Tentamen
Datastrukturer D
DAT 036/DIT960

17 december 2010

- Tid: 8.30 - 12.30
- Ansvarig: Peter Dybjer, tel 0736-341480 eller ankn 1035
- Max poäng på tentamen: 60.
- Betygsgränser, CTH: 3 = 24 p, 4 = 36 p, 5 = 48 p, GU: G = 24 p, VG = 48 p.
- Hjälpmedel: *handskrivna* anteckningar på *ett* A4-blad. Man får skriva på båda sidorna och texten måste kunna läsas utan förstoringsglas. Anteckningar som inte uppfyller detta krav kommer att beslagtas!
- Skriv tydligt och disponera papperet på ett lämpligt sätt.
- Börja varje ny uppgift på nytt blad.
- Skriv endast på en sida av papperet.
- **Kom ihåg:** alla svar ska motiveras väl!
- Poängavdrag kan ges för onödigt långa, komplicerade eller ostrukturerade lösningar.
- Lycka till!

1. Avgör för vart och ett följande påståenden om det är sant eller falskt! Svaren måste motiveras med en kort förklaring för att ge poäng.

   (a) Binärsökning är en effektiv metod för att söka upp ett givet element i en sorterad länkad lista. (2p)

   Svar: Nej. För att hitta mittersta elementet i en länkad lista måste man göra linjärsökning och då går hela poängen med binärsökning förlorad.

   (b) Man kan implementera en kö effektivt genom att använda en enkellänkad lista. (2p)

   Svar: Ja. Man kan implementera alla kö-operationerna i konstant tid om man har pekare både till den första och den sista noden i den länkade listan.

   (c) Den amorterade värstafallskomplexiteten per metod är lika bra eller bättre än värstafallskomplexiteten för metoden. (2p)

   Svar: Ja, den amorterade värstafallskomplexiten beräknas genom att ta medeltalet av en sekvens av operationer som utförs efter varandra. Därför kan den inte vara sämre än den vanliga värstafallskomplexiteten.

   (d) Det är viktigare att ha en bra hashfunktion när man använder hinkar ("buckets") än när man använder öppen adressering ("open addressing") med linear probing. (2p)

   Svar: Nej, öppen adressering med linear probing är mer känslig för kollisioner eftersom den lätt leder till att elementen klumpar ihop sig. Om man har dålig haschfunktion får man på så sätt fler "sekundära" kollisioner.

   (e) Antag att du har två hashtabeller. Båda hashtabellerna använder sig av hinkar ("hashing in buckets" eller "chaining"). Den ena använder ett fält med M celler och innehåller m element. Den andra använder ett fält med N celler och innehåller n element. Då finns det en $O(n+N)$-algoritm som bygger en hashtabell som lagrar unionen av elementen i de två ursprungliga hashtabellerna. (3p)

   Svar: Ja, man går genom alla n elementen och alla N celler i den andra hashtabellen och sätter in dem i den första.

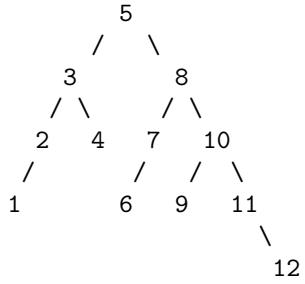
   (f) Man kan skriva en iterator för grafer som (i) besöker noderna i grafen i djupet först ordning och (ii) vars asymptotiska komplexitet för `next`-metoden i en graf med m bågar och n noder är $O(1)$ i värsta fall. (3p)

   Svar: Snabbaste sättet att traversera en graf är med BFS eller DFS som båda tar $O(m+n)$. Om initieringen av iteratorn är $O(1)$ kan inte `next`-metoden exekveras med komplexitet $O(1)$, för i så fall skulle man kunna traversera grafen på tiden $O(n)$. Man kan dock implementera en iterator med `next`-metod som exekveras med komplexitet $O(1)$, om man utför hela traverseringen vid initieringen av iteratorn och t ex placerar elementen i en länkad lista.

2. (a) Sätt in följande element ett efter ett i ett AVL-träd och rita det resulterande trädet! (Du behöver inte rita mellanliggande steg.)

   `5, 3, 8, 2, 4, 7, 10, 1, 6, 9, 11, 12`

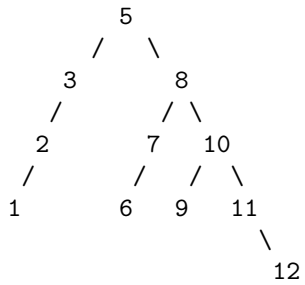
   Du ska använda standardalgoritmen för insättning. (2p)
   Svar.



