$

$f_1 = \mathcal{O}(g)$ und $f_2 = \mathcal{O}(g) \rightarrow f_1 + f_2 = \mathcal{O}(g)$

Multiplikation mit Konstante

$f = \mathcal{O}(g) \rightarrow kf = \mathcal{O}(g)$

20. Welche Laufzeit hat folgender Code?

Listing 5: Java example

```
sum = 0;
for i = 1 .. N
    for j = 1 to M
        for k = 1 to M
            sum += i*j
print(sum)
```

Solution:

Der Algorithmus hat Laufzeit $\mathcal{O}(M^2 + N)$

21. Nennen Sie für vier Komplexitätsklassen (O-Notation) einen Algorithmus mit dieser Laufzeit.

Solution:

Konstant $\mathcal{O}(1)$ Prüfe ob eine Zahl gerade oder ungerade ist.

Logarithmisch $\mathcal{O}(\log(n))$ Binäre Suche

Linear $\mathcal{O}(n)$ Lineare Suche in unsortiertem Array.

Exponentiell $\mathcal{O}(c^n)$ Travel Salesman Problem mit exakter Lösung

22. Welche Laufzeit hat folgender Code? Gegeben ein Array F der Länge n

Listing 6: Java example

```
for i = 1 to n
    x = binarySearch(i in F)
```

Solution:

Der Algorithmus hat Laufzeit $\mathcal{O}(n \log(n))$

23. Warum sind iterative Algorithmen oft schneller als ihre rekursiven Varianten?

Solution:

Rekursive Algorithmen bauen initial den Rekursionsheap auf was je nach Art des Problems aufwändig sein kann.

24. Wie lautet die Definition der Fibonacci-Zahlen?

Solution:

$f_n = f_{n-1} + f_{n-2}$ für $n \geq 3$ mit Anfangsbedingung $f_1 = f_2 = 1$

25. Wie kann man die Fibobacci-Zahlen iterativ berechnen?

Solution:

Listing 7: Code example

```
int fibIterative(int n) {
    if(n <= 1) {
        return n;
    }
    int fib = 1;
    int prevFib = 1;

    for(int i=2; i<n; i++) {
        int temp = fib;
        fib+= prevFib;
        prevFib = temp;
    }
    return fib;
}
```

26. Wie lautet die Rekursionsgleichung für die maximale Anzahl Vergleiche bei binärer Suche

Solution:

Sei $S(n)$ die maximale Anzahl der Schleifendurchläufe bei einer erfolglosen Suche.

$$S(n) = \begin{cases} 0, & \text{falls } n = 0. \\ 1+S(\lceil (n-1)/2 \rceil), & \text{falls } n > 0. \end{cases} \quad (1)$$

27. Nennen Sie ein Beispiel rekursiver Programmierung aus der Vorlesung.

Solution:

Quicksort Algorithmus

28. Wie kann man die Fibonacci-Zahlen rekursiv berechnen? Was passiert dabei mit der Laufzeit

Solution:

Listing 8: Code example

```
int fibonacci(int a){
    if (a==1||a==2) return 1;
    else return fibonacci(a-1)+fibonacci(a-2);
}
```

Es entsteht eine komplexe Aufruffolge der Methode und es wird die Methode recht häufig mit den gleichen Parametern aufgerufen, was die Effizienz des Algorithmus schwer beeinträchtigt. Eine nicht rekursive Methode wäre wesentlich schneller und würde weniger Speicherplatz benötigen.

29. Was ist eine Endrekursion?

Solution:

Eine rekursive Funktion f ist endrekursiv, wenn der rekursive Funktionsaufruf die letzte Aktion zur Berechnung von f ist.[1] Vorteil dieser Funktionsdefinition ist, dass kein zusätzlicher Speicherplatz zur Verwaltung der Rekursion benötigt wird.

30. Ist folgender Code endrekursiv?

Listing 9: Code example

```
public int foo(int m, int n) {
    assert(m >= 0);
    assert(n >= 0);
    if (m == 0) {
        return n + 1;
    } else if (n == 0) {
        return foo(m - 1, 1);
    } else
        return foo(m - 1, foo(m, n - 1));
}
```

Solution: Nein, dieser Code ist nicht endrekursiv. Obwohl bei jedem Programmfluss der rekursive Aufruf am Schluss erfolgt. Dieser Algorithmus bietet allerdings keine einfache Überführung in einen iterativen Algorithmus somit verletzt dieser Algorithmus die Regeln der Endrekursivität.
Quelle: answers.pdf, Patrick Egli

31. Ist folgender Code endrekursiv?

Listing 10: Java example

```
public int foo(int n) {
    assert(n >= 0);
    if (n == 0) {
        return 1;
    } else {
        return n + foo(n - 1);
    }
}
```

Solution: Nein, da der Code im letzten Aufruf nochmals eine Addition durchführt um das Endresultat zu berechnen ist er nicht Endrekursiv.

32. Aus der Denition der Fakultät $n! = 1 \cdot 2 \cdot \dots \cdot n$ kann man die Beziehung $n! = n(n - 1)!$ herleiten. Auf der Basis dieser Beziehung sollte die folgende Funktion zu einem gegebenen $n \geq 0$ die Fakultät $n!$ berechnen. Irgendetwas ist aber schief gelaufen. Was ist es und wie muss man das korrigieren?

Solution:

Listing 11: Java example

```
public int wrongFac(int n) {  
    assert(n >= 0);  
    return n * wrongFac(n-1);  
}
```

Assert entfernen, $0! = 1$ ist nicht berücksichtigt, Abbruchbedingung für $n = 0$ einführen.

33. Definieren Sie den Datentyp "Liste" rekursiv.

Solution:

RekursiveListe: $\text{List}(n) = \text{List}(n - 1) + n$

34. Skizzieren Sie eine rekursive Methode, welche die Elemente einer einfach verketteten Liste in umgekehrter Reihenfolge ausgibt (Pseudocode).

Solution:

Listing 12: Java example

```
public void reverseOutwithStream(List list) {  
    Collections.reverse(list)  
    System.out.println(list)  
}  
  
public void reverseOutWithLoop(List list) {  
    for(int i = list.size(); i >= 0; i--) {  
        System.out.println(list.get(i))  
    }  
}
```

35. Was ist der Unterschied zwischen einer einfach und einer doppelt verketteten Liste?

Solution:

SingleLinkedList: Einfache Verkettung der Elemente, jedes Element zeigt auf seinen Nachfolger.

