Velkommen til IN2010 gruppe 6

Praktisk info

- Denne uken er repitisjon
  - Hva skal vi gjøre i dag?
  - Lab eller eksamens oppgaver

- Veldig lurt å komme i gang med innlevering 4
  - Vanskeligste er å bygge grafen, men dere får det til lett

Mentimeter

Lab
mafredri@ifi.uio.no

Mentimeter

Kødda!
Nå over til eksamens eksempler

Starter med sortering

- Generelt så handler sortering om å få de minste elementene først og de største sist... altsa i en sortert rekkefølge
- Veldig kult, tillater Binary Search og andre ting sikkert

- Var mange forskjellige måter å gjøre det på
  - Bubblesort
  - Selectionsort
  - Insertionsort
  - Mergesort
  - Heapsort
  - Quicksort
  - Bucketsort
  - Radixsort

Finn duplikat

12 poeng

Du er gitt et array A med sammenlignbare elementer. Du får vite at A inneholder nøyaktig ett duplikat. Altså er alle elementer i A unike, bortsett fra *ett* element, som forekommer nøyaktig to ganger.

(a) Anta at du er gitt et *sortert* array A, og får vite at x er duplikatet i A. Skriv en effektiv prosedyre som skriver ut de to posisjonene som inneholder x. Oppgi kjøretidskompleksiteten på algoritmen.

Input: Et *sortert* array A med n sammenlignbare elementer, og et element x som forekommer nøyaktig to ganger
Output: Skriver ut de to posisjonene $0 \leq i < j < n$ hvor `A[i]` $=$ `A[j]` $= x$ forekommer
Procedure `FindSortedIndicesOfDuplicate(A, x)`

```
// ···
```

(b) Anta at du er gitt et *sortert* array A (og nå får du ikke oppgitt elementet som er duplisert). Skriv en effektiv prosedyre som skriver ut de to posisjonene til elementet som forekommer to ganger. Oppgi kjøretids-kompleksiteten på algoritmen.

Input: Et *sortert* array A med n sammenlignbare elementer
Output: Skriver ut de to posisjonene $0 \leq i < j < n$ hvor `A[i]` $=$ `A[j]`
Procedure `FindSortedDuplicateIndices(A)`

```
// ···
```

(c) Anta at du er gitt et array A (nå kan du ikke anta at arrayet er sortert). Skriv en effektiv prosedyre som skriver ut de to posisjonene til elementet som er duplisert. Oppgi kjøretidskompleksiteten på algoritmen.

Input: Et array A med sammenlignbare n elementer
Output: Skriver ut de to posisjonene $0 \leq i < j < n$ hvor `A[i]` $=$ `A[j]`
Procedure `FindDuplicateIndices(A)`

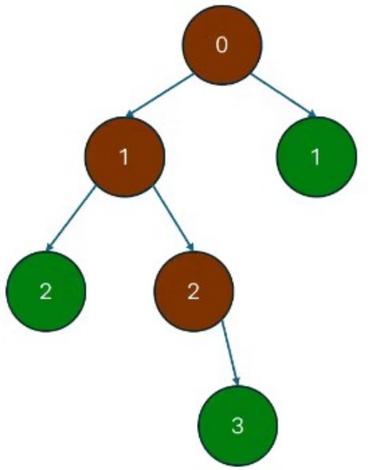
```
// ···
```

Over til trær

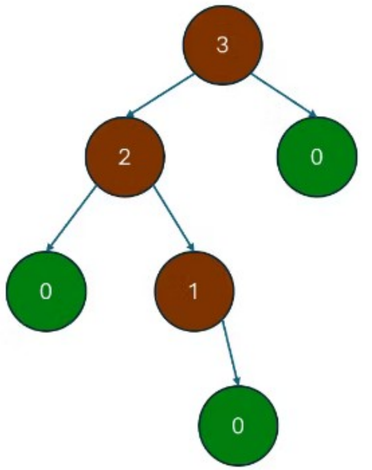
- Generelt er det noder som har barnenoder som igjen har barnenoder... etc.
- Det tomme treet er bare null, høyde/dybde = -1

- Vi har vanlig trær
- Vi har søketrær
- Vi har binære søketrær
- Vi har AVLtrær

Dybde



Høyde



Er binærtreet et søketre? 10 poeng

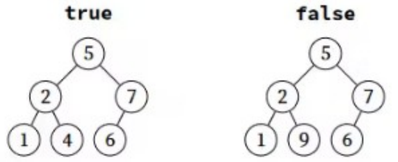
Anta at du er gitt et binærtre B med unike heltall. Hvis v er en node i det binære treet, så gir

- v.`element` heltallet som er lagret i noden
- v.`left` venstre barn av v
- v.`right` høyre barn av v

Vi ønsker å sjekke om det binære treet også er et binært *søketre*. Under finner du spesifikasjonen for algoritmen, og to eksempler på trær som henholdsvis bør gi **`true`** og **`false`**.

