FACIT TILL OMTENTAMEN I
DATASTRUKTURER OCH ALGORITMER DVG B03

140818 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) Vad är ”big-O” för en funktion som skriver ut en adjacency matrix? Varför?
$O(n^2)$ – matrix is 2D which implies 2 nested for loops to display the content.

(b) Vad gör Dijkstras algorithm?
Calculates the length of the shortest PATH between a given node (the start node) and the remaining nodes in the graph.

(c) Vad gör Floyds algorithm?
All pairs shortest path algorithm. Calculates the length of the shortest PATH between each pair of nodes ((a, b) a != b)) in the graph.

(d) Vad gör Warshalls algorithm?
Calculates the transitive closure of the graph, i.e. if there is a PATH between any pair of nodes (a, b).

(e) Vad gör Topologisk sortering?
Given a DAG as input, produces a sequence which represents a partial ordering of the nodes in the DAG (Directed Acyclic Graph).

(f) Vad är en heap?
A data structure, which may be represented as an array or as a (binary) tree with the property that the parent node has a value which is greater than (or less than) its children. Is used to implement a priority queue (PQ)

(g) Vad är fördelen med hashning?
The add and find operations are O(1).

(h) Vad är en rekursiv funktion?
A function which calls itself – usually in a conditional call otherwise the function will ”disappear” in an endless sequence of recursive calls.

(i) Vad är ett AVL-träd?
A BST, Binary Search Tree, with an added constraint that the height of the left and right sub-trees may not differ by more than 1.

(j) Vad är dubbel hashning?
A conflict resolution technique where the f(i) function is a second hash function. Give an example.

Totalt 5p

(2) Heap

Diskutera ingående hur koden till heap operationer (**se Bilaga A**) fungerar? Använd sekvensen 46, 13, 18, 33, 72, 9, 11, 44, 27, 15, 66 som ett exempel. Anta att det största värdet hamnar i roten.

5p

Apply heapify to the above sequence of values.
Solution 1 – calculate the values using the algorithm
Input: 46, 13, 18, 33, 72, 9, 11, 44, 27, 15, 66
Array size = 11
NB: i, **l**, **r** and **largest** are **positions** in the array and not values
Exercise: draw the corresponding trees for each instance of the array.

step 1: for i = 5 downto 1 do Heapify(A, i)

the call to Heapify(A, 5)

i = 5 A = 46, 13, 18, 33, 72, 9, 11, 44, 27, 15, 66
i = 5; (value 72) **l = 10;** (value 15) **r = 11;** (value 66) **largest = 5;** (value 72)
largest = 5 (value 72) largest = i hence **no swap** giving

46, 13, 18, 33, 72, 9, 11, 44, 27, 15, 66

the call to Heapify(A, 4)

i = 4 A = 46, 13, 18, 33, 72, 9, 11, 44, 27, 15, 66
i = 4; (value 33) **l = 8;** (value 44) **r = 9;** (value 27) **largest = 8;** (value 44)
largest = 8 (value 44) largest != i hence **swap A[4]** and **A[8]** giving

46, 13, 18, 44, 72, 9, 11, 33, 27, 15, 66
Heapify(A, 8) has no effect on A (A[8] is a leaf node)

the call to Heapify(A, 3)

i = 3 A = 46, 13, 18, 44, 72, 9, 11, 33, 27, 15, 66
i = 3; (value 18) **l = 6;** (value 9) **r = 7;** (value 11) **largest = 3;** (value 18)
largest = 3 (value 44) largest = i hence no **swap** giving

46, 13, 18, 44, 72, 9, 11, 33, 27, 15, 66

the call to Heapify(A, 2)

i = 2 A = 46, 13, 18, 44, 72, 9, 11, 33, 27, 15, 66

i = 2; (value 13) **l = 4;** (value 44) **r = 5;** (value 72) **largest = 5;** (value 72)
largest = 5 (value 72) largest != i hence **swap A[2]** and **A[5]** giving

46, 72, 18, 44, 13, 9, 11, 33, 27, 15, 66

Heapify(A, 5) is a recursive call – **reorganize the sub-tree**

i = 5 A = 46, 72, 18, 44, 13, 9, 11, 33, 27, 15, 66

i = 5; (value 13) **l = 10;** (value 15) **r = 11;** (value 66) **largest = 11;** (value 66)
largest = 11 (value 66) largest != i hence **swap A[11]** and **A[5]** giving

