Summary of exam results 170117 (ordinary exam) & 170331 (resit exam)

Topic	type of question
Max p	maximum points (5p or 10p)
Avg p	average result (points)
# >= 50%	number of students with >= 50% of points for question (e.g. 2.5p for max 5p; 5p for max 10p)
% >= 50%	percentage of students with >= 50% of points for question

170117 – 57 exam papers – 19 students passed (33%)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Topic	Short	Seq	Abs	Heap	Hash	Tree	graph
Max p	**5**	**5**	**5**	**5**	**5**	**5**	**10**
Avg p	**2,8**	**1,06**	**1,18**	**2,6**	**2,32**	**1,6**	**3,9**
# >= 50%	**40**	**15**	**11**	**37**	**30**	**20**	**24**
% >= 50%	**70%**	**26%**	**19%**	**65%**	**53%**	**35%**	**42%**

Ranking: short, heap, hash, graph, tree, abstraction, sequence

170331 – 32 exam papers – 9 students passed (31%)

	Q1	Q2	Q3	Q4	Q5	Q6	Q7
Topic	Short 1	Short 2	AVL	Topsort	Rec	Hash	graph
Max p	**5**	**5**	**5**	**5**	**5**	**5**	**10**
Avg p	**2,83**	**1,48**	**3,13**	**0,55**	**2,05**	**3,0**	**4,05**
# >= 50%	**22**	**5**	**23**	**3**	**13**	**26**	**12**
% >= 50%	**69%**	**16%**	**72%**	**9%**	**41%**	**81%**	**38%**

Ranking: AVL, hash, Short 1, Recursion, graph, Short 2, Topological Sort

General problems with answers to exam questions

1. Not knowing the topic – some students have obviously not read the lecture notes and the course material OR you have not understood these AND have not asked questions during the course.
    a. You show know the basic definitions of set, sequence, tree & graph as well as the recursive definitions of sequence & tree AND the operations on these data structures:- create, add, find, remove, count (the number of elements)
    b. You should know what the algorithms discussed in the course actually do and how they work
    c. There were 5 named graph algorithms – people mix these up especially Dijkstra and Prim
        i. **Dijkstra & Dijkstra-SPT** – **single source shortest paths** in a graph – builds a component (result: array D or for SPT (Shortest Path Tree) 3 arrays D, E, L)
        ii. **Floyd** – **all pairs shortest path** (result: matrix).
        iii. **Warshall** – **the transitive closure** – i.e. is there a path between any two given nodes (a, b) not necessarily distinct – a = b shows that there are cycles in the graph (result: matrix).
        iv. **Prims** – constructs a **Minimal Spanning Tree** (MST) for an undirected graph – like Dijkstra Prim's builds a component from the start node. The MST is a FREE TREE with n nodes and (n-1) edges and gives the cheapest way of connecting all the nodes in the graph.
        v. **Kruskal's** - constructs a **Minimal Spanning Tree** (MST) by making each node a component, makes a PQ (priority queue) of the edges and uses this to join nodes in two **SEPARATE** components until there is one component which is the MST.
2. Not reading the questions carefully and hence not answering all the points
3. Not knowing how to articulate answers i.e. **not prepared in advance**
    a. Many answers do not contain a single sentence of explanation (in Swedish or English) – this is unacceptable at University level – you must be able to articulate ideas and explain your answers – if not, more problems arise when you come to write your C-dissertation
4. I have also listed these comments on https://www.cs.kau.se/cs/DFR/index.php?examcomm=1 but many students appear not to have read this. I have written course notes - https://www.cs.kau.se/cs/education/courses/dvgb03/p5/index.php?studyplan=1 and revision notes & facits to the last 5 years' exams - https://www.cs.kau.se/cs/education/courses/dvgb03/revision/ https://www.cs.kau.se/cs/education/courses/dvgb03/exam_papers/ - you are expected to have read these. Again few ask questions- why not?

Specific problems with the resit exam questions 170331

Q2a – the question said "**ett binärt träd (OBS ej BST)**" – many knew the code for a BST and wrote that BUT that was not what was asked. Read the question **carefully**!

Q2b – add a value v to a heap – (i) the space for a new element is created (ii) if the value of the parent is less than v COPY the parent value to the space below (iii) when the while loop terminates, the index i is the place for the new element in the array. The new element is NOT added in the new space in the beginning. People do not read the code carefully!

Q2e rebalancing an AVL tree. People knew the rotations BUT not the rebalancing criteria (i) calculate the balance factor bf = h(LC(T) – h(RC(T)) to decide if this is a LEFT or RIGHT rotation (ii) then look at the LC(T) or RC(T) and calculate the balance factor there to decide if the rotation is SINGLE (addition to the outside of the tree - i.e. LC of LC or RC of RC)) or DOUBLE (addition to the inside of the tree – i.e RC of LC or LC of RC).

Q3 – definition of an **AVL tree -** BST (not BT!) with a balance constraint
| h(LC(T)) – h(RC(T)) | < 2 – NB absolute value | …. |
You should absolutly know these basic definitions.

