Klausur 2 – 30.09.2015

(90 Min., keine Hilfsmittel außer nicht-programmierbarer Taschenrechner)

Name, Vorname: [redacted] Matrikelnr.: [redacted]

[letzte Prüfungsmöglichkeit wg. 3. Versuch, zutreffenden falls ankreuzen: ❑]

1. Aufgabe
Geben Sie die formatierte Ausgabe des folgenden Programm-Fragments an: 3 /4

```
unsigned short i = 0xbeef;
unsigned short n = ( i << 7 ) | ~0x6dce;
printf("%x\n",n);
```

2. Aufgabe
Erläutern Sie kurz, was unter einem *Abstrakten Datentyp* (*abstract datatype, ADT*) verstanden wird und warum diese eingesetzt werden. 6 /6

3. Aufgabe
Erklären Sie kurz, was im Zusammenhang einer als *ADT* entworfenen *Listen*-Datenstruktur ein *Iterator-Interface* ist und warum es benötigt wird.
Beschreiben Sie, welche Funktionen typischerweise dieses Interface bilden. 5 /6

4. Aufgabe
Gegeben sei eine einfach verkettete, lineare Liste mit den Definitionen 11 /12

```
typedef struct lnode_t *llink;
struct lnode_t {
      int data;          /* payload */
      llink next;        /* pointer to successor */
};
```

sowie einem `llink head` auf den Kopfknoten der Liste, im letzten Listen-Knoten der Kette ist `next==NULL`. Schreiben Sie eine nicht-rekursive Funktion `reverseList`, die den Kopfknoten erhält und alle Knoten der Liste in umgekehrter Reihenfolge anordnet. Die Liste darf dazu nur einmal durchlaufen werden.

5. Aufgabe
Begründen Sie bzgl. der Datenstruktur eines *binären Suchbaums* (*binary search tree, BST*), warum für eine *level-order*-Traversierung keine *stack*-Implementierung möglich ist und welche meist stattdessen verwendet wird. Erläutern Sie den entscheidenden Unterschied zu den gewöhnlich rekursiv implementierten, *stack*-basierenden Traversierungsmethoden (z. B. *pre-order*, *in-order*, etc.). 0 /8

6. Aufgabe
Beschreiben Sie kurz, was im Zusammenhang der Behandlung von Kollisionen in Hash-Tabellen unter *open addressing* und *separate chaining* verstanden wird. 4 /4

Übertrag: 29 /40

Übertrag: 29 /40

7. Aufgabe

Gegeben sei ein *binärer Suchbaum* (*binary search tree*, *BST*) mit den Definitionen

```
typedef struct tnode_t *tlink;
struct tnode_t {
      void *data;             /* pointer to payload */
      unsigned int height;    /* node's height */
      unsigned int subsize;   /* subtree size */
      tlink left, right;      /* pointers to children */
};
```

sowie einem `tlink root` auf den Wurzelknoten des BST, in den Blätterknoten sind die entsprechenden `left==NULL` bzw. `right==NULL`.

a) Skizzieren Sie einen vollständig ausgeglichenen BST mit 4 *levels* (*level* 0 - 3), tragen Sie als Sortierschlüssel/`data` Großbuchstaben in umgekehrter alphabetischer Reihenfolge ein. 4 /4

b) Geben Sie eine Formel an für die Anzahl `N` der Knoten eines vollständig ausgeglichenen BST als Funktion der Anzahl seiner *level* `L: N = f(L)`. 0 /2

c) Schreiben Sie eine rekursive Funktion `setNodeHeight`, die (für beliebige) BST in jedem Knoten das Attribut `height` auf die Höhe des Knotens setzt. (Dabei haben Blattknoten die Höhe 0, die Höhe des Wurzelknotens entspricht der Anzahl-1 der *level* des BST.) 7 /8

8. Aufgabe

Gegeben sei eine BST-Definition wie in Aufgabe 7 und die bekannte Funktion zur Links-Rotation eines Knotens in einem BST:

```
tlink rotateLeft(tlink node) {
      /* node becomes left subtree of its right child,
       * whose left subtree becomes right subtree of node.
       * Returns the new root, i. e. former right child. */

      tlink rightChild = node->right;
      node->right      = rightChild->left;
      rightChild->left = node;

      return rightChild;
}
```

