Relazione

Progetto di Laboratorio di Programmazione

“Il Pedaggio”

Introduzione

Sindbad il marinaio vendette 66 cucchiai d'argento al sultano di Samarcanda. La vendita fu parecchio semplice, ma la consegna fu alquanto complicata. I pezzi furono trasportati via terra, attraversando diversi villaggi e città. Per entrare in ciascuna città e in ciascun villaggio veniva richiesto un pedaggio. Non c'era invece nessun pedaggio da pagare per uscire. Il pedaggio per entrare in un villaggio era uno degli oggetti trasportati. Per entrare in una città, invece, bisognava pagare un pezzo per 20 oggetti trasportati. Per esempio, per entrare in una città trasportando 70 pezzi, bisognava pagare 4 pezzi come pedaggio. Villaggi e città erano situati strategicamente tra rupi, paludi e fiumi così da non poter evitarne il passaggio.
Preventivare i pedaggi da pagare in ogni città e villaggio è abbastanza semplice, ma trovare il percorso migliore (quello più economico) da percorrere è una vera e propria sfida. Il migliore percorso dipende dal numero degli articoli trasportati. Per numeri fino a 20, villaggi e città tassano nello stesso modo. Per numeri più grandi, può avere senso evitare le città e viaggiare attraverso i villaggi. L'obiettivo è scrivere un programma che risolva il problema di Sindbad. Dato il numero dei pezzi da consegnare ad un certo villaggio o ad una certa città della mappa, il programma deve determinare il numero di pezzi richiesti all'inizio del viaggio prima di percorrere la via più economica.

Formato dell'input

L'input consiste di un file contenente diversi casi da esaminare. Ogni caso consiste di due parti: la mappa stradale seguita da dettagli utili per la consegna. La prima riga della mappa stradale contiene un intero n, che rappresenta il numero di strade nella mappa (n>=0). Ognuna delle successive n righe contiene esattamente due lettere che rappresentano i due estremi della strada. Una lettera maiuscola rappresenta una città, una lettera minuscola un villaggio. Le strade possono essere attraversate in entrambe le direzioni.
Dopo la descrizione della mappa, segue un'unica riga contenente i dettagli di consegna. Questa riga consiste di 3 cose: un intero p (0 <= p <= 1000) per il numero di articoli che devono essere consegnati, una lettera che indica il posto di partenza e una lettera per il posto della consegna. La mappa stradale è tale che la consegna sia sempre possibile. L'ultimo caso da analizzare è seguito da una riga contenente il numero -1.

Formato dell'output

L'output consiste di un'unica riga per ogni caso analizzato. Ogni riga contiene il numero che

contraddistingue il caso analizzato e il numero degli articoli richiesti all'inizio del viaggio.

Esempio:

input.txt
1
a Z
19 a Z
-1

schermo
Caso 1: 20

Analisi Funzionale

Parsing dell'input

I dati acquisiti dall'input vengono memorizzati in due strutture temporanee: un array contenente gli archi del grafo, un albero binario di ricerca contenente i nodi. L'array degli archi è un array di record a due campi char, ognuno dei quali contiene i due nodi estremi di un arco. I nodi dell'ABR contengono un campo char con l'etichetta dei nodi, e un campo integer con un numero ID progressivo da 0 al numero dei nodi - 1.

La costruzione dell'array degli archi ha complessità lineare, cioè O(n), dove n è il numero degli archi. La costruzione dell'ABR dei nodi avviene tramite la funzione di inserimento ordinato, che ha complessità O(log n), ripetuta per ogni estremo degli archi, quindi complessivamente ha una complessità di O(2n log n) = O(n log n), dove n è sempre il numero degli archi.

Non vengono effettuati controlli sugli errori dell'input, in quanto le specifiche del problema garantiscono l'integrità e la correttezza formale e logica dell'input.

