

# Kapitel 15

## Ausgewählte Datenstrukturen

## Worum geht es in diesem Abschnitt?

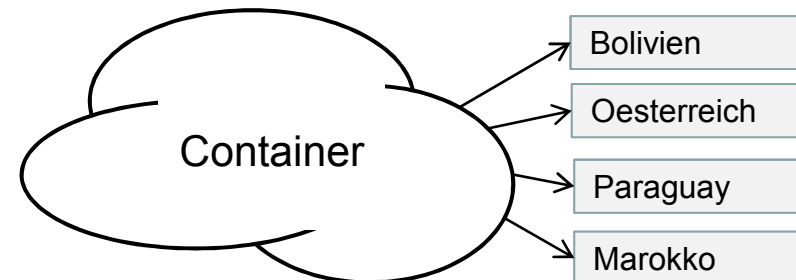
In diesem Abschnitt betrachten wir eine vereinfachte Länderspieldatei, die für jeden Gegner der Nationalmannschaft nur den Ländernamen und die Anzahl der Länderspiele enthält. Die Reihenfolge der Länder ist dabei nicht alphabetisch sondern zufällig gewählt. Für die Datensätze in der Datei haben wir auch bereits eine Datenstruktur `gegner` angelegt.

Wir wollen jetzt einen "Container" programmieren, in dem wir eine unbeschränkte Anzahl von Gegnern speichern können. Der Container soll dabei folgende Funktionen haben:

- Erzeuge einen leeren Container
- Füge einen Gegner zum Container hinzu
- Suche einen Gegner im Container
- Lösche den Container mit seinem Inhalt

...	
Bolivien	1
Oesterreich	38
Paraguay	2
Marokko	4
	25

```
struct gegner
{
    char *name;
    int spiele;
};
```



Wir wollen verschiedene Speichertechniken für den Container entwickeln und diese bezüglich ihrer Laufzeit- und Speichereffizienz miteinander vergleichen.

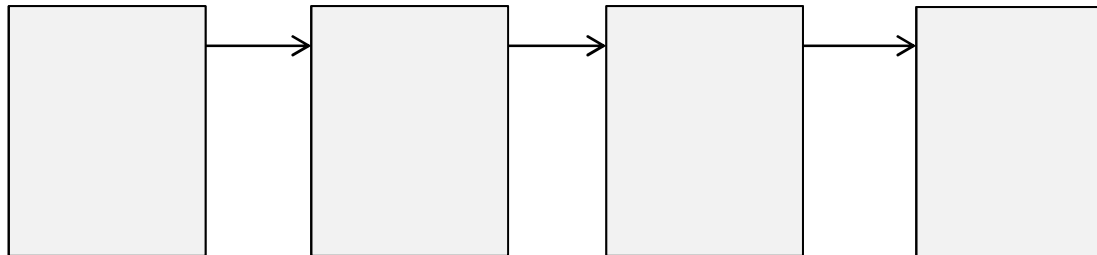
Implementiert wird der Container als:

- Liste
- Binärbaum
- Treap
- Hashtabelle

## Listen 1

Eine **Liste** ist eine endliche Menge von (Daten-)Elementen, die durch eine Nachfolgeoperation miteinander verbunden oder verkettet sind. Über die Nachfolgeoperation sind dann die Begriffe Nachfolger und Vorgänger eines Elements definiert. Es gibt genau ein Element, das keinen Vorgänger hat. Dieses Element heißt Listenanfang. Weiterhin gibt es genau ein Element, das keinen Nachfolger hat. Dieses Element heißt Listenende. Jedes Element der Liste ist vom Listenanfang aus durch eine genau bestimmte Anzahl von Nachfolgeoperationen erreichbar.

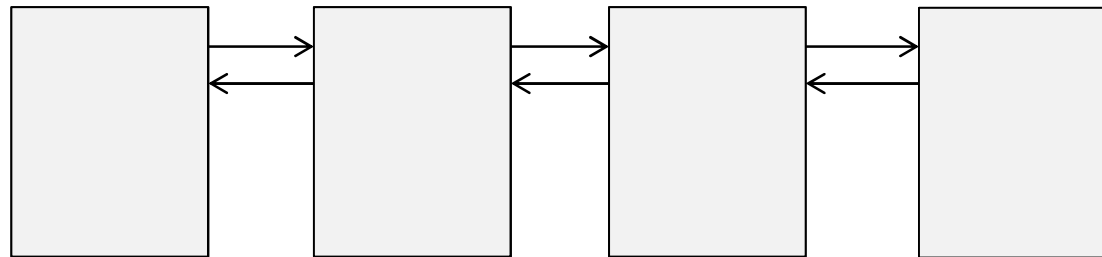
Als grafische Notation für ein Element wählen wir ein Rechteck. Die Nachfolgeoperation visualisieren wir durch Pfeile:



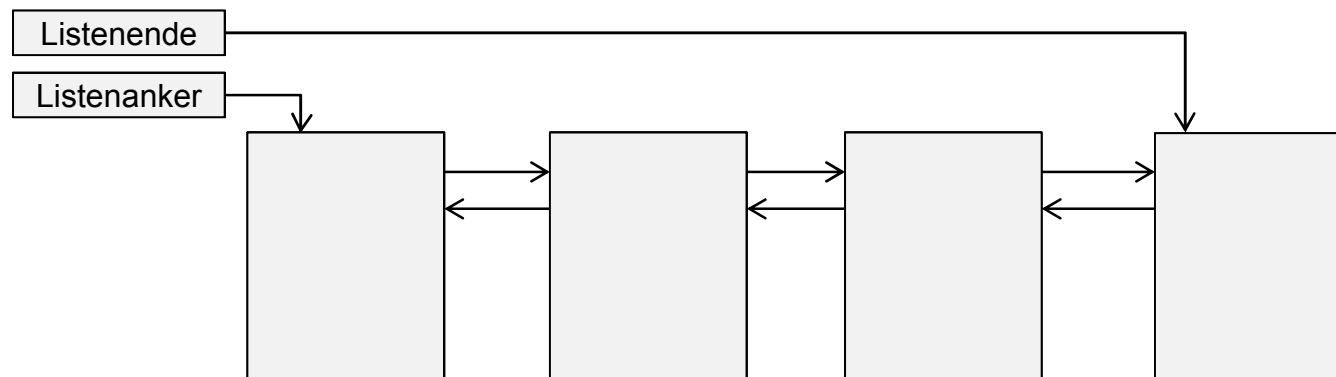
Listen sind eine häufig anzutreffende und sehr flexible Form der Speicherung von vorrangig sequentiell zu verarbeitenden Daten. Die Daten können dabei durchaus inhomogen sein, müssen also untereinander weder die gleiche Struktur noch die gleiche Größe haben. Jedes Datenelement enthält einen Zeiger auf das nächstfolgende Element. Das heißt, in jeder Datenstruktur ist die Adresse der nachfolgenden Datenstruktur eingetragen. Die Liste wird durch den NULL-Zeiger abgeschlossen. Der NULL-Zeiger ist ein Zeiger mit Wert 0. Da 0 nicht als normaler Adresswert vorkommt, kann man mit diesem Wert das Listenende markieren.

## Listen 2

Enthält jedes Element der Liste auch einen Rückverweis auf seinen Vorgänger, so sprechen wir von einer **doppelt verketteten Liste**.



Häufig gehören zu einer Liste noch zwei weitere Zeiger: Ein Zeiger auf das erste Element (Anker), um für eine sequentielle Verarbeitung in die Liste einsteigen zu können, ein weiterer Zeiger auf das letzte Element, um am Ende anfügen zu können, ohne die ganze Liste sequentiell durchlaufen zu müssen:



Dies ist eine logische Sicht. Im Speicher können die einzelnen Listenelemente in beliebiger Reihenfolge oder verstreut liegen.

## Listen oder Arrays 1

Wann verwendet man eine Liste, wann einen Array?

Operation	Array	einfach verkettete Liste	doppelt verkettete Liste
Wahlfreier Zugriff	Sehr gut durch Zugriff über Index	Schlecht, da die Liste sequentiell durchlaufen werden muss	
Einfügen hinter einem Element	Schlecht, da alle nachfolgenden Elemente verschoben werden müssen. Wenn kein Platz im Array ist, sogar sehr aufwendig	Sehr einfach durch Einketten des neuen Elements	
Einfügen vor einem Element		Aufwendig, da der Vorgänger gesucht werden muss	Einfach durch Einketten des neuen Elements
Entfernen hinter einem Element	Schlecht, da alle nachfolgenden Elemente verschoben werden müssen	Sehr einfach durch Ausketten des Elements	
Entfernen vor einem Element		Aufwendig, da der Vorgänger gesucht werden muss	Einfach durch Ausketten des Elements
Gehe zum Nachfolger	Einfach durch Erhöhung des Index	Einfach durch Ausnutzung der Vorwärtsverkettung	
Gehe zum Vorgänger	Einfach durch Erniedrigung des Index	Aufwendig, da der Vorgänger gesucht werden muss	Einfach durch Ausnutzung der Rückwärtsverkettung
Vertauschen zweier Elemente	Einfach	Aufwendig, da zur Aktualisierung der Verkettung eine Vorgängersuche erforderlich ist	Etwas verzwickter, aber nicht so aufwendig wie bei einfach verketteten Listen

Operationen auf Listen können häufig durch Ändern von Zeigerwerten ausgeführt werden, ohne dass große Datenmengen im Speicher bewegt werden müssen.

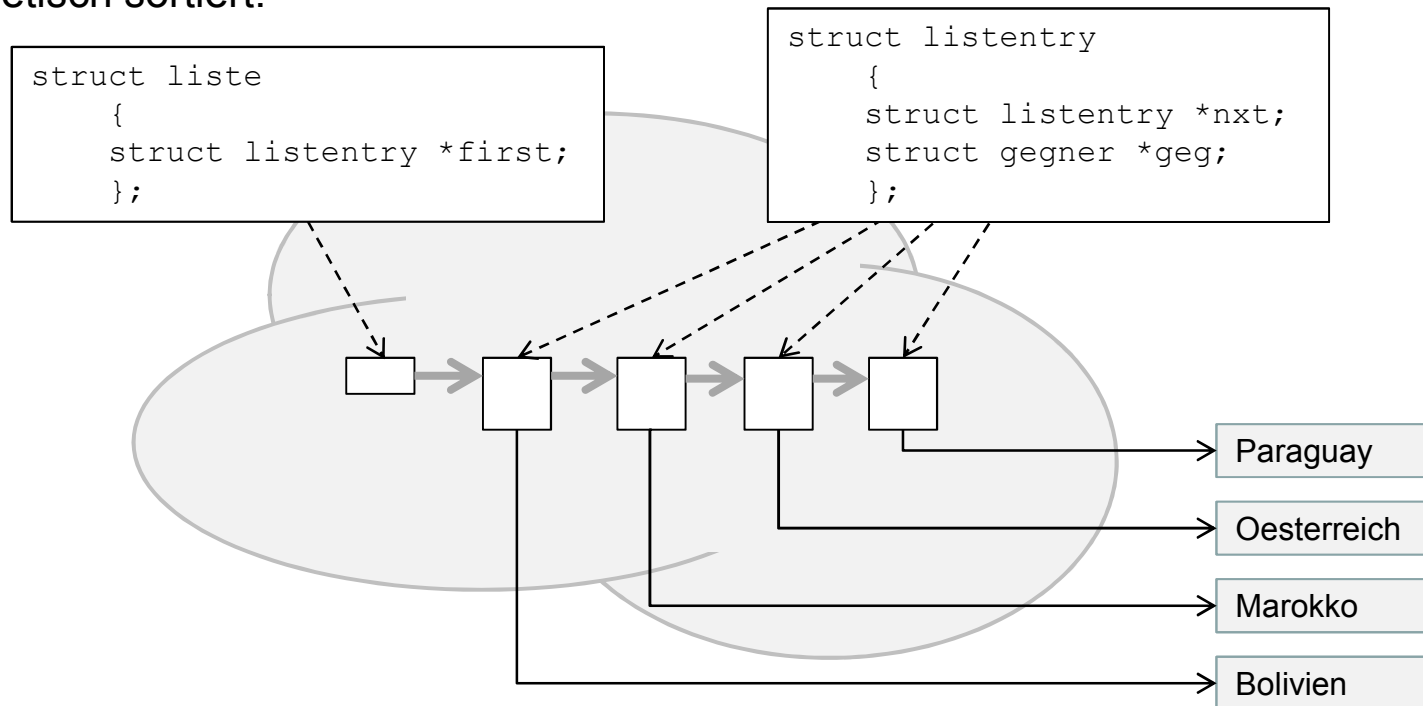
## Listen oder Arrays 2

Bei dynamisch wachsenden und schrumpfenden, vielleicht sogar inhomogenen Datenbeständen stark variierender Anzahl mit häufigen Einsetz- und Löschoperationen und vorrangig sequentiellm Zugriff sind Listen zu bevorzugen.

Bei homogenen Datenbeständen fester Anzahl, die einen effizienten und wahlfreien Zugriff erfordern, sind Arrays die erste Wahl.

## Der Container als Liste - Datenstrukturen

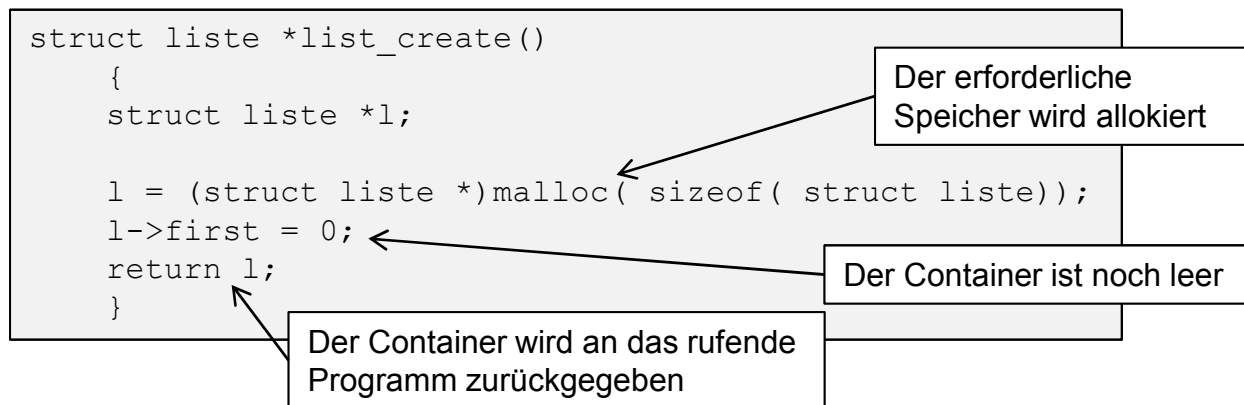
Im Container implementieren wir eine einfach verkettete Liste. Die Listeneinträge werden dabei alphabetisch sortiert.



Der Container besteht aus einem Header (`struct liste`) der nur einen Zeiger auf das erste Listenelement (`first`) enthält. Die eigentliche Liste ist eine Verkettung von Listenelementen (`struct listentry`), die jeweils einen Zeiger auf den durch sie verwalteten Gegner (`geg`) und einen Zeiger auf das nächste Listenelement (`nxt`) enthalten.

## Der Container als Liste – Erzeugen eines leeren Containers

Alle Datenstrukturen im Container werden dynamisch erzeugt. Ein leerer Container besteht aus einem Header (`struct liste`) dessen `first`-Zeiger den Wert 0 hat, da es noch keine Listeneinträge gibt:



Die erforderliche Struktur wird allokiert, initialisiert und an das rufende Programm zurückgegeben.

Bei jedem Aufruf der `list_create`-Funktion wird ein neuer Container erzeugt. Ein Anwendungsprogramm kann daher mehrere Container erzeugen und unabhängig voneinander verwenden:

```
struct liste *container1;
struct liste *container2;

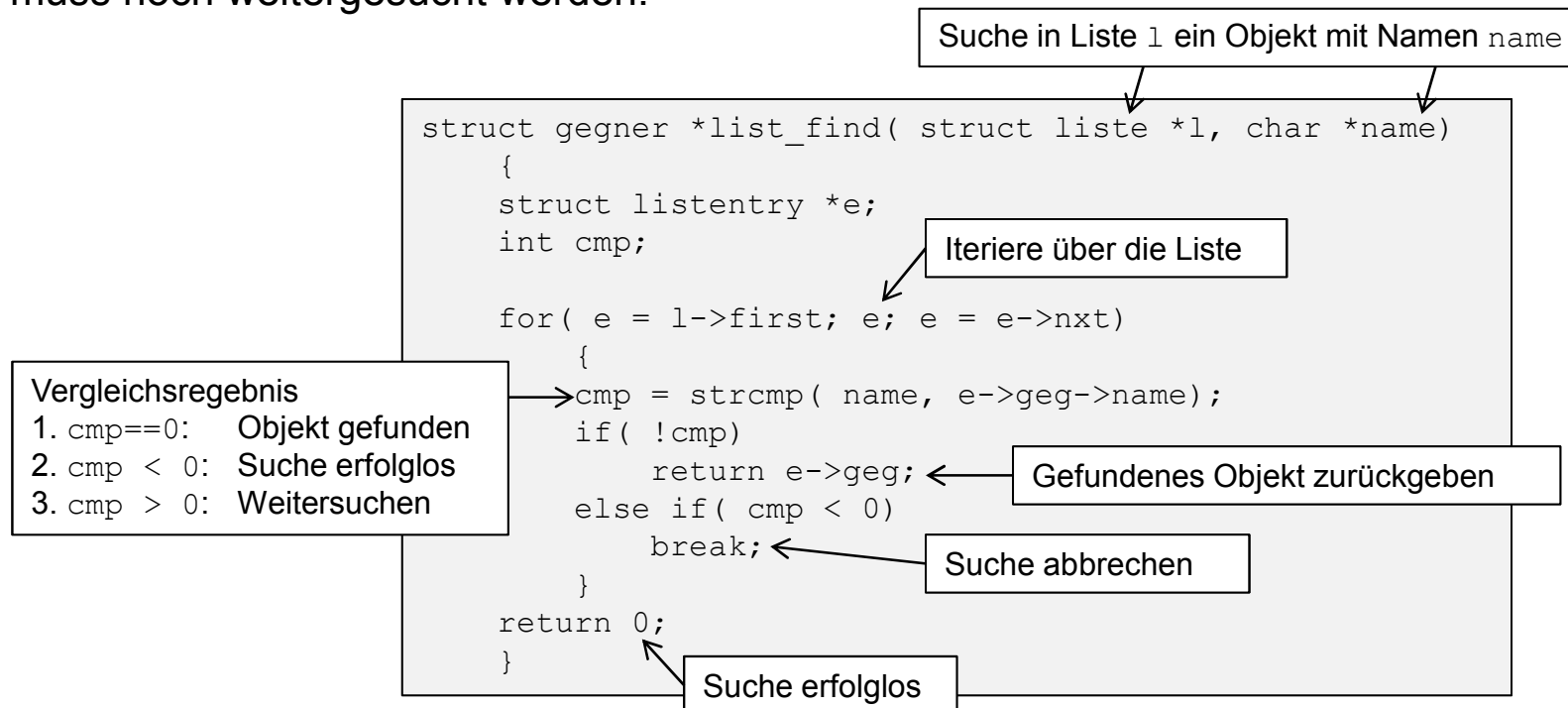
container1 = list_create();
container2 = list_create();
```



## Der Container als Liste – Finden eines Elements

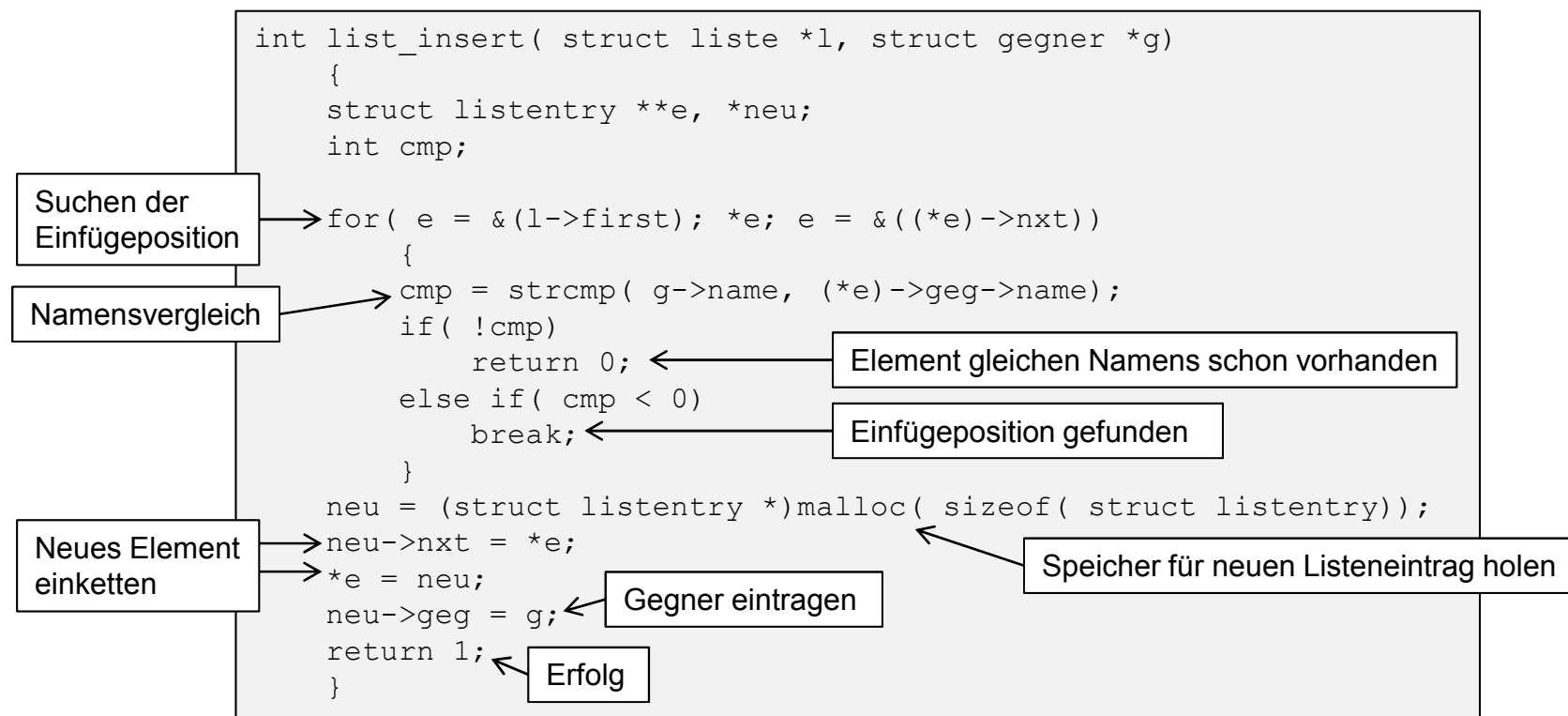
Um ein Element zu finden, wird über die Liste iteriert. Für jedes Listenelement wird der Elementname mit dem gesuchten Namen verglichen. Dabei gibt es drei Möglichkeiten:

1. Die Namen sind gleich, dann ist das Objekt gefunden und der Zeiger auf das Objekt kann zurückgegeben werden.
2. Der gesuchte Name ist alphabetisch kleiner als der Name des betrachteten Objekts. Dann kann der Name in der restlichen Liste nicht mehr vorkommen, da ja nur noch größere Elemente folgen. Die Suche muss erfolglos abgebrochen werden.
3. Der gesuchte Name ist alphabetisch größer als der Name des betrachteten Objekts. Dann muss noch weitergesucht werden.



