Verkettete Listen

>> Bisher hatten wir nur eine Art von Datenstruktur zur Darstellung von Sammlungen gleichartiger Werte:

Arrays sind gut für den Elementzugriff, aber das Einfügen von Elementen ist aufwendig (mit Ausnahme vom Einfügen am Ende).

structs geben uns „Container“ für Variablen verschiedener Datentypen.

>> Arrays leiden auch unter mangelnder Flexibilität: was passiert, wenn wir ein größeres Array benötigen?

Durch geschickte Verwendung von Pointern, dynamischer Speicherverwaltung und structs können wir eine neue Datenstruktur bauen, die mit unseren Bedürfnissen wachsen und schrumpfen kann.

Verkettete Listen

Diese Kombination von Elementen nennen wir eine **verkettete Liste**.

Ein Knoten einer verketteten Liste ist eine spezielle *struct* mit zwei Komponenten:

1. Daten eines bestimmten Datentyps (int, char, float...)
2. Ein Pointer auf eine struct, die einen weiteren Knoten aufnimmt.

So entsteht eine Kette von Elementen, der wir von Anfang bis Ende folgen können.

Eine (einfach) verkettete Liste
singly-linked list

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

Operationen auf verketteten Listen

Um mit verketteten Listen zu arbeiten, verwenden wir folgende Operationen:

1. **create** – Eine verkettete Liste erstellen, wenn noch keine existiert.
2. **find** – Eine verkettete Liste durchsuchen, um ein Element zu finden.
3. **insert** – Einen neuen Knoten in die verkettete Liste einfügen.
4. **delete** – Ein einzelnes Element aus der verketteten Liste löschen.
5. **destroy** – Eine verkettete Liste komplett löschen.

Liste erstellen

```c
sllnode* create(VALUE val);
```

a. Speicher für neuen *sllnode* allozieren
b. Prüfen, ob Speicher verfügbar war
c. Den *val*-Teil des Knotens initialisieren
d. Den *next*-Teil des Knotens initialisieren
e. Pointer auf den neuen *sllnode* zurückgeben

```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

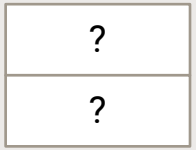
Liste erstellen

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

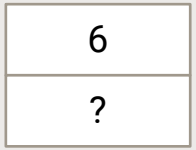
```
sllnode* create(VALUE val);
```

a. Speicher für neuen *sllnode* allozieren
b. Prüfen, ob Speicher verfügbar war
c. Den *val*-Teil des Knotens initialisieren
d. Den *next*-Teil des Knotens initialisieren
e. Pointer auf den neuen *sllnode* zurückgeben

```
sllnode* new = create(6);
```

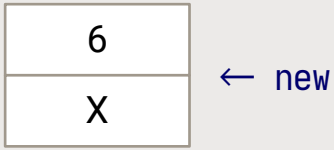


(a) nach malloc



(c) val-Initialisierung

(d) next-Initialisierung



(e) Pointer-Rückgabe

Liste durchsuchen

```
bool find(sllnode* head, VALUE val);
```

a. Einen Traversierungspointer auf den Listenkopf setzen
b. Prüfen ob der *val*-Teil des aktuellen Knotens der gesuchte ist
c. Falls nicht, zum nächsten Knoten gehen und bei (b) fortfahren
d. Am Ende der Liste: Fehlschlag melden

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

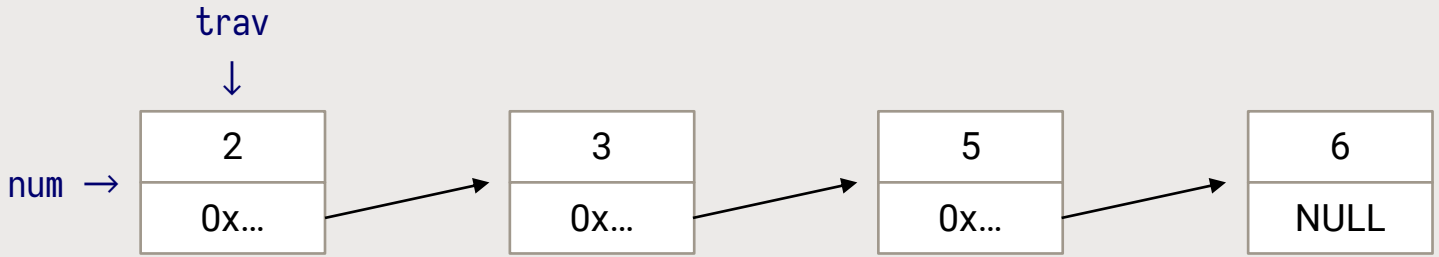
```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

Liste durchsuchen

```c
bool find(sllnode* head, VALUE val);
```

a. Einen Traversierungspointer auf den Listenkopf setzen
b. Prüfen ob der *val*-Teil des aktuellen Knotens der gesuchte ist
c. Falls nicht, zum nächsten Knoten gehen und bei (b) fortfahren
d. Am Ende der Liste: Fehlschlag melden

```c
bool contains_five = find(num, 5);
```