Generazione delle strutture principali

La mappa è trattata internamente come un grafo non orientato, implementato tramite liste di adiacenza. La costruzione di questo grafo avviene a partire dalle due strutture create in fase di parsing, con l'utilizzo delle due funzioni *graph_set_arc* e *bst_get_id_from_inf*.

La prima serve per impostare un arco bidirezionale tra i due nodi passati come parametri, la seconda restsituisce l'ID associato all'etichetta del nodo passato come parametro.

L'inserimento degli archi ha una complessità O(1), mentre la ricerca degli ID sfrutta la struttura

ordinata dell'ABR ottenendo una complessità nel caso medio pari a O(log n) dove n è il numero dei nodi dell'ABR, nel caso peggiore la complessità è pari a O(n).

Ricerca del cammino minimo sul grafo (Dijkstra)

La ricerca del cammino minimo avviene tramite l'applicazione dell'algoritmo di Dijkstra.

Per spiegare il funzionamento del grafo procediamo con l'analisi di un grafo di esempio, ottenuto dal seguente input:

```
12
A C
A b
A D
A a
a b
a D
C b
C D
C X
b c
c X
X D
19 a X
```

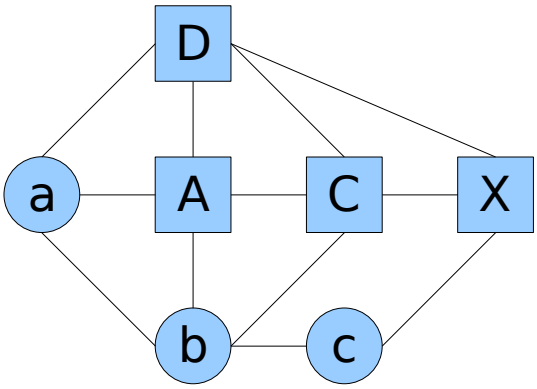

fig. 1 Il grafo appena creato

L'ultima linea di input specifica il numero di cucchiai da consegnare alla città **X** partendo dal villaggio **a**. Applico l'algoritmo di ricerca partendo dalla città di destinazione **X**, assegnandole un peso iniziale di **19** (cucchiai). Tutti gli altri nodi hanno un peso iniziale settato a +∞ (internamente rappresentato con la costante `MAXINT`).

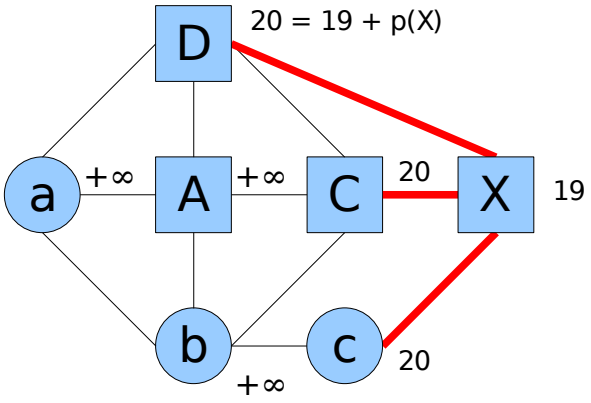

fig. 2 Esplorazione dei nodi adiacenti ad **X**

Controllo i nodi adiacenti a **X** e se il loro peso è maggiore del peso di **X** più il pedaggio (che corrisponde alla funzione peso associata agli archi), allora aggiorno gli adiacenti con il nuovo peso come in **fig. 2** (*rilassamento* del peso dei nodi) e inserisco in una coda di priorità i nodi **D**, **C** e **c** con il rispettivo peso. Elimino **X** dalla coda dei nodi da visitare, il suo peso è stabilizzato a 19. Estraggo dalla coda il nodo con peso minimo e reitero l'algoritmo.

Alla seconda iterazione, il grafo si trova nella situazione di **fig. 3**, dove l'algoritmo è stato applicato al nodo **D**. Notare che i pesi di **C** e **X** sono rimasti invariati perché inferiori alla soglia di rilassamento.