Schreiben Sie eine erweiterte Version dieser Funktion, die ein `subsize`-Attribut jedes Knotens (s. Aufgabe 7, `subsize`: Größe seines Teilbaums, d. i. die Anzahl der Knoten inkl. des Knotens selbst) bei der Rotation korrekt mitführt, d. h. korrigieren Sie jeweils die vorhandene `subsize` (nur!) der an der Rotation beteiligten Knoten. 7 /8

2,3

03.10.2015, Ra

47 /62

1. i = 6 c e ff

	1011	1110	1110	1111
	6	d	c	e
	1010	1101	1100	1110
~ 6dce	0101	0010	0011	0001
i<<7	0111	0111	1000	0000
i<<7|~6dce	0111	0111	1011	0001

Ausgabe: 77b1 f761

2. Ein ADT ist eine Menge von Daten und eine Sammlung von Operationen auf diesen Daten. Das ADT verwendende Programm wird Klient genannt.
Die Werte, im ADT, werden in abstrakter Form und einer dem Client unbekannten Struktur gehalten (Information Hiding).
Der Zugriff erfolgt über ein Interface. Ein Programm, das den ADT spezifiziert wird Implementierung genannt.

3. Da die vom ADT abstrahierten Werte in einer, dem Client unbekannten Struktur gehalten werden, stellt das Iterator-Interface, als Teil des ADT, Funktion zur Verfügung, die es dem Client erlaubt die abgekapselten Werte zur Weiterverarbeitung zu durchlaufen. über diese zu iterieren.

4. Link reverseList (Link head) {
    Link temp, current;
    while (head -> next != NULL) {
        temp = head -> next;
        head -> next = temp -> next,
        ~~temp -> next = head;~~
        current = temp;

und welche Funktionen?!

5. Aufgabe

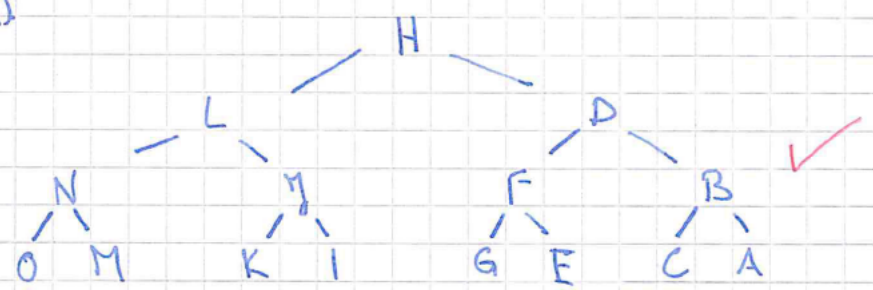
Für ein level-order-Traversierung ist keine stack-Implementierung möglich, da eine Stack-Implementierung, Aufgrund des first in first out Prinzips nur bei rekursiven Traversierungsprinzipien greifen.

6. Beim open adressing wird im Falle einer

6. Beim open addressing wird im Falle einer Kollision, in einem durch die Sondierungsfunktion definierten Intervall die Hashtabelle nach dem nächsten freien Index durchsucht. Die bekanntesten Sondierungsfunktionen sind linear probing, quadratic probing und double hashing ✓

Beim seperate chaining wird ~~im Falle einer Kollision für~~ für alle Kollisionen desselben Index eine Liste genutzt in denen diese abgespeichert werden. Die Liste hat denselben Index. ?
Bildlich beschrieben ist es ein Array von § z.B. Linked Lists. ✓

7. a)

