Team: 06; Atabek Amanov, Thomas Bednorz

Aufgabenaufteilung:

1. Skizze (adp1_skizze-amanov_bednorz.pdf), Atabek Amanov
2. Skizze (adp1_skizze-amanov_bednorz.pdf), Thomas Bednorz

Quellenangaben: -

Bearbeitungszeitraum:

- Di, 22.03.16, 2 Stunden
- Mi, 23.03.16, 7 Stunden
- Do, 24.03.16 1 Stunde

Aktueller Stand: Erste Version der Skizze fertig und zur Begutachtung eingereicht

Änderungen in der Skizze: -

Aufgabe 1: Die Kunst der Abstraktion

Allgemeine Vorbemerkungen

- Das Design ist in einer objektorientierten Denkweise gehalten. Die Methoden des ADTs sind an das ADT-Objekt gebunden. Dies führt zu Abweichungen der Signaturen von der semantischen Struktur der Operationen aus der Aufgabe. So soll beispielsweise beim Einfügen eines Elements in eine Liste auf dem ADT Listen-Objekt die Methode *insert(pos, elem)* gerufen werden, die vorgegebene semantische Struktur dazu ist aber *insert: list x pos x elem → list*.

- Die ADT-Objekte werden auf dem konventionellen Weg der Objekterzeugung in Java erstellt. Dies kann dazu führen, dass die *create*-Methoden (bzw. *init-Methoden* bei den Arrays und dem Binärbaum) der ADTs keinerlei Funktionalität aufweisen und lediglich zur Einhaltung der Schnittstellenvorgaben vorhanden sein müssen. Falls das den implementierenden Entwicklern sinnfrei erscheint, ist es möglich, dass die interne Repräsentation durch einen anderen ADT (bzw. Java int[]) erst beim Aufruf der *create*-Methode initialisiert wird. Dadurch wäre der ADT erst nach einmaligem Aufruf der *create*-Methode benutzbar.

    - Alternativ wäre es denkbar, die *create*-Methode des jeweiligen ADTs an ein Factory-Objekt zu binden.

- Es steht den implementierenden Entwicklern frei, die ADTs funktional zu implementieren. Dies hätte einige Konsequenzen für die folgenden Spezifikationen:

    - Der Rückgabewert von allen Methoden, die den Rückgabewert „keiner (Mutator)“ haben würde sich auf den ADT des entsprechenden Typs ändern.

    - Die zu implementierenden Tests würden nicht mehr prüfen, ob das ADT-Objekt gemäß den Erwartungen verändert wurde, sondern ob das neu erzeugte Objekt die erwarteten Eigenschaften aufweist.

- Im folgenden wird davon ausgegangen, dass die Implementation nicht-funktional vorgenommen wird.

ADT Liste:

Vorgaben / Anmerkungen

- Funktional

  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

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

Objektmengen

- pos := Die Position innerhalb der Liste als natürliche Zahl größer oder gleich 1. Realisiert als Java Integer.
- elem := Das Element innerhalb der Liste als ganze Zahl. Realisiert als Java Integer
- list := Eine Liste von dem hier spezifizierten Typ ADT Liste

Operationen

- **create: Ø → list**
  - Signatur: *#create()*
  - Beschreibung:
    - Erzeugt eine leere Liste
  - Rückgabewert:
    - keiner (Mutator)
  - Fehlerbehandlung
    - keine

- **isEmpty: list → bool**
  - Signatur: *#isEmpty()*
  - Beschreibung:
    - Prüft, ob die Liste leer ist
  - Rückgabewert
    - *true*, falls die Liste leer ist
    - *false*, falls die Liste 1 oder mehr Elemente enthält
  - Fehlerbehandlung:
    - keine