Poiché sono stati visitati tutti gli adiacenti di **D**, il suo peso è stabile, per cui lo elimino dalla coda e

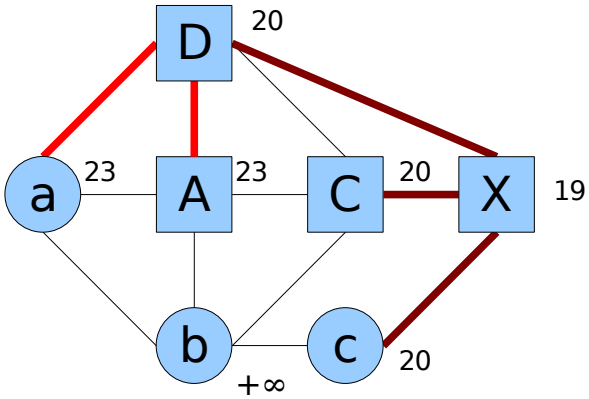

fig. 3 Esplorazione dei nodi adiacenti a **D**

procedo nuovamente con l'estrazione del nodo di peso minimo. Reiterando ancora una volta il procedimento, ci ritroviamo in una situazione simile a quella di **fig.4**, dove vengono esplorati i nodi adiacenti a **C**.

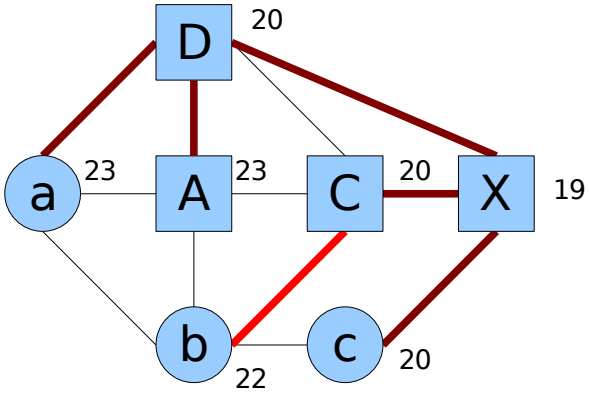

fig. 4 Esplorazione dei nodi adiacenti a **C**

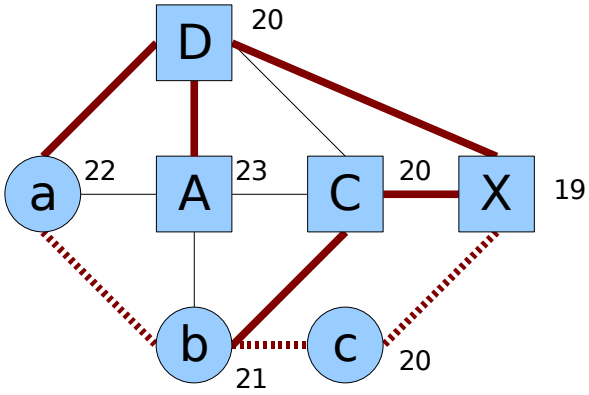

fig. 5 Grafo esplorato integralmente.
Il percorso minimo da **X** ad **a** è segnato con la linea tratteggiata

Alla fine dell'applicazione dell'algoritmo, la situazione del grafo è quella mostrata in **fig. 5** dove tutti i pesi sono stati stabilizzati. Notare che il peso di **b** e di **a** è stato modificato (rilassato) passando attraverso percorsi meno costosi. La struttura

L'efficienza dell'algoritmo in genere è pari a O(A + N^2), dove A è il numero degli archi e N il numero dei nodi. Per aumentare l'efficienza, la coda viene implementata tramite un ABR autobilanciato, quindi inserimento ed estrazione hanno una complessità nel caso medio di O(log n), dove n è il numero di nodi del grafo. Questa ottimizzazione porta la complessità a O((A + N) log N). L'algoritmo termina quando la coda è vuota, quindi il grafo è stato visitato per intero, oppure quando la distanza del nodo di arrivo è stabilizzata. Per ottenere questo effetto viene utilizzata una colorazione parziale del grafo: viene colorato soltanto il vertice di arrivo quando la sua distanza è stabile.

