FACIT TILL
OMTENTAMEN I
DATASTRUKTURER OCH ALGORITMER DVG B03

150609 kl. 14:15 – 19:15

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.

Studenter som har läst kursen from HT2014 ska svara på samtliga uppgifter.

Studenter som har läst kursen from HT2006 tom HT2013 ska välja uppgifter värt 30 poäng.

Studenter som har läst kursen tom HT2005 ska svara på samtliga uppgifter.

(1) Ge en kortfattad definition av följande begrepp ((a)-(j)).

(a) **Mängd –** a collection of elements in which no duplicate values are allowed and in which there is no ordering.

(b) **Sekvens –** a collection of elements in which duplicates may be allowed and in which the relationships predecessor and successor are defined; i.e. linear ordering is defined

(c) **Träd -** a collection of elements in which duplicates may be allowed and in which a parent-child relationship (i.e.a hierarchy) is defined. Each parent may have n children except at the lowest level (leaf nodes) and each child has exactly one parent (except for the root node). May be onordered or ordered.

(d) **Binärt träd** – an ordered tree i.e. there is a left child and a right child, with the restriction that each parent may have at most 2 children

(e) **AVL-träd –** a binary search tree with the restriction that the tree is ordered i.e. there is a left child and a right child and that the heights of each left and right subtree may not differ by more that 1

(f) **Graf –** a pair of sets G = (V, E) where E is a set of nodes and V a set of edges where each of the edges connects nodes a and b in V.

(g) **Enkel cykel i en graf** – a simple path where the start and end nodes are the same. A simple path is a path where the edges and vertices are distinct. Directed graph (A → B → A) undirected graph (A → B → C → A)

(h) **Artikuleringspunkt** (articulation point) – a node, which when removed breaks the graph into 2 or more components.

(i) **Prestande** (performance) – the decription of the relationship between the number of entities handled (for example in a sort) and the time and/or space required

(j) **Big-oh (O(n))** – an upper limit on the time required by a given algorithm to handle n entities

Totalt 5p

(2) Rekursion

Beskriv **ingående** varför rekursion är så pass viktigt inom datastrukturer och algoritmer. Använd gärna **exempel.** Ge flera exempel där man använder sig av rekursion.

Totalt 5p

Importance

(i) Recursion is a very succinct (shorthand) way of defining an entity or writing programming language code
(ii) Non-recursive code solutions tend in certain cases to be longer and more complicated than recursive solutions
(iii) Recursion changes your mindset i.e. your way of thinking about programming constructs and writing code
(iv) Recursion is also useful in defining data structures (eg a sequence or a tree) and definitions used in Computer Science (eg defining the production rules P in a formal grammar (G = (S, P, NT, T)) definition of a programming language

Examples of definitions

(v) Sequence (see above) **Seq** ::= Head Tail | empty; Head ::= element; Tail ::= **Seq**
(vi) Binary Tree: **BT** ::= LC Node RC | empty; Node ::= element; LC ::= **BT**; RC ::= **BT**
(vii) Grammar definition **[id list] ::= id | [id list] , id** i.e. the **id list** is **id, id, id, …., id**

Examples of programming operations on the above

(viii) Sequence display

Recursive version

```
List: display_L (Reftype List)
{
  if not is_empty (List) {
     display_el ( head (List) );
     display_L ( tail (List) );
     }
  return List;
  }
```

Iterative version

```
List: display_L (Reftype List)
{
  Reftype ref;

  ref = first (List); (last)
  while (not_null (ref)) {
      display_el ( ref );
      ref = next ( ref );
      }
  return List;
  }
```

(ix) Binary tree – number of elements

Recursive

```
Int: size (BT)
{
  if is_empty(BT) return 0;
  return 1 + size(left(BT)) + size(right(BT));
  }
```

Non-recursive – would be more difficult!

(x) Id list

```
Non-recursive:
id_list() {  match(id); while (lookahead == ','){ match(','); match(id); } }

Recursive
id_list() {  match(id); if (lookahead == ',') { match(','); id_list(); } }
```

Marks also for good arguments and examples.

