FACIT TILL
TENTAMEN I
DATASTRUKTURER OCH ALGORITMER DVG B03

140114 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) En definition av en mängd
An unordered collection of unique entities having a common property (attribute).

(b) En definition av en sekvens
An ordered collection of entities (not necessarily unique) having a common property (attribute) with a successor relationship defined between entities.

(c) En definition av ett träd
An unordered/ordered collection of entities (not necessarily unique) having a common property (attribute) with a descendant relationship defined between entities.

(d) En definition av en graf
An unordered collection of entities (not necessarily unique) having a common property (attribute) with a (general) relationship defined between entities.

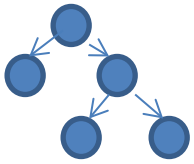
(e) En definition av ett AVL-träd
A binary search tree (BST) with the additional property that the height of the left and right sub-trees may not differ by more than 1.

Förklara Dina exempel ((f)-(j)) med några meningar

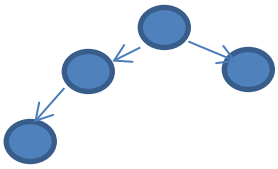
(f) Ett exempel av ett fullt träd som inte är komplett

(g) Ett exempel av ett komplett träd som inte är fullt

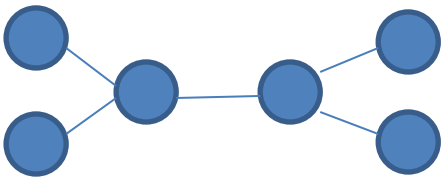
f



g



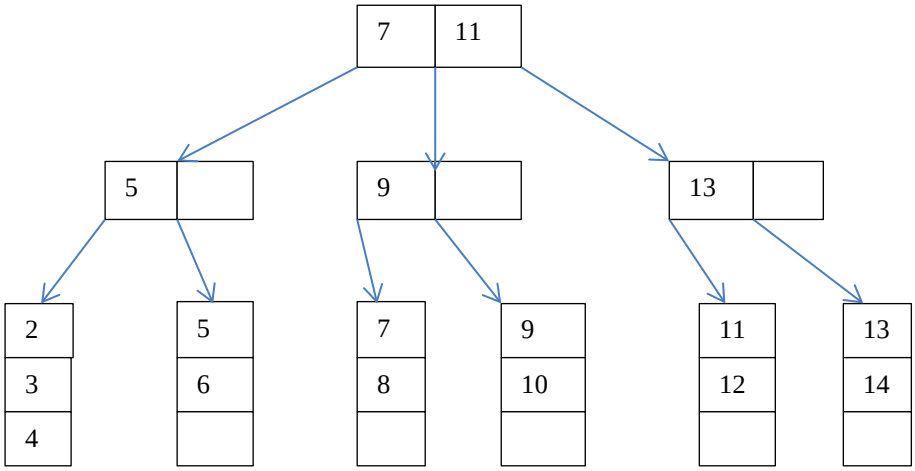
(h) Ett exempel av ett free tree – n nodes & (n-1) edges – example MST



(i) Ett exempel av en algoritm som är $O(n^3)$

Floyd's / Warshall's

(j) Ett exempel av ett B-tree



Totalt 5p

(2) Ge ett kortfattat svar till följande uppgifter.

(a) En definition av rekursion

An entity which is **PARTIALLY** defined in terms of itself – e.g. definition of a sequence
S ::= H T | ¤ ; H ::= element ; T ::= S

A function which **CONDITIONALLY** calls itself

(b) En definition av abstraktion

MODELLING ABSTRACTION
The process of selecting certain properties (attributes) of an entity to represent that entity in a given situation.

COLLECTION ABSTRACTION
The process of generalising common properties and operations of the set, sequence, tree and graph.

