FACIT TILL

TENTAMEN I

DATASTRUKTURER OCH ALGORITMER DVG B03

170117 kl. 14:00-19:00

Ansvarig Lärare: Donald F. Ross

Hjälpmedel: Inga. Algoritmerna finns i de respektive uppgifterna eller i bilagarna.

***** 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) I prestanda analys av **linjärsökning** (linear search) vilket resultat skulle förväntas för det slumpmässiga (random) fallet och det värsta fallet? Vad är förhållandet de emellan?

Result: Big-Oh is O(n) and O(n). For the average case the Big-Oh would be O(n/2) which becomes O(n). The times for the average should be half the times for the worst case on the assumption that the random values would average out to a search on the middle value.

(b) I prestanda analys av **bubble sort** vad är Big-Oh för det bästa, snitt samt värsta fallet?

O(n), O(n^2), O(n^2) respectively.

(c) Skriv **den rekursiva definitionen** av en sekvens.

S ::= H T | ¤; H ::= element; T ::= S; ¤ = empty

(d) Skriv **den rekursiva definitionen** av ett binärt träd.

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

(e) Ge **ett exempel** av an en ”**single left rotation**” i ett AVL-träd.

Case where add 9, add 10, add 11 gives a tree (*, 9, (*, 10, 11)) ➔ (9, 10, 11).

(f) Ge **ett exempe**l av an en ”**double right rotatio**n” i ett AVL-träd.

Case where add 11, add 9, add 10 gives a tree ((*,9,10), 11, *). The rotations required are an SLR to give ((9, 10, *), 11, *) + a SRR to give (9, 10, 11).

(g) Vad är invarianten i ett binärt sökträd?

The values in the left sub-tree are less than the value in the node and the values in the right sub-tree are greater that the value in the node.

(h) Vad är invarianten i en heap?

The values of the left- and right-child are less than the value in the parent.

(i) Vad är Big-Oh för hitta (find) i en heap?

O(1) (constant)

(j) I en heap array (som börjar med index 1), hur beräknar man höger och vänster barns position i arrayen?

The left-child index = (2*i) and right-child index is (2*i + 1) where i is the index of the parent.

Totalt 5p

(2) ADT Sekvens

Förklara ingående hur referenserna ”**pPrevious**” och ”**pCurrent**” fungerar i en iterativ implementation av operationerna **lägga till** (add), **ta bort** (remove) och **hitta** (find). Vilka ”**speciella fall**” måste man ta hand om?
Skriv (pseudo)kod till operationerna **lägga till**, **ta bort** samt **hitta** för att illustrera Ditt svar.

5p

See the sequence code example in the lab.

pPrevious and pCurrent a pair of references that move along the sequence. Two navigation operations control these.

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

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

New elements are added between pPrevious and pCurrent

```
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 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 is implemented as follows – pCurrent refers to the element found or is NULLREF if not found.

```
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 is implemented using find + unlink

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

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

The special cases are at the beginning and the end.

For add and remove at the beginning, pPrevious will be == NULLREF and a liststart reference will be set to the new element..

In linkin - if (is_empty(pPrevious)) liststart = pNew;

In unlink - if (is_empty(pPrevious)) liststart = get_next(pCurrent);

At the end, pPrevious refers to the last element and pCurrent is NULLREF.

In a singly linked sequence this is taken care of by the above code.

In a doubly-linked sequence for add there would be a listend reference which would be set to reference to the new element.

See http://www.cs.kau.se/cs/education/courses/dvgb03/lectures/IterRec.pdf

And http://www.cs.kau.se/cs/education/courses/dvgb03/p5/index.php?studyplan=1

(3) Abstraktion

(a) I labben om prestandamätning fanns det fem storleksfall som skulle mätas, nämligen 1024, 2048, 4096, 8196 och 16384. Tre sorteringsalgoritmer (bubble sort, insertion sort samt quicksort) och två sökningsalgoritmer (linear search, bibary search) skulle mätas.

