TENTAMEN: Algoritmer och datastrukturer

Läs detta!

- *Uppgifterna är inte avsiktligt ordnade efter svårighetsgrad.*
- Börja varje uppgift på ett nytt blad.
- Skriv ditt namn och personnummer på varje blad (så att vi inte slarvar bort dem).
- **Skriv rent dina svar. Oläsliga svar r ä t t a s e j!**
- Programmen skall skrivas i C++, vara indenterade och kommenterade, och i övrigt vara utformade enligt de principer som lärts ut i kursen.
- Onödigt komplicerade lösningar ger poängavdrag.
- Programkod som finns i tentamenstesen behöver ej upprepas.
- Givna deklarationer, parameterlistor, etc. får ej ändras, såvida inte annat sägs i uppgiften.
- Läs igenom tentamenstesen och förbered ev. frågor.

I en uppgift som består av flera delar får du använda dig av funktioner klasser etc. från tidigare deluppgifter, även om du inte löst dessa.

Lycka till!

Uppgift 1

Välj **ett** svarsalternativ på varje fråga. Varje korrekt svar ger två poäng.

1. Två av bitföljderna är möjliga Huffman-kompressioner av ett och samma meddelande, vilka?
   a. 0110000111010
   b. 01110100110
   c. 1000101101
   d. 0111101
   e. 0000101111001
   f. 11001100
   g. b och c
   h. a och e
   i. d och f
   j. inget av ovanstående

2. Vilken datastruktur lämpar sig bäst för ickerekursiv utforskning av en labyrint om alla rum på samma avstånd från startpunkten skall besökas före rum som ligger längre bort?
   a. binärt sökträd
   b. mängd
   c. hashtabell
   d. stack
   e. kö
   f. ingen av ovanstående

3. Hur många inre noder måste ett binärt träd med N löv ha minst?
   a. N-1
   b. N
   c. N+1
   d. N/2
   e. 2^N
   f. log N
   g. inget av ovanstående

4. Vilket uttryck ger den snävaste övre begränsningen för antalet exekveringar av S om exekveringen av S antas ta konstant tid.

```
for ( I = 1; I < N; I++ )
    for ( J = 0; J < I*I; J++ )
        if ( J % I == 0 )
            S;
```

   a. $O(N)$
   b. $O(\log N)$
   c. $O(N \log N)$
   d. $O(N^3)$
   e. $O(N^2)$
   f. inget av ovanstående

forts.

5. Följande funktion bryter mot vilken/vilka regler för rekursion?

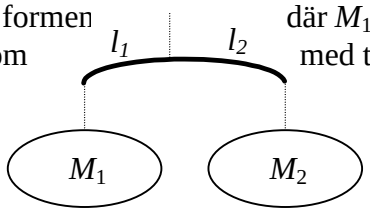
```
int F( unsigned int N ) {
    if ( N == 0 )
        return 0;
    else
        return N + F( N/2 ) + F( N/2+1 );
}
```

a. basfall saknas
b. terminerar ej
c. utför redundant arbete
d. två av ovanstående
e. samtliga (a), (b) och (c)

(10 p)

Uppgift 2

En mobil är antingen

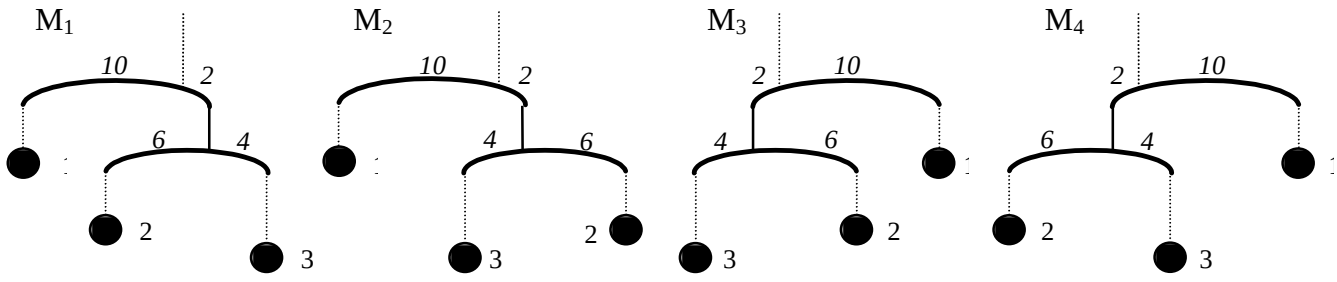
1. ett löv med en massa m:  *m*
2. eller på formen  där M_1 och M_2 är mobiler och l_1 och l_2 är dellängderna på en stång som med två snören förbinder de två delmobilerna med varandra.

