Sorterede vækstfunktioner

- O(1)
- O(log(n))
- $O(n^c)$, $0 < c < 1$
- O(n) — Radixsort, bucketsort (average)
- O(n*log(n)) = O(log(n!)) — heapsort, quicksort (average), mergesort
- $O(n^c)$, $c > 1$ — insertionsort, quicksort (worst)
- $O(c^n)$, $c > 1$
- O(n!)

Algoritmer til O-notation

```
while j ≤ n do
        j ← j + i
        i ← i + 1                       O( sqrt(n) )

while j ≤ n do
        j ← j + 1                       O( n )

while j ≤ n do
        j ← j * 2                       O( log(n) )

for i ← 1 to n do
        y ← x
        for j ← 1 to y do
                x ← x + 1               O( 2^n )

while j ≤ n do
        j ← j * j                       O( log log n )

for i ← 1 to n do
        j ← j * j
        k ← 1
        while k ≤ j do
                k ← k + 1               O( ((2)^2)^n ) [modsat af log log n]
```

Vigtige sidetal

- Insertionsort — Kap 2 s.16
- Heapsort — Kap 6 s. 151
- Quicksort — Kap 7 s. 170
- Radixsort — Kap 8.3 s. 197
- Bucketsort — Kap 8.4 s. 200
- Hash tables — Kap 11 s. 253
- Binary search trees — Kap 12 s. 286
- Red-Black trees — Kap 13 s. 308
- Amortized Analysis — Kap 17 s. 451

Definitioner

Dynamic set	A set which can grow, shrink and change over time. Operations supported are SEARCH, INSERT, DELETE, MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR
Dictionary	A dynamic set which supports the operations INSERT, SEARCH, DELETE
Priority queue	Som en stak, hvor elementet med størst prioritet bliver udtaget først
Disjoint sets	En mængde af sæt, der hver har en repræsentant. Understøtter operationerne MAKE-SET, UNION, FIND-SET
Path compression	Få knuder i et træ til at pege direkte på roden. Figur s. 570

Red-Black trees - Kap 13 s. 308

1. Every node is either **red** or **black**
2. The root node is **black**
3. Every leaf (NIL) is **black**
4. If a node is **red**, then both its children are **black**
5. For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes

Rotations **s. 313 + 314**

RB-INSERT s. 315
Indsæt elementet med TREE-INSERT og farv knuden **rød**. Derefter Kaldes RB-INSERT-FIXUP der gør træet validt igen.

RB-INSERT-FIXUP s. 316
Regel 2 og 4 kan blive brudt når der indsættes. Det er disse overtrædelser RB-INSERT-FIXUP retter op på.

1. z.p.p.right er **rød** (onkel er **rød**). Onkel og z.p farves **sorte** og z.p.p farves **rød**. Dermed er problemet flyttet 2 skridt op i træet.
2. z = z.p.right. LEFT-ROTATE på z.p så z = z.p.left
3. Kommer altid efter case 2. z = z.p.left. Farver z.p **sort** og z.p.p **rød**. RIGHT-ROTATE på z.p.p.

Figur s. 317, 320 & 321

RB-DELETE s. 324
Fungerer som TREE-DELETE. Til sidst kaldes RB-DELETE-FIXUP der genopretter orden i træet.

RB-DELETE-FIXUP s. 326
Se forklaringer af cases s. 327 og figur s. 329

1. x's sibling w is **red**
2. x's sibling w is **black**, and both of w's children are **black**
3. x's sibling w is **black**, w's left child is **red**, and w's right child is **black**
4. x's sibling w is **black**, and w's right child is **red**

Binary search trees - Kap 12 s. 286

TREE-SUCCESSOR s. 292

Hvis x har et højre barn kaldes TREE-MINIMUM på dette barn. Hvis ikke, går funktionen opad i træet indtil elementet er et venstrebarn af sin far. Faren vil her være successor.

