INF1020 – Algoritmer og datastrukturer

Forelesning 14: Gjennomgang av eksamen vår 2001 oppgave 1,2,4

Arild Waaler

Institutt for informatikk, Universitetet i Oslo

27. november 2006

Oppgave 1–a

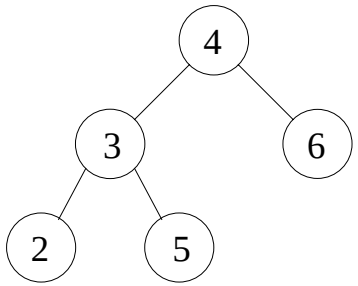
Programmer en ikke-rekursiv metode som returnerer antall forekomster av tallet `n` *i et binært søketre med rot* `t`.

```
int antall( BinNode t, int n ) {
  int i = 0;
  BinNode peker = t;

  while ( peker != null ) {
    if ( peker.data == n ) i++;
    if ( peker.data <= n ) peker = peker.vsub;
    else  peker = peker.hsub;
  }
  return i;
}
```


Oppgave 1–b

Programmer en metode som returnerer `true` *dersom treet med rot* `t` *er et binært søketre i henhold til definisjonen over og* `false` *ellers.*

Trikset er å unngå at følgende tre kommer ut som et binært søketre:



Det er ikke tilstrekkelig å kun teste en nodes verdier lokalt mot verdiene til barna (som vi gjør når vi setter inn i et binært søketre).

Løsning: Angi et intervall av lovlige verdier inn til metoden.

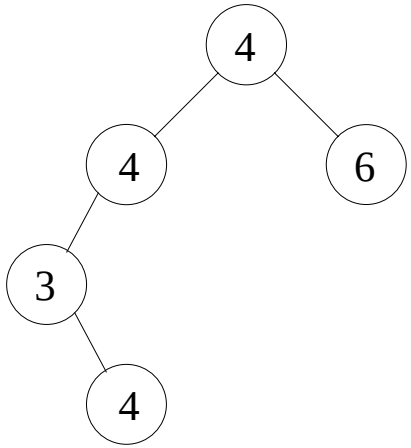
```
boolean bst( BinNode t ) {
  return bst( t, -UENDELIG, UENDELIG );
}

boolean bst( BinNode t, int min, int max ) {
  if( t == null ) return true;
  int d = t.data;
  if( d < min || d > max ) return false;
  return( bst( t.vsub, min, d ) && bst( t.hsub, d+1, max ) );
}
```

Oppgave 1–c

Programmer en metode som gir en sortert utskrift av alle verdiene i treet med rot `t` *som er i intervallet fra og med* `min` *til og med* `max`*. Du skal ikke skrive ut samme tall mer enn en gang. Legg vekt på at metoden skal være så rask som mulig. Angi kompleksitet.*

Oppgaven løses elegant med infiks-traversering samtidig med at vi minker intervallet av tall som skal skrives ut.

```
void skrivIntervall( BinNode t, int min, int max ) {
  if( t == null || max < min ) return;
  int d = t.data;
  if( d > min ) skrivIntervall( t.vsub, min, d-1 );
  if( d >= min && d <= max ) System.out.println( t.data );
  if( d < max ) skrivIntervall( t.hsub, d+1, max );
}
```

Anta at 4 ligger i intervallet.

- I treet vil den øverste 4–verdien skrives ut.
- Den midterste 4–noden vil besøkes, men ikke skrives ut.
- Den nederste noden med 4 vil ikke besøkes i traverseringen

Alternativ løsning:

- la metoden returnere den høyeste verdi som har blitt skrevet ut
- traverser hele treet inntil det besøkes en node med tall utenfor intervallet
- I treet over vil da den nederste 4–noden skrives ut, men hele treet vil besøkes (hvis alle nodene ligger i intervallet)

Selv om den første løsningen i noen tilfeller vil avskjære søkerommet mest, vil begge løsningene være lineære, dvs. $O(n)$, der n er antall noder i treet.

