```cpp
//lab singlelink problem

//singlelink.cpp
#include<iostream>
using std::ostream; using std::cout; using std::endl;
#include<sstream>
using std::ostringstream;
#include<stdexcept>
using std::out_of_range;
#include<string>
using std::string;
#include <algorithm>
using std::swap;

#include "singlelink.hpp"


ostream& operator<<(ostream & out, Node const & n){
  out << n.data_<<", ";
  return out;
}


SingleLink::SingleLink() : head_(nullptr), tail_(nullptr) {};

ostream& operator<<(ostream & out, SingleLink const & sl){
  // out << "List:";
  ostringstream oss;
  for(auto itr=sl.head_; itr !=nullptr; itr=itr->next_)
    oss <<itr->data_<<", ";
  string s = oss.str();
  s = s.substr(0,s.size() - 2);
  out << s;
  return out;
};


SingleLink::SingleLink(int dat){
  head_ = new Node(dat);
  tail_ = head_;
}

// append node with data dat to the end of the list
// fast because of the tail_ pointer
void SingleLink::append_back(int dat){
  Node *node = new Node(dat);
  if (tail_ != nullptr){
    tail_->next_ = node;
    tail_ = node;
  } else {
    head_= node;
    tail_= node;
  }
}

bool SingleLink::del(int val){
  Node *current_to_check;
  Node *previous;
  current_to_check = head_;
  if (head_ == nullptr){
    return false;
  }

  if (head_->data_ == val){
    head_ = head_->next_;
    delete current_to_check;
```

```
    return true;
  }
  previous = head_;
  current_to_check = head_->next_;
  while(current_to_check != nullptr){
    if (current_to_check->data_ == val){
      previous->next_ = current_to_check->next_;
      delete current_to_check;
      return true;
    }
    previous = current_to_check;
    current_to_check = current_to_check->next_;
  }
  return false;
}


Node & SingleLink::operator[](size_t indx){

  size_t cnt = 0;
  auto itr = head_;
  while (itr != nullptr) {
    if (cnt == indx)
      return *itr;
    itr = itr->next_;
    cnt++;
  }
  throw out_of_range("Index not found");
}

/*
bit of work. we need to remember a pointer that walks
down sl, and tail_ walks down the new list.
make a new node (copy the current sl node), update
*/
SingleLink::SingleLink(const SingleLink& sl){
  if (sl.head_ == nullptr){
    head_ = nullptr;
    tail_ = nullptr;
  } else {
    head_ = new Node(sl.head_->data_);
    tail_ = head_;
    Node* sl_ptr = sl.head_->next_;
    Node* new_node;
    while (sl_ptr != nullptr){
      new_node = new Node(sl_ptr->data_);
      tail_->next_ = new_node;
      sl_ptr = sl_ptr->next_;
      tail_ = new_node;
    }
  }
}


SingleLink& SingleLink::operator=(SingleLink sl){
  // x = y
  // x.operator=(y);
  swap(head_, sl.head_);
  swap(tail_, sl.tail_);
  return *this;
}


// walk down the list, moving head_ but remember it in to_del
// delete each node in turn, the set head_ and tail_
SingleLink::~SingleLink(){
```

```
  Node* to_del = head_;
  while (to_del != nullptr){
    head_ = head_->next_;
    delete to_del;
    to_del = head_;
  }
  head_ = nullptr;
  tail_ = nullptr;
}

// singlelink.hpp
#pragma once

#include <iostream>

struct Node {
    public:
        int data_;
        Node *next_;

        Node() : data_(0), next_(nullptr) {};
        Node(int d) : data_(d), next_(nullptr) {};
};

class SingleLink {
    private:
        Node *head_;
        Node *tail_;

    public:
        SingleLink();
        SingleLink(int dat);
        void append_back(int);

        friend std::ostream& operator<<(std::ostream &out, const SingleLink &s);
        bool del(int val);
        Node& operator[](size_t index);

        // Rule of three stuff
        // ~SingleLink();
        // SingleLink(const SingleLink &);
        // SingleLink& operator=(SingleLink);
};

// main.cpp
#include <iostream>
#include "singlelink.hpp"

int main() {
    SingleLink s;
