UCLA Computer Science Department TA: Kung-Hua Chang

Student Name and ID _____

CS 32, WINTER 2015, PRACTICE MIDTERM II.

Problem #	Maximal Possible Points	Received
1	10	
2.1	10	
2.2	10	
2.3	10	
2.4	10	
3	10	
4	5	
Total	65	

Problem #1: Stack is a LIFO (Last In First Out) container while Queue is a FIFO (First In First Out) container. If you were given 2 sequence of numbers in which the first sequence of numbers are the numbers entering a mystery container, and the second sequence of numbers are the numbers leaving a container, are you able to determine whether that mystery container is a Stack, Queue, Might Be Either One, or Neither?

For example, if you were given these two sequences:

1 2 3 // 1 entered the container first, then 2, and finally 3.

3 2 1 // 3 left the container first, then 2, and finally 1.

then that mystery container must be a Stack.

Implement the functions on the next page (which are called by the main routine on the page after that) so that this input:

10	*the number of values in test case #1*
1 2 3 4 5 6 7 8 9 10	*the values entered into the container for test #1*
10 9 8 7 6 5 4 3 2 1	*the values leaving the container for test #1*
5	*the number of values in test case #2*
1 2 3 4 5	*etc.*
1 2 3 4 5	
7	
1 2 3 4 3 2 1	
1 2 3 4 3 2 1	
4	
1 3 4 2	
1 4 2 3	

produces this output:

This is a Stack!

This is a Queue!

Either a Stack or a Queue!

Neither a Stack nor a Queue!

```
#include <iostream>
#include <vector>
using namespace std;

bool isQueue(const vector<int>& seq1, const vector<int>& seq2)
{









}


bool isStack(const vector<int>& seq1, const vector<int>& seq2)
{









}
```

```
int main()
{
	int numberOfValues;
	while( cin >> numberOfValues )    // leave loop if there is no next test case
	{
		vector<int> v1, v2;
		for (int i = 0; i < numberOfValues; i++) {
			int value;
			cin >> value;
			v1.push_back(value);
		}
		for (int i = 0; i < numberOfValues; i++) {
			int value;
			cin >> value;
			v2.push_back(value);
		}
		bool s = isStack(v1, v2);
		bool q = isQueue(v1, v2);
		if (s) {
			if (q)
				cout << "Either a Stack or a Queue!" << endl;
			else
				cout << "This is a Stack!" << endl;
		} else {
			if (q)
				cout << "This is a Queue!" << endl;
			else
				cout << "Neither a Stack nor a Queue!" << endl;
	}
}
```

Problem #2: Below is an implementation of a singly linked list with no dummy node.

```
#include <iostream>
using namespace std;

class LinkedList
{
public:
	LinkedList(): head(nullptr) { }
	~LinkedList();
	void append(int value);		// append value to the list
	void print() const;		// show the items in the list
	void printReverse() const;	// show the items in the list in the opposite order
	void reverse();			// change list so items are in the opposite order
	int sum() const;		// return the sum of the values in the list
private:
	struct Node
	{
		int num;
		Node* next;
	};
	Node* head;    // this is the only data member; do not add any others

	void printReverseHelper(const Node* p) const;
	Node* reverseHelper(Node* current, Node* previous);
	Int sumHelper(const Node* p) const;
	void removeNodes(Node* p);
};

int main()
{
	LinkedList list;
	cout << list.sum() << endl;		// writes 0
	int values[4] = { 30, 10, 40, 20 };
	for (int i = 1; i <= 4; i++)
		list.addToList(values[i]);
	list.print();				// writes	30 10 40 20
	cout << list.sum() << endl;		// writes	100
	list.printReverse();			// writes	20 40 10 30
	list.print();				// writes	30 10 40 20 (list wasn't changed)
	list.reverse();				// this changes the list
	list.print();				// writes	20 40 10 30
}
```

```
void LinkedList::append(int value)
{
	Node* current = new Node;
	current->num = value;
	current->next = nullptr;
	if (head == nullptr)
		head = current;
	else {
		Node* ptr = head;
		while (ptr->next != nullptr)
			ptr = ptr->next;
		ptr->next = current;
	}
}

void LinkedList::print() const
{
	for (const Node* ptr = head; ptr != nullptr; ptr = ptr->next)
		cout << ptr->num << " ";
	cout << endl;
}
```

Problem #2.1: Please fill in the following blanks to complete the implementations of printReverse() and printReverseHelper() to print the list elements in reverse order **recursively**.

```
void LinkedList::printReverse() const
{
	printReverseHelper( _____ );
	cout << endl;
}
void LinkedList::printReverseHelper(const Node* p) const
{
	if ( _____ )
		return ;

	printReverseHelper( _____ );

	cout << _____ << " ";
}
```