## Der Container als Liste – Einfügen eines Elements

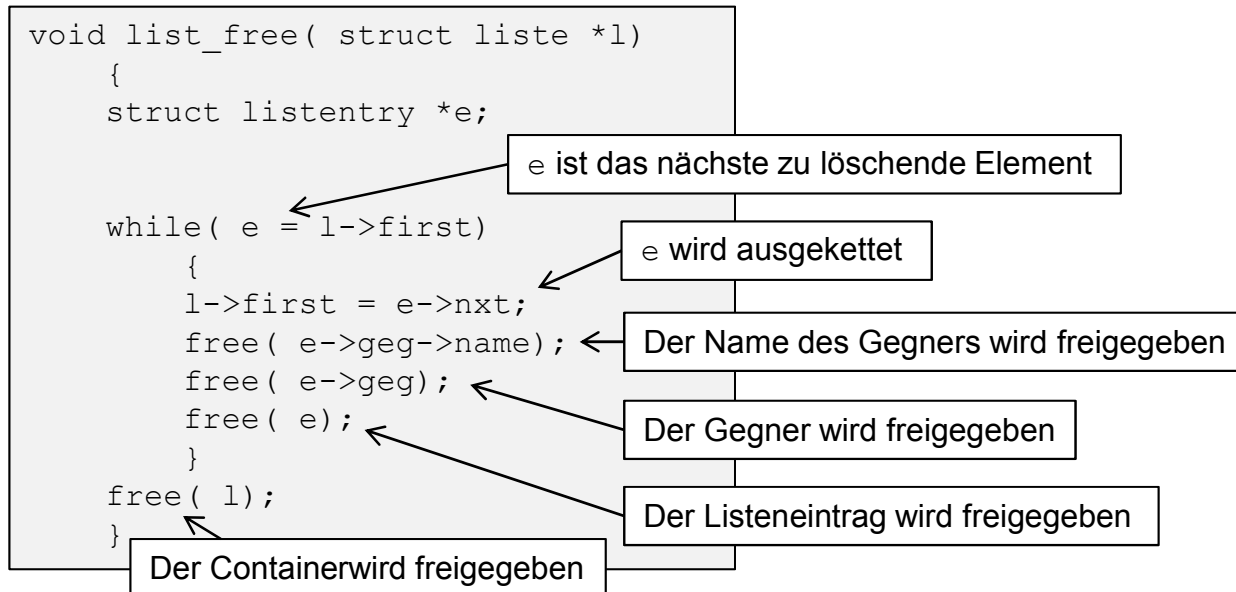
Soll ein Element eingefügt werden, muss zunächst die Einfügeposition gesucht werden (vgl. vorherige Folie). Gibt es schon ein Element gleichen Namens, kann es nicht eingefügt werden. Wenn das Element eingefügt werden kann, wird der Speicher für einen weiteren Listeneintrag (`struct listentry`) allokiert und die erforderlichen Verkettungen werden hergestellt:



Zum Einsetzen gehen wir mit doppelter Indirektion über die Adressen der Verkettungsfelder, wie wir es schon im letzten Abschnitt (Datenstrukturen, Folie 40) gemacht haben.

## Der Container als Liste – Freigabe des Containers

In dieser Funktion wird der Container einschließlich seines Inhalts freigegeben:



Beachten Sie dass bei `while( e = l->first)` eine Zuweisung an `e` erfolgt. Sollte dabei der Null-Zeiger zugewiesen worden sein, wird die Schleife abgebrochen.

## Der Container als Liste – Laden der Daten in den Container

Hier interessiert uns nur der Aufbau des Containers mit den Funktionen `list_create` und `list_insert`. Das Öffnen der Datei und das Einlesen der Daten aus der Datei kennen Sie bereits (vgl. Datenstrukturen, Folien 23 und 24):

```
struct liste *list_load( char *datei)
{
    FILE *pf;
    char land[100];
    struct liste *l;
    struct gegner *g;

    pf = fopen( datei, "r");

    l = list_create(); ← Erzeugen eines leeren Containers

    for( ; ; )
    {
        fscanf( pf, "%s", land);
        if( feof( pf))
            break;

        g = (struct gegner *)malloc( sizeof( struct gegner));
        g->name = (char *)malloc( strlen( land)+1);
        strcpy( g->name, land);
        fscanf( pf, "%d", &g->spiele);

        list_insert( l, g); ← Einfügen eines neuen Elements in
                               den Container
    }
    fclose( pf);
    return l; ← Rückgabe des Containers
}
```

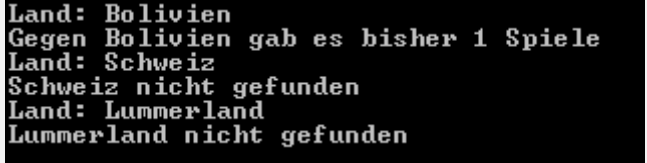
## Der Container als Liste – Anwendungsprogramm

Im Hauptprogramm kann der Container jetzt verwendet werden:

```
void main()
{
    struct liste *l;
    char land[100];
    struct gegner *g;
    int i;

    l = list_load( "Laenderspiele.txt");

    for( i = 0; i < 3; i++)
    {
        printf( "Land: ");
        scanf( "%s", land);
        g = list_find( l, land);
        if( g)
            printf( "Gegen %s gab es bisher %d Spiele\n", g->name, g->spiele);
        else
            printf( "%s nicht gefunden\n", land);
    }
    list_free( l);
}
```



Land: Bolivien  
Gegen Bolivien gab es bisher 1 Spiele  
Land: Schweiz  
Schweiz nicht gefunden  
Land: Lummerland  
Lummerland nicht gefunden

Laden der Daten

Suchen eines Datensatzes

Freigabe des Containers

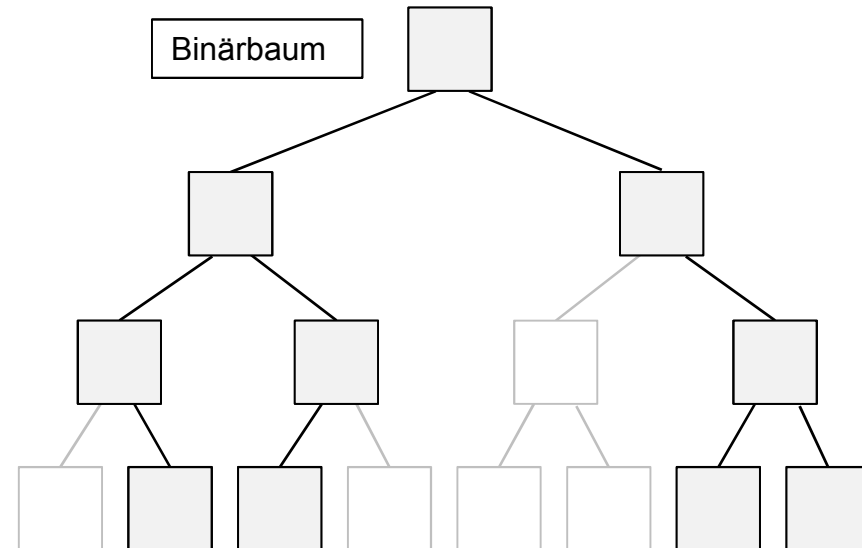
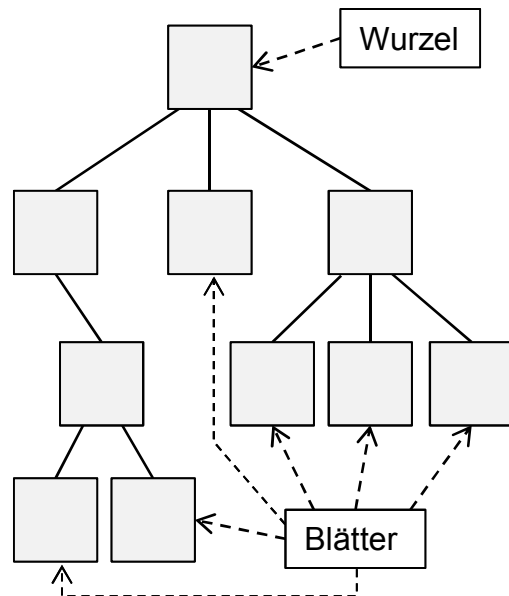
Liste für 50 zufällig gewählte Gegner

Maximale Suchtiefe: 50

Mittlere Suchtiefe: 25.5

+--Wales  
+--V-A-Emirate  
+--Ukraine  
+--USA  
+--Tunesien  
+--Tuerkei  
+--Thailand  
+--Suedkorea  
+--Serbien-Montenegro  
+--San-Marino  
+--Russland  
+--Rumaenien  
+--Portugal  
+--Polen  
+--Peru  
+--Paraguay  
+--Oman  
+--Oesterreich  
+--Norwegen  
+--Nordirland  
+--Niederlande  
+--Neuseeland  
+--Mexiko  
+--Marokko  
+--Luxemburg  
+--Kolumbien  
+--Kasachstan  
+--Jugoslawien  
+--Japan  
+--Italien  
+--Israel  
+--Island  
+--Irland  
+--GUS  
+--Frankreich  
+--Finnland  
+--Faeroeer  
+--Estland  
+--Ecuador  
+--Daenemark  
+--China  
+--Chile  
+--Bulgarien  
+--Bosnien-Herzegowina  
+--Bolivien  
+--Belgien  
+--Argentinien  
+--Algerien  
+--Albanien  
Aegypten

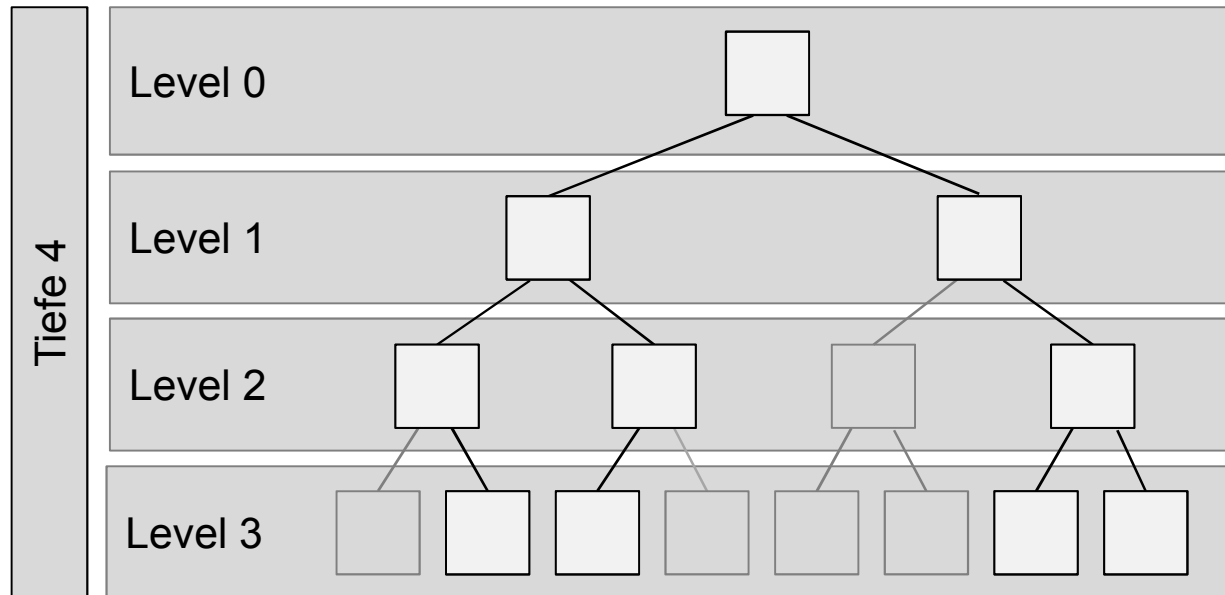
Ein **Baum** verallgemeinert den Begriff der Liste dahingehend, dass jedes Element eine endliche Folge von **Nachfolgern** haben kann. Wir sprechen dann vom 1., 2., 3. Nachfolger usw. Die Elemente im Baum bezeichnen wir als **Knoten**. Einen Baum zeichnen wir in der folgenden Weise:



In einem Baum gibt es genau einen Knoten, der keinen Vorgänger hat. Diesen bezeichnen wir als den **Wurzelknoten** oder die **Wurzel**. Alle Knoten sind von der Wurzel aus durch endlich viele Nachfolgeroperationen auf genau einem Weg erreichbar (es gibt keine Schleifen oder Zyklen). Knoten, die keine Nachfolger haben, bezeichnen wir als **Blätter**.

**Binärbaume** sind Bäume, bei denen ein Knoten maximal zwei Nachfolger hat. Bei einem Binärbaum sprechen wir dann vom linken und vom rechten Nachfolger, obwohl die Begriffe "links" und "rechts" softwaretechnisch keinen Sinn haben. **Wenn wir in diesem Kapitel von einem Baum sprechen, meinen wir immer einen Binärbaum.**

Durch die Anzahl von Nachfolgeroperationen, die man benötigt, um von der Wurzel aus einen bestimmten Knoten zu erreichen, sind in einem Baum **Levels** definiert. Jeder Knoten ist genau einem Level zugeordnet. Das maximale Level eines Baumes + 1 bezeichnen wir als die Tiefe des Baums.



Auf dem Level  $i$  eines Binärbaums sind maximal  $2^i$  Knoten.

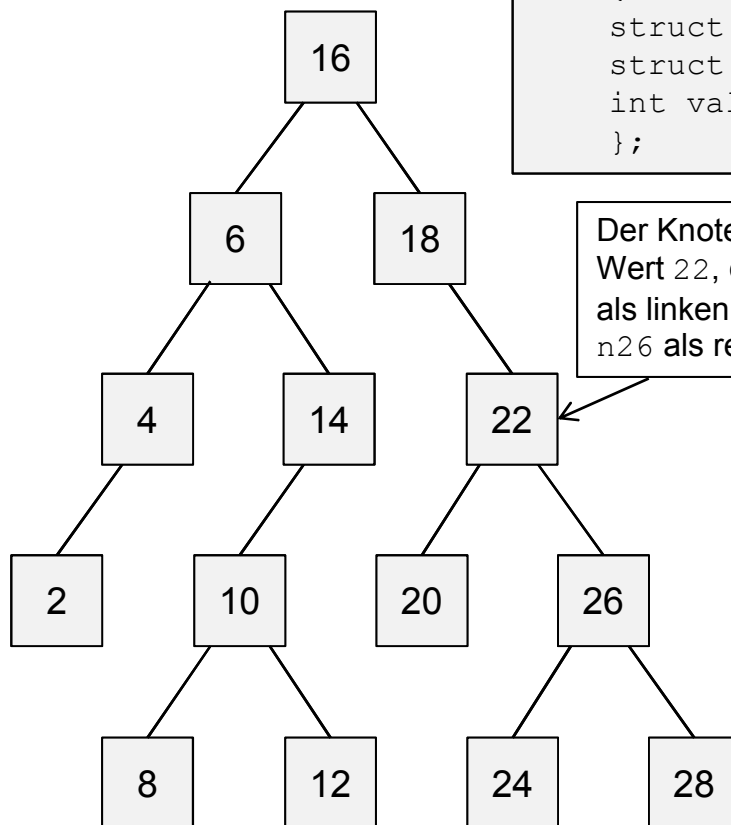
Ein Binärbaum der Tiefe  $t$  hat maximal  $2^0 + 2^1 + 2^2 + \dots + 2^{t-1} = \frac{2^t - 1}{2 - 1} = 2^t - 1$  Knoten.

Ein Binärbaum mit  $n$  Knoten hat mindestens die Tiefe  $\log_2(n)$ .



## Ein einfacher Baum

Als Beispiel legen wir einen Baum mit 14 Knoten an. Jeder Knoten (`struct node`) hat neben seinen Nutzdaten (hier nur ein Zahlenwert, `value`) Zeiger auf einen möglichen linken oder rechten Nachfolger (`left`, `right`). Diese Zeiger haben den Wert 0, wenn der Nachfolger nicht existiert. Ausgehend von dieser einfachen Knotenstruktur werden die Knoten statisch angelegt und miteinander verkettet.



```
struct node
{
    struct node *left;
    struct node *right;
    int value;
};
```

Der Knoten `n22` hat den Wert 22, den Knoten `n20` als linken und den Knoten `n26` als rechten Nachfolger.

```
/* Knoten des Levels 4 */
struct node n08 = { 0, 0, 8};
struct node n12 = { 0, 0, 12};
struct node n24 = { 0, 0, 24};
struct node n28 = { 0, 0, 28};

/* Knoten des Levels 3 */
struct node n02 = { 0, 0, 2};
struct node n10 = { &n08, &n12, 10};
struct node n20 = { 0, 0, 20};
struct node n26 = { &n24, &n28, 26};

/* Knoten des Levels 2 */
struct node n04 = { &n02, 0, 4};
struct node n14 = { &n10, 0, 14};
struct node n22 = { &n20, &n26, 22};

/* Knoten des Levels 1 */
struct node n06 = { &n04, &n14, 6};
struct node n18 = { 0, &n22, 18};

/* Wurzel */
struct node n16 = { &n06, &n18, 16};
```

Dieser Baum wird als ein Beispiel für weitere Überlegungen dieses Abschnitts dienen.

## Traversierung von Bäumen

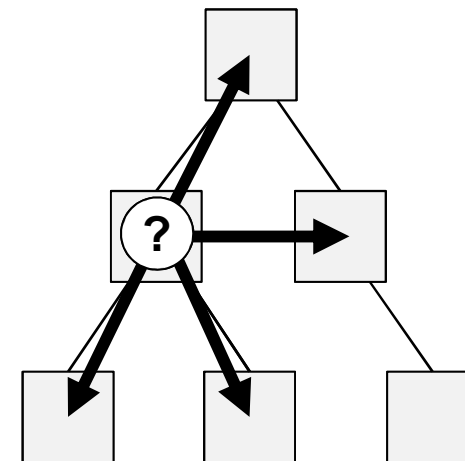
Bei Listen gab es eine natürliche Weise vorwärts oder rückwärts durch alle Elemente zu iterieren. Bei Bäumen gibt es keine natürliche Besuchsreihenfolge. Man kann unterschiedliche Besuchsstrategien entwickeln:

Unter der **Traversierung** eines Baumes verstehen wir das systematische Aufsuchen aller Knoten des Baumes, um an den Knoten gewisse Operationen durchführen zu können.

Zum Beispiel kann man "Kinder" vor "Geschwistern" (Tiefensuche, depth-first) oder "Geschwister" vor "Kindern" (Breitensuche, breadth-first) besuchen.

Wir werden die folgenden Traversierungsstrategien behandeln:

- Preorder-Traversierung
- Inorder-Traversierung
- Postorder-Traversierung
- Levelorder-Traversierung



## Preorder-Traversierung

Wir starten an einem Knoten bearbeiten den Knoten gehen danach zum linken Nachfolger und fahren dort rekursiv mit der Bearbeitung fort. Wenn wir vom linken Knoten und allen darunter liegenden Knoten zurückkommen, starten wir rekursiv mit der Bearbeitung des rechten Nachfolgers:

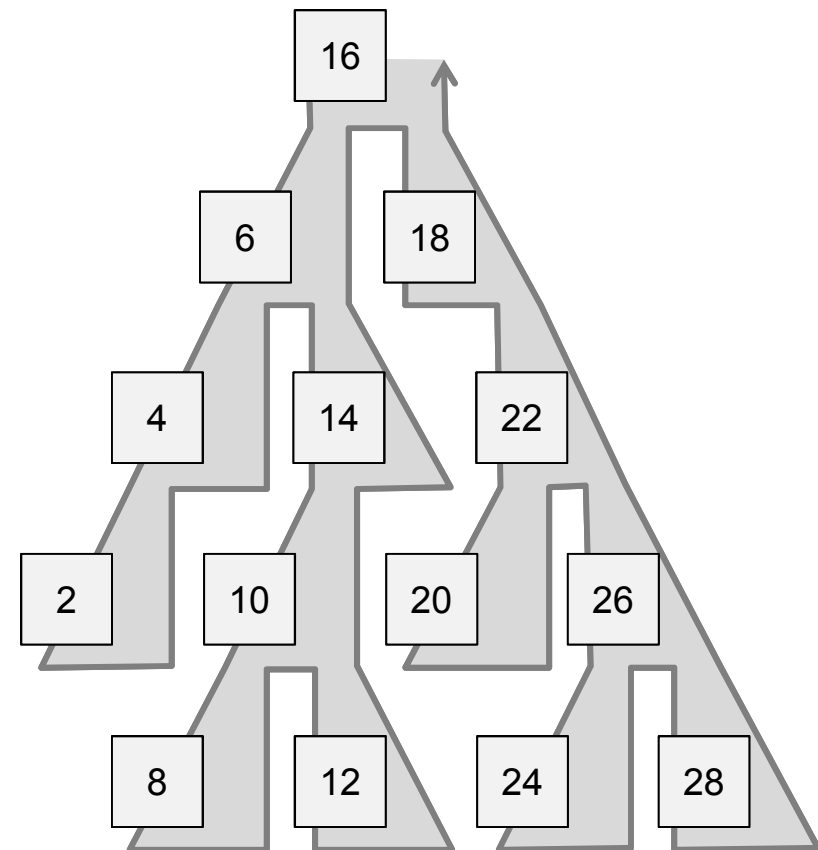
```
void preorder( struct node *n)
{
    if( n)
    {
        printf( " %2d", n->value);
        preorder( n->left);
        preorder( n->right);
    }
}
```

### Aufgerufen an der Wurzel

```
void main()
{
    printf( "Preorder:\n");
    preorder( &n16);
}
```

ergibt sich die folgende Besuchsreihenfolge:

```
Preorder:
16 6 4 2 14 10 8 12 18 22 20 26 24 28
```



## Inorder-Traversierung

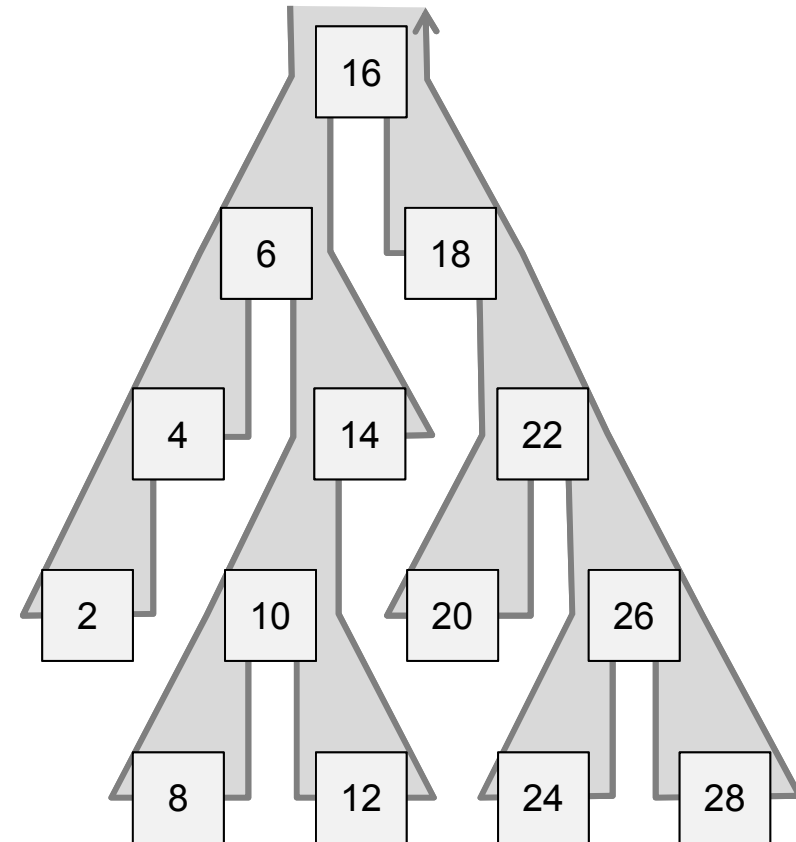
Bei der Inorder-Traversierung tauschen wir gegenüber der Preorder-Traversierung nur zwei Zeilen im Quellcode:

```
void inorder( struct node *n)
{
    if( n)
    {
        inorder( n->left);
        printf( " %2d", n->value);
        inorder( n->right);
    }
}
```

Wir tauchen in die Behandlung des linken Teilbaums eines Knotens ab, bevor wir den Knoten selbst behandeln. Dadurch ergibt sich die folgende Besuchsreihenfolge:

**Inorder:**  
2 4 6 8 10 12 14 16 18 20 22 24 26 28

```
void main()
{
    printf( "Inorder:\n");
    inorder( &n16);
}
```



Bei einem aufsteigend sortierten Baum (siehe spätere Folie) entspricht die Besuchsreihenfolge der Knotenordnung.

## Postorder-Traversierung

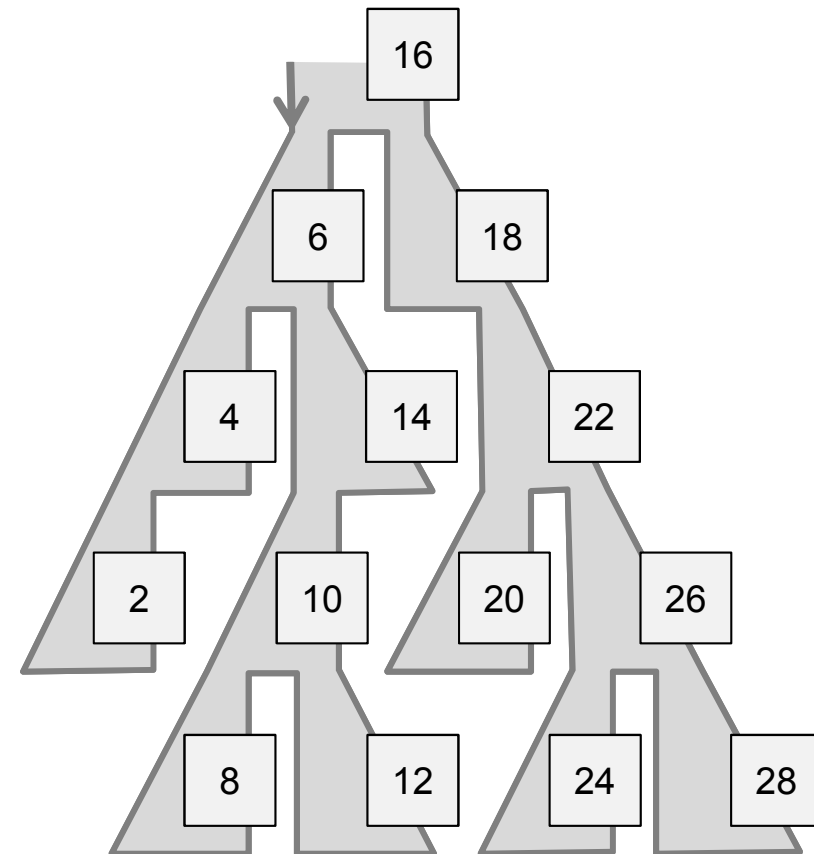
Bei der Postorder-Traversierung findet die Behandlung eines Knotens statt, nachdem beide am Knoten anhängenden Teilbäume bearbeitet wurden:

```
void postorder( struct node *n)
{
    if( n)
    {
        postorder( n->left);
        postorder( n->right);
        printf( " %2d", n->value);
    }
}
```

Dementsprechend ergibt sich eine andere  
Besuchsreihenfolge:

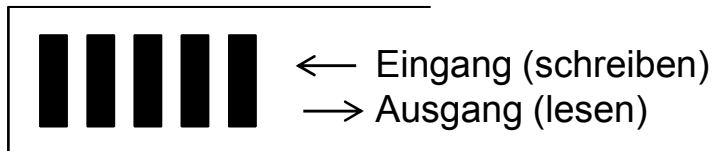
```
void main()
{
    printf( "Postorder:\n");
    postorder( &n16);
}
```

**Postorder:**  
2 4 8 12 10 14 6 20 24 28 26 22 18 16

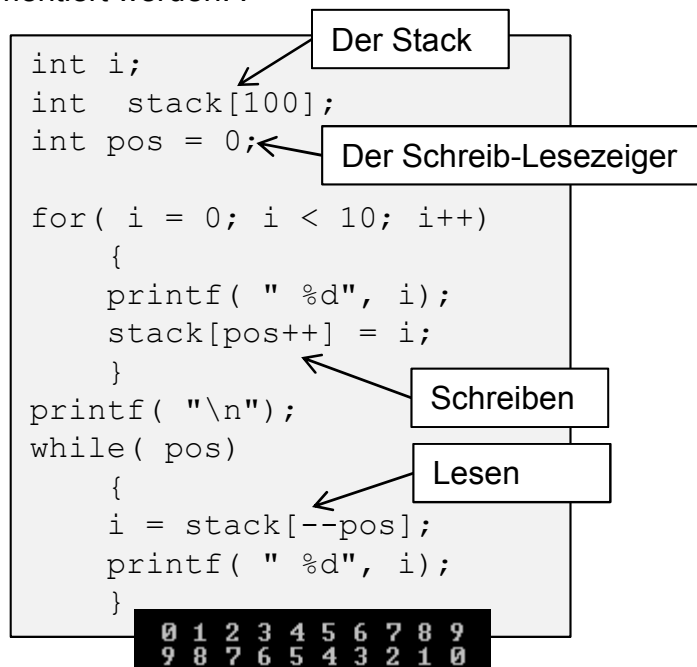


## Exkurs über Stacks und Queues

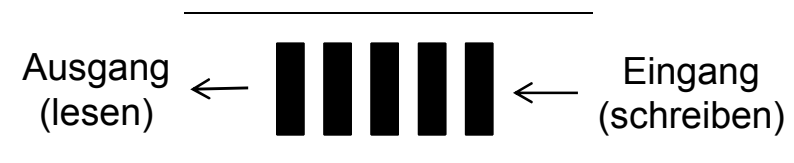
Stacks sind unfaire Warteschlangen, weil der, der zuletzt kommt, zuerst bedient wird (LIFO, Last In First Out). Bei einem **Stack** wird am gleichen Ende geschrieben und gelesen



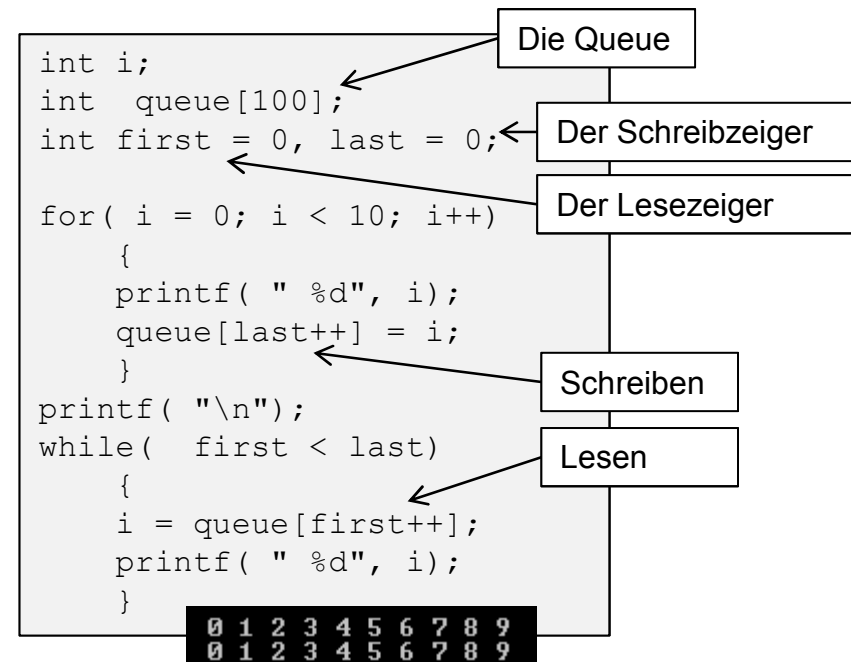
Ein Stack kann als Array mit einem Schreib-Lesezeiger implementiert werden: .



Queues sind faire Warteschlangen, weil der, der zuerst kommt, zuerst bedient wird (FIFO, First In First Out). Bei einer **Queue** wird an verschiedenen Enden geschrieben und gelesen.



Eine Queue kann als Array mit einem Schreib- und einem Lesezeiger implementiert werden:



Die Implementierungen sind einfach, aber problematisch, der Stack kann überlaufen, die Queue kann aus dem Array "herauslaufen"


## Rekursionsfreie Preorder-Traversierung

Stacks sind ein Hilfsmittel, um Rekursion zu vermeiden. Mit Blick auf die Levelorder-Traversierung erstellen wir zunächst eine rekursionsfrei Variante der Preorder-Traversierung:

Statt in die Rekursion zu gehen, legen wir die anstehenden Knoten auf den Stack, um sie in nachfolgenden Schleifenläufen wieder vom Stack zu holen und zu bearbeiten.

Die Reihenfolge, in der wir die Knoten (`left`, `right`) auf den Stack legen, unterscheidet sich von der Reihenfolge der rekursiven Aufrufe, weil der Stack die Reihenfolge dreht.

```
void preorder( struct node *n)
{
    if( n)
    {
        printf( " %2d", n->value);
        preorder( n->left);
        preorder( n->right);
    }
}
```



```
void preorder( struct node *n)
{
    struct node *stack[100];
    int pos = 0;

    stack[pos++] = n;
    while( pos)
    {
        n = stack[--pos];
        if( n)
        {
            printf( " %2d", n->value);
            stack[pos++] = n->right;
            stack[pos++] = n->left;
        }
    }
}
```

## Levelorder-Traversierung

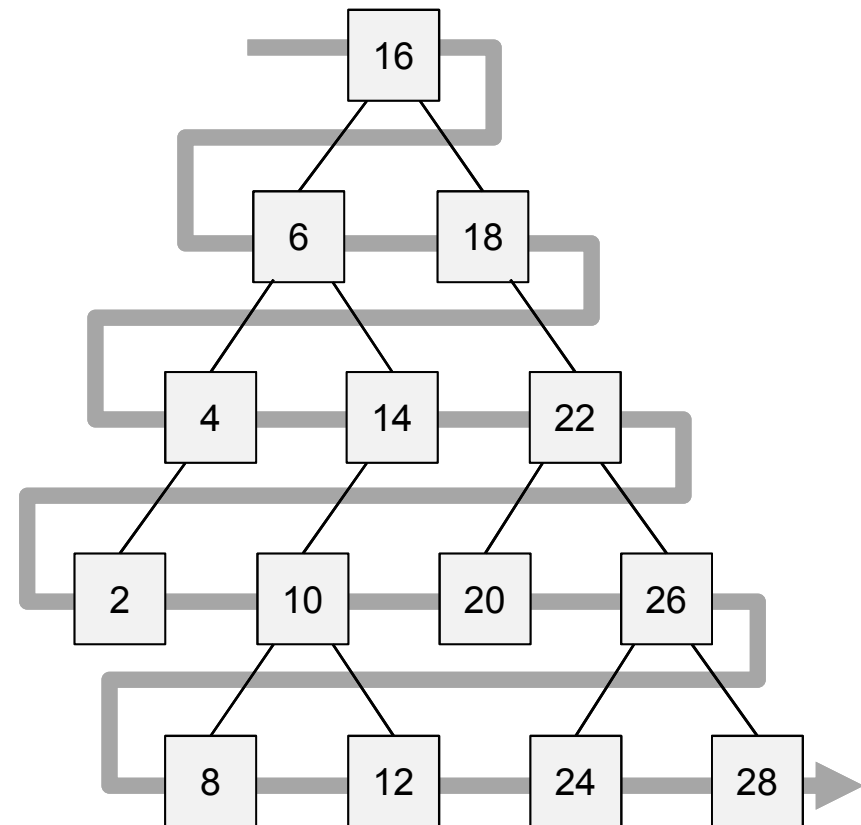
Wenn wir den Stack in der rekursionsfreien Preorder-Traversierung durch eine Queue ersetzen, erhalten wir die sogenannte **Levelorder-Traversierung**. Die Nachfolgeknoten eines Knotens werden jetzt in die Queue gelegt und von dort in der Reihenfolge ihres Speicherns wieder abgerufen. Dadurch wird der Baum Level für Level durchlaufen.

```
void levelorder( struct node *n)
{
    struct node *queue[100];
    int first = 0, last = 0;

    queue[last++] = n;
    while( first < last)
    {
        n = queue[first++];
        if( n)
        {
            printf( " %2d", n->value);
            queue[last++] = n->left;
            queue[last++] = n->right;
        }
    }
}
```

```
void main()
{
    printf( "Levelorder:\n");
    levelorder( &n16);
}
```

```
Levelorder:
16  6 18  4 14 22  2 10 20 26  8 12 24 28
```





## Aufsteigend sortierte Bäume

Wir betrachten Bäume, deren Knoten der Größe nach verglichen werden können, wobei mit "Größe" nicht notwendig eine numerische Größe gemeint ist. Es können zum Beispiel auch den Knoten zugeordnete Namen bezüglich ihrer lexikographischen Ordnung verglichen werden.

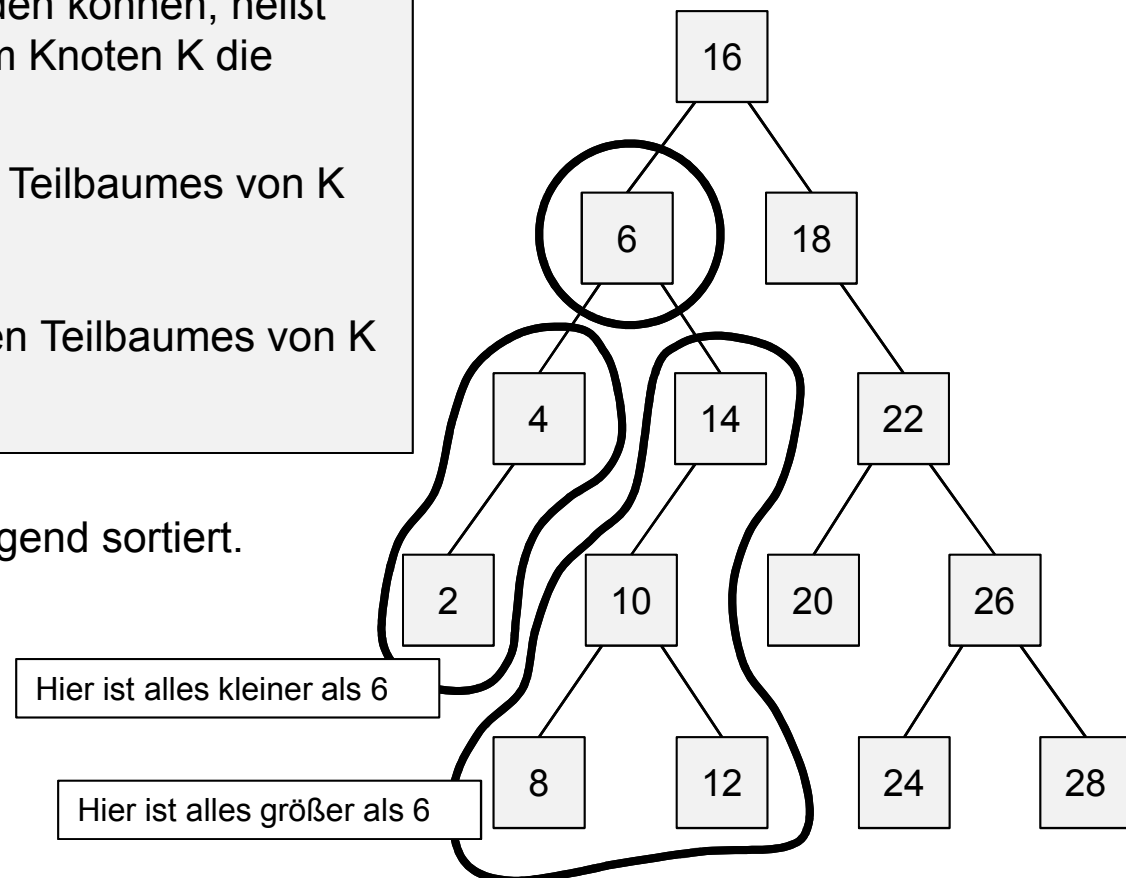
Ein Binärbaum, bei dem die Knoten mittels einer Ordnungsrelation ( $<$ ) verglichen werden können, heißt **aufsteigend sortiert**, wenn an jedem Knoten  $K$  die Bedingungen

$X < K$  für alle Knoten  $X$  des linken Teilbaumes von  $K$   
und

$X > K$  für alle Knoten  $X$  des rechten Teilbaumes von  $K$   
gelten.

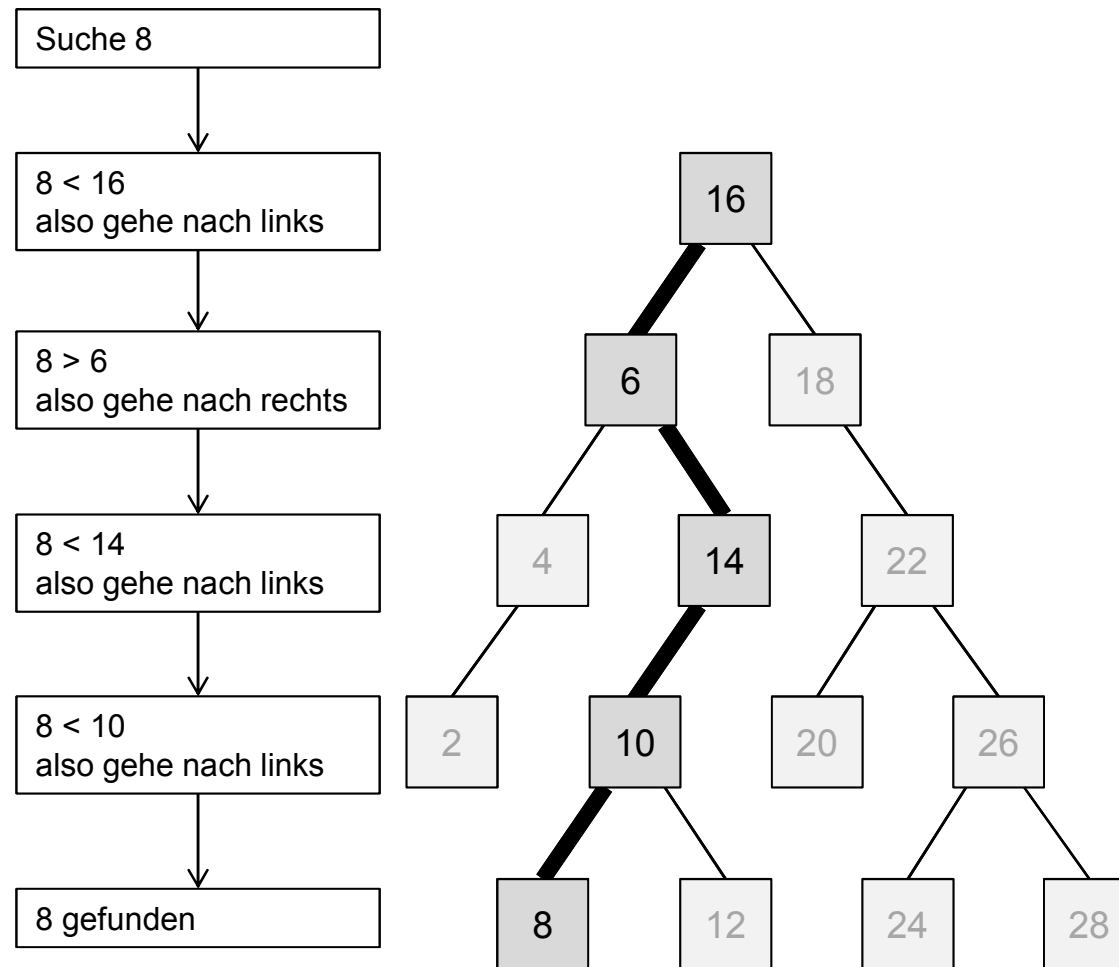
Der nebenstehende Baum ist aufsteigend sortiert.

