Innhold

Notasjon Uttrykk

O(1) Konstant tid

O(log(n)) Logaritmisk tid

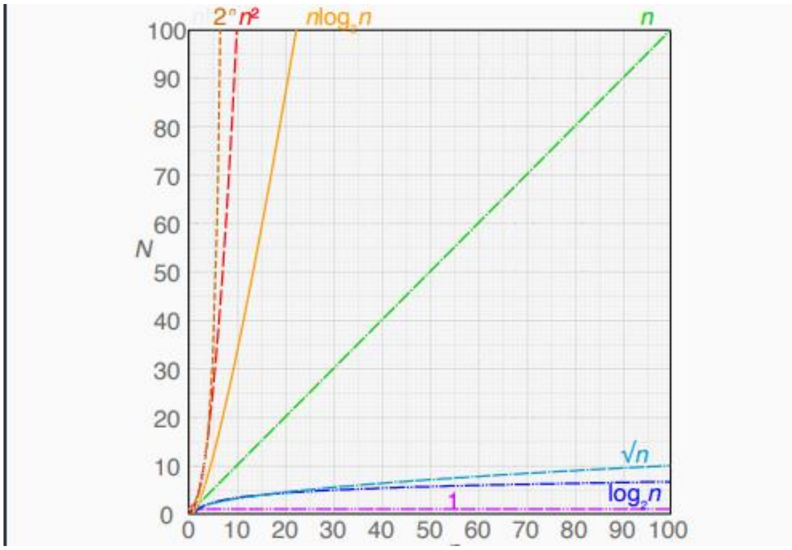
O(n) Lineær tid

O(n · log(n)) Lineæritmisk tid

$O(n^2)$ kvadratisk tid

$O(n^k)$ polynomiell tid

$O(2^n)$ eksponensiell tid



Introduksjon

Kodeeksempler

```
Procedure Constant(n)
    return n · 3                          // O(1)
```

```
Procedure Log(n)
    i ← n
    while i > 0 do
        Constant(i)
        i ← ⌊i/2⌋                         // O(log(n))
```

```
Procedure Linear(n)
    for i ← 0 to n − 1 do
        Constant(i)                       // O(n)
```

```
Procedure Linearithmic(n)
    for i ← 0 to n − 1 do
        Log(n)                            // O(n · log(n))
```

```
Procedure Quadratic(n)
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            Constant(i)                   // O(n^2)
```

```
Procedure Polynomial(n)
    for i_1 ← 0 to n − 1 do
        for i_2 ← 0 to n − 1 do
            ⋮
                for i_k ← 0 to n − 1 do
                    Constant(i)           // O(n^k)
```

```
Procedure Exponential(n)
    if n = 0 then
        return 1
    a ← Exponential(n − 1)
    b ← Exponential(n − 1)
    return a + b                          // O(2^n)
```

Rett frem søk:

O(n) Lineær tid

```
ALGORITHM: RETT FREM SØK
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
    return false
```

Binærsøk:

O(log(n)) Logaritmisk tid

```
ALGORITHM: 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 ← 0
    high ← |A| − 1
    while low ≤ high do
        i ← ⌊(low+high)/2⌋
        if A[i] = x then
            return true
        else if A[i] < x then
            low ← i + 1
        else if A[i] > x then
            high ← i − 1
    return false
```

Trær

Dybde:

O(log(n)) Logaritmisk tid

Ikke-balansert O(n) Lineær tid

```
ALGORITHM: 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)
```

Høyde:

O(n)

ALGORITHM: FINN HØYDEN AV EN GITT NODE

Input: En node v

Output: Høyden av noden

```
Procedure height(v)
    h ← −1
    if v = null then
        return h
    for c ∈ v.children do
        h ← Max(h, height(c))
    return 1 + h
```

Binæretrær

Innsetting I binærtrær:

Balansert O(log(n)) Logaritmisk tid

Ikke-balansert O(n) Lineær tid

ALGORITHM: 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 $\mathcal{O}(h)$
  - der h er høyden på treet
- Dersom n er antall noder i treet har vi $\mathcal{O}(n)$ i verste tilfelle
  - men hvis treet er balansert, så er kompleksiteten $\mathcal{O}(log(n))$

Opplsag:

Balansert O(log(n)) Logaritmisk tid

Ikke-balansert O(n) Lineær tid

ALGORITHM: 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)
```

Sletting:

Balansert O(log(n)) Logaritmisk tid

Ikke-balansert O(n) Lineær tid

Sletting

ALGORITHM: 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
```

