Oppsummering

I. HVA VAR DET?

I.1 Programutvikling

I.2 Datastrukturer

I.3 Algoritmer

II. PENSUM

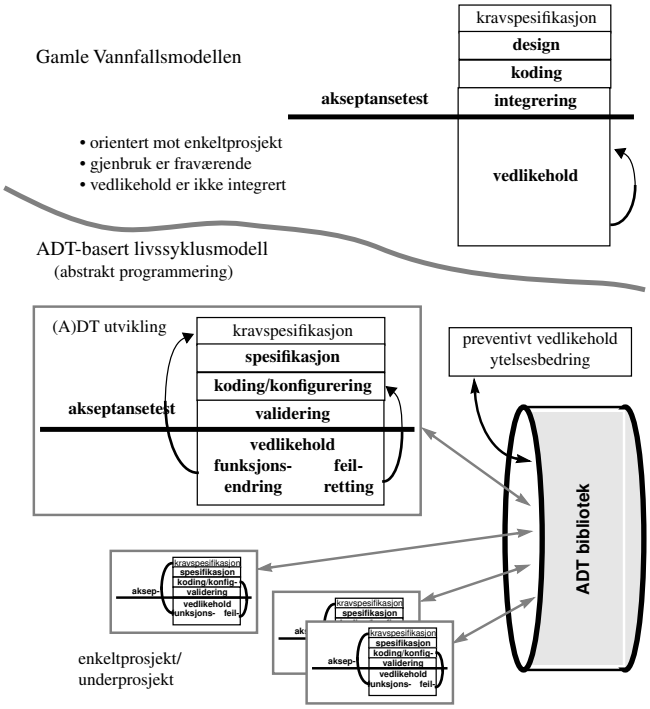
III. EKSAMEN

IV. ØNSKER
ROTASJON AV AVL-TRÆR

Programutvikling

krever – men er ikke det samme som –

Koding



Hva oppnår vi med

ADT-modulbegrepet?

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 som gir et program ulike egenskaper (kan velges avhengig av behov)
   - enkelte moduler kan vedlikeholdes uten at det krever endringer i resten av systemet (ingen ekstra integrering)
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 en eller flere 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 representerere abstrakte verdier
   - analyser plassbehovet til datastrukturen
4. Implementer metodene
   - finn passende algoritme (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

Husk at programutvikling, i likhet med ethvert kreativt arbeid, ikke foregår som en rettlinjet prosess.

DataTyper og Abstrakte DataTyper

Algoritmer

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

Pensum

kap	unntatt	kursorisk	
1.	1.4.2		OO, ABSTRAKSJON: INTERFACE, ARV
2.			O-NOTASJON: VERSTE- VS. GJENNOMSNITTSTILFELLE
3.	3.2.4, 3.5	3.1.3, 3.2.3, 3.4	STABEL, KØ, LISTE (TILPASSING)
4.			SEQUENCE; RANK-, POS-; ENUMERATION
5.	5.5	5.4.4	TRÆR, BINÆRETRÆR: BFS/DFS (PRE-/POST-/INORDER)
6.	6.4	6.3.4	PRIORITETSKØ, HEAP: TOTALORDNING/COMPARATOR
7.	7.5, 7.7	7.6.2, 7.6.3	ORDBOK, BINÆRESØKETRÆR, AVL-TRÆR, HASHTAB
8.	8.1.3, 8.2, 8.5, 8.6	8.4	QUICKSORT (MERGESORT, INSERTION-/SELECTIONSORT)
9.		9.4.5	GRAPH, DIGRAPH, DAG: DFS/BFS, TC (FLOYDWARSHALL), TS
10.	10.1.5, 10.3 10.2.2-10.2.4		VEKTEDEGRAFER: SS-SP (DIJKSTRA), MST (KRUSKAL)

Eksamen

skal sjekke at

1. en kan bruke kjente datastrukturer og algoritmer
2. evt. gjennom tilpassing 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 og 3)
3. Implementasjon og abstrakt (modulær) programmering (oblig. 3)

typiske eksempler:

- *gitt et problem:* deklarer **datastruktur** og design en **algoritme med en gitt kompleksitet** som løser problemet
- *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 gitte moduler og 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 – ellers kan man snakke generelt om kompleksitet ***relativt til*** implementasjon av andre interface.

