Prioritetskøer



Repetisjonstime

Abstrakte datatyper

Sier noe om oppførsel, f.eks. hvilke metoder datastrukturen har, men ingenting om implementasjon

Den konkrete implementasjonen er en **datastruktur**

Kø

Enqueue(x)

Dequeue( )


ESCAPE

Enqueue(Jens)



Enqueue(Sofia)



Enqueue(Tom)



Dequeue( )



Dequeue( )



Prioritetskø

Også en abstrakt datatype!

Insert(e)

RemoveMin( )

Implementasjon: Usortert lenket liste



Implementasjon: Usortert lenket liste

Insert(e) – O(1)



Implementasjon: Usortert lenket liste

Insert(e) – O(1)



Implementasjon: Usortert lenket liste

RemoveMin( )



Implementasjon: Usortert lenket liste

RemoveMin( )



Implementasjon: Usortert lenket liste

RemoveMin( )



Implementasjon: Usortert lenket liste

RemoveMin( )



Implementasjon: Usortert lenket liste

RemoveMin( )



Implementasjon: Usortert lenket liste

RemoveMin( ) – O(n)



Implementasjon: Sortert lenket liste

Insert(e) – O(n)



Implementasjon: Sortert lenket liste

Insert(e) – O(n)



Implementasjon: Sortert lenket liste

Insert(e) – O(n)



Implementasjon: Sortert lenket liste

Insert(e) – O(n)



Implementasjon: Sortert lenket liste

Insert(e) – O(n)



Implementasjon: Sortert lenket liste

Insert(e) – O(n)



Implementasjon: Sortert lenket liste

Insert(e) – O(n)



Implementasjon: Sortert lenket liste

RemoveMin( )



Implementasjon: Sortert lenket liste

RemoveMin( ) = O(1)



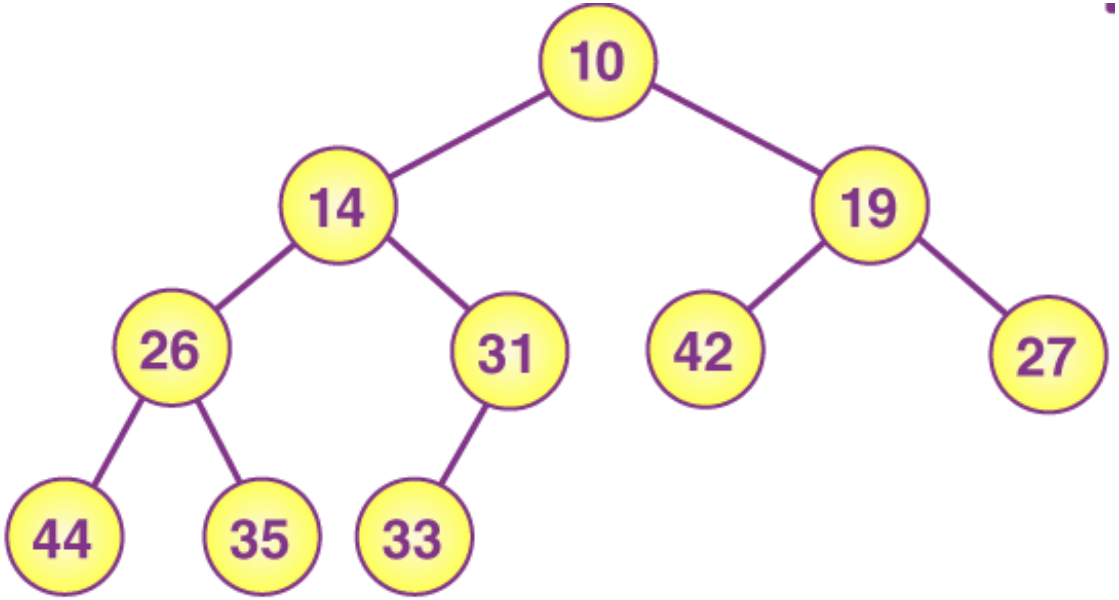
Prioritetskø - Implementasjon

- Usortert lenket liste: O(1) på insert, O(n) på removeMin
- Sortert lenket liste: O(n) på insert, O(1) på removeMin

- AVL trær: O(log(n)) på insert, O(log(n)) på removeMin

Vi skal lære om hvordan implementere med binære heaps!

Binære heaps



Hva er binære heaps?

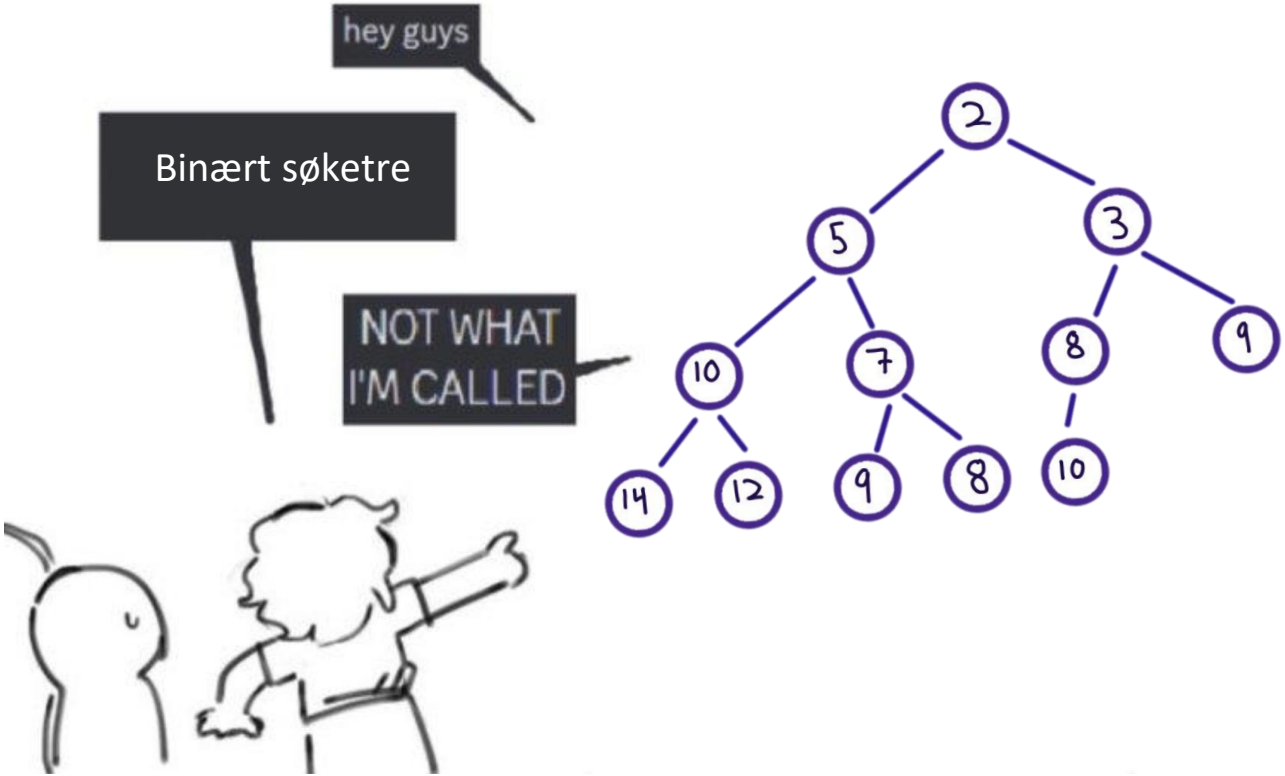
- Et balansert binært tre (Men ikke et balansert binært **søke**tre)

Annerledes enn AVL trær:

AVL:
Venstre barn er lavere enn noden
Høyre barn er høyere enn noden

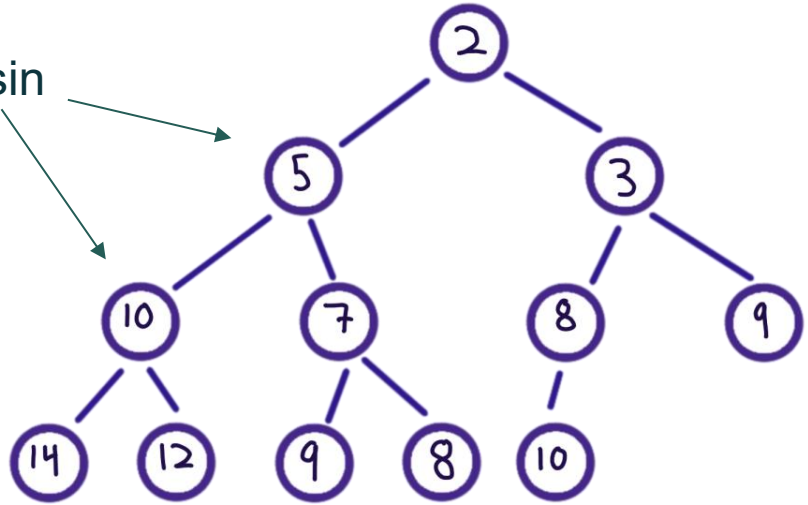
Heaps:
Alle noder er større (eller lik) foreldernoden sin

AVL vs heaps


hey guys
Binært søketre
NOT WHAT I'M CALLED
2
5
3
10
7
8
9
14
12
9
8
10

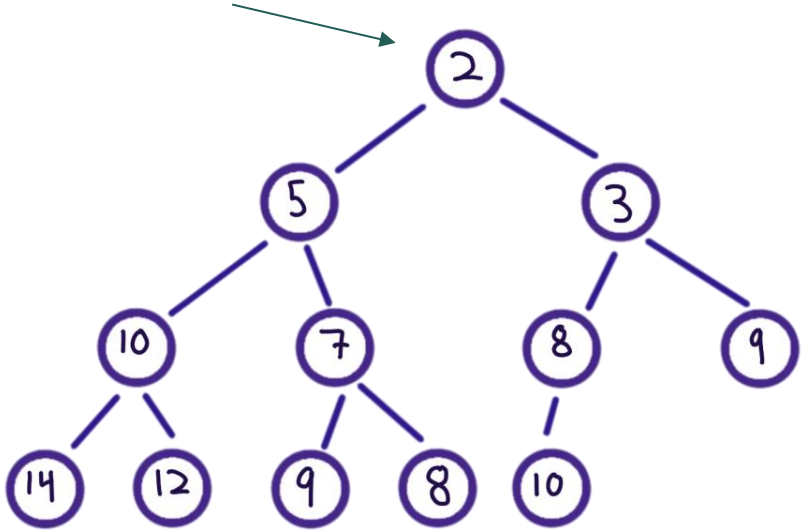
AVL vs heaps

Venstre barna er større enn foreldrenoden sin



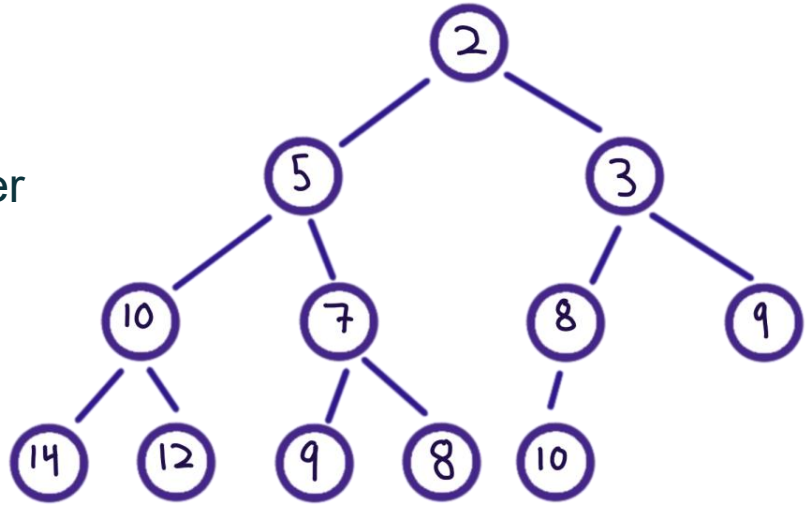
AVL vs heaps

Minste tallet er på topp

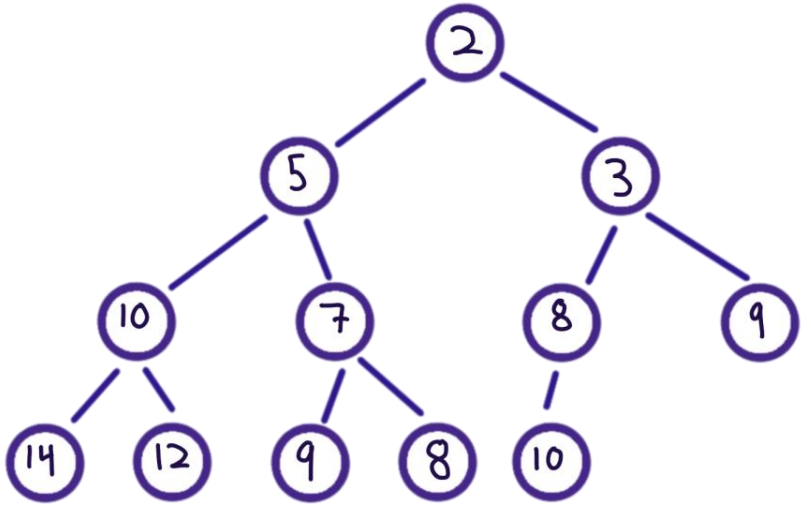


AVL vs heaps

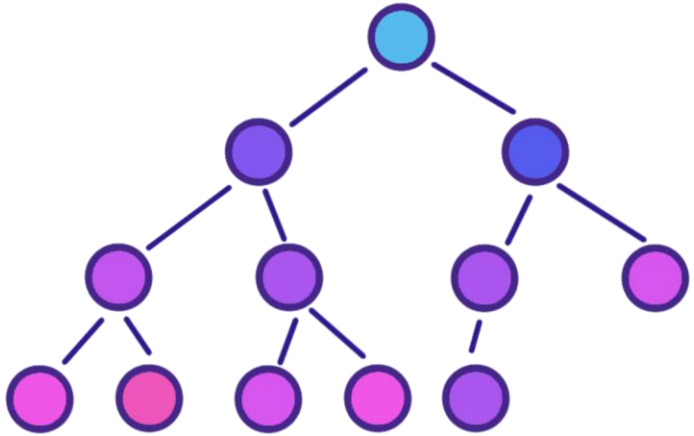
Blir større jo lenger ned du går



Min-heap



Min-heap

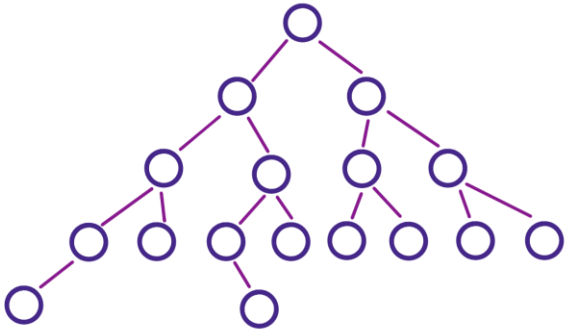


Heaps må også være komplette!

