Analyse der ADTs und Datenstrukturen

1. Die Nutzer sind in einer **Dictionary/Map (ADT)** gespeichert. Die Datenstruktur wird als Hashtabelle implementiert, da Python **Hashes für (dict)** nutzt, außerdem ist **user_id** der Key und das Value das **User-Objekt**.

2. Zugriff auf einen Nutzer über ein Dictionary: **O(1)** amortisiert.
   get_groupmembers() durchläuft alle Nutzer und ist daher **O(n)**.

3. Die Auktionen werden in einem **Dictionary/Map (ADT)** gespeichert, da die Klasse Auctions direkt von dict erbt. Die implementierte Datenstruktur ist eine Hashtabelle, weil Python-dict Hashes verwendet und die auction_id der Key ist.

4. In systemmessages.py wird der **ADT Queue (FIFO)** verwendet, da Nachrichten in der Reihenfolge ihres Eintreffens verarbeitet werden. Implementiert ist dieser ADT als **Python-Liste (ArrayList)**, da Einfügen mit **append()** und Entfernen mit **pop(0)** realisiert wird.

5. Die Reihenfolge der Gebote wird in der Klasse Auction über das Attribut **_bids_ordered** gespeichert. Dieses Attribut ist ein **Stack (LIFO)**. Jedes neue Gebot wird oben auf dem Stack gelegt, wodurch das neueste Gebot immer ganz oben liegt und schnell abrufbar ist. Die Methode **get_last_bid()** liefert das zuletzt abgegebene Gebot, indem sie das oberste Stackelement zurückgibt. Das Stack speichert damit die Gebote in der **korrekten zeitlichen Reihenfolge**, von ältestem (unten) bis neuestem (oben).

6. Der Trie speichert Produktnamen zeichenweise in einer Baumstruktur.
   Die Methode **insert()** legt jeden Buchstaben als Pfad im Trie an.
   Mit **search(prefix)** kann effizient nach allen Wörtern gesucht werden, die mit einem bestimmten Präfix beginnen.
   Die rekursive Methode **_find_words()** sammelt alle passenden Wörter.
   Durch diese Struktur ist die Auto-Vervollständigung sehr schnell, da die Laufzeit nur von der Länge des Präfixes abhängt **(O(k)).**

7. Die Autoergänzung kann auch mittels eines AVL-Baums erfolgen. Über die Methode **find_most_likely_words()** durchsucht der AVL-Baum alle Knoten, deren Schlüssel im Bereich des gesuchten Präfixes liegen, und sammelt die passenden Wörter. In **show_suggestions()** muss dafür statt des Tries die AVL-Methode verwendet werden. Laufzeittechnisch ist der Trie effizienter: er findet Präfixe in **O(k)**, während der AVL-Baum **O(log n + m)** benötigt. Für reine

Präfixsuche ist der Trie daher schneller, der AVL-Baum aber bietet stabile Leistung und sortierte Einordnung aller Wörter.