Analyse der ADTs und Datenstrukturen

Teil 1:

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 AVLMethode 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.

Teil 2:

1 .
Zum Verwalten der bids wird ein **Heap und ein Stack** benutzt.

Der **Heap** wird als Min-Heap Array in _users_bidding implementiert, wobei der user mit dem **höchsten bid die root bildet** und mit dem **negativen amount von bid** sortiert wird. Dies wird getan, da im Min-Heap immer der kleinste Schlüssel als Root dient und somit der höchste bid einfach eingesehen werden kann.
Die Laufzeiten der benötigten Funktionen wären hierbei **maximal log(n)** bei add() und removeMin(), wobei add() **in der Praxis** häufig auf eine **konstante Laufzeit** tritt, wenn percolateUp() nicht angewendet werden muss.
Somit ist der Heap für alle relevanten Funktionen **entweder gleich gut oder besser als ein AVL-Baum** und zudem einfacher zu implementieren.

Der **Stack** wird verwendet, um eine sortierte Version der bids zu besitzen.

2.
Durch einen Max-Heap ist der Abruf der aktivsten Auktion linear O(1), da diese den Root bildet. Das Einfügen einer neuen und entfernen der aktivsten Aktion ist O(log n) mit Heapify.
Wenn jedoch eine Auktion verändert wird, muss diese potentiell seine Position verändern, wobei dies ohne Hashtabelle eine lineare Suche O(n) verlangt, um die Auktion zuerst in der Liste zu finden. Mit einer Hashtabelle, die die Position speichert, kann die Suche umgangen werden, und es muss nur Heapify O(log n) angewendet werden. Danach muss noch linear die Hashtabelle geupdated werden. Wenn sich die Werte im Heap häufig ändern, lohnt es sich, eine Hashtabelle zu verwenden. Sollte dies nicht der Fall sein, kann man den Max-Heap auch ohne Hashtabelle verwenden.