IMPLEMENTATION ABSTRACTION
The process of selecting certain properties of a Data Type independent of the implementation of that Data Type - hence the expression Abstract Data Type

(c) Principerna bakom hashning

A hash function hf maps a key value to an index in the hash space hf(key)➔index
Main problem is collision handling – a second function kf based on the number of collisions to give a new index hf(key) + kf(i) for the ith collision
kf(i) may be kf: i → i; kf: i → i^2; kf: i → i * hf_2(key)
an alternative is to chain the values giving rise to collisions in a linked list

(d) Ett exempel samt beskrivning av en girig algoritm – Dijkstra or Prim

A locally best solution is chosen and previously calculated values are readjusted to see if a better solution may be obtained via the last selected node (locally best)

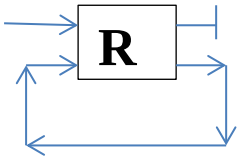
(e) En definition av Big-Oh – O(x)

An upper bound indicator (usually worst case time performance) for a given algorithm.

Totalt 5p

(3) Rekursion

(a) Beskriv vilket **mönster** man hittar hos rekursiva funktioner (1p)



(i) An initial call

And this is the structure of the function:-

(ii) the stop condition – usually with is_empty(X)
(iii) a non recursive call
(iv) the recursive call

The recursive definition of a sequence is

```
S  ::=  H  T | ¤        // ¤ = empty
H  ::=  element         // a non-recursive definition
T  ::=  S               // the recursive part of the definition – often tail
```
recursion

The pseudocode – which follows from the above definition is :-

```
static listref b_add(valtype v, listref L)
{
 return is_empty(L)              ?  create_e(v)                    // stop
      :  v < get_value(head(L)) ?  cons(create_e(v), L)            // non-recursive
      :                             cons(head(L), b_add(v, tail(L))); // recursive (tail)
}
```

Or if you prefer the code with if-statements:- **stop + non-recursive + tail recursive**

```
static listref b_add(valtype v, listref L)
{
 if (is_empty(L))                return create_e(v);
 if (v < get_value(head(L)))     return cons(create_e(v), L);
                                 return cons(head(L), b_add(v, tail(L)));
}
```

(b) Utifrån din definition i (a) ovan, skriv rekursiv pseudokod för att lägga till ett element i en sekvens i stigande ordning. **Ange alla antaganden.**
Visa hur Din pseudokod fungerar med detta exempel – **lägga till 7 till (1,4,5)**

(2)

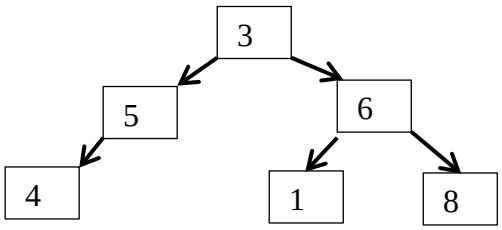
```
static listref b_add(valtype v, listref L)
{
 return is_empty(L)              ?  create_e(v)
      :  v < get_value(head(L)) ?  cons(create_e(v), L)
      :                             cons(head(L), b_add(v, tail(L)));
}
```

Assume: **cons**: insert an element at the **head** of the list
head: returns a reference to the head of the list
tail: returns a reference to the tail of the list

b_add(7, (1, 4, 5)) --- initial call
⇨ cons(1, b_add(7, (4,5))) --- recursive call 1
⇨ cons(1, cons(4, b_add(7, (5)))) --- recursive call 2
⇨ cons(1, cons(4, cons(5, b_add(7, (¤))))) --- recursive call 3
--- empty case (**stop condition**)
Start the returns from the recursive calls
⇨ cons(1, cons(4, cons(5, (7)))) --- return from recursive call 3
⇨ cons(1, cons(4, (5,7))) --- return from recursive call 2
⇨ cons(1, (4, 5, 7)) --- return from recursive call 1
⇨ (1, 4, 5, 7) --- return from initial call

(c) Utifrån din definition i (a) ovan, skriv rekursiv pseudokod för funktionen **T2Q( )** (träd till kö) från trädlabben **Ange alla antaganden.**
Visa hur Din pseudokod fungerar med detta exempel:

(2p)
Totalt 5p



One solution might be:-

```
static void T2Q(treeref T, int qpos) {
    hqarr[qpos] = T;
    if (!is_empty(T)) { T2Q(LC(T), qpos*2); T2Q(RC(T), qpos*2+1);  }
}
```