Oppgave 1–d

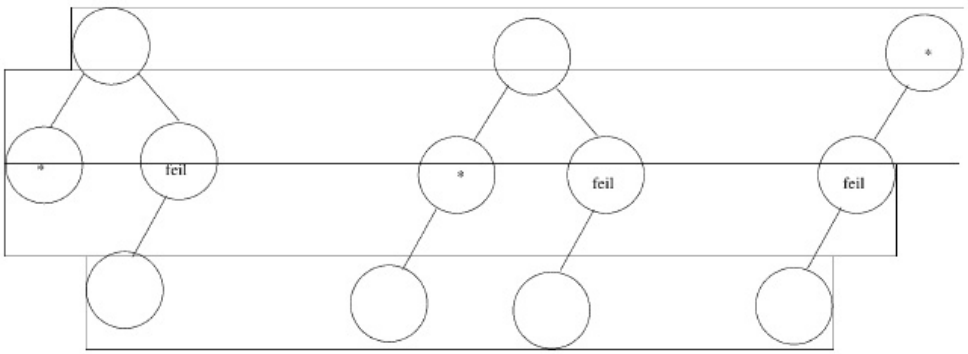
Et binærtre er *komplett* dersom hvert nivå i treet er fullt, med et mulig unntak for det laveste nivået, som skal være fylt fra venstre mot høyre.

Programmer en boolsk metode

```
boolean komplett( BinNode t )
```

som returnerer `true` *dersom treet med rot* `t` *er komplett og* `false` *ellers.*

- Oppgaven kan løses både ved bredde-først og dybde-først traversering.
- Viktig å lage figurer og analysere



Oppgave 1–d: Observasjon

Det finnes fire muligheter for strukturen til en node i et binærtre. En node kan ha

a ingen barn,

b venstre barn, men ikke høyre barn,

c høyre barn, men ikke venstre barn,

d to barn.

Merk at:

- Tilfellet c er uforenelig med at treet er komplett.
- Hvis treet er komplett og tilfelle a eller b inntreffer når vi går bredde–først fra venstre mot høyre, må det være fordi vi har kommet til nederste nivå i treet.
- Tilfellet d er uproblematisk når vi søker bredde–først gitt at noden kan ha barn.

Oppgave 1–d: Løsning ved traversering bredde–først

Koden traverserer treet nivåvis og gjør bruk av en kø-abstraksjon.

- *Hvis a eller b inntreffer, og vi senere besøke noder som har barn, er ikke treet komplett. Ellers er det komplett.*
- En boolsk variabel `flagg` til true når tilfelle a eller b inntreffer.
- Metoden returnerer false når vi tar ut en node med barn av køen mens flagget er true.
- Hvis denne situasjonen aldri inntreffer, returneres true.
- Koden blir enklere hvis vi behandler null-referanser i køen:

```
class BinNodeQueue {
  Queue q = new Queue();

  void enqueue(BinNode node){if(node!=null) q.enqueue(node);}

  BinNode dequeue(){ return (BinNode) q.dequeue(); }
}
```



```
boolean komplett( BinNode t ) {
  boolean flagg = false;
  BinNodeQueue q = new BinNodeQueue();
  BinNode denne;

  q.enqueue( t );
  while( !q.isEmpty() ) {
    denne = q.dequeue();
    if( denne.vsub==null && denne.hsub!=null ) return false;
    if( flagg && denne.vsub!=null ) return false;
    // (flagg && denne.hsub != null) har alt gitt false
    if( denne.vsub==null || denne.hsub==null ) flagg = true;
    q.enqueue( denne.vsub );
    q.enqueue( denne.hsub );
  }  // q.isEmpty()
  return true;
}
```

Oppgave 1–d: Løsning ved traversering dybde–først

Et tre er

- *rett* dersom det er komplett og alle løvnoder ligger på samme nivå
- *halt* dersom det er komplett og ikke alle løvnoder ligger på samme nivå.

