Universitetet i Bergen
Det matematisk-naturvitenskapelige fakultet

English

Examination in	:	I-120 Algorithms, Data Structures og Programing
Date	:	11 December 2002
Time	:	9:00 – 15:00
No. pages	:	3 (+ appendix)
Auxiliary materials	:	All printed and written

- You may use all `interface`s and data strucutres from the syllabus – use preferably `interface`s and avoid unnecessary use of concrete data types.

  (The appendix contains some `interface`s which may be useful in the solutions.)
- You may assume availability of implementations of the `interface`s used in your programs.
- Formulate explicitly all additional assumptions which you make and which are not specified in the problem description.
- If you do not have enough time for writing a full program, you can get almost full credit for pseudo-code with sufficient comments.
- If you can't solve a (sub)problem, you may nevertheless continue and assume that its result is available in later solutions.
- Percentages at each (sub)problem suggest approximate weight at evaluation (*not necessarily* the expected time/workload needed for producing an answer).

1 Stack (10%)

We have some implementation(s) of `interface BlackWhite { boolean isWhite();}`. Write a method `boolean equalNumber(ObjectIterator I, Stack S)` which uses *only one* stack (the actual parameter `S` at the method call is assumed to be an empty stack) and which returns `true` if and only if the parameter iterator `I` contains the same number of objects with `isWhite()==true` as with `isWhite()==false`. (`I` contains a series (of unknown length) of only such objects, appearing in an arbitrary order).

2 Binary trees (50%)

You are given (an implementation of) the `interface Binary Tree` (see appendix) which, in addition, satisfies some data invariant – in subproblem 2.1 the invariant for binary search trees, while in 2.2 the invariant for heap.

We assume that keys are integers with the usual ordering/comparison and are stored in the `element()` attribute of the tree's `Position`s. If necessary, you may assume that `int k` is given as an appropriate key-object, e.g., `Integer K`.

2.1 Search trees (25%)

1. Formulate the invariant/condition which a binary tree must satisfy to be a search tree, BST.
2. Draw the BST resulting from the insertions of elements with the keys: 5, 3, 8, 6, 9, 7, 1, 4 (in this order, i.e., first 5, then 3, then 8, etc.). The insertion is assumed to respect only the search invariant and does not perform any additional balancing of the tree.
3. Draw the BST resulting from the removal of the element with key 5 from the above tree.
4. Program a method `void less(BinaryTree T, int x)` which prints out all keys (integers) collected in the search tree `T` and which are less than or equal to `x` (all `k` such that `k<=x`).

5. Which numbers – and *in which order* – will be printed when we call your method `less(T,7)`, with the tree `T` which you obtained in point 2?
6. What is the worst-case complexity of your method (expressed as a function of n – the number of object in the tree)?

2.2 Heap (25%)

1. Formulate the invariant/condition which a binary tree must satisfy to be a heap.
2. Draw the heap which will result from inserting the elements with the keys: 7, 5, 9, 3, 8, 7, 1 (in the given order, i.e., first 7, then 5, then 9, etc.).
3. Draw so the heap which one obtains after the removal of the least element from the heap you drew in point 2.
4. Program a method `void less(BinaryTree T, int x)` which prints out all keys (integers) collected in the binary tree `T` – which is a heap – and which are less than or equal to `x` (all `k` such that `k<=x`).

   The method shall have the complexity $\mathcal{O}(m)$, where m is the number of printed elements, i.e., those with key `k<=x`.
5. Which numbers – and *in which order* – wil be printed when we call your method `less(T,7)`, where `T` is the tree you obtained in point 2 of this subproblem?

3 Graphs and generic programming (40%)

We work with directed graphs, and consider three variants of such graphs:

i. Edges have no attributes

ii. Edges have an integer attribute, weight – $weight \geq 0$;

iii. Edges have an integer attribute, color – $1 \leq color \leq F$, where F is the total number of possible colors.