TREE-INSERT s. 294
Elementet der skal indsættes tjekkes først med roden og går derefter til højre eller venstre. Derefter tjekkes med den næste knude og går til højre eller venstre. Når den har nået et blad indsættes den og pointere opdateres. Figur s. 295

TREE-DELETE s. 298
Hvis z ikke har nogen børn sættes z til NIL
Hvis z har et barn erstattes z med barnet
Hvis z har to børn findes z's successor y. Denne erstatter z hvorefter z's venstre undertræ bliver y's nye venstre træ. Ombytninger af knuder skal til hvis y ikke er z's right child. Forklaring i punktform s. 296. Figur s. 297

TREE-TRANSPLANT s. 296
Tager et subtræ og indsætter i et andet træ. Figur s. 297

Hash tables - Kap 11 s. 253

Hash functions s. 262
Hashing by division s. 263
Hashfunktionen $h(k) = k \bmod m$
- **k** er key (elementet der skal indsættes i tabellen)
- **m** er antallet af slots i hashtabellen. Optimalt skal denne være et primtal der ikke er tæt på en potens af 2.

Hashing by multiplication s. 263
Hashfunktionen $h(k) = \lfloor m(kA \bmod 1) \rfloor$
- **k** er key (elementet der skal indsættes i tabellen)
- **m** er antallet af slots i hashtabellen
- **A** er et tal mellem 0 og 1. (kan vælges frit)

Universal hashing s. 265

Open adressing s. 269
Elementerne gemmes direkte i hashtabellen. Hvis et slot er optaget prøves det næste, dette gentages til et tomt slot findes. Denne metode kan kun bruges hvis antallet af celler er større end antallet af elementer der skal indsættes. Hvis dette ikke er tilfældet bruges chaining (s.257).
HASH-INSERT s. 270
HASH-SEARCH s. 271

Linear probing s. 272
Hashfunktionen $h(k,i) = (h'(k) + i) \bmod m$, for alle $i \in [1,m-1]$
Kigger først på cellen h'(k). Hvis den er fuld, tages den næste (h'(k) + 1). Hvis den er fuld tages den næste (h'(k) + 2) indtil en tom celle findes.
- **k** er key (elementet der skal indsættes i tabellen)
- **m** er antallet af slots i hashtabellen
- **h'(k)** er en hjælpe-hashfunktion
- **i** er en loopcounter

Quadratic probing s. 272

Hashfunktionen $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$, for alle $i \in [1,m-1]$
Tager først i = 0. Hvis den er fuld tages i = 1 osv. indtil en tom celle findes.

- **k** er key (elementet der skal indsættes i tabellen)
- **m** er antallet af slots i hashtabellen
- **h'(k)** er en (hjælpe)hashfunktion
- **i** er en loopcounter
- $\mathbf{c_1}$ & $\mathbf{c_2}$ er (hjælpe)konstanter

Double hashing s. 272
Hashfunktionen $h(k,i) = (h_1(k) + i*h_2(k)) \bmod m$, for alle $i \in [1,m-1]$
Denne er bedre end de to ovenstående da den afhænger af k på to måder i stedet for en.

- **k** er key (elementet der skal indsættes i tabellen)
- **m** er antallet af slots i hashtabellen
- $\mathbf{h_1(k)}$ & $\mathbf{h_2(k)}$ er (hjælpe)hashfunktioner
- **i** er en loopcounter

Insertion sort - $O(n^2)$ - Kap 2 s. 16

In-place, stable

Starter med 2. element og bobler det tilbage. Derefter tager det 3. element og bobler tilbage. Derefter 4. så 5. osv. Figur s. 18

Heapsort - O(n log n) - Kap 6 s. 151

In-place, not stable

HEAPSORT s. 160 O(n lg n)
Kalder BUILD-MAX-HEAP. Bytter det største element (roden) ud med det sidste. Dermed er første element på plads. Heap-size tælles ned og MAX-HEAPIFY kaldes. Dette gøres indtil heap-size = 1. Figur s. 161

