

AD Exam notes

Mia & Izabela

March 2024

Contents

1	Notation	5
1.1	Analyse af løkker	6
1.2	Invarianter	7
1.2.1	Eksempel 1	7
1.2.2	Eksempel 2	8
2	Sorting algorithms	9
2.1	Mergesort	9
2.2	Quicksort	9
2.3	Heapsort	10
2.3.1	Min-heap + max-heap	10
2.3.2	Build-max-heap	11
2.3.3	INSERT in heap (min/max)	12
2.3.4	Build-MIN-heap	14
2.3.5	Heap-extract	15
2.4	Insertionsort	16
2.5	Bubblesort	16
3	Proof of correctness	16
3.1	Loop invariant	16
3.1.1	Example	17
3.2	Proof by induction	17
3.2.1	Template	17
3.2.2	Example	17
3.2.3	Example 2	17
3.3	Proof by contradiction	18
3.3.1	Example	18
4	Shortest path and MST	18
4.1	Egenskaber ved Shortest paths	18
4.2	Eksempel på opgave	19
4.3	Dijkstras algoritme - shortest path	20
4.3.1	Example	20

4.4	Prims algorithm - MST	21
4.4.1	Example	21
4.5	Kruskals algorithm - MST	21
4.5.1	Eksempel	22
4.6	Bellmannford algorithm - Shortest path	22
4.6.1	Example	23
4.7	Breadth First Search (BFS)	23
4.8	BFS Step-by-step:	24
4.9	Depth first search (DFS)	26
4.9.1	Eksempel	26
4.9.2	Edge classification	27
4.10	Topological sort	28
4.11	Strongly connected components SCC	29
4.11.1	Example of SCC	29
4.11.2	Example of SCC (Fra Iza noter)	30
4.12	Safe edges	31
4.12.1	eksempel	33
5	Greedy algorithms	34
5.1	Greedy choice property	34
5.2	Acticity selection example	34
5.3	Huffman codes	35
5.3.1	Huffman algorithm	35
5.3.2	example	37
6	Binarysearch trees and fibonacci heaps	38
6.0.1	Time complexity	38
6.1	Red-black trees intro	38
6.2	Main operations on Red-Black Trees	39
6.3	Red-Black Trees Rotation	39
6.4	Red-Black Trees Insert (Adding an element to RB-trees)	41
6.4.1	Solutions	43
6.5	Red-Black Trees Deletion	45
6.6	Examples on Red-Black Trees Deletion	47
6.6.1	Time complexity (RB trees)	48
6.7	Fibonacci heaps	48
6.7.1	Extract min	50
6.7.2	Decrease key	54
6.7.3	Insert	56
6.7.4	Union	56
6.7.5	Delete	57

7	Dynamic programming	58
7.1	The rod cutting problem	59
7.2	Top down memoization	59
7.3	Bottom up approach	59
7.4	The longest common subsequence problem	59
8	Divide and conquer	60
8.1	Recursion tree	60
8.2	Master Method	61
8.2.1	Example	61
8.3	MC example	62
8.4	Substitution method	63
8.4.1	Hint for Multiple choice	63
9	Amortized costs	63
9.1	Aggregated analysis	63
9.2	Accounting method	64
9.3	The potential method	64
9.3.1	example	64
9.4	Dynamiske tabeller potential method	65
9.4.1	køretider	65
10	Disjoint sets	65
10.1	Example - simple	66
10.2	Example - elaborated	67
10.3	Runningtimes	68
11	Planesweep	68
12	Convex hulls	69
12.1	Graham scan algorithm	70
12.2	Jarvis march	71
12.3	Point pruning	71
12.4	Edge pruning	71
13	Closest pair problem	71
13.1	Closest pair problem - divide and conquer	72
14	LSM Træer	73
14.1	Eksempel	74
15	Maximum Sub-array	76
15.1	Eksempel	77
16	Karatsuba	78

17 Matrix chain multiplication	79
17.1 Varianser af MCM	79
17.1.1 Palindrome partitioning	79
17.1.2 Optimal binary search tree	80
17.1.3 Minimum/Maximum Cost to Multiply Chains of Matrices	81
17.1.4 EXAMPLE: Minimum Cost to Multiply Chains of Matrices	84
17.1.5 Partitioning Array to Minimize Sum	85
17.1.6 Chain of Pairs	85
17.1.7 Burst Balloons	86
18 knapsack	87

1 Notation

“>”

“≥”

“=”

“≤”

“<”

$$f(n) = \omega(g(n)) \quad f(n) = \Omega(g(n)) \quad f(n) = \Theta(g(n)) \quad f(n) = O(g(n)) \quad f(n) = o(g(n))$$

Asymptotisk notation

Hierarki over funktioner og deres ”størrelse”

1. n^n
2. $n!$
3. $a^n \quad a \in [1; \infty)$
4. $n^a \quad a \in [1; \infty)$
5. $n \cdot \log(n)$
6. n
7. $\sqrt{x} \sim x^a \quad a \in (0; 1)$
8. $\log(n)$

Husk potens regneregler:

$$a^{b \log_a c} = c^b$$

Husk:

$$n! = n \cdot \log(n)$$

1.1 Analyse af løkker

Algoritme loop3(n)

```
 $i = 0$   
 $j = 0$   
while  $i < n$   
    if  $j < n$   
         $j = j + 1$   
    else  
         $j = 0$   
         $i = i + 1$ 
```

I dette eksempel kører den ydre løkke fra i til n , og indre løkke kører fra j til n , derfor er køretid n^2

Algoritme loop4(n)

```
 $i = 1$   
while  $i \leq n * n$   
     $j = 1$   
     $k = 1$   
    while  $k \leq i$   
         $k = k + j$   
         $j = j + 1$   
     $i = 2 * i$ 
```

I ovenstående eksempel kører den ydre løkke $O(\log(n))$, Dette betyder, at den ydre løkke vil køre i $O(\log(n))$ tid, fordi det er det antal iterationer, der kræves for at værdien af i når op til $n * n$ gennem eksponentiel vækst ved fordobling. Den indre løkke har en køretid på $O(n)$ fordi den kører fra k op til

1.2 Invarianter

1.2.1 Eksempel 1

Givet to ikke-negative heltal n og m , så beregner nedenstående algoritme $n \cdot m$.

```
Algoritme Multiplikation( $n$ )  
Inputbetingelse : Heltal  $n \geq 0$  og  $m \geq 0$   
Outputkrav      :  $r = n_0 \cdot m_0$   
Metode          :  $r \leftarrow 0$   
                  { $I$ } while  $n > 0$  do  
                      if  $n$  er ulige then  
                           $r \leftarrow r + m$   
                           $n \leftarrow n - 1$   
                      else  
                           $m \leftarrow m * 2$   
                           $n \leftarrow n/2$ 
```

For hvert af nedenstående udsagn, angiv om de er en invariant I for algoritmen Multiplikation, hvor n_0 og m_0 angiver værdierne for henholdsvis n og m i starten.

	Ja	Nej
$0 \leq n \leq n_0$	☒	☐
$0 \leq m \leq m_0$	☐	☒
$r = m \cdot n$	☐	☒
$n_0 \cdot m_0 = r + n \cdot m$	☒	☐
$n_0 \cdot m_0 + r = n \cdot m$	☐	☒

1. $0 \leq n \leq n_0$: Dette ser ud til at være en invariant, fordi værdien af ' n ' falder ved hver iteration af løkken og aldrig øges eller bliver negativ.
2. $0 \leq m \leq m_0$: Dette er ikke en invariant. Værdien af ' m ' fordobles, når ' n ' er lige, så ' m ' kan overstige m_0 .
3. $r = m \cdot n$: Dette er ikke korrekt, fordi ' x ' bliver kun justeret, når ' n ' er ulige, og dermed vil denne lighed ikke altid være sand.
4. $n_0 \cdot m_0 = r + n \cdot m$: Dette ser ud til at være den korrekte invariant, fordi det samlede produkt af de oprindelige tal $n_0 \cdot m_0$ bevares igennem hvert trin i algoritmen. Når ' n ' er ulige, og ' m ' tilføjes til ' r ', reduceres ' n ' med 1, så det samlede produkt forbliver konstant. Når ' n ' er lige, bliver både ' n ' og ' m ' ændret, men produktet $n \cdot m$ forbliver det samme, da halvering af ' n ' og fordobling af ' m ' balancerer hinanden ud.
5. $n_0 \cdot m_0 + r = n \cdot m$: Dette er ikke korrekt, da det strider mod logikken af, hvordan invarianten skal opretholdes.

1.2.2 Eksempel 2

Betragt følgende funktion, i pseudo-kode notationen fra CLRS, hvor input A er en tabel af heltal og n er længden af A .

SECOND-SMALLEST(A, n)

```
1   $m = A[1]$ 
2   $r = \infty$ 
3  for  $i = 2$  to  $n$ 
4      if  $A[i] < m$ 
5           $r = m$ 
6           $m = A[i]$ 
7      elseif  $A[i] < r$ 
8           $r = A[i]$ 
9  return  $r$ 
```

Hvilke udsagn gyldige løkke-invarianter ved starten af hver iteration i **for**-løkken? Vælg ét eller flere korrekte svar.

1. $m = \min(A[1], \dots, A[i-1])$.
2. $m = \min(A[1], \dots, A[n])$.
3. $r \geq m$.
4. $m \geq 0$.
5. Mindst 1 element i $A[1], \dots, A[i-1]$ er mindre end eller lig med r .
6. Højst 2 elementer i $A[1], \dots, A[i-1]$ er mindre end eller lig med r .

1. $m = \min(A[1], \dots, A[i-1])$. - Korrekt, da m ved starten af hver iteration holder det mindste værdi af undersøgte elementer indtil nu.
2. $m = \min(A[1], \dots, A[n])$. - Ikke korrekt, da m kun vil indeholde den mindste værdi af hele arrayet efter den sidste iteration.
3. $r \geq m$. - Korrekt, da r altid bliver sat til en værdi, der er enten uendelig eller en værdi, der er større end eller lig med m .
4. $m \geq 0$. - Ikke altid korrekt, da der ikke er nogen garanti i pseudokoden for, at elementerne i A er ikke-negative.
5. Mindst 1 element i $A[1], \dots, A[i-1]$ er mindre end eller lig med r . - Korrekt, efter den første iteration og alle efterfølgende iterationer, hvor r bliver sat til en værdi fra arrayet, vil der være mindst ét element, som er lig med eller mindre end r .
6. Højst 2 elementer i $A[1], \dots, A[i-1]$ er mindre end eller lig med r . - Ikke altid korrekt, da der kan være flere end to elementer, der er mindre end eller lig med r , især efter at r har fået en værdi fra arrayet.

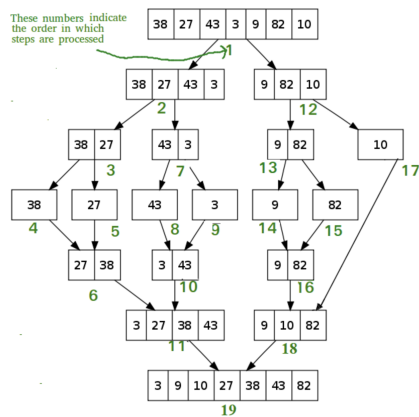
2 Sorting algorithms

2.1 Mergesort

Time complexity: $O(n \log n)$

When doing mergesort

Example



2.2 Quicksort

Worst case time complexity: $O(n^2)$

Partition takes (A,start,end) Choosing a pivot, and looking at

- ItemFromRight that is smaller than the pivot
- ItemFromLeft that is greater than the pivot

Then u swap them, and repeat until $itemFromleft > ItemFromRight$, and then switch the pivot with item from left, and this is now in the correct spot. This can now be repeated with the partiotions now made.

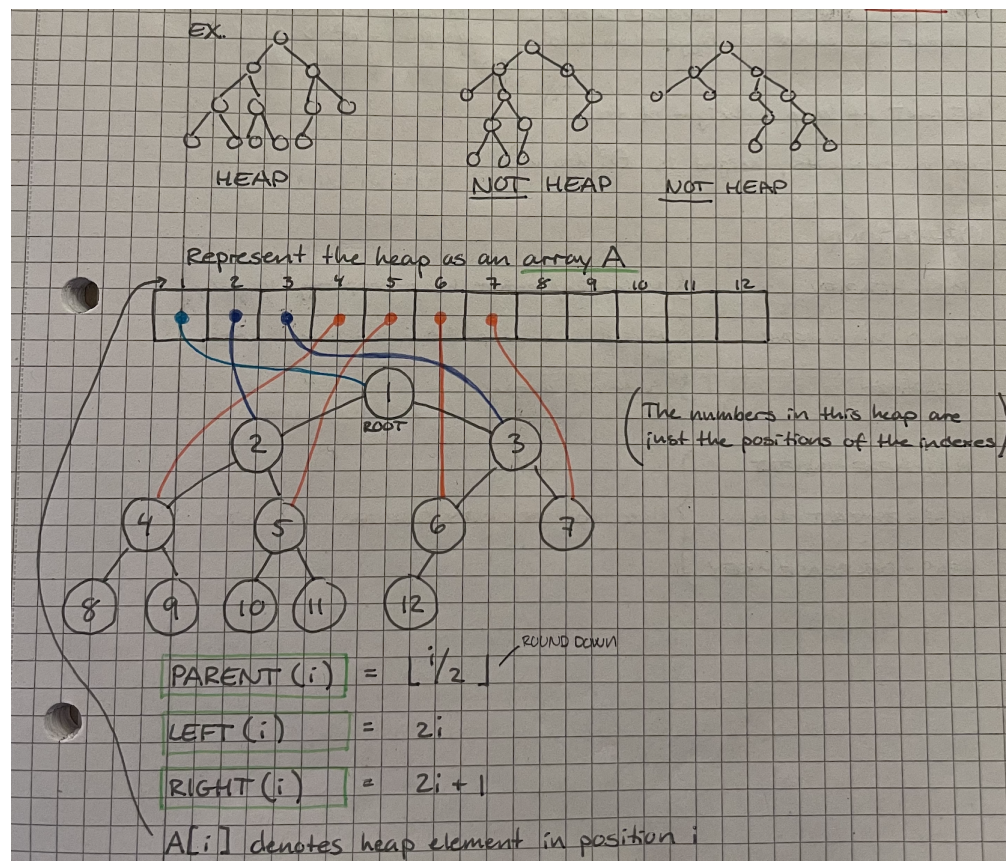
2.3 Heapsort

Heap Property:

1. For a min-heap: Every parent node has a value less than or equal to both of its children.
2. For a max-heap: Every parent node has a value greater than or equal to both of its children.

Complete Binary Tree:

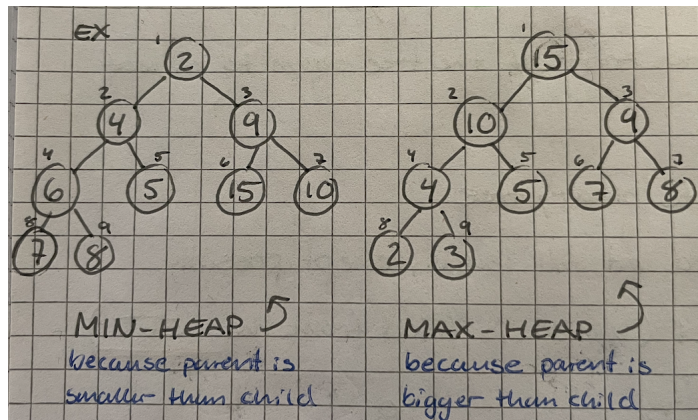
1. The heap must remain a complete binary tree. This means that every level, except possibly the last, must be fully filled, and all nodes are as far left as possible.



2.3.1 Min-heap + max-heap

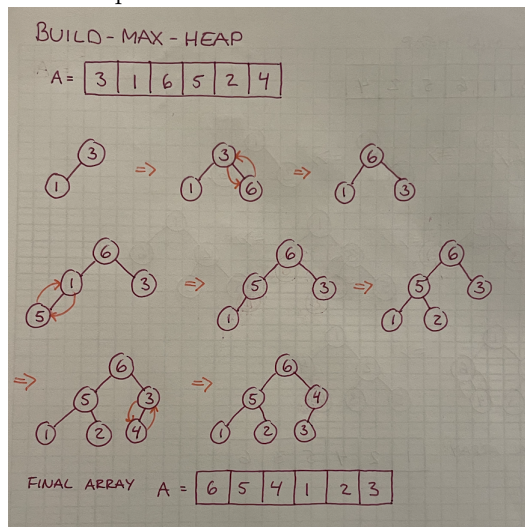
MIN-HEAP: For all i $A[PARENT(i)] \leq A[i]$

MAX-HEAP: For all i $A[PARENT(i)] \geq A[i]$



2.3.2 Build-max-heap

When building a max-heap We write the given array as a tree, and when inserting we see if the children is less than or equal to its parent, if not we switch with the parent.



Step-by-step

1. sæt tallene ind i rækkefølge i et træ fra højre til venstre.
2. Ryk rundt på træet, start nederst til højre.
3. Herefter bevæg dig "op" af, fra venstre og op.
4. Sæt tallene ind i et array i rækkefølge (dvs, fra toppen og ned, hvor man går fra venstre mod højre)

2.3.3 INSERT in heap (min/max)

1) Insert at the last available position

2) Min-Heap

Compare data we just inserted to parents data. If its less than the parents, we swap. Continue to do this until we reach the root node

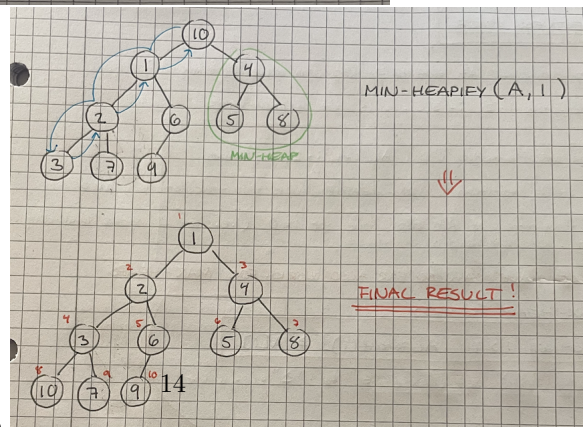
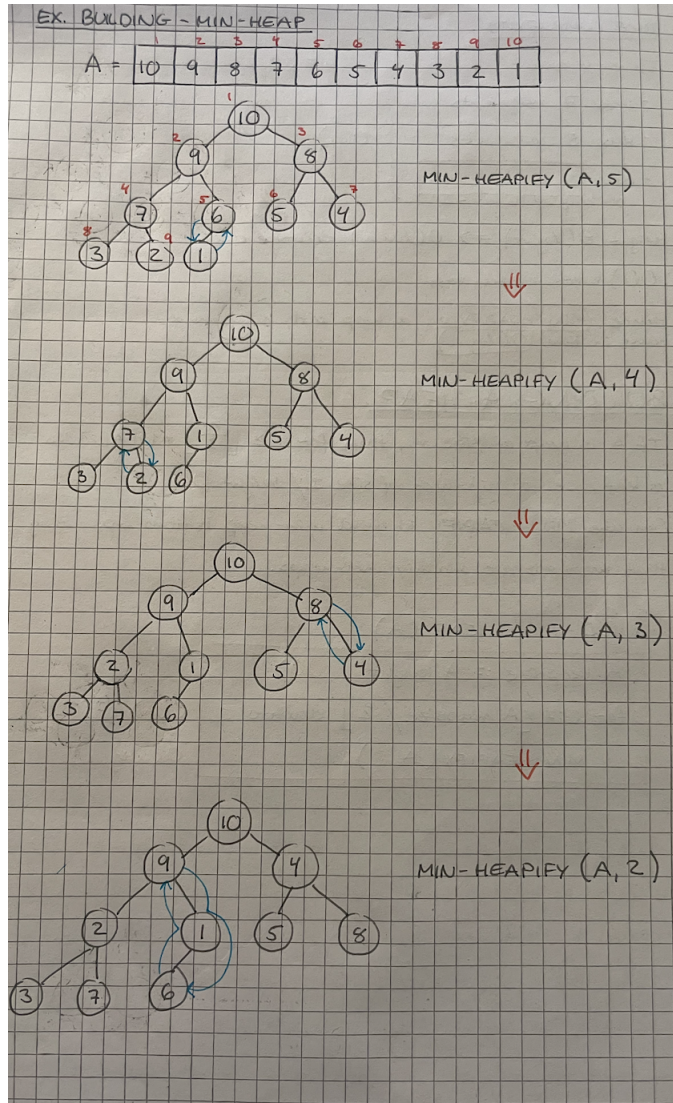
Max-Heap

Compare data we just inserted to parents data. If its greater than the parents, we swap. Continue to do this until we reach the root node

INSERTION PROCEDURE:

1. Step 1: Insert the element at the end of the heap, ensuring that the complete binary tree property is maintained. **This is typically the next available position at the bottom level, all the way to the left.**
2. Step 2: Compare the inserted element with its parent node (for min-heap, if it's less than its parent; for max-heap, if it's greater than its parent).
3. Step 3: If the heap property is violated (the element is smaller than its parent in a min-heap, or larger in a max-heap), swap the element with its parent.
4. Step 4: Repeat Step 3 until the element is in a position where the heap property is no longer violated, or it becomes the root of the heap.

