```
// -*- C++ -*-

// ダイクストラ法のデモ

#include "RS_FHeap.h"
#include "RS_Dijkstra.h"

#include <cstdlib>
#include <iostream>
#include <sstream>

using std :: cin ;
using std :: ostream ;
using std :: ostringstream ;


const int LARGENUM = 10000 ;


int
main
()
{
  ostringstream ss ;

  // Dijkstra's Algorithm: Starting from Node i
  // Creating nodes
  for ( int k = 0 ; k < 15 ; ++ k )
  {
    for ( int i = 0 ; i < 15 ; ++ i )
    {
      ss << i ;

      RS_Dijkstra < int , string > *
        x = new RS_Dijkstra < int , string > ( LARGENUM ) ;
      ostringstream sx ;

      // Creating nodes
      for ( int j = 0 ; j < 15 ; ++ j )
      {
        sx << j ;
        x -> setNode ( sx . str () ) ;
        sx . clear () ;
        sx . str ( "" ) ;
      }

      // Creating branches
      x -> setBranch ( "0" , "1" , 14 ) ;
      x -> setBranch ( "0" , "2" ,  9 ) ;
      x -> setBranch ( "0" , "3" , 31 ) ;
      x -> setBranch ( "1" , "10" , 16 ) ;
      x -> setBranch ( "1" , "13" , 20 ) ;
      x -> setBranch ( "2" , "3" , 20 ) ;
      x -> setBranch ( "2" , "4" ,  6 ) ;
      x -> setBranch ( "2" , "5" , 18 ) ;
      x -> setBranch ( "3" , "11" , 26 ) ;
      x -> setBranch ( "3" , "14" , 27 ) ;
      x -> setBranch ( "4" , "7" , 29 ) ;
      x -> setBranch ( "4" , "8" ,  1 ) ;
      x -> setBranch ( "5" , "6" ,  5 ) ;
      x -> setBranch ( "5" , "10" ,  2 ) ;
      x -> setBranch ( "6" , "7" , 15 ) ;
      x -> setBranch ( "6" , "11" , 10 ) ;
      x -> setBranch ( "6" , "12" ,  3 ) ;
      x -> setBranch ( "7" , "8" , 16 ) ;
      x -> setBranch ( "7" , "9" ,  1 ) ;
      x -> setBranch ( "8" , "9" , 30 ) ;
      x -> setBranch ( "8" , "10" ,  2 ) ;
      x -> setBranch ( "9" , "13" , 15 ) ;
      x -> setBranch ( "9" , "14" ,  3 ) ;
      x -> setBranch ( "10" , "11" , 14 ) ;
      x -> setBranch ( "11" , "12" ,  2 ) ;
      x -> setBranch ( "12" , "13" ,  7 ) ;
      x -> setBranch ( "13" , "14" , 26 ) ;

      x -> run ( ss . str () ) ;
      ss . clear () ;
      ss . str ( "" ) ;
      if ( i < 2 ) {
        int xxx = 0 ;
        cin >> xxx ;
      }
      delete x ;
    }
    int xxy = 0 ;
    cin >> xxy ;
  }
}
```

```
// -*- C++ -*-

// ----
// Dijkstra's algorithm
// ダイクストラ法
// ----

#ifndef RS_DIJKSTRA_H
#define RS_DIJKSTRA_H

#include <map>
#include <iostream>
#include <vector>
#include "RS_FHeap.h"

using std :: cerr ;
using std :: endl ;
using std :: vector ;
using std :: map ;
using std :: ostream ;
using std :: string ;



// ----
// Forward declarations
// ----
template < class KC , class NID >
class RS_DKBranch ;

template < class KC , class NID >
class RS_DKNode ;

template < class KD , class NIX >
std :: ostream & operator<<
( std :: ostream & ostr ,
  const RS_DKNode < KD , NIX > & dkn ) ;

// ----
// Dijkstra's Algorithm: the "node" of the graph
// ダイクストラのアルゴリズム: グラフの「ノード」
// ----
template < class KC , class NID >
class RS_DKNode
{

