

ADP Aufgabe 2

2.A

Lernziele

1. Umgang mit verketteten Strukturen vertiefen.
2. Umgang mit abstrakten Datentypen durch Erweiterung lernen.
3. Die Basis für die Implementierung von Listen in Java kennenlernen.
4. Das Laufzeitverhalten eines Algorithmus aus einer doppelt logarithmischen Darstellung hochrechnen sowie den Verdoppelungstest anwenden können.
5. Ursachen für schlechtes Laufzeitverhalten identifizieren und beheben können.

2.A1

Implementieren Sie eine doppelt verkettete Liste als generische Klasse `DoublyLinkedList` mit Wächterknoten für Anfang und Ende. Gehen Sie folgt vor.

1. Leiten Sie von `AbstractList` ab und implementieren Sie **nur die absolut notwendigen** Methoden. Das sind natürlich alle abstrakten Methoden, aber auch alle Methoden, die aus der doppelt verketteten Liste eine modifizierbare Liste machen (Hinzufügen, Löschen, Ändern von Elementen). Studieren Sie die Implementierung von `AbstractList` und identifizieren die **minimale** Menge an Methoden, die Sie überschreiben müssen. **Hinweis:** Iteratoren müssen Sie nicht implementieren. **In A1 geht es nicht um Zeit-Optimierung sondern nur um Funktionalität!**
2. Verwenden Sie eine private innere Klasse `Node` analog der Implementierung von `Stack` etc., und ergänzen Sie die Klasse `Node`, so dass eine doppelte Verkettung entstehen kann.
3. Zu den Wächterknoten (engl. guards): Wächterknoten sind Knoten ohne Inhalt (Inhalt ist `null`), die nur dazu dienen Anfang und Ende einer Liste zu markieren. Zu Beginn besteht die leere Liste nur aus zwei Wächterknoten, einem für `head` und einem für `tail`. Der Nachfolger von `head` ist `tail` und der Vorgänger von `tail` ist `head`. Wächterknoten haben den Vorteil, dass Operationen am Anfang und Ende der Liste nicht gesondert behandelt werden müssen. Das Kopieren und Adaptieren der Implementierung von `LinkedList` der Java Bibliothek ist nicht erlaubt.
4. Implementieren Sie auch den verallgemeinerten Kopier-Konstruktor für Collections.
5. Testen Sie Ihre Implementierung wie folgt:
 - a. Fügen Sie Elemente am Ende, am Anfang und in der Mitte der Liste ein.
 - b. Ändern Sie einen Wert an einem Index.
 - c. Löschen Sie Elemente am Anfang, am Ende und in der Mitte der Liste.
 - d. Löschen Sie die Liste vollständig (mit `clear`).
 - e. Iterieren Sie über die Liste und geben Sie die enthaltenen Elemente aus.
 - f. Erzeugen Sie einen Stream auf der Liste und geben Sie die enthaltenen Elemente aus.
 - g. Wandeln Sie den Inhalt der Liste in ein Array um.
 - h. Wandeln Sie eine Menge und eine `ArrayList` in eine `DoublyLinkedList` um. (Dazu benötigen wir den verallgemeinerten Kopier-Konstruktor!)

2.A2

1. Führen Sie mit der Implementierung von `DoublyLinkedList` einen **Verdoppelungstest mit Gewichtung der Zeiten** durch. In dem Verdoppelungstest soll die Zeit, die für das Iterieren über die gesamte Liste benötigt wird, gemessen werden. Lassen Sie den Test arbeiten bis sich die gewichteten Zeiten ungefähr einpendeln. Ein Beispiel für das Gerüst eines Verdoppelungstests finden Sie im Skript.

2. Tragen Sie die Messungen in einer doppelt logarithmischen Darstellung ein und bestimmen Sie die Steigung (näherungsweise). Die Steigung sei b .
3. Berechnen Sie die Größe a der Formel $T(N) = a * N^b$ und rechnen Sie für den darauffolgenden Verdoppelungsschritt die Laufzeit aus. Vergleichen Sie diese mit der tatsächlich gemessenen Zeit.
4. Führen Sie denselben Test mit der `LinkedList` von Java durch.
5. Identifizieren Sie die Ursache für das schlechte Laufzeitverhalten Ihrer Implementierung (mit Begründung) und verbessern Sie die Implementierung.