Q4 – **topological sorting** very few people knew how to do this – even given the code for tsort – this is applied to the adjacency list and is recursive

Q5 – **recursion** – definition – **partially defined** in terms of itself and function – a function which **conditionally** calls itself
Sequence definition S::= H T | empty; H ::= element; T ::= S
Several seemed not to know this
The resulting **programming pattern** is therefore (i) empty case (ii) head case (non-recursive) (iii) tail case – recursive.
Many forgot to start with the **empty case** – **if (is_empty(S)) …** – this should be automatic by now.
BT definition BT ::= LC N RC | empty; N ::= element; LC, RC ::= BT
The resulting **programming pattern** is therefore (i) empty case (ii) LC (iii) RC (iv) N – in many cases is is useful to leave the node until last.

The code reflects this – see below in the facit.

Some people have difficulties in articulating an answer – **you should prepare in advance**.

Q7a & Q7b – lack of detail and not following the given code for the algorithm. Not reading the question "instructions"

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

People do not show each step in the calculations. If you do not do this is is difficult to know if you are following the algorithm or writing a "**by inspection**" solution. If I do not see stepwise calculations if will assume the latter and if the answer is correct you will get 0,5p but will probably fail the exam.

Some people write their own solutions but it is clear that they are not following given code for the algorithm – therefore you have not answered the question in the required manner. Zero points!

Q7c – describe in detail the principles behind Prim's algorithm (see the facit below). I wrote

"**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."

Many ignored this. I had also mention this in class but with an attendance rate of around 50%, many were not there to hear this.

The course is Data Structures and **Algorithms** – you are expected to understand and be able to describe the principles behind the algorithms

FACIT TILL
OMTENTAMEN I
DATASTRUKTURER OCH ALGORITMER DVG B03

170331 kl. 08:15-13:15

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) Vad är ”big-O” för Dijkstras algoritm?
(b) Vad är ”big-O” för Floyds algoritm?
(c) Vad är ”big-O” för lägga-till-operationen i hashning?
(d) Vad gör Warshalls algoritm?
(e) Vad gör en postorder traversering av ett binärt träd?
(f) Vad är en graf?
(g) Vad representerar kanterna i en graf?
(h) Vad är ett ”Free Tree”?
(i) Vad händer när man lägger till en kant till i ett ”Free Tree”?
(j) Vad är abstraktion?

Totalt 5p

(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 abstraktion?

ModellingAbstraction is where certain common or relevant properties of a system or entity are used to describe or model that system or entity.

Collection Abstraction is when common operations for set, sequence, tree and graph are identified and described. Add, find, remove an element; count.

Implementation Abstraction is when ADTs are implemented as an abstract machine – i.e. the implementation details (arrays / pointers and structures) are hidden.

Marks for good answers.

Totalt 5p

(2) <u>Ge ett kortfattat svar till följande uppgifter (a)-(e).</u>

(a) Skriv **rekursiv (pseudo)kod** för hitta-operation (find) i **<u>ett binärt träd (OBS ej BST)</u>**.

```
int find(BT, v) {
        return      is_empty(BT)                ?  F
                 :  v == get_value(node(BT))    ?  T
                 :                                 find(LC(BT), v) || find(RC(BT), v);
}
```

(b) Förklara hur lägga till operationen (add) i en heap fungerar. Se koden nedan. **<u>OBS</u>** – ge inte en översättning kod ➔ svenska! **Förklara <u>principen</u> och/eller ge ett <u>exempel</u>**.

```
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
```

Marks for good answers.

(c) Skriv **<u>abstrakt</u> <u>rekursiv (pseudo)kod</u>** för hitta-operation (**find**) i en sekvens. Funktionen ska returnera en **<u>referens</u>** till elementet eller **<u>NULLREF</u>** om inte värdet finns.

```
seqref be_find_val( seqref S, int v)
{
  return (is_empty(S) || (v==get_value(head(S)))) ? S
      :   be_find_val(tail(S), v);
}
```

(d) Beskriv hur man förvandla ett **generellt träd** till ett **binärt träd**.
(1) The first child becomes the left child of the parent
(2) The subsequent children become the right child of their predecessor

(e) Ge definitionen av ett **komplett träd** (complete tree).
(1) A perfect BT of height h has exactly 2^h -1 elements
(2) Complete – perfect on the next lowest level and the lowest level is filled from the left.

5p

(3) AVL-träd

(a) Ge en definition av ett AVL-träd **1p**

A Binary Search Tree (BST) where |height(LC(BST) – height(RC(BST))| < 2

(b) Beskriv **med exempel** de fyra rotationsoperationerna på ett AVL-träd. **2p**

SLR – single left rotation – e.g. (*, 9, (*, 10, 11)) ➔ (9, 10, 11)
SRR – mirror image
DLR - DLR(T) = SRR(RC(T)) + SLR(T)
DRR – mirror image

See http://www.cs.kau.se/cs/education/courses/dvgb03/lectures/AVL_InsertFP.pdf for examples

(c) Beskriv hut man bestämmer vilken rotationsoperation som behövs
för att ombalancera ett AVL-träd. **2p**

If the balance factor (bf = height(LC(T)) – height(RC(T)))
bf >1 ➔ right rotation i.e. the LC is deeper)
bf <-1 ➔ left rotation i.e. the RC is deeper
and
addition to the OUTSIDE of the tree ➔ single rotation
addition to the INSIDE of the tree ➔ double rotation

