TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Software Entwicklung 1

Annette Bieniusa

AG Softech
FB Informatik
TU Kaiserslautern

Überblick

- Strings in C
- Verwendung von Structs
- Sicherheit von C Programmen: Der Heartbleed-Bug

Strings

Strings in C

- Arrays von Zeichen, am Ende durch ein Nullzeichen `\0` markiert

Beispiel:

```
char name [30]; /* max. 29 Zeichen plus Nullzeichen */
name[0] = 'H';
name[1] = 'u';
name[2] = 'g';
name[3] = 'o';
name[4] = '\0';
printf("%s",name);
```

Fehlerquellen:

- Array wird vollständig mit Zeichen gefüllt wird, sodass kein Platz mehr für das Nullzeichen `\0` bleibt
- Beim zeichenweise Kopieren von Zeichenketten wird `\0` oft vergessen zu kopieren, da das Zeichen zum Testen bei der Abbruchbedingung genutzt wird.

Modifizieren von Strings I

- Stringliterale werden vom C-Compiler im Datenbereich gespeichert
- Nicht modifizierbar (*read-only* Speicherbereicht)

```
1  #include <string.h>
2  #include <stdio.h>
3
4  int main(void)
5  {
6     char *s1 = "Das ist ein String Literal";
7     char s2[] = {'D','a','s',' ','a','u','c','h'};
8     char s3[] = {'S','t','u','d','i','\0','e','r','e','r'};
9
10    printf("%s\n",s1);  // "Das ist ein String Literal"
11    printf("%s\n",s2);  // "Das auch"
12    printf("%s\n",s3);  // "Studi"
13
14    s1[1] = 'x'; // undefiniert!!
15
16    return 0;
17 }
```

Modifizieren von Strings II

- Zum Modifizieren muss der String in Heap oder Stack kopiert werden
- `void *memcpy (void *to, const void *from, size_t n)`

```
1  #include <string.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  int main(void)
6  {
7    char *old = "C sucks";
8    char *new = malloc(8 * sizeof(char));
9    memcpy (new, old, 8 * sizeof(char));
10
11   new[2] = 'r';
12   new[3] = 'o';
13   printf("\"%s\" wird zu \"%s\"\n",old, new);
14   free(new);
15   return 0;
16 }
```

Länge von Strings

- `size_t strlen(const char *s)` liefert die Länge des adressierten Strings `s` **ohne** das Nullzeichen.

```
1  int main(void)
2  {
3    char s[] = "I <3 C";
4    int length = strlen(s);
5    printf("Der String \"%s\" hat %d Zeichen\n", s, length);
6
7    // Achtung: strlen liefert die Position des ersten \0
8    //  und nicht die Groesse des Arrays
9    char s3[] = {'S','t','u','d','i','\0','e','r','e','r'};
10   printf("Der String \"%s\" hat %lu Zeichen\n", s3, strlen(s3))
       ;
11   printf("Das Array s3 hat %lu Bytes\n", sizeof(s3));
12
13   return 0;
14 }
```

Konkatenation von Strings

- Funktion `char* strcat(char* s1, const char* s2)`
- `\0` von `s1` wird dabei überschrieben
- Voraussetzung: Speicherbereich für `s1` ausreichend groß zur Aufnahme von `s2`
- Alternative: `char* strncat(char* s1, const char* s2, size_t n)`
- Für überlappende Speicherbereiche ist das Verhalten nicht definiert.

```
1  int main(void)
2  {
3     const char x[] = "Urin";
4     const char y[] = "stinkt";
5
6     char* z = malloc(strlen(x) + strlen(y) + 1);
7     memcpy (z, x, strlen(x));
8     strncat(z, y, strlen(y));
9     printf("%s\n",z);
10    free(z);
11    return 0;
12 }
```

