Sekvensielle datastrukturer

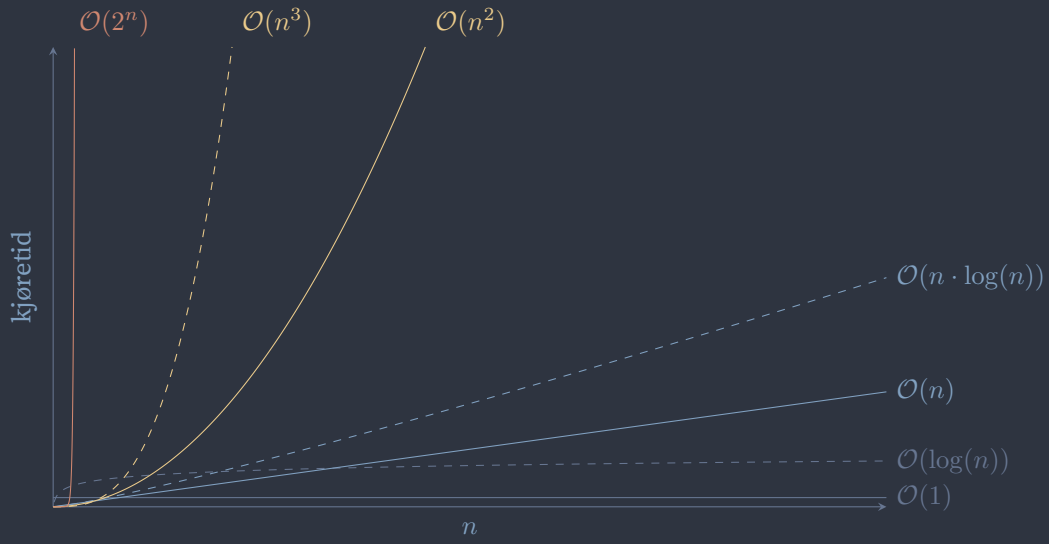
IN2010 – Algoritmer og datastrukturer

Lars Tveito

Institutt for informatikk, Universitetet i Oslo
larstvei@ifi.uio.no

Kjøretidskompleksitet

Vanlige uttrykk for kjøretidskompleksitet (graf)



Lister

Lister

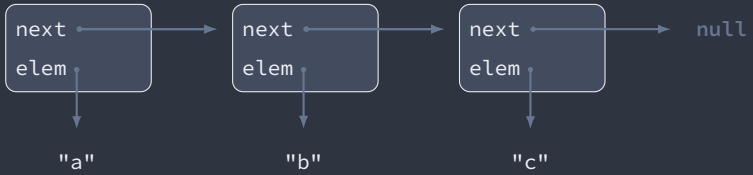
- En liste er en *samling* av elementer av *endelig* størrelse i en *bestemt ordning*
- Vi kan referere til elementene ved hjelp av en *indeks*
  - Vi definerer indeksen til et element e i en liste L som antall elementer som forekommer før e i L
  - Indeksen er et tall mellom 0 og $n - 1$ der n er antall elementer i L
- I en liste forventer vi å kunne:

Listeoperasjon	Beskrivelse
`Get`(L, i)	Hente ut elementet i L med indeks i
`Set`(L, i, e)	Erstatt elementet på indeks i med elementet e
`Add`(L, i, e)	Legg til elementet e i L slik at det får indeks i
`Remove`(L, i)	Fjern elementet med indeks i fra L