En student har implementerat labben med följande **FEPerf.h** och **BEPerf.h filer**.

```
void FE_Do_All();                                   /* FEPerf.h   */
void FE_RunTest(testtype ftest);
```

```
unsigned int BE_RunTest(testtype ftest, int size);      /* BEFPerf.c */
```

1. Förklara (med text) hur ”**back-enden**” kommer att fungera. 1p

The back-end function is called 5 times from the front-end function with a ”code” for the test to be executed. The back-end function will

a. **Initiate an array according to the test code (using a switch statement)**
b. **Start the timer**
c. **Execute the test according to the test code (using a switch statement)**
d. **Stop the timer**
e. **Return the result (elapsed time)**

2. Skriv (pseudo)kod till functionen ”**FE_RunTest(testtype ftest)**” på så sätt att (pseudo)koden är ganska generell – dvs att (pseudo)koden kan hantera flera fall än dem som beskrivs ovan.
Koden ska följa principerna för abstrakt programmering.
2p

```
void FE_RunTest(testtype ftest) {
   int size = STARTSIZE, i;
   for (i=0; i<NUMTEST; i++) {
      results[i] = BE_RunTest(ftest, size);
      size = size * 2;
      }
   ptable(ftest);
}
```

Where: **STARTSIZE is 1024** **- this is doubled for each test**
NUMTEST is 5 **- the number of tests**
ptable(ftest) **- chooses the right table display for ftest (using a switch statement)**
results[NUMTEST] **- holds the time for each test**

The code is thus made more general than the specific requirements in the specification statement.

(b) En annan form av abstraktion är **implementeringsabstraktion** där man försöka att gömma implementationsdetaljerna såsom arrayer eller strukturer och pekare så mycket som man kan så att resten av implementationen **följer principerna för abstrakt programmering**. Beskriv hur denna process fungerar. Vilka funktioner använder man sig av för att gömma implementationsdetaljerna?

2p

Totalt 5p

Again, this may be seen in the sequence code example.

The code where the implementation is ”hidden” is:-

```
/* list element reference type        */
typedef  struct listelem * listref;

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;
}
```

I.e.

1. **The definition of the reference type**
2. **The get and set functions, 1 per attribute**
3. **The create a new element fmunction**

Marks for a good explanation / description.

(4) Heap algoritmer

(a) Titta på koden nedan för **Build & Heapify**. Tillämpa koden på denna sekvens:

1, 2, 3, 4, 7, 8, 9, 10, 14, 16

Visa varje steg i Dina beräkningar inklusive de rekursiva anropen.
Förklara hur koden fungerar.

3p

```
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
```

See the revision notes for the answer to this question.

Input: 1 2 3 4 7 8 9 10 14 16
Array size = 10
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 = 1 2 3 4 7 8 9 10 14 16
i = 5; (value 7) **l = 10;** (value 16) **r = 11;** (node does not exist) **largest = 10;** (value 16)
largest = 10 (value 16) largest != i hence **swap A[5]** and **A[10]** giving **1 2 3 4 16 8 9 10 14 7**
Heapify(A, 10) has no effect on A (A[10] is a leaf node)

the call to Heapify(A, 4)

i = 4 A = 1 2 3 4 16 8 9 10 14 7
i = 4; (value 4) **l = 8;** (value 10) **r = 9;** (value 14) **largest = 9;** (value 14)

largest = 9 (value 14) largest != i hence **swap A[4]** and **A[9]** giving **1 2 3 14 16 8 9 10 4 7**
Heapify(A, 9) has no effect on A (A[9] is a leaf node)

the call to Heapify(A, 3)

i = 3 A = 1 2 3 14 16 8 9 10 4 7
i = 3; (value 3) **l = 6;** (value 8) **r = 7;** (value 9) **largest = 7;** (value 9)
largest = 7 (value 9) largest != i hence **swap A[3]** and **A[7]** giving **1 2 9 14 16 8 3 10 4 7**
Heapify(A, 7) has no effect on A (A[7] is a leaf node)

the call to Heapify(A, 2)

i = 2 A = 1 2 9 (value 14) (value 16) **8 3 10 4 7**
i = 2; (value 2) **l = 4;** (value 14) **r = 5;** (value 16) **largest = 5;** (value 16)
largest = 5 (value 16) largest != i hence **swap A[2]** and **A[5]** giving **1 16 9 14 2 8 3 10 4 7**

1st RECURSIVE CALL Heapify(A, 5) (deal with the sub-tree to which 2 was swapped)

the (recursive) call to Heapify(A, 5)

i = 5 A = 1 16 9 14 2 8 3 10 4 7
i = 5; (value 2) **l = 10;** (value 7) **r = 11;** (node does not exist) **largest = 10;** (value 7)
largest = 10 (value 7) largest != i hence **swap A[5]** and **A[10]** giving **1 16 9 14 7 8 3 10 4 2**
Heapify(A, 10) has no effect on A (A[10] is a leaf node)
END OF THE 1st RESURSIVE CALL

the call to Heapify(A, 1)

i = 2 A = 1 16 9 14 7 8 3 10 4 2
i = 1; (value 1) **l = 2;** (value 16) **r = 3;** (value 9) **largest = 2;** (value 16)
largest = 2 (value 16) largest != i hence **swap A[1]** and **A[2]** giving **16 1 9 14 7 8 3 10 4 2**

1st RECURSIVE CALL Heapify(A, 2) (deal with the sub-tree to which 1 was swapped)

the (recursive) call to Heapify(A, 2)

i = 2 A = 16 1 9 14 7 8 3 10 4 2
i = 2; (value 1) **l = 4;** (value 14) **r = 5;** (value 7) **largest = 4;** (value 14)
largest = 4 (value 14) largest != i hence **swap A[2]** and **A[4]** giving **16 14 9 1 7 8 3 10 4 2**

2nd RECURSIVE CALL Heapify(A, 4) (deal with the sub-tree to which 1 was swapped)

the (recursive) call to Heapify(A, 4)

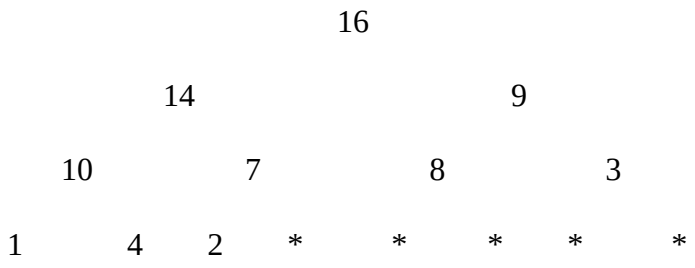
i = 4 A = 16 14 9 1 7 8 3 10 4 2
i = 4; (value 1) **l = 8;** (value 10) **r = 9;** (value 4) **largest = 8;** (value 10)
largest = 8 (value 10) largest != i hence **swap A[4]** and **A[8]** giving **16 14 9 10 7 8 3 1 4 2**
Heapify(A, 8) has no effect on A (A[8] is a leaf node)

END OF THE 2nd RECURSIVE CALL

END OF THE 1st RECURSIVE CALL

Final Result
Array ==> 16 14 9 10 7 8 3 1 4 2

Exercise: draw the tree to confirm the result.



See lecture and revision notes:-

http://www.cs.kau.se/cs/education/courses/dvgb03/lectures/heaps_&_PQs.pdf

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

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

```
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
```

Totalt 5p

(b) Titta på koden nedan för **Add**. Förklara **principen** bakom koden. Visa hur koden fungerar genom att lägga till (add) 42 till resultatet från (a) ovan.

2p.

Exercise: draw the corresponding trees for each instance of the array.

