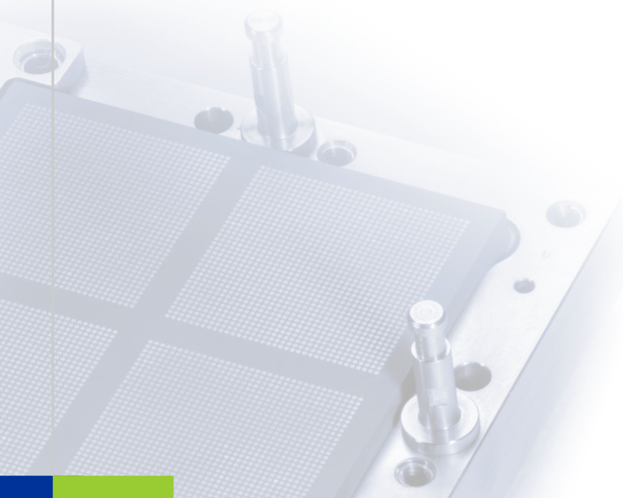
R|R|Z|N

Programmierung mit C

Strukturen



Regionales Rechenzentrum für Niedersachsen

Universität Hannover

Strukturen

- Variablen mit unterschiedlichen Datentypen werden zusammengefasst.
- Variablen zu einem Thema werden zusammengefasst.
- Beispiele für Strukturen:
  - Personendaten (Name, Vorname, Straße, Ort)
  - Koordinaten eines Punktes (x-, y-, z-Koordinate)
  - Beschreibung von Gegenständen etc. (Bezeichnung, Preis, Artikelnummer)

Strukturen in C

R|R|Z|N|

- `struct`
  - Jedes Element der Struktur besitzt einen eigenen Speicherplatz.
  - Jedes Element kann jederzeit angesprochen werden.
  - ... ist ähnlich einem Array.
- `union`
  - Alle Elemente der Union liegen an der gleichen Speicheradresse.
  - Die Größe des Speicherbereichs richtet sich nach dem größten Element.
- `enum`
  - Aufzählung von Konstanten.
- `typedef`
  - Strukturen etc. als einen neuen Datentyp definieren.

Aufbau einer Struktur in C

Deklaration der Struktur

```
struct STRUKTUR_NAME {
        Datentyp variable1;
        Datentyp variable2;
        ...
        Datentyp variableN;
};
```

Komponenten der Struktur

Struktur und Strukturelemente

- Eine Struktur
  - ... liefert ein Template (Schablone) zu einem bestimmten Thema.
  - ... erstellt einen Prototyp zum Zusammenfassen von verschiedenen Elementen.
  - ... fasst Variablen, die unterschiedliche Datentypen besitzen können, zusammen.
  - ... ist als globale oder lokale Variable deklariert.
- Ein Strukturelement
  - ... definiert die Merkmale eines Objektes.
  - ... wird als Komponente oder Member bezeichnet.
  - ... wird innerhalb der Struktur deklariert.
  - ... wird wie eine lokale Variable behandelt.
  - ... ist nur innerhalb der Struktur sichtbar.

Struktur und Strukturelemente in C

R|R|Z|N

- `struct STRUKTUR_NAME { };`
  - Jede Struktur wird mit dem Schlüsselwort `struct` eingeleitet.
  - Anschließend kommt der Name der Struktur. Strukturnamen werden in Großbuchstaben geschrieben. Der Unterstrich kann als Trennzeichen zwischen zwei Wörtern genutzt werden.
  - Mit Hilfe der geschweiften Klammern werden die dazugehörigen Komponenten zusammengefasst.
  - Die Struktur endet mit einem Semikolon. Jede Struktur wird als Anweisung vom Compiler interpretiert.
- `Datentyp variable1;`
  - Innerhalb der geschweiften Klammern einer Struktur werden die dazugehörigen Komponenten aufgelistet.
  - Jede Deklaration einer Komponente endet mit einem Semikolon.
  - Jede Deklaration einer Komponente entspricht der Deklaration einer lokalen Variablen.
  - Als Datentyp kann `int`, `float`, `char`, etc. genutzt werden.
  - Komponenten können als Array oder als Typ einer anderen Struktur definiert werden.

Definition einer Strukturvariablen

R|R|Z|N|

Deklaration der Struktur

```
struct STRUKTUR_NAME {
        Datentyp variable1;
        Datentyp variable2;
        ...
        Datentyp variableN;
}varStruktur01;

struct STRUKTUR_NAME varStruktur02;
```

Deklaration einer Strukturvariablen

Struktur und Strukturvariablen

- Eine Struktur
  - ... liefert ein Template (Schablone) zu einem bestimmten Thema.
  - ... erstellt einen Prototyp zum Zusammenfassen von verschiedenen Elementen.
  - ... fasst Variablen, die unterschiedliche Datentypen besitzen können, zusammen.
  - ... kann keiner anderen Struktur zugewiesen werden.
- Die Strukturvariable
  - ... stellt eine Instanz dar, die ein Gegenstand (Struktur) beschreibt.
  - ... erzeugt die Struktur. Es wird Speicherplatz für die Struktur bereitgestellt.
  - Der Inhalt einer Strukturvariablen kann einer anderen Strukturvariablen übergeben werden.
  - ... kann lokal oder global deklariert werden.

Strukturvariablen deklarieren

R | R | Z | N

```
struct ARTIKEL {
   int    artikelnummer;
   char   artikelBezeichnung[20];
   double preis;
   float  istBestand;
   float  minBestand;
};

struct PERSONEN{
   int    kundennummer;
   char   kundeBezeichnung[20];
   char   strasse[20];
   char   wohnort[20];
}lieferant, kunde;

int main() {
   struct ARTIKEL art;
   return 0;
}
```

Die Strukturvariablen können gleichzeitig mit der Struktur deklariert werden.

Direkt am Anschluss der schließenden geschweiften Klammern werden die Strukturvariablen, durch Kommata getrennt, aufgeführt.

In diesem Beispiel werden die Variablen `lieferant` und `kunde` von der Struktur `ARTIKEL` deklariert.

Hier wird eine lokale Strukturvariable erzeugt.

Zugriff auf Komponenten

R | R | Z | N

```
struct ARTIKEL {
   int    artikelnummer;
   char   artikelBezeichnung[20];
   double preis;
   float  istBestand;
   float  minBestand;
};


int main() {
   struct ARTIKEL ware;

   ware.artikelnummer = 4789;
   ware.preis = 3.45

   return 0;
}
```

Mit Hilfe des Punktoperators wird auf die einzelnen Strukturelemente zugegriffen.

Der Punktoperator verbindet eine Strukturvariable mit einer Komponenten.

Die Struktur der Variablen muss die Komponenten enthalten. Andernfalls zeigt der Compiler einen Fehler an.

Komponenten initialisieren


```
struct ARTIKEL {
   int    artikelnummer;
   char   artikelBezeichnung[20];
   double preis;
   float  istBestand;
   float  minBestand;
};


int main() {
   struct ARTIKEL ware;

   struct Artikel art = {
        1, "Apfel pro kg", 1.99, 5, 3
   };

   return 0;
}
```

Die Strukturvariable wird deklariert und gleichzeitig werden jeder Komponenten ein Anfangswert zugewiesen.

Die Komponenten werden mit einem bestimmten Wert initialisiert.

Die Anfangswerte werden durch Kommata getrennt.

Der erste Anfangswert wird der ersten Komponenten in der Liste zugewiesen und so weiter.

Strukturelemente und Arrays

R|R|Z|N|

```
struct Artikel {
   int artikelnummer;
   char artikelBezeichnung[20];
   double preis;
   float istBestand;
   float minBestand;
};

int main() {
   struct Artikel art;

   art.artikelBezeichnung[0] = 'A';

   strcpy(art.artikelBezeichnung, "Apfel");

   return 0;
}
```

Jede Komponente kann genauso wie eine lokale Variable als ein- oder mehrdimensionales Array deklariert werden.