AVL-trær

Balansering:

O(log(n)) Logaritmisk tid

ALGORITHM: BALANSERING AV ET AVL-TRE

Input: En node v

Output: En balansert node

```
Procedure Balance(v)
    if BalanceFactor(v) < −1 then
        if BalanceFactor(v.right) > 0 then
            v.right ← RightRotate(v.right)
        return LeftRotate(v)
    if BalanceFactor(v) > 1 then
        if BalanceFactor(v.left) < 0 then
            v.left ← LeftRotate(v.left)
        return RightRotate(v)
    return v
```

Innsetting:

O(log(n)) Logaritmisk tid

ALGORITHM: INNSETTING I ET AVL-TRE

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)
    SetHeight(v)
    return Balance(v)
```

Sletting:

O(log(n)) Logaritmisk tid

Sletting

ALGORITHM: 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
```

Binæreheaps

Insetting:

O(log(n)) Logaritmisk tid

Binære heaps – innsetting (implementasjon)

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

```
Procedure Insert(A, x)
    A[n] ← x
    i ← n
    while 0 < i and A[i] < A[ParentOf(i)] do
        A[i], A[ParentOf(i)] ← A[ParentOf(i)], A[i]
        i ← ParentOf(i)
```

Sletting:

O(log(n)) Logaritmisk tid

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

```
Procedure RemoveMin(A)
    x ← A[0]
    A[0] ← A[n - 1]
    i ← 0
    while LeftOf(i) < n − 1 do
        j ← LeftOf(i)
        if RightOf(i) < n − 1 and A[RightOf(i)] < A[j] then
            j ← RightOf(i)
        if A[i] ≤ A[j] then
            return x
        A[i], A[j] ← A[j], A[i]
        i ← j
    return x
```

Sorteringer

Bubble sort:

$O(n^2)$ kvadratisk tid

```
ALGORITHM: BUBBLE SORT
Input: Et array A med n elementer
Output: Et sortert array med de samme n elementene
Procedure BubbleSort(A)
    for i ← 0 to n − 2 do
        for j ← 0 to n − i − 2 do
            if A[j] > A[j + 1] then
                A[j], A[j + 1] ← A[j + 1], A[j]
```

Selection sort:

$O(n^2)$ kvadratisk tid

```
ALGORITHM: SELECTION SORT
Input: Et array A med n elementer
Output: Et sortert array med de samme n elementene
Procedure SelectionSort(A)
    for i ← 0 to n − 1 do
        k ← i
        for j ← i + 1 to n − 1 do
            if A[j] < A[k] then
                k ← j
        if i ≠ k then
            A[i], A[k] ← A[k], A[i]
```

Insertion sort:

$O(n^2)$ kvadratisk tid

```
ALGORITHM: INSERTION SORT
Input: Et array A med n elementer
Output: Et sortert array med de samme n elementene
Procedure InsertionSort(A)
    for i ← 1 to n − 1 do
        j ← i
        while j > 0 and A[j-1] > A[j] do
            A[j-1], A[j] ← A[j], A[j-1]
            j ← j − 1
```

Merge sort:

O(n · log(n)) Lineæritmisk tid

ALGORITHM: SORTERT FLETTING AV TO ARRAYER

Input: To sorterte arrayer A_1 og A_2 og et array A, der $|A_1| + |A_2| = |A| = n$

Output: Et sortert array A med elementene fra A_1 og A_2

```
Procedure Merge(A₁, A₂, A)
    i ← 0
    j ← 0
    while i < |A₁|  and  j < |A₂| do
        if A₁[i] ≤ A₂[j] then
            A[i + j] ← A₁[i]
            i ← i + 1
        else
            A[i + j] ← A₂[j]
            j ← j + 1
    while i < |A₁| do
        A[i + j] ← A₁[i]
        i ← i + 1
    while j < |A₂| do
        A[i + j] ← A₂[j]
        j ← j + 1
    return A
```

ALGORITHM: MERGE SORT

Input: Et array A med n elementer

Output: Et *sortert* array med de samme n elementene

```
Procedure MergeSort(A)
    if n ≤ 1 then
        return A
    i ← ⌊n/2⌋
    A₁ ← MergeSort(A[0..i-1])
    A₂ ← MergeSort(A[i..n-1])
    return Merge(A₁, A₂, A)
```

