

ALGORITHMEN UND DATENSTRUKTUREN

ÜBUNG 6: LISTEN & BÄUME

Eric Kunze

`eric.kunze@tu-dresden.de`

TU Dresden, 20. November 2021

letzte Änderung:
20.11.2021, 15:00

VERKETTETE LISTEN

Wir betrachten *verkettete* Listen.

- Listenelemente
- Verkettungen



VERKETTETE LISTEN

```
1 typedef struct element *list;  
2 struct element {int value; list next};
```

- ▶ `struct element` definiert einen neuen Datentypen
- ▶ `int value` ist der Wert des Listenelements
- ▶ `typedef` definiert nur einen Alias
 - ▷ Wir dürfen jeden Pointer auf ein `struct element` auch einfach `list` nennen.
- ▶ `list next` ist ein Element vom Typ `list`
 - ▷ ein Pointer auf ein `struct element` (das nächste Listenelement)

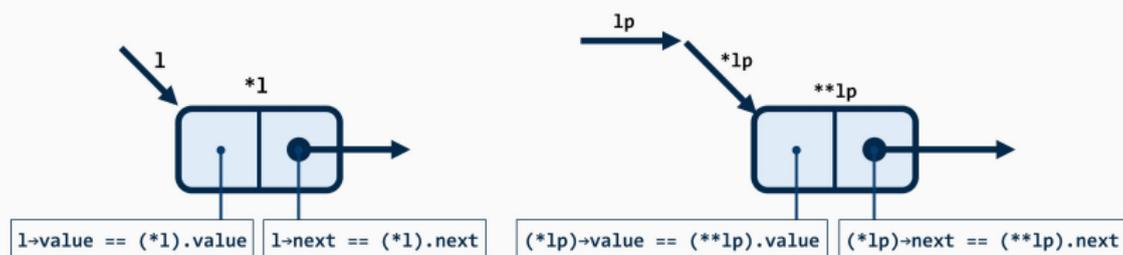
```
struct element { int value; list next }
```



```
typedef struct element *list
```



DIE OPERATOREN `&`, `*`, `->`, `.`

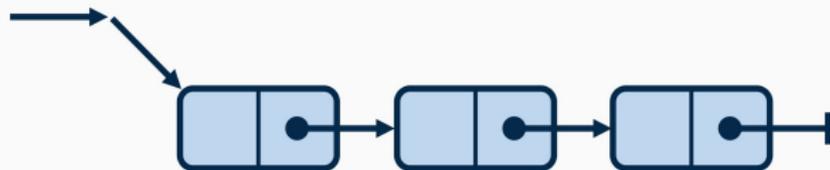


Die Operatoren `&` und `*` binden schwächer als `.` und `->`.

- ▶ `l->value == (*l).value`
- ▶ `&l->next == &((*l).next)`

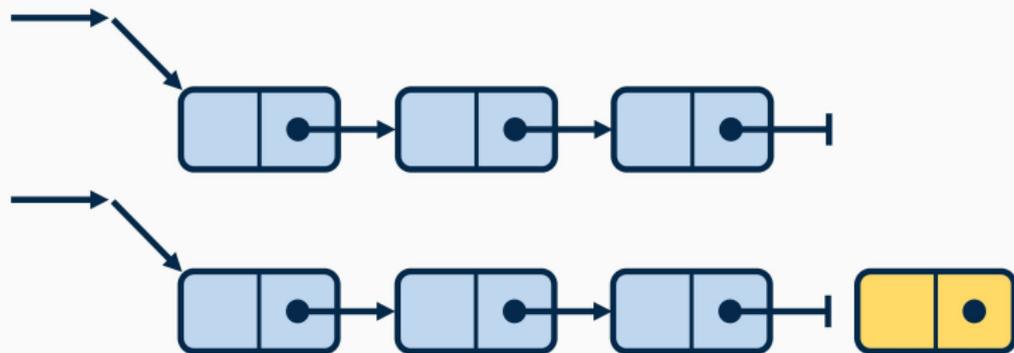
AUFGABE 1 — TEIL (A)

