FACIT TILL
OMTENTAMEN I
DATASTRUKTURER OCH ALGORITMER DVG B03

150817 kl. 08:15 – 13:15

Ansvarig Lärare: Donald F. Ross

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

***** 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) Vad är ”big-O” för Dijkstras algoritm? **$O(n^2)$**

(b) Vad är ”big-O” för Floyds algoritm? **$O(n^3)$**

(c) Vad är ”big-O” för lägga-till-operationen i hashning? **O(1)**

(d) Vad gör Warshalls algoritm?

Calculates the transitive closure of a graph. Det transitive höljden.

(e) Vad gör en postorder traversering av ett binärt träd?

Visits the nodes in the order LRN and maps the tree to a sequence.

(f) Vad är en graf?

G = (V, E) where G is a set of nodes and E a set of edges (a,b)
where a, b are members of V

(g) Vad representerar kanterna i en graf?

A relationship between two nodes. E.g. distance, cost, is reachable from.

(h) Vad är ett ”Free Tree”?

An undirected graph with n nodes and (n-1) edges.

(i) Vad händer när man lägger till en kant till i ett ”Free Tree”?

A cycle is created.

(j) Vad är kvadratisk probning (quadratic probing)?

In collision management in hashing, for each collision i a function is defined to calculate a new index value for the key. This can be stated as h(key) + f(i).
In quadratic probing $f(i) = i^2$

Totalt 5p

(2) Ge ett kortfattat svar till följande uppgifter ((a)-(e)).

(a) Under vilka förutsättningar skulle SPT:et (Shortest Path Tree) och MST:et (Minimal Spanning Tree) vara identiska när man tillämpar Dijkstra-SPT och Prims algoritmer på samma oriktade graf?

(i) All the edges in the graph have the same cost value
(ii) The nodes are stored in the adjacency list in alphabetical order

(b) Skriv rekursiv pseudokod till en funktion för att hitta det maximala värdet i ett vänsterbarn i ett BST (Binärt SökTräd).

```
static int find_max(treeref T) {
return  is_empty(RC(T)) ? get_value(T) : find_max(RC(T));
}
```

(c) Ge en rekursiv definition av ett BT (Binärt Träd).

BT ::= LC N RC | empty
N ::= element
LC ::= BT
RC ::= BT

(d) Vilka begränsningar måste man specificera för att förvandla ett binärt träd (BT) till ett binärt sökträd (BST) och ett BST till ett AVL-träd?

(i) BT ➔ BST: all values in the left child must be less than the value at the root and all values in the right child must be greater than the value at the root. One of these conditions may be changed to include "or equal to"
(ii) BST ➔ AVL-träd | height(LC(T)) – height(RC(T)) | < 2

(e) Förklara vad en ADT (abstrakt datatyp) är.

(i) A set of values plus operations on those values.
(ii) The ADT is implementation independent.

Totalt 5p

(3) Sekvens

(a) Ange en rekursiv definition av en sekvens.

(1p)

S ::= H T | ¤ **H = head, T = tail, ¤ = empty**
H ::= element
T ::= S

(b) Utifrån din definition i (a) ovan, skriv rekursiv pseudokod för att räkna antalet element i en sekvens.
Ange alla antaganden.
Ange ett exempel för att visa hur din kod fungerar.

(2p)

```
int card(SequenceRef  S) {
  return is_empty(S) : 0 ? 1 + card(tail(S));
  }
```

==========

```
int card(SequenceRef  S) {
  if ( is_empty(S)) return  0  else return  1 + card(tail(S));
  }
```

(c) Utifrån din definition i (a) ovan, skriv rekursiv pseudokod för att lägga till ett element i en skevens i stigande (sorterad) ordning. Anta att det redan finns en funktion som heter Cons som lägger till ett element i huvudet av sekvensen.
Cons: element x sekvens ➔ sekvens.
Ange alla andra antaganden.
Ange ett exempel för att visa hur din kod fungerar.

(2p)

Totalt 5

```
static SequenceRef add_val(SequenceRef S, ValType v)
{
  if is_empty(S)                  return create_e(v);
  if v < getval(head(S))          return cons(create_e(v), S);
                                  return cons(head(S),  add_val(tail(S) ,v));
}
```

==========

```
static SequenceRef add_val(SequenceRef S, ValType v)
{
 return is_empty(S)             ?   create_e(v)
      :   v < get_value(head(S)) ?   cons(create_e(v), S)
      :                              cons(head(S), add_val(tail(S) ,v));
}
```

(4) Rekursion

Skriv pseudokod till **två rekursiva funktioner** (alltså en funktion plus en hjälpfunktion) för att räkna fram antalet kanter (”edges”) i en graf. Ange alla antagande.

5p

Assumptions:

Structure:

```
typedef struct nodeelem * noderef;

typedef struct nodeelem {
      char        nname;
      int         ninfo;
      noderef   edges;
      noderef   nodes;
      } nodeelem;
```

+ corresponding get/set functions per attribute and head/tail operations for both the node list (nodes) and the edge list (edges).

G is a reference to the graph, the is_empty(R) function is defined.

```
static int b_nedges(noderef E) {
  return is_empty(E) ? 0 : 1 + b_nedges(etail(E));
  }

static int b_esize(noderef G) {
  return is_empty(G) ? 0 : b_nedges(get_edges(nhead(G))) + b_esize(ntail(G));
  }
```

(5) Topologisk sortering

Vid ett universitet har vissa kurser förkunskapskrav. I datavetenskap kräver kompilatorkonstruktion (DAV D02) programspråk (DAV C02) som förkunskap. Datastrukturer och algoritmer (DAV B03) är ett förkunskapskrav till programspråk, avancerad programmering i C++ (DAV C05), samt projektarbete i Java (DAV C08). Datastrukturer och algoritmer kräver diskret matematik (MAA B06) samt programutvecklingsmetodik (DAV A02). Operativsystem (DAV B01) kräver i sin tur programutvecklingsmetodik och datorsystemteknik (DAV A14) och är förkunskapskrav till C och UNIX (DAV C18), tillämpad datasäkerhet (DAV C17) samt realtidssystem (DAV C01). Objektorienterade designmetoder (DAV D11) kräver bägge avancerad programmering i C++ och software engineering (DAV C19).