Vergleichen von Strings I

- Funktion `int strcmp(char* s1, char* s2)` durchläuft Zeichen für Zeichen und vergleicht die ASCII-Codes der Zeichen
- s1 identisch mit s2 $\Rightarrow$ liefert den Wert 0
- Erster ungleicher Buchstabe in s1 hat einen größeren (kleineren) ASCII-Code als der entsprechende Buchstabe in s2
  $\Rightarrow$ Rückgabewert kleiner als 0 (größer als 0)

```
1  int main(void)
2  {
3    char s1[] = "C rocks";
4    char s2[] = "C sucks";
5
6    int not_equal = strcmp(s1,s2);
7    if(not_equal) {
8      printf("\"%s\" und \"%s\" sind unterschiedlich",s1,s2);
9    } else {
10     printf("\"%s\" und \"%s\" sind gleich",s1,s2);
11   }
12
13   return 0;
14 }
```

Vergleichen von Strings II

- Funktion `int strncmp(const char *x, const char *y, size_t n)` vergleicht die ersten `n` Zeichen von `x` und `y` miteinander
- Sicherere Alternative!

```
1  int main(void)
2  {
3    const char x[] = "abce";
4    const char y[] = "abcd";
5
6    for(int i = strlen(x); i > 0; --i) {
7      if(strncmp( x, y, i) != 0){
8        printf("Die ersten %d Zeichen der beiden Strings sind nicht gleich
       \n", i);
9      } else {
10       printf("Die ersten %d Zeichen der beiden Strings sind gleich\n", i
       );
11       break;
12     }
13   }
14   return 0;
15 }
```

Vergleichen von Strings III

Ausgabe:

```
Die ersten 4 Zeichen der beiden Strings sind nicht gleich
Die ersten 3 Zeichen der beiden Strings sind gleich
```

Umwandlung von Strings in Zahlenwerte

- Hilfsfunktionen zur Umwandlung von Strings in Zahlenwerte in `stdlib.h`
- Funktion `double atof(const char *s)` wandelt String `s` in double
- Funktion `int atof(const char *s)` wandelt String `s` in int

Strukturen

Strukturen in C

- Zusammenfassen von Daten unterschiedlicher Typen

```
struct datum
 {
   int tag;
   char monat[10];
   int jahr;
 } ;
typedef struct datum datum_t;
```

Erzeugen von Instanzen einer Struktur

Statisches Allozieren auf dem Stack:

```
datum_t d1 = {5, "Juli", 1987};
d1.tag = 15;
```

Dynamische Allozieren während des Programmlaufs auf dem Heap:

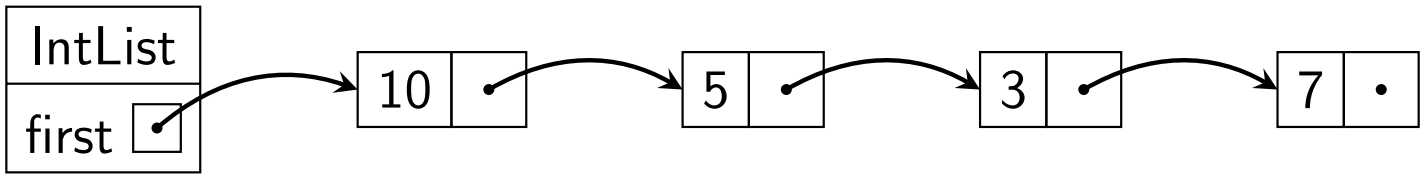
```
// liefert Zeiger auf die Struktur
datum_t* d2 = malloc(sizeof(datum_t));

d2->tag = 5;
d2->jahr = 1987;
// Alloziert den String mit den anderen Struktur-
    Komponenten auf dem Heap
strcpy(d2->monat, "Juli");

free(d2);
```

Adressieren der Komponenten

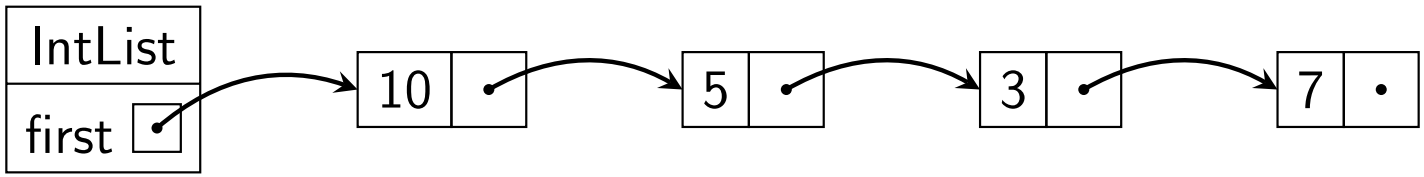
```
datum_t geburtstag = {5, "Juli", 1987};
// Zugriff auf Komponenten ueber .
geburtstag.tag = 15;

//d ist Zeiger auf geburtstag
datum_t *d = &geburtstag;
//Wert von d ist die Struktur, daher hier Zugriff ueber .
(*d).tag = 1950;
// syntaktische Alternative
d->tag = 1950;
```