Anfügen an eine Liste



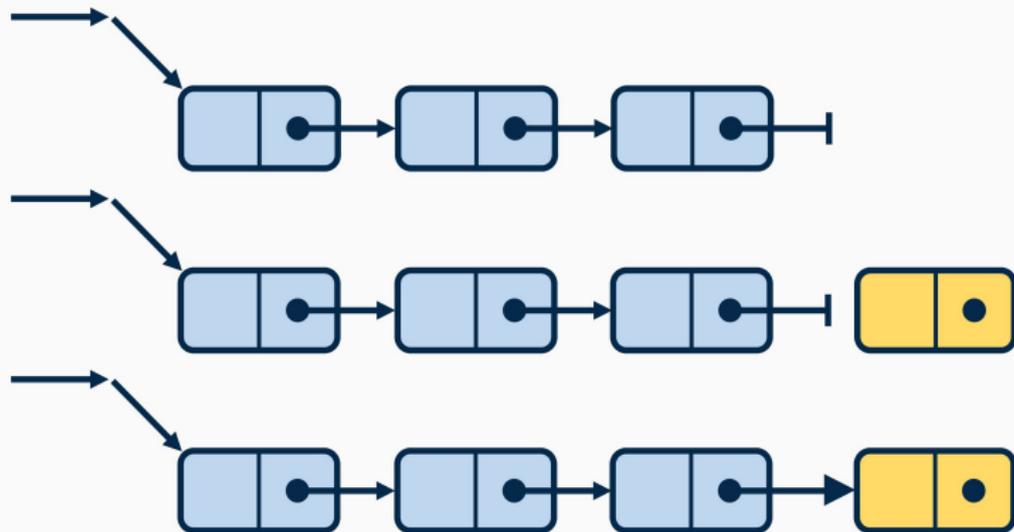
AUFGABE 1 — TEIL (A)

Anfügen an eine Liste



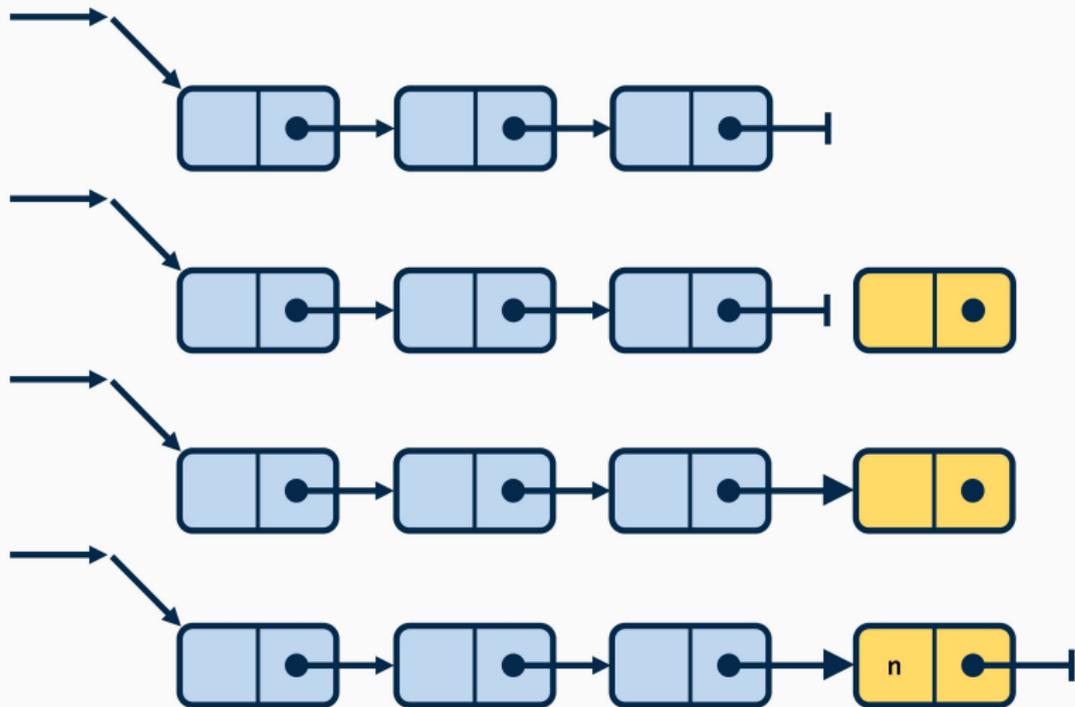
AUFGABE 1 — TEIL (A)