  template < class KD , class NIX >
  friend std :: ostream & operator<<
  ( std :: ostream & ostr , const RS_DKNode < KD , NIX > & dkn ) ;

private :

  string _name ;
  NID _nid ;
  typename RS_FHeap < KC , RS_DKNode < KC , NID > > :: Entry * _heapent ;
  RS_DKBranch < KC , NID > * _dkb_in ;
  vector < RS_DKBranch < KC , NID > * > _blist ;
  mutable typename vector < RS_DKBranch < KC , NID > * > :: size_type _iter ;
  mutable bool _iter_to_start ;
  bool _marked ;

  explicit RS_DKNode ( const RS_DKNode < KC , NID > & ) ;

public :
  explicit RS_DKNode ( const NID & _nid_in , const string & _n_in )
    : _nid ( _nid_in ) , _name ( _n_in ) , _heapent ( 0 ) , _dkb_in ( 0 ) ,
      _blist () , _iter ( 0 ) , _iter_to_start ( true ) , _marked ( false ) {}
  void setHeapEntry
  ( typename RS_FHeap < KC , RS_DKNode < KC , NID > > :: Entry * x )
  { _heapent = x ; }
  void setIncomingBranch ( RS_DKBranch < KC , NID > * x ) { _dkb_in = x ; }
  void setBranch ( RS_DKBranch < KC , NID > * x ) { _blist . push_back ( x ) ; }
  void setAsMarked () { _marked = true ; }

  typename RS_FHeap < KC , RS_DKNode < KC , NID > > :: Entry *
  getHeapEntry () const { return _heapent ; }
  RS_DKBranch < KC , NID > * getIncomingBranch () const { return _dkb_in ; }

  const NID & getNodeID () const { return _nid ; }
  const string & getName () const { return _name ; }
  const bool isMarked () const { return _marked ; }

  RS_DKBranch < KC , NID > * getNextBranch () const
  {
    while ( true )
    {
      if ( _iter_to_start )
      {
        _iter_to_start = false ;
      }
      else
      {
        ++ _iter ;
      }
      if ( _iter >= _blist . size () )
        return 0 ;
      if ( _blist [ _iter ] == _dkb_in )
      {
        continue ;
      }
      return _blist [ _iter ] ;
    }
  }
} ;


template < class KD , class NIX >
ostream &
operator<<
( ostream & ostr ,
  const RS_DKNode < KD , NIX > & dkn )
{
  ostr << dkn . getName () ;
  return ostr ;
}




// ----
// Declaration of template class RS_DKBranch
// RS_DKBranch テンプレートクラスの宣言
// ----
template < class KC , class NID >
class RS_DKBranch
{

private :
  KC _weight ;
  RS_DKNode < KC , NID > * _n_in ;
  RS_DKNode < KC , NID > * _n_out ;

public :

  explicit RS_DKBranch ( RS_DKNode < KC , NID > * _in ,
                         RS_DKNode < KC , NID > * _out , KC x )
    : _weight ( x ) , _n_in ( _in ) , _n_out ( _out )
  { _in -> setBranch ( this ) ; _out -> setBranch ( this ) ; }

  RS_DKNode < KC , NID > * getOpposite
  ( RS_DKNode < KC , NID > * x )
  { if ( _n_in == x ) return _n_out ; if ( _n_out == x ) return _n_in ;
    cerr << "Error: could not find opposite node" << endl ; exit ( 1 ) ; }

  const KC & getWeight () const { return _weight ; }

} ;



// ----
// Declaration of template class RS_Dijkstra
// RS_Dijkstra テンプレートクラスの宣言
// ----
template < class KC , class NID >
class RS_Dijkstra
{

  // "Large number" for initialising a node.