Dem Komponenten-Namen folgt in eckigen Klammern die Anzahl der Elemente.

Mit Hilfe des Indizes kann ein Feld angesprochen werden.

Für Strings können alle vordefinierten Funktionen genutzt werden.

Zugriff auf die Array-Elemente

R | R | Z | N |

- `strukturvariable.strukturelement[Index] = ausdruck;`
  - Mit Hilfe des Indizes kann auf ein bestimmtes Element des Arrays zugegriffen werden.
- `strcpy(art.artikelBezeichnung, "Apfel");`
  - Eine Komponente ist als String (Array vom Datentyp `char`) deklariert. Die Funktionen aus der Bibliothek *string.h* können genutzt werden.

Arrays aus Strukturen

R | R | Z | N

```
struct Artikel {
   int    artikelnummer;
   char   artikelBezeichnung[20];
   double preis;
   float  istBestand;
   float  minBestand;
};

int main() {
   struct Artikel art[3];

   art[0].artikelnummer= 1;
   strcpy(art[0].artikelBezeichnung, "Apfel");

   return 0;
}
```

Hier wird eine Strukturvariable als Array deklariert.

Im Anschluss an den Namen der Strukturvariablen wird die Anzahl der Elemente in eckigen Klammern angegeben.

Es wird für x Elemente von der Struktur Speicherplatz bereitgestellt.

Zugriff auf die Komponenten

R|R|Z|N|

`art[1].artikelnummer`

`strukturvariable[Index].element`

`art[1].artikelBezeichnung[2]`

`strukturvariable[Index].element[index]`

Universität Hannover

Zugriff auf die Komponenten

- `strukturvariable[index].element`
  - Mit Hilfe des Punktoperators wird eine Komponente mit einer Strukturvariablen verbunden.
  - Es wird eine bestimmte Struktur und deren Komponente innerhalb des Arrays angesprochen.
- `strukturvariable[index].element[Index]`
  - Mit Hilfe des Punktoperators wird eine Komponente mit einer Strukturvariablen verbunden.
  - Es wird eine bestimmte Struktur und deren Komponente innerhalb des Arrays angesprochen.
  - Die Komponenten selber ist wiederum als Array definiert. Mit Hilfe des Index im Anschluss des Komponenten-Namens wird ein bestimmtes Feld innerhalb des Arrays der Komponenten angesprochen.
  - Der Index der Strukturvariablen darf nicht mit dem Index der Komponente verwechselt werden. Verwenden Sie für die Indizes unterschiedliche Namen!

Zeiger und Komponenten

R|R|Z|N

```
int main() {
   struct wert {
        int    *betrag;
        int    *rate;
   }firstwert;

   int kosten;
   int zinsen;

   firstwert.betrag = &kosten;
   firstwert.rate = &zinsen;
   *firstwert.betrag = 100.99;
   return 0;
}
```

Zeiger als Komponenten werden genauso deklariert wie normale Zeiger.

Zeiger als Komponenten müssen mit einer Adresse initialisiert werden!

Mit Hilfe des Adressierungsoperators wird der Komponente eine Adresse zugewiesen.

Mit Hilfe des Dereferenzierungsoperators wird auf den Wert an der angegebenen Adresse zugegriffen.

Beispiel: Array vom Datentyp char


```
int main() {
   struct KUNDE {
        char   *kundenname;
        char    strasse[20];
   }myKunde;

   strcpy(myKunde.kundenname, "Meier");
   strcpy(myKunde.strasse, "Schlosstrasse");

   return 0;
}
```

- Jede Strukturvariable legt einen Speicherbereich für das Array an.
- In das Array können nicht mehr Zeichen abgelegt werden als angegeben. Hier können 19 Zeichen plus das String-Endezeichen abgelegt werden.
- Wenn ein Zeiger genutzt wird, wird der String Teil der Struktur, obwohl er dort nicht gespeichert ist.

Zeiger auf Strukturen


```
struct strukturName {
        Datentyp variable1;
        Datentyp variable2;
        ...
        Datentyp variableN;
};

struct strukturName *pointer;
struct strukturName variable;
pointer = &variable;
```

- ... werden wie Zeiger auf einfache Datentypen deklariert und verwendet.
- Zeiger auf den Typ `strukturname`. Es wird keine Instanz erzeugt.

Initialisierung

R|R|Z|N

```
struct ARTIKEL {
   int    artikelnummer;
   double preis;
   float  istBestand;
   float  minBestand;
};

int main() {
    struct ARTIKEL *pointer;
    struct ARTIKEL ware;
    pointer = &ware;

   (*pointer).artikelnummer = 0.99;
   pointer->artikelnummer = 1,99;

   return 0;
}
```

Deklaration eines Zeigers vom Datentyp einer bestimmten Struktur.

Mit Hilfe des Adressoperators wird der Zeiger initialisiert. Durch die Initialisierung verweist der Zeiger auf eine Instanz eines bestimmten Strukturtyps. Der Zeiger zeigt auf den Anfang der Struktur.

Den Komponenten werden Werte zugewiesen.

Zugriff auf die Komponenten

R|R|Z|N|

- `(*pointer).strukturElement = ausdruck;`
  - `*pointer` verweist auf eine bestimmte Strukturvariable.
  - Mit Hilfe des Punktoperators kann auf die einzelnen Elemente zugegriffen werden.
  - Der Zeiger muss geklammert werden, weil der Punktoperator eine höhere Priorität hat als der Dereferenzierungsoperator.
- `pointer->zahl = ausdruck`
  - Der Elementverweis-Operator besteht aus einem Bindestrich gefolgt von einen Größer-Zeichen. Zwischen dem Größer-Zeichen und dem Bindestrich darf kein Leerzeichen stehen.
  - Der Operator befindet sich zwischen dem Zeigernamen und der Komponente.
  - Mit Hilfe des Zeigers wird auf eine bestimmte Strukturvariable verwiesen. Die Strukturvariable hat bestimmte Merkmale, die über den Elementverweis-Operator mit dem Zeiger verknüpft werden.
  - Der Dereferenzierungsoperator wird nicht benötigt.

Vorteile des Elementverweis-Operators

R | R | Z | N

```
int main() {
        struct wert {
                int     *betrag;
                int     *rate;
        };

        struct wert firstwert;
        struct wert *nextwert;
        int kosten;
        int zinsen;

        firstwert.betrag = &kosten;
        *firstwert.betrag = 100.99;

        nextwert = &firstwert;
        (*nextwert).rate = &zinsen;

       *(*nextwert).rate = 200;
        *nextwert->rate = 300;

        return 0;
}
```

Das zweite Sternchen wird für das Strukturelement benötigt.

Universität Hannover

Strukturelemente für Funktionen nutzen

- Der Wert der Komponente wird übergeben.
- Es ist ohne Bedeutung, dass die Komponente Bestandteil einer Struktur ist.
- Der Datentyp der Komponente muss mit dem Funktionstyp oder dem Typ des Arguments übereinstimmen.

Strukturen für Funktionen nutzen

R|R|Z|N|

- Methode "Call-by-Value:
  - Es wird eine Kopie der Struktur angelegt.
  - Änderungen an der Kopie haben keine Auswirkungen auf das Original.
  - Der Datentyp des Arguments muss mit dem Datentyp des Parameters übereinstimmen.
  - Die Übergabe erfolgt über den Stack. Bei älteren Rechnern kann es zu Laufzeit- und Speicherproblemen kommen.
- Methode "Call-by-Reference"
  - Es wird der Zeiger auf eine Struktur übergeben.
  - Änderungen in der Funktion an der Struktur haben Auswirkungen auf das Original.
  - Häufigste Form.