DoubleLinkedList: Doppelte Verkettung der Elemente, jedes Element zeigt auf seinen Vorgänger (ausser das TAIL Element zeigt auf den HEAD falls vorhanden) sowie seinen Nachfolger (ausser das HEAD Element zeigt auf den TAIL)

36. Wie kann man in einer einfach verketteten Liste ein Element an Position i einfügen?

Solution:

Mit einem Iterator über die Liste iterieren bis Index i erreicht. Im Anschluss den Pointer des aktuellen i -ten Elements auf das neue Element zeigen lassen. Final den Pointer des neuen Elements auf Element $i-1$ zeigen lassen.

37. Welche Laufzeit hat das Einfügen eines Elements an der ersten Position in einer einfachverketteten Liste?

Solution:

$O(1)$

38. Nennen Sie zwei Implementierungen von Listen in Java

Solution:

ArrayList, LinkedList, Stack

39. Welche Methoden hat ein Java Iterator?

Solution:

Object next(), boolean hasNext(), void remove(), forEachRemaining(Consumer<? super E> action)

40. Wie verwendet man einen Java Iterator?

Solution:

Listing 13: Java example

```
ArrayList al = new ArrayList()
Iterator itr = al.iterator();

while(itr.hasNext()) {
    Object element = itr.next();
    System.out.print(element + " ");
}
```

41. Was ist in Java der Unterschied zwischen einer LinkedList und einer ArrayList? Was bedeutet das für die Performance.

Solution:

LinkedList: LinkedList verwendet eine doppelt verkettete Liste um ihre Elemente zu speichern. Implementiert List und Queue Interface. Schneller im manipulieren von Elementen.

ArrayList: ArrayList verwendet ein dynamisches Array um ihre Elemente zu speichern. Implementiert List Interface. Schneller im Hinzufügen/Zugriff von Elementen. Teure Arraycopy Operationen.

42. Welche Laufzeit hat das Löschen eines Elementes in einer doppelt verketteten Liste?

Solution:

$\mathcal{O}(1)$ für das Umhängen der Vor/Nachfolger.

43. Welche Laufzeit hat das Einfügen eines Elements an der ersten Position in einer ArrayList?

Solution:

$\mathcal{O}(n)$ nämlich wenn das Array voll ist und daher n Elemente in ein neues grösseres Array kopiert werden müssen. Amortisiert jedoch auf $\mathcal{O}(n)$ da kopieroperationen auf andere Operationen umgewälzt werden können, sowie mit zunehmender Grösse das Array weniger verdoppelt werden muss.

44. Warum ist ein "tail"-pointer bei Listen oft hilfreich?

Solution:

Damit kann auch von hinten über die Liste iteriert werden. Weiterhin kann der Sprung von HEAD zu TAIL in $\mathcal{O}(1)$ durchgeführt werden.

45. Gegeben sei ein unbekannter ADT mit zwei Operationen `void foo(int x) int bar()`. Sie rufen nun `foo()` und `bar()` nacheinander in folgender Reihenfolge auf:

Listing 14: Java Example

```
foo (1);  
foo (2);  
System.out.println (bar ()); // Prints "2"  
foo (3);  
foo (4);  
System.out.println (bar ()); // Prints "4"  
System.out.println (bar ()); // Prints "3"  
System.out.println (bar ()); // Prints "1"
```

Handelt es sich bei dem ADT um einen Stack, eine Queue oder etwas anderes? Konkrete Abfolge der `foo()`- und `bar()`-Operationen (und natürlich die richtige Antwort!) kann in der Prüfung abweichen.

Solution:

Stack, `foo() = pop()`, LIFO

46. Nennen Sie eine Operation, die typischerweise in Java in einer LinkedList schneller geht als in einer ArrayList. Wieso?

Solution:

`remove()` kann auf einer LinkedList in $\mathcal{O}(1)$ durchgeführt werden, während die ArrayList im worst case nämlich beim Entfernen des ersten Elements $n-1$ Elemente an den richtigen Ort verschieben muss.

47. Gegeben ist eine einfach-verkettete Liste OHNE Tail-Pointer. Welche Datenstruktur sollte man damit NICHT implementieren, einen Stack oder eine Queue?

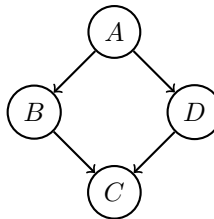
Solution:

Man sollte damit keinen Stack implementieren, da der Stack seine Operationen `push()`, `pop()`, `peek()` stets auf dem letzten Element realisiert. Ohne Tailpointer muss stets durch die ganze Liste iteriert werden.

48. Nennen Sie eine Operation, die typischerweise in Java in einer ArrayList schneller geht als in einer LinkedList. Wieso?

Solution:

`get()` kann auf einer ArrayList in $\mathcal{O}(1)$ durchgeführt werden, während die LinkedList im worst case nämlich beim holen des letzten Elements $n-1$ Elemente über n Elemente iterieren muss (oder $N/2$ falls binäre Suche).



49. Ist das ein Baum (Abbildung 1)? Begründen Sie!

Solution:

Nein, ein Baum setzt Kreisfreiheit voraus. Da der Graph aber nicht azyklisch ist, ist er kein Baum.

50. Was ist ein binaerer Suchbaum?

Solution:

Ein binärer Suchbaum, häufig abgekürzt als BST (von englisch Binary Search Tree), ist ein binärer Baum, bei dem die Knoten „Schlüssel“ tragen, und die Schlüssel des linken Teilbaums eines Knotens nur kleiner (oder gleich) und die des rechten Teilbaums nur größer (oder gleich) als der Schlüssel des Knotens selbst sind.

51. Was ist ein binärer Baum?

Solution:

Ein Binärbaum ist ein Baum mit n Knoten und maximal 2^n Blättern. Jeder Knoten kann maximal 2 direkte Nachkommen haben. Damit hat ein Baum mit n Knoten eine maximale tiefe von $\log(n)$

52. Welche Traversierungsarten bei Bäumen kennen Sie?

Solution:

PreOrder: (Root, Left, Right)

Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

PostOrder: (Left, Right, Root)

Algorithm Postorder(tree)

1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)
3. Visit the root.

InOrder: (Left, Root, Right)

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

LevelOrder: Level für Level beginnend bei der Root, von links nach rechts.

