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. Antag att $T_1(N) = O(F(N))$ och $T_2(N) = O(F(N))$.Vilket av följande är sant?
   a. $T_1(N)+T_2(N) = O(F(N))$
   b. $T_1(N)\times T_2(N) = O(F(N))$
   c. $T_1(N)/T_2(N) = O(1)$
   d. $T_2(N)/T_1(N) = \Omega(F(N))$
   e. fler än ett av ovanstående
   f. inget av ovanstående är korrekt

2. För vilken av följande algoritmer utgör en redan sorterad följd värsta fallet?
   a. mergesort
   b. quicksort
   c. shellsort
   d. insertion sort
   e. en sorterad följd kan aldrig vara värsta fallet för en sorteringsalgoritm

3. Vilken/vilka av bitföljderna är möjliga Huffman-kompressioner av strängen ”mallaga”?
   a. 0010100110110101
   b. 11010010011
   c. 1010111101000
   d. 01011001000
   e. 0001010110011
   f. 0101011111011010

4. Om stackar implementeras med fält, vad är kostnaden i värsta fall för N push-operationer i följd om fältdubblering används?
   a. $O(1)$
   b. $O(N)$
   c. $O(N \log N)$
   d. $O(N^2)$
   e. inget av ovanstående

5. Vilken av följande datastrukturer lämpar sig bäst för att implementera en mängd med operationerna += (addera ett element) och < (tillhör)?
   a. binärt sökträd
   b. lista
   c. hashtabell
   d. prioritetskö
   e. alla passar lika bra
   f. ingen är lämplig

(10 p)

Uppgift 2

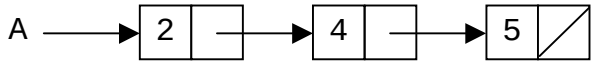
En enkellänkad lista kan representeras med följande nodtyp:

```
struct Node {
    int Data;
    Node *Next;
    Node( int D, Node *N ) : Data(D), Next(N) {}
};
```

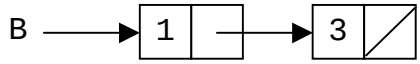
Definiera *rekursivt* funktionen `Merge` som returnerar en ordnad sammanflätning av två ordnade listor

```
Node *Merge( const Node *L1, const Node *L2 );
```

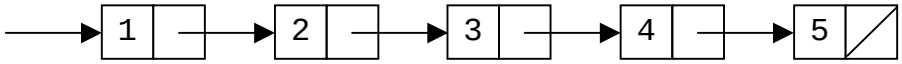
Exempel: Om A pekar ut listan



och B listan



så skall `Merge(A,B)` returnera den nya unika listan

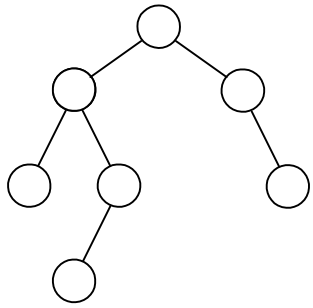


Lämplig hjälpfunktion får införas.

(10 p)

Uppgift 3

a) Noderna i det binära trädet till höger är numrerade 1-7.
Vid inorder genomlöpning av trädet besöktes noderna i ordningen ? 1 ? 6 7 ? 5 och vid preorder genomlöpning av samma träd i ordningen ? ? 2 ? 4 3 ?. Fyll i rätt tal i rätt nod!



(2 p)

Antag att binära träd med heltalsnoder representeras med typen

```
struct Tree {
    int Data;
    Tree *Left, *Right;
};
```

Ett tomt träd representeras med en NULL-pekare. Följande två färdiga funktioner får användas i uppgifterna:

```
int max( int a, int b ) { return a > b ? a : b; }
int min( int a, int b ) { return a < b ? a : b; }
```

b) För binära träd kan man definiera funktionen Max så att Max(T) är -∞ om T är tomt och det största elementet i T annars. Implementera Max som en rekursiv funktion:

```
int Max( const Tree *T );
```

Antag att -∞ kan approximeras med konstanten `INT_MIN`.