Kopie einer Struktur übergeben

R|R|Z|N|

```
#include <stdio.h>
struct KOORDINATEN{
        int xAchse;
        int yAchse;
};

void ausgabe(struct KOORDINATEN);

int main(){
        struct KOORDINATEN messung;

        messung.xAchse = 10;
        messung.yAchse = 10;
        ausgabe(messung);
        return 0;
}

void ausgabe(struct KOORDINATEN messpunkt){
                printf("\nx-Achse: %d", messpunkt.xAchse);
                printf("\ny-Achse: %d", messpunkt.yAchse);
                printf("\n");
}
```

Zeiger nutzen

R|R|Z|N

```
struct KOORDINATEN{
        int xAchse;
        int yAchse;
};

void ausgabe(struct KOORDINATEN *);

int main(){
        struct KOORDINATEN messung;
        int *ptrStruct;

        ptrStruct = &messung;
        messung.xAchse = 10;
        messung.yAchse = 10;
        ausgabe(ptrStruct);
        return 0;
}
void ausgabe(struct KOORDINATEN *messpunkt){
                printf("\nx-Achse: %d", (*messpunkt).xAchse);
                printf("\ny-Achse: %d", (*messpunkt).yAchse);
                printf("\n");
}
```

Unions

```
union unionName {
    Datentyp variable1;
    Datentyp variable2;
    ...
    Datentyp variableN;
};
```

- ... werden genauso deklariert und verwendet wie Strukturen.
- Alle Elemente einer Union belegen den gleichen Speicherplatz.
- Die Größe einer Union orientiert sich am größten Element der Union.
- Die Elemente einer Union liegen sozusagen übereinander.
- In einer Union kann immer nur eins der deklarierten Elemente genutzt werden.

Instanz einer Union

R|R|Z|N|

```
union unionName {
    Datentyp variable1;
    Datentyp variable2;
    ...
    Datentyp variableN;
};
union unionName variable;
```

- Es wird genauso wie bei einer Struktur eine Variable erstellt.
- Die Variable stellt eine Instanz der Union dar.

Beispiel

R|R|Z|N

```
#include <stdio.h>
#define CHARACTER 'C'
#define INTEGER 'I'

struct ArtDef{
   char typ;
   union Ascii {
      char charZeichen;
      int intZeichen;
   }zeichen;
};

void ausgabe(struct ArtDef z);
```

```
int main(){
   struct ArtDef wert;
   wert.typ = CHARACTER;
   wert.zeichen.charZeichen = 'x';

   ausgabe (wert);

   wert.typ = INTEGER;
   wert.zeichen.intZeichen = 'x';

   ausgabe (wert);
   system("Pause");
   return 0;
}
```

Beispiel

R | R | Z | N

```
void ausgabe(struct ArtDef z) {
    printf("\nDer generische Wert ist ...");

    switch(z.typ) {
      case CHARACTER:
         printf("%c\n", z.zeichen.charZeichen);
         break;
      case INTEGER:
         printf("%d\n", z.zeichen.intZeichen);
         break;
     default:
        printf("Typ unbekannt");
    }
}
```

Aufzählung


```
enum variable{symbol, ... , symbol};
```

- ... oder Enumeration.
- Mit Hilfe von `enum` wird eine Aufzählung eingeleitet.
- Jede Aufzählung besitzt einen Namen.
- Die zu einer Aufzählung gehörenden Symbole werden als Block zusammengefasst.
- Die einzelnen Symbole werden mit Hilfe von Kommata getrennt.
- Jedes Symbol steht für einen Integer-Wert.
  - Dem ersten Symbol wird der Wert 0 zugewiesen.
  - Dem nächsten Symbol wird ein Wert zugewiesen, der um eins höher ist als sein Vorgängerwert.

Werte festlegen

R|R|Z|N|

```
enum variable{symbol, symbol = 3, ... , symbol};
```

- Mit Hilfe des Gleichheitszeichen kann jedem Symbol ein Wert zugewiesen werden.
- Falls nachfolgende Symbole nicht initialisiert werden, wird ihnen der Wert des Vorgängers plus eins zugewiesen.

Beispiel

R|R|Z|N|

```
#include <stdio.h>

enum BOOL {FALSE, TRUE};

int main(){
   int zahl;

   printf("Zahl von 1 bis 9 eingeben: ");
   scanf("%d", &zahl);

   if(zahl == FALSE){
               printf("Die Zahl ist 0\n");
   }
   else if (zahl < FALSE) {
        printf("Negative Zahl\n");
   }

   return 0;
}
```

Folgende Werte werden zugewiesen:
FALSE = 0
TRUE = 1

Neue Typen definieren

R|R|Z|N|

```
typedef Typendefinition variable;
```

- ... erzeugt einen neuen Namen für einen bestimmten Datentyp.
- Es wird kein neuer Datentyp erzeugt. Ein existierender Datentyp wird umbenannt.
- Vorteile:
  - Das Programm wird portabler. Bei maschinenabhängigen Programmen muss nur die typedef-Anweisung geändert werden, um alle anderen Variablen einen neuen Datentyp zu zuweisen.
  - Das Programm wird durch sprechende Namen lesbarer.

Beispiel

R|R|Z|N

```
#include <stdio.h>
#define CHARACTER 'C'
#define INTEGER 'I'

struct ArtDef{
   char typ;
   union Ascii {
      char charZeichen;
      int intZeichen;
   }zeichen;
};

typedef struct ArtDef ZEICHEN;

void ausgabe(ZEICHEN z);
```

```
int main(){
   ZEICHEN wert;
   wert.typ = CHARACTER;
   wert.zeichen.charZeichen = 'x';

   ausgabe (wert);

   wert.typ = INTEGER;
   wert.zeichen.intZeichen = 'x';

   ausgabe (wert);
   return 0;
}
```

Beispiel

R|R|Z|N

```
void ausgabe(ZEICHEN z) {
   printf("\nDer generische Wert ist ...");

   switch(z.typ) {
     case CHARACTER:
        printf("%c\n", z.zeichen.charZeichen);
        break;
     case INTEGER:
        printf("%d\n", z.zeichen.intZeichen);
        break;
    default:
       printf("Typ unbekannt");
   }
}
```

Bitfelder

- Zugriff auf einzelne Bits.
- Falls der Speicherplatz beschränkt ist, können mehrere boolsche Variablen in einem Byte gespeichert werden.
- In einem Byte werden verschiedene Statusinformationen codiert.
- Verschlüsselungroutinen greifen auf die einzelnen Bits zu.
- Eine Bit-Definition sieht folgendermaßen aus: `typename : length`.

Beispiel

R|R|Z|N

```
#include <stdio.h>
#include <stdlib.h>

int main() {

  struct {
    unsigned INTERRUPT_WAITING : 1;
    unsigned INP_BUFFER_FULL   : 1;
    unsigned                   : 4;
     /*Frei, nicht belegt*/
    unsigned OUT_BUFFER_EMPTY  : 1;
  } status_reg;

    if (status_reg.INP_BUFFER_FULL )
      printf("Senden starten\n");
    .........
}
```

Für die hardwarenahe Programmierung will man häufig auf einzelne Bits in einem Register zugreifen, z. B. um den Status eines Controllers abzufragen.

Zugriff über Strukturname.Bitbezeichner

Dynamische Datenstrukturen

- ... nutzen häufig Arrays oder Strukturen.
- ... werden aus einzelnen Elementen (Knoten) gebaut, die eine Beziehung zueinander haben.
- Die Beziehung zwischen den Elementen werden mit Hilfe von Zeigern realisiert.
- ... ändern die Speichergröße während der Programmausführung.
- ... benötigen Funktionen für das Einfügen, Sortieren und Löschen von Elementen. Die Funktionen werden selbst definiert.

