

Uke 3, Forelesning 1



HUSK – Hittil...

- Essensen i faget: Effektive algoritmer
 - Matematiske forutsetninger
 - Logaritmer, regneregler for logaritmer
 - Eksponenter, regneregler for eksponenter
 - Rekker og summer
 - Bevis
 - Begreper/teknikker: Pseudo-kode; Måle reel tid eller beregne/estimere (teoretisk) tid
 - Introduksjon til metodekall og rekursjon
 - Introduksjon til kombinatoriske søk, kombinasjoner og permutasjoner
 - Introduksjon til dronning oppgaven
 - Hvordan en kan GENERERE PERMUTASJONER
 - Introduksjon til ANALYSE av ALGORITMER
 - Avrunding
 - Gjennomgang av matematisk grunnlaget
- Uke 1.1
 Uke 1.2
 Uke 2.1
 Uke 2.2



OVERSIKT – Uke 3, Forelesning 1 (W3.L1)

- Denne uken: **LISTER**, **STABLER** og **KØER**.
- I DAG (W3.L1):
 - TEMA #1: Introduksjon til abstrakte datatyper (ADT'er)
 - TEMA #2: **Lister**
- NESTE FORELESNING (W3.L2):
 - TEMA #3: **Stabler** (Engelsk: Stacks, Norsgelsk: Stakker)
 - TEMA #4: **Køer**



TEMA #1



Introduksjon til Abstrakte Datatyper



- **ADT** = Abstrakt datatype
(Engelsk: Abstract Data Type)
- ADT definerer **hva** en ny datatype er og skal gjøre, **ikke hvordan** datatypen skal implementeres.
 - Det kan være mange mulige implementasjoner (både matematiske og programmatisk, dvs. måter å skrive kode) som gir oss en og samme datatype
 - Hva som er beste implementasjon må avgjøres etter hvilke bruk vi har
- ADT er en abstraksjon av en (mye brukt) datatype
- ADT er en modell av en (mye brukt) struktur og mekanismer/atferd tilknyttet strukturen



- ADT er en **modell av** en (mye brukt) **struktur** og **mekanismer/atferd**
- Hver ADT har:
 - En tilknyttet struktur
 - En semantikk (betydning)
 - Operasjoner (grensesnitt)
- For hver ADT skal vi se på ADT'ens struktur, semantikk og operasjoner.
- I tillegg skal vi se på
 - Tidsforbruket til operasjonene
 - Eksempler på bruk





Lister



LISTER – Begrepsapparatet #1

- LISTE-ADT'en modellerer det de fleste forbinder med begrepet liste, nemlig en liste av elementer:

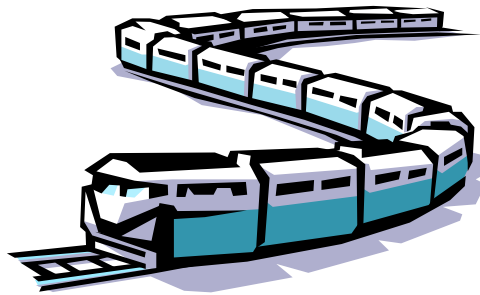
A_1, A_2, \dots, A_n .

- Begrepet er del av det mere generelle begrepet "samling" (Engelsk: Collections).
- De 5 vanlige samlinger er:
 - Sets (mengder),
 - Bags (sekker ?),
 - Lists (lister),
 - Arrays (tabeller ?),
 - Dictionaries (ordbøker)
- Forskjellene er om de er ordnede eller ikke, og om gjentakelse er tillatt.



LISTER – Begrepsapparatet #2

- I forhold til andre samlingsbegreper...
En liste impliserer ORDEN (rekkefølge):
 - At noe kommer før eller etter noe,
 - At det har et hode og en hale (en først og sist)...
- Det er som et tog med liste-elementer som vogner!



LISTER – Begrepsapparatet #3

- Fordi liste-ADT'en er "abstrakt":
 - Elementene i listen kan være hva som helst – som personer, biler og varer.
 - I vår liste-ADT legger vi selvfølgelig inn *modeller* av elementene. Dette kan være vilkårlig komplekse objekter eller ganske enkelt navn, bilnummer og varebetegnelser.
 - Vi kan lage mange forskjellige ADT'er for lister avhengig av hvilket grensesnitt vi definerer for den.
 - Grensesnittet (dvs. ADT'en) definerer hvilke operasjoner vi kan utføre på lister, men sier ingenting om hvordan disse operasjonene blir utført.