Lenkede lister

Noder

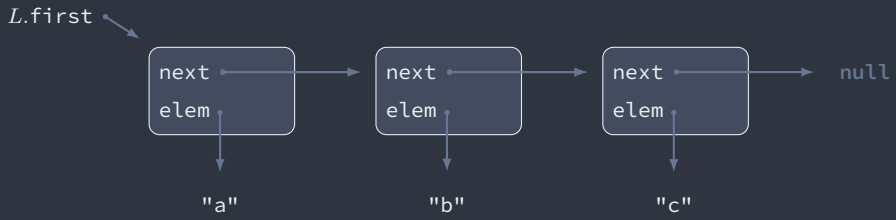
- En tilnærming til lister er å koble sammen *noder* i en *lenkestruktur*
- En node v holder på et element $v.\texttt{elem}$, og har en peker $v.\texttt{next}$ til *neste* node
- Posisjonen til en node er *relativ* til nodene som kommer før og etter




```
1 a ← new Node("a")
2 a.next ← new Node("b")
3 a.next.next ← new Node("c")
4 ···
```

Enkeltlenket liste

- Ved å knytte noder sammen med *neste*-pekere får vi en *enkeltlenket liste*
- Vi kan lage en liste L som et objekt med peker til den første noden L.`first`
- **`null`** representerer den tomme listen



Finne en node

- Ofte må vi løpe gjennom en liste ved å følge nestepekere
- Anta at vi vil finne noden med en gitt indeks i
  - Det gjøres ved å opprette en nodepeker `it` som initialiseres til å peke på det L.`first` peker på
  - Deretter følger vi i nestepekere

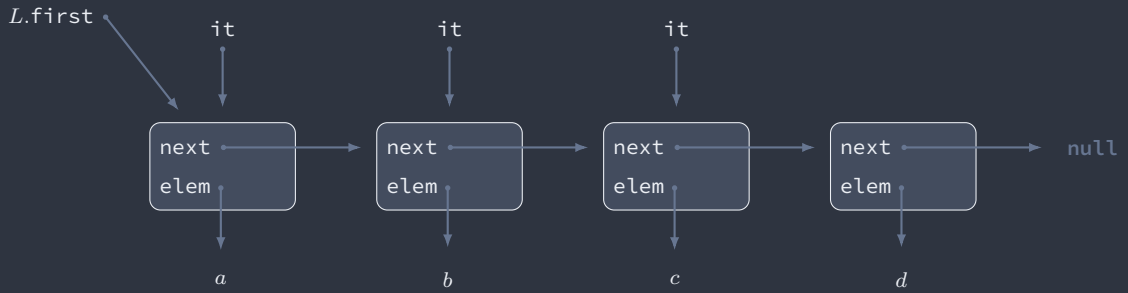
ALGORITHM: FINN NODEN PÅ EN GITT INDEKS

Input: En lenket liste L og en indeks i
Output: Noden med indeks i i L

```
1 Procedure NodeAt(L, i)
2     it ← L.first
3     for j ← 0 to i − 1 do
4         it ← it.next
5     return it
```

- Algoritmen er i $\mathcal{O}(n)$ der n er antall noder i listen
  - Dersom $i = n - 1$ løper vi til slutten av listen som krever $n - 1$ iterasjoner

Finne en node (illustrasjon)



$i = 2$

Sett inn etter

- Når vi setter inn etter en annen node v i en enkeltlenket liste må vi:
  - Lage en ny node u
  - La den nye noden u sin neste peke på det v.`next` peker på
  - Sette v.`next` til å peke på den nye noden u

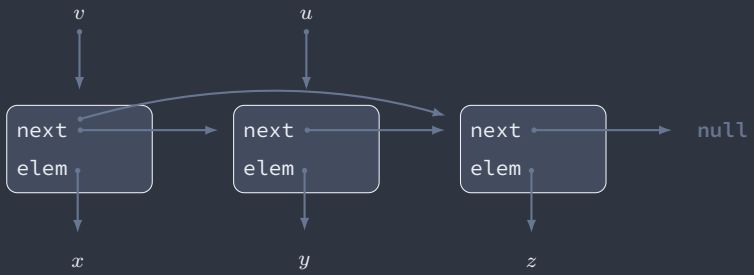
Algorithm: Innsetting etter en gitt node

Input: En node v i en lenket liste og et element e
Output: En ny node med elementet e er satt inn rett etter v

```
1 Procedure InsertAfter(v, e)
2     u ← new Node(e)
3     u.next ← v.next
4     v.next ← u
```