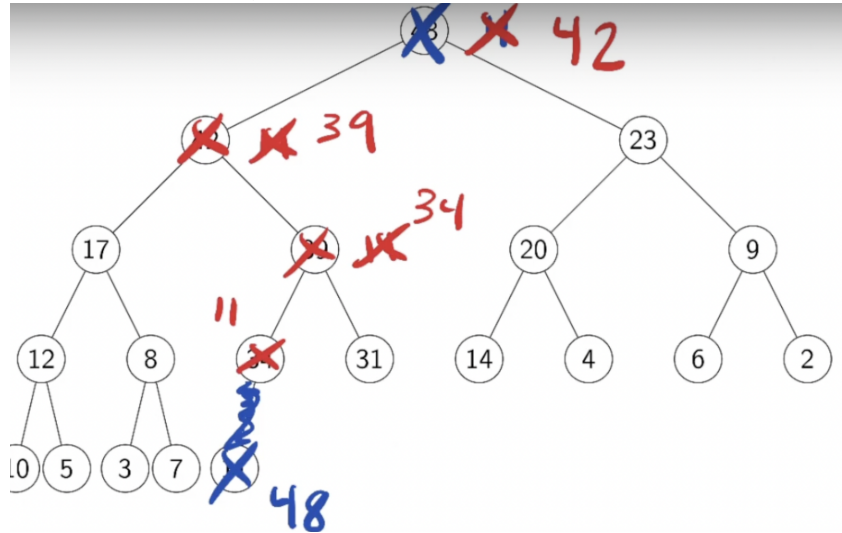
2.3.4 Build-MIN-heap



Fortsætter på næste side

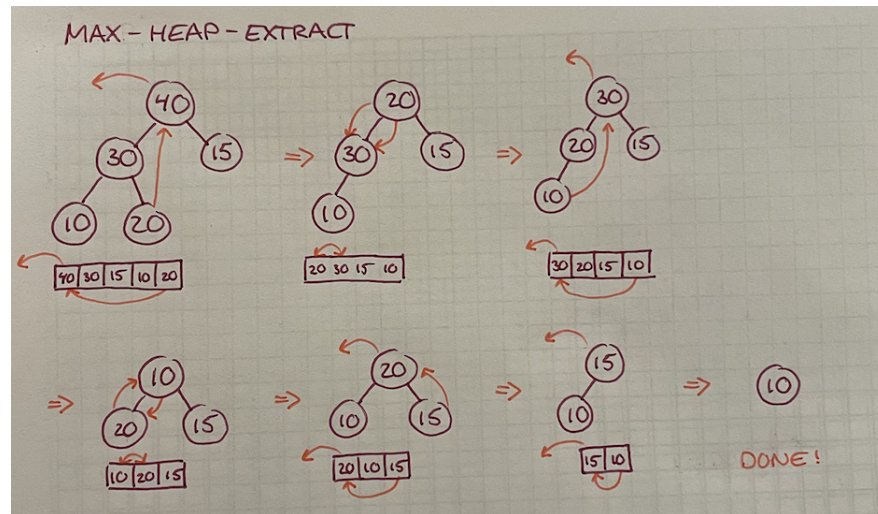
2.3.5 Heap-extract

To make heap extract max, we put the last element in the largest elements spot, and then fix the tree, so that rules aren't broken.

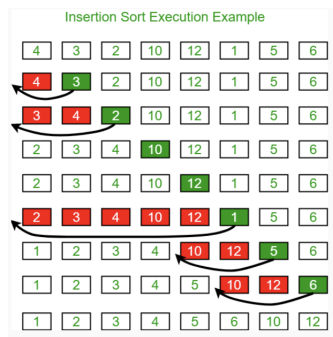


Step-By-Step

1. Først skal det sidste element i heapen byttes ud med rod elementet
2. Herefter skal der byttes rundt, sådan at de nederste niveauer ikke er numerisk større end niveauerne over
3. Heapen skal ikke bygges på samme måde som et binært søgetræ, der skal kun kigges på niveauer i heapen og ikke højre, venstre



2.4 Insertionsort



2.5 Bubblesort

3 Proof of correctness

3.1 Loop invariant

The loop invariant must hold true

- Before the loops starts
- Before each iteration of the loop starts
- After the loop terminates

3.1.1 Example

Code:

```
for i := 1 upto n {  
    B[i] := 1  
    for j := 1 upto i {  
        B[i] := B[i] * A[j]  
    }  
}
```

Loop invariant: Elementerne i B[i] stiger med $i!$ for hver iteration.

Initialization: Når $i=1$ vil B[i] være en, det må derfor være sand når vi starter

Maintenance: Når vi starter med et element B[i]=1, vil B[i] for hver iteration i linje 3-4, ganges med en højere værdi, for hver iteration i linje 1-4.

Termination: I sidste ende vil vi derfor have et array, hvor hvert element er $i!$ for $i=1$ op til n

3.2 Proof by induction

3.2.1 Template

1. Define som $p(n)$ to do the induction on
2. State the proof is by induction
3. State an proof the base case
example "For our base case, we'll prove $P(0)$, namely, that ..."
4. State an proof the inductive step
Typically, you'd prove this by assuming $P(k)$ and then proving $P(k+1)$.
We recommend specifically writing out both what the assumption $P(k)$ means and what you're going to prove when you show $P(k+1)$
5. conclude the proof

3.2.2 Example

For all $n \in \mathbb{N}$ holds $S(n) : \sum_{i=1}^n i = n(n+1)/2$

The base case will be $S(0) : \sum_{i=1}^0 i = 0(0+1)/2$

The inductive step will be $\sum_{i=1}^{n+1} i = (n+1) + \sum_{i=1}^n i = (n+1) + n(n+1)/2 = (n+1)(n+2)/2$

3.2.3 Example 2

For at bevise ved induktion, at $high_i - low_i \leq \frac{n}{2^i}$ for i iterationer af while-løkken i den givne algoritme, starter vi med basis-trinnet og går videre til induktion-strinnet.

Basis-trinnet: For $i = 0$ (før den første iteration), er $high_0 = \max(p, r + 1)$ og $low_0 = p$, så $high_0 - low_0 = \max(p, r + 1) - p$.

Da $p \leq r$, vil det største denne forskel kan være, være $r + 1 - p$, som er mindre end eller lig med n (da p og r er indekser i en liste af størrelsen n). Dette opfylder vores betingelse, da $\frac{n}{2^0} = n$.

Induktionstrinnet: Antag at for en given i , er $high_i - low_i \leq \frac{n}{2^i}$ sandt.

I næste iteration, $i + 1$, opdaterer vi enten $high$ eller low baseret på sammenligningen mellem x og $T[mid]$. - Hvis $x \leq T[mid]$, sætter vi $high$ til mid , som er $\left\lfloor \frac{low + high}{2} \right\rfloor$. - Hvis $x > T[mid]$, sætter vi low til $mid + 1$, som er $\left\lfloor \frac{low + high}{2} \right\rfloor + 1$.

I begge tilfælde reduceres intervallet mellem $high$ og low til det halve af det, det var før iterationen. Derfor kan vi udtrykke forskellen mellem $high$ og low efter $i + 1$ iterationer som $high_{i+1} - low_{i+1} \leq \frac{high_i - low_i}{2}$.

Ved at bruge vores induktionsantagelse:

$$high_{i+1} - low_{i+1} \leq \frac{\frac{n}{2^i}}{2} = \frac{n}{2^{i+1}}$$

Dette fuldender vores induktionsbevis. Efter i iterationer af while-løkken er forskellen mellem $high$ og low højst $\frac{n}{2^i}$.

3.3 Proof by contradiction

To prove a statement by contradiction, start by assuming the opposite of what you would like to prove. Then show that the consequences of this premise are impossible. This means that your original statement must be true.

3.3.1 Example

Prove that there is no largest number.

Suppose that there exist a largest number L , then there can be no number larger than L . But if we look at $Y=L+1$, y would be larger than L , and therefor no largest number exists.

4 Shortest path and MST

Spanning tree's will have a size of $|V| - 1$

4.1 Egenskaber ved Shortest paths

1. Subpath-egenskaben (Optimal substructure): Enhver delsti af en korteste sti er selv den korteste sti mellem dens endepunkter. Dette betyder, at hvis en sti P fra A til C via $B \rightarrow B \rightarrow C$ er den korteste sti, så er undervejen P_{AB} fra A til B den korteste sti fra A til B , og undervejen P_{BC} fra B til C er den korteste sti fra B til C .

2. Unikke vs. multiple korteste stier: I nogle grafer kan der være mere end én korteste sti mellem to knuder med den samme længde. Dette skaber muligheden for, at et korteste-veje træ kan variere baseret på, hvilken sti der vælges, selvom alle valgte stier opfylder kriteriet for at være den korteste.
3. Konsistens ved vægtændringer: Hvis vægten af en enkelt kant i grafen reduceres, kan de korteste stier, der ikke passerer gennem denne kant, ikke blive længere. Dette kan dog føre til, at nye stier bliver de korteste, hvis de inkluderer den kant, hvis vægt er reduceret.
4. Følsomhed over for kanttilføjelse og -fjernelse: Tilføjelse af en kant kan potentielt reducere længden af den korteste sti mellem to knuder, hvis den nye kant skaber en kortere rute. Omvendt kan fjernelse af en kant forlænge den korteste sti mellem to knuder, hvis den fjernede kant var del af den oprindelige korteste sti.
5. Påvirkning af negative cykler: I en graf med en eller flere negative cykler (hvor den samlede vægt af en lukket sti er negativ), kan konceptet med en korteste sti blive udefinerbart, da det at passere gennem en negativ cyklus gentagne gange kan reducere stiens samlede vægt ubegrænset.

4.2 Ekspempel på opgave

Et korteste-veje træ (eng. shortest paths tree) $G' = (V', E')$ er en subgraf af en graf $G = (V, E)$, der indeholder alle de korteste stier fra en given startknode til alle andre tilgængelige knuder i grafen. Dette træ har nogle specifikke egenskaber:

1. Rodknode: Træet har en specifik rodknode, som er startknuden, hvorfra de korteste stier til alle andre knuder udgår.
2. Akryklisk: Træet er akryklisk, hvilket betyder, at der ikke er nogen cykler. Dette er en direkte følge af definitionen på et træ.
3. Rttede kanter: Hvis den oprindelige graf er en rettet graf, vil alle kanterne i det korteste-veje træ også være rettede og pege væk fra rodknuden.
4. Dækker alle nåbare knuder: For hver knude i den oprindelige graf, som kan nås fra startknuden, vil der være en vej til denne knude i træet. Det betyder, at V' indeholder startknuden og alle knuder i V , der er nåbare fra startknuden.
5. Minimalt antal kanter: Træet indeholder det minimale antal kanter, der er nødvendige for at bevare den korteste vej fra rodknuden til alle andre knuder i V' . Dette betyder, at for hver knude i V' , undtagen rodknuden, er der præcis én indgående kant i E' , som er en del af den korteste vej fra rodknuden til denne knude.

6. Optimal sti-egendom: For enhver knude v i G' , repræsenterer vejen fra rod-knuden til v i træet den korteste vej fra rodknuden til v i den oprindelige graf G . Dette inkluderer, at vægten af den korteste vej fra startknuden til enhver anden knude i træet er lig med vægten af den korteste vej mellem disse to knuder i den oprindelige graf.
7. Undergraf: G' er en undergraf af G , hvilket betyder, at alle knuder og kanter i G' også findes i G .

4.3 Dijkstras algoritme - shortest path

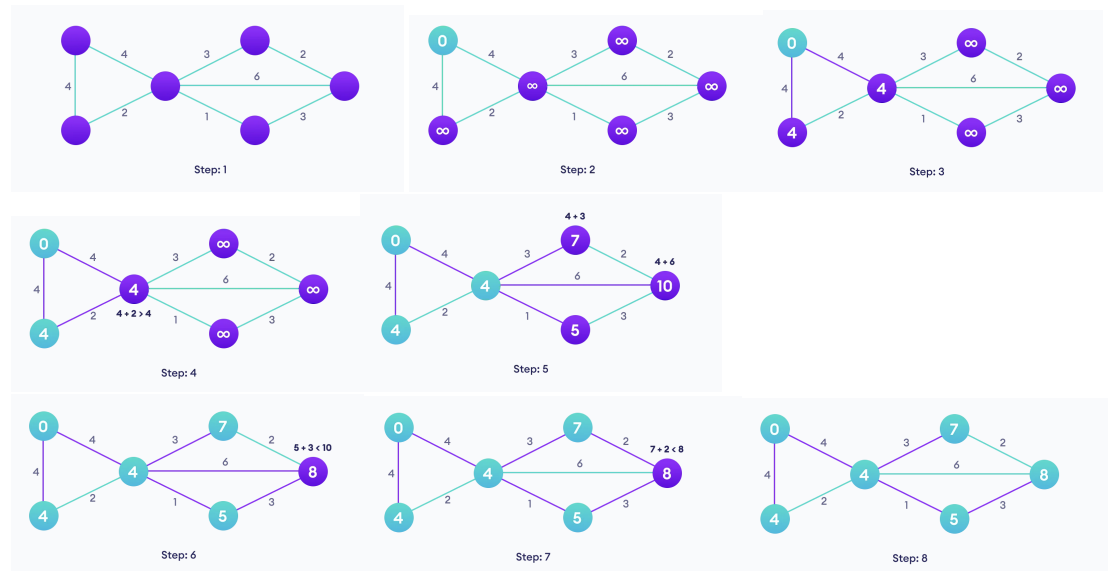
Gives the shortest path from one node to every other node

Step-by-step

1. When doing dijkstras, you have a list of unvisited nodes, where the starting nodes will be crossed of first.
2. You will then set all distances, apart from the starting node, to infinity.
3. You then Examine the edges leaving our starting node, and then update the shortest distance
4. We then cross of that vertex from our unvisited list, and the process keeps on until all vertexes has been visited.

Time complexity: $O(|E| \cdot \log|V|)$

4.3.1 Example



4.4 Prims algorithm - MST

This algorithm results in a minimum spanning tree, minimum weight connected graph with no cycles.

It is a greedy algorithm, choosing the smallest node first.

If the tree is distinct, the tree found is unique.

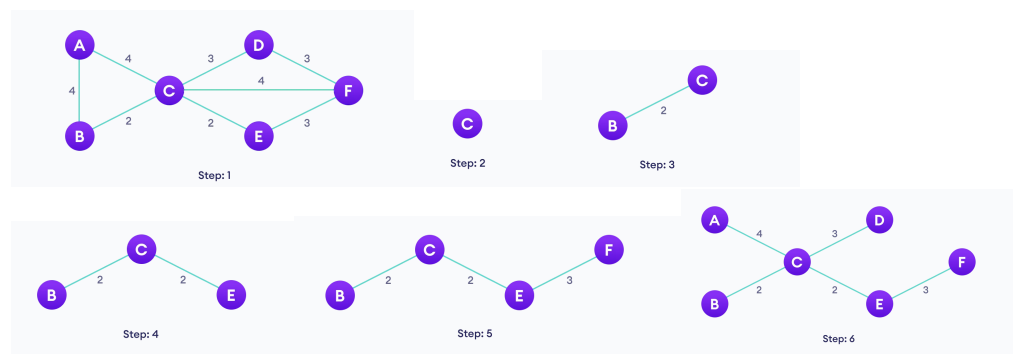
Time complexity depend on the data structure used

- Adjency matrix searching: $O(v^2)$
- Binary heap and adjency matrix: $O(v \log v + e \log v)$

Step-by-step

1. Start med en startknode
2. Find alle kanter der connecter nye noder til træet, det vil sige alle de kanter der går ud fra startnoden
3. Tilføj kanten med den laveste kant-værdi til træet
4. Find alle kanter der connecter nye noder til det eksisterende træ, og tilføj kanten med lavest værdi til træet
5. Gentag step 3 indtil alle noder er besøgt

4.4.1 Example



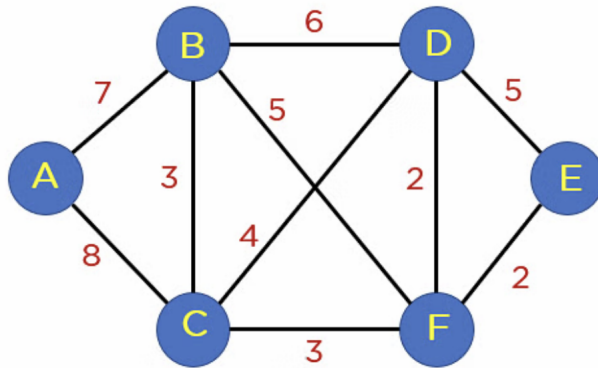
4.5 Kruskals algorithm - MST

- Sorts the edges in non decreasing order of their weight
- picks the smallest edge, and checks if a cycle is made, if not the edge is included
- Repeats step 2 until $|v| - 1$ edges is in the spanning tree

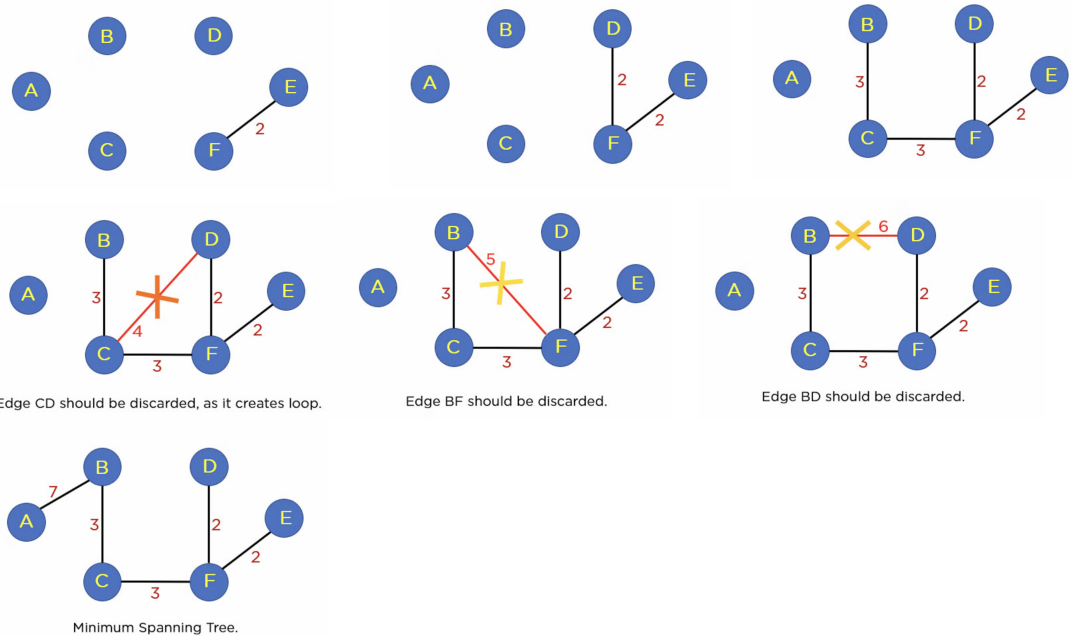
Timecomplexity: $O(E \log V)$

4.5.1 Eksempel

Vi starter med følgende graf



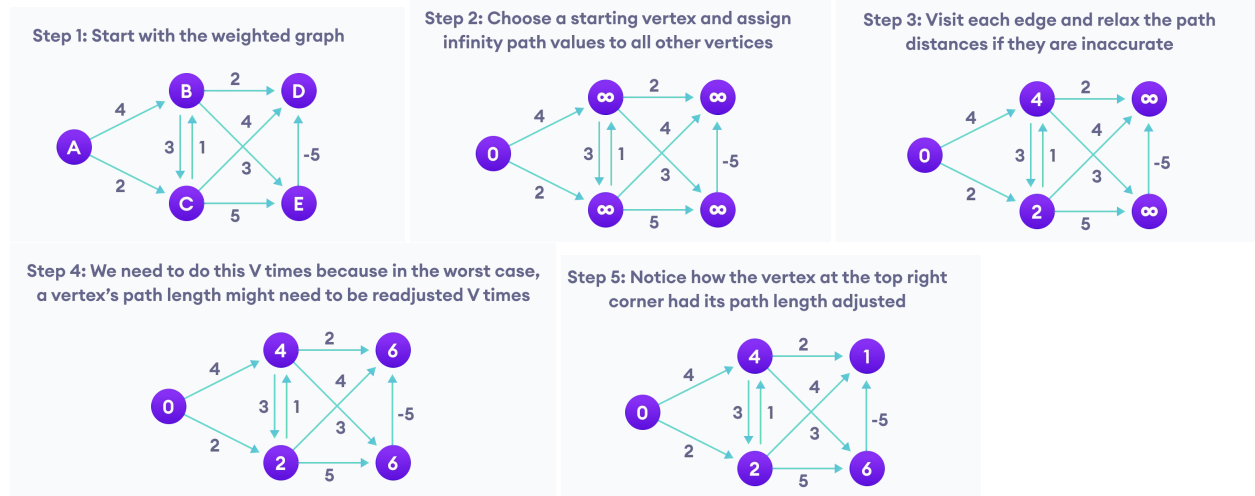
Herefter sortere vi vægtene i kanterne og tilføjer de kanter med mindst vægt, så længe de ikke danner en cyklus.



4.6 Bellmannford algorithm - Shortest path

Will have at most $|v| - 1$ edges at will update all edges at every iteration.
Time complexity: $O(V \cdot E)$

4.6.1 Example



4.7 Breadth First Search (BFS)

When doing breadth first search, you start of at some arbitrary node of a graph and explores the neighbor nodes, before moving on to the next level of neighbors.

All vertices start out WHITE, and vertices not reachable from the source vertex s stay WHITE the entire time.

A vertex that is reachable from s is DISCOVERED for the first time it is encountered during the search, at which, time it becomes grey indicating that it is now on the frontier at the search.

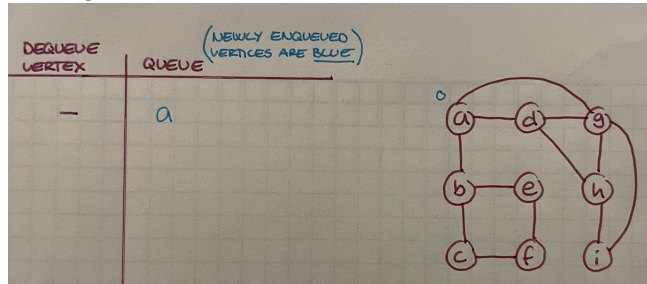
The queue contains all the gray vertices.

All the edges of a grey vertex will be explored, so that all of its neighbours will be discovered.

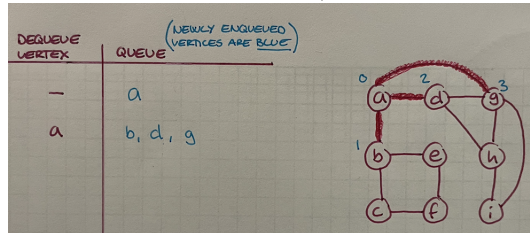
One all of a vertex's edges have been explored, the vertex is behind the frontier of the search, and it goes from GRAY to BLACK.

