# Universitetet i Bergen
## Det matematisk-naturvitenskapelige fakultet

English

| | | |
|---|---|---|
| Examination in | : | I-120 Algorithms, Data Structures and Programming |
| Date | : | 15 December 1999 |
| Time | : | 9:00 – 15:00 |
| No. pages | : | 4 |
| Auxialiary materials | : | None |

- You may use all `interface`s and data structures from the lectures and exercises – whenever possible use `interface`s and avoid unnecessary use of concrete data types.
- "Program"/"implement" means 'write a full Java code'. (If you can't do that, write at least precise pseudo-code.) If your program needs an *implementation* of some `interface`, you may assume that such an implementation is available without programming it yourself. Indicate places where you use this assumption.
- "Sketch"/"explain" means preferably 'write precise pseudo-code with necessary comments".
- Explain your code (if you have time, describe preconditions and results of methods, specify invariants for central loops and recursions, etc.).
- Percentages at each (sub)problem indicate approximate and expected time needed for solution as well as approximate weight at marking.

# 1 Implementation (20%)

You shall implement a (simplified) editor line ADT. It keeps `Character`-objects with a cursor placed always *between* two characters in the line (possibly before the first/after the last one).

```
public interface EditorLine {
/** moves the cursor one position to the left;
    if the cursor was before the first character, nothing happens */
  void left();
/** moves the cursor one position to the right;
    if the cursor was after the last character, nothing happens */
  void right();
/** inserts a new character to the left of the cursor;
    the cursor remains to the right for the inserted character */
  void insert(Character c);
/** removes the character to the left for the cursor;
    if the cursor was before the first character, nothing happens */
  void remove();
/** move the cursor to the left for the first character */
  void start();
/** moves the cursor to the right for the last character */
  void end();               }
```

Program `public class ELwithStack implements EditorLine {...}` so that your implementation satisfies the following conditions:

1. the data structure of `class ELwithStack` contains only variables of type `Stack` (see appendix)
2. the first four methods have complexity $\mathcal{O}(1)$ (relatively to possible implementation of `Stack`).

[Hint: Use two stacks to represent the contents of the line to the left, resp. to the right of the cursor.]

## 2 Binary Search Trees (35%)

You shall extend an implementation of ordered dictionary with a new method for finding the $k$-th element in the dictionary. We start with a standard implementation with binary serach tree:

```
public class BST implements OrderedDictionary {
  protected BinaryTree T;  protected Comparator cp;
... }
```

(See appendix for the declaration of `interface OrderedDictionary` – we are using the `Key-Data` version and not the `Locator` based version of this interface.)

We suppose that we have such an implementation where all the operations (in particular, `find, insert, remove`) have complexity $\mathcal{O}(height(T))$ where $height(T)$ is the height of the actual tree `T`. That the implementation is "standard" means, in particular, that the objects stored at each tree-position keep merely the key-data pairs, i.e., are of the type:

```
public class Item {
   private Object k,e;
   Item(Object key, Object elem) { k= key; e= elem; }
   Object key() { return k; }
   Object element() { return e; }
}
```

You shall use `BST` to make an implementation of

```
public interface OrderedDictionaryK extends OrderedDictionary {
   Object findKth(int k);
}
```

The method `findKth(k)` finds the k-th element in the dictionary – we assume that $1 \leq k \leq n$, where $n$ is the number of elements in the dictionary, and that all the elements have distinct keys.

### 2.1 A direct extension of `BST` (15%)

Formulate first the data invariant for the data structure Binary Search Tree. Then, program

```
public class BSTk1 extends BST implements OrderedDictionaryK {
   Object findKth(int k) { ... }        }
```

The implementation of the method `findKth(k)` shall have complexity not greater than $\mathcal{O}(n)$. It is *not allowed* to extend the data structure (neither for `Item` not for `BST`) except for *one* single global variable of type `int` in `BSTk1`. (New methods may, of course, use local variables.)

### 2.2 An improvement (20%)

You have now again the `class BST` and shall extend it to implement `findKth(k)` with complexity $\mathcal{O}(height(T))$. To achieve this, you are now allowed to extend the data structure for `class BST` and/or `class Item`, but you should make sure that other methods do not get higher complexity.

1. Explain the principle for the new implementation, declare the necessary extensions to the data structures for `Item`/`BST` and *specify the data invariant* for the new implementation:

   ```
   public class BSTk2 extends BST implements OrderedDictionaryK {
      Object findKth(int k) { ... }        }
   ```

2. Program `findKth(k)`
3. Program the method `void insert(Object key, Object elem)` which, probably, has to be overwritten. If not – just state that.

[Hint: It is enough with one additional `int` attribute in `class Item`.]

# 3 Graphs (45%)

This problem addresses only *directed* graphs. We shall consider variations of the following algorithm:

```
LiFi TS(Graph G) {
  Q,R = new empty LiFi's
  for each node v in G {
    v.in = v's indegree in G //number of incoming edges to v
    if (v.in==0) add v to Q
  }
  while (Q is not empty) {
    h = fetch a node from Q
    for each outgoing edge (h,v) {
      v.in--
      if (v.in==0) add v to Q
    }
    insert h into R
  }
  return R
}
```

`Interface LiFi`, and its two subtypes `Queue` and `Stack`, are specified in the appendix. We hav also an implementation of each subtype:

- `class QueueIM implements Queue` – the method `front()` (and `remove()`) return the object which has been longest time in the queue (was inserted earliest)
- `class StackIM implements Stack` – the method `front()` (and `remove()`) return the object which has been shortest time in the stack (was inserted latest)

## 3.1 (10%)

1. Program this algorithm as a method with the following profile:

   ```
   LiFi TS(Graph G, LiFi Q, LiFi R)
   ```

   where the parameter `Q` (resp. `R`) is used as `Q` (resp. `R`) from the sketch.
2. Describe the effect of the call `TS(G,new QueueIm(),new QueueIM())`.
3. Describe the effect of the call `TS(G,new StackIm(),new QueueIM())`. Justify your answer.
4. Describe the effect of the call `TS(G,new StackIm(),new StackIM())`. Justify your answer.

"Describe the effect" means: give the postcondition for the result, possibly depending on the properties of the input graph `G`. Write also the results for the following graph (where adjacent vertices are traversed in alphabetical order):

$$\begin{tikzcd} A \arrow[r] \arrow[d] & B \arrow[r] \arrow[d] & D \arrow[d] \\ C \arrow[r] & E \arrow[r] & F \end{tikzcd}$$

## 3.2 A modification (35%)

You shall modify the `TS`-algorithm so that it finds a special, so called *maximal APC* subgraph of `G`, or else concludes that such a subgraph does not exist.

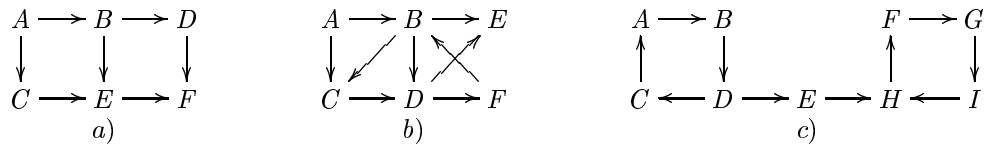
- An *all-paths-cyclic graph* $G'$ (APC graph) is a graph where each vertex $v \in G'$ has at least one incoming *and* at least one outgoing edge, i.e., for each vertex $v \in G'$ both $G'.inDegree(v) \neq 0$ and $G'.outDegree(v) \neq 0$.

- A *maximal* APC subgraph of a graph $G$ is a graph $G'$ such that
  - $G'$ is a subgraph of $G$ (not necessarily a *strict* subgraph. i.e., $G' = G$ is allowed)
  - $G'$ is an APC graph
  - if $G'$ is a *strict* subgraph of $K$ which, in turn, is a subgraph of $G$ (possibly $K = G$), then $K$ is *not* an APC graph: $K$ contains at least one vertex $x$ with either $K.inDegree(x) = 0$ or $K.outDegree(x) = 0$.