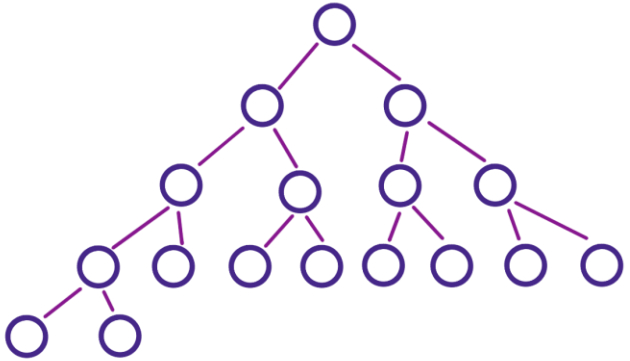
Hva er et komplett tre???

- Et tre som fylles opp fra venstre til høyre

Ikke valid:

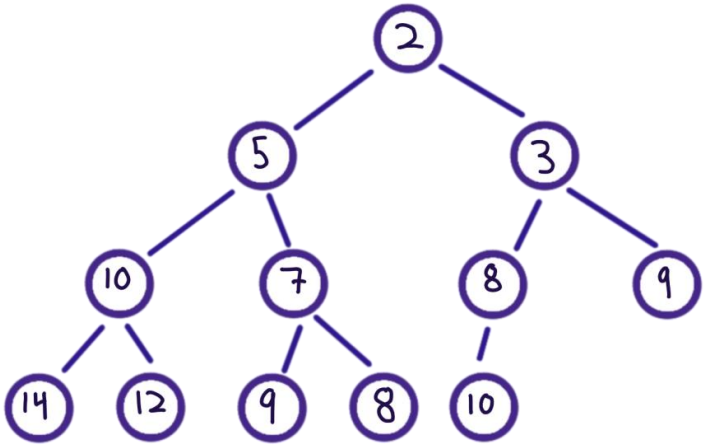


Valid:



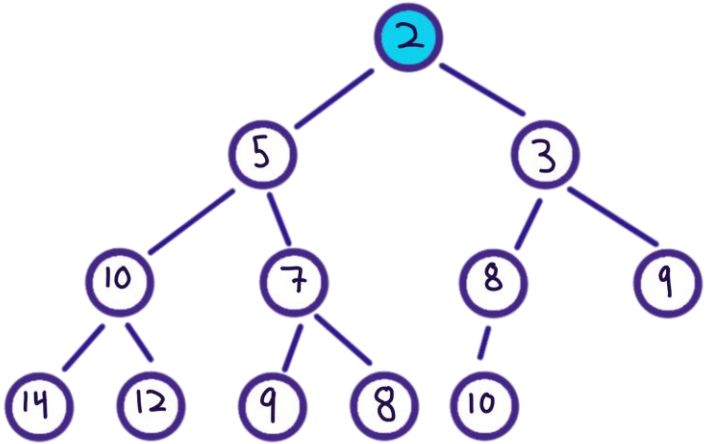
Heaps i praksis

- Vi konstruerer en heap ved å bruke et array

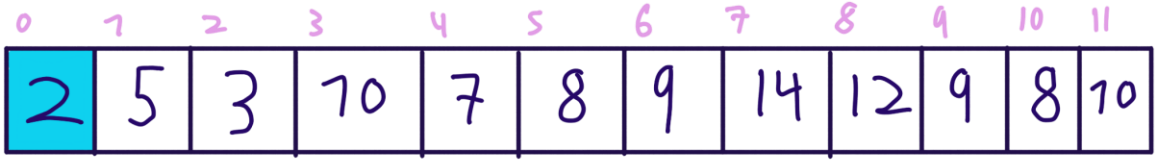


Heaps i praksis

- Vi konstruerer en heap ved å bruke et array

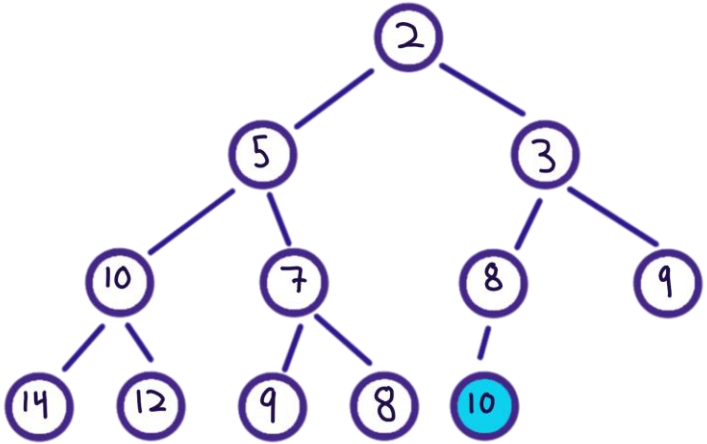


A[0] = rotnoden

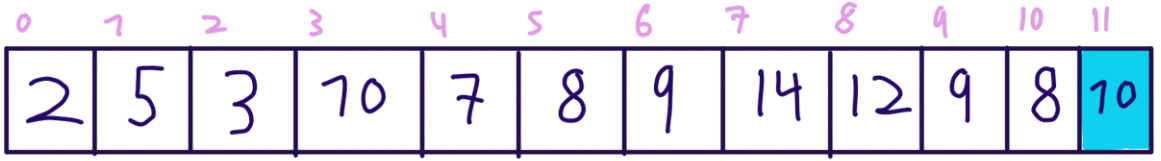


Heaps i praksis

- Vi konstruerer en heap ved å bruke et array



A[n-1] = Siste noden i treet



Tre veldig viktige hjelpemetoder

```
1 Procedure ParentOf(i)
2   return ⌊(i−1)/2⌋
```

```
1 Procedure LeftOf(i)
2   return 2 · i + 1
```

```
1 Procedure RightOf(i)
2   return 2 · i + 2
```

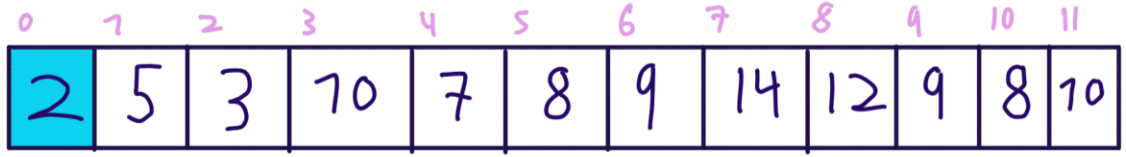
Tre veldig viktige hjelpemetoder

```
1 Procedure ParentOf(i)
2 |  return ⌊(i−1)/2⌋
```

```
1 Procedure LeftOf(i)
2 |  return 2 · i + 1
```

```
1 Procedure RightOf(i)
2 |  return 2 · i + 2
```

a) Hva er forelderen til tallet ‘12’?
b) Hva er det høyre barnet til tallet ‘3’?
c) Hva er det venstre barnet til tallet ‘7’



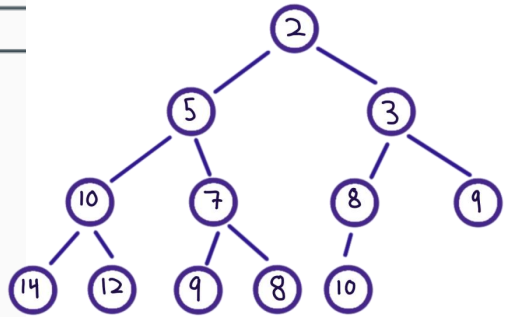
Innsetting i heaps

- Setter inn det nye elementet ved posisjon A[n]
- ‘Shuffler’ det nye elementet opp til den har en foreldrenode som er mindre enn den

ALGORITHM: INNSETTING I HEAP

Input: Et array A som representerer en heap med n elementer, og et element x
Output: Et array som representerer en heap, som inneholder x

```
1 Procedure Insert(A, x)
2     A[n] ← x
3     i ← n
4     while 0 < i and A[i] < A[ParentOf(i)] do
5         swap A[i] and A[ParentOf(i)]
6         i ← ParentOf(i)
```

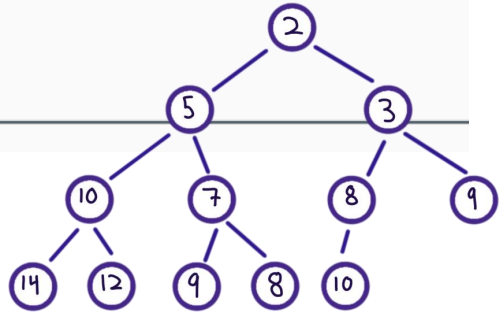


Sletting i heaps

ALGORITHM: FJERNING AV MINSTE ELEMENT FRA HEAP

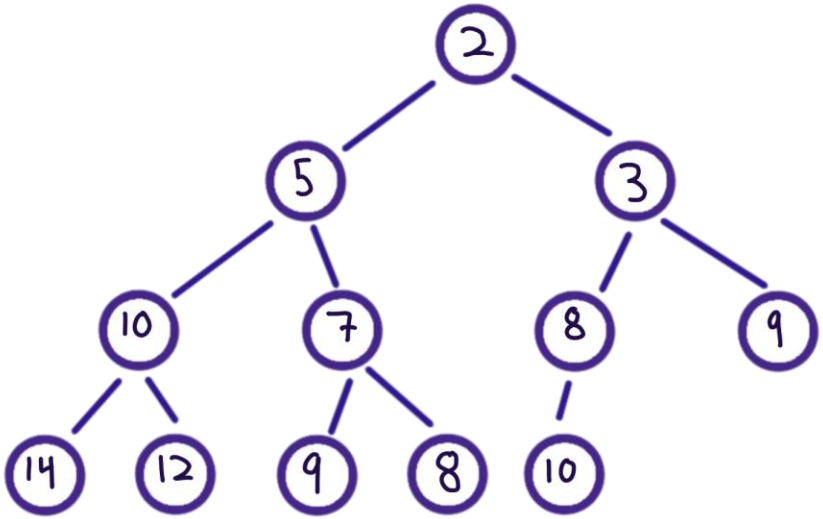
Input: Et array A som representerer en ikke-tom heap med n elementer
Output: Det minste elementet fjernes og returneres

```
1  Procedure RemoveMin(A)
2      x ← A[0]
3      A[0] ← A[n − 1]
4      i ← 0
5      while LeftOf(i) < n − 1 do
6          j ← LeftOf(i)
7          if RightOf(i) < n − 1 and A[RightOf(i)] < A[j] then
8              j ← RightOf(i)
9          if A[i] ≤ A[j] then
10             return x
11         swap A[i] and A[j]
12         i ← j
13     return x
```

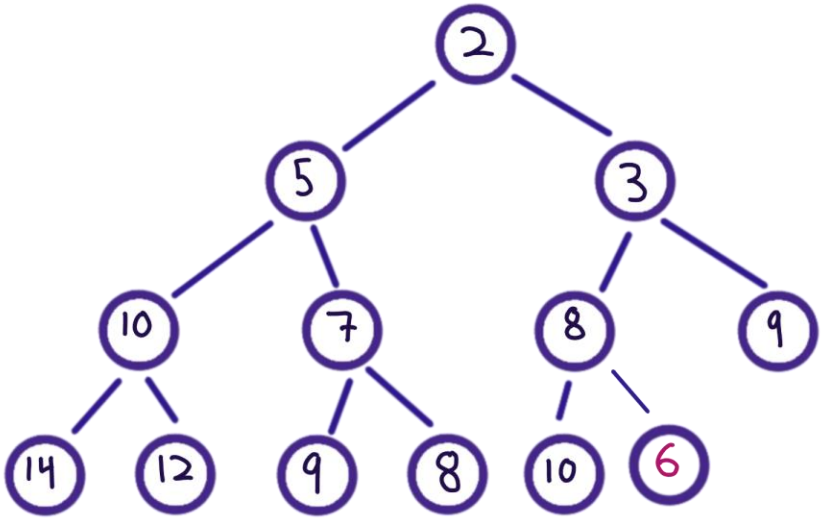


- Fjerner roten (som vi vet er minst)
- ‘Oppgraderer’ siste node (A[n-1]) til den nye roten
- ‘Shuffler’ den nye roten nedover til den har barn som er større enn seg selv

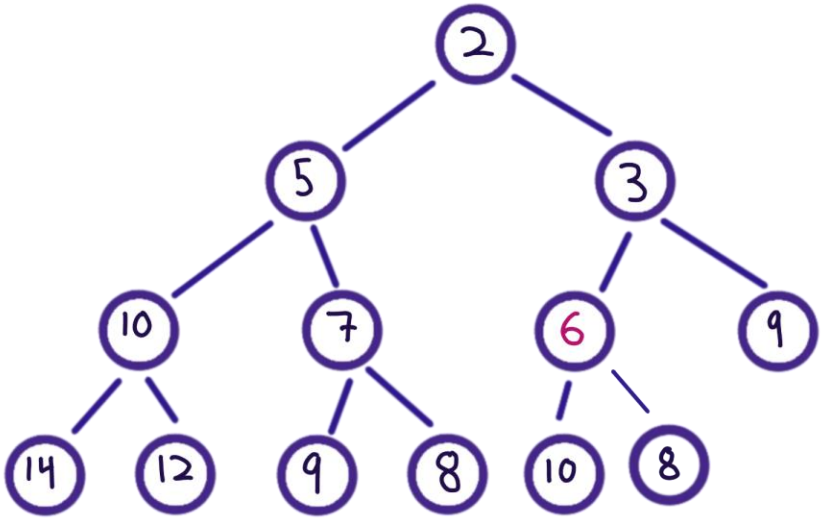
Hva skjer dersom vi setter inn tallet ‘6’ i heapen?