LISTER – Mulige operasjoner (grensesnitt)

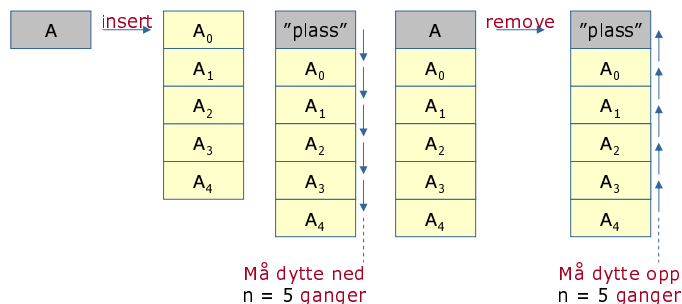
```
public interface ListeInterface
{ // Sette inn et element på en gitt plass
  void insert(Object x, int posisjon);
  // Slette et gitt element fra listen
  void remove(Object x);
  // Sjekke om et element finnes i listen
  boolean find(Object x);
  // Få tak i elementet på en angitt plass
  Object findKth(int k);
  // Skrive ut alle elementene i listen
  void printList();
  // Sjekke om listen er tom
  boolean isEmpty();
  // Tømme (nullstille) listen
  void makeEmpty();
}
```



LISTER – Array implementasjon av lister #1

- **insert: $O(n)$.**

I verste fall: Innsetning først i listen fører til at vi må flytte alle de $n-1$ andre elementene ett hakk "nedover" i array'en.



- **remove: $O(n)$.**

Som for insert men må flytte alle de $n-1$ andre elementene ett hakk "oppover" i array'en.



LISTER – Array implementasjon av lister #2

I tillegg til **insert** og **remove** som "i verste fall var $O(n)$...

- **find**: $O(n)$ — så bra som man kan forvente.
- **findKth**: $O(1)$ — positivt!
- **printList**: $O(n)$.
- **isEmpty**: $O(1)$.
- **makeEmpty**: $O(1)$.

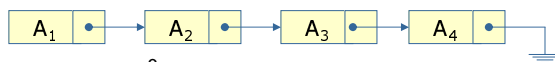
- **Problem:**

Array'er kan ikke vokse dynamisk etter behov — størrelsen må bestemmes senest i det øyeblikket array'en opprettes.

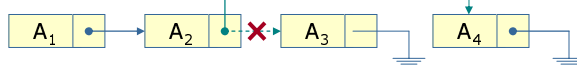


LISTER – Pegerkjede implementasjon av lister #1

- Alternativet til array-implementasjon er en kjede av elementer der et element peker til neste element



- Eksempel på **remove**: Fjern element A3

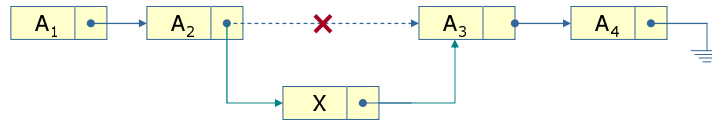


- De logiske trinnene:
 - Husk hvor A_3 er: $X \leftarrow A_2.neste$.
NB! Ikke alltid nødvendig. Kun når søppel samles manuelt, eller om A_3 skal brukes til noe annet osv...
 - La A_2 nå peke til A_4 : $A_2.neste \leftarrow A_4$.
NB! Dette trinnet er den egentlige "remove" operasjonen
 - La A_3 nå peke til "intet": $A_3.neste \leftarrow null$.
NB! Ikke alltid nødvendig.



LISTER – Pegerkjede implementasjon av lister #2

- Eksempel på **insert**: Sett inn element X

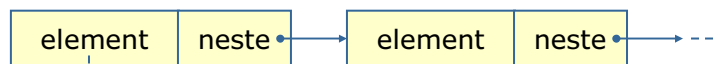


- De logiske trinnene:
 - Husk hvor A₃ er (eller hva A₂ peker til): $Y \leftarrow A_2.neste$.
NB! Ikke nødvendig hvis A₃ er tilgjengelig uansett.
 - La A₂ nå peke til X: $A_2.neste \leftarrow X$.
og
La X peke til A₃: $X.neste \leftarrow A_3$ (eller $X.neste \leftarrow Y$).
NB! Dette trinnet er den egentlige "insert" operasjonen NB!
Ikke alltid nødvendig.



