SET.H

```
#ifndef SET_H
#define SET_H

#include <string>

using ItemType = std::string;

class Set
{
public:
    Set();
    Set(const Set& other);
    Set& operator=(const Set& rhs);
    ~Set();
    bool empty() const;
    int size() const;
    bool insert(const ItemType& value);
    bool erase(const ItemType& value);
    bool contains(const ItemType& value) const;
    bool get(int pos, ItemType& value) const;
    void swap(Set& other);

    void dump() const;
private:
    int m_size;
    struct Node
    {
        ItemType m_data;
        Node* m_next;
        Node* m_prev;
    };

    Node* head;
    Node* dummy;

    void insertAtFront(const ItemType& value);
};


//Non Member Function Declarations
void unite(const Set& s1, const Set& s2, Set& result);
void butNot(const Set& s1, const Set& s2, Set& result);

#endif //SET_H
```

SET.CPP

```
#include <iostream>

#include "Set.h"

using namespace std;

//Constructor, creates an empty set with no elements
Set::Set()
{
    m_size = 0;

    //Create an empty circular doubly-linked list with dummy node
    dummy = new Node;
    dummy->m_next = dummy;
    dummy->m_prev = dummy;
    head = dummy;
}

//Copy constructor
Set::Set(const Set& other)
{
    m_size = other.m_size; //copy size
    dummy = new Node; //create empty circular doubly-linked list
    dummy->m_next = dummy;
    dummy->m_prev = dummy;
    head = dummy;

    //iterate through other set and copy each node
    for (Node* p = other.head->m_next; p != other.head; p = p->m_next)
    {
        insertAtFront(p->m_data);
    }
}

//Assignment operator
Set& Set::operator=(const Set& rhs)
{
    if (this != &rhs)
    {
        Set temp(rhs);
        swap(temp);
    }

    return *this;
}

//Destructor
Set::~Set()
{
    //Traverse the list to deallocate each node
    Node* curr = head->m_next;
    while (curr != head)
    {
```

```
        Node* next = curr->m_next; //Save the next node so curr can store it after
the current node is deleted
        delete curr;
        curr = next;
    }

    delete dummy; //Delete dynamically allocated dummy node
}

//Return true is the set is empty
bool Set::empty() const
{
    if (m_size == 0)
    {
        return true;
    }

    return false;
}

//Returns how many elements
int Set::size() const
{
    return m_size;
}

//Return true if the value is in the set
bool Set::contains(const ItemType& value) const
{
    //Traverse the linked list, if the element value equal value, return true
    for (Node* p = head->m_next; p != head; p = p->m_next)
    {
        if (p->m_data == value)
        {
            return true;
        }
    }

    return false;
}

//This is a helper function for Set::insert, it inserts every node at the front of
the linked list
void Set::insertAtFront(const ItemType& value)
{
    Node* toAdd = new Node; //Create a new node
    toAdd->m_data = value;  //Set value

    //Set next and previous pointers for new node
    toAdd->m_next = dummy->m_next;
    toAdd->m_prev = dummy;

    //This differs between an empty list or a filled list
    //if empty: dummy->m_next is the dummy node, if filled: dummy->m_next will be
```

```
the original first node
    dummy->m_next->m_prev = toAdd;

    dummy->m_next = toAdd; //Set dummy node's next pointer
}

//Returns true if value is successfully inserted
bool Set::insert(const ItemType& value)
{
    if (!contains(value))
    {
        insertAtFront(value);
        m_size++;

        return true;
    }

    return false;
}

bool Set::erase(const ItemType& value)
{
    //If list is not empty, we can delete something
    if (!empty())
    {
        //Iterate through the list to find value, if we finish iteration and did
not find a value, return false
        for (Node* p = head->m_next; p != head; p = p->m_next)
        {
            //If found equal value, begin deletion
            if (p->m_data == value)
            {
                Node* toBeDeleted = p;
                p->m_prev->m_next = p->m_next; //Set previous node's next pointer
                p->m_next->m_prev = p->m_prev; //Set next node's prev pointer
                delete toBeDeleted;

                m_size--;

                return true;
            }
        }

        return false;
    }

    return false;
}

//Copy into value the element that is strictly greater than pos number of elements
if 0 < pos < size()
//otherwise return false and leave value unchanged
bool Set::get(int pos, ItemType& value) const
{
```

```
    //If pos > size, there will be no value that satisfies the pos, return false
immediately
    if (pos < m_size && pos >= 0)
    {

        //The outer loop keeps the node to be compared
        for (Node* p = head->m_next; p != head; p = p->m_next)
        {
            //The count variable keeps track of how many times the current element
is greater than an element in the list
            int count = 0;

            //The inner loop compares each node to the current node
            //If the current node is greater than a node, count is incremented
            for (Node* q = head->m_next; q != head; q = q->m_next)
            {

                if (p->m_data > q->m_data)
                {
                    count++;
                }
            }

            //If count equals to pos, than our current node is greater than
exactly 'pos' number of nodes, return true
            if (count == pos)
            {
                value = p->m_data;
                return true;
            }
        }
    }

    return false;
}

//Swaps the other set with this one
void Set::swap(Set& other)
{
    //Swap size
    int tempSize = other.m_size;
    other.m_size = m_size;
    m_size = tempSize;

    //Swap head pointers
    Node* tempHead = other.head;
    other.head = head;
    head = tempHead;

    //Swap dummy
    Node* tempDummy = other.dummy;
    other.dummy = dummy;
