Probeklausur

Einführung in die Informatik - Vertiefung

Universität:	Technische Universität Berlin
Kurs/Modul:	Einführung in die Informatik - Vertiefung
Bearbeitungszeit:	120 Minuten
Erstellungsdatum:	September 20, 2025

Bearbeitungszeit: 120 Minuten.

Aufgabe 1.

(a) Geben Sie eine Java-ähnliche Schnittstelle für einen generischen Stack an. Verwenden Sie `T` als Typparameter. Beschreiben Sie die wichtigsten Operationen und deren Typen.

```
public interface Stack<T> {
  void push(T x);
  T pop();
  boolean isEmpty();
  int size();
  Iterator<T> iterator();
}
```

(b) Beschreiben Sie, wie man einen Iterator über die Elemente eines generischen Stack implementieren könnte. Skizzieren Sie eine inneren Klasse `StackIterator` mit minimaler Funktionalität.

(c) Diskutieren Sie die Laufzeitkomplexität der Grundoperationen `push`, `pop`, `size` und der Iterator-Methoden `hasNext`, `next` in amortisierter Sicht. Begründen Sie Ihre Aussagen.

(d) Erläutern Sie kurz, welche Eigenschaften ein abstrakter Datentyp *Stack* charakterisieren. Gehen Sie auf LIFO-Eigenschaft und mögliche Invarianten ein.

Aufgabe 2.

(a) Gegeben sei eine einfach verkettete Liste mit Knoten

```
class Node<T> { T data; Node<T> next; }.
```

Schreiben Sie eine iterative Funktion zur Umkehrung der Liste. Benennen Sie Eingabe und Ausgabe eindeutig.

(b) Diskutieren Sie die Laufzeitkomplexität des Einfügens am Kopf vs. am Tail. Gehen Sie davon aus, dass der Listenkopf bekannt ist, aber kein Tail-Zeiger vorliegt.

(c) Gegeben sei ein binärer Suchbaum mit n Knoten. Beschreiben Sie eine Inorder-Traversierung und begründen Sie, warum die Laufzeit $\Theta(n)$ beträgt.

(d) Definieren Sie die Höhe eines Binärbaums rekursiv. Geben Sie eine Formel an, die die Höhe in Abhängigkeit von linken und rechten Teilbäumen beschreibt.

Aufgabe 3.

(a) Ein gerichteter gewichteter Graph $G = (V, E)$ besitzt $|V|$ Knoten und $|E|$ Kanten. Beschreiben Sie den Dijkstra-Algorithmus und geben Sie die Zeitkomplexität in Abhängigkeit von $|V|$ und $|E|$ an. Erläutern Sie, unter welchen Voraussetzungen der Algorithmus sinnvoll eingesetzt wird.

(b) Beschreiben Sie den Unterschied zwischen BFS und DFS in Bezug auf Traversierung eines ungerichteten Graphen. Geben Sie die jeweilige Laufzeitkomplexität in Abhängigkeit von $|V|$ und $|E|$ an.

(c) Skizzieren Sie in Pseudocode einen BFS-Algorithmus zur Suche eines kürzesten Weges in ungewichteten Graphen. Begründen Sie die Korrektheit der Methode.

(d) Geben Sie eine einfache Begründung dafür, warum die Zeitkomplexität von BFS $\mathcal{O}(|V| + |E|)$ beträgt.

Aufgabe 4.

(a) Beschreiben Sie die Binärsuche in einem sortierten Array $\{a_i\}_{i=1}^n$. Geben Sie den Algorithmus in Klartext und erklären Sie die Laufzeit $\mathcal{O}(\log n)$.

(b) Geben Sie eine kurze Beschreibung von QuickSort mit Pivotwahl. Diskutieren Sie die mittlere und die schlechteste Laufzeit und die Gründe dafür.

(c) Unterscheiden Sie stabile von in-stabile Sortieralgorithmen. Geben Sie jeweils ein kurzes Beispiel an, das den Unterschied illustriert.

