Polynomial: nx  , hvor x er en konstant

Eksponentiel: xn, hvor x er konstant

Overhead: Antallet af dublicates i et array

Dublicates: antallet af et bestemt element i et array, minus 1. Hvis der kun er et element vil der derfor være 0 duplikater. Det vil sige ved i ens elementer i et array, er dublicates i-1

# Algoritme:

Vigtige proporties:

Er algoritmen ”in-place”? – in-place er når algoritmen bruger en konstant mængde plads lige gyldig hvor stor n er. Hvis den opretter nye arrays er den ikke.

Korrekthed: En algoritme er korrekt hvis den for alle legale input afslutter og producerer det ønskede resultat.

En veldefineret computer procedure der omdanner input til output.

Der kan være flere rigtige algoritmer til hver opgave

Kriterier: for hvert input giver den det korrekte output

## Kompleksitet:

Space: hvor meget hukommelse bruger den

Tid: hvor lang tid tager algoritmen

God eller dårlig algoritme

Vil måske afhænge af input: best, worst og average case.

Incorrect algoritme: giver måske ikke output på input, eller giver forkert output

Brugbar i tilfælde hvor der ikke er nogen bedre korrekt algoritme, men man kender fejlprocenten og margin.

## Lineær søgning:

Defineret af størrelse på input

Kompleksitet som funktion af input størrelse n

Space used: array størrelse

Best case: A[1] = q, running time =1, konstant tid

Worst case: A[n] = q, running time = n, lineær tid

Avarage case: q er forventet I midten af sekvensen, running time = n/2, stadig lineær tid

Ikke godt hvis der er rigtig mange data. Big Data æra

## Binær søgning

Virker hvis data er sorterede.

Ideer bag: del og hersk (divide and conqure), en af nøgle design teknikkerne.

Rækkefølgen af data sørger for at elementer man leder efter ikke er ignoreret.

**Sorterings problem:**

Input sekvens er kaldt en instans af sorterings problem.

Der kan være uendeligt mange input

Nogle algoritmer er hurtigere end andre.

**Fremgangsmåde**

1. Analyser specifikationer på forholdet mellem input og output
2. Design algoritmer og datastrukturer
3. Implementer algoritmer i rigtigt programmeringssprog

**Algoritme mål**

* Korrekthed
* Effektivitet

Datastruktur:

Gemmer og organiserer date for at få tilgang til og

Datastruktur: Hvordan organiserer vi data der skal håndteres

Algoritme: Beskriv processen hvor data er håndteret

# Ram model:

Hver instruktion tager en konstant tid betinget af maskinen den kører på.

Instruktioner er operationer som aritmetiske, data bevægelse, kontrol(return fx), sammenligning.

Kompleksitet handler om hvor mange instruktioner ved et givent input.

Husk alle sammenligninger – også i if og for.

tj er afhængig af hvor meget j er i et forloop.

Best case vil altid være 1.

**Average case**: svær at prædefinerer, handler om hvilke input er mest sandsynlige. Bliver kun talt om i Quick sort(i kurset).

**Worst case:** giver en højeste grænse, som garanti for at det ikke bliver ofte. Kan ske ofte. Avarage er ofte lige så slemt som worst.

# Asymptotic notation

Bruges til at simplificere analyse af køretid ved at fjerne ligegyldige detaljer, 3\*n^2 = n^2. Ignorer maskine afhængig tid.

For at lave Theta Θ som ingeniør: ignorer forstående konstanter fx 3x, ignorer 3. Find det stærkeste led. Det er det der siger noget.

Konstant c< poly-logarithm, lgkn<polynomial, na< exponential, bn

Man skal finde punktet n0 hvor f(x) altid vil være større end g(x)

Sammenlign to algoritmer. Hvor hurtigt er g(x) vokset højere end f(x). Hvilken der er bedst ændrer sig i forhold til x.

O notation: asymptotic upper bound. Worst case running time is no greater than O(f(x))

Θ notation: asymptotic tight bound. Der er en konstant der altid vil gøre f(x) højere end g(x), og der er en konstant der altid vil gøre f(x) lavere end g(x).

Ω notation: asymptotic lower bound. F(x) gror asymptotically hurtigere end g(n) hvis f(n) = Ω(g(n))

Asymptotic positive: at den er større end 0 fra et bestemt punkt – n0. Også kaldet nonnegativ. Den kan altså aldrig falde til under nul efter dette punkt, men kan godt være under 0 før n0.

Concrete complexity: Refererer til resultat af ram analyse, med mange detaljer

Abstrakt kompleksitet: Refererer til resultat af symptotic analyse, theta, Big-O…

# Abstrakte data typer(ADT):

En specifikation på et sæt data og et sæt operationer der bliver udført på dataerne. Operations sættene er i fokus. ADT er abstrakt fordi det er uafhængigt af forskellige konkrete implementationer. Det indkapsler datastrukturer og relevante algoritmer og giver access interface.

En matematisk enhed og operationer der kan blive gjort på denne:

Fx en integer, her kan man gange, plusse osv.

Fordele: brugere skal bare vide hvad man kan gøre, men ikke hvordan det er implementeret.

Man kan vælge de bedst mulige implementationer for ADT når man skal implementere det. Så kan man separere correctness og performance analysen af algoritmer.

# Common time compleksitet:

## Perfekt

Konstant tid  Θ(i)

## Feasible alogrithmes – godt

Log time Θ(lg n)

Linear time Θ(n)

Log linear time Θ(n log n)

## Polynomial time – okay til små

Quadratic time Θ(n^2)

Cubic time Θ(n^3)

## Infeasible – duer slet ikke, umuligt for computer

Eksponential Θ(2^n)

# Monotenicity:

En function er “monotonically increasing” hvis m er mindre end eller lig n også betyder at f(m) er mindre end eller lig f(n): m <= n, f(m) <= f(n)

En funktion er ”monotonically decreasing” hvis m er mindre end eller lig n betyder at f(m) er større end eller lig f(n): m <= n, f(m) >= f(n)

En funktion er ”strictly increasing” hvis m er mindre end n betyder at f(m) er mindre end f(n)

En funktion er ”stricktly decreasing” hvis m er mindre end n betyder at f(m) er større end f(n)

# Design af algoritmer

(s. 29)

Forskellige typer design teknikker. Der er incremental fremgangsmåde.