Möglichkeiten

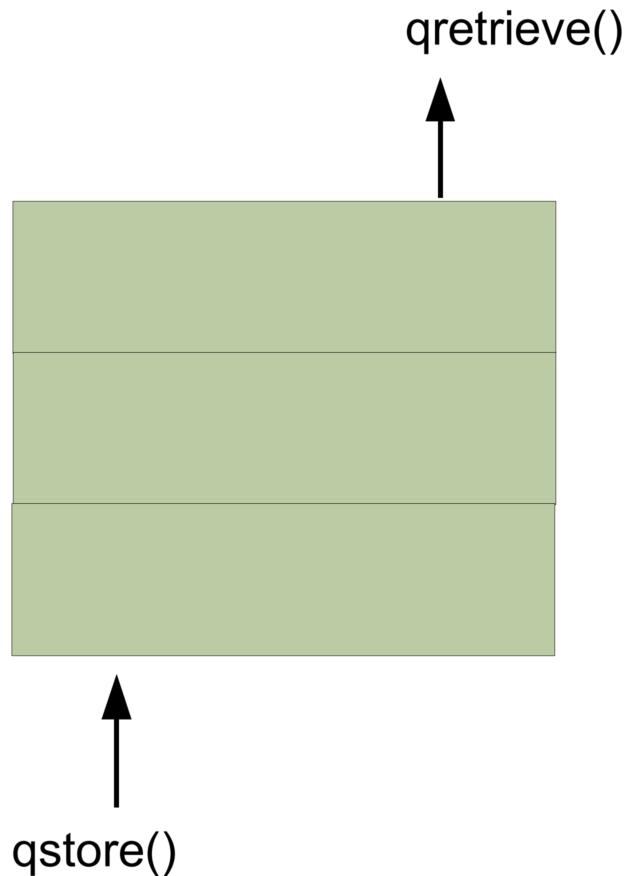
- Listen
  - Einfach und doppelt verkettete Listen
  - Queue (Warteschlange)
  - Stack (Kellerspeicher)
- Bäume
- Graphen

Queues

R|R|Z|N|

- ... sind lineare Listen.
- ... nutzen das FIFO (First in – First out)-Prinzip.
- ... entsprechen
  - ... einer Warteschlange an einer Supermarkt-Kasse.
  - ... Druckaufträge, die an einen Drucker geschickt werden.
- ... benötigen folgende Funktion:
  - `qstore()` zum Speichern von Elementen am Ende der Liste.
  - `qretrieve()` zum Entnehmen des ersten Elements der Liste. Anschließend wird das Element entfernt.

Funktionsweise



Aktion	Inhalt der Queue
qstore(1)	(1)
qstore(2)	(1) (2)
qstore(3)	(1) (2) (3)
qretrieve() ruft 1 ab	(2) (3)
qstore(4)	(2) (3) (4)
qretrieve() ruft 2 ab	(3) (4)
qretrieve() ruft 3 ab	(4)

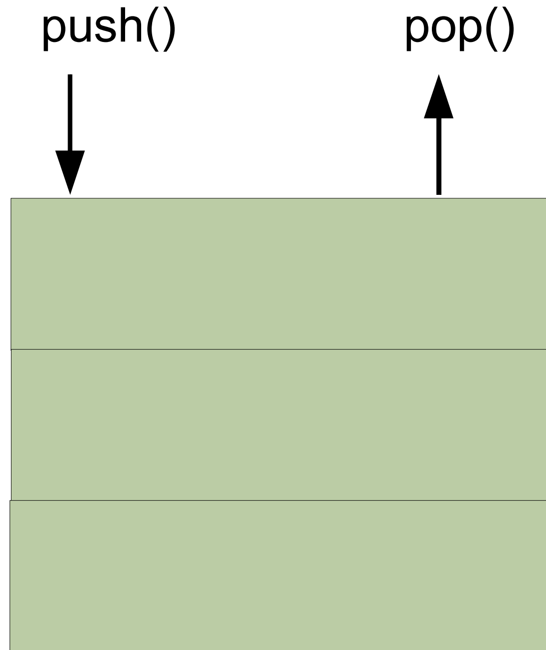
Stacks

R|R|Z|N

- ... oder Stapelspeicher.
- ... werden nach dem LIFO (Last in – First out)-Prinzip abgearbeitet.
- Parameter und lokale Variablen werden beim Aufruf einer Funktion auf dem Stack abgelegt.
- ... entsprechen
  - ... einem Papierstapel. Das oben auf liegende Blatt wird zuerst vom Stapel herunter genommen.
  - ... einer Liste von Aktionen für die Schaltfläche "Rückgängig". Die zuletzt ausgeführte Aktion wird als erstes rückgängig gemacht.
- ... benötigen folgende Funktionen:
  - `push()`, um Elemente auf dem Stack abzulegen
  - `pop()`, um Elemente vom Stack zu entfernen.

Funktionsweise

R|R|Z|N



Aktion	Inhalt des Stacks
push(1)	(1)
push(2)	(2) (1)
push(3)	(3) (2) (1)
pop() ruft 3 ab	(2) (1)
push(4)	(4) (2) (1)
pop() ruft 4 ab	(2) (1)
pop() ruft 2 ab	(1)
pop() ruft 1 ab	( )

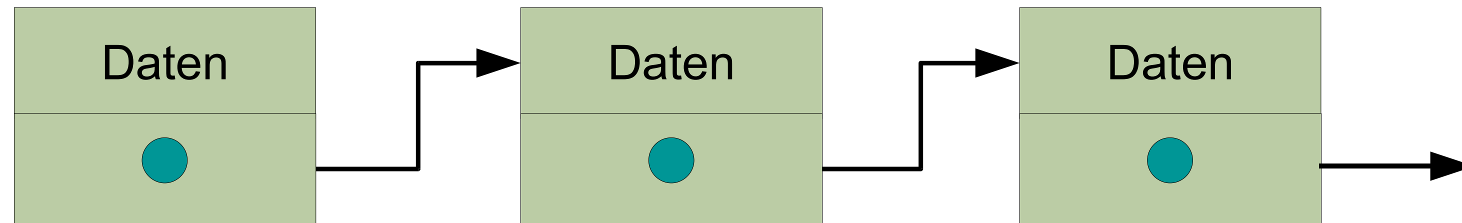
Verkettete Listen

- Listen bestehen aus Elementen, die in einer bestimmten Reihenfolge verkettet sind.
- Jede Liste besteht aus
  - ... einem Startpunkt und
  - ... einer variablen Anzahl von Elementen.
  - Jedes der Elemente gehört der gleichen Kategorie an.
- Jedes Element besteht aus
  - ... den darin gespeicherten Daten und
  - ... ein oder zwei Zeigern.

Möglichkeiten

- Einfach verkettete Listen
  - Jedes Element kennt seinen Nachfolger.
  - Die Liste kann mit Hilfe von Zeigern durchlaufen werden.
- Doppelt verkettete Listen
  - Jedes Element kennt seinen Nachfolger und seinen Vorgänger.
  - Die Liste kann mit Hilfe von Zeigern durchlaufen werden.
- Ringförmige Listen
  - Jedes Element hat einen bekannten Nachfolger und eventuell einen bekannten Vorgänger.
  - Das letzte Element ist mit dem ersten Element durch einen Zeiger verbunden.

