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. Motivering krävs ej. Varje korrekt svar ger två poäng. Garderingar ger noll poäng.

1. Vad karakteriserar kodningsträd i Huffmans algoritm?
   a. trädet är ett AVL-träd
   b. delträdens inbördes ordning är oväsentlig
   c. minst en nod har ett barn
   d. inget av ovanstående

2. En sorteringsalgoritm som gör successiva jämförelser av två element i taget måste vara
   a. $O(N^2)$
   b. $O(N \log N)$
   c. $\Omega(N^2)$
   d. $\Theta(N \log N)$
   e. inget av ovanstående

3. Hur många gånger exekveras rad 4 i följande kodavsnitt?

```
1   for ( int i = 0; i < N; i++ )
2       for ( int j = i; j <= N; j++ )
3           for ( int k = i; k <= j; k++ )
4               Sum++;
```

   a. $O(N)$
   b. $O(N^2)$
   c. $O(N^3)$
   d. $O(N^4)$
   e. inget av ovanstående

4. Vilket är det mest troliga resultatet när en rekursiv beräkning inte närmar sig basfallet?
   a. kompilatorn går in i en oändlig loop
   b. man får ett kompileringsfel
   c. minnet tar slut
   d. den rekursiva funktionen går in i en oändlig loop
   e. den rekursiva funktionen terminerar med ett felaktigt värde
   f. inget av ovanstående

5. I ett prefixuttryck sätts operatorn före operanderna. Vilken/vilka är prefixform för postfixuttrycket ’a b + c * d e – /’ ?
   a. / – e d * c + b a
   b. / * + a b c – d e
   c. / – d e * + a b c
   d. / – d e * c + a b
   e. två är möjliga
   f. inget av ovanstående

(10 p)

Uppgift 2

Enkellänkade listor kan byggas upp av noder av följande typ:

```
template <class T>
struct Node {
    Node( const T & d, Node<T> *n ) : data(d), next(n) {}
    T data;
    Node *next;
};
```

Definiera *rekursivt* den generiska funktionen `zip` som tar pekare till två listor som argument och returnerar en pekare till en lista av par som är en elementvis sammanflätning av argumentlistorna. Om listorna har olika längd kastas undantaget `std::length_error`.

```
    template <class T,class U>
    Node< Pair<T,U> > *zip( Node<T> *l1, Node<U> *l2 )
                       throw (std::length_error);
```

Par representeras med klassen:

```
template <class T,class U>
class Pair {
public:
    Pair( const T & f, const U & s ) : fst(f), snd(s) {}
    T first() const { return fst; }
    U second() const { return snd; }
private:
    T fst;
    U snd;
};
```

Exempel: Om l1 innehåller 1 2 3 och l2 'a' 'b' 'c' så skall l3 peka på en lista som innehåller paren (1,'a'), (2,'b') och (3,'c').

```
Node<int> *l1 = new Node<int>(1,new Node<int>(2,
                                    new Node<int>(3,NULL)));
Node<char> *l2 = new Node<char>('a',new Node<char>('b',
                                    new Node<char>('c',NULL)));
Node< Pair<int,char> > *l3 = zip(l1,l2);
```

(10 p)

Uppgift 3

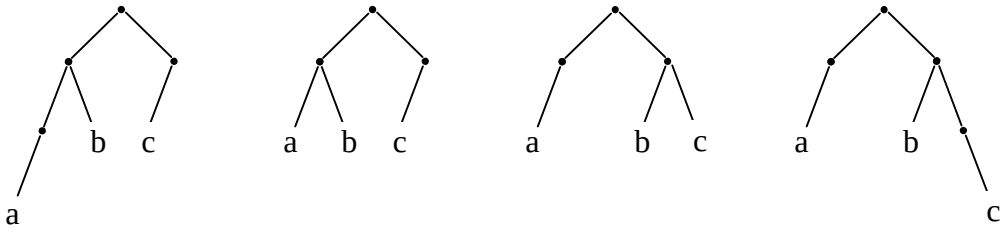
Binära träd kan byggas upp av noder av följande typ:

```
template <class T>
struct TreeNode {
   // Make a leaf
   TreeNode( const T & x ) : data(x), left(NULL), right(NULL) {}
   // Make an internal node
   TreeNode( TreeNode *l, TreeNode *r ) : left(l), right(r) {}
   T data;
   TreeNode *left, *right;
   bool isLeaf() const { return left == NULL && right == NULL; }
};
```

Ett löv representeras alltså som en nod med två tomma delträd. Om man utnyttjar lövnoderna för informationslagring men inte de inre noderna kan man t.ex. använda typen för att representera kodningsträd.