- **equal: list × list → bool**
  - Signatur: *#equal(list)*
  - Beschreibung:
    - Prüft, ob die Liste strukturell identisch zu einer anderen Liste ist
    - Strukturelle Identität ist gegeben genau dann, wenn für jede Position einer Liste gilt, dass die selbe Zahl gespeichert ist wie in der verglichenen Liste
    - Beide Listen müssen die gleiche Längen haben
  - Rückgabewert
    - *true*, falls beide Listen strukturell identisch sind
    - *false*, wenn für mindestens ein verglichenes Element gilt, dass es nicht identisch ist
  - Fehlerbehandlung:
    - keine

- **laenge: list → int**
  - Signatur: *#laenge()*
  - Beschreibung:
    - Liefert die Länge der Liste
  - Rückgabewert
    - Ganze Zahl größer oder gleich 0 als Java Integer
  - Fehlerbehandlung:
    - keine

- **insert: list × pos × elem → list**
  - Signatur: *#insert(pos, elem)*
  - Beschreibung:
    - Fügt das Element an die spezifizierte Position
    - Ist die Position bereits belegt, wird das belegende Element und alle darauf folgenden Elemente um eine Position höher verschoben, anschließend wird *elem* an die Stelle gespeichert
  - Rückgabewert
    - keiner (Mutator)
  - Fehlerbehandlung
    - *pos* muss größer als 0 und kleiner als *laenge()* sein, ist dies nicht gegeben, wird die Liste nicht verändert

- **delete: list × pos → list**
  - Signatur: *#delete(pos)*
  - Beschreibung:
    - Löscht das Element an der Stelle *pos*

        - Alle Elemente, die an Positionen höher *pos* liegen werden um eine Position nach unten verschoben
    - Rückgabewert:
        - keiner (Mutator)
    - Fehlerbehandlung:
        - *pos* muss größer als 0 und kleiner als *laenge()* sein, ist dies nicht gegeben, wird die Liste nicht verändert

- **find: list × elem → pos**
    - Signatur: *#find(elem)*
    - Beschreibung:
        - Findet die erste Instanz von *elem* in der Liste
    - Rückgabewert
        - Die Position der ersten Instanz von *elem* in der Liste als Java Integer
        - 0, falls *elem* in der Liste nicht gefunden werden konnte
    - Fehlerbehandlung
        - keine

- **retrieve: list × pos → elem**
    - Signatur: *#retrieve(pos)*
    - Beschreibung:
        - Liefert das Element an der Position *pos* zurück
        - Das Element wird aus der Liste entfernt
    - Rückgabewert:
        - Das Element an der spezifizierten Position als Java Integer
    - Fehlerbehandlung:

        - *pos* muss größer als 0 und kleiner als *laenge()* sein, ist dies nicht gegeben, wird *null* zurückgegeben

- **concat: list × list → list**
    - Signatur: *#concat(list)*
    - Beschreibung:
        - Fügt die Elemente der Liste die als Argument übergeben wurde unter Erhaltung der Reihenfolge in die bestehende Liste ein
        - Wird die Liste mit einer leeren Liste konkateniert, wird die Liste nicht verändert
        - Wird eine leere Liste mit einer befüllten Liste konkateniert, werden alle Elemente aus der übergebenen Liste in die leere Liste geschrieben
    - Rückgabewert:
        - keiner (Mutator)
    - Fehlerbehandlung:
        - keine

JUnit-Tests

- Dateiname: *adt_liste_test.jar*
- Die Tests prüfen, ob die Schnittstellen erfüllt werden, dies impliziert insbesondere die spezifizierte Fehlerbehandlung

ADT Stack

Vorgaben / Anmerkungen

- Technisch

  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 := Ein Element (ganze Zahl), das auf den Stack gestapelt wird. Realisiert als Java Integer
- stack := Ein Stack (Stapel) vom hier spezifizierten Typ ADT Stack

Operationen

- **createS: Ø → stack**
  - Signatur: *#createS()*
  - Beschreibung:
    - Erzeugt einen leeren Stack vom Typ ADT Stack
  - Rückgabewert:
    - keiner (Mutator)
  - Fehlerbehandlung:
    - keine

