FACIT TILL
OMTENTAMEN I
DATASTRUKTURER OCH ALGORITMER DAV B03

130819 kl. 08:15 – 13:15

Ansvarig Lärare: Donald F. Ross

Hjälpmedel: Bilaga A: Algoritmerna.

***** OBS *****

Ni som har läst från och med HT 2006

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

Ni som har läst tidigare än HT 2006

Betygsgräns:

Kurs:	Max 60p, Med beröm godkänd 50p, Icke utan beröm godkänd 40p, Godkänd 30p **(varav minimum 20p från tentamen, 10p från labbarna)**
Tentamen:	Max 40p, betyg 5: 34p-40p, betyg 4: 27p-33p, betyg 3: 20p-26p
Labbarna:	Max 20p, betyg 5: 18p-20p, betyg 4: 14p-17p, betyg 3: 10p-13p

SKRIV TYDLIGT – LÄS UPPGIFTERNA NOGGRANT

***** OBS *** Denna tentamen är kopierad på båda sidor *** OBS *****

---

(1) Ge ett kortfattat svar till följande uppgifter.

(a) Vad är ett ”free tree”?
N nodes connected by (N-1) undirected edges.

(b) Ge **ett exempel** av ett ”free tree”.
An MST (Minimal Spanning Tree)

(c) Ge **ett exempel** av hur man kan använda en riktade graf i verkligheten.
A course prerequisite graph (DAG) for university courses.
A transport map for Ryanair with ticket costs (A➔B) and (B➔A) journeys do not always have the same cost!

(d) I en graf, vad representerar kanterna?
A relationship between the node e.g. distance, cost, there exists a direct connection etc.

(e) Vad är nackdelen med hashning?
The collection is not sorted; requires linear search O(n) to find min/max.

(f) Vad är ”kvadratisk probning” inom hashning?
f(i) = i*i

(g) I stället för att mäta tid för sorterings algoritmer som vi har gjort i labb 2, föreslå ett annat sätt att mäta prestandet.
In sorts that use swaps, the number of swaps; in recursive algorithms, the number of calls.

(h) Varför är begreppet ”samling” (collection) viktigt inom datastrukturer och algorithmer?
It is a generalisation (abstraction) of set, sequence, tree, graph.Common operations may then be listed e.g. add, remove, find, cardinality, is_empty

(i) Vad är skillnaden mellan begreppen ”sorterad” och ”ordnad”?
Sorted ➔ the sequence is given in ascending (or descending) order of value.
Ordered is an inherent property of for e.g. a sequence (i.e. by position).

(j) Vad är fördelen med binära träd jämfört med generella träd?
A BT is easier to implement and there is a collection of known operations and algorithms which may be applied to a BT.

Totalt 5p

(2) Sekvens

Diskutera ingående följande påstående ”**sekvensen är den viktigaste ADT:en inom datavetenskap**”.

5p

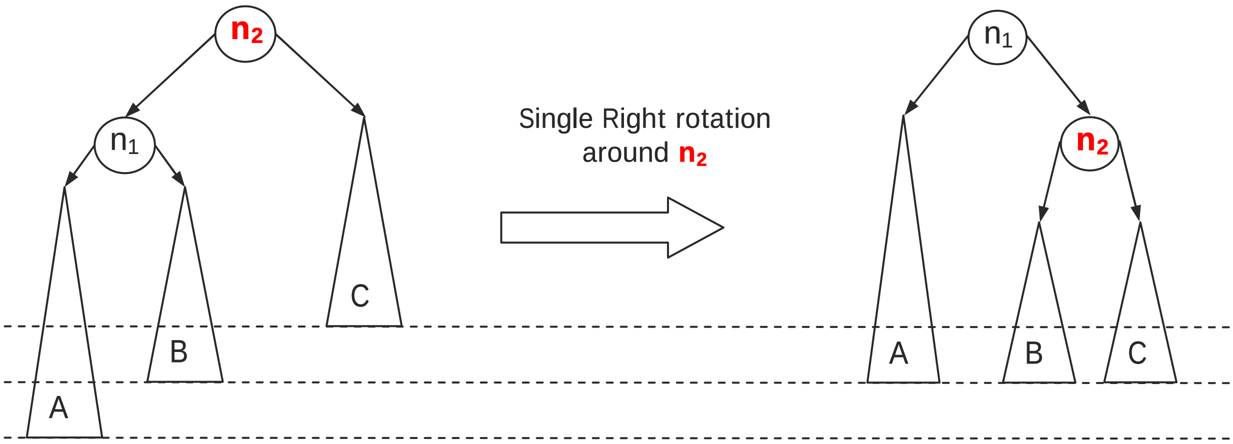
Marks for a good discussion and presentation of relevant points.