(3) Heap

(a) Läs koden till heap operationer (**se Bilaga A**), Svara på följande uppgifter:-

i. Vad är parameter ”i” i **Heapify**(A, i)?

i is the index in the array representation of the heap

ii. Skriv (pseudo) kod till funktionen **Right**(i)

```
Left(i)   {  return  2*i;      }  // assumption index starts at 1
Right(i)  {  return 2*i + 1; }    // assumption index starts at 1

Left(i)   {  return  2*(i+1)-1; } // assumption index starts at 0
Right(i)  {  return 2*(i+1); }    // assumption index starts at 0
```

iii. I **Heapify**, vad gör koden nedan? Vad betyder koden?

```
if l <= A.size and A[l] > A[i] then largest = l else largest = i
if r <= A.size and A[r] > A[largest] then largest = r
```

- **if the left child exists and the value of the left child is greater than the value in the parent then set largest to the left child, otherwise set largest to the parent**
- **if the right child exists and the value of the right child is greater than the value of largest then set largest to r**
- **this code compares the value of the parent with the values in the left and right children (if they exist)**

iv. I Heapify, vad gör koden nedan? Vad betyder koden?

```
if largest != i then
    swap(A[i], A[largest])
    Heapify(A, largest)
end if
```

- **If the value in the parent is not the largest (of P, LC, RC) i.e. largest != i, then swap the values of largest and i**
  Now heapify the (sub-)tree with parent at largest

v. Varför skriver man **A.size/2** i funktion Build?

- **This finds the last parent in the heap with 1 child or 2 children.**
- **This works since the heap is a complete tree and children are always added at the next node position on the lowest level.**

vi. Hur fungerar funktion **Remove**?

- **Remove swaps the value of the node at index r with the last node in the array**
- **The size of the array is reduced by 1, hence removing the value that was at index r**
- **The (sub-)tree is then re-Heapified**