  KC _large ;

  // Heap
  RS_FHeap < KC , RS_DKNode < KC , NID > > _hp ;

  // Vector container of RS_DKNode object instances
  // *** May have to consider using map container instead of vector ***
  map < const NID , RS_DKNode < KC , NID > * > _nlist ;

  // Vector container of RS_DKBranch object instances
  // *** May have to consider using map container instead of vector ***
  vector < RS_DKBranch < KC , NID > * > _blist ;

public :

  explicit RS_Dijkstra ( const KC & _l_in )
    : _large ( _l_in ) , _hp () , _nlist () , _blist ()
  {
  }

  ~RS_Dijkstra ()
  {
    for ( typename vector < RS_DKBranch < KC , NID > * > :: iterator
            ii = _blist . begin () ; ii != _blist . end () ; ++ ii )
    {
      delete ( * ii ) ;
    }
    for ( typename map < const NID , RS_DKNode < KC , NID > * > :: iterator
            ii = _nlist . begin () ; ii != _nlist . end () ; ++ ii )
    {
      typename RS_FHeap < KC , RS_DKNode < KC , NID > > :: Entry *
        iix = ii -> second -> getHeapEntry () ;
      delete iix ;
      delete ii -> second ;
    }
  }

  void setNode ( const NID & _n_in )
  {
    RS_DKNode < KC , NID > *
      _n = new RS_DKNode < KC , NID > ( _n_in , _n_in ) ;
    _nlist [ _n_in ] = _n ;
    typename RS_FHeap < KC , RS_DKNode < KC , NID > > :: Entry * _e
      = new typename RS_FHeap < KC , RS_DKNode < KC , NID > > :: Entry
      ( _large , _n , & _hp ) ;
    _n -> setHeapEntry ( _e ) ;
    _hp . insert ( _e ) ;
  }

  RS_DKNode < KC , NID > * getNode
  ( const NID & _n_in )
  {
    typename map < const NID , RS_DKNode < KC , NID > * > :: iterator
      _i_x = _nlist . find ( _n_in ) ;
    if ( _i_x == _nlist . end () )
    {
      cerr << "Error: no node with ID ¥"" << _n_in
           << "¥" in class RS_Dijkstra" << endl ;
      exit ( 1 ) ;
    }
    return _i_x -> second ;
  }

  void setBranch
  ( const NID & _nn_f , const NID & _nn_t , const KC & _wt )
  {
    RS_DKNode < KC , NID > * _n_f = getNode ( _nn_f ) ;
    RS_DKNode < KC , NID > * _n_t = getNode ( _nn_t ) ;
    RS_DKBranch < KC , NID > *
      _b = new RS_DKBranch < KC , NID > ( _n_f , _n_t , _wt ) ;
    _blist . push_back ( _b ) ;
  }

  void run ( const NID & _start_nid )
  {
    cerr << "Dijkstra's Algorithm: starting from node ¥"" << _start_nid
         << "¥"" << endl ;
    RS_DKNode < KC , NID > * _n_st = getNode ( _start_nid ) ;
    _n_st -> getHeapEntry () -> decreaseKey ( 0 ) ;
    typename RS_FHeap < KC , RS_DKNode < KC , NID > > :: Entry * _min
      = _hp . deleteMinimum () ;
    _min -> getEntryData () -> setAsMarked () ;
    cerr << "delete min, node = " << _min -> getEntryData () -> getName ()
         << ", key = " << _min -> getKey () << endl ;

    while ( _hp . size () )
    {
      while ( RS_DKBranch < KC , NID > * _bx
              = _min -> getEntryData () -> getNextBranch () )
      {
        RS_DKNode < KC , NID > * _op
          = _bx -> getOpposite ( _min -> getEntryData () ) ;
        if ( _op -> isMarked () )
          continue ;
        KC _nkey = _min -> getKey () + _bx -> getWeight () ;
        cerr << "decrease key, node = " << _op -> getName () << ", key "
             << _op -> getHeapEntry () -> getKey () << " -> " << _nkey ;
        if ( _nkey < _op -> getHeapEntry () -> getKey () )
        {
          cerr << ", perform" << endl ;
          _op -> getHeapEntry () -> decreaseKey ( _nkey ) ;
          _op -> setIncomingBranch ( _bx ) ;
        }
        else
        {
          cerr << ", not perform" << endl ;
        }
      }
      _min = _hp . deleteMinimum () ;
      _min -> getEntryData () -> setAsMarked () ;