(d) Welche Kosten fallen bei einer vollständigen Traversierung eines Graphen mit BFS bzw. DFS an? Geben Sie die jeweiligen Komplexitäten an und vergleichen Sie sie.

Aufgabe 5.

(a) Gegeben die Boolesche Funktion

$$F(A, B, C) = AB'C + A'BC + ABC' + A'B'C,$$

bestimmen Sie eine möglichst einfache Summe aus Produkten. Skizzieren Sie die Vorgehensweise einer Minimierung durch Boolesche Algebra oder Karnaughkarte. Begründen Sie, warum die gefundene Form eine Reduktion darstellt.

(b) Leiten Sie mithilfe von DeMorgan und Distribution eine alternative Darstellung von $\overline{F}$ her. Interpretieren Sie das Ergebnis als logische NOT-Gates in einem Schaltungsentwurf.

(c) Beschreiben Sie kurz, wie man die vereinfachte Boolesche Funktion in eine Schaltungslogik überführt. Welche Grundgatter sind ausreichend, um die Funktion zu realisieren?

Lösungen

Bearbeitungszeit: 120 Minuten.

Aufgabe 1.

(a) Geben Sie eine Java-ähnliche Schnittstelle für einen generischen Stack an. Verwenden Sie `T` als Typparameter. Beschreiben Sie die wichtigsten Operationen und deren Typen.

```
public interface Stack<T> {
  void push(T x);
  T pop();
  boolean isEmpty();
  int size();
  Iterator<T> iterator();
}
```

Lösung: Die Schnittstelle `Stack<T>` definiert die grundlegenden Operationen eines Stacks mit LIFO-Verhalten. Wichtige Punkte: - `push(T x)` fügt ein Element `x` oben auf dem Stack hinzu; Rückgabetyp `void`. - `pop()` entfernt das oberste Element und liefert es zurück; Rückgabewert `T`. - `isEmpty()` gibt an, ob der Stack keine Elemente enthält; Typ `boolean`. - `size()` liefert die Anzahl der Elemente im Stack; Typ `int`. - `iterator()` liefert einen Iterator über die Elemente des Stacks von oben nach unten (entspricht der Iteration über das aktuelle Stack-Top-zu-Unten-Reihenfolge). Typ `Iterator<T>` (Java-Standardtyp).

Aus Sicht der Semantik ist zu beachten: - Der Iterator muss keine Modifikation des Stack erlauben; optional kann `remove()` unsupported sein. - Der Typ `T` ist generisch, daher können alle Referenztypen verwendet werden, primitive Typen würden durch deren Wrapper-Typen benötigt.

(b) Beschreiben Sie, wie man einen Iterator über die Elemente eines generischen Stack implementieren könnte. Skizzieren Sie eine inneren Klasse `StackIterator` mit minimaler Funktionalität.

```
public class Stack<T> {
  private Node<T> top; // oder Head, je nach Implementierung

  private class StackIterator implements Iterator<T> {
    private Node<T> current = top;

    public boolean hasNext() {
      return current != null;
    }

    public T next() {
      if (!hasNext()) throw new NoSuchElementException();
      T val = current.data;
      current = current.next;
      return val;
    }

    public void remove() {
```

```
      throw new UnsupportedOperationException();
    }
  }

  public Iterator<T> iterator() {
    return new StackIterator();
  }

  // weitere Stack-Methoden (push/pop/...), die Node<T> top verwenden
}
```

Lösung: Die innere Klasse `StackIterator` hält einen Zeiger `current` auf das aktuelle Stack-Element und liefert in `next()` das aktuelle data-Feld sowie den nächsten Knoten. Die Methode `hasNext()` prüft, ob noch ein weiteres Element vorhanden ist. Die Methode `remove()` wird hier als nicht unterstützte Operation implementiert. Die `iterator()`-Methode erzeugt eine Instanz von `StackIterator`.