Se prøveeksamen på hjemmesiden (prøv å løse den selv, se på løsningsforslag, sammenlign de to, se hva som evt. kunne forbedres i begge)

På eksamen

prosentsatsene ved enkle oppgaver angir forventet og omtrentlig tidsforbruk – disse trenger ikke å stemme for hver person!

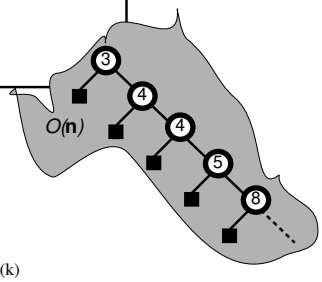
- Les ***hele*** eksamensettet
- Løs ***først alle*** de oppgavene du kan
- For de oppgavene du ikke ser en løsning på – disponer tiden!!!
  - begynn med en som virker enklest
  - har du brukt for mye tid på denne uten å lykkes,
  - forsøk heller å løse en annen oppgave
    istedenfor å bli stående ved den ene
    (du kan returnere til denne dersom du får tid senere)
- løkke- og rekursjonsinvarianter, forbetingelser, dokumentasjon kan du skrive ***først når du er ferdig*** med alle dine løsninger

- får du problemer med å skrive kode, skriv i det minste pseudo-kode som viser at du vet hvordan problemet skal løses.
- 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

Balansering av BST

find(k), insert(k,e), rem(k) er alle O(h(BST)) som bør være O(log n)

```
settInn(Position v, Object k, Object e) {
    Position p= findP(k,v);
    if (isExternal(p))
        – en ny intern node
        – sett inn (k,e) der
    else
        – settInn(rightChild(p),k,e) }
```



insert(3,A);
insert(4,B);
insert(4,B);
insert(5,C);
insert(8,D); h(BST) = O(n)

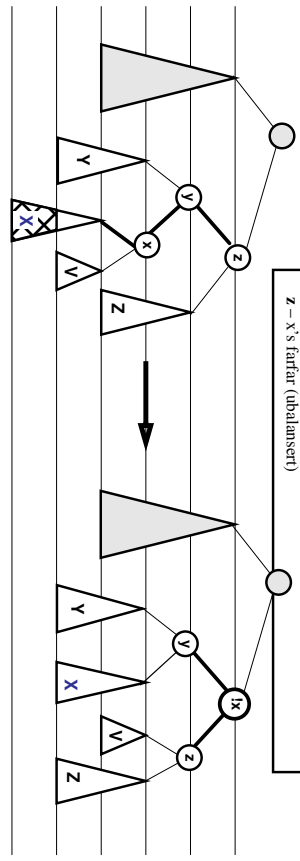
tilsvarende uhell kan skje ved en serie rem(k)

AVL Trær

er et Binært Tre T (for lagring av nøkler, eller objekter med nøkler) som tilfredstiller :

1. **BST INVARIANT** (“relasjonell”) – for hver intern node p :
   for hver node v i p’s venstre subtre : key(v) £ key(p)
   for hver node h i p’s høyre subtre : key(h) ≥ key(p)
2. **AVL INVARIANT** (“strukturell”) – hver intern node p er **balansert** :
   | height(leftChild(p)) - height(rightChild(p)) | ≤ 1

