H6.1 Zyklische verkettete Listen

zu 2.: Gebt die Laufzeitkomplexitäten eurer beiden Implementierungen an (jeweils in Abhängigkeit von der Anzahl *n* der Knoten in der zyklischen Liste.

- Die Funktion delete_next_node(node) löscht den Nachfolger des Elements, auf das node zeigt. Diese Funktion hat **konstante Laufzeit**, denn die Funktion besteht nur aus einer elementaren Operation, die prüft, ob der Knoten, auf den node zeigt, auf einen weiteren Nachfolger verweist (if node['next'] == node:). Wenn das der Fall ist, folgen im worst case zwei weitere elementare Operationen. Die bestehen darin, den Verweis unter dem Schlüssel „next“ des Nachfolger des Knotens, auf den node zeigt, in einer Variable zu speichern (x = node["next"]["next"]) und den Verweis dann an node am Schlüssel „next“ zu übergeben (node["next"] = x).

- Die Funktion delete_node(node) löscht das Element selbst, auf das node zeigt und hat **lineare Laufzeit in Abhängigkeit von der Anzahl *n* der Knoten**. Denn zusätzlich zu einer elementaren Operation, die prüft, ob der Knoten, auf den node zeigt, auf einen weiteren Nachfolger verweist (if node['next'] == node:), muss man denjenigen Knoten finden, der am Schlüssel „next“ auf den Knoten von node verweist, um den Knoten aus der Liste löschen zu können. In der Schleife (while find_before["next"] != node:) muss die find_before = find_before["next"] Operation im worst-case für n-Elemente der Liste durchgeführt werden, bevor der Vorgänger des Knotens gefunden wurde, auf den node zeigt. Hinzu kommt dann noch die konstante Operation find_before["next"] = node["next"] bei der dem gefundenen Vorgänger von node am Schlüssel „next“ der Verweis auf den ehemaligen Nachfolger von node zugewiesen wird.