Algorithmen und Datenstrukturen

Wer braucht Algorithmen und Datenstrukturen?

Na, zum Beispiel er:

Für den

Bundestrojaner

Bugfixes verschicken

Listen der Verdächtigen

(Von BKA, LKA, BND, BfV, MAD, BPOL, Stasi, CIA, MI6 …)

Problem:
Viele Duplikate

$$10 \times 10 \times 1.000.000$$

$= 100.000.000$ Vergleiche

„JOIN“ über große Tabellen

Dateien durchsuchen

Abgleich mit Index verbotener Wörter

Problem:
Suche in großen Strings

Problem:
Großer Index

Genom indizieren

Terrornetzwerke aufdecken

Organisationen als Netzwerk darstellen

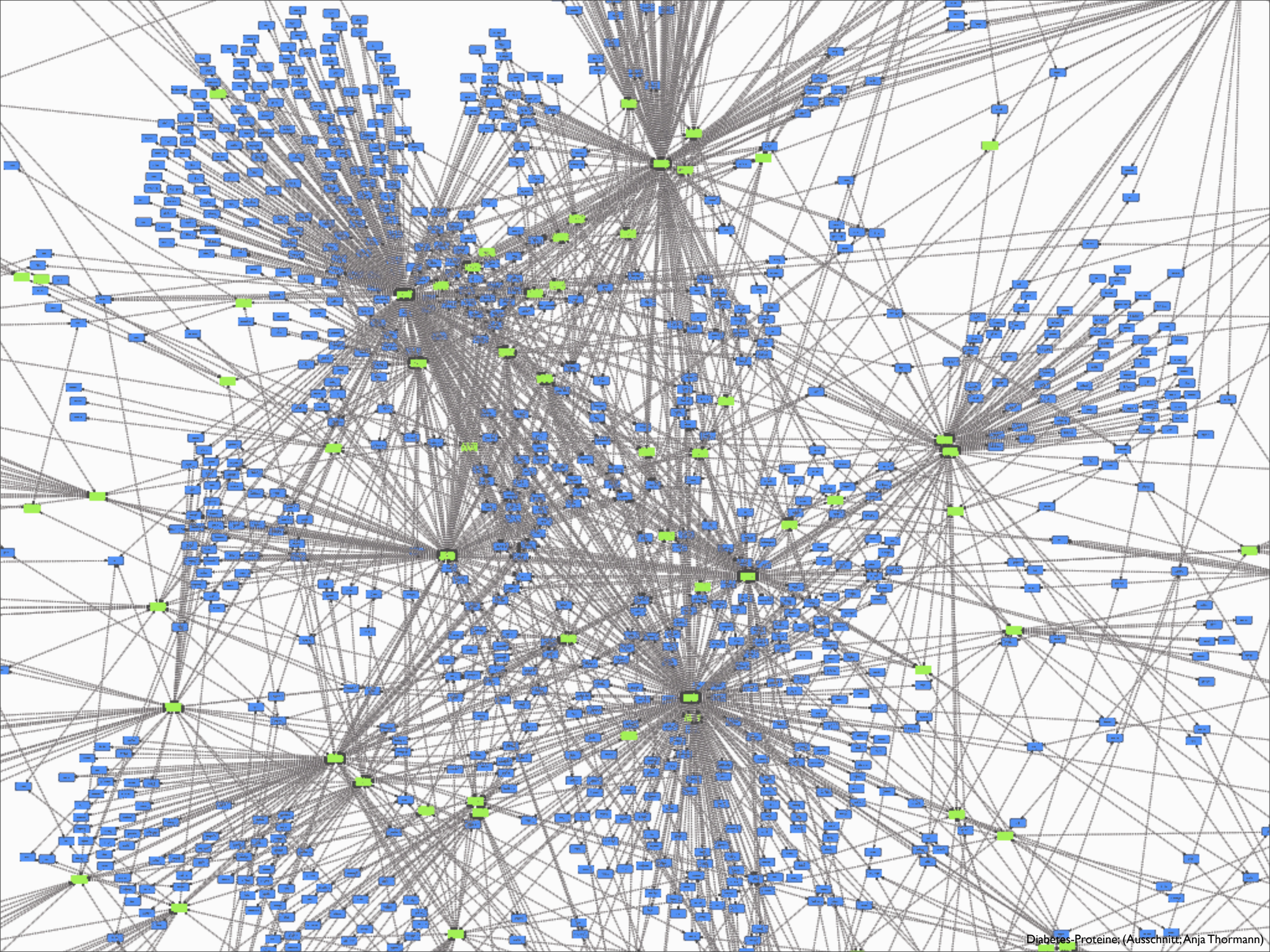
Problem:

Viele Knoten, da viele Täter

$$82.000.000 \times 82.000.000$$

= 6.724.000.000.000.000

Vergleiche

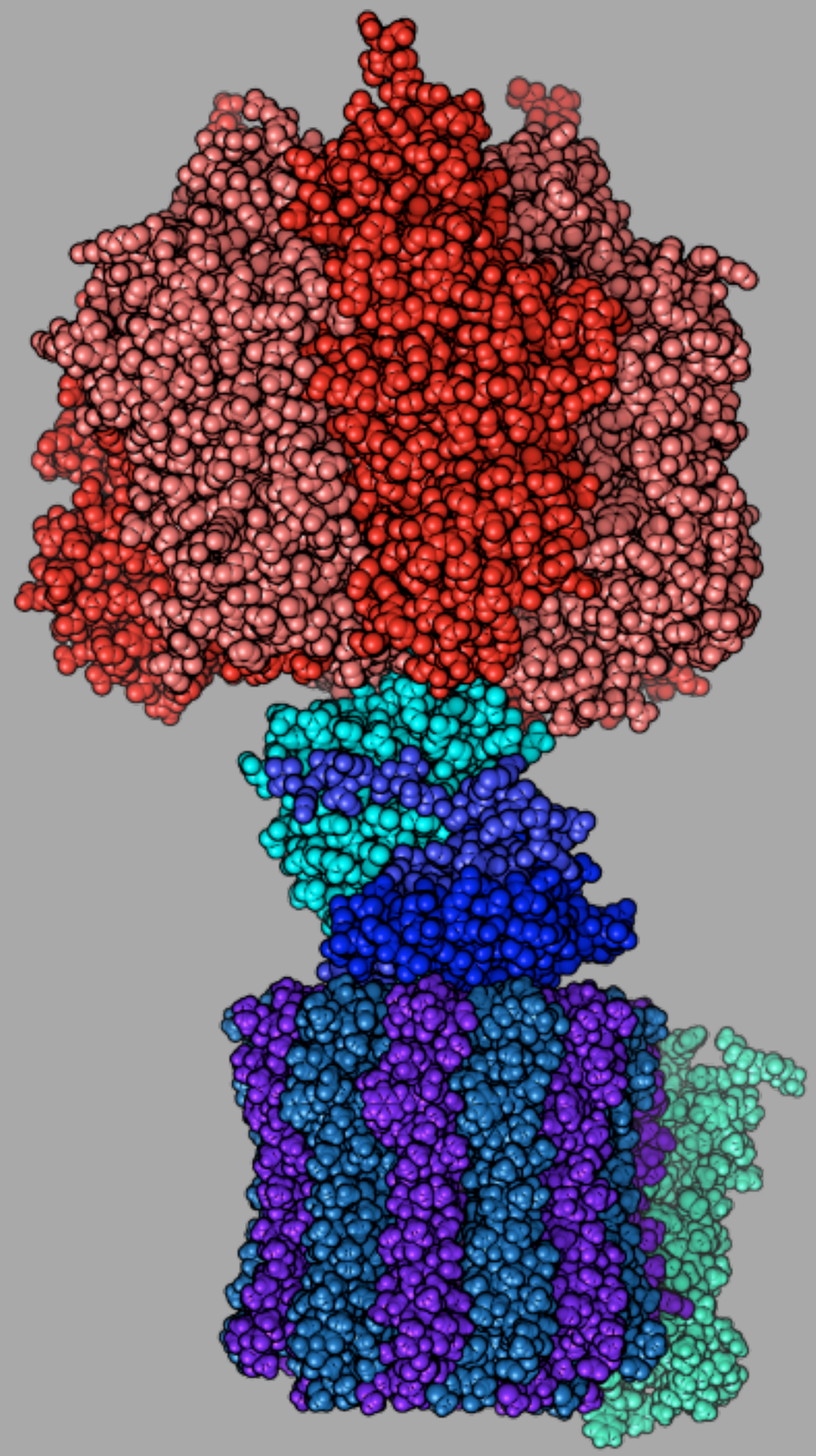


Diabetes-Proteine; (Ausschnitt; Anja Thormann)

3D-Struktur von Proteinen

Exkurs:

Proteinstrukturen



(ATP-Synthase; verändert nach http://en.wikipedia.org/wiki/Image:Atp_synthase.PNG)