4.8 BFS Step-by-step:

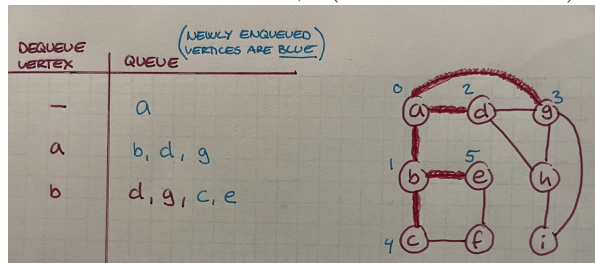
We begin in vertex **a**:



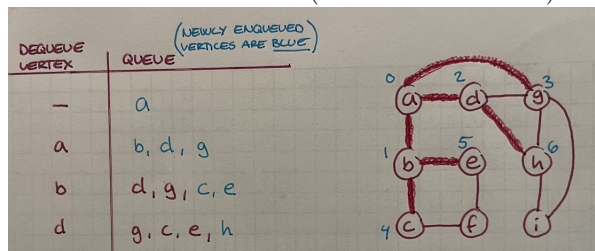
From **a** we discover **b, d, g** (mark road with red):



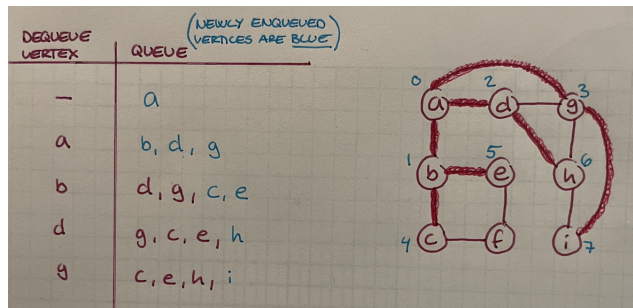
From **b** we discover **c, e** (mark road with red):



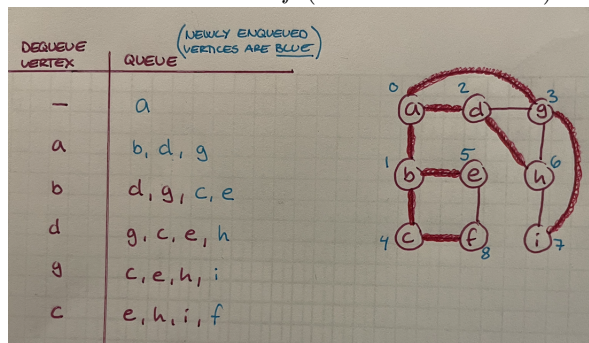
From **d** we discover **h** (mark road with red):



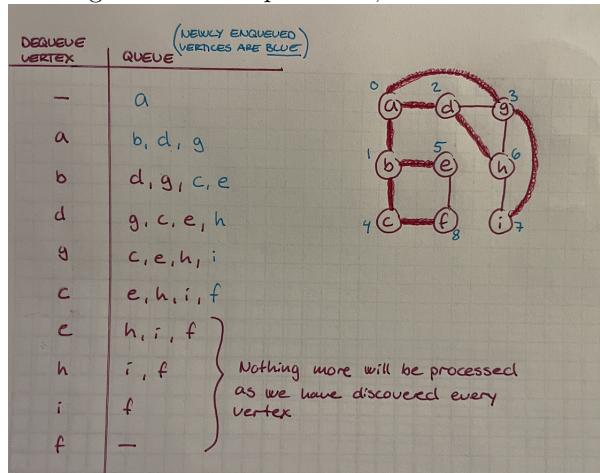
From **g** we discover **i** (mark road with red):



From **c** we discover **f** (mark road with red):



Nothing more will be processed, as we have discovered every vertex:



4.9 Depth first search (DFS)

Plunges into graph until it cannot go any further, at which point it backtracks and continues.

Step-by-step:

1. Gå fra startnoden så langt man kan, give hver besøgte
2. Når der nås et punkt hvor man ikke kan komme længere, gå tilbage, indtil der nås en node, der fører videre til en ubesøgt node
3. gentag step 1 og 2 så mange gange som nødvendigt for at alle noder besøges, og der er gået tilbage til startnoden

Discoverytime: Første node skal have nummer 0, herefter skal tallet for hver node besøgt stige med 1. Discovery time, er hvor i rækken af noder, den pågældende node besøges

Finishingtime: Anden gang en node besøges (ved "tilbagetrækning") skal noden have en finishing time. Denne finishing time kan godt være mindre end en anden nodes discoverytime, det vil altså sige at alle noder ikke skal besøges før der gås tilbage i rækken.

4.9.1 Eksempel

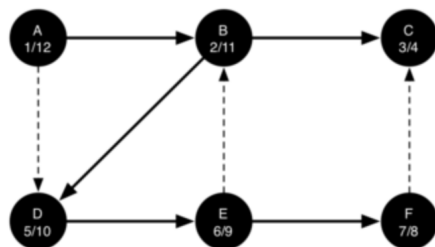
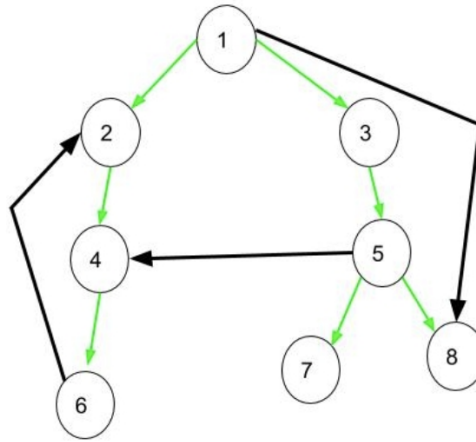


Figure 26: The Resulting Depth First Search Tree

4.9.2 Edge classification



- **Tree Edge:** It is an edge which is present in the tree obtained after applying DFS on the graph. All the Green edges are tree edges.
- **Forward Edge:** It is an edge (u, v) such that v is a descendant but not part of the DFS tree. An edge from **1** to **8** is a forward edge.
- **Back edge:** It is an edge (u, v) such that v is the ancestor of node u but is not part of the DFS tree. Edge from **6** to **2** is a back edge. Presence of back edge indicates a cycle in directed graph.
- **Cross Edge:** It is an edge that connects two nodes such that they do not have any ancestor and a descendant relationship between them. The edge from node 5 to 4 is a cross edge.

4.10 Topological sort

$G(V, E)$ is acyclic, i.e., NO cycles.

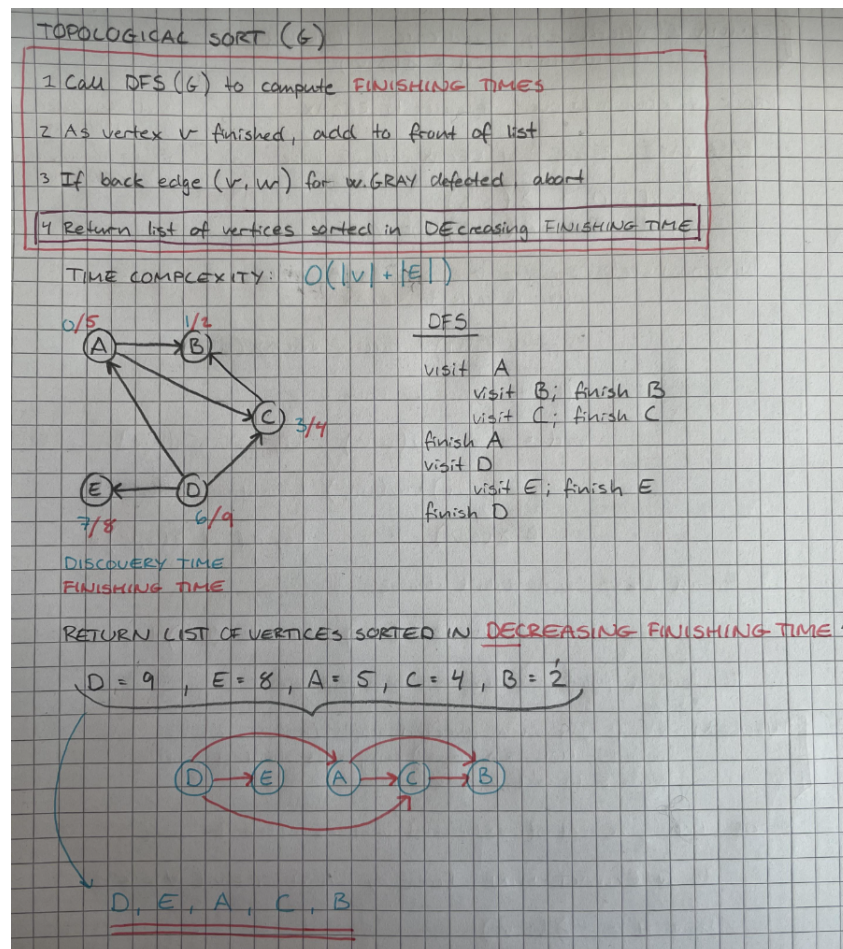
Directed edges.

DFS with timestamps.

TOPOLOGICAL-SORT(G)

- 1 call DFS(G) to compute finish times $v.f$ for each vertex v
- 2 as each vertex is finished, insert it onto the front of a linked list
- 3 **return** the linked list of vertices

Run DFS, compute finishing times og når en vertex er finished så indsæt i en liste i DECREASING order.



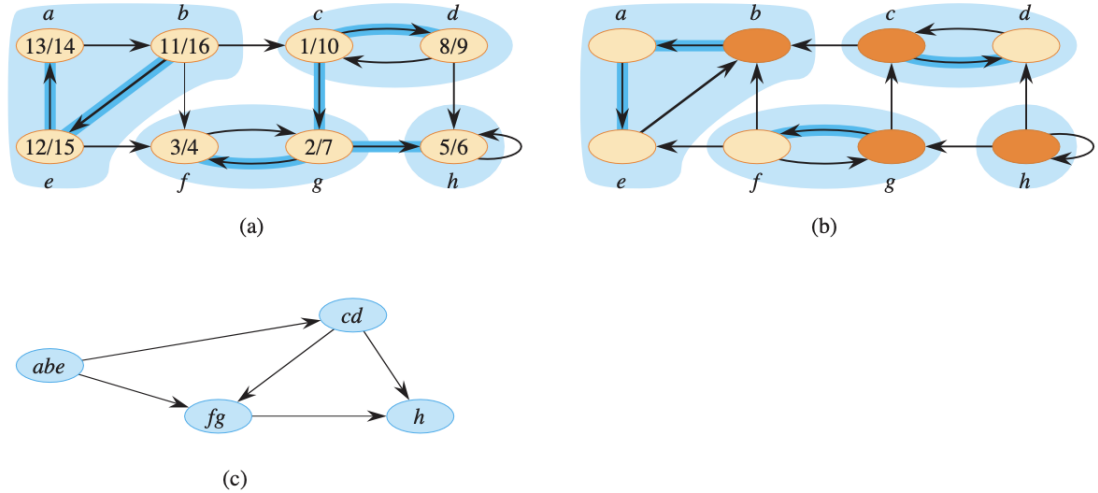
4.11 Strongly connected components SCC

A strongly connected component is the component of a directed graph that has a path from every vertex to every other vertex in that component. It can only be used in a directed graph.

STRONGLY-CONNECTED-COMPONENTS(G)

- 1 call DFS(G) to compute finish times $u.f$ for each vertex u
- 2 create G^T
- 3 call DFS(G^T), but in the main loop of DFS, consider the vertices in order of decreasing $u.f$ (as computed in line 1)
- 4 output the vertices of each tree in the depth-first forest formed in line 3 as a separate strongly connected component

4.11.1 Example of SCC



(a) A directed graph G . Each region shaded light blue is a strongly connected component of G . Each vertex is labeled with its discovery and finish times in a depth-first search, and tree edges are dark blue.

(b) The graph G^T , the transpose of G , with the depth-first forest computed in line 3 of STRONGLY-CONNECTED-COMPONENTS shown and tree edges shaded dark blue. Each strongly connected component corresponds to one depth-first tree. Orange vertices b, c, g , and h are the roots of the depth-first trees produced by the depth-first search of G^T .

(c) The acyclic component graph G^{SCC} obtained by contracting all edges within each strongly connected component of G so that only a single vertex remains in each component.

4.11.2 Example of SCC (Fra Iza noter)

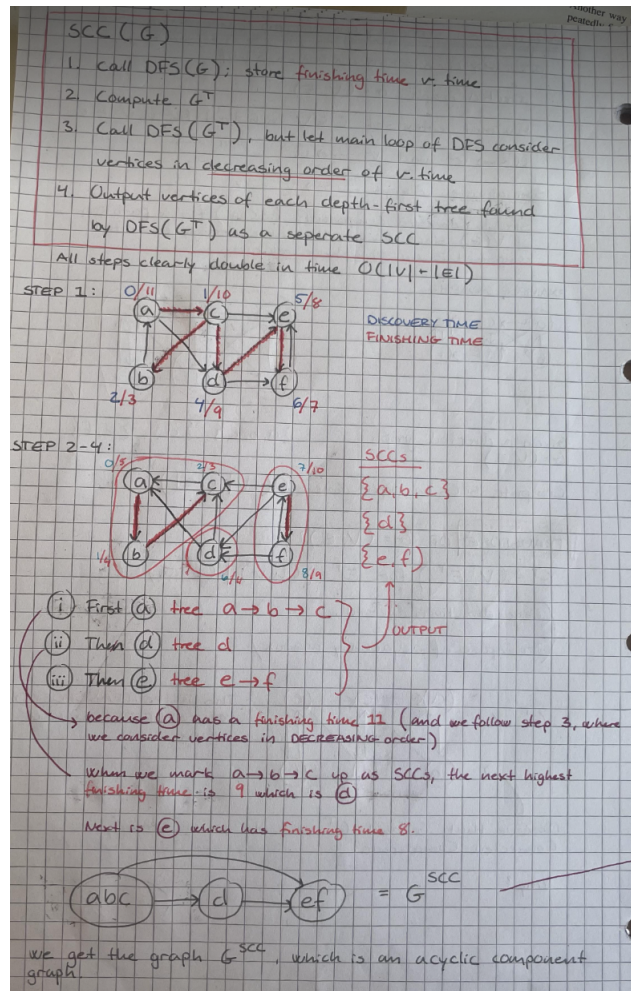


Figure 1: Eksempel af SCC (Fra Iza noter)

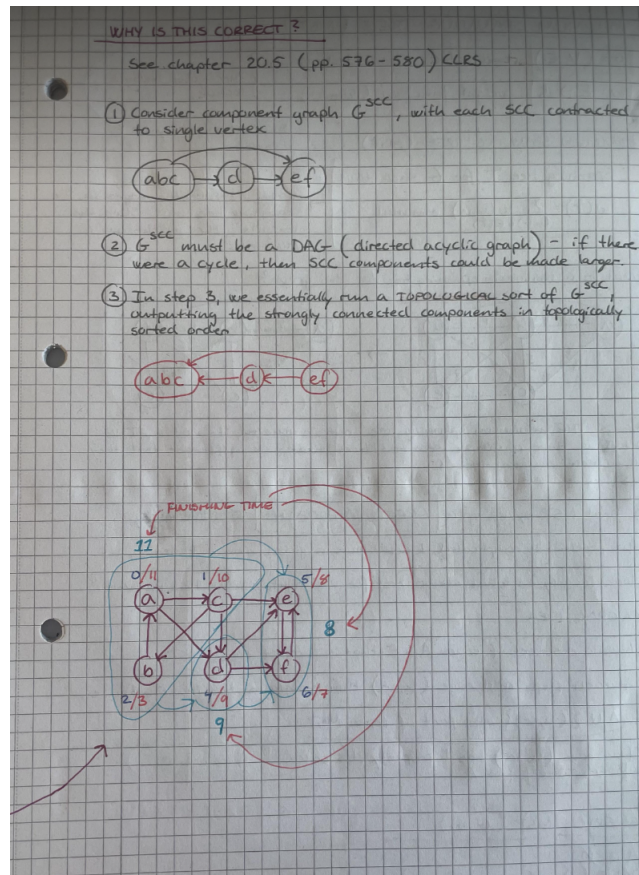


Figure 2: Eksempel af SCC (fortsat) (Fra Iza noter)

4.12 Safe edges

"Prior to each iteration, A is a subset of some minimum spanning tree."

Safe edge: An edge that may be added to A without violating the invariant that A is a subset of some minimum spanning tree.

Cut: is a partition of V into S and $V - S$.

Crossing: An edge $(u, v) \in E$ crosses the cut $(S, V-S)$ if one of its endpoints is in S and the other is in $(V-S)$.

Respecting: A cut respects a set A of edges if no edge in A crosses the cut.

Light edge: An edge is a light edge crossing a cut if its weight is the minimum of any edge crossing the cut. Note that there may be more than one light edge.

Rule for recognizing safe edges is given by the following theorem:

(Theorem 23.1) Let $G = (V, E)$ be a connected, undirected graph with a real-valued weight function w defined on E . Let A be a subset of E that is included in some minimum spanning tree for G , let $(S, V - S)$ be any cut of G that respects A , and let (u, v) be a light edge crossing $(S, V - S)$. Then, edge (u, v) is safe for A .

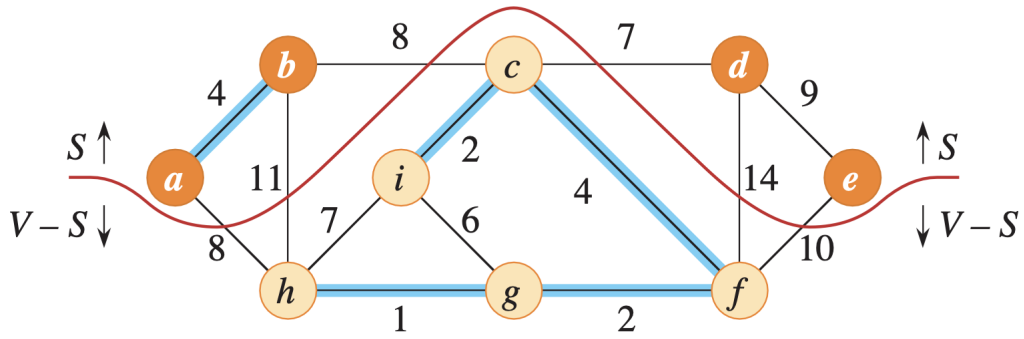


Figure 3: Safe edge (d, c) example

A cut $(S, V - S)$ of the graph from Figure 21.1. Orange vertices belong to the set S , and tan vertices belong to $V - S$. The edges crossing the cut are those connecting tan vertices with orange vertices. The edge (d, c) is the unique light edge crossing the cut. Blue edges form a subset A of the edges. The cut $(S, V - S)$ respects A , since no edge of A crosses the cut.

CHATGPT's svar på hvad safe edges er:

In graph theory, a "safe edge" typically refers to an edge that can be added to a subset of edges in a graph without creating a cycle. This concept is often used in the context of algorithms like Kruskal's algorithm for finding a minimum spanning tree.

In Kruskal's algorithm, edges are added to the spanning tree one at a time, always selecting the smallest edge that doesn't create a cycle when added to the current tree. Before adding an edge, it's necessary to check if adding that edge would create a cycle. If it would not, the edge is considered safe to add.

So, in summary, a safe edge is an edge that can be added to a subset of edges without creating a cycle in the graph.

4.12.1 eksempel

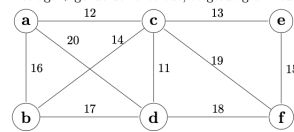
Vi betragter et snit i en graf defineret ved $S_2 = \{a, c, e\}$ og den komplementære mængde $V - S_2 = \{b, d, f\}$. Kantmængden A er givet ved $A = \{\{a, c\}, \{c, d\}\}$. For at snittet $(S_2, V - S_2)$ skal respektere kantmængden A , må ingen af kanterne i A krydse snittet. Med andre ord:

- Ingen kant i A skal have en ende i S_2 og den anden ende i $V - S_2$.

Men, da kanten $\{c, d\}$ har $c \in S_2$ og $d \in V - S_2$, krydser denne kant snittet, og snittet respekterer derfor ikke kantmængden A . Så udsagnet

”Lad $S_2 = \{a, c, e\}$. Snittet $(S_2, V - S_2)$ respekterer kantmængden A .”

Betragt følgende uorienterede, vægtede graf med knudemængde $V = \{a, b, c, d, e, f\}$:



Betragt kantmængden $A = \{\{a, c\}, \{c, d\}\}$, som er en delmængde af et minimalt udspændende træ (eng. *minimum spanning tree*). Vi er interesserede i snit (eng. *cuts*), sikre kanter (eng. *safe edges*) for A , og snit der respekterer (eng. *respects*) en kantmængde A . Hvilke af følgende udsagn er sande? Vælg ét eller flere korrekte svar.

er ikke sandt.

5 Greedy algorithms

A greedy algorithm solves the problem by repeatedly making the choice that looks best in the moment.

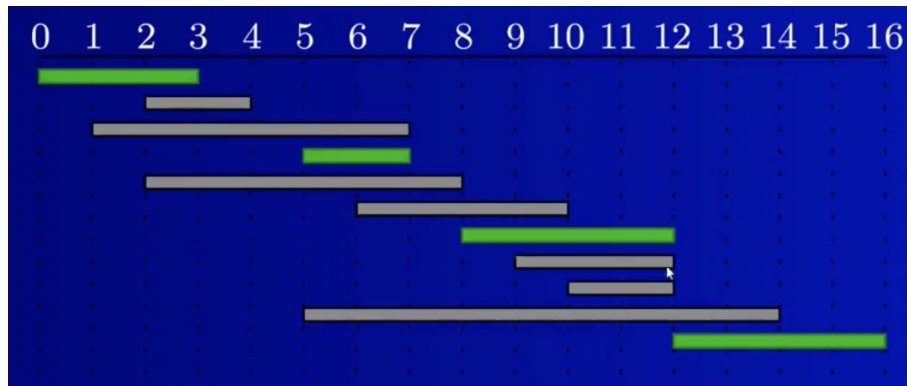
It makes the choice first, and then it solves the problem, in contrary to dynamic programming, which solves the problem and then make a choice.

5.1 Greedy choice property

If there exists a optimal solution to our problem P and which includes the greedy choice, we have greedy choice property.

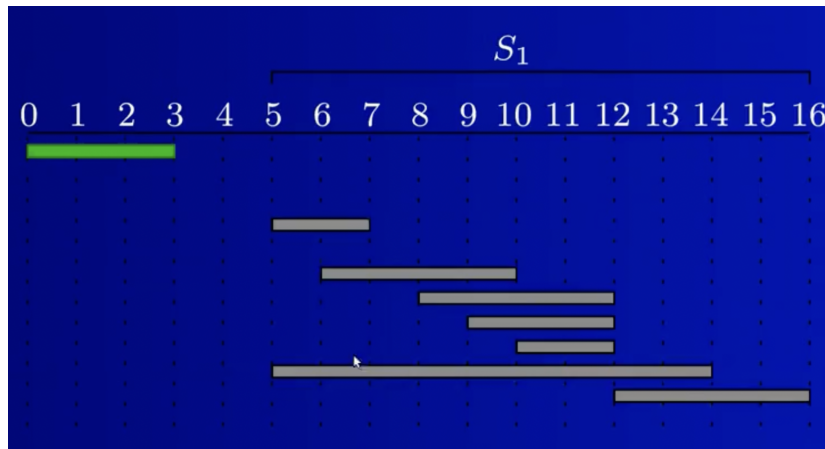
We also need to have optimal substructure, meaning that if greedy choice x is in an optimal solution to P then this optimal solution consists of x and an optimal solution to P' .