46, 72, 18, 44, 66, 9, 11, 33, 27, 15, 13

Heapify(A, 11) has no effect on A (A[11] is a leaf node)

the call to Heapify(A, 1)

i = 1 A = 46, 72, 18, 44, 66, 9, 11, 33, 27, 15, 13

i = 1; (value 46) **l = 2;** (value 72) **r = 3;** (value 18) **largest = 2;** (value 72)
largest = 2 (value 72) largest != i hence **swap A[1]** and **A[2]** giving

72, 46, 18, 44, 66, 9, 11, 33, 27, 15, 13

Heapify(A, 2) is a recursive call – **reorganize the sub-tree**

i = 2 A = 72, 46, 18, 44, 66, 9, 11, 33, 27, 15, 13

i = 2; (value 13) **l = 4;** (value 44) **r = 5;** (value 66) **largest = 5;** (value 66)
largest = 5 (value 66) largest != i hence **swap A[2]** and **A[5]** giving

72, 66, 18, 44, 46, 9, 11, 33, 27, 15, 13

Heapify(A, 5) is a recursive call – **reorganize the NEXT sub-tree**

i = 5 A = 72, 66, 18, 44, 46, 9, 11, 33, 27, 15, 13

i = 5; (value 46) **l = 10;** (value 15) **r =11;** (value 13) **largest = 5;** (value 46)
largest = 5 (value 46) largest = i hence **no swap** giving

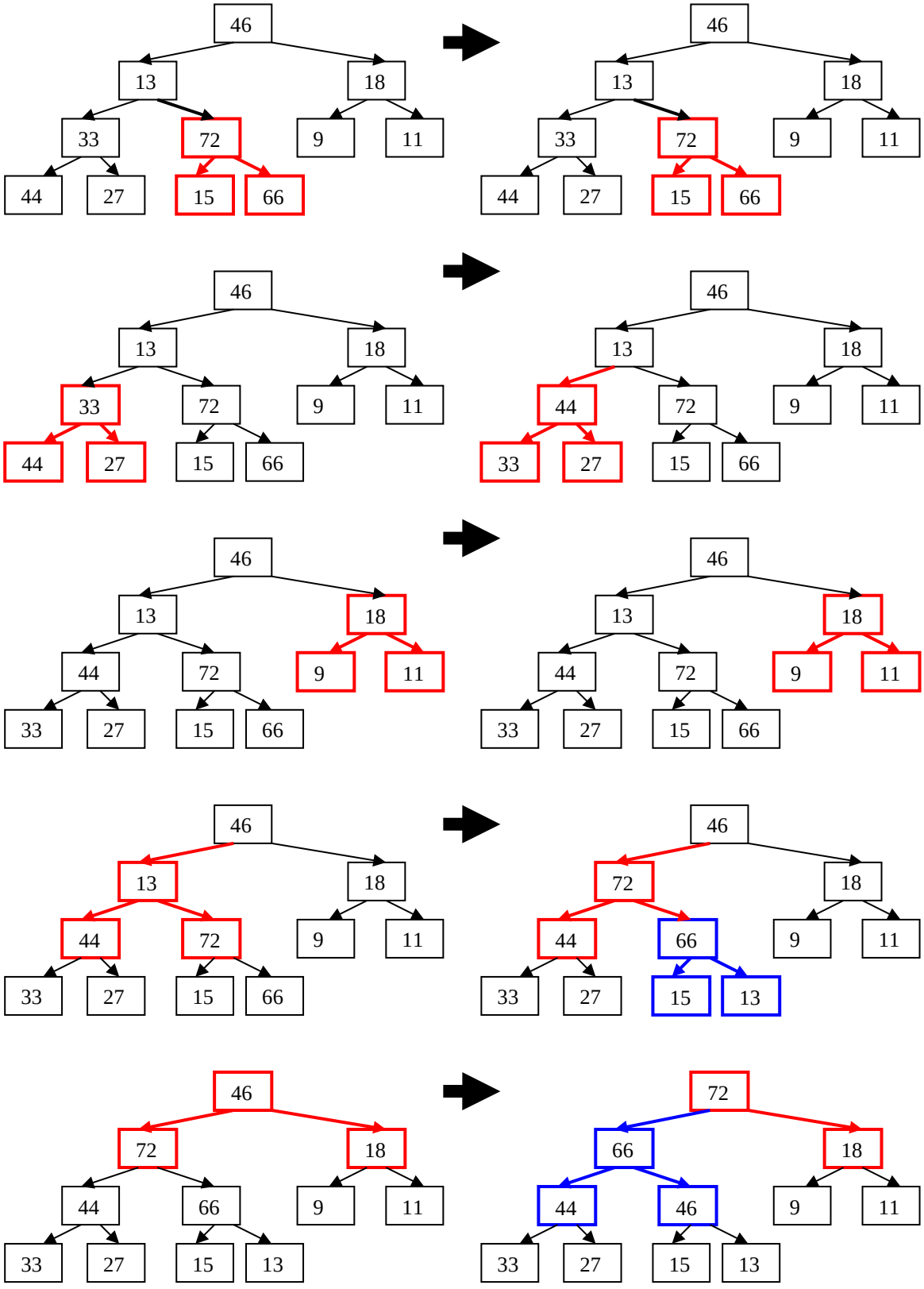
72, 66, 18, 44, 46, 9, 11, 33, 27, 15, 13

