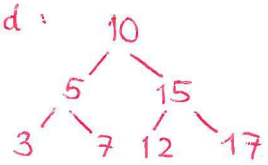
TENTAMEN 20140814 DVA104 LÖSNINGSFÖRSLAG

UPPGIFT 1

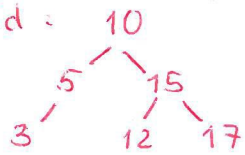
a) Divide & conquer innebär att man delar upp problemet i delproblem tills delproblemet är så pass trivialt (litet) att det enkelt kan lösas, därefter sätts lösningarna på delproblemen ihop till en lösning på originalproblemet. Exempel på algoritmer är MergeSort och Quicksort.

b) Ett fullt binärt träd betyder att den understa nivån har löv på samtliga platser.

ex. fullt träd :



ex icke fullt träd :



c) Basfallet behövs i en rekursiv funktion för att den ska kunna avslutas, annars fastnar funktionen i en loop av eviga rekursiva anrop (tills programmet kraschar).

d) En algoritm är stabil om den bibehåller den relativa ordningen mellan element med lika nyckel.

e) Stack är ett annat namn på en LIFO-kö (Last In First Out)

g) För att man ska kunna utföra binärsökning på en mängd så måste mängden vara linjär och sorterad.

UPPGIFT 2

a) Basfallet : return -1; som gör att de rekursiva anropen avbryts och return 1; funktionen börjar returnera tillbaka.

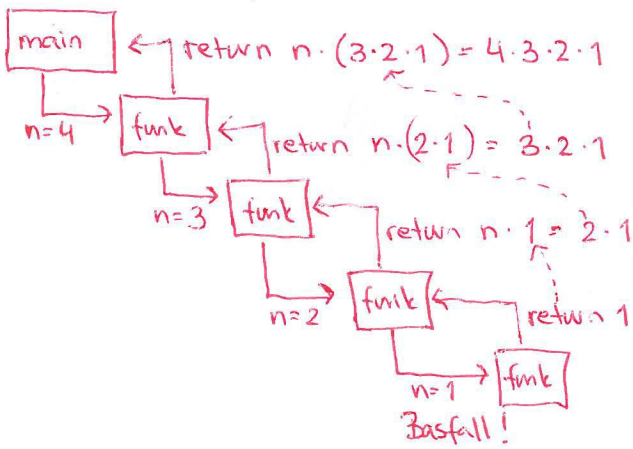
Rekursiva fallet : return n * funk(n-1); som anropar funktionen igen (rekursivt)

Villkoret : if(n<0) if(n==0 || n==1) : som bestämmer om det är det rekursiva fallet eller basfallet som ska köras.

Förändring av villkoret : funk(n-1) : som ser till att något av basfallen tillslut nås.

b) Funktionen beräknar fakulteten av det i anropet angivna värdet på n.

ex
funk(4)



TENTAMEN 20140814 DVA104
LÖSNINGSFÖRSLAG

UPPGIFT 2 FORTS.

c)
```
int funk(int n)
{
    int sum = 1;
    if (n < 0)
        return -1;
    else
    {
        for( ; n>0; n--)
            sum *= n;
    }
    return sum;      // 0! = 1
}
```

UPPGIFT 3

a)
```
int arrQueue[10] = { };  // kö/array med plats för 10 heltal, initieras till nollvärden.
int front=0, back=0;     /* tex return front visar det element som legat längst tid i kön, back visar den plats där nästa element ska läggas in. När kön är tom pekar båda på index 0. När front == back anses kön vara tom.
```

b)
```
void deQueue (int *Queue, int *front)
{
    Queue[*front] = 0; //nollvärde
    *front = (*front +1)%10; // flytta front till nästa (den som nu legat längst i kön) och snurra om det behövs
}
```

Anrop:
```
if (front != back) // vi kan inte ta bort någonting från en tom kö (front == back) men kan också testa if(!QueueEmpty) och anta att det finns en sådan funktion
    deQueue (arrQueue, &front);
```
Alternativt lägga till en räknare som håller reda på antalet element i kön och testa den.

TENTAMEN 20140814 DVA104
LÖSNINGSFÖRSLAG

UPPGIFT 4

a) Bubblesort: Går igenom arrayen N-1 ggr, där N är antal element, och jämför de två närliggande, ligger de i inbördes felordning byter de plats.