      cerr << "delete min, node = " << _min -> getEntryData () -> getName ()
           << ", key = " << _min -> getKey () << endl ;
    }
    for ( typename map < const NID , RS_DKNode < KC , NID > * >
            :: const_iterator ii = _nlist . begin () ;
          ii != _nlist . end () ; ++ ii )
    {
      RS_DKNode < KC , NID > * _n = ii -> second ;
      cerr << "Node ¥"" << _n -> getName () << "¥": "
          << _n -> getHeapEntry () -> getKey () << ": "
          << _n -> getName () ;
      while ( RS_DKBranch < KC , NID > * _bi = _n -> getIncomingBranch () )
      {
        cerr << "(" << _bi -> getWeight () << ")" ;
        _n = _bi -> getOpposite ( _n ) ;
        cerr << _n -> getName () ;
      }
      cerr << endl ;
    }
  }

} ;


#endif // RS_DIJKSTRA_H
```

```
// -*- C++ -*-

// ----
// Heap
// ヒープ
// ----

#ifndef RS_FHEAP_H
#define RS_FHEAP_H

#include <iostream>
#include <cstdlib>
#include <cmath>
#include <vector>
#include <string>

using std :: cerr ;
using std :: endl ;
using std :: vector ;
using std :: string ;

//#define DEBUG_FHEAP



template < class KC , class ENT >
class RS_FHeap
{
public :

  typedef unsigned int size_type ;

  class Entry
  {

    friend class RS_FHeap ;

  private :

    KC _key ;
    ENT * _ent ;
    typename RS_FHeap :: size_type _deg ;
    bool _marked ;
    typename RS_FHeap :: Entry * _right ;
    typename RS_FHeap :: Entry * _left ;
    typename RS_FHeap :: Entry * _parent ;
    typename RS_FHeap :: Entry * _child ;
    RS_FHeap * _myheap ;

    void cutFromChildList () ;
    void execCascadingCut () ;
    void addToChildList ( Entry * ) ;

    explicit Entry () ;
    explicit Entry ( const Entry & ) ;

  public :

    Entry ( const KC & x , ENT * _e , RS_FHeap * hp )
      : _key ( x ) , _ent ( _e ) , _deg ( 0 ) , _marked ( false ) ,
        _right ( this ) , _left ( this ) , _parent ( 0 ) , _child ( 0 ) ,
        _myheap ( hp ) {}

    void decreaseKey ( const KC & ) ;

    const KC & getKey () const { return _key ; }

    Entry * getRight () { return _right ; }
    Entry * getLeft () { return _left ; }
    Entry * getParent () { return _parent ; }
    Entry * getChild () { return _child ; }
    typename RS_FHeap :: size_type getDegree () const { return _deg ; }
    typename RS_FHeap :: size_type countEntriesInMe () const ;
#ifdef DEBUG_FHEAP
    void printChildren ( unsigned int = 0 ) const ;
#endif

    ENT * getEntryData () { return _ent ; }
    const ENT * getEntryData () const { return _ent ; }

  } ;

  friend class Entry ;

private :

  typename RS_FHeap :: Entry * _min ;
  size_type _num ;
  mutable size_type _d_cal ;
  mutable size_type _num_d_cal ;

  vector < typename RS_FHeap :: Entry * > cvec ;

  size_type getD () const ;

  void putIntoRoot ( RS_FHeap :: Entry * ) ;
  void execConsolidate () ;

public :

  explicit RS_FHeap ()
    : _min ( 0 ) , _num ( 0 ) , _d_cal ( 0 ) , _num_d_cal ( 0 ) , cvec () {}

  Entry * getMinimum () const { return _min ; }

  void insert ( RS_FHeap :: Entry * ) ;
  Entry * deleteMinimum () ;

  size_type size () const { return _num ; }

} ;


// ----
// Some implementation of class Heap
// ヒープクラスの実装の一部
// ----

template < class KC , class ENT >
void
RS_FHeap < KC , ENT > :: putIntoRoot
( typename RS_FHeap < KC , ENT > :: Entry * _new )
{
#ifdef DEBUG_FHEAP
  cerr << "putIntoRoot: start, entry = " << * ( _new -> getEntryData () )
       << "(" << * ( _new -> getLeft () -> getEntryData () ) << " - "
       << * ( _new -> getEntryData () ) << " - "
       << * ( _new -> getRight () -> getEntryData () ) << ")"<< endl ;
#endif
  if ( _min )
  {