Add 42 - Input: 16 14 9 10 7 8 3 1 4 2

A = 16 14 9 10 7 8 3 1 4 2 A.size = 10
Increase the size of A giving A = 16 14 9 10 7 8 3 1 4 2 § A.size = 11 and **i = 11**
§ represents the "hole" (i.e. node) for the new value

while iterations

i = 11; parent(i) = 5; v = 42; A[5] = 7; and 7 is less than 42 so move the parent value and move the value of the parent to **node i** giving 16 14 9 10 **7** 8 3 1 4 2 **7** - (§ has value 7)
set i to 5

i = 5; parent(i) = 2; v = 42; A[2] = 14; and 14 is less than 42 so
move the value of the parent to **node i** giving 16 **14** 9 10 **14** 8 3 1 4 2 7 - (§ has value 14)
set i to 2

i = 2; parent(i) = 1; v = 42; A[1] = 16; and 16 is less than 42 so
move the value of the parent to **node i** giving **16 16** 9 10 14 8 3 1 4 2 7 - (§ has value 16)
set i to 1

END OF WHILE ITERATION
ADD THE NEW VALUE - 42 giving **42** 16 9 10 14 8 3 1 4 2 7
Final Result
Array ==> 42 16 9 10 14 8 3 1 4 2 7

Exercise: draw the tree to confirm the result.

(5) Hashning

I hashning har fyra metoder för kollisionshantering presenterats, nämligen

1) Separate chaining (open hashing)
2) Open addressing (closed hashing) f(i) = i (linear probing)
3) Quadratic probing f(i) = i * i
4) Double hashing f(i) = i * H_2(key)

där H_2(key) = 7 – (key mod 7)

Metoderna (2), (3) och (4) hanterar kollisioner med H(key) + f(i) där 'i' representerar antalet kollisioner. Anta att **H(key) = key mod 10**. **Hash space = array H[10].**

Tillämpa dessa 4 metoder på sekvensen **4, 36, 44, 5, 7, 64, 24** och **visa varje steg** i Dina beräkningar.

3p

Vilka problem kan förekomma med dessa metoder?

2p

Totalt 5p

See lecture notes
http://www.cs.kau.se/cs/education/courses/dvgb03/lectures/hashing.pdf

Separate Chaining (Open Hashing)

1. Add 4 to slot 4 (4 mod 10)
2. Add 36 to slot 6 (6 mod 10)
3. Add 44 to slot 4 (44 mod 10) → **collision** → chain to slot 4
4. Add 5 to slot 5 (5 mod 10)
5. Add 7 to slot 7 (7 mod 10)
6. Add 64 to slot 4 (64 mod 10) → **collision** → chain to slot 4
7. Add 24 to slot 4 (24 mod 10) → **collision** → chain to slot 4

0		
1		
2		
3		
4	**4**	**→ 44 → 64 → 24**
5	**5**	
6	**36**	
7	**7**	
8		
9		

Open addressing (closed hashing) **f(i) = i**

1. Add 4 to slot 4 (4 mod 10)
2. Add 36 to slot 6 (36 mod 10)
3. Add 44 to slot 4 (44 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
4. Add 5 to slot 5 (5 mod 10) → **collision** find next free slot 5 + f(1) = 5+1 = 6
   → **collision** find next free slot 5 + f(2) = 5+2 = 7
5. Add 7 to slot 7 (7 mod 10) → **collision** find next free slot 7 + f(1) = 7+1 = 8
6. Add 64 to slot 4 (64 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
   → **collision** find next free slot 4 + f(2) = 4+2 = 6
   → **collision** find next free slot 4 + f(3) = 4+3 = 7
   → **collision** find next free slot 4 + f(4) = 4+4 = 8
   → **collision** find next free slot 4 + f(5) = 4+5 = 9
7. Add 24 to slot 4 (24 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
   → **collision** find next free slot 4 + f(2) = 4+2 = 6
   → **collision** find next free slot 4 + f(3) = 4+3 = 7
   → **collision** find next free slot 4 + f(4) = 4+4 = 8
   → **collision** find next free slot 4 + f(5) = 4+5 = 9
   → **collision** find next free slot 4 + f(6) = 4+6 = 0*

* (10 mod 10)

15 collisions

slot	+4	+36	+44	+5	+7	+64	+24
0							24
1							
2							
3							
4	4	4	4	4	4	4	4
5			44	44	44	44	44
6		36	36	36	36	36	36
7				5	5	5	5
8					7	7	7
9						64	64