Return from the 2 levels of recursion and the algorithm is finished.

Explain the basic principles behind **heapify**

- Iterate over all the PARENTS i.e. lower(n/2)
  - Compare the parent LC and RC and move the largest value to the parent
  - Repeat the process recursively for the LC/RC **if** a swap took place

- Solution 2 – pictorial explanation



Remove: Swap the element to be removed with the last element in the heap and then remove the element. Re-heapify the resultant heap. (See above). You may use an example.

- Example: **72, 66, 18, 44, 46, 9, 11, 33, 27, 15, 13**
- **delete 72 ➔ heapify 13, 66, 18, 44, 46, 9, 11, 33, 27, 15, 13** (see above for heapify)

(3) Rekursion

Skriv (pseudo)kod till en **rekursiv funktion** (eller **två rekursiva funktioner**) 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));
  }
```

(4) Grafoperationer

En graf kan beskrivas som en virtuell (eller abstrakt) maskin. I graflaborationen, filen ”begraph.h” är faktiskt en beskrivning av denna virtuell maskin. Innehållet ges nedan.

```
/*************************************************************************/
/* function prototypes - operations on the Graph (a virtual machine)     */
/*************************************************************************/
/* Graph = (V, E) where V is a set of vertices/nodes and E a set of edges */
/* There are a limited number of operations (9) which can be applied to G */
/*************************************************************************/
void be_display_adjlist();             /* display G as an adjacency list  */
void be_display_adjmatrix();           /* display G as an adjacency matrix*/

void be_addnode(char c);               /* add a vertex (node) to G        */
void be_remnode(char c);               /* remove a vertex (node) from G   */

void be_addedge(char cs, char cd, int v);/* add an edge (with weight) to G*/
void be_remedge(char cs, char cd);     /* remove an edge from G           */

int  be_is_nmember(char c);            /* is a  node a member of G?       */
int  be_is_emember(char cs, char cd);  /* is an edge a member of G?       */

int  be_size();                        /* the number of nodes in G        */
/*************************************************************************/
```

I **front-end:en** kan man specifiera förvillkor (preconditions) till varje operation ovan med hjälp av en eller flera av dessa 9 funktioner. Skriv (pseudo)kod till föjlande front-end-funktioner på så sätt att änvändaren inte behöver mäta in mer information än absolut nödvändigt. Anta att det finns ett visst antal användargränssnittfunktioner som kan ta emot information samt skicka tillbaka informationsmeddelande.

(i)	fe_display_adjlist()	// display adjacency list	1p
(ii)	fe_remnode()	// remove node	1p
(iii)	fe_remedge()	// remove edge	3p

Totalt 5p

(i) fe_display_adjlist()

```
void fe_display_adjlist()  {
  if (be_size()==0) ui_putGraphEmpty();
  else { ui_putTitleList(); be_display_adjlist(); }
  }
```

(ii) fe_remnode()