Insertionsort: "Dela" mängden/arrayen i två delar (mha en pekare/variabel), den vänstra delen innehåller från början 1 element och är den sorterade delen, den högra delen håller från början resterande element och anses vara osorterad. Algoritmen tar det första elementet från H och sätter det på rätt plats i V, samt flyttar fram gränsen mellan V och H. Detta görs tills H är tom.

Selectionsort: "Dela" mängden/arrayen i två delar, alla element ligger från början i den högra (osorterade) delen. Leta upp det minsta i H, placera det först i H och flytta gränsen mellan den sorterade och den osorterade delen (alltså hamnar elementet sist i den sorterade delen. Detta upprepas tills H endast innehåller ett element (det största i hela mängden)

b)

Bubblesort: Best case: $O(n)$ - sorterad mängd i den optimerade versionen
$O(n^2)$ - den ooptimerade versionen (n-1 rundor, i varje runda n-1 jämförelser, och 0 byten om mängden är sorterad)

Worst case: $O(n^2)$ - bakåtsorterad (n-1 rundor, i varje runda n-1 jämförelser (och byten))

Insertionsort: Best case: $O(n^2)$ - sorterad om vi letar från vänster i den sorterade delen. => för alla i H, leta igenom alla i V för att hitta rätt plats.
$O(n)$ - sorterad om vi letar från höger i den sorterade delen => för alla i H hittar vi direkt rätt plats i V.

Worst case: $O(n^2)$ - för alla i H, flytta alla i V för att skapa en lucka på rätt plats. Bakåtsorterad.

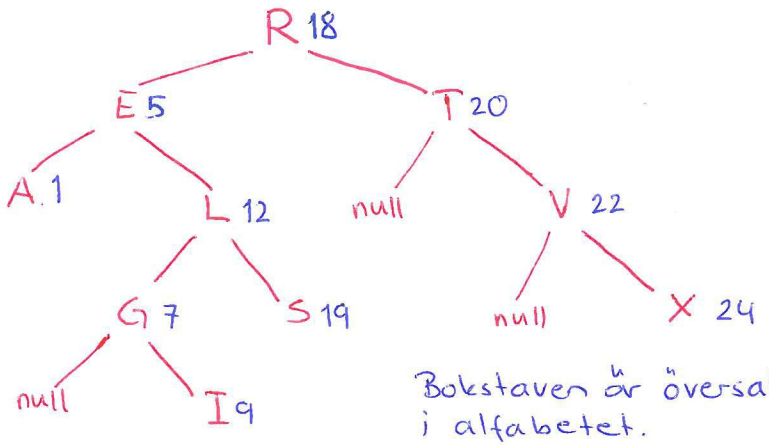
Selectionsort: Best case = Worst case: $O(n^2)$ - samma arbete genomförs oavsett hur mängden är sorterad från början.

Om mängden är sorterad eller nästan sorterad bör en optimerad bubblesort eller en insertionsort som letar från höger i V användas (av dessa 3 algoritmer)

c) Bubblesort kan optimeras genom att
1: avsluta algoritmen om den gått en hel runda utan att genomföra några byten.
2: avbryta antalet jämförelser ett steg tidigare för varje varv. För varje varv kan man vara säker på att ett nytt element fått sin rätta plats, detta/dessa behöver inte jämföras.

TENTAMEN 20140814 DVA104
LÖSNINGSFÖRSLAG

UPPGIFT 5



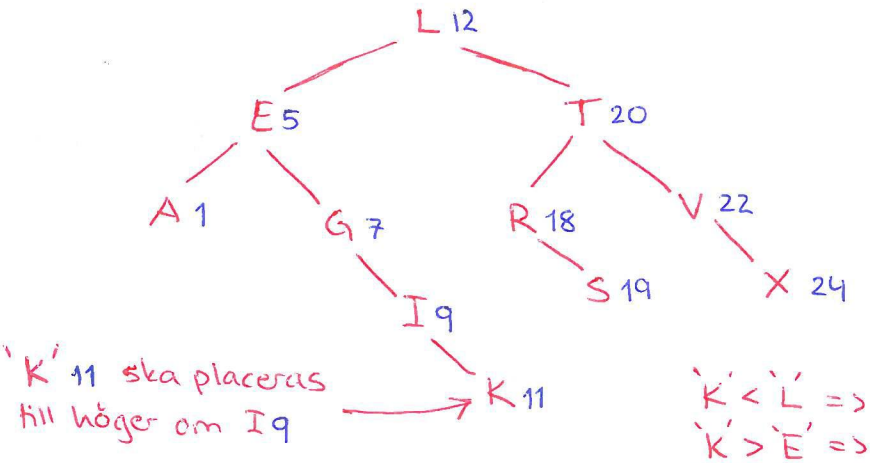
Bokstaven är översatt till nummer i alfabetet.