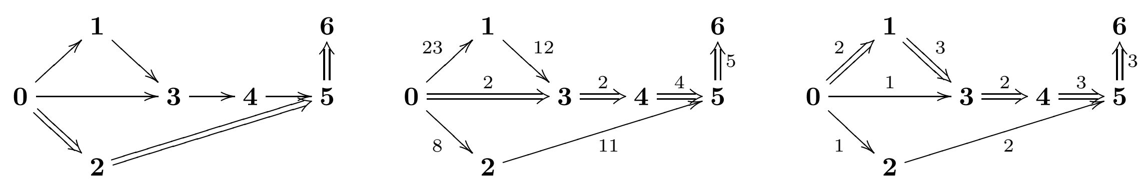
Given a directed graph $G = (V, E)$, and vertices $a, b \in V$, depending on the graph variant, we can ask essentially the same question in three different versions:

i. What is (the path with) the minimal number of edges from a to b?

ii. What is (the path with) the least total weight from a to b?

iii. What is (the path with) the least number of colors from a to b?

Below there is an example of three variants of the graph – i. without any attributes, ii. with weights and iii. with colors – together with the correct answer (marked with the double arrows and the number below) to the respective question for the vertices $a = \mathbf{0}, b = \mathbf{6}$:



i. : 3 *ii.* : 13 *iii.* : 2

The problem is to program *one generic* method with appropriate parameter(s) which allows one to answer all these questions when called with the respective graph variant and an adequate implementation of the parameter interface. (One can use an algorithm which has been presented at the lectures – see the Hint after the last subproblem.)

3.1 (30%)

1. Declare first an appropriate `interface Cp` which abstracts away the common features of the first two cases, i.e., i. directed graphs without any attributes on edges, and ii. directed graphs with weights. Explain briefly the intension of each method. (Preferably, `Cp` shall be used also for the third case, iii., but here you will get full credit even if it can not be used there.)

2. Program so the generic method

   $$\texttt{int SSSP}(\texttt{Graph G}, \texttt{Node a}, \texttt{Node b}, \texttt{Cp V})$$

   which shall give answers to these two questions. As a result we do not require the actual path, but only the corresponding number (3 for the graph i. and 13 for the graph ii.).

   [You do not have to worry so much about efficiency here (nor in 3.2 below). The primary issue is correctness and genericity.]

3. Give two implementations:

   i. `class NoEdgeCp implements Cp`, and
   ii. `class WeightCp implements Cp`

   such that the call

   i. `SSSP(G,a,b,new NoEdgeCp())` with a given graph `G` without weights, gives the correct answer to the question i. – number of edges on the shortest path from `a` to `b`, while
   ii. `SSSP(G,a,b,new WeightCp())` with a given graph `G` with weights on edges, gives the correct answer to the question ii. – the weight/length of the shortest path from `a` to `b`.

You will have to mark vertices/edges in the graph with some extra values and will therefore utilize the fact that they extend `interface Decorable`. You can assume that objects identifying appropriate attributes are given (e.g., `DD, weight`, etc.) but make such assumptions explicit in your answer.

3.2 (10%)

Program whatever is needed to handle the third case iii. of graphs with colored edges. Your method from point 2 in subproblem 3.1 should work unchanged – the answer shall now be the minimal number of colors on the edges forming a path from `a` to `b`. The main problem will be an appropriate implementation of `class ColorCp implements Cp`. Explain first the main idea of your solution. (Precise and well explained pseudo-code may give approximately the same as a detailed program-code.)

[Hint: Notice that, in the example iii. above, we can not assume that the solution is constructed incrementally, as it is done for instance in Dijkstra's algorithm. A solution for veritces **0-5** could be **0-2-5**, but the solution for the vertices **0-6** is not a atraightforward extension of that.

One has to collect, for each vertex, all possible paths leading to it from the start vertex, and in each iteration the algorithm has to take into account (the relevant among) all such possibilities.

If needed, you may make some simplifying assumptions, e.g., that you have available methods which convert uniquely between vertices/edges and some integers. Whatever your additional assumptions are, state them clearly in the answer.]

Michał Walicki Arild Waaler

Solution – prob. 1

We push new object (from the iterator) on the stack if either the stack is empty or else the top object is of the same kind as the one we are trying to push (isWhite()==true for both or for neither). If the two are of different kind, we merely pop the top of the stack. The answer is 'equal' if on termination the stack is empty. The idea is as follows:

```
boolean equalNumber(ObjectIterator I, Stack S) {
   BlackWhite o; // sorry, need this extra variable
   while (I.hasMore()) {
     o = (BlackWhite)I.nextObject();
     if (S.isEmpty()) S.push(o);
     else if ( ((BlackWhite)S.top()).isWhite() ) {
       if (o.isWhite()) S.push(o);
       else S.pop(); }
     else {
       if (!o.isWhite()) S.push(o);
       else S.pop(); }
   return S.isEmpty();
}
```

