Algoritmer og Datastrukturer Høst 2021

Oblig 2

Frist: Fredag 8. oktober 2021 klokken 20:00

Det skal leveres et github repository som inneholder klassen **`public class DobbeltLenketListe<T> implements Liste<T>`** som svarer på oppgavene. De mer teoretiske spørsmålene skal besvares i en javadoc-kommentar foran den tilhørende metoden og i Readme-filen. Pass på at metodene dine er testet og at de behandler alle spesialtilfellene korrekt. Klassen skal være selvforsynt med metoder. Hvis du bruker en metode fra undervisningen, må den kopieres inn i klassen **`Oblig2`**.

En gruppe på til og med 5 studenter kan levere en felles løsning. Ta utgangspunkt i utlevert kildekode, og fullfør oppgavene i DobbeltLenketListe.java. Skriv i tillegg navn og studentnummer i filen som heter «README.md», samt beskrivelse av obligen. Beskriv (veldig kort) hvordan hver deloppgave er løst der. Hvis det er en gruppe som leverer, må det stå navn og studentnummer på alle i gruppen. Gruppestørrelsen bestemmer hvor mange oppgaver man må utføre for å få godkjent (merk at alle oppgaver kan bli spurt om på eksamen)

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**. Oppgavene leveres via github classroom (se lenke under).

Invitasjon til prosjekt på Github:
https://classroom.github.com/g/Y8vnoWEd

Krav til innlevering (hvis dette ikke oppfylles blir den ikke godkjent):

1. Git er brukt for å dokumentere arbeid med obligen. Du må ha
    a. Minst to commits per oppgave spredt over tid (ikke lov å committe alt rett før innleveringsfristen!)
    b. Beskrivende commit-meldinger, f.eks.
        i. "Laget første prototype med kildekodekommentarer over hvordan jeg har tenkt til å løse oppgave 1."
        ii. "Implementerte oppgave 1 slik at den passerer test 1a og 1b. Test 1c feiler fortsatt"
        iii. "Implementerte oppgave 1 slik at den passerer test 1c"
2. Alle filene ligger som levert ut - ingen av filene skal flyttes på.
3. Beskrivelse av hvordan oppgaven er løst (4-8 linjer/setninger per oppgave) står i Readme.md.
4. Ingen main-metode eller debug-utskrifter (system.out.println) når den leveres inn.
5. Warnings er enten beskrevet i readme.md eller fjernet.
6. Alle oppgavene består den utleverte testen.

    a. Alene: Oppgave 1, 2, 3, 4, 5, 6, 8. (fjern test av oppgave 7, 9, 10)
    b. To: Oppgave 1, 2, 3, 4, 5, 6, 7, 8. (fjern test av oppgave 9 og 10)
    c. Tre til fem: Oppgave 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. (alle tester skal kjøre)

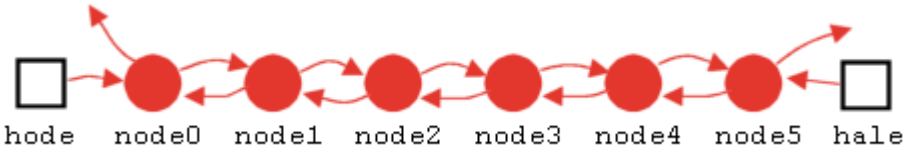
7. Du må skrive hva hver person på gruppen har gjort i innleveringen i readme.md

OBS: Hvis det som leveres inn som 2. oblig ikke blir godkjent, vil en få det i retur med krav om at det må forbedres.

Hint: Oppgavene er veldig mye lettere å løse om man lager tegning på papir før man begynner implementere.

Oppgaven:

En toveis liste (dobbelt lenket) har et hode og en hale. Hode peker til den første noden og hale til den siste slik som denne figuren viser:



En node har både en forrige-peker og en neste-peker. Neste-peker i den siste noden og forrige-peker i den første noden er begge null.

I kildekoden står et skjelett av klassen `DobbeltLenketListe` og dens metoder. Et par av dem er kodet. I de øvrige kastes det en `UnsupportedOperationException` som signal om at de ikke er kodet. Det gjør at kompilatoren ikke gir noen feilmeldinger, men alle testene feiler. Alle disse metodene skal kodes.