5.2 Activity selection example



It is seen here that none of them are overlapping, and it is not possible to find more than 4 that are not overlapping, hence this is the optimal solution (maximum activities).

To show the greedy choice property and optimal substructure in this we can look at the following



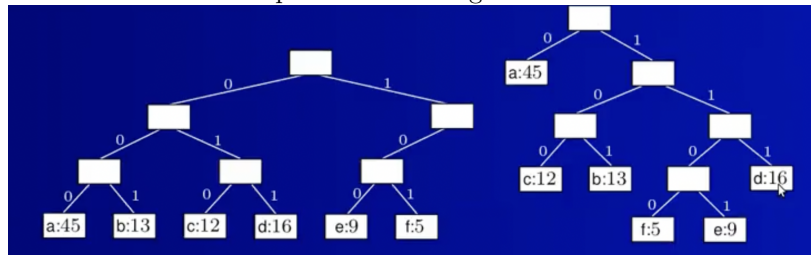
It is here seen that we choose the first activity, and let's call it A_1 . A_1 must be part of the optimal solution together with the first and best solution to S_1

5.3 Huffman codes

Let's say we have a prefix code looking like this

Letters:	a	b	c	d	e	f
Frequency:	45	13	12	16	9	5
3 bits:	000	001	010	011	100	101
Variable length:	0	101	100	111	1101	1100

We would then have a parse tree looking like this

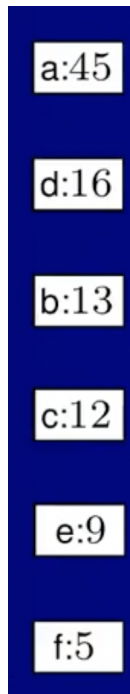


Where the tree to the left is the original tree, and the tree to the right is the tree matching the variable length prefix code.

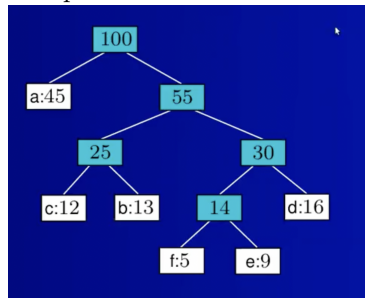
5.3.1 Huffman algorithm

Pairs the nodes with minimum frequency and set their parent node to the sum of their frequency.

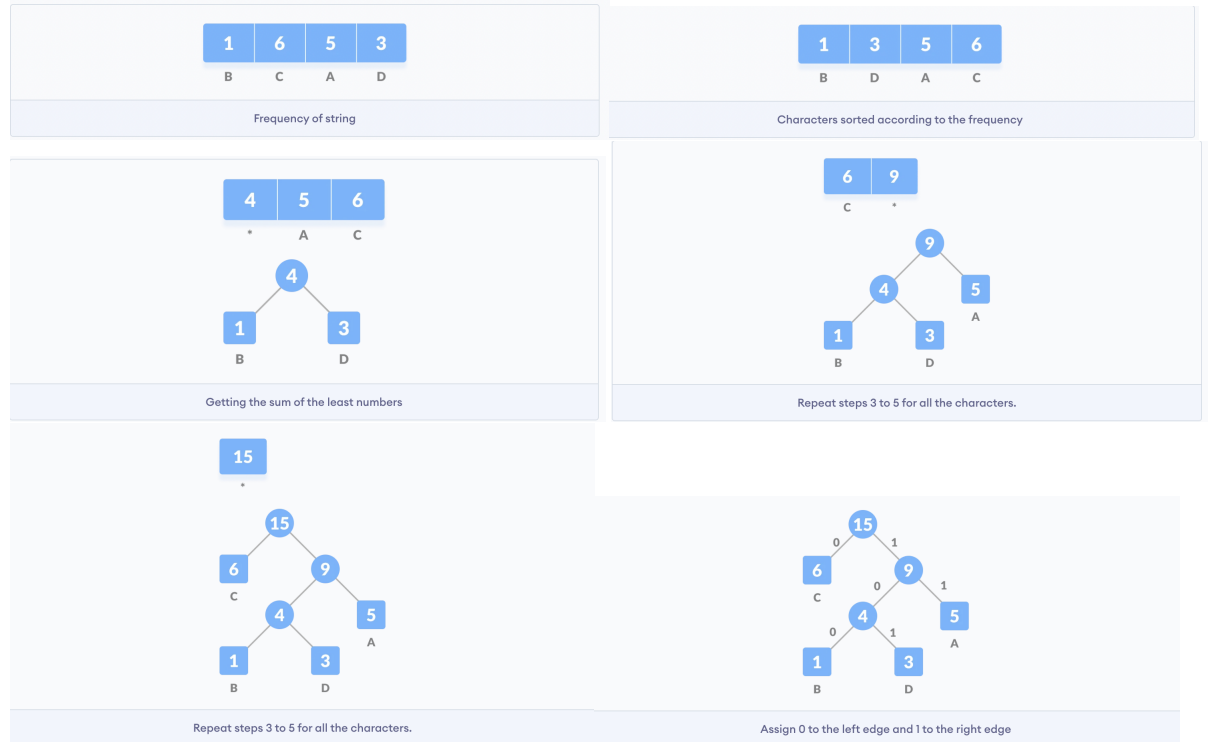
Given the following input



The parse tree will look like the following



5.3.2 example



6 Binarysearch trees and fibonacci heaps

Binary search trees

6.0.1 Time complexity

	AVL Tree		Binary Search Tree	
	Average	Worst	Average	Worst
Insertion	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Searching	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Traversal	$O(n)$	$O(n)$	$O(n)$	$O(n)$

6.1 Red-black trees intro

Every node is colored either red or black. Root of the tree is black.

All leaves are black.

If a node is red, then its children are black i.e., there can't be consecutive red nodes.

Small elements are to the left and large elements are to the right

All the simple paths from a node to descendant leaves contain the same number of black nodes.

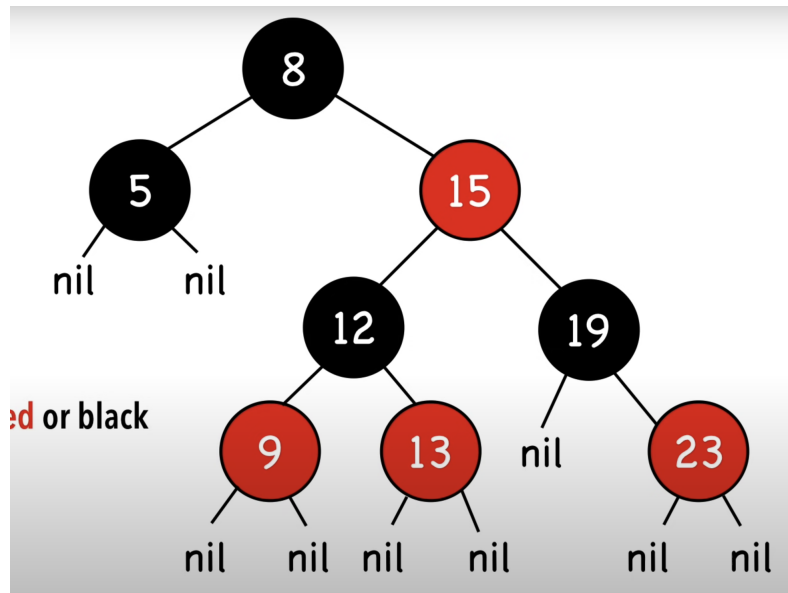
General for binary search trees

1. Sorten/ordered search tree
2. Nodes can have 2 subtrees
3. Items to the left of a given node is smaller
4. Item to the right of a given node is larger

Balanced search trees has following criterias:

- A node is either red or black
- has a guranteed height of $O(\log n)$ for n items

- The root and leaves are black
- If a node is red, its children are black
- All paths from a Node to its NIL descendants contains the same number of black nodes



6.2 Main operations on Red-Black Trees

- Search
- Insert
- Remove

Inserting and removing items in a RB tree, may result in violation of RB tree properties. To fix this we use rotations:

6.3 Red-Black Trees Rotation

1. alters the structure of a tree by rearranging subtrees
2. goal is to decrease the height of the tree
 - red-black trees: maximum height of $O(\log n)$
 - larger subtrees up, smaller subtrees down
3. does not affect the order of elements
4. smaller elements will still be to the left

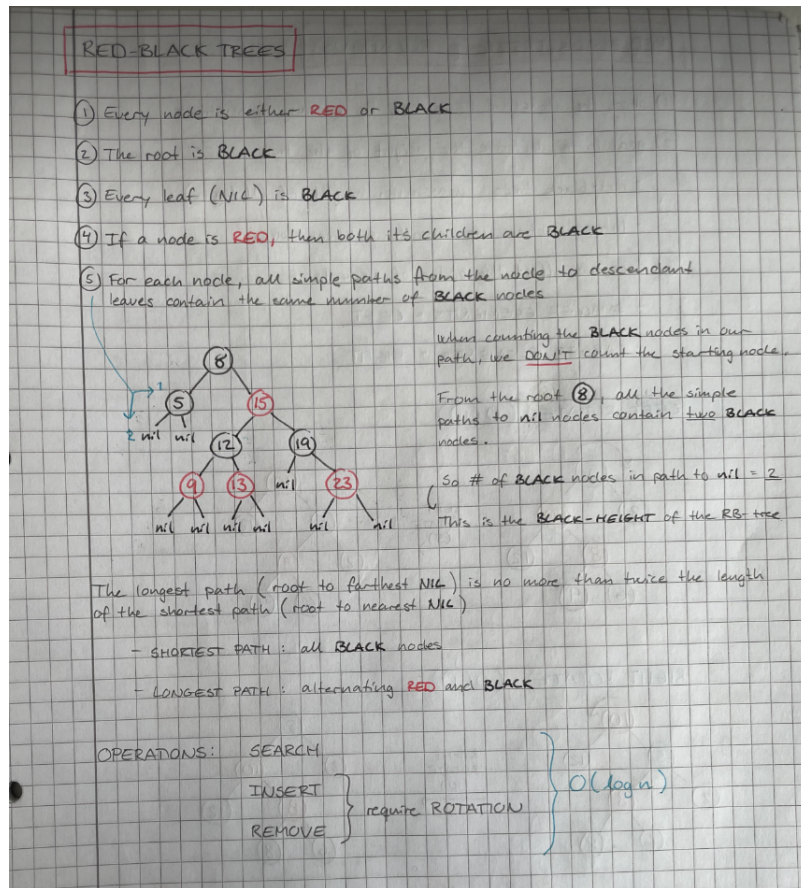


Figure 4:

5. larger elements will still be to the right

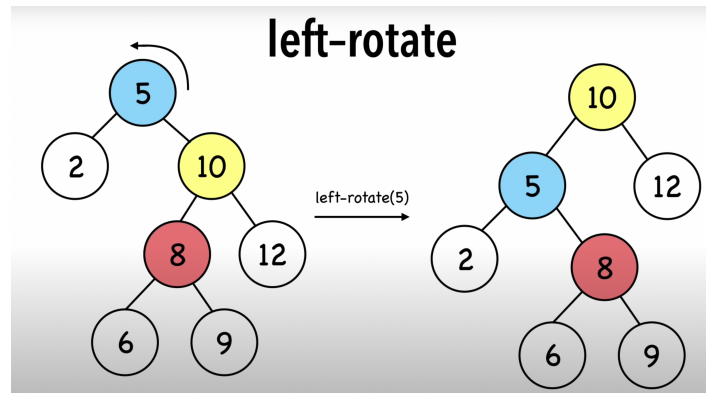


Figure 5: Left-rotation

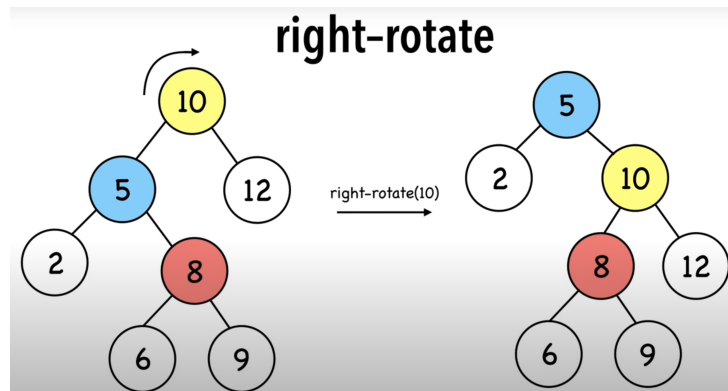


Figure 6: Right-rotation

6.4 Red-Black Trees Insert (Adding an element to RB-trees)

When adding an element, if it is smaller than the root it will go to the **left**, if it is greater than it will go to the **right**. If it is less than the node which is a child to the root it will become a left child, if it is greater it will become a right child. If the node is already there, you do nothing.

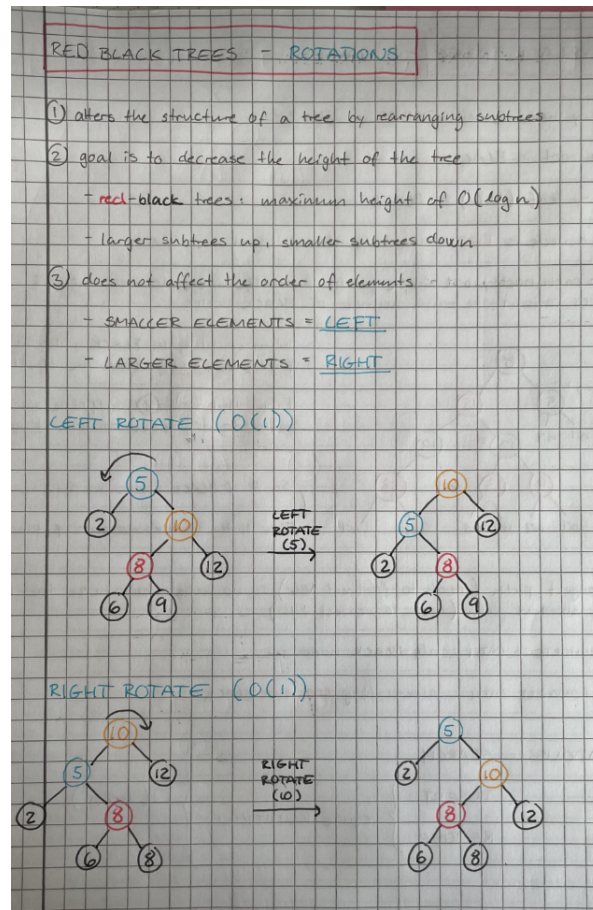
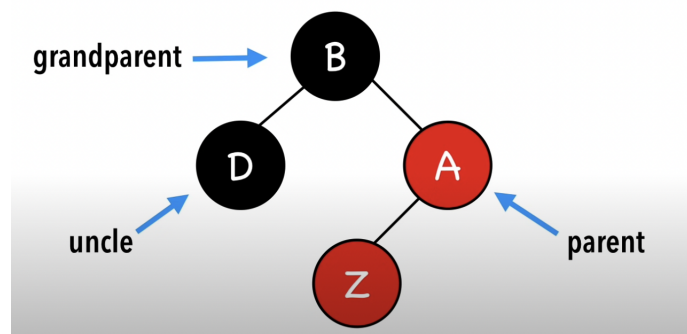


Figure 7: Rotatation

Z's relationships



1. Insert z and colour it red
2. recolor and rotate to fix violations

There are 4 scenarios

1. Case 0: $Z = \text{root}$
2. Case 1: $Z.\text{uncle} = \text{Red}$
3. Case 2: $Z.\text{uncle} = \text{black}(\text{triangle})$
4. Case 3: $Z.\text{uncle} = \text{black}(\text{line})$

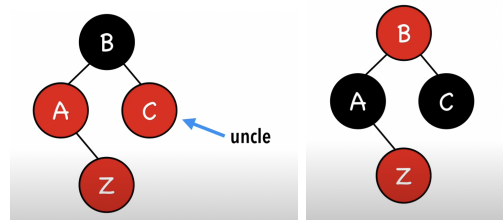
6.4.1 Solutions

Case 0:

When $Z = \text{root}$, the solution is to colour the node black

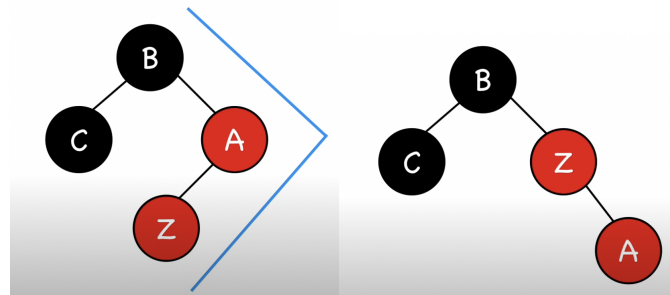
Case 1:

When $Z.\text{uncle} = \text{red}$, the solution is to recolour, the parent, grandparent and uncle



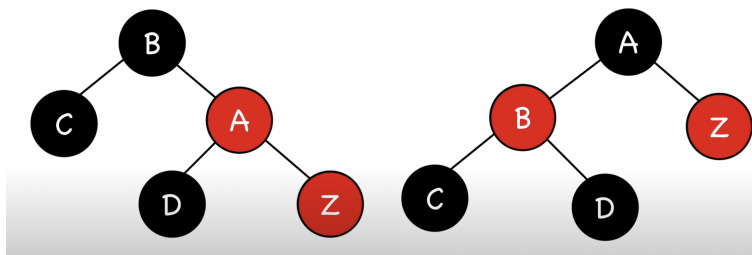
Case 2:

When $Z.\text{uncle} = \text{black}(\text{triangle})$, the solution is to rotate z.parent

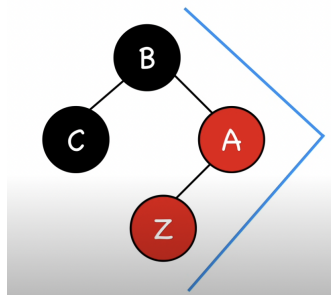
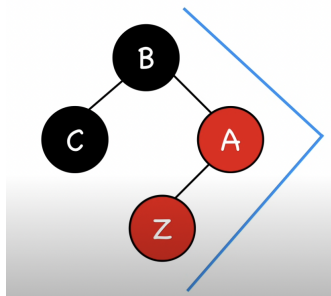


Case 3:

When $z.\text{uncle} = \text{black}(\text{line})$, the solution is to rotate z.grandparent and recolour



When one solution is used, we use the other solutions until we no longer have any violations.



6.5 Red-Black Trees Deletion

Steps:

RB-DELETE: Step 1

- If the node we deleted has 2 NIL children, its replacement x is NIL.
- If the node we deleted has 1 NIL child and 1 non-NIL child, its replacement x is the non-NIL child.
- If the node we deleted has 2 non-NIL children, set x to the replacement's right child before the replacement is spliced out.

RB-DELETE: Step 2

- If the node we deleted is red and its replacement is red or NIL, we are done.
- If the node we deleted is red and its replacement is black and non-NIL, color the replacement red and proceed to the appropriate case.
- If the node we deleted is black and its replacement is red, color the replacement black. We are done.
- If the node we deleted is black, its replacement is black, and x is the root of the tree, we are done.
- If the node we deleted is black, its replacement is black, and x is not the root of the tree, proceed to the appropriate case.

RB-DELETE-FIXUP: Cases

0. Node x is red
1. Node x is black and its sibling w is red
2. Node x is black, its sibling w is black, and both of w 's children are black
3. Node x is black, its sibling w is black, and
 - If x is the left child, w 's left child is red and w 's right child is black
 - If x is the right child, w 's right child is red and w 's left child is black
4. Node x is black, its sibling w is black, and
 - If x is the left child, w 's right child is red
 - If x is the right child, w 's left child is red

Cases:

Case 1

Node x is black and its sibling w is red

1. Color w black
2. Color $x.p$ red
3. Rotate $x.p$
 - a. If x is the left child do a left rotation
 - b. If x is the right child do a right rotation
4. Change w
 - a. If x is the left child set $w = x.p.right$
 - b. If x is the right child set $w = x.p.left$
5. With x and our new w , decide on case 2, 3, or 4 from here.

Case 0

Node x is red

1. Color x black. We are done.

Case 3

Node x is black, its sibling w is black, and

- If x is the left child, w 's left child is red and w 's right child is black
- If x is the right child, w 's right child is red and w 's left child is black

1. Color w 's child black
 - a. If x is the left child, color $w.left$ black
 - b. If x is the right child, color $w.right$ black
2. Color w red
3. Rotate w
 - a. If x is the left child do a right rotation
 - b. If x is the right child do a left rotation
4. Change w
 - a. If x is the left child set $w = x.p.right$
 - b. If x is the right child set $w = x.p.left$
5. Proceed to case 4.

Case 2

Node x is black, its sibling w is black, and both of w 's children are black

1. Color w red
2. Set $x = x.p$
 - a. If our new x is red, color x black. We are done.
 - b. If our new x is black and is the root of the tree, we are done.
 - c. If our new x is black and is not the root of the tree, decide on case 1, 2, 3, or 4 from here. Note that we have a new w now.

Case 4

Node x is black, its sibling w is black, and

- If x is the left child, w 's right child is red
- If x is the right child, w 's left child is red

1. Color w the same color as $x.p$
2. Color $x.p$ black
3. Color w 's child black
 - a. If x is the left child, color $w.right$ black
 - b. If x is the right child, color $w.left$ black
4. Rotate $x.p$
 - a. If x is the left child do a left rotation
 - b. If x is the right child do a right rotation
5. We are done.

6.6 Examples on Red-Black Trees Deletion

RED - BLACK TREES (DELETIONS):

RECTIONS:

- 1) Transplant
 - helps us move subtrees within the tree
- 2) Delete
 - deletes the node
- 3) Delete - Fixup
 - fixes any red-black violations

IF A NODE HAS TWO CHILDREN, FIND ITS SUCCESSOR (THE NODE WITH THE SMALLEST KEY GREATER THAN THE NODE TO BE DELETED)

VIGILANCE!

