```cpp
#include <iostream>
#include <cstdlib>
#include "llcpInt.h"
using namespace std;

// definition of DelOddCopEven of Assignment 5 Part 1
// (put at near top to facilitate printing and grading)
void DelOddCopEven(Node* headPtr)
{
    if (headPtr == 0) //check for empty list
    {
        return;
    }

    Node* currNode = headPtr;
    Node* prevNode = headPtr;

    while (currNode != 0)
    {
        if (currNode->data % 2 != 0) //checks if odd and del
        {
            if (currNode == headPtr) //if head needs to be del
            {
                headPtr = headPtr->link;
                delete currNode;
                currNode = headPtr;
                prevNode = headPtr;
            }
            else
            {
                Node* tempNode = currNode;
                currNode = currNode->link;
                delete tempNode;
                prevNode->link = currNode; //connect prevNode to currNode

            }
        }
        else //duplicate even
        {
            Node* dupEven = new Node;
            dupEven->data = currNode->data;
            dupEven->link = currNode->link;
            currNode->link = dupEven;
            prevNode = dupEven;
            currNode = dupEven->link; //prevents infinite loop of evens
        }
    }
}

```

```
50
51 int FindListLength(Node* headPtr)
52 {
53    int length = 0;
54
55    while (headPtr != 0)
56    {
57       ++length;
58       headPtr = headPtr->link;
59    }
60
61    return length;
62 }
63
64 bool IsSortedUp(Node* headPtr)
65 {
66    if (headPtr == 0 || headPtr->link == 0) // empty or 1-node
67       return true;
68    while (headPtr->link != 0) // not at last node
69    {
70       if (headPtr->link->data < headPtr->data)
71          return false;
72       headPtr = headPtr->link;
73    }
74    return true;
75 }
76
77 void InsertAsHead(Node*& headPtr, int value)
78 {
79    Node *newNodePtr = new Node;
80    newNodePtr->data = value;
81    newNodePtr->link = headPtr;
82    headPtr = newNodePtr;
83 }
84
85 void InsertAsTail(Node*& headPtr, int value)
86 {
87    Node *newNodePtr = new Node;
88    newNodePtr->data = value;
89    newNodePtr->link = 0;
90    if (headPtr == 0)
91       headPtr = newNodePtr;
92    else
93    {
94       Node *cursor = headPtr;
95
96       while (cursor->link != 0) // not at last node
97          cursor = cursor->link;
98       cursor->link = newNodePtr;
```

```
 99     }
100  }
101
102  void InsertSortedUp(Node*& headPtr, int value)
103  {
104     Node *precursor = 0,
105          *cursor = headPtr;
106
107     while (cursor != 0 && cursor->data < value)
108     {
109        precursor = cursor;
110        cursor = cursor->link;
111     }
112
113     Node *newNodePtr = new Node;
114     newNodePtr->data = value;
115     newNodePtr->link = cursor;
116     if (cursor == headPtr)
117        headPtr = newNodePtr;
118     else
119        precursor->link = newNodePtr;
120
121     ////////////////////////////////////////////////////////
122     /* using-only-cursor (no precursor) version
123     Node *newNodePtr = new Node;
124     newNodePtr->data = value;
125     //newNodePtr->link = 0;
126     //if (headPtr == 0)
127     //   headPtr = newNodePtr;
128     //else if (headPtr->data >= value)
129     //{
130     //   newNodePtr->link = headPtr;
131     //   headPtr = newNodePtr;
132     //}
133     if (headPtr == 0 || headPtr->data >= value)
134     {
135        newNodePtr->link = headPtr;
136        headPtr = newNodePtr;
137     }
138     //else if (headPtr->link == 0)
139     //   head->link = newNodePtr;
140     else
141     {
142        Node *cursor = headPtr;
143        while (cursor->link != 0 && cursor->link->data < value)
144           cursor = cursor->link;
145        //if (cursor->link != 0)
146        //   newNodePtr->link = cursor->link;
147        newNodePtr->link = cursor->link;
```

```
148         cursor->link = newNodePtr;
149      }
150
151      ////////////////// commented lines removed /////////////////
152
153      Node *newNodePtr = new Node;
154      newNodePtr->data = value;
155      if (headPtr == 0 || headPtr->data >= value)
156      {
157         newNodePtr->link = headPtr;
158         headPtr = newNodePtr;
159      }
160      else
161      {
162         Node *cursor = headPtr;
163         while (cursor->link != 0 && cursor->link->data < value)
164            cursor = cursor->link;
165         newNodePtr->link = cursor->link;
166         cursor->link = newNodePtr;
167      }
168      */
169      ////////////////////////////////////////////////////////
170   }
171
172   bool DelFirstTargetNode(Node*& headPtr, int target)
173   {
174      Node *precursor = 0,
175           *cursor = headPtr;
176
177      while (cursor != 0 && cursor->data != target)
178      {
179         precursor = cursor;
180         cursor = cursor->link;
181      }
182      if (cursor == 0)
183      {
184         cout << target << " not found." << endl;
185         return false;
186      }
187      if (cursor == headPtr) //OR precursor == 0
188         headPtr = headPtr->link;
189      else
190         precursor->link = cursor->link;
191      delete cursor;
192      return true;
193   }
194
195   bool DelNodeBefore1stMatch(Node*& headPtr, int target)
196   {
```

```
197     if (headPtr == 0 || headPtr->link == 0 || headPtr->data == target)
          return false;
198     Node *cur = headPtr->link, *pre = headPtr, *prepre = 0;
199     while (cur != 0 && cur->data != target)
200     {
201        prepre = pre;
202        pre = cur;
203        cur = cur->link;
204     }
205     if (cur == 0) return false;
206     if (cur == headPtr->link)
207     {
208        headPtr = cur;
209        delete pre;
210     }
211     else
212     {
213        prepre->link = cur;
214        delete pre;
215     }
216     return true;
217  }
218
219  void ShowAll(ostream& outs, Node* headPtr)
220  {
221     while (headPtr != 0)
222     {
223        outs << headPtr->data << "  ";
224        headPtr = headPtr->link;
225     }
226     outs << endl;
227  }
228
229  void FindMinMax(Node* headPtr, int& minValue, int& maxValue)
230  {
231     if (headPtr == 0)
232     {
233        cerr << "FindMinMax() attempted on empty list" << endl;
234        cerr << "Minimum and maximum values not set" << endl;
235     }
236     else
237     {
238        minValue = maxValue = headPtr->data;
239        while (headPtr->link != 0)
240        {
241           headPtr = headPtr->link;
242           if (headPtr->data < minValue)
243              minValue = headPtr->data;
244           else if (headPtr->data > maxValue)
```

```
245             maxValue = headPtr->data;
246       }
247    }
248 }
249
250 double FindAverage(Node* headPtr)
251 {
252    if (headPtