Eksamen i

Algoritmer og Datastrukturer

IAI 21899

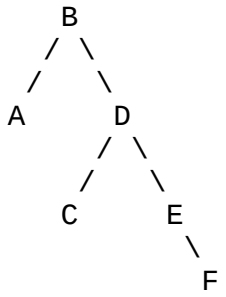
Høgskolen i Østfold
Avdeling for informatikk
og automatisering

Torsdag 30. november 2000, kl. 09.00 - 14.00

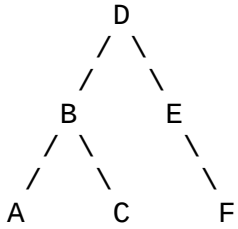
LØSNINGSFORSLAG

Del 1, Binære søketrær – Totalt 20%

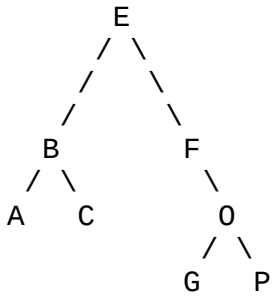
Oppgave 1.1 - 5%



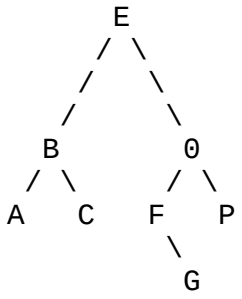
Oppgave 1.2 - 5%



Oppgave 1.3 - 5%



Oppgave 1.4 - 5%



Slutt på del 1

Del 2, Sortering – Totalt 20%

Oppgave 2.1 - 10%

Spennet mellom minste og største verdi er 80 grader, og det er hundre verdier i hver grad (2 desimalers nøyaktighet), dvs. 8001 mulige verdier. Dette tyder på at hashing/bøttesortering (alt etter hvordan man ser det, avrundingen er jo egentlig en hashfunksjon, som i tillegg gir en sortering) med en tabell av størrelse 8001. I hver "bøtte" må det så sorteres på løpenummer. I og med at originaldataene er nettopp sortert på løpenummer, kan vi bare legge en ny måling sist i lista i den aktuelle bøtta. En lenket liste i hver bøtte gir altså en optimal løsning.

Innsetting i den sorterte strukturen er altså en konstant operasjon, vi finner riktig "bøtte" i konstant tid, og legger til i slutten av en lenket liste i konstant tid. Totalt altså *O(N)* for sorteringen, fordi N her er betydelig større enn antall bøtter (bøttesortering er jo formelt sett O(M+N), der M er antall bøtter).

```
class TempSort {

  // Metoden tar i mot en usortert tabell med double presisjon og
  // returnerer en float tabell med sorterte temperaturer slik
  // det er beskrevet i oppgaveteksten.

  float[] sorterAvrundet(double[] tab) {
    // Oppretter bøtte/hashtabell hvor hvert element
    // er en liste av temperaturer
    // OBS: Her kunne vi bare hatt en teller som holdt
    // rede på hvor mange vi hadde av denne verdien
    // Den valgte metode åpner for lagre andre data, f.eks. løpenr.

    TempListe[] sortert_hash = new TempListe[SIZE];
    TempListe liste  = null;
    int indeks         = 0;

    // Går gjennom hele input tabellen
    for (int i = 0; i < tab.length; i++) {
      // Finner riktig indeks i bøttetabellen
      indeks = finnIndeks(tab[i]);
      liste = sortert_hash[indeks];

      // Sjekker om denne temperaturen allerede er satt inn
      if (liste == null) {
        sortert_hash[indeks] = new TempListe(tab[i]);
      }
      else {
        sortert_hash[indeks].leggTil(tab[i]);
      }
    }

    // oppretter output tabellen
    float[] ferdig_sortert = new float[tab.length];
    int j = 0;
    TempNode node = null;

    // Bygger opp output fra bøttetabellen
    for (int i = 0; i < MAX; i++) {
      if (sortert_hash[i] != null) {

        // Hvis vi kun tok vare på antallet av disse
        // verdiene ville denne delen typisk bli erstattet
        // av en for-løkke og en omregning av indeksen
        // tilbake til korresponderende temperatur
        node = sortert_hash[i].første;
        while (node != null) {

          // Egentlig overkill, men åpner for håndtering
          // av tillegsdata, f.eks. løpenr.
          ferdig_sortert[j] = node.temp;
          j++;
          node = node.neste;
        }
      }
    }

    // Returnerer ferdig oppbygget og sortert tabell
    return ferdig_sortert;
  }
```

```
  int finnIndeks (double t) {
    // Må gjøre alle temperaturer positive, og de
    // havner dermed i intervallet [0.0, 80.0]
    double temp = t + 40;

    // Ganger opp temperaturene slik at de havner
    // i intervallet [0.0, 8000.0]
    temp        = temp * 100.0;

    // Runder av slik av temperaturen blir
    // et heltall mellom 0 og 8000
    // Tilsvarer og runde av til 2 desimaler
    // fordi vi har ganget med 100
    return Math.round(temp);
  }

  // Vi har 80001 mulige verdier av temperaturene etter avrunding
  final static int SIZE = 8001;

}

class TempListe {
  // Klasse som representerer en liste av TempNoder med
  // referanser til første og siste node
  TempListe (double t) {
    første = new TempNode(t);
    siste  = første;
  }

  void leggTil(double t) {
    TempNode ny = new TempNode(t);

    if (første = null) {
      første = ny;

    }
    else {
      første.neste = ny;
    }
    siste = ny;

  }

  TempNode  første;
  TempNode  siste;
}

class TempNode {
  // Kunne inneholdt mere informasjon, f.eks. løpenummer
  // hvis det ville være interessant
  TempNode (double t) {
    temp = (float)t;
    next = null;
  }

  float    temp;
  TempNode  neste;
}
```