EXAMPLE (1) HOW DO WE DELETE (6):

If the node we deleted has 2 NIL children, its replacement x is NIL.

If the node we deleted is RED and its replacement is RED or NIL, we are DONE.

EXAMPLE (2) HOW DO WE DELETE (1):

If the node has 2 NIL child and 2 NON-NIL child, its replacement x is the NON-NIL child. So the replacement here is (11).

If the node we deleted is BLACK (1) and its replacement (11) is RED, color the replacement (11) BLACK (6) ... we are done.

EXAMPLE (3) HOW DO WE DELETE (17):

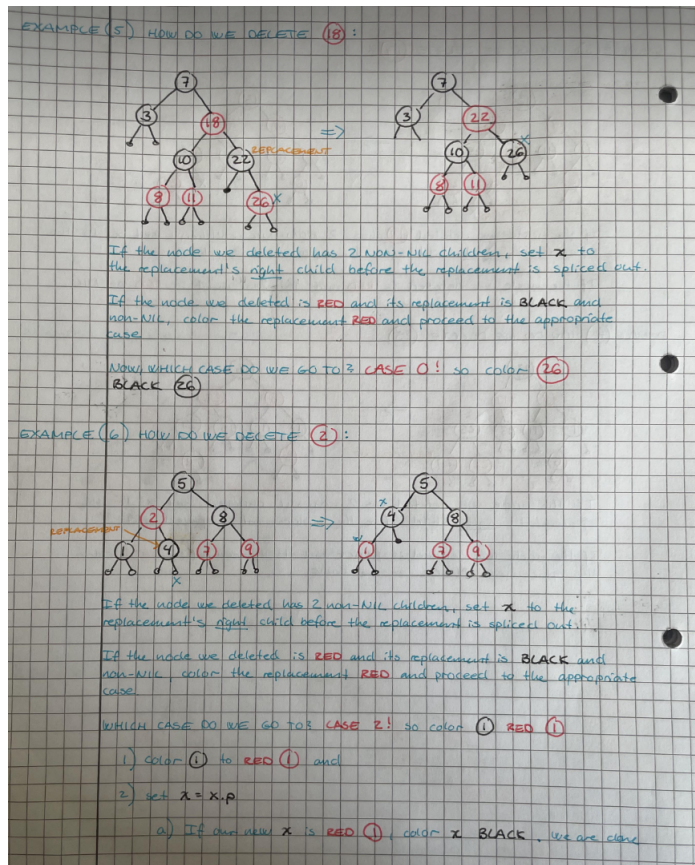
If the node we deleted (17) has 2 NON-NIL children, set x to the replacement's right child before the replacement is spliced out.

If the node we deleted is RED and its replacement is RED or NIL, we are DONE.

EXAMPLE (4) HOW DO WE DELETE (25):

If the node we deleted has 2 non-NIL children, set x to the replacement's right child before the replacement is spliced out.

If the node we deleted is BLACK and its replacement is RED, color the replacement BLACK. we are DONE.



6.6.1 Time complexity (RB trees)

Operation	Average Case
Search	$O(\log n)$
Insertion	$O(\log n)$
Deletion	$O(\log n)$

6.7 Fibonacci heaps

Supported operations:

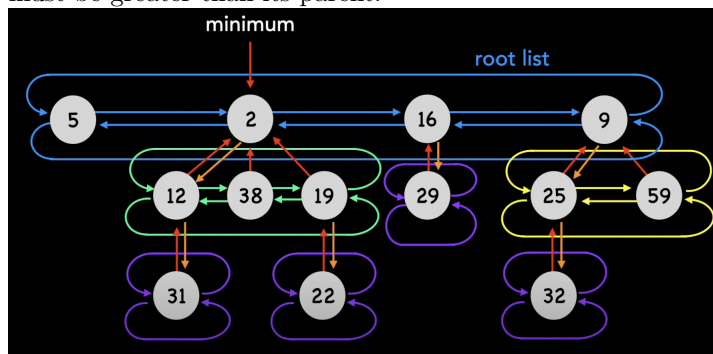
- Make-Heap(): returns a new empty heap

- Insert (H, x) : inserts element x into heap H
- Minimum (H) : returns pointer to element in H with min key
- Extract-Min (H) : deletes min-key element from H and returns pointer to it
- Union (H_1, H_2) : returns a new heap whose set of elements is the union of the sets of elements of heaps H_1 and H_2 ; H_1 and H_2 are destroyed in the process
- Decrease-Key (H, x, k) : decreases the key of element x in heap H to the new value k

Runningtimes binary vs fibonacci

Operation	Binary Heap (worst-case)	Fibonacci Heap (amortized)
make-heap	$\Theta(1)$	$\Theta(1)$
insert	$\Theta(\log n)$	$\Theta(1)$
minimum	$\Theta(1)$	$\Theta(1)$
extract-min	$\Theta(\log n)$	$O(\log n)$
union	$\Theta(n)$	$\Theta(1)$
decrease-key	$\Theta(\log n)$	$\Theta(1)$
delete	$\Theta(\log n)$	$O(\log n)$

Is essentially a collection of min heap ordered trees, meaning that each child must be greater than its parent.



From the graphic it is seen that siblings is in a double linked list, children is pointing to themselves and the roots is also in a double linked list, with a pointer to the minimum node.

N is the total nodes in the fibonacci Heap.

Degree is the number of children in a nodes childs list, så hvis man kigger på det træ der er ned ved 2, har den en degree på 3, fordi den har 3 børn(12, 38,19)

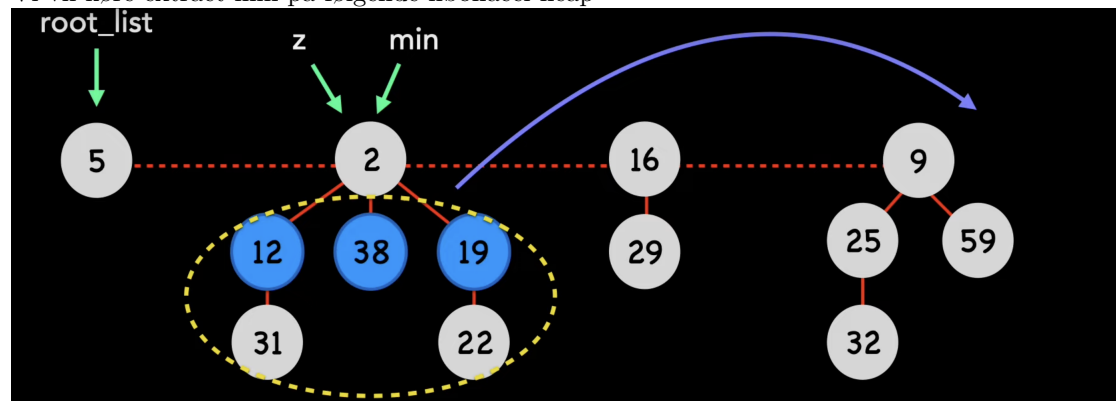
Marked indicates whether or not the node has lost a child since last it was made a child of another node

Det er muligt at tilføje en node, eller concatenate to lister i $O(1)$

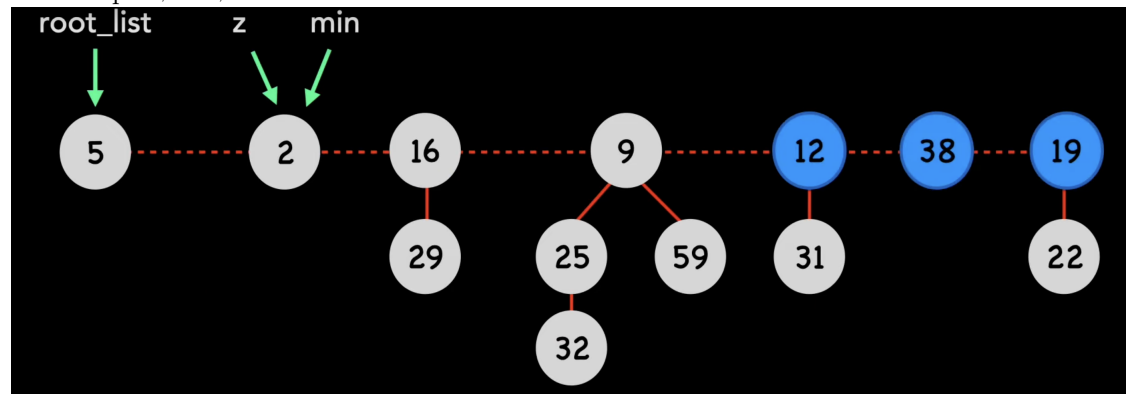
Der er ikke nogen specifik orden i hvilken række følge siblings står i (dvs. de grønne og gule i ovenstående grafik)

6.7.1 Extract min

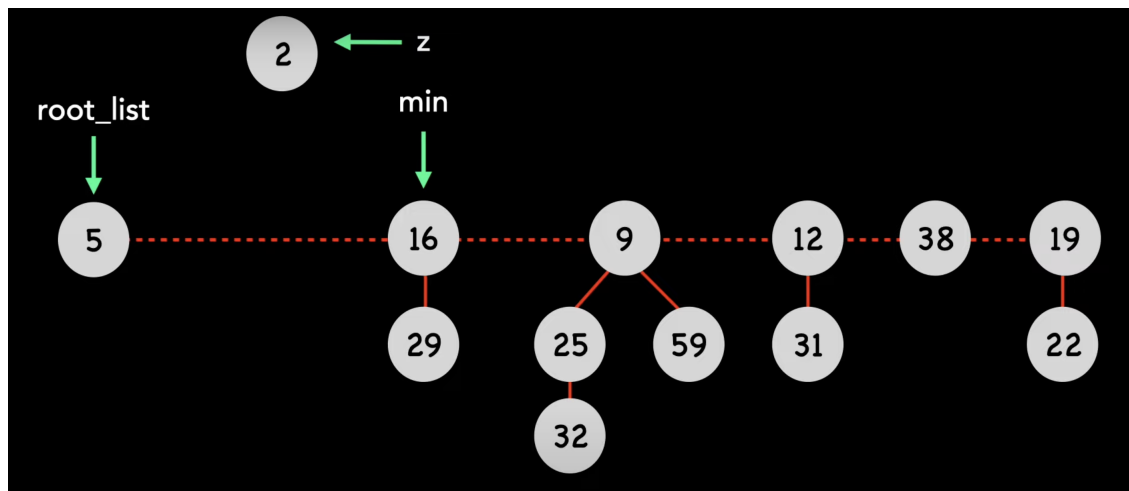
Vi vil køre extract min på følgende fibonacci heap



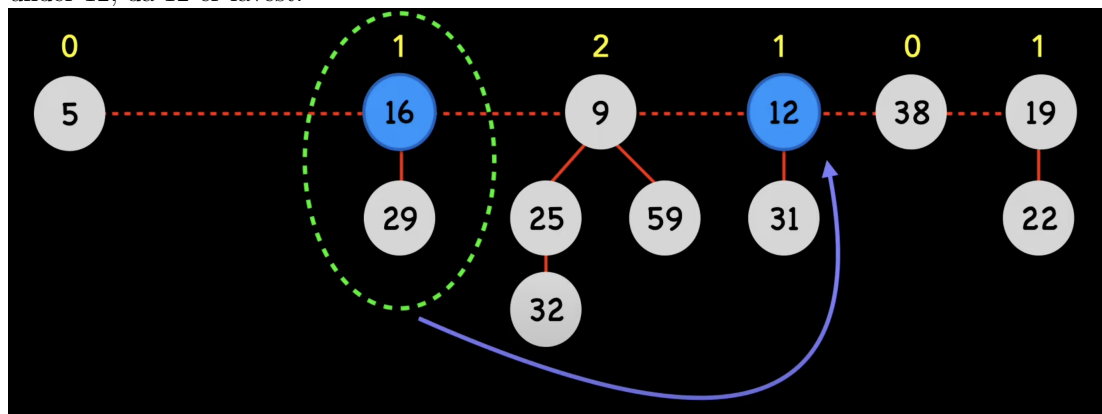
Min Heaps børn søttes ind som noder i root list



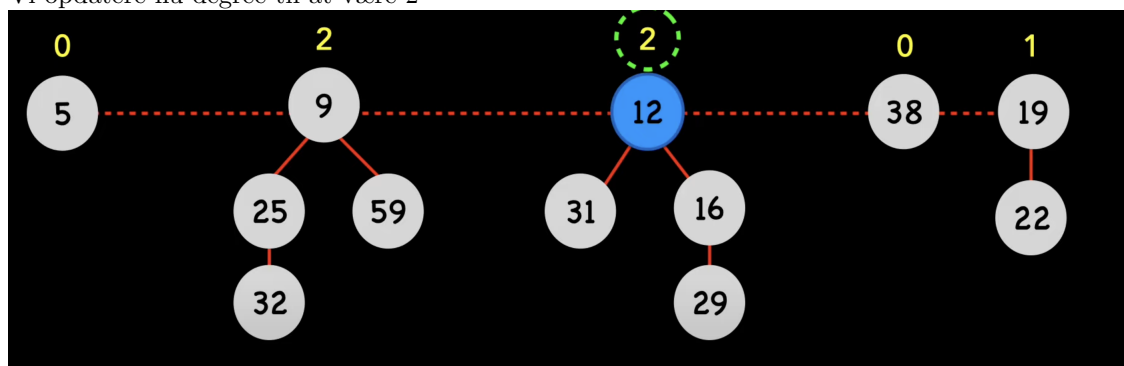
Vi fjerner den mindste node



Vi skal nu køre consolidate, dvs hver node i rootlist skal have en unik degree (dvs hvor mange børn de har), 16 og 12 har samme antal børn, derfor rykker vi 16 under 12, da 12 er lavest.

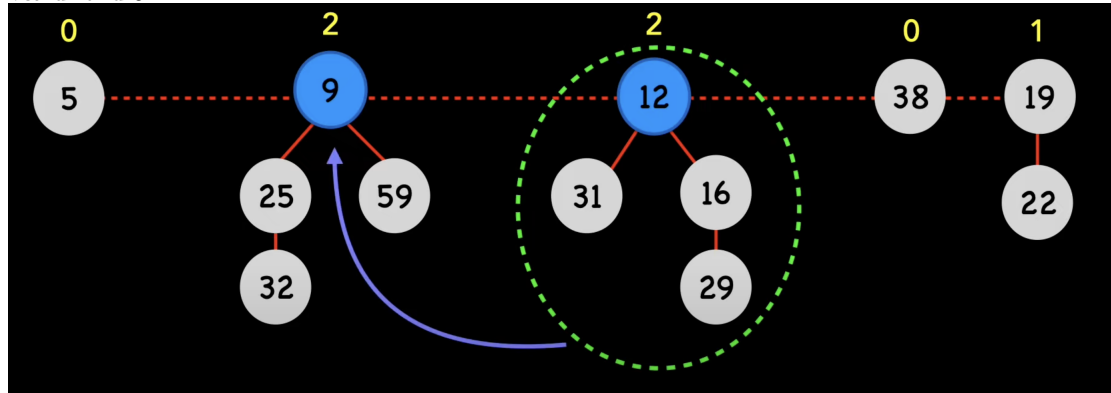


Vi opdatere nu degree til at være 2

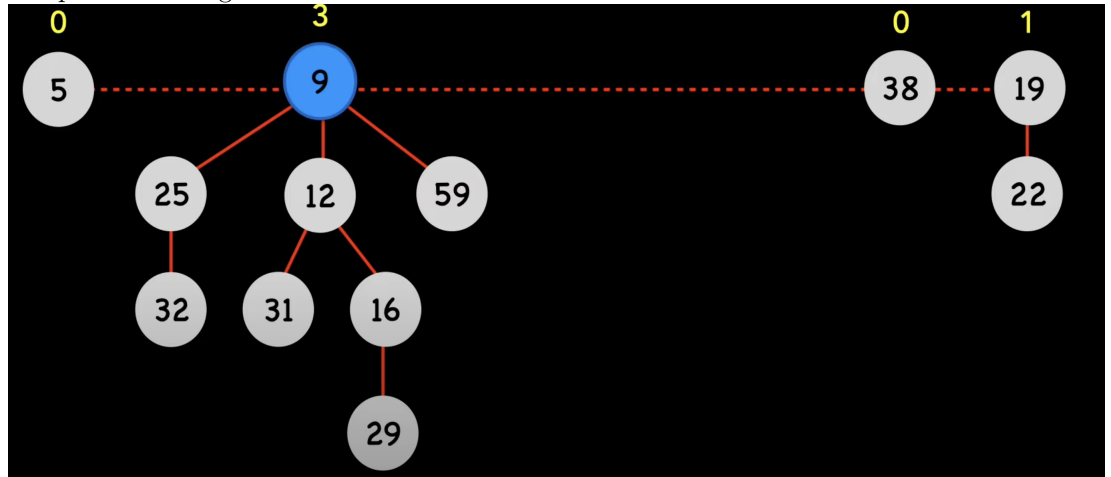


12 har nu samme degree som 9, og vi indsætter 12 under 9, da den har en højere

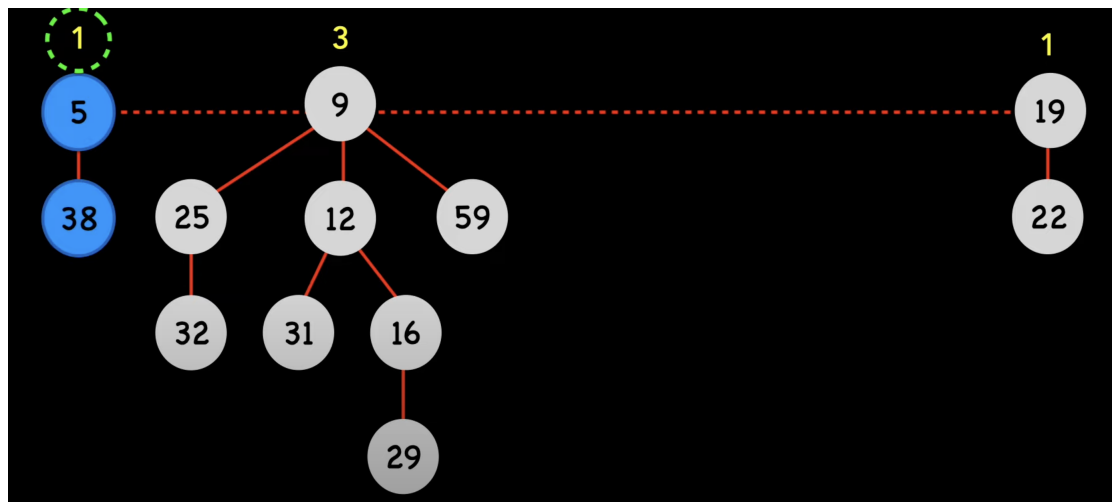
værdi end 9



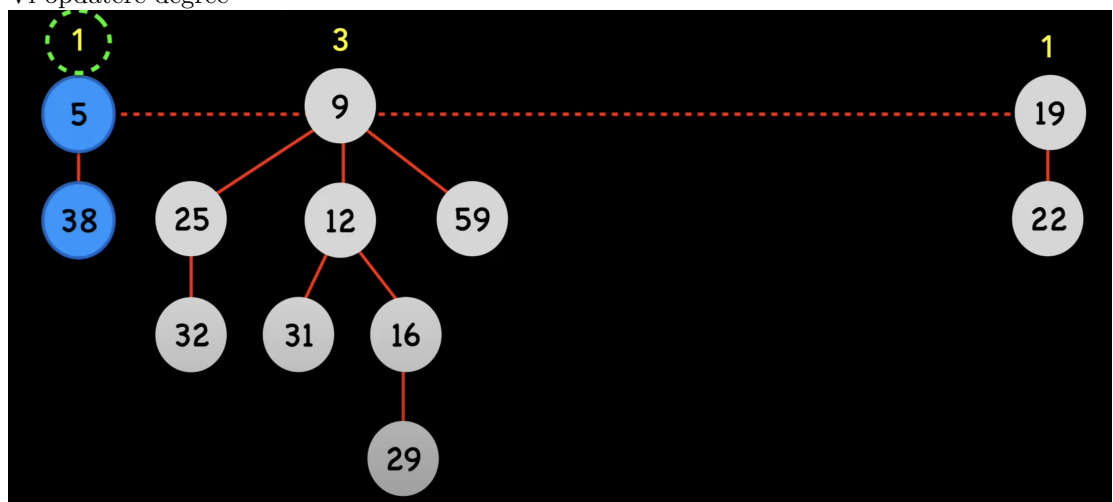
Vi opdatere nu degree for 9 til at være 3



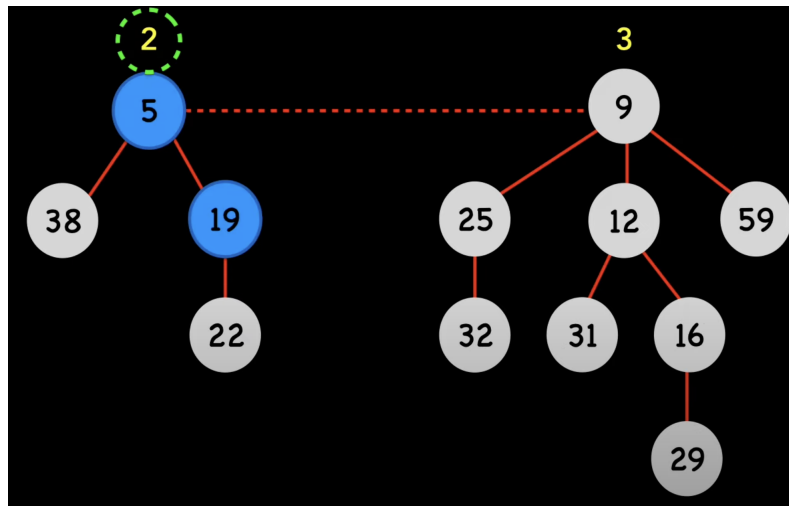
5 og 38 har samme degree, vi indsætter derfor 38 under 5