Beispiel: Einfach verkettete Listen

Wiederholung: Einfachverkettete Liste



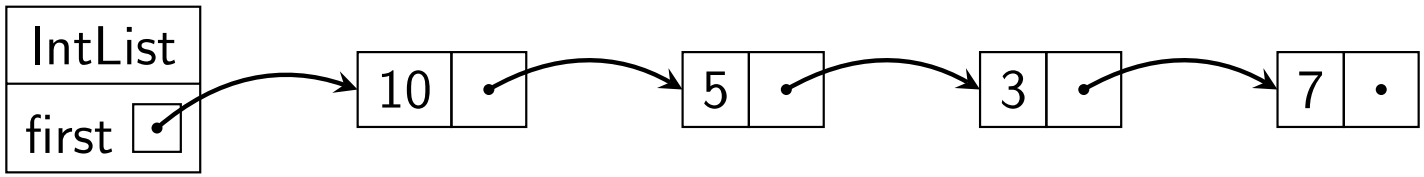
Wiederholung: Einfachverkettete Liste



in Java:

```
class Node {
    private int value;
    private Node next;
    ...
}
public class IntList {
  private Node first;
  ...
}
```

Wiederholung: Einfachverkettete Liste



in Java:

```
class Node {
    private int value;
    private Node next;
    ...
}
public class IntList {
  private Node first;
  ...
}
```

in C:

```
typedef struct node node_t;
struct node
{
  int value;
  node_t *next;
};

typedef struct linked_list
        linked_list_t;
struct linked_list
{
  node_t *first;
};
```

Erstellen einer Liste

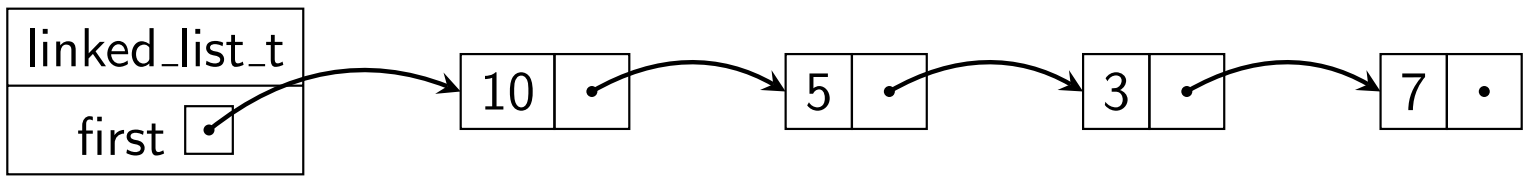
```
linked_list_t *new_list(void)
{
  // Alloziere Speicher auf dem Heap fuer die Liste
  linked_list_t *ll = malloc(sizeof(linked_list_t));
  if (NULL != ll)
  {
    // Initialisere die Komponenten der Liste
    ll->first = NULL;
    return ll;
  }
  else
  {
    printf("Couldn't allocate linked_list\n");
    abort(); // Programmabbruch
  }
}
```