Anfügen an eine Liste



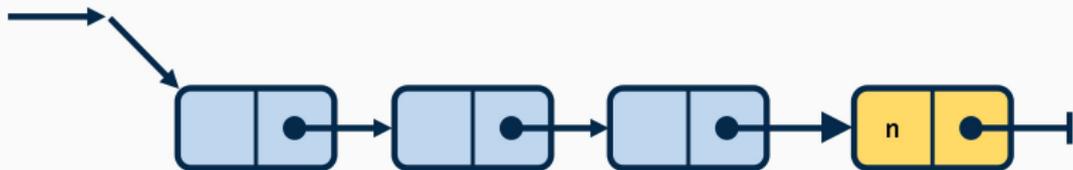
AUFGABE 1 — TEIL (A)

Anfügen an eine Liste

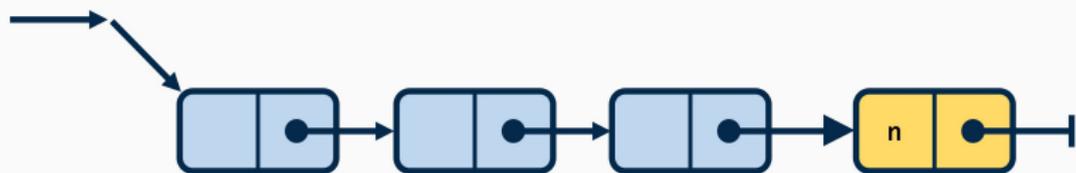


AUFGABE 1 — TEIL (A)

Anfügen an eine Liste



Anfügen an eine Liste



- ▶ Gehe zum Ende der Liste
- ▶ Allokiere neuen Speicherplatz und verknüpfe mit neues Element mit bisheriger Liste
 - ▷ Nachfolger-Pointer des bisherigen letzten Listenelements auf auf neues Listenelement zeigen lassen
- ▶ Fülle das Listenelement mit Schlüsselwert und Nachfolger (= NULL)

Anfügen an eine Liste

- ▶ Gehe zum Ende der Liste
- ▶ Allokieren neuen Speicherplatz und verknüpfe mit neues Element mit bisheriger Liste
 - ▷ Nachfolger-Pointer des bisherigen letzten Listenelements auf auf neues Listenelement zeigen lassen
- ▶ Fülle das Listenelement mit Schlüsselwert und Nachfolger (= NULL)

Anfügen an eine Liste

- ▶ Gehe zum Ende der Liste
- ▶ Allokiere neuen Speicherplatz und verknüpfe mit neues Element mit bisheriger Liste
 - ▷ Nachfolger-Pointer des bisherigen letzten Listenelements auf auf neues Listenelement zeigen lassen
- ▶ Fülle das Listenelement mit Schlüsselwert und Nachfolger (= NULL)

```
1 void append(list *lp, int n){  
2     while(*lp != NULL)  
3         lp = &((*lp)->next) ;  
4     (*lp) = malloc(sizeof(struct element));  
5     (*lp)->value = n;  
6     (*lp)->next = NULL;  
7 }
```

Liste erstellen (mit `append`)

- ▶ erzeuge leere Liste
- ▶ hänge Listenelemente an leere Liste an (durch Aufrufe von `append`)

Liste erstellen (mit `append`)

- ▶ erzeuge leere Liste
- ▶ hänge Listenelemente an leere Liste an (durch Aufrufe von `append`)

```
1 list l = NULL;  
2 append(&l, 4);  
3 append(&l, 2);  
4 append(&l, 0);
```

