DAT102 – Obligatorisk nr 3

Består av deler fra Øving 6, 7 og Øving 8 (denne)

Innleveringsfrist: 6. april 2022

Vi henstiller at dere er 3-4 på gruppen, men aldri flere. Hvis dere lager grupper på 4, kan 2 og 2 jobbe tett sammen i smågrupper med å programmere samtidig og diskutere underveis. Pass på å veksle mellom hvem som skriver. De to smågruppene blir enige om en felles innlevering slik at dere leverer inn som en gruppe på 4.

Dere eksporterer javaprosjektene, zipper og leverer inn på Canvas. Prosjektene skal kunne kjøres. Teorioppgaver leverer dere inn på canvas som pdf-fil.

OBS! Husk å ta med navn på alle gruppemedlemmene. Canvas tillater ikke å gjør det i ettertid.

Oppgave 1 (fra øving 6)

Når man skal implementere en algoritme, kan dette gjøres på flere måter. Ofte vil det være mulig å gjøre «triks» som sparer tid. Algoritmen vil ha samme orden, men konstanten, c framfor leddet som vokser raskest blir mindre. Dere skal undersøke om slike triks har betydning i sorterting ved innsetting (insertion sort).

Starten på den indre løkken kan se slik ut

```
while (!ferdig && j >= 0)
```

der første del av betingelsen er at vi ikke har funnet rett plass for elementet som skal settes inn og andre del at det er flere elementer å sammenligne med. Dersom vi flytter det minste elementet fremst i i tabellen før vi starter selve sorteringen, kan betingelsen forenkles siden vi aldri skal sette inn et element i posisjon 0 i tabellen. En av fordelene til sortering ved innsetting er at den er stabil (stable). Det vil si at like elementer vil ha samme innbyrdes rekkefølge etter sortering. For å beholde denne egenskapen kan vi gå fra høyre mot venstre i tabellen og bytte om naboelementer om de står feil i forhold til hverandre. Da er det minste elementet kommet først.

a) Modifiser koden som angitt ovenfor og se om det har betydning for tidsbruken. La antall elementer være så stort at det tar minst 5 sekunder å utføre sorteringen. Skriv kort hva dere observerer. For å generere tilfeldige data og måle tid, se etter oppgave 2.

b) Modifiser koden slik at i stedet for å sette inn ett element om gangen, setter vi inn to. Så lenge det største elementet er mindre enn elementet vi sammenligner med i sortert del, så kan vi flytte elementet to plasser til høyre. Når vi finner rett plass for det største, forsetter vi som vanlig med å sette inn det minste. Kombiner med å flytte det minste elementet først (som i a) før sorteringen starter. Pass på at metoden fungerer for både odde og jevne n. Skriv kort hva dere observerer.

I tillegg til observasjonene, skal dere levere kode for både a) og b).

Oppgave 2 (fra øving 6)

I denne oppgaven skal dere få praktisk erfaring med hvor lang tid sorteringsmetodene trenger for å sortere tabeller med heltall. Det blir forskjell (men bare i konstanten framfor leddet som vokser raskest) for en og samme sorteringsmetode om vi for velger primitiv type heltall (int) eller om vi velger heltalsobjekt (Integer). Implementer metodene nedenfor i Java.

- Sortering ved innsetting (Insertion sort)
- Utvalgssortering (Selection sort)
  Samme som Plukksortering som vi så i DAT100
- Kvikksortering (Quick sort)
- Flettesortering (Merge sort)

Ta gjerne utgangspunkt i en tabell av heltalsobjekt (*Integer*). Før dere går videre, kontroller at sorteringsmetodene er korrekte ved å bruke de på en liten tabell (n = 10) og skriv ut tabellen etter sortering.

La T(n) være tiden det tar å sortere n element. At en algoritme bruker tid O(f(n)) betyr at T(n) = c*f(n). I sorteringsmetodene ovenfor er f(n) lik n^2 eller $n*\log_2 n$. Bestem først c slik at T(n) = c*f(n). Dette kan gjøres ved å måle tiden for en spesiell n-verdi, for eksempel n = 32000, og så løse ligningene med c som ukjent. Konstanten c er avhengig av

- algoritmen
- implementasjonen
- maskinen som kjører programmet

Til slutt i oppgaven finner du forklaring på hvordan du kan måle tiden og hvordan du kan få en tabell med tilfeldige tal.

a) For hver sorteringsmetode kan utskriften se ut som under (men med overskrift og f(n) erstattet med det som er relevant). Diskuter hvordan de teoretiske resultatene samsvarer med de målte og prøv å forklare eventuelle avvik.