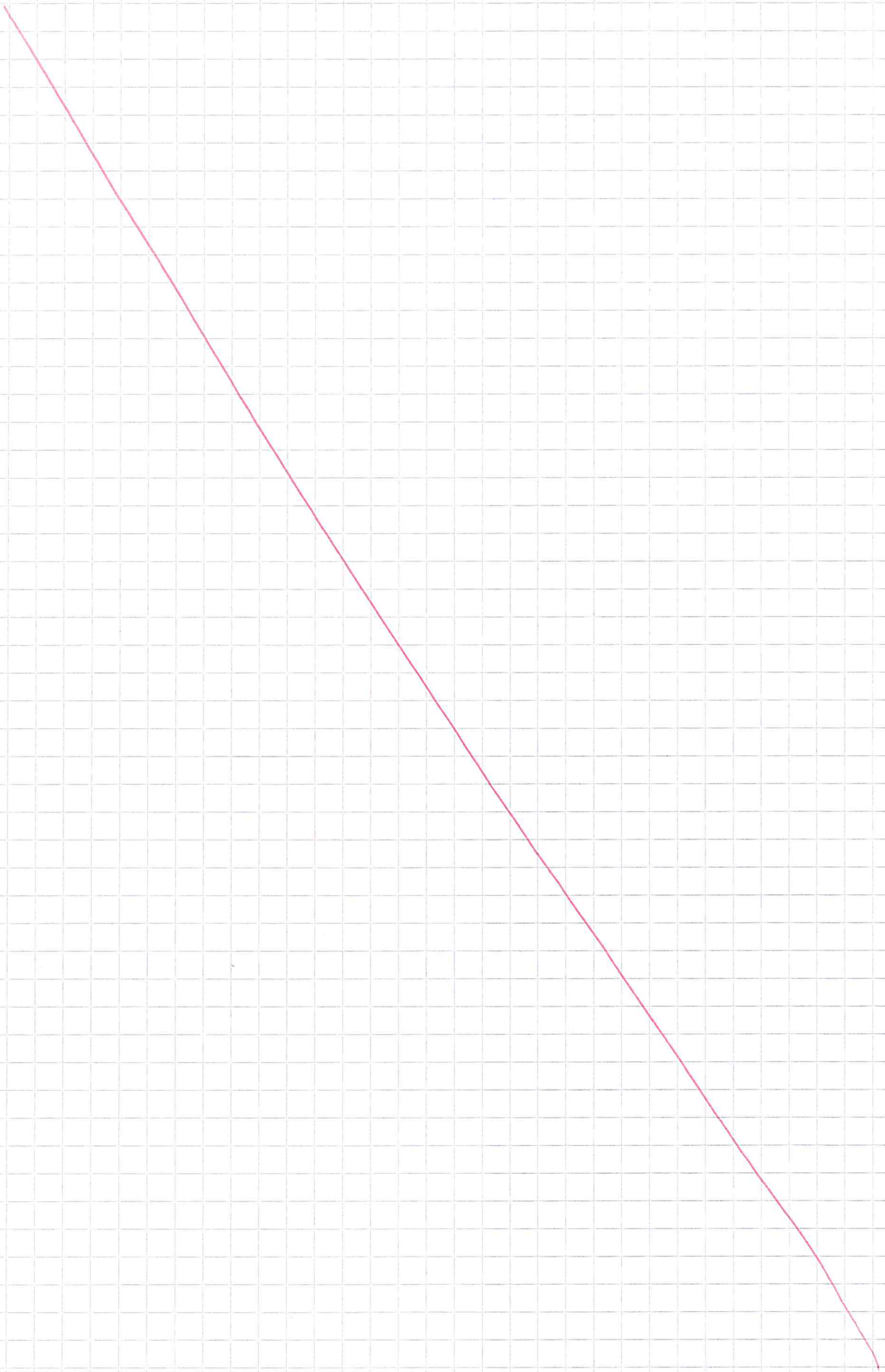


b.) $f(L) = \sum_{x=1}^{L} 1 + (L \cdot 2^{L})$? ? ? ? was soll das heißen?!

c.) 'L' ist (Konstante) unabhängige Var. und kann nicht indiziert werden! $N = 2^L - 1$

```
~~unsigned~~ int
(c) ~~void~~ setNode Height ((link nd ) {
    unsigned int l, r;
    if (nd == NULL) return (0; -1
    (if((nd-> left) == NULL) return 0;
    if (nd->re
    if((nd -> left && nd->right) == NULL) return 0;
    (~~nd =~~
    nd -> height + = 1;
    ~~set N~~
    l = setNode Height ((link nd -> left);
    r = setNode Height ((link nd -> right);
    if (r > l) {
        nd -> height = 1 + r;
    } else {
        nd -> height = 1 + l;
    }
    return nd -> height;
}
```

unnötig, ergibt sich durch rekursive Aufrufe unten von selbst...