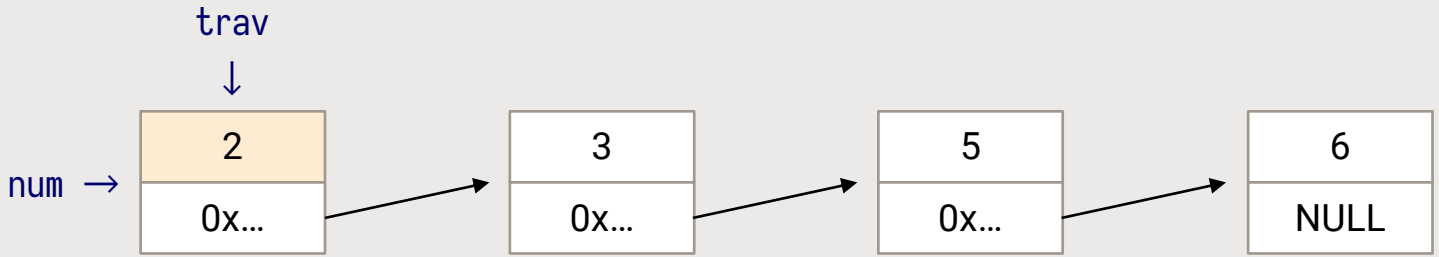
Liste durchsuchen

```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
bool find(sllnode* head, VALUE val);
```

a. Einen Traversierungspointer auf den Listenkopf setzen
b. Prüfen ob der *val*-Teil des aktuellen Knotens der gesuchte ist
c. Falls nicht, zum nächsten Knoten gehen und bei (b) fortfahren
d. Am Ende der Liste: Fehlschlag melden

```c
bool contains_five = find(num, 5);
```



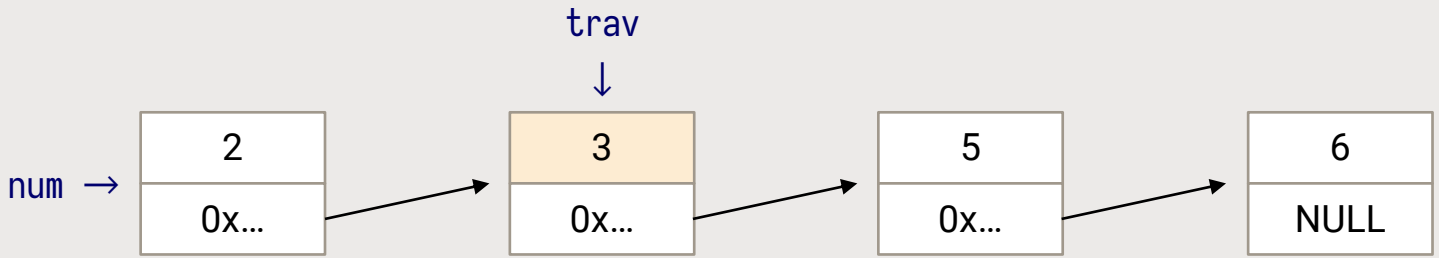
Liste durchsuchen

```
bool find(sllnode* head, VALUE val);
```

a. Einen Traversierungspointer auf den Listenkopf setzen
b. Prüfen ob der *val*-Teil des aktuellen Knotens der gesuchte ist
c. Falls nicht, zum nächsten Knoten gehen und bei (b) fortfahren
d. Am Ende der Liste: Fehlschlag melden

```
bool contains_five = find(num, 5);
```

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```



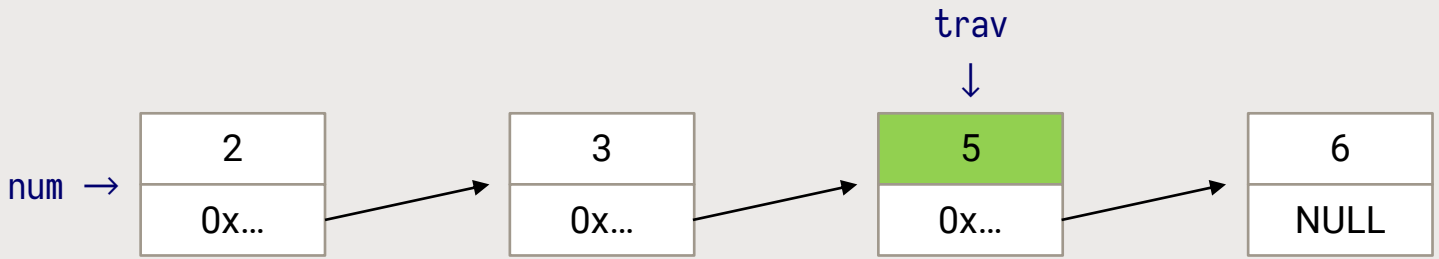
Liste durchsuchen

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```
bool find(sllnode* head, VALUE val);
```

a. Einen Traversierungspointer auf den Listenkopf setzen
b. Prüfen ob der *val*-Teil des aktuellen Knotens der gesuchte ist
c. Falls nicht, zum nächsten Knoten gehen und bei (b) fortfahren
d. Am Ende der Liste: Fehlschlag melden

