PROJECT 1 SOLUTION

```
==========

// globals.h

#ifndef GLOBALS_INCLUDED
#define GLOBALS_INCLUDED

////////////////////////////////////////////////////////////////////////
// Global constants
////////////////////////////////////////////////////////////////////////

const int MAXROWS = 20;             // max number of rows in the pit
const int MAXCOLS = 40;             // max number of columns in the pit
const int MAXSNAKES = 190;          // max number of snakes allowed

const int UP    = 0;
const int DOWN  = 1;
const int LEFT  = 2;
const int RIGHT = 3;

////////////////////////////////////////////////////////////////////////
//  Utility function declarations
////////////////////////////////////////////////////////////////////////

int decodeDirection(char dir);
bool directionToDeltas(int dir, int& rowDelta, int& colDelta);
void clearScreen();

#endif // GLOBALS_INCLUDED

// utilities.cpp

#include "globals.h"

int decodeDirection(char dir)
{
    switch (dir)
    {
      case 'u':  return UP;
      case 'd':  return DOWN;
      case 'l':  return LEFT;
      case 'r':  return RIGHT;
    }
    return -1;  // bad argument passed in!
}
```

```
bool directionToDeltas(int dir, int& rowDelta, int& colDelta)
{
    switch (dir)
    {
      case UP:     rowDelta = -1; colDelta =  0; break;
      case DOWN:   rowDelta =  1; colDelta =  0; break;
      case LEFT:   rowDelta =  0; colDelta = -1; break;
      case RIGHT:  rowDelta =  0; colDelta =  1; break;
      default:     return false;
    }
    return true;
}

//////////////////////////////////////////////////////////////////////////
//  clearScreen implementations
//////////////////////////////////////////////////////////////////////////

// DO NOT MODIFY OR MAKE DELETIONS TO THE TEXT BETWEEN HERE AND THE END OF
// THE FILE!!!  THE CODE IS SUITABLE FOR VISUAL C++, XCODE, AND g++ UNDER
// LINUX.

// IF YOU ARE GOING TO MOVE THE IMPLEMENTATION OF clearScreen TO ANOTHER
// FILE, MOVE ALL THE TEXT FROM THE clearScreen implementations COMMENT
// BLOCK ABOVE THROUGH THE END OF THIS FILE TO the NEW LOCATION.

// Note to Xcode users:  clearScreen() will just write a newline instead
// of clearing the window if you launch your program from within Xcode.
// That's acceptable.

#ifdef _MSC_VER  //  Microsoft Visual C++

#pragma warning(disable : 4005)
#include <windows.h>

void clearScreen()
{
    HANDLE hConsole = GetStdHandle(STD_OUTPUT_HANDLE);
    CONSOLE_SCREEN_BUFFER_INFO csbi;
    GetConsoleScreenBufferInfo(hConsole, &csbi);
    DWORD dwConSize = csbi.dwSize.X * csbi.dwSize.Y;
    COORD upperLeft = { 0, 0 };
    DWORD dwCharsWritten;
    FillConsoleOutputCharacter(hConsole, TCHAR(' '), dwConSize, upperLeft,
                                                        &dwCharsWritten);
    SetConsoleCursorPosition(hConsole, upperLeft);
}

#else  // not Microsoft Visual C++, so assume UNIX interface
```

```
#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;

void clearScreen()  // will just write a newline in an Xcode output window
{
    static const char* term = getenv("TERM");
    if (term == nullptr  ||  strcmp(term, "dumb") == 0)
        cout << endl;
    else
    {
        static const char* ESC_SEQ = "\x1B[";  // ANSI Terminal esc seq:  ESC [
        cout << ESC_SEQ << "2J" << ESC_SEQ << "H" << flush;
    }
}

#endif


// Game.h

#ifndef GAME_INCLUDED
#define GAME_INCLUDED

class Pit;

class Game
{
  public:
        // Constructor/destructor
    Game(int rows, int cols, int nSnakes);
    ~Game();

        // Mutators
    void play();

  private:
    Pit* m_pit;
};

#endif // GAME_INCLUDED

// Game.cpp

#include "Game.h"
#include "Pit.h"
```

```
#include "Player.h"
#include "History.h"
#include "globals.h"
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;

Game::Game(int rows, int cols, int nSnakes)
{
    if (nSnakes < 0)
    {
        cout << "***** Cannot create Game with negative number of snakes!" << endl;
        exit(1);
    }
    if (nSnakes > MAXSNAKES)
    {
        cout << "***** Cannot create Game with " << nSnakes
             << " snakes; only " << MAXSNAKES << " are allowed!" << endl;
        exit(1);
    }
    if (rows == 1  &&  cols == 1  &&  nSnakes > 0)
    {
        cout << "***** Cannot create Game with nowhere to place the snakes!" <<
endl;
        exit(1);
    }

        // Create pit
    m_pit = new Pit(rows, cols);

        // Add player
    int rPlayer = 1 + rand() % rows;
    int cPlayer = 1 + rand() % cols;
    m_pit->addPlayer(rPlayer, cPlayer);

      // Populate with snakes
    while (nSnakes > 0)
    {
        int r = 1 + rand() % rows;
        int c = 1 + rand() % cols;
          // Do not put a snake where the player is
        if (r == rPlayer  &&  c == cPlayer)
            continue;
        m_pit->addSnake(r, c);
        nSnakes--;
    }
}
```

```
Game::~Game()
{
    delete m_pit;
}

void Game::play()
{
    Player* p = m_pit->player();
    if (p == nullptr)
    {
        m_pit->display("");
        return;
    }
    string msg = "";
    while ( ! m_pit->player()->isDead()  &&  m_pit->snakeCount() > 0)
    {
        m_pit->display(msg);
        msg = "";
        cout << endl;
        cout << "Move (u/d/l/r//h/q): ";
        string action;
        getline(cin,action);
        if (action.size() == 0)
            p->stand();
        else
        {
            switch (action[0])
            {
              default:   // if bad move, nobody moves
                cout << '\a' << endl;  // beep
                continue;
              case 'q':
                return;
              case 'u':
              case 'd':
              case 'l':
              case 'r':
                p->move(decodeDirection(action[0]));
                break;
              case 'h':
                m_pit->history().display();
                cout << "Press enter to continue.";
                cin.ignore(10000, '\n');
                m_pit->display("");
                continue;  // doesn't count as a turn, so don't move snakes
            }
        }
```

```
        m_pit->moveSnakes();
    }
    m_pit->display(msg);
}

// History.h

#ifndef HISTORY_INCLUDED
#define HISTORY_INCLUDED

#include "globals.h"

class History
{
  public:
    History(int nRows, int nCols);

        // Accessors
    void display() const;

        // Mutators
    bool record(int r, int c);

  private:
    int  m_grid[MAXROWS][MAXCOLS];
    int  m_rows;
    int  m_cols;
};

#endif // HISTORY_INCLUDED

// History.cpp

#include "History.h"
#include "globals.h"
#include <iostream>
#include <cstdlib>
using namespace std;

History::History(int nRows, int nCols)
 : m_rows(nRows), m_cols(nCols)
{
    if (nRows <= 0  ||  nCols <= 0  ||  nRows > MAXROWS  ||  nCols > MAXCOLS)
    {
        cout << "***** History created with invalid size " << nRows << " by "
             << nCols << "!" << endl;
        exit(1);
    }
```

```
    for (int r = 0; r < m_rows; r++)
       for (int c = 0; c < m_cols; c++)
            m_grid[r][c] = 0;
}

void History::display() const
{
    clearScreen();

      // Draw the grid
    for (int r = 0; r < m_rows; r++)
    {
        for (int c = 0; c < m_cols; c++)
        {
            char ch = '.';
            int n = m_grid[r][c];
            if (n >= 26)
                ch = 'Z';
            else if (n > 0)
                ch = 'A' + n-1;
            cout << ch;
        }
        cout << endl;
    }
    cout << endl;
}

bool History::record(int r, int c)
{
    if (r <= 0  ||  c <= 0  ||  r > m_rows  ||  c > m_cols)
        return false;
    m_grid[r-1][c-1]++;
    return true;
}

// Pit.h

#ifndef PIT_INCLUDED
#define PIT_INCLUDED

#include "globals.h"
#include "History.h"
#include <string>

class Snake;
class Player;

class Pit
```

```
{
  public:
        // Constructor/destructor
    Pit(int nRows, int nCols);
    ~Pit();

        // Accessors
    int      rows() const;
    int      cols() const;
    Player*  player() const;
    int      snakeCount() const;
    int      numberOfSnakesAt(int r, int c) const;
    void     display(std::string msg) const;
    History& history();

        // Mutators
    bool   addSnake(int r, int c);
    bool   addPlayer(int r, int c);
    bool   destroyOneSnake(int r, int c);
    bool   moveSnakes();

  private:
    int     m_rows;
    int     m_cols;
    Player* m_player;
    Snake*  m_snakes[MAXSNAKES];
    int     m_nSnakes;
    History m_history;
};

#endif // PIT_INCLUDED

// Pit.cpp

#include "Pit.h"
#include "Player.h"
#include "Snake.h"
#include "History.h"
#include "globals.h"
#include <iostream>
#include <string>
#include <cstdlib>
using namespace std;

Pit::Pit(int nRows, int nCols)
 : m_rows(nRows), m_cols(nCols), m_player(nullptr), m_nSnakes(0),
   m_history(nRows, nCols)
{
```

```
    if (nRows <= 0  ||  nCols <= 0  ||  nRows > MAXROWS  ||  nCols > MAXCOLS)
    {
        cout << "***** Pit created with invalid size " << nRows << " by "
             << nCols << "!" << endl;
        exit(1);
    }
}

Pit::~Pit()
{
    for (int k = 0; k < m_nSnakes; k++)
        delete m_snakes[k];
    delete m_player;
}

int Pit::rows() const
{
    return m_rows;
}

int Pit::cols() const
{
    return m_cols;
}

Player* Pit::player() const
{
    return m_player;
}

int Pit::snakeCount() const
{
    return m_nSnakes;
}

int Pit::numberOfSnakesAt(int r, int c) const
{
    int count = 0;
    for (int k = 0; k < m_nSnakes; k++)
    {
        const Snake* sp = m_snakes[k];
        if (sp->row() == r  &&  sp->col() == c)
            count++;
    }
    return count;
}

void Pit::display(string msg) const
```

```
{
      // Position (row,col) in the pit coordinate system is represented in
      // the array element grid[row-1][col-1]
    char grid[MAXROWS][MAXCOLS];
    int r, c;

        // Fill the grid with dots
    for (r = 0; r < rows(); r++)
        for (c = 0; c < cols(); c++)
            grid[r][c] = '.';

        // Indicate each snake's position
    for (int k = 0; k < m_nSnakes; k++)
    {
        const Snake* sp = m_snakes[k];
        char& gridChar = grid[sp->row()-1][sp->col()-1];
        switch (gridChar)
        {
          case '.':  gridChar = 'S'; break;
          case 'S':  gridChar = '2'; break;
          case '9':  break;
          default:   gridChar++; break;  // '2' through '8'
        }
    }

        // Indicate player's position
    if (m_player != nullptr)
    {
        char& gridChar = grid[m_player->row()-1][m_player->col()-1];
        if (m_player->isDead())
            gridChar = '*';
        else
            gridChar = '@';
    }

        // Draw the grid
    clearScreen();
    for (r = 0; r < rows(); r++)
    {
        for (c = 0; c < cols(); c++)
            cout << grid[r][c];
        cout << endl;
    }
    cout << endl;

        // Write message, snake, and player info
    cout << endl;
    if (msg != "")
```

```
        cout << msg << endl;
    cout << "There are " << snakeCount() << " snakes remaining." << endl;
    if (m_player == nullptr)
        cout << "There is no player." << endl;
    else
    {
        if (m_player->age() > 0)
            cout << "The player has lasted " << m_player->age() << " steps." <<
endl;
        if (m_player->isDead())
            cout << "The player is dead." << endl;
    }
}

History& Pit::history()
{
    return m_history;
}

bool Pit::addSnake(int r, int c)
{
      // Dynamically allocate a new Snake and add it to the pit
    if (m_nSnakes == MAXSNAKES)
        return false;
    m_snakes[m_nSnakes] = new Snake(this, r, c);
    m_nSnakes++;
    return true;
}

bool Pit::addPlayer(int r, int c)
{
      // Do not add a player if one already exists
    if (m_player != nullptr)
        return false;

      // Dynamically allocate a new Player and add it to the pit
    m_player = new Player(this, r, c);
    return true;
}

bool Pit::destroyOneSnake(int r, int c)
{
    for (int k = 0; k < m_nSnakes; k++)
    {
        if (m_snakes[k]->row() == r  &&  m_snakes[k]->col() == c)
        {
            delete m_snakes[k];
            m_snakes[k] = m_snakes[m_nSnakes-1];
```

```
            m_nSnakes--;
            return true;
        }
    }
    return false;
}

bool Pit::moveSnakes()
{
    for (int k = 0; k < m_nSnakes; k++)
    {
        Snake* sp = m_snakes[k];
        sp->move();
        if (sp->row() == m_player->row()  &&  sp->col() == m_player->col())
            m_player->setDead();
    }

      // return true if the player is still alive, false otherwise
    return ! m_player->isDead();
}

// Player.h

#ifndef PLAYER_INCLUDED
#define PLAYER_INCLUDED

class Pit;

class Player
{
  public:
        // Constructor
    Player(Pit *pp, int r, int c);

        // Accessors
    int  row() const;
    int  col() const;
    int  age() const;
    bool isDead() const;

        // Mutators
    void   stand();
    void   move(int dir);
    void   setDead();

  private:
    Pit*  m_pit;
    int   m_row;
```

```
    int   m_col;
    int   m_age;
    bool  m_dead;
};

#endif // PLAYER_INCLUDED
// Player.cpp

#include "Player.h"
#include "Pit.h"
#include "History.h"
#include "globals.h"
#include <iostream>
#include <cstdlib>
using namespace std;

Player::Player(Pit* pp, int r, int c)
{
    if (pp == nullptr)
    {
        cout << "***** The player must be in some Pit!" << endl;
        exit(1);
    }
    if (r < 1  ||  r > pp->rows()  ||  c < 1  ||  c > pp->cols())
    {
        cout << "**** Player created with invalid coordinates (" << r
             << "," << c << ")!" << endl;
        exit(1);
    }
    m_pit = pp;
    m_row = r;
    m_col = c;
    m_age = 0;
    m_dead = false;
}

int Player::row() const
{
    return m_row;
}

int Player::col() const
{
    return m_col;
}

int Player::age() const
{
```

```
    return m_age;
}

void Player::stand()
{
    m_age++;
}

void Player::move(int dir)
{
    m_age++;
    int maxCanMove = 0;  // maximum distance player can move in direction dir
    switch (dir)
    {
      case UP:     maxCanMove = m_row - 1;             break;
      case DOWN:   maxCanMove = m_pit->rows() - m_row; break;
      case LEFT:   maxCanMove = m_col - 1;             break;
      case RIGHT:  maxCanMove = m_pit->cols() - m_col; break;
    }
    if (maxCanMove == 0)  // against wall
        return;
    int rowDelta;
    int colDelta;
    if ( ! directionToDeltas(dir, rowDelta, colDelta))
        return;

      // No adjacent snake in direction of movement

    if (m_pit->numberOfSnakesAt(m_row + rowDelta, m_col + colDelta) == 0)
    {
        m_row += rowDelta;
        m_col += colDelta;
        m_pit->history().record(m_row, m_col);
        return;
    }

      // Adjacent snake in direction of movement, so jump

    if (maxCanMove >= 2)  // need a place to land
    {
        m_pit->destroyOneSnake(m_row + rowDelta, m_col + colDelta);
        m_row += 2 * rowDelta;
        m_col += 2 * colDelta;
        m_pit->history().record(m_row, m_col);
        if (m_pit->numberOfSnakesAt(m_row, m_col) > 0)  // landed on a snake!
            setDead();
    }
}
```

```
bool Player::isDead() const
{
    return m_dead;
}

void Player::setDead()
{
    m_dead = true;
}

// Snake.h

#ifndef SNAKE_INCLUDED
#define SNAKE_INCLUDED

class Pit;

class Snake
{
  public:
        // Constructor
    Snake(Pit* pp, int r, int c);

        // Accessors
    int  row() const;
    int  col() const;

        // Mutators
    void move();

  private:
    Pit* m_pit;
    int  m_row;
    int  m_col;
};

#endif // SNAKE_INCLUDED

// Snake.cpp

#include "Snake.h"
#include "Pit.h"
#include "globals.h"
#include <iostream>
#include <cstdlib>
using namespace std;
```

```
Snake::Snake(Pit* pp, int r, int c)
{
    if (pp == nullptr)
    {
        cout << "***** A snake must be in some Pit!" << endl;
        exit(1);
    }
    if (r < 1  ||  r > pp->rows()  ||  c < 1  ||  c > pp->cols())
    {
        cout << "***** Snake created with invalid coordinates (" << r << ","
             << c << ")!" << endl;
        exit(1);
    }
    m_pit = pp;
    m_row = r;
    m_col = c;
}

int Snake::row() const
{
    return m_row;
}

int Snake::col() const
{
    return m_col;
}

void Snake::move()
{
      // Attempt to move in a random direction; if we can't move, don't move
    switch (rand() % 4)
    {
      case UP:     if (m_row > 1)             m_row--; break;
      case DOWN:   if (m_row < m_pit->rows()) m_row++; break;
      case LEFT:   if (m_col > 1)             m_col--; break;
      case RIGHT:  if (m_col < m_pit->cols()) m_col++; break;
    }
}

// main.cpp

#include "Game.h"
#include <cstdlib>
#include <ctime>
using namespace std;

int main()
```

```
{
      // Initialize the random number generator.  (You don't need to
      // understand how this works.)
    srand(static_cast<unsigned int>(time(0)));

      // Create a game
      // Use this instead to create a mini-game:   Game g(3, 3, 2);
    Game g(9, 10, 40);

      // Play the game
    g.play();
}
```

```
================================================================================
=====
HOMEWORK 1 SOLUTION
================================================================================
========
```

Summer 2016 CS 32
Homework 1 Solution

Problems 1, 2, and 3
Problem 4
Problem 5

Problems 1, 2, and 3:

In this solution, the functions with small, fast implementations are inlined. Alternatively, the inline keyword can be removed and the function definitions moved to Multiset.cpp. (inline will be mentioned at some point in class, so don't worry if you've never seen it before.)

Notice which member functions are const, and observe the use of the typedef name ItemType.

```
// Multiset.h

#ifndef MULTISET_INCLUDED
#define MULTISET_INCLUDED

  // Later in the course, we'll see that templates provide a much nicer
  // way of enabling us to have Multisets of different types.  For now, we'll
  // use a typedef.

typedef unsigned long ItemType;

const int DEFAULT_MAX_ITEMS = 200;
```

```
class Multiset
{
  public:
    Multiset();               // Create an empty multiset.
    bool empty() const;  // Return true if the multiset is empty, otherwise false.

    int size() const;
      // Return the number of items in the multiset.  For example, the size
      // of a multiset containing "cumin", "cumin", "cumin", "turmeric" is 4.

    int uniqueSize() const;
      // Return the number of distinct items in the multiset.  For example,
      // the uniqueSize of a multiset containing "cumin", "cumin", "cumin",
      // "turmeric" is 2.

    bool insert(const ItemType& value);
      // Insert value into the multiset.  Return true if the value was
      // actually inserted.  Return false if the value was not inserted
      // (perhaps because the multiset has a fixed capacity and is full).

    int erase(const ItemType& value);
      // Remove one instance of value from the multiset if present.
      // Return the number of instances removed, which will be 1 or 0.

    int eraseAll(const ItemType& value);
      // Remove all instances of value from the multiset if present.
      // Return the number of instances removed.

    bool contains(const ItemType& value) const;
      // Return true if the value is in the multiset, otherwise false.

    int count(const ItemType& value) const;
      // Return the number of instances of value in the multiset.

    int get(int i, ItemType& value) const;
      // If 0 <= i < uniqueSize(), copy into value an item in the multiset
      // and return the number of instances of that item in the multiset.
      // Otherwise, leave value unchanged and return 0.

    bool geLeastFrequentValue(ItemType& value) const;
      // If there exists a single item that has the least number of instances in the
multiset,
      // then copy into value that item in the multiset and return true.
      // However, if there exist more than 1 item that have the least number of
instances in the multiset,
      // then do not copy into value any item in the multiset and return false. In
other words, value should remain unchanged.
```

```
      // If there's no item in the multiset, return false.

    bool getSmallestValue(ItemType& value) const;
      // If there exists a value that is the smallest value among all the values in
the multiset,
      // then copy into value that item in the multiset and return true
      // Otherwise, return false.
      // For both unsigned long and string data type, the lower value can be found
by using less than operator (<).
      // For example, 10 is smaller than 20, so 10 < 20 is true.
      // "ABC" is smaller than "XYZ", so "ABC" < "XYZ" is true.

    bool getSecondSmallestValue(ItemType& value) const;
      // Similar to getSmallestValue(), but this time you need to find the second
smallest value.
      // If there exists a value that is the 2nd smallest value among all the values
in the multiset,
      // then copy into value that item in the multiset and return true.
      // Otherwise, return false.
      // Please note that you cannot use any sorting algorithm to sort the multiset.

    bool replace(ItemType original, ItemType new_value);
      // Replace the item that has the value equal to original by the new value
      // new_value.  For example, replace("ABC","XYZ") will search the multiset
      // for the item "ABC" and replace all occurrences of "ABC" as "XYZ".
      // If the replacement is successful, then return true.  If there is no
      // item to be replaced, then return false.

    int countIf(char op, ItemType value) const;
      // Count the number of items that the item is greater than, less than, or
      // equal to value.  For example:  countIf('>',100) returns the number of
      // items in multiset in which the item is greater than 100.

    void swap(Multiset& other);
      // Exchange the contents of this multiset with the other one.

    void copyIntoOtherMultiset(Multiset& other) const;
      // Insert all the items into the multiset in other.

  private:

      // Since this structure is used only by the implementation of the
      // Multiset class, we'll make it private to Multiset.

    struct Node
    {
        ItemType m_value;
        int      m_count;
```

```
    };
    Node m_data[DEFAULT_MAX_ITEMS];  // the items in the multiset, with counts
    int  m_uniqueSize;               // how many distinct items in the multiset
    int  m_size;                     // total number of items in the multiset

      // At any time, the elements of m_data indexed from 0 to m_uniqueSize-1
      // are in use.  m_size is the sum of the m_counts of those elements.

    int find(const ItemType& value) const;
      // Return index of the node in m_data whose m_value == value if there is
      // one, else -1

    bool insertMany(const ItemType& value, int nToInsert, int& uniqueSize);
      // If no room to insert nToInsert items, return false.  Otherwise,
      // insert value with (additional) count nToInsert using (and possibly
      // updating) uniqueSize as the number of distinct existing items, and
      // return true.

    int doErase(const ItemType& value, bool all);
      // Remove one or all instances of value from the multiset if present,
      // depending on the second parameter.  Return the number of instances
      // removed.