Erstellen einer Liste

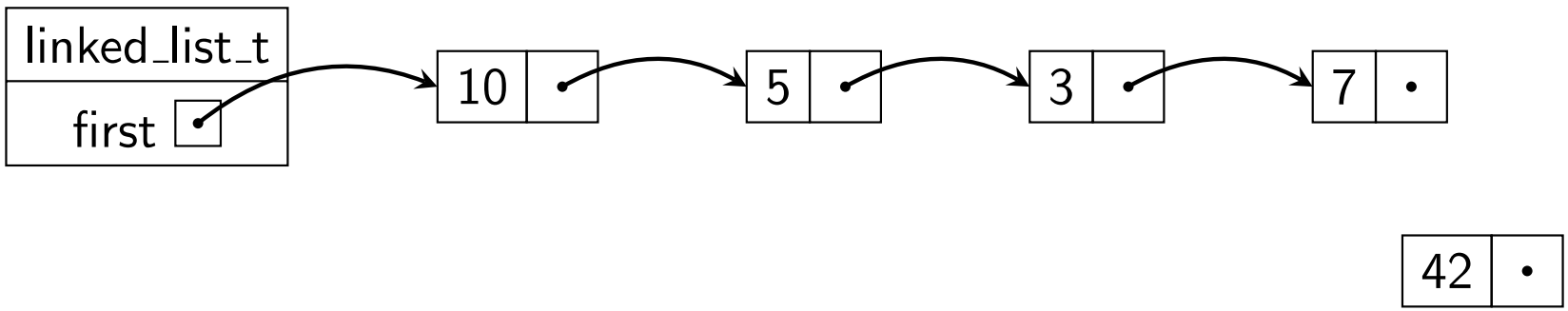
```
linked_list_t *new_list(void)
{
  // Alloziere und initalisiere Speicher auf dem Heap
    fuer die Listen-Struktur
  linked_list_t *ll = calloc(1,sizeof(linked_list_t));
  if (NULL != ll)
  {
    return ll;
  }
  else
  {
    printf("Couldn't allocate linked_list");
    abort(); // Programmabbruch
  }
}
```

Mit `calloc` (c wie clear) wird Speicherbereich mit Nullen initialisiert.

Einfügen am Ende der Liste

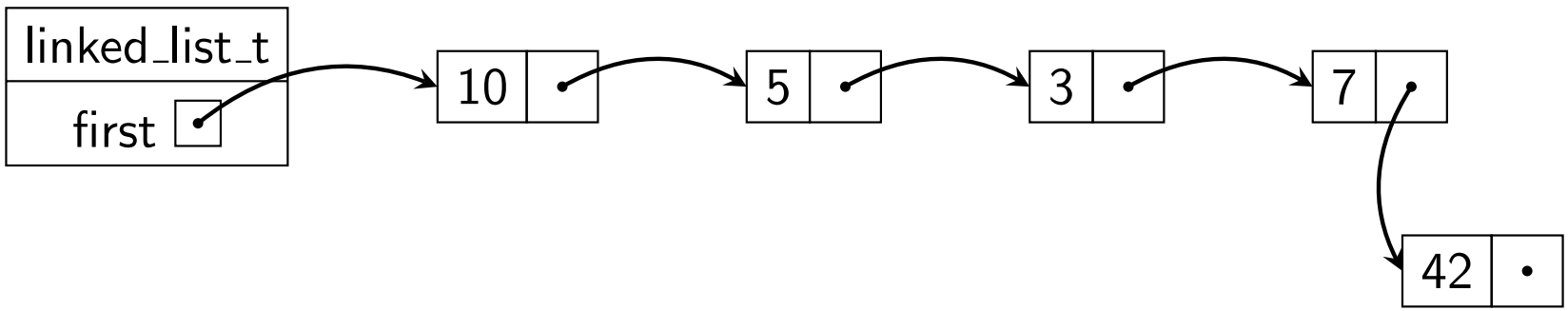


Einfügen am Ende der Liste



1. Speicher für neuen Knoten allozieren und Werte eintragen

Einfügen am Ende der Liste



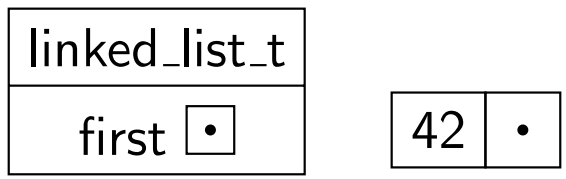
1. Speicher für neuen Knoten allozieren und Werte eintragen
2. Letzten Knoten finden und Zeiger auf neuen Knoten setzen

Einfügen am Ende der Liste

Spezialfall für leere Liste:

Einfügen am Ende der Liste

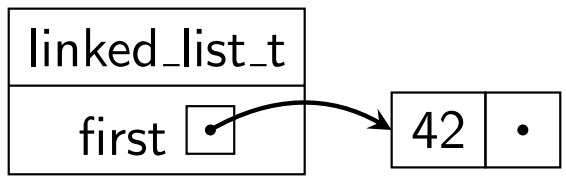
Spezialfall für leere Liste:



1. Speicher für neuen Knoten allozieren und Werte eintragen

Einfügen am Ende der Liste

Spezialfall für leere Liste:

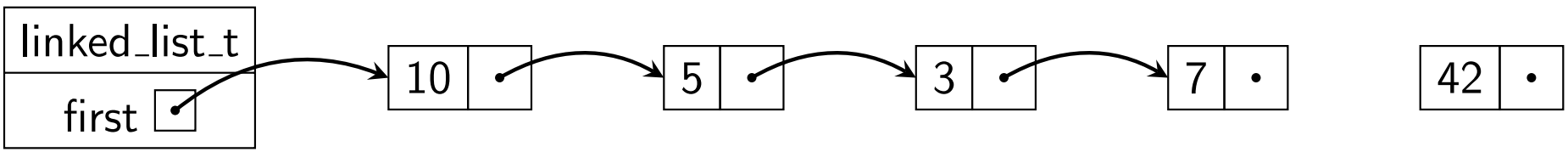


1. Speicher für neuen Knoten allozieren und Werte eintragen
2. First Zeiger auf neuen Knoten setzen

Einfügen am Ende der Liste

```
void add_elem(linked_list_t *ll, int i) {
  node_t *new_node = malloc(sizeof(node_t));
  if (NULL == new_node) {
    printf("Couldn't allocate new node\n");
    exit(-1);
  }
  new_node->value = i;
  new_node->next = NULL;
  if (NULL == (ll->first)) {
    ll->first = new_node;
  } else {
    node_t *n = ll->first;
    while (NULL != (n->next)) {
      n = n->next;
    }
    n->next = new_node;
  }
}
```

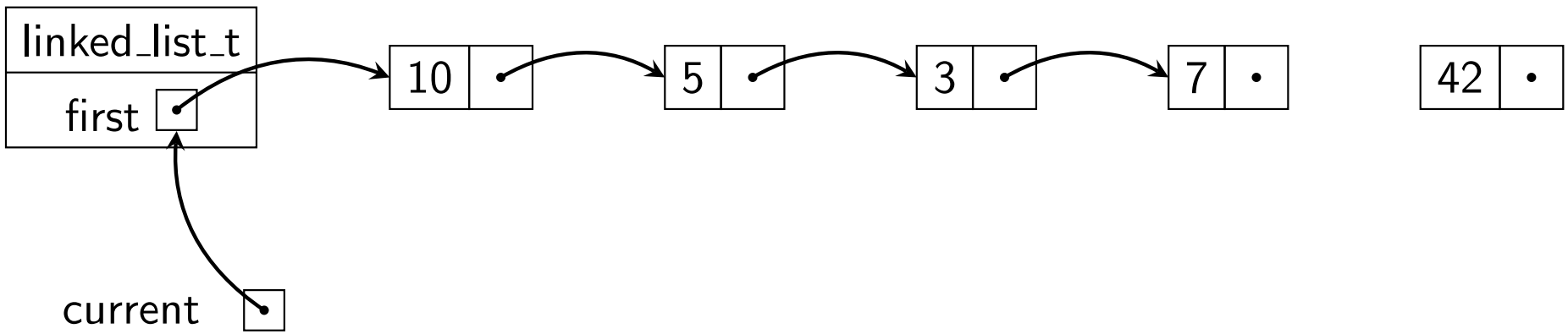
Einfügen am Ende der Liste - ohne Spezialfall



