Aufgabe a5x1

Funktionsweise:

- Karten werden solange aus dem Kartenstapel gezogen und in einen Stack hinzugefügt, bis die gesuchte Karte gefunden ist,
  oder
  maximal 52 Karten gezogen wurden (Sicherheitsmaßnahme)
- Wurde die gesuchte Karte gefunden, werden alle gespeicherte Karten in umgekehrten Reihenfolge nach dem LIFO-Prinzip (Last In First Out) ausgegeben.
- Wurde die Karte nicht gefunden, wird ein Assert-Error ausgelöst.

Datenorganisation:

- **Ein Stack wurde gewählt**, da der Stack prinzipiell nach dem LIFO-Prinzip funktioniert.
- **Die Verwendung einer Do-While-Schleife** lässt sich damit begründen, dass mindestens eine Karte gezogen werden muss um zu überprüfen, ob es sich um die gesuchte Karte handelt.
- **Für die Ausgabe der gespeicherten Karten** ist eine While-Schleife geeignet, da man nicht weiß, wann der Stack leer ist bzw. nach wie vielen Iterationen.

Sicherheit:

- Es wird überprüft, ob die Referenzen *deck* und *cardToCheck* nicht null sind.
- Es sind maximal 52 Zügen erlaubt, da ein Kartenstapel 52 Karten hat.

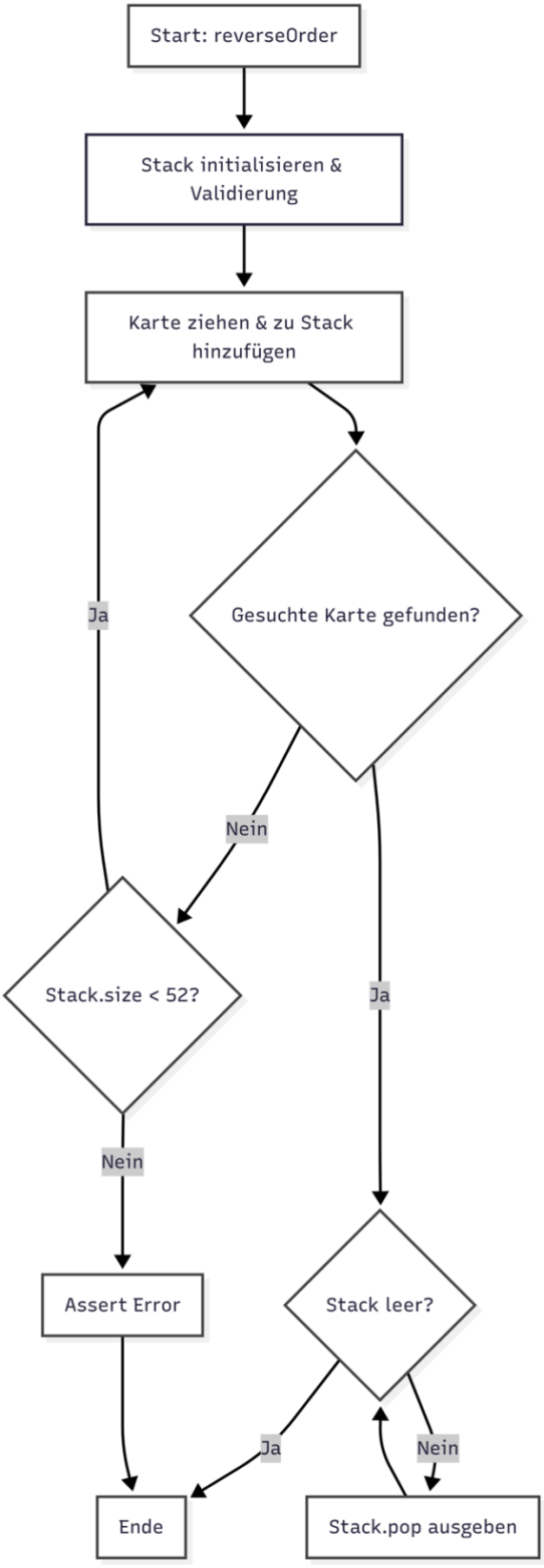

Abb. 1: *Flussdiagramm der Methode* reverseOrder

Aufgabe a5x2

Funktionsweise:

- Die Karten aus dem übergebenen Array über Karten werden in Set gespeichert.
- Danach wird ein neues Array über Karten mit der Größe erstellt, die der Anzahl der Elemente in Set entspricht. Gleichzeitig wird es mit allen Elementen aus dem Set ausgefüllt.
- Die “bereinigte” Array wird zurückgegeben

Datenorganisation:

- **Es wurde ein Set gewählt**, da ein Set Duplikate nicht erlaubt und diese automatisch ignoriert werden. Konkret wurde ein HashSet verwendet, da kein Bedarf besteht, die Karten in derselben Reihenfolge zurückzugeben, und ein HashSet ist schneller als ein TreeSet.