```
bool contains_five = find(num, 5);
```



Element in Liste einfügen

```c
sllnode* insert(sllnode* head, VALUE val);
```

a. Dynamisch Speicher für einen neuen *sllnode* allozieren
b. Prüfen, ob genügend Speicher verfügbar war
c. Knoten am Anfang der verketteten Liste einfügen
d. Pointer auf den neuen Listenkopf zurückgeben

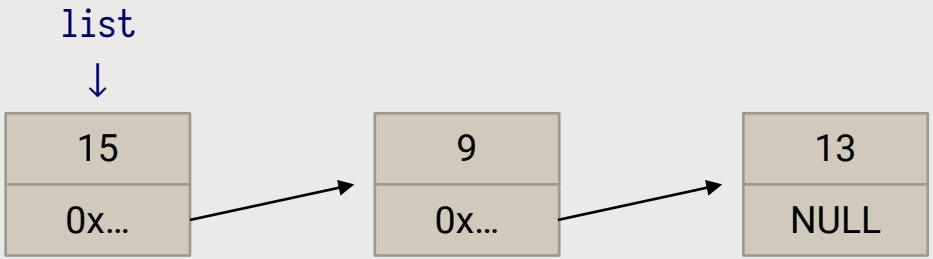
```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

Element in Liste einfügen

```
sllnode* insert(sllnode* head, VALUE val);
```

a. Dynamisch Speicher für einen neuen *sllnode* allozieren
b. Prüfen, ob genügend Speicher verfügbar war
c. Knoten am Anfang der verketteten Liste einfügen
d. Pointer auf den neuen Listenkopf zurückgeben

```
list = insert(list, 12);
```




```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

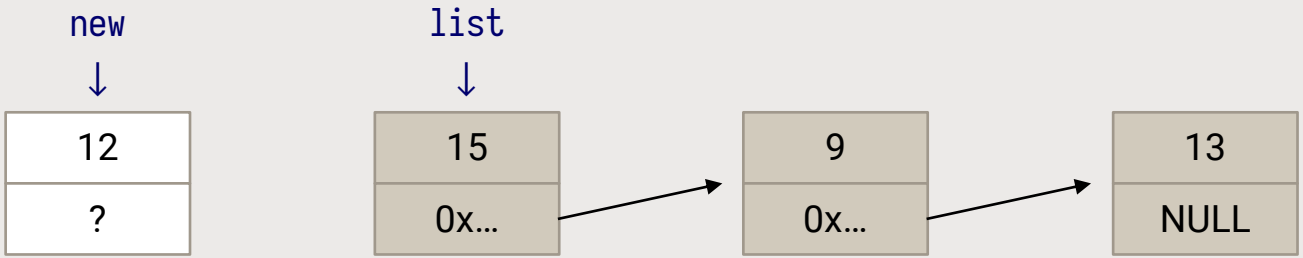
Element in Liste einfügen

```
sllnode* insert(sllnode* head, VALUE val);
```

Knoten am Anfang der verketteten Liste einfügen:

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```
list = insert(list, 12);
```



Element in Liste einfügen

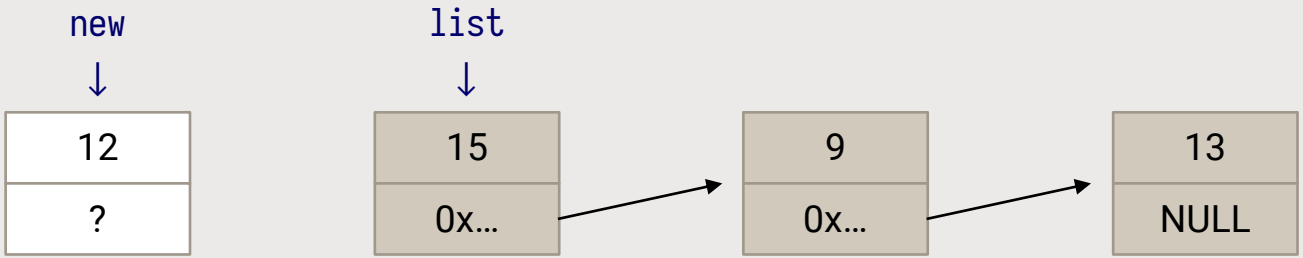
```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
sllnode* insert(sllnode* head, VALUE val);
```

Knoten am Anfang der verketteten Liste einfügen:

Zuerst *next*-Pointer des neuen Knotens setzen, dann *list*-Pointer aktualisieren. Falsche Reihenfolge führt zu Datenverlust.

```c
list = insert(list, 12);
```



