Doubly Linked List - tidskompleksitet

Skemaer – til sammenligning

Skriv noget klogt her!

Indeholder nodes (objekter) som har referencer til andre nodes. Så fx. en node har en .next og en .prev. Det første objekt (hvis listen ikke er tom), ville være head. Så head.next ville være index(1), head.prev vil være null, der er ikke noget før den.
Samme med tail, så tail.prev ville være index(size()-2), da tail's index er size-1. Size er antallet af nodes i listen, tail.next vil være null da der ikke er noget efter den.

Hvis vi skal iterere igennem listen, fx. finde en node på i'te position, så kan vi enten starte fra head eller tail. I værste tilfælde ville vores i'te position være i midten af listen. Så der er lige langt fra head eller tail. Derfor vil big O højst være O(n/2), hvor n er længden på listen. Vi iterere højst igennem halvdelen af listen.

Doubly Linked List

Læs et element[1]	første	sidste	midterste	i'te	næste[2]
	$O(1)$	$O(1)$	$O(n/2)$	$O(n/2)$	$O(1)$
Find element [3]	eksisterer *usorteret liste*	eksisterer *sorteret liste*	eksisterer ikke *usorteret liste*	eksisterer ikke *sorteret liste*	
	$O(n)$	$O(n/2)$	$O(n)$	$O(n/2)$	
Indsæt nyt element	i starten	i slutningen	i midten	efter node	før node
	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Fjern element	første	sidste	i'te	efter node	før node
	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Byt om på to elementer	første og sidste	første og i'te	sidste og i'te	i'te og j'te	nodes
	$O(1)$	$O(n/2)$	$O(n/2)$	$O(n/2)$	$O(1)$

Disse scenarier går ud fra worst case.

[1] At læse et element er som regel det samme som at skrive nyt indhold i et eksisterende element
[2] Hvis vi allerede har fat i ét element i en datastruktur, kan vi måske læse det "næste" hurtigere end i+1'te
[3] Find et element med en bestemt værdi – alt efter om vi ved at listen er sorteret eller ej, og om elementet findes eller ej.