Nota: Leggere in chiave "critica" la parte sull'efficienza; Il nostro calcolo all'esame si è rivelato erroneo.

Appendice A

Sorgente principale

“pedaggio.lpr”

```
program pedaggio;

uses
     tkb_graph in 'tkb_graph.pas',
       tkb_bst in 'tkb_bst.pas',
 tkb_pri_queue in 'tkb_pri_queue',
 SysUtils ;

type arc = record
         x,y: char;
         end;
     arc_v = array of arc;

var g: t_graph;          // Grafo rappresentante la mappa
    x: integer;          // Numero degli archi
    a: t_bst;            // BST usato per parsare l'input
    f: textfile;         // File di input
    arcs : arc_v;        // Vettore contenente gli archi da settare nel grafo
    num_nodes: integer; // Numero dei nodi del grafo
    i: integer;          // Contatore ausiliario
    n1,n2: char;         // Variabili ausiliarie per parsare l'input
    caso: integer;       // Contatore della mappa corrente

{Questa funzione ricorsiva serve a settare il tipo di nodo.
Influenza il calcolo dei pesi nell'esplorazione dei cammini del grafo.}
procedure set_cities(g: t_graph; a: t_bst);
begin
     if a<>nil then begin
        set_cities(g, a^.sx);
        {Le città sono indicate con lettere maiuscole.}
        graph_set_node_type(g, a^.id, (a^.inf = upcase(a^.inf)));
        set_cities(g, a^.dx);
     end;
end;

begin
     if fileexists('input.txt') then begin
       assign(f, 'input.txt');
       reset(f);
       caso := 0;  {Contatore del loop}
       readln(f, x);
       repeat      {Loop principale}
         a:=nil;
         inc(caso);

         {Il vettore arcs contiene gli archi da settare nel grafo}
         setlength(arcs, x);

         {Creo un albero binario di ricerca (BST) con struttura modificata}
         for i := 0 to length(arcs)-1 do begin
             readln(f, n1, n2, n2);   {Salto lo spazio leggendo 2 volte n2}
             bst_insert(a, n1);
             bst_insert(a, n2);
             arcs[i].x := n1;
```

```
            arcs[i].y := n2;
        end;

        {Adesso l'albero binario 'a' contiene tutti i nodi del grafo.
        Associo ad ogni nodo dell'albero un numero intero partendo da 0.
        Serve per fare corrispondere ad ogni nodo dell'albero uno ed un solo
        nodo del grafo.}
        num_nodes := bst_assign_ids(a);

        {Inizializzo il grafo}
        graph_init_nodes(g, num_nodes);

        {Adesso procedo a settare gli archi nel grafo}
        for i:=0 to length(arcs)-1 do
            graph_set_arc(g, bst_get_id_from_inf(a,arcs[i].x),  {Nodo 1}
                             bst_get_id_from_inf(a,arcs[i].y)   {Nodo 2}
                             );

        {Imposto il tipo dei nodi: città o villaggio}
        set_cities(g, a);

        {Leggo il numero di cucchiai, la partenza e la destinazione}
        readln(f, x, n1, n1, n2, n2);

        {Calcolo il cammino minimo e stampo il risultato.
        Notare che i nodi di partenza e destinazione sono invertiti rispetto
        all'input. Vedere la relazione per una spiegazione del funzionamento
        dell'algoritmo.}
        writeln('Caso ',caso,': ',
                        graph_get_best_path(g, bst_get_id_from_inf(a,n2),
                                               bst_get_id_from_inf(a,n1), x));

        {Pulizia dell'heap.}
        graph_dispose(g);
        bst_dispose(a);

        {Proseguo con la lettura del file}
        readln(f, x);
      until x = -1; {quando trova -1 nel file, termina il loop}
      close(f);
    end
    else writeln('Errore nell''apertura di input.txt');
    readln;
end.
```