DNS

↓

RNS

↓

Protein

ATGACTAACTCACCT

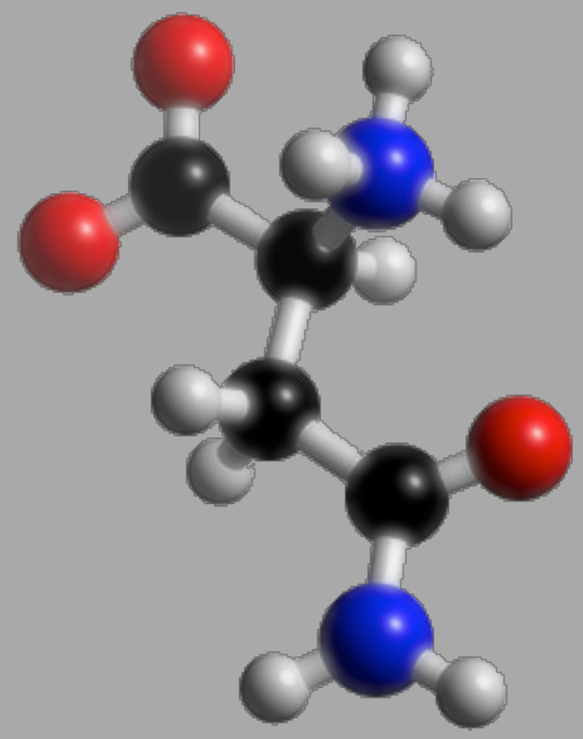
↓

AUGACUAACUCACCU

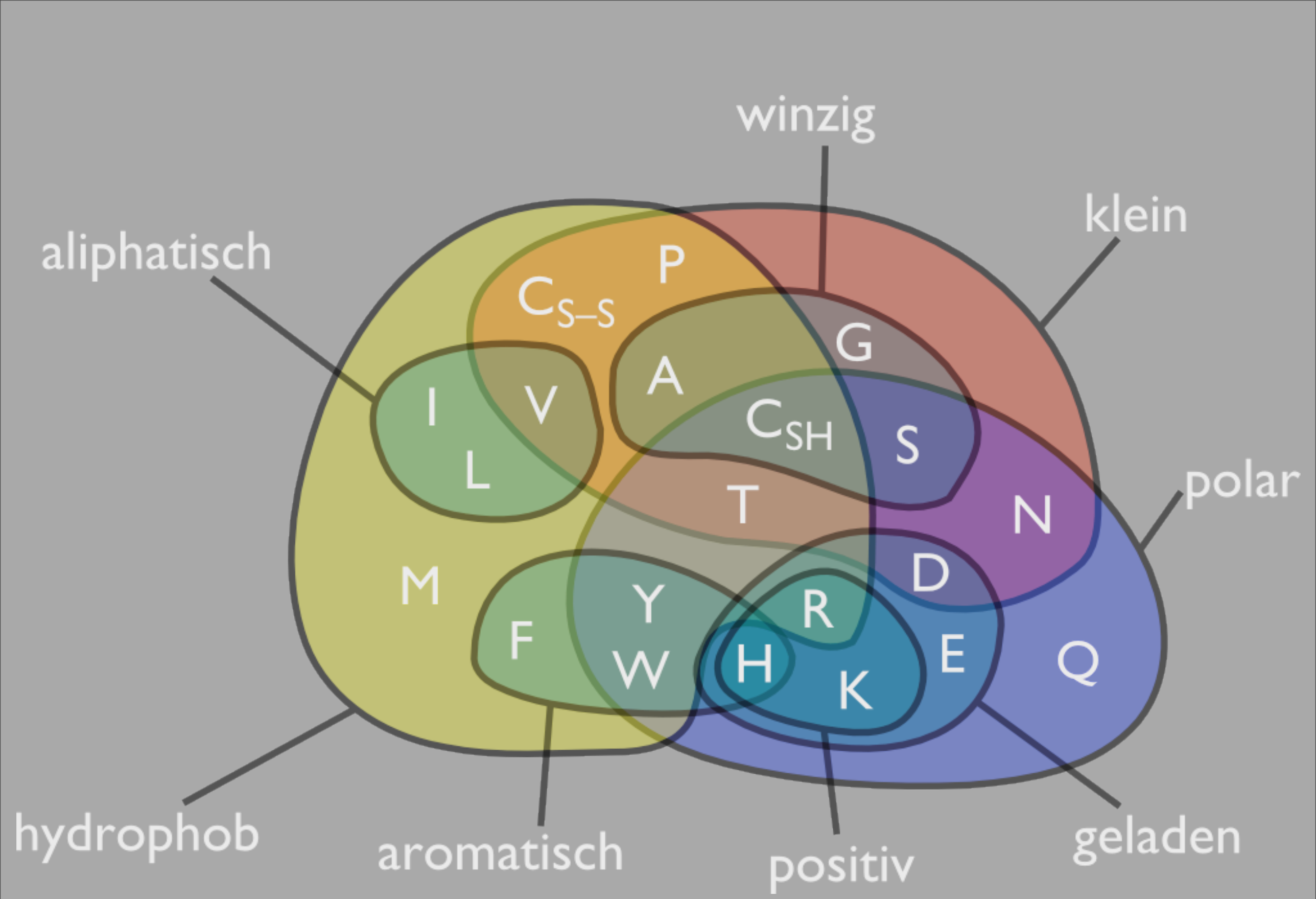
↓

MetThrAsnSerPro

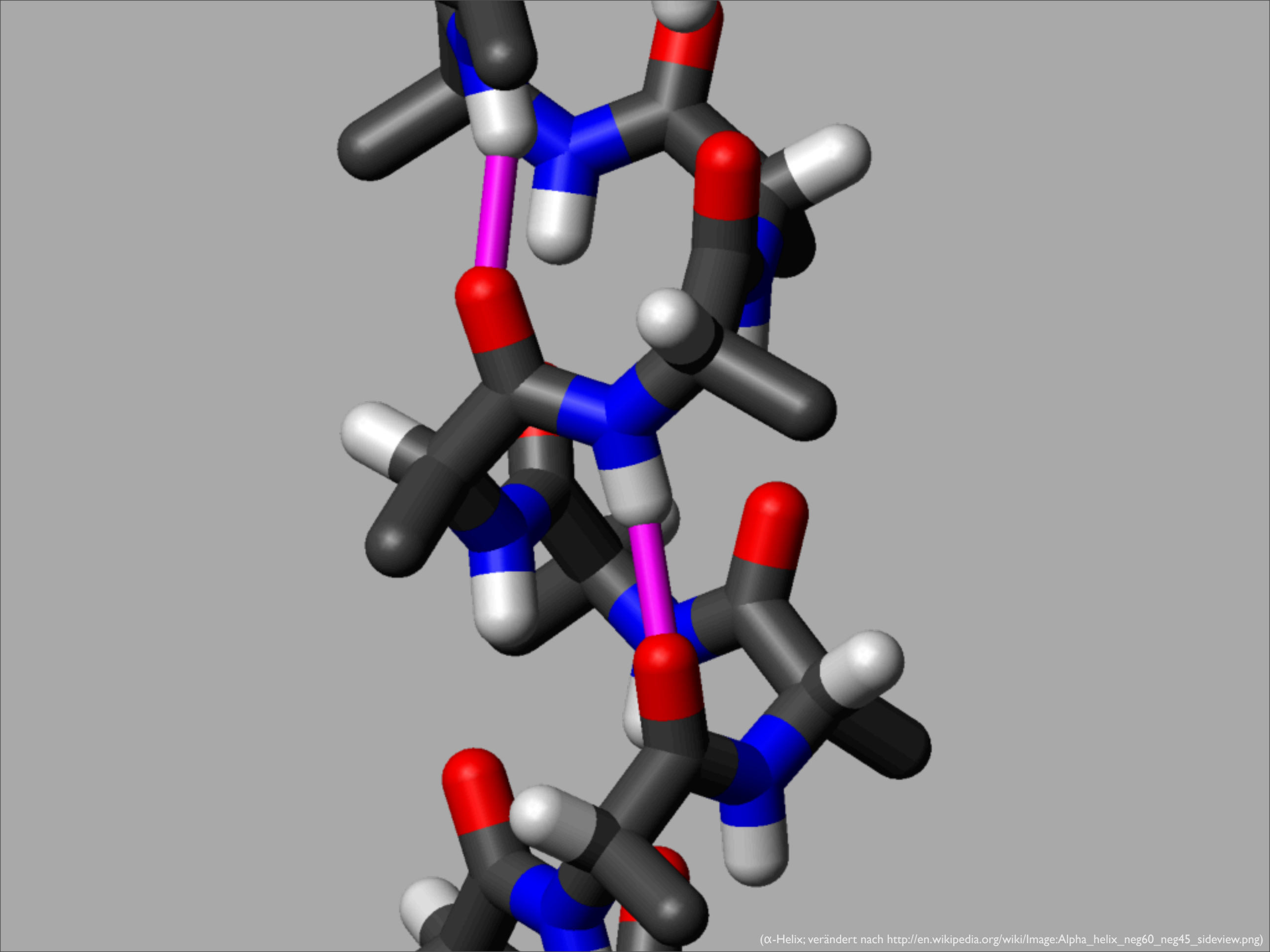
MTNSPLIQFN IKKLIDIKMF GLDVSFTNSS IYMLLASTLA LTYFYLAFYN RKLIPSRLQV
SAEIVYNFVV DMLNQNIGIK GRKFIPLVFS LFIFILFCNL LGMTPYSFTA TSHIIVTFTL
ALLIFLTVTI VGFIKHGMSF LTLFLPQGTP VWLAPLMIVI ELFTYLAKPV SLSLRLAANM
MAGHVLLKVI AGFTVSLMIY LKFLPIPLIV ILIGFEIFVA ILQAYIFTIL SCMYLNDAIN
LHMLSRVVLS AAAAAAPSLK NAALLGPGVL QATRIFHTGQ PSLAPVPPLP EHGGKVRFGL
IPEEFFQFLY PKTGVTGPYV LGTGLILYLL SKEIYVITPE TFSAISTIGF LVYIVKKYGA
SVGEFADKLN EQKIAQLEEV KQASIKQIQD AIDMEKSQQA LVQKRHYLFD VQRNNIAMAL
EVTYRERLHR VYREVKNRLD YHISVQNMMR QKEQEHMINW VEKRVVQSIS AQQEKETIAK
CIADLKLLSK KAQAQPVM



