FACIT TILL OMTENTAMEN I
DATASTRUKTURER OCH ALGORITMER DVG B03

160823 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 skulle ett alternativ för Floyds algoritm vara?

Apply Dijkstra's algorithm to each node in the graph.

(b) Vad är en rekursiv definition?

A definition which is PARTIALLY defined in terms of itself.

Example – a sequence:	**Sequence**	**::= Head Tail | empty**
	Head	**::= element**
	Tail	**::= Sequence**

(c) Vad är en rekursiv funktion?

A function which CONDITIONALLY calls itself

Example **int size(Seq) { return is_empty(Seq) 0 : 1 + size(Tail(Seq)); }**

(d) Ge en definition av ett set (en mängd)?

An unordered collection of unique elements.

(e) Vad är Big-Oh?

A performance indicator usually of the length of running time an algoritm requires as a function of the numberf of elements to be processed.
Example: Dijkstra O(n^2)

(f) Prims och Dijkstras algoritmer är ”giriga” algoritmer. Vad betyder girig?

Girig = greedy - meaning that the algorithm applies the LOCALLY best solution.

(g) Vad är skillnaden mellan ordnat och sorterat?

Ordnat – ordered means that the collection is viewed as a sequence and each element has a position as an attribute.
Example: an ordered tree – the children of the parent are viewed as a sequence.
(the alternative is to view the children as a set in which case the tree is unordered)

Sorterat – sorted – the elements of the collection are ordered according to their value-

Example –	**a sequence:**	**3 5 9 12 17**
	position:	**1 2 3 4 5**

(h) Vad gör topologisk sortering?

Given a DAG (Direcred Acyclic Graph), returns a PARTIAL ORDER of the graph (which is a sequence).

(i) Vad är en heap?
A data structure which if implemented as a binary tree, places the largest/smallest value of the parent and children (if they exist) in the parent. The largest/smallest value in the collection is thus in the root.
If implemented as an array, the largest/smallest value is in the first element.

(j) Hur fungerar operation heapify på en heap?

```
Heapify(A, i)
    l = Left(i)            // l = 2*i
    r = Right(i)           // l = 2*i + 1
    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
```

Or a textual explanation.

Totalt 5p

(2) ADT Sequence

Operationer på en sekvens kan implementeras antigen på **ett iterativt sätt** eller på **ett rekursivt sätt.** Om man implementerar operationerna på **ett iterativt sätt** används oftast två pekare, nämligen **pPrevious** och **pCurrent. pCurrent** pekar på det aktuella elementet i sekvensen och **pPrevious** pekar på det föregående elementet till **pCurrent** (om sådant existerar). **listStart** pekar på det första elementet i sekvensen.

Anta att sekvensen är storterad i stigande ordning och implementerad som en enkel länkade lista. **listref** är en pekaretyp som är en referens till **ett element** i sekvensen.

(a) **Skriv abstrakt iterativ (pseudo)kod** till 2 operationer **void link_in(listref pNew)** samt **void unlink**() där **pNew** är en pekare till det nya elementet som man ska sätta in i sekvensen mellan **pPrevious** och **pCurrent**. **void unlink()** tar bort **pCurrent** från sekvensen.

(2p)

(b) **Skriv abstrakt iterativ (pseudo)kod** till **add** (lägga till ett element) operation. Ge exempel av hur din kod fungerar (i) när man lägger till ett värde i början av sekvensen, (ii) när man lägger till ett värde i mitten av sekvensen och när man lägger till ett värde i slutet av sekvensen. Använd funktionen **void link_in(listref pNew).**
Skriv abstrakt iterativ (pseudo)kod till alla **hjälp funktioner** som du behöver.

(4p)

(c) **Skriv abstrakt iterativ (pseudo)kod** till operationer **find** (hitta ett element i sekvensen) samt **remove** (ta bort ett element från sekvensen). Visa hur du kan inkorporera operation **find** i operation **remove**. Använd funktionen **void unlink()** i remove.
Skriv abstrakt iterativ (pseudo)kod till alla **hjälp funktioner** som du behöver.

