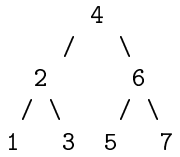
Datastrukturer för D2 (TDA 131)
tentamen med lösningar

21 december 2002

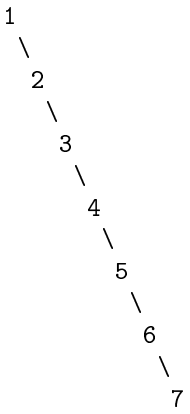
1. Antag att du ska lagra talen i mängden $\{1, 2, 3, 4, 5, 6, 7\}$ i ett binärt sökträd. Först ska du använda algoritmen för insättning i binära sökträd (se t ex Goodrich och Tamassia 9.1 eller Budd sid 357-365) och sedan ska du använda algoritmen för insättning i AVL-träd (se t ex Goodrich och Tamassia 9.2 och Budd 14.2). Vilket binärt sökträd eller AVL-träd man får beror förstås på i vilken ordning man sätter in elementen. Vi ska nu titta på de bästa och värsta fallen.

   (a) Allmänna binära sökträd kan bli mycket obalanserade. Varför vill man undvika detta och se till att trädets höjd blir så liten som möjligt? (2p)
   Svar. Värstafallskomplexiteten för de viktigaste operationerna på binära sökträd (sökning, insättning, borttagning) är $O(h)$ där h är trädets höjd.

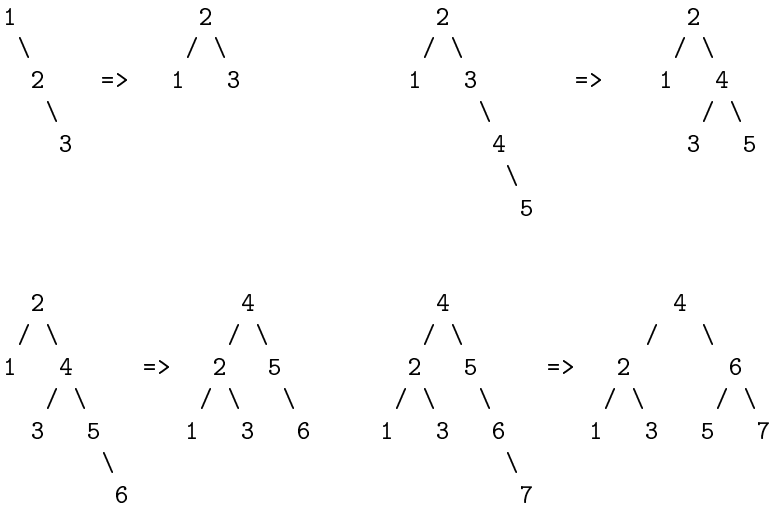
   (b) Vilken är den minimala höjden hos ett binärt sökträd som innehåller elementen $\{1, 2, 3, 4, 5, 6, 7\}$? Ange i vilken ordning man kan sätta in elementen så att du får denna minimala höjd! (Flera ordningar är möjliga, det räcker om du anger en!) Rita också det resulterande trädet! (2p)
   Svar. Den minimala höjden är 2. Om man t ex sätter in elementen i ordningen 4,2,6,1,3,5,7 får man trädet



(c) Vilken är den maximala höjden hos ett binärt sökträd som innehåller elementen $\{1, 2, 3, 4, 5, 6, 7\}$? Ange i vilken ordning man kan sätta in elementen så att du får denna maximala höjd! (Flera ordningar är möjliga, det räcker om du anger en!) Rita också det resulterande trädet! (2p)
Svar. Den maximala höjden är 6. Om man t ex sätter in elementen i ordningen 1,2,3,4,5,6,7 får man trädet



(d) Algoritmen för insättning i AVL-träd ser till att träden alltid är höjdbalanserade genom att strukturera om dem när det behövs. Vilket är det maximala antalet omstruktureringar du kan behöva göra när du sätter in talen $\{1, 2, 3, 4, 5, 6, 7\}$? Ange vilken ordning elementen ska komma för att få detta värsta fall! Rita en bild på varje omstrukturering du behöver göra! (3p)
Svar. Om elementen sätts in i ordningen 1,2,3,4,5,6,7 behöver man göra 4 omstruktureringar:



Detta är det maximala antalet. Omstruktureringar behöver uppenbarligen aldrig göras efter insättning av det första och det andra elementet. Omstrukturering behöver inte heller göras efter insättning av det fjärde elementet eftersom ett treelements AVL-träd alltid är perfekt balanserat, dvs vänster och höger delträd har samma höjd.