Sicherheit:

- Der Aufruf der Methode *isKartenNull* gewährleistet, dass das Array nicht null ist und keine Karte im Array einen null Wert hat.

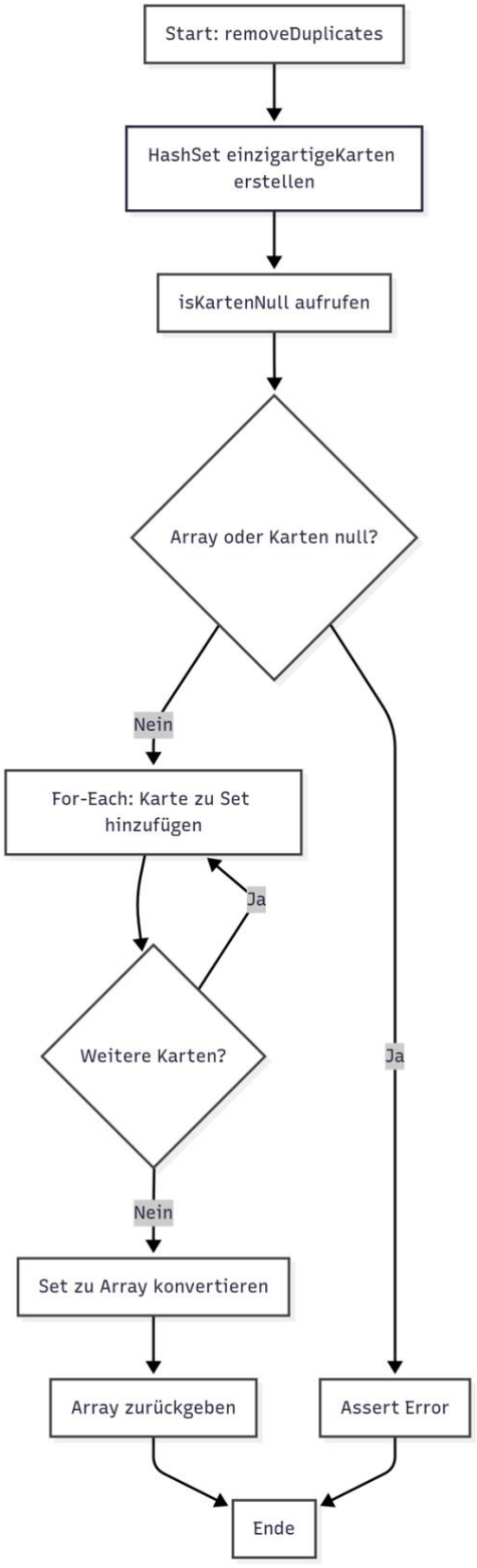

Abb. 2: *Flussdiagramm der Methode* removeDuplicates

Aufgabe a5x3

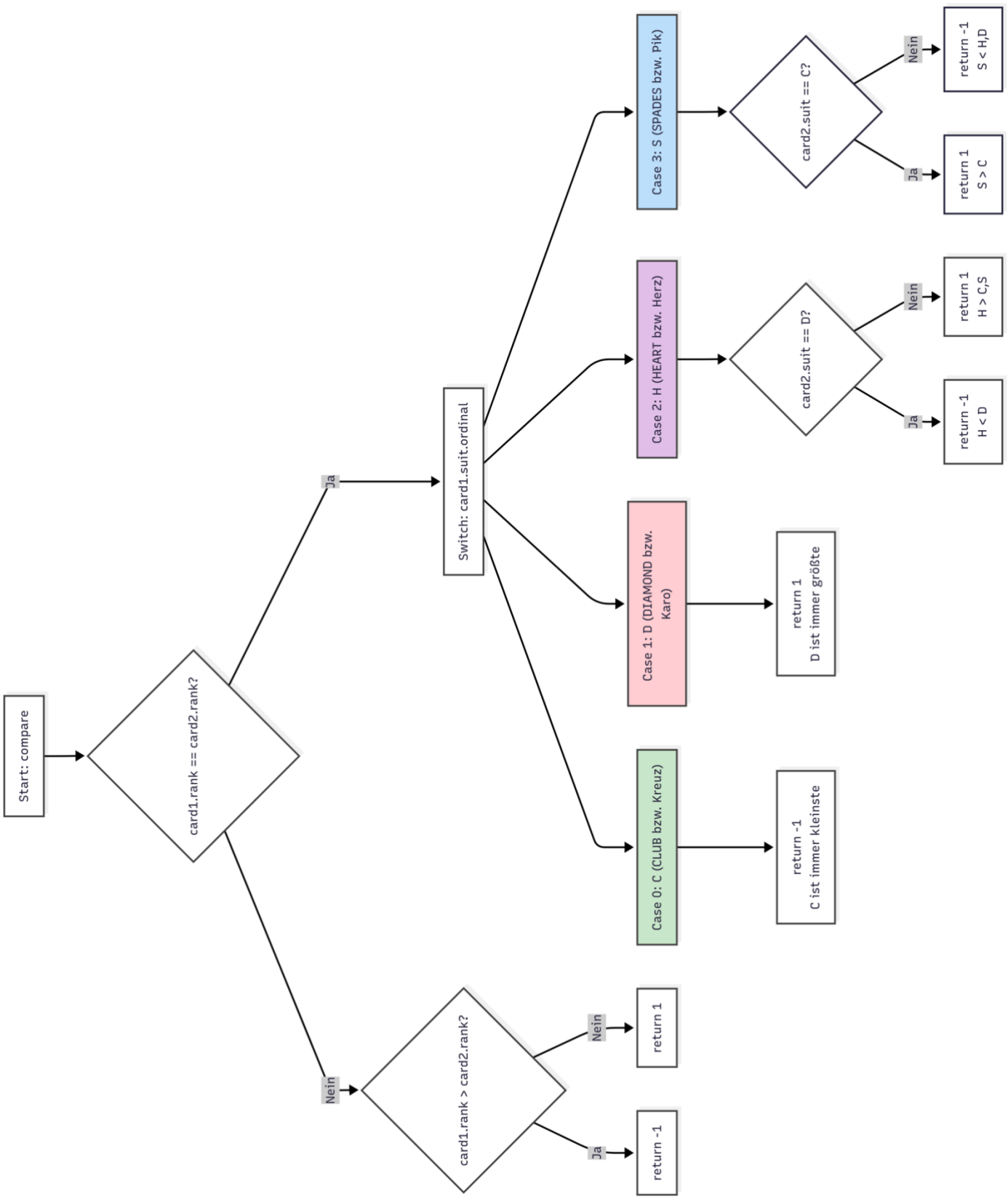

Abb. 3: *Flussdiagramm des Comparators* UsualOrder

Aufgabe a5x4

Funktionsweise

- Existiert kein HashSet in deer Map mit dem entsprechenden Rank als Key der übergebenen Karte, so wird ein neues HashSet für diesen Key in der Map erstellt und die Karte unter diesem Key in HashSet gespeichert.
- Sobald in der HashMap unter einem Key ein Drilling vorliegt, so werden alle Karte mit diesem Rank als Array zurückgegeben und das entsprechende HashSet geleert ausgegeben.

Datenorganisation:

- Für die Lösung dieser Aufgabe wurde eine HashMap mit dem Rank der Karten als Key und ein HashSet gewählt.

  Begründung der Auswahl:

  - *Map ist besonders gut geeignet, weil…*

    - Sie sehr effizient für diese Aufgabe ist, da durchlauf über alle Elemente nicht nötig ist. → **Map ermöglicht direkten Zugriff über den Key.**
    - HashMap die Karten ungeordnet speichert, wodurch die Bearbeitung schneller wird als bei geordneten Strukturen.
    - Jedem Rank entsprechende Karten zugewiesen werden, deren Anzahl direkt über den Key bestimmt werden kann.

  - *Set wurde gewählt, weil…*

    - Karten nicht doppelt vorkommen können, da **Set doppelte Karten ausschließt.**
    - schneller und effizienter ist, wobei die Reihenfolge keine Rolle spielt.
    - Die Kombination HashMap<Rank, HashSet<Card>> sowohl schnellen Zugriff als auch das Vermeiden von Duplikaten ermöglicht.

Es sind auch andere Maps denkbar, beispielsweise Rank als Key und eine Liste über Karten. In diesem Fall könnte eine Karte doppelt vorkommen, was wahrscheinlich zu einem Fehler führen kann. Eine Map mit Rank der Karten als Key und einer Klasse mit den notwendigen Parametern wäre auch eine Lösung.

Sicherheit:

- Mit dem assert Anweisung wird überprüft, dass die übergebene Karte nicht null ist.