Binærsøk, O-notasjon, Trær og Binære Søketrær

IN2010 – Algoritmer og Datastrukturer

Lars Tveito og Daniel Lupp

Høsten 2020

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

Oversikt uke 35

Oversikt uke 35

- Vi skal lære en algoritme som kalles *binærsøk*
    - Den avgjør om et element er tilstede i et array eller ikke
- Vi skal lære om O-notasjon
    - En måte å snakke om hvor rask en algoritme er
- Vi skal lære om *trær*
    - En datastruktur som dukker opp overalt!
- Vi skal lære om binære søketrær
    - En datastruktur for raskt oppslag

Binærsøk

Søk – spesifikasjon

Algorithm 1: Søk (spesifikasjon)

Input: Et array A og et element *x*
Output: Hvis *x* er i arrayet A, returner **true** ellers **false**

- Dette er en spesifikasjon for en algoritme som avgjør om et element er tilstede i et array eller ikke
- Vi har to input, et array A og et element *x*
- Output er **true** eller **false**, avhengig av om *x* er med i A eller ikke
- En algoritme som *oppfyller spesifikasjonen* må
    - terminere på et endelig antall steg
    - gi riktig svar uansett hva A og *x* er

Rett-frem søk – implementasjon

Algorithm 2: En enkel søkealgoritme

Input: Et array A og et element x

Output: Hvis x er i arrayet A, returner **true** ellers **false**

```
Procedure Search(A, x)
    for i ← 0 to |A| − 1 do
        if A[i] = x then
            return true
    end
    return false
```

- Denne algoritmen oppfyller spesifikasjonen fra forrige slide
- I *verste tilfelle* må vi løpe gjennom hele arrayet
    - Det vil si vi må gjøre $|A|$ sammenligninger
    - Her angir $|A|$ størrelsen på A
    - Vi bruker 0-indekserte arrayer
    - (Merk at boken bruker 1-indekserte arrayer)

Binærsøk – spesifikasjon

Algorithm 3: Binærsøk (spesifikasjon)

Input: Et ordnet array A og et element x

Output: Hvis x er i arrayet A, returner **true** ellers **false**

- Merk ordet *ordnet*
- Det vil si at hvis $0 \leq i \leq j < |A|$, så $A[i] \leq A[j]$
- Et rett-frem søk (som på forrige slide) vil fungere fint!
- Ved å anta at arrayet er ordnet, kan vi finne på noe mye lurere

Binærsøk – idé

- Vi bruker samme idé som du helt naturlig ville brukt, dersom du skal slå opp et navn i en telefonbok
- Utfordringen er å formulere dette som en presis algoritme
    - Altså oversette fremgangsmåten din til entydige steg

Binærsøk – implementasjon

Algorithm 4: Binærsøk

Input: Et ordnet array A og et element x
Output: Hvis x er i arrayet A, returner **true** ellers **false**

Procedure `BinarySearch`(A, x)
    low $\leftarrow 0$
    high $\leftarrow |A| - 1$
    while low $\leq$ high **do**
        $i \leftarrow \lfloor \frac{\text{low}+\text{high}}{2} \rfloor$
        if $A[i] = x$ **then**
            return true
        else if $A[i] < x$ **then**
            low $\leftarrow i + 1$
        else if $A[i] > x$ **then**
            high $\leftarrow i - 1$
    end
    return false

Binærsøk – eksempel 1

low high x: 34

0	1	9	9	18	22	24	24	29	31	31	31	32	33	34	34	39	42	43	46	54	55	56	57	63	63	64	66	70	71	73	76	78	78	82	85	87	88	94	95	97	98
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41

i

low high x: 34

0	1	9	9	18	22	24	24	29	31	31	31	32	33	34	34	39	42	43	46	54	55	56	57	63	63	64	66	70	71	73	76	78	78	82	85	87	88	94	95	97	98
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41

i

low high x: 34

0	1	9	9	18	22	24	24	29	31	31	31	32	33	34	34	39	42	43	46	54	55	56	57	63	63	64	66	70	71	73	76	78	78	82	85	87	88	94	95	97	98
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41

i

Binærsøk – eksempel 2

x: 82

low ↓ (0) high ↓ (49)