};

// Inline implementations

inline
int Multiset::size() const
{
    return m_size;
}

inline
int Multiset::uniqueSize() const
{
    return m_uniqueSize;
}

inline
bool Multiset::empty() const
{
    return size() == 0;
}

inline
int Multiset::erase(const ItemType& value)
{
    return doErase(value, false);
```

```
}

inline
int Multiset::eraseAll(const ItemType& value)
{
    return doErase(value, true);
}

inline
bool Multiset::contains(const ItemType& value) const
{
    return find(value) != -1;
}

#endif // MULTISET_INCLUDED
=====================================================================
// Multiset.cpp

#include "Multiset.h"

void exchange(int& a, int& b)
{
    int t = a;
    a = b;
    b = t;
}

Multiset::Multiset()
 : m_uniqueSize(0), m_size(0)
{}

bool Multiset::insert(const ItemType& value)
{
    if (!insertMany(value, 1, m_uniqueSize))
        return false;
    m_size++;
    return true;
}

int Multiset::count(const ItemType& value) const
{
    int pos = find(value);
    return pos == -1 ? 0 : m_data[pos].m_count;
}

int Multiset::get(int i, ItemType& value) const
{
    if (i < 0  ||  i >= m_uniqueSize)
```

```
        return 0;
    value = m_data[i].m_value;
    return m_data[i].m_count;
}

bool Multiset::getLeastFrequentValue(ItemType& value) const
{
    if (m_uniqueSize == 0)
        return false;

      // Start by assuming position 0 has the unique most frequent value

    int posLeastFrequent = 0;
    bool uniqueLeastFrequent = true;

      // See if any other value is the most frequent

    for (int k = 1; k < m_uniqueSize; k++)
    {
        int diff = (m_data[k].m_count - m_data[posLeastFrequent].m_count);
        if (diff == 0)  // most frequent value, but not unique
            uniqueLeastFrequent = false;
        else if (diff < 0)  // a new unique most frequent value
        {
            posLeastFrequent = k;
            uniqueLeastFrequent = true;
        }
    }

      // Set value and return true only if the most frequent value is unique

    if (uniqueLeastFrequent)
        value = m_data[posLeastFrequent].m_value;
    return uniqueLeastFrequent;
}

bool Multiset::getSmallestValue(ItemType& value) const
{
    if (m_uniqueSize == 0)
        return false;
    int posSmallest = 0;
    for (int k = 1; k < m_uniqueSize; k++)
        if (m_data[k].m_value < m_data[posSmallest].m_value)
            posSmallest = k;
    value = m_data[posSmallest].m_value;
    return true;
}
```

```
bool Multiset::getSecondSmallestValue(ItemType& value) const
{
    if (m_uniqueSize < 2)
        return false;

      // posSmallest[0] contains position of smallest seen so far
      // posSmallest[1] contains position of second smallest seen so far
      // Start by assuming they're in positions 0 and 1 (or 1 and 0).

    int posSmallest[2] = { 0, 1 };
    if (m_data[0].m_value > m_data[1].m_value)
        exchange(posSmallest[0], posSmallest[1]);

      // Check frequencies of other values, and adjust

    for (int k = 2; k < m_uniqueSize; k++)
    {
        const ItemType& v = m_data[k].m_value;
        if (v < m_data[posSmallest[0]].m_value)  // smallest value so far?
        {
            posSmallest[1] = posSmallest[0];  // old smallest now second smallest
            posSmallest[0] = k;
        }
        else if (v < m_data[posSmallest[1]].m_value)  // second smallest?
            posSmallest[1] = k;
    }

      // Set value to second smallest

    value = m_data[posSmallest[1]].m_value;
    return true;
}

bool Multiset::replace(ItemType original, ItemType new_value)
{
    int posOriginal = find(original);
    if (posOriginal == -1)  // no occurrence of original
        return false;
    int posNew = find(new_value);
    if (posNew == -1)  // new_value not already present?
        m_data[posOriginal].m_value = new_value;
    else if (posNew != posOriginal)  // new_value present and distinct from
original?
    {
          // Absorb original count into new_value count and move last array
          // item to eliminate original

        m_data[posNew].m_count += m_data[posOriginal].m_count;
```

```
        m_uniqueSize--;
        m_data[posOriginal] = m_data[m_uniqueSize];
    }
    return true;
}

int Multiset::countIf(char op, ItemType value) const
{
    if (op != '<'  &&  op != '>'  &&  op != '=')
        return -1;
    int count = 0;
    for (int k = 0; k < m_uniqueSize; k++)
    {
        bool meetsCriterion = (op == '<' ? m_data[k].m_value < value :
                               op == '>' ? m_data[k].m_value > value :
                                           m_data[k].m_value == value);
        if (meetsCriterion)
            count += m_data[k].m_count;
    }
    return count;
}

void Multiset::swap(Multiset& other)
{
      // Swap elements.  Since the only elements that matter are those up to
      // m_uniqueSize and other.m_uniqueSize, only they have to be moved.

    int minUniqueSize = (m_uniqueSize < other.m_uniqueSize ?
                                        m_uniqueSize : other.m_uniqueSize);
    for (int k = 0; k < minUniqueSize; k++)
    {
        Node tempNode = m_data[k];
        m_data[k] = other.m_data[k];
        other.m_data[k] = tempNode;
    }

      // If the uniqueSizes are different, assign the remaining elements from
      // the longer one to the shorter.

    if (m_uniqueSize > minUniqueSize)
        for (int k = minUniqueSize; k < m_uniqueSize; k++)
            other.m_data[k] = m_data[k];
    else if (other.m_uniqueSize > minUniqueSize)
        for (int k = minUniqueSize; k < other.m_uniqueSize; k++)
            m_data[k] = other.m_data[k];

      // Swap uniqueSizes and sizes.
```

```
    exchange(m_uniqueSize, other.m_uniqueSize);
    exchange(m_size, other.m_size);
}

void Multiset::copyIntoOtherMultiset(Multiset& other) const
{
      // Works even if this == &other

      // If we add new items to other, we won't update other's m_uniqueSize
      // yet so that the call to find doesn't waste time looking at
      // items it doesn't need to.

    int newUniqueSize = other.m_uniqueSize;
    int newSize = other.m_size;

    for (int k = 0; k < m_uniqueSize; k++)
    {
        if (!other.insertMany(m_data[k].m_value, m_data[k].m_count, newUniqueSize))
            continue;  // No way to signal failure, so just go on
        newSize += m_data[k].m_count;
    }

      // Commit the new elements

    other.m_uniqueSize = newUniqueSize;
    other.m_size = newSize;
}

int Multiset::find(const ItemType& value) const
{
      // Do a linear search through the array.

    for (int pos = 0; pos < m_uniqueSize; pos++)
        if (m_data[pos].m_value == value)
            return pos;
    return -1;
}

bool Multiset::insertMany(const ItemType& value, int nToInsert, int& uniqueSize)
{
    int pos = find(value);

    if (pos != -1)  // found
        m_data[pos].m_count += nToInsert;
    else
    {
        if (uniqueSize == DEFAULT_MAX_ITEMS)  // no room to insert
            return false;
```

```
        m_data[uniqueSize].m_value = value;
        m_data[uniqueSize].m_count = nToInsert;
        uniqueSize++;
    }
    return true;
}

int Multiset::doErase(const ItemType& value, bool all)
{
    int pos = find(value);

    if (pos == -1)  // not found
        return 0;

      // If erasing one, and there are more than one, just decrement

    if (!all  &&  m_data[pos].m_count > 1)
    {
        m_data[pos].m_count--;
        m_size--;
        return 1;
    }

      // If erasing all, or erasing one whose count is 1, move last array
      // item to replace the one to be erased

    int nErased = m_data[pos].m_count;
    m_size -= nErased;
    m_uniqueSize--;
    m_data[pos] = m_data[m_uniqueSize];
    return nErased;
}
```

Problem 4:

```
// StudentMultiset.h

#ifndef STUDENTMULTISET_INCLUDED
#define STUDENTMULTISET_INCLUDED

#include "Multiset.h"  // ItemType is typedef'd to unsigned long

class StudentMultiset
{
  public:
    StudentMultiset();       // Create an empty StudentMultiset.