(c) Diskutieren Sie die Laufzeitkomplexität der Grundoperationen `push`, `pop`, `size` und der Iterator-Methoden `hasNext`, `next` in amortisierter Sicht. Begründen Sie Ihre Aussagen.

Lösung: - `push`: Bei einer verketteten Liste (Node-Impl.) ist `push` stets O(1). Bei einer dynamischen Array-Implementierung kann eine Kopier- oder Realloc-Operation auftreten; in der amortisierten Sicht ist `push` O(1) durchschnittlich, da teure Reallocs seltener auftreten. - `pop`: O(1) in der verketteten Liste; bei Array-Implementierung ggf. O(1) plus gelegentliches Shrinking (wenn implementiert), amortisiert ebenfalls O(1). - `size`: O(1), wenn die Größe als Feld geführt wird; sonst O(n) durch Zählen bei Bedarf. - `hasNext()` und `next()` der Iterator: `hasNext()` ist O(1). `next()` ist O(1) pro Element, insgesamt O(m) für das Durchlaufen von m Elementen. Die Gesamtkosten des Iterierens über alle Elemente betragen also O(n) für n Elemente.

Hinweis: Reale Implementierungen oft eine `size`-Variable verwenden, um `size()` O(1) zu halten; ansonsten können Zählungen während der Iteration erforderlich sein.

(d) Erläutern Sie kurz, welche Eigenschaften ein abstrakter Datentyp *Stack* charakterisieren. Gehen Sie auf LIFO-Eigenschaft und mögliche Invarianten ein.

Lösung: Ein Stack ist ein abstrakter Datentyp mit folgender Haupt-Eigenschaft: - LIFO (Last In, First Out): Das zuletzt eingefügte Element wird als Erstes entfernt. - Grundinvarianten (typisch): - Es existiert ein Zeiger/Referenz *top* auf das oberste Element oder eine ähnliche Repräsentation. - Alle Elemente befinden sich in einer linearen Folge, die die Reihenfolge der Einfügungen widerspiegelt. - Der Stack enthält genau alle Elemente, die durch vorherige `push`-Operationen eingefügt wurden und noch nicht durch `pop` entfernt wurden. - Die `size()`-Anzeige entspricht der Anzahl der Elemente zwischen dem unteren Anfangspunkt und dem aktuellen Top. - Implementationsoptionen: Verkettete Liste oder dynamisches Array; beide erfüllen die LIFO-Eigenschaft, unterscheiden sich aber in Speicherverwaltung, Iterator-Verhalten und amortisierten Kosten.

Aufgabe 2.

(a) Gegeben sei eine einfach verkettete Liste mit Knoten

`class Node<T> { T data; Node<T> next; }`.

Schreiben Sie eine iterative Funktion zur Umkehrung der Liste. Benennen Sie Eingabe und Ausgabe eindeutig.

Lösung:

```
public static <T> Node<T> reverse(Node<T> head) {
  Node<T> prev = null;
  Node<T> curr = head;
  while (curr != null) {
    Node<T> next = curr.next;
    curr.next = prev;
    prev = curr;
    curr = next;
  }
  return prev;
}
```

Lösung: Die Methode durchläuft die Liste einmal, verschiebt jeden Zeiger von next so, dass er auf den vorhergehenden Knoten zeigt, und gibt am Ende den neuen Kopf `prev` zurück. Zeitkomplexität: $\Theta(n)$; Platzkomplexität: $\Theta(1)$ zusätzlich (in-place).

(b) Diskutieren Sie die Laufzeitkomplexität des Einfügens am Kopf vs. am Tail. Gehen Sie davon aus, dass der Listenkopf bekannt ist, aber kein Tail-Zeiger vorliegt.