#ifdef DEBUG_FHEAP
    cerr << "putIntoRoot: _min = " << * ( _min -> getEntryData () )
       << "(" << * ( _min -> getLeft () -> getEntryData () ) << " - "
       << * ( _min -> getEntryData () ) << " - "
       << * ( _min -> getRight () -> getEntryData () ) << ")"<< endl ;
```

```
#endif
    _min -> _left -> _right = _new ;
    _new -> _left = _min -> _left ;
    _new -> _right = _min ;
    _min -> _left = _new ;
    if ( _new -> _key < _min -> _key )
    {
      _min = _new ;
    }
  }
  else
  {
#ifdef DEBUG_FHEAP
    cerr << "putIntoRoot: _min not found, heap size = " << _num << endl ;
#endif
    _min = _new ;
  }
#ifdef DEBUG_FHEAP
  cerr << "putIntoRoot: finished, entry = " << * ( _new -> getEntryData () )
       << "(" << * ( _new -> getLeft () -> getEntryData () ) << " - "
       << * ( _new -> getEntryData () ) << " - "
       << * ( _new -> getRight () -> getEntryData () ) << ")"<< endl ;
#endif
}



template < class KC , class ENT >
typename RS_FHeap < KC , ENT > :: size_type
RS_FHeap < KC , ENT > :: Entry :: countEntriesInMe
()
  const
{
  typename RS_FHeap :: size_type _retv = 1 ;
  if ( _child )
  {
    Entry * _csc = _child ;
    do
    {
      _retv += _csc -> countEntriesInMe () ;
      _csc = _csc -> _right ;
    }
    while ( _csc != _child ) ;
  }
  return _retv ;
}



template < class KC , class ENT >
void
RS_FHeap < KC , ENT > :: Entry :: cutFromChildList
()
{
#ifdef DEBUG_FHEAP
  cerr << "cut: object = " << * ( getEntryData () ) << "<"
       << getDegree () << ">, parent = "
       << * ( _parent -> getEntryData () ) << "<"
       << _parent -> getDegree () << ">" << endl ;
#endif
  -- ( _parent -> _deg ) ;
  _right -> _left = _left ;
  _left -> _right = _right ;
  if ( _parent -> _child == this )
  {
#ifdef DEBUG_FHEAP
    cerr << "cut: parent's child pointer points to this" << endl ;
#endif
```

```
  if ( this -> _right == this )
  {
#ifdef DEBUG_FHEAP
   cerr << "cut: no child must be present, degree actually = "
        << _parent -> getDegree () << endl ;
#endif
   _parent -> _child = 0 ;
  }
  else
  {
   _parent -> _child = _left ;
  }
 }
 _left = _right = this ;
 _parent = 0 ;
 _marked = false ;
 _myheap -> putIntoRoot ( this ) ;
}



#ifdef DEBUG_FHEAP
template < class KC , class ENT >
void
RS_FHeap < KC , ENT > :: Entry :: printChildren
( unsigned int _indent /* = 0 */ )
 const
{
 if ( _child )
 {
  Entry * _csc = _child ;
  do
  {
   for ( unsigned int i = 0 ; i < _indent ; ++ i )
   {
    cerr << "  " ;
   }
   cerr << "+--" << * ( _csc -> getEntryData () ) << "<"
        << _csc -> getDegree () << ">[" << _csc -> getKey () << "]("
        << _csc -> countEntriesInMe () << ")" << endl ;
   _csc -> printChildren ( _indent + 1 ) ;
   _csc = _csc -> _right ;
  }
  while ( _csc != _child ) ;
 }
}
#endif



template < class KC , class ENT >
void
RS_FHeap < KC , ENT > :: Entry :: execCascadingCut
()
{
 if ( _parent )
 {
#ifdef DEBUG_FHEAP
  cerr << "cascading cut: parent: " << * ( _parent -> getEntryData () ) << "["
       << _parent -> getKey () << "]" ;
  if ( _parent -> _marked )
   cerr << "[M]" ;
  cerr << endl ;
#endif
  if ( _parent -> _marked )
  {
#ifdef DEBUG_FHEAP
   cerr << "parent has been marked, cut from child list" << endl ;
```

```
#endif
      typename RS_FHeap < KC , ENT > :: Entry * px = _parent ;
      cutFromChildList () ;
      px -> execCascadingCut () ;
    }
    else
    {
#ifdef DEBUG_FHEAP
      cerr << "mark parent: " << * ( _parent -> getEntryData () ) << "["
           << _parent -> getKey () << "]" ;
      if ( _parent -> _marked )
        cerr << "[M]" ;
#endif