Heapsort:

O(n · log(n)) Lineæritmisk tid

ALGORITHM: HJELPEPROSEDYRE FOR Å BYGGE EN MAX-HEAP

Input: En (uferdig) heap A med n elementer der i er roten
Output: En mindre uferdig heap

```
Procedure BubbleDown(A, i, n)
    largest ← i
    left ← 2i + 1
    right ← 2i + 2

    if left < n and A[largest] < A[left] then
        largest, left ← left, largest

    if right < n and A[largest] < A[right] then
        largest, right ← right, largest

    if i ≠ largest then
        A[i],A[largest] ← A[largest],A[i]
        BubbleDown(A, largest, n)
```

ALGORITHM: BYGG EN MAX-HEAP

Input: Et array A med n elementer
Output: A som en max-heap

```
Procedure BuildMaxHeap(A, n)
    for i ← ⌊n/2⌋ down to 0 do
        BubbleDown(A, i, n)
```

Algorithm: Hjelpeprosedyre for å bygge en max-heap

```
Procedure BubbleDown(A, i, n)
    largest ← i
    left ← 2i + 1
    right ← 2i + 2

    if left < n and A[largest] < A[left] then
        largest, left ← left, largest

    if right < n and A[largest] < A[right] then
        largest, right ← right, largest

    if i ≠ largest then
        A[i], A[largest] ← A[largest], A[i]
        BubbleDown(A, largest, n)
```

ALGORITHM: HEAPSORT

Input: Et array A med n elementer
Output: Et *sortert* array med de samme n elementene

```
Procedure HeapSort(A)
    BuildMaxHeap(A, n)
    for i ← n − 1 down to 0 do
        A[0],A[i] ← A[i],A[0]
        BubbleDown(A, 0, i)
```

Quicksort:

Verste tilfellet: $O(n^2)$ kvadratisk tid

ALGORITHM: PARTITION

Input: Et array A med n elementer, low og high er indekser

Output: Flytter elementer som er hhv. mindre og større til venstre og høyre enn en gitt index som returneres

```
Procedure Partition(A, low, high)
    p ← ChoosePivot(A, low, high)
    A[p], A[high] ← A[high], A[p]

    pivot ← A[high]
    left ← low
    right ← high − 1

    while left ≤ right do
        while left ≤ right and A[left] ≤ pivot do
            left ← left + 1

        while right ≥ left and A[right] ≥ pivot do
            right ← right − 1

        if left < right then
            A[left], A[right] ← A[right], A[left]

    A[left], A[high] ← A[high], A[left]
    return left
```

ALGORITHM: QUICKSORT

Input: Et array A med n elementer, low og high er indekser

Output: Et *sortert* array med de samme n elementene

```
Procedure Quicksort(A, low, high)
    if low ≥ high then
        return A
    p ← Partition(A, low, high)
    Quicksort(A, low, p - 1)
    Quicksort(A, p + 1, high)
    return A
```

- Vi kaller på Quicksort(A, 0, $n-1$) for å sortere hele arrayet

Bucketsort:

$O(n)$ Lineær tid

```
ALGORITHM: BUCKET SORT
Input: Et array A med n elementer
Output: Et array med de samme n elementene sortert etter nøkler
Procedure BucketSort(A)
    La B være et array med N tomme lister
    for i ← 0 to n − 1 do
        La k være nøkkelen assosiert med A[i]
        Legg til A[i] på slutten av listen B[k]
    j ← 0
    for k ← 0 to N − 1 do
        for hver x i listen B[k] do
            A[j] ← x
            j ← j + 1
    return A
```

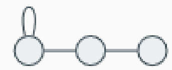
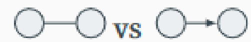
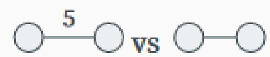
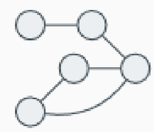
Radix-sort:

O(n) Lineær tid

```
ALGORITHM: RADIX SORT FOR POSITIVE HELTALL
Input: Et array A med n positive heltall
Output: Et sortert array med de samme n positive heltallene
Procedure RadixSort(A)
    d ← antall siffer i det største tallet
    for i ← d − 1 down to 0 do
        A ← BucketSort(A) etter det ite sifferet
    return A
```