2	7	7	8	11	16	17	17	20	21	22	29	31	34	39	39	41	41	42	43	45	47	50	50	54	58	59	63	66	68	70	70	70	70	71	72	73	75	76	82	84	85	86	89	89	91	97	98	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49

↑ i (24)

x: 82

low ↓ (25) high ↓ (49)

2	7	7	8	11	16	17	17	20	21	22	29	31	34	39	39	41	41	42	43	45	47	50	50	54	58	59	63	66	68	70	70	70	70	71	72	73	75	76	82	84	85	86	89	89	91	97	98	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49

↑ i (37)

x: 82

low ↓ (38) high ↓ (49)

2	7	7	8	11	16	17	17	20	21	22	29	31	34	39	39	41	41	42	43	45	47	50	50	54	58	59	63	66	68	70	70	70	70	71	72	73	75	76	82	84	85	86	89	89	91	97	98	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49

↑ i (43)

Binærsøk – eksempel 2

x: 82

																																						low ↓				high ↓							
2	7	7	8	11	16	17	17	20	21	22	29	31	34	39	39	41	41	42	43	45	47	50	50	54	58	59	63	66	68	70	70	70	70	71	72	73	75	76	82	84	85	86	89	89	91	97	98	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
																																								↑ i									

x: 82

																																						low ↓	high ↓										
2	7	7	8	11	16	17	17	20	21	22	29	31	34	39	39	41	41	42	43	45	47	50	50	54	58	59	63	66	68	70	70	70	70	71	72	73	75	76	82	84	85	86	89	89	91	97	98	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
																																						↑ i											

x: 82

																																							low high ↓										
2	7	7	8	11	16	17	17	20	21	22	29	31	34	39	39	41	41	42	43	45	47	50	50	54	58	59	63	66	68	70	70	70	70	71	72	73	75	76	82	84	85	86	89	89	91	97	98	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49
																																							↑ i										

Binærsøk – eksempel 3

low ↓ x: 30 high ↓

0	3	5	8	11	13	14	16	17	19	19	24	24	35	37	42	45	47	52	53	54	55	57	58	59	60	62	63	64	64	66	70	72	76	76	80	88	92	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

↑ i

low ↓ high ↓ x: 30

0	3	5	8	11	13	14	16	17	19	19	24	24	35	37	42	45	47	52	53	54	55	57	58	59	60	62	63	64	64	66	70	72	76	76	80	88	92	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

↑ i

low ↓ high ↓ x: 30

0	3	5	8	11	13	14	16	17	19	19	24	24	35	37	42	45	47	52	53	54	55	57	58	59	60	62	63	64	64	66	70	72	76	76	80	88	92	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

↑ i

Binærsøk – eksempel 3

low ↓ (10) high ↓ (13) x: 30

0	3	5	8	11	13	14	16	17	19	19	24	24	35	37	42	45	47	52	53	54	55	57	58	59	60	62	63	64	64	66	70	72	76	76	80	88	92	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

↑ i (11)

low ↓ (12) high ↓ (13) x: 30

0	3	5	8	11	13	14	16	17	19	19	24	24	35	37	42	45	47	52	53	54	55	57	58	59	60	62	63	64	64	66	70	72	76	76	80	88	92	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

↑ i (12)

low high ↓ (13) x: 30

0	3	5	8	11	13	14	16	17	19	19	24	24	35	37	42	45	47	52	53	54	55	57	58	59	60	62	63	64	64	66	70	72	76	76	80	88	92	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

↑ i (13)

high ↓ low ↓ x: 30

0	3	5	8	11	13	14	16	17	19	19	24	24	35	37	42	45	47	52	53	54	55	57	58	59	60	62	63	64	64	66	70	72	76	76	80	88	92	99	100
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39

O-notasjon

Algoritmeanalyse

Grunnleggende spørsmål:

1. Hvor vanskelig er et problem å løse?
2. Hvordan sammenligne algoritmer som løser problemet?

Foretrekker algoritmer som bruker få ressurser:

- tid
- minne

Hvor mange trinn har Eiffeltårnet?