Lösung: - Einfügen am Kopf: $\Theta(1)$, da ein neuer Knoten vor dem aktuellen Kopf eingefügt wird und der Kopfzeiger entsprechend angepasst wird. - Einfügen am Tail ohne Tail-Zeiger: $\Theta(n)$ im schlechtesten Fall, da der Liste von Kopf aus bis zum letzten Element iteriert werden muss, um den letzten Knoten anzuhängen. Falls ein Tail-Zeiger eingeführt wird, kann das Einfügen am Tail auf $\Theta(1)$ reduziert werden.

(c) Gegeben sei ein binärer Suchbaum mit n Knoten. Beschreiben Sie eine Inorder-Traversierung und begründen Sie, warum die Laufzeit $\Theta(n)$ beträgt.

Lösung:

```
void inorder(Node<T> root) {
  if (root == null) return;
  inorder(root.left);
  visit(root.data);
  inorder(root.right);
}
```

Begründung: Inorder traversiert jeden Knoten genau einmal (jeweils Besuch links, Knotenwert, rechts). Für einen Baum mit n Knoten wird jeder Knoten genau einmal besucht, sodass

die Laufzeit $\Theta(n)$ ist. Zusätzlich erzeugen die rekursiven Aufrufe eine maximale Stapelgröße proportional zur Baumhöhe, aber die Gesamtkosten bleiben $\Theta(n)$.

(d) Definieren Sie die Höhe eines Binärbaums rekursiv. Geben Sie eine Formel an, die die Höhe in Abhängigkeit von linken und rechten Teilbäumen beschreibt.

Lösung: Eine gängige Definition (Konvention) ist: - Höhe eines leeren Baums ($\emptyset$) sei -1. - Für einen Nicht-Null-Knoten gilt: $\mathrm{H}(T) = 1 + \max\big(\mathrm{H}(\mathrm{L}), \mathrm{H}(\mathrm{R})\big)$, wobei L und R die linken bzw. rechten Teilbäume des Knotens sind.

Folglich hat ein Blatt die Höhe 0. Ggf. kann man alternativ $\mathrm{H}(\emptyset) = 0$ und $\mathrm{H}(T) = 1 + \max(\mathrm{H}(\mathrm{L}), \mathrm{H}(\mathrm{R}))$ verwenden; dann hat ein Blatt die Höhe 1. Die hier genutzte Konvention ergibt eine klare Interpretation: Höhe ist die maximale Pfadlänge von der Wurzel bis zu einem Blatt in Kanten.

Aufgabe 3.

(a) Ein gerichteter gewichteter Graph $G = (V, E)$ besitzt $|V|$ Knoten und $|E|$ Kanten. Beschreiben Sie den Dijkstra-Algorithmus und geben Sie die Zeitkomplexität in Abhängigkeit von $|V|$ und $|E|$ an. Erläutern Sie, unter welchen Voraussetzungen der Algorithmus sinnvoll eingesetzt wird.

Lösung: Dijkstra bestimmt für jeden Knoten die kürzesten Pfade vom Startknoten s. Zentraler Ablauf: - Initialisiere Abstände $\text{dist}[v] = \infty$ für alle $v \in V$, $\text{dist}[s] = 0$. - Behalte eine Priority-Queue (PQ) der Knoten nach dist sortiert. - Solange die PQ nicht leer ist, entferne den Knoten u mit kleinstem $\text{dist}[u]$, aktualisiere alle Nachbarkanten (u, v) mit Gewicht $w(u, v)$ und passe $\text{dist}[v]$ ggf. an (Relaxation). - Wenn gewünscht, kann man Vorwärtszeiger (Prev) speichern, um den Pfad zurückzuverfolgen.

Zeitkomplexität: - Mit einer Binär-Heap-PQ und einer Adjazenzliste: $\mathcal{O}((|V|+|E|)\log|V|)$. - Bei Verwendung einer adjazenzmatrix oder eines einfachen linearem Durchlaufes: $\mathcal{O}(|V|^2+|E|)$ bzw. $\mathcal{O}(|V|^2)$ insgesamt. - Voraussetzungen: Alle Kantengewichte sind nicht negativ. Bei negativen Gewichten ist Dijkstra nicht korrekt; hier würde man auf Algorithmen wie Bellman-Ford zurückgreifen.