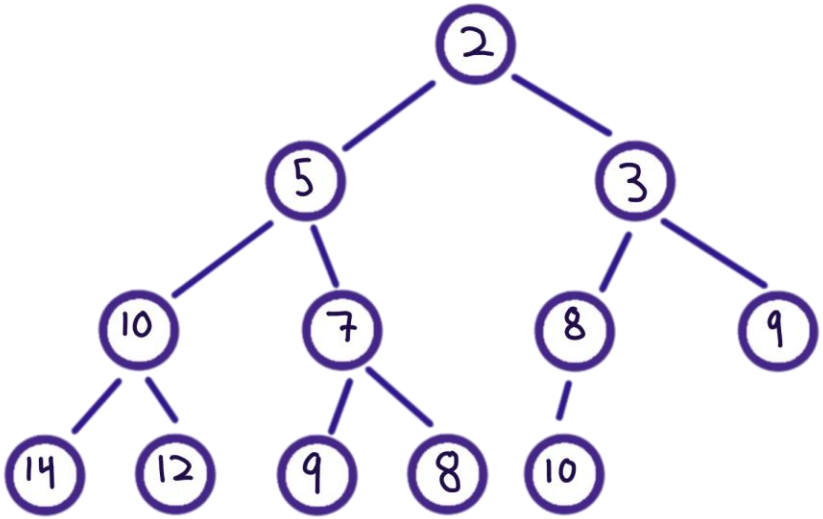
Hva skjer dersom vi setter inn tallet ‘6’ i heapen?



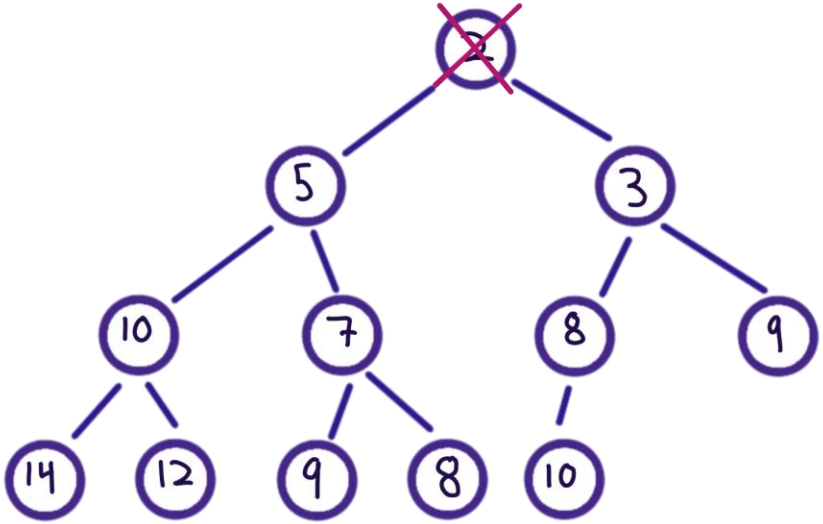
Hva skjer dersom vi setter inn tallet ‘6’ i heapen?



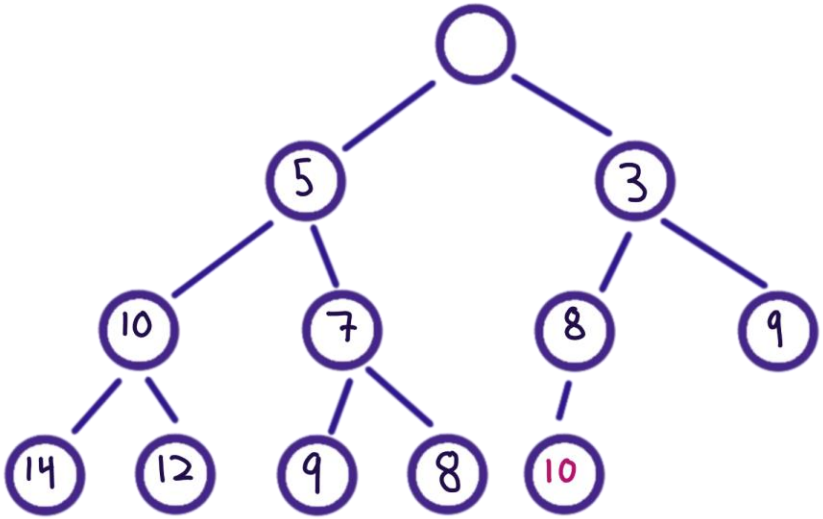
Hva skjer dersom vi gjøre removeMin() på heapen?



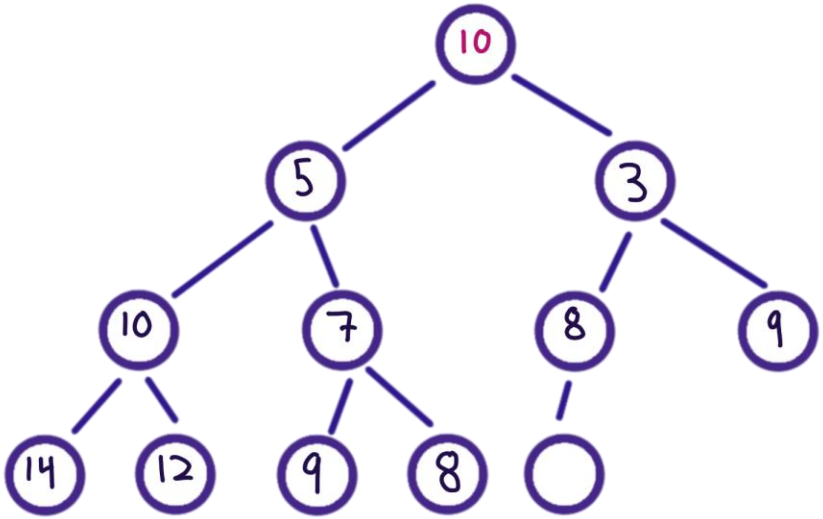
Hva skjer dersom vi gjøre removeMin() på heapen?



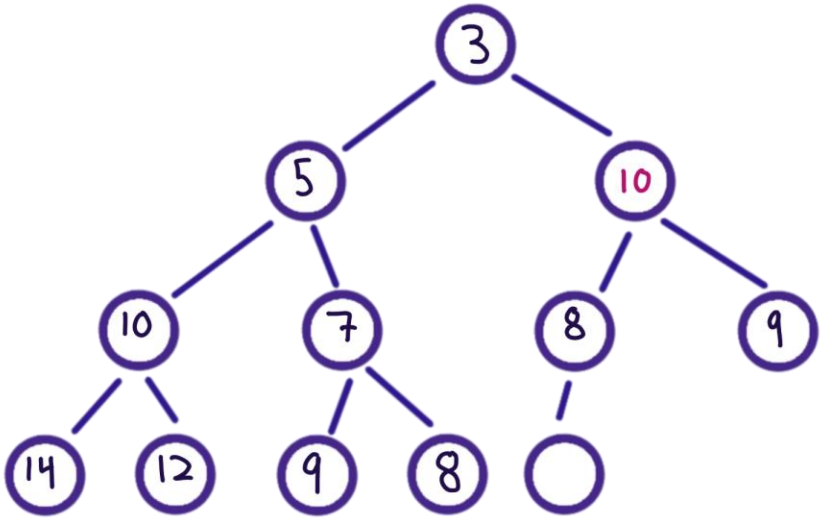
Hva skjer dersom vi gjøre removeMin() på heapen?



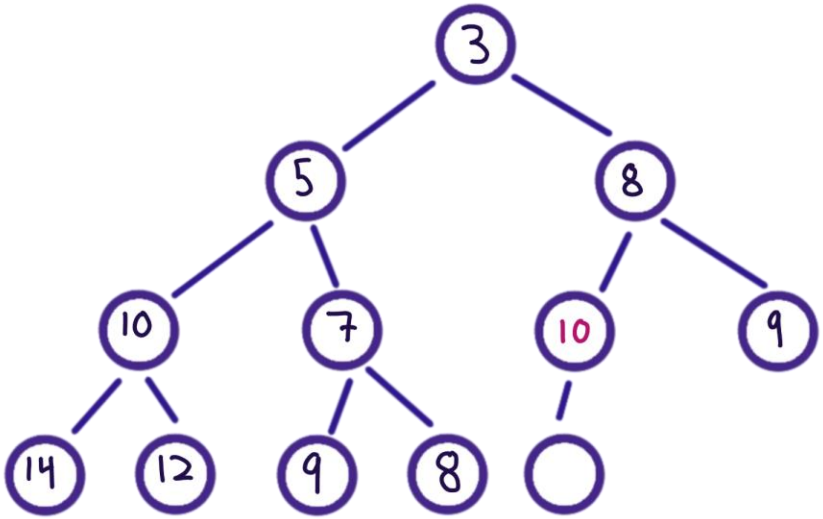
Hva skjer dersom vi gjøre removeMin() på heapen?



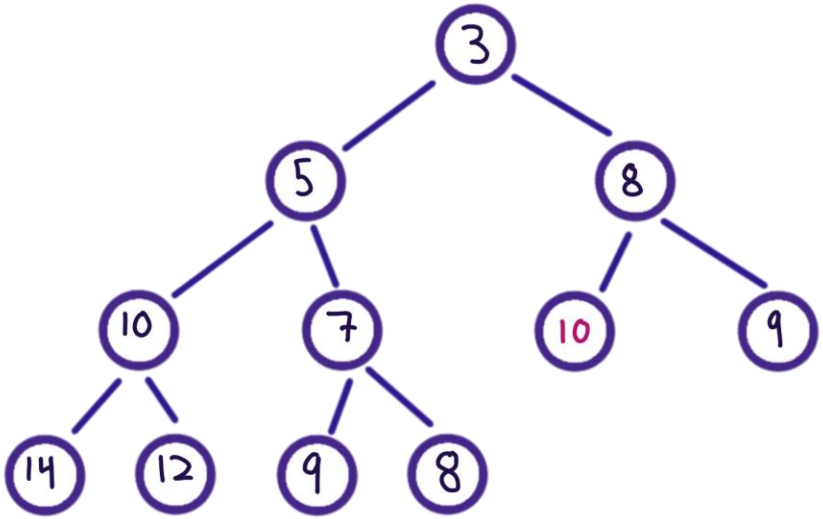
Hva skjer dersom vi gjøre removeMin() på heapen?



Hva skjer dersom vi gjøre removeMin() på heapen?



Hva skjer dersom vi gjøre removeMin() på heapen?



Litt om prioritetskøer og binære heaps 10 poeng

Med *heap* mener vi arrayer som representerer binære min-heaps.

(a) ~~En array med n elementer kan gjøres om til en heap i $\mathcal{O}(n)$.~~

(b) En min-heap blir en max-heap ved å reversere arrayet.

(c) Alle elementer på dybde d i en heap er mindre enn alle elementer på dybde $d+1$.

(d) Innsetting i en heap med n elementer er $\mathcal{O}(\log(n))$ i verste tilfelle.

(e) Et AVL-tre kan brukes som en prioritetskø med samme kjøretidskompleksitet som en heap.

(f) Nodene langs en sti fra rotnoden til en løvnode i en heap er ordnet fra minst til størst.

(g) Dersom vi bytter plass på to søskennoder i en heap, er det fremdeles en heap.

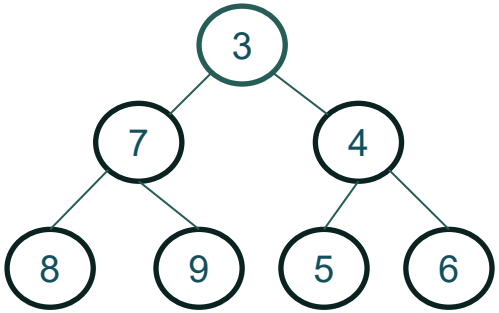
(h) En prioritetskø med konstant tid på alle operasjoner ville gjort at Dijkstra hadde kjøretidskompleksitet $\mathcal{O}(|V|+|E|)$.

(i) Man kan finne det *største* elementet i en *min*-heap i $\mathcal{O}(\log(n))$.

(j) I en heap ligger *over halvparten* av elementene på de *to dypeste* nivåene.

(b) En min-heap blir en max-heap ved å reversere arrayet.

3	7	4	8	9	5	6

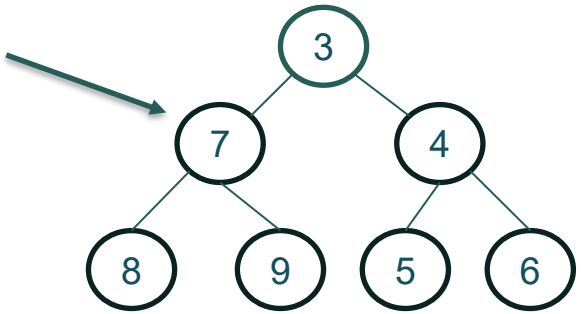


(b) En min-heap blir en max-heap ved å reversere arrayet.