Input: Rotnoden v av et binærtre B
Output: Returnerer **`true`** hvis binærtreet er et binært søketre, **`false`** ellers

```
1 Procedure CheckBST(v)
  | // ···
```



Det finnes flere gode løsninger på dette problemet. Følgende kan være behjelpelig når du tenker på en løsning:

- Du kan anta at du har prosedyrer `FindMin` og `FindMax` som henholdsvis finner minste og største tall i det binære treet. Siden treet ikke er garantert å være et binært søketre (eller balansert) så vil disse prosedyrene ha lineær tid.
- Det kan være lurt å dele algoritmen opp ved å lage en hjelpeprosedyre.

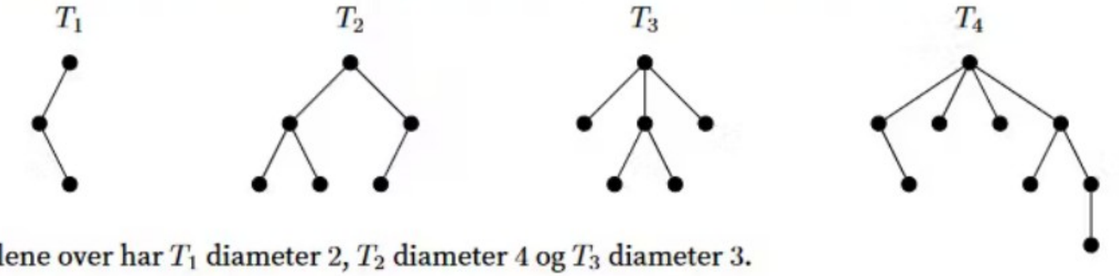
(a) Skriv ned egenskapen et binærtre må ha for å kunne kalles et binært *søketre*.

(b) Fullfør prosedyren over. Lavere kjøretidskompleksitet er mer poenggivende.

(c) Oppgi kjøretidskompleksiteten på algoritmen din med hensyn til antall noder i binærtreet.

Diameteren til et tre — 12 poeng

Vi definerer *diameteren* til et tre som lengden til den lengste stien mellom to noder.



I eksemplene over har T_1 diameter 2, T_2 diameter 4 og T_3 diameter 3.

(a) Hva er diameteren til T_4?

I de neste deloppgavene begrenser vi oss til *binære* trær. Hvis v er en node, så er:

- v.`left` venstre barn av v
- v.`right` høyre barn av v

(b) Vil den lengste stien i et binært tre alltid gå gjennom rotnoden? Begrunn svaret.

(c) Skriv en prosedyre som finner diameteren til et gitt *binærtre*.

```
Input: Rotnoden v av et binærtre
Output: Returnerer diameteren til treet
1 Procedure Diameter(v)
  | // ···
```

Lavere kjøretidskompleksitet er mer poenggivende.

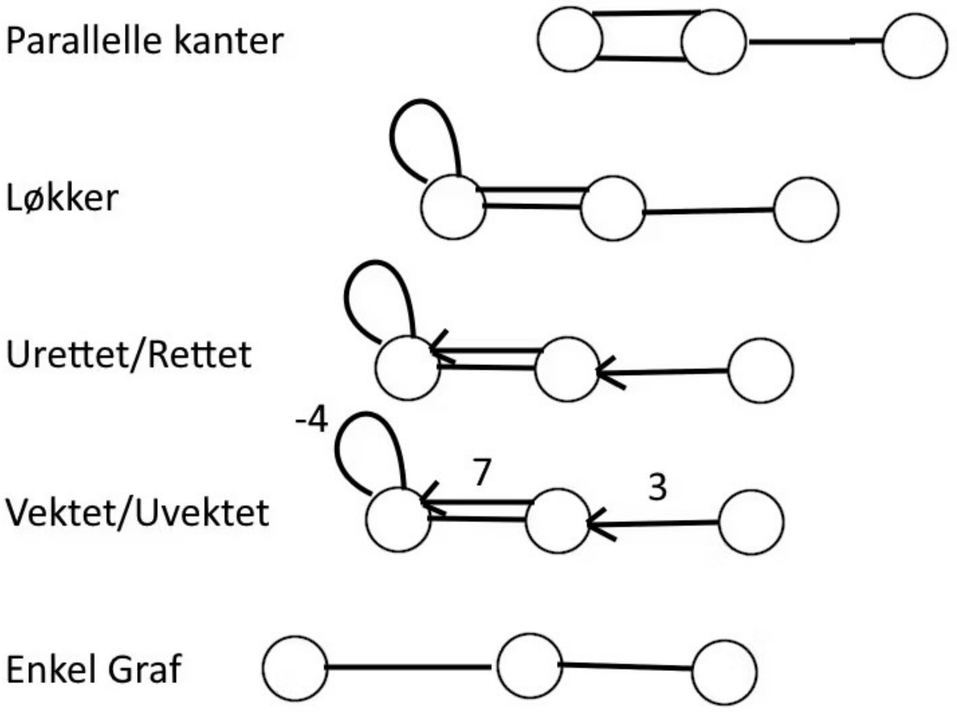
(d) Oppgi kjøretidskompleksiteten på algoritmen din for et binærtre med n noder.