(e) Vad är det värsta som kan hända om du sätter in talen $\{1, 2, 3, 4, 5, 6, 7\}$ i en skiplista? Med "värsta" menar vi här värsta fallet med avseende på sökning efter ett visst element i skiplistan. Vad är den asymptotiska komplexiteten (uttryckt i O-notation) för värsta fallet hos denna operation som funktion av antalet element n i den mängd som lagrats i listan? (Mängden $\{1, 2, 3, 4, 5, 6, 7\}$ innehåller alltså 7 element, även om skiplistan lagrar flera kopior av elementen.) Svaren på ovanstående frågor kan bero på exakt hur man valt att implementera skiplistorna. Diskutera! (3p)

Svar. Om höjden av skiplistan är h så är komplexiteten hos sökning $O(h + n)$ i värsta fallet, eftersom vi behöver göra h `drop down` och n `scan forward` operationer (see G & T sid 365) i värsta fallet. Ett exempel på ett sådant värsta fall är att vi söker efter 7 i skiplistan

```
1
|
1
|
: h nivåer
|
1
|
1 - 2 - 3 - 4 - 5 - 6 - 7
```

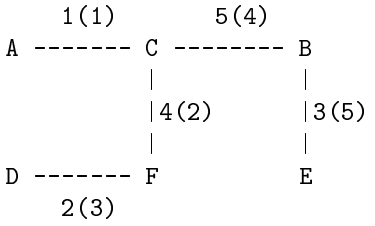
Om vi inte begränsar antalet nivåer i våra skiplistor, kan h bli godtyckligt stort i förhållande till n. Om vi däremot begränsar antalet nivåer så att t ex h är $O(\log n)$ (som föreslås i G & T sid 368), så blir värsta fallet $O(\log n + n) = O(n)$.

2. (a) Personerna A, B, C, D, E och F bor tillsammans i ett hus. De vill koppla samman sina datorer, men vill använda så lite kabel som möjligt. Nätverket ska alltså bilda en *sammanhängande* graf, där summan av alla bågars vikter är så liten som möjligt. Minimala kabelavståndet mellan två datorer ges av följande grannmatris:

	A	B	C	D	E	F
A	0	6	1	5	7	5
B	6	0	5	10	3	9
C	1	5	0	5	6	4
D	5	10	5	0	8	2
E	7	3	6	8	0	6
F	5	9	4	2	6	0

Rita en bild som visar hur A, B, C, D, E och F bör koppla samman sina datorer! Förklara också hur algoritmen du använt fungerar genom att rita bilder som visar hur algoritmen steg för steg konstruerar kabelnätet. (5p)

Svar. Grannmatrisen ovan representerar en viktad oriktad graf G (vi kan bortse från självlooparna med vikt 0). Vi använder oss av Prims algoritm på G med A som startnod (se 12.7.2 i G & T för en beskrivning av algoritmen). Den genererar följande minsta uppspännande träd:



Vi har markerat bågarna med $v(n)$ där v är vikten och n är ordningen i vilken Prims algoritm lagt till bågen till nätverket. T ex betyder markeringen `5(4)` på bågen mellan C och B att vikten är 5 och detta var den fjärde båge algoritmen lade till.

(b) Man kan använda Dijkstras algoritm för att finna kortaste vägen mellan två noder i en viktad graf. Antag att alla vikterna (avstånden mellan två grannoder) är 1. Hur kan vi då förenkla Dijkstras algoritm? Vilken asymptotisk komplexitet har den förenklade algoritmen? Du ska ange O-komplexitet som en funktion av antalet bågar och antalet noder i grafen. (5p)

Svar. Om alla vikterna är 1 kan vi använda en kö i stället för en prioritetskö (t ex en heap) för att lagra kandidatnoderna. (I G & T sid 587 heter prioritetskön Q och lagrar alla noder utanför "molnet" av färdiga noder till vilka kortaste vägen redan är bestämd.) Dijkstras algoritm degenererar i detta fall till bredden först sökning.