Def. Randen av ett träd (*eng.* fringe) är en uppräkning av trädets löv från vänster till höger.

Exempel: nedanstående träd har alla randen a, b, c. [1] Randen "glömmer" alltså trädets inre struktur.



a) Implementera funktionen

```
template <class T>
List<T> fringe( const TreeNode<T> *aTree );
```

som returnerar randen av ett träd som en lista av innehållet i löven. För träden ovan skall alltså `fringe` returnera en lista innehållande a, b, c. Funktionen skall vara rekursiv.

(8 p)

b) Implementera funktionen `sameFringe` som avgör om två träd har samma rand:

```
template <class T>
bool sameFringe( const TreeNode<T> *t1, const TreeNode<T> *t2 );
```

(2 p)

För att underlätta lösningen är listklassen i bilagan något utvidgad jämfört med kursbokens version. Dessutom finns till din hjälp den färdiga funktionen `append` för att sätta samman två listor, den skapar en ny lista som är sammansättningen av innehållet i de två argumentlistorna `l1` och `l2`:

```
template <class T>
List<T> append( const List<T> & l1, const List<T> & l2 );
```

[1] Träden i exemplet är ej avsedda att vara kodningsträd.

Uppgift 4

En hashtabell innehåller följande heltal

0	39
1	
2	41
3	
4	
5	
6	
7	
8	
9	
10	
11	
12	77

Hashfunktionen definieras *Hash(x,M) = x mod M*, där *M* är antalet platser i tabellen..

b) Ange tabellens utseende vid linjär resp. kvadratisk sondering efter operationerna: [2]

```
Insert(60); Insert(64); Remove(39); Insert(12);
```

(3 p)

c) Antag att man vill sätta in fyra element $a_1,\ldots, a_4$ i en tom tabell med åtta platser och att hashvärdet råkar bli samma för alla fyra. Kvadratisk sondering används och tabellstorleken är fixerad till åtta platser. Förklara vad som händer!

(3 p)

[2] sondering = probing

Uppgift 5

En graf representeras av grannmatrisen

	A	B	C	D	E	F	G	H
A	0	4	∞	∞	∞	∞	∞	∞
B	∞	0	∞	∞	∞	∞	∞	∞
C	3	1	0	∞	12	6	3	∞
D	2	∞	4	0	15	∞	8	∞
E	∞	∞	∞	∞	0	∞	∞	∞
F	∞	∞	∞	∞	6	0	∞	3
G	∞	∞	∞	∞	∞	1	0	2
H	∞	∞	∞	∞	∞	∞	∞	0

a) Rita grafen efter samma mönster som i kursboken.

(2 p)

b) Ange noderna i de kortaste oviktade vägarna, resp. de kortaste viktade vägarna från nod D till samtliga övriga noder.

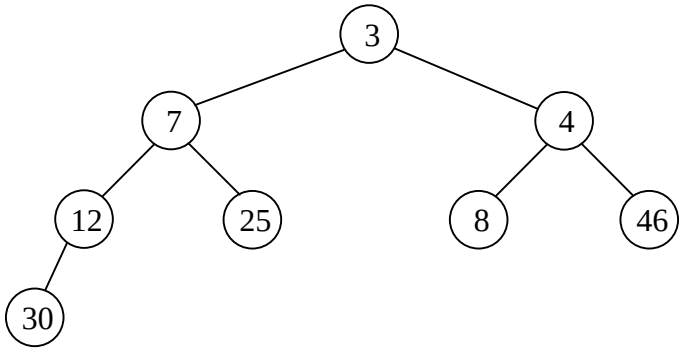
(2 p)

c) Går det att ge en topologisk ordning av noderna? Ange i så fall en sådan, om inte förklara varför det inte går.

(2 p)

Uppgift 6

Om *H* är den binära högen



a) Hur ser *H* ut efter tre anrop av DeleteMin?

(3 p)

b) Hur ser *H* ut efter insättning av talen 15, 20, 6, 2 med operationen Insert?