## Divide and conquer

Ofte let at bestemme running time. Deler sig op i forskellige dele der ligner hovedproblemet, men er mindre. De gør dette rekursivt så mange af delene bliver ordnet hver for sig, og samler sig så til sidst og løser problemet.

Tre step i hver rekursions niveau:

**Divide:** Del problemet ind i x antal underproblemer der hver er små instanser af samme problem.

**Conquer:** Løs underproblemerne rekursivt. Hvis underproblemerne er små nok skal man bare løse dem ligefremt.

**Combine:** Kombiner e forskellige underproblmer til løsningen af det originale problem.

## Basal idé:

Del et problem ind i mindre dele, så det er lettere at håndtere.

## Maksimum subarray problem

(side 68)

Hvis man har et array og finder det underarray med den højeste sum, er det ”maximum subarray”. Der kan være flere i et array. Dette er kun interessant når der er negative indgange i arrayet.

# Løsning af rekursioner:

## Analyser divide and conquer algoritmer:

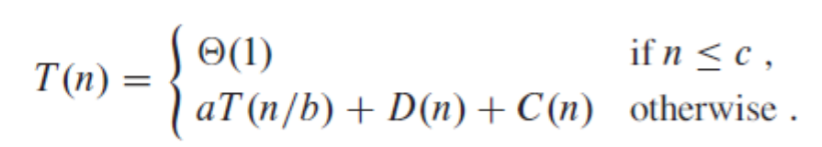
### Recurrence:

En ligning eller ulighed der beskriver en funktion ud fra dens værdi på mindre input.

*Hvis problemstørrelset er lille nok kan det blive løst i ”constant time” Θ(1).*

*Deling af et problem giver* ***a*** *underproblemer. Hvert af disse er 1/****b*** *af størrelsen af det originale*

b referer til hvor mange dele problemet bliver delt ind i. Fx i binary seach bliver det delt i 2, så da er b = 2. Her er der også kun et underproblem, hvilket vil sige at a = 1.



D(n) er kompleksiteten af at dele sig.

C(n) er antallet af kald man skal foretage for at samle resultatet efter man har delt det ud i alle underproblemerne.

c er (sandsynligvis) antallet af elementer der højest kan være for at problemet kan løses i konstant tid. Altså hvor der ikke er nogle delinger.

## The ”Repeated Substitution Method:

Forelæsning 3.

T(n/2)+f hvis n>1

e repræsenterer T(n) hvis n er 1, altså T(1).

1. Observer et mønster og skriv hvordan udtrykket ser ud efter den i’te substitution

* Find et mønster. Se udviklingen igennem flere lag. Første gang den kaldes, anden gang den kaldes. Her er det ***T(n/2i) + i+f***

1. Find ud af hvad i’s værdi skal være for at få base case af rekursion T(1)

Find hvad i skal være for at den indsatte værdi i T, altså hvad der svarer til n i T(n) er 1. Da T(1) = e vil dette vær base case. Her er det lg n = i, da dette giver n/n hvilket er 1. Her får vi ***T(n) = e+f\*lgn***

1. Indsæt værdien af T(1) og udtrykket af i ind i udtryk.

Fjern de led med mindre kompleksitet og konstanter. Her bliver resultatet ***T(n) = Θ(lgn)***

(s. 34)

Når en algoritme har et rekursivt kald til sig selv, kan man ofte beskrive dets running time med ”recurrence equation” eller ”recurrence”. Denne beskriver den overordnede køretid af en algoritme med størrelsen n ud fra køretiden på mindre input.

(side 65)

Når et rekursivt problem bliver så småt at man ikke kan dele det rekursivt igen, siger man at ”the recursion bottoms out” og at man er kommet ned til ”the base case”.

Nogle gange skal man løse underproblemer der ikke er helt de samme som det originale problem.

Her er beskrevet tre metoder til at løse rekursisioner:

* Substitution method: gætter på en grænse og bruger matematisk induktion til at bevise at det er rigtigt
* Recursion tree method: Laver rekursionen om til et træ hvis grene repræsenterer de forskellige niveauer af rekursionen. Der bruges teknikker til ”bounding summations (midt s. 66) til at løse denne rekursion.
* Master method: giver grænser for rekursioner i formen T(n) = aT(n/b)+f(n), hvor a >= 1, b > 1 og f(n) er en given funktion.

Hvis man kun kan bestemme upper bound af en funktion bruges O(n), hvis man kun kan bestemme lower bruges Ω.

## Substitutions metoden:

Gæt løsningen.

Bekræft løsningen med matematisk induktion.

## Rekursionstræ:

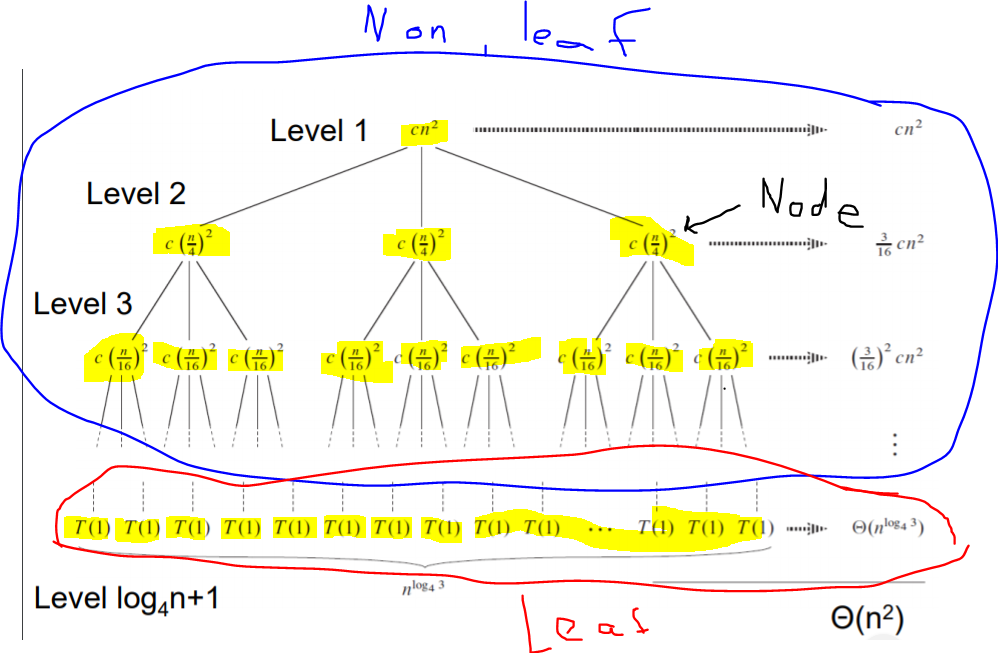
En let måde at visualisere hvad der sker når en rekursion er itereret. Hvert blad (node) repræsenterer et enkelt underproblem. Hvis man tæller prisen for hver enkelt level af træet op, kan man på et sæt af ”per-level cost”. Summer man disse sammen, kan man bestemme den totale pris for alle levels.

Node: Hvert punkt på træet er en ”node”.

Non leaf level:De dele der ikke er slutninger. Prisen for denne kommer an på kosten af at dele og kombinere.

Leaf level:Slutningen af træet. Prisen kommer an på hvor mange ”nodes” der er. Hver node har en konstant pris.

Den fulde pris for træet er både non leaf og leaf levels priser lagt sammen.



## ”Master Method”:

Forelæsning 4, slides.

**Mergesort eksempel:**

a=2m b=2. Hver gang deler vi problemet op i to underproblemer, derfor a=2. Hvert underproblem er halvt så stort, derfor b=2.

Det at dele tager konstant tid, altså:

Det at samle tager lineær tid, da ”Merge” funktionen tager lineær tid, altså:

Vi har derfor at , da dette er det stærkeste led. Dermed får vi, når vi sætter værdierne ind i formlen fra før, køretiden:

Er en skabelon for at løse en ligning i formen:

hvor er konstanter og f(n) er asymptotisk positiv.  
T(n) er køretiden af algoritmen. Om denne ved vi at:

* Et underproblem af størrelsen er løst rekursivt, hver især i tiden .
* f(n) er prisen for at dele problemet og sammensætte resultatet igen: *D(n) (divide) er prisen for at dele, og C(n) (combine) for at kombinere*.

Sådan noget som **Factorial** algoritmen kan ikke lægges på den formel, da dennes runningtime er på formlen , og altså ikke er kompatibel med Master method.

### Theorem:

Første case:

Hvis for en konstant q>0, så er .

Anden case:

Hvis f(n) = , så er .

Tredje case:

Hvis for en konstant , og Regulatity contidion er opfyldet, vil T(n) = Θ(f(n))

Regulatity condition:

for en konstant c<1 og alle n der er store nok.

### Hvordan bruges master method:

**Mergesort eksempel:**

a=2, b=2, f(n)=Θ(n)

Og også, f(n) = Θ(n)

Derfor bruger vi case 2,

1. find a, b og f(n) fra en given rekursion.
2. Find
3. Sammenlign f(n) og

* f(n) vokser polynomisk langsommere, case 1
* De vokser ligeligt, case 2
* f(n) vokser polynomisk hurtigere, case 3

1. Bestem en passende MM case, og anvend

**Bemærk:**

Hvis du kan skal du bruge MM, det er den simpleste måde. Hvis du ikke er sikker på hvilken case der skal bruges, specielt for at undersøge regularity condition i case 3, så brug repeated substitution eller rekursionstræ.

Hvis du ikke kan bruge MM, så brug repeated substitution eller rekursionstræ (eks. factorial)

Du kan altid bruge repeated substitution eller rekursionstræ til en hvilken som helst rekursion.

# Loop invariants:

Invarianter:

Udtryk omkring tilstanden af udførsel (assertions) der er validt på et hvilket som helst tidspunkt de er nået. Ofte i udførsel af algoritme specielt i loops.

## Regler:

Du skal vise tre ting om loop invarianter:

* Initialisering: det skal være sandt før første iteration.
* Maintenance: Hvis det er sandt før en iteration, skal det blive ved med at være sandt før næste.
* Termination: Når loopet ender, skal invarianten give nyttig information for at vise korrektheden af algoritmen.

## k Nearest Neighbor problem (kNN)!!!

# Data strukturer:

## Binary heap:

Er et array der kan blive set som et næsten komplet binært træ. Alle niveauer undtagen det sidste er helt fyldt. Det sidste kan være mere eller mindre fyldt. Der er to typer heap, min heap og max heap. Ved min heap er roden mindre eller lig med alle sine børn, og højre og venstre ”subrees” er igen binary heaps. Det samme ved max heap, bortset fra at roden her er større end sine børn.

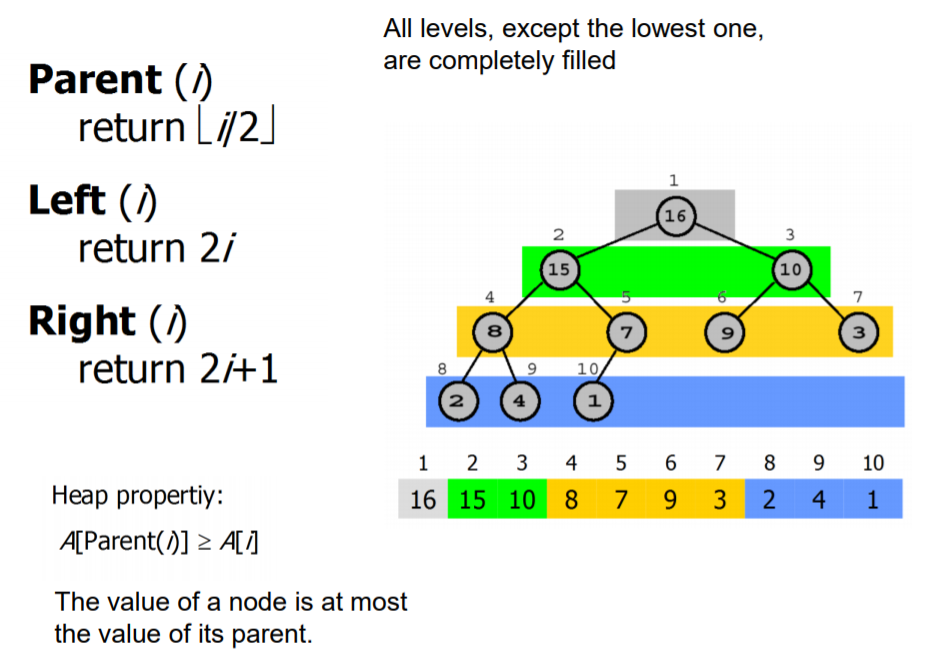
En heap har to attributter:

