Algoritmer og datastrukturer

I dette forløb arbejder vi med algoritmer og datastrukturer der kan optimere løsning af forskellige opgaver. Meget af dette findes allerede indbygget i .NET-bibliotekerne, men vi bygger dem selv fordi det giver en grundigere forståelse af løsningerne.

Formål med dette forløb

Formålet med et forløb omkring forholdsvis avancerede algoritmer og datastrukturer er at udfordre eleven, at træne ham/hende i at udtænke strategier til at løse komplicerede eller komplekse opgaver.

En gevinst ved at arbejde med disse områder er også at eleven får kendskab til hvordan mange af de datastrukturer og metoder der findes i .Net-bibliotekerne er konstrueret, og derfor vil have bedre forudsætninger for at vælge den mest hensigtsmæssige datastruktur/method til løsning af en konkret opgave.

Faglige læringsmål

* Forståelse af begrebet tidskompleksitet, samt O-notationen
* Give en dybere forståelse for datalogiske problemstillinger
* Erfaring med og forståelse af divide and conquer strategier
* Erfaring med at arbejde med mere komplicerede opgaver end administrative systemer
* At eleven får en indsigt der sætter ham/hun i stand til at

Konkrete elementer

* Rekursion
* Lineære datastrukturer arrays, linkede lister
* QuickSort
* MergeSort
* Træstrukturer: Søgetræer, AVL, bunke/hob (heap)
* Traversering af træer.
* HeapSort
* Grafer, den korteste vej i en graf

Forudsætninger

Det er en forudsætning at man har et grundigt kendskab til programmering og trives ved løsning af små forholdsvis komplekse problemstillinger.

Det forudsættes at man har gennemført modulerne S1, S2 og S3.

Links

Her er lidt links til yderligere information om forskellige sorteringsalgoritmer. Hvis du savner ekstra information til nogle emner, kan det være en idé at dykke ned her. Som udgangspunkt har jeg dog forsøgt selv at forklare alt relevant i dette dokument, så benyt kun disse links hvis du har behov for yderligere viden.

<https://en.wikipedia.org/wiki/Sorting_algorithm>

<https://www.cs.au.dk/~gerth/dADS1-13/Java-C++/>

<https://cs.au.dk/~gerth/dPersp15/>

<https://slideplayer.dk/slide/2323461/>

<https://dotnetcoretutorials.com/2020/05/10/basic-sorting-algorithms-in-c/>

Ordbog:

<https://thorehusfeldt.com/2011/03/29/danskalgoritmeterminologi/>

Indhold

[1. Introduktion 7](#_Toc87523606)

[2. Rekursion 7](#_Toc87523607)

[2.1. Hvad er rekursion? 7](#_Toc87523608)

[2.2. Øvelser **Fejl! Bogmærke er ikke defineret.**](#_Toc87523609)

[2.2.1. Factorial 7](#_Toc87523610)

[2.2.2. Fibonacci 7](#_Toc87523611)

[2.2.3. Udskriv alle filer i en mappe og dens undermapper 7](#_Toc87523612)

[2.3. Yderligere info 8](#_Toc87523613)

[3. Tidskompleksitet 8](#_Toc87523614)

[3.1. Lineær tidskompleksitet 8](#_Toc87523615)

[3.2. Logaritmisk tidskompleksitet 8](#_Toc87523616)

[3.3. Kvadratisk-tidskompleksitet O(n2) 9](#_Toc87523617)

[3.4. O(n∙log(n)) 10](#_Toc87523618)

[4. Søgning 10](#_Toc87523619)

[4.1. Uorganiseret data 10](#_Toc87523620)

[4.1.1. Lineær søgning 10](#_Toc87523621)

[4.1.2. Hurtig indsættelse 11](#_Toc87523622)

[4.2. Sorteret data 11](#_Toc87523623)

[4.2.1. Lineær søgning 11](#_Toc87523624)

[4.2.2. Binær søgning 11](#_Toc87523625)

[4.2.3. Indsættelse/sletning i sorteret data 11](#_Toc87523626)

[4.3. Datastrukturer med hurtig indsættelse og sletning 12](#_Toc87523627)

[4.3.1. Linked list (dobbelt-linkede lister) 12](#_Toc87523628)

[4.3.2. Køstrukturer 12](#_Toc87523629)

[4.3.3. Opgave 12](#_Toc87523630)

[4.4. Træer - vigtige datalogiske datastrukturer 13](#_Toc87523631)

[4.4.1. Hvad er et træ? 13](#_Toc87523632)

[4.4.2. Hvad er et binært træ? 13](#_Toc87523633)

[4.5. Søgetræ – en datastruktur til søgning 14](#_Toc87523634)

[4.5.1. Hvad er et søgetræ? 14](#_Toc87523635)

[4.5.2. Funktioner i søgetræ 15](#_Toc87523636)

[4.5.3. Opgave: Implementér et søgetræ 16](#_Toc87523637)

[4.5.4. Opgave: Gennemløb af søgetræ 16](#_Toc87523638)

[4.5.5. Opgave: Test af søgetræ 16](#_Toc87523639)

[5. Sortering 17](#_Toc87523640)

[5.1. De umiddelbare sorteringsmetoder O(n2) 17](#_Toc87523641)

[5.1.1. Udvælgelsessortering 17](#_Toc87523642)

[5.1.2. Opgave 18](#_Toc87523643)

[5.1.3. Indsættelsessortering 18](#_Toc87523644)

[5.1.4. Opgave 18](#_Toc87523645)

[5.1.5. Bubble sort 18](#_Toc87523646)

[5.1.6. Opgave 19](#_Toc87523647)

[5.1.7. Opgave 19](#_Toc87523648)

[5.2. Optimeret sortering () 19](#_Toc87523649)

[5.2.1. Sortering ved hjælp af søgetræ (O(n∙log(n)/O(n2) ) 19](#_Toc87523650)

[5.2.2. Opgave sortering med søgetræ 19](#_Toc87523651)

[5.3. Divide and conquer metoder (O(n∙log(n)) 19](#_Toc87523652)

[5.3.1. Quicksort (O(n∙log(n)/O(n2)) 20](#_Toc87523653)

[5.3.2. MergeSort O(n∙log(n)) 22](#_Toc87523654)

[5.4. Heap (bunke/hob) - en anden tilgang til sortering 24](#_Toc87523655)

[5.4.1. Operationer på en bunke 25](#_Toc87523656)

[5.4.2. Bunkesortering/hobsortering – heapsort O(n∙log(n)) 30](#_Toc87523657)

[5.4.3. Opgave 30](#_Toc87523658)

[5.4.4. En bunke i et array 30](#_Toc87523659)

[5.4.5. Opgave 31](#_Toc87523660)

[6. Teori versus virkelighed 31](#_Toc87523661)

[6.1. Opgave 31](#_Toc87523662)

[6.2. Opgave 31](#_Toc87523663)

[6.3. Opgave Grafer 32](#_Toc87523664)

[7. Hvad gør .NET? 35](#_Toc87523665)

[7.1. List.Sort/Array.Sort 35](#_Toc87523666)

[7.2. Enumerable.OrderBy 36](#_Toc87523667)

[8. Specielle trætyper 36](#_Toc87523668)

[8.1. AVL træer - avanceret binært søgetræ 36](#_Toc87523669)

[8.1.1. Hvad er et AVL træ? 36](#_Toc87523670)

[8.1.2. Links 36](#_Toc87523671)

[8.2. Opgaver: Implementér et AVL-træ 37](#_Toc87523672)

[8.2.1. Opgave: Indsættelse 37](#_Toc87523673)

[8.2.2. Opgave: Sletning 37](#_Toc87523674)

[8.2.3. Opgave 37](#_Toc87523675)

[8.3. Opgave: Tidskompleksitet 37](#_Toc87523676)

[8.4. Andre træstrukturer 37](#_Toc87523677)

[8.4.1. M-ary trees 37](#_Toc87523678)

[8.4.2. Splay tree 37](#_Toc87523679)

[8.4.3. Rød-sorte træer 37](#_Toc87523680)

[9. Hashtabeller 37](#_Toc87523681)

[9.1. Hvad er en Hashtabel 37](#_Toc87523682)

[9.2. Hvordan implementeres en hashtabel? 37](#_Toc87523683)

[9.2.1. Implementation af hashfunktion 37](#_Toc87523684)

[9.2.2. Implementation af selve hashtabellen 38](#_Toc87523685)

[9.2.3. Opgave: Hashtabel 38](#_Toc87523686)

[9.3. Hvor bruger man hashtabeller? 38](#_Toc87523687)

[9.3.1. Databaser 38](#_Toc87523688)

[9.3.2. Hashtabel i .NET 38](#_Toc87523689)

[9.4. Resizing 38](#_Toc87523690)

[9.5. Opgave: Automatisk resizing 39](#_Toc87523691)

[9.6. Tidskompleksitet 39](#_Toc87523692)

[9.6.1. Ny værdi/opdateret værdi og læsning 39](#_Toc87523693)

[9.7. Online information 39](#_Toc87523694)

[9.7.1. hash tables 39](#_Toc87523695)

[9.7.2. Hash functions 39](#_Toc87523696)

[10. Grafer 40](#_Toc87523697)

[10.1. Gennemløb af grafer 40](#_Toc87523698)

[10.2. Dybde først traversering 40](#_Toc87523699)

[10.3. Bredde først traversering 41](#_Toc87523700)

[10.4. Uddybende information findes på nettet 41](#_Toc87523701)

[10.5. Opgave: Implementér gennemløb af en graf 41](#_Toc87523702)

[10.5.1. Implementér en graf 41](#_Toc87523703)

[10.5.2. Implementér en dybde-først traversering af grafen 41](#_Toc87523704)

[10.6. Edsger W. Edsger Dijkstras algoritme 41](#_Toc87523705)

[10.6.1. Opgave: Implementér Dijkstra’s algoritme 42](#_Toc87523706)

[10.6.2. Opgave: Optimér søgning efter knude 42](#_Toc87523707)

[10.6.3. Tidskompleksitet for Dijkstras algoritme 42](#_Toc87523708)

[10.6.4. Opgave: Redegør for tidskompleksitet 43](#_Toc87523709)

[11. Ekstra: Flere spændende datastrukturer, algoritmer og problemer 43](#_Toc87523710)

[11.1. Rød/sort træer 43](#_Toc87523711)

[11.2. Splay trees 43](#_Toc87523712)

[11.3. Grafteori 43](#_Toc87523713)

[11.4. Travelling salesman problem 43](#_Toc87523714)

[11.5. Fibonacci heap 43](#_Toc87523715)

[11.6. Søgning efter koordinater 43](#_Toc87523716)

# Introduktion

En central disciplin i programmering er søgning og sortering. I dag er forskellige søgnings- og sorteringsalgoritmer implementeret i standardbiblioteker som f.eks. .NET. Det betyder at man som programmør ikke selv skal implementere sine søgerutiner.

I dette forløb skal vi selv kreere alle de klasser vi skal bruge. Det vil kun være de indbyggede C#-datatyper vi må benytte[[1]](#footnote-1).

I forhold til forståelse af programmering af forholdsvise komplekse opgaver giver søgnings- og sorteringsalgoritmer mulighed for god læring af del-og-hersk-principper i opgaver af overskueligt omfang. Derfor er det i høj grad relevant at arbejde med søgnings- og sorteringsalgoritmer i forbindelse med uddannelse i programmering.

# Rekursion

## Hvad er rekursion?

Rekursion er det der foregår når en metode kalder sig selv. De første gange de fleste støder på rekursion er når de ved en fejl er kommet til at lade en metode kalde sig selv i det uendelige. I disse tilfælde får man en *stack overflow* fejl fordi hvert metodekald gemmes på et sted i computerens hukommelse der kaldes kaldsstakken. Så en uendelig rekursion får programmet til at gå ned (smide en undtagelse).

Men rekursion kan også bruges til at lave elegante løsninger til opgaver der samlet set virker komplicerede, men som med en rekursiv løsning virker nem og overskuelig.

Se denne video for en introduktion til hvad rekursion er (den er ret underholdende): [Video](https://www.youtube.com/watch?v=AfBqVVKg4GE).

## Opgaver

Her er nogle øvelser du sikkert allerede har løst tidligere, men med iterative metoder (løkker). Nu skal du løse de samme problemstillinger ved hjælp af *rekursive metoder*.

Hvis du allerede tidligere har løst nogle af opgaverne med rekursion, kan du eventuelt springe nogle disse opgaver over. Det er dog under alle omstændigheder godt lige at genopfriske rekursion, da vi skal bruge det ret meget i dette forløb.

### Factorial

Lav en metode der tager i mod et naturligt tal *n* og som udskriver *n* fakultet.

Hjælp: Løsning findes i denne [video](https://www.youtube.com/watch?v=6Mnn7KOZ9l0):

### Fibonacci

Lav en metode der tager i mod et tal *n* og som finder det *n’te* tal i Fibonacci talrækken.

Lav en metode der tager i mod et tal *n* og som udskriver de første *n* tal af Fibonacci talrækken (husk den skal også være rekursiv).

### Udskriv alle filer i en mappe og dens undermapper

Lav en metode der tager i mod en sti, og som udskriver alle filer i mappen og dens undermapper, og dens undermapper etc., indtil der ikke er flere undermapper.

## Yderligere info

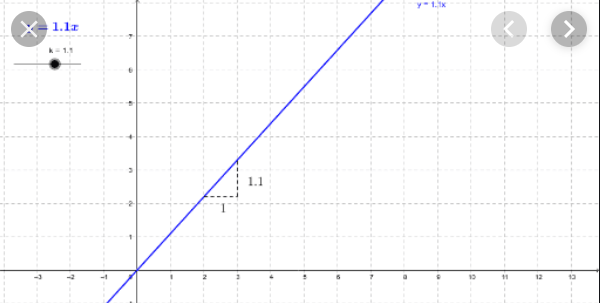
<https://www.educative.io/blog/recursion>

# Tidskompleksitet

I forbindelse med søgning og sortering skal vi arbejde med et begreb der hedder tidskompleksitet. Tidskompleksiteten er et udtryk for hvor lang tid en opgave tager i forhold til hvor mange elementer der er i det data man arbejder med (antallet af elementer benævnes ofte ’n’).

## Lineær tidskompleksitet

En opgave hvor man skal undersøge alle elementer siges at have lineær tidskompleksitet. Det betyder at tiden det tager at udføre opgaven vokser proportionalt med antallet af elementer. Dette skrives formelt sådan: .

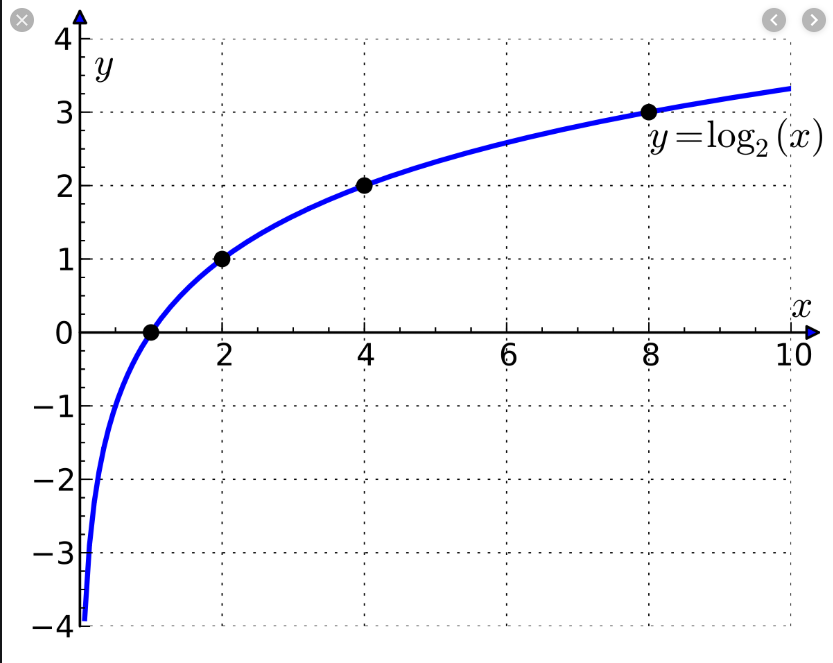


Stejlheden af grafen har ikke nogen betydning for hvilken tidskompleksitet opgaven har, så det vigtige i denne sammenhæng er at grafen hele tiden vokser lige hurtigt.