Element in Liste einfügen

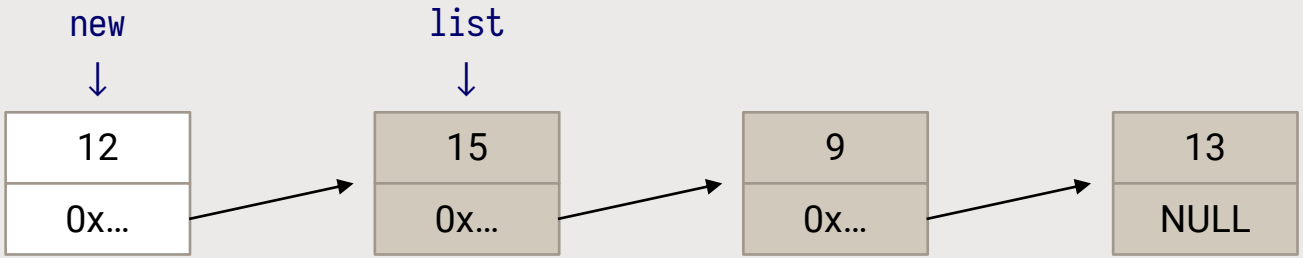
```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
sllnode* insert(sllnode* head, VALUE val);
```

Knoten am Anfang der verketteten Liste einfügen:

Zuerst *next*-Pointer des neuen Knotens setzen, dann *list*-Pointer aktualisieren. Falsche Reihenfolge führt zu Datenverlust.

```c
list = insert(list, 12);
```



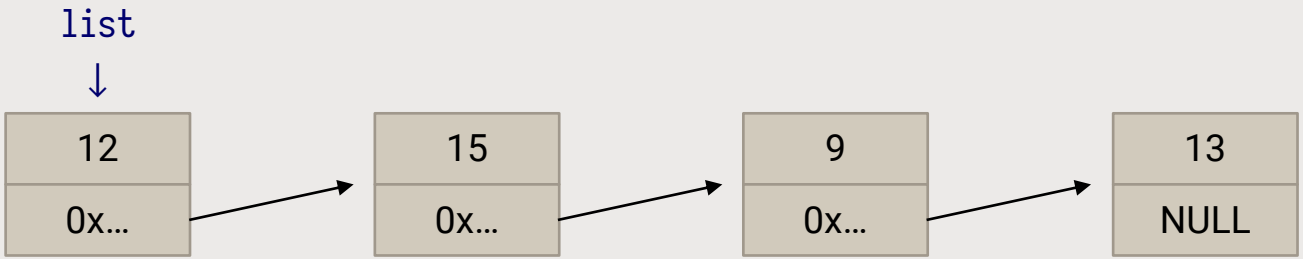
Element in Liste einfügen

```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
sllnode* insert(sllnode* head, VALUE val);
```

Knoten am Anfang der verketteten Liste einfügen:
Zuerst *next*-Pointer des neuen Knotens setzen, dann *list*-Pointer aktualisieren. Falsche Reihenfolge führt zu Datenverlust.

```c
list = insert(list, 12);
```

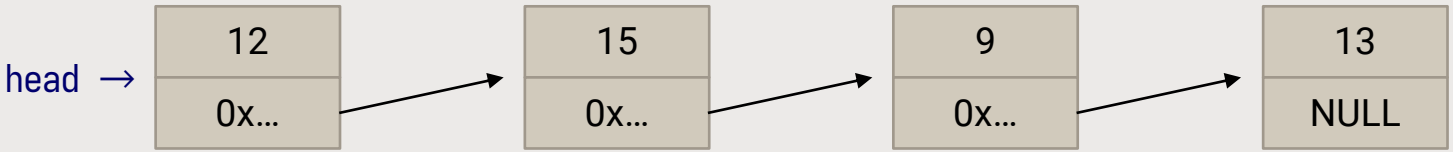


Liste komplett löschen

```
void destroy(sllnode* head);
```

a. Wenn NULL-Pointer erreicht, stoppen
b. Rest der Liste löschen
c. Aktuellen Knoten freigeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

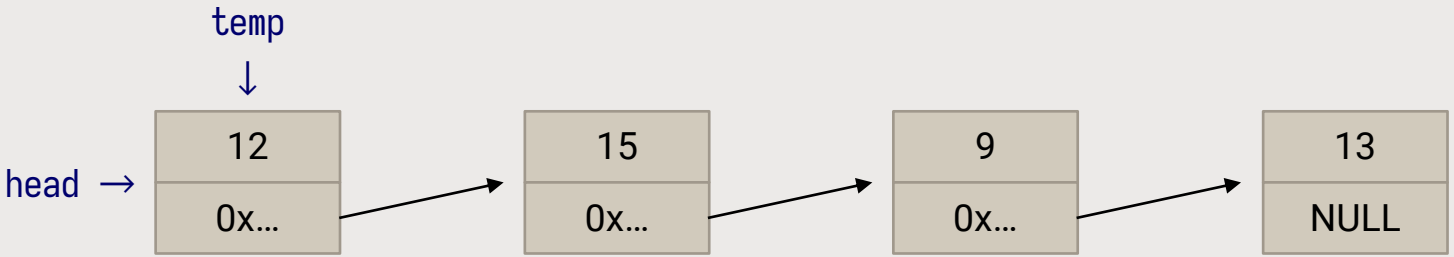


Liste komplett löschen

```
void destroy(sllnode* head);
```

a. Wenn NULL-Pointer erreicht, stoppen
b. Rest der Liste löschen
c. Aktuellen Knoten freigeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