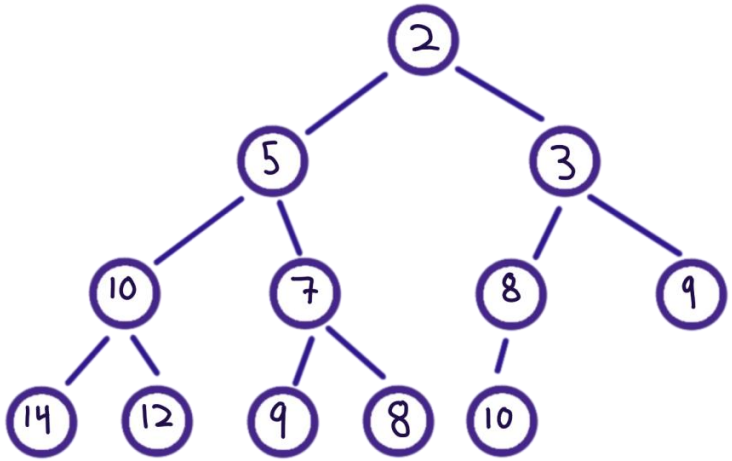
6	5	9	8	4	7	3



(c) Alle elementer på dybde d i en heap er mindre enn alle elementer på dybde $d + 1$.



(d) Innsetting i en heap med n elementer er $\mathcal{O}(\log(n))$ i verste tilfelle.

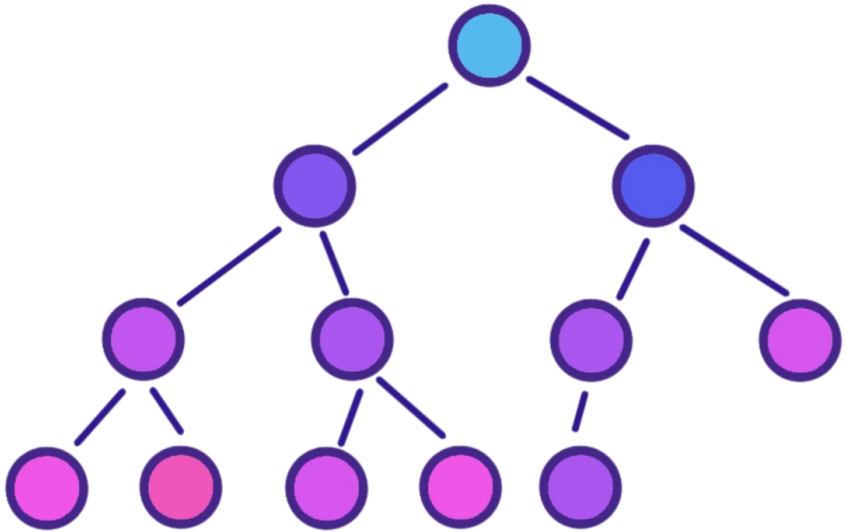


(e) Et AVL-tre kan brukes som en prioritetskø med samme kjøretidskompleksitet som en heap.

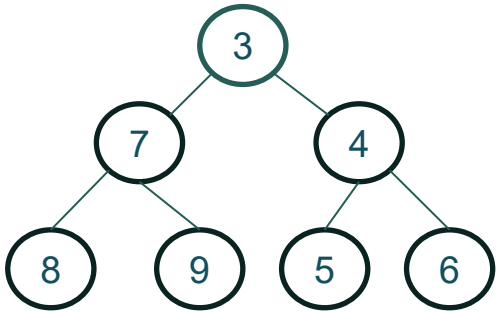
Sant!

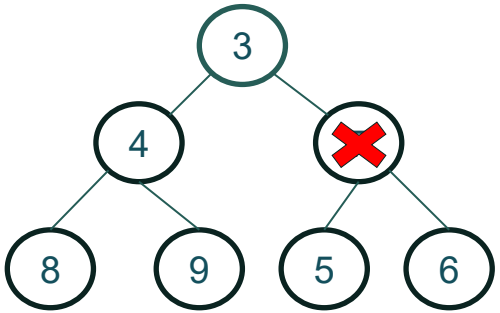
Men…

(f) Nodene langs en sti fra rotnoden til en løvnode i en heap er ordnet fra minst til størst.



(g) Dersom vi bytter plass på to søskennoder i en heap, er det fremdeles en heap.





(h) En prioritetskø med konstant tid på alle operasjoner ville gjort at Dijkstra hadde kjøretidskompleksitet $\mathcal{O}(|V| + |E|)$.

RemoveMin() og DecreasePriority() er egentlig O(log(n))

Dijkstra - Kompleksitet

- Det er |V| på prioritetskøen
- Så vi gjør RemoveMin |V| ganger
- O(|V| * log(|V|))

- Vi besøker alle kanter (E) èn gang
- Så vi gjør DecreasePriority |E| ganger
- O(|E| * log(|V|))

ALGORITHM: DIJKSTRAS ALGORITME FOR KORTESTE STIER (T

Input: En vektet graf $G = (V, E)$ med vektfunksjon w og en startnc
Output: Et map `dist` som angir korteste vei fra s til alle noder i G

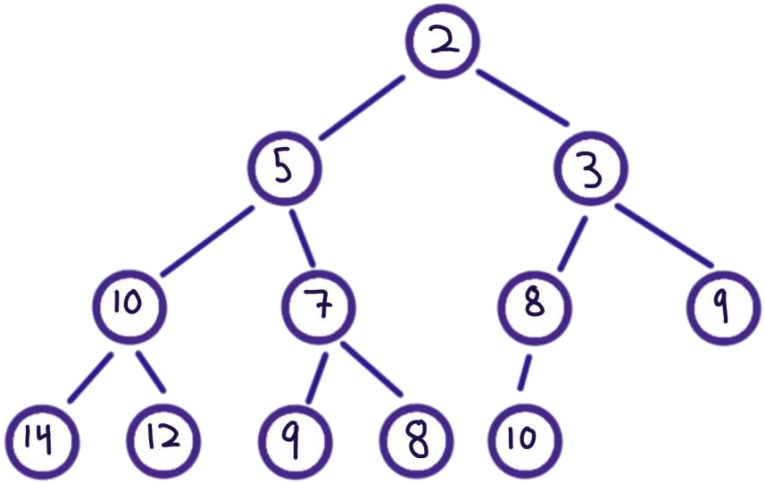
```
1  Procedure Dijkstra(G, s)
2      queue ← empty priority queue
3      dist ← empty map
4      for v ∈ V do
5          dist[v] ← ∞
6          Insert(queue, v) with priority ∞
7      dist[s] ← 0
8      DecreasePriority(queue, s, 0)
9
10     while queue is not empty do
11         u ← RemoveMin(queue)
12         for (u, v) ∈ E do
13             c ← dist[u] + w(u, v)
14             if c < dist[v] then
15                 dist[v] ← c
16                 DecreasePriority(queue, v, c)
17     return dist
```

(h) En prioritetskø med konstant tid på alle operasjoner ville gjort at Dijkstra hadde kjøretidskompleksitet $\mathcal{O}(|V| + |E|)$.

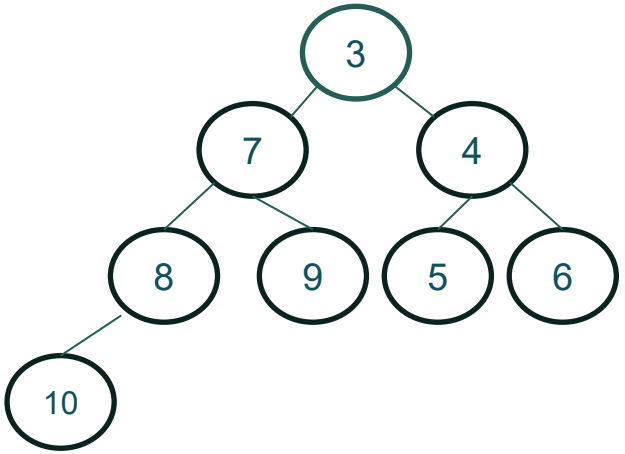
- Det er |V| på prioritetskøen
- Så vi gjør RemoveMin |V| ganger
- O(|V| * 1) = O(|V|)

- Vi besøker alle kanter (E) èn gang
- Så vi gjør DecreasePriority |E| ganger
- O(|E| * 1) = O(|E|)

(i) Man kan finne det *største* elementet i en *min*-heap i $\mathcal{O}(\log(n))$.



(j) I en heap ligger *over halvparten* av elementene på de *to dypeste* nivåene.



Huffman-koding

Strengrepresentasjon

La oss si at dette er deg

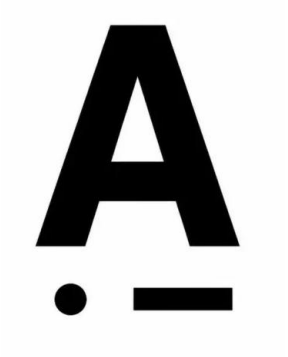


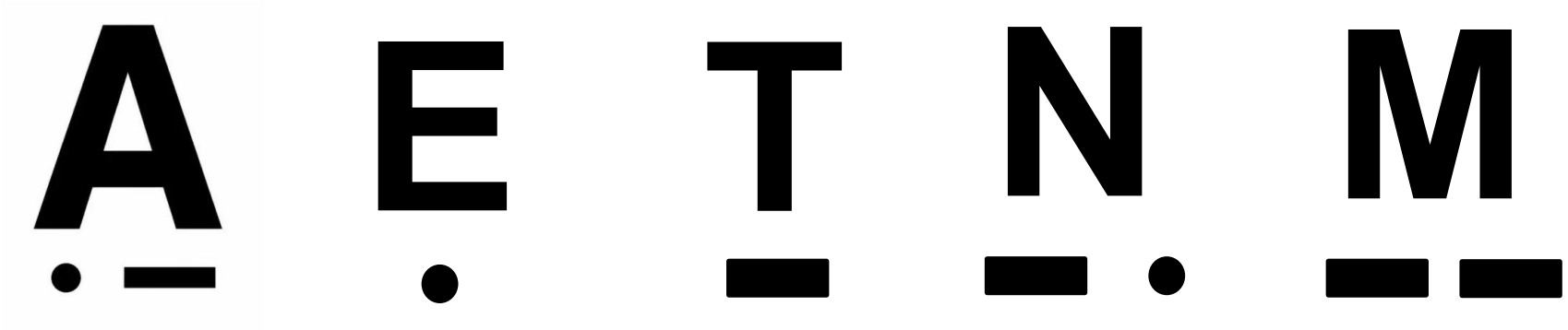
NB! Slides med stoff som ikke er pensum er rosa



- Morsekode fungerer veldig likt som binærtall
- Alt er enten et langt pip (1) eller et kort pip (0)

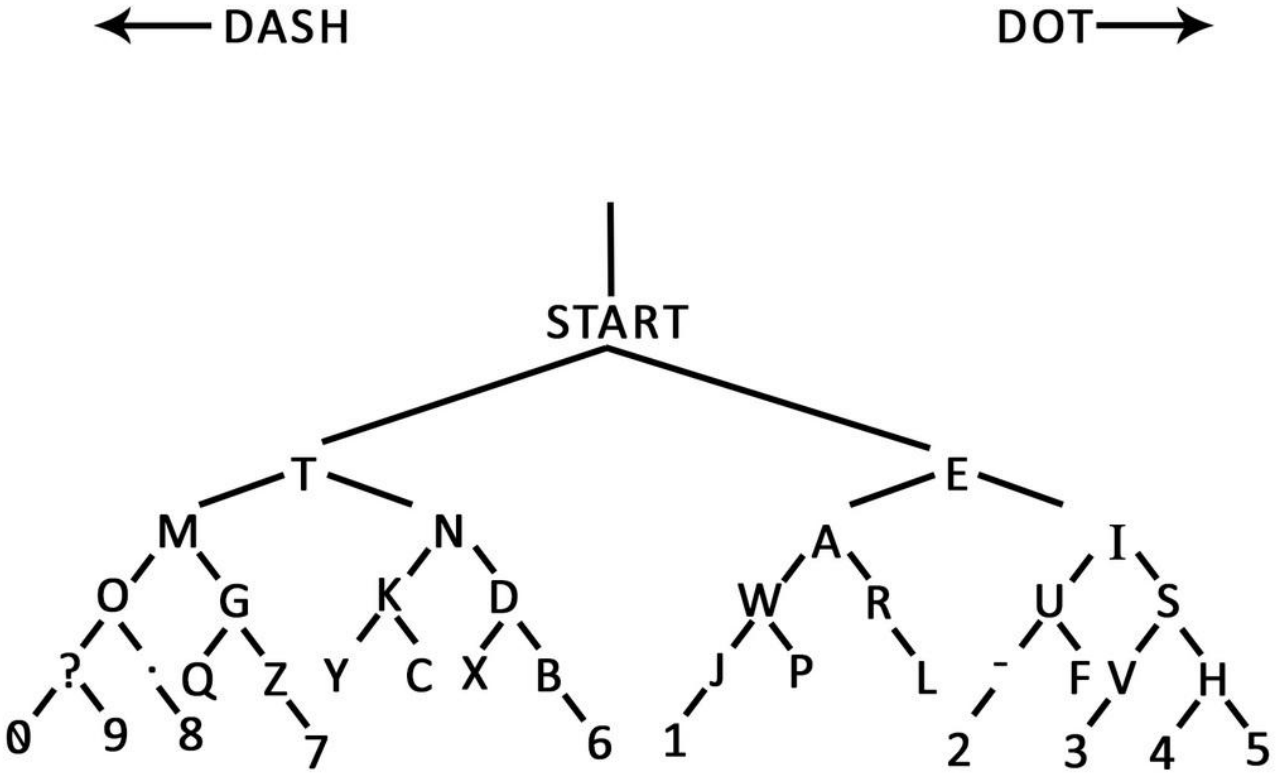
A E T





Hva er 10111?
1_0_1_1_1 = Tettt ?
10_1_11 = NTM? Norsk teknisk museum?