- Each graph $G$ either has no APC subgraph or else it has a *unique* maximal APC subgraph.

The drawing below shows an example of a graph $G$ and its maximal APC subgraph $G'$:


Describe (possibly, draw) the maximal APC subgraph for each of the graphs a), b) and c).




You shall *modify* the `TS`-algorithm (which you implemented in the prvious subproblem), so that it returns the maximal APC delgraph or else throws an exception if the argument graph does not contain such a subgraph.

1. Explain the principle for the modified algorithm
2. Program the algorithm as the method

   ```
   Graph maxAPC(Graph G, LiFi Q, LiFi R)
   ```

   The method is allowed to modify the input graph `G`. (If your algorithm does not work for arbitrary implementation of `LiFi`, you shall mark it with a more specific typing of the formal parameters. If you do not need all the input parameters, you may modify the profile of the method.)
3. Suppose that `G'` is an APC graph (e.g., returned by the above method). Sketch with a precise pseudo-code a simplest possible algorithm which would print out a cycle from `G'`. (You shall *not* use here DFS nor BFS, but you may utilize some additional attributes.)

   [Hint: State first a property of APC graphs which makes use of full DFS/BFS unnecessary.]

*Good luck*

*Michał Walicki*

# Løsningsforslag – eksamen i I-120, 15 Desember 1999

## Opp. 1 – Implementasjon

Stabler peker "mot hverandre" med sine topper

```
public class ELss implements EditorLine {
  private Stack L,R;
  public ELss(Stack l, Stack r) { L=l; R=r; }
  public void left() { if (!L.isEmpty()) R.push(L.pop()); }
  public void right() { if (!R.isEmpty()) L.push(R.pop()); }
  public void insert(Character c) { L.push(c); }
  public void remove() { if (!L.isEmpty()) L.pop(); }
  public void start() { while (!L.isEmpty()) { left(); } }
  public void right() { while (!R.isEmpty()) { right(); } }
}
```

Alle er $\mathcal{O}(1)$ med unntak av de to siste som er – i verste fall – $\mathcal{O}(n)$ der $n$ er antall tegn i hele linjen.

## Opp. 2 – BST

### 2.1. En direkte utvidelse

Antar at kun interne noder i BST holder dataobjekter.

```
class BSTk extends BST implements OrderedDictionaryK {
  int ind;
  Object findKth(int k) { ind=0; return findKth(root(),k); }
  Object findKth(Position v,int k) {
    if (isExternal(v)) { return null; }
    else {
       Object r= findKth(v.leftChild(),k);
       ind++;
       if (k < ind) return r;
       else if (k==ind) return v.element();
       else return findKth(v.rightChild(),k);
    }
} }
```

### 2.2. En forbedring

Antar en global `Comparator cp` i BST. Utvider `class Item` med et attributt `int leftSize` som skal holde antall elementer (interne noder) i venstre subtre av en gitt posisjon.

```
class Item {
  private int leftSize; ...
  public Item(Object k, Object e, int l) {
    leftSize=l; ... }
  public int leftSize() { return leftSize; }
  public void incLS() { leftSize++; }
... }

class BSTk2 extends BST implements OrderedDictionaryK {
  Object findKth(int k) { return findKth(root(),k); }
  Object findKth(Position v,int k) {
```

```
    if (isExternal(v)) { return null; }
    else { Item vi = (Item)v.element();
       if (k <= vi.leftSize()) return findKth(v.leftChild(),k);
       else if (k==vi.leftSize()+1) return vi;
       else return findKth(v.rightChild(),k);
    }
  }
  public void insert(Object k, Object e) { settInn(root(), new Item(k,e,0)); }
  private void settInn(Position v, Item i) {
    if (isExternal(v)) {
        expandExternal(v);
        v.setElement(i);
    } else if (cp.isGreaterThan(v.element(),i)) {
        settInn(v.leftChild(),i);
        ((Item)v.element()).incLS();
    } else settInn(v.rightChild(),i);
  }
...}
```

