FACIT till ORDINARIE TENTAMEN I DATASTRUKTURER OCH ALGORITMER DVG B03

160119 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

<u>SKRIV TYDLIGT – LÄS UPPGIFTERNA NOGGRANT</u>

<u>Ange alla antaganden.</u>

(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?
(b) Vad gör Dijkstras algoritm?
(c) Vad gör Floyds algoritm?
(d) Vad gör Warshalls algoritm?
(e) Vad gör Topologisk sortering?
(f) Vad är en heap?
(g) Vad är fördelen med hashning?
(h) Vad är en rekursiv definition?
(i) Vad är ett AVL-träd?
(j) Vad är dubbel hashning?

Totalt 5p

Question 1 - Common mistakes – not knowing the material!

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

(a) Vad är ”big-O” för en funktion som skriver ut en adjacency matrix? Varför?

a. O(n^2) – matrix is 2D which implies 2 nested for loops to display the content.

(b) Vad gör Dijkstras algorithm?

a. 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?

a. 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?

a. 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?

a. 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. 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?

a. The add and find operations are O(1) i.e. constant.

(h) Vad är en rekursiv definition?

a. A definition which is PARTLY defined in terms of itself – e.g. sequence, BT

(i) Vad är ett AVL-träd?

a. 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. A conflict resolution technique where the f(i) function is a second hash function.

b. Give an example.

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

(a) Skriv **abstrakt rekursiv pseudokod** för en back-end Boolsk ”find” funktion för att hitta ett värde i en sekvens. Skriv om funktionen så att den returnerar en referens till ett element eller NULLREF i stället för sant eller falsk.

2p

```
static int be_find_val(sequence S, int v) {
return      is_empty(S)                    ? 0          // false
       :    v==get_value(head(S))          ? 1          // true
       :                                     be_find_val(tail(S), v);
}

static sequence be_find_val(sequence S, int v) {
return      is_empty(S)                    ? S          // not found   (S==NULLREF)
       :    v==get_value(head(S))          ? S          // found       (S!=NULLREF)
       :                                     be_find_val(tail(S), v);
}

Alternative:
int  be_is_member(int v)       { return !is_empty(be_find_val(S, v)); }
```

(b) Skriv **abstrakt rekursiv pseudokod** till en funktion för att lägga till (add) ett värde i en sekvens. Skriv om denna funktion med bara två rader i stället för de tre rader som man får om man programmerar enligt sekvensmönstret. Dvs hur kan man kombinera 2 operationer?

2p

```
/* standard form according to the sequence pattern */
static sequence be_add_val(sequence S, int v)  {
   return is_empty(S)                ? create_e(v)
    :    v < get_value(head(S))      ? cons(create_e(v),  S)
    :                                  cons(head(S), be_add_val( tail(S), v) );
}

/* compact (short) form */
static sequence be_add_val(sequence S, int v)  {
   return (is_empty(S)  ||  v < get_value(head(S)))   ?   cons(create_e(v),  S)
    :   cons(head(S), be_add_val( tail(S),v) );
}
```

Assumption: Sequence is sorted in ascending order

(c) Skriv **abstrakt rekursiv pseudokod** till en funktion för att ta bort (remove) ett värde från en sekvens. Förklara hur denna funktion fungerar om inte värdet finns i sekvensen.
2p

Assumption: Sequence is sorted in ascending order

```
static sequence be_rem_val(sequence S, int v) {
  return is_empty(S)                    ? S
   :    v == get_value(head(S))         ? tail(S)
   :                                      cons(head(S), be_rem_val(tail(S), v));
}
```

If the value v is NOT in the sequence then the function will repeatedly call itself via the **cons**(**head**(S), **be_rem_val**(**tail**(S), v)); until the end of the sequence is reached at which point the recursion ceases at **is_empty**(S) at which point the recursion "unwinds" and eventually returns the original sequence which is actually the correct result.
Remove d from (a, b, c) is (a, b, c)!