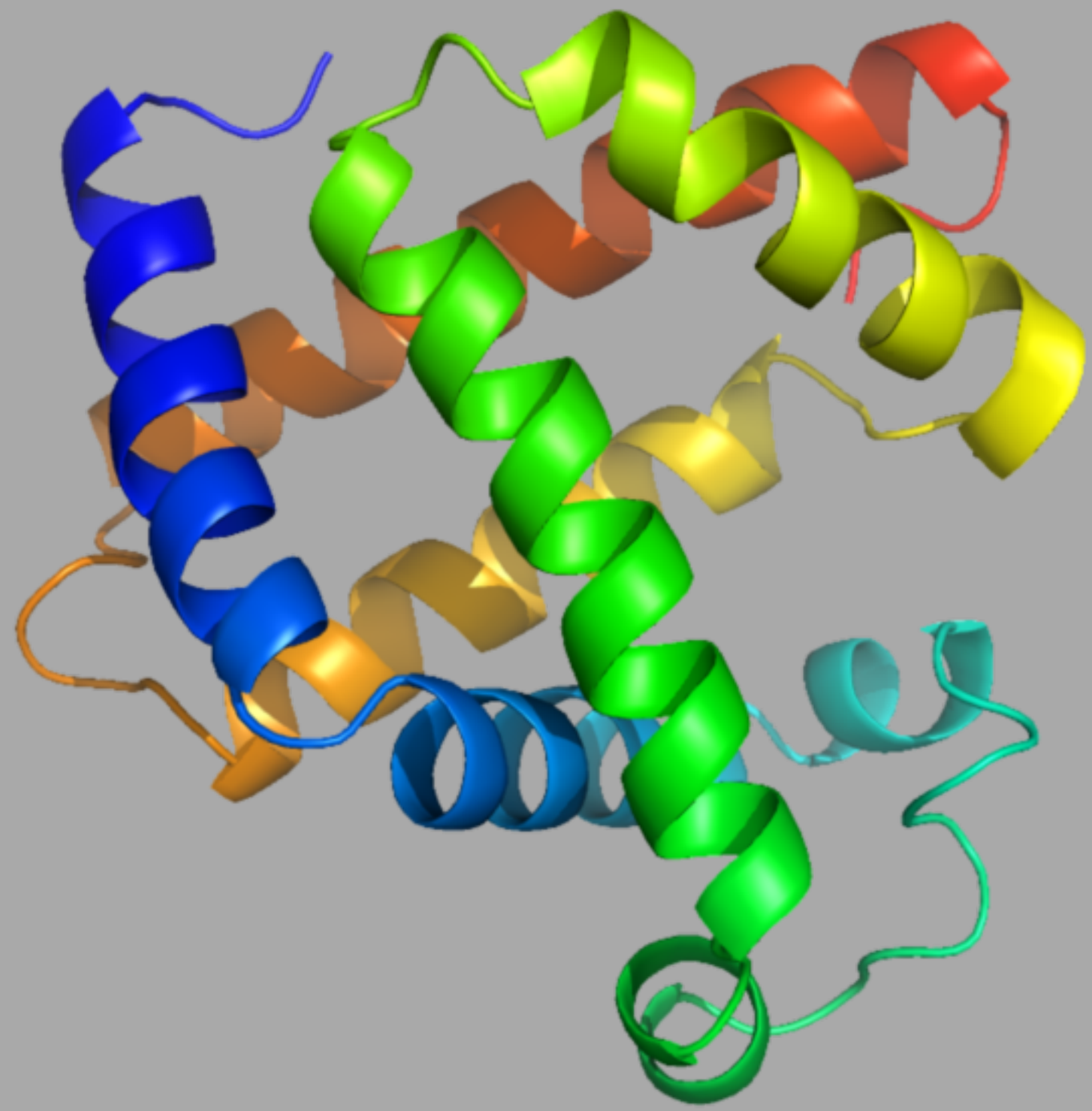
(Asparagin; verändert nach http://www.3dchem.com/molecules.asp?ID=37)



(Aminosäure-Eigenschaften; verändert nach http://de.wikipedia.org/wiki/Bild:Amino_Acids_Venn_Diagram_(de).svg)



(α-Helix; verändert nach http://en.wikipedia.org/wiki/Image:Alpha_helix_neg60_neg45_sideview.png)