2.B Zusammenhangskomponenten

Lernziele

1. Varianten der Algorithmen für die Lösung des Union-Find-Problems anwenden.
2. Laufzeitverhalten unterschiedlicher Implementierungen analysieren.
3. Abhängigkeit des Laufzeitverhaltens von der Eingabe verstehen.

Aufgabenstellung

Sie bekommen die folgende Eingabe (Abfolge von Paaren) für ein Union-Find-Problem mit anfänglich $N = 10$ Komponenten.

```
0 1
2 1
1 3
6 9
4 7
5 8
1 5
1 8
9 1
8 2
8 3
8 4
```

1. Zeichnen Sie für den Quick-Union-Algorithmus nach jedem `union`-Schritt den Inhalt des `id[]` Arrays. Zeichnen Sie auch den entsprechenden Wald von Bäumen. ✓
2. Wie ist die Wachstumsordnung des Quick-Union-Algorithmus, wenn am Ende alle Komponenten in einer Zusammenhangskomponente liegen sollen? ✓
3. Zeichnen Sie für den Weighted-Quick-Union-Algorithmus nach jedem `union` Schritt den Inhalt des `id[]` und des `sz[]` Arrays. ✓
4. Wie ist die Wachstumsordnung des Weighted-Quick-Union-Algorithmus, wenn am Ende alle Komponenten in einer Zusammenhangskomponente liegen sollen? ✓
5. Implementieren Sie Pfadkompression für den Quick-Union-Algorithmus und vergleichen Sie die Laufzeiten von Quick-Union und Quick-Union mit Pfadkompression mit Hilfe eines Verdoppelungstests. ✓
6. Welches Kostenmodell wird bei der Bestimmung der Wachstumsordnung der Union-Find Algorithmen verwendet? ✓
7. Geben Sie für alle Union-Find Algorithmen Beispiele für eine beste und eine schlechteste Eingabe.

2.C Speicherbedarf

Berechnen Sie für Integer, Date, Counter, int[], double[], double[][], String, Node und Stack (beide Implementierungen) den Speicherbedarf auf einem fiktiven 48 Bit Rechner. Gehen Sie davon aus, dass die Referenzen 6 Byte und der Objekt-Overhead 12 Byte belegen, dass auf ein Vielfaches von 6 Byte aufgefüllt wird und die Klasse Node als statische innere Klasse implementiert ist.

- Umgesehen
- Fixed Size Array
 - Resize Array
 - Linked List

2.D Insertion- und Selection-Sort

1. Warum ist die Wachstumsordnung für Selection-Sort immer quadratisch? Wieviel Tauschoperationen benötigt Selection-Sort im besten Fall? $\rightarrow \text{immer } N^2, \text{ egal welcher Fall}$
2. Im Skript finden Sie die Behauptung: **Für ein Array mit N Elementen unterscheidet sich die Laufzeit von Insertion- und Selection-Sort nur um einen kleinen Faktor.** Finden Sie Argumente, die die Behauptung stützen und berücksichtigen Sie dabei die Aussagen der Sätze zur Anzahl der Vergleichs- und Tauschoperationen für beide Algorithmen.
3. Gegeben die Eingabe V I R E N B E F A L L. Fertigen Sie für Selection- oder Insertion-Sort ein Handprotokoll der ersten 8 Sortierdurchläufe an. Markieren Sie das Element, dass aktuell betrachtet wird, die Bereiche die sich ändern und die Bereiche, die nicht betrachtet werden. Geben Sie auch die Werte der typischen Variablen des Sortierverfahrens an.