Enligt G & T sid 591 har Dijkstras algoritm komplexiteten $O((n + m)\log n)$ där n är antalet noder och m är antalet bågar. Anledningen är att vi plockar ut minsta elementet ur prioritetskön n gånger och uppdaterar prioritetskön m gånger och varje utplockning och uppdatering har komplexiteten $O(\log n)$ om prioritetskön implementeras med en heap. Om däremot prioritetskön ersätts med en kö har både utplockning och uppdatering komplexiteten $O(1)$. Alltså förbättrar vi komplexiteten till $O(n + m)$.

(c) Antag att du inte behöver returnera kortaste vägen, utan bara vill veta längden hos den kortaste vägen. Hur påverkas då den asymptotiska komplexiteten hos Dijkstras algoritm? (2p)

Svar. Det påverkar inte den asymptotiska komplexiteten. Algoritmen på sid 587 i G & T returnerar bara längden hos den kortaste vägen. Om vi dessutom vill returnera själva kortaste vägen (en sekvens av noder t ex), så lagrar vi för varje nod förutom preliminära avstånd (D-värden i G & T) även information om föregångarnod längs den preliminärt kortaste vägen. Komplexitetsanalysen för den algoritm som även returnerar kortaste vägen blir identisk med den som bara returnerar kortaste avståndet: varje uppdatering och urplockning blir fortfarande $O(\log n)$, endast konstanten påverkas av att man måste hålla reda på föregångarnoderna.

3. (a) Skriv en Java-metod

```
public int fib(int n) { ... }
```

så att `fib(n)` returnerar det `n`te fibonaccitalet, dvs resultaten ska uppfylla

```
fib(0) = 0
fib(1) = 1
fib(n+2) = fib(n) + fib(n+1)
```

Din metod ska ha asymptotisk tidskomplexitet $O(n)$ under antagandet att tiden det tar att utföra en addition är $O(1)$. (4p)
Svar. Vi skriver en iterativ algoritm:

```
public int fib(int n) {
   int fst = 0;
   int snd = 1;
   int current = 0;
   if (n == 1) {current = 1;};
   for (int i = 2; i <= n; i++) {
   current = fst + snd;
   fst = snd;
   snd = current;
   };
   return current;
}
```

(b) Låt $T(\texttt{n})$ vara antalet primitiva operationer din metod utför när den beräknar `fib(n)`. Vad är $T(0), T(1)$ och $T(2)$? Du ska alltså räkna antalet additioner, antalet tilldelningar, antalet jämförelser, osv! Skriv ner vilka antaganden du gör om vad som utgör en primitiv operation och skriv ner exakt vilka primitiva operationer din `fib`-metod använder! (3p)
Svar. Vi räknar följande primitiva operationer:

- tilldelning (inkl initiering av lokal variabel);
- operationerna $==, <=, +, ++$;
- returnera värdet av en variabel.

(En mer exakt beräkning kräver kunskaper om hur Java kompileras, vi försöker här analysera primitiva operationer på ungefär samma nivå som i 3.5.1 i G & T.)

Med dessa antaganden får vi $T(0) = 7$; vi markerar i algoritmen:

```
public int fib(int n) {
   int fst = 0;                      -- 1
   int snd = 1;                      -- 1
   int current = 0;                  -- 1
   if (n == 1) {current = 1;};       -- 1
   for (int i = 2; i <= n; i++) {    -- 1 + 1
   current = fst + snd;
   fst = snd;
   snd = current;
   };
   return current;                   -- 1
}
```

$T(1) = 8$. Som ovan fast vi exekverar också `{current = 1;}`.
$T(2) = 13$

```
public int fib(int n) {
   int fst = 0;                      -- 1
   int snd = 1;                      -- 1
   int current = 0;                  -- 1
   if (n == 1) {current = 1;};       -- 1
   for (int i = 2; i <= n; i++) {    -- 1 + 2 + 1
   current = fst + snd;              -- 1 + 1
   fst = snd;                        -- 1
   snd = current;                    -- 1
   };
   return current;                   -- 1
}
```