    dummy = tempDummy;
}
```

```
//Dump function for debugging
void Set::dump() const
{
    for (Node* p = head->m_next; p != head; p = p->m_next)
    {
        cerr << p->m_data << endl;
    }
}

//Non member function implementations

//Sets result set to values from both s1 and s2, no duplicates
void unite(const Set& s1, const Set& s2, Set& result)
{
    int resultSize = result.size();
    ItemType curr;
    Set toRemove;

    //Check for elements in result set
    for (int i = 0; i < resultSize; i++)
    {
        result.get(i, curr); //grabs current element in result set
        if (!s1.contains(curr) && !s2.contains(curr)) //if current element in
neither s1 or s2, add to toRemove set; else do nothing
        {
            toRemove.insert(curr);
        }
    }

    //Remove unnecessary elements from result
    int removeSize = toRemove.size();
    for (int i = 0; i < removeSize; i++)
    {
        toRemove.get(i, curr);
        result.erase(curr);
    }

    //Insert remaining elements from s1 and s2 into result set
    int size_1 = s1.size();
    int size_2 = s2.size();

    //Iterate through set 1, grab element, insert into result
    for (int i = 0; i < size_1; i++)
    {
        s1.get(i, curr);
        result.insert(curr); //Insertion will be successful if element did not
already exist, else just does nothing
    }

    //Iterate through set 2, grab element, insert into result
    for (int i = 0; i < size_2; i++)
    {
        s2.get(i, curr);
```

```
        result.insert(curr);
    }
}

//Put elements that are in s1 but not in s2 into result set
void butNot(const Set& s1, const Set& s2, Set& result)
{
    int resultSize = result.size();
    ItemType curr;
    Set toRemove;

    //Check for elements in result set
    for (int i = 0; i < resultSize; i++)
    {
        result.get(i, curr); //grabs current element in result set
        if (!s1.contains(curr)) //if element is not in s1, put into toRemove
        {
            toRemove.insert(curr);
        }
        else if (s1.contains(curr))
        {
            if (s2.contains(curr)) //if element is in s1 AND s2, put into
toRemove; if not, do nothing
            {
                toRemove.insert(curr);
            }
        }
    }

    //Remove unnecessary elements from result
    int removeSize = toRemove.size();
    for (int i = 0; i < removeSize; i++)
    {
        toRemove.get(i, curr);
        result.erase(curr);
    }

    //Insert remaining elements from s1 into result set
    int size_1 = s1.size();

    //Iterate through set 1, grab element, insert into result
    for (int i = 0; i < size_1; i++)
    {
        s1.get(i, curr);
        //Insert ONLY if s2 does not contain current element
        if (!s2.contains(curr))
        {
            result.insert(curr); //Insertion will be successful if element did not
already exist, else just does nothing
        }
    }
}
```

SOLUTION

```
// 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 type alias.

using ItemType = std::string;

class Set
{
  public:
    Set();               // Create an empty set (i.e., one whoe size() is 0).
    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 is actually inserted.  Leave the set unchanged
      // and return false if value is not inserted (perhaps because it
      // was 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 it is 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 the item in the set that is
      // strictly greater than exactly i items 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
      //   If p and p->m_next point to nodes other than the dummy node,
      //      p->m_value > p->m_next->m_value

    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 insertBefore(Node* p, const ItemType& value);
      // Insert value in a new Node before Node p, incrementing m_size.

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

    Node* findFirstAtLeast(const ItemType& value) const;
      // Return pointer to first 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 butNot(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
{
    Node* p = findFirstAtLeast(value);
    return p != m_head  &&  p->m_value == value;
}

#endif // SET_INCLUDED

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

// Set.cpp

#include "Set.h"

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

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

    Node* p = findFirstAtLeast(value);
    if (p != m_head  &&  p->m_value == value)
        return false;

      // Insert new Node preserving ascending order and incrementing m_size

    insertBefore(p, value);
    return true;
}

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

    Node* p = findFirstAtLeast(value);
    if (p == m_head  ||  p->m_value != value)
        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.  Since the values are stored in
      // ascending order, the value at position i will be greater than
      // exactly i items in the set, meeting get's specification.
```

```
      // 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 before the dummy node that m_head points to
      // puts the new node at the end of the list.

    for (Node* p = other.m_head->m_next; p != other.m_head; p = p->m_next)
        insertBefore(m_head, 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::insertBefore(Node* p, const ItemType& value)
{
      // Create a new node

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

      // Insert new item before p

    newp->m_prev = p->m_prev;
    newp->m_next = p;
    newp->m_prev->m_next = newp;
    newp->m_next->m_prev = newp;

    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::findFirstAtLeast(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 butNot(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);
    }
}
```