DATS2300 / ITPE2300
Algoritmer og Datastrukturer Høst 2019

Oblig 3

Frist: Onsdag 6. november kl. 18

Oppdatert 9. oktober 2019

Det skal sendes inn en fil (navn: **`ObligSBinTre.java`**) som inneholder klassen **`public class ObligSBinTre<T> implements Beholder<T>`**. Grensesnittet `Beholder` er det som er satt opp i *Avsnitt* 3.1.1 i kompendiet. Pass på at alle metodene dine er testet og at de behandler alle spesialtilfellene korrekt.

Det er ok at det under arbeidet med å lage metodene legges inn utskrifts-setninger for testing i metodene. Men de SKAL FJERNES når obligen leveres inn. Klassen `ObligSBinTre` skal heller ikke ha noen main-metode når den leveres inn. Man kan f.eks. ha en egen klasse for testing.

En gruppe på til og med 5 studenter kan levere en felles løsning. Filen **`ObligSBinTre.java`** og resultatet av kjøring av testprogrammet (som en tekstfil). Skriv i tillegg navn og studentnummer i en fil som heter «README.md». Hvis det er en gruppe som leverer, må det stå navn og studentnummer på alle i gruppen. Dere kan bruke samme Github repository for oblig 3 som for oblig 1/2 dersom alle på gruppen fortsatt er med i gruppen.

Det er et krav at gruppen bruker github for å lagre koden og dokumentere sitt arbeid. **Alle studenter som er med i gruppen må dokumentere sitt bidrag via commits til Github**. Oppgaven leveres ved at foreleser og studentassistent inviteres med som samarbeidspartner på Github
(se https://help.github.com/en/articles/inviting-collaborators-to-a-personal-repository for detaljer hvordan dette kan gjøres).

Huskeliste før du sender inn løsningen:

- Står navn/studentnummer på alle i gruppen øverst på filen `ObligSBinTre.java`?
- Har du laget filen readme.md med navn på alle på gruppen?
- Har du lagt ved resultatet av kjøring av testprogrammet (som en tekstfil)?
- Unngå å bruke norske tegn (æ, ø, å) i kildekode og kommentarer
- Er alle utskriftssetninger inne i metodene fjernet?
- Er en eventuell main-metode inne i `class ObligSBinTre`?
- Gir din IDE (NetBeans, Eclipse, IntelliJ eller hva annet du måtte bruke) noen advarsler (warnings)? Det må du unngå! Gjør de nødvendige endringene/tilleggene i koden din!
- Er alle metodene testet? De må passere testprogrammet før løsningen sendes inn.

Hvis det som leveres inn som 3. oblig ikke blir godkjent, vil en få det i retur med krav om at det må forbedres. Med andre ord får en flere sjanser.

`ObligSBinTre` er et binært søketre av samme type som beskrevet i *Delkapittel* 5.2. Men `ObligSBinTre` har litt mer struktur. En node har i tillegg til referanser til venstre og

høyre barn, en referanse til dens forelder. I skjelettet er en del metoder ferdigkodet og de andre kaster en `UnsupportedOperationException`.

Hvis du i din løsning bruker noen av de strukturene som er laget i undervisningen (f.eks. en liste, en stakk, en kø eller noe annet), skal ***ikke*** koden for dem følge med løsningen.

Obs: De metodene i `ObligSBinTre` som skal kodes, kan kodes på mange forskjellige måter. I flere av oppgavene blir det bedt om at en metode skal kodes på en bestemt måte. Det er ikke fordi det nødvendigvis er den beste måten. Men poenget er å lære kodeteknikk, dvs. den teknikken som det bes om.

`ObligSBinTre` har variabelen *endringer* og `BladnodeIterator` har *iteratorendringer*. De skal fungere på samme måte her som i klassen `DobbeltLenketListe` fra Oblig 2.

0. Innledning: Legg klassen `ObligSBinTre` inn i ditt Java-prosjekt. For at det skal virke må grensesnittet `Beholder` være tilgjengelige. Lag så noen instanser av klassen `ObligSBinTre`. Sjekk at det ikke gir noen syntaksfeil (eller kjørefeil). Bruk f.eks. både `Integer`, `Character` og `String` som datatyper. Da kan du bruke en «naturlig» komparator i konstruktøren. Dvs. slik for datatypen `String`:

```
ObligSBinTre<String> tre = new ObligSBinTre<>(Comparator.naturalOrder());
System.out.println(tre.antall());  // Utskrift: 0
```