Klassen inneholder instansvariablene *antall* og *endringer*. Variabelen *antall* skal økes med 1 for hver innlegging og reduseres med 1 for hver fjerning. Variabelen *endringer* skal økes med 1 for hver endring i listen (innlegging, oppdatering, fjerning, nullstill). Dvs. for alle mutatorer. Hensikten med *endringer* tas opp i forbindelse med kodingen av iteratoren.

Arbeidet, kodingen og testingen bør skje i følgende rekkefølge:

Oppgave 1

Lag metodene `int antall()` og `boolean tom()`. Den første skal returnere antallet verdier i listen og den andre skal returnere true/false avhengig av om listen er tom eller ikke. Sjekk så at følgende programbit gir rett utskrift:

```
Liste<String> liste = new DobbeltLenketListe<>();
System.out.println(liste.antall() + " " + liste.tom());

// Utskrift: 0 true
```

Lag så konstruktøren `public DobbeltLenketListe(T[] a)`. Den skal lage en dobbeltlenket liste med verdiene fra tabellen *a*. Verdiene skal ligge i samme rekkefølge i listen som i tabellen. Hvis *a* er null, skal det kastes en `NullPointerException` med teksten "Tabellen a er null!" (bruk f.eks. en *requireNonNull*-metode fra klassen `Objects`). Hvis *a* inneholder en eller flere null-verdier, skal de ikke tas med. Dvs. at listen skal inneholde de verdiene fra *a* som ikke er null. Hvis alle verdiene i *a* er null, får vi en tom liste.

Her må du passe på at *hode* peker til den første i listen og *hale* til den siste. Pass også på at *neste* og *forrige* er satt riktig i alle noder. Spesielt skal både *hode.forrige* og *hale.neste* være null. Pass også på at hvis tabellen *a* har kun én verdi som ikke er null (listen får da én node), så vil *hode* og *hale* peke til samme node. Hvis *a* er tom (lengden er 0), skal det ikke opprettes noen noder og dermed at *hode* og *hale* fortsatt er null (dvs. en tom liste). Variabelen *endringer* skal være 0.

Her kan det være fristende å bygge opp listen ved gjentatte kall på metoden *leggInn*() (den skal kodes i *Oppgave* 2). Men konstruktøren skal kodes **direkte** (uten bruk av *leggInn*()). Årsaken er at hvis en senere skal lage en

subklasse av `DobbeltLenketListe` og overskriver (overrides) *`leggInn`*`()`, så kan det føre til at konstruktøren ikke lenger virker som den skal.

Sjekkliste for konstruktøren `DobbeltLenketListe(T[])`:

- Stoppes en null-tabell? Kastes i så fall en `NullPointerException`?
- Blir det korrekt hvis parametertabellen inneholder en eller flere null-verdier?
- Blir det korrekt hvis parametertabellen er tom (har lengde 0)?
- Blir det korrekt hvis parametertabellen kun har null-verdier?
- Blir det korrekt hvis parametertabellen har kun én verdi som ikke er null?
- Blir antallet satt korrekt?
- Får verdiene i listen samme rekkefølge som i tabellen?

Det er så langt ikke enkelt å få testet om konstruktøren lager en korrekt toveisliste. Det kan først gjøres når metodene *`toString`*`()` og *`omvendtString`*`()` er ferdiglaget i *Oppgave* 2. Men følgende programbit skal i hvert fall gi rett utskrift:

```
String[] s = {"Ole", null, "Per", "Kari", null};
Liste<String> liste = new DobbeltLenketListe<>(s);
System.out.println(liste.antall() + " " + liste.tom());

// Utskrift: 3 false
```

Oppgave 2

a) Lag først metoden `String toString()`. Den skal returnere en tegnstreng med listens verdier. Hvis listen f.eks. inneholder tallene 1, 2 og 3, skal metoden returnere strengen "[1, 2, 3]" og kun "[]" hvis listen er tom. Du skal bruke en `StringBuilder` (eller en `StringJoiner`) til å bygge opp tegnstrengen og verdiene i listen finner du ved å traversere fra hode til hale ved hjelp av neste-pekere. Lag så metoden `String omvendtString()`. Den skal returnere en tegnstreng på samme form som den `toString()` gir, men verdiene skal komme i omvendt rekkefølge. Her skal du finne verdiene i omvendt rekkefølge ved å traversere fra hale til hode ved hjelp av forrige-pekere. Hensikten med *`omvendtString`*`()` er å kunne sjekke at forrige-pekerne er satt riktig.