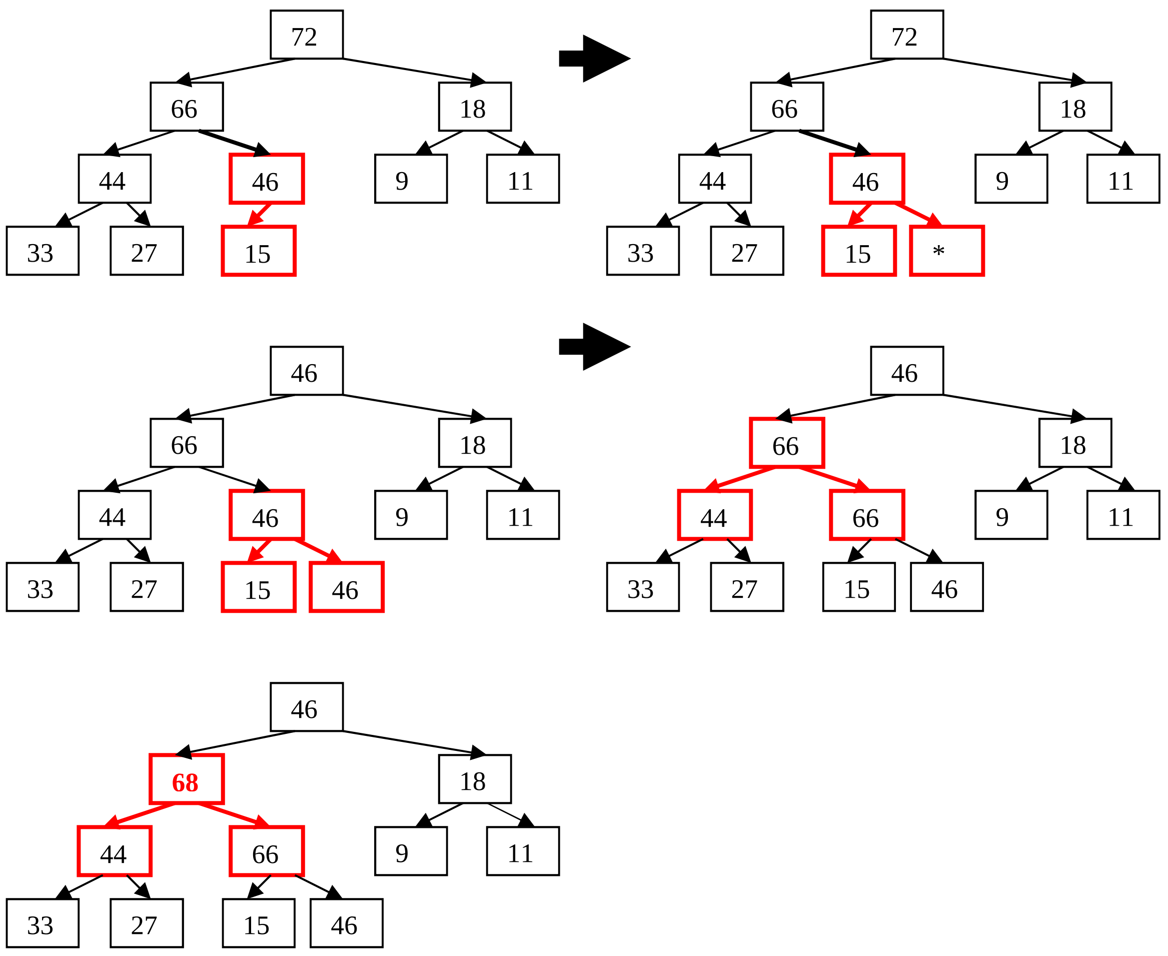
3p

(b) Givit heapen: **72, 66, 18, 44, 46, 9, 11, 33, 27, 15**
Förklara **stegvis** hur funktionen **Add** fungerar när man lägger till värdet **68** i heapen.
Rita en bild av den **trädrepresentationen** av heapen **vid varje steg**.

2p

Totalt 5p

The tree version of the heap and add 68 process is



The explanation is:-

1. Increase the size of the array by 1
2. Set i to be the index of the last node
3. Bottom up, if the value in the parent of i is less than the value to be added to the heap then **copy this value to the node given by i (NB)**
4. Set i to reference its parent (moving up the tree)
5. Repeat the process until either i refers to the root of the heap i.e. the first element in the heap and the value in the parent of i is less than the value to be added
6. **Set the value of A[i] to the new values to be added – this is the add stage – not before!**

1. In the example above, the size of the heap is 10 (10 elements)
2. The value to be added is 68
3. An extra node is ”added” at position 11
4. Set i to position 11
5. The parent is position 5
6. Value 46 in position 5 is less than 68 → copy 46 to position 11
7. Set i to position 5
8. The parent is position 2
9. Value 66 in position 2 is less than 68 → copy 66 to position 5
10. Set i to position 2
11. The parent is position 1
12. 72 is NOT less than 68 → exit the while loop
13. Copy the new value 68 to position 2

(4) Binärt Träd

(a) Ge en rekursiv definition av ett binärt träd.

1p

```
BT      ::= LC Node RC | empty
Node    ::= element
LC      ::= BT
RC      ::= BT
```

(b) Utifrån din definition i (a) skriv pseudokod till en **rekursiv version** av den lägga-till funktionen (add) för ett binärt träd. Anta att dubbleter inte tillåtas.

2p

If you assume a BST then the answer is as below.
Alternatives: that a find finction is applied as a precondition (except that this would be redundant for the BST)

```
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;
}
```

If you assume a BT then the problem becomes a little more difficult.
You have to make assumptions

1. That there is a find function available as a prcondition
2. Where in the BT are you going to add?
3. Do you add to the smaller tree of the LC and RC?

There is no general solution for the BT case in the same way as the BST above.

(c) Att ta bort ett element från ett binärt träd kan reduceras till ett annat problem, nämligen att ta bort roten från ett binärt träd. Då finns det fyra möjligheter. Vilka är dessa? **Diskutera** utförligt vad händer i varje fall.

2p

1. The root consists of a node only.
   Removing the node (root) ➔ return nil

2. The root consists of a node with a left child only
   Removing the node (root) ➔ return left sub-tree

3. The root consists of a node with a right child only
   Removing the node (root) ➔ return right sub-tree

4. The root consists of a node with both a left and right child
   This is the most interesting case – see below

Case 4: if you assume a BST then
The value in the root may be replaced by
(i) The biggesst value in the left child
(ii) The smallest value in the right child

This value must then be removed from the corresponding child.

