Algoritmer og datastrukturer

Pensumsammendrag i INF2220

Mathias Lohne

Noen kommentarer

Dette notatet begynte som min egen eksamenstrening, og har blitt litt oppdatert, korrigert og fylt ut etter det. Vær oppmerksom på at det sikkert inneholder feil, mangler og alt mulig sånt. Finner du noen teite blemmer er det kult om du sender meg en mail: mathialo@student.matnat.uio.no

Innhold og eksempler er basert på stoff fra læreboka, forelesninger, tidligere eksamener, mine egne notater og obliger, Wikipedia og et tilsvarende kompendium laget av Veronika Heimsbakk[1].

Det er bare å bruke, modifisere, og stjele innholdet i notatet. Hvis du gjør det er det kult om du krediterer. Alle LaTeX-filer ligger tilgjengelig på GitHub[2].

Denne utgaven er kompilert 15. september 2017.

[1] http://folk.uio.no/veronahe/

[2] https://github.com/mathialo/INF2220

Innhold

Del I

Algoritmeteori og kompleksitetsanalyse

1 Kompleksitet og tidsanalyse

1.1 Terminologi og begreper

Alfabeter og språk

Når vi bruker begrepene alfabet og språk snakker vi som regel ikke om språk som engelsk, norsk eller Java (selv om disse også er språk i formell forstand), men om en samling strenger av tegn (tenk: ord). Hvilke tegn vi kan bruke avhenger av hvilket alfabet vi har.

Et **alfabet** er en ikke-tom mengde av tegn (også kalt symboler og bokstaver). Vi betegner ofte et alfabet med Σ. Eksempler på alfabeter kan være det binære alfabetet $\{0, 1\}$ eller det norske alfabetet {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z, æ, ø, å}.

Gitt et alfabet Σ bruker vi Σ^* for å betegne mengden av alle mulige kombinasjoner (strenger) av tegn fra Σ av en endelig lengde. En mengde av strenger, med eller uten noen bestemte regler, kalles et **språk**. Vi konstruerer språk enten ved å liste opp alle ordene i språket, eller ved å gi noen regler som ordene i språket må følge. Under følger noen eksempler på språk, og hvordan de defineres:

- $L = \{$*"a"*, *"b"*, *"ab"*, *"ba"*$\}$ (eksempel på et endelig språk)
- $L = \Sigma^*$ (alle ordene over Σ) (eksempel på et uendelig språk)
- $L = \{i : M$ stopper på input $i\}$ (Den inputen som gjør at en turingmaskin stopper. En variant av haltingproblemet, se 3.3)

Eksempel. Gitt alfabetet $\{0, 1, \text{‘,’}\}$, hva er det formelle språket som tilsvarer sorteringsproblemet over dette alfabetet?

Svar:

$$L = \{(0), (1), ..., (0, 1), (0, 10), (1, 10), ...\}$$

♣

Turingmaskinen

Turingmaskinen er en teoretisk maskin. Den eksisterer ikke i virkeligheten, men er noe vi bruker for å bevise teoremer. Selvfølgelig finnes det ingen formell definisjon på *hva* en Turingmaskin egentlig er, så alle læreverk definerer den litt forskjellig. Det høres kanskje helt Texas ut, men alle definisjonene er tilnærmet ekvivalente.

Definisjon 1.1.1. Litt forenklet kan vi si at en Turingmaskin $M = (\Sigma, \Gamma, Q, \delta)$ består av fire komponenter:

- Et inputalfabet Σ. Alle mulige tegn Turingmaskinen kan forstå.
- Et teipalfabet Γ. Alle mulige tegn som kan finnes på teipen. Σ er alltid inneholdt i Γ. Γ inneholder også et blankt symbol (mellomrom), og kan inneholde andre symboler.
- En liste Q over mulige statuser for Turingmaskinen.
- En liste δ over 'responser' på input. For eksempel: *"Hvis jeg er i status 7 og leser en 'a' skal jeg gå til status 21"*.

Hvis vi skal prøve å se for oss en Turingmaskin intuitivt kan vi se for oss et uendelig lang papirremse (formelt kalt teip). På denne papirremsa står det symboler (fra Σ). Dette er inputen til Turingmaskinen. Maskinen leser ett og ett symbol, og reagerer på symbolet (etter hva δ sier den skal gjøre). Det kan deretter la symbolet stå, viske det vekk eller erstatte det med et nytt symbol (fra Γ).

1.2 Kompleksitetsklasser

Noen problemer kan løses veldig enkelt, for eksempel som å søke i et binært søketre, andre problemer er mye mer kompliserte, og noen er uløselige. I dette kurset skiller vi i hovedsak mellom P, NP og uløselige problemer, selv om kompleksitetsklasser er mye finere oppdelt enn det.

En av grunnene til at vi deler opp problemer i kompleksitetsklasser er at vi på forhånd kan si noe om løsbarheten til problemene, før vi begynner å løse dem. En annen grunn er at problemer i samme klasse kan ha løsninger som ligner, selv om problemene er ganske ulike. Se kapitlet om paradigmer (**??**). For eksempel er både søk i binærtre og quicksort eksempler på splitt og hersk-algoritmer. De ligner i utforming, men gjør ganske forskjellige ting. Begge de to algoritmene ligger i P.

P (polynomial time) P er mengden av alle problemene som kan løses i polynomisk tid, altså de problemene der det finnes en algoritme som løser problemet med kjøretid på formen $O(n^k)$ for en $k \in \mathbb{N}$

NP (nondeterministic polynomial time) NP kan defineres på flere måter. En enkel måte å tenke om NP på er at NP er alle problemene som kan være vanskelige å løse, men hvor en løsning kan sjekkes i polynomisk tid. For eksempel kan sudoku være veldig vanskelig å løse, men å skjekke at en løsning er rett er ganske enkelt, det er bare å skjekke at alle rader, kolonner og bokser er riktig utfylt (ikke inneholder en verdi mer enn 1 gang).

Siden alle problemer i P kan skjekkes i polynomisk tid (man kan bare løse problemet på nytt å se om man får samme svar) er det klart at $P \subseteq NP$. Det er faktisk ikke helt klart hvorvidt $NP \subseteq P$ også, dvs om det faktisk er en reell forskjell på P og NP. Dette er faktisk et av millenniumsproblemene, og et av de store uløste problemene i matematikk. Det er verdt å nevne at de fleste tror at $P \neq NP$, det mangler bare et formelt bevis.

Alle problemene som er like vanskelig eller vanskeligere enn alle problemene i NP kalles NP-hard. De problemene som er i både NP-hard og NP er dermed de vanskeligste problemene i NP. Denne mengden kalles NP-komplett.

EXPTIME (exponetial time) EXPTIME er mengden av alle problemer som løses og sjekkes i eksponentiell tid. Hvis jeg for eksempel ber deg fortelle meg hva det beste trekket jeg burde gjøre i et parti sjakk er, er det vanskelig for deg å regne ut, men også vanskelig for meg å sjekke om svaret ditt stemmer. I mange tilfeller er brute force den eneste muligheten vi har.

På samme måte som for NP har vi EXPTIME-hard og EXPTIME-komplett. Det å avgjøre hvilket trekk i et parti sjakk som er det beste er et EXPTIME-komplett .

R R er mengden av problemer som er løselig i endelig tid. De kan ta flere ganger av alderen til universet å komme til en løsning, men det er mulig.

Uløselige problemer En del problemer har ikke en mulig løsning. Et eksempel på et slikt problem er halting-problemet. Dette problemet går ut på om en turingmaskin kan vite om den noen gang vil gi et resultat (det vil si stoppe, engelsk: halt), eller om den vil gå i evig loop. Et bevis for hvorfor haltingproblemet er uløselig finnes i 3.3.

Vi skal se på noen eksempler på problemer:

Eksempel. Traveling salesperson (TSP)

Et av de mest kjente og studerte optimeringsproblemene kalles *Traveling salesperson*. Problemet går slik: En handelsmann har en liste med byer han må innom. Han vil reise innom hver by én gang, og vil bruke minst mulig penger på turen. Han vet prisen det vil koste å reise mellom hver by. Hvilken rekkefølge burde handelsmannen besøke byene i for å betale minst mulig i reisepenger?

Dette problemet er av eksponentiell karakter. Vi kan ikke si noe om hvilken vei som blir billigst uten å sjekke alle muligheter. Problemet er NP-komplett.

♣

Eksempel. Subset sum

Problemet er slik: Gitt en mengde M av tall, skal vi avjøre om det finnes en delmengde $M' \subset M$ slik at

$$\sum_{m \in M'} m = 0$$

Mengden $A = \{-2, -3, 1, 4, 6\}$ er en slik mengde, siden $1 + 4 + (-2) + (-3) = 0$. Mengden $B = \{-6, -3, 1, 4, 7\}$ er ikke en slik mengde, siden det ikke er mulig å plukke ut noen elementer slik at summen av elementene blir 0.

Å løse dette problemet er ganske vanskelig siden det er NP-komplett. Vi må (slik som i TSP) prøve oss fram med forskjellige kombinasjoner. Å skjekke om en antatt delmengde har sum $= 0$ er enkelt: det er bare å summere og skjekke, og kan gjøres i $O(n)$ tid.

♣

1.3 O-notasjon

Når vi skal analysere kjøretid er vi sjeldent opptatt av et nøyaktig svar, men mer opptatt av hva slags størrelsesorden kjøretiden befinner seg i. Dette er litt av motivasjonen for O-notasjon. Formelt kan vi definere det slik:

Definisjon 1.3.1. La f og g være to funksjoner $f, g \colon \mathbb{N} \to \mathbb{R}$. Vi sier da at $f(n) = O(g(n))$ hvis det eksisterer positive heltall c og N slik at for hvert heltall $n \geq N$ er $f(n) \leq c\, g(n)$

$O(g(n))$ blir dermed en øvre skranke for kjøretid.

Når vi i denne sammenhengen bruker O-notasjon vil vi bruke det som et mål på hvordan kjøretiden øker med inputen. Vi ser på et eksempel:

Eksempel. Vi er gitt en array `int a[]`, med n elementer, og skal summere alle elementene i `a[]`. Det kan gjøres slik:

```
1  int sum = 0;
2
3  for (int i=0; i<n; i++) {
4      sum += a[i];
5  }
```

Vi lurer nå på, hva er kjøretiden til denne algoritmen? Vi ser at vi gjør to forskjellige operasjoner her: oppretter en variabel, og plusser sammen to tall. Den siste operasjonen gjør vi n ganger på grunn av løkka.

Vi har dermed at kjøretiden, $T(n)$, blir:

$$T(n) = t_{\text{opprett variabel}} + n \cdot t_{\text{pluss}}$$

der $t_{\text{opprett variabel}}$ angir tiden det tar å opprette en variabel i minne, og t_{pluss} angir tiden det tar å plusse sammen to tall. Problemet er at disse konstantene varierer ut fra hvilken datamaskin vi bruker, hvilket språk vi implementerer i, etc. Det eneste vi vet noe om er at tiden øker lineært med størrelsen. Derfor er

$$T(n) = O(n)$$

♣

Som nevnt tidligere er vi mer opptatt av størrelsesorden enn den konkrete kjøretiden. Vi bryr oss derfor ikke om konstanter. Hvis vi i eksempelet hadde måttet gjøre 2 tester for hver input hadde vi fortsatt hatt $O(n)$ tid, selv om kjøretiden hadde vært $T(n) = 2n$. Vi kan sette opp noen regneregler for O-notasjon:

Teorem 1.3.2. *Regneregler for O-notasjon:*

i *Hvis* $T(n) = c \cdot f(n)$ *for en konstant* c *og en funksjon* f*, så er* $T(n) = O(f(n))$

ii *Hvis* $T(n) = f(n) + g(n)$ *for to funksjoner* f *og* g*, og det finnes et tall* N *slik at* $f(n) > g(n)$ *for alle* $n > N$ *, så er* $T(n) = O(f(n))$

Dette er ikke veldig storlagne resultater, de følger egentlig direkte fra definisjonen. Det del *i* egentlig sier er at vi kan droppe konstanter når vi jobber med O-notasjon. Vi vil aldri se uttrykk som $O(4n^2)$, vi skriver bare $O(n^2)$. Det er fordi at forskjellen mellom $4n^2$ og n^2 er så liten i forhold til forskjellen mellom n^2 og n^3.

Del *ii* sier at vi bare trenger å se på den største funksjonen. Vi vil for eksempel aldri skrive $O(n^2 + n)$, vi vil bare skrive $O(n^2)$ siden n blir så liten i forhold til n^2.

Eksempel. Beregning av kjøretid. Vi har gitt følgende program

```
1  for (int i=0; i<n; i++) {
2      for (int j=i; j<n; j++) {
3          // Do something simple...
4      }
5  }
6
7  for (int i=0; i<n-3; i++) {
8      // Do something else...
9  }
```

og skal beregne worst case kjøretid til programmet. Vi ser at den indre for-løkka i den øverste for-løkka starter på `i`, og ikke på 0. Fra teorem 1.3.2 del i har vi at det ikke har noe å si. Vi regner med den løkka. Vi ser også at det er en enkel for-løkke etterpå, men fra teorem 1.3.2 del ii har vi at vi kan se bort fra den. Kjøretiden blir altså $O(n^2)$.

♣

Rekursjon

Å finne kjøretid for en rekursiv metode kan være litt mer vrient, men fullstendig gjørbart. Se for eksempel på

```
1  int printNums(int n) {
2  if (n > 0) printNums(n-1);
3  System.out.println(n);
4  }
```

Metoden vil printe alle tall fra 0 til n. Vi ser at for hvert kall får vi ett nytt kall, dette hinter til at det vil være lineær tid. Vi kan sette opp tidsfunksjonen:

$$T(n) = t_{print} + T(n-1) = t_{print} + (t_{print} + T(n-2)) = \cdots = t_{print} + \cdots + t_{print}$$

Vi ser at vi får tilsammen n print-statements, og funksjonen har kompleksitet $O(n)$.

Et annet eksempel:

```
1  void func(int n) {
2  // do something
3
4  if (n>1) func(n/2);
5  }
```

Denne gir $O(\log_2 n)$ siden vi halverer n hver gang. Vi trenger derfor $\log_2 n$ funksjonskall før vi når basistilfelle i rekursjonen.

Eksempel. Finn kjøretid for følgende program: (Ex14 1b)

```
1  for (i=n; i >= 1; i = i/2) {
2      for (j=1; j<n; j++) {
3          // do something
4      }
5  }
```

Vi ser at den indre løkka vil gå n ganger. Den ytre løkka halverer telleren hver gang, det vil si at den kjører $\log_2 n$ ganger. Kjøretiden blir derfor $O(n \log_2 n)$.

♣

Eksempel. Finn kjøretid for følgende program: (Ex11 2b)

```
1  float foo(A) {
2      n = A.length;
3
4      if (n==1) {
5          return A[0];
6      }
7
8      // let A1, A2, A3 and A4 be arrays of length n/2
9
10     for (i=0; i<=n/2; i++) {
11         for (j=0; j<=n/2; j++) {
12             A1[i] = A[i];
13             A2[i] = A[i+j];
14             A3[i] = A[n/2 + j];
15             A4[i] = A[j];
16         }
17     }
18
19     b1 = foo(A1);
20     b2 = foo(A2);
21     b3 = foo(A3);
22     b4 = foo(A4);
23 }
```

Her kan vi ikke like lett se løsninga siden funksjonen er rekursiv. Vi går sakte gjennom hva programmet gjør og forsøker å sette opp en funksjon for kjøretiden. Vi ser at `foo` har to nestede for-løkker, hver av dem går $n/2$ ganger. Vi har derfor at hver gang vi kommer til denne løkka blir kjøretiden $T_{\text{løkke}}(m) = (m/2)^2$. Mot slutten av programmet har vi fire rekursive kall. Alle kallene kjører `foo` med input av lengde $n/2$. Vi kan dermed sette opp en funksjon for kjøretiden:

$$T(n) = C + 4\left(\frac{n}{2}\right)^2 + 4\,T\left(\frac{n}{2}\right)$$

der C er en konstant. Vi kan sette inn for $T(n/2)$:

$$T(n) = C + 4\left(\frac{n}{2}\right)^2 + 4\left(C + 4\left(\frac{n/2}{2}\right)^2 + 4\,T\left(\frac{n/2}{2}\right)\right)$$

Igjen kan vi sette inn for $T(n/2)$, og slik kan vi fortsette. Siden vi halverer n hver gang ser vi at vi må gjøre dette $\log_2(n)$ ganger før vi får at $n = 1$, og rekursjonen stoppes av den øverste if-testen. Vi ser at i hvert rekursjonssteg så gjør vi $O(n^2)$ operasjoner, og siden vi gjør $O(\log_2 n)$ rekusjonssteg, får vi at

$$T(n) = O(n^2 \log_2 n)$$

♣

2 Paradigmer for algoritmedesign

2.1 Splitt og hersk

Splitt og hersk er en teknikk som i stor grad benytter seg av rekursjon. Vi deler problemet opp i mindre delproblemer, deler de delproblemene opp i mindre deldelproblemer også videre. Slik fortsetter vi helt til problemene er så små at løsningen er triviell. Deretter setter vi sammen løsningen på småproblemene til en løsning på hele problemet.

Eksempler på algoritmer som bruker denne teknikken er søking i binære trær (se 4.1) og Quicksort (14.6).

2.2 Grådige algoritmer

Grådige algoritmer er algoritmer som løser optimeringsproblemer. En grådig algoritme vil gå steg for steg gjennom problemet, og gjøre det som ser best ut på hvert tidspunkt.

Eksempler på grådige algoritmer er Dijkstras algoritme (se 10.2), Prims algoritme (11.1), Kruskals algoritme (11.2) og Huffmankoding (12.1).

2.3 Dynamisk programmering

Dynamisk programmering er en designteknikk som går ut på å forsøke å gjøre komplekse optimeringsproblemer enklere ved å dele problemet opp i mindre delproblemer, og løse dem hver for seg. Vi lagrer løsningene, og bruker resultatet fra dem til å konstruere med en endelig løsning. Prinsippet går ut på at en optimal løsning på hele problemet vil være et resultat av optimale løsninger på delproblemene. Det er ikke alltid tilfelle, men når det er det kan dynamisk programmering forbedre kjøretiden dramatisk.