To avoid using *any* extra variables (i.e., `o`), we utilze the method `object()` from the `ObjectIterator` which observes the most recent object returned by `nextObject()`:

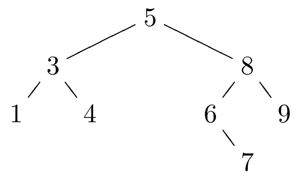
```
boolean equalNumber(ObjectIterator I, Stack S) {
   while (I.hasMore()) {
     if (S.isEmpty()) S.push(I.nextObject());
     else if ( ((BlackWhite)S.top()).isWhite() ) {
       S.push(I.nextObject());
       if ( !((BlackWhite)I.object()).isWhite() ) {
         S.pop(); S.pop(); } //remove both the current black and previous white
     } else { // top object on S is Black
       S.push(I.nextObject());
       if ( ((BlackWhite)I.object()).isWhite() ) {
         S.pop(); S.pop(); } //remove both the current white and previous black
   } }
   return S.isEmpty();
}
```

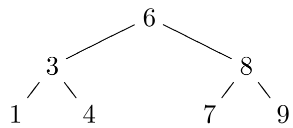
Solution – prob. 2

2.1. BST

1. for each node: all keys in the left subtree are not-greater, and all in the right subtree are not-smaller than the node's key

2.-3. The tree for 5, 3, 8, 6, 9, 7, 1, 4 will be as on the left, and after removal of 5 as on the right:





4. `void less(BinaryTree T, int x) { less(T, T.root(), x); }`

```
void less(BinaryTree T, Position v, int x) {
   if (v != nil) {
     if (((Integer)v.element()).intValue() <= x) {
       print(v.element());
       less(T, T.leftChild(v), x);
       less(T, T.rightChild(v), x); }
     else less(T, T.leftchild(v), x);
} }
```

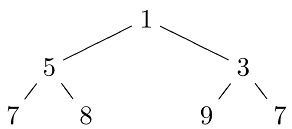
5. This question concerns in-, pre- or post-order. We made it pre-order, so our result will be: 5,3,1,4,6,7.

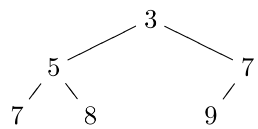
6. $\mathcal{O}(n)$, when the tree is unbalanced, essentially a linear structure.

2.2. Heap

1. each node $\leq$ all its children

2.-3. the tree will be as on the left, and after removal as on the right:





4. `void less(BinaryTree T, int x) { less(T, T.root(), x); }`

```
void less(BinaryTree T, Position v, int x) {
   if (v != nil) {
     if (((Integer)v.element()).intValue() <= x) {
       print(v.element());
       less(T, T.leftChild(v), x);
       less(T, T.rightChild(v), x); }
} }
```

Here we can terminate the recursion at once we find something bigger than x, since in a heap everything below will be biger, too. We get $\mathcal{O}(m)$, since we write everything until we reach this condition, as all smaller than... is stored in the "upper part" of the heap.

5. Again, this is pre-order, so we will get: 1,5,7,3,7.

Solution – prob. 3

The first part could be obtained by a generalization of Dijkstra but, as sugggested in the hint, the second could not. We use Ford-Bellmann from start.

3.1. Number of edges and Length

1. ```
public interface Sml {
    /* for initializing the X-attribute in the node v with MAX_VALUE */
    void initMax(Vertex v,Object X);
    /* for initializing (the starting) v with minimal value */
    void initMin(Vertex v,Object X);
    /* returns the value of the test for update,
      wrt. the X-atribute, e.g. X[c]+e.weight<X[d] */
    boolean less(Vertex c,Vertex d,Edge e,Object X);
    /* performs the update, if the test was true */
    void update(Vertex c,Vertex d,Edge e,Object X);
    /* converts the value stored in the X-atribute of v to int */
    int intValue(Decorable v,Object X);   }
```

2. Assume DD is a globally available Object identifying the respective atribute in the Vertex (which extends Decorable interface). Ignored all the exceptions which the metods from the given interface may rise.