If You take the BT case then vou can add the LC to the RC or vice-versa with an add function, but again you have to choose where to add the sub-tree

Totalt 5p

(5) AVL-Träd

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

2p

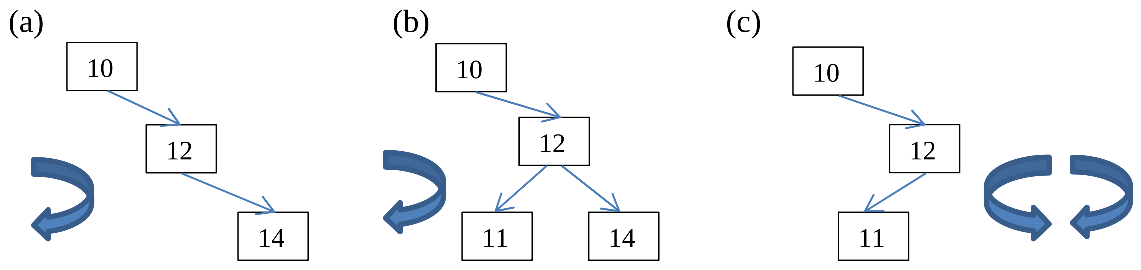
An AVL-tree is a BST (Binary Search Tree) where the balance difference (balance factor) between the height of the left child and the height of the right child may be at most 1.

i.e. | height(LC) – height(RC) | <= 1

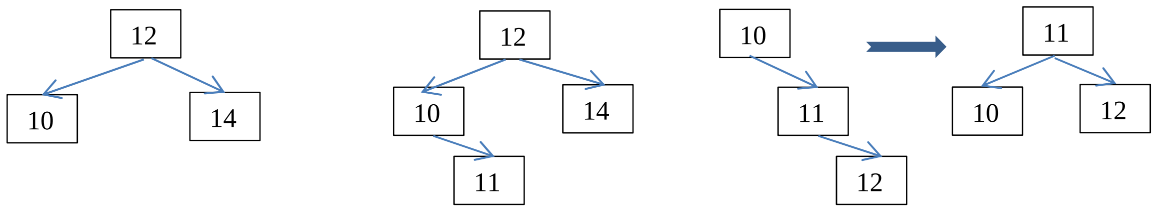
The rotation functions are

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

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. <u>**Skriv balansfunktionen i pseudokod.**</u>

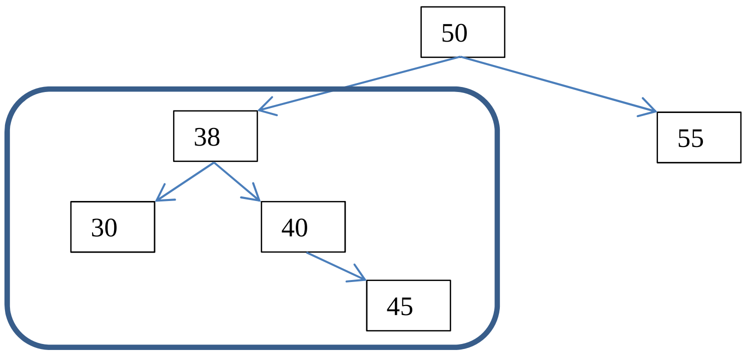
Define the balance factor br = height(LC(T) – height(RC(T))

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).

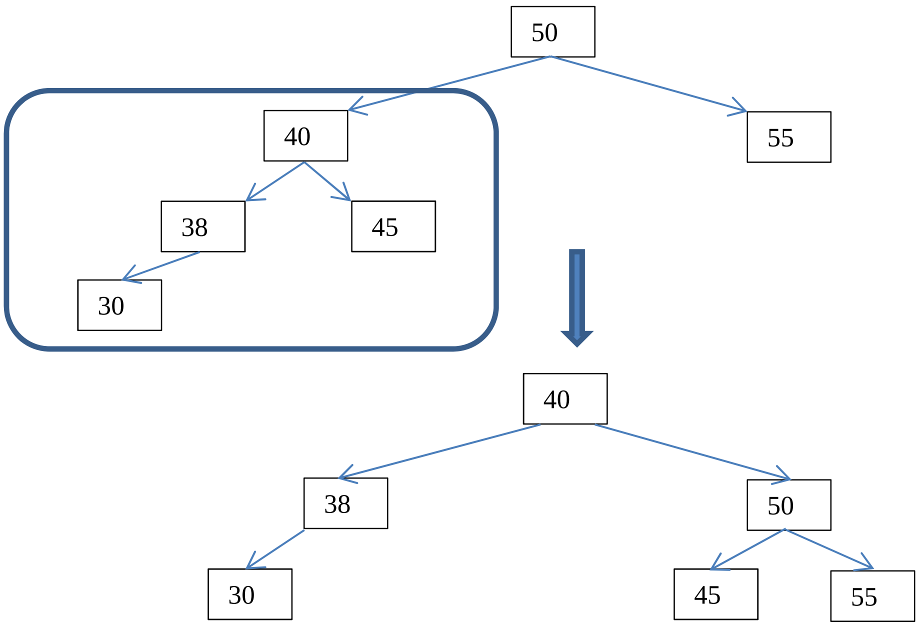
Förklara vad ”balansfaktorn” (balance factor) är och använd denna för att ta fram balansfunktionen från första principer. **Skriv balansfunktionen i pseudokod.** Tillämpa Din pseudokod på trädet nedan