(b) Beschreiben Sie den Unterschied zwischen BFS und DFS in Bezug auf Traversierung eines ungerichteten Graphen. Geben Sie die jeweilige Laufzeitkomplexität in Abhängigkeit von $|V|$ und $|E|$ an.

Lösung: - BFS (Breitensuche): Nutzt eine Queue. Besucht Knoten ebenenweise, von der Wurzel aus, wobei Abstände (bzw. Pfadlängen) in Schritten erhöht werden. Eigenschaften: Liefert kürzeste Pfade in ungewichteten Graphen. Laufzeit: $\mathcal{O}(|V| + |E|)$. Speicherbedarf: $\mathcal{O}(|V|)$ für die Queue. - DFS (Tiefensuche): Nutzt Stack bzw. Rekursion. Vertieft so lange wie möglich in einen Pfad, bevor es zurückkehrt. Eigenschaften: Liefert eine Tiefensortierung; in manchen Anwendungen eignet sich DFS besser für Backtracking-Probleme. Laufzeit: $\mathcal{O}(|V| + |E|)$. Speicherbedarf: $\mathcal{O}(|V|)$ im schlechtesten Fall (Tiefe des Baums/Rekursionstapel).

Hinweis: Für Graphen mit Adjazenzlisten ist die gegebene bound $\mathcal{O}(|V| + |E|)$ typisch. In Graphen mit Adjazenzmatrix kann BFS/DFS zu $\mathcal{O}(|V|^2)$ werden.

(c) Skizzieren Sie in Pseudocode einen BFS-Algorithmus zur Suche eines kürzesten Weges in ungewichteten Graphen. Begründen Sie die Korrektheit der Methode.

Lösung:

```
BFS_shortest_path(Graph G, Vertex s, Vertex t):
  for alle V in G: dist[V] <- -1; prev[V] <- null
  dist[s] <- 0
  Q <- leere Warteschlange
  enqueue(Q, s)

  while nicht empty(Q):
      u <- dequeue(Q)
      if u == t: break
      for jeden Nachbarn v von u:
          if dist[v] == -1:
              dist[v] <- dist[u] + 1
```

```
                prev[v] <- u
                enqueue(Q, v)

    if dist[t] == -1: return "kein Pfad"
    // Pfad rekonstruieren
    Path <- leere Liste
    for (x = t; x != null; x = prev[x]) push_front(Path, x)
    return Path
```

Begründung der Korrektheit: - BFS entdeckt Knoten in aufsteigender Distanz vom Startknoten, da jeder Schritt genau eine Kante weitergeht. - Der erste Zeitpunkt, zu dem ein Knoten v aus dem Nachbarschaftsfenster von u entdeckt wird, setzt dist[v] auf die minimale Distanz von s nach v, da alle früheren Pfade kürzer waren und BFS sie bereits berücksichtigt hat. - Der rückwärts gerichtete Aufbau des Pfades über `prev` liefert den kürzesten Weg von s nach t, sofern ein solcher existiert.

(d) Geben Sie eine einfache Begründung dafür, warum die Zeitkomplexität von BFS $\mathcal{O}(|V| + |E|)$ beträgt.

Lösung: Jedes Knoten ($|V|$) wird höchstens einmal in die Queue aufgenommen, und jede Kante ($|E|$) wird genau einmal durchlaufen, wenn ihr Ursprungsknoten betrachtet wird. Die Gesamtkosten setzen sich somit aus dem Zuschalten aller Knoten plus dem Durchlaufen aller Kanten zusammen, also $\mathcal{O}(|V| + |E|)$.

Aufgabe 4.

