Lösningsförslag till tentamen i ID1020, P2 2015

Måndagen 2015-01-19 kl 8.00–12.00

Examinator: Johan Karlander, Jim Dowling
Hjälpmedel: Inga
Tentamensfrågorna behöver inte återlamnas efter avslutad tentamen.

Varje inlämnat blad skall förses med följande information:

1. Namn och personnummer
2. Nummer for behandlade uppgifter
3. Sidnummer

Tentamen består av två delar. Del I innehåller tio uppgifter för betyget E. Del II, som inte är obligatorisk, innehåller fyra uppgifter för betygen A-D. Maximal skrivtid är fyra timmar.

Varje lösning kommer att bedömas med 0-10 poäng.

Kravet för godkänt (E) är 50 poäng på del I. Du måste bli godkänd på del I för att kunna erhålla högre betyg. Poängen från del I förs inte vidare till del II.

Under förutsättning att du är godkänd på del I gäller följande betygsgränser för del II:

- Betyg D: 5-10 poäng
- Betyg C: 11-20 poäng
- Betyg B: 21-30 poäng
- Betyg A: 31-40 poäng

Lösningar anslås på kursens hemsida.

Uppgifter Del 1

Följande uppgifter avser betyget E. Krav på *fullständiga lösningar med motiveringar* gäller för samtliga uppgifter.

1. Skriv en rekursiv implementation av Fibonacci-funktionen nedan antingen i Java eller pseudokod. Sedan, i Java, jämför minneskomplexiteten för Fibonacci-funktionen nedan med din rekursiva lösningen.

```
public int fibonacciLoop(int number) {
    if (number == 1 || number == 2) {
        return 1;
    }
    int fibo1 = 1, fibo2 = 1, fibonacci = 1;
    for (int i = 3; i <= number; i++) {
        //Fibonacci number is sum of previous two Fibonacci numbers
        fibonacci = fibo1 + fibo2;
        fibo1 = fibo2;
        fibo2 = fibonacci;
    }
    return fibonacci; //Fibonacci number
}
```

Lösningen

```
public long fibonacci(int i) {
    if (i == 0)
        return 0;
    if (i <= 2)
        return 1;
    return fibonacci(i - 1) + fibonacci(i - 2);
}
```

(8 pts)

Den rekursiva implementation har högre minneskomplexitet i Java eftersom stacken växer för högre input värden. (2 pts)

2. Stabilitet är en viktig egenskap hos sorteringsalgoritmer. Ge ett exempel på en stabil sortering och en sortering som inte är stabil. Ange en sorteringsalgoritm som ger en stabil sortering och en annan som inte ger en stabil sortering. Ange en sorteringsalgoritmen som har tidskomplexitet $O(N * logN)$ i värsta fall.

Lösningen Se föreläsningar för exempel på stabila och ej stabila sorteringar. (3 pts)

Sortering algoritmer som är stabila: insertion sort och mergesort. Sortering algoritmer som inte är stabila: selection sort, Quicksort och shell sort. (3 pts)

Mergesort har tidskomplexitet $O(N * logN)$. (4 pts)

3. Ange tidskomplexiteten i värsta fall (worst-case) för följande javakodstycken. Skriv dina svar i tilde-notation.

   (a)

```
for (int i = 0; i < n * n; i++)
    sum++;
```

   (b)

```
for (int i = n; i > 0; i=i/2)
    sum++;
```

   (c)

```
for (int i = 0; i < n * n; i++)
  for (int j = i; j < n; j++)
    sum++;
```

   (d)

```
for (int i = 0; i < n * n; i++)
  for (int j = 1; j < n; j = j*2)
sum++;
```

   Lösningen (2,5 pts each)

   Notera att tilde-notation är efterfrågat.

   (a) O(N^2)

   (b) O(log(N))

   (c) O(N^3).

   (d) O(log(N^2)log(N)) (Notera att j=1.)

4. Skriv (i Java eller pseudokod) en algoritm för att hitta en element i en sorterad array med unika element i logaritmisk tidskomplexitet (i värsta fall).

   Lösningen (10 pts)

```
public static int binarySearch(int[] a, int key)   {
     int lo = 0, hi = a.length-1;
     while (lo <= hi) {
         int mid = lo + (hi - lo) / 2;
         if (key < a[mid]) {
             hi = mid - 1;
         }
         else if (key > a[mid]) {
             lo = mid + 1;
         }
```

```
        else {
            return mid;
        }
    }
   return -1;
}
```

5. Vi har en oriktad graf $G = (V, E)$ och en viss kant e angiven. Beskriv en algoritm som avgör om e ingår i någon cykel i G eller inte.

   Lösning.
   Antag att $e = (a, b)$. En enkel lösning är att vi tar bort kanten e så att vi får grafen $G' = G - e$. Vi gör sedan en en DFS-sökning från a. Om det går att nå b från a så ingår e i någon cykel, annars inte. Tidskomplexiteten är samma som för DFS, d.v.s. $O(|V| + |E|)$.

6. Låt G vara en sammanhängande graf med ett minimalt spännande träd T. Låt G' vara en *inducerad* delgraf i G. Det betyder att $V(G') \subseteq V(G)$ och att varje kant som finns i G' också finns i G.

   Avgör om kantmängden $E(T) \cap E(G')$ alltid utgör ett minimalt spännande träd i grafen G' eller inte.

   Lösning
   Svaret är nej. Detta visas enklast med ett motexempel. Enklast tänkbara exempel är att vi har en graf G med minst 3 noder och ett MST sådant att en viss kant (a, b) inte ingår i trädet. Bilda sedan grafen G' med $V(G') = a, b$. Då ser vi att $E(T) \cap E(G') = \emptyset$ och inte ett spännande trär i G'.

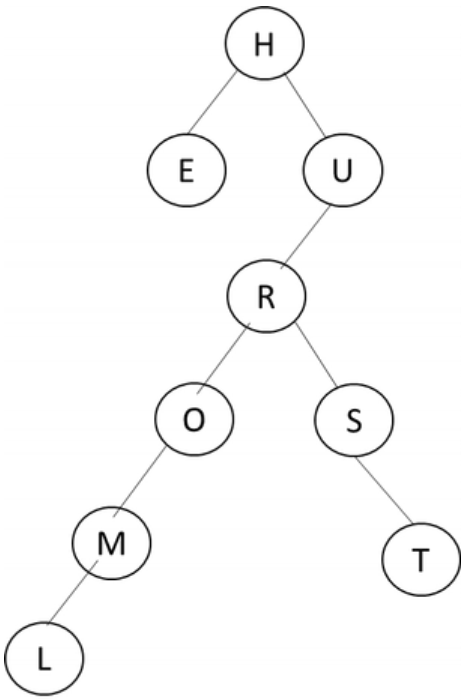
7. Antag att G är en riktad graf med n noder och n kanter. Antag vidare att den underliggande (oriktade) grafen är sammanhängande. Ange, med utförlig motivering, vad det största och det minsta antal starkt sammanhängande komponenter som G kan ha är.

   Lösning
   Det minsta antalet är 1 och det största är n. Detta visa kan visas med exempel. I båda fallen tänker vi oss att vi startar med en oriktad cykel med n noder. Om vi lägger på riktningar så att vi får en riktad cykel får vi uppenbart en riktad graf i en sammanhängande komponent. Om vi sedan tar denna graf och vänder riktningen på *en* av kanterna så får vi en graf som är sådan att om det finns en väg från a till b så finns det inte en väg från b till a. I en sådan graf är varje nod en maximal sammanhängande delgraf. Grafen har alltså n sammanhängande komponenter.

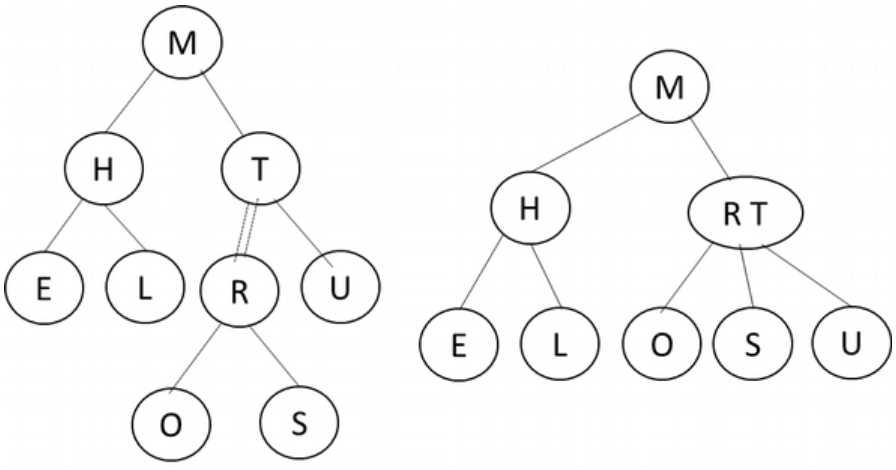
8. Rita trädet när man matar in nycklarna H U R S O M H E L S T i ett binärt träd. Här menas ett enkel obalanserat träd. Använd bokens konvention när det gäller duplikat, d.v.s. att man skriver över. Hur många jämföroperationer görs totalt? Rita trädet när man matar in nycklarna H U R S O M H E L S T i ett balanserat röd-svart binärt träd. Använd bokens konvention när det gäller duplikat, d.v.s. att man skriver över. Hur många jämföroperationer görs totalt. Hur många rotationer görs totalt?

Lösning

Svar: a) (4 poäng för trädet, 2 poäng för antalet jamförelser)

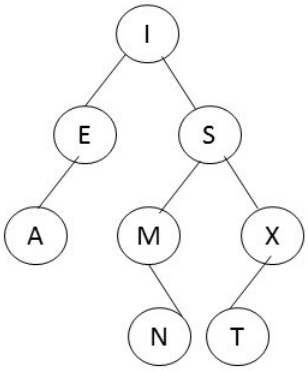


Jamför operationer (där man jamför nycklarna med varandra) H:0, U:1, R:2, S:3, O:3, M:4, H:1, E:1, L:5, S: 4, T:4 Totalt: 28 Svar: b) 2 poäng för trädet, 1 för jamförelser, 1 för rotationer



Till vänster ritat direkt som röd-svart BST (med en enda röd länk) Till höger som 2-3 träd (ekvivalent) Till varje element ges paret C:R - jamför (compare): rotationer H:(0:0), U (2:1), R(2,2), S(2,0), O(2,1),M (3,2), H (3,0), E (3, 0), L(3,1), S(4,0), T(4, 3) Totallt 28 jämförelser, och 10 rotationer

9. Skriv svaren och sekvensen av de undersökta noderna i trädet vid följande operationer



   a. floor (P)
   b. select(5)
   c. ceiling(S)
   d. rank(M)
   e. size(F,O)

   Lösning
   a) N (I,S,M,N) b) S (I,S) c) S (I, S) d) 3 (I, S, M) e) 3 (I, E, S, M, N)

10. Du har tillgång till en linjärsonderingshashtabell (linear probling hash table). Tabellen innehåller N element och storleken på den underligande arrayen är $2N$. Beräkna tidskomplexiteten i följande fall, både förväntat och värsta fall. (expected and worst case).

    a. En sökning med träff där alla element har olika hashkod (2 poäng)
    b. En sökning med träff där alla element har samma hashkod (2 poäng)
    c. En sökning utan träff där alla element har samma hashkod (2 poäng)

d. En sökning med träff där alla element hashar till ett index som är en multipel av M. Fördelningen är jämn mellan de olika möjliga multiplarna. T.ex om M=4 finns det lika många element som hashar till 4, 8, 12, osv. (4 poäng)

Lösning
a) Förväntat: 1 Värsta: 1 (kluster storleken 1) b) Förväntat: N/2 Värsta: N (kluster storleken N) c) Förväntat: N Värsta: N (kluster storleken N) d) Förväntat: M/4 Värsta: M/2 (kluster storleken M/2 om M>1)

Uppgifter Del 2

Följande uppgifter avser betygen A-D. Krav på *fullständiga lösningar med motiveringar* gäller för samtliga uppgifter.

1. Ange tidskomplexiteten i värsta fall för find() och union() metoder i följande javakodstycken. Skriv dina svar i tilde-notation. Ange minneskomplexiteten för följande javakodstycken.

```
public class WeightedQuickUnionUF {
    private int[] id;
    private int[] sz;
    public int count;

    public WeightedQuickUnionUF(int N) {
        count = N;
        id = new int[N];
        sz = new int[N];
        for (int i = 0; i < N; i++) {
            id[i] = i;
            sz[i] = 1;
        }
    }
    public int find(int p) {
        while (p != id[p])
            p = id[p];
        return p;
    }

    public boolean connected(int p, int q) {
        return find(p) == find(q);
    }


    public void union(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ) return;
```

```
        if   (sz[rootP] < sz[rootQ]) { id[rootP] = rootQ; sz[rootQ] += sz[rootP]; }
        else                         { id[rootQ] = rootP; sz[rootP] += sz[rootQ]; }
        count--;
    }
}
```