Note (i) there are 2 recursive calls and (ii) the contents of the Q are references to the tree nodes however these will be represented as the values of the node or ¤ (empty)

Initial call T2Q(T, 1) and the hqarr is [-,-,-,-,-,-,-,-,-,-,-,-,-,-,-]

Rewriting the initial tree as (((4) 5 (¤)) 3 ((1) 6 (8))) gives

Initial call T2Q((((4) 5 (¤)) 3 ((1) 6 (8))), 1) and hqarr is [-,-,-,-,-,-,-,-,-,-,-,-,-,-,-]

hqarr[1] = 3 ➔ [3,-,-,-,-,-,-]

left sub-tree

recursive call T2Q(((4) 5 (¤)), 2) gives hqarr[2] = 5 ➔ [3,5,-,-,-,-,-,-,-,-,-,-,-,-,-]
recursive call T2Q(((¤) 4 (¤)), 4) gives hqarr[4] = 4 ➔ [3,5,-,4,-,-,-,-,-,-,-,-,-,-,-]
recursive call T2Q(¤, 8) gives hqarr[8] = ¤ ➔ [3,5,-,4,-,-,-,¤,-,-,-,-,-,-,-]
recursive call T2Q(¤, 9) gives hqarr[9] = ¤ ➔ [3,5,-,4,-,-,-,¤,¤,-,-,-,-,-,-]
recursive call T2Q(¤, 5) gives hqarr[5] = ¤ ➔ [3,5,-,4,¤,-,-,¤,¤,-,-,-,-,-,-]

right sub tree

recursive call T2Q(((1) 6 (¤)), 3) gives hqarr[3] = 6 ➔ [3,5,6,4,¤,-,-,¤,¤,-,-,-,-,-,-]
recursive call T2Q(((¤) 1 (¤)), 6) gives hqarr[6] = 1 ➔ [3,5,6,4,¤,1,-,¤,¤,-,-,-,-,-,-]
recursive call T2Q(¤, 12) gives hqarr[12] = ¤ ➔ [3,5,6,4,¤,1,-,¤,¤,-,-,¤,-,-,-]
recursive call T2Q(¤, 13) gives hqarr[12] = ¤ ➔ [3,5,6,4,¤,1,-,¤,¤,-,-,¤,¤,-,-]
recursive call T2Q(((¤) 8 (¤)), 7) gives hqarr[7] = 8 ➔ [3,5,6,4,¤,1,8,¤,¤,-,-,¤,¤,-,-]
recursive call T2Q(¤, 14) gives hqarr[14] = ¤ ➔ [3,5,6,4,¤,1,8,¤,¤,-,-,¤,¤,¤,-]
recursive call T2Q(¤, 15) gives hqarr[15] = ¤ ➔ [3,5,6,4,¤,1,8,¤,¤,-,-,¤,¤,¤,¤]

(4) Prims algoritm

a) Tillämpa **Prims algoritm** (nedan) på den **oriktade** grafen:

(a-7-b, a-1-c, a-10-d, b-3-c, b-2-e, c-15-d, c-4-e, c-9-f, d-4-f, e-8-f).

Börja med nod a. Ange alla antaganden och visa alla beräkningar och mellanresultat. Vad representerar resultatet? **Ange varje steg i din beräkning.**

Rita delresultatet efter varje iteration.

(2p)

b) **Förklara principerna bakom Prims** algoritm.

(2p)

c) Visa hur Du skulle använda **principerna** bakom **Kruskals algoritm** för att bekräfta resultatet från Prims algoritm.