For illustrere tankegangen skal vi se på et eksempel. Fibonaccitallene er definert rekursivt slik:

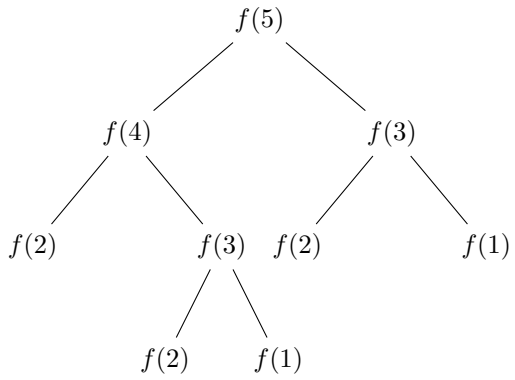
$$f(n) = \begin{cases} 1 & \text{for } n \in \{1, 2\} \\ f(n-1) + f(n-2) & \text{ellers} \end{cases}$$

Vi skal programmere en funksjon som regner ut $f(n)$. Det er fristende å gjøre det helt likt som definisjonen:

```
1  public int recursiveFib(int n) {
2      if (n == 1 || n == 2) {
3          return 1;
4      else {
5          return recursiveFib(n-1) + recursiveFib(n-2);
6      }
7  }
```

Dette er en grei og oversiktlig implementasjon, i den forstand at den finner tallet vi ber den om, men la oss foreta en liten tidsanalyse. For hvert tall vi ber den om må den regne ut to tall. For hver av disse to tallene må vi igjen regne ut to tall. Slik baller det på seg. Vi kan tegne opp et tre over funksjonskallene:

Figur 2.1: Funksjonskall for $f(5)$



Vi ser at vi vil regne ut $f(3)$ to ganger, det virker litt overflødig. Generelt vil denne algoritmen bruke $O(2^n)$ tid, som er veldig dårlig. Hvis vi prøver å tenke dynamisk kan vi løse problemet mye bedre. I stedet for å programmere fibonaccifunksjonen vår rekursivt vil vi gjøre det iterativt, og hvor vi lagrer løsningene underveis. Da kan vi, i stedet for å regne ut de foregående tallene, finne dem i en tabell.

```
1  public int dynamicFib(int n) {
2      int last = 1, lastlast = 1, fibNum = 1;
3
4      for (int i=3; i<=n; i++) {
5          fibNum = last + lastlast;
6          lastlast = last;
7          last = fibNum;
8      }
9
10     return fibNum;
11 }
```

Denne koden er kanskje litt mindre intuitiv enn den forrige, men den er mye raskere! Her ser vi at algoritmen kun har én loop, og kjøretiden er opplagt $O(n)$.

Et eksempel på en algoritme som bruker denne teknikken er Floyds algoritme (se 10.3).

2.4 Kombinatorisk søk

Kombinatorisk søking er en strategi vi bruker når vi ikke har noen andre muligheter. Strategien går i hovedsak ut på å teste alle muligheter. Dette er veldig tregt, og har ofte eksponentiell tid. Når det er sagt kan vi ofte gjøre visse **avskjæringer** for å forbedre tiden. Avskjæringer er når vi kan fjerne mange muligheter fordi vi veit at svaret ikke finnes der uansett.

Eksempler på problemer vi må bruke kombinatorisk søk på er TSP, subset sum eller å plassere åtte dronninger på et skjakkbrett uten at noen står i slag.

3 Bevisføring

3.1 Noen bevisteknikker

Bevis ved selvmotsigelse Teknikken går ut på å gjøre noen antagelser, og vise til at det fører til en motsigelse. Da må noen av antagelsene være gale, og hvis vi kun har to muligheter (for eksempel rett/galt) kan vi konkludere med at det andre alternativet må være rett. Eksempel på et slik bevis er beviset for haltingproblemet (3.3)

Induksjon Bevis ved induksjon går ut på å anta at påstanden holder for alle mulige tall k opp til n, og vise at det også medfører at det også holder for $k = n + 1$. Dermed har man vist at hvis det også holder for $n + 1$ vil det holde for $n + 2$, også videre. Det siste man må gjøre er å *"starte"* induksjonen, med for eksempel å vise at påstanden holder for $k = 1$ (da vil den også holde for $k = 2, 3, ...$)

Bevis ved moteksempel Skal man vise at en påstand er falsk, kan man ganske enkelt finne et eksempel som viser at den opprinnelige påstanden er falsk. Påstår jeg at alle hester er hvite, kan du motbevise den påstanden ved å vise meg en hest som er svart.

3.2 Reduksjon

Reduksjon er å starte med noe ukjent, og *"omforme"* det til noe kjent. Vi kan for eksempel ha et problem, og lure på hvilken kompleksitetsklasse det tilhører. Hvis vi kan vise at en løsning på problemet vårt vil generere en løsning på f.eks TSP-problemet, som vi vet er NP-hard, må problemet vårt også være i NP-hard. Vi ser på noen eksempler:

Anta at vi har et problem A, og vi lurer på om problemet er løselig eller ikke. A kan være veldig vanskelig å forstå, men kanskje vi kan uttrykke det på en annen måte? Hvis vi kan vise at hvis A er løselig er også B løselig, og hvis B er løselig er også C løselig. Slik kan vi fortsette til vi kommer til noe kjent, for eksempel haltingproblemet (3.3). For matematikere:

$$A \text{ er løselig} \Rightarrow B \text{ er løselig} \Rightarrow ... \Rightarrow \text{haltingproblemet er løselig}$$

Hvis vi kan vise en slik rekke av implikasjonspiler har vi *redusert* A til haltingproblemet, og dermed vet vi at A er uløselig.

Vi ser på et annet eksempel. Vi skal vise at mengden av alle løselige problemer er tellbart uendelig. Alle løselige problemer kan løses ved å implementere et program i et turingkomplett programmeringsspråk, for eksempel Java. Dette programmet lagres som en tekstfil på dataen, enkodet i UTF-8 eller en annen enkoding. Denne enkodingen består av en sekvens med `1` og `0`. Denne sekvensen kan sees på som et binært tall, altså et tall i $\mathbb{N}$. Vi har nå *redusert* mengden av løselige problemer til mengden av naturlige tall, som vi vet er tellbart uendelig. Dermed må mengden av løselige problemer også være tellbart uendelig. Det er også mulig å redusere mengden av alle problemer (løselige og uløselige) til $\mathbb{R}$, og dermed vise at mengden av uløselige problemer er større enn mengden av løselige problemer (ved å bruke $|\mathbb{R}| \gg |\mathbb{N}|$ som vi har fra Cantors diagonaliseringsargument)

3.3 Noen beviser

Haltingproblemet

Kan en Turingmaskin avgjøre om en annen Turingmaskin noen sinne vil stoppe, eller om den vil gå i evig loop, om den ser på inputen til maskinen? Som nevnt i 1.1 kan vi definere en språk som mengden av input som vil få en maskin til å stoppe:

$$L = \{i : M \text{ stopper på input } i\}$$

Det kan vises at en Turingmaskin som løser dette problemet *ikke* kan eksistere. Det kan vises på flere måter, Alan Turing beviste det slik:

Vi skal bevise at haltingproblemet er uløselig, og vi skal gjøre det ved selvmotsigelse.

Anta at vi har en Turingmaskin M som bestemmer om en annen Turingmaskin H vil stoppe, gitt input i. Vi kan da bygge en annen Turingmaskin M' rundt denne som tar de samme parametrene (H og i) som input, og gir resultat hvis M gir false (Altså at H ikke vil stoppe, gitt input i), og går i en evig loop hvis M gir true (at H vil stoppe, gitt i).

Hva vil skje hvis vi bruker denne Turingmaskinen på seg selv? Hvis M sier at M' vil stoppe vil M' gå i en evig loop, og følgelig vil ikke M' stoppe. Hvis M sier at M' ikke vil stoppe vil M' gi et resultat, og så stoppe. Uansett får vi en motsigelse, og derfor kan ikke Turingmaskinen M eksistere. □

Beviset er analogt med Cantors diagonaliseringsargument.

Cantors diagonaliseringsargument

Vi skal vise at størrelsen av $\mathbb{R}$ er større enn $\mathbb{N}$, altså at det er flere reelle tall enn naturlige tall. Dette er et eksempel på bevis ved selvmotsigelse. Vi skal anta at vi kan vise at $\mathbb{R}$ og $\mathbb{N}$ er like store, og så vise at det fører til en motsigelse.

Vi kan begynne med noe som kanskje går litt mot intuisjonen. Det er nemlig like mange rasjonelle tall som naturlige tall. For å vise dette må vi vise at vi har en 1-1-korrespondanse mellom naturlige tall og rasjonale tall. Vi starter med å liste opp "*alle*" de naturlige og rasjonale tall. Vi setter rasjonale tall inn i en tabell:

$$\mathbb{N} = 1, 2, 3... \qquad \mathbb{Q} = \begin{array}{c|cccc} & 1 & 2 & 3 & ... \\ \hline 1 & {}^{1}/_{1} & {}^{2}/_{1} & {}^{3}/_{1} & ... \\ 2 & {}^{1}/_{2} & {}^{2}/_{2} & {}^{3}/_{2} & ... \\ 3 & {}^{1}/_{3} & {}^{2}/_{3} & {}^{3}/_{3} & ... \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

Sånn rent intuitivt kan det se ut som at det er mange flere rasjonale tall enn naturlige tall, men hvis vi går på skrå i tabellen over $\mathbb{Q}$ kan vi knytte hvert rasjonalt tall til ett naturlig tall, og hvert naturlige tall til ett rasjonalt tall:

$$1 : \frac{1}{1} \qquad 2 : \frac{2}{1} \qquad 3 : \frac{1}{2} \qquad 4 : \frac{3}{1} \quad ...$$

Vi har dermed vist at vi kan skape en 1-1-korrespondanse mellom $\mathbb{N}$ og $\mathbb{Q}$, og dermed er de like store.

Vi skal nå se på $\mathbb{N}$ og $\mathbb{R}$, altså de reelle tallene. Vi ønsker å prøve å skape en lignende 1-1-kobling som vi gjorde for de rasjonale tallene. $\mathbb{R}$ er ikke tellbart uendelig, derfor kan vi ikke liste opp alle tallene i $\mathbb{R}$ på samme måte som før med en smart tabell (dette er forøvrig grunnen til at $\mathbb{R}$ er større enn $\mathbb{N}$, vi skal nå vise *hvorfor*). En ting vi kan gjøre er å liste opp naturlige tall på en side, og tilfeldige og unike reelle tall på den andre. Det har seg faktisk slik at det er flere reelle tall mellom 0 og 1 enn det er naturlige tall tilsammen, det holder å liste opp tilfeldige reelle tall mellom 0 og 1. Vi får da en slik tabell:

$\mathbb{N}$	$\mathbb{R}$
1	0.182947...
2	0.294817...
3	0.132849...
$\vdots$	$\vdots$

Vi kan tenke oss at hvis vi fortsetter slik i all evighet vil vi til slutt ende opp med en 1-1-korrespondanse mellom $\mathbb{R}$ og $\mathbb{N}$. Vi vil jo aldri gå tom for naturlige eller reelle tall å ta av. Det viser seg likevel å være feil, og det er her Cantors diagonaliseringsargument kommer inn:

Vi kan nemlig lage et nytt reellt tall som ikke eksisterer i denne tabellen. Hvis vi tar første siffer fra første tall og legger til 1, andre siffer fra andre tall og legger til 1, også videre. Generelt tar vi det i-te sifferet i i-te rad og legger til 1. Hvis tallet er 9 kan vi gå til 0[1]. Gjør vi det for denne tabellen får vi:

$$0.203...$$

Siden dette tallet er ulikt tall nummer i i tabellen på i-te siffer vet vi at det ikke er inneholdt i tabellen, og vi har dermed vist at det ikke kan lages en 1-1 korrespondanse mellom $\mathbb{R}$ og $\mathbb{N}$. Følgelig er $\mathbb{R}$ større enn $\mathbb{N}$ □

[1]Formelt kan vi bruke klokkeaddisjon (moduloaddisjon): $(9 + 1) \mod 10 = 0$

Høyden til et komplett binært tre

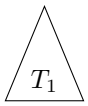
Vi skal vise at høyden til et komplett binært tre er $\log(n+1)$. Før vi gjør det skal vi vise et hjelperesultat: nemlig at antall noder n i et komplett binært tre er $2^n - 1$. Vi skal vise dette ved induksjon.

Vi starter med å vise at formelen gjelder for et tre med høyde $h = 1$:

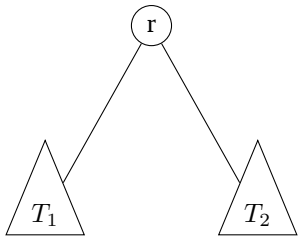
$$n = 2^1 - 1 = 1$$

som åpenbart stemmer siden et tre med høyde 1 har kun 1 node, nemlig rota.

Vi antar at formelen gjelder for alle h opp til n, og skal vise at det impliserer at den også gjelder for $n + 1$. Vi har et tre T_1 med høyde n:



Vi antar at formelen stemmer, og dermed at antall noder i T_1 er $2^n - 1$. Vi skal nå øke høyden til $h + 1$. Vi lager en ny rot, med to fulle subtrær som barn:



Fra induksjonsantagelsen har vi at T_1 og T_2 har $2^n - 1$ noder hver. Legger vi sammen antall noder i treet nå får vi:

$$n \;=\; \text{noder i } T_1 + \text{noder i } T_2 + 1 \;\leq\; 2(2^n - 1) + 1 \;=\; 2^{n+1} - 1$$

Altså: Vi har vist at hvis formelen gjelder for $h = k$ gjelder den også for $h = ks+1$, vi har vist at den gjelder for $h = 1$, og dermed har vi vist at den gjelder for alle $h \in \mathbb{N}$ □

Vi kan nå vise det endelige resultatet. Vi har at antall noder n er gitt slik:

$$n = 2^h - 1$$

Vi flytter over 1, og tar logaritmen på begge sider:

$$\begin{aligned} n &= 2^h - 1 \\ n + 1 &= 2^h \\ \log_2(n+1) &= \log_2(2^h) \\ \log_2(n+1) &= h \log(2) \\ \log_2(n+1) &= h \end{aligned}$$

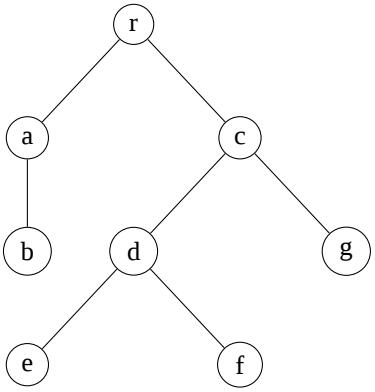
Del II

Datastrukturer

4 Trær

Et tre en et spesielt tilfelle av en rettet graf der hver node har inngrad 1 (med unntak av rota i treet). Vi ser på et eksempel:

Figur 4.1: Et lite binært tre



Terminologi

Vi skal se litt på ord og uttrykk for trær. Gjennomgående bruker vi treet i figur 4.1 som eksempel

I figuren blir nodene tegnet som rundinger. For å betegne relasjonen mellom nodene bruker vi ofte familierelasjoner. Vi sier at e og f er **søsken**, d er **forelder** til e, og e er **barn** av d. Vi kan også si at g er **onkel** til f, men dette er mindre vanlig, da vi sjeldent har bruk for å snakke om *"onkelnoder"*.

Nodene r, a, c og d kalles **indre noder**, det vil si at disse nodene har barn. Noder som ikke har noen barn kalles **løvnoder**

I figur 4.1 er r **rotnoden**. Rota i treet er den eneste noden uten noen foreldre. Rota er derfor et naturlig startpunkt når vi skal søke eller traversere gjennom treet.

4.1 Binære søketrær

Binære søketrær er trær med noen spesielle krav. Hver node kan ikke ha mer enn to barn, vi kaller dem ofte venstre og høyre barn. Venstre barn er alltid mindre enn noden selv, og høyre barn er alltid større enn noden. Dette gjør binære søketrær meget godt egnet for søking.

Teorem 4.1.1. *Å sette inn, fjerne eller søke etter noder i et binært søketre har*

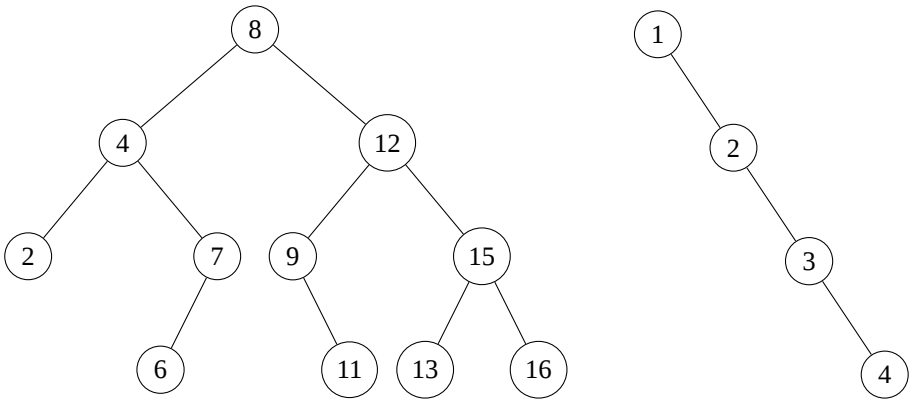
i I beste fall $O(\log n)$ tid

ii I verste fall $O(n)$ tid

Vi skal ikke bevise teorem 4.1.1 her[1], men vi kan se på et eksempel som viser ytterpunktene. I figur 4.2 ser vi to forskjellige binære søketrær. I treet til venstre har vi et fint, balansert tre. Det er lett å se at høyden (og dermed antall operasjoner vi må gjøre for å komme til bunn i treet) er lik $\lceil \log_2 n \rceil$ der n er antall noder i treet.

I treet til høyre har hver node kun ett barn. Vi har i praksis en lenkeliste. Hvis vi skal søke etter 4, må vi tråkke gjennom alle de andre nodene for å komme dit.