BubbleSort är en känd sorteringsalgoritm som vi inte gick igenom i kursen. Kommer följande javakodstycken att sortera array:n i stigande eller nedstigande ordning? Ange vilka ändringar som behöver göras för att implementationen ska sortera i omvänd ordningen (dvs. från stigande till nedstigande eller vice versa). Räkna ut tidskomplexitetet i värsta fall (worst-case) för BubbleSort och ange för vilka typer av input BubbleSort har lägste möjliga tidskomplexitet (i bästa fall). Skriv dina svar i tilde-notation.

```
public static void BubbleSort( int [ ] num )
{
     int j;
     boolean flag = true;
     int temp;

     while ( flag )
     {
            flag= false;    //set flag to false awaiting a possible swap
            for( j=0;  j < num.length -1;  j++ )
            {
                   if ( num[ j ] < num[j+1] )
                   {
                            temp = num[ j ];                //swap elements
                            num[ j ] = num[ j+1 ];
                            num[ j+1 ] = temp;
                           flag = true;              //shows a swap occurred
                  }
            }
      }
}
```

Lösningen WeightedQuickUnionUF:

find är log(N) (1 pt)

union är log(N) (1 pt)

Om vi antar en 64-bitars JVM, minneskomplexiteten är: 16 bytes overhead för objektet int[] id (24 + 4N) + 8 bytes (array object) int[] sz (24 + 4N) + 8 bytes (array object) int count (4 bytes)

Roughly: 8N + 76 bytes

(2 pts)

BubbleSort: nedstigande ordning.

Ändra följande raden nedan från:

```
if ( num[ j ] < num[j+1] )
```

till

```
if ( num[ j ] > num[j+1] )
```

(3 pts)

BubbleSort tidskomplexiteten är: O(N^2) (2 pts)

Bästa möjliga input som ger O(N) tidskomplexitet är om array:n är redan sorterad. (1 pt)

2. En filmdatabas är uppbyggd med filmens namn som nyckel. Värdena består av par <filmens betyg, lista-av-skådespelare>. Databasen är byggd på ett röd-svart binärt sökträd.

   a. Skriv ett program (pseudokod) som genererar den inverterade databasen, där nycklarna är skådespelare med namn, och som innehåller en lista av par <filmens-namn, filmens-betyg>.

   b. Anta att antalet filmer är F, och att antalet unika skådespelare är S. Ge tidskomplexiteten för att skapa den inverterade databasen. Eventuellt behöver du göra ytterligare rimliga antaganden. I så fall ange dem tydligt.

   c. Skriv ett program (pseudokod) som listar de T top rankade skådespelarna. En skådespelares rank är snittbetyg över alla filmer som skådespelaren är med i.

   d. Ange tidskomplexiteten i c) ovan.

   Det viktiga i lösningen är att det är entydigt, och det framgår tydligt vilka datastrukturer och operationer från boken du använder (du får naturligtvis bara använda det som vi har gått igenom i kursen).

   Lösning

   a. FDB är Filmdatabasen

```
BST IDB= new BST()  // IDB is the inverted database
for all E in FDB.keys() {    // E are film name keys
     for all E2 in FDB.get(E).list { // E2 are the actors/actresses
          Pair = IDB.get(E2)
                  if Pair == NULL then
                      {IDB.put( E2,new List(<E2.name,E.grade>)
                  else {
IDB.put(E2),List.add(<E2.name,E.grade>),Pair)}
```

   b. Vi antar att i snitt P skådesplelare per film. Första for-loopen körs F gånger, andra for-loopen i snitt P gånger. Get operationen (4dje raden) har tidkomplexitet O(S). Samma med put operationer. Tidskomplexiteten är då O(F*P*log S). Man kan också anta att P är litet (det är ju vanligtvis bara huvudskådesplarna som är intressanta), så svaret O(F*S) accepteras också.

c. Vi använder oss av en maximum-orienterad prioritetskö. MaxPQ<>pq= new MaxPQ<>(IDB.size()) for all E in IDB.keys() // E is actor key int gradeSum=0; int filmSum=0; for all E2 in IDB.get(E).list gradeSum=gradeSum+E2.grade); filmSum++ pq.insert(E,gradeSum/filmSum); for(i=0,i++,i<T) printPair(pq.max());

d. Man kan anta M=Average number of films per actor. Tidsomplexiteten är då för att skapa prioritetskön O(S*M*log S) är ett. S för yttre loopen, M för listan, och logS för insättning i prioritetskön. Men P och M är relaterat. M=FP/S !. Så att svaret kan också uttryckas O(F*P*logS)

3. Om vi har en riktad graf G som är starkt sammanhängande så kan man nå alla andra noder från vilken nod som helst. även om grafen inte är starkt sammanhängande kan det hända att det finns en nod som är sådan att man kan nå varje annan nod i G från den. Vi kan kalla en sådan nod för en *basnod*. (Inte någon standardbenämning.)

