```
 1: //Name: Ghazi Najeeb Al-Abbar
 2: //ID: 2181148914
 3: //problem-1.cpp
 4:
 5: //Stacks, Queues, and LinkedLists implemented using nodes
 6: //Let n be the number of nodes
 7:
 8: #include <iostream>
 9: using namespace std;
10:
11: class LinkedList{
12:
13:   public:
14:
15:   //Constructor
16:   LinkedList()
17:   {
18:     this->next = nullptr; //makes the next node after the newly constructed node point to not
19:   }
20:
21:   //Returns true if the list is empty
22:   //Time Complexity: O(1) because there is only one command
23:   bool isEmpty(){ return first == nullptr;}
24:
25:   //Adds a node to the tail of the list
26:   //Time Complexity: O(n) because the size of the list is variable, and the loop iterat
27:   void add(int Data){
28:
29:       LinkedList* newPtr = new LinkedList(); //creates a new node
30:       newPtr->data = Data;//Makes the node data equal to the user input data
31:
32:       LinkedList* temp = first; //temporary node to hold the first node
33:
34:       //Checks if the list is empty
35:       //if that's the case, then the the first node would equal newPtr
36:       if (isEmpty())
37:       {
38:        first = newPtr;
39:
40:        cout << Data << " was added!\n";
41:
42:        return; //leaves function
43:       }
44:
45:       //The case where the list is not empty
46:       //The loop goes through the list just until the next node points to nothing
47:       while (temp->next != nullptr)
48:         temp = temp->next; //goes to the next node
49:
50:       temp->next = newPtr; //The new node is added after last node
51:
52:       cout << Data << " was added!\n";
53:     }
54:
55:     //Looks for a node with data x to remove. Prints an error message if there is no node
```

```
56:     //Time Complexity: O(n) because the loop iterates at least n times
57:     void Remove(int x){
58:
59:         //Checks if the list is empty
60:         if (isEmpty()){
61:             cout << "There are no elements in this Linked List!\n";
62:             return; //leaves the function if the list is empty
63:         }
64:
65:         //Checks if the first node is x
66:         if (first->data == x)
67:         {
68:             //checks if the first node is the only node
69:             if (first->next == nullptr){
70:                 first = nullptr; //points the first node no nothing
71:
72:                 cout << x << " was removed from the linked List.\n";
73:
74:                 return; //leaves the function
75:             }
76:
77:             //checks if the first node is not the only node
78:             else{
79:
80:                 first = first->next; //removes the first node and makes the second node the f
81:
82:                 cout << x << " was removed from the linked List.\n";
83:
84:                 return; //leaves function
85:             }
86:         }
87:
88:         bool isElement = false;//boolean value that is false when x is not a data for a node,
89:         LinkedList* temp = first;//temporary node to hold the first node
90:
91:         //goes through the list and checks if x is a data
92:         while(temp->next != nullptr){
93:
94:             //checks if the data of the next node is x
95:             //if it is the case, then the loop breaks
96:             if (temp->next->data == x){
97:
98:                 isElement = true;
99:
100:                break;
101:            }
102:
103:            temp = temp->next; //goes to the next node
104:        }
105:
106:        //checks if isElement is true
107:        if (isElement){
108:
109:            temp->next = temp->next->next; //Makes the next node equal to the node next to th
110:
```

```
111:             cout << x << " was removed from the linked List.\n";
112:
113:             return; //leaves the function
114:         }
115:
116:         //The case where x is not in the List
117:         //prints a message telling the user that the element does not exist
118:         cout << "There is no element with data " << x << "!\n";
119:     }
120:
121:     //returns true if x is the data of one of the elements. False if it is not
122:     //Time Complexity: O(n) because the loop iterates at least n times
123:     bool search(int x){
124:
125:         LinkedList* temp = first; //temporary node to hold the first node
126:
127:         //goes through the list and checks whether x is in the list or not
128:         while (temp != nullptr){
129:
130:             //checks if x is the data of the current node
131:             if (temp->data == x)
132:                 return true; //returns true if x is the current data
133:
134:             temp = temp->next; //goes to next node
135:         }
136:
137:         //if x is not in the list, then return false
138:         return false;
139:     }
140:
141:     //Returns the number of nodes in the list. Will return 0 if the list is empty