Det tomme treet er rett, mens i et halt tre er det laveste nivået av noder ikke fullt, men fylt fra venstre mot høyre.
Observasjon: Et tre med rot `t` er komplett hvis og bare hvis en av følgende betingelser holder:

- `t` er `null`
- `t.vsub` er rett med høyde h og `t.hsub` er enten rett eller halt med høyde enten h eller $h-1$
- `t.vsub` er halt med høyde h og `t.hsub` er rett med høyde $h-1$

Vi kan teste betingelsene ved å traversere treet rekursivt postorder.
Vi får ekstra kompakt kode ved å definere en integerverdi *tag* på følgende måte:

- tag er 0 hvis treet ikke er komplett
- (2 + høyden) hvis det rett
- -(1 + høyden) hvis det er halt.

Eksempler:

- tomme trær er rette, har høyde -1 og tag 1
- trær med kun en rotnode er rette, har høyde 0 og tag 2
- trær med kun en rotnode og et venstre barn er halte, har høyde 1 og tag -2
- rette trær med 3 noder har høyde 1 og tag 3

Merk at negative tall fungerer som en signal om at treet er halt.


```
boolean komplett( BinNode t ){ return( tag(t) != 0 );}

int tag( BinNode t ) {
  if( t == null ) return 1;
  int v = tag( t.vsub ); if ( v == 0 ) return 0;
  int h = tag( t.hsub ); if ( h == 0 ) return 0;
  if( v < 0 ) {
    if( v == -h ) return v - 1;
    else return 0;
  }
  if( h < 0 ) {
    if( v == 1 - h ) return h - 1;
    else return 0;
  }
  if( v == h ) return 1 + v;
  else if ( v == 1+h ) return -v;
  else return 0;
}
```


Oppgave 2

Oppgaven går ut på å lage metoder som skal bli brukt til å generere dagens samtalelogg. Dette er en fil der all samtaleinformasjon er ordnet primært etter kundenummer, sekundært etter start-tidspunkt for samtalene. Logg-fila skal altså begynne med data for den første samtalen til brukeren med lavest kundenummer og skal slutte med data for den siste samtalen til brukeren med høyest kundenummer.

- Anta at det er K kunder som har transaksjoner en gitt dag og at det er totalt
- Merk at samtalene i arrayen som skal sorteres ligger kronologisk.
- Dette betyr at de allerede er sortert etter tid, som er det ene kriteriet for sortering.
- Det gjelder da å sortere arrayen etter knr også *på en slik måte at den kronologiske ordningen bevares*.
- Vi sier at en sorteringsmetode som bevarer den kronologiske ordningen når den brukes til å sortere etter knr er *stabil*.

Stabile sorteringsalgoritmer

En algoritme er stabil dersom ethvert objekt B som initielt ligger et sted etter et objekt A i inputarrayen også ligger etter A etter sortering hvis B ikke ligger foran A i ordningen som det sorteres etter.

Hvis f.eks. A og B er like i henhold til kriteriet for sortering (i oppgaven gjelder det hvis to samtaler har samme knr), skal deres innebyrdes rekkefølge ikke endres under sortering.

- Innstikksortering (*insertion sort*) er alltid stabil.
- Flettesortering (*mergesort*) er stabil hvis flettingen velger objekt fra venstre del av den sorterte subarrayen før den høyre.
- *Quicksort* er ikke uten videre er stabil.

Oppgave 2–a

Programmer en metode som tar inn dagens samtaler som en array av lengde T og returnerer en array av samtaler ordnet som en logg.

Stikkord for løsning: Bucket-sort.

- Bucket-sort tar lineær tid i forhold til antall samtaler, det vil si $O(T)$.
- For å bevare den kronologiske ordningen mellom samtale–objektene kan vi la hver "bøtte" være bygget opp som en kø.
- Køen bør være implementert med en dynamisk struktur siden dette vil utnytte minnet bedre.
- Etter selve sorteringen, må innholdet i "bøttene" kopieres over til en array av SamtaleInfo-objekter.
- Også denne kopieringen tar lineær tid, siden hver samtale kopieres en gang.


```
SamtaleInfo[] loggOrdner( SamtaleInfo[] samtale ) {
  int index = 0;
  SamtaleInfo[] logg = new SamtaleInfo[samtale.length];
  // Buckets for hver kunde
  Queue hylle = new Queue[K+1];
  // initialiser hyllene
  for( int i = 1; i <= K; i++ )
    hylle[i] = new Queue();
  // legg objekter inn i hyllene
  for ( int i = 0; i < samtale.length; i++ )
    hylle[samtale[i].knr()].enqueue( samtale[i] );
  // flytt fra hyllene over i loggen
  for( int i = 1; i <= K; i++ )
    while( !hylle[i].isEmpty() )
      logg[index++] = (SamtaleInfo) hylle[i].dequeue();

  return logg;
}
```