- **push: stack × elem → stack**
  - Signatur: *#push(elem)*
  - Beschreibung:
    - Legt das Element an die oberste Position im Stack

  - Rückgabewert
    - keiner (Mutator)
  - Fehlerbehandlung:
    - keine

- **pop: stack → stack**
  - Signatur: *#pop()*
  - Beschreibung:
    - Entfernt das Element an der höchsten Position im Stack
  - Rückgabewert:
    - keiner (Mutator)
  - Fehlerbehandlung:
    - Ist der Stack leer, wird *null* zurückgegeben

- **top: stack → elem**
  - Signatur: *#top()*
  - Beschreibung:
    - Liefert das Element an der höchsten Position im Stack, ohne es jedoch zu entfernen
  - Rückgabewert:
    - Das Element an der höchsten Position im Stack, sofern Elemente auf dem Stack sind
  - Fehlerbehandlung:
    - Ist der Stack leer, wird *null* zurückgegeben

- **isEmptyS: stack → bool**
  - Signatur: *#isEmptyS()*
  - Beschreibung:
    - Prüft, ob der Stack leer ist
  - Rückgabewert:
    - *true*, falls der Stack leer ist
    - *false*, falls Elemente auf dem Stack abgelegt sind
  - Fehlerbehandlung:
    - keine

- **equalS: stack × stack → bool**
  - Signatur: *#equalS(stack)*
  - Beschreibung:
    - Prüft, ob zwei Stack strukturell identisch sind
    - Strukturelle Identität ist gegeben genau dann, wenn die Anzahl und Reihenfolge der Elemente auf beiden Stacks identisch ist
    - Es bietet sich an, *equalS(stack)* mittels dem Vergleich der Rückgabewerte von wiederholten *pop()* und *top()* Aufrufen über beiden Stacks zu realisieren. Hierzu müssten ggf. Kopien der beiden Stacks erstellt werden, damit nach dem Vergleich weiter mit den Daten gearbeitet werden kann
    - Wird ein leerer Stack mit einem anderen leeren Stack verglichen, so ist strukturelle Identität gegeben
  - Rückgabewert:
    - *true*, falls alle Elemente beider Stacks identisch sind
    - *false*, wenn für mindestens ein verglichenes Element gilt, dass es nicht identisch ist mit dem Element aus dem anderen Stack
  - Fehlerbehandlung:
    - keine

- **reverseS: stack → stack**
  - Signatur: *#reverseS()*
  - Beschreibung:
    - Kehrt die Reihenfolge der Elemente im Stack um
    - Gemäß den Anforderungen dürfen die Elemente nicht in vertauschter Reihenfolge kopiert werden, es ist folglich nötig, die Position und Laufrichtung des Zeigers auf das höchste Element zu manipulieren.
    - *reverseS()* hat keinen Effekt bei einem leeren Stack
    - *reverseS()* hat keinen Effekt bei einem Stack mit nur einem Element
  - Rückgabewert:
    - keiner (Mutator)
  - Fehlerbehandlung
    - keine

JUnit-Tests

- Dateiname: *adt_stack_test.jar*
- Die Tests prüfen, ob die Schnittstellen erfüllt werden, dies impliziert insbesondere die spezifizierte Fehlerbehandlung

ADT Queue

Vorgaben / Bemerkungen

- Technisch

  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 ur 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 := Ein Element (ganze Zahl), das auf den Stack gestapelt wird. Realisiert als Java Integer
- queue := Ein Queue (Schlange) vom hier spezifizierten Typ ADT Queue

Operationen

- **createQ: Ø → queue**
  - Signatur: *#createQ()*
  - Beschreibung:
    - Erzeugt einen leeren ADT Queue
  - Rückgabewert:
    - keiner
  - Fehlerbehandlung
    - keiner