(3) Träd

Hur fungerar balansering i ett AVL-träd? . **Diskutera ingående.**
Ange gärna konkreta exempel. **Ange alla antagande.**

5p

- **Firstly, define and AVL tree: BST + balance restraint |Ht(LC(T)) – Ht(RC(T))| < 2**
- **The 2 kind of rotation used to rebalance the tree are**
  - **Single left/right rotations: SRR/SLR**
    - **imbalance is created on the outside of the tree**
  - **Double right/left rotations: DRR/DLR**
    - **Imbalance is created in the inside of the tree**
  - **The double rotations may be expressed in terms of 2 single rotations**
- **Rotations are sometimes expressed pictorially**



SRR may be written as **and its mirror image SLR as**

```
SRR(n2) {
  n1   = n2.L
  n2.L = n1.R
  n1.R = n2
  return n1
  }
```

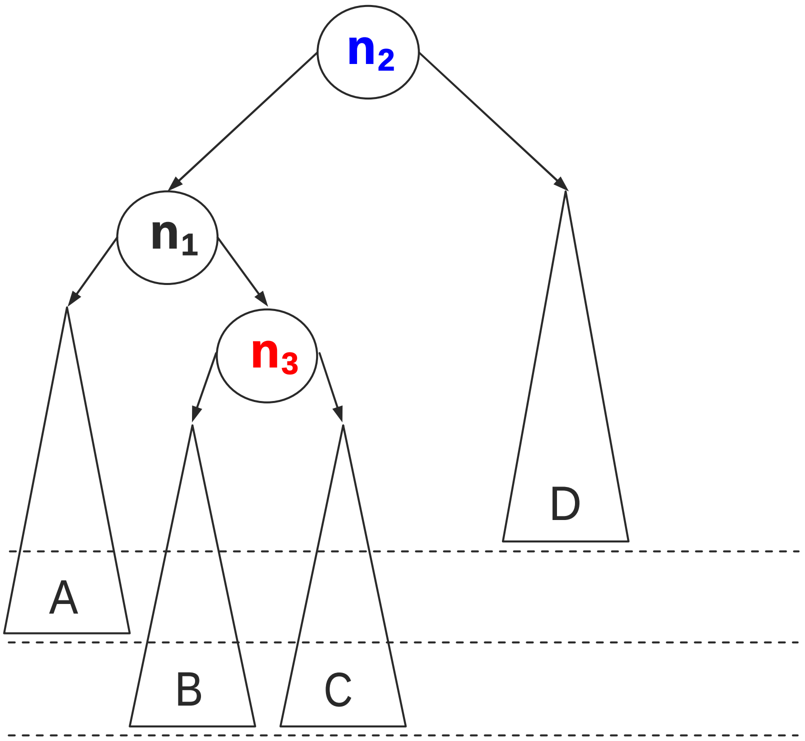
```
SLR(n2) {
  n1   = n2.R
  n2.R = n1.L
  n1.L = n2
  return n1
  }
```

DRR may be written as **and its mirror image DLR as**

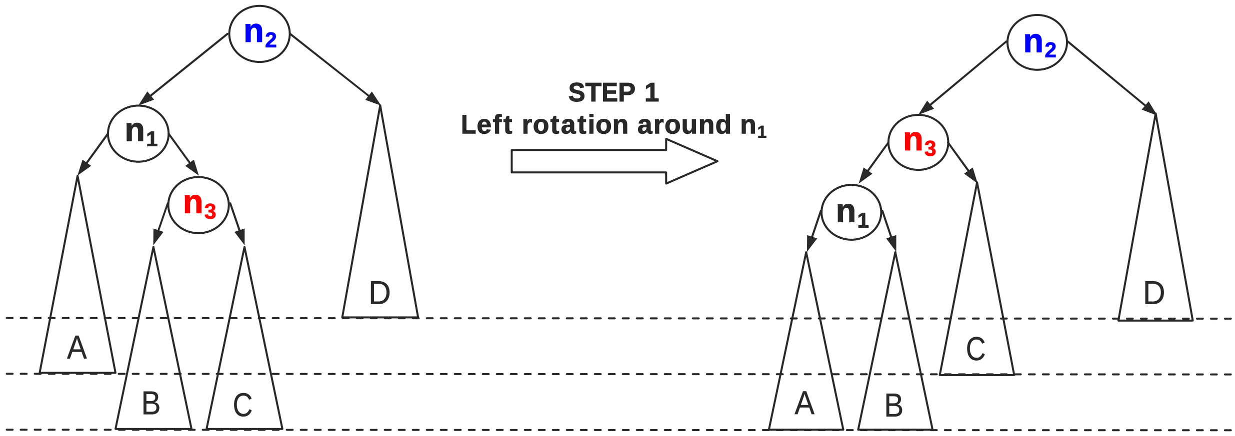
```
DRR(n2) {
  n2.L = SLR(n2.L)
  return SRR(n2)
  }
```

