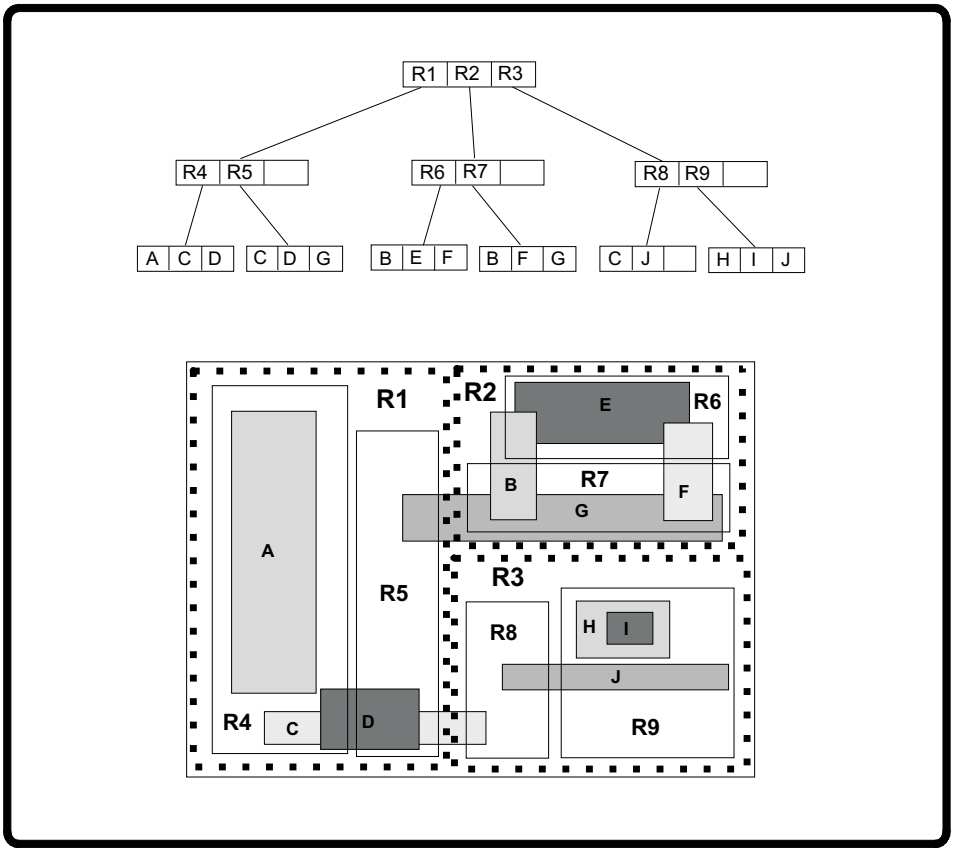
DATAMODELLER, ALGORITMER OG DATASTRUKTURER FOR GEOGRAFISK INFORMASJONSBEHANDLING



JAN TERJE BJØRKE

2005

Digital kartografi

Jan Terje Bjørke

4. oktober 2005

Forord

Den første utgaven av denne boken er fra 1992. I den neste utgaven fra 1997 er topologikapitlet utvidet med en beskrivelse av 9-snittmodellen og at et nytt kapittel om flytende mengder er innført. I utgaven fra 2005 er enkelte skrivefeil rettet opp. I forordet til utgaven av 1992 heter det:

> *Om framtidens forhandlere av digitale geodata og utviklere av geografisk informasjonsteknologi vil rektruttere kandidater fra fagmiljøer som i dag kaller seg kartografiske, vil avhenge av vår egen dyktighet og evne til å tilpasse oss de krav den teknologiske utvikling stiller til kartbransjen.*

Geografiske informasjonssystemer (GIS) er i dag en vidtfavnende teknologi som griper inn i en rekke fagområder, både når det gjelder anvendelser og når det gjelder teorigrunnlaget. Boken ble opprinnelig skrevet med tanke på undervisningen i digital kartografi for sivilingeniører ved NTH, nå NTNU, men inngår i dag som pensumlitteratur for geomatikkstudenter ved UMB. Stoffvalget går dels lengere enn det som er pensum i de grunnleggende kurs i digital kartografi. Tanken er imidlertid at boken skal kunne leses på flere kunnskapsnivåer og dels kunne benyttes som en referansebok for praktiserende sivilingeniører.

Boken, som i hovedsak er metodeorientert, behandler prinsipper for konstruksjon av romlige datastrukturer og viser hvordan disse kan benyttes som fundamenter i algoritmer for romlige søk og analyser. Videre gis en innføring i topologi og det blir vist hvordan dette begrepsapparatet kan benyttes ved utforming av romlige datamodeller og geodatabaser. Boken drøfter også modeller og metoder for automatisert kartografisk generalisering. Kartografi blir her oppfattet i vid forstand slik at generalisering ikke bare blir satt inn i en visuell sammenheng, men også i en digital sammenheng (modellgeneralisering).

Siden dette er en foreløpig utgave, mangler deler eller hele kapitler. Undertegnede tar gjerne imot reaksjoner på boken (emnevalg, skrivefeil, logiske feil, konstruktive forslag til forbedringer o.l.).

UMB 4. oktober 2005

Jan Terje Bjørke[1]

[1]Jan Terje Bjørke er professor i geografisk informasjonsvitenskap ved UMB, 1432 Ås og forsker ved Forsvarets forskningsinstitutt.
epost: jtb@ffi.no

Innhold

1 DATAMASKINER OG DATAUTSTYR **1**
1.1 Tallsystemer 1

2 GENERELLE DATASTRUKTURER OG ALGORITMER **3**
2.1 Datatyper og datastrukturer 3
2.1.1 Celle 4
2.1.2 Array 4
2.1.3 Record 5
2.1.4 Klasser 5
2.1.5 Pekere og markører 6
2.1.6 Lister 7
2.1.7 Rekursive prosedyrer 8
2.1.8 Tidsforbruket til et program 8
2.2 Trær 10
2.2.1 Datastrukturer for trær 11
2.2.2 Traversering av trær 13
2.3 Prioritetskø 14
2.4 2-3Trær 20
2.5 B-trær 21
2.6 Nettverk 26
2.6.1 Datastrukturer for nettverk 27
2.6.2 Korteste vei 27
2.7 Hash-metoder 32
2.7.1 Åpen hashing 32
2.7.2 Lukket hashing 33
2.7.3 Lineær hashing 34
2.7.4 Extendible hashing 35
2.7.5 Cellemetoden 37
2.7.6 Mortonindekser og bitfletting 38
2.8 Sortering 39
2.8.1 Boblesortering 39
2.8.2 Quicksort 40
2.8.3 Flette-sortering 44

2.8.4 Ordning av romlige data 45

3 TOPOLOGI 51
3.1 Noen begreper fra mengdelæren 51
3.2 Topologisk rom og kontinuerlig avbildning 54
3.3 Metrisk rom og metrisk topologi 55
3.4 Generell topologi 59
3.4.1 Generalisering av begrepet topologi 59
3.4.2 Den lukkede, den indre og omrisset til en mengde 60
3.4.3 Sammenhengende rom 63
3.5 Topologiens betydning for GIS 65
3.5.1 Eksempler på anvendelse av topologisk terminologi i GIS . . . 66
3.5.2 Betydningen av topologisk informasjon i GIS 67
3.6 Topologiske relasjoner 68
3.6.1 Metode for å beskrive topologiske relasjoner 68
3.6.2 Topologiske relasjoner mellom romlige regioner 71
3.6.3 Topologiske relasjoner i n-dimensjonale rom 74
3.6.4 9-snittmodellen 75
3.6.5 Relasjoner mellom objekter som har uskarpe avgrensninger . . 76

4 FLYTENDE MENGDER 79
4.1 Elementer fra teorien om flytende mengder 80
4.1.1 Skarpe mengder 81
4.1.2 Flytende mengder og operasjoner på disse 81
4.1.3 Flytende relasjoner 83
4.1.4 Flytende tall 84
4.1.5 Mulighetsteori 86
4.2 Eksempler 87
4.2.1 Turist-GIS 87
4.2.2 Topologiske relasjoner 91

5 ROMLIGE DATAMODELLER 93
5.1 Topologiske og geometriske modeller 93
5.2 Tradisjonelle romlige modeller i GIS 94
5.2.1 Vektormodeller 94
5.2.2 Rastermodeller 95
5.3 Klassifikasjon av geometriske modeller 96
5.3.1 Topologisk dimensjon 96
5.3.2 Modellbegrepet 97
5.3.3 $\overline{A}$ og ∂A som klassifikasjonskriterium 97
5.3.4 Kodimensjon som klassifikasjonskriterium 98
5.3.5 Dekomponering som klassifikasjonskriterium 98
5.3.6 Klassifikasjonstre 99

6 ROMLIGE DATASTRUKTURER 103
6.1 Operasjoner på romlige data . . . 104
6.2 Kvadtrær . . . 105
6.3 Regulær firedeling av planet . . . 106
6.3.1 Plassbehov . . . 107
6.3.2 Posisjonskode . . . 109
6.3.3 Lineære kvadtre . . . 110
6.3.4 Teknikker for å finne naboer . . . 111
6.3.5 PR-kvadtre . . . 114
6.3.6 MX-kvadtre . . . 122
6.3.7 PM-kvadtrær . . . 124
6.3.8 Regionkvadtre . . . 130
6.4 Irregulær firedeling av planet . . . 132
6.4.1 Punktkvadtre . . . 132
6.5 Binære trær . . . 139
6.5.1 KD-trær . . . 139
6.5.2 Stripetre . . . 142
6.6 Flate datastrukturer . . . 146
6.6.1 EXCELL . . . 146
6.6.2 Gridfile . . . 149
6.6.3 Base85 . . . 152
6.7 Ojektorienterte indekserings-metoder . . . 155
6.7.1 MX-CIFkvadtre . . . 156
6.7.2 R-trær . . . 157
6.8 Konklusjon . . . 161

7 GENERALISERING 165
7.1 Generaliseringsmodeller . . . 165
7.2 Finne karakteristiske punkter til en linje . . . 165
7.2.1 DouglasPeucker-algoritmen . . . 166
7.2.2 Langs algoritme . . . 169
7.3 Diverse enkle silere . . . 172
7.3.1 N-punktsiler . . . 172
7.3.2 Avstandssiler . . . 172
7.3.3 Tidsintervallsiler . . . 172
7.4 Formendring av en linje . . . 172
7.4.1 Middeltall etter vekt . . . 172
7.4.2 Minste kvadraters tilpasning . . . 174
7.4.3 Epsilon-filtrering . . . 174
7.5 Legge en glatt linje gjennom en punktfølge . . . 174
7.5.1 Akimas metode . . . 174
7.5.2 Bezier-kurver . . . 174
7.5.3 B-spline . . . 174

7.5.4 Kubisk spline 174
7.6 Forenkling av nettverk 174
7.7 Rastermetoder 174
7.7.1 Svell og krymp 174
7.7.2 Filtermetoder 176
7.7.3 Lineær strekk 176
7.8 Datakomprimering 176
7.8.1 Parametriserte kurver 176
7.8.2 Fraktaler 176
7.9 Generalisering av visse objekttyper 176
7.9.1 Hus 176
7.10 Prinsipale komponenter 177

Kapittel 1

DATAMASKINER OG DATAUTSTYR

Dette kapitlet vil etter hvert bli utvidet til en sammenfattende oversikt over datamaskiners virkemåte og konstruksjon, lagringsmedier, grafiske skjermer, digitaliseringsutstyr, tegnemaskiner, skrivere etc.. Foreløpig er kapitlet lavt prioritert slik at andre deler av boken først vil bli fullført.

1.1 Tallsystemer

Ulike systemer for telling har vært i bruk hos forskjellige folk. Mest utbredt er *titallsystemet* eller det dekadiske tallsystem, som naturlig svarer til telling på fingrene. I titallsystemet har vi de ti talltegnene $0, 1, 2, \cdots, 8, 9$. Tallverdier blir representert som en kombinasjon av talltegn. Titallsystemet er et *posisjonssystem* i det posisjonen til talltegnene angir hvilken høyere enhet de representerer. For eksempel er tegnkombinasjonen 9365 i titallsystemet det samme som:

$$9 \cdot 10^3 + 3 \cdot 10^2 + 6 \cdot 10^1 + 5 \cdot 10^0$$

Titallsystemet har eksistert så langt som tilbake hos egypterne ca. 3400 f.Kr.

Datamaskiner benytter det *binære* tallsystem. I det binære tallsystem har vi bare to talltegn $0, 1$. En tallverdi må derfor representeres ved en kombinasjon av 0-ere og 1-ere. For eksempel er det binære tallet 1011 i titallsystemet:

$$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 11_{10}$$

Ved summasjon i det binære tallsystem må vi huske at 1+1=10. For eksempel har vi at 1011+1011=10110.

I det *oktale* tallsystem har vi åtte talltegn og i det *heksadesimale tallsystem* har vi 16 talltegn. De 16 talltegnene i det heksadesimale system er:

$$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F$$

For eksempel er tallkombinasjonen 4F det samme som:

$$4 \cdot 16^1 + F \cdot 16^0 = 64_{10} + 15_{10} = 79_{10} = 1001111_2$$

Det heksadesimale tallsystem står i et spesielt gunstig forhold til det binære tallsystem, fordi vi kan dele en binær tegnkombinasjon i grupper á fire biter og så erstatte hver slik bitgruppe med det tilsvarende heksadesimale talltegn i intervallet $[0, F]$. For eksempel har vi at

$$\mid 01 \mid 1101 \mid 1111 \mid 0001 \mid_2 = \mid 1 \mid D \mid F \mid 1 \mid_{16} = 7665_{10}$$

Tilsvarende gunstige forhold har vi også for firetallsystemet og det oktale tallsystem. Overgang fra det binære tallsystem til de to nevnte tallsystemer, skjer ved å dele bitstrengen i grupper på henholdsvis 2 og 3 biter.

Andre tallsystemer som benyttes er 12-tallsystemet (dusin=12), 20-tallsystemet (snes = 20) og 60-tallsystemet.

Tabell 1.1: Dekadisk, binært og heksadesimalt tallsystem.

Dekadisk	Binært	Heksadesimalt
00	0000	00
01	0001	01
02	0010	02
03	0011	03
04	0100	04
05	0101	05
06	0110	06
07	0111	07
08	1000	08
09	1001	09
10	1010	0A
11	1011	0B
12	1100	0C
13	1101	0D
14	1110	0E
15	1111	0F
16	10000	10

Kapittel 2

GENERELLE DATASTRUKTURER OG ALGORITMER

Dette kaptitlet forutsetter at man har en viss erfaring fra programmering i Pascal, C, Simula ol.. Stoffet er valgt med tanke på å gi en beskrivelse av de viktigste prinsipper for utforming av datastrukturer og formulering av algoritmer. En mere utfyllende behandling av emnet vil man for eksempel finne i [AHU83].

I eksempler og framstilling er det lagt vekt på å finne en romlig, spatial", vinkling av stoffet. Det som skiller GIS fra tradisjonelle administrative informasjonssystem, er blant annet at i GIS har vi flerdimensjonale nøkler (x,y,z,tid) og romlige relasjoner mellom objektene. Dette øker kompleksiteten til den programvaren som skal behandle slike data.

2.1 Datatyper og datastrukturer

Den programteksten vi skriver og de programdata vi manipulerer, lagres som en sekvens av biter i maskinens minne. Et *bit* er en celle som holder verdien 0 eller 1. Rent fysisk er disse verdiene representert ved en elektrisk ladning som enten er på eller av. Et typisk segment av maskinens minne kan se slik ut:

$$\cdots 00111001000011110101011110010111101001 \cdots$$

Denne samling av biter er på dette nivået uten struktur. Det er vanskelig å snakke om en bitstreng på en meningsfull måte.

Struktur legges på bitstrengen ved å betrakte grupper av biter, ofte kalt *byte* og *ord*. Vanligvis består en byte av 8 biter. Et ord er typisk satt sammen av 16, 32 eller 64 biter. Lengdene av en byte og et ord er *maskinavhengig*.

Organiseringen av minnet tillater oss å referere til en bestemt samling biter. Det er derfor mulig å snakke om ordet i adresse 1024 eller byten i adresse 1040 eller at

ordet i adresse 1032 ikke er lik ordet i adresse 1048, men det er selvsagt ikke mulig å snakke menigsfullt om innholdet i adresse 1032 uten at vi kjenner typen til den verdi som er representert.

Typeabstraksjon tillater oss å gjøre meningsfull tolking av en bitstreng av en bestemt lengde i en gitt adresse. Programmeringsspråk som Pascal har noen forhåndsdefinerte datatyper som integer, real, boolean og char, men det er også mulig å definere sine egne datatyper.

Begrepene datatype, datastruktur og abstrakt datatype høres like ut, men de har forskjellig mening. I et programmeringsspråk forteller *datatypen* til en variabel hvor stor lagerplass variabelen blir tildelt og hvilke operasjoner det er lov å benytte på den variable. For eksempel dersom vi oppretter en integer variabel, får vi på en PC vanligvis reservert 16 biter mens en dobbel integer vil reservere 32 biter. På en integer har vi i Pascal lov til å utføre de fire regnearter, mens dette ikke er tillatt på datatypen character. I maskinen vil selvsagt både integer og character representeres ved en samling biter, slik at ved kun å studere bitstrengen til en variabel er det ikke mulig å avgjøre hvilken datatype vi har med å gjøre.

En *abstrakt datatype* (ADT) er en matematisk modell, sammen med visse operasjoner som defineres på modellen. Formulert på en annen måte: en ADT er en datastruktur pluss operasjoner på datastrukturen. Et punktvadtre, for eksempel, sammen med operasjoner på treet, er en ADT. Operasjoner det er naturlig å definere på et punktkvadtre er: tøm-treet, sett-inn-punkter, slett-punkter, finn-alle-punkter-innenfor-et-område etc. .

For å representere en ADT benyttes en *datastruktur*, som er en samlig variable, ofte av ulik datatype, knyttet sammen på en eller annen måte. Datastrukturen opprettes ved å gi navn til grupper av celler og tolke verdiene til noen av cellene som forbindelseslinjer (pekere) mellom cellene. Vi kan si at datastrukturen definerer en gruppe variable og deres adkomstbaner. En trestruktur kan oppfattes som en datastruktur!

2.1.1 Celle

Med *celle* mener vi en gruppe biter som skal tolkes med en eller annen datatype. Cellen er den grunnleggende byggestein i en datastruktur.

2.1.2 Array

Den enkleste mekanisme for å gruppere celler i Pascal og de fleste andre programmeringsspråk, er det endimensjonale array. Et array er en sekvens av celler av en gitt type. Vi kan tenke oss et array som en mapping fra et sett indekser (slik som heltallene 1,2,.....,n) til cellene i arrayet. Deklarasjonen

linje: **array** [*indekstype*] **of** *koordinattype;*

definerer *linje* til å være en sekvens av celler av typen *koordinattype*.

I det tilfellet at dataene i et array er ordnet på en eller annen måte, vil settet av indekser representere implisitte relasjoner mellom cellene i arrayet. Dersom vi digitaliserer en linje, for eksempel, og legger koordinatene fortløpende inn i et array, blir punktenes rekkefølge i arrayet den samme som langs linja. I dette tilfelles er punktenes topologi (naboskap) implisitt definert ved settet av indekser.

2.1.3 Record

En annen vanlig mekanisme for å gruppere celler i programmeringsspråk, er *record* strukturen. En record er en blokk som vi gir bestemte egenskaper. Blokken gis et navn og alle celler i blokken identifiseres ved blokknavn (familienavn) og cellenavn (egennavn). Slike blokker benyttes ofte som celler i array, som forgreiningspunkter (noder) i trestrukturer eller som knutepunkter (noder) i nettverksstrukturer.

For eksempel vil en definisjon av et array som består av celler som inneholder en del informasjon om hus i et distrikt, kunne se slik ut:

```
type
    hustype = record
        x,y,z: integer;
        husnummer: integer;
        huseier: character
    end;
var
    hus: array [ 1..500 ] of hustype;
```

2.1.4 Klasser

I den seneste tiden har de objektorientere språk kommet stadig sterke på banen. Objektorienterte språk er for eksempel: Simula, C^{++}, Java og Pascal 5.5.

Innenfor disse språkene benyttes *klassebegrepet*. Vi kan betrakte en klasse som en utvidelse av en record. I en klasse finner vi data (variable) tilsvarende som i en record, men i tillegg kan vi i en klasse også definere funskjoner (*metoder*). Metodene benyttes til å gi verdier til klassens variable. Vi kan sette restriksjoner på hvilke deler av programsystemet som får lov til å benytte data og metoder i en klasse. Dersom dataene erklæres *private* og metodene *public* betyr det at dataene i klassen kun kan endres ved bruk av klassens metoder mens klassens metoder kan benyttes av alle klasser. I dette tilfellet er klassen en *abstrakt datatype.*

Ved å sette adgangsrestriksjoner på data og metoder oppnås en stor grad av datasikkerhet. Dersom vi oppdager at programmet gir feil verdier til en variabel, kan vi konsentrere feilletingen om den/de klasser som har fått lov til å endre verdien til variabelen. Innenfor systemeringsteori har man et gyldent prinsipp om å maksimere kohesjonen innenfor en blokk og minimere koblingen mellom blokker. Som vi nylig så

av eksemplet, støtter objektorientert programmering dette prinsippet. Det er nevnte prinsipp som ligger til grunn for innføring av klassebegrept i programmeringsspråk.

En slagkraftig egenskap ved klasser er arve-mekanismene. Ved hjelp av arving kan vi overføre data og metoder fra én klasse til en annen. Dette gjør at vi kan benytte eksisterende klasser når vi skal lage nye klasser.

Her er et lite eksempel på en klassedefinisjon i C^{++} syntaks:

```
class Punkt // definerer klassen Punkt
{
private:
//kan kun benyttes av denne klassen
     int x,y; // data
public:
// kan benyttes av alle klasser
     Punkt(int ,int ); // metode
};

Punkt::Punkt(int x, int y )
// metode som oppretter en hendelse av klassen Punkt
// og gir verdier til dataelementene.
{
     Punkt.x=x;
     Punkt.y=y;
}
```

Operatoren :: er den såkalte *tilhørighetsoperator*. Punkt::Punkt() blir da å lese som: til klassen Punkt hører metoden Punkt()". Operatoren . leses på samme måte som i Pascal. Punkt.x=x tolkes som: klassen Punkt sin x settes lik x".

2.1.5 Pekere og markører

En *peker* er en variabel hvis bitgruppe skal tolkes som en maskinadresse. Relasjoner mellom celler kan representeres ved bruk av pekere. En peker er som nevnt en celle hvis verdi indikerer en annen celle. Når vi tegner et bilde (lager en modell) av en datastruktur, vises det faktum at at celle A er en peker til celle B ved å tegne en pil fra A til B.

I Pascal kan vi opprette en pekervariabel p ved deklarasjoner av typen:

```
p: ↑celletype;
```

p er her en peker til celler av typen *celletype*.

I noen programmeringsspråk finnes ikke datatypen pekere, for eksempel Fortran-85. I dette tilfellet kan vi benytte integer variable som pekere til elementer i array. Pekere av denne typen benevnes av og til som *markører* (eng. cursor). Et eksempel på bruk av markører finnes i figurene 2.26 og 2.30.

2.1.6 Lister

En liste er en sekvens av elementer av en bestemt type. Vi skriver gjerne en liste ved å skille elementene med et komma, som for eksempel:

$$a_1, a_2, a_3, a_4, \cdots, a_n$$

hvor $n \geq 0$ og a_i er av elementtype. Antall elementer kalles gjerne *lengden* av listen. Dersom $n \geq 1$ sier vi at a_1 er det *første* elementet og a_n det *siste* elementet. Dersom $n = 0$ har vi en *tom* liste.

En viktig egenskap ved lister er at de kan ordnes lineært i samsvar med posisjonen i listen (for eksempel et sortert array).

Dersom vi definerer operasjoner på den matematiske modellen liste, har vi den abstrakte datatype liste". Operasjoner det er naturlig å definere er: sett-inn-element, finn-element, slett-element, neste-element, tøm-listen osv. .

Lister kan implementeres ved bruk av array eller ved bruk av Pascalpekere. Når man benytter Pascalpekere, kjedes et element til sine to naboer enten ved at vi har pekere i bare en retning, eller ved at vi har pekere i begge retninger. Vi snakker henholdsvis om en *enkeltlenket* liste og en *dobbeltlenket* liste.

Stakk

Stakk er en en spesiell liste der all innsetting og sletting gjøres i den ene enden, gjerne kalt stakkens *topp*. Et annet navn på stakk er *sist-inn-først-ut* liste. Konsekvensen av at innsetting og sletting gjøres i stakkens topp, er at det elementet som sist kom inn, er det elementet som først blir slettet (behandlet).

Kø

En *kø* ligner på en stakk, men med den forskjell at i en kø gjøres innsetting og sletting i hver sin ende av listen. Følgen av dette er at det elementet som først kom inn er det elementet som først blir slettet (behandlet). En kø kalles derfor ofte for en *først-inn-først-ut* liste.

Dersom vi benytter array-representasjon av en kø, er det lurt å tenke på arrayet som *sirkel*, hvor første posisjon følger etter den siste. La oss anta at vi etter en stund har fylt opp arrayet til vi har nådd siste posisjon. Dersom et nytt element nå skal settes inn, ville vi normalt ha gitt en melding tilbake om at arrayet er fullt. På grunn av at vi oppfatter arrayet som en sirkel, sjekker vi i stedet om posisjon 1 er ledig. Dersom den er det, settes det nye elementet inn der. Grunnen til at posisjon 1 er ledig, er selvsagt den at det elementet som har befunnet seg der, er slettet. For å holde rede på starten og slutten av køen, innføres to variable *start* og *slutt* som holder indeksen til henholdsvis første og siste element. Kriteriet på at køen er full, er:

if *(slutt + 1)* **mod** n $\geq$ *start* **then** *feil('køen er full')*

hvor **mod** gir resten ved heltallsdivisjon og n angir maksimal lengde til arrayet.

2.1.7 Rekursive prosedyrer

Språk som Pascal, C, Simula og Prolog tillater bruk av *rekursive prosedyrer*, men høynivåspråket Matlab gjør desverre ikke det. Rekursive prosedyrer er nøye knyttet til anvendelsen av stakk.

Anta av vi har en prosedyre P. Dersom P kaller seg selv, sies P å være en rekursiv prosedyre. For hver gang P kaller seg selv, opprettes en ny record på en stakk. I recorden finner vi informasjon om verdier til lokale variable og hvor i prosedyren P ble kalt fra. Vi benevner en serie rekursive kall til P med $P_1, P_2, \cdots, P_n$. Når P_i returnerer, hentes en record fra stakken og P_{i-1} fortsetter på linjen rett etter kallet til P_i.

Noen problemer løses enkelt ved å formulere løsningen rekursivt. Ulempen er at vi må bygge opp en stakk som vil konkurrere om lagerplass i primærminnet. Før man implementerer en rekursiv prosedyre, bør man derfor gjøre seg opp en mening om stakkens lengde. Se for eksempel senere om traversering av nettverk der både rekursive og ikke-rekursive prosedyrer blir diskutert.

Figur 2.1 gir et eksempel på en rekursiv prosedyre og hvordan stakken vokser og avtar. I eksemplet har vi valgt det problemet at alle pixel i et binærbilde som tilhører en sammenhengende figur, skal listes ut. Prosessen starter med at rutinen sjekk kalles med et pixel som tilhører figuren.

Stakken vil ha sin maksimale lengde dersom alle pixel i bildet er svarte, altså at samtlige pixel i bildet tilhører den samme figuren. Maksimal lengde er $\approx 3n$ når n er antall pixel i figuren.

2.1.8 Tidsforbruket til et program

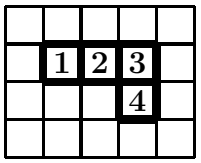
Anta følgende algoritme:

```
for i:=1 to n do
    for j:=1 to n do
        for k:=1 to n do
            writeln('hei');
```

Som vi ser har vi her tre nøstede løkker som vil gjøre at *hei* blir skrevet ut n^3 ganger. Den tid det tar å gjøre dette, vil være avhengig av hvor lang tid det tar å skrive ut en linje og hvor lang tid det tar å administrere løkkene. Tidsforbruket vil vi kunne beregne av $T(n) = cn^3$, hvor c er en konstant.

En meget viktig karakteristikk av en algoritme, er dens *tidskompleksitet* og *lagerkompleksitet*. Med tidskompleksitet mener vi vekstraten til tidsforbruket som en funksjon av størrelsen på problemet som skal løses (altså størrelsen på n). Lagerkompleksiteten defineres tilsvarende, men gjelder den lagerplass algoritmen krever.

```
procedure (rute: rutepeker);
    {finner alle pixel som tilhører en sammenhengende figur}
    begin
        if (hvit(rute) or (besøkt(rute)) then return ;
        {fortsetter dersom ruten ikke er besøkt
        eller den ikke er hvit}
            merk(rute); {merker ruten besøkt}
            writeln( 'rutenummer:', rute.nummer );
            sjekk(rute.høyre); {kaller de fire direkte naboer}
            sjekk(rute.venstre);
            sjekk(rute.opp);
            sjekk(rute.ned)
    end;
```



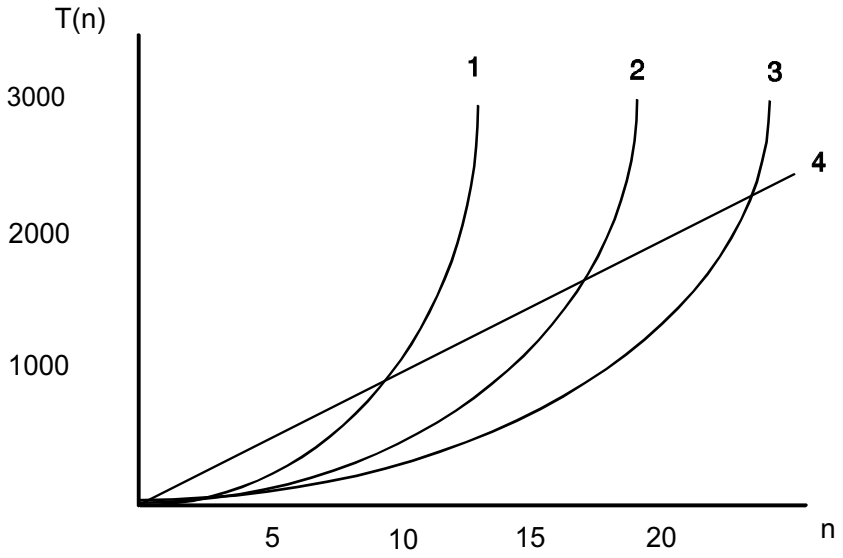
binært bilde

1.høyre = 2
1 venstre opp ned
2.høyre = 3
2 venstre opp ned
1 venstre opp ned
3.høyre = hvit
3 venstre opp ned
2 venstre opp ned
1 venstre opp ned
3.venstre = besøkt
3 opp ned
2 venstre opp ned
1 venstre opp ned
3.opp = hvit
3 ned
2 venstre opp ned
1 venstre opp ned

3.ned = 4
2 venstre opp ned
1 venstre opp ned
4.høyre = hvit
4 venstre opp ned
2 venstre opp ned
1 venstre opp ned
4.venstre = hvit
4 opp ned
2 venstre opp ned
1 venstre opp ned
4.opp = besøkt
4 ned
2 venstre opp ned
1 venstre opp ned
4.ned = hvit
2 venstre opp ned
1 venstre opp ned

2.venstre = besøkt
2 opp ned
1 venstre opp ned
2.opp = hvit
2 ned
1 venstre opp ned
2.ned = hvit
1 venstre opp ned
1.venstre = hvit
1 opp ned
1.opp = hvit
1 ned
1.ned = hvit
.
stakken er tom
rekursjonen terminerer

Figur 2.1: Stakk og rekursiv prosedyre.



Figur 2.2: Tidsforbruket til fire program.

Tidskompleksiteten angis ved den såkalte O-notasjon. Når vi sier at beregningstiden til et program er $O(n^2)$, mener vi at det finnes positive konstanter c og n_0 slik at for $n \geq n_0$ har vi at $T(n) \leq cn^2$. Dersom vi for eksempel har at $T(n) = an^3 + bn^2$ vil vi ved å velge $c =\mid a + b \mid$ og $n \geq 1$ ha at:

$$T(n) = an^3 + bn^2 \leq cn^3 = O(n^3)$$

Vi sier at $T(n)$ er $O(f(n))$ dersom vi vet at $f(n)$ er en *øvre grense* for vekstraten til $T(n)$. For å spesifisere en *nedre grense* for vekstraten til $T(n)$ benyttes notasjonen $T(n)$ er $\Omega(g(n))$. Vi mener da at det finnes en konstant $c \geq 0$ slik at $T(n) \geq cg(n)$.

For å kontrollere at funksjonen $T(n) = n^3 + 2n^2$ er $\Omega(n^3)$, velger vi $c = 1$ og får

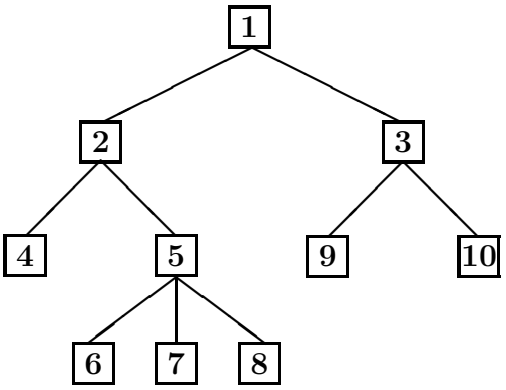
$$T(n) = n^3 + 2n^2 \geq cn^3 = \Omega(n^3)$$

Vi skal være klar over at to program som har samme tidskompleksitet, kan ha meget forskjellig beregningstid. Tidskompleksiteten er derfor ingen nøyaktig modell for tidsforbruket, men den sier bare noe om hvor *slemt* problemet er. Figur 2.2 gir et eksempel på dette. Som vi ser av figuren vil program 1 ha en kvadratisk vekst i tidsforbruket, men når $n \leq 8$ er tidsforbruket mindre enn for program 4 som har en lineær vekst i tidsforbruket. For store n vil selvsagt den lineære algoritmen vinne.

2.2 Trær

Dersom $n_1, n_2, \cdots, n_k$ er en sekvens av noder slik at n_i er forgjengeren til n_{i+1} for $n \leq i < k$, kalles denne sekvensen en *sti* fra n_1 til n_k. *Lengden* til en sti er 1 mindre enn antall noder i stien.

Høyden til en node i treet defineres som den *lengste sti* fra noden til et av dens blader. Høyden til et tre er høyden til roten. Samtlige blad har høyden 0. Høyden til treet i figur 2.3 er 3, høyden til node 5 er 1 mens node 4 har høyden 0.



Figur 2.3: Et tre.

Etterfølgerne til en node er vanligvis ordnet etter et eller annet prinsipp. Vi snakker da om et *ordnet* tre. Dersom etterfølgerne ikke er ordnet, har vi et *uordnet* tre.

Ved analyse av trestrukturer kommer ofte formelen for summen av en geometrisk rekke til anvendelse. Hvis første ledd i en geometrisk rekke er a_1 og kvotienten er k, blir summen av rekkens n første ledd:

$$S_n = a_1 \frac{k^n - 1}{k - 1} \tag{2.1}$$

I et binærtre med høyde h, er det maksimale antall noder treet kan inneholde

$$N = \frac{2^{h+1} - 1}{2 - 1} \approx 2^{h+1}$$

som gir:

$$h = \log_2 N - 1$$

2.2.1 Datastrukturer for trær

Det finnes flere metoder for å representere et tre i en datamaskin. I hovedsak kan man velge mellom arrayrepresentasjon og bruk av Pascal-pekere eller en kombinasjon av disse to. Hvorvidt man skal legge inn mulighet til å kunne navigere fra roten til bladene, fra bladene til roten eller begge deler, kan tilpasses behovet til den aktuelle applikasjon. I noen tilfeller er det også hensiktmessig å kjede sammen noder som ligger på samme nivå i treet (horisontale pekere).

Foreldrerepresentasjon

En enkel implementasjon som tilfredsstiller bunn-oppnavigasjon i et tre, baserer seg på at en node har kun én forgjenger. Ved å benytte et 1-dimensjonalt array får vi følgende representasjon av treet i figur 2.3:

indeks	1	2	3	4	5	6	7	8	9	10
Array	0	1	1	2	2	5	5	5	3	3

Figur 2.4: Foreldre-representasjon av et tre.

Roten i treet har ingen forgjenger. Dette er markert ved å sette en umulig adresse (0) inn i rotens posisjon i arrayet. Alternativt kunne man latt roten peke til seg selv.

Arrayrepresentasjon

Binære trær og kvadtrær er det ofte hensiktsmessig å representere i et lineært array. Et binærtre kan for eksempel defineres på følgende måte:

```
var
    tre : array [ 1..maksnoder ] of record
        venstrebarn: integer
        høyrebarn: integer
    end;
```

En liste av etterfølgere

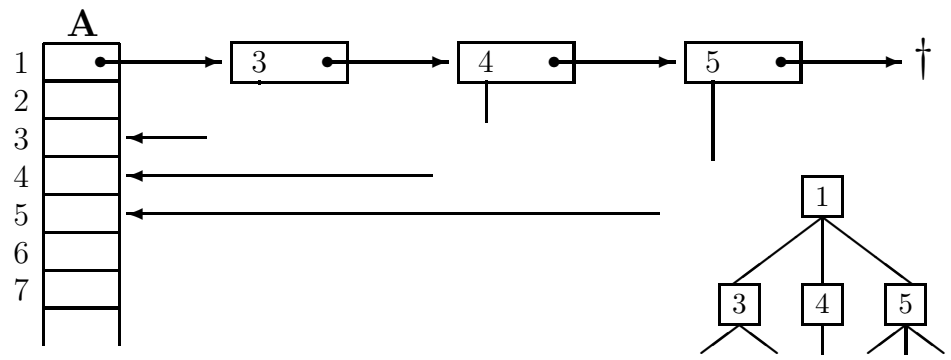
En meget viktig metode for representasjon av trær er å lage en liste av etterfølgere for hver av nodene i treet. Dersom listen av etterfølgere er av svært varierende lengde for de ulike noder, er det en fordel å benytte pekerbaserte lister.

Figur 2.5 viser en modell av datastrukturen samt en mulig definisjon. Nodene i treet legges i arrayet A. Den variable *rot* er strengt tatt ikke nødvendig dersom man alltid legger roten i posisjon 1 i arrayet. Cellene i arrayet er definert i record *trenode*. Her finner vi en peker *neste* til den første noden i listen av etterfølgere til vedkommende trenode. Nodene i listen av etterfølgere er definert i record *listenode*.

Pekerbasert representasjon

Det er selvsagt også mulig å benytte Pascal-pekere i stedet for markører. En definisjon av et kvadtre vil for eksempel da kunne se slik ut:

```
type
trepeker = ↑trenode;
    trenode = record
        nordvest: trepeker;
```




```
type
    listepeker = ↑listenode;
    listenode = record
        etterfølger: integer;
        neste: listepeker;
    end;
    trenode = record
        neste: listepeker;
        info: nodeinfo;
    end;
    tre = record
        A: array [ 1..maksnoder ] of trenode;
        rot: integer
    end;
```

Figur 2.5: Datastruktur for lenket liste-representasjon av et tre.

```
        nordøst: trepeker;
        sydvest: trepeker;
        sydøst: trepeker;
        info: infotype {informasjon i noden}
    end;
```

2.2.2 Traversering av trær

Det finnes flere nyttige måter for systematisk å besøke nodene i et tre. De viktigste av disse måtene er *preorder, inorder* og *postorder* traversering. Følgende tabell definerer disse traverserings- teknikkene for et binærtre:

Preorder	Inorder	Postorder
rot	venstre subtre	venstre subtre
venstre subtre	rot	høyre subtre
høyre subtre	høyre subtre	rot

En generalisering av disse teknikkene til trær med flere greiner enn to, er rett fram for preorder og postorder. For inorder må man velge hvor mange av en nodes etterfølgere som skal besøkes før man skal gjøre noe med noden.

```
procedure inorder(p: trepeker);
{inorder-traversering av et binærtre}
    begin
        if p=nil then return ;
        inorder(p↑.venstre);
        writeln(p↑.info); {skriver ut innholdet i noden}
        inorder(p↑.høyre);
    end;
```

Figur 2.6: Rekursiv procedyre for inorder traversering av et binærtre.

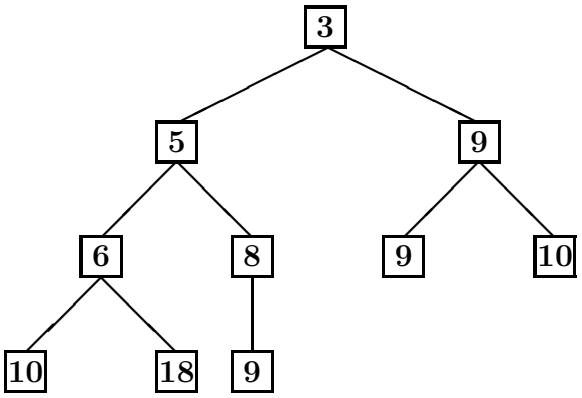
I figur 2.6 har vi beskrevet inordertraversering av et binærtre. Inordertraversering av et ordnet binærtre vil føre til at nodene blir skrevet ut i stigende tallfølge.

2.3 Prioritetskø

En *prioritetskø* er en abstrakt datatype (ADT) basert på en sett-modell med operasjonene NULLSTILL, SETTINN og SLETTMINSTE. Begrepet prioritetskø kommer av at en rekke forespørsler legges i kø etter en eller annen vedtatt prioritet. Vi kan for eksempel tenke oss et flebruker datasystem og at en rekke brukere er interessert i å benytte de samme ressurser på samme tid. Siden systemet kan håndtere kun en bruker av gangen, oppstår en konfliktsituasjon. Dette løses ved at brukerne må vente på tur, det opprettes en kø. Rekkefølgen i køen bestemmes av et eller annet kriterium (prioritet). Prioriteten kan for eksempel beregnes av oppdragets størrelse, det tidspunkt ressursen ble bestilt på, administrative bestemmelser om prioritet til ulike brukergrupper etc. .

Det finnes flere mulige datastrukturer for en prioritetskø, men det vanligste er kanskje å baserer seg på et *balansert, delvis ordnet* binærtre som representeres ved et lineært array.

I et delvis ordnet binærtre er tallet som er lagret i en node, mindre enn eller lik tallene i det etterfølgende subtre. Roten i treet vil følgelig inneholde det minste tallet. Figur 2.7 viser et delvis ordnet binærtre. Det er selvsagt ikke noe veien for å snu betingelsen slik at det sørste tallet legges i roten.



Figur 2.7: Et delvis ordnet tre.

For å holde treet så *balansert* som mulig, gjøres innsetting og sletting på en bestemt måte.

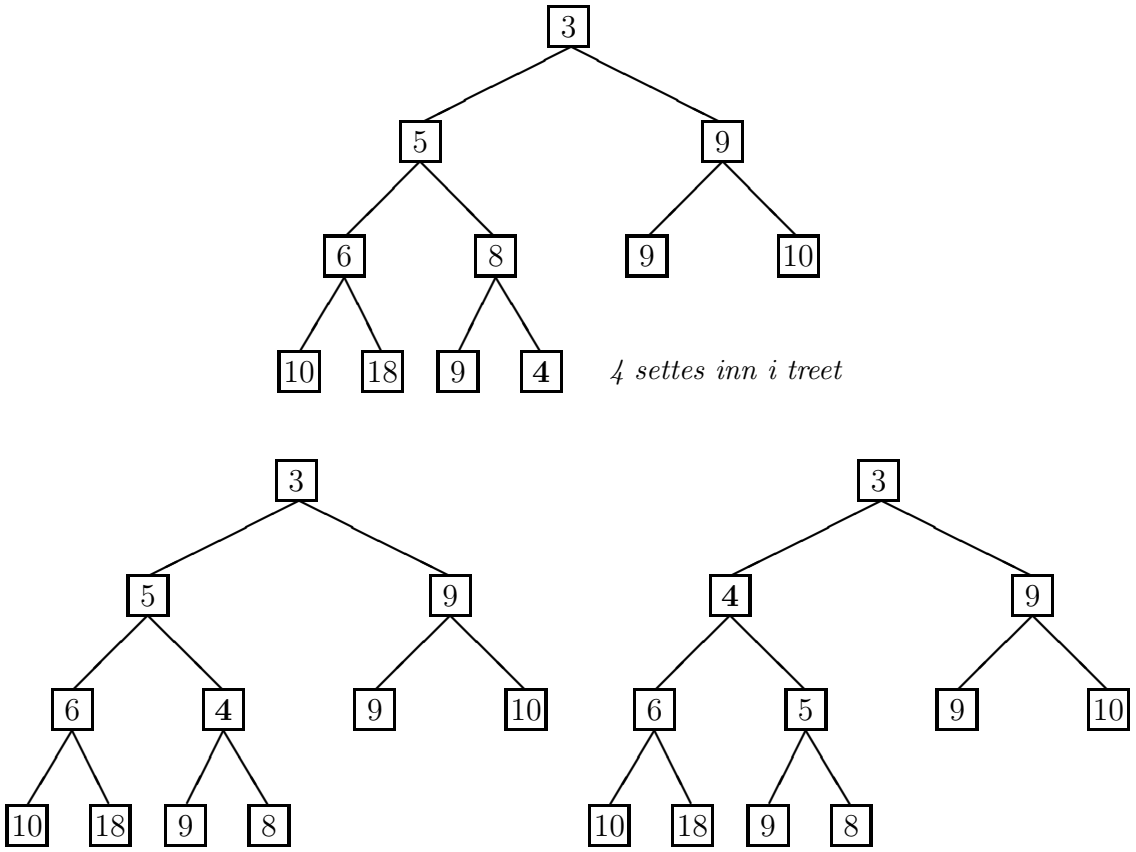
SETTINN starter med at det nye tallet settes inn på treets laveste nivå og så langt til venstre som mulig. Dette sikrer oss at treet holdes balansert. Betingelsen om ordningen i treet tilfredsstilles ved at vi lar det nye tallet boble opp så langt i treet det kan komme. Dersom tallet er mindre enn sin forgjenger, bytter de plass. Slik fortsetter det inntil tallet har kommet på en plass der forgjengeren er mindre enn eller lik tallet. Figur 2.8 viser innsetting av tallet 4.

SLETTMINSTE er i første omgang enkelt siden det minste tallet ligger i treets rot. Etter at tallet i roten er fjernet, er den tom og trenger påfyll. Påfyll gjør vi ved å hente det tallet som ligger lengst til venstre på treets nederste nivå. Dette sikrer oss at treet holdes balansert. For å ivareta kravet til ordning i treet, lar vi denne gangen tallet falle så langt ned i treet det kan komme. Dette gjør at det minste tallet blir flyttet opp til treets rot, forutsatt at vi bytter tallet med den minste av etterfølgerne til den noden tallet til enhver tid befinner seg i. For eksempel i figur 2.9 slettes 3 fra roten. Deretter flyttes 8 opp til roten og fallprosessen starter. Siden 4 er rotens minste etterfølger, lar vi 8 og 4 bytte plass. Deretter bytter 4 og 5 plass og prosessen kan terminere.

Det faktum at treet er binært, at det er så balansert som mulig og at alle noder på nederste nivå ligger så langt til venstre som mulig, gjør at vi kan benytte en heller uvanlig representasjon av treet. Dersom det finnes n noder, benytter vi de n første posisjoner i et array A. Roten legges i $A[1]$. Det venstre barnet til node $A[i]$, dersom det eksisterer, befinner seg i node $A[2i]$, og det høyre barnet til node $A[i]$, dersom det eksisterer, befinner seg i node $A[2i+1]$. En ekvivalent betraktningsmåte er at foreldrenoden til $A[i]$ er $A[i$ **div** $2]$.

Basert på den beskrivelse av prioritetskø som nå er gitt, skal vi detaljere algoritmer og datastrukturer. Til orientering så kommer vi blant annet tilbake til prioritetskø i forbindelse med algoritmen korteste vei i et nettverk.

type

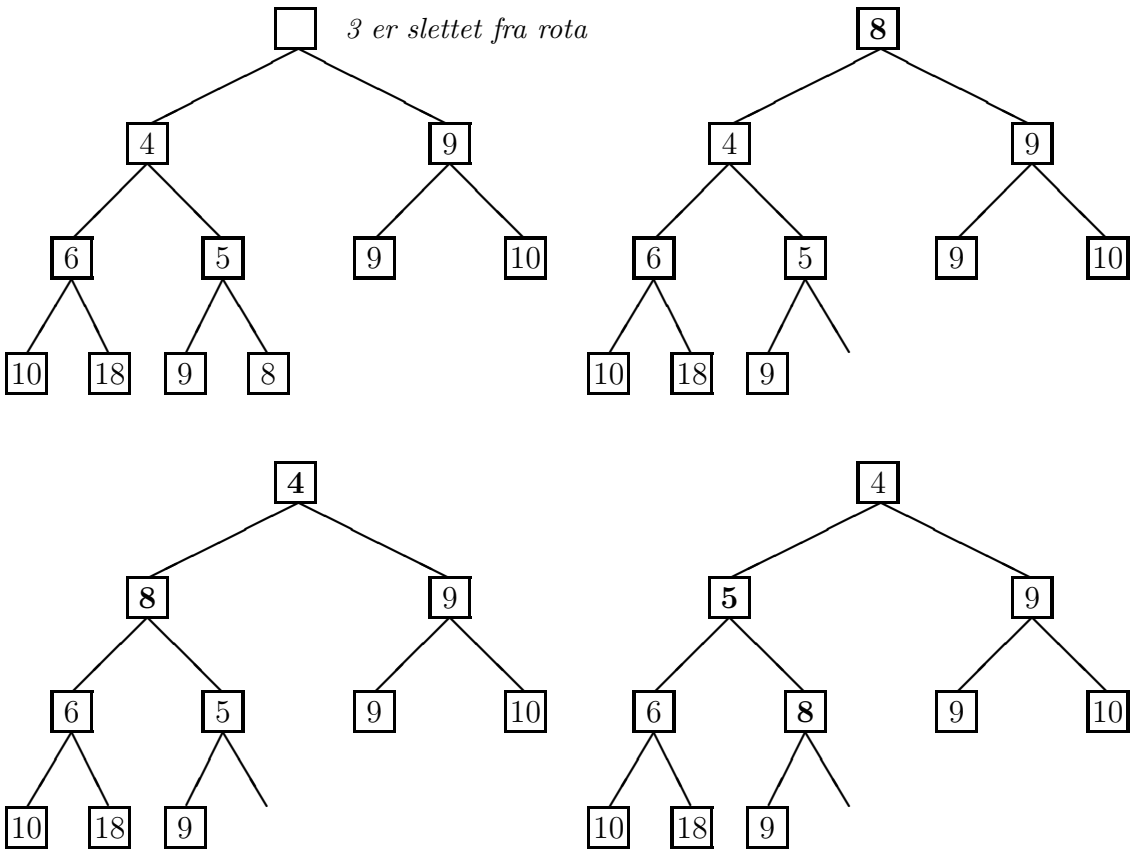


Figur 2.8: Innsetting av et element.

```
prosesstype = record {node i prioritetskøen}
        id: integer; {prosessidentifikator}
        prioritet: integer
    end;


PRIORITETSKØ = record
        innhold := array [ 1..makslengde ] of prosesstype;
        siste: integer
    end;


function p( a: processtype ) : integer;
    {beregner prioriteten til et element}
    begin
        p:= a.prioritet {tilpasses den aktuelle oppgave}
```



Figur 2.9: Sletting av minste element.

end;

procedure *NULLSTILL* (**var** *A: PRIORITETSKØ*);
{nullstiller prioritetskøen}
begin
A.siste := 0
end; {MAKENULL}

procedure *SETTINN* (*x: prosesstype;* **var** *A: PRIORITETSKØ*);
{setter et nytt element inn i køen}
var
i: integer;
temp: prosesstype;
begin

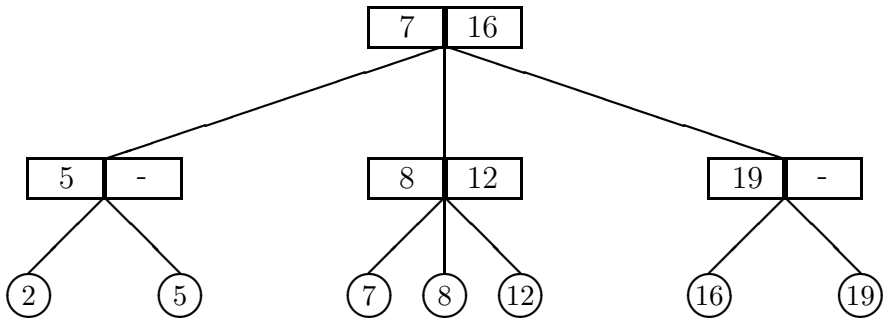
```
        if A.siste ≥ makslengde then feil('køen er full')
        else begin
            A.siste := A.siste + 1;
            A.innhold[ A.siste ] := x;
            i := A.last; {i er aktuell posisjon til x}
            while (i > 1)
            and (p(A.innhold[ i ]) < p(A.innhold[ i div 2 ])) do be-
gin
                {x bobler oppover i treet}
                temp := A.innhold[ i ];
                A.innhold[ i ]:= A.innhold[ i div 2 ];
                A.innhold[ i div 2 ] := temp;
                i := i div 2;
            end
        end
    end; { SETTINN }
```

```
procedure SLETTMINSTE (
            var A: PRIORITETSKØ ): ↑prosesstype;
    {sletter minste element}
    var
        i,j: integer;
        temp: prosesstype;
        minste: ↑prosesstype;
    begin
        if A.last = 0 then
            feil('køen er tom');
        else begin
            new(minste);
            minste↑.id := A.innhold[ 1 ].id;
            minste↑.prioritet := A.innhold[ 1 ].prioritet;
            SLETTMINSTE := minste;
                {vi skal returnere en kopi av roten}
            A.innhold[ 1 ] := A.innhold[ A.siste ];
            A.siste := A-siste - 1;
                {flytter siste element til begynnelsen}
            i := 1;
            while i ≤ A.siste div 2 do begin
                {lar elementet falle nedover i treet}
                if (p(A.innhold[ 2*i ]) < p(A.innhold[ 2*i+1 ]))
                or (2*i = A.siste) then
                    j:= 2*i
                else
                    j := 2*i + 1;
                if p(A.innhold[ i ]) > p(A.innhold[ j ]) then begin
                {bytter med det element som har lavest prioritet}
                    temp := A.innhold[ i ];
                    A.innhold[ i ] := A.innhold[ j ];
                    A.innhold[ j ] := temp;
                    i := j;
                end
                else
                    return {kan ikke dytte tallet lenger ned}
            end;
            return {tallet dyttet helt ned til et blad}
        end
    end; { SLETTMINSTE }
```

2.4 2-3Trær

For å kunne foreta effektive søkeopresjoner på trær, er det viktig at trærne er balanserte. I et balansert tre er kostnadene ved å gå fra treets rot til et av dets blader lik for alle bladene i treet. Ved gjentatte innsettinger og slettinger i treet er det nødvendig at man har en god strategi for å holde treet balansert, slik at kostnadene til å holde treet balansert er små. Et *2-3tre* oppfyller de nevnte ønsker og har følgende egenskaper:

- En intern node i treet har enten to eller tre etterfølgere.
- Treet holdes til enhver tid balansert.
- Treet er et ordnet tre.
- Dataene lagres vanligvis kun i løvnodene, men det er også mulig å implementere treet slik at dataene også lagres i de interne noder.



Figur 2.10: Et 2-3tre.

Vi vil i det etterfølgende forutsette at dataene (nøklene) kun lagres i treets løvnoder. I de interne noder finner vi derfor veivisere som bare har til hensikt å understøtte navigasjon i treet.

Datastrukturen for et 2-3tre kan defineres slik:

```
type
    nodetyper = set of (løv,intern);
    trepeker = ↑totrenode;
    totrenode = record
        case slag: nodetyper of
            løv: (nøkkel: integer);
            {løvnode}
            intern: (førstebarn,andrebarn,tredjebarn: trepeker;
```

minstetilandre, minstetiltredje:integer)
{intern node}
end;

I en intern node lagres to tall som gir henholdsvis den minste nøkkel som finnes i nodens midtre subtre (andrebarn) og den minste nøkkel som finnes i nodens høyre subtre (tredjebarn), dersom det høyre subtreet eksisterer. I figur 2.10 er for eksempel 7 og 16 lagret i roten siden 7 er det minste tallel som finnes i rotens midtre subtre og 16 det minste tallet vi finner i rotens høyre subtre.

Det prinsipp som ligger til grunn for balansering av treet, kan formuleres slik:

BALANSERING: Dersom en node q allerede har tre etterfølgere og får tilbud om en fjerde etterfølger, splittes q slik at vi får to nye noder q_1 og q_2, hver med to etterfølgere. De to minste tallene i q legger vi i q_1 mens de to øvrige går til q_2.

Deretter blir forgjengeren p til q spurt om den har plass til både q_1 og q_2. Dersom den har plass, er saken grei; q fjernes fra p og i stedet hektes q_1 og q_2 på p og veiviserne i p oppdateres.

Dersom p ikke har plass til q_1 og q_2, det vil si om p har tre etterfølgere, splittes p i nodene p_1 og p_2 og q_1 og q_2 hektes på hver sin av disse. Slik fortsettes prosessen ved at forgjengeren til p blir spurt om den har plass til p_1 og p_2.

Treet får et nytte nivå kun i det tilfellet at roten splittes.

Figurene 2.11 og 2.12 demonstrerer innsetting og sletting i et 2-3tre.

Vi kan nå gjøre den observasjon at antall nivåer k i et 2-3tre vil være gitt ved:

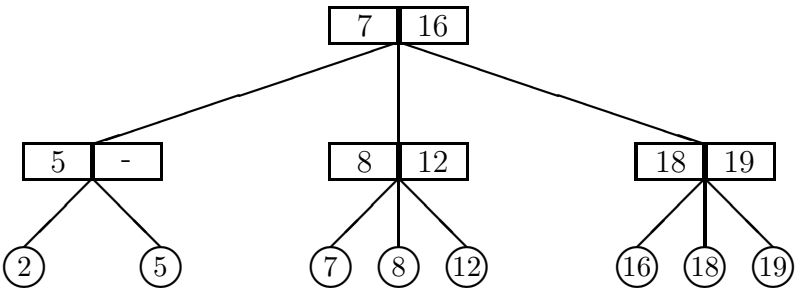
$$1 + \log_2(n) \leq k \leq 1 + \log_3(n)$$

Følgelig kan vi teste om en nøkkelverdi finnes i treet i $O(\log(n))$ tid.

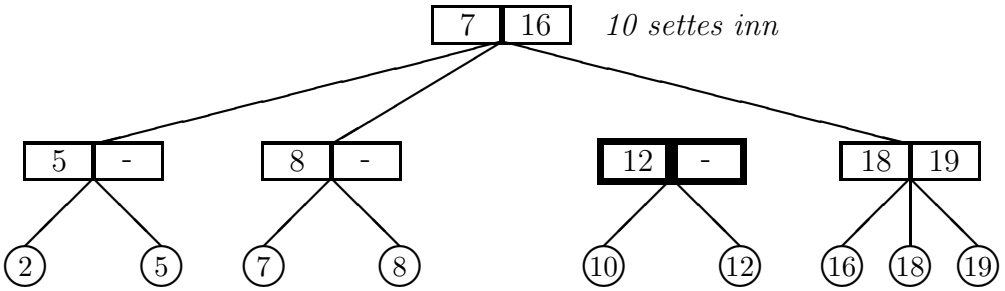
En annan observasjon vi gjør, er at nøklene på løvnivå kommer i stigende rekkefølge fra venstre mot høyre. Ved å legge inn horisontale pekere mellom løvnodene, kan vi derfor gjøre sekvensielle søk etter nøkler større enn nøkkelen i den løvnoden vi står i. Dette er en aktuell problemstilling ved spørsmål av typen: finn alle nøkler som ligger innenfor et gitt tallområde. Spørsmålet besvares ved først å søke opp den minste nøkkelverdien i intervallet for deretter å søke horisontalt etter de øvrige nøkler i intervallet.

