4,5 Punkte

Tipp: Lesen Sie erst den gesamten Lückentext, bevor Sie die ersten Lücken ausfüllen.

In einem nichtleeren Array aus Buchstaben wollen wir den Index eines Buchstaben finden, welcher im Alphabet früher als alle seine Nachbarn im Array liegt. Beispielsweise wären im Array `[a, c, b, d]` die Buchstaben `a` und `b` mögliche Kandidaten. Dazu können wir den folgenden divide-and-conquer Algorithmus verwenden.

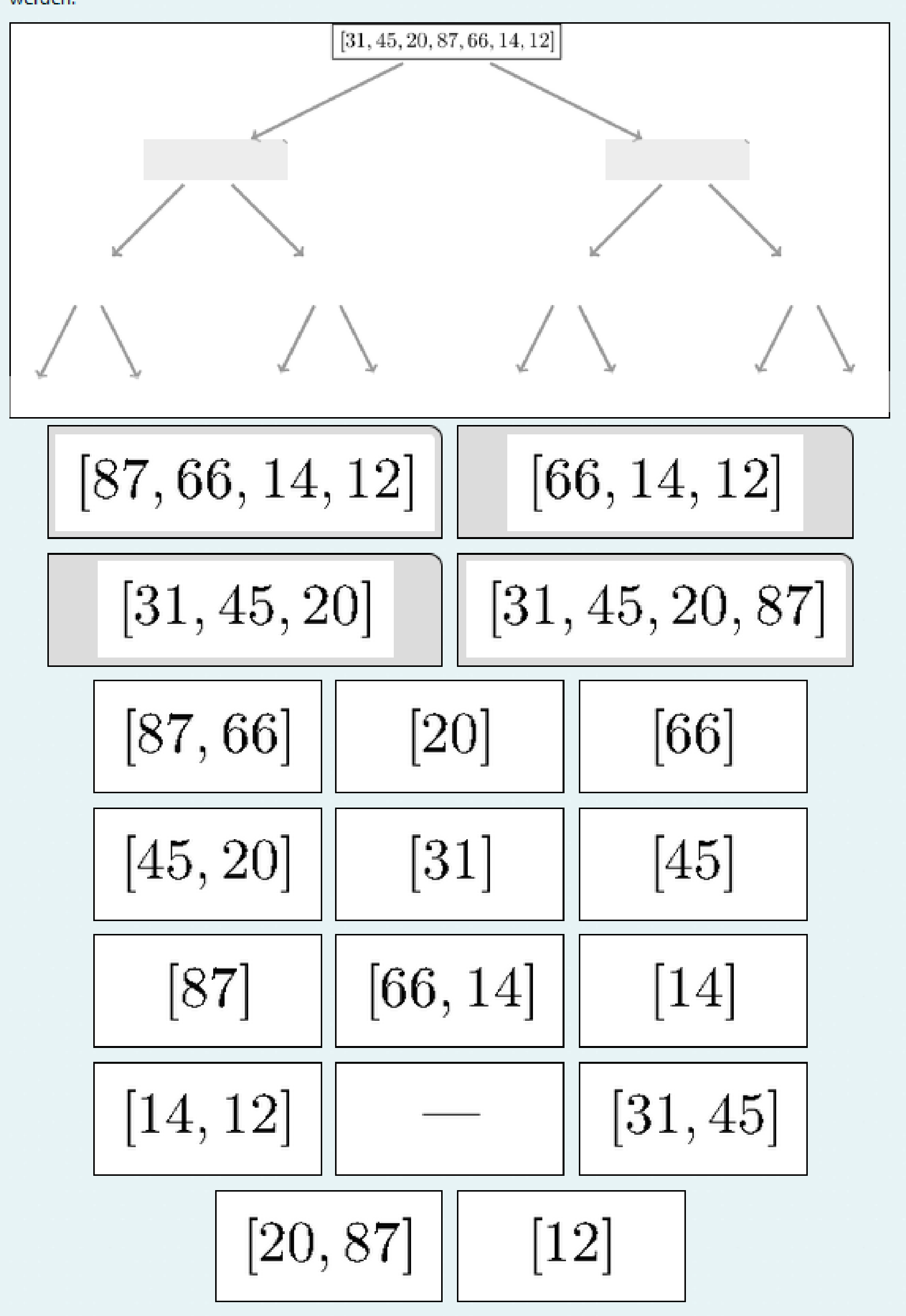
[ ] tritt bei jedem Array [ ] ein, da die Lösung in konstanter Zeit ermittelt werden kann. Darum können wir für [ ] Arrays das Problem [ ] . Dafür betrachten wir das mittlere Element des Arrays und dessen Nachbarn. Sind alle Nachbarelemente später im Alphabet, haben wir eine Lösung gefunden. Andernfalls können wir das Problem [ ] für ein Teilarray von einem alphabetisch früheren Nachbarelement bis zum Ende des Arrays (so, dass das ursprünglich betrachtete Element *nicht* in diesem Teilarray liegt) lösen. Diese Lösung ist dann auch für das ursprüngliche Array korrekt, darum also geschieht [ ] in konstanter Zeit.

Die Anzahl der Teilprobleme, in die wir in jedem Schritt zerlegen, ist also [ ] , jedes von jeweils [ ] Größe. Insgesamt liegt die Laufzeit damit in [ ] .

4,5 Punkte

Vervollständigen Sie den Rekursionsbaum beim Mergesort-Aufruf auf dem Array in der Wurzel. Für Arrays mit ungerade vielen Elementen soll das rechte Teilarray ein Element mehr als das linke Teilarray enthalten.

Jeder Knoten im Baum entspricht einem rekursiven Aufruf, und soll mit dessen Eingabearray gelabelt werden. Der rekursive Aufruf mit allen Elementen *bis* zur Mitte ist das *linke* Kind eines Knoten, der rekursive Aufruf mit den Elementen *ab* der Mitte ist das *rechte* Kind eines Knoten. Beispielsweise würde ein Knoten mit dem Label `[1, 2, 3, 4]` as linkes Kind `[1, 2]`, und als rechtes Kind `[3, 4]` haben. Falls im Diagramm Kindknoten vorgesehen sind, aber der Knoten keine rekursiven Aufrufe erfordert, sollen – (Bindestrich) Label für die Kinder verwendet werden. Insbesondere sollen also **alle** BOXEN ausgefüllt werden.



4 Punkte

Sie finden folgende C-Code Fragmente:

```
 1 typedef struct {
 2     int* data;
 3     int int1;
 4     int int2;
 5 } mystruct;
 6
 7 void foo(mystruct *var1, int value)
 8 {
 9     if(var1->int1 >= var1->int2){ exit(1); }
10     var1->data[var1->int1] = value;
11     (var1->int1)++;
12     return;
13 }
14
15 int bar(mystruct *var1)
16 {
17     if (var1->int1 < 1) { exit(1); }
18     int value = var1->data[var1->int1 - 1];
19     var1->data[var->int1 - 1] = 0;
20     (var1->int1)--;
21     return value;
22 }
23
24 /*
25  * Erstelle die Datenstruktur und reserviere den nötigen Speicher
26  * für capacity Elemente und setze die Hilfsvariablen int1 und int2
27  */
28 mystruct* data_create(int capacity) {
29     ...
30 }
```

Analysieren Sie den Code und beantworten Sie folgende Fragen. Gehen Sie dabei davon aus, dass jeglicher Speicher erfolgreich allokiert werden konnte und keine Zugriffe auf ungültige Speicherbereiche stattfinden.

Gegeben folgender Programmcode.

```
 1 int main(int argc, char** args)
 2 {
 3     mystruct* data = data_create(7);
 4
 5     foo(data, 32);
 6     foo(data, 32);
 7     bar(data);
 8     foo(data, 52);
 9     foo(data, 9);
10     foo(data, 10);
11     /* Zustand ? */
12     printf("Ich bin fertig.\n");
13 }
```

Geben Sie den Zustand der Datenstruktur an, nachdem Zeile 10 ausgeführt wurde, aber das Programm noch ausgeführt wird. Achten Sie darauf, wie die Speicheradressen wachsen. Fügen Sie die Zahlen für den Wert der in den Elementen der Datenstruktur (untere Reihe) gespeichert ist per **Drag&Drop** ein. Benennen Sie die Pointer oberhalb der Elemente. Falls Elemente nicht belegt oder Pointer nicht beschrieben sind, nutzen Sie die "-" Elemente.

Pointer:

Elemente:

0x..24 0x..20 0x..1c 0x..18 0x..14 0x..10 0x..0c 0x..08 0x..04 0x00

9 | data+int1 | Root | data+int2 | Head | 10 | data | Tail | data+int3 | 52 | 32 | Next

-

1 Punkt

Der oben gezeigte C-Code implementiert folgende abstrakte Datenstruktur: [ ] .

1 Punkt

Der oben gezeigte C-Code benutzt folgende Implementierung der Datenstruktur: [ ] .

Aufgabe: Programmierung

Peter Pan und Wendy haben es sich zum Hobbie gemacht, ihrem Erzfeind Kapitän Hook regelmäßig seine neuen Goldschätze zu entwenden und zu verstecken. Sie haben ein System entwickelt, den Standort mithilfe einer Zahlenfolge zu kodieren. Sie senden den Standort an die anderen verlorenen Kinder per E-Mail, damit diese die Schätze bergen können.

Die Zahlenfolge $[43, 17, 70, -14, 20]$ bedeutet, dass ausgehend vom geheimen Versteck der verlorenen Kinder, der Schatz zu finden ist, wenn man 43m nach Norden, dann 17m nach Osten, 70m nach Süden, -14m nach Westen (also 14m nach Osten) und zu guter Letzt wieder 20m nach Norden geht. Das heißt, dass sich die Richtungen zyklisch wiederholen.