Sjekk at følgende programbit gir rett utskrift (forutsetter at Oppgave 1 er ferdig):

```
String[] s1 = {}, s2 = {"A"}, s3 = {null,"A",null,"B",null};
DobbeltLenketListe<String> l1 = new DobbeltLenketListe<>(s1);
DobbeltLenketListe<String> l2 = new DobbeltLenketListe<>(s2);
DobbeltLenketListe<String> l3 = new DobbeltLenketListe<>(s3);

System.out.println(l1.toString() + " " + l2.toString()
  + " " + l3.toString() + " " + l1.omvendtString() + " "
  + l2.omvendtString() + " " + l3.omvendtString());

// Utskrift: [] [A] [A, B] [] [A] [B, A]
```

b) Lag metoden `boolean leggInn(T verdi)`. Null-verdier er ikke tillatt. Start derfor med en sjekk (bruk en `requireNonNull`-metode fra klassen `Objects`). Innleggingsmetoden skal legge en ny node med oppgitt *`verdi`* bakerst i listen og returnere true. Her må du skille mellom to tilfeller:
1) at listen på forhånd er tom og

2) at den ikke er tom.
I en tom liste skal både hode og hale være null (og antall lik 0). I tilfelle 1) skal både hode og hale etter innleggingen peke på den nye noden (både forrige-peker og neste-peker i noden skal da være null). I tilfelle 2) er det kun hale-pekeren som skal endres etter innleggingen. Pass da på at forrige-peker og neste-peker i den nye noden og i den noden som opprinnelig lå bakerst, får korrekte verdier. Husk at antallet må økes etter en innlegging. Det samme med variabelen *endringer*. Metoden skal returnere `true`.

Sjekkliste for metoden *leggInn(T verdi)*:

- Stoppes null-verdier? Kastes i så fall en `NullPointerException`?
- Blir det korrekt hvis listen fra før er tom?
- Blir det korrekt hvis listen fra før ikke er tom?
- Blir antallet økt?
- Blir endringer økt?
- Er det rett returverdi?

Sjekk så at følgende programbit gir rett utskrift:

```
DobbeltLenketListe<Integer> liste = new DobbeltLenketListe<>();
System.out.println(liste.toString() + " " + liste.omvendtString());
for (int i = 1; i <= 3; i++) {
  liste.leggInn(i);
  System.out.println(liste.toString() + " " + liste.omvendtString());
}

// Utskrift:
// [] []
// [1] [1]
// [1, 2] [2, 1]
// [1, 2, 3] [3, 2, 1]
```

Oppgave 3

a) Lag den private hjelpemetoden `Node<T> finnNode(int indeks)`. Den skal returnere noden med den gitte indeksen/posisjonen. Hvis indeks er mindre enn *antall*/2, så **skal** letingen etter noden starte fra hode og gå mot høyre ved hjelp av neste-pekere. Hvis ikke, **skal** letingen starte fra halen og gå mot venstre ved hjelp av forrige-pekere. Lag deretter metoden `public T hent(int indeks)` ved å bruke *finnNode*(). Pass på at `indeks` sjekkes. Bruk metoden *indeksKontroll*() som arves fra `Liste` (bruk *false* som parameter). Lag også metoden `T oppdater(int indeks, T nyverdi)`. Den skal erstatte verdien på plass *indeks* med *nyverdi* og returnere det som lå der fra før. Husk at indeks må sjekkes, at null-verdier ikke skal kunne legges inn og at variabelen *endringer* skal økes.

b) Lag metoden `Liste<T> subliste(int fra, int til)`. Den skal returnere en liste (en instans av klassen `DobbeltLenketListe`) som inneholder verdiene fra intervallet [fra:til> i «vår» liste. Her må det først sjekkes om indeksene *fra* og *til* er lovlige. Hvis ikke, skal det kastes unntak slik som i metoden *fratilKontroll*(). Legg derfor den inn som en privat metode i klassen `DobbeltLenketListe` og bytt ut `ArrayIndexOutOfBoundsException` med `IndexOutOfBoundsException` siden vi ikke har noen tabell (array) her. Bytt også ut ordet *tablengde* med ordet *antall*. Denne kontrollmetoden

kan da kalles med *antall*, *fra* og *til* som argumenter. Husk at et tomt intervall er lovlig. Det betyr at vi får en tom liste.