```
void fe_remnode()  {

  char c;

  if (be_size()==0) ui_putGraphEmpty();
  else {
    c = ui_getNode();
    if (!be_is_nmember(c)) ui_putNoNodeError(c);
    else be_remnode(c);
    }
  }
```

(iii) fe_remedge()

```
void fe_remedge()  {

  char cs, cd;

  if (be_size()==0) ui_putGraphEmpty();
  else {
    cs = ui_getNode();
    if (!be_is_nmember(cs)) ui_putNoNodeError(cs);
    else {
      cd = ui_getNode();
      if (!be_is_nmember(cd)) ui_putNoNodeError(cd);
      else if (!be_is_emember(cs, cd)) ui_putNoEdgeError(cs, cd);
      else {
        be_remedge(cs, cd);
        if (ui_isModeU()) {
          if (!be_is_emember(cd, cs)) ui_putNoEdgeError(cs, cd);
          else be_remedge(cd, cs);
          }
        }
      }
    }
  }
```

(5) Labbkod

(a) I graflabben har en student skrivit följande kod för att ta bort 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 reme(char cs, char cd) {
  set_edges(b_findn(cs, G), b_reme(cd, 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) (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_reme(e, Elist) removes cd from 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) I trädlabben har en student skrivit kod för att söka efter ett värde i ett BST (binärt sökträd). Sedan har studenten kommit på att denna kod kunde lätt anpassas för att söka efter ett värde i ett komplett träd. Dessa funktioner finns nedan. Vad har studenten skrivit för ”**xxx**” och ”**yyy**”? **Ange alla antagande.**

```
static int b_findb(treeref T, int v)
{
  return is_empty(T) ? 0
        : v <  get_value(node(T)) ? b_findb(LC(T), v)
        : v >  get_value(node(T)) ? b_findb(RC(T), v)
        : 1;
}

static int b_findc(treeref T, int v)
{
  return is_empty(T) ? 0
        : xxx ? 1          xxx ➔ v == get_value(node(T))
        : yyy;             yyy ➔ b_findc(LC(T), v) || b_findc(RC(T), v);
}
```

1p

(c) Skriv (pseudo)kod för att lägga till ett element i ett binärt träd.
Ange alla antagande. **2p**

```
T: Add(T,v) {
    if IsEmpty(T) then return v
    if IsEmpty(v) then return T
    if value(v) < value(T) then return cons(Add(left(T), v), T, right(T))
    if value(v) > value(T) then return cons(left(T), T,  Add(right(T), v))
    return T
   }
```

Totalt 5p

(6) Dijkstra + SPT (Shortest Path Tree)

Tillämpa **den givna Dijkstra_SPT algoritmen (nedan)** 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)

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

Börja med nod ”a”.
Visa varje steg i dina beräkningar.
Ange *alla* antaganden och visa *alla* beräkningar och mellanresultat
Rita **varje steg** i konstruktionen av SPT:et – dvs visa till och med de noder och kanter som läggs till men sedan tas bort.

(3p)

Förklara **principerna** bakom **Dijkstras_SPT** algoritm.

(2p)
Totalt 5p

Dijkstras algoritm med en utökning för SPT

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

        for (i in V-S) {
          D[i]   = C[a, i]          --- initialise D - (edge cost)
          E[i]   = a                --- initialise E - SPT (edge)
          L[i]   = C[a, i]          --- initialise L - SPT (cost)
        }

        for (i in 1..(|V|-1)) {
                choose w in V-S such that D[w] is a minimum
                S = S + {w}
                foreach ( v in V-S )  if (D[w] + C[w,v] < D[v]) {
                        D[v] = D[w] + C[w,v]
                        E[v] = w
                        L[v] = C[w,v]
                 }
        }
}
```

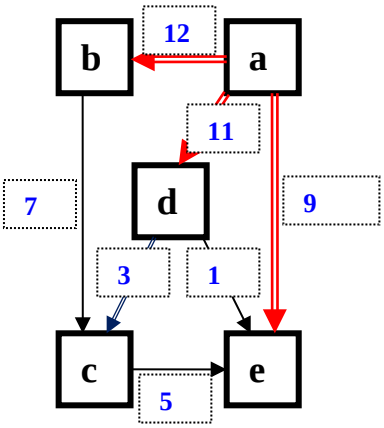
	a	b	c	d	e
a		**12**		**11**	**9**
b			**7**		
c					**5**
d			**3**		**1**
e					