Oppgave 2.2 - 10%

I og med at vi her har å gjøre med tall med dobbel presisjon uten avrunding må vi se bort fra hashing/bøttesorteringsprinsipper.

Å finne riktig innsettingspunkt i arrray med binærsøk er O(logN). Reorgansiering ved innsetting er dessverre O(N). Innsetting i lenket liste er O(1), men her koster det O(N) å finne innsettingspunktet. Hvis vi derimot bruker et balansert søketre, f.eks. AVL, vil de totale kostnadene ved en innsetting være O(logN). Altså er et balansert tre et åpenbart godt valg.

En heap kan også brukes, hvor innsetting er en logaritmisk operasjon, men dette er strengt tatt ingen sortert struktur. Hvis vi skal skrive ut den ut på sortert form, koster dette *O(NlogN)*. Ved.f.eks. AVL tre er utskrift av den til enhver tid sorterte strukturen en lineær operasjon.

Alternativt kan vi bruke en bøttesortering tilsvarende forrige oppgave, men denne gang med et sortert, balansert tre i hver "bøtte" (jfr. Oppgave 1.5, eksamen høst 99).

I vedlegget ser vi at temperaturmålingene intuitivt nok følger en kurve, og er rimelig saktevarierende i forhold til den aktuelle målehyppighete. Vi kan da tenke oss en innsettingsmetode som husker posisjonen der siste måling ble satt inn. Ved neste måling går vi til denne posisjonen, og søker oss derifra for å finne riktig innsettingspunkt. Dette skulle være en rimelig god avskjæring, og redusere konstantleddet en god del.

Denne strategien vil ikke forbedre metodens orden, men vil pga avskjæringene sannsynligvis være raskere i praksis.

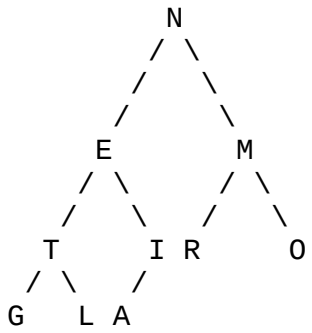
Slutt på del 2

Del 3, Binærheap – Totalt 30%

Oppgave 3.1 – 5%

Forelder: floor av (k-1)/2
Venstre: 2k+1
Høyre: (2k+1)+1 = 2k+2 =2(k+1)