Durch Vertauschen von »links« und »rechts« in obiger Definition erhält man den Begriff des absteigend sortierten Baumes. Wenn wir von sortierten Bäumen reden, meinen wir immer aufsteigend sortierte Bäume



## Suche in aufsteigend sortierten Bäumen

In nicht sortierten Bäumen kann man Elemente nur durch Traversierung finden. In sortierten Bäumen gibt es effizientere Suchstrategien:



In einer Schleife wird gezielt nach links bzw. rechts im Baum abgestiegen, bis das gesuchte Element gefunden oder das Ende des Baums erreicht wurde.

## Implementierung der Suche im aufsteigend sortierten Baum

```
void find( struct node *n, int v)
```

```
{
    while( n)
    {
        if( n->value == v)
        {
            printf( " %d gefunden\n", v);
            return;
        }
        if( v < n->value)
        {
            printf( " %2d->li", n->value);
            n = n->left;
        }
        else
        {
            printf( " %2d->re", n->value);
            n = n->right;
        }
    }
    printf( " %d nicht gefunden\n", v);
}
```

Element gefunden

Gesuchtes Element  
ist kleiner, Abstieg  
nach links

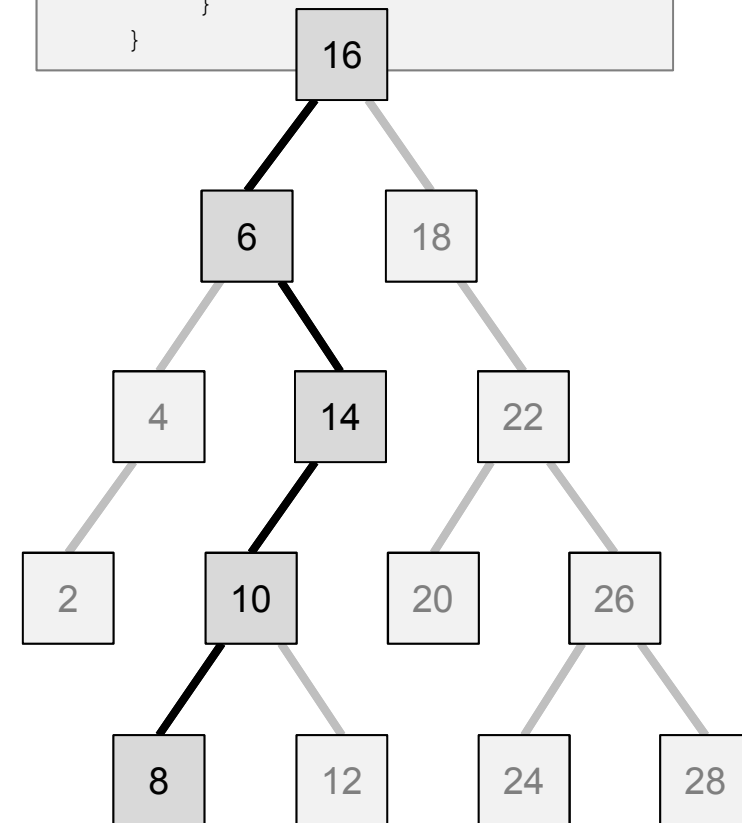
Gesuchtes Element  
ist größer, Abstieg  
nach rechts

Element  
nicht  
gefunden

```
Suche 1 16->li 6->li 4->li 2->li 1 nicht gefunden
Suche 2 16->li 6->li 4->li 2 gefunden
Suche 3 16->li 6->li 4->li 2->re 3 nicht gefunden
Suche 4 16->li 6->li 4 gefunden
Suche 5 16->li 6->li 4->re 5 nicht gefunden
Suche 6 16->li 6 gefunden
Suche 7 16->li 6->re 14->li 10->li 8->li 7 nicht gefunden
Suche 8 16->li 6->re 14->li 10->li 8 gefunden
Suche 9 16->li 6->re 14->li 10->li 8->re 9 nicht gefunden
Suche 10 16->li 6->re 14->li 10 gefunden
```

```
void main()
{
    int i;

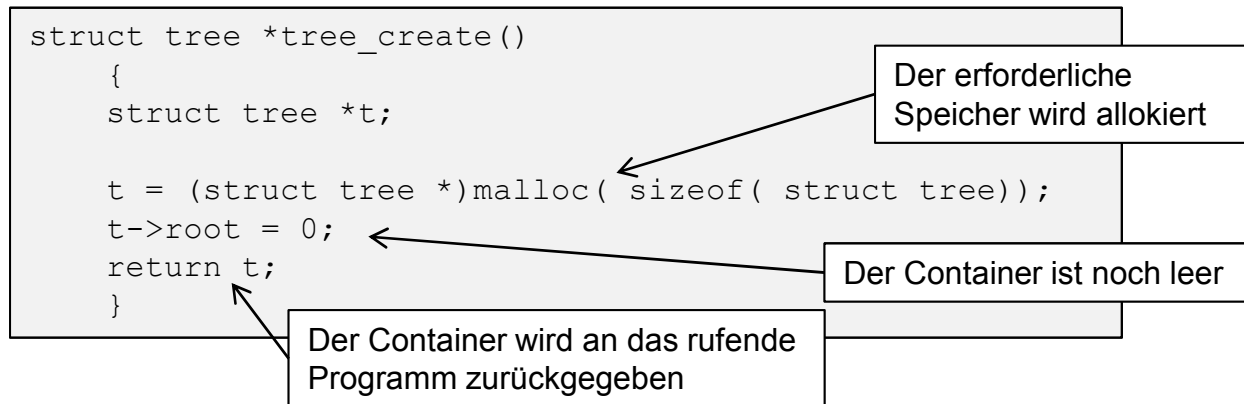
    for( i = 1; i <= 10; i++)
    {
        printf( "Suche %2d ", i);
        find( &n16, i);
    }
}
```





## Der Container als Baum – Erzeugen eines leeren Containers

Alle Datenstrukturen im Container werden dynamisch erzeugt. Ein leerer Container besteht aus einem Header (`struct tree`) dessen `root`-Zeiger den Wert 0 hat, da es noch keine Knoten im Baum gibt:



Die erforderliche Struktur wird allokiert, initialisiert und an das rufende Programm zurückgegeben.

Bei jedem Aufruf der `tree_create`-Funktion wird ein neuer Container erzeugt. Ein Anwendungsprogramm kann daher mehrere Container erzeugen und unabhängig voneinander verwenden:

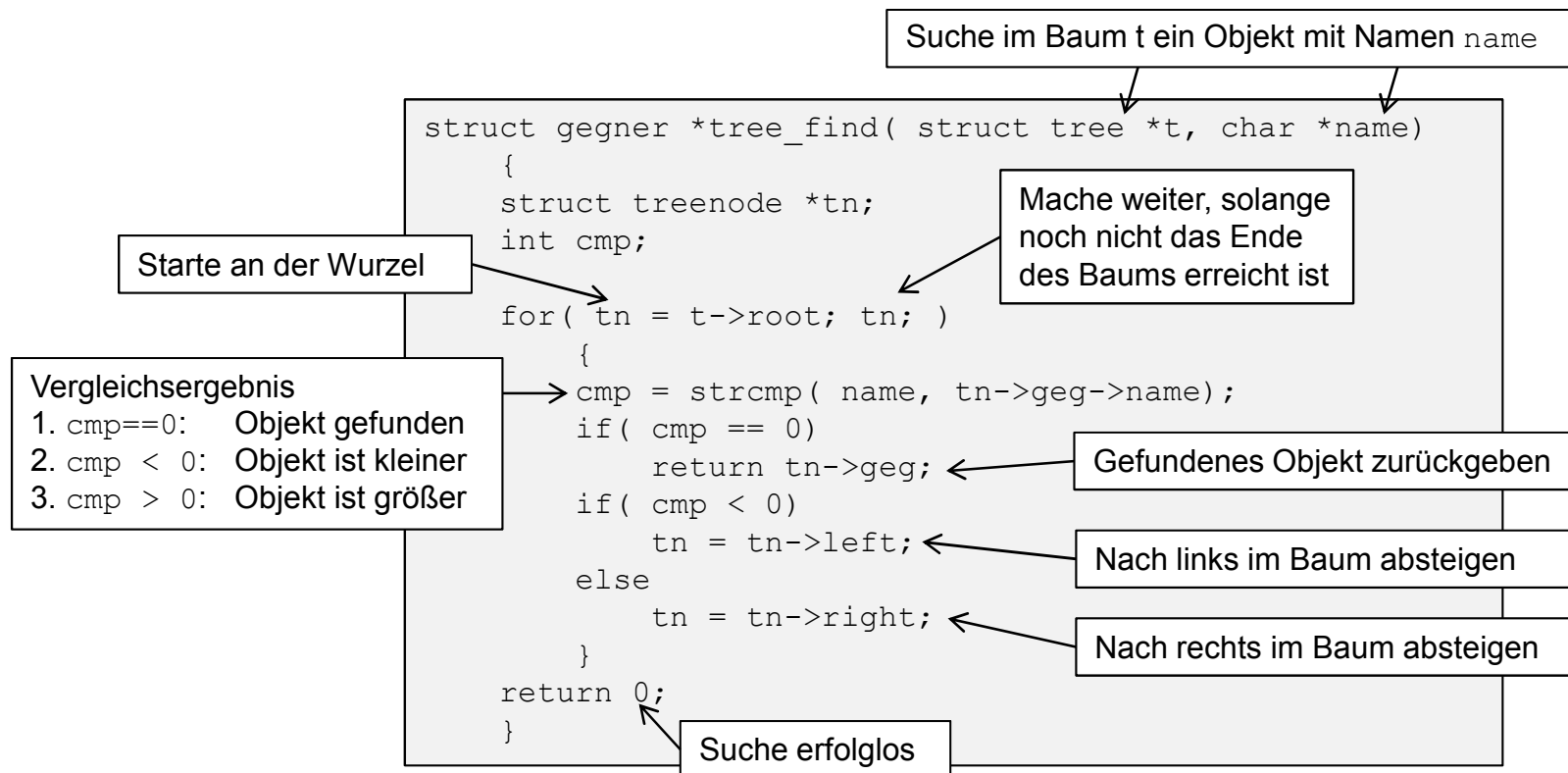
```
struct tree *container1;
struct tree *container2;

container1 = tree_create();
container2 = tree_create();
```

## Der Container als Baum – Finden eines Elements

Von der Wurzel startend wird im Baum nach links oder nach rechts abgestiegen, je nachdem, ob das gesuchte Objekt alphabetisch vor oder hinter dem betrachteten Knoten liegen müsste.

Entweder wird das Objekt auf diese Weise gefunden oder man erreicht das Ende des Baums:



## Der Container als Baum – Einfügen eines Elements

Soll ein Element eingefügt werden, muss zunächst die Einfügeposition gesucht werden (vgl. vorherige Folie). Gibt es schon ein Element gleichen Namens, kann es nicht eingefügt werden. Wenn das Element eingefügt werden kann, wird der Speicher für einen weiteren Knoten (`struct treenode`) allokiert und die erforderlichen Verkettungen werden hergestellt:

```
int tree_insert( struct tree *t, struct gegner *g)
{
    struct treenode **node, *neu;
    int cmp;

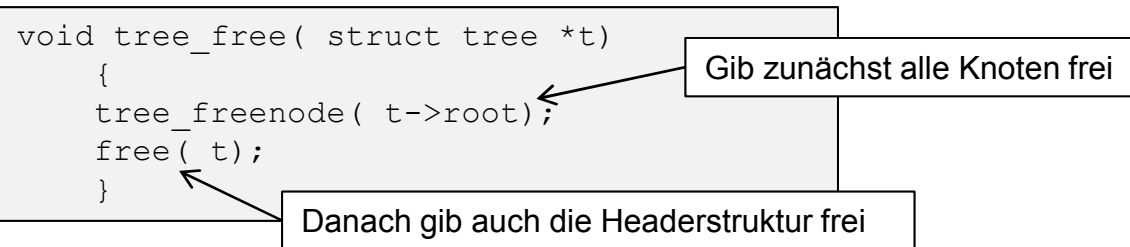
    for( node = &(t->root); *node; )
    {
        cmp = strcmp( g->name, (*node)->geg->name);
        if( !cmp)
            return 0;
        if( cmp < 0)
            node = &((*node)->left);
        else
            node = &((*node)->right);
    }
    neu = (struct treenode *)malloc( sizeof( struct treenode));
    neu->left = 0;
    neu->right = 0;
    neu->geg = g;
    *node = neu;
    return 1;
}
```

Diese Funktion  
sollten Sie  
inzwischen ohne  
weiteren Kommentar  
verstehen können

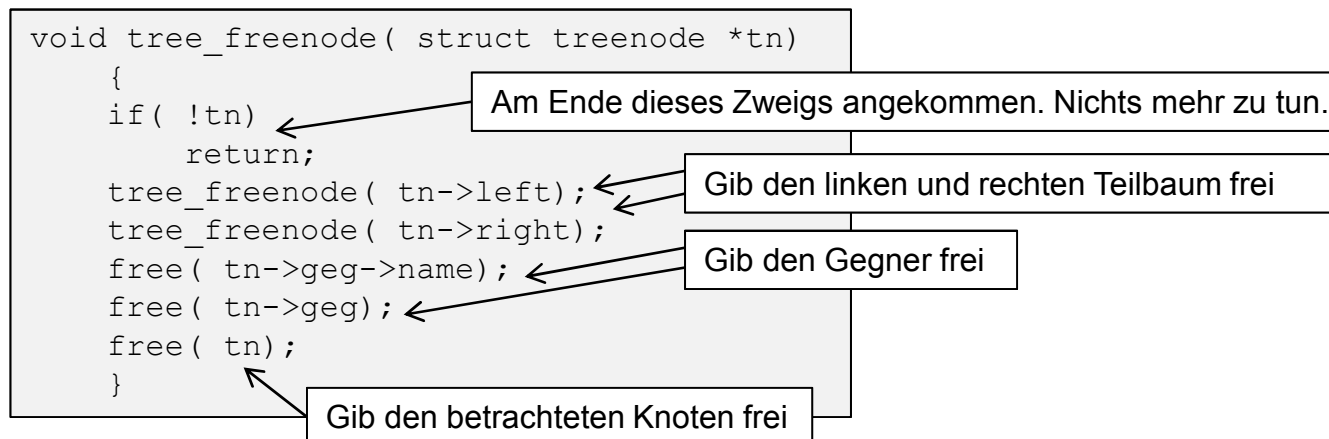
Zum Einsetzen gehen wir mit doppelter Indirektion über die Adressen der Verkettungsfelder.

## Der Container als Baum – Freigabe des Containers

In dieser Funktion wird der Container einschließlich seines Inhalts freigegeben.



Die Funktion zur Freigabe der Knoten arbeitet rekursiv, um zunächst die an einem Knoten anhängenden linken und rechten Teilbäume freizugeben, bevor der Knoten selbst inclusive des referenzierten Gegners freigegeben wird:





## Der Container als Baum – Laden der Daten in den Container

Das Laden Daten in den Container durch die Funktion `tree_load` unterscheidet sich nicht von der entsprechenden Funktion (`list_load`, Folie 11) für Listen. Der einzige Unterschied ist, dass jetzt die Funktionen `tree_create` und `tree_insert` anstelle von `list_create` und `list_insert` verwendet werden.

Man hätte sogar eine abstraktere, für beide Containertypen identische Schnittstelle verwenden können, sodass der Anwender gar nicht hätte erkennen können, welche Datenstruktur (Liste oder Baum) der Implementierung des Containers zugrunde liegt.

Einen Datentyp, der seine Implementierung vor dem Anwender verbirgt und nur durch eine Funktionsschnittstelle bedient wird, nennt man **Abstrakter Datentyp**.

Da abstrakte Datentypen ein wichtiger Zwischenschritt zur objektorientierten Programmierung sind, werde ich in einem eigenen Kapitel noch einmal auf dieses Programmierkonzept eingehen.

## Der Container als Baum – Anwendungsprogramm

Auch das Anwendungsprogramm ist bis auf Funktionsnamen identisch mit dem entsprechenden Programm für Listen (siehe Folie 12) und muss hier nicht noch einmal eigens gezeigt werden.

Viel interessanter ist die Frage nach der Suchtiefe in einem Baum im Vergleich zu einer Liste.

Bei der Liste hatten wir die folgenden Ergebnisse erhalten:

Liste für 50 zufällig gewählte Gegner Maximale Suchtiefe: 50 Mittlere Suchtiefe: 25.5
---

Bei einem Baum erwarten wir kürzere Suchwege, was sich dann auch in schnelleren Suchalgorithmen niederschlagen dürfte. Tiefenmessungen an einem Beispiel zeigen dies:

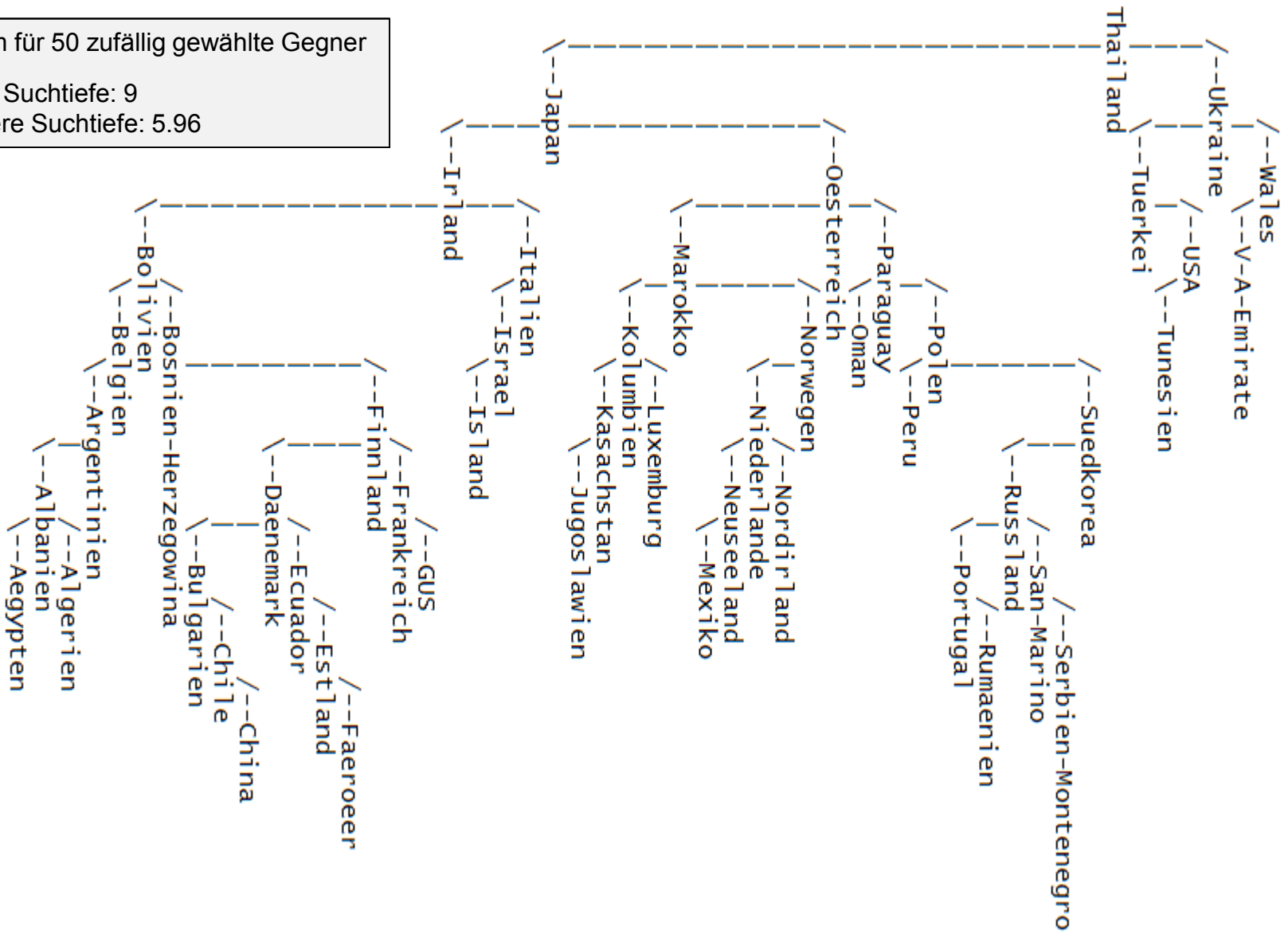
Baum für 50 zufällig gewählte Gegner Max. Suchtiefe: 9 Mittlere Suchtiefe: 5.96
---

Die folgende Folie zeigt das Ergebnis am Beispiel der Länderspieldatei, die ich, aus Gründen der Übersichtlichkeit auf 50 Einträge beschränkt habe.

Baum für 50 zufällig gewählte Gegner

Max. Suchtiefe: 9

Mittlere Suchtiefe: 5.96



## Ein vorläufiges Fazit

Der konkrete Testfall zeigt bei einem Baum deutlich bessere Werte als bei einer Liste.

Liste für 50 zufällig gewählte Gegner

Maximale Suchtiefe: 50

Mittlere Suchtiefe: 25.5

Baum für 50 zufällig gewählte Gegner

Max. Suchtiefe: 9

Mittlere Suchtiefe: 5.96

Mit einer maximalen Suchtiefe von 9 liegt der Baum nicht weit vom theoretischen Optimum für Binärbäume entfernt, das für 50 Elemente bei 6 ( $\log_2(50) = 5.64$ ) liegt.

Wenn wir uns aber vorstellen, dass die Daten in der Datei aufsteigend sortiert vorliegen und in einen Baum eingelesen werden, geht diese Qualität des Baums verloren, da in dieser Situation neue Elemente immer rechts angefügt werden. Der Baum entartet dann zu einer Liste (siehe nächste Folie).

Baum für 50 sortierte Gegner

Max. Suchtiefe: 50

Mittlere Suchtiefe: 25.5

Der Baum ist in dieser Situation sogar schlechter als eine Liste, da er für die gleiche Suchqualität mehr Speicher verbraucht und aufwendigere Algorithmen hat.

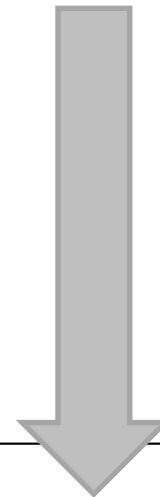
Mit der Frage, wie man die "Entartung" des Baums vermeiden kann, werden wir uns bei unserem nächsten Containertyp – dem Treap – beschäftigen.