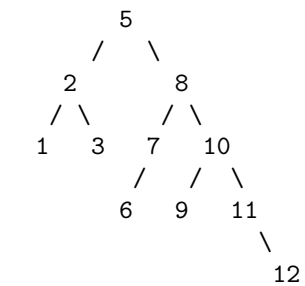
   Notera att man inte behöver göra några rotationer.

   (b) Räkna upp elementen i postordning, dvs gör "postorder traversal" av noderna i AVL-trädet i (a)! (2p)
   Svar: 1, 2, 4, 3, 6, 7, 9, 12, 11, 10, 8, 5

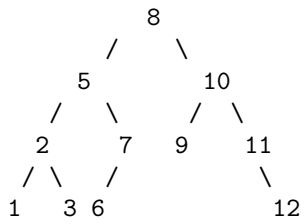
   (c) Ta nu bort elementet 4 från trädet i (a) med standardalgoritmen för borttagning i AVL-träd! Rita det resulterande trädet och förklara steg för steg hur borttagningsalgoritmen fungerar i detta fall! (3p)
   Svar. Efter att vi tagit bort elementet 4 får vi följande träd:



   Vi har nu en lokal obalans i elementet 3. Vi gör en enkelrotation av elementen 1, 2, 3 för att återupprätta balansen:



Genom denna rotation får vi dock en obalans i roten med elementet 5, eftersom höjden på dess vänstra delträd minskat. Vi gör därför ytterligare en enkelrotation kring elementen 5, 8, 10. Detta resulterar i följande balanserade träd.



(d) Skriv en Javaklass `AVL` för AVL-träd! Du behöver bara implementera tillståndsvariablerna i klassen och inte några metoder – inte ens någon konstruerarmetod.

Kom ihåg att ett AVL-träd är ett balanserat binärt sökträd. Implementeringen ska vara sådan att man direkt (på $O(1)$ tid) kan avgöra om obalans har uppstått. För enkelhets skull kan du förutsätta att AVL-träden innehåller heltal. (3p)

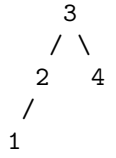
Svar:

```
class AVL {
   AVLNode root;
}

class AVLNode {
   int elem;
   AVLNode left;
   AVLNode right
   int height;
}
```

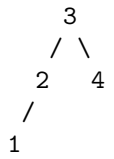
Notera en AVL-nod är som en vanlig nod i ett länkat binärt träd, som dessutom innehåller en tillståndsvariabel `height` som beskriver hur högt det delträd är som har noden som rot. På det sättet blir det lätt att avgöra (på $O(1)$ tid) om en lokal obalans har uppstått i noden: jämför höjden på delträden och kontrollera att den maximalt skiljer sig med en enhet.

(e) Rita en bild som visar hur du representerar (i) det tomma AVL-trädet (ii) AVL-trädet

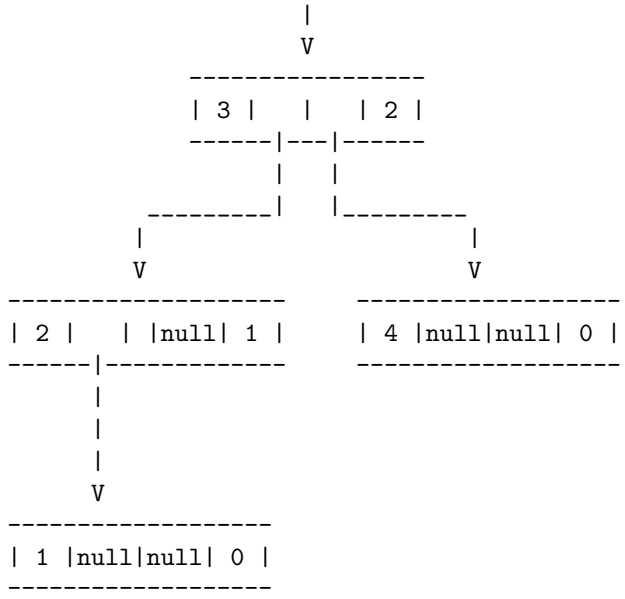


Du ska rita ut alla minnesceller som din implementering i deluppgift (d) använder. (2p)

Svar: (i) Det tomma AVL-trädet representeras av en nollpekare.
(ii) AVL-trädet



representeras av följande pekarstruktur:



`null` står för nollpekare. Notera också att vi låter ett löv ha höjden 0.

3. (a) Skriv en algoritm (i pseudokod) som beräknar diametern hos en given sammanhängande oriktad oviktad graf! Diametern hos grafen definieras som det största avståndet mellan två noder i grafen, och avståndet definieras som antalet bågar på den kortaste vägen mellan noderna.

   För full poäng på uppgiften krävs att du har en algoritm med optimal asymptotisk komplexitet. Mindre effektiva algoritmer ger lägre poäng. Om du använder standardalgoritmer från kursen behöver du inte skriva pseudokod för dem.