Peter und Wendy haben ein System entwickelt, um die Zahlenreihen sicher an die verlorenen Kinder übersenden zu können. Sie haben sich eine geheime Zahl key ausgedacht und mit den verlorenen Kindern geteilt. Bevor sie die Zahlenfolge mailen, wird diese mit dem key verändert. Die o.a. Liste würde z.B. mit dem angenommenen $key = 42$ addiert, dann ergibt sich für die Übertragung $[43 + 42, 17 + 42, 70 + 42, -14 + 42, 20 + 42] = [85, 59, 112, 28, 62]$.

Implementieren Sie die unten geforderten Funktionen.

Sie können die Aufgaben unabhängig voneinander bearbeiten, wir empfehlen jedoch die vorgeschlagene Reihenfolge.

Die Vorlage-Dateien und Teilaufgaben finden Sie in der Detailbeschreibung unten.

Allgemeine Vorgaben:

- Achten Sie darauf, dass Ihre Implementierung mit den bekannten Optionen (`-std=c11 -Wall`) fehlerfrei kompiliert und keine Speicherfehler erzeugt.
- Verwenden Sie die hier bereitgestellten Vorlagen und erweitern Sie diese ausschließlich an den dafür vorgesehenen Stellen.
- Beachten Sie auch die korrekte Benutzung dieser Methoden gemäß der Signatur in der Header-Datei aus der Vorlage.
- Verwenden Sie beim Erstellen der Funktionen geeignete Variablennamen.

7 Punkte

Teil 1: Einfache Verschlüsselung

Die folgenden Funktionen sind in der `Vorlage geheimnis.c` zu implementieren. Laden Sie in dieser Aufgabe den Programmcode `geheimnis.c` hoch. Sie können mit der in der Vorlage ausgelieferten Funktion `void print_integer_sequenz(integersequenz* i_seq)` Beispiele auf der Konsole ausgeben lassen.

a) Erstellen Sie die Funktionen `delta()` [2 Punkte]

Erstellen Sie die Funktion mit der Signatur `int delta(integersequenz*)`:

- Die der Funktion übergebene integersequenz darf nicht geändert werden.
- Die Funktion soll die in nördlicher Richtung zurückgelegte Distanz bestimmen.

b) Erstellen Sie die Funktionen `encrypt()` [2 Punkte]

Erstellen Sie die Funktion mit der Signatur `integersequenz encrypt(integersequenz*, int)`:

- Die der Funktion übergebene integersequenz darf nicht geändert werden. Der zweite Parameter ist die vereinbarte geheime Zahl.
- Für die Verschlüsselung soll dabei die Addition genutzt werden.
- Reservieren Sie für das Integerarray dynamisch allokierten Speicher.

c) Erstellen Sie die Funktionen `decrypt()` [2 Punkte]

Erstellen Sie die Funktion mit der Signatur `integersequenz decrypt(integersequenz*, int)`:

- Die der Funktion übergebene integersequenz darf nicht geändert werden. Der zweite Parameter ist die vereinbarte geheime Zahl.
- Für die Funktionen soll folgende Eigenschaft gelten $seq = encrypt(decrypt(seq, key), key)$
- Reservieren Sie für das Integerarray dynamisch allokierten Speicher.

d) Erstellen Sie die Funktion `free_integersequenz()` [1 Punkte]

Erstellen Sie die Funktion `void free_integersequenz(integersequenz*)`:

- Geben Sie den dynamisch allokierten Speicher wieder frei. Hinweis: Beachten Sie die Implementierungsdetails der Funktion `integersequenz build_integersequenz(int integers[], int len)` der Vorlage.

Uploads unmittelbar vor Ende der Frist, insbesondere in den letzten Sekunden, können fehlschlagen. Das führt dann dazu, dass diese nicht gewertet werden.
Laden Sie daher Ihre Lösung deutlich vor Ablauf der Frist hoch.

Eine Variante von CountSort in Pseudocode folgt. Der Code dieser Implementierung unterscheidet sich von der in der Vorlesung vorgestellten Version, zeigt jedoch das gleiche Verhalten.

CountSort(Array A)

```
1 // C: 0-gefülltes Array der Länge m (m groß genug)
2 // n: Länge von A
3 for i ← 1 to n do // n inklusiv
4     C[A[i]]++
5 k ← 1
6 for v ← 1 to m do // m inklusiv
7     while C[v] ≠ 0
8         A[k] ← v
9         k++
10        C[v] ← C[v]-1
```

Die Korrektheit dieser Varianten von CountSort ist mit vollständiger Induktion zu beweisen.

1,5 Punkte

$\mathbb{N}$ bezeichnet die Menge an streng positiven Ganzzahlen. Es wird angenommen, dass alle Werte in A in $\mathbb{N}$ enthalten sind. | bezeichnet der Kardinalitätsoperator, d.h., wieviele Elemente es in der Menge gibt, bspw. $|\{1, 5, 2\}| = 3$. j und v sind freie Laufindexe im Array A bzw. Array C. $\Rightarrow$ bezeichnet eine Induktion, bspw. $(v > 1) \Rightarrow (v > 0)$. i ist der Laufindex in A bei der ersten For-Schleife.

