# Aufgabe 1: Die Kunst der Abstraktion

In dieser Aufgabe werden wir unterschiedliche ADT's gemäß einer vorgegebenen Skizze implementieren. Die Skizze wird aus einer innerhalb der Praktikumsteilnehmer_innen erstellten Skizze ausgewählt. Ziel ist, die Kunst der Abstraktion zu üben. Daher sind die Programme so zu gestalten, dass die jeweiligen ADT's einzeln mit anderen Teams getauscht werden können, z.B. tauscht Team 06 mit Team 03 die ADT Stack, ohne jedoch die ADT Liste zu tauschen.

## Aufgabenstellung

Die exakten technischen Vorgaben (syntaktische Signatur, Dateinamen etc.) werden in einer Skizze von einem Team beschrieben werden. Zur Fehlerbehandlung: Sollten nicht vorhandene Elemente gelöscht werden,
in der Liste an unmöglicher Stelle eingefügt werden, etc. ist die Fehlerbehandlung durch „Ignorieren“ durchzuführen, d.h. es wird so gehandelt, als hätte die Operation nicht stattgefunden. Aus Dokumentationsgründen können log-Dateien erstellt werden.

Eine **ADT Liste** ist zu implementieren:

**Vorgabe**:
Funktional (nach außen)

1. Die Liste beginnt bei Position 1.
2. Die Liste arbeitet nicht destruktiv, d.h. wird ein Element an einer vorhandenen Position eingefügt, wird das dort stehende Element um eine Position verschoben.
3. Die Elemente sind vom Typ „ganze Zahl“.
4. equal testet auf strukturelle Gleichheit

Technisch (nach innen)

1. Die Liste ist intern mittels Java Array `int[ ]` zu realisieren.

**Objektmengen:** pos, elem, list
**Operationen**: (semantische Signatur)

create: ∅ → list
isEmpty: list → bool
equal: list × list → bool
laenge: list → int
insert: list × pos × elem → list
delete: list × pos → list
find: list × elem → pos
retrieve: list × pos → elem
concat: list × list → list

Eine **ADT Stack** ist zu implementieren:

**Vorgabe**:
Technisch (nach innen)

1. Die ADT Stack ist mittels ADT Array zu realisieren.
2. equalS testet auf strukturelle Gleichheit
3. Die Operation reverseS muss ohne Verschiebung von Elementen, d.h. nur über Indizes implementiert werden (konstanter Zeitaufwand O(1)!).

**Objektmengen:** elem, stack
**Operationen**: (semantische Signatur)

createS: ∅ → stack
push: stack × elem → stack
pop: stack → stack
top: stack → elem
isEmptyS: stack → bool
equalS: stack × stack → bool
reverseS: stack → stack

Eine **ADT Queue** ist zu implementieren:

**Vorgabe**:
Technisch (nach innen)

1. Die ADT Queue ist mittels ADT Stack, wie in der Vorlesung vorgestellt, zu realisieren. Es sind z.B. zwei explizite Stacks zu verwenden und das „umstapeln“ ist nur bei Zugriff auf einen leeren „out-Stack“ durchzuführen.
2. equalQ testet auf strukturelle Gleichheit
3. Beim Umstapeln darf kein Element bewegt werden. Die Operation ist mittels der reverseS Operation der ADT Stack zu implementieren (konstanter Zeitaufwand O(1)!).

**Objektmengen:** elem, queue
**Operationen**: (semantische Signatur)

createQ: ∅ → queue
enqueue: queue × elem → queue
dequeue: queue → queue (Mutator)
front: queue → elem
isEmptyQ: queue → bool
equalQ: queue × queue → bool

Eine **ADT Array** ist zu implementieren:

**Vorgabe**:
Funktional (nach außen)

1. Das Array beginnt bei Position 0.
2. Das Array arbeitet destruktiv, d.h. wird ein Element an einer vorhandenen Position eingefügt, wird das dort stehende Element überschrieben.
3. Die Länge des Arrays wird bestimmt durch die bis zur aktuellen Abfrage größten vorhandenen und explizit beschriebenen Position im array.
4. Das Array ist mit 0 initialisiert, d.h. greift man auf eine bisher noch nicht beschriebene Position im Array zu erhält man 0 als Wert.
5. Das Array hat keine Größenbeschränkung, d.h. bei der Initialisierung wird keine Größe vorgegeben.

Technisch (nach innen)

1. Die ADT Array ist mittels ADT Liste zu realisieren.
2. equalA testet auf strukturelle Gleichheit

**Objektmengen:** pos, elem, array
**Operationen**: (semantische Signatur)

initA: ∅ → array

setA: array × pos × elem → array
getA: array × pos → elem
lengthA: array → pos
equalA: array × array → bool

Eine **ADT Array** ist als Alternative zur vorangegangenen Version zu implementieren (also beide können nicht gleichzeitig genutzt werden!):

**Vorgabe**:
Funktional (nach außen)

1. Das Array beginnt bei Position 0.
2. Das Array arbeitet destruktiv, d.h. wird ein Element an einer vorhandenen Position eingefügt, wird das dort stehende Element überschrieben.
3. Die Länge des Arrays wird bestimmt durch die bis zur aktuellen Abfrage größten vorhandenen und explizit beschriebenen Position im array.
4. Das Array ist mit 0 initialisiert, d.h. greift man auf eine bisher noch nicht beschriebene Position im Array zu erhält man 0 als Wert.
5. Das Array hat keine Größenbeschränkung, d.h. bei der Initialisierung wird keine Größe vorgegeben.

Technisch (nach innen)

1. ist mittels Java Array `int[ ]` zu realisieren.
2. equalA testet auf strukturelle Gleichheit
3. wird auf eine noch nicht vorhandene Position schreibend zugegriffen oder ist das Array zu 75% gefüllt, wird das Array vergrößert. Alle bisherigen Einträge müssen dann kopiert werden.

**Objektmengen:** pos, elem, array
**Operationen**: (semantische Signatur)
initA: ∅ → array
setA: array × pos × elem → array
getA: array × pos → elem
lengthA: array → pos
equalA: array × array → bool

Eine **ADT BTree** ist zu implementieren:

**Vorgabe**:
Funktional (nach außen)

1. Der binäre Baum kann in einem Knoten eine Zahl speichern, die Höhe des Baums und einen linken oder rechten Nachfolger (rekursive Struktur).
2. Alle Zahlen im rechten Teilbaum eines Knotens sind größer oder gleich seiner Zahl und alle Zahlen im linken Teilbaum sind kleiner seiner Zahl.
3. Für einen leeren Baum wie auch einen nicht vorhandenen Nachfolger ist das selbe Symbol zu verwenden.

Technisch (nach innen)

1. equalBT testet auf strukturelle Gleichheit.

**Objektmengen:** elem, btree, high
**Operationen**: (semantische Signatur)

initBT: ∅ → btree
setLeftSuc: btree × btree → btree
setRightSuc: btree × btree → btree
setHigh: btree × high → btree
setVal: btree × elem → btree
getLeftSuc: btree → btree
getRightSuc: btree → btree
getHigh: btree → high
getVal: btree → elem
isEmptyBT: btree → bool
equalBT: btree × btree → bool
print: btree × filename → png

Zudem sind für die jeweilige ADT **JUnit-Tests** zu implementieren, die einfach ausgetauscht werden können (als *.jar z.B. adtlistJUt.jar speichern, siehe Vorgabe in der Skizze). Beachten Sie: die Tests sollten auch Belastungen (große Datenmengen) und Grenzfälle beinhalten. Die Tests werden ausgetauscht. **Deshalb dürfen die Tests nicht von der internen Darstellung der ADTs abhängen**, sondern nur rein die Funktion betreffen! Die durchgeführten Tests sind zu dokumentieren (als *.pdf) und gehören zur Abgabe: Testkonzept, Testspezifikation und Testbericht zur eigenen Software und zu mindestens einer Software eines anderen Teams.

# Abnahme

Da die Aufgaben des Praktikums sehr frühzeitig im WWW zur Verfügung stehen, wird die Abnahme stark auf eine **vorbereitende Arbeit** aufgebaut.

**Bis Mittwoch Abend 20:00 Uhr zwei Wochen** vor Ihrem Praktikumstermin ist eine erste Skizze der Aufgabe als *.pdf Dokument (Dokumentationskopf nicht vergessen!) mir per E-Mail zuzusenden (Gruppe 1 **07.04**; Gruppe 2 **24.03**; Gruppe 3 **30.03**). Ggf. können offene Fragen mit gesendet werden. Die Skizze **muss** grob beschreiben, wie Sie sich die Realisierung denken. Als erfolgreich wird eine Skizze bewertet, wenn Ihre Kenntnisse bzgl. der gestellten Aufgabe eine erfolgreiche Teilnahme an dem Praktikumstermin in Aussicht stellen.

**Am Tag vor dem Praktikumstermin** bis **20:00 Uhr** : aktuellen Stand (als *.zip) zusenden.

Am Tag des Praktikums findet eine Befragung von Teams statt. Die **Befragung muss erfolgreich absolviert werden**, um weiter am Praktikum teilnehmen zu können. Ist die Befragung nicht erfolgreich, gilt die Aufgabe als nicht erfolgreich bearbeitet. Als erfolgreich wird die Befragung bewertet, wenn Ihre Kenntnisse bzgl. der gestellten Aufgabeausreichend oder besser sind. Dazu gehört insbesondere eine mindetsens ausreichende Kenntnis über Ihren gesamten Code. Bei der Befragung handelt es sich nicht um die Abnahme.

**Abgabe**: Unmittelbar am Ende des Praktikums, spätestens bis 19:00 Uhr am selben Tag, ist von allen Teams der Code abzugeben. Zu dem Code gehören die Sourcedateien, die ggf. erzeugten *.log etc. Dateien, die während der Tests erzeugt wurden, und eine Readme.txt Datei, in der ausführlich beschrieben wird, wie die Software zu starten ist! Zudem ist der aktuelle Dokumentationskopf abzugeben. Die Dateien sind als *.zip Ordner (mit cc an den/die Teampratner_in) per E-Mail abzugeben. Die Abgabe gehört zu den PVL-Bedingungen und ist einzuhalten, terminlich wie auch inhaltlich!

Wird eine Aufgabe nicht erfolgreich bearbeitet, gilt die **PVL** als **nicht bestanden**. Damit eine

Aufgabe als erfolgreich gewertet wird, muß die Befragung als erfolgreich gewertet werden sowie die Abgabe abgenommen worden sein. **Alle gesetzten Termine sind einzuhalten**. Dies ist notwendig, da sonst erhebliche zeitliche Verzögerungen stattfinden würden.