## Opp. 3 – TS

### 3.1.

1. Vi antar at `Vertex` er implementert med `class Node` som har et attributt `public int in`.

```
LiFi TS(Graph G, LiFi Q, LiFi R) {
  Enumeration en = G.vertices();
  while (en.hasMoreElements() {
    Node v = (Node)en.nextElement();
    v.in = G.inDegree(v);
    if (v.in==0) Q.add(v);
  }
  while (!Q.isEmpty()) {
    Node h = (Node)Q.remove();
    en = G.outAdjacentVertices(h);
    while (en.hasMoreElements()) {
      Node v = (Node)en.nextElement();
      v.in--;
      if (v.in==0) Q.add(v);
    }
    R.add(h);
  }
  return R
}
```

2. Dette blir vanlig topologisk sortering: er grafen `G` asyklisk, får vi ut en kø der alle dens noder står i en topologisk ordning: resultat = ABCDEF

3. Dette blir også en topologisk sortering men resultatet vil (typisk) være en annen ordning enn i 2. `R` vil til enhver tid inneholde kun noder $v$ med $v.in == 0$, men her blir de hentet fra køen (stabel) i LiFo ordning: resultat = ACBEDF

4. Dette blir en topologisk sortering men resultatet blir en stabel med noder i omvend rekkefølge til den aktuelt konstruerte topologiske ordningen – noden på topen av stabelen er den siste, den under nest siste, osv.: resultat = FDEBCA.

### 3.2. maxAPC

Maksimal APC delgraf av a) er tom, av b) er det en uten noder A og E og av c) er hele grafen c).

1. Algoritmen trenger ikke output LiFi siden den skal nå returnere en delgraf. Essensielt, så kjører vi først TS' som fjerner alle noder som går gjennom `Q` fra grafen `G` (dette innebærer at også nabo-kanter til disse nodene blir fjernet). Deretter kjører vi den samme TS' på den reverserte grafen. Resulterende graf blir den vi trenger (eller blir den tom - da finnes det ikke en maksimal ikke-null delgraf.)

2. I implementasjon bruker vi heller `Node implements Vertex` med to attributter `public int in, ut` og kjører en modifisert TS en gang som fjerner noder med inn- eller utgradstall likt 0.

```
Graph maxAPC(Graph G, LiFi Q) throws TomDelgrafUnntak {
  Enumeration en = G.vertices();
  while (en.hasMoreElements() {
    Node v = (Node)en.nextElement();
    v.in = G.inDegree(v);
    v.ut = G.outDegree(v);
    if (v.in==0 || v.ut==0) Q.add(v);
  }
  while (!Q.isEmpty()) {
    Node h = (Node)Q.remove();
    if (h.in==0) {
      en = G.outIncidentEdges(h);
      while (en.hasMoreElements()) {
        Edge e = (Edge)en.nextElement();
        Node v = (Node)e.destination();
        v.in--;
        if (v.in==0) Q.add(v);
    } }
    if (h.ut==0) {
      en = G.inIncidentEdges(h);
      while (en.hasMoreElements()) {
        Edge e = (Edge)en.nextElement();
        Node v = (Node)e.origin();
        v.ut--;
        if (v.ut==0) Q.add(v);
        G.removeEdge(e);
    } }
    G.remove(h);
  }
  en = G.vertices();
  if (!en.hasMoreElements()) throw new TomDelgrafUnntak();
  else return G;
}
```

3. Velg en vilkaarlig node v fra G og kjør (initielt er all noder og kanter umerket):

```
Syklus(Node v) {
  if (merket(v)) skrivSyklus(v)
  else {
    merk v
    velg en vilkaarlig utgaaende kant fra v e=(v,x)
    merk e
    Syklus(x)
} }
```

```
skrivSyklus(v) {
  if (merket(v))
    skriv v
    umerk v (fjern merkning av v)
    gaa langs den merkete kanten (v,x) og
    fortsett skrivSyklus(x)
}
```