(d) Skriv **abstrakt rekursiv pseudokod** till funktionen **T2Q** från labb 1. **T2Q** förvandlar ett binärt träd till en array.
2p

No facit since this is still a current lab exercise. ☺

(e) Skriv den **rekursiva definitionen** av ett AVL-träd.
2p

```
BT  ::=     LC  N  RC  | empty        // non-empty or empty
N   ::=     element                   // N is the node / root
LC  ::=     BT                        // LC is the Left Child
RC  ::=     BT                        // RC is the Right Child
```

An AVL-tree is a BST with a height constraint

| height(LC) – height(RC) | < 2

And a BST is a BT with the constraint that

All values in the LC are less than the value in the node
All values in the RC are greater than the value in the node

Hence the BT recursive definition is also the recursive definition for an AVL-tree

Totalt 10p

Question 2 - Common mistakes:-

NOT READING THE QUESTION CAREFULLY!!!

Many missed the fact that there were 2 parts to each question in 2a, 2b, 2c.

Sequence –not following the pattern for a sequence – namely

1. **The EMPTY case – if the sequence is empty no further actions may be taken!**
2. **The non-empty, NON_RECURSIVE case i.e. process the HEAD of the sequence**
3. **The non-empty, RECURSIVE case i.e. process the TAIL of the sequence**

Note that case 3 involves a cons to re-construct the list with the head and MODIFIED tail.

Example:

```
static sequence be_rem_val(sequence S, int v) {
    1. return is_empty(S)                    ? S
    2. :      v == get_value(head(S))        ? tail(S)
    3. :                                       cons(head(S), be_rem_val(tail(S), v));
}
```

Question 3 - Common mistakes:-

1. **Not reading the question carefully enough – many did not give an explanation for ADD and REMOVE.**
2. **Not using the code properly – many explanations were "by inspection" from the tree representation and not the array representation given in the code in the appendix.**
3. **Lack of detail in the answers – no step-wise calculations (see the question)**
4. **Explanations of the code were based on a "translation" of the code to English/Swedish rather than an "interpretation" of the code. I underlined this point in the lecture (for those who were there!)**

E.g.

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

if the left child exists then decide which of the LC or Parent is the larger (**interpretation**)

NOT if l is less than or equal to the size of the array and A[l] is greater than A[i] then set largest to l otherwise set largest to i == **translation** code to English!

(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

Remove

The code should have read

```
Remove (H, r)
    let  A = H.array
    A[r] = A[A.size]
    A.size--
    Heapify(A, r)        error ➔➔➔➔ Heapify(A,1)      i.e. heapify the whole array
    end Remove
```

Heapify(A, r) will not work in some cases – one student discovered this!

Steps

1. Set the value to be removed (index r) to the last value in the array
2. Reduce the size of the array (effectively removing the last value)
3. Re-heapify the whole array