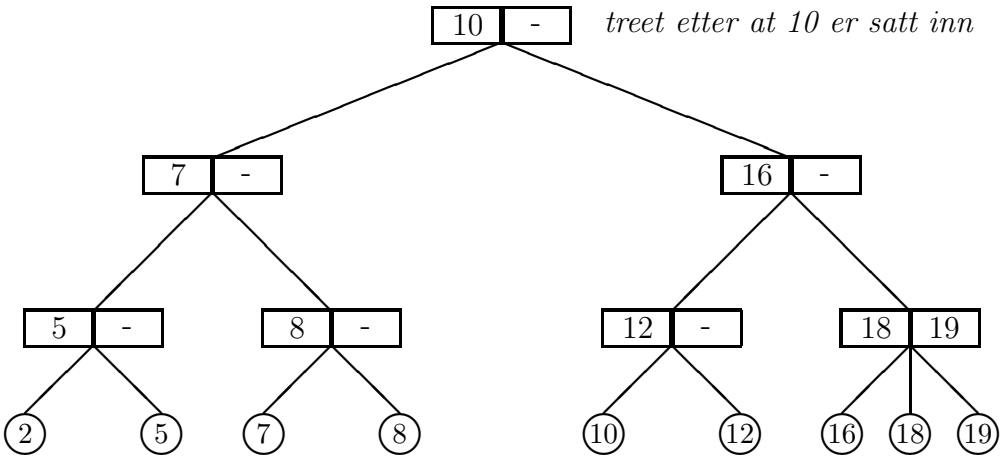
2.5 B-trær

De såkalte *B-trær* har blitt svært populære i databasesystem. Grunnen til deres popularitet er nok at de er godt egnet til å besvare intervall-spørsmål. Hvorfor navnet B-tre har blitt innført, er noe usikkert. Som utfyllende litteratur om B-trær kan anbefales [Com79]

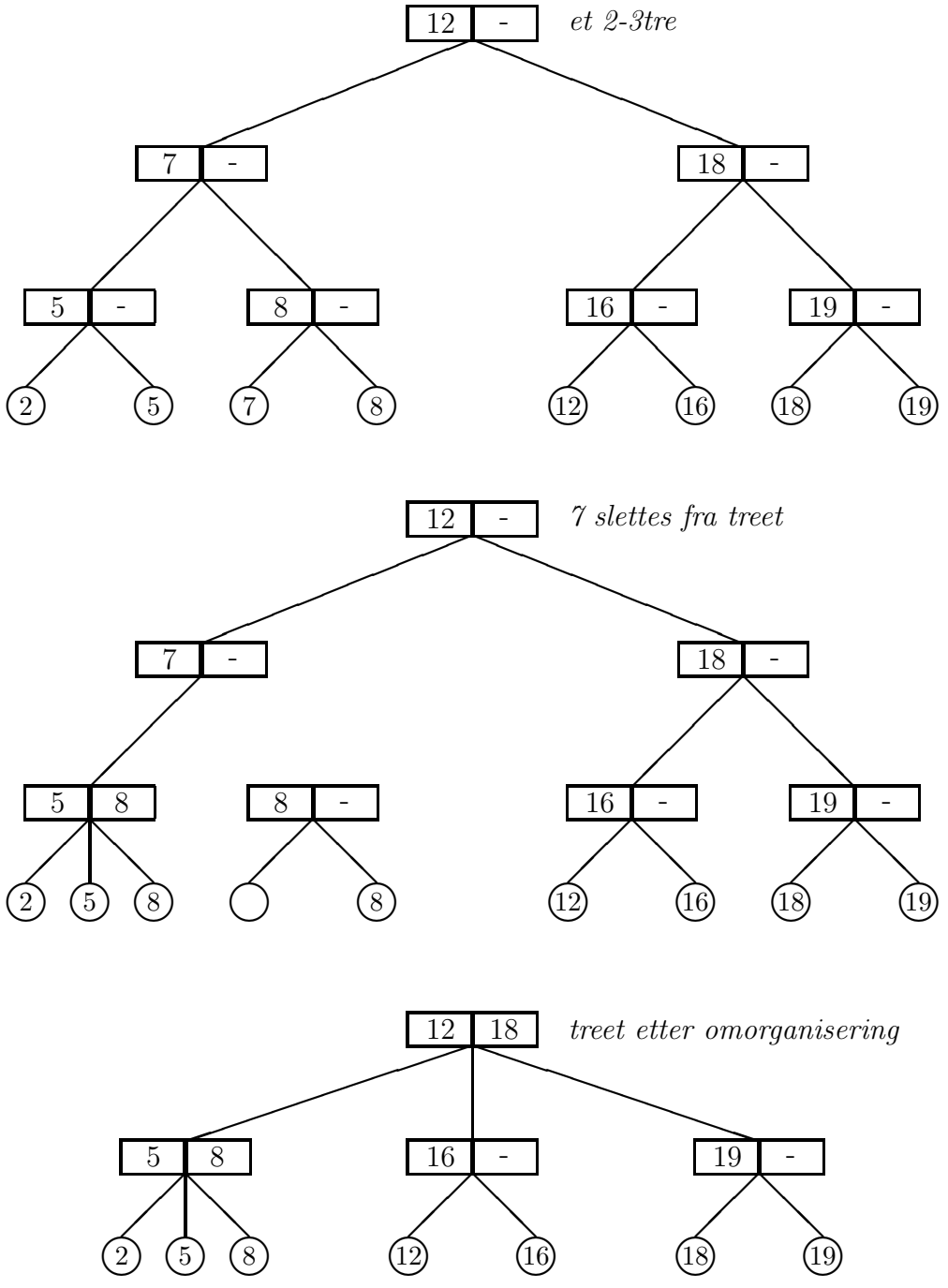


treet i figur 2.10 etter at 18 er satt inn





Figur 2.11: Innsetting i et 2-3tre.



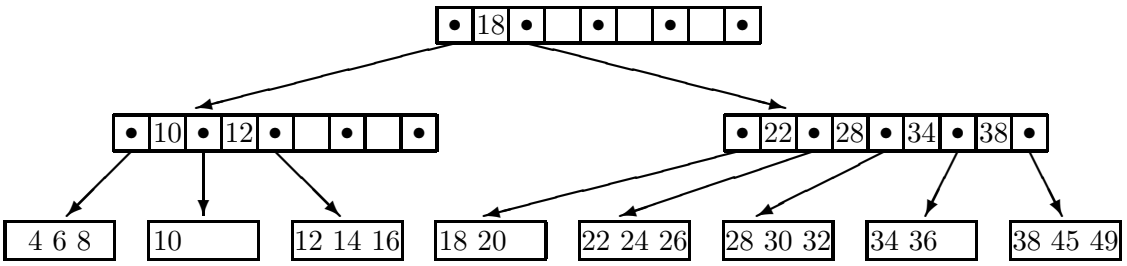
Figur 2.12: Sletting i et 2-3tre.

Prinsippet for å konstruere B-trær følger de samme regler som konstruksjon av 2-3trær. Vi vil derfor vise til vår gjennomgang av 2-3trær. Forskjellen mellom 2-3trær og B-trær stikker i at B-trær kan ha en stor forgreiningsfaktor. Dette gjør at B-trær er egnet for lagring på platelager. Ved å velge en stor nok forgreiningsfaktor, kan spørsmål om en nøkkel finnes i treet, besvares ved et lite antall diskaksesser. For å gjøre en presisering: B-trær benyttes på eksterne lagringsmedier mens 2-3trær benyttes som en intern datastruktur (i primærminnet). Figur 2.13 gir et eksempel på et B-tre.

De interne noder i et B-tre er av formen

$$(p_0, k_1, p_1, k_2, p_2, \cdots, k_n, p_n,)$$

hvor p_i er en peker til det i^{te} barn til en node, $0 \leq i \leq n$, og k_i er en nøkkel, $0 \leq i \leq n$. Nøklene i en node plasseres i sortert rekkefølge slik at $k_1 \leq k_2 \leq \cdots \leq k_n$. Alle nøkler i det subtreet p_0 peker på, er mindre enn k_1 og alle nøkler i subtreet til p_i er større enn eller lik k_i og mindre enn k_{i+1}. Nøklene i subtreet til p_n er større enn k_n.



Figur 2.13: B-tre av grad 5.

Et B-tre sies å være av *grad* m dersom hver node, med unntak av roten og bladene, har inntil m barn. Et 2-3tre er et B-tre av grad 3. Vanligvis forlanger man at nodene skal ha mellom $\lceil m/2 \rceil$ og m barn, men det finnes også eksempler på B-trær der man forlanger at nodene skal ha mellom $\lceil 2m/3 \rceil$ og m barn (de såkalte B^*-trær).

I et B^*-tre splittes en node først når 2 naboer er fulle. Ved en splitting vil innholdet i de to noder bli jevnt fordelt på tre noder, derav får vi at en node vil inneholde minimum 2/3 av sitt maksimale innhold. Når en node går full, testes det derfor først på om naboen har plass til flere nøkler. Dersom det er tilfellet, omstruktureres de to nodene slik at det blir plass til den nye nøkkelen. En splitt av forgjengeren får vi derfor først når også naboen er full.

Dersom nøklene lagres i løvnodene og de interne noder har mellom $m/2$ og m barn, snakker vi om et B^+-tre. B^+-trær er nok den mest utbredte varianten i B-tre familien. Grunnen til dette er nok de attraktive fordeler det oppnås ved å lagre nøklene i løvnodene. Nøklene vil nemlig komme i sortert rekkefølge fra venstre mot høyre. Ved å legge inn horisontale pekere mellom bladene, kan derfor intervall-søk gjøres med et minimum av diskaksesser.

De interne noder i et B^+-tre tjener kun som veivisere til nøklene i løvnodene. Vi kan derfor betrakte et B^+ som et indekstre til en sortert liste av nøkler.

Vi skal finne høyden i et B^+-tre under følgende forutsetninger:

1. treet er av graden m
2. i treet finnes n nøkler
3. hvert blad inneholder i gjennomsnitt b nøkler

På grunnlag av disse forutsetningene finner vi at det er $\lceil n/b \rceil$ løvnoder i treet og at det det minste antall barn til en intern node er $m/2$. Antall forgjengere til løvnodene er da

$$\frac{\left\lceil \frac{n}{b} \right\rceil}{\frac{m}{2}}$$

og antall forgjengere til forgjengerne er

$$\frac{\left\lceil \frac{n}{b} \right\rceil}{\left(\frac{m}{2}\right)^2}$$

Dersom det er j noder langs stien fra roten til et blad, har vi at

$$\frac{\left\lceil \frac{n}{b} \right\rceil}{\left(\frac{m}{2}\right)^{j-1}} \geq 1$$

for ellers ville det vært mindre enn 1 node på rotnivå. Herav finner vi

$$j \leq 1 + \log_{m/2} \left\lceil \frac{n}{b} \right\rceil$$

For eksempel dersom $n/b = 10^6$ og $m = 200$, får vi $j = 4$. Dersom vi i dette tilfellet legger rotblokken og blokkene på neste nivå inn i primærminnet, vil forespørsler om bestemte nøkler kunne besvares i løpet av 2 diskaksesser. De nevnte blokker må selvsagt også være lagret på platelageret, slik at det er ved oppstart av B-treet at den øvre del av treet blir løftet inn i primærminnet.

Datastrukturen for et B^+-tre kan defineres slik:

```
type
    nodetyper = (løv,intern);
    trepeker = ↑B-pluss-node;
    B-pluss-node = record
        case slag: nodetyper of
            løv: (nøkkel: integer; høyre-løvnode-nabo: trepeker);
            {løvnode, her lagres kun nøkler og en horisontal peker}
```

intern {intern node, benyttes for navigasjon i treet} *:*
(P: **array** [*1..m*] **of** *trepeker;*
K: **array** [*1..m-1*] **of** *integer)*
end*;*

Vi har valgt å kjede sammen løvnodene ved å etablere horisontale pekere. På denne måten etableres en sortert liste av løvnodene. Intervall-søk blir derfor, som tidligere nevnt, langt mere effektive i treet.

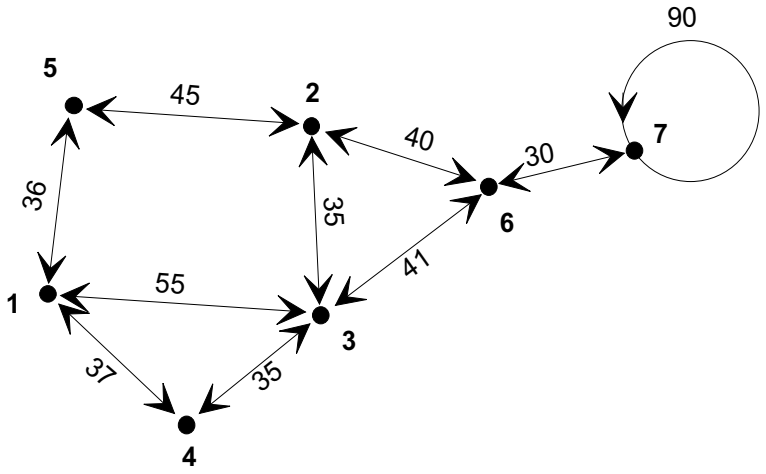
2.6 Nettverk

Innen en rekke fagdisipliner har man ofte behov for å representere vilkårlige relasjoner mellom objekter. Grafer (nettverk) er naturlige modeller for slike relasjoner. Innen den matematiske topologi spiller grafer en sentral rolle.

Et nettverk består av knutepunkter (noder, vertices) og veier (arcs). Veiene er forbindelser mellom knutepunktene og beskriver derfor sammenhengen i nettverket. I et nettverk kan en node ha forbindelse til alle andre noder i nettverket. En node kan derfor ha veier til n noder. Umiddelbart skulle man vente at en node bare kan ha inntil $n-1$ veier, men vi kan også ha en vei som starter i en gitt node og ender i den samme noden slik som vei 90 i figur 2.14.

Vi kan oppfatte et tre som et spesialtilfelle av et nettverk og en liste som et spesialtilfelle at et tre.

Vi snakker og *rettede* grafer og *urettede* grafer. I en rettet graf har veiene en retning, mens i en urettet graf tillegger vi ikke veiene atributtet retning.



Figur 2.14: Eksempel på et nettverk.

Eksempler på nettverk er: veinettverk, flyruteforbindelser, kabelnett, telekommunikasjonsnett, rørsystem osv. . Innenfor GIS spiller nettverk en viktig rolle. Elektrisitetsverket vil for eksempel vite hvor de elektriske kabler ligger, hvor koblingskummer

og koblingskasser er plassert, hvor lange kablene mellom to koblingspunkter er osv. .

2.6.1 Datastrukturer for nettverk

Nettverk kan representeres på flere måter. Den enkleste måten er kanskje å benytte et 2-dimensjonalt array (nabomatrise). Tabell 2.15 viser nabomatrisen til nettet i figur 2.14. Tallene i tabellen definerer avstanden mellom nodene. Dersom en forbindelse ikke eksisterer mellom to noder, settes avstanden til ∞ .

En ulempe med den todimensjonale nabomatrisen, er at den tar like mye plass i maskinen om det er få eller mange veier i nettverket. Fordelen med denne datastrukturen er at vi med et enkelt tabelloppslag kan få svar på om det finnes en vei fra node i til node j.

I det tilfellet at vi har mange noder og få veier, kan det være plassbesparende å benytte listestrukturen i figur 2.16. Ulempen er at vi da må bla i en liste for finne ut om det eksisterer en vei fra node i til node j.

		T I L						
		1	2	3	4	5	6	7
	1	∞	∞	55	37	36	∞	∞
	2	∞	∞	35	∞	45	40	∞
F	3	∞	∞	∞	35	∞	41	∞
R	4	37	∞	35	∞	∞	∞	∞
A	5	36	∞	∞	∞	∞	∞	∞
	6	∞	40	41	∞	∞	∞	30
	7	∞	∞	∞	∞	∞	∞	90

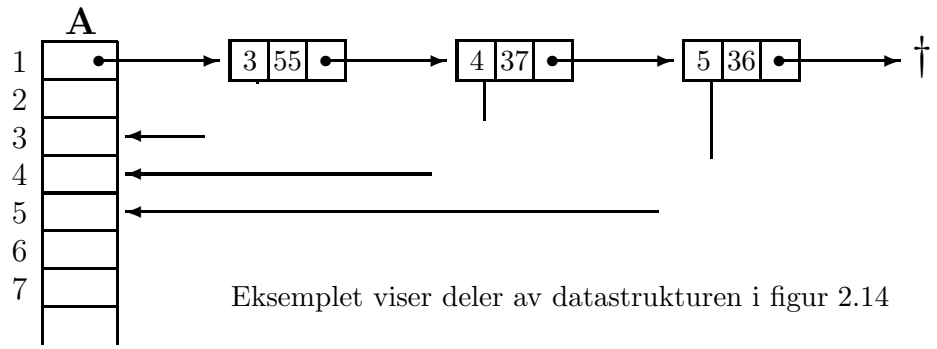
Figur 2.15: Array-representasjon av nett.

2.6.2 Korteste vei

En vanlig operasjon på nettverk er å finne *korteste vei* fra et knutepunkt til et annet. Med vei mener vi ikke bare den Euklidiske avstand mellom knutepunktene, men en hvilken som helst kvantitativ størrelse som uttrykker kostnaden ved å bevege seg i nettverket. Veilengden mellom to knutepunkter kan for eksempel være: den tid det tar å kjøre strekningen, bensinforbruket på strekningen, transportkostnader målt i kroner, vertikal- horisontal- eller skrå-avstand osv. .

Korteste vei problematikk er i dag en meget aktuell problemstilling for ambulanser og varetransport i de større byer i verden.

I [AHU83] gis to algoritmer for å beregne korteste vei (Dijkstras og Floyds algoritmer). Forskjellen på algoritmene er at Dijkstra forutsetter at alle veilengder er positive, mens Floyd takler negative veilengder. Tidskompleksiteten til Dijkstra er



Eksemplet viser deler av datastrukturen i figur 2.14

```
type
    veilistepeker = ↑veilistenode;
    knutepunktpeker = integer;
    veilistenode = record
        avstand: integer;
        neste: veilistepeker;
        vei: knutepunktpeker
    end;
    knutepunkt = record
        x,y: integer;
        utvei: veilistepeker;
        info: string[ 30 ];
        s: integer; {avstand fra et angitt knutepunkt}
        nærmeste: knutepunktpeker;
        {nærmeste nabo på korteste vei til knutepunkt Start}
    end;
var
    nettverk: record
        A: array[ 1..50 ] of knutepunkt;
        Np: integer; {antall knutepunkt i nettverket}
        Max: integer; {lengden på A}
        Start: knutepunktpeker
        {det punkt i nettverket minste vei er beregnet fra}
    end;
```

Figur 2.16: Nettverksstruktur.

```
procedure korteste-vei;
    {finner korteste vei fra knutepunkt Start til}
    {alle andre knutepunkter i nettverket}
var
    i: integer;
procedure rekursiv-korteste-vei(k: integer);
{rekursiv prosedyre}
var
    p: veilistepeker;
begin
01 with nettverk do begin
02      p := A[k].utvei;
03      while p <> nil do begin
04          if p↑.avstand < 0 then return
              {Negative avstander tillates ikke!};
05          else if (A[k].s + p↑.avstand < A[p↑.vei].s) then begin
                {naboen har fått er bedre tilbud}
06              A[p↑.vei].nærmeste := k;
07              A[p↑.vei].s := A[k].s + p↑.avstand;
08              rekursiv-korteste-vei(p↑.vei); {naboen vekkes}
09          end;
10          p := p↑.neste;
11      end;{while}
12 end;{with}
end; {rekursiv-korteste-vei}
begin {gir startverdier}
    with nettverk do begin
        for i := 1 to Np do begin
            A[i].s := ∞ ; {initierer en uendelig stor verdi}
            A[i].nærmeste := -1;
        end;
        A[start].s := 0;
        rekursiv-korteste-vei(Start);
    end;{with}
end;
```

Figur 2.17: Rekursiv prosedyre som finner korteste vei i et nettverk.

```
procedure skriv-korteste-vei;
    {skriver ut korteste vei fra Start til alle andre knutepunkter}
var
    i,j: integer;
begin
    with nettverk do begin
        for i := 1 to Np do begin
            j := i;
            writeln;
            while j <> -1 do begin
                write(j);
                if A[j].s = ∞ then feil; {kan ikke nås fra Start}
                j := A[j].nærmeste;
            end;{while}
        end;{do}
    end;{with}
end;
```

Figur 2.18: Prosedyre som skriver ut korteste vei.

$O(n^2)$ mens Floyd har tidskompleksiteten $O(n^3)$. Prisen man må betale for større generalitet er altså økt tidsforbruk.

Vi skal først beskrive en rekursiv korteste-vei-algoritme som er basert på datastrukturen i figur 2.16. Deretter skal Dijkstras algoritme skissere.

I figur 2.16 representeres knutepunktene som et 1-dimensjonalt array av cellegrupper (record). I knutepunktene etableres lister over utveier. Nodene i disse listene inneholder nummeret på det knutepunkt veien går til samt den tilhørende veilengde. Enveiskjørte veier representeres ved at de definerer utvei kun fra det ene knutepunktet. Toveiskjørte veier derimot, blir definert som utvei i begge knutepunkter.

Basert på denne datastrukturen finner vi i figur 2.17 en rekursiv kortesteveialgoritme. Algoritmen forutsetter at veilengdene er *positive* verdier, men toveiskjørte veier behøver ikke ha samme lengde i de to retninger. Negative verdier vil føre til at algoritmen definerer en uendelig løkke.

I knutepunktene er definert en variabel *nærmeste* som gir nærmeste nabopunkt langs korteste vei til det angitte startpunktet. Initielt settes *nærmeste* = ∞ .

Når startpunktet er valgt, settes dette punktets *nærmeste* = 0. Startpunktet får med dette et bedre tilbud (fra ∞ til 0) og vi vekker dets naboer for å se om dette kan føre til at naboene også vil få et bedre tilbud. Slik forsetter prosessen rekursivt inntil ingen knutepunkt blir tilbudt kortere vei fra startpunktet. Fordelen med algoritmen er at den gir en enkel implementasjon. Ulempen er den høye tidskompleksiteten.

I følge algoritmen kan en node bli "vekketflere ganger. Når node a_i har fått et bedre tilbud, vekker den sine naboer. Dette tilbudet til a_i trenger ikke være det

beste tilbudet a_i kan få. Derfor vil a_i på nytt vekke sine naboer når den får et bedre tilbud.

Vi antar at hver node har e utveier og at nettverket består av n noder. I et ugunstig tilfelle (ikke værste tilfelle!) vil node 1 sende en bølge av rekursive kall til alle andre noder i nettverket. Dette vil føre til at linjene 03-11 blir utført $e(n-1)$ ganger. Etter denne bølgen antar vi at minst en node i nettverket har fått løst sitt korteste vei problem. Videre antar vi at hver node i nettverket generer en serie rekursive kall av den typen vi fikk fra node 1 og at en ny node får løst sitt korteste vei problem etter hver bølge. Det totale antall ganger linjene 03-11 må gjennomløpes blir derfor $e(n-1)+e(n-2)+e(n-3)+\cdots+e(1)+e(0)$. Dette gir oss tidsforbruket $O(e \cdot n^2)$. Dersom alle noder er koblet til alle andre noder, blir $e = n$ og vi får $T = O(n^3)$. Vi vil understreke at vår analyse ikke gjelder det teoretisk ugunstigste tilfelle.

I beste fall vil vi besøke hver node bare én gang og tidsforbruket blir $O(e \cdot n)$. Et forsøk med n=7 og e=6 ga 100-366 gjennomløp av linjene 03-11, avhengig av hvor vi valgte startpunktet. Rekkefølgen til nodenes utveier var i dette tilfellet med hensikt valgt ugunstig.

Dersom vi lar a_i vente med å vekke sine naboer til den har fått sitt beste tilbud, vil vi redusere tidskompleksiteten til 2. grad. Dette er ideen bak *Dijkstras* algoritme. Det springende punkt er å fastslå når a_i har fått sitt beste tilbud. Dette løses på følgende måte:

Dijkstras algoritme Et sett B inneholder til enhver tid de noder i A som har fått beregnet sin korteste vei. For hvert steg adderes til B den av de gjenværende noder, node i settet $A - B$, som har den korteste avstand til startpunktet. Denne noden kaller vi w. Beregningen stanser når alle noder i A befinner seg i B. Initelt legges startnoden i B.

Dersom vi antar at alle veilengder er positive, vil vi alltid finne en korteste vei fra startpunktet til w som bare passere noder som befinner seg i B. Dersom det ikke var slik, måtte det finnes en node x i $A - B$ som har en kortere vei til startpunktet enn w. Dette er ikke tilfellet, fordi da skulle vi valgt x i stedet for w.

Forutsetningen for at dette resonnementet skal holde, er at alle veilengder er *positive*.

Dijkstras algoritme er lett å implemetere når det benyttes en nabomatrise. Tidsforbruket på matriseimplementasjonen er $O(n^2)$.

På datastrukturen vi har benyttet i vårt eksempel, lønner det seg å benytte en prioritetskø for å organisere nodene i $A - B$. Prioritetskøen kan implementeres som et delvis ordnet tre. I et delvis ordnet tre vil innsetting og sletting av minste verdi kunne utføres i $O(\log n)$ tid. Den node i $A - B$ som har kortest avstand til startpunktet, vil bli rot i treet. Når w er funnet, oppdateres $A[i].s$ til de av w sine

naboer som befinner seg i $A - B$. Deretter oppdateres prioritetskøen med de nye verdiene $A[i].s$. Hver oppdatering gjøres i $O(\log n)$ tid. Dersom vi i nettverket har $e \cdot n$ veier, vil den totale tid for å finne korteste vei bli $O(e \cdot n \cdot \log n)$. I det tilfellet at alle noder er koblet til alle andre noder, blir $e = n - 1$ og tidsforbruket blir $O(n^2 \cdot \log n)$.

Størrelsen e er følgelig den kritiske faktor når man skal vurdere matrise- eller listeimplementasjon av nettverket. Listeimplementasjon vil lønne seg dersom det går få veier ut fra hvert knutepunkt. For eksempel vil et gatenett i en by være et slikt tilfelle. I en by har vi mange gatekryss, men sjelden mer enn 4 veier ut fra hvert kryss.

2.7 Hash-metoder

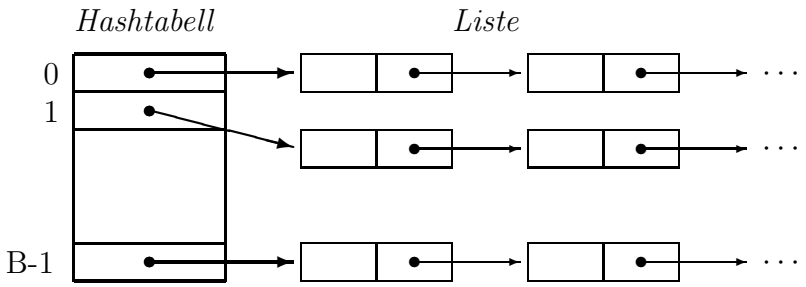
En viktig og mye brukt teknikk for å svare på spørsmål om medlemsskap i sett, er den såkalte *hashing*. På norsk benyttes av og til begrepene nøkkeltransformasjon og spredefunksjoner. Det finnes en rekke hashmetoder, men felles for alle er at de beregner maskinadresser på grunnlag nøkkelverdiene. Hashfunksjoner skrives ofte som: $a = h(k)$, hvor a er adressen, k er nøkkelen og h er hashfunksjonen. Anta at nøklene er personnummer i intervallet $[1, 1000]$ og at hashfunksjonen setter a lik det siste sifferet i personnummeret. Dette gir oss 10 mulige adresser $[0, 1, \cdots, 9]$.

Vi kan ta et annet eksempel. Anta at vi ringer til likningskontoret i Trondheim og spør om skatteberegningen. Fra sentralbordet blir vi spurt om fødselsmåned. Deretter blir vi satt over til en saksbehandler. Grunnen til det innledende spørsmålet er ganske klar. Fødselsmåned benyttes som et kriterium for å fordele arbeidet mellom de ulike saksbehandlere (finne en bestemt saksbehandeler i *settet* av seksbehandlere). Dersom vi hadde 12 saksbehandlere ved likningskontoret og hver saksbehandler ble tildelt folk som var født i en bestemt måned, er det god grunn til å anta at arbeidet ville bli noenlunde jevnt fordelt på saksbehandlerne. Dersom det skulle vise at befolkningen hadde en tendens til å bli født i bestemte deler av året, ville vårt fordelingsprinsipp føre til ujevn arbeidsdeling mellom saksbehandlerne. Dette illustrerer et generelt problem med hashing, nemlig at hashing er mest effektiv når man oppnår en uniform fordeling av nøklene i adresserommet.

Hashing kan utføres etter to prinsipper, åpen hashing og lukket hashing. Åpen hashing setter ingen grenser på størrelsen av datasettet mens lukket hashing benytter en begrenset lagerplass.

2.7.1 Åpen hashing

Figur 2.19 viser sprinsippet for åpen hashing. Hovedideen er at medlemme av det aktuelle datasettet A delles i et endelig antall klasser. Dersom vi ønsker B klasser, nummerert fra $0, 1, \cdots, B - 1$, må vi benytte en hashfunksjon h slik at vi for hvert element x_i i A har at $h(x_i)$ er et *heltall* i intervallet 0 til $B - 1$.



Figur 2.19: Datatsruktur for åpen hashing.

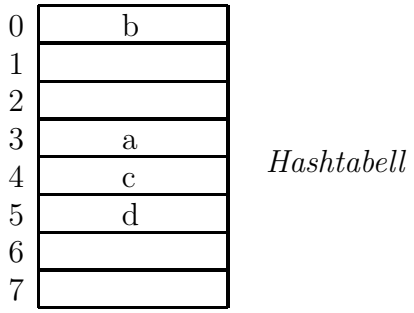
Klassene representeres ved arrayet *hashtabell* som holder pekere til listene av data. I prinsippet kan listene bli uendelig lange, men de må i praksis selvsagt holde seg innenfor maskinens maksimale lagerkapasitet.

Den tid det å spørre etter et element i A med en bestemt dataverdi, besvares ved å benytte elementets hashadresse som inngang i hashtabellen for deretter å søke i den tilhørende liste inntil elementets påtreffes. Dersom elementet ikke finnes i A, må vi søke helt til slutten av listen for å komme til konklusjonen: finnes ikke.

Som vi ser av dette eksemplet vil kostnadene ved søking i dataene være avhengig av hvor lange datalistene er.

2.7.2 Lukket hashing

Lukket hashing holder dataene i hashtabellen. Prinsippet er vist i figur 2.20.



Figur 2.20: Datatsruktur for lukket hashing.

Dersom det etter gjentatte innsettinger viser seg at en celle i hashtabellen går full, oppstår en kollisjon. En slik situasjon kan løses ved å benytte en alternativ hash-funksjon for å se om det er mulig å finne en ledig celle til det elementet som skapte kollisjonen.

2.7.3 Lineær hashing

W. Litwin [Lit80] beskriver en hashmetode som kalles *lineær hashing*. I lineær hashing kan adresseområdet vokse og avta dynamisk, vi har altså en metode for å utvide adresseområdet i takt med endringer i datamengden. Anta at vi har hashfunksjonen:

$$H_0\text{: } k \textbf{ mod } L$$

hvor k er nøkkelverdien, L antall adresser og **mod** en operator som gir resten ved heltallsdivisjon. For eksempel ved å velge $L = 5$ vil verdiområdet til H_0 være gitt ved settet $\{0, 1, 2, 3, 4, 5\}$. Dersom vi velger hashfunksjonen

$$H_1\text{: } k \textbf{ mod } 2L$$

får vi tilsvarende at H_1 har verdiområdet $\{0, 1, \cdots, 2L - 1\}$ som for L=6 gir verdiområdet $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$.

Litwin's lineære hashmetode går ut på at man benytter de to hashfunksjonene H_0 og H_1 og at man formulerer en strategi for når den ene eller den andre funksjonen skal benyttes. I denne strategien spiller en peker p en sentral rolle. Initielt settes p til å peke på det første elementet i hashtabellen. Dersom en eller annen celle i adressen går full, oppstår et overløpsproblem. Problemet løses i flere trinn. I første trinn lenkes en overløpsblokk til den cellen i hashtabellen som gikk full og de overskytende data lagres i overløpsblokken. I neste trinn benyttes H_1 på de nøkler som befinner seg i celle p. Deretter inkrementeres p med 1. Vi kan nå formulere strategien for å velge hashfunksjon.

> Vi gjør en innledende beregning med H_0. Dersom dette gir en adresse som er mindre enn p, skal H_1 benyttes for å finne hashadressen til vedkommende nøkkel, ellers skal H_0 benyttes.

Vi skal illustrere metoden ved et eksempel. Initielt starter vi med $L = 6$, $p = 0$ og en tom hashtabell. Videre forutsetter vi at det i hver celle i hashtabellen er plass til bare én nøkkel. Vi skal så sette nøklene $\{0, 4, 3, 6, 14, 7, 10\}$ inn i tabellen. Først gjøres en innledende beregning med H_0, som gir adressen 0. Siden dette gir oss en adresse $\geq p$, skal H_0 velges. Nøkkel 0 hashes deretter til adresse 0. Siden cellen i adresse 0 er ledig, settes nøkkel 0 inn her. Tilsvarende får vi for nøklene 4 og 3. Etter disse innsettingene ser hashtabellen slik ut:

peker	p					maks
adresse	0	1	2	3	4	5
nøkkel	0			3	4	

Når vi nå kommer til innsetting av 6, oppstår et overløpsproblem i hashtabellens celle 0. Først inkrementeres p til 1, men vi må da huske at H_1 nå skal anvendes på celleadresse p-1. Siden $p - 1$ i dette tilfellet er lik 0, betyr det at vi tilfeldigvis anvender H_1 på den cellen som har overløpsproblemet. Etter å ha anvendt H_1, ser tabellen slik ut:

peker		p					maks
adresse	0	1	2	3	4	5	6
nøkkel	0			3	4		6

Innsetting av 14 og 7 går uten overløpsproblem. Vi har nå følgende hashtabell:

peker		p					maks
adresse	0	1	2	3	4	5	6
nøkkel	0	7	14	3	4		6

Når 10 nå skal settes inn, oppstår på nytt et overløpsproblem, men denne gangen er det celle 4 som får problemet. Først inkrementeres p til 2 og så benyttes H_1 på celle 1. Dette gjør at nøkkel 7 må flyttes til celle 7. Deretter legges 10 inn i overløpsblokken til celle 4. Hashtabellen ser nå slik ut:

peker			p					maks
adresse	0	1	2	3	4	5	6	7
nøkkel	0		14	3	4		6	7
overløpsblokk					10			

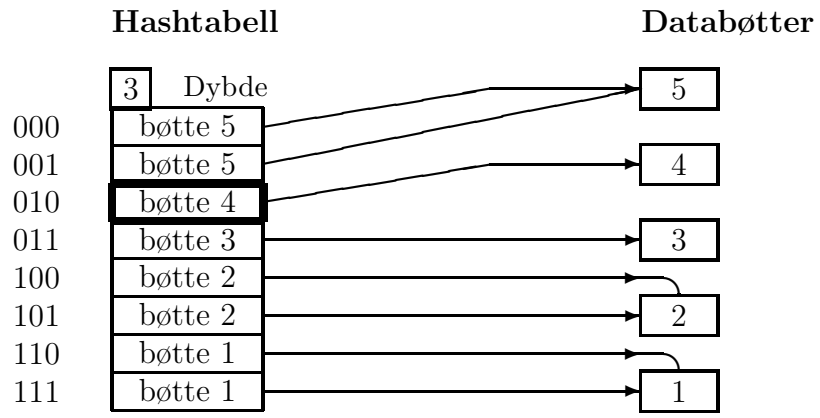
Som vi ser, fikk vi ikke løst overløpsproblemet til celle 4, men det er først når p er flyttet til celle 4 at denne cellen får løst sitt overløpsproblem. Da vil 10 gå til celle 10 og 4 vil gå til celle 4.

Vi skal vise et eksempel på en forespørsel. Finnes nøkkel 31 i hashtabellen? Den innledende beregningen med H_0 gir adresse 1. Siden $1 < p$ skal H_1 benyttes. Dette gir oss adressen 7, men der finnes ikke nøkkel 31. Svaret blir derfor nei.

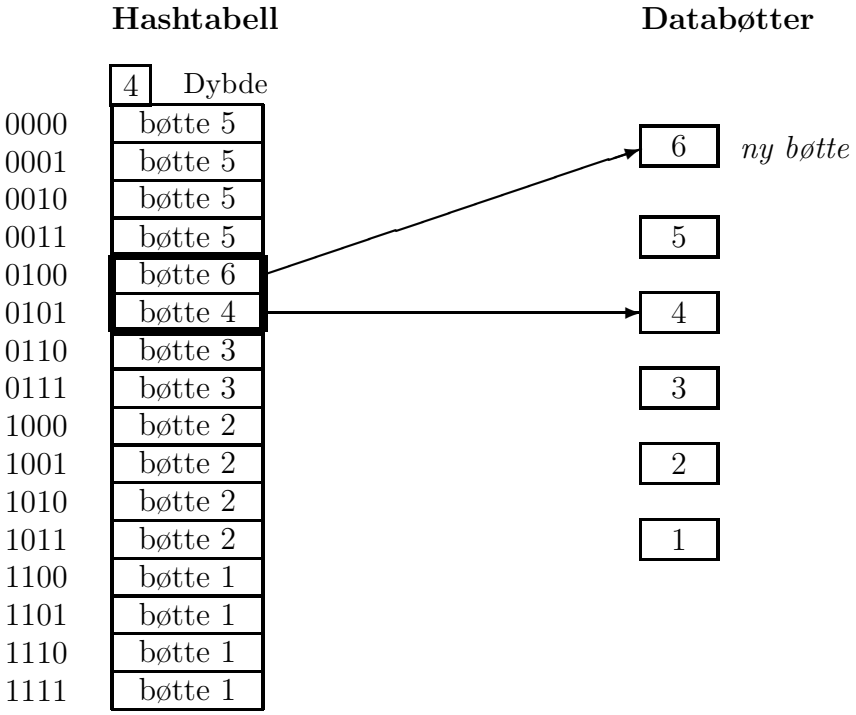
Vi skal merke oss at de nøkler H_0 mapper til celle i, vil H_1 mappe til cellene i og $L + i$. Dette gjør at adresseområdet vokser lineært etter hvert som p inkrementeres. Når p passerer $(L - 1)$, må vi flytte p tilbake til begynnelsen av tabellen og gå over til hashfunksjonene H_1 og H_2, som behandles på tilsvarende måte som H_0 og H_1.

2.7.4 Extendible hashing

Fagin beskriver i [FNPS79] en hashmetode han kaller *extendible hashing*. Metoden er utviklet for rask søking i data som ligger lagret på *platelager*. Extendible hashing benytter de tre elementene: *hashfunksjon, hashtabell* og *databøtte*. Data lagres ikke i hashtabellen, bare i databøttene. I hashtabellen finner vi derfor pekere til databøttene som vist i figur 2.21. Ideelt sett bør hashtabellen ligge i primærminnet, men dersom den blir svært stor, er det selvsagt mulig å legge den ut på platelageret. Deretter kan man laste inn i primærminnet mindre deler av tabellen, de deler som trengs for å finne de aktuelle databøtter. Størrelsen på databøttene avpasses etter platelagerets karakteristika. Et fornuftig valg kan være å la en databøtte vær lik platelagerets minste adresserbare enhet (den delen av en sektor som ligger innenfor et spor).



Databøtte 4 til tabellinngang 010 går full og tabellens lengde dobles:



Figur 2.21: Prinsippet for extendible hashing.

Hashfunksjonen ser på én-dimensjonale nøkler. Adressene finnes ved å tolke de d første biter (de mest signifikante) i nøkkelen som en integer. For eksempel dersom $d = 3$, vil nøkkelen 256 (100000000 binært) få hashadressen 100. Adresserommet's størrelse er i dette eksemplet $2^3 = 8$. Størrelsen d benevnes som hastabellens *dybde*. Ved å velge $d = 4$, vil adresserommet bli dobbelt så stort (16 adresser). Generelt har vi at ved å øke d til $d + 1$, vil lengden på adresserommet bli doblet. Dette forhold utnyttes til å løse overløpsproblemer i databøttene.

Man kan spørre seg om det er nødvendig å benytte en hashtabell med pekere til databøtter. Hvorfor beregnes ikke bøtteadressene direkte. Grunnen er at vi oppnår en bedre utnyttelse av kapasiteten til databøttene. Dersom vi for eksempel har få nøkler som starter på 000, 001 og 010, kan vi la alle disse nøklene få dele den samme databøtten om det er plass til det.

> Prinsippet for tilordning av databøtter går ut på at naboceller i hashtabellen deler samme databøtte så lenge plassen i bøtten tillater det. En celle i hashtabellen har ikke lov til å fordele dataene sine på flere databøtter.

Dersom en databøtte går full, lenkes ikke inn noen overløpsblokk. Extendible hashing er derfor en lukket hashing. Overløpsproblemet løses ved å gjøre hashtabellen lengere. For hver gang en databøtte går full, inkrementeres derfor d med 1, slik at lengden på hashtabellen blir doblet. Figure 2.21 viser en hashtabell med dybde 3. Etter noen innsettinger tenker vi oss at databøtten til hashadresse 010 går full. Ved at d endres til 4, vil hashadresse 010 bli splittet i to nye adresser: 0100 og 0101. Innholdet i databøtten til den gamle adressen 010, kan nå fordeles på hver av databøttene til 0100 og 0101 som vist i figur 2.21.

Kostnadene ved dobling av adresserommet er forholdsvis beskjedne, fordi vi ikke trenger flytte innholdet i databøttene. Det eneste vi trenger å gjøre, er å gå igjennom den nye tabellen og oppdatere pekere til databøttene og fordele datainnholdet til den bøtten som gikk full, på to nye bøtter.

2.7.5 Cellemetoden

En populær hash-funksjon i digitale kartografiske system, er den såkalte cellemetode. Metoden går ut på å beregne entydige indekser til ruter i xy-planet. Alle punkter som befinner seg innenfor en bestemt rute, vil få tilordnet den samme indeks. Rutene velges rektangulære, men det er ingen betingelse at de må være kvadratiske. I figur 2.22 er vist et eksempel på oppdeling av et plan med cellemetoden.

Vi kaller antall ruter (adresseringsområdet) for N og antall ruter i de to akseretninger N_x og N_y. Hashfunksjonen blir da:

$$h(x, y) = \left\lceil \frac{x - x_{min}}{S_x} \right\rceil + \left\lfloor \frac{y - y_{min}}{S_y} \right\rfloor \cdot N_x \tag{2.2}$$

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

Figur 2.22: Oppdeling av xy-planet med cellemetoden.

hvor

$$S_x = \frac{x_{max} - x_{min}}{N_x}$$

$$S_y = \frac{y_{max} - y_{min}}{N_y}$$

Dersom vi velger like mange ruter i begge akseretninger, blir

$$N_x = N_y = \left\lfloor \sqrt{N} \right\rfloor$$

En ulempe med cellemetoden er at den forutsetter at de geografiske objekter har en homogen fordeling i xy-planet. Dersom dette ikke er tilfellet, vil vi risikere at mange hashadresser ikke får tilordnet noen objekter mens andre adresser blir svært populære". Løsninger på de problem dette skaper, vil vi komme tilbake til i forbindelse med Excell og Grid-file.

2.7.6 Mortonindekser og bitfletting

En mye anvendt teknikk for å transformere fra flerdimensjonale nøkler til 1-dimensjonale nøkler er å beregne de såkalte *Mortonindekser*, ofte kalt *bitfletting*. Mortonindeksene ble opprinnelig anvendt for tranformasjon fra 2-dimensjonalt rom, men metoden kan lett generaliseres til n-dimensjoner. Transformasjonen gjøres ved å flette sammen bitstrengen for koordinatparet. Dette forutsetter at koordinatene er *positive heltall*.

Vi antar at vi har koordinater i et 2-dimensjonalt rom og at koordinatene er representert i en 16-biters streng. Bitfletting for koordinatparet (x, y) vil resultere i bitstrengen:

$$\mid x_0 \mid y_0 \mid x_1 \mid y_1 \mid x_2 \mid y_2 \mid \cdots \mid x_{14} \mid y_{14} \mid x_{15} \mid y_{15} \mid$$

Som vi ser er den nye bistrengen på 32 biter. Den inverse transformasjon, fra Mortonidekser til xy-koordinater, utføres lett ved å plukke annet hvert bit fra Mortonindeksen og sette de sammen til en x eller y-koordinat.

Figur 2.23 viser Mortonindeksene for xy-par i intervallet $[0, 7]$. Tallene med fet type er i titallssystemet mens de øvrige tall er i det binære tallsystem. Som vi ser av figuren, vil nærhet i xy-planet også gi en viss nærhet i indeks-rommet. Selvsagt gjelder ikke dette fullt ut, fordi vi har jo gått fra 2 dimensjoner til 1 dimensjon. For eksempel er rutene 26 og 48 naboer i xy-planet, men de ligger med en betydelig avstand i Mortonrommet. Derimot er blokkene 0,1,2 og 3 naboer både i

	000	001	010	011	100	101	110	111
	0	**2**	**8**	**10**	**32**	**34**	**40**	**42**
000	000000	000010	001000	001010	100000	100010	101000	101010
	1	**3**	**9**	**11**	**33**	**35**	**41**	**43**
001	000001	000011	001001	001011	100001	100011	101001	101011
	4	**6**	**12**	**14**	**36**	**38**	**44**	**46**
010	000100	000110	001100	001110	100100	100110	101100	
	5	**7**	**13**	**15**	**37**	**39**	**45**	**47**
011	000101	000111	001101	001111	100101	100111		
	16	**18**	**24**	**26**	**48**	**50**	**56**	**58**
100	010000	010010	011000		110000		111000	
	17	**19**	**25**	**27**	**49**	**51**	**57**	**59**
101	010001	010011	011001	011011				
	20	**22**	**28**	**30**	**52**	**54**	**60**	**62**
110	010100	010110	011100	011110	110100	110110	111100	111110
	21	**23**	**29**	**31**	**53**	**55**	**61**	**63**
111	010101	010111	011101	011111	110101	110111	111101	111111

Figur 2.23: Mortonindekser

xy-planet og i Mortonrommet. Ved å sortere rutene i figuren etter sine Mortonindekser, oppnår vi at naboskap i xy-rommet til en viss grad blir overført til naboskap i det 1-dimensjonale rom.

I motsetning til cellemetoden krever ikke beregning av Mortonindekser kjennskap til koordinatenes maksimalverdier.

2.8 Sortering

Sortering av data er en oppgave vi ofte blir stilt overfor. Formålet med sortering vil i regelen være at man ønsker å effektivisere ulike utvalgsoperasjoner som for eksempel: finn alle nøkkelverdier som ligger innenfor et bestemt intervall, undersøk om en gitt nøkkelverdi finnes i datasettet etc.

Vi skal beskrive noen standardalgoritmer for sortering. En av algoritmene gjelder sortering av så store datamengder at eksternt platelager må tas i bruk. De øvrige algoritmene forutsetter at dataene i sin helhet får plass i hurtiglageret.

Det er bare de viktigste metoder for sortering som her blitt tatt opp. En mere uttømmende framstilling av emnet finner vi for eksempel hos [AHU83].

2.8.1 Boblesortering

Den enkleste sorteringsmetode er kanskje boblesortering. Den grunnleggende ide bak algoritmen er forestillingen om at postene, som skal sorteres, blir holdt i et vertikalt

array og at postene med lave nøkkelverdier er *lette* og bobler opp til toppen av arrayet. Vi gjør gjentatte passeringer av arrayet, fra bunnen til toppen.

Dersom to naboelementer ikke er i korrekt rekkefølge, det vil si om det letteste er underst, bytter vi deres rekkefølge. Effekten av denne operasjonen er at etter første passering av arrayet, har den letteste posten boblet opp til toppen av arrayet. Etter andre passering vil den nest letteste posten ha boblet opp til andre posisjon. Slik fortsetter det inntil arrayet er gjennomløpt det nødvendige antall ganger. For det andre gjennomløpet behøver vi ikke boble opp til posisjon en, fordi vi vet at den laveste nøkkelen allerede befinner der. Generelt vil gjennomløp i ikke behøve å boble forbi posisjon i.

Algoritmen er skissert i figur 2.24.Den benytter prosedyren bytt.

Tidsforbruket til boblesortering er $O(n^2)$ både i værste fall og i gjennomsnitt, fordi testen i linje (3) alltid må utføres $n(n-1)/2$ ganger.

For små datasett (mindre enn 100 elementer) som skal sorteres, kan enkle algoritmer som boblesortering være akseptable, men for større datasett er det påkrevet å benytte algoritmer med en snillere tidskompleksitet.

2.8.2 Quicksort

Quicksort er en mye benyttet algoritme for sortering av store datasett. Tidskompleksiteten til quicksort er i gjennomsnitt $O(n \cdot \log n)$. Blant de hittil kjente $O(n \cdot \log n)$ sorterings algoritmer er quicksort den raskeste.

Quicksort er en rekursiv algoritme som opererer på et array A med elementene $A[1], ..., A[n]$. Ideen er at man deler arrayet i subarray (intervall) og foretar en delvis sortering av intervallene omkring en nøkkelverdi som ligger nærmest en valgt splittverdi. Ideelt sett ønsker man å velge det aktuelle intervallets median som splittverdi, men som vi senere skal se er kostnadene ved å finne medianen så store at man i praksis ofte velger noe mindre gunstige splittverdier (til lavere kostnad).

Alle nøkler mindre enn den valgte splittverdi, flyttes til venstre i intervallet, mens de øvrige nøkler flyttes til høyre i intervallet. Dette gjøres ved å la nøklene bytte plass inntil alle har havnet på rett side av splittverdien. Deretter gjentas prosedyren for de to halvdeler av intervallet. Slik fortsetter rekursjonen inntil arrayet er sortert. Initielt starter man med et intervall som spenner over hele arrayet.

Et eksempel på bruk av quicksort er vist i figur 2.25. Vær oppmerksom på at man i eksemplet benytter en primitiv teknikk for å finne et *estimat for medianen*, se funksjon finn-splittverdi i figur 2.26.

Algoritmens effektivitet er avhengig av hvor godt splittverdiene halverer sine intervall.

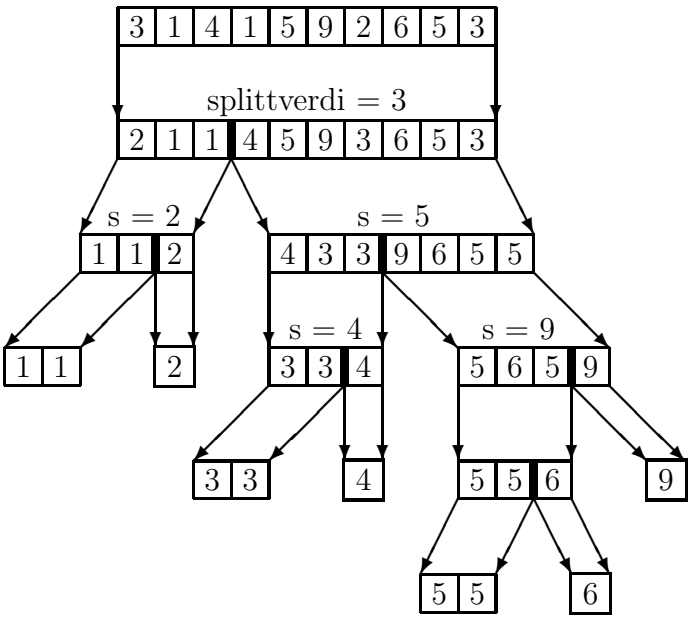
Et søk etter medianen krever at hele intervallet må gjennomløpes. Derfor velges ofte en noe enklere framgangsmåte ved at man plukker en vilkårlig nøkkel fra intervallet og benytter denne som splittverdi. I noen tilfeller vil dette gi en skjev deling av intervallet, mens i andre tilfeller er man noe mere heldig og oppnår en halvering av intervallet. I værste fall vil man alltid være maksimalt uheldig og plukke ut

```
procedure bytt (var x,y: recordtype)
{bytter verdiene til x og y}
var
    temp: recordtype;
begin
    temp := x;
    x := y;
    y := temp;
end; {bytt}



type
    recordtype = record
        nøkkel: nøkkeltype;
        info: infotype
    end;
⋮
var
    A: array [ 1..N ] of recordtype;
⋮
{Kjernen i boblesortering}
(1)     for i := 1 to n-1 do
(2)         for j := n downto i+1 do
(3)             if A [ j ].nøkkel < A [ j-1 ].nøkkel then
(4)                 bytt(A [ j ],A [ j-1 ])
⋮
```

Figur 2.24: Algoritmen boblesortering



Figur 2.25: Quicksort av et datasett.

splittverdier som deler intervallet i henholdsvis (n-1) og (1) elementer. Derfor vil vi i værste fall få en sortering i $O(n^2)$ tid. Sannsynligheten for denne situasjonen er meget liten, og det lar seg vise at selv med tilfeldig valg av splittverdi, vil quicksort i gjennomsnitt ha et tidsforbruk på $O(n \cdot \log n)$.

Et alternativ til et eksakt søk etter medianen, er å estimere medianen blant et utvalg på k nøkler av intervallets n nøkler. Intuitivt virker det fornuftig å tilpasse størrelsen på k til intervallets lengde. En interessant øvelse er å bestemme den beste verdi til k, som en funksjon av lengden n til det intervall som skal sorteres.

Vi skal gi en detaljert algoritmisk beskrivelse av quicksort. Det faller her naturlig å definere finn-splittverdi og delvis-sorter som funksjoner. Quicksort definerer vi som en prosedyre. Funksjonen finn-splittverdi har vi valgt å implementere slik at den plukker ut splittverdien på en vilkårlig måte. De tre algoritmene er vist i figurene 2.26 og 2.27.

Det er mulig å forbedre quicksort ved å studere hva som skjer når intervallene blir små. Det viser seg nemlig at algoritmer som tilhører $O(n^2)$ klassen, er raskere enn quicksort når intervallene blir små nok. Enkelte forfattere hevder at når intervallene får en lengde mindre enn 9, bør quicksort kalle en enklere sorteringsalgoritme, for eksempel boblesortering.

```
function finn-splittverdi (i,j: integer): integer;
    {returnerer 0 dersom A[i],...,A[j] har identiske nøkler}
    {ellers returneres indeksen til den største av de to nøkler}
    {som ligger lengst til venstre}
    var
        førstenøkkel: nøkkeltype;
        k: integer; {løper fra venstre mot høyre og ser etter
        ulike nøkler}
    begin
        førstenøkkel := A[i].nøkkel;
        for k := i+1 to j do {ser etter ulike nøkler}
            if A[k].nøkkel > førstenøkkel then {velg den største
            nøkkel}
                return (k);
            else if A[k].nøkkel < førstenøkkel then
                return (i);
        return (0) {ulike nøkler ble aldri funnet}
end; {finn-splittverdi}


function delvis-sorter (i,j: integer; splittverdi: nøkkeltype): integer;
    {deler A[i],...,A[j] slik at nøkler < splittverdi kommer til venstre}
    {og nøkler ≥ splittverdi kommer til høyre. Returnerer begynnel-}
    {sen på gruppen til høyre.}
    var
        l,r: integer; {markører som starter i hver sin ende av arrayet
        og flyttes mot hverandre.}
    begin
        repeat
            bytt(A[l],A[r];
            {nå begynner flyttingen av markørene}
            while A[l].nøkkel < splittverdi do
                l := l+1
            while A[r].nøkkel ≥ splittverdi do
                r := r-1
        until
            l > r;
    return (l);
end; {delvis-sorter}
```

Figur 2.26: Funksjonene finn-splittverdi og delvis-sorter.

```
procedure quicksort (i,j: integer): integer;
    {sorterer elementene A[i],...,A[j] til arrayet A}
    var
        splittverdi: nøkkeltype; {splittverdien}
        splittindeks: integer; {indeksen til det element i A som har
        en nøkkelverdi lik splittverdi}
        k: integer; {starten på gruppen av elementer ≥ splittverdi}
    begin
        splittindeks := finn-splittverdi(i,j);
        if splittindeks <> 0 then begin {ikke gjør noe dersom alle
        nøkler er like}
            splittverdi := A[splittindeks].nøkkel;
            k := delvis-sorter(i,j,splittverdi);
            quicksort(i,k-1); {rekursive prosedyrekall}
            quicksort(k,j);
    end
end; {delvis-sorter}
```

Figur 2.27: Prosedyren quicksort.

2.8.3 Flette-sortering

Flette-sortering benyttes for sortering av data som er lagret i et eksternt minne (platelager).

Et internt minne er langt raskere enn et eksternt minne, derfor vil bruk av eksterne minne representere en flaskehals for tidsforbruket. For eksempel, anta at vi har en disk som roterer med en hastighet på 20 ms og at det er plass til 1000 heltall pr. minste adresserbare enhet (blokk). Den tid det tar å hente fram en blokk vil være summen av den tid det tar å posisjonere lesehodet til rett spor og den tid det tar å vente på at den aktuelle blokken (sektoren) har passert lesehodet. I gjennomsnitt må vi vente 1/2 omdreining på at rett sektor skal komme under lesehodet. Vi antar at posisjoneringstida for lesehodet er 30 ms. Når vi adderer den tida det tar for sektoren å passere lesehodet, kommer vi til at det i gjennomsnitt vil ta noe i overkant av 40 ms å hente fram blokken. Dette er nok tid til å gjøre enkle behandlinger av de tusen heltallene om de lå i primærminnet. Vi ville kanskje hatt tilstrekkelig tid til å kjøre quicksort på tallene. Konklusjonen blir at man må minimalisere lesing og skriving mot sekundærlageret ved at primærminnet utnyttes så langt det er mulig.

Den grunnleggende ide bak flette-sortering er at vi organiserer en fil i økende seksjoner og hele tiden sørger for at dataene innenfor hver seksjon holdes sortert.

Vi starter med å fordele vår n data på to filer f_1 og f_2. Disse filene organiseres i seksjoner med lengde k, og datene innenfor seksjonene sorteres. Det er nå en enkel oppgave å flette de to filene sammen slik at vi får to nye filer g_1 og g_2 med

28	3	93	10	54	65	30	90	10	69	8	22
31	5	96	40	85	9	39	13	8	77	10	

(a) initielle filer

28	31	93	96	54	85	30	39	8	10	8	10
3	5	10	40	9	65	13	90	69	77	22	

(b) organisert i seksjoner med lengde 2

3	5	28	31	9	54	65	85	8	10	69	77
10	40	93	96	13	30	39	90	8	10	22	

(c) organisert i seksjoner med lengde 4

3	5	10	28	31	40	93	96	8	8	10	10	22	69	77
9	13	30	39	54	65	85	90							

(d) organisert i seksjoner med lengde 8

3	5	9	10	13	28	30	31	39	40	54	65	85	90	93	96
8	8	10	10	22	69	77									

(e) organisert i seksjoner med lengde 16

3	5	8	8	9	10	10	10	13	22	28	30	31	39	40	54	65	69	77	85	90	93	96									

(f) organisert i seksjoner med lengde 32

Figur 2.28: Flette-sortering av en liste.

seksjonslengde $2k$. Deretter tømmes filene f_1 og f_2 og så flettes g_1 og g_2 på f_1 og f_2, men nå med seksjonslengde $4k$. Slik alterneres mellom f- og g-filene inntil samtlige data får plass i en enkelt seksjon. Metoden er vist ved et eksempel i figur 2.28

Dersom vi starter med en seksjonslengde $k = 1$, blir antall passeringer h av det omtalte slaget lik $O(\log n)$. Ved å starte med en seksjonslengde større enn 1, kan vi beregne h av uttrykket $k \cdot 2^h = n$. Dette gir oss

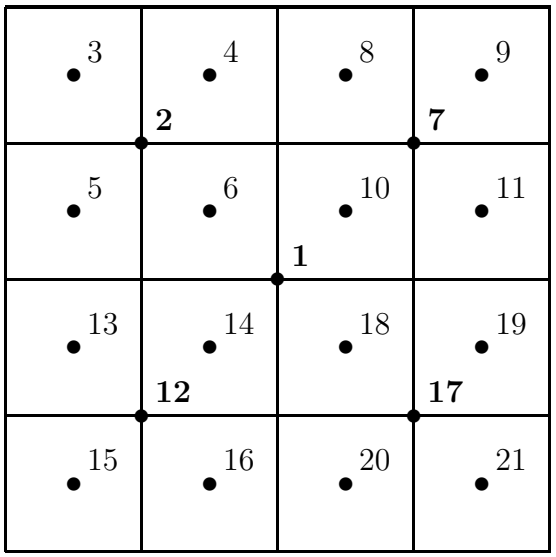
$$h = O(\log_2 \frac{n}{k})$$

2.8.4 Ordning av romlige data

Den ordning (sortering) vi hittil har behandlet, har forutsatt endimensjonale nøkler. En geografisk posisjon derimot, vil ofte være gitt i et flerdimensjonalt rom. Det å ordne romlige data i en eller annen rekkefølge, innebærer en transformasjon fra et flerdimensjonalt rom til et endimensjonalt rom. Vi skal i det etterfølgende se på noen slike teknikker.

