Lösningsförslag till tentamen *

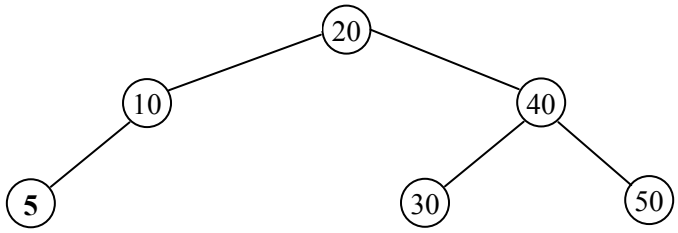
Kursnamn	**Algoritmer och datastrukturer, 5p**
Tentamensdatum	**2005-03-18**
Program	**DAI 2**
Läsår	**2004/2005, lp III**
Examinator	**Uno Holmer**

*** se även `http://www.chl.chalmers.se/~holmer/` där finns det mesta av kursmaterialet.**

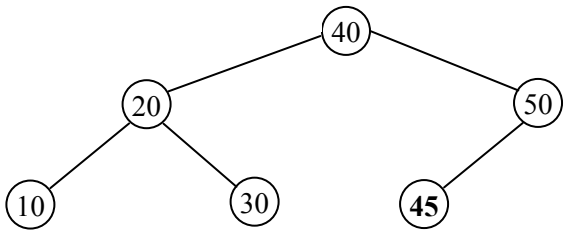
Uppgift 1 (10 p)
Ingen lösning ges. Se kurslitteraturen.

Uppgift 2
a) (5 p)

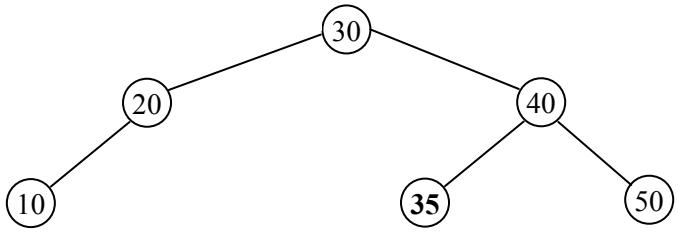
`insert(5);`



`insert(45);`



`insert(35);`



b) (1 p)
Ett AVL-träd av höjd 4 har minst 12 noder och högst $2^{4+1} - 1 = 31$ noder om trädet är fullt.

c) (2 p)
Det minimala antalet noder i ett AVL-träd ges av funktionen

$$f(h) = \begin{cases} 0 & (h = -1) \\ 1 & (h = 0) \\ 1 + f(h-2) + f(h-1) & (h > 0) \end{cases}$$

För $h \geq 0$ gäller att $f(h) = fib(h+2) - 1$, där $fib(n)$ är det n:te Fibonaccitalet.

Uppgift 3

a) (2 p) Använd kvadratisk sondering istället för linjär.

b) (3 p) Med ytterligare ett element i tabellen skulle belastningsfaktorn λ överstiga 0.5, varför omhashning tillämpas. En större tabell av primtalsstorlek allokeras varefter samtliga element, inklusive det nya, sätts in i den nya tabellen med `insert`. I Weiss används fältdubblering så den nya storleken väljs som det minsta primtalet minst lika stort som 2M, i detta fall blir M=17. Tabellen får utseendet

0	
1	
2	
3	
4	
5	
6	
7	58
8	24
9	
10	
11	92
12	
13	
14	
15	
16	**126**

Uppgift 4

a) (5 p)

```
bool matches( const TreeNode *t1, const TreeNode *t2 ) {
    if ( t1 == NULL )
        return true;
    else if ( t2 == NULL )
        return false;
    else
        return t1->element == t2->element &&
               matches( t1->left, t2->left ) &&
               matches( t1->right, t2->right );
}
```

b) (5 p)

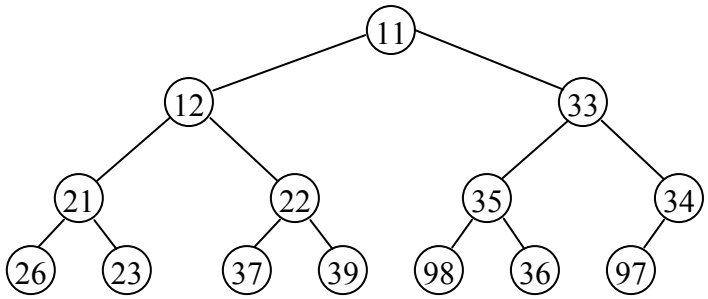
```
bool includes( const TreeNode *t1, const TreeNode *t2 ) {
    return matches( t2, t1 ) ||
           includes( t1->left, t2 ) ||
           includes( t1->right, t2 );
}
```