Add

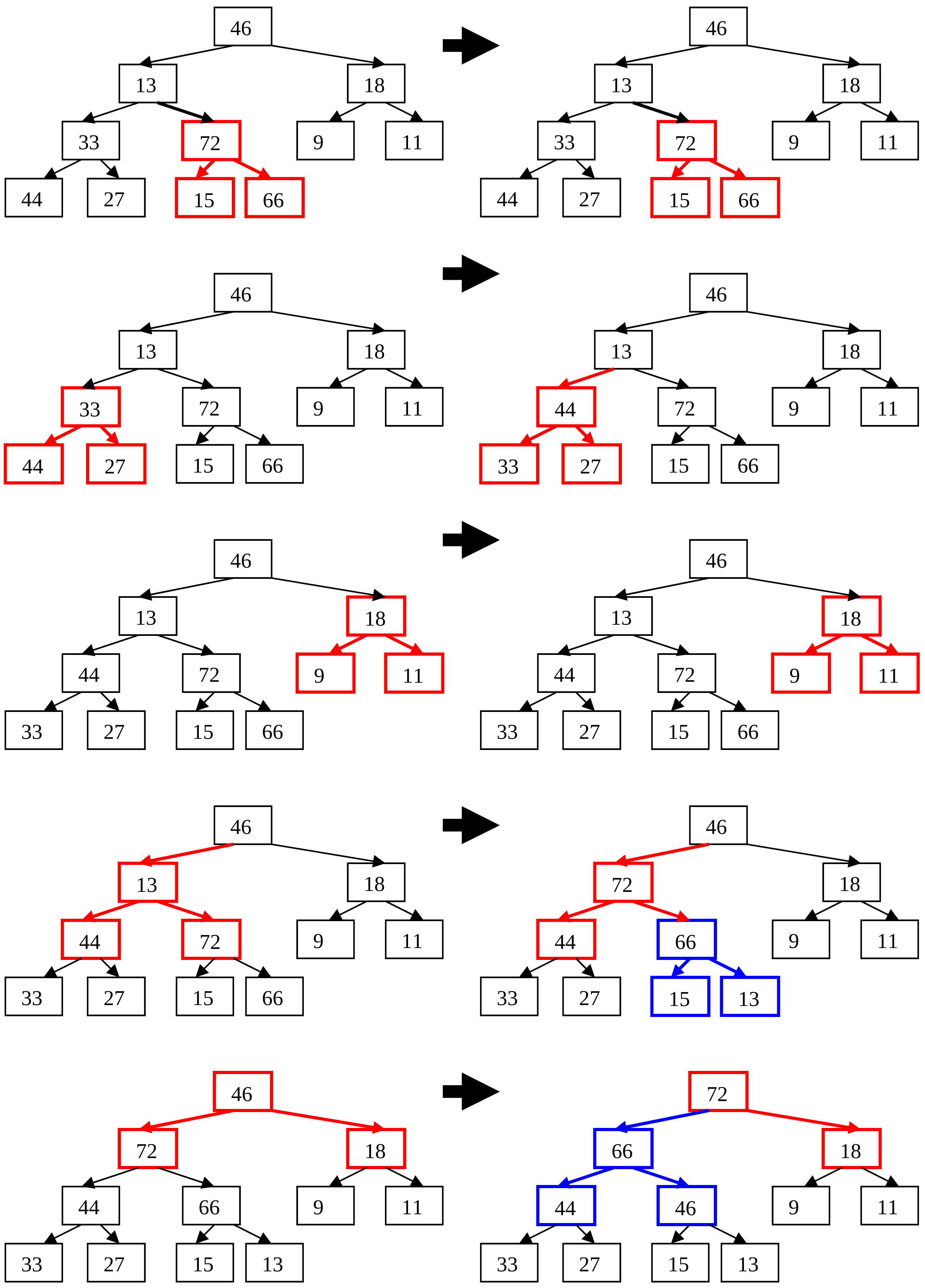
```
Add (H, v)
        let A = H.array
        A.size++
        i = A.size
        while i > 1 and A[Parent(i)] < v do
                A[i] = A[Parent(i)]
                i = Parent(i)
            end while
        A[i] = v
end Add
```

Steps

1. Increase the size of the array by one – create a new entry in the array
2. Set i to point to this entry – the last entry
3. As long as the first entry in the array (the root in the tree) has not been reached AND the value of the parent of index i is less than v (the value to be added)
    a. **Copy** the value of the parent to the current entry (i)
    b. Set i to the parent

The values of the parent "bubble" down the tree / move to the right of the array until the correct position for the new value is found.
The algorithm searches bottom up (tree) or right to left (array) for the correct position of the new value.

Pictorial explanation



(4) Rekursion

Skriv **abstrakt rekursiv pseudokod** till en funktion som tar bort (remove) ett värde från ett binärt sökträd. Vilka aspekter måste man ta hänsyn till när man skriver en sådan funktion? **Ange alla antagande**.