(4p)

Totalt 10p

The following help functions are required (you may assume these exist)

```
static int     get_value  (listref E)              { return E->value; }
static listref get_next   (listref E)              { return E->next;  }
static void    set_value  (listref E, int     v)   { E->value = v;    }
static void    set_next   (listref E, listref n)   { E->next  = n;    }

static listref create_e  (int fval)
{
  listref pNew = (listref) malloc(sizeof(listelem));

  set_value(pNew, fval);
  set_next (pNew, NULLREF);
  return pNew;
}

static int    is_empty(listref E) { return E == NULLREF; }
```

For link_in, unlink, add, find and remove, the following help functions are required

```
static listref get_curr_ref()          { return pCurrent;              }
static int    is_seq_empty()           { return is_empty(pCurrent);    }
static int    get_element_value()      { return get_value(pCurrent);   }
```

```
static void   get_seq_first() { pPrevious = NULLREF; pCurrent = liststart; }

static void   get_seq_next () {
  if (!is_seq_empty()) { pPrevious = pCurrent; pCurrent = get_next(pCurrent);}
  }
```

```
/* link_in  and unlink operations */

static void link_in(listref pNew) {
   if (!is_empty(pNew)) {
      set_next(pNew, pCurrent);
      if (is_empty(pPrevious)) liststart = pNew; else set_next(pPrevious, pNew);
   }
}

static void unlink() {
   if (!is_empty(pCurrent)) {
      if (is_empty(pPrevious)) liststart = get_next(pCurrent);
      else set_next(pPrevious,  get_next(pCurrent));
   }
}
```

```
/* add operation */

static void be_add_val(int fval)
{
   get_seq_first();
   while ((!is_seq_empty()) && (fval > get_element_value()))  get_seq_next();
   link_in(create_e(fval));
}
```

```
/* find operation */

static listref be_find_val(int fval)
{
   get_seq_first();
   while ((!is_seq_empty()) && (fval != get_element_value()))  get_seq_next();
   return get_curr_ref();
}
```

```
/* remove operation */

static void be_rem_val(int fval) { be_find_val(fval); unlink(); }
```

(3) **Algoritmer - Principer.** – **Använd gärna bilder** i Ditt svar till (a), (b) och (c) nedan.
Använd den oriktade grafen:

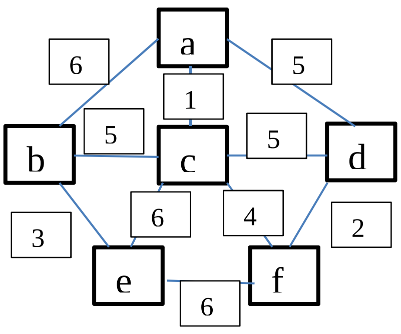
(a b 6), (a c 1), (a d 5), (b c 5), (b e 3), (c d 5), (c e 6), (c f 4), (d f 2), (e f 6)

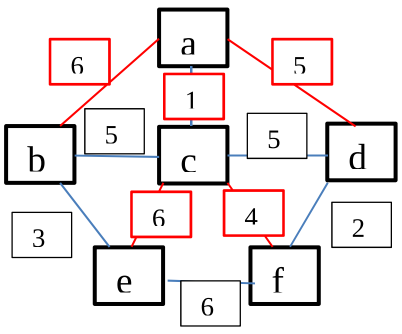
(a) **Beskriv principerna** bakom **Dijkstras** algoritm (**OBS:** **inte** Dijkstra_SPT). **(3p)**

Given a start node (x), choose the shortest edge from that node to a node y. Mark nodes x and y as visited. Check to see if there is a shorter path to the remaining (unvisited) nodes in the graph from x via y. If so, update the path lengths so far calculated. Repeat the process until all nodes have been visited.

The principle is:-

$$\begin{tikzcd} & y \arrow[dr] & \\ x \arrow[ur] \arrow[rr] & & z \end{tikzcd}$$





Graph **SPT**

Explanation - § == infinity i.e. no edge.

start node a – **cheapest (a c 1)** – initial costs [6, 1, 5, §, §] – visited {a, c} **unvisited {b, d, e, f}**
(a **c** b 6) not cheaper; (a c d 6) not cheaper; (a c e 7) **cheaper**; (a c f 5) **cheaper** ➔ [6, **1**, 5, **7**, **5**]

cheapest (a d 5) – visited {a, c, d} **unvisited {b, e, f}**
(a d b §) not cheaper; (a d e §) not cheaper; (a c f 7) not cheaper; no change ➔ [6, 1, 5, 7, 5]