1_01_1_1 = Tatt?


DASH
DOT
START
T
E
M
N
A
I
O
G
K
D
W
R
U
S
?
.
Q
Z
Y
C
X
B
J
P
L
-
F
V
H
0
9
8
7
6
1
2
3
4
5

Ascii	Dec	Binary
A	65	0100 0001
B	66	0100 0010
C	67	0100 0011
D	68	0100 0100
E	69	0100 0101
F	70	0100 0110
G	71	0100 0111
H	72	0100 1000
I	73	0100 1001
J	74	0100 1010
K	75	0100 1011
L	76	0100 1100
M	77	0100 1101
N	78	0100 1110
O	79	0100 1111
P	80	0101 0000
Q	81	0101 0001

Data-løsningen

8 bits per bokstav, ingen misforståelser!

Men uhmm, meldingene blir kanskje litt lange?

HALLO = 0000 01 0100 0100 111 med morse

01001000 01000001 01001100 01001100 01001111 med bytes

Huffman-trær

Problem: Vi vil representere så mange strenger/symboler som mulig med minst mulig plass

Løsning: Et huffman tre! Et morsekode-aktig tre som sikrer at symbolene aldri er tvetydig og at de mest brukte symbolene er de korteste

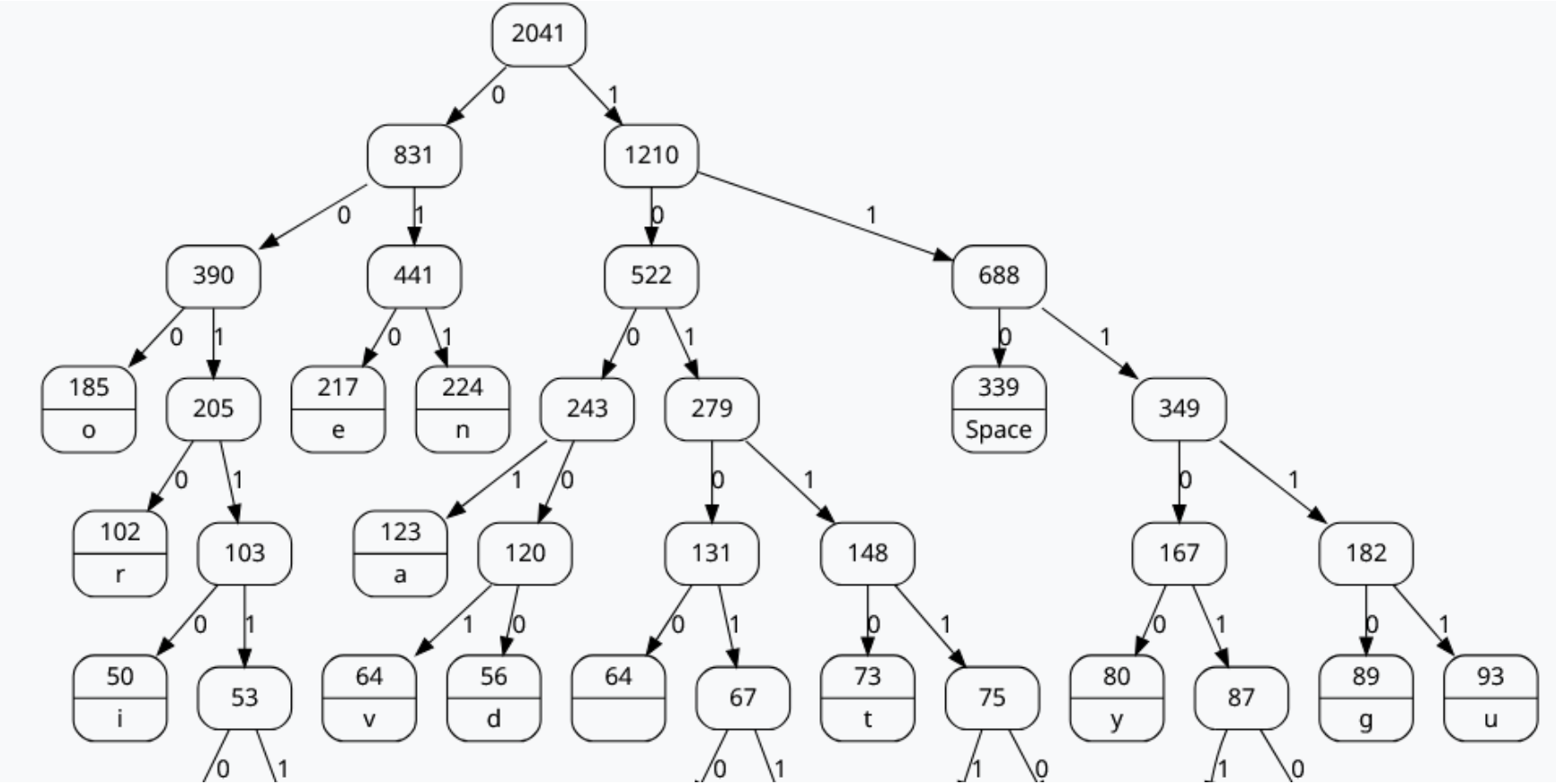
Vi lager en frekvenstabell over alle bokstavene som dukker opp i strengen vi vil representere

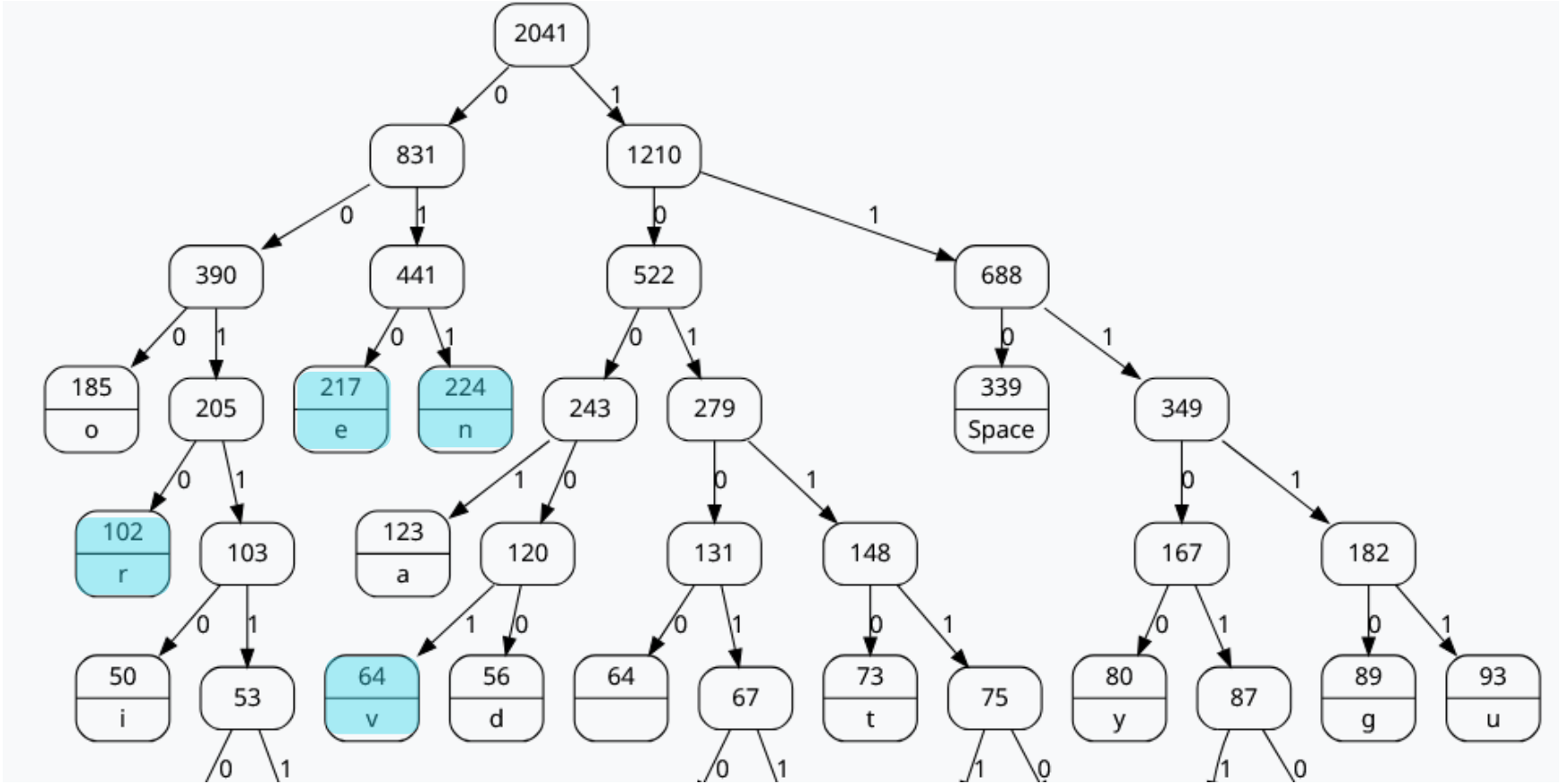
- Da er bokstavene vi bruker mest representert mest færrest mulig bits

We're no strangers to love
You know the rules and so do I
A full commitment's what I'm thinking of
You wouldn't get this from any other guy
I just wanna tell you how I'm feeling
Gotta make you understand
Never gonna give you up
Never gonna let you down
Never gonna run around and desert you
Never gonna make you cry
Never gonna say goodbye
Never gonna tell a lie and hurt you
We've known each other for so long
Your heart's been aching, but you're too shy to say it
Inside, we both know what's been going on
We know the game, and we're gonna play it
And if you ask me how I'm feeling
Don't tell me you're too blind to see
Never gonna give you up
Never gonna let you down
Never gonna run around and desert you
Never gonna make you cry
Never gonna say goodbye
Never gonna tell a lie and hurt you
Never gonna give you up
Never gonna let you down

Never gonna run around and desert you
Never gonna make you cry
Never gonna say goodbye
Never gonna tell a lie and hurt you
Ooh (Give you up)
Ooh-ooh (Give you up)
Ooh (Never gonna give, never gonna give)
Give you up
Ooh-ooh (Never gonna give, never gonna give)
Give you up
We've known each other for so long
Your heart's been aching, but you're too shy to say it
Inside, we both know what's been going on
We know the game, and we're gonna play it
I just wanna tell you how I'm feeling
Gotta make you understand
Never gonna give you up
Never gonna let you down
Never gonna run around and desert you
Never gonna make you cry
Never gonna say goodbye
Never gonna tell a lie and hurt you
Never gonna give you up
Never gonna let you down
Never gonna run around and desert you
Never gonna make you cry

Never gonna say goodbye
Never gonna tell a lie and hurt you
Never gonna give you up
Never gonna let you down
Never gonna run around and desert you
Never gonna make you cry
Never gonna say goodbye
Never gonna tell a lie and hurt you
Never gonna give you up
Never gonna let you down
Never gonna run around and desert you
Never gonna make you cry
Never gonna say goodbye
Never gonna tell a lie and hurt you


2041
831
1210
390
441
522
688
185
o
205
217
e
224
n
243
279
339
Space
349
102
r
103
123
a
120
131
148
167
182
50
i
53
64
v
56
d
64
67
73
t
75
80
y
87
89
g
93
u


2041
0
1
831
1210
0
1
0
1
390
441
522
688
0
1
0
1
0
1
185
o
205
217
e
224
n
243
279
339
Space
349
0
1
0
1
0
1
0
1
102
r
103
123
a
120
131
148
167
182
0
1
0
1
0
1
0
1
0
1
50
i
53
64
v
56
d
64
67
73
t
75
80
y
87
89
g
93
u

Hvordan lager vi et huffman-tre?