   Svar: Man gör bredden-först sökning för varje nod v i grafen. På så sätt bestämmer man största avståndet från v till en annan nod. Diametern beräknas genom att ta maximum av alla dessa största avstånd. (För fullständig lösning ska du skriva pseudokod för denna algoritm.) (5p)

   (b) Vilken asymptotisk tidskomplexitet har din algoritm? Du ska beskriva komplexiteten som en funktion av antalet noder och bågar i grafen och bestämma O-komplexiteten. För full poäng ska du ange ett så enkelt O-uttryck som möjligt. (3p)

   Svar: Bredden först sökning från en nod är $O(m + n)$ om grafen har m bågar och n noder. Eftersom grafen är sammanhängande är $m \geq n - 1$ och alltså kan vi förenkla $O(m + n) = O(m)$. Eftersom vi gör n bredden först sökningar är algoritmens komplexitet $O(mn)$. (Om grafen är tät kan en variant av Floyd-Warshalls algoritm vara ett intressant alternativ till upprepade bredden först sökning. Då får vi en $O(n^3)$-algoritm, vilket kan vara bättre än $O(mn)$ för en tät graf eftersom Floyd-Warshall har liten konstant.)

4. `ArrayList` är en användbar klass i Java Collections, som implementerar ett dynamiskt fält. I beskrivningen av klassen står det "Resizable-array implementation of the List interface." och "The `add` operation runs in amortized constant time, that is, adding n elements requires $O(n)$ time." [1] Definiera en förenklad form av klassen som bara har följande metoder.

   - Konstruerarmetoden

     `ArrayList()`

     med specifikationen "Constructs an empty list with an initial capacity of ten".

   - Metoden

     `void add(int index, Object element)`

     med specifikationen "Inserts the specified element at the specified position in this list. Shifts the element currently at that position (if any) and any subsequent elements to the right (adds one to their indices)."

   - Metoden

     `void ensureCapacity(int minCapacity)`

     med specifikation "Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument."

   För full poäng krävs detaljerad objekt-orienterad pseudokod. Korrekt Javakod går förstås också bra. Du ska också motivera varför din `add`-metod har konstant amorterad komplexitet. (8p)

[1] Detta gäller bara när man lägger till ett element sist i fältet. Texten i Java Collections underförstår detta.

Svar:

```
class ArrayList {
   Object[] array;
   int size;

   ArrayList() {
      array = new Object[10];
      size = 0;
   }

   void add(int index, Object element) throws OutOfBoundsException {
      if (index < 0 || index > size) throw OutOfBoundsException;
      if (size == array.length) ensureCapacity(2*size);
      for (int i = size; i > index; i--) array[i] = array[i-1];
      array[index] = element;
      size++;
   }

   void ensureCapacity(int minCapacity) {
      if (array.length >= minCapacity) return;
      Object[] newarray = new Object[minCapacity];
      for (int i = 0; i < size; i++) newarray[i] = array[i];
      array = newarray;
   }
}
```

Som vi visade på föreläsningarna kan vi få den önskade amorterade komplexiteten hos `add`-metoden (för fallet att vi lägger till ett nytt element sist i fältet) om vi fördubblar kapaciteten när fältet blir fullt.

5. I videon vi såg under första föreläsningen frågade en Google-chef Barack Obama hur man sorterar en miljon 32-bitars heltal på mest effektiva sätt.

   (a) Obama svarade "bubble sort would not be the way to go". Varför är detta ett bra svar? Motivera med asymptotisk komplexitet! (2p)

   Svar: Bubble sort är $O(n^2)$ vilket är långsamt i förhållande till de mer effektiva $O(n \log n)$-algoritmerna.

   (b) Exakt hur många jämförelser gör bubble sort när man sorterar en miljon heltal? Du kan utgå från den enklaste versionen av bubble sort som gör lika många jämförelser som urvalssortering ("selection sort"). (2p)

   Svar: Bubble sort (och selection sort) gör $(n-1) + (n-2) + \cdots + 2 + 1 = n(n-1)/2$ jämförelser. Om $n = 1\,000000$ så blir alltså antalet jämförelser $1\,0001\,000 * 999\,999/2 = 499\,999\,500\,000$, dvs nästan 500 miljarder jämförelser.

