FACIT TILL
ORDINARIE TENTAMEN I
DATASTRUKTURER OCH ALGORITMER DVG B03

150112 kl. 08:15 – 13:15

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.

---

(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.

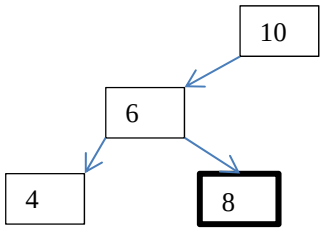
(j) Vad är dubbel hashning?
A conflict resolution technique where the f(i) function is a second hash function. Give an example.

Totalt 5p

(2) Ge ett kortfattat svar till följande uppgifter ((a)-(e)).

(a) Kan noden med det maximala värdet i ett vänsterbarn i ett BST (Binärt SökTräd) ha ett högerbarn? Förklara varför!

NO since the original node (6) would not be a maximum value by definition which gives a contradiction. Draw an example to show this!



(b) Skriv rekursiv pseudokod till en funktion för att hitta det minimala värdet i ett högerbarn i ett BST (Binärt SökTräd).

```
static int find_min(treeref T) {
return  is_empty(LC(T)) ? get_value(T) : find_min(LC(T));
}
```

(c) Ge en rekursiv definition av ett BT (Binärt Träd)

BT ::= LC N RC | empty
N ::= element
LC ::= BT
RC ::= BT

(d) Skriv en rekursiv sök (”find”) funktion för ett BT (Binärt Träd) – **OBS – ej BST!**

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

(e) Förklara hur Du skulle representera ett BT (Binärt Träd) med hjälp av en array. Vad är förhållandet mellan trädet och arrayen?

A binary tree may be represented as an array with the root in position 1, the left child in position 2 and the right child in position 3.

If the indexing starts at 1:
In general the left child is found at index i where i = 2 * index(parent) and
the right child is found at index j where j = 2 * index(parent)+1.

The tree is stored in the array in breadth-first order.

Totalt 5p

(3) 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. **Visa varje steg i Dina beräkningar.**

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 46) **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