5p

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

Then

```
static  treeref removeAtRoot(treeref T)
{
  return  is_LeafNode(T)          ? NULLREF          // no LC, no RC
       :      is_empty(LC(T))     ? RC(T)            // RC only
       :      is_empty(RC(T))     ? LC(T)            // LC only
       :                             twoChild(T);    // LC and RC
  }
```

Now write twoChild(T) and the corresponding help functions.

removeAtRoot(T) has 4 cases – shown above!

twoChild(T) – has 2 cases

1. Replace the root node value with the **maximum** value of the **LC**
2. Replace the root node value with the **minimum** value of the **RC**

The maximum/minimum values are removed from the LC, RC respectively

To maintain some balance in the tree, the deeper of the LC, RC should be chosen. I.e. use the height difference.

NOTE: there are alternative ways of coding these ideas.
removeAtRoot(T) & twoChild(T) may be combined in 4 lines! ☺
No more pseudocode since this is part of the current labs! ☺

Question 4 – common mistakes.

Not using the recursive pattern for a BST

1. The empty case – if the BST is empty, no further action may be taken
2. The LC case – RECURSIVE – process the left child
3. The RC case – RECURSIVE – process the right child
4. The ROOT case – NON-RECURSIVE – process the (local) root

Note that cases (2) and (3) call cons which for a BST is cons(LC, N, RC)
Note that case (4) is the most complicated

Example

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

Case (4) (removeAtRoot) is the most "complicated" and has 4 cases – see the example above.

1. (local) root with no children – return NULLREF
2. (local) root with LC only - return LC
3. (local) root with RC only - return RC
4. (local) root with 2 children

Cases (1), (2), (3) are simple but case (4) requires 2 further decisions

1. replace the (local) root with the maximum value of the LC
2. replace the (local) root with the minimum value of the RC

To maintain a reasonable balance in the tree, the deeper of LC, RC should be chosen.

NOTE: the principle here is to deal with the simple cases first and then the more "complicated" case.

(5) Diskussionsuppgift

Man använder Big-oh när man pratar om en algoritmens prestanda. Vad betyder Big-oh? Vilka aspekter måste man ta hänsyn till när man tillämpar Big-oh. Vilka är de för- och nackdelarna?
Svara ingående. Ange alla antagande.

5p

Big-oh is an estimate of the time requirement of a given algorithm with respect to the number of elements in the collection.

The most interesting Big-oh estimates are

1.	$O(1)$	- constant time	e.g.: stack push, pop
2.	$O(n)$	- linear time	e.g.: list searching
3.	$O(n^2)$	- quadratic time	e.g.: bubble sort
4.	$O(n^3)$	- cubic time	e.g.: Floyd/Warshall
5.	$O(\log n)$	- logarithmic time	e.g.: find in a BST
6.	$O(n \log n)$	- ”n log n” time	e.g.: Quicksort

The ”not-so-interesting” Big-oh estimates are

1.	$O(n!)$	- factorial time
2.	$O(a^n)$ $(a>1)$	- exponential time

The aspects one should think of are

1. **Dividing problems into computable and non-computable classes**
2. **Dividing computable problems into tractable and intractable classes**
3. **Tractable problems may be solved in polynomial time e.g. $O(n^2)$**
4. **Intractable problems are those in the class $O(n!)$, $O(a^n)$**

Running times for code

1.	assignment, read, write	- usually $O(n)$
2.	statement sequence	- sum the cost of the statements (max rule)
3.	if then else	- $O(1)$ for the if condition + max(the, else)
4.	loops	- sum cost of loop body
5.	k nested loops	- $O(n^k)$ e.g. sorts with swaps $O(n^2)$
6.	functions	- sum of costs – recursive often $O(n)$

Marks for good answers.

Question 5 – common mistakes

Not knowing the material and not writing enough.

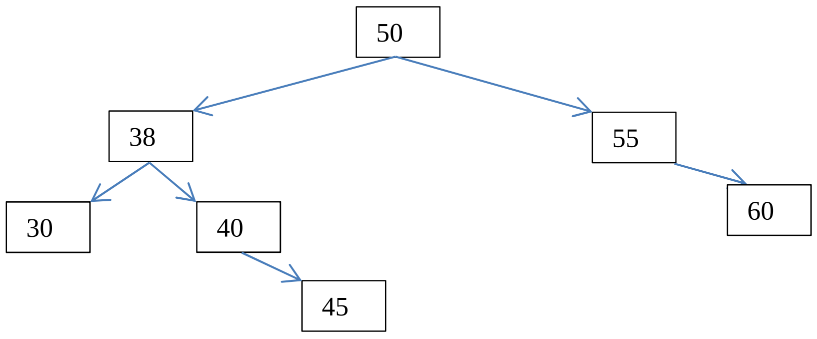
(6) AVL-Träd

Skriv **abstrakt pseudokod** till de 4 rotationsfunktionerna från första principer.