A.length: antallet af elementer i arrayet.

A.heapsize: Antallet af elementer i heapen der er gemt i arrayet.

0

Alle elementer til og med A.length kan indeholde værdier, men det er kun dem op til A.heap-size der er valide i heapen.



Du finder elementerne i heapen fra arrayet ved brug af overstående formler.

Med metoden heapify laver A en heap ved at rykke A[i] ned i heapen indtil den er ordentligt sorteret igen.

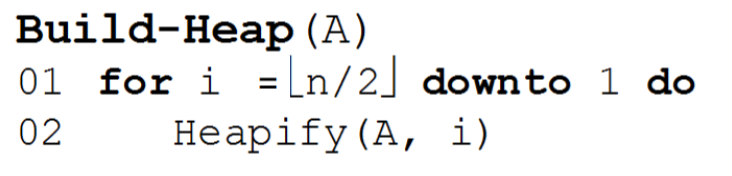
#### Analyse af heapify (O(h):

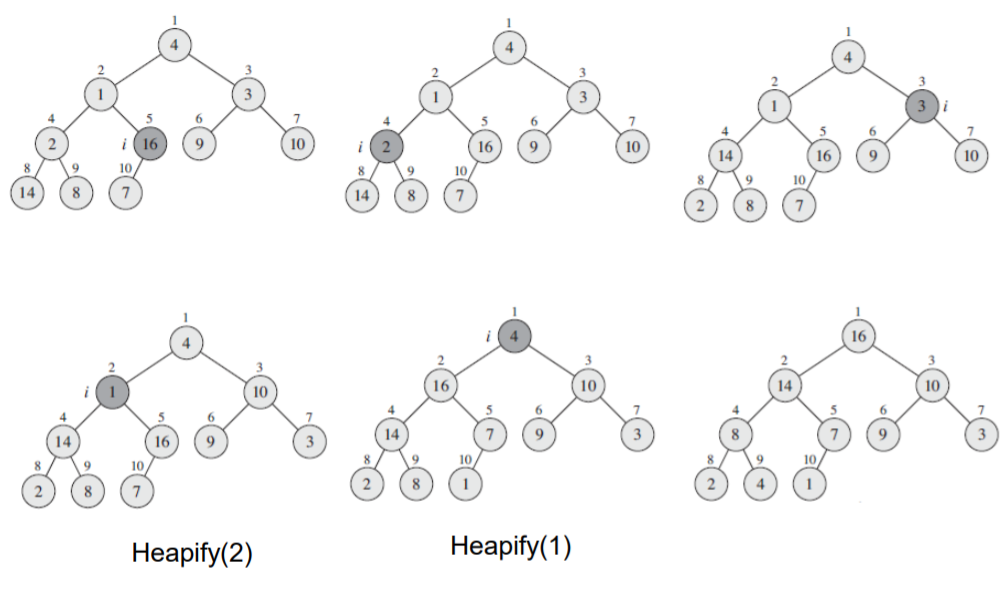
Dividing har konstant tid.

Combine: ingenting

Conquer:

#### Lav array om til heap:





Build-Heap har for hvert kald en kompleksitet på O(n). Hvert kald til heapify har en på O(lgn).

Derfor tager det O(nlgn) at bygge en heap med n elementer. Det er dog ikke tight bound, fordi det ikke er altid heapify behøver at tage O(lgn). Det tager O(h) afhængig af nodens højde.

O(n) er tight upper bound.

### Heapsort:

Sorterer ”in place”, men har en lav running time.

## Stack:

En stack er en beholder der indeholder en samling af objekter. Disse er tilføjet og fjernet så den sidste tilføjet er den første til at blive fjernet. Det er kun det toppen af stacken der er tilgængelig.

Objekter kan altid blive tilført, men kun det øverste element kan blive fjernet. Det øverste element (kan være) er noteret som S.top af arrayet/ stacken S.

### Funktioner:

Push(S,x) sætter element x til toppen af stacken S

Pop(S) sletter det øverste element på stacken.

Stack-Empty(S) returnerer om stacken er tom.

## Queue:

Er som en normal kø.

Enqueue: et element er tilført til køen i slutningen af denne.

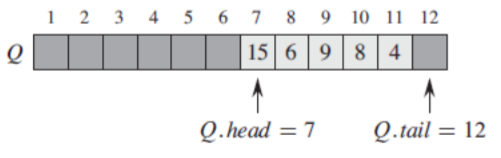
Dequeue: Det første element i køen, det element der kom først, bliver fjernet fra køen.

Går efter konceptet FIFO – first in first out.

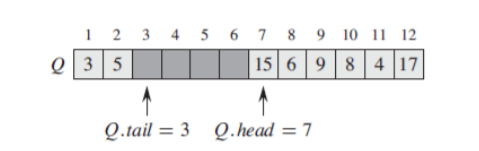
Det er kun det første element der er tilgængeligt som ”head” af køen.

Head af en kø, Q, kan blive betegnet som Q.head. Q.tail er det næste sted et element vil blive indsat i en kø. Andre elementer kan betegnes som Q.head+1 osv, og Q.tail-1 osv.