Grafer

Terminologi

Betegnelse	Forklaring	Eksempel
Parellelle kanter	Flere enn én kant mellom to noder	
(Enkle) løkker	En kant fra en node til seg selv	
Urettet/rettet	Kantene i grafen har retning	
Vektet/uvektet	Kantene har en verdi	
Enkel graf	Uten løkker, parallelle kanter, retning og vekt	

4

DFS:

O(|V| + |E|)

ALGORITHM: ITERATIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode s og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
Procedure DFSVisit(G, s, visited)
    stack ← stack containing s
    while stack is not empty do
        u ← stack.pop()
        if u ∉ visited then
            add u to visited
            for (u, v) ∈ E do
                stack.push(v)
```

Dybde-først søk (rekursiv)

ALGORITHM: REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$, en startnode u og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra u nøyaktig én gang

```
Procedure DFSVisit(G, u, visited)
    add u to visited
    for (u, v) ∈ E do
        if v ∉ visited then
            DFSVisit(G, v, visited)
```

ALGORITHM: FULLT REKURSIVT DYBDE-FØRST SØK

Input: En graf $G = (V, E)$

Output: Besøker alle noder i G nøyaktig én gang dybde først

```
Procedure DFSFull(G)
    visited ← empty set
    for v ∈ V do
        if v ∉ visited then
            DFSVisit(G, v, visited)
```

BFS:

$O(|V| + |E|)$

ALGORITHM: BREDDE-FØRST SØK

Input: En graf $G = (V, E)$ og en startnode s, og en mengde `visited` med besøkte noder

Output: Besøker alle noder i G som kan nås fra s nøyaktig én gang

```
Procedure BFSVisit(G, s, visited)
    add s to visited
    queue ← queue containing s
    while queue is not empty do
        u ← queue.dequeue()
        for (u, v) ∈ E do
            if v ∉ visited then
                add v to visited
                queue.enqueue(v)
```

ALGORITHM: FULLT BREDDE-FØRST SØK

Input: En graf $G = (V, E)$

Output: Besøker alle noder i G nøyaktig én gang bredde først

```
Procedure BFSFull(G)
    visited ← empty set
    for v ∈ V do
        if v ∉ visited then
            BFSVisit(G, v, visited)
```

Topologisk sortering:

O(|V| + |E|)

ALGORITHM: TOPOLOGISK SORTERING VED DFS

Input: En rettet asyklisk graf $G = (V, E)$

Output: En topologisk ordning av nodene i G

```
Procedure DFSTopSort(G)
    stack ← empty stack
    visited ← empty set
    for u ∈ V do
        if u ∉ visited then
            DFSVisit(G, u, visited, stack)
    return stack

Procedure DFSVisit(G, u, visited, stack)
    add u to visited
    for (u, v) ∈ E do
        if v ∉ visited then
            DFSVisit(G, v, visited, stack)
    stack.push(u)
```

Vektede grafer

Dijkstra:

O((|V|+|E|)·log(|V|))

Kan forenkles til O(|E | · log(|V |)) hvis grafen er sammenhengende

ALGORITHM: DIJKSTRAS ALGORITME FOR KORTESTE STIER (TRADISJONELL)

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

```
Procedure Dijkstra(G, s)
    queue ← empty priority queue
    dist ← empty map
    for v ∈ V do
        dist[v] ← ∞
        Insert(queue, v) with priority ∞
    dist[s] ← 0
    DecreasePriority(queue, s, 0)

    while queue is not empty do
        u ← RemoveMin(queue)
        for (u, v) ∈ E do
            c ← dist[u] + w(u, v)
            if c < dist[v] then
                dist[v] ← c
                DecreasePriority(queue, v, c)
    return dist
```

ALGORITHM: DIJKSTRAS ALGORITME FOR KORTESTE STIER

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

```
Procedure Dijkstra(G, s)
    dist ← empty map with ∞ as default
    queue ← priority queue containing s with priority 0
    dist[s] ← 0

    while queue is not empty do
        u ← RemoveMin(queue)
        for (u, v) ∈ E do
            c ← dist[u] + w(u, v)
            if c < dist[v] then
                dist[v] ← c
                Insert(queue, v) with priority c
    return dist
```

Bellman-Ford:

Forenkling $O(|V|^3)$

$O(|V| \cdot |E|)$

ALGORITHM: BELLMAN-FORDS ALGORITME FOR KORTESTE STIER

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