- Problem: Du har bare et kritt og må finne ut hvor mange trinn Eiffeltårnet har.
- Mange mulig måter å løse dette problemet

Naiv metode

1. Marker første trinn.
2. Gå ned, tegn et strek for det markerte trinnet.
3. Gå opp til første umarkerte trinn. Hvis alle er markert, gå ned og returner antall strek. Ellers, gå til 4.
4. Marker det umarkerte trinnet. Gå til 2.

Hvor mange trinn har Eiffeltårnet?

- Problem: Du har bare et kritt og må finne ut hvor mange trinn Eiffeltårnet har.
- Mange mulig måter å løse dette problemet

Mer naiv metode

1. Skriv et vilkårlig tall.
2. Gå opp like mange trinn som det skrevede tallet. Hvis du akkurat når toppen, returner det skrevede tallet. Hvis det er flere eller færre trinn enn det skrevede tallet, gå ned trappene. Gå til 1.

Hvor mange trinn har Eiffeltårnet?

- Problem: Du har bare et kritt og må finne ut hvor mange trinn Eiffeltårnet har.
- Mange mulig måter å løse dette problemet

Mindre naiv metode

1. Skriv ”1” på det første trinnet.
2. Gå opp et trinn.
3. Hvis du står på et umarkert trinn, skriv det siste tallet + 1 på det trinnet og gå til 2.
4. Hvis du står på toppen, returner det siste tallet du skrev.

Hvor mange trinn har Eiffeltårnet?

Naiv metode

1. Marker første trinn.
2. Gå ned, tegn et strek for det markerte trinnet.
3. Gå opp til første umarkerte trinn. Hvis alle er markert, gå ned og returner antall strek. Ellers, gå til 4.
4. Marker det umarkerte trinnet. Gå til 2.

Mer naiv metode

1. Skriv et vilkårlig tall.
2. Gå opp like mange trinn som det skrevede tallet. Hvis du akkurat når toppen, returner det skrevede tallet. Hvis det er flere eller færre trinn enn det skrevede tallet, gå ned trappene. Gå til 1.

Mindre naiv metode

1. Skriv "1" på det første trinnet.
2. Gå opp et trinn.
3. Hvis du står på et umarkert trinn, skriv det siste tallet + 1 på det trinnet og gå til 2.
4. Hvis du står på toppen, returner det siste tallet du skrev.

Må formaliseres for å sammenligne!

Grunnleggende operasjoner

Vi abstraherer:

- implementasjonsdetaljer
- programmeringsspråk
- hardware
- nøyaktig kjøretid
- spesifkke instanser, f.eks. små eller enkle instanser

Definerer primitive steg:

- tilordning ($a = 3$)
- metodekall ($b.method()$)
- aritmetiske operasjoner ($a + b$)
- indeks aksessering (A[n])
- returnering (return a)

Eksempel

Algorithm 5: En enkel søkealgoritme

```
   Input: Et array A og et element x
   Output: Hvis x er i arrayet A, returner true ellers false
1  Procedure Search(A, e)
2      for i ← 0 to |A| − 1 do
3          if A[i] = x then
4              return true
5      end
6      return false
```

2 addisjon, tilordning, og sammenligning, gjort $|A| - 1$ ganger.

(første gang, for i=0, bare tilordning og sammenligning)

3 indeksaksessering og sammenligning, gjort $|A|$ ganger.

4 returnering

6 returnering

Totalt ved false

$2+3 \cdot (|A| - 1)+2|A|+1$

Worst Case vs. Average Case?

Hvorfor beregner vi ikke gjennomsnittskjøretid?

- ganske komplisert (men mulig)!
- for å definere ”gjennomsnitt” trenger vi sannsynlighetsfordelingen for mulige instanser
- istedet bruker vi oftest worst case for sammenligning
- ser kjøretiden som en funksjon over størrelsen på instansen

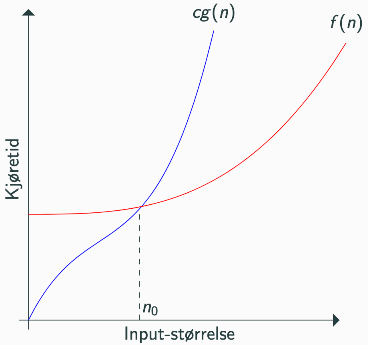
Spørsmål: hvordan utvikler seg kjøretiden når instansen blir stor?