    s.append_back(3);
    s.append_back(4);
    std::cout << "Two Items: " << s << std::endl;

    SingleLink s2(10);
    s2.append_back(3);
    s2.append_back(4);
    std::cout << "Three Items: " << s2 << std::endl;

    s2.del(3);
    s2.del(4);
    s2.del(10);
    std::cout << "Removed 3: " << s2 << std::endl;


    SingleLink s3(56);
```

```cpp
    s3.append_back(73);
    s3.append_back(345);
    s3.append_back(1);
    s3.append_back(15);
    std::cout << "Indexing 0: " << s3[0].data_ << std::endl;
    std::cout << "Indexing 1: " << s3[1].data_ << std::endl;
    std::cout << "Indexing 2: " << s3[2].data_ << std::endl;
    std::cout << "Indexing 3: " << s3[3].data_ << std::endl;
    std::cout << "Indexing 4: " << s3[4].data_ << std::endl;
}


// more linked list types
#include <iostream>
#include <cassert>
#include <stdexcept>

class Node {
public:
    int data_;
    Node* next_;

    Node(int value) : data_(value), next_(nullptr) {}
};

class SingleLink {
private:
    Node* head_;

public:
    SingleLink() : head_(nullptr) {}

    // 1. PrintList
    void PrintList() {
        Node* curr = head_;
        while (curr != nullptr) {
            std::cout << curr->data_ << " -> ";
            curr = curr->next_;
        }
        std::cout << "nullptr" << std::endl;
    }

    // 2. GetLength
    int GetLength() {
        int length = 0;
        Node* curr = head_;
        while (curr != nullptr) {
            length++;
            curr = curr->next_;
        }
        return length;
    }

    // 3. FindNode
    Node* FindNode(int value) {
        Node* curr = head_;
        while (curr != nullptr) {
            if (curr->data_ == value) {
                return curr;
            }
            curr = curr->next_;
        }
        return nullptr;
    }

    // 4. IsEmpty
```

```
    bool IsEmpty() {
        return head_ == nullptr;
    }

    // 5. Append
    void Append(int value) {
        Node* newNode = new Node(value);
        if (head_ == nullptr) {
            head_ = newNode;
            return;
        }
        Node* curr = head_;
        while (curr->next_ != nullptr) {
            curr = curr->next_;
        }
        curr->next_ = newNode;
    }

    // 6. Prepend
    void Prepend(int value) {
        Node* newNode = new Node(value);
        newNode->next_ = head_;
        head_ = newNode;
    }

    // 7. DeleteHead
    void DeleteHead() {
        if (head_ == nullptr) return;
        Node* temp = head_;
        head_ = head_->next_;
        delete temp;
    }

    // 8. DeleteTail
    void DeleteTail() {
        if (head_ == nullptr) return;
        if (head_->next_ == nullptr) {
            delete head_;
            head_ = nullptr;
            return;
        }
        Node* curr = head_;
        while (curr->next_->next_ != nullptr) {
            curr = curr->next_;
        }
        delete curr->next_;
        curr->next_ = nullptr;
    }

    // 9. Reverse
    void Reverse() {
        Node* prev = nullptr;
        Node* curr = head_;
        while (curr != nullptr) {
            Node* next = curr->next_;
            curr->next_ = prev;
            prev = curr;
            curr = next;
        }
        head_ = prev;
    }

    // 10. Merge
    void Merge(SingleLink& other) {
        if (head_ == nullptr) {
            head_ = other.head_;
```

```
            other.head_ = nullptr;
            return;
        }
        Node* curr = head_;
        while (curr->next_ != nullptr) {
            curr = curr->next_;
        }
        curr->next_ = other.head_;
        other.head_ = nullptr;
    }

    // 11. FindKthFromEnd
    Node* FindKthFromEnd(int k) {
        if (k <= 0) return nullptr;
        Node* fast = head_;
        Node* slow = head_;
        for (int i = 0; i < k; i++) {