Hierarkisk-sortering

Hierarkisk-sortering ligner på quicksort, men med den forskjell at sorteringen er utvidet til k-dimensjoner. Bakgrunnen for metoden er at den kan benyttes til å etablere balanserte punkt-kvadtre (kvadtrær vil bli forklart senere). Begrepet hierarkisk-sortering er innført av forfatteren. Metoden er første gang beskrevet av [Bjø88]. Algoritmen er vist i figur 2.30.

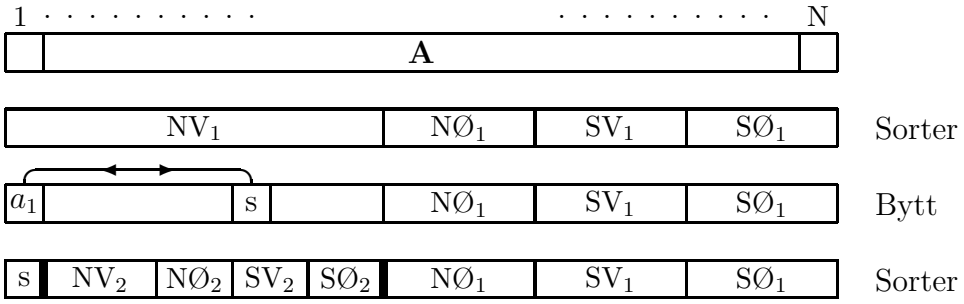


Figur 2.29: Hierarkisk-sortering av punkter i et plan.

Metoden går ut på å splitte rekursivt et sett av datapunkter i fire like store deler inntil samtlige subsett inneholder bare ett punkt. Kriteriet for splittingen baseres på punktenes xy-koordinater. Splittlinjene er rette linjer som ligger parallelt med aksesystemet. For å definere en splittlinje i hver av de to akseretninger er det tilstrekkelig å oppgi koordinatene til deres skjæringspunkt (splittpunkt). Det forlanges at man som splittpunkt benytter punkter som befinner seg i det datasettet som skal sorteres.

I figur 2.29 er gitt et eksempel på sortering av punkter med hierarkisk-sortering. Etter sortering vil punktene komme i stigende nummer-rekkefølge.

Funksjonen *splittpunkt* returnerer indeksen til splittpunktet. Splittpunktet kan finnes ved at man først beregner tyngdepunktet i datasettet og deretter finner indeksen til det punktet som ligger nærmest tyngdepunktet. Denne framgangsmåten vil ikke garantere at man alltid finner det punktet som deler datasettet i fire like store deler, men metoden vil gi et godt estimat. På grunn av at datasettet deles etter de fire kvadranter nv,nø,sv og nø, vil det forekomme tilfeller hvor det ikke er mulig å finne et splittpunkt som deler datasettet i nøyaktig i fire like store subsett (for eksempel om alle punktene ligger langs en rett linje).




```
type
    recordtype = record
        x,y : koordinattype;
    end;
⋮
var
    A: array [ 1..N ] of recordtype;
⋮
procedure hierarkisk-sortering (i,j: integer): integer;
    {sorterer A [ i ],...,A [ j ] slik at punktene kommer i rekkefølgen:}
    {NordVest, NordØst, Sydvest, SydØst i forhold til et valgt}
    {splitttpunkt.}
    var
        p: integer; {splittpunktets indeks}
        nø,nv,sø,sv: integer; {markører til første punkt i hver
        av de fire kvadranter}
    begin
        if (i > j) then return ; {intervallet er tomt eller inneholder
        bare ett element}
        p := splittpunkt(i,j);
        bytt(A [ i ],A [ p ];
        sorter-etter-de-fire-kvadranter(i,j,nv,nø,sv,sø);
        hierarkisk-sortering(nv,nø-1);
        hierarkisk-sortering(nø,sv-1);
        hierarkisk-sortering(sv,sø-1);
        hierarkisk-sortering(sø,j);
end; {hierarkisk-sortering}
```

Figur 2.30: Prosedyren hierarkisk-sortering.

Procedyren sorter-etter-de-fire-kvandranter implementeres på tilsvarende måte som funksjonen delvis-sorter i figur 2.26. Siden vi sørger for at splittpunktet i intervall (i, j) ligger i element A [i], behøver vi ikke oppgi indeksen til splittpunktet i prosedyrekallet. Ved å basere oss på funksjonen delvis-sorter, får vi følgende algoritme:

```
procedure sorter-etter-de-fire-kvadranter (
              i,j: integer; var nv,nø,sv,sø: integer);
    begin
        nv := i-1;
        sv := delvis-sorter(nv,j,A [ i ].y);
              {deler intervallet langs en horisontal akse}
        nø := delvis-sorter(nv,sv-1,A [ i ].x);
        sø := delvis-sorter(sv,j,A [ i ].x);
              {deler nord- og sydintervallene vertikalt}
    end;
```

En analyse av tidsforbruket til hierarkisk-sortering kan følge tilsvarende framgangsmåte som for quicksort. I gjennomsnitt antas det at algoritmen vil ha en tidskompleksitet $O(n \cdot \log n)$.

Peano-kurver

Ved å transformere fra det n-dimensjonale rom til det 1-dimensjonale rom, vil vi kunne anvende de vanlige sorteringsalgoritmer på de transformerte nøklene. Det finnes selvsagt en rekke muligheter for å velge slike transformasjoner, men for romlig databehandling er det ofte et poeng at nærhet i det n-dimensjonale rom også gir nærhet i det 1-dimensjonale rom.

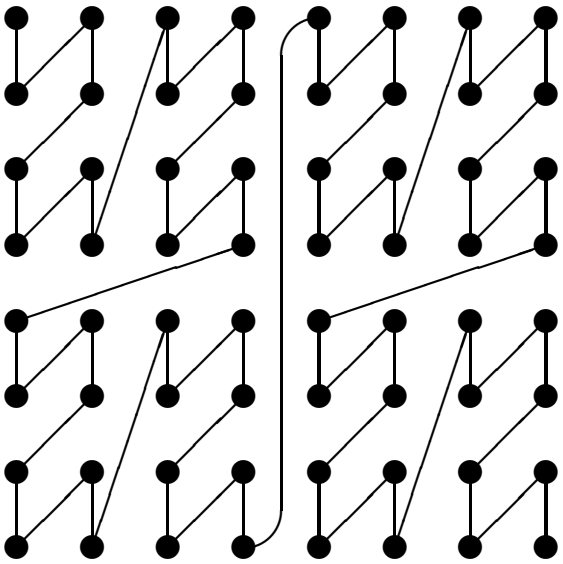
En mye anvendt teknikk for dette formål er transformasjon til *Mortonindekser*.

Matematikeren Giuseppe Peano oppdaget i 1890 at et n-dimensjonalt rom kunne transformeres til en linje. Figur 2.31 viser forholdet mellom Mortonindekser og Z-formet Peano kurve. Vi har hentet Mortonindeksene fra figur 2.23 og forbundet sirklene i figur 2.31 etter stigende indeks. Dette gir oss den såkalte Mortonorden.

Som vi ser gir dette oss en rekursiv **Z**-bevegelse (i figuren riktignok en speilvendt N). Fire Z-er på laveste nivå setter seg sammen til en ny Z på nivået over osv. . Sagt på en annen måte: man traverserer alle punkter innenfor en kvadrant før en ny kvadrant påbegynnes. Den Z-formede Peanokurve ivaretar noe av den romlige nærhet mellom objektene.

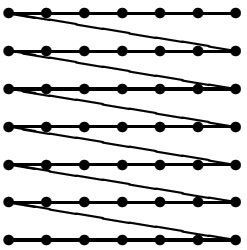
En annen peanokurve er den såkalte rad-ordning (eng. scan-raster order) som vist i figur 2.32. Av figur 2.22 framkommer det at cellemetoden gir radordning av rutene i xy-planet.

Det finnes også andre metoder for å ordne romlige data. Av kjente teknikker kan nevnes: spiralorden, Cantor-diagonalorden og Peano-Hilbertorden. Den mest interessante metode er nok likevel Mortonorden. Dette på grunn av sin enkle konvertering



Figur 2.31: Z-formet peanokurve (Mortonorden).

mellom det n-dimensjonale rom og nøkkelrommet, samt sin nære forbindelse til kvadtrær.



Figur 2.32: Radorden (raster-scan order).

Kapittel 3

TOPOLOGI

Innenfor geodatabaser og geografiske informasjonssystemer har man stor nytte av å anvende begrepsdannelser innenfor den matematiske topologi. Dette kapitlet gir først en innføring i grunnleggende begreper innen mengdelæren og topologien. Deretter blir dette matematiske grunnlaget utnyttet til å beskrive og utdype romlige relasjoner mellom geografiske objekter.

Topologi er et stort og sentralt område av matematikken. Den har oppstått ved en gradvis abstraksjonsprosess ut fra en lang rekke begrepsdannelser og et stort eksempelmateriale innenfor analyse, funksjonsteori og geometri.

Det finnes er rekke lærebøker innen emnet. Som utfyllende litteratur til dette kapitlet vil jeg spesielt anbefale [Sch79], [HR90] og [Lip65]. Av disse tre bøkene er [Lip65] den som er lettest tilgjengelig. Andre anbefalinger er [Gem67], [Bla82], de to første kapitlene i [Hus77] og avsnittene i [Mas91] om triangulering av mangfoldigheter. Det kan forøvrig nevnes at [Sch79] gir en forklaring på Peanokurver som vi husker fra lineære kvadtrær.

Enkelte av de kommende seksjoner har en typisk matematisk stil. For å gjøre stoffet lettere tilgjengelig er det laget en rekke eksempler og figurer. Teksten er imidlertid ment å kunne leses på flere nivåer. Dette understøttes ved at definisjoner, teorem, bevis og eksempler er tydelig merket. De som ønsker en viss innføring i emnet, kan konsentrere seg om eksemplene, figurene og de deler av teksten som diskuterer behovet for å anvende det topologiske begrepsapparatet innen GIS. På dette nivået kan alle bevis og teoremer hoppes over. Teoremer og bevis er i første rekke ment for systemutviklere og studenter som ønsker en grunndig behandling av emnet.

Den grunnleggende matematikk til dette kapitlet, er stort sett hentet fra [Sch79].

3.1 Noen begreper fra mengdelæren

Innen mengdelæren har begrepet *mengde* fått en sentral plass. Med mengde menes en samling objekter, til vanlig kalt *elementer*, som kan være virkelige eller tenkte, men

i alle tilfeller entydig definerte. Oftest er elementene tall, symboler for tall, punkter eller linjer. Som grunnlegger for mengdelæren regnes den tyske matematiker Georg Cantor (1845-1918).

$\{\}$ Mengde betegnes vanligvis med store bokstaver A, B, C osv.. De tilhørende elementer blir i regelen stilt etter hverandre innenfor en klammeparantes $\{\}$, i mengdelæren kalt mengdeparantes. Eksempel: $A = \{3, 4, 5, 10\}$ som leses: A er mengden av tallene 3,4,5 og 10.

$\in$, $\notin$ Ønsker man å gi uttrykk for at x er element i A benyttes elementsymbolet $\in$, som leses *er element i* og skrives $x \in A$. Dersom x ikke er element i A, skrives $x \notin A$.

$\{\cdots \mid \cdots\}$ Ofte oppgis ikke elementene i mengder direkte, men man innfører i stedet innenfor mengdeparantesen en variabel og bak denne en loddrett strek som varsler at det etterfølgende er en forklaring på hva den variable står for. Symbolet $\{\cdots \mid \cdots\}$ kalles *mengdebygger*. For eksempel kan mengden av alle reelle tall i det åpne intervallet 1 til 9 skrives som $A = \{x \in R \mid 1 < x < 9\}$.

$\subset$, $\subseteq$ En mengde A er en delmengde av mengden B dersom alle elementer i A også er elementer i B. Dersom vi skriver $A \subset B$, utelukkes at $A = B$ og vi sier at A er en *ekte delmengde* av B. Dersom vi ikke utelukker at $A = B$ skrives $A \subseteq B$.

$\emptyset$ En mengde uten elementer kalles *den tomme mengde* og skrives $\emptyset$. Merk at $\{0\} \neq \emptyset$, men en mengde med ett element, nemlig null.

$\cap$ Med *snittet* av to mengder A og B menes alle elementer som tilhører både A og B. Denne mengden kalles for *snittmengden* eller *fellesmengden* til A og B. Symbolet for snitt er $\cap$. Snittet av A og B defineres som:
$A \cap B = \{x \mid x \in A \ \ og \ \ x \in B\}$

$\cup$ Med *unionen* av to mengder A og B menes mengden av alle elementer som tilhører enten A eller B eller både A og B. Symbolet for union (forening) er $\cup$. Unionen (foreningsmengden) til A og B defineres som:
$A \cup B = \{x \mid x \in A \ \ eller \ \ x \in B\}$

$\setminus$ Med *differansmengden* mellom to mengder A og B menes mengden av alle elementer som hører til A, men ikke til B. Symbolet for mengdedifferens er $\setminus$ (leses minus). Differensmengden mellom A og B defineres som:
$A \setminus B = \{x \mid x \in A \ \ og \ \ x \notin B\}$

$A^c = G \setminus A$ Med *komplementmengden* til en delmengde A når grunnmengden er G, menes mengden av alle elementer i G som ikke hører til A. Dette skrives som:
$A^c = G \setminus A = \{x \mid x \in G \ \ og \ \ x \notin A\}$

$\times$ Med *produktmengden* av A og B (les A kryss B) menes mengden av alle ordnede par (x, y) slik at x hører til A og y hører til B. Dette kan skrives som: $A \times B = \{(x, y) \mid x \in A \ \ og \ \ y \in B\}$ og kalles gjerne *det kartesiske produktet* til A og B. Eksempel: $A = \{1, 5\}$ og $B = \{3, 7, 9\}$.
Her er $A \times B = \{(1, 3), (1, 7), (1, 9), (5, 3), (5, 7), (5, 9)\}$.

$\wedge$ brukes i stedet for ordet *og* (både og). Eksempel:
$A \cap B = \{x \mid x \in A \wedge x \in B\}$

$\vee$ brukes i stedet for ordet *eller* (enten-eller). Eksempel:
$A \cup B = \{x \mid x \in A \vee x \in B\}$

$\neg$ kalles negasjon og brukes i stedet for ordet *ikke*.

$\Rightarrow$ kalles implikasjonspil og brukes i stedet for ord som *medfører, herav følger*.

$\Leftrightarrow$ kalles ekvivalenspil og brukes i stedet for ordet *ekvivalent med*. Eksempel:
$x \in A \cap B \Leftrightarrow (x \in A) \wedge (x \in B)$

$\forall$ kalles allkvantor og brukes i stedet for ordet *for alle .... gjelder at*.

$\exists$ kalles eksistenskvantor og brukes i stedet for ordet *eksisterer en .... slik at*.

disjunkte mengder To mengder A og B sies å være disjunkte dersom $A \cap B = \emptyset$.

Relasjon En *relasjon* R i en mengde A er en undermengde i produktmengden $A \times A$. Dersom $(a, b) \in R$, skriver vi aRb og sier at b (en av mange elementer i A) er $R-$relatert til a.

Eksempel: La A være mengden av alle mennesker med norsk statsborgerskap, videre la $a \in A$ og $b \in A$ og la R være en bror-relasjon. Relasjonen aRb er da mengden av alle par (a, b) hvor bror-relasjonen er sann, altså mengden av norske statsborgere som har brødre samt deres brødre.

Eksempel: Andre relasjoner er: $a < b$ eller $a > b$ eller $a \neq b$ eller $a = b$.

Innenfor databaseteknologien har relasjonsmodellen i den senere tid fått stor utbredelse. Modellen henter sitt teoretiske grunnlag fra mengdelæren og er derfor godt forankret i matematisk teori.

Relasjoner har tre viktige egenskaper. En relasjon R i en mengde A er:
1. $refleksiv$ dersom aRa er sann for alle $a \in A$. Eksempel: Si at et universitet er inndelt i avdelinger. Alle studenter ved universitetet tilhører derfor en avdeling.
2. $symmetrisk$ hvis aRb medfører bRa for alle $a, b \in A$
3. $transitiv$ hvis aRb og bRc medfører aRc for alle $a, b, c \in A$

Funksjon (avbildning) En regel som til et gitt element i mengde A tilordner ett og bare ett element i mengde B, kalles en funksjon (avbildning). En funksjon f fra mengde A til mengde B skrives $f : A \to B$ og kan defineres som det kartesiske produkt $A \times B$ slik at for hver $a \in A$ finnes det en unik $b \in B$ med $(a, b) \in f$. Vi kaller A *definisjonsmengden* (domenet) til f og B *verdimengden* (billedmengden) til f. Dersom $(a, b) \in f$, skriver vi $f(a) = b$. Merk at kravet til unikhet skiller en funksjon fra en relasjon. I en relasjon har vi at $a \in A$ kan relateres til mange b-er i B, men det er bare en $f(a)$ når f er en funksjon.

Surjektiv (på) Dersom f er en funksjon fra A til B og billedområdet til A er B, er f surjektiv og vi sier at f avbilder A *på* B. Sagt på en annen måte $f : A \to B$ er surjektiv dersom alle elementer i B kan forklares ved en avbildning fra A, altså dersom $f(A) = B$.

Injektiv (en-entydig) Dersom $f(x) = f(y) \Rightarrow x = y$, er f injektivk, en-entydig.

Bijektiv En funksjon som er både surjektiv (på) og injektiv (en-entydig), sies å være bijektiv.

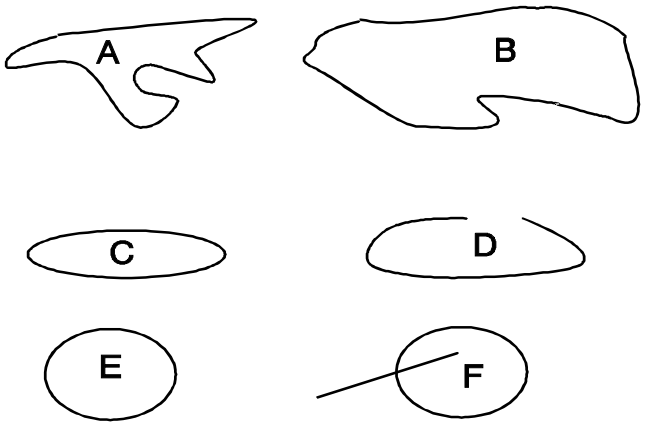
3.2 Topologisk rom og kontinuerlig avbildning

Innefor topologien har man de to hovedbegreper *topologisk rom* og *kontinuerlig avbildning*. Et topologisk rom defineres som en mengde M hvor hvert element p i M har et tilordnet system av delmengder i M, som kalles *omegner* til p og som oppfyller noen enkle presise betingelser (aksiomer for omegner). Om vi for eksempel lar M være et plan og p et punkt i dette planet, vil delmengden som består av alle punkter i M som har en avstand fra p som er mindre enn en gitt avstand δ større enn null, være et typisk eksempel på en omegn til p.

Vi sier at en avbildning f fra et topologisk rom M_1 inn i et topologisk rom M_2 er kontinuerlig i punktet p dersom det til enhver omegn O_2 til $f(p)$ i M_2 finnes en omegn O_1 til p i M_1 slik at O_1 avbildes inn i O_2 ved f.

To topologiske rom M_1 og M_2 sies å være topologisk ekvivalente eller *homeomorfe* hvis det eksisterer en en-entydig avbildning f av M_1 på M_2 slik at både f og dens inverse avbildning f^{-1} er kontinuerlige.

Eksempel: To geometriske objekter A og B er homeomorfe, dersom vi kan avlede det ene objektet fra det andre ved å tøye, krympe, rotere eller forskyve et av objektene. Så lenge man kan forme det ene objektet fra det andre uten å måtte ty til splitt eller klipping, er objektene homeomorfre. Figur 3.1 viser noen eksempler på homeomorfe og ikke-homeomorfe figurer.



Figurene A,B,C og E er homeomorfe.

D og F er ikke homeomorfe med hverandre eller med noen av de andre figurene.

Figur 3.1: Eksempler på homeomorfe rom.

3.3 Metrisk rom og metrisk topologi

Definisjon 1 *En metrikk på mengden X er en funksjon d fra $X \times X$ inn i de reelle tall R, som tilfredsstiller:*

1. $d(x, y) \geq 0$ *for alle* $x, y \in X$;
2. $d(x, y) = 0$ *hvis og bare hvis* $x = y$ *for* $x, y \in X$;
3. $d(x, y) = d(y, x)$ *for alle* $x, y \in X$;
4. $d(x, z) \leq d(x, y) + d(y, z)$ *for alle* $x, y, z \in X$;

Innen geometrien benyttes ofte de såkalte Evklidiske (eller Eu..) avstandsfunksjoner. Disse avstandsfunksjonene tilfredsstiller kravet til en metrikk.

Eksempel: La X være de reelle tall R og $d(x, y) =\mid x - y \mid$. Her er d en Evklidisk avstandsfunksjon i et endimensjonalt rom.

Eksempel: La X være lik planet R^2 og

$$d((x_1, x_2), (y_1, y_2)) = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2}$$

I dette tilfellet er d en Evklidisk avstandsfunksjon i et 2-dimensjonalt rom.

Den Evklidiske metrikk er den mest anvendte metrikk innen GIS-anvendelser, men supremumsmetrikken og Manhattanmetrikken er også aktuelle. Figur 3.2 illustrerer disse metrikkene. Alle punkter som ligger på omrisset til en av figurene, har samme avstand fra figuren's sentralpunkt (markert med en svart prikk).

I engelsk GIS litteratur benevnes ofte Manhattanmetrikken med taxicab metriceller "city block distance"mens supremumsmetrikken gjerne kalles "chessboard metriceller "max metric".

Manhattanmetrikken definerer avstanden mellom to punkter som summen av koordinatdifferanser langs koordinataksene. Ulempen med dette er at avstanden ikke

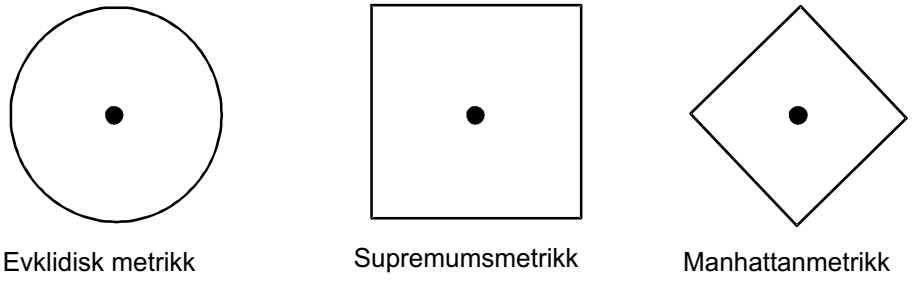
er invariant under rotasjoner. Supremumsmetrikken ligner på Manhattanmetrikken, men med den forskjell at supremumsmetrikken velger den største av koordinatdifferansene som avstanden mellom to punkter.

De omtalte metrikker har følgende formelle definisjoner:

Evklidisk metrikk: $d(a, a') = \sqrt{(a.x - a'.x)^2 + (a.y - a.y')^2}$

Manhattanmetrikk: $d(a, a') = \mid a.x - a'.x \mid + \mid a.y - a'.y \mid$

Supremumsmetrikk: $d(a, a') = \max(\mid a.x - a'.x \mid, \mid a.y - a'.y \mid)$



Figur 3.2: Noen aktuelle metrikker for GIS-anvendelser.

Eksempel: I forbindelse med geodatabaser kan det være aktuelt å be om alle punkter som ligger innenfor et kvadratisk vindu, eventuelt en terning. Vi forutsetter at vinduet ligger parallelt med koordinataksene. Sjakkbrett-metrikken kan da komme til anvendelse. Senterpunktet i vinduet benyttes som referansepunkt for utregning av koordinatdifferanser. Når lengden av vinduets sidekant benevnes s, er punktmengden innenfor vinduet definert ved:

$$\{p \mid \max(\Delta x, \Delta y, \Delta z) \leq s/2\}$$

Eksempel: Den geodetiske avstand ρ er lengden av den korteste storsirkelbue som forbinder to punkter på kuleflaten S^2. Det lar seg vise at ρ er en metrikk på S^2.

Definisjon 2 *Et metrisk rom er et par (X, d) hvor X er en mengde og d en metrikk på X.*

Definisjon 3 *La (X, d) være et metrisk rom og la p være et punkt i X. Dersom $U_p \subset X$ og $U_p = \{x \mid d(p, x) < \varepsilon \ \wedge \ \varepsilon > 0 \}$, er U_p en basisomegn til p.*

Eksempel: På den reelle tall-linje med den Evklidiske metrikk er basisomegnen til et punkt x et åpent intervall $(x - \epsilon, x + \epsilon) = \{y \mid x - \epsilon < y < x + \epsilon\}$.

Eksempel: Tilsvarende vil i planet med den Evklidiske metrikk basisomegnen til et punkt (x_1, y_1) være en sirkulær skive uten omriss med senter i (x_1, y_1) og radius ϵ. Gitt ved $U_p = \{(x, y) \in R^2 \mid (x - x_1)^2 + (y - y_1)^2 < \epsilon^2\}$.

Definisjon 4 *La (X, d) være et metrisk rom. En delmengde O til X sies å være en åpen mengde dersom O inneholder en basisomegn for alle $x \in O$. En delmengde C til X er lukket dersom dens komplement X/C er åpen.*

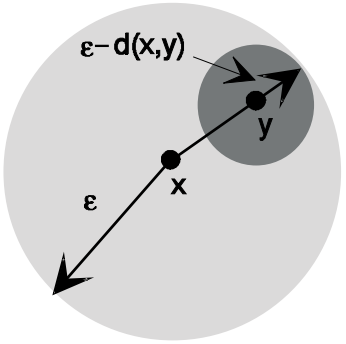
Eksempel: Anta R med den Evklidiske metrikk. Vi skal vise at mengden $O = \{x \in R \mid x > 0\}$ er åpen. La $x \in O$. Dette medfører at $x > 0$ og følgelig vil størrelsen $\epsilon = x/2$ være positiv. Fortsettelsen av beviset behandler så de to tilfellene $y > x$ og $y < x$.

1. Dersom $y > x$, er det uten videre klart at $y \in O$.

2. Dersom $y < x$ og $\mid x - y \mid < \epsilon$, er $x - y < x/2$, slik at $x/2 < y$. Av dette kan vi slutte at $y \in O$. Følgelig har x en basisomegn som i sin helhet er innbefattet i O. Siden x var et tilfeldig valgt punkt i O, har vi vist at O er åpen. Merk at størrelsen på basisomegnen til x, altså den valgte ϵ, er avhengig av x. Siden O er åpen, er dens komplement $\{x \in R \mid x \leq 0\}$ lukket.

Teorem 1 *En basisomegn er en åpen mengde.*

Bevis: La (X, d) være et metrisk rom, la $x \in X$ og la ϵ være positiv. Vi må vise at $N(x, \epsilon)$ er åpen, så derfor velger vi et element y i $N(x, \epsilon)$ og setter oss til oppgave å vise at en basisomegn til y er inbefattet i $N(x, \epsilon)$. Se figur 3.3.



Figur 3.3: En basisomegn er en åpen mengde.

Siden $d(x, y) < \epsilon$, er $\epsilon - d(x, y)$ positiv. La $d(y, z)$ være mindre enn $\epsilon - d(x, y)$. Da gjelder

$$d(x, z) \leq d(x, y) + d(y, z) < d(x, y) + \epsilon - d(x, y) = \epsilon$$

Herav følger at $z \in N(x, \epsilon)$, og vi har funnet en basisomegn til y som er innbefattet i $N(x, \epsilon)$. $\square$

Definisjon 5 *Samlingen av åpne mengder i et metrisk rom (X, d), kalles en topologi på (X, d).*

Vi kan si at et metrisk rom er et topologisk rom hvor topologien *induseres* av en metrikk. Metriske rom er derfor topologiske rom. For GIS anvendelser spiller de metriske rom en sentral rolle.

Det neste teoremet handler om åpne mengder og vil senere bli benyttet i en generalisert definisjon av topologisk rom.

Teorem 2 *La (X, d) være et metrisk rom og la $\mathcal{T}$ være samlingen av åpne delmengder til X. Da er følgende utsagn sanne:*

- $\emptyset \in \mathcal{T}$ *og* $X \in \mathcal{T}$*;*
- *Hvis* O_1 *og* O_2 *er i* $\mathcal{T}$*, da er også* $O_1 \cap O_2$ *i* $\mathcal{T}$*;*
- *Dersom* $\mathcal{L}$ *er en samling av mengder og hver av disse er i* $\mathcal{T}$*, da er* $\cup\{O \mid O \in \mathcal{L}\}$ *en mengde i* $\mathcal{T}$*.*

Bevis: Beviset overlates leseren eller han kan finne det i [Sch79]. □

Teorem 3 *La (X, d) være et metrisk rom og la $\mathcal{C}$ være samlingen av lukkede delmengder til X. Da er følgende utsagn sanne:*

- $\emptyset \in \mathcal{C}$ *og* $X \in \mathcal{C}$*;*
- *Hvis* C_1 *og* C_2 *er i* $\mathcal{C}$*, da er også* $C_1 \cup C_2$ *i* $\mathcal{C}$*;*
- *Dersom* $\mathcal{L}$ *er en samling av mengder og hver av disse er i* $\mathcal{C}$*, da er* $\cap\{C \mid C \in \mathcal{L}\}$ *en mengde i* $\mathcal{C}$*.*

Bevis: Beviset overlates leseren eller han kan finne det i [Sch79]. □

Definisjon 6 *La (X, d) være et metrisk rom, la A være en delmengde i X og la x være et punkt i X. Vi sier at x er et grensepunkt (eng. limit point) i A hvis hver basisomegn til x inneholder et annet punkt i A enn x.*

Popolært sagt er x et grensepunkt til A dersom det finnes andre punkter i A enn x som ligger så nære x vi bare måtte ønske.

Eksempel: La $A = \{x \in R \mid 0 < x \leq 1\} \cup \{2\}$, og la R ha en Evklidisk metrikk. Siden $d(0, \epsilon/2) = \epsilon/2 < \epsilon$, inneholder hver basisomegn til 0 et annet punkt i A enn 0. Derfor er 0 et grensepunkt til A. Et lignende argument viser at alle punkter mellom 0 og 1 inklusive er grensepunkter til A. Punkt 2 er ikke et grensepunkt til A, fordi vi kan finne en basisomegn til 2 som ikke inneholder noe annet punkt i A enn 2. Dette er tilfellet om vi velger $\epsilon = 1/2$. På lignende måte lar det seg vise at ikke noe punkt større enn 1 eller mindre enn 0 kan være et grensepunkt til A.

3.4 Generell topologi

Begrepet topologi har vi hittil knyttet til metriske rom, men begrepet kan gis et langt mere generelt innhold ved å basere definisjonen kun på åpne mengder. Dette gjør at topologiske rom kan gjøres til en *paraply* som samler en rekke matematiske rom.

3.4.1 Generalisering av begrepet topologi

Ved å anvende teorem 2 kan gis følgende generelle definisjon av en topologi på en mengde:

Definisjon 7 *En topologi på en mengde X er en samling $\mathcal{T}$ av delmengder til X som tilfredsstiller følgende betingelse:*

1. $\emptyset \in \mathcal{T}$ *og* $X \in \mathcal{T}$*;*
2. *Dersom* O_1 *og* O_2 *er i* $\mathcal{T}$*, så er også* $O_1 \cap O_2$ *det;*
3. *Hvis* $\mathcal{L}$ *er en samling delmengder og hver enkelt av disse er i* $\mathcal{T}$*, så er også* $\cup\{O \mid O \in \mathcal{L}\}$ *i* $\mathcal{T}$*.*

Punktene 1,2 og 3 kan oppfattes som aksiomer. Medlemmene i $\mathcal{T}$ kalles *åpne mengder*, og deres komplement i X kalles *lukkede mengder*. Et *topologisk rom* er et par $(X, \mathcal{T})$ hvor X er en mengde og $\mathcal{T}$ er en topologi på X.

Begrepet topologisk rom er her gitt en generell definisjon, som langt overgår vår definisjon som var knyttet til metriske rom. Dette fører til at omegnsbegrepet også får en mere generell definisjon:

Definisjon 8 *La $(X, \mathcal{T})$ være et topologisk rom og la x være et punkt i X. En omegn til x er enhver åpen mengde som inneholder x.*

Begrepet grensepunkt kan gis en mere generell definisjon ved å bytte ut basisomegn med åpen mengde. Dette leder til følgende teorem:

Teorem 4 *La (X, d) være et metrisk rom og A en delmengde til X. Et punkt x er et grensepunkt til A hvis og bare hvis hver åpen mengde som inneholder x, også inneholder et annet punkt i A enn x.*

Bevis: Beviset kan finnes i [Sch79]. □

For å illustrere det generelle i de foregåendende definisjoner, skal gis noen eksempler.

Eksempel: La $X = \{a, b, c, d, e\}$. Avgjør om de følgende mengder er en topologi på X:

$$\mathcal{T}_1 = \{X, \emptyset, \{a\}, \{a, b\}, \{a, c\}\}$$
$$\mathcal{T}_2 = \{X, \emptyset, \{a, b, c\}, \{a, b, d\}, \{a, b, c, d\}\}$$
$$\mathcal{T}_3 = \{X, \emptyset, \{a\}, \{a, d\}, \{a, c, d\}, \{a, b, c, d\}\}$$

$\mathcal{T}_1$ er ikke en topologi på X siden $\{a, b\} \cup \{a, c\} = \{a, b, c\} \notin \mathcal{T}_1$
$\mathcal{T}_2$ er ikke en topologi på X siden $\{a, b, c\} \cap \{a, b, d\} = \{a, b\} \notin \mathcal{T}_2$
$\mathcal{T}_3$ er en topologi på X siden alle nødvendige aksiomer er tilfredsstilt.

Eksempel: La $(X, \mathcal{T})$ være et toplogisk rom. Vi velger $X = \{a, b, c, d, e\}$ og $\mathcal{T} = \{X, \emptyset, \{a\}, \{c, d\}, \{a, c, d\}, \{b, c, d, e\}\}$

Siden komplementet til en åpen mengde er lukket, finnes de lukkede delmengder til X ved å ta komplementet til X sine de åpne mengder. Dette gir følgende lukkede mengder i X:

$$\{\emptyset, X, \{b, c, d, e\}, \{a, b, e\}, \{b, e\}, \{a\}\}$$

Legg merke til at det er mengder i X som er både lukkede og åpne slik som $\{b, c, d, e\}$ og mengder slik som $\{a, b\}$ som hverken er åpne eller lukkede.

Eksempel: For å spesifisere en topologi på en mengde, trenger vi ikke angi alle åpne mengder. For eksempel kan vi innføre den vanlige topologien på R ved bare å spesifisere intervallene av formen (a, ∞) og $(-\infty, b)$ for $a, b \in R$ som åpne mengder. Av definisjonens punkt 2. følger at alle intervaller (a, b) for $a, b \in R$ må være åpne mengder og ved punkt 3. at alle mengder av formen $O = \cup_{i \in I}(a_i, b_i)$ for $(a_i, b_i) \in R$ og en vilkårlig indeksmengde I, må være åpen. Dermed har vi samtlige åpne mengder i R med den metriske topologien.

Eksempel: På tilsvarende måte kan den metriske topologien på R^2 spesifiseres ved å ta utgangspunkt i klassen av basisomegner og alle endelige snitt av disse for deretter å danne alle unioner av foregående mengder.

Eksempel: La $(X, \mathcal{T})$ være et topologisk rom hvor

$$\mathcal{T} = \{X, \emptyset, \{a\}, \{c, d\}, \{a, c, d\}, \{b, c, d, e\}\}.$$

Anta en delmengde $A = \{a, b, c\}$ til X. Vi har da at $b \in X$ er et grensepunkt til A siden alle åpne mengder til $\mathcal{T}$ som inneholder b, også inneholder et annet punkt i A enn b. Derimot er a ikke et grensepunkt i A, fordi den åpne mengden $\{a\}$ ikke inneholder et annet punkt i A enn a.

3.4.2 Den lukkede, den indre og omrisset til en mengde

Definisjon 9 *La $(X, \mathcal{T})$ være et topologisk rom og la A være en delmengde i X. Vi definerer den lukkede (eng. the closure) til A som mengden $\overline{A} = A \cup \{x \mid x$ er et grensepunkt til $A\} = A \cup A'$.*

Vi kan si at x tilhører $\overline{A}$ hvis alle omegner til x snitter A. Altså $\{x \mid U_x \cap A \neq \emptyset\} \subset \overline{A}$

Eksempel: La $A = \{x \in R \mid 0 < x \leq 1\} \cup \{2\}$, og la R ha en Evklidisk metrikk. Da er $\overline{A} = \{x \in R \mid 0 \leq x \leq 1\} \cup \{2\}$.

Den tomme mengde er den eneste mengde som har en tom lukkede.

Teorem 5 *La $(X, \mathcal{T})$ være et topologisk rom og la A være en delmengde til X. Da gjelder:*
1. $\overline{A}$ er en lukket mengde;
2. $\overline{\overline{A}} = \overline{A}$;
3. $\overline{A} = \cap\{F : A \subset F \subset X\}$ og F er lukket.

Teoremet sier at $\overline{A}$ er den minste mengde som er lukket og som inneholder A.

Bevis: Beviset kan finnes i [Sch79]. □

Eksempel: Anta et topologisk rom $(X, \mathcal{T})$ hvor $X = \{a, b, c, d, e\}$ og hvor de lukkede delmengder til X er: $\emptyset, X, \{b, c, d, e\}, \{a, b, e\}, \{b, e\}, \{a\}\}$ Vi har da:

$$\overline{\{b\}} = \{b, e\} \text{ og } \overline{\{a, c\}} = X \text{ og } \overline{\{b, d\}} = \{b, c, d, e\}.$$

Definisjon 10 *Den indre (eng. the interior) A° til A er den største åpne mengden som er innbefattet i A.*

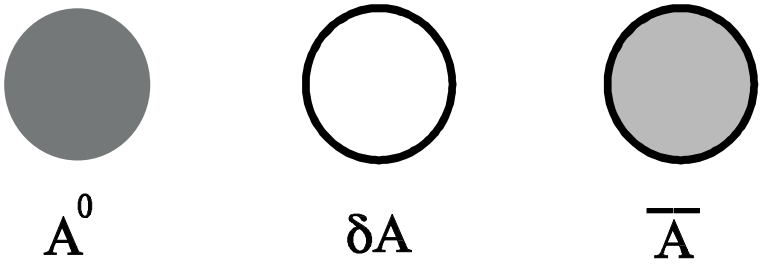
Gitt $A \subset X$. A° kan sies å være mengden av alle x som tilfredsstiller betingelsen at det finnes en omegn til x som er innbefattet i A. Altså $A^\circ = \{x \mid U_x \subset A\}$.
Den indre til A kan være en tom mengde.

Eksempel: $A = \{(x, y) \in R^2 \mid x^2 + y^2 \leq 1\}$ er en sirkelskive med omriss. Den indre er gitt ved $A^0 = \{(x, y) \in R^2 \mid x^2 + y^2 < 1\}$.

Definisjon 11 *Omrisset (eng. the boundary) ∂A til A er definert ved $\partial A = \overline{A} \cap \overline{X \setminus A}$.*

Gitt $A \subset X$. ∂A er en lukket mengde og kan sies å være mengden av alle x som tilfredsstiller betingelsen at omegnen til x snitter både A og komplementet til A. Altså: $\partial A = \{x \mid U_x \cap A \neq \emptyset \ \wedge \ U_x \cap (X \setminus A) \neq \emptyset\}$

Eksempel: Omrisset til mengden i foregående eksempel er gitt ved:
$\partial A = \{(x, y) \in R^2 \mid x^2 + y^2 = 1\}$.



Figur 3.4: Illustrasjon av begrepene indre, omriss og lukkede.

Eksempel: Begrepene den indre til A, den lukkede til A og omrisset til A er illustrert i figur 3.4.

Vi skal vise at det er overensstemmelse mellom vår visuelle oppfattelse av relasjoner mellom $A^0, \partial A$ og $\overline{A}$ og de matematiske relasjoner mellom de nevnte størrelser.

Teorem 6 $A^\circ \cap \partial A = \emptyset$

Teoremet hevder at A^0 og ∂A ikke har noen elementer felles, som er i samsvar med vår umiddelbare visuelle oppfattelse av objektene i figur 3.4.

Bevis: Dersom $x \in \partial A$, vil hver omegn U_x til x snitte $X \setminus A$ slik at U_x kan ikke være en delmengde av A. Siden ingen omegn U_x er delmengde av A, følger det at $x \notin A^0$ og derfor: $A^\circ \cap \partial A = \emptyset$. □

Teorem 7 $\overline{A} = A^\circ \cup \partial A$

Det at teoremet hevder at den lukkede til A er unionen av den indre til A og omrisset til A, er vel heller ikke overraskende fra et visuelt synspunkt.

Bevis: Ifølge definisjonene 9, 10 og 11 har vi at $A^0 \subset A \subset \overline{A}$ og at $\partial A \subset \overline{A}$. Siden A^0 og ∂A begge er delmengder til $\overline{A}$, følger det at $\partial A \cup A^0 \subset \overline{A}$. Vi velger så en x slik at $x \in \overline{A}$ og antar at $x \notin A^0$ (at det finnes en x som tilfredsstiller denne antagelsen følger av at $A^0 \subset \overline{A}$, $\partial A \subset \overline{A}$ og $A^0 \cap \partial A = \emptyset$). Forutsetningen om at $x \notin A^0$ impliserer at ingen omegn til x er delmengder av A, derfor vil hver omegn til x snitte $X \setminus A$. Dette fører til at x må være et element i $\overline{X \setminus A}$. Da har vi ifølge definisjon 11 at $x \in \partial A$. Herav følger det at $\overline{A} \subset (A^0 \cup \partial A)$, men siden både A^0 og ∂A ifølge forutsetningene er delmengder av $\overline{A}$, har vi at $\overline{A} = A^0 \cup \partial A$. □

Eksempel: Anta at X er en delmengde av R (alle reelle tall) og at Y er en ekte delmengde av X gitt ved: $X = \{x \in R \mid -200 < x < 100\}$ og $Y = \{y \in X \mid 3 < y < 7\}$. Vi har da:

$$Y^\circ = \{y \mid 3 < y < 7\}$$
$$\overline{Y} = \{y \mid 3 \leq y \leq 7\}$$

$$Y_X^c = X \setminus Y = \{y \mid -200 < y \leq 3 \ \vee \ 7 \leq y < 100\}$$
$$\overline{Y_X^c} = \{y \mid -200 \leq y \leq 3 \vee \ \leq 7y \leq 100\}$$
$$\partial Y = \overline{Y} \cap \overline{Y^c} = \{3, 7\}$$
$$Y^0 \cap \partial Y = \emptyset$$
$$Y^0 \cup \partial Y = \{y \mid 3 \leq y \leq 7\}$$

3.4.3 Sammenhengende rom

Begrepene sammenhengende rom og separasjon spiller en viktig rolle når vi senere skal definere topologiske relasjoner mellom mengder. Vi skal derfor utdype og presisere hva vi legger i disse begrepene.

Definisjon 12 *Et rom X er sammenhengende dersom det ikke er unionen av to ikke-tomme disjunkte lukkede delmengder. Rommet X er usammenhengende dersom det ikke er sammenhengende.*

Den matematiske definisjonen svarer til vår intuitive oppfatning av begrepet sammenhengende.

Eksempel: Rommet $X = [0, 1] \cup [2, 3]$ er usammenhengende fordi X er unionen av to disjunkte lukkede mengder. To mengder A og B er som kjent disjunkte (atskilte) dersom de ikke har noen elementer felles.

For å kunne avgjøre om to rom er sammenhengende eller ikke, trenger vi noen verktøy. De etterfølgende definisjoner og teoremer er nyttige i en slik sammenheng.

Definisjon 13 *To delmengder K og L av et rom X, er gjensidig separert (adskilt) dersom $\overline{K} \cap L = K \cap \overline{L} = \emptyset$*

Eksempel: Intervallene $[0, 1/2)$ og $(1/2, 1]$ er gjensidig separert siden $\overline{[0, 1/2)} \cap (1/2, 1] = [0, 1/2] \cap (1/2, 1] = \emptyset$ og $[0, 1/2) \cap \overline{(1/2, 1]} = [0, 1/2) \cap [1/2, 1] = \emptyset$.

Teorem 8 *En delmengde A i et rom X er sammenhengende hvis og bare hvis A ikke er unionen av to ikke-tomme gjensidig separerte delmengder til X.*

Bevis: Siden teoremet inneholder uttrykket "hvis og bare hvis", må vi bevise at begge "hvis"holder.

Anta at $A = K \cup L$, hvor K og L er ikke-tomme gjensidig separerte delmengder i X. Da er $\overline{K_A} = \overline{K_X} \cap A = \overline{K_X} \cap (K \cup L) = (\overline{K_X} \cap K) \cup (\overline{K_X} \cap L) = K \cup \emptyset = K$, slik at K er lukket i A. På lignende måte lar det seg vise at L er lukket i A. Dette viser at A ikke er sammenhengende.

Anta at A er usammenhengende. Da er $A = M \cup N$, hvor M og N er ikke-tomme disjunkte lukkede delmengder til A. Siden M er lukket i A og $N \subset A$, har vi at: $\overline{M_X} \cap N = \overline{M_X} \cap (A \cap N) = (\overline{M_X} \cap A) \cap N = \overline{M_A} \cap N = M \cap N = \emptyset$. På lignende måte lar det seg vise at $M \cap \overline{N_X} = \emptyset$. Dette viser at A er unionen av to ikke-tomme gjensidig separerte mengder. □

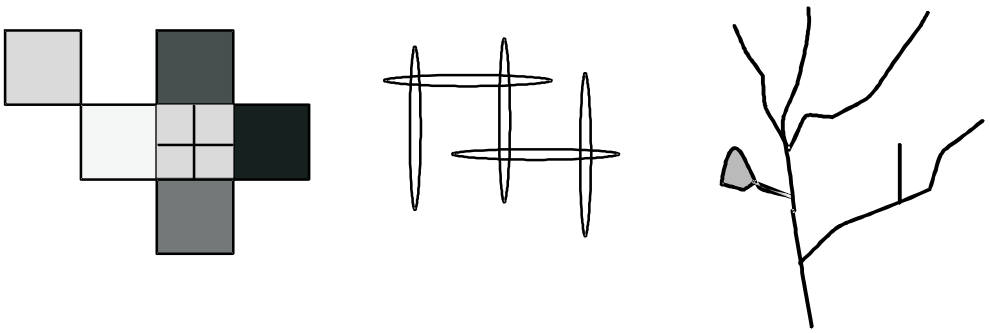
Teorem 9 *La A være en sammenhengende delmengde til X som er innbefattet i unionen av to gjensidig separerte mengder K og L. Da er enten $A \subset K$ eller $A \subset L$.*

Bevis: Siden $A \subset K \cup L$, har vi at $A = (A \cap K) \cup (A \cap L)$. Siden K og L er gjensidig separerte, $\overline{(A \cap K)} \cap (A \cap L) \subset \overline{K}_A \cap L = \emptyset$. Tilsvarende har vi at $(A \cap K) \cap \overline{(A \cap L)} \subset K \cap \overline{L}_A = \emptyset$. Følgelig er A unionen av de to gjensidig separerte mengder $A \cap K$ og $A \cap L$, men siden A er sammenhengende er enten er $A \cap K = \emptyset$ eller så er $A \cap L = \emptyset$. I det førstnevnte tilfellet er $A \subset L$ og i det sistnevnte tilfellet er $A \subset K$. □

Teorem 10 *La $\{A_\alpha \mid \alpha \in \mathcal{P}\}$ være en samling av sammenhengende delmengder til et rom X. Dersom hver enkelt av disse inneholder punktet x_0, da er $A = \cup\{A_\alpha \mid \alpha \in \mathcal{P}\}$ sammenhengende.*

Bevis: Anta at A er unionen av de gjensidig separerte mengdene K og L, og at $x_0 \in K$. Siden $A_\alpha \subset A$ og A_α er sammenhengende, sier teorem 9 at $A_\alpha \subset K$ eller $A_\alpha \subset L$. Men $x_0 \in K$ slik at $A_\alpha \subset K$. Dette er er sant for hver α, så derfor er $A = \cup\{A_\alpha \mid \alpha \in \mathcal{P}\}$, og herav følger at $L = \emptyset$. Følgelig er A ikke unionen av to ikke-tomme gjensidig separerte mengder og A er derfor sammenhengende. □

Som figur 3.5 viser, gir teoremet stor fleksibilitet med hensyn på konstruksjon av sammenhengende mengder.



Figur 3.5: Eksempler på sammenhengende mengder.

Teorem 11 *La A være en sammenhengende delmengde til et rom X, og la B være slik at $A \subset B \subset \overline{A}$. Da er B sammenhengende.*

Bevis: Det vises til [Sch79]. □

Definisjon 14 *La X være et sammenhengende rom. En delmengde S til X sies å splitte X dersom $X \setminus S$ ikke er sammenhengende, og vi sier at S er en splitt til X. Dersom $S = \{x\}$ er en splitt til X, da kalles x et splittpunkt til X. Dersom $X \setminus \{x\}$ er sammenhengende, kalles x et ikke-splittpunkt til X.*

Teorem 12 *Gitt et sammenhengende rom X. Anta at $A \subset X$. Dersom $A^\circ \neq \emptyset$ og $\overline{A} \neq X$, da er A° og $X \setminus \overline{A}$ gjensidig separerte. Følgelig vil ∂A splitte X.*

Bevis: Vi har antatt at $A^\circ \neq \emptyset$ og $X \setminus \overline{A} \neq \emptyset$. Ved å anvende teoremene 6 og 7 har vi at $A^\circ \cap \partial A = \emptyset$ og $\overline{A} = A^\circ \cup \partial A$. Dette impliserer at $A^\circ \cap X \setminus \overline{A} = \emptyset$. Følgelig har vi at A° og $X \setminus \overline{A}$ er disjunkte åpne mengder. Av definisjon 13 har vi da at A° og $X \setminus \overline{A}$ er gjensidig separerte.

Teorem 7 impliserer at $X \setminus \partial A = (\overline{A} \setminus \partial A) \cup (X \setminus \overline{A}) = A^\circ \cup X \setminus \overline{A}$. Av dette kan vi slutte at $X \setminus \partial A$ er usammenhengende og følgelig vil ∂A splitte X. □

Følgende teorem, som skal gjengis uten bevis, gir oss en forståelse av hva som ligger i begrepet at to rom er homeomorfe.

Teorem 13 .

1. *Dersom X og Y er homeomorfe, har X og Y det samme antall splittpunkt.*

2. *Dersom X og Y er homeomorfe, har X og Y det samme antall ikke-splittpunkt.*

Eksempel: En sirkel S^1 har ingen splittpunkt. I intervallet $[0, 1]$ er for eksempel $1/2$ et splittpunkt. Derfor er S^1 og $[0, 1]$ ikke homeomorfe.

Eksempel: Når vi ønsker å lukke polygoner i en kartdatabase, så er det for å oppnå at representasjonen av omrisset til et objekt skal bli homeomorf med modellen av omrisset til objektet.

Eksempel: Topologisk sett er det ingen forskjell på en sirkel og en firkant.

Vi benytter ofte begrepet komponent. Her skal vi gi en definisjon som tar utgangspunkt i forestillingen om største sammenhengende delmengde.

Definisjon 15 *La x være et punkt i rommet X. Komponenten til X som inneholder x er $\cup\{K \mid K$ er en sammenhengende delmengde til X som inneholder $x\}$.*

3.5 Topologiens betydning for GIS

Begrepsdannelser fra den matematiske topologi kommer ofte til anvendelse innen GIS. Dette er bakgrunnen for den formelle vinklingen av stoffet i de foregående avsnitt. Så lenge et begrep ikke er gitt en presis definisjon, kan vi nemlig legge hva vi vil i det. De kommende avsnitt belyser med en rekke eksempler hvorfor topologi er viktig i GIS.

3.5.1 Eksempler på anvendelse av topologisk terminologi i GIS

For arbeidet med standarder såvel som systemutvikling, er det viktig med presise og tilstrekkelig generelle begreper. Fra praksis kan det vises til GIS-standarder som lider av at forankringen i topologisk teori er for svak. Dette gir seg utslag i uklar og inkonsistent terminologi samt at topologiske relasjoner mellom objekter ikke lar seg uttrykke på en fullstendig måte. Noen eksempler vil belyse hvordan begreper fra topologien kan anvendes innen GIS.

Eksempel: Den generelle definisjonen på topologi er ikke knyttet mot en metrikk (avstandsmål). Derfor er egenskaper som hvor langt, hvor bredt, hvor høyt, hvor mange kvadratmeter hvor mange kubikkmeter, hvilken form etc. ikke topologiske egenskaper, men geometriske egenskaper.

Eksempel: Topologisk sett er det ingen forskjell på en krum linje og en rett linje, på en firkant og en sirkel eller på en terning og en kule. De er med andre ord homeomorfe. Derimot har et lukket og et ikke-lukket polygon ulike topologiske egenskaper. Egenskapen lukket kurve, eller lukket polygon er derfor en topologisk egenskap.

Eksempel: Når vi ønsker å lukke polygoner i en kartdatabase, så er det for å oppnå at representasjonen av omrisset til et objekt skal bli homeomorf med modellen av omrisset til objektet. Tilsvarende eksempler finner vi når små gap fjernes ved at linjer forlenges til skjæring eller når små ender fjernes i nærheten av skjæringspunktet mellom linjer. Vi sier at slike redigeringer har som formål å korrigere topologien til representasjonen av objektet, og vi kan derfor snakke om *topologisk editering*.

Eksempel: Fra et topologisk synspunkt er det ingen forskjell på en sirkel og en ellipse. En korrigering av en ellipse til å bli en sirkel, er derfor ikke en topologisk korreksjon, men en *geometrisk korreksjon*.

Eksempel: En eiendomsteig oppfattes vanligvis som en mengde i et to-dimensjonalt metrisk rom, altså som en mengde i R^2. Siden jorda er et tre-dimensjonalt objekt, kan det i noen tilfeller være nødvendig å definere eiendomsteiger som mengder i R^3. En eiendom har nemlig en utstrekning ned i bakken eller opp i lufta. Vi skal senere komme tilbake med en presisering av hva som menes med et objekts topologiske dimensjon.

Eksempel: Innen kartografien benyttes begrepet informasjonsbærende enheter om punkter, linjer, areal og volum. Topologisk kan vi skille mellom disse elementene ved deres topologiske dimensjon. De er henholdsvis 0- 1- 2- og 3-dimensjonale.

Eksempel: Grenselinjen for en eiendomsteig kan defineres som dens omriss. Omrisset såvel som den indre til en mengde er en topologisk egenskap.

Eksempel: At to eiendomsteiger har en felles grenselinje eller at den ene teigen ligger innenfor den andre, er en topologisk egenskap.

Eksempel: For å avgjøre om to elver tilhører samme vassdrag, vil det være nyttig å støtte seg på begrepsdannelser om sammenhengende mengder. Det at to mengder er sammenhengende, er en topologisk egenskap de har.

3.5.2 Betydningen av topologisk informasjon i GIS

Når vi innenfor geografiske informasjonssystemer snakker om topologisk informasjon, mener vi informasjon om topologiske relasjoner, naboskap eller egenskaper til objektene som er uforandret ved kontinuerlige forandringer som skalering, strekk, krymp, rotasjoner eller forskyvninger av det underliggende topologiske rom.

Eksempel: Dersom vi vet at objekt A ligger innenfor objekt B, vil dette forhold ikke bli forandret om kartmålestokken endres, om en ny kartprojeksjon velges (gjelder ikke projeksjoner generelt) eller om det underliggende topologiske rom (altså alle objekter) forskyves avstand s i retning ϕ. De topologiske rom i eksemplet er homeomorfe.

Eksempel: For en rekke GIS-anvendelser er det nødvendig å kjenne topologiske relasjoner mellom objektene. Det er her to veier å gå, enten å uttrykke informasjon om topologiske relasjoner eksplisitt i geodatabasen eller å utlede topologiske relasjoner på grunnlag av den geometriske beskrivelsen for hver gang det er behov for topologisk informasjon. I det siste tilfellet skal man være oppmerksom på at beregningsproblemet fort kan få et betydelig omfang. Dessuten vil unøyaktigheter i den geometriske beskrivelsen kunne føre til at topologiske relasjoner ikke blir korrekt utledet.

Eksempel: Et typisk spørsmål mot et eiendomskartverk er hvilke eiendommer som grenser mot en gitt eiendom. I et informasjonssystem over eiendommer, vil det derfor være fornuftig å uttrykke eksplisitt hvilke eiendommer som er naboer til eiendom A.

Eksempel: For analyseformål i GIS er det ofte av vesentlig betydning at representasjonen har korrekt topologi. Dersom vi skal fylle et polygon med en farge, vil farge tyte ut der hvor det er åpninger i polygonet. Følgen kan bli at hele kartet blir fargelagt.

Eksempel: Vi skal lage et topografisk kart og ønsker å fargelegge alle vann blå. Det er her for enkelt å si at alt innenfor et vann-polygon skal fargelegges blått, fordi et vann kan ha øyer, og øyene skal ha en annen farge enn blå. I dette tilfellet må vi derfor vite om et polygon har hull eller ikke.

3.6 Topologiske relasjoner

Dagens programvare til GIS-markedet lider av at topologiske relasjoner mellom geografiske objekter ofte ikke lar seg modellere på en konsistent og fullstendig måte. Det er derfor ønskelig å ha en systematisk beskrivelse av topologiske relasjoner mellom geografiske objekter. Dette vil sette oss i stand til å evaluere fullstendigheten til ulike datamodeller.

Eksempel: Anta at vi har en database over veier og skog. Vi kan spørre om hvilke topologiske relasjoner som skal tillates for disse objektene. At en vei kan ligge innenfor en skog er vel greit, men kan en skog ligge innenfor en vei? Kanskje, dersom en motorvei har en midtrabatt som er beplantet og vi definerer midtrabatten som en del av veien. Midtrabatten blir i såfall et areal som får tilordnet både egenskapen vei og egenskapen skog.

Egenhofer og Franzosa [EF91] gir et verdifullt bidrag til en systematisk beskrivelse av topologiske relasjoner mellom geografiske objekter. Den etterfølgende framstilling av emnet er basert på deres arbeid.

3.6.1 Metode for å beskrive topologiske relasjoner

Egenhofer og Franzosa's metode for å beskrive relasjoner mellom to mengder A og B i et topologisk rom X, baserer seg på de fire snitt mellom omriss og indre til de to mengdene, altså snittene: $\partial A \cap \partial B, A^\circ \cap B^\circ, \partial A \cap B^\circ$ og $A^\circ \cap \partial B$.

Definisjon 16 *La A og B være to delmengder av det topologiske rom $(X, \mathcal{T})$. En topologisk-romlig relasjon mellom A og B er beskrevet ved fire-tuplet:*

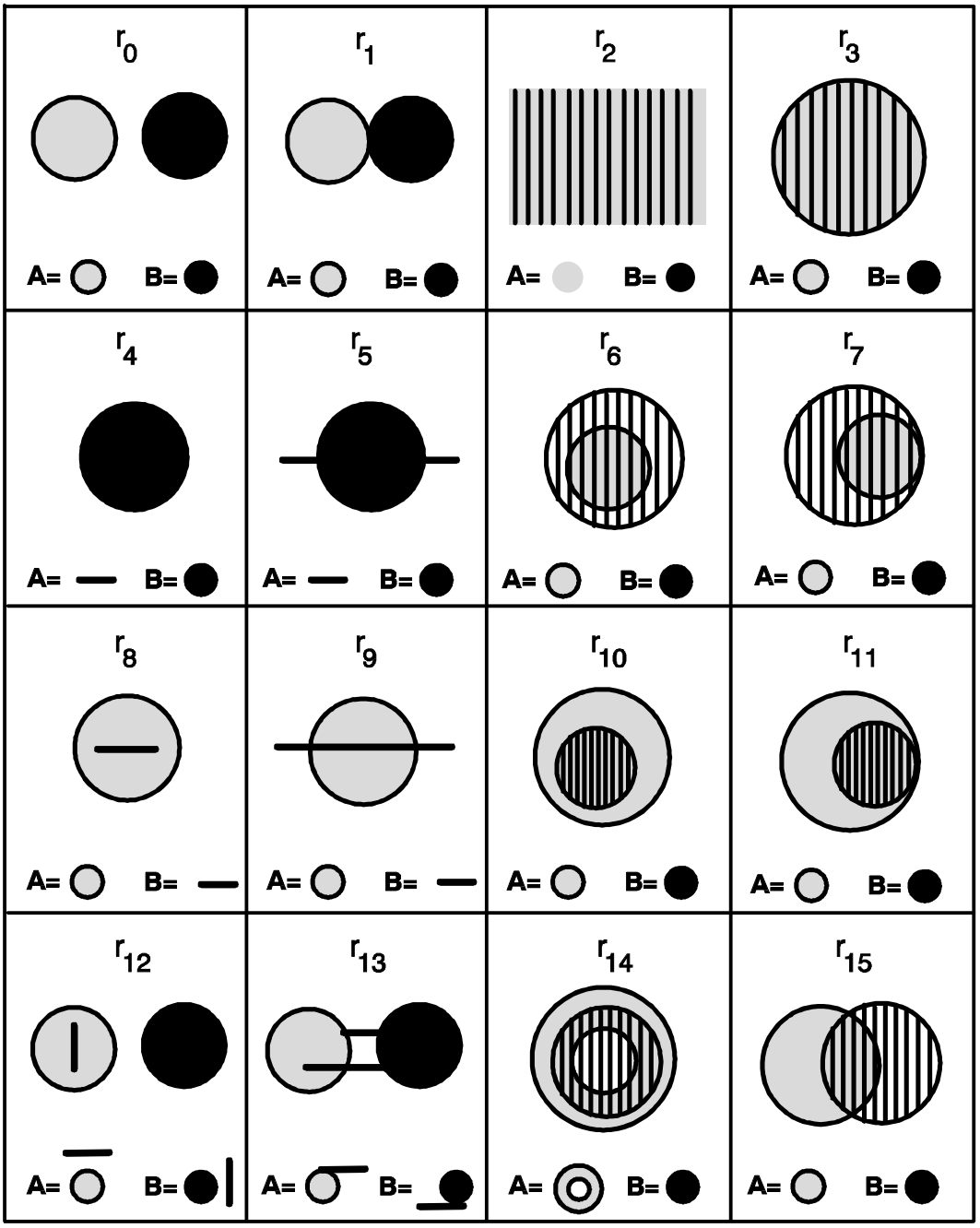
$$r = (\partial A \cap \partial B, A^\circ \cap B^\circ, \partial A \cap B^\circ, A^\circ \cap \partial B)$$

Siden en mengde enten er tom eller ikke-tom, kan vi slutte at det finnes 16 mulige topologisk-romlige relasjoner mellom to mengder A og B. Disse 16 relasjonene er listet opp i tabell 3.1. Romlige relasjoner mellom to mengder A og B vil senere bli omtalt som *binære* topologisk-romlige relasjoner.

Generelt kan alle de 16 romlige relasjonene forekomme. Avhengig av hvilke restriksjoner som legges på mengdene og det underliggende topologiske rom, vil de aktuelle romlige relasjoner være en delmengde av de 16 relasjoner i tabell 3.1. Dette skal vi komme tilbake i forbindelse med romlige relasjoner mellom regioner.

Det topologiske rom A og B tilhører, spiller en avgjørende rolle for den romlige relasjonen mellom A og B.

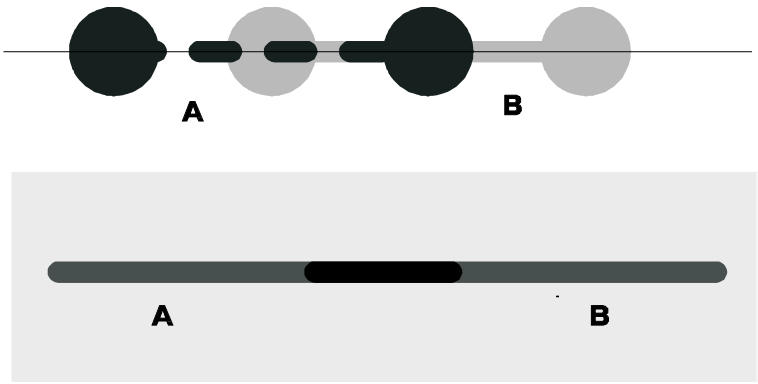
Eksempel: I figur 3.7 er det vist et en-dimensjonalt og et to-dimensjonalt tilfelle. I det en-dimensjonale tilfellet er de to mengdene A og B delmengder av R^1. Vi kan her tenke oss A og B som intervaller på den reelle tallinjen. Som figuren viser, har A og B i dette tilfellet relasjonen $(\emptyset, \neg\emptyset, \neg\emptyset, \neg\emptyset)$. I det andre tilfellet er A og B



Figur 3.6: Eksempler på de 16 binære topologisk-romlige relasjoner.

Tabell 3.1: De 16 binære topologisk-romlige relasjoner.

relasjon	$\partial \cap \partial$	$^\circ \cap ^\circ$	$\partial \cap ^\circ$	$^\circ \cap \partial$
r_0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
r_1	$\neg\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
r_2	$\emptyset$	$\neg\emptyset$	$\emptyset$	$\emptyset$
r_3	$\neg\emptyset$	$\neg\emptyset$	$\emptyset$	$\emptyset$
r_4	$\emptyset$	$\emptyset$	$\neg\emptyset$	$\emptyset$
r_5	$\neg\emptyset$	$\emptyset$	$\neg\emptyset$	$\emptyset$
r_6	$\emptyset$	$\neg\emptyset$	$\neg\emptyset$	$\emptyset$
r_7	$\neg\emptyset$	$\neg\emptyset$	$\neg\emptyset$	$\emptyset$
r_8	$\emptyset$	$\emptyset$	$\emptyset$	$\neg\emptyset$
r_9	$\neg\emptyset$	$\emptyset$	$\emptyset$	$\neg\emptyset$
r_{10}	$\emptyset$	$\neg\emptyset$	$\emptyset$	$\neg\emptyset$
r_{11}	$\neg\emptyset$	$\neg\emptyset$	$\emptyset$	$\neg\emptyset$
r_{12}	$\emptyset$	$\emptyset$	$\neg\emptyset$	$\neg\emptyset$
r_{13}	$\neg\emptyset$	$\emptyset$	$\neg\emptyset$	$\neg\emptyset$
r_{14}	$\emptyset$	$\neg\emptyset$	$\neg\emptyset$	$\neg\emptyset$
r_{15}	$\neg\emptyset$	$\neg\emptyset$	$\neg\emptyset$	$\neg\emptyset$



Figur 3.7: Det topologiske rom er avgjørende for den topologisk-romlige relasjonen mellom A og B.

linjesegmenter i R^2. Her har vi at omrisset til A er lik A, tilsvarende for B. Videre er $A^0 = \emptyset$ og $B^0 = \emptyset$. Den romlige relasjonen mellom A og B blir i dette tilfellet $(\neg\emptyset, \emptyset, \emptyset, \emptyset)$.

Siden de 16 romlige relasjoner som her er beskrevet, er topologiske egenskaper til mengdene, vil de romlige relasjoner bevares under homeomorfe avbildninger av *det underliggende rom.*

Altså dersom en funksjon $f : X \to Y$ er homeomorf og $A, B \subset X$, har $f(A)$ og $f(B)$ den samme romlige relasjonen som A og B.

Eksempel: Dersom vi endrer målestokken på kartet, endres ikke de topologisk-romlige relasjoner mellom objektene.

3.6.2 Topologiske relasjoner mellom romlige regioner

Vi skal nå legge en del restriksjoner både på det topologiske rom såvel som de mengder som skal studeres og ut fra dette vise hvilke topologiske relasjoner som kan bestå mellom to mengder. Som vi etter hvert vil se, vil begrepene *sammenhengende rom* og *separasjon* spille en avgjørende rolle i den etterfølgende behandlingen av temaet.

Selv om hovedhensikten i det etterfølgende er å studere relasjoner mellom areal begrenset av lukkede kurver i planet, beholdes likevel en generell framstilling slik at man ikke låser betraktningene til det todimensjonale rom R^2.

Definisjon 17 *La $(X, \mathcal{T})$ være et sammenhengende topologisk rom. En romlig region i X er en ekte delmengde A til X som tilfredsstiller følgende tre krav:*

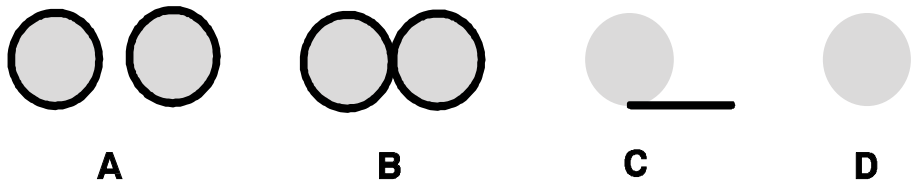
1. *A° er sammenhengende;*
2. *$A = \overline{A^\circ}$;*
3. *$A \neq \emptyset$.*

Fra definisjonen følger at den indre til enhver romlig region er ikke-tom. Videre er en romlig region lukket og sammenhengende. Dessuten sier betingelse 2. at omrisset til en region er en del av regionen.

Korollar 1 *Dersom A er en romlig region i X, da er $\partial A \neq \emptyset$*

Bevis: Ifølge definisjonen på en romlig region er $A^\circ \neq \emptyset$ og $X \setminus \overline{A} \neq \emptyset$. Av teorem 12 har vi at A° og $X \setminus \overline{A}$ er gjensidig separerte og at ∂A splitter X. Dersom vi velger $\partial A = \emptyset$, får vi at X blir unionen av to gjensidig separerte mengder. Ifølge teorem 8 vil dette bety at X er usammenhengende, men siden X er forutsatt å være sammenhengende, må $\partial A \neq \emptyset$. □

Eksempel: Figur 3.8 viser eksempler på mengder som ikke er romlige regioner. A og B er ikke romlige regioner fordi unionen av A° og B° ikke er sammenhengende. C og D er heller ikke romlige regioner fordi $C \neq \overline{C^\circ}$ og $D \neq \overline{D^\circ}$.



Figur 3.8: Mengder i planet som ikke er romlige regioner.

Eksempel: Mengden $A = \{(x, y) \in R^2 \mid 0 \leq x \leq 1\ ,\ 0 \leq y \leq 1\}$ er en romlig region. I dette tilfellet et kvadrat. Den indre til A er gitt ved $A^0 = \{(x, y) \in R^2 \mid 0 < x < 1\ ,\ 0 < y < 1\}$

Anta en annen romlig region gitt ved $B = \{(x, y) \in R^2 \mid 1 \leq x \leq 2\ ,\ 0 \leq y \leq 1\}$ Den indre til B er $B^0 = \{(x, y) \in R^2 \mid 1 < x < 2\ ,\ 0 < y < 1\}$

Ved å anvende definisjon 13, ser vi at A^0 og B^0 er gjensidig separerte. Derfor er $A^0 \cup B^0$ ifølge teorem 8 usammenhengende og følgelig kan ikke $A^0 \cup B^0$ være en romlig region.

Eksempel: Definisjonen av en romlig region har god overføringsverdi til praksis. I et eiendomskartverk sier vi at naboeiendommer har en felles grenselinje. Derimot dersom den ene eiendommen flyttes litt i forhold til den andre slik at den indre til eiendommene får et fellesområde, vil vi si at eiendommene overlapper hverandre. Eiendommer svarer derfor til vår definisjon av romlige regioner.

I et digitalt eiendomskartverk vil det være fornuftig å spesifisere hvilke romlige relasjoner i tabell 3.1 som kan godtas for eiendommer. Ved så å teste mot disse kriteriene for hver gang databasen oppdateres, vil man oppnå en konsistent beskrivelse av eiendommenes romlige utbredelse.

Gyldige relasjoner mellom romlige regioner

Det viser seg at noen av de 16 binære topologiske relasjoner mellom to romlige regioner ikke kan forekomme. Vi skal studere dette forhold litt nærmere.

Teorem 14 *Relasjonene $r_4, r_5, r_8, r_9, r_{12}, r_{13}$ kan ikke eksistere mellom to romlige regioner.*

Bevis: La A og B være to romlige regioner i X. Vi skal vise at dersom $\partial A \cap B^\circ \neq \emptyset$ eller $A^\circ \cap \partial B \neq \emptyset$, så er $A^\circ \cap B^\circ \neq \emptyset$. Dette innebærer i såfall at de seks relasjonene $r_4, r_5, r_8, r_9, r_{12}, r_{13}$ ikke kan forekomme mellom A og B .

Av teorem 7 har vi at $A^\circ \cup \partial A = \overline{A}$ og $A^\circ \cup \partial(A^\circ) = \overline{A^\circ}$. Av definisjonen på en romlig region følger at $\overline{A} = A = \overline{A^\circ}$, så derfor er $A^\circ \cup \partial A = A^\circ \cup \partial(A^\circ)$. Ved å anvende teorem 6, har vi at $A^\circ \cap \partial A = \emptyset$ og $A^\circ \cap \partial(A^\circ) = \emptyset$. Herav følger at $\partial(A^\circ) = \partial A$.

La så $x \in \partial A \cap B^\circ$, da må $x \in \partial(A^\circ)$. Av definisjonen på omrisset til en mengde har vi at enhver omegn til x vil snitte A°. Siden B° er åpen og inneholder x, vil

det være mulig å finne en omegn til x som er innebefattet i B°. Dette innebærer at $A^\circ \cap B^\circ \neq \emptyset$. Hvilket skulle vises. □

Teorem 15 *Relasjonen r_2 kan ikke eksistere mellom to romlige regioner.*

Bevis: La A og B være romlige regioner i X slik at $\partial A \cap \partial B = \emptyset$ og $A^\circ \cap B^\circ \neq \emptyset$. Vi skal vise at dersom $\partial A \cap B^\circ = \emptyset$ da er $A^\circ \cap \partial B \neq \emptyset$. Dette innebærer i såfall at relasjon r_2 ikke kan forekomme mellom A og B.

Anta at $A^\circ \cap \partial B = \emptyset$. Siden $B = B^\circ \cup \partial B$ og vi har antatt $\partial A \cap \partial B = \emptyset$ og $\partial A \cap B^\circ = \emptyset$, er $\partial A \cap B = \emptyset$. Forutsetningen om at $A^\circ \cap B^\circ \neq \emptyset$ fører til at $B \subset X \setminus \partial A$. Teorem 12 sier at A° og $X \setminus A$ er gjensidig separerte, men siden B er sammenhengende, vil i følge teorem 9 enten $B \subset A^\circ$ eller $B \subset X \setminus A$. Siden vi har antatt $A^\circ \cap B^\circ \neq \emptyset$, er $B \subset A^\circ$. Følgelig må $\partial B \subset A^\circ$. Det er da klart at $\partial B \cap A^\circ \neq \emptyset$. Hvilket skulle vises. □

Av de 16 binære topologiske relasjoner har vi nå vist at 7 ikke kan forekomme mellom to romlige regioner. Vi står da tilbake med de 9 relasjonene r_0, r_1, r_3, r_6, r_7, r_{10}, r_{11}, r_{14} og r_{15}.

Tolkning av topologiske relasjoner

Definisjon 18 *Deskriptive termer for de 9 topologiske relasjoner mellom to romlige regioner er gitt i tabell 3.2.*

Tabell 3.2: Terminologi for de 9 topologiske relasjoner mellom to romlige regioner.

relasjon	$(\partial \cap \partial,\ ^\circ\cap^\circ,\ \partial\cap^\circ,\ ^\circ \cap \partial)$	terminologi
r_0	$(\emptyset, \emptyset, \emptyset, \emptyset)$	A og B er disjunkte
r_1	$(\neg\emptyset, \emptyset, \emptyset, \emptyset)$	A og B berører hverandre
r_3	$(\neg\emptyset, \neg\emptyset, \emptyset, \emptyset)$	A og B er identiske
r_6	$(\emptyset, \neg\emptyset, \neg\emptyset, \emptyset)$	A er innenfor B eller B inneholder A
r_7	$(\neg\emptyset, \neg\emptyset, \neg\emptyset, \emptyset)$	A er dekket av B eller B dekker A
r_{10}	$(\emptyset, \neg\emptyset, \emptyset, \neg\emptyset)$	A inneholder B eller B er innenfor A
r_{11}	$(\neg\emptyset, \neg\emptyset, \emptyset, \neg\emptyset)$	A dekker B eller B er dekket av A
r_{14}	$(\emptyset, \neg\emptyset, \neg\emptyset, \neg\emptyset)$	A og B overlapper med disjunkte omriss
r_{15}	$(\neg\emptyset, \neg\emptyset, \neg\emptyset, \neg\emptyset)$	A og B overlapper med skjærende omriss

Vi skal foreta en sammenlikning av terminologien for de 9 topologiske relasjonene med terminologi fra mengdelæren.

Dersom den topologiske relasjonen mellom A og B er r_0, er det uten videre klart at $A \cap B = \emptyset$. Det topologiske begrepet *er disjunkte* svarer derfor til mengdelærens begrep *disjunkte*.

De følgende teorem og korollar tar for seg de resterende relasjonene i tabell 3.2.

Teorem 16 *La A og B være romlige regioner i X. Dersom $A^\circ \cap B^\circ \neq \emptyset$ og $A^\circ \cap \partial B = \emptyset$, da er $A^\circ \subset B^\circ$ og $A \subset B$.*

Bevis: A° er sammenhengende. Teorem 12 medfører at B° og $X \setminus B$ er gjensidig separerte. Siden $A^\circ \cap \partial B = \emptyset$, har vi $A^\circ \subset X \setminus \partial B$, som ved å anvende teorem 6 og teorem 7 kan omformes til $A^\circ \subset B^\circ \cup X \setminus B$. Ifølge teorem 9 er enten $A^\circ \subset B^\circ$ eller $A^\circ \subset X \setminus B$. Men $A^\circ \cap B^\circ \neq \emptyset$, derfor $A^\circ \subset B^\circ$. Siden $A^\circ \subset B^\circ$, er $\overline{A^\circ} \subset \overline{B^\circ}$. Ved å anvende definisjon 17 får vi at $A \subset B$. Hvilket skulle vises. □

Teorem 16 sier at dersom A er dekket av B, så er $A \subset B$. De topologiske relasjonene *er dekket av* sammenfaller derfor med mengdelærens begrep *er en delmengde av*.

Følgende korollar til teorem 16 viser at den topologiske relasjonen *er identiske* svarer til mengdelærens begrep *likhet*.

Korollar 2 *La A og B være romlige regioner. Dersom den topologiske relasjonen mellom A og B er r_3, da er $A = B$.*

Bevis: Vi har forutsatt at $A^\circ \cap B^\circ \neq \emptyset$ og $A^\circ \cap \partial B = \emptyset$. Dette gir ifølge teorem 16 at $A \subset B$. Videre er forutsatt at $\partial A \cap B^\circ = \emptyset$. Ved igjen å anvende teorem 16 får vi at $B \subset A$. Følgelig er $A = B$. □

Det neste korollar til teorem 16 viser at den topologiske relasjonen *innenfor* svarer til mengdelærens begrep *er innbefattet i den indre*.

Korollar 3 *La A og B være romlige regioner. Dersom den topologiske relasjonen mellom A og B er r_6, da er $A \subset B^\circ$.*

Bevis: Teorem 16 fører til at $A^\circ \subset B^\circ$ og $A \subset B$. Av teorem 7 har vi at $A = A^\circ \cup \partial A$ og $B = B^\circ \cup \partial B$. Derfor er $\partial A \subset B$. Siden $\partial A \cap \partial B = \emptyset$, følger det at $\partial A \subset B^\circ$. Sammen med $A^\circ \subset B^\circ$ fører dette til at $\partial A \cup A^\circ = A \subset B^\circ$. □

Resultatet av sammenlikningene er sammenstilt i tabell 3.3.

3.6.3 Topologiske relasjoner i n-dimensjonale rom

Vi har nylig vist at av de 16 teoretisk mulige binære relasjoner, er det bare 9 som kan forekomme mellom romlige regioner. Man kan spørre seg om ytterligere restriksjoner på det topologiske rom eller de mengder som betraktes, vil føre til færre mulige topologiske relasjoner.

La R^n betegne et n-dimensjonalt Evklidisk rom. En *n-celle* i R^n er mengden av alle punkter i R^n som har en avstand fra origo mindre enn eller lik 1.

Vi skal nøye oss med å angi to teorem.

Teorem 17 *Den topologiske relasjonen r_{14}, overlapp med disjunkte omriss, kan ikke eksistere mellom n-celler i R^n med $n \geq 2$*

Teorem 18 *Den topologiske relasjonen r_{15} overlapp med skjærende omriss, kan ikke eksistere mellom romlige regioner i R^1.*

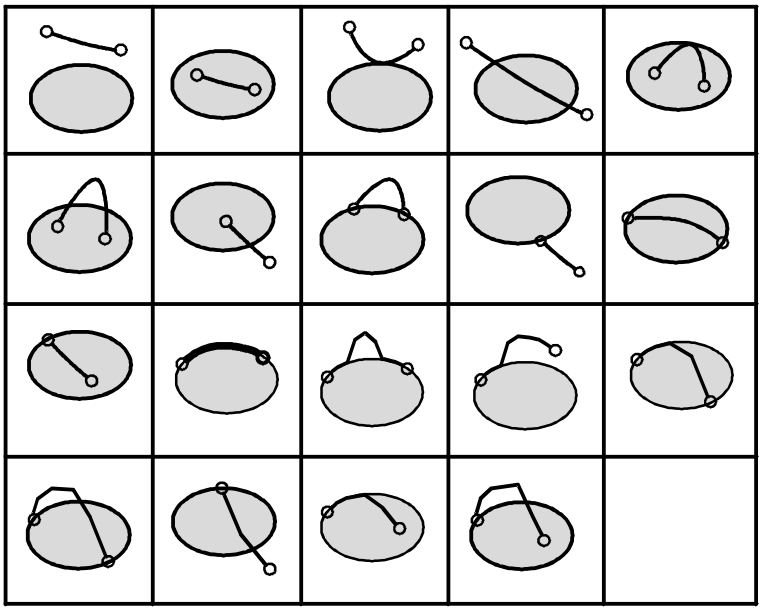
Tabell 3.3: Sammenlikning av terminologi for topologiske relasjoner med begreper fra mengdelæren.

relasjon	terminologi	mengdelærens notasjon
r_0	A og B er disjunkte	$A \cap B = \emptyset$
r_1	A og B berører hverandre	
r_3	A og B er identiske	$A = B$
r_6	A er innenfor B eller B inneholder A	$A \subset B^\circ$
r_7	A er dekket av B eller B dekker A	$A \subset B$
r_{10}	A inneholder B eller B er innenfor A	$B \subset A^\circ$
r_{11}	A dekker B eller B er dekket av A	$B \subset A$
r_{14}	A og B overlapper med disjunkte omriss	
r_{15}	A og B overlapper med skjærende omriss	

3.6.4 9-snittmodellen

Egenhofer og Franzosa's [EF91] metode for å klassifisere binære topologiske relasjoner kalles 4-snittmodellen (eng. the 4-intersection), og metoden baserer seg som kjent på de fire snitt mellom omriss og indre til de to mengdene. På grunn av at 4-snittmodellen ikke tar i betraktning den mengde som ligger utenfor objektene, er den noe utilstrekkelig for modellering av relasjoner der linjer inngår. 9-snittmodellen [EH91] er en utvidelse of 4-snittmodellen til en modell som også tar i betrakting den punktmengden som ligger utenfor A og B. Modellen representeres ved de 9 snitt det er mulig å danne på basis av omriss, indre og ytre til A og B. Teoretisk kan 9-snittmodellen skille mellom $2^9 = 512$ forskjelling topologiske relasjoner, men bare en del av disse relasjonene lar seg realisere for 0-, 1-, og 2-dimensjonale objekter i R^2. Figur 3.9 illustrere de 19 relasjoner som lar seg realisere for en linje og et areal. Det må understrekes at 9-snittmodellen definerer omrisset til en linje i et plan som dens endepunkter, men i følge topologien er omrisset til en linje i R^2 lik hele linja. Omrisset til et linjesegment i et 1-D rom er endepunktene til linjesegmentet. Den øvrige delen av linja er dens indre. Derimot i det tilfellet at det underliggende rommet er 2-dimensjonalt, blir den indre til linjesegmentet en tom mengde og omrisset til linjesegmentet lik linjesegmentet. Grunnen til at 9-snittmodellen overfører definisjonene fra 1-D til 2-D, er at dette øker det antall relasjoner modellen kan skille mellom.

Selv om det finnes et utall topologiske relasjoner [EF93], klassifiserer likevel 9-snittmodellen mengden av topologiske relasjoner i noen hovedklasser som reflekterer hvordan vi ofte tenker om relasjoner mellom romlige objekter. En undersøkelse av Mark og Egenhofer [ME94] viser at 9-snittmodellen kan beskrive store deler av meningsinnholdet i romlige begreper som krysser", går gjennom", går langsetc. Begreper som for eksempel går langs", tar også sitt meningsinnhold fra andre geometriske komponenter enn de topologiske (form og avstand).



Figur 3.9: De nitten topologiske relasjonene i 9-snittmodellen som lar seg realisere for en enkel linje og et areal.

3.6.5 Relasjoner mellom objekter som har uskarpe avgrensninger

4- og 9-snittmodellen til Egenhofer og medforfattere forutsetter at objektene har skarpe avgrensninger og veldefinerte indre. I den senere tiden har forskningsmiljøene fattet stor interesse for modellering av objekter med uskarpe avgrensninger. Topologiske relasjoner mellom slike objekter blir selvsagt mere komplisert å modellere enn for skarpt avgrensede objekter, se for eksempel [Bjø95].

Kontrollspørsmål

1. Hva menes med topologien på en mengde i et metrisk rom?

2. Er en tett skive og en skive med hull homeomorfe?

3. Er eiendomsteiger i Norge homeomorfe? Dersom nei, gi eksempler på eiendomsteiger som ikke er homeomorfe.

4. Ser du noen nytte av støtte seg på begrepet homeomorfe figurer ved datamodellering for GIS?

5. Beskriver en landmåler geografiske objekter i et metrisk rom?

6. Hvilken metrikk brukes vanligvis i en kartprojeksjon? Enn på rotasjonsellipsoiden?

7. Vil du definere en eiendomsteig som en åpen eller en lukket mengde?

8. Er arealet til en eiendomsteig en topologisk egenskap den har?

9. Nevn tre topologiske egenskaper til en eiendomsteig.

10. Hva menes med sammenhengende mengder?

11. Definer en eiendomsteig ved hjelp av topologiske begreper. Tenk igjennom hvilke tester som må legges inn for å sikre at ditt GIS gir en beskrivelse som er i samsvar med foregående definisjon.

12. Hvilken metode kan benyttes for å beskrive topologiske relasjoner mellom geografiske objekter?

13. Hvilke topologiske relasjoner vil du tillate i en eiendomsdatabase?

14. Hvilke topologiske relasjoner vil du tillate i en database over skog, vann og veier? Er det mulig å modellere dette i de GIS du kjenner?

Kapittel 4

FLYTENDE MENGDER

Vi ser i dag at geografiske informasjonssystemer (GIS) utvikler seg fra kartsystemer til systemer for romlig beslutningsstøtte. Dette reiser i stigende grad behovet for å kunne håndtere upresise betingelser eller beskrivelser av objekter med utflytende avgrensninger. Det er for eksempel i en rekke av dagens GIS tilgjengelig funksjoner som kan gi oss svar på spørsmål av typen finn korteste vei fra A til B i: lengdemål, tidsmål eller i kjørekostnader. I framtida kan vi imidlertid se for oss systemer som kan svare på spørsmål av typen: finn triveligste rutevalg, finn minst stressende rutevalg eller finn mest komfortable rutevalg. Den siste kategorien betingelser er subjektive mens den første kategorien betingelser har en objektiv basis. Et annet eksempel er: (1) tegn ut alle hoteller som ligger innenfor Lillehammer kommune eller (2) tegn ut alle hoteller som ligger i Lillehammerområdet. I det første tilfellet har vi en presis angivelse av et geografisk område mens vi i det andre tilfellet har angitt et område med en utflytende avgrensning. Området kan nok sies å ha en kjerne som definitivt tilhører Lillehammerområdet, men det har også en yttersone der det faller naturlig å snakke om graden av tilhørighet til Lillehammerområdet.

I dagens GIS er det vanlig å basere datamodellene på en oppfatning av at verden består av forekomster som har skarpe avgrensninger eller jevn konsentrasjon av attributtene, men som vi nylig har vist, en slik betraktningsmåte vil kunne komme til kort. Egentlig er det vel slik at de fleste geografiske forekomster har uskarpe avgrensninger og at det nesten bare er for eiendomsteiger vi kan snakke om skarpe avgrensninger. Derfor vil de skarpt avgrensede objekter i dagens GIS ofte representere forenklinger av de underliggende geografiske forekomster. Teorien om flytende mengder (*eng. fuzzy set theory*) tilbyr et begrepsapparat som kan understøtte modelleringen av den typen forekomster som i det foregående er skissert. Vi vil i dette kapitlet gi en oversikt over teoriens grunnleggende elementer og belyse teorien med noen GIS-relaterte eksempler.

4.1 Elementer fra teorien om flytende mengder

Vi vil benytte norske ord for begrepene innen *fuzzy set theory*. Tabell 4.1 viser de oversettelser vi benytter. Begrepet fuzzy kan oversettes med loet, tåket eller uklar.

Tabell 4.1: Engelske og norske begreper

Engelsk	**Norsk**
$\alpha-$cut	$\alpha-$kutt
Characteristic function	Karakteristisk funksjon
Crisp sets	Skarpe mengder
Membership function	Medlemskapsfunksjon
Fuzzy numbers	Flytende tall
Fuzzy relations	Flytende relasjoner
Fuzzy sets	Flytende mengder
Fuzzy set theory	Teorien om flytende mengder
Possibility theory	Mulighetsteori
Probability theory	Sannsynlighetsteori
Support	Tilslutning

Norsk språkråds komité for dataterminologi [HLS93] foreslår at fuzzy sets og fuzzy logic oversettes med flytende mengde og flytende logikk. Vi vil rette oss etter denne anbefalingen.

Teorien om og anvendelser av flytende mengder har blitt godt dokumentert i litteraturen siden Zadeh's artikkel i 1965. Det finnes flere utmerkede lærebøker på området, for eksempel [KF88] og [KG91]. Selv om det finnes flere bidragsytere, regnes Lotfi A. Zadeh for å være grunnleggeren av teorien om flytende mengder. Teorien har utviklet flere retninger: flytende mengder, flytende tall, flytende logikk og mulighetsteori. Læren om flytende mengder kan lett forveksles med sannsynlighetsteori, men det er grunnleggende forskjeller. Sannsynlighetsteori baserer seg på måling av random hendelser, mens læren om flytende mengder beskriver i hvilken grad et element tilhører en gitt mengde. La oss anta en person som ønsker å overnatte på hotell i nærheten av Lillehammer, og at vedkommende får velge mellom et hotell i Øyer og et hotell i Moelven. La oss videre anta at vedkommende er ukjent med disse to stedsnavnene og har skaffet seg følgende informasjon: (1) sannsynligheten for at Øyer ligger i Lillehammerområdet er 0,8 og (2) Moelven har en tilhørighet til Lillehammerområdet på 0,8. Hvilket hotell vil vedkommende velge? I det første tilfellet kan vi frykte at det er 20% sannsynlighet for at Øyer ligger et helt annet sted i verden enn i nærheten av Lillehammer, mens vi i det andre tilfellet neppe risikerer mere enn at det blir noen Km å kjøre inn til Lillehammer. Forskjellen blir enda klarere om vi får vite at sannsynligheten for at Moelven ligger i Lillehammerområdet er 100%, men Moelven's tilhørighet til Lillehammerområdet er fortsatt bare 0,8.

4.1.1 Skarpe mengder

La X betegne en universell mengde. Den *karakteristiske* funksjonen til en *skarp mengde* A tilordner en verdi $\mu_A(x)$ til hver $x \in X$ slik at:

$$\mu_A(x) = \begin{cases} 1 & x \in A \\ 0 & x \notin A \end{cases}$$

Dette kan skrives som:

$$\mu_A : X \to \{0, 1\}$$

4.1.2 Flytende mengder og operasjoner på disse

Medlemskapsfunskjonen μ_A til en *flytende mengde* A har formen:

$$\mu_A : X \to [0, 1]$$

hvor $[0, 1]$ betegner intervallet av reelle tall fra og med 0 til og med 1.

Tilslutningen til en flytende mengde A i den universelle mengde X er en skarp mengde som inneholder alle elementer i X som har en grad av medlemskap i A større enn null, altså:

$$\text{supp } A = \{x \in X \mid \mu_A(x) > 0\}$$

En notasjon som ofte benyttes i litteraturen for å definere flytende mengder med en endelig tilslutning, er:

$$A = \mu_1/x_1 + \mu_2/x_2 + \cdots + \mu_n/x_n = \sum_{i=1}^{n} \mu_i/x_i$$

(skrivemåten må ikke forveksles med aritmetiske tegn som summetegn og brøkstrek).

For flytende mengder med en kontinuerlig tilslutning over et intervall av reelle tall, byttes tegnet $\sum$ med tegnet $\int$ og vi skriver:

$$A = \int_X \mu_A(x)/x$$

(skrivemåten må ikke forveksles med integralregning).

Høyden til en flytende mengde er lik den høyeste grad av medlemskap som er tilordnet noe element i mengden. Formelt:

$$h(A) = \max[\mu_1, \mu_2, \ldots, \mu_n] \tag{4.1}$$

En mengde kalles normalisert når minst ett av dets elementer oppnår maksimal grad av medlemskap.

Et *α-kutt* til en flytende mengde A er en skarp mengde A_α som inneholder alle elementer i den universelle mengden X som har en grad av medlemskap i A større enn eller lik den angitte verdien α:

$$A_\alpha = \{x \in X \mid \mu_A(x) \geq \alpha\} \tag{4.2}$$

Standardoperasjoner som union, snitt og komplement er også definert for flytende mengder. Gitt to flytende mengder A and B som er definert over et univers X. Vi har da:

$$\begin{array}{lll} \text{Komplement:} & \mu_{A^-}(x) & = 1 - \mu_A(x) \\ \text{Union:} & \mu_{A\cup B}(x) & = \max\left[\mu_A(x), \mu_B(x)\right] \\ \text{Snitt:} & \mu_{A\cap B}(x) & = \min[\mu_A(x), \mu_B(x)] \end{array} \tag{4.3}$$

Vi skal imidlertid være klar over at snitt- og unionoperatorene kan defineres på andre måter enn den enkle metoden som er angitt i likning 4.3. Det forlanges imidlertid at disse operatorene skal oppfylle en del aksiomer. Dette sikrer at operatorene blant annet har kommutative ($a{\cdot}b = b{\cdot}a$), kontinuerlige og assosiative egenskaper ($(a{\cdot}b){\cdot}c = a \cdot (b \cdot c)$). Yagerklassen av operatorer kan stå som en representant for denne klassen av operatorer.

$$\begin{array}{lll} \text{Yager komplement:} & c_w(a) & = (1 - a^w)^{1/w} \\ \text{Yager union:} & u_w(a,b) & = \min\left[1, (a^w + b^w)^{1/w}\right] \\ \text{Yager snitt:} & i_w(a,b) & = 1 - \min\left[1, ((1-a)^w + (1-b)^w)^{1/w}\right] \end{array} \tag{4.4}$$

I likning 4.4 er størrelsen w, som kan være et hvilket som helst tall i intervallet $[0, \infty]$, en brukerstyrt parameter som benyttes til å bestemme egenskapene til operatorene. For eksempel dersom $w \to \infty$, vil Yager union og Yager snitt nærme seg de enkle union- og snittoperatorene i likning 4.3. Figur 4.1 illustrerer hvordan valg av verdier for w virker inn på union- og snittberegninger.

Yager union

a= \ b=	0	.25	.5	.75	1
1	1	1	1	1	1
.75	.75	1	1	1	1
.5	.5	.75	1	1	1
.25	.25	.5	.75	1	1
0	0	.25	.5	.75	1

w=1

a= \ b=	0	.25	.5	.75	1
1	1	1	1	1	1
.75	.75	.75	.75	.8	1
.5	.5	.5	.54	.75	1
.25	.25	.27	.5	.75	1
0	0	.25	.5	.75	1

w=10

Yager snitt

a= \ b=	0	.25	.5	.75	1
1	0	.25	.5	.75	1
.75	0	0	.25	.5	.75
.5	0	0	0	.25	.5
.25	0	0	0	0	.25
0	0	0	0	0	0

w=1

a= \ b=	0	.25	.5	.75	1
1	0	.25	.5	.75	1
.75	0	.25	.5	.73	.75
.5	0	.25	.46	.5	.5
.25	0	.20	.25	.25	.25
0	0	0	0	0	0

w=10

Figur 4.1: Eksempler på union- og snittoperatorer fra Yagerklassen

Som vi ser av figuren, vil valget av verdi for w avgjøre i hvilken grad beregningene trekkes mot ytterverdiene 0 og 1. Jo mindre verdi vi velger for w, jo mere vil unionen trekkes mot verdien 1 og snittet mot verdien 0. Problemet i de praktiske anvendelser av Yagerklassen, er å fastsette egnede verdier for w. Som en illustrasjon av hvordan Yagerklassen kan benyttes, kan vi si at vi stiller et krav om at vi ønsker å ha oppfylt to betingelser. Den ene betingelsen kan være at vi skal finne hoteller med billige rompriser og den andre betingelsen kan være at vi ønsker en beliggenhet i Lillehammerområdet. Dersom vi er veldig strenge på betingelsene, kan vi velge å si at dersom begge betingelsene i liten grad er tilfredsstilt, er totalløsningen så dårlig at vi ikke ønsker denne løsningen. Dette kan modelleres ved et Yager snitt med en liten verdi for w, for eksempel $w = 1$. På den andre siden kan vi også velge å si at ut fra våre behov er løsningen like svak som den betingelsen som er svakest oppfylt. Dette tilsvarer en høy verdi for w i snittberegningen, for eksempel $w = 100$.

Flytende mengder kan også kombineres ved hjelp av *middeltallsberegninger*. En klasse av slike operatorer som dekker hele intervallet mellom de enkle union- og snittoperatorene, består av de generaliserte middeltall (se figur 4.2):

$$h_\alpha(a_1, a_2, \ldots, a_n) = \left(\frac{a_1^\alpha + a_2^\alpha + \cdots, a_n^\alpha}{n}\right)^{1/\alpha} \tag{4.5}$$

Dersom for eksempel α velges lik -1,0 eller 1, får vi henholdsvis det harmoniske, det geometriske og det aritmetiske middeltal. Når den aktuelle anvendelsen krever det, kan det innføres vekter i middeltallsberegningen. For eksempel dersom vi velger å si at nærhet til Lillehammerområdet teller mere enn billig hotell, kan dette modelleres ved hjelp av vekter:

$$h_\alpha(a_1, a_2, \ldots, a_n; w_1, w_2, \cdots, w_n) = \left(\sum_{i=1}^{n} w_i a_i^\alpha\right)^{1/\alpha} \tag{4.6}$$

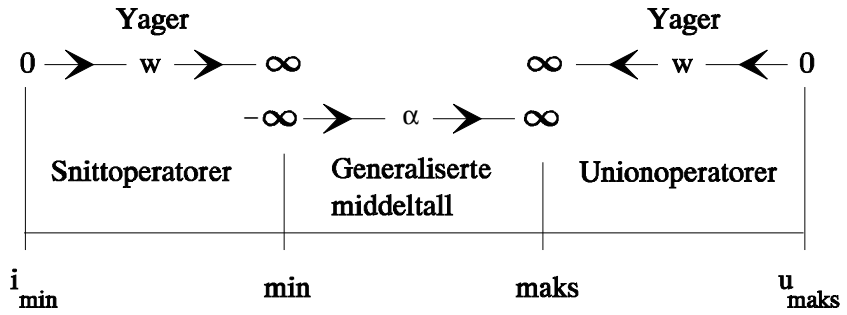
hvor summen av vektene forutsettes å være lik 1.

Forholdet mellom union/snitt-operatorer og middeltalsberegninger er illustrert i figur 4.2. De ordinære union og snittoperatorene på flytende mengder er markert ved maks og min punktene i figuren. Ved kombinasjon av to eller flere flytende mengder, må det velges operatorer og parametre som passer i den aktuelle situasjonen. Dette valget er ikke alltid opplagt, og det vil derfor i et GIS være behov for at bruker kan styre valget av operatorer og parameterverdier.

4.1.3 Flytende relasjoner

Produktmengden til to skarpe mengder X og Y, skrives som $X \times Y$ og er mengden av alle ordnede par slik at det første elementet i hvert par er medlem i X og det andre elementet er medlem i Y. Formelt:

$$X \times Y = \{(x, y) \mid x \in X \text{ og } y \in Y\}$$



Figur 4.2: Virkemåten til aggregeringsoperatorer på flytende mengder

En *relasjon* mellom X og Y er en delmengde i $X \times Y$ og skrives som $R(X, Y)$. På lignende måte som for mengder kan vi operere med skarpe relasjoner og flytende relasjoner. *En skarp relasjon* R mellom X og Y kan defineres ved en karakteristisk funksjon som tilordner verdien 1 til hvert par i den universelle mengden $X \times Y$ som tilhører relasjonen og verdien 0 til hvert par som ikke gjør det. Følgelig:

$$\mu_R(x, y) = \begin{cases} 1 & \text{hvis og bare hvis } (x, y) \in R \\ 0 & \text{ellers} \end{cases}$$

En *flytende relasjon* mellom X og Y er en flytende mengde definert over $X \times Y$ hvor paret (x, y) kan ha varierende grad av tilhørighet i relasjonen. Graden av medlemskap, som angis ved et reelt tall i det lukkede intervallet $[0, 1]$, indikerer styrken til relasjonen mellom elementene i paret. For eksempel kan vi la X bety mengden av alle fylker i Norge og Y bety mengden av alle elver i Norge. La oss så anta relasjonen: renner gjennom". For elementet Gudbransdalslågen"vil vi kunne si at tilhørigheten i relasjonen renner gjennom Oppland fylke"har graden 1.0. Derimot vil vi si at tilhørigheten i relasjonen renner gjennom Hedmark fylkeer null eller svært liten. Usikkerheten kan stikke i om vi regner nordre deler av Mjøsa som en del av Lågen og om sidebekker til Lågen som har forgreininger inn i Hedmark fylke, er en del av Lågen. La oss si at vi setter graden av medlemskap i denne relasjonen til 0.05.

[Alt94] definerer en *flytende region* som en binærrelasjon over mengden av naturlige tall, gitt ved:

$$R = \{\mu_R(x, y)/(x, y) \mid x, y \in N\}$$

hvor graden av medlemskap $\mu_R(x, y)$ angir graden av konsentrasjon av en eller annen attributt i punktet (x, y). Det er selvsagt ikke noe i veien for å definere en flytende region over mengden av reelle tall, men Altman opererer i sine eksempler på rasterbaserte kart, så derfor ble naturlige tall valgt.

4.1.4 Flytende tall

Flytende tall representerer en spesiell klasse av medlemskapsfunksjoner. Et flytende tall i R er en flytende delmengde i R som er konveks og normal, se [KG91]. En

flytende mengde $\mathbf{A} \subset \mathbf{R}$ er konveks hvis og bare hvis hvert ordinære A_α er et lukket intervall i R. Med andre ord, et uskarpt tall har en medlemskapsfunksjon som har bare ett toppunkt og ingen bunnpunkt. En annen definisjon som benyttes er:

$$\mu_A[\lambda x_1 + (1-\lambda)x_2] \geq \mu_A(x_1) \wedge \mu_A(x_2) \quad \forall\lambda \quad \in [0,1]$$

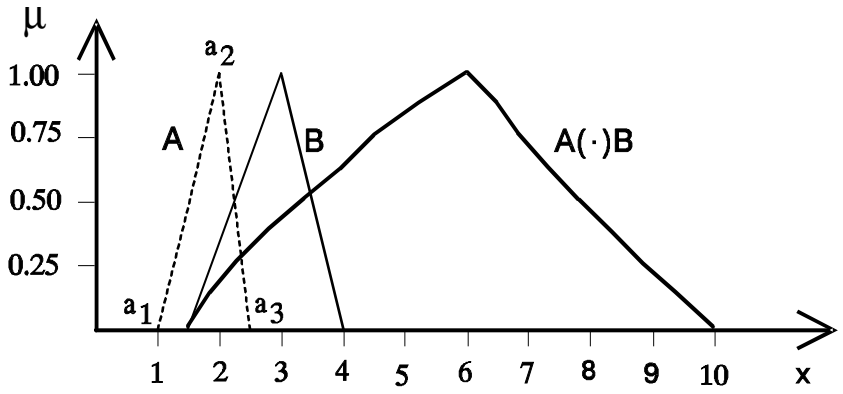
Vi kan betrakte et uskarpt tall som en generalisering av de såkalte konfidensintervall i statistikken. Sannynlighetsteorien, som har utviklet seg fra måleteori, baserer seg imidlertid på et objektivt datum. Derimot baserer flytende tall seg på et subjektivt datum. Flytende tall representerer ofte anslag og er derfor gjerne ikke framkommet ved nøyaktige målinger. Likevel finnes det forbindelser mellom random variable og flytende tall som kan være nyttige for praktiske anvendelser, se [KG91] for nærmere detaljer.

[KG91] introduserer en type tall de kaller for *hybride tall*. Dette er en matematisk konstruksjon som kan holde informasjon om både random variable og flytende tall. Vi vil ikke her gå nærmere inn på hybride tall, men bare nevne at de representerer en addisjon av et flytende tall til en random variabel. Hybride tall er absolutt interessante i kartsammenheng, fordi vi her har å gjøre med innmåling av objekter som ofte har uskarpe avgrensninger. I slike tilfeller har vi derfor både en uskarp komponent og en statistisk komponent. Hybride tall kan derfor tenkes å bli anvendt i kvalitetsmål ved kartlegging av geografiske forekomster.

Triangulære flytende tall (TFN) er en spesiell klasse flytende tall. Et TFN er fullstendig beskrevet ved tre tall

$$\mathbf{A} = (a_1, a_2, a_3)$$

slik som vist i figur 4.3.



Figur 4.3: Multiplikasjon av to TFN'er vil ikke gi et nytt TFN. I figuren tilnærmer $A(\cdot)B$ et TFN, men grafen avviker i noen grad fra en fullkommen triangulær form.

I løpet av de siste ti årene har det blitt utviklet en aritmetikk (addisjon, subtraksjon, multiplikasjon og divisjon) på flytende tall. Egenskapene konveks og normal

spiller en viktig rolle i definisjonen av denne aritmetikken. Dersom de flytende tallene er triangulære, forenkles definisjonen av de aritmetiske operasjonene. Addisjon og subtraksjon av to TFN'er **A** og **B** defineres som:

$$\mathbf{A}(+)\mathbf{B} = (a_1 + b_1, a_2 + b_2, a_3 + b_3)$$

$$\mathbf{A}(-)\mathbf{B} = (a_1 - b_3, a_2 - b_2, a_3 - b_1)$$

Multiplikasjon med et ordinært tall defineres som:

$$k(\cdot)\mathbf{A} = (ka_1, ka_2, ka_3)$$

Siden $\mathbf{A}(\cdot)\mathbf{B}$ eller $\mathbf{A}(:)\mathbf{B}$ generelt ikke er triangulære TFN'er, som vist i figur 4.3, anvendes max-min konvolusjonen på flytende tall i definisjonen av disse operatorene:

$$\mu_{\mathbf{A}(\cdot)\mathbf{B}}(z) = \bigvee_{z=x\cdot y} (\mu_{\mathbf{A}}(x) \wedge \mu_{\mathbf{B}}(y))$$

$$\mu_{\mathbf{A}(:)\mathbf{B}}(z) = \bigvee_{z=x/y} (\mu_{\mathbf{A}}(x) \wedge \mu_{\mathbf{B}}(y))$$

Siden denne formelen ikke er egnet for direkte implementasjon på en digital datamaskin, baseres beregningen heller på omrisset til de lukkede intervallen A_α og B_α. La oss anta to flytende tall **A** og **B** i R^+ (merk bare positive tall, for negative tall blir definisjonen litt annerledes). For en gitt verdi av α har vi under nevnte forutsetning:

$$A_\alpha(\cdot)B_\alpha = [a_1^{(\alpha)} \cdot b_1^{(\alpha)}, a_2^{(\alpha)} \cdot b_2^{(\alpha)}]$$

$$A_\alpha(:)B_\alpha = [a_1^{(\alpha)}/b_2^{(\alpha)}, a_2^{(\alpha)}/b_1^{(\alpha)}]$$

Det vises til [KG91] for en utdyping av dette emnet.

La oss så anta to flytende tall **A** og **B**. Dersom

$$a_1^{(\alpha)} \leq b_1^{(\alpha)} \text{ og } a_2^{(\alpha)} \leq b_2^{(\alpha)} \quad \forall\alpha \in [0,1] \tag{4.7}$$

kan vi skrive at $\mathbf{A} \leq \mathbf{B}$. Dersom betingelse 4.7 ikke er tilfredsstilt, er ikke de to flytende tallene sammenlignbare. Flytende tall har ikke fullstendig ordning, de har bare delvis ordning.

4.1.5 Mulighetsteori

Mulighetsteorien har utviklet seg fra teorien om flytende mengder og benyttes blant annet innenfor ekspertsystemer til å håndtere usikker og ufullstendig informasjon [Tim87]. I mulighetsteorien brukes måltallet *mulighet* som kan oppfattes som en generalisering av sannsynlighetsbegrepet. Forskjellen mellom mulighetsteori og sannsynlighetsteori stikker i at sannsynlighetsteorien baserer seg på et noe strengere

additivitets aksiom enn mulighetsteorien [KF88]. Innenfor mulighetsteorien er det utviklet et begrepsapparat som har likhetstrekk med begreper innen statistikken, for eksempel snakkes det om mulighetsfordeling og betinget mulighet.

I mulighetsteorien tilordnes til hver skarpe mengde i den universelle mengden, et tall som angir graden av tro på at et gitt element tilhører mengden. Derimot for flytende mengder tilordnes til hvert element i den universelle mengden et tall som angir graden av tilhørighet i en mengde. Teorien har flere mulige anvendelsesområder innen GIS, for eksempel i systemer for kartografisk generalisering. [KV91] beskriver hvordan fuzzy-teorien er benyttet i systemet FRSIS. FRSIS er et relasjonsbasert informasjonssystem for jordbunnens beskaffenhet.

4.2 Eksempler

Vi vil her demonstrere bruken av teorien om flytende mengder på noen GIS relaterte eksempler.

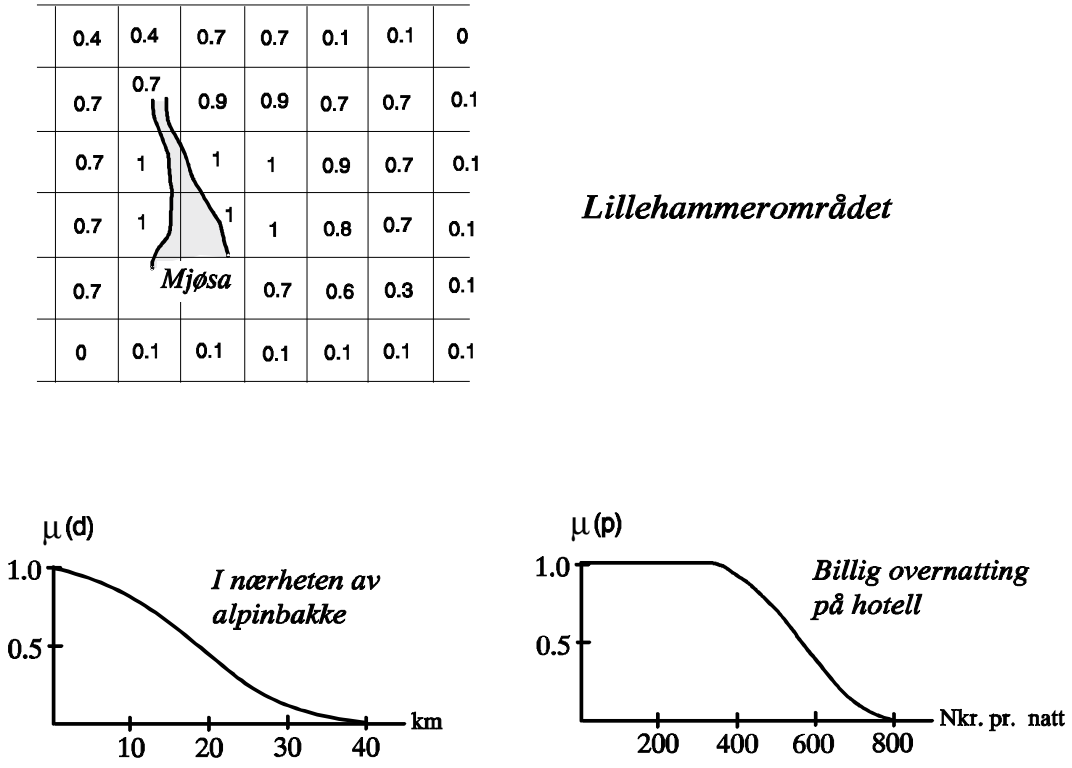
4.2.1 Turist-GIS

La oss anta en utenlandsk familie som planlegger en ferietur i Norge. Ved tilkopling til Internett har de tilgang til en turistdatabase som kan hjelpe dem i planleggingen. Familien prøver seg i første omgang med følgende spørsmål: gi oss hoteller i Lillehammerområdet som både har billige priser på overnatting og som ligger i nærheten av alpinanlegg. For å kunne svare på dette spørsmålet, må vi definere hva vi mener med Lillehammerområdet, billige priser og nære alpinanlegg. Figur 4.4 illustrerer hvordan disse utsagnene lar seg modellerer. Figuren viser at Lillehammerområdet er modellert som en flytende region ved hjelp av et raster. Hver rute i rasteret er tilordnet en tallverdi som representerer graden av medlemskap i regionen Lillehammerområdet. Videre er de to andre utsagnene modellert ved hver sin medlemskapsfunksjon. For eksempel har vi sagt at vi definitivt har billig overnatting dersom prisen pr. natt er lavere enn 400 kroner. Dersom prisen ligger over 800 kroner pr. natt, er prisen avgjort ikke billig. Det ville vært mulig å modellert kostbar overnatting som komplementet til billig overnatting, men riktigheten av dette vil avhenge av hva brukerne legger i begrepet kostbar overnatting. Medlemskapsfunksjonen "i nærheten av alpinbakkesier at nærhet forutsetter at avstanden ikke er større enn 40km. De modellene som er skissert i figur 4.4, kan selvsagt diskuteres. For en konkret anvendelse vil det være nødvendig å basere modellene på visse brukerundersøkelser.

La oss etablere tre flytende mengder L, S og P som betyr henholdsvis hoteller i Lillehammerområdet, hoteller nær alpinbakke og hoteller med billig overnatting:

$$L = \sum_{x \in X} \mu_L(x)/x; \qquad S = \sum_{x \in X} \mu_S(x)/x; \qquad P = \sum_{x \in X} \mu_P(x)/x;$$

X betyr her mengden av hoteller i Norge. Vi antar at vi bare har fem hoteller som tilfredsstiller kravet $\mu_L(x) > 0$. Disse hotellene kaller vi $\{a, b, c, d, e\}$. Ved å avlese



Figur 4.4: Medlemskapsfunksjoner i turist-GIS

på de aktuelle medlemskapsfunksjonene, antar vi følgende flytende mengder:

$$\begin{aligned} L &= 0,2/a + 0,9/b + 0,6/c + 0,1/d + 1,0/e \\ S &= 0,4/a + 0,1/b + 0,7/c + 0,2/d + 0,3/e \\ P &= 0,1/a + 0,9/b + 0,1/c + 0,9/d + 0,6/e \end{aligned}$$

Det må imidlertid bemerkes at det finnes flere alpinanlegg i det aktuelle området, men slik spørsmålet er stilt av vår familie, er det tilstrekkelig å plukke ut det nærmeste alpinanlegget til hvert av hotellene. Vi kan nå vende tilbake til det opprinnelige spørsmålet og se på hvilke operatorer som kan komme på tale. En opplagt løsning er å basere seg på den enkle snittberegningen gitt i likning 4.3. Dette gir oss:

$$\begin{aligned} R &= L \cap S \cap P \\ &= 0,1/a + 0,1/b + 0,1/c + 0,1/d + 0,3/e \end{aligned}$$

Ved så å benytte likning 4.1, finner vi at høyden til R er 0,3. Et $\alpha-$kutt (likning 4.2) for denne verdien gir:

$$R_{0,3} = \{e\}$$

Med andre ord kan vi rapportere tilbake at hotell e er det hotellet som best tilfredsstiller alle tre kravene.

Som et alternativ til denne enkle snittberegningen, vil vi nå prøve oss med Yager snitt, først med $w = 1$ og deretter med $w = 5$, se likning 4.4. Dette gir oss:

$$\begin{aligned} R_1 &= 0/a + 0/b + 0/c + 0/d + 0/e \\ R_5 &= 0/a + 0,1/b + 0,1/c + 0,02/d + 0,29/e \end{aligned}$$

Som vi ser, gir $w = 1$ et noe strengt snitt siden vi må rapportere tilbake at ingen hoteller tilfredsstiller kravene. Derimot slipper hotellene b, c, d, e igjennom for $w = 5$. Altså dette snittet er ikke fullt så strengt som snittet i det første tilfellet. Yagersnittet $w = 5$ vurderer hotellene b, c som litt bedre løsninger enn hotellene a, d. Derimot i den enkle snittberegningen ble disse fire hotellene vurdert som likeverdige. Dette kommer av at Yagersnittet sier at det er tross alt bedre med en løsning der i alle fall noen av betingelsene er godt oppfylt enn om alle betingelsene er like dårlig oppfylt.