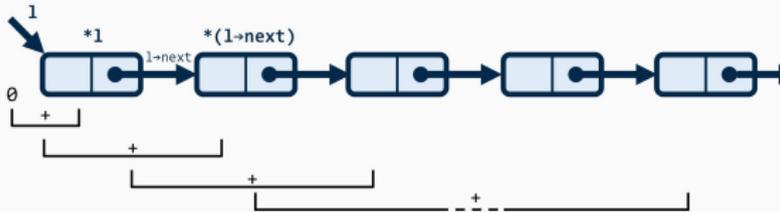
Summe einer Liste — iterativ

- ▶ addiere Schlüsselwerte in Hilfsvariable `result` auf
- ▶ laufe Liste entlang → Pointer immer weiter „schalten“

AUFGABE 1 — TEIL (B)

Summe einer Liste — iterativ

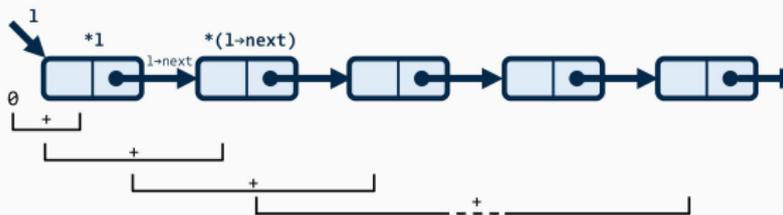
- ▶ addiere Schlüsselwerte in Hilfsvariable `result` auf
- ▶ laufe Liste entlang → Pointer immer weiter „schalten“



AUFGABE 1 — TEIL (B)

Summe einer Liste — iterativ

- ▶ addiere Schlüsselwerte in Hilfsvariable `result` auf
- ▶ laufe Liste entlang → Pointer immer weiter „schalten“



```
1  int sum_it(list l) {  
2      int result = 0;  
3      while (l != NULL) {  
4          result = result + l->value;  
5          l = l->next;    // "start"zeiger weiterschalten  
6      }  
7      return result;  
8  }
```

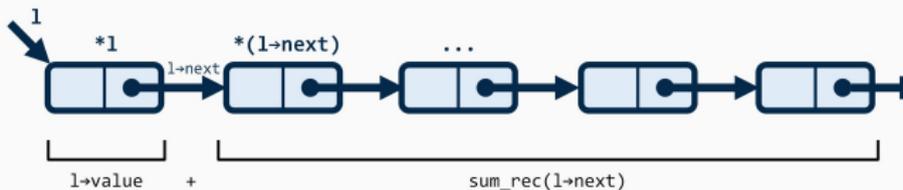
Summe einer Liste — rekursiv

- ▶ definiere Basisfall: leere Liste → Rückgabewert 0
- ▶ rufe Funktion rekursiv auf — mit `next`-Pointer als neuen Startpointer

AUFGABE 1 — TEIL (B)

Summe einer Liste — rekursiv

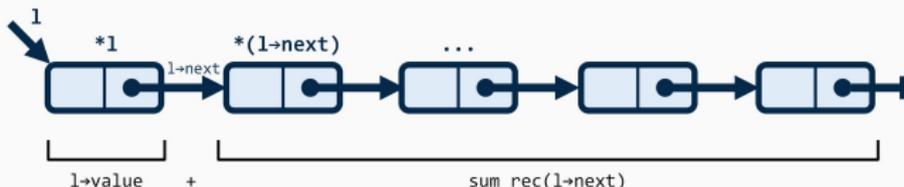
- ▶ definiere Basisfall: leere Liste → Rückgabewert 0
- ▶ rufe Funktion rekursiv auf — mit `next`-Pointer als neuen Startpointer



AUFGABE 1 — TEIL (B)