3p



From the diagram, the left child is clearly higher than the right child
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.
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;

(6) Rekursiva funktioner

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));
  }
```

(7) Prim

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).

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; }
    }
}
```

Börja med nod "a".
Visa varje steg i dina beräkningar.
Ange *alla* antaganden och visa *alla* beräkningar och mellanresultat

(3p)

Visa att man har kollat på varje kant under algoritmens exekvering.
Använd bilder. I vilka ordning har algoritmen kollat på kanterna?
Vilka kanter **ersätter** andra kanter under algoritmensexekvering?

(2p)

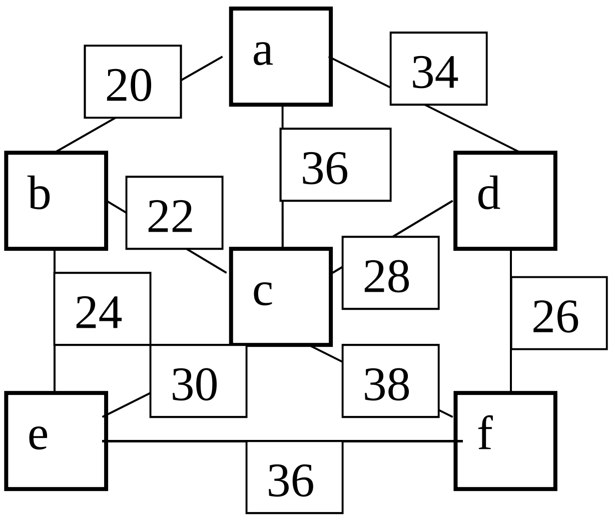
Totalt 5p

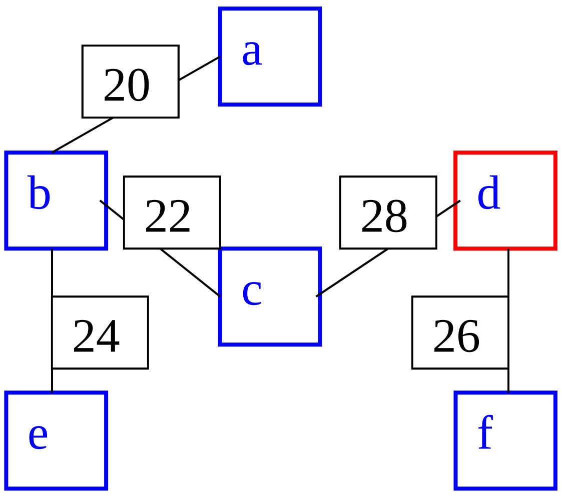
The principle is that the MST " grows" from the **one component** (here "a") by connecting this component to any other component (a node) by the shortest **EDGE SO FAR FOUND** – this last proviso reveals that Prim's is a GREEDY algorithm i.e. used a local best solution.

See below for the calculations.

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

Cost 120


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


a
b
c
d
e
f
20
22
28
24
26

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: <u>20</u> 36 34 § § closest: <u>a</u> a a a a** U = {a,b} V-U = {c,d,e,f} min = **20**; k = <u>**b**</u>
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 ➔ **b-22-c**
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 <u>22</u> 34 24 § closest: a <u>b</u> a b a** U = {a,b,c} V-U = {d,e,f} min = **22**; k = <u>c</u>
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 <u>24</u> 38 closest: a b c b c** U = {a,b,c,e} V-U = {d,f} min = **24**; k = <u>**e**</u>
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 <u>28</u> 24 36 closest: a b c b e** U = {a,b,c,e,d} V-U = {f} min = **28**; k = <u>**d**</u>
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 = {¤}