Vi opdatere degree

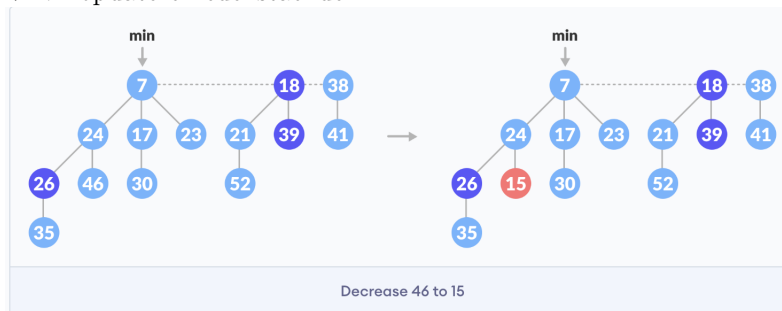


Vi indsætter nu 19 under 5 og opdatere degree

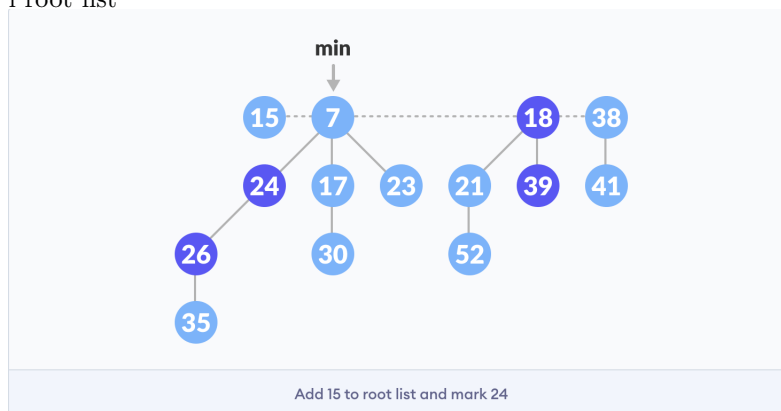


6.7.2 Decrease key

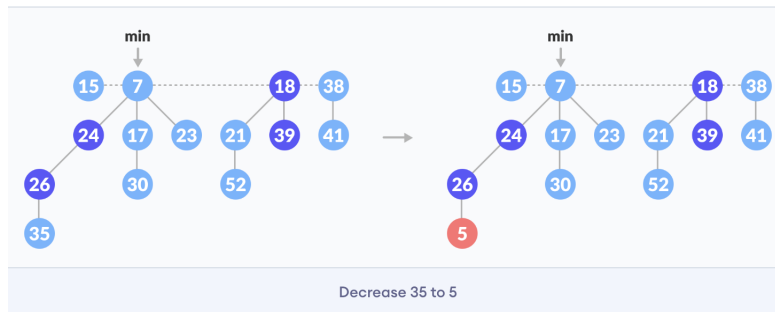
Vi vil opdatere nedenstående



Da 15 nu overtræder at childs aldrig må være mindre end root, rykker vi 15 op i root list



Vi opdatere nu 35, til at være 5



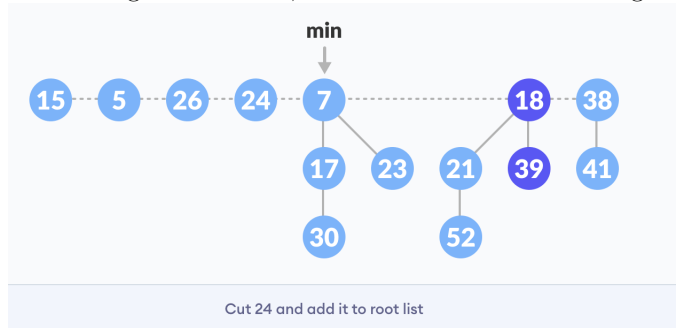
Dette overtræder også, og vi rykker 5 op i root list



fordi 26 er markeret, skal vi lave et cut, og rykke nummer to markeret knude op i root list og lave marked til false.



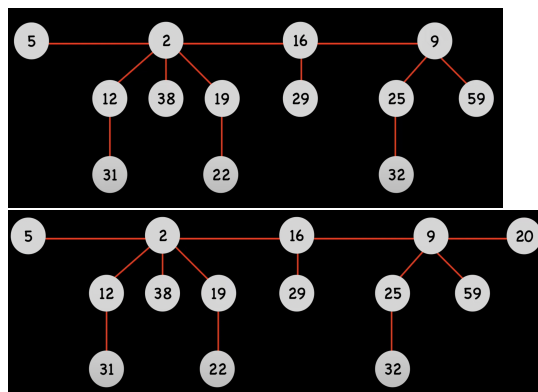
Fordi 24 også er marked, skal den addes til root list og markeres som falsk



Vi skal nu opdatere minimum



6.7.3 Insert

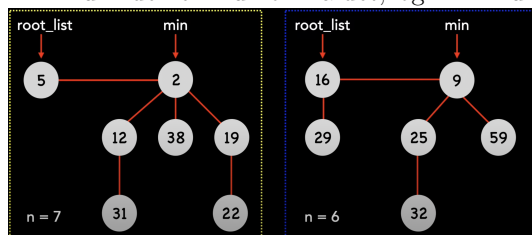


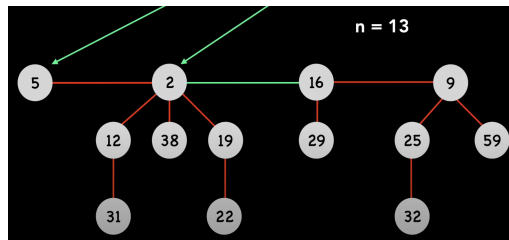
when inserting, the inserted node will be inserted into the root list. Det kan i illustrationen ses at 20 blot er indsat i rootlisten.

6.7.4 Union

When calling union the heaps will be put together into one heap. Minimum heap and n will be updated.

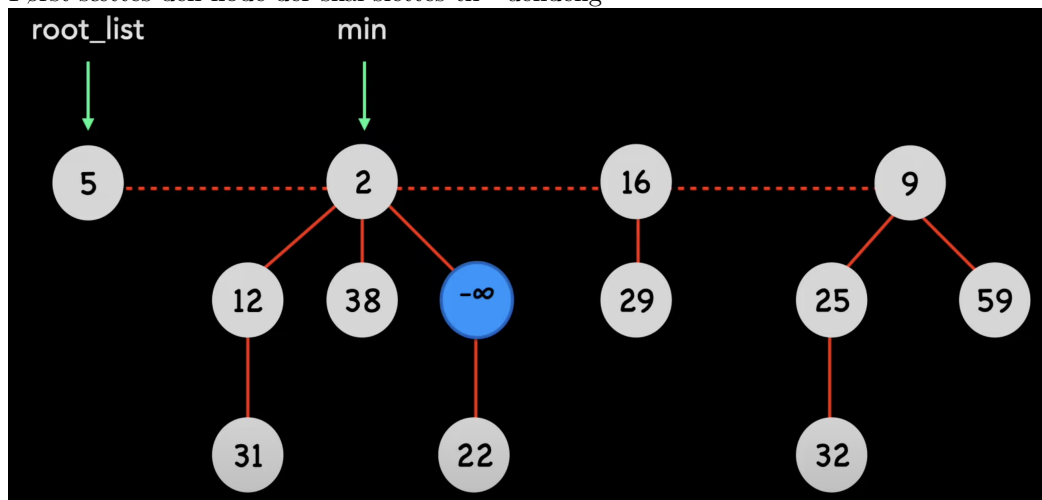
Root, vil være root list, i den første heap, vi ser derefter på minimum i de to minimum der er i union kaldet, og minimum sættes til den mindste af dem.



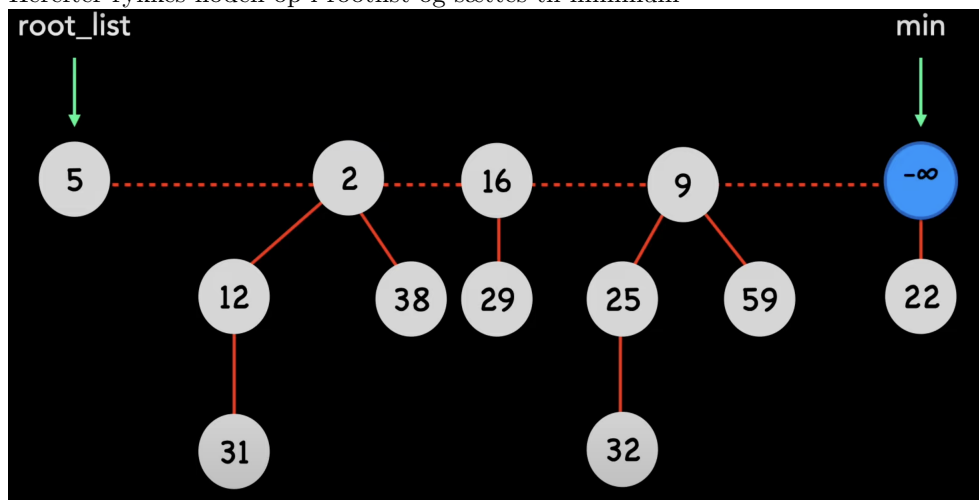


6.7.5 Delete

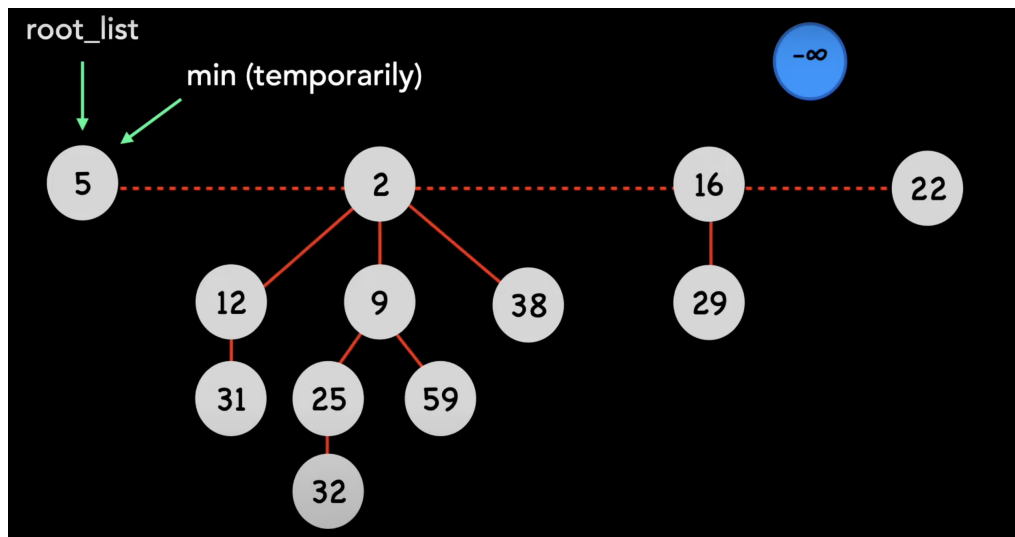
Først sættes den node der skal slettes til - uendelig



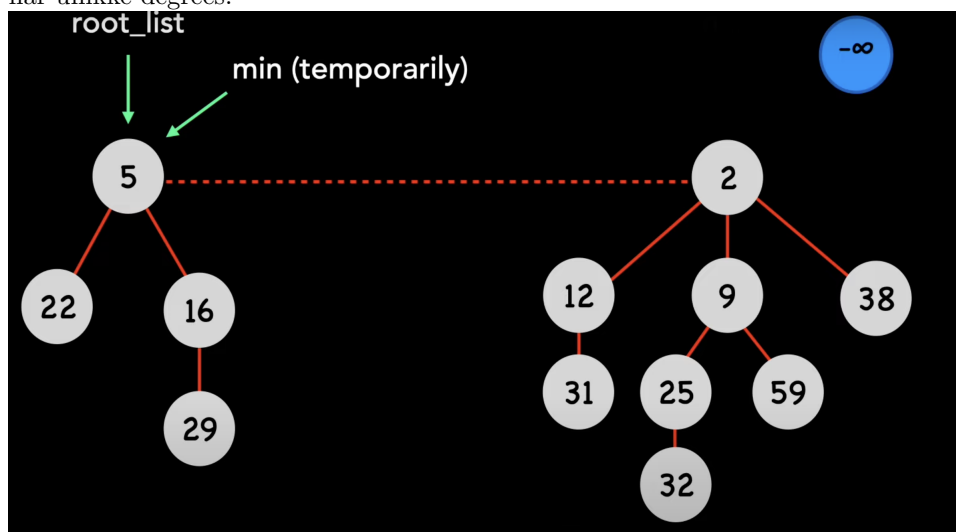
Herefter rykkes noden op i rootlist og sættes til minimum



Nodens fjernes, og dens børn sættes op i rootlist



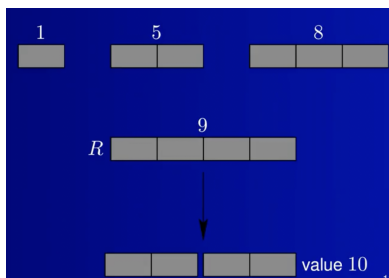
Der køres nu consolidate, ligesom i ekstract min, hvor der sikres at hver træ har unikke degrees.



7 Dynamic programming

For dynamic programming to be usefull we need overlapping subproblems(the same subproblems are visited repeatedly)

7.1 The rod cutting problem



We want to pick the maximum revenue, and the revenue if we don't cut the rod is 9, and if we have one block the revenue will be 1, two blocks will be 5 and 3 blocks will be 8. So the maximal revenue will be 10, if we split the rod into two pieces with a revenue of 5.

There will be $2^n - 1$ ways to cut the rod.

When expanding the problems into subproblems, they will be independent of one another. so the optimal solution will be the optimal solutions of the subproblems.

7.2 Top down memoization

Solves the problem from the top using recursion.

7.3 Bottom up approach

The idea is to sort the subproblems after increasing size. The subproblems will be solved in this order and stored.

7.4 The longest common subsequence problem

Y skal være øverst (vandret) og x skal være lodret. We consider 3 different cases:

- $i = 0$ or $j = 0$

$$c[i, j] = 0$$

- $i, j > 0$ and $x_i = y_j$

$$c[i, j] = c[i - 1, j - 1] + 1$$

- $i, j > 0$ and $x_i \neq y_j$

$$c[i, j] = \max\{c[i, j - 1], c[i - 1, j]\}$$

It can be visualized in the following way



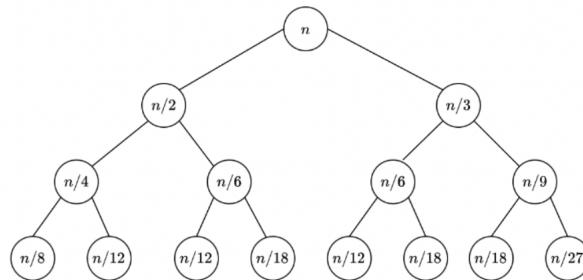
Where the first visualization shows how it is completed and the second shows the characters in the common subsequence.

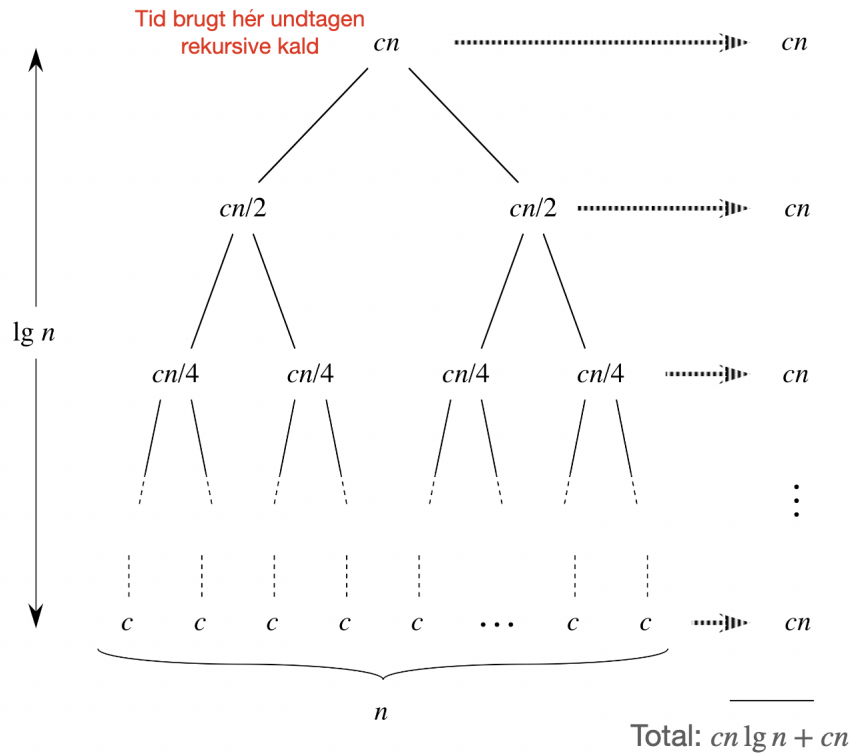
8 Divide and conquer

8.1 Recursion tree

Examples

1. $p(n) = p(n/2) + p(n/3) + n$





8.2 Master Method

It is given on the form $T(n) = aT(n/b) + \Theta(f(n))$

1. $f(n) = O(n^{\log_b(a) - \varepsilon}) \implies T(n) = \Theta(n^{\log_b a})$
2. $f(n) = \Theta(n^{\log_b(a)}) \implies T(n) = \Theta(n^{\log_b a} \log n)$
3. $f(n) = \Omega(n^{\log_b(a) + \varepsilon}) \implies T(n) = \Theta(f(n))$

How to find log

Lets say we have $\log_3(9)$, we can find the value as $3^x = 9 \implies x = 3$, derfor er svaret 3.

8.2.1 Example

If we have the following

$$p(n) = 8p(n/2) + n^2$$

we will find that $a = 8$, $b = 2$ and $f(n) = n^2$ we find $\log_2(8) = 3$. Therefore we have that $f(n) = O(n^{\log_b(a) - \varepsilon}) \implies p(n) = \Theta(n^{\log_b(a)})$. The upper bound will be given by $\Theta(n^3)$.

Asymptotisk notation

Hierarki over funktioner og deres "størrelse"

1. n^n
2. $n!$
3. $a^n \quad a \in [1; \infty)$
4. $n^a \quad a \in [1; \infty)$
5. $n \cdot \log(n)$
6. n
7. $\sqrt[n]{x} \sim x^a \quad a \in (0; 1)$
8. $\log(n)$

8.3 MC example

Antag at du har en rekursiv algoritme A , og lad n betegne størrelsen på algoritmens input X . Hvis $n = 1$ beregnes resultatet $A(X)$ i konstant tid. For $n > 1$ bruger algoritmen først $O(n \lg n)$ tid på at beregne fire inputs X_1, X_2, X_3, X_4 , hver af størrelse $\lfloor n/2 \rfloor$, og $A(X_1), A(X_2), A(X_3), A(X_4)$ beregnes rekursivt. Endelig kombineres de rekursive svar til et output $A(X)$ i tid $O(n^2)$.

Hvilken af disse rekursionsligninger beskriver køretiden $T(n)$ af A på input X ? Vælg præcis ét svar.

1. $T(n) = 4T(\lfloor n/2 \rfloor) + O(n \lg n)$.
2. $T(n) = 2T(\lfloor n/4 \rfloor) + O(n \lg n)$.
3. $T(n) = 4T(\lfloor n/2 \rfloor) + O(n^2)$.
4. $T(n) = 2T(\lfloor n/4 \rfloor) + O(n^2)$.

I dette tilfælde, kan vi se at der er 4 elementer, der hver har størrelse $\lfloor n/2 \rfloor$ og kombineres med køretid $O(n^2)$, derfor er svar 3 korrekt.

Mere uddybende fra chatgpt: I tilfældet med den rekursive algoritme A , som du først nævnte, er der fire underproblemer i hver rekursion ($a = 4$), og størrelsen af hvert underproblem er halvdelen af det oprindelige problem ($b = 2$), da problemet deles i fire dele af størrelsen $\lfloor n/2 \rfloor$. Kombinationstrinnet, som er omkostningen ved at sætte underproblemerne sammen igen, er givet til at være $O(n^2)$, hvilket betyder at $f(n) = O(n^2)$.

- $T(n)$ er den funktion, vi ønsker at finde tidskompleksiteten for.
- a er antallet af underproblemer i rekursionen.
- $\frac{n}{b}$ er størrelsen af hvert underproblem. (Her antager vi, at alle underproblemer er af samme størrelse).
- $f(n)$ er omkostningen ved at dele problemet op og kombinere resultaterne af underproblemerne.
- n er størrelsen på det oprindelige problem.

8.4 Substitution method

Technique for verifying a solution to a recursion equation. It will include the following

- Guess for a solution, ex $T(n) = O(f(n))$
- Assumption $p(n') \leq cn'$
- Show by induction that $T(n) \leq cf(n)$

Example

Recursion equation: $p(n) = p(n/2) + p(n/3) + n$

Guess: $O(n)$

Induction:

$$p(n) \leq p(n/2) + p(n/3) + n \quad (1)$$

$$\leq c(n/2) + c(n/3) + cn \quad (2)$$

$$\leq cn + cn + cn \quad (3)$$

$$= cn \quad (4)$$

8.4.1 Hint for Multiple choice

An easier way to see it in the multiple choice part is if have the following recursion formula

$$T(n) = T(n-1) + n^2$$

we can simply multiply the n in the paratheses and the $+n^2$, hence $O(n^3)$

9 Amortized costs

9.1 Aggregated analysis

Upperbound $T(n)$ on the worst case time of n operations, and then calculate the amortized cost as $T(n)/n$

9.2 Accounting method

We assign artificial cost to each operation: $\hat{c}_1 \dots \hat{c}_n$, where $\hat{c}_i$ is the cost of the i th operation

A requirement for any sequence of n is

$$\sum_{i=1}^n c_i \leq \sum_{i=1}^n \hat{c}_i$$

credit is saved at an object

9.3 The potential method

credit is stored in a "bank" (ϕ) rather than at the objects.

The amortized cost is defined as

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})$$

So the ammortized cost is given by the actual cost plus the change in credit we have in the bank from right before we make the i th operation to right after we make the i th operation.