Oppgave 3.2 – 10%



Oppgave 3.3 – 15%

```
class BinHeap {

  // Denne skal implementeres i oppgave 3.2
  // ved hjelp av den rekursive lagGrein
  public void lagTre(char[] tre) {
    // Starter rekursiv oppbygging ve å sette første
    // element i tabellen til å bli rota
    rot = lagGrein(tre, 0);
  }

  // Rekursiv metode som bygger opp et subtre fra en tabell
  // Skal implementeres i oppgave 3.2

  public Node lagGrein(char[] tre, int i) {

    // Hvis vi prøver å finne noe utenfor tabellen
    // returnerer vi en null-node
    if (i > tre.length-1) {
      return null;
    }

    // Bygger venstre og høyre subtre rekursivt
    vBarn = lagGrein(tre,(2*i)+1);
    hBarn = lagGrein(tre,2*(i+1);

    // Lager og returnerer ny node med verdien
    // i det aktuelle tabellelementet
    // og med venstre og høyre subtre
    return new Node(tre[i], vBarn, hBarn);
  }

  Node rot;
}
```

Slutt på del 3

Del 4, Algoritmeanalyse – Totalt 20%

Oppgave 4.1 – 10%

Det er naturlig å anta at vi her har enten *NlogN* eller kvadratiske algoritmer å gjøre med.

rimiSort: Vi ser av plotting av rådataene at dette muligens er en kvadratisk algoritme. Kurven er ikke ulik standard innsettingssortering, så dette støtter hypotesen. Når vi deler ut med *NlogN*, ser vi at kurven fremdeles er stigende, derfor må ordene være høyere enn dette. Ved deling med N^2 ser vi at vi får en horisontal linje, dvs. en konstant funksjon. *T(N)* må da være omtrentlig lik cN^2, og vi kan faktisk lese av konstanten: $c = 0.00022$. rimiSort er altså $O(N^2)$. Vi ser også at innsettingssortering også gir en horisontal kurve i samme plott, og dette styrker også vårt resultat, siden vi vet at denne metoden er kvadratisk på vilkårlig sammensatt input.

lrSort: Vi ser av plotting av rådataene at dette muligens er en subkvadratisk algoritme. Når vi deler ut med *NlogN*, ser vi at vi får en horisontal linje, dvs. en konstant funksjon. *T(N)* må da være omtrentlig lik *cNlogN*, og vi kan faktisk lese av konstanten: $c = 0.0005$. Ved deling med N^2 ser vi at vi får en synkende kurve, og dette gjør oss sikker i vår sak.: *NlogN*.

Oppgave 4.2 – 10%

lrSort: Flere hint tyder (lrSort...left/rightSort, variabel venstre, flytt, bytt....) på at dette er en slags tresortering. Av dem har vi bare gjennomgått en i pensum, heapsort. Det burde ikke være vrient å finne ut at dette er heapsort hvor treet er representert i tabellform. Jfr. også oppgave 3. Heapsortering er *NlogN*.

rimiSort: Denne var kanskje verre. En nesten identisk metode ble gjennomgått summarisk i en forelesning. Dette er rett og slett en rekursiv innsettingssortering. Vi nøster oss først rekursivt til nest siste element, så startes en rekursiv innsetting ved ”bobling” til vi finner riktig plass. Rekursjonen fører oss deretter til nest nest siste element, vi foretar innsetting av dette elementet i den allerede sorterte siste del av tabellen. Slik bygger vi altså opp halen på tabellen, som da hele tiden vil være sortert. I og med at metoden fungerer etter innsettingsprinsippet, må den være kvadratisk, $O(N^2)$.

Empirisk test og teoretisk analyse stemmer overens.

Slutt på del 4

Del 5, Nettverk – Totalt 10%

Oppgave 5.1 - 10%

Trinn #	Noder med oppdatert distanse fra A, samt forrige referanse							Current Node	Naboer	Ferdig behandlet
	A	B	C	D	E	F	G			
Init	A, 0	*	*	*	*	*	*			
1		A, 2	A, 4					A	B C	A
2			B, 3	B, 9				B	C D	A B
3					C, 18	C, 12		C	E F	A B C
4							D, 12	D	G	A B C D
5					F, 15			F	E	A B C D F
6								G	---	A B C D F G
7								E	---	A B C D F G E

Korteste vei fra A til G (12): A B D G

Slutt på del 5

Slutt på oppgavesettet