O notasjon

La $f(n)$ være kjøretiden på en instans av størrelse n og la g være en funksjon fra heltall til reelle tall.

$f(n)$ er $O(g(n))$ hvis det finnes en konstant c og en $n_0 \geq 1$ slikt at for alle $n \geq n_0$:

$$f(n) \leq cg(n)$$



O notasjon

La $f(n)$ være kjøretiden på en instans av størrelse n og la g være en funksjon fra heltall til reelle tall.

$f(n)$ er $O(g(n))$ hvis det finnes en konstant c og en $n_0 \geq 1$ slikt at for alle $n \geq n_0$:

$$f(n) \leq cg(n)$$

Eksempler:

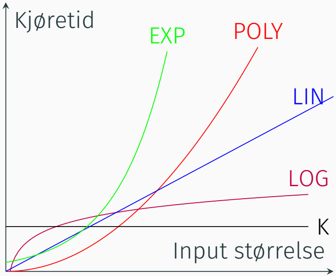
n er $O(n^2)$: siden $n \leq n^2$ for alle $n \geq 0$

$3n$ er $O(n)$: for $c = 4$ er $3n \leq cn = 4n$.

10^{141} er $O(1)$: for $c = 10^{141} + 1$ er $10^{141} \leq 10^{141} + 1$.

O notasjon

Betegnelse	Stor-O	Notat
Konstant tid	$O(1)$	Vokser ikke når n blir større
Logaritmisk tid	$O(log(n))$	I praksis veldig raskt
Lineær tid	$O(n)$	
Kvadratisk tid	$O(n^2)$	Ofte raskt nok
Kubisk tid	$O(n^3)$	
Polynomiell tid	$O(n^k)$	regnes som medgjørlig
Eksponensiell tid	$O(a^n), a > 1$	Ofte for treigt. Blir skjeldent* brukt hvis det finnes polynomielle alternativer.



$*$ Det finnes unntak! F.eks. innen matematisk optimisering

Binærsøk

Algorithm 6: Binærsøk

Input: Et ordnet array A og et element x

Output: Hvis x er i arrayet A, returner **true** ellers **false**

Procedure `BinarySearch`(A, x)
- low $\leftarrow 0$
- high $\leftarrow |A| - 1$
- **while** low $\leq$ high **do**
  - $i \leftarrow \lfloor \frac{\text{low}+\text{high}}{2} \rfloor$
  - **if** $A[i] = x$ **then**
    - **return true**
  - **else if** $A[i] < x$ **then**
    - low $\leftarrow i + 1$
  - **else if** $A[i] > x$ **then**
    - high $\leftarrow i - 1$
- **end**
- **return false**

- $high - low$ er størrelsen av arrayet vi leter i. La n være størrelsen av A.
- $n, \frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots$
- dette gjøres $\log_2(n)$ ganger

$\rightarrow O(log(n))$

Husk at vi abstraherer!

I algoritmeanalyse abstraherer vi mye bort, som gir et bedre sammenligningsgrunnlag. Men disse kan spille en stor rolle likevel!

- $10^{100}n$ er $O(n)$, men de fleste algoritmene kommer til å være raskere enn dette
- En implementasjon av binærsøk som bruker linked lists istedenfor arrays kjører i $O(nlog(n))$ tid, som er dårligere enn rett-frem søk!

Med andre ord: algoritmeanalyse er et nyttig verktøy, men forteller ikke

hele historien.

Træer

Trær

- Det er to store kategorier med anvendelser av trær:
    - Det du jobber med har en iboende hierarkisk struktur
    - Effektiv implementasjon for datasamlinger som mengder og oppslagstabeller
- Dere kjenner allerede til lister, som er defiert som
    - en tom liste — ofte representert med `null` — eller
    - en node som består av en peker til et element og en peker til en liste
- Alle lister *er trær*, men ikke alle trær er lister
- Vi kan se på trær som en enkel utvidelse av lister
    - der vi tillater at en node har flere neste-pekere

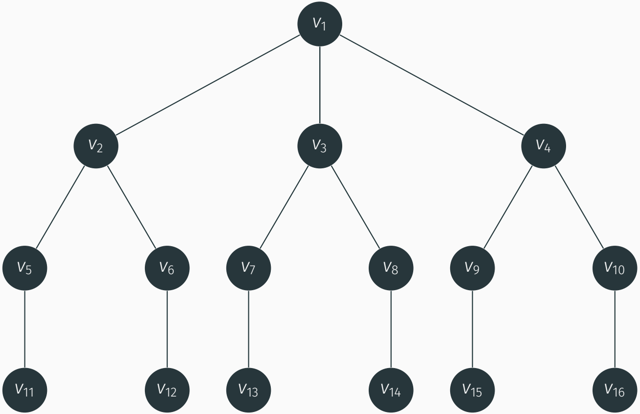
Trær – eksempler

- Syntaksen i programmeringspråk utgjør trær
    - Det første en kompilator gjør, er å gjøre koden din om til et *abstrakt syntakstre*
- HTML er et filformat som lar deg uttrykke trær
    - Så en nettside er bare en bestemt måte å vise frem et tre
- Filsystemer er trær
- Alle mulige sjakkpartier kan representeres som et (enormt) tre

Trær – definisjon

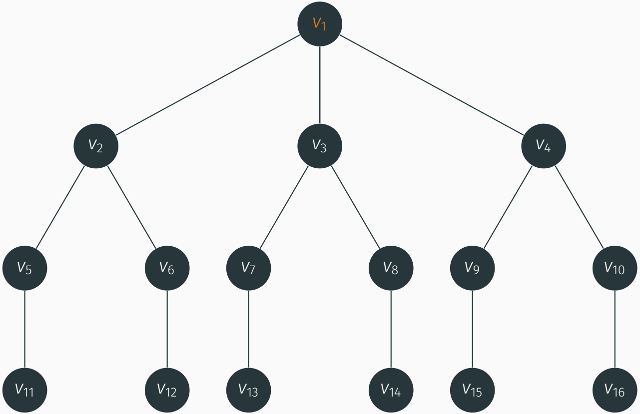
- Et tre er definert som
  - det tomme treet — ofte representert med `null` — eller
  - en node med en peker til
    - et element
    - 0 eller flere pekere til barnenoder, og
    - nøyaktig én forelernode
  - Et tre kan ikke inneholde sykler
    - Altså: fra en node v kan du ikke nå v ved å følge pekere fra v
- Merk: Boka tillater ikke tomme trær

Trær – terminologi



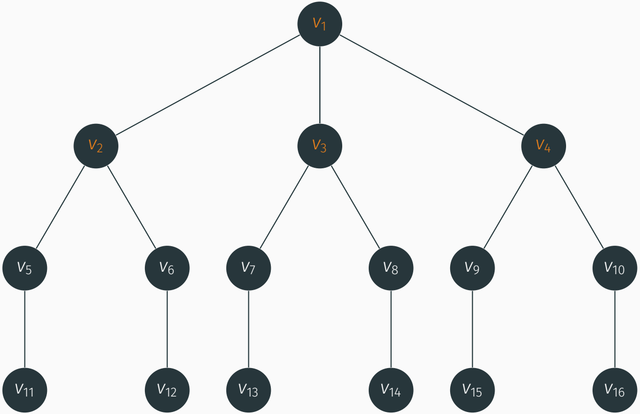
Dette er et tre, hver v_i er en node

Trær – terminologi



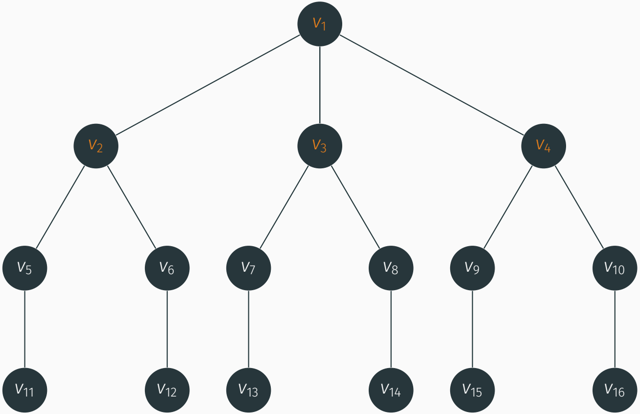
v_1 er *roten* av treet

Trær – terminologi



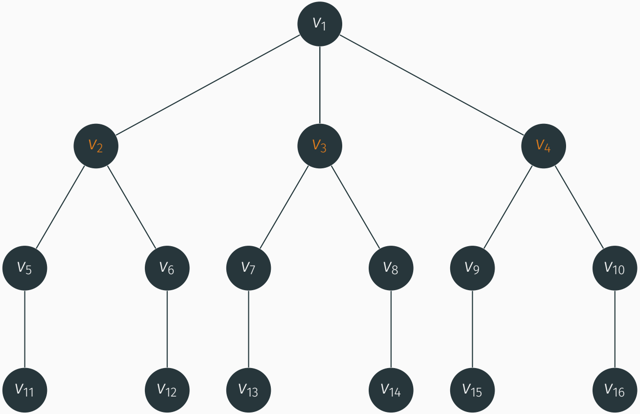
v_2, v_3 og v_4 er *barn* av v_1

Trær – terminologi



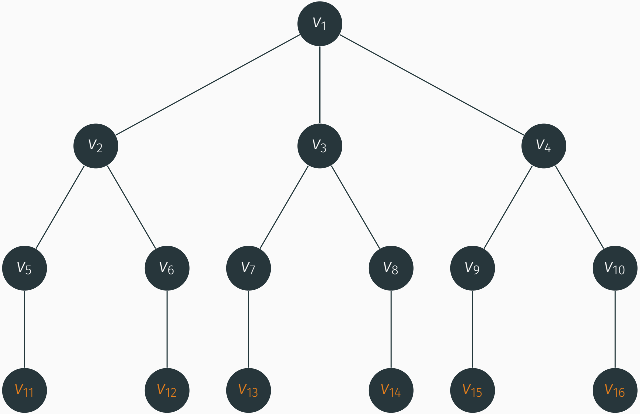
v_1 er *forelder* til v_2, v_3 og v_4

Trær – terminologi



v_2, v_3 og v_4 er *søsken*

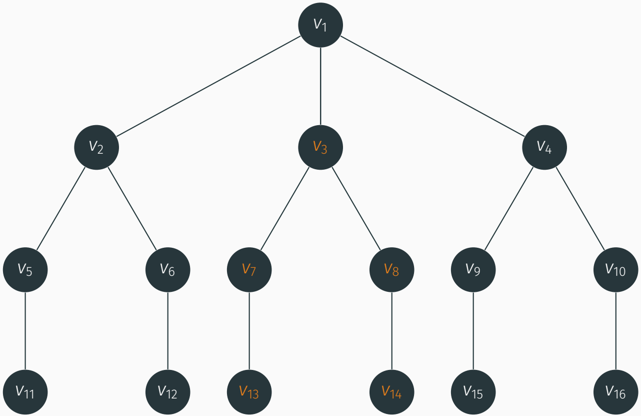
Trær – terminologi



$v_{11}, \ldots, v_{16}$ er *løvnoder*, eller *eksterne* noder

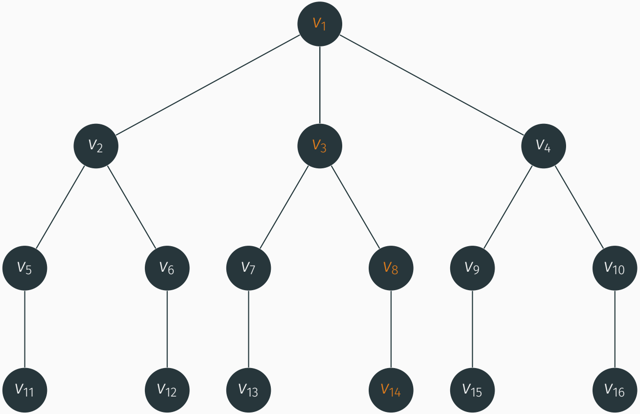
Nodene $v_1, \ldots, v_{10}$ er ikke løvnoder, eller *interne* noder

Trær – terminologi



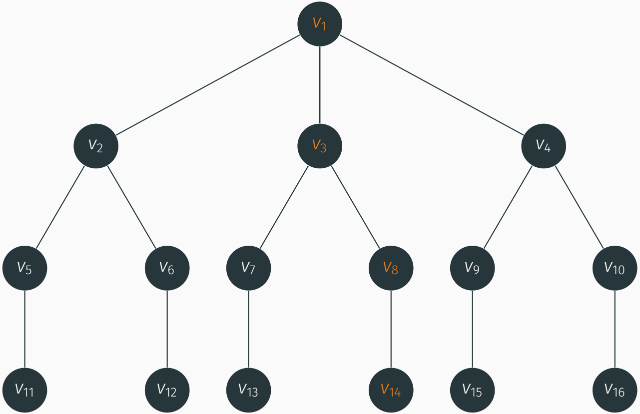
v_3, v_7, v_8, v_{13} og v_{14} utgjør et *subtre*, hvor v_3 er roten

Trær – terminologi



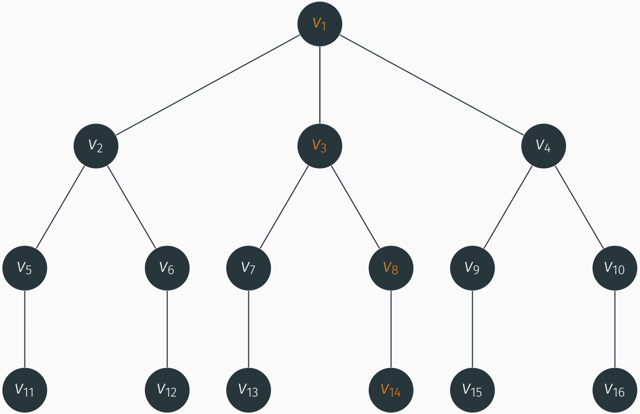
v_1, v_3, v_8 og v_{14} er *forfedre* av v_{14}

Trær – terminologi



v_1, v_3, v_8 og v_{14} er *etterkommere* av v_1

Trær – terminologi



Sekvensen v_1, v_3, v_8, v_{14} kalles en *sti*

Trær – Datastruktur

- **null** representerer et tomt tre
- Anta at v er en node, da gir
  - v.**element** dataen som er lagret i noden
  - v.**parent** foreldrenoden til v
  - v.**children** barnenodene til v

Trær – dybde

- Dybden til en node er én mer enn foreldrenoden
- Roten har dybde 0
- Siden vi tillater et tomt tre gir vi det dybde -1

Algorithm 7: Finn dybden av en gitt node

Input: En node v
Output: Dybden av noden

```
Procedure Depth(v)
    if v = null then
        return −1
    return 1 + Depth(v.parent)
```

Trær – høyde

- Høyden av et tre er gitt av den høyeste avstanden til en etterkommer
- Det vil si dybden av den dypeste *løvnoden*

Algorithm 8: Finn høyden av en gitt node

Input: En node v
Output: Høyden av noden

```
Procedure Height(v)
    if v = null then
        return −1
    h ← 0
    for v′ ∈ v.children do
        h ← Max(h, Height(v′))
    end
    return 1 + h
```

Trær – traversering

- Vi har måter for å systematisk gå gjennom (traversere) et tre på
- Underveis har vi en operasjon vi ønsker å utføre
- For mange operasjoner har *rekkefølgen* vi utfører operasjonen i en betydning
    - *preorder* utfører operasjonen på seg selv først, og barna etterpå
    - *postorder* utfører operasjonen på barna først, og seg selv etterpå
- For å kopiere et tre kan man bruke *preorder*, men ikke *postorder*
- For å slette et tre kan man bruke *postorder*, men ikke *preorder*

Trær – preorder og postorder

Algorithm 9: Preorder traversering

Input: En node v (som ikke er **null**)