- if $\Phi(D_i) - \Phi(D_{i-1}) > 0$ we overcharge and put $\Phi(D_i) - \Phi(D_{i-1})$ in the bank
- if $\Phi(D_i) - \Phi(D_{i-1}) < 0$ we undercharge and withdraw $\Phi(D_{i-1}) - \Phi(D_i) > 0$ from the bank to help pay the i th operation

9.3.1 example

The amortized cost for the operations will be: - push

$$\text{Potential change} = \phi D_i - \phi D_{(i-1)} = (s+1) - s = 1.$$

Amortized cost =

$$c'_i = c_i + \phi D_i - \phi D_{(i-1)} = 1 + 1 = 2.$$

- pop Potential change =

$$\phi D_i - \phi D_{(i-1)} = (s-1) - s = -1.$$

Amortized cost =

$$c'_i = c_i + \phi D_i - \phi D_{(i-1)} = 1 + (-1) = 0.$$

- multi-pop(s, k) = $k' = \min(s, k)$ Potential change = $\phi D_i - \phi D_{(i-1)} = -k$,
Amortized cost =

$$c'_i = c_i + \phi D_i - \phi D_{(i-1)} = k' + (-k') = 0.$$

Therefore, the amortized cost for each operation will be $O(1)$, and the total amortized cost for n operations will be $O(n)$.

9.4 Dynamiske tabeller potential method

Egenskaberne for potentialefunktionen i konteksten af dynamiske tabeller ved sletninger kunne være:

1. ****Ikke-negativ****: Potentialefunktionen er altid ikke-negativ.
2. ****Begynder ved nul****: Potentialefunktionen starter ved nul, når tabellen er tom.
3. ****Reflekterer tabellens tilstand****: Potentiallet stiger, når tabellen fyldes, og falder, når elementer fjernes.
4. ****Omkostninger ved sletning****: Når elementer slettes og tabellen bliver for løs (for eksempel når den kun er en fjerdedel fuld), afspejles dette ved en potentialeændring, som indikerer omkostningen ved at halvere tabellen.
5. ****Amortiseret omkostning****: Den amortiserede omkostning ved sletninger bliver konstant, fordi potentialefaldet (ved sletning) kan betale for den faktiske omkostning ved at flytte elementerne til en mindre tabel.

9.4.1 køretider

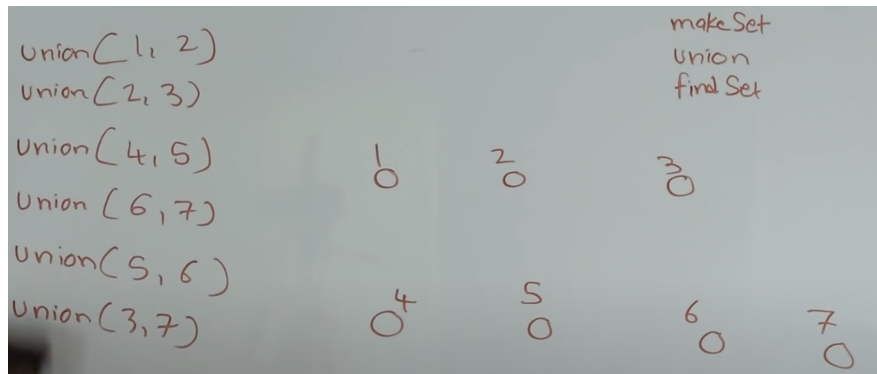
- **Indsættelse (Insert)**: Amortiseret køretid er konstant, $O(1)$. Dette er på grund af at selvom en indsættelse, som kræver en tabeludvidelse, kan tage $O(n)$ tid, bliver denne omkostning amortiseret over mange tidligere operationer, hvilket fører til en konstant amortiseret køretid.
- **Sletning (Delete)**: Ligesom ved indsættelse er den amortiserede køretid konstant, $O(1)$. Enkelte sletninger, som fører til en tabelhalvering, kan også tage $O(n)$ tid, men igen er denne omkostning amortiseret over mange operationer.
- **Søgning (Search)**: I gennemsnit er søgningen konstant, $O(1)$, for en velimplementeret hashtabel. I værste tilfælde, afhængigt af hashfunktionen og håndtering af kollisioner, kan søgningen tage $O(n)$ tid.
- **Udvidelse (Table Doubling)**: Den reelle køretid er lineær, $O(n)$, men den amortiserede køretid er konstant, $O(1)$, da operationen kun sker når antallet af elementer overstiger tabellens kapacitet.
- **Halvering (Table Halving)**: Ligesom med udvidelsen er den reelle køretid lineær, $O(n)$, men den amortiserede køretid forbliver konstant, $O(1)$, fordi omkostningen fordeles over mange operationer.

10 Disjoint sets

Datastructure supporting the following:

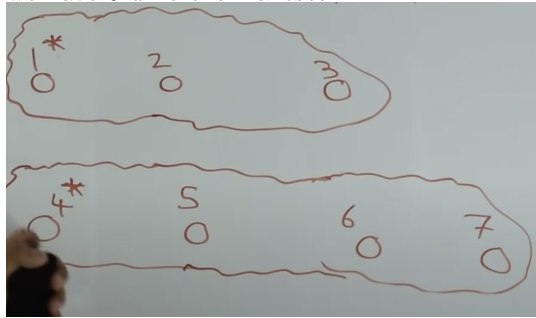
- `makeSet(x)`: creation of a set x with one element
- `find(x)`: returns canonical element of the set containing x
- `link(x,y)`: forms a union of two sets with x and y as their canonical elements

10.1 Example - simple



In this

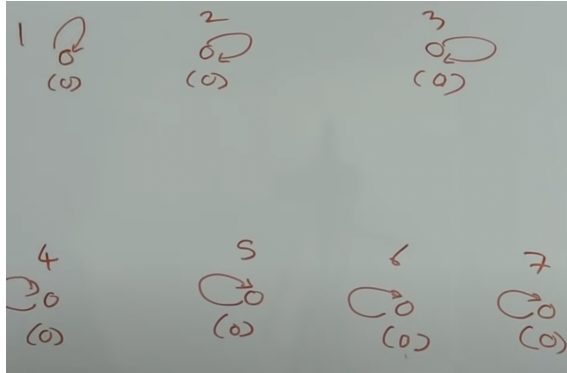
we have 6 different makeset.



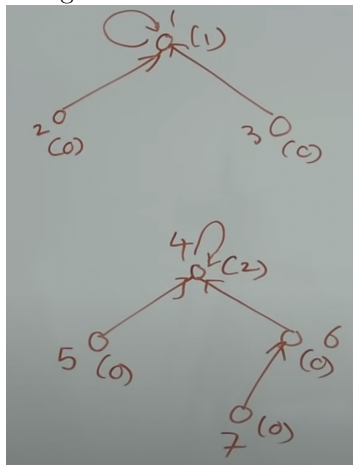
Here some union has been called, and when we call findset, we return the number with the star.

10.2 Example - elaborated

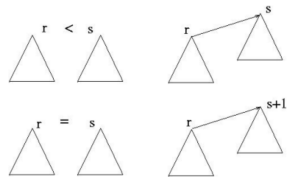
When using path compression you also look at the rank. It is illustrated below



It is here seen that they all start with rank 0. In the following we see how it changes when we call union and how the rank changes.



In general when you link by rank it follows the illustration below



10.3 Runningtimes

	Makeset	Find	Link	Total
Vector	$O(1)$	$O(1)$	$O(n)$	$O(m+n \log n)$
Tree	$O(1)$	$O(n)$	$O(1)$	$O(mn)$
Link by rank	$O(1)$	$O(\log n)$	$O(1)$	$O(n+m \log n)$
Compression	$O(1)$	$O(\log n)$	$O(1)$	$O((m+n) \log^* n)$

1. Tiden for et kald til UNION (x, y) er $O(\log n)$. - Korrekt. Med både union by rank og path compression heuristikker, vil tiden for at forene to træer være begrænset af træernes dybde, hvilket er logaritmisk i værste fald.
2. k kald til FIND-SET(42) lige efter hinanden tager tid $O(k + \log n)$. - Ikke korrekt. Efter det første kald af FIND-SET, vil path compression minimere træets højde, og alle efterfølgende kald vil tage konstant tid. Så k på hinanden følgende kald ville tage $O(k)$ tid.
3. Kaldene MAKE-SET(x_1), ..., MAKE-SET(x_k) tager total tid $O(k)$. - Korrekt. Hvert MAKE-SET operation tager konstant tid, så k af dem ville tage $O(k)$ tid.
4. En LINK(x, y) operation gør den maksimale værdi af rank i træerne for x og y med 1. - Ikke altid korrekt. LINK-operationen øger kun ranken, hvis træerne har samme rank; ellers hægtes det træ med lavere rank under det træ med højere rank, og ranken for det træ, der bliver roden, ændres ikke.
5. I værste fald tager en FIND-SET operation $\Omega(n)$ tid. - Ikke korrekt. Med path compression og union by rank heuristikker, selv i værste fald, vil FIND-SET ikke tage lineær tid. Det vil i stedet tage en tid, der er næsten konstant, amortiseret over en sekvens af operationer.
6. UNION (x, x) tager konstant tid. - Korrekt. UNION-operationen mellem et element og sig selv vil hurtigt tjekke, at begge elementer er i samme sæt, hvilket ikke kræver nogen ændringer i datastrukturen og derfor tager konstant tid

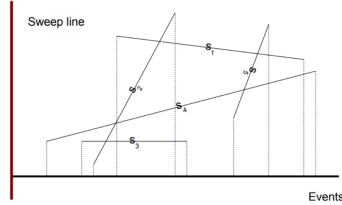
11 Planesweep

Necessary condition for the two lines to intersect: must be next to each other on some vertical line.

Assumptions:

- no vertical segments
- No segments overlap
- No three segments meet at the same point

The algorithm will do as illustrated below



Here we have a event line and a sweep line. The sweep line will stop at some points, where it will identify and update intersections. The stoppoints will be where the lines starts and ends. The algorithm will look from top to bottom (look at which lines is at the top) and it will go from left to right.

When arriving at the left endpoint of a line s , the algorithm will do as follows:

- Insert s to the correct spot on the sweepline
- check if s intersects the elemt above, if so add the intersect to the events
- Check if s intersects the element below, if so add the intersect to the events

When arriving at the right endpoint of line s , the algorithm will do as follows:

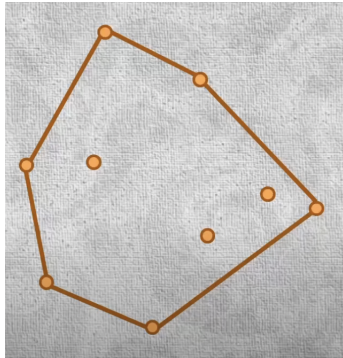
- Check if the element above and below s intersects to the right of s . If so the element was not reported earlier and should be added to the events
- Remove s from the status

When arriving at a intersection between two line segments, the algorithm will do as follows:

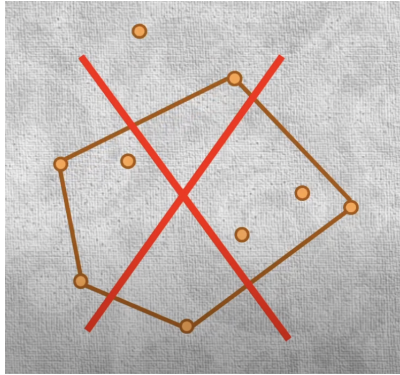
- swap the two lines in the status
- check if s_2 and the element above it intersects to the right of the intersection point. If so and it was not added earlier add it to events
- check if s_1 and the element below it intersects to the right of the intersection point. If so and it was not added earlier add it to events

12 Convex hulls

Given a set of points a convex hull is a polygon hat encloses all of those points. The vertices maximizes the area, while minimizing the cirkumference. An example could be the illustration below.

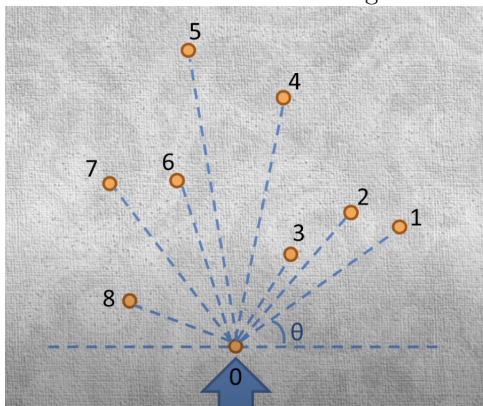


If some of the points is omitted it will not be a convex hull



12.1 Graham scan algorithm

Selects the point with the lowest y coordinate, and sorts the points by the angle relative to the bottom most angle and the horizontal.



It iterates in sorted order over the points and add the to the stack if they make a clockwise turn relative to the previous to points on the stack. If making a clockwise turn relative to the 2 previous turns, it will pop off the previous point.

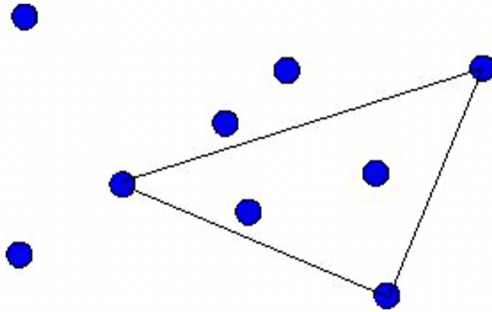
12.2 Jarvis march

Selects lowest point to be the first vertex, and then it selects the point with smallest counterclockwise relative to the previous elements. This is repeated until the starting vertex is reached again.

The overall runningtime is $O(nh)$, where h is the number of vertices.

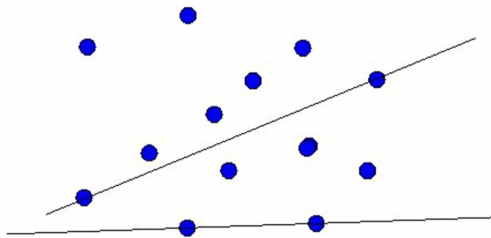
12.3 Point pruning

Make a triangle from three points, if any points is within the triangle it is not a part of the convex hull. This keeps on until to triangle has points within.



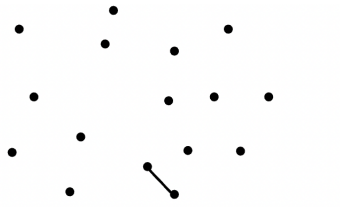
12.4 Edge pruning

Make a line between two points. If it has point on both sides it cannot be a corner. If it only has points on one side it is a corner. This is repeated until a convex hull is formed.



13 Closest pair problem

When given n points we want to find the closest pair. The naive way would be to pick a point and find all lengths to all other points.

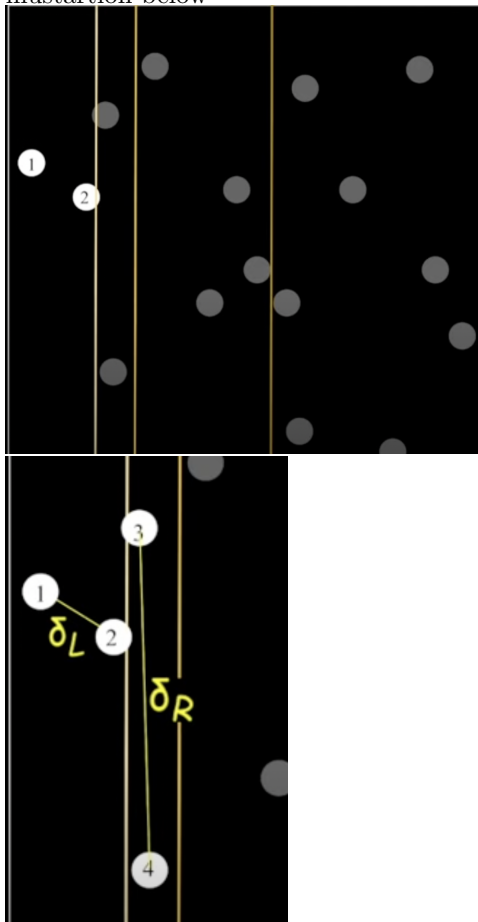


This can be improved by doing it in 1-dimension(a line) and sort them by distance

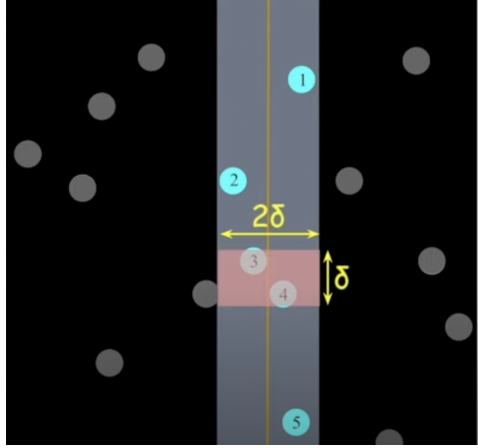


13.1 Closest pair problem - divide and conquer

When having points inside a plane, we can use divide and conquer to find the shortest distance. We call it recursively until there is only two points left. see illustration below



We define δ as the shortest distance so far. This is not considering one point from one pair and another point from the other pair. To to this we must do as illustrated below



14 LSM Træer

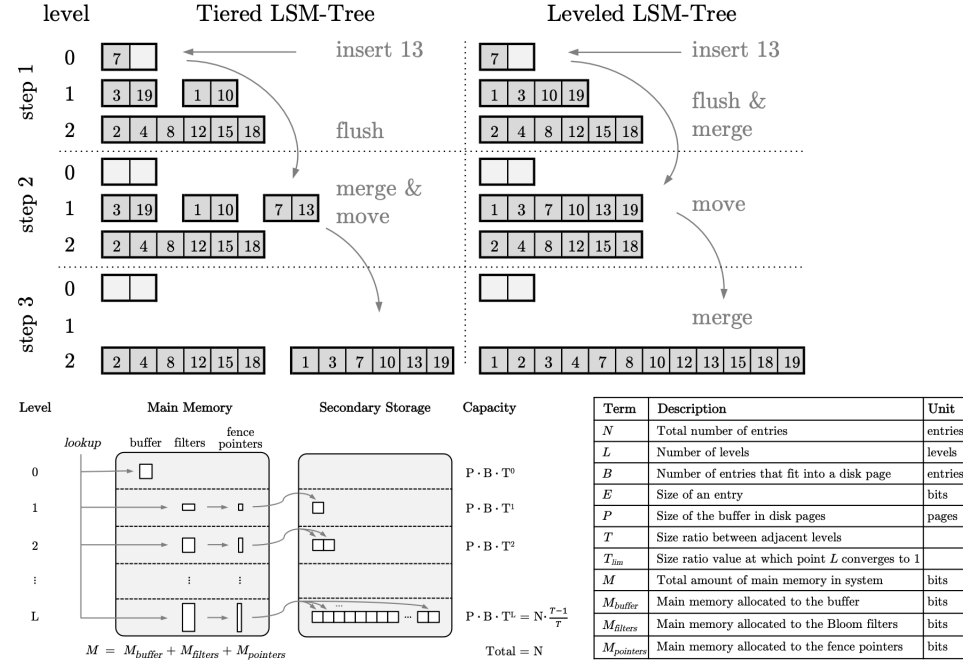


Figure 2: Overview of an LSM-tree and list of terms used throughout the paper.

14.1 Eksempel

L_0 :

42

L_1 :

--	--

L_2 :

17	21	68	99
----	----	----	----

L_3 :

1	3	5	27	36	49	51	82
---	---	---	----	----	----	----	----

9 skal indsættes på L_0 , men da dette er fyldt, skal det rykkes ned på næste niveau L_1 og flettes med elementet på L_0 . Dette opfylder $L_i = 0$ og 2^i , derfor vil det se sådan ud når 9 tilføjes

L_0 :

--

L_1 :

9	42
---	----

L_2 :

17	21	68	99
----	----	----	----

L_3 :

1	3	5	27	36	49	51	82
---	---	---	----	----	----	----	----

Indsættelse af 81, skal ind på L_0 , da dette er tomt, og opfylder $L_i = 0$ og 2^i

L_0 :

81

L_1 :

9	42
---	----

L_2 :

17	21	68	99
----	----	----	----

L_3 :

1	3	5	27	36	49	51	82
---	---	---	----	----	----	----	----

Ved indsættelse af 49, skal den indsættes på L_0 , da denne er fyldt skal den flettes med L_1 , vi vil derfor finde

L_0 :

--

L_1 :

9	42	49	81
---	----	----	----

L_2 :

17	21	68	99
----	----	----	----

L_3 :

1	3	5	27	36	49	51	82
---	---	---	----	----	----	----	----

Dette overskrider $L_i = 0$ og 2^i , derfor skal Det nu flettes sammen og rykkes ned til næste niveau. Vi finder

L_0 :

--

L_1 :

--	--

L_2 :

17	21	68	99
----	----	----	----

9	42	49	81
---	----	----	----

L_3 :

1	3	5	27	36	49	51	82
---	---	---	----	----	----	----	----

Dette overskrider igen $L_i = 0$ og 2^i , og vi fletter elementerne og rykker et niveau ned.

L_0 :

--

L_1 :

--	--

L_2 :

--	--	--	--

L_3 :

1	3	5	27	36	49	51	82
---	---	---	----	----	----	----	----

9	17	21	42	49	68	81	99
---	----	----	----	----	----	----	----

Dette overskrider igen $L_i = 0$ og 2^i , vi bliver derfor nødt til at oprette et nyt niveau, og finder altså

L_0 :

--

L_1 :

--	--

L_2 :

--	--	--	--

L_3 :

--	--	--	--	--	--	--	--

L_4 :

1	3	5	9	17	21	27	36	42	49	49	51	68	81	82	99
---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----

15 Maximum Sub-array

Maximum subarray ligger i en af følgende:

- entirely in the subarray $A[low \dots mid]$, so that $low \leq i \leq j \leq mid$,
- entirely in the subarray $A[mid + 1 \dots high]$, so that $mid < i \leq j \leq high$, or
- crossing the midpoint, so that $low \leq i \leq mid < j \leq high$.

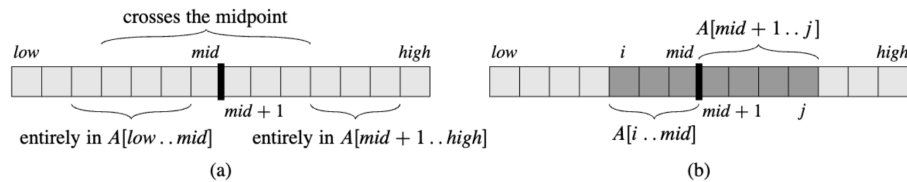


Figure 4.4 (a) Possible locations of subarrays of $A[low \dots high]$: entirely in $A[low \dots mid]$, entirely in $A[mid + 1 \dots high]$, or crossing the midpoint mid . (b) Any subarray of $A[low \dots high]$ crossing the midpoint comprises two subarrays $A[i \dots mid]$ and $A[mid + 1 \dots j]$, where $low \leq i \leq mid$ and $mid < j \leq high$.

