Project 2 Solution

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

```
// Set.h

#ifndef SET_INCLUDED
#define SET_INCLUDED

#include <string>

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

typedef std::string ItemType;

class Set
{
  public:
    Set();               // Create an empty set.
    bool empty() const;  // Return true if the set is empty, otherwise false.
    int size() const;    // Return the number of items in the set.

    bool insert(const ItemType& value);
      // Insert value into the set if it is not already present.  Return
      // true if the value was actually inserted.  Leave the set unchanged
      // and return false if the value was not inserted (perhaps because it
      // is already in the set or because the set has a fixed capacity and
      // is full).

    bool erase(const ItemType& value);
      // Remove the value from the set if present.  Return true if the
      // value was removed; otherwise, leave the set unchanged and
      // return false.

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

    bool get(int i, ItemType& value) const;
      // If 0 <= i < size(), copy into value an item in the set and
      // return true.  Otherwise, leave value unchanged and return false.

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

```
      // Housekeeping functions
    ~Set();
    Set(const Set& other);
    Set& operator=(const Set& 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_size == 0  iff  m_head->m_next == m_head->m_prev == m_head

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

    Node* m_head;
    int   m_size;

    void createEmpty();
      // Create an empty list.  (Will be called only by constructors.)

    void insertAtTail(const ItemType& value);
      // Insert value in a new Node at the tail of the list, incrementing
      // m_size.

    void doErase(Node* p);
      // Remove the Node p, decrementing m_size.

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

// Declarations of non-member functions

void unite(const Set& s1, const Set& s2, Set& result);
      // result = { x | (x in s1) OR (x in s2) }

void subtract(const Set& s1, const Set& s2, Set& result);
      // result = { x | (x in s1) AND NOT (x in s2) }

// Inline implementations

inline
```

```
int Set::size() const
{
    return m_size;
}

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

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

#endif // SET_INCLUDED

=====================================================================

// Set.cpp

#include "Set.h"

Set::Set()
{
    createEmpty();
}

bool Set::insert(const ItemType& value)
{
      // Fail if value already present

    if (contains(value) )
        return false;

      // Insert new Node (at tail; choice of position is arbitrary),
      // incrementing m_size

    insertAtTail(value);
    return true;
}

bool Set::erase(const ItemType& value)
{
      // Find the Node with the value, failing if there is none.
```

```
    Node* p = find(value);
    if (p == m_head)
        return false;

      // Erase the Node, decrementing m_size
    doErase(p);
    return true;
}

bool Set::get(int i, ItemType& value) const
{
    if (i < 0  ||  i >= m_size)
        return false;

      // Return the value at position i.  This is one way of ensuring the
      // required behavior of get:  If the Set doesn't change in the interim,
      // * calling get with each i in 0 <= i < size() gets each of the
      //   Set 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_size / 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_size-1; k != i; k--)
            p = p->m_prev;
    }

    value = p->m_value;
    return true;
}

void Set::swap(Set& other)
{
      // Swap head pointers

    Node* p = other.m_head;
    other.m_head = m_head;
    m_head = p;
```

```
      // Swap sizes

    int s = other.m_size;
    other.m_size = m_size;
    m_size = s;
}

Set::~Set()
{
      // Delete all Nodes from first non-dummy up to but not including
      // the dummy

    while (m_head->m_next != m_head)
        doErase(m_head->m_next);

      // delete the dummy

    delete m_head;
}

Set::Set(const Set& other)
{
    createEmpty();

      // Copy all non-dummy other Nodes.  (This will set m_size.)
      // Inserting each new node at the tail rather than anywhere else is
      // an arbitrary choice.

    for (Node* p = other.m_head->m_next; p != other.m_head; p = p->m_next)
        insertAtTail(p->m_value);
}

Set& Set::operator=(const Set& rhs)
{
    if (this != &rhs)
    {
          // Copy and swap idiom

        Set temp(rhs);
        swap(temp);
    }
    return *this;
}

void Set::createEmpty()
{
    m_size = 0;
```

```
      // Create dummy node

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

void Set::insertAtTail(const ItemType& value)
{
      // Create a new node

    Node* newNode = new Node;
    newNode->m_value = value;

      // Insert new item at tail of list (predecessor of the dummy at m_head)
      //     Adjust forward links

    newNode->m_next = m_head;
    m_head->m_prev->m_next = newNode;

      //     Adjust backward links

    newNode->m_prev = m_head->m_prev;
    m_head->m_prev = newNode;

    m_size++;
}

void Set::doErase(Node* p)
{
      // Unlink p from the list and destroy it

    p->m_prev->m_next = p->m_next;
    p->m_next->m_prev = p->m_prev;
    delete p;

    m_size--;
}

Set::Node* Set::find(const ItemType& value) const
{
      // Walk through the list looking for a match

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

```
}

void unite(const Set& s1, const Set& s2, Set& result)
{
      // Check for aliasing to get correct behavior or better performance:
      // If result is s1 and s2, result already is the union.
      // If result is s1, insert s2's elements into result.
      // If result is s2, insert s1's elements into result.
      // If result is a distinct set, assign it s1's contents, then
      //   insert s2's elements in result, unless s2 is s1, in which
      //   case result now already is the union.

    const Set* sp = &s2;
    if (&result == &s1)
    {
        if (&result == &s2)
                return;
    }
    else if (&result == &s2)
        sp = &s1;
    else
    {
        result = s1;
        if (&s1 == &s2)
            return;
    }
    for (int k = 0; k < sp->size(); k++)
    {
        ItemType v;
        sp->get(k, v);
        result.insert(v);
    }
}

void subtract(const Set& s1, const Set& s2, Set& result)
{
      // Guard against the case that result is an alias for s2 by copying
      // s2 to a local variable.  This implementation needs no precaution
      // against result being an alias for s1.

    Set s2copy(s2);
    result = s1;
    for (int k = 0; k < s2copy.size(); k++)
    {
        ItemType v;
        s2copy.get(k, v);
        result.erase(v);
    }
```

```
}
```