(4 p)

c) Implementera en rekursiv funktion `IsOrdered` som avgör om ett binärt träd är ordnat, d.v.s. om det är ett sökträd:

```
bool IsOrdered( const Tree *T );
```

Funktionen skall returnera `true` om `T` pekar på ett ordnat träd och `false` annars. Du får anta att funktionen Min på träd är definierad på analogt sätt som i b: Min(T) är ∞ om T är tomt och det minsta elementet i T annars. ∞ approximeras med `INT_MAX`.

(6 p)

Uppgift 4

I båda uppgifterna definieras hashfunktionen *Hash(x,M) = x mod M,* där *M* är tabellstorleken.

a) Antag att en sluten hashtabell med 9 positioner är tom. Ange tabellens utseende vid linjär sondering efter operationerna: [1]

```
Insert(67); Insert(20); Insert(12); Insert(101); Remove(67); Insert(3);
```

(3 p)

b) Antag att en sluten hashtabell ser ut så här

0	13
1	
2	24
3	47
4	
5	
6	72
7	2
8	
9	
10	

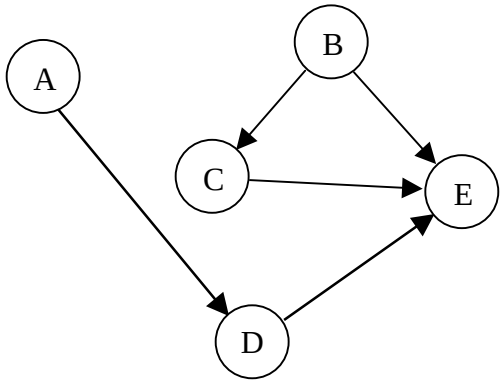
efter att talen satts in i följd med kvadratisk sondering i en tom tabell. Inga andra operationer än insättning har använts. Ange en möjlig insättningsordning som ger denna placering av talen!

(4 p)

[1] sondering = probing

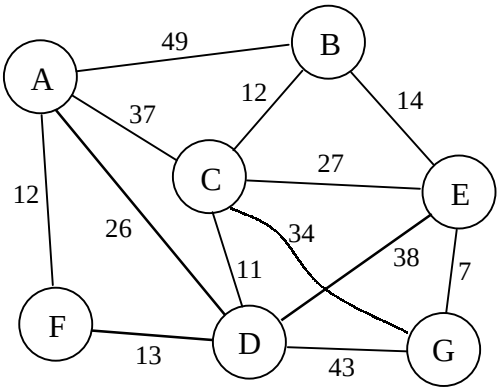
Uppgift 5

a) Ange alla möjliga topologiska ordningar av noderna i grafen



(3 p)

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



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

(3 p)

Uppgift 6

En tom binär hög fylls på med 10 heltal:

```
BinaryHeap<int> H(0);
H.Toss(3); H.Toss(5); H.Toss(6); H.Toss(10); H.Toss(7);
H.Toss(8); H.Toss(4); H.Toss(2); H.Toss(9); H.Toss(1);
H.FixHeap();
```

Rita `H` som ett träd

a) efter sista `Toss` men före `FixHeap`

(1 p)

b) efter `Fixheap`

(4 p)

Uppgift 7

Skriv ett program som läser ett antal textrader från tangentbordet och skriver ut vilka tecken som förekommer på samtliga rader. I utskriften får varje tecken bara förekomma en gång. Till exempel skall indata

```
Några tecken
råkar finnas
på varje rad
```

ge utskriften `arå`, och indata

```
medan
andra
texter
saknar
denna
egenskap
```

inte ge någon utskrift alls. Indata avslutas med CTRL-Z (end of file). För poäng krävs att du använder lämplig datastruktur (se bilagorna). Krånglig lösning som baseras på olämplig datastruktur ger 0p. Dela in i lämpliga underprogram.

(10 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 ...
};
```

```
// 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);
```