Pass på at variablen *antall* i den listen som metoden *subliste*() returnerer, får korrekt verdi. Variabelen *endringer* skal være 0. Her kan alle metoder brukes - også *leggInn*().

Sjekk så at følgende programbit gir rett utskrift:

```
Character[] c = {'A','B','C','D','E','F','G','H','I','J',};
DobbeltLenketListe<Character> liste = new DobbeltLenketListe<>(c);
System.out.println(liste.subliste(3,8));  // [D, E, F, G, H]
System.out.println(liste.subliste(5,5));  // []
System.out.println(liste.subliste(8,liste.antall()));  // [I, J]
// System.out.println(liste.subliste(0,11));  // skal kaste unntak
```

Oppgave 4

Lag metoden `int indeksTil(T verdi)`. Den skal returnere indeksen/posisjonen til *verdi* hvis den finnes i listen og returnere -1 hvis den ikke finnes. Her skal det ikke kastes unntak hvis `verdi` er `null`. Metoden skal isteden returnere -1. Det er logisk siden `null` ikke finnes i listen. Hvis `verdi` forekommer flere ganger, skal indeksen til den første av dem (fra venstre) returneres. Lag så metoden `boolean inneholder(T verdi)`. Den skal returnere true hvis listen inneholder *verdi* og returnere false ellers. Her lønner det seg å bruke et kall på metoden `indeksTil` som en del av koden.

Oppgave 5

Lag metoden `void leggInn(int indeks, T verdi)`. Den skal legge *verdi* inn i listen slik at den får indeks/posisjon *indeks*. Husk at negative indekser og indekser større enn antall er ulovlige (indekser fra og med 0 til og med antall er lovlige). Her må du passe på de fire tilfellene 1) listen er tom, 2) verdien skal legges først, 3) verdien skal legges bakerst og 4) verdien skal legges mellom to andre verdier. Sørg for at neste- og forrige-pekere får korrekte verdier i alle noder. Spesielt skal forrige-peker i den første noden være null og neste-peker i den siste noden være null.

Sjekkliste for `leggInn(int indeks, T verdi)`:

- Stoppes null-verdier? Kastes i så fall en `NullPointerException`?
- Sjekkes indeksen?
- Blir det korrekt hvis listen fra før er tom?
- Blir pekerne (forrige og neste) korrekte i alle noder hvis ny verdi legges først?
- Blir pekerne (forrige og neste) korrekte i alle noder hvis ny verdi legges bakerst?
- Blir pekerne (forrige og neste) korrekte i alle noder hvis ny verdi legges mellom to verdier?
- Blir *antall* økt?
- Blir *endringer* økt?

Oppgave 6

Lag de to fjern-metodene, dvs. `T fjern(int indeks)` og `boolean fjern(T verdi)`. Den første skal fjerne (og returnere) verdien på posisjon *indeks* (som først må sjekkes). Den andre skal fjerne *verdi* fra listen og så returnere true. Hvis det er flere forekomster

av *verdi*er det den første av dem (fra venstre) som skal fjernes. Hvis *verdi* ikke er i listen, skal metoden returnere false. Her skal det ikke kastes unntak hvis *verdi* er `null`. Metoden skal isteden returnere false. Det er logisk siden `null` ikke finnes i listen. Begge metodene skal være så effektive som mulig. Her kan det være fristende å kode `fjern(T verdi)` ved hjelp av `indeksTil(T verdi)` og `fjern(int indeks)`. Men det fører til to gjennomganger av listen og er dermed ineffektivt. Derfor skal `fjern(T verdi)` kodes direkte.