Unit

“tkb_graph.pas”

```
unit tkb_graph;

{Graph - Grafi liste concatenate di adiacenza - by TheKaneB

Questa Unit implementa una versione modificata della BFS per
risolvere il problema dei "Pedaggi" della sessione di esami di
Giugno 2007}

interface

uses tkb_pri_queue;
type
    adj_list = ^adj_list_node;
    adj_list_node = record
                  node: integer;
                  next: adj_list;
                  end;
    graph_node_type = boolean; {true = città -\\--> false = villaggio}
    graph_node_type_v = array of graph_node_type;
    t_graph = record
            matrix: array of adj_list;
            t: graph_node_type_v;
            end;

procedure graph_init_nodes(var g: t_graph; n: integer);
procedure graph_set_arc(var g: t_graph; i,j: integer);
procedure graph_set_node_type(var g: t_graph; i: integer; t: graph_node_type);
procedure graph_dispose(var g: t_graph);
function graph_get_node_type(g: t_graph; i: integer): graph_node_type;
function graph_get_best_path(g: t_graph; i,j: integer; n:integer): integer;

implementation

{Inizializzazione del grafo, n è il numero dei nodi}
procedure graph_init_nodes(var g: t_graph; n: integer);
var i,j: integer;
begin
     setlength(g.matrix, n);
     setlength(g.t, n);
     for i:=0 to n-1 do begin
         g.t[i] := false;
         g.matrix[i] := nil;
     end;
end;

{Questo grafo è sempre NON orientato, secondo le specifiche dell'esercizio}
procedure graph_set_arc(var g: t_graph; i, j: integer);
var aux: adj_list;
begin
     {Arco i --> j}
     new(aux);
     aux^.node := j;
     aux^.next := g.matrix[i];
     g.matrix[i] := aux;

     {Arco j --> i}
```

```
     new(aux);
     aux^.node := i;
     aux^.next := g.matrix[j];
     g.matrix[j] := aux;
end;

{Imposta il nodo come città o come villaggio}
procedure graph_set_node_type(var g: t_graph; i: integer; t: graph_node_type);
begin
     g.t[i] := t;
end;

{Pulizia dell'heap}
procedure graph_dispose(var g: t_graph);
var i: integer;

    {Procedura ricorsiva interna}
    procedure _dispose(var l: adj_list);
    begin
         if l<>nil then begin
            _dispose(l^.next);
            dispose(l);
            l:=nil;
         end;
    end;

begin
     {Scorre tutte le liste di adiacenza}
     for i:=0 to length(g.matrix)-1 do
         _dispose(g.matrix[i]);
     g.matrix := nil;
     g.t := nil;
end;

{Restituisce il tipo di nodo}
function graph_get_node_type(g: t_graph; i: integer): graph_node_type;
begin
     graph_get_node_type := g.t[i];
end;

{Funzione di ricerca del cammino minimo. Si tratta essenzialmente di
una BFS modificata. Per i dettagli dell'implementazione vedere la relazione.}
function graph_get_best_path(g: t_graph; i, j: integer; n: integer): integer;
var Q: t_pri_queue;
    dist: array of integer;
    idx: integer;
    adj : adj_list;
    last_node: boolean;

    {Funzione peso}
    function _p(x: integer): integer;
    begin
         if graph_get_node_type(g, i) then begin    {Città}
            if (x mod 19) = 0 then
               _p := x + (x div 19)
            else
               _p := x + (x div 19) + 1;
         end
         else                                       {Villaggio}
             _p := x + 1;
    end;
```

```
begin
     {Inizializzazione}
     setlength(dist, length(g.matrix));
     for idx := 0 to length(dist)-1 do
         dist[idx]:=MAXINT;
     pri_queue_init(Q);
     last_node := FALSE;

     {Inserisco il nodo di arrivo in coda}
     dist[i] := n;
     pri_queue_push(Q, i, n);

     {BFS}
     while (not pri_queue_empty(Q)) and (not last_node) do begin

           {Prelevo un nodo dalla coda}
           i := pri_queue_pop(Q);
           if i=j then last_node := TRUE;// Colorazione parziale. Vedi
relazione
           adj := g.matrix[i];

           {Scorro la lista degli adiacenti}
           while adj<>nil do begin
                 if dist[adj^.node] > _p(dist[i]) then begin

                   {Aggiorna il peso del nodo adiacente}
                   dist[adj^.node] := _p(dist[i]);

                   {Inserisco il nodo adiacente in coda}
                   pri_queue_push(Q, adj^.node, dist[adj^.node]);
                 end;

                 {Passo al successivo adiacente di i}
                 adj := adj^.next;
           end;
     end;

     {Pulisco la coda e restituisco la distanza del nodo di partenza j}
     pri_queue_dispose(Q);
     graph_get_best_path := dist[j];
end;

end.
```