- **enqueue: queue × elem → queue**
  - Signatur: *#enqueue(elem)*
  - Beschreibung:
    - Fügt das Element hinten in den Queue ein
  - Rückgabewert:
    - keiner (Mutator)
  - Fehlerbehandlung:
    - keine

- **dequeue: queue → queue (Mutator)**
  - Signatur: *#dequeue()*
  - Beschreibung:
    - Löscht das vorderste Element aus dem Queue
  - Rückgabewert:
    - keiner (Mutator)
  - Fehlerbehandlung:
    - Ist der Queue beim Aufruf von *dequeue()* leer, wird *null* zurückgegeben

- **front: queue → elem**
  - Signatur: *#front()*
  - Beschreibung:
    - Liefert das vorderste Element aus dem Queue zurück
    - Das Element wird nicht aus dem Queue entfernt (analog zu *top()* beim ADT Stack)
  - Rückgabewert:
    - Das vorderste Element aus dem Queue als Java Integer

- Fehlerbehandlung:
  - Ist der Queue leer, wird *null* zurückgegeben

- **isEmptyQ: queue → bool**
  - Signatur: *isEmptyQ()*
  - Beschreibung:
    - Prüft, ob der Queue leer ist
  - Rückgabewert:
    - *true*, falls der Queue leer ist
    - *false*, falls der Queue mindestens ein Element enthält

- **equalQ: queue × queue → bool**
  - Signatur: *#equalQ(queue)*
  - Beschreibung:
    - Prüft, ob der übergebene Queue strukturell identisch mit dem Queue ist, an welchem die Methode gerufen wird
    - Strukturelle Identität ist gegeben genau dann, wenn für jede Position aus beiden Queues gilt, dass die Elemente identisch sind.
    - Es bietet sich an eine funktional ähnliche Implementation wie bei ADT Stack vorgeschlagen durchzuführen
    - Leere Queues sind zueinander strukturell identisch
  - Rückgabewert:
    - *true,* falls beide Queues strukturell identisch sind
    - *false,* falls für mindestens ein verglichenes Element gilt, dass es nicht identisch ist zu dem Element aus dem anderen Queue
  - Fehlerbehandlung:
    - keine

JUnit-Tests

- Dateiname: *adt_queue_test.jar*
- Die Tests prüfen, ob die Schnittstellen erfüllt werden, dies impliziert insbesondere die spezifizierte Fehlerbehandlung

ADT Array

Vorgaben / Bemerkungen

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

Objektmengen

- pos := Die Position innerhalb des Arrays als natürliche Zahl größer oder gleich 1. Realisiert als Java Integer
- elem := Das Element (ganze Zahl) innerhalb des Arrays. Realisiert als Java Integer
- array := Ein Array von dem hier spezifizierten Typ ADT Array

Operationen

- **initA: Ø → array**
  - Signatur: *#initA()*
  - Beschreibung:
    - Erzeugt ein leeres ADT Array
  - Rückgabewert
    - keiner
  - Fehlerbehandlung:
    - keine

- **setA: array × pos × elem → array**
  - Signatur: *#setA(pos, elem)*
  - Beschreibung:
    - Speichert an der spezifizierten Position ein neues Element
    - Falls sich an der Position bereits ein Element befindet, wird dieses durch das neue Element ersetzt
  - Rückgabewert:
    - keiner (Mutator)
  - Fehlerbehandlung
    - *pos* darf nicht negativ und nicht größer als *lengthA()* sein, ist dies nicht gegeben, wird das Array nicht verändert

- **getA: array × pos → elem**
  - Signatur: *#getA(pos)*
  - Beschreibung:
    - Liefert das Element an der Position *pos* zurück

- Rückgabewert:
  - Das Element an der spezifizierten Stelle als Java Integer
- Fehlerbehandlung:
  - pos darf nicht negativ und nicht größer als lengthA() sein, ist dies nicht gegeben, wird *null* zurückgegeben