```
int SSSP(Graph G, Node a, Node b, Sml V) {
    EdgeIterator edges; Vertex v,c,d; Edge e;
    int n= G.numVertices();
    VertexIterator nodes= G.vertices();
    while (nodes.hasNext()) {
      v= nodes.nextVertex();
      V.initMax(v,DD); }
    V.initMin(a,DD);
    for (int i=1;i<n;i++) {
      edges= G.directedEdges(); // .edges() should be the same as G is directed
      while (edges.hasNext()) {
        e= edges.nextEdge();
        c= G.origin(e); d= G.destination(e);
        if (V.less(c,d,e,DD)) V.update(c,d,e,DD);
      }
    }
    return V.intValue(b,DD);
}
```

3. ```
class AntallSml implements Sml {
    public void initMax(Vertex v,Object X) {v.set(X,new Integer(Integer.MAX_VALUE);}
    public void initMin(Vertex v,Object X) { v.set(X,new Integer(0)); }
    public boolean less(Vertex c,Vertex d,Edge e,Object X) {
      if (intValue(c,X)==Integer.MAX_VALUE) return false;
      // this case is necessary to avoid overflow (wrap around for int in Java)
      else return intValue(c,X)+1 < intValue(d,X) ; }
    public void update(Vertex c,Vertex d,Edge e,Object X) {
      d.set(X, new Integer(intValue(c,X)+1); }
    public int intValue(Decorable v,Object X) {
      return (Integer(v.get(X))).intValue(); }
  }
```

Assume that also Edge uses the DD-atribute for storing weight:

```
class VektSml implements Sml {
   public void initMax(Vertex v,Object X) {
     v.set(X,new Integer(Integer.MAX_VALUE)); }
   public void initMin(Vertex v,Object X) { v.set(X,new Integer(0)); }
   public boolean less(Vertex c,Vertex d,Edge e,Object X) {
     if (intValue(c,X)==Integer.MAX_VALUE) return false;
     else return (intValue(c,X)+intValue(e,X) < intValue(d,X)); }
   public void update(Vertex c,Vertex d,Edge e,Object X) {
     d.set(DD, new Integer( intValue(c,X)+intValue(e,X) ); }
   public int intValue(Decorable v,Object X) {
     return (Integer(v.get(X))).intValue(); }
}
```

3.2. Colors

We do what the Hint suggests. The DD for a node v will hold the array $From$, with indices corresponding to the incoming edges. I.e., we assume given 1-1 functions between

$$\begin{array}{rcll} edgeNo: & G.incidentEdges(v, EdgeDirection.IN) & \longrightarrow & [0...k] \\ & G.incidentEdges(v, EdgeDirection.IN) & \longleftarrow & [0...k] \times v \quad : edgeOf \end{array} \tag{1}$$

where $k+1$ is the number of incoming edges. Then, $From[i]$ is a sequence with objects corresponding to all the paths leading to v with the last edge being $i - v$.

Each path to v is identified with the colors it uses, i.e., is itself a set (sequence) of numbers $\leq F$. We represent it by, first, associating each color $1 \leq f \leq F$, with the $(f+1)$-th prime number $prim(f)$, and then the set of colors $S = \{f_1, f_2, ..., f_n\}$ (with $f_i \neq f_j$ for $i \neq j$) is simply the number $prim(S) = prim(f_1) \cdot prim(f_2) \cdot ... \cdot prim(f_n)$. (We take $(f+1)$-th prime number to avoid using 1, i.e., the first color $f = 1$ gets the number 2, the second 3, the third 5, etc.) Membership is then given by:

$$f \in S \iff prim(S)\ mod\ prim(f) = 0. \tag{2}$$

For convenience, we also let each v keep the sequence $paths$ of all paths leading to it, i.e., a kind of flattening of the whole $From$. (This is the invariant which makes a lot things simpler.)

The initial value (corresponding also to the MAX_VALUE), for each node, is array where $From[i].isEmpty()$ for all i, i.e., $paths.isEmpty()$. Filling in happens when, considering the edge $i - v$, the $From$ array in i is less than MAX_VALUE (has some paths) and $From[i]$ in v has fewer elements than the total number of paths in $From$ in i, i.e.,