Mobiler kan modelleras som dataobjekt, en klassdefinition finns i bilaga.

Konstruera en rekursiv medlemsfunktion i mobilklassen som avgör om två mobiler är ekvivalenta. Två mobiler är ekvivalenta om de kan göras lika genom av vända en eller flera stänger 180 grader.

```
bool Equivalent( const Mobile & Rhs ) const;
```

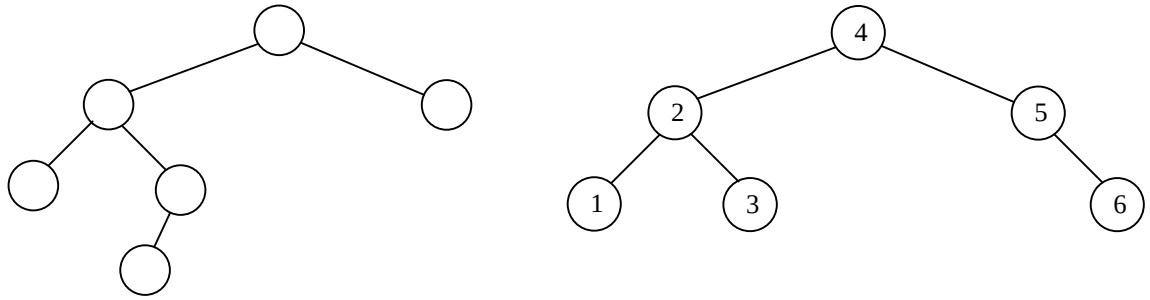
Exempel: följande mobiler är ekvivalenta:



(8 p)

Uppgift 3

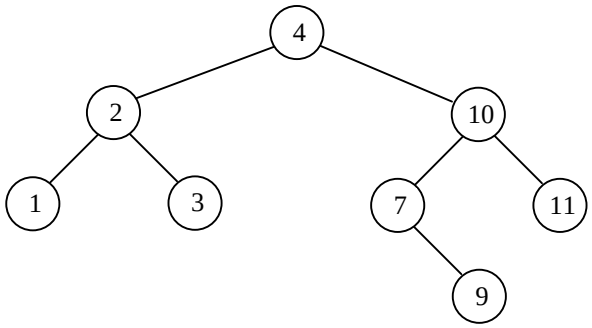
Omedelbart efter insättning av ett visst tal i ett AVL-träd fick trädet strukturen till vänster, och efter balansering den till höger.



a) Vilka noder innehåller vilka tal i det vänstra trädet? Vilket är det sist insatta talet? Visa hur balanseringen gick till. *Ledning:* märk noderna i det vänstra trädet med bokstäver.

(3 p)

b) Hur ser trädet nedan ut efter Remove(4)?



(3 p)

Uppgift 4

En hashtabell har följande innehåll

0	
1	15
2	
3	31
4	17
5	
6	

Hashfunktionen definieras *Hash(x,M) = x mod M,* där *M* är antalet platser i tabellen. Två tal sätts in med operationerna Insert(24); Insert(49). Kvadratisk sondering används.[1] Ange tabellens nya utseende om

a) $\lambda > 0.5$ är tillåtet. (3 p)

b) $\lambda > 0.5$ är otillåtet. (3 p)