Welche Invariante gilt bei der ersten For-Schleife (wo das Array C befüllt wird)? Unterschiede zwischen den einzelnen Antwortmöglichkeiten sind **fett** markiert.

- a. $\mathbf{(1 \leq j \leq i-1) \Rightarrow (A[j] \leq A[j+1])}$
- b. $(1 \leq v \leq m) \Rightarrow (C[v] = |\{j \in \mathbb{N}, 1 \leq j < \mathbf{i} : A[j] = \mathbf{v+1}\}|)$
- c. $(1 \leq v \leq m) \Rightarrow (C[v] = |\{j \in \mathbb{N}, 1 \leq \mathbf{j} \leq \mathbf{i} : A[j] = \mathbf{v}\}|)$
- d. $(1 \leq v \leq m) \Rightarrow (C[v] = |\{j \in \mathbb{N}, 1 \leq \mathbf{j} \leq \mathbf{n} : A[j] = \mathbf{v}\}|)$
- e. $(1 \leq v \leq m) \Rightarrow (C[v] = |\{j \in \mathbb{N}, 1 \leq j < \mathbf{i} : A[j] = \mathbf{v}\}|)$

1,5 Punkte

Um die Schreibweise zu vereinfachen, wird in den Formeln davon ausgegangen, dass k (laufender Index bei der Befüllung von A) streng größer als 1 ist. v ist der Laufindex in C bei der zweiten For-Schleife.

Welche Invariante gilt bei der zweiten For-Schleife (wo das Array A befüllt wird)?

- a. $A[1] \leq \ldots \leq A[k-1]$ und $A[k-1] = v$
- b. $A[1] \leq \ldots \leq A[k]$ und $A[k] < v$
- c. $A[1] \leq \ldots \leq A[k+1]$ und $A[k+1] < v$
- d. $A[1] \leq \ldots \leq A[k]$ und $A[k] = v$
- e. $A[1] \leq \ldots \leq A[k-1]$ und $A[k-1] < v$

2 Punkte

Gegeben seien folgende Code-Ausschnitte.

```c
1 int search(char *h, char *n);
```

search.h

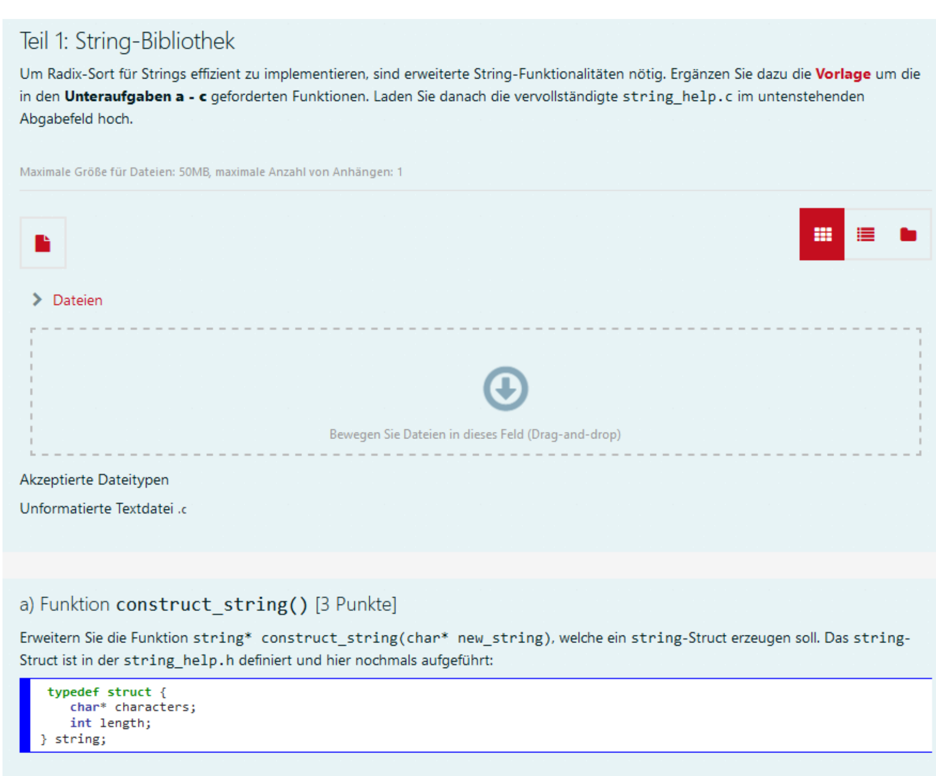
```c
1 #include <stdio.h>
2
3 int search(char *h, char *n)
4 {
5     int match = 0;
6     int count = 0;
7     int pos = 0;
8     int k = 0;
9
10    while (h[pos] != '\0')
11    {
12        match = 1;
13        k = 0;
14        while (h[pos + k] != '\0' && n[k] != '\0')
15        {
16            if (h[pos + k] != n[k])
17            {
18                match = 0;
19            }
20            k++;
21        }
22        if (match == 1 && n[k] == '\0')
23        {
24            count++;
25            printf("%02d. Treffer ab Zeichen %d.\n", count, pos + 1);
26        }
27        pos++;
28    }
29    return count;
30 }
```

search.c

Welche Aussage ist wahr?