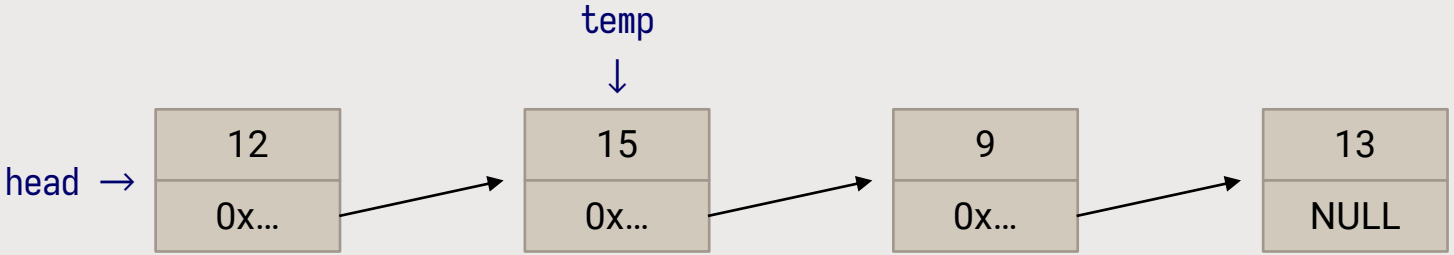


Liste komplett löschen

```
void destroy(sllnode* head);
```

a. Wenn NULL-Pointer erreicht, stoppen
b. Rest der Liste löschen
c. Aktuellen Knoten freigeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

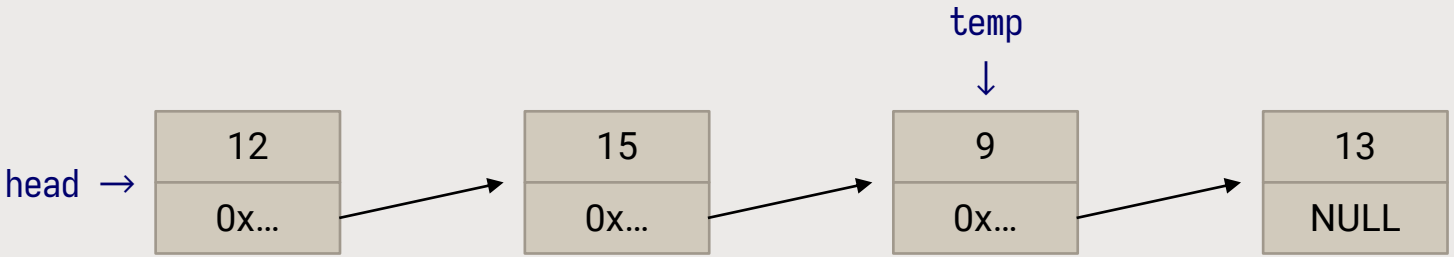


Liste komplett löschen

```
void destroy(sllnode* head);
```

a. Wenn NULL-Pointer erreicht, stoppen
b. Rest der Liste löschen
c. Aktuellen Knoten freigeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

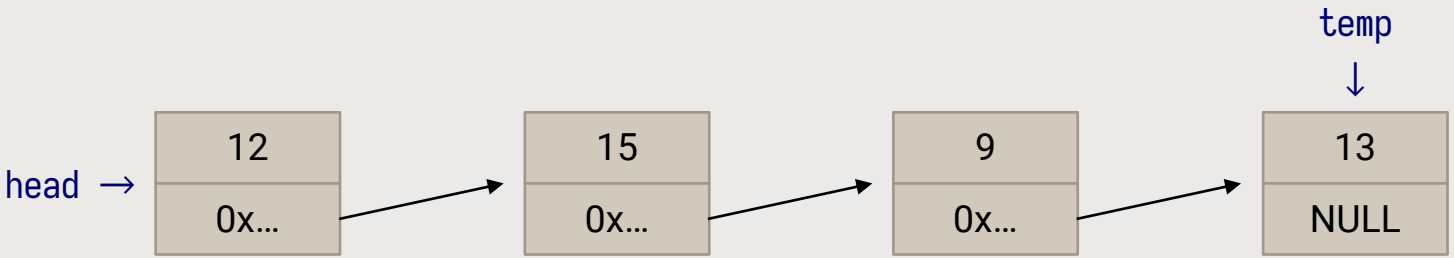


Liste komplett löschen

```
void destroy(sllnode* head);
```

a. Wenn NULL-Pointer erreicht, stoppen
b. Rest der Liste löschen
c. Aktuellen Knoten freigeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