“tkb_bst.pas”

```
unit tkb_bst;

{Binary Search Tree - by TheKaneB
Versione modificata per l'esercizio "Il Pedaggio"}

interface

type bst_inf = char;
    t_bst = ^bst_node;
     bst_node = record
          id: integer;
          inf: bst_inf;
          sx: t_bst;
          dx: t_bst;
          end;

procedure bst_insert(var a: t_bst; x: bst_inf);
procedure bst_dispose(var a: t_bst);
procedure bst_print(a: t_bst);  // Utile per il debugging
function bst_assign_ids(a: t_bst): integer;
function bst_get_id_from_inf(a: t_bst; x: bst_inf): integer;

implementation

{Inserisco un nodo nell'albero in ordine alfabetico crescente}
procedure bst_insert(var a: t_bst; x: bst_inf);
begin
     if a<>nil then begin{Visito l'albero per cercare la posizione di
inserimento}
        if x < a^.inf then bst_insert(a^.sx, x)
        else if x > a^.inf then bst_insert(a^.dx, x)
     end
          {Non inserirsco duplicati}
     else begin {Inserisco l'elemento al posto giusto}
        new(a);
        a^.inf := x;
        a^.id := 0;
        a^.sx := nil;
        a^.dx := nil;
     end;
end;

{In questo albero modificato gli ID servono a creare una corrispondenza
tra i nodi dell'albero etichettati come char e i nodi del grafo etichettati
come numeri interi. Gli ID sono numeri progressivi da 0 a #nodi - 1.
Restituisce il numero di nodi dell'albero.}
function bst_assign_ids(a: t_bst): integer;
var id: integer;

    procedure _assign_ids(a: t_bst);
    begin
         if a<>nil then begin
            _assign_ids(a^.sx);
            a^.id := id;
            inc(id);
            _assign_ids(a^.dx);
         end;
    end;
```

```
begin
     id := 0;
     _assign_ids(a);
     bst_assign_ids := id + 1;
end;

{Un po' di pulizia ;)}
procedure bst_dispose(var a: t_bst);
begin
     if a<>nil then begin
        bst_dispose(a^.sx);
        bst_dispose(a^.dx);
        dispose(a);
        a:=nil;
     end;
end;

{Cerca un'etichetta e ne restituisce il corrispondente ID.
Serve per creare gli archi nel grafo, etichettati nel file di input
come coppie di char.}
function bst_get_id_from_inf(a: t_bst; x: bst_inf): integer;
begin
     if a=nil then bst_get_id_from_inf := -1    {Albero vuoto, elemento non
trovato}
     else
         if a^.inf = x then bst_get_id_from_inf := a^.id   {Elemento trovato}
         else
             if x < a^.inf then
                if a^.sx <> nil then
                   {Cerca a sinistra}
                   bst_get_id_from_inf := bst_get_id_from_inf(a^.sx, x)
                else bst_get_id_from_inf := -1
             else
                if a^.dx <> nil then
                   {Cerca a destra}
                   bst_get_id_from_inf := bst_get_id_from_inf(a^.dx, x)
                else bst_get_id_from_inf := -1;
end;

{Stampa in ordine dell'albero. Serve solo in fase di debugging,
per controllare che l'albero sia stato creato correttamente}
procedure bst_print(a: t_bst);
          procedure _print(a: t_bst);
          begin
               if a<>nil then begin
               _print(a^.sx);
               writeln(a^.inf, ' ', a^.id);
               _print(a^.dx);
               end;
          end;
begin
     _print(a);
     writeln('End');
end;

end.
```