   Antag nu att G är en riktad graf vars underliggande (oriktade) graf är sammanhängande.

   a. Antag först att G är *acyclisk*, d.v.s. att det inte finns någon *riktad* cykel i G. Designa en algoritm som avgör om G har någon basnod.
   b. Antag nu att G inte nödvändigtvis är acyklisk. Designa en algoritm som avgör om G har någon basnod i detta fall. (Det kan t.ex. vara en vidareutveckling av algoritmen från a.)

   För full poäng bör båda algoritmerna ha en tidskomplexitet $O(n)$ där $n = |E|$. Du ska därför också göra en noggrann komplexitetsanalys av dina algoritmer.

   Lösning

   a. Om grafen är acyklisk så kan vi använda DFS för att få en topologisk numrering av grafen. Det gå r i tid $O(|E|)$ (underliggande grafen är sammanhängande). *Om* det finns en basnod b går det att se att den i så fall måste numreras 1. Vi tar nu och undersöker den nod som numrerats 1. Vi gör en DFS-sökning från den noden och ser om alla andra noder går att nå från den. Om de gör det har vi hittat en basnod. Om inte så finns det ingen basnod. Den sista sökningen går i tid $O(|E|)$ vilket också blir den totala komplexiteten.
   b. Antag nu att grafen inte är acyklisk. Det finns flera lösningsmetoder men en som utnyttjar resultatet i a. är att vi först använder algoritmen för att hitta starkt sammanhängande komponenter. Den tar tid $O(|E|)$. Antag att vi får komponenterna $K_1, K_2, ..., K_r$. Vi kan nu bilda en ny graf G'' med komponenterna K_i som noder. Vi drar en kant från K_i till K_j om och endast om det går en kant från någon nod i K_i till någon nod i K_j. Denna nya graf måste vara acyklisk. Vi använder nu metoden från a. för att se om denna graf har någon basnod. Om den har det så är det någon komponent K_a. Då är alla noder i K_a basnoder i G. Om G'' inte har någon basnod finns det ingen i G heller. Detta tar tid $O(|E|)$ att göra.

4. Nedan, hitter du en implementation av Quicksort algoritmen. När man anropar quicksort metoden nedan, skickar man in en input array av heltal och pekare till start och end elementen. I vilken ordning ska input array:n vara i för att den här implementationen av Quicksort tar värstafalletstidskomplexitet? Sänk förväntad tidskomplexitet av algoritmen nedan för input array:n som resulterar i värstafalletstidskomplexiteten. Beräkna tidskomplexiteten av din modifierade algoritmen för samma input array:n. Vad är förväntad tidskomplexitetet av din modifierade algoritmen för slump input? Notera att du inte behöver skriva om hela algoritmen i din lösning, bara de delarna du har gjort ändringar i.

```
public static void main(String[] args) {

    int[] input = ...
```

```
    quicksort(input, 0, input.length - 1);
}

public static void quicksort(int[] input, int s, int e) {
    int pivot = s;
    int startPtr = s;
    int endPtr = e;

    while (s <= e) {
        while (input[s] < input[pivot]) {
            s++;
        }
        while (input[e] > input[pivot]) {
            e--;
        }
        if (s <= e) {
            swap(input, s, e);
            s++;
            e--;
        }
    }
    if (startPtr < e) {
        quicksort(input, startPtr, e);
    }

    if (s < endPtr) {
        quicksort(input, s, endPtr);
    }
}

public static void swap(int[] input, int s, int e) {
    int tmp = input[s];
    input[s] = input[e];
    input[e] = tmp;
}
```

Lösningen Quicksort tar längsta möjliga tid när man (i varje rekursiv anrop) väljer en pivot som antingen är den minsta eller största element som ska sorteras (eller om alla element har samma värde). I det här fallet, när array är redan sorterade i nedstigande ordningen, då tar det O(N^2). (4 pts)

Man kan sänka tidskomplexiteten genom att välja en slump pivot eller median element av 3 element. (4 pts)

Förväntad tidskomplexiteten blir då O($Nlog(N)$) (2 pts)