a) Binärt: Ja - varje nod har max två barn.
Sökträd: Nej - S ligger inte sorterad (bör ligga till höger om R)

b) Preorder = hantera nod, gå vänster, gå höger
R E A L G I S T V X

c) Balanserat: Nej - Faktisk djup är 5 (R-E-L-G-I)
Teoretiskt djup är 4 ($\log_2(10)+1 = 4$)

d) Trädet omritat till ett balanserat binärt sökträd:



'K' 11 ska placeras till höger om I 9

'K' < 'L' => vänster
'K' > 'E' => höger
'K' > 'G' => höger
'K' > 'I' => höger

e)
```
typedef struct Node{
        char data;
        struct Node *leftChild;
        struct Node *rightChild;
}Node;
```

UPPGIFT 5 FORTS

f)

```
Node *insertSorted (Node *subTree, char data)
{
    if (subTree == NULL)
    {
        Node * newNode = (Node*) malloc (sizeof(Node));
        if (newNode != NULL)
        {
            newNode -> data = data;
            newNode -> leftChild = NULL;
            newNode -> rightChild = NULL;
            subTree = newNode;
        }
        else
            printf ("error");
    }
    else
    {
        if (data < subTree -> data)
            subTree -> leftChild = insertSorted (subTree -> leftChild, data);
        else
            subTree -> rightChild = insertSorted (subTree -> rightChild, data);
    }
    return subTree;
}
```

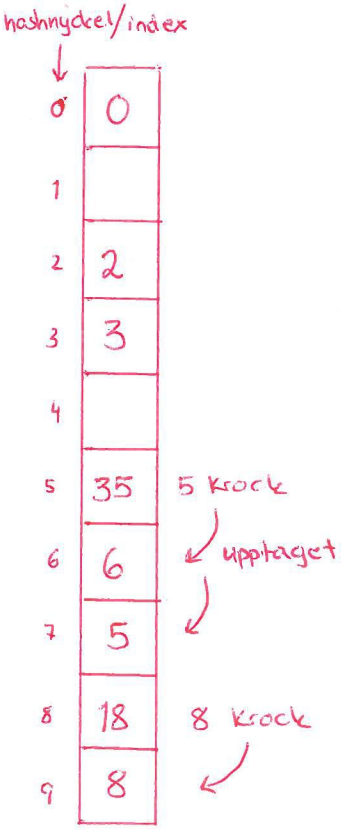
g) Funktionen som tar reda på djupet på trädet bör vara rekursiv då
1: träd (binära) naturligt är en rekursiv datastruktur
2: alla deltråds djup måste mätas och jämföras för att ta reda på vilket som är djupast. Man kan inte veta från början hur många deltråd som finns i trädet.

TENTAMEN 20140814 DVA104
LÖSNINGSFÖRSLAG

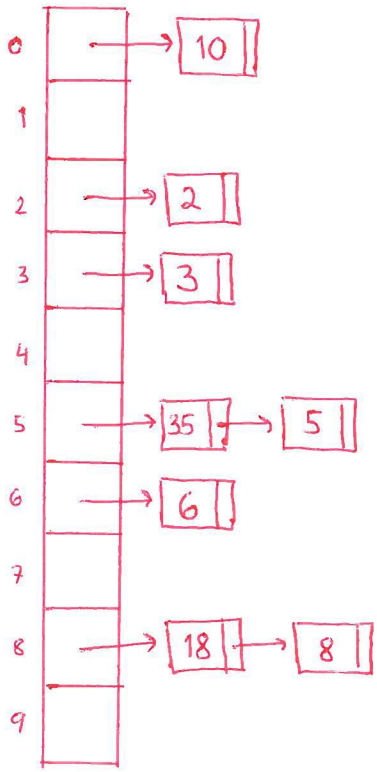
UPPGIFT 6

x :	35	2	18	6	3	10	8	5
x%10 :	5	2	8	6	3	0	8	5

a)