142:     //Time Complexity: O(n) because the loop iterates at least n times
143:     int Size(){
144:
145:         LinkedList* temp = first; //temporary node to hold the first node
146:         int count = 0; //the counter starts at 0
147:
148:         //goes through the list and counts each element
149:         while (temp != nullptr){
150:
151:             count++; //increments the counter
152:             temp = temp->next; //goes to the next node
153:         }
154:
155:         return count; //returns the size
156:     }
157:
158:        /*Print function*/
159:     void print(){
160:
161:         if (isEmpty()){
162:             cout << "Linked list is empty!\n";
163:             return;
164:         }
165:
```

```
166:         LinkedList* temp = first;
167:
168:         while (temp != nullptr){
169:
170:             if (temp->next == nullptr)
171:                 cout << temp->data << endl;
172:             else
173:                 cout << temp->data << " -->";
174:
175:             temp = temp->next;
176:         }
177:     }
178:
179:   private: //encapsulated attributes
180:
181:   LinkedList* first = nullptr; //first node
182:   LinkedList* next; //Next node. it could be a nullptr
183:   int data; //The data stored in the node
184: };
185:
186: class Stack{ //first in, last out
187:
188: public:
189:
190:     //Constructor
191:     Stack()
192:     {
193:         this->next = nullptr; //The newest node always pointing to nothing
194:     }
195:
196:     //Checks whether the stack is empty or not (Returns true if it is empty. False otherw
197:     //Time Complexity: O(1) because there is only one command
198:     bool isEmpty(){return first == nullptr;}
199:
200:     //Adds a new node with data x to the top of the stack
201:     //Time Complexity: O(n) because the loop iterates at least n times
202:     void push(int x){
203:
204:         //Creates a new node with data x
205:         Stack* newStack = new Stack();
206:         newStack->data = x;
207:
208:         //Checks  whether the stack is empty or not
209:         if (isEmpty()){
210:
211:             first = newStack; //makes the first node equal to the new node
212:             //Since there is only one node, then the first node is the last node
213:             last = first;
214:
215:             cout << x << " was pushed!\n";
216:
217:             return; //Leaves function
218:         }
219:
220:         Stack* temp = first; //Temporary node to hold the first node
```

```
221:
222:         //Goes through the stack until it reaches the end
223:         while (temp->next != nullptr)
224:             temp = temp->next; //Goes to the next node
225:
226:         //Makes the new node into the last node
227:         temp->next = newStack;
228:         last = temp->next;
229:
230:         cout << x << " was pushed!\n";
231:     }
232:
233:     //Removes the last element pushed into the stack
234:     //Time Complexity: O(n) because the loop iterates at least n times
235:     int pop(){
236:
237:         int removed; //variable stores popped data
238:
239:         //Checks whether the stack is empty or not
240:         if (isEmpty()){
241:
242:             cout << "There are no elements to be popped!\n";
243:         }
244:
245:         //Checks if the first node is the only node
246:         else if (first->next == nullptr){
247:
248:             cout << first->data << " was removed.\n";
249:
250:             removed = first->data;
251:
252:             //If the first node was the only node, and it points to nothing
253:             //Then the last node should point to nothing as well
254:             first = nullptr;
255:             last = nullptr;
256:
257:             return removed; //return the removed data
258:         }
259:
260:         Stack* temp = first; //Creates temporary node to store the first node
261:         removed = last->data;
262:
263:         //Goes through the stack until it reaches the node before the last
264:         while (temp->next != last)
265:             temp = temp->next; //goes to the next node
266:
267:         //Since temp is the node before the last, then the last node should point to nothing
268:         //and temp should be the new last node
269:         temp->next = nullptr;
270:         last = temp;
271:
272:         return removed; //return the removed data
273:     }
274:
275:     //Returns the data of the last node in the stack, or the the top of the stack
```

```
276:     //Time Complexity: O(1) because there are a constant amount of instructions
277:     int top(){
278:
279:         //Checks if the stack is empty
280:         if (isEmpty()){
281:
282:             cout << "The stack is empty!\n";
283:
284:             return 0; //Returns 0 or nothing if the stack is empty
285:         }
286:
287:         //If the stack is not empty, then the data of the last node will be returned
288:         return last->data;
289:     }
290:
291:     //Returns the size of the Stack
292:     //Time Complexity: O(n) because the loop iterates at least n times
293:     int Size(){
294:
295:         //If the stack is empty, then the size is zero (returns 0)
296:         if (isEmpty())
297:             return 0;
298:
299:         Stack* temp = first; //temporary node to hold the first node
300:         int count = 0; //counter for the size
301:
302:         //Goes through the Stack and counts how many nodes are there
303:         while (temp != nullptr)
304:         {
305:             temp = temp->next; //Goes to next node
306:             count++; //increments the counter
307:         }
308:
309:         return count; //Returns counter (the size of the Stack)
310:     }
311:
312:            /*Print function*/
313:     void print(){
314:
315:         if (isEmpty()){
316:             cout << "Linked list is empty!\n";
317:             return;
318:         }
319:
320:         Stack* temp = first;
321:
322:         while (temp != nullptr){
323:
324:             if (temp->next == nullptr)
325:                 cout << temp->data << endl;
326:             else
327:                 cout << temp->data << " -->";
328:
329:             temp = temp->next;
330:         }
```

```
331:     }
332:
333: private:
334:
335:     Stack* first = nullptr; //Bottom node
336:     Stack* last = nullptr; //Top node
337:     Stack* next;
338:     int data;
339: };
340:
341: class Queue{ //first in, first out
342: public:
343:
344:     //Constructor
345:     Queue()
346:     {
347:         this->next = nullptr; //Newest pointer always points to nothing
348:     }
349:
350:     //Checks whether the stack is empty or not (Returns true if it is empty. False otherw
351:     //Time Complexity: O(1) because there is only one command
352:     bool isEmpty(){return first == nullptr;}
353:
354:     //Pushes a new node with data x to the end of the Queue
355:     //Time Complexity: O(n) because the loop iterates at least n times
356:     void enqueue(int x){
357:
358:         Queue* newQ = new Queue(); //Creates new node
359:         newQ->data = x; //new node has data equal to x
360:
361:         //Checks if the queue is empty
362:         if (isEmpty()){
363:
364:             first = newQ; //makes the first node equal to the new node
365:             //Since there is only one node, then the first node is the last node
366:             last = first;
367:
368:             cout << x << " Was enqueued!\n";
369:
370:             return; //Leaves the function
371:         }
372:
373:         Queue* temp = first; //Temporary node to hold the first node
374:
375:         //Goes through the queue until it reaches the last node
376:         while (temp->next != nullptr)
377:             temp = temp->next; //Goes to the next node
378:
379:         //Makes the new node into the last node
380:         temp->next = newQ;
381:         last = temp;
382:
383:         cout << x << " Was enqueued!\n";
384:     }
385:
```

```
386:     //Removes the first node
387:     //Time Complexity: O(1) because it has a constant amount of instructions
388:     int dequeue(){
389:
390:             int removed; //Variable to store removed data
391:             //Checks if the queue is empty
392:             if (isEmpty())
393:                 cout << "The queue is empty. It cannot be dequeued." << endl;
394:
395:             //Checks if the first node is the only node
396:             //In that case, the first node will point to nothing
397:             else if (first->next == nullptr){
398:
399:                 cout << first->data << " Was dequeued.\n";
400:
401:                 removed = first->data;
402:
403:                 first = nullptr; //First node is pointing to nothing
404:
405:                 return removed; //returns removed data
406:             }
407:
408:             //The case where the first node is not the only node
409:             //Then the node after the first node will be the new first node
410:             else{
411:                 cout << first->data << " Was dequeued.\n";
412:
413:                 removed = first->data;
414:
415:                 first = first->next;
416:
417:                 return removed; //returns removed data
418:             }
419:     }
420:
421:     //Returns the data of the first node
422:     //Time Complexity: O(1) because it has a constant amount of instructions
423:     int top(){
424:
425:         //Checks whether the queue is empty or not
426:         if (isEmpty()){
427:
428:             cout << "ERROR: queue is EMPTY!" << endl;
429:
430:             return NULL; //Returns nothing if the queue is empty
431:         }
432:
433:         //The case where the queue is not empty.