I begge metodene må du passe på tilfellene 1) den første fjernes, 2) den siste fjernes og 3) en verdi mellom to andre fjernes. Alle neste- og forrige-pekere må være korrekte etter fjerningen. Variabelen *antall* skal også reduseres og variabelen *endringer* økes. Sjekk også tilfellet at listen blir tom etter fjerningen, blir korrekt behandlet. Bruk metodene `toString()`og `omvendtString()` til å sjekke at alle pekerne er satt riktig.

Sjekkliste for `fjern`-metodene:

- Blir det korrekt hvis listen fra før er tom?
- Blir pekerne (forrige og neste) korrekte i alle noder hvis første verdi (indeks 0) fjernes?
- Blir pekerne (forrige og neste) korrekte i alle noder hvis siste verdi fjernes?
- Blir pekerne (forrige og neste) korrekte i alle noder hvis det fjernes en verdi mellom to verdier?
- Blir pekerne (forrige og neste) korrekte hvis listen etter fjerningen får kun én verdi? Hva med ingen verdier?
- Blir *antall* redusert?
- Blir *endringer* økt?

Oppgave 7

Lag metoden `void nullstill()`. Den skal «tømme» listen og nulle alt slik at «søppeltømmeren» kan hente alt som ikke lenger brukes. Kod den på to måter og velg den som er mest effektiv (gjør tidsmålinger):

1. måte: Start i *hode* og gå mot *hale* ved hjelpe pekeren *neste*. For hver node «nulles» nodeverdien og alle nodens pekere. Til slutt settes både *hode* og *hale* til null, *antall* til 0 og *endringer* økes. Hvis du er i tvil om hva som det bes om her, kan du slå opp i kildekoden for metoden *clear*() i klassen `LinkedList` i Java.

2. måte: Lag en løkke som inneholder metodekallet *fjern*(0) (den første noden fjernes) og som går inntil listen er tom

Oppgave 8

Metoden `boolean hasNext()` og konstruktøren `public DobbeltLenketListeIterator()` i klassen `DobbeltLenketListeIterator` er ferdigkodet og skal **ikke** endres.

a) Lag metoden `T next()`. Den skal først sjekke om *iteratorendringer* er lik *endringer*. Hvis ikke, kastes en `ConcurrentModificationException`. Så en `NoSuchElementException` hvis det ikke er flere igjen i listen (dvs. hvis *hasNext*() ikke er sann/true). Deretter settes `fjernOK` til sann/true, verdien til *denne* returneres og *denne* flyttes til den neste node.

b) Lag metoden `Iterator<T> iterator()`. Den skal returnere en instans av iteratorklassen.

c) Lag konstruktøren `private DobbeltLenketListeIterator(int indeks)`. Den skal sette pekeren *denne* til den noden som hører til den oppgitte indeksen. Resten skal være som i den konstruktøren som er ferdigkodet.

d) Lag til slutt metoden `Iterator<T> iterator(int indeks)`. Det må først sjekkes at indeksen er lovlig. Bruk metoden *`indeksKontroll`*`()`. Deretter skal den ved hjelp av konstruktøren i punkt c) returnere en instans av iteratorklassen.

Nå vil `default`-metoden `void forEach(Consumer<? super T> handling)` i grensesnittet `Iterable` virke. En vanlig forAlle-løkke bruker implisitt en iterator. Sjekk at følgende kode virker for deg:

```
String[] navn = {"Lars","Anders","Bodil","Kari","Per","Berit"};
Liste<String> liste = new DobbeltLenketListe<>(navn);

liste.forEach(s -> System.out.print(s + " "));
System.out.println();
for (String s : liste) System.out.print(s + " ");

// Utskrift:
// Lars Anders Bodil Kari Per Berit
// Lars Anders Bodil Kari Per Berit
```

Oppgave 9

a) Lag metoden `void remove()` i iteratorklassen. Her kan du ikke bruke noen av de ordinære fjern-metodene. For det første vet vi ikke hvilken indeks det er snakk om her og for det andre kan verdien som skal fjernes, forkomme flere steder og dermed kan gal verdi bli fjernet. Her må `remove()` kodes direkte.

Hvis det ikke er tillatt å kalle denne metoden, skal det kastes en IllegalStateException. Hvis `endringer` og `iteratorendringer` er forskjellige, kastes en Concurrent-ModificationException. Hvis disse hindrene passeres, settes `fjernOK` til usann/false. Så skal noden rett til venstre for *p* fjernes. Den finner vi lett siden det går en peker dit. Her må en passe på alle tilfellene.