Uppgift 5

a) (2 p)

Ett träd är en binär hög om ett strukturvillkor och ett ordningsvillkor uppfylls. Strukturvillkoret är att trädet skall vara komplett, vilket gäller då alla nivåer är fyllda, så när som på den sista som, om den inte är helt fylld, fylls på från vänster. Ett sådant träd kan lagras i en vektor. Trädet uppfyller ordningsvillkoret om det för varje nod x som har ett barn y gäller att $x \leq y$.

b) (2 p)

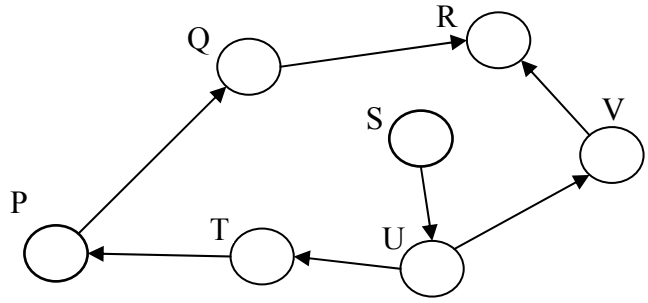


Uppgift 6

a) (2 p)

Grafen måste vara riktad och acyklisk (DAG).

b) (5 p)



Uppgift 7

a) (nivå 5 p)

Konstruera det optimerade linjenätet som ett minimalt uppspännande träd i form av en graf med Kruskals algoritm. Läs först in bågarna från textfilen till en prioritetskö. Bågarna tas ur kön efter växande restid. Antingen används en båge för att koppla ihop två noder, eller så förkastas den. Använd `DisjSets` för att hålla reda på vilka orter i det optimerade linjenätet som förbundits i tidigare steg. Lägg alltid in två motriktade bågar mellan varje nodpar för att göra grafen navigerbar i alla riktningar. Den första tabellen fås sedan enkelt med ett anrop av `printEdges`. Den andra görs enklast med ett antal anrop av `dijkstra` från alla möjliga startpunkter.

b) (nivå 16 p)

```
class TrafikAnalys {
public:
    TrafikAnalys( istream &in ) : inStream(in) {}
    void readFile();
    void create();
    void search();
private:
    Graph g;
    priority_queue< Linjestracka,     // BinaryHeap går lika bra!
                    vector<Linjestracka>,
                    greater<Linjestracka> > pq;
    istream &inStream;
    int noOfCities;
};

void TrafikAnalys::readFile() {
    inStream >> noOfCities;
    cout << "Antal orter: " << noOfCities << endl
         << "Ursprungliga linjesträckor:" << endl;
    inStream.get();
    while ( true ) {
        int fran, till, ts;
        float t;
        inStream >> fran >> till >> ts >> t;
        inStream.get();
        if ( inStream.eof() )
            break;
        Linjestracka ls( fran, till, (Trafikslag)ts, t );
        ls.print();
        pq.push( ls );
    }
    cout << endl;
}
```

forts.

```
void TrafikAnalys::create() {
    // Kruskals algoritm
    DisjSets ds(noOfCities);
    int n = noOfCities;
    do {
        if ( pq.empty() )  // if the graph is very sparse
            break;
        // get edge with smallest cost from pq
        Linjestracka ls = pq.top();
        pq.pop();
        int set1 = ds.find( ls.fran );
        int set2 = ds.find( ls.till );
        if ( set1 != set2 ) {
            ds.unionSets( set1, set2 );
            g.addEdge(ls.fran, ls.till, ls.tid, ls.trafikslag );
            g.addEdge(ls.till, ls.fran, ls.tid, ls.trafikslag );
            n--;
        }
    } while ( n > 0 );
}

void TrafikAnalys::search() {
    cout << "Optimerat linjenät:" << endl;
    g.printEdges();
    cout << "Restider:" << endl;
    for ( int from = 0; from < noOfCities; from++ ) {
        g.dijkstra( from );
        for ( int to =  from + 1; to < noOfCities; to++ )
            cout << "från " << from << " till " << to
                 << ": "<< g.getDistance( to )
                 << " minuter" << endl;
    }
}
```