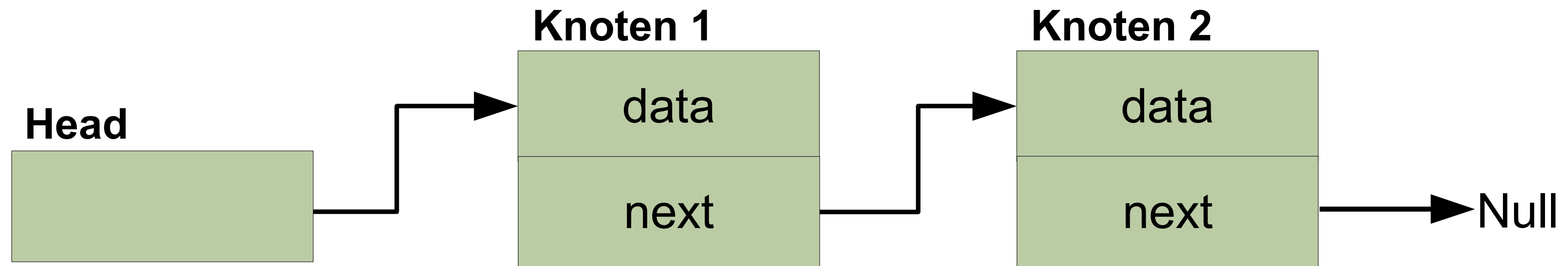
Einfach verkettete Listen



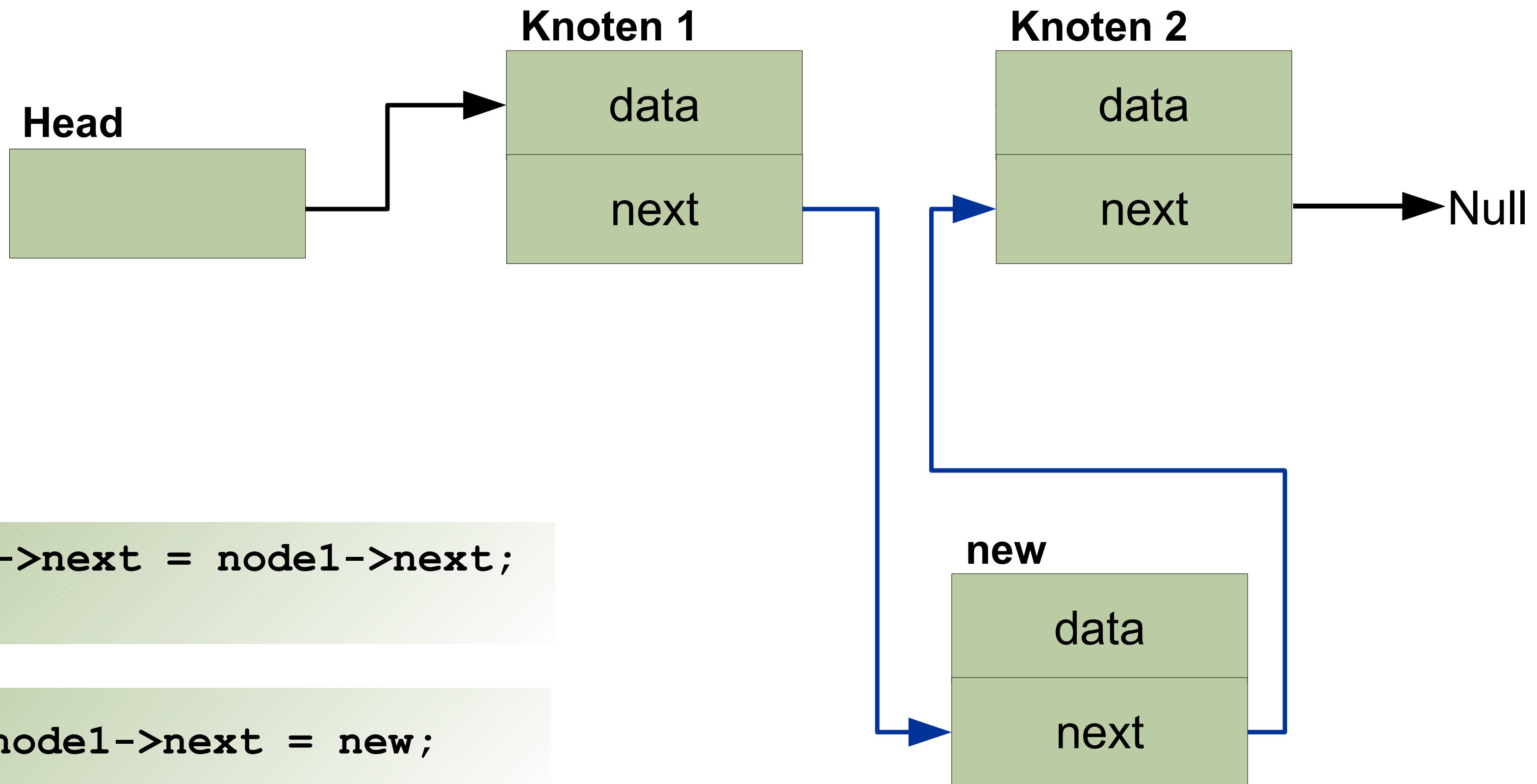
Struktur


```
struct NODE{
        int data;
        struct NODE *next;

}
```