- Algoritmen er i $\mathcal{O}(1)$
  - Vi oppretter kun et nytt objekt og oppdaterer et par pekere
  - Innsetting etter v krever like mange steg *uavhengig* av hvor stor listen er

Sett inn etter (illustrasjon)



Fjern etter

- For å fjerne noden som kommer etter v må vi sette v.`next` til å peke på v.`next.next`
- Vi antar at v sin neste ikke er **`null`**

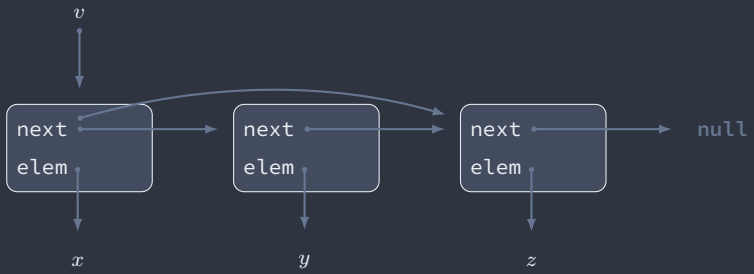
Algorithm: Sletting av noden etter en gitt node

`Input:` En node v i en lenket liste der v.`next` $\neq$ **`null`**
`Output:` Noden etter v fjernes og elementet returneres

```
1 Procedure RemoveAfter(v)
2     e ← v.next.elem
3     v.next ← v.next.next
4     return e
```

- Algoritmen er i $\mathcal{O}(1)$

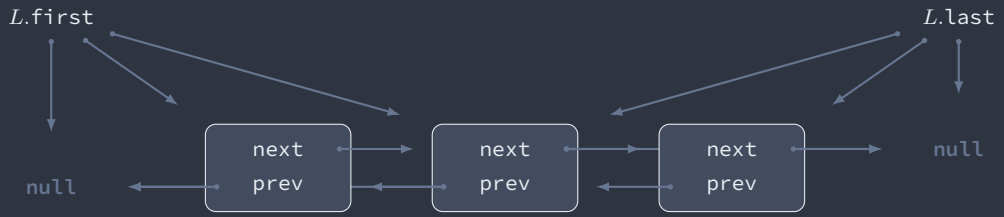
Fjern etter (illustrasjon)



Dobbeltlenket liste

- Ved å knytte noder sammen med *neste-* og *forrige-*pekere får vi en *dobbeltlenket liste*
- Ved å ha pekere i begge retninger er det mulig å sette inn en node *før* en annen
- Vi kan lage en dobbeltlenket liste L som et objekt med peker til den første noden L.`first` og den siste noden L.`last`
- En dobbeltlenket liste gjør både innsetting og sletting på begge ender av listen raskt og forholdsvis enkelt

Illustrasjon



Sett inn etter

- Når vi setter inn en node etter en annen node v i en dobbeltlenket liste må vi:
  - Lage en ny node u
  - Oppdatere pekere slik at den posisjoneres mellom v og v sin neste
- Her antar vi at v sin neste ikke er `null`

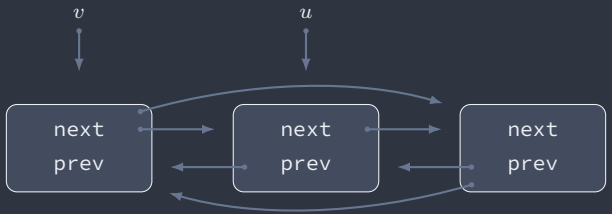
Algorithm: Innsetting etter en gitt node

Input: En node v i en dobbeltlenket liste og et element e
Output: En ny node med elementet e er satt inn rett etter v

```
1 Procedure InsertAfter(v, e)
2     u ← new Node(e)
3     u.prev ← v
4     u.next ← v.next
5     v.next.prev ← u
6     v.next ← u
```