434:         //Returns the data of the first node
435:         return first->data;
436:     }
437:
438:     //Returns the size of the queue
439:     //Time Complexity: O(n) because the loop iterates at least n times
440:     int Size(){
```

```
441:
442:         //Checks if the queue is empty. (The size is 0 if the queue is empty, so it returns 0
443:         if (isEmpty())
444:             return 0;
445:
446:         Queue* temp = first; //Temporary node to hold the first node
447:         int count = 0; //counter for the size
448:
449:         //Goes through the queue to count the size
450:         while (temp != nullptr){
451:
452:             temp = temp->next; //Goes to the next node
453:
454:             count++;//increments counter
455:         }
456:
457:         //Returns the counter (the size)
458:         return count;
459:     }
460:
461:            /*Print function*/
462:     void print(){
463:
464:         if (isEmpty()){
465:             cout << "Linked list is empty!\n";
466:             return;
467:         }
468:
469:         Queue* temp = first;
470:
471:         while (temp != nullptr){
472:
473:             if (temp->next == nullptr)
474:                 cout << temp->data << endl;
475:             else
476:                 cout << temp->data << " -->";
477:
478:             temp = temp->next;
479:         }
480:     }
481:
482: private:
483:
484:     Queue* first = nullptr; //First node in the queue
485:     Queue* last = nullptr; //Last node in the queue
486:     Queue* next;
487:     int data;
488: };
489:
490: int main(){
491:
492:     /*Linked List test*/
493:     cout << "Linked list test:\n";
494:
495:     LinkedList L;
```

```
496:
497:     L.add(1);
498:     L.add(2);
499:     L.add(3);
500:
501:     L.print();
502:
503:     cout << "Does the list have 2 as a data? The answer is: ";
504:
505:     if (L.search(2)) cout << "True\n";
506:     else cout << "False\n";
507:
508:     cout << "The size of the list is: " << L.Size() << endl;
509:
510:     L.Remove(2);
511:
512:     L.print();
513:
514:     /*Stack test*/
515:     cout << "Stack test:\n";
516:
517:     Stack S;
518:
519:     S.push(1);
520:     S.push(2);
521:     S.push(3);
522:
523:     S.print();
524:
525:     cout << "The top of the stack is: " << S.top() << endl;
526:     cout << "The size of the list is: " << S.Size() << endl;
527:
528:     S.pop();
529:
530:     S.print();
531:
532:     /*Queue test*/
533:     cout << "Queue test:\n";
534:
535:     Queue Q;
536:
537:     Q.enqueue(1);
538:     Q.enqueue(2);
539:     Q.enqueue(3);
540:
541:     Q.print();
542:
543:     cout << "The top of the stack is: " << Q.top() << endl;
544:     cout << "The size of the list is: " << Q.Size() << endl;
545:
546:     Q.dequeue();
547:
548:     Q.print();
549:
550:     return 0;
```

```
551: }
```

15

```
 1: //Name: Ghazi Najeeb Al-Abbar
 2: //ID: 2181148914
 3: //problem-2.cpp
 4:
 5: //A Queue class implemented using 4 stacks from the STL
 6: //Assume the n is the number of elements in the stack
 7:
 8: #include <iostream>
 9: #include <stack>
10: using namespace std;
11:
12: class Queue{
13:
14: public:
15:
16:     Queue(){/*Empty*/}
17:
18:     //Time Complexity: O(1) because it has a constant number of instructions
19:     bool isEmpty(){ return S1.empty(); }
20:
21:     //enqueues x into the queue (the first stack)
22:     //Time Complexity: O(n) because both loops are iterated n - 2 times (worst case scena
23:     void enqueue(int x){
24:
25:         //Checks if the first stack is empty
26:         //If it is, x is pushed into the first stack
27:         if (isEmpty()){
28:
29:             S1.push(x);
30:
31:             cout << x << " Was enqueued!\n";
32:
33:             return; //leaves the function
34:         }
35:
36:         //The case where the first stack is not empty
37:
38:         //Pushes the first element in the queue to the second stack
39:         if (S2.empty() && !isEmpty()){
40:
41:             S2.push(S1.top());