2.D.1.

```
public class Selection {
    public static <T extends Comparable<? super T>
        int N = a.length;
        for (int i = 0; i < N; i++) { → Für jedes Element → N mal
            int min = i;
            for (int j = i + 1; j < N; j++) { → vergleiche mit allen verbliebenen
                if (less(a[j], a[min])) min = j;
            }
            exch(a, i, min); → Tausche immer, auch mit gleichwertigen Elementen → immer  $N^2$ -Tausch Operationen.
        }
    }
}
```

2.D.2

Selection Sort: Vergleiche: $\frac{N(N-1)}{2} \approx \frac{N^2}{2} \sim N^2$
 Tauschop.: immer N^2

Insertion Sort:

Worst Case	Normal Case	Best Case
$\frac{N(N-1)}{2}$ Vergleiche	$\sim \frac{N^2}{4}$ Vergleiche	$N-1$ Vergleiche
$\frac{N(N-1)}{2}$ Tauschop.	$\sim \frac{N^2}{4}$ Tauschop.	$\Theta(n^2)$ Tauschop.

Verhältnis Selection zu Insertion sort:

	Worst Case	Normal Case	Best Case
Vergleiche	$\frac{N(N-1)}{2} / \frac{N(N-1)}{2} = 1 \rightarrow \text{klein}$	$\frac{N(N-1)}{2} / \frac{N^2}{4} \approx \frac{N^2}{2} \cdot \frac{4}{N^2} \approx 2 \rightarrow \text{Faktor 2} \rightarrow \text{"klein"}$	$\frac{N(N-1)}{2} / (N-1) = \frac{N}{2} \rightarrow \text{für kleine } N \rightarrow \text{kleiner Faktor}$ $\rightarrow \lim_{N \rightarrow \infty} \frac{N}{2} = \infty$
Tausch Operationen	$\frac{N^2}{4} / \frac{N(N-1)}{2} \approx \frac{N}{2} \rightarrow \text{Faktor: } \frac{1}{N} \rightarrow \lim_{N \rightarrow \infty} \frac{1}{N} = 0 \rightarrow \text{Sehr klein}$	$\frac{N^2}{4} = N \cdot \frac{4}{N^2} = \frac{4}{N} \rightarrow \lim_{N \rightarrow \infty} \frac{4}{N} = 0 \rightarrow \text{kleiner Faktor}$	$N/Q \rightarrow \text{Insertion Sort ist quasi "unendlich" viel besser als Selection Sort, weil kein Um-Tauschen erforderlich ist.}$

Zu Aufgabe 2 B 1

Quick-Union Tabelle & Graph

Quick Union - Übung

p	q	Id	0	1	2	3	4	5	6	7	8	9
		Start	0	1	2	3	4	5	6	7	8	9
0	1		1	-1	-2	-3	-4	5	6	7	8	9
2	1		1	1	1	3	4	5	6	7	8	9
1	3		1	3	1	3	4	5	6	7	8	9
6	9		1	3	1	3	4	5	3	7	8	9
4	7		1	3	1	3	7	5	3	7	8	9
5	8		1	3	1	3	7	8	9	7	8	9
1	5		1	3	1	8	7	8	9	7	8	9
1	8		1	3	1	8	7	8	9	7	8	9
9	1		1	3	1	8	7	8	9	7	8	8
8	2		1	3	1	8	7	8	9	7	8	8
8	3		1	3	1	8	7	8	9	7	8	8
8	4		1	3	1	8	7	8	9	7	7	8

Quicks Union:

- Finde die Wurzeln der Knoten p, q
- Hänge die Wurzel von p unter die Wurzel von q

Quick Union ~ Walder

$p = q$
0-1:

2-1:

1-3:

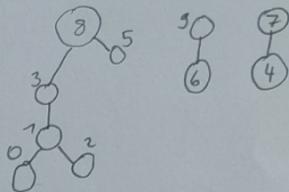
6-9:

4-7:

5-8:

1-5:

1-8: (nix zutun)

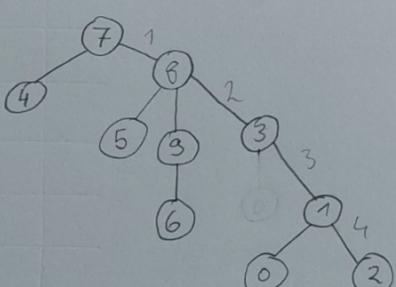


9-1:

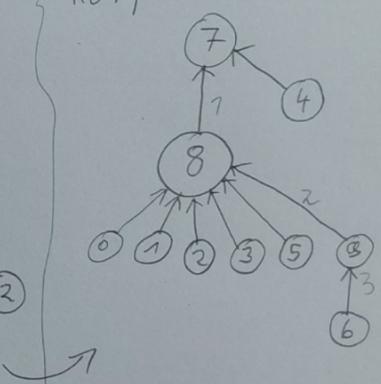
8-2: nix zutun

8-3: nix zutun

8-4:



Komprimierter Baum



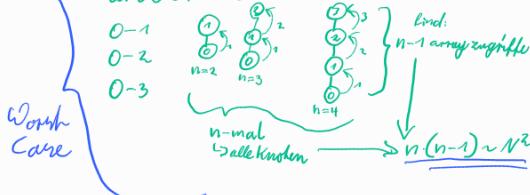
Kompression
im Kind.

Zu 2.B.2:

→ Wachstumsordnung:

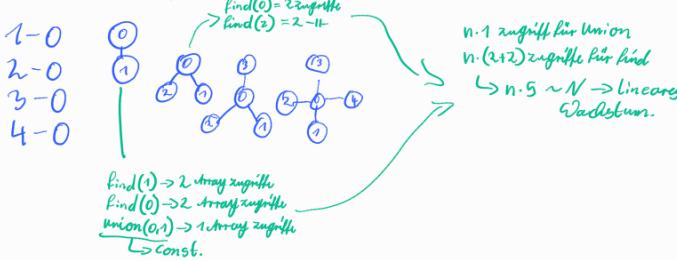
↳ Quick Union muss nicht immer alle Knoten/Elemente im 1D-Array durchlaufen sondern nur Teile Bäume

↳ Ausnahme Ketten und man fügt am Blatt an = Quadratisches Wachstum



Best Case:

↳ Kette, & man hängt an der Wurzel an oder Stern



Zu 2.B.3

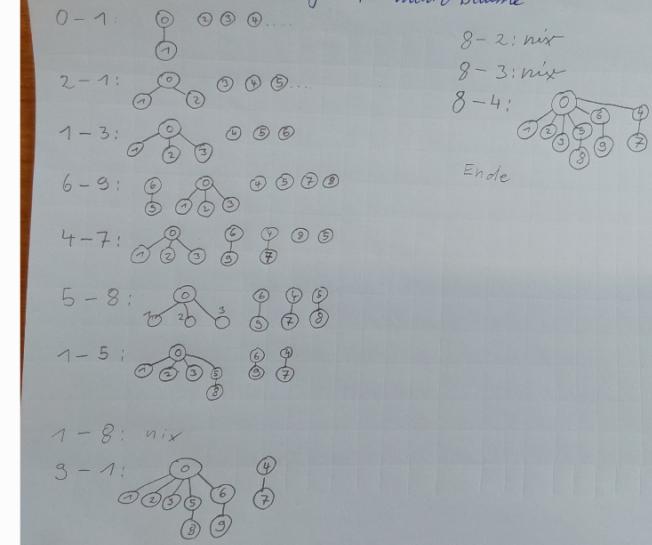
Weighted Quick Union

ADP 02 Weighted Quick Union																							
p	q	id	0	1	2	3	4	5	6	7	8	9	sz	0	1	2	3	4	5	6	7	8	9
1	1	0	0	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	1
2	0	1	0	0	2	3	4	5	6	7	8	9	2	1	1	1	1	1	1	1	1	1	1
3	2	1	0	0	0	3	4	5	6	7	8	9	3	1	1	1	1	1	1	1	1	1	1
4	1	3	0	0	0	0	4	5	6	7	8	9	4	1	1	1	1	1	1	1	1	1	1
5	6	3	0	0	0	0	4	5	6	7	8	9	5	4	1	1	1	1	1	1	1	1	1
6	4	7	0	0	0	0	4	5	6	4	8	6	6	4	1	1	1	2	1	1	1	1	1
7	5	8	0	0	0	0	4	5	6	4	5	6	6	4	1	1	1	2	2	1	1	1	1
8	1	5	0	0	0	0	4	0	6	4	5	6	6	1	1	1	2	2	2	1	1	1	1
9	1	8	nix	0	0	0	0	4	0	6	4	5	6	6	1	1	1	2	2	2	1	1	1
10	3	1	0	0	0	0	4	0	0	4	5	6	6	8	1	1	1	2	2	2	1	1	1
11	8	2	nix	0	0	0	0	4	0	0	4	5	6	8	1	1	1	2	2	2	1	1	1
12	8	3	nix	0	0	0	0	4	0	0	4	5	6	8	1	1	1	2	2	2	1	1	1
8	4		0	0	0	0	0	0	0	4	5	6	10	1	1	1	2	2	2	1	1	1	1