(1p)

Totalt 5p

Ange *alla* antaganden och visa *alla* beräkningar och mellanresultat

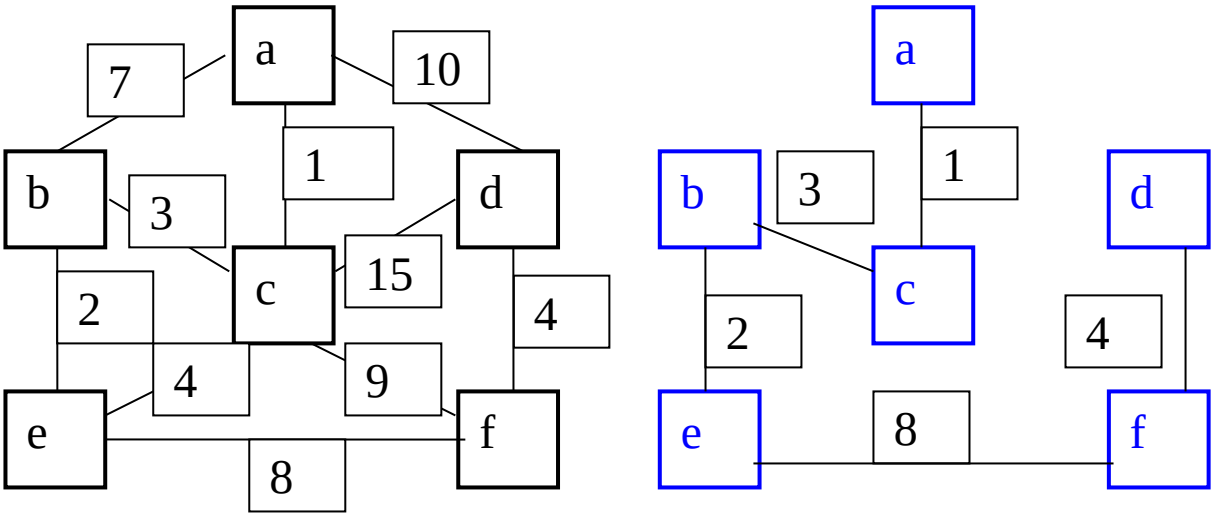
```
Prim ( node v) -- v is the start node

{   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; }
   }
}
```