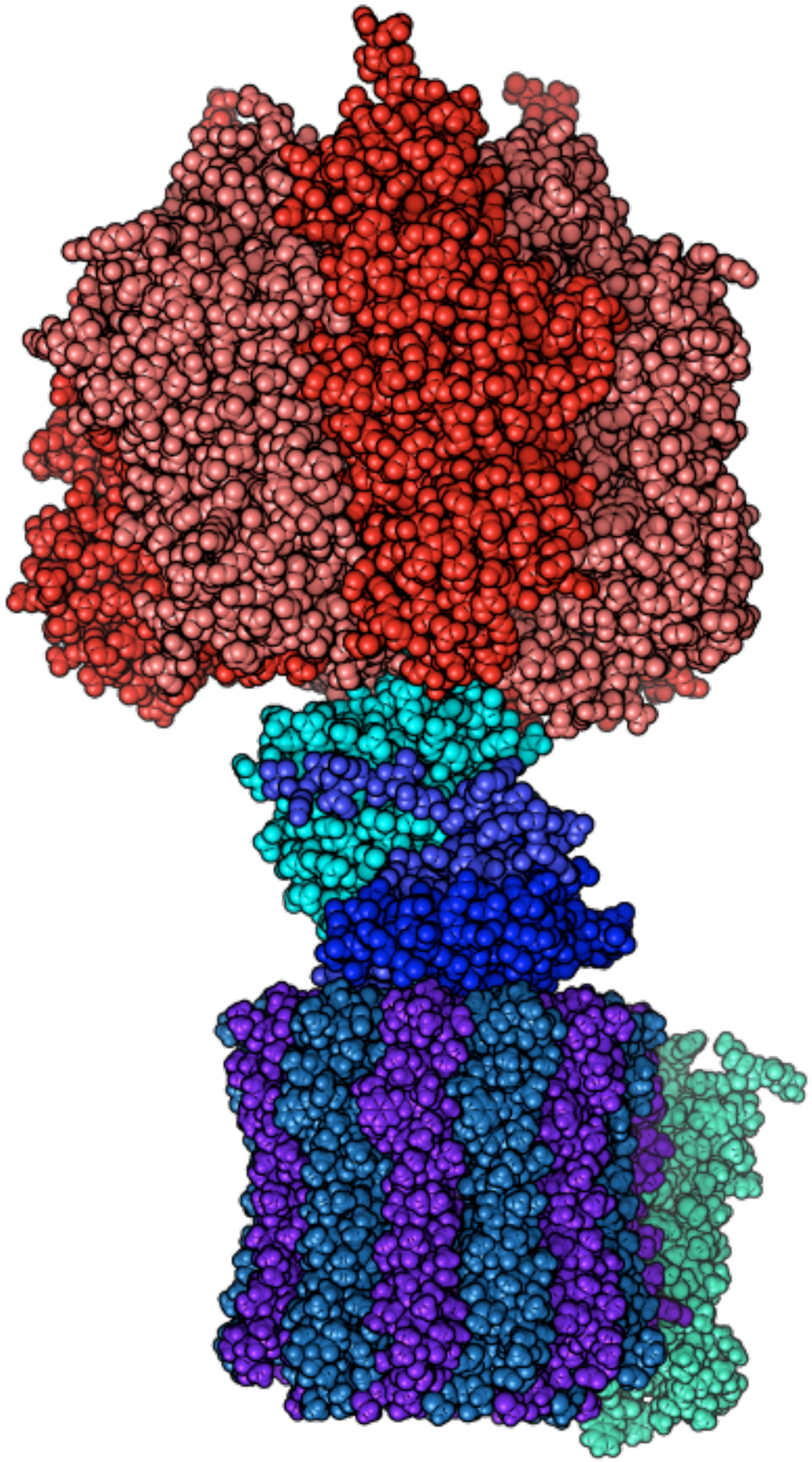


(Pottwal-Myoglobin; verändert nach http://en.wikipedia.org/wiki/Image:Myoglobin.png)

Wir wollen von hier:

MTNSPLIQFN IKKLIDIKMF GLDVSFTNSS IYMLLASTLA LTYFYLAFYN RKLIPSRLQV
SAEIVYNFVV DMLNQNIGIK GRKFIPLVFS LFIFILFCNL LGMTPYSFTA TSHIIVTFTL
ALLIFLTVTI VGFIKHGMSF LTLFLPQGTP VWLAPLMIVI ELFTYLAKPV SLSLRLAANM
MAGHVLLKVI AGFTVSLMIY LKFLPIPLIV ILIGFEIFVA ILQAYIFTIL SCMYLNDAIN
LHMLSRVVLS AAAAAAPSLK NAALLGPGVL QATRIFHTGQ PSLAPVPPLP EHGGKVRFGL
IPEEFFQFLY PKTGVTGPYV LGTGLILYLL SKEIYVITPE TFSAISTIGF LVYIVKKYGA
SVGEFADKLN EQKIAQLEEV KQASIKQIQD AIDMEKSQQA LVQKRHYLFD VQRNNIAMAL
EVTYRERLHR VYREVKNRLD YHISVQNMMR QKEQEHMINW VEKRVVQSIS AQQEKETIAK
CIADLKLLSK KAQAQPVM

nach hier:



(ATP-Synthase; verändert nach http://en.wikipedia.org/wiki/Image:Atp_synthase.PNG)

Der Algorithmus ist einfach

Aber leider zu komplex

Was bedeutet „komplex“?

Exkurs:

Komplexität

Algorithmus „SUMME“ (V: Liste)

$s \leftarrow 0$

Für alle v in V

$s \leftarrow s + v$

Wird einmal durchlaufen

Algorithmus „SUMME“ (V: Liste)

$s \leftarrow 0$

Für alle v in V

$s \leftarrow s + v$

Wird n-mal durchlaufen

$$O(n)$$

= wächst ähnlich schnell

wie n

Algorithmus „SIEB-DES-ERATHOSTENES“ (n: Zahl)

Für i in 2 … n

Schreibe i in eine Liste

Für i in 2 … n

Für j in 2 … n

Streiche $i \times j$ aus der Liste

Wird n-mal durchlaufen

Algorithmus „SIEB-DES-ERATHOSTENES“ (n: Zahl)

Für i in 2 … n

Schreibe i in eine Liste

Für i in 2 … n

Für j in 2 … n

Streiche $i \times j$ aus der Liste

Wird n-mal durchlaufen

Algorithmus „SIEB-DES-ERATHOSTENES“ (n: Zahl)

Für i in 2 … n

Schreibe i in eine Liste

Für i in 2 … n

Für j in 2 … n

Streiche $i \times j$ aus der Liste

Wird n-mal durchlaufen

$$O(n^2)$$

Algorithmus „FIBONACCI-ZAHL“ (n: Zahl)

Wenn $n = 1$ oder $n = 2$ dann

Gib 1 zurück.