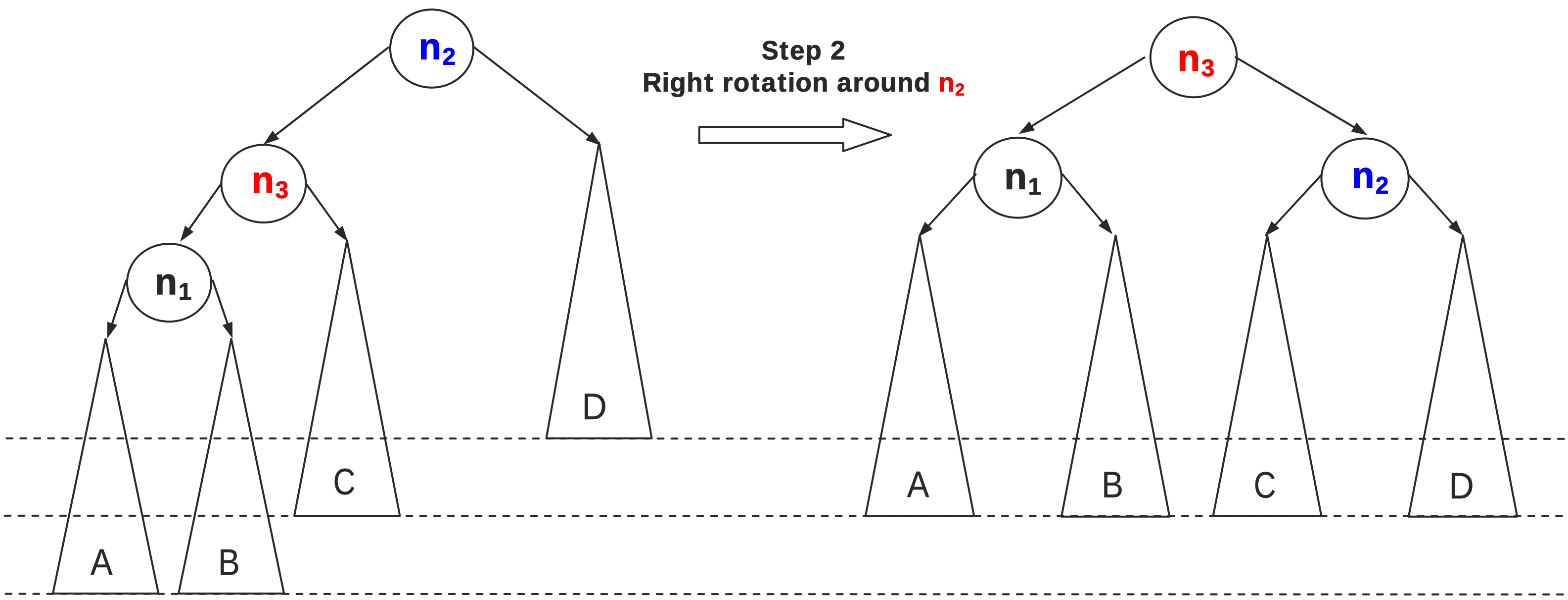
```
DLR(n2)
  n2.R = SRR(n2.R)
  return SLR(n2)
  }
```

See below for a DRR example




```
This may be expressed as
DRR(n2) {
  n2.L = SLR(n2.L)
  return SRR(n2)
  }
See the diagrams below
SLR(n2) {
  n1    = n2.R
  n2.R = n1.L
  n1.L = n2
  return n1
  }
SRR(n2) {
  n1    = n2.L
  n2.L = n1.R
  n1.R = n2
  return n1
  }
```





(4) Övriga frågor – Dijkstra kontra Prim - Se Bilaga A för algoritmerna.

(a) Förklara principerna bakom **Dijkstras algorithm**.

2p

Starting with the given node, Dijkstra's builds a single component by adding and removing edges such that at each stage, the path between the start node and the current node is the shortest (known) path at that moment in time. This is achieved by comparing the current shortest path to node w plus a jump to node v (w, v) with the currently known shortest path to node v (see algorithm in appendix A). The shorter of the two is chosen.

(b) Förklara principerna bakom **Prims algorithm**.