Quick Union

```
find(p){  
    node=p  
    while(id[p] != p){  
        node=find(node)  
    }  
    return node  
}  
  
union(p,q){  
    proot=find(p)  
    qroot=find(q)  
    if(proot!=qroot){  
        id[proot]=qroot  
        count--  
    }  
}
```

ADP 02 Quick Union Weighted - Wald & Bäume



Zu 2.B.4 Wachstumsordnung

find(p){

if (id[p] != p){ → 1 Zugriff + 1 Vergleich
 p = find(id[p]) } → 1 Zugriff + 1 Wiederholung

return p

union(p, q)

if (idp = find(p)) } 2 mal Find aufrufe

idq = find(q)

if (idp == idq) return; → 1 Vergleich

if (sz(idp) < sz(idq)){ → 2 Zugriffe + 1 Vergleich

then: id[idp] = idq → 1 Zugriff

sz(idp) += sz(idq) } → 2 Zugriffe + 1 Op.

else{

id[idq] = idp → 1 Zugriff

sz(idp) += sz(idq) } → 2 Zugriffe + 1 Op.

counter --; → 1 Op.

Kir n Knoten → n-1 aufrufe

↳ Script: Tiefe eines Knoten ist höchsten $\log_2 N \rightarrow$ damit höchsten.

$\log_2 N$ Aufrufe für find(p) Script 1.6 Seite 41

↳ Wachstumsordnung:

$\sim N \cdot \log_2 N$

Zu 2.B.6: Union find - Wachstum & Kosten

Quick Find!

find(p):
return id[p]

union(p, q)

idp = find(p) - Arrayzugriff
idq = find(q) - Arrayzugriff
if(idp != idq) - Vergleich

for(i=0; i < id.length; i++) - Schleife!

if(id[i] == idp) - Zugriff & Vergleich

then id[i] = idp - Zugriff

count--

- Arithmetische Operation const=1

Kostenmodell: Array-zugriff (willkürliche Operation in Schleife)

Wachstumsordnung: N^2

Alle Knoten in eine Komponente
 \hookrightarrow mal union aufheben

Union: müssen alle vorhanden
Knoten überlaufen $\rightarrow n$

$$(n-1) \cdot (3+1+2 \cdot n) = 2n^2 - 2n + 4n - 4 = 2n^2 + 2n - 4$$

$\sim N^2$

Zu Aufgabe 2.C - Speicherbedarf

AD-Speicherbedarf (Praktikum 2.C)

Maschinen Adressen auf 48Bit Architektur
 $\Rightarrow 6\text{ Byte benötigt für eine Adresse}$

