TENTAMEN I
DATASTRUKTURER OCH ALGORITMER DVG B03

170117 kl. 14:00-19:00

Ansvarig Lärare: Donald F. Ross

Hjälpmedel: Inga. Algoritmerna finns i de respektive uppgifterna eller i bilagarna.

***** OBS *****

Betygsgräns:	Kurs:	Max 60p, Med beröm godkänd 50p, Icke utan beröm godkänd 40p, Godkänd 30p (varav minimum 20p från tentan, 10p från labbarna)
	Tenta:	Max 40p, Med beröm godkänd 34p, Icke utan beröm godkänd 27p, Godkänd 20p
	Labbarna:	Max 20p, Med beröm godkänd 18p, Icke utan beröm godkänd 14p, Godkänd 10p

SKRIV TYDLIGT – LÄS UPPGIFTERNA NOGGRANT

Ange alla antaganden.

---

(1) Ge ett kortfattat svar till följande uppgifter (a)-(j).

(a) I prestanda analys av **linjärsökning** (linear search) vilket resultat skulle förväntas för det slumpmässiga (random) fallet och det värsta fallet? Vad är förhållandet de emellan?
(b) I prestanda analys av **bubble sort** vad är Big-Oh för det bästa, snitt samt värsta fallet?
(c) Skriv **den rekursiva definitionen** av en sekvens.
(d) Skriv **den rekursiva definitionen** av ett binärt träd.
(e) Ge **ett exempel** av an en ”**single left rotation**” i ett AVL-träd.
(f) Ge **ett exempe**l av an en ”**double right rotatio**n” i ett AVL-träd.
(g) Vad är invarianten i ett binärt sökträd?
(h) Vad är invarianten i en heap?
(i) Vad är Big-Oh för hitta (find) i en heap?
(j) I en heap array (som börjar med index 1), hur beräknar man höger och vänster barns position i arrayen?

Totalt 5p

(2) ADT Sekvens

Förklara ingående hur referenserna ”**pPrevious**” och ”**pCurrent**” fungerar i en iterativ implementation av operationerna **lägga till** (add), **ta bort** (remove) och **hitta** (find). Vilka ”**speciella fall**” måste man ta hand om?
Skriv (pseudo)kod till operationerna **lägga till**, **ta bort** samt **hitta** för att illustrera Ditt svar.

5p

(3) Abstraktion

(a) I labben om prestandamätning fanns det fem storleksfall som skulle mätas, nämligen 1024, 2048, 4096, 8196 och 16384. Tre sorteringsalgoritmer (bubble sort, insertion sort samt quicksort) och två sökningsalgoritmer (linear search, bibary search) skulle mätas.

En student har implementerat labben med följande **FEPerf.h** och **BEPerf.h filer**.

```
void FE_Do_All();                                   /* FEPerf.h   */
void FE_RunTest(testtype ftest);
```

```
unsigned int BE_RunTest(testtype ftest, int size);   /* BEFPerf.c */
```

1. Förklara (med text) hur ”**back-enden**” kommer att fungera. 1p

2. Skriv (pseudo)kod till functionen ”**FE_RunTest(testtype ftest)**” på så sätt att (pseudo)koden är ganska generell – dvs att (pseudo)koden kan hantera flera fall än dem som beskrivs ovan.
   Koden ska följa principerna för abstrakt programmering.

   2p

(b) En annan form av abstraktion är **implementeringsabstraktion** där man försöka att gömma implementationsdetaljerna såsom arrayer eller strukturer och pekare så mycket som man kan så att resten av implementationen **följer principerna för abstrakt programmering**. Beskriv hur denna process fungerar. Vilka funktioner använder man sig av för att gömma implementationsdetaljerna?

2p

Totalt 5p

(4) Heap algoritmer

(a) Titta på koden nedan för **Build & Heapify**. Tillämpa koden på denna sekvens:

1, 2, 3, 4, 7, 8, 9, 10, 14, 16

Visa varje steg i Dina beräkningar inklusive de rekursiva anropen.
Förklara hur koden fungerar.