   (c) Vad är bästa svaret på Google-chefens fråga? Observera att det inte räcker för full poäng att säga vilken sorteringsmetod som är bäst utan du måste även motivera varför denna metod är bättre än alla andra sorteringsmetoder för just detta problem. Tänk framför allt på att jämföra huvudkandidaterna! Ditt svar poängsätts efter hur övertygande utredning du gör av detta problem, och alltså efter hur imponerad Google-intervjuaren skulle vara av dina kunskaper. Utgå från asymptotiska komplexiteter, men försök även väga in vilken algoritm som troligen har bäst "konstant". Skriv max en sida och tänk på att bara ge information som är relevant för frågans besvarande. (6p)

   Svar: Huvudkandidaterna är effektiva $O(n \log n)$-algoritmer som quicksort och heapsort (och eventuellt mergesort) å ena sidan och radix sort som är $O(d(n+N))$ å andra sidan. Här är $n = 1\,000\,000$ så $n \log n \approx 20\,000\,000$. Vidare är d antalet dimensioner och N antalet hinkar som radix sort använder. Det är inte så bra att använda radix sort med $d = 32$ och $N = 2$, dvs en version av radix sort där man använder bucket sort för varje bit. Bättre är att använda radix sort med $d = 2$ och alltså använda bucket sort 2 gånger på 16-bitars heltal. Då för vi $N = 2^{16} \approx 64\,000$. Vi får då $d(n+N) \approx dn = 2\,000\,000$. Eftersom konstanten för heapsort eller quicksort knappast kan vara 10 gånger mindre än konstanten för radix sort är alltså radix sort bästa alternativet.

6. (a) Konstruera en så effektiv algoritm som möjligt som kontrollerar att ett heltalsfält med längden n är sorterat. Vilken O-komplexitet har din algoritm i värsta fallet? (2p)

Svar: Att utdatafältet är sorterat är enkelt att kontrollera:

```
static boolean isSorted(int[] a) {
   for (int i = 0; i < a.length; i++)
      if (a[i] > a[i+1]) return false;
   return true;
}
```

Komplexiteten för denna algoritm är $O(n)$.

(b) Konstruera en så effektiv algoritm som möjligt som kontrollerar att två heltalsfält är permutationer av varandra! (Kom ihåg att en permutation av ett fält kan erhållas genom att man låter elementen i fältet byta plats med varandra. Man får inte ta bort eller lägga till element. Två fält är alltså permutationer av varandra om de innehåller samma multimängd av heltal - de kan innehålla flera förekomster av ett element.)

För full poäng ska din algoritm ha så god asymptotisk komplexitet som möjligt i värsta fallet. Om du använder standardalgoritmer från kursen behöver du inte ge pseudokod för dem. Vilken asymptotisk komplexitet har din algoritm i värsta fallet? (4p)

Svar: En effektiv metod är att använda en hashtabell för att lagra multimängden av element i det ena fältet. Sedan går man genom det andra fältet och kontrollerar att dess element alla förekommer i denna multimängd. Varje gång man kontrollerar förekomsten av ett element tar man bort det (räknar ner antalet förekomster) från hashtabellen. Om fälten är permutationer av varandra, dvs de om de innehåller samma multimängd, kommer denna process att sluta med att hashtabellen är tom. Pseudokod:

```
bool isPermutation(int a[], int b[]):
    if a.length != b.length: return false // optional optimization
    h = new hashtable
    for x in a:
        h.set(x, h.get(x,0) + 1)
    for x in b:
        if x not in h:
            return false
        h.set(x, h.get(x) - 1)
    for x in a:
        if h.get(x) != 0:
            return false
    return true
}
```

Komplexiteten för denna algoritm blir $O(n)$ om vi gör antagandet att insättning (`set`) och sökning (`get`) är $O(1)$ (amorterat).

(c) Man kan testa att en sorteringsalgoritm är korrekt genom att exekvera algoritmen på ett antal olika indatafält och testa följande två egenskaper:

- att utdatafältet är sorterat;
- och att utdatafältet är en permutation av indatafältet.

Man kallar dessa två egenskaper en *specifikation* av sorteringsalgoritmen. De talar om *vad* algoritmen ska göra, men inte *hur*.

Ett bra testprogram är effektivt och man kan alltså använda dina algoritmer i (a) och (b) för att testa sorteringsalgoritmer. Det är dock ännu viktigare att ett testprogram är korrekt än att det är effektivt. Är dina program i (a) och (b) lämpliga ur denna synvinkel? Varför eller varför inte? Motivera kort. (2p)

Svar: Programmet `isSorted` är enkelt att implementera korrekt. Programmet `isPermutation` är också relativt enkelt att implementera; vi kan t ex använda oss av en hashtabellimplementering från något standardbibliotek. Om vi inte vill göra oss beroende av korrektheten hos någon hashtabellimplementering kan vi i stället skriva `isPermutation` på ett mer direkt sätt. Man kan t ex gå genom elementen i indatafältet ett efter ett och kontrollera att de finns med i utdatafältet och om så är fallet ta bort dem. Effektiviteten blir dock sämre: $O(n^2)$.