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

Cost 18

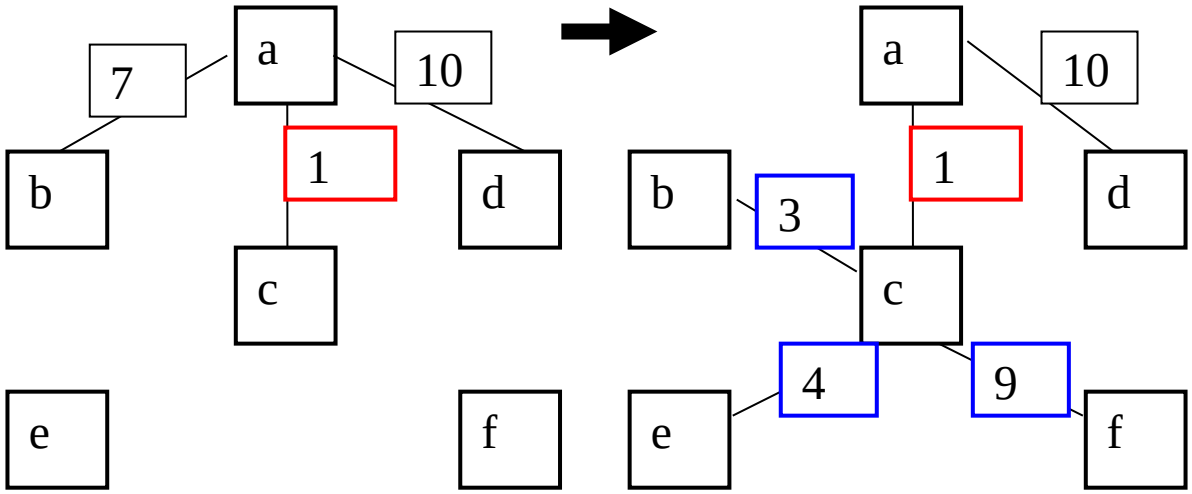


Draw the cost matrix C and array D

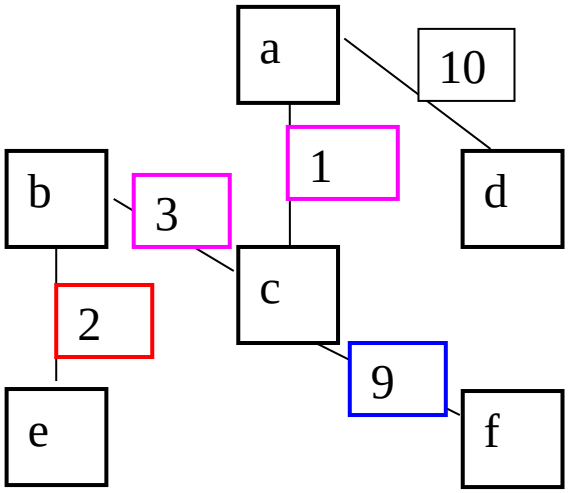
	a	b	c	d	e	f
a		**7**	**1**	**10**		
b	**7**		**3**		**2**	
c	**1**	**3**		**15**	**4**	**9**
d	**10**		**15**			**4**
e		**2**	**4**			**8**
f			**9**	**4**	**8**	

a	**b**	**c**	**d**	**e**	**f**
lowcost	**7**	**1**	**10**	**§**	**§**
closest	**a**	**a**	**a**	**a**	**a**

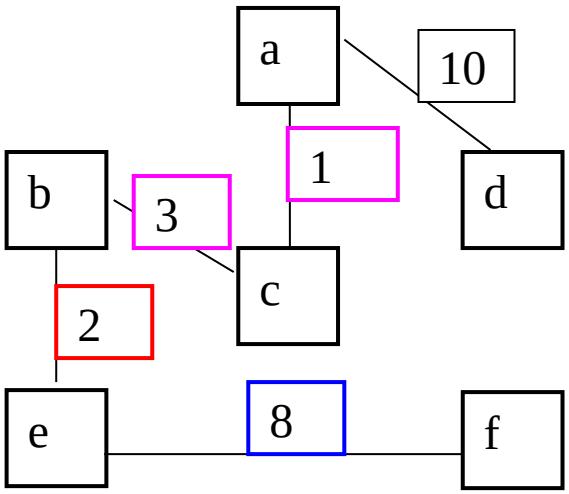
Minedge**: lowcost: 7 1 10 § § closest: a a a a a** U = {a,c} V-U = {b,d,e,f} min = **1**; k = **c**
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = b; if C[c,b] < lowcost[b] then { lowcost[b] = C[c,b]; closest[b] = c } → 3<7 ➔ **c-3-b**
j = d; if C[c,d] < lowcost[d] then { lowcost[d] = C[c,d]; closest[d] = c }→ 15<10 ➔ no change
j = e; if C[c,e] < lowcost[e] then { lowcost[e] = C[c,e]; closest[e] = c } → 4<§ ➔ **c-4-e**
j = f; if C[c,f] < lowcost[f] then { lowcost[f] = C[c,f]; closest[f] = c } → 9<§ ➔ **c-9-f**



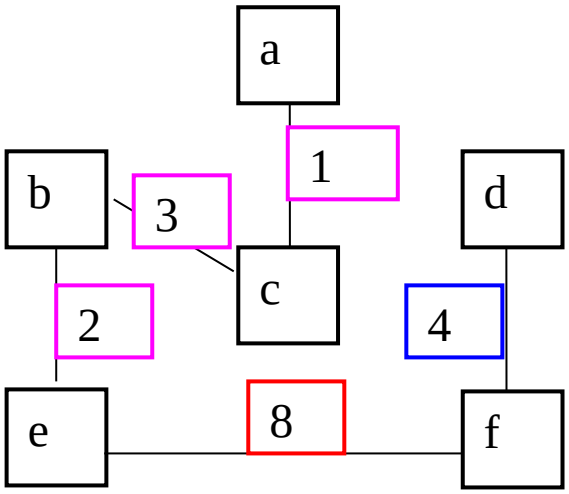
Minedge**: lowcost: 3 1 10 4 9 closest: c a a c c** U = {a,c,b} V-U = {d,e,f} min = **3**; k = **b**
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = d; if C[b,d] < lowcost[d] then { lowcost[d] = C[b,d]; closest[d] = b}→ §<10 ➔ no change
j = e; if C[b,e] < lowcost[e] then { lowcost[e] = C[b,e]; closest[e] = b } → 2<4 ➔ **b-2-e**
j = f; if C[b,f] < lowcost[f] then { lowcost[f] = C[b,f]; closest[f] = b } → §<9 ➔ no change