53. Was passiert, wenn Sie die Knoten eines binären Suchbaums inorder ausgeben?

Solution:

Die Elemente werden in aufsteigender Reihenfolge entsprechend ihrer Ordnungsrelation ausgegeben.

54. Gegeben sei ein Baum, bei dem die inneren Knoten Operationen sind wie $+$ $-$ $*$ $/$ und die Blätter Zahlen. Was erhalten Sie bei Inorder, Preorder Postorder?

Solution:

InOrder Bsp. $(x_1, OP, x_2) \Rightarrow$ Infix Notation

PreOrder Bsp. $(OP, x_1, x_2) \Rightarrow$ Polnische Notation

PostOrder Bsp. $(x_1, x_2, OP) \Rightarrow$ Umgekehrte Polnische Notation

55. Fügen Sie die Zahlen der Reihe nach in einen binären Suchbaum ein. Was erkennen Sie?

Solution:

Der Suchbaum besitzt genau Höhe n und ist damit maximal.

56. In einem binären Suchbaum mit ungerade vielen Elementen $n = 2k + 1$ steht an jedem Knoten, wieviele Knoten der gesamte Teilbaum hat. Wie kann man mit dieser Information den Median bestimmen.

Solution:

Da der binäre Suchbaum eine ungerade Anzahl ($n = 2k + 1$ Knoten besitzt können wir uns den Fall der Berechnung des Medians sparen. Folgende Fälle werden unterschieden

Case 1: Linker Teilbaum hat gleichviel Knoten wie rechter Teilbaum \rightarrow Root ist Median

Case 2: Linker Teilbaum hat mehr Knoten als rechter Teilbaum \rightarrow Knoten im linken Teilbaum mit $\lceil n/2 \rceil$ Knoten ist Median.

Case 3: Rechter Teilbaum hat mehr Knoten als linker Teilbaum \rightarrow Knoten im rechter Teilbaum mit $\lceil n/2 \rceil$ Knoten ist Median.

57. Wo findet man in einem binären Suchbaum den grössten Knoten?

Solution:

Man folgt dem rechten Teilbaum bis zum Knoten welcher keinen rechten Nachfolger besitzt. Dieser Knoten ist damit der grösste Knoten. Kann auch die Root sein wenn.

58. Wie hoch ist ein binärer Baum mit n Elementen mindestens und höchstens?

Solution:

Minimale Höhe: $\lfloor \log(n + 1) \rfloor$

Maximale Höhe: $n / n - 1$ je nach zählweise.

59. Kann man aus der Pre- und Postorder-Reihenfolge einen Baum immer eindeutig wiederherstellen?

Solution:

Nein, siehe bsp. mit Preorder 123, Postorder 321, kein eindeutiger Baum.

60. Was macht man, wenn ein Knoten in einem B-Baum "überläuft"

Solution:

Bei einem Überlauf eines Knotens in einem B-Baum, muss der Knoten aufgeteilt werden. Dabei wird der Median genommen welches nun als neuer Parent fungiert. Die Teilung erfolgt am Median daher muss der neue linke/rechte Teilbaum noch an den Parent gehängt werden.

61. Wie lautet die Definition eines AVL Baums?

Solution:

Ein AVL Baum ist ein binärbaum mit zusätzlicher Bedingung:

$\text{BalanceFaktor}(X) = \text{Höhe}(\text{RechterTeilbaum}(X)) - \text{Höhe}(\text{LinkerTeilbaum}(X))$ so dass

$\text{BalanceFaktor}(X) \in \{-1, 0, 1\}$

62. Welche Komplexität hat eine Rotation in einem AVL-Baum?

Solution:

Falls Rotationsstelle bekannt in linearer Zeit $\mathcal{O}(1)$. Falls gesucht werden muss $\mathcal{O}(\log n)$

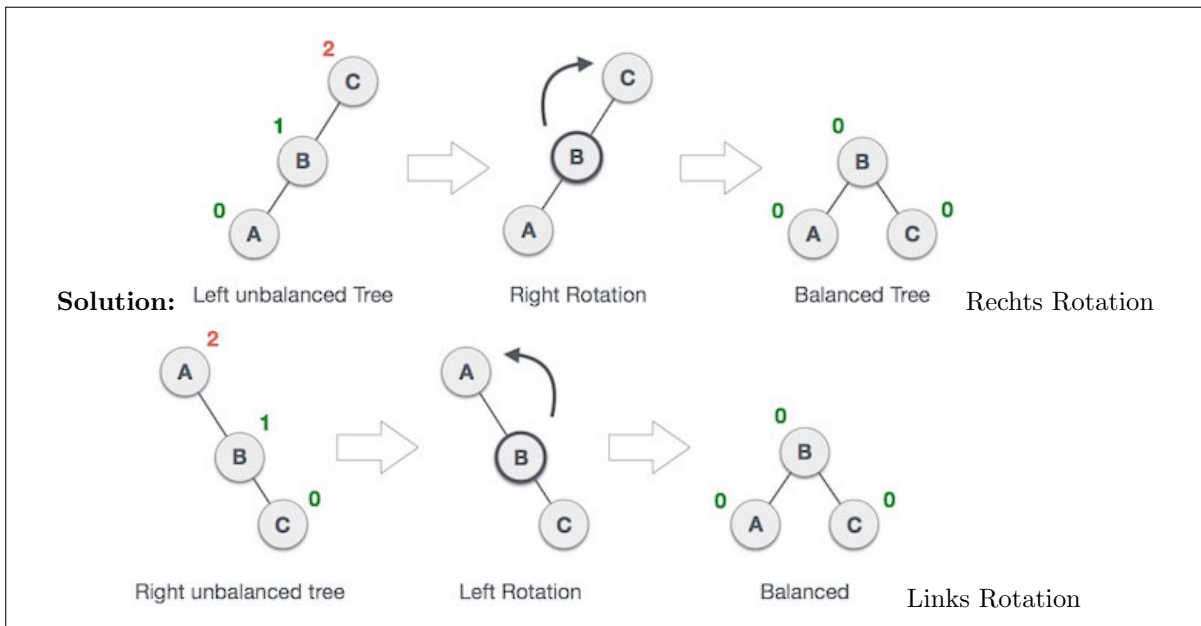
63. Ein AVL-Baum sei durch Einfügen eines Elements aus der Balance geraten. Wie kann man erkennen, ob eine einfache oder eine doppelte Rotation gebraucht wird, um die Balance wiederherzustellen?