Hur kan man **visa** att kompilatorkonstruktion och objektorienterade designmetoder kräver diskret matematik?

I vilken ordning ska en student som vill läsa på D-nivå ta alla de ovannämnda kurserna? Metoden som kan användas för att komma fram till en lösning heter topologisk sortering. En variant av topologisk sortering är följande algoritm:

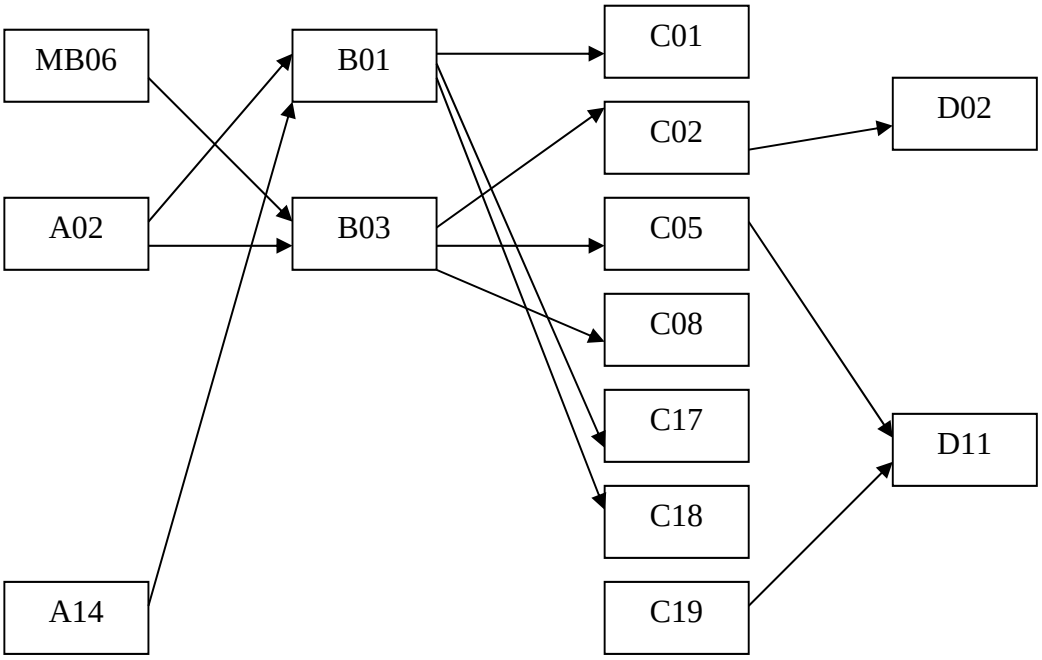
Topological Sort

```
tsort(v) -- prints reverse topological order of a DAG from v
{       mark v visited
        for each w adjacent to v if w unvisited tsort(w)
        display(v)
        }
```

Vilken är den andra varianten? Tillämpa både varianter i din lösning till problemet ovan.

Vilka begränsningar måste man ta hänsyn till? **DAG.** **5p**

Step 1: Draw the graph



Step 2: Construct the adjacency matrix

	MB06	A02	A14	B01	B03	C01	C02	C05	C08	C17	C18	C19	D02	D11
MB06					1									
A02				1	1									
A14				1										
B01						1				1	1			
B03							1	1	1					
C01														
C02													1	
C05														1
C08														
C17														
C18														1
C19														
D02														
D11														

Step 3: Apply Warshall's by inspection (you do not need to show every step)

	MB06	A02	A14	B01	B03	C01	C02	C05	C08	C17	C18	C19	D02	D11
MB06					1		1	1	1				1	1
A02				1	1	1	1	1	1	1	1		1	1
A14				1		1				1	1			
B01						1				1	1			
B03							1	1	1				1	1
C01														
C02													1	
C05														1
C08														
C17														
C18														
C19														1
D02														
D11														

To show that OODM (D11) requires Discrete Mathematics (MB06) look at the row MB06 in the **transitive closure** to find that D11 is reachable from MB06 i.e. there is a **path** from MB06 to D11.

Note also that the diagonal shows that there are no cycles in the graph. Hence the graph is a DAG.

Course order – method 1: Use the in-degree for each node.

Construct a list with the in-degree.

((MB06, 0), (A02, 0), (A14, 0), (B01, 2), (B03, 2), (C01, 1), (C02, 1), (C05, 1), (C08, 1), (C17, 1), (C18, 1), (C19, 0), (D02, 1), (D11, 2))

Remove the nodes with in-degree 0 **and** their edges and readjust the list. Add these nodes to the output list.

Output: (MB06, A02, A14, C19)

((B01, 0), (B03, 0), (C01, 1), (C02, 1), (C05, 1), (C08, 1), (C17, 1), (C18, 1), (D02, 1), (D11, 1))

Repeat the process until the degree list is empty.

Output: (MB06, A02, A14, C19, B01, B03)

((C01, 0), (C02, 0), (C05, 0), (C08, 0), (C17, 0), (C18, 0), (D02, 1), (D11, 1))

Output: (MB06, A02, A14, C19, B01, B03, C01, C02, C05, C08, C17, C18)

((D02, 0), (D11, 0))

Output: **(MB06, A02, A14, C19, B01, B03, C01, C02, C05, C08, C17, C18, D02, D11)**

( ) – degree list is empty – finished.