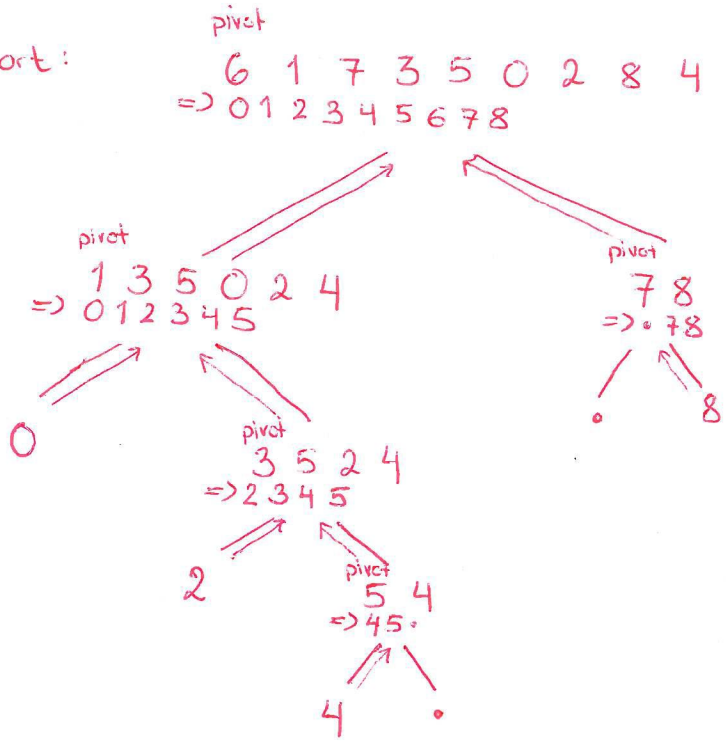


b)



UPPGIFT 7

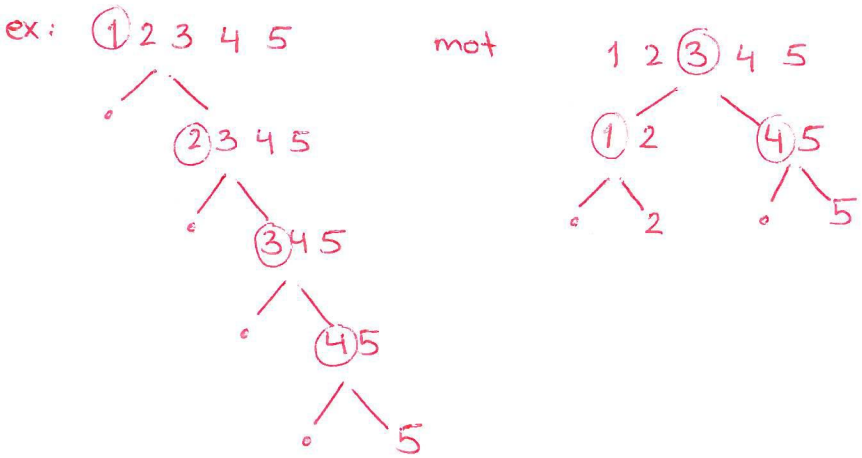
a) Quicksort:



UPPGIFT 7 FORTS.

b) Väljs ett dåligt pivot kan algoritmen degenerera, alltså gå mot $O(n^2)$. Ett bra pivot delar hela tiden mängden på hälften men att beräkna vilket som är ett optimalt pivot, för varje "runda", är kostsamt ($O(n^2)$). I medeltal har algoritmen $O(n \log n)$ men då finns det risk att den degenererar, alltså att värsta fallet $O(n^2)$ uppstår.

Att välja första eller sista elementet som pivot i en sorterad eller bakåtsorterad sekvens ger en degenererad algoritm.



UPPGIFT 8 Pseudokod:

```
MergeSort (integer first, integer last)
        integer mid
        if first < last
                mid = (first+last)/2
                MergeSort (first, mid)
                MergeSort (mid+1, last)
                Merge (first, mid, last)
```