DHBW
Duale Hochschule
Baden-Württemberg

AIRBUS
DEFENCE & SPACE

Listenstrukturen

Lukas Abelt
lukas.abelt@airbus.com

DHBW Ravensburg
Wirtschaftsinformatik

Ravensburg
22. April 2019

Inhalt I

Inhalt

Allgemeines

Zu Listenstrukturen

- Daten können verschieden strukturiert werden
  - Abhängig vom Inhalt
  - Oder Verwendungszweck
- Algorithmen sind oft von der darunterliegenden Datenstruktur abhängig
- Je nach Problem sind verschiedene Datenstrukturen besser oder schlechter geeignet
- Wie wir unsere Daten strukturieren hängt also ab von:
  - Der Beschaffenheit der Daten
  - Wofür werden die Daten verwendet

Allgemeines

Praktische Umsetzung

- Listenstrukturen sind als „ready to use “ Strukturen in allen gängigen Sprachen vorhanden:
  - *Collections* Framework in Java
  - *Collection* Interface in C#
  - *STL-Container* in C++
- Die grundlegenden Eigenschaften haben wir schon besprochen...
- ...jetzt gehen wir etwas genauer auf die technischen Hintergründe ein

Ziele des Moduls

- Am Ende des Moduls sollt ihr:
  - Die Unterschiede der einzelnen Listenstrukturen kennen
  - Die Vor- und Nachteile benennen können
  - Die Grundoperationen von Listenstrukturen kennen
  - Die Unterschiede der Grundoperationen in den Listenstrukturen verstehen

Grundoperationen

In Listenstrukturen

- Für Listen existieren die folgenden Grundoperationen:
  - Anlegen der Listenstruktur
  - Zugriff auf ein Listenelement
  - An-/Einfügen von Listenelementen
  - Ermitteln der Länge der Liste(Bzw. bestimmen der Anzahl der Elemente)
  - Entfernen von Elementen
  - Konkatinieren von Listen

Inhalt

Allgemeines

Über Arrays

- Einfachste Listenstruktur
- Speichert Daten sequentiell im Speicher
- In der Regel nur in einer Dimension
- Aber auch „mehrdimensionale“ Arrays möglich
- Logisch (und auch vom Zugriff) könnte man Arrays vergleichen mit:
  - Vektoren bei eindimensionalen Arrays
  - Matrizen für zwei- oder höherdimensionale Arrays

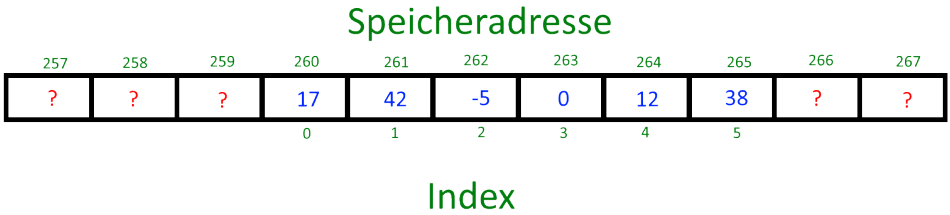
Eigenschaften

Von Arrays

- Belegen einen fortlaufenden Bereich im Speicher
- Dadurch:
  - Muss die Größe bei Initialisierung bekannt sein
  - Sind Zugriffe auf die Elemente sehr schnell
  - Löschen-/Einfügen von Elementen jedoch vergleichsweise „teuer“
- Technisch ist ein Array meist lediglich ein Zeiger auf den Beginn des Arrays
- Arrays sind meist für Datenmengen mit vielen Elementen(Ab 65535) ungeeignet
- Warum?

Datenstruktur von Arrays

Bildlich dargestellt



Grundoperationen

Anlegen eines Arrays

- Bei Anlegen eines Arrays muss immer die Größe festgelegt werden
- Vorher können keine Elemente hinzugefügt bzw. manipuliert werden
- Bei Initialisierung des Arrays wird ein Speicherbereich für dieses reserviert
- Größe des Speicherbereichs für ein Array der Größe N ergibt sich aus:

$$Speicher_{Byte} = ElementSize_{Byte} \cdot N$$

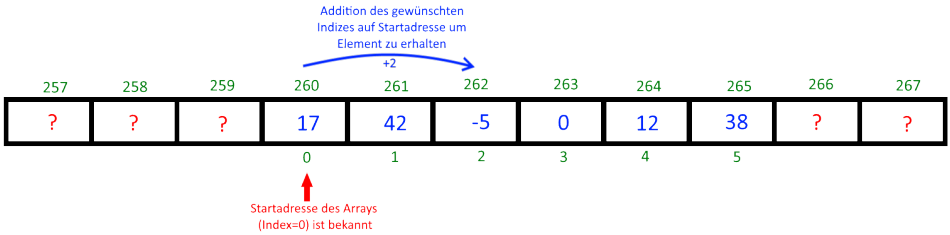
Grundoperationen

Zugriff auf Elemente

- Zugriffe auf Listenelemente passieren immer in konstanter Zeit
- Dadurch ergibt sich die Komplexität zu: $O(1)$
- Begründung:
  - Elemente liegen „hintereinander“ im Speicher $\rightarrow$ Haben fortlaufende Speicheradressen
  - Die Adresse des ersten Elements ist immer bekannt
  - Heißt, bei Abrufen des n-ten Elements muss von der Startadresse nur eine bestimmte Schrittzahl addiert werden
  - Jeder Zugriff auf ein Element ist somit (auf unterster Ebene) eine Addition und eine Leseoperation

Graphische Darstellung

Zugriff auf Elemente



Grundoperationen

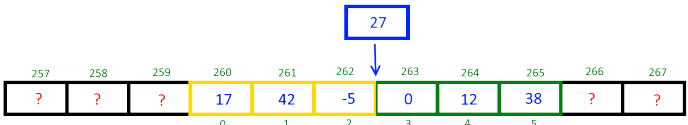
An-/Einfügen von Elementen

- Nachträgliches Einfügen von Elementen ist nicht trivial
- Grund dafür ist die Speicherstruktur von Arrays
- Dadurch, dass die Elemente fortlaufend im Speicher liegen...
  - ...müsste bei Einfügen sichergestellt sein, dass der nachfolgende Speicher noch nicht genutzt ist (Was meist nicht der Fall ist)
  - ...und alle Elemente die nach dem eingefügten Element liegen, nach rechts „verschoben“ werden
- Dadurch häufig das anlegen eines neuen Arrays nötig
- Und das ausführen von vielen Kopieroperationen

Einfügen in Arrays

Visualisiert

Einfügen von "27" an Index 3:



Komplexität

Beim einfügen

- Theoretischer Best-Case:
  - Einfügen am Ende des Arrays
  - ...solange der nachfolgende Speicher noch ungenutzt ist
  - Dann wäre lediglich eine Kopieroperation erforderlich
- In der Regel kann jedoch davon ausgegangen werden, dass...
  - ...ein neues Array (an einem anderen Speicherbereich) angelegt werden muss
  - ...und jedes Element des Arrays kopiert werden muss
- Komplexität ergibt sich hierbei zu: $O(N)$

Ermitteln der Länge

In Arrays

- Bestimmen der Elemente eines Arrays nicht trivial möglich
- Da Array nur einen Speicherbereich beschreibt und seine eigene Größe nicht kennt
- Zählen von Elementen nicht möglich, da Abbruchbedingung nicht bekannt
- Größe muss somit selbst gespeichert und verwaltet werden
  - Passiert in Java automatisch
  - Da ein Array auch immer ein Objekt ist
  - Bestimmen der Größe über das `length` Attribut