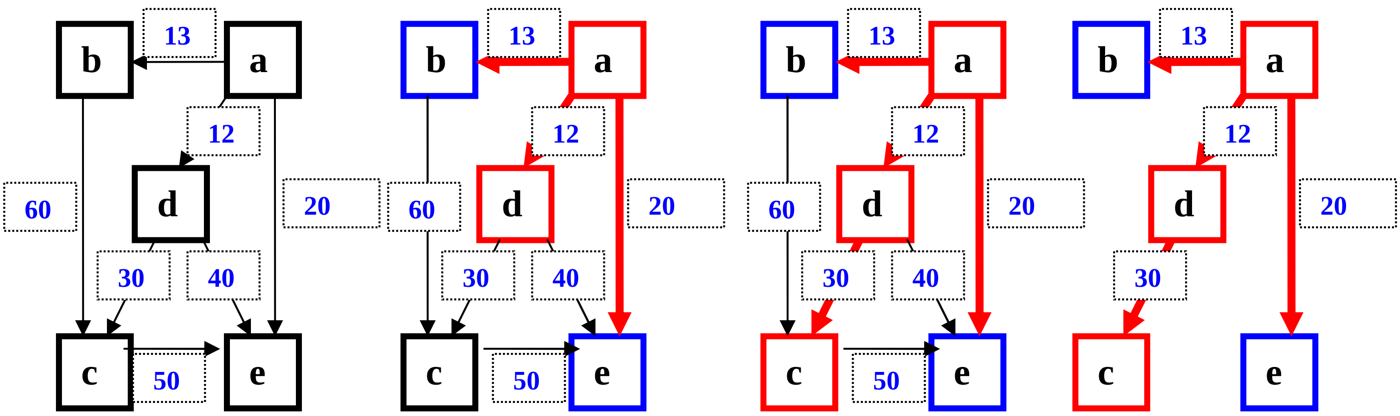
2p

- **From the start node (a) find the closest node (x) i.e. least valued edge to form a component a-cost-x; the cost of the edges from (a) to the remaining nodes are already known**
- **Compare the costs from the new node (x) to the remaining nodes (V-a-x) and if cheaper update the costs from a-x to (V-a-x).**
- **Choose the cheapest edge from x to (V-a-x) and add this node to the component**
- **Repeat this process until the MST has been found**

(c) Jämför **Dijkstras algoritm** med **Prims algoritm.** 1p

Totalt 5p

- **Both are greedy algorithms i.e. a "best choice" is made locally without reference to the overall result**
- **Both "grow" a component from a start node – Dijkstra's uses the path length in the selection; Prim's uses the edge length**
- **See the answer to question 6 for diagrams of how the component grows for Prims**
- **See below for an example of Dijkstra**