```
Procedure BellmanFord(G, s)
    dist ← empty map with ∞ as default
    dist[s] = 0

    repeat |V| − 1 times
        for (u, v) ∈ E do
            c ← dist[u] + w(u, v)
            if c < dist[v] then
                dist[v] ← c

    for (u, v) ∈ E do
        c ← dist[u] + w(u, v)
        if c < dist[v] then
            error G contains a negative cycle
    return dist
```

Korteste sti:

$O(|V| + |E|)$

ALGORITHM: KORTESTE STIER I EN DAG

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

```
Procedure DAGShortestPaths(G, s)
    dist ← empty map with ∞ as default
    dist[s] = 0

    for u ∈ TopSort(G) do
        for (u, v) ∈ E do
            c ← dist[u] + w(u, v)
            if c < dist[v] then
                dist[v] ← c
    return dist
```

Prims:

$O(|E| \cdot \log(|V|))$

ALGORITHM: PRIMS ALGORITME FOR MINIMALE SPENNTRÆR

Input: En sammenhengende, vektet, urettet graf $G = (V, E)$ med vektfunksjon w
Output: Et minimalt spenntre for G

```
Procedure Prim(G)
    queue ← empty priority queue
    parents ← empty map
    Insert(queue, (null, s)) with priority 0, for some arbitrary s ∈ V
    while queue is not empty do
        (p, u) ← RemoveMin(queue)
        if u ∉ parents then
            parents[u] ← p
            for (u, v) ∈ E do
                Insert(queue, (u, v)) with priority w(u, v)
    return parents
```

Sammenhengende grafer

Naiv 2-sammenhengende

$O(|V|^3)$

ALGORITHM: NAIV ALGORITME FOR Å SJEKKE OM EN GRAF ER 2-SAMMENHENGENDE

Input: En sammenhengende graf $G = (V, E)$
Output: Gir **true** hvis grafen er 2-sammenhengende, **false** ellers

```
Procedure IsBiconnectedNaive(G)
    for v ∈ V do
        G′ = (V′, E′) ← G with v removed
        visited ← empty set
        u ← any vertex u ∈ V′
        DFSVisit(G′, u, visited)
        if visited ≠ V′ then
            return false
    return true
```

Finne seprasjonsnoder:

O(|V | + |E|)

ALGORITHM: FINN ALLE SEPARASJONSNODER I EN SAMMENHENGENDE GRAF

```
Input: En graf sammenhengende G = (V, E)
Output: Returnerer alle separasjonsnoder i G
depth ← empty map
low ← empty map
seps ← empty set
Procedure SeparationVertices(G)
    s ← choose arbitrary vertex from V
    depth[s] ← 0
    low[s] ← 0
    children ← 0

    for (s, u) ∈ E do
        if u ∉ depth then
            SepRec(G, s, u, 1)
            children ← children + 1

    if children > 1 then
        add s to seps
    return seps

Procedure SepRec(G, p, u, d)
    depth[u] ← d
    low[u] ← d

    for (u, v) ∈ E do
        if v = p then
            continue
        if v ∈ depth then
            low[u] ← min(low[u], depth[v])
            continue

        SepRec(G, u, v, d + 1)
        low[u] ← min(low[u], low[v])
        if d ≤ low[v] then
            add u to seps
```

Sterkt sammenhengende komponenter:

O(|V | + |E|)

ALGORITHM: FINN DE STERKT SAMMENHENGENDE KOMPONENETENE AV EN GRAF

```
Input: En rettet graf G = (V, E)
Output: Returner de sterkt sammenhengende komponenetene til G
Procedure StronglyConnectedComponents(G)
    stack ← DFSTopSort(G)
    G_r ← ReverseGraph(G)
    visited ← empty set
    components ← empty set
    while stack is not empty do
        u ← stack.pop()
        if u ∉ visited then
            component ← empty set
            DFSVisit(G_r, u, visited, component)
            add component to components
    return components
```

~~HASHING~~

ALGORITHM: BYGGE HUFFMAN TRÆR

Input: En mengde C med par (s, f) der s er et symbol og f er en frekvens

Output: Et Huffman-tre

```
Procedure Huffman(C)
    Q ← new PriorityQueue
    for (s, f) ∈ C do
        Insert(Q, new Node(s, f, null, null))
    while Size(Q) > 1 do
        v1 ← RemoveMin(Q)
        v2 ← RemoveMin(Q)
        f ← v1.freq + v2.freq
        Insert(Q, new Node(null, f, v1, v2))
    return RemoveMin(Q)
```