            if (fast == nullptr) return nullptr;
            fast = fast->next_;
        }
        while (fast != nullptr) {
            fast = fast->next_;
            slow = slow->next_;
        }
        return slow;
    }

    // 12. Sort
    void Sort() {
        if (head_ == nullptr || head_->next_ == nullptr) return;
        for (Node* i = head_; i != nullptr; i = i->next_) {
            for (Node* j = i->next_; j != nullptr; j = j->next_) {
                if (i->data_ > j->data_) {
                    std::swap(i->data_, j->data_);
                }
            }
        }
    }
};

// Test cases
int main() {
    SingleLink sl;

    // Append and print
    sl.Append(1);
    sl.Append(2);
    sl.Append(3);
    sl.PrintList(); // 1 -> 2 -> 3 -> nullptr

    // Prepend and print
    sl.Prepend(0);
    sl.PrintList(); // 0 -> 1 -> 2 -> 3 -> nullptr

    // GetLength
    std::cout << "Length: " << sl.GetLength() << std::endl; // 4

    // FindNode
    Node* found = sl.FindNode(2);
    if (found) std::cout << "Found: " << found->data_ << std::endl; // 2

    // IsEmpty
    std::cout << "Is Empty: " << sl.IsEmpty() << std::endl; // 0

    // DeleteHead
    sl.DeleteHead();
```

```
    sl.PrintList(); // 1 -> 2 -> 3 -> nullptr

    // DeleteTail
    sl.DeleteTail();
    sl.PrintList(); // 1 -> 2 -> nullptr

    // Reverse
    sl.Reverse();
    sl.PrintList(); // 2 -> 1 -> nullptr

    // Merge
    SingleLink sl2;
    sl2.Append(3);
    sl2.Append(4);
    sl.Merge(sl2);
    sl.PrintList(); // 2 -> 1 -> 3 -> 4 -> nullptr

    // FindKthFromEnd
    Node* kth = sl.FindKthFromEnd(2);
    if (kth) std::cout << "2nd from end: " << kth->data_ << std::endl; // 3

    // Sort
    sl.Sort();
    sl.PrintList(); // 1 -> 2 -> 3 -> 4 -> nullptr

    return 0;
}







// more practive:

// type1: flip the last n elements of a linked list
void SingleLink::ReverseLastN(int n) {
    // Step 1: Handle edge cases
    if (n <= 1 || head_ == nullptr) return;

    // Step 2: Find the length of the list
    int length = 0;
    Node* curr = head_;
    while (curr != nullptr) {
        length++;
        curr = curr->next_;
    }

    // Step 3: If n >= length, reverse the entire list
    if (n >= length) {
        ReverseList();
        return;
    }

    // Step 4: Traverse to the (length - n)th node
    Node* prev = nullptr;
    Node* start = head_;
    for (int i = 0; i < length - n; i++) {
        prev = start;
        start = start->next_;
    }

    // Step 5: Reverse the last n nodes
    Node* reversedHead = ReverseSublist(start);
```

```cpp
    // Step 6: Connect reversed part back to the main list
    if (prev != nullptr) {
        prev->next_ = reversedHead;
    } else {
        head_ = reversedHead;
    }
}

// type2: reverse the linked list
void SingleLink::ReverseList() {
    Node* prev = nullptr;
    Node* curr = head_;
    Node* next = nullptr;

    while (curr != nullptr) {
        next = curr->next_;
        curr->next_ = prev;
        prev = curr;
        curr = next;
    }

    head_ = prev;
}

// type3: delete node which has value equal to target
void SingleLink::DeleteNode(int value) {
    // 检查链表是否为空
    if (head_ == nullptr) {
        return; // 链表为空，直接返回
    }

    // 特殊情况：删除头节点
    if (head_->data_ == value) {
        Node* temp = head_;
        head_ = head_->next_;
        delete temp; // 释放旧头节点的内存
        return;
    }

    // 遍历链表找到要删除的节点
    Node* prev = nullptr; // 前一个节点
    Node* curr = head_;   // 当前节点
    while (curr != nullptr && curr->data_ != value) {
        prev = curr;
        curr = curr->next_;
    }

    // 如果找到了值为 `value` 的节点
    if (curr != nullptr) {
        prev->next_ = curr->next_; // 跳过当前节点
        delete curr;               // 释放当前节点的内存
    }
}
```