(a) Beschreiben Sie die Binärsuche in einem sortierten Array $\{a_i\}_{i=1}^n$. Geben Sie den Algorithmus in Klartext und erklären Sie die Laufzeit $\mathcal{O}(\log n)$.

Lösung: Klartext-Algorithmus: - Setze low = 1, high = n. - Solange low $\leq$ high: - midd es = $\lfloor \text{low} + (\text{high} - \text{low})/2 \rfloor - Falls a_{\text{mid}} = x$, gib mid zurück. - Falls $a_{\text{mid}} < x$, setze low = mid + 1. - Sonst setze high = mid - 1. - Wenn kein Treffer, gib Fehlermeldung aus (z. B. -1).

Begründung der Laufzeit: - In jedem Schritt wird das Suchintervall grob halbiert, wodurch sich die Anzahl möglicher Positionen exponentiell reduziert. Nach höchstens $\lceil \log_2 n \rceil$ Schritten bleibt entweder der Treffer oder der Bereich leer. Daher $\mathcal{O}(\log n)$.

(b) Geben Sie eine kurze Beschreibung von QuickSort mit Pivotwahl. Diskutieren Sie die mittlere und die schlechteste Laufzeit und die Gründe dafür.

Lösung: QuickSort-Algorithmus: - Wähle einen Pivot p aus dem Array. - Teile das Array in drei Teile: Werte kleiner als p, Werte gleich p, Werte größer als p. - Rekursiv sortiere die Teile links und rechts des Pivots; füge die drei Teile zusammen.

Laufzeiten: - Mittlere (durchschnittliche) Zeit: $\mathcal{O}(n \log n)$. - Schlechteste Zeit: $\mathcal{O}(n^2)$, z. B. wenn der Pivot immer der größte oder kleinste Restwert ist (falsche Pivotwahl bei fast sortiertem oder umgekehrt sortiertem Input). - Begründung: Die Partitionierung teilt das Problem jedes Mal fast in zwei Hälften, was zu einer Rekursionstiefe von $\mathcal{O}(\log n)$ bei guter Pivotwahl führt; bei schlechtester Pivotwahl bleibt eine Hälfte nahezu unverändert, wodurch die Rekursionstiefe n erreicht und die Gesamtkosten $\mathcal{O}(n^2)$ werden.

(c) Unterscheiden Sie stabile von in-stabile Sortieralgorithmen. Geben Sie jeweils ein kurzes Beispiel an, das den Unterschied illustriert.

Lösung: - Stabil: Ein Sortierverfahren, das bei gleichen Schlüsseln die relative Reihenfolge der Elemente beibehält. Beispiele: MergeSort, InsertionSort (in typischer Implementierung). Beispiel: Gegeben Paare (Name, Alter): [(Bob, 30), (Alice, 25), (Bob, 25)]. Sortiert nach Alter stabil: [(Alice,25), (Bob,30), (Bob,25)] – die beiden "Bob"-Elemente behalten ihre ursprüngliche Reihenfolge relativ zueinander. - Instabil: Ein Sortierverfahren, bei dem gleiche Schlüssel ihre relative Reihenfolge ändern können. Beispiele: Standard-Quicksort (in vielen Implementationen), SelectionSort ist allgemein instabil. Beispiel: Obige Liste sortiert nach Alter stabil vs. instabil könnte sich unterscheiden, wobei im instabilen Fall die beiden "Bob"-Elemente die Reihenfolge tauschen könnten.

(d) Welche Kosten fallen bei einer vollständigen Traversierung eines Graphen mit BFS bzw. DFS an? Geben Sie die jeweiligen Komplexitäten an und vergleichen Sie sie.