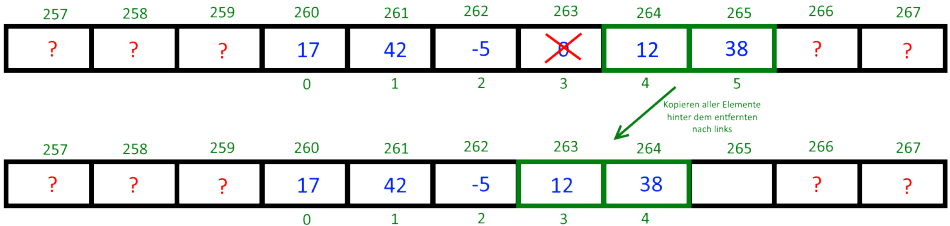
Entfernen von Elementen

In Arrays

- Entfernen führt zu Problem im Speicherbereich:
- Durch das Entfernen würde eine „Lücke“ im Array entstehen
- Daten im Array müssen jedoch fortlaufenden Speicher belegen
- D.h., Alle Elemente hinter dem entfernten Element müssen eine Position nach links verschoben werden
  - „Teure“ Kopiervorgänge nötig
- Komplexität ergibt sich somit zu $O(N)$ (Worst-Case)

Entfernen von Elementen

Visualisiert



Entfernen von Elementen

Weitere Aspekte

- Beim entfernen von Elementen wird hinten Speicher „frei“
- Dieser ist (technisch) noch dem Array zugeordnet
  - Heißt: Größe des Arrays muss theoretisch aktualisiert werden (Wenn man diese manuell verwalten muss)
- Freigewordener Speicher kann jedoch später wiederverwendet werden:
  - Elemente können theoretisch hinzugefügt werden ohne, dass das Array vergrößert werden muss
  - Dafür müssen dann jedoch zwei Werte getracked werden:
  - Die reservierte Größe des Arrays
  - Die aktuell genutzte Größe des Arrays
  - `ArrayList` Implementierung arbeitet ähnlich

Konkatinieren von Arrays

- Beim kombinieren zweier Arrays ergibt sich ähnliches Problem wie beim Einfügen
- Array hat ggf. nicht genug Platz um Elemente von beiden Arrays aufzunehmen
- Daher sind meist folgende Schritte nötig um zwei Arrays mit Länge N und M zu kombinieren:
  - Anlegen eines neues Arrays mit Größe $N + M$
  - Kopieren aller Elemente des ersten Arrays in den vorderen Teil
  - Kopieren aller Elemente des zweiten Arrays in den hinteren Teil
- Dadurch ergibt sich die Komplexität zu $O(N + M)$ (inklusive Overhead für Anlegen des neuen Arrays)

Inhalt

Allgemeines

Über Linked Lists

- Simple Listenstruktur
- Elemente können verteilt im Speicher liegen
- Jedes Element speichert einen Datenwert
- ...und eine Referenz zum Nachfolger
- Speichern mehrdimensionaler Daten theoretisch möglich
  - Wenn der Datenwert auch wieder eine Linked List ist
  - Zugriff allerdings weniger intuitiv als im Array

Eigenschaften

Von Linked Lists

- Muss keinen fortlaufenden Teil im Speicher belegen
- Dadurch entfallen einige Nachteile des Arrays:
  - Größe muss nicht bei Initialisierung bekannt sein
  - Liste kann dynamisch wachsen bzw. schrumpfen (Ohne „teuere“ Kopiervorgänge)
  - Einfügen bzw. Entfernen ist schneller
- Allerdings ist der Zugriff auf Elemente langsamer

Datenstruktur von Linked Lists

Listenelemente (Vgl. [3])

- Jedes Element einer Linked List ist ein Objekt
- Jedes Objekt vom Typ *ListElement* besteht aus:
  - Einem Member das den Wert des aktuellen Elements speichert. Datentyp ist der Typ, den die Liste speichert (`Double`, `Integer` etc.)
  - Einem Member welches das nächste Element (bzw. eine Referenz darauf) in der Liste repräsentiert. Datentyp ist hier `ListElement`
- Gibt es kein nächstes Element in der Liste kann dies über ein spezielles `tail` Objekt oder eine `null` Referenz dargestellt werden

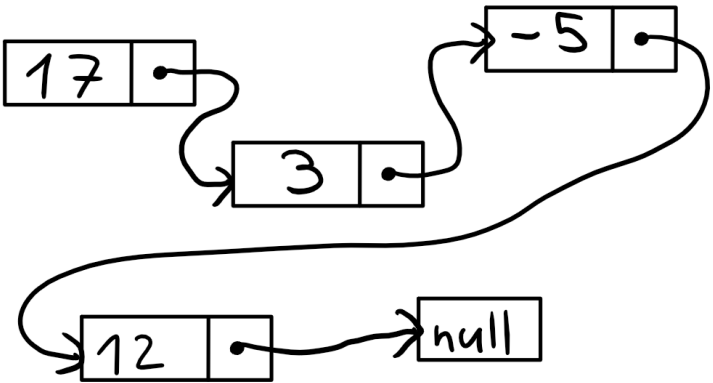
Datenstruktur von Linked Lists

Liste (Vgl. [3])

- Die Liste vom Typ `LinkedList` besteht an sich lediglich aus:
  - Einem Member `head`(Auch: `first`, `top` o.Ä.) vom Typ `ListElement`, welches das erste Element der Liste repräsentiert.
  - Den für die Liste benötigten Operationen zum Zugriff und Manipulation von Listenelementen
- Sollte die Liste leer sein, so ist `head` ein Verweis auf `null` oder ein spezielles Element, das das Ende einer Liste repräsentiert.

Datenstruktur von Linked Lists

Visualisiert



Grundoperationen

Anlegen einer Linked List

- Bei Anlegen einer neuen Linked List wird diese als leere Liste initialisiert
- Anders als beim Array muss kein Speicher für die Listenelemente reserviert werden
- Das heißt, das `head` Element wird mit einer `null` Referenz initialisiert
- Das erste Element, das hinzugefügt wird, wird automatisch zum `head` Elemente der Liste
- Wird ein Element hinzugefügt, wird erst zum Zeitpunkt des Hinzufügens der Speicher für dieses Element reserviert

Grundoperationen

Zugriff auf Elemente (Vgl. [3])

- Zugriff auf Elemente muss immer sequentiell vom `head` Element aus erfolgen
- Zugriff auf das n-te Element erfolgt durch wiederholtes Zugreifen auf das `next` Element
- Bedeutet:
  - Je weiter hinten das gesuchte Element steht, desto mehr Operationen sind möglich
  - Somit ergibt sich die Komplexität zu $O(N)$

Zugriff auf Listenelemente

Mögliche Codeimplementierung

```
1 public T getElement(int index){
2     ListElement res = head;
3     for(int i=0;i<index;i++){
4         res = res.getNext();
5     }
6     return res.getValue();
7 }
```

Grundoperationen

An-/Einfügen von Elementen

- Nachträgliches Einfügen vergleichsweise simpel
- Dadurch, dass Elemente an beliebigen Stellen im Speicher liegen
- Dadurch kein neuanlegen der Liste notwendig
- Oder kopieren von Elementen
- Beim Einfügen werden lediglich die Referenzen der Listenelemente aktualisiert