Figur 4.2: To eksempler på binære søketrær



Innsetting

Når vi skal sette inn en node i et binært søketre starter vi i rota. Vi sammenligner verdien vi skal sette inn med verdien i rota. Hvis verdien vi setter inn er mindre enn rota går vil til venstre, er den større går vi til høyre. Hva som skjer ved likhet er opp til oss å bestemme, men vi må være konsekvente. Når vi kommer til en nullpeker kan vi sette denne pekeren til å peke på noden vi setter inn.

[1] Det kan vises ved induksjon at høyden til et binært tre i beste fall er $\log_2 n$

Vi kan implementere denne funksjonen rekursivt. I en ytre klasse kan vi skrive en skallfunksjon som kaller på rotas `insert`-metode. Vi har en indre `Node`-klasse med en rekursiv insertmetode. Den kan implementeres slik:

```
1  public void insert(T otherElement) {
2      int comparison = element.compareTo(otherElement);
3
4      if (comparison > 0) {
5          // This element is bigger than the one we're
6          // inserting => left subtree
7          if (leftChild == null) {
8
9              // nothing there, insert
10             leftChild = new Node(otherElement);
11             return;
12         } else {
13
14             // occupied, move down
15             leftChild.insert(otherElement);
16         }
17
18     } else if (comparison < 0) {
19         // The element we're inserting is bigger than
20         // this one => right subtree
21         if (rightChild == null) {
22
23             // nothing there, insert
24             rightChild = new Node(otherElement);
25             return;
26         } else {
27
28             // occupied, mode down
29             rightChild.insert(otherElement);
30         }
31     } else {
32         // The two elements are the same. Since this is a tree
33         // for searching, we'll just forget about it
34         return;
35 }}
```

Sletting

Når vi skal fjerne en node fra et binært søketre har vi tre forskjellige situasjoner som vi må se på.

Noden har ingen barn (løvnode). Denne situasjonen er ganske grei. Siden noden ikke har noen barn å forholde seg til er det bare å fjerne den fra treet.

Noden har ett barn. For å fjerne en node a med ett barn kan vi ganske enkelt flytte pekeren fra foreldernoden til a, til barnet til a.

Noden har to barn. Hvis noden vi skal fjerne har to barn tar vi opp den minste noden som er større en noden vi skal fjerne og flytter den til den posisjonen noden vi skulle fjærne var. Vi finner den minste noden som er større ved å gå ett steg til høyre, og så til venstre helt til vi er i bunn av treet.

Eksempel på implementasjon i Java (rekursiv metode i en `Node`-klasse):

```java
public void remove() {
    if (leftChild == null && rightChild == null) {
        // no children, just unlink
        if (parent.leftChild == this)
            parent.leftChild = null;
        else
            parent.rightChild = null;

    } else if (leftChild == null || rightChild == null) {

        // one child: unlink, and set parents link to this' child
        if (parent.leftChild == this)
            parent.leftChild = this.leftChild;
        else
            parent.rightChild = this.rightChild;

    } else {
        // two children: take the smallest element to the right and
        // replace with this.
        Node replacement = rightChild.findSmallestChild();

        // I will edit the content of this and delete the
        // replacement node, instead of deleting this and edit the
        // pointers of the replacement node.
        this.element = replacement.getElement();
        replacement.remove();

    }
}
```

Søking

Når vi skal søke etter et element i et søketre starter vi i rota og sammenligner det elementet vi søker etter med det elementet som finnes i rota. Er elementet vår mindre enn noden vi ser på går vi til venstre, og er det større går vi til høyre. Deretter foretar vi samme testen på nytt og fortsetter slik til vi enten finner elementet vi leter etter, eller kommer til en nullpeker (bunnen av treet).

Eksempel på implementasjon i Java:

```java
public Node find(T otherElement) {
    int comparison = element.compareTo(otherElement);

    if (comparison == 0) {
        return this;

    } else if (comparison < 0) {
        // This element is smaller than the one we're searching for, go left
        if (leftChild == null) {
            // the element doesn't exist
            return null;

        } else {
            return leftChild.find(otherElement);
        }

    } else {
        // This element is bigger than the one we're searching for, go right
        if (rightChild == null) {
            // the element doesn't exist
            return null;

        } else {
            return rightChild.find(otherElement);
        }
    }
}
```

4.2 Rød-svarte trær

Vanlige binære søketrær blir fort veldig ubalanserte. Prøv å sett inn 1, 2, 3, 4, 5, 6, 7, ... (i den rekkefølgen) i et binært søketre. Da vil vi i praksis ha en lenket liste. Da er også kjøretiden for å søke, sette inn og slette i verste fall $O(n)$, i stedet for $O(\log n)$ som er det vi vanligvis har for søketrær, og som er det vi ønsker. Vi trenger derfor trær som vil balansere seg selv. Vi skal se på rød-svarte trær, men det finnes også andre typer (som AVL-trær)

Rødsvarte trær er binære søketrær med noen spesielle strukturkrav. Kravene er designet for å motkjempe skjevhet (som illustrert i figur 4.2), og dermed forbedre kjøretid.

Vi deler nodene opp i to kategorier, røde og svarte. Vi følger noen bestemte regler på hvordan vi skal farge nodene, og fargen på nodene avgjør som vi må benytte oss av rotasjon eller ikke (se 4.4).

Definisjon 4.2.1. Et rød-svart tre er et binært søketre med noen spesielle tilleggsregler:

i Roten er svart.

ii Hvis en node er rød, må barna være svarte.

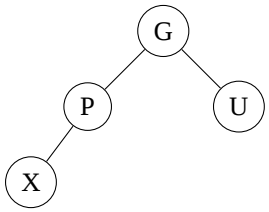
iii Enhver vei fra en node til en null-peker må inneholde samme antall svarte noder.

Fargen til en node er ikke statisk, den kan endre seg. Disse reglene sikrer at høyden maksimalt er $2\log_2(n+1)$ (bevis for dette finnes på Wikipedia).

Insetting

Vi zoomer inn på en del av treet og jobber kun med de nodene som har noe å si for innsetting. X er noden vi skal sette inn, P er foreldernoden, U er onkelnoden (søsken av forelder), G er besteforeldernoden. Når vi setter inn ei node i et rød-svart tre er den nye noda alltid rød. Deretter må vi se på fargene til P, U og G for å finne ut om vi må ta noen videre steg:

Figur 4.3: Situasjon for innsetting



Teorem 4.2.2. *Innsetting i rød-svarte trær*

- *Hvis P er svart er alt OK, og vi er ferdig*
- *Hvis P er rød må vi ta noen steg for å opprettholde krav ii og iii:*
  - *Hvis U også er rød: Farg P og U svarte og farg G rød. Dette kan føre til trøbbel videre oppover i treet, så vi må fortsette oppover å rette opp feil.*
  - *Hvis U er svart, utfør en rotasjon, og utfør nødvendige fargeendringer:*
    - *X venstre barn av P: Zig mhp P.*
    - *X høyre barn av P: Zig-zag mhp X.*
- *Hvis rota på noe tidspunkt blir farget rød kan man bare farge den svart igjen.*

Sletting

Sletting fra rød-svarte trær handler til syvende og sist om å slette løvnoder. Skal vi slette en indre node kan vi gå en plass til høyre, og deretter gå til venstre helt til vi er i bunnen av treet, og swappe de to nodene. Deretter kan vi slette (det som nå er) løvnoden. Når vi gjør dette kan det hende vi bryter med noen av kravene i definisjon **??**. Vi må derfor bruke samme type strategi som for innsetting for å rette opp feil.

Sletting er noe mer komplisert enn innsetting. Se seksjon 12.2.3 i læreboka for gjennomgang av en del situasjoner med figurer og forklaringer.

Tidsanalyse

Siden høyden til et rød-svart tre maksimalt er $2\log_2(n+1)$ har vi worst case tid $O(\log_2 n)$. Selv om vi risikerer å måtte gjøre rotasjoner er innsetningstiden også $O(\log_2 n)$.

4.3 B-trær

B-trær er konstruert for å effektivisere antall disklesninger, og gir mening å bruke hvis vi har et tre som er så stort at det må lagres på gammeldagse spinnedisker, og ikke i RAM. Vi lagrer dataene i blokker, og leser en og en blokk av gangen. Alle dataene er lagret i løvnodene, mens de indre nodene brukes for søking. En vanlig måte å implementere B-trær på er å ha så mye av toppen i RAM som mulig, og lagre resten på disk. B-trær er ikke binære, dvs at de kan ha flere enn 2 barn.

Definisjon 4.3.1. La M angi antall mulige nøkler i hver indre node, og L angi maksimalt antall dataelementer i hver løvnode. B-trær er søketrær der følgende kriterier er oppfylt:

i Alle dataene er lagret i løvnodene

ii Interne noder lagrer inntil $M-1$ nøkler for søking: nøkkel i angir den minste verdien i subtre $i+1$.

iii Roten er enten en løvnode, eller har mellom 2 og M barn.

iv Alle andre indre noder har mellom $\lceil M/2 \rceil$ og M barn.

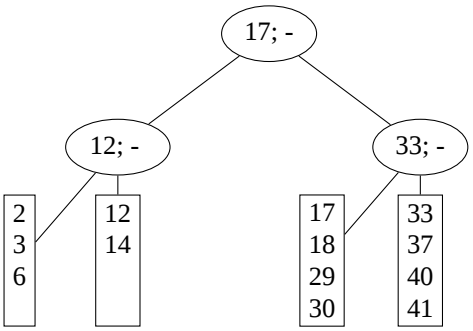
v Alle løvnoder har samme dybde.

vi Alle løvnoder har mellom $\lceil L/2 \rceil$ og L dataelementer

Innsetting

Når vi skal sette inn et element i et B-tre finner vi plassen elementet skal på, og setter det der. Hvis løvnoden vi setter elementet inn i er full må vi splitte noden i to like store deler. Vi må da oppdatere nøklene i foreldrenoden P. Hvis P ikke har plass til en ekstra nøkkel må vi splitte den også, og oppdatere nøklene i foreldrenoden til P. Slik fortsetter vi oppover til vi får en node som har plass til en ekstra nøkkel. Den eneste måten et B-tre kan øke høyden på er at rota splittes i to, og at vi lager en ny rot.

Eksempel. Sett inn 13 og 42 i følgende B-tre ($M = 3$ og $L = 4$):

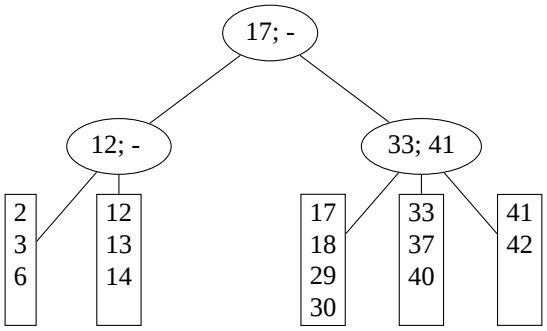


Vi starter med å sette inn 13. Vi ser først på rota: (17; -). 13 er mindre enn 17, så vi går til første barn. Så sammenligner vi med (12; -). 13 er sørre enn 13, og mindre enn uendelig (- symboliserer minste verdi i tredje barn, men siden det barnet ikke eksisterer tenker vi på verdier som ∞). Vi går derfor til andre barn. Der ser vi at vi kan sette inn 13 uten problemer.

Vi skal nå sette inn 42. 42 er større enn 33 så vi går til andre barn. Igjen er 42 større enn 33 så vi går til andre barn igjen. Her ser vi at hvis vi legger til 42 i bunn av noden (33, 37, 40, 41) blir størrelsen lik 5, altså større enn N. Vi må splitte noden. Når vi

splitter en node med et odde antall elementer er det ikke helt klart som vi skal ta med oss størsteparten, eller la størsteparten være igjen. I dette tilfellet vil vi få en node med 3 og en node med 2 elementer. Vi kan selv velge om den nye noden skal ha 2 eller 3 elementer, men vi må være konsekvente. I dette tilfellet lar vi den nye noden ha 2 elementer. Vi oppdaterer foreldrenoden. Siden den har to barn fra før og $M = 3$ går det fint, vi trenger ikke splitte videre oppover.

Resultatet etter innsetting ser slik ut:

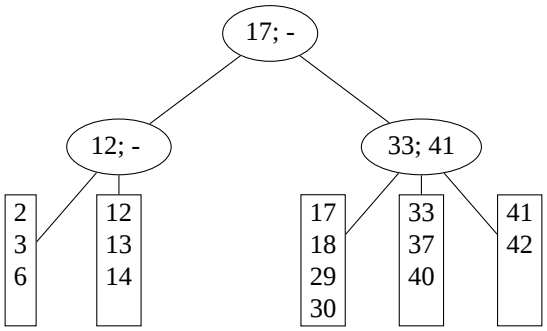


♣

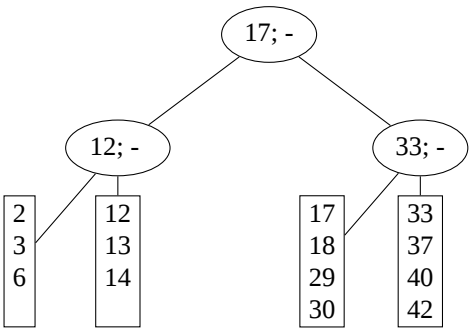
Sletting

Hvis vi skal fjerne et element fra et B-tre søker vi opp elementet, tar det vekk, og hvis mengden elementer i noden blir mindre enn grensen satt i def. 4.3.1.vi slår vi sammen to noder. Den eneste måten B-trær kan minke i høyde på er om alle barna til rota blir slått sammen til én node (vi fjerner da rota).

Eksempel. Fjern 41 fra følgende B-tre:



Vi ser at 41 er mellom 17 og ∞, så vi går til andre barn. Deretter ser vi at 41 er større enn 33 og større enn eller lik 41. Vi går derfor til tredje barn og fjerner 41. Vi ser at da vi noden kun ha ett element, og vi slår den sammen med noden før. Resultatet etter fjerning blir



♣

Tidsanalyse

Følgende teorem angir kompliksteten til de ulike operasjonene på et B-tre:

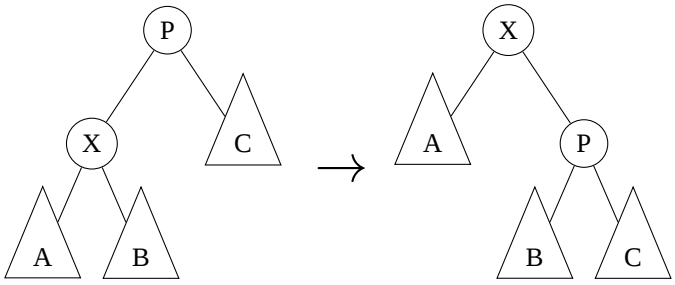
Teorem 4.3.2. *Å søke etter, sette inn og fjerne elementer fra et B-tre tar, både i beste og verste fall,* $O(\log n)$ *tid.*

4.4 Rotasjon

Zig

For å skjønne zig-rotasjon ser vi på en figur:

Figur 4.4: Zig-rotasjon mhp X



En zig-rotasjon vil beholde egenskapene til et binært søketre siden alle nodene i A er mindre enn X og P, alle nodene i B er større enn X og mindre enn P, og alle nodene

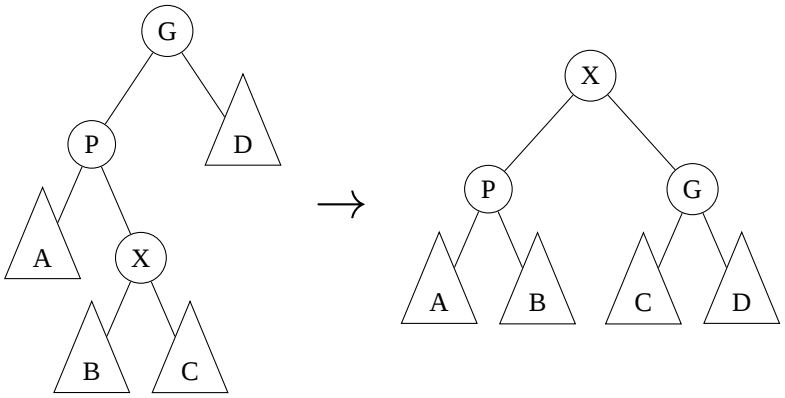
i C er større enn både X og P. Vi ser at alle høyre barn av P også er høyre barn av P *etter* rotasjonen, etc.

Vi har også noe som heter zig-zig-rotasjon, som er å utføre en zig-rotasjon to ganger.

Zig-zag

Igjen ser vi på en figur:

Figur 4.5: Zig-zag-rotasjon mhp X:



Som i tilfellet med zig-rotasjon ser vi at egenskapene til et binært søketre er bevart.

5 Grafer

En graf er en samling av kanter og noder. Vi kaller mengden av alle nodene V (vertices), og mengden av alle kantene E (edges). Grafen G blir da en samling av disse to mengdene. Formelt kan vi definere en graf slik:

Definisjon 5.0.1. En graf G er et par (V, E), der V er en ikke-tom mengde noder og E en mengde nodepar $\langle v_1, v_2 \rangle$; $v_1, v_2 \in V$ der $\langle v_1, v_2 \rangle$ angir at grafen inneholder en kant fra v_1 til v_2.

Terminologi

Vi bruker $|E|$ for å betegne antall kanter og $|V|$ for å betegne antall noder.

Vi sier at en node er **nabo** med en annen node hvis de har en kant mellom seg. I figur 5.1 er B og A naboer, men A og C er ikke naboer.

En **sti** eller **vei** er en sekvens av noder (og kantene mellom dem) fra en node til en annen. En sti fra A til G i figuren under kan for eksempel være A, D, C, G. (siden grafen inneholder løkker er ikke stien entydig)

En graf er **rettet** hvis kantene har en spesiell retning, og **urettet** hvis vi ikke bryr oss om retningen på kantene. I en rettet graf vil kanten $\{v_1, v_2\}$ bety at det går en kant fra v_1 til v_2, men ikke nødvendigvis fra v_2 til v_1. Grafen i figur 4.1 er urettet. Hvis en graf er rettet kaller vi ofte naboene for **etterfølgere**.