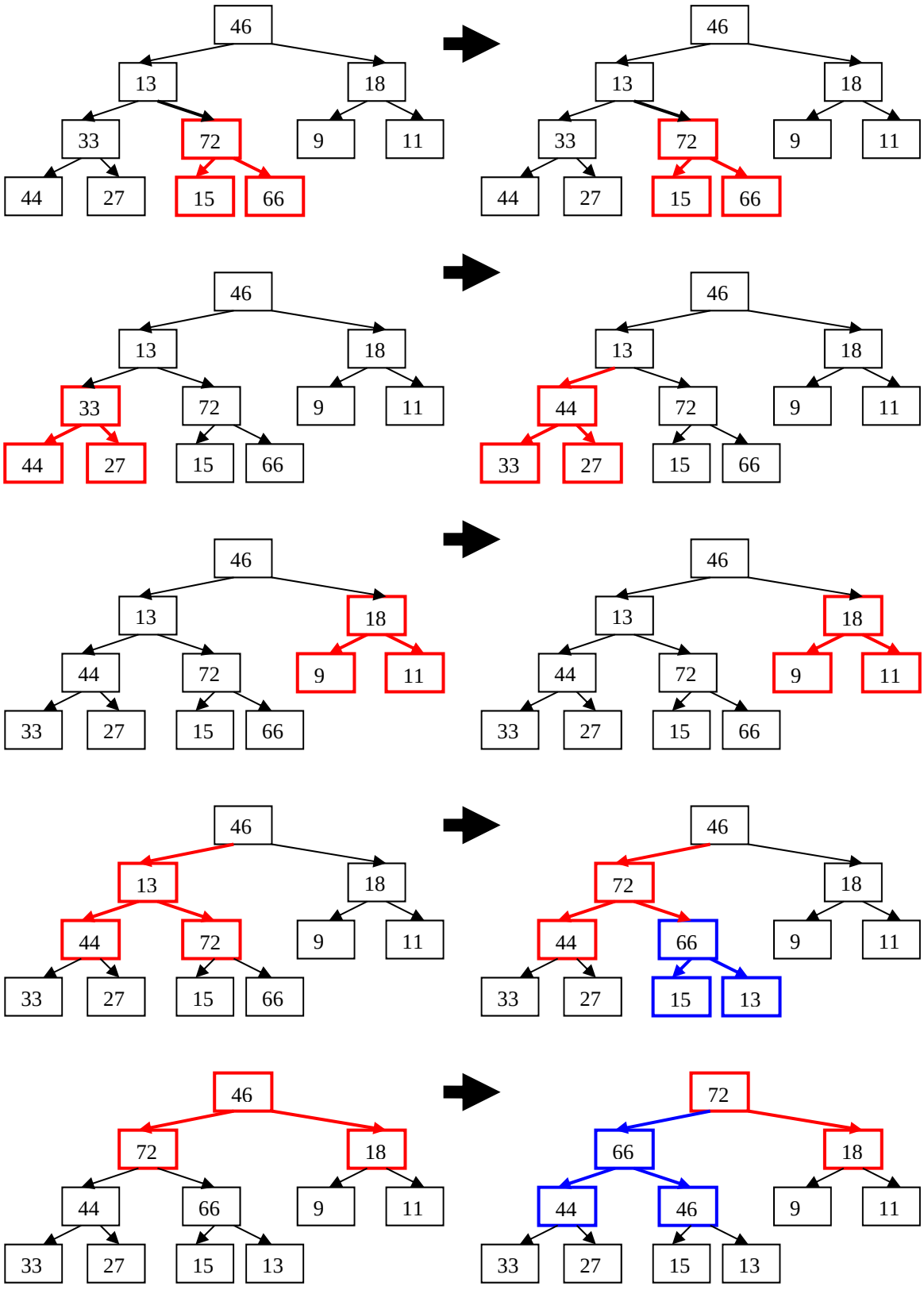
72, 66, 18, 44, 46, 9, 11, 33, 27, 15, 13

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

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** (see above for heapify)

(4) Rekursion

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));
  }
```

(5) Diskussionsuppgift

Vad menas med **implentationsabstraktion**? Presentera för- och nackdelar till implementationsabstraktion. Ledar det fram till att man producerar bättre kod? **Diskutera ingående. Ge alla antagande.**

5p

Definition 3: IMPLEMENTATION ABSTRACTION
The process of selecting certain properties of a Data Type independent of the implementation of that Data Type - hence the expression Abstract Data Type

Example 3: A sequence may be implemented using

- an array of values and an index to that array
- arrays (value, next) and an index
- records/structures and pointers (a linked list)

Advantages

- The implementation is hidden from most of the code – the exception is
  - the definition of the NULLREF value (-1 for arrays, NULL for pointers)
  - the get/set functions, one per attribute
  - the create_element function
- the implemenntation can be more easily changed (array ➔ linked lists or vice versa)
- the rest of the code becomes more abstract
- self-documenting function names may then be chosen
- the code becomes shorter (especially when combined with recursion and a more functional style of programming)
  - e.g.

```
static void be_add_pos(int fval, int fpos) {
   moveToPosition(fpos); link_in(create_e(fval));
}
```

  - e.g.