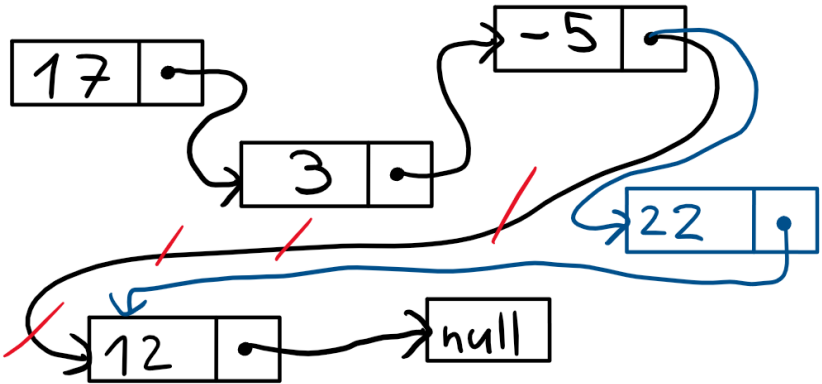
Grundoperationen

Komplexität beim Einfügen

- Komplexität auch hier $O(i)$
  - i bezeichnet hier die Position an der Liste in der eingefügt wird
  - Somit ist das Einfügen hier (anders als bei Array) von der Länge der Liste unabhängig
- Selbst ein Einfügen an letzter Position ist aber in der Regel schneller als in Arrays
  - Da keine Kopieroperationen durchgeführt werden müssen

Einfügen von Elementen

Visualisiert



Grundoperationen

Zählen von Elementen

- Anders als beim Array kann Länge alleine aus Listenelementen ermittelt werden
- Dadurch, dass das Ende der Liste über eine `null` Referenz erkennbar ist
- Vorgehen im Groben(Siehe Struktogramm auf nächster Folie):
  - Ausgehend vom `head` Element, erhöhe den Zähler solange, wie es ein `next` Element gibt
- Komplexität ist somit $O(N)$
- In der Regel wird die Länge jedoch als `int` von der Liste gespeichert und bei Hinzufügen/Entfernen aktualisiert

Zählen von Elementen

Struktogramm

Setze *curElement* = *head*
Setze *count* = 0
curElement! = *null*
curElement=curElement.next
count++
Gib *count* zurück

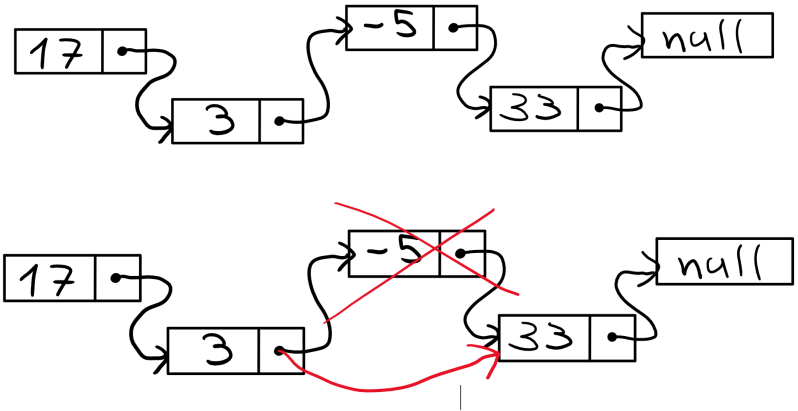
Grundoperationen

Entfernen von Elementen

- Elemente müssen nicht sequentiell im Speicher liegen
  - Dadurch entsteht keine „Lücke“ in der Liste
- Dadurch ähnlicher Vorteil gegenüber dem Array wie beim Einfügen:
  - Keine Kopieroperationen(der nachfolgenden Elemente) notwendig
- Um ein Element zu entfernen muss lediglich das `next` Element des Vorgängers auf das `next` Element des zu entfernenden Elements geändert werden
- Entferntes Element liegt dann unreferenziert im Speicher $\rightarrow$ Wird in Java durch die Garbage Collection entfernt
- Komplexität ist hier wie beim einfügen von Elementen $O(i)$

Entfernen von Elementen

Visualisiert



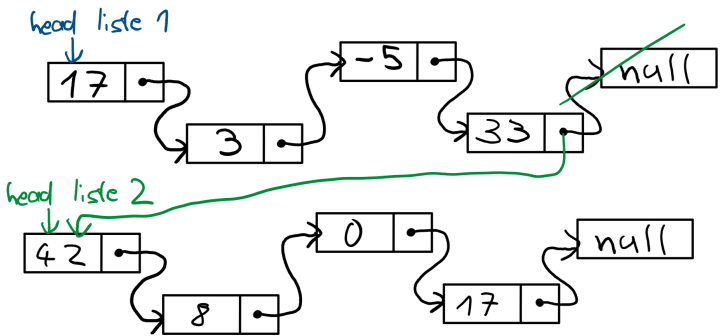
Grundoperationen

Konkatinieren von Listen

- Kombinieren von Listen ist simpel
  - Kein neuer/zusätzlicher Speicher muss reserviert werden
  - Kopieren von Elementen nicht zwingend notwendig (Je nach Implementierung jedoch sinnvoll)
- Zum konkatinieren muss lediglich die `next` Referenz des letzten Elements der ersten Liste aktualisiert werden
  - Auf das `head` Element der zweiten Liste
- Komplexität ist somit (theoretisch) $O(n)$ um bis zum letzten Element der ersten Liste zu gelangen

Konkatinieren von Listen

Visualisiert



Inhalt

Allgemeines

Zu Double Linked Lists

- Erweiterung der Linked List
- Gleiche Vorteile wie in dieser
- Jedes Element speichert nun Referenz auf Nachfolger (`next`) **und** Vorgänger(`previous`)
- Dadurch verringerte Komplexität bei einigen Operationen
- Zusätzlich kennt die Liste das erste und letzte Element (`tail`)
- Im ersten Element ist `previous` eine Referenz auf `null`

Grundoperationen

Anlegen der Liste

- Double Linked List wird als leere Liste initialisiert
- `head` und `tail` werden als leere Liste intialisiert
- Sobald Elemente hinzugefügt werden, wird `head` und `tail` automatisch aktualisiert
- Speicher für Elemente wird beim hinzufügen reserviert

Grundoperationen

Bestimmen der Größe

- Größe bestimmen funktioniert theoretisch wie in der Linked List
- Wird jedoch in der Regel parallel gespeichert und verwaltet
  - Sonst könnten einige Vorteile der Double Linked List nicht genutzt werden
  - Da jedes mal erst die Größe bestimmt werden müsste

Grundoperationen

Zugriff auf Elemente

- Ähnlich wie in der Linked List
- Jedoch kann die Anzahl der Operationen verringert werden
  - Dadurch, dass wir von beiden Enden der Liste aus traversieren können
  - Für Effizienzsteigerung muss jedoch Größe bekannt sein
- Je nachdem, ob das Element im hinteren oder vorderen Teil der Liste ist
- Wird beginnend von hinten bzw. vorn traversiert um das Element abzurufen
- Dadurch sind theoretisch maximal $\frac{n}{2}$ Schritte nötig
- Da O jedoch den konstanten Faktor $\frac{1}{2}$ ignoriert wird die Komplexität trotzdem mit $O(n)$ angegeben

Grundoperationen

Einfügen und Entfernen von Elementen

- Operationen unterscheiden sich im Grunde kaum zur Linked List
- Nur, dass jetzt in den betroffenen Elementen mehr Referenzen geupdated werden müssen
- Zusätzlich zum `next` Element jetzt noch das `previous` Element
- Wie beim Zugriff verringert sich die Anzahl der Operationen, jedoch bleibt die Komplexität $O(N)$

Grundoperationen

Konkatinieren von Listen

- Funktioniert analog wie bei Linked Lists
- Lediglich wird noch die `previous` Referenz des Kopfes der zweiten Liste aktualisiert
  - Auf das `tail` Element der ersten Liste
- Verringerte Komplexität
  - Dadurch, dass auf `tail` Element direkt zugegriffen werden kann
  - Komplexität ist somit konstant: $O(1)$
  - Teilweise speichern auch normale Linked Lists die `tail` Referenz für verringerte Komplexität beim konkatinieren

Aufgabe 1

ArrayList

Im ersten Teil haben wir über die Eigenschaften von Arrays gesprochen. Unter anderem ging es darum, dass Arrays durch ihre Struktur relativ unflexibel sind, was das hinzufügen von Elementen betrifft.
Das `Collections` Interface bietet mit der `ArrayList` Klasse eine Liste an, die im Hintergrund ein Array zum speichern von Daten anbietet. Diese Liste bietet die zwei Funktionen `size()` und `capacity` an.