Over til grafer

- Vi har graf terminologi til høyre

- Vi har BFS
- Vi har DFS
- Vi har TOPsort
  - Får fra noder uten inngang og gjør et "vanlig" søk

- Vi har Dijkstras
  - Søk med prioritetskø (akkumulert dist)
- Vi har Bellmanford
  - Bruteforce, prøver alt flere ganger
- Vi har Prim
  - Søk med prioritetskø (ren vekt)



Kjøretid på grafalgoritmer

8 poeng

For hver grafalgoritme, kryss av på den (laveste) korrekte kjøretidskompleksiteten.

	$O(1)$	$O(\|V\| + \|E\|)$	$O((\|V\| + \|E\|) \cdot \log(\|V\|))$	$O(\|V\| \cdot \|E\|)$
`DFSFull`				
`Prim`				
`TopSort`				
`BellmanFord`				

Korte beskrivelser av algoritmene:

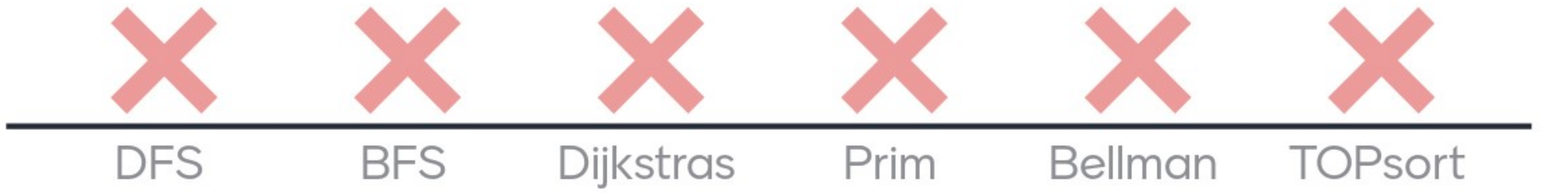
- `DFSFull`: Besøker alle noder i en graf nøyaktig én gang (dybde-først)
- `Prim`: Finner et minimalt spenntre av en gitt graf
- `TopSort`: Gir en topologisk ordning av nodene i en gitt graf
- `BellmanFord`: Finner korteste stier fra én til alle andre noder

Mentimeter

Bli med i mentien plis

Hvilke graf algoritmer har O(1) kjøretid?

None of the options are correct!

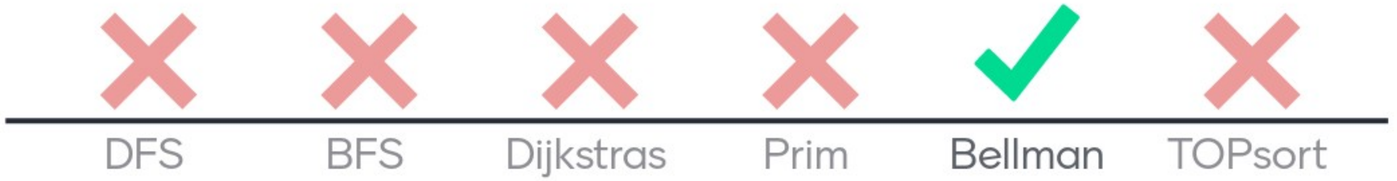


Mentimeter

Hvilke graf algoritmer har O(|V| + |E|) kjøretid?

Hvilke graf algoritmer har O(|V| + |E| * log(|V|)) kjøretid?

Hvilke graf algoritmer har O(|V| + |E| * log(|V|)) kjøretid?



Garbage collection — 8 poeng

Mange moderne programmeringsspråk har en *garbage collector*, som er en prosedyre som frigjør minne som garantert ikke vil brukes i programmet lenger. Du skal utlede en enkel algoritme for garbage collection.

Vi kan anta at alt som lagres er *objekter*, der et objekt kan referere til andre objekter. Vi lar en $G = (V, E)$ være en objektgraf, der V representerer alle objektene som er opprettet, og en (rettet) kant fra u til v betyr at objektet u har en referanse til objektet v. I tillegg har vi en mengde R med alle objektene som kan refereres til direkte (typisk objekter som refereres til av programvariabler). Alle objekter i R er også i V. Objekter det *ikke* finnes en referanse til via objekter i R er garantert å ikke bli brukt i programmet, og skal derfor frigjøres.

Det betyr at ingen av objektene i R skal frigjøres, og heller ingen objekter som kan nås gjennom referanser fra objekter i R skal frigjøres. Objektene som skal frigjøres er ikke i R, og kan heller ikke nås fra noe objekt i R.

Anta at du har en prosedyre `Free` som frigjør et objekt. Du skal gi en prosedyre `GarbageCollect` som tar en graf $G = (V, E)$ og en mengde R som input, og kaller `Free` på alle objekter det *ikke* kan nås fra R.

Input: En objektgraf $G = (V, E)$ og en mengde R med objekter
Output: Frigjør alle objekter det ikke finnes en referanse til

```
1 Procedure GarbageCollect(G, R)
  | // ···
```

Kort oppsummert ->
- Har en mengde V og E og R
  - V er alle noder
  - E er alle kanter
  - R er alle noder vi har tilgang på

That's it folks!
Hvis dere lurer på noe: mafredri@ifi.uio.no