## Logaritmisk tidskompleksitet = Log2(n2)

En opgave hvor man hele tiden halverer det resterende antal elementer man skal håndtere har en tidskompleksitet på . Det betyder at den tid en opgave tager er proportional med (den siges at have logaritmisk tidskompleksitet).

Som det fremgår af grafen nedenfor, så vokser en del langsommere end . som der står i figuren betyder 2‑talslogaritmen – det er egentlig ikke så vigtigt hvilken logaritme der er tale om, men totalslogaritmen af et tal er netop hvor mange gange tallet kan halveres (før man når 1), så det er normalt den man tænker på i forbindelse med tidskompleksitet af programmeringsløsninger.



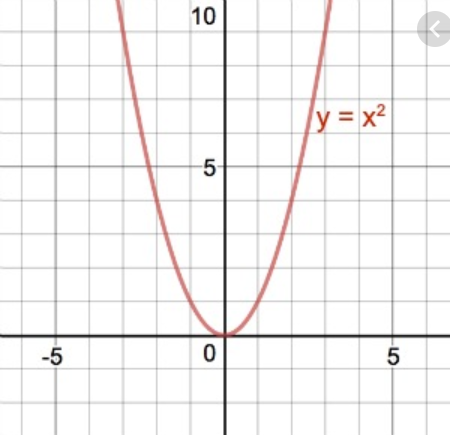
Som det fremgår, så er grafen ret stejl i starten, men når n vokser (kaldet x i grafen ovenfor), så vokser funktionsværdien - altså y - med mindre og mindre. Det viser tydeligt at en logaritmisk tidskompleksitet vil være foretrække frem for en lineær når antallet af elementer er stort.

## Kvadratisk-tidskompleksitet O(n2)

Nogle opgaver er mere komplekse og kræver at man besøger alle elementer en gang for (at sammenligne med) hvert element. Dette giver en tidskompleksitet på .

De mest simple sorteringsalgoritmer har denne tidskompleksitet fordi hvert element skal sammenlignes med alle allerede sorterede elementer eller alle usorterede elementer (afhængig af hvilken sorterings­algoritme der er i spil).

Som det fremgår af nedenstående graf, så vokser tidskompleksiteten (op ad y-aksen) meget hurtigt når n vokser (når vi går ud ad x-aksen).

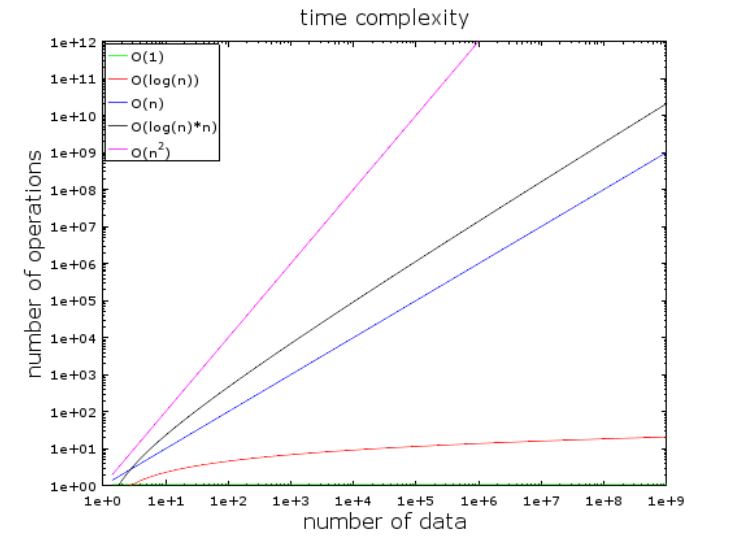


Bemærk, de negative værdier af x er ikke interessante i forbindelse med tidsforbrug da det ikke giver mening at snakke om f.eks. at sortere et negativt antal elementer.

## O(n∙log(n))

For at undgå at sortering af store mængder data tager alt for lang tid er der arbejdet meget med at finde sorteringsalgoritmer der er hurtige. Man er kommet frem til en del forskellige algoritmer med en tidskompleksitet der enten gennemsnitligt eller selv i værste tilfælde har tidskompleksiteten .

Der er en stor gevinst ved at gå fra til som det fremgår af grafen nedenfor. Graferne er indtegnet i et logaritmisk koordinatsystem, hvilket gør at graferne ser anderledes ud end når de indtegnes i et almindeligt (lineært) koordinatsystem, men dette koordinatsystem fungerer godt til at sammenligne de forskellige tidskompleksiteter.



[Kilde](http://coding-geek.com/how-databases-work/)

Som man kan se, så er de blå () og grønne () grafer nærmest parallelle i dette koordinat­system, mens pink () er meget stejlere.

# Søgning

Ofte har man brug for at finde noget bestemt data i en datamængde. Det kunne være det højeste eller laveste pointtal, en person med et bestemt navn – der er mange muligheder – det er noget de fleste programmer anvender ofte.

Derfor er det også ofte vigtigt at en søgning er forholdsvis effektiv.

En effektiv søgealgoritme kræver at data er organiseret på en måde som søgealgoritmen kan udnytte til at optimere søgetiden.

## Uorganiseret data

Ofte har man som udgangspunkt for en opgave en mængde data der er uorganiseret og bare ligger i et array eller en anden datastruktur.

### Lineær søgning

Hvis man skal søge i uorganiseret data, bliver man nødt til at undersøge alle elementer indtil man finder det man søger efter. Dette giver en lineær tidskompleksitet .

### Hurtig indsættelse

Det er hurtigt at indsætte et nyt element i en uorganiseret struktur, da elementet bare skal tilføjes til strukturen, der er ikke behov for at finde ud af hvor det skal indsættes. Dette giver en konstant tidskompleksitet .

## Sorteret data

Hvis data er sorteret, er det nemmere at søge i. Vi ved f.eks. at tallet 9 må ligge efter 8 og før 10 hvis disse tal er en del af data (eller omvendt afhængig af om vi har mindste tal først eller sidst).

### Lineær søgning

Umiddelbart kunne man søge efter 9 ved at undersøge om det første tal var 9 og så fortsætte indtil vi finder 9 eller et tal der er større. På den måde slipper vi for at lede hele datasamlingen igennem hvis ikke tallet findes.

Tidskompleksiteten for denne søgemetode er dog stadig , da vi kan risikere at skulle sammenligne 9 med alle elementer i samlingen (hvis 9 er større end eller lig det største tal i samlingen). I gennemsnit vil vi skulle sammenligne med halvdelen af elementerne, hvilket også giver tidskompleksitet .

### Binær søgning

Ved binær søgning halverer man hele tiden antallet af resterende mulige kandidater i en søgning. Det betyder at vi får en tidskompleksitet på .

Binær søgning foregår ved at man i en sorteret liste undersøger om det element vi leder efter er større eller mindre end det midterste element i listen. Ud fra det kan vi udelukke halvdelen af elementerne. Denne procedure gentages indtil der kun er 1 eller 2 elementer tilbage.

### Opgave: Binær søgning

Lav en klasse SortedArray - som indeholder et array som initialiseres med et sorteret datasæt[[2]](#footnote-2).

Implementér en binær søgealgoritme som kan finde et givet tal i et sorteret array af tal ved hjælp af binær søgning.

### Indsættelse/sletning i sorteret data

Hvis man har en sorteret liste, så tager det lidt tid at indsætte nye elementer fordi listen skal holdes sorteret. Det betyder at vi skal finde det rigtige sted at indsætte og derefter indsætte det.

Selve indsættelsen kræver at de eksisterende elementer flyttes for at gøre plads. Da vi kan risikere at skulle flytte alle elementer for at gøre plads, giver dette en tidskompleksitet på .

Den samlede tidskompleksitet findes ved at lægge de to tidskompleksiteter sammen, og så kan man se at tiden for søgningen bliver uvæsentlig i denne sammenhæng. Når man lægger tidskompleksiteter sammen falder de mindre størrelser bort.

Indsættelse vil altså have en tidskompleksitet på .

Ved tilsvarende argumentation ser vi at tidskompleksiteten for sletning af elementer også er .

### Opgave: Indsættelse i sorteret array

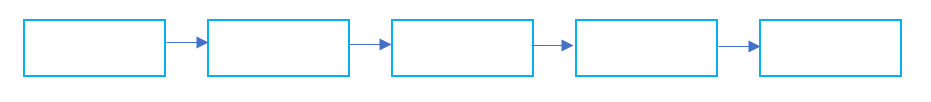
Implementér Insert i klassen SortedArray. Metoden skal indsætte et nyt element i arrayet på en sådan måde at arrayet holdes sorteret.

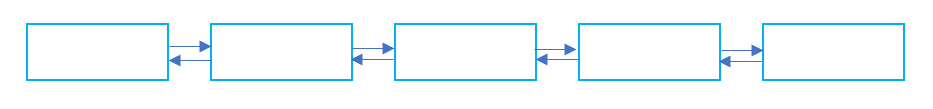
## Datastrukturer med hurtig indsættelse og sletning

Som vi har set i forbindelse med sorterede lister ovenfor, så bliver indsættelse langsom på grund af alle de elementer der skal flyttes for at give plads til det nye element. Måske kunne man finde en datastruktur hvor det er nemt at indsætte nye elementer og fjerne gamle.

### Linked list (dobbelt-linkede lister)

Linkede lister er meget hurtige at indsætte i og fjerne fra hvis man ved hvor man skal indsætte eller fjerne. Der er ikke eksisterende elementer som skal flyttes rundt, der er bare nogle referencer der skal opdateres.

 Linket liste



Dobbelt-linket liste

Til gengæld er det langsomt at søge i linkede lister, man kan ikke som i arrays gå direkte til de enkelte elementer men er nødt til at løbe dem igennem fra en ende af, så søgning vil altid blive lineær ().

På linkede lister har man ofte funktionerne Prepend, Append, RemoveFirst, RemoveLast (eller andre navne). Er alle metoderne nemme at implementere i begge typer linkede lister? Overvej hvordan det kan gøres.

### Køstrukturer

Linkede lister kan med stor fordel bruges når man altid indsætter eller fjerner et bestemt sted. Så bliver der ingen søgning i forbindelse med indsættelse og udtagning, hvilket betyder at det kommer til at foregå med konstant tidskompleksitet ().

To eksempler på køstrukturer hvor man med fordel kan benytte linkede lister er kø og stak som beskrevet nedenfor.

#### Fifo (Kø)

En fifo (first in first out) kø kan nemt implementeres som en linked list fordi indsættelse og sletning skal foregå hurtigt og der er ikke behov for søgning. Kan man nøjes med en enkelt-linket liste?

#### Lifo (stak)

En lifo (last in first out) stak kan også nemt implementeres som en linked list, igen fordi indsættelse og sletning er hurtig og der ikke skal søges. Kan man nøjes med en enkelt-linket liste?

Lifo kan dog også nemt implementeres i et almindeligt array (eller List<>) da vi altid skal indsætte og slette i samme ende – i et array kan dette indeks udpeges af antallet af elementer i listen.

### Opgave

Implementér en Fifo kø og en Lifo kø ved hjælp af dobbeltlinkede lister som du selv implementerer *helt fra bunden*. I begge datastrukturer skal der være metoder til indsættelse (push) og udtagelse (pop).

Lav et lille program til at teste dine køer, hvor du sætter en række tal ind i køen og derefter trækker dem ud.

## Træer - vigtige datalogiske datastrukturer

I næste afsnit skal vi til at arbejde med søgetræer og senere med en anden træstruktur som kaldes bunke (også kaldet hob, på engelsk heap). For lige at have en vis forståelse af hvad et træ i det hele taget er, definerer vi i dette afsnit begrebet træ og underbegrebet binært træ.

Både søgetræet og bunken som vi skal arbejde med i dette opgavesæt er binære træer.

### Hvad er et træ?

Et træ er en datastruktur bygget op på samme måde som en linket liste, bortset fra at hvert element (knude) har links til flere underknuder. Oftest vil underknuderne (child) også kende deres overknude (parent) på samme måde som elementerne i en dobbeltlinket liste kender begge naboelementer.

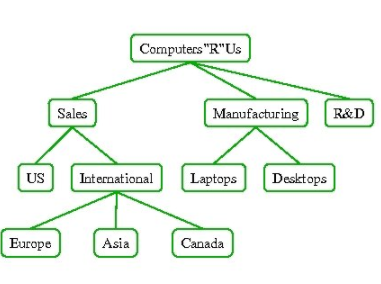
En træstruktur består af et antal knuder og nogle forbindelser mellem knuderne der er ordnet sådan at man kan tegne strukturen som et ”træ”, f.eks. et stamtræ. En folderstruktur hvor en folder kan have underholdere der igen kan have underfoldere etc. er også et eksempel på en træstruktur.

Samtidig gælder der at to knuder ikke kan have samme knude som child. Det betyder der ikke kan være nogen cyklisk struktur, man kan bevæge sig op og ned i træet, men ikke på tværs.

Der vil altid være en såkaldt *rod* i et træ (root), det vil sige en knude som ikke er child til nogen anden knude (og dermed ikke selv har nogen parent).

Et træ har også altid et antal *blade*, det er knuder som ikke har children.

En virksomheds organisation kan ofte illustreres som en træstruktur:

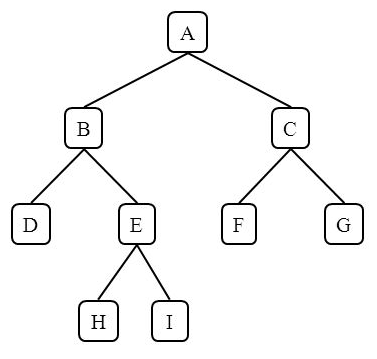


Rod

Blad

### Hvad er et binært træ?

Et binært træ er et træ hvor hver knude højest kan have to underknuder. Disse to knuder betragtes som ”søskende” (siblings), dvs. i eksemplet nedenfor er B og C søskende, D og E er søskende osv.



Binære træer er meget anvendelige til løsning af forskellige datalogiske problemstillinger.

