

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.

2.D Insertion- und Selection-Sort

1. Warum ist die Wachstumsordnung für Selection-Sort immer quadratisch? Wieviel Tauschoperationen benötigt Selection-Sort im besten 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.

Zu Aufgabe 2B1

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:

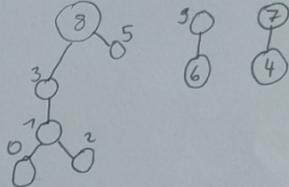
6-9:

4-7:

5-8:

1-5:

1-8: (nix zutun)

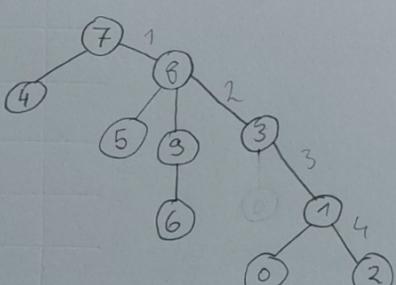


9-1:

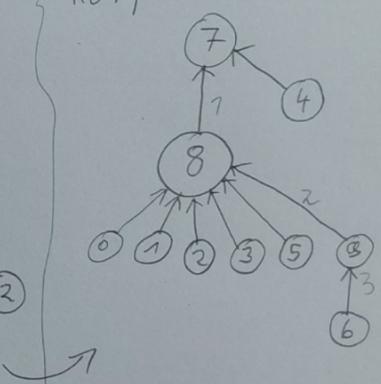
8-2: nix zutun

8-3: nix zutun

8-4:



Komprimierter Baum



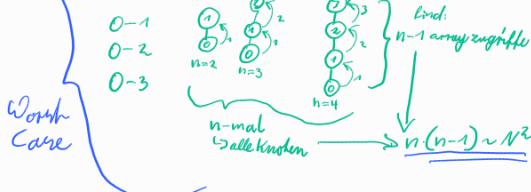
Kompression im Kind.

Zu 2.B.2:

→ Wachstumsordnung:

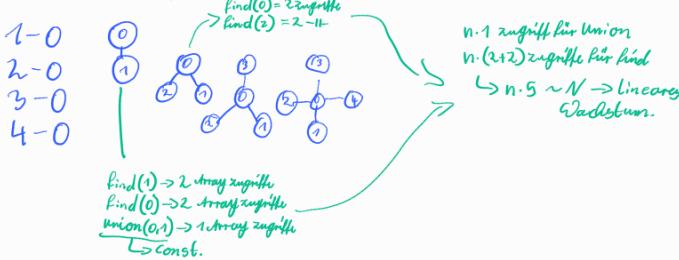
↳ Quick Union muss nicht immer alle Knoten/Elemente im 1D-Array durchlaufen sondern nur Teile Bäume

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



Best Case:

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



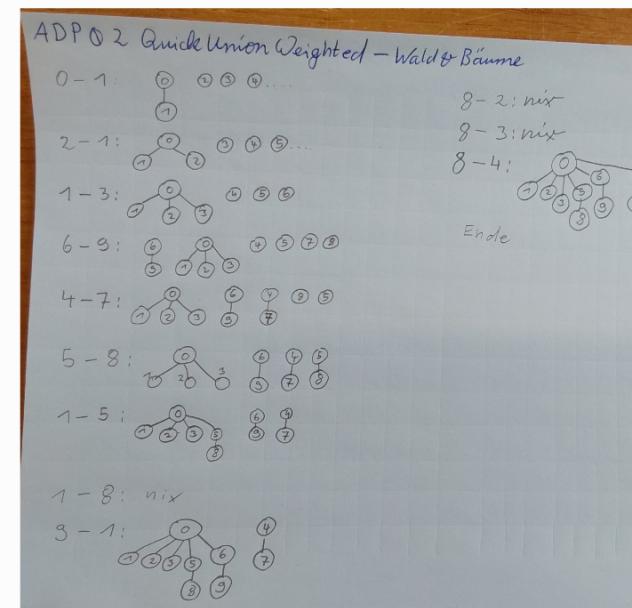
Quick Union

```
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--  
    }  
}
```

Zu 2.B.3

Weighted Quick Union

ADP 02 Weighted Quick Union		proz < q root → hängt p unter q count → hängt unter p																					
p	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	1	2	3	4	5	6	7	8	9	1	1	1	1	1	1	1	1	1	1	1	
2	0	1	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	3	4	5	6	7	8	9	3	1	1	1	1	1	1	1	1	1	1	
4	1	3	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	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	4	1	1	1	2	1	1	1	1	1	
7	5	8	0	0	0	0	4	5	6	4	5	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	
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	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	2	2	2	1	1	1	1	1



Zu 2.B.4 Wachstumsordnung

```
find(p){  
    if(id[p] != p){  
        p = find(id[p])  
    }  
    return p  
}
```

union(p, q)

```
    idp = find(p)  
    idq = find(q)  
    if(idp == idq){  
        return i  
    }  
    if(sz(idp) < sz(idq)){  
        idp = idq  
        sz(idp) += sz(idq)  
    }  
    else{  
        idq = idp  
        sz(idq) += sz(idp)  
    }  
    counter--i  
}
```

Kir n Knoten → n-1 Aufrufe

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

$\log_2 N$ 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! Kostenmodell: Array-Zugriff (willkürliche Operation in Schleife)

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--

Quicks Find!

Kostenmodell: Array-Zugriff (willkürliche Operation in Schleife)

Wachstumsordnung: N^2

Alle Knoten in eine Komponente
 \hookrightarrow mal union aufheben

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

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

$\sim N^2$

2 Zugriffe pro Element

- Arithmetische Operation count = 1