Problem #2.2: Please fill in the following blanks to complete the implementations of reverse() and reverseHelper() to reverse the linked list **recursively**.

```
void LinkedList::reverse()
{
    head = reverseHelper(head, _____ );
}

LinkedList::Node* LinkedList::reverseHelper(Node* current, Node* previous)
{
    if ( _____ )
        return previous;
    Node* last_node = reverseHelper(current->next, _____ );

    current->next = _____ ;

    return _____ ;
}
LinkedList::~LinkedList()
{
    removeNodes(head);
}
```

Problem #2.3: Please fill in the following blanks to complete the implementation of sumHelper() to compute the sum of the values in the list **recursively**.

```
int LinkedList::sum() const
{
    return sumHelper(head);
}

int LinkedList::sumHelper(const Node* p) const
{




}
```

Problem #2.4: Please fill in the following blanks to complete the implementation of removeNodes() to correctly implement the destructor **recursively**.

```
LinkedList::~LinkedList()
{
        removeNodes(head);
}

void LinkedList::removeNodes(Node* p)
{









}
```

Problem #3: Please fill in the missing blanks and code blocks below so that the output of this program is:
Base Homer Simpson
Derived Doctor Beverly Crusher
Derived Doctor Who

```
#include <iostream>
#include <vector>
#include <string>
using namespace std;

class Base {
public:
	Base (string nm): name(nm) {}
	string getName() const { return name; }
	virtual void printName() const { cout << "Base " << name << endl; }
private:
	string name;
};

class Derived : public Base
{
public:
	Derived (string nm): Base( _____ ) { }
	virtual void printName() { cout << "Derived " << getName() << endl; }
};

void printAll(const vector<Base*>& vec)
{
	for (int i = 0; i != vec.size(); i++)    // fill in the body of the loop




};

int main() {
	vector<Base*> v;
	v.push_back(new Base("Homer Simpson"));
	v.push_back(new Derived("Beverly Crusher"));
	v.push_back(new Derived("Who"));
	printAll(v);
	for (int i = 0; i < 3; i++)

		delete _____ ;
}
```

Problem #4: You and your friends got caught by 200 cannibals (again!?). After you fought bravely for your lives, the cannibals agree that they will stop attacking you if you can answer this question: Given unlimited water and any 2 jugs (jug A and jug B with different capacity), is it possible to come up with X gallons of water in either of the jugs in 9 steps? For example, if jug A has a 3 gallon capacity and jug B can hold 5 gallons, is it possible to come up with exactly 4 gallons of water (in jug B, of course, since jug A can hold only 3 gallons)? Although you suspect that these cannibals have watched "Die Hard 3" before, you decided to write a program to solve it first. Fortunately, you have your old friend, your laptop, with you with the following code fragments. Now you just need to complete the code fragments and you will be free.

Sample Input:
3 5 4
7 10 2
Sample Output:
This can be solved!
This cannot be solved within 9 steps!

If you do not know why 3 5 4 is doable within 9 steps, the steps are as below:

Fill B	// A: 0	B: 5	
Pour from B to A	// A: 3	B: 2	
Empty A	// A: 0	B: 2	
Pour from B to A	// A: 2	B: 0	
Fill B	// A: 2	B: 5	
Pour from B to A	// A: 3	B: 4	← Jug B now has 4 gallons.

```
#include <iostream>
#include <algorithm> // defines int min(int a, int b); returns the minimum of a and b
using namespace std;

bool isDoable(int jug1, int cap1, int jug2, int cap2, int target, int depth);

int main()
{
	int jug1_capacity, jug2_capacity, target;
	cin >> jug1_capacity >> jug2_capacity >> target;
	if (isDoable(0, jug1_capacity, 0, jug2_capacity, target, 0))
		cout << "This can be solved!" << endl;
	else
		cout << "This cannot be solved within 9 steps!" << endl;
}
```

```
bool isDoable(int jug1, int cap1, int jug2, int cap2, int target, int depth)
{
    if (jug1 == target || jug2 == target)
        return true;
    if (depth == 9)    // our limit on the depth of a recursion
        return false;

    // Can you solve it by filling A first?
    if (isDoable(cap1, cap1, jug2, cap2, target, depth+1))
        return true;

    // Can you solve it by filling B first?

    if ( _____ )
        return true;

    // Can you solve it by emptying A first?
    if (isDoable(0, cap1, jug2, cap2, target, depth+1))
        return true;

    // Can you solve it by emptying B first?

    if ( _____ )
        return true;

    // Can you solve it by pouring from B to A first?
    int amt = min(cap1 – jug1, jug2);    // unused capacity in A, or all of what's in B
    if (isDoable(jug1 + amt, cap1, jug2 - amt, cap2, target,depth+1))
        return true;

    // Can you solve it by pouring from A to B first? [fill in code below]




    // Nothing leads to a solution

    _____ ;
}
```