LISTER – Pegerkjede implementasjon av lister #3

- Datastrukturen, som i diagrammet, er en "Node" med en referanse (peker) til en annen – i vårt tilfelle "neste" – node.



objektet
(selve
elementet)

```
class Node
{ Object element; // Selve elementet
  Node neste;    // Ref til "neste" element

  public Node(Object x, Node n)
  { element = x;
    neste = n;
  }
}
```



LISTER – Tider for pekerkjede implementasjon av lister

- **insert: $O(1)$** dersom vi allerede har funnet stedet der elementet skal settes inn.
- **remove: $O(1)$** (tilsvarende som for insert).
- **find: $O(n)$.**
- **findKth: $O(n)$** — må gå gjennom k-1 elementer for å komme til element k.
- **printList: $O(n)$.**
- **isEmpty: $O(1)$.**
- **makeEmpty: $O(1)$.**

- Dersom vi inkluderer tiden vi bruker på å finne riktig sted, blir innsetting og sletting $O(n)$, men med svært liten konstant.

- **MERK:** Innsetting og sletting blir veldig raske i gjennomsnitt dersom vi som oftest setter inn først/tidlig i listen.



LISTER – Implementasjon av pekerkjedelister #1

```
public class PekerListe
{ // NB! Vi bruker Node slik klassen er definert tidligere.
  Node liste = null;

  // Skriver ut hele listen (alle elementene i listen).
  public void printList()
  { Node n = liste;
    while (n != null)
    { n.element.print();
      n = n.neste;
    }
  }

  // Metoder på de neste foliene skal settes inn her!

}
```



LISTER – Implementasjon av pekerkjedelister #2

```
public class PekerListe
{ // Flere metoder. Fortsetter fra tidligere foil...

    //Tømmer listen
    public void makeEmpty()
    { liste = null;
    }

    //Tester om listen er tom
    public boolean isEmpty()
    { return (liste == null);
    }

    //Metoder på de neste foliene skal settes inn her!

}
```



LISTER – Implementasjon av pekerkjedelister #3

```
public class PekerListe
{ // Flere metoder. Fortsetter fra tidligere foil...

    // insert...
    // Antar at posisjon ikke er større enn listelengde pluss en.
    public void insert(Object x, int posisjon)
    { if (posisjon == 1)
      { liste = new Node(x, liste);
      }
      else
      { Node n = liste;
        for (int i = 2; i < posisjon; i++)
        { n = n.neste;
        }
        n.neste = new Node(x, n.neste);
      }
    }

    // Metoder på de neste foliene skal settes inn her!

}
```



LISTER – Implementasjon av pekerkjedelister #4

```
public class PegerListe
{ // Flere metoder. Fortsetter fra tidligere foil...

    // remove...
    public void remove(Object x)
    { Node n = liste;
      if (n != null)
        { while (n.neste != null && !n.neste.element.equals(x))
          { n = n.neste;
            }
          if (n.neste != null)
            { n.neste = n.neste.neste;
              }
          }
      }

    // Metoder på de neste foliene skal settes inn her!

}
```



LISTER – Implementasjon av pekerkjedelister #5

```
public class PegerListe
{ // Flere metoder. Fortsetter fra tidligere foil...

    // find...
    public boolean find(Object x)
    { Node n = liste;
      while (n != null && !n.element.equals(x))
        { n = n.neste;
          }
      return (n != null);
    }

    // Metoder på de neste foliene skal settes inn her!

}
```



LISTER – Implementasjon av pekerkjedelister #6

```
public class PekerListe
{ // Flere metoder. Fortsetter fra tidligere foil...

    // find kth...
    // Antar at listen består av minst k elementer.
    public Object findKth(int k)
    { Node n = liste;
      for (int i = 2; i <= k; i++)
        { n = n.neste;
        }
      return n;
    }

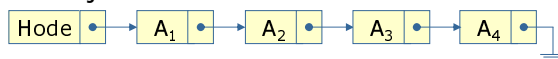
    // Klassen avslutter her!
}
```



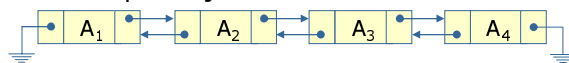
LISTER – Alternative implementasjoner av pekerkjedelister

Alternative liste-implementasjoner:

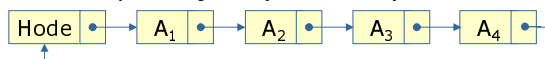
- Pegerkjede med hode:



- Toveis pekerkjede:



- Sirkulær pekerkjede (med hode):



- Toveis pekerkjede med hode og hale, toveis sirkulær pekerkjede...

Variant: sortert liste

- Find blir fortsatt $O(n)$ hvis vi bruker pekerkjede, men $O(\log n)$ hvis vi bruker array og binærøk.



Neste gang skal vi...

- Fortsette med lister (og avslutte lister):
 1. Sorterte lister; Radix-sort og bøtter...
 2. Implementasjonseksempel av pekerkjedelister med hode + siste-peker.
 3. Implementasjonseksempel av radix-sort.
- Begynne med stabler og køer.