- Algoritmen er i $\mathcal{O}(1)$
  - Det gjøres omtrent dobbelt så mange steg som for enkeltlenkede lister
  - Fremdeles er antall steg *uavhengig* av lengden på listen

Sett inn etter (illustrasjon)



Sett inn før

- Å sette inn en ny node *før* en annen node v er *enkelt* i en dobbeltlenket liste
- Det er det samme som å sette inn *etter* v.`prev`!
- Vi antar at v sin forrige ikke er **`null`**

ALGORITHM: INNSETTING FØR EN GITT NODE

Input: En node v i en dobbeltlenket liste og et element e
Output: En ny node med elementet e er satt inn rett før v

```
1 Procedure InsertBefore(v, e)
2 |   InsertAfter(v.prev, e)
```

- Algoritmen er i $\mathcal{O}(1)$
  - Den følger en peker og gjør et kall på en prosedyre som er i $\mathcal{O}(1)$

Fjern en node

- I en dobbeltlenket liste kan vi fjerne noden v *selv*
  - Vi trenger ikke noden foran, slik vi gjorde i en enkeltlenket liste
- Vi antar at hverken v sin forrige eller neste er `null`

Algorithm: Sletting av en gitt node

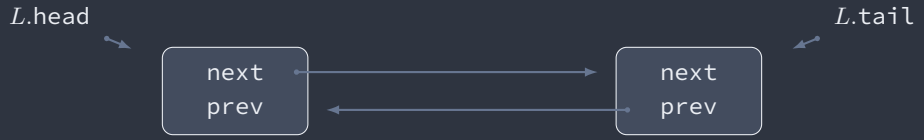
Input: En node v i en dobbeltlenket liste
Output: v er fjernet fra listen, og elementet returneres

```
1 Procedure RemoveNode(v)
2     v.prev.next ← v.next
3     v.next.prev ← v.prev
4     return v.elem
```

- Algoritmen er i $\mathcal{O}(1)$

Hode- og halenoder

- Så langt har vi ikke håndtert innsetting eller sletting i endene av listen
- Vi kan legge til spesielle tilfeller for dette
- En annen løsning er å heller la listen ha to *spesielle* noder; et *hode* og en *hale*
  - De holder ikke på noe element, og de fjernes aldri
  - De ligger alltid på hver sin ende av listen
- Da har alle noder som holder på et element både en forrige og en neste som ikke er `null`
- Den første noden er L.`head.next`, og den siste er L.`tail.prev`
- Listen er tom når L.`head.next` $= L$.`tail`



Lenkede lister *er* lister

Listeoperasjon	Implementasjon	$\mathcal{O}$
`Get`(L, i)	`NodeAt`(L, i)`.elem`	$\mathcal{O}(n)$
`Set`(L, i, e)	`NodeAt`(L, i)`.elem` $\leftarrow e$	$\mathcal{O}(n)$
`Add`(L, i, e)	`InsertBefore`(`NodeAt`$(L, i), e)$	$\mathcal{O}(n)$
`Remove`(L, i)	`RemoveNode`(`NodeAt`(L, i))	$\mathcal{O}(n)$

- Merk at kompleksiteten i alle kallene kommer fra `NodeAt`
  - Alle andre listeoperasjoner vi har sett er i $\mathcal{O}(1)$

Dynamiske arrayer

Arrayer

- Et array A av lengde N lagrer N elementer av samme type
- Ligger som en sammenhengende segment av minne
- Elementene aksesseres ved hjelp av en *indeks*
- Hvis $0 \leq i < N$, så
  - $A[i]$ gir det i-te elementet i $\mathcal{O}(1)$
  - $A[i] \leftarrow x$ setter (og overskriver) det i-te elementet til x i $\mathcal{O}(1)$
  - $|A|$ gir lengden på arrayet i $\mathcal{O}(1)$
    - Siden vi antok at A er av lengde N, så er $|A| = N$
- Et array har *fast størrelse*
- Det er den enkleste og raskeste datastrukturen, men ikke den mest fleksible