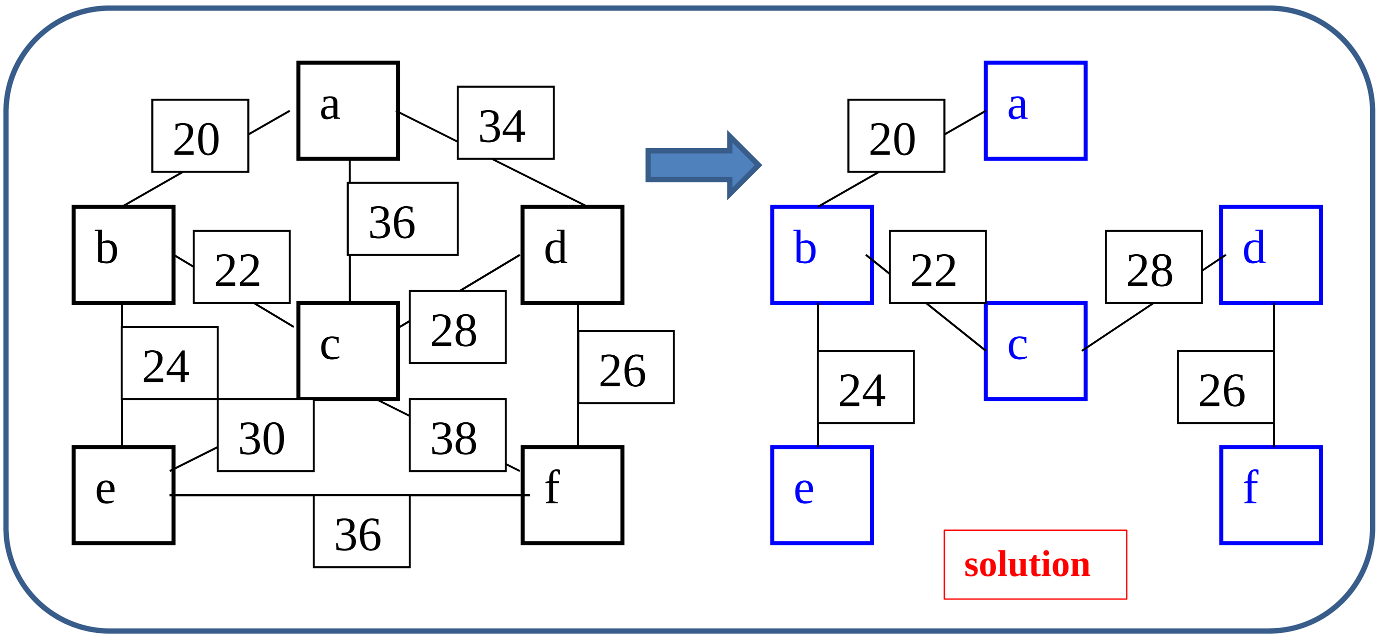
QED ☺ MST edges **a-20-b, b-22-c, c-28-d, b-24-e, d-26-f Total cost = 120**

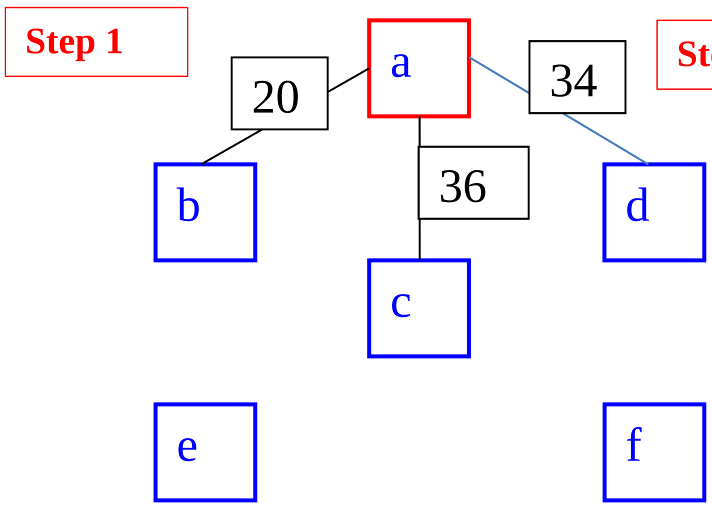
(Confirm using Kruskal's)

