Oppsummering

I. HVA VAR DET?

I.1 ADTer og Programutvikling

I.2 Datastrukturer

I.3 Algoritmer

II. PENSUM

III. EKSAMEN

IV. ØNSKER

Hva oppnår vi med ADT-modul begrepet ?

1. *utvider programmeringsspråket* med nye primitive typer
   - typer med tilhørende operasjoner
2. ulike implementasjoner av en ADT *kan erstatte hverandre*
   - ulike implementasjoner har ulike plass- og tidsforbruk (og kan velges avhengig av behov)
   - enkelte moduler kan vedlikeholdes uten at det krever endringer i resten av systemet (ingen ekstra integrering av ny implementasjon)
3. implementasjon = *konfigurasjon* + koding
   - konfigurasjon blir en egen gren av programmering på lik linje med vanlig (tradisjonell) implementering

En ADT-modul oppfyller krav til en god modularisering (Parnas)

1. den utgjør en *logisk* enhet
   - hver ADT definerer et begrep
2. den har et *klart grensesnitt* mot omverden (*enkapsling*)
   - gitt ved grensesnittsmetoder beskrevet i dokumentasjon
3. den er *gjenbrukbar*
   - i alle kontekster der det er behov for begrepet som ADT definerer

Implementasjon av en ADT-modul

1. Ta utgangspunkt i spesifikasjon (interface & dokumentasjon)
2. Se om det finnes noen moduler – i standard eller i din lokale bibliotek – som kan brukes
3. Implementer typen
   - finn datastruktur som kan lagre nok og relevant informasjon
   - beskriv **datainvarianten**
   - beskriv **abstraksjonsfunksjonen**
     - hvordan konkrete instanser av datastrukturen som oppfyller datainvarianten representerer abstrakte verdier
   - analyser **plassbehovet** til datastrukturen
4. Implementer metodene
   - finn passende algoritmer (ofte modifiser en eksisterende en)
   - vis at den er korrekt
     - at algoritmen **gjenetablerer datainvarianten** forutsatt at denne var oppfylt ved start
     - at dersom inndata oppfyller forkravene så vil algoritmen returnere en verdi som tilsvarer den abstrakte verdien som kreves av spesifikasjonen

       (løkke- og rekursjonsinvarianter)
   - analyser **tidsforbruket** til algoritmen
5. Skriv kode

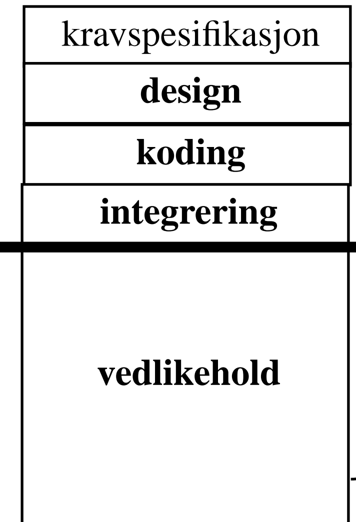
Programutvikling, i likhet med ethvert kreativt arbeid,
foregår ikke som en rettlinjet prosess.

Programutvikling krever – men er ikke det samme som – koding

Gamle Vannfallsmodellen

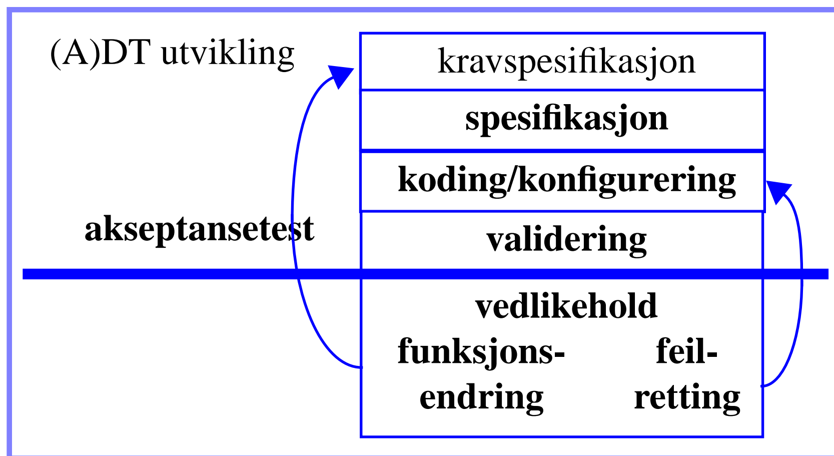
- orientert mot enkeltprosjekt
- gjenbruk er fraværende
- vedlikehold er ikke integrert