42:             S1.pop();
43:         }
44:
45:         //Pushes the second element in the queue to the third stack
46:         if (S3.empty() && !isEmpty()){
47:
48:             S3.push(S1.top());
49:             S1.pop();
50:         }
51:
52:         //Fills the fourth stack with the rest of the elements from the first stack
53:         while (!isEmpty()){
54:
55:             S4.push(S1.top());
```

```
56:             S1.pop();
57:         }
58:
59:         S1.push(x); //Pushes the newest element to the bottom of the first stack (or the begi
60:
61:         cout << x << " Was enqueued!\n";
62:
63:         //Empties the 4th stack while filling the first stack
64:         while (!S4.empty()){
65:
66:             S1.push(S4.top());
67:             S4.pop();
68:         }
69:
70:         //Puts the second element of the queue as the second top of the first stack
71:          if (!S3.empty()){
72:
73:             S1.push(S3.top());
74:             S3.pop();
75:         }
76:
77:         //Puts the first element of the queue as the top of the first stack
78:         if (!S2.empty()){
79:
80:             S1.push(S2.top());
81:             S2.pop();
82:         }
83:             }
84:
85:     //removes the first element from the queue (or the top element from the first stack)
86:     //Time Complexity: O(1) because there are a constant number of instructions
87:     int dequeue(){
88:
89:         //Checks if the queue is empty
90:         if (isEmpty())
91:             cout << "The queue is empty!\n";
92:
93:         //When the queue is not empty
94:         else{
95:
96:             int removed = S1.top(); //Stores the top element of the first stack in a variable
97:
98:             S1.pop(); //Pops the top element
99:
100:            return removed; //returns the removed element (the recent top)
101:        }
102:    }
103:
104:    //returns the size of the queue
105:    //Time Complexity: O(1) because there are a constant number of instructions
106:    int Size(){
107:        //If the first Stack is empty, the function returns zero
108:        if (isEmpty()) return 0;
109:
110:        //Otherwise return the size of the first stack
```

```
111:         return S1.size();
112:     }
113:
114:     //returns the first element of the queue (or the last element of the first stack)
115:     //Time Complexity: O(1) because there are a constant number of instructions
116:     int top(){ return S1.top(); }
117:
118:         /*Print function*/
119:     void print(){
120:
121:         if (isEmpty()){
122:             cout << "The queue is empty!\n";
123:             return;
124:         }
125:
126:         while (!isEmpty()){
127:
128:             cout << S1.top() << "\t";
129:             S2.push(S1.top());
130:             S1.pop();
131:         }
132:
133:         cout << "\n";
134:
135:         while (!S2.empty()){
136:             S1.push(S2.top());
137:             S2.pop();
138:         }
139:     }
140:
141: private:
142:
143:     //S1 is used to represent the queue
144:     //S2, S3, S4 are used to store the elements when a new element is being pushed
145:     stack<int> S1, S2, S3, S4;
146:
147: };
148:
149:
150: int main(){
151:
152:     Queue Q;
153:
154:     Q.enqueue(1);
155:     Q.enqueue(2);
156:     Q.enqueue(3);
157:
158:     Q.print();
159:
160:     cout << "The size of the queue is: " << Q.Size() << endl;
161:     cout << "The top of the queue is: " << Q.top() << endl;
162:
163:     Q.dequeue();
164:
165:     Q.print();
```

```
166:
167:     return 0;
168: }
```

15

```cpp
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//problem-3.cpp

//A Stack class implemented using two queues from the STL
//Assume that n is the number of elements in the queue

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

class Stack{

public:

    //Constructor
    Stack(){/*Empty*/}

    //Returns true if the first queue is empty. False otherwise
    //Time Complexity: O(1) since there is a constant number of commands