current

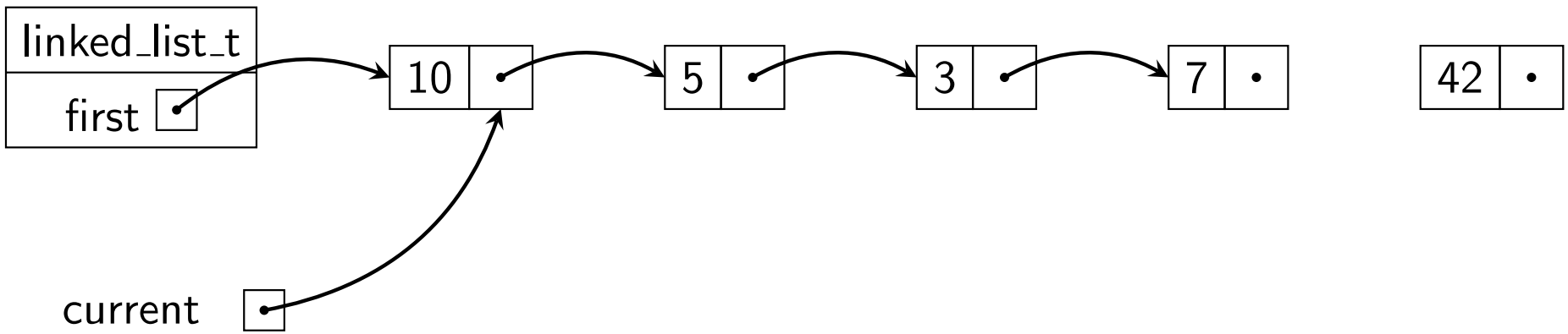
1. Variable `current` als Zeiger auf den Zeiger, der auf das aktuelle Element zeigt

Einfügen am Ende der Liste - ohne Spezialfall



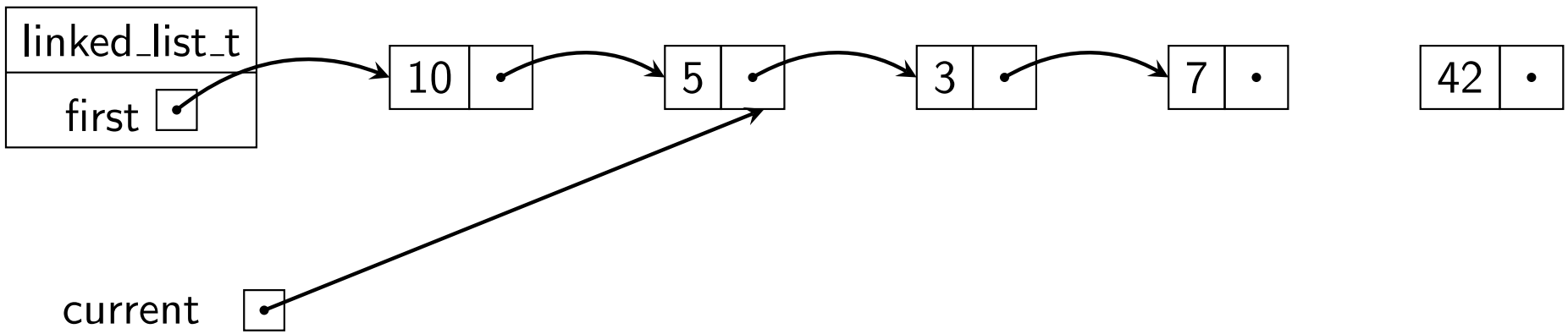
1. Variable `current` als Zeiger auf den Zeiger, der auf das aktuelle Element zeigt
2. Suche erste Position, wo current auf einen `NULL`-Zeiger zeigt

Einfügen am Ende der Liste - ohne Spezialfall



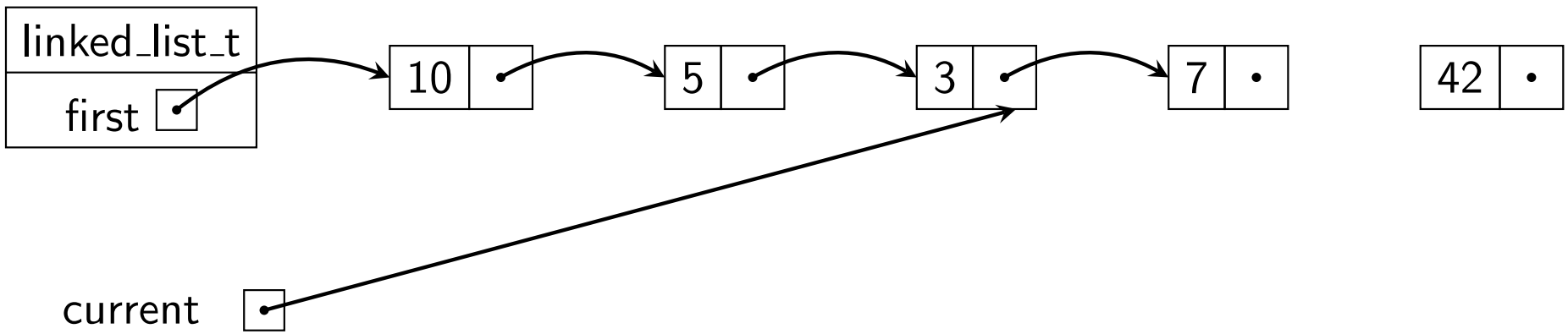
1. Variable `current` als Zeiger auf den Zeiger, der auf das aktuelle Element zeigt
2. Suche erste Position, wo current auf einen `NULL`-Zeiger zeigt