Baum für 50 sortierte Gegner

Max. Suchtiefe: 50

Mittlere Suchtiefe: 25.5

Aegypten  
/--Albanien  
/--Algerien  
/--Argentinien  
/--Armenien  
/--Aserbaidschan  
/--Australien  
/--Belgien  
/--Bohmen-Maehren  
/--Bolivien  
/--Bosnien-H  
/--Bras  
/--



## Erinnerung – Heaps

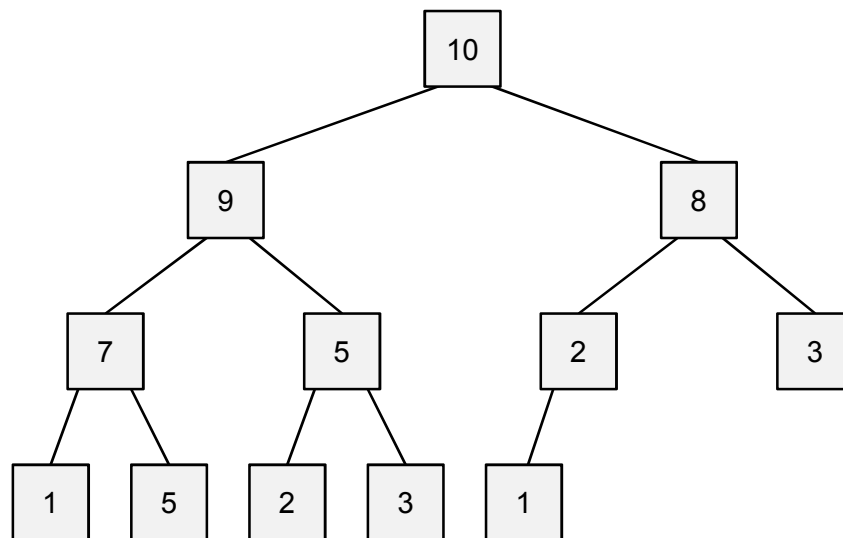
Heaps hatten wir bereits im Zusammenhang mit dem Sortierverfahren Heapsort kennengelernt. Heaps sind neben Stacks und Queues die wichtigsten Warteschlangenstrukturen. Heaps sind Prioritätswarteschlangen:

Stack: Wer zuletzt kommt, wird zuerst bedient

Queue: Wer zuerst kommt, wird zuerst bedient

Heap: Wer am wichtigsten ist, wird zuerst bedient

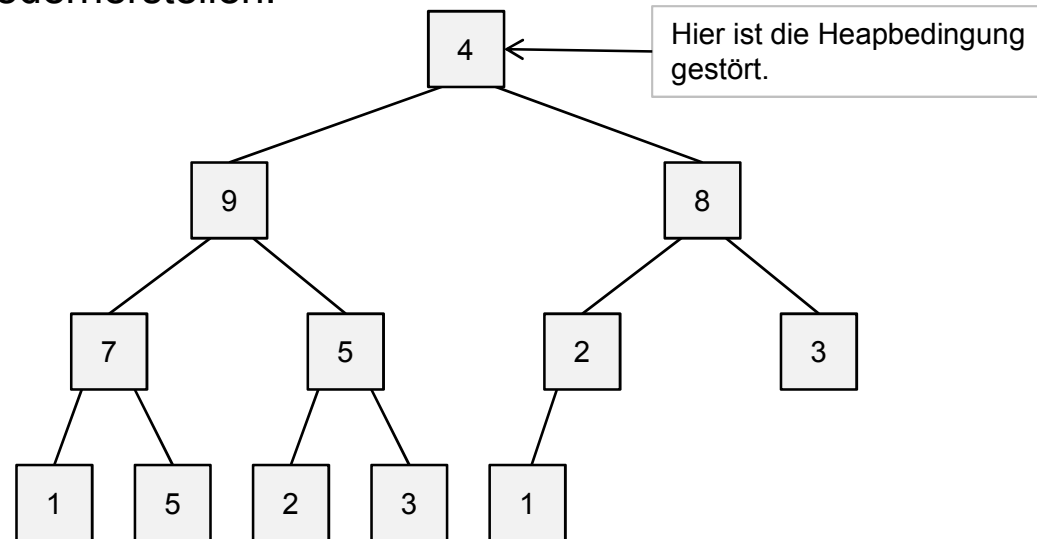
Ein **Heap** ist ein Baum, in dem jeder Knoten eine Priorität hat und jeder Knoten eine höhere Priorität hat, als seine Nachfolgerknoten.



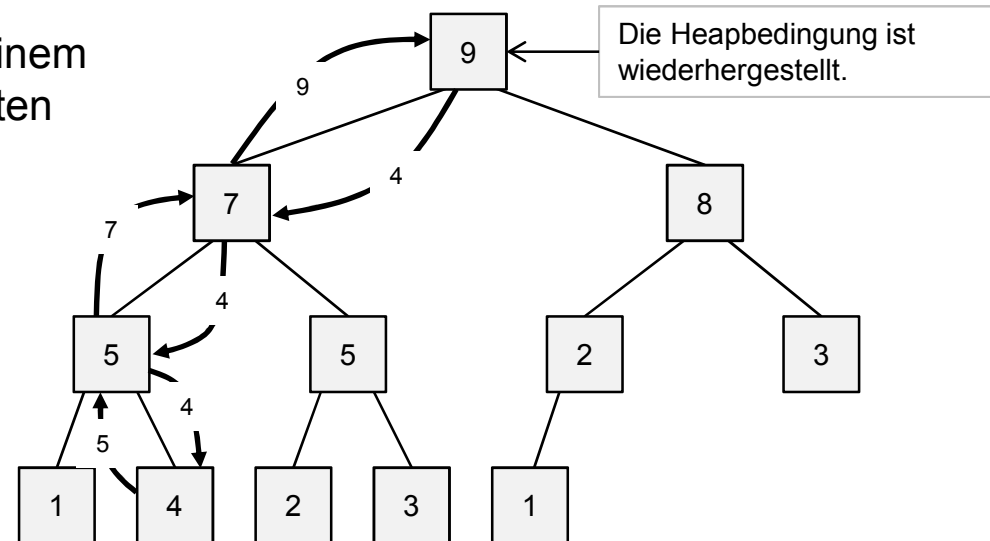
Bei einem Heap steht an der Wurzel des Baums (und an der Wurzel jeden Teilbaums) das Element mit der höchsten Priorität im Baum (Teilbaum).

## Erinnerung - Wiederherstellung der Heapbedingung

Wenn die Heapbedingung an einer (und nur einer) Stelle im Baum gestört ist, so kann man sie sehr einfach wiederherstellen.



Man tauscht den Störenfried so lange mit seinem größten Nachfolger, bis die Störung nach unten aus dem Baum herausgewachsen ist:



## **Der Heap als Prioritätswarteschlange**

Prioritätswarteschlangen spielen überall dort eine wichtige Rolle, wo Aufgaben prioritätsgesteuert abgearbeitet werden müssen. Man muss neue Elemente in eine Prioritätswarteschlange einfügen können und das Element mit der höchsten Priorität abrufen und entfernen können.

### **Entnehmen des Elements mit der höchsten Priorität:**

1. Entnimm das Element an der Wurzel.
2. Bring ein Blatt des Baums an die Wurzel.
3. Stelle die Heapbedingung wieder her

### **Einfügen eines neuen Elements**

1. Füge das Element als Blatt ein
2. Gehe von dem Element zurück zur Wurzel und führe dabei jeweils einen Reparaturschritt (Tausch mit größtem Nachfolger) durch.

Beide Operationen erzeugen, wenn sie auf einem intakten Heap ausgeführt werden, am Ende wieder einen intakten Heap. Die Laufzeitkomplexität ist bei beiden Operationen proportional zur Tiefe des Baumes.



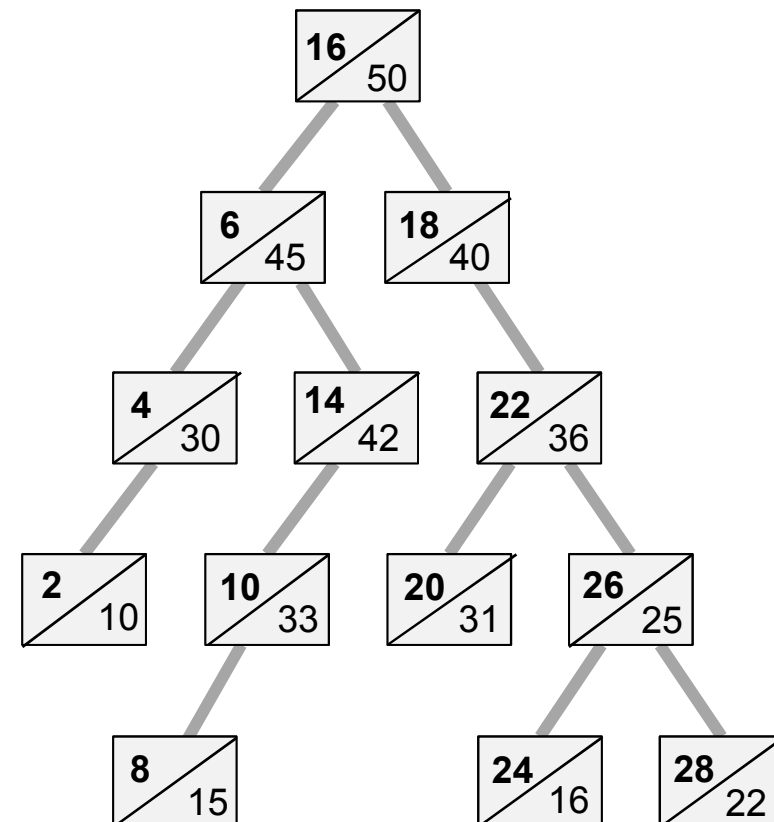
## Treaps

Zur Implementierung des nächsten Containers werden wir ein randomisiertes Verfahren betrachten. Darunter verstehen wir ein Verfahren, das sich in geschickter Weise den Zufall zunutze macht.

Ausgangspunkt ist ein Baum, bei dem jeder Knoten zwei Ordnungskriterien trägt. Das erste Ordnungskriterium nennen wir **Schlüssel**, das zweite **Priorität**. Das folgende Beispiel zeigt einen solchen Baum, wobei der Schlüssel an jedem Knoten links oben und die Priorität rechts unten notiert ist:

Ein Baum mit zwei Ordnungskriterien Schlüssel und Priorität heißt **Treap**, (Tree+Heap) wenn er bezüglich des Schlüssels ein aufsteigend sortierter Baum und bezüglich der Priorität ein Heap ist.

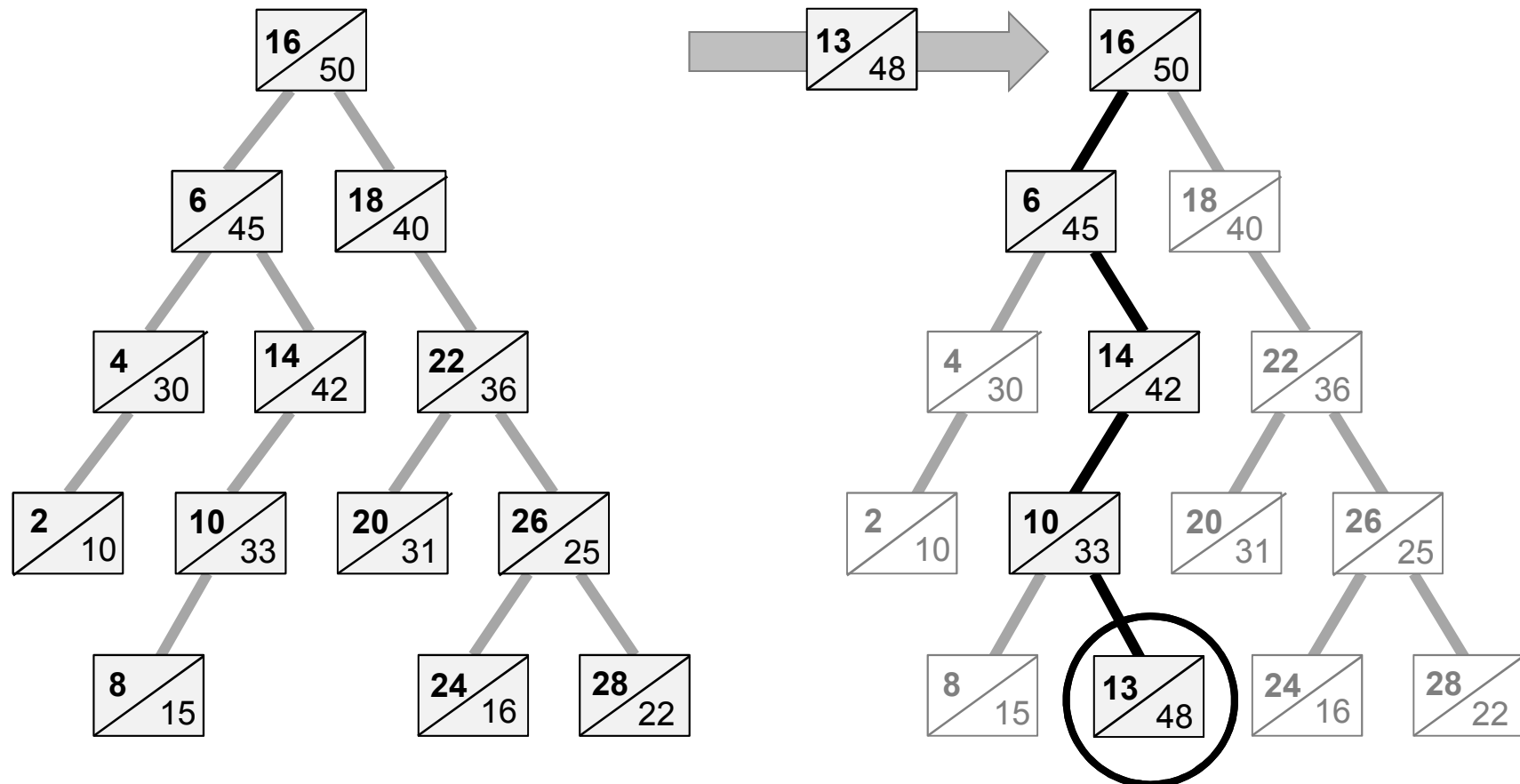
Der nebenstehende Baum ist ein Treap.



Welche Rolle der Zufall beim Aufbau eines Treaps spielt, werden wir später sehen.

## Einfügen von Elementen in einen Treap

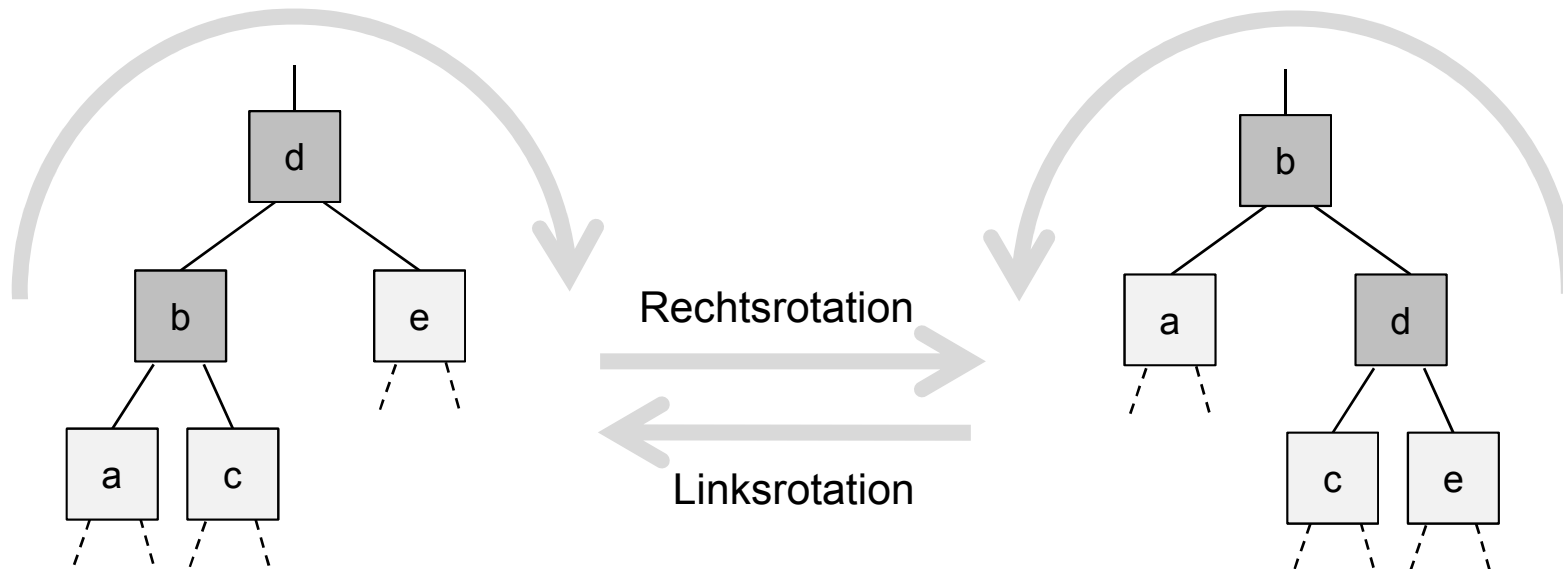
Wir wollen ein neues Element in einen Treap einfügen. Dazu fügen wir das Element entsprechend seines Schlüssels in den aufsteigend sortierten Baum ein:



Dabei ist allerdings die Heap-Eigenschaft verloren gegangen. Man könnte den Treap wiederherstellen, wenn es gelänge, den eingesetzten Knoten in Richtung Wurzel zu tauschen, ohne die aufsteigende Sortierung zu zerstören.

## Exkurs – Rotation in aufsteigend sortierten Bäumen

Wir betrachten ein Fragment in einem aufsteigend sortierten Baum ( $a \leq b \leq c \leq d \leq e$ ):



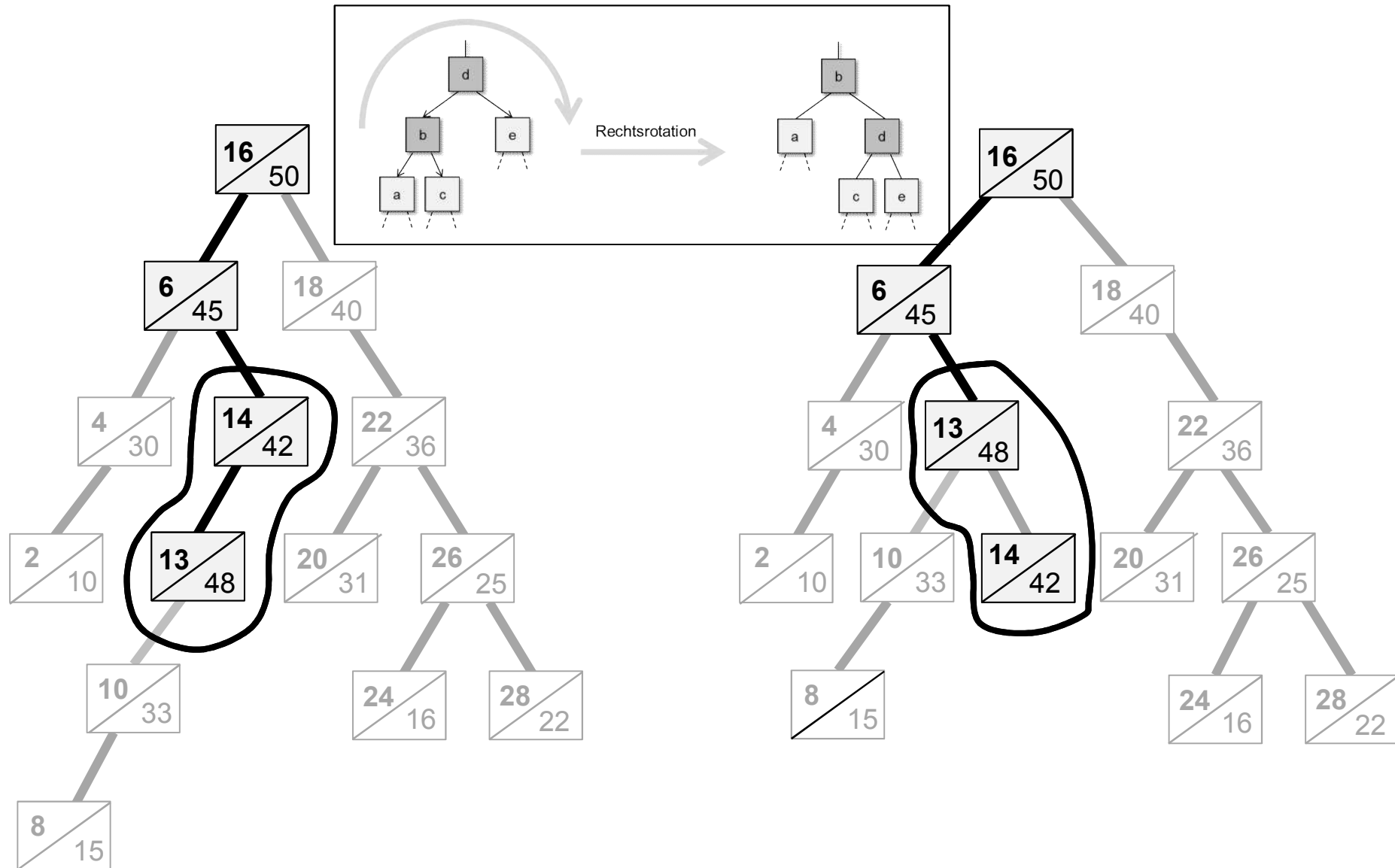
In diesem Fragment kann man die "Vater-Sohn"-Beziehung von Knoten b und d durch "Rotation" tauschen, ohne die aufsteigende Ordnung zu zerstören.

Dies bedeutet, dass man einen Knoten Schritt für Schritt im Baum zur Wurzel bewegen kann, ohne die Ordnung zu zerstören.