Course order – method 2: perform a depth-first search on the graph

Construct the adjacency list:

MB06: B03
A02: B01, B03
A14: B01
B01: C01, C17, C18
B03: C02, C05, C08
C01:
C02: D02
C05: D11
C08:
C17:
C18:
C19: D11
D02:
D11:

Perform a topological sort:

Start with node MB06:

MB06
  B03
    C02
      D02
      Stop – output D02 **D02**
    Stop – output C02 **D02, C02**
    C05
      D11
      Stop – output D11 **D02, C02, D11**
    Stop – output C05 **D02, C02, D11, C05**
    C08
    Stop – output C08 **D02, C02, D11, C05, C08**
  Stop – output B03 **D02, C02, D11, C05, C08, B03**
Stop – output MB06 **D02, C02, D11, C05, C08, B03, MB06**

A02
  B01
    C01
    Stop – output C01 **D02, C02, D11, C05, C08, B03, MB06, C01**
    C17
    Stop – output C17 **D02, C02, D11, C05, C08, B03, MB06, C01, C17**
    C18
    Stop – output C17 **D02, C02, D11, C05, C08, B03, MB06, C01, C17, C18**
  Stop – output B01 **D02, C02, D11, C05, C08, B03, MB06, C01, C17, C18, B01**
Stop – output A02 **D02, C02, D11, C05, C08, B03, MB06, C01, C17, C18, B01, A02**

A14
Stop – output A14 **D02, C02, D11, C05, C08, B03, MB06, C01, C17, C18, B01, A02, A14**

C19
Stop – output C19 **D02,C02, D11,C05,C08, B03, MB06,C01,C17,C18, B01, A02,A14,C19**

Now **reverse** the list

C19, A14, A02, B01, C18, C17, C01, MB06, B03, C08, C05, D11, C02, D02

The restriction is that **the graph must be a DAG** in order to perform a topological sort. This was shown above using Warshall's.

(6) AVL-Träd

Visa hur Du skulle ta fram de 4 rotationsfunktionerna från första principer.

2p

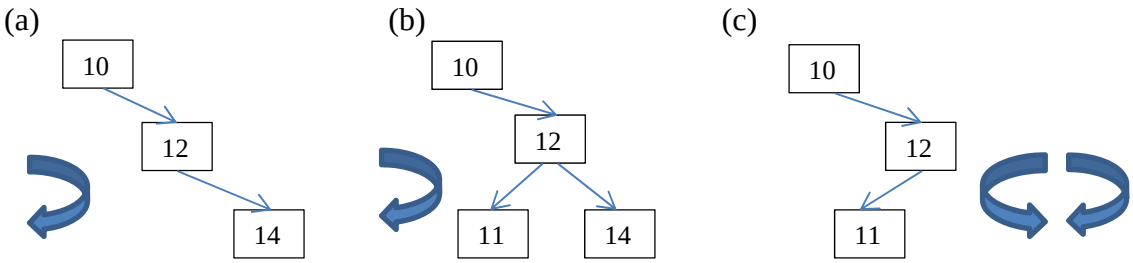
An AVL-tree is a BST (Binary Search Tree) where the balance difference (balance factor) between the height of the left child and the height of the right child may be at most 1.

i.e. | height(LC) – height(RC) | <= 1

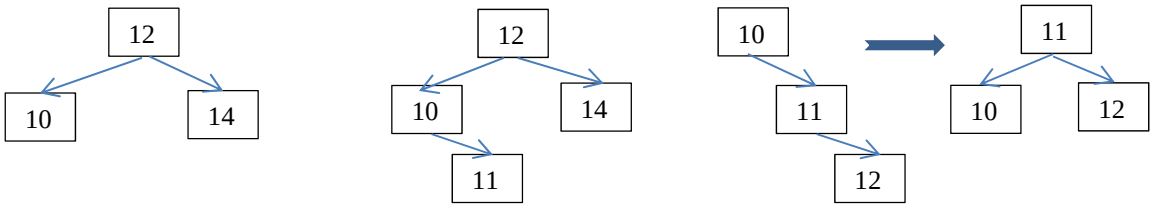
The rotation functions are

- SLR (Single Left Rotation)
- SRR (Single Right Rotation)
- DLR (Double Left Rotation) or Right-Left Rotation
- DRR (Double Right Rotation) or Left-Right Rotation
-

The rotation functions may be derived using a few simple examples.



(a) and (b) require a SLR to rebalance while (c) requires a DLR (right-left rotation) to give



And the code becomes (SRR and DRR are mirror images of SLR and DLR respectively)

```
static treeref SLR(treeref T) {
  treeref  RT = RC(T); set_RC(T, LC(RT)); set_LC(RT, T); return RT;
  }

static treeref SRR(treeref T) {
  treeref  RT = LC(T); set_LC(T, RC(RT)); set_RC(RT, T); return RT;
  }

static treeref DLR(treeref T) {   set_RC(T, SRR(RC(T)));  return SLR(T); }
static treeref DRR(treeref T) {   set_LC(T, SLR(LC(T)));  return SRR(T); }
```

This is another way of writing what was in the notes

```
Treeref RotateLeft(n2)          Treeref RotateRight(n2)
n1        = n2.right            n1        = n2.left
n2.right = n1.left              n2.left   = n1.right
n1.left  = n2                   n1.right = n2
return n1                       return n1
end RotateLeft                  end RotateRight
```

Is implemented as (using the set and get functions)

/* RotateLeft */ /* n2 = T and n1 = RT */

```
static treeref SLR(treeref T) {
  treeref  RT = RC(T); set_RC(T, LC(RT)); set_LC(RT, T); return RT;
  }
```

/* RotateRight */

```
static treeref SRR(treeref T) {
  treeref  RT = LC(T); set_LC(T, RC(RT)); set_RC(RT, T); return RT;
  }
```

Förklara vad ”balansfaktorn” (balance factor) är och använd denna för att ta fram balansfunktionen från första principer. **Skriv balansfunktionen i pseudokod.**

Define the balance factor bf = height(LC(T) – height(RC(T))

Where the height(T) is defined as 1 + max (height(LC(T)), height(RC(T)))
An enpty tree has height 0

Now you can decide which sub-tree is the highest.
This decides whether the rotation is left or right.