Einfügen am Ende der Liste - ohne Spezialfall



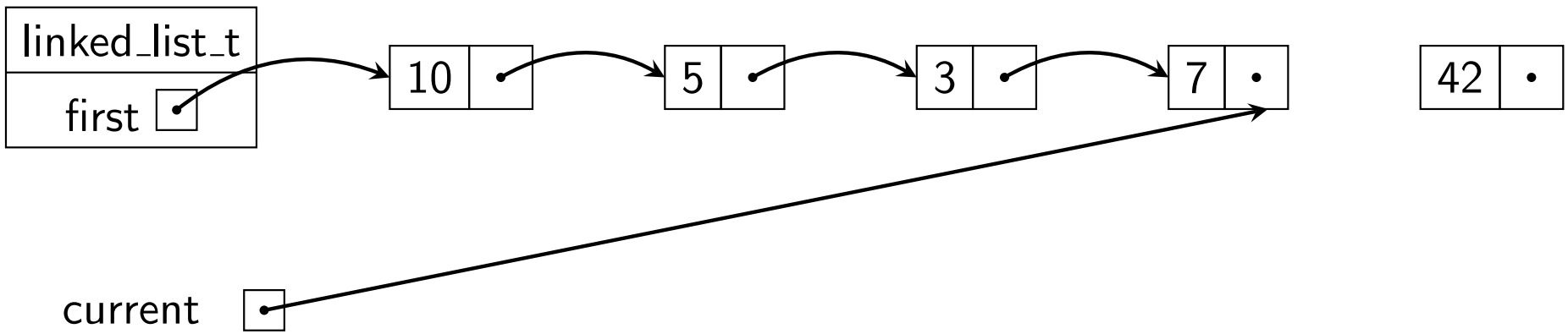
1. Variable `current` als Zeiger auf den Zeiger, der auf das aktuelle Element zeigt
2. Suche erste Position, wo current auf einen `NULL`-Zeiger zeigt

Einfügen am Ende der Liste - ohne Spezialfall



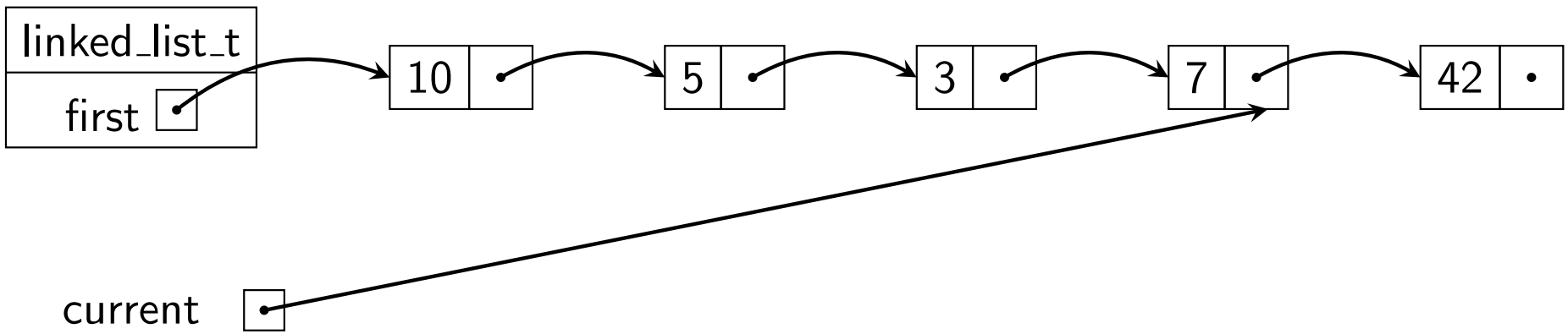
1. Variable `current` als Zeiger auf den Zeiger, der auf das aktuelle Element zeigt
2. Suche erste Position, wo current auf einen `NULL`-Zeiger zeigt