akseptansetest

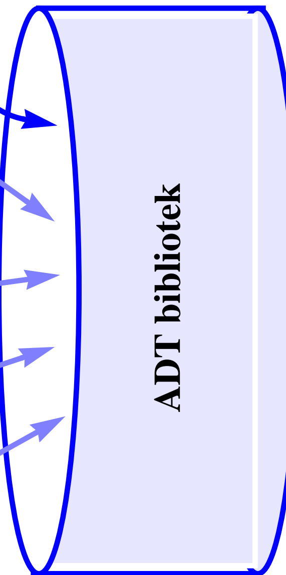


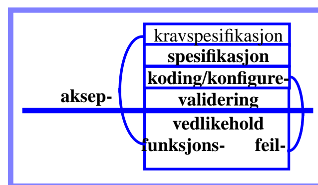
ADT-basert livssyklusmodell

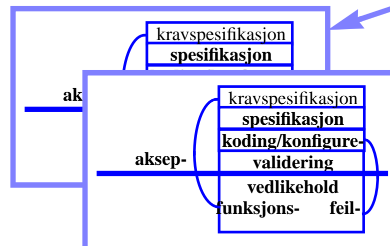
(ADT-programmering)



preventivt vedlikehold
ytelsesbedring

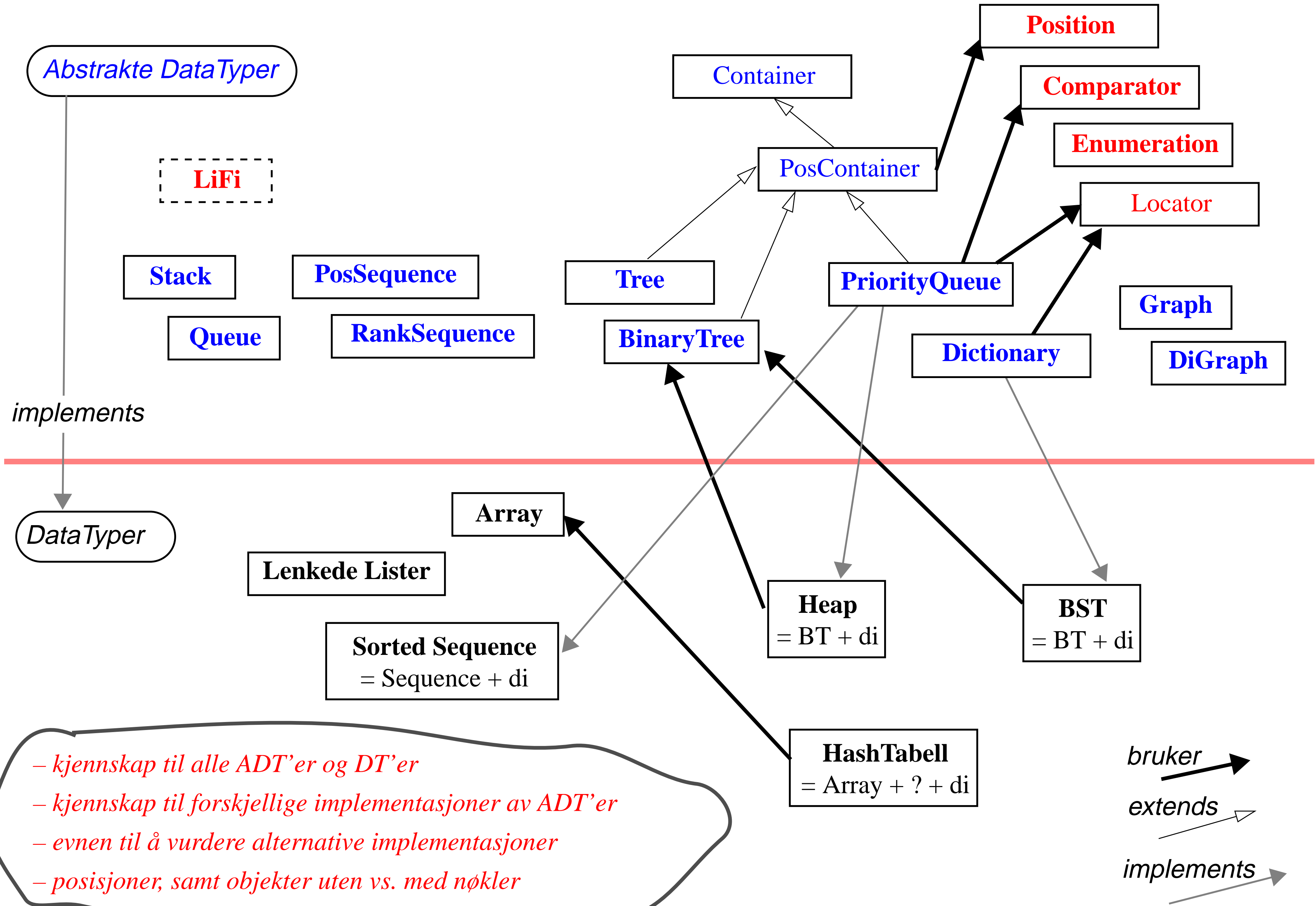






enkeltprosjekt/
underprosjekt

Abstarkte DataTyper og DataTyper



Algoritmer

- enkel **korrekthets-** og **kompleksitetsanalyse** av algoritmer
- design av **rekursive algoritmer**
- **sorteringsalgoritmer** (“in-place” eller ikke?)
  - **prioritetskø-sortering**: instikk- / seleksjon- / haugsort (heapsort)
  - **flettesortering** (mergesort) / **kvikksort** (quicksort)
  - boblesortering (bubblesort)
- **trealgoritmer**
  - traversering: **DFS**, **BFS**
  - DFS: **preorder**, **postorder** (**innorder** for Binære Trær)
- ***heap*** (implementasjon av PriorityQueue)
  - **innsetting** og **fjerning** med opprettholdelse av heap-invarianten
- ***binære søketrær*** (implementasjon av Dictionary)
  - søk i BST: **innsetting** og **fjerning** med opprettholdelse av BST-invarianten (multiple nøkler)
- ***hashtabeller***
  - litt om hashfunksjoner og kollisjonshåndtering
- **grafalgoritmer**
  - graftraversering: **DFS** og **BFS**
  - transitiv tillukking (**n*DFS** / **Floyd-Warshall**)
  - rettede grafer: **topologisk sortering** (samt forskjeller i DFS / BFS fra ikke-rettede grafer)
  - vektede grafer: SS-SP (**Dijkstra** / **Ford-Bellman**)
  - vektede grafer: minimum utspennende tre, MST (**Kruskal**)

Pensum: fra boken (H-99)

	unntatt	kursorisk	poenger
KAP. 1	1.4.2		**OO, ABSTRAKSJON ...**
KAP. 2			**O-NOTASJON**
KAP. 3	3.2.4, 3.5	**3.1.3, 3.2.3, 3.4**	**STABEL, KØ, LISTE, ADAPTER**
KAP. 4			**SEKVENS; RANK, POSITION; ENUMERATION-ITERATOR**
KAP. 5	5.5	**5.4.4**	**TRÆR, B-TRÆR + BSF/DFS (PRE/POST/IN-ORDER)**
KAP. 6		**6.3.4**	**PRIORITETSKØ, HEAP; TOTALORDNING/COMPARATOR**
KAP. 7	7.4, 7.5	**7.6.2, 7.6.3, 7.7**	**ORDBOK, BST, HASHTAB**
KAP. 8	8.1.3, 8.2, 8.6	**8.4, 8.5**	**QUICKSORT (MERGESORT); RANDOMISERING**
KAP. 9		**9.4.5**	**GRAF, DIGRAF, DFS/BFS, FW**
KAP. 10	10.2.2-10.2.4, 10.3		**SS-SP** (DIJKSTRA,BELLMAN-FORD)**, MST** (KRUSKAL)

Eksamen

– skal sjekke at