1. Untersucht den Unterschied zwischen den beiden Funktionen
2. Untersucht, wie sich die Ergebnisse der beiden Funktionen verhalten, wenn Elemente hinzugefügt und entfernt werden
3. Welche Rückschlüsse lassen sich daraus über die Datenorganisation der ArrayList treffen?

Aufgabe 2

Linked List

1. Entwerft eine `LinkedListElement<T>` Klasse mit den in der Vorlesung vorgestellten Eigenschaften
2. Entwerft eine `LinkedList<T>` Klasse, die zur Speicherung der Daten die `LinkedListElement<T>` Klasse verwendet. Die Liste sollte folgende Funktionen integrieren:
   - `get(int index)` - Gibt das Element an einem bestimmten Index zurück
   - `size()` - Gibt die Anzahl der Elemente in der Liste zurück
   - `insert` - Fügt ein bestimmtes Element an einem gegebenen Index ein
   - `remove(index i)` - Entfernt das Element an dem gegebenen Index

Aufgabe 3

Double Linked List

Entwerft analog zu Aufgabe 2 die Klasse `DoubleLinkedListElement<T>` und `DoubleLinkedList<T>`

Inhalt

Allgemeines

Zu Stacks

- „Stapel“ $\rightarrow$ Beschreibt Funktionsweise sehr passend
- Listenstruktur in der nur Zugriff auf das „oberste“ Element möglich ist
- Oberstes Element ist hier das zuletzt hinzugefügte
- Daten werden intern (als private Variable) verwaltet
  - z.B. als Array oder Linked List

Allgemeines

Formale Definition Stack (Vgl. [3])

Definition Stack

Ein Stack lässt sich **rekursiv** definieren. Zu jedem Zeitpunkt ist ein Stack entweder:

- Eine leere Datenstruktur
- Bestehend aus einem obersten Element und dem Rest, wobei der Rest wieder ein Stack ist

Allgemeines

Stack Interface (Vgl. [**stacksqueues**])

- Grundlegendes Interface besteht aus zwei Operationen:
  - `push` - Fügt ein Element hinten an den Stack an
  - `pop` - Gibt das oberste Elemente des Stacks zurück und entfernt dieses aus dem Stack
- Außerdem in der Regel noch Teil des Interfaces:
  - `peek` - Gibt das oberste Element zurück, ohne es zu entfernen
  - `isEmpty`

Grundoperationen

Anlegen eines Stacks

- Stack wird als leere Datenstruktur initialisiert
- Je nachdem wie die Daten intern strukturiert sind muss ggf. eine Größe initialisiert werden
- Sobald das erste Element hinzugefügt wird, wird dieses zum obersten (`top`) Element

Grundoperationen

Zugriff auf Elemente

- Zugriff nur auf das letzte Element der Liste (Zletzt hinzugefügt)
- pop gibt das letzte Element zurück und entfernt dieses aus der Liste
- `peek` Gibt das letzte Element zurück ohne es zu entfernen
  - Wiederholte Zugriffe auf `peek` geben also immer das gleiche Ergebnis zurück (Sofern der Stack zwischenzeitlich nicht manipuliert wurde)
- Zugriff auf beliebige Elemente im Stack nicht möglich (Zuminndest sollte der Stack dafür kein Interface anbieten)

Grundoperationen

An-/Einfügen von Elementen

- Hinzufügen nur für das Ende des Stacks vorgesehen
- Dafür ist die Komplexität $O(1)$
- Hinzufügen in der Mitte des Stacks nicht vorgesehen
  - Daher sehr aufwändig
  - Es müssen von hinten sukzessiv Elemente entfernt werden, bis der gewünschte Index die letzte Position des Stacks wäre
  - Dann wird das neue Element hinzugefügt
  - Dann müssen alle zuvor entfernten Elemente wieder hinzugefügt werden
  - Diese Operation sollte aber **nicht** Teil des Stack Interfaces sein

Grundoperationen

Bestimmen der Größe

- Größe des Stacks wird in der Regel parallel verwaltet
- Und bei Aufrufen von `push` und `pop` aktualisiert
- Sonst wäre zum zählen folgendes Vorgehen nötig:
  - Kopiere den Stack
  - Entferne aus dem kopierten Stack solange Elemente bis der Stack leer ist
  - Bei jedem entfernen erhöhe einen Zähler um 1

Grundoperationen

Entfernen von Elementen

- Stack Interface bietet nur Möglichkeit, letztes Element zu entfernen
- Über die pop Funktion
- Entfernen in der Mitte vom Stack nicht möglich
  - Kann theoretisch durch einen Benutzer realisiert werden (Ähnliches Vorgehen wie beim Einfügen in der Mitte)
  - Sollte jedoch nicht Teil des Interfaces sein

Grundoperationen

Konkatinieren von Stacks

- Kombinieren von zwei Stack relativ aufwendig
- Wenn das Ende des zweiten Stacks auch das Ende des zusammengefügten Stacks sein sollte
- Denn: Bei entfernen aller Elemente vom Stack wird die Reihenfolge der Elemente invertiert
- Nötiges Vorgehen:
  - Entferne über `pop` sukzessive die Elemente vom zweiten Stack bis dieser leer ist und füge sie (über `push`) zu einem temporären Stack hinzu
  - Entferne sukzessive die Elemente vom temporären Stack und füge sie an den ersten Stack an

Anwendungsfälle

Für Stacks

- Backtracking Algorithmen - Jeder Schritt wird auf dem Stack gespeichert und kann über den Stack „zurückgegangen werden“
  - Beispiel: Labyrinth Algorithmus.
  - Jeder Schritt wird auf einem Stack gespeichert
  - Bei erreichen einer Sackgasse werden die zuletzt gemachten Schritte so lange zurück gegangen, bis ein anderer Schritt möglich ist
- Rückgängig Funktion - Jede Aktion wird auf einem Stack abgelegt und kann somit rückgängig gemacht werden

Inhalt

Allgemeines

Zu Queues (Vgl. [**stacksqueues**])

- Funktionieren ähnlich wie Stacks
- Hier jedoch andere Zugriff:
  - Elemente können nur hinten angefügt werden
  - ...und vorn entfernt werden
- Stellt eine „Warteschlange“ dar
- Interne Datenstruktur wie beim Stack variabel
- Wichtig ist, dass von außen nur über das Queue Interface zugegriffen werden kann

Allgemeines

Queue Interface (Vgl. [**stacksqueues**])

- Queue Interface besteht aus zwei grundlegenden Operationen:
  - `enqueue` - Hängt ein Element hinten an die Queue an
  - `dequeue` - Gibt das ersten Element der Queue zurück und entfernt dieses
- Wie im Stack gibt es meist noch weitere Hilfsfunktionen wie `isEmpty` oder `peek`

Grundoperationen

An-/Einfügen von Elementen

- Anfügen von Elementen nur am Ende möglich
- Einfügen in der Mitte nicht vorgesehen (In der Standardqueue)
- Um ein Element an der Stelle i einzufügen müsste man:
  - Die Größe der Queue bestimmen (n)
  - Die ersten $i-1$ Elemente der Queue entfernen und direkt hinten anfügen
  - Dann das einzufügende Element anhängen
  - Dann die vordersten $n-i+1$ Elemente entfernen und direkt hinten anfügen
- Solche Operationen sind jedoch nicht Teil des Standardinterfaces

Grundoperationen

An-/Einfügen von Elementen

- Es gibt Queues in denen neue Element nicht zwangsweise hinten angehängt werden
- Sondern nach anderen Kriterien in der Queue sortiert werden, z.B.
  - Angegebener Priorität
  - (Hash-)Wert des Elements
- Wichtig hierbei: Das Interface sollte sich hierbei nach außen nicht maßgeblich verändern
- Die Logik wie das Element in die interne Struktur eingefügt wird, bleibt intern!