Sonst

$a \leftarrow$ FIBONACCI-ZAHL$(n - 1)$

$b \leftarrow$ FIBONACCI-ZAHL$(n - 2)$

Gib $a + b$ zurück.

$$O(n!)$$

$O(\log n)$	$O(n)$	$O(n^2)$	$O(n^3)$	$O(n!)$
1	**2**	4	8	2
2	**4**	16	64	24
3	**8**	64	512	40.320
4	**16**	256	4.096	20.922.789.888.000
5	**32**	1.024	32.768	263.130.836.933.693.530.167.218.012.160.000.000

Wichtiges Merkmal:
Elementare Bausteine
mit Laufzeit $O(1)$

Arithmetische Operationen
Einfügen in eine Menge
Löschen aus einer Menge
Zugriff auf ein einzelnes Element
Ausgabe
Kontrollstrukturen (If, Goto, …)

Aber auch:

Algorithmus „WARTEN“

$n \leftarrow 0$

Solange $n < 1.000.000$

$n \leftarrow n + 1$

Datenstrukturen

Speichern Daten

Definieren Zugriffs-Methoden

Definieren Zugriffs-Zeiten

Array

Lineare Sequenz

Feste Größe

Effizienter Zugriff über den *Rang* eines Elements


S
0
1
2
N – 1

Exkurs:

Rang

Anzahl der Elemente vor
dem aktuellen Element

(Index – 1)

$$R \in [0, N[$$

Zurück zum Array

Definierte Operationen:

Lesen eines Elements
Ändern eines Elements

Beide laufen in $O(1)$

Aber: Die Größe des Arrays ist fest

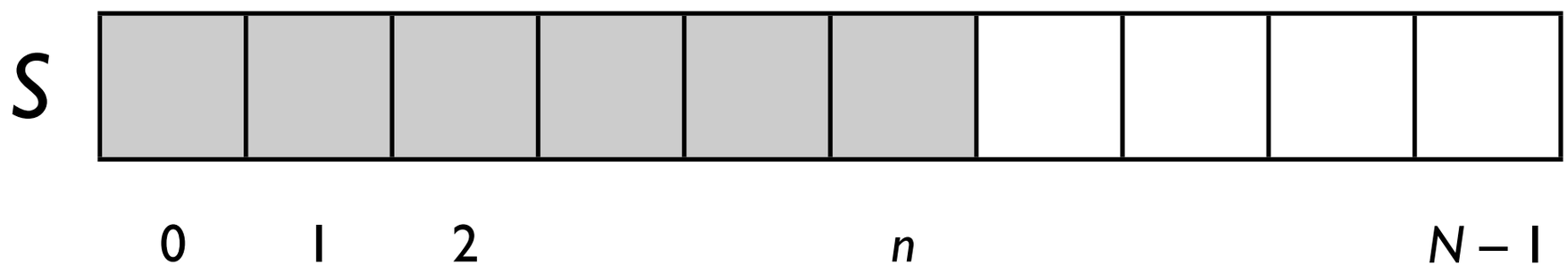
Das ist schlecht

Abhilfe:

Vektor

Dynamische Größe

Trotzdem laufen alle Operationen in $O(1)$


S
0
1
2
n
N – 1

Definierte Operationen:

Lesen eines Elements
Ändern eines Elements
Anhängen eines neuen Elements ans Ende
Löschen des letzten Elements

Algorithmus „ANHÄNGEN“ (A: Vektor, e: Element)

Wenn $n = N$ dann

$B \leftarrow$ neuer Vektor der Länge $2 \times N$

Für i in $0 \ldots n$

$B[i] \leftarrow A[i]$

$A \leftarrow B$

$n \leftarrow n + 1$

$A[n] \leftarrow e$

Das Einfügen von n Elementen in einen leeren Vektor benötigt $O(n)$

$$\Longrightarrow$$

Das Einfügen eines Elements läuft in $O(1)$

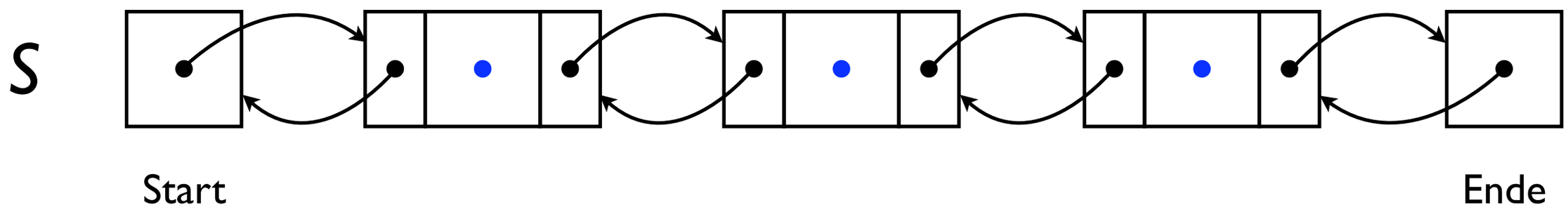
Unter .NET:

List(Of T)

```
Dim V As New List(Of Integer)()
V.Add(42)
Console.Write(V(0))
V(0) = 23
V.RemoveAt(0)
```

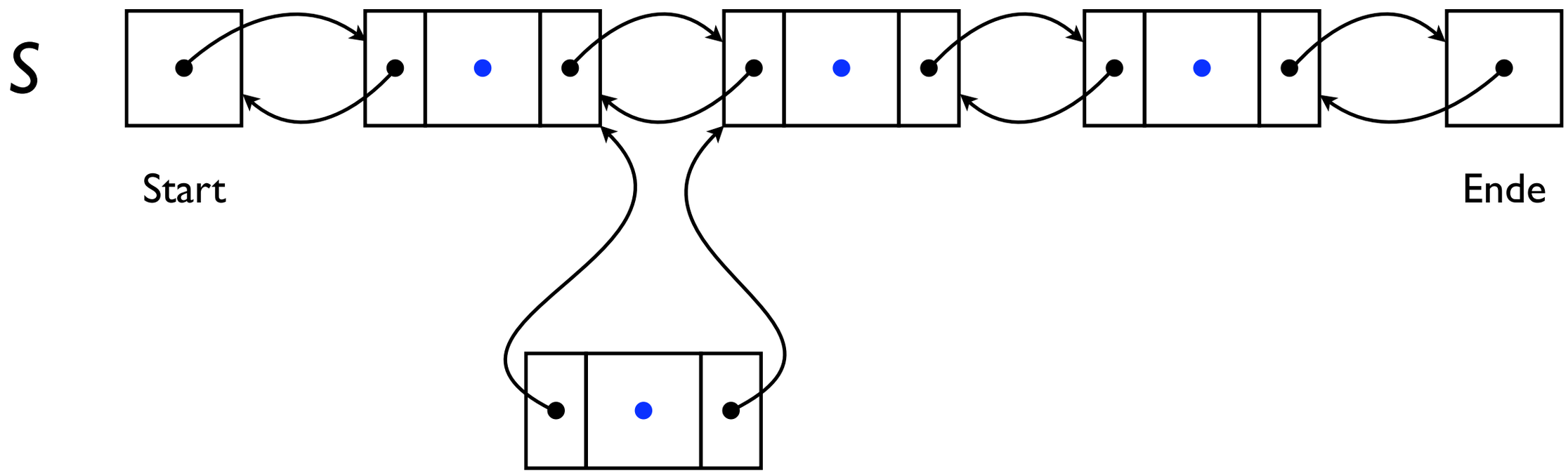
Liste

Zugriff über Verweise


S
Start
Ende

```
Interface Reference(Of T)
    Property Previous() As Reference(Of T)
    Property Next() As Reference(Of T)
    Property Element() As T
End Interface
```