Totalt 5p

(4) 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)
        }
```

Tillämpa den givna algoritmen (ovan) till problemet för att hitta ordning i vilken en student ska läsa kurserna.

Använd denna ordning i kurslistan:
MB06, A02, A14, B01, B03, C01, C02, C05, C08, C17, C18, C19, D02, D11

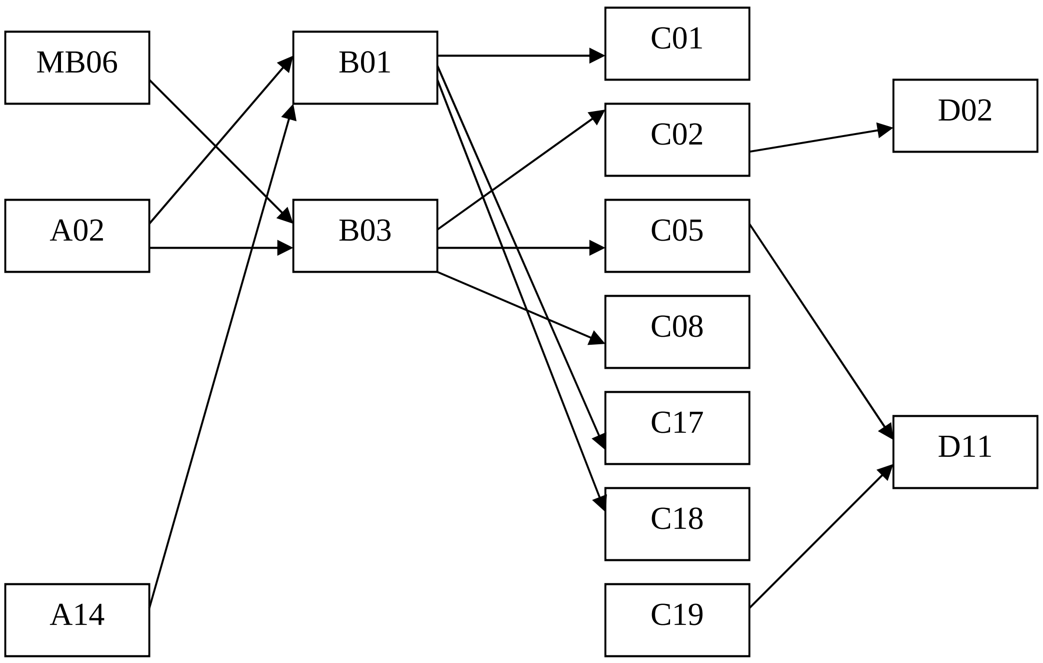
4p

Beskriv ett annat sätt att utföra en topologisk sortering?

1p

Totalt 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: 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:

Method 1: 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

Course order – method 2: 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.

(5) Rekursion

Förklara utförligt de för- och nackdelarna med rekursion. Förklara vad en rekursiv definition samt en rekursiv funktion är. Vilka **programmeringsmönster** finns som direkt resultat av de rekursiva definitionerna för en sekvens och ett binärt träd? Ge exempel och (pseudo)kod för att illustrera din diskussion.

5p

Pattern – Sequence – empty case / head case / tail case

Empty test / non-recursive case for head / recursive case for tail

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

Pattern – Tree – empty case / LC case / RC case / node case

Empty test / recursive case for LC / recursive case for RC / non-recursive case for node