“tkb_pri_queue.pas”

```
unit tkb_pri_queue;

{Priority Queue - by TheKaneB
Questa Unit implementa una coda dove l'elemento in testa è quello con
il minore peso di tutta la struttura. L'implementazione fa uso di una
struttura ad albero binario di ricerca, basata sulla unit tkb_bst.
La priorità degli elementi viene stabilita dal parametro w, cioè dal loro
peso.}

interface

type pri_queue_inf = integer;
     pri_queue_w = integer;
     t_pri_queue = ^pri_queue_node;
     pri_queue_node = record
                      inf: pri_queue_inf;
                      w: pri_queue_w;
                      sx, dx: t_pri_queue;
                      end;

procedure pri_queue_push(var a: t_pri_queue; i: pri_queue_inf; w: pri_queue_w);
procedure pri_queue_init(var a: t_pri_queue);
procedure pri_queue_dispose(var a: t_pri_queue);
function pri_queue_pop(var a: t_pri_queue): pri_queue_inf;
function pri_queue_empty(a: t_pri_queue): boolean;

implementation

{Inserisco un nodo nella coda. I nodi sono ordinati in ordine crescente
secondo il peso w}
procedure pri_queue_push(var a: t_pri_queue; i: pri_queue_inf; w: pri_queue_w);
begin
     if a=nil then begin {Inserimento}
        new(a);
        a^.sx := nil;
        a^.dx := nil;
        a^.inf := i;
        a^.w := w;
     end
     else {Ricerca della posizione}
        if w < a^.w then
           pri_queue_push(a^.sx, i, w)
        else
           pri_queue_push(a^.dx, i, w);
end;

{Inizializzo la coda}
procedure pri_queue_init(var a: t_pri_queue);
begin
     a := nil;
end;

{Elimino la coda dall'heap}
procedure pri_queue_dispose(var a: t_pri_queue);
begin
     if a<>nil then begin
        pri_queue_dispose(a^.sx);
        pri_queue_dispose(a^.dx);
```

```
        dispose(a);
        a:=nil;
     end;
end;

{Restituisce l'elemento con peso minimo e lo cancella dalla coda.}
function pri_queue_pop(var a: t_pri_queue): pri_queue_inf;

    {Sostituisco un nodo con un figlio, e lo cancello}
    procedure _replace(var a1: t_pri_queue; a2: t_pri_queue);
    var aux: t_pri_queue;
    begin
         aux := a1;
         a1 := a2;
         dispose(aux);
    end;

begin
    {Eseguire esternamente il controllo pri_queue_empty !!!!! }

    {Elemento trovato, eseguo la cancellazione}
    if a^.sx = nil then begin
       pri_queue_pop := a^.inf;
       _replace(a, a^.dx);
    end
    else
       pri_queue_pop := pri_queue_pop(a^.sx);
end;

{Restituisce TRUE se la coda è vuota, altrimenti FALSE.}
function pri_queue_empty(a: t_pri_queue): boolean;
begin
     pri_queue_empty := (a = nil);
end;

end.
```