Minedge**: lowcost: 3 1 10 2 9 closest: c a a b c** U = {a,c,b,e} V-U = {d,f} min = **2**; k = **e**
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = d; if C[e,d] < lowcost[d] then { lowcost[d] = C[b,d]; closest[d] = e}→ §<10 ➔ no change
j = f; if C[e,f] < lowcost[f] then { lowcost[f] = C[b,f]; closest[f] = e } → 8<9 ➔ **e-8-f**

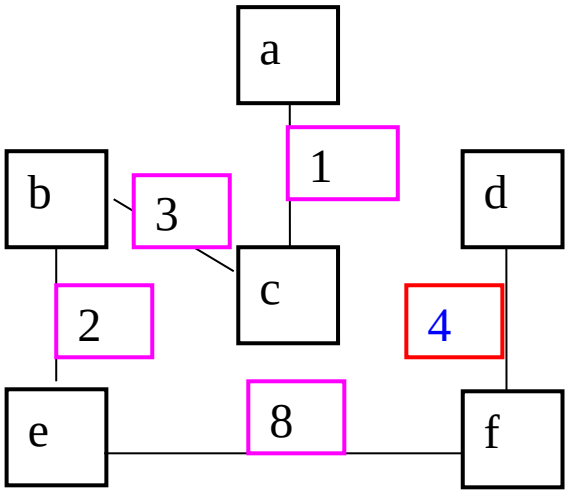


Minedge**: lowcost: 3 1 10 2 8 closest: c a a b e** U = {a,c,b,e,f} V-U = {d} min = **8**; k = **f**
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = d; if C[f,d] < lowcost[d] then { lowcost[d] = C[f,d]; closest[d] = f} → 4<10 ➔ **d-4-f**



Minedge**: lowcost: 3 1 4 2 8 closest: c a f b e** U = {a,c,b,e,f, d} V-U = {¤} min = **4**; k = **d**

V-U is empty – STOP. **COST = 18**



Principles behind Prim's – to derive the MST Minimal Spanning Tree for a graph

Method:-

1. Choose a start node (usually a) and mark as visited U = {a} U is a component
2. Calculate the distances between a and the remaining nodes V-U and draw the "MST" calculated so far
3. Choose the closest node x to a and mark as visited U = {a,x}
4. If the distance x to any node y in V-U is shorter then readjust costs by removing the previous edge and adding the edge (x y) – now we have a new "MST"
5. U represents the visited nodes and V-U the unvisited nodes – choose the shortest edge from the visited nodes to a node in V-U and add this to U
6. Repear 4 & 5 until the MST has been found and V-U = ¤ (empty)

Principles:-

Prim's finds the shortest edge from the component (visited nodes) to the unvisited nodes and then adds this node x to the visited nodes (U) and readjusts the edges in the MST if an edge (x y) where y is in V-U is shorter than the previously chosed edge.

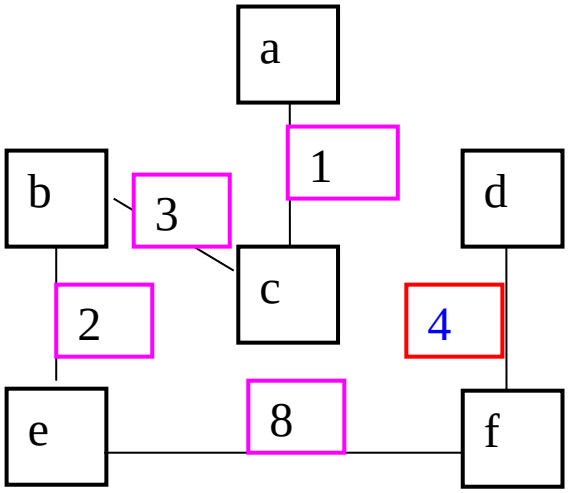
Double check with Kruskal's

Build the Priority Queue (PQ) – double check to ensure that all 10 edges are present in the PQ!