(5) Labbkod

(a) I graflabben har en student skrivit följande kod för att lägga till 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 adde(char cs, char cd, int v) {
  set_edges(b_findn(cs, G), b_adde(cd, v, get_edges(b_findn(cs, G))));
  }
```

2p

Assumptions: (i) G is a reference to the graph, (ii) the graph is represented as an adjacency list (AL) (draw this!) (iii) (cs, cd) define the edge. Working from the inside out (functional thinking) b_findn(cs, G) gives a reference to the node in the AL; get_edges(N) then gives a reference to the EDGE LIST for this node and b_adde(e, Elist) adds cd to this edge list and returns a (new) reference to the edge list which is "reconnected" to the edge list of the node cs by set_edges(N, Elist)

(b) **Beskriv ingående** hur du skulle **testa** träd labben (BST samt komplett träd). Vilka operationer har man på ett träd och hur skulle Du testa varje operation?

3p

Totalt 5p

The main operations are add, remove and find followed by cardinality, height and display (in-order, pre-order, post-order and 2D and Q).

- **Firstly, apply remove, find, cardinality, height and display to the EMPTY TREE.**
- **Then add to the empty tree, add a low, high and middle value (not in the tree) to test the boundary values**
- **Then remove a non-existent value, the lowest value, the highest value and a middle value followed by the removal of the remaining values until the tree is empty.**
- **Add more values to the tree**
- **Then find a non-existent value, the lowest value, the highest value and a middle value.**
- **Check at each stage with add/remove that the cardinality and height operations work**
- **Check display after each operation above**

(6) **Graf - Prims algorithm**

Se Bilaga A för algoritmerna.

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

(a-6-b, a-3-c, a-7-d, b-1-c, b-12-e, c-4-d, c-5-e, c-9-f, d-3-f, e-8-f).

Börja med nod "a".

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

(3p)

Rita en bild av varje steg under algoritmens exekvering för att förklara processen.

(2p)

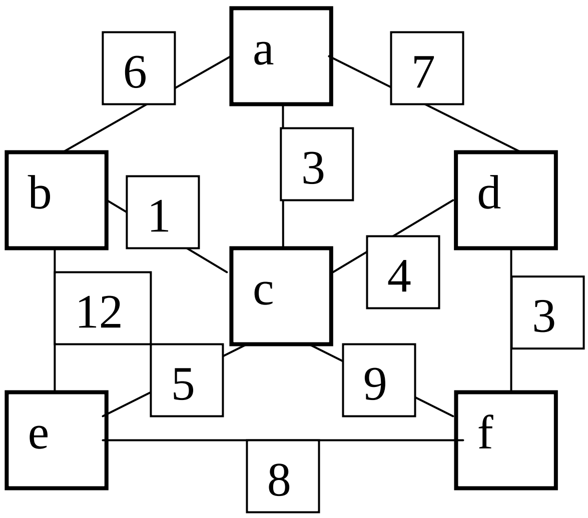
Totalt 5p

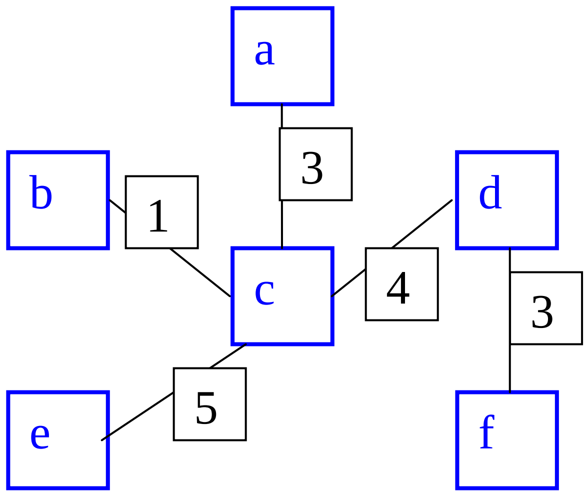