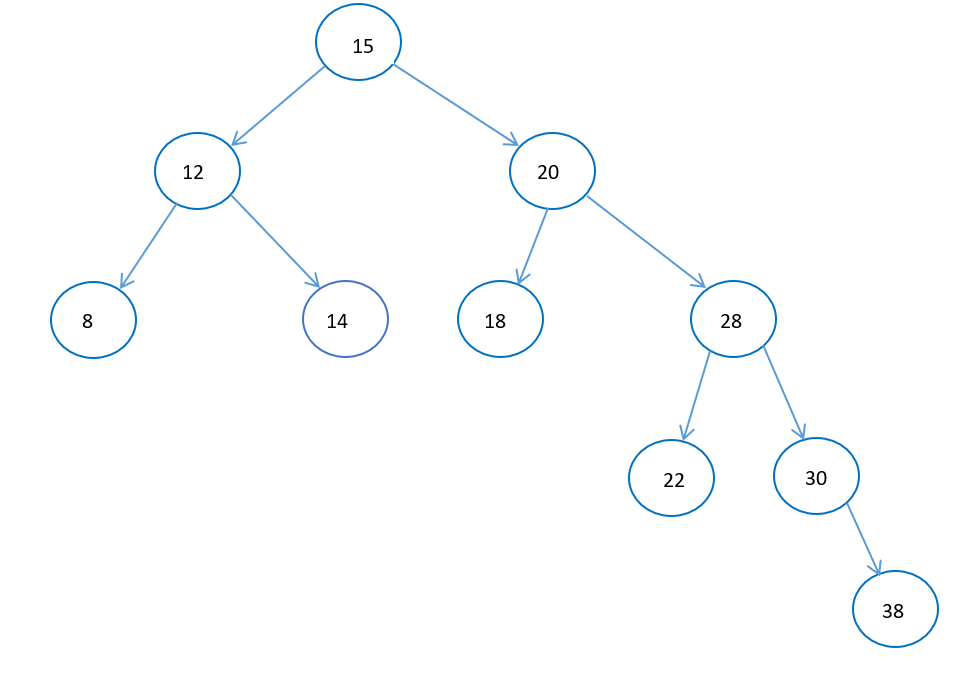
## Søgetræ – en datastruktur til søgning

Ved linkede lister kunne man opnå hurtig indsættelse og sletning men søgning er langsom (lineær). Det kunne være nærliggende at forsøge at opbygge en struktur der kunne udnytte den hurtige indsættelse og udtagning og samtidig kunne give en noget hurtigere søgning.

I dette afsnit skal vi se hvordan man kan anvende et binært træ organiseret som et såkaldt søgetræ til at foretage hurtig søgning i et dataset samtidig med at vi har hurtig indsættelse og sletning.

### Hvad er et søgetræ?

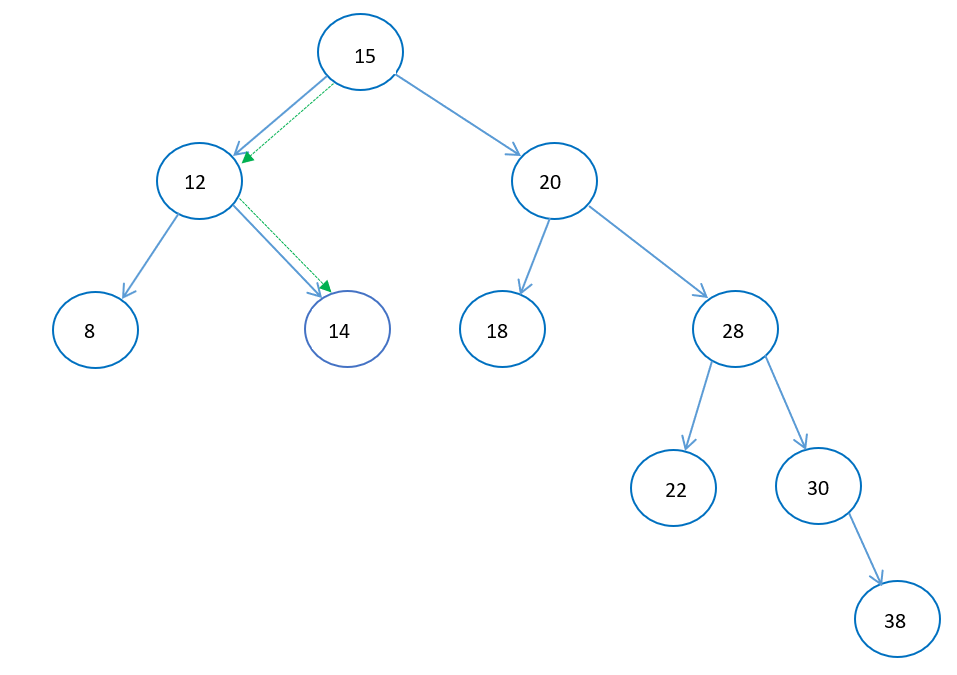
Et søgetræ er et binært træ, struktureret på den måde at elementer der er mindre end det aktuelle element ligger i undertræet til venstre, mens elementer der er større end det aktuelle ligger i undertræet til højre.



#### Søgning i søgetræ

Hvis vi altså skal finde en knude med en bestemt værdi - f.eks. 14 - starter vi med at sammenligne værdien vi søger efter (14) med værdien af træets rod (altså 15). Da 14 er mindre end 15 skal vi lede i det venstre undertræ under (15). Nu sammenligner vi så 14 og 12. Da 14 er større end 12, leder vi videre i højre undertræ under (12). Det næste element vi støder på er (14), så den ønskede værdi er fundet, og søgningen er slut.

Hvis man når et blad uden at finde elementet, er søgningen også slut, og elementet findes ikke i søgetræet.



Vi var lidt heldige her at finde (14) meget hurtigt, det havde taget noget længere tid at finde (38). Men vi får et indtryk af at søgning kræver forholdsvis få sammenligninger – metoden her minder lidt om binær søgning idet vi hele tiden *halverer[[3]](#footnote-3)* antallet af resterende elementer ved at fravælge det ene undertræ.

For at vi kunne sige *halverer* uden at angive en fodnote skulle søgetræet være balanceret[[4]](#footnote-4). Det vil vi kigge nærmere på senere, indtil videre må vi acceptere at et søgetræ nogle gange kan give lange søgetider og andre gange hurtige søgetider. I *gennemsnit* vil søgetiden være i størrelsesorden , mens den i værste fald kan være , hvis elementerne tilføjes i sorteret orden eller i omvendt sorteret orden.

### Funktioner i søgetræ

#### Indsættelse

Man foretager først en søgning efter elementet. Når man når ned i bunden, indsættes det nye element som venstre eller højre undertræ alt efter om elementet der skal indsættes er mindre end eller større end det aktuelt fundne element.

Hvis elementet har samme værdi, indsættes normalt i venstre undertræ.

#### Sletning

Man foretager først en søgning efter elementet, så vi ved hvilket element der skal slettes.

Derefter finder man det største element der er mindre eller det mindste element der er større og indsætter det *i stedet for* det element der skal slettes. Når man programmerer systemet, skal man bare vælge én af de to strategier.

Det kan være en fordel at implementere en *method* swap (som betyder byt-om), der bytter to knuder rundt. Så kan man når man har fundet den knude der skal erstatte den knude der skal slettes, swappe de to knuder og derefter slette den knude der er swappet ned i bunden af træet ved at opdatere referencer til objektet.

For at få sletning til at fungere effektivt, er det en god idé at tilføje en parent-reference til knuderne, sådan at man også kan bevæge sig opad i træet.

### Opgave: Implementér et søgetræ

Implementér et søgetræ. Søgetræet skal have funktioner til søgning, til indsættelse og til sletning.

Sletning er lidt kompliceret og kan eventuelt springes over.

### Opgaver: Gennemløb af søgetræ

#### Udskrift af elementer i sorteret orden

Hvis man gerne vil have udskrevet elementerne i et søgetræ i sorteret orden, kan man gennemløbe træet sådan at man først udskriver det venstre undertræ, så det aktuelle element og derefter udskriver højre undertræ. Dette kan med fordel implementeres ved hjælp af rekursion.

#### Udskrift af søgetræets hierarki (a la file manager).

Man kan specielt til testformål have fordel af at kunne udskrive træet sådan at man ser om det er blevet konstrueret korrekt. Dette svarer meget til at udskrive en folderstruktur i et filsystem. Sørg for at hver knude også udskriver værdien af deres parent knude. Det er vigtigt for at kunne kontrollere om parent referencer er sat korrekt i opgave 4.5.5.

### Opgave: Test af søgetræ

Det kan være lidt svært at sikre sig at søgetræet virker i alle tilfælde. Ved at implementere nedenstående kode til aftestning, kommer man godt rundt i alle hjørner af søgetræet.

Udskrifterne fra 4.5.4 kan benyttes til at vise om søgetræet er korrekt ud efter hver aktion. Jeg har lavet metoderne Traverse og WriteTree. Traverse udskriver alle elementerne i sorteret rækkefølge, mens WriteTree udskriver træstrukturen sådan at man kan se om datastrukturen er intakt efter Insert/Remove.

Jeg har benyttet min egen generiske klasse SearchTree<T>, det skal erstattes af dit eget søgetræ.

static int[] Elements = {12,6,14,9,2,21,15,4,20,8,13,5,17,10,11,7,18,1,16,3,19};

static void SearchTree()

{

SearchTree<int> tree = new SearchTree<int>();

foreach (int e in Elements)

{

tree.Insert(e);

}

Console.WriteLine("Traverse tree");

tree.Traverse();

Console.WriteLine("Tree structure:");

tree.WriteTree();

foreach (int e in Elements)

{

Console.WriteLine($"tree.Remove({e});");

tree.Remove(e);

Console.WriteLine("Tree structure:");

tree.WriteTree();

}

}

# Sortering

Som vi har set ovenfor, kan et søgetræ også bruges til at foretage sortering. Ud over den sorteringsmetode, findes der en del forskellige sorteringsalgoritmer hver med deres fordele og ulemper.

De sorteringsalgoritmer der er nemmest at programmere er dem der performer dårligst ved et højt antal elementer. De mere avancerede sorteringsalgoritmer kræver ofte en lidt kompleks strukturering af data som vi så i søgetræet.

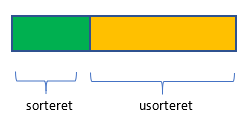
## De umiddelbare sorteringsmetoder O(n2)

De to sorteringsmetoder udvælgelsessortering og indsættelsessortering præsenteret nedenfor er ret intuitive og forholdsvis nemme at forstå. De er ikke så hurtige fordi de er bygget op med to løkker inden i hinanden der gennemløber alle data, men de fungerer fint på mindre datamængder.

### Udvælgelsessortering

Udvælgelsessortering baserer sig på en algoritme hvor en del af data er sorteret (den venstre del i figuren nedenfor). Man finder så det mindste element i den usorterede del og ”flytter ned” som nyt sidste element i den sorterede del. Denne flytning kan nemt implementeres ved at lade det element der skal flyttes bytte plads med det element der allerede står på den plads. Implementér en metode Swap() som kan bytte plads på to elementer i dit array/din liste.

I praksis bytter man det mindste element i den usorterede del med det element der står først i den usorterede del. Nu kan vi udvide det sorterede område med det nye element som vi har flyttet på plads.



Denne proces fortsætter indtil den usorterede del er tom.

Se eventuelt [denne video](https://youtu.be/So8kx3vLfjk) for et konkret eksempel på implementation af udvælgelsessortering.

### Opgave

Implementér et program der kan sortere en mængde tal ved hjælp af udvælgelsessortering.

### Tidskompleksitet

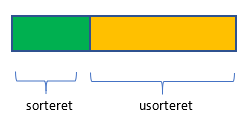
Hvis vi øger antallet af elementer der skal sorteres med 1, så voksers tidsforbrug med O(n).

Hvis vi øger antallet af elementer der skal sorteres med n, so vokser tidsforbruget med n\*O(n) -> O(n2).

### Indsættelsessortering

Indsættelsessortering minder en del om udvælgelsessortering. Vi har samme opdeling i en sorteret del og en usorteret del.

Processen er dog lidt anderledes. Har tages det første element i den usorterede del og flyttes på plads i den sorterede del. Dette kan nemt gøres ved at swappe indtil elementet er på plads. Når elementet er på plads, er den sorterede del vokset med ét element og den usorterede er blevet ét element mindre.



Denne proces fortsætter indtil den usorterede del er tom.

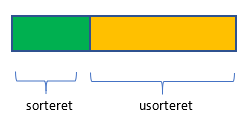
### Opgave

Implementér et program der kan sortere en mængde tal ved hjælp af indsættelsessortering.

### Bubble sort

De fleste programmører har hørt om bubble sort, og de har sikkert også hørt at det ikke er nogen hurtig sorteringsmetode. Den er ikke helt så intuitiv nem at forstå som indsættelses- og udvælgelsessortering, men den er ret nem at implementere.

Metoden er at gennemløbe alle elementer og bytte alle naboelementer der står i forkert orden. Ved hvert gennemløb vil man få det mindste element i den usorterede del helt til venstre. Det vi sige at vi i næste gennemløb kan undlade at håndtere dette element.



Alt i alt minder bubble sort altså en del om udvælgelsessortering fordi det mindste (eller største afhængigt af om sorteringen er i stigende eller faldende orden) element efter hvert gennemløb er blevet placeret korrekt.

Se [Wikipedia](https://da.wikipedia.org/wiki/Boblesortering) for en lidt grundigere forklaring.

### Opgave

Implementér et program der kan sortere en mængde tal ved hjælp af bubble sort metoden.

### Opgave

Gør dine sorteringsmetoder generiske, dvs. man skal kunne sortere arrays (eller lister) af alle sorterbare typer (dem der implementerer IComparable).

## Optimeret sortering ()

Ved store datamængder vokser den tid det tager at sortere ret hurtigt. De to algoritmer indsættelsessortering og udvælgelsessortering har en udførselstid af størrelsesorden , hvor n er antallet af elementer der skal sorteres. Det hænger sammen med at vi skal sammenligne hvert element med alle[[5]](#footnote-5) andre elementer for at finde elementets rette position. Man kan også se på det sådan at hvis vi tilføjer ét element til antallet af elementer der skal sorteres, så vokser tiden for sortering med fordi dette element potentielt skal sammenlignes med alle elementer.

Det er altså ret dyrt at sortere store datamængder med -algoritmer. Det kan man overbevise sig om ved at betragte graferne i afsnittene 3.3 og 3.4. I afsnit 6 skal vi måle tiderne det tager at køre de forskellige algoritmer, der kommer vi til at mærke forskellen på effektive og mindre effektive søgealgoritmer.

Der findes en del sorteringsalgoritmer der er såkaldt logaritmiske, det vil sige at de har en udførelsestid af størrelsesorden . Nogle af dem som gennemsnitlige tider, andre som worst case tider.

### Sortering ved hjælp af søgetræ (O(n∙log(n)/O(n2))

Når man bruger et søgetræ til sortering skal man udføre følgende handlinger

1. Indsæt alle elementer i søgetræet. Da søgetræets dybde er af *gennemsnitlig* størrelsesorden log(n) og indsættelse skal udføres for hvert element, bliver tidsforbruget for den totale indsættelse af størrelsesorden .
2. Derefter skal alle elementer udskrives ved et gennemløb af træet. Da alle træets knuder besøges én gang, er dette af størrelsesorden .

Tallene skal i princippet lægges sammen, så gennemløbet af søgetræet bliver mindre væsentlig i det samlede billede – derfor vokser tidsforbruget i størrelsesorden når n bliver stor. Bemærk at sortering ved hjælp af søgetræ kun gennemsnitligt har en tidskompleksitet på , hvis søgetræet er dårligt balanceret bliver tidskompleksiteten i værste tilfælde .

### Opgave: Sortering med søgetræ

Konstruér et program der kan indsætte en række tal i et søgetræ og derefter hente dem ud i et array i sorteret rækkefølge.

