Algoritmer og datastrukturer
Kapittel 5 – Delkapittel 5.2

5.2 Binære søketrær

5.2.1 Hva er et binært søketre?

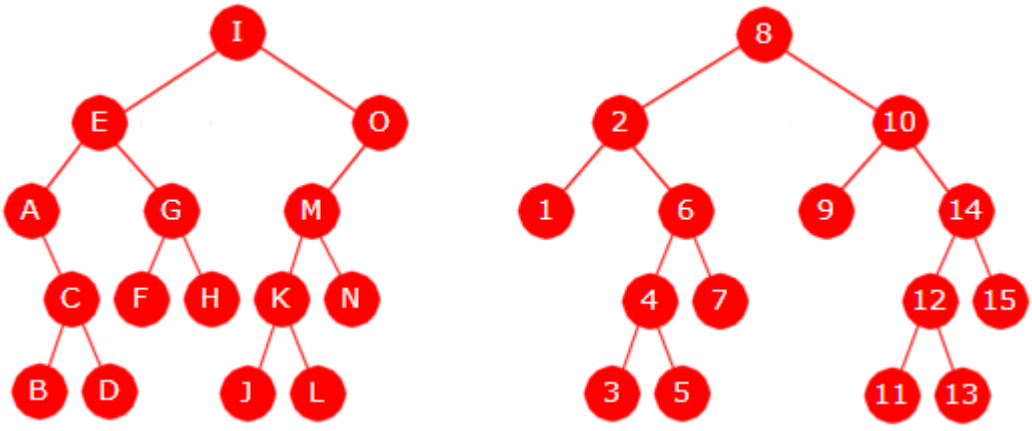
Navnet *binært søketre* (eng: binary search tree) indikerer at det er et binærtre som er tilrettelagt for søking. I *Delkapittel* 1.3 så vi på ordnede tabeller og binærsøk. Binærsøk er effektiv siden den er av logaritmisk orden. Det er imidlertid kostbart å vedlikeholde en ordnet tabell. Innlegging på rett sortert plass er av orden *n*. Også fjerning er av orden *n* siden «hullet» må tettes igjen. Et binært søketre er en datastruktur der søkingen i gjennomsnitt er omtrent like effektiv som binærsøk. I tillegg er også innlegging og fjerning i gjennomsnitt av logaritmisk orden. Prisen er imidlertid at vi bruker mer plass enn det en tabell trenger.

I et generelt binærtre - se *Delkapittel* 5.1 - var det ingen spesielle krav til nodenes verdier. I et binært søketre, der det jo handler om søking, må imidlertid verdiene kunne sammenlignes og ordnes. Binærsøk i en tabell krevde at tabellen var ordnet, men ikke at alle verdiene nødvendigvis var forskjellige. I noen lærebøker er det et krav at et binært søketre kun skal ha forskjellige verdier. Men her skal vi, på lik linje med ordnede tabeller og binærsøk, tillate like verdier, dvs. tillate såkalte duplikater.

For at et binærtre skal kunne være et søketre, må treets verdier oppfylle flg. krav:

> **Definisjon 5.2.1** *For hver node p gjelder: 1) hvis p har et ikke-tomt venstre subtre, er alle verdiene der* ***mindre enn*** *verdien i p og 2) hvis p har et ikke-tomt høyre subtre, er alle verdiene der* ***større enn eller lik*** *verdien i p. Et tomt tre og et tre med kun én node er per definisjon et binært søketre.*

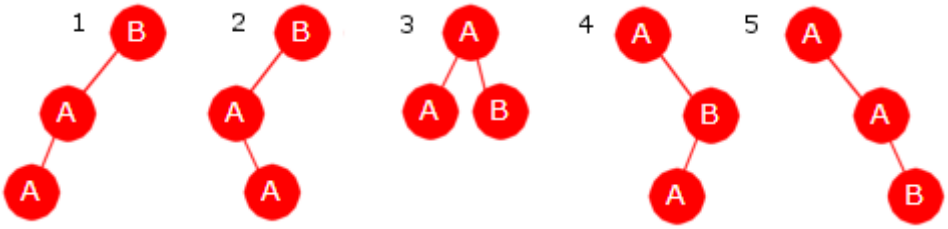
OBS: Definisjonen gir at verdiene i et binært søketre er sortert (stigende) i *inorden*.



Figur 5.2.1 a) : To binære søketrær - et med bokstaver og et med heltall

I *Figur* 5.2.1 *a)* har vi to binære søketrær. Nodeverdiene i det venstre er bokstaver med alfabetisk ordning og i det andre er det tall med vanlig ordning. Begge oppfyller *Definisjon* 5.2.1. I inorden blir rekkefølgene henholdsvis *A*, *B*, *C*, . . . , *O* og 1, 2, 3, ... , 15.

Definisjon 5.2.1 er strengere enn det at verdiene er sortert stigende i inorden. Det finnes binære trær som oppfyller det siste, men som ikke oppfyller definisjonen. I *Figur* 5.2.1 *b)* under er det tegnet fem binære trær som alle inneholder *A* to ganger og *B* en gang:



Figur 5.2.1 b) : Binære trær som inneholder A, A og B

Det første og midterste treet er ikke binære søketrær. De er sortert i inorden, dvs. *A, A, B*. Men *A* kan ikke være i det venstre subtreet til *A*. De andre trærne er binære søketrær. Det finnes ikke flere binære søketrær enn de tre som kun inneholder *A* to ganger og *B* en gang.

OBS: Hvis alle verdiene er forskjellige, er *Definisjon* 5.2.1 ekvivalent med at verdiene er sortert stigende i inorden. Men, som nevnt over, tillater vi her at et binært søketre kan ha duplikatverdier. Se *Oppgave* 3 for en alternativ måte å definere et binært søketre.

Oppgaver til Avsnitt 5.2.1

1. Søk på internett med *binary search tree* som søkestreng. Hva finner du?
2. *Definisjon* 5.2.1 definerer et binært søketre. Anta at vi i steden bruker flg. definisjon: «Et tre er et binært søketre hvis vi for enhver node *p* har at verdien i venstre barn (hvis *p* har venstre barn) er mindre enn verdien i *p*, og at verdien i høyre barn (hvis *p* har høyre barn) er større enn eller lik verdien i *p*». Er det mulig å lage et binærtre som oppfyller dette, men som ikke oppfyller *Definisjon* 5.2.1?
3. Vi kan definere et binært søketre slik: «Et binærtre er et binært søketre hvis verdiene er sortert stigende i inorden og hvis *p* og *q* er to noder med samme verdi slik at *q* kommer etter *p* i inorden, så ligger *q* i det høyre subtreet til *p*». Vis at denne definisjonen er ekvivalent med *Definisjon* 5.2.1.
4. I teksten etter *Figur 5.2.1 a)* står det at i det venstre treet kommer verdiene slik i inorden: *A, B, C, . . . , O* og i det høyre treet slik: 1, 2, 3, ... , 15. Sjekk at det stemmer. Sjekk definisjonen av *inorden* hvis du ikke husker den.
5. Det høyre treet i *Figur 5.2.1 a)* inneholder heltall. Anta at treet også kan inneholde desimaltall. Hvor måtte i så fall en node med verdien 2,5 plasseres for at treet forsatt skal være et binært søketre?
6. Tegn alle de forskjellige binære søketrærne vi kan få som kun inneholder verdiene A, B og C (med vanlig alfabetisk sortering).
7. Som *Oppgave* 6, men med verdiene A, B, C og D.
8. Som *Oppgave* 6, men med verdiene A, A, B og B, dvs. to A-er og to B-er.

5.2.2 Datastruktur for et binært søketre

En klasse `SBinTre` (*S* for søketre) kunne være en subklasse til `BinTre`. Men da måtte flere metoder i overskrives (eng: override) og noen blokkeres. Et bedre alternativ hadde vært å la de to ha en felles abstrakt superklasse `AbstraktBinTre`. Den kunne inneholde nodeklassen og de metodene som er like for de to, f.eks. alle traverseringsmetodene.

Men for å gjøre det enkelt lager vi en selvstendig klasse `SBinTre`. Metodene fra `BinTre` som trengs og som virker på samme måte i `SBinTre`, kan enkelt kopieres ved «klipp og lim».

```
import java.util.Comparator;

public class SBinTre<T> // implements Beholder<T>
{
  private static final class Node<T> // en indre nodeklasse
  {
    private T verdi;                  // nodens verdi
    private Node<T> venstre, høyre;   // venstre og høyre barn

    private Node(T verdi, Node<T> v, Node<T> h)  // konstruktør
    {
      this.verdi = verdi;
      venstre = v;
      høyre = h;
    }

    private Node(T verdi)  // konstruktør
    {
      this(verdi, null, null);
    }
  } // class Node

  private Node<T> rot;                        // peker til rotnoden
  private int antall;                         // antall noder
  private final Comparator<? super T> comp;   // komparator

  public SBinTre(Comparator<? super T> c)    // konstruktør
  {
    rot = null;
    antall = 0;
    comp = c;
  }

  public int antall()        // antall verdier i treet
  {
    return antall;
  }

  public boolean tom()       // er treet tomt?
  {
    return antall == 0;
  }

} // class SBinTre
```

Programkode 5.2.2 a)

I *Programkode* 5.2.2 *a)* er det foreløpig bare én konstruktør og den krever en komparator som parameterverdi. Komparatoren må virke for den aktuelle datatypen. Hvis vi f.eks. ønsker å bruke `Integer` som datatype, kan vi bruke en «naturlig» komparator. Da kan en instans av `SBinTre`-klassen lages på denne måten:

```
SBinTre<Integer> tre = new SBinTre<>(Comparator.naturalOrder());
System.out.println(tre.antall() + " " + tre.tom());
// Utskrift: 0 true
```

Programkode* 5.2.2 *b)

Det kan også være gunstig å ha *konstruksjonsmetoder* i klassen `SBinTre`, dvs. statiske metoder som returnerer instanser av klassen:

```
public static <T extends Comparable<? super T>> SBinTre<T> sbintre()
{
  return new SBinTre<>(Comparator.naturalOrder());
}

public static <T> SBinTre<T> sbintre(Comparator<? super T> c)
{
  return new SBinTre<>(c);
}
```

Programkode* 5.2.2 *c)

Flg. eksempel viser hvordan instanser av `SBinTre`-klassen, med `Integer` som typeparameter, kan lages ved hjelp av konstruksjonsmetodene:

```
SBinTre<Integer> tre1 = SBinTre.sbintre();          // 1. konstruksjonsmetode

Comparator<Integer> c = Comparator.naturalOrder();
SBinTre<Integer> tre2 = SBinTre.sbintre(c);         // 2. konstruksjonsmetode

System.out.println(tre1.antall() + " " + tre2.antall());

// Utskrift: 0 0
```

Programkode* 5.2.2 *d)

Vi kan også lage en komparator ved hjelp av et lambda-uttrykk. I flg. tre vil tegnstrenger bli ordnet etter lengde:

```
SBinTre<String> tre3 = new SBinTre<>((s,t) -> s.length() - t.length());
```

Programkode* 5.2.2 *e)

I `SBinTre` står det `// implements Beholder<T>`, dvs. setningen er kommentert vekk. Vi skal implementere alle metodene i `Beholder`, men inntil videre lar vi kommentartegnet stå. Vi skal lage én metode om gangen og teste den før vi går videre.

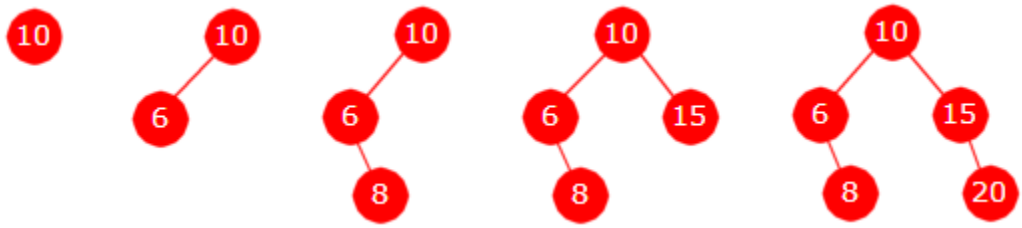
Oppgaver til Avsnitt 5.2.2

1. Filen `SBinTre` inneholder en foreløpig versjon av klassen, dvs. alt det som er diskutert i *Avsnitt* 5.2.2. Legg den over til deg. Sjekk at *Programkode* 5.2.2 *d)* virker for deg. Sjekk så at koden også virker hvis du bytter ut datatypen `Integer` med en annen type som er «sammelignbar med seg selv» (Java: Comparable). Prøv f.eks. med `Character`, `String`, `Boolean` og `Person`.

5.2.3 Innlegging av verdier

En ny verdi må legges på en plass som gjør at treet fortsetter å være et binært søketre, dvs. at det fortsatt oppfyller *Definisjon* 5.2.1.

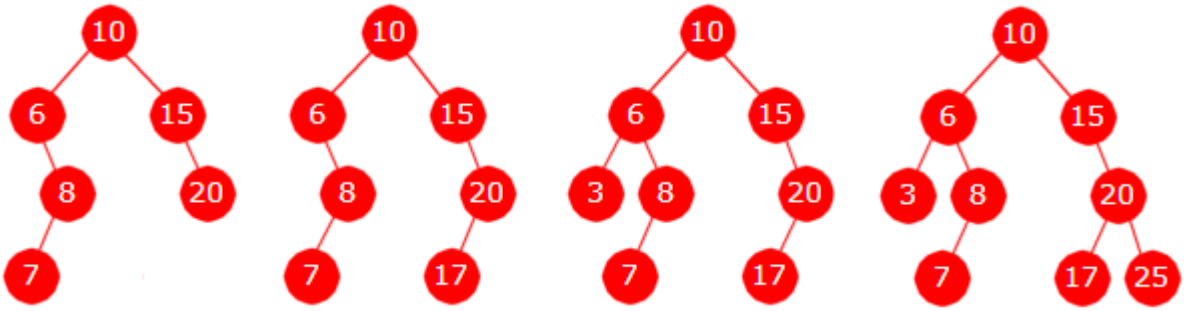
I flg. eksempel skal vi legge inn disse verdiene i et på forhånd tomt tre: 10, 6, 8, 15, 20, 7, 17, 3, 25. Den første blir alltid rotnodeverdi. Så kommer 6. Den er mindre enn 10 og må da legges til venstre for rotnoden. Verdien 8 er også mindre enn 10 og må legges til venstre for roten. Men den må ligge til høyre for 6 siden 8 er større enn 6. Verdien 15 er større enn 10 og må legges til høyre for roten. Verdien 20 er også større enn 10 og må legges til høyre for roten. Men siden 20 er større enn 15 må den også legges til høyre for 15. Se *Figur* 5.2.3 *a)*:



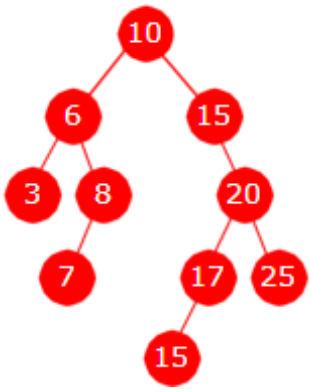
Figur 5.2.3 a) Verdiene 10, 6, 8, 15 og 20 er fortløpende lagt inn

Innleggingsalgoritmen: Gitt at *verdi* skal legges inn. Start i rotnoden. Hvis *verdi* er mindre enn nodeverdien, går vi til venstre og ellers til høyre. Dette gjentas for hver node inntil vi havner «utenfor treet». Dvs. vi skal videre til venstre eller høyre, men der er det ingen node. Der skal det legges inn en ny node med *verdi* som nodeverdi. Den blir da barn til den siste noden vi passerte: Venstre barn hvis siste sammenligning var «mindre» og ellers høyre barn.

Neste verdi er 7 som er mindre enn 10. Dermed skal vi til venstre for roten. Videre er 7 større enn 6, dvs. til høyre. Så til venstre siden 7 er mindre enn 8, men der stopper treet. Dvs. 7 blir venstre barn til 8. Det blir tilsvarende med 17, 3 og 25. Se *Figur* 5.2.3 *b)*:



Figur 5.2.3 b) Innleggingen fortsetter med verdiene 7, 17, 3 og 25



Figur 5.2.3 c)

Duplikater: En verdi som skal legges inn og som allerede er i treet, kalles et *duplikat*. I et binært søketre av vår type er det generelt tillatt med duplikater.

Gitt at vi skal legge inn 15 i treet lengst til høyre i *Figur* 5.2.3 *b)* over. Den finnes fra før. Med andre ord et duplikat. Algoritmen er som vanlig: Til venstre hvis *verdi* er mindre enn verdien i noden og ellers til høyre (dvs. hvis *verdi* er større enn eller lik verdien i noden). Vi starter i rotnoden. Deretter blir det høyre (15 > 10), høyre (15 ≥ 15), venstre (15 < 20) og venstre (15 < 17). På *Figur* 5.2.2 *c)* til venstre ser vi at 15 er lagt inn som venstre barn til 17.