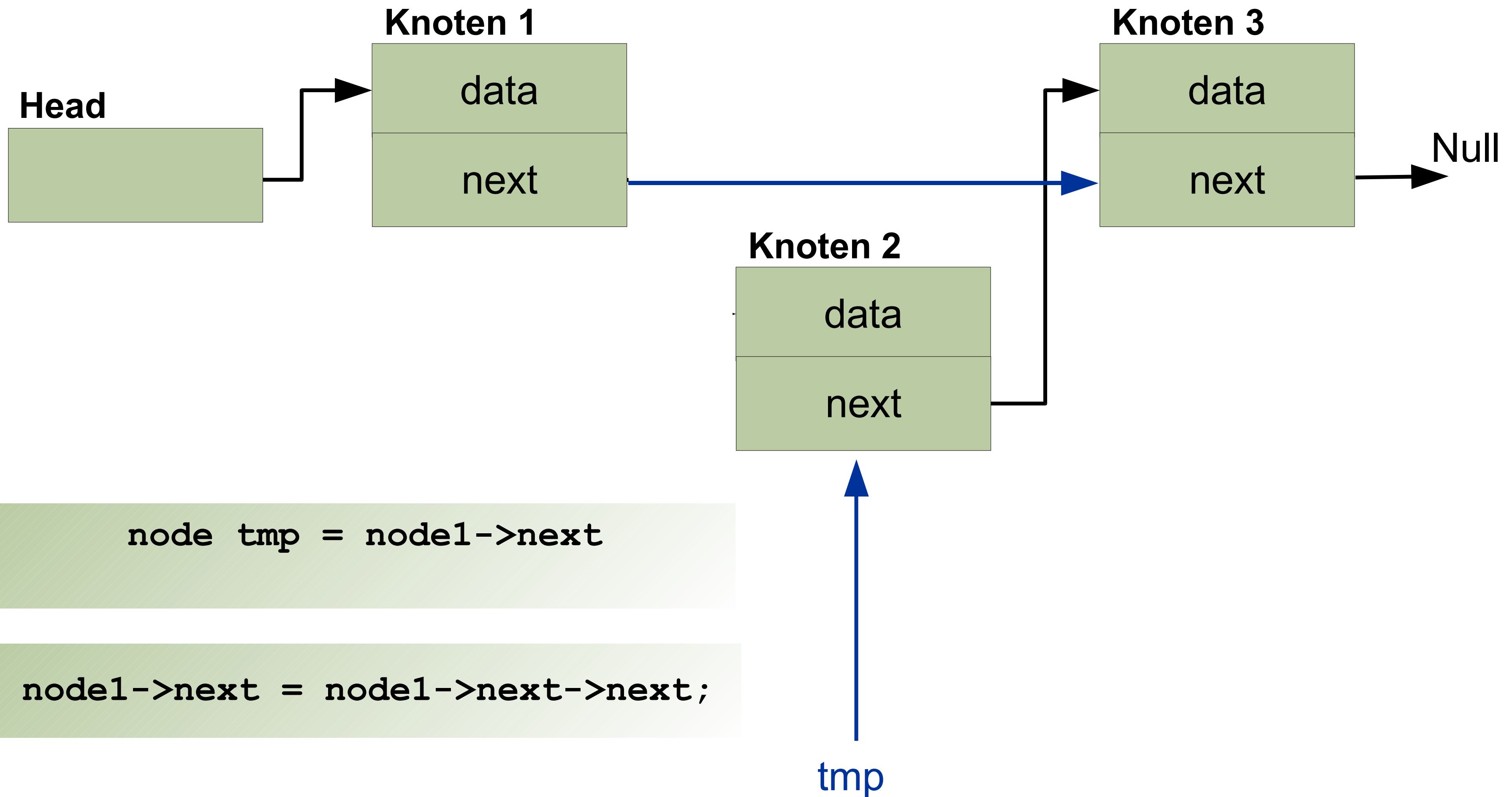
Element einfügen



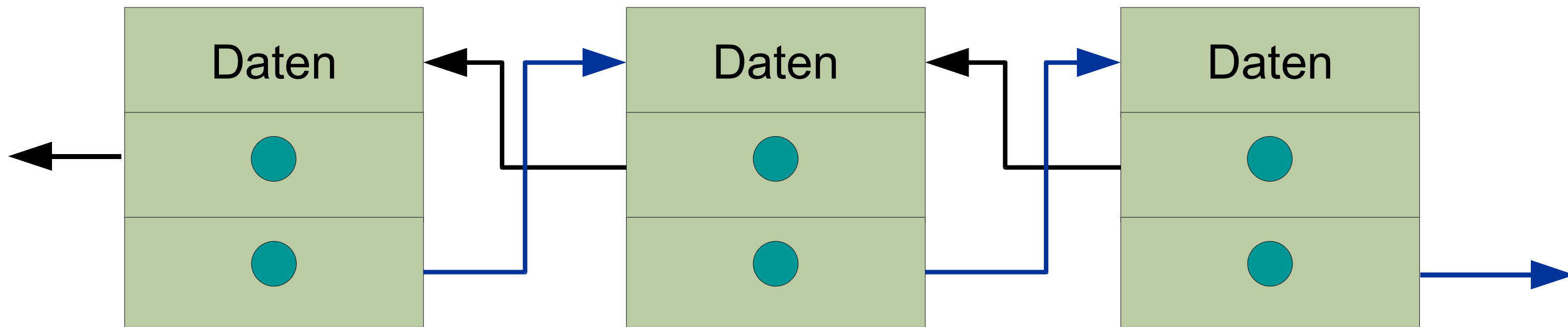

```
new->next = node1->next;
```

```
node1->next = new;
```

Element löschen



Doppelt verkettete Listen

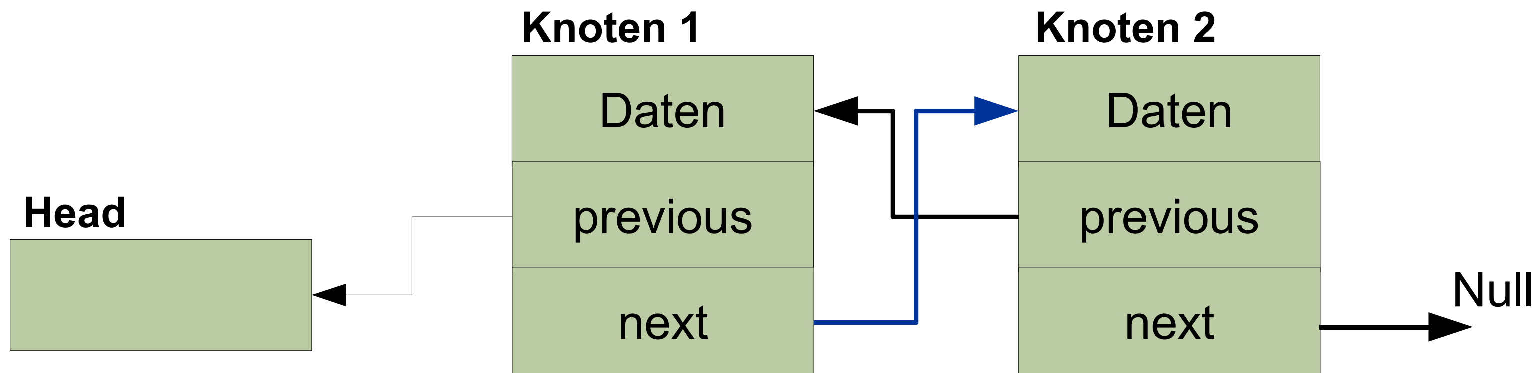


Struktur

R | R | Z | N

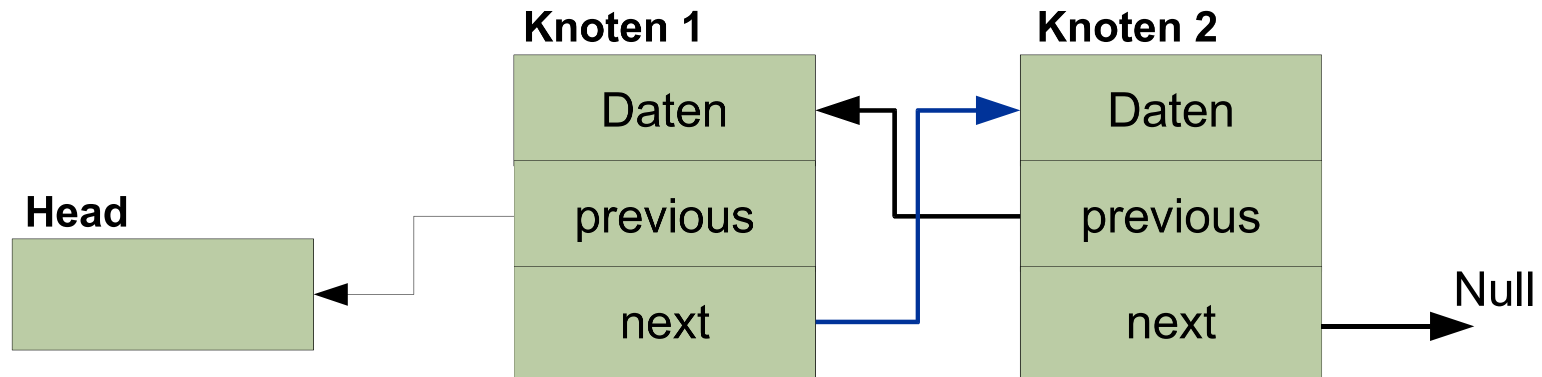
```
struct NODE{
        int data;
        struct NODE *next;
        struct NODE *previous;

}
```