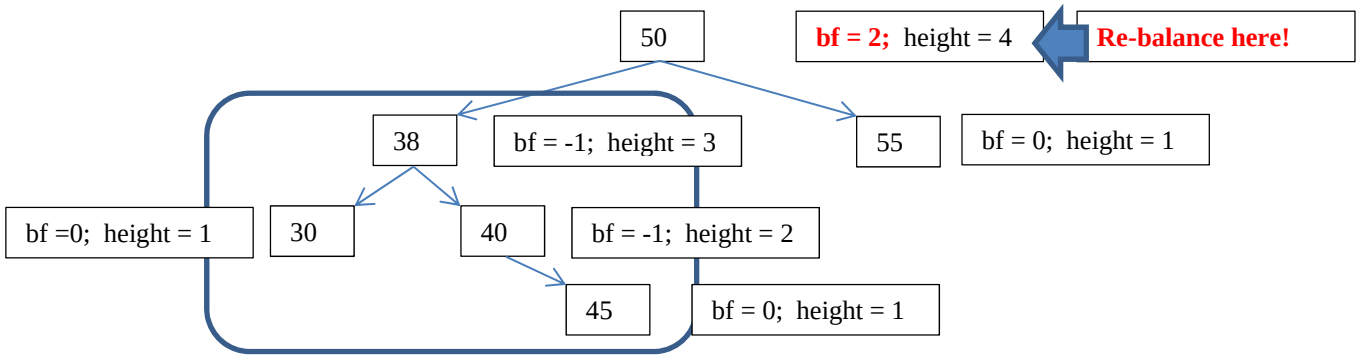
Then look at that sub-tree to decide if the imbalance is on the inside (➔ double rotation) or outside (➔ single rotation).

Then you are required to write pseudo-code to re-balance the tree

NB this part of the question. The answer is NOT supplied here. This is the point of the question!

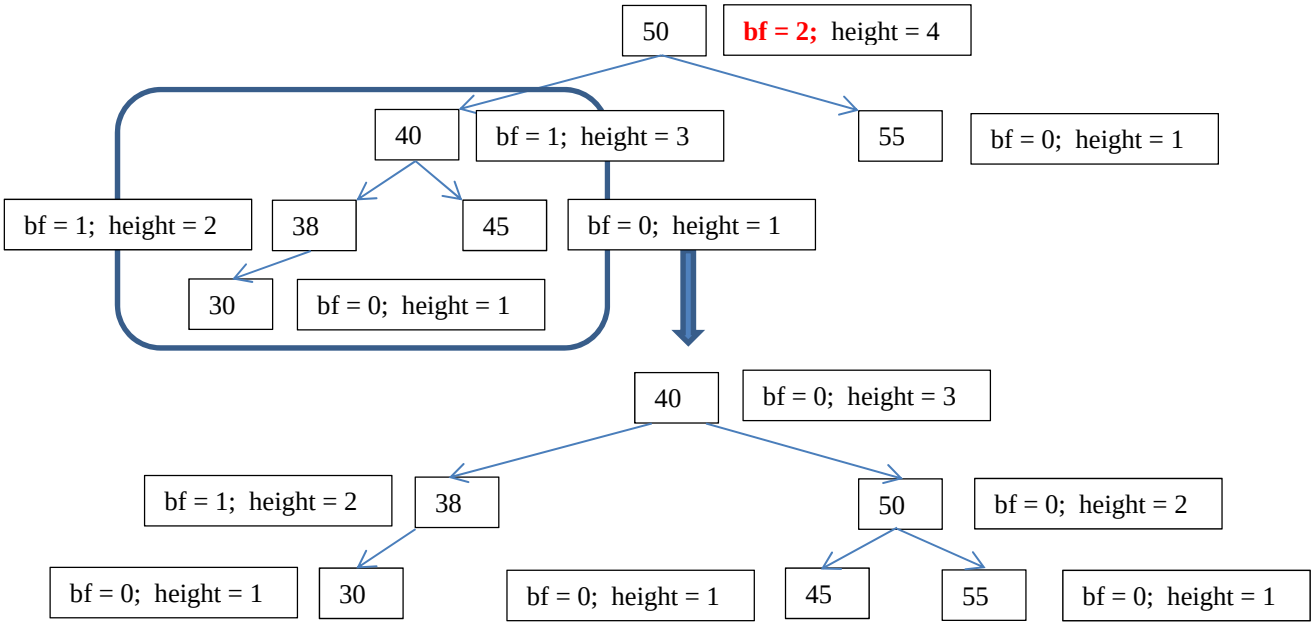
Tillämpa Din pseudokod på trädet nedan

3p



From the diagram, the left child is clearly higher than the right child – **how do you show this?** The right child of the left child is higher that the left child of the left child indicating a possible addition of 45 i.e. to the INSIDE of the left child of 50 hence a DRR is required.
How do you show this using the balance factor?
NB delete 60 from the above tree + 60 would give the same requirement.