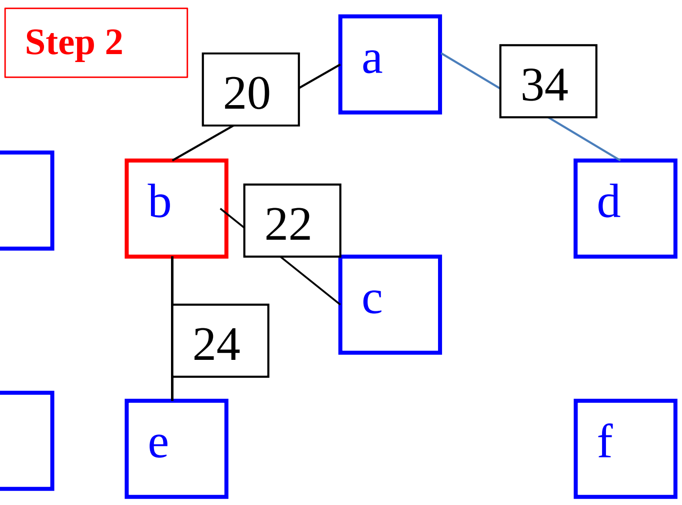
The edges are examined in the following order – see the steps above.
Initialisation: **(a,b,20)**, (a,c,36), (a,d,34)

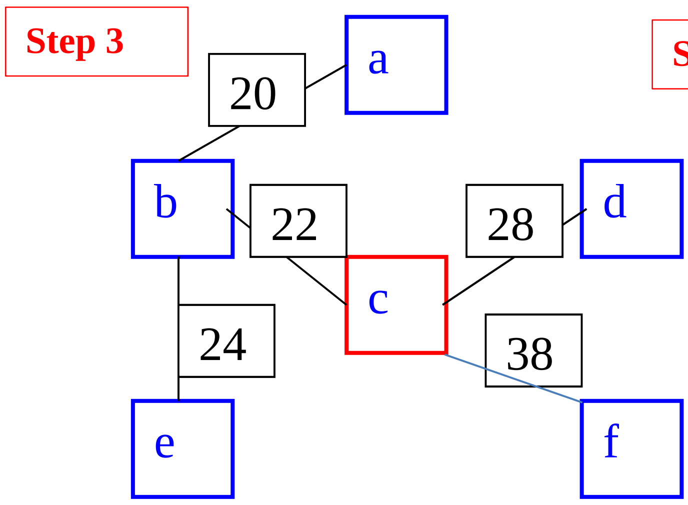
Iteration 1:	**(b,c,22) (b,e,24)**	**(b,c,22) ersätter (a,c,36)**
Iteration 2:	**(c,d,28)**, (c,e,30), (c,f,38)	**(c,d,28) ersätter (a,d,34)**
Iteration 3:	(e,f,36)	**(e,f,36) ersätter (c,f,38)**
Iteration 4:	**(d,f,26)**	**(d,f,26) ersätter (e,f,36)**

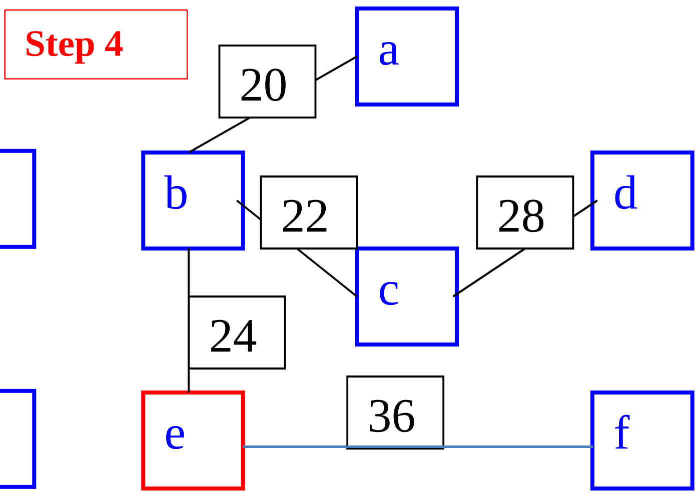
The process in pictures











Then the final step (step 5) to the solution above – (d,f,26) replaces (e,f,36) (cheaper!)