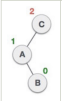




Solution:

Durch den Vergleich der Höhen des linken und rechten Teilbaumes. Wenn die Differenz grösser als 1 ist, und damit die AVL Bedingung verletzt ist, muss eine Rotation vorgenommen werden.

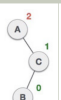




- * node.right.right am schwersten, dann Links-Rotation
- * node.left.left am schwersten, dann Rechts-Rotation
- * node.left.right am schwersten, dann Links-Rechts-Rotation
- * node.right.left am schwersten, dann Rechts-Links-Rotation

64. Welche Operationen werden benötigt, um einen durch Einfügen unbalancierten AVL-Baum wieder zu balancieren? Beschreiben Sie diese anhand einer Skizze



State	Action
	A node B has been inserted into the right subtree of A the left subtree of C, because of which C has become an unbalanced node having balance factor 2. This case is L R rotation where: Inserted node is in the right subtree of left subtree of C
	As LR rotation = RR + LL rotation, hence RR (anticlockwise) on subtree rooted at A is performed first. By doing RR rotation, node A, has become the left subtree of B.
	After performing RR rotation, node C is still unbalanced, i.e., having balance factor 2, as inserted node A is in the left of left of C
	Now we perform LL clockwise rotation on full tree, i.e. on node C. node C has now become the right subtree of node B, A is left subtree of B
	Balance factor of each node is now either -1, 0, or 1, i.e. BST is balanced now.

Links Rechts Rotation

State	Action
	A node B has been inserted into the left subtree of C the right subtree of A, because of which A has become an unbalanced node having balance factor - 2. This case is RL rotation where: Inserted node is in the left subtree of right subtree of A
	As RL rotation = LL rotation + RR rotation, hence, LL (clockwise) on subtree rooted at C is performed first. By doing RR rotation, node C has become the right subtree of B.
	After performing LL rotation, node A is still unbalanced, i.e. having balance factor -2, which is because of the right-subtree of the right-subtree node A.
	Now we perform RR rotation (anticlockwise rotation) on full tree, i.e. on node A. node C has now become the right subtree of node B, and node A has become the left subtree of B.
	Balance factor of each node is now either -1, 0, or 1, i.e., BST is balanced now.

Rechts Links Rotation Quelle:

<https://www.javatpoint.com/avl-tree>

65. Welche Komplexität hat das Einfügen in einen AVL-Baum?

Solution:

Suchen $\mathcal{O}(\log(n))$
dann Einfügen $\mathcal{O}(1)$

66. Warum hat das Einfügen/Suchen in AVL-Bäumen logarithmische Laufzeit?

Solution:

Suche eines Elements in balanciertem Binärbaum ist immer garantiert logarithmisch da max $\log(n)$ Knoten angeschaut werden müssen.

67. Was ist ein B-Baum der Ordnung k?

Solution:

Ein B-Baum ist ein vollständig balancierter Baum mit Ordnung k in welchem jeder Knoten ausser der Wurzel, mindestens $k/2$ und höchstens k Schlüssel enthält. Jeder Knoten ist entweder ein Blatt oder hat $m+1$ Nachfolger, wobei m die Anzahl Schlüssel des Knotens ist.

68. Wieviele Knoten (Pages) hat ein B-Baum der Ordnung k und Höhe h höchstens?

Solution:

Maximal $(k + 1)^h$ Knoten.

69. Wofür werden B-Bäume in der Praxis eingesetzt

Solution:

Datenbanksysteme verwenden für Indexe in der Regel Varianten von B-Bäumen. Um die Performance zu optimieren, versuchen Datenbanksysteme die Knoten in der Regel möglichst gut, zum Teil zu 100 Prozent, zu füllen. Organisation der Daten auf Disk mit fester Blockgrösse

70. Wie funktioniert Einfügen in einem B-Baum?

Solution:

Zuerst muss die entsprechende Stelle gefunden werden. Dabei geht man gleich vor wie bei der Suche in einem sortierten Baum, z.B. Binärbaum. Wenn man die Stelle gefunden hat, fügt man den neuen Schlüssel dort ein. Wenn nun dieser Knoten bereits voll ist, muss der mittlere Schlüsselwert eine Ebene nach oben wandern und der linke und rechte Teil des Knotens mit dem neuen Parent vernetzt werden.

Quelle: answers.pdf, Patrick Egli

71. Wie wird die Höhe eines B-Baumes erhöht?

Solution:

Erst wenn ein Baum komplett voll ist, wird seine Höhe angepasst. Wenn nun ein Baum voll ist und ein weiterer Schlüssel hinzugefügt wird, muss die Höhe angepasst werden. Dabei wird der aktuelle Root-Knoten aufgesplittet. Der mittlere Schlüssel des Root-Knotens wird dabei der einzige Schlüssel des neuen Root Knotens. Die anderen Schlüssel werden des alten Root-Knotens werden nun mit dem neuen Root-Knoten vernetzt. Somit ist nun die Höhe um eins gewachsen.

Quelle: answers.pdf, Patrick Egli

72. Was ist Breitensuche? Was ist Tiefensuche?

Solution:

Breitensuche Vom Ausgangsknoten werden sukzessive alle direkten Nachbarn besucht. Dann das gleiche für jeden Nachbarsknoten usw. $\mathcal{O}(|V| + |E|)$

Tiefensuche Vom Ausgangsknoten wird nach und nach aus weiter in die Tiefe gesucht. $\mathcal{O}(|V| + |E|)$

73. Was ist eine Adjazenzmatrix?

Solution:

Eine Adjazenzmatrix (manchmal auch Nachbarschaftsmatrix) eines Graphen ist eine Matrix, die speichert, welche Knoten des Graphen durch eine Kante verbunden sind. Sie besitzt für jeden Knoten eine Zeile und eine Spalte, woraus sich für n Knoten eine $n \times n$ Matrix ergibt. Ein Eintrag in der i -ten Zeile und j -ten Spalte gibt hierbei an, ob eine Kante von dem i -ten zu dem j -ten Knoten führt. Steht an dieser Stelle eine 0, ist keine Kante vorhanden – eine 1 gibt an, dass eine Kante existiert

Quelle: <https://de.wikipedia.org/wiki/Adjazenzmatrix>

74. Betrachten Sie den gerichteten gewichteten Graphen in Abbildung 3. Bestimmen Sie den maximalen

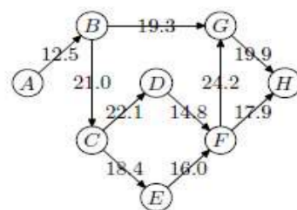


Abbildung 3: Ein Fluss.