- ○ Die Funktion `search()` sucht das letzte Vorkommen der Zeichenkette h in der Zeichenkette n.
- ○ Durch das Einbinden der Header-Datei `search.h` müssen die Funktionen nicht mehr implementiert werden.
- ○ Beim Kompilieren dieser Dateien mit `clang -Wall -std=c11 search.c -o search` wird kein Kompilierungsfehler auftreten.
- ○ Diese beiden Dateien sind syntaktisch korrekt.

3 Punkte



Teil 1: String-Bibliothek

Um Radix-Sort für Strings effizient zu implementieren, sind erweiterte String-Funktionalitäten nötig. Ergänzen Sie dazu die **Vorlage** um die in den **Unteraufgaben a - c** geforderten Funktionen. Laden Sie danach die vervollständigte `string_help.c` im untenstehenden Abgabefeld hoch.

Maximale Größe für Dateien: 50MB, maximale Anzahl von Anhängen: 1

Dateien

Bewegen Sie Dateien in dieses Feld (Drag-and-drop)

Akzeptierte Dateitypen

Unformatierte Textdatei .c

a) Funktion `construct_string()` [3 Punkte]

Erweitern Sie die Funktion `string* construct_string(char* new_string)`, welche ein `string`-Struct erzeugen soll. Das `string`-Struct ist in der `string_help.h` definiert und hier nochmals aufgeführt:

```c
typedef struct {
    char* characters;
    int length;
} string;
```

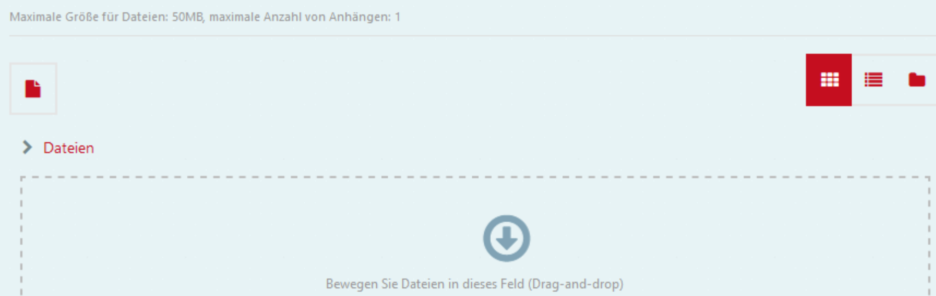
Implementieren Sie folgende Funktionalität:

- Allokieren Sie dynamisch Speicher für das `string`-Struct und das darin enthaltene `characters`-Feld.
- Achten Sie darauf, dass das `characters`-Feld maximal MAX_STRING_LENGTH (siehe `string_help.h`) Felder hat.
- Beenden Sie das Programm mit der Funktion `exit(int <status>)`, falls nicht genügend Speicher allokiert werden kann.
- Achten Sie außerdem darauf, dass jeder String mit einem Null-Terminator enden muss, `strncpy()` diesen aber unter Umständen entfernt.

9 Punkte

Teil 2: Radix-Sort

Implementieren Sie Radix-Sort für Zeichenketten unter der Nutzung der String-Bibliothek aus Teil 1. Erweitern Sie dazu die **Vorlage** um die in den **Unteraufgaben d - f** geforderten Funktionen. Laden Sie danach die vervollständigte `radixsort.c` im untenstehenden Abgabefeld hoch.



Akzeptierte Dateitypen

Unformatierte Textdatei .c

d) Präprozessordirektiven [1 Punkt]

Tragen Sie Ihren Namen und Ihre Matrikelnummer in die dafür vorgesehenen Präprozessordirektiven ein.

e) Radix-Sort [1 Punkt]

Gegeben ist folgender Pseudocode für den Algorithmus Radix-Sort:

```
  RadixSort(Array A)
1 // Array A ist Liste von Strings
2 for j ← MaxStringLength(A) down to 1 do
3     CountSort(A, j, Length(A))
```

Implementieren Sie den Algorithmus innerhalb der Funktion `void radix_sort(string**, int)`. Passen Sie den Aufruf der Count-Sort-Funktion an die Signatur in der Vorlage an.

f) Count-Sort [7 Punkte]