2p

Skriv **abstrakt pseudokod** för att ombalancera (re-balance) ett AVL-träd.
Tillämpa dessa funktioner på AVL-trädet nedan efter att man har tagit bort 60 från trädet.
Visa varje steg i dina beräkningar fram till lösningen. Förklara hur du använder din pseudokod vid varje steg.

3p



Totalt 5p

Question 6 – common mistakes

1. Not knowing the material (the SLR, SRR, DLR, DRR)
2. The single rotation SLR, SRR has actually 2 cases (see below (a) and (b))
3. The explanation was based on a ”by inspection” approach and did not reflect the actual code written in the answer to the first apart of the question
4. Not checking that the final result given was in fact an AVL tree i.e. **carelessness**!

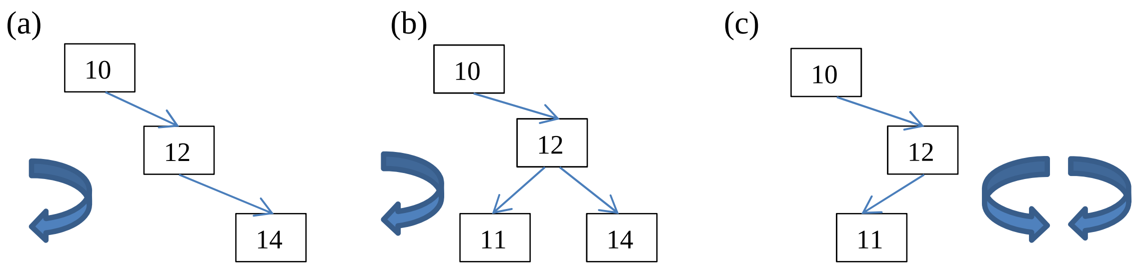
An AVL-tree is a BST (Binary Search Tree) where the balance difference (balance factor) between the height of the left child and the height of the right child may be at most 1.

i.e. | height(LC) – height(RC) | <= 1

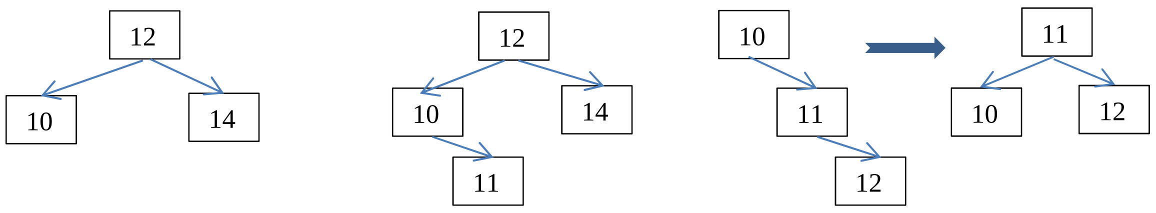
The rotation functions are

- 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.**

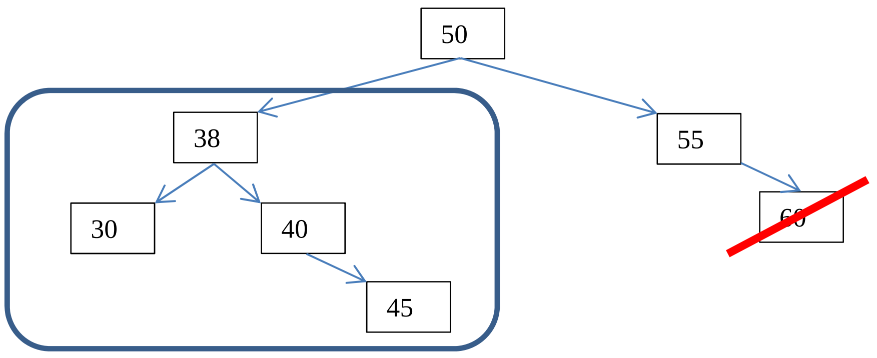
Define the balance factor br = height(LC(T) – height(RC(T))

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).