Oppgave 2–b. Forslag 1: Hashing

Anta nå at selskapet bruker et 8-sifret telefonnummer som kundenummer.

Merk:

- Siden vi nå ikke lenger kan bruke kundenummerene som indekser i "buckets", kan vi ikke lenger bruke Bucket–sort direkte. T transaksjoner.

Bruk av hashing innebærer:

- Vi må hashe kundenummerene over til en array av køer og legge inn objektene som i 2–a.
- Når vi skal skrive til logg, må vi først sortere arrayen etter kundenummer siden hash-funksjonen vil spre kundenummer fritt over indeksene.
- krever $O(T)$ for å lese fra arrayen og skrive til logg og $O(K \log(K))$ for å sortere arrayen etter kundenummer.
- Dermed blir algoritmen $O(K \log(K) + T)$.

Oppgave 2–b. Forslag 2: Søketre

- Bygg opp et søketre basert på kundenummer.
- Dataverdiene i en node ligger i en kø (som i Oppgave 2–a).
- Når vi skal skrive til logg, traverserer vi treet infiks.
- Detter krever $O(T)$ for å lese fra arrayen og skrive til logg. Oppbyggingen av et binært søketre er $O(K \log(K))$ hvis vi bruker B–tre eller et rød-sort tre.
- Bruker vi et vanlig binært søketre, er oppbyggingen $O(K^2)$ i verste tilfelle og algoritmen $O(K^2 + T)$.
- Siden telefonnummer normalt vil spre seg jevnt over samtalene, kan vi forvente at algoritmen vil eksekvere i $O(K \log(K) + T)$ tid også hvis vi bruker binært søketre.

Oppgave 2–b. Forslag 3: Bruk en stabil sorteringsmetode.

- Hvis sorteringsmetoden er $n \log(n)$, vil løsningen være $O((K + T) \log(K + T))$.
- Merk at invarianten i oppgaven tilsier at vi kan bruke flettesortering. Denne er rask og enkel å gjøre stabil.
- Det er mulig å bruke *quicksort* og *heapsort* også, men disse er vanskeligere å gjøre stabile.
- Hvis vi skal gjøre det, kan vi ikke sortere bare etter knr–feltet, men må inkludere tidsattributtet også.
- Vi kan da bruke samme type ordning som man bruker i telefonkatalogen, der navnene først ordnes etter etternavn og dernest etter fornavn (her: først etter knr, dernest etter tid). Dette kalles en leksiografisk ordning.

Implementasjon av en leksiografisk ordning

I Java kan vi ved å bruke følgende `compareTo`–metode for klassen `SamtaleInfo`:

```
int compareTo( Object rhs ) {
  SamtaleInfo o = (SamtaleInfo ) rhs;
  if ( knr < o.knr ) return -1;
  if ( knr == o.knr ) return tid.compareTo( o.tid );
  return 1;
}
```