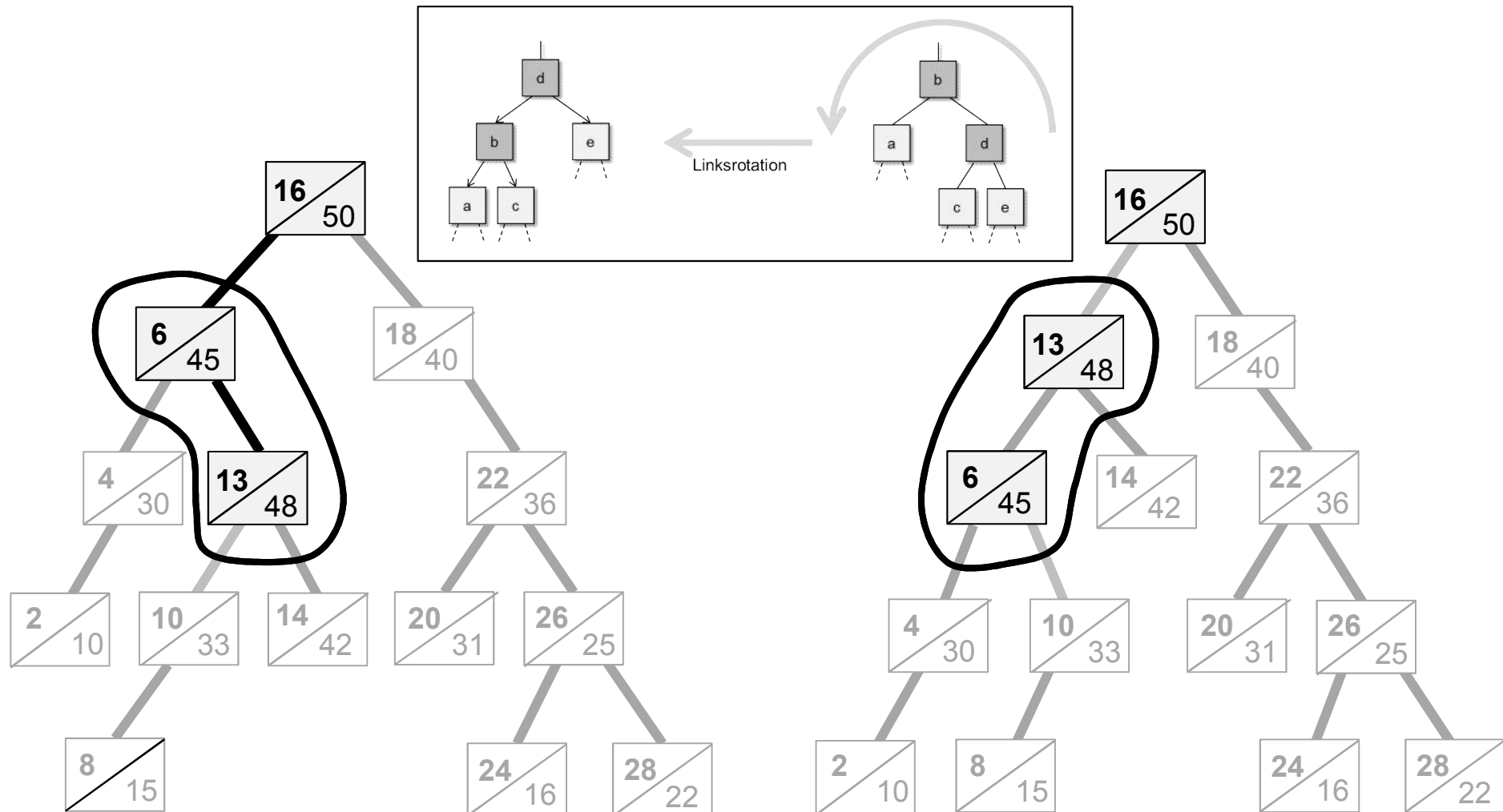
Für das Einsetzen im Treap bedeutet das, dass man die durch Einsetzen eines Elements zerstörte Heapeigenschaft durch Rotationen wiederherstellen kann, ohne die aufsteigende Sortierung zu verlieren.



## Einfügen in einen Treap - Tausche die Knoten 13 und 14 durch Rechtsrotation



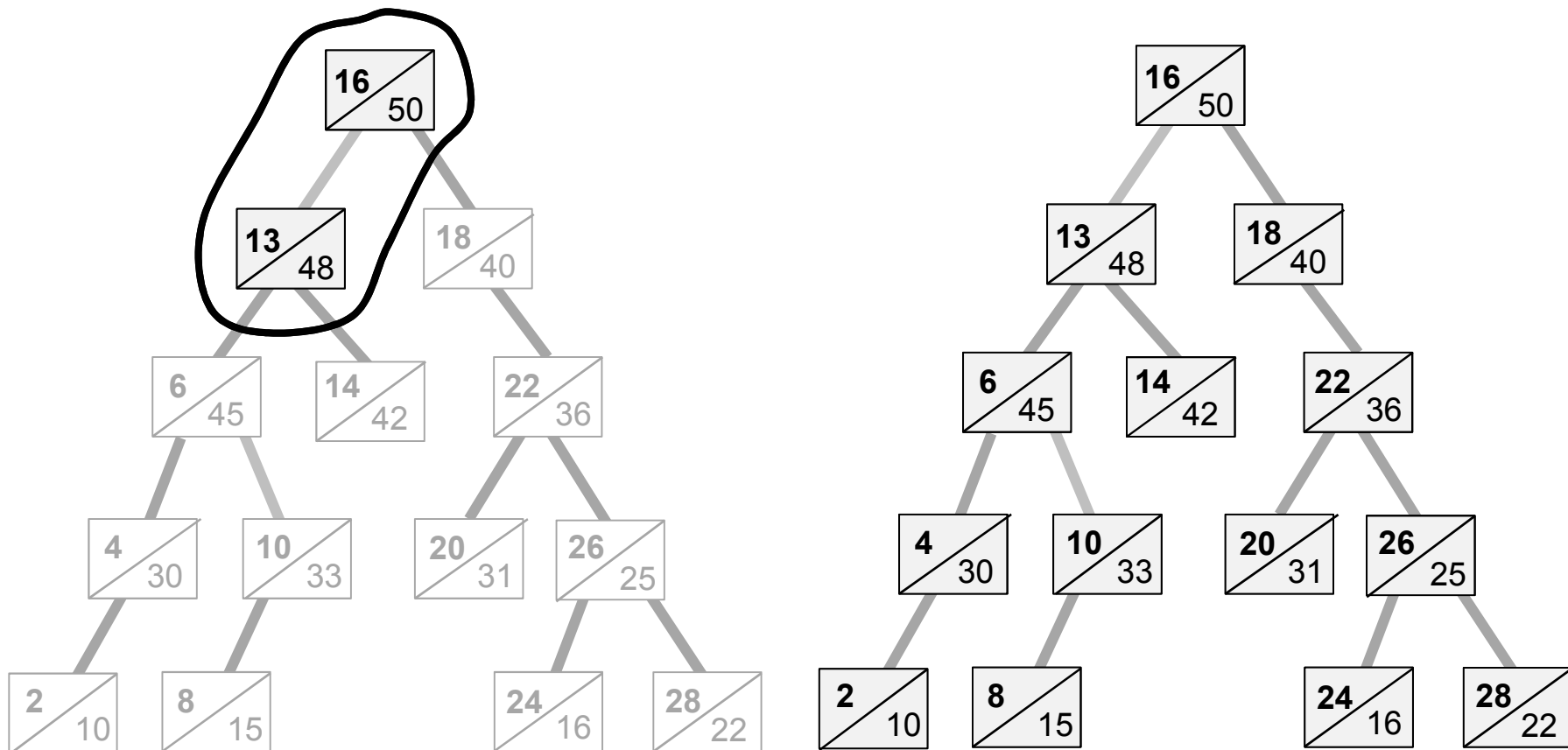
## Einfügen in einen Treap – Tausche die Knoten 13 und 6 durch Linksrotation



Jetzt ist der Heap wieder hergestellt und der Baum ist nach wie vor aufsteigend geordnet.

## Einfügen in einen Treap

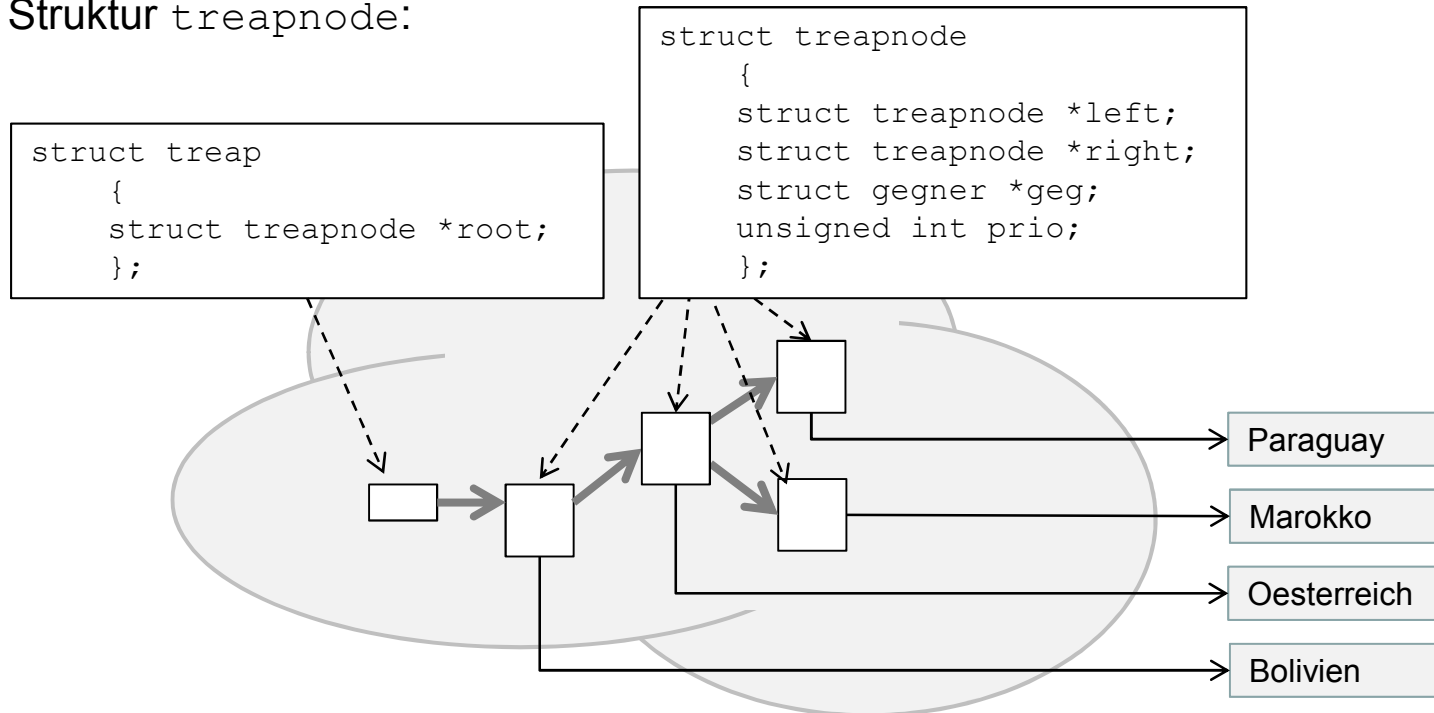
Im letzten Vergleichsschritt ist nichts mehr zu machen, die Heapbedingung ist wiederhergestellt. Da bei allen Tauschoperationen die aufsteigende Ordnung erhalten wurde, handelt es sich jetzt wieder um einen Treap.



Der Schlüssel entscheidet über die Lage eines Knotens im Baum, während über die Priorität die Höhe im Baum eingeregelt wird.

## Der Container als Treap - Datenstrukturen

Der einzige Unterschied zum Baum besteht in dem zusätzlichen Feld für die Priorität (`prio`) in der Struktur `treapnode`:

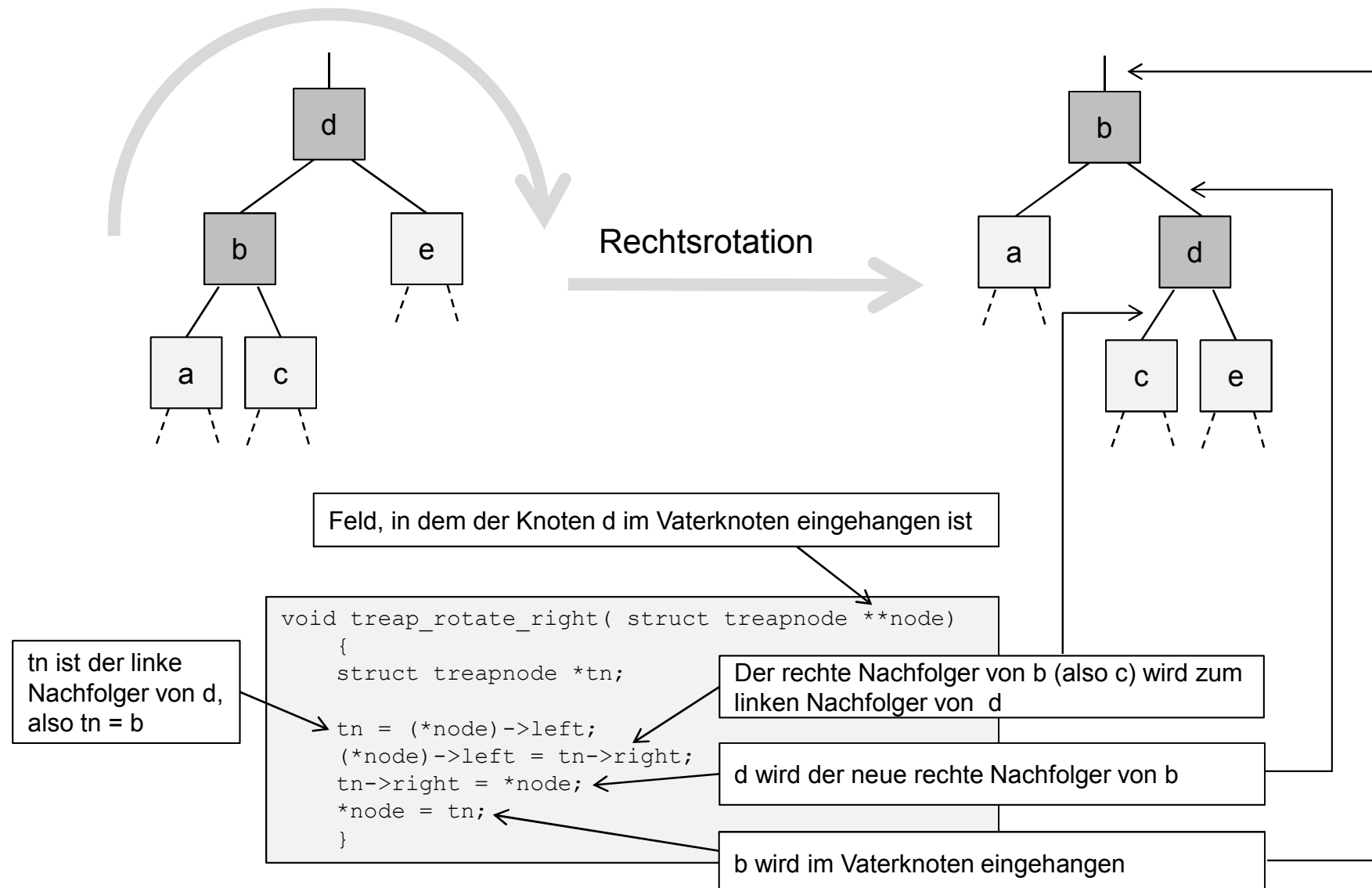


Der Container besteht aus einem Header (`struct treap`) der nur einen Zeiger auf die Wurzel des Treaps (`root`) enthält. Der eigentliche Treap ist eine Verkettung von Knoten (`struct treapnode`), die jeweils einen Zeiger auf den durch sie verwalteten Gegner (`geg`) und einen "linken" (`left`) sowie "rechten" (`right`) Nachfolger sowie die Priorität des Knotens (`prio`) enthalten.

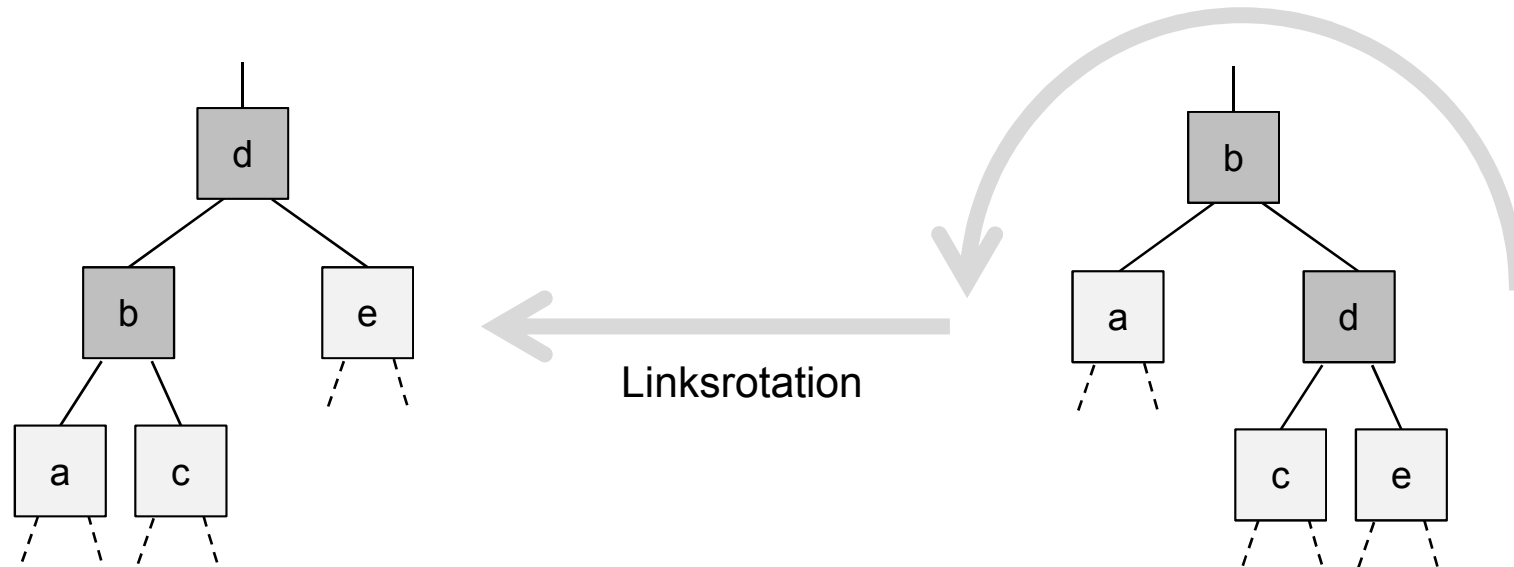
Bis auf das Einsetzen neuer Elemente sind alle Algorithmen beim Treap identisch mit denen eines Baums. Wir betrachten daher im Folgenden nur die Insert-Funktion.



## Der Container als Treap – Implementierung der Rechtsrotation



## Der Container als Treap – Implementierung der Linksrotation



Die Linksrotation wird analog zur Rechtsrotation implementiert:

```
void treap_rotate_left( struct treapnode **node)
{
    struct treapnode *tn;

    tn = (*node)->right;
    (*node)->right = tn->left;
    tn->left = *node;
    *node = tn;
}
```

## Der Container als Treap – Rekursives Einfügen eines Elements

```

int treap_insert_rek( struct treapnode **node, struct gegner *g)
{
    int cmp;

    if( *node) ← Platz besetzt
    {
        ← Namensvergleich
        cmp = strcmp( g->name, (*node)->geg->name);
        if( cmp > 0)
        {
            if( !treap_insert_rek(&((*node)->right), g))
                return 0;
            if ((*node)->prio < (*node)->right->prio)
                treap_rotate_left( node);
            return 1;
        }
        if( cmp < 0)
        {
            if( !treap_insert_rek(&((*node)->left), g))
                return 0;
            if ((*node)->prio < (*node)->left->prio)
                treap_rotate_right( node);
            return 1;
        }
        return 0; ← Element schon vorhanden
    }
    ← Platz frei, Abstieg beendet, Knoten wird eingesetzt
    *node = (struct treapnode *) malloc( sizeof( struct treapnode));
    (*node)->left = 0;
    (*node)->right = 0;
    (*node)->geg = g;
    (*node)->prio = rand();
    return 1;
}

```

Abstieg nach  
rechts,  
anschließend  
ggf. Rotation  
nach links.

Abstieg nach  
links,  
anschließend  
ggf. Rotation  
nach rechts.

Element schon vorhanden

Platz frei, Abstieg beendet, Knoten wird eingesetzt

Hier bekommt das Element seine Priorität.

In der Rekursion wird die Einfügeposition im aufsteigend sortierten Baum gesucht. Wenn noch nicht vorhanden, wird das Element eingefügt.

Dem Element wird beim Einfügen eine zufällige Priorität gegeben.

Beim Rückzug aus der Rekursion wird durch Rotationen die Heapbedingung hergestellt, wenn sie verletzt ist. Wurde beim Abstieg nach links gegangen erfolgt beim Rückzug eine Rechtsrotation. Wurde beim Abstieg nach rechts gegangen erfolgt beim Rückzug eine Linksrotation.