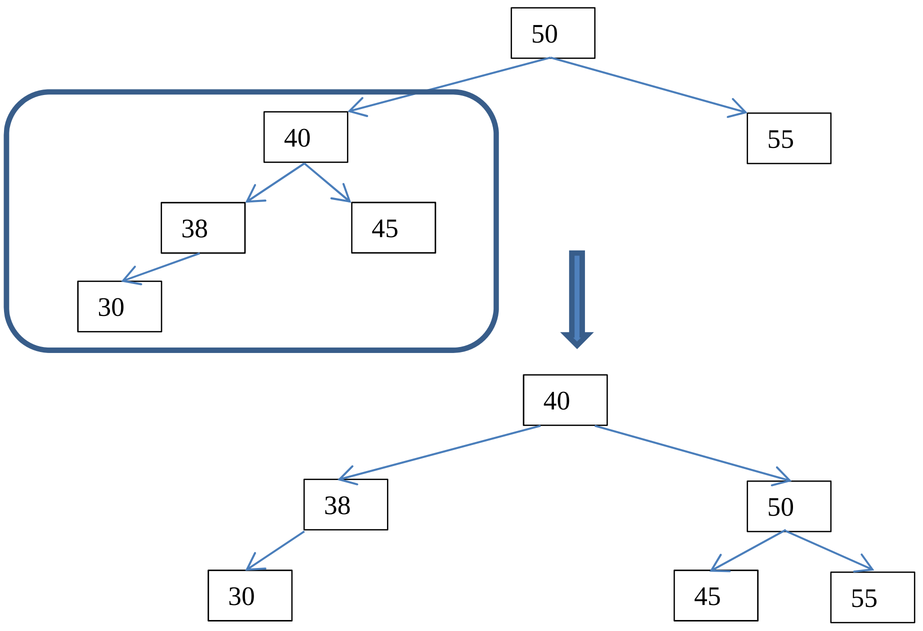
Tillämpa Din pseudokod på trädet nedan

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.

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;

***NB since this is still part of the current lab exercises, the actual code is not given here BUT you ARE required to write (pseudo)code in your answer.**

”By inspection” answers will NOT be awarded points!

(7) 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 **…and DRAW THE GRAPH !!!**

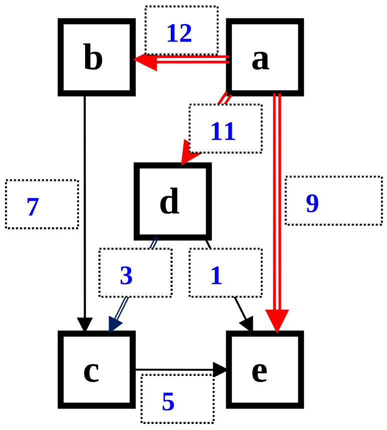
	a	b	c	d	e
a		12		11	9
b			7		
c					5
d			3		1
e					

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**

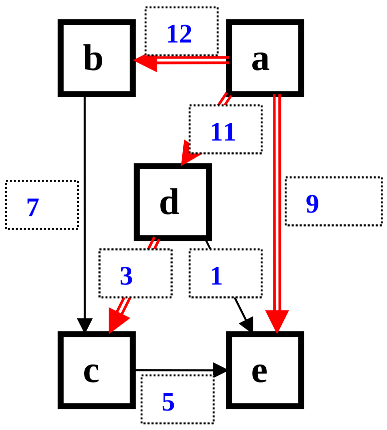
D: ¤ 12 § 11 9
E: ¤ a a a a
L: ¤ 12 § 11 9