Element einfügen

R|R|Z|N|



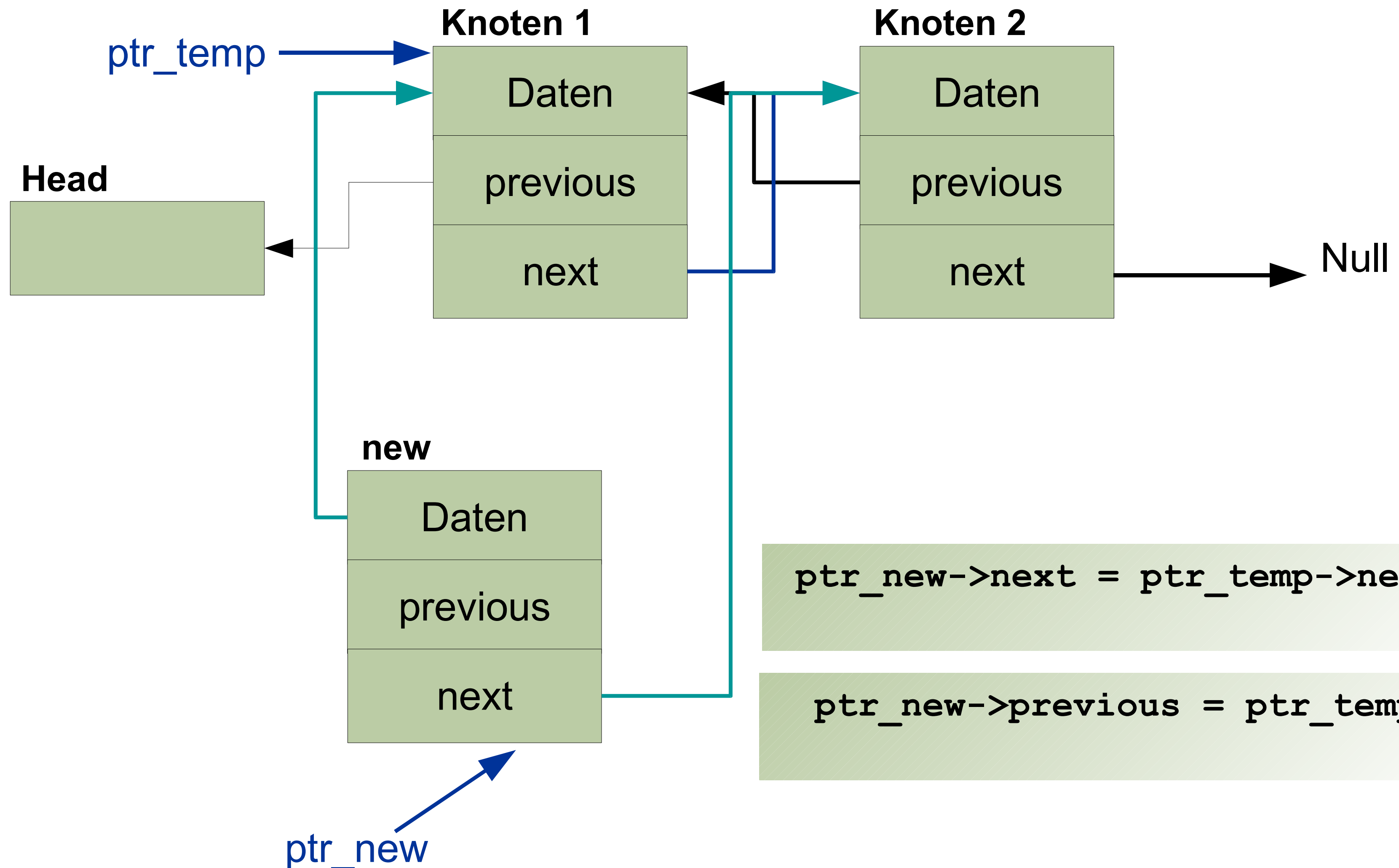
```
struct NODE *ptr_temp
```

```
ptr_new =
 (struct NODE *)malloc(sizeof(struct NODE));
```

ptr_new

Element einfügen

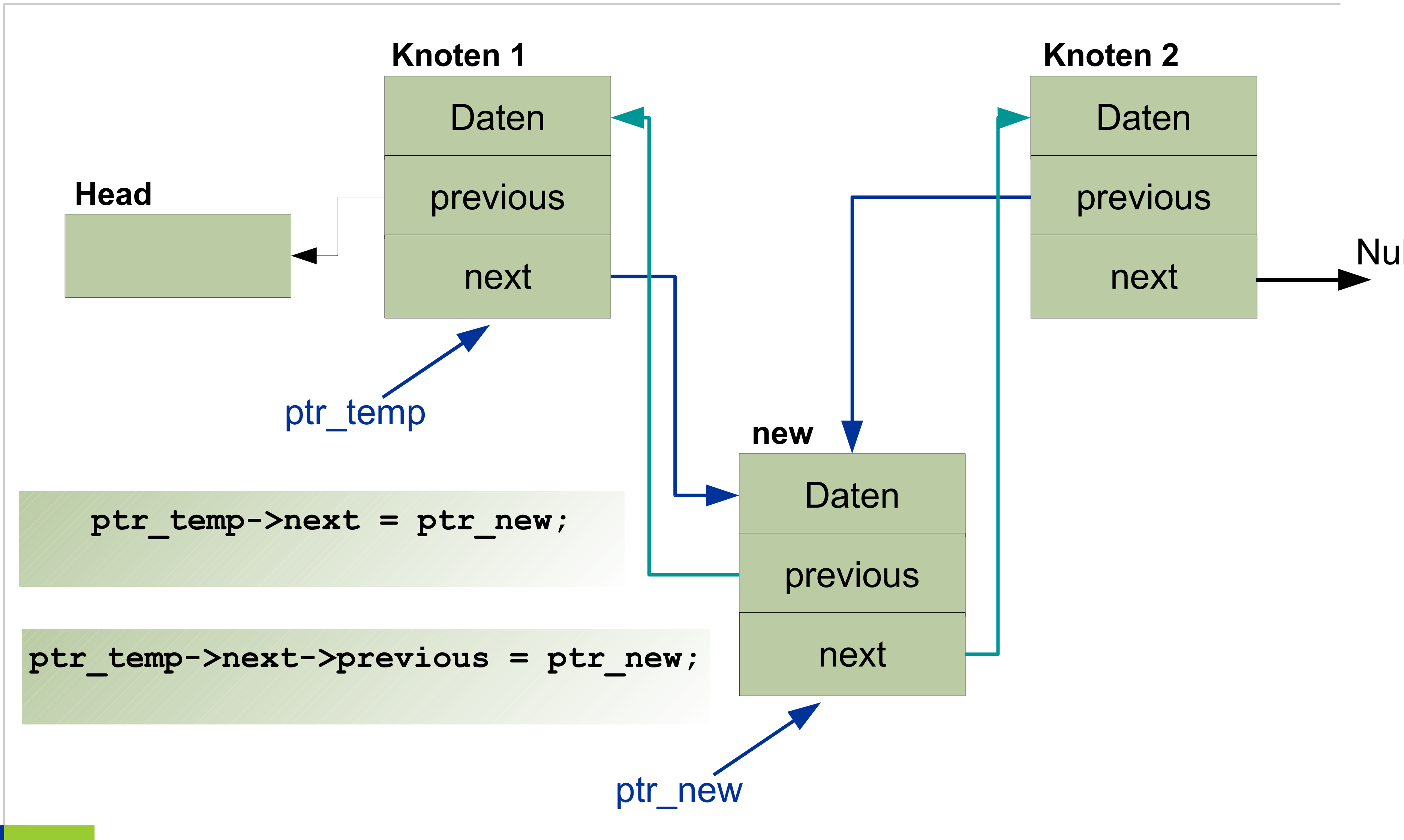
R|R|Z|N|




```
ptr_new->next = ptr_temp->next;
```

```
ptr_new->previous = ptr_temp;
```