- **lengthA: array → pos**
  - Signarur: *#lengthA()*
  - Beschreibung:
    - Liefert die Länge des ADT Arrays
  - Rückgabewert:
    - Die Länge des Arrays als Java Integer
  - Fehlerbehandlung:
    - keine

- **equalA: array × array → bool**
  - Signatur: *#equalA(array)*
  - Beschreibung:
    - Prüft, ob das übergebene Array strukturell identisch ist mit dem Array, auf dem die Methode gerufen wurde
    - Strukturelle Identität ist gegeben genau dann, wenn die Anzahl und Reihenfolge der Elemente in beiden Arrays identisch ist
    - Leere Arrays sind zueinander strukturell identisch
  - Rückgabewert:
    - *true,* falls strukturelle Identität zwischen beiden Arrays besteht
    - *false,* wenn für mindestens ein Element an einer identischen Position in beiden Arrays keine Identität besteht

- Fehlerbehandlung:
  - keine

JUnit-Tests

- Dateiname: *adt_array_test.jar*
- Die Tests prüfen, ob die Schnittstellen erfüllt werden, dies impliziert insbesondere die spezifizierte Fehlerbehandlung

ADT Btree

Vorgaben / Bemerkungen

- 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 := Das Element (ganze Zahl) innerhalb eines Knoten des Baumes. Realisiert als Java Integer

- btree := Ein Knoten oder die Verkettung von Knoten. Ein Knoten wird definiert durch das 4-Tupel *{leftSuc,elem,rightSuc,high}*, wobei *elem* die zu speichernde Zahl darstellt und *leftSuc* sowie *rightSuc* vom Typ *btree* sind. Sie stellen Verweise auf den linken bzw. rechten Nachfolger dar. Hat ein Knoten keinen Nachfolger, ist der Wert von *btree* mit *null* zu setzen.
- high := Bezeichnet die Tiefe in der sich der Knoten relativ zum gesamten Baum befindet. Beim ersten Knoten des Baumes ist die Tiefe 0. Realisiert als Java Integer.

Operationen

- **initBT: Ø → btree**
  - Signatur: *#initBT()*
  - Beschreibung:
    - Erzeugt eine neuen Baum, bestehend aus einem Knoten
    - Der Wert von *high* wird mit 0 initialisiert
    - Die beiden Nachfolger sowie das gespeicherte Element werden mit *null* initialisiert
  - Rückgabewert:
    - keiner
  - Fehlerbehandlung:
    - keine

- **setLeftSuc: btree × btree → btree**
  - Signatur: *#setLeftSuc(btree)*
  - Beschreibung:
    - Setzt ein Element vom Typ *btree* als linken Nachfolger
  - Rückgabewert:
    - keiner (Mutator)

- Fehlerbehandlung:
  - Gemäß den Anforderung muss gelten, dass *getVal(btree)* von dem Knoten, bei welchem der Nachfolger gesetzt werden soll größer ist als *getVal(btree)* vom Knoten, der eingefügt werden soll. Ist dies nicht erfüllt, wird der Knoten nicht als Nachfolger gesetzt

- **setRightSuc: btree × btree → btree**
  - Signatur: *#setRightSuc(btree)*
  - Beschreibung:
    - Setzt ein Element vom Typ *btree* als rechten Nachfolger
  - Rückgabewert:
    - keiner (Mutator)
  - Fehlerbehandlung:

    Gemäß den Anforderung muss gelten, dass *getVal(btree)* von dem Knoten, bei welchem der Nachfolger gesetzt werden soll kleiner ist als *getVal(btree)* vom Knoten, der eingefügt werden soll. Ist dies nicht erfüllt, wird der Knoten nicht als Nachfolger gesetzt

- **setHigh: btree × high → btree**
  - Signatur: *#setHigh(high)*
  - Beschreibung:
    - Setzt die Tiefe des Knoten auf den übergebenen Wert
  - Rückgabewert:
    - keiner
  - Fehlerbehandlung:
    - Ist *high* negativ, wird der Wert nicht gesetzt