    bool isEmpty(){ return Q1.empty();}

    //Stack push implemented using two queues
    //Time Complexity: O(n) because both loops are iterated n times
    void push(int x){

        //Goes through the first queue and emptys it it into the second queue
        while (!Q1.empty()){

            Q2.push(Q1.front());
            Q1.pop();
        }

        //Pushes the data x into the first queue
        //So it can resemble the top element of the stack
        Q1.push(x);

        cout << x << " Was pushed!\n";

        //Pushes everything from the second queue into the first in order
        while (!Q2.empty()){

            Q1.push(Q2.front());
            Q2.pop();
        }
    }

    //Stack pop implemented using queues
    //Time Complexity: O(1), since there is a constant number of commands
    int pop(){

        if (isEmpty()) //Checks if the queue is empty
            cout << "Stack is Empty!\n";

        //The case where it is not empty
```

```
56:         else{
57:             int removed = Q1.front(); //variable to store the queue top
58:
59:             cout << Q1.front() << " Was popped!\n";
60:
61:             Q1.pop();
62:
63:             return removed; //returns the removed top
64:         }
65:     }
66:
67:     //Stack size implemented using two queues
68:     //Time Complexity: O(n), because both loops are iterated n times
69:     int Size(){
70:
71:         int count = 0; //counter to check how many elements in the queue
72:
73:         //Goes through the queue and counts each element while emptying the first queue into
74:         while (!Q1.empty()){
75:
76:             Q2.push(Q1.front());
77:
78:             Q1.pop();
79:
80:             count++;
81:         }
82:
83:         //Returning everything from the second queue into the first in order
84:         while (!Q2.empty()){
85:
86:             Q1.push(Q2.front());
87:
88:             Q2.pop();
89:         }
90:
91:         return count; //returns counter
92:     }
93:
94:     //Returns the top of the stack
95:     //Time Complexity: O(1) since there is only one command
96:     int top(){ return Q1.front();}
97:
98:        /*Print function*/
99:     void print(){
100:
101:        while (!Q1.empty()){
102:
103:            Q2.push(Q1.front());
104:            cout << Q1.front() << "\t";
105:            Q1.pop();
106:        }
107:
108:        cout << " <--  first\n";
109:
110:        while (!Q2.empty()){
```

```
111:
112:             Q1.push(Q2.front());
113:             Q2.pop();
114:         }
115:
116:     }
117:
118: private:
119:     //STL queue
120:     queue<int> Q1, Q2;
121: };
122:
123: int main(){
124:
125:     Stack S;
126:
127:     S.push(1);
128:     S.push(2);
129:     S.push(3);
130:
131:     S.print();
132:
133:     cout << "The size of the stack is: " << S.Size() << endl;
134:     cout << "The top of the stack is: " << S.top() << endl;
135:
136:     S.pop();
137:
138:     S.print();
139:
140:     return 0;
141: }
```

```cpp
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//problem-4.cpp

#include <iostream>
#include <list>
#include <iterator>
using namespace std;

     /*List print function*/
void showlist(list <int> g)
{
    list <int> :: iterator it;
    for(it = g.begin(); it != g.end(); ++it)
        cout << '\t' << *it;
    cout << '\n';
}
/*****************************************/

//Merge two sorted linked lists in sorted order
//Let n and m be the sizes of list1 and list2
//Time Complexity: O(m+n) because because both loop is being iterated n+m times
list<int> Merge(list<int> list1, list<int> list2){

    //Checks if list1 is empty. It will return list2 if list1 is empty
    if (list1.empty())
        return list2;

    //Checks if list2 is empty. It will return list1 if list2 is empty
    if (list2.empty())
        return list1;

    list<int> list3; //A new list to contain all the nodes in sorted order

    //Goes through both list1 and list2 until they're both empty and appends the nodes in