Einfügen und Entfernen an beliebigen Positionen in $O(1)$


S
Start
Ende

Algorithmus „EINFÜGENNACH“

(p: Verweis, e: Element)

Erstelle neuen Verweis v

$v.Element \leftarrow e$

$v.Previous \leftarrow p$

$v.Next \leftarrow p.Next$

$p.Next.Previous \leftarrow v$

$p.Next \leftarrow v$

Unter .NET:
LinkedList(Of T)

```
Dim V As New LinkedList(Of Integer)()
V.AddLast(42)
V.AddAfter(V.Last, 23)
```

Suchen in Sequenzen

Algorithmus „ENTHÄLT“ (V: Vektor, e: Element)

Für i in $0 \ldots n$

Wenn $V[i] = e$ dann

Gib „*wahr*“ zurück.

Gib „*falsch*“ zurück.

Algorithmus „ENTHÄLT“ (L: Liste, e: Element)

$n \leftarrow L.Start$

Solange $n \neq L.Ende$

Wenn $n.Element = e$ dann

Gib „*wahr*“ zurück.

$n \leftarrow n.Next$

Gib „*falsch*“ zurück.

Laufzeit $O(n)$

Vereinigen mehrerer Sequenzen

Algorithmus „VEREINIGUNG“

(*A*: Sequenz, *B*: Sequenz)

Für alle *e* in *B*

Wenn nicht ENTHÄLT(*A*, *e*) dann

ANHÄNGEN(*A*, *e*)

Laufzeit $O(n^2)$

Das geht effizienter!

Mengen

Unsortierte Sequenzen

Jedes Element kommt nur einmal vor

Algorithmus „VEREINIGUNG“ (A: Menge, B: Menge)

Für alle e in B

Wenn nicht ENTHÄLT(A, e) dann

ANHÄNGEN(A, e)

Derselbe Algorithmus

Mit erwarteter Laufzeit

$O(n)$

Algorithmus „VEREINIGUNG“ (A: Menge, B: Menge)

Für alle e in B

Wenn nicht ENTHÄLT(A, e) dann

ANHÄNGEN(A, e)

Läuft in $O(1)$ für Mengen

Mögliche Implementierung:
Als Hashtabelle

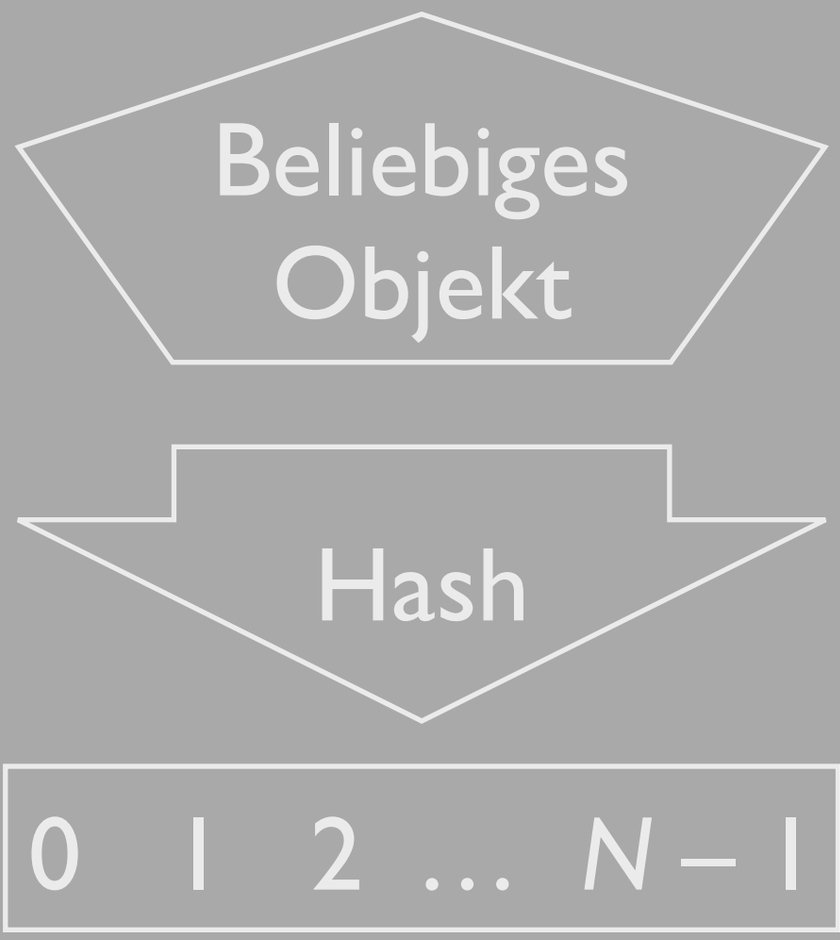
Exkurs:

Hashtabelle

Elemente stehen in Array

Rang wird durch
Hashfunktion berechnet

Hash bildet eine Menge A auf eine Menge B ab, sodass $|A| \leq |B|$


Beliebiges Objekt
Hash
0 1 2 … N – 1

Beispiel: Paritätsbit

Algorithmus „PARITÄT“ (S: Bit-Sequenz)

$p \leftarrow 0$

Für alle b in S

$p \leftarrow p \oplus b$

Gib p zurück.

Beispiel: FNV

Algorithmus „FNV“ (S: String)

$h \leftarrow 2166136261$

Für alle c in S

$h \leftarrow (h \times 16777619) \oplus c$

Gib h zurück.