8. Aufgabe

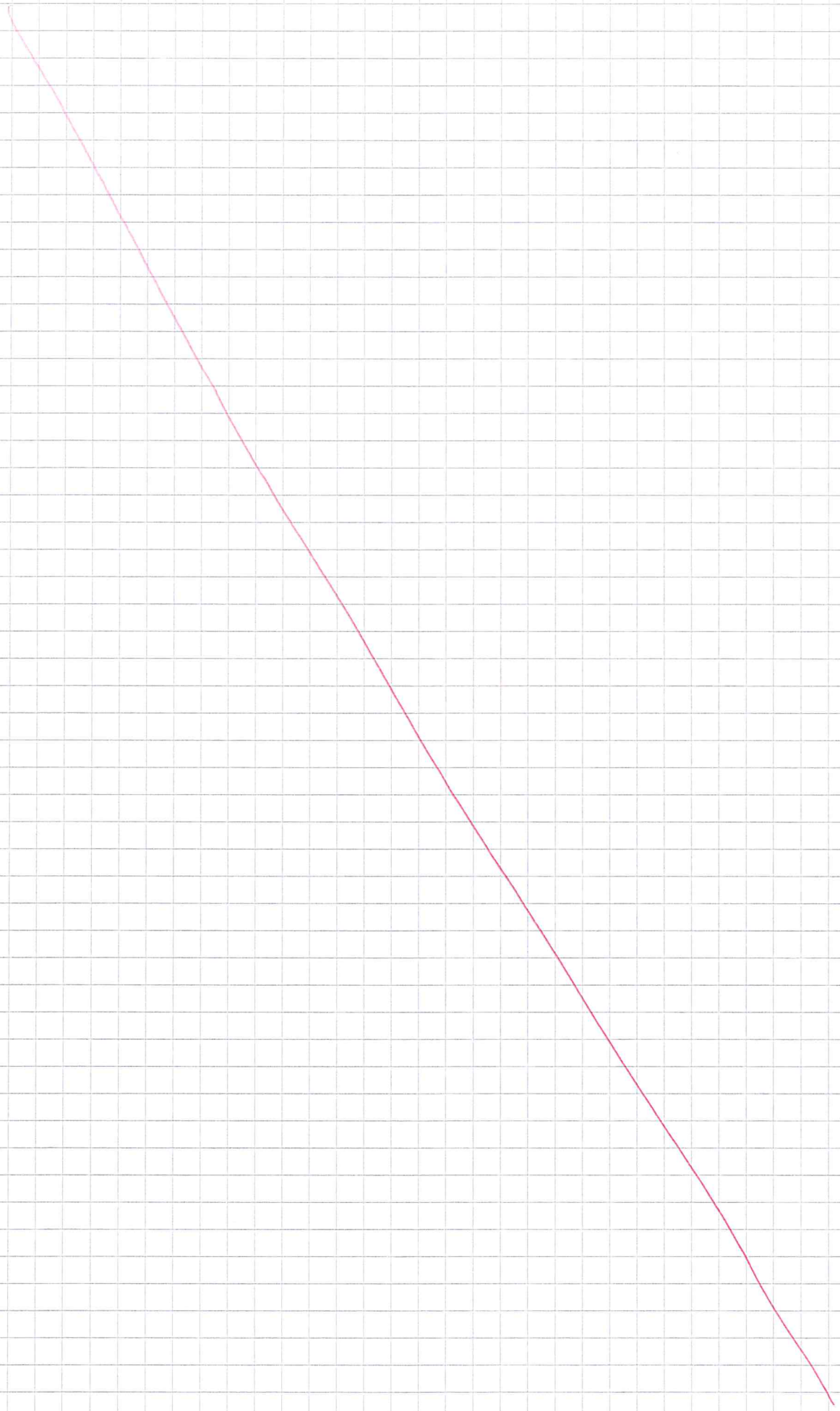
```
tlink rotateLeft(tlink node) {

    tlink rightChild = node -> right;
    node -> right      = rightChild -> left;
    rightChild -> left = node;

    rightChild -> subsize = node -> subsize;
    node -> subsize -= 1 + (rightChild -> right ? rightChild -> right -> subsize : 0);

    return rightChild;
}
```

4. Aufgabe

```
Ulink reverseList(Ulink head) {

    Ulink temp, current;
    if (head == NULL || head -> next == NULL) return NULL;
    temp = head -> next;
    head -> next = temp -> next;

    current = temp;
    current -> next = head;

    while (head -> next != NULL) {

        temp = head -> next;
        head -> next = temp -> next;

        temp -> next = current;
        current      = temp;
    }
    return current;
}
```

ungeprüfte Dereferenzierung eines möglichen NULL-pointers!

etwas verwirrend, 'head' und 'current' zu verwenden, einer hätte auch gereicht ...



5. Aufgabe

Aufgrund dessen, dass Stacks dem first in first out Prinzip folgen, sind sie nur für rekursive ~~Funktionen nutzbar~~ Traversierungen nutzbar.
Der Grundlegende Unterschied zur rekursiven Implementation ist, dass die Geschwisterknoten auf jedem Level präsent sein müssen.

stack: LIFO

~~7. b)~~