Generelt: En duplikatverdi vil havne lengst til venstre i det høyre subtreet til den siste noden med samme verdi som vi passerer. I *Figur* 5.2.3 *c)* ligger siste (og første) forekomst av 15 i rotnodens høyre barn. Neste forekomst vil derfor havne lengst til venstre i denne nodens høyre subtre og vi ser at det er nettopp det som har skjedd.

```
public final boolean leggInn(T verdi)    // skal ligge i class SBinTre
{
  Objects.requireNonNull(verdi, "Ulovlig med nullverdier!");

  Node<T> p = rot, q = null;              // p starter i roten
  int cmp = 0;                            // hjelpevariabel

  while (p != null)       // fortsetter til p er ute av treet
  {
    q = p;                                // q er forelder til p
    cmp = comp.compare(verdi,p.verdi);    // bruker komparatoren
    p = cmp < 0 ? p.venstre : p.høyre;    // flytter p
  }

  // p er nå null, dvs. ute av treet, q er den siste vi passerte

  p = new Node<>(verdi);                  // oppretter en ny node

  if (q == null) rot = p;                 // p blir rotnode
  else if (cmp < 0) q.venstre = p;        // venstre barn til q
  else q.høyre = p;                       // høyre barn til q

  antall++;                               // én verdi mer i treet
  return true;                            // vellykket innlegging
}
```

Programkode* 5.2.3 *a)

I *Programkode* 5.2.3 *a)* brukes to nodereferanser *p* og *q*. Referansen *p* starter i rotnoden. Den flyttes så nedover i treet - til venstre når *verdi* er mindre enn nodeverdien og til høyre ellers. Sammenligningene utføres ved hjelp av *compare*-metoden til komparatoren *comp*. Referansen *q* skal ligge et nivå over *p*, dvs. være forelder til *p*. Når *p* blir *null*, vil *q* være den siste noden som ble passert. Dermed skal *verdi* legges inn som et barn til *q*. Den siste verdien som *compare*-metoden returnerte, forteller om det skal være venstre eller høyre barn. Hvis treet i utgangspunktet var tomt, lages en rotnode.

Eksempel: Hvordan kan vi lage trærne i *Figur* 5.2.1 *a)*? I bokstavtreet må *I* legges inn først siden første verdi alltid blir rotverdi. Hvis *E* legges inn etter *I* vil *E* bli venstre barn til *I*. Hvis vi fortsetter med *A* vil den bli venstre barn til *E*. Vi kan rett og slett legge inn verdiene i det som svarer til preorden. Det er også mulig å legge inn i nivåorden - se *Oppgave* 2. Hvis *leggInn*-metoden er tatt inn i SBinTre, vil flg. kode virke:

```
SBinTre<Character> tre = SBinTre.sbintre();       // et tomt tre
char[] verdi = "IEACBDGFHOMKJLN".toCharArray();   // verdiene i preorden

for (char t : verdi) tre.leggInn(t);              // venstre tre i Figur 5.2.1 a)
System.out.println(tre.antall());                 // Utskrift: 15
```

Programkode* 5.2.3 *b)

Det andre treet fra *Figur* 5.2.1 *a)* kan lages på en tilsvarende måte - se *Oppgave* 3.

I *Programkode* 5.2.3 *b)* får vi kun sjekket at metoden *antall*() returnerer korrekt verdi. Det hadde også vært gunstig å få sjekket treets høyde. Et binært søketre er et binærtre. Det betyr at vi kan bruke de fleste metodene vi laget for et vanlig binærtre i *Kapittel* 5.1 og da spesielt metoden *høyde*() i *Programkode* 5.1.12 *b)*. Kopier den inn i klassen SBinTre og legg inn flg. kodelinje i *Programkode* 5.2.3 *b)*. Hva blir utskriften?

```
System.out.println(tre.høyde());
```

Det vil ofte være slik som i *Programkode* 5.2.3 *b)*, at vi bygger opp et tre ved å hente én og én verdi fra en eller annen datastruktur, f.eks. en tabell eller en subtype til Collection (f.eks. en liste). En konstruksjonsmetode kan gjøre jobben for oss. Det mest generelle er da å la en Stream gå inn som parameter siden de andre strukturene kan gjøres om til en Stream. Da vil dens metode *forEach*(Consumer<? **super** T> *action*) gi oss verdiene.

Vi lager to konstruksjonsmetoder. I den første inngår en eksplisitt komparator *c* for typen *T*, mens den andre bruker naturlig ordning. Siden *leggInn*() er konstant/final, kan den brukes i en slik metode. Skal dette virke må vi ha **import** java.util.stream.Stream; øverst:

```
public static <T> SBinTre<T> sbintre(Stream<T> s, Comparator<? super T> c)
{
  SBinTre<T> tre = new SBinTre<>(c);              // komparatoren c
  s.forEach(tre::leggInn);                        // bygger opp treet
  return tre;                                     // treet returneres
}

public static <T extends Comparable<? super T>> SBinTre<T> sbintre(Stream<T> s)
{
  return sbintre(s, Comparator.naturalOrder());  // naturlig ordning
}
```

Programkode* 5.2.3 *c)

Metodene over kan brukes slik:

```
Integer[] a = {8,2,10,1,6,9,14,4,7,12,15,3,5,11,13};      // en tabell

SBinTre<Integer> tre1 = SBinTre.sbintre(Stream.of(a));    // et binært søketre

Comparator<Integer> c = Comparator.reverseOrder();        // omvendtkomparator
SBinTre<Integer> tre2 = SBinTre.sbintre(Stream.of(a),c);  // speilvendt tre
```

Programkode* 5.2.3 *d)

Vi bør nå kode metoden *toString* i klassen SBinTre. For et binært søketre er det best å få verdiene i inorden siden det gir oss sortert rekkefølge. En slik metode har vi allerede laget i klassen BinTre (se *Programkode* 5.1.7 *f*) og den kan vi kopiere inn i vår klasse SBinTre. Hvis du har gjort det, kan flg. kode legges inn til slutt i *Programkode* 5.2.3 *d)* og utføres:

```
System.out.println(tre1 + "\n" + tre2);

// Utskrift:
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
// [15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
```

Programkode* 5.2.3 *e)

Oppgaver til Avsnitt 5.2.3

1. Ta utgangspunkt i treet i *Figur 5.2.3 c)*. Legg så inn verdiene 5, 13, 16, 10, 12, 15.
2. Legg inn fortløpende de følgende gitte verdiene i et på forhånd tomt tre. Tegn treet.

   a) H, J, C, F, D, M, A, I, E, K, G, L, B *b)* E, H, B, E, G, F, D, I, H, A, E, C

   c) 4, 1, 8, 5, 3, 10, 7, 2, 6, 9 *d)* 9, 4, 17, 12, 15, 1, 8, 10, 2, 5, 4, 20, 11, 6, 16, 9

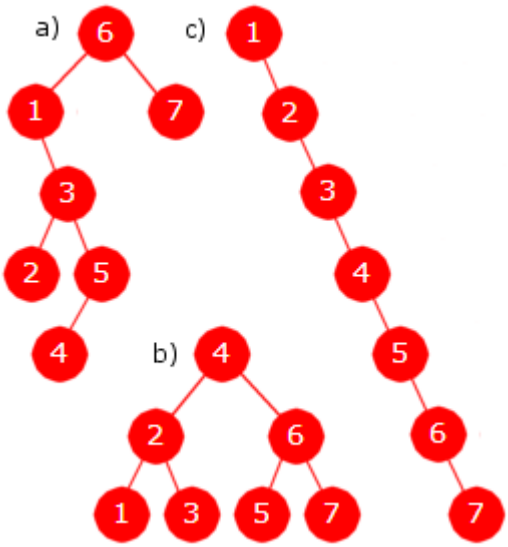
   e) Sohil, Per, Thanh, Fatima, Kari, Jasmin

   f) 10, 5, 20, 10, 3, 8, 13, 18, 7, 5, 6, 12, 4, 9, 11, 10, 22
3. Filen `SBinTre` inneholder det som ble diskutert og laget i *Avsnitt 5.2.2* og *Avsnitt 5.2.3*. Bruk den hvis du ikke allerede har bygget opp klassen. Lag så et program som bygger opp det venstre treet i *Figur 5.2.1 a)* ved å legge inn verdiene i nivåorden. Sjekk så resultatet ved å bruke metodene `antall()`, `høyde()` og `toString()`. Lag så kode som bygger opp det høyre treet i *Figur 5.2.1 a)*.
4. Lag kode som bygger trærne fra *Oppgave* 2.
5. I *Programkode 5.2.3 d)* brukes en statisk metode fra grensesnittet `Stream` til å lage en strøm av en tabell. Der er det flere andre statiske metoder (konstruksjonsmetoder) som lager en strøm. Sjekk hva grensesnittet inneholder! Klassen `Arrays` har også slike metoder. Sjekk dem!
6. Lag en `int`-tabell som inneholder en tilfeldig permutasjon av tallene fra 1 til n der n er 100. Lag så et binært søketre ved å legge inn ett og ett tall. Skriv ut treets høyde. Kjør programmet flere ganger. La så n være 1000 og 10000. Får trærne stor høyde? Sammenlign med tallet $\log_2(n)$.
7. Metoden *`toString()`* skal kodes slik at det blir komma og mellomrom mellom hver verdi. Men ikke hvis det er kun én verdi. En `StringJoiner` ordner opp i det for oss. Bruk en `StringJoiner` i kodingen av *`toString()`* i *Programkode 5.2.3 e)*. Gå til kildekoden for `StringJoiner` og se hvordan det foregår.

5.2.4 Gjennomsnittlig nodedybde

Hvis vi skal bygge opp et binært søketre ved hjelp av en samling verdier, vil treets form bli bestemt av den rekkefølgen verdiene legges inn. Anta at vi skal lage trær som inneholder tallene fra 1 til 7. Se på flg. tre rekkefølger (eller permutasjoner):

a) 6 1 3 7 5 2 4 *b)* 4 2 3 6 1 5 7 *c)* 1 2 3 4 5 6 7



Figur 5.2.4 a) : 3 binære søketrær

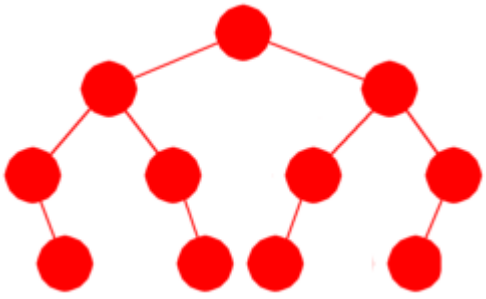
Figur 5.2.4 *a)* viser hvilke trær vi får når verdiene settes inn i den gitte rekkefølgen. Den første *a)* gir et noe skjevt tre, den neste *b)* gir et perfekt tre og den siste *c)* gir et ekstremt skjevt tre.

Tallene fra 1 til 7 kan permuteres på 7! = 5040 forskjellige måter og hver gir et tre. Nå blir mange av dem like. En formel (se *Avsnitt* 5.1.2) sier at for n = 7 blir det 429 forskjellige trær.

Det er effektivt å søke etter tall som har kort avstand fra roten. Derfor er treet i *b)* åpenbart best og *c)* dårligst. Hver av de 5040 permutasjonene gir som sagt et tre. Spørsmålet er hvor langt fra roten en node i gjennomsnitt ligger i et binært søketre.

Avstanden fra en node til roten kalles nodens *dybde*. Summen av nodedybdene blir treets *indre veilengde*.

Den *gjennomsnittlige nodedybden* for et binært søketre med n noder er lik forholdet mellom den indre veilengden og antallet n. *Perfekte* binærtrær har minst gjennomsnittlig nodedybde. Den er tilnærmet lik (n stor) $\log_2(n+1) - 2$.



Figur 5.2.4 b) : Balansert tre

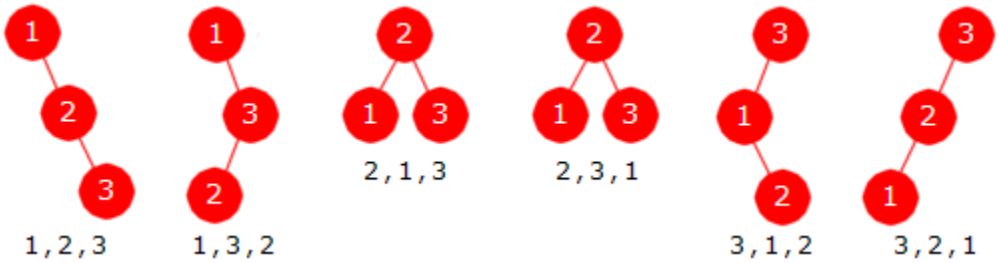
Hvis antallet noder n ikke er på formen $2^k - 1$, er et *balansert* tre det beste vi kan oppnå. Det betyr at alle nivåene i treet, unntatt det nederste (siste), er fulle av noder. Antallet noder på det nederste nivået kan være fra én og oppover. *Figur* 5.2.4 *b)* til venstre viser et balansert tre med 11 noder. Nederste nivå inneholder 4 noder, men har plass til 8. Obs: Et balansert tre er ikke det samme som et *komplett* tre. I et *komplett* tre må nederste nivå være fylt opp fra venstre.

Formelen $\log_2(n+1) - 2$ gir også en god tilnærming for gjennomsnittlig nodedybde i et balansert tre. Spørsmålet nå er hvor mye «dårligere» et tilfeldig binært søketre er enn et balansert tre?

Hver permutasjon av n forskjellige verdier gir et binært søketre. Den *gjennomsnittlige nodedybden* for binære søketrær med n forskjellige verdier er gjennomsnittet over alle de $n!$ (n fakultet) forskjellige permutasjonene gitt at de alle er like sannsynlige.

Eksempel 1: I *Figur* 5.2.4 *a)* viser tre trær med verdiene 1, 2, . . , 7. Det første (*a*) har en gjennomsnittlig nodedybde på 14/7 = 2, det andre (*b*) 10/7 ≈ 1,4 og det tredje (*c*) 21/7 = 3. Gjenomsnittlig nodedybde for disse tre trærne blir (2 + 10/7 + 3)/3 = 15/7. Men hva vil gjennomsnittet bli for alle de 7! = 5040 trærne?

Eksempel 2: Tallene 1, 2 og 3 kan permuteres på 3! = 6 måter. Hver permutasjon gir oss et tre. Tilsammen 6 trær. De to permutasjone 2, 1, 3 og 2, 3, 1 gir samme tre:



Figur 5.2.4 c) : De 6 pemutasjonene av 1, 2 og 3 gir disse 6 binære søketrærne.

Den indre veilengden for de seks trærne er henholdsvis 3, 3, 2, 2, 3, 3 med sum lik 16. For å finne gjennomsnittlig nodedybde må vi først dele summen med 3 siden det er 3 noder og så dele med 6 siden det er 6 trær. Svaret blir 16/18 = 8/9.

Det finnes en formel (den utledes i *Avsnitt* 5.2.15) for den gjennomsnittlige nodedybden D_n for binære søketrær med n noder. Den ser slik ut:

(5.2.4 a) $$D_n = 2(1+\frac{1}{n})H_n - 4$$

H_n er det n-te harmoniske tallet, dvs. summen av de inverse heltallene fra 1 til n. Hvis n er stor, vil $1/n$ ha liten effekt. Videre vil (se *Avsnitt* 1.1.6) H_n være tilnærmet lik $\log(n) + 0{,}577$ der *log* er den naturlige logaritmen (grunntall e). Formelen $\log(n) = \log(2) \cdot \log_2(n)$ der $\log(2) = 0{,}693$, gjør om til grunntall 2. Det gir flg. formel for gjennomsnittlig nodedybde:

(5.2.4 b) $$D_n \approx 2H_n - 4 \approx 1{,}386 \log_2(n) - 2{,}846$$

Det «beste» binære søketreet vi kan oppnå er som tidligere nevnt, et balansert tre. Et slikt tre har $\log_2(n+1) - 2$ som gjennomsnittlig nodedybde. Resultatet over sier at binære søketrær er i gjennomsnitt kun ca. 38 prosent dårligere enn dette. Problemet er imidlertid at når vi bygger opp et binært søketre har vi normalt ingen garanti for at verdiene legges inn i en tilfeldig rekkefølge. I praktiske anvendelser er det heller slik at verdiene kan være delvis sortert og da blir treet skjevt. Det meste ekstreme er trær der ingen noder har to barn. Det får vi f.eks. når verdiene legges inn i sortert rekkefølge. I et slikt tre (med n noder) er gjennomsnittlig nodedybde lik $(n - 1)/2$.

Gjennomsnittlig nodedybde for binære søketrær uten duplikatverdier			
	Balansert tre	Gjennomsnitt for alle binære søketrær	Ekstremt skjevt tre
n noder	$\log_2(n+1) - 2$	$1{,}386 \log_2(n) - 2{,}846$	$(n - 1)/2$
n = 100	4,7	6,4	49,5
n = 1000	8,0	11,0	499,5
n = 100.000	14,6	20,2	49999,5

Figur 5.2.4 d) : Gjennomsnittlig nodedybde for binære søketrær med n noder

Den gjennomsnittlige nodedybden i et binært søketre med n noder er av logaritmisk orden både i det beste tilfellet og i gjennomsnitt, men av lineær orden i det verste tilfellet.

Obs: Analysen over forutsetter at vi har binære søketrær uten duplikatverdier. Hvis det er like verdier, kan det bli helt annerledes. Det mest ekstreme er at alle verdiene er like. Da vil treet bli ekstremt høyreskjevt, dvs. ingen noder har venstre barn. Hvis treet har n verdier og alle er like, vil gjennomsnittlig nodedybde bli $(n - 1)/2$.