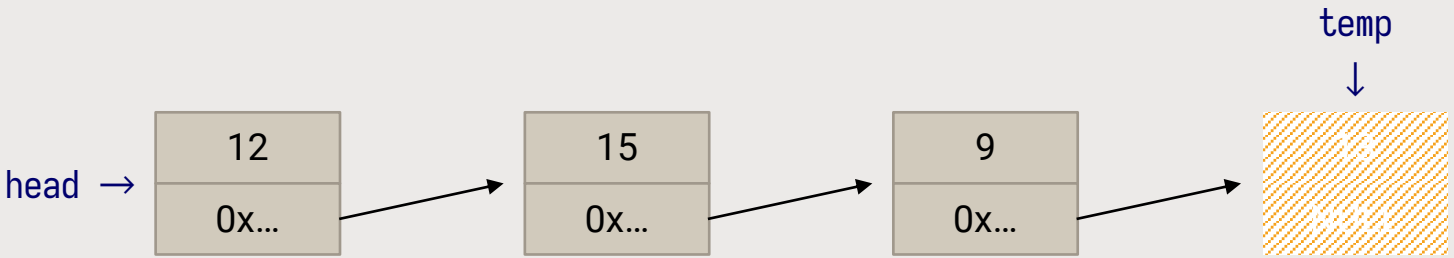


Liste komplett löschen

```
void destroy(sllnode* head);
```

a. Wenn NULL-Pointer erreicht, stoppen
b. Rest der Liste löschen
c. Aktuellen Knoten freigeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

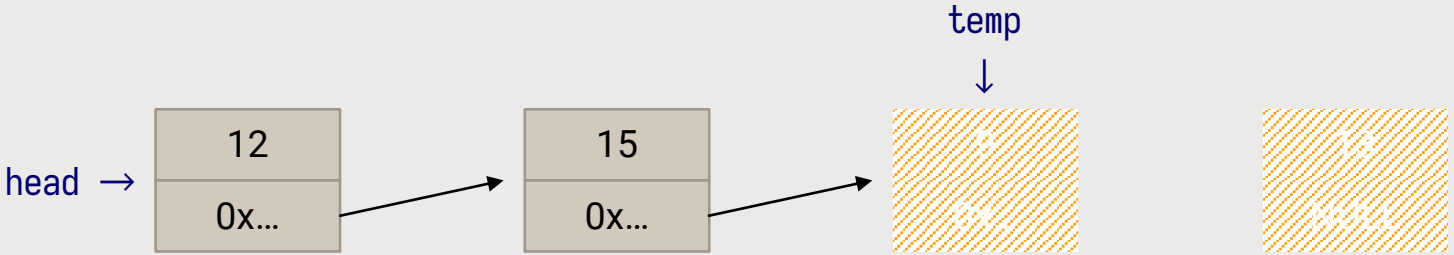


Liste komplett löschen

```
void destroy(sllnode* head);
```

a. Wenn NULL-Pointer erreicht, stoppen
b. Rest der Liste löschen
c. Aktuellen Knoten freigeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

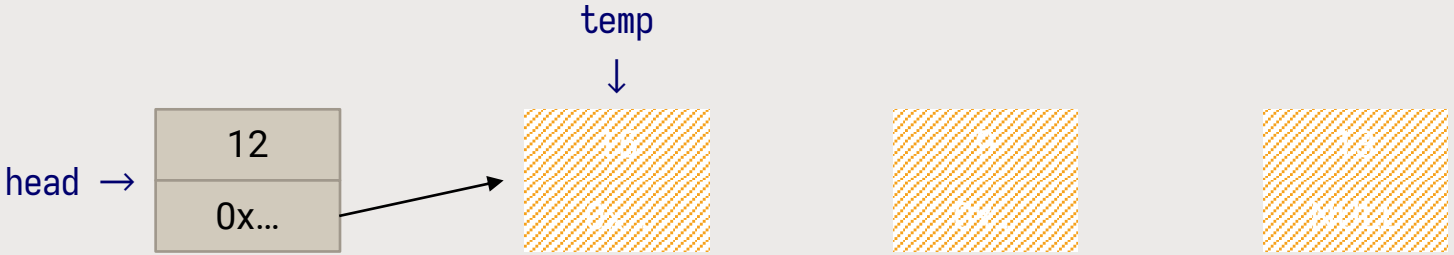


Liste komplett löschen

```
void destroy(sllnode* head);
```

a. Wenn NULL-Pointer erreicht, stoppen
b. Rest der Liste löschen
c. Aktuellen Knoten freigeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```



Liste komplett löschen

```
void destroy(sllnode* head);
```

a. Wenn NULL-Pointer erreicht, stoppen
b. Rest der Liste löschen
c. Aktuellen Knoten freigeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

>> Im Folgenden sehen wir uns an, wie man diese Operationen mit den uns bekannten Werkzeugen **programmiert**.

Dank *structs*, dynamischer Speicherallokation und Pointern brauchen wir dafür nur wenige Zeilen C-Code.

Versuchen Sie es selbst! Das ist eine gute Übung, um mehr Routine zu bekommen.

Also: Gegeben ist die bereits bekannte *struct*-Definition auf der rechten Seite.

Implementieren Sie damit *create*, *find*, *insert* und *destroy*.