Gegeben ist der Pseudocode einer Variante des Algorithmus Count-Sort. Dieser arbeitet mit zwei Hilfs-Arrays zum Zählen der Häufigkeiten und zur Organisation der zu sortierenden Zeichenketten. Um die Stabilität des Algorithmus zu garantieren, soll die Reihenfolge der Pointer aufrecht erhalten werden.

```
   CountSort(Array A, i, len)
 1 // C ist Hilfs-Array mit 0 initialisiert
 2 // P ist 2-dimensionales Hilfs-Array mit 0 initialisiert
 3 // Length(C) = Length(P) = maximaler Wertebereich
 4 // A[x][1] ist das erste Zeichen in der Zeichenkette A[x]
 5
 6 for j ← 1 to len do
 7     wert ← A[j][i]
 8     P[wert][C[wert]] ← A[j]
 9     C[wert] ← C[wert] + 1
10
11 k ← 1
12 for j ← to Length(C) do
13     m ← C[j] + 1
14     while C[j] > 0 do
15         t ← m - C[j]
16         A[k] ← P[j][t]
17         C[j] ← C[j] - 1
18         k ← k + 1
```

Implementieren Sie den Algorithmus innerhalb der Funktion `void count_sort(string**, int, int)`. Beachten Sie dabei folgende Hinweise:

- Bestimmen Sie einen minimalen Wert als Wertebereich und definieren Sie als Präprozessordirektive `MAX_VALUE`.
- Deklarieren Sie die Hilfs-Arrays `counter` und `pointers` und initialisieren Sie diese mit geeigneten Werten.
- Implementieren Sie die Zählschleife gemäß des Pseudocodes in Zeile 6-9.
- Implementieren Sie die Kopierschleife gemäß des Pseudocodes in Zeile 11-18.
- Achten Sie darauf, dass die Funktion die Stabilitätseigenschaft garantiert und die Zeichenketten unverändert bleiben.

4 Punkte

Gegeben sei eine einfach zyklisch verkettete Liste: Jedes Element zeigt auf das nächste Element, und das letzte Element zeigt auf das erste Element. Angenommen, man verfügt nur über einen Pointer zum ersten Element.
Was ist die kleinste obere Schranke für den asymptotischen Aufwand beim Einfügen eines Elements am Anfang der Liste?

- a. `O(n^2)`
- b. `O(1)`
- c. `O(nlogn)`
- d. `O(n)`

Es werden nur unsortierte Arrays betrachtet, in denen jedes Element maximal zehn Stellen entfernt von der korrekten Position nach einer Sortierung liegt.
Was ist dann die kleinste obere Schranke der Worst-Case-Komplexität von Insertionsort?

- a. `O(n)`
- b. `O(nlogn)`
- c. `O(n^2)`
- d. `O(1)`

Unter welcher Bedingung ist ein Sortieralgorithmus stabil?

- a. Wenn eine beliebige aber endlich große Eingabe übergeben wird, terminiert der Sortieralgorithmus stets nach endlicher Zeit.
- b. Wenn eine beliebige aber endlich große Eingabe übergeben wird, wird die Reihenfolge von gleichwertigen Elementen nach der Sortierung beibehalten.
- c. Wenn eine beliebige aber endlich große Eingabe übergeben wird, hat nach der Sortierung jedes Element einen Wert kleiner als den Wert des darauffolgenden Elements.

Bei einer Variante von Quicksort wird das letzte Element als Pivotelement gewählt, und alle Elemente mit dem gleichen Wert wie das Pivotelement werden in die linke Liste eingefügt.
Ist diese Variante von Quicksort stabil?

- a. Ja, diese Variante von Quicksort ist stabil.
- b. Nein, diese Variante von Quicksort ist instabil.
- c. Es ist nicht klar, ob diese Variante stabil ist.

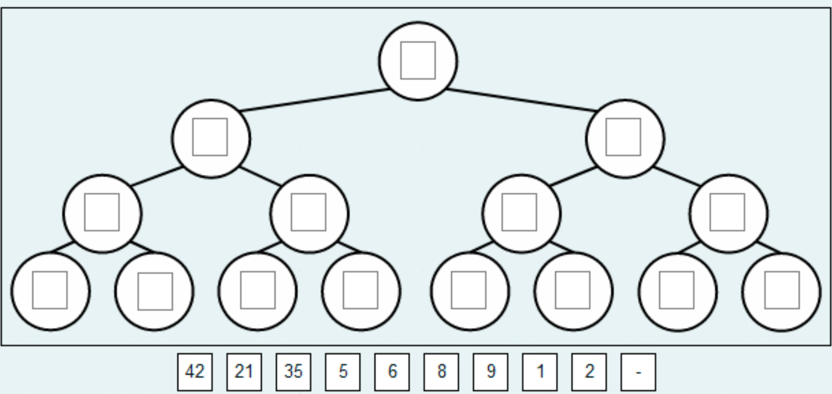
Bei einer Variante von Quicksort wird das Pivotelement zufällig gewählt. Alle Elemente mit dem gleichen Wert wie das Pivotelement, die davor vorkommen, werden in die rechte Liste eingefügt. Alle Elemente mit dem gleichen Wert wie das Pivotelement, die danach vorkommen, werden in die linke Liste eingefügt.
Ist diese Variante von Quicksort stabil?

