```
#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;
public:
    LinkedList();
    ~LinkedList();
    Node<T>* getHead();
    Node<T>* getTail();
    size_t getLength();
    bool isOutputReversed();
    void setOutputReversed( bool value );
    // Append value to the end of the list.
    void append( T value );
    // 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 );
};

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()
{ }

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() {
    while( tail != head ) {
        remove( tail );
    }
    delete head;
}

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

template <class T>
Node<T>* LinkedList<T>::getTail()  {
    return tail;
}

template <class T>
```

```cpp
size_t LinkedList<T>::getLength() {
    return length;
}

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>
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.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();
}

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.append( word );
        }
    }
}

int main(int argc, char **argv) {
    if( argc != 2 ) {
        return -1;
    }
    char *file_name = argv[1];
    string command, par_1, par_2;
    ifstream fin( file_name );
    if( fin.is_open() ) {
        LinkedList<string> llist;
        while( fin >> command ) {
            if( command == "Read" ) {
                fin >> par_1; // file name;
                readToList( par_1, llist );

            } else if ( command == "Print" ) {
                fin >> par_1; // forward or backward
                if( par_1 == "forward" ) {
                    llist.setOutputReversed( false );
                } else if( par_1 == "backward" ) {
                    llist.setOutputReversed( true );
                }
                cout << llist;

            } else if ( command == "Insert" ) {
                fin >> par_1 >> par_2; // insert par_1 after par_2
                Node<string> *pos = llist.search( par_2 );
                if( pos != NULL ) {
                    llist.insert( par_1, pos );
                }
```

```cpp
            } else if ( command == "Delete" ) {
                fin >> par_1; // word to delete
                Node<string> *pos = llist.search( par_1 );
                if( pos != NULL ) {
                    llist.remove( pos );
                }
            }
        }
    }
    return 0;
}
```