## Divide and conquer metoder (O(n∙log(n))

Fælles for nogle af de mere avancerede sorteringsalgoritmer er at de bruger et princip kaldet *divide and conquer* (del og hersk). Det handler om at dele problemet op i mindre problemer som løses hver for sig, hvorefter resultaterne af de løste delproblemer kombineres til en samlet løsning. Hvis man benytter rekursion opnår man ofte den største læsbarhed af disse programmer. Det hænger sammen med at delproblemerne skal løses på samme måde som det overordnede problem, så man kalder samme metode på delproblemerne.

Resultatet vil være en programafvikling hvor problemet opdeles i mindre og mindre opgaver, indtil opgaverne er så små at de er meget simple at løse. Derefter kombineres løsningerne indtil alle delløsninger er kombinerede og opgaven er løst.

### Quicksort (O(n∙log(n)/O(n2))

Sortering ved hjælp af søgetræ kræver at man opbygger en søgetræsstruktur. Ofte har man dog data i et array eller en liste, og så er det nemmere hvis man bare kan sortere elementerne i dette array, ligesom vi gjorde med indsættelsessortering og udvælgelsessortering. [QuickSort](https://en.wikipedia.org/wiki/Quicksort) eller QSort som algoritmen også kaldes, er en algoritme der i gennemsnit har en tidskompleksitet af størrelsesorden . Worst case er dog O(n2). Tanken bag algoritmen er at hvis man nu havde medianen (det midterste element hvis de var sorterede), så kunne man flytte alle de elementer der var mindre end medianen over til venstre og alle der var større over til højre, og så kunne man gentage denne øvelse på de to halvdele rekursivt. Vi kan dog ikke finde medianen uden at sortere elementerne, så vi er lidt endt i en deadlock. I stedet kan vi forsøge at ”gætte” medianen, så fungerer det.

Beskrivelse af algoritmen:

1. Vælg et tal der ligger inden for intervallet af værdier i arrayet. Det kan være en god idé at vælge et tal der står midt i arrayet sådan at man sikrer sig at det flytter sig (og man dermed vælger et nyt tal næste gang). Tallet kaldes almindeligvis for [Pivot](https://en.wikipedia.org/wiki/Quicksort).
2. Split arrayet op sådan at alle tal mindre end Pivot står til venstre og alle tal højere står til højre.
3. Foretag samme sortering (brug recursion) på hver af de to dele.
4. Når der kun er ét eller to elementer kan man vælge en simpel metode - swap elementerne hvis de står i uorden.

#### Lad os se

Lad os prøve at gennemføre et eksempel som om vi er computeren. Vi starter altså med nogle tal der skal sorteres. Vi vælger et pivot-tal som afgør hvilke tal der skal forstås som store og hvilke der skal forstås som små. Lad os i dette eksempel altid vælge det første tal. Pivot bliver altså i første omgang 12.

Vi løber gennem tallene fra venstre og højre og bytter rundt når begge tal står på den forkerte side. Vi benytter variablerne L og R til at pege henholdsvis til højre og til venstre.

12,6,14,9,2,21,15,4,20,8,13,5,17,10,11,7,18,1,16,3,19

L R

Pr. definition siger vi at tal der er lig pivot skal være i den højre side af arrayet - så sikrer vi os at tallet ikke bliver stående. Da 12 står forkert, flytter vi R indtil den også udpeger et tal der står forkert, det gør den ved tallet 3.

12,6,14,9,2,21,15,4,20,8,13,5,17,10,11,7,18,1,16,3,19

L R

Da begge tal nu står forkert, kan vi bytte dem rundt. Så ved vi de kommer til at stå rigtigt, så vi kan flytte L og R mod hinanden.

3,6,14,9,2,21,15,4,20,8,13,5,17,10,11,7,18,1,16,12,19

L R

Vi tæller L op til den udpeger et element der hører til på højre side. Derefter tæller vi R ned indtil den udpeger et element der skal stå til venstre.

3,6,14,9,2,21,15,4,20,8,13,5,17,10,11,7,18,1,16,12,19

L R

Sådan fortsætter det.

3,6,1,9,2,21,15,4,20,8,13,5,17,10,11,7,18,14,16,12,19

L R

3,6,1,9,2,21,15,4,20,8,13,5,17,10,11,7,18,14,16,12,19

L R

3,6,1,9,2,7,15,4,20,8,13,5,17,10,11,21,18,14,16,12,19

L R

3,6,1,9,2,7,11,4,20,8,13,5,17,10,15,21,18,14,16,12,19

L R

3,6,1,9,2,7,11,4,20,8,13,5,17,10,15,21,18,14,16,12,19

L R

3,6,1,9,2,7,11,4,10,8,13,5,17,20,15,21,18,14,16,12,19

L R

3,6,1,9,2,7,11,4,10,8,13,5,17,20,15,21,18,14,16,12,19

L R

3,6,1,9,2,7,11,4,10,8,5,13,17,20,15,21,18,14,16,12,19

R L

2, 1, <> 6 9, 3, 7, 11, 4, 10, 8, 5

1, 2

Så er alle elementer blevet behandlet og udvælgelsesdelen er færdig. Nu udpeger R det sidste element i den del med de små værdier og L udpeger den første af de store elementer.

Nu fortsættes sorteringen ved at kalde samme sorteringsmetode rekursivt på de to ”halve” arrays.

#### Hvordan ser vi det er Divide and conquer?

Princippet omkring divide and conquer kommer her i spil ved at opgaven deles i to opgaver, sortér de ”små” tal for sig og de ”store” tal for sig. Det med at kombinere løsningerne foretages i dette tilfælde inden løsningen af delproblemerne - opdelingen inkluderer i sig selv kombineringsdelen da alle tal i venstre del af arrayet vil være mindre end alle tal i højre del, og hvis hver del i sig selv er sorteret, vil det færdige resultat være et sorteret array.

#### Tidskompleksitet

Quicksort er kun logaritmisk i gennemsnit. I QuickSort kan vi ikke i konstant tid vælge det midterste tal, men vi må vælge en værdi arbitrært[[6]](#footnote-6). Det betyder at QuickSort i værste tilfælde kan have tidskompleksiteten - hvis vi konsekvent er uheldige med valget af pivot-værdier.

Hvis vi f.eks. bruger den første værdi som pivot-værdi som i ovenstående eksempel, så vil sortering af et array der i forvejen er sorteret have tidskompleksitet , da vi hele tiden vil dele arrayet op i to dele hvor den ene del indeholder ét element, mens den anden indeholder resten. Antallet af opdelinger vil derfor være , og da processen med at flytte alle elementer til venstre eller højre side kræver gennemløb af arrayet ( skal udføres for hver opsplitning, bliver den samlede tid .

QuickSort er altså hurtig til at sortere de fleste datamængder, men i enkelte tilfælde kan QuickSort være langsom. Der findes nogle sorteringsalgoritmer som er logaritmiske også i værste tilfælde, f.eks. MergeSort og HeapSort (sortering ved hjælp af såkaldt heap – på dansk bunke eller hob). Dem skal vi se på senere.

#### Memoryforbrug

QuickSort er meget effektiv i forhold til brug af hukommelse. Da der hele tiden arbejdes på det samme array af elementer hvor elementerne flyttes rundt, skal der ikke allokeres hukommelse specifikt til sorteringen. Det betyder at QuickSort ofte i praksis er meget effektiv også i hastighed fordi der ikke undervejs vil skulle foretages allokering og sletning af hukommelse.

### Opgave: QuickSort

Implementér QuickSort til at sortere en række tal. Til aftestning af algoritmen kan du lade dig inspirere af det testprogram vi lavede til søgetræet i afsnit 4.5.5.

Hjælp til implementation kan eventuelt findes i denne video hvor [implementation af QuickSort](https://youtu.be/0nWJ0vLqFQI) gennemgås.

### Opgave: Generisk QuickSort

Implementér QuickSort<T> til generelt at sortere klasser der implementerer interfacet IComparable.

### MergeSort O(n∙log(n))

Flettesortering foregår ved at man hele tiden deler det er skal sorteres op i to lige store halvdele, sorterer dem hver for sig og derefter fletter de to sorterede dele sammen. MergeSort implementeres ofte ved hjælp af rekursion. Det gør programmet forholdsvis nemt at læse og forstå.

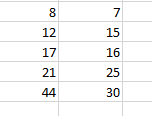
For at MergeSort kan fungere, er man nødt til at oprette nye arrays når man fletter fordi MergeSort ikke som de fleste andre sorteringsalgoritmer benytter sig af en swap-funktion.

#### MSort

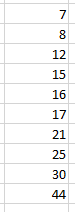
Den rekursive metode MSort (find gerne på dit eget navn) tager en liste (array) som parameter. Den fungerer ved at kalde sig selv rekursivt indtil der kun er ét element i listen. Det element returneres. Selve sorteringen foregår ved at metoden Merge fletter to sorterede lister sammen til en ny sorteret liste.

#### Merge

Flettemetoden skal tage to lister (arrays) og returnere en ny liste med resultatet af sammenfletningen.



Returværdi:



#### Memory forbrug

Da man løbende skal kopiere arrays både i forbindelse med divide-delen og merge, bruger MergeSort en del hukommelse. Imidlertid frigives hukommelsen løbende, sådan at det samlede forbrug på intet tidspunkt er højere end .

#### Tidskompleksitet worst case (O(n∙log(n))

Lad os betragte det kaldstræ MergeSort giver anledning til.

##### Opdeling

Under opdelingsfasen deles arrayet i to lige store halvdele igen og igen, indtil størrelsen af arrayet er 1.

12,6,14,9,2,21,15,4,20,8,13,5,17,10,11,7,18,1,16,3,19

12,6,14,9,2,21,15,4,20,8 | 13,5,17,10,11,7,18,1,16,3,19

12,6,14,9,2 | 21,15,4,20,8 | 13,5,17,10,11 | 7,18,1,16,3,19

12,6 | 14,9,2 | 21,15,4,20,8 | 13, 5 | 17,10,11 | 7,18,1 | 16,3,19

12|6 | 14,9 | 2 | 21|15 | 4,20 | 8 | 13|5 | 17,10 | 11 | 7,18 | 1 | 16,3 | 19

12|6 | 14|9 | 2 | 21|15 | 4|20 | 8 | 13|5 | 17|10 | 11 | 7|18 | 1 | 16|3 | 19

Sidste linje er lidt misvisende ved de elementer der ikke kan opdeles. Der er opdelingen fra forrige linje bare gentaget nedenunder. Det er udelukkende af visuelle grunde, programmet skal ikke forsøge at opdele helt opdelt data.

Antallet af elementer der håndteres i hver linje er n. Da hver opdeling halverer størrelsen af opdelingerne, vil der være opdelinger. Det giver en samlet tidskompleksitet for opdelingen på .

##### Fletning

Efter opdelingen flettes arrayene sammen. Det er faktisk flettedelen der sørger for den egentlige sortering.

Sammenfletningerne er illustreret i rammen nedenfor. I hver række håndteres hvert element én gang - dermed har hver række en tidskompleksitet på .

12|6 | 14|9|2 | 21|15 | 4|20|8 | 13|5 | 17|10|11 | 7|18|1 | 16|3|19

6,12 | 9,14|2 | 15,21 | 4,20|8 | 5,13 | 10,17|11 | 7,18|1 | 3,16|19

6,12 | 2,9,14 | 15,21 | 4,8,20 | 5,13 | 10,11,17 | 1,7,18 | 3,16,19

2,6,9,12,14 | 4,8,15,20,21 | 5,10,11,13,17 | 1,3,7,16,18,19

2,4,6,8,9,12,14,15,20,21 | 1,3,5,7,10,11,13,16,17,18,19

1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21

Antallet af linjer med sammenfletninger er det samme som antallet af linjer med opsplitning, og hermed får vi tidskompleksiteten for sammenfletningerne til at være .

### Opgave: MergeSort

Implementér MergeSort som en metode der kan sortere tal.

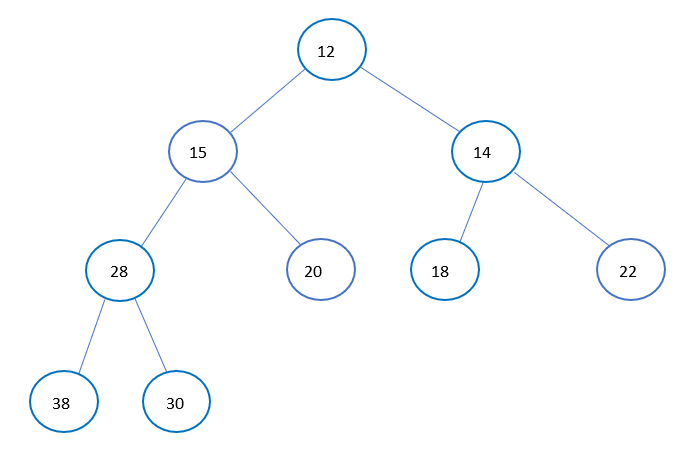
### Opgave: Generisk MergeSort

Implementér MergeSort<T> som en generisk metode der kan sortere elementer af alle typer der implementerer interfacet IComparable.

## Heap (bunke/hob) - en anden tilgang til sortering

En [bunke](https://en.wikipedia.org/wiki/Heap_(data_structure)) er et binært træ ordnet på en særlig måde. I modsætning til et søgetræ holder bunken ikke den færdige sortering. I stedet er bunken ordnet således at hver knudes to underknuder har højere værdier. På den måde opnår man at det mindste element vil være placeret i træets rod.

Om en bunke gælder der at hver knudes undertræer har samme højde eller det venstre undertræ er præcis 1 højere end det højre.



En af fordelene ved en bunke frem for et søgetræ er at bunken altid kan holdes balanceret, det vil sige at der ikke vil være manglende grene i træet, kun på allernederste niveau (bladene) vil der mangle knuder (fordi antallet af elementer kun sjældent vil passe til et perfekt træ (at antallet af elementer er en to-potens minus 1).

Da roden i knuden altid er største/mindste værdi, er denne datastruktur rigtig god at implementere en såkaldt prioritetskø, det vil sige en kø hvor man vælger den med højest/lavest ”score”. Man kan sammenligne med at en skadestue vælger at behandle de alvorlige uheld først frem for at behandle patienterne i den rækkefølge de ankommer til skadestuen.

### Operationer på en bunke

Som vi skal se senere, så kan man benytte en bunke til at implementere en effektiv sorteringsalgoritme. Man kan også bruge den i andre sammenhænge hvor man er interesseret i den mindste (eller største) værdi - det kunne f.eks. være i forbindelse med at beregne den korteste afstand mellem to byer.