Til at starte med er Q.head = Q.tail.



Hvis der er overflow fortsætter arrayet fra den anden side.



De fjernede elementer vil stadig være i arrayet, men tilhører ikke køen mere.

### Funktioner:

Q= køen, x = elementet:

Enqueue(Q,x) indsætter element x i køen Q.

Dequeue(Q) sletter det første element, the head, fra køen Q.

Alle operationer tager konstant tid, og størrelsen af n er ligegyldig.

## Linked-list:

Der er forskellige typer linked lists, det kommer an på hvordan de er forbundet. Hvert element er linket med en key og en eller flere pointere.

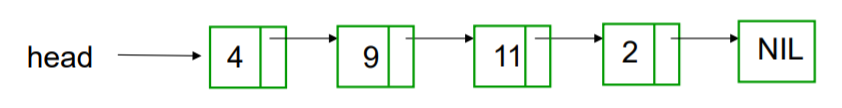
### Single key list:

Element:

En Key.

En pointer ”next”. Denne peger på elementet der kommer efter.

Next fra det sidste element i rækken peger til ”NIL”

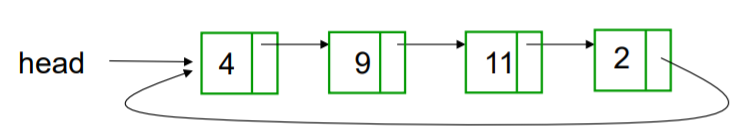


Next pointere er for elementer, x.next.

Head pointer er for hele listen, L.head.

#### Singly linked list:

Det sidste elements ”next” peger på det første element.

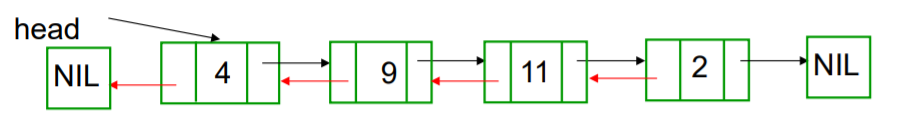


### Dobbelt linked list:

Elementer:

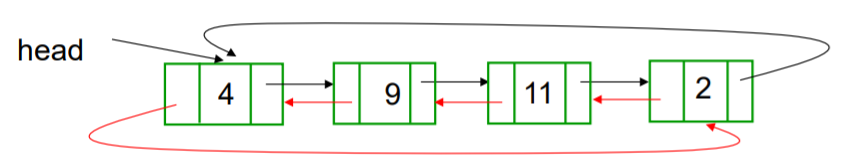
* En nøgle.
* To pointere – en ”next” som peger på den næste og en ”prev” der peger på den før.

Her er også pointeren for hele listen, head pointeren.



#### Alternativ cirkular liste:

Som dobbelt linked list, bortset fra at det sidste element peger til det første og det første til det sidste i stedet for ”NIL”.



#### Operationer i doubly linked list:

Seach(L,k) – finder det første element med key k i liste l.

Insertion(L,x) – indsætter element x i L

Deletion(L,x) – sletter element x fra listen L.

Ved de to sidste indsættes og slettes element x, ikke elementet med key x.

## Priority queues:

(der må gerne være duplicates, isEmpty, add, peekMin, removeMin, rækkefølgen er ligegyldig

En priority queue (PQ) er en datastruktur til at vedligeholde et sæt A af elementer. Disse har hver en associeret værdi kaldet key.

Følgende operationer understøttes (med heap):

Insert(A,x) insætter element x i sæt A (A=∪{x})

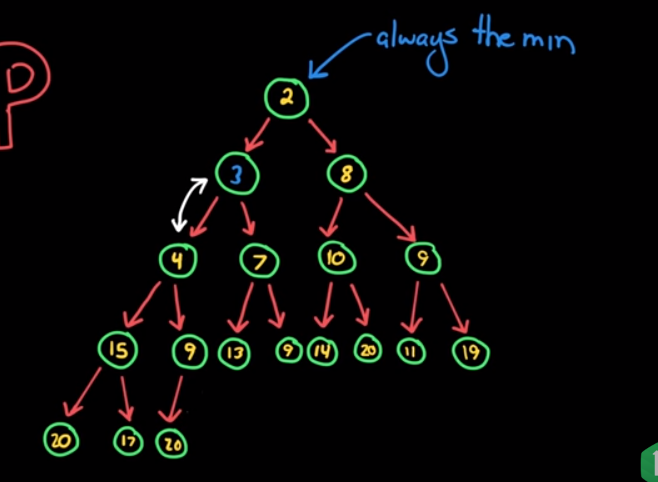
Maximum(A) returnerer elementet i A med højeste key.

Extract-Max(A) returnerer og sletter elementet i A med den højeste key.

### Heap implementering:

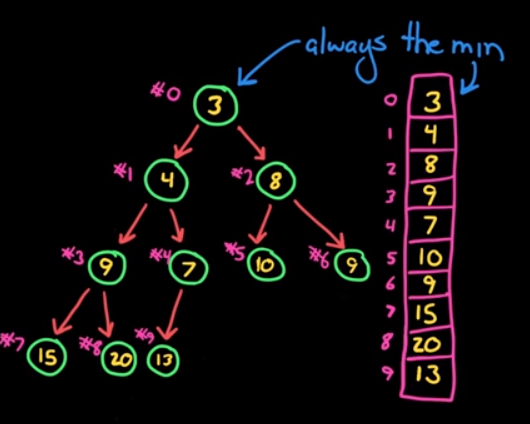
Brug af heap:

Der er minimum og maksimum heap.

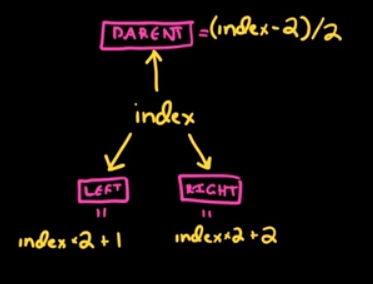


Billedet repræsenterer minimum heap – det laveste skal altid være på toppen. Max er omvendt. Man indsætter altid nye elementer ved bunden og bobler dem så op til de ligger rigtigt. Man ved at det laveste element altid vil være ved bunden. Når toppen fjernes, bliver den byttet ud med det sidste element i strukturen. Så tager man elementet og sammenligner med dem under, den med pilen til venstre, og den med pilen til højre, og bytter rundt indtil at det er på plads.