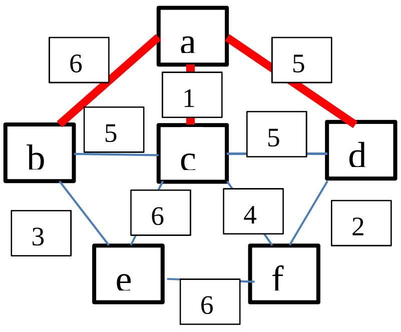
cheapest (a f 5) – visited {a, c, d, f} **unvisited {b, e}**
(a f b §) not cheaper; (a f e 11) not cheaper; no change ➔ [6, 1, 5, 7, 5]

cheapest (a b 6) – visited {a, b, c, d, f} **unvisited {e}**
(a b e 9) not cheaper; no change ➔ [6, 1, 5, 7, 5]

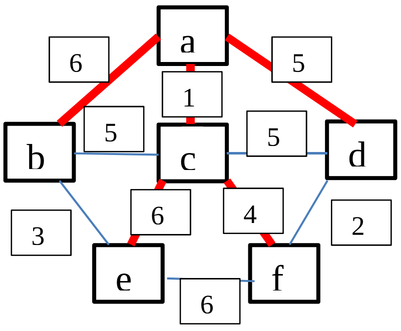
cheapest (a e 7) – visited {a, b, c, d, e, f} **unvisited { }** - empty – STOP!

SPT (a b 6) (a c 1) (a d 5) (a c 1 + c e 6 ➔ a e 7) (a c 1 + c f 4 ➔ a f 5)

In pictures (see above) this becomes



Initial configuration, start node a
[6, 1, 5, §, §] where § = infinite cost
a-1-c is the cheapest edge
check to see if there is a cheaper
PATH from a via c to nodes {b, d, e, f}
i.e. the unvisited nodes



from a via c to {b, d, e, f} gives
[6, 1, 5, 7, 5]
i.e. path lengths from a to {b, c, d, e, f}
the paths are:-
a-6-b, a-1-c, a-5-d, a-1-c-6-e, a-1-c-4-f

From [6, 1, 5, 7, 5] visited = {a, c}, unvisited = {b, d, e, f}. In order a-5-d is the cheapest path hence try to find cheaper paths from a-5-d to {b, e, f}
a-5-d-§-b (§) is NOT cheaper than a-6-b (6) since there is no edge d-b
a-5-d-§-e (§) is NOT cheaper than a-1-c-6-e (7) since there is no edge d-e
a-5-d-2-f (7) is NOT cheaper than a-1-c-4-f (5)
Hence there is no change from this iteration and visited = {a, c, d}, unvisited = {b, e, f}

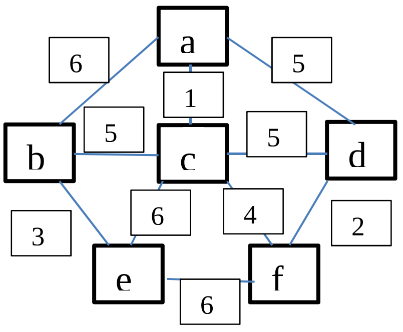
A similar argument applies to the next iteration: a-1-c-4-f (5) is the next cheapest path.
Visited = {a, c, d, f} unvisited = {b, e} Test paths to {b, e} via f
a-1-c-4-f-§-b (§) is NOT cheaper than a-6-b (6) since there is no edge f-b
a-1-c-4-f-6-e (11) is NOT cheaper than a-1-c-6-e (7)
No change

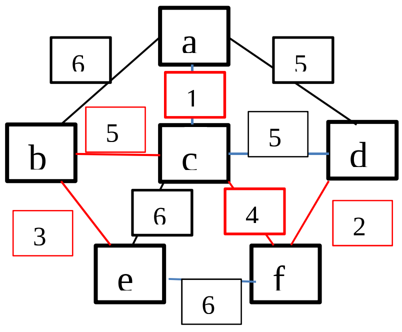
a-6-b (6) is the next cheapest path – visited = {a, c, d, f, b}, unvisited = {e}
a-6-b-3-e (9) is NOT cheaper than a-1-c-6-e (7)
No change

a-1-c-6-e (7) is the next cheapest path visited ={a, c, d, f, b, e} unvisited = { } i.e. empty.
STOP!
No change and now the result is ready. The paths are:-
a-6-b (6); a-1-c (1); a-5-d (5); a-1-c-6-e (7); a-1-c-4-f (5); i.e. [6, 1, 5, 7, 5]

(b) **Beskriv principerna** bakom **Prims** algoritm.

