```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
using namespace std;

template <class T> class LinkedList;

template <class T>
class Node {
private:
    T content;
    Node<T> *prev;
    Node<T> *next;
public:
    Node();
    Node( T value );
    ~Node();
    T getContent();
    void setContent( T value );
    Node<T>* getPrev();
    Node<T>* getNext();
    bool isHead();
    bool isTail();
    friend class LinkedList<T>;
};

template <class T>
class LinkedList {
private:
    Node<T> *head;
    Node<T> *tail;
    size_t length;
    bool reversed;
    string name;
public:
    LinkedList();
    ~LinkedList();
    Node<T>* getHead();
    Node<T>* getTail();
    size_t getLength();
    bool isOutputReversed();
    void setOutputReversed( bool value );
    string getName();
    void setName( string value );
    // Append value to the end of the list.
    void append( T value );
    void appendUnique( T value );
    void insertInOrder( T value, int (*cmp)( T a, T b ) );
    // Search value and return the pointer to its pos.
    Node<T>* search( T value );
    // Insert value after pos.
    void insert( T value, Node<T> *pos );
    // Remove node at pos.
    void remove( Node<T> *pos );
    void clear();
};

template <class T>
Node<T>::Node() {
    prev = next = NULL;
}

template <class T>
Node<T>::Node( T value ) {
    prev = next = NULL;
    setContent( value );
}

template <class T>
Node<T>::~Node()
{ }
```

```cpp
template <class T>
T Node<T>::getContent() {
    return content;
}

template <class T>
void Node<T>::setContent( T value ) {
    content = value;
}

template <class T>
Node<T>* Node<T>::getPrev() {
    return prev;
}

template <class T>
Node<T>* Node<T>::getNext() {
    return next;
}

template <class T>
bool Node<T>::isHead() {
    return prev == NULL;
}

template <class T>
bool Node<T>::isTail() {
    return next == NULL;
}

template <class T>
ostream& operator << ( ostream& stream, Node<T>& node ) {
    stream << node.getContent();
    return stream;
}

template <class T>
istream& operator >> ( istream stream, Node<T>& node ) {
    T value;
    stream >> value;
    node.setContent( value );
    return stream;
}

template <class T>
LinkedList<T>::LinkedList() {
    length = 0;
    reversed = false;
    Node<T> *h = new Node<T>();
    head = tail = h;
}

template <class T>
LinkedList<T>::~LinkedList() {
    clear();
    delete head;
}

template <class T>
void LinkedList<T>::clear() {
    while( tail != head ) {
        remove( tail );
    }
}

template <class T>
Node<T>* LinkedList<T>::getHead() {
    return head;
}

template <class T>
```

```cpp
Node<T>* LinkedList<T>::getTail()  {
    return tail;
}

template <class T>
size_t LinkedList<T>::getLength() {
    return length;
}

template <class T>
string LinkedList<T>::getName() {
    return name;
}

template <class T>
void LinkedList<T>::setName(string value) {
    name = value;
}

template <class T>
bool LinkedList<T>::isOutputReversed() {
    return reversed;
}

template <class T>
void LinkedList<T>::setOutputReversed( bool value ) {
    reversed = value;
}

template <class T>
void LinkedList<T>::append( T value ) {
    insert( value, tail );
}

template <class T>
void LinkedList<T>::appendUnique( T value ) {
    if( ! tail->getContent().compare(value) == 0 ) {
        insert( value, tail );
    }
}

template <class T>
void LinkedList<T>::insertInOrder( T value, int (*cmp)( T a, T b ) ) {
    Node<T>* curr = head->next;
    Node<T>* old = head;
    while( curr && cmp( curr->content, value ) < 0 ) {
        old = curr;
        curr = curr->next;
    }
    insert( value, old );
}

template <class T>
Node<T>* LinkedList<T>::search( T value ) {
    Node<T>* curr = head;
    while( curr != NULL ) {
        if( curr->content == value ) {
            return curr;
        }
        curr = curr->next;
    }
    return NULL;
}

template <class T>
void LinkedList<T>::insert( T value, Node<T> *pos ) {
    Node<T> *newNode = new Node<T>( value );
    newNode->prev = pos;
    newNode->next = pos->next;
    pos->next = newNode;
    if( tail == pos ) {
        tail = newNode;
    } else {
        newNode->next->prev = newNode;
    }
    length++;
}

template <class T>
void LinkedList<T>::remove( Node<T> *pos ) {
    if( pos == head ) {
        return;
    }
    pos->prev->next = pos->next;
    if( pos != tail ) {
        pos->next->prev = pos->prev;
    } else {
        tail = pos->prev;
    }
    delete pos;
    length--;
}

template <class T>
ostream& operator << ( ostream& stream, LinkedList<T>& llist ) {
    if( llist.getLength() == 0 ) {
        stream << endl;
        return stream;
    }
    if( llist.isOutputReversed() ) {
        Node<T> *curr = llist.getTail();
        while( curr != llist.getHead() ) {
            stream << *curr << endl;
            curr = curr->getPrev();
        }
    } else {
        Node<T> *curr = llist.getHead()->getNext();
        while( curr ) {
            stream << *curr << endl;
            curr = curr->getNext();
        }
    }
    return stream;
}

template <class T>
istream& operator >> ( istream& stream, LinkedList<T>& llist ) {
    T value;
    stream >> value;
    llist.append( value );
    return stream;
}

string preProcess( string word ) {
    string::iterator iter;
    stringstream ss;
    for( iter=word.begin(); iter!=word.end(); iter++ ) {
        if(! ispunct( *iter ) ) {
            ss << *iter;
        }
    }
    return ss.str();
}

int cmp( string a, string b ) {
    return a.compare(b);
}

void readToList( string file_name, LinkedList<string> *llist ) {
    ifstream fin( file_name.c_str() );
    string word;
    if( fin.is_open() ) {
        while( fin >> word ) {
            word = preProcess( word );
            llist->insertInOrder( word, cmp );
        }
    }
}

template <class T>
Node< LinkedList<T>* >* searchCatalog( LinkedList< LinkedList<T>* > &catalog, string name ) {
    Node< LinkedList<T>* > *curr = catalog.getHead()->getNext();
    while( curr ) {
        if( curr->getContent()->getName().compare( name ) == 0 ) {
            return curr;
        }
        curr = curr->getNext();
    }
    return NULL;
}

// Note that the input list is given in alphabetical order.