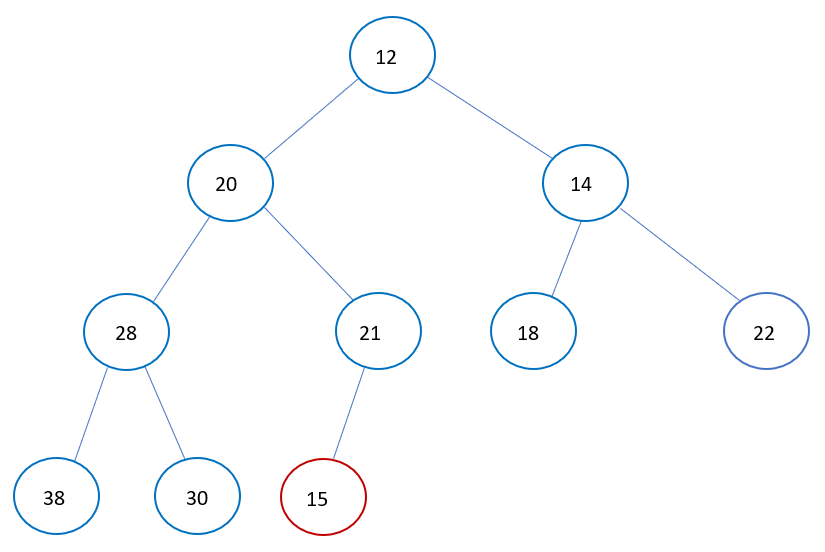
Vi skal nu se på hvordan man kan indsætte elementer og fjerne rodelementet[[7]](#footnote-7) i en bunke samtidig med at den hele tiden holdes balanceret.

For at kunne implementere disse to operationer får vi brug for at kunne finde bunden af træet. Dette er det mest komplicerede ved en bunke, så vi gennemgår først insert og remove ved intuitivt at bestemme bunden visuelt. Senere går vi mere i dybden med hvordan vi skal lave et stykke kode der kan bestemme den nye bund ved insert og remove.

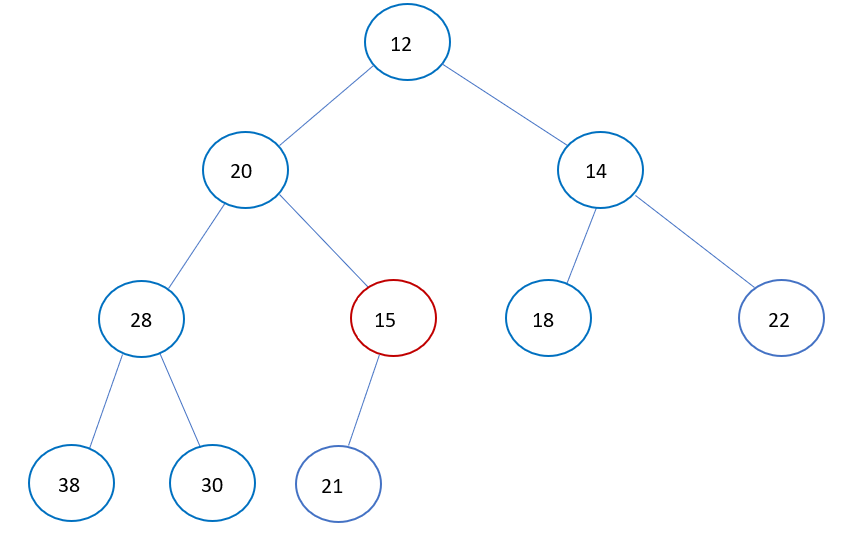
#### Indsætte elementer i bunke

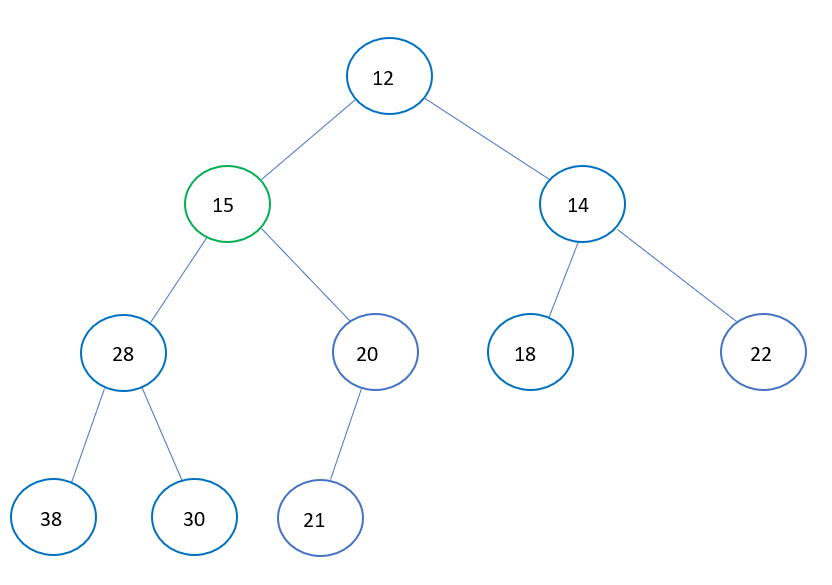
Hvis bunken er tom, indsættes det nye element som bunkens knude.

Ellers: Vi sætter først det nye element ind i bunden af bunken. Resultatet er et balanceret træ, men træet overholder nu ikke længere forudsætningerne for at være en bunke.



Vi swapper op indtil knuden er på plads.





#### Tidskompleksitet

Da vi højest swapper i træets højde bliver tidskompleksiteten for at swappe et element på plads[[8]](#footnote-8).

#### Implementation af swap

Det kan være lidt svært at holde styr på alle detaljer når man skal swappe to knuder. Det skyldes at der er mange referencer der skal opdateres.

##### Den nemme løsning

Man kan bare ”snyde” og bytte værdierne på de to knuder om, sådan at knuderne som sådan ikke bytter plads, man bytter bare værdierne om mellem de to knuder. Det vil nok i praksis være den metode man oftest vil anvende.

Man kan tænke at det er lidt snyd fordi knuderne ikke byttes om, hvad nu hvis en knude bestod af mange forskellige informationer som man således skulle flytte rundt med? Det ville også være nemt at løse ved simpelt hen at erklære en klasse til at holde informationen og så lade knudernes værdi være objekter af denne klasse.

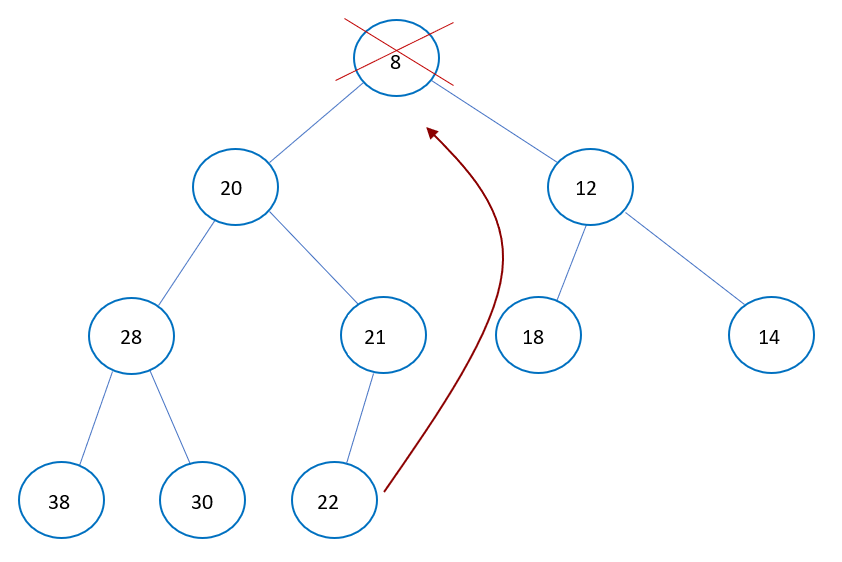
##### Den svære løsning (hvis man har mod på lidt referencegymnastik)

Til nogle opgaver kan man dog være nødt til at bytte rundt ved at ændre objekternes referencer - det gælder i de tilfælde hvor man har referencer udefra til knuder i træet.

Alternativt kan man bytte to knuder rundt ved at flytte rundt med deres referencer. Det kræver man har overblik og styr på detaljerne. I nogle tilfælde kan man have fordel af at implementere Swap på denne måde, f.eks. hvis knudens værdi skal kende selve knuden (det er lidt kryptisk, men det bliver aktuelt i forbindelse med grafer i afsnit 10). Se følgende beskrivelse: [Swap med referencer](https://drive.google.com/file/d/1bqACvvUgH8P84QA2XrCIn7bGQrK74pXs/view?usp=sharing).

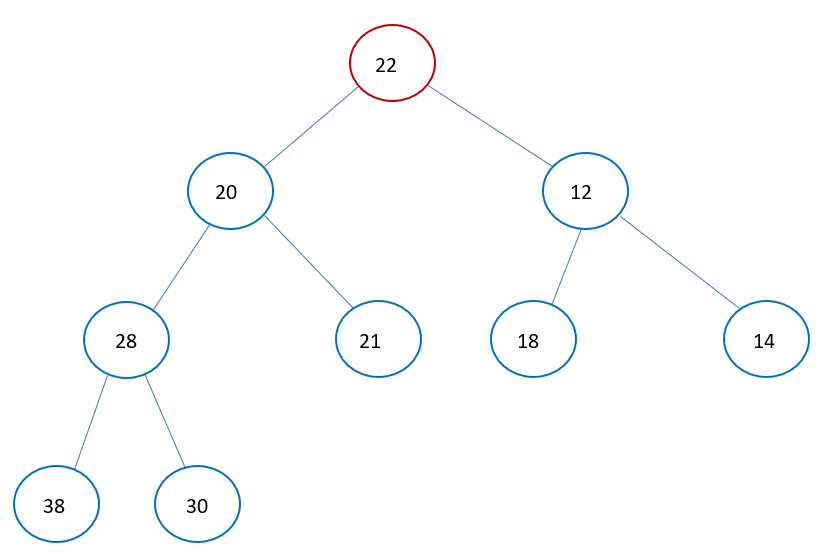
#### Tage roden ud af bunke

Det er forholdsvis simpelt at slette roden fra en bunke.

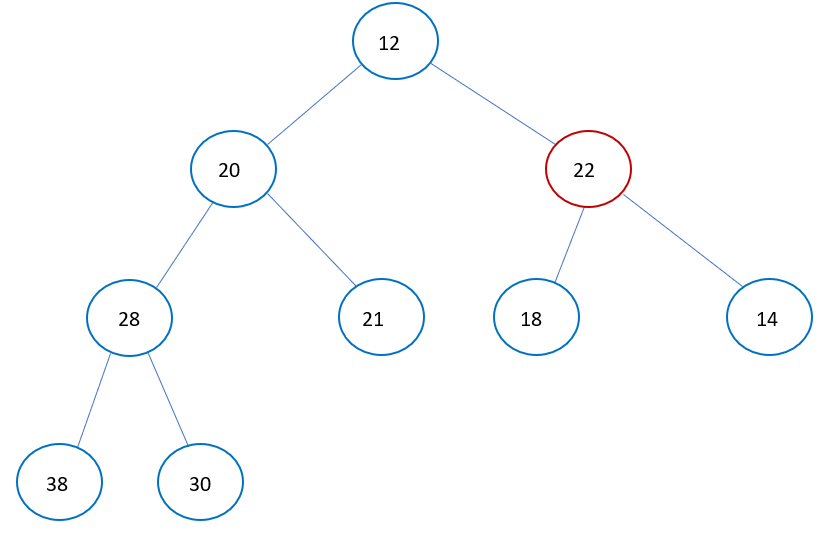


Vi fjerner først bare roden og flytter det sidste element op på dets plads.

Det giver følgende træ:

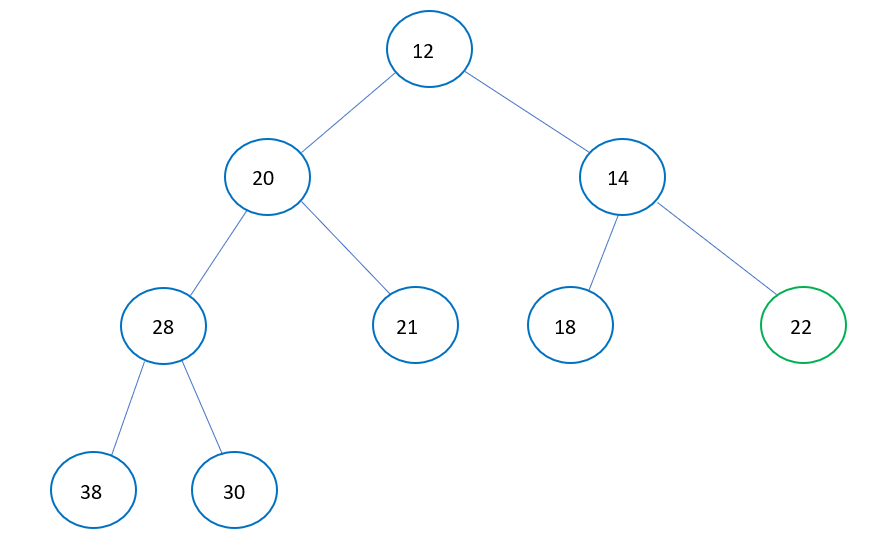


Bemærk at vi på den måde bevarer at træet stadig er balanceret, men nu er træstrukturen ikke længere en bunke. Men det kan ordnes ved at swappe (22) på plads sådan at kravene til en bunke igen er opfyldt. Umiddelbart kan vi vælge at bytte mod venstre eller mod højre. Hvis vi bytter mod venstre, har vi dog et problem fordi 20 ikke er mindre end 12. Vi bytter derfor med den mindste af de to underelementer.



22 står stadig forkert.

Vi bytter med *det mindste* af de to underknuder:



Nu har vi igen en korrekt, balanceret bunke.

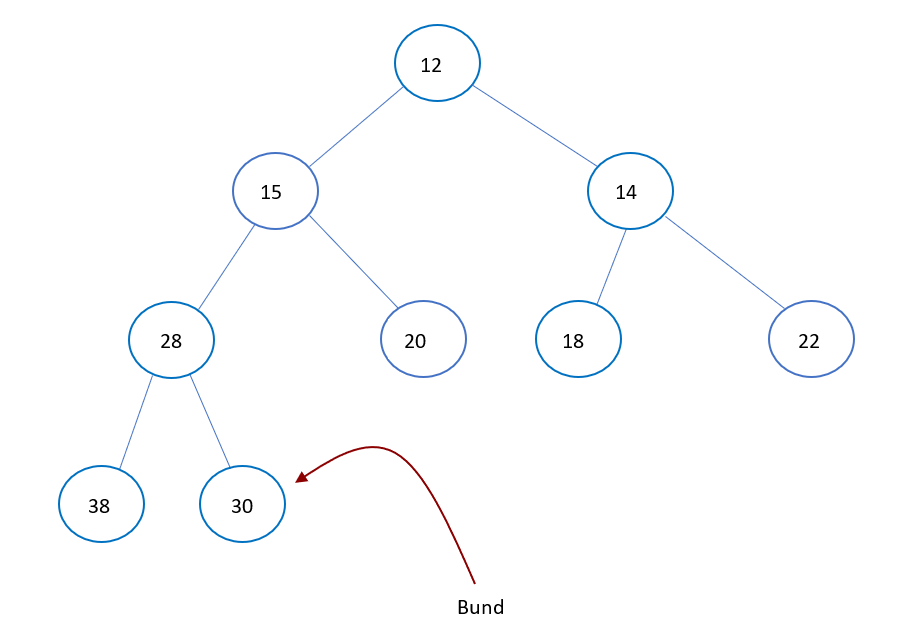
##### Tidskompleksitet

At tage roden ud af bunken har en tidskompleksitet på . Dette kan vi se fordi det antal operationer vi foretager vokser med højden af træet, hver gang vi vælger at bytte til højre eller venstre halverer vi antallet af mulige ombytninger.