1. *en kan bruke kjente datastrukturer og algoritmer*
2. *evt. gjennom tilpassing, modifikasjon og abstraksjon*
3. *til å lage* ***selvstendige og effektive løsninger på nye problemer***

– kjennskap til definisjoner, invarianter, algoritmer er en opplagt forutsetning

1. Algoritmer, effektivitet (O-notasjon) og rekursjon (oblig. 1 og 2)
2. DataStrukturer (oblig. 2)
3. Implementasjon og ADT (modulær) programmering (oblig. 2)

– typiske eksempler:

- ***gitt et problem****:* deklarer **datastruktur** og design en **algoritme** (med evt. krav til datastrukt/kompleksitet)
- ***gitt flere problemer av liknende karakter****:* lag en **generisk algoritme** som løser hvert problem når passelig instansiert (design et grensesnitt som abstraherer forskjeller)
- ***gitt en ADT****:* design/beskriv en **implementasjon som tilfredstiller visse (kompleksistets)krav**

– bruk kjente interface (disse er ofte gitt i vedlegg, men man må kjenne til deres intenderte virkemåte)

NB! Ved bruk av andre interface i en implementasjon, vil kompleksiteten avhenge av implementasjon av disse interface'ne
– man snakke om kompleksitet ***relativt til*** implementasjon av andre interface.

– se tidligere eksamener på kurssiden
(prøv å løse de selv, se på løsningsforslag, sammenlign og se hva som evt. kunne forbedres i begge)

På eksamen (15 desember)

prosentsatsene ved enkle oppgaver angir forventet og omtrentlig tidsforbruk

– disse trenger ikke å stemme for hver person!

- Les ***hele*** eksamensettet
- Løs ***først alle*** de oppgavene du kan
- For de oppgavene du ikke ser en løsning på – disponer tiden!!!
  - begynn med en som virker enklest
  - har du brukt for mye tid på denne uten å lykkes,
  - forsøk heller å løse en annen oppgave istedenfor å bli stående ved den ene (du kan komme tilbake til denne dersom du får tid senere)
- løkke- og rekursjonsinvarianter, forbetingelser, dokumentasjon kan du skrive først når du er ferdig med alle dine løsninger
- klarer man ikke å vurdere kompleksiteten, er det naturlig å lage så effektive algoritmer som mulig
- det er bedre å skrive noe – selv om veldig lite – riktig enn ingenting
- det er bedre å skrive ingenting enn masse tull