      _parent -> _marked = true ;
#ifdef DEBUG_FHEAP
      cerr << " -> " << * ( _parent -> getEntryData () ) << "["
           << _parent -> getKey () << "]" ;
      if ( _parent -> _marked )
        cerr << "[M]" ;
      cerr << endl ;
#endif
    }
  }
  else
  {
#ifdef DEBUG_FHEAP
    cerr << "cascading cut: no parent. done." << endl ;
#endif
  }
}


template < class KC , class ENT >
void
RS_FHeap < KC , ENT > :: Entry :: decreaseKey
( const KC & x )
{
  if ( _key < x )
  {
    cerr << "Error: Heap::Entry::decreaseKey(...): new key must be smaller "
         << "than the original key" << endl ;
    exit ( 1 ) ;
  }
  _key = x ;
#ifdef DEBUG_FHEAP
  cerr << "decreaseKey: key set" << endl ;
#endif
  if ( _parent )
  {
#ifdef DEBUG_FHEAP
    cerr << "parent exists" << endl ;
#endif
    if ( _key < _parent -> getKey () )
    {
#ifdef DEBUG_FHEAP
      cerr << "parent has larger key" << endl ;
#endif
      typename RS_FHeap < KC , ENT > :: Entry * px = _parent ;
#ifdef DEBUG_FHEAP
      cerr << "before cut" << endl ;
#endif
      cutFromChildList () ;
#ifdef DEBUG_FHEAP
      cerr << "before cascading cut" << endl ;
#endif
      px -> execCascadingCut () ;
#ifdef DEBUG_FHEAP
      cerr << "cut complete" << endl ;
```

```
#endif
    }
  }
  else
  {
#ifdef DEBUG_FHEAP
    cerr << "parent does not exist" << endl ;
#endif
  }
  if ( _myheap -> _min -> _key > x )
  {
#ifdef DEBUG_FHEAP
    cerr << "this is the minimum entry" << endl ;
#endif
    _myheap -> _min = this ;
  }
}


template < class KC , class ENT >
typename RS_FHeap < KC , ENT > :: Entry *
RS_FHeap < KC , ENT > :: deleteMinimum
()
{
  typename RS_FHeap < KC , ENT > :: Entry * ret_val = _min ;
#ifdef DEBUG_FHEAP
  cerr << "deleteMin starts, current heap size: " << _num << " elements"
       << endl ;
#endif
  while ( _min -> _child )
  {
#ifdef DEBUG_FHEAP
    cerr << "deleteMin: child found, _min = " << * ( _min -> getEntryData () )
         << endl ;
    typename RS_FHeap < KC , ENT > :: Entry * _mc
      = _min -> _child ;
    typename RS_FHeap < KC , ENT > :: Entry * _mp = _mc -> _parent ;
    cerr << "child: " << _mc -> getKey () << " [" << _mc -> getDegree ()
         << "]: " << * ( _mc -> getEntryData () ) << " ("
         << * ( _mc -> getLeft () -> getEntryData () )
         << " - " << * ( _mc -> getEntryData () ) << " - "
         << * ( _mc -> getRight () -> getEntryData () ) << ")" << endl ;
    cerr << "child's parent: " << _mp -> getKey () << " ["
         << _mp -> getDegree () << "]: " << * ( _mp -> getEntryData () )
         << " (" << * ( _mp -> getLeft () -> getEntryData () )
         << " - " << * ( _mp -> getEntryData () ) << " - "
         << * ( _mp -> getRight () -> getEntryData () ) << ")" << endl ;
#endif
    _min -> _child -> cutFromChildList () ;
  }
#ifdef DEBUG_FHEAP
  cerr << "deleteMin: no child list, _min = "
       << * ( _min -> getEntryData () ) << endl ;
#endif
  if ( ret_val -> _right == ret_val )
  {
#ifdef DEBUG_FHEAP
    cerr << "deleteMin: right of ret_val is ret_val: _num should be 1, "
         << "actually: " << _num << endl ;
#endif
    _min = 0 ;
  }
  else
  {
#ifdef DEBUG_FHEAP
    cerr << "deleteMin: set _min to _right of ret_val, "
         << * ( ret_val -> _right -> getEntryData () ) << endl ;
#endif
```

```
    _min = ret_val -> _right ;
    ret_val -> _right -> _left = ret_val -> _left ;
    ret_val -> _left -> _right = ret_val -> _right ;
    execConsolidate () ;
  }
  -- _num ;
#ifdef DEBUG_FHEAP
  if ( _min )
  {
    cerr << "deleteMin: root list" << endl ;
    typename RS_FHeap < KC , ENT > :: Entry * _nc = _min ;