Man kan bruge et array til at indeholde disse elementer.



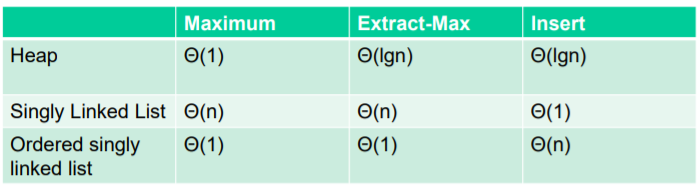
En udregning kan tage dig fra index til venstre child, højre child eller barn.



Ved en min heap er en node altid mindre end sine child nodes, ved en max heap er den altid større.

Maximum(A)

Heapstørrelsen repræsenterer det største element i arrayet. Du ved derfor hvor mange elementer der er. Når du indsætter skal du gøre denne en større og indsætte på den nye heap plads. Hvis du har et sorteret array skal du bytte rundt på elementerne indtil det nye er på plads. Når du sletter et element skal heapen gøres mindre. Du rykker så alle elementerne så der ikke er hul.



## Søg:

### Dictonary:

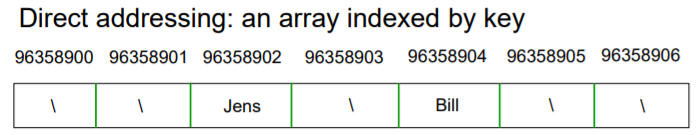
Indeholder elementer der kan blive fundet hurtigt ved brug af keys. Dictionary ADT- et dynamisk sæt med metoder:

* Seach(S,k) – returnerer en ponter x til elementet hvor x.key = k (Access operation)
* Insert(S,x) – tilføjer element x til S (Manipulation operation)
* Delete(S,x) – en manipulations operation der fjerner element x fra S

Elementer har en ”key” del og en ”satellite data” del.

Satellite data: kan være fx navn, alder – alle relevante informationer på et objekt

Key: er objektets ”cpr-nummer”



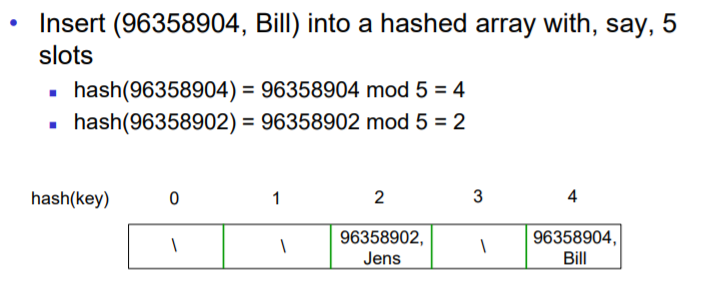
Denne metode har en konstant tid, men en space på . Der er kæmpe mængder ubrugt data ved telefonnumre, da der er mange ubrugte numre.

### Linked list implementering:

Her er der bedre space, men dårligere seach time – Theta(n) begge, da der kun er de relevante data, men man skal lede dem alle igennem.

### Hash tabeller:

Man laver en udregning der giver elementerne en hashkode ud fra deres info. Så kan man søge i hashkoderne (som kan være ens), og så kun se på de hashkoder der kan repræsentere det relevante info.



# Algoritmer i kursus:

## Sortering:

### Merge sort -:

(forelæsning 3 slides)

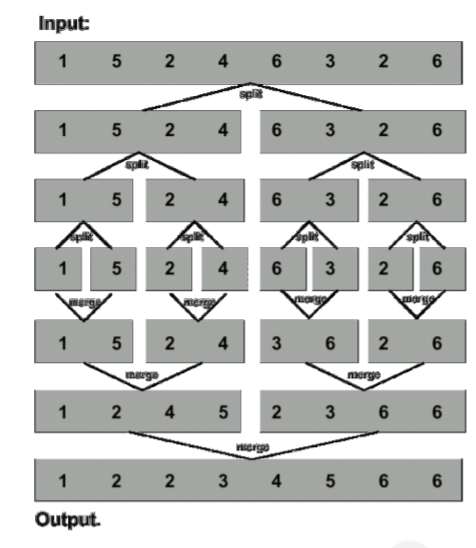
Worst case er

Kan løse sorting problem. Bruger divide-and-conquer teknik.

Ikke in-place sortering. Jo større n er, jo større er den lagerplads der skal bruges. Dette er blandt andet fordi der hver gang oprettes nye array.

Hvis n = 1 er vi færdige.

Man deler talmængden ind i to dele igen og igen, indtil de er i rækkefølge to og to. Bagefter samler man, så de bliver samlet i rækkefølge.



Bruger meget memory Θ(n) i forhold til Θ(1) på insertion sort, da merge i midten har n array.

### Factorial:

#### Rekursiv version:

Input: n, et ikke negativt heltal

Output: fac, et ikke negativt heltal der er lig n!

**Factorial(n)**

**int** fac 🡨 1;

**if** n=1 then fac 🡨 1

**else** fac 🡨 n\* Factorial(n-1)

return fac

Her er T(n):

hvis n = 1

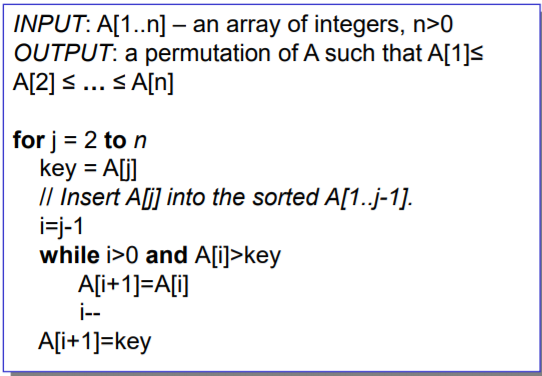
ellers, altså hvis n>1.