Resultat Kvikksortering (tilsvarande tabeller for de andre metodene)**:**

n	**Antall målinger**	**Målt tid (gjennomsnitt)**	**Teoretisk tid c*f(n)**
32000			
64000			
128000			

På første linje i tabellen vil målt og teoretisk tid være like dersom dere bruker 32000 for å bestemme c.

b) Prøv å sortere en tabell der alle elementene er like med Quicksort og mål tiden. Forklar hva som skjer og hvorfor det skjer?

Hvordan måle utføringstider i Java

For å måle brukt tid kan du bruke klassene **`Instant`** og **`Duration`**. Generelt om pakken java.time:

https://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html.

Det er også mulig å bruke: **`System.nanoTime().`** Se

https://www.baeldung.com/java-measure-elapsed-time

Av ulike grunner blir ikke tidtakingen helt nøyaktig, spesielt for små tider. Derfor kan det for små n-verdier være aktuelt å utføre den kritiske delen flere ganger og så bruke gjennomsnittstiden. Dette gjør dere ved å legge den kritiske koden innenfor en løkke, utføre løkken et visst antall ganger og så finne gjennomsnittlig tid. Eksempel på kode:

```
Random tilfeldig = new Random(…);
int n = 32000;
int antal = 10;

Integer[][] a = new Integer[antal][n];

// set inn tilfeldige heiltal i alle rekker
for (int i = 0; i < antal; i++){
      for (int j = 0; j < n; j++){
            a[i][j] = tilfeldig.nextInt();
      }
}

// start tidsmåling
for (int i = 0; i < antal; i++){
      sorter(a[i]); // blir ein eindimensjonal tabell
}
// slutt tidsmåling
```

Hvordan få en tabell med n tilfeldige heltall

Java har en forhåndsdefinert klasse for tilfeldige tal som heter Random. Den har to konstruktører

- `Random()` - bruker systemklokken for å gi generatoren en startverdi.
- `Random(long starverdi)` - vi gir startverdi. Det vil si at vi kan generere nøyaktig same tallrekke på nytt flere ganger. Dette er aktuelt ved testing.

Etter å ha initiert generatoren, kan du bruke flere metoder, men den vi trenger i øvelsen er:

- `int nextInt()` - som gir et tilfeldig heltall. Skisse for å lage in tabell med tilfeldige heltall er vist under:

```
import java.util.Random;
...
Random tilfeldig = new Random(); // brukar maskinen sin klokke for å gi startverdi
...
for (int i=0; i < n; i++){
   tabell[i] = tilfeldig.nextInt();
}
```

Dersom dere vil ha heltall fra og med 0 til (men ikke med) M, kan dere skrive

```
tabell[i] =  tilfeldig.nextInt(M);
```

Oppgave 3 (fra øving 7))

a) Gi en definisjon av binært tre.
Gi en definisjon av høyden til et binært tre.
Definer fullt binært tre og komplett binært tre.

b) Tabeller kan brukes for å implementere trær. Tegn de to binære trærne som også kan illustreres ved hjelp av følgende tabeller:

i)

0	1	2	3	4	5	6	7	8	9
i	E	n	s	t	O	r	b	å	t

ii)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
i	e	N		s		t			O	r			b	å					t

c) Vis rekkefølge ved gjennomgang i i) pre-orden, ii) in-orden, iii) post-orden & iv) nivå-orden for de to trærne dere har tegnet under oppgave 1b) over.

Oppgave 4

Ta utgangspunkt i koden knyttet til F18 – BS-tre.

a) Boken definerer høyden av et binært til å være lengden på den lengste stien fra roten til et blad. Vi kan dermed definere at et tomt tre har høyde -1. Lag en rekursiv metode i BSTre for å finne høyden av et binært tre.

   Generer noen tre ved å bruke ved å bruke leggTil-metoden for å se at høymetoden gir rett svar.

b) Lag et klientprogram som genererer for eksempel 100 tilfeldige binære søketre med 1023 noder. (Lag først klientprogram med et lite tre.) For å generere et tilfeldig binært søketre må nøklene komme i tilfeldig ordning. Dette kan de få til ved å bruke Random-klassen og bruke int/Integer som nøkler. Skisse nedenfor:

```
import java.util.*;

…

Random terning = new Random();

…

int tall = terning.nextInt();

…
```

Programmet skal skrive ut

i) antall noder n
ii) den minimale teoretiske høyden (her vil det være naturlig å lage en metode som regner ut denne verdien for en gitt verdi av n).
iii) den maksimale teoretiske høyden
iv) minste høyde i løpet av kjøringene
v) største høyde i løpet av kjøringene
vi) gjennomsnittlig høyde av alle kjøringene