DRR = SLR (38) then a SRR(50) to give




```
static treeref DRR(treeref T) {   set_LC(T, SLR(LC(T)));  return SRR(T); }
static treeref SLR(treeref T) { treeref  RT = RC(T); set_RC(T, LC(RT)); set_LC(RT, T); return RT; }
static treeref SRR(treeref T) { treeref  RT = LC(T); set_LC(T, RC(RT)); set_RC(RT, T); return RT; }
```

SLR(T=38) ➔ RT = 40; RC(38) = LC(40) (null); LC(40) = 38; return 40;

SRR(T=50) ➔ RT = 40; LC(50) = 45; RC(40) = 50; return 40;

(7) Prims algoritm

Tillämpa **den givna Prims algoritm nedan** på **den oriktade grafen**,

(a-20-b, a-36-c, a-34-d, b-22-c, b-24-e, c-28-d, c-30-e, c-38-f, d-26-f, e-36-f).

1. **Börja med nod ”a”**
2. **Visa varje steg i beräkningarna**
3. **Ange *alla* antaganden och visa *alla* beräkningar och mellanresultat!**

(3p)

```
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; }
        }
    }
```

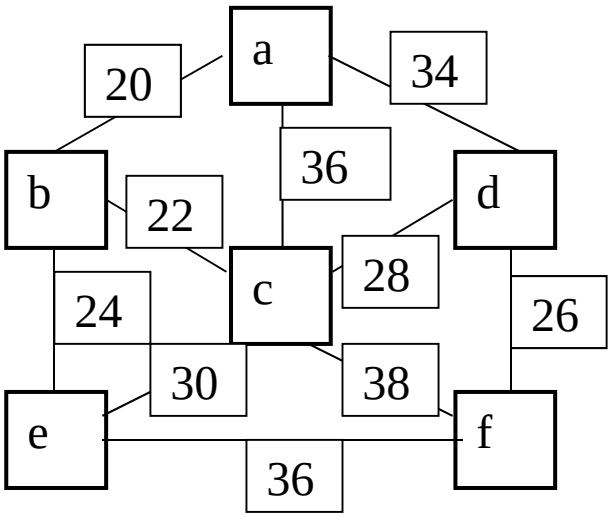
Totalt 5p

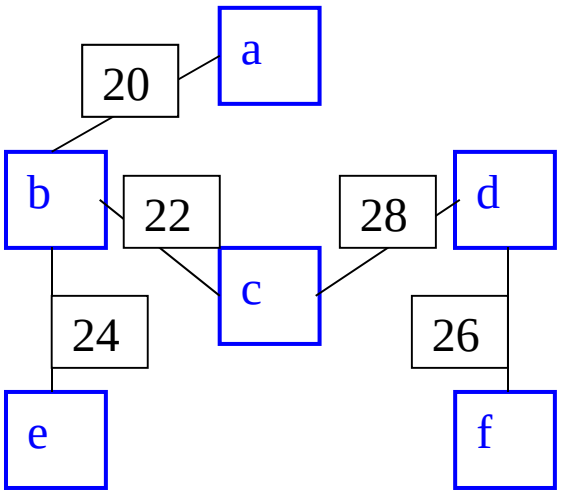
The principle is that the MST ” grows” from the **one component** (here ”a”) by connecting this component to any other component (a node) by the shortest edge and then this component “grows” to form the MST – this last proviso reveals that Prim’s is a GREEDY algorithm i.e. used a local best solution.

See below for the calculations.

Draw the graph (and possibly sketch the answer – use Kruskal’s for a quick check!):

Cost 120





Draw the cost matrix C and array D

	a	b	c	d	e	f
a		**20**	**36**	**34**		
b	**20**		**22**		**24**	
c	**36**	**22**		**28**	**30**	**38**
d	**34**		**28**			**26**
e		**24**	**30**			**36**
f			**38**	**26**	**36**	

a	**b**	**c**	**d**	**e**	**f**
lowcost	**20**	**36**	**34**	**§**	**§**
closest	**a**	**a**	**a**	**a**	**a**

Minedge**: lowcost: 20 36 34 § § closest: a a a a a** U = {a,b} V-U = {c,d,e,f} min = **20**; k = **b**
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = c; if C[b,c] < lowcost[c] then { lowcost[c] = C[b,c]; closest[b] = b } → 22<36 ➔ **c-22-b**
j = d; if C[b,d] < lowcost[d] then { lowcost[d] = C[b,d]; closest[d] = b } → §<34 ➔ no change
j = e; if C[b,e] < lowcost[e] then { lowcost[e] = C[b,e]; closest[e] = b } → 24<§ ➔ **b-24-e**
j = f; if C[b,f] < lowcost[f] then { lowcost[f] = C[b,f]; closest[f] = b } → §<§ ➔ no change

Minedge**: lowcost: 20 22 34 24 § closest: a b a b a** U = {a,b,c} V-U = {d,e,f} min = **22**; k = **c**
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = d; if C[c,d] < lowcost[d] then { lowcost[d] = C[c,d]; closest[d] = c } → 28<34 → **c-28-d**
j = e; if C[c,e] < lowcost[e] then { lowcost[e] = C[c,e]; closest[e] = c} → 30<24 → no change
j = f; if C[c,f] < lowcost[f] then { lowcost[f] = C[c,f]; closest[f] = c } → 38<§ → **c-38-f**

Minedge**: lowcost: 20 22 28 24 38 closest: a b c b c** U = {a,b,c,e} V-U = {d,f} min = **24**; k = **e**
j = d; if C[e,d] < lowcost[d] then { lowcost[d] = C[e,d]; closest[d] = e } → §<28 → no change
j = f; if C[e,f] < lowcost[f] then { lowcost[f] = C[e,f]; closest[f] = e } → 36<38 ➔ **e-36-f**

Minedge**: lowcost: 20 22 28 24 36 closest: a b c b e** U = {a,b,c,e,d} V-U = {f} min = **28**; k = **d**
j = f; if C[d,f] < lowcost[f] then { lowcost[f] = C[d,f]; closest[f] = d} → 26<36 → **d-26-f**