(3p)





Graph **MST**

The principles: given start node x mark as visited; note the edge values from x to the remaining nodes; this uses 2 arrays L for the edge lengths and C for the node name; find the shortest edge from x to y; mark y as visited; build a COMPONENT (x y) i.e. y is then added to the component (i.e. the visited nodes); now examine the edge costs from y to the remaining nodes; if this edge is cheaper, replace the current edge with this edge. Repeat for the unvisited nodes. The component grows node by node and cheaper edges replace those edges previously found as cheaper.

For the above graph - § == infinity

Start node a – visited {a} – unvisited {b, c, d, e, f} L = [6, 1, 5, §, §] C = [a, a, a, a, a]
Shortest edge (a **c** 1) – visited {a, c} – **unvisited {b, d, e, f}**
(c b 5) is cheaper ➔ L = [**5**, **1**, 5, §, §] C = [**c**, a, a, a, a]
(c d 5) not cheaper ➔ no change
(c e 6) is **cheaper** ➔ L = [**5**, **1**, 5, **6**, §] C = [**c**, a, a, **c**, a]
(c f 4) is **cheaper** ➔ L = [**5**, **1**, 5, **6**, **4**] C = [**c**, a, a, **c**, **c**]

Shortest edge (c **f** 4) – visited {a, c, f} – **unvisited {b, d, e}**
(f b §) not cheaper ➔ no change
(f d 2) is **cheaper** ➔ L = [**5**, **1**, **2**, **6**, **4**] C = [**c**, a, **f**, **c**, **c**]
(f e 6) not cheaper ➔ no change

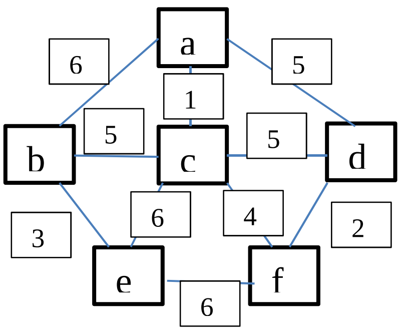
Shortest edge (f **d** 2) – visited {a, c, d, f} – **unvisited {b, e}**
(d b §) not cheaper ➔ no change
(d e §) not cheaper ➔ no change L = [**5**, **1**, **2**, **6**, **4**] C = [**c**, a, **f**, **c**, **c**]

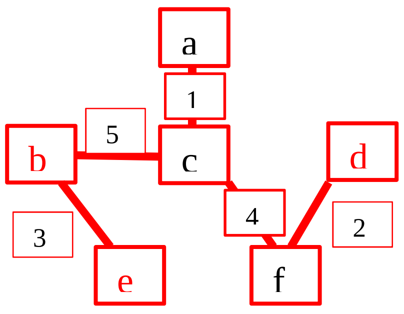
Shortest edge (c **b** 5) – visited {a, b, c, d, f} – **unvisited {e}**
(b e 3) is cheaper ➔ L = [**5**, **1**, **2**, **3**, **4**] C = [**c**, a, **f**, **b**, **c**]

Shortest edge (b **e** 3) – visited {a, b, c, d, e, f} – **unvisited {}** empty – STOP

Result L = [**5**, **1**, **2**, **3**, **4**] C = [**c**, a, **f**, **b**, **c**]