Et annet interessant aspekt ved gjennomsnittlige binære søketrær er antallet noder av ulike typer. Hva er f.eks. det gjennomsnittlige antallet bladnoder (noder uten barn), antallet noder med ett barn og antallet med to barn?

La B_n være det gjennomsnittlige antallet bladnoder i binære søketrær med n (forskjellige) verdier, E_n det gjennomsnittlige antallet som har nøyaktig ett barn og til slutt T_n det gjennomsnittlige antallet som har to barn. En node hører til nøyaktig én av disse tre kategoriene. Dermed $B_n + E_n + T_n = n$. Da gjelder flg. formler:

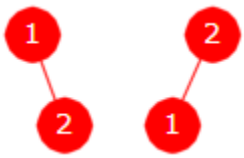
$$(5.2.4.c) \qquad B_0 = 0,\ B_1 = 1,\ B_n = (n + 1)/3\ ,\quad n > 1$$

$$(5.2.4.d) \qquad E_0 = 0,\ E_1 = 0,\ E_n = (n + 1)/3\ ,\quad n > 1$$

$$(5.2.4.e) \qquad T_0 = 0,\ T_1 = 0,\ T_n = (n - 2)/3\ ,\quad n > 1$$

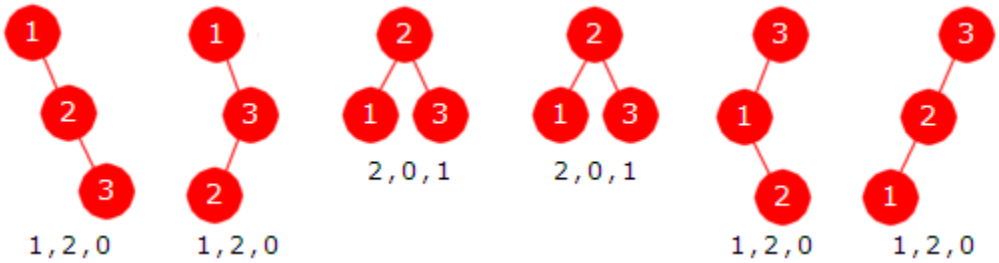
Formel (5.2.4.d) gir det gjennomsnittlige antallet som har nøyaktig ett barn. Et binærtre kan speilvendes. Det betyr at det er like mange noder med kun et høyre barn som noder med kun et venstre barn, dvs. $(n + 1)/6$ for hver av dem.

Formlene for B_n, E_n og T_n kan utledes - se *Oppgave* 13. Her nøyer vi oss med å sjekke at de stemmer for tilfellene $n = 2$ og $n = 3$. Hvis $n = 2$, blir det to trær:



Figur 5.2.4 e)

Hvert av trærne i *Figur* 5.2.4 *e)* har én bladnode. Dermed blir også gjennomsnittet lik 1, dvs. $B_2 = 1$. Vi får samme svar ved å sette $n = 2$ i formelen $B_n = (n + 1)/3$. Det finnes ingen noder med to barn og én node med ett barn i hvert av trærne. Dermed $E_2 = 1$ og $T_2 = 0$.



Figur 5.2.4 f) : De seks trærne med tre noder

Under hvert av seks trærne i *Figur* 5.2.4 *f)* er det satt opp tre tall. Det første står for antallet bladnoder, det andre for antallet noder med kun et barn og det tredje tallet for antallet noder med to barn. Legger vi sammen antallet bladnoder får vi summen 1 + 1 + 2 + 2 + 1 + 1 = 8 og dermed 8/6 = 4/3 som gjennomsnitt. Vi får det samme ved å sette $n = 3$ i formelen $B_n = (n + 1)/3$. Tilsvarende finner vi at også formlene for E_3 og T_3 stemmer.

Oppgaver til Avsnitt 5.2.4

1. I *Eksempel* 2 vises det at gjennomsnittlig nodedybde for $n = 3$ er 8/9. Sjekk at du får samme svar ved å sette $n = 3$ i *Formel* 5.2.4 *a)*.
2. Tegn de 24 trærne du får ved å bruke de 24 permutasjonene av tallene 1,2,3,4. Flere av dem blir like. Regn så ut gjennomsnittlig nodedybde for de 24 trærne. Sjekk at du får samme svar ved å sette $n = 4$ i *Formel* 5.2.4 *a)*.
3. Lag metoden `public static <T> int indreVeilengde(SBinTre<T> tre)` i klassen `SBinTre`. Den skal returnere treets indre veilengde, dvs. summen av nodedybdene til nodene. Vi definerer at et tomt tre har 0 som indre veilengde. Det betyr at både et tomt tre og et med kune én node har indre veilengde lik 0. Lag metoden på en av flg. måter:

   a) Bruk rekursjon. For hver node registerer nodens dybde ved hjelp av en parameter og den addereres til de tidligere dybdene ved hjelp av en tabellparameter.

   b) Hvis vi kjenner antallet noder og indre veilengde til hvert av rotnodens to subtrær, så blir indre veilengde til hele treet lik summen av de to subtrærnes antall og indre veilengder. Lag en statisk rekursiv hjelpemetode som traverserer treet og returnerer en int-tabell med to elementer. Første element (nr. 0) skal inneholde indre veilengde og andre element (nr. 1) skal inneholde antallet noder.
4. En indre veilengde kan bli et svært stort tall og i verste fall for stort for datatypen *int*. Et alternativ er da å bruke typen *long* i koden. Hva er det minste antallet noder et tre kan ha for at det i verste tilfellet får en indre veilengde som er for stor for datatypen *int*? Hint: Lag et ekstremt skjevt tre med n noder og finn dets indre veilengde.
5. Lag en metode `public static <T> double gjNodedybde(SBinTre<T> tre)` i klassen `SBinTre`. Den skal returnere treets gjenomsnittlige nodedybde. Gjennomsnittlig nodedybde er ikke definert for et tomt tre.
6. Lag binære søketrær ved å legge inn permutasjoner av tallene fra 1 til n med n = 100, 1000 og 100.000. Finn gjennomsnittlig nodedybde - se *Oppgave* 5. og sammenlign resulatet med det teoretiske gjennomsnittet i *Figur* 5.2.4 *d)*.
7. Lag metoden `public static double gjNodedybde(int n)`. Den skal ta alle de $n!$ permutasjonene av tallene fra 1 til n og for hver permutasjon bygge opp et binært søketre med tallene i permutasjonen som verdier og summere de indre veilengdene. La summen være et long-tall. Deretter skal summen først deles med n siden det er n verdier og deretter med $n!$ (n-fakultet) siden det er $n!$ trær. Resultatet skal returneres. Pass på at de siste utregningene skjer via desimaltall (double). Til å generer alle permutasjonene kan du f.eks. bruke *Programkode* 1.3.3. *a)*. Se også *Oppgave* 8.
8. Lag metoden `public static double D(int n)`. Den skal returnere verdien D_n definert i *Formel* 5.2.4 *a)*. Pass på at utregningene skjer ved hjelp av desimaltall (double). Hvis du ikke allerede har klassen *Matte* under *hjelpeklasser*, bør du legge den inn hos deg. Der finner du blant annet en metode som finner $n!$ (n-fakultet) og en som finner det n-te harmoniske tallet (H_n). Se også *Oppgave* 9.
9. Sjekk at de to metodene fra *Oppgave* 8 og 9 gir samme svar f.eks. for n fra 1 til 10.
10. Sjekk at formlene (5.2.4.c) - (5.2.4.e) stemmer for $n = 4$. Se trærne i *Oppgave* 2.
11. Finn største og minste antall noder med ingen, ett og to barn i et tre med n noder?
12. Lag en metode som finner antallet noder av ulike typer i et gitt binærtre, dvs. antallet bladnoder, ettbarnsnoder og tobarnsnoder.
13. Vis at formlene (5.2.4.c) - (5.2.4.e) stemmer for alle n. Start med å finne B_n. Hint: Antall bladnoder er lik summen av antallene i rotnodens to subtrær.

5.2.5 Fra ordnet tabell til binært søketre

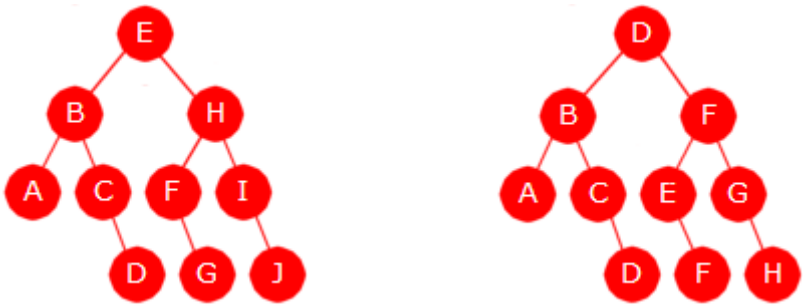
I *Avsnitt* 5.2.3 handler om hvordan én og én verdi kan legges inn i et binært søketre slik at treet bevares som et binært søketre. Da er det en kjensgjerning at treets form er helt avhengig av den rekkefølgen verdiene legges inn. I verste fall kan treet bli ekstremt skjevt.

I noen tilfeller vil det være aktuelt å bygge opp et binært søketre ved hjelp av verdier som allerede er sortert. For at treet da ikke skal bli skjevt, må verdiene legges inn i en annen rekkefølge. Tabellen under inneholder bokstavene fra *A* til *J* i sortert rekkefølge:

A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9

Figur 5.2.5 a) : En sortert tabell

La verdien på midten av tabellen (*E* i posisjon (0 + 9)/2 = 4) bli verdien i roten, midtverdien i intervallet til venstre for *E* (*B* i posisjon (0 + 3)/2 = 1) bli verdien i venstre barn til roten og midtverdien i intervallet til høyre for *E* (*H* i posisjon (5 + 9)/2 = 7) bli verdien i høyre barn til roten. Osv. Husk at midten på et intervall med endepunkter *v* og *h* er lik (*v* + *h*)/2 (heltallsdivisjon). Dette vil gi oss det binære søketreet til venstre i figuren under:



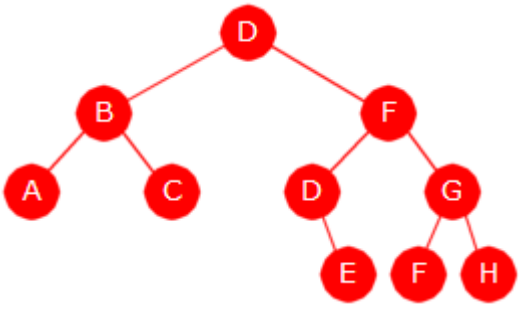
Figur 5.2.5 b) : Treet til venstre er et binært søketre, men ikke det til høyre

Tabellen i *Figur* 5.2.5 *a)* er sortert og uten duplikater. Den under er sortert og har duplikater:

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

Figur 5.2.5 c) : En sortert tabell med like verdier

Idéen over brukt på tabellen i *Figur* 5.2.5 *c)* gir treet til høyre i *Figur* 5.2.5 *b)*. Verdiene er sortert i inorden, men det er ikke et binært søketre. *Definisjon* 5.2.1 sier at for hver node skal alle verdiene i nodens venstre subtre være mindre enn verdien i noden. Men det stemmer ikke her. F.eks. er det en *D* i det venstre subtreet til rotnoden. Men *D* er ikke mindre enn *D*.



Figur 5.2.5 d) : Et binært søketre

Idéen må finpusses. Hvis midtverdien har duplikater til venstre for seg, tar vi den av dem som ligger lengst til venstre. Tabellen i *Figur* 5.2.5 *c)* har *D* som midtverdi. Men den har en *D* til venstre for seg, men ikke flere. Da velger vi den (posisjon 3) som rotnode. I midten av intervallet til venstre ligger *B*. Den blir venstre barn til rotnoden. I midten av intervallet til høyre (dvs. posisjon 6) ligger den første av to *F*-er. Den blir høyre barn til rotnoden. Osv. Vi får treet i *Figur* 5.2.5 *d)* til venstre.

Gitt et intervall av typen *a*[*v*:*h*]. Hvis det er tomt, får vi et tomt tre (dvs. null). Hvis ikke, har intervallet et midtpunkt og en midtverdi. Da returneres en node med midtverdien (eventuelt en justert midtverdi hvis det er duplikater) som nodeverdi. Nodens to barn er det vi får ved å fortsette med venstre og høyre delintervall. Metoden *equals*() brukes for å lete mot venstre etter duplikater. Da er det en forutsetning at den er konsistent med datatypens ordning:

```
private static <T> Node<T> balansert(T[] a, int v, int h)  // en rekursiv metode
{
  if (v > h) return null;                          // tomt intervall -> tomt tre

  int m = (v + h)/2;                               // midten
  T verdi = a[m];                                  // midtverdien

  while (v < m && verdi.equals(a[m-1])) m--;       // til venstre

  Node<T> p = balansert(a, v, m - 1);              // venstre subtre
  Node<T> q = balansert(a, m + 1, h);              // høyre subtre

  return new Node<>(verdi, p, q);                  // rotnoden
}
```

Programkode 5.2.5 a)

Gir metoden over et binært søketre? Det er en forutsetning at tabellen *a* er sortert. Hvis ikke, er resultatet uforutsigbart. Vi starter på midten av *a*[*v*:*h*] og går mot venstre hvis det er like verdier. De til venstre for den i posisjon *m* havner i venstre subtre og de til høyre i høyre subtre. Dermed oppfylles *Definisjon* 5.2.1. Det er også mulig å lage en metode som sjekker at definisjonen er oppfylt. Se *Oppgave* 9.

Vi lager to konstruksjonsmetoder som begge lager et binært søketre ved hjelp av en sortert tabell. Den bruker en spesifikk komparator og den andre at datatypen er sammenlignbar med seg selv (Java: comparable) Metodenavnet er *balansert*, dvs. det samme som den rekursive metoden. Det er kun når tabellen er uten duplikater at treet garantert blir balansert (lavest mulig høyde). Hvis det er mange duplikater blir treet skjevt. Det verste tilfellet er når alle verdiene er like. Se *Oppgave* 7. Men de blir så balanserte som det er mulig å få til. I *Kapittel* 9 skal vi se mer generelt på begrepet *balanserte* binære trær.

```
public static <T> SBinTre<T> balansert(T[] a, Comparator<? super T> c)
{
  SBinTre<T> tre = new SBinTre<>(c);          // oppretter et tomt tre
  tre.rot = balansert(a, 0, a.length - 1);    // bruker den rekursive metoden
  tre.antall = a.length;                      // setter antallet
  return tre;                                 // returnerer treet
}

public static <T extends Comparable<? super T>> SBinTre<T> balansert(T[] a)
{
  return balansert(a, Comparator.naturalOrder());
}
```

Programkode 5.2.5 b)

Flg. kodebit gir treet i *Figur* 5.2.5 *d)*:

```
SBinTre<String> tre = SBinTre.balansert("ABCDDEFFGH".split(""));
System.out.println(tre.antall() + " " + tre.høyde() + " " + tre);
```

Programkode 5.2.5 c)

Oppgaver til Avsnitt 5.2.5

1. Legg metoden *balansert()* og de to konstruksjonsmetodene i *Programkode* 5.2.5 *b)* inn i klassen `SBinTre`. Sjekk at *Programkode* 5.2.5 *c)* virker. Du må ha metoden *høyde()* i klassen `SBinTre`. Bytt ut med andre bokstaver! Pass på at tabellen alltid er sortert.
2. Som i *Oppgave* 1, men la tabellen inneholde bokstavene fra *A* til *O* i sortert rekkefølge.
3. Som i *Oppgave* 1, men bruk en `Integer`-tabell med heltallene fra 1 til 31.
4. Tegn treet du får ved å bruke algoritmen som lager et balansert binært søketre ved hjelp av en sortert tabell (*Programkode* 5.2.5 *a*) når tabellen inneholder bokstavene fra *A* til *O* i sortert rekkefølge.
5. Som i *Oppgave* 4, men med en tabell med navnene Ali, Ann, Eli, Per, Siv, Tor, Ulf.
6. Som i *Oppgave* 4, men med en tabell som inneholder *A* fem ganger og så *B* fem ganger.
7. Som i *Oppgave* 4, men nå med en tabell som inneholder *A* ti ganger.
8. Lag metoden `public boolean erSortertInorden()`. Den skal returnere *true* hvis treet er sortert i stigende rekkefølge i inorden og *false* ellers. Det er kanskje enklest å bruke en iterativ inordentraversering.
9. Lag metoden `public boolean erSøketre()`. Den skal gi *true* hvis treet er et binært søketre. Bruk f.eks. flg. rekursive idé: Et binærtre er et søketre hvis begge subtrærne til rotnoden er søketrær, den største verdien i venstre subtre er mindre enn rotnodeverdien og rotnodeverdien er mindre enn eller lik den minste verdien i høyre subtre.
10. Lag metoden `private static <T> Node<T> random(T[] a, int v, int h, Random r)` slik som metoden `balansert` i *Programkode* 5.2.5 *a)*. Ikke med midtverdien som rotnodeverdi, men en tilfeldig verdi mellom *v* og *h*. Bruk randomgeneratoren *r*. Det forutsettes at tabellen *a* er sortert og at den kan ha like verdier. Lag så to konstruktørmetoder med navn *randomTre* slik som i *Programkode* 5.2.5 *b)*.