Min edge**: lowcost: 20 22 28 24 26 --- closest: a b c b d ---** U = {a,c,b,d,f e} V-U = {¤}

QED ☺ MST edges **a-20-b, b-22-c, b-24-e, c-28-d, d-26-f Total cost = 120**

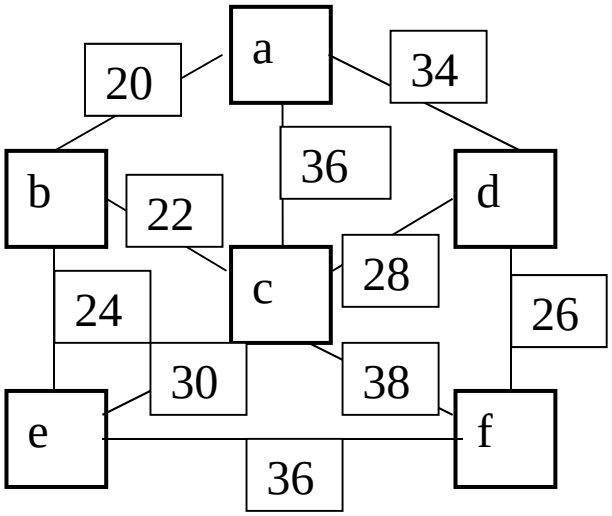
(Confirm using Kruskal's)

Förklara hur Prims algoritm fungerar genom att rita bilder som representerar varje mellanresultat under algoritmensexekvering. Använd exemplet ovan.
Vad är principen bakom Prims algoritm?

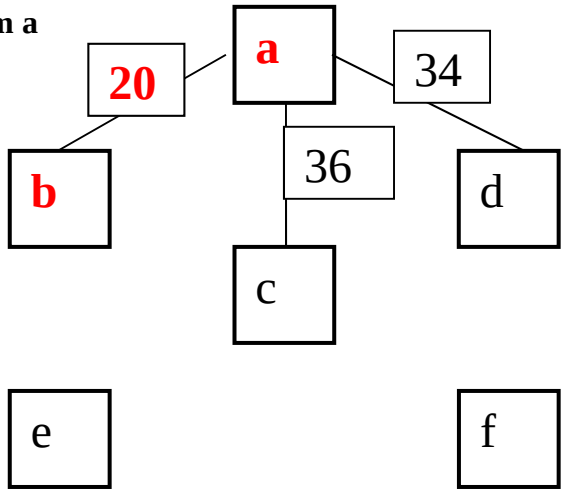
(2p)

See above for the principle.