a-c-1
b-e-2
b-c-3
c-e-4
d-f-4
a-b-7
e-f-8
c-f-9
a-d-10
c-d-15

Now work through the PQ to connect the 6 components: a, b, c, d, e, f

a-c-1 gives 5 components a-1-c, b, d, e, f
b-e-2 gives 4 components a-1-c, b-2-e, d, f
b-c-3 gives 3 components a-1-c-3-b-2-e, d, f
c-e-4 gives no result since c and e are in the same component a-1-c-3-b-2-e
d-f-4 gives 2 components a-1-c-3-b-2-e, d-4-f
a-b-7 gives no result since a and b are in the same component a-1-c-3-b-2-e
e-f-8 gives 1 component a-1-c-3-b-2-e-8-f-4-d
finished!



(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

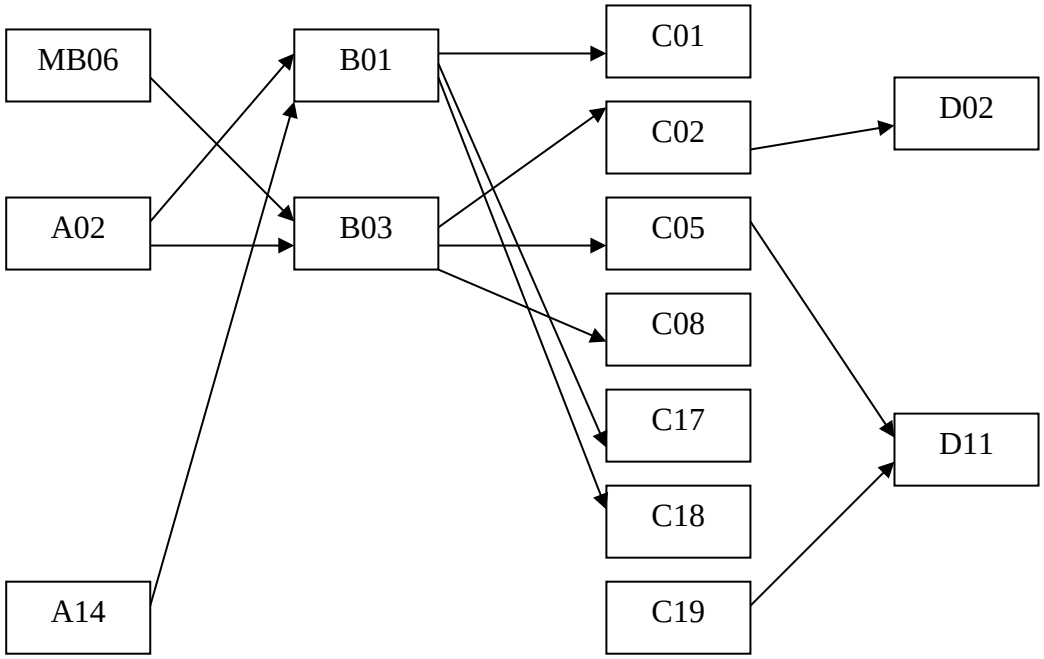
```
tsort(v) -- prints reverse topological order of a DAG from v
{       mark v visited
        for each w adjacent to v if w unvisited tsort(w)
        display(v)
        }
```

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

Vilka begränsningar måste man ta hänsyn till?

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) Diskussion

Diskutera ingående meningen bakom denna kurs. Vilka var de huvudidéer som studerats under kursens lopp och vilken relevans har dessa idéer för datavetenskap, för andra ämnen inom datavetenskap och för programmering? **Ange gärna exempel** för att stödja Din diskussion.

5p

Open Question – marks for good answers.