Lag en metode som beregner høyden: **`public int hoyde()`**{…} og bruk den i kjøringene.
Denne metoden kaller en rekursiv metode: **`private int hoydeRek(BinaerTreNode<T> t){…}.`**

c) Det kan vises at den gjennomsnittlige høyden av et binært søketre når vi setter inn n tilfeldige nøkler er $O(\log_2 n)$, dvs at høyda er tilnærmet lik $c \log_2 n$ der c er en konstant. For å bestemme konstanten kan du måle den gjennomsnittlige høyden for en bestemt verdi av n, for eksempel n = 1023. Dermed får du en ligning med c som ukjent. Bruk formelen for å finne c og deretter gi et overslag for den gjennomsnittlige høyden når n = 8191. Kjør programmet med n = 8191 og vurder resultatet. Stemmer det med den utregnete verdien for høyden?

d) Frivillig

Metoden under skriver ut verdier i dette BS-treet mellom to grenseverdier. Vi får sortert ordning når vi går gjennom BS-treet i inorden.

```
public void skrivVerdier(T nedre, T ovre){

        skrivVerdierRek(rot, nedre, ovre);

}//


private void skrivVerdierRek(BinærTreNode<T> t, T min, T maks){

  if(t != null){
    skrivVerdierRek(t.getVenstre(), min, maks);
    if((t.hentElement().compareTo(min) >= 0)
           &&(t.hentElement().compareTo(maks) <=0)) {
      System.out.print(t.getElement() + " ");
    }
    skrivVerdierRek(t.getHoyre(), min, maks);
  }
}
```

i. Forklar hvorfor metdoen `skrivVerdierRek` ikke er optimal. Ta metoden med i klassen `KjedetBSTre` og prøv den ut.

ii. Modifiser metoden litt slik at den blir mer optimal. Ta metoden med i klassen `KjedetBSTre` og prøv den ut.

Oppgave 5

a) En spesiell type binære tre er kalt en haug ("heap"). Definer hva som menes med en haug. (Tips få fram de to vesentlige egenskapene definisjonen.)

b) I minst en av de tre tabellene under er elementene ordnet slik at de danner en **makshaug** (node >= venstrebarn og høgrebarn). Vi bruker ikke plass 0 i tabellen, så roten ligger eventuelt i posisjon 1 i tabellen. Finn hvilke(n). Begrunn svaret

a[0]										a[10]
	9	15	12	7	4	2	1	6	5	3

b[0]										b[10]
	15	12	10	11	2	6	3	4	8	1

c[0]										c[10]
	15	10	14	8	7	13	6	2	5	4

c) Elementene i tabellen d under utgjør en makshaug.

d[0]										d[10]
	15	10	14	8	7	13	6	2	5	4

i) Tegn haugen som tre og vis hvordan treet ser ut når vi først fjerner elementet med verdi 14 og deretter organiserer til en haug (makshaug) (tips: etter fjerning av 14, flyttes først siste element (4) til rotposisjon, osv..

ii) Etter at vi har utført pkt i) setter vi nå et nytt element med verdi 13 inn sist i D og organiserer igjen til en haug. Vis hvordan treet nå ser ut.

d) Forklar kort hvordan prinsippet i pkt c) kan brukes til å sortere elementer

e) Det skal lages en minimumshaug ved å bruke reparerNed (reheap). Du starter på siste interne node (node som ikke er blad og så går du mot venstre helt til roten). Ved start er elementene plassert i en tabell som på figuren under.

d[0]										d[10]
	10	9	8	7	6	5	4	3	2	1

Tegn først opp treet med de elementene som her er gitt. Du skal nå lage en minimumshaug. Vis alle overgangene ved å tegne opp treet på nytt etter hvert kall av reparerNed inntil du har en minimumshaug.

f) På github ligger et uferdig prosjekt for TabellHaugU som dere kan bygge videre på. Merk at dette er en minimumshaug der roten er lagret i posisjon 0. Da vil venstrebarn for en node j være i posisjon 2 * j + 1 og høyrebarn i posisjon 2 * j + 2.

Legg merke til at i denne implementasjonen er det ikke brukt exception (men det kan dere endre hvis dere vil) som vi har brukt i flere typer samlinger tidligere. Lag metoden reparerOpp i klassen TabellHaugU.java. Legg ved koden for denne metoden + utskrift av kjøring av klientprogrammet.