The principle is that the MST " grows" from the one component (here "a") by connecting this component to any other component (a node) by the cheapest edge SO FAR found – this last proviso reveals that Prim's is a GREEDY algorithm i.e. used a local best solution.

See below for the calculations.

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

Cost 16





Draw the cost matrix C and array D

	a	b	c	d	e	f
a		**6**	**3**	**7**		
b	**6**		**1**		**12**	
c	**3**	**1**		**4**	**5**	**9**
d	**7**		**4**			**3**
e		**12**	**5**			**8**
f			**9**	**3**	**8**	

a	b	c	d	e	f
lowcost	**6**	**3**	**7**	**§**	**§**
closest	**a**	**a**	**a**	**a**	**a**

Min edge: **lowcost: 6 3 7 § § --- closest: a a a a a ---** U = {a,c} V-U = {b,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 = b; if C[c,b] < lowcost[b] then { lowcost[b] = C[c,b]; closest[b] = c } → 1<6 ➔ **c-1-b**
j = d; if C[c,d] < lowcost[d] then { lowcost[d] = C[c,d]; closest[d] = c } → 4<7 ➔ **c-4-d**
j = e; if C[c,e] < lowcost[e] then { lowcost[e] = C[c,e]; closest[e] = c } → 5<§ ➔ **c-5-e**
j = f; if C[c,f] < lowcost[f] then { lowcost[f] = C[c,f]; closest[f] = c } → 9<§ ➔ **c-9-f**

Min edge: **lowcost: 1 3 4 5 9 --- closest: c a c c c ---** U = {a,c,b} V-U = {d,e,f} min = 1; 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[b] = b } → §<4 → no change
j = e; if C[b,e] < lowcost[e] then { lowcost[e] = C[b,e]; closest[d] = b} → 12<5 → no change
j = f; if C[b,f] < lowcost[f] then { lowcost[f] = C[b,f]; closest[e] = b } → §<9 → no change

Min edge: **lowcost: 1 3 4 5 9 --- closest: c a c c c ---** U = {a,c,b,d} V-U = {e,f} min = 4; k = d
j = e; if C[d,e] < lowcost[b] then { lowcost[e] = C[d,e]; closest[e] = d } → 5<4 → no change
j = f; if C[d,f] < lowcost[e] then { lowcost[f] = C[d,f]; closest[f] = d } → 3<9 ➔ **d-3-f**

Min edge: **lowcost: 1 3 4 5 3 --- closest: c a c c d ---** U = {a,c,b,d,f} V-U = {e} min = 3; k = f
j = e; if C[f,e] < lowcost[e] then { lowcost[e] = C[f,e]; closest[e] = f } → 8<5 → no change

Finally add the remaining node – node e (there are no further calculations)
Min edge: **lowcost: 1 3 4 5 3 --- closest: c a c c d ---** U = {a,c,b,d,f e} V-U = {¤}

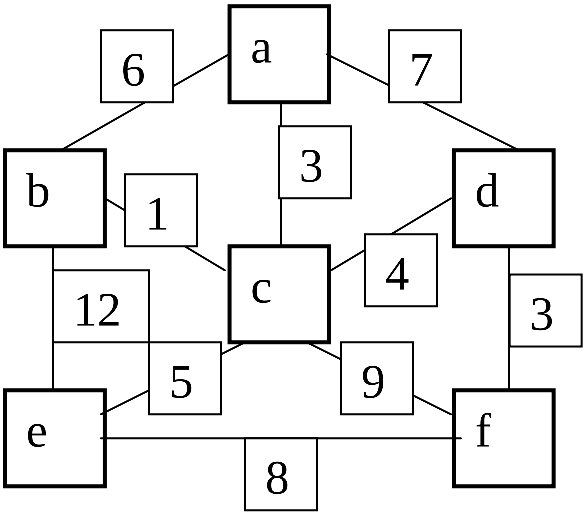
QED ☺ MST edges **c-1-b, a-3-c, c-4-d, c-5-e, d-3-f** **Total cost = 16**

(Confirm using Kruskal's)

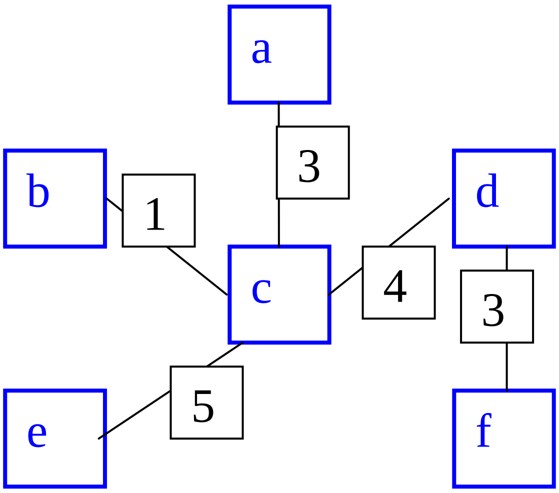