void Intersection( Node<string> *a, Node<string> *b, LinkedList<string> *c ) {
    if (a && b) {
        int cmp = a->getContent().compare( b->getContent() );
        if( cmp == 0 ) { // a == b
            c->appendUnique( a->getContent() );
            Intersection( a->getNext(), b->getNext(), c );
        } else if( cmp > 0 ) { // a > b
            Intersection( a, b->getNext(), c );
        } else { // a < b
            Intersection( a->getNext(), b, c );
        }
    }
}

// Note that the input list is given in alphabetical order.
void Union( Node<string> *a, Node<string> *b, LinkedList<string> *c ) {
    if (a && b) {
        int cmp = a->getContent().compare( b->getContent() );
        if( cmp == 0 ) { // a == b
            c->appendUnique( a->getContent() );
            Union( a->getNext(), b->getNext(), c );
        } else if( cmp > 0 ) { // a > b
            c->appendUnique( b->getContent() );
            Union( a, b->getNext(), c );
        } else { // a < b
            c->appendUnique( a->getContent() );
            Union( a->getNext(), b, c );
        }
    } else {
        while ( a ) {
            c->appendUnique( a->getContent() );
            a = a->getNext();
        }
        while ( b ) {
            c->appendUnique( b->getContent() );
            b = b->getNext();
        }
    }
}

int main(int argc, char **argv) {
    if( argc != 2 ) {
        return -1;
    }
    char *file_name = argv[1];
    string command, par_1, par_2, par_3;
    ifstream fin( file_name );
    LinkedList< LinkedList<string>* > catalog;
    Node< LinkedList<string>* > *catalogEntry;
    LinkedList<string> *llist = NULL;
    if( fin.is_open() ) {
        while( fin >> command ) {
            if( command == "Read" ) {
                fin >> par_1 >> par_2; // file name and list name.
                llist = new LinkedList<string>();
                llist->setName( par_2 );
                readToList( par_1, llist );
                catalogEntry = searchCatalog( catalog, par_2 );
                if( catalogEntry != NULL ) {
                    catalog.remove( catalogEntry );
                }
                catalog.append( llist );

            } else if ( command == "Print" ) {
                fin >> par_1 >> par_2; // list name and forward or backward
                catalogEntry = searchCatalog( catalog, par_1 );
                if( catalogEntry == NULL ) {
                    continue;
                }
                llist = catalogEntry->getContent();
                if( par_2 == "forward" ) {
                    llist->setOutputReversed( false );
                } else if( par_2 == "backward" ) {
                    llist->setOutputReversed( true );
                }
                cout << *llist;

            } else if ( command == "Intersection" ) {
                fin >> par_1 >> par_2 >> par_3;
                llist = new LinkedList<string>();
                llist->setName( par_3 );
                Node< LinkedList<string> *> *a, *b;
                a = searchCatalog( catalog, par_1 );
                b = searchCatalog( catalog, par_2 );
                if( a && b ) {
                    Intersection( a->getContent()->getHead()->getNext(),
b->getContent()->getHead()->getNext(), llist );
                    catalogEntry = searchCatalog( catalog, par_3 );
// If there is already a list with the specified name, overwrite it with the new list.
                    if( catalogEntry != NULL ) {
                        catalog.remove( catalogEntry );
                    }
                    catalog.append( llist );
                }

            } else if ( command == "Union" ) {
                fin >> par_1 >> par_2 >> par_3;
                llist = new LinkedList<string>();
                llist->setName( par_3 );
                Node< LinkedList<string> *> *a, *b;
                a = searchCatalog( catalog, par_1 );
                b = searchCatalog( catalog, par_2 );
                if( a && b ) {
                    Union( a->getContent()->getHead()->getNext(),
b->getContent()->getHead()->getNext(), llist );
                    catalogEntry = searchCatalog( catalog, par_3 );
// If there is already a list with the specified name, overwrite it with the new list.
                    if( catalogEntry != NULL ) {
                        catalog.remove( catalogEntry );
                    }
                    catalog.append( llist );
                }
            }
        }
    }
    return 0;
}
```