(c) Ställ sedan upp rekursionsekvationer för $T(\texttt{n})$! Dvs definiera $T(\texttt{n})$ genom att säga vad $T(0)$ är och hur man kan räkna ut $T(\texttt{n}+1)$ om man vet $T(\texttt{n})$. Motivera varför $T(\texttt{n})$ är $O(\texttt{n})$! (3p)
Svar. $T(0), T(1)$, och $T(2)$ angavs i (b). För $\texttt{n} > 2$ får vi $T(\texttt{n}+1) = \texttt{T(n)} + 6$. Alltså blir $T(\texttt{n}) = 6\texttt{n} + 1 \leq 7\texttt{n}$ för $\texttt{n} \geq 2$. Det följer att $T(\texttt{n})$ är $O(\texttt{n})$.

(d) Man kan också mäta tidskomplexiteten som funktion av längden `m` av talet `n` i binär representation. Exempelvis har talet 3 binärrepresentationen 11 som har längden 2, och talet 12 har binärrepresentationen 1100 som har längden 4. Låt $T'(\texttt{m})$ vara maximala antalet operationer din algoritm utför för att beräkna `fib(n)` där `m` är längden av `n` i binärrepresentation. Ange lämplig O-komplexitetsklass för $T'(\texttt{m})$! (2p)
Svar. $T'(\texttt{m})$ är $O(2^{\texttt{m}})$ eftersom `n` är $O(2^{\texttt{m}})$.

(e) Antagandet att tiden det tar att beräkna en addition är $O(1)$ förutsätter att summan verkligen kan lagras som en `int` utan att man får "overflow". Hur gör man om man vill addera godtyckligt stora heltal? Man vet alltså inte på förhand hur många binära siffror man behöver. Vilken O-komplexitet har addition is så fall? Uppskatta också O-komplexiteten för din `fib`-metod om den använder denna metod för att addera stora heltal? (2p)
Svar. Man kan addera godtyckligt stora heltal genom att använda den algoritm som man lärde sig i lågstadiet. (Väsentligen samma algoritm för binära tal används ju i hårdvaruadderare, som dock bara klarar tal av begränsad storlek.) Tidskomplexiteten hos lågstadiealgoritmen är $O(\texttt{m})$ eller $O(\log \texttt{n})$ där `n` är maximum av de två talen och `m` är antalet siffor i `n`.

Tidskomplexiteten för fibonaccifunktionen som klarar godtyckligt stora n blir alltså $O(\texttt{n}\log(\texttt{fib n}))$. Eftersom `fib n` är $O(2^{\texttt{n}})$ så är den alltså även $O(\texttt{n}^2)$.

4. (a) Antag att du ska göra en hashtabell för ett fastighetsregister för Göteborg. Söknycklarna är fastighetsbeteckningar som består av ett områdesnamn följt av ett 2-siffigt och ett 3-siffigt tal, t ex `Torp 44:106`. Du vet dock inte vilka områdesnamnen är, bara att de är representerade som strängar med maximalt 12 bokstäver. Vi antar vidare att hashtabellens storlek är ett 5-siffrigt primtal. Hitta på en bra hashfunktion! (4p)
Svar. För att få en så bra hashfunktion som möjligt bör man utnyttja alla tre komponenterna (områdesnamnet och de två talen). Om man t ex bara använder de två talen riskerar man att få onödigt många kollisioner genom att det kan hända att samma talpar används i många olika områden. Sedan gäller det att kombinera dem på ett lämpligt sätt. Först konverterar vi områdesnamnet till ett heltal i t ex genom att först använda ASCII värdena för bokstäverna i namnet och sedan använda en polynomisk hashfunktion (se G & T sid 345). Sedan beräknar vi hashfunktionen

$$h(i, j, k) = (i + 1000j + k) \bmod N$$

där j är det 2-siffriga talet, k är det 3-siffriga talet och N är hashtabellens storlek.

(b) Vilka av följande abstrakta datatyper är lämpliga och vilka är olämpliga att implementera med hashtabeller:

i. prioritetsköer,

ii. mängder,

iii. multimängder?

Motivera alla svar!