```
static treeref b_add(treeref T, int v)
{
 return is_empty(T)           ? create_node(v)
   : v < get_value(node(T)) ? cons(b_add(LC(T), v), node(T), RC(T))
   : v > get_value(node(T)) ? cons(LC(T), node(T), b_add(RC(T), v))
   :                          T;
}
```

Recursion +ve & -ve

+ve: definitions lead to program patterns
+ve: more compact code
+ve: allows coding on a more abstract level

-ve: code may be harder to understand

Marks for good answers.

(6) Hashning

I hashning har föjlande metoder för kollisionshantering presenterats, nämligen

1) Open addressing (closed hashing/linear probing)	f(i) = i
2) Quadratic probing	f(i) = i * i
3) Double hashing	f(i) = i * H_2(key)

Där "i" är värdet på antalet kollisioner (d.v.s. 1, 2, …).

Allmänt sett kan man beskriva kollisionshantering som **H(key) + f(i)** i varje fall.
Anta att **H(key) = key mod 10**. **Hash space = array H[10].**
Anta att **H_2(key) = 7 – (key mod 7).**

Dessa metoder kan betraktas som en historisk utveckling d.v.s. att varje metod försöker att lösa problem med den föregående metoden men i sin tur kan introducera nya problem.

Tillämpa dessa 3 metoder på sekvensen **4, 36, 44, 5, 7, 64, 24** och **visa varje steg** i Dina beräkningar samt diskutera för- och nackdelarna med dessa metoder.

3p

See below for the calculations and advantages / disadvantages.

Vilka resultat är **mätbara** när man tillämpar dessa metoder? **1p**

(1) The number of collisions
(2) The reduction in the number of collisions from 1 method to the next
(3) The load in the hash space i.e. % slots occupied

Under vilka omständigheter skulle dubbelhashningsmetoden ovan bete sig på ett likadant sätt som linjärprobning?

1p

This implies that **H_2(key)** gives a result of 1 hence f(i) = 1*1, 1*2, etc
This would happen if (key mod 7) is 6 since 7-6 ➔ 1
Example: if the input sequence were 6, 13, 20, 27, 34, 41

Totalt 5p

Open addressing (closed hashing) f(i) = i

1. Add 4 to slot 4 (4 mod 10)
2. Add 36 to slot 6 (36 mod 10)
3. Add 44 to slot 4 (44 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
4. Add 5 to slot 5 (5 mod 10) → **collision** find next free slot 5 + f(1) = 5+1 = 6
   → **collision** find next free slot 5 + f(2) = 5+2 = 7
5. Add 7 to slot 7 (7 mod 10) → **collision** find next free slot 7 + f(1) = 7+1 = 8
6. Add 64 to slot 4 (64 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
   → **collision** find next free slot 4 + f(2) = 4+2 = 6
   → **collision** find next free slot 4 + f(3) = 4+3 = 7
   → **collision** find next free slot 4 + f(4) = 4+4 = 8
   → **collision** find next free slot 4 + f(5) = 4+5 = 9
7. Add 24 to slot 4 (24 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
   → **collision** find next free slot 4 + f(2) = 4+2 = 6
   → **collision** find next free slot 4 + f(3) = 4+3 = 7
   → **collision** find next free slot 4 + f(4) = 4+4 = 8
   → **collision** find next free slot 4 + f(5) = 4+5 = 9
   → **collision** find next free slot 4 + f(6) = 4+6 = 0*

* (10 mod 10)

15 collisions

slot	**+4**	**+36**	**+44**	**+5**	**+7**	**+64**	**+24**
0							24
1							
2							
3							
4	4	4	4	4	4	4	4
5			44	44	44	44	44
6		36	36	36	36	36	36
7				5	5	5	5
8					7	7	7
9						64	64

Quadratic Probing **f(i) = i * i**

1. Add 4 to slot 4 (4 mod 10)
2. Add 36 to slot 6 (36 mod 10)
3. Add 44 to slot 4 (44 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
4. Add 5 to slot 5 (5 mod 10) → **collision** find next free slot 5 + f(1) = 5+1 = 6
   → **collision** find next free slot 5 + f(2) = 5+4 = 9
5. Add 7 to slot 7 (7 mod 10)
6. Add 64 to slot 4 (64 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
   → **collision** find next free slot 4 + f(2) = 4+4 = 8
7. Add 24 to slot 4 (24 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
   → **collision** find next free slot 4 + f(2) = 4+4 = 8
   → **collision** find next free slot 4 + f(3) = 4+9 = 3*

* (13 mod 10)

8 collisions **a reduction of 7 from 15**

slot	+4	+36	+44	+5	+7	+64	+24
0							
1							
2							
3							24
4	4	4	4	4	4	4	4
5			44	44	44	44	44
6		36	36	36	36	36	36
7					7	7	7
8						64	64
9				5	5	5	5

Double Hashing **$f(i) = i * H_2(key)$ where $H_2(key) = 7 - (key \bmod 7)$**

1. Add 4 to slot 4 (4 mod 10)
2. Add 36 to slot 6 (36 mod 10)
3. Add 44 to slot 4 (44 mod 10) → **collision** find next free slot 4 + f(1) = 4+5 = 9 (a)
4. Add 5 to slot 5 (5 mod 10)
5. Add 7 to slot 7 (7 mod 10)
6. Add 64 to slot 4 (64 mod 10) → **collision** find next free slot 4 + f(1) = 4+6 = 0* (b)
7. Add 24 to slot 4 (24 mod 10) → **collision** find next free slot 4 + f(1) = 4+4 = 8 (c)

(a) (44 mod 7) = 2, (7-2) = 5, (1 * 5) = 5
(b) (64 mod 7) = 1, (7-1) = 6, (1 * 6) = 6 *(10 mod 10) = 0
(c) (24 mod 7) = 3, (7-3) = 4, (1 * 4) = 4

3 collisions **a reduction of 5 from 8 (quadratic probing) and 12 from 15 (linear probing)**

slot	+4	+36	+44	+5	+7	+64	+24
0						64	64
1							
2							
3							
4	4	4	4	4	4	4	4
5				5	5	5	5
6		36	36	36	36	36	36
7					7	7	7
8							24
9			44	44	44	44	44

Potential problems with many collisions

(1) Open Adressing (Closed Hashing) – Linear Probing

i. the **search** gets longer and the search time goes from O(1) to O(n)
ii. primary clustering (sequence from a given slot)

(2) Quadratic Probing

i. the **search** gets longer and the search time goes from O(1) to O(n)
ii. secondary clustering
iii. when the load (number of occupied slots) >50% this method may no longer work!

(3) Double Hashing

i. Disadvantage – 2 hash functions must be calculated

Measurable

(i) The number of collisions
(ii) The reduction in number of collisions between methods
(iii) Load factor – occupied places as a percentage of the total number of places

(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 varje mellanresultatet**.

(a, b, 10), (a, d, 30), (a, e, 100), (b, c, 50), (c, e, 10), (d, c, 20), (d, e, 60)