1. Hvis den som skal fjernes er eneste verdi (`antall == 1`), så nulles hode og hale.
2. Hvis den siste skal fjernes (`denne == null`), så må `hale` oppdateres.
3. Hvis den første skal fjernes (`denne.forrige == hode`), så må `hode` oppdateres.
4. Hvis en node inne i listen skal fjernes (noden `denne.forrige`), så må pekerne i nodene på hver side oppdateres.

Til slutt reduseres `antall` og både `endringer` og `iteratorendringer` økes.

Metoden `boolean fjernHvis(Predicate<? super T> sjekk)` er en `default`-metode i grensesnittet `Beholder`. Den vil virke så sant metoden `remove()` i iteratorklassen er kodet på en korrekt måte. Her er et eksempel på hvordan den kan brukes. Sjekk at det virker. Lag så for din egen del et predikat som gjør at de navnene som inneholder e fjernes. Og så et der kun de navnene som inneholder 5 bokstaver blir igjen.

```
DobbeltLenketListe<String> liste =
  new DobbeltLenketListe<>(new String[]
    {"Birger","Lars","Anders","Bodil","Kari","Per","Berit"});
```

```
liste.fjernHvis(navn -> navn.charAt(0) == 'B'); // fjerner navn som starter med B

System.out.println(liste + " " + liste.omvendtString());

// Utskrift: [Lars, Anders, Kari, Per] [Per, Kari, Anders, Lars]
```

Oppgave 10

Lag `public static <T> void sorter(Liste<T> liste, Comparator<? super T> c)`. Den skal sortere *liste* ved hjelp av komparatoren *c*. Siden parametertypen for *liste* er `Liste<T>`, vil det kun være de metodene som står grensesnittet `Liste` som kan brukes. Her vil det selvfølgelig være fristende å kopiere alle verdiene over i en tabell, sortere tabellen og så kopiere dem tilbake i listen. Men oppgaven **skal** løses uten bruk av hjelpestrukturer. Dvs. det skal være en såkalt «på plass»-sortering (eng: in place).

Målet er å få metoden til å virke (at den sorterer korrekt). Det er det eneste som testprogrammet undersøker. Men selv om det er umulig å lage en effektiv «på plass»-sortering i en lenket liste, er det jo likevel et poeng å ikke lage koden unødvendig ineffektiv. Det vil være mulig å finne ut hvilken orden metoden din har ved en analyse. Men du kan få en indikasjon ved hjelp av tidsmålinger. Lag en liste med så mange tilfeldige verdier at metoden f.eks. bruker ca. 1 sekund. Lag så en som har dobbelt så mange verdier. Hvis tidsforbruket da blir ca. 4 sekunder, kan metoden din være av kvadratisk orden. Bruken den ca. 8 sekunder, er den kanskje av kubisk orden.

Argumentet *liste* i metoden *sorter*() vil være en instans av en klasse som implementerer grensesnittet `Liste`. Det betyr at også instanser av TabellListe og EnkeltLenketListe vil bli sortert av denne metoden. Hvis du har en eller begge av disse klassene tilgjengelig, vil følgende kodebit vise hvordan metoden skal virke:

```
String[] navn = {"Lars","Anders","Bodil","Kari","Per","Berit"};

Liste<String> liste1 = new DobbeltLenketListe<>(navn);
Liste<String> liste2 = new TabellListe<>(navn);
Liste<String> liste3 = new EnkeltLenketListe<>(navn);

DobbeltLenketListe.sorter(liste1, Comparator.naturalOrder());
DobbeltLenketListe.sorter(liste2, Comparator.naturalOrder());
DobbeltLenketListe.sorter(liste3, Comparator.naturalOrder());

System.out.println(liste1);  // [Anders, Berit, Bodil, Kari, Lars, Per]
System.out.println(liste2);  // [Anders, Berit, Bodil, Kari, Lars, Per]
System.out.println(liste3);  // [Anders, Berit, Bodil, Kari, Lars, Per]

// Tabellen navn er upåvirket:
System.out.println(Arrays.toString(navn));
// [Lars, Anders, Bodil, Kari, Per, Berit]
```