Vi tenker oss at vår familie ikke er fornøyd med noen av de svarene vi hittil har rapportert tilbake, og at systemet foreslår at familien tenker igjennom kravene sine pånytt og ser om det kan gis lettelser i noen av betingelsene. Dette resulterer i at spørsmålet modifiseres i retning av at nærhet til Lillehammer gis høy vekt i forhold til de to andre betingelsene. Siden vekter ikke lar seg direkte føre inn i snittberegningen, velger vi denne gangen å gjennomføre en middeltallsberegning over de tre betingelsene. La oss anta at vektene gis slik:

$$w_L = 0,6; \qquad w_S = 0,2; \qquad w_P = 0,2$$

Vi anvender likningene 4.5 og 4.6, setter $\alpha = 1$ og får den aggregerte flytende mengden:

$$R = 0,22/a + 0,74/b + 0,52/c + 0,28/d + 0,78/e$$

Hotellene b, e kommer her ut som vinnere med hotell e som det beste. La oss anta at familien fremdeles ikke er helt sikre på hva de skal velge og finner ut at egentlig vil de holde fast på betingelsen Lillehammerområdet, men at de vil være fornøyd dersom i det minste den ene av de to andre betingelse blir oppfylt. Dette gir oss en grad av frihet som nå modelleres ved at en snittoperator byttes ut med en unionoperator. Basert på de enkle operatorene får vi:

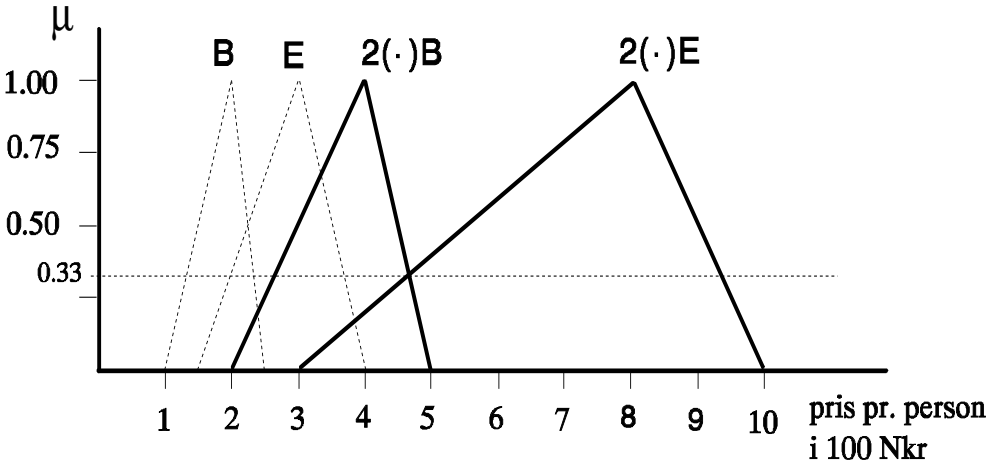
$$\begin{aligned} R &= L \cap (S \cup P) \\ &= 0,2/a + 0,9/b + 0,6/c + 0,1/d + 0,6/e \end{aligned}$$

Denne gangen kommer hotell b ut som en klar vinner.

Det kan kanskje synes frustrerende at ikke én enkelt operator og én enkelt parameterverdi klart peker seg ut som de størrelser beregningen skal basere seg på. Vi skal imidlertid være klar over at det eksemplet egentlig illustrerer, er at den menneskelige måten å tenke på er meget kompleks og at når vi gjør en beslutning, er det en rekke faktorer og relasjoner mellom disse vi evaluerer. Teorien om flytende mengder tilbyr imidlertid et formalisert rammeverk vi kan benytte for å redusere

kompleksitet. Begreper som billig hotell, nær alpinbakke og Lillehammerområdet representerer nettopp forenklinger av underliggende faktorer vår familie vektlegger ved planleggingen av sin ferietur. Siden vi ikke har en fullstendig modell av alle de forhold vår familie vektlegger, alle de hensyn som skal tas, de avbalanseringer som må gjøres, kan vi heller ikke vente å komme opp med et entydig svar. Det kan jo også tenkes at vår familie endrer oppfatning i løpet av prosessen, fordi etter hvert som de skaffer seg bedre innsikt i de ulike valgmuligheter, bør det ikke komme uventet på oss om de endrer vektsforholdet mellom faktorene. Ventelig vil noe av utfordringen i framtidas GIS ligge i det å lage brukergrensenitt som samtidig skjuler kompleksiteten i de oppgaver som skal løses og som er så fleksible og så enkle å bruke at systemene oppfattes som slagkraftige verktøy til å gjøre resonnement om geografiske forekomster. Teorien om flytende mengder kan betraktes som et verktøy til å redusere kompleksitet ved resonnement om geografiske forekomster.

Vi antar at vår familie har bestemt seg for å satse enten på hotell b eller på hotell e, og at de ønsker å trekke inn prisen på besøk i hotellenes restauranter i det endelige valget av hotell. Figur 4.5 modellerer hva vi kan kalle "vanlige kostnader pr. person ved restaurantbesøk". Flytende tall er benyttet. Siden vi forutsetter at vår famile gjør to restaurantbesøk, er kostnadene multiplisert opp med faktoren 2. Av figuren ser vi at hotell b stort sett kommer rimeligere ut enn hotell e, men i



Figur 4.5: Vanlige kostnader pr. person ved besøk i hotellenes restauranter. B og E representerer henholdsvis hotellene b og e.

henhold til likning 4.7 kan vi ikke uten videre trekke den slutningen at vår familie vil komme rimeligere unna et besøk i hotell b sin restaurant enn et besøk i hotell e sin restaurant. Det er først når vi legger inn en terskelverdi $\alpha \geq 0.33$ at vi kan gjøre en slik slutning. Etter dette kan vi rapportere tilbake: dersom dere unngår de dyreste rettene i restaurant b og dersom vi kan anta at dere ikke vil velge de aller rimeligste rettene i hotell e, vil restauranten i hotell b falle rimeligere enn restauranten i hotell e."

4.2.2 Topologiske relasjoner

Dersom vi benytter teorien om flytende mengder til å modellere avgrensningen til geografiske objekter, vil vi også måtte gå videre mot modellering av begreper om topologiske relasjoner mellom geografiske objekter. [Bjø95, Bjø96] presenterer et konsept for hvordan dette kan la seg gjøre ved å modellere både den indre og omrisset til slike objekter som flytende regioner. Ved å generalisere Egenhofers metode [EF91] for beskrivelse av topologiske relasjoner, etableres så et system for å definere topologiske relasjoner mellom regioner som har uskarpe avgrensninger. [Bjø95, Bjø96] viser også hvordan teorien kan benyttes til å lage forenklede modeller av en elv og dens flomområder.

Kapittel 5

ROMLIGE DATAMODELLER

Virkeligheten er meget kompleks og inneholder en uendelighet av variasjon. For å gjøre det mulig for oss å forstå virkeligheten eller beskrive den i en datamaskin, benyttes modeller. En modell av virkeligheten er ikke identisk med virkeligheten, men uttrykker bare visse egenskaper og forhold ved virkeligheten. Et kart, for eksempel, er en modell av objekter og fenomener som har en romlig utbredelse.

I dette kapitlet skal vi se på noen teknikker for å modellere geografiske objekter med hovedvekt på objektenes romlige egenskaper (geometri og topologi).

5.1 Topologiske og geometriske modeller

Definisjon 19 *En topologisk modell beskriver topologiske egenskaper til geografiske objekter.*

Definisjon 20 *En geometrisk modell beskriver egenskaper til geografiske objekter som form, størrelse, utstrekning, posisjon, lengde og bredde.*

Vi har tidligere vært inne på at et metrisk rom er et topologisk rom der topologien blir indusert av en metrikk. Dette forhold er årsaken til at topologiske relasjoner til geografiske objekter kan utledes fra en geometrisk modell. Derimot kan vi ikke utlede geometriske egenskaper fra en topologisk modell. I praksis opplever vi at topologiske egenskaper til et objekt ikke alltid lar seg korrekt utlede av en geometrisk modell. Dette skyldes unøyaktigheter i representasjonen eller avrundingsfeil i beregningene.

Eksempel: Høyre og venstre veikant skal ikke krysse hverandre, men på grunn av målefeil kan vi risikere at den geometriske beskrivelsen ikke er i samsvar med dette. Den topologiske modellen av veien kan derfor benyttes til å kontrollere og eventuelt korrigere den geometriske modellen.

Eksempel: På enkelte kart er hus tegnet slik at de overlapper veisymbolene. Ved å digitalisere slike kart, vil geometrien måtte rettes opp for at de topologiske relasjoner mellom vei og hus skal bli korrekte.

Eksempel: Dersom vi vet vilke øyer som ligger ute i et vann, har vi informasjon om topologiske relasjoner mellom vann og øyer. Men på dette grunnlaget har vi ikke nok til å beregne arealet av vannet. Det eneste vi kan si er at vannet har en del hull og at arealet kan finnes av det omskrivende polygon minus arealet av øyene.

Eksempel: Et veisystem kan modelleres ved hjelp av veikryss og veier mellom veikryssene. Dersom vi vet hvilke veier som går ut/inn i et kryss, har vi nok informasjon til å finne svar på om det finnes en vei fra kryss A til kryss B. For denne typen spørsmål er det tilstrekkelig med kun en topologisk modell av veinettverket.

Dersom spørsmålet inneholder krav som korteste vei, maksimal bratthet eller minste veibredde, må vi i tillegg til den topologiske modellen også ha en modell av geometriske egenskaper til veisystemet.

Dersom man spør etter alle veier som har en gitt veibredde, er det tilstrekkelig med kun en geometrisk modell av veisystemet.

5.2 Tradisjonelle romlige modeller i GIS

5.2.1 Vektormodeller

I matematikken og fysikken er en vektor en størrelse som har et måltall og en retning. I kartografien defineres en vektormodell av en linje som en mengde av linjesegmenter (vanligvis rette linjer).

Vektormodeller kan være *spaghettimodeller* eller *topologiske* modeller. I en spaghettimodell av en linje er det ikke definert noen rekkefølge av vektorene. Derimot i en topologisk vektor modell, er topologiske relasjoner mellom vektorene definert. Valget mellom spaghettimodell og topologisk modell er applikasjonsavhengig.

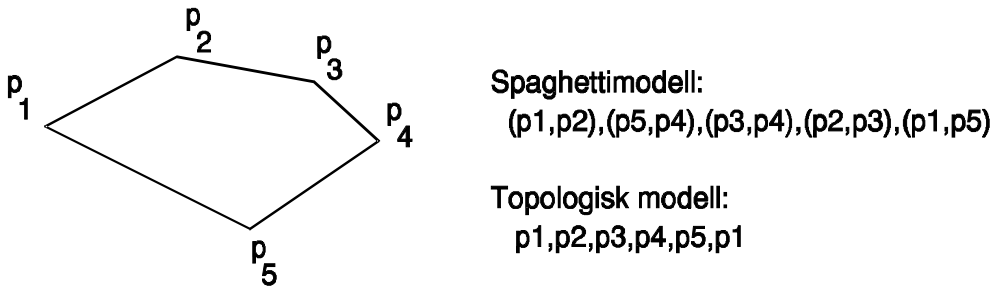
Eksempel: Dersom vi skal tegne det omskrivende polygon (omrisset) til region A på en grafisk skjerm, spiller det neppe noen rolle om vektorene tegnes i en vilkårlig rekkefølge. Polygonet kan for denne anvendelsen gjerne foreligge på spaghettiform. Skal vi derimot beregne arealet til A, må vi kjenne rekkefølgen til polygonets hjørner i en gitt omløpsretning. For den siste anvendelsen har vi derfor behov for å vite noe om topologiske relasjoner mellom vektorene.

Eksempel: For analyseformål i GIS er det som regel nødvendig å benytte topologiske modeller.

I en topologisk vektormodell av en linje er vektorene ordnet slik at naboskap i den fysiske virkelighet gjenspeiles ved vektorenes rekkefølge.

I figur 5.1 har vi en spaghettimodell og en topologisk vektor modell av et polygon med de 5 hjørnene $p_1 \cdots p_5$.

Generelt sett er man ikke bundet til å avbilde linjer som rette linjestykker, men også kurver er aktuelle. I det generelle tilfelle må man derfor i tillegg til vektorenes



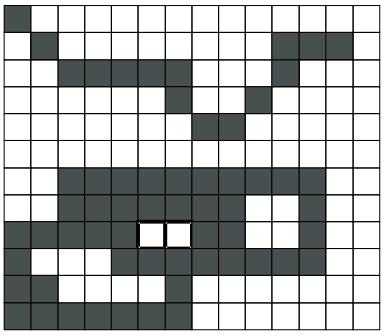
Figur 5.1: Spaghettimodell og topologisk vektormodell av et polygon.

endepunkter også oppgi hvilken interpolasjonsmetode og hvilke parametre som skal benyttes. I praksis sørger man som regel for å digitalisere så mange punkter langs linjene at objektet blir beskrevet nøyaktig nok med lineær interpolasjon, altså rette linjesegmenter mellom punktene.

5.2.2 Rastermodeller

Rastermodellen til et objekt definerer objektets geometri ved en mengde av ruter (pixler). Vanligvis oppfattes rastermodeller som ikke-topologiske. Enkelte sier i den forbindelse at et rasterbilde ikke inneholder intelligens. Dette er sjargong, fordi det har ikke mening å snakke om hverken intelligente maskiner eller intelligente datamodeller.

I figur 5.2 er vist et rasterkart. Ved å definere hjørnenaboer til et pixel som å tilhøre samme figur, finner vi to sammenhengde fargelagte figurer i kartet. Den ene av figurene har tre hvite hull.



Figur 5.2: Rasterkart

Dersom vi scanner et kart, får vi et rasterbilde av kartet. Vanligvis resulterer en slik scanning i et binærbilde av kartet. På dette stadiet er topologisk informasjon om figurer i bildet ikke eksplisitt uttrykt. Så lenge man ikke har behov for å identifisere objekter i bildet, er det ikke nødvendig med topologisk informasjon.

Vi kan tilføre et rasterbilde topologisk informasjon ved eksplisitt å definere hvilke sammenhengende mengder som finnes. En kvadtre-representasjon av et kart kan da bli en skog av kvadtrær, altså hvert objekt får sitt eget kvadtre. Ytterligere topologisk informasjon kan tilføres bildet ved å si noe om topologiske relasjoner mellom objektene. Prinsipielt kan et rasterkart tilføres den topologiske informasjon vi måtte ønske.

5.3 Klassifikasjon av geometriske modeller

Selv om begrepet vektormodeller og rastermodeller er innarbeidet, benyttes begrepene ofte på en forvirrende og inkonsistent måte. Dessuten er begrepene utilstrekkelige ved klassifikasjon av alle de forksjelligartede metoder som benyttes for å modellere geografiske objekter. Terminologien på dette området er hittil ikke velutviklet innenfor problemdomenet til GIS. Av denne grunn må vi introdusere en del nye begreper. Vi tar imidlertid utgangspunkt i topologien, da begrepsdannelser innenfor denne delen av matematikken gir en velfundert plattform for vårt formål. Den klassifikasjonen av geometriske modeller som vi etter hvert kommer fram til, vil derfor basere seg på topologiske egenskaper til modellene.

5.3.1 Topologisk dimensjon

Definisjon 21 *La R^n betegne et n-dimensjonalt Evklidisk rom.*

Et Evklidisk rom er tidligere definert som et metrisk rom med den Evklidiske metrikk.

R^n kan oppfattes som en produktmengde av R. For eksempel er punktene i et plan og et volum gitt ved henholdsvis $R^2 = R \times R$ og $R^3 = R \times R \times R$.

Definisjon 22 *En standard n-celle (n-ball) er $B^n = \{x_1, x_2, \ldots, x_n \in R^n \mid x_1^2 + x_2^2 + \cdots + x_n^2 \leq 1\}$ og en standard n-sfære er $S^n = \{x_1, x_2, \ldots, x_{n+1} \in R^{n+1} \mid x_1^2 + x_2^2 + \cdots + x_{n+1}^2 = 1\}$. En n-sfære og en n-celle (n-ball) er ethvert rom som er homeomorf med henholdsvis S^n og B^n.*

I følge foregående definisjon er en n-sfære skallet til en (n+1)-celle. For eksempel er en sirkulær skive en 2-celle avgrenset av en 1-sfære (sirkel), mens en rotasjonsellipsoide er en 2-sfære som tilnærmer jordas overflate.

Neste definisjon er basert på definisjon 17:

Definisjon 23 *En romlig region A er n-dimensjonal dersom det for alle $x \in A^\circ$ finnes en omegn hvis lukkede er homeomorf med B^n.*

En (n+1)-dimensjonal romlig region A har et n-dimensjonalt omriss ∂A. Dette er grunnen til at foregående definisjon er knyttet opp mot A°.

Det er imidlertid for restriktivt å knytte den etterfølgende teori kun mot romlige regioner. Dette er motivet bak neste definisjon.

Definisjon 24 *La $(X, \mathcal{T})$ være et sammenhengende topologisk rom og la A være en ikke-tom ekte delmengde til X. Vi sier at A er et n-dimensjonalt objekt i X dersom alle punkter til A ligger i en åpen mengde hvis lukkede er homeomorf med B^n. Størrelsen n benevnes den topologiske dimensjon til A.*

Vår definisjon av n-dimensjonalt objekt svarer til begrepet n-mangfoldighet i topologien.

Eksempel: La p og q være disjunkte punkter i R^n. Linjesegmentet fra p til q er

$$[p, q] = \{tp + (1 - t)q \mid 0 \leq t \leq 1\}$$

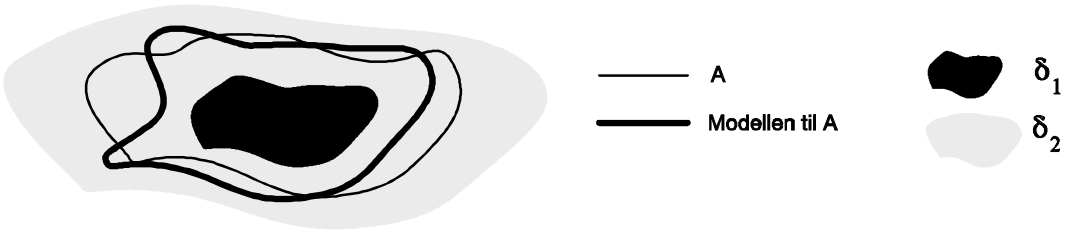
$[p, q]$ er homeomorf med med $[0, 1]$ i R^1 og er følgelig 1-dimensjonal.

Definisjon 25 *Dersom A og $\mathcal{A}$ har forskjellig topologisk dimensjon, sier vi at kodimensjonen til A og $\mathcal{A}$ er $\neq 0$.*

5.3.2 Modellbegrepet

Definisjon 26 *La A og δ være delmengder av et topologisk rom $(X, \mathcal{T})$. Dersom $\delta_1 \subset \mathcal{A} \subseteq \delta_2$, sier vi at $\mathcal{A}$ tilnærmer A og kaller $\mathcal{A}$ en modell til A.*

Mengden δ_1 og δ_2 skal her oppfattes som toleranser som skal benyttes til å angi maksimalt tillatt avvik mellom objektet og modellen av objektet. Merk at δ er applikasjonsavhengig. Se figur 5.3.



Figur 5.3: Illustrasjon av modellbegrepet.

Vi har nå skaffet oss et apparat for å definere begreper som kan benyttes til å klassifisere de metoder som benyttes for å modellere geometri innen digital kartografi. Noen sentrale begreper i de kommende definisjoner er *modell, den lukkede og omrisset til A, dekomponering samt kodimensjon.*

5.3.3 $\overline{A}$ og ∂A som klassifikasjonskriterium

Definisjon 27 *La $(X, \mathcal{T})$ være et topologisk rom, og la A og $\overline{\mathcal{A}}$ være ekte delmengder av X. Dersom $\overline{\mathcal{A}}$ tilnærmer $\overline{A}$, kalles $\overline{\mathcal{A}}$ en heldekkende modell til A.*

Definisjon 28 *La $(X, \mathcal{T})$ være et topologisk rom, og la A og $\partial\mathcal{A}$ være ekte delmengder av X. Dersom $\partial\mathcal{A}$ tilnærmer ∂A, kalles $\partial\mathcal{A}$ en omrissmodell til A.*

5.3.4 Kodimensjon som klassifikasjonskriterium

Definisjon 29 *La A og $\mathcal{A}$ være k-dimensjonale objekter i R^n med kodimensjon $= 0$ slik at $0 \leq k \leq n$. Dersom $\mathcal{A}$ tilnærmer $\overline{A}$, kalles $\mathcal{A}$ en normalmodell til A.*

Definisjon 30 *La et k-dimensjonalt objekt A og et m-dimensjonalt objekt $\mathcal{H}$ i R^n ha kodimensjon $\neq 0$ slik at $0 \leq k < m \leq n$. Dersom $\mathcal{H}$ tilnærmer $\overline{A}$, kalles $\mathcal{H}$ en supermodell til A.*

Super kommer av latin, betyr over og brukes som første ledd i sammensetninger for å uttrykke overordnet, hevet over, høyere enn etc.. Vi finner en parallell til begrepet supermodell i den engelske betegnelsen superset ($A \supset B$).

Definisjon 31 *La et k-dimensjonalt objekt A og et m-dimensjonalt objekt $\mathcal{P}$ i R^n ha kodimensjon $\neq 0$ slik at $0 \leq m < k \leq n$. Dersom $\mathcal{P}$ tilnærmer $\overline{A}$, kalles $\mathcal{P}$ en submodell til A.*

Sub kommer av latin og betyr under. Vi finner en parallell til begrepet submodell i den engelske betegnelsen subset ($A \subset B$).

Når $\mathcal{M}$ betyr modellen til A, kan ideen i de tre foregående definisjoner 29, 30 og 31 oppsummeres slik:

1. Dersom A har samme dimensjon som $\mathcal{M}$, er $\mathcal{M}$ en normalmodell til A;
2. Dersom $\mathcal{M}$ har høyere dimensjon enn A er $\mathcal{M}$ en supermodell til A;
3. Dersom $\mathcal{M}$ har lavere dimensjon enn A, er $\mathcal{M}$ en submodell til A.

5.3.5 Dekomponering som klassifikasjonskriterium

Schurle [Sch79] diskuterer oppdeling av mengder og gjør følgende definisjon:

Definisjon 32 *La $(X, \mathcal{T})$ være et topologisk rom. La $\mathcal{D}$ være en samling av disjunkte ikke-tomme delmengder til X hvis union er X. $\mathcal{D}$ kalles en dekomponering til X.*

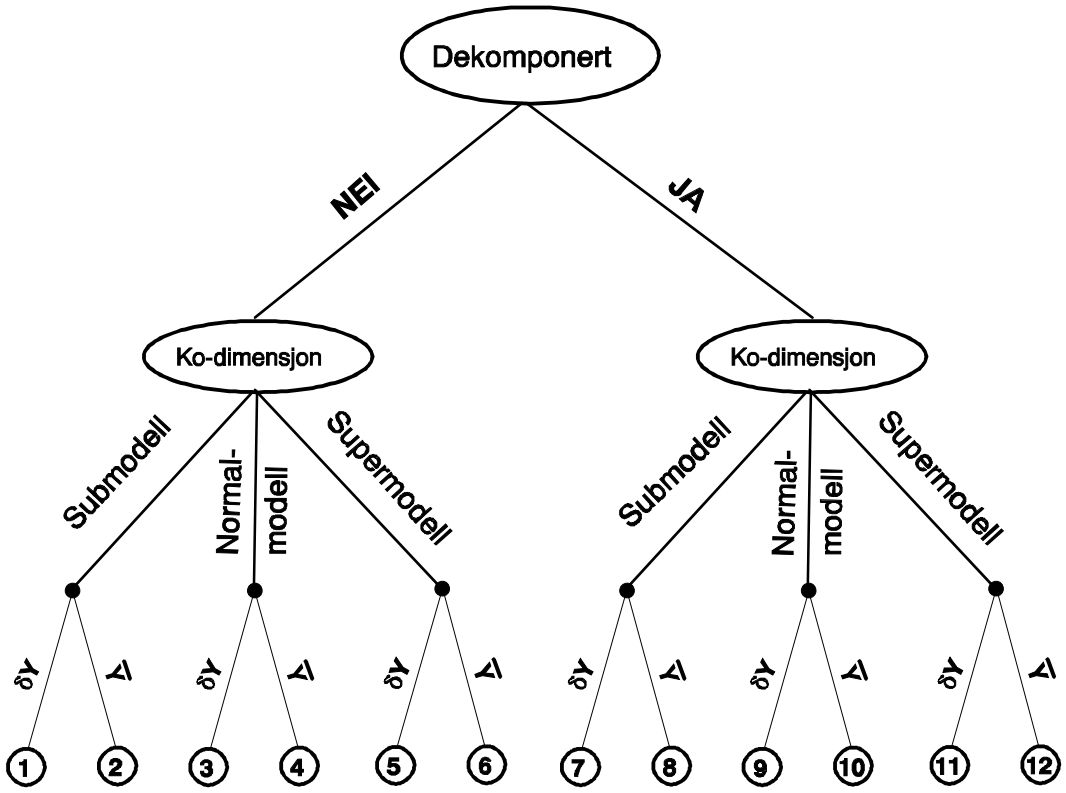
For vårt formål er foregående definisjon for restriktiv. Derfor gjør vi følgende modifikasjon:

Definisjon 33 *La $(X, \mathcal{T})$ være et topologisk rom og la A være en ekte delmengde til X. Dersom $\mathcal{M}$ er en samling av ikke-tomme n-celler til X hvis union tilnærmer A, kalles $\mathcal{M}$ en dekomponert modell til A.*

Definisjonen 33 er som nevnt inspirert av dekomponeringstanken i definisjon 32, men med den vesentlige forskjell at definisjon 33 ikke forlanger disjunkte mengder og godtar dessuten at $\mathcal{M}$ er en modell til A.

5.3.6 Klassifikasjonstre

En fordel vel å basere klassifikasjonen av geometriske modeller på deres topologiske egenskaper (dekomponering, kodimensjon og omriss-lukkede), er at klassifikasjonen blir uavhengig av kontinuerlige forandringer av det underliggende topologiske rom som endringer i målestokk, krymp, strekk etc..



Figur 5.4: Klassifikasjonstre for geometriske modeller basert på modellenes topologiske egenskaper.

Figur 5.4 er basert på de nevnte topologiske egenskaper og viser et klassifikasjonstre for geometriske modeller. De interne noder i treet representerer klassifikasjonskriteriene mens de ulike modeller er å finne i treets løvnoder.

Treet har i alt 12 løvnoder, og inneholder det vi kan kalle de 12 hovedtyper geometriske modeller. For å finne betegnelsen på en modelltype, lages en ordnet sammenstilling hvor det minst signifikante begrepet kommer først. I denne forbindelse vil vi vedta følgende orden : dekomponering, kodimensjon, omriss/lukkede. Tabell 5.1 summerer opp de tolv modellbetegnelsene og gir et eksempel for hver av modellene.

I noen tilfeller kan det være aktuelt å benytte underklasser av de nevnte 12 modeller. La oss for eksempel anta at vi har en rasterepresentasjon av en høydekurve til et terrengområde. Høydekurven ligger på omrisset til jorda. En høydekurve er

derfor en sub-omrissmodell av jorda, men siden vi har modellert høydekurven ved hjelp av et raster, blir betegnelsen : super(sub-omrissmodell). Parantesen i uttrykket tilkjennegir at vi har å gjøre med en modell av en modell. Høydekurver er nemlig en modell av jordas topografi mens rasteret er en modell av høydekurven. Vi skal gi noen eksempler på ulike modelltyper.

Tabell 5.1: Terminologi og eksempler på de 12 hovedtyper geometriske modeller.

Modell	terminologi
M_1	sub-omrissmodell
M_2	sub-heldekkende modell
M_3	normal omrissmodell
M_4	normal heldekkende modell
M_5	super-omrissmodell
M_6	super-heldekkende modell
M_7	dekomponert sub-omrissmodell
M_8	dekomponert sub-heldekkende modell
M_9	dekomponert normal omrissmodell
M_{10}	dekomponert normal heldekkende modell
M_{11}	dekomponert super-omrissmodell
M_{12}	dekomponert super-heldekkende modell

Modell	eksempel
M_1	En enkelt kurve på overflaten til et volum
M_2	En enkelt snittflate til et volum
M_3	Omrisset til et areal tilnærmet med S^1
M_4	Et areal tilnærmet med B^2
M_5	Et omskrivende rektangel til omrisset av et areal
M_6	Et omskrivende volum til et areal
M_7	En skare av kurver på overflaten til et volum
M_8	En skare av snittflater til et volum
M_9	En vektormodell av omrisset til et areal
M_{10}	En rastermodell av et areal
M_{11}	En rastermodell av omrisset til et areal
M_{12}	En terningmodell (3d-rastermodell) av et areal

Eksempel: En trekantmodell (terrengmodell) er en omrissmodell til et 3d-objekt og er følgelig 2-dimensjonal. Siden den også er en fasettmodell, blir betegnelsen: dekomponert normal omrissmodell = M_9.

Eksempel: Trekantmodeller kan generaliseres til 3d-modeller ved å gå over til tetraeder. Modellbetegnelsen blir i såfall dekomponert normal heldekkende modell = M_{10}.

Eksempel: La oss anta at en vei (et areal) blir modellert ved sin senterlinje. En vektormodell av senterlinjen får da betegnelsen M_{10}, mens rasterrepresentasjonen av senterlinjen er modell M_{12}. Siden senterlinjen er modell M_2 til veien, blir de fullstendige modellbetegnelser henholdsvis $M_{10}(M_2)$ og $M_{12}(M_2)$.

Eksempel: La oss anta en serie snittflater som modellerer et volum og at snittflatene modelleres ved en samling av terninger. Modellbetegnelsen blir $M_{12}(M_8)$.

Eksempel: En vektormodell av en enkelt høydekurve er $M_{10}(M_1)$, mens en vektormodell av en skare av høydekurver er $M_{10}(M_7)$.

Eksempel: Et representasjonspunkt y til en eiendomsteig A, er modell M_2 til A. Toleransene i dette tilfellet kan settes slik at y skal ligge innenfor A og ikke ligge nærmere A sitt omriss enn en gitt avstand r. Dette kan uttrykkes slik: $\delta_1 = \emptyset$ og $\delta_2 = \cup\{x \in A \mid U_x \in A \wedge \epsilon \geq r\}$.

Med tanke på utveksling av data mellom ulike GIS, er det av interesse å kjenne modellbetegnelsen til de geometriske objekter. Dersom vi for eksempel har en rasterrepresentasjon av et objekt og modellen er M_{12}, er objektet av dimensjon mindre enn rasterets dimensjon. Dette betyr at operasjoner på objektet som arealberegning, må pålegges visse restriksjoner.

Kapittel 6

ROMLIGE DATASTRUKTURER

Vi skal starte med noen eksempler som kan motivere for å sette seg inn i de metoder dette kapitlet beskriver. Anta at vi har en stor geodatabase over Norge og at vi stiller følgende spørsmål til databasen:

1. Finn alle kornareal som ligger i høydeintervallet 0 til 50 m o.h.. Kostnadene ved å gjøre et slikt søk vil variere sterkt avhengig av hvordan dataene er strukturert.

2. Finn om E6 og E18 krysser hverandre. Dersom vi antar at veiene er bygget opp av en sekvens vektorer og at vektorene i gjennomsnitt er 100m lange, vil vi over en strekning på 100 mil ha 10.000 vektorer. Om dataene ikke er strukturert på en formålstjenlig måte, vil tiden det tar å besvare spørsmålet kunne bli uakseptabel lang.

3. Finn alle tellekretser som ligger innenfor en radius av 50km fra Trondheim sentrum og som har mindre enn 1000 innbyggere. En rett fram løsning er å besøke samtlige tellekretser i databasen og spørre om de ligger mindre enn 50km fra Trondheim. En bedre strategi er å benytte en datastruktur som tillater at vi raskt kan finne fram til aktuelle kandidater (snevre inn søkeområdet). De tellekretser som opplagt ikke kan være kandidater, bør kunne elimineres ved å si: alt syd for Støren og alt Nord for Steinkjær kan umulig være kandidater. Dermed er søkeområdet radikalt innskrenket.

De fleste datastrukturer som her skal presenteres, er *hierarkiske*, men også andre datastrukturer som er hensiktsmessige for våre formål, vil bli gjennomgått. Hierarkiske datastrukturer har blitt svært populære på grunn av deres evne til å fokusere på interessante delmengder av dataene. Dette resulterer i at mengdeoperasjoner kan utføres med et forbedret tidsforbruk.

6.1 Operasjoner på romlige data

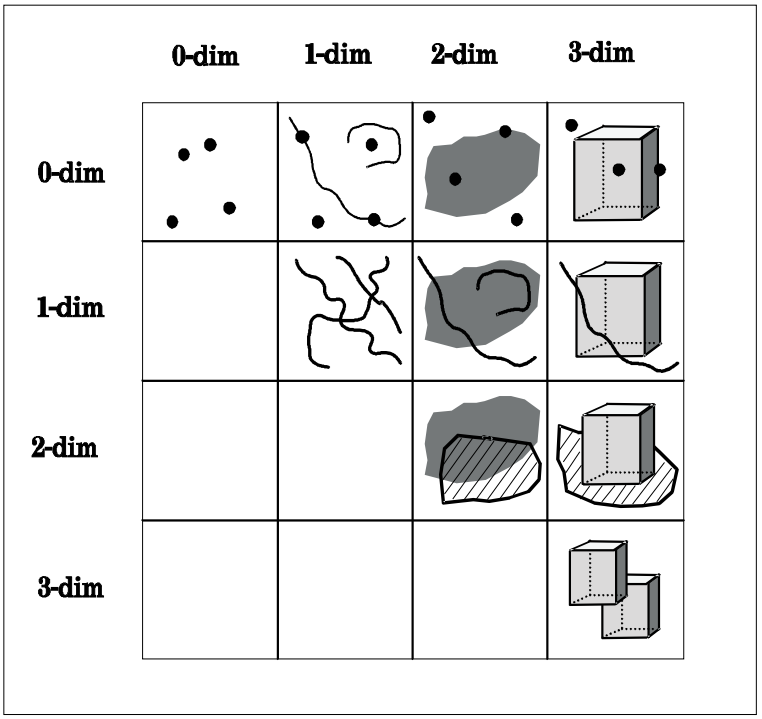
Før vi diskuterer de ulike typer romlige datastrukturer, er det nyttig å tenke igjennom hvilke klasser romlige operasjoner disse datastrukturene skal understøtte.

På et overordnet nivå kan vi skille mellom følgende klasser av operasjoner:

1. Oppdatering av datastrukturen (innsetting og sletting).
2. Romlige søk basert på geometriske og topologiske kriterier til geografiske objekter.
3. Romlige søk basert på semantisk informasjon som er knyttet til de geografiske objekter.

Overgangen mellom klassene 2. og 3. er noe flytende og avhengig av hvordan vi vil modellere objektene. Skal for eksempel z-verdien modelleres som et attributt til et (x, y)-par eller skal vi modellere objektet som et ekte tredimensjonalt objekt (x, y, z).

Typer romlige søk som kan etableres på grunnlag av geometriske og topologiske kriterier er illustrert i figure 6.1.



Figur 6.1: Romlige søk som kan avledes av geometriske og topologiske relasjoner.

Tabellen i figur 6.1 har en vertikal og en horisontal inngang over objektenes topologiske dimensjon. En vilkårlig inngang (i-dim,j-dim) viser relasjoner mellom i- og j-dimensjonale objekter.

La oss for eksempel velge inngang (0-dim,0-dim). Romlige søk som her kan etableres er av typen: finn nærmeste punkt til punkt P, finn alle punkter som ligger nærmere P enn en gitt avstand.

Under inngang (1-dim,1-dim) finner vi romlige søk av typen: finn skjæringspunktet mellom to linjer, finn alle linjer som ligger innenfor en avstand s fra linjen L.

Videre har vi under inngang (2-dim,2-dim) romlige søk av typen: finn alle polygoner som ligger innenfor et gitt polygon, finn alle polygoner som har felles grenselinje med polygon P, finn snittet mellom to romlige regioner, finn alle romlige regioner som har en innbyrdes avstand mindre enn s, finn alle romlige regioner som har et areal større enn A.

Prinsippet for å konstruere romlige søk baserer seg på geometriske og topologisk-romlige relasjoner til objektene. Dette gir oss et stort antall kombinasjonsmuligheter, noe som viser den kompleksiteten og de utfordringer vi står overfor når datastrukturer skal velges.

6.2 Kvadtrær

Begrepet *kvadtre* betegner en klasse hierarkiske datastrukturer som har det til felles at de baserer seg på en rekursiv dekomponering av et 2d-objekt. En utvidelse av konseptet til flere dimensjoner er mulig. For eksempel er *okttre* en tredimensjonal slektning til kvadtreet.

Kvadtrær kan klassifiseres på grunnlag av:

- De data trærne skal representere.
- Prinsippet for oppdeling av rommet.
- Om oppløsningen i treet er variabel eller ikke.

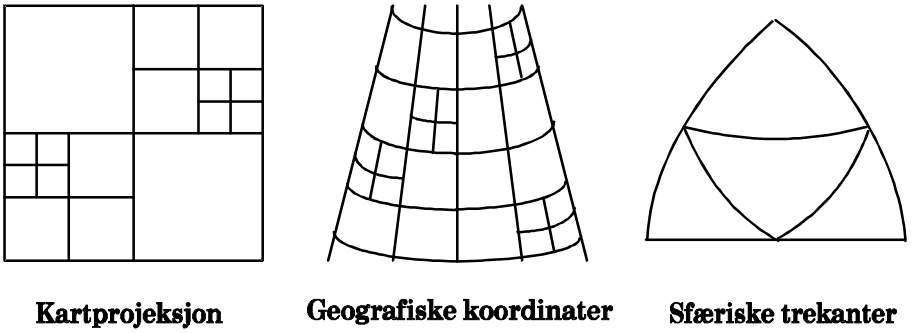
Kvadtrær utgjør en meget viktig gruppe datastrukturer for punkter, linjer og areal. I GIS-sammenheng kan vi gjerne oppfatte kvadtrær som en abstrakt datatype der operasjonene på trærne er de vanlige GIS-forespørsler som: finn nærmeste objekt, finn overlappingsområde osv. .

Et kvadtre deler planet rekursivt i de fire kvadranter nordvest (NV), nordøst (NØ), sydvest (SV) og sydøst (SØ). Oppdelingen pågår inntil informasjonsmengden i en kvadrant når en viss nedre grense (betydningen av dette vil bli vist senere). Oppdeling av rommet kan gjøres etter følgende prinsipper:

- *Regulær firedeling:* delelinjene skal halvere sine tilhørende intervall.
- *Irregulær firedeling:* delelinjene legges der det er mest hensiktsmessig.

Vanligvis skjer oppdelingen i en kartprojeksjon, men dette vil føre til problemer dersom kvadtreet skal strekke seg over store landområder. Et alternativ er å benytte geografiske koordinater (ϕ, λ) og foreta oppdelingen på kuleoverflaten. Dette har

imidlertid den ulempen at arealet som utspennes innenfor kvadrantene vil bli avhenging av posisjonen på jordkula. På grunn av meredianenes konvergens vil nemlig avstanden mellom to meridianer være størst ved ekvator og minst i polpunktene (0).



Figur 6.2: Prinsipper for å etablere kvadtrær over store landområder

For å unngå dette problemet, er det i stedet foreslått å benytte sfæriske trekanter. Man starter med noen få sfæriske trekanter som dekker hele jordkula. Deretter deles disse ved å halvere deres sider. Halveringen fører til at én sfærisk trekant blir delt i fire mindre trekanter. Figur 6.2 viser de nevnte prinsipper for kvadtrær som skal dekke store landområder. Denne problematikken ligger imidlertid utenfor problemdomenet til dette kapitlet. Vi vil derfor forutsette at oppdelingen gjøres i en kartprojeksjon.

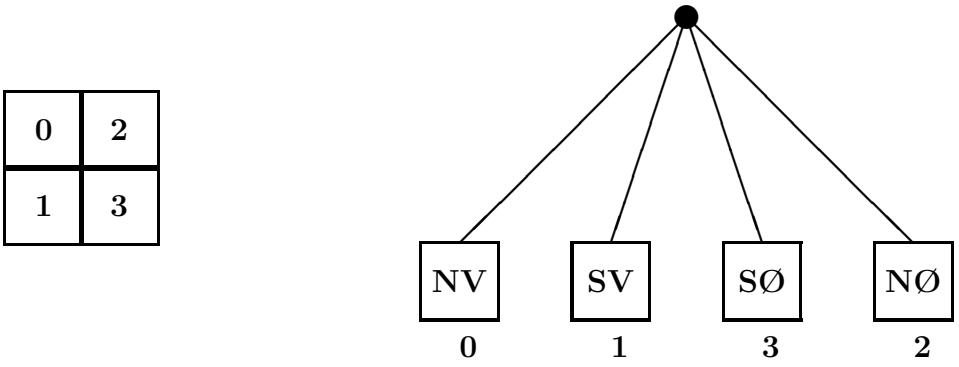
Det finnes en stor internasjonal litteratur om kvadtrær. Den person som står for den største produksjon, er nok Hanan Samet. Samet har nedlagt et imponerende arbeid i å samle og systematisere litteratur om kvadtrær. Framstillingen i dette kapitlet bygger i stor grad på hans arbeider. Av Samet's publikasjoner anbefales artikkel [Sam84] og hans to bøker [Sam90b] og [Sam90a]

6.3 Regulær firedeling av planet

Kvadtrær med regulær firedeling er innenfor GIS den mest brukte gruppen kvadtrær. Denne gruppen kvadtrær kalles i engelsk litteratur ofte for *region based quadtrees*. Oversatt skulle dette bli *arealbaserte kvadtrær*. Grunnen til denne terminologien er nok at kvadtrær ganske tidlig ble benyttet som en datastruktur for binære bilder, men i dag har kvadtrær en langt bredere anvendelse. Vi vil heretter benytte terminologien *regulære kvadtrær* om gruppen region based quadtrees", men det kan nok forekomme at betegnelsen arealbasert kvadtre også blir benyttet.

Definisjon 34 *Et regulært kvadtre gjør en rekursiv oppdeling av et to-dimensjonalt rom i fire kvadranter ved at delelinjene halverer sine respektive intervall.*

Som vi senere skal se, gjør den regulære oppdelingen det mulig å finne er formel som transformerer alle koordinatpar innefor en kvadblokk til en entydig blokkadresse (et enkelt tall).



Figur 6.3: Vår konvensjon for opptegning av kvadtrær.

Ved opptegning av et kvadtre er det et spørsmål om i hvilken rekkefølge greinene skal stråle ut fra nodene. Vi vil benytte en litt annen konvensjon enn Samet, i det vi tenker oss at en nodes greiner mot nordvest og nordøst faller ned med rotasjonspunkt i noden. Dette gir oss konvensjonen i figur 6.3.

Figur 6.4 viser et kvadtre og dets oppdeling av planet. Tallene i rutene er mortonindekser i fire-tallsystemet (se figur 2.23). Ruter som ikke er delt opp til minste rutestørrelse, har fått tildelt nummeret til den etterfølgende elementærrute som har det laveste nummer. Dette gjelder for eksempel rutene 000, 230 og 120.

6.3.1 Plassbehov

I et regulært kvadtre med N løvnoder, er den minste høyde treet kan ha:

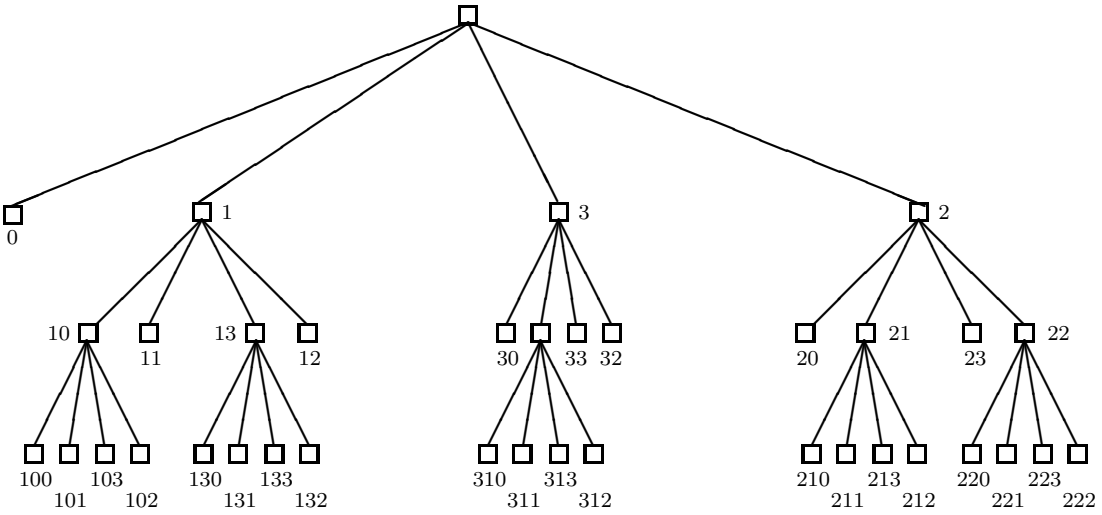
$$h_{min} = \lceil \log_4 N \rceil$$

som inntreffer når samtlige løvnoder ligger på samme nivå i treet (treet er balansert). Noen tallverdier for dette er vist i tabell 6.1.

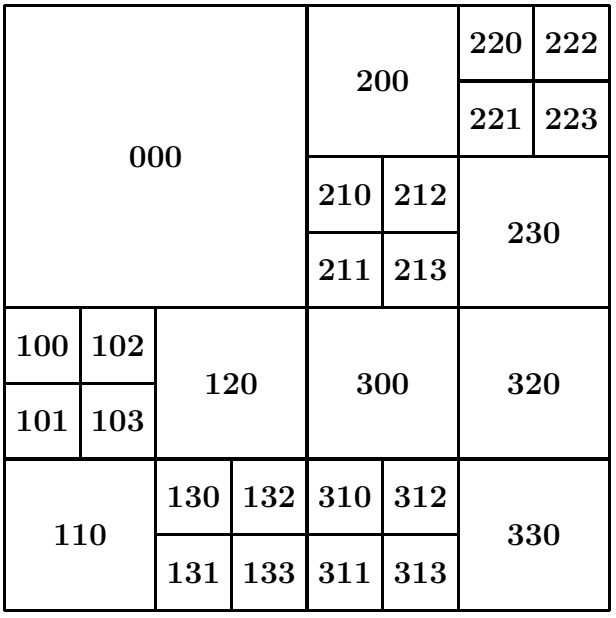
Tabell 6.1: Minste trehøyde i et regulært kvadtre for noen verdier av N.

h_{min}	antall løvnoder
0	1
1	4
2	16
3	64
4	256
5	1.024
6	4.096
7	16.384

h_{min}	antall løvnoder
8	65.536
9	262.144
10	1.048.576
11	4.194.304
12	16.777.216
13	67.108.864
14	268.435.456
15	1.073.741.824



(Bare de signifikante siffer i nodenes posisjonskoder er anført på figuren.)



Figur 6.4: Et regulært kvadtre og dets oppdeling av planet.

Den maksimale trehøyde h_{maks} kan beregnes når minste ruteside d og det kartlagte områdets størrelse er gitt. Vi forutsetter at det kartlagte området er kvadratisk med sidekant s. Den maksimale trehøyde er da gitt ved:

$$h_{maks} = \lceil \log_2(s/d) \rceil$$

I tabell 6.2 er det regnet ut noen verdier for maksimal trehøyde for $d = 1$. Ved å velge enheten meter, vil vi av tabellen se at et geografisk område av størrelse 2097km×2097km vil kreve en maksimal trehøyde lik 21 for å kunne operere med ruter på 1m×1m.

Tabell 6.2: Maksimal trehøyde i et regulært kvadtre for noen verdier av s når d er satt lik 1.

h_{maks}	sidekant
0	1
1	2
2	4
3	8
4	16
5	32
6	64
7	128

h_{maks}	sidekant
8	256
9	512
10	1.024
11	2.048
12	4.096
13	8.192
14	16.384
15	32.768

h_{maks}	sidekant
16	65.536
17	131.072
18	262.144
19	524.288
20	1048.576
21	2097.152
22	4194.304
23	8388.608

6.3.2 Posisjonskode

Av figur 6.4 kan vi gjøre noen interessante observasjoner. For det første ser vi at rutenes nummer definerer stien fra roten i treet til vedkommende rute. For eksempel kan rute 130 nås ved å starte i roten og så gå til etterfølgeren i retning 1, for derfra å gå til dennes etterfølger i retning 3, for til slutt å gå i retning 0. Stien er altså SV,SØ,NV. Dette viser at mortonindeks og sti er likeverdige. Riktigheten av dette framkommer ved å se på sammenhengen mellom det binære tallsystemet og firetallsystemet. Konvertering til firetallsystemet skjer som kjent ved å dele bitstrengen i grupper av 2 biter, men 2 biter er jo nettopp resultatet av fletting av bitene x_i og y_i. Siden mortonindeksen framkommer ved en slik fletting, vil vi ved å tolke bitstrengen til mortonindeksen i firetallsystemet få stien til vedkommende kvadblokk.

Mortonindeksen 000 i figur 6.4 vil ved en konvertering til stikode gi NV,NV,NV. Dette skaper er problem, fordi rute 000 i dette tilfellet ikke er en elementærrute. Problemet takles ved å innføre en opplysning i tillegg til mortonindeksen. Denne ekstra informasjonen er en verdi som angir hvilket nivå ruten ligger på. Den fullstendige nummerkoden for rutene består derfor av *mortenindeksen* pluss en *nivåangivelse*. Dette leder oss til følgende definisjon:

Definisjon 35 *En kode som entydig fastlegger posisjonen til en kvadblokk, kalles en posisjonskode.*

Det har vært foreslått å angi posisjonskoder i 5-tallsystemet (0,1,2,3,4), fordi man da får et talltegn som kan brukes til å fortelle benyttes ikke". Ved å reservere tallene 1,2,3,4 for de fire kvadranter, kan 0 brukes til å fortelle benyttes ikke". For eksempel dersom vi tolker en bitstreng og får 001332, skal 00 hoppes over. Stien til kvadblokken er derfor 1332. Selv om nivåkoden kan sløyfes, oppnås totalt sett neppe fordeler i forhold til å benytte posisjonskoder i firetall-systemet, fordi for å representere 5 talltegn må avsettes 3 biter mens for å representere 4 talltegn behøves bare 2 biter.

6.3.3 Lineære kvadtre

Posisjonskoden kan benyttes til å lage trær som ikke har pekere, siden posisjonskoden gir oss et unikt tall for hver enkelt kvadblokk. Det er i denne forbindelse nærliggende å utnytte posisjonskoden til bare å lagre løvnodene i treet. Kvadtreet kan derfor reduseres til en mengde løvnoder. Denne typen kvadtrær benevnes som *lineære kvadtrær*.

Begrunnelsen for å benytte lineære kvadtre framfor pekerbaserte kvadtre, er gjerne hensynet til store datamengder. Et tre som er så stort at det ikke kan holdes i primærminnet, er vanligvis enklere (avhengig av programmeringssomgivelser) å manipulere når det foreligger som et lineært kvadtre.

Vi forutsetter at *dybden* til nodene benyttes som nivåkode. Dybden til en node er lik lengden av stien fra roten til noden. Roten i treet får følgelig dybde 0.

Dersom nivåkoden settes på mest signifikante plass i posisjonskoden, vil en sortering på posisjonskode føre til at alle blokker på høyere nivå i treet kommer foran blokkene på lavere nivå. Kvadtreet i figuer 6.4 vil da være gitt ved følgende liste av sorterte løvnoder (nivåkode med uthevet skrift):

1000, **2**110, **2**120, **2**200, **2**230, **2**300, **2**320, **2**330, **3**100, **3**101, **3**102, **3**103, **3**130, **3**131, **3**132, **3**133, **3**210, **3**211, **3**212, **3**213, **3**220, **3**221, **3**222, **3**223, **3**310, **3**311, **3**312, **3**313.

Vi har altså fått en nivåvis sortering av kvadblokkene. For de praktiske anvendelser er det vanligvis å foretrekke at nivåkoden legges på minst signifikante plass i posisjonskoden. Dette gir oss følgende sorterte liste:

000**1**, 100**3**, 101**3**, 102**3**, 103**3**, 110**2**, 120**2**, 130**3**, 131**3**, 132**3**, 133**3**, 200**2**, 210**3**, 211**3**, 212**3**, 213**3**, 220**3**, 221**3**, 222**3**, 223**3**, 230**2**, 300**2**, 310**3**, 311**3**, 312**3**, 313**3**, 320**2**, 330**2**.

I et lineært kvadtre har man ikke noen gitt mekanisme for å finne blokker med bestemte nummer. For å minimalisere søketiden, må det derfor legges en eller annen struktur over kvadblokkene.

En metode er å benytte sorterte lister og gjøre framhenting ved binært søk. Dette tillater at en kvadblokk kan lokaliseres i $O(\log_2 n)$ tid. Fordelen ved å benytte

sorterte lister er at man slipper å bruke lagerplass til et søkeapparat.

Et 2-3tre vil også være en aktuell datastruktur over posisjonskodene. På grunn av at intervallspørsmål er vanlige ved geografiske søk, er det hensiktsmessig å kjede sammen løvnodene i 2-3treet.

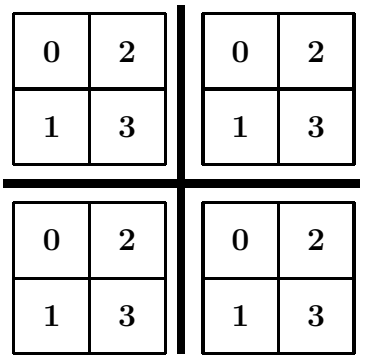
Ved lagring av store datamengder på sekundærlager, er B^+-tre et fornuftig valg.

6.3.4 Teknikker for å finne naboer

En kvadblokk kan ha *sidenaboer* og *hjørnenaboer*. Sidenaboer til blokk 212 i figur 6.4 er blokkene 200, 210, 213 og 230. Før vi finner hjørnenaboene til blokk 212, er det nødvendig å presisere at vi vil søke etter naboer som er av samme størrelse som blokk 212. Hjørnenaboene til blokk 212 er følgelig: 221,201,211 og 231, hvor blokkene 201 og 231 framkommer ved en oppdeling av henholdsvis blokk 200 og blokk 230.

Naboene kan gis navn etter den retning de befinner seg i. Dette gir oss følgende 8 betegnelser: N,S,Ø,V,NV,SV,NØ,SØ. For eksempel er blokk 211 SV nabo til blokk 212 og blokk 213 er Ø nabo til blokk 211. Dette kan uttrykkes ved følgende funksjoner: SV(212)=211 og Ø(211)=213.

En sentral problemstilling for operasjoner på kvadtrær, er å finne naboblokker. Dette vil nå bli forklart. La oss for eksempel anta at vi vil finne Ø nabo til 211. Siden blokk 211 ligger i vestre kvadrant til sin forgjenger 21, vet vi at den søkte nabo er en øst etterfølger til 21. Naboen har derfor posisjonskoden 213. Vi skal så finne øst nabo til blokk 132, som kalles Ø(232). Siden blokk 132 ligger i østre del av sin forgjenger 13, kan ikke 13 være forgjenger til både 132 og 310. Hva gjør vi så? Jo, fortsetter på samme måten oppover i treet. Blokk 13 ligger også i østre del av sin forgjenger, derfor kan heller ikke blokkene 13 og 31 ha en felles forgjenger. Vi må derfor fortsette enda et nivå opp. Her finner vi blokkene 1 og 3, men disse har en felles forgjenger. Den felles forgjenger spiller en sentral rolle i beregning av posisjonskoden. Den videre utledning lar seg enkelt forklare med utgangspunkt i figur 6.5.



Figur 6.5: Skjema for definisjon av retninger til naboer.

Skjemaet i figur 6.5 benyttes for å transformere fra stien til node 132 til stien til dens østnabo. Transformasjonen foretas med utgangspunkt i den felles forgjenger. Ø(1) er 3, Ø(3) er 1 og Ø(2) er 0. Ø(132) er følgelig 310. Tilsvarende teknikk benyttes for å finne de øvrige naboer.

For å gjøre transformasjonen så rask som mulig, er det fordel å benytte oppslagstabeller av følgende type:

ØST-nabo		
nodens sti	naboens sti	felles forgjenger?
0	2	ja
1	3	ja
2	0	nei
3	1	nei

Tabellen forutsetter at transformasjonen starter med det minst signifikante tallet i stikoden for så å gå oppover til felles forgjenger er funnet. Dette er angitt ved koden "ja"i kolonnen felles forgjenger?". På tilsvarende måte kan det etableres tabeller for samtlige 8 retninger. Disse tabellene er det hensiktmessig å slå sammen til to større tabeller. Den ene tabellen *retning(r,k)* gir transformasjonen mens den andre tabellen *felles-forgjenger(r,k)* gir verdien true (T) dersom transformasjonen skal terminere (felles forgjenger er nådd). Se tabellene 6.3 og 6.4.

Tabell 6.3: Felles-forgjenger(r,k)

r (retning)	k (kvadrant)			
	0 (NV)	1 (SV)	2 (NØ)	3 (SØ)
N	F	**T**	F	**T**
Ø	**T**	**T**	F	F
S	**T**	F	**T**	F
V	F	F	**T**	**T**
NV	F	F	F	**T**
NØ	F	**T**	F	F
SV	F	F	**T**	F
SØ	**T**	F	F	F

Bruken av tabellene 6.4 og 6.3 skal vises ved et eksempel. La oss finne SØ nabo til en kvadblokk med sti 112103. I dette tilfellet skal vi finne en hjørnenabo. Det å finne hjørnenaboer direkte krever en litt mere innviklet logikk enn for sidenaboer. Blant annet må logikken inneholde regler for speilvending. Kompleksiteten kan imidlertid reduseres ved at vi splitter søket opp i en horisontal bevegelse og en verikal bevegelse. Vi søker derfor først øst nabo til den aktuelle kvadblokken. Vi starter bakerst i stikoden og får:

Tabell 6.4: Retning(r,k)

r (retning)	k (kvadrant)			
	0 (NV)	1 (SV)	2 (NØ)	3 (SØ)
N	1	0	3	2
Ø	2	3	0	1
S	1	0	3	2
V	2	3	0	1
NV	3	2	1	0
NØ	3	2	1	0
SV	3	2	1	0
SØ	3	2	1	0

retning(Ø,3)=1 og felles-forgjenger(Ø,3)=F,
retning(Ø,0)=2 og felles-forgjenger(Ø,0)=T.

Øst nabo finner vi av altså av tabellene til å bli 112121. Deretter søker vi syd nabo til blokken 112121. Vi starter bakerst i stikoden og får:

retning(S,1)=0 og felles-forgjenger(S,1)=F,
retning(S,2)=3 og felles-forgjenger(S,2)=T.

SØ nabo til 112103 er følgelig: 112130.

La oss finne N nabo til blokk 102. Bruk av tabellene gir oss blokk 013. Metoden ga oss ikke blokk 000, men den N nabo av samme størrelse som 102. Dette demonstrerer følgende viktig prinsipp ved vår algoritme:

> tabellene 6.4 og 6.3 gir posisjonskoder til naboblokker som er *like store* som den blokken naboer spørres etter.

De teknikker for å finne naboer som her er beskrevet, kan benyttes både for lineære kvadtre og for pekerbasert kvadtre. For det pekerbaserte tilfellet, finnes stikoden til en node ved å traversere treet fra roten til noden. Deretter benyttes tabellene 6.4 og 6.3 for å transforere stikoden til den ønskede nabos stikode. Av tabell 6.3 går det fram at sidenaboer har 2T og 2F, mens hjørnenaboer har 1T og 3F. Dette kan tyde på at det er mere tidkrevende å finne hjørnenaboer enn sidenaboer. For å finne en sidenabo må under visse forutsetninger om treets fasong, i gjennomsnitt besøkes 2 forgjengere, mens for å finne en hjørnenabo må under de samme forutsetninger i gjennomsnitt besøkes 2,7 forgjengere (forutsatt en direkte søkemetode).

6.3.5 PR-kvadtre

Et *PR-kvadtre* (eng. Point-Region-quadtree) er et regulært kvadtre som holder koordinatene om 0-dimensjonale objekter i løvnodene. Det finnes to varianter. En for intern lagring og en annen for eksterne lagringsmedier. Forskjellen stikker i kapasiteten til løvnodene. For den interne varianten lagres maksimalt ett punkt i en løvnode. For den eksterne varianten derimot, dimensjoneres lagerkapasiteten til løvnodene etter det eksterne lagringsmediets karakteristika. Et hensiktsmessig valg kan være å dimensjonere løvnodene etter lagringsmediets minste adresserbare enhet.