Fluss von A nach H.

Solution:

Max Fluss: 12.5

75. Was ist ein minimaler Spannbaum? Sind minimale Spannbäume eindeutig bestimmt? Begründung/Gegenbeispiel

Solution:

Spannbaum (Alle Knoten verbunden) mit minimalem Summe der Kantengewichte. Falls jede Kante eindeutiges Kantengewicht dann gibt es einen eindeutigen minimalen Spannbaum.

76. Wie lautet die Definition eines ungerichteten (gerichteten, gewichteten) Graphen?

Solution:

Knoten: $v_i \in V$

Kanten: $e \in E$

Kantengewichte: c_{uv} für $u, v \in V$

77. Wie kann man einen ungerichteten Graphen als gerichteten Graphen repräsentieren?

Solution:

Jede Kante wird durch zwei entgegengesetzte gerichtete Kanten ersetzt.

78. Was ist ein Spannbaum?

Solution:

Ein Teilgraph eines ungerichteten Graphen welcher alle Knoten eines Graphen enthält.

79. Angenommen es gibt in einem Graphen genau eine Kante minimalen Gewichts. Muss die in einem minimalen Spannbaum vorhanden sein? Begründung

Solution:

Ja, Beweis durch Gegenbeispiel mit Annahme das Kante nicht enthalten sein muss und damit der Spannbaum minimal ist.

80. Gegeben ist ein Spannbaum eines ungerichteten Graphen. Nun verbinden Sie zwei bisher nicht verbundene Knoten. Ist das Resultat immer noch ein Spannbaum?

Solution:

Ja

81. Welche Graphenrepräsentation würden Sie verwenden, um einen Graphen als Textfile zu speichern?

Solution:

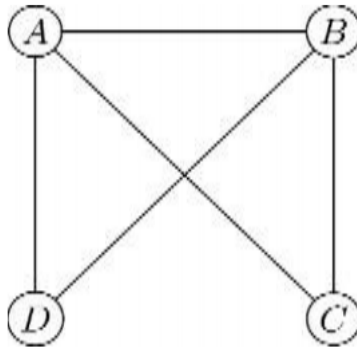
Als Adjazenzliste, mit Format Line = $\{v_i, e_j, v_k\} \dots$

82. Sind Spannbäume eindeutig bestimmt? Wenn ja, begründen Sie, wenn nein geben Sie ein Gegenbeispiel.

Solution:

Minimale Spannbäume = \emptyset Falls Graph gewichtet und disjunkte Kantengewichte, MST eindeutig.

83. Sie haben einen unger. Graphen mit den vorgegebenen Kanten. Finden Sie eine geschlossene Tour (Anfangsknoten = Zielknoten), bei der jede Kante genau einmal genommen wird, oder begründen Sie,



warum es keine solche Tour geben kann.

Solution:

Für eine geschlossene Tour (Eulerkreis) müssen alle Knotengrade gerade sein. Dies ist in diesem Graph nicht der Fall. Der Graph enthält lediglich einen Eulerweg (keiner oder genau zwei Knoten ungerade grade, Anfangsknoten ungleich Endknoten)

84. Was berechnet der Algorithmus von Dijkstra? Wie funktioniert er? Wieso ist das ein Beispiel für einen Greedy-Algorithmus?

Solution:

Single Source Shortest Path to all Nodes. Jedoch eigentlich zuerst Shortest Path zwischen zwei Nodes. Die Grundidee des Algorithmus ist es, immer derjenigen Kante zu folgen, die den kürzesten Streckenabschnitt vom Startknoten aus verspricht. Andere Kanten werden erst dann verfolgt, wenn alle kürzeren Streckenabschnitte (auch über andere Knoten hinaus) beachtet wurden. Dieses Vorgehen gewährleistet, dass bei Erreichen eines Knotens kein kürzerer Pfad zu ihm existieren kann. Eine einmal berechnete Distanz zwischen dem Startknoten und einem besuchten Knoten wird gespeichert. Die aufsummierten Distanzen zu noch nicht abgearbeiteten Knoten können sich hingegen im Laufe des Algorithmus durchaus verändern, nämlich verringern. Dieses Vorgehen wird fortgesetzt, bis die Distanz des Zielknotens berechnet wurde (single-pair shortest path) oder die Distanzen aller Knoten zum Startknoten bekannt sind (single-source shortest path).

Greedy: Aufgrund der Eigenschaft, einmal festgelegte Distanzen zum Startknoten nicht mehr zu verändern, gehört der Dijkstra-Algorithmus zu den Greedy-Algorithmen, die in jedem Schritt die momentan aussichtsreichste Teillösung bevorzugen. Anders als manche andere Greedy-Algorithmen berechnet der Dijkstra-Algorithmus jedoch stets eine optimale Lösung.

```

private void dijkstra(String from, String to) {
    //init
    Queue<DijkstraNode<Edge>> queue = new PriorityQueue<>();
    DijkstraNode<Edge> startNode = graph.findNode(from);
    DijkstraNode<Edge> current;
    //upgrade graph with DijkstraNodes
    graph.getNodes().forEach(node -> {
        node.setDist(Double.MAX_VALUE);
        node.setPrev(null);
        node.setMark(false);
    });

    startNode.setDist(0); //root has dist 0 to itself
    queue.add(startNode); //add root

    while (!queue.isEmpty()) {
        current = queue.remove();

        current.setMark(true); //mark as visited
        for (Edge edge : current.getEdges()) {
            DijkstraNode<Edge> neighbour = (DijkstraNode<Edge>) edge.getDest();

            if (!neighbour.getMark()) { //only if not marked yet
                //calc new distance
                double dist = current.getDist() + edge.getWeight();

                if (dist < neighbour.getDist()) {
                    //set new distance
                    neighbour.setDist(dist);
                    //set new previous => shortest path
                    neighbour.setPrev(current);
                    //Remove outdated neighbour
                    queue.remove(neighbour);
                    //Add updated neighbour
                    queue.add(neighbour);
                }
            }
        }
    }
}

```

85. Wie funktioniert Insertionsort?

Solution:

Iteriere über Array.