Anm. Multimängder (multisets) kallas också påsar ("bags") och skiljer sig från mängder genom att de kan ha flera kopior av samma element. Multimängden $\{1, 1, 2\}$ skiljer sig alltså från påsen $\{1, 2\}$. Operationerna på multimängder motsvarar operationerna på mängder, t ex union, snitt, om ett visst element finns i mängden, osv. Skillnaden är dock att multimängdsoperationerna måste hålla reda på hur många förekomster det finns av ett visst element. (4p)

Svar.

i. Det är olämpligt att implementera prioritetsköer med hashtabeller. För att finna minsta elementet måste man söka genom hela hashtabellen.

ii. Det kan ofta vara lämpligt att implementera mängder med hashtabeller. Elementrelationen ($\in$) är ju ingenting annat en sökning, och poängen med hashtabeller är ju bl a att denna operation i praktiken är $O(1)$. Betrakta vidare två mängder som implementerats med hinkhashning med samma hashfunktion. Vi kan nu implementera unionen helt enkelt genom att ta unionen av hinkar med samma index! Denna implementering har komplexiteten $O(N)$, där N är hashtabellens storlek och vi förutsätter att vi har en bra hashfunktion så att antalet kollisioner är litet. Detta är bra om inte N är alltför stort i förhållande till n, antalet lagrade element. På samma sätt kan snittet beräknas genom att ta snittet av hinkar med samma index. (Implementeringen av mängder i G & T avsnitt 10.2.1 använder ju sorterade sekvenser. Union och snitt kan då beräknas med "generic merge" och har komplexiteten $O(n + n')$ där n och n' är antalet element i de två sekvenserna. Vidare är (binär) sökning efter ett element i en sorterad sekvens $O(\log n)$.)

iii. Man kan implementera multimängder med hjälp av hashtabeller på liknande sätt som man kan implementera mängder. Skillnaden är att när man implementerar mängder lagrar man bara elementen (söknycklarna) medan när man implementerar multimängder lagrar man par av element och tal som representerar antalet förekomster av elementet i fråga. Även här blir elementen söknycklar i hashtabellen. Anledningen till att det ofta är lämpligt att implementera multimängder med hashtabeller är därför väsentligen desamma som för mängder, eftersom operationerna kan implementeras på liknande sätt.

(c) Du har bestämt dig för att använda en hashtabell, men du vill också veta värstafalls-komplexiteten för sökning, insättning, och borttagning av element. Hur beror värstafallskomplexiteten på antalet lagrade element n och på hashtabellens storlek N för

i. öppen adressering; (Här får du förutsätta att belastningsfaktorn är mindre än 1, dvs att hashtabellen inte är full).

ii. hashning med hinkar ("hashing in buckets, chained hashing"), där hinkarna implementerats med länkade listor. (4p)

Svar.

i. Öppen adressering. Vi antar först att inga borttagningar har gjorts så att hashtabellen inte innehåller några "gravstenar". I så fall är värsta fallet att hashfunktionen givit samma hashkod till alla element och därför placerat alla elementen efter varanda (antag för enkelhets skull linjär probing). Sökning och borttagning är då $O(n)$ (vi måste i värsta fall leta genom hela sekvensen). Insättning är $O(n)$ (vi måste alltid stega oss genom hela sekvensen för att hitta en ledig plats).
Om även gravstenar finns kan vi i värsta fall behöva leta genom hela hashtabellen. Alla tre operationerna blir därför $O(N)$.

ii. Hashning med hinkar, där hinkarna implementerats med länkade listor. Även här är värsta fallet att hashfunktionen givit samma hashkod till alla elementen och vi alltså bara har en hink (länkad lista). Komplexiteten är därför densamma som för motsvarande operationer på länkade listor, dvs insättning är $O(1)$ och sökning och borttagning är $O(n)$.

5. Givet två sorterade vektorer, båda av längd n, konstruera en algoritm (i pseudokod) som returnerar det lägre medianelementet i den sammanslagna vektorn! Om du t ex har vektorerna $[1, 2, 5]$ och $[3, 4, 6]$ är alltså 3 "det lägre medianelementet" och 4 "det högre medianelementet", eftersom den sammanslagna (sorterade) vektorn är listan $[1, 2, 3, 4, 5, 6]$. För att få full poäng ska din algoritm ha komplexiteten $O(\log n)$ och du ska motivera varför den har denna komplexitet. Du kan dock få delpoäng för en mindre effektiv algoritm, om du gör en korrekt komplexitetsanalys. Antalet delpoäng beror då på hur effektiv din algoritm är. (10 p)
Svar. Vi använder "söndra och härska" ("divide and conquer") strategin för att få en enkel och effektiv algoritm på följande sätt.