3p

```
Heapify(A, i)
   l = Left(i)
   r = Right(i)
   if l <= A.size and A[l] > A[i] then largest = l
   else largest = i
   if r <= A.size and A[r] > A[largest] then largest = r
   if largest != i then
      swap(A[i], A[largest])
      Heapify(A, largest)
      end if
   end Heapify

Build(A)
   for i = [A.size / 2] downto 1 do Heapify(A, i)
   end Build
```

(b) Titta på koden nedan för **Add**. Förklara **principen** bakom koden. Visa hur koden fungerar genom att lägga till (add) 42 till resultatet från (a) ovan.

2p.

```
Add(H, v)
   let A = H.array
   A.size++
   i = A.size
   while i > 1 and A[Parent(i)] < v do
      A[i] = A[Parent(i)]
      i = Parent(i)
      end while
   A[i] = v
   end Add
```

Totalt 5p

(5) Hashning

I hashning har fyra metoder för kollisionshantering presenterats, nämligen

1) Separate chaining (open hashing)
2) Open addressing (closed hashing) f(i) = i (linear probing)
3) Quadratic probing f(i) = i * i
4) Double hashing f(i) = i * H_2(key)

där H_2(key) = 7 – (key mod 7)

Metoderna (2), (3) och (4) hanterar kollisioner med H(key) + f(i) där 'i' representerar antalet kollisioner. Anta att **H(key) = key mod 10**. **Hash space = array H[10].**

Tillämpa dessa 4 metoder på sekvensen **4, 36, 44, 5, 7, 64, 24** och **visa varje steg** i Dina beräkningar.

3p

Vilka problem kan förekomma med dessa metoder?

2p

Totalt 5p

(6) Träd

I hantering av **binära sökträd** är operation "**ta bort**" (remove) den svåraste operationen att implementera. Vilka aspekter måste man ta hänsyn till när man implementerar denna operation?
Skriv (pseudo)kod till en **rekursiv implementation** av ta bort från ett binärt sökträd.
Du får använda "**hjälpfuktioner**" i Din (pseudo)kod.

5p

(7) Graf algoritmer

(a) Tillämpa **Dijkstra_SPT algoritm** på den oriktade grafen nedan.

- **Visa varje steg** i Dina beräkningar.
- **Rita en bild av grafen och ge kostnadsmatrisen**.
- **Börja med nod a**.
- **Rita en bild av resultatet**.

(a-6-b), (a-1-c), (a-5-d), (b-5-c), (b-3-e), (c-5-d), (c-6-e), (c-4-f), (d-2-f), (e-6-f)

```
Dijkstra_SPT ( a ) {
   S = {a}
   for (i in V-S) {
      D[i] = C[a, i]
      E[i] = a
      L[i] = C[a, i]
      }
   for (i in 1..(|V|-1)) {
      choose w in V-S such that D[w] is a minimum
      S = S + {w}
      foreach ( v in V-S ) if (D[w] + C[w,v] < D[v]) {
         D[v] = D[w] + C[w,v]
         E[v] = w
         L[v] = C[w,v]
      }
   }
}
```

3p

(b) Tillämpa **Prims algoritm** på den oriktade grafen nedan.

- **Visa varje steg** i Dina beräkningar.
- **Rita en bild av grafen och ge kostnadsmatrisen**.
- **Börja med nod a**.
- **Rita en bild av resultatet**.

(a-6-b), (a-1-c), (a-5-d), (b-5-c), (b-3-e), (c-5-d), (c-6-e), (c-4-f), (d-2-f), (e-6-f)

```
Prim's Algoritm
-- antagande: att det finns en kostnadsmatrix C

Prim (node v)                    //v is the start node
{
   U = {v};
   for i in (V-U) { low-cost[i] = C[v,i]; closest[i] = v; }
   while (!is_empty (V-U) ) {
      i = first(V-U); min = low-cost[i]; k = i;
      for j in (V-U-k) if (low-cost[j] < min) {
         min = low-cost[j]; k = j;
         }
      display(k, closest[k]);
      U = U + k;
      for j in (V-U)
       if ( C[k,j] < low-cost[j] ) {
          low-cost[j] = C[k,j];
          closest[j] = k;
          }
       }
}
```

3p

(c) **Förklara ingående** principen bakom Prims algoritm.

OBS: Skriv inte en rad för rad "**översättning**" från koden till svenska (engelska) utom ge **en beskrivning** (dvs en tolkning) av hur algoritmen fungerar vid varje steg. Använd gärna bilder i Din beskrivning.

Sedan ge en sammanfattning av principen i två eller tre meningar.

4p

Totalt 10p