    bool add(unsigned long id);
      // Add a student id to the StudentMultiset.  Return true if and only
```

```
      // if the id was actually added.

    int size() const;
      // Return the number of ids in the StudentMultiset.  If an id was added
      // n times, it contributes n to the size.

    void print();
      // Print every student id in the StudentMultiset one per line; print
      // as many lines for each id as it occurs in the StudentMultiset.

    private:
      Multiset m_idMultiset;
};

// Inline implementations

inline
int StudentMultiset::size() const
{
    return m_idMultiset.size();
}

#endif // STUDENTMULTISET_INCLUDED
=====================================================================
// StudentMultiset.cpp

#include "Multiset.h"
#include "StudentMultiset.h"
#include <iostream>
using namespace std;

  // Actually, we did not have to declare and implement the default
  // constructor:  If we declare no constructors whatsoever, the compiler
  // writes a default constructor for us that would do nothing more than
  // default construct the m_idMultiset data member.

  // We do not have to declare a destructor, copy constructor, or assignment
  // operator, because the compiler-generated ones do the right thing.

StudentMultiset::StudentMultiset()
{}

bool StudentMultiset::add(unsigned long id)
{
    return m_idMultiset.insert(id);
}

void StudentMultiset::print()
```

```
{
    for (int k = 0; k < m_idMultiset.uniqueSize(); k++)
    {
        unsigned long id;
        int count = m_idMultiset.get(k, id);
        for (int m = 0; m < count; m++)
            cout << id << endl;
    }
}
```

Problem 5:

The few differences from the Problem 3 solution are indicated in boldface.

```
// newMultiset.h

#ifndef NEWMULTISET_INCLUDED
#define NEWMULTISET_INCLUDED

  // Later in the course, we'll see that templates provide a much nicer
  // way of enabling us to have Multisets of different types.  For now, we'll
  // use a typedef.

typedef unsigned long ItemType;

const int DEFAULT_MAX_ITEMS = 200;

class Multiset
{
  public:
    Multiset(int capacity = DEFAULT_MAX_ITEMS);
      // Create an empty multiset with the given capacity.

    bool empty() const;  // Return true if the multiset is empty, otherwise false.

    int size() const;
      // Return the number of items in the multiset.  For example, the size
      // of a multiset containing "cumin", "cumin", "cumin", "turmeric" is 4.

    int uniqueSize() const;
      // Return the number of distinct items in the multiset.  For example,
      // the uniqueSize of a multiset containing "cumin", "cumin", "cumin",
      // "turmeric" is 2.

    bool insert(const ItemType& value);
      // Insert value into the multiset.  Return true if the value was
      // actually inserted.  Return false if the value was not inserted
      // (perhaps because the multiset has a fixed capacity and is full).
```

```
    int erase(const ItemType& value);
      // Remove one instance of value from the multiset if present.
      // Return the number of instances removed, which will be 1 or 0.

    int eraseAll(const ItemType& value);
      // Remove all instances of value from the multiset if present.
      // Return the number of instances removed.

    bool contains(const ItemType& value) const;
      // Return true if the value is in the multiset, otherwise false.

    int count(const ItemType& value) const;
      // Return the number of instances of value in the multiset.

    int get(int i, ItemType& value) const;
      // If 0 <= i < uniqueSize(), copy into value an item in the multiset
      // and return the number of instances of that item in the multiset.
      // Otherwise, leave value unchanged and return 0.

    bool getLeastFrequentValue(ItemType& value) const;
      // If there exists a single item that has the least number of instances in the
multiset,
      // then copy into value that item in the multiset and return true.
      // However, if there exist more than 1 item that have the least number of
instances in the multiset,
      // then do not copy into value any item in the multiset and return false. In
other words, value should remain unchanged.
      // If there's no item in the multiset, return false.

    bool getSmallestValue(ItemType& value) const;
      // If there exists a value that is the smallest value among all the values in
the multiset,
      // then copy into value that item in the multiset and return true
      // Otherwise, return false.
      // For both unsigned long and string data type, the lower value can be found
by using less than operator (<).
      // For example, 10 is smaller than 20, so 10 < 20 is true.
      // "ABC" is smaller than "XYZ", so "ABC" < "XYZ" is true.

    bool getSecondSmallestValue(ItemType& value) const;
      // Similar to getSmallestValue(), but this time you need to find the second
smallest value.
      // If there exists a value that is the 2nd smallest value among all the values
in the multiset,
      // then copy into value that item in the multiset and return true.
      // Otherwise, return false.
      // Please note that you cannot use any sorting algorithm to sort the multiset.
```

```
  bool replace(ItemType original, ItemType new_value);
    // Replace the item that has the value equal to original by the new value
    // new_value.  For example, replace("ABC","XYZ") will search the multiset
    // for the item "ABC" and replace all occurrences of "ABC" as "XYZ".
    // If the replacement is successful, then return true.  If there is no
    // item to be replaced, then return false.

  int countIf(char op, ItemType value) const;
    // Count the number of items that the item is greater than, less than, or
    // equal to value.  For example:  countIf('>',100) returns the number of
    // items in multiset in which the item is greater than 100.

  void swap(Multiset& other);
    // Exchange the contents of this multiset with the other one.

  void copyIntoOtherMultiset(Multiset& other) const;
    // Insert all the items into the multiset in other.

    // Housekeeping functions
  ~Multiset();
  Multiset(const Multiset& other);
  Multiset& operator=(const Multiset& rhs);

private:

    // Since this structure is used only by the implementation of the
    // Multiset class, we'll make it private to Multiset.

  struct Node
  {
      ItemType m_value;
      int      m_count;
  };
  Node* m_data;       // dynamic array of the items in the multiset, with counts
  int   m_uniqueSize; // how many distinct items in the multiset
  int   m_size;       // total number of items in the multiset
  int   m_capacity;   // the maximum number of items

    // At any time, the elements of m_data indexed from 0 to m_uniqueSize-1
    // are in use.  m_size is the sum of the m_counts of those elements.

  int find(const ItemType& value) const;
    // Return index of the node in m_data whose m_value == value if there is
    // one, else -1

  bool insertMany(const ItemType& value, int nToInsert, int& uniqueSize);
    // If no room to insert nToInsert items, return false.  Otherwise,
    // insert value with (additional) count nToInsert using (and possibly
```

```
      // updating) uniqueSize as the number of distinct existing items, and
      // return true.

    int doErase(const ItemType& value, bool all);
      // Remove one or all instances of value from the multiset if present,
      // depending on the second parameter.  Return the number of instances
      // removed.