Kalla de två vektorerna $a = a[0], \ldots, a[n-1]$ och $b = b[0], \ldots, b[n-1]$.

Bafallet $n = 1$ beräknas genom att välja det mindre av $a[0]$ och $b[0]$.

Om $n > 1$ är udda och $m = (n-1)/2$ kan vi beräkna medianelementen $a[m]$ och $b[m]$ i konstant tid. Vi observerar nu att det lägre medianelementet x för a och b måste ligga i intervallet $a[m] \leq x < b[m]$ om $a[m] < b[m]$ och i intervallet $b[m] \leq x < a[m]$ om $b[m] < a[m]$. Om $a[m] = b[m]$ så är $x = a[m] = b[m]$.

Det lägre medianelementet för a och b kan alltså beräknas genom att jämföra $a[m]$ och $b[m]$ och

- om $a[m] < b[m]$ beräkna det lägre medianelementet för $a[m], ..., a[n-1]$ och $b[0], \ldots, b[m]$. (För att kunna anropa algoritmen rekursivt behöver vi två lika långa vektorer. Därför tar vi med det redan eliminerade värdet $b[m]$.)
- om $b[m] < a[m]$ beräkna det lägre medianelementet för $b[m], ..., b[n-1]$ och $a[0], \ldots, a[m]$.
- om $a[m] = b[m]$ returnera $a[m]$.

Om n är jämt och $l = n/2 - 1$ kan vi beräkna de lägre medianelementen $a[l]$ och $b[l]$ på konstant tid. Här observarar vi att $a[l] < x \leq b[l]$ om $a[l] < b[l]$ och i intervallet $b[l] \leq x < a[l]$ om $b[l] < a[l]$. Om $a[l] = b[l]$ så är $x = a[l] = b[l]$.

Det lägre medianelementet för a och b kan alltså beräknas genom att jämföra $a[l]$ och $b[l]$ och

- om $a[l] < b[l]$ beräkna det lägre medianelementet för $a[l+1], ..., a[n-1]$ och $b[0], \ldots, b[l]$.
- om $b[l] < a[l]$ beräkna det lägre medianelementet för $b[l+1], ..., b[n-1]$ och $a[0], \ldots, a[l]$.
- om $a[l] = b[l]$ returnera $a[l]$.

Vi visar även en implementering av algoritmen i Java. Den skiljer sig på några punkter från den principiella skissen ovan. T ex sparar vi kod genom att utnyttja gemensamma delar hos det udda och det jämna fallet. Vidare är algoritmen "in-place": vi använder särskilda variabler för att markera undre (`a_lo`, `b_lo`) och övre (`a_hi`, `b_hi`) gräns för de delvektorer som algoritmen arbetar på vid ett visst tillfälle. På så sätt behöver vi inte kopiera elementen i vektorerna.

```
int median(int[] a, int[] b)
{
    // Vet att 'a.length == b.length'
    int a_lo = 0, a_hi = a.length-1;
    int b_lo = 0, b_hi = b.length-1;

    while (a_lo != a_hi){
        int a_mid = (a_lo + a_hi) / 2;
        int b_mid = (b_lo + b_hi) / 2;
        if (a[a_mid] <= b[b_mid]){
            a_lo = a_mid;
            b_hi = b_mid;
            if (a_hi-a_lo > b_hi-b_lo) a_lo++;  // (håll intervallen lika stora)
        }else{
            a_hi = a_mid;
            b_lo = b_mid;
            if (b_hi-b_lo > a_hi-a_lo) b_lo++;  // (håll intervallen lika stora)
        }       }
    return min(a[a_lo], b[b_lo]);
}
```

Algoritmen har komplexiteten $O(\log n)$ eftersom exekveringstiden i värsta fallet är proportionell mot antalet gånger vi behöver halvera n för att komma till 1.