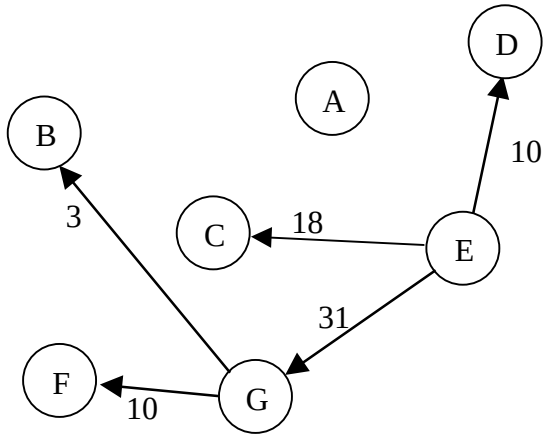
[1] sondering = probing

Uppgift 5

Besvara gärna denna uppgift direkt i tesen.

Betrakta följande riktade graf

	A	B	C	D	E	F	G
A	0	38	∞	∞	5	∞	∞
B	∞	0	4	∞	∞	6	∞
C	∞	∞	0	∞	∞	22	12
D	∞	∞	7	0	∞	∞	∞
E							
F							
G							



a) Komplettera grafen och dess grannmatris ovan.

(3 p)

b) Ange kortaste oviktade vägar, resp. de kortaste viktade vägarna från nod A till samtliga övriga noder.

	B	**C**	**D**	**E**	**F**	**G**
oviktad						
viktad						

(3 p)

c) Är ABEDCGF eller AEGBDCF en topologisk ordning av grafens noder? Motivera svaret!

(3 p)

Uppgift 6

En binär hög *H* representeras av fältet

	8	12	9	25	16	11			
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

a) Rita den binära högen *H* som ett träd.

(1 p)

b) Låt H_2 vara *H* efter sekvensen Toss(7); Toss(2); Toss(4); Rita H_2 som ett träd.

(1 p)

c) Hur ser H_2 ut efter ett anrop av FixHeap()?

(4 p)

Uppgift 7

För att kunna ta emot och ordna datoranvändarnas utskrifter i ett nätverk brukar det finnas någon form av skrivarkö i systemet. Vanligen behandlas utskriftsjobben i samma ordning som de kommer till kön vilket oftast är en acceptabel turordning. Ibland uppkommer dock irriterande situationer, t.ex. när en stor och tidskrävande utskrift ligger före många små i kön. I de flesta fall uppnås ett ”rättvisare” köbeteende om man låter de små jobben passera de stora. För att få en uppfattning om hur ett sådant skrivarsystem kommer att fungera i verkligheten kan man göra en simulering, där ett stort antal utskriftsjobb av slumpmässig storlek och tidpunkt skapas.
Uppgiften går ut på att skriva färdigt ett givet simuleringsprogram (se bilaga). Ett utskriftsjobb modelleras av ett objekt av klassen `PrintJob` innehållande: ett unikt id-nummer, storleken i bytes på filen som skall skrivas ut, samt tidpunkten då jobbobjektet läggs i kön (all tid mäts i sekunder). Turordningsreglerna mellan jobben definieras av operatorn < i klassen `PrintJob`. Jobbet J_1 behandlas före J_2 om $J_1 < J_2$, vilket definieras:

- J_1:s storlek är mindre än J_2:s
- båda är lika stora och J_1 kom till kön före J_2
- båda är lika stora och kom till kön samtidigt och J_1 har ett lägre idnummer än J_2
- annars är $J_1 < J_2$ falskt

a) Implementera operatorn < i klassen `PrintJob`. (5 p)

b) Implementera funktionen `Simulate` i klassen `Simulator`. `Simulate` skall gå igenom så många tidssteg (sekunder) som anges av parametern. Nya slumpmässiga jobb skall läggas i kön, och även tas bort ur kön för att ”skrivas ut”. Reglerna är följande:
Konstruktion och insättning av nytt jobbobjekt i kön:
- Id sätts till lägsta ej tidigare använda heltal (med början på 1).
- Storleken sätts slumpmässigt till ett tal i intervallet [1,100000] bytes.
- Tiden för insättningen i kön sätts till aktuell tid.
- Tiden mellan två på varandra följande insättningar i kön väljs slumpmässigt mellan en sekund och tio minuter.
- Högst ett jobb sätts in i kön vid varje tidpunkt. Dock kan ett annat jobb tas ut vid samma tidpunkt (ordningen oväsentlig).