Summe einer Liste — rekursiv

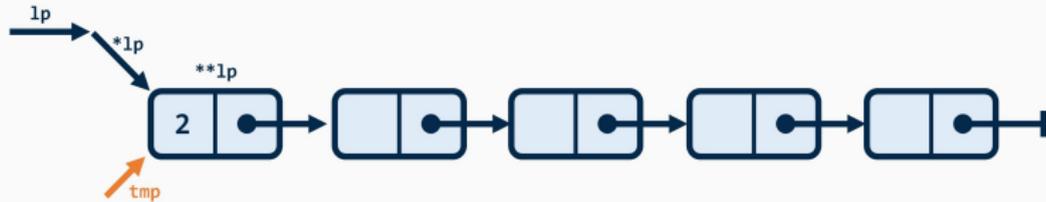
- ▶ definiere Basisfall: leere Liste → Rückgabewert 0
- ▶ rufe Funktion rekursiv auf — mit `next`-Pointer als neuen Startpointer



```
1 int sum_rec(list l) {  
2     if (l == NULL) return 0;  
3     /* nach letztem element nichts mehr addieren */  
4     return l->value + sum_rec(l->next);  
5     /* nimm key und addiere summe der restliste */  
6 }
```

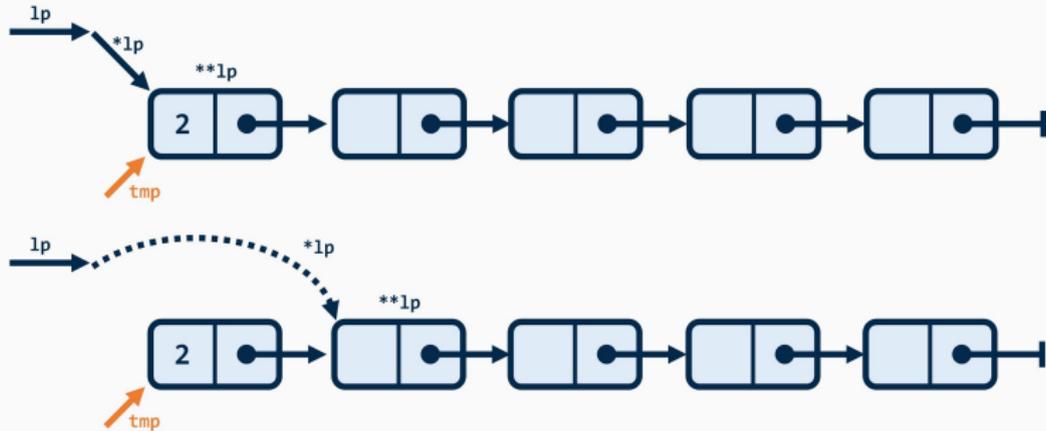
AUFGABE 1 — TEIL (C)

Löschen von Elementen



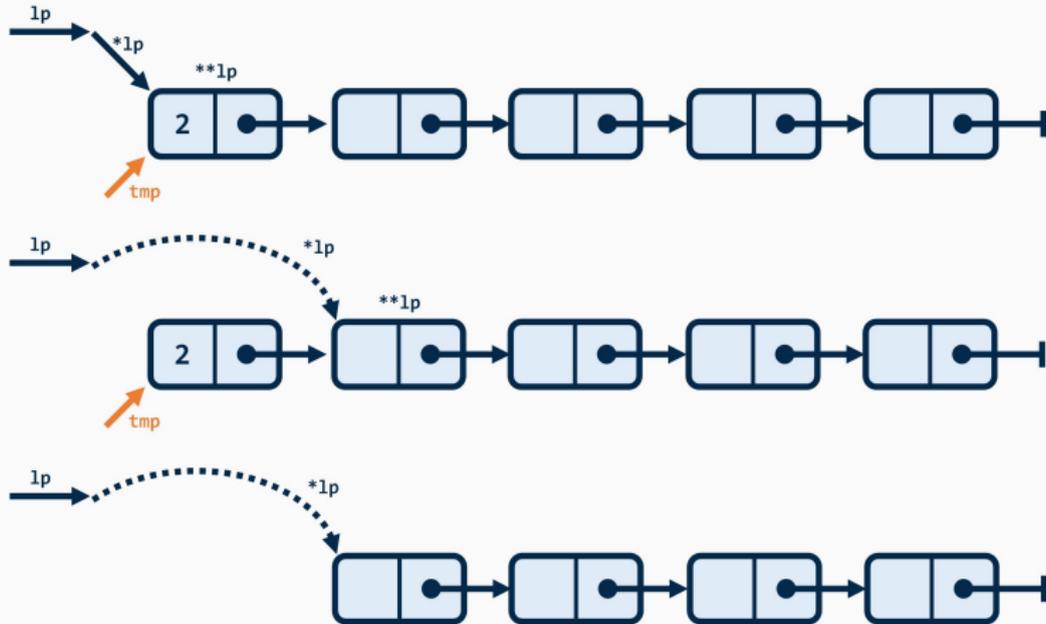
AUFGABE 1 — TEIL (C)

