OMTENTAMEN I
DATASTRUKTURER OCH ALGORITMER DVG B03

140818 kl. 08:15 – 13:15

Ansvarig Lärare: Donald F. Ross

Hjälpmedel: Inga. Algoritmerna finns i de respektive uppgifterna.

***** OBS *****

Betygsgräns:

Kurs:	Max 60p, Med beröm godkänd 50p, Icke utan beröm godkänd 40p, Godkänd 30p **(varav minimum 15p från tentamen, 15p från labbarna)**
Tentamen:	Max 30p, betyg 5: 26p-30p, betyg 4: 21p-25p, betyg 3: 15p-20p
Labbarna:	Max 30p, betyg 5: 26p-30p, betyg 4: 21p-25p, betyg 3: 15p-20p

SKRIV TYDLIGT – LÄS UPPGIFTERNA NOGGRANT

Ange alla antaganden.

---

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

(a) Vad är ”big-O” för en funktion som skriver ut en adjacency matrix? Varför?
(b) Vad gör Dijkstras algorithm?
(c) Vad gör Floyds algorithm?
(d) Vad gör Warshalls algorithm?
(e) Vad gör Topologisk sortering?
(f) Vad är en heap?
(g) Vad är fördelen med hashning?
(h) Vad är en rekursiv funktion?
(i) Vad är ett AVL-träd?
(j) Vad är dubbel hashning?

Totalt 5p

(2) Heap

Diskutera ingående hur koden till heap operationer (**se Bilaga A**) fungerar? Använd sekvensen 46, 13, 18, 33, 72, 9, 11, 44, 27, 15, 66 som ett exempel. Anta att det största värdet hamnar i roten.

5p

(3) Rekursion

Skriv (pseudo)kod till en **rekursiv funktion** (eller **två rekursiva funktioner**) för att räkna fram antalet kanter (edges) i en graf. Ange alla antagande.

5p

(4) Grafoperationer

En graf kan beskrivas som en virtuell (eller abstrakt) maskin. I graflaborationen, filen ”begraph.h” är faktiskt en beskrivning av denna virtuell maskin. Innehållet ges nedan.

```
/************************************************************************/
/* function prototypes - operations on the Graph (a virtual machine)    */
/************************************************************************/
/* Graph = (V, E) where V is a set of vertices/nodes and E a set of edges */
/* There are a limited number of operations (9) which can be applied to G */
/************************************************************************/
void be_display_adjlist();            /* display G as an adjacency list  */
void be_display_adjmatrix();          /* display G as an adjacency matrix*/

void be_addnode(char c);              /* add a vertex (node) to G        */
void be_remnode(char c);              /* remove a vertex (node) from G   */

void be_addedge(char cs, char cd, int v);/* add an edge (with weight) to G*/
void be_remedge(char cs, char cd);    /* remove an edge from G           */

int  be_is_nmember(char c);           /* is a  node a member of G?       */
int  be_is_emember(char cs, char cd); /* is an edge a member of G?       */

int  be_size();                       /* the number of nodes in G        */
/************************************************************************/
```

I **front-end:en** kan man specifiera förvillkor (preconditions) till varje operation ovan med hjälp av en eller flera av dessa 9 funktioner. Skriv (pseudo)kod till föjlande front-end-funktioner på så sätt att änvändaren inte behöver mäta in mer information än absolut nödvändigt. Anta att det finns ett visst antal användargränssnittfunktioner som kan ta emot information samt skicka tillbaka informationsmeddelande.

(i)	fe_display_adjlist()	// display adjacency list	1p
(ii)	fe_remnode()	// remove node	1p
(iii)	fe_remedge()	// remove edge	3p

Totalt 5p

(5) Labbkod

(a) I graflabben har en student skrivit följande kod för att ta bort en kant (edge) från an adjacency lista. Förklara **ingående** hur koden fungerar. Använd gärna exempel. **Ange alla antagande.**
Vilka är förutsättningarna för att koden ska fungera?

```
void reme(char cs, char cd) {
  set_edges(b_findn(cs, G), b_reme(cd, get_edges(b_findn(cs, G))));
  }
```

2p

(b) I trädlabben har en student skrivit kod för att söka efter ett värde i ett BST (binärt sökträd). Sedan har studenten kommit på att denna kod kunde lätt anpassas för att söka efter ett värde i ett komplett träd. Dessa funktioner finns nedan. Vad har studenten skrivit för ”**xxx**” och ”**yyy**”? **Ange alla antagande.**

```
static int b_findb(treeref T, int v)
{
  return is_empty(T) ? 0
        : v <  get_value(node(T)) ? b_findb(LC(T), v)
        : v >  get_value(node(T)) ? b_findb(RC(T), v)
        : 1;
}

static int b_findc(treeref T, int v)
{
  return is_empty(T) ? 0
        : xxx ? 1
        : yyy;
}
```

1p

(c) Skriv (pseudo)kod för att lägga till ett element i ett binärt träd.
Ange alla antagande. **2p**

Totalt 5p

(6) Dijkstra + SPT (Shortest Path Tree)

Tillämpa **den givna Dijkstra_SPT algoritmen (nedan)** på **den riktade grafen**,

(a, b, 12), (a, d, 11), (a, e, 9), (b, c, 7), (c, e, 5), (d, c, 3), (d, e, 1)

SPT = Shortest Path Tree - dvs kortaste väg trädet (KVT) från en nod till alla de andra.

Börja med nod ”a”.
Visa varje steg i dina beräkningar.
Ange *alla* antaganden och visa *alla* beräkningar och mellanresultat
Rita **varje steg** i konstruktionen av SPT:et – dvs visa till och med de noder och kanter som läggs till men sedan tas bort.

(3p)

Förklara **principerna** bakom **Dijkstras_SPT** algoritm.

(2p)

Totalt 5p

Dijkstras algoritm med en utökning för SPT

```
Dijkstra_SPT ( a )
{
    S = {a}

        for (i in V-S) {
          D[i]   = C[a, i]              --- initialise D - (edge cost)
          E[i]   = a                    --- initialise E - SPT (edge)
          L[i]   = C[a, i]              --- initialise L - SPT (cost)
        }

        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]
                 }
        }
}
```

Bilaga A

Heap Algoritmer

```
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
```

```
Remove (H, r)
    let  A = H.array
    A[r] = A[A.size]
    A.size--
    Heapify(A, r)
     end Remove
```

```
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
```