Lösung: - Beides (mit Adjazenzliste): $\mathcal{O}(|V| + |E|)$. Jede Knoten wird höchstens einmal besucht, jede Kante höchstens zweimal (je nachdem wie oft Knoten in der Traversierung betrachtet wird). - Mit Adjazenzmatrix: BFS/DFS können auf $\mathcal{O}(|V|^2)$ kommen, da jeder Knoten potenziell alle anderen Knoten prüfen muss. - Unterschied: BFS ist gut für Distanz- oder Pfad-Informationen in ungewichteten Graphen; DFS neigt dazu, tiefe Pfade zu erkunden, was in backtracking-basierten Problemen nützlich ist.

Aufgabe 5.

(a) Gegeben die Boolesche Funktion

$$F(A, B, C) = AB'C + A'BC + ABC' + A'B'C,$$

bestimmen Sie eine möglichst einfache Summe aus Produkten. Skizzieren Sie die Vorgehensweise einer Minimierung durch Boolesche Algebra oder Karnaughkarte. Begründen Sie, warum die gefundene Form eine Reduktion darstellt.

Lösung: Wir minimieren durch Aggregation benachbarter Minoterme in der Karnaughkarte (3 Variablen). Die Minoperatoren sind - m1: A' B' C - m3: A' B C - m5: A B' C - m6: A B C' Durch Gruppenbildung erhält man: - Gruppe m1 und m3 liefern A' C - Gruppe m1 und m5 liefern B' C - Rest bleibt m6 geliefert durch A B C' Daraus ergibt sich F = A' C + B' C + A B C' Kürzung: F = C(A' + B') + A B C' Diese Form ist eine Reduktion gegenüber der ursprünglichen Summe von vier Produkten.

(b) Leiten Sie mithilfe von DeMorgan und Distribution eine alternative Darstellung von $\overline{F}$ her. Interpretieren Sie das Ergebnis als logische NOT-Gates in einem Schaltungsentwurf.

Lösung: Zunächst Definiere $F = A'C + B'C + ABC'$. Dann

$$\overline{F} = \overline{A'C + B'C + ABC'} = (A'C)' \cdot (B'C)' \cdot (ABC')' = (A + C')(B + C')(A' + B' + C).$$

Dies ist eine Produkt-der-Summen-Form (PDSF). Interpretation als Schaltung: - Verwende Inverter (NOT)en, um die Literale A', B', C' bereitzustellen. - Baue drei Summen-out (OR-Gatter) mit je zwei bzw. drei Eingängen: (A + C'), (B + C'), (A' + B' + C). - Verbinde diese drei Ergebnisse durch ein dreifaches UND-Gatter (AND). Damit realisiert man $\overline{F}$ via eine NOR/NAND-Logik-Architektur mit Zwischenstufen (je nach Gattersatz) inklusive der benötigten NOT-Gatter.

(c) Beschreiben Sie kurz, wie man die vereinfachte Boolesche Funktion in eine Schaltungslogik überführt. Welche Grundgatter sind ausreichend, um die Funktion zu realisieren?

Lösung: Eine einfache Umsetzung der vereinfachten Funktion F = A'C + B'C + AB C' erreicht man mit NOT-, AND- und OR-Gattern (NOR/NAND-Kombination optional, falls man nur NAND- oder NOR-Gates verwenden möchte): - Invertiere A, B, C nach Bedarf: A', B', C'. - Berechne die drei Produkt-Ausdrücke: - X1 = A' AND C - X2 = B' AND C - X3 = A AND B AND C' - Verknüpfe die drei Ergebnisse durch OR: - F = X1 OR X2 OR X3 Dies benötigt mindestens NOT, AND und OR. Falls gewünscht, kann man statt dreier 2-Eingangs-AND-Gatter auch eine 3-Eingangs-AND-Gatter verwenden (für X3: A AND B AND C'). Die Kosten: 3 NOT-Gatter (für A', B', C'), 3 AND-Gatter-Konstrukte (zwei 2-input für X1, X2 und ein 3-input für X3 oder zwei 2-input-Gatter), und ein 3-input OR-Gatter (oder zwei 2-input-OR-Gatter). Diese Konfiguration realisiert die Funktion eindeutig und stabil.