```
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

See http://www.cs.kau.se/cs/education/courses/dvgb03/revision/index.php?SPTex1=1

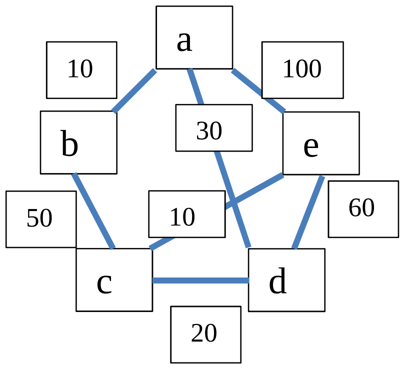
See http://www.cs.kau.se/cs/education/courses/dvgb03/revision/DijkstraSPTEx.pdf
for the pictures at each stage.

See below for the calculations

Graph: **(a, b, 10), (a, d, 30), (a, e, 100), (b, c, 50), (c, e, 10), (d, c, 20), (d, e, 60)**

Cost matrix C:- ¤ = infinity

	a	**b**	**c**	**d**	**e**
a	**-**	**10**	**¤**	**30**	**100**
b	**10**	**-**	**50**	**¤**	**¤**
c	**¤**	**50**	**-**	**20**	**10**
d	**30**	**¤**	**20**	**-**	**60**
e	**100**	**¤**	**10**	**60**	**-**



D E L – initial values

	a	**b**	**c**	**d**	**e**
D	-	10	¤	30	100
E	-	a	a	a	a
L	-	10	¤	30	100

1st iteration

S = {a}, V-S = {b, c, d, e} closest = b ➔ w = b, S = {a, b}, V-S = {c, d, e}

w = b	if D[w] + C[w,v] < D[v]	➔ change values i.e. a shorter path has been found	
v = c	D[b] + C[b,c] < D[c]	➔ 10 + 50 = 60 < ¤	➔ yes - change
v = d	D[b] + C[b,d] < D[d]	➔ 10 + ¤ = ¤ < 30	➔ no change
v = e	D[b] + C[b,e] < D[e]	➔ 10 + ¤ = ¤ < 100	➔ no change

	a	**b**	**c**	**d**	**e**
D	-	10	**60**	30	100
E	-	a	**b**	a	a
L	-	10	**50**	30	100

2nd iteration

S = {a, b}, V-S = {c, d, e} closest = d ➔ w = d, S = {a, b, d}, V-S = {c, e}

w = d	if D[w] + C[w,v] < D[v]	➔ change values i.e. a shorter path has been found	
v = c	D[d] + C[d,c] < D[c]	➔ 30 + 20 = 50 < 60	➔ yes - change
v = e	D[d] + C[d,e] < D[e]	➔ 30 + 60 = 90 < 100	➔ yes - change

	a	**b**	**c**	**d**	**e**
D	-	10	**50**	**30**	**90**
E	-	a	**d**	**a**	**d**
L	-	10	**20**	**30**	**60**

3rd iteration

S = {a, b, d}, V-S = {c, e} closest = c ➔ w = c, S = {a, b, d, c}, V-S = {e}

w = c if D[w] + C[w,v] < D[v] ➔ change values i.e. a shorter path has been found
v = e D[c] + C[c,e] < D[e] ➔ 50 + 10 = 60 < 90 ➔ yes - change

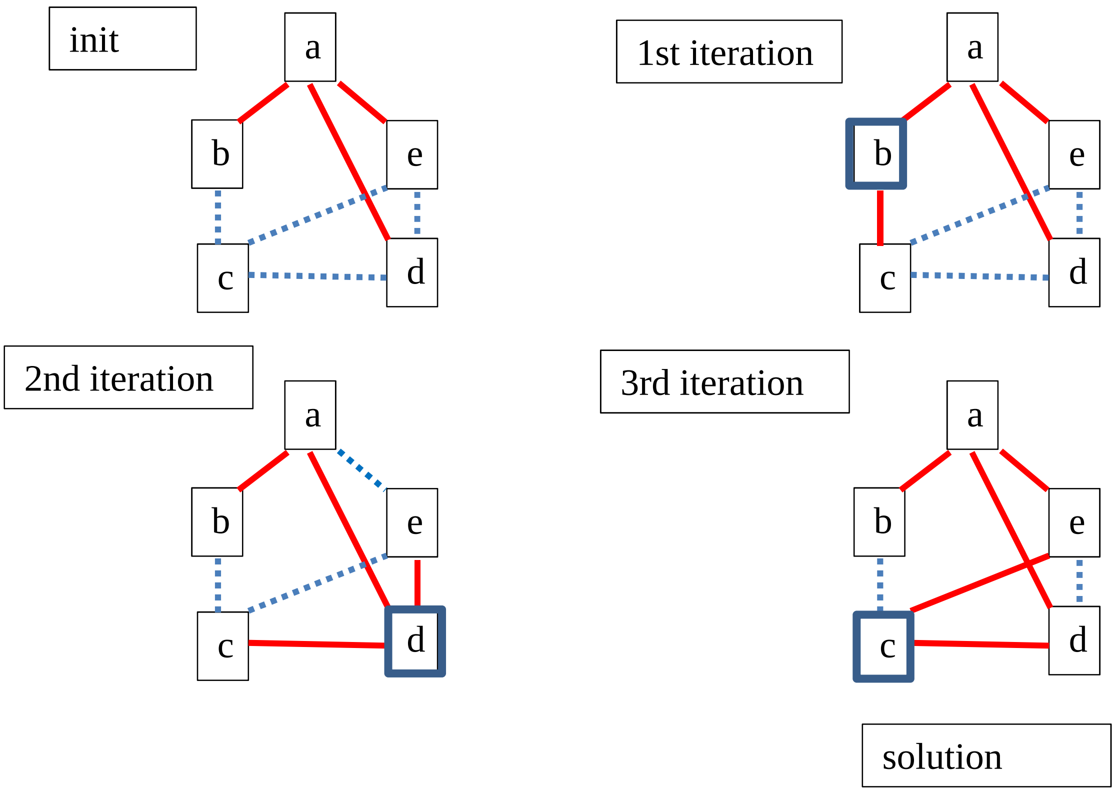
	a	**b**	**c**	**d**	**e**
D	-	10	**50**	30	**60**
E	-	a	**d**	a	**c**
L	-	10	**20**	30	**10**