Dynamiske arrayer

- Et *dynamisk* array er et array som kan utvides ved behov
- Det dynamiske arrayet D kan implementeres med:
  - et *ordinært* array $D.A$ av fast størrelse $|D.A|$, og
  - et tall $D.n$ som angir antall elementer lagret
- (Det kan implementeres som et objekt eller struct med to felter)
- Det dynamiske arrayet kan implementere en *liste*
- Merk at `list` i Python og `ArrayList` i Java er implementert som dynamiske arrayer

Innsetting på slutten

- Når vi setter inn et element, så må vi først sikre at vi har kapasitet
  - Det vil si vi må sørge for at et er plass til *minst* ett element til
  - Vi trenger en prosedyre `EnsureCapacity` som sørger for dette
- Vi setter verdien på posisjon $D.n$ til å være verdien som ble gitt som argument
- Til slutt inkrementerer vi $D.n$

Algorithm: Insetting på slutten i et dynamisk array

Input: Et dynamisk array D og et element e
Output: Et dynamisk array med e lagt til på slutten av D

```
1 Procedure PushBack(D, e)
2     EnsureCapacity(D)
3     D.A[D.n] ← e
4     D.n ← D.n + 1
```

- Algoritmen er i $\mathcal{O}(1)$ hvis det er ledig kapasitet
  - Ellers har den samme kjøretidskompleksitet som `EnsureCapacity`

Sikre kapasitet

- Dersom $D.n$ er *mindre enn* $|D.A|$ gjør vi *ingenting*
- Hvis ikke, så oppretter vi et nytt array med dobbelt så mye plass
  - Det nye arrayet fylles opp med verdiene fra $D.A$
- Til slutt setter vi $D.A$ til å peke på det nye arrayet

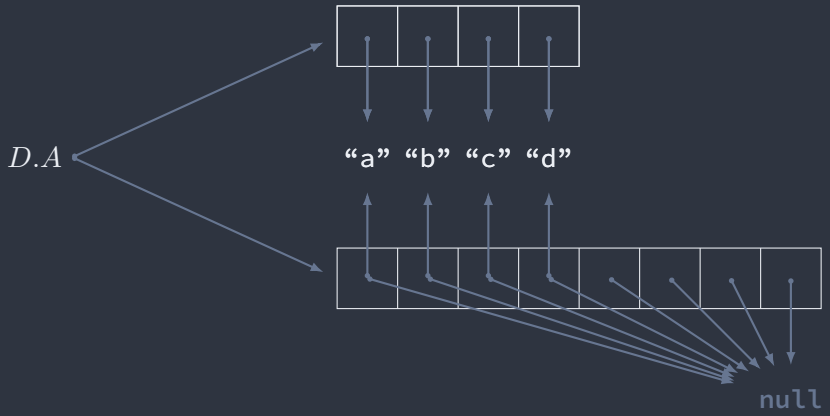
Algorithm: Doble kapasiteten til et dynamisk array ved behov

Input: Et dynamisk array D
Output: Et dynamisk array med plass til minst ett element til

```
1 Procedure EnsureCapacity(D)
2     if D.n < |D.A| then
3         return
4     A ← new array of size |D.A| · 2
5     for i ← 0 to D.n − 1 do
6         A[i] ← D.A[i]
7     D.A ← A
```

- Algoritmen er i $\mathcal{O}(1)$ hvis det er ledig kapasitet
- Hvis ikke opprettes et nytt array og elementene kopieres fra det gamle til det nye i $\mathcal{O}(D.n)$ tid

Sikre kapasitet (illustrasjon)



Sletting på slutten

- For å fjerne elementet på slutten må vi
  - senke $D.n$, og
  - sette elementet på posisjon $D.n$ til `null`
- Vi må anta at det dynamiske arrayet inneholder minst ett element

Algorithm: Sletting på slutten av et dynamisk array

Input: Et ikke-tomt dynamisk array D
Output: Det siste elementet fjernes og returneres

```
1 Procedure PopBack(D)
2     e ← D.A[D.n − 1]
3     D.A[D.n − 1] ← null
4     D.n ← D.n − 1
5     return e
```