Datastruktur

Datastrukturen for et pekerbasert PR-kvadtre kan defineres slik:

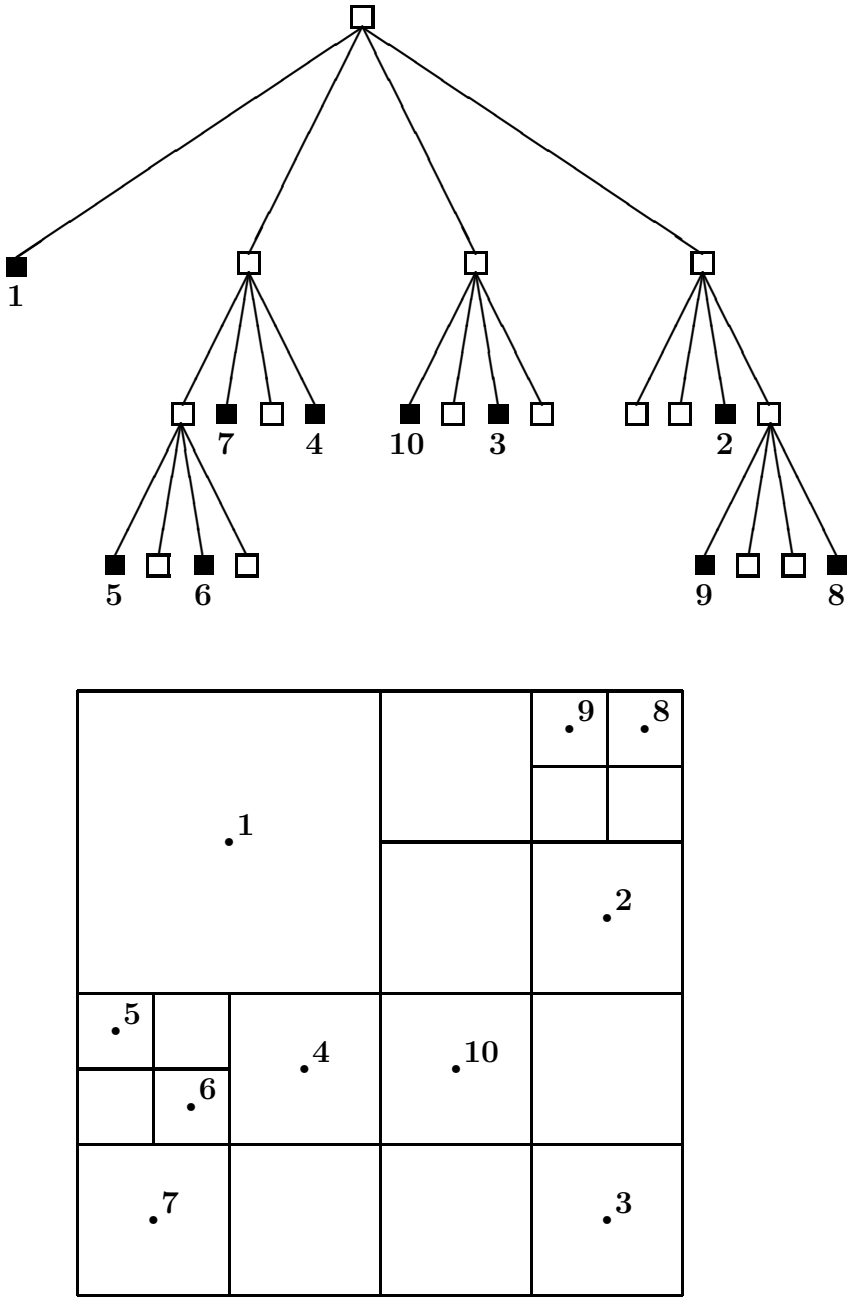
```
type
    nodetype = (løv,intern);
    trepeker = ↑PRkvadnode;
    PRkvadnode = record
        case slag: nodetype of
            (1) løv: (x,y: integer); {primærminnet}
            (2) løv: (C: array [ 1..N ] of koordinattype);
                        {alternativ databøtte på platelager}
            intern: (NV,NØ,SV,SØ: trepeker);
            {intern node}
    end;
```

Innsetting

Treets fasong påvirkes ikke av den rekkefølge punktene settes inn i. Derimot vil treets fasong variere (ikke være invariant) under rotasjoner eller forskyvninger av koordinataksene.

Innsetting i treet er ganske rett fram. For hver gang en løvnode går full, blir den splittet i de fire kvadranter og en ny internnode p_1 kjedes inn i treet. La oss anta at løvnode q splittes i nodene q_1, q_2, q_3, q_4 of at q har forgjengeren p. Før splittingen har vi rekkefølgen p, q, mens etter splittingen har vi rekkefølgen $p, p_1, \{q_1, q_2, q_3, q_4\}$. Den nye internnoden p_1 har overtatt plassen til q og p_1 har fått fire etterfølgere. Dersom punktene etter en splitt fortsatt ligger i den ene av subkvadrantene, tas en ny splitt. Slik fortsetter det inntil ingen løvnode inneholder mere enn sitt maksimale antall punkter. I figur 6.6 er vist et PR-kvadtre og dets oppdeling av planet.

Et problem er å takle innsetting av punkter som ligger svært nære hverandre. Teoretisk er dette mulig ved å gjøre rutestørrelsen uendelig liten, men i praksis vil dette ikke være gjennomførbart. Det man kan gjøre er å stanse oppdelingen når den når en viss praktisk nedre grense og så innføre en overløpsblokk for de løvnoder som fortsatt har for mange datapunkt.



Figur 6.6: Et PR-kvadtre og dets oppdeling av planet.

I det etterfølgende er gitt en fullstendig kode for innsetting av punkter i et PR-kvadtre. Hovedrutinen Sett-inn-PR-trebenytter funksjonen Kvadrant"og prosedyren Del-Rute". Funksjonen Kvadrantfinner den etterfølgende subkvadrant q til kvadrant p punktet (x_i, y_i) ligger i. I de tilfeller at punktet faller nøyaktig på en av delelinjene, må det gjøres et vedtak om hvilken side av delelinjen punktet tilhører. Prosedyren Del-rutefinner hjørnekoordinatene til kvadrant p sin etterfølgende subkvadrant q, hvor $q \in$ { NV,SV,NØ,SØ }.

Programmet avviser mere enn én forekomst av samme koordinatpar.

```
type
    nodetype = (løv,intern);
    trepeker = ↑PR-node;
        PR-node = record
        case slag : nodetype of
            løv: (x,y: integer);
            intern: (A: array [ 1..4 ] of trepeker)
            {1=NV, 2=NØ, 3=SV, 4=SØ}
    end;
var
    xmin,ymin,xmax,ymax: integer; {geografisk område}
    rot-PR: trepeker;


function Kvadrant(x,y,x1,y1,x2,y2: integer): integer;
    {finner den kvadrant punktet (xy) ligger i}
    begin
        if (x ≥ (x1+x2) div 2) and (y < (y1+y2) div 2) then
                    Kvadrant:= 1 {NV}
        else if (x ≥ (x1+x2) div 2) and (y ≥ (y1+y2) div 2)
                    then Kvadrant:= 2 {NØ}
        else if (x < (x1+x2) div 2) and (y < (y1+y2) div 2) then
                    Kvadrant:= 3 {SV}
        else if (x < (x1+x2) div 2) and (y ≥ (y1+y2) div 2) then
                    Kvadrant:= 4 {SØ}
        else error('feil i kvadrantrutinen');
end;


procedure Del-Rute(kv: integer; var x1,y1,x2,y2: integer);
    {beregner hjørnekoordinatene til subkvadrant kv}
begin
    if (kv=1) or (kv=2) then x1:= (x1+x2) div 2;
    if (kv=2) or (kv=4) then y1:= (y1+y2) div 2;
```

```
    if (kv=3) or (kv=4) then x2:= (x1+x2) div 2;
    if (kv=1) or (kv=3) then y2:= (y1+y2) div 2;
end; {Del-Rute}


procedure Sett-inn-PR-tre(x,y: integer; p: trepeker);
    setter punktet (x,y) inn i et PR-tre
var
    q,forgjenger,løvnode,ny-intern,ny-løv: trepeker;
    kv,x1,y1,x2,y2,i: integer;
begin
    if (x > xmax) or (y > ymax) or (x < xmin) or
    (y < ymin) then begin
        error('ulovlig koordinatverdi');
    end
    else begin
        q:= p;
        løvnode:= nil;
        x1:= xmin; y1:= ymin; {SV-hjørne}
        x2:= xmax; y2:= ymax; {NØ-hjørne}
        while (q <> nil) and (q↑.slag = intern) do begin
            {finner løvnode}
            kv:= Kvadrant(x,y,x1,y1,x2,y2);
            løvnode:= q↑.A[kv];
            forgjenger:= q;
            q:= q↑.A[kv];
            Del-Rute(kv,x1,y1,x2,y2);
        end; {while}
        if (løvnode <> nil) and (løvnode↑.x = x) and
        (løvnode↑.y = y) then begin
            error('duplikate punkter');
        end
        else begin
            if (løvnode <> nil) or (p = nil) then begin
                {oppretter ny internnode}
                new (ny-intern);
                ny-intern↑.slag:= intern;
                for i:= 1 to 4 do
                    ny-intern↑.A[i]:= nil;
            end; {if}
            if p = nil then begin
                {den nye internnoden blir treets rot}
                rot-PR:= ny-intern;
```

```
                    forgjenger:= ny-intern;
                    kv:= Kvadrant(x,y,x1,y1,x2,y2);
                end;
                if løvnode = nil then begin
                    {ledig, kjeder inn ny løvnode}
                    new (ny-løv);
                    ny-løv↑.slag:= løv;
                    forgjenger↑.A[kv]:= ny-løv;
                    forgjenger↑.A[kv]↑.x:= x;
                    forgjenger↑.A[kv]↑.y:= y;
                end
                else begin {løvnoden er ikke ledig}
                    {kjeder inn den nye internnoden}
                    forgjenger↑.A[kv]:= ny-intern;
                    kv:= Kvadrant(løvnode↑.x,løvnode↑.y,x1,y1,x2,y2);
                    ny-intern↑.A[kv]:= løvnode;
                    Sett-inn-PR-tre(x,y,rot-PR);
                    {fortsetter oppdelingen}
                end;
            end; {if duplikate punkter}
        end; {if lovlige koordinater}
    end; {Sett-inn-i-PR-tre}
```

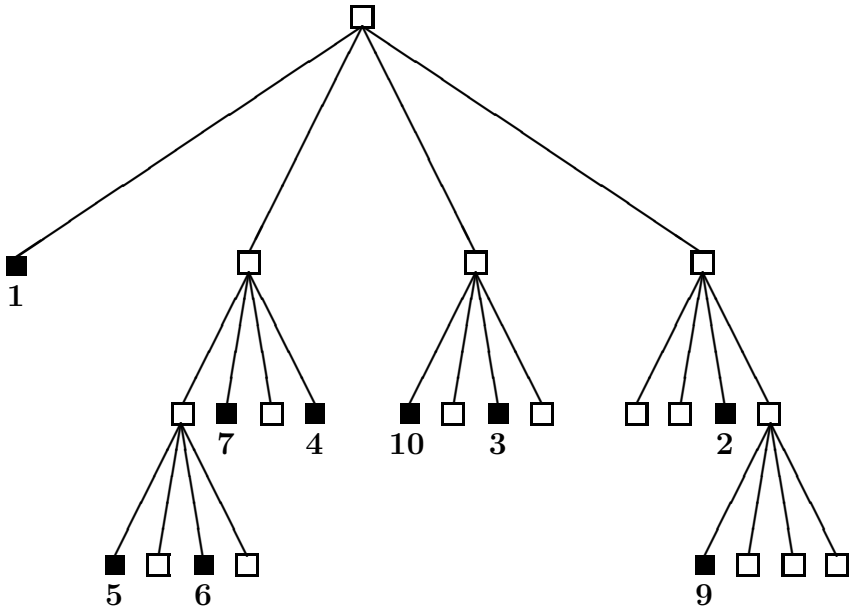
Sletting

Sletting av punkter er enkelt. På grunn av at dataene ligger i løvnodene, er det nemlig ikke nødvendig å gjøre noe med de interne noder, unntatt når forgjengeren p til en løvnode peker til mindre enn 2 løvnoder. I såfall flyttes den ene løvnoden opp til p sin forgjenger og p slettes. På den måten reduseres antall nivå i treet. For eksempel dersom punkt 8 i figur 6.6 slettes, vil forgjengeren nå peke til bare én løvnode (punkt 9). Punkt 9 kan derfor flyttes opp til sin forgjenger som vist i figur 6.7.

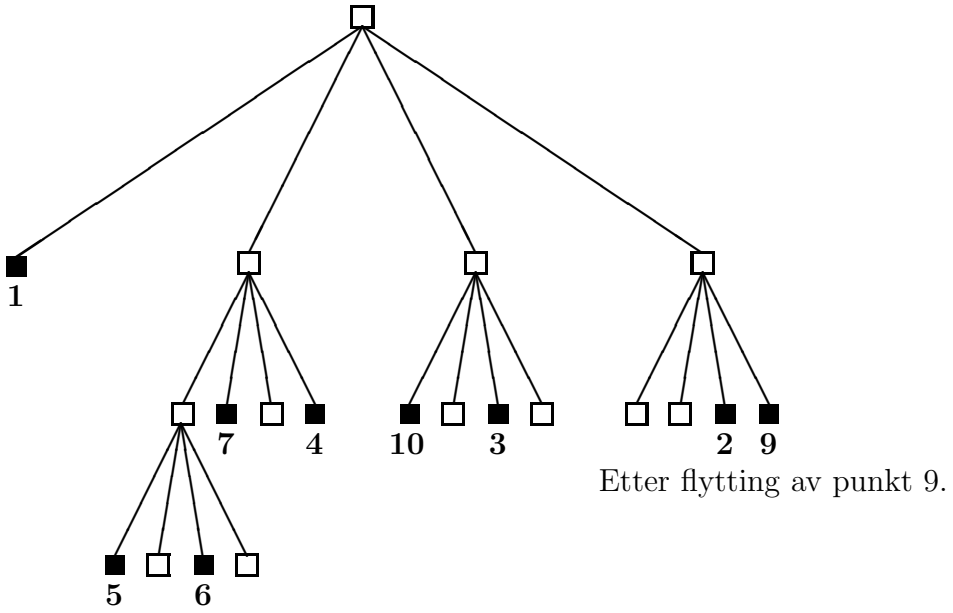
Områdesøk

Søk etter punkter som ligger innenfor et et gitt polygon, krever at samtlige kvadruter som ligger helt eller delvis innefor polygonet, besøkes. I værste fall er polygonet så stort at samtlige løvnoder i treet må besøkes. I beste fall er arealet til polygonet så lite at tidsforbruket blir $O(h)$. Det er selvsagt i det siste tilfellet at kvadtreets egenskaper kommer til sin fulle rett. Kvadtreets styrke ligger jo nettopp i dets evne til å plukke ut deler av en større mengde.

Eksempel: Dersom den typiske anvendelse er å tegne ut samtlige datapunkter innenfor det kartlagte området, har det neppe noe for seg å etablere en komplisert



Etter at 8 er slettet.



Etter flytting av punkt 9.

Figur 6.7: Endringer av PR-kvadtreet som følge av at punkt 8 blir slettet.

datastruktur. Det er først når det blir snakk om å hente fram punkter som ligger innenfor bestemte koordinatintervaller, at kvadtreet har noe for seg.

Finne nærmeste punkt

Et aktuelt søk i GIS-applikasjoner er å finne det punktet R som ligger nærmest et angitt punkt P.

Søketida kan minimaliseres ved å utnytte de topologiske egenskaper til PR-kvadtreet. Formuleringen av en algoritme vil i noen grad være avhengig av om treet er pekerbasert eller lineært. Hovedideen vil imidlertid være den samme. Den etterfølgende behandling av emnet går ikke i detalj på de to nevnte varianter PR-kvadtrær, men vil ha som formål å belyse hovedideene i en strategisk utnyttelse av PR-kvadtreet.

Søket starter med å finne den løvnode q_0 punktet P befinner seg i. Dersom q_0 ikke inneholder noen datapunkter, forsøkes neste skall. Slik fortsetter søket fra skall til skall inntil det blir funnet datapunkter som ligger så nære P at det ikke har noen hensikt å fortsette søket i ytterligere skall. Skallene kan vi sammenlikne med ringer som oppstår når en en stein kastes ut i et vann.

Vi må se litt nærmere på definisjonen av skallene. Det første skallet består av bare en kvadblokk og er gitt ved:

$$S_0 = \{q_0\}$$

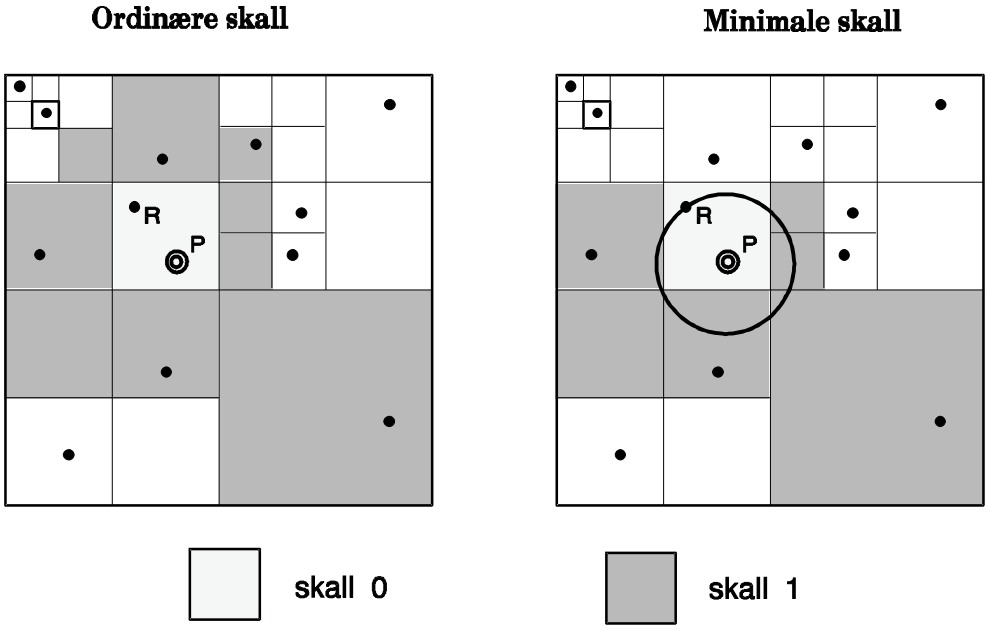
Det neste skallet består av naboblokkene til S_0. Som naboer regnes både sidenaboer og hjørnenaboer. På grunn av kvadtreets hierarkiske struktur, vil en rekke kvadblokker på ulike nivåer i treet være kandidater til å være naboer. Siden datapunktene ligger i løvnodene, er det disse nodene vi her vil definere som kandidater til naboer. Som kjent har en kvadblokk 4 sidenaboer av samme størrelse som blokken selv, men på grunn av den måten vi her definerer naboer på, vil antall sidenaboer kunne bli større enn 4.

Skall S_{i+1} kan etter dette defineres som:

$$S_{i+1} = \{q \mid q \in \mathcal{N}(S_i); i \geq 0\}$$

hvor $\mathcal{N}(S)$ er en funksjon som finner S sine naboer i samsvar med vår definisjon. Et skall i samsvar med definisjonen ovenfor, kaller vi *ordinært skall*. Se figur 6.8.

For at søket ikke skal fortsette i det uendelige, må det innføres et kriterium som terminerer prosessen. La R betegne det punktet som på et visst stadium i søket er funnet som nærmeste og la $\varepsilon = d(P, R)$ være den Evklidiske avstanden mellom P og R. En sirkulær skive med senter i P og radius ε vil vi kalle $U(P, \varepsilon)$. Termineringskriteriet innføres ved å forlange at kvadblokkene i S skal oppfylle den betingelsen at de må ligge helt eller delvis innenfor $U(P, \varepsilon)$. Et skall som tilfredsstiller dette kravet, vi vil benevne *minimalt skall* og skrive S' (merket). Et minimalt skall defineres som:



Figur 6.8: Definisjon av skall i et PR-kvadtre.

$$S_i' = \{q \mid q \in S_i \wedge q \cap U(P, \varepsilon) \neq \emptyset\}$$

Se illustrasjonen i figur 6.8. Søket terminerer så snart det aktive minimale skall er tomt. Altså når:

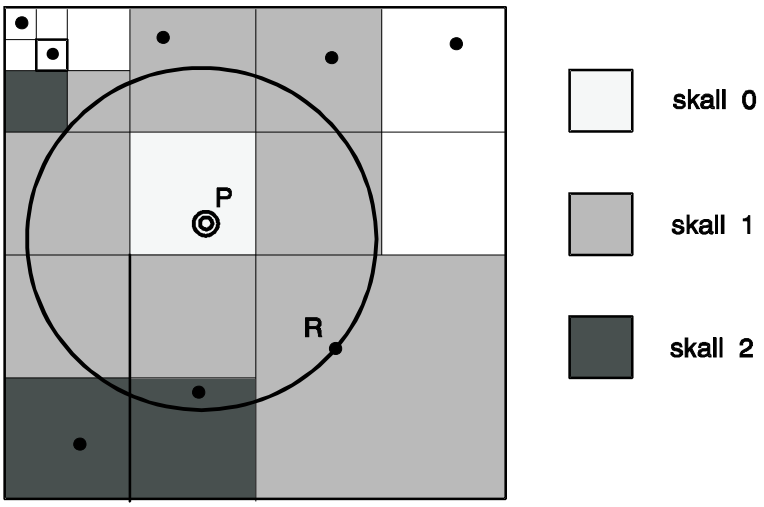
$$S_i' = \emptyset$$

Figur 6.9 illustrer algoritmen. Slik algoritmen nå er formulert, kan den karakteriseres som et *bredde først* søk.

Algoritmen er derimot svært enkel å formulere om vi benytter et *dybde først* søk ut fra punkt P. Her er en pseudokode for en slik formulering:

procedure *nærmeste-punkt (P: punkt,* **var** *R: punkt);*
{dybde først søk som finner det punkt R som ligger nærmest P}
begin
    q:=finn-rute(P); {den rute P befinner seg i}
    rekursiv-nærmeste (q);
end*;*

procedure *rekursiv-nærmeste (rute: trepeker);*
begin
    if *besøkt(rute)* **then return** {ruten er allerede besøkt}
    else if



Figur 6.9: Strategi for søk etter nærmeste datapunkt i et PR-kvadtre.

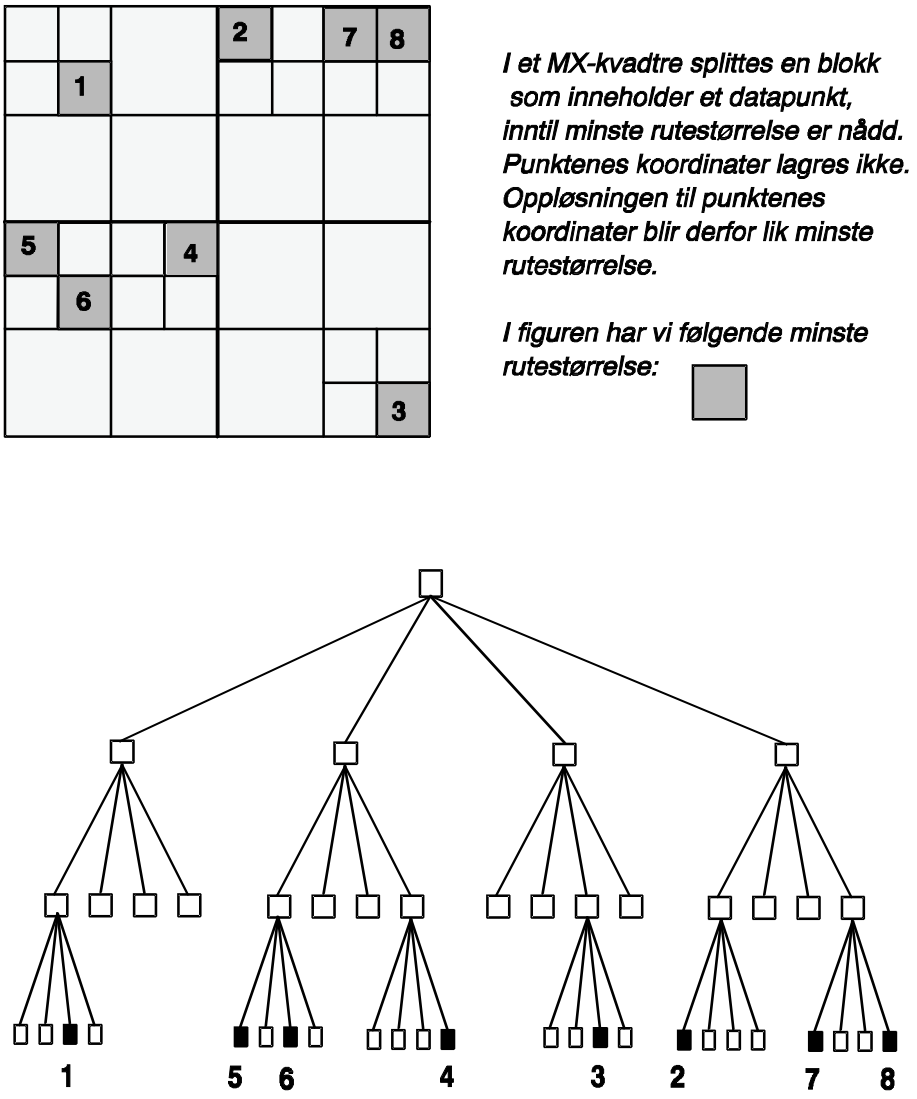
```
        merk-besøkt(rute); {ruten merkes besøkt};
        if rute ∩ sirkel(P,Smin) ≠ 0 then begin
            {ruten er helt eller delvis innenfor Smin}
            if rute↑.P ◁ P ⊢ R then R:= rute↑.P
            { ◁ leses: nærmere}
            { ⊢ leses: enn}
            rekursiv-nærmeste(N(rute)); rekursiv-nærmeste(S(rute);
            rekursiv-nærmester(V(rute)); rekursiv-nærmeste(Ø(rute));
            rekursiv-nærmeste(NV(rute)); rekursiv-nærmeste(SV(rute));
            rekursiv-nærmeste(NØ(rute)); rekursiv-nærmeste(SØ(rute));
        end;
    end;
end;
```

Siden algoritmen er formulert som et *dybde først* søk, tar den ikke skall for skall i forhold til P. I en implementasjon anbefales bredde førstsøk på grunn av at denne metoden i gjennomsnitt vil besøke færre noder enn foregående formulering. En implementasjon basert på ideene i figur 6.9 er derfor å anbefale framfor dybde førstsøket.

6.3.6 MX-kvadtre

Et *MX-kvadtre* (eng. MX quadtree, MX er forkortelse for matrix) har lignende egenskaper som PR-kvadtreet med unntak av at punktenes koordinater blir assosiert med posisjonen til vedkommende kvadblokk. MX-kvadtreet kan derfor klassifiseres som en super heldekkendemodell av datapunktene. Et MX-kvadtre gir kun en tilnærmet angivelse av punktenes koordinater der nøyaktigheten er avhengig av hvor små løvnoder som tillates.

En kvadblokk som inneholder et datapunkt, splittes inntil minste tillatte rutestørrelse er nådd, se figur 6.10.



Figur 6.10: Et MX-kvadtre og dets oppdeling av planet.

Høyden til et MX-kvadtre kan beregnes av:

$$h = \lceil \log_2(s/d) \rceil$$

hvor s er sidekanten til det kartlagte områdets omskrivende rektangel og d er løvnodenes sidekant.

Som vi ser er trehøyden kun en funksjon av størrelsen på det omskrivende rektangel (rotblokkens størrelse) og minste blokkstørrelse (løvnodenes størrelse). Treets fasong er uavhengig av den rekkefølgen datapunktene settes inn i treet.

Et MX-kvadtre er en fornuftig datastruktur så lenge punktene har en viss minsteavstand og dersom nøyaktigheten til punktene er tilfredsstillende etter avrunding til kvadblokkens posisjon.

Et eksempel på en aktuell envendelse av MX-kvadtrær har vi fra sjøkartlegging. I dag benyttes i stor grad flatedekkende sensorer (multistråle-ekkolodd). Et multistråle-ekkolodd måler dybden til flere punkter samtidig (32 punkter). Ved å montere ekkoloddet ombord i en båt, får man dekket havbunnen med et finmasket nett av målepunkter. Et MX-kvadtre over disse datapunktene etbaleres ved å lagre dybdeverdier i de respektive løvnoder. Punktenes (x, y)-koordinater assosieres med løvnodenes posisjon. Dersom flere målepunkter skulle falle innenfor samme løvnode, kan man beregne en gjennomsnittlig høydeverdi. Oppløsningen til treet (løvnodenes størrelse) må velges ut fra målenøyaktigheten og hvilke krav som stilles til nøyaktigheten av beskrivelsen av terrengoverflaten.

Ved å benytte et lineært kvadtre, har vi her et konsept som passer lett inn i en *relasjonsdatabase*. Alle noder i treet får tildelt unike nøkler gjennom sine posisjonskoder. Videre er det en enkel sak å konvertere fra posisjonskoden tilbake til $(x, y)-$koordinater. Riktignok med det tap av nøyaktighet som skyldes avrundingen til kvadblokkens posisjon.

```
type
    MX-node = record {løvnode i et lineært MX-kvadtre}
        posisjonskode, dybde: integer;
    end;
```

Det lineære kvadtreet byr på den fordel at vi bare behøver lagre løvnodene i treet og av disse bare de nodene som inneholder datapunkter. På grunn av at alle løvnoder som inneholder datapunkter ligger på samme nivå i treet, er det strengt tatt ikke nødvendig å ta med en nivåangivelse i posisjonskoden.

Det er selvsagt ikke noe til hinder for å utvide MX-kvadtreet til et konsept for en ekte tredimensjonal modell. I dette tilfellet trenger vi bare å merke de løvnoder som inneholder datapunkter. Dersom vi snakker om en terrengmodell, vil mengden av merkede noder befinne deg på modellens omriss. Der hvor de merkede noder eventuelt ikke danner en sammenhengende mengde, kan hullene tettes til ved interpolasjon. Etter dette vil de interpolerte- og de observerte punkter tilsammen definere en sammenhengende mengde terninger på terrengmodellens overflate.

6.3.7 PM-kvadtrær

PM-kvadtrær er en gruppe av beslektede kvadtrær. PM-kvadtrær har stor likhet med PR-kvadtrær. Forskjellen stikker i at PM-kvadtrær lagrer informasjon om linjer.

Språklig sett hadde det vært fordelaktig om treet hadde fått betegnelesen LR-tre slik at slektskapet til PR-treet ble vist. PM kommer forøvrig av den engelske betegnelsen polygonal map. Samet [Sam90b] nevner 5 varianter PM-kvadtrær. Fire

av variantene betegnes PM_1, PM_2, PM_3 og PM_4-kvadtre mens den femte varianten kalles PMR-kvadtre. Variantene skiller seg fra hverandre ved hvilket kriterium som benyttes for å splitte en kvadblokk.

PM_3-kvadtre

I et PM_3-kvadtre kan en løvnode inneholde inntil ett knekkpunkt. Derimot gis det ingen øvre grense for hvor mange linjesegmenter en løvnode kan inneholde. Teoretisk sett kan uendelig mange linjesegmenter møtes i et enkelt punkt. Med linjesegment menes her den rette forbindelseslinje mellom to punkter.

splittkriterium som for PM_3 kvadtreet.

En node i PM_3-treet inneholder følgende informasjon:

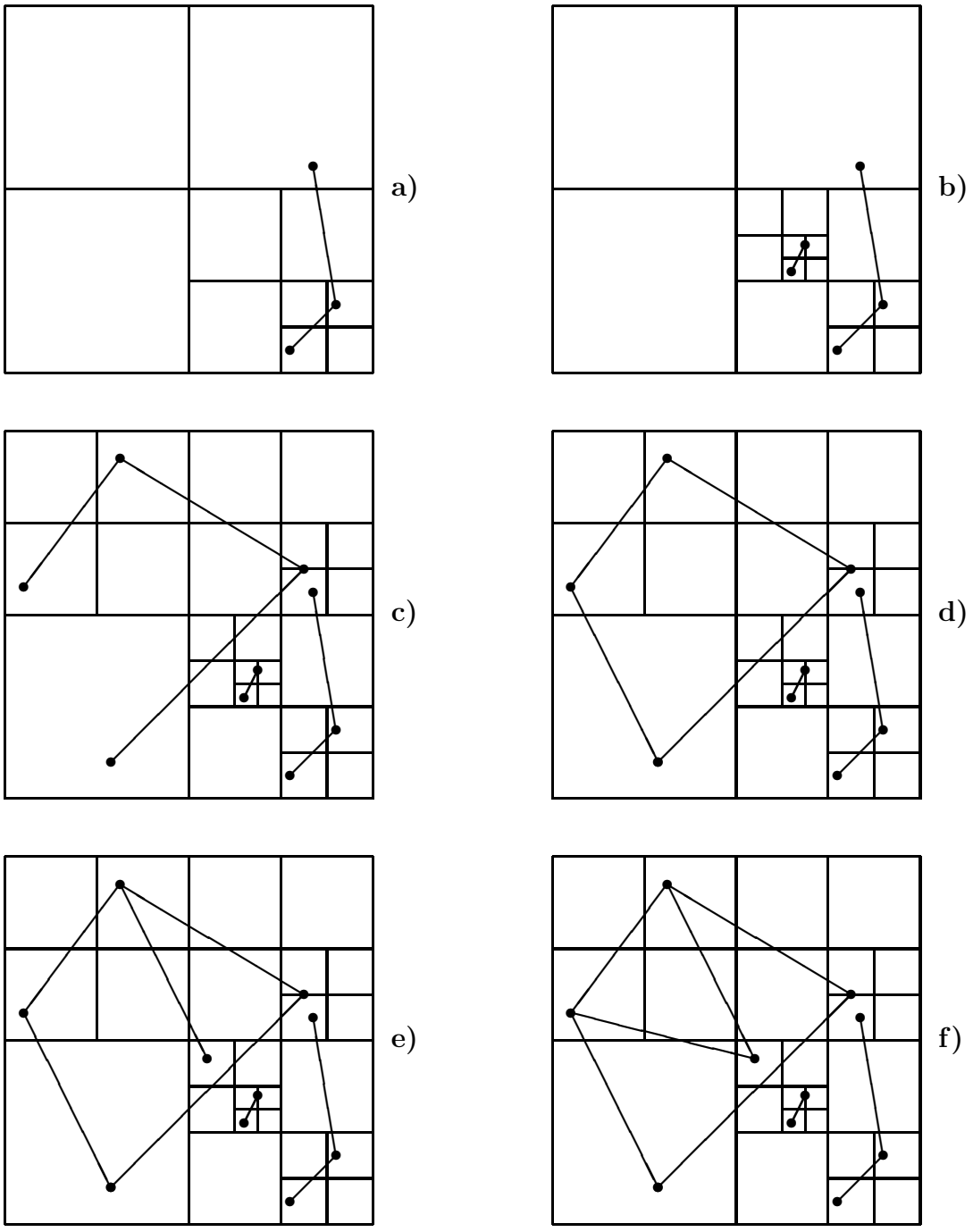
1. felt som angir om en node er en internnode eller en løvnode.
2. pekere til nodens etterfølgere.
3. felt som definerer størrelsen på kvadruten samt koordinatene for dens sentralpunkt.
4. liste over de linjesegmenter som befinner seg helt eller delvis innenfor kvadblokken.

Denne definisjonen av PM_3-kvadtreet lar ikke topologisk informasjon komme eksplisitt til uttrykk. Topologi kan selvsagt avledes av geometrien, men for en rekke GIS-anvendelser vil det lett føre til uakseptable kjøretider om den geometriske topologi ikke er eksplisitt definert i datastrukturen. Foreslå en utvidelse av datastrukturen for PM_3-kvadtreet slik at et polygon kan eie ett eller flere linjesegmenter og at et linjesegment kan ha et areal (lukket polygon) til venstre og et areal til høyre for seg.

Her er en pseudokode for datastrukturen i et PM_3-kvadtre:

```
type
    punkt-peker: ↑punkt;
    punkt = record {liste av punkter}
        x,y: integer;
        neste: punkt-peker
    end;

    l-peker: ↑l-segment;
    l-segment = record {liste av linjesegmenter}
        frapunkt,tilpunkt: punkt-peker;
        neste: l-peker
    end;
```



Figur 6.11: Innsetting i et PM_3-kvadtre, splittfaktor ett knekkpunkt.

```
k-peker = ↑k-segment;
k-segment = record
      { liste av linjesegment i en kvadblokk}
      l-liste: l-peker;
      neste: k-peker
end;

nodetype = (løv,intern);
trepeker = ↑PMkvadnode;
PMkvadnode = record {Kvadtrenode}
      case slag: nodetype of
            intern: (NV,NØ,SV,SØ: trepeker);
            løv: (
                  x,y: integer; {sentralpunkt i kvadblokken}
                  sidekant: integer; {blokkens dimensjon}
                  k-liste: ↑k-segment; {liste av linjesegmenter}
                  );
end;
```

PMR-kvadtre

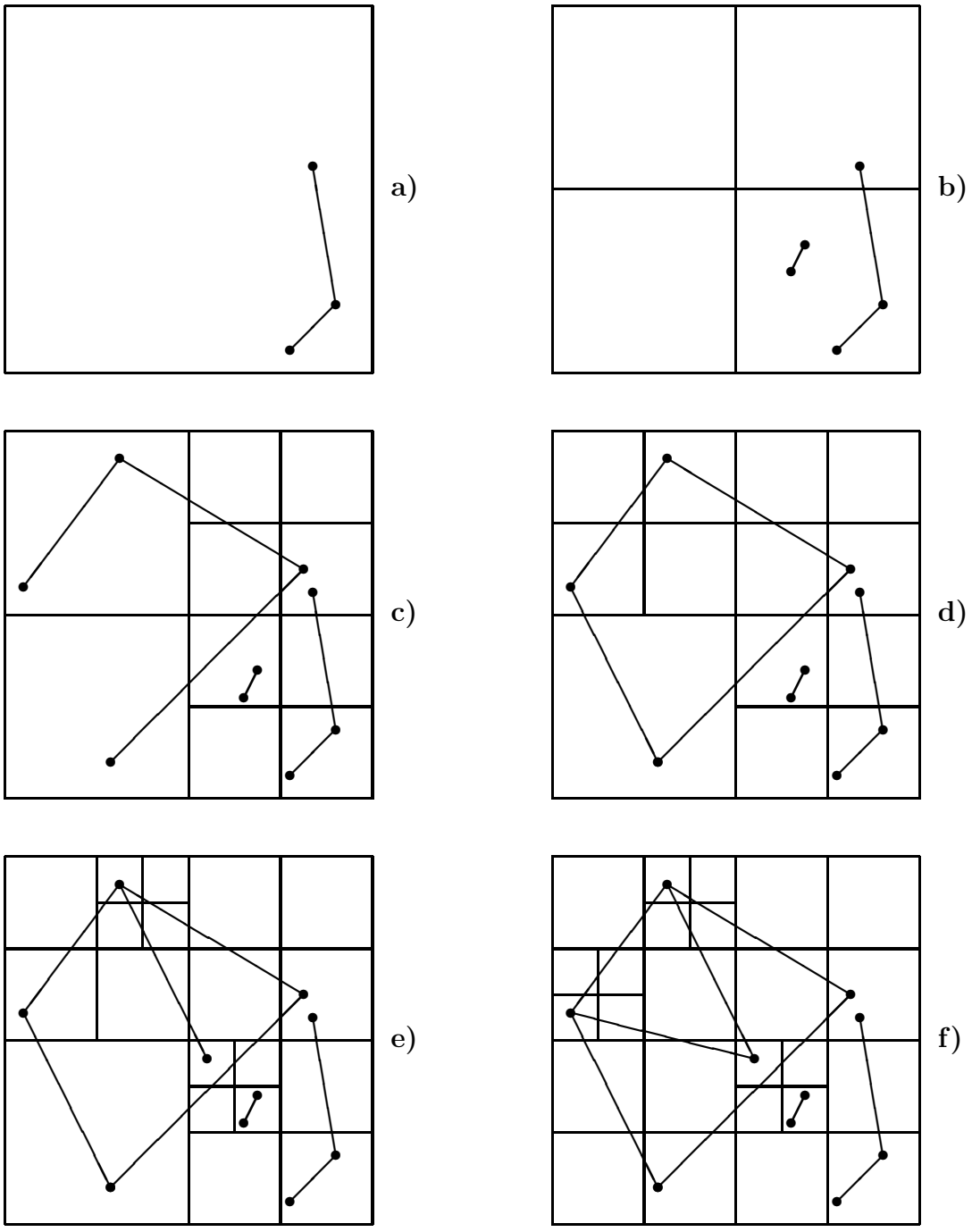
I et PMR-kvadtre settes splittkriteriet etter antall linjesegmenter i en kvadblokk. På grunn av at uendelig mange linjesegmenter kan stråle ut fra ett enkelt punkt, vil en oppdeling i stadig mindre ruter ikke garantere at splittkriteriet blir tilfredsstillet. Derfor innføres et system med overløpsblokker. Når en blokk går full, splittes den en gang og bare en gang. Dersom blokken fortsatt er full, legges overskytende informasjon inn i en overløpsblokk.

Figur 6.12 viser prinsippet for innsetting i et PMR-kvadtre. Splittfaktoren er i eksemplet valgt lik 2. Vi starter med bilde a) der vi har 2 linjesegmenter. I bilde b) er et nytt linjesegment satt inn slik at kvadblokken nå inneholder 3 segmenter. Dette fører til en splitt, men på grunn av at splittlinjene faller litt uheldig, vil blokk SØ fortsatt inneholde 3 linjesegmenter. På grunn av at vi har valgt å splitte en blokk bare en gang, opprettes en overløpsblokk for blokk SØ. I bilde c) får blokk SØ tilført enda et linjesegment. Dette gir oss anledning til å splitte blokken. Denne gangen faller splittlinjene heldig, slik at blokk SØ nå får løst sitt overløpsproblem.

PM-kvadtre med topologisk informasjon

For anvendelser i GIS-sammenheng, spiller topologi en sentral rolle. Hittil er vår definisjon av PM-kvadtreet basert på en spaghetti-modell av geometrien til de kartlagte objekter.

Vi kan bøte på dette ved å utvide nodene i treet med informasjon om topologi. Topologisk informasjon som kan være aktuell for GIS-anvendelser er: hvilket polygon



Figur 6.12: Innsetting i et PMR-kvadtre med splittfaktor 2.

et linjesegment tilhører og hvilket areal (lukket polygon) som ligger henholdsvis til høyre og til venstre for et linjesegment. Et areal defineres her ved sin omrissfigur. Dersom et linjesegment har samme areal til høyre som til venstre for seg, betyr det at linjesegmentet ligger innenfor et og samme areal.

Noe som er typisk for kart, er korte linje-segmenter (tett med punkter). Dette gjør det hensiktsmessig å gruppere linjesegmenter som har felles topologisk informasjon. En slik gruppe linjesegmenter kalles et *kjede*. Derved kan den topologiske informasjon som er felles for alle linjesegmentene i et kjede, knyttes til kjedet. På denne måten vil både plass og kjøretid bli redusert i forhold til om all topolgisk informasjon ble knyttet til hvert enkelt linjesegment. Forutsetningen er selvsagt at kjedene defineres på en hensiktsmessig måte.

Beregning av *knutepunkter* (skjæringspunkter) mellom linjesegmentene spiller en sentral rolle for å kunne finne topologien og gjøre en fornuftig definisjon av kjedene. Kjedene defineres gjerne fra knutepunkt til knutepunkt med den restriksjon at et kjede kun kan være grenselinje mellom to polygoner og at kjedene ikke skal overlappe hverandre. Disse forutsetninger innebærer at kun endepunktene i et kjede kan være forgreiningspunkter.

Den datamodell som her foreslås, oppfatter polygoner som sammensatt av kjeder og kjeder som sammensatt av linjesegmenter. Datastrukturen vil derfor bestå av følgende mengde:

- mengden av alle polygoner: $P = \{P_i \mid i = 1, n\}$
- et polygon er en mengde av kjeder: $P_i = \{K_i \mid i = 1, m\}$
- et kjede er en mengde av linjesegmenter: $K_i = \{l_i \mid i = 1, n\}$

Polygonene såvel som kjedene og linjesegmentene, tildeles unike identifikatorer (ID). Siden et linjesegment kun kan inngå i et enkelt kjede, er det tilstrekkelig at linjesegmentenes ID er unik innenfor den aktuelle kjeden.

Topologien innenfor et enkelt kjede er det hensiktsmessig å beskrive implisitt ved å nummerer linjesegmentene slik at linjesegment l_i har felles endepunkter med linjesegmentene l_{i-1} og l_{i+1}. Nummerrekkefølgen reflekterer altså den rekkefølge linjesegmentene vil bli besøkt i om et kjede traverseres fra den ene enden til den andre anden.

Kjedene tildeles også et nummer, men på grunn av at et kjede kan inngå i flere polygoner, er det enklest å ta en global nummerering av kjedene fra 1 til N.

I en kvadblokk skal det finnes informasjon om hvilke kjeder som befinner seg helt eller delvis innenfor blokka, men det vil sannsynligvis også være lurt å lagre informasjon om hvilke linjesegmenter som ligger innenfor blokka. Måten dette kan gjøres på, er å oppgi første og siste linjesegment som ligger innefor den aktuelle

blokken, alså ved å oppgi m og n i delmengden k_{mn} av K_i hvor $k_{mn} = \{l_i \mid i = m, n\}$. Dersom kjede K_i passer blokken flere ganger, må i såfall oppgis flere delmengder av K_i.

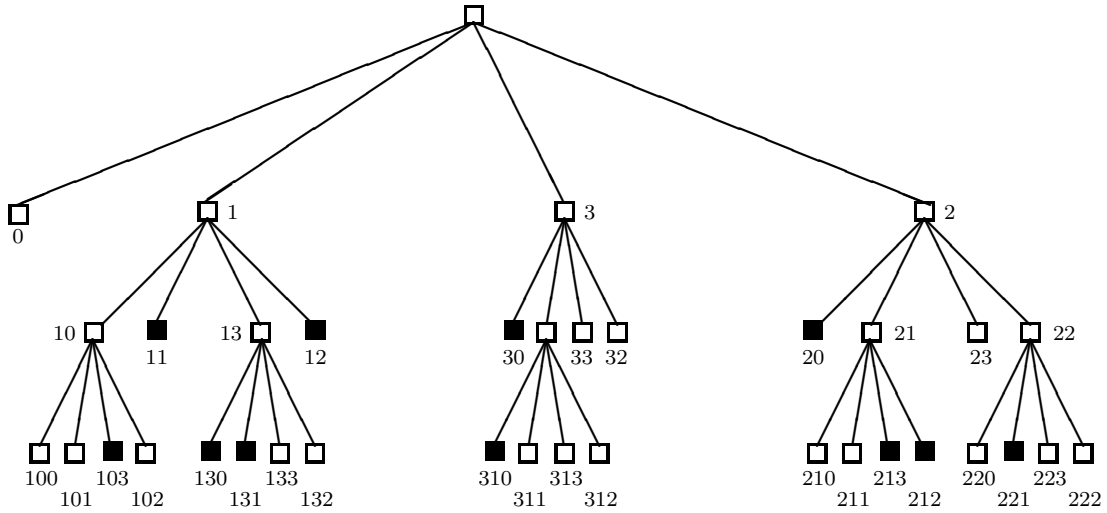
Ved valg av datastrukturer for de nevnte mengdene, må det legges vekt på at innsetting, splitting, sammenslåing samt sletting og gjenfinning kan gjøres så effektivt som mulig. Samet foreslår her at et 2-3tre benyttes, men også andre valg kan være aktuelle.

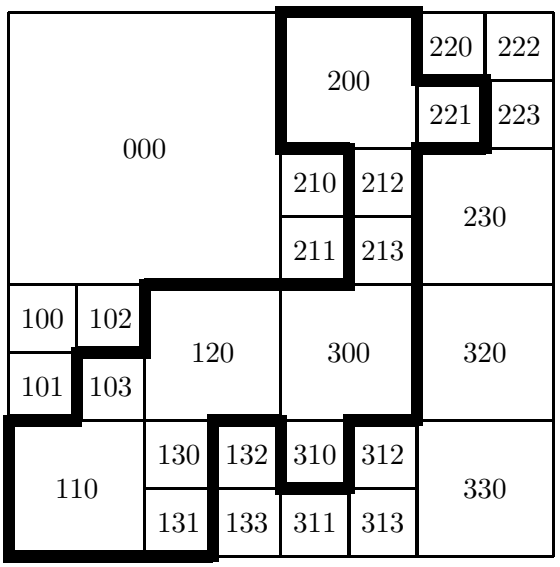
6.3.8 Regionkvadtre

Regionkvadtre (eng. region quadtree) er et regulært kvadtre for rasterkart. I et regionkvadtre splittes en kvadblokk inntil alle rasterruter som ligger innefor kvadblokka, er av samme farge. Regionkvadtreet av et binærbilde vil derfor ha hvite eller svarte løvnoder mens de interne noder i treet er gråe. Figur 6.13 gir et eksempel på et binærbilde og dets tilhørende regionkvadtre.

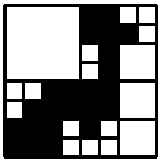
De hvite og grå noder i kvadtreet er ikke nødvendig å lagre om vi benytter et lineært kvadtre. Det lineære kvadtreet til rasterbildet i figuren er gitt ved posisjonskodene: 103-3, 110-2, 120-2, 130-3, 131-3, 300-2, 310-3, 200-2, 212-3, 213-3, 221-3; hvor -x angir nodens nivå.

I værste fall er det ikke mulig å danne homogene kvadblokker (sjakkbrett-tilfellet) før kvadblokkene er på elementærrrute-nivå. I beste fall derimot er samtlige rasterruter av samme farge. For GIS-anvendelser er de to ekstreme tilfellene lite sannsynlige. Det typiske er at regionkvadtreet til et kart krever langt mindre lagerplass enn matriserepresentasjonen av kartet.





Figurens binærbilde



Figur 6.13: En figur og dens regionkvadtre.

6.4 Irregulær firedeling av planet

Definisjon 36 *Et irregulært kvadtre foretar en rekursiv oppdeling av et todimensjonalt rom i fire komponenter slik at delelinjene kan ha en vilkårlig retning og en vilkårlig plassering innenfor det rommet som skal deles.*

Definisjonen stiller ikke så strenge krav til et irregulært kvadtre som til et regulært kvadtre. Begrepene regulært- og irregulært kvadtre er forøvrig innført av forfatteren.

6.4.1 Punktkvadtre

Punktkvadtreet er en datastruktur for 0-dimensjonale objekter, og det deler planet som vist i figur 6.14. Som det framgår av figuren, ligger delelinjene parallelt med koordinataksene og går gjennom datapunktene. Siden en node i treet bare inneholder inntil ett datapunkt, er punktkvadtreet en datastruktur som ikke egner seg for platelagere. Vi har derfor her å gjøre med en intern datastruktur.

De interne noder i treet vil alltid inneholde datapunkter mens løvnodene kan være tomme.

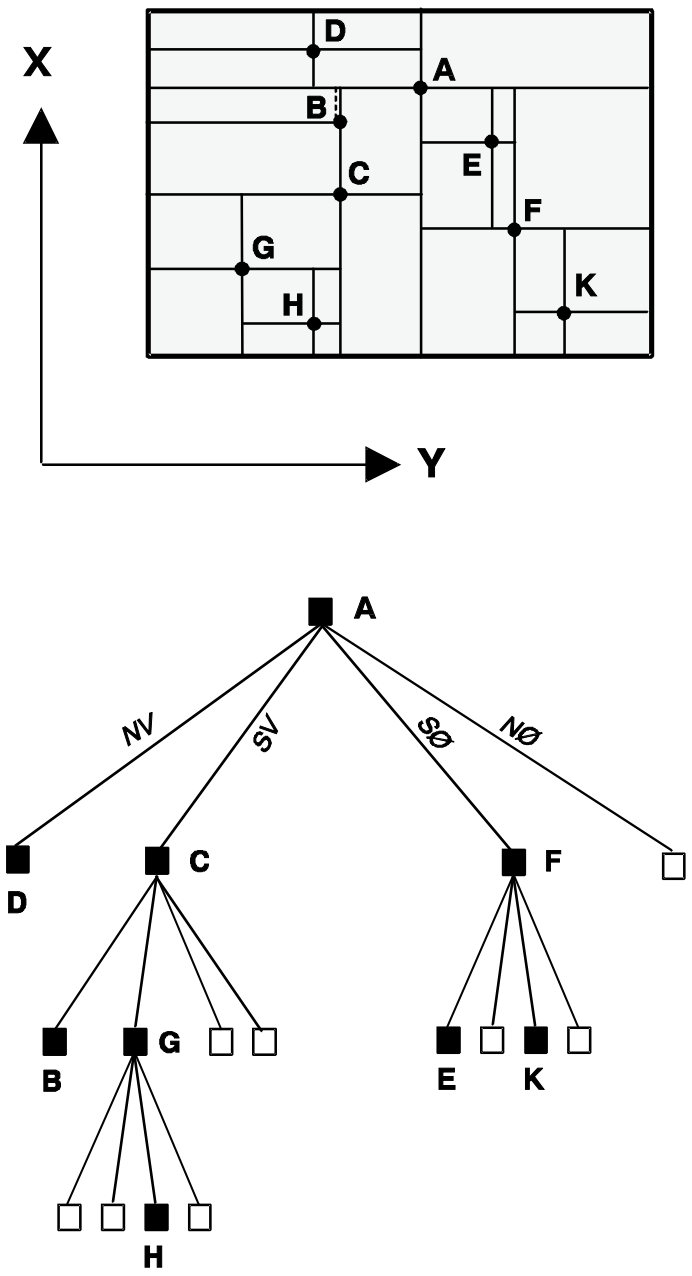
Datastrukturen for et punktkvadtre kan defineres slik:

```
type
    trepeker = ↑trenode;
    trenode = record {node i et punktkvadtre}
        nø,nv,sø,sv: trepeker;
        x,y: integer
    end;
```

Innsetting

Innsetting i punktkvadtreet følger en logikk som minner om innsetting i et PR-kvadtre. Siden algoritmen for PR-kvadtreet ble gjennomgått i detalj, vil vi derfor nøye oss med å angi hovetrekkene i en algoritme for innsetting i et punktkvadtre.

```
procedure SettInn-punktkvadtre(rot: trepeker; x,y: integer);
    {Rutine for å sette punkter inn i et punktkvadtre}
    {rot: er peker til treets rot}
    {x,y: koordinatene til det punktet som skal settes inn}
var
    p,forgjenger: trepeker;
begin
    p:=rot;
    while p <> nil do begin {traverserer stien fra treets rot}
    {til den noden som vil bli (x,y) sin foreldrenode}
```



Figur 6.14: Et punktkvadtre og dets oppdeling av planet.

```
            forgjenger:=p; {husker forgjengeren}
            p:=Grein(p,x,y); {neste node i stien}
        end;
        new (p); {oppretter ny node}
        Lagre(p,x,y); {lagrer (x,y) i den nye noden}
        KjedSammen(forgjenger,p); {kjeder den nye noden sammen}
        { med (x,y) sin forgjenger}
    end;
```

Vi skal se litt nærmere på hvilke faktorer som bestemmer fasongen til et punktkvadtre, men først trenger vi en definisjon.

Definisjon 37 *En sti fra treets rot til node i benevnes l_i. La alle noder som inneholder datapunkter være nummererte fortløpende fra 1 til N. Total stilengde defineres da som $L = \sum_i^N l_i$.*

Fasongen til punktkvadtreet avhenger av den rekkefølgen punktene settes inn i treet. I figur 6.14 er punktene satt inn i rekkefølgen (A),(D,C,F), (B,G,E,K),(H). Her betyr parantesene at rekkefølgen av punktene innenfor parantesen ikke er av betydning for treets fasong. For eksempel er (D,C,F) ekvivalent med (C,D,F). Dersom rekkefølgen byttes til G,H,D,B,C,A,E,F,K, får vi et tre som er temmelig dårlig balansert. Det innebærer at om vi beregner total stilengde i treet, vil den siste punktfølgen gi en større L enn den første punktfølgen.

Kostnadene med innsetting i treet minimaliseres ved å gjøre L så liten som mulig.

Balansering av punktkvadtreet

Når vi bruker begrepet balansering i forbindelse med punktkvadtrær, skal vi være klar over at begrepet brukes i en mindre restriktiv betydning. For eksempel i det tilfellet at punktene ligger langs en rett linje, vil punktkvadtreet kunne utnytte bare to av sine fire kvadranter. Dersom vi mener balansering i en restriktiv betydning, vil vi bruke termen perfekt balansert. For å gjøre denne presiseringen klarere, gjør vi følgende definisjoner:

Definisjon 38 *I et balansert punktkvadtre er total stilengde så liten som den aktuelle punktmengden tillater.*

Definisjon 39 *I et perfekt balansert punktkvadtre er et nytt nivå ikke påbegynt før alle tidligere nivåer er fylt helt opp. Løvnodene i treet tilhører derfor enten nivå i eller nivå $(i + 1)$.*

Et perfekt balansert punktkvadtre kan oppfattes som et spesialtilfelle av et balansert punktkvadtre. Av definisjonene følger at et perfekt balansert punktvadtre kun kan opprettes i de tilfeller at datapunktene har en spesielt gunstig romlig utbredelse.

I litteraturen forekommer uttrykket *optimalisert* kvadtre om det mindre restriktive begrepet balansert punktkvadtre.

I forbindelse med gjennomgangen av sortering, ble det vist en todimensjonal variant av quicksort. Under den forutsetning at det ikke er tilgang eller avgang av datapunkter, kan prosedyren hierarkisk-sortering i figur 2.30 anvendes for å opprette balanserte punktkvadtrær. Dette forutsetter at alle punkter som skal settes inn i treet, må være gitt før treet opprettes. Prosedyren for å opprette et balansert tre blir derfor: kjør først hierarkisk-sortering gjør deretter innsetting i punktkvadtreet i den sorterte rekkefølgen.

Den mest ugunstige punktfordelingen for et balansert punktkvadtre, er når punktene ligger slik til at bare to kvadranter blir utnyttet. I beste fall derimot er punktfordelingen slik at hver oppdeling fører til at vedkommende punktmengde blir delt i fire delmengder som hver inneholder like mange elementer. En slik punktmengde vil føre til at vi får et *perfekt balansert* punktkvadtre. Trehøyden i et *balansert* punktkvadtre med N datapunkter vil av dette være gitt ved:

$$\log_4 N \leq h \leq \log_2 N$$

Det værste tilfellet er lite sannsynlig, slik at for de praktiske anvendelser kan h påregnes å ligge et sted mellom de angitte grenseverdier. Dersom vi kan forutsette at punktmengden passer inn i et *perfekt* balansert punktkvadtre, vil kostnadene ved den romlige sorteringen såvel som kostnadene ved innsettingen av den sorterte punktfølgen i treet være $O(N \cdot \log_4 N)$. Dette gjør at de totale kostnader ved å etablere et perfekt balansert punktkvadtre blir av graden $O(N \cdot \log_4 N)$.

Sletting

Sletting i et punktkvadtre blir noe innviklet om man både ønsker å minimalisere tidsforbruket og å holde treet balansert. Vi skal ikke gå i detalj på dette punktet, men skissere noen enkle løsninger.