Grundoperationen

Bestimmen der Größe

- Wie auch im Stack ist die Größenbestimmung sehr aufwendig
- Kopieren der Queue und sukzessives entfernen notwendig
- Daher wird auch hier die Größe meist „manuell“ verwaltet und beim Hinzufügen/Entfernen aktualisiert

Grundoperationen

Entfernen von Elementen

- Das Interface erlaubt nur das entfernen des vordersten Elements
- Um Elemente in der MItte zu entfernen müsste ähnlich vorgegangen werden wie beim Einfügen
- Auch hier gilt wieder: Das Entfernen „in der Mitte“ sollte nicht Teil des Interfaces sein

Grundoperationen

Konkatinieren von Queues

- Kombinieren von Queues ist - vergleichsweise - „simpel“
  - Im Grunde muss nur wiederholt das erste Element der zweiten Queue entfernt werden
  - ...und direkt an die erste angefügt werden
- Je nachdem wie die Queue intern strukturiert ist, kann dies über eine bereitgestellte „append“ Funktion optimiert werden
  - Zum Beispiel durch direktes zusammenfügen der internen Linked Lists

Anwendungsfälle

Für Queues

- Scheduling von nacheinander ablaufenden Operationen
- Planung von Prozessen in gemeinsam genutzten Ressourcen
  - Wenn eine Ressource gerade belegt ist und ein weterer Prozess sie aber benötigt
  - Wird der Prozess zur nächsten Berechnung in der Queue gespeichert

Inhalt

Allgemeines

Baumstrukturen (Vgl. [3])

- Unterscheiden sich von den bisherigen Strukturen
- Bisher hatte jedes Element einer Datenstruktur immer einen „Nachfolger“
- ...und einen Vorgänger
- Bäume können jedoch mehrere „Nachfolger“ haben
  - Man spricht in der Regel von **Knoten**
  - Es gibt genau einen Knoten im Baum, der keinen Eingang („Vorgänger“) besitzt
    - Dies ist der **Wurzelknoten** des Baumes
  - Alle anderen Knoten haben genau einen Eingang
- In der Regel beschäftigt man sich hauptsächlich mit *Binärbäumen*

Bäume

Graphentheoretische Definition (Vgl. [3])

Graphentheoretische Definition

Ein Baum ist ein endlicher, schwach zusammenhängender gerichteter Graph, für dessen Knotenpunkte gilt:

1. Es gibt genau einen Knoten, der keinen Eingang hat (Wurzel des Baumes)
2. Alle übrigen Knotenpunkte haben genau einen Eingang.

Allgemeines

Rekursive Definition (Vgl. [3])

Rekursive Definition(Binärbaum)

Eine Baumstruktur vom Grundtyp T ist:

1. Eine leere Struktur
2. Ein Knoten vom Typ T mit genau zwei disjunkten Teilbäume vom Grundtyp T

Allgemeines

Struktur der Knoten im Baum (Vgl. [3])

- Grundsätzlich besteht ein Knoten(Im Binärbaum) aus drei Elementen
  - Einem Schlüssel (=Datenwert) für den Knoten
  - Einen linken Teilbaum
  - Einen rechten Teilbaum

Eigenschaften

Vollständige Binärbäume (Vgl. [3])

Vollständige Binärbäume

In einem Binärbaum kann jede Ebene maximal $2^{(Tiefe-1)}$ Knoten besitzen. Ein Binärbaum ist dann vollständig ausgeglichen, wenn auf jeder Ebene, mit Ausnahme der letzten, genau diese $2^{(Tiefe-1)}$ Knoten existieren und die Knoten auf der letzten Ebene soweit links wie möglich stehen

Eigenschaften

Suchbäume (Vgl. [3])

Suchbaum

Wenn für alle Knoten K eines Baumes gilt, dass...

1. ...alle Schlüssel im linken Teilbaum von K kleiner...
2. ...alle Schlüssel im rechten Teilbaum von K größer...

... als der Schlüssel des Knotens K sind, so handelt es sich um einen Suchbaum

Grundoperationen

In Binärbäumen

- Zur Vereinfachung betrachten wir für Binärbäume (Suchbäume) folgende Operationen:
  - Einfügen von Elementen
  - Entfernen von Elementen
  - Bestimmen der Anzahl von Elementen
- Grundoperationen basieren fast immer auf rekursivem Aufruf auf den Teilbäumen!

Grundoperationen

Einfügen von Elementen (Vgl. [3])

- Für das Einfügen eines Elements X in einen Baum B werden vier Fälle unterschieden:
  - Ist B leer, so ist das Ergebnis der Baum mit dem Schlüssel X
  - Ist X identisch mit dem Schlüssel von B so ist X bereits im Baum und wird nicht hinzugefügt
  - Ist X kleiner als der Schlüssel von B so füge X im linken Teilbaum ein
  - Ist X größer als der Schlüssel von B so füge X im rechten Teilbaum ein

Grundoperationen

Entfernen aus Suchbäumen

- Beim Entfernen eines Elements mit dem Schlüssel X können vier Fälle auftreten:
  - Der Baum enthält X nicht - Fertig.
  - Der Knoten mit dem Schlüssel X hat keinen Nachfolger - X wird entfernt, Fertig.
  - Der Knoten mit dem Schlüssel X hat genau einen Nachfolger
  - Der Knoten mit dem Schlüssel X hat genau zwei Nachfolger

Entfernen

X hat einen Nachfolger (Vgl. [2], [4])

- Simpler Fall
- Hier „rutscht“ der Nachfolger (Gesamter Teilbaum) einfach an die Stelle des zu entfernenden Elements
- Keine weiteren Operationen - Fertig.

Entfernen

X hat zwei Nachfolger (Vgl. [2], [4])

- Problematik: Welcher Nachfolger tritt an die Stelle von X
- ...damit die Suchbaumbedingung bestehen bleibt?
- Suchen des sogenannten **in-order Nachbarn** N von X
  - Ist das jeweils nächstgrößere bzw. -kleinere Element an X
  - Also das kleinste („linkeste“) Element des rechten Teilbaums
  - Oder größtes („rechteste“) Element des linken Teilbaums
- N tritt an die Stelle von X
- Teilbäume von X bleiben (beinahe) unervändert
  - Außer natürlich, dass N aus dem Teilbaum entfernt wird

Grundoperationen

Bestimmen der Anzahl der Elemente

- Wie in allen Strukturen kann die Anzahl der Elemente natürlich manuell verwaltet werden
- Allerdings ist das Zählen auch über eine rekursive Definition möglich
- Hierbei Unterscheidung zwischen zwei Fällen:
  - Ist der Baum B die leere Struktur, so ist die Größe 0
  - Sonst gilt: $size(B) = size(L) + size(R) + 1$, wobei L und R den linken bzw. rechten Teilbaum bezeichnen

Ausgeglichenheit

Das Ying und Yang von Bäumen

- Bisher können unsere (Teil-)Bäume verschiedene Tiefen haben
- Dies ist jedoch nicht optimal für z.B. Suchalgorithmen
- Die maximale Suchzeit lässt sich so nur schwer abschätzen
- Außerdem ist die durchschnittliche Suchzeit meist höher
- Daher:
  - Definition eines Ausgeglichenheitskriteriums
  - Dadurch Tiefe der Teilbäume ähnlich
  - Führt zu besserer Performance bei Suchbäumen