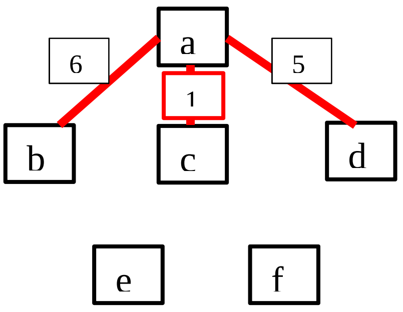
In pictures, this becomes:-

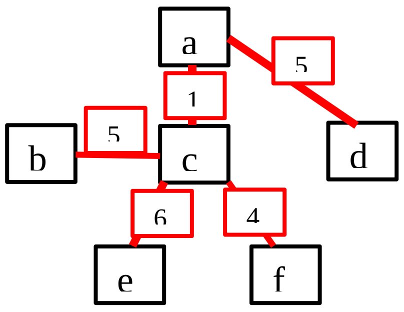




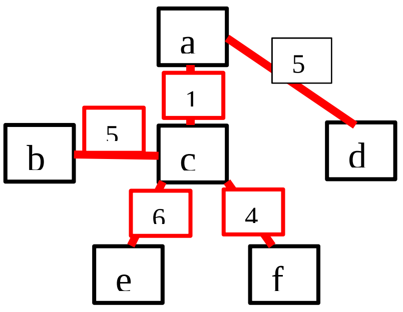
Graph **MST**

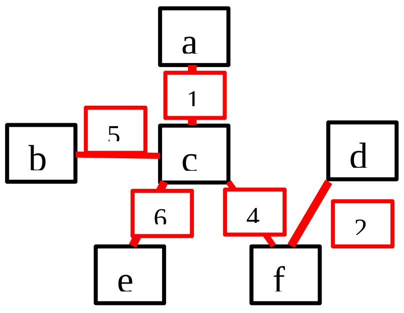
Visited = {a, c}, unvisited = {b, d, e, f}



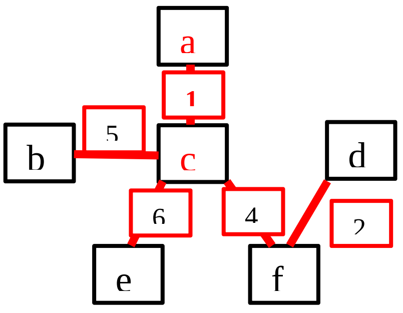


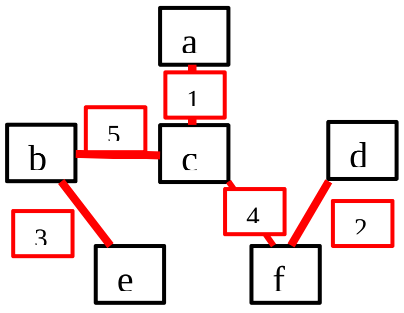
Visited = {a, c, f}, unvisited = {b, d, e}





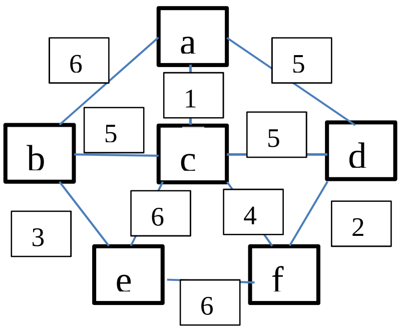
Visited = {a, c, f, d} unvisited = {b, e}
No change from d; change from b
Visited = {a, c, f, d, b}, unvisited = {e}

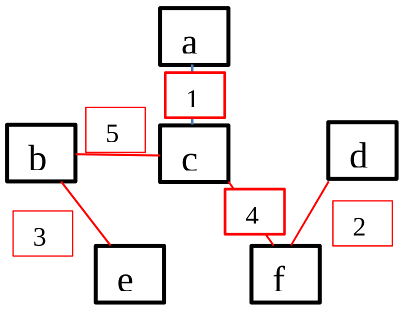




(c) **Beskriv principerna** bakom **Kruskals** algoritm.

(3p)





Graph **MST**

The principles: build a priority queue (PQ) with the edges, shortest edges first
⇨ **(a c 1), (d f 2), (b e 3), (c f 4), (a d 5), (b c 5), (c d 5), (a b 6), (c e 6), (e f 6)**

Each node in the graph becomes a component [a], [b], [c], [d], [e], [f]
Choose an edge from the PQ such that the edge connects 2 distinct components
until there is only one component – this is the MST

(a c 1) ➔ [a-c], [b], [d], [e], [f]	**- 5 components**
(d f 2) ➔ [a-c], [b], [d-f], [e]	**- 4 components**
(b e 3) ➔ [a-c], [b-e], [d, f]	**- 3 components**
(c f 4) ➔ [a-c, c-f, f-d], [b-e]	**- 2 components**
(a d 5) ➔ not chosen	**- a & d are in the same component**
(b c 5) ➔ [a-c, c-b, b-e, c-f, f-d]	**- 1 component (MST)**

(d) Vad är
i. ett **SPT** (Shortest Path Tree – KVT – kortaste väg träd) och
ii. ett **MST** (Minimal Spanning Tree – MST - Minimum Spänning Träd).