(3 p)

Uppgift 7

De flesta banker har ett maxbelopp samt en gräns för hur många uttag som får göras under en 24-timmarsperiod. Antag t.ex. att en viss bank tillämpar följande villkor:

- Maximalt 10 uttag om totalt högst 5000 kr får göras under en sammanhängande 24-timmarsperiod, oavsett ev. insättningar under perioden.
- Uttag under 300 kr belastar kontot med en småuttagsavgift om 10 kr.
- Uppkommer negativt saldo på kontot belastas kontot med 100 kr i övertrasseringsavgift.
- Uttag kan ej ske från konto med negativt saldo.

Klassen `Konto` modellerar ett bankkonto:

```
class Konto {
public:
    Konto( int startSaldo,  // kontots startsaldo
           int period,      // löpande period i timmar
           int maxUttag,    // max antal uttag per period
           int maxBelopp ); // max uttag under löpande period
    int saldo() const { return _saldo; }
    int transaktion( int belopp );
private:
    int _saldo;
};
```

Komplettera klassen Konto. Inför ytterligare nödvändiga dataattribut samt implementera operationerna. Ev. hjälpfunktioner får införas som privata medlemsfunktioner. I funktionen `transaktion` tolkas positivt belopp som insättning och negativt som uttag. Funktionen skall returnera insättnings/uttagsbeloppet vid lyckad transaktion, och 0 om uttag ej medges. Använd lämplig datastruktur för att hålla reda på uttagen. Funktionen `time` returnerar antalet sekunder som förflutit sedan 1971-01-01. Till din hjälp finns följande typ för att tidsstämpla uttag:

```
struct Uttag {
    int belopp;         // uttaget belopp
    time_t tidpunkt;    // tidpunkten för uttaget
    Uttag() : belopp(0), tidpunkt(0) {}
    Uttag( int b ) : belopp(b), tidpunkt(time(0)) {}
};
```

(12 p)