Insert-Funktion

```

int treap_insert( struct treap *t, struct gegner *g)
{
    return treap_insert_rek( &(t->root), g);
}

```

Einstieg in die Rekursion

## Der Container als Treap – Beseitigung der Rekursion 1

Zur Beseitigung der Rekursion verwenden wir einen Stack, auf dem wir den Pfad zur Suche der Einfügestelle speichern, da wir diesen Pfad zur Herstellung der Heapbedingung nach dem Einsetzen rückwärts durchlaufen müssen. Auf dem Stack speichern wir für jeden Verzweigungspunkt des Pfades das Feld in dem der betrachtete Knoten eingehangen ist (`node`) und die Richtung (`direction`, 0=links, 1 = rechts), in die abgestiegen wird:

```
struct stackentry
{
    struct treapnode **node;
    int direction;
};
```

Grundsätzlich läuft das Einsetzen dann wie folgt ab:

```
int treap_insert( struct treap *t, struct gegner *g)
{
    struct treapnode **node, *neu, *tn;
    int cmp;
    struct stackentry stack[256];
    int pos;

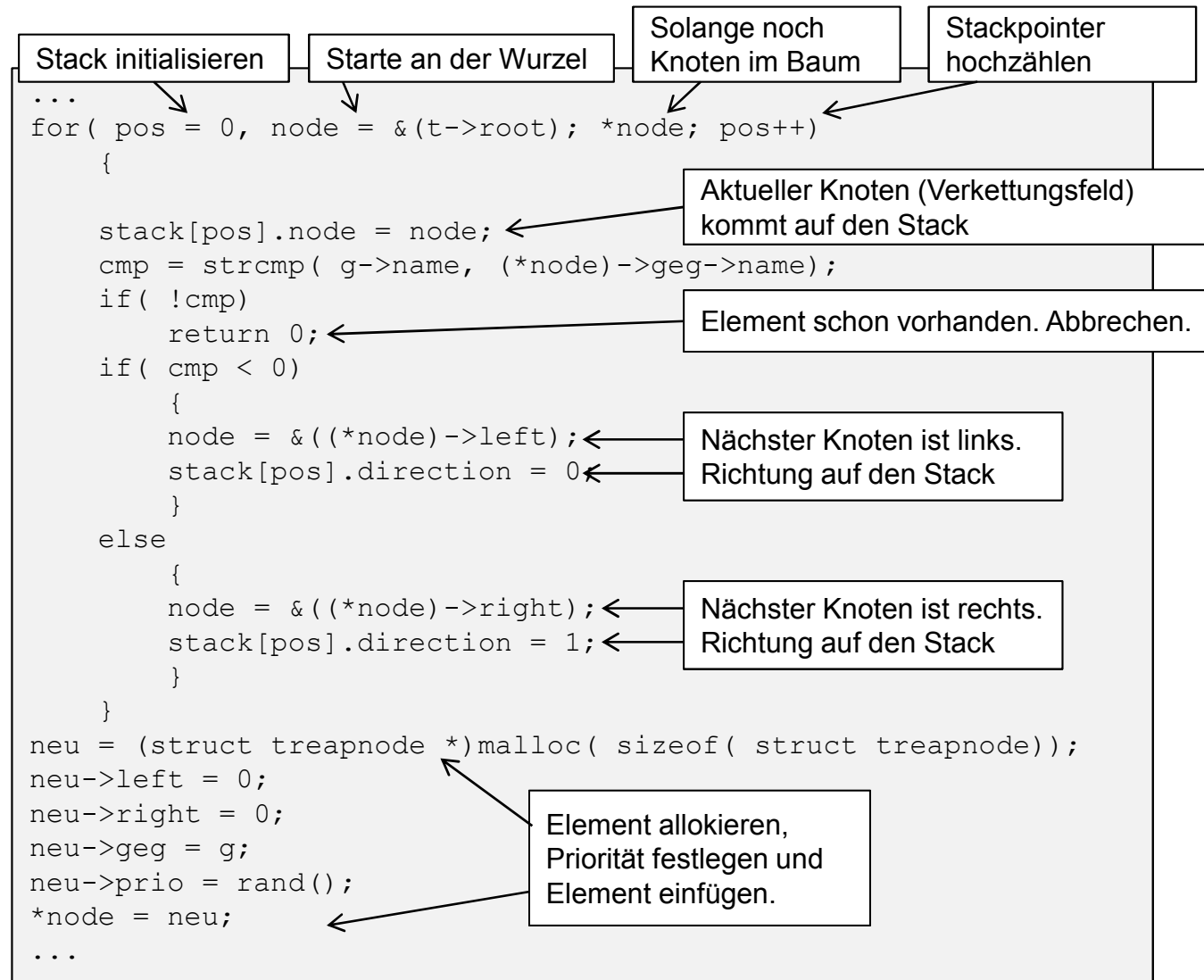
    // Einfügeposition suchen und Pfad speichern

    // Neues Element einfügen

    //Pfad rueckwaerts durchlaufen und dabei Heapbedingung durch Rotation herstellen
}
```

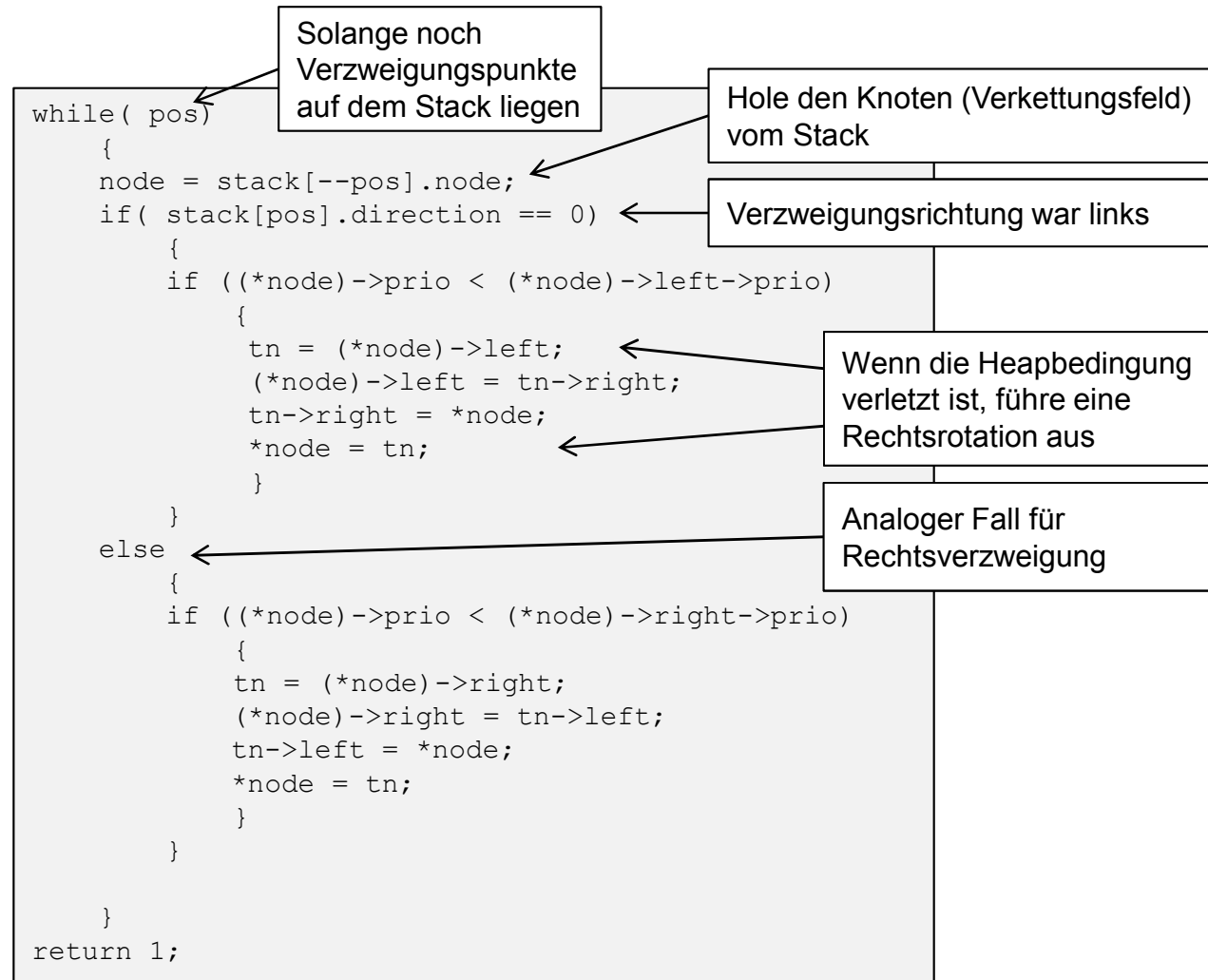
## Der Container als Treap – Beseitigung der Rekursion 2

Einfügeposition suchen, Pfad speichern und neues Element einfügen:



## Der Container als Treap – Beseitigung der Rekursion 3

Pfad rückwärts durchlaufen und dabei die Heapbedingung durch Rotation herstellen.



Die Rotationen sind hier direkt implementiert, um die Kosten für den Unterprogrammaufruf zu sparen.

## Der Container als Treap – Vergleich mit unbalanciertem Baum

Bei der Eingabe von Zufallsdaten ist bei Treaps keine Verbesserung gegenüber unbalancierten Bäumen zu erwarten, da die Daten ja bereits randomisiert sind und eine zusätzliche Randomisierung allenfalls zu zufälligen Veränderungen führt. Tiefenmessungen am Beispiel der Länderspielbilanz zeigen auch nur Unterschiede im Bereich zufälliger Schwankungen:

Baum für 50 zufällig gewählte Gegner

Max. Suchtiefe: 9

Mittlere Suchtiefe: 5.96

Treap für 50 zufällig gewählte Gegner

Max. Suchtiefe: 10

Mittlere Suchtiefe: 5.58

Der Treap ist in dieser Situation sogar schlechter einzuschätzen, da er zusätzlichen Aufwand beim Einsetzen der Objekte hat.

Bei einer sortierten Eingabe zeigt sich jedoch ein deutlicher Unterschied zugunsten des Treaps:

Baum für 50 sortierte Gegner

Max. Suchtiefe: 50

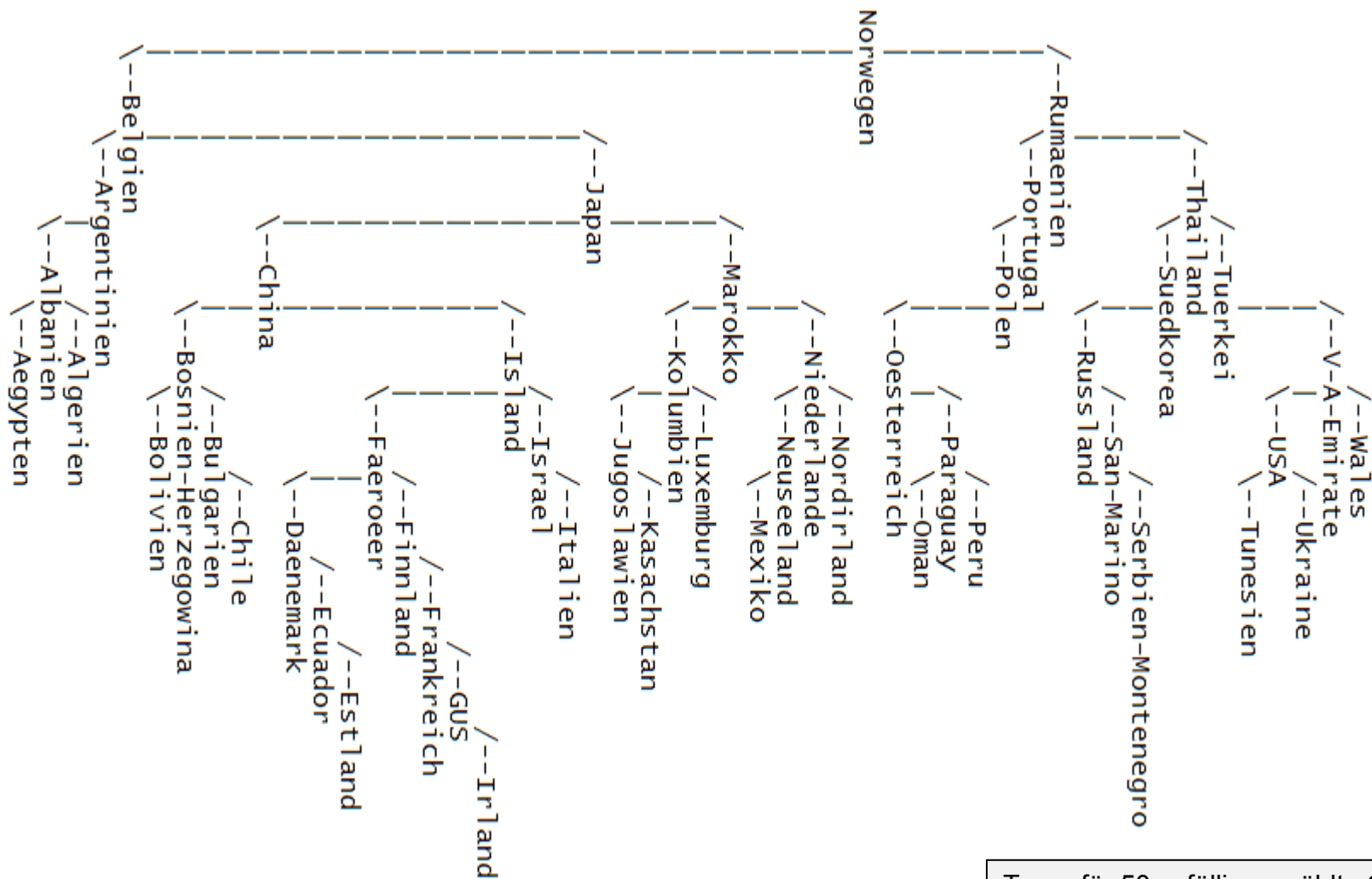
Mittlere Suchtiefe: 25.5

Treap für 50 sortierte Gegner

Maximale Suchtiefe: 9

Mittlere Suchtiefe: 5.60

Während der Baum zur Liste entartet, bleibt der Treap unbeeindruckt von der Vorsortierung der Daten. Dies zeigen auch die beiden folgenden Folien.

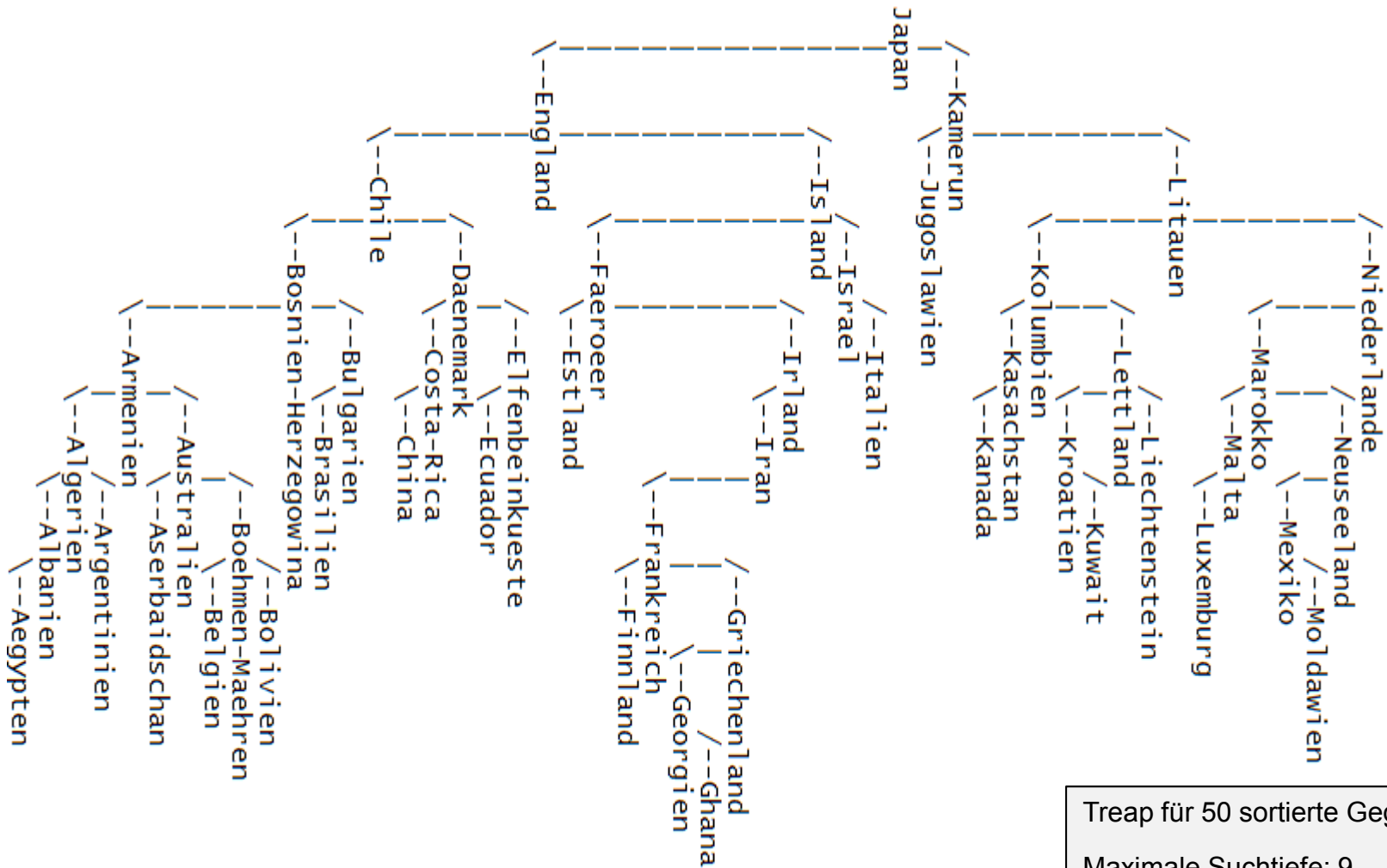


Treap für 50 zufällig gewählte Gegner

Max. Suchtiefe: 10

Mittlere Suchtiefe: 5.58





Treap für 50 sortierte Gegner

Maximale Suchtiefe: 9

Mittlere Suchtiefe: 5.60

## Hashtabellen 1

Wir stellen uns vor, dass wir für ein Übersetzungsprogramm alle Wörter eines Wörterbuchs (ca. 500000 Stichwörter) mit ihrer Übersetzung in einem Programm speichern wollen. Ein balancierter Binärbaum hätte in dieser Situation eine Suchtiefe von ca. 20. Damit sind wir nicht zufrieden. Wir haben das ehrgeizige Ziel, die Suchtiefe unter 2 zu drücken.

Ideal wäre ein Array, der für jedes Wort genau einen Eintrag hätte. Dazu müssten wir aus dem Wort einen eindeutigen Index berechnen, der dann die Position im Array festlegt. Wenn wir uns auf Worte der Länge 20 und die 26 Kleinbuchstaben a-z (gegeben durch Werte 0-25) beschränken, können wir eine einfache Funktion zur Indexberechnung angeben.

$$h(b_0, b_1, \dots, b_{19}) = b_0 \cdot 26^0 + b_1 \cdot 26^1 + b_2 \cdot 26^2 + \dots + b_{19} \cdot 26^{19}$$

Der dazu benötigte Array müsste allerdings  $26^{20}$  Felder haben, da theoretisch so viele verschiedene Wörter vorkommen können. Das ist nicht möglich.

Man könnte die Streuung der Funktion  $h$  reduzieren, indem man zum Beispiel am Ende der Berechnung eine Modulo-Operation mit der gewünschten Tabellengröße vornimmt:

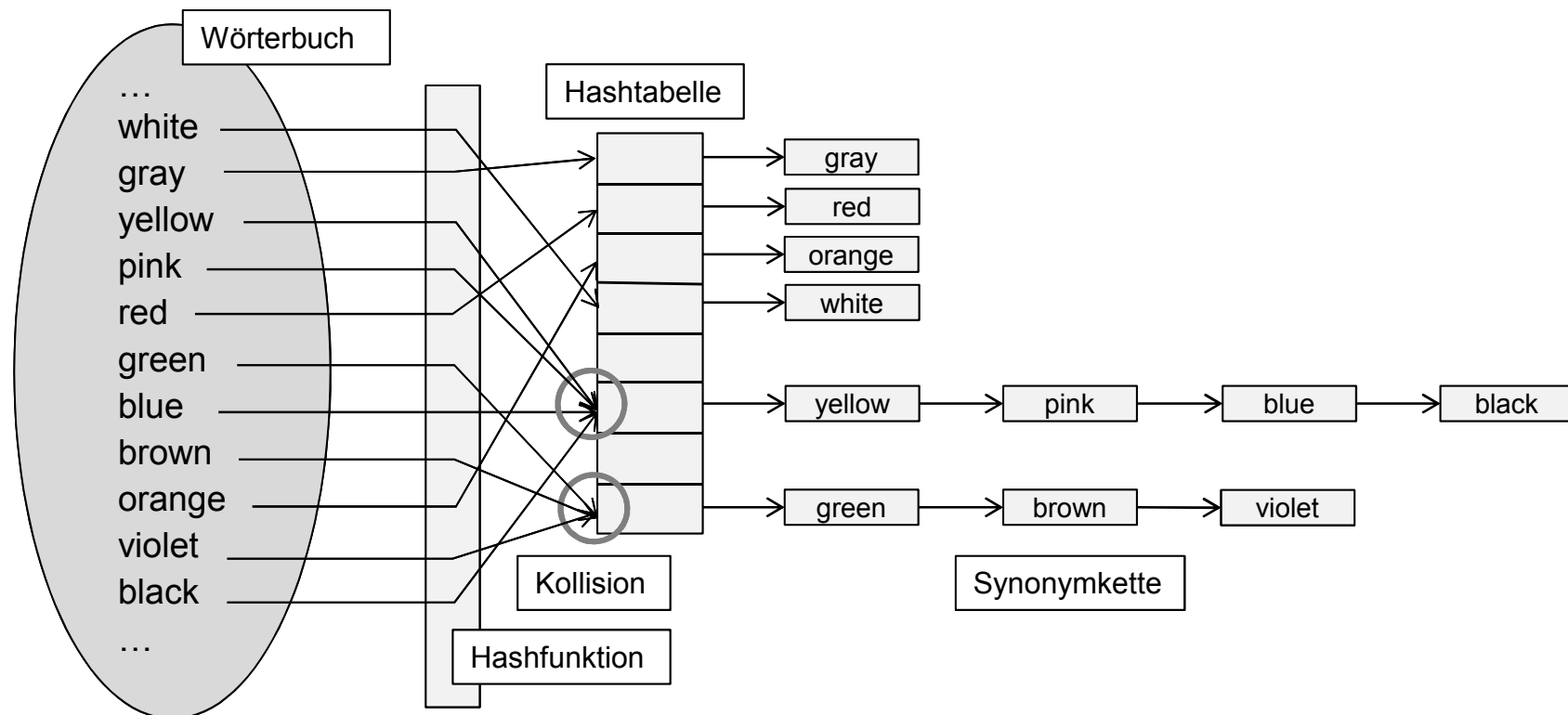
$$h(b_0, b_1, \dots, b_{19}) = (b_0 \cdot 26^0 + b_1 \cdot 26^1 + b_2 \cdot 26^2 + \dots + b_{19} \cdot 26^{19}) \% 500000$$

Eine solche Funktion nennt man eine **Hashfunktion**. Jetzt wäre allerdings nicht mehr gewährleistet, dass jedes Wort genau einen Index bekommt. Es kann jetzt vorkommen, dass verschiedene Wörter auf den gleichen Index abgebildet werden. Wir nennen dies eine **Kollision**. Im Fall einer Kollision könnte man die kollidierenden Einträge in Form einer Liste (**Synonymkette**) an den Array anhängen und hoffen, dass diese Listen relativ kurz bleiben.

Die auf diese Weise entstehende Datenstruktur nennt man ein Hashtabelle.