7.2. Et AVL Tre T som lagrer n nøkler har høyden h(T) = O(log n)

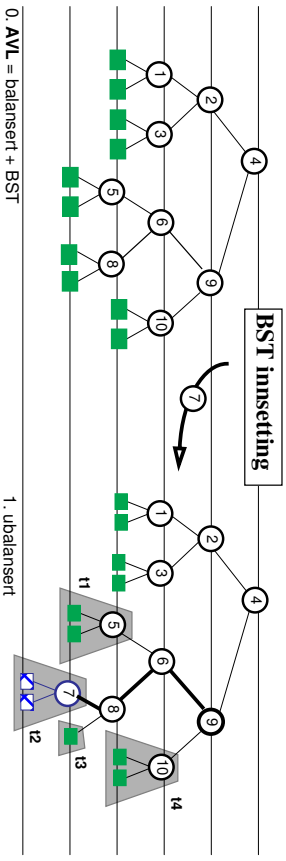


2. Rotasjon(x,y,z)

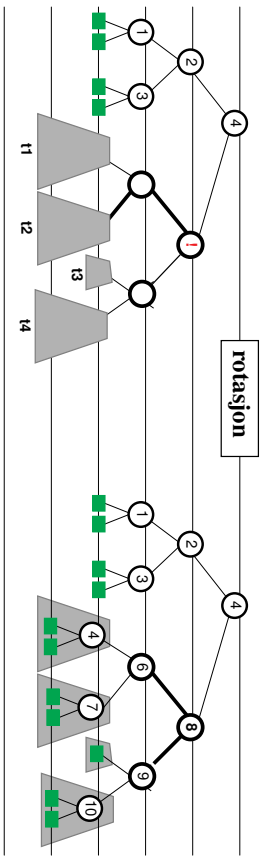
p – ny insatt node (1)
ubalanse kan oppstå kun på stien **S** fra **p** til roten
x – første noden på **S** (over el. lik p) hvis farfar er ubalanser
y – x’s far
z – x’s farfar (ubalansert)

inorder (DFS) liste av noder: a b c (2)
y x z
inorder (DFS) liste av subtrær rotet under x, y, z: T1 T2 T3 T4
Y X V Z
BST invariant : T1 a T2 b T3 c T4
her Y y X x V z Z

! = et nytt subtre med b.elem() i roten (3)
!.leftChild = a ; !.rightChild = c ;
// a b c
a.leftChild = T1; a.rightChild = T2 ;
// T1 a T2 b c
c.leftChild = T3 ; c.rightChild = T4 ;
// T1 a T2 b T3 c T4



0. **AVL** = balansert + BST
1. ubalansert
BST **t1** 6 **t2** 8 **t3** 9 **t4**



2. balansert tre
subtrærene **t1**, **t2**, **t3**, **t4** er i riktig order
3. **AVL** = balansert +
BST **t1** 6 **t2** 8 **t3** 9 **t4**

Er forutsettningene alltid oppfylt?

Hvis det oppstår ubalanse, vil det alltid finnes

1. **en første** ubalansert node som er **farfar til en intern node**
2. **fire trær** som antatt i algoritmen over

1. a) Hvis en node ikke er farfar til noen intern node, så er den balansert!

b) Siden ubalanse oppstår kun på stien fra den nye noden p mot roten (dvs. på en total ordning av noder), vil det finnes den første ubalanserte farfar noden

2. Følger av 1.
Har vi en ubalansert farfar z, så velger vi bare barnet og barnebarnet til z på stien mot p. (Et subtre av z, et til av z's barnet y, og to av z's barnebarnet x.)
Det er kun 4 muligheter:

rem(k) fra et AVL Tre

1. BST . rem(k) ; if (AVL invariant er ok) – ferdig

2. else:

p – faren til fjernet node
ubalanse oppstår på stien **S** fra **p** til roten
z – første noden på **S** (over **p**) som er **ubalanser**
y – z's **høyste barn** (ligger ikke på **S**)
x – y's **høyste barn** (ikke alltid entydig)

rotasjon(x,y,z)

fortsett mot roten finner du en ubalansert z gjenta 2.

= O(height(AVL)) = O(log n)