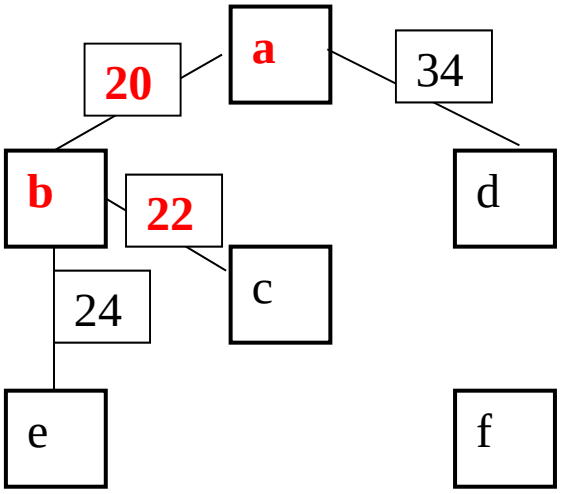
Diagrams:-



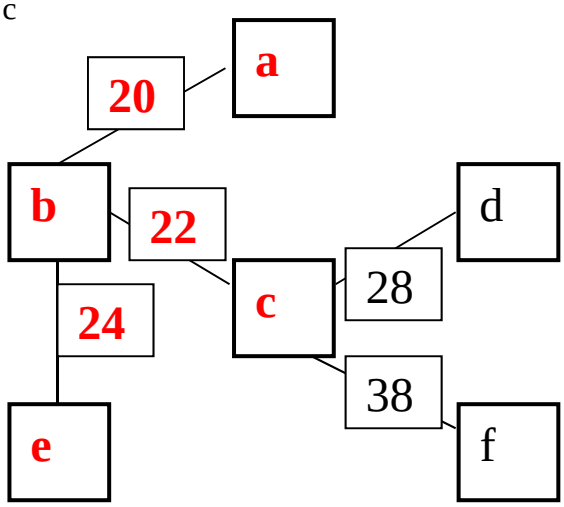
from a



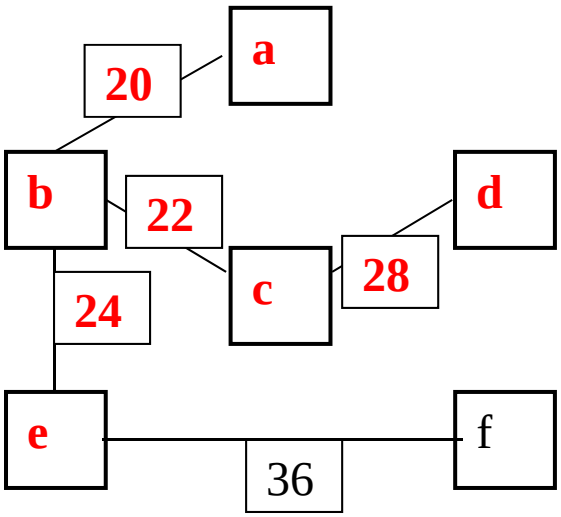
from b



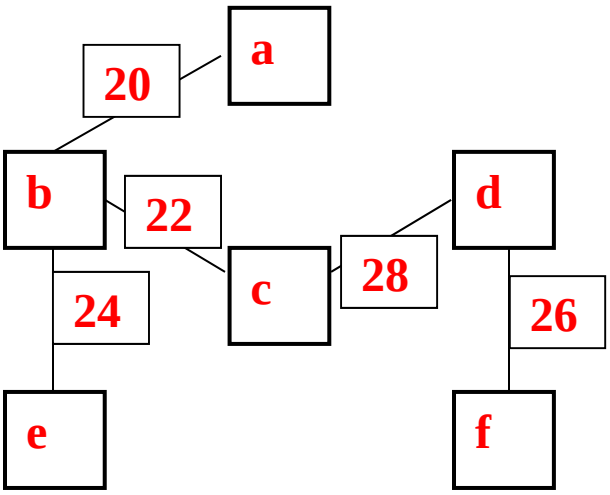
from c



from e



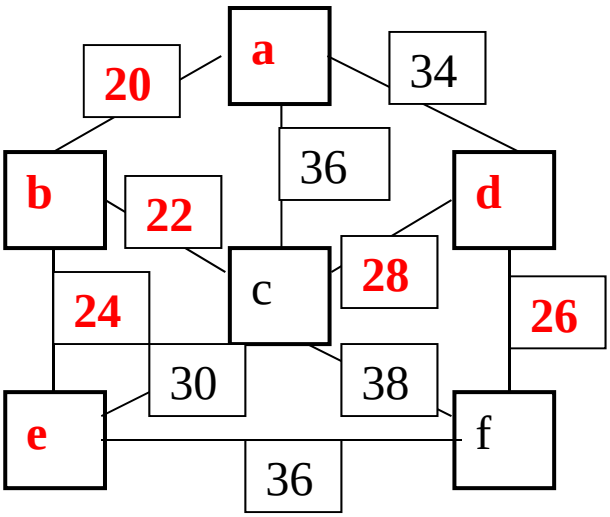
from d



Check this result using Kruskall's

a-20-b
b-22-c
b-24-e
d-26-f
c-28-d

the rest!



(8) Dijkstra + SPT (Shortest Path Tree)

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

(a-20-b, a-36-c, a-34-d, b-22-c, b-24-e, c-28-d, c-30-e, c-38-f, d-26-f, e-36-f).

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

1. **Börja med nod ”a”**
2. **Visa varje steg i beräkningarna**
3. **Ange *alla* antaganden och visa *alla* beräkningar och mellanresultat**
4. 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 hur Dijkstra-SPT algoritm fungerar genom att rita bilder som representerar varje mellanresultat under algoritmensexekvering. Använd exemplet ovan.

(2p)

Totalt 5p

Dijkstras algoritm med en utökning för SPT

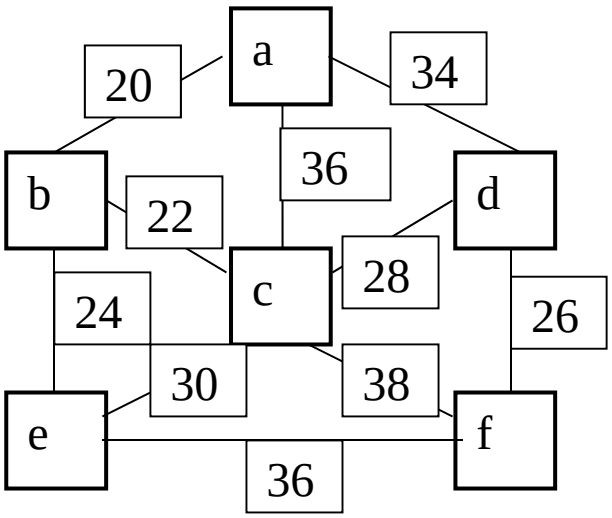
```
Dijkstra_SPT ( a )
{
      S = {a}

    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]
                                    }
    }
}
```

Draw the cost matrix C and arrays D, E, L

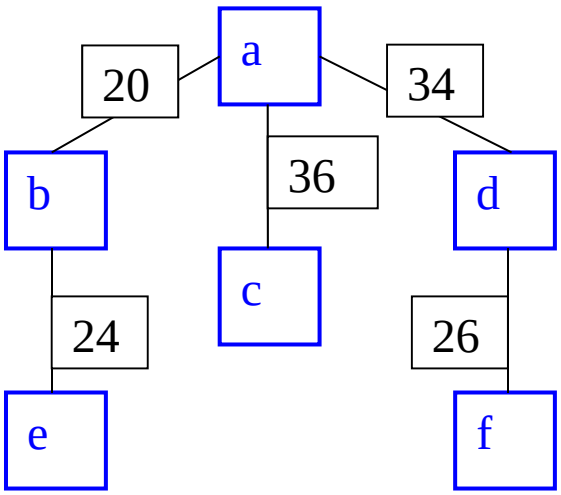
	a	b	c	d	e	f
a		**20**	**36**	**34**		
b	**20**		**22**		**24**	
c	**36**	**22**		**28**	**30**	**38**
d	**34**		**28**			**26**
e		**24**	**30**			**36**
f			**38**	**26**	**36**	

a	b	c	d	e	f
D	**20**	**36**	**34**	**§**	**§**
E	**a**	**a**	**a**	**a**	**a**
L	**20**	**36**	**34**	**§**	**§**

Draw the graph



SPT is



	a	b	c	d	e	f
D:	¤	**20**	**36**	**34**	**§**	**§**
E:	¤	**a**	**a**	**a**	**a**	**a**
L:	¤	**20**	**36**	**34**	**§**	**§**