};

// Inline implementations

inline
int Multiset::size() const
{
    return m_size;
}

inline
int Multiset::uniqueSize() const
{
    return m_uniqueSize;
}

inline
bool Multiset::empty() const
{
    return size() == 0;
}

inline
int Multiset::erase(const ItemType& value)
{
    return doErase(value, false);
}

inline
int Multiset::eraseAll(const ItemType& value)
{
    return doErase(value, true);
}

inline
bool Multiset::contains(const ItemType& value) const
{
    return find(value) != -1;
}

#endif // NEWMULTISET_INCLUDED
```

```
================================================================
// newMultiset.cpp

#include "newMultiset.h"
#include <iostream>
#include <cstdlib>

void exchange(int& a, int& b)
{
    int t = a;
    a = b;
    b = t;
}

Multiset::Multiset(int capacity)
 : m_uniqueSize(0), m_size(0), m_capacity(capacity)
{
    if (capacity < 0)
    {
        std::cout << "A Multiset capacity must not be negative." << std::endl;
        std::exit(1);
    }
    m_data = new Node[m_capacity];
}

Multiset::Multiset(const Multiset& other)
 : m_uniqueSize(other.m_uniqueSize), m_size(other.m_size),
   m_capacity(other.m_capacity)
{
    m_data = new Node[m_capacity];
    for (int k = 0; k < m_uniqueSize; k++)
        m_data[k] = other.m_data[k];
}

Multiset::~Multiset()
{
    delete [] m_data;
}

Multiset& Multiset::operator=(const Multiset& rhs)
{
    if (this != &rhs)
    {
        Multiset temp(rhs);
        swap(temp);
    }
    return *this;
}
```

```
bool Multiset::insert(const ItemType& value)
{
    if (!insertMany(value, 1, m_uniqueSize))
        return false;
    m_size++;
    return true;
}

int Multiset::count(const ItemType& value) const
{
    int pos = find(value);
    return pos == -1 ? 0 : m_data[pos].m_count;
}

int Multiset::get(int i, ItemType& value) const
{
    if (i < 0  ||  i >= m_uniqueSize)
        return 0;
    value = m_data[i].m_value;
    return m_data[i].m_count;
}

bool Multiset::getLeastFrequentValue(ItemType& value) const
{
    if (m_uniqueSize == 0)
        return false;

      // Start by assuming position 0 has the unique most frequent value

    int posLeastFrequent = 0;
    bool uniqueLeastFrequent = true;

      // See if any other value is the most frequent

    for (int k = 1; k < m_uniqueSize; k++)
    {
        int diff = (m_data[k].m_count - m_data[posLeastFrequent].m_count);
        if (diff == 0)  // most frequent value, but not unique
            uniqueLeastFrequent = false;
        else if (diff < 0)  // a new unique most frequent value
        {
            posLeastFrequent = k;
            uniqueLeastFrequent = true;
        }
    }

      // Set value and return true only if the most frequent value is unique
```

```
    if (uniqueLeastFrequent)
        value = m_data[posLeastFrequent].m_value;
    return uniqueLeastFrequent;
}

bool Multiset::getSmallestValue(ItemType& value) const
{
    if (m_uniqueSize == 0)
        return false;
    int posSmallest = 0;
    for (int k = 1; k < m_uniqueSize; k++)
        if (m_data[k].m_value < m_data[posSmallest].m_value)
            posSmallest = k;
    value = m_data[posSmallest].m_value;
    return true;
}

bool Multiset::getSecondSmallestValue(ItemType& value) const
{
    if (m_uniqueSize < 2)
        return false;

      // posSmallest[0] contains position of smallest seen so far
      // posSmallest[1] contains position of second smallest seen so far
      // Start by assuming they're in positions 0 and 1 (or 1 and 0).

    int posSmallest[2] = { 0, 1 };
    if (m_data[0].m_value > m_data[1].m_value)
        exchange(posSmallest[0], posSmallest[1]);

      // Check frequencies of other values, and adjust

    for (int k = 2; k < m_uniqueSize; k++)
    {
        const ItemType& v = m_data[k].m_value;
        if (v < m_data[posSmallest[0]].m_value)  // smallest value so far?
        {
            posSmallest[1] = posSmallest[0];  // old smallest now second smallest
            posSmallest[0] = k;
        }
        else if (v < m_data[posSmallest[1]].m_value)  // second smallest?
            posSmallest[1] = k;
    }

      // Set value to second smallest

    value = m_data[posSmallest[1]].m_value;
```

```
    return true;
}

bool Multiset::replace(ItemType original, ItemType new_value)
{
    int posOriginal = find(original);
    if (posOriginal == -1)  // no occurrence of original
        return false;
    int posNew = find(new_value);
    if (posNew == -1)  // new_value not already present?
        m_data[posOriginal].m_value = new_value;
    else if (posNew != posOriginal)  // new_value present and distinct from
original?
    {
          // Absorb original count into new_value count and move last array
          // item to eliminate original

        m_data[posNew].m_count += m_data[posOriginal].m_count;
        m_uniqueSize--;
        m_data[posOriginal] = m_data[m_uniqueSize];
    }
    return true;
}

int Multiset::countIf(char op, ItemType value) const
{
    if (op != '<'  &&  op != '>'  &&  op != '=')
        return -1;
    int count = 0;
    for (int k = 0; k < m_uniqueSize; k++)
    {
        bool meetsCriterion = (op == '<' ? m_data[k].m_value < value :
                               op == '>' ? m_data[k].m_value > value :
                                           m_data[k].m_value == value);
        if (meetsCriterion)
            count += m_data[k].m_count;
    }
    return count;
}

void Multiset::swap(Multiset& other)
{
      // Swap the m_data pointers to dynamic arrays.

    Node* tempData = m_data;
    m_data = other.m_data;
    other.m_data = tempData;
```

```
      // Swap uniqueSize, size, and capacity.

    exchange(m_uniqueSize, other.m_uniqueSize);
    exchange(m_size, other.m_size);
    exchange(m_capacity, other.m_capacity);
}

void Multiset::copyIntoOtherMultiset(Multiset& other) const
{
      // Works even if this == &other

      // If we add new items to other, we won't update other's m_uniqueSize
      // yet so that the call to find doesn't waste time looking at
      // items it doesn't need to.

    int newUniqueSize = other.m_uniqueSize;
    int newSize = other.m_size;

    for (int k = 0; k < m_uniqueSize; k++)
    {
        if (!other.insertMany(m_data[k].m_value, m_data[k].m_count, newUniqueSize))
            continue;  // No way to signal failure, so just go on
        newSize += m_data[k].m_count;
    }

      // Commit the new elements

    other.m_uniqueSize = newUniqueSize;
    other.m_size = newSize;
}

int Multiset::find(const ItemType& value) const
{
      // Do a linear search through the array.

    for (int pos = 0; pos < m_uniqueSize; pos++)
        if (m_data[pos].m_value == value)
            return pos;
    return -1;
}

bool Multiset::insertMany(const ItemType& value, int nToInsert, int& uniqueSize)
{
    int pos = find(value);

    if (pos != -1)  // found
        m_data[pos].m_count += nToInsert;
    else
```

```
    {
        if (uniqueSize == m_capacity)  // no room to insert
            return false;
        m_data[uniqueSize].m_value = value;
        m_data[uniqueSize].m_count = nToInsert;
        uniqueSize++;
    }
    return true;
}

int Multiset::doErase(const ItemType& value, bool all)
{
    int pos = find(value);

    if (pos == -1)  // not found
        return 0;

      // If erasing one, and there are more than one, just decrement

    if (!all  &&  m_data[pos].m_count > 1)
    {
        m_data[pos].m_count--;
        m_size--;
        return 1;
    }

      // If erasing all, or erasing one whose count is 1, move last array
      // item to replace the one to be erased

    int nErased = m_data[pos].m_count;
    m_size -= nErased;
    m_uniqueSize--;
    m_data[pos] = m_data[m_uniqueSize];
    return nErased;
}

================================================================================
================
PROJECT 2 SOLUTION
================================================================================
==================

Summer 2016 CS 32
Project 2 Solution

In this solution, the functions with small, fast implementations are inlined.
Alternatively, the inline keyword can be removed and the function implementations
moved to Multiset.cpp. (inline will be mentioned at some point in class, so don't
```

worry if you've never seen it before.)

```
// Multiset.h

#ifndef MULTISET_INCLUDED
#define MULTISET_INCLUDED

  // Later in the course, we'll see that templates provide a much nicer
  // way of enabling us to have Multisets of different types.  For now, we'll
  // use a typedef.

typedef some type ItemType;

class Multiset
{
  public:
    Multiset();          // Create an empty multiset.
    bool empty() const;  // Return true if the multiset is empty, otherwise false.

    int size() const;
      // Return the number of items in the multiset.  For example, the size
      // of a multiset containing "cumin", "cumin", "cumin", "turmeric" is 4.