```
static listref be_find_val(listref L, int v) {
return (is_empty(L) || (v==get_value(head(L)))) ? L :
be_find_val(tail(L), v);
}
```

Disadvantages

- The technique requires more thought in the beginning

Better code? Is a topic for discussion and argument

Marks for good arguments.

(6) AVL-Träd

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

2p

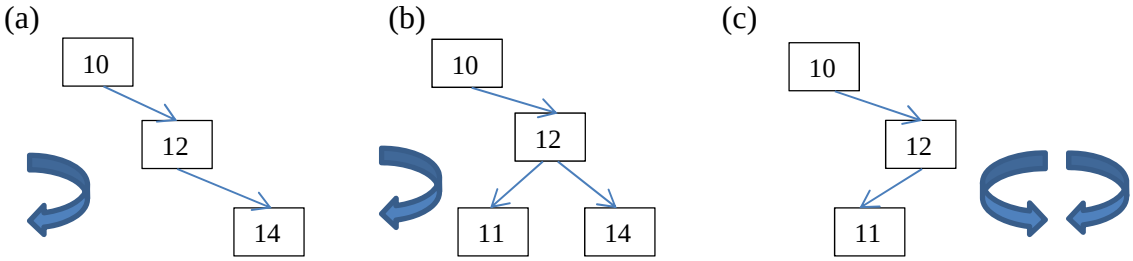
An AVL-tree is a BST (Binary Search Tree) where the balance difference (balance factor) between the height of the left child and the height of the right child may be at most 1.

i.e. | height(LC) – height(RC) | <= 1

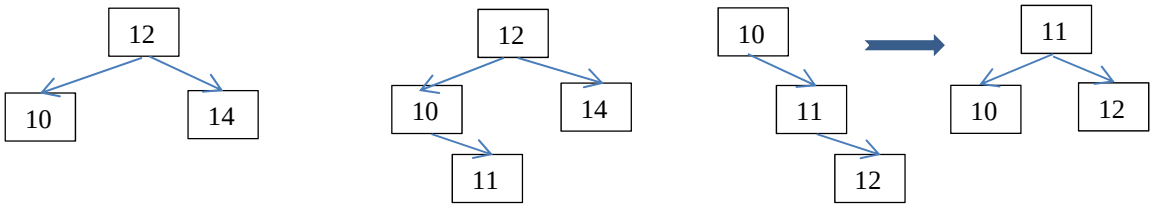
The rotation functions are

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

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. **Skriv balansfunktionen i pseudokod.**

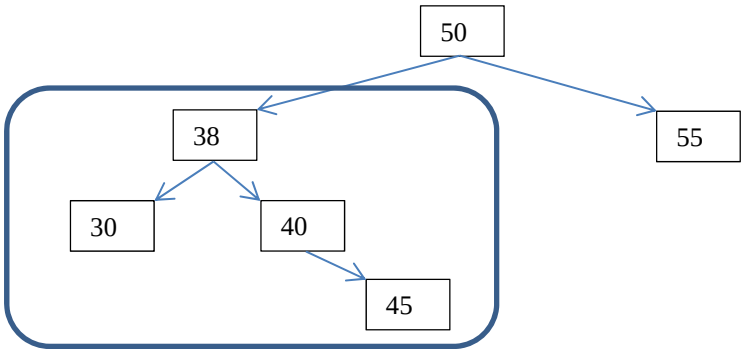
Define the balance factor br = height(LC(T) – height(RC(T))

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).

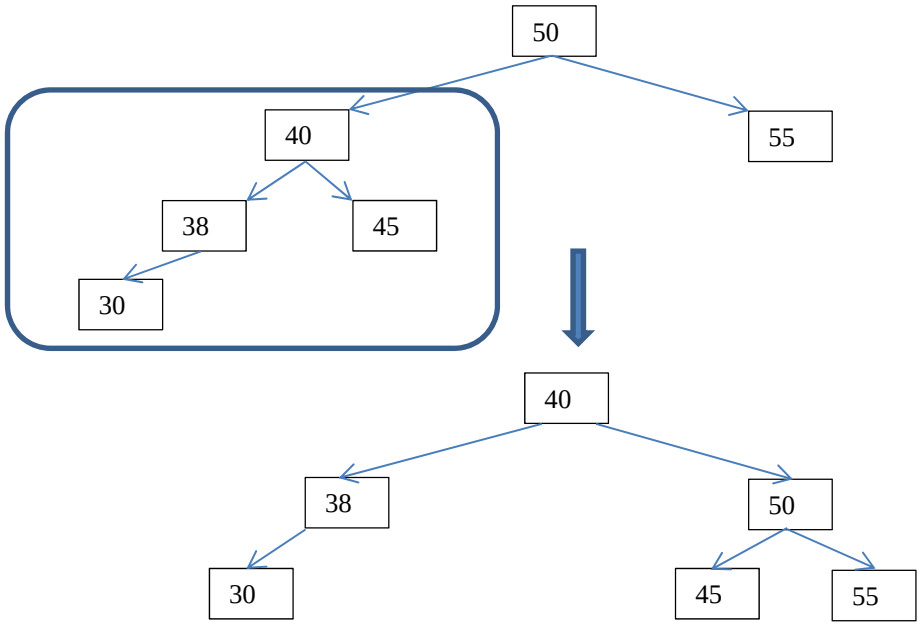
Tillämpa Din pseudokod på trädet nedan

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;

Totalt 5p

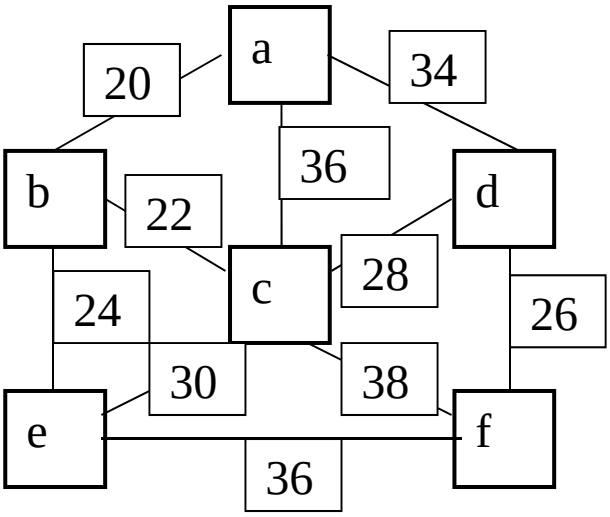
(7) Kruskals algoritm

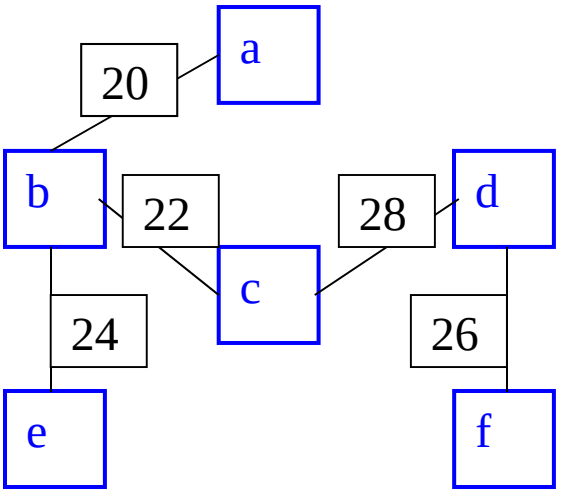
Beskriv principerna bakom Kruskals algoritm. Vad blir resultatet av en tillämpning av algoritmen? Använd grafen nedan (oriktad) som exempel.

(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)

3p

Use an example to show how Kruskal's works





1. Construct a **priority queue (PQ)** with the edges – lowest value first ➔

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

2. Remove all edges from the graph and consider each node as a component of the graph

3. Choose and edge from the PQ which connects **2 distinct components**
   a. a-20-b connects component a to b giving component a-20-b
   b. b-22-c connects a-20-b to c giving component a-20-b, b-22-c
   c. b-24-e connects a-20-b, b-22-c to e giving component a-20-b, b-22-c, b24-e
   d. d-26-f connects component d to f giving d-26-f
   e. c-28-d connects component a-20-b, b-22-c, b24-e to d-26-f
   f. the MST (Minimal Spanning Tree) has now been found (see picture above)

Hur skulle man kunna anpassa idéer från Kruslals algoritmen för att ta fram en heuristik för att ge en lösning till det resande försäljare-problemet (Travelling Salesman Problem)?

Use a variant of Kruskal by adding an extra condition that no node may have a degree greater than 2 and that no edges except the last may result in a cycle.

2p

Totalt 5p

(8) 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]
                 }
        }
}
```