#### Hold styr på ”bunden af bunken” ved indsættelse

Det kan være lidt svært at holde styr på præcis hvor i træet nye elementer skal indsættes. En hjælp kan være at holde en reference til det sted hvor bunden af bunken er. Ud fra den kan vi finde ud af hvor det næste element skal indsættes.

Men det er alligevel ikke helt simpelt at finde det nye indsættelsessted ud fra den nuværende *bund* af bunken, lad os prøve at se på det.



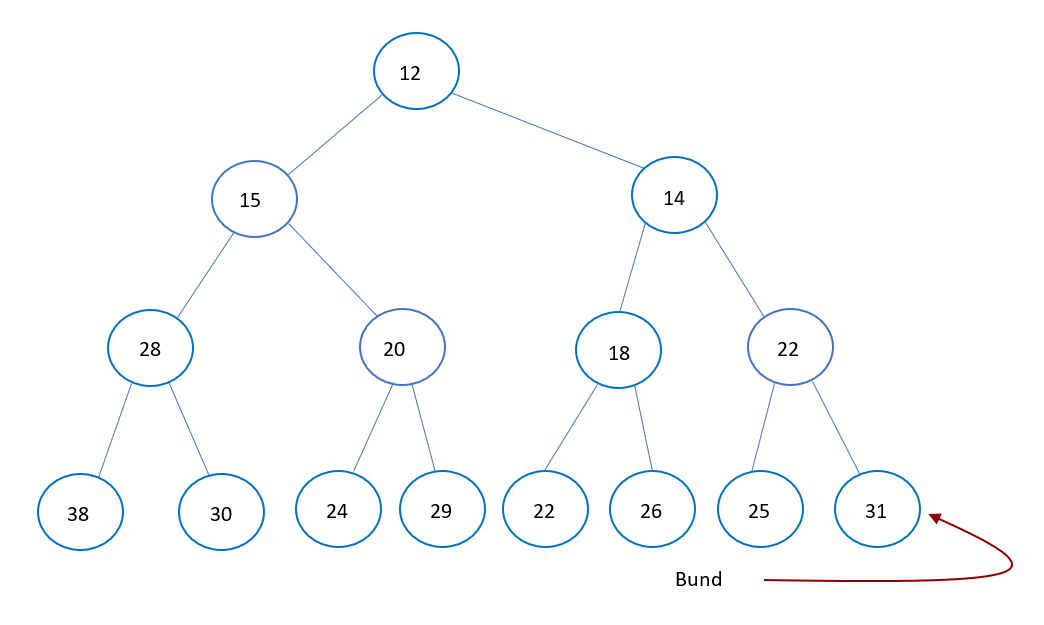
Hvis vi skal indsætte en ny knude, hvor skal det så være?

1. Da den nuværende bund kan betegnes som et af træets ”blade”, altså det yderste niveau, skal den næste knude placeres på samme niveau hvis det er muligt.
2. Vi undersøger derfor først om parent knude (28) har plads til flere blade under sig. Havde bunden været knuden med værdien 38 (og den nuværende knude med værdien 30 derfor ikke havde været en del af træet) ville der være plads til et blad mere under knuden.
3. I dette tilfælde har parent knude (28) ikke plads til flere blade, så vi må finde en anden gren højere oppe i træet. Vi går op til (28)’s parent (15).
4. Da vi kommer fra venstre undergren til (15) og vi kommer fra bunden, ved vi at det nye element skal indsættes i højre undertræ til (15). Vi går derfor til højre til knuden (20). Herefter følger vi venstre undergren gentagne gange (i dette tilfælde nul gange) indtil vi når laveste niveau. Vi ved også at det laveste niveau vil være ét niveau højere oppe i træet end vores eksisterende bund (30) sådan at der er plads til et nyt blad i venstre undertræ.

Algoritmen er altså kort beskrevet at hvis man ikke kan indsætte den nye knude som højre undertræ til nuværende parent, så gå et ekstra niveau op indtil der er en knude til højre som ikke er den man kommer fra. Følg derefter dennes underknuder til hele tiden til venstre, indtil der ikke er flere knuder under. Indsæt derefter knuden som venstre underknude.

En anden måde at betragte denne algoritme på er at forestille sig at man fysisk bevæger sig rundt i træet i f.eks. en bil. Hvis man starter fra (30) og kører opad og hele tiden holder mod højre, vil man komme til (28), derefter (15) og endelig (20). Bemærk, det højre set fra føreren af bilen hele tiden.

Et specialtilfælde opstår når der ikke er mere plads i træet på samme niveau som den nuværende bund. Det opdager man når man leder op gennem træet efter en ledig højre underknude, men da vi selv kommer fra den højre underknude når vi op til roden uden at finde en.



Nu ved vi at den nye knude skal indsættes som underknude til den knude vi får ved at følge alle venstre-underknuderne helt ned til der ikke er flere venstre-underknuder.

Når den nye knude er indsat, sættes den som bunkens *bund*.

#### Holde styr på bunden af bunken ved sletning

Når man sletter

##### Tidskompleksitet

At finde den nye bund når vi indsætter eller sletter kræver at vi bevæger os op i træet indtil det er muligt at bevæge os nedad igen. I værste tilfælde kommer vi helt op til roden før vi kan begynde at gå nedad. Det betyder at vi i værste tilfælde skal gennemløbe træets højde to gange. Da træets højde vokser med , bliver det tidskompleksiteten for at finde den nye *bund*.

### Bunkesortering/hobsortering – heapsort O(n∙log(n))

Lidt tilsvarende sortering ved hjælp af søgetræ, så fung erer heapsort ved først at indsætte alle elementerne i bunken og derefter hente dem ud ét efter ét.

At indsætte et nyt element har tidskompleksitet da følgende skal udføres:

1. Find knude til indsættelse (to gange træets højde – i værste tilfælde helt op til roden fra bunden og helt ned til den nye bund): .
2. Indsæt ny knude (træets højde - knuden indsættes og swappes op – vi værste tilfælde helt op til roden): .
3. Holde styr på træets *bund*.

At indsætte alle (n) elementer får derfor tidskompleksiteten .

### Opgave: HeapSort (bunkesortering)

Implementér en løsning af heapsort med en bunke implementeret ved hjælp af et træ bestående af knuder og referencer.

### En bunke i et array

Som beskrevet ovenfor, så er den store fordel ved en bunke at den kan holdes balanceret hele tiden. En konsekvens af dette er at det giver os mulighed for at holde bunken i et array (eller en List<>) i stedet for en træstruktur bygget op med referencer.

Mange ting bliver lettere i array’et.

* Root er det første element
* LastElement er det sidste element
* parent findes ved en simpel beregning
* Left og right findes ved en simpel beregning
* Ombytning foregår ved at bytte værdierne af to indekserede objekter

Da vi overordnet set benytter samme metoder som i afsnit 5.4.2 bliver tidskompleksiteten ved heapsort med array-løsningen også den samme, altså .

### Opgave

Implementér en løsning af heapsort ved hjælp af en bunke i et array (eller en List<>).

# Teori versus virkelighed

Nu har vi set en del forskellige sorteringsalgoritmer og regnet på tidskompleksiteten. Men et er teori, et andet praksis.

Nu skal vi prøve at måle hvor hurtige vores sorteringsprogrammer er i virkeligheden. Dette gøres ved at foretage et tidsstempel før sorteringen og et nyt efter og trække de to fra hinanden.

## Opgave: Tilføj tidsmålinger til søgninger

Tilret dit program sådan at det udskriver antal millisekunder hver sortering har taget. Det vil være en god idé at rette programmet til sådan at det foretager samme sortering med alle de forskellige sorteringsmetoder.

## Opgave: Udfør tidsmålinger

Foretag flere målinger (mindst fem) af alle sorteringsalgoritmer. Målingerne skal foretages på forskellige størrelser rådata. Nogle af sorteringsalgoritmerne bliver hurtigt meget langsomme, slå dem fra ved store mængder rådata så du ikke skal sidde og vente i rigtig lang tid. Det største antal elementer i en sortering bør være over en million sådan at vi kan føle os sikre på der ikke sker spændende ting ved store antal elementer som vi ikke ser ved små antal.

Indlæs tallene fra opgave i Excel, sådan at både antal elementer der skal sorteres og sorteringstiden fremgår, f.eks. sådan:



Gør det samme med alle sorteringsmetoderne sådan at du får en række for hver metode. Faktisk kan det være nemmest at lave kolonnerne én ad gangen hvis du for hver kørsel af programmet får data fra alle sorteringsmetoder.

Eksempel på udskrift:

Tidsforbrug: (2 elementer)

SearchTree: 31,3195

InsertionSort: 0,2481

SelectionSort: 0,1582

QuickSort: 0,3091

MergeSort: 0,3757

HeapBinaryTree: 2,0786

HeapArray: 1,7917

HeapList: 1,4249

Det er nemt at benytte sådan en udskrift fra programmet til at selecte alle tallene og lægge dem ind i Excel til videreforarbejdning.

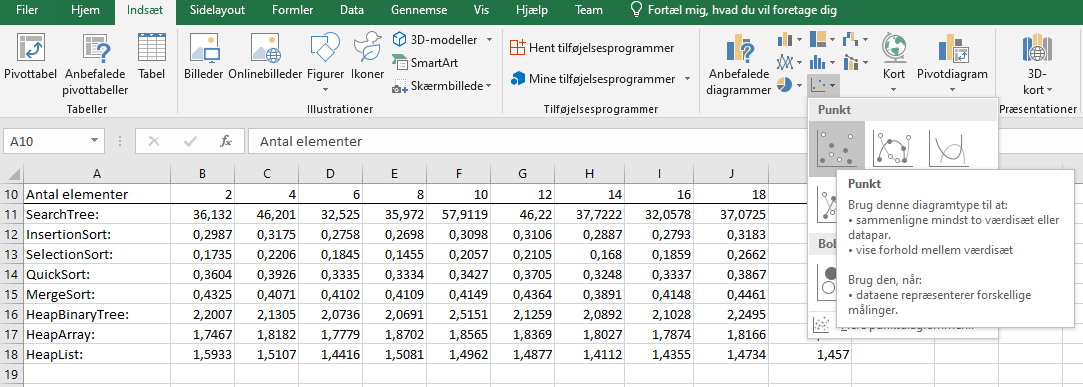
**Opgave**: Hvilke algoritmer er hurtigst ved små mængder data? Hvilke algoritmer er hurtigst ved store mængder data?

**Opgave**: Er svaret overraskende? Hvorfor/Hvorfor ikke?

**Opgave**: Hvilke årsager kan der være til at netop *de* er de hurtigste algoritmer? Hvad gør de andre langsommere?

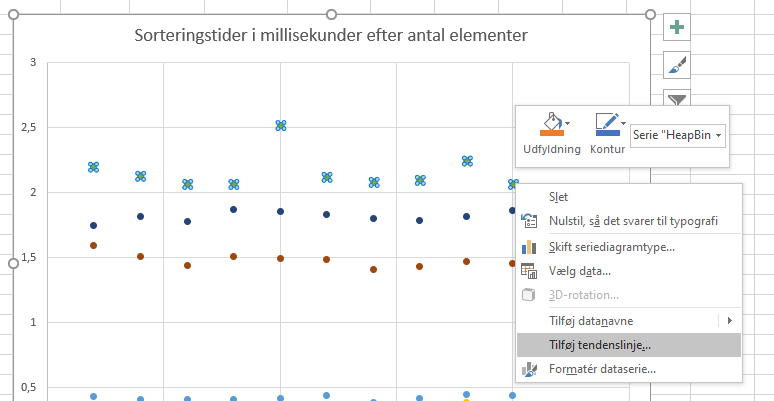
## Opgave Grafer

Indsæt nu tallene i en graf. I Excel gøres det ved at markere alle tallene og vælge Indsæt punktserie:



Bemærk: Mine tal ovenfor er baseret på sortering af kun få elementer. Jeg har bevidst vist mine tider på alt for få elementer for ikke at mine personlige resultater skal påvirke den måde du arbejder med tallene på. De små datamængder i eksemplet ovenfor er ikke retvisende for hvilke sorteringsalgoritmer der er mest effektive ved et stort antal elementer, så find tiderne for både få og mange elementer (op til i hvert fald 1.000.000 elementer).

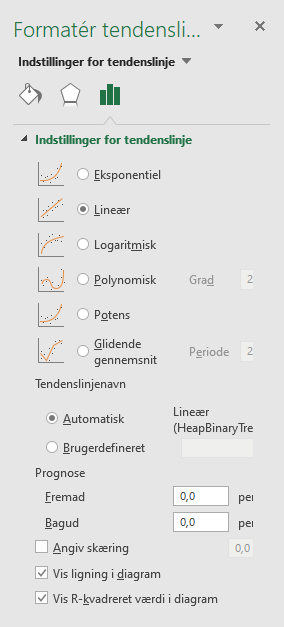
Nu får du alle tallene i én graf, men Excel hjælper med at holde styr på punkterne ved at benytte farvekoder:



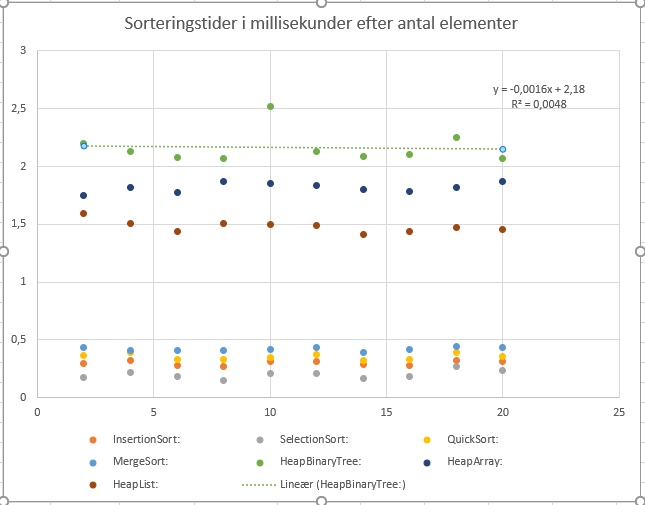
Når man indsætter en tendenslinje, vil Excel finde den linje - ellere mere generelt kurve - der bedst beskriver de punkter der er plottet ind i koordinatsystemet.

Under indstillinger for tendenslinje kan man vælge forskellige funktionstyper (kurver). Vi kommer til at arbejde med de to funktionstyper polynomisk og lineær.

Som vi så i afsnit 0, vil vi forvente at nogle af sorteringsalgoritmerne opfører sig som , mens andre opfører sig som . Hvis man vælger Polynomisk af grad 2, får man en -graf. Man kan ikke vælge , men som vi så i afsnit 3.4, så opfører sig næsten ligesom (det vil sige lineært) når bliver stor. Derfor kan vi vælge en lineær graf til de algoritmer hvor vi forventer en tidskompleksitet på .



Et eksempel på en lineær tendenslinje ses nedenfor:



Ligningen er en matematisk beskrivelse af den rette linje der bedst beskriver data. Tallet fortæller hvor god linjen beskriver tallene. Hvis er 1, beskriver linjen tallene 100%. I dette eksempel fortæller at linjen kun meget dårligt beskriver de observerede tal. I dette tilfælde kan vi konkludere at observationerne ikke afhænger lineært af opgavens størrelse. Specielt sorteringstiden ved 10 elementer skiller sig ud.

Det hænger sammen med at antallet af elementer der skal sorteres her er så lille at selve sorteringsopgaven ikke har så meget indflydelse på hvad computeren bruger tid på som de andre programmer der kører. For eksempel foretages garbage collection fra tid til anden, hvilket (kortvarigt) kan påvirke computerens performance. Det er sandsynligvis det der har påvirket udførelsestiden ved sortering af 10 elementer.