Quadratic Probing **f(i) = i * i**

1. Add 4 to slot 4 (4 mod 10)
2. Add 36 to slot 6 (36 mod 10)
3. Add 44 to slot 4 (44 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
4. Add 5 to slot 5 (5 mod 10) → **collision** find next free slot 5 + f(1) = 5+1 = 6
   → **collision** find next free slot 5 + f(2) = 5+4 = 9
5. Add 7 to slot 7 (7 mod 10)
6. Add 64 to slot 4 (64 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
   → **collision** find next free slot 4 + f(2) = 4+4 = 8
7. Add 24 to slot 4 (24 mod 10) → **collision** find next free slot 4 + f(1) = 4+1 = 5
   → **collision** find next free slot 4 + f(2) = 4+2 = 8
   → **collision** find next free slot 4 + f(3) = 4+9 = 3*

* (13 mod 10)

8 collisions **a reduction of 7 from 15**

slot	+4	+36	+44	+5	+7	+64	+24
0							
1							
2							
3							24
4	4	4	4	4	4	4	4
5			44	44	44	44	44
6		36	36	36	36	36	36
7					7	7	7
8						64	64
9				5	5	5	5

Double Hashing **f(i) = i * H_2(key) where H_2(key) = 7- (key mod 7)**

1. Add 4 to slot 4 (4 mod 10)
2. Add 36 to slot 6 (36 mod 10)
3. Add 44 to slot 4 (44 mod 10) → **collision** find next free slot 4 + f(1) = 4+5 = 9 (a)
4. Add 5 to slot 5 (5 mod 10)
5. Add 7 to slot 7 (7 mod 10)
6. Add 64 to slot 4 (64 mod 10) → **collision** find next free slot 4 + f(1) = 4+6 = 0* (b)
7. Add 24 to slot 4 (24 mod 10) → **collision** find next free slot 4 + f(1) = 4+4 = 8 (c)

(a) (44 mod 7) = 2, (7-2) = 5, (1 * 5) = 5
(b) (64 mod 7) = 1, (7-1) = 6, (1 * 6) = 6 *(10 mod 10) = 0
(c) (24 mod 7) = 3, (7-3) = 4, (1 * 4) = 4

3 collisions **a reduction of 5 from 8 (quadratic probing) and 12 from 15 (linear probing)**

slot	+4	+36	+44	+5	+7	+64	+24
0						64	64
1							
2							
3							
4	4	4	4	4	4	4	4
5				5	5	5	5
6		36	36	36	36	36	36
7					7	7	7
8							24
9			44	44	44	44	44

Potential problems with many collisions

(1) Separate chaining (Open Hashing)
i. No collisions BUT
ii. the **list** gets longer and the search time goes from O(1) to O(n)

(2) Open Adressing (Closed Hashing) – Linear Probing
i. the **search** gets longer and the search time goes from O(1) to O(n)
ii. primary clustering (sequence from a given slot)

(3) Quadratic Probing
i. the **search** gets longer and the search time goes from O(1) to O(n)
ii. secondary clustering
iii. when the load (number of occupied slots) >50% this method may no longer work!

(4) Double Hashing
i. Disadvantage – 2 hash functions must be calculated

(6) Träd

I hantering av **binära sökträd** är operation ”**ta bort**” (remove) den svåraste operationen att implementera. Vilka aspekter måste man ta hänsyn till när man implementerar denna operation?
Skriv (pseudo)kod till en **rekursiv implementation** av ta bort från ett binärt sökträd.
Du får använda ”**hjälpfuktioner**” i Din (pseudo)kod.

5p

See http://www.cs.kau.se/cs/education/courses/dvgb03/lectures/NTrees_2.pdf

Since this is part of a lab exercise, actual code will not be presented.