Cost Matrix

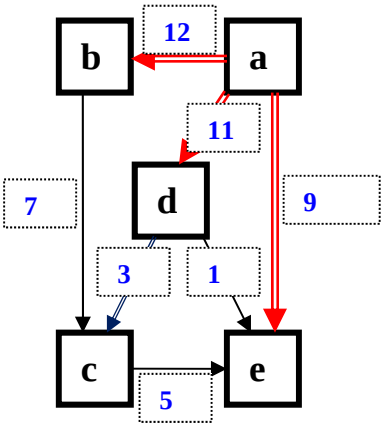
	a	b	c	d	e
a		**12**		**11**	**9**
b			**7**		
c					**5**
d			**3**		**1**
e					

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**

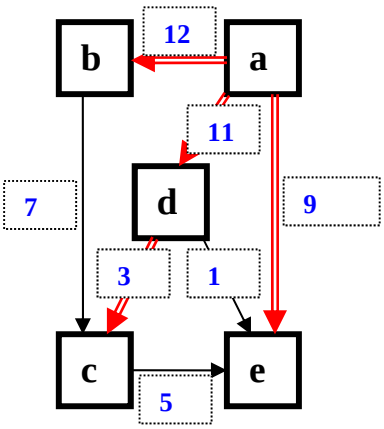
D: ¤ 12 § 11 9
E: ¤ a a a a
L: ¤ 12 § 11 9



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**

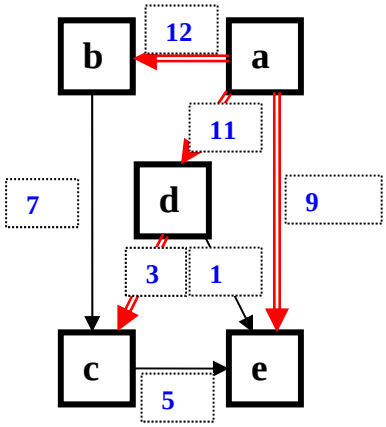
D: ¤ 12 14 11 9
E: ¤ a d a a
L: ¤ 12 3 11 9



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

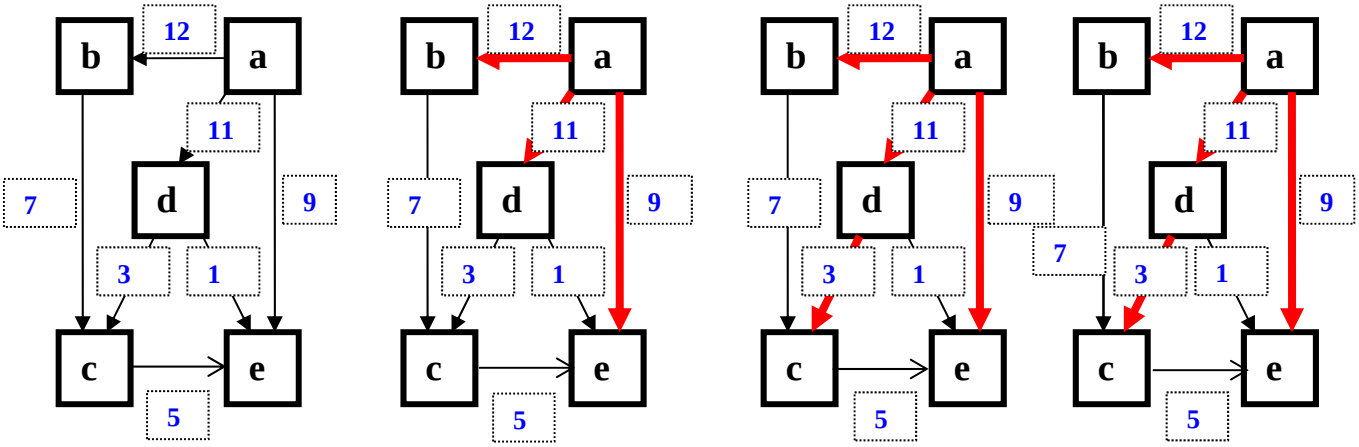


This is the final result.

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
```