MAX-HEAPIFY s. 154 O(lg n)
Det antages at træerne under LEFT(i) og RIGHT(i) er maxheaps men elementet i kan være mindre end børnene. Elementet i bobles nedad (bytter plads med største barn) rekursivt indtil elementet er på plads. Figur s. 155

BUILD-MAX-HEAP s. 157 O(n)
Kalder MAX-HEAPIFY på den første halvdel (alle elementer der har børn) af elementerne i heapen. Elementerne kaldes nedefra. Figur s. 158

MAX-HEAP-INSERT s. 164 O(lg n)
Indsæt et element i en maxheap. Indsæt -∞ i enden af heapen. Kald MAX-INCREASE-KEY på elemenet. Denne funktion bobler elementet op på plads.

HEAP-EXTRACT-MAX s. 163 O(lg n)
Tager værdien af det roden. Bytter roden med det sidste element og kalder MAX-HEAPIFY på det nye rodelement. Ligesom HEAPSORT.

HEAP-INCREASE-KEY s. 164 O(lg n)
Hæv værdien af et element og genopret maxheap invarianterne. Først ændres værdien af elementet. Derefter bobles det op på plads. Figur s. 165

Quicksort - O(n log n) - Kap 7 s. 170

In-place (possible), not stable

QUICKSORT s. 171 O(n lg n) average case - $O(n^2)$ worst case
Kald funktionen PARTITION og kald derefter QUICKSORT rekursivt på de to subarrays som PARTITION giver. Figur s. 172

PARTITION s. 171
Pivotelementet sættes til at være det sidste element i (del)arrayet. Elementerne der er mindre end pivoten sættes først i arrayet og elementer større end pivoten sættes sidst. Til sidst indsætte pivotelementerne mellem disse mængder og indexet for pivoten returneres. Figur s. 172

RANDOMIZED-QUICKSORT s. 179 O(n lg n) average case - $O(n^2)$ worst case
Identisk med QUICKSORT.

RANDOMIZED-PARTITION s. 179
Først vælges et random element som byttes med det sidste element. Derpå kaldes PARTITION.

Linæer sortering - Kap 8.3 s. 197

Counting sort s. 194 (Figur s. 195)
Laver et array der indeholder antallet af hvert tal fra 0 til k. Derefter puttes disse ind i et nyt array. Se s. 194 og 195 for forklaring.
Stable

Radix sort s. 197 (Figur s. 198)
Sorterer en række d-cifrede tal. Ved først at sortere tallene efter første ciffer, derefter andet ciffer osv. Sorteringen skal være stabil, og der bruges ofte counting sort til at sortere, da denne er stabil. Figur s. 198
In-place, Stable

Bucket sort s. 200 (Figur s. 201)
Det antages at elementerne er uniformt distribueret i intervallet [0,1). Der oprettes n lige store buckets som tallene puttes i. Derefter sorteres hver bucket med insertion sort. Da det antages at elementerne er uniformt distribueret vil disse buckets blive ca. lige store og den forventede udførselstid er O(n). Figur s. 201
Stable

Disjoint sets - Kap 21 s. 561

MAKE-SET	Laver et nyt sæt hvor det eneste medlem er x
UNION	Kombinerer to sæt
FIND-SET	Returnerer en pointer til repræsentanten af sættet som x befinder sig i
Path compression	Få knuder i et træ til at pege direkte på roden. Figur s. 570

Transitions systemer - Kap 21 s. 561

Invariant s. 10-12
Hvis et udtryk holder for startkonfigurationen og det holder for alle transitioner, er udtrykket en invariant for systemet. Se s. 37-38

Termineringsfunktion s. 12-14
En funktion der dekrementeres hver gang en transition bliver udført. Funktionsværdien må aldrig blive mindre end 0. Se s. 39-40

1. Heltallig
2. Aftagende for alle transitioner
3. Må ikke blive negativ