1. Divide the given array in two halves
2. Return the maximum of following three
 - Maximum subarray sum in left half (Make a recursive call)
 - Maximum subarray sum in right half (Make a recursive call)
 - Maximum subarray sum such that the subarray crosses the midpoint

The lines 2.a and 2.b are simple recursive calls. How to find maximum subarray sum such that the subarray crosses the midpoint? We can easily find the crossing sum in linear time. The idea is simple, find the maximum sum starting from mid point and ending at some point on left of mid , then find the maximum sum starting from $mid + 1$ and ending with some point on right of $mid + 1$. Finally, combine the two and return the maximum among left, right and combination of both.

Input : -2, 1, -3, 4, -1, 2, 1, -5, 4

Left Sum =4, Right Sum =4,
Cross Sum = ??

-2	1	-3	4	-1	2	1	-5	4
----	---	----	---	----	---	---	----	---

Cross Sum Calculation :
 Pivot Element: -1
Left Sub Sum = Max (-1, -1+4, -1+4-3, -1+4-3+1, -1+4-3+1-2)
 = Max (-1, 3, 0, 1, -2)
 = 3
Right Sub Sum = Max (2, 2+1, 2+1-5, 2+1-5+4)
 = Max (2, 3, -2, 2)
 = 3
 Cross Sum = Left Sub Sum + Right Sub Sum = (3+3) = 6
Cross Sum = 6

Maximum Sum = Max (Left Sum, Right Sum, Cross Sum) = 6

15.1 Eksempel

-2	1	-3	4	-1	2	1	-5	4
----	---	----	---	----	---	---	----	---

For at finde max array i dette eksempel, kan man først kigge på -2 og 1, hvilket giver -1. Vi glemmer derfor -2 og kigger derefter på 1 og -3, dette giver -2, vi glemmer derfor 1 og -3, og kigger nu på 4 og -1, hvilket giver 2. Herefter inkludere vi nu næste element(2), dette inkludere vi da det giver en større sum og det samme gør elementet efter(1).

16 Karatsuba

The image shows a handwritten solution for multiplying 1234 by 5678 using the Karatsuba algorithm. The numbers are split into two parts: 12 (A) and 34 (B) for 1234, and 56 (C) and 78 (D) for 5678. The steps are as follows:

KARATSUBA

$X = \overset{a}{12} \overset{b}{34}$ $Y = \overset{c}{56} \overset{d}{78}$

4 MAIN STEPS:

STEP 1: compute $a \cdot c = 12 \cdot 56 = \underline{672}$

STEP 2: compute $b \cdot d = 34 \cdot 78 = \underline{2652}$

STEP 3: compute $(a + b)(c + d) = (12 + 34)(56 + 78)$
 $= (46)(134)$
 $= \underline{6164}$

STEP 4: compute $(3) - (2) - (1) = 6164 - 2652 - 672$
 $= \underline{2840}$

$6720000 + 284000 + 2652 = \underline{7006652}$

FINAL ANSWER

Step 1: Split the Numbers

We split each number into two parts:

- For 1234, we split it into 12 (A) and 34 (B).
- For 5678, we split it into 56 (C) and 78 (D).

Step 2: Perform Three Multiplications

- Multiply A and C: $12 \times 56 = 672$ (let's call this AC)
- Multiply B and D: $34 \times 78 = 2652$ (BD)
- Multiply (A + B) and (C + D): $(12 + 34) \times (56 + 78) = 46 \times 134 = 6164$ (let's call this E)

Step 3: Calculate the Missing Products

- Calculate $E - AC - BD = 6164 - 672 - 2652 = 2840$

This gives us the sum of the products of A times D and B times C.

Step 4: Recombine the Results

- The final result of the multiplication is not just a direct sum of the three products we've calculated. We need to place them correctly, taking into ac-

count their place value: We have AC , which is the result of multiplying the first halves of the numbers (the most significant parts), so it needs to be shifted to the left by the length of both parts B and D in decimal places (which is 4 places in this example). This gives us $672 \times 10^4 = 6720000$.

Then, we have BD , which doesn't need to be shifted because it's the product of the least significant parts: 2652 .

The middle part is the sum of the products of A times D and B times C , and it needs to be shifted to the left by the length of either part B or D (2 places in this example): $2840 \times 10^2 = 284000$.

- Now we add up these three numbers:

$$6720000 + 284000 + 2652 = 7006652.$$

So, by using the Karatsuba algorithm, we've found that 1234 multiplied by 5678 equals 7006652 , and we did it by doing only three multiplications instead of four and then adding the results together with the proper shifts for place values.

17 Matrix chain multiplication

Hvis vi har to nedenstående matricer $A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$ $B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$

Kan det ses at A er 2×3 og B er 3×2 , betingelsen for at de kan multipliceres er at de har samme dimensioner (antal kolonner i den første og antal rækker i den anden skal være de samme), dvs. i vores eksempel 3, og 3.

når vi ganger de to matricer finder man

$$C = A \times B = \begin{bmatrix} a_{11} \times b_{11} + a_{12} \times b_{21} + a_{13} \times b_{31} & a_{11} \times b_{12} + a_{12} \times b_{22} + a_{13} \times b_{32} \\ a_{21} \times b_{11} + a_{22} \times b_{21} + a_{23} \times b_{31} & a_{21} \times b_{12} + a_{22} \times b_{22} + a_{23} \times b_{32} \end{bmatrix}$$

17.1 Varianser af MCM

17.1.1 Palindrome partitioning

Though not a direct variant, this problem shares a similar structure with MCM. Given a string, the task is to split it into segments such that each segment is a palindrome, minimizing the number of cuts. Dynamic programming is used to find the optimal splits.

Recursion Equation:

$$DP[i] = \min_{0 \leq j < i} \{ DP[j] + 1 \text{ if substring}(j+1, i) \text{ is palindrome} \}$$

```
function MinPalindromeCuts(s):
```

```
    n = længden af s
```

```
    DP = array af størrelse n, fyldt med 0
```

```

palindrome = 2D-array af størrelse n x n, fyldt med falsk
// Forudberegner palindrom-tabel
for i fra 0 til n-1:
    for j fra 0 til i:
        hvis s[i] == s[j] og (i-j <= 2 eller palindrome[j+1][i-1]):
            palindrome[j][i] = sandt
for i fra 1 til n:
    minCuts = i
    for j fra 0 til i:
        hvis palindrome[j][i-1]:
            minCuts = j == 0 ? 0 : min(minCuts, DP[j-1] + 1)
    DP[i-1] = minCuts
return DP[n-1]

```

Time Complexity: $O(n^2)$, where n is the length of the string. This is because we need to check every substring to see if it is a palindrome (which can be done in $O(n^2)$ time using DP), and then we make a cut at the optimal point.

17.1.2 Optimal binary search tree

Given a set of keys with their probabilities of being searched for, the task is to construct a binary search tree with the minimum expected search cost. Like MCM, OBST is solved using dynamic programming by finding the optimal subtree structure.

Recursion Equation:

$$DP[i][j] = \min_{i \leq k \leq j} \left\{ DP[i][k-1] + DP[k+1][j] + \sum_{m=i}^j \text{freq}[m] \right\}$$

Pseudokode:

```

function OptimalBST(keys, freq, n):
    // keys[] -- Array of keys
    // freq[] -- Frequency of keys
    // n -- Number of keys
    DP = 2D-array of size n x n, filled with 0
    for i from 0 to n-1:
        DP[i][i] = freq[i]
    for L from 2 to n: // L is the chain length
        for i from 0 to n-L+1:
            j = i + L - 1
            DP[i][j] = Infinity
            totalFreq = sum(freq, i, j)
            for r from i to j:
                cost = ((DP[i][r-1] if r > i else 0) +
                        (DP[r+1][j] if r < j else 0) +

```

```

        totalFreq)
    if cost < DP[i][j]:
        DP[i][j] = cost
    return DP[0][n-1]

```

Time Complexity: $O(n^3)$, where n is the number of keys, because we evaluate each possible root for every subtree.

17.1.3 Minimum/Maximum Cost to Multiply Chains of Matrices

Beyond just finding the minimum number of scalar multiplications, variations might ask for minimizing or maximizing other cost functions associated with matrix multiplication, such as actual computational cost based on matrix dimensions beyond simple scalar counts.

Recursion Equation:

$$DP[i][j] = \min_{i \leq k < j} \{DP[i][k] + DP[k+1][j] + \text{cost}(i, k, j)\}$$

Pseudocode:

```

function MinMaxMatrixChainMultiplication(matrices):
    // matrices is an array of matrix dimensions
    n = length(matrices)
    DP = 2D-array of size n x n, filled with Infinity (for min) or 0 (for max)
    for i from 1 to n:
        DP[i][i] = 0
    for L from 2 to n:
        for i from 1 to n-L+1:
            j = i + L - 1
            for k from i to j-1:
                cost = DP[i][k] + DP[k+1][j] + calculationBasedOnCost(i, k, j)
                DP[i][j] = updateCost(DP[i][j], cost) // updateCost depends
                on min or max requirement
    return DP[1][n]

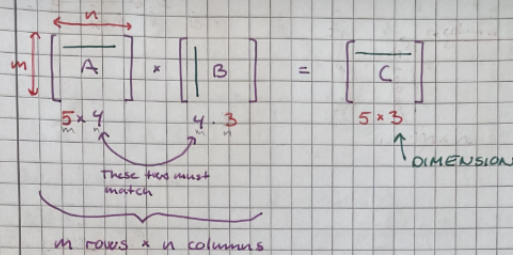
```

Time Complexity: $O(n^3)$, assuming the cost function can be computed in constant time for each split.

MATRIX CHAIN MULTIPLICATION - DYNAMIC PROGRAMMING

$$A_1 \cdot A_2 \cdot A_3 \cdot A_4$$

5x4 4x6 6x2 2x7



MATRIX MULTIPLICATION is not about to multiply them, The problem is to KNOW how to multiply them

$$\begin{matrix} A_1 & \times & A_2 & \times & A_3 \\ \underline{2} & \underline{3} & \underline{3} & \underline{4} & \underline{2} \\ d_0 & d_1 & d_1 & d_2 & d_3 \end{matrix} \} \text{dimensions}$$

METHOD 1:

METHOD 2:

$$\left\{ \begin{array}{l} (A_1 \times A_2) \times A_3 \\ A_1 \times (A_2 \times A_3) \end{array} \right\} \text{2 methods}$$

HOW MANY MULTIPLICATIONS DO I HAVE TO PERFORM?

2 * 3 * 4 = 24

NO MULTIPLICATION IS REQUIRED HERE

$(A_2 \times A_3) = 3 \cdot 4 \cdot 2 = 24$

$A_1 = 0$ not required

How many multiplications does the whole expression need?

DIMENSIONS: $(A_1 \times A_2) \times A_3$

$\underline{2} \cdot \underline{3} \cdot \underline{4} \cdot \underline{2}$

$2 \cdot 4 \cdot 2 = 16$ multiplications

Now all together:

$24 + 0 + 2 \cdot 4 \cdot 2 = 40$

So 40 multiplications are required for METHOD 1

$C[i, k] + C[k+1, j] + d_{i-1} \cdot d_k \cdot d_j = 40$

FORMULA: $C[i, j] = \min_{i \leq k < j} \{ C[i, k] + C[k+1, j] + d_{i-1} \cdot d_k \cdot d_j \}$

How many multiplications does the whole expression need?

$C[i, j] = 0$ $C[2, 3] = 24$

$A_1 \times (A_2 \times A_3)$

$\underline{2} \cdot \underline{3} \cdot \underline{3} \cdot \underline{2}$

$2 \cdot 3 \cdot 2 = 12$ multiplications

Now all together:

$0 + 24 + 2 \cdot 3 \cdot 2 = 36$

So 36 multiplications are required for METHOD 2.

$C[i, k] + C[k+1, j] + d_{i-1} \cdot d_k \cdot d_j = 36$

[MATRIX CHAIN MULTIPLICATION] COST OF MULTIPLYING

$$C[i,j] = \min_{i \leq k < j} \{ C[i,k] + C[k+1,j] + d_{i-1} \cdot d_k \cdot d_j \}$$

$$\begin{matrix} 1 & & 2 & & 3 & & 4 & & \# \text{ matrices} \\ A_1 & \times & A_2 & \times & A_3 & \times & A_4 & & \downarrow \\ d_0 & d_1 & d_1 & d_2 & d_2 & d_3 & d_3 & d_4 & n = 4 - 1 \end{matrix}$$

5 DIFFERENT METHODS:

- 1) $A_1 (A_2 (A_3 A_4))$
- 2) $A_1 ((A_2 A_3) A_4)$
- 3) $(A_1 A_2) (A_3 A_4)$
- 4) $(A_1 (A_2 A_3)) A_4$
- 5) $((A_1 A_2) A_3) A_4$

$$\frac{2n(n-1)}{n+1}$$

↓

$$\frac{2(n-1)}{n} C_{(n-1)}$$

n=4 ↓

$$\frac{2(4-1)}{4} C_{(4-1)} = \frac{2(3)}{4} C_{(3)} = \frac{6}{4} C_3$$

$$= \frac{6 \cdot 5 \cdot 4}{4 \cdot 3 \cdot 2 \cdot 1} = 5 \text{ different methods}$$

FORMULA:

$$C[1,4] = \min_{1 \leq k < 4} \begin{cases} k=1: C[1,1] + C[2,4] + d_0 \cdot d_1 \cdot d_4, \\ k=2: C[1,2] + C[3,4] + d_0 \cdot d_2 \cdot d_4, \\ k=3: C[1,3] + C[4,4] + d_0 \cdot d_3 \cdot d_4 \end{cases}$$

One of these three will give us the minimum cost of multiplying.

$$(k=1): C[1,1] + C[2,4] = A_1 (A_2 A_3 A_4)$$

$$(k=2): C[1,2] + C[3,4] = (A_1 A_2) (A_3 A_4)$$

$$(k=3): C[1,3] + C[4,4] = (A_1 A_2 A_3) A_4$$

$$C[2,4] = \min_{2 \leq k < 4} \begin{cases} k=2: C[2,2] + C[3,4] + d_1 \cdot d_2 \cdot d_4 \\ k=3: C[2,3] + C[4,4] + d_1 \cdot d_3 \cdot d_4 \end{cases}$$

dette er ikke effektivt.

SR er lavere skema i stedet for, på næste side

17.1.4 EXAMPLE: Minimum Cost to Multiply Chains of Matrices

MATRIX MULTIPLICATION - EXAMPLE

$C[i, j] = \min_{i \leq k < j} \{ C[i, k] + C[k+1, j] + d_i \cdot d_k \cdot d_j \}$

$A_1 \times A_2 \times A_3 \times A_4$
 $d_0 \quad d_1 \quad d_2 \quad d_3 \quad d_4$

Cost Table:

	1	2	3	4
1	0	24	28	58
2		0	16	36
3			0	40
4				0

Labels: FIRST (3,4), SECOND (2,3), THIRD (1,2)

Calculations:

$$C[1, 2] = \min_{1 \leq k < 2} \{ C[1, 1] + C[2, 2] + d_0 \cdot d_1 \cdot d_2 \} = 0 + 0 + 3 \cdot 2 \cdot 4 = 24$$

$$C[2, 3] = \min_{2 \leq k < 3} \{ C[2, 2] + C[3, 3] + d_1 \cdot d_2 \cdot d_3 \} = 0 + 0 + 2 \cdot 4 \cdot 2 = 16$$

$$C[3, 4] = \min_{3 \leq k < 4} \{ C[3, 3] + C[4, 4] + d_2 \cdot d_3 \cdot d_4 \} = 0 + 0 + 4 \cdot 2 \cdot 5 = 40$$

$$C[1, 3] = \min_{1 \leq k < 3} \begin{cases} k=1: C[1, 1] + C[2, 3] + d_0 \cdot d_1 \cdot d_3 = 0 + 16 + 3 \cdot 2 \cdot 2 = 28 \\ k=2: C[1, 2] + C[3, 3] + d_0 \cdot d_2 \cdot d_3 = 24 + 0 + 3 \cdot 4 \cdot 2 = 48 \end{cases} = 28$$

$$C[2, 4] = \min_{2 \leq k < 4} \begin{cases} k=2: C[2, 2] + C[3, 4] + d_1 \cdot d_2 \cdot d_4 = 0 + 40 + 2 \cdot 4 \cdot 5 = 80 \\ k=3: C[2, 3] + C[4, 4] + d_1 \cdot d_3 \cdot d_4 = 16 + 0 + 2 \cdot 2 \cdot 5 = 36 \end{cases} = 36$$

$$C[1, 4] = \min_{1 \leq k < 4} \begin{cases} k=1: C[1, 1] + C[2, 4] + d_0 \cdot d_1 \cdot d_4 = 0 + 36 + 3 \cdot 2 \cdot 5 = 66 \\ k=2: C[1, 2] + C[3, 4] + d_0 \cdot d_2 \cdot d_4 = 24 + 40 + 3 \cdot 4 \cdot 5 = 124 \\ k=3: C[1, 3] + C[4, 4] + d_0 \cdot d_3 \cdot d_4 = 28 + 0 + 3 \cdot 2 \cdot 5 = 58 \end{cases} = 58$$

Therefore we can conclude, that the minimum cost of multiplying the 4 matrices in this example = 58

BUT HOW SHOULD THE PARENTHIZATION BE? LOOK AT K-VALUE SCHEMA.

$((A_1) \times (A_2 \times A_3)) \times A_4$

K-VALUE SCHEMA:

	1	2	3	4
1		1	1	3
2			2	3
3				3
4				

Diagram:

```

graph TD
    A1((A1)) --- Node1(( ))
    A2((A2)) --- Node1
    Node1 --- Node2(( ))
    A3((A3)) --- Node2
    Node2 --- Node3(( ))
    Node3 --- A4((A4))
  
```

17.1.5 Partitioning Array to Minimize Sum

Similar in spirit to MCM, where the task is to partition an array into k contiguous subarrays to minimize the maximum sum among them. Dynamic programming can be used to find the optimal partitions.

Recursion Equation:

$$DP[i][k] = \min_{0 < j < i} \left\{ \max \left(DP[j][k-1], \sum_{m=j+1}^i \text{arr}[m] \right) \right\}$$

Pseudocode:

```
function PartitionArray(arr, k):
    // arr is the array to partition
    // k is the number of partitions
    n = length(arr)
    DP = 2D-array of size (n+1) x (k+1), filled with Infinity
    prefixSum = array of size n+1
    for i from 1 to n:
        prefixSum[i] = prefixSum[i-1] + arr[i-1]
    for i from 1 to n:
        DP[i][1] = prefixSum[i]
    for j from 1 to k:
        DP[1][j] = arr[0]
    for i from 2 to n:
        for j from 2 to k:
            for m from 1 to i-1:
                DP[i][j] = min(DP[i][j], max(DP[m][j-1], prefixSum[i]-prefixSum[m]))
    return DP[n][k]
```

Time Complexity: $O(n^2 \cdot k)$, where n is the number of elements in the array, because for each segment, we look at all possible partition points.

17.1.6 Chain of Pairs

Given pairs of numbers, the task is to form a chain to maximize the length of the chain. A pair (a, b) can follow another pair (c, d) only if $b < c$. This problem can be solved using a variation of the MCM strategy by dynamically selecting the optimal pairs to link.

Recursion Equation:

$$DP[i] = \max_{0 \leq j < i} \{1 + DP[j] \text{ if pair } [j].\text{second} < \text{pair } [i].\text{first} \}$$

Pseudocode:

```

function MaxLengthChain(pairs):
    // pairs is an array of tuples (a, b)
    n = length(pairs)
    sort(pairs) based on first element
    DP = array of size n, filled with 1
    for i from 1 to n:
        for j from 0 to i-1:
            if pairs[j].second < pairs[i].first and DP[j] + 1 > DP[i]:
                DP[i] = DP[j] + 1
    return max(DP)

```

Time Complexity: $O(n^2)$, where n is the number of pairs, because for each pair, we potentially look at all other pairs to find the longest chain that can be formed.

17.1.7 Burst Balloons

Given n balloons, each with a certain number of coins associated with it, bursting a balloon i gets you the coins associated with it multiplied by the numbers of its adjacent balloons. The goal is to find the maximum coins you can collect by bursting all the balloons wisely. This problem can be approached similarly to MCM by considering the last balloon to be burst.

Rekursionsligning:

$$DP[i][j] = \max_{i < k < j} \{DP[i][k] + DP[k][j] + nums[i] \cdot nums[k] \cdot nums[j]\}$$

Pseudokode:

```

function MaxCoins(nums):
    // Tilføj virtuelle balloner med 1 mønt
    nums = [1] + nums + [1]
    n = længden af nums
    DP = 2D-array af størrelse n x n, fyldt med 0
    for L fra 2 til n:
        for i fra 0 til n-L:
            j = i + L
            for k fra i+1 til j-1:
                q = DP[i][k] + DP[k][j] + nums[i]*nums[k]*nums[j]
                DP[i][j] = maks(DP[i][j], q)
    return DP[0][n-1]

```

Time Complexity: $O(n^3)$, where n is the number of balloons, similar to MCM due to considering all subarrays and splits.

18 knapsack

Antag at vi har n objekter, og hvert objekt i har en vægt w_i og en værdi v_i . Rygsækken har en kapacitetsgrænse W . Vi skal bestemme, hvilke objekter der skal inkluderes i rygsækken, så den samlede værdi af de inkluderede objekter er maksimeret, uden at den samlede vægt overskrider W .

Vi anvender en dynamisk programmeringstilgang til at løse dette problem ved at oprette en tabel DP hvor $DP[i][w]$ repræsenterer den maksimale værdi, der kan opnås med de første i objekter og en maksimal vægt w .

Initialiser en tabel DP med størrelse $(n + 1) \times (W + 1)$, hvor alle værdier sættes til 0:

$$DP[0][w] = 0 \quad \text{for alle } w = 0, 1, \dots, W$$

$$DP[i][0] = 0 \quad \text{for alle } i = 0, 1, \dots, n$$

For hvert objekt i fra 1 til n og for hver vægt w fra 1 til W , opdateres $DP[i][w]$ som følger:

$$DP[i][w] = \begin{cases} DP[i-1][w] & \text{hvis } w_i > w \\ \max(DP[i-1][w], DP[i-1][w - w_i] + v_i) & \text{hvis } w_i \leq w \end{cases}$$

Den maksimale værdi, der kan opnås uden at overskride rygsækkens kapacitet, findes i $DP[n][W]$.