- a. Ja, diese Variante von Quicksort ist stabil.
- b. Nein, diese Variante von Quicksort ist instabil.
- c. Es ist nicht klar, ob diese Variante stabil ist.

1 Punkt

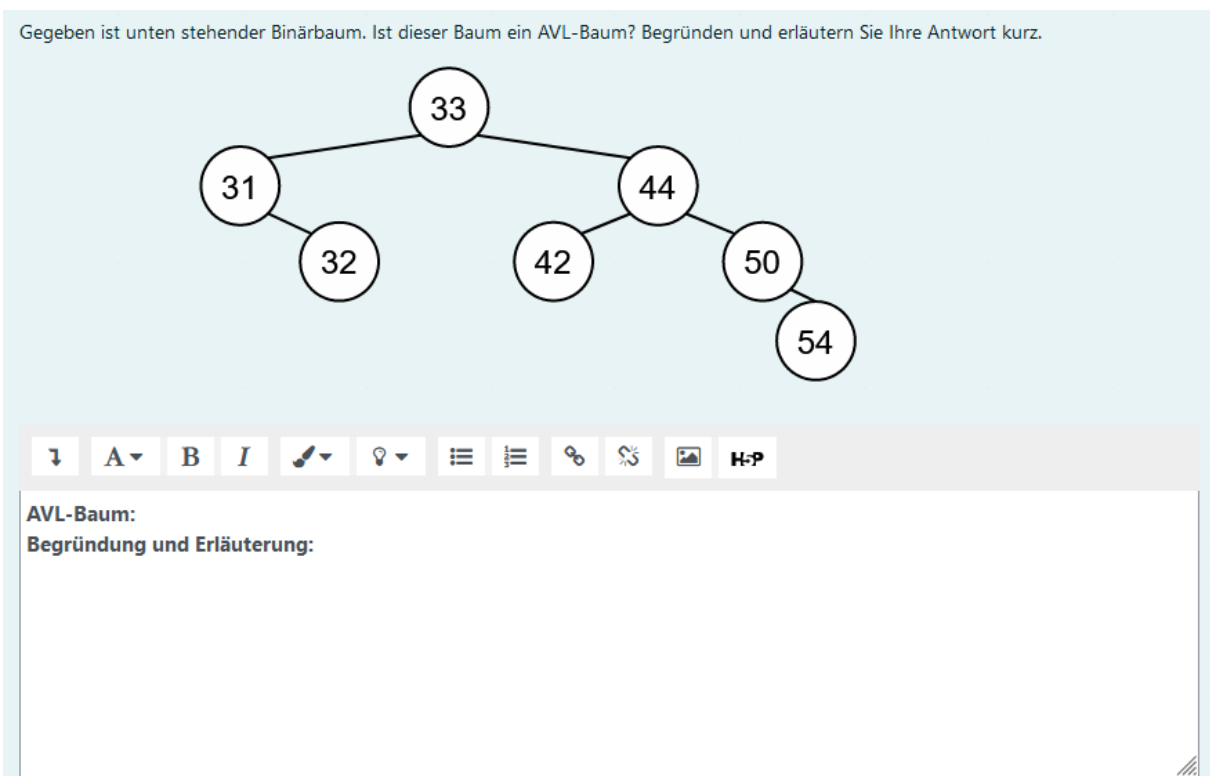
Gegeben ist ein Max-Heap in der Array-Notation: $[42, 21, 9, 8, 6, 1, 5]$. Fügen Sie in diesen Heap mit der *heap_insert*-Funktion die Zahl $\mathbf{35}$ ein.

Geben Sie das Ergebnis in der unten stehenden Baumstruktur an. Besetzen Sie nicht benötigte Knoten mit einem Strich (-).



2 Punkte

Gegeben ist unten stehender Binärbaum. Ist dieser Baum ein AVL-Baum? Begründen und erläutern Sie Ihre Antwort kurz.



5 Punkte

Gegeben ist folgender Algorithmus in Pseudocode:

```
1 for j ← 1 to length(A) do
2     for i ← length(A) downto j+1 do
3         if A[i] < A[i-1] then
4             swap(A, i, i-1)
```

Unten finden Sie eine C-Implementierung, die einer exakten Umsetzung des Pseudocodes entsprechen soll. Die Funktion `printArray` soll dabei den Inhalt eines Integer-Arrays mit anschließenden Zeilenumbruch ausgeben (z.B. `[36,48,2,36,8]`). Der unten stehende Code beinhaltet allerdings vier syntaktische (je 0.5 P) und drei semantische Fehler (je 1 P).

Finden Sie die sieben Fehler und **korrigieren** Sie diese **ausschließlich in derselben Zeile**. Fügen Sie am Zeilenende jeweils einen **Kommentar** ein und **benennen Sie darin die Art des Fehlers (syntaktisch oder semantisch)**.