BILAGOR TILL TENTAMEN

```
// 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:
    // Data representation ...
};
```

```
// List class interface
//
// Etype: must have zero-parameter and copy constructor
//     operator!= must be provided
// CONSTRUCTION: with (a) no initializer;
//     copy construction of List objects is DISALLOWED
// Deep copy is supported
// Access is via ListItr class
//
// ******************PUBLIC OPERATIONS*********************
// int IsEmpty( )         --> Return 1 if empty; else return 0
// int IsFull( )          --> Return 1 if full; else return 0
// void MakeEmpty( )      --> Remove all items

template <class Etype>   // Incomplete class declaration
class ListItr;           // so friend is visible

template <class Etype>
class List {
  public:
    List( );
    List( const List & rhs );   // copy constructor
    ~List( );

    const List & operator=( const List & Rhs );
    bool operator==( const List &rhs ) const;    // equality

    int IsEmpty( ) const;
    int IsFull( )  const;
    void MakeEmpty( );
  private:
    List( const List & ) { } // Disable copy constructor
    // Data representation ...
};


// ListItr class interface;
//
// Etype: same restrictions as for List
// CONSTRUCTION: with (a) List to which ListItr is permanently
//     bound or (b) another ListItr;
//     Copying of ListItr objects not supported in current form
//
// ******************PUBLIC OPERATIONS*********************
// void Insert( Etype X ) --> Insert X after current position
// int Remove( Etype X )  --> Remove X
// int RemoveNext()       --> Remove next cell
// int Find( Etype X )    --> Set current position to view X
// int IsFound( Etype X ) --> Return 1 if X would be found
// void Zeroth( )         --> Set current position to prior to first
// void First( )          --> Set current position to first
// void operator++        --> Advance (both prefix and postfix)
// int operator+( )       --> True if at valid position in list
// Etype operator( )      --> Return item in current position
// ******************ERRORS********************************
// Predefined exception is propogated if new fails
// EXCEPTION is thrown for illegal access, advance,
//     insertion, or First on empty list
```

```
template <class Etype>
class ListItr
{
  public:
    ListItr( const List<Etype> & L );
    ~ListItr( ) { }

    // Insert X after Current position
    // Errors: Current position is NULL or memory is exhausted
    // Current: Set to new node on success; unchanged otherwise
    void Insert( const Etype & X );

    // Set Current position to first node containing X
    // Returns: 1 if Find is successful, 0 for failure
    // Current: is unchanged if X is found
    int Find( const Etype & X );

    // Returns 1 if X is in the list, 0 otherwise
    int IsFound( const Etype & X ) const;

    // Removes first occurrence of X; do nothing if X is not found
    // Returns: 1 if remove succeeded, 0 otherwise
    // Current: is moved to the header if X is deleted
    int Remove( const Etype & X );

    // Remove next cell
    // Returns: 1 if remove succeeded, 0 otherwise
    // Current: is not moved
    int RemoveNext( );

    // Returns 1 if Current is not NULL or Header, 0 otherwise
    int operator+( ) const;

    // Returns the Element in Current node
    // Errors: If Current is pointing at a node in the list
    const Etype & operator() ( ) const;

    // Set Current to the header node
    void Zeroth( );

    // Set Current to first node in the list
    // Errors: If List is empty
    void First( );

    // Set Current to Current->Next; no return value
    // Errors: If Current is NULL on entry
    void operator++( );
    void operator++( int );
  private:
     // Data representation
};
```

```
// 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
};
```

```
// SearchTree class interface
// Etype: must have zero-parameter and copy constructor,
//     and must have operator<
// CONSTRUCTION: with (a) no initializer;
// All copying of SearchTree objects is DISALLOWED
// ******************PUBLIC OPERATIONS*********************
// int Insert( Etype X )  --> Insert X
// int Remove( Etype X )  --> Remove X
// Etype Find( Etype X )  --> Return item that matches X
// int WasFound( )        --> Return 1 if last Find succeeded
// int IsFound( Etype X ) --> Return 1 if X would be found
// Etype FindMin( )       --> Return smallest item
// Etype FindMax( )       --> Return largest item
// int IsEmpty( )         --> Return 1 if empty; else return 0
// void MakeEmpty( )      --> Remove all items
// ******************ERRORS********************************
// Predefined exception is propogated if new fails
// ItemNotFound returned on various degenerate conditions

template <class Etype>
class SearchTree {
  public:
    SearchTree();
    ~SearchTree();
    // Add X into the tree. If X already present, do nothing.
    // Return true if successful
    bool Insert( const Etype & X );

    // Remove X from the tree. Return true if successful.
    bool Remove( const Etype & X );

    // Remove minimum item from the tree. Return true if successful.
    bool RemoveMin( );

    // Return minimum item in tree. If tree is empty,
    // return ItemNotFound.
    const Etype & FindMin( ) const;

    // Return maximum item in tree. If tree is empty,
    // return ItemNotFund.
    const Etype & FindMax( ) const;

    // Return item X in tree. If X is not found, return ItemNotFound.
    // Result can be checked by calling WasFound.
    const Etype & Find( const Etype & X );

    // Return true if X is in tree.
    bool IsFound( const Etype & X );

    // Return true if last call to Find was successful.
    bool WasFound( ) const;

    // MakeEmpty tree, and test if tree is empty.
    void MakeEmpty( )
    bool IsEmpty( ) const;
  protected: // private data representation
};
```

```
// PreOrder class interface; maintains "current position"
//          in Preorder Tree Traversal
//
// Etype: same restrictions as for BinaryTree
// CONSTRUCTION: with (a) Tree to which iterator is bound
// All copying of PreOrder objects is DISALLOWED
//
// ******************PUBLIC OPERATIONS*********************
// int operator+( )     --> True if at valid position in tree
// Etype operator( )    --> Return item in current position
// void First( )        --> Set current position to first
// void operator++      --> Advance (prefix)
// ******************ERRORS********************************
// EXCEPTION is thrown for illegal access or advance


// PostOrder class interface; maintains "current position"
//           in Postorder Tree Traversal
//
// Etype: same restrictions as for BinaryTree
// CONSTRUCTION: with (a) Tree to which iterator is bound
// All copying of PostOrder objects is DISALLOWED
//
// ******************PUBLIC OPERATIONS*********************
// int operator+( )     --> True if at valid position in tree
// Etype operator( )    --> Return item in current position
// void First( )        --> Set current position to first
// void operator++      --> Advance (prefix)
// ******************ERRORS********************************
// EXCEPTION is thrown for illegal access or advance


// InOrder class interface; maintains "current position"
//          in Inorder Tree Traversal
//
// Etype: same restrictions as for BinaryTree
// CONSTRUCTION: with (a) Tree to which iterator is bound
// All copying of InOrder objects is DISALLOWED
//
// ******************PUBLIC OPERATIONS*********************
// int operator+( )     --> True if at valid position in tree
// Etype operator( )    --> Return item in current position
// void First( )        --> Set current position to first
// void operator++      --> Advance (prefix)
// ******************ERRORS********************************
// EXCEPTION is thrown for illegal access or advance


Exempel:
    SearchTree<int> T;
    PreOrder<int> It(T);
```