1. En `Node` i `ObligSBinTre` har i tillegg til referanser til venstre og høyre barn, en referanse til nodens forelder. Denne må få riktig verdi ved hver innlegging. Spesielt skal forelder være null i rotnoden. Lag metoden *public boolean leggInn(T verdi)*. Der kan du kopiere *Programkode 5.2.3 a)*, men i tillegg må du gjøre de endringene som trengs for at referansen *forelder* får korrekt verdi i hver node. Teknikken med en forelder-referanse brukes f.eks. i klassen `TreeSet` i `java.util`. Sjekk at flg. kode er feilfri (kaster ingen unntak):

```
Integer[] a = {4,7,2,9,5,10,8,1,3,6};
ObligSBinTre<Integer> tre = new ObligSBinTre<>(Comparator.naturalOrder());
for (int verdi : a) tre.leggInn(verdi);
System.out.println(tre.antall());  // Utskrift: 10
```

2. Metodene *inneholder()*, *antall()* og *tom()* er ferdigkodet. Den første avgjør om en verdi ligger i treet eller ikke. De to andre fungerer på vanlig måte. Lag kode for metoden *public int antall(T verdi)*. Den skal returnere antall forekomster av `verdi` i treet. Det er tillatt med duplikater og det betyr at en verdi kan forekomme flere ganger. Hvis `verdi` ikke er i treet (*null* er ikke i treet), skal metoden returnere 0. Test koden din ved å lage trær der du legger inn flere like verdier. Sjekk at metoden din da gir rett svar. Her er ett eksempel:

```
Integer[] a = {4,7,2,9,4,10,8,7,4,6};
ObligSBinTre<Integer> tre = new ObligSBinTre<>(Comparator.naturalOrder());
for (int verdi : a) tre.leggInn(verdi);

System.out.println(tre.antall());      // Utskrift: 10
System.out.println(tre.antall(5));     // Utskrift: 0
System.out.println(tre.antall(4));     // Utskrift: 3
System.out.println(tre.antall(7));     // Utskrift: 2
System.out.println(tre.antall(10));    // Utskrift: 1
```

3. Lag hjelpemetoden *private static <T> Node<T> nesteInorden(Node<T> p)*. Siden dette er en privat metode, tas det som gitt at parameteren *p* ikke er *null*. Det er når metoden brukes at en må sikre seg at det ikke går inn en nullreferanse. Den skal returnere den noden som kommer etter *p* i ***inorden***. Hvis *p* er den siste i inorden, skal metoden returne *null*. Husk at hvis *p* har et høyre subtre, så vil den neste i inorden være den noden som ligger lengst ned til venstre i det subtreet. Hvis *p* ikke har et høyre subtre og *p* ikke er den siste, vil den neste i inorden være høyere opp i treet. Den finner du ved hjelp forelder-referansene. Lag så metoden *public String toString()*. Den skal returnere en tegnstreng med treets verdier i inorden. Verdiene skal rammes inn av [og]. Mellom verdiene (hvis det er flere) skal det være komma og mellomrom. Et tomt tre skal "[]". Du skal bruke **verken** rekursjon eller hjelpestakk. Du **skal** bruke hjelpemetoden *nesteInorden()*. Start med å finne den første noden *p* i inorden. Deretter vil (f.eks. i en while-løkke) setningen: *p = nesteInorden(p);* gi den neste. Osv. til *p* blir *null*.

Et eksempel på hvordan det skal virke:

```
int[] a = {4,7,2,9,4,10,8,7,4,6,1};
ObligSBinTre<Integer> tre = new ObligSBinTre<>(Comparator.naturalOrder());
for (int verdi : a) tre.leggInn(verdi);

System.out.println(tre);  // [1, 2, 4, 4, 4, 6, 7, 7, 8, 9, 10]
```

4. Lag metoden *public String omvendtString()*. Den skal gjøre som metoden *toString()*, men med verdiene i motsatt rekkefølge. Du skal løse dette ved å traversere treet i omvendt inorden (dvs. motsatt vei av inorden) iterativt. Her ***skal*** du bruke en hjelpestakk (og ikke rekursjon). F.eks. en `TabellStakk` eller en stakk fra java.util (f.eks. en `ArrayDeque`). Koden din skal ikke noe sted benytte forelderpekerne. Med andre ord skal koden din også kunne virke i et binærtre uten forelderpekere. Ta f.eks. utgangspunkt i den iterative inorden-metoden fra *Programkode* 5.1 10 e). Men i denne oppgaven skal traverseringen gå motsatt vei. Det betyr at du må gjøre noen endringer for å få det til å gå den motsatte veien.