5.2.6 Søking etter en verdi

Ordningen i et binært søketre gjør at verdien i rotnoden deler treets verdier i to. I venstre subtre ligger de verdiene som er mindre og i høyre subtre de som er større enn eller lik rotverdien. Det betyr med andre ord at hvis den søkte verdien ikke ligger i roten, må den ligge i det ene av de to subtrærne. Men det subtreet som vi leter videre i, er også et binært søketre og det hele fortsetter til vi enten finner verdien eller til vi går ut av treet.

Det å søke i et binært søketre bygger på samme idé som binærsøk for sorterte tabeller. Se *Avsnitt* 1.3.6. I et gjennomsnittlig binært søketre er det mer sannsynlig at den søkte verdien ligger i et av rotnodens to subtrær enn i rotnoden selv. Dermed kan vi bruke samme kodeidé som i *Programkode* 1.3.6 *b)*:

```
public boolean inneholder(T verdi)     // skal ligge i klassen SBinTre
{
  if (verdi == null) return false;            // treet har ikke nullverdier

  Node<T> p = rot;                            // starter i roten
  while (p != null)                           // sjekker p
  {
    int cmp = comp.compare(verdi, p.verdi);   // sammenligner
    if (cmp < 0) p = p.venstre;               // går til venstre
    else if (cmp > 0) p = p.høyre;            // går til høyre
    else return true;                         // cmp == 0, funnet
  }
  return false;                               // ikke funnet
}
```

Programkode* 5.2.6 *a)

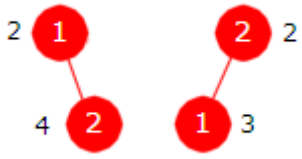
Hvor effektiv er denne algoritmen? Hvis vi har et gjennomsnittlig binært søketre med n forskjellige verdier, er sannsynligheten kun $1/n$ for at verdien vi søker etter ligger i rotnoden. Hvis n er stor, vil det dermed være ca. 50% sannsynlighet for at den ligger i rotnodens venstre subtre. I så fall (`cmp < 0`) holder det med én sammenligning i while-løkken. Hvis ikke, blir det to. Dermed 1,5 sammenligninger i gjennomsnitt. Hvis verdien vi søker etter ligger i treet, vil while-løkken i *Programkode* 5.2.6 *a)* bringe oss ned til rett node. Den gjennomsnittlige avstanden dit blir lik *gjennomsnittlig nodedybde* for binære søketrær. Dermed blir det gjennomsnittlige antallet sammenligninger A_n ca. 1,5 ganger gjennomsnittlig nodedybde. En eksakt analyse gir flg. verdi for A_n :

(5.2.6.a) $$A_n = 3\left(1+\frac{1}{n}\right)H_n - 4$$

Hvis n er forholdsvis stor, får vi:

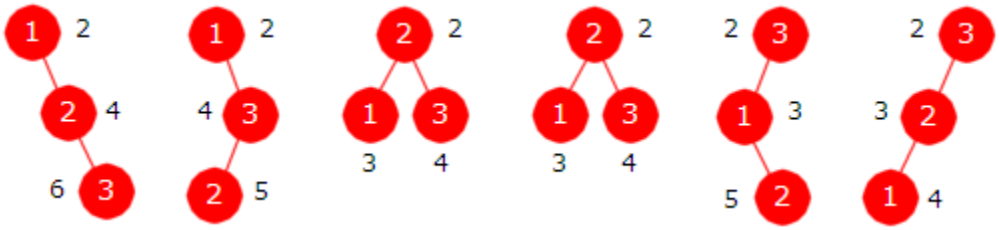
(5.2.6.b) $$A_n \approx 3H_n - 4 \approx 1{,}5 \cdot 1{,}386 \log_2(n) - 2{,}269$$

Formel 5.2.6 *a)* kan utledes - se *Avsnitt* 5.2.15. Her nøyer vi oss med å sjekke at den stemmer for tilfellene $n = 2$ og $n = 3$. Hvis $n = 2$, blir det to trær:



Figur 5.2.6 a)

I *Figur* 5.2.6 *a)* er det ved siden av hver node satt på det antallet sammenligninger som trengs for å finne verdien i noden. Det trengs to sammenligninger for å finne verdien i roten - en for å avgjøre at den søkte verdien ikke er mindre og så en til for å avgjøre at den heller ikke er større. I det venstre treet trengs det fire sammenligner for å finne tallet 2. Først to stykker for å avgjøre at den søkte verdien ikke ligger i roten og så to til for å avgjøre at den ligger i rotens høyre barn. I det høyre treet trengs det tre sammenligninger for å finne tallet 1. Først én for å avgjøre at den søkte verdien ligger til venstre for roten og så to til for å avgjøre at den ligger i rotens venstre barn. Summen blir 2 + 4 + 2 + 3 = 11. Gjennomsnittet finner vi ved først å dele med 2 siden det er to verdier og så med 2 igjen siden det er to trær. Det blir 11/4. *Formel* 5.2.6 *a)* gir $A_2 = 3\,(1 + 1/2)\,H_2 - 4 = 11/4$.



Figur 5.2.6 b) : De seks binære søketrærne med tre noder

Figur 5.2.6 *b)* viser de seks binære søketrærne med tre forskjellige verdier. Også der er det ved siden av hver node satt på det antallet sammenligninger som trengs for å finne verdien i noden. Summen blir 60. Gjennomsnittet finner vi ved først å dele med 3 siden det er tre verdier og så med 6 siden det er seks trær. Det blir 60/18 = 10/3. *Formel* 5.2.6 *a)* gir at $A_3 = 3\,(1 + 1/3)\,H_3 - 4 = (3 + 1)(1 + 1/2 + 1/3) - 4 = 2 + 4/3 = 10/3$.

Hva hvis den verdien vi søker etter ikke ligger treet? Da må vi tenke litt annerledes når det gjelder treets verdier. La treet nå inneholde tallene fra 1 til n som desimaltall, dvs. tallene 1.0 , 2.0 , 3.0 osv. Dermed kan vi søke etter verdier som ligger mellom to tall, f.eks. 1.5. Det ligger ikke i treet. Anta så at den verdien vi søker etter har samme sannsynlighet for å være mindre enn 1.0, være mellom 1.0 og 2.0, mellom 2.0 og 3.0, osv. oppover til å være større enn n. Da viser det seg (se *Avsnitt* 5.2.15) at det gjennomsnittlig antallet sammenligner for å avgjøre at verdien ikke ligger i treet, er gitt ved:

(5.2.6.c) $$A_n = 3H_{n+1} - 3 \approx 1{,}5 \cdot 1{,}386 \log_2(n+1) - 1{,}269$$

Formel 5.2.6 *c)* sier at det i gjennomsnitt trengs én sammenligning mer enn når den søkte verdien ligger i treet.

Gjennomsnittlig antall sammenligninger - søking i binære søketrær med n ulike verdier			
	Perfekt tre	Gjennomsnitt for alle binære søketrær	Ekstremt høyreskjevt tre
n noder	$1{,}5 \cdot \log_2(n+1) - 1$	$1{,}5 \cdot 1{,}386 \log_2(n) - 2{,}269$	$n + 1$
n = 100	9,0	11,6	1001
n = 1000	14,0	18,5	1.001
n = 100.000	23,9	32,3	100.001

Tabell 5.2.6 c) : Antall sammenligninger for søking i binære søketrær med n noder

I *Tabell* 5.2.6 *c)* står at at i et perfekt binært søketre er det gjennomsnittlige antallet sammenligninger gitt ved $1{,}5 \log_2(n+1) - 1$. Til å finne det kan en analysere på nøyaktig samme måte som for 2. versjon av binærsøk i en sortert tabell. Se *Avsnitt* 1.3.7.

Vi laget tre versjoner av binærsøk for sorterte tabeller - se *Avsnitt* 1.3.6. Den 3. versjonen hadde kun én sammenligning i hver iterasjon. Det kan vi også få til her ved å gå til venstre hvis den søkte verdien er mindre enn nodeverdien og til høyre ellers. Det oppstår imidlertid et problem hvis den søkte verdien ligger i noden vi forlater. Det kan vi imidlertid takle ved å ha en hjelpepeker som oppdateres når vi går til høyre.

```
public boolean inneholder(T verdi)          // ny versjon
{
  if (verdi == null) return false;          // treet har ikke nullverdier

  Node<T> p = rot;                          // starter i roten
  Node<T> q = null;                         // hjelpepeker

  while (p != null)                         // sjekker p
  {
    if (comp.compare(verdi, p.verdi) < 0)   // sammenligner
    {
      p = p.venstre;                        // går til venstre
    }
    else
    {
      q = p;                                // oppdaterer q
      p = p.høyre;                          // går til høyre
    }
  }

  return q == null ? false : comp.compare(verdi,q.verdi) == 0;
}
```

Programkode 5.2.6 *b)*

En kan sammenligne *Programkode* 5.2.6 *b)* og *Programkode* 5.2.6 *a)* ved å lage et stort binært søketre, søke fortløpende etter alle verdiene og så måle tiden. Se *Oppgave* 7.

Oppgaver til Avsnitt 5.2.6

1. Anta at en verdi forekommer flere ganger i treet. Hvilken av dem (i inorden) er det metoden i *Programkode* 5.2.6 *a)* finner?
2. Lag metoden `public int antall(T verdi)` i klassen `SBinTre`. Den skal returnere antallet forekomster av *verdi* og dermed 0 hvis *verdi* ikke er i treet.
3. Metoden `public Liste<T> intervallsøk(T fraverdi, T tilverdi)` skal returnere en TabellListe med verdiene fra og med *fraverdi* og til (men ikke med) *tilverdi*. Lag den.
4. Lag metoden *inneholder*() i *Programkode* 5.2.4 *a)* ved rekursjon. Dvs. lag en rekursiv metode som kalles av den offentlige metoden. Det har egentlig ingen hensikt å bruke rekursjon her. Men se på det som en øvelse i å lage rekursive algoritmer.
5. Vis at *Formel* 5.2.6 *a)* stemmer for $n = 4$. Bruk trærne fra *Oppgave* 2 i *Avsnitt* 5.2.4.
6. I *Tabell* 5.2.6 *c)* står det at gjennomsnittlig antall sammenligninger i et perfekt tre er tilnærmet gitt ved $1{,}5 \cdot \log_2(n+1) - 1$. Vis at det stemmer!
7. Bygg opp et binært søketre ved å legge inn verdiene fra en tilfeldig permutasjon av tallene fra 1 til 1000000. Søk så fortløpende etter alle tallene fra 1 til 1000000. Mål tiden det tar. Bruk metodene både i *Programkode* 5.2.6 *a)* og *Programkode* 5.2.6 *b)*. Hvem av dem er mest effektiv? Sammenlign også med den rekursive versjonen i *Oppgave* 3.

5.2.7 Min, maks, gulv, tak, mindre, større

Den minste verdien i et binært søketre er den som kommer først i inorden. Den finner vi ved å starte i roten og gå nedover mot venstre så langt det går. De nodene vi da «besøker» kalles forøvrig treets *venstre ryggrad* (eng: left backbone). Flg. metode finner den minste verdien:

```
public T min()                  // skal returnere treets minste verdi
{
  if (tom()) throw new NoSuchElementException("Treet er tomt!");

  Node<T> p = rot;                              // starter i roten
  while (p.venstre != null) p = p.venstre;      // går mot venstre
  return p.verdi;                               // den minste verdien
}
```

Programkode 5.2.7 a)

I et ekstremt venstreskjevt tre (ingen noder har høyre barn), vil venstre ryggrad utgjøre hele treet. I det tilfellet er metoden av orden n. I gjennomsnitt er den imidlertid av logaritmisk orden. Den gjennomsnittlige nodedybden til den første i inorden for binære søketrær med n forskjellige verdier er gitt ved $H_n - 1$. Se *Oppgave 7*.

Største verdi ligger lengst ned langs treets høyre ryggrad (eng: right backbone). En *maks*-metode blir derfor en «speilvending» av *Programkode* 5.2.7 *a)*. Hvis den største verdien forekommer flere ganger, er det imidlertid den siste av dem vi da finner. Se *Oppgave* 1-2.

I matematikk har vi begrepet *gulv* (eng: floor). Hvis x er et reelt tall, betyr gulv(x) = ⌊x⌋ avrundingen av x nedover til nærmeste heltall. Hvis *x* allerede er et heltall, så er *gulv(x)* lik x. Her skal vi isteden tenke oss at vi har en samling verdier. Da skal *gulv(x)* bety den største verdien i samlingen som er mindre enn eller lik x. Det betyr at hvis x er i samlingen, er gulv(x) lik x. Hvis derimot x er mindre enn alle verdiene, sier vi at *gulv(x)* er null.

Vi bruker flg. idé: Hvis x er mindre enn en nodeverdi, så vil *gulv(x)* (hvis den finnes) ligge i nodens venstre subtre. Hvis ikke, vil nodeverdien være en mulig kandidat for *gulv(x)*.

```
public T gulv(T verdi)
{
  Objects.requireNonNull(verdi, "Treet har ingen nullverdier!");
  if (tom()) throw new NoSuchElementException("Treet er tomt!");

  Node<T> p = rot; T gulv = null;

  while (p != null)
  {
    int cmp = comp.compare(verdi, p.verdi);

    if (cmp < 0) p = p.venstre;  // gulvet ligger til venstre
    else if (cmp > 0)
    {
      gulv = p.verdi;            // nodeverdien er en kandidat
      p = p.høyre;
    }
    else return p.verdi;         // verdi ligger i treet
  }
  return gulv;
}
```

Programkode 5.2.7 b)

Metoden i *Programkode* 5.2.7 *b)* er slik at hvis *verdi* ikke ligger i treet og det er flere verdier som oppfyller kravet til *gulv*(verdi), så er det den siste (i inorden) av dem som returneres. Hvis verdi er i treet, så er det den første av dem som returneres. Se *Oppgave* 3-4.

Flg. eksempel viser hvordan metoden skal virke:

```
Integer[] a = {5,10,3,8,13,7,16,2,6,11};

SBinTre<Integer> tre = SBinTre.sbintre(Stream.of(a));  // Programkode 5.2.3 c)

System.out.println(tre.gulv(10)); // Utskrift: 10
System.out.println(tre.gulv(9));  // Utskrift: 8
System.out.println(tre.gulv(1));  // Utskrift: null
```

Programkode* 5.2.7 *c)

Begrepet tak (eng: ceiling) defineres slik: *tak* skal bety den minste verdien i samlingen som er større enn eller lik x. Det betyr at hvis x er i samlingen, er *tak*(x) lik x. Hvis derimot x er større enn alle verdiene, sier vi at *tak*(x) er null. Se *Oppgave* 5.

Hvis vi har en *verdi*, skal *større*(verdi) være den minste av de verdiene i treet som er større enn *verdi*. Det betyr den verdien som kommer rett etter *verdi* i inorden. Hvis *verdi* er større enn eller lik den største verdien i treet, skal *større*(verdi) være null.

Metoden public T *større*(T verdi) kan kodes ved hjelp av flg. idé: Hvis *verdi* er mindre enn verdien i en node, så er nodeverdien en foreløpig kandidat. Hvis ikke, dvs. hvis *verdi* er større enn eller like nodeverdien, så ligger verdien vi leter etter i nodens høyre subtre. Hvis verdien vi leter etter forekommer flere ganger, får vi da den første av dem i inorden:

```
public T større(T verdi)
{
  if (tom()) throw new NoSuchElementException("Treet er tomt!");
  if (verdi == null) throw new NullPointerException("Ulovlig nullverdi!");

  Node<T> p = rot;
  T større = null;

  while (p != null)
  {
    int cmp = comp.compare(verdi, p.verdi);

    if (cmp < 0)
    {
      større = p.verdi;  // en kandidat
      p = p.venstre;
    }
    else                 // den må ligge til høyre
    {
      p = p.høyre;
    }
  }

  return større;
}
```

Programkode* 5.2.7 *d)

Flg. eksempel viser hvordan metoden skal virke:

```
Integer[] a = {5,10,3,8,13,7,16,2,6,11};

SBinTre<Integer> tre = SBinTre.sbintre(Stream.of(a));  // Programkode 5.2.3 c)

System.out.println(tre.større(10));  // Utskrift: 11
System.out.println(tre.større(12));  // Utskrift: 13
System.out.println(tre.større(16));  // Utskrift: null
```

Programkode 5.2.7 e)

Oppgaver til Avsnitt 5.2.7

1. Lag metoden `public T maks()`. Den skal returnere den en siste i inorden. Det er den største verdien i treet. Hvis treet er tomt skal det kastes et unntak.
2. Hvis den største verdien forekommer flere ganger, vil den siste i inorden være den siste av dem. Gjør om metoden i *Oppgave* 1 slik at den finner den første (i inorden) av dem hvis den største verdien forekommer flere ganger.
3. Metoden *gulv*() i *Programkode* 5.2.7 *b)* returnerer den første av dem (i inorden) hvis *verdi* forekommer flere ganger i treet. Hvis *verdi* ikke forekommer i treet og den største av dem som er mindre enn *verdi* forekommer flere ganger, er det den siste av dem i inorden som skal returneres. Sjekk at det er slik.
4. Gjør om metoden *gulv*() i *Programkode* 5.2.7 *b)* slik at hvis *verdi* forekommer flere ganger i treet, skal den siste av dem i inorden returneres. Se også *Oppgave* 3.
5. Lag metoden `public T tak(T verdi)`. Den skal returnere den minste av de verdiene som er større enn *verdi*. Hvis *verdi* er større enn alle verdiene i treet skal metoden returnere `null`. Hvis *verdi* forekommer flere ganger i treet, skal metoden returnere den første av dem i inorden. Hvis *verdi* ikke forekommer i treet og den minste av de som er større forekommer flere ganger, skal den første av dem i inorden returneres.
6. Lag metoden `public T mindre(T verdi)`. Den skal returnere den største av de verdiene i treet som er mindre enn *verdi*. Hvis denne største verdien forekommer flere ganger, skal den siste av dem i inorden returneres. Hvis *verdi* er mindre enn eller lik den minste verdien i treet, skal `null` returneres.
7. Hver permutasjon av tallene fra 1 til *n* gir et binært søketre. Den gjennomsnittlige nodedybden for den første noden i inorden er gjennomsnittet over alle de *n*! (n fakultet) forskjellige permutasjonene.

   a) Finn, ved å studere trærne, den gjennomsnittlig nodedybden for den første noden i inorden for tilfellene $n = 1$, 2, 3 og 4. I tilfellet $n = 3$ kan du se på de 6 trærne i *Figur* 5.2.4 *c)*. I tilfellet $n = 4$ kan du se på de 24 trærne i fasiten til *Oppgave* 2) i *Avsnitt* 5.2.4.

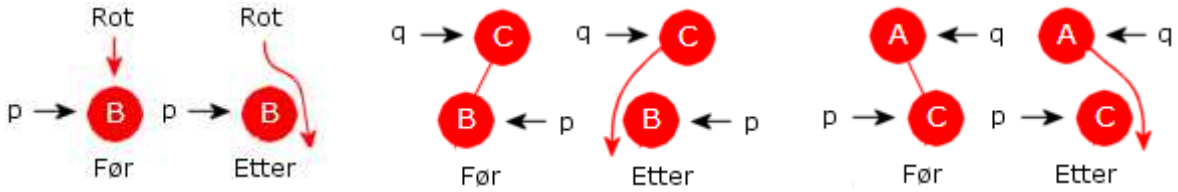
   b) Vis at den gjennomsnittlige nodedybden til den første i inorden i et binært søketre med *n* forskjellige verdier er lik $H_n - 1$ der H_n er summen av de inverse heltallene fra 1 til *n*.

5.2.8 Fjerning av en verdi

Det er normalt tillatt med like verdier (duplikater). Derfor bestemmer vi at hvis det er flere forekomster av en verdi som skal fjernes, fjerner vi den første (i inorden) av dem. La *p* være noden som inneholder verdien som skal fjernes. Da har vi tre hovedtilfeller:

1. *p* har ingen barn (*p* er en bladnode)
2. *p* har nøyaktig ett barn (venstre eller høyre barn)
3. *p* har to barn

Tilfelle 1): *p* har ingen barn. Da fjernes noden *p* ved at referansen ned til *p* «nulles». Dvs. hvis *p* er rotnoden, settes rotreferansen til null. Hvis ikke, vil *p* ha en forelder *q*. Hvis *p* er venstre barn til *q*, settes *q.venstre* lik null og ellers settes *q.høyre* lik null. Se figurene under:



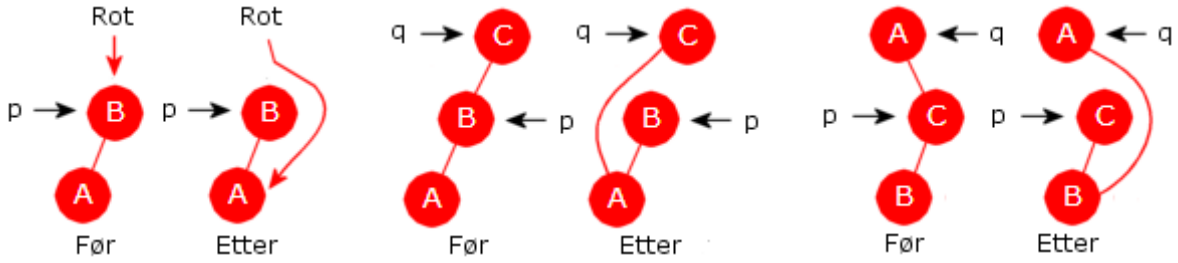
Figur 5.2.8 a) : *p* «løsrives» fra treet ved at referansen ned dit «nulles»

Dette kodes slik:

```
if (p == rot) rot = null;                    // venstre i figuren
else if (p == q.venstre) q.venstre = null;   // midten i figuren
else q.høyre = null;                         // høyre i figuren
```

Programkode 5.2.8 a)

Tilfelle 2): *p* har nøyaktig ett barn. Da er det to muligheter. Først at *p* har et venstre barn og dernest at *p* har et høyre barn. Disse to mulighetene er speilbilder av hverandre. Vi nøyer oss derfor med å beskrive det første tilfellet, dvs. at *p* har et venstre barn. Figurene under viser de tre situasjonene vi kan ha når *p* har et venstre (og ikke et høyre) barn:



Figur 5.2.8 b) : *p* == rot

Til venstre i *Figur* 5.2.8 *b)* skal rotnoden fjernes siden *p* = *rot*. Fjerningen skjer ved at *rot* settes lik *p.venstre*. Se Før og Etter. I midten vises situasjonen (Før) der *p* er venstre barn til sin forelder *q*. Fjerningen skjer ved at *q.venstre* settes lik *p.venstre* (se Etter). Så til høyre i figuren er (Før) *p* høyre barn til sin forelder *q*. Da fjernes *p* ved at *q.høyre* settes lik *p.venstre*. Dette kan kodes slik:

```
// p inneholder verdien som skal fjernes, q er forelder til p
if (p == rot) rot = p.venstre;                    // venstre i figuren
else if (p == q.venstre) q.venstre = p.venstre;   // midten i figuren
else q.høyre = p.venstre;                         // høyre i figuren
```

Programkode 5.2.8 b)

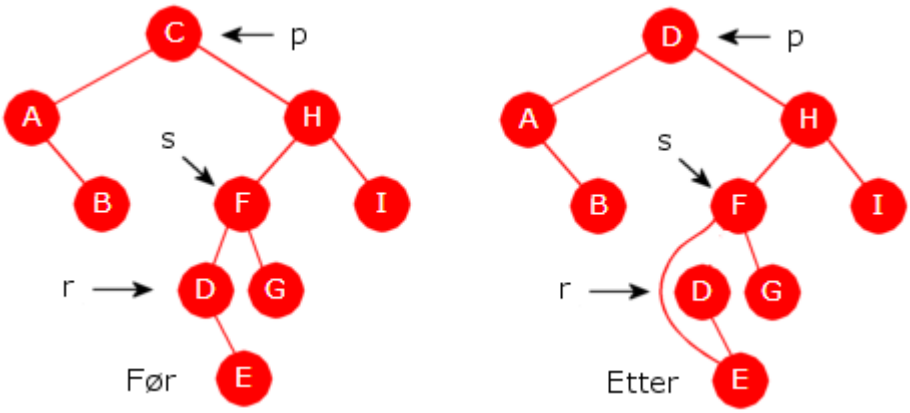
Legg merke til at *Programkode* 5.2.8 *a)* som dekker tilfelle 1) (dvs. at *p* ikke har barn), egentlig er unødvendig siden *Programkode* 5.2.8 *b)* over også tar seg av det tilfellet. Det kommer av at hvis *p* ikke har barn, vil spesielt *p.venstre* være null. Når vi senere skal lage fullstendig kode for *fjern*-metoden, tar vi derfor ikke med egen kode for tilfelle 1).

Vi tar ikke her opp den andre delen av tilfelle 2), dvs. at *p* har et høyre barn. Som nevnt over er det et speilbilde av den første delen. Vi må imidlertid ta det med når vi skal lage fullstendig kode for *fjern*-metoden.

Det er ikke helt korrekt å si at teknikken ovenfor gjør at noden som *p* refererer til, blir fjernet. I koden for en *fjern*-metode vil *p* være en lokal hjelpereferanse. Omdirigeringen fører til at *p* etterpå blir den eneste referansen til noden. Når metodekallet er ferdig, «dør» *p* og dermed vil det ikke lenger være referanser til noden. Plassen som noden okkuperer, går derfor til såkalt resirkulering (hentes etter hvert inn av garbage collector).

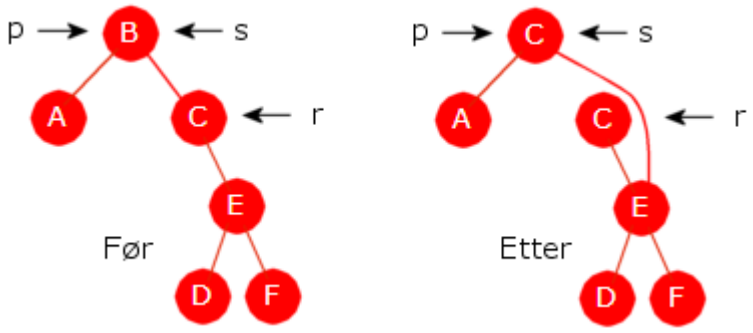
Tilfelle 3): *p* har to barn. Her kan vi ikke fjerne *p* siden det vil ødelegge treet. Vi lar isteden *p* få en ny verdi. Vi velger en verdi som gjør at treet bevares som et binært søketre. Da bruker vi verdien i den noden *r* som kommer rett etter *p* i inorden og fjerner isteden *r*. Husk at *r* er den noden som ligger lengst ned til venstre i det høyre subtreet til *p*. Det betyr spesielt at *r* ikke har et venstre barn og kan dermed «fjernes» ved hjelp av en referanseomdirigering.

Først kopieres verdien til *r* inn i *p*. La videre *s* være forelder til *r*. Da blir det to muligheter:
i) Hvis *s* er forskjellig fra *p* (se figuren under) «fjernes» *r* ved at *s.venstre* settes lik *r.høyre*:



Figur 5.2.8 c) : i) Forelderen *s* til *r* er forskjellig fra *p*

ii) Hvis *s* er lik *p* (se figuren under), «fjernes» *r* ved at *s.høyre* settes lik *r.høyre*:



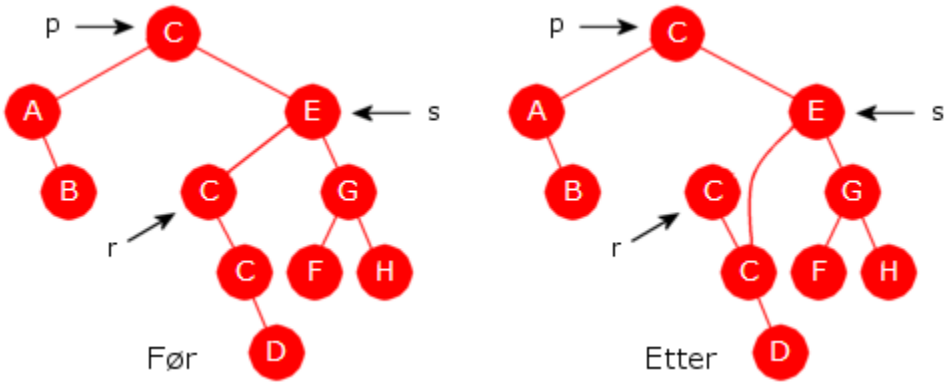
Figur 5.2.8 d) : ii) Forelderen *s* til *r* er lik *p*

Det som er beskrevet over og som vises i figurene 5.2.8 *c)* og 5.2.8 *d)*, kodes slik:

```
// r er etterfølgeren til p i inorden, s er forelder til r

p.verdi = r.verdi;   // kopierer verdien i r til p

if (s != p) s.venstre = r.høyre;
else s.høyre = r.høyre;
```

Programkode* 5.2.8 *c)

Vi har illustrert de ulike tilfellene for fjerning av verdier ved hjelp av figurer. Det er lett å se at den beskrevne teknikken sørger for at trærne bevares som binære søketrær. Men det forekommer ikke like verdier i noen av eksemplene. Vil teknikken også fungere for like verdier? På figuren under er det satt opp et eksempel på et tre med like verdier:



Figur 5.2.8 e) : Fjerning av første forekomst av en verdi som er flere steder

Vi har tidligere sagt (øverst i *Avsnitt* 5.2.8) at hvis vi skal fjerne en verdi som forekommer flere steder i treet, fjerner vi den første vi finner. Vi tenker oss nå at verdien *C* skal fjernes og at *p* i *Figur* 5.2.8 *e)* refererer til første (fra roten og nedover) forekomst av *C*. De to andre forkomstene ligger (og må ligge) i det høyre subtreet til *p*. Siden *p* har to barn, har vi nå tilfelle 3). Det er *r* som kommer etter *p* i inorden og *s* er forelder til *r*. Teknikken sier at først skal verdien i *r* kopieres over i *p*. Deretter skal *r* fjernes ved at *s.venstre* settes lik *r.høyre*. Vi ser at treet fortsatt er et binært søketre.

```
public boolean fjern(T verdi)  // hører til klassen SBinTre
{
  if (verdi == null) return false;  // treet har ingen nullverdier

  Node<T> p = rot, q = null;   // q skal være forelder til p

  while (p != null)            // leter etter verdi
  {
    int cmp = comp.compare(verdi,p.verdi);      // sammenligner
    if (cmp < 0) { q = p; p = p.venstre; }      // går til venstre
    else if (cmp > 0) { q = p; p = p.høyre; }   // går til høyre
    else break;    // den søkte verdien ligger i p
  }
  if (p == null) return false;   // finner ikke verdi

  if (p.venstre == null || p.høyre == null)  // Tilfelle 1) og 2)
  {
    Node<T> b = p.venstre != null ? p.venstre : p.høyre;  // b for barn
    if (p == rot) rot = b;
    else if (p == q.venstre) q.venstre = b;
```

```
      else q.høyre = b;
    }
    else  // Tilfelle 3)
    {
      Node<T> s = p, r = p.høyre;   // finner neste i inorden
      while (r.venstre != null)
      {
        s = r;    // s er forelder til r
        r = r.venstre;
      }

      p.verdi = r.verdi;   // kopierer verdien i r til p

      if (s != p) s.venstre = r.høyre;
      else s.høyre = r.høyre;
    }

    antall--;   // det er nå én node mindre i treet
    return true;
  }
```

Programkode 5.2.8 d)

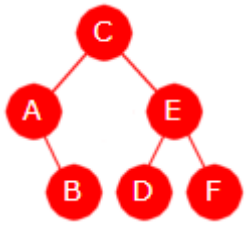
Hvis en verdi forekommer flere ganger, vil *Programkode* 5.2.8 *d)* kun fjerne første forekomst (første i inorden). Vi kan imidlertid få fjernet alle ved å kalle metoden på nytt og på nytt. Se *Oppgave* 3. Men det er egentlig ineffektivt siden verdien må letes opp på nytt for hver gang. En bedre måte er å gå nedover i treet kun én gang (alle ligger på en og samme *gren*) og på veien nedover ta vare på (f.eks. ved hjelp av en stakk) alle nodene (og deres foreldre) som inneholder verdien. Deretter kan man fjerne nodene i motsatt rekkefølge. Legg merke til (se verdien *C* i *Figur* 5.2.8 *e*) at alle nodene som inneholder samme verdi (bortsett fra den første) ikke har venstre barn. Da blir det enkelt å fjerne dem. Se *Oppgave* 4.

I noen tilfeller er det av interesse å kunne fjerne den minste (eller den største) verdien i et binært søketre. Den minste ligger lengst ned til venstre. Se *Programkode* 5.2.7 *a)*. Noden som inneholder den minste verdien har ikke venstre barn. Derfor kan den enkelt fjernes. Vi må imidlertid passe på det tilfellet at det er rotnoden som inneholder den minste verdien:

```
  public void fjernMin()  // hører til klassen SBinTre
  {
    if (tom()) throw new NoSuchElementException("Treet er tomt!");

    if (rot.venstre == null) rot = rot.høyre;  // rotverdien er minst
    else
    {
      Node<T> p = rot.venstre, q = rot;
      while (p.venstre != null)
      {
        q = p;  // q er forelder til p
        p = p.venstre;
      }
      // p er noden med minst verdi
      q.venstre = p.høyre;
    }
    antall--;  // det er nå én node mindre i treet
  }
```

Programkode 5.2.8 e)

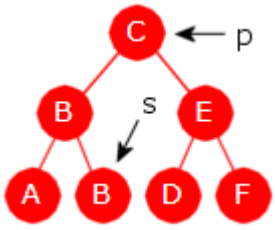
Mulige problemer: En serie kall på *fjern*-metoden i *Programkode* 5.2.8 *d)* kan i prinsippet føre til at treet blir skjevt siden det er en asymmetri i algoritmen. I tilfelle 3) i algoritmen er det alltid *p*-nodens høyre subtre som mister en node. Ta utgangspunkt i treet i *Figur* 5.2.8 *f)*



Figur 5.2.8 f)

til venstre. *C* fjernes ved at den først erstattes med *D* og så fjernes isteden *D*-noden. Legg så inn *C* igjen. Den vil havne nederst i venstre subtre. Vi kan gå videre med dette: Fjern *D*, legg inn *D*, fjern *E* og til slutt legg inn *E*. Da vil treet fortsatt ha de samme seks verdiene, men det har blitt skjevt. Tegn treet og se selv!