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```
sllnode* create(VALUE val);
```

a. Speicher für neuen *sllnode* allozieren
b. Prüfen, ob Speicher verfügbar war
c. Den *val*-Teil des Knotens initialisieren
d. Den *next*-Teil des Knotens initialisieren
e. Pointer auf den neuen *sllnode* zurückgeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
sllnode* create(VALUE val);
```

a. Speicher für neuen *sllnode* allozieren
b. Prüfen, ob Speicher verfügbar war
c. Den *val*-Teil des Knotens initialisieren
d. Den *next*-Teil des Knotens initialisieren
e. Pointer auf den neuen *sllnode* zurückgeben

```c
sllnode* create(VALUE val)
{
    sllnode* new = malloc(sizeof(sllnode));
    if (new == NULL)
    {
        return NULL;
    }

    new->val = val;
    new->next = NULL;
    return new;
}
```

```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
bool find(sllnode* head, VALUE val);
```

a. Einen Traversierungspointer auf den Listenkopf setzen
b. Liste durchlaufen und Werte vergleichen
c. Bei Fund: *true* zurückgeben
d. Ende der Liste erreicht: *false* zurückgeben

```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
bool find(sllnode* head, VALUE val);
```

a. Einen Traversierungspointer auf den Listenkopf setzen
b. Liste durchlaufen und Werte vergleichen
c. Bei Fund: *true* zurückgeben
d. Ende der Liste erreicht: *false* zurückgeben

```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
bool find(sllnode* head, VALUE val)
{
    for (sllnode* trav = head; trav != NULL; trav = trav->next)
    {
        if (trav->val == val)
        {
            return true;
        }
    }
    return false;
}
```

```
sllnode* insert(sllnode* head, VALUE val);
```

a. Neuen Knoten erstellen
b. Neuen Knoten auf den alten Kopf zeigen lassen
c. Neuen Knoten als neuen Kopf zurückgeben

```
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
sllnode* insert(sllnode* head, VALUE val);
```

a. Neuen Knoten erstellen
b. Neuen Knoten auf den alten Kopf zeigen lassen
c. Neuen Knoten als neuen Kopf zurückgeben

```c
sllnode* insert(sllnode* head, VALUE val)
{
    sllnode* new = create(val);

    // Fehler beim Anlegen? => alten Kopf zurückgeben
    if (new == NULL)
    {
        return head;
    }

    // Alter Kopf kommt nach neuem Kopf
    new->next = head;

    return new;
}
```

```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
void destroy(sllnode* head);
```

a. Basisfall: NULL-Pointer check
b. Rekursiver Aufruf für Rest der Liste
c. Aktuellen Knoten freigeben

```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

```c
void destroy(sllnode* head);
```

a. Basisfall: NULL-Pointer check
b. Rekursiver Aufruf für Rest der Liste
c. Aktuellen Knoten freigeben

```c
void destroy(sllnode* head)
{
    // Basisfall
    if (head == NULL)
    {
        return;
    }

    // Rekursiver Aufruf
    destroy(head->next);

    // Aktuellen Knoten freigeben
    free(head);
}
```

```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

EXTRAS IN 3 MINUTEN

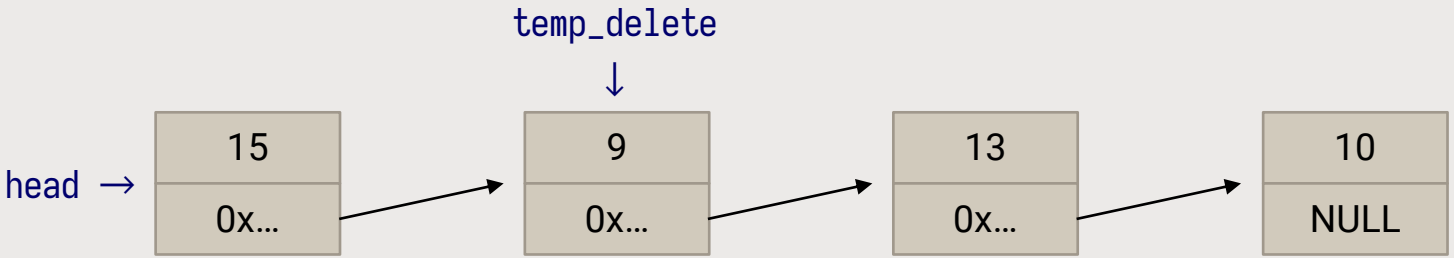
FRAGEN – ANTWORTEN – RÄTSEL
UND KURZE ZUSAMMENFASSUNG

Die „vergessene“ Operation: delete

```c
bool delete(sllnode* head, VALUE val);
```

```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

Problem beim Löschen eines Elements:
Wir müssen die Verbindung zum nächsten Element erhalten.
Aber: Dazu brauchen wir Zugriff auf den Vorgänger.
Mit nur einem Pointer beim Traversieren kommen wir nicht mehr zurück.