AVL Bäume

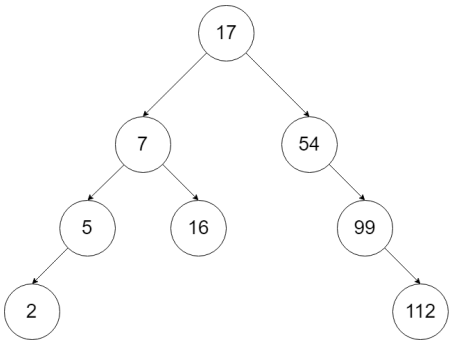
Definition (Vgl. [3], [5])

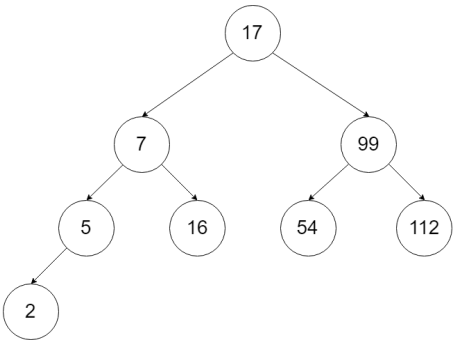
Ausgeglichenheit nach AVL-Bedingung

Ein Baum ist dann ausgeglichen, wenn sich für jeden Knoten die Höhe der zugehörigen Teilbäume um höchstens 1 unterscheidet.

AVL Bäume

Visualisiert





Einfügen

In AVL Bäume (Vgl. [5], [6])

- Einfügen erfolgt vorerst nach normalem Verfahren von binären Suchbäumen
- Dadurch kann jedoch die AVL-Bedingung verletzt werden
- Dann muss der Baum wieder ausgeglichen werden
- Hierfür wird folgendermaßen vorgegangen:
  - Wandere beginnend vo eingefügten Element w nach oben
  - Prüfe für jeden Knoten ob die AVL Bedingung verletzt ist
  - Wenn ein Knoten z gefunden wurde, für den die AVL-Bedingung verletzt wurde, gleiche den Baum aus

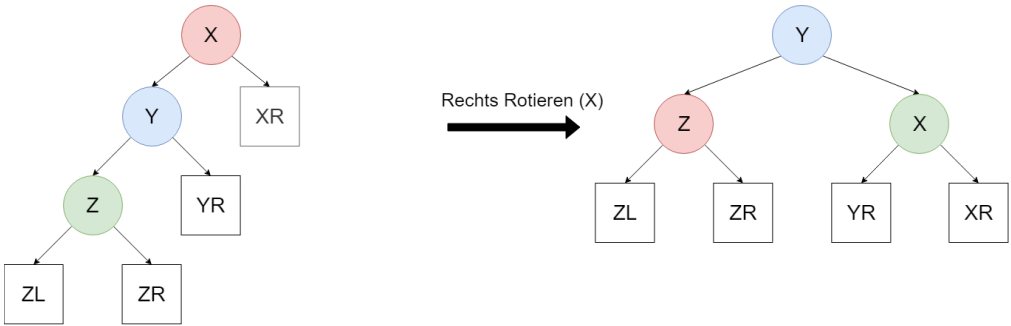
Einfügen

Ausgleichen von AVL Bäumen (vgl. [1], [6])

- Zum Ausgleichen von AVL-Bäumen gibt des die Links- bzw. Rechtsrotation
- Für einen unbalancierten Knoten x sei:
  - y Der „Kindknoten“ (Erste schritt) von x auf dem Weg zum eingefügten Knoten w
  - z Der „Enkelknoten“ (Zweite Schritt) von x auf dem Weg zu w
- Hier gibt es vier mögliche Fälle nach denen unterschiedlich rotiert werden muss:
  - Links-Links
  - Links-Rechts
  - Rechts-Rechts
  - Rechts-Links

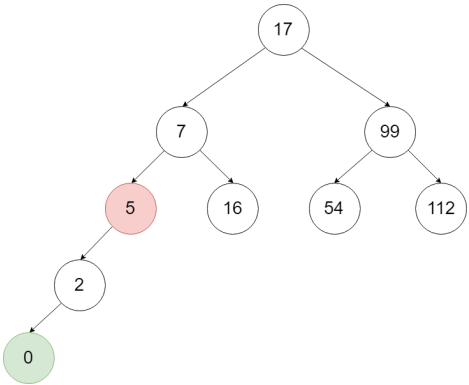
Links-Links

Rechtsrotation (Vgl. [1])



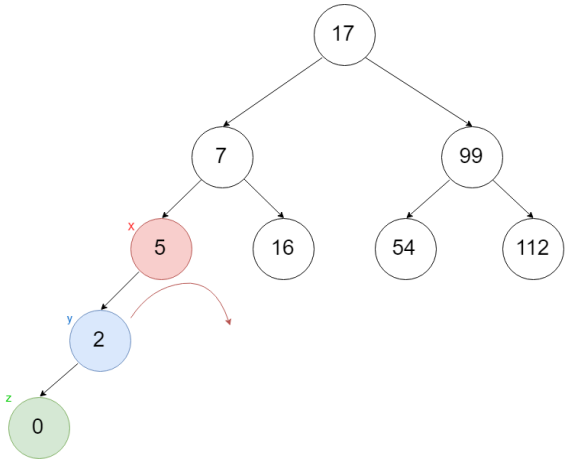
Links-Links I

Konkretes Beispiel



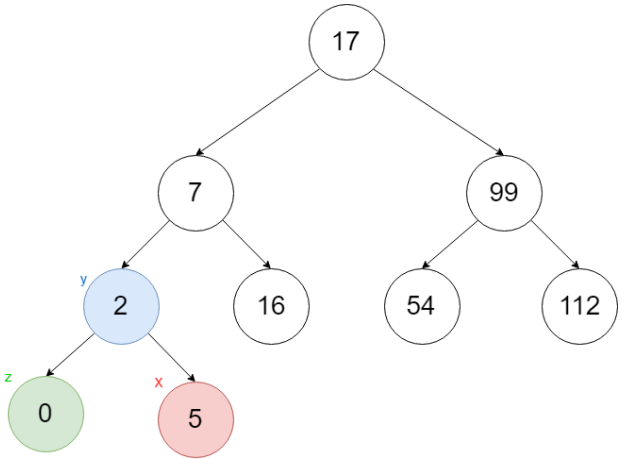
Links-Links II

Konkretes Beispiel



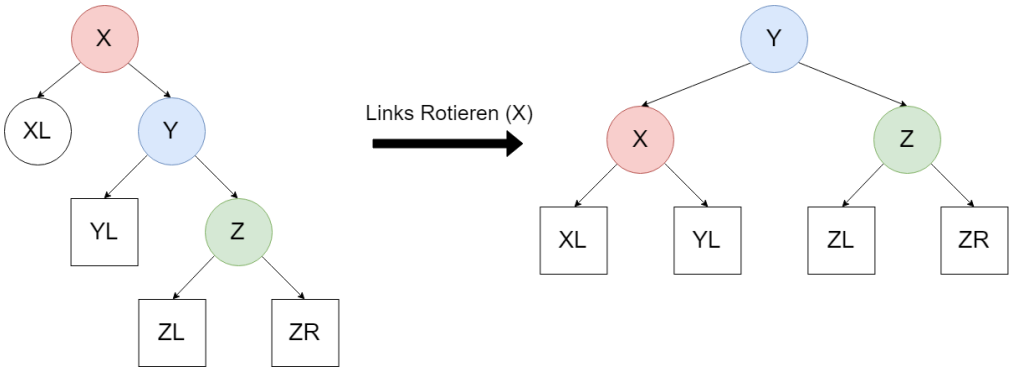
Links-Links III

Konkretes Beispiel



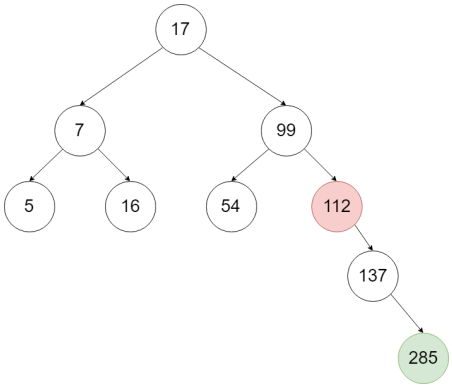
Rechts-Rechts

Linksrotation(Vgl. [1])