En graf er **vektet** hvis kantene har en verdi knyttet til seg, ofte kalt *kosten* til kanten. I en **uvektet** graf har ikke kantene noen spesiell verdi. Vi kan tenke på en uvektet graf som en vektet graf der alle kantene har kost = 1.

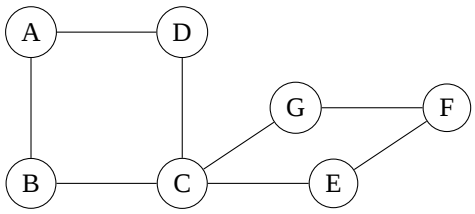
Grafen er **syklisk** hvis den inneholder løkker, og **asyklisk** hvis den ikke inneholder løkker. I figur 5.1 ser vi at A, B, C, D danner en løkke (det gjør også C, G, F, E). Grafen er derfor syklisk.

En type grafer vi jobber mye med er **DAG**er. DAG står for *Directed Asyclic Graph*, på norsk: en rettet, asyklisk graf.

Implementasjon

Det finnes i hovedsak to måter å implementere en graf på, **Adjacency matrix** (nabomatrise) og **Adjacency list** (naboliste). Vi kan også implementere grafen som instanser av en `Node`-klasse med pekere til sine naboer, men det er mindre utbredt (dette vil også

Figur 5.1: Eksempel på en urettet, uvektet graf



være en slags implementasjon av en naboliste). Måten vi implementerer grafen på har faktisk en del å si for kompleksiteten til enkelte algoritmer.

Nabolisten til en graf er en liste over nodene og deres tilhørende naboer. Nabomatrisen er en $n \times n$-matrise A der $a_{i,j}$ angir vekten til kanten fra node nr i til node nr j. Hvis det ikke finnes en kant er $a_{i,j} = 0$, og som diskutert tidligere kan en uvektet graf sees på som en vektet graf der alle vektene er 1.

For tynne grafer er nabolister klart mer plasseffektive enn nabomatriser. Nabolister gir også bedre kjøretid i enkelte algoritmer, som for eksempel Prims algoritme, som er $O(n^2)$ hvis grafen er representert ved en matrise, og $O(n \log n)$ hvis grafen er representert som en liste.

Figur 5.2: Grafen i figur 5.1 representert som naboliste (venstre) og nabomatrise (høyre)

Node	Naboer
A	B, D
B	A, C
C	B, D, G, E
D	A, C
E	C, F
F	E, G
G	C, F

$$\begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Merk at for en urettet graf vil nabomatrisen alltid være symmetrisk, siden at kantene går begge veier.

5.1 Hamiltonske og eulerske stier

En hamiltonsk sti er en sti som besøker alle *nodene* én gang. Dette kan virke som en triviell sak, men i en del tilfeller er det faktisk ikke mulig å finne en hamiltonsk sti. I grafen i figur 5.1 er B, A, D, C, G, F, E en hamiltonsk sti (det finnes fler). Å avgjøre om en generell graf har en hamiltonsk sti eller ikke er et NP-komplett problem.

En eulersk sti er en sti som besøker alle *kantene* én gang. I grafen i figur 5.1 er C, B, A, D, C, G, F, E, C en eulersk sti. Den er lukket siden den starter og slutter i samme node. Hvis stien slutter på en ulik node enn den startet på er stien åpen. Et kjent eksempel på en graf som ikke har noen eulersk sti er broene i Königsberg.

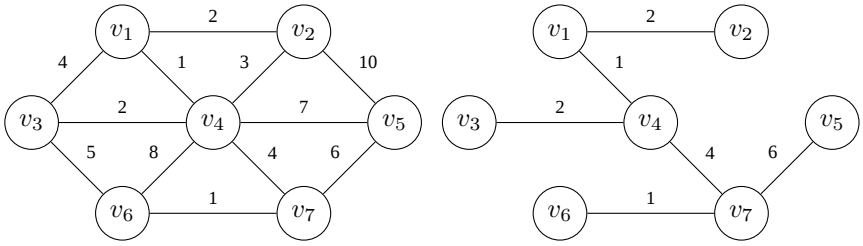
5.2 Spenntrær

Vi begynner med en definisjon:

Definisjon 5.2.1. Et **spenntre** for en urettet graf G er et tre med kanter fra grafen slik at alle nodene i G er forbundet. Spesielt er et **minimalt spenntre** de(t) spenntre(ene) med lavest total kostnad.

Minimale spenntrær er sjeldent entydig bestemt. Vi har et entydig minimalt spenntre hvis alle kantene har forskjellig vekt. Vi tegner opp et eksempel:

Figur 5.3: En graf (venstre) og et minimalt spenntre for grafen (høyre)

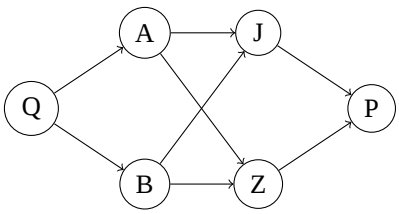


For å finne et minimalt spenntre til en graf kan vi bruke Prims algoritme eller Kruskals algoritme (se 11.2)

5.3 Topologisk sortering

Topologisk sortering er en ordning av noder i en DAG slik at dersom det finnes en vei fra v_i til v_j i grafen kommer v_i før v_j i sorteringa. En topologisk sortering er ikke nødvendigvis entydig bestemt, ofte finnes det veldig mange. Hvis en graf er syklisk eksisterer det ikke en topologisk sortering av grafen. Vi ser på et eksempel:

Eksempel. Gitt følgende graf, finn alle topologiske sorteringer.



Vi ser at Q har inngrad 0, det er derfor et logisk sted å begynne. Etter Q kommer A og B. Vi ser at vi kan ikke gå videre fra A før uten å ha vært innom B, siden både J og Z er avhengige av B. Tilsvarende kan vi ikke gå videre fra B før vi har vært innom A. Til slutt ser vi at P er avhengig av både J og Z. Begge de to må derfor komme før P i sorteringa. Vi har da dekt alle mulige utfall. Vi kan liste opp alle mulige sorteringer:

- Q, A, B, J, Z, P
- Q, B, A, J, Z, P
- Q, A, B, Z, J, P
- Q, B, A, Z, J, P

Vi kan ta med et moteksempel også: Q, A, J, B, Z, P er ikke en gyldig topologisk sortering siden J kommer før B i sorteringa, men i grafen ser vi at J er avhengig av B (det går en kant fra B til J).

♣

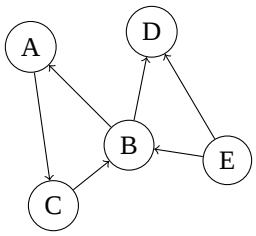
5.4 Strongly connected components (SCC)

Vi ser på definisjonen av strongly connected components (heretter SCC):

Definisjon 5.4.1. Anta at vi har en rettet graf $G = (V, E)$. Vi har da at en **SCC** av G er et maksimalt sett av noder $U \subseteq V$ slik at for alle u_i, $u_j \in U$ har vi at $u_i \rightsquigarrow u_j$ og $u_j \rightsquigarrow u_i$.

Med andre ord: en SCC er en del (partisjon) av grafen der alle nodene i den partisjonen kan nå alle andre noder i partisjonen. Vi ser på et eksempel:

Eksempel. Finn alle SCCer i grafen: Vi ser at {A, B, C} danner en SCC. D har ingen



kanter ut. Den må derfor være sin egen SCC. E ingen kanter inn, den må også være sin egen SCC. Vi har da at vi kan partisjonere grafen slik: {{A, B, C}, {D}, {E} }

♣

5.5 Articulation points og biconnectivity

Et **articulation point** er et kritisk punkt i en sammenhengende graf som ikke kan fjernes uten at grafen ikke lenger er sammenhengende. I grafen i figur 5.1 er C et slikt punkt. Vi ser at vi kan ta vekk A uten noen store komplikasjoner, men hvis vi tar vekk C vil ikke lenger B og E være sammenknyttet.

En **biconnected** graf er en graf uten articulation points, altså en graf der det alltid finnes mer enn én vei fra en node til en annen. Grafen (til venstre) i figur 5.3 er en slik graf.

6 Heap

6.1 Heap (prioritetskø)

En heap (også kalt prioritetskø) er en type binært tre med noen spesielle struktur- og ordningskrav. Vi har to typer heap: min- og maksheap. Vi vil beskrive minheap i dette kapitlet, men maksheap fremgår helt analogt.

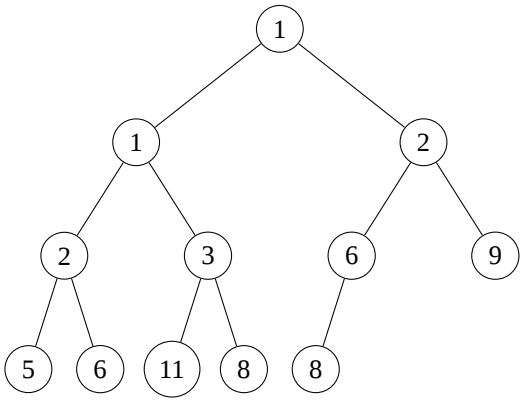
Definisjon 6.1.1. En minheap er et binært tre der følgende krav er oppfylt:

1. Treet er komplett
2. En node har alltid (sorterings)verdi mindre enn, eller lik sine barn.

En maksheap defineres nesten likt, eneste forskjell er at pt. 2 i definisjonen blir "En node alltid er større enn, eller lik sine barn."

Hver node i en heap inneholder to ting: et element, og en verdi vi sorterer etter. I motsetning til i et binært søketre der vi sorterer på elementet selv, vil vi med en heap tilordne en sorteringsverdi til hvert element som ikke trenger å ha noe med elementet å gjøre.

Figur 6.1: En (min)heap. For oversiktens skyld er kun sorteringsverdiene tatt med.



Innsetting

Når vi skal sette inn et element i en heap setter vi den på første ledige plassen. Deretter sammenligner vi nodens verdi med forelderens verdi. Hvis forelderen har større verdi enn noden vi setter inn bytter vi plass[1]. Så sammenligner vi med den nye forelderen, bytter plass om nødvendig, og fortsetter slik til enten forelderen er mindre enn noden, eller at noden er rot.

Fjerning (deleteMin)

I en heap er vi egentlig bare interessert i å ta ut det minste elementet fra en heap (hvorfor blir diskutert i avsnittet om anvendelser). På grunn av krav 2 i definisjonen vet vi at rota er det minste elementet i heapen. Derfor fjerner vi rota. For å skaffe en ny rot tar vi det siste elementet i heapen og setter som rot. Deretter sammenligner vi verdien av den nye rota med verdien av barna. Hvis et (eller begge) av barna har mindre verdi enn den nye rota bytter rota plass med **det minste** barnet[2]. Vi fortsetter slik til ordningskravet er oppfylt (noden har mindre verdi enn barna).

Implementasjon

Vi kan implementere en heap som et tre (dvs, lage nodeobjekter med pekere til barn etc), men som oftest implementerer vi en heap ved hjelp av en array. Vi lar rota være på index 1. På grunn av kompletthetskravet kan vi legge nodene radvis bak rota. Vi finner da barna til en node på index i ved å til index $2i$ (venstre barn) og $2i + 1$ (høyre barn), og forelder ved å gå til index $\lfloor i/2 \rfloor$. Når vi implementerer en heap som en array kan vi risikere å gå tom for plass i arrayen. Da må vi lage en ny array, og flytte over alle elementene til den nye arrayen. Vanligvis legger vi til en og en rad i slengen (eventuelt to og to, tre og tre, etc..). Å gjøre dette tar åpenbart $O(n)$ tid, men vi må gjøre det sjeldnere og sjeldnere jo større heapen blir.

Figur 6.2: Heapen i fig 6.1 representert som array

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
verdi	null	1	1	2	2	3	6	9	5	6	11	8	8	null	null	null

I Java har vi en ferdig heap: `java.util.PriorityQueue<E>`. Java krever at `E` er sammenlignbar med seg selv (`E` implementerer `comparable<E>`) og vil bruke den sammenligningen som grunnlag for sortering. Javas heap er implementert som array.

[1] Foreleser og lærebok kaller denne prosessen *"percolate up"*
[2] *"percolate down"*

Tidsanalyse

Vi ser på kjøretiden til de to omtalte operajonene:

Teorem 6.1.2. *Kjøretid for heapoperasjoner*

1. *Innsetting i en heap tar $O(\log_2 n)$ tid*
2. *Fjerne minste element tar $O(\log_2 n)$ tid*

Beviset følger av strukturkravet i definisjon 6.1.1:

Bevis. Teorem 6.1.2, del 1:
Når vi setter inn et element i en heap må vi først sette elementet på slutten av heapen. Siden vi vet hvor siste element er vil dette ta $O(1)$ tid. Deretter må vi justere plassen til elementet ved å la elementet sive oppover. Siden treet er komplett vil høyden på treet være $\lceil \log_2(n+1) \rceil$[3]. Vi vil maksimalt foreta $\lceil \log_2(n+1) \rceil - 1$ byttinger. Total tid blir derfor være $O(\log_2 n)$ □

Argumentet for teorem 6.1.2, del 2 er helt analogt.

Anvendelser

I dette kurset ser vi på tre anvendelser av en heap: Prioritetskø, sortering og Huffmankoding. Når vi bruker en (min)heap som en prioritetskø lar vi viktige jobber ha lav verdi, og mindre viktige jobber ha høy verdi. Når vi setter jobbene våre inn i en heap og tar dem ut vil de viktigste jobbene komme først. Fordelen med dette mot å bare sortere lista med jobber er at vi dynamisk kan legge til flere jobber underveis.

Vi kan også bruke en heap til å sortere ei liste. Se 14.4 for detaljer.

En siste anvendelse av heaps er i huffmankoding. Se 12.1.

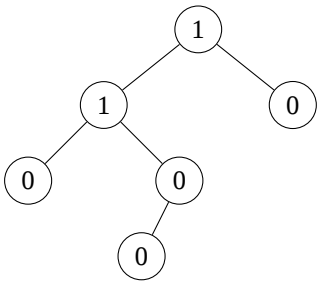
6.2 Venstreorientert (leftist) heap

En venstreorientert heap er en en heap med et annet strukturkrav enn vanlig heap. Motivasjonen bak venstreorienterte heaper er at å flette (merge) to heaper av samme størrelse tar $O(n)$ tid, vi ønsker å forbedre det. Før vi kan definere en venstreorientert heap må vi definere *"null path length"* (herfra: npl).

[3] se bevis for høyden av et komplett tre i 3.3

Definisjon 6.2.1. Npl til en node n er lengden av den korteste veien fra n til en etterkommer uten to barn (0 hvis n har < 2 barn).

Figur 6.3: Et tre med npl(n) skrevet inn



Vi er nå klare til definisjonen av venstreorientert heap:

Definisjon 6.2.2. En venstreorientert heap er et binært tre der følgende krav er oppfylt:

1. For hver node n i treet er npl$(l) \geq$ npl(r), der l er venstre og r er høyre barn til n.
2. En node har alltid (sorterings)verdi mindre enn, eller lik sine barn.

Merk: En venstreorientert heap forsøker å være ute av balanse (for å gjøre fletting enklere).

Fletting, innsetting og sletting

Hovedoperasjonen på venstreorienterte heaper er **fletting** (engelsk: merging). Fletting er forklart i teorem 6.2.3

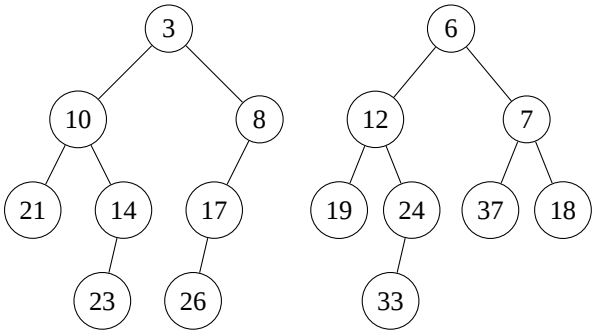
For å **sette inn** en node i en venstreorientert heap kan vi flette en heap bestående av den ene noden vi vil sette inn, med hele heapen vi vil sette noden inn i.

For å **fjerne** minste node kan vi ta vekk rota (som vi vet er minst fra def. pt. 2), og flette venstre og høyre subheap.