Vergleiche Aktuelles Element mit seinem Vorgänger.

Falls kleiner vergleiche mit Vorgänger. Verschiebe grössere Elemente +1

Laufzeit: $\mathcal{O}(n^2)$

Platz: $\mathcal{O}1$

Boundaries: Falls aufsteigend sortiert $\mathcal{O}n$, Falls absteigend sortiert $\mathcal{O}n^2$

86. Wie funktioniert Quicksort?

Solution:

QuickSort funktioniert nach dem Prinzip "Teile und herrsche". Dabei wird das zu sortierende Array schrittweise in kleinere Arrays zerlegt. Bei dieser Zerlegung (Partitionierung) wird ein Pivot-Element gewählt. Alle Elemente, welche wertmässig kleiner sind als das Pivot-Element kommen ins linke Teil-Array und alle anderen Element ins rechte Teil-Array. Dies hat den Vorteil, dass man nicht alle Elemente miteinander vergleichen muss, sondern man vergleicht nur die Elemente mit dem jeweiligen Pivot-Element. Der linke und rechte Teil sind nun bereits in Bezug auf das Pivot-Element sortiert. Alle Elemente links des Pivot-Elements sind kleiner als das Pivot-Element und alle Elemente rechts des Pivot-Elements sind grösser als das Pivot-Element. Nun wird der linke Teil und der rechte Teil separat sortiert. Dabei wird auf beiden Seiten ein Pivot-Element gewählt und die gleiche Logik angewendet. Das Pivot-Element wird nach der Sortierung nicht mehr verändert, weil es bereits sortiert wurde. Man macht diesen Algorithmus solange, bis die Teil-Arrays die Länge eins haben. Dann ist das Array komplett sortiert. Wichtig ist, dass beide Teil-Arrays ungefähr gleich gross sind, das heisst, dass das Pivot-Element gut gewählt werden muss. Dabei ist wichtig, dass die Suche nach dem Pivot-Element $\mathcal{O}(1)$ nicht übersteigen darf, sonst wird die Performance schlechter.

Eine mögliche Methode das Pivot-Element möglichst schnell zu wählen ist, den mittleren Wert der beiden Randelemente und dem Mittelelement wählt. * Anzahl benötigter Partitionen, Partitionierung wird immer halbiert: $\mathcal{O}(\log(n))$ * Pivot-Suche $\mathcal{O}(1)$ * Aufwand für eine Partitionierung $\mathcal{O}(n)$
Zeitkomplexität von QuickSort: $\mathcal{O}(n \cdot \log(n))$ Quelle: answers.pdf, Patrick Egli

87. Warum ist die Worse-Case-Laufzeit von Quicksort quadratisch?

Solution:

Bei ungünstiger Pivotwahl (bsp. erstes oder letztes Element des Arrays/der Partition) sowie bereits sortiertem Array werden $n-1$ vergleiche mit dem Pivotelement durchgeführt. Da partitionieren rekursiv von dem vorherigen partitionieren aufgerufen wird summieren sich die Zeitkomplexitäten jedes Aufrufs. $n + (n - 1) + \dots + 3 + 2 = \mathcal{O}(n^2)$

88. Wie funktioniert Selectionsort?

Solution:

Sei S der sortierte Teil des Arrays (vorne im Array) und U der unsortierte Teil (dahinter). Am Anfang ist S noch leer, U entspricht dem ganzen (restlichen) Array. Das Sortieren durch Auswählen läuft nun folgendermaßen ab: Suche das kleinste Element in U und vertausche es mit dem ersten Element von U (= das erste Element nach S). Danach ist das Array bis zu dieser Position sortiert. Das kleinste Element wird in S verschoben (indem S einfach als ein Element länger betrachtet wird, und U nun ein Element später beginnt). S ist um ein Element gewachsen, U um ein Element kürzer geworden. Anschließend wird das Verfahren so lange wiederholt, bis das gesamte Array abgearbeitet

worden ist; S umfasst am Ende das gesamte Array, aufsteigend sortiert, U ist leer.

Laufzeit: $\mathcal{O}(n^2)$

Platz: $\mathcal{O}1$

Boundaries: Falls aufsteigend sortiert $\mathcal{O}(n)$, Falls absteigend sortiert $\mathcal{O}(n^2)$

Vergleiche: Kleiner Gauss

Quelle: <https://de.wikipedia.org/wiki/Selectionsort>

89. Wie ist die Laufzeit von BubbleSort bei absteigend oder aufsteigend sortierter Folge?

Solution:

Laufzeit: $\mathcal{O}n$

90. Wie funktioniert Bubblesort? Was ist die Laufzeit, und warum?

Solution:

In der Bubble-Phase wird die Eingabe-Liste von links nach rechts durchlaufen. Dabei wird in jedem Schritt das aktuelle Element mit dem rechten Nachbarn verglichen. Falls die beiden Elemente das Sortierkriterium verletzen, werden sie getauscht. Am Ende der Phase steht bei auf- bzw. absteigender Sortierung das größte bzw. kleinste Element der Eingabe am Ende der Liste. Die Bubble-Phase wird solange wiederholt, bis die Eingabeliste vollständig sortiert ist. Dabei muss das letzte Element des vorherigen Durchlaufs nicht mehr betrachtet werden, da die restliche zu sortierende Eingabe keine größeren bzw. kleineren Elemente mehr enthält. Je nachdem, ob auf- oder absteigend sortiert wird, steigen die größeren oder kleineren Elemente wie Blasen im Wasser (daher der Name) immer weiter nach oben, das heißt, an das Ende der Liste. Es werden stets zwei Zahlen miteinander in „Bubbles“ vertauscht.