Calculations:

w is b (min value in D) S = {a,b} V-S = {c,d,e,f}

v = c **if** (D[b]+C (b,c) < D[c]) ➔ **if** (20+22 < 36) → **no change**
v = d **if** (D[b]+C (b,d) < D[d]) ➔ **if** (20+§ < 34) → **no change**
v = e **if** (D[b]+C (b,e) < D[e]) ➔ **if** (20+24 < §) → **a-b-e 44**
(a-20-b)(b-24-e)
v = f **if** (D[b]+C (b,f) < D[f]) ➔ **if** (20+§ < §) → **no change**

	a	b	c	d	e	f
D:	¤	**20**	**36**	**34**	**44**	**§**
E:	¤	**a**	**a**	**a**	**b**	**a**
L:	¤	**20**	**36**	**34**	**24**	**§**

w is d (min value in D) S = {a,b,d} V-S = {c,e,f}

v = c **if** (D[d]+C (d,c) < D[c]) ➔ **if** (34+28 < 36) → **no change**
v = e **if** (D[d]+C (d,e) < D[e]) ➔ **if** (34+§ < 44) → **no change**
v = f **if** (D[d]+C (d,f) < D[f]) ➔ **if** (34+26 < §) → **a-d-f 60**
(a-34-d)(d-26-f)

	a	b	c	d	e	f
D:	¤	**20**	**36**	**34**	**44**	**60**
E:	¤	**a**	**a**	**a**	**b**	**d**
L:	¤	**20**	**36**	**34**	**24**	**26**

w is c (min value in D) S = {a,b,d,c} V-S = {e,f}

v = e **if** (D[c]+C (c,e) < D[e]) ➔ **if** (36+30 < 44) → **no change**
v = f **if** (D[c]+C (c,f) < D[f]) ➔ **if** (34+38 < 60) → **no change**

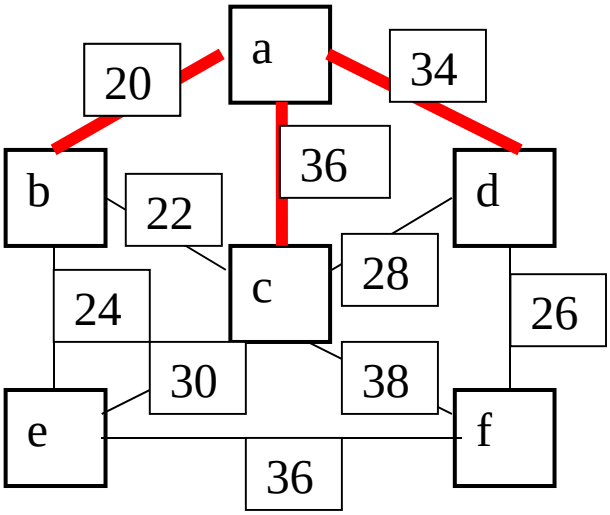
	a	b	c	d	e	f
D:	¤	**20**	**36**	**34**	**44**	**60**
E:	¤	**a**	**a**	**a**	**b**	**d**
L:	¤	**20**	**36**	**34**	**24**	**26**

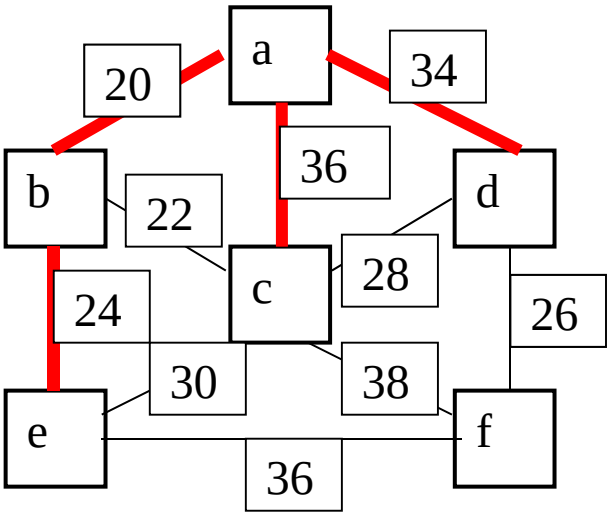
w is e (min value in D) S = {a,b,d,c,e} V-S = {f}

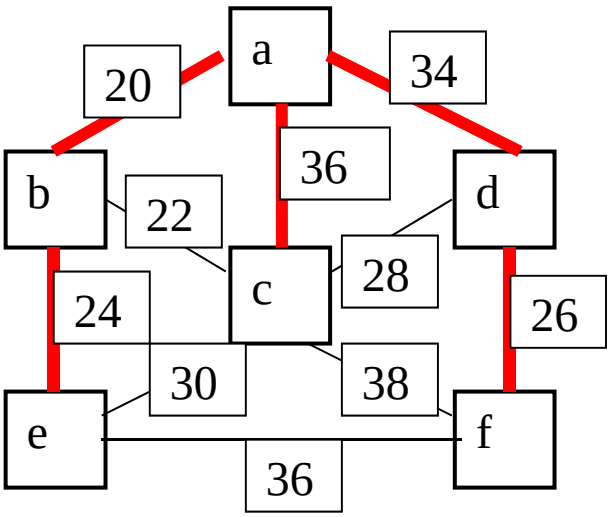
v = f **if** (D[e]+C (e,f) < D[f]) ➔ **if** (44+36 < 60) → **no change**

	a	b	c	d	e	f
D:	¤	**20**	**36**	**34**	**44**	**60**
E:	¤	**a**	**a**	**a**	**b**	**d**
L:	¤	**20**	**36**	**34**	**24**	**26**

w is f (min value in D) S = {a,b,d,c,e,f} V-S = {¤} – **STOP!**


a
20
34
36
b
22
d
c
28
24
26
30
38
e
f
36


a
20
34
36
b
22
d
c
28
24
26
30
38
e
f
36


a
20
34
36
b
22
d
c
28
24
26
30
38
e
f
36