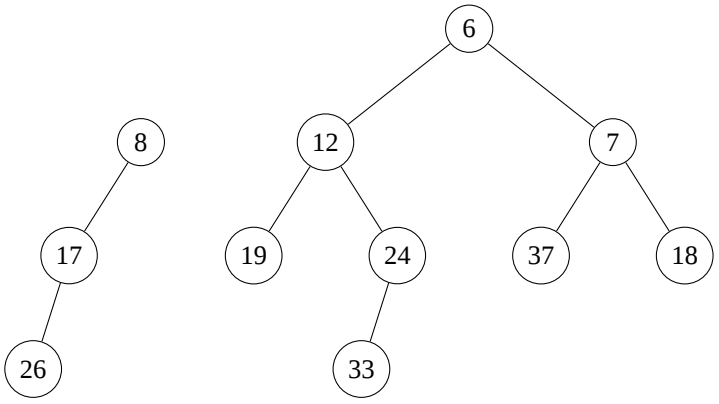
Teorem 6.2.3. *La H_1 og H_2 være to venstreorienterte heaper. Vi definerer fletting reukursivt:*

- *Sammenlign rota i H_1 og H_2. Antar videre at H_1 er minst.*
- *La høyre subheap til H_1 være heapen som framkommer av å flette høyre subheap av H_1 med H_2. Fortsett slik til problemet er trivielt.*
- *Underveis i oppnøstinga etter rekursjonen kan høyre og venstre supheap swapes for å opprettholde strukturkrav 1 i def 6.2.2*

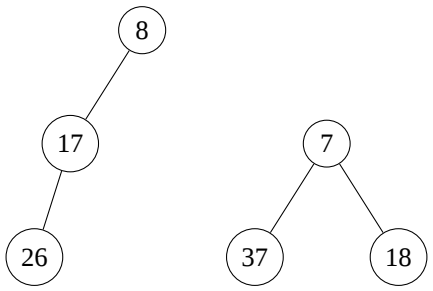
Eksempel. Flett følgende to venstreorienterte heaper:



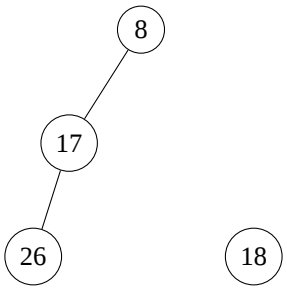
Vi kaller heapen til venstre for H_1 og heapen til høyre for H_2. Vi begynner med å sammenligne røttene (steg 0): $3 < 6$, så fletter høyre subheap av H_1 med H_2, altså:



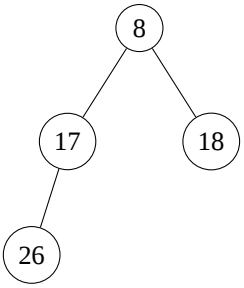
Steg 1: nå ser vi at $6 < 8$, dermed fletter vi venstre heap med høyre subheap av høyre heap:



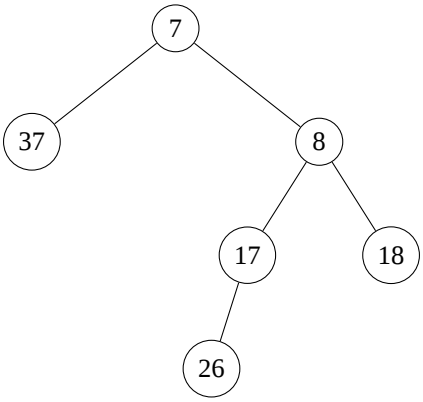
Steg 2: siden $7 < 8$ fletter vi heapen til venstre med høyre subheap av heapen til høyre:



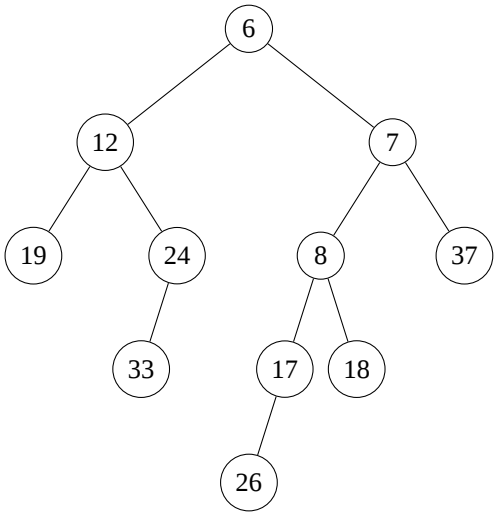
Steg 3: dette er trivielt, og blir basisen i rekursjonen. Vi fletter sammen heapene på vanlig måte:



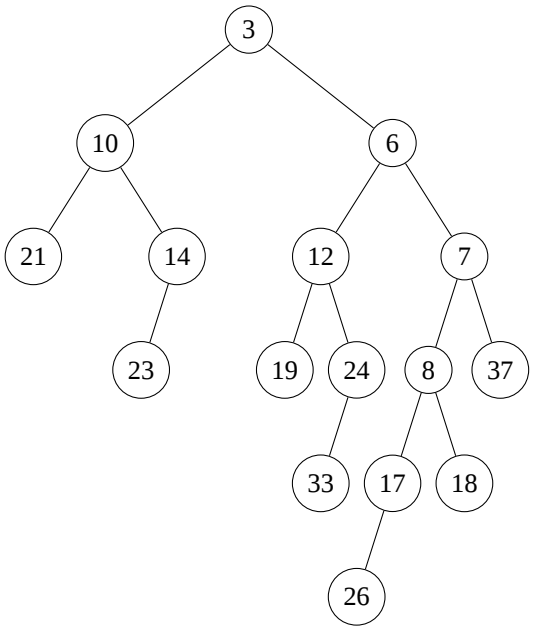
Nå begynner oppnøstingen i rekursjonen. Vi setter inn denne i steg 2 (erstatter høyre subheap i høyre heap med denne):



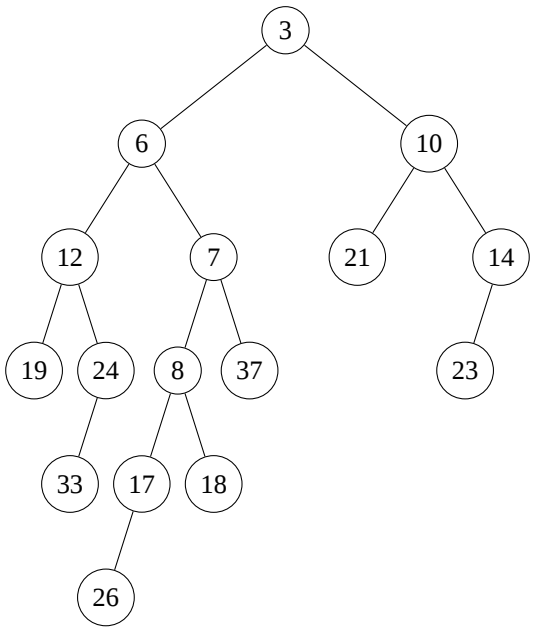
Denne heapen oppfyller ikke krav 1 i definisjon 6.2.2, vi må derfor swappe høyre og venstre supheap. Vi gjør dette og setter inn i steg 1:



Npl er 2 i begge subheaps, vi kan derfor sette rett inn i steg 0:



Nå er vi nesten ferdige, vi ser at npl til venstre subheap er 1, og npl til høyre subheap er 2, vi må derfor swape venstre og høyre subheap:


3
6
10
12
7
21
14
19
24
8
37
23
33
17
18
26

♣

7 Map

En map er en form for ADT som inneholder par av nøkler og verdier. Hvis vi for eksempel skal lagre aldere til mennesker gir det ingen mening å kun lagre verdiene i en array/liste: `[21, 19, 32, 25]`. Hvis vi derimot lagrer par med (navn, alder), er situasjonen en annen:

Anne	21
Knut	19
Nils	32
Kari	25

I Java har vi interfacet `Map<K,V>` som beskriver denne funksjonaliteten. De med python-bakgrunn har jobbet med maps før, men kalt dem *dictionaries* i stedet. Dictionaries i Python er implementert som hashmaps bak kulissene. `OrderedDict` i Python er implementert etter samme prinsipp tom `TreeMap<K,V>` i Java

7.1 Hashmap (hashtabell)

Gitt map-en i forrige seksjon. Hvis jeg spør: *"hva er alderen til Kari?"*, så må man rett og slett søke gjennom hele lista, i $O(n)$ tid.

Den grunnleggende idéen bak hashtabeller er å la nøkkelen til elementet vi skal sette inn bestemme indexen i en array. Vi bruker en hashfunksjon $H : \mathbb{N} \to \mathbb{N}$ på nøkkelen vår x, og lar $H(x)$ betegne indexen til elementet i en array.

- Når vi oppretter en hashtabell lager vi en ny array som er $tableSize$ lang. Størrelsen på arrayen er et primtall.
- Når vi skal sette inn et element x i en hashtabell regner vi ut $H(x)$ og setter x inn i arrayen på plass $H(x)$
- Når vi skal søke etter et element x i hashtabellen regner vi ut $H(x)$ og slår opp på den plassen i arrayen. Hvis elementet er der har vi funnet det. Hvis elementet ikke er der må vi muligens sjekke noen andre steder. Hvor vi må lete videre avhenger av hvordan vi håndterer like hashresultater.
- Når vi skal fjerne et element x i en hashtabell søker vi opp elementet i tabellen, og sletter det.

Vi har ofte en hash-funksjon på formen:

$$H(x) = f(x) \mod tableSize$$

Håndtering av like hashresultater

Hashfunksjoner trenger ikke å være injektive (de er i praksis aldri det). Det vil si at vi fort kan få $f(x) = f(y)$ selv om $x \neq y$. Hvis vi skal sette inn y i en hashtabell, men plassen med index $f(y)$ allerede er okkupert av x må vi finne en ny plass til y. Det kan gjøres på flere måter, vi deler opp i to hovedgrupper: Seperate chaining og probing.

Seperate chaining En mulig løsning på problemet med like hashresultater er å la arrayen vår være en array av lenkede lister. Vi kan dermed ha flere elementer i hver index. Når vi skal sette inn x, regner vi ut $f(x)$, og setter y bakerst i lista på den plassen. Dermed har det ingenting å si om det eksisterer elementer på indexen eller ikke.

Problemet med seperate chaining er at vi må tråkke oss gjennom en lenket liste etter at vi har funnet hashresultatet. Dette tar lineær tid (men dog med (forhåpentligvis) færre elementer enn n)

Lineær probing En annen mulig løsning på problemet er å gå til plassen bak $f(x)$, og hvis den er opptatt går vi til indexen bak det igjen. Skulle vi komme til enden av arrayen begynner vi fra toppen igjen. Generelt for probing har vi at plassen vi setter elementet inn på er gitt ved

$$index = (f(x) + g(k)) \mod tableSize$$

der f er hashfunksjonen, g er i dette tilfellet $g(k) = k$, k er antall skritt vi har tatt fra den opprinnelige indexen (antall ganger vi har støtt på et element) og $tableSize$ er antall plasser i tabellen. Igjen får vi det problemet at vi i praksis må søke igjennom en liste etter å ha funnet hashresultatet.

Kvadratisk probing Kvadratisk probing ligner veldig på lineær probing, med den forskjellen at $g(k) = k^2$. Vi går altså til den plassen som ligger $1, 4, 9, ...$ plasser bak $f(x)$

Dobbel hashing Med dobbel hashing har vi en ekstra hashfunksjon $f_2(x)$ som vi regner ut hvis vi skulle støte på problemer. Vi får da at $g(k) = kf_2(x)$. Et eksempel på en funksjon f_2 kan være $f_2(x) = R - (x \mod R)$ der R er det største primtallet som er mindre enn $tableSize$. Den totale hashfunksjonen blir dermed

$$H(x; i) = (f_1(x) + if_2(x)) \mod tableSize$$

der i er antall kollisoner, og er 0 når vi starter.

Rehashing

Hvis hashtabellen begynner å bli full kan det lønne seg å rehashe. Det betyr ganske greit å lage en ny tabell med større $tableSize$ og flytte alle elementet over. Når vi rehasher lager vi som regel en tabell som er ca dobbelt så stor (men fortsatt primtall). Siden $H(x)$ ofte avhenger av $tableSize$ må vi regne ut alle hashverdiene på nytt. Rehashing er derfor en ganske dyr affære, så vi gjør det veldig sjeldent, men hvis tabellen begynner å bli for full kan vi tjene ganske mye tid i lengden.

Gode hashfunksjoner

Vi står helt fritt til å velge hashfunksjoner selv. Gode hashfunksjoner er raske å regne ut, kan gi alle mulige verdier fra 0 til $tableSize - 1$, og har en god fordeling (spredning). utover tabellindexene.

Ofte bruker vi strenger som nøkler. Vi må derfor ha en måte å regne ut et tall av en tekststreng på. Det kan gjøres på mange måter, her er et par eksempler:

- Summer verdiene til hvert tegn:

$$H(x) = \left(\sum_{i=0}^{keyLength-1} = \text{charVal}(key_i) \right) \mod tableSize$$

  der charVal er en funksjon som tilordner en numerisk verdi til hver bokstav, feks ascii/unicode-verdien til tegnet. key_i betegner det i-te tegnet i key. Denne funksjonen er rask og enkel å implementere, men vil gi dårlig spredning for store tabeller.

- En vektet sum av de tre første tegnene:

$$H(x) = (c_1 \text{ charVal}(key_1) + c_2 \text{ charVal}(key_2) + c_3 \text{ charVal}(key_3)) \mod tableSize$$

  der c-ene er konstanter vi velger. Denne er rask å enkel å beregne, og gir en grei fordeling for tilfeldige strenger, problemet er at naturlig språk ikke er tilfeldig.

Tidsanalyse

Det er åpenbart at best case tid for innsetting, søking og fjerning i hashtabeller er $O(1)$. Det oppstår når vi kommer rett til elementet/tom index på første forsøk. Worst case er når vi får kollisjoner på hvert eneste treff, og vi i praksis har en liste. Da vil alle operasjoner på tabellen være $O(n)$.

Vi ser at en tabell starter som $O(1)$, og beveger seg mot $O(n)$ når den blir fullere. Det er derfor vi ofte velger å rehashe når tabellen blir for full. Selv om rehashing har $O(n)$ tid, er det en operasjon vi gjør én gang. Det vil forbedre kjøretiden på alle andre operasjoner drastisk, siden det er mindre sannsynlighet for å få kollisjoner i en tabell med mindre tetthet. Som regel prøver vi å holde andelen av tabellen som er opptatt (kalt *"load factor"*, ofte forkortet LF) under en gitt grense. Java sin innebygde `HashMap<K,V>` tvinger $LF < 0.75$.

7.2 Extendible hashing

Utvidbar (extendible) hashing er en annen måte å lagre hashresultater på enn å stappe alt i en array og la hashverdien betegne indexen, slik vi gjodre med hashtabeller. Motivasjonen bak utvidbar hashing er litt den samme som bak B-trær, vi ønsker å minimere antall disklesninger for store datamengder. Før vi definerer utvidbar hashing må vi introdusere en ny type struktur, nemlig bøtter (eng: buckets).

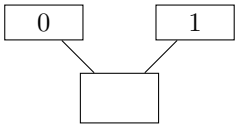
Definisjon 7.2.1. En *bøtte* (eng: bucket) er en mengde elementer med en gitt øvre størrelse M.

Bøtter er kanskje et nytt begrep, men vi har allerede sett dem. Løvnodene i B-trær er et eksempel på bøtter.

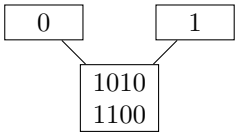
For å gjøre saken litt enklere nå skal vi kun se på hashverdien av nøklene, men i virkeligheten er det vi sender inn til en utvidbar hashtabell det samme som for vanlige hashtabeller, altså nøkkel-verdi-par.

Den første observasjonen vi gjør oss er at resultatet av en hashfunksjon alltid er en bitstreng. For enkelhetens skyld skal vi nå jobbe med 4-bits hashfunksjoner. Ideen bak utvidbar hashing er å bruke de første n bitene av hashverdien som prefiks, og la alle verdiene med samme prefiks ligge i samme bøtte. Vi begynner med å bare bruke det første bitet som prefiks, dvs $n = 1$, og øker n når bøttene blir fulle. Dette blir tydeligst ved å se på et eksempel.

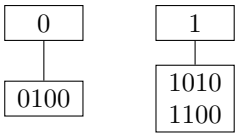
Eksempel. Vi skal sette inn hashverdiene 1100, 1010, 0100, 1110 og 0010 i en utvidbar hashtabell. Bøttene har størrelse $M = 2$. I utgangspunktet ser situasjonen slik ut:



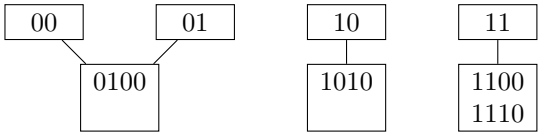
Merk at både 0- og 1-prefikset går til samme bøtte. Etter å ha satt inn 1100 og 1010 ser situasjonen slik ut:



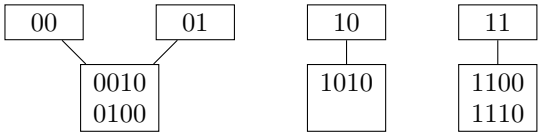
Hvis vi nå prøver å sette inn 0100 ser vi at bøtta blir full, vi splitter bøtta i to, og sorterer slik at de med 0-prefiks kommer i en bøtte og de med 1-prefiks kommer i den andre:



Når vi nå prøver å sette inn 1110 støter vi på enda et problem: For det første er 1-erbøtta full og må splittes, men det er ikke noen flere indeksnoder og ta av. Vi må derfor øke n til 2, og se på de to første bitene som prefiks:



Til slutt setter vi inn 0010, vi ser at 00 er 2-prefiks og går i bøtta som 00-noden peker på. Det endelige resultatet er alstå:



♣

Vi ser at vi kommer fram til riktig bøtte på kun 2 diskaksesser, som er det i tråd med hva vi ville oppnå med utvidbar hashing.

De aller fleste filsystemer i bruk i dag er implementert som enten B-trær eller utvidbare hashtabeller.

8 Abstrakte datatyper

8.1 ADT

En ADT (*eng: Abstract Data Type*) er en generalisering av datastrukturer. Det er en matematisk modell som beskriver semantikken til en datastruktur fra brukerens synsvinkel, men ikke konkret hvordan den blir implementert.

Definisjon 8.1.1. En ADT består av en mengde elementer, og en mengde operasjoner på disse elementene.

Eksempler

- FIFO. Mengden er alle elementene, mulige operasjoner er legg til of ta ut, der elementet som tas ut er det første som ble lagt til.
- LIFO. Mengden er alle elementene. Operasjoner: push og pop. I tillegg kan vi snakke om opersjoner som peek.
- Map. Mengden er alle nøkkel/verdi-parene, under operasjonene legg til par, fjern par, modifiser par og søk opp verdi basert på nøkkel. Eksempel på en implementasjon av ADTen *"map"* er en Hashmap eller Treemap
- Prioritetskø. Mengden er alle elementene i køen, operasjonene er legg til, og ta ut elementet med høyest prioritet. Eksempel på implementasjoner av denne ADTen er heap og leftist heap. Man kan også bruke et binært søketre til å implementere en prioritetskø.
- For matematikere: $\mathbb{Z}$ under operasjonene $\{+, -, \cdot, /\}$ danner en ADT. Vi bryr oss ikke om hvordan heltallsaritmetikk er implementert på datamaskinen, men vi vet hvordan $+$ burde oppføre seg.

8.2 Kø/stack

Kø og stack er to forskjellige typer lister. Felles for dem er at vi kun opererer med én `insert()`- og én `remove()`-metode. Forskjellen er hvilket element vi vil hente ut når vi kaller `remove()`-metoden. Innsetting og fjerning i både kø og stack tar $O(1)$ tid, siden vi på forhånd vet hvor elementet skal fjernes fra/legges til.

Kø (FIFO)

En kø er en liste der første element inn er det første som kommer ut, derav navnet. Vi kaller også slike lister for FIFO-lister (First In, First Out). Elementene som settes inn stilles bakerst i køen, og når vi henter ut et element starter vi foran.