Initialise D, E, L

D: ¤ 12 § 11 9
E: ¤ a a a a
L: ¤ 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**

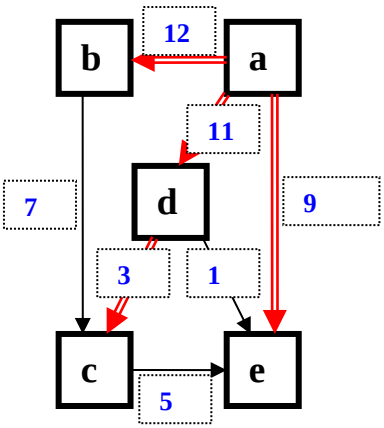
D: ¤ 12 § 11 9
E: ¤ a a a a
L: ¤ 12 § 11 9



D: ¤ 12 § 11 9
E: ¤ a a a a
L: ¤ 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**

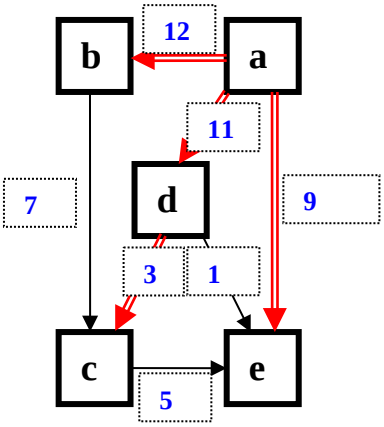
D: ¤ 12 14 11 9
E: ¤ a d a a
L: ¤ 12 3 11 9



D: ¤ 12 14 11 9
E: ¤ a d a a
L: ¤ 12 3 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
E: ¤ a d a a
L: ¤ 12 3 11 9

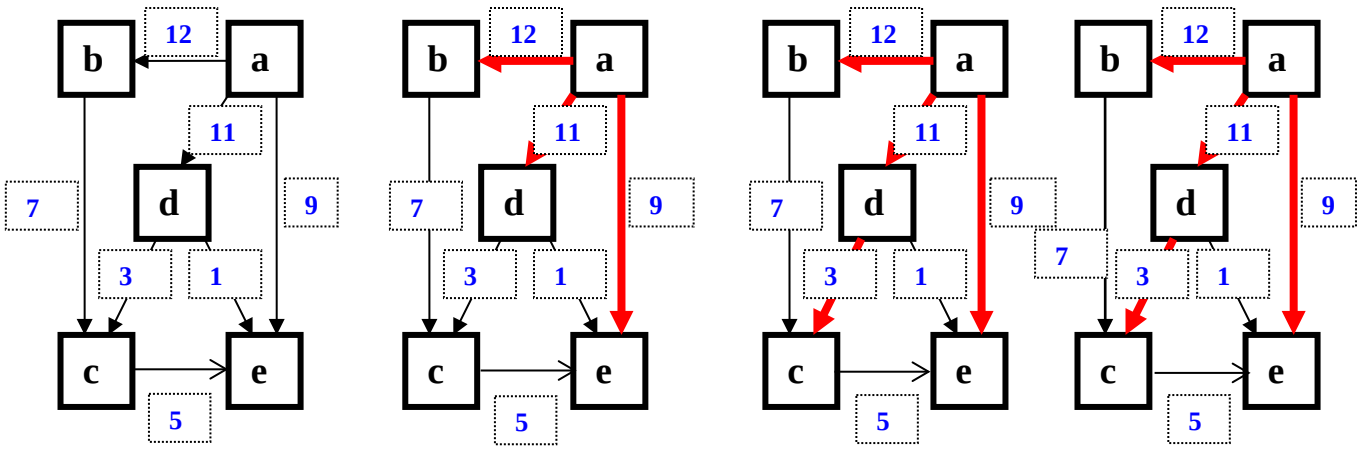


This is the final result.

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

SPT edges: a→b (12), d→c (3), a→d (11), a→e (9)

Principle – marks for a good explanation!



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
```