Löschen von Elementen



AUFGABE 1 — TEIL (C)

Löschen von Elementen



Löschen von Elementen — iterativ

- ▶ laufe Liste entlang, solange nicht am Ende angekommen
- ▶ zeigt `*lp` auf einen geraden Schlüssel → *löschen*
 - ▷ merke Zugriff auf `*lp` (zu löschen) in `tmp`
 - ▷ überspringe zu löschendes Element
 - ▷ befreie zu löschendes Element (Zugriff via `tmp`)

Löschen von Elementen — iterativ

- ▶ laufe Liste entlang, solange nicht am Ende angekommen
- ▶ zeigt *lp auf einen geraden Schlüssel → *löschen*
 - ▷ merke Zugriff auf *lp (zu löschen) in tmp
 - ▷ überspringe zu löschendes Element
 - ▷ befreie zu löschendes Element (Zugriff via tmp)

```
1 void rmEvens_it(list *lp) {
2     while (lp != NULL && *lp != NULL) {
3         if ((*lp)->value % 2 == 0) {
4             list tmp = *lp;
5             *lp = (*lp)->next;
6             free(tmp);
7         } else
8             lp = &(*lp)->next;
9     }
10 }
```

Löschen von Elementen — rekursiv

- ▶ Basisfall: keine Liste oder Liste leer → tue nichts
- ▶ Fallunterscheidung bzgl. Schlüsselwert
 - ▷ gerade: löschen & Speicher befreien wie in iterativer Variante
 - ▷ ungerade: überspringe dieses Element
- ▶ verfare so weiter mit der Restliste

Löschen von Elementen — rekursiv

- ▶ Basisfall: keine Liste oder Liste leer → tue nichts
- ▶ Fallunterscheidung bzgl. Schlüsselwert
 - ▷ gerade: löschen & Speicher befreien wie in iterativer Variante
 - ▷ ungerade: überspringe dieses Element
- ▶ verfare so weiter mit der Restliste

```
1 void rmEvens_rec(list *lp) {  
2     if (lp == NULL || *lp == NULL) return ;  
3     if ((*lp)->value % 2 == 0) {  
4         list tmp = *lp;  
5         *lp = (*lp)->next;  
6         free(tmp);  
7     } else  
8         lp = &(*lp)->next;  
9     rmEvens_rec(lp);  
10 }
```

Next Level: Bäume

```
1 typedef struct element *list;  
2 struct element { int value; list next; };
```

```
1 typedef struct element *list;  
2 struct element { int value; list next; };
```

```
1 typedef struct node *tree;  
2 struct node { int key; tree left, right; };
```

AUFGABE 2 — TEIL (A)

**Verknüpfen zweier Bäume mit einem neuen
Wurzelnoten**

Verknüpfen zweier Bäume mit einem neuen Wurzelnoten

- ▶ lege neuen Wurzelknoten an
- ▶ verknüpfe Wurzelknoten mit bestehenden Bäumen
- ▶ fülle neue Wurzel mit Inhalt

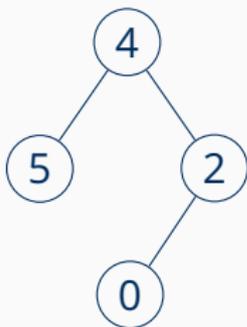
Verknüpfen zweier Bäume mit einem neuen Wurzelnoten