### Insertion sort :

In-place sorterings algoritme. Uanset hvor stor n er, vil den bruge samme plads i lageret.

Et konstant antal elementer i input arrayet er lagret udenfor arrayet.

Worst case:



Figur 1 Insertion sort

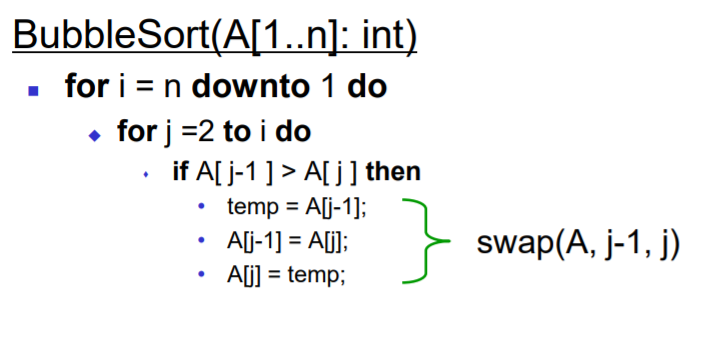
Ikke rekursiv.

### Bubble sort :

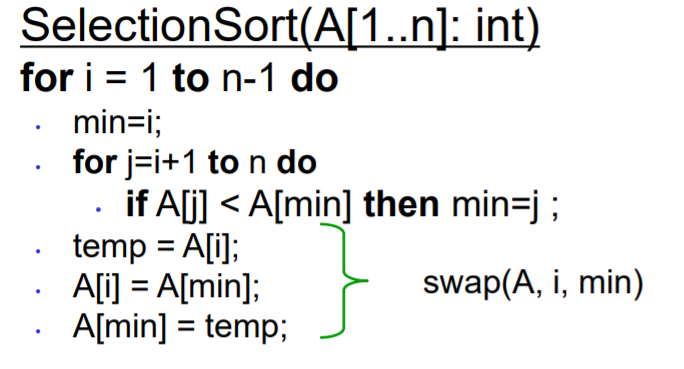
Ikke effektiv.

Er en in-place algoritme

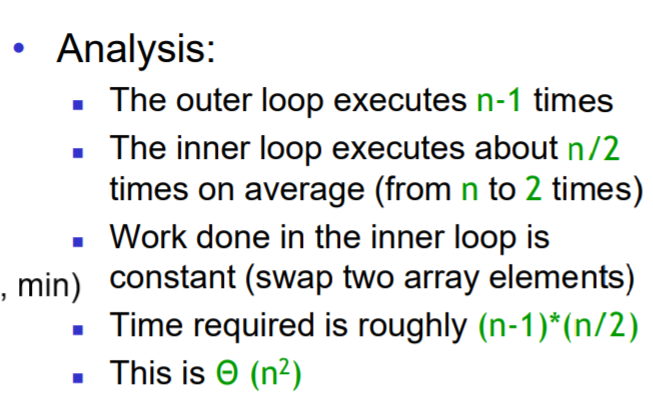
Man bytter igen og igen elementer der ikke er i rækkefølge.

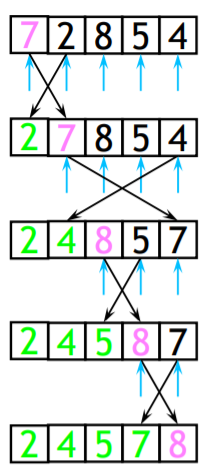


### Selection sort:

Søger igennem alle elementer og finder det mindste. Herefter bytter den element 1 med dette. Så gøres det samme med element to… indtil alle elementer er gennemsøgt og byttet hvis nødvendigt. 

Den er in-place.





### Quick sort :

(forelæsning 5)

Worst case =

Avarage case

Her er der en fordel i forhold til avarage case, da de fleste sort har samme worst og avarage case.

Denne er divide-and-conquer.

**Divide:** Man vælger et element i arrayet. Denne kaldes pivot. Alle større elementer kommer efter, og alle mindre kommer før. Værdier der er lig pivot kan gå til begge sider. Dette kaldes ”partition” operation.

**Conquer:** Man kalder rekursivt quick sort til at sortere de to underarrays.

**Combine:** Trivielt da sorteringen sker in place.

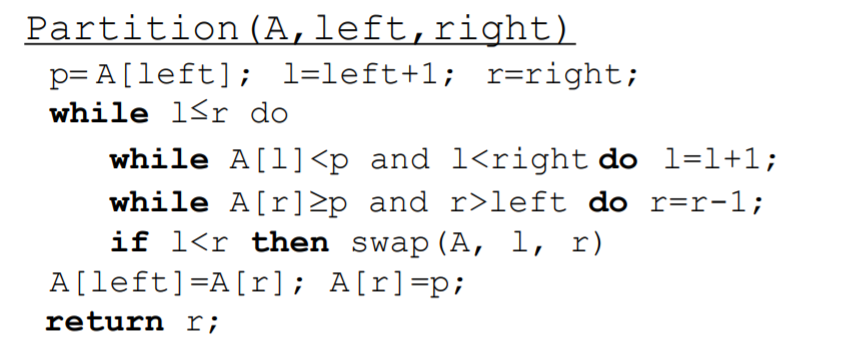
Kan bruges på andet end tal. For at sænke sandsynligheden for at få worst case kan man tage flere forskellige tal og finde det midterste som første pivot.

Fremgangsmåde er som følger.

Vælg en pivot 🡪 del array op i tre dele, pivot og to subarrays. Partition returnerer det sidste index af p i array. 🡪 Quicksort laves rekursivt.