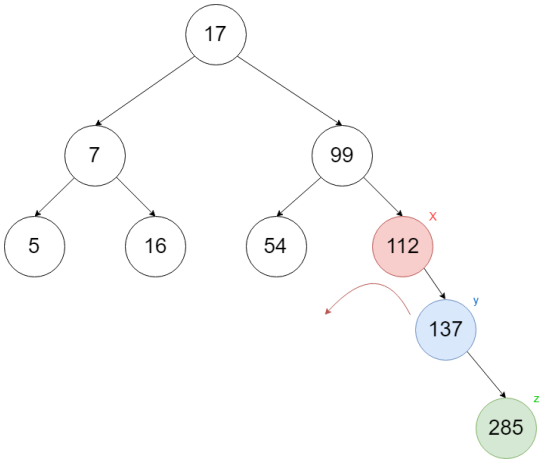
Rechts-Rechts I

Konkretes Beispiel



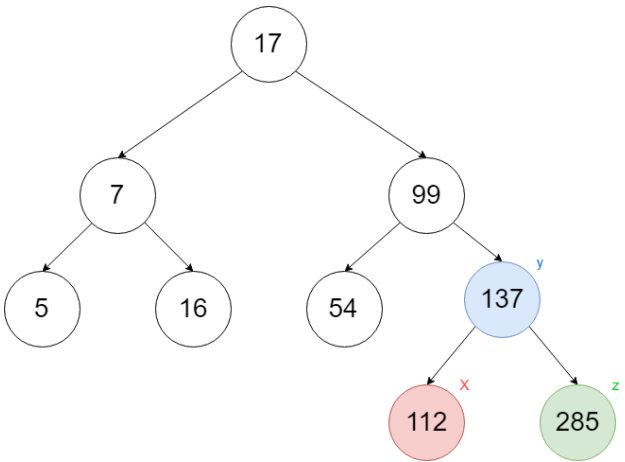
Rechts-Rechts II

Konkretes Beispiel



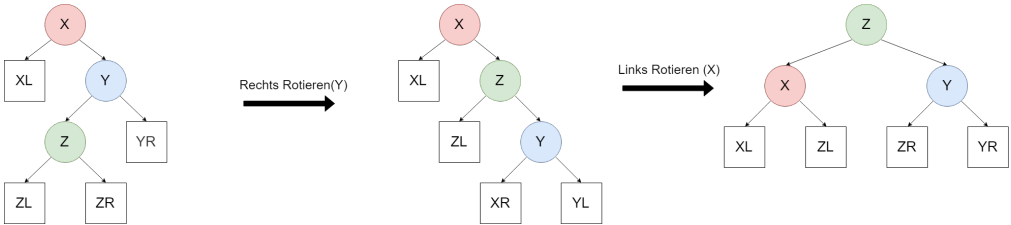
Rechts-Rechts III

Konkretes Beispiel



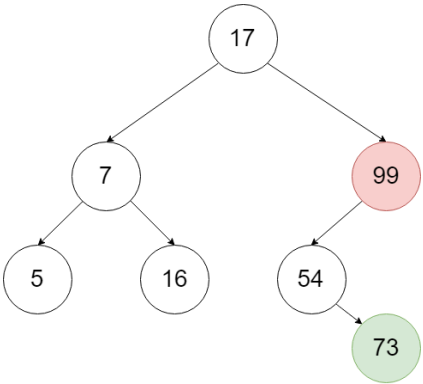
Links-Rechts

Links-Rechts-Doppelrotation(Vgl. [1])



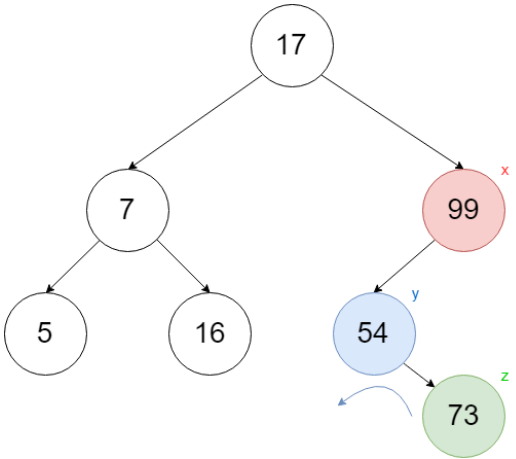
Links-Rechts I

Konkretes Beispiel



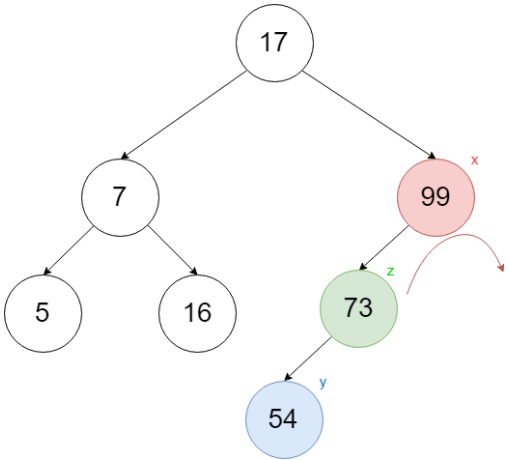
Links-Rechts II

Konkretes Beispiel



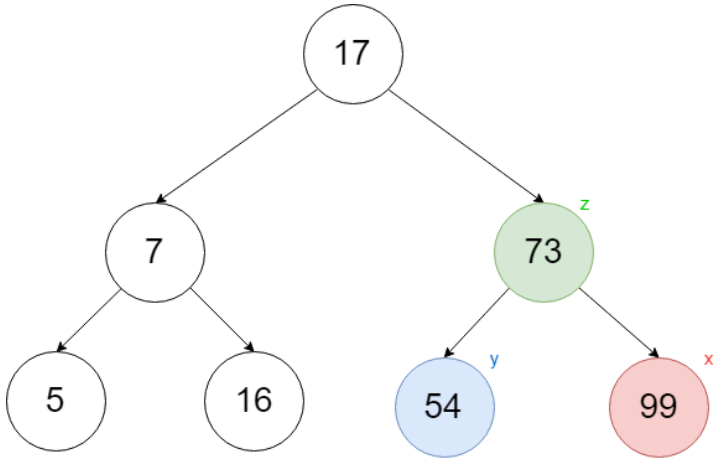
Links-Rechts III

Konkretes Beispiel



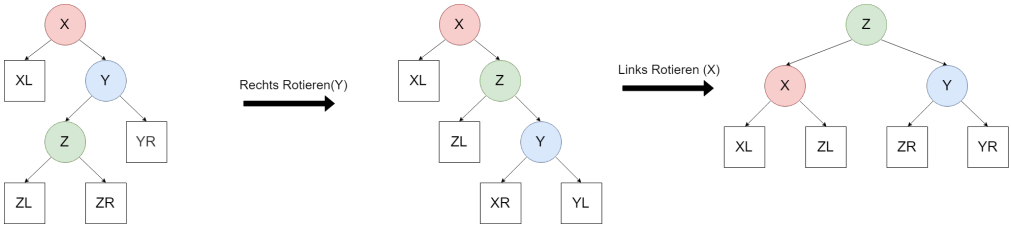
Links-Rechts IV

Konkretes Beispiel



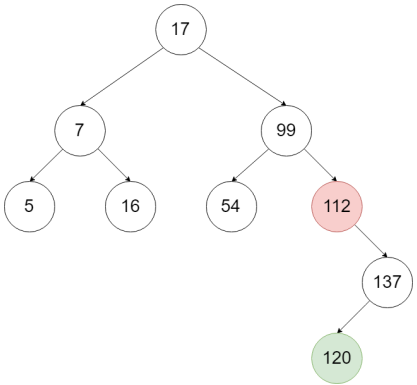
Rechts-Links

Rechts-Links-Doppelrotation (Vgl. [1])