I algoritmens tilfelle 3) kunne det være et alternativ å bruke den speilvendte teknikken, dvs. la nodens verdi bli erstattet med verdien til dens forgjenger i inorden og isteden fjerne forgjengeren. Se *Figur* 5.2.8 *g)* til venstre. Der skal verdien til *p* (dvs. *C*) fjernes. Dens forgjenger i inorden er noden *s* med verdien *B*. Vi kopierer *B* inn i *p* og fjerner *s*. (Har *p* venstre barn vil forgjengeren ikke ha høyre barn).



Figur 5.2.8 g)

Det kan imidlertid oppstå et problem med den speilvendte teknikken. Hvis treet, som her, har like verdier, vil det kunne bli ødelagt som et binært søketre. Her (*Figur* 5.2.5 *g*) forekommer *B* to ganger. Den speilvendte teknikken gjør at *B* vil havne både i *p*-noden og i dens venstre barn. Men det er i strid med *Definisjon* 5.2.1.

Det er imidlertid lett å identifisere de tilfellene der den speilvendte teknikken feiler. La *p* være noden som inneholder verdien som skal fjernes. Da går det galt kun hvis den forrige til *p* i inorden har samme verdi som sin forelder. Slik er det f.eks. *Figur* 5.2.8 *g)*. Dette kunne vi utnytte til å lage en tilnærmet symmetrsk algoritme får å fjerne en verdi. Se *Oppgave* 8.

Hvis vi derimot skal operere med et binært søketre der det ikke er tillatt med duplikater (like verdier), så kan begge teknikkene brukes uten problemer for tilfelle 3). Da kunne f.eks. `fjern`-metoden konstrueres slik at de to teknikkene alternerte. Se *Oppgave* 9.

Oppsummering:

- Den vanlige algoritmen for å fjerne en verdi i et generelt binært søketre kan deles i tre tilfeller. La *p* være noden som inneholder verdien som skal fjernes:
  1. *p* har ingen barn (dvs. *p* er en bladnode). Hvis *p* er rotnoden, settes rotreferansen til *null*. Hvis ikke, settes referansen fra forelderen til *p* lik *null*.
  2. *p* har nøyaktig ett barn (et venstre eller et høyre barn). Hvis *p* er rotnoden, settes rotreferansen og hvis ikke, referansen fra forelderen, til barnet til *p*.
  3. *p* har to barn. Da erstattes verdien til *p* med verdien til etterfølgeren til *p* i inorden og isteden fjernes etterfølgeren.
- Den vanlige fjerningsalgoritmen har en innebygd asymmetri siden det i tilfelle 3) alltid er *p*-nodens høyre subtre som mister en node. Men det har normalt liten betydning for effektiviteten. Praktiske studier viser at det må et svært stort antall tilfeldige fjerninger og innlegginger til før treet blir vesentlig skjevt. Tilfelle 3) oppstår i gjennomsnitt kun hver tredje gang.
- Hvis treet er uten duplikater, kan man gjøre fjerningen i tilfelle 3) symmetrisk ved å alternere mellom å fjerne den neste og den forrige.
- Det er også mulig å gjøre fjerningen i tilfelle 3) tilnærmet symmetrisk i generelle binære søketrær (dvs. når like verdier er tillatt). En «forrigefjerning» kan imidlertid kun utføres i de tilfellene der treet ikke «ødelegges».
- Algoritmen er på samme måte som innleggingsmetoden i gjennomsnitt av logaritmisk orden. Det er fordi vi går fra roten og nedover i treet langs en gren kun én gang.

Oppgaver til Avsnitt 5.2.8

1. Fjern, i den gitte rekkefølgen, verdiene C, A, H, J og L fra det treet som ble laget i *Oppgave 2a)* i *Avsnitt* 5.2.3.
2. Fjern, i den gitte rekkefølgen, verdiene 4, 7, 3 og 8 fra det treet som ble laget i *Oppgave 2c)* i *Avsnitt* 5.2.3.
3. Lag metoden `int fjernAlle(T verdi)`. Den skal fjerne alle forekomstene av *verdi* og returnere det antallet som ble fjernet. Det betyr spesielt at den skal returnere 0 hvis treet ikke inneholder *verdi*. Lag den ved å gjøre gjentatte kall på *fjern*-metoden inntil det ikke er flere forekomster av *verdi* igjen i treet.
4. Hvorfor vil alle forekomster av samme verdi ligge på en og samme gren i treet? Hvorfor vil alle nodene som inneholder samme verdi (bortsett fra den første/øverste) ikke kunne ha venstre barn? Lag en forbedret versjon av `int fjernAlle(T verdi)` (se *Oppgave* 3). Gå nedover langs den grenen som inneholder alle forkomstene av *verdi*. Bruk en stakk. For hver forekomst legg både noden og så nodens forelder på stakken. Fjern deretter hver node (ta fra stakken) som inneholder *verdi* (bortsett fra den første/øverste) ved en pekeromdirigering. Den første/øverste noden må fjernes på vanlig måte siden den kan ha både ingen, ett og to barn.
5. Klassen `SBinTre` skal implementere grensesnittet `Beholder`, men det er inntil videre kommentert vekk. Det som mangler er metodene *nullstill*() og *iterator*(). Den siste blir tatt opp i neste avsnitt. Lag metoden *nullstill*(). Den skal «tømme» treet. Et første forsøk kan være å kalle metoden *fjernMin*() inntil treet er tomt. Prøv det! Det optimale løsningen er imidlertid å traversere treet og så fortløpende nulle verdier og pekere. Prøv det! Se også Oppgave 7b) i *Avsnitt* 5.1.7.
6. Hvis det er flere forekomster av den minste verdien, vil metoden *fjernMin*() fjerne den første (i inorden) av dem. Lag metoden *public int fjernAlleMin*(). Den skal fjerne alle forekomster av den minste verdien og returnere antallet som ble fjernet. Hvis det var bare en forekomst av den minste, skal den returnere 1.
7. Lag metoden `public void fjernMaks()`. Den skal hvis treet ikke er tomt, fjerne den største verdien i treet. Lag det slik at hvis den største verdien forekommer flere ganger, er det den av dem som kommer først i inorden som fjernes. Lag så metoden `public int fjernAlleMaks()`. Den skal fjerne alle forekomster av den største verdien og returnere antallet som ble fjernet. Hvis det var bare en forekomst, skal den returnere 1.
8. Start med treet i *Figur* 5.2.8 *f)*. Fjern så *C*, legg *C* inn igjen, fjern *D*, legg *D* inn igjen, fjern *E* og legg så til slutt *E* inn igjen. Tegn det treet du da får. Er det skjevt?
9. Start med treet i *Figur* 5.2.8 *f)*. Fjern *C* på vanlig måte og sett så *C* inn igjen. Fjern så *D* ved å bruke den forrige i inorden. Sett så inn *D* igjen. Hvordan er treet nå?
10. Gjør om fjern-metoden i *Programkode* 5.2.8 *d)* slik at de to speilvendte teknikkene for tilfelle 3) om mulig brukes annenhver gang. Hvis den forrige til *p* i inorden har samme verdi som sin forelder, er det ikke tillatt å fjerne ved å kopiere fra *p* sin forgjenger til *p* og så fjerne forgjengeren.
11. Lag en spesiell versjon av klassen der duplikater ikke er tillatt. Da må det lages en ny versjon av leggInn. Nå kan fjern-metoden kodes med nestfjerning og forrigefjerning annenhver gang.

5.2.9 Traversering

Traverseringsteknikkene utviklet i *Delkapittel* 5.1 for vanlige binære trær, virker i et binært søketre. Enkelte av dem kan utvikles videre, f.eks. iterator-teknikken. Grensesnittet `Iterator` har metodene *hasNext*(), *next*() og *remove*(). Men i klassen `BinTre` ble kun de to første implementert. Metoden *remove*() er en default-metode i grensesnittet og er der kodet med en `UnsupportdOperationException`. Klassen `InordenIterator` har det som ble laget for `BinTre` og kan legges inn som en indre klasse i `SBinTre`. I tillegg må klassen ha en iterator-metode:

```
public Iterator<T> iterator()  // returnerer en iterator
{
  return new InordenIterator();
}
```

Programkode* 5.2.9 *a)

Dermed er (hvis *nullstill*() også er kodet - se Oppgave 5 i *Avsnitt* 5.2.8) alle metodene i `Beholder` implementert og vi kan ta vekk kommentartegnet i `SBinTre`:

```
public class SBinTre<T> implements Beholder<T>
```

Iteratoren brukes implisitt i en forAlle-løkke som dermed skriver verdiene i inorden:

```
Integer[] a = {2,8,6,1,7,4,3,9,5,10};
SBinTre<Integer> tre = SBinTre.sbintre(Stream.of(a));  // Programkode 5.2.3 c)
for (int k : tre) System.out.print(k + " ");
// Utskrift: 1 2 3 4 5 6 7 8 9 10
```

Programkode* 5.2.9 *b)

To utvidelser av iterator-teknikken er aktuelle i klassen `SBinTre`. Hvis en iterator er satt i gang, er det likevel fullt mulig å gjøre endringer i treet ved vanlige innlegginger og fjerninger av verdier. Da kan resultatet av traverseringen bli annerledes enn ventet (eng: the behavior is unspecified). Dette er det aktuelt å gjøre noe med. Den andre utvidelsen går ut på å kode metoden *remove*() i iteratorklassen. Det er mulig å få til siden verdier kan fjernes fra et binært søketre slik at treet bevares som et søketre.

Når det gjelder endringer i treet kan en velge, slik som det er gjort i *java.util*, en konservativ tilnærmingsmåte. Det betyr at så fort det har blitt gjort en endring utenfor iteratoren (en innlegging eller en fjerning) blir alle iteratorer blokkert. Blokkering betyr at det kastes unntak hvis *next*() eller *remove*() kalles. Det kan vi få til ved å registere alle endringer ved hjelp av en heltallsvariabel. Den kan f.eks. få navnet *endringer*. Den legges inn som instansvariabel i `SBinTre`. Dette er markert med rød skrift under:

```
private Node<T> rot;                        // peker til rotnoden
private int antall;                         // antall noder
private final Comparator<? super T> comp;   // komparator
private int endringer;                      // antall endringer
```

Programkode* 5.2.9 *c)

Neste skritt er å få den oppdatert i alle metoder som gjør endringer. F.eks. kan de to siste setningene før `return` i metoden *leggInn*() være:

```
endringer++;          // det er gjort en endring i treet
antall++;             // en verdi mer i treet
```

Programkode* 5.2.9 *d)

Tilsvarende må setningen `endringer++;` inn i alle metoder som fjerner verdier.

Klassen `InordenIterator` må ha en ekstra instansvariabel som registerer endringene som skjer i iteratoren (*remove*). Den kan f.eks. få navnet *iteratorendringer*. Dermed får klassen tre instansvariabeler. Den nye er markert med rød skrift:

```
private Stakk<Node<T>> s = new TabellStakk<>();  // for traversering
private Node<T> p = null;                        // nodepeker
private int iteratorendringer;                   // iteratorendringer
```

Programkode 5.2.9 e)

Poenget nå er at konstruktøren setter den siste endringsvariabelen lik den første:

```
public InordenIterator()           // konstruktør
{
  if (rot == null) return;         // treet er tomt
  p = først(rot);                  // bruker hjelpemetoden
  iteratorendringer = endringer;   // setter treets endringer
}
```

Programkode 5.2.9 f)

og dermed kan metoden *next*() sjekke om de fortsatt er like eller ikke. Hvis ikke, har det skjedd en endring i treet utenfor iteratoren:

```
public T next()
{
  if (iteratorendringer != endringer)
                throw new ConcurrentModificationException();

  // resten av koden for next() er som før
}
```

Programkode 5.2.9 g)

Hvis alt dette er lagt inn i klassen `SBinTre` (og i den lokale klassen `InordenIterator`), vil flg. kodebit vise hvilken effekt dette får:

```
Integer[] a = {2,8,6,1,7,4,3,9,5,10};                  // verdier
SBinTre<Integer> tre = SBinTre.sbintre(Stream.of(a));  // Programkode 5.2.3 c)

Iterator<Integer> i = tre.iterator();      // en iterator er opprettet

tre.leggInn(6);  // en innlegging er en endring
i.next();        // kaster en ConcurrentModificationException
```

Programkode 5.2.9 h)

I eksemplet over skjer endringen (en innlegging) etter at iteratoren er opprettet. Dermed kastes det en `ConcurrentModificationException` når *next*() kalles. Også de andre metodene (*nullstill*(), *fjern*() og *fjernMin*()) som endrer treet, må få kode slik at det samme skjer. Se oppgavene under.

Oppgaver til Avsnitt 5.2.9

1. Legg inn setningen `endringer++` som nest siste setning i `fjern` og `fjernMin`. Gjør så endringer i *Programkode* 5.2.9 *h)* slik at et kall på en av disse to metodene etter at en iterator er opprettet, fører til at `next()` kaster en `ConcurrentModificationException`.
2. Metoden *`nullstill`*`()` tømmer hele treet. Se Oppgave 5 i *Avsnitt* 5.2.8. Legg inn kode der slik at den fungerer som de andre metodene som gjør endringer i treet.
3. Sjekk at *Programkode* 5.2.9 *h)* virker slik det er beskrevet. Legg inn kode som sjekker at endringer gjort ved hjelp av `fjern`, `fjernMin` eller `nullstill` får samme effekt.
4. La klassen `SBinTre` ha metoden `public Iterator<T> iterator(T verdi)`. Den skal returnere en iterator der første kall på `next` gir den minste av verdiene i treet som er større enn eller lik `verdi`. Hvis denne minste verdien forekommer flere ganger skal første kall på `next` gi den første av dem i inorden. Parameterverdien `verdi` kan, men behøver ikke ligge i treet. Hint: Lag en konstruktør i `InordenIterator` med en verdi av type *T* som parameter. Se også metoden `tak` fra *Avsnitt* 5.2.7.
5. La klassen `SBinTre` få en iterator som går i omvendt inorden. Da må det lages en egen iteratorklasse. La den få navnet `OmvendtInordenIterator`. La metoden som returnerer en instans av klassen få navnet `public Iterator<T> riterator()` (r for reversert). Lag også metoden `public Iterator<T> riterator(T verdi)`. Den skal returnere en iterator der første kall på `next` gir den største av verdiene i treet som er mindre enn eller lik `verdi`. Hvis denne største verdien forekommer flere ganger skal første kall på `next` gi den første av dem i omvendt inorden. Se også metoden `gulv` fra *Avsnitt* 5.2.7.

5.2.10 Metoden `remove` i iteratoren

Metoden *remove()* er *default* i `Iterator` og er dermed formelt implementert: Den kaster en `UnsupportedOperationException`. Hvis den kodes (overstyres), må flg. tre krav oppfylles:

- *remove()* skal fjerne verdien som sist ble returnert av *next()*
- *remove()* kan kun kalles én gang for hvert kall på *next()*
- hvis *remove()* kalles i én iterator, skal alle andre iteratorer blokkeres

Et lovlig kall på *remove()* vil føre til en endring i treet. Det betyr at variabelen `endringer` må økes. Det får som konsekvens at andre iteratorer som måtte være i gang samtidig (eng: concurrent), blir blokkeret. Men den iteratoren der kallet på *remove()* skjedde, skal kunne fortsette. Poenget er at fjerningen skjer på et sted som iteratoren allerede har passert og får dermed ingen konsekvenser for hva *next()* gir neste gang. Koden for *remove()* må derfor starte og slutte slik som i koden under (de to spesielle «tellerne» er markert med rødt):

```
public void remove()
{
  if (iteratorendringer != endringer)
              throw new ConcurrentModificationException();
  // her skal koden for selve fjerningen komme
  iteratorendringer++;     // en endring i treet via iteratoren
  endringer++;             // en endring i treet
  antall--;                // en verdi mindre i treet
}
```

Programkode* 5.2.10 *a)

Den vanskeligste delen er å kode selve fjerningen. Treet må bevares som et søketre og senere kall på *next()* skal virke som normalt. Det blir mange spesialtilfeller. I tillegg vil en fjerning kunne få konsekvenser for stakken som styrer traverseringen. I *fjern*-metoden i *Programkode* 5.2.8 *d)* fant vi først verdien som skulle fjernes. Det gav oss samtidig en peker til dens forelder. Dermed var det mulig å kode fjerningen på en ikke alt for komplisert måte.

Men i iteratoren har vi kun en peker *p*. Et kall på *next()* vil flytte den til neste node i inorden. Men det er den dom *p* opprinnelig stod på som skal fjernes. Det er mulig å finne den forrige til *p*, men det er enklere med en ekstra nodepeker *q*. Den settes lik *p* før *p* flyttes. Variablene til iteratoren (se *Programkode* 5.2.9 *e)*) blir nå:

```
private Stakk<Node<T>> s = new TabellStakk<>();  // for traversering
private Node<T> p = null;                        // nodepeker
private Node<T> q = null;                        // ekstra nodepeker
private int iteratorendringer;                   // iteratorendringer
```

Det er *q* sin verdi som skal fjernes når *remove()* kalles. I *next*() settes *q* lik *p* før *p* flyttes:

```
public T next()
{
  if (iteratorendringer != endringer)
                throw new ConcurrentModificationException("Treet er endret!");
  if (!hasNext()) throw new NoSuchElementException("Finnes ikke!");

  T verdi = p.verdi;               // tar vare på verdien i noden p
  q = p;                           // q oppdateres før p flyttes
  // resten av next skal være som før
}
```

Programkode* 5.2.10 *b)

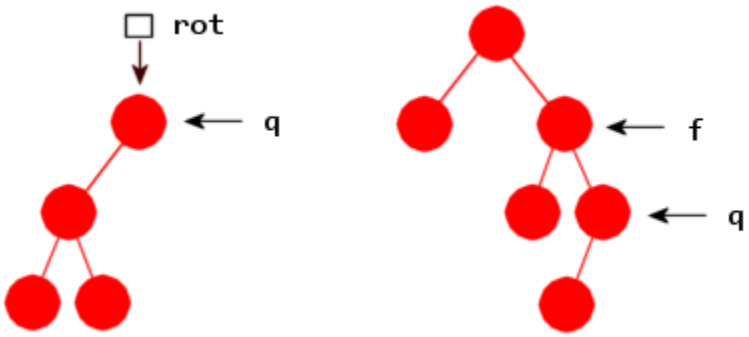
Kravet til *remove*() er at den ikke kan kalles før det er gjort et kall *next*(). Den kan heller ikke kalles to ganger på rad, dvs. uten at det har vært et kall på *next*() i mellomtiden. Dette kan løses ved å sette *q* til null når *remove*() avslutter. Dermed kan vi teste på om *q* er null eller ikke når *remove*() starter. Dermed må *remove*() ha denne koden:

```
public void remove()
{
  if (q == null) throw new IllegalStateException("Fjerning er ulovlig!");

  if (iteratorendringer != endringer)
              throw new ConcurrentModificationException("Treet er endret!");

  // her skal koden for selve fjerningen komme

  q = null;                 // q settes til null
  iteratorendringer++;      // en endring i treet via iteratoren
  endringer++;              // en endring i treet
  antall--;                 // en verdi mindre i treet
}
```

Programkode* 5.2.10 *c)

Verdien i noden *q* skal fjernes. Vi deler det i de to tilfellene **1)** at `q.høyre` er `null` og **2)** at `q.høyre` ikke er `null`. I et binært søketre vil en node i gjennomsnitt ha et tomt høyre subtre, dvs. at `q.høyre` er null, i halvparten av tilfellene. Se *Formlene* 5.2.4 *c) - e)*.

Tilfellet **1)** har de to undertilfellene ***a)*** at *p* er `null` ***b)*** at *p* ikke er `null`. Hvis *p* er `null`, vil *q* være den siste i inorden. Da slettes *q* ved en omdirigering av en peker, dvs. ved å sette `f.høyre = q.venstre` der *f* er forelder til *q*. Vi finner *f* ved å starte i roten og gå nedover mot høyre. Vi må imidlertid passe på det spesialtilfellet der *q* er lik roten. I så fall må vi sette `rot = q.venstre`.



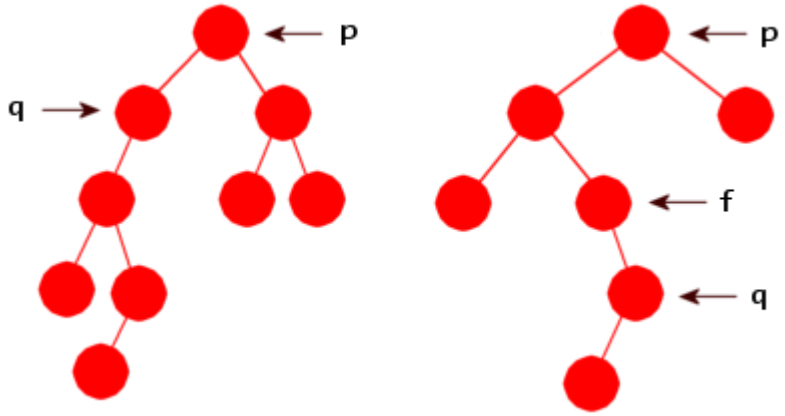
Figur 5.2.10 a): Tilfellet 1a): q.høyre == null og p == null

Tilfellet **1a)** kodes slik:

```
if (q == rot)                          // q er lik roten
{
  rot = q.venstre;                     // q fjernes
}
else
{
  Node<T> f = rot;                     // starter i roten
  while (f.høyre != q) f = f.høyre;    // går mot høyre
  f.høyre = q.venstre;                 // q fjernes
}
```

Programkode* 5.2.10 *d)

I tilfellet **1b)** (`q.høyre` er `null` og *p* ikke `null`) må vi ha at `p.venstre` ikke er `null` siden *q* må ligge i det venstre subtreet til *p*. Videre vil `p.venstre` normalt ha et høyre subtre. I så fall vil *q* ligge lengst ned til høyre i det subtreet. Men vi må ta hensyn til at `p.venstre` ikke har et høyre subtre. I det tilfellet må *q* være lik `p.venstre`.



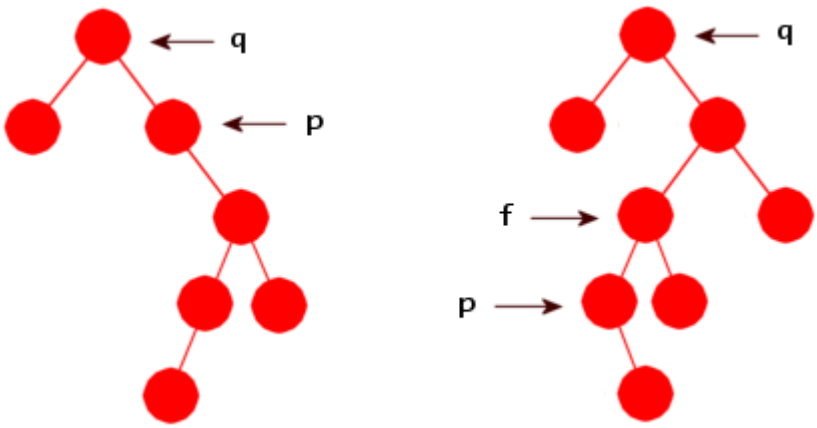
Figur 5.2.10 b): Tilfellet 1b): q.høyre == null og p != null

Tilfellet **1b)** kodes slik:

```
if (q == p.venstre)                    // p.venstre har ikke høyre subtre
{
  p.venstre = q.venstre;               // q fjernes
}
else
{
  Node<T> f = p.venstre;               // starter i p.venstre
  while (f.høyre != q) f = f.høyre;    // går mot høyre
  f.høyre = q.venstre;                 // q fjernes
}
```

Programkode* 5.2.10 *e)

I tilfellet **2)** (dvs. `q.høyre != null`) må *p* ligge i det høyre subtreet til *q*. Hvis `q.høyre` ikke har et venstre subtre, må nødvendigvis *p* være lik `q.høyre`. Hvis ikke, vil *p* ligge lengst ned til venstre i det høyre subtreet til `q.høyre`.



Figur 5.2.10 c): Tilfellet 2): q.høyre forskjellig fra null

I *Figur* 5.2.10 *c)* er tilfellet at `q.venstre = null` ikke tatt med. I den vanlige algoritmen for å fjerne en nodeverdi skilte vi mellom det at noden hadde to barn og at noden hadde ett barn. I det siste tilfellet ble noden fjernet ved at en peker fra dens forelder ble omdirigert. Det var

enkelt å få til siden foreldernoden var kjent. I vårt tilfelle har vi ikke foreldernoden til *q*. Derfor velger vi her å bruke «erstatningsteknikken» både når `q.venstre` er `null` og ikke `null`. Vi kopierer verdien til *p* inn i *q* og fjerner isteden noden *p*.

Hvis `q.høyre = p` (til venstre *Figur* 5.2.10 *c*), vil setningen `q.høyre = p.høyre` fjerne *p*. Hvis ikke (til høyre *Figur* 5.2.10 *c*), må vi ha tak i forelderen *f* til *p*. Men i dette tilfellet er *f* den som kommer etter *p* i inorden og må dermed ligge øverst på stakken. Når *p* blir satt til å peke på den sammen noden som *q* må nodene fra og med `q.høyre` og nedover til *f* fjernes fra stakken (de ble lagt på stakken når *p* ble flyttet). Tilfellet **2)** kan kodes slik:

```
q.verdi = p.verdi;                      // kopierer

if (q.høyre == p)                       // q.høyre har ikke venstre barn
{
  q.høyre = p.høyre;                    // fjerner p
}
else                                    // q.høyre har venstre barn
{
  Node<T> f = s.taUt();                 // forelder f til p ligger på stakken
  f.venstre = p.høyre;                  // fjerner p
  while (f != q.høyre) f = s.taUt();    // fjerner fra stakken
}

p = q;                                  // setter p tilbake til q
```

Programkode* 5.2.10 *f)

Vi kan nå fullføre metoden *remove()* (*Programkode* 5.2.10 *c*) ved å legge inn koden som behandler tilfellene **1a)**, **1b)** og **2)**. Vi må imidlertid passe på (i tillegg) å få med det som trengs av `if - else`. Se *Oppgave* 1.

I flg. eksempel brukes *remove*() til å fjerne eventuelle duplikater i et tre:

```
Integer[] a = {4,8,3,1,7,4,9,1,6,10,2,1,5,10,7,8};     // duplikater
SBinTre<Integer> tre = SBinTre.sbintre(Stream.of(a));  // lager treet

System.out.println(tre);                               // skriver

Iterator<Integer> i = tre.iterator();                  // en iterator
int verdi = i.next();                                  // første verdi

while (i.hasNext())                                    // traverserer
{
  int nesteverdi = i.next();                           // neste verdi
  if (verdi == nesteverdi) i.remove();                 // fjerner
  verdi = nesteverdi;                                  // oppdaterer
}

System.out.println(tre);                               // skriver ut

// Utskrift:
// [1, 1, 1, 2, 3, 4, 4, 5, 6, 7, 7, 8, 8, 9, 10, 10]
// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Programkode* 5.2.10 *g)

Eksemplet over viser en av fordelene med iteratorens `remove`-metode. Det hadde ikke vært så enkelt å fjerne alle duplikater ved hjelp av den ordinære `fjern`-metoden i klassen `SBinTre`. Se *Oppgave* 2.

Koden for *`remove()`* ser komplisert ut. En kan lett tro at den er ineffektiv siden det inngår flere while-løkker. While-løkken i tilfellet 1a) starter i roten og går ned til den siste i inorden. Den er i gjennomsnitt av logaritmisk orden (orden n i det verste tilfellet). Men det å fjerne den siste har en sannsynlighet på $1/n$. De øvrige while-løkkene er imidlertid i gjennomsnitt ganske korte (få iterasjoner). I et binært søketre er avstanden (antall kanter) mellom en node og dens neste i inorden i gjennomsnitt konstant (tilnærmet lik 2 - se 5.2.15.5). Med andre ord er *`remove()`*-metoden i gjennomsnitt av konstant orden.

Oppgaver til Avsnitt 5.2.10

1. Ta utgangspunkt i *Programkode* 5.2.10 *c)*. Legg inn det som mangler for å gjøre *`remove()`* ferdig. Da må det være en `if - else` for tilfellene 1) og 2). Inne i `if`-delen for tilfellet 1) må det være en `if - else` for 1a) og 1b). Osv. Sjekk så, når din *`remove()`*-metode er ferdig, at *Programkode* 5.2.10 *g)* virker. Husk å bruke den nye versjonen av *`next()`*.
2. Lag et tre som bygger opp treet ved hjelp av en tilfeldig permutasjon av tallene fra 1 til 20. Bruk så iteratorens *`remove()`*-metode til å fjerne alle partallene.
3. Start med treet i *Programkode* 5.2.10 *g)*. Lag så kode som fjerner alle duplikatene uten å bruke iteratorens *`remove()`*-metode. Da må en bruke den vanlige *`fjern()`*-metoden, men husk at hvis en bruker en iterator, vil den bli blokkert etter et kall på *`fjern()`*.
4. Sjekk at det kommer en `IllegalStateException` hvis det gjøres to kall på *`remove()`* uten at det har vært et kall på *`next()`* mellom eller at det gjøres et kall på *`remove()`* uten at det har vært et kall på *`next()`* først. Sjekk også at hvis det er opprettet to eller flere iteratorer, vil det det komme en i `ConcurrentModificationException` en av de andre hvis det gjøres et kall på *`remove()`* i en av dem.
5. Er *`remove()`*-metoden stabil? Dvs. vil den bevare den rekkefølgen (i inorden) som like verdier hadde tidligere?
6. *Oppgave* 5 i forrige avsnitt handler om å lage en `OmvendtInordenIterator`. Lag *`remove()`*-metoden for den. Her er det viktig å være klar over at den vanlige `remove`-metoden ikke uten videre kan «speilvendes». Se diskusjonen på slutten av *Avsnitt* 5.2.8.
7. Lag metoden *`public int fjernIntervall(T fraverdi, T tilverdi)`*. Den skal fjerne alle verdier i treet som måtte ligge i intervallet `[fraverdi,tilverdi>` og returnere antallet verdier som ble fjernet. Intervallet `[fraverdi,tilverdi>` består av de verdiene i treet som er større enn eller lik `fraverdi` og mindre enn `tilverdi`. Hvis intervallet er tomt skal det ikke fjernes noe og metoden skal returnere 0.

   a) Lag den ved hjelp av iteratorens *`remove()`*-metode. Hvilken orden vil den få?

   b) Lag den ved hjelp av metoden *`intervallsøk`* (se *Oppgave* 3 i *Avsnitt* 5.2.6) og den vanlige *`fjern()`*-metoden. Fjern-metoden brukes til å fjerne alle verdiene som intervallsøk returnerer. Hvilken orden vil den få?

5.2.11 Trær med forelderpekere

I *Avsnitt* 5.1.15 ble binære trær der nodene har en *forelderpeker*, diskutert. Denne teknikken kan også brukes for binære søketrær. I java.util inngår en spesiell type binære søketrær (*rød-svarte trær*) i klassene `TreeSet` og `TreeMap`. Der brukes en forelderpeker i hver node. En av fordelene er at traverseringer kan gjøres uten bruk av ekstra hjelpemidler. Det blir også enklere å kode de metodene der en er avhengig av å kunne gå oppover i treet, f.eks. fra en node og opp til dens forelder. Ulempen er at hver node får en ekstra variabel og det gjør at treet bruker større plass. Flere av metodene vi laget for vanlige binære søketrær kan brukes som de i binære søketrær med forelderpeker. Men alle metoder som gjør endringer i treet (innlegginger og fjerninger) må kodes om. De må lages slik at forelderpekeren i hver node peker til rett node etter endringen.

Nodeklassen i binære søketrær med forelderpeker må se slik ut (det som er forskjellig fra nodeklassen for vanlige binære søketrær, er markert med rødt):

```
private static final class Node<T>
{
  private T verdi;                  // nodens verdi
  private Node<T> venstre;          // venstre barn
  private Node<T> høyre;            // høyre barn
  private Node<T> forelder;         // forelder

  private Node(T verdi, Node<T> v, Node<T> h, Node<T> f)
  {
    this.verdi = verdi;
    venstre = v;
    høyre = h;
    forelder = f;
  }

  private Node(T verdi, Node<T> forelder)
  {
    this(verdi,null,null,forelder);
  }
}
```

Programkode* 5.2.11 *a)

Vi setter navnet *SFBinTre* på klassen binære søketrær med forelderpeker. Som før står *S* før søk, mens bokstaven *F* står for forelder. Den foreløpgige klassen `SFBinTre` er satt opp med lokal nodeklasse, variabler og en del metoder fra klassen *SBinTre* som kan brukes som de er. Iteratorklassen er tatt med, men der mangler `next()` kode.

Hvis du har flyttet klassen `SFBinTre` over til deg, vil flg. kodebit teste om ting så langt fungerer som de skal:

```
SFBinTre<String> tre = SFBinTre.sfbintre();
System.out.print("Antall verdier: " + tre.antall());
System.out.println("  Verdier: " + tre);

// Utskrift: Antall verdier: 0  Verdier: []
```

Programkode* 5.2.11 *b)

Det er bare små endringer som trengs i *leggInn*-metoden i klassen `SBinTre` for at den også skal kunne virke for `SFBinTre`. I flg. versjon er den endringen som er gjort markert med rødt:

```
public boolean leggInn(T verdi)
{
  Node<T> p = rot;    // p starter i roten
  Node<T> q = null;   // hjelpevariabel
  int cmp = 0;        // hjelpevariabel

  while (p != null)
  {
    q = p;                                  // q er forelder til p
    cmp = comp.compare(verdi,p.verdi);      // bruker komparatoren
    p = cmp < 0 ? p.venstre : p.høyre;      // flytter p
  }

  p = new Node<>(verdi,q);  // q er forelder til ny node

  if (q == null) rot = p;
  else if (cmp < 0) q.venstre = p;
  else q.høyre = p;

  antall++;     // én verdi mer i treet
  endringer++;  // innlegging er en endring

  return true;  // vellykket innlegging
}
```

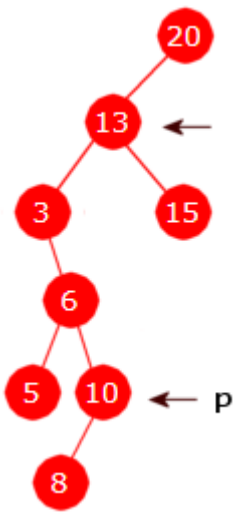
Programkode* 5.2.11 *c)

Hvis vi erstatter den «tomme» leggInn-metoden i SFBinTre med metoden over, vil flg. kodebit virke:

```
Integer[] a = {3,8,6,1,7,10,4,9,5,2};
SFBinTre<Integer> tre = SFBinTre.sfbintre(Stream.of(a));
System.out.println("Verdier: " + tre);

// Utskrift: Verdier: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Programkode* 5.2.11 *d)

Det skal litt mer arbeid til å omkode *fjern*-metoden for klassen SBinTre slik at den vil virke korrekt for klassen SFBinTre, men det er ikke vanskelig. Se *Oppgave* 5.



Klassen InordenIterator som er satt opp i SFBinTre, mangler kode for *next*-metoden. Hvis noden *p* har et ikke-tomt høyre subtre, finner vi den neste ved å gå lengst ned til venstre i det subtreet.

Hvis p.høyre er null må vi oppover i treet for å finne den neste. Figuren til venstre viser at vi må oppover mot roten til den nærmeste noden som er slik at *p* ligger i dens venstre subtre. Det er noden med verdi 13. Dette får vi til ved å gå oppover mot venstre så langt det går. Hvis vi da kommer til roten, var *p* den siste i inorden. Hvis ikke, er forelderen den neste.

Metoden *først(Node<T> q)*, som returnerer den første i inorden i treet med *q* som rot, er ferdigkodet. Vi koder *next*-metoden ved hjelp *først*-metoden og forelder-pekere:

```
public T next()
{
  if (iteratorendringer != endringer)
    throw new ConcurrentModificationException();

  T verdi = p.verdi;              // tar vare på verdien i noden p

  if (p.høyre != null)            // p har høyre subtre
  {
    p = først(p.høyre);           // går til venstre i subtreet
  }
  else                            // p har ikke høyre subtre
  {
    while (p.forelder != null && p.forelder.høyre == p)
    {
      p = p.forelder;             // fortsetter opp mot venstre
    }
    p = p.forelder;               // nå er p den neste (eller null)
  }
  return verdi;                   // returnerer verdien
}
```

Programkode 5.2.11 e)

Oppgaver til Avsnitt 5.2.11

1. Flytt klassen SFBinTre over til deg og erstatt den den «tomme» leggInn-metoden med leggInn-metoden i *Programkode* 5.2.11 *b)*. Sjekk så at *Programkode* 5.2.11 *d)* virker.
2. Bytt ut den foreløpige *next*-metoden i iteratorklassen i SFBinTre med *next*-metoden i *Programkode* 5.2.11 *e)*. Lag en programbit der du tester at iteratoren virker som den skal.
3. Klassen SFBinTre har en *toString*-metode som bruker rekursjon, dvs. det er en privat hjelpemetode som gjør rekursjonen og en offentlig *toString*-metode som kaller hjelpemetoden. Fjern hjelpemetoden. Traverser treet (og bygg opp tegnstrengen) i den offentlige metoden ved hjelp av forelderpekerne. Bruk samme idé som i *next*-metoden i iteratoren.
4. Lag metoden *public String omvendtString*(). Den skal returnere en tegnstreng som inneholder treets verdier i omvendt inorden. Traverser treet (og bygg opp tegnstrengen) ved hjelp av forelderpekerne. Se *Oppgave* 3.
5. Lag kode for *fjern*-metoden i SFBinTre. Ta utgangspunkt i *fjern*-metoden i *Programkode* 5.2.8 *d)*. Her må du passe på at nodene for korrekte foreldre etter en fjerning.
6. Lag kode for *remove*-metoden i klassen InordenIterator. Den inneholder nå kun kode som kaster et unntak. Her kan en bruke samme idé som i *Avsnitt* 5.2.10, dvs. innføre en ekstra peker *q* som ligger «en bak» *p*. Videre må en benytte forelderpeker hvis det er behov for å gå oppover i treet. Fordelen er at vi har direkte aksess til forelder.
7. Gjør som i *Oppgave* 6, men uten bruk av pekeren *q*. Det er den forrige til *p* som skal fjernes. Dermed må den finnes først. Da må en også ha en annen teknikk for å hindre at *remove* kalles før det har vært et kall på *next*. Bruk f.eks. en variabel removeOK som er usann i starten, som settes til sann etter et kall på *next* og igjen til usann etter et kall på *remove*.
8. Lag en fullstendig iteratorklasse OmvendtInordenIterator i klassen SFBinTre. Der skal det ikke bruken en stakk, men forelderpekere til å traversere. Se også *Oppgave* 5 i *Avsnitt* 5.2.9.

5.2.12 Trær med tråder - tredde trær

Et binærtre kan ha en «tråd». Da kalles det et tredd tre. Se *Avsnitt* 5.1.16. Denne idéen kan også brukes i binære *søketrær*. En «enveistråd» krever en boolsk variabel (f.eks. med navn harHøyreBarn) i hver node. Dette fører imidlertid til at omtrent alle metodene må forandres i forhold til hvordan de er kodet i klassen SBinTre. Spesielt må leggInn-metoden ta hensyn til at en høyrepeker enten peker på et høyre barn eller til den neste i inorden:

```
public boolean leggInn(T verdi)
{
  Node<T> p = rot, q = null;               // pekere
  int cmp = 0;                             // hjelpevariabel

  while (p != null)
  {
    q = p;                                 // q er forelder til p
    cmp = comp.compare(verdi,p.verdi);     // sammenligner
    if (cmp < 0) p = p.venstre;            // går til venstre
    else if (p.harHøyreBarn) p = p.høyre;  // går til høyre
    else break;                            // har ikke høyre barn
  }

  p = new Node<>(verdi,null,null);        // oppretter ny node

  if (q == null)
  {
    rot = p;                               // første node - rotnoden
  }
  else if (cmp < 0)                        // verdi < q.verdi
  {
    q.venstre = p;                         // p blir venstre barn til q
    p.høyre = q;                           // q er den neste til p
  }
  else                                     // verdi >= q.verdi
  {
    p.høyre = q.høyre;                     // p's neste settes lik q's neste
    q.høyre = p;                           // p blir høyre barn til q
    q.harHøyreBarn = true;                 // q har fått et høyre barn
  }

  endringer++;                             // en endring i treet
  antall++;                                // en ny verdi i treet

  return true;                             // vellykket innlegging
}
```

Programkode* 5.2.12 *a)

Idéen er altså slik: Hvis harHøyreBarn er sann i en node *p*, så peker p.høyre til nodes høyre barn, dvs. at *p* har et ikke tomt høyre subtre. Hvis derimot harHøyreBarn er usann, så har ikke *p* et høyre barn. Da peker isteden p.høyre til den neste noden i inorden. Spesielt vil da harHøyreBarn være usann og p.høyre være null hvis *p* er den siste i inorden.

Det at treet har en «tråd» kan brukes til å gjøre en inorden traversering uten rekursjon og uten bruk av en stakk. F.eks. kan toString kodes ved å gjøre en slik traversering:

```
public String toString()
{
  if (tom()) return "[]";                    // et tomt tre
  StringBuilder s = new StringBuilder();     // oppretter en StringBuilder
  s.append('[');                             // startparentes: [

  Node<T> p = rot;                           // starter i roten
  while (p.venstre != null) p = p.venstre;   // finner første i inorden
  s.append(p.verdi);                         // legger inn minste verdi

  while (true)
  {
    if (p.harHøyreBarn)                      // har p høyre barn?
    {
      p = p.høyre;                           // p har høyre barn - går dit
      while (p.venstre != null)
      {
        p = p.venstre;                       // går videre til venstre
      }
    }
    else if (p.høyre == null) break;         // p er sist i inorden
    else p = p.høyre;                        // går til neste i inorden

    s.append(',').append(' ').append(p.verdi);   // legger inn p.verdi
  }

  s.append(']');                             // avslutningsparentes: ]

  return s.toString();                       // returnerer tegnstrengen
}
```

Programkode* 5.2.12 *b)

Vi kaller klassen for `STBinTre` - S for søketre og T for tredd. Hvis nodeklassen, variabler, konstruktører og metodene over er lagt inn i klassen (se *Oppgave* 1), vil flg. program virke:

```
int[] a = {4,8,3,1,7,4,9,1,6,10,2,1,5,10,7,8};    // tabell med duplikater
STBinTre<Integer> tre = STBinTre.stbintre();      // oppretter et tomt tre

for (int k : a) tre.leggInn(k);                   // bygger treet
System.out.println(tre);                          // skriver ut treet

// Utskrift: [1, 1, 1, 2, 3, 4, 4, 5, 6, 7, 7, 8, 8, 9, 10, 10]
```

Programkode* 5.2.12 *c)

Oppgaver til Avsnitt 5.2.12

1. Klassen `STBinTre` inneholder nodeklassen, instansvariabler, aktuelle konstruktører og `leggInn`- og `toString`-metodene. `STBinTre` skal implementere `Beholder`. De aktuelle metodene er satt opp. De som enkelt lar seg kode, er kodet, mens de andre er satt opp med «dummy» kode. Flytt klassen over til deg og sjekk at *Programkode* 5.2.12 *c)* virker.
2. Lag metoden `public boolean inneholder(T verdi)` i klassen `STBinTre`. Bruk idéen fra første del av `leggInn`-metoden.
3. Lag metoden `public boolean fjern(T verdi)` i klassen `STBinTre`. Her må en passe nøye på at «tråden» blir korrekt etter fjerningen.

4. Lag den indre klassen `InordenIterator` i `STBinTre`. Ta utgangspunkt i hvordan den er laget i klassen `SBinTre`. Metoden `next` og `remove` må omkodes. I `next` skal «tråden» brukes for å flytte *p* til den neste i inorden. I `remove` må en passe på at «tråden» ikke brytes.
5. Utvid klassen `STBinTre` slik at det går «tråder» i begge retninger. Se *Oppgave* 4 i *Avsnitt* 5.1.16. Lag så den indre klassen `OmvendtInordenIterator` og metoden `public Iterator<T> iterator()`.

5.2.13 Binære søketrær med mange like verdier

Hvis vi skal lage et binært søketre der det inngår mange like verdier, kan det være aktuelt å ha en annerledes nodestruktur. Vi kan ha en «teller» i hver node. Den angir hvor mange forekomster vi har av nodeverdien:

```
private static final class Node<T>
{
  private T verdi;               // nodens verdi
  private Node<T> venstre;       // venstre barn
  private Node<T> høyre;         // høyre barn
  private int forekomster;       // antall forekomster av verdi

  private Node(T verdi, Node<T> v, Node<T> h)
  {
    this.verdi = verdi;
    venstre = v;
    høyre = h;
    forekomster = 1;
  }

  private Node(T verdi)
  {
    this(verdi,null,null);
  }
}
```

Programkode* 5.2.13 *a)

De fleste metodene må kodes litt annerledes enn i et vanlig binært søktre. I `leggInn` sjekkes det om verdien ligger i treet. Hvis ja, økes `forekomster`. Hvis nei, må det, som i et vanlig binært søketre, opprettes en ny node på rett sortert plass. Tilsvarende blir det for `fjern`-metoden. Variabelen `forekomster` i den noden som inneholder verdien, reduseres. Hvis det var den siste, må den fjernes og da på samme måte som i et vanlig binært søketre.

Oppgaver til Avsnitt 5.2.13

1. Klassen `SMBinTre` (S: søketre, M: mange forekomster) har et «skjelett» av klassen. Det inneholder, i tillegg til den nye noden, de delene som er identisk med klassen `SBinTre`.

   a) Lag metoden `public boolean leggInn(T verdi)` i klassen `SMBinTre`.

   b) Lag metoden `public boolean fjern(T verdi)` i klassen `SMBinTre`.

   c) Lag metoden `public boolean inneholder(T verdi)` i klassen `SMBinTre`.

   d) Lag metoden `public String toString()` i klassen `SMBinTre`. Husk at hvis det er flere forekomster av en verdi, skal alle være med i tegnstrengen.

   e) Lag en iterator i klassen `SMBinTre`. Den skal ta hensyn til antall forekomster.

5.2.14 Trær i java.util

I java.util er det to klasser med ordet *tree* i klassenavnet. Det er TreeSet og er TreeMap. En *map* er en lagringsstruktur der en verdi kobles med en nøkkel (eng: key). F.eks. kan en person (navn) kobles med et nummer:

```
Map<Integer,String> m = new TreeMap<>();

m.put(123,"Ole Olsen");     // 123 er nøkkel, Ole Olsen er verdi
m.put(321,"Kari Jensen");   // 321 er nøkkel, Kari Jensen er verdi

System.out.println(m.get(321) + " " + m.get(123) + " " + m.get(111));

// Utskrift: Kari Jensen Ole Olsen null
```

Programkode* 5.2.14 *a)

Klassen TreeSet er laget ved hjelp av klassen TreeMap. TreeSet er rett og slett nøkkeldelen av TreeMap. En TreeMap (og dermed en TreeSet) består av et binært søketre av en spesiell type - et såkalt *rød-svart* tre. Nodene i en TreeSet inneholder en forelderpeker slik som i vårt SFBinTre - se *Avsnitt* 5.2.11. I tillegg har hver node en farge - rød eller svart. Det er ikke tillatt med duplikater i en TreeSet. Se også *Delkapittel* 9.2. Klassen TreeSet er deklarert slik:

```
public class TreeSet<E> extends AbstractSet<E>
    implements NavigableSet<E>, Cloneable, java.io.Serializable
```

TreeSet arver en klasse og implementerer tre grensesnitt. Det som er mest interessant for oss er grensesnittet NavigableSet som er deklarert ved hjelp av et hierarki som inneholder grensesnittene SortedSet, Set, Collection og Iterable:

```
public interface NavigableSet<E> extends SortedSet<E>

public interface SortedSet<E> extends Set<E>

public interface Set<E> extends Collection<E>

public interface Collection<E> extends Iterable<E>
```

Dette betyr spesielt at en Treemap har en iterator (Iterable) og at iteratoren gir oss verdiene i sortert rekkefølge (SortedSet). Innlegging av verdier skjer ved hjelp av metoden add:

```
int[] a = {4,8,3,1,7,4,9,1,6,10,2,1,5,10,7,8};  // tabell med duplikater

TreeSet<Integer> tre = new TreeSet<>();          // en sortert mengde

for (int k : a) tre.add(k);     // legger inn - duplikater forkastes

for (int k : tre) System.out.print(k + " ");    // bruker iteratoren

// Utskrift: 1 2 3 4 5 6 7 8 9 10
```

Programkode* 5.2.14 *b)

Alle grensesnittene gir at en TreeSet inneholder mange forskjellige metoder. Her setter vi kun opp de som svarer til de vi har diskutert i forbindelse med vårt binære søketre:

```
public class TreeSet<E>
{
  public int size()                        // public int antall()
  public boolean isEmpty()                 // public boolean tom()
  public void clear()                      // public void nullstill()

  public boolean add(E e)                  // public boolean leggInn(E e)

  public boolean contains(Object o)        // public boolean inneholder(E e)
  public String toString()                 // public String toString()

  public boolean remove(Object o)          // public boolean fjern(E e)
  public E pollFirst()                     // public E fjernMin()
  public E pollLast()                      // public E fjernMaks()

  public E first()                         // public E min()
  public E last()                          // public E maks()

  public E floor(E e)                      // public E gulv(E e)
  public E ceiling(E e)                    // public E tak(E e)

  public E lower(E e)                      // public E mindre(E e)
  public E higher(E e)                     // public E større(E e)

  public Iterator<E> iterator()            // public Iterator<E> iterator()
  public Iterator<E> descendingIterator()  // public Iterator<E> riterator()

  // + flere
}
```

Programkode 5.2.14 c)

Oppgaver til Avsnitt 5.2.14

1. Klassen `TreeSet` inneholder også metoder som gir oss «delmengder» (eng: subsets) av treet. Metoden `public SortedSet<E> headSet(E toElement)` gir oss en `SortedSet` som inneholder alle elementene i treet som er mindre enn `toElement`. Lag kode der du tester ut denne metoden.
2. Klassen `TreeSet` inneholder også `public NavigableSet<E> headSet(E toElement, boolean inclusive)`. Hvis `inclusive` er sann, vil delmengden bestå av elementene i treet som er mindre enn eller lik `toElement`. Hvis derimot `inclusive` er usann, blir det mindre enn `toElement`. Lag kode der du tester ut denne metoden.
3. `headSet`-metodene i *Oppgave* 1 og 2 gir kun et «innblikk» eller en peker til delmengden, dvs. metoden returnerer ingen separat kopi. Det betyr at hver endring som etterpå gjøres i den del av treet som vi har «innblikk» i, er synlig gjennom innblikket. Det er også omvendt. Endrer vi via «innblikket», skjer endringen i treet. Lag kode som tester dette.
4. Klassen har også metodene `subSet` og `tailSet` - to versjoner av hver. De virker på tilsvarende måte som `headSet`. Lag kode som tester disse metodene.
5. Sjekk restene av metodene i `TreeSet` og finn ut hva de kan brukes til.

5.2.15 Algoritmeanalyse

Gjennomsnittlige tall for binære søketrær med *n* forskjellige verdier:

Oppgaver til Avsnitt 5.2.15

1. Ingen oppgaver ennå.