Stack (LIFO)

En stack (norsk: stabel) er en liste der siste element som settes inn er det første som kommer ut. Slike lister kalles også LIFO-lister (Last In, First Out). Elementene som settes inn legges på toppen av stabelen, og når vi skal hente ut et element henter vi det øverste elementet.

Konvensjonelt kalles insert-metoden i en stack `push()`, og remove-funksjonen `pop()`

Del III

Noen konkrete algoritmer

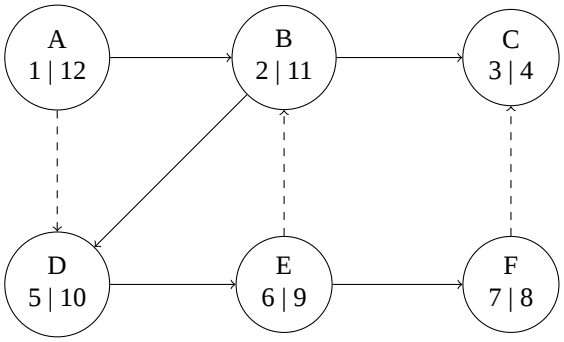
9 Traversering av grafer

9.1 Dybde-først-søk

Dybde-først-traversering (herfra dfs: depth first search) er en måte å traversere gjennom en graf på. Det kan tenkes på som en generalisering av dybde-først-traversering gjennom trær. Vi velger oss en (hvilken som helst) startnode, og beveger oss ned til etterfølgerne til noden. Når vi kommer til en node som ikke har kanter til en ubesøkt node (enten at noden har utgrad 0, eller at alle etterfølgerne er besøkt) går vi tilbake.

Underveis i traverseringen har vi en teller som holder styr på hvor mange skritt vi har tatt. I hver node lagrer vi verdien av telleren prefix og postfix, det vil si første gang vi kommer til noden og når vi kommer tilbake etter å ha vært nedom barna.

Figur 9.1: En rettet graf med tallene lagret etter et dfs. Heltrukne linjer er kanter vi besøkte i dfs-et, striplede linjer er kanter vi ikke besøkte



I figur 9.1 har vi brukt A som rotnode. Vi gikk deretter til et av barna, B. Videre gikk vi til C, og siden C har utgrad 0 måtte vi gå tilbake til B igjen. Fra B gikk vi videre til D, så til E og til F. Fra F går det en kant til C, men siden vi har besøkt C fra før går vi tilbake igjen.

DFS-algorigmer er velegnet for å programmere rekursivt. Et eksempel på en enkel implementasjon i Java (som en metode i en Node-klasse):

```java
public void dfs(int num) {
    if (this.visited) return;

    this.lowNum = num;
    num++;

    for (Node child : this.children) {
        child.dfs(num);
    }

    this.highNum = num;
    num++;

    this.visited = true;
}
```

9.2 Kosarajus algoritme (Finne SCC)

Vi kan finne SCCer i en graf G ved å utføre to dfs. Et på G og et på G^t, som er grafen vi får ved å snu alle kantene i G. Denne algoritmen kalles Kosarajus algoritme.

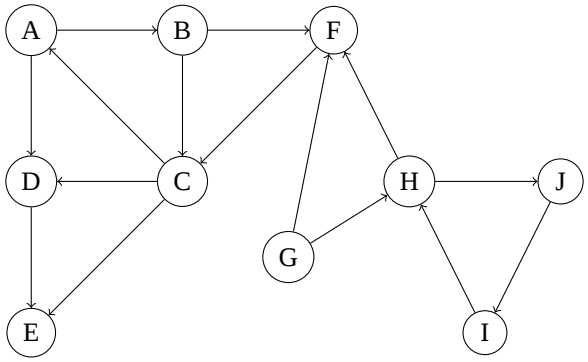
Teorem 9.2.1. *Kosarajus algoritme*

La G være en rettet graf. Utfør et dfs på G fra en hvilken som helst startnode, og lagre postfix-telleren (det som ble kalt `highNum` *i kodeekempelet i 9.1). Hvis søket slutter før alle nodene er besøkt, starter vi fra en ny, ubesøkt node.*

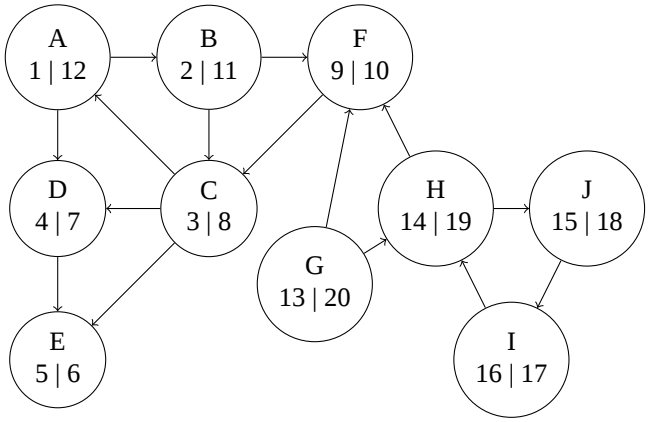
Snu alle kantene i (transponer) G og dann G^t. Sorter nodene i G synkende etter verdien på postfix-telleren. Start et dfs i G^t fra den noden som har høyest verdi. Alle nodene vi da besøker danner en SCC. Fjern så alle nodene vi besøkte fra grafen G^t, og utfør et nytt dfs fra den noden som nå har høyest tellerverdi. Fortsett slik til alle nodene er besøkt.

Vi ser på et eksempel:

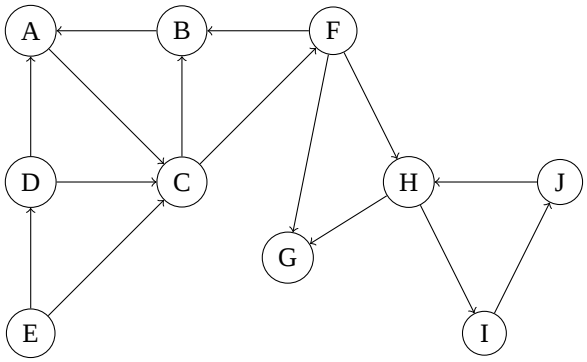
Eksempel. Finn alle SCCer i grafen:



Vi begynner med å gjøre et dfs. Vi starter i A. Søket stopper etter å ha besøkt A, B, C, D, E og F, og må starte på nytt i G. Startpunktene er valg tilfeldig, og kunne like gjerne vært noen andre.



Det hadde vært nok å bare lagre postfixverdiene, men vi har tatt med begge for oversiktens skyld. Vi transponerer G:



Siden G har høyest postfixteller (20) begynner vi der. Vi ser i G^t at G ikke har noen kanter ut. Første SCC er derfor {G}. Når vi tar vekk G fra grafen er det H som har høyest postfixverdi (19), vi starter derfor neste dfs derfra. Vi går fra H til I og J, før vi treffer på H igjen. Vi går tilbake, men siden hverken J eller I har flere etterfølgere stopper søket, og neste SCC er derfor {H, I, J}. Nå er det A som har høyest postfixteller (12), så vi starter derfra. Fra A går vi til C, og til B. Da møter vi på A igjen, så vi går tilbake til C og videre til F. Vi går *ikke* fra F til G eller H, siden de allerede er besøkt tidligere. Vi går derfor tilbake til C og tilbake til A. I det søke besøkte vi {A, B, C, F}, det blir derfor neste SCC. Til slutt starter vi i D (7), men siden D ikke har noen kanter til ubesøkte noder stopper søket med en gang. Det samme skjer i E (6).

Alle SCCer til G er: { {G}, {H, I, J}, {A, B, C, F}, {D}, {E} }

Det finnes en forbedring av denne algoritmen som bare bruker ett DFS. Søk opp *Tarjans algoritme*

Kompleksitet

Vi ser at vi må gjøre to DFS, og en transponering. DFS besøker alle noder og alle kanter én gang, og går derfor i $O(|V| + |E|)$. Transponering må besøke alle kantene én gang, og går derfor i $O(|E|)$ tid. Tilsammen har Kosarajus algoritme $O(|V|+|E|)$.

10 Korteste vei

10.1 Bredde-først-søk

Hvis grafen vi skal finne korteste vei, en til alle, er uvektet kan vi gjøre et enkelt bredde-først-søk (ofte forkortet bfs siden akronymet blir likt på engelsk). Vi starter i noden vi skal ta avstand fra, merkrer den som avstand 0, går til alle etterfølgere, merker dem som avstand 1, går til deres etterfølgere igjen, merker som avstand 2, og fortsetter slik til det ikke er mer å gjøre igjen.

10.2 Dijkstras algoritme

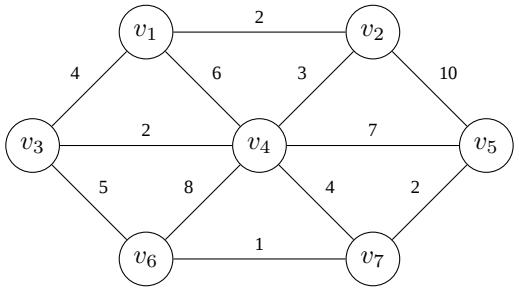
Dijkstras algoritme er en algoritme for å finne korteste vei fra en node til alle andre noder i en urettet graf. Algoritmen er i utgangspunktet for vektede grafer, men kan også anvendes på uvektede. Vi tenker da på en uvektet graf som en vektet graf der alle vektene er like ($= 1$). Algoritmen er grådig siden vi alltid velger den noden med lavest avstand, og fortsetter derfra.

Teorem 10.2.1. *Dijkstras algoritme*
Vi skal finne korteste vei fra en node s til alle andre noder i en vektet, urettet graf G.

1. *For alle noder: Sett avstanden fra startnoden s lik ∞. Sett avstanden fra s til seg selv lik 0*
2. *Velg den naboen v til s med lavest avstand og marker den som kjent.*
3. *For hver nabonode w til v: Dersom avstanden vi får ved å følge veien gjennom v er kortere enn den gamle avstanden, reduserer vi avstanden fra s til w, og setter bakoverpekeren i w til v.*

Vi ser på et eksempel:

Eksempel. Finn korteste vei fra $v1$ til alle andre noder:



Vi begynner med å sette opp en tabell over nodene:

Node	Kjent	Avstand	Vei
v_1	T	0	-
v_2	F	∞	-
v_3	F	∞	-
v_4	F	∞	-
v_5	F	∞	-
v_6	F	∞	-
v_7	F	∞	-

Vi ser på kantene ut fra v_1. Vi oppdaterer veien til v_2, v_3 og v_4:

Node	Kjent	Avstand	Vei
v_1	T	0	-
v_2	F	2	$v_1 \leftarrow v_2$
v_3	F	4	$v_1 \leftarrow v_3$
v_4	F	6	$v_1 \leftarrow v_4$
v_5	F	∞	-
v_6	F	∞	-
v_7	F	∞	-

Nå kommer Dijkstras grådige kriterium: Vi velger den noden som har lavest avstand til v_1 og som ikke er kjent, altså v_2 og fortsetter derfra. Fra v_2 har vi kant til v_4 og v_5. Ved å gå via v_2 får vi litt kortere avstand mellom v_4 og v_1. Vi oppdaterer tabellen:

Node	Kjent	Avstand	Vei
v_1	T	0	-
v_2	T	2	$v_1 \leftarrow v_2$
v_3	F	4	$v_1 \leftarrow v_3$
v_4	F	5	$v_1 \leftarrow v_2 \leftarrow v_4$
v_5	F	12	$v_1 \leftarrow v_2 \leftarrow v_5$
v_6	F	∞	-
v_7	F	∞	-

Neste ukjente node med lavest avstand er v_3. Vi fortsetter derfra. Vi hopper over et par steg. Slutttabellen ser slik ut:

Node	Kjent	Avstand	Vei
v_1	T	0	-
v_2	T	2	$v_1 \leftarrow v_2$
v_3	T	4	$v_1 \leftarrow v_3$
v_4	T	5	$v_1 \leftarrow v_2 \leftarrow v_4$
v_5	T	11	$v_1 \leftarrow v_2 \leftarrow v_4 \leftarrow v_7 \leftarrow v_5$
v_6	T	9	$v_1 \leftarrow v_3 \leftarrow v_6$
v_7	T	9	$v_1 \leftarrow v_2 \leftarrow v_4 \leftarrow v_7$

♣

Tidsanalyse

Hvis vi søker gjennom grafen etter den noden med lavest avstand vil det ta $O(|V|)$ tid, og dette gjør vi $|V|$ ganger. Total tid for å finne minste avstand er derfor $O(|V|^2)$. I tillegg må vi oppdatere avstandene. Det er maksimalt én oppdatering for hver kant, det vil si at det tilsammen tar $O(|E|)$. Total kjøretid for algoritmen er derfor:

$$\text{Kjøretid} = |V|^2 + |E| = O\left(|V|^2\right)$$

10.3 Floyds algoritme

Floyds algoritme er en algoritme som beregner korteste vei fra alle noder til alle andre noder. Algoritmen er et eksempel på en algoritme som benytter seg av dynamisk programmering.

Teorem 10.3.1. *Floyds algoritme*
Denne betraktningen gjentas systematisk for alle tripler i, k og j:

- *Initielt: Avstanden fra node i til node j settes lik vekten på kanten fra i til j, uendelig hvis det ikke går noen kant fra i til j*
- *Trinn 0: Se etter forbedringer ved å velge node 0 som mellomnode*
- *Etter trinn k: Avstanden mellom to noder er den korteste veien som bare bruker nodene $0, 1, ..., k$ som mellomnoder*

Eksempel på implementasjon i Java:

```java
public static void floyd(int[][] naboer, int[][] avstander, int[][] vei) {
    int n = avstand.length

    // Initialisering:
    for (int i=0; i<n; i++) {
        for (int j=0; j<n; j++) {
            avstand[i][j] = naboer[i][j];
            vei[i][j] = -1;
        }
    }

    // Finne veier:
    for (int k=0; k<n; k++) {
        for (int i=0; i<n; i++){
            for (int j=0; j<n; j++){
                if (avstand[i][k] + avstand[k][j] < avstand[i][j]) {
                    avstand[i][j] = avstand[i][k] + avstand[k][j];
                    vei[i][j] = k;
                }
            }
        }
    }
}
```

Tidsanalyse

Vi ser fra kodeeksempelet at det er to løkker som må kjøres gjennom. En dobbel og en trippel for-løkke. Kjøretiden blir derfor:

$$\text{Kjøretid} \;=\; n^2 + n^3 \;=\; O\left(n^3\right)$$

Floyds algoritme har forøvrig samme kompleksitet som å kjøre Dijkstras algoritme én gang for alle nodene i grafen. Fordelen med Floyds algoritme er at den er litt mer robust i møte med løkker og negative vekter.

Hvordan tolke input/output fra Floyds algoritme

Ved start inneholder `nabo[i][j]` lengden av kanten fra i til j, `nabo[i][j]`= $|\infty$ hvis det ikke er noen kant. Algoritmen lar `nabo` være uendret, og legger resultatene i `avstand` og `vei`. `avstand[i][j]` er lengden av korteste vei fra i til j. Når vi oppdager at den korteste veien fra i til j går gjennom k setter vi `vei[i][j] = k`. `vei` sier derfor om den korteste veien. Vi finner korteste vei slik:

- `k1 = vei[i][j]` er den største verdien av k slik at k ligger på den korteste veien fra i til j
- `k2 = vei[i][k1]` er den største verdien av k slik at k ligger på den korteste veien fra i til k_1

  $\vdots$
- Når `vei[i][km] = -1` er (i, k_m) den første kanten i korteste vei fra i til j

Dermed er korteste vei fra i til j slik: $i \to k_m \to k_{m-1} \to \ldots \to k_2 \to k_1 \to j$.

11 Minimale spenntrær

11.1 Prims algoritme

Prims algoritme er en algoritme for å finne minimale spenntrær. Det er en grådig algoritme siden den bygger treet trinnvis ved å velge den kanten som går ut av treet med lavest vekt.

Teorem 11.1.1. *Prims algoritme*
Vi skal finne et minimalt spenntre for en graf G. Velg først en startnode (rotnode). Den kan velges helt tilfeldig. Så lenge treet ikke spenner hele G legger vi til den kanten fra treet, til en ukjent node som har lavest vekt.

Vi ser på et eksempel.

Eksempel. Finn et minimalt spenntre for grafen i figur 5.3.

Vi velger v_1 som rotnode (vi kunne valgt hvilken som helst annen). Den kanten med lavest vekt ut fra treet (som kun består av v_1) er kanten $v_1 - v_4$, så vi legger den til. Videre er kanten $v_1 - v_2$ den kanten med lavest vekt som går til en ukjent node, så vi legger til den. $v_4 - v_3$ er neste kant vi legger til. Kanten $v_2 - v_4$ er nå den kanten med lavest vekt som går ut fra treet, men siden v_2 og v_4 begge er inneholdt i treet er ikke den interessant. Neste kant vi legger til er $v_4 - v_7$, deretter $v_7 - v_6$ og $v_7 - v_5$. Treet er nå et komplett spenntre for G.

♣

Tidsanalyse

Samme argument som tidsanalysen til Dijkstra. Total kjøretid: $O\left(|V|^2\right)$

11.2 Kruskals algoritme

Kruskals algoritme er en annen algoritme for å finne minimale spenntrær. Det er også en grådig algoritme, siden vi til enhver tid velger den kanten med lavest vekt som ikke danner en løkke.

Teorem 11.2.1. *Kruskals algoritme*
Vi skal finne et minimalt spenntre for en graf G. La F være en skog (en mengde av trær). Så lenge F ikke utspenner alle nodene velger vi den kanten med lavest vekt fra G, og som ikke danner en løkke, og legger den til i F. Når F har blitt ett sammenhengende tre og alle nodene er nådd har vi et minimalt spenntre.

Vi ser på et eksempel.

Eksempel. Finn et minimalt spenntre for grafen i figur 5.3.