Einfügen am Ende der Liste - ohne Spezialfall



1. Variable `current` als Zeiger auf den Zeiger, der auf das aktuelle Element zeigt
2. Suche erste Position, wo current auf einen `NULL`-Zeiger zeigt

Einfügen am Ende der Liste - ohne Spezialfall



1. Variable `current` als Zeiger auf den Zeiger, der auf das aktuelle Element zeigt
2. Suche erste Position, wo current auf einen `NULL`-Zeiger zeigt
3. Ändere diesen Zeiger, so dass er auf den neuen Knoten zeigt

Einfügen am Ende der Liste - ohne Spezialfall

```
void add_elem(linked_list_t *ll, int i)
{
  node_t *new_node = malloc(sizeof(node_t));
  if (!new_node) {
    printf("Couldn't allocate new node");
    exit(-1);
  }
  new_node->value = i;
  new_node->next = NULL;
  node_t **current = &(ll->first);
  while (*current) {
    current = &((*current)->next);
  }
  *current = new_node;
}
```

Entfernen aus Liste

```
void remove_elem(linked_list_t *ll, int pos)
{
  node_t **prev = &(ll->first);
  while (pos > 0 && (*prev)->next) {
    pos--;
    prev = &(*prev)->next;
  }
  node_t *n = *prev;
  if (pos > 0) {
    *prev = n->next;
    free(n);
  }
}
```

Ausgeben der Liste

```
void print_list(linked_list_t *ll)
{
  node_t *n = ll->first;
  while (n)
  {
    // Ausgabe des Knotenwerts
    printf("%d  ", n->value);
    // Weiter mit dem Nachfolger
    n = n->next;
  }
  printf("\n");
}
```

Speicher Freigeben

```
void free_list(linked_list_t *ll)
{
  node_t *n = ll->first;
  while (n)
  {
    node_t *next = n->next;
    free(n);
    n = next;
  }
  free(ll);
}
```

Verwenden der Liste

```
// Erstelle leere Liste
linked_list_t *ll = new_list();
// Fuege Elemente in die Liste ein
for (int i = 1; i < 10; i++)
{
  add_elem(ll, i);
}
// Gebe die Liste aus
print_list(ll);
// Entferne Elemente aus der Liste
for (int i = 9; i >= 0; i -= 2)
{
  remove_elem(ll, i);
}
print_list(ll);
// Entferne die Liste vom Heap
free_list(ll);
```

Wenn Dinge so richtig schief gehen

Der Heartbleed Bug

- Bug in der OpenSSL-Bibliothek (verschlüsselte Verbindung)
- Am 31.Dezember 2011 in der Codebase eingegeben und am 14. März 2012 veröffentlich
- Erst im April 2014 entdeckt und behoben

Was war geschehen? I

- Server schicken Nachricht, um Verbindung offen zu halten (Herzschlag = heart beat)
- Protokoll in OpenSSL:
  - Sender schickt einen beliebigen Text an den Empfänger
  - Empfänger antwortet mit gleichem Text

```
typedef struct message {
  int length;
  char *data;
} message_t;
```

Was war geschehen? II

- Server liest Länge n aus und kopiert n Bytes in ausgehende Nachricht

```
message_t *incoming = ...;
message_t *outgoing = ...;

memcpy(outgoing->data, incoming->data, incoming->length);
```

Was war geschehen? III

- Was passiert, wenn ein Angreifer folgende Nachricht schickt?

```
message_t *bad_message = ...;
bad_message->length = 65535;
bad_message->data = "Hallo";
```

Zusammenfassung: C

- Kein automatisches Speichermanagement
- Keine Unterscheidung von Arrays und Zeigern
- Schwache Typisierung
- Keine Fehlerbehandlung mittels Exception