Integer: Braucht 12 Byte (Objekt overhead)
 $+ 4\text{ Byte (int-Wert)}$

$= 16\text{ Byte Daten}$

\rightarrow Auffüllen auf Vielfache von 6: 2Byte Füllung

Integer $\rightarrow \sum = 18\text{ Byte}$

Date: 12 Byte Overhead

$3 \times 4\text{ Byte} : d, m, y$

$= 12 + 12 = 24$

\rightarrow Vielfache von 6: 0Byte Füllung

Date: $\sum = 24\text{ Byte}$

Counter: 12 Byte Overhead

6 Byte Stringref

4 Byte Int-Wert

$= 22\text{ Byte}$

\rightarrow Vielfache von 6: 2Byte Füllung

Counter: $\sum = 24\text{ Byte}$

int[]: 12 Byte Overhead	4 Byte: int size/N	6
	+ 2 Byte: Füllung	2 12
18 -	+ N/4 Byte für ints	12 18
	+ 2 Byte Füllung	16 24
	oder 4 Byte Füllung	20 30
	$\Sigma = (24 + N/4) \text{ Byte} + 2/4 \text{ Byte Füllung}$	24 36
double: 12 Byte Overhead	42	
Füllung, $\geq 4\text{ Byte } N(\text{Size})$	48	
2 Byte N.8 Byte (doubles)	54	
+ 2 Byte	60	
$\Sigma = (16 \text{ Byte} + N \cdot 8 + 2) = (18 + 8N) \text{ Byte}$		
double[][]:		
+ 12 Byte Overhead		
+ 4 Byte N		
+ 2 Byte Füllung		
+ N · (18 + 8 · m) Byte		
+ N · 6 Byte Referenzen		
String:	"char Array"	
12 Byte Overhead	12 Byte Overhead	
24) 6 Byte Referenz	6 Byte N	
Byte 1 Byte Coder	+ 0 Byte Füllung	
4 Byte Hash	+ N · 2 Byte (2 Byte je char)	
1 Byte Füllung	18 + N · 2 Byte	

Node \rightarrow statische Inner Klasse		
12 Byte Overhead	30 Byte	2x 6 static, keinisch
Content ref.: 6 Byte	$\frac{30}{24}$	Existiert nur eine Instanz
Ref. von Outer Class: 6 Byte		\rightarrow Es gibt keinen Extrahover
Next Node Ref.: 6 Byte		
Stack: Fixed Size-Array		
12 Byte Overhead	24 Byte	
4 Byte Länge (int) N		
6 Byte Ref. auf First Node		
2 Byte Füllung		
$\Sigma = 24 + N \cdot 24 \text{ Byte}$		

\hookrightarrow Stack: Fixedsize Array

Einfache Daten Typen, Wrapper & Arrays

ADP02 - Speicherbedarf

Stack: Resizable Array

\hookrightarrow Fallunterscheidung der Array Größe:

- 1) $\frac{1}{2}\text{len} < N < \text{len} \rightarrow$ "Normal Größe" || legal
- 2) $N < \frac{1}{2}\text{len} \rightarrow$ Verkleinerung $\text{len} = \frac{1}{2}\text{len}$
- 3) $N = \text{len} \rightarrow$ Vergrößern $\text{len} = 2 \cdot \text{len}$

12 Byte Overhead

1 Byte Länge N

6 Byte Referenz auf Node[] Obj

24 - 2 Byte Füllung

Node Arry \rightarrow Vergrößern

(12 Byte Overhead)

18 Byte 4 Byte Länge

{ 2 Byte Füllung

$6 \cdot N^2 \text{ Byte} \rightarrow$ Referenzen auf Nodes

$+ N \cdot 24 \text{ Byte} \rightarrow$ Node Objekte

$$\Rightarrow 14 + 18 + N(24 + 6) = 42 + 3ON$$

$\frac{14}{42}$

$\frac{30}{30}$

Node Arry Verkleinern

18 { 12 Byte Overhead

4 Byte Länge

2 Byte Füllung

$N \cdot 4 \cdot 6 \rightarrow$ Referenzen auf Nodes

\hookrightarrow 4 mal array 4 mal größer als N vorhandene Node

$N \cdot 24 \text{ Byte} \rightarrow$ Node Objekte

$$\Rightarrow 18 + 24 + N(24 + 4 \cdot 8)$$

$$\Rightarrow 42 + N \cdot (56) \text{ Byte}$$

\hookrightarrow Stack: Resizable Array

ADP02 Speicherbedarf

Stack: Linked list

12 Byte Overhead

4 Byte Länge

6 Byte \rightarrow Referenz auf First Node

2 Byte Füllung

24 - $N \cdot 24 \text{ Byte} \rightarrow$ Node Objekte

$$\Rightarrow 24 + N \cdot 24 \text{ Byte}$$

\hookrightarrow Stack: linked list