Kode for flettesortering

```
SamtaleInfo[] loggOrdner( SamtaleInfo[] samtale ) {
  SamtaleInfo[] tmp = new SamtaleInfo[samtale-length];
  flettSorter( samtale, tmp, 0, samtale.length - 1 );
  return samtale;
}

void flettSorter( SamtaleInfo[] a, SamtaleInfo[] tmp,
                  int lav, int hoy ) {
  if( lav < hoy ) {
    int m = ( lav + hoy ) / 2;
    flettSorter( a, tmp, lav, m );
    flettSorter( a, tmp, m + 1, hoy );
    flett( a, tmp, lav, m + 1, hoy );
  }
}
```

```
void flett( SamtaleInfo[] a, SamtaleInfo[] tmp,
            int i, int j, int jSlutt ) {
  int iSlutt = j - 1;
  int start = i;
  int k = i;
  while( i <= iSlutt && j <= jSlutt ) {
    if( a[i].knr <= a[j].knr ) tmp[k++] = a[i++];
    else tmp[k++] = a[j++];
  }
  // naa er enten i>iSlutt eller j>jSlutt
  while( i <= iSlutt ) tmp[k++] = a[i++];
  // kopier tilbake, k er en for stor
  for( k--; start <= k; k-- ) a[k] = tmp[k];
}
```


Oppgave 2–c

Anta nå at internminnet ikke lenger er stort nok til å lagre alle samtalene på en gang. Hvordan vil du endre datastrukturen?

- Merk at algoritmen må bruke disk til å lagre SamtaleInfo-objekter, og det er essensielt å minimalisere antall diskaksesser.
- B–trær er da et nærliggende valg. Vi kan bruke treet til å lagre telefonnummere og la løvnodene i treet inneholde referanse direkte til diskblokken.
- Dette vil gi T diskaksesser når fila leses.
- Dersom det er ledig plass i internminnet, vil en bedre løsning være å utnytte denne til samtale–objekter, for så å frigjøre plassen når minnet blir fullt.
- Dette kan organiseres ved å legge objektene i arrayer tilsvarende blokk–størrelsen på disken. Når en kunde har objekter som fyller en slik array, kan denne skrives til disk.
- På den måten kan man skrive flere samtaler til disken i en aksess, og dermed redusere antall aksesser.

Oppgave 4

- Gitt et $N \times N$-brett der hver rute kan være enten hvit eller sort.
- `boolean[][] brett = new boolean[N][N]`

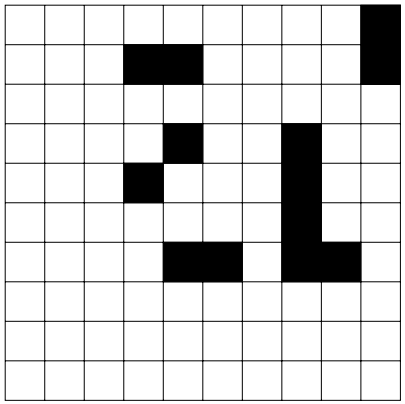
En *sort sky* er en mengde av sorte felter slik at

(1) en sky inneholder minst ett sort felt,

(2) hvis en sky har mer enn ett felt, er ethvert felt i skyen nabo til et annet felt i skyen,

(3) alle sorte felt på brettet som er nabo til et felt i skyen, er selv med i skyen.

(i) Tegn en figur som illustrerer begrepet om en sort sky.



Figuren inneholder 6 skyer (sorte felt diagonalt nær hverandre er ikke naboer ifølge definisjonen i oppgaveteksten).

(ii) *Programmer en metode som finner ut hvor mange sorte skyer det finnes på et gitt brett. Metoden har lov til å "ødelegge" representasjonen av brettet underveis.*

- Løsning: dybde-først-søk fra alle feltene på brettet, hvor sorte felt "viskes ut" underveis.

```
int antallSkyer() {
  int antall = 0;
  for (int i = 0; i < N; i++ ) {
    for (int j = 0; j < N; j++ ) {
      antall += viskut( i, j );
    }
  }
  return antall;
}
```



```
int viskut( int i, int j ) {
  if ( i<0 || j<0 || i>=N || j>=N ) return 0;
  if ( ! brett[i][j] ) return 0;
  else {
    brett[i][j] = false;
    viskut( i-1, j );
    viskut( i+1, j );
    viskut( i, j-1 );
    viskut( i, j+1 );
    return 1;
  }
}
```

- Returverdien fra `viskut` benyttes ikke av de rekursive kallene.
- Ellers angir denne tilbake til hovedmetoden at parameter-feltet var svart, og at vi altså fant en ny sky.