- ▶ lege neuen Wurzelknoten an
- ▶ verknüpfe Wurzelknoten mit bestehenden Bäumen
- ▶ fülle neue Wurzel mit Inhalt

```
1  tree createNode(int n, tree l, tree r) {  
2      tree t    = malloc(sizeof(struct node));  
3      t->left   = l;  
4      t->right  = r;  
5      t->key    = n;  
6      return t;  
7  }
```

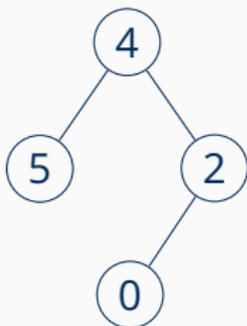
AUFGABE 2 — TEIL (A)

Beispielbaum:



AUFGABE 2 — TEIL (A)

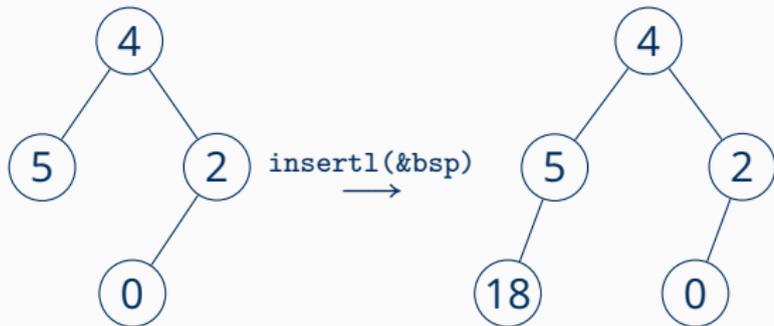
Beispielbaum:



```
1  tree bsp =  
2  createNode(4,  
3  createNode(5, NULL, NULL),  
4  createNode(2,  
5  createNode(0, NULL, NULL),  
6  NULL));
```

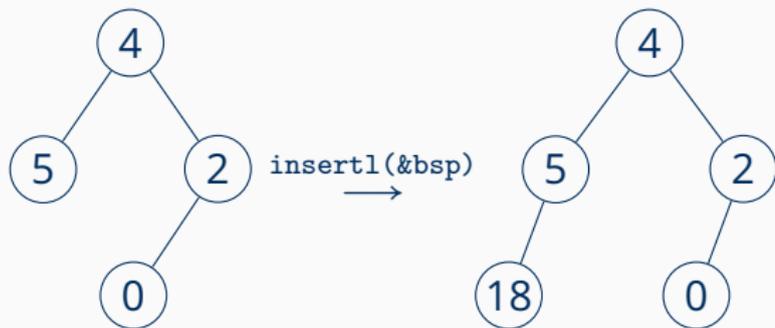
AUFGABE 2 — TEIL (B)

Beispiel:



AUFGABE 2 — TEIL (B)

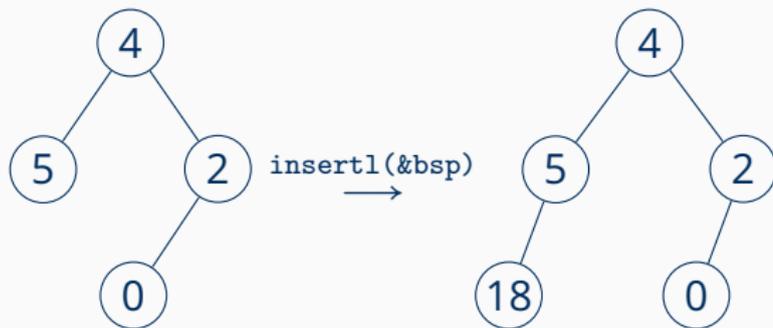
Beispiel:



- ▶ solange kein Blatt erreicht: füge in den linken Teilbaum ein
- ▶ am Blatt: ein neuen Knoten einfügen (`createNode`)

AUFGABE 2 — TEIL (B)

Beispiel:



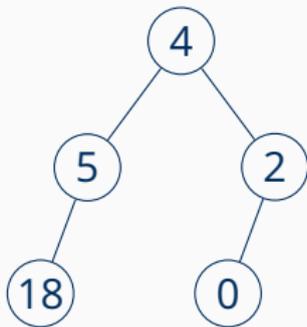
- ▶ solange kein Blatt erreicht: füge in den linken Teilbaum ein
- ▶ am Blatt: ein neuen Knoten einfügen (createNode)

```
1 void insertl(tree *tp, int n) {  
2     if (*tp != NULL)  
3         insertl(&((*tp)->left), n);  
4     else  
5         *tp = createNode(n, NULL, NULL);  
6 }
```

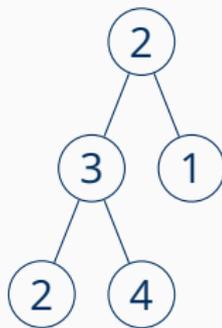
AUFGABE 2 — TEIL (C)

Beispiele:

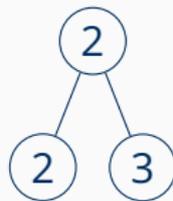
bsp



s



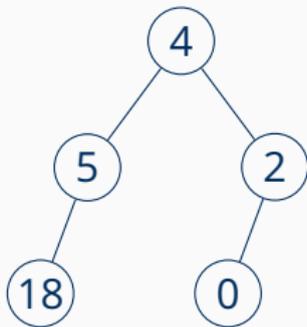
t



AUFGABE 2 — TEIL (C)

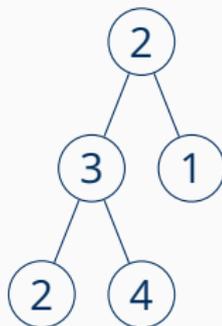
Beispiele:

bsp



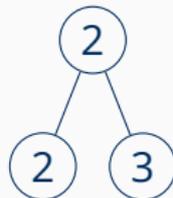
$$\text{leafprod}(\text{bsp}) = 0$$

s



$$\text{leafprod}(s) = 8$$

t



$$\text{leafprod}(t) = 6$$

AUFGABE 2 — TEIL (C)

- ▶ leerer Baum: `leafprod = 1`
- ▶ Rekursion: berechne `leafprod` in linkem und rechtem Teilbaum

AUFGABE 2 — TEIL (C)

- ▶ leerer Baum: leafprod = 1
- ▶ Rekursion: berechne leafprod in linkem und rechtem Teilbaum

```
1  int leafprod(tree t){
2      if (t == NULL)
3          return 1;
4      if (t->left == NULL && t->right == NULL)
5          return t->key;
6      return leafprod(t->left) * leafprod(t->right);
7  }
```

AUFGABE 2 — TEIL (D)

Man übernehme nur gerade Elemente!

Beispiel:



AUFGABE 2 — TEIL (D)

Man übernehme nur gerade Elemente!

Beispiel:



Inorder - Durchlauf:

- ▶ traversiere durch linken Teilbaum
- ▶ Wurzel gerade? → anhängen an Liste mit `append`
- ▶ traversiere durch rechten Teilbaum

AUFGABE 2 — TEIL (D)

```
1 void append(list *lp, int n)
```

AUFGABE 2 — TEIL (D)

```
1 void append(list *lp, int n)
```

```
1 void treeToList_rec(tree t, list *lp){  
2     if (t == NULL) return ;  
3     treeToList_rec(t->left, lp);  
4     if (t->key % 2 == 0)  
5         append(lp, t->key);  
6     treeToList_rec(t->right, lp);  
7 }
```

AUFGABE 2 — TEIL (D)

```
1 void append(list *lp, int n)
```

```
1 void treeToList_rec(tree t, list *lp){  
2     if (t == NULL) return ;  
3     treeToList_rec(t->left, lp);  
4     if (t->key % 2 == 0)  
5         append(lp, t->key);  
6     treeToList_rec(t->right, lp);  
7 }
```

```
1 list treeToList(tree t){  
2     list l = NULL;  
3     treeToList_rec(t,&l);  
4     return l;  
5 }
```