Output: Utfør en operasjon på v først og barna til v etterpå

```
Procedure Preorder(v)
    Operate on v
    for v′ ∈ v.children do
        Preorder(v)
    end
```

Algorithm 10: Postorder traversering

Input: En node v (som ikke er **null**)

Output: Utfør en operasjon på barna til v først og v etterpå

```
Procedure Postorder(v)
    for v′ ∈ v.children do
        Postorder(v)
    end
    Operate on v
```

Binære søketrær

Binære trær

- Et binærtre er et tre hvor hver node har maksimalt to barn
- I binære trær referer vi til *venstre* og *høyre* barn
- Hvis *v* er en node i et binærtre, så gir
    - *v*.**element** dataen som er lagret i noden
    - *v*.**left** venstre barn av *v*
    - *v*.**right** høyre barn av *v*

Binære søketrær

- Et binært søketre er et binærtre som oppfyller følgende egenskap
  - For hver node *v* så er *v*.**element**
    - større enn alle elementer i venstre subtre, og
    - mindre enn alle elementer i høyre subtre
- Merk at vi kan si større *eller lik* dersom vi ønsker å tillate duplikater
- For at vi skal kunne bruke binære søketrær må elementene være *sammenlignbare*
- Binære trær er spesielt gode når de er *balanserte*
  - Dette er tema for neste uke

Sammenheng mellom binærsøk og binære søketrær

- Idéen bak binærsøk er å *halvere søkerommet* hver gang vi gjør en sammenligning, som gir $O(log(n))$ tid på oppslag
- Det fungerer strålende, men fordrer at vi har et *sortert* array
- Et problem oppstår når vi trenger en *dynamisk struktur*
    - En datastruktur hvor vi stadig legger til og fjerner elementer
- Et binært søketre er en datastruktur
    - som gjør binærsøk *enkelt*
    - støtter effektiv innsetting og sletting

Innsetting

Algorithm 11: Innsetting i et binært søketre

Input: En node v og et element x

Output: En oppdatert node v der en node som inneholder x er en etterkommer av v

```
Procedure Insert(v, x)
    if v = null then
        v ← new Node(x)
    else if x < v.element then
        v.left ← Insert(v.left, x)
    else if x > v.element then
        v.right ← Insert(v.right, x)
    return v
```

- Denne algoritmen har kompleksitet $O(h)$
    - der h er høyden på treet
- Dersom n er antall noder i treet har vi $O(n)$ i verste tilfelle
    - men hvis treet er balansert,
    - så er kompleksiteten $O(log(n))$

Oppslag

Algorithm 12: Oppslag i et binært søketre

Input: En node v og et element x

Output: Dersom x forekommer i en node u som en etterkommer av v, returner u, ellers **null**.

```
Procedure Search(v, x)
    if v = null then
        return null
    if v.element = x then
        return v
    if x < v.element then
        return Search(v.left, x)
    if x > v.element then
        return Search(v.right, x)
```

- Oppslag i et binærtre har samme kompleksitet som innsetting

Sletting

- Sletting fra et binærtre er litt vanskeligere enn innsetting og oppslag
  - men har samme kompleksitet!
- Vi må passe på å tette eventuelle ”hull”
- Vi skiller mellom tre tilfeller
  - Noden vi vil slette har ingen barn
  - Noden vi vil slette har ett barn
  - Noden vi vil slette har to barn

Finn minste

- For sletting trenger vi en prosedyre for å finne minste element

Algorithm 13: Finn minste node

Input: En node v

Output: Returner noden som inneholder den minste etterkommeren av v

```
Procedure FindMin(v)
    Etterlatt som øvelse!
```

Sletting

Algorithm 14: Slett en node i et binært søketre

Input: En node v og et element x

Output: Dersom x forekommer i en node u som en etterkommer av v, fjern u.

```
Procedure Remove(v, x)
    if v = null then
        return null
    if x < v.element then
        v.left ← Remove(v.left, x)
        return v
    if x > v.element then
        v.right ← Remove(v.right, x)
        return v
    if v.left = null then
        return v.right
    if v.right = null then
        return v.left
    u ← FindMin(v.right)
    v.element ← u.element
    v.right ← Remove(v.right, u.element)
    return v
```