$$!i.paths.isEmpty() \ \ \&\& \ \ v.From[i].size() < i.paths.size() \tag{3}$$

The latter means that some new paths leading to i have appeared since we considered the edge $i - v$ last time. We keep also the invariant that later appearing paths are put at the end of the $paths$. Copying the new paths to i and extending them with $i - v$ color for putting in $v.From[i]$, will thus concern only the $i.paths.size() - v.From[i].size()$ number of the paths from the end of $i.paths$.
We assume that in the environment where the following class is defined, we have access to the Graph G, the attribute Object DD, and the number of colors used by the G as well as the corresponding prim-function (other functions come from class Info on the following page):

```
class FargerSml implements Sml {
   private Graph G; private Object Attr; private int noF;
   public FargerSml(Graph Gr,Object X,int F) { G=Gr; Attr=X; noF=F;}
   public void initMax(Vertex v,Object X) {
     int no= 0;
     EdgeIterator ei= G.incidentEdges(v,EdgeDirection.IN);
     while (ei.hasNext()) { ei.nextEdge(); no= no+1; }
     v.set(X,new Info(no,v,Attr,noF)); }

// Assume that initMin is called on v after initMax and before any other updates
// i.e., v.From[i].isEmpty() for all i and v.paths.isEmpty()
   public void initMin(Vertex v,Object X) { of(v,X).initOne(); }

   public boolean less(Vertex c,Vertex d,Edge e,Object X) {
     return (!of(c,X).isEmpty() &&
              of(c,X).paths.size() > of(d,X).sizeOf(edgeNo(e))); }

// Assume that e:c->d and the conversion function edgeNo, as explained in (1).
   public void update(Vertex c,Vertex d,Edge e,Object X) {
     of(d,X).add(edgeNo(e),of(c,X).paths); }

   public int intValue(Decorable v,Object X) { return of((Vertex)v,X).intValue();}
   private Info of(Vertex v, Object X) { return (Info)v.get(X); }
} // end FargerSml
```

```
public class Info { // objects to be stored in the DD-attribute for each node v
   private Vertex v; // vertex for which this is the Info
   private int k; // number of incoming edges to v
   private int noF; // number of colors (assume given prim-function)
   private Sequence[] From; // will have k entries
   private Object Attr; // the attribute to work on (could be more dynamic)
   protected Sequence paths= new SequenceImpl();

   public Info(int i,Vertex u,Object X,int F) {
     k= i; From= new Sequence[k]; v= u; Attr= X; noF= f;
     for (int z=0;z<k;z++) From[z]= new SequenceImpl(); }
   private int intValue(Decorable d,Object X) {
     return ((Integer)d.get(X)).intValue(); }
   private int intObj(Object d) { return ((Integer)d).intValue(); }

   public void add(int i,Sequence pth) {
// extend From[i] with ``last'' elements from pth (adding i-v);
// assuming:  pth.size() >= From[i].size(), and !pth.isEmpty()
   int seq; Position q,p;
     q= pth.first();
     if (!From[i].isEmpty()) { //iterate its size through pth
       p= From[i].first();
       while (p != From[i].last()) { p= From[i].after(p); q= pth.after(q); }
       q= pth.after(q); }
// now copy and extend the rest of pth:  add the color of i-v, if needed
     int col= prim(color(i)); // color gives int stored in Attr
     while (!pth.isLast(q)) { // each element/int in pth represents a set of colors
       seq= intObj(q.element()); // the set stored at q
       if (seq%col != 0) seq= seq*col; // add col if not in seq
       From[i].insertLast(new Integer(seq));
       paths.insertLast(new Integer(seq));
       q= pth.after(q); }
     seq= intObj(q.element()); // the sequence stored at the last Position
     if (seq% col != 0) seq= seq*col;
     From[i].insertLast(new Integer(seq));
     paths.insertLast(new Integer(seq));
   } // end add(...)

   public int intValue() { // the minimal number of colors at any From[i]
     int mi= 0; int seq,no;
     for (int i=0;i<k;i++) {
       ObjectIterator ith= From[i].elements();
       while (ith.hasNext()) {
         seq= intObj(ith.nextObject()); no= 0;
         for (int k=1;k<=noF;k++) { if (seq%prim(k) != 0) no= no+1; }
         if (no<mi) mi= no;
     } }
     return mi;
   } // end intValue()

   public int sizeOf(int i) { return From[i].size(); }
   public boolean isEmpty() { return paths.isEmpty(); }
   public void initOne() { paths.insertFirst(new Integer(1)) }
   private int color(int i) { // Edge stores color in Attr; edgeOf - see (1)
     return ((Integer)edgeOf(i,v).get(Attr)).intValue(); }
} // end Info
```