**Opgave**: Foretag nu denne øvelse med alle punktserierne. For nogle af punktserierne har du måske ikke tider for de største mængder data, men det betyder bare at der er færre punkter.

**Opgave**: Passer sorteringsalgoritmernes tidsforbrug med det du havde forventet?

**Opgave**: QuickSort har jo kun gennemsnitligt en tidskompleksitet på - worst case er . Kan du opstille et (ja, jo helst i flere størrelser for at vi kan undersøge det) datasæt til sortering der gør QuickSort til ? QuickSort har jo kun gennemsnitligt en tidskompleksitet på . Kan du opstille et (ja jo helst flere for at vi kan undersøge det) datasæt til sortering der gør QuickSort til

**Opgave**: Prøv at implementere de to hurtigste sorteringsmetoder både som generiske metoder der kan sortere alle arrays der implementerer ICompare. Ændrer det på billedet af hvilken af de to der er hurtigst?

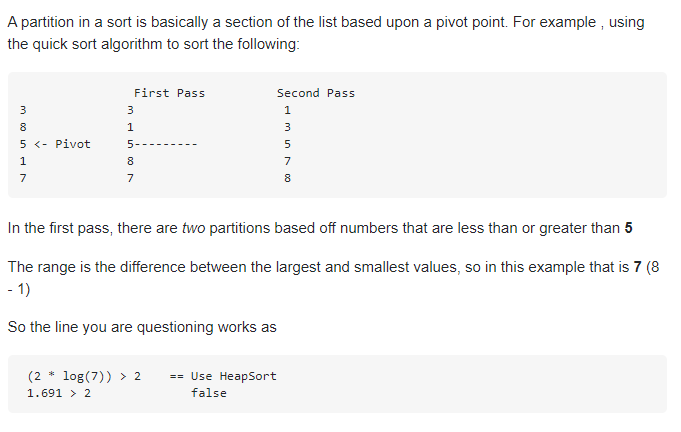
# Hvad gør .NET?

## List.Sort/Array.Sort

[Array.Sort](https://docs.microsoft.com/en-us/dotnet/api/system.array.sort?redirectedfrom=MSDN&view=netcore-3.1#System_Array_Sort_System_Array_) og [List.Sort](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.list-1.sort?view=netcore-3.1) (List.Sort benytter Array.Sort) vælger ud fra input hvilken sorteringsalgoritme der er bedst at bruge: InsertionSort, HeapSort eller QuickSort.

* If the partition size is less than or equal to 16 elements, it uses an insertion sort algorithm
* If the number of partitions exceeds 2 log n, where n is the range of the input array, it uses a [Heapsort](https://en.wikipedia.org/wiki/Heapsort) algorithm.
* Otherwise, it uses a Quicksort algorithm.

Der foretages en såkaldt *unstable sort*, det vil sige at man ikke kan sige noget om hvilken rækkefølge to objekter med samme værdi vil komme i.



Range er dog antallet af elementer [Kilde](https://stackoverflow.com/questions/18181210/what-is-the-number-of-partitions-and-range-of-an-array).

## Enumerable.OrderBy

Denne [metode](https://docs.microsoft.com/en-us/dotnet/api/system.linq.enumerable.orderby?view=netcore-3.1#System_Linq_Enumerable_OrderBy__2_System_Collections_Generic_IEnumerable___0__System_Func___0___1__System_Collections_Generic_IComparer___1__) er tilsyneladende implementeret med QuickSort-algoritmen ([stable](https://en.wikipedia.org/wiki/Sorting_algorithm#Stability) udgave - det vil sige den bevarer den oprindelige orden på objekter med samme værdi). Det fremgår dog ikke umiddelbart af dokumentationen.

# Specielle trætyper

## AVL træer - avanceret binært søgetræ

### Hvad er et AVL træ?

Et AVL-træ er populært sagt et søgetræ der automatisk holdes balanceret[[9]](#footnote-9). Dette håndteres gennem såkaldte rotationer hvor man trækker en underknude op og lader den være roden for et deltræ i stedet for deltræets tidligere rod[[10]](#footnote-10).

### Links

Der er mange muligheder for at finde gode beskrivelser og forklaringer på AVL-træer på nettet. Her er angivet nogle få links, men søg også gerne selv.

#### Skriftlige forklaringer

<http://www.docjava.dk/datastrukturer/balancerede_traeer/avl_traeer/avl_traeer.htm>

#### Videoforklaringer

Video: <https://www.youtube.com/watch?v=vRwi_UcZGjU>

Video: <https://www.youtube.com/watch?v=FNeL18KsWPc>

## Opgaver: Implementér et AVL-træ

Du skal nu implementere dit eget AVL-træ.

### Opgave: Indsættelse

Lav dit eget AVL-træ der sørger for at holde sig selv balanceret ved indsættelse.

Prøv først selv. Hvis du går i stå, så kontakt din lærer eller se denne [video der implementerer AVL-træet](https://youtu.be/8J_CoOhbwVw).

### Opgave: Sletning

Implementér sletning fra AVL-træet på en sådan måde at træet holdes balanceret

### Opgave

Implementér AVL-træet som en generisk klasse

## Opgave: Tidskompleksitet

Gør rede for tidskompleksiteten ved indsættelse og sletning i AVL-træer.

## Andre træstrukturer

Vi vil ikke gå dybere ind i de træstrukturer der er præsenteret nedenfor, de præsenteres bare kort sådan at man ved hvad datastrukturerne går ud på.

### M-ary trees

M-ary træer er træer der kan have flere end to undertræer - op til *m* underknuder til hver knude. Til disse holdes ofte en liste af undertræer på hver knude. Se mere på [Wikipedia](https://en.wikipedia.org/wiki/M-ary_tree).

### Splay tree

Et Splay træ er et træ hvor man hurtigt kan finde en tidligere brugt knude. Se mere på [Wikipedia](https://en.wikipedia.org/wiki/Splay_tree).

### Rød-sorte træer

Rød-sorte træer er ligesom AVL-træer (forholdsvis) balancerede søgetræer. Her anvendes blot en anden metode til at holde træerne balancerede. Se mere på [Wikipedia](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree).

# Hashtabeller

## Hvad er en Hashtabel

En hashtabel er en datastruktur der kan benyttes til at gemme par (key, value), hvor key-værdien bruges til at slå op i datastrukturen, mens value er den værdi der gemmes. En hashtabel minder altså meget om et array, blot med den forskel at indeks i en hashtabel kan være mange forskellige typer (i arrays er indeks altid en integer). En anden forskel er at key-værdierne i en hashtabel ikke behøver at være fortløbende. Hvis man skulle gemme kunder i et array med deres cpr-nummer som indeks, skulle man bruge et enormt stort array, selvom antallet af kunder ikke var nær så stort.

## Hvordan implementeres en hashtabel?

Vi vil gerne have det hurtige opslag fra et array, så det er nærliggende at implementere hashtabellen i et array. Så skal vi have fundet en måde til hurtigt at komme fra key-værdien til et indeks i arrayet.

Se evt. denne video for en gennemgang: <https://www.youtube.com/watch?v=KyUTuwz_b7Q>.

### Implementation af hashfunktion

Som nævnt ovenfor, implementeres hashtabellen som et array. Men da vi skal kunne slå op med en vilkårlig key, skal vi finde en nem måde at omregne key-værdien til et indeks i arrayet.

Den mest almindelige metode er først at omregne key-værdien til et tal og derefter bruge modulus[[11]](#footnote-11)-operatoren på dette tal med arrayets størrelse som divisor.

### Implementation af selve hashtabellen

Ja, umiddelbart lyder det ikke til at der er så meget at gøre her da vi jo allerede har fastslået at vi skal benytte et array, og hashfunktionen som vi har beskrevet ovenfor sørger for at beregne det korrekte indeks.

Men der er noget vi ikke har taget højde for endnu. Hvordan sikrer vi at to keys ikke bliver tildelt samme indeks i arrayet? Det er jo ganske sandsynligt at hashfunktionen vil beregne samme indeks for to forskellige keys. Den situation, som kaldes en collision, bliver vi nødt til at håndtere.

Der findes flere forskellige måder at håndtere collisions på. Her er to af de mest anvendte.

1. Hvis den pågældende plads i arrayet er optaget, så find den næste ledige plads.

Med denne strategi skal vi når vi skal finde objektet igen, *starte med* at kigge i det beregnede indeks og så søge lineært mod højre.

1. Arrayet holder ikke værdier, men en reference til en linked list hvor alle objekter der er registreret under dette indeks kan findes.

### Opgave: Hashtabel

Implementér en klasse HashTable<T> som er generisk i den forstand at nøglen skal være en string, mens værdierne skal være generiske, sådan at f.eks. HashTable<Player> vil være en hashtabel der kan holde spillere (nøglen kunne f.eks. være brugernavnet). Altså skal hashtabellen være generisk på den type objekter der gemmes i systemet (men ikke på nøglen).

Lav et program der tester hashtabellen, f.eks. oprettes et antal spillere med navn og score i en konkurrence. Der tildeles løbende point i forskellige discipliner, så brugeren skal kunne indtaste navn og antal point der skal adderes. Test derefter at den rigtige spillers points opdateres.

## Hvor bruger man hashtabeller?

### Databaser

Hashtabeller bruges i databaser til at sikre hurtig søgning på de felter man ofte søger på, det som i databaseverdenen kaldes for indeks. Der vil altid være indeks på primær- og fremmenøgler.

### Hashtabel i .NET

Datatypen Dictionary er netop en datatype hvor vi kan gemme værdier med en key af selvvalgt type.

public class Dictionary<TKey, TValue> {…}

Dictionary er implementeret som en hashtabel.

## Resizing - rehashing

Hvis man benytter et array til at gemme objekter i sin hashtabel, kan der opstå behov for at øge arrayets størrelse når det bliver fyldt.

Dette gøres forholdsvis nemt ved at oprette et nyt array og kopiere elementerne fra det gamle array over i det nye. Man skal dog være opmærksom på at Hashfunktionen sandsynligvis benytter arrayets størrelse til beregning af indeks, så indekset der svarer til en key i det gamle array duer ikke i det nye array. Man bliver derfor nødt til at indsætte elementerne forfra i det nye array med den ”nye” hashfunktion. Dette kaldes *rehashing*.

## Opgave: Automatisk resizing

Implementér automatisk resizing i din hashtabel sådan at hashtabellen gøres dobbelt så stor når antallet af elementer i tabellen bliver større end arrayets størrelse[[12]](#footnote-12).

## Tidskompleksitet

### Ny værdi/opdateret værdi og læsning

Da hashfunktionen udføres i konstant tid, vil tidskompleksiteten ved læsning og indsættelse/opdatering af værdier i det optimale tilfælde være konstant, da der herefter bare skal indsættes en værdi på den rigtige plads i det tilknyttede array, altså i bedste fald en tidskompleksitet på .

Imidlertid kan man med hashtabeller risikere at to nøgler giver samme hashværdi og dermed samme indeks til arrayet. Som nævnt kaldes dette en kollision, og hvis der opstår en kollision bliver man nødt til lineært at søge efter elementet med udgangspunkt i det indeks hashfunktionen gav. I værste tilfælde kan man risikere at skulle lede hele arrayet igennem, hvilket giver en tidskompleksitet for disse operationer på O(n).

I gennemsnit kan man dog forvente at antallet af kollisioner vil være forholdsvis lavt, sådan at man i langt de fleste tilfælde vil kunne finde objektet direkte under det beregnede indeks eller meget tæt på det, så man kan regne med en gennemsnitlig tidskompleksitet på .

## Online information

Der findes flere gode beskrivelser af hashtabeller på Internettet. Nedenfor findes nogle få links, men du kan også søge selv.

### hash tables

<https://en.wikipedia.org/wiki/Hash_table>

<https://www.tutorialspoint.com/data_structures_algorithms/hash_data_structure.htm>

<https://www.youtube.com/watch?v=shs0KM3wKv8>

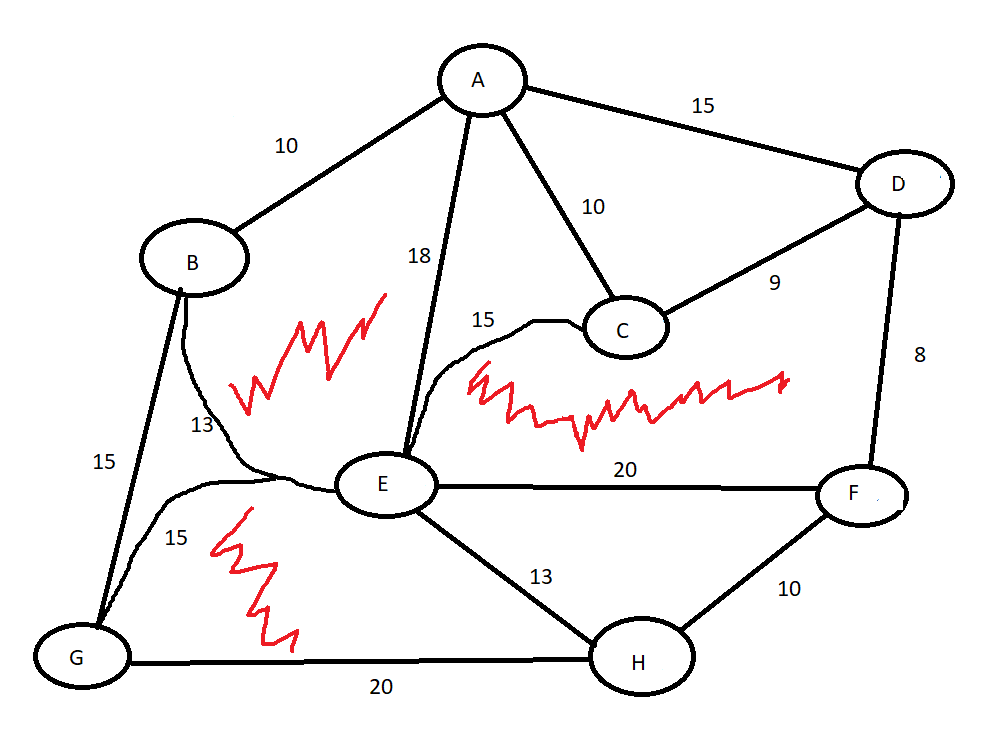
<https://www.youtube.com/watch?v=ea8BRGxGmlA>

### Hash functions

<https://www.tutorialspoint.com/cryptography/cryptography_hash_functions.htm>

# Grafer

En graf består af et antal knuder og kanter. Kanterne er forbindelser mellem knuder. En graf kan altså illustreres ved nedenstående figur:



Figur : Graf - det røde krimskrams forestiller bjerge

Man kan eventuelt tænke på knuderne som byer eller vejkryds og kanterne som veje, men grafer kan bruges til at modellere mange andre ting også.

Der findes forskellige typer grafer:

* Vægtet/uvægtet: I Vægtede grafer har kanterne en værdi - en vægt. Grafen ovenfor er vægtet - hver kant har en værdi.
* Orienteret/uorienteret: Er der retningsangivelse på kanter (f.eks. ensrettede gader)? Ovenstående graf er uorienteret. En uorienteret graf er et specialtilfælde af en orienteret graf, da man kan implementere en uorienteret graf i en orienteret ved at angive to kanter (en hver vej) for hver kant i den uorienterede graf.
* Sammenhængende/usammenhængende: I en sammenhængende graf er der en sti (bestående af et antal kanter) sådan at man fra alle knuder kan komme til alle andre knuder i grafen.

