# 02. Datastrukturer

#### Bonus:

### Forelesning 2

For å unngå grunnleggende kjøretidsfeller er det viktig å kunne organisere og strukturere data fornuftig. Her skal vi se på hvordan enkle strukturer kan implementeres i praksis, og hva vi vinner på å bruke dem i algoritmene våre.

#### Pensum

- ☐ Innledningen til del III
- ☐ Kap. 10. Elementary data structures: Innl., 10.1 og 10.2
- $\square$  Kap. 11. Hash tables: Tom. 11.3.1
- $\square$  Kap. 16. Amortized analysis: Innl., 16.1 og s. 460–463 (tom. at most 3)

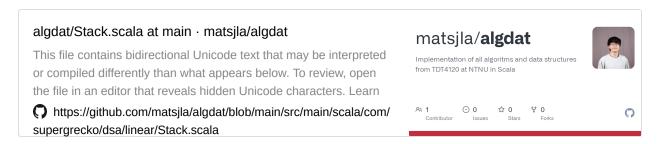
### Læringsmål

- $[B_1]$  Forstå stakker og  $k \not e r$
- [B<sub>2</sub>] Forstå lenkede lister
- [B<sub>3</sub>] Forstå direkte adressering og hashtabeller
- B<sub>4</sub> Forstå konfliktløsing ved kjeding
- [B<sub>5</sub>] Kjenne enkle hashfunksjoner
- [B<sub>6</sub>] Kjenne perfekt hashing; O(1) søk for statiske data
- $[B_7]$  Kunne definere amortisert analyse
- [B<sub>8</sub>] Forstå aggregert analyse
- B<sub>9</sub>] Forstå dynamiske tabeller



En Stack er en lineær last-in-first-out (LIFO) datastruktur. En stack fungerer som en stabel hvor elementene som legges på først, kommer ut sist. Relevant funksjonalitet på en Stack i pensum er:

- *StackEmpty* er stacken tom?
- ullet StackPush legg til et element på toppen av stacken i konstant tid O(1)
- StackPop fjern et element fra toppen av stacken i konstant tid O(1)

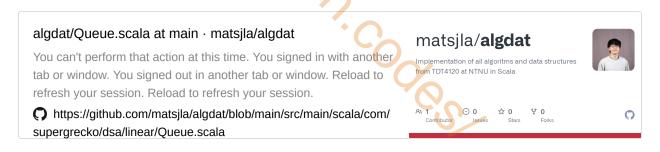


### Queue

En Queue er en lineær first-in-first-out (FIFO) datastrukturo. En Queue fungerer på lik måte som en Stack, bortsett fra at elementene som ble lagt på først, kommer ut først. Relevant funksjonalitet på en Queue fra pensum er:

- QueueEnqueue legg til et element i køen i konstant tid O(1)
- QueueDequeue ta det fremste elementet i køen i konstant tid O(1)
- QueueSize størrelsen på køen, som regel kun relevant hvis køen ikke backes av en resizable container.

Implementasjonen i pseudokode i pensum tar ikke hensyn til overflow eller underflow av køen. Implementasjon i Scala tar hensyn til dette.



### LinkedList

En lenket liste (i pensum, doubly-linked-list) er en liste som består av noder som er allokert tilfeldige steder i minnet som er koblet sammen via pekere. Det tar dermined lineær tid å slå opp på en gitt posisjon, men det er konstant tid for å sette inn eller slette elementer. Relevant funksjonalitet på en LinkedList fra pensum er:

- ListSearch søk igjennom en liste. Beste tilfelle på O(1) hvis første element er søkeelementet, ellers er det gjennomsnittslig og verste tilfelle på O(n).
- ListPrepend legg til en node i frontend av lista i konstant tid O(1).

- ListInsert legg til en ny node x foran en eksisterende node y i konstant tid O(1). Denne funksjonen trenger ikke et liste objekt da den kun operer på to noder.
- ListDelete slett en node x fra lista i konstant tid O(1)

#### 

### HashTable

En hashtabell er en relasjon eller mapping mellom en nøkkel på en verdi. En hashtabell implementeres som regel ved hjelp av en lineær datastruktur av LinkedList-er. Nøkkelen går igjennom en hash-funksjon som bestemmer hvilken bøtte verdien skal havne i. En god hash-funksjon vil ha så få kollisjoner som mulig som gjør innsetting til, og henting fra tabellen ta konstant tid.

I tilfelle hvor det er kollisjoner, lagres også noe ekstra data (gjerne nøkkelen eller annen data som kan gjenkjenne nøkkelen) for å løse kollisjoner. Dette resulterer i veldig simple algoritmer for å bruke hashtabellen. Relevant funksjonalitet på et HashTable fra pensum er:

- ChainedHashSearch finn verdien til en nøkkel i hashtabellen. Kjører i beste og gjennomsnittslig tilfelle i konstant tid O(1), men siden det kan være kollisjoner er verste tilfelle O(n).
- ChainedHashInsert sett inn gitt nøkkel og verdi i lista i konstant tid O(1).
- ChainedHashDelete fjern verdien til en nøkkel i hashtabellen. Kjører i beste og gjennomsnittslig tilfelle i konstant tid O(1), men siden det kan være kollisjoner er verste tilfelle O(n).



## DynamicTable

En dynamisk tabell er en liste eller et array som har evnen til å gro eller minke i størrelse basert på antall elementer i lista. Det betyr at lista kan bli (i teorien) uendelig stor siden vi kan gro lista når den nærmer seg full. Det er vanlig å bruke en større faktor enn n+1 for å gro tabellen for å unngå mange allokasjoner. Relevant funskjonalitet på et DynamicTable fra pensum er:

• TableInsert - legg til et nytt element i den dynamiske lista. Kjører beste og gjennomsnittslig tilfelle i konstant tid O(1), men siden lista må gro innimellom er verste tilfelle O(n).



## Amortisert arbeid

Kjøretiden for én enkelt operasjon er ikke alltid like informativt. I en dynamisk tabell kan en insert operasjon få lista til å gro og dermed bli O(n), men siden vi får en større grofaktor vil dette som regel ikke skje. Amortisert arbeid er aggregert analyse av et problem. Vi finner totalt arbeid og deler på antall operasjoner.

Average-case kommer dermed av forventet kjøretid for en algoritme, selv om den kan i noen tilfeller være mye tregere (som i DynamicTableInsert). Amortisert arbeid er snitt over operasjoner.