The recursive definition of a BT gives a pattern for the code

1) **Stop condition** **(empty, non-recursive)**
2) **Left child** **(recursive)**
3) **Right child** **(recursive)**
4) **Node** **(non-empty, non-recursive)**

Remove then becomes

Stage 1:

```
BST remove(BST T, int v){

if IsEmpty(T)   then return T
if v < value(T) then return cons(remove(LC(T), v), T, RC(T))
if v > value(T) then return cons(LC(T), T, remove(RC(T), v))
/* v = value(T) */   return remove_Root(T);
}
```

Stage 2: remove_Root(T)

There are 4 cases

1) **(0 0)** **no left child & no right child** **return ¤ (empty)**
2) **(1 0)** **no right child** **return left child**
3) **(0 1)** **no left child** **return right child**
4) **(1 1)** **both left & right child**

Stage 3: step (4) of stage 2

1) **The (local) root node value may be replaced by**
   a. **The maximum value in the left sub-tree OR**
   b. **The minimum value in the right sub-tree**
2) **Remember to remove the node that replaces the root node!**
3) **Another aspect is how to choose max(left) or min(right)**
   a. **If you want to keep the BST balanced the choose the bigger of the left/right sub-tree**

(7) Graf algoritmer

(a) Tillämpa **Dijkstra_SPT algoritm** på den oriktade grafen nedan.

- **Visa varje steg** i Dina beräkningar.
- **Rita en bild av grafen och ge kostnadsmatrisen**.
- **Börja med nod a**.
- **Rita en bild av resultatet**.

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

```
Dijkstra_SPT ( a ) {
   S = {a}
   for (i in V-S) {
      D[i] = C[a, i]
      E[i] = a
      L[i] = C[a, i]
      }
   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]
      }
   }
}
```

3p

See the revision notes on Dijkstra.

http://www.cs.kau.se/cs/education/courses/dvgb03/revision/index.php?SPTex2=1

See also

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

ugraph1 example
http://www.cs.kau.se/cs/education/courses/dvgb03/lab_info/newGraphLab/ugraph1.pdf

http://www.cs.kau.se/cs/education/courses/dvgb03/lab_info/newGraphLab/ugraph1.out

(b) Tillämpa **Prims algoritm** på den oriktade grafen nedan.

- **Visa varje steg** i Dina beräkningar.
- **Rita en bild av grafen och ge kostnadsmatrisen**.
- **Börja med nod a**.
- **Rita en bild av resultatet**.

(a-6-b), (a-1-c), (a-5-d), (b-5-c), (b-3-e), (c-5-d), (c-6-e), (c-4-f), (d-2-f), (e-6-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;
          }
       }
}
```

3p

3p

See the revision notes on Prim.

http://www.cs.kau.se/cs/education/courses/dvgb03/revision/PrimExample.pdf
http://www.cs.kau.se/cs/education/courses/dvgb03/revision/PrimExa.pdf
http://www.cs.kau.se/cs/education/courses/dvgb03/revision/index.php?PrimEx=1

See also

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

ugraph1 example
http://www.cs.kau.se/cs/education/courses/dvgb03/lab_info/newGraphLab/ugraph1.pdf
http://www.cs.kau.se/cs/education/courses/dvgb03/lab_info/newGraphLab/ugraph1.out

(c) **Förklara ingående** principen bakom Prims algoritm.

OBS: Skriv inte en rad för rad ”**översättning**” från koden till svenska (engelska) utom ge **en beskrivning** av hur algoritmen fungerar vid varje steg. Använd gärna bilder i Din beskrivning.

Sedan ge en sammanfattning av principen i två eller tre meningar.

4p

Totalt 10p

Prim’s calculates the Minimal Spanning Tree for an undirected graph.

Principles

1. **Choose a node a - this node is the first in a component**
2. **Choose the shortest EDGE from a to any other node b**
3. **Each chosen node ias marked as visited**
4. **Repeat from b to the remaining unvisited nodes**
5. **The result is an MST**

The short version:- choose a start node, the shortest edge to another node and that node to build a component. Add the node and edge with the shortest edge from the component. Repeat until all nodes have been visited.

The even shorter version:-
Prim’s builds a component and repeatedly adds nodes with the shortest edge from the component until all nodes have been visited.