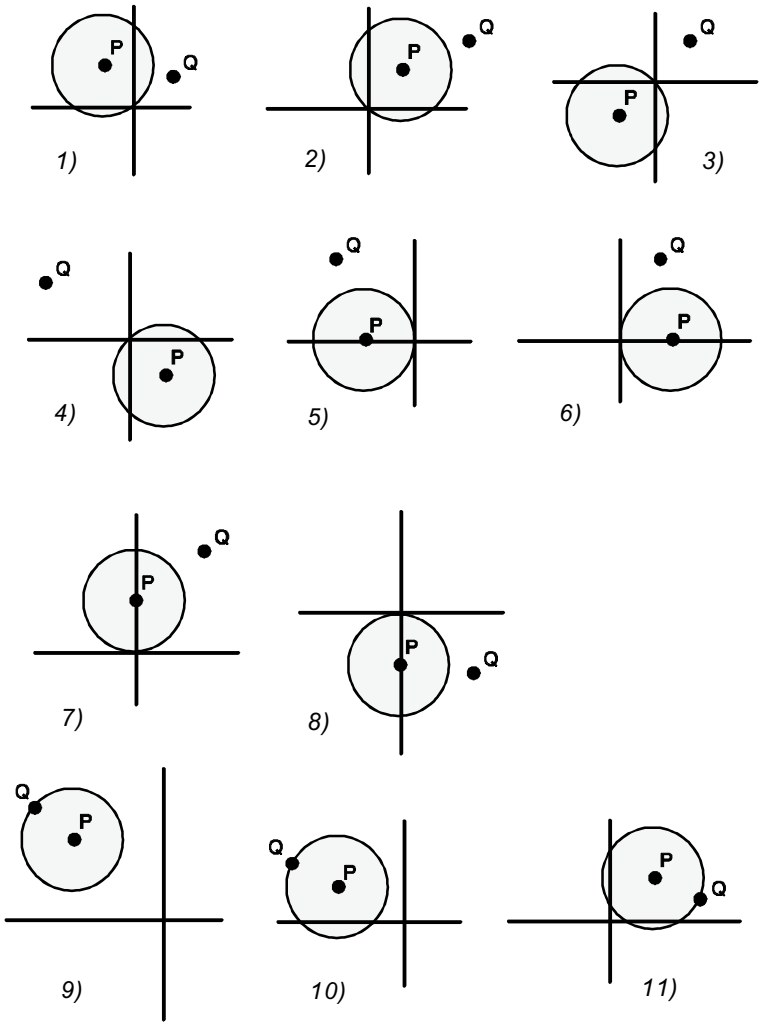
En nærliggende tanke er å merke de punkter som skal slettes, og la de fortsatt beholde sin plass i treet. Deres funksjon blir da kun som veivisere. Dersom det blir mange slike punkter i treet, vil trehøyden kunne bli merkbart stor og spørsmålet om en restrukturering vil melde seg.

En annen enkel metode er å fjerne punktet fra treet og restrukturere dets subtre, eventuelt bygge opp hele treet på nytt.

Praksis viser at for en rekke GIS-anvendelser er sletting av punkter en mindre aktuell problemstilling.

Søk etter nærmeste punkt

Søk etter nærmeste punkt kan formuleres på flere måter, men det som her er viktig er å finne en formulering som utnytter punktkvadtreets topologiske egenskaper.



Figur 6.15: Kvadranter som kan inneholde kandidater til nærmeste punkt.

La oss anta at vi traverserer treet på en eller annen måte og at vi søker nærmeste punkt til P. Det punktet som til enhver tid er funnet som nærmeste kaller vi Q. Den Evklidiske avstand mellom P og Q er gitt ved $\varepsilon = d(P, Q)$ og sirkelskiven med senter i P og radius ε benevnes $U(P, \varepsilon)$.

Vi skal nå angi en foreløpig formulering av en algoritme. Anta at vi står i en node i kvadtreet. Det første spørsmålet som her må besvares, er om datapunktet i den noden vi står i ligger nærmere P enn det Q gjør. Avstanden fra P til nodens datapunkt må derfor beregnes og holdes opp mot ε. Dersom nodens datapunkt ligger nærmere P enn det Q gjør, oppdateres Q og ε. Bildene 1) til 8) i figur 6.15 viser varianter av denne situasjonen. I motsatt fall beholdes Q. Se bildene 9) til 11) i figur 6.15.

Det neste spørsmålet som må besvares, er hvilke av nodens fire kvadranter som kan inneholde punkter som er kandidater til nærmeste. Her har vi at alle kvadranter

som helt eller delvis overlapper $U(P, \varepsilon)$, kan inneholde kandidater til nærmeste. Disse kvadrantene må derfor besøkes. Som det framgår av figur 6.15 vil antall slike kvadranter kunne variere fra 1 til 3.

Før vi forsøker å optimalisere algoritmen, skal vi se på tidsforbruket til algoritmen slik den nå foreligger. Den etterfølgende diskusjonen baserer seg på forutsetningen om at punktkvadtreet er *perfekt balansert*. Dette er selvsagt en idelle forutsetning som vil være oppfylt i større eller mindre grad.

Et perfekt balansert kvadtre med høyde h kan maksimalt inneholde dette antall noder:

$$N = \frac{4^{h+1} - 1}{3}$$

Dersom N er stor, kan vi med god tilnærmesle skrive:

$$h = \log_4 N \tag{6.1}$$

I værste og i beste fall må henholdsvis tre og en kvadrant besøkes. Av dette kan vi angi en øvre og en nedre grense for det *gjennomsnittlige* tidsforbruket $T(N)$. Den øvre og den nedre grense for $T(N)$ vil nemlig være kostnadene ved å besøke alle noder i et tre med høyde h og forgreiningsfaktor henholdsvis 3 og 1. Når c er kostnadene ved å besøke en node har vi:

$$c(h+1) \leq T(N) \leq c\frac{3^{h+1} - 1}{3 - 1} \approx \frac{c}{2}3^{h+1} < cN$$

som ved å benytte O-notasjon skrives som:

$$O(h) \leq T(N) \leq O(3^{h+1}) < O(N) \tag{6.2}$$

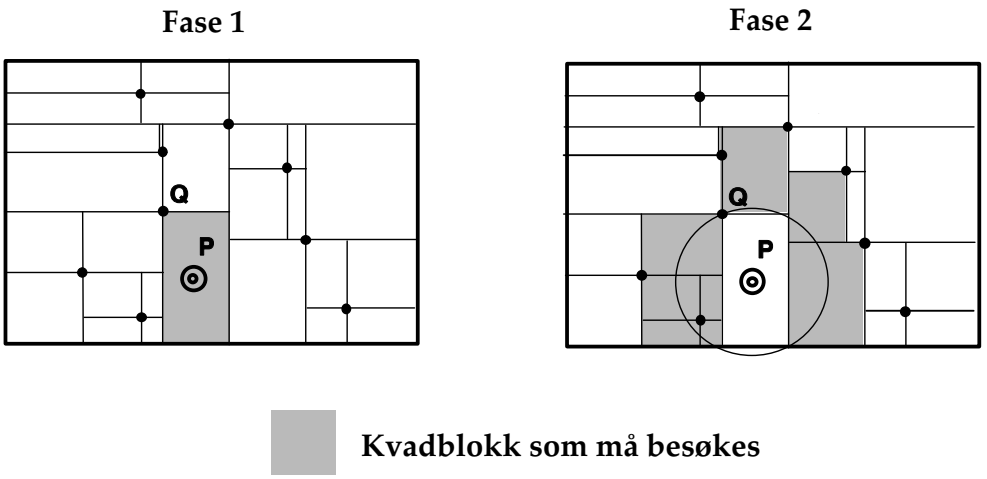
Ved å innføre uttrykket for h fås:

$$O(\log_4 N) \leq T(N) \leq O(3^{1+\log_4 N}) < O(N) \tag{6.3}$$

Den øvre grense er egentlig ikke akseptabel. Dette kan vi imidlertid gjøre noe med ved å skaffe oss en god startverdi for ε. Det er her to veier å gå. Enten innføre startverdien på grunnlag av en antagelse om hvor stor avstanden til nærmeste punkt kan være, eller benytte en beslekted strategi som vi utledet for PR-kvadtreet.

I det siste tilfellet traverseres treet to ganger. Først traverseres stien til den kvadblokk P tilhører, og for hver node som besøkes, oppdateres om nødvendig Q og ε. Denne operasjonen tar $O(h)$ tid. Etter at stien er traversert, er det stor sannsynlighet for at nærmeste punkt allerede er funnet, men det er en mulighet for at det finnes punkter i treet som ligger enda nærmere P enn det Q gjør. Dette gjør at punktkvadtreet må traverseres en gang til. Se figur 6.16.

Figur 6.16 illustrerer de to faser i søket. Man må her være oppmerksom på at blokker som er naboer i planet, ikke nødvendigvis er naboer i treet. Dette innebærer at fase 2 starter i treets rot.



Figur 6.16: Strategi for søk etter nærmeste punkt i et punktkvadtre.

Det andre søket representerer en kritiske fase med omsyn til tidsforbruket. Dersom den verdien vi har funnet for ε er god, kan vi håpe at det i fase 2 er nok å besøke bare en av etterfølgerne til de interne noder. Vi håper altså på situasjonen i bilde 9 i figur 6.15. Dersom dette er holder stikk, vil tidsforbruket for å finne nærmeste punkt bli $O(h)$.

Sammenlikning mellom punktkvadtre og PR-kvadtre

For å velge mellom de nevnte datastrukturer, kan man evaluere følgende punkter:

- har datapunktene en jevn eller en svært ujevn fordeling i planet?
- er det ønskelig å etablere et lineært kvadtre?

PR-kvadtreets svakhet er at punkter som ligger nære hverandre krever mange delinger (stor trehøyde). Punktkvadtreets styrke er at det lettere tilpasser seg datapunktenes romlige fordeling. I et punktkvadtre kan for eksempel punkter ha uendelig liten avstand seg imellom uten at det skaper problemer for datastrukturen.

En fordel med PR-kvadtreets regulære oppdelingen, er at det blir mulig å etablere lineære kvadtrær. Denne muligheten eksisterer ikke for punktkvadtrær.

Fasongen til et PR-kvadtre er kun avhengig av punktenes romlige fordeling, mens punktkvadtreet i tillegg til dette også er avhengig av den rekkefølgen punktene settes inn i treet. Det siste kan oppfattes både som en svakhet og en styrke. Svakheten er at om man ikke innfører tiltak for å balansere punktkvadtreet, kan det bli så skjevt at ytelsen blir klart redusert. På den andre siden er denne svakheten et utslag av at punktkvadtreets totale stilengde lar seg minimalisere. Det er jo nettopp dette balanseringen har til formål.

6.5 Binære trær

Et alternativ til kvadtrær, er i noen grad binære trær. Det binære konseptet går ut på en rekursiv oppdeling av et rom i to komponenter. Vi skal her se på to datastrukturer. En binær datastruktur for 0-dimensjonale elementer og en annen binær datastruktur for 1-dimensjonale elemeter.

6.5.1 KD-trær

KD-treet har store likhetstrekk med punktkvadtreet. Forskjellen stikker i at kd-treet er et binært tre mens punktkvadtreet er et kvadtre. Figur 6.17 gir et eksempel på et kd-tre og dets oppdeling av planet. Delelinjene legges gjennom datapunktene og parallet med akseretningene. På grunn av at kd-treet er et binærtre, kan en node ha bare en delelinje. Det må derfor gjøres et valg av hvilke av de mulige akseparalleller som skal velges som delelinje. Her består en valgfrihet vi om litt skal komme tilbake til.

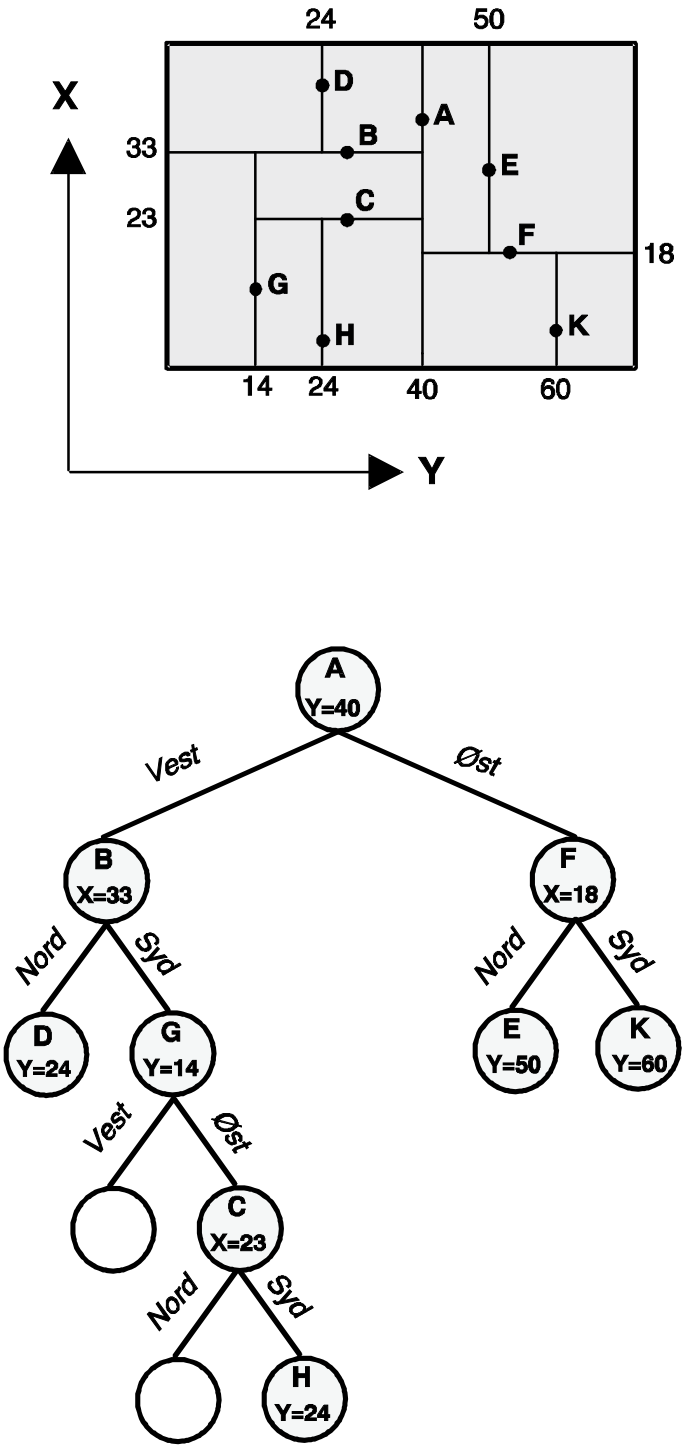
KD-treet's benevnelse kommer av eng. k-dimensional tree. Med treets dimensjon tenker man på dimensjonen til det rommet som treet deler. Punkter er 0-dimensjonale elementer, men de kan være gitt i et k-dimensjonalt rom. Se figur 6.18 som gir eksempler på todimensjonalt og tredimensjonalt kd-tre. I figuren angir X, Y og Z hvilket intervall som splittes.

Datastrukturen for et kd-tre kan defineres slik (her et 3-dimensjonalt kd-tre):

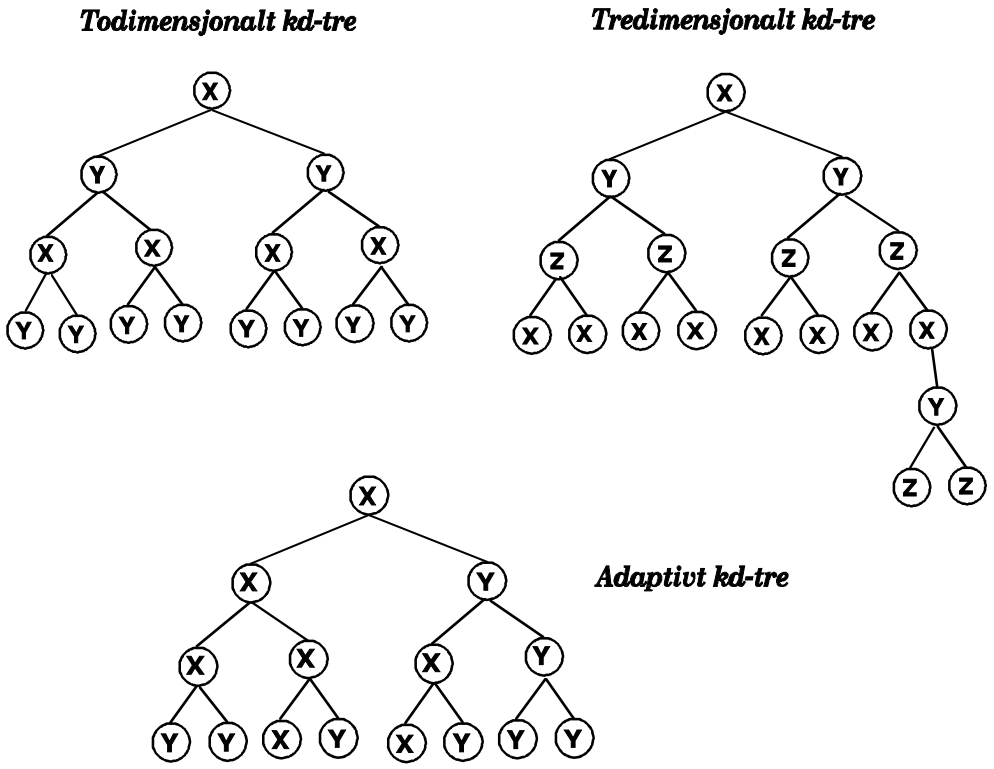
type
trepeker = ↑*trenode;*
trenode = **record** {node i et kd-tre}
venstre,høyre: trepeker;
x,y,z: integer {datapunktets koordinater}
akseretning: integer {delelinjens retning}
regulær: boolean {flagg som viser nodens status}
end*;*

På grunn av at kd-treet er et binærtre, kan en node i treet bare dele rommet i en akseretning. Dette gjør at det må finnes informasjon om hvilke koordinatintervall nodene deler. Det er to måter å innrette seg på her. Enten lagre en kode i hver node eller innføre en norm av typen: $x, y, z, x, y, z \cdots$. Roten i treet deler x-intervallet, neste nivå deler y-intervallet osv..

Innsetting og optimalisering av et kd-tre følger samme mønster som tilsvarende operasjoner på punktkvadtreet. Det samme kan også sies om algoritmer for å finne alle punkter innenfor et gitt utsnitt eller finne nærmeste punkt til et gitt punkt P. Det vises derfor til drøftingen av disse algoritmene under avsnittet om punktkvadtreet.



Figur 6.17: Et kd-tre og dets oppdeling av planet.



Figur 6.18: Varianter av kd-treet.

Adaptivt kd-tre

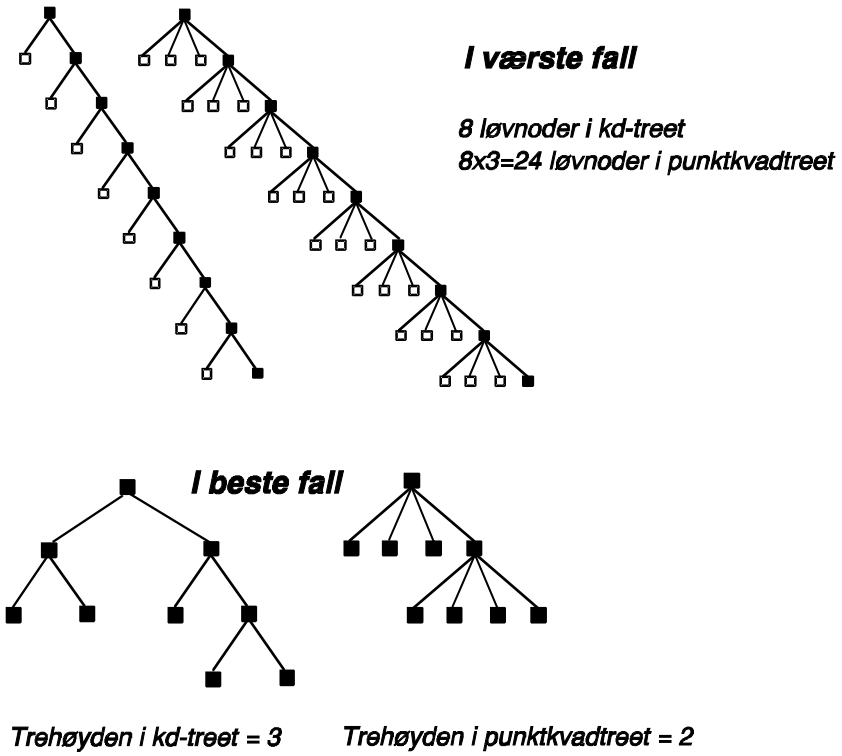
Den fleksibilitet som ligger i å velge splittaksens retning, utnyttes av de såkalte *adaptive kd-trær*. Anta at vi har en punktmengde i et 2-dimensjonalt rom og at vi innfører det vanlige koordinatsystemet. Vi stiller så spørsmålet om i hvilken akseretning delelinjen helst bør legges. Siden det er et viktig poeng at treet har liten høyde, velges den akseretning som gir en deling av punktmengden i så nær som mulig to like store delmengder.

I et adaptivt kd-tre må det i hver node lagres en kode som sier i hvilken akseretning delelinjene går. Figur 6.18 gir et eksempel på et adaptivt kd-tre. Roten deler X-intervallet, og deretter deler dens venstre og høyre etterfølger henholdsvis X- og Y-intervallet. På neste nivå deler tre av nodene sine X-intervall mens den fjerde noden deler sitt Y-intervall.

Sammenlikning mellom kd-tre og punktkvadtre

Siden kd-treet og punktkvadtreet likner mye på hverandre, er det nærliggende å spørre om hvilken av disse to datastrukturene som er best. Vi skal her nøye oss med å peke på noen åpenbare fordeler den ene strukturen har framfor den andre.

Figure 6.19 illustrerer et tilfelle der kd-treet kommer best ut og et annet tilfelle



Figur 6.19: Sammenlikning mellom kd-tre og punktkvadtre.

der punktkvadtreet kommer best ut.

Som vi ser av figuren har kd-treet færre løvnoder enn punktkvadtreet i det værste tilfellet. I det tilfellet med balanserte trær, kommer punktkvadtreet best ut på grunn av sin lavere trehøyde.

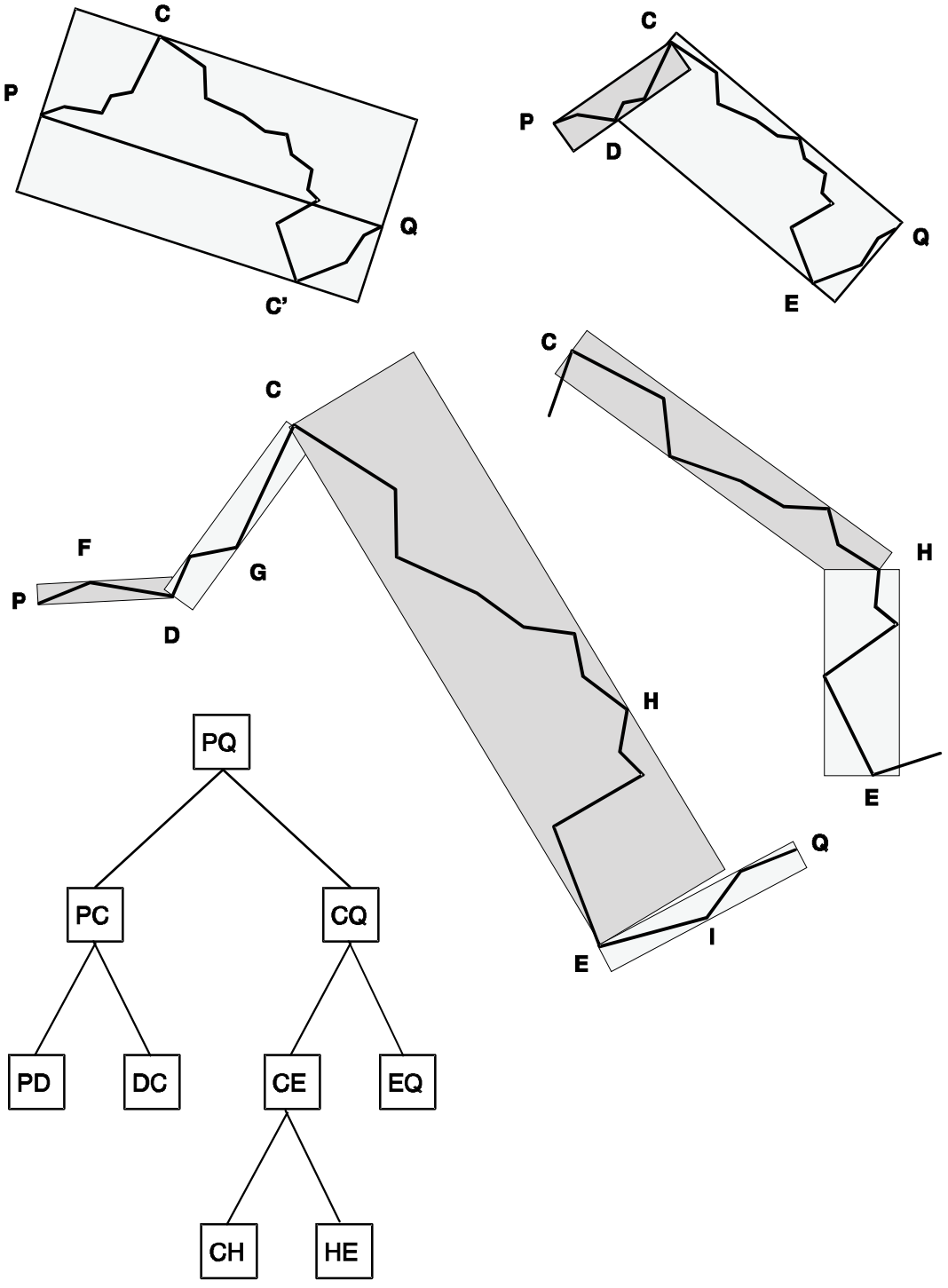
Momenter som må vektlegges i et valg mellom de nevnte datastrukturene blir derfor blant annet:

- antall løvnoder
- treets høyde

Det kan forøvrig nevnes at i instituttets TIN-modell er benyttet et optimalisert punktkvadtre.

6.5.2 Stripetre

Et stripetre (eng. *strip tree*) er en hierarkisk representasjon av 1-dimensjonale objekter. Hierarkiet framkommer ved en gradvis tilnærming av kurve-segmenter med omskrivende rektangler. Figur 6.20 viser et stripetre og dets dekomponering av en sammenhengende mengde av rette linjesegmenter (et kjede).



Figur 6.20: Et stripetre og dets oppdeling av en linje.

Stripetreet er et binært tre. Treet etableres ved først å legge et omskrivende rektangel rundt det 1-dimensjonale objektet. Rektanglet legges slik at to av dets sider blir parallelle med den rette linjen mellom objektets endepunkter. For figur 6.20 er dette punktene P og Q. De nevnte sider legges gjennom de punkter på objektet som har størst avstand fra linjen $l(P, Q)$. I figuren er dette punktene C og C'. Avstandene fra $l(P, Q)$ benevnes henholdsvis t_v og t_h.

Deretter splittes rektanglet i to komponenter ved at et punkt på objektet velges som *splittpunkt*. Som splittpunkt velges et punkt som ligger på det omskrivende rektangel. Det finnes minst ett slikt splittpunkt, punktene C og C' i figuren. Dersom det er flere, velges det punktet som har størst avstand fra den rette linjen mellom objektets endepunkter, punkt C i figuren. Deretter fastlegges det omskrivende rektangel rundt hver av de to komponentene, i figuren segmentene PC og CQ.

Slik fortsetter oppdelingen inntil t når en viss minimumsverdi. Størrelsen t fastlegger hvor godt stripetreet tilnærmer objektet. Et stripetre kan derfor oppfattes som en modell av et 1-dimensjonalt objekt. I figur 6.20 tenker vi oss en t-verdi som er slik at segmentene PD, DC og EQ ikke trenger å deles. Derimot må segment CE splittes i punktet H for å nå den antatte t-verdi.

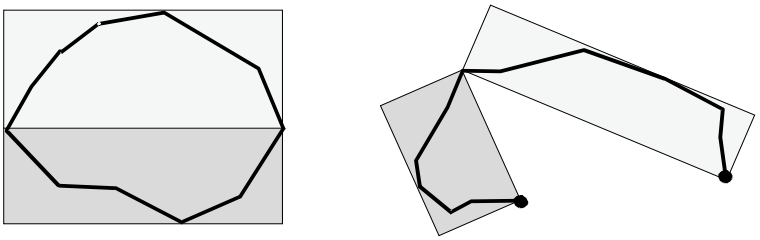
Valget av splittpunkt er avgjørende for treets høyde. Dersom vi har et kjede av linjesegmenter og vi ønsker å etablere et balansert stripetre, er det klart at splittpunktet må velges slik at det aktuelle intervallet blir delt i to like store mengder. På den andre siden ligger det et klart poeng i å innrette seg slik at de omskrivende rektangler når den fastsatte minimumsbredde så raskt som mulig. Det å velge som splittpunkt det punktet som har størst avstand fra den rette forbindelseslinje mellom det aktuelle intervallets endepunkter, har nettopp denne fordelen. Vi kan si at denne metoden velger det aktuelle intervallets mest *karakteristiske punkt* som splittpunkt. Også sett fra et kartografisk synspunkt har denne metoden klare fordeler framfor å velge det midterste punktet. Vi vil heretter forutsette at det mest karakteristiske punktet blir valgt som splittpunkt.

Datastrukturen for et stripetre kan defineres slik:

```
type
    xypeker = integer; {peker til koordinattabell}
    trepeker = ↑trenode;
    trenode = record {node i et stripetre}
        venstre,høyre: trepeker;
        p1,p2: xypeker {segmentets endepunkter}
        tv,th: integer {informasjon om stripens bredde}
    end;
```

Figur 6.20 illustrerer ikke den grad av generalitet vi vanligvis må kreve av et stripetre. Hvordan skal vi for eksempel håndtere lukkede kurver og kurver som går utenfor sine endepunkter. Se figur 6.21.

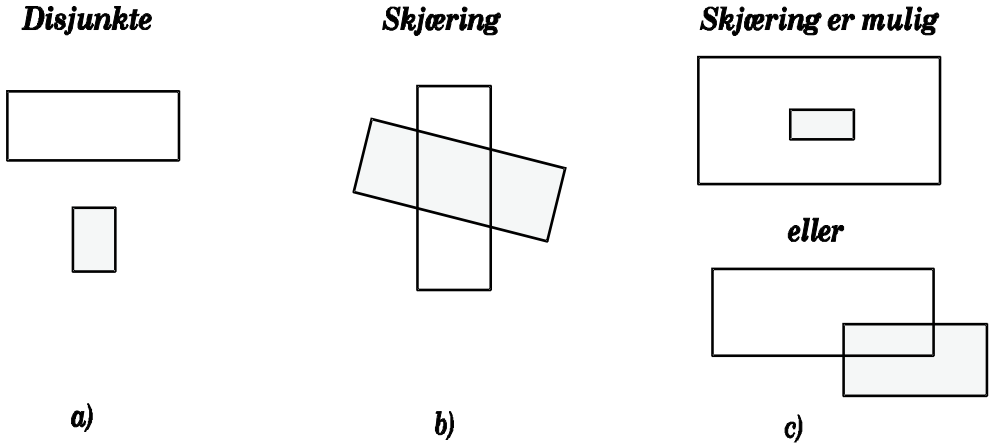
Det man her kan gjøre er å splitte det omskrivende rektangel i to rektangulære striper som vist i figuren. Deretter fortsetter oppdelingen som før. Vi kan etter dette



Figur 6.21: Stripetrerepresentasjon av lukke kurver og kurver som går utenfor sine endepunkter.

klassifisere segmenter som *regulære* eller *irregulære*. Et segment er irregulært dersom endepunktene for segmentet ikke berører endene av det omskrivende rektangel. Et flagg innføres for å indikere segmentets status.

Et stripetre har sin styrke ved romlige søk av typen: finn skjæringspunktet mellom to linjer, finn om to linjer skjærer hverandre osv.. Ved å finne snittet mellom linjenes stripetrær, kan vi finne ut om linjene skjærer hverandre eller ikke. Søket starter på toppen av trærne. Dersom rektanglene i trærnes røtter ikke overlapper hverandre, vil ikke linjene kunne skjære hverandre. Om nødvendig oppsøkes subrektanglene. Slik fortsetter søket inntil det med sikkerhet kan fastslås om linjene skjærer hverandre eller ikke.



Figur 6.22: Snitt mellom to stripetrær.

Se figur 6.22 som illustrere følgende tre situsjoner: a) linjene skjærer hverandre ikke, b) linjene skjærer hverandre og c) skjæring er mulig.

Skjæringspunktet blir selvsagt mest nøyaktig bestemt om beregningen baserer seg på løvnodene i trærne.

Som vi senere vil se har stripetreet et nært slektskap med en av de mest kjente algoritmer innenfor kartografisk generalisering, nemlig den såkalte Douglas-Peucker algoritmen. Vi vil komme tilbake til dette i kapitlet om generalisering, men vi vil

allerede her nevne at ved å ta en prefikstraversering av stripetreet og stanse rekursjonen når rektanglene når en viss minstebredde t, får vi en forenklet versjon av det 1-dimensjonale objektet. Verdien som gis t, bestemmer graden av forenklingen.

Stripetreet er invariant under rotasjoner av det underliggende rom. På dette punkt har derfor stripetreet en fordel i forhold til PM-kvadtrær. Mengden av rektangler i stripetreet kan vi oppfatte som en fleroppløselig dekomponert super-heldekkende modell av et 1-dimensjonalt objekt. På grunn av at nodene i treet også inneholder start og sluttpunkt for segmentene, kan vi også betrakte stripetreet som en fleroppløselig dekomponert normal heldekkende modell.

6.6 Flate datastrukturer

Selv om de hierarkiske datastrukturer er svært populære innen GIS, er også *flate* datastrukturer aktuelle for denne typen anvendelse.

Definisjon 40 *En flat datastruktur er en datastruktur som er ikke-hierarkisk.*

Flate datastrukturer etableres gjerne på grunnlag av en adresseberegning (hashfunksjon), men også andre prinsipper kan benyttes. Eldre kartsystemer benytter vanligvis flate datastrukturer.

6.6.1 EXCELL

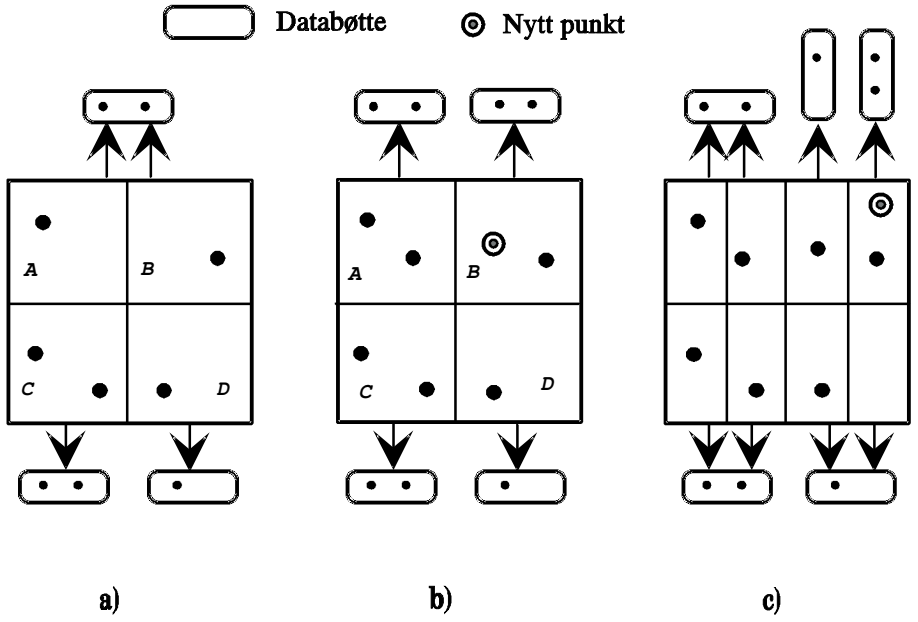
EXCELL-metoden til Tamminen [Tam81] er en forbedring av cellemetoden som vi presenterte i seksjon 2.7.5. Den kan også oppfattes som en utvidelse av extendible hashing til n-dimensjoner (jfr. seksjon 2.7.4).

Metoden kan lett tilpasses n-dimensjonale rom, men vi vil for enkelhets skyld begrense framstillingen til 2-dimensjonale rom.

Som vist i figur 6.23 baserer EXCELL seg på en regulær oppdeling av planet i rektangler og en tilordning av databøtter til rektanglene. Størrelsen på databøttene kan tilpasses karakteristiske egenskaper til datalageret, for eksempel minste adresserbare enhet til platelageret.

Rektanglene, som utgjør en *katalog* over databøttene, gis entydige nummer i intervallet $[0, n]$. Ved hjelp av en hashfunksjon avbildes alle (x, y)-par innenfor et rektangel på rektanglets unike adresse. Funksjonen har samme form som cellemetodens hashfunksjon, men med et viktig unntak. EXCELL har nemlig en strategi for å tilpasse seg endringer i datavolumet (tilsvarende strategi som i extendible hashing). Strategien består av følgende to elementer:

- Naborektangler kan legge sine data i den samme databøtten.
- Antall rektangler dobles dersom datamengden blir tilstrekkelig stor.



Figur 6.23: Innsetting med ECELL-metoden.

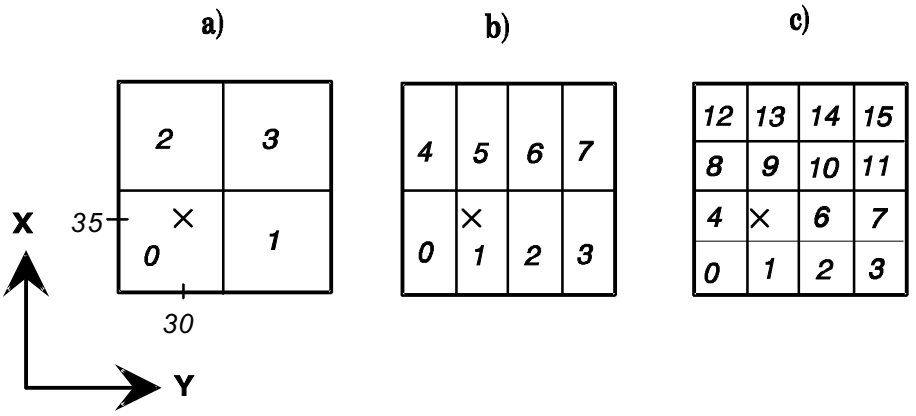
Innsettting i EXCELL illustreres i figur 6.23. Vi antar her at en databøtte kan inneholde inntil 2 datapunkter. Utgangssituasjonen i bilde a) viser at rektanglene A og B deler den samme databøtten, mens rektanglene C og D får tilordnet hver sin databøtte. Dersom C og D skulle delt databøtte, ville kapasiteten til databøtta bli overskrededet (3 punkter). Gentatte innsettinger vil en eller annen gang føre til at kapasiteten til en databøtte blir sprengt. Et slikt overløp takles ved følgende to typer splittinger:

1. *Splitt av en databøtte.* I bilde b) får rektangel B tilført et nytt datapunkt, men siden databøtten allerede inneholder 2 datapunkter, får bøtten et overløpsproblem. På grunn av at A og B deler databøtte, løses overløpsproblemet i dette tilfellet ved at A og B tilordnes hver sin databøtte.

2. *Splitt av katalogen.* I bilde c) er tilført enda et nytt datapunkt, men her nytter det ikke å løse overløpsproblemet slik som forrige gang. Denne gangen gjøres katalogen større ved at rommet deles i flere rektangler. Etter denne splittingen har katalogen kapasitet til å referere til dobbelt så mange databøtter, men dette betyr ikke at antall databøtter umiddelbart blir dobbelt så mange. EXCELL benytter seg nemlig av sin strategi om at naborektangler kan dele samme databøtten. Bildet c) viser situasjonen etter den nye delingen. Som det framgår av figuren, har alle vertikale striper i bilde c) halve bredden i forhold til de vertikale stripene i bilde b). Legg merke til at det bare er databøtten til rektangel B som er splittet og at antall bøtter er økt fra 4 til 5.

   Når et rektangel går fullt, dobles antall rektangler ved at bredden til enten

de vertikale eller de horisontale striper halveres. Dette tilsvarer doblingen av katalogens lengde i extendible hashing. Som vist i figur 6.24 alterneres det mellom horisontal og vertikal splitting. På den måten unngås lange tynne rektangler, noe som antakelig er en fordel ved romlige søk.



Figur 6.24: Prinsippet for utvidelse av katalogen i EXCELL.

Legg merke til at ved splittinger som fører til at katalogen må utvides, så er det bare datapunktene i den bøtten som splittes, det er nødvendig å flytte. I tillegg må selvsagt katalogen oppdateres, men denne forutsettes vanligvis å ligge i primærminnet slik at kostnadene ved oppdateringen ikke blir tyngende.

En av fordelene med å benytte en katalog, er at flere rektangler kan dele databøtte. Dette gir en bedre utnyttelse av bøttene enn om vi hadde sløyfet katalogen og hashet direkte til bøttene, men vel så viktig er at parametrene i hashfunskjonen kan gis andre verdier uten at databøttene må restruktureres.

Hashfunskjonen som benyttes i EXCELL kan skrives som:

$$h(x,y) = 2^{\delta y}\lfloor 2^{\delta x} \cdot x \rfloor + \lfloor 2^{\delta y} \cdot y \rfloor \tag{6.4}$$

hvor (x, y) er skalert til intervallene: $0 \leq x < 1$ og $0 \leq y < 1$. Størrelsene δx og δy er antall halveringer i de to respektive akseretninger.

Eksempel: Vi skal anvende formel 6.4 på de tre bildene a) b) og c) og beregner adressen til et punkt (x, y) med koordinatene $(0.35, 0.30)$. I figuren er punktet markert med et kryss.

Bilde a): $\delta x = 1$ og $\delta y = 1$

$$h(x,y) = 2^1\lfloor 2^1 \cdot 0.35 \rfloor + \lfloor 2^1 \cdot 0.30 \rfloor = 2\lfloor 0.7 \rfloor + \lfloor 0.6 \rfloor = 0 + 0 = 0$$

Bilde b): $\delta x = 1$ og $\delta y = 2$

$$h(x,y) = 2^2\lfloor 2^1 \cdot 0.35 \rfloor + \lfloor 2^2 \cdot 0.30 \rfloor = 4\lfloor 0.7 \rfloor + \lfloor 1.2 \rfloor = 0 + 1 = 1$$

Bilde c): $\delta x = 2$ og $\delta y = 2$

$$h(x,y) = 2^2\lfloor 2^2 \cdot 0.35 \rfloor + \lfloor 2^2 \cdot 0.30 \rfloor = 4\lfloor 1.4 \rfloor + \lfloor 1.2 \rfloor = 4 + 1 = 5$$

Datastrukturen for EXCELL blir enkel, siden katalogen kan realiseres som et en-dimensjoanlt array over pekere til databøttene.

```
type
    EXCELL=record
        katalog: array [ 0..Nmax ] of bøttepeker;
        n: integer; {katalogens aktuelle lengde}
        dx,dy: integer {antall splittinger}
    end;
```

På grunn av at katalogen ikke har en statisk lengde, er det fordelaktig å benytte dynamisk plassreservasjon. I standard Pascal har man desverre ikke lov til å endre lengden av et array under programutførelsen. C har denne muligheten.

På grunn av at EXCELL benytter et regulært rutenett, er antall splittinger av katalogen svært følsom overfor datapunktenes romlige fordeling.

Siden EXCELL skal tilpasse seg endringer datavolumet, bør man definere de inverse operasjoner til de to typer splittinger. Av disse er sammenslåing av databøtter mest aktuelt. Sammenslåing av innganger i katalogen (redusere δx eller δy med 1) er mindre aktuelt på grunn av at muligheten for en påfølgende splitt kan være stor. Man må unngå en ustabil situasjon hvor sletting og tilførsel av objekter fører til stadige sammenslåinger og slettinger. Vi går ikke nærmere inn på dette her.

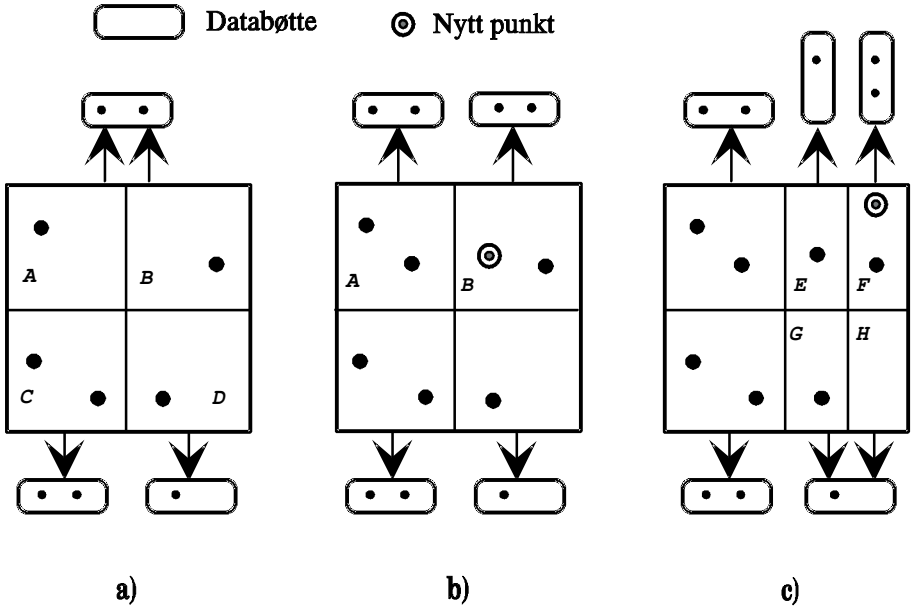
EXCELL er en metode som kan anvendes for n-dimensjonale geografiske objekter. For eksempel benytter Tamminen [Tam81] EXCELL til å lagre punkter, linjer og flater. I tilfellet med linjer benyttes en framgangsmåte som likner på den vi beskrev under PM-kvadtrær.

6.6.2 Gridfile

Selv om *gridfile* er et metode som kan anvendes på n-dimensjonale rom, vil vi delvis begrense den etterfølgende forklaringen til et 2-dimensjonalt rom.

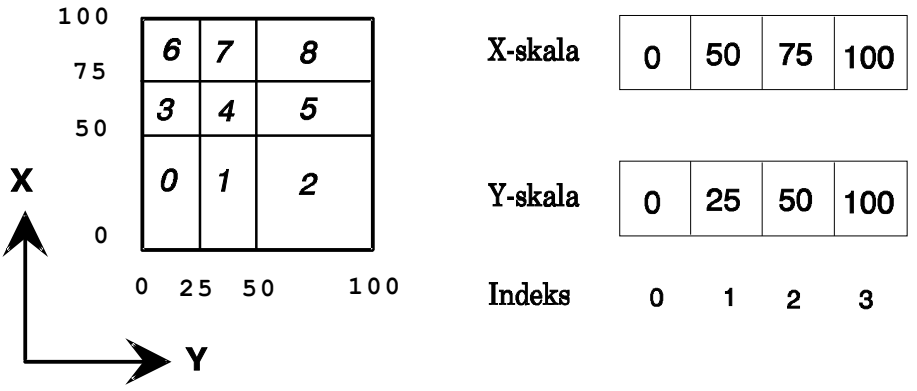
Gridfile, som er lansert av Hinrichs [Hin85], har store likhetstrekk med EXCELL, men det er en vesentlig forskjell som vises i figur 6.25. Som det framgår av bilde c) har et overløp i rektangel B ført til en splitt av rektanglene B og D. I EXCELL ble i tillegg også rektanglene A og C splittet. Dette gjør at gridfile har en langt større evne til å tilpasse seg objektenes romlige fordeling enn det EXCELL har. Prisen man må betale for denne fleksibiliteten, er at den enkle adresseberegningen i EXCELL må oppgis.

Når det oppstår overløp i et av katalogens rektangler, deles den horisontale eller vertikale stripen som rektanglet er en del av. Delingen er gjennomgående for hele stripen, men merk at det kun er den stripen som har et overløpsproblem som blir



Figur 6.25: Innsetting med gridfile-metoden.

delt. En slik oppdeling av rommet gjør at adresseberegningen i gridfile trenger mere informasjon enn adresseberegningen i EXCELL. Gridfile benytter til dette formål en *skala* i hver av akseretningene.



Figur 6.26: Skalaer i gridfile-metoden.

Skalaene, som gir koordinatene til delelinjene, realiseres som en-dimensjonale array. Figur 6.26 viser en oppdeling av planet med gridfile og de tilhørende skalaer. Skalaene forutsettes å ligge i maskinens primærminne. Datastrukturen for gridfile kan etter dette defineres slik:

type
skala=**record** {skala}

```
        s: array [ 0..Nmax ] of koordinat;
        n: integer {skalaens aktuelle lengde}
    end;
    katalog=record {katalog}
        k: array [ 0..Nmax ] of bøttepeker;
        n: integer {katalogens aktuelle lengde}
    end;
    gridfile=record {Gridfile}
        gridkatalog: katalog;
        xskala,yskala: skala
    end;
```

For å finne kataloginngangen til et vilkårlig datapunkt, finnes først indeksen til grenselinjene for de aktuelle intervall. Deretter brukes disse indeksene til å beregne nummeret til vedkommende rektangel. La oss anta at punktet (x, y) ligger i intervallene $x(i) \leq x < x(i+1)$ og $y(j) \leq y < y(j+1)$. Nummeret til vedkommende rektangel kan da beregnes av:

$$h(x, y) = (n_y - 1)(i - 1) + (j - 1) \tag{6.5}$$

hvor n_y er den aktuelle lengden av Y-skalaen, $0 \leq i \leq n_x$ og $0 \leq j \leq n_y$.

Eksempel: Vi skal beregne inngangen i katalogen for punktet $(x, y) = (80, 30)$ i figur 6.26. Av figuren finner vi at $i = 3$, $j = 2$ og $n_y = 4$. Dette gir:
$h(x, y) = (4 - 1)(3 - 1) + (j - 1) = 6 + 1 = 7$

Gridfile har god evne til å tilpasse seg både dataenes romlige fordeling og endringer i datavolumet. På samme måte som EXCELL har også gridfile to typer overløp, nemlig når en databøtte som deles av flere kataloginnganger går full, og når en inngang i katalogen går full. Ved splitting av en inngang i katalogen, er det vanlig å halvere det tilhørende koordinatintervallet. Begrunnelsen for dette er at man antar at objektene har en jevn romlig utbredelse. Dersom dette holder stikk, vil halvering gi det beste resultatet i det lange løp.

Den inverse funksjonen til splitting er sammenslåing. Det er to situasjoner som kan gjøre sammenslåing hensiktsmessig: sammenslåing av databøtter og sammenslåing av innganger i katalogen. Av de samme grunner som angitt for EXCELL, er sammenslåing av kataloginnganger mindre aktuelt.

Gridfile stiller visse krav til hvilke innganger i katalogen som kan få dele en databøtte. Kravet er at deres rektangler skal være direkte naboer og at unionen av rektanglene skal være et rektangulært område som skal kunne framkomme etter et visst antall splittinger av hele området. Det vises til [Hin85] for nærmere detaljer og diskusjon av visse vranglåssituasjoner (eng. deadlock).

Gridfile har en viss hierarkisk struktur, i det katalogen har to nivåer. Et argument imot en slik struktur er at det ikke gis noen overbevisende begrunnelse for

hvorfor man skal stoppe hierarkiet etter to nivåer. Vi går ikke nærmere inn på denne diskusjonen her, men vil oppfatte gridfile som en flat datastruktur.

6.6.3 Base85

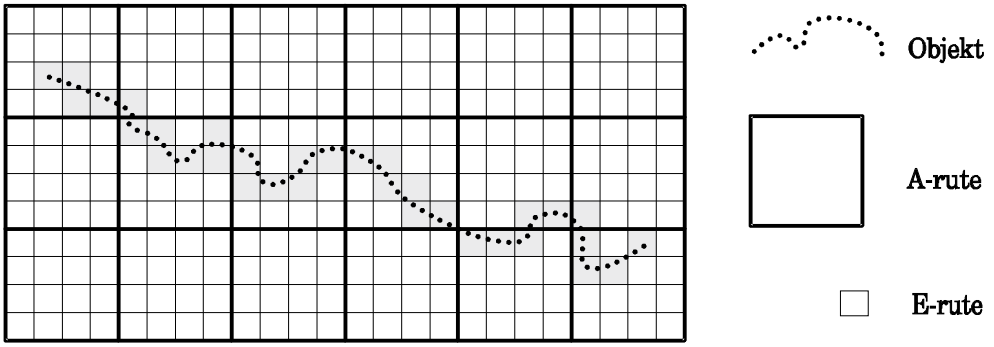
I det norske kartmiljøet ble det i begynnelsen av 1980-årene tatt initiativ til å utvikle en ny generasjon geodatabaser. Et resultat av dette initiativet er Base85. Det finnes tre rapporter fra Norsk Regnesentral om Base85 samt en sammenfattende diskusjon av Einbu i [Ein87]. Base85 er designet for å kunne håndtere 0-dimensjonale og 1-dimensjonale elementer samt omrissmodeller av 2-dimensjonale elementer.

Base85 har store likhetstrekk med EXCELL, men den adskiller seg fra EXCELL på følgende punkter:

1. Dataene legges i tre tabeller (*fil-I, fil-II og fil-III*) og ikke i databøtter slik som i EXCELL.

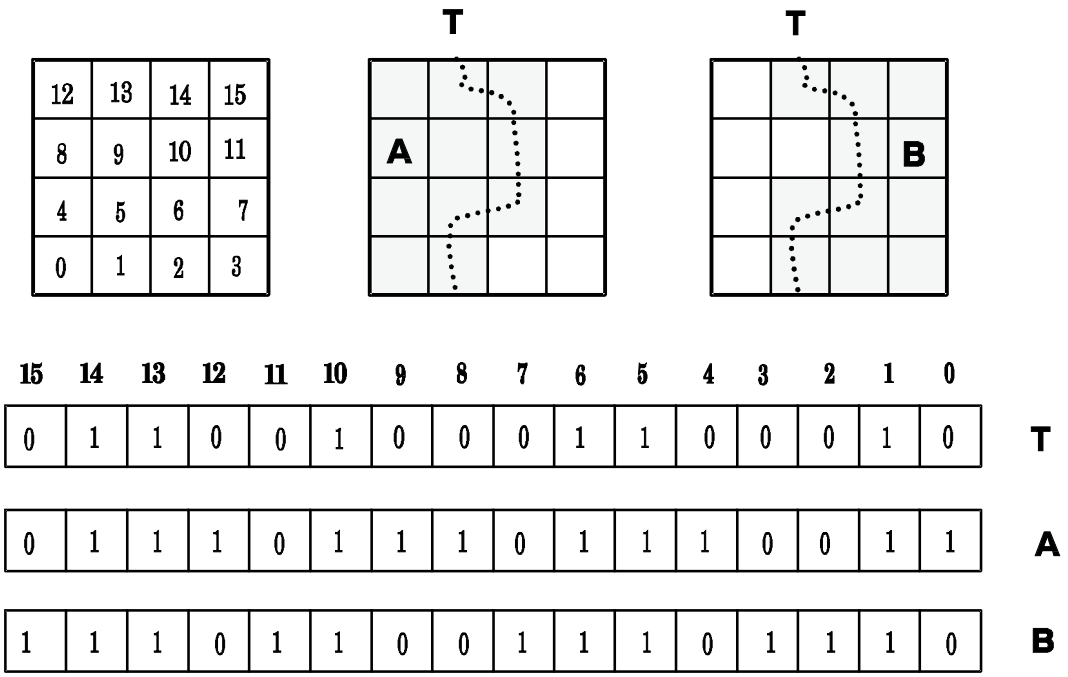
2. Det er ikke angitt en veldefinert strategi for en dynamisk tilpasning av adressefunksjonen (hashfunksjonen) til dataenes tetthet og romlige fordeling.

Adresseberegningen baserer seg på en funksjon av typen vi finner i seksjon 2.7.5. De kvadratiske ruter som her blir definert, kalles **A**-ruter. I tillegg til A-rutene defineres et system av **E**-ruter som framkommer ved å dele A-rutene i 16 subruter. Selv om Base85 her har et visst hierarki, vil vi likevel definere den som en flat datastruktur. Figure 6.27 gir et eksempel på A-ruter og E-ruter i Base85.



Figur 6.27: Søkeapparatet i Base85.

Hver E-rute får tilordnet ett bit. Følgelig vil 16 E-ruter kunne assosieres med et maskinord på 16 biter. Figur 6.28 illustrerer hvordan bitmønsteret framkommer. Her er objekt T en linje som krysser følgende E-ruter: 13,14,10,6,5 og 1. I bitstrengen representeres disse rutene med tilstand 1. Dette gir oss derfor bitstrengen 0110010001100010. Objektene A og B er flater og deres bitmønster finnes på tilsvarende måte som for T. Vi ser at ved å ta snittet mellom A og B, blir resultatet grenselinjen T.



Figur 6.28: E-ruter i Base85.

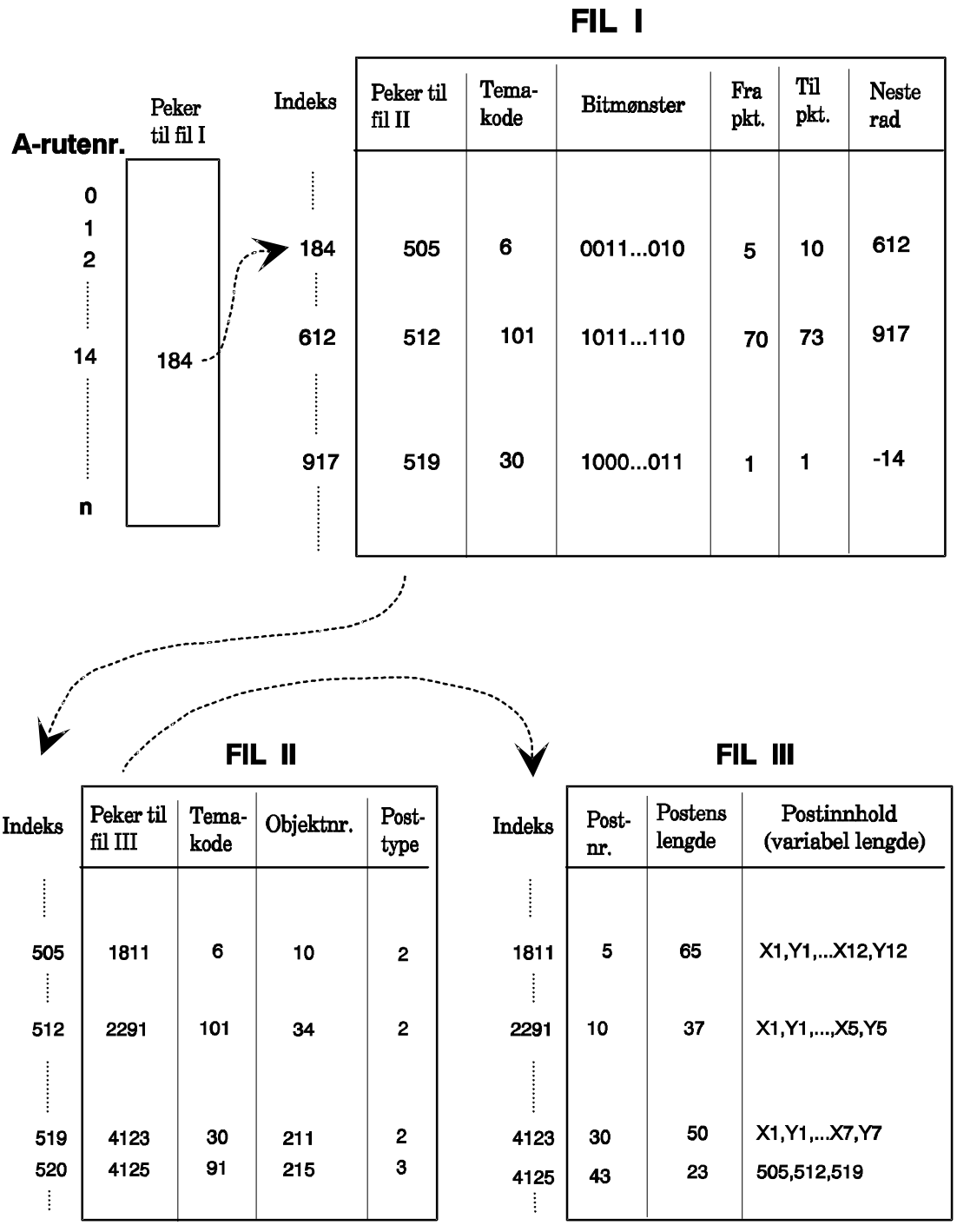
E-rutene i Base85 representerer noe nytt i forhold til EXCELL. De krever liten lagerplass og er et godt hjelpemiddel ved romlige søk innenfor en A-rute.

Base85 benytter seg av de tre tabellene fil I, II og III. Her er fil I systemets søkeapparat mens filene II og III inneholder beskrivelser av de geografiske objekter. Figur 6.29 viser hvordan disse tabellene er bygget opp. Eksemplet i figuren starter med at vi går inn i A-rutetabellen med adressen 14 (beregnet på grunnlag av adressefunksjonen). Her finner vi tallet 184 som gir en inngang i fil I. Under denne inngangen finner vi følgende data: en peker til fil II (505), objektets temakode (6), bitmønsteret, fra pkt., til pkt. og neste rad (612). Neste rad referer til neste objekt innenfor vedkommende A-rute. Kolonnene fra pkt. og til pkt. angir hvilke deler av en linje som ligger innenfor A-ruta. *Neste rad* peker til inngang 612 og her finner vi et objekt med temakode 101. Kjeden fortsetter så til inngang 917. Her finner vi et objekt med temakode 30, men pekeren til neste rad er denne gangen et negativt tall. Dette betyr at vi er kommet til slutten av listen.

Fil I kan oppfattes som en mengde av lister, en liste for hver A-rute. Lengden av listene er selvsagt en kritisk faktor. Dessuten dersom fil I ligger på et sekundærlager, er det av stor betydning at hele objektlisten som tilhører en A-rute, kan hentes fram ved et minimum av diskaksesser. På dette punktet kan konseptet i Base85 forbedres ved følgende tiltak:

1. Det innføres en metode for å splitte listene når de blir lange nok.

2. Det etableres en strategi for at lister kan dele samme minste adresserbare enhet



Figur 6.29: Fil I,II og III i Base85.

på sekundærlageret (dele databøtte).

Disse forbedringene kan gjennomføres ved å tilpasse de metoder som benyttes i EXCELL, gridfile eller PM-kvadtrær.

Koordinater til objektene finner vi i fil III. På grunn av at flere objekter kan ha felles grenselinje, har Base85 mulighet til å danne *sammensatte objekter*. For eksempel finner vi at objekt 215 (fil II) har posttype 3, som betyr at vi har med et sammensatt objekt å gjøre. Referansen til fil III gir oss postinnhold 505,512 og 519. Disse tallene refererer til innganger i fil II og vi finner at objekt 43 består av objektene 10, 34 og 211. Fordelen med en slik strategi er at koordinater til felles grenselinjer lagres bare en gang.

Et spørsmål er hvor store A-ruter det er lønnsomt å ha. Svaret er ikke entydig på grunn av at det både er avhengig av objektenes romlige fordeling, deres antall og hvilke typer romlige søk som legges til grunn.

I det tilfellet at man spør etter nærmeste objekt til et en angitt posisjon, kan det synes som om det er en fordel med så små A-ruter som mulig. Problemet er at dersom mange av A-rutene er tomme, blir det mange tomme ruter å søke igjennom før objektet er funnet.

Dersom spørsmålet er: finn alle objekter innenfor et gitt vindu", er det klart at det er en fordel at søkeområdet berører så få A-ruter som mulig. Dette viser at det kan være motstridende interesser mellom ulike typer romlige søk. Forøvrig husker vi fra R-trær og MX-CIF trær at de opererte med omskrivende rektangler til objektene. For den type søk vi her snakker om, er det en fordel å kjenne objektenes omskrivende rektangler.

6.7 Ojektorienterte indekserings-metoder

For å kunne gjøre raske oppslag i en geodatabase, er man avhengig av at det geografiske rom er oppdelt på en eller annen måte. Oppdelingen kan gjøres etter et regelmessig mønster slik som for eksempel i PM-kvadtrær, Excell og Gridfile, men det er også aktuelt å innføre objektorienterte restriksjoner på oppdelingen.

Eksempel: I et kart over eiendomsforhold, administrative og politiske enheter, er det naturlig å definere et hierarki av objekter som: krets, kommune, fylke, land, overnasjonale enheter. En objektorientert oppdeling av rommet kan her være å lage et system av omskrivende rektangler slik at hver krets får sitt rektangel, hver kommune får sitt rektangel osv.. Objektene i dette tilfellet er de nevnte enheter.

Eksempel: Innenfor kartlegging har man lange tradisjoner med oppdeling av rommet i kartblad. Vanligvis er utstrekningen av de enkelte kartblad fastsatt i henhold til en regelmessig oppdelingen av rommet. Dette skaper visse vanskeligheter for brukere som opererer i skjøten mellom kartblad. For enkelte populære utfartsområder lages det kartutsnitt slik at hele området blir vist på ett enkelt kart. Objektet i dette tilfellet er utfartsområde.

Ved etablering av felles geodatabaser, er det ikke alltid like enkelt å definere objekter og hierarkier av objekter. Vi skal ikke gå inn i denne diskusjonen her, men nøye oss med å presentere noen datastrukturer som kan håndtere hierarkier av rektangler. Det er opp til den aktuelle implementasjon å definere hvilken informasjon rektanglene skal inneholde og i hvilken grad objektorienterte restriksjoner skal legges til grunn for definisjon av rektanglene.

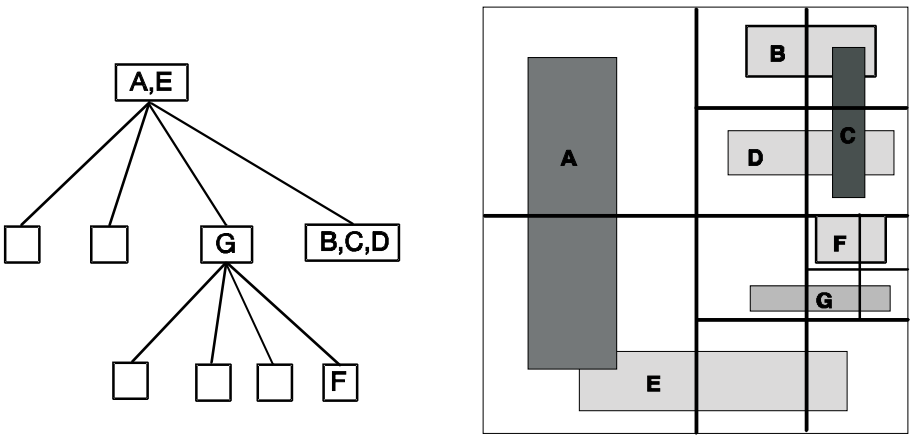
6.7.1 MX-CIFkvadtre

MX-CIFkvadtreet (CIF står for Caltech Intermediate Form) er en datastruktur som opererer på en mengde av rektangler. Det finnes flere varianter av MX-CIFkvadtreet, men vi skal her nøye oss med å beskrive grunnformen.

Definisjon 41 *Et* MX-CIFkvadtre *assosierer hvert rektangel R med den minste kvadblokk som inneholder eller dekker R.*

Rektangler kan assosieres med såvel interne noder som løvnoder, men med den restriksjon at et rektangel ikke kan assosieres med mere enn en node. Oppdelingen av rommet pågår inntil en kvadblokk ikke inneholder noen rektangler eller kvadblokken oppnår en minstestørrelse. Minstestørrelsen kan velges ut fra kunnskap om hvor små objekter databasen inneholder.

MX-CIFkvadtreet skiller seg fra MX-kvadtreet ved at MX-CIFkvadtreet kan lagre objekter i både interne noder og løvnoder. Likheten stikker i at begge assosierer sine objekter med den minste kvadblokk som dekker objektet. For MX-kvadtreet fører dette til at punktene blir assosiert med løvnodene i treet, mens for MX-CIFkvadtreet er konsekvensen at rektangler kan assosieres med såvel interne noder som løvnoder.



Figur 6.30: Et MX-CIFkvadtre og dets rektangler.

Figur 6.30 viser et MX-CIFkvadtre og dets rektangler. I utgangspunktet legges et omskrivende rektangel rundt hvert objekt. Disse rektanglene kan overlappe hverandre.

De topologisk-romlige relasjoner vi tillater mellom et rektangel og dets node i kvadtreet er r_6 (nodens kvadblokk inneholder rektanglet) eller r_7 (nodens kvadblokk dekker rektanglet). I figur 6.30 er for eksempel rotblokken den minste kvadblokk som inneholder eller dekker rektanglene A og E. De topologisk-romlige relasjoner er i dette tilfellet r_6 mens den topologisk-romlige relasjonen for rektangel F er r_7. Rektangler som oppfyller de nevnte kravene, ligger slik til at kvadblokkens delelinjer krysser rektanglene (merk at å gjøre slutningen den andre veien ikke holder).

Innsetting av et rektangel starter med et søk fra treets rot og nedover i treet inntil den minste kvadblokk som inneholder eller dekker rektanglet, er funnet. Rektanglet settes så inn i vedkommende node. Et rektangel assosieres kun med en node i treet.

Innsetting, sletting og søk i MX-CIFkvadtreet er relativt rett fram. Tilsvarende som for MX-kvadtreet er treets fasong uavhengig av den rekkefølgen rektanglene settes inn i treet, skjønt under innsettingen vil de midlertidige trær være avhengig av rektanglenes rekkefølge.

Det værste tilfellet både når det gjelder lagerplass og tidsyteløse, har vi når samtlige rektangler må settes inn på laveste nivå i treet, d.v.s. når ingen rektangler blir satt inn i de interne noder i treet.

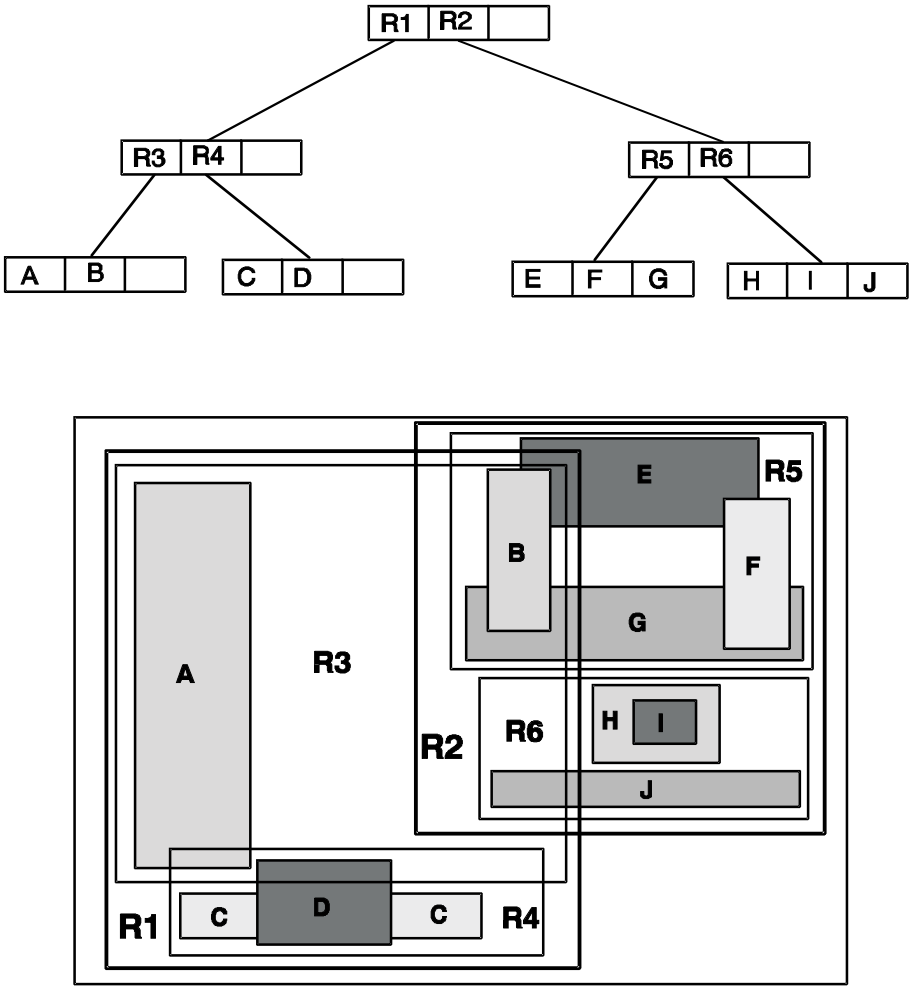
På grunn av at et stort antall rektangler kan bli lagret i en node, er det aktuelt å etablere en datastruktur innenfor nodene. En måte å gjøre dette på, er å sortere rektanglene etter deres tyngdepunktskoordinater.

6.7.2 R-trær

R-treet [Gut84] er en hierarkisk datastruktur som er avledet fra B-treet. Benevnelsen kommer av *rectangle-tree*. Vilkårlige geometriske objekter representeres i R-treet som n-dimensjonale rektangler. En node i treet svarer til det minste n-dimensjonale rektangel som inneholder dets etterfølgere. De interne noder har pekere til sine etterfølgere, mens løvnodene i stedet har pekere til de aktuelle geometriske objekter i databasen. Et objekt blir representert i treet ved det minste rektangel det er en delmengde av.

Ofte dimensjoneres løvnodene i forhold til platelagerets minste adresserbare enhet.

Figure 6.31 viser et R-tre og dets hierarki av rektangler. Generelt tillates alle de ni binære topologisk-romlige relasjoner mellom rektanglene, men vi kommer tilbake til varianter av R-trær som her setter visse begrensninger. I figuren tilhører for eksempel rektanglene R1 og R2 samme node, men de får likevel lov til å overlappe hverandre. Likeså ser vi at rektangler som tilhører ulike noder også kan overlappe hverandre, men de kan bare assosieres med en av nodene. For eksempel ligger rektangel B innenfor både R3 og R5, men B er bare koblet til en av nodene. Dette betyr at et



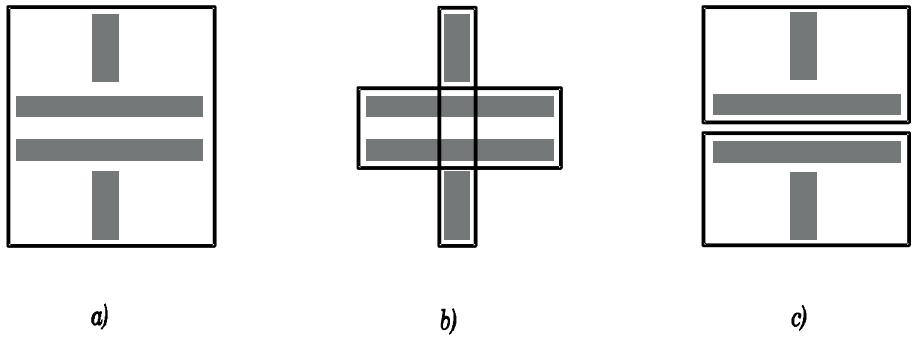
Figur 6.31: Et R-tre og dets hierarki av rektangler.

romlig søk ofte må besøke mange rektangler før det kan avgjøres om et objekt finnes eller ikke. I eksemplet må derfor både R3 og R5 besøkes.

Basisreglene for å forme et R-tre tilsvarer reglene for et B-tre. Alle løvnoder skal ligge på samme nivå. Hver inngang i en løvnode er et 2-tuppel av formen (R, O), hvor R er det minste rektangel som omslutter dataobjekt O. Hver inngang i en intern node er et 2-tuppel av formen (R, p), hvor R er det minste rektangel som omslutter rektanglene i etterfølgeren som pekes på av p.

Legg merke til at fasongen til et R-tre ikke er unik. Ut fra en gitt mengde objekter finnes det en rekke mulige konfigurasjoner av R-treet. Den rekkefølge objektene settes inn i treet, spiller en avgjørende rolle for treets konfigurasjon.

Algoritmen for å sette et objekt inn i R-treet er analog til innsetting i et B-tre. Den aktuelle løvnoden finnes ved å traversere R-treet fra roten, og for hver node velge det subtreet som kan omslutte objektets omskrivende rektangel. Så snart løvnoden er funnet, undersøkes om innsetting av det nye rektanglet vil gjøre at noden går full. Dersom så blir tilfellet, må noden splittes, og de $M + 1$ rektangler må fordeles på de to nodene. Splittingen forplanter seg oppover i treet på tilsvarende måte som i et B-tre.



Figur 6.32: Prinsipper for å splitte en node i R-treet.

Det er mange måter å splitte en node på. Figur 6.32 illustrere to prinsipper for splitting. Rektangel a) blir splittet i de to variantene b) og c). I b) minimaliseres det totale arealet til de omskrivende rektangler, mens i c) er det totale overlappingsområdet mellom rektanglene minimalisert. De to metodene har ulike fordeler ved romlige søk. Metode b) har som formål å minimalisere sannsynligheten for at nodene blir besøkt (minst mulig totalareal til de omskrivende rektangler), mens metode c) har som formål å minimalisere sannsynligheten for at begge nodene må besøkes (minst mulig overlapp).

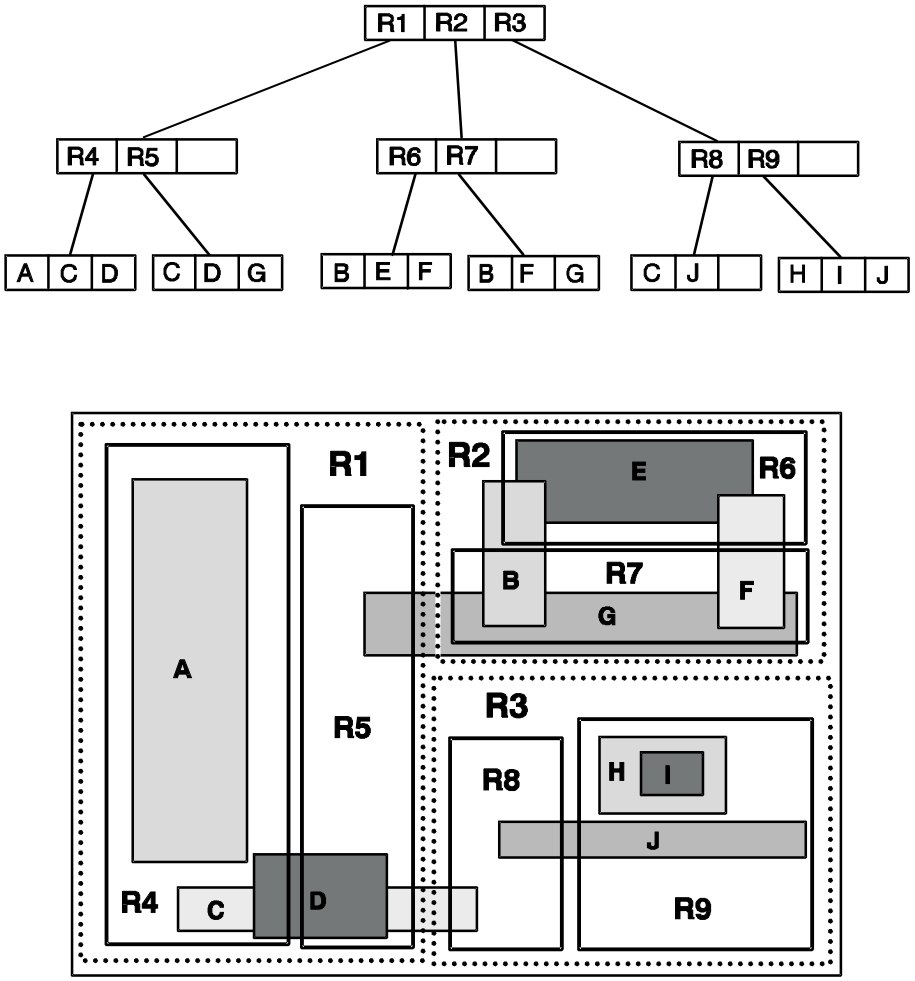
Kostnadene ved å finne de splittinger som tilfredsstiller minimaliseringskravene, har en eksponentiell vekstrate med omsyn til antall rektangler i den noden som skal splittes (alle kombinasjoner testes). Forsøk har vist at selv om man treffer noe på siden av den ønskede splittingen, vil ytelsen til R-treet ikke bli vesentlig redusert. Dette gjør at en lineær splitt-algoritme er funnet fordelaktig selv om splittingen blir

noe forringet i forhold til den beste splittingen.

Sletting av rektangler i et R-tre adskiller seg noe fra sletting i et B-tre i det tilfellet at en node blir *underfylt*. I stedet for å slå sammen nabonoder, foretas en reinnsetting. Vi går ikke nærmere inn på dette her, men viser til [Gut84].

Romlige søk i et R-tre er rett fram. Det eneste problemet er muligheten for at et stort antall rektangler må besøkes. Grunnen er at søkeområdet kan ligge i overlappingsområdet mellom flere rektangler, mens de ønskede objekter bare finnes i en av løvnodene. For eksempel i figur 6.31 ligger objekt B innenfor både rektangel R3 og rektangel R5, men B er assosiert kun med rektangel R3. Følgelig må både R3 og R5 besøkes for å finne de objekter som ligger helt eller delvis inenfor overlappingsområdet mellom R3 og R5.

For å bøte på denne svakheten, kan benyttes modifikasjoner av R-treet. Et alternativ er det såkalte R^+-tre. Figur 6.33 viser et R^+-tre og dets hierarki av rektangler.



Figur 6.33: Et R^+-tre og dets hierarki av rektangler.

Forskjellen mellom et R-tre og et R^+-tre er at R^+-treet forlanger at de omskri-

vende rektangler til de *interne* noder ikke skal overlappe hverandre. Derfor assosieres objektene med alle de omskrivende rektangler de snitter. Dette fører til at trehøyden vil øke, men romlige søk vil likevel effektiviseres. I figur 6.33 finnes det for eksempel en sti til objekt G både via rektangel R5 og rektangel R7.

R^+-treet har den bakdel at B-treytelse (min. halvfulle databøtter) ikke kan garanteres uten kompliserte innsettings og slettingsrutiner. Likevel viser empiriske tester at R^+-treet har god ytelse i forhold til det vanlige R-treet.

I stedet for å benytte omskrivende rektangler er mere fleksible geometriske former foreslått. For eksempel er celle-treet til Günther [GÖ89] en utvidelse av R^+-treet ved at de interne noder i treet er konvekse n-dimensjonale polyeder i stedet for n-dimensjonale rektangler.

6.8 Konklusjon

Vi har i dette kapitlet presentert en rekke datastrukturer og man kan spørre seg: "hvilken datastruktur er best?". Det kan ikke gis ett enkelt svar på spørsmålet, fordi en rekke faktorer vil spille inn. Faktorer som her er av betydning er:

1. Hvilke type romlige søk skal datastrukturen understøtte?
2. Skal datastrukturen understøtte kartografisk zoom?
3. Hvor store datamengder dreier det seg om?
4. Får hele datavolumet plass i primærminnet eller må storparten av dataene ligge på sekundære lager?
5. Hvilken romlig fordeling har dataene?
6. Hvilken topologisk dimensjon har dataene?
7. Hvilke typer geometriske modeller skal benyttes (omrissmodell, heldekkende modell)?
8. Hvor mye topologisk informasjon skal datastrukturen håndtere?
9. Hvor stor lagerplass opptar datastrukturen?
10. Hvilke krav stilles til datastrukturens tidsytelse?
11. Hvor store er utviklingskostnadene ved å implementere datastrukturen?

Vi har i dette kapitlet i liten grad gått inn på at geografisk informasjon er en meget kompleks datatype. Geografiske data består som kjent av romlige data (x, y, z, t) og semantiske data (kvalitative og kvantitative) samt topologiske relasjoner mellom de geografiske objekter. Ved å kombinere romlige og semantiske kriterier, kan man

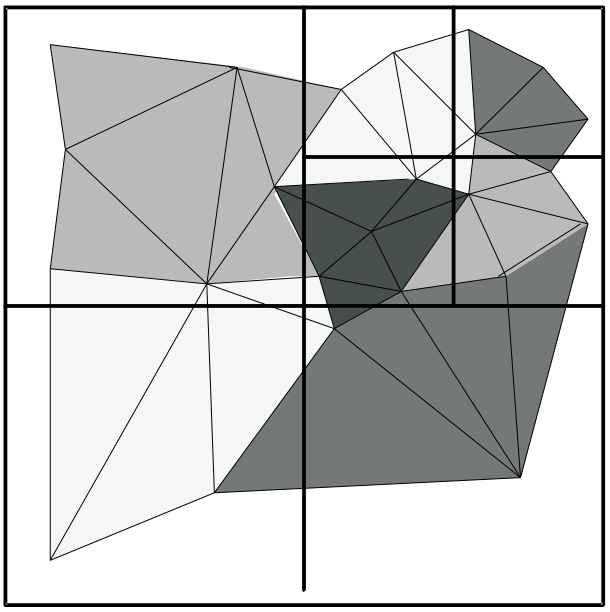
få meget komplekse geografiske søk. Dette vil bli diskutert i større bredde i senere kapitler om geografiske databaser.

Vi skal med noen eksempler utdype anvendelsen av romlige datastrukturer innen GIS.

Eksempel: La oss si at vi skal etablere et TIN (T=triangulert, I=irregulært, N=nettverk, altså en trekantmodell) over Norge. Vi antar at det dreier som meget store datamengder. Problemet består i å dele opp datamengden i porsjoner slik at hver porsjon får en behagelig størrelse. Oppdelingen kan følge prinsippene til EXCELL, gridfile eller kvadtrær. Jeg vil foreslå en strategi som baserer seg på en kvadtrestruktur og at det gis en øvre grense for antall trekanter en kvadblokk kan inneholde. Dersom denne grensen nås, splittes kvadblokken. Dette gir oss en dynamisk blokkoppdeling som tilpasser seg trianglenes tetthet.

Vi velger et lineært kvadtre og bestemmer posisjonskoden ved en bitfletting og en nivåangivelse.

Triangler som krysser en blokkgrense, legges til en av blokkene etter en eller annen regel; for eksempel den av blokkene som har færrest trekanter. Figur 6.34 viser kvadtreblokker som er lagt over et triangulert område. I eksemplet er splittfaktoren valgt lik 6.



Figur 6.34: Eksempel på en indekseringsmetode for et TIN.

Nummeret til trekantene settes sammen av blokkas nummer pluss et lokalt nummer innenfor blokka.

En variant av den strategien som er beskrevet ovenfor, går ut på å fjerne det båndet av triangler som krysser blokkgrensene. Dette båndet kan nemlig lett rekon-

strueres ut fra par av naboblokker. Fordelen med denne strategien er at det oppnås større uavhegighet mellom blokkene som følge av det ikke finnes referanser fra triangler i en blokk til triangler i en annen blokk. Dette er forøvrig detaljer som vil bli nærmere diskutert i kompendiet om digitale terrengmodeller [Bjø89].

Kontrollspørsmål

1. Nevn noen datastrukturer du mener kan understøtte kartografisk zoom?
2. Hvorfor er hierarkiske datastrukturer flinke til å fokusere på delområder i en mengde?
3. Gi noen eksempler på anvendelser der du mener en flat datastruktur er velegnet?
4. Anta at polygonpunkter og triangelpunkter i en kommune skal legges inn i et dataregister. Hvilken datastruktur vil du velge dersom systemet skal plukke ut punkter på grunnlag av koordinater og punktnummer.
5. Vi skal beregne skjæringspunkter mellom polygoner. Hvilken datastruktur kan velges?
6. Hvilken datastruktur vil du velge for et kart som foreligger på rasterform?
7. Vi har en global database og spør: tegn ut alle elver som krysser grensen mellom Norge og Sverige. Foreslå en datastruktur.

Kapittel 7

GENERALISERING

Dette kapitlet er bare i sin første begynnelse side, derfor den skissemessige og ufullstendige framstillingen av stoffet.

Det forutsettes at leseren er kjent med den grunnleggende teori for kartografisk kommunikasjon og kartografisk generalisering, herunder informasjonsteori. Det vises til [Bjø87].

Det vil her bli lagt vekt på å presentere metoder for hvordan generaliseringen kan utføres.

Metodene som presenteres, er tilpasset problemstillinger i 2-dimensjonale rom. Konseptene kan mer eller mindre opplagt utvides til 3-dimensjoner. Dette vil delvis bli diskutert i et senere kapittel og i neste versjon av kompendiet digitale terrengmodeller [Bjø89].

7.1 Generaliseringsmodeller

Generaliseringsmodell = funksjonell modell + informasjonsmodell

Databasegeneralisering og kartografisk generalisering

Semantisk- og romlig zoom, (Kartografisk zoom)

7.2 Finne karakteristiske punkter til en linje

Det å finne karakteristiske punkter til en linje, er en problemstilling som er viet stor oppmerksomhet i den kartografiske litteratur. Hva menes med et karakteristisk punkt? Svaret er ikke uten videre opplagt, fordi det må sees i forhold til den aktuelle bruken av dataene. Vi vil derfor måtte begrense oss til følgende noe upresise definisjon:

Definisjon 42 *De karakteristiske punkter til en linje, er de punkter som i forhold til en gitt anvendelse, har stor informasjonsverdi.*

Vanligvis vil karakteristiske punkter være linjens ekstremalpunkter, altså punkter i områder av linjen der skaren av tangenter til linjen raskt skifter retning.

Eksempel: En landmåler som skal kartlegge et terreng, måler inn de karakterisktiske punkter.

Hensikten med å finne en linjes karakteristiske punker kan være tofoldig.

1. Redusere antall punkter i en digitalisert linje for å oppfylle datatekniske krav som minsket lagerplass, kortere beregningstid etc. .

2. Fjerne unødvendige buktninger ut fra et perseptuelt krav (redusere entropien).

De metoder som her skal behandles, kalles ofte for *silere*. Før vi går videre, skal det presiseres hva vi legger i dette begrepet.

Definisjon 43 *En siler er en metode som på grunnlag av et eller annet kriterium fjerner punkter fra en digitalisert linje.*

Siden vi har å gjøre med digitale data, vil vi måtte anvende en *interpolasjonsmetode* for å rekonstruere de mellomliggende deler av den aktuelle linjen. Det er uten videre klart at det må bestå en avhengighet mellom siler og interpolator. Selv om den etterfølgende framstilling baserer seg på lineære silere, kan metodene lett tilpasses ikke-lineære silere. De lineære silere har forøvrig den fordel at de kan brukes sammen med en stor gruppe av interpolatorer.

Selv om det finnes en rik litteratur om metoder for å finne de karakteristiske punkter til en linje, er det få metoder som er ibruk i kommersielt tilgjengelige kartsystemer. Vi skal i det efferfølgende beskrive noen sentrale metoder innenfor nevnte gruppe silere.

7.2.1 DouglasPeucker-algoritmen

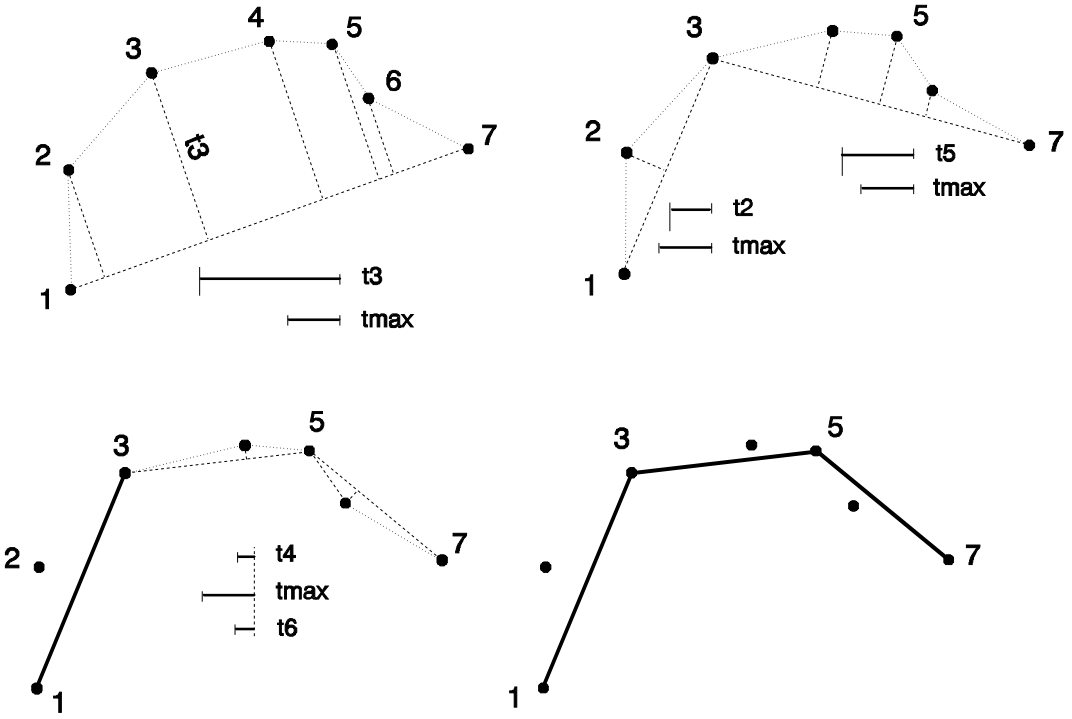
DouglasPeucker-algoritmen [DP73] kan avledes av stripetreet. (Peucker har skiftet navn til Poicker, så algoritmen forekommer i nyere litteratur også under denne betegnelsen.) Algoritmen, som vi vil betegne *DP-algoritmen*, kan formuleres på flere måter, men en rekursiv formulering er den som anbefales. Algoritmen er illustrert i figur 7.1. DP-algoritmen må sies å være en vellykket formulering. Den har en stor utbredelse og oppfattes som en standard innen digital kartografi (alle fagfolk med respekt for seg selv, kjenner DP-algoritmen). Som et minimumskrav bør alle systemer som kaller seg for kartsystemer, tilby DP-algoritmen eller varianter av denne.

Siden DP er en *global* algoritme, må alle punkter i linjen vær gitt før algoritmen kan startes. La punktsekvensen være gitt ved $(p_1, p_2, \cdots, p_n)$. Algoritmen begynner med å trekke en rett linje (*basislinje*) mellom endepunktene p_1 og p_n. For alle

mellomliggende punkter p_i beregnes avstanden t_i til basislinjen. Deretter velges den største av disse avstandene, altså

$$t_j = \max_{i=2}^{n-1}(t_i)$$

som deretter sammenliknes mot en applikasjonsavhengig toleranse t_{max}. Dersom $t_j > t_{max}$, velges det tilhørende punkt j som splittpunkt og algoritmen gjentas rekursivt med de to segmentene $(p_1, p_2, \cdots, p_j)$ og $(p_j, p_{j+1}, \cdots, p_n)$. Dersom $t_j \leq t_{max}$, kastes de mellomliggende punkter ut og endepunktene i det aktuelle segmentet tas inn i den forenklede linjen.



Figur 7.1: DouglasPeucker-algoritmen.

```
type
    punkt = record
        x,y: real
    end;
    linje = record
        L: array [ 1..Nmax ] of punkt;
        Np: integer {aktuell lengde av L}
    end;
var
    A: linje; {linje som skal forenkles}
```

```
    B: linje; {forenklet linje}
    tmax: real; {toleranse}


function FinnSplittpunkt(fra,til: integer; var splittpunkt: integer;
            t: real);
    {Global variabel: A}
    {fra,til: første og siste punkt i linjesegmentet}
    {splittpunkt: det mest karakteristiske punkt i linjesegmentet}
    {t: avstanden fra splittpunkt til den rette linje mellom fra og til}
var
    i: integer;
begin
    if fra = til - 1 then begin
        t := 0;
        splittpunkt := til;
    end
    else begin
        t := 0;
        for i := (fra + 1) to (til - 1) do begin
            if Avstand(fra,til,i) > t then begin
                t := Avstand(fra,til,i);
                splittpunkt := i
            end; {if}
        end; {do}
    end; {if}
end; {FinnSplittpunkt}


procedure DouglasPeucker(fra,til: integer);
    {rekursiv DouglasPeucker-algoritme}
    {Globale variable: A, B og tmax}
    {fra,til: første og siste punkt i det aktuelle linjesegmentet}
var
    splittpunkt: integer;
    t: real;
begin
    FinnSplittpunkt(fra,til,splittpunkt,t);
    if t > tmax then begin
        DouglasPeucker(fra,splittpunkt);
        DouglasPeucker(splittpunkt,til);
    end
    else begin
```

```
            TegnLinje(A.L[fra],A.L[til]);
            B.Np := B.Np + 1; { oppdaterer forenklet linje}
            B.L[Np] := A.L[splittpunkt];
        end; {if}
    end; {DouglasPeucker}
```

Tidsforbruket til algoritmen er i verste fall $O(n^2)$, hvor n er antall punkter i den linjen som skal forenkles. Dersom vi kan anta at hvert splittpunkt halverer sitt tilhørende intervall, blir tidsforbruket:

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

Det lar seg vise at dette også blir det gjennomsnittlige tidsforbruket.

De kartografiske egenskaper til DP-algoritmen har vært gjenstand for en rekke studier. Som hovedkonklusjon kan man si at algoritmen er flink til å plukke ut karakteristiske punkter, men den har et begrenset aksjonsområde. Dersom t_{max} blir stor nok, vil nemlig fasongen til det forenklede objektet kunne bli ugjenkjennelig. Dette vil forøvrig bli diskutert i større bredde i neste utgave av kompendiet.

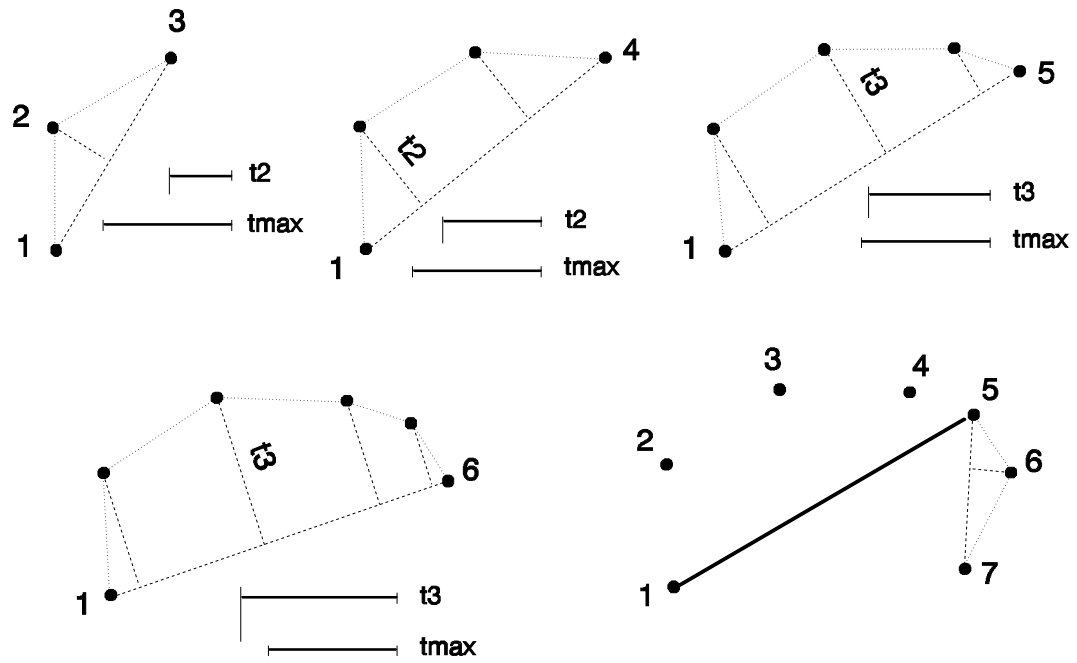
Som nevnt i forbindelse med stripetreet, er dette en hierarkisk datastruktur som benytter DP-kriteriet som forgreiningskriterium. Ved å ta en *prefikstraversering* av stripetreet og *stanse rekursjonen* når det omskrivende rektangel får en høyde over grunnlinjen større enn t_{max}, har vi funnet de punkter i linjen som tilfredsstiller DP-kravet t_{max}. Fordelen med en slik framgangsmåte er at vi raskt kan skifte mellom ulike kartmålestokker.

7.2.2 Langs algoritme

Langs algoritme har visse likhetstrekk med DouglasPeucker-algoritmen (DP), men det er en vesentlig forskjell idet Lang er en *lokal* algoritme mens DP er global. Dette gjør at Langs algoritme ikke er så flink til å plukke ut karakteristiske punkter som DP-algoritmen. Ved å angi en forsiktig grense til toleransen, kan de mindre gode sider ved Langs algoritme holdes innenfor en akseptabel grense.

Når man gir avkall på kartografisk egnethet, så er det for å oppnå en annen fordel. På grunn av at Lang er en lokal algoritme, er den nemlig egnet til å sile ut overflødige punkter fra en *strøm* av punkter, for eksempel strømmen av data fra et digitaliseringsbord eller en analytisk plotter.

Hovedideen i algoritmen skal kort forklares. Når de tre første punktene er kommet inn, sjekkes om det mellomliggende punktet kan kastes vekk. Dersom dette er tilfellet, hentes inn et nytt punkt og det sjekkes om de to mellomliggende punkter kan kastes vekk. Slik fortsetter innhenting av nye punkter og testing av mellomliggende punkter inntil minst ett mellomliggende punkt ikke kan kastes ut. Dersom dette skjer etter at punkt p_i er hentet inn, vet vi at ved å gå tilbake til punkt p_{i-1}, vil alle mellomliggende punkter kunne kastes ut. Punkt p_{i-1} blir følgelig et punkt i



Figur 7.2: Langs algoritme.

den forenklede linjen. Algoritmen gjentas ved å initiere p_{i-1} og p_i som de to første punktene i datastrømmen. Algoritmen er illustrert i figur 7.2.

```
var
    A: linje; {datastrøm som skal forenkles, innstrømmen}
    B: linje; {forenklet datastrøm}
    tmax: real; {toleranse}


procedure Lang;
    {Langs algoritme}
    {Globale variable: A, B og tmax}
var
    slutt: boolean;
    i,j: integer;
    t: real;
begin
    LesPunkt(A.L[1]); {initierer innstrømmen}
    LesPunkt(A.L[2]);
    B.Np := 1; {initierer forenklet datastrøm}
    B.L[Np] := A.L[1];
    repeat {for alle segmenter}
        i := 2;
```

```
        t := 0;
        while t < tmax then begin {danner et segment}
            i := i + 1; {henter inn et nytt punkt}
            LesPunkt(A.L[ i ]);
            for j := 2 to i - 1 do begin
                if Avstand(1,i,j) > t then t := Avstand(1,i,j)
            end; {do}
        end; {while}
        {splittpunktet er nå funnet og blir punkt (i-1)}
        TegnLinje(A.L[ 1 ],A.L[ i-1 ])
        B.Np := B.Np + 1; {oppdaterer forenklet datastrøm}
        B.L[ B.Np ] := A.L[ i-1 ];
        A.L[ 1 ] := A.L[ i-1 ]; {initierer innstrømmen på nytt}
        A.L[ 2 ] := A.L[ i ];
    until slutt;
end; {Lang}
```

For hver gang et nytt punkt blir hentet inn, må avstanden t_j fra de mellomliggende punkter ($p_j;\ j = 2, i-1$) til den rette linjen mellom punkt p_1 og punkt p_i beregnes og sammenliknes med den gitte toleransen t_{max}. Dette gjør at algoritmen får en kvadratisk vekstrate med omsyn til antall punkter i segmentet.

La oss anta at datastrømmen inneholder n punkter og at algoritmen plukker ut m av disse punktene. Dersom vi kan anta at datastrømmen på denne måten blir delt i like store segmenter, vil hvert segment inneholde $(\frac{n-1}{m-1} + 1)$ punkter som gir $j = (\frac{n-1}{m-1} - 1)$ mellomliggende punkter. For hvert segment må den indre løkken gjennomløpes følgende antall ganger:

$$1 + 2 + \cdots + j = \frac{j(j+1)}{2} = \frac{(\frac{n-1}{m-1} - 1)(\frac{n-1}{m-1})}{2} \approx \frac{1}{2}\left(\frac{n}{m}\right)^2$$

Det å behandle ett segment vil derfor ha tidskompleksiteten $T = O((n/m)^2)$. Siden linjen består av flere segmenter, etter forutsetningen (m-1), er tidsforbruket ved å behandle hele linja gitt ved:

$$T = O\left(\frac{n^2}{m}\right)$$

Som vi ser er det for tidsforbrukets vedkommende gunstig at at m er stor, altså at segmentene er små. Dersom m blir stor nok, vil ingen punkter i datastrømmen bli kastet ut, og vi får et tidsforbruk som vokser lineært med n. Dette kan forøvrig tyde på at den valgte verdi for t_{max} er for liten. På den andre siden er 2 den minste verdi m kan ha. Når m antar denne verdien, blir linjen delt i bare ett segment og tidsforbruket får sin største verdi.

7.3 Diverse enkle silere

En gruppe silere tar ikke eller i liten grad tar hensyn til karakteristiske punkter i de linjer som skal forenkles. Metodene er lite ressurkrevende og benyttes som regel på strømmen av data fra et digitaliseringsinstrument.

7.3.1 N-punktsiler

Hvert n-te punkt overføres til til den forenklede linjen.

7.3.2 Avstandssiler

Den Evklidiske avstand mellom punktene i den forenklede linjen skal holde et visst minstemål S_{min}. det er vanlig å benytte en fast verdi for S_{min}.

7.3.3 Tidsintervallsiler

Ved digitalisering kan det i noen tilfeller være hensiktmessig å registrere punkter med visse tidsintervall.

7.4 Formendring av en linje

Vi skal her beskrive noen metoder som har til hensik å endre formen til en digitalisert linje. Man kan si at DP-algoritmen også har en slik hensikt og derfor burde behandles i dette avsnittet. Dette er forsåvidt riktig, men DP har også ha en annen hensikt, nemlig å redusere datavolumet. DP-algoritmen har derfor både en perseptuell hensikt og en datateknisk hensikt.

De algoritmer vi her tenker på, har til eneste hensikt å endre formen til en linje, men begrunnelsen for formendringen kan bero på flere forhold. En begrunnelse kan være gitt ut fra et perseptuelt krav, en annen begrunnelse kan være ønsket om å fjerne unøyaktigheter i datagrunnlaget.

7.4.1 Middeltall etter vekt

Metoden middeltall etter vekt går ut på å beregne nye posisjoner for punktene. Gitt punktfølgen $P_1, P_2, \cdots, P_n$. Punktenes nye koordinater finnes av følgende formel:

$$x'_i = \sum_{j=-k}^{k} p_{i+k} \cdot x_{i+k} \tag{7.1}$$

$$y'_i = \sum_{j=-k}^{k} p_{i+k} \cdot y_{i+k} \tag{7.2}$$

hvor $1 \leq i + k \leq n$ og hvor summen av vektene er lik 1, altså:

$$\sum_{j=-k}^{k} p_{i+k} = 1$$

Det springende punkt er å fastsette størrelsen k og fordelingen av vektene p_{i+k}. Et typisk valg er $k = 1$ eller $k = 2$.

Slik metoden er formulert, har vi dårlig kontroll med hvor store de enkelte forskyvninger blir. Det anbefales derfor at differansene $\mid x' - x \mid$ og $\mid y' - y \mid$ testes mot en toleranse t_{max}. Dersom denne verdien overskrides, kan man for eksempel gå over til en vektsfunksjon som har mindre glattingseffekt.

Eksempel: Vi velger $k = 2$ og en vektsfunksjon der det punktet som skal få tilordnet nye koordinater, gis størst vekt.

$$\begin{aligned} x'_i &= 0.05x_{i-2} + 0.20x_{i-1} + 0.50x_i + 0.20x_{i+1} + 0.05x_{i+2} \\ y'_i &= 0.05y_{i-2} + 0.20y_{i-1} + 0.50y_i + 0.20y_{i+1} + 0.05y_{i+2} \end{aligned}$$

Det er klart at jo større vekt punkt i tillegges, jo mindre blir effekten av glattingen.

Metoden kan ha to hensikter:

1. Fjerne unødvendige småbevegelser ut fra et persepuelt synspunkt.

2. Fjerne støy i dataene som skyldes unøyaktigheter i datagrunnlaget. For eksempel vil en rekke små buktninger kunne forklares ved slenger under digitaliseringen.

7.4.2 Minste kvadraters tilpasning

7.4.3 Epsilon-filtrering

7.5 Legge en glatt linje gjennom en punktfølge

7.5.1 Akimas metode

7.5.2 Bezier-kurver

7.5.3 B-spline

7.5.4 Kubisk spline

7.6 Forenkling av nettverk

7.7 Rastermetoder

Ved å formulere generaliseringmetoder på rasterbilder (kvadtrær), kan enkelte operasjoner bli langt mere effektive enn om de ble formulert på vektordata.

7.7.1 Svell og krymp

Ved å svelle eller krympe objekter i et binært bilde, oppnås generaliseringeffekter. Svell vil gjøre at nærliggende objekter gror sammen, mens krymp vil gjøre at objekter under en viss størrelse forsvinner.

Svell og krymp lar seg implementere på linære kvadtrær slik at algoritmene blir meget effektive.

I forbindelse med GIS er det ofte aktuelt å etablere såkalte *korridorer*. En korridor framkommer ved å ta en svell av det aktuelle objektet.

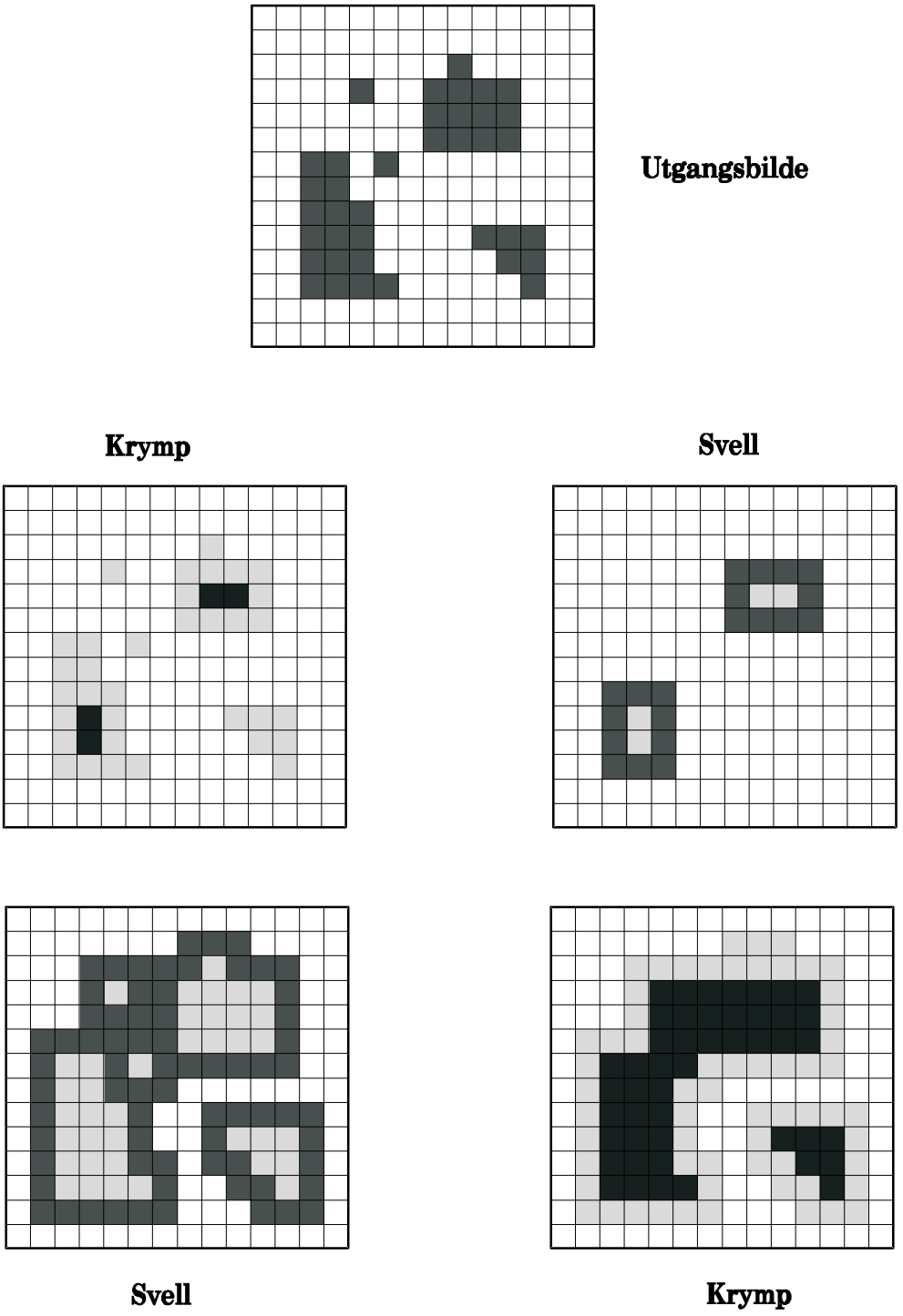
Som det framgår av figur 7.3, kombineres svell og krymp til sammensatte operasjoner.

1. Svell, krymp. Slår sammen nærliggende objekter.

2. Krymp, svell. Fjerner objekter under en viss størrelse.

3. Kombinasjoner av 1. og 2. Ved først å kjøre 1. lar vi nærliggende objekter gro sammen. Deretter fjernes små objekter ved å kjøre 2. Rekkefølgen av 1. og 2. kan selvsagt byttes om. Dette er avhengig av den enkelte applikasjon.

Grunnen til at en krymp etterfølger en svell eller omvendt, er for å kompensere for de lokale målestokksendringer vi ellers ville fått.



Figur 7.3: Generaliseringseffekter ved svell og krymp av objekter.

Eksempel: Hus som ligger nære hverandre blir representert som ett objekt (superobjekt) i små kartmålestokker. Svell kan benyttes for å finne ut hvilke hus som skal tilhøre et superobjekt.

7.7.2 Filtermetoder

Filtermetoder, som er middeltallsberegninger etter vekt, er analoge til metodene i seksjon 7.4.1. Dersom samtlige vekter gis positive verdier, er filteret et *glattingsfilter*. Ved å innføre negative vekter i filteret, oppnås *kantforsterking* ved at differanser mellom nabopixler blir forsterket.

Metodene benyttes i stor grad innen digital bildebehandling, men de har også anvendelser innen digital kartografi.

Eksempel: Ved analytisk skyggelegging kan det være aktuelt å foreta en kontrastjustering. Dersom det er store sprang mellom gråtonene til nabopixler, benyttes et glattingsfilter. I motsatt fall benyttes et kantforsterkingsfilter. Dette filteret kan gjerne utformes slik at det bare er kanter som går i visse retninger som blir forsterket.

7.7.3 Lineær strekk

Anta et en kvantitativ variabel g er assosiert til hvert pixel i et rasterbilde. Nye verdier til g beregnes av:

$$g' = a \cdot g + b$$

hvor a og b er parametre i den lineær transformasjonen.

Metoden kan anvendes i forbindelse med analytisk skyggelegging. Dersom gråverdiene klumper seg sammen innenfor et lite gråtoneintervall, kan en lineær strekk benyttes for å spre gråtonene ut over hele gråtoneintervallet til det aktuelle tegneutstyret.

7.8 Datakomprimering

7.8.1 Parametriserte kurver

7.8.2 Fraktaler

7.9 Generalisering av visse objekttyper

7.9.1 Hus

Det å kjøre en DouglasPeucker-algoritme på omrissmodellen av et hus, er lite vellykket. For denne objekttypen må andre metoder benyttes.

7.10 Prinsipale komponenter

Bibliografi

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullmann. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[Alt94] David Altman. Fuzzy set theoretic approaches for handling imprecision in spatial analysis. *Int. Journal of Geographical Information Systems*, 8(3):271–289, 1994.

[Bjø87] Jan Terje Bjørke. *Cartographic Communication in Computer-Based Environments*. PhD thesis, The Norwegian Institute of Technology, Trondheim, 1987.

[Bjø88] Jan Terje Bjørke. Quadtrees and triangulation in digital elevation models. In *International Archives of Photogrammetry and Remote Sensing*, pages 38–44. Committee of the 16the International Congress of ISPRS, 1988. Volume 27, part B4.

[Bjø89] Jan Terje Bjørke. Digitale terrengmodeller. Kompendium, institutt for kart og oppmåling, NTH, 1989. Foreløpig utgave.

[Bjø95] Jan T. Bjørke. Fuzzy set theoretic approach to the definition of topological spatial relations. In J.T. Bjørke, editor, *ScanGIS'95*, pages 197–206, 7034 Trondheim, Norway, 12-14 June 1995. Department of Surveying and Mapping, Norwegian University of Science and Technology.

[Bjø96] Jan Terje Bjørke. Teorien om uskarpe mengder (fuzzy set): Eksempler på anvendelser innen gis med hovedvekt på topologiske relasjoner. In *Kartdagene 1996*, pages 168–181, Storgt. 11, 3500 Hønefoss, Norway, 20.-22. (23.) mars 1996. Norges Karttekniske Forbund.

[Bla82] Donald W. Blackett. *Elementary Topology, A Combinatorial and Algebraic Approach*. Academic Press, Inc., 1982.

[Com79] Douglas Comer. The ubiquitous b-tree. *Comp. Surveys*, 11(2):121–137, 1979.

[DP73] David H. Douglas and Thomas K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its carricature. *The Canadian Cartographer*, 10(2):112–122, des. 1973.

[EF91] Max J. Egenhofer and Robert D. Franzosa. Point-set topological spatial relations. *International Journal of Geographical Information Systems*, 5(2):161–174, 1991.

[EF93] Max J. Egenhofer and Robert D. Franzosa. *On the Equivalence of Topological Relations.* Technical Report Department of Surveying Engineering, University of Maine, Orono, ME, 1993.

[EH91] Max J. Egenhofer and J. Herring. *Categorizing Binary Topological Relations Between Regions, Lines and Points in Geographical Databases.* Technical Report Department of Surveying Engineering, University of Maine, Orono, ME, 1991.

[Ein87] John Einbu. *Geodatabaser.* Tapir forlag, Universitetet i Trondheim, 1987. ISBN 82-519-0818-3.

[FNPS79] Ronald Fagin, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[G$\ddot{8}$9] O. Günther. The design of the cell tree: an object-oriented index structure for geometric databases. In *Proceedings of the Fith IEEE International Conference on Data Engineering*, pages 598–605, Los Angeles, februar 1989.

[Gem67] Michael C. Gemignani. *Elementary Topology.* Addison Wesley, 1967.

[Gut84] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the SIGMOD Conference*, pages 47–57, Boston, juni 1984.

[Hin85] Klaus Helmer Hinrichs. *The grid file system: implementation and case studies of application.* PhD thesis, Swiss Federal Institute of Technology, Zürich, 1985.

[HLS93] K. Hofstad, S. Løland, and P. Scott. *Norsk Data Ordbok.* Universitetsforlaget, 5 edition, 1993. Norsk språkråds komité for dataterminologi.

[HR90] Per Holm and Jon Reed. *Topologi.* Universitetsforlaget, 1990.

[Hus77] Taqdir Husain. *Topics and Maps.* Plenum Press, New York, 1977.

[KF88] George J. Klir and Tina A. Folger. *Fuzzy sets, uncertainty, and information.* Prentice Hall, 1988.

[KG91] Arnold Kaufmann and Madan M. Gupta. *Introduction to fuzzy arithmetic*. International Thompson Computer Press, USA, 1991.

[KV91] Vassiliki J. Kollias and Achilleas Voliotis. Fussy reasoning in the development of geographical information systems. FRSIS: a prototype soil information system with fuzzy retrieval capabilities. *International Journal of Geographical Information Systems*, 5(2):209–223, 1991.

[Lip65] Seymour Lipschutz. *General Topology*. Schaum Publishing co., 1965.

[Lit80] Witold Litwin. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th International Conference on Very Large Data Bases*, pages 212–223, Montreal, oktober 1980. IEEE.

[Mas91] William S. Massey. *A Basic Course in Algebraic Topology*. Springer-Verlag New York Inc., 1991.

[ME94] David M. Mark and Max J. Egenhofer. Calibrating the meaning of spatial predicates from natural language: Line-region relations. In T. Waugh and R. Healey, editors, *SDH 94, Sixth International Symposium on Spatial Data Handling, 5th-9th September 1994, Edinburgh, Scotland*, pages 538–553. Taylor&Francis, 1994.

[Sam84] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[Sam90a] Hanan Samet. *Applications of Spatial Data Structures*. Addison-Wesley, 1990. ISBN 0-201-50300-X.

[Sam90b] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990. ISBN 0-201-50255-0.

[Sch79] Arlo W. Schurle. *Topics in Topology*. Elsevier North Holland, Inc., 1979. NTUB 515.1.

[Tam81] Markku Tamminen. *The EXCELL method for efficient geometric access to data*. PhD thesis, Helsinki University of Technology, 1981. Mathematics and Computer Science Series No. 34.

[Tim87] E. Time. *Behandling av usikkerhet i ekspertsytem ved hjelp av Fuzzylogikk*. SINTEF Runit, Trondheim, 1987. rapportnr. STF14 A87022.