Vi finner kanten(e) i grafen med den minste vekten. $v_6 - v_7$ og $v_1 - v_4$ har begge vekt 1. Vi legger de til i spenntreet. Kantene $v_3 - v_4$ og $v_1 - v_2$ har begge vekt 2, og ingen av dem danner en løkke, så vi legger dem til i spenntreet. Neste kant med lavest vekt er $v_2 - v_4$ (3), men den vil danne en løkke, så vi hopper over den. Videre ser vi at $v_1 - v_3$ og $v_4 - v_7$ begge har vekt 4. Kanten $v_1 - v_3$ vil danne en løkke så den hopper vi over, men $v_4 - v_7$ legger vi til. Neste kant med lavest vekt, og som ikke danner en løkke er $v_5 - v_7$. Vi legger den til i spenntreet, og da er alle noder nådd, og vi har en sammenhengende graf. Vi har derfor et minimalt spenntre (se figur 5.3).

♣

Tidsanalyse

Vi må loope gjennom alle kantene én gang, det gir tid $O(|E|)$. Vi kan implementere algoritmen ved hjelp av en stack. Gjør vi det må vi utføre én sletting, to søk og en union. Sletting og søk tar $O(\log_2 n)$ tid (for sletting er $n = |E|$, for søk er $n = |V|$), og union tar $O(1)$ tid. For hver iterasjon har vi da:

$$\text{Kjøretid, hver iterasjon} \;=\; O(\log_2 |E| + \log_2 |V| + 1) \;=\; O(\log_2 |V|)$$

Siste likhet har vi siden $|V| > |E|$. Vi kan dermed sette opp total kjøretid:

$$\text{Kjøretid, total} \;=\; O(|E| \cdot \log_2 |V|)$$

Prim vs Kruskal

Prim er som regel noe mer effektiv enn Kruskal, spesielt på tette grafer. Prim har likevel den svakheten at den kun fungerer på sammenhengende grafer. Kruskal anvendt på en usammenhengende graf vil gi et minimalt spenntre for hver sammenhengende del av grafen.

12 Komprimering

12.1 Huffmankoding

Huffmankoding er en måte å kode informasjon på (her tekst) på en måte der vi ikke mister noe informasjon (*lossless*). Den forutsetter at vi har tilgang på teksten før vi lager en kode. Huffmankoding går ut på å se på frekvensen til tegnene som forekommer, og gi kort kode til de tegnene som forekommer ofte, og lang kode til de som forekommer sjelden. Når vi velger disse kodene er det viktig at vi beholder det vi kaller *"prefiksegenskapen:"* Ingen koder er et prefiks av noen andre. Kodealfabetet $\{9, 55\}$ har prefiksegenskapen, men $\{9, 5, 59, 55\}$ har ikke, siden 5 er prefiks i 55 og 59. Måten Huffmankoding beholder denne egenskapen blir fort tydelig når vi ser på algoritmen.

Teorem 12.1.1. *Algoritme for Huffman-koding*

1. *Lag frekvenstabell for alle tegn som forekommer i teksten.*
2. *Sett alle frekvensene inn i en heap P*
3. *Mens P har mer enn ett element:*
   - *Ta ut de to minste nodene fra P.*
   - *Slå sammen nodene, og summer frekvensene.*
   - *Legg den nye noden inn i P.*
4. *Vi leser koden til et tegn ved å følge stien fra rotnoden ned til tegnets løvnode. Vi får en '0' når vi går til venstre, og en '1' når vi går til høyre. (Dette fungerer selvfølgelig ved å gjøre motsatt, så lenge man er konsekvent.)*

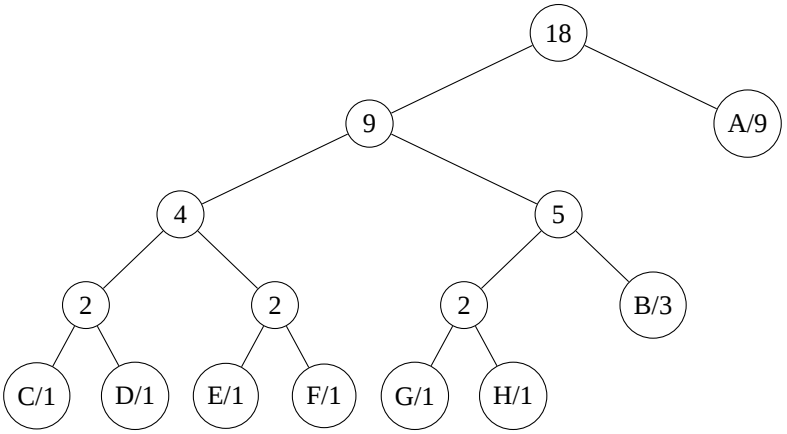
Merk: En tekst kan ha flere gyldige Huffman-trær, og dermed flere gyldige Huffmankodinger.

Algoritmen burde bli klar når vi ser på et eksempel.

Eksempel. Anta at vi har en tekst “BACADAEAFABBAAAGAH”. Vi setter opp en frekvenstabell for bokstavene i koden.

Tegn	Frekvens
A	9
B	3
C	1
D	1
E	1
F	1
G	1
H	1

Vi ser at vi kan parvis slå sammen to og to av nodene som har vekt 1: Vi slår sammen C og D, E og F, og G og H. Nå har (CD), (EF) og (GH) alle vekt 2. Vi slår så sammen B og (CD) til en node (B, CD) med vekt 5, og vi slår sammen (EF) og (GH) til en node (EF, GH) med vekt 4. Vi slår så igjen sammen (B, CD) og (EF, GH) slik at de får en foreldrenode med vekt 9. Til slutt slår vi sammen denne med noden A. Det resulterende treet er:



Hvis vi nå følger regelen fra algoritmen om å skrive ‘0’ når vi går venstre og 1 når vi går høyre får vi kodene:

Tegn	Frekvens	Kode
A	9	1
B	3	011
C	1	0000
D	1	0001
E	1	0010
F	1	0011
G	1	0100
H	1	0101

Vi kan lett sjekke at prefiksegenskapen er intakt, og teksten “BACADAEAFABBA-AAGAH” får koden

```
01110000100011001010011101101111110100101 01
```

som er 42 bits lang. Dette er åpenbart mye kortere enn standard enkoding. Hvis vi for eksempel hadde brukt ASCII-enkoding, der hvert tegn har en 7-bits kode, hadde vi hatt en kode med $7 \cdot 18 = 126$ bits.

♣

13 Tekstsøk

Generelt problem: Vi er interesserte i å finne ut om en streng er en substreng av en annen streng. Vi kaller substringen for en nål i, av lengde m, og stringen for en høystakk h, av lengde n.

13.1 Brute force

Brute force-metoden er som navnet tilsier ikke spesielt gjennomtenkt: Til å begynne med sjekker vi hver karakter i høystakken mot den første i nåla. Hvis vi har likhet sjekker vi det neste tegnet i høystakken mot det neste i nåla. Vi fortsetter slik til vi enten finner hele nåla i høystakken eller til vi har ulikhet, og i så fall fortsetter vi med å sjekke det neste tegnet i høystakken mot det første i nåla.

Analyse

Worst case så får vi mismatch $(n-m)$ ganger og suksess $(n-m)+1$ ganger, Totale sammenligninger er $((n-m)+1) \cdot m$ som gir kjøretid $O(n^2)$.

13.2 Boyer-Moore(-Horspool)

Dette er en rask substringalgoritme. Med Boyer-Moore preprosserer vi nåla før vi begynner å søke. Vi regner ut hvor mange hopp vi kan gjøre for hvert enkelt tegn vi får mismatch på. Dette kaller vi *bad character shift*. Vi beregner også *good character shift*, som er verdier vi kan hoppe på et gitt sted i nåla hvis vi har hatt match tidligere i nåla, men ikke på denne plassen.

Teorem 13.2.1. *Boyer-Moore*

1. *Beregn bad character shift for alle tegn.*
2. *Sammenlign nåla med teksten, starter med tegnet lengst til høyre i nåla.*
3. *Hvis mismatch på* `c`*, flytt nåla fram med den* ***største*** *verdien av* `badCharShift[c]` *og* `goodSuffixShift[c]`*. Hvis match, dytt nåla en til høyre og sjekk neste tegn. Gjenta.*

Merk at når vi snakker om å skippe så er det alltid i forhold til det siste tegnet i nåla. Vær også obs på, som det står i algoritmen, at når vi bruker denne vil vi skippe maks av `badCharShift[c]` og `goodSuffixShift[c]`, der `c` er tegnet vi sammenligner.

Bad character shift

Vi beregner avstand til neste gang nål er på linje med høystakk. Ved en mismatch vil vi dytte nåla framover til en av to ting skjer: Mismatch blir match eller nåla beveger seg forbi tegnet vi fikk mismatch på. I praksis gjør vi dette slik: Verdien til et tegns bad character shift er lengden på nåla minus den siste indeksen som tegnet befinner seg på, minus 1. Altså `value = length - index - 1`.

Eksempel. La nåla være "tooth", og teksten være "trusthardtoothbrushes".

	T	O	O	T	H
index	0	1	2	3	4

Vi konstruerer bad character shift for denne med regelen `value = length - index - 1`.

T =	$5-0-1$	$=4$	
O =	$5-1-1$	$=3$	
O =	$5-1-2$	$=2$	Erstatter her forrige verdi av O med ny verdi til O.
T =	$5-3-1$	$=1$	Erstatter her forrige verdi av T med ny verdi til T.
H =	5		Verdien skal ikke være mindre enn 1. Får da verdi lik lengden.

Vi sitter igjen med dette som bad character shift-tabell:

Bokstav	T	O	H	$*$
Verdi	1	2	5	5

Vi bruker nå bad character shift på eksemplet vårt:

	T	R	U	S	T	H	A	R	D	T	O	O	T	H	B	R	U	S	H	E	S
1.	T	O	O	T	H																
2.		T	O	O	T	H															
3.							T	O	O	T	H										
4.									T	O	O	T	H								
5.										T	O	O	T	H							

I steg **1.** får vi mismatch på H, fordi den ikke er det samme som T. Da må vi slå opp i tabellen på den bokstaven som ***vi møter i teksten***, og hoppe fram tilsvarende antall steg. I første tilfellet skal vi hoppe 1 plass fram (for 1 er verdien til T).

Sjekker på nytt i steg **2.**, her får vi match på T og H, men mismatch på O. Da hopper vi H-plasser frem.

I steg **3.** får vi mismatch på H mot O, og må hoppe O-plasser frem—altså 2. I steg **4.** er det mismatch mellom H og T, vi hopper T-plasser frem—altså 1. Og nå har vi funnet nåla vår!

♣

Good suffix shift

Anta at vi har en substring t av nåla (altså lengst til høyre i nåla) som allerede er matchet. Vi får så mismatch på neste tegn. Vi kan nå være smarte og finne ut, og vi må tenke på to tilfeller: t forekommer et annet sted i nåla. Da kan vi ikke skippe lenger enn til neste gang det skjer. Eller, en del av t forekommer i starten av nåla. Da må vi skippe til vi er på linje med dette.

Eksempel. Vi ser på nåla "TTCTATTCTT".

1. Sjekker først *ikke*-`T`, det er to shift før vi finner dette.

```
TCCTATTCTT
  TCCTATTCTT
```

2. Så sjekker vi *ikke*-`TT`. Dette finner vi etter ett shift.

```
TCCTATTCTT
 TCCTATTCTT
```

3. For å finne *ikke*-`CTT` må vi flytte 3 steg; til `ATT`.

```
TCCTATTCTT
   TCCTATTCTT
```

4. Så kommer vi til *ikke*-`TCTT`. Denne eksisterer ikke, MEN om nålen har lik suffix som prefiks (her `T` som start og slutt), så flytter vi bare fram til prefiksen. Da må vi flytte nålen 9 hakk, selv om lengden er 10.

```
TCCTATTCTT
          TCCTATTCTT
```

Alle de neste shiftene vil også være dette, fordi de får ingen andre treff i nålen. Vi får da følgende good suffix table:

index	mismatch	shift
0	T	2
1	TT	1
2	CTT	3
3	TCTT	9
4	TTCTT	9
5	ATTCTT	9
6	TATTCTT	9
7	CTATTCTT	9
8	CCTATTCTT	9
9	TCCTATTCTT	9

♣

14 Sortering

Før vi kan begynne å se på noen spesielle sorteringsalgoritmer må vi formalisere hva vi mener med sortering. Vi definerer sortering slik:

Definisjon 14.0.1. Anta at `a[]` er en liste av sammenlignbare elementer. Vi sier at `a'[]` er den tilhørende sorterte lista hvis følgende kriterier er oppfylt:

i `a'[i]` $<$ `a'[i+1]` for alle `i`$= 0, 1, \ldots, n-1$

ii Alle elementene i `a[]` er med i `a'[]`

Det andre kriteriet kan virke litt snodig, men uten det ville sortering vært veldig enkelt. Vi kunne i så fall bare generert en ny liste med elementer i sortert rekkefølge, og det første kriteriet ville vært oppfylt. Vi trenger derfor bevaringskriteriet.

Med "sammenlignbare" mener vi at det finnes en måte å entydig bestemme om et element er større enn, mindre enn eller lik et annet element. Hvis vi skal sortere tall er jobben enkel: vi sammenligner numerisk verdi. Hvis vi skal sammenligne to tekststrenger er det ikke like opplagt hvordan vi skal gjøre det. Skal vi sortere alfabetisk? Etter lengde på ordet? I et sånt tilfelle er det opp til oss å velge et fornuftig sammenligningskriterie. Det er vilkårlig hvordan vi sammenligner to elementer, så lenge vi gjør det likt gjennom hele sorteringen.

Paradigmer og begreper

Klasser av sorteringsalgoritmer

Avhengig av hvordan de virker, deler vi sorteringsalgoritmer inn i to klasser: Sammenligningsbaserte og Verdibaserte. Førstnevnte baserer seg på å sammenligne to og to verdier i arrayen, mens de verdibaserte finner den riktige plasseringen til hvert element kun ut ifra verdien på elementet.

Stabil Sortering

Noen ganger ønsker vi å sortere etter flere kriterier. Et eksempel kan være epost, der vi f.eks. vil sortere både etter avsender og alfabetisk, slik at vi får at alle som er fra samme avsender står alfabetisk i forhold til hverandre.

Definisjon 14.0.2. En algoritme er **stabil** dersom innbyrdes rekkefølge beholdes etter sortering. Dvs at dersom `a[i]` $=$ `a[j]`, og `a[i]` kommer før `a[j]` i `a[]`, så skal `a[i]` komme før `a[j]` i den sorterte lista `a'[]`.

Følgende algoritmer (altså ikke alle) gjennomgått i kurset oppfyller kravet om stabil sortering:

- Innstikksortering
- Radixsort
- Flettesortering kan være stabil, avhengig av hvordan flettemetoden er implementert. I vårt (og bokas) eksempel er det en stabil sortering.

Nedre grense for enkle sorteringsalgoritmer

Teorem 14.0.3. *(7.1 i læreboka): Det gjennomsnittlige antall inversjoner i en array med n unike elementer er $\frac{n(n-1)}{4}$*

Beviset bruker at summen av en aritmetisk rekke er $n(n-1)/2$:

Bevis. La L være en array og L_r den reverserte. Et hvilket som helst ordnet par i L vil være en inversjon i enten L eller L_r. Det totale antallet slike par i L og L_r vil være $\frac{n(n-1)}{2}$, altså har en gjennomsnittlig liste halvparten så mange, altså $\frac{n(n-1)}{4}$ □

Vi har da et korollar til teorem 14.0.3 som gir oss gjennomsnittlig tid for algoritmer som bytter naboelementer:

Korollar 14.0.4. *(7.2 i læreboka): Enhver sorteringsalgoritme som sorterer ved å bytte to naboelementer krever i gjennomsnitt $O(n^2)$ tid.*

Bevis. Fra teorem 14.0.3 har vi at gjennomsnittlig antall inversjoner er $n(n-1)/4$. Hvert plassbytte av to naboer inversjon fjerner én inversjon, dermed tar algoritmer som baserer seg på plassbytte av naboer gjennomsnittlig $O(n^2)$ tid. □

14.1 Boblesortering

Boblesortering (engelsk: **bubble/ (sinking) sort**) er en veldig ineffektiv sorteringsalgoritme, og brukes derfor lite i den virkelige verden. Ideen bak er derimot forholdsvis enkel: Vi parvis sammenligner naboer i lista, og bytter om plassene deres dersom de står på feil plass.

Wikipedias utgave

Wikipedias versjon av boblesortering er forskjellig fra den som er gjennomgått i forelesning. Her flytter vi oss hele tiden oppover i lista, og begynner på nytt når vi har kommet til slutten. Dette gjentas helt til lista er ferdig sortert.

```
1  void bubbleSort(int[] A){
2      boolean swapped = true;
3      while (swapped){
4          swapped = false;
5          for (int i=1; i<A.length; i++){
6              // pair out of order
7              if (A[i-1] > A[i]){
8                  swap(A[i-1],A[i]);
9                  swapped = true;
10             }
11         }
12     }
13 }
```

Utgaven presentert i forelesning

I forelesningen ble en noe alternativ utgave av boblesortering presentert, og det kan være lurt å forholde seg til denne, siden det er denne som er pensum. Denne utgaven ligner veldig på innstikksortering, fordi den "bobler" et element som står på feil plass nedover i lista helt til den står på riktig plass.

```
1  void optBubbleSort(int[] A){
2      int i = 0;
3
4      while (i < A.length){
5          if (A[i] > A[i+1]){
6              swap(A[i], A[i+1]);
7              if (i > 0){
8                  i -= 1;
9              }
10         }
11         else {
12             i += 1;
13         }
14     }
15 }
```

Kompleksitet

Merk at selv om utgaven av boblesortering presentert i forelesning i mange tilfeller kan være betydelig raskere, vil disse to algoritmene ha samme kompleksitet.