1) Vi lager en frekvenstabell med alle symbolene i teksten vi vil encode
2) Setter inn alle symbolene våre i en min-heap
3) Fjerner de to minste elementene fra heapen vår, setter dem inn i et tre som noder
4) Kobler dem sammen i treet, med en felles foreldrenode, og setter foreldrenoden tilbake inn i min-heapen (med den sammenslåtte frekvensen til de to nodene)

Huffman

«HELLO WORLD»



Huffman

L | 3

O | 2

H | 1

E | 1

W | 1

D | 1

R | 1

Huffman

L | 3

O | 2

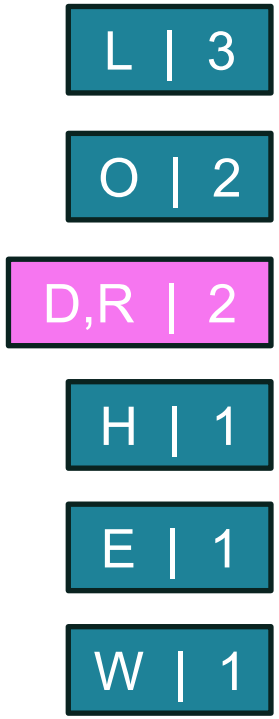
H | 1

E | 1

W | 1

D,R | 2

Huffman



Huffman

L | 3

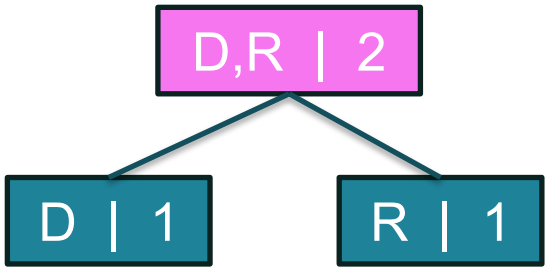
O | 2

D,R | 2

H | 1

E | 1

W | 1



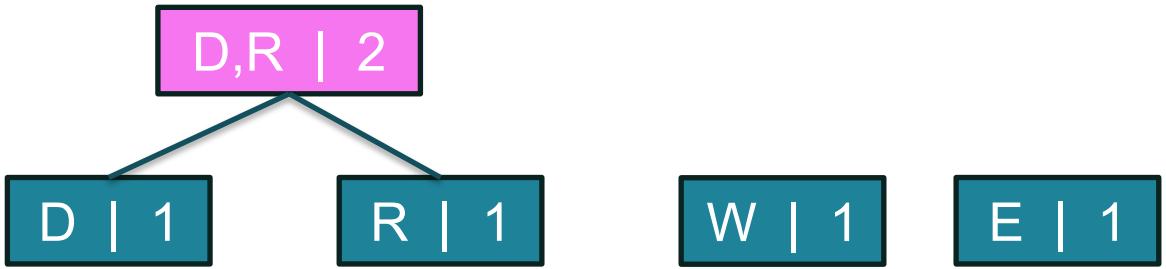
Huffman

L | 3

O | 2

D,R | 2

H | 1



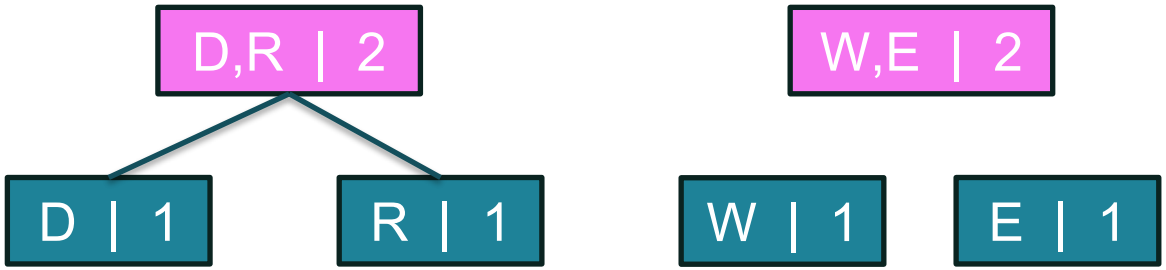
Huffman

L | 3

O | 2

D,R | 2

H | 1



Huffman