    int uniqueSize() const;
      // Return the number of distinct items in the multiset.  For example,
      // the uniqueSize of a multiset containing "cumin", "cumin", "cumin",
      // "turmeric" is 2.

    bool insert(const ItemType& value);
      // Insert value into the multiset.  Return true if the value was
      // actually inserted.  Return false if the value was not inserted
      // (perhaps because the multiset has a fixed capacity and is full).

    int erase(const ItemType& value);
      // Remove one instance of value from the multiset if present.
      // Return the number of instances removed, which will be 1 or 0.

    int eraseAll(const ItemType& value);
      // Remove all instances of value from the multiset if present.
      // Return the number of instances removed.

    bool contains(const ItemType& value) const;
      // Return true if the value is in the multiset, otherwise false.

    int count(const ItemType& value) const;
      // Return the number of instances of value in the multiset.

    int get(int i, ItemType& value) const;
```

```
      // If 0 <= i < uniqueSize(), copy into value an item in the multiset
      // and return the number of instances of that item in the multiset.
      // Otherwise, leave value unchanged and return 0.

    bool geLeastFrequentValue(ItemType& value) const;
      // If there exists a single item that has the least number of instances in the
multiset,
      // then copy into value that item in the multiset and return true.
      // However, if there exist more than 1 item that have the least number of
instances in the multiset,
      // then do not copy into value any item in the multiset and return false. In
other words, value should remain unchanged.
      // If there's no item in the multiset, return false.

    bool getSmallestValue(ItemType& value) const;
      // If there exists a value that is the smallest value among all the values in
the multiset,
      // then copy into value that item in the multiset and return true
      // Otherwise, return false.
      // For both unsigned long and string data type, the lower value can be found
by using less than operator (<).
      // For example, 10 is smaller than 20, so 10 < 20 is true.
      // "ABC" is smaller than "XYZ", so "ABC" < "XYZ" is true.

    bool getSecondSmallestValue(ItemType& value) const;
      // Similar to getSmallestValue(), but this time you need to find the second
smallest value.
      // If there exists a value that is the 2nd smallest value among all the values
in the multiset,
      // then copy into value that item in the multiset and return true.
      // Otherwise, return false.
      // Please note that you cannot use any sorting algorithm to sort the multiset.

    bool replace(ItemType original, ItemType new_value);
      // Replace the item that has the value equal to original by the new value
      // new_value.  For example, replace("ABC","XYZ") will search the multiset
      // for the item "ABC" and replace all occurrences of "ABC" as "XYZ".
      // If the replacement is successful, then return true.  If there is no
      // item to be replaced, then return false.

    int countIf(char op, ItemType value) const;
      // Count the number of items that the item is greater than, less than, or
      // equal to value.  For example:  countIf('>',100) returns the number of
      // items in multiset in which the item is greater than 100.

    void swap(Multiset& other);
      // Exchange the contents of this multiset with the other one.
```

```
    void copyIntoOtherMultiset(Multiset& other) const;
      // Insert all the items into the multiset in other.

      // Housekeeping functions
    ~Multiset();
    Multiset(const Multiset& other);
    Multiset& operator=(const Multiset& rhs);

  private:
      // Representation:
      //   a circular doubly-linked list with a dummy node.
      //   m_head points to the dummy node.
      //   m_head->m_prev->m_next == m_head and m_head->m_next->m_prev == m_head
      //   m_uniqueSize == 0 and m_size == 0  if and only if
      //                             m_head->m_next == m_head->m_prev == m_head
      //   In addition to the dummy node, the list has m_uniqueSize nodes.
      //   Nodes are in no particular order.

    struct Node
    {
        ItemType m_value;
        int      m_count;
        Node*    m_next;
        Node*    m_prev;
    };

    Node* m_head;
    int   m_uniqueSize;
    int   m_size;

    Node* find(const ItemType& value) const;
      // Return pointer to Node whose m_value == value if there is one,
      // else m_head

    Node* insertHelper(const ItemType& value, int nToInsert);
      // If value is present, increase its count by nToInsert and return nullptr.
      // Otherwise, create a new node with the value and count and return a
      // pointer to it (with m_next and m_prev being uninitialized).

    int doErase(const ItemType& value, bool all);
      // Remove one or all instances of value from the multiset if present,
      // depending on the second parameter.  Return the number of instances
      // removed.
};

// Declarations of non-member functions

void combine(const Multiset& ms1, const Multiset& ms2, Multiset& result);
```

```
      // If a value occurs n1 times in ms1 and n2 times in ms2, then
      // it will occur n1+n2 times in result upon return from this function.

void subtract(const Multiset& ms1, const Multiset& ms2, Multiset& result);
      // If a value occurs n1 times in ms1 and n2 times in ms2, then
      // it will occur n1-n2 times in result upon return from this function
      // if n1 >= n2.  If n1 <= n2, it will not occur in result.

// Inline implementations

inline
int Multiset::size() const
{
    return m_size;
}

inline
int Multiset::uniqueSize() const
{
    return m_uniqueSize;
}

inline
bool Multiset::empty() const
{
    return size() == 0;
}

inline
int Multiset::erase(const ItemType& value)
{
    return doErase(value, false);
}

inline
int Multiset::eraseAll(const ItemType& value)
{
    return doErase(value, true);
}

inline
bool Multiset::contains(const ItemType& value) const
{
    return find(value) != m_head;
}

#endif // MULTISET_INCLUDED
```

```
// Multiset.cpp

#include "Multiset.h"
#include <utility>

Multiset::Multiset()
 : m_uniqueSize(0), m_size(0)
{
      // create dummy node
    m_head = new Node;
    m_head->m_next = m_head;
    m_head->m_prev = m_head;
}

Multiset::~Multiset()
{
      // Delete the m_uniqueSize non-dummy nodes plus the dummy node

    for (Node* p = m_head->m_prev ; m_uniqueSize >= 0; m_uniqueSize--)
    {
        Node* toBeDeleted = p;
        p = p->m_prev;
        delete toBeDeleted;
    }
}

Multiset::Multiset(const Multiset& other)
 : m_uniqueSize(other.m_uniqueSize), m_size(other.m_size)
{
      // Create dummy node; don't initialize its m_next

    m_head = new Node;
    m_head->m_prev = m_head;

      // Copy each node from the other list; each iteration will set the
      // m_next of the previous node copied

    for (Node* p = other.m_head->m_next ; p != other.m_head; p = p->m_next)
    {
          // Create a copy of the node p points to
        Node* pnew = new Node;
        pnew->m_value = p->m_value;
        pnew->m_count = p->m_count;

          // Connect the m_prev pointers
        pnew->m_prev = m_head->m_prev;
        m_head->m_prev = pnew;
```

```
          // Connect the previous Node's m_next
        pnew->m_prev->m_next = pnew;
    }

      // Connect the last Node's m_next
    m_head->m_prev->m_next = m_head;
}

Multiset& Multiset::operator=(const Multiset& rhs)
{
    if (this != &rhs)
    {
        Multiset temp(rhs);
        swap(temp);
    }
    return *this;
}

bool Multiset::insert(const ItemType& value)
{
    Node* p = insertHelper(value, 1);

    if (p != nullptr)
    {
          // Insert new node at tail of list (arbitrary choice of position)
          //     Connect it to tail
        p->m_prev = m_head->m_prev;
        p->m_prev->m_next = p;

          //     Connect it to dummy node
        p->m_next = m_head;
        m_head->m_prev = p;

        m_uniqueSize++;
    }

    m_size++;
    return true;
}

int Multiset::count(const ItemType& value) const
{
    Node* p = find(value);
    return p == m_head ? 0 : p->m_count;
}

int Multiset::get(int i, ItemType& value) const
{
```

```
    if (i < 0  ||  i >= m_uniqueSize)
        return 0;

      // Get the value at position i.  This is one way of ensuring the required
      // behavior of get:  If the Multiset doesn't change in the interim,
      // * calling get with each i in 0 <= i < size() gets each of the
      //   Multiset elements, and
      // * calling get with the same value of i each time gets the same element.