Worst Case: $O(n^2)$ Dersom vi ser for oss at vi skal sortere en liste som er den reverserte av den sorterte, må vi gå gjennom lista n ganger. Dette fordi vi plasserer det største elementet i første gjennomgang, nest største i andre gjennomgang osv. Det minste elementet behøver vi ikke å flytte, men til slutt vil vi gå gjennom lista en gang for å sjekke at den er sortert. Altså gjør vi $n(n-1)$ sammenligninger. $O(n(n-1)) = O(n^2)$

Best Case: $O(n)$ En allerede sortert liste: Vi sammenligner alle nabo-par i arrayen, og siden vi ikke foretar noen bytter, vil algoritmen terminere etter å ha gjort $(n-1)$ sammenligninger. Altså $O(n-1) = O(n)$

Average Case: $O(n^2)$ fra korollar 14.0.4

14.2 Innstikksortering

Innstikksortering (engelsk: **insertion sort**) er også en veldig enkel sorteringsalgoritme, og den er faktisk best for $n < 50$. Ideen her er at elementene vi hittil har besøkt, er sorterte, isolert sett. Deretter tar vi neste element, "skyver" de foregående elementene oppover, helt til det nye elementet står på riktig plass. Altså kan denne metoden minne mye om den optimaliserte boblesorteringen.

```
1  void insertionSort(int[] a ){
2      int i;
3      int t;
4      int max = a.length - 1;
5
6      for (int k=0; k<max; k++){
7          // Invariant: a[0..k] sortert
8          // nå sortere a[k+1] inn på riktig plass
9          if (a[k] > a[k+1]){
10             t = a[k+1];
11             i = k;
12
13             do { //gå bakover, skyv de andre
14                  //og finn riktig plass for 't'
15                 a[i+1] = a[i];
16                 i--;
17             } while (i>=0 && a[i]>t) {
18                 a[i+1] = t;
19             }
20         }
21     }
22 }
```

Merk at if-testen ikke trengs for at algoritmen skal fungere. Faktisk lønner det seg å ikke ha med denne testen. Siden if-tester er relativt dyre operasjoner, lønner det seg å "plukke opp" et element, for deretter å sette den ned igjen på samme plass, enn å sjekke om det står på feil plass.

Kompleksitet Vi ser at gjennomsnittlig er innstikksortering $O(n^2)$, som gjør algoritmen uegnet til store lister. Når lista er kort[1] derimot, er innstikksortering en ganske god algoritme - til og med bedre enn Quicksort. Gode quicksortimplementasjoner bruker derfor innstikksortering hvis lista blir for kort.

Worst case: $O(n^2)$ Et enkelt eksempel på et worst case tilfelle er hvis lista er sortert i motsatt rekkefølge. Vi må derfor flytte alle elementene i snitt $n/2$ ganger.

Best case: $O(n)$ Hvis lista vi sender inn allerede er sortert vil den ytterste for-løkka gå gjennom lista én gang og så avslutte.

Avarage case: $O(n^2)$ fra korollar 14.0.4

[1] Generell tommelfingerregel: mindre enn 50 elementer

14.3 Flettesortering

Flettesortering (engelsk: **mergesort**) er et ekempel på en splitt-og-hersk-algoritme, og er et eksempel på elegant rekursjon. Anta at vi har en liste L som er n elementer lang. Vi deler først L opp i n biter slik at vi har n lister på ett element. Deretter fletter vi listene sammen til vi bare har én liste.

Spørsmålet blir da hva vi mener med å *"flette listene sammen"*. Vi beskriver samme metode som boka: Anta at vi skal flette to lister A og B. A er m elementer lang, og B er o elementer. Vi har en resultatliste C som har $m + o$ elementer. Vi har tre tellere: i, j og k som går over henholdsvis A, B og C. Vi starter med å sette $i = j = k = 0$ og sammenligner A_i med B_j. Den minste av dem setter vi inn i C_k. Hentet vi elementet fra A øker vi i med 1, hentet vi elementet fra B øker vi j med 1. Uansett øker vi k med 1. Deretter sammenligner vi A_i med B_j, setter minste inn i C_k, og øker riktig teller. Slik fortsetter vi så lenge $k < m + o$. Se seksjon 7.6 (side 302) i boka for et nais eksempel med figurer.

Kompleksitet Vi setter opp en rekkurant følge for kjøretiden T. Fra definisjonen av flettesortering fremgår det at

$$T(1) = 1$$
$$T(n) = 2\,T\left(\frac{n}{2}\right) + n$$

Vi ser at for $n > 1$ må vi utføre $\log_2 n$ rekursjoner (siden vi halverer n for hver gang) før vi kommer til et trivielt problem, altså base case $T(1) = 1$. For hver rekursjon har vi lineær tid (selve flettinga). Dermed har flettesortering $O(n \log n)$ tid.

14.4 Heapsort

I heapsort utnytter vi egenskapene til en heap. Her bruker vi en max-heap, som vi setter alle elementene inn i. Da vet vi at rota er det største elementet per detfinisjon av heap. Dersom vi så tar ut et og et element av heapen, og plasserer på den bakerste ikke-opptatte plassen i resultat-arrayen, vil vi få arrayen ut i sortert form.

Kompleksitet Fra teorem 6.1.2 har vi at innsetting og sletting i en heap tar, i verste fall, $O(\log_2 n)$ tid. I beste fall har vi at innsetting tar $O(1)$ tid, og at sletting tar $O(\log_2 n)$ tid. Siden lista er n elementer lang må vi gjøre n innsettinger og n slettinger. Det gir at total kjøretid er

$$\text{I beste fall:} \quad T(n) = n + n \log_2 n = O(n \log_2 n)$$
$$\text{I verste fall:} \quad T(n) = n \log_2 n + n \log_2 n = O(n \log_2 n)$$

14.5 Tresortering

I likhet med Heapsort utnytter vi her egenskapene til en datastruktur, nemlig binære søketrær. I et slikt tre vet vi at venstre barn til en node er mindre enn noden selv, og at høyre barn er større eller lik noden. Deretter traverserer vi treet i infix-rekkefølge. På den måten vil vi altså gå gjennom treet i stigende rekkefølge, og vi har dermed sortert arrayen.

Kompleksitet Fra teorem 4.1.1 har vi at innsetting i et binært søketre tar $O(\log_2 n)$ beste tid, og $O(n)$ verste tid. En traversering av treet tar opplagt $O(n)$ tid, siden vi går gjennom hver node én gang. Tilsammen har vi da at kjøretiden for algoritmen er

$$\begin{aligned} \text{I beste fall / gjennomsnittlig:} \quad & T(n) = O(n \log_2 n) \\ \text{I verste fall:} \quad & T(n) = O(n \cdot n) = O(n^2) \end{aligned}$$

14.6 Quicksort

Som navnet antyder, er dette en ganske rask algoritme. Ideen er at vi velger et (vilkårlig) element i arrayen (pivotelement), og ordner resten av elementene slik at de som er mindre pivotelementet står på venstre side, og de som er større på venstre. Deretter gjentar vi denne prosessen for disse to subarrayene på hver side av pivotelementet.

```
1  void quickSort(int[] a, int l, int r){
2      int i = l;
3      int j = r;
4      int t;
5      int part = a[(l+r)/2];
6
7      while (i <= j){
8          while (a[i] < part){
9              i++; //hopp over små
10         }
11         while (part < a[j]){
12             j--; //hopp over store
13         }
14
15         if (i <= j){
16             t = a[j];
17             a[j] = a[i];
18             a[i] = t;
19
20             i++;
21             j--;
22         }
23     }
24     if (l < j){
25         quicksort(a,l,j);
26     }
27     if (l < r){
28         quicksort(a,i,r);
29     }
30 }
```

Som man kan se i koden ovenfor er det litt problematisk å foreta selve flyttingen av elementene i forhold til pivoten når vi jobber med arrayer, siden vi ikke kan dele opp og sette sammen arrayer. En detaljert beskrivning (og illustrering) av tankegangen finnes i boka i seksjon 7.7.2 (side 312). En noe forenklet utgave følger her:

1. Bytt plass på pivoten og det siste elementet, slik at vi unngår at den kommer i veien for oss.
2. La i og j starte på henholdsvis første og **nest siste** plass i arrayen.
3. La i bevege seg oppover, og j nedover, helt til i står på et element som er større enn pivotelementet, og j står på et element som er mindre.
4. Bytt plass på elementene som i og j peker på.
5. Gjenta 3 og 4 helt til i og j har passert hverandre. Nå peker j på et element mindre enn pivot, og i på et som er større. Dersom vi nå bytter plass på verdien på i og pivoten.

Eksempel.

3	1	4	$\underline{5}$	9	2	6	8	7	Der elementet med linje under er pivoten.
3	1	2	4	$\underline{5}$	9	6	8	7	Etter at vi har ordnet elementene i forhold til pivoten

Nå står pivoten på riktig plass. Vi gjentar prosessen på subarrayen som består av elementene på høyre side, og på elementene på venstre side av pivoten.

♣

Om å velge pivot Hvilket element vi velger som pivotelement kan ha veldig mye å si for kjøretiden til algoritmen. Dersom vi får en array som allerede er sortert, og vi velger det første elementet (altså det minste) som pivotelement, vil vi bruke kvadratisk tid på å gjøre egentlig ingenting!

Det beste valget av pivot, ville vært medianen. Da vil størrelsen på subarrayene bli det samme (med en forskjell på 1 hvis vi har partall antall elementer). Dessverre vil det å regne ut medianen til en stor array ta alt for lang tid. Boka foreslår en mulig løsning ved å trekke tilfeldige tall, og deretter bruke medianen til disse som pivot. Det tilfeldige elementet viser seg imidlertid å hjelpe lite. Derfor blir det til slutt anbefalt å bruke medianen til det første, siste og midterste elementet i arrayen som pivot.

Kompleksitet Det er flere måter å analysere kompleksiteten til Quicksort. En metode er ved å bruke diskret sannsynlighetsteori (god innføring i dette på Wikipedia). Her skal vi bruke en annen metode. Vi setter opp det som kalles en *rekurrent følge* for kjøretiden og analyserer den. For hver rekursjon må vi gjennom lista og dele den i to kategorier, vi har altså lineær tid $O(n)$ for hvert steg. Analysen går derfor i hovedsak ut på å finne ut hvor mange rekursjoner som må til.

Worst case: $O(n^2)$ Hvis vi i hver rekursjon velger det største elementet som pivot vil vi i den i-te rekursjonen gjøre $n - i$ flyttinger. Setter vi opp et uttrykk (rekurrent følge) for kjøretiden får vi:

$$T(n) \;=\; O(n) + T(n-1) \;=\; O(n) + O(n-1) + T(n-2) \;=\; \ldots$$

Vi ser at vi må gjøre n rekursjoner før vi når et problem der løsninga er triviell, og vi kan nøste sammen den totale løsninga. Følgelig er worst case for quicksort $O(n^2)$.

Best case: $O(n \log n)$ I et perfekt tilfelle vil vi velge medianen som pivot i hver eneste rekursjon. Vi vil dermed halvere problemet for hver gang. Setter vi opp et uttrykk for kjøretiden får vi:

$$T(n) \;=\; O(n) + 2\,T\left(\frac{n}{2}\right) \;=\; \ldots$$

Vi ser at vi halverer argumentet til T hver gang. Vi må derfor gjøre $\log n$ rekursjoner før vi når et trivielt problem. Følgelig er best case for quicksort $O(n \log n)$

Avarage case: $O(n \log n)$ Vi har fra forrige avsnitt at hvis vi velger medianen som pivot vil vi ha $O(n \log n)$ tid. Vi kjenner ikke medianen, men estimerer den med medianen av et tilfeldig utvalg elementer. Dette er en forventningsrett estimator for median. Pivoten vi velger vil derfor være en god tilnærming til den optimale pivoten. Det at estimatoren for medianen er forventningsrett gir oss at i det lange løp vil gjennomsnittet gå mot tilfellet der vi velger medianen som pivot, altså $O(n \log n)$

Nå har det seg slik at som diskuterte tidligere velger vi sjeldent tre *tilfeldige* elementer i lista, men heller første, midterste og siste element. Likevel, hvis vi antar at innholdet i lista er tilfeldig (uniformt) fordelt vil *innholdet* i de tre elementene være tilfeldig, selv om *indexen* er deterministisk.

14.7 Radixsort

Radix sort sorterer en array på et siffer av gangen. Metoden forekommer i to varianter, en fra høyre til venstre, og en som gjør motsatt. Generelt for begge algoritmene har vi

1. Finn max verdi i arrayen. (Største siffer i alle tallene)
2. Tell opp hvor mange elementer det er av hver verdi av det sifferet vi ser på. (hvor mange 0-er, 1-ere osv)
3. Ved å addere disse antallene, finner vi ut hvor i arrayen tallene skal stå ved å addere verdiene i arrayen vi lagde i 2, men starte på 0.

En spesiell ting ved Radix sort er at den aldri gjør sammenligninger mellom to tall.

Right-Radix (RR)

Den "vanlige" formen for Radix-sort, og er iterativ. Den begynner med det minst gjeldende sifferet (det bakerste). Etter "fellesstegene" blir den siste arrayen vi lagde brukt til å slå opp hvor vi skal plassere elementet. Vi går gjennom arrayen fra venstre til høyre, og setter inn elementer i den nye arrayen. At vi gjør det fra høyre til venstre er viktig, siden vi ønsker en stabil sortering. Etter at vi har plassert et element på sin plass i den nye "sorterte" arrayen, bør vi øke verdien vi akkurat brukte i telle-arrayen med 1 så vi lett kan finne ut hvor neste tall med samme siffer på gjeldende plass. Dette gjentar vi helt til vi har vært gjennom maks antall gjeldende siffer.

Eksempel.

$$170 \quad 45 \quad 75 \quad 90 \quad 802 \quad 2 \quad 24 \quad 66$$

Teller deretter opp antall tall som slutter på 0, 1 osv. resulterer i

$$2 \quad 0 \quad 2 \quad 0 \quad 1 \quad 2 \quad 1 \quad 0 \quad 0 \quad 0$$

Altså har vi to tall som slutter på 0, to som slutter på 2 osv. For å finne den nye plasseringen til elementene, summerer vi opp plassene i denne arrayen, og får

$$0 \quad 2 \quad 2 \quad 4 \quad 4 \quad 5 \quad 7 \quad 8 \quad 8 \quad 8$$

Da vet vi f.eks. at tall som slutter på 0 skal settes inn på indeks 0. Siden vi har to slike, kan vi ikke sette inn de som slutter på 2 noe lenger ut i arrayen enn indeks 2. Når vi setter inn, og beholder innbyrdes ordning, får vi.

$$170 \quad 90 \quad 802 \quad 2 \quad 24 \quad 45 \quad 75 \quad 66$$

Deretter gjentar vi prosesen men denne arrayen, og sorterter på siffer nr 2 fra høyre. Dette gjentar vi til vi har vært gjennom så mange gjeldende siffer som det største tallet i arrayen har. I dette tilfellet 3 (siden 802 har tre gjeldende siffer.) Merk at f.eks. 2 er å betrakte som 002 videre i sorteringen.

♣

Kompleksitet I radix sort gjør vi kun lineære operasjoner: Først går vi gjennom lista for å finne den største verdien, deretter går vi gjennom lista for å telle, så går vi gjennom for å plassere elementene på sine nye plasser. De siste to punktene gjøres like mange ganger som vi har gjeldende sifre. La oss kalle denne verdien for k. Da får vi $O(n + 2kn)$, som er lik $O(kn)$ siden $k \geq 1$. Alternativt kan vi betrakte k som en konstant, og vi får at radix sort er $O(n)$

14.8 Parallell Sortering

Amdahls lov

Amdahls lov gir oss en slags øvre grense for hvor mye tid vi kan spare på å parallellisere en algoritme.

Teorem 14.8.1. ***Amdahls lov:*** *Anta at du har en algoritme der en andel p% må kjøres sekvensielt, men at resten kan gjøres i parallell. Med k lik antall prosessorer, får vi at maksimal speedup er*

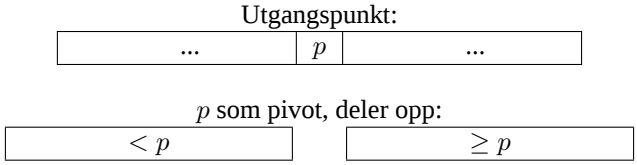
$$S = \frac{\textit{Sekvensiell tid}}{\textit{Parallell tid}} \leq \frac{100}{p + \frac{100-p}{k}}$$

Denne loven har imidlertid møtt motstand i Gustafson, som hevder at andelen av programmet som er sekvensielt blir mindre etterhvert som problemet blir større. Altså er ikke Amdahls lov fast.

Parallell Quicksort

Siden Quicksort deler lista opp i to deler der alle elementene i den ene lista er mindre enn alle elementene i den andre lista er den velegnet til å parallellisere. Vi må gjøre de første rekursjonene sekvensielt og deler opp lista i n biter (merk at n på være på formen $2^k, k \in \mathbb{N}$). Deretter starter vi n tråder med hver sin del å sortere. For å n tråder må vi gjøre $\log_2 n$ nivåer med rekursjon sekvensielt først. Når alle trådene er ferdig har vi n sortere biter, der vi vet at alle elementene i tråd 1 er mindre enn elementene i tråd 2, som er mindre enn elementene i tråd 3, osv. Da kan vi sette sammen bitene sekvensielt til slutt og få en sortert liste.

Figur 14.1: Illustrasjon av parallell quicksort med to tråder



Starter egne tråder som sorterer disse to bitene, setter sammen til slutt.

A Big-O Cheat sheet

Datastrukturer

Type	Sett inn		Søke		Slette		Access	
	Avarage	Worst	Avg.	Worst	Avg.	Worst	Avg.	Worst
Array	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$
Lenket liste	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Kø/stack	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$
Hashmap (hashtabell)	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$		
Binære søketrær	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$
Rød-svarte trær	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
B-trær	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$

Grafalgoritmer

Algoritme	Kjøretid (worst)	Plass
Dybde-først-søk	$O(\|E\|)$	$O(\|V\|)$
Dijkstras algoritme	$O(\|V\|^2)$	$O(\|V\|)$
Prims algoritme	$O(\|V\|^2)$	$O(\|V\|)$
Kruskals algoritme	$O(\|E\| \cdot \log \|V\|)$	$O(\|V\|)$
Floyds algoritme	$O(\|V\|^3)$	$O(\|V\|^2)$

Sorteringsalgoritmer

Sorteringsalgoritme	Tid			Plass
	Best	Avarage	Worst	
Boblesortering	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Innstikksortering	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Flettesortering	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Heapsort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Tresortering	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$
Quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Radixsort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$