## Gennemløb af grafer

### Dybde først traversering

Hvis man vil gennemløbe en graf, kan man starte i en knude og følge alle kanter fra den knude. I de knuder man møder, vil man igen kunne følge alle kanter osv. Dette vil give et dybde-først-gennemløb, det vil sige at man besøger en naboknude og dens naboer, naboers naboer etc. før man går videre til at besøge de andre af den oprindelige knudes naboer.

Ovenstående simple algoritme skal dog justeres lidt for at fungere i praksis, for hvornår skal programmet standse og hvordan undgår vi at køre i ring?

Strategien er at markere en knude hvis vi allerede har besøgt den. På den måde vil vi ikke begynde at følge kanter vi allerede har fulgt én gang, så når vi følger en kant og ender i en knude vi allerede har besøgt, så er vi ”færdige” med denne del af grafen.

Ved at implementere dybde-først traversering rekursivt, udnytter vi kaldsstakken til at holde styr på hvilken knude vi skal besøge næste gang.

### Bredde først traversering

Hvis man vil traversere en graf med bredde først, skal vi også løbende holde styr på hvilke knuder vi mangler at besøge. Vi kan dog ikke bruge kaldsstakken fordi den rækkefølge vi vil besøge knuderne ikke passer med den rækkefølge kaldsstakkens vil blive besøgt.

I stedet anvender vi en kø hvor vi gemmer referencer til de næste knuder vi skal besøge. For hver knude vi besøger, lægges alle naboknuder bagerst i køen, sådan at vi besøger disse knuder når vi har besøgt alle foranliggende.

Også her har vi selvfølgelig brug for at kunne holde øje med om en knude allerede har været besøgt. Det kan gøres med en property på knude-klassen.

## Uddybende information findes på nettet

Se f.eks. her: [Wikipedia](https://en.wikipedia.org/wiki/Graph_traversal).

## Opgave: Implementér gennemløb af en graf

### Implementér en graf

Implementér en graf der kan indeholde knuder og kanter.

Der findes avancerede måder at gøre dette på, men det er ikke

### Implementér en dybde-først traversering af grafen

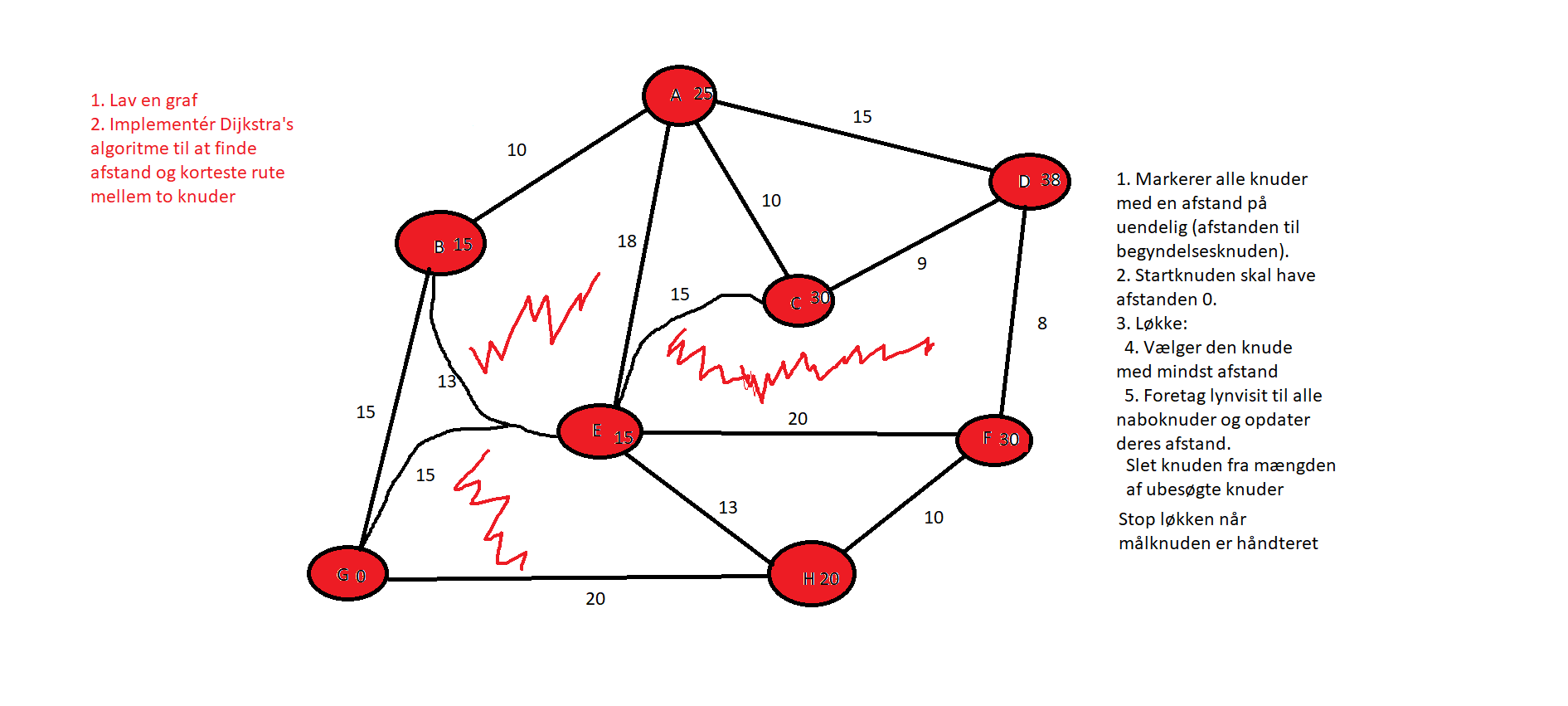
Implementér herefter en algoritme der kan gennemløbe hele grafen. Programmet skal kunne svare på om grafen er sammenhængende eller ej.

### Implementér en bredde-først traversering af grafen

Implementér en algoritme der kan gennemløbe grafen bredde først, det vil sige at alle en knudes naboer skal besøges inden naboernes naboer. Hint: Benyt en FIFO-kø (se afsnit 4.3.2).

## Edsger W. Edsger Dijkstras algoritme

En af de opgaver man ofte vil løse i grafer, er at finde den korteste vej mellem to knuder. Hollandske Dijkstra opfandt i 1956 en algoritme til at løse dette problem på en elegant måde.



En graf - det røde krimskrams forestiller bjerge

Hans algoritme beskriver et optimeret gennemløb af grafen på en sådan måde at den hele tiden holder styr på hvilken af alle besøgte knuder der har den korteste sammenlagte afstand til begyndelsesknuden.

Dijkstras algoritme er en afart af en bredde-først traversering, her benyttes dog ikke en simpel kø til at finde ud af hvilken knude der skal besøges næste gang, men i stedet en prioritetskø hvor den knude man vælger er den med kortest samlet afstand til begyndelsesknuden.

Find grundige beskrivelser her:

[Wikipedia](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm)

[Videogennemgang af algoritmen](https://www.youtube.com/watch?v=eFZCPlZCyIM)

### Opgave: Implementér Dijkstra’s algoritme

Implementér en graf der kan indeholde knuder og kanter. Opbyg en graf hvor du ønsker at finde de korteste distancer.

Implementér Dijkstra’s algoritme til at finde den korteste vej mellem to knuder.

### Opgave: Optimér søgning efter knude

Når man skal finde den tilbageværende knude med mindst afstand til begyndelsesknuden, skal man potentielt lede alle knuder igennem. Dette kan optimeres ved at benytte en datastruktur hvor det er hurtigt at fjerne det mindste element - en bunke (heap).

Benyt en bunke til at holde knuderne. Når en knudes distance ændrer sig (bliver mindre), skal knuden swappes på plads i bunken. Det betyder at knuden skal kende sin egen plads i bunken, hvilket er et problem hvis man swapper ved at bytte værdier. Hvis man swapper ved at flytte på referencer, kan knuden altid referere det samme objekt i bunken, også selvom der skulle blive flyttet rundt på objektet. Det gør det nemmere at håndtere knuden, men sværere at implementere swap i bunken.

Her er en beskrivelse af hvordan man implementerer swap ved at flytte referencer frem for udskiftning af værdier: [Swap ved referenceflytning](https://drive.google.com/file/d/1bqACvvUgH8P84QA2XrCIn7bGQrK74pXs/view?usp=sharing).

### Tidskompleksitet for Dijkstras algoritme

Tidskompleksiteten for Dijkstras algoritme er lidt sværere at forstå og gøre rede for end for de andre algoritmer vi har arbejdet med i dette forløb. I stedet for at tage udgangspunkt i de løkker algoritmen benytter, kigger vi på de objekter vi håndterer. Hvis man lægger alle objekterne og tiden for at finde dem sammen, får vi den samlede tidskompleksitet.

For nemt at kunne beskrive antallet af knuder, kalder vi dette tal for V (eng. Vertex (pl. Vertices) ≈ hjørne). Tilsvarende kalder vi antallet af kanter for E (Edge ≈ kant).

Vi skal håndtere alle knuder som vi tager en ad gangen ved at vælge den med mindst registreret afstand til begyndelsesknuden. Ved at optimere udvælgelsen kan den bringes ned på log(V), sådan at den samlede tid for at håndtere knuder bliver på ).

Godt nok håndteres kanterne i forbindelse med håndteringen af knuderne, men hver kant håndteres kun to gange i alt (én gang fra hver side), så vi kan i stedet se på håndteringen af kanterne. Hvor lang tid tager det så at håndtere en kant? Den opdaterer måske knudens distance-tal - dette tager i sig selv konstant tid, men hvis vi skal bruge tid på at reorganisere de uhåndterede knuder når distancen ændres, skal det regnes med her. Hvis vi benytter en bunke vil en ændret distance betyder at knuden skal swappes på plads opad (værdien er ændret til noget mindre). Da højden af træet er af størrelsen ,vil håndteringen af kanterne altså have tidskompleksitet E∙log(V)).

Da vi skal udføre de to opgaver tidsmæssigt uafhængigt af hinanden (selvom rækkefølgen af deres udførelse er blandet sammen), vil den samlede tidskompleksitet blive beregnet ved at lægge de to tidskompleksiteter sammen. Vi får altså den samlede tidskompleksitet til at være:

Men da antallet af kanter er større end antallet af knuder, så vi den endelige tidskompleksitet til:

, da det bemærkes at konstanter udgår i forbindelse med tidskompleksiteter, så

Ved at benytte en såkaldt Fibonacci heap kan man opnå en lidt bedre gennemsnitlig tidskompleksitet. For interesserede, kan man selv søge information om denne datastruktur: <https://en.wikipedia.org/wiki/Fibonacci_heap>.

### Opgave: Redegør for tidskompleksitet

Redegør *med egne ord* for tidskompleksiteten for Dijkstras algoritme. Hint: Tag udgangspunkt i min forklaring ovenfor eller [Wikipedia](https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm).

# Ekstra: Flere spændende datastrukturer, algoritmer og problemer

## Rød/sort træer

<https://da.wikipedia.org/wiki/R%C3%B8d-sort_tr%C3%A6>

## Splay trees

<https://en.wikipedia.org/wiki/Splay_tree>

## Grafteori

<https://en.wikipedia.org/wiki/Minimum_spanning_tree>

## Travelling salesman problem

<https://en.wikipedia.org/wiki/Travelling_salesman_problem>

<https://www.youtube.com/watch?v=hh-uFQ-MGfw>

<https://www.youtube.com/watch?v=cY4HiiFHO1o>

<https://en.wikipedia.org/wiki/NP-completeness>

## Fibonacci heap

<https://en.wikipedia.org/wiki/Fibonacci_heap>

## Søgning efter koordinater

<https://en.wikipedia.org/wiki/Voronoi_diagram>

1. Det vil dog være i orden f.eks. at oprette et objekt af typen Random hvis man har brug for at få oprettet en større mængde testdata, ligesom man sikkert skal have fat i et bibliotek for at kunne indlæse filer og foldere i en sti. [↑](#footnote-ref-1)
2. Det vil være ok her at benytte en List<int> for at senere undgå problemer med indsættelse versus memory allokering. [↑](#footnote-ref-2)
3. Hvis træet er nogenlunde balanceret, kan vi tale om at vi efter hver sammenligning halverer antallet af elementer vi mangler at undersøge. Er træet meget ubalanceret, kan vi risikere at vi kun fravælger en lille del ved hver sammenligning. [↑](#footnote-ref-3)
4. Et balanceret træ har samme antal underknuder under hver knude i alle niveauer, bortset fra nederste niveau hvor antallet af blade ofte ikke fylder hele niveauet ud. [↑](#footnote-ref-4)
5. Det er selvfølgelig ikke helt korrekt. Ved Indsættelsessortering skal vi højest sammenligne med alle de indtil nu sorterede elementer, ved udvælgelsessortering skal vi sammenligne med alle de ikke-sorterede elementer. [↑](#footnote-ref-5)
6. Arbitrært betyder tilfældigt i den måde at forstå ordet på at vi ikke på forhånd kan vide hvor godt/dårligt valget er. Normalt er der tilknyttet en sandsynlighed når man snakker tilfældighed, sådan en har vi ikke her (eller den er i hvert fald ikke særlig interessant), derfor er ordet arbitrært mere korrekt her. [↑](#footnote-ref-6)
7. Bemærk: At fjerne vilkårlige elementer fra en bunke, ville kræve at vi potentielt søgte hele bunken igennem for at finde elementet. Det ville give en tidskompleksitet på O(n). Rodelementet, derimod, kan fjernes i logaritmisk tid. Derfor implementerer man kun mulighed for at fjerne rod-elementet fra en bunke. [↑](#footnote-ref-7)
8. Dette forudsætter at swap kan foretages i konstant tid, hvilket ikke er svært at overbevise sig om ved at studere metoden i afsnit 0. [↑](#footnote-ref-8)
9. Bemærk: Træet er ikke perfekt balanceret, det skal bare overholde kravet om balancering som gælder for AVL-træer. Med et krav om maksimal forskel i undertræernes højde på 1, er træet dog så balanceret som det er muligt. [↑](#footnote-ref-9)
10. Bemærk: Brugen af ordet *rod* skal her forstås som den øverste knude i et undertræ. Med denne brug af ordet, vil alle knuder være rod i det *deltræ* hvor de selv er øverste knude. Med denne rekursive forståelse af træer vil ethvert træ bestå af en rodknude og to undertræer (som eventuelt er tomme træer). [↑](#footnote-ref-10)
11. Oversat til dansk betyder modulus *rest ved division*. I C# benyttes operatoren % til at angive en modulus-operation. Hvis arrayets størrelse af f.eks. 20, beregner hashfunktionen indekset i arrayet ved beregningen keyValue % 20. Resultatet af denne udregning vil være et tal mellem 0 og 19. Det kan derfor benyttes til at indeksere i arrayet. [↑](#footnote-ref-11)
12. Hvis man benytter linkede lister til værdierne i hashtabellen er det teknisk set ikke nødvendigt at ændre på størrelsen af arrayet, men for at fastholde hurtige opslag vil det være en stor fordel. Hvis man benytter strategien med at finde første ledige plads i tilfælde af kollision, er det *nødvendigt* at resize arrayet når det bliver fyldt helt op. [↑](#footnote-ref-12)