      // If i is closer to the head of the list, go forward to reach that
      // position; otherwise, start from tail and go backward.

    Node* p;
    if (i < m_uniqueSize / 2)  // closer to head
    {
        p = m_head->m_next;
        for (int k = 0; k != i; k++)
            p = p->m_next;
    }
    else  // closer to tail
    {
        p = m_head->m_prev;
        for (int k = m_uniqueSize-1; k != i; k--)
            p = p->m_prev;
    }

    value = p->m_value;
    return p->m_count;
}

bool Multiset::geLeastFrequentValue(ItemType& value) const
{
    if (empty())
        return false;

      // Start by assuming first value is the unique least frequent value

    Node* pLeastFrequent = m_head->m_next;
    bool uniqueLeastFrequent = true;

      // See if any other value is the least frequent

    for (Node* p = pLeastFrequent->m_next; p != m_head; p = p->m_next)
    {
        int diff = (p->m_count - pLeastFrequent->m_count);
        if (diff == 0)  // least frequent value, but not unique
            uniqueLeastFrequent = false;
        else if (diff < 0)  // a new unique least frequent value
```

```
        {
            pLeastFrequent = p;
            uniqueLeastFrequent = true;
        }
    }

      // Set value and return true only if the least frequent value is unique

    if (uniqueLeastFrequent)
        value = pLeastFrequent->m_value;
    return uniqueLeastFrequent;
}

bool Multiset::getSmallestValue(ItemType& value) const
{
    if (empty())
        return false;

      // Start by assuming first value is the smallest

    Node* pSmallest = m_head->m_next;

      // See if any other value is smaller

    for (Node* p = pSmallest->m_next; p != m_head; p = p->m_next)
        if (p->m_value < pSmallest->m_value)
            pSmallest = p;
    value = pSmallest->m_value;
    return true;
}

bool Multiset::getSecondSmallestValue(ItemType& value) const
{
    if (m_uniqueSize < 2)
        return false;

      // pSmallest1 points to smallest seen so far
      // pSmallest2 points to second smallest seen so far
      // Start by assuming they're in first two positions.

    Node* pSmallest1 = m_head->m_next;
    Node* pSmallest2 = pSmallest1->m_next;
    Node* third = pSmallest2->m_next;

    if (pSmallest1->m_value > pSmallest2->m_value)
        std::swap(pSmallest1, pSmallest2);

      // Check frequencies of other values, and adjust
```

```
    for (Node* p = third; p != m_head; p = p->m_next)
    {
        const ItemType& v = p->m_value;
        if (v < pSmallest1->m_value)  // smallest value so far?
        {
            pSmallest2 = pSmallest1;  // old smallest now second smallest
            pSmallest1 = p;
        }
        else if (v < pSmallest2->m_value)  // second smallest?
            pSmallest2 = p;
    }

      // Set value to second smallest

    value = pSmallest2->m_value;
    return true;
}

bool Multiset::replace(ItemType original, ItemType new_value)
{
    Node* pOriginal = find(original);
    if (pOriginal == m_head)  // no occurrence of original
        return false;
    Node* pNew = find(new_value);
    if (pNew == m_head)  // new_value not already present?
        pOriginal->m_value = new_value;
    else if (pNew != pOriginal)  // new_value present and distinct from original?
    {
          // Absorb original count into new_value count and eliminate original

        pNew->m_count += pOriginal->m_count;
        pOriginal->m_prev->m_next = pOriginal->m_next;
        pOriginal->m_next->m_prev = pOriginal->m_prev;
        delete pOriginal;
        m_uniqueSize--;
    }
    return true;
}

int Multiset::countIf(char op, ItemType value) const
{
    if (op == '=')
        return count(value);
    if (op != '<'  &&  op != '>')
        return -1;
    int total = 0;
    for (Node* p = m_head->m_next; p != m_head; p = p->m_next)
```

```
    {
        bool meetsCriterion = (op == '<' ? p->m_value < value :
                                           p->m_value > value);
        if (meetsCriterion)
            total += p->m_count;
    }
    return total;
}

void Multiset::swap(Multiset& other)
{
    std::swap(m_head, other.m_head);
    std::swap(m_uniqueSize, other.m_uniqueSize);
    std::swap(m_size, other.m_size);
}

void Multiset::copyIntoOtherMultiset(Multiset& other) const
{
      // Works even if this == &other

      // If we add new items to other, we won't insert them into  other's
      // list yet so that the call to find doesn't waste time looking at
      // items it doesn't need to.

    int newUniqueSize = other.m_uniqueSize;
    int newSize = other.m_size;
    Node* oldTail = other.m_head->m_prev;
    Node* newTail = oldTail;

    for (Node* p = m_head->m_next ; p != m_head; p = p->m_next)
    {
        Node* pNew = other.insertHelper(p->m_value, p->m_count);
        if (pNew != nullptr)
        {
            pNew->m_prev = newTail;
            newTail = pNew;
            newUniqueSize++;
        }
        newSize += p->m_count;
    }

      // Commit any new elements

    other.m_uniqueSize = newUniqueSize;
    other.m_size = newSize;

    if (newTail != oldTail)
    {
```

```
        other.m_head->m_prev = newTail;
        Node* p = other.m_head;
        do
        {
            p->m_prev->m_next = p;
            p = p->m_prev;
        } while (p != oldTail);
    }
}

Multiset::Node* Multiset::find(const ItemType& value) const
{
      // Do a linear search through the list

    Node* p;
    for (p = m_head->m_next; p != m_head && p->m_value != value; p = p->m_next)
        ;
    return p;
}

Multiset::Node* Multiset::insertHelper(const ItemType& value, int nToInsert)
{
    Node* p = find(value);

    if (p != m_head)  // found
    {
        p->m_count += nToInsert;
        return nullptr;
    }
    else
    {
          // Create a new node
        p = new Node;
        p->m_value = value;
        p->m_count = nToInsert;
        return p;
    }
}

int Multiset::doErase(const ItemType& value, bool all)
{
    Node* p = find(value);

    if (p == m_head)  // not found
        return 0;

    int nErased = (all ? p->m_count : 1);  // number to erase
```

```
      // If erasing one, and there are more than one, just decrement;
      // otherwise, we're erasing all, or erasing one whose count is 1,
      // so unlink the Node from the list and destroy it

    if (!all  &&  p->m_count > 1)
        p->m_count--;
    else
    {
        p->m_prev->m_next = p->m_next;
        p->m_next->m_prev = p->m_prev;
        delete p;

        m_uniqueSize--;
    }

    m_size -= nErased;
    return nErased;
}

void combine(const Multiset& ms1, const Multiset& ms2, Multiset& result)
{
      // Guard against the case that result is an alias for ms1 or ms2
      // (i.e., that result is a reference to the same multiset that ms1 or ms2
      // refers to) by building the answer in a local variable res.  When
      // done, swap res with result; the old value of result (now in res) will
      // be destroyed when res is destroyed.

    Multiset res(ms1);
    for (int k = 0; k < ms2.uniqueSize(); k++)
    {
        ItemType v;
        for (int n = ms2.get(k, v); n > 0; n--)
            res.insert(v);
    }
    result.swap(res);
}

void subtract(const Multiset& ms1, const Multiset& ms2, Multiset& result)
{
      // Guard against the case that result is an alias for ms1 or ms2
      // by building the answer in a local variable res.  When done, swap res
      // with result; the old value of result (now in res) will be destroyed
      // when res is destroyed.

    Multiset res;
    for (int k = 0; k != ms1.uniqueSize(); k++)
    {
        ItemType v;
```

```
        int n = ms1.get(k, v);
        for (n -= ms2.count(v); n > 0; n--)
            res.insert(v);
    }
    result.swap(res);
}
```