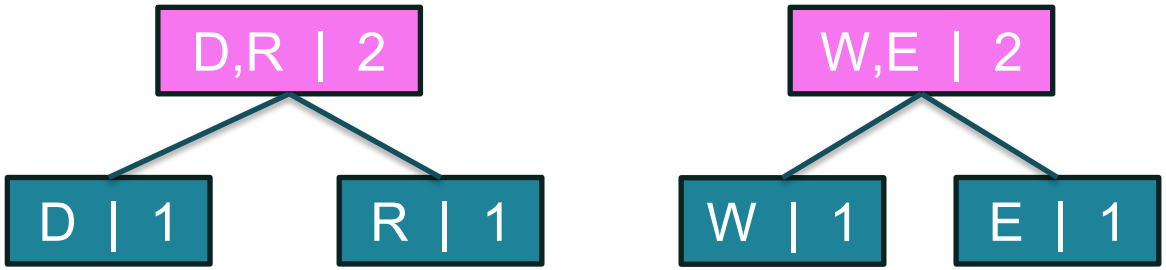
L | 3

O | 2

D,R | 2

W,E | 2

H | 1

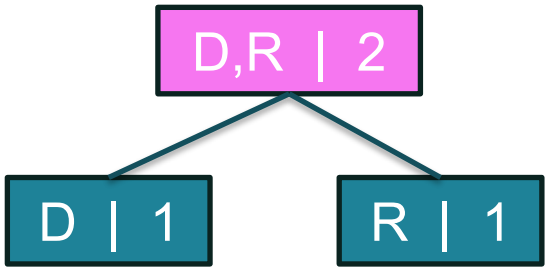


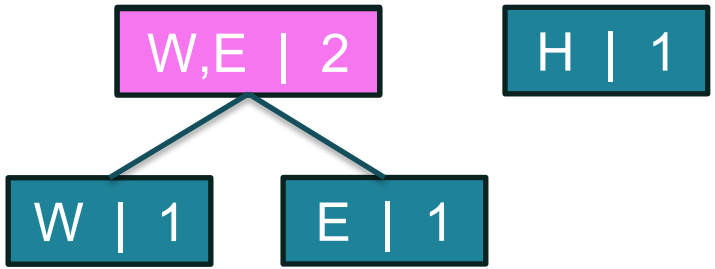
Huffman

L | 3

O | 2

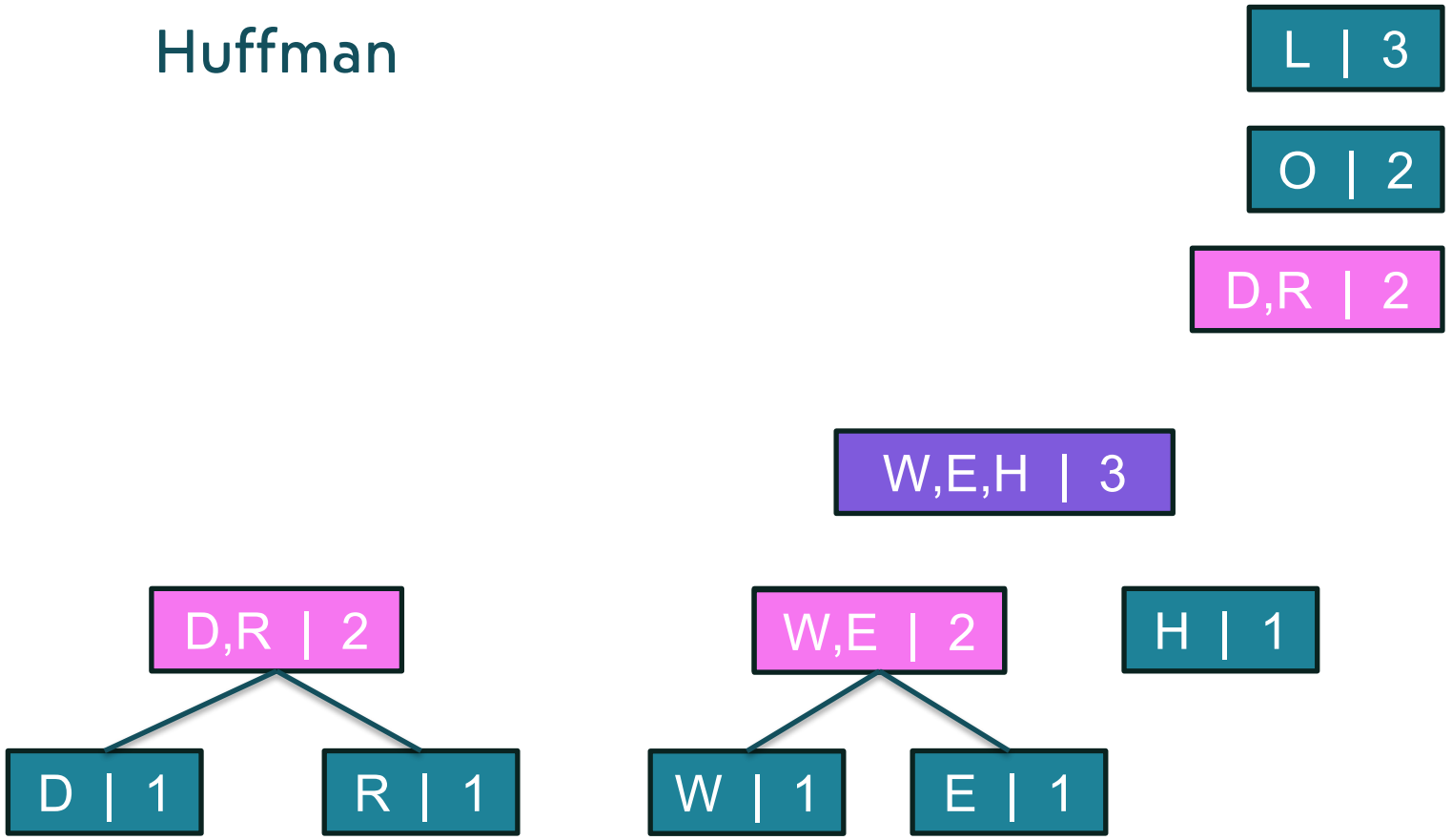
D,R | 2



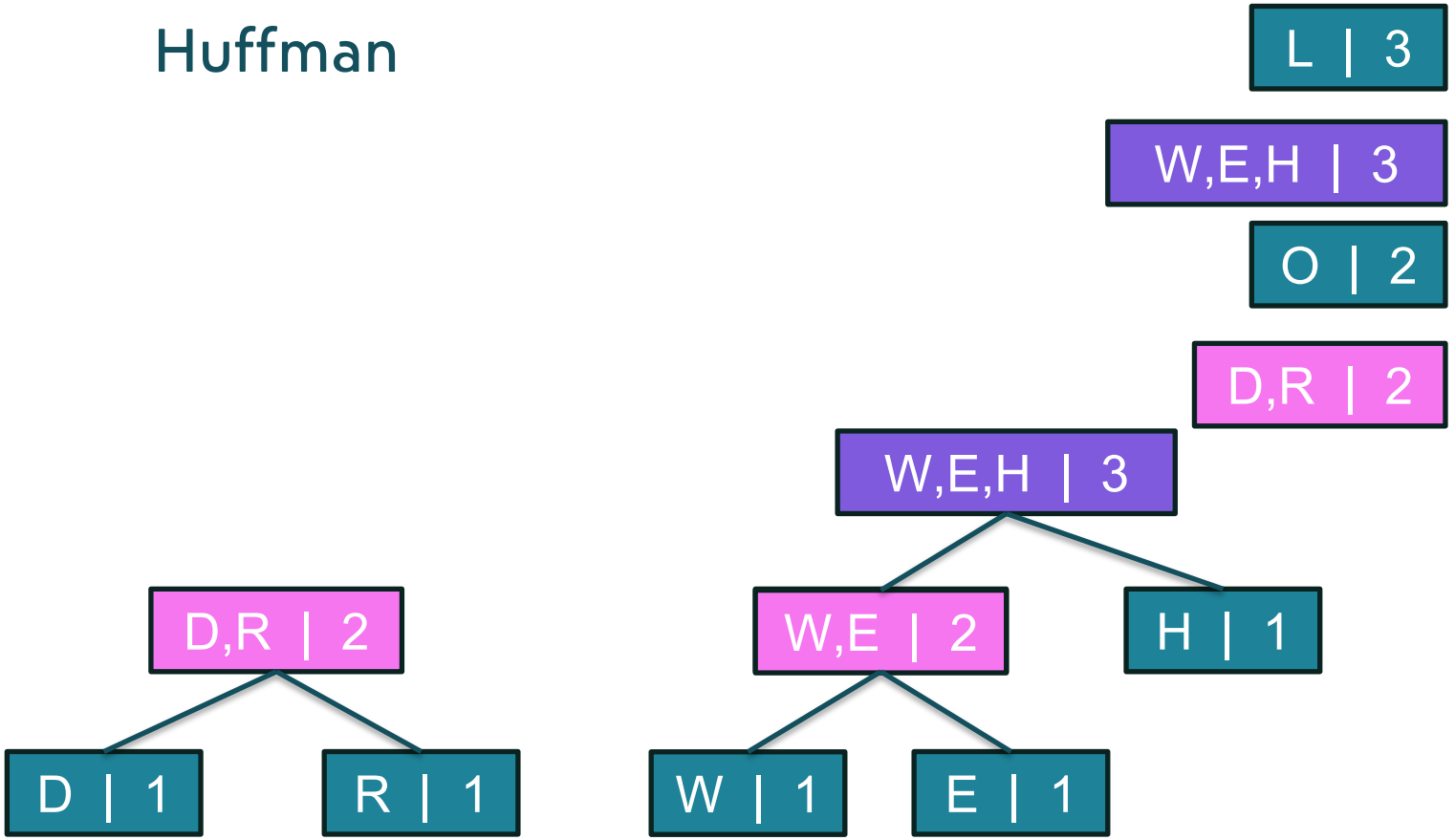


H | 1

Huffman



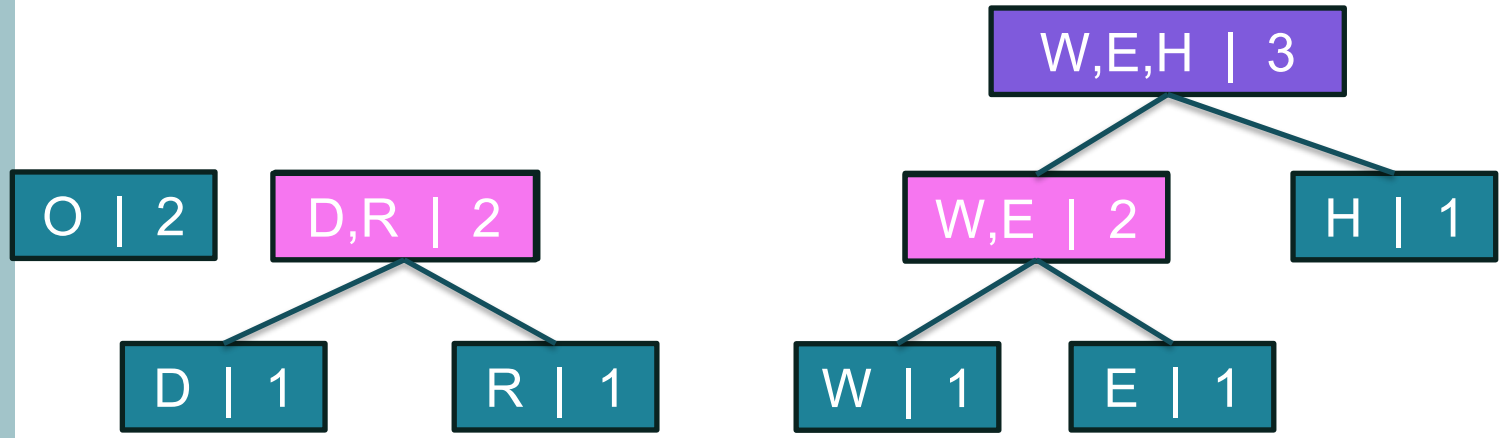
Huffman



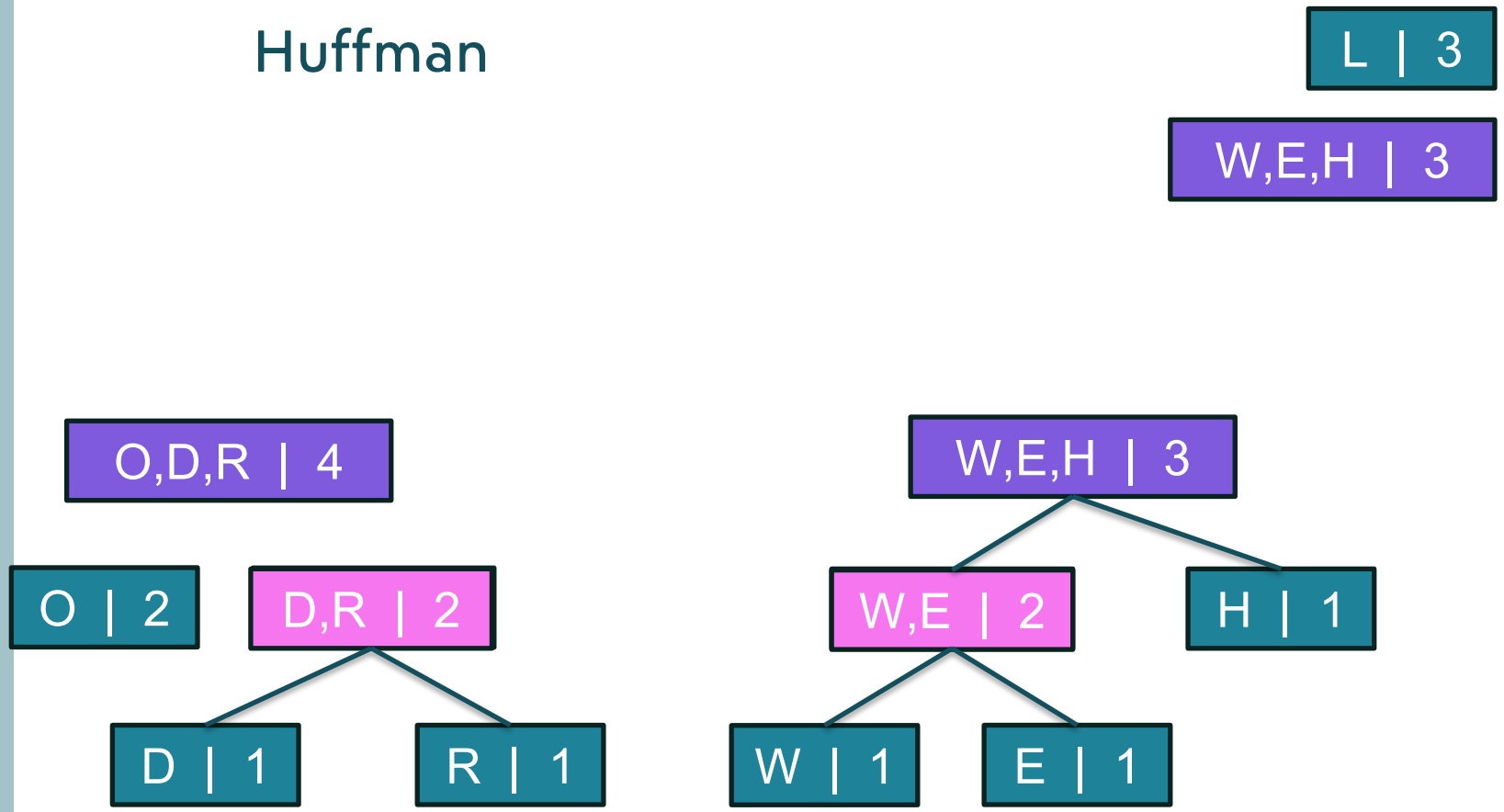
Huffman

L | 3

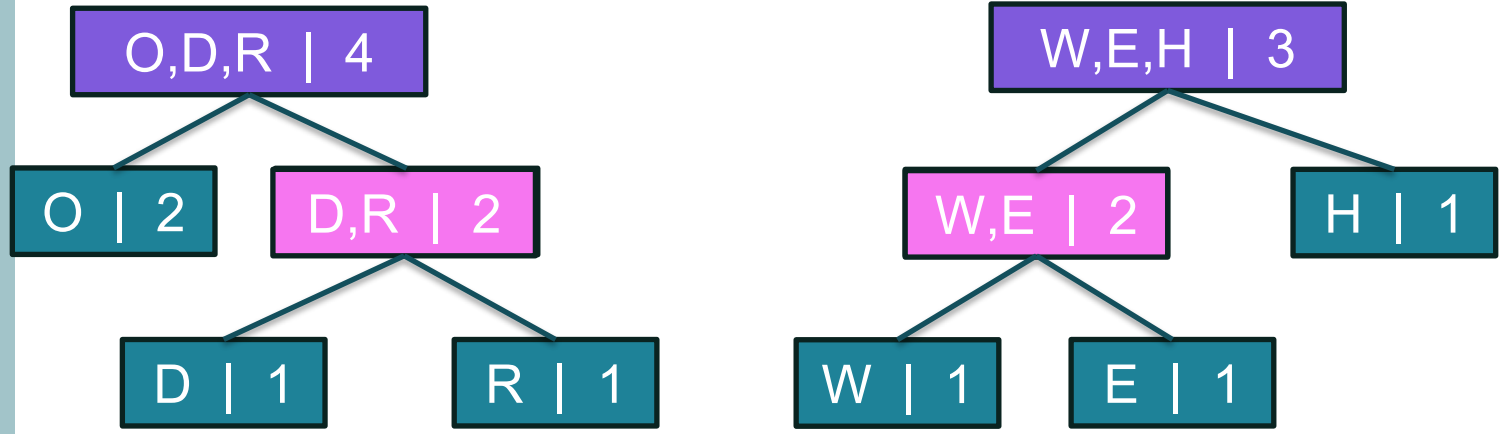
W,E,H | 3



Huffman



Huffman



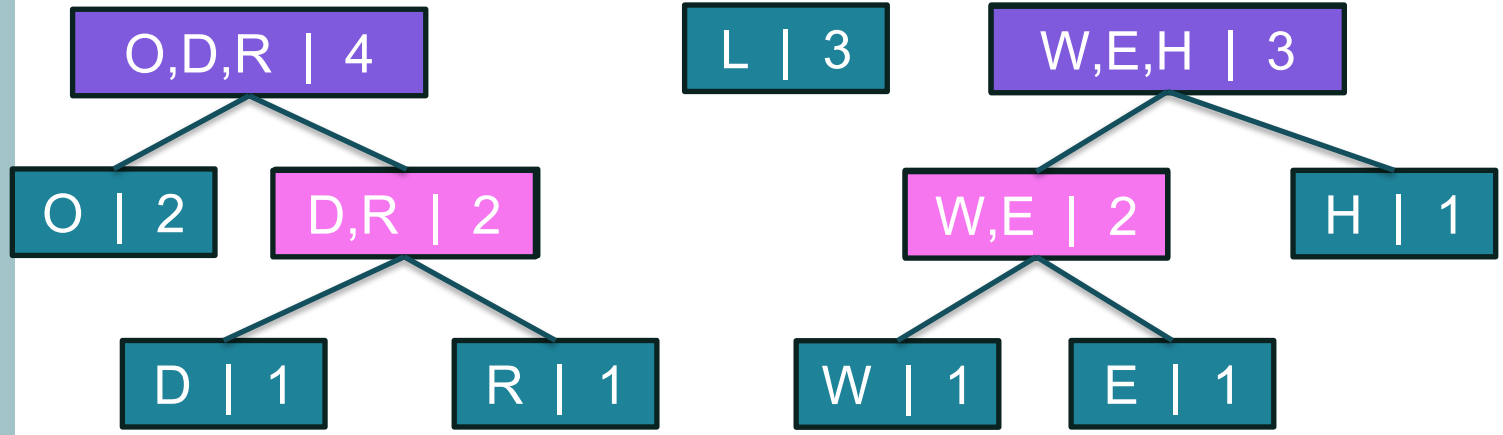
O,D,R | 4

L | 3

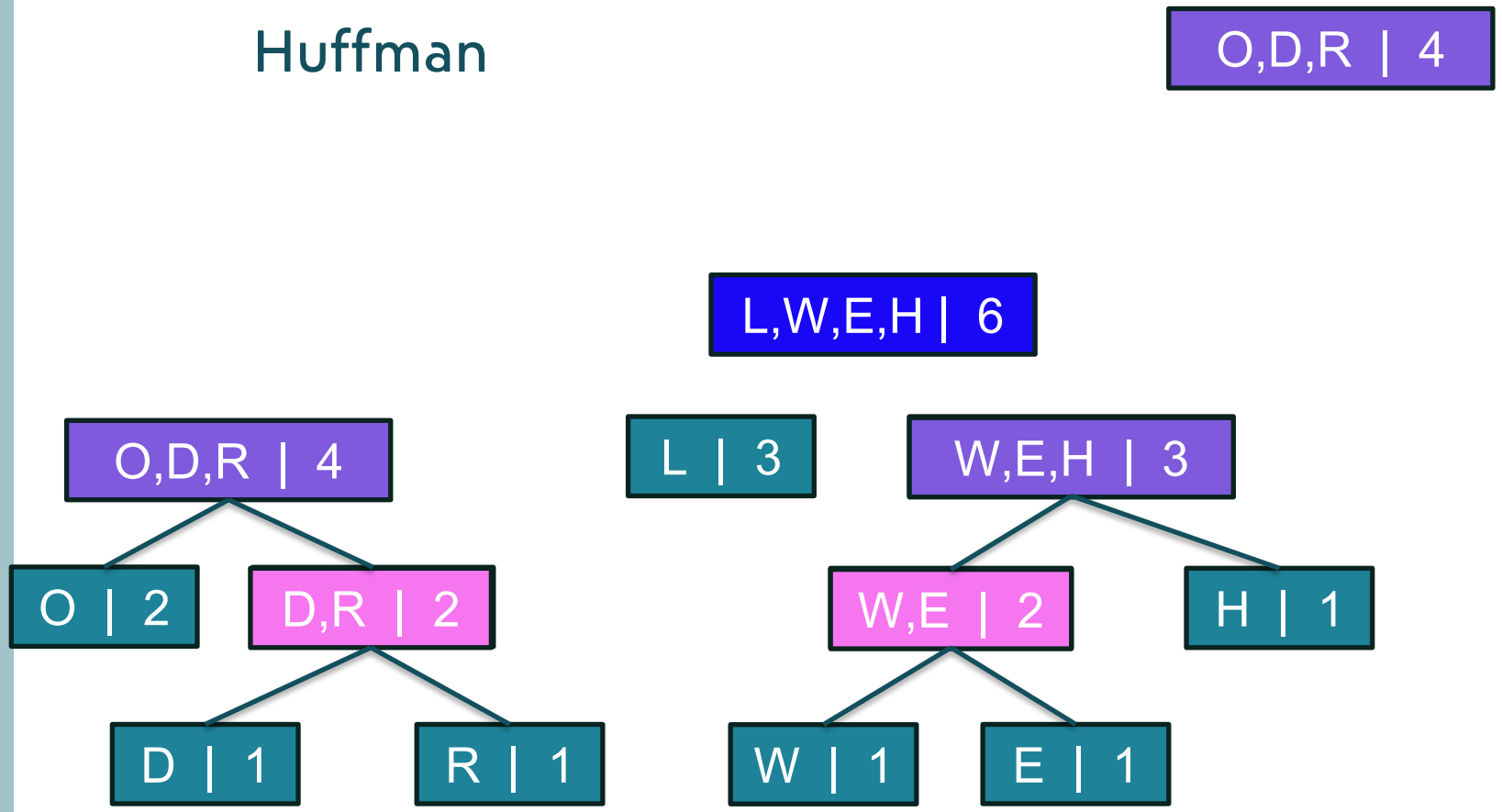
W,E,H | 3

Huffman

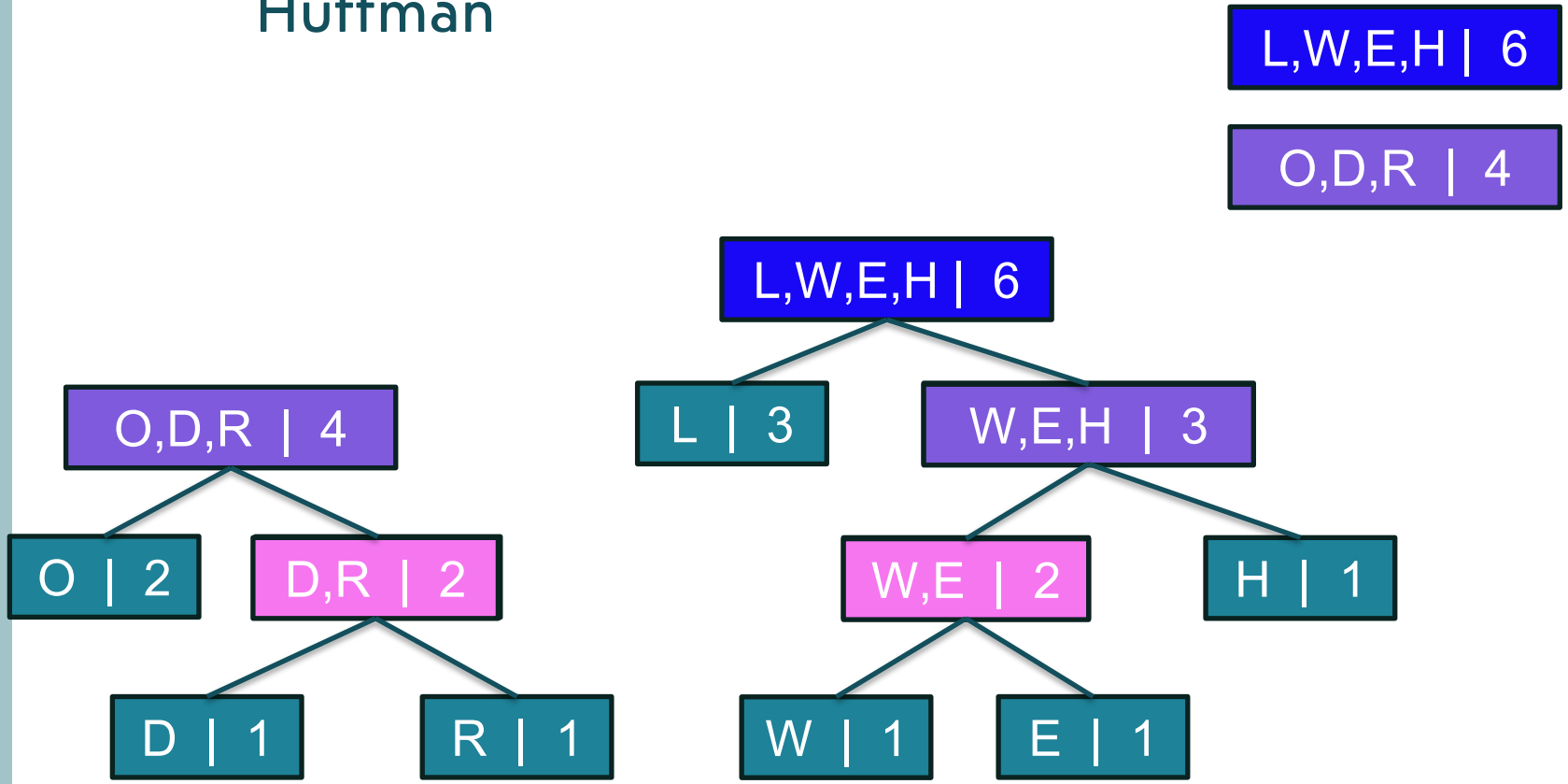
O,D,R | 4



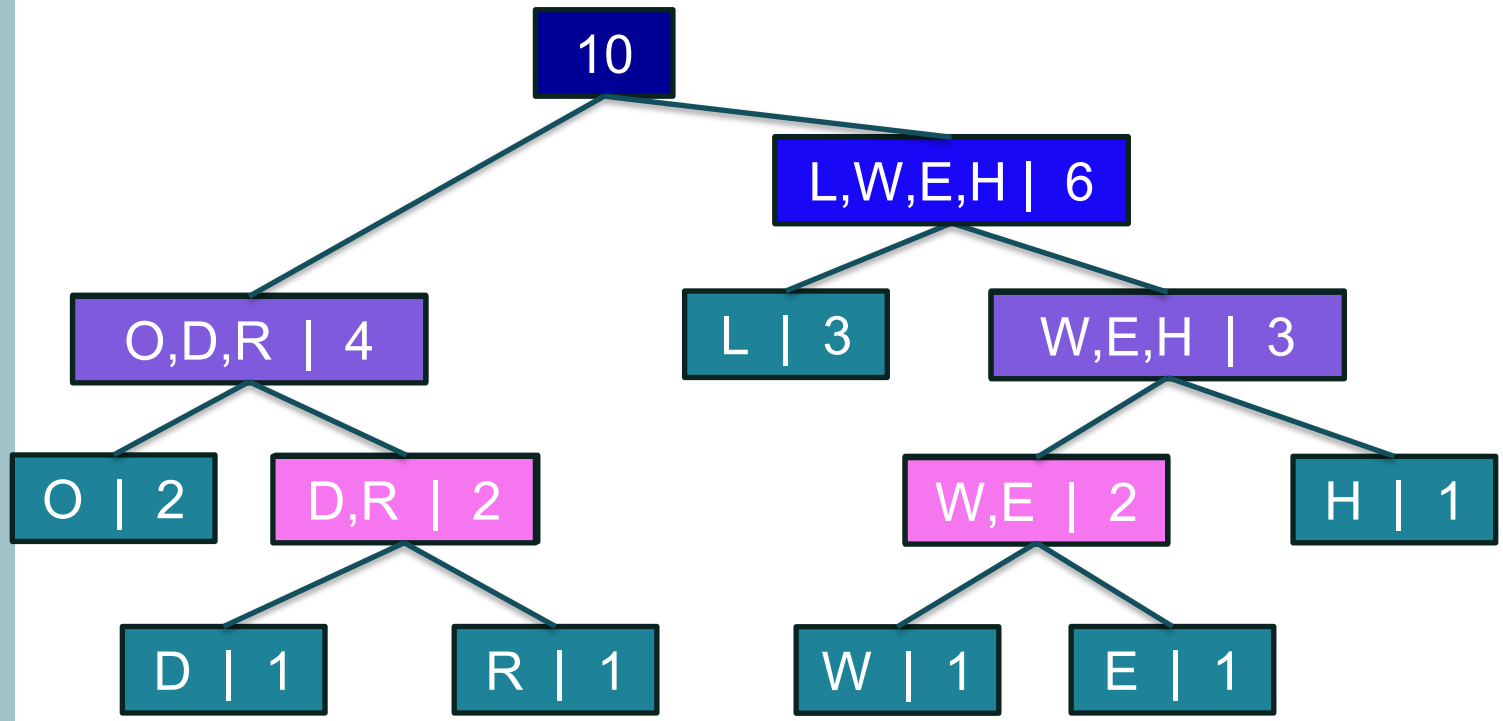
Huffman



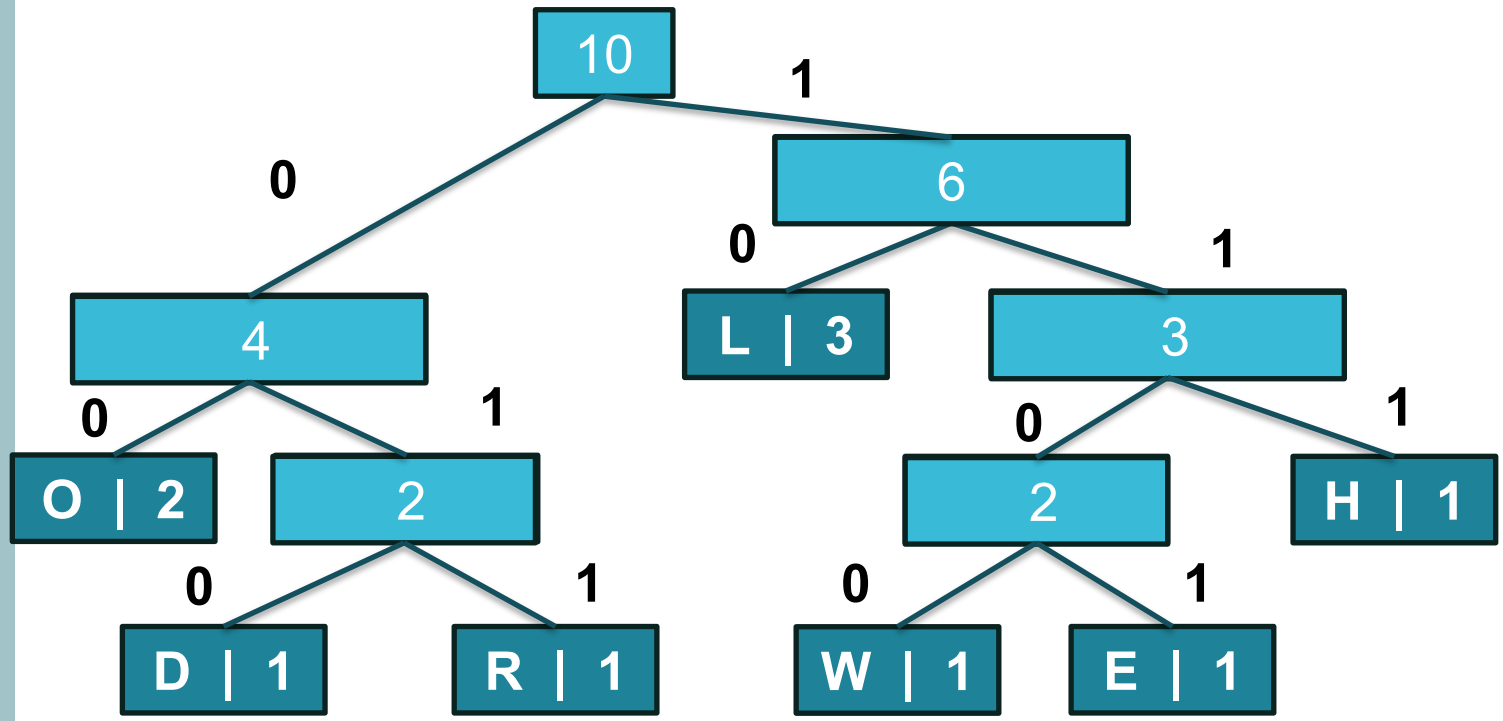
Huffman



Huffman

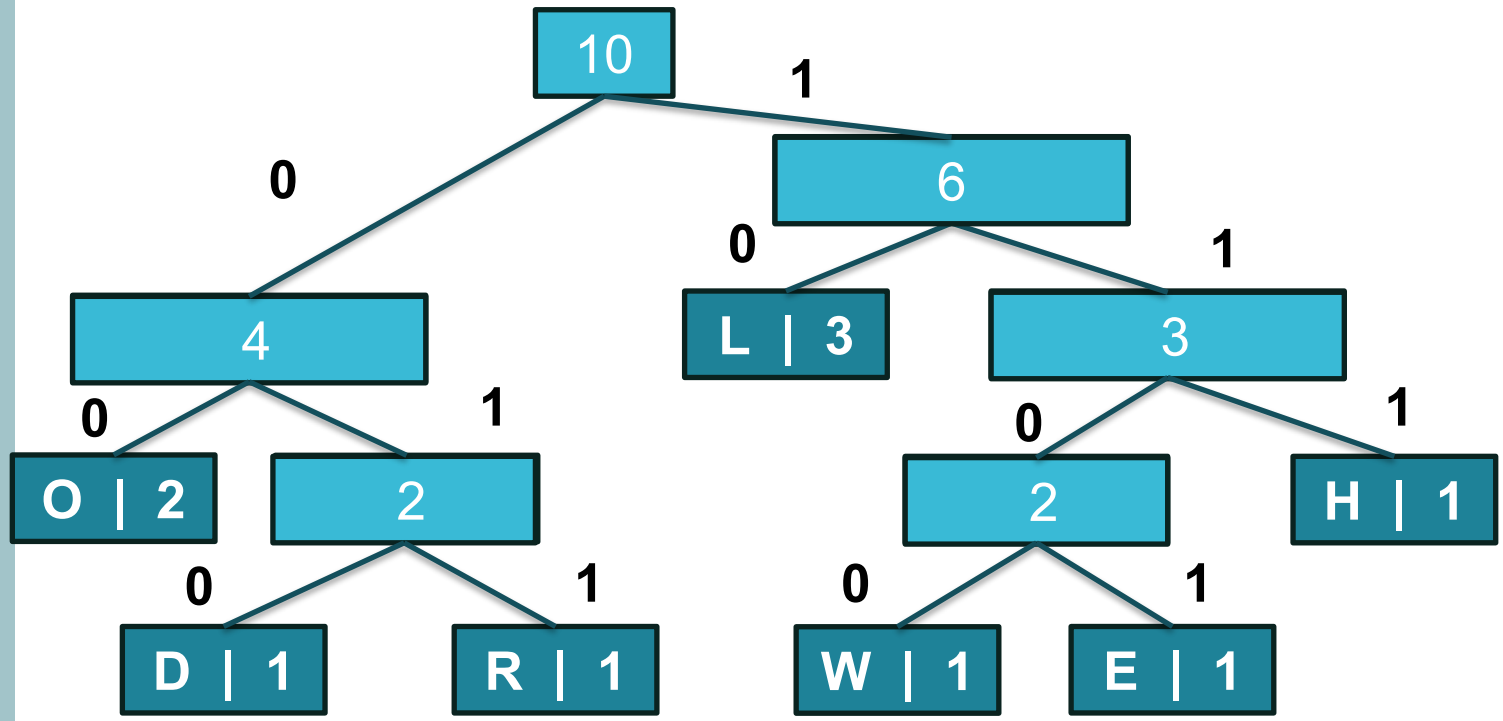


Huffman



Huffman

Hva betyr: 111 1101 10 10 00



Huffman-trær

10 poeng

Vi vil komprimere tekststrenger som består av symbolene A, B, C og D ved hjelp av Huffman-koding. De relative frekvensene er gitt av følgende frekvenstabell:

A	B	C	D
1	3	7	2

(a) Hva blir kodelengdene for de ulike symbolene i det tilhørende Huffman-treet?

(b) Hva inneholder løvnodene i det resulterende treet?

Generelt i et Huffman-tre har hver node v:

- v.`symbol` – Symbolet til noden (eller **`null`**)
- v.`freq` – Frekvensen til noden
- v.`left` – Venstre barn av noden (eller **`null`**)
- v.`right` – Høyre barn av noden (eller **`null`**)

Et Huffman-tre bygges fra en mengde med par (s, f) der s er et symbol og f er en frekvens.

(c) Beskriv algoritmen for å bygge et Huffman-tre. Du kan skissere den med pseudokode, men en god beskrivelse med naturlig språk gir like god uttelling.