4th iteration

S = {a, b, d, c}, V-S = {e} closest = e ➔ w = e, S = {a, b, d, c, e}, V-S = { } (empty) STOP!

Answer:-

	a	**b**	**c**	**d**	**e**
D	**-**	**10**	**50**	**30**	**60**
E	**-**	**a**	**d**	**a**	**c**
L	**-**	**10**	**20**	**30**	**10**



(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 varje mellanresultatet**.
- **Anta att närlistan (adjacency list) skapas i alfabetisk ordning**.

(a-3-b, a-3-c, a-3-d, b-3-c, b-3-e, c-3-d, c-3-e, c-3-f, d-3-f, e-3-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

Prim's calculates the Minimal Spanning Tree for an undirected graph.

<u>Principles</u>

1. **Choose a start node a - this node is the first in a component**
2. **<u>All nodes</u> + all <u>incident edges</u> from the start node form the component**
3. **Choose the <u>shortest</u> <u>EDGE</u> from a to any other unvisited node x in the component**
4. **This node x is marked as visited**
5. **Check if there is an <u>edge of lower cost</u> from x to the remaining <u>unvisited</u> nodes – if so update the component**
6. **Choose the <u>shortest edge</u> to an unvisited node and repeat from (3)**
7. **The result is an MST**

The short version:- choose a start node a and incident edges as the component, choose the shortest <u>edge</u> to another node x, mark as visited, check edge weights from x to the remaining unvisited nodes and update the component if shorter. Repeat until all nodes have been visited.

The even shorter version:-
Prim's builds a component and repeatedly adds nodes with the shortest edge from the component until all nodes have been visited.

For this example

1. A is the start node
2. The component is (a, b, 3), (a, c, 3), (a, d, 3), (e), (f) – visited {a}, unvisited {b,c,d,e,f}
3. b is the nearest node
4. mark as visited - visited {a,b}, unvisited {c,d,e,f}
5. (b, c, 3), (b, d, 3) are NOT shorter but (b, e, 3) is – the component is now (a, b, 3), (a, c, 3), (a, d, 3), (b, e, 3), (f) – visited {a,b}, unvisited {c,d,e,f}

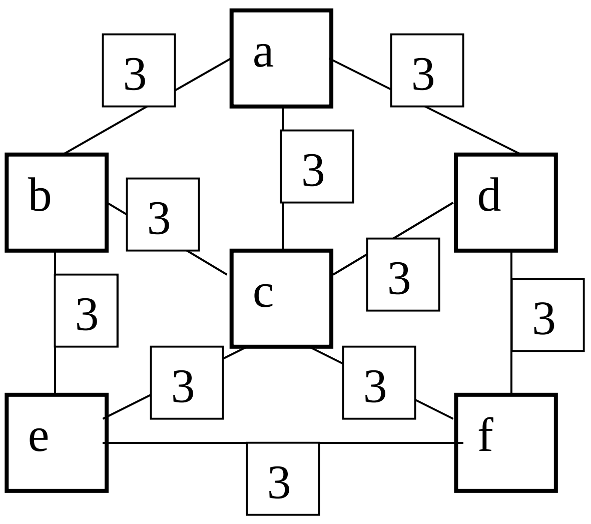
3. c is the nearest node
4. mark as visited - visited {a,b,c}, unvisited {d,e,f}
5. (c, d, 3), (c, e, 3) are NOT shorter but (c, f, 3) is – the component is now (a, b, 3), (a, c, 3), (a, d, 3), (b, e, 3), (c, f, 3) – visited {a,b,c}, unvisited {d,e,f}