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**

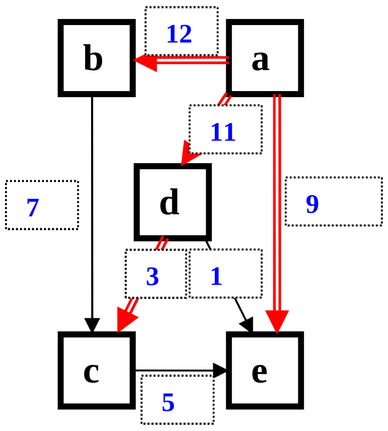
D: ¤ 12 14 11 9
E: ¤ a d a a
L: ¤ 12 3 11 9



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

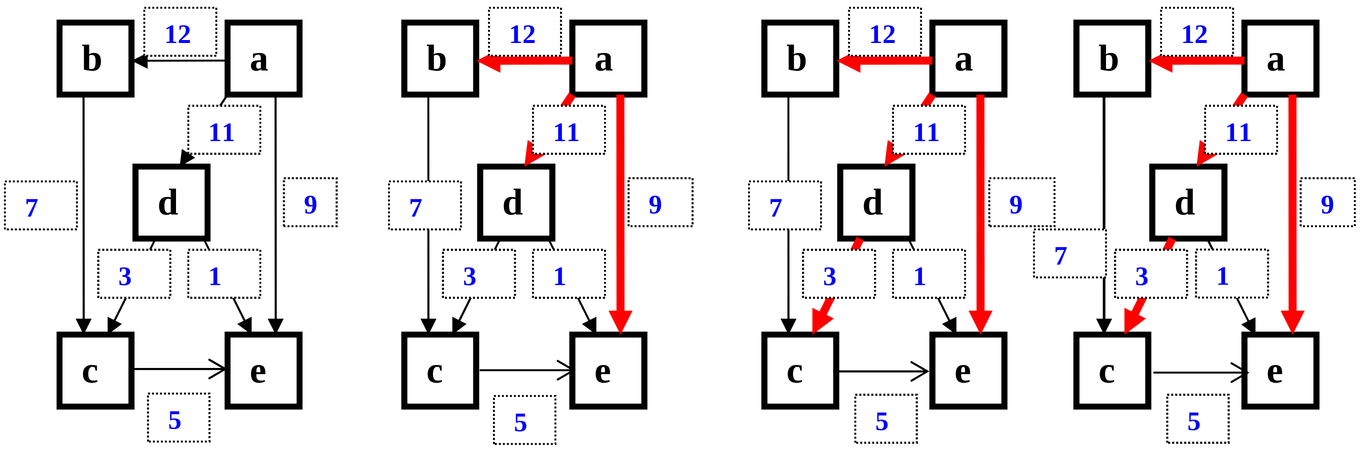


This is the final result.

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!



Question 7 – common mistakes

1. Not reading the question carefully enough.
2. Not checking whether the graph was directed or undirected – it was directed! This is stated in the question!
3. Not drawing the graph AND the cost matrix in the beginning – this is not stated in the question but is understood – I have mentioned this in the lectures!
4. Not checking that the number of edges (7) was reflected in the graph and the cost matrix.
5. Not checking the DIRECTION of the edges – errors gave a different graph hence not what the question asked – hence fewer or no marks!
6. Not giving step-wise calculations i.e. “by inspection” results – again I have warned against this in the lectures. Examples were given in the lectures, revision notes and previous facits!

In general, common mistakes for all questions.

1. Not **reading the question carefully enough** – **carelessness** - which is in many cases why you failed the exam.
2. **Lack of detail** in the answer – especially in not giving stepwise calculations.
3. **Not knowing the material** – in a few cases I really wonder if you (a) have attended lectures and (b) read the course material.
4. **Not double checking results** e.g. that the rotated AVL-tree was in fact AVL

I also wonder how many students have read (and studied) the FACITs for previous exams!
I cannot give you much more help than these!

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)        error ➔➔➔➔ Heapify(A,1)     i.e. heapify the whole array
     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
```