## Hashtabellen 2

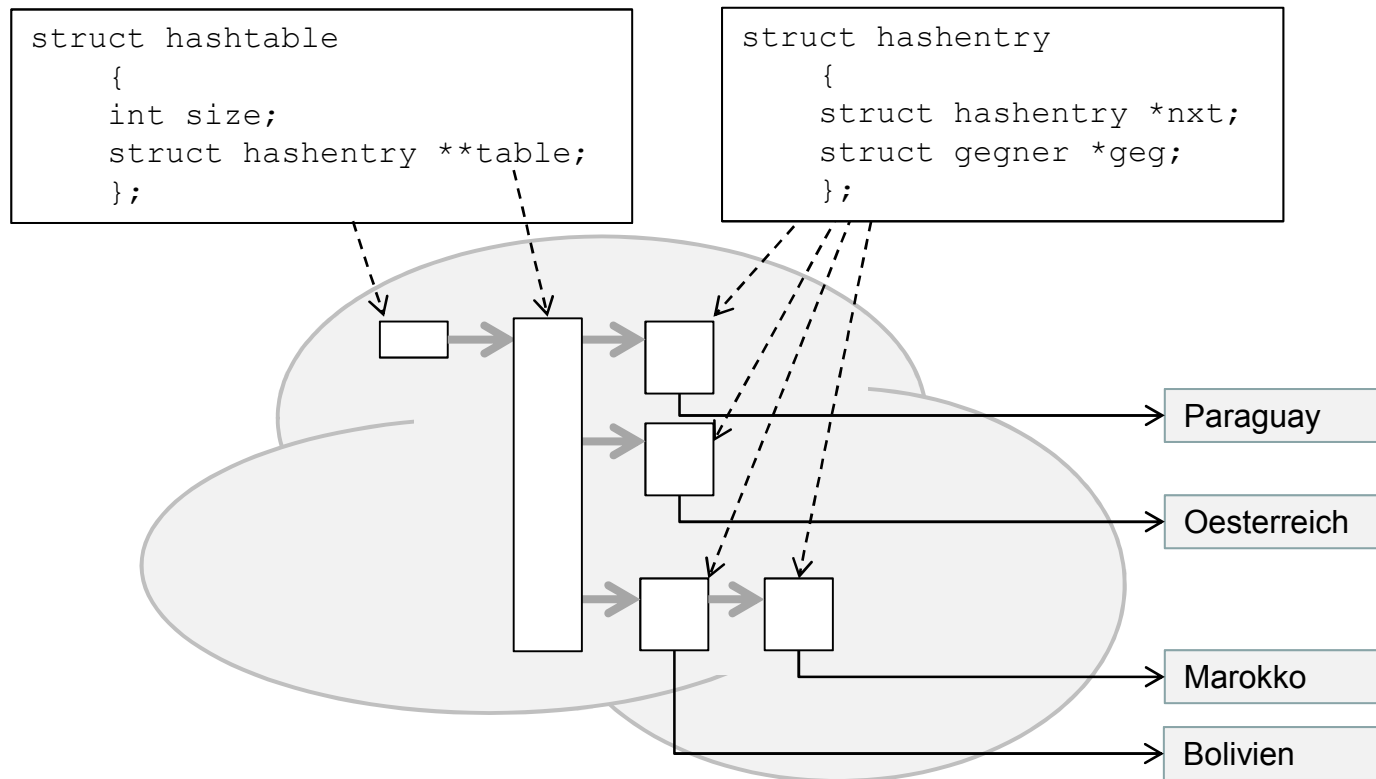
Hashtabellen kombinieren die Geschwindigkeit von Arrays mit der Flexibilität von Listen. Durch eine breite Vorselektion über einen Array findet man eine hoffentlich kurze Liste, die dann zu durchsuchen ist:



Die Hashfunktion hat entscheidenden Einfluss auf die Performance der Hashtabelle. Die Hashfunktion sollte möglichst zufällig und breit streuen, um wenig Kollisionen zu erzeugen und sehr effizient zu berechnen sein, damit durch die bei jedem Zugriff stattfindende Vorselektion möglichst wenig Rechenzeit verloren geht.

## Der Container als Hashtabelle - Datenstrukturen

Im Container implementieren wir einen dynamisch allokierten Array, an dem die Synonymketten angehängen werden.



Der Container besteht aus einem Header (`struct hashtable`) der neben der Größe der Tabelle einen Zeiger auf die eigentliche Hashtabelle (`struct hashentry **`) enthält. In der Hashtabelle stehen Zeiger auf die Synonymkette, die aus Verkettungselementen (`struct hashentry`) besteht, die jeweils einen Zeiger auf den durch sie verwalteten Gegner (`geg`) und einen Zeiger auf das nächste Listenelement (`nxt`) enthalten. Die Synonymketten sind strukturell genauso aufgebaut wie die Listen im Listencontainer.

## Der Container als Hashtabelle – Erzeugen eines leeren Containers

Ein leerer Container besteht aus einem Header (`struct hashtable`) an dem bereits eine Tabelle angehängen ist:

```
struct hashtable *hash_create( int siz)
{
    struct hashtable *h;

    h = (struct hashtable *)malloc( sizeof( struct hashtable));
    h->size = siz;
    h->table = (struct hashentry **)calloc( siz, sizeof( struct hashentry *));
    return h;
}
```

Die gewünschte Tabellengröße (`siz`) wird als Parameter übergeben und in die Headersstruktur eingetragen (`h->size`). Danach wird die Tabelle allokiert. Die Tabelle enthält initial nur NULL-Zeiger (`calloc`), da noch keine Daten verlinkt sind.

Bei jedem Aufruf der `hash_create`-Funktion wird ein neuer Container erzeugt. Ein Anwendungsprogramm kann daher mehrere Container erzeugen und unabhängig voneinander verwenden:

```
struct hashtable *container1;
struct hashtable *container2;

container1 = hash_create(1000);
container2 = hash_create(5000);
```

## Der Container als Hashtabelle – Die Hashfunktion

Hashtabellen und Hashfunktionen (man sagt auch Streuwertfunktion) sind keine Erfindung der Informatik, es gibt sie schon seit ewigen Zeiten. Zum Beispiel ist eine Registratur, in der Akten nach dem ersten Buchstaben eines Stichworts abgelegt werden, eine Hashtabelle. Kollidierende Akten kommen dann in das gleiche Fach und müssen dort sequentiell gesucht werden. Die zugehörige Hashfunktion ist:

```
unsigned int hashfunktion( char *name)
{
    return *name;←
```

Erstes Zeichen des Namens

Diese Hashfunktion sehr einfach, aber für große Registraturen unbrauchbar, da sie nur sehr gering streut. Die mathematische Analyse von Hashfunktionen ist sehr komplex und soll hier nicht betrieben werden. Wir verwenden in unseren Beispielen die folgende Funktion:

```
unsigned int hashfunktion( char *name, unsigned int size)
{
    unsigned int h;

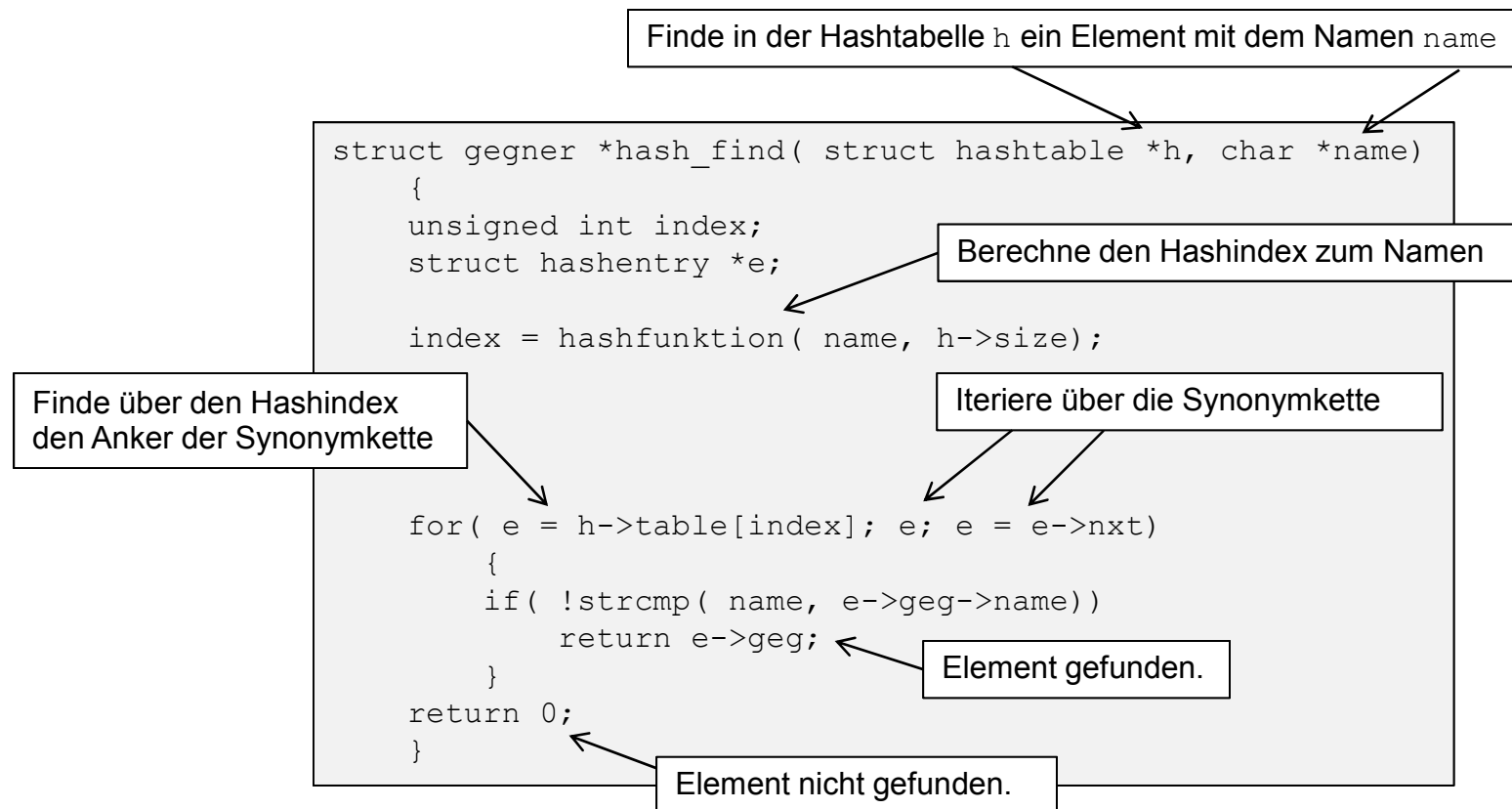
    for( h = 0; *name; name++)
        h = ((h << 6) | (*name - '@')) % size;
    return h;
}
```

Komplexe Berechnung, in die alle Zeichen des Namens "gleichberechtigt" eingehen.

Hashfunktionen haben auch in anderen Bereichen der Informatik (z.B. Kryptologie) eine große Bedeutung. Mit Hashfunktionen (z.B. MD5, Message-Digest Algorithm 5) versucht man "Fingerabdrücke" von Daten zu gewinnen, aus denen man keine Rückschlüsse auf die Ausgangsdaten gewinnen kann.

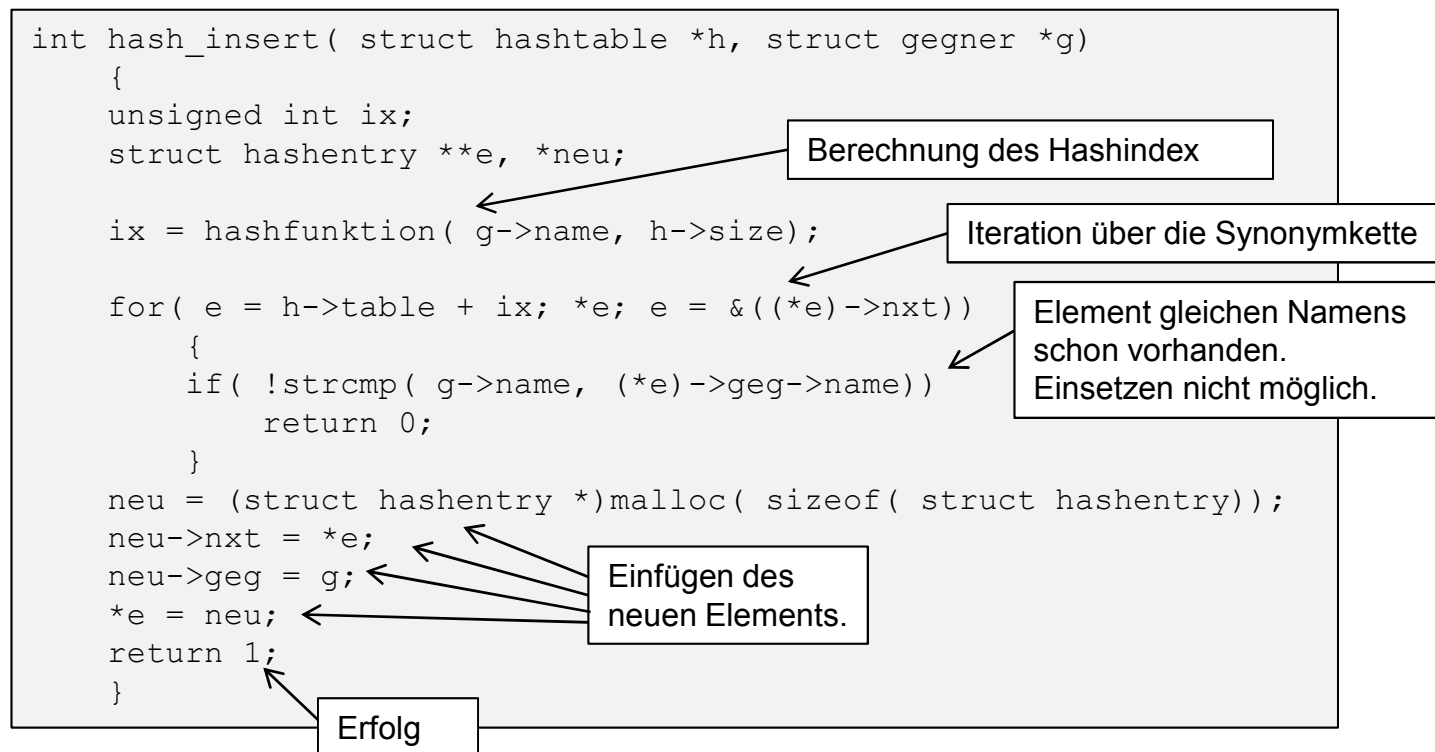
## Der Container als Hashtabelle – Finden eines Elements

Um ein Element zu finden, wird zunächst mit der Hashfunktion der Einstieg in die Hashtabelle berechnet. In der Tabelle steht dann der Anker der Synonymkette, oder 0, wenn zu dem Hashwert noch nichts gespeichert wurde. In der Synonymkette wird das Element dann gesucht.



## Der Container als Hashtabelle – Einfügen eines Elements

Das Einsetzen in die Hashtabelle verläuft analog zur Suche. Mit der Hashfunktion wird der Einstieg in die Synonymkette berechnet. Das dann folgende Einsetzen in die Synonymkette mittels doppelter Indirektion kennen wir bereits als Listenoperation:

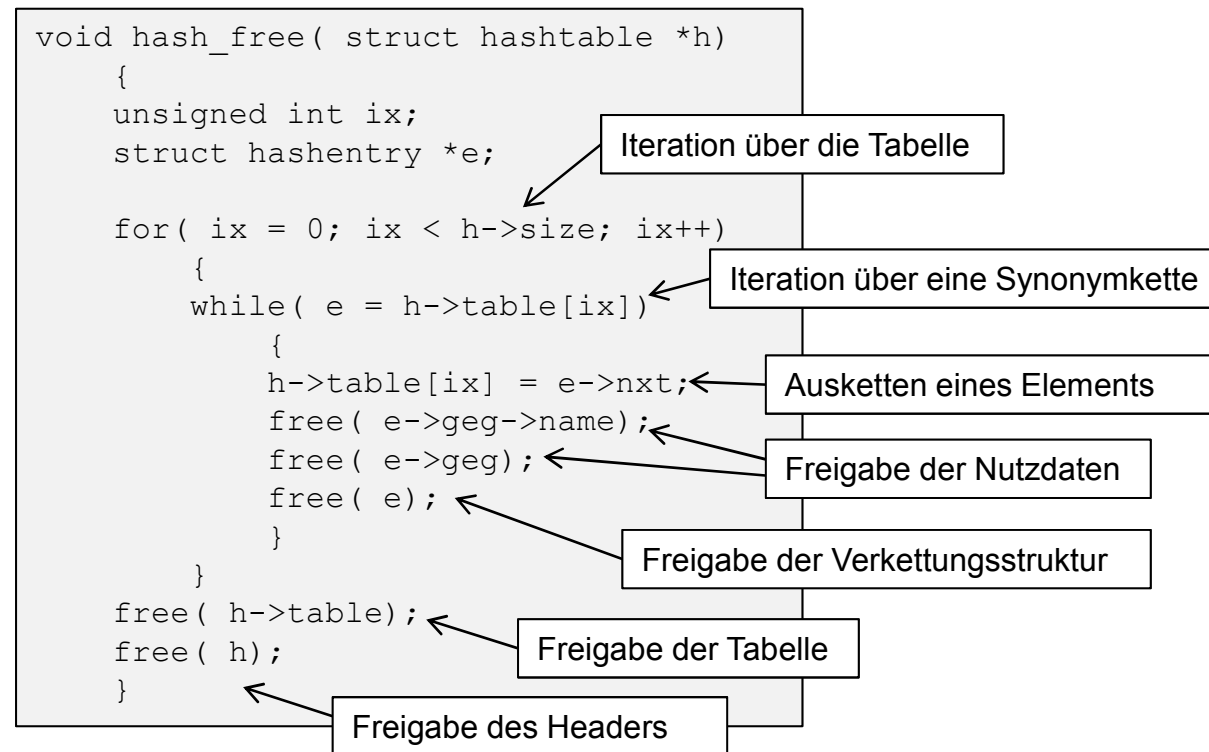


Im Gegensatz zum Listencontainer werden die Listen hier nicht alphabetisch sortiert aufgebaut. Die Listen werden kurz sein, sodass sich der Zusatzaufwand für das Sortieren wahrscheinlich nicht auszahlt.



## Der Container als Hashtabelle – Freigabe des Containers

Bevor die Hashtabelle und der Header freigegeben werden können, muss über die Tabelle iteriert werden, um alle Synonymketten mit allen anhängenden Datensätzen freizugeben:



Beachten Sie dass bei `while( e = h->table[ix])` eine Zuweisung an `e` erfolgt. Sollte dabei der Null-Zeiger zugewiesen worden sein, wird die Schleife abgebrochen.

## Der Container als Hashtabelle – Anwendungsprogramm

Das Einlesen der Daten und das Anwendungsprogramm enthalten nur minimale Abweichungen von den zuvor betrachteten Containertypen und müssen daher nicht erneut betrachtet werden.

Viel interessanter sind die Ergebnisse für unterschiedliche Tabellengrößen.

Die Hashtabelle zeigt, sehr geringe Suchtiefen, selbst dann, wenn die Tabelle nur so groß ist, wie die Anzahl der zu erwartenden Nutzdaten.

Hashtabelle für 50 Gegner

Tabellengröße 50

Maximale Suchtiefe: 5

Mittlere Suchtiefe 1.44

Tabellengröße 100

Maximale Suchtiefe: 4

Mittlere Suchtiefe 1.24

Tabellengröße 200

Maximale Suchtiefe: 3

Mittlere Suchtiefe 1.16

49:	Rumaenien.
48:	Chile.
47:	
46:	
45:	
44:	
43:	Argentinien.
42:	
41:	
40:	San-Marino.
39:	Bolivien, Oesterreich, Oman, Faeroeer, Kasachstan.
38:	
37:	
36:	Italien.
35:	GUS.
34:	Ecuador.
33:	USA.
32:	
31:	Luxemburg.
30:	Belgien.
29:	Bosnien-Herzegowina, Suedkorea.
28:	Japan, Tunesien.
27:	
26:	Kolumbien.
25:	Daenemark, Serbien-Montenegro.
24:	Finnland, Jugoslawien.
23:	
22:	Bulgarien, Nordirland.
21:	China, Tuerkei.
20:	Portugal, Estland.
19:	Niederlande.
18:	Frankreich.
17:	
16:	Algerien, Island.
15:	
14:	Norwegen.
13:	Wales.
12:	Polen, Israel.
11:	
10:	Neuseeland.
9:	Peru.
8:	
7:	Ukraine, V-A-Emirate.
6:	Albanien.
5:	Marokko.
4:	Russland.
3:	Paraguay, Mexiko.
2:	Thailand, Aegypten.
1:	
0:	Irland.

Hashtabelle für 50 Gegner

Tabellengröße 50

Maximale Suchtiefe: 5

Mittlere Suchtiefe 1.44

## Speicherkomplexität der Container

Alle Verfahren benötigen zusätzlichen Speicher, zum Aufbau der Datenstrukturen. Wir bezeichnen den Speicherbedarf für einen Pointer/Integer mit  $p$ . Dann ergibt sich

### Listen

$$s(n)=2pn$$

### Bäume

$$s(n)=3pn$$

### Treaps

$$s(n)=4pn$$

**Hashtabellen** (bei dreimal so großer Tabelle, wie Daten erwartet werden)

$$s(n)=5pn$$

## Laufzeitkomplexität Containeralgorithmen

Die Laufzeit von insert und find ist bei allen Containern proportional zur Suchtiefe. Die folgende Tabelle zeigt die Suchtiefen für zufällig generierte Daten:

	Liste		Baum		Treap		Hashtabelle	
Anzahl	Maximum	Durchschnitt	Maximum	Durchschnitt	Maximum	Durchschnitt	Maximum	Durchschnitt
1000	1000	500	20	11	23	13	2	1,16
10000	10000	5000	30	16	29	16	2	1,03
100000	100000	50000	40	21	41	21	2	1,07
1000000	1000000	500000	52	25	49	25	4	1,34

Wie zu erwarten wachsen die Suchtiefen bei Listen linear, bei Bäumen und Treaps logarithmisch und die Suchtiefe beim Hashing ist konstant. Letzteres gilt allerdings nur, wenn die Tabellengröße proportional zum Datenvolumen ist.

## Vergleich Tree und Treap für vorsortierte Daten

Bei sortierten Daten hat der Baum lineare Suchtiefe, während der Treap die logarithmische Tiefe behält.

Auf das Hashing hat eine Vorsortierung keinen Einfluss.

	Baum		Treap	
Anzahl	Maximum	Durchschnitt	Maximum	Durchschnitt
1000	1000	500	20	11
10000	10000	5000	29	17
100000	100000	50000	39	21
1000000	1000000	500000	49	25