Uttag av jobbobjekt ur kön:
- Skrivaren kan skriva ut ett jobb i taget med hastigheten 256 bytes (tecken) per sekund. ”Utskriften” i simuleringen består i att skriva ut objektets attribut (Print) (se ex. nedan).
- Under pågående ”utskrift” hämtas inga fler jobb från kön.
- Så snart ett jobb är ”utskrivet” hämtas nästa från kön (om den inte är tom).

Välj lämplig datastruktur så att utskriftsjobben behandlas enligt ordningen som beskrivits ovan. Funktionsanropet `random(x)` returnerar ett slumptal i intervallet [0,x-1]. *Ledning:* Utför simuleringen i en loop som räknar upp sekundtid från 0. Exempel på simulering:

```
Enter simulation time (seconds): 3000
Simulation begins
 1 (58259 bytes) enqueued at 0
 2 (90976 bytes) enqueued at 360
 3 (11772 bytes) enqueued at 738
 4 (12324 bytes) enqueued at 1248
 5 (47592 bytes) enqueued at 1655
 6 (99387 bytes) enqueued at 1748
 8 (80842 bytes) enqueued at 2209
10 (17807 bytes) enqueued at 2523
 9 (43655 bytes) enqueued at 2497
12 (16722 bytes) enqueued at 2655
11 (53249 bytes) enqueued at 2651
```

Varje anrop av Print ger en sådan utskriftsrad

(10 p)

Bilagor till tentamen

Bilaga till uppgift 2

```
class Mobile {
public:
    Mobile( float Mass );                       // Simple case
    Mobile( const Mobile *L, float LLength,     // Composite case
            const Mobile *R, float RLength );
    ~Mobile();
    // structural equivalence
    bool Equivalent( const Mobile & Rhs ) const;
private:
    enum MobileType { Simple, Composite };
    MobileType Type;
    float theMass;                  // Simple
    float LeftLength, RightLength;  // Composite
    const Mobile *Left, *Right;     //    -" -
    int IsSimple() const { return Type == Simple; }
};

// make a leaf mobile
Mobile::Mobile( float M )
    : Type(Simple), theMass(M), Left(NULL), Right(NULL) { }

// make a composite mobile
Mobile::Mobile( const Mobile *L,  float LLength,
                const Mobile *R, float RLength )
    : Type(Composite), Left(L), Right(R),
      LeftLength(LLength), RightLength(RLength) { }
```