Et eksempel på hvordan det skal virke:

```
int[] a = {4,7,2,9,4,10,8,7,4,6,1};
ObligSBinTre<Integer> tre = new ObligSBinTre<>(Comparator.naturalOrder());
for (int verdi : a) tre.leggInn(verdi);

System.out.println(tre.omvendtString());  // [10, 9, 8, 7, 7, 6, 4, 4, 4, 2, 1]
```

5. Lag metoden *public boolean fjern(T verdi)*. Der kan du kopiere *Programkode* 5.2 8 d), men i tillegg må du gjøre de endringene som trengs for at pekeren *forelder* får korrekt verdi i alle noder etter en fjerning. Lag så metoden *public int fjernAlle(T verdi)*. Den skal fjerne alle forekomstene av *verdi* i treet. Husk at duplikater er tillatt. Dermed kan en og samme verdi ligge flere steder i treet. Metoden skal returnere antallet som ble fjernet. Hvis treet er tomt, skal 0 returneres. Lag så metoden *public void nullstill()*. Den skal traversere (rekursivt eller iterativt) treet i **en** eller annen rekkefølge og sørge for at samtlige pekere og nodeverdier i treet blir nullet. Det er med andre ord ikke tilstrekkelig å sette *rot* til *null* og *antall* til 0.

Et eksempel på hvordan det skal virke:

```
int[] a = {4,7,2,9,4,10,8,7,4,6,1};
```

```
ObligSBinTre<Integer> tre = new ObligSBinTre<>(Comparator.naturalOrder());
for (int verdi : a) tre.leggInn(verdi);

System.out.println(tre.fjernAlle(4));  // 3
tre.fjernAlle(7); tre.fjern(8);

System.out.println(tre.antall());  // 5

System.out.println(tre + " " + tre.omvendtString());
// [1, 2, 6, 9, 10] [10, 9, 6, 2, 1]
// OBS: Hvis du ikke har gjort oppgave 4 kan du her bruke toString()
```

6. En *gren* i treet består av nodene på veien fra og med rotnoden ned til og med en bladnode. Det betyr at treet har like mange grener som bladnoder. Her skal vi lage de to metodene *public String høyreGren()* og *public String lengstGren()*. Begge skal returnere en tegnstreng med grenens verdier. Tegnstrengen skal som vanlig være «innrammet» av hakeparentesene [og]. Verdiene skal (hvis det er flere) være adskilt med komma og mellomrom. Hvis treet er tomt (dvs. ingen grener) skal kun "[]" returneres. a) Metoden *public String høyreGren()* skal gi den grenen som ender i den bladnoden som ligger lengst til høyre i treet. b) Metoden *public String lengstGren()* skal gi den lengste grenen, dvs. grenen som ender i den bladnoden som ligger lengst ned i treet. (Obs: En gren ender alltid i en bladnode). Hvis det er flere lengste grener, skal den av dem som ligger lengst til venstre, returneres Pass på at hvis treet har kun én gren, så er denne grenen både høyre gren og lengste gren. Hvis treet har kun én node (kun rotnoden), er dette også en gren. Flg. kodebit viser hvordan dette skal virke (Hint: Tegn treet først. Da kan du sammenligne utskriften med tegningen din):

```
ObligSBinTre<Character> tre = new ObligSBinTre<>(Comparator.naturalOrder());
char[] verdier = "IATBHJCRSOFELKGDMPQN".toCharArray();
for (char c : verdier) tre.leggInn(c);

System.out.println(tre.høyreGren() + " " + tre.lengstGren());

// Utskrift: [I, T, J, R, S] [I, A, B, H, C, F, E, D]
```

7. Lag metoden *public String[] grener()*. Den skal returnere en `String`-tabell (dvs. `String[]`) som inneholder (som tabellelementer) alle grenene i treet i rekkefølge fra venstre mot høyre. Hvis treet er tomt skal det returneres en tom tabell. Hvis du utvider kodeeksemplet fra Oppgave 6 med flg. kode, skal du få en tilleggsutskrift som nedenfor:

```
String[] s = tre.grener();

for (String gren : s) System.out.println(gren);

// Utskrift:
// [I, A, B, H, C, F, E, D]
// [I, A, B, H, C, F, G]
// [I, T, J, R, O, L, K]
// [I, T, J, R, O, L, M, N]
// [I, T, J, R, O, P, Q]
// [I, T, J, R, S]
```