Neues Element einfügen:

Algorithmus „EINFÜGEN“

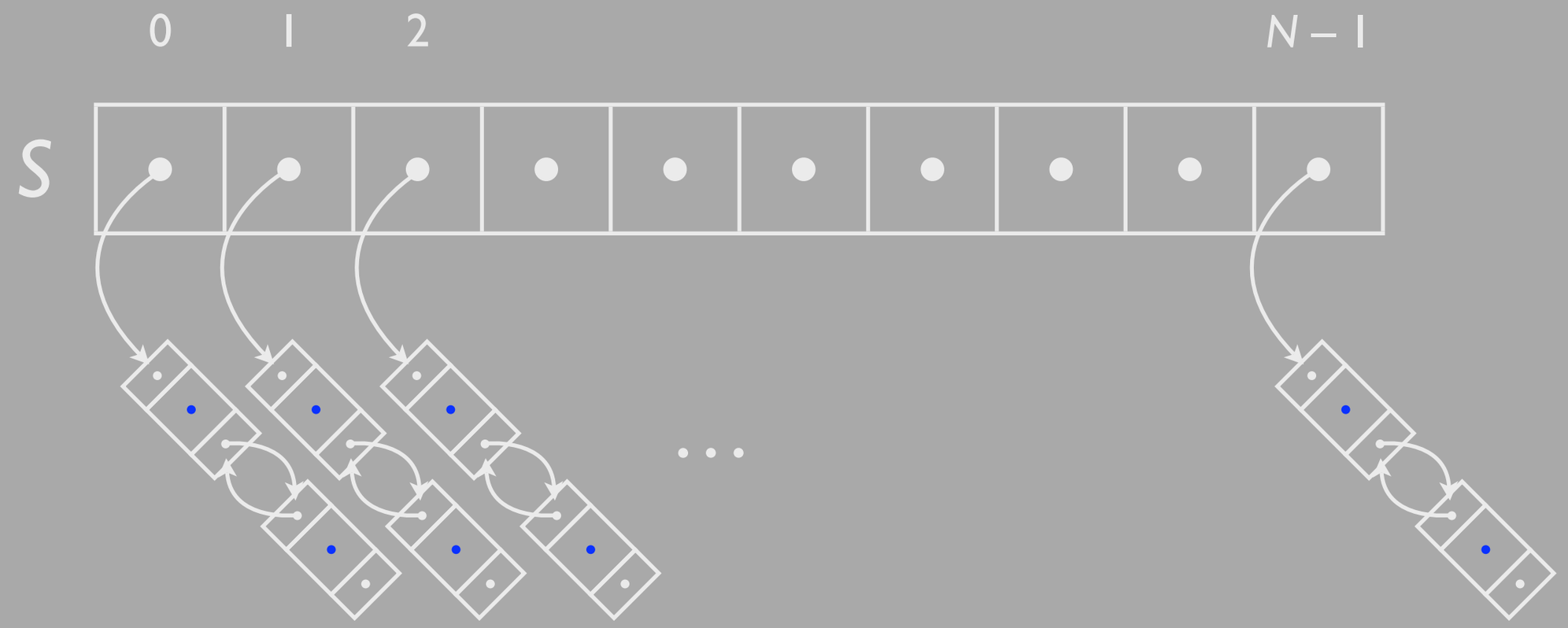
(A: Hashtabelle, e: Element)

$i \leftarrow \text{HASH}(e) \bmod N$

$A[i] \leftarrow e$

Problem: Kollisionen

Lösung: Array mit „Eimern“


0
1
2
N − 1
S

Algorithmus „EINFÜGEN“

(*A*: Hashtabelle, *e*: Element)

$i \leftarrow$ HASH(e) mod N

Wenn $A[i] = null$ dann

Erstelle eine neue Liste L

$A[i] \leftarrow L$

Sonst

$L \leftarrow A[i]$

EINFÜGENNACH($L.Last$, e)

Algorithmus „ENTHÄLT“

(*A*: Hashtabelle, *e*: Element)

$i \leftarrow$ HASH(e) mod N

$L \leftarrow A[i]$

Wenn $L \neq \emptyset$ dann

Wenn ENTHÄLT(L, e) dann

Gib „*wahr*“ zurück.

Gib „*falsch*“ zurück.

Worst-case: $O(n)$

Worst-case-Laufzeit ist also nach wie vor $O(n^2)$

Hat sich etwas verbessert?

Ja: die *erwartete* Laufzeit

Eine gute Hashfunktion
verhindert den worst case

Normalerweise enthält jeder „Eimer“ nur ein Element

D.h. Laufzeit von ENTHÄLT ist im Normalfall $O(1)$

Algorithmus „VEREINIGUNG“ (A: Menge, B: Menge)

Für alle e in B

Wenn nicht ENTHÄLT(A, e) dann

ANHÄNGEN(A, e)

Läuft in $O(1)$ für Mengen

Algorithmus „VEREINIGUNG“ (A: Menge, B: Menge)

Für alle e in B

Wenn nicht ENTHÄLT(A, e) dann

ANHÄNGEN(A, e)

Läuft in $O(n)$

Fazit

Datenstrukturen verwalten

Daten

Datenstrukturen stellen Bindeglieder zwischen Algorithmen dar

Algorithmen sind oft generisch

Je nach Aufgabe sind andere Datenstrukturen geeignet

Algorithmen sollten so allgemein wie möglich gehalten werden

Damit das klappt, müssen Datenstrukturen einheitliche Schnittstellen haben

- „Schäublone“: Schrift „Parole“ von www.fontblog.de
- „FNV“-Hash von www.isthe.com/chongo/tech/comp/fnv/
- Michael T. Goodrich et al., „Data Structures and Algorithms in Java“ (2nd ed., 2001)
- Lubert Stryer et al., „Biochemistry“ (5th ed., 2002)
- Bruce Alberts et al., „Molecular Biology of the Cell“ (4th ed., 2002)
- Donald E. Knuth, „The Art of Computer Programming“, Volumne 1 (3rd ed., 1997)
- Anja Thormann, Analyse von Diabetes-Proteinen
- RCSB Protein Data Bank, www.rcsb.org/pdb/
- Wikipedia, en.wikipedia.org