Bilaga till uppgift 7

```
class PrintJob {
public:
    PrintJob() : _Size(0), _ArrivalTime(0) {}
    PrintJob( int I, int S, int AT )
        : _Id(I), _Size(S), _ArrivalTime(AT) {}
    unsigned int Id()          const { return _Id; }
    unsigned int ArrivalTime() const { return _ArrivalTime; }
    unsigned int Size()        const { return _Size; }
    void Print() const; // Prints attribute values on screen
    // Din version av ... operator< ...

private:
    int _Id;            // identification number
    int _Size;          // size measured in bytes
    int _ArrivalTime;   // time of arrival to queue
};


class Simulator {
public:
    Simulator();
    void Simulate( unsigned int SimulationTime );
private:
    // ...
};

// Din implementering av Simulate
// ...


void main() {
    randomize();
    int Time;
    cout << "Enter simulation time (seconds): ";
    cin >> Time;
    cout << "Simulation begins" << endl;
    Simulator S;
    S.Simulate( Time );
}
```

```
// Set class interface
//
// CONSTRUCTION: with (a) size of maximal integer to be stored
//      in the set, (b) initialization with existing set
// ************************  public operations ************************
// Set operator+= (int)     --> Add an integer to the set
// Set operator= (Set)      --> Copying assignment
// bool operator< (int,Set) --> Membership relation
// bool operator<= (Set)    --> Subset relation
// bool operator== (Set)    --> Equality relation
// Set operator|| (Set)     --> Set union
// Set operator&& (Set)     --> Set intersection
// Set operator- (Set)      --> Set difference
// int size()               --> Number of distinct elements in the set
// istream operator>> (istream,Set) --> Reads a set from the keyboard
// ostream operator<< (ostream,Set) --> Prints a set on the screen

class Set {
public:
    Set(int maxNum);                              // constructor
    Set(const Set & s);                           // copy constructor
    ~Set();                                       // destructor
    Set & operator += (int x);                    // add an integer
    const Set & operator = (const Set & Rhs);     // copying assignment
    friend bool operator < (int x, const Set & Rhs); // membership
    bool operator <= (const Set & Rhs) const;     // subset
    bool operator == (const Set & Rhs) const;     // equality
    Set operator || (const Set & Rhs) const;      // union
    Set operator && (const Set & Rhs) const;      // intersection
    Set operator - (const Set & Rhs) const;       // difference
    int Size() const { return theSize; }          // number of elements
    friend istream & operator >> (istream & in,Set & value);
    friend ostream & operator << (ostream & out,const Set & value);
private:
    int *theSet;  // array pointer
    int max;      // max size of stored numbers (= array size - 1)
    int theSize;  // number of distinct elements in the set
};
```

```
// Queue class interface
//
// Etype: must have zero-parameter and constructor
// CONSTRUCTION: with (a) no initializer;
//     copy construction of Queue objects is DISALLOWED
// Deep copy is supported
//
// ******************PUBLIC OPERATIONS*********************
// void Enqueue( Etype X )--> Insert X
// void Dequeue( )        --> Remove least recently inserted item
// Etype Front( )         --> Return least recently inserted item
// int IsEmpty( )         --> Return 1 if empty; else return 0
// int IsFull( )          --> Return 1 if full; else return 0
// void MakeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Predefined exception is propogated if new fails
// EXCEPTION is called for Front or Dequeue on empty queue

template <class Etype>
class Queue
{
  public:
    Queue( );
    ~Queue( );

    const Queue & operator=( const Queue & Rhs );

    void Enqueue( const Etype & X );    // Insert
    void Dequeue( );                    // Remove
    const Etype & Front( ) const;       // Find
    int IsEmpty( ) const;
    int IsFull( ) const;
    void MakeEmpty( );
  private:
    Queue( const Queue & ) { }    // Disable copy constructor
    // Data representation ...
 };
```

```
// BinaryHeap class interface
//
// Etype: must have zero-parameter constructor and operator=;
//     must have operator<
// CONSTRUCTION: with (a) Etype representing negative infinity
// Copy construction of BinaryHeap objects is DISALLOWED
// Deep copy is supported
//
// ******************PUBLIC OPERATIONS******************
// void Insert( Etype X ) --> Insert X
// Etype FindMin( )       --> Return smallest item
// void DeleteMin( )      --> Remove smallest item
// void DeleteMin( Etype & X ) --> Same, but put it in X
// int IsEmpty( )         --> Return 1 if empty; else return 0
// int IsFull( )          --> Return 1 if full; else return 0
// void MakeEmpty( )      --> Remove all items
// void Toss( Etype X )   --> Insert X (lazily)
// void FixHeap( )        --> Reestablish heap order property
// ******************ERRORS*****************************
// Predefined exception is propogated if new fails
// EXCEPTION is thrown for FindMin or DeleteMin when empty

template <class Etype>
class BinaryHeap
{
  public:
        // Constructor, destructor, and copy assignment
    BinaryHeap( const Etype & MinVal );
    ~BinaryHeap( ) { delete [ ] Array; }

    const BinaryHeap & operator=( const BinaryHeap & Rhs );

        // Add an item maintaining heap order
    void Insert( const Etype & X );

        // Add an item but do not maintain order
    void Toss( const Etype & X );

        // Return minimum item in heap
    const Etype & FindMin( );

        // Delete minimum item in heap
    void DeleteMin( );
    void DeleteMin( Etype & X );

        // Reestablish heap order
    void FixHeap( );

    int IsEmpty( ) const;
    int IsFull( ) const;
    void MakeEmpty( );
private:
    // Data representation
};
```