- Algoritmen er i $\mathcal{O}(1)$

Legg til på indeks

- Når vi skal legge til et element i et dynamisk array må vi først gjøre plass til elementet
- Legg til et element på plass i, må alle elementer fra posisjon i til $D.n - 1$ flyttes ett hakk til høyre

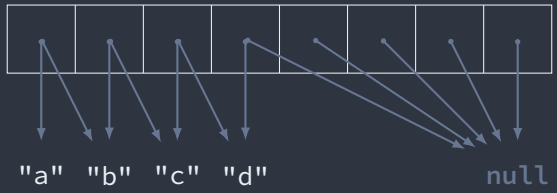
Algorithm: Sett inn i et dynamisk array på indeks

Input: Et dynamisk array D, en indeks i og et element e
Output: Et dynamisk array med e lagt til på indeks i

```
1 Procedure Add(D, i, e)
2     EnsureCapacity(D)
3     for j ← D.n − 1 down to i do
4         D.A[j + 1] ← D.A[j]
5     D.A[i] ← e
6     D.n ← D.n + 1
```

- Algoritmen er i $\mathcal{O}(D.n)$
  - Når $i = 0$ gjøres det $D.n$ iterasjoner
  - I tillegg kan `EnsureCapacity` føre til $D.n$ ytterligere iterasjoner
  - To lineærtidsoperasjoner etter hverandre gir fremdeles lineær kjøretidskompleksitet

Legg inn først (illustrasjon)



Fjern på indeks

- Når vi fjerner et element på indeks i må vi flytte alle elementer etter i en posisjon tilbake
- Vi må også huske å sette siste posisjon til **`null`**

ALGORITHM: FJERN FRA ET DYNAMISK ARRAY PÅ INDEKS

`Input:` Et dynamisk array D og en indeks i
`Output:` Et dynamisk array med elementet på indeks i fjernet

```
1 Procedure Remove(D, i)
2     for j ← i to D.n − 2 do
3         D.A[j] ← D.A[j + 1]
4     D.A[D.n − 1] ← null
5     D.n ← D.n − 1
```

- Algoritmen er i $\mathcal{O}(D.n)$
  - Siden $D.n - 1$ elementer flyttes innover når $i = 0$

Dynamiske arrayer *er* lister

Listeoperasjon	Implementasjon	$\mathcal{O}$
`Get`(D, i)	$D.A[i]$	$\mathcal{O}(1)$
`Set`(D, i, e)	$D.A[i] \leftarrow e$	$\mathcal{O}(1)$
`Add`(D, i, e)	`Add`(D, i, e)	$\mathcal{O}(D.n)$
`Remove`(D, i)	`Remove`(D, i)	$\mathcal{O}(D.n)$

- Her er oppslag konstant tid
- Endringer nær begynnelsen av arrayet forårsaker forflytning av alle elementene som gir lineær tid

Litt mer om kjøretidskompleksiteten til `PushBack`

- `PushBack` er $\mathcal{O}(D.n)$ i verste tilfelle, altså når kapasiteten må dobles
- Anta at vi setter inn n elementer i et *tomt* array med n kall på `PushBack`
  - La oss si størrelsen dobles når arrayet inneholder $1, 2, 4, 8, \ldots$ elementer
  - Til sammen kopieres $1 + 2 + 4 + \cdots + n < 2 \cdot n$ elementer
- Altså koster *alle* de n innsettingene $\mathcal{O}(n)$ *til sammen*
  - Det vil si $\mathcal{O}(1)$ per innsetting *i snitt*
- Når vi ser en sekvens av operasjoner under ett, kaller vi det en *amortisert* kjøretidsanalyse
  - Å gjøre slike amortiserte kjøretidsanalyser er ikke pensum i IN2010
  - Men vi skal vite hva det betyr
  - Vi kommer tilbake til det senere i semesteret når vi snakker om *hashing*