    typename RS_FHeap < KC , ENT > :: size_type _cx = 0 ;
    do
    {
      cerr << * ( _nc -> getEntryData () ) << "<" << _nc -> getDegree ()
           << ">[" << _nc -> getKey () << "](" << _nc -> countEntriesInMe ()
           << ")" << endl ;
      _nc -> printChildren () ;
      _nc = _nc -> _right ;
      _cx += _nc -> countEntriesInMe () ;
    }
    while ( _nc != _min ) ;
    cerr << "_num, _num actually: " << _num << ", " << _cx << endl ;
  }
  else
  {
    cerr << "deleteMin: root list empty" << endl
         << "_num, _num actually: " << _num << ", 0" << endl ;
  }
#endif
  return ret_val ;
}


template < class KC , class ENT >
typename RS_FHeap < KC , ENT > :: size_type
RS_FHeap < KC , ENT > :: getD
()
  const
{
  if ( _num_d_cal == _num )
  {
    return _d_cal ;
  }
  _num_d_cal = _num ;
  double _dn = log ( _num ) / log ( ( 1 + sqrt ( 5 ) ) / 2 ) ;
  if ( _dn < 0 )
    _dn = 0 ;
  return ( _d_cal = static_cast < size_type > ( ceil ( _dn ) ) ) ;
}


template < class KC , class ENT >
void
RS_FHeap < KC , ENT > :: execConsolidate
()
{
#ifdef DEBUG_FHEAP
  cerr << "consolidation starts, getD: " << getD () << endl ;
#endif
  cvec . assign ( getD () , 0 ) ;
  typename RS_FHeap < KC , ENT > :: Entry * _nw = _min ;
  typename RS_FHeap < KC , ENT > :: Entry * _minx = _min ;
  do
  {
    _min = _minx ;
#ifdef DEBUG_FHEAP
    cerr << "check: " << _nw -> getKey () << " [" << _nw -> getDegree ()
```

```
         << "]: " << * ( _nw -> getEntryData () ) << " ("
         << * ( _nw -> getLeft () -> getEntryData () )
         << " - " << * ( _nw -> getEntryData () ) << " - "
         << * ( _nw -> getRight () -> getEntryData () ) << ")" << endl ;
#endif
    typename RS_FHeap < KC , ENT > :: Entry * _nx = _nw ;
    _nw = _nw -> _right ;
    while ( _nx -> _deg >= cvec . size () )
    {
      cerr << "Warning: degree vector not sufficient, deg, size ="
           << _nx -> _deg << ", " << cvec . size () << endl ;
      cvec . push_back ( 0 ) ;
    }
    size_type _d = _nx -> _deg ;
#ifdef DEBUG_FHEAP
    cerr << "_min degree: " << _d << endl ;
#endif
    if ( cvec [ _d ] == _nx )
    {
      cerr << "Warning: this degree checked, continue" << endl ;
      continue ;
    }
    while ( cvec [ _d ] )
    {
      typename RS_FHeap < KC , ENT > :: Entry * _ny = cvec [ _d ] ;
#ifdef DEBUG_FHEAP
      cerr << "degree " << _d << " already there, " ;
      cerr << _ny -> getKey () << " [" << _ny -> getDegree ()
           << "]: " << * ( _ny -> getEntryData () ) << " ("
           << * ( _ny -> getLeft () -> getEntryData () )
           << " - " << * ( _ny -> getEntryData () ) << " - "
           << * ( _ny -> getRight () -> getEntryData () ) << ")" << endl ;
#endif
      if ( _ny -> _key < _nx -> _key )
      {
        typename RS_FHeap < KC , ENT > :: Entry * _nz = _nx ;
        _nx = _ny ;
        _ny = _nz ;
#ifdef DEBUG_FHEAP
        cerr << "exchanged nx and ny, ny = " ;
        cerr << _ny -> getKey () << " [" << _ny -> getDegree ()
             << "]: " << * ( _ny -> getEntryData () ) << " ("
             << * ( _ny -> getLeft () -> getEntryData () )
             << " - " << * ( _ny -> getEntryData () ) << " - "
             << * ( _ny -> getRight () -> getEntryData () ) << ")" << endl ;
#endif
      }
      if ( _ny == _minx )
      {
        _minx = _minx -> _right ;
      }
#ifdef DEBUG_FHEAP
      cerr << "before addtochildlist, _nx: " << _nx -> getKey ()
           << " [" << _nx -> getDegree ()
           << "]: " << * ( _nx -> getEntryData () ) << " ("
           << * ( _nx -> getLeft () -> getEntryData () )
           << "<" << _nx -> getLeft () -> getDegree () << ">"