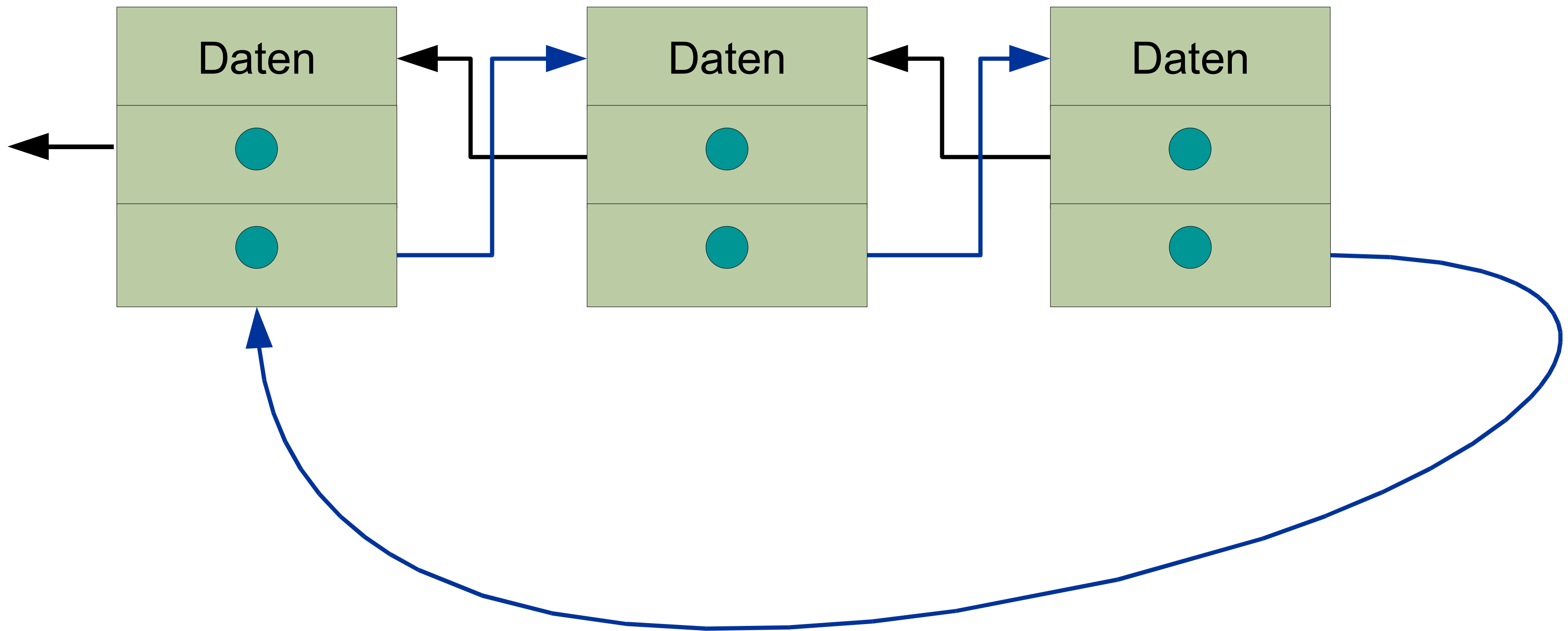
Element einfügen




```
ptr_temp->next = ptr_new;
```

```
ptr_temp->next->previous = ptr_new;
```

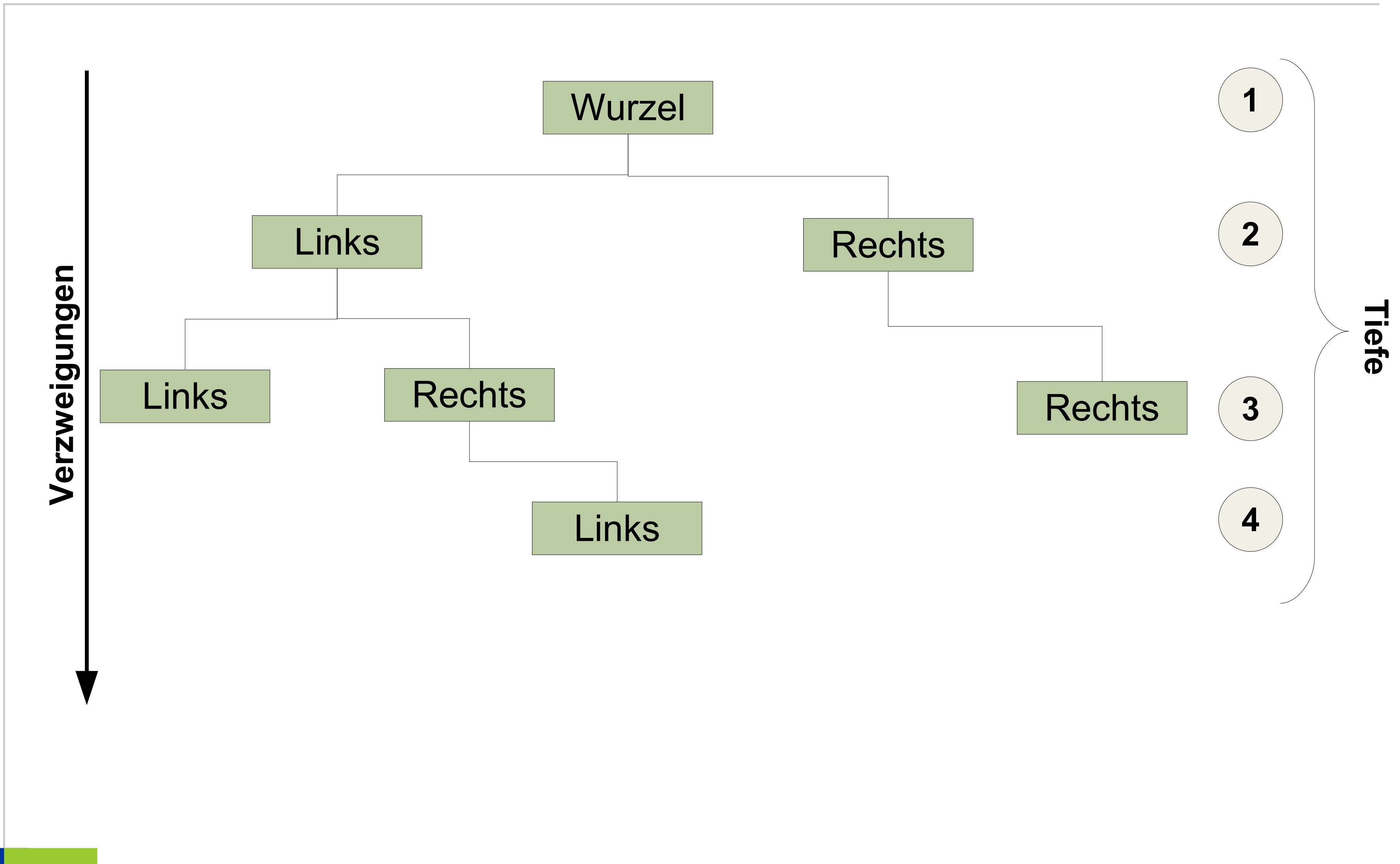
Ringförmig verkettete Listen



Binäre Bäume

- ... werden für die Sortierung von Zahlen oder Wörtern genutzt.
- ... bestehen aus Knoten (Node), welche die verschiedenen Werte enthalten. Als Knoten wird ein Datenelement bezeichnet.
- ... haben eine Wurzel als obersten Knoten.
- Die Teile des Baums heißen Äste (subtree).
- ... haben Nachfolger (Verzweigungen).
  - Jeder Knoten kann einen Verweis auf seinen linken und / oder rechten Nachfolger besitzen.
  - Knoten, die keine Nachfolger haben, nennt man Blatt (terminal node / leaf).
- Die Tiefe (height) des Baumes ist gleich der Anzahl der Ebenen, die ab seiner Wurzel existierten.

Beispiel



Beschreibung eines Baums

- Jeder Knoten eines Baums besitzt höchstens zwei Nachfolger.
- Es gibt ein Knoten, die so genannte Wurzel (root). Die Wurzel besitzt keinen Vorgänger.
- Die Gesamtheit eines Nachfolgeknotens bilden zusammen einen Teilbaum.