Principle: to build the MST from a single component by choosing the cheapest **edge** to non-component nodes from the last node added. Above start with a, add **edge** distances (infinite if no edge), choose the cheapest (a-3-c) and add this to the component. Now recheck if there are cheaper **edges** from c to the non-component nodes. Repeat until all the nodes are connected. So the component develops as (a), (a-3-c), (a-3-c, c-b-1, c-4-d, c-5-e, d-3-f) (see above).

See below for the example in the question.

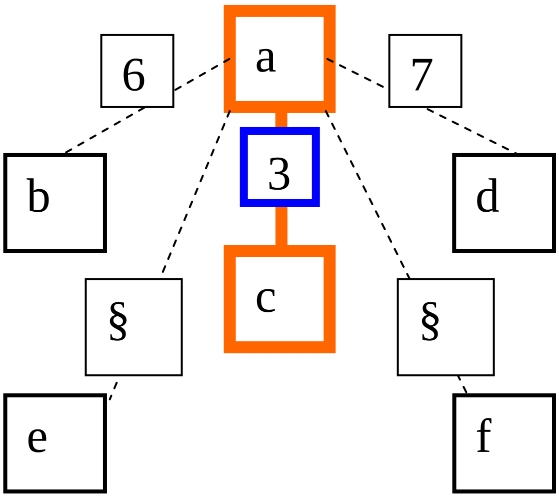
Start Graph

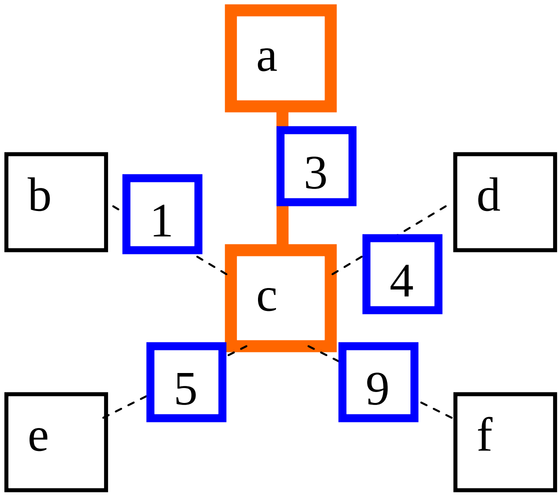


Solution



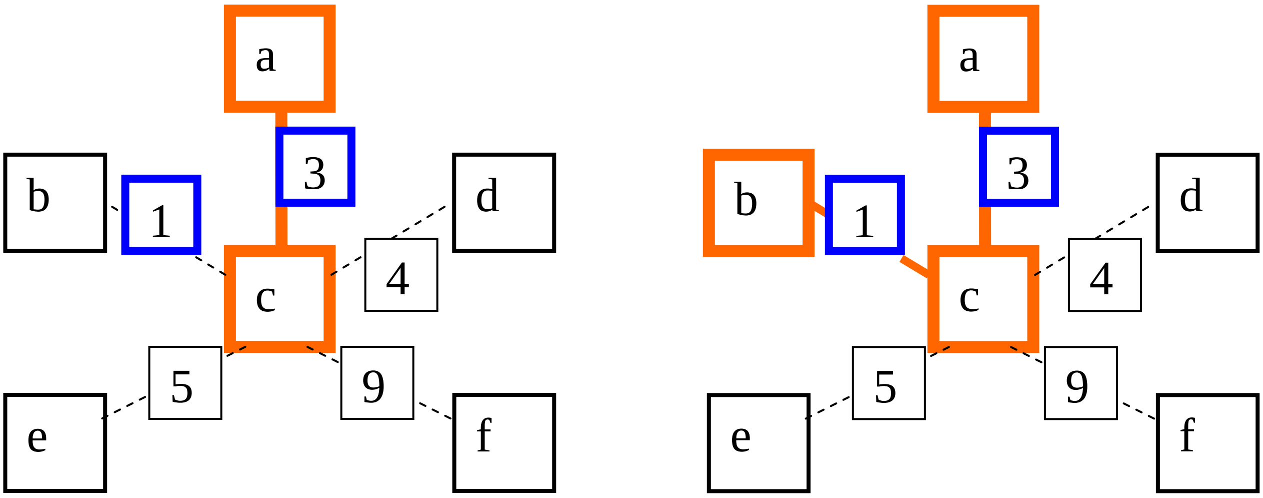
Start with node a – a is the component
Calculate the edges distances from a to the remaining nodes: no edge=infinity (§)
Calculate the shortest edge (lines 3-4 of the algorithm) – a-3-c
Add node c to the component (line 7 of the algorithm)
Recalculate the edge distances from c to (b,d,e,f) to see if there is a shorter edge than so far calculated (line 8 of the algorithm)





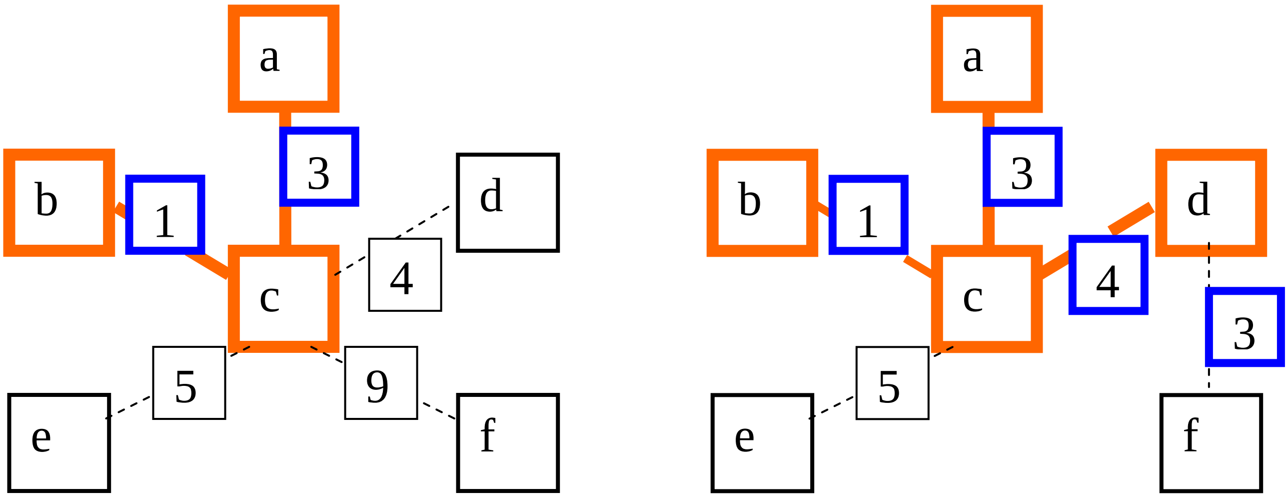
Min edge: **lowcost: 6 3 7 § § --- closest: a a a a a ---** U = {a,c} V-U = {b,d,e,f} min = 3; k = c
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = b; if C[c,b] < lowcost[b] then { lowcost[b] = C[c,b]; closest[b] = c } → 1<6 ➔ **c-1-b**
j = d; if C[c,d] < lowcost[d] then { lowcost[d] = C[c,d]; closest[d] = c } → 4<7 ➔ **c-4-d**
j = e; if C[c,e] < lowcost[e] then { lowcost[e] = C[c,e]; closest[e] = c } → 5<§ ➔ **c-5-e**
j = f; if C[c,f] < lowcost[f] then { lowcost[f] = C[c,f]; closest[f] = c } → 9<§ ➔ **c-9-f**
Result after this iteration: lowcost: 1 3 4 5 9 --- closest: c a c c c

Continue with node c – c is the new node in the component
Calculate the shortest edge (lines 3-4 of the algorithm) – c-1-b
Add node b to the component (line 7 of the algorithm)
Recalculate the edge distances from b to (d,e,f) to see if there is a shorter edge than so far calculated (line 8 of the algorithm)



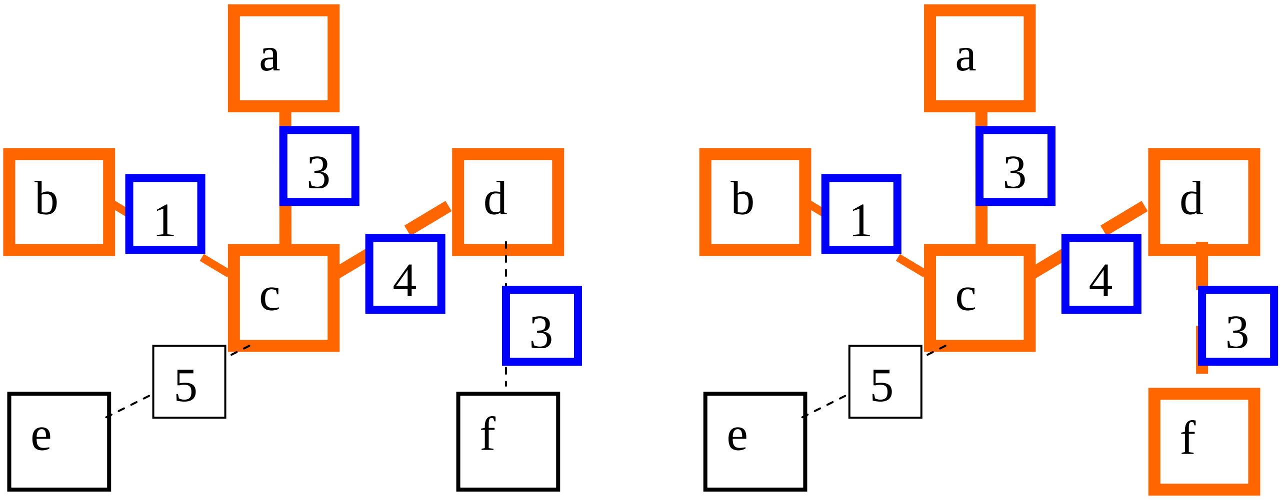
Min edge**: lowcost: 1 3 4 5 9 --- closest: c a c c c ---** U = {a,c,b} V-U = {d,e,f} min = 1; k = b
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = d; if C[b,d] < lowcost[d] then { lowcost[d] = C[b,d]; closest[b] = b } → §<4 → no change