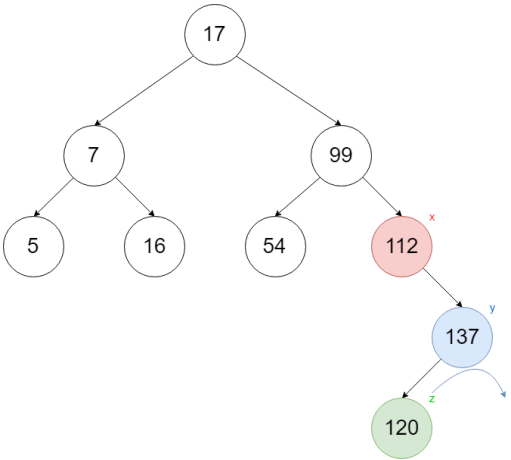
Rechts-Links I

Konkretes Beispiel



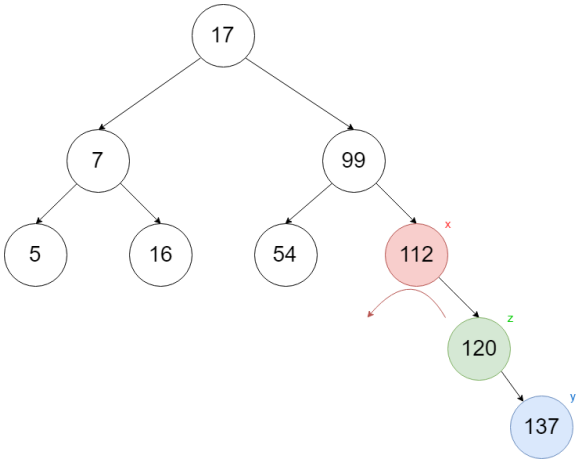
Rechts-Links II

Konkretes Beispiel



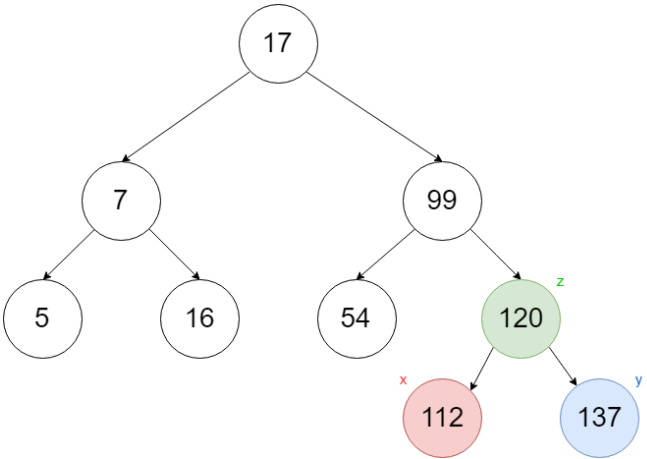
Rechts-Links III

Konkretes Beispiel



Rechts-Links IV

Konkretes Beispiel



Aufgabe 1

In der Vorlesung haben wir darüber gesprochen, dass die interne Datenstruktur von Queues und Stacks frei gewählt werden kann. Soo könnte beispielsweise ein Array oder auch eine Linked List verwendet werden.

- Welche Vorteile ergeben sich aus dieser Austauschbarkeit der Datenstruktur?
- Ist die Verwendung eines simplen Arrays als Datenstruktur für eine Queue bzw. einen Stack sinnvoll?
  - Welche Vor- und Nachteile der Array Struktur sollte man berücksichtigen?

Aufgabe 2

Entwerft und implementiert eine Klasse zum repräsentieren eines Stacks oder einer Queue. Die interne Datenstruktur kann von euch frei gewählt werden. Die Klasse sollte mindestens die folgenden Methoden zur Verfügung stellen:

- `isEmpty` - Gibt zurück, ob die Struktur leer ist
- `peek` - Gibt das vorderste (für Queues) bzw. letzte (für Stacks) zurück
- `clear` - Entfernt alle Elemente aus der Datenstruktur
- `size` - Gibt die Anzahl der Elemente zurück
- Sowie die entsprechenden Operationen zum hinzufügen bzw. entfernen

Aufgabe 3

In der Vorlesung haben wir über binäre Suchbäume gesprochen.

- Zeichnet den entstehenden Baum, wenn ihr nacheinander die Elemente $(47, 63, 25, 13, 57, 18, 99, 14, 22, 37, 24, 28, 30,)$ hinzufügt.
- Wie sieht der resultierende Baum aus, wenn ihr das Element 13 entfernt?
- Wie sieht der Baum aus, wenn ihr weiterhin das Element 25 entfernt?

Aufgabe 4

Mithilfe von ausgeglichenen Bäumen kann die Suchzeit in Binärbäumen optimiert werden. Eine Möglichkeit zur Erstellung eines ausgeglichenen Baumes stellt die AVL Bedingung dar. Durch diese wird sichergestellt, dass sich die Tiefen der Teilbäume zu jedem Zeitpunkt um maximal 1 unterscheiden.

- Fügt in der gegebenen Reihenfolge die Elemente in einen intial leeren Binärbaum ein: $3, 2, 1, 4, 5, 6, 7, 16, 15$

Quellen I

[1] *AVL Tree*. URL: https://www.geeksforgeeks.org/avl-tree-set-1-insertion/ (besucht am 19. 04. 2019).

[2] DU. *Einen Knoten aus einem Binärbaum löschen*. 2018. URL: https://www.inf-schule.de/algorithmen/suchbaeume/binaerbaum/andere/loeschen (besucht am 19. 04. 2019).

[3] Prof. E. Fahr. *Theoretissche Informatik II - Datenstrukturen*. 2016.

[4] Ulrich Helmich. *Das Löschen von Elementen*. 2017. URL: http://u-helmich.de/inf/BlueJ/kurs121/folge17/folge17-8.html (besucht am 19. 04. 2019).

Quellen II

[5] Petra Mutzel. *Binäre Suchbäume*. 2009. URL: http://ls11-www.cs.tu-dortmund.de/people/beume/dap2-09/folien/12-AVLTree.pdf (besucht am 19. 04. 2019).

[6] Wikiversity. *Kurs:Algorithmen und Datenstrukturen/Vorlesung/AVL Bäume — Wikiversity*, 2016. URL: https://de.wikiversity.org/w/index.php?title=Kurs:Algorithmen_und_Datenstrukturen/Vorlesung/AVL_B%C3%83%C2%A4ume&oldid=477454 (besucht am 19. 04. 2019).

Kontakt

- E-Mail: `lukas.abelt@airbus.com`
- GitHub: `https://www.github.com/LuAbelt`
- GitLab: `https://www.gitlab.com/LuAbelt`
- Telefon(Firma): 07545 - 8 8895
- Telegram: LuAbelt