Listing 15: Code Example

```
bubbleSort(Array A)
  for (n=A.size; n>1; --n){
    for (i=0; i<n-1; ++i){
      if (A[i] > A[i+1]){
        A.swap(i, i+1)
      }
    }
  }
```

91. Definieren sie einen Sortieralgorithmus

Solution:

Name	Best	Average	Worst	Memory	Stable	Method	Other notes
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	No	Partitioning	Quicksort is usually done in-place with $O(\log n)$ stack space. ^{[5][6]}
Merge sort	$n \log n$	$n \log n$	$n \log n$	n	Yes	Merging	Highly parallelizable (up to $O(\log n)$ using the Three Hungarians' Algorithm). ^[7]
In-place merge sort	—	—	$n \log^2 n$	1	Yes	Merging	Can be implemented as a stable sort based on stable in-place merging. ^[8]
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No	Partitioning & Selection	Used in several STL implementations.
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No	Selection	
Insertion sort	n	n^2	n^2	1	Yes	Insertion	$O(n + d)$, in the worst case over sequences that have d inversions.
Block sort	n	$n \log n$	$n \log n$	1	Yes	Insertion & Merging	Combine a block-based $O(n)$ in-place merge algorithm ^[9] with a bottom-up merge sort.
Quadsort	n	$n \log n$	$n \log n$	n	Yes	Merging	Uses a 4-input sorting network. ^[10]
Timsort	n	$n \log n$	$n \log n$	n	Yes	Insertion & Merging	Makes n comparisons when the data is already sorted or reverse sorted.
Selection sort	n^2	n^2	n^2	1	No	Selection	Stable with $O(n)$ extra space or when using linked lists. ^[11]
Cubesort	n	$n \log n$	$n \log n$	n	Yes	Insertion	Makes n comparisons when the data is already sorted or reverse sorted.
Shellsort	$n \log n$	$n^{4/3}$	$n^{3/2}$	1	No	Insertion	Small code size.
Bubble sort	n	n^2	n^2	1	Yes	Exchanging	Tiny code size.

Quelle: https://en.wikipedia.org/wiki/Sorting_algorithm

92. Damenproblem

Solution:

Eine Möglichkeit, eine Lösung zu finden, besteht im Backtracking. Dabei wird mit der Platzierung einer Dame begonnen und sukzessive werden weitere Damen platziert, bis eine Lösung gefunden wird oder eine Kollision entsteht, dann wird die letzte Dame umplatziert und man fährt fort. Da ein solcher Algorithmus sicherlich im worst case exponentielle Laufzeit hat, eignet er sich nicht für große n . Laufzeit: $\mathcal{O}(n! * n^2)$

93. Welchen Sortieralgorithmus würden Sie nehmen, wenn Sie die Einträge in einer Datei sortieren wollen, die nicht komplett in den Hauptspeicher passt?

Solution:

Mit dem Konzept der External Sorting Algorithms: Chunkwise Mergesort mit anschließendem Schreiben auf das externe Memory. Final dann mit einem k-way merge die Chunks über Buffering mergen.

94. Welches Charakteristikum von Quicksort ist dafür verantwortlich, dass seine Worst-Case-Performance nicht besser sein kann als $O(n \log n)$? Wieso ist es überhaupt möglich, dass Distribution Sort eine Chance auf eine bessere Performance als $O(n \log n)$ hat?

Solution:

Quicksort \Leftrightarrow Binary Search Tree

DistributionSort baut auf einem völlig anderem Prinzip auf. Beim DistributionSort gibt es ein vorsortiertes Array mit Fächern. Die zu sortierenden Elemente werden dann einfach in diese Fächer eingetragen und es braucht keine Vergleiche zwischen Elementen. Die Zeitkomplexität von DistributionSort ist $O(n)$. Allerdings braucht dieses Verfahren eine spezielle Datenstruktur für jedes Problem und der Werte-Bereich sollte endlich sein und eher klein. Sobald der Wertebereich zu gross ist, macht dieses Verfahren keinen Sinn mehr. QuickSort ist ein Algorithmus der ohne Anpassungen

ein Array sortieren kann. DistributionSort muss hingegen auf die jeweiligen Probleme angepasst werden

Quelle: answers.pdf, Patrick Egli

95. Wie beeinflusst bei Quicksort die Wahl des Pivotelements das Laufzeitverhalten, wenn das Array schon sortiert ist?

Solution:

Im Idealfall wird immer das mittlere Element als Pivot gewählt so ergibt sich der Best Case von $O(n \log n)$

Im worst case wird das erste oder das letzte Element gewählt und somit die Worst Case Laufzeit von Quicksort erreicht. Dies weil dann die Arraygrösse 1 vs n-1 wird und somit partitionieren n-1 mal aufgerufen wird.

96. Bei Distribution Sort kann es u.U. notwendig sein, die Schlüssel noch auf Array-Indizes abzubilden. Beim Hashing ergibt sich dasselbe Problem. Trotzdem kann man bei Distribution Sort keine Hashfunktion verwenden, um das Problem zu lösen. Wieso nicht?

Solution:

Bei Distributionsort muss rückgeschlossen werden können welches Element y sich an Stelle x befindet. Beim Hashing wird hingegen ist nur die Frage ob dass Element y an Stelle x vorhanden ist, relevant.

97. Eine Folge von Zahlen die aufsteigend sortiert werden soll, ist bereits absteigend sortiert. Wie viele Vergleiche und Vertauschungen brauchen Bubblesort und Selection Sort?

Solution:

Vergleiche:

* Selectionsort: $(n - 1) + (n - 2) + \dots + 2 + 1 = n \cdot (n - 1) / 2$

* Bubblesort: $(n - 1) + (n - 2) + \dots + 2 + 1 = n \cdot (n - 1) / 2$

Vertauschungen:

* Selectionsort: $n - 1$ (in jedem Fall)

* Bubblesort: $n \cdot (n - 1)$ (worst-case)

98. Was ist die Levenshtein-Distanz?

Solution:

Die Levenshtein-Distanz (auch Editierdistanz) zwischen zwei Zeichenketten ist die minimale Anzahl von Einfüge-, Lösch- und Ersetz-Operationen, um die erste Zeichenkette in die zweite umzuwandeln.

99. Geben Sie ein Beispiel an, das man mit Backtracking gut lösen kann

Solution:

Travel Salesman Problem.
Weg durch ein Labyrinth mit DFS.

100. Wie kann man das Rucksack-Problem mit Brute-Force lösen? Laufzeit?

Solution:

Ja, es wird das totale gewicht sowie der wert aller möglichen subsets berechnet und dann das subset genommen welches denn höchsten wert besitzt.
Laufzeit: $\mathcal{O}(2^n)$ also exponentiell.