Hinweis: Die Vorlage Sie hier erneut herunterladen.

```
$include <stdio.h>

void printArray(int* arr, int len) {
    printf("[");
    for(int i = 0; i < len - 1; i++) {
        printf("%d,", arr[i]);
    }
    printf("%d]\n", arr[len -- 1]);
}

void swap(int arr[], int i, int j) {
    int tmp = arr[i];
    arr[i] -= arr[j];
    arr[j] = tmp;
}

void sort(int arr[], int len) {
    for(int j = 0; j <= len; j++) {
        for(int i = len - 1; i > j; i--) {
            if(arr[i] > arr[i-1]) {
                swap(arr, i, i+1);

            }
    }
}

int main() {
    int unordered[] =(21,11,5,8,17};
    printf("Start algorithm with input: ");
    printArray(unordered, 5);
    sort(unordered, 5);
    printf("Final result: ");
    printArray(unordered, 5);
    return 0;
}
```

5 Punkte

In einem Programm wurde folgende Variante von Countsort gefunden. Es ist unklar, ob diese Variante korrekt ist.

Die Funktion f rundet eine Zahl auf die nächste Zehnerstelle auf: z.B. $f(353) = 360$. Die Operation $\circ$ ist eine Konkatenation von Arrays, z.B.: $[0,1] \circ [1,6,1] = [0,1,1,6,1]$. A sei das Eingabearray, A' das Ausgabearray, n die Länge des Eingabearrays und m die Anzahl an Indizes im Wertearray (die Werte sind Ganzzahlen).

```
CountSort1(A,f)
    for i ← 1 to n do
        w ← A[i]
        w' ← f(w)
        j ← w'+1
        counts[j]++
    for j ← 1 to m do
        w' ← j-1
        for k ← 1 to counts[j] do
            A' = A' ∘ [w']
    return A'
```

Wie lautet die kleinste obere Schranke der Laufzeit von `CountSort1()`?

a. $O(m)$

b. $O(n+m)$

c. $O(n)$

d. $O(n^2)$

Ein Sortieralgorithmus ist korrekt, wenn für eine beliebige Eingabe deren Werte in richtiger Sortierung ausgegeben werden.
Ist `CountSort1()` ein korrekter Sortieralgorithmus?

a. Man kann nicht wissen, ob `countSort1()` ein korrekter Sortieralgorithmus ist.

b. Nein, `countSort1()` ist kein korrekter Sortieralgorithmus.

c. Ja, `countSort1()` ist ein korrekter Sortieralgorithmus.

Es sei A_0 ein Array, das aus A entsteht, indem jeder Wert durch sein nächstes Vielfaches von 10 ersetzt wird. A_0 wird nun als Eingabe für `CountSort1()` verwendet.
Gibt `CountSort1()` die Werte von A_0 in richtiger Reihenfolge aus?

a. Nein, `CountSort1()` gibt die Werte von A_0 in falscher Reihenfolge aus.

b. Man kann nicht wissen, ob `CountSort1()` die Werte von A_0 in richtiger Reihenfolge ausgibt.

c. Ja, `CountSort1()` gibt die Werte von A_0 in richtiger Reihenfolge aus.

Es wird angenommen, dass Werte im Eingabearray mehrfach vorkommen dürfen und die Funktion $pos()$ gibt die erste Position eines Wertes im Array zurück, z.B. $pos(5, A) = 2$ für $A = [37, 5, 13, 5, 4]$.
Welche weitere Aussage trifft nach der Terminierung von `CountSort1()` für beliebige i, i' aus dem Intervall $[1, n]$ zu?

a. $(f(A[i]) < f(A[i'])) \Leftarrow (pos(f(A[i]), A') < pos(f(A[i']), A'))$

b. $(f(A[i]) > f(A[i'])) \Rightarrow (pos(f(A[i]), A') < pos(f(A[i']), A'))$

c. $(A[i] < A[i']) \Leftrightarrow (pos(f(A[i]), A') < pos(f(A([i']), A'))$

d. $(f(A[i]) < f(A[i'])) \Rightarrow (pos(f(A[i]), A') < pos(f(A[i']), A'))$

4,5 Punkte

Gegeben ist folgender Algorithmus als Pseudocode:

```
function(Array A, i, j)
01. if i >= j then
02.     return
03. m ← ⌊ (i + j) / 2 ⌋
04. function(A, i, m)
05. function(A, m+1, j)
06. if A[j] > A[m] then
07.     tmp ← A[j]
08.     A[j] ← A[m]
09.     A[m] ← tmp
10.     printArray(A, length(A))
11. function(A, i, j-1)
```

Führen Sie eine Handsimulation des oben angegebenen Algorithmus durch. Der initiale Aufruf erfolgt mit **function(A, 1, 5)** und der Wertebelegung **[15, 14, 50, 19, 17]** für das Array A.

Geben Sie die Belegung des Arrays in Zeile 1 an, nachdem die Funktion zum ersten Mal aufgerufen wurde.

A[1]	A[2]	A[3]	A[4]	A[5]

Geben Sie für jeden Durchlauf die Belegung des Arrays in Zeile 10 an.

A[1]	A[2]	A[3]	A[4]	A[5]