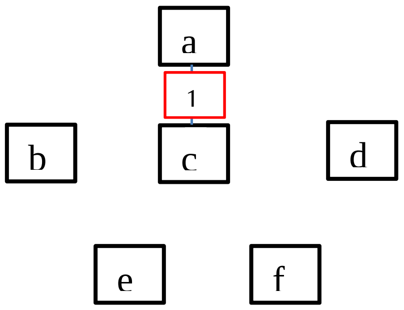
(1p)

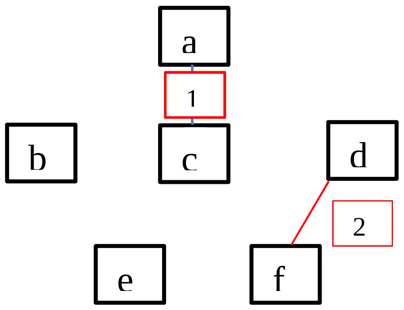
An SPT is a tree giving the shortest PATHS from a given node to the remaining nodes in the graph. The tree may be considered as a directed or undirected graph.

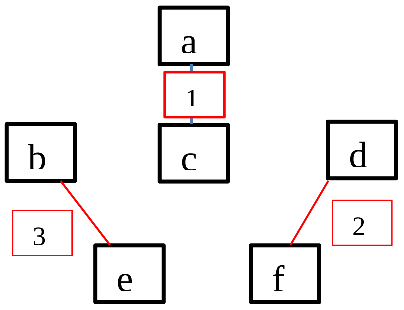
An MST is a tree which connects all the nodes in a graph in the cheapest way possible. This is a FREE TREE and is an undirected graph.

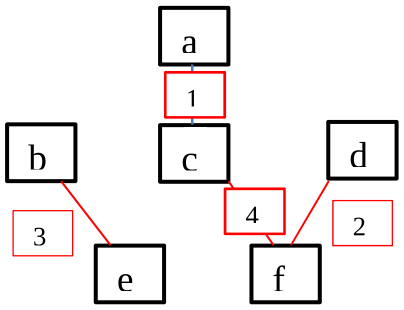
Totalt 10p

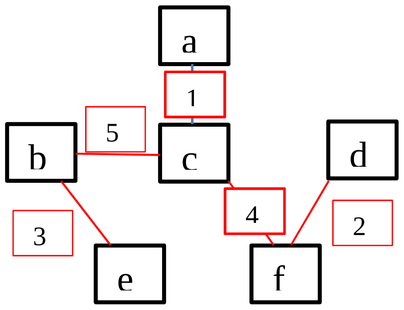
In Pictures, this becomes











(4) Hashning

Tillämpa både **linjär probning** samt **kvadratiskt probning** på följande sekvensen:

1, 27, 6, 87, 47, 7, 8, 17, 37, 67

Vilka problem kan uppstå? Hur lösa man dessa problem? Vilka aspekter bör man ta hänsyn till?

Anta att H(key) är key mod 10.

Svara ingående. Ange alla antagande.

5p

For the solution see:-

http://www.cs.kau.se/cs/education/courses/dvgb03/revision/index.php?hashingexs=1

(5) ADT Träd.

Beskriv utförligt alla aspekter av ADT:en **träd** som har presenterats under kursens lopp.

5p

Points to mention:-

1. General tree definition + conversion generat tree to binary tree
2. Tree properties – ordered and unordered
3. Binary tree (BT); properties; traversals: general, depth-first: in-, pre-, postorder; breadth first, operations (add, find, remove); applications – e.g. arithmetic expressions
4. BT recursive definition
5. BT properties: full, perfect, complete
6. BT & array representation
7. Binary Search Tree (BST); properties;
8. AVL Tree; properties; balancing operations; tree rotations
9. B-trees; Databases; balanced bushy trees (search path minimised)

(6) Diskussionsuppgift - Abstraktion

Diskutera utförligt begreppet ”abstraktion” och hur det har använts under denna kurs. Vilka sorters abstraktion finns? Varför är abstraktion så viktigt? Hur använder man abstraktion när man skriver kod till operationer på ADT:er?

Ge gärna exempel. Marks for a good discussion.

5p