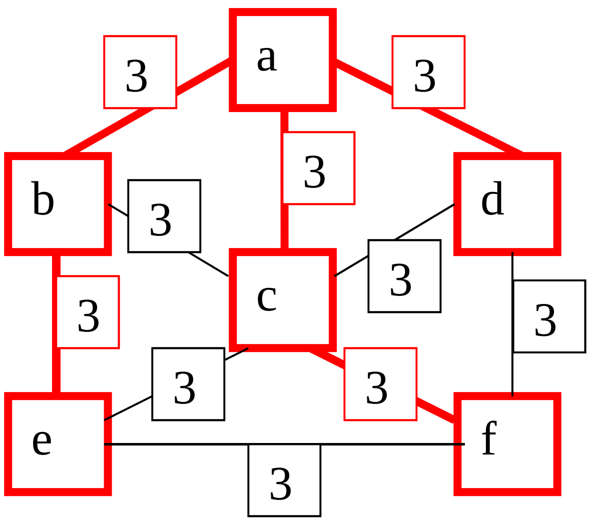
There are no changes for the remaining unvisited nodes d, e, f
MST - (a, b, 3), (a, c, 3), (a, d, 3), (b, e, 3), (c, f, 3) - six nodes / five edges.

See below for the calculations.

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

Cost 15



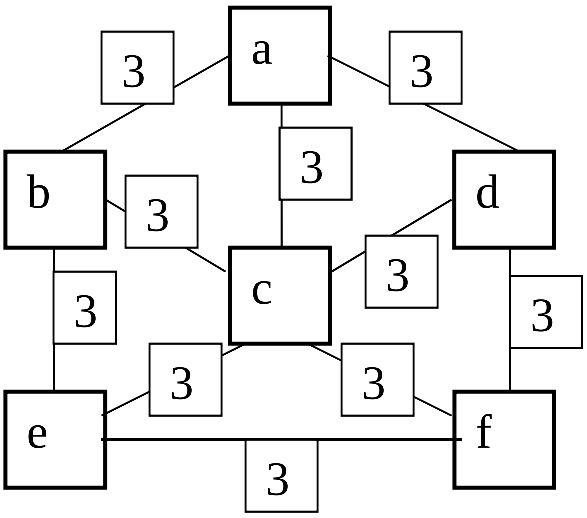


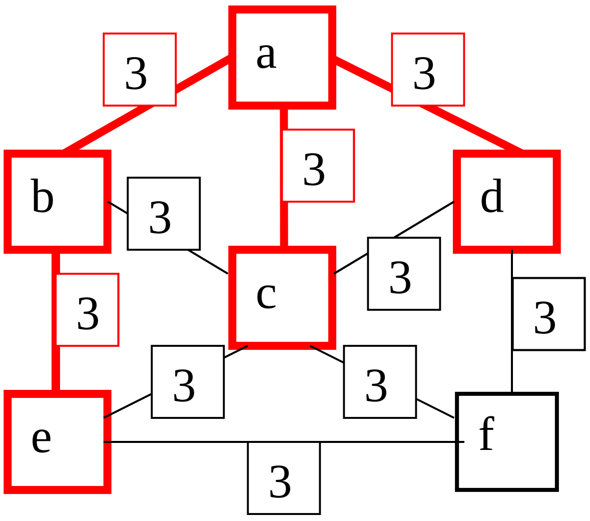
Draw the cost matrix C and arrays lowcost & closest

	a	**b**	**c**	**d**	**e**	**f**
a		**3**	**3**	**3**		
b	**3**		**3**		**3**	
c	**3**	**3**		**3**	**3**	**3**
d	**3**		**3**			**3**
e		**3**	**3**			**3**
f			**3**	**3**	**3**	

a	**b**	**c**	**d**	**e**	**f**
lowcost	**3**	**3**	**3**	**§**	**§**
closest	**a**	**a**	**a**	**a**	**a**

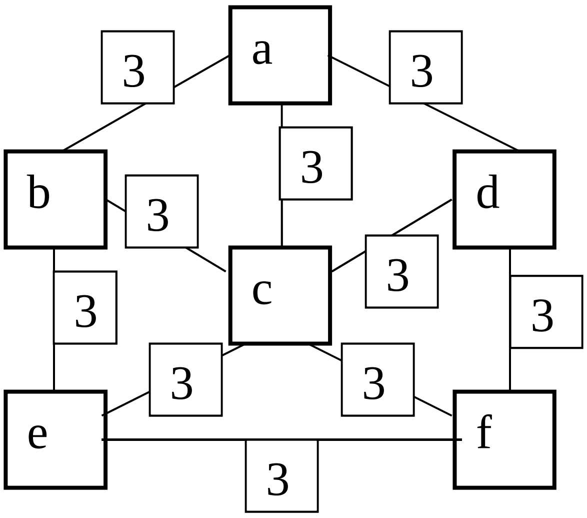
Min edge: **lowcost: 3 3 3 § § --- closest: a a a a a ---** U = {a,b} V-U = {c,d,e,f} min = 3; k = b
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = c; if C[b,c] < lowcost[c] then { lowcost[c] = C[b,c]; closest[c] = b } → 3<3 ➔ no change
j = d; if C[b,d] < lowcost[d]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