    while (!list1.empty() || !list2.empty()){

        //Checks if list1 and list2 are not empty
        if (!list1.empty()){
            if (!list2.empty()){

                //Checks if the list1's front is less than or equal to list2's front
                //If that's the case, then list1's front will be appended in list3
                //Then list1's front is popped
                if (list1.front() <= list2.front()){

                    list3.push_back(list1.front());
                    list1.pop_front();
                }
            }
            //The case where list1 is not empty and list2 is empty, which means the rest of l
            //If that's the case, then list1's front will be appended in list3
            //Then list1's front is popped
            else{
                list3.push_back(list1.front());
```

```
56:                 list1.pop_front();
57:             }
58:         }
59:
60:         //Checks if list2 and list1 are not empty
61:         if (!list2.empty()){
62:             if (!list1.empty()){
63:
64:                 //Checks if the list2's front is greater than list1's front
65:                 //If that's the case, then list2's front will be appended in list3
66:                 //Then list2's front is popped
67:                 if (list1.front() > list2.front()){
68:
69:                     list3.push_back(list2.front());
70:                     list2.pop_front();
71:                 }
72:             }
73:
74:             //The case where list2 is not empty and list1 is empty, which means the rest of l
75:             //If that's the case, then list2's front will be appended in list3
76:             //Then list2's front is popped
77:             else{
78:                 list3.push_back(list2.front());
79:                 list2.pop_front();
80:             }
81:         }
82:     }
83:
84:     return list3; //list3 is returned
85: }
86:
87: //Beginning of program
88: int main(){
89:
90:     //Three lists were declared
91:     list<int> L1,L2,L3;
92:
93:     L1.push_back(1);
94:     L1.push_back(3);
95:     L1.push_back(5);
96:
97:     L2.push_back(2);
98:     L2.push_back(4);
99:     L2.push_back(6);
100:
101:    //L1: 1    3    5
102:    //L2: 2    4    6
103:    showlist(L1);
104:    showlist(L2);
105:
106:    L3 = Merge(L1,L2);
107:
108:    //L3: 1    2    3   4   5   6
109:    showlist(L3);
110:
```

```
111:     return 0;
112: }
113: //End of program
```

30

```cpp
//Name: Ghazi Najeeb Al-Abbar
//ID: 2181148914
//problem-5.cpp

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

//The function takes a string, which is one or more balanced parenthesis, and prints t
//Assuming the user entered a balanced parenthesis
//Suppose n = s.length()
//Time Complexity: O(n), because both loops are iterated n times
void split(string s){

    int Parenthesis_Count = 0; //Counter to keep track of the balanced parenthesis
    stack<char> stk; //STL stack declaration to store each character of the string

    //Goes through the string and pushes each character into the stack, beginning with the
    for (int i = s.length() - 1; i >= 0; i--)
            stk.push(s[i]); //Pushes the character into the stack

    //Goes through the stack and prints a new line whenever a balanced parenthesis is reac
    for (int i = 0; i < s.length(); i++){

        cout << stk.top(); //Prints an character from the top of the stack

        //If the character is '(', then the counter is incremented
        if (stk.top() == '(')
                Parenthesis_Count++;

        //If the character is ')', then the counter is decremented
        if (stk.top() == ')')
            Parenthesis_Count--;

        //If the counter reaches zero, a new line will be printed
        if (Parenthesis_Count == 0)
            cout << endl;

        stk.pop(); //Removes the top of the stack
    }
}

int main(){

    split("(()()()()()())");
    cout << "\n\n";
    split("()((()))(()()()())");

    return 0;
}
```