```c
bool delete(sllnode** head, VALUE val)
{
    if (*head == NULL) // Check: Leere Liste?
        return false;

    // Pointer für aktuelle und vorherige Position
    sllnode* current = *head;
    sllnode* prev = NULL;

    // Element suchen
    while (current != NULL && current->val != val)
    {
        prev = current;
        current = current->next;
    }

    if (current == NULL) // Element nicht gefunden
        return false;

    // Element gefunden - Verbindungen anpassen
    if (prev == NULL)
        *head = current->next; // Erstes Element
    else
        prev->next = current->next;  // Späteres Element

    // Speicher freigeben
    free(current);
    return true;
}
```

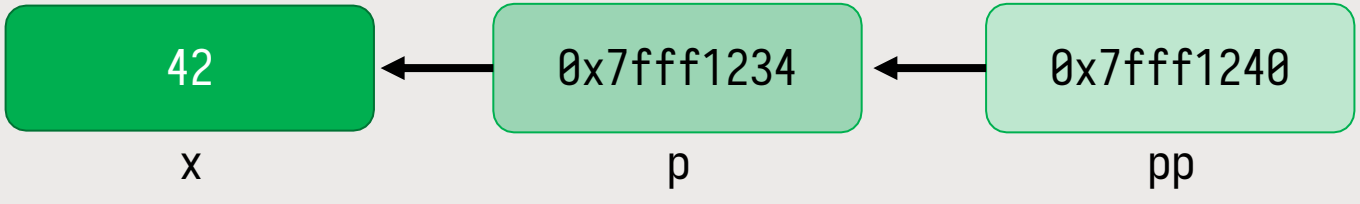
```c
typedef struct sllist
{
    VALUE val;
    struct sllist* next;
}
sllnode;
```

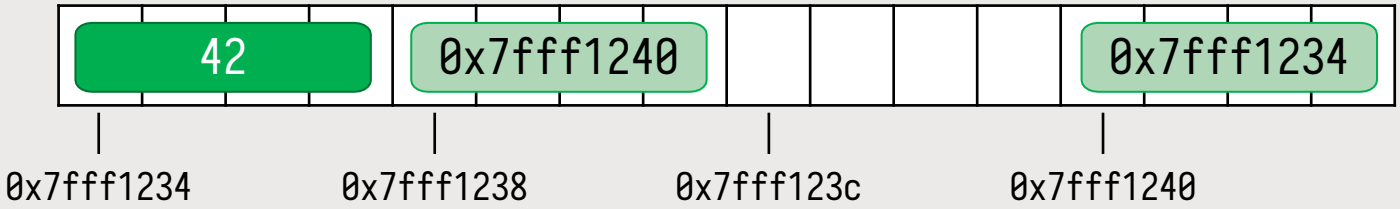
Pointer auf Pointer

```
int x = 42;
int* p = &x;   // p zeigt auf x
int** pp = &p; // pp zeigt auf p
```

```
bool delete(sllnode** head, VALUE val);
```

In der delete-Funktion: *head ist wie pp ein Pointer auf einen Pointer, damit wir den Original-Pointer ändern können.*





Ohne Pointer auf Pointer gibt es ein Problem.

```
// Mit einfachem Pointer - FUNKTIONIERT NICHT:

void delete_wrong(sllnode* head) {
    head = head->next;  // Ändert nur lokale Kopie
}



// Mit Pointer auf Pointer - FUNKTIONIERT:

void delete_correct(sllnode** head) {
    *head = (*head)->next;  // Ändert Original
}
```

Folge: unterschiedliche Aufrufe

```
// create, find und insert
// nutzen einen Pointer:

list = insert(list, 42);
found = find(list, 42);
destroy(list);



// delete benötigt die Adresse
// des list-Pointers:

delete(&list, 42);
```

```
bool delete(sllnode** head, VALUE val)
{
    if (*head == NULL) // Check: Leere Liste?
        return false;

    // Pointer für aktuelle und vorherige Position
    sllnode* current = *head;
    sllnode* prev = NULL;

    // Element suchen
    while (current != NULL && current->val != val)
    {
        prev = current;
        current = current->next;
    }

    if (current == NULL) // Element nicht gefunden
        return false;

    // Element gefunden - Verbindungen anpassen
    if (prev == NULL)
        *head = current->next; // Erstes Element
    else
        prev->next = current->next;  // Späteres Element

    // Speicher freigeben
    free(current);
    return true;
}
```

Beispiel

```
sllnode* list = create(5); // list zeigt auf {5}
list = insert(list, 3);    // list zeigt auf {3,5}
list = insert(list, 1);    // list zeigt auf {1,3,5}


// Falscher Aufruf:
delete(list, 3);           // Kompilierfehler!


// Richtiger Aufruf:
delete(&list, 3);          // list zeigt auf {1,5}
```

Alternative zu Arrays:
dynamisch wachsend

delete benötigt
Pointer auf Pointer

Knoten: Wert + Pointer
auf nächstes Element

Wichtig: Pointer-Updates in
richtiger Reihenfolge

destroy arbeitet
rekursiv von hinten

Nach malloc immer
auf NULL prüfen!