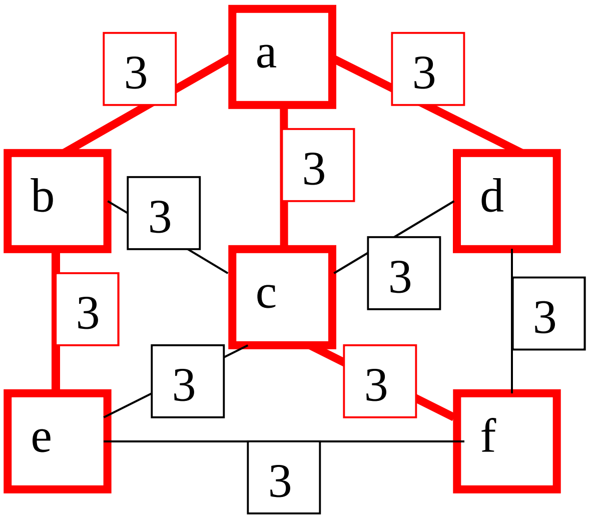




a	b	c	d	e	f
lowcost	**3**	**3**	**3**	**3**	**§**
closest	**a**	**a**	**a**	**b**	**a**

Min edge**: lowcost: 3 3 3 3 § --- closest: a a a b a ---** U = {a,b,c} V-U = {d,e,f} min = 3; 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[b] = c } → 3<3 → no change
j = e; if C[c,e] < lowcost[e] then { lowcost[e] = C[c,e]; closest[d] = c} → 3<3 → no change
j = f; if C[c,f] < lowcost[f] then { lowcost[f] = C[c,f]; **closest[e] = c** } → **3<§ → c-3-f**





a	b	c	d	e	f
lowcost	**3**	**3**	**3**	**3**	**3**
closest	**a**	**a**	**a**	**b**	**c**

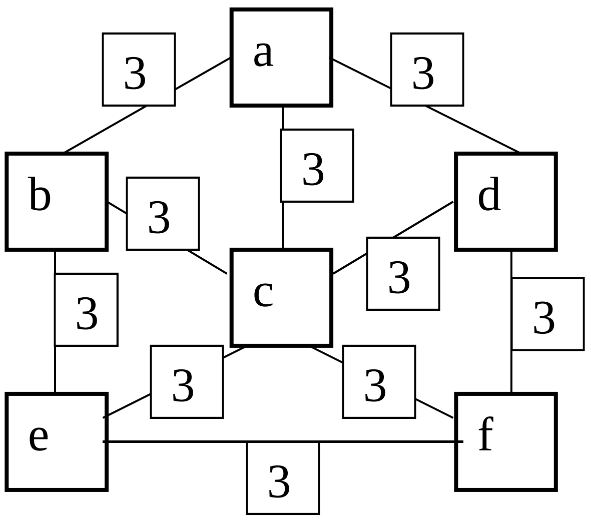
Min edge**: lowcost: 3 3 <u>3</u> 3 3 --- closest: a a <u>a</u> b c ---** U = {a,c,b,d} V-U = {e,f} min = 3; k = <u>d</u>
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = e; if C[d,e] < lowcost[e] then { lowcost[e] = C[d,e]; closest[e] = d } → §<3 → no change
j = f; if C[d,f] < lowcost[f] then { lowcost[f] = C[d,f]; closest[f] = d } → 3<3 ➔ no change

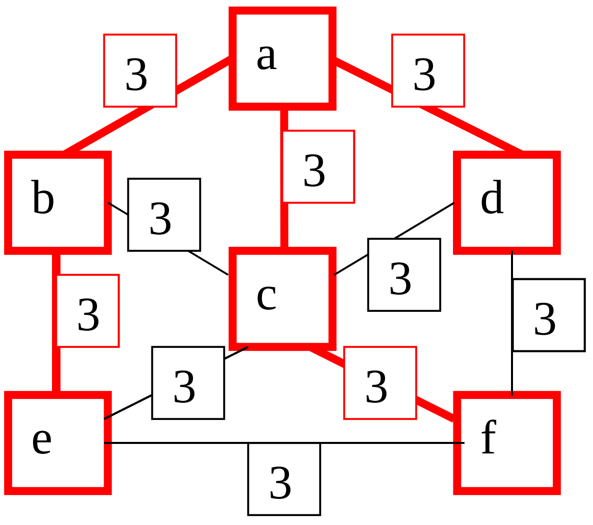
Min edge**: lowcost: 3 3 3 <u>3</u> 3 --- closest: a a a <u>b</u> c ---** U = {a,c,b,d,e} V-U = {f} min = 3; k = <u>e</u>
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = f; if C[e,f] < lowcost[f] then { lowcost[f] = C[e,f]; closest[f] = 3 } → 3<3 → no change

Finally add the remaining node – node e (there are no further calculations)
Min edge**: lowcost: 3 3 3 3 3 --- closest: a a a b c ---** U = {a,c,b,d,f e} V-U = {¤}

QED ☺ MST edges **a-3-b, a-3-c, a-3-d, b-3-e, c-3-f** **Total cost = 15**

(Confirm using Kruskal's – NB alphabetical order is important in this example)


a
b
c
d
e
f
3
3
3
3
3
3
3
3
3
3
3
3


a
b
c
d
e
f
3
3
3
3
3
3
3
3
3
3
3
3