           << " - " << * ( _nx -> getEntryData () ) << " - "
           << * ( _nx -> getRight () -> getEntryData () )
           << "<" << _nx -> getRight () -> getDegree () << ">"
           << ")" ;
      if ( _nx -> _child )
      {
        cerr << " (child: " ;
        string x_str = "" ;
        typename RS_FHeap < KC , ENT > :: Entry * _nc = _nx -> _child ;
        do
        {
          cerr << x_str << * ( _nc -> getEntryData () )
               << "<" << _nc -> getDegree () << ">" ;
          x_str = " - " ;
          _nc = _nc -> _right ;
        }
        while ( _nc != _nx -> _child ) ;
        cerr << ")" ;
      }
      cerr << endl ;
#endif
      _ny -> _right -> _left = _ny -> _left ;
      _ny -> _left -> _right = _ny -> _right ;
      _ny -> _right = _ny -> _left = _ny ;
      _nx -> addToChildList ( _ny ) ;
#ifdef DEBUG_FHEAP
      cerr << "after addtochildlist, _nx: " << _nx -> getKey ()
           << " [" << _nx -> getDegree ()
           << "]: " << * ( _nx -> getEntryData () ) << " ("
           << * ( _nx -> getLeft () -> getEntryData () )
           << "<" << _nx -> getLeft () -> getDegree () << ">"
           << " - " << * ( _nx -> getEntryData () ) << " - "
           << * ( _nx -> getRight () -> getEntryData () )
           << "<" << _nx -> getRight () -> getDegree () << ">"
           << ")" ;
      if ( _nx -> _child )
      {
        cerr << " (child: " ;
        string x_str = "" ;
        typename RS_FHeap < KC , ENT > :: Entry * _nc = _nx -> _child ;
        do
        {
          cerr << x_str << * ( _nc -> getEntryData () )
               << "<" << _nc -> getDegree () << ">" ;
          x_str = " - " ;
          _nc = _nc -> _right ;
        }
        while ( _nc != _nx -> _child ) ;
        cerr << ")" ;
      }
      cerr << endl ;
#endif
      cvec [ _d ] = 0 ;
      ++ _d ;
    }
#ifdef DEBUG_FHEAP
    cerr << "degree " << _d << ": this one" << endl ;
#endif
    cvec [ _d ] = _nx ;
  }
  while ( _nw != _min ) ;
  _min = 0 ;
#ifdef DEBUG_FHEAP
  cerr << "before search, _min set to Null" << endl ;
#endif
  for ( vector < size_type > :: size_type i = 0 ; i < cvec . size () ; ++ i )
  {
#ifdef DEBUG_FHEAP
    cerr << "degree " << i << " scanning" << endl ;
#endif
    if ( cvec [ i ] )
    {
#ifdef DEBUG_FHEAP
      cerr << "degree " << i << " found: "
           << * ( cvec [ i ] -> getEntryData () ) << "["
           << cvec [ i ] -> getKey () << "]" << endl ;
#endif
      if ( ! _min || cvec [ i ] -> _key < _min -> _key )
      {
```

```
#ifdef DEBUG_FHEAP
        cerr << " ... this is the new minimum" << endl ;
#endif
        _min = cvec [ i ] ;
      }
    }
  }
#ifdef DEBUG_FHEAP
  cerr << "end: " << _min -> getKey () << " [" << _min -> getDegree ()
       << "]: " << * ( _min -> getEntryData () ) << " ("
       << * ( _min -> getLeft () -> getEntryData () )
       << " - " << * ( _min -> getEntryData () ) << " - "
       << * ( _min -> getRight () -> getEntryData () ) << ")" << endl ;
  cerr << "consolidation ended" << endl ;
#endif
}



template < class KC , class ENT >
void
RS_FHeap < KC , ENT > :: insert
( RS_FHeap < KC , ENT > :: Entry * _in )
{
  putIntoRoot ( _in ) ;
  ++ _num ;
}



template < class KC , class ENT >
void
RS_FHeap < KC , ENT > :: Entry :: addToChildList
( typename RS_FHeap < KC , ENT > :: Entry * _in )
{
  if ( _child )
  {
    _child -> _left -> _right = _in ;
    _in -> _left = _child -> _left ;
    _child -> _left = _in ;
    _in -> _right = _child ;
  }
  else
  {
    _child = _in ;
  }
  _in -> _parent = this ;
  ++ _deg ;
}




#endif // RS_FHEAP_H
```