Quicksort består af to algoritmer, også partition:



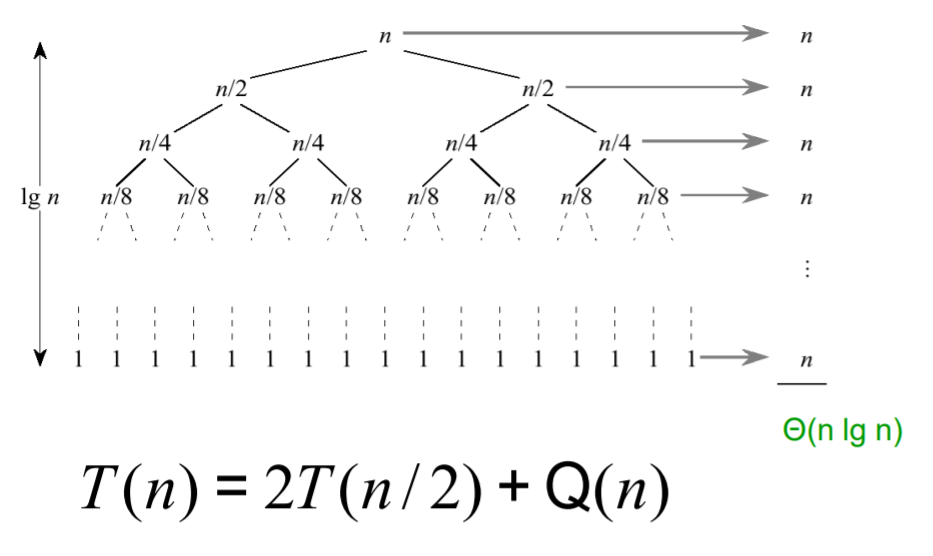
* Vælg en array værdi (som den første) som pivot.
* Fra venstre findes det første element der er større end eller lig pivot.
* Fra højre findes det første element der er mindre end pivot.
* Byt elementer.
* Gentag til alle er byttet.

Runtime af partition er – vi skal bare igennem dem alle.

Når partition er udført, udføres det igen på begge subarrays, det under og det over pivot.



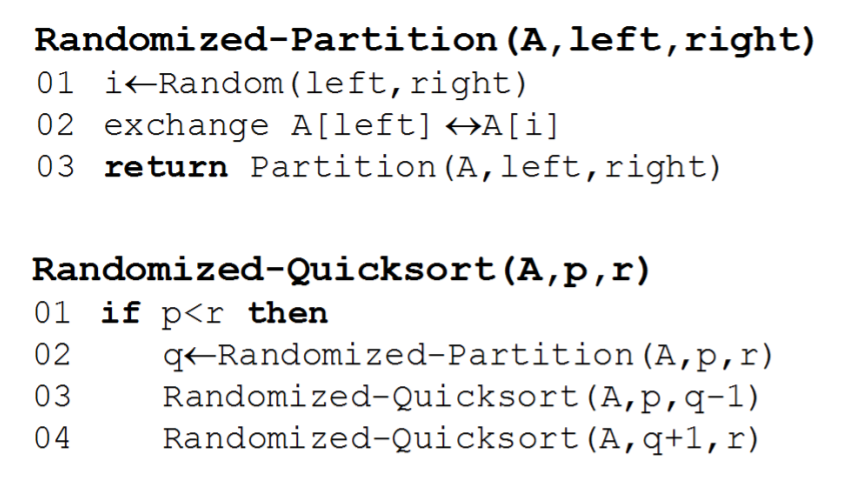
Hvis man er heldig bliver mængden altid delt ligeligt – hvis man er uheldig bliver den slet ikke rigtig delt, men er bare det højeste/ laveste element altid.

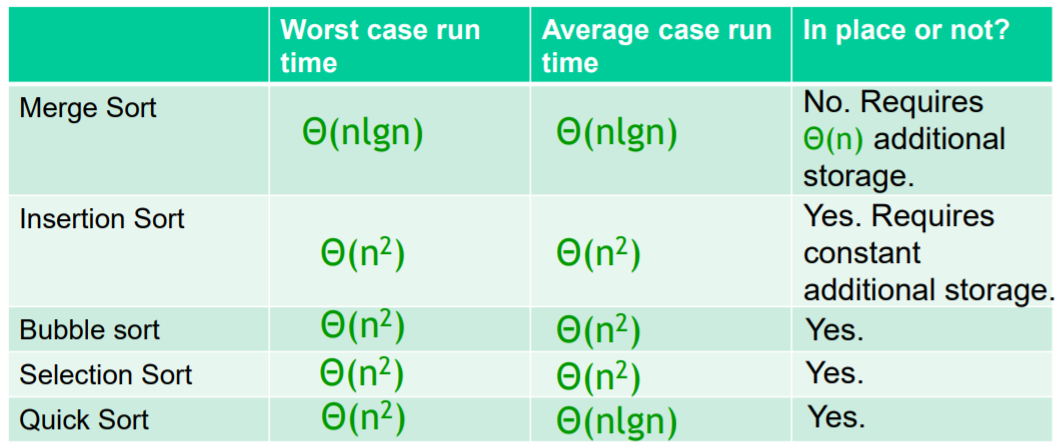


**s**

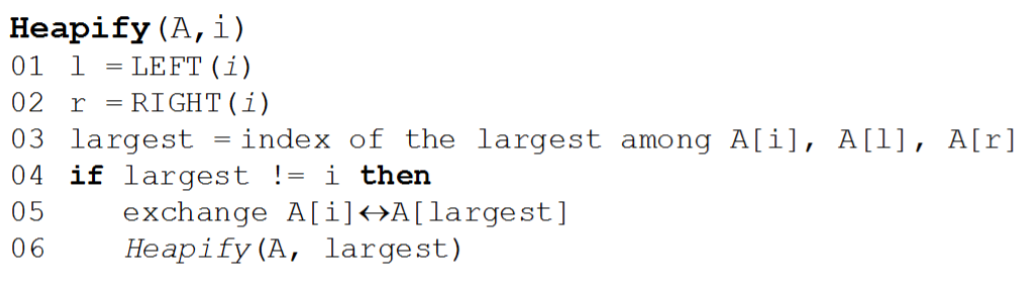
Ved worst case bliver den delt op i:

* En del med nul elementer
* En del med pivot
* En del med længden n-1 der indeholder alt andet end pivot.





### Heapify:



### Heapsort (Onlgn):