(8) Dijkstra

Algoritmen kan skrivas som nedan

```
Dijkstra( a )
{
    S = {a}
    for (i in V-S) D[i] = C[a, i];

    while(!is_empty(V-S)) {
        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]);
    }
}
```

Visa hur du skulle **utöka algoritmen** för att bygga den ”Shortest Path Tree” (kortaste väg trädet) samtidigt som man utför algoritmen.

(2p)

```
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)
    }

    while(!is_empty(V-S)) {
        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]
        }
    }
}
```

Tillämpa **den givna Dijkstra algoritmen (ovan)** på **den riktade grafen**,

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

Börja med nod ”a”.
Visa varje steg i Dina beräkningar.
Ange *alla* antaganden och visa *alla* beräkningar och mellanresultat

(3p)

Totalt 5p

Cost Matrix

	a	b	c	d	e
a		**12**		**11**	**9**
b			**7**		
c					**5**
d			**3**		**1**
e					

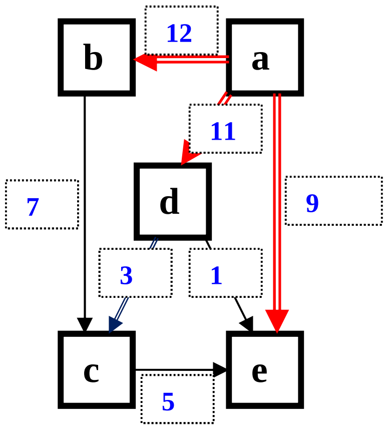
Initialise D

D: ¤ 12 § 11 9

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

v = b	min (D[b], D[e]+C (e,b))	➔ min(12, 9+§)	→ **no change**
v = c	min (D[c], D[e]+C (e,c))	➔ min(§, 9+§)	→ **no change**
v = d	min (D[d], D[e]+C (e,d))	➔ min(11, 9+§)	→ **no change**

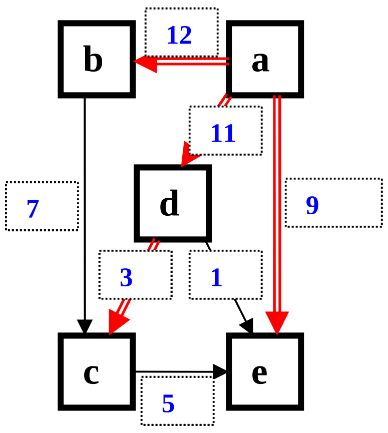
D: ¤ 12 § 11 9



D: ¤ 12 § 11 9

w is d (min value in D) S = {a,d,e} V-S = {b, c}
v = b min (D[b], D[d]+C (d,b)) ➔ min(12, 11+§) → **no change**
v = c min (D[c], D[d]+C (d,c)) ➔ min(§, 11+3) →**change** **a-d-c 14**

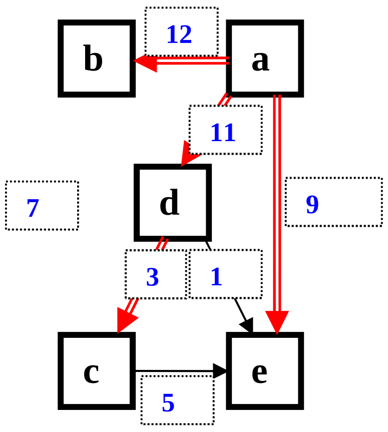
D: ¤ 12 14 11 9



D: ¤ 12 14 11 9

w is b (min value in D) S = {a,d,b,e} V-S = {c}
v = c min (D[c], D[b]+C (b,c)) ➔ min(14, 12+7) → **no change**

D: ¤ 12 14 11 9



This is the final result.

Costs: a→b (12), a→d→c (14), a→d (11), a→e (9)

Bilaga A

Heap Algoritmer

```
Heapify(A, i)
    l = Left(i)
    r = Right(i)
    if l <= A.size and A[l] > A[i] then largest = l else largest = i
    if r <= A.size and A[r] > A[largest] then largest = r
    if largest != i then
        swap(A[i], A[largest])
        Heapify(A, largest)
         end if
end Heapify

Build(A)
     for i = [A.size / 2] downto 1 do Heapify(A, i)
end Build
```

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

```
Add (H, v)
    let A = H.array
    A.size++
    i = A.size
    while i > 1 and A[Parent(i)] < v do
       A[i] = A[Parent(i)]
        i = Parent(i)
        end while
    A[i] = v
end Add
```