- **setVal: btree × elem → btree**
  - Signatur: *#setVal(elem)*
  - Beschreibung:
    - Speichert den übergebenen Wert im Knoten

- Rückgabewert:
  - keiner
- Fehlerbehandlung:
  - keine

- **getLeftSuc: btree → btree**
  - Signatur: *#getLeftSuc()*
  - Beschreibung:
    - Liefert den linken Nachfolger für einen gegebenen Knoten
    - Hat der Knoten keinen Nachfolger, wird *null* zurückgeliefert
  - Rückgabewert
    - Der linke Nachfolger des Knotens als *btree*
  - Fehlerbehandlung:
    - keine

- **getRightSuc: btree → btree**
  - Signatur: *#getRightSuc()*
  - Beschreibung:
    - Liefert den rechten Nachfolger für einen gegebenen Knoten
    - Hat der Knoten keinen Nachfolger, wird *null* zurückgeliefert
  - Rückgabewert
    - Der rechte Nachfolger des Knotens als *btree*
  - Fehlerbehandlung:
    - keine

- **getHigh: btree → high**
  - Signatur: *#getHigh()*
  - Beschreibung:
    - Liefert den Wert für die Tiefe des Knoten
    - Ist bei dem Knoten kein Wert gesetzt, wird *null* zurückgegeben
  - Rückgabewert:
    - Die Tiefe des Knoten als Java Integer

- Fehlerbehandlung:
  - keine

- **getVal: btree → elem**
  - Signatur: *#getVal()*
  - Beschreibung:
    - Liefert den gespeicherten Wert für einen gegebenen Knoten
    - Ist in dem Knoten kein Wert gespeichert, wird *null* zurückgegeben
  - Rückgabewert
    - Der im Knoten gespeicherte Wert als Java Integer

- **isEmptyBT: btree → bool**
  - Signatur: *#isEmptyBT()*
  - Beschreibung:
    - Prüft, ob der Baum leer ist
    - Ein Baum ist leer genau dann, wenn *getLeftSuc(), getRightSuc()* und *getVal() null* zurückgeben
  - Rückgabewert:
    - *true*, falls der Baum leer ist
    - *false,* wenn der Baum nicht leer ist

- **equalBT: btree × btree → bool**
  - Signatur: *#equalBT(btree)*
  - Beschreibung:
    - Prüft, ob der übergebene ADT Btree und der ADT Btree, auf welchem die Methode gerufen wird strukturell identisch sind
    - Strukturelle Identität ist gegeben genau dann wenn für jeden Knoten beider Bäume gilt, dass *getVal(), getHigh(), getLeftSuc()* und *getRightSuc()* identisch sind
  - Rückgabewert:
    - *true,* falls beide Bäume strukturell identisch sind
    - *false,* wenn für mindestens einen Knoten strukturelle Identität nicht gilt

    - Fehlerbehandlung:
        - keine

- **print: btree × filename → png**
    - Signatur: *#print()*
    - Beschreibung:
        - Verwendet die *digraph* Bibliothek, um den gesamten Baum als Bilddatei zu speichern
        - Der benötigte Dateipfad *filename* (siehe semantische Struktur) kann in die Methode Hardcoded werden.
    - Rückgabewert:
        - keiner, allerdings wird erwartet, dass eine Datei am spezifizierten Ort erstellt wurde
    - Fehlerbehandlung:
        - Bei I/O Fehlern o.ä. soll das Programmverlauf nicht gestört werden. Etwaige Fehler sollen zumindest keinen Programmabbruch oder ein Blockieren verursachen. Optional können Fehler mittels Exceptionhandling behandelt werden.

JUnit-Tests

- Dateiname: *adt_btree_test.jar*
- Die Tests prüfen, ob die Schnittstellen erfüllt werden, dies impliziert insbesondere die spezifizierte Fehlerbehandlung.