j = e; if C[b,e] < lowcost[e] then { lowcost[e] = C[b,e]; closest[d] = b} → 12<5 → no change
j = f; if C[b,f] < lowcost[f] then { lowcost[f] = C[b,f]; closest[e] = b } → §<9 → no change
Result after this iteration: lowcost: 1 3 4 5 9 --- closest: c a c c c – NO CHANGE

Now look for the closest non-component node to the component – node d
Calculate the shortest edge (lines 3-4 of the algorithm) – c-4-d
Add node d to the component (line 7 of the algorithm)
Recalculate the edge distances from d to (e,f) to see if there is a shorter edge than so far calculated (line 8 of the algorithm)



Min edge**: lowcost: 1 3 4 5 9 --- closest: c a c c c ---** U = {a,c,b,d} V-U = {e,f} min = 4; k = d
j = e; if C[d,e] < lowcost[e] then { lowcost[e] = C[d,e]; closest[e] = d } → 5<4 → no change
j = f; if C[d,f] < lowcost[f] then { lowcost[f] = C[d,f]; closest[f] = d } → 3<9 ➔ **d-3-f**
Result after this iteration: lowcost: 1 3 4 5 3 --- closest: c a c c d

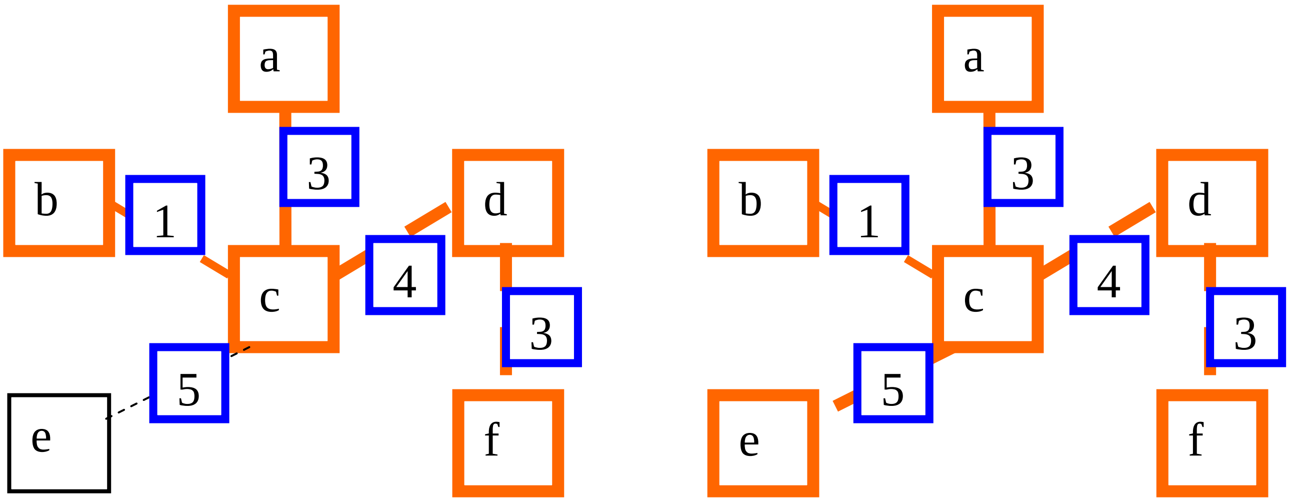
Now look for the closest non-component node to the component – node f
Calculate the shortest edge (lines 3-4 of the algorithm) – d-3-f
Add node f to the component (line 7 of the algorithm)
Recalculate the edge distances from f to (e) to see if there is a shorter edge than so far calculated (line 8 of the algorithm)



Min edge**: lowcost: 1 3 4 5 <u>3</u> --- closest: c a c c <u>d</u> ---** U = {a,c,b,d,f} V-U = {e} min = 3; k = <u>f</u>
j = e; if C[f,e] < lowcost[e] then { lowcost[e] = C[f,e]; closest[e] = f } → 8<5 → no change
Result after this iteration: lowcost: 1 3 4 5 <u>3</u> --- closest: c a c c <u>d</u>

Now look for the closest non-component node to the component – node e
Calculate the shortest edge (lines 3-4 of the algorithm) – c-5-e
Add node e to the component (line 7 of the algorithm)
Recalculate the edge distances from e to (¤) to see if there is a shorter edge than so far calculated (line 8 of the algorithm) – no edges

Result after this iteration: lowcost: 1 3 4 5 3 --- closest: c a c c d



Bilaga A: Algoritmerna

```
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; }
  }
}
```

```
Dijkstra ( a ) -- a is the start node
{
    S = {a}
    for (i in V-S) D[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 ) D[v] = min(D[v], D[w]+C[w,v])
      }
    }
```