101. Welche Entwurfsmuster für Algorithmen kennen Sie?

Solution:

- * Iterator-Pattern (Vorlesung 2) Mit dem Iterator-Pattern kann man über beliebige Datenstrukturen iterieren
- * Visitor-Pattern (Vorlesung 5) Jedes Element wird der visit(T element) Methode übergeben. Die visit-Methode kann dann Operationen auf dem Element durchführen.
- * Singleton-Pattern
- * Factory-Pattern

102. Wie berechnet man Levenshtein-Distanz?

Solution:

Der Algorithmus ist durch folgende **Matrix-Rekurrenzen** spezifiziert, wobei u und v die beiden Eingabe-Zeichenketten bezeichnen:

$$\begin{aligned}
 m &= |u| \\
 n &= |v| \\
 D_{0,0} &= 0 \\
 D_{i,0} &= i, 1 \leq i \leq m \\
 D_{0,j} &= j, 1 \leq j \leq n \\
 D_{i,j} &= \min \begin{cases} D_{i-1,j-1} & +0 \text{ falls } u_i = v_j \\ D_{i-1,j-1} & +1 \text{ (Ersetzung)} \\ D_{i,j-1} & +1 \text{ (Einfügung)} \\ D_{i-1,j} & +1 \text{ (Löschung)} \end{cases} \\
 &1 \leq i \leq m, 1 \leq j \leq n
 \end{aligned}$$

Quelle: <https://de.wikipedia.org/wiki/Levenshtein-Distanz>

103. Findet ein Greedy-Algorithmus immer eine optimale Lösung? Wenn nein, kennen Sie ein Beispiel?

Solution:

Nein, Travel Salesman Problem mit nearest Neighbour heuristik führt sogar zur schlechtesten Lösung

104. Was ist der Unterschied zwischen Lokaler Suche und Random Sampling?

Solution:

Random Sampling = Eine Auswahl aus gleichwahrscheinlichen Elementen wird getroffen. Lokale Suche = Das Grundprinzip besteht darin, ausgehend von einer gegebenen Startlösung eine bessere Lösung zu finden, indem durch eine lokale Änderung der aktuellen Lösung eine bessere Lösung aus der gerade betrachteten Nachbarschaft gefunden wird

Quelle: https://de.wikipedia.org/wiki/Lokale_Suche

105. Warum kann man nicht jedes Problem mit Brute-Force lösen?

Solution:

Weil man zu wenig Zeit oder zu wenig Speicher hat.

106. Geben Sie ein Beispiel für einen Greedy-Algorithmus aus der Vorlesung

Solution:

Dijkstra-Algorithmus: Single Source All Pairs Shortest Path.

107. Findet Backtracking immer eine optimale Lösung?

Solution:

Ja, wenn eine beste Lösung existiert, wird diese auch gefunden mit der Tiefensuche. Es kann sein, dass es einfach ziemlich lange dauern kann, weil gegebenenfalls alle Möglichkeiten durchprobiert werden müssen.

108. Hashing: Was versteht man unter einer Kollision und warum muss es (in der Regel) Kollisionen geben?

Solution:

Da Hashfunktionen im Allgemeinen nicht eindeutig (injektiv) sind, können zwei unterschiedliche Schlüssel zum selben Hash-Wert, also zum selben Feld in der Tabelle, führen. Dieses Ereignis wird als Kollision bezeichnet. In diesem Fall muss die Hashtabelle mehrere Werte an derselben Stelle / in demselben Bucket aufnehmen.

109. Hashing: Erklären Sie, warum trotz Kollisionen beim Hashing die Elemente wiedergefunden werden.

Solution:

Kollisionsbehandlung z.B. mit Quadratischem Sondieren garantiert das Wiederfinden des Wertes.

110. Sie machen Hashing mit linearem oder quadratischem Sondieren. Sie rechnen damit, dass in Ihrer Hashtabelle mehrfach eingefügt, gelöscht und gesucht wird. Die Reihenfolge, in der das geschieht, kennen Sie nicht. Um den Eintrag in der Tabelle "map" am Index h zu löschen, schreiben Sie "map[h] = null". Warum ist das falsch und was müssen Sie stattdessen tun?

Solution:

Wenn mehrere Elemente mit dem Hash-Code h eingefügt wurden, dann besteht die Möglichkeit, dass man das falsche Element löscht. Um das korrekte Element zu löschen, muss man es zuerst suchen. Die Suche funktioniert gleich wie beim Einfügen, ausser das man noch die Elemente vergleicht an der entsprechenden Stelle. Wenn das Element gefunden ist kann man es löschen. Nach dem löschen könnte man noch ein Rehashing durchführen, sodass das Einfügen und die Suche etwas effizienter ist. Durch das Rehashing passt man die Reihenfolge in der Hashtable an, falls es mehrere Elemente mit dem gleichen Hashcode gibt.

111. Sie machen Hashing mit linearem oder quadratischen Sondieren. Sollten Sie den Füllgrad der Tabelle beim Löschen eines Elements nach unten korrigieren oder nicht? Warum?

Solution:

Falls ein Rehashing durchgeführt wird nach dem Löschen, dann muss der Füllgrad um eins kleiner gemacht werden. Falls man beim gelöschten Eintrag "DELETED" schreibt, dann darf man den Füllgrad nicht anpassen.

112. Hashing: Skizzieren Sie quadratisches Probing. Wie geht man vor?

Solution:

Quadratic probing is an open addressing scheme in computer programming for resolving hash collisions in hash tables. Quadratic probing operates by taking the original hash index and adding successive values of an arbitrary quadratic polynomial until an open slot is found.
Quelle: https://en.wikipedia.org/wiki/Quadratic_probing

113. Sie möchten in einem Text mit dem Knuth–Morris–Pratt-Algorithmus nach dem Muster ABCDEFGH suchen. Wie sieht die next-Tabelle aus? Begründen Sie.

Solution:

Alle Einträge in der Next-Tabelle sind 0, weil dieses Pattern keine sich wiederholenden Sub-Patterns beinhaltet.

114. Sie möchten in einem Text mit dem Knuth-MorrisPratt-Algorithmus nach dem Muster BANANANANANANA suchen. Wie sieht die next-Tabelle aus? Begründen Sie.

Solution:

Ein sich wiederholendes Such-Pattern muss das Präfix des Such-Patterns beinhalten, ansonsten wird es nicht berücksichtigt. Bei diesem Beispiel gibt es zwar sich wiederholende Sub-Patterns, allerdings beinhaltet keines von denen das Präfix. Somit sind auch hier alle Einträge der Next-Tabelle 0.