8. a) Lag metoden *public String bladnodeverdier()*. Den skal returnere en tegnstreng med verdiene i bladnodene. Tegnstrengen skal se ut som vanlig (med [og] og med komma

og mellomrom). Her skal du bruke en ***rekursiv*** hjelpemetode som traverserer treet. Bladnodeverdiene skal i tegnstrengen stå i rekkefølge fra venstre mot høyre. Hvis du utvider kodeeksemplet fra Oppgave 6 med flg. kode, skal du få en tilleggsutskrift som nedenfor:

```
System.out.println(tre.bladnodeverdier());
// Utskrift: [D, G, K, N, Q, S]
```

8. b) Lag metoden *public String postString()*. Den skal returnere en tegnstreng med treets verdier i ***postorden***. Tegnstrengen skal se ut som vanlig (med [og] og med komma og mellomrom). Her ***skal*** du lage en iterativ løsning (dvs. ikke rekursjon). Hva slags iterativ løsning du velger er opp til deg. Pass på at de løsning virker som i eksemplene under (Hint: Tegn trærne først. Da vil du kunne se hva svaret skal bli):

Treet fra eksempelet i Oppgave 6:

```
System.out.println(tre.postString());
// [D, E, G, F, C, H, B, A, K, N, M, L, Q, P, O, S, R, J, T, I]
```

Et annet eksempel:

```
int[] a = {4,7,2,9,4,10,8,7,4,6};
ObligSBinTre<Integer> tre = new ObligSBinTre<>(Comparator.naturalOrder());
for (int verdi : a) tre.leggInn(verdi);

System.out.println(tre.postString());  // [2, 6, 4, 4, 7, 8, 10, 9, 7, 4]
```

9. Klassen `BladnodeIterator` er satt opp med instansvariabler og konstruktør. Det er fullt mulig (og ikke vanskelig) å løse alt det spørres om uten at det legges inn flere intansvariabler i iteratorklassen. Men hvis du skulle mene at det er lettere å få til dette ved å bruke en hjelpestakk, så må du gjerne gjøre det. Metoden `iterator()` er ferdigkodet. Lag konstruktøren private BladnodeIterator() og metoden *public T next()*. Konstruktøren skal sørge for å flytte pekeren *p* til første bladnode, dvs. til den som er lengst til venstre hvis det er flere bladnoder. Hvis treet er tomt, skal ikke *p* endres. Metoden *next()* skal kaste en *NoSuchElementException* hvis det ikke er flere bladnoder igjen. Hvis ikke, skal den returnere en ***bladnodeverdi***. Bladnodeverdiene skal komme i rekkefølge fra venstre mot høyre. Husk også på at *endringer* og *iteratorendringer* skal brukes som i Oblig 2. Metoden *hasNext()*er ferdigkodet og skal ikke endres. Hvis du utvider kodeeksemplet fra Oppgave 6 med flg. kode, skal du få en tilleggsutskrift som nedenfor:

```
// En for-alle-løkke bruker iteratoren implisitt
for (Character c : tre) System.out.print(c + " ");  // D G K N Q S
```

Hvis du bruker det andre eksempelet i Oppgave 8b):

```
for (Integer k : tre) System.out.print(k + " ");  // 2 6 7 10
```

10. Lag metoden *public void remove()* i klassen `BladnodeIterator`. Pass på at metoden `next()` setter variablen `removeOK` til *true* og at `remove()` setter den til *false*. Pekeren *q* skal ligge én bak *p*. Dvs. at når *p* i metoden *next()* flyttes til neste bladnode (eller til null hvis det var den siste), skal *q* peke på den som *p* pekte på. Med andre ord er det noden som *q* peker på som skal fjernes. Det skal gjøres med direkte kode. Metoden *fjern()* i klassen`ObligSBinTre` kan ikke brukes her fordi verdien i noden *q* kan også ligge et sted høyere opp i treet. Pass på at metoden kaster

en *IllegalStateException* hvis det er ulovlig å kalle den. Husk også på at *endringer* og *iteratorendringer* skal brukes som i Oblig 2. Hvis du utvider kodeeksemplet fra Oppgave 6 med flg. kode, skal du få en tilleggsutskrift som nedenfor:

```
while (!tre.tom())
{
  System.out.println(tre);
  tre.fjernHvis(x -> true);
}
/*
[A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T]
[A, B, C, E, F, H, I, J, L, M, O, P, R, T]
[A, B, C, F, H, I, J, L, O, R, T]
[A, B, C, H, I, J, O, R, T]
[A, B, H, I, J, R, T]
[A, B, I, J, T]
[A, I, T]
[I]
*/
```

Hvis du bruker det andre eksempelet i Oppgave 8b):

```
/*
[2, 4, 4, 4, 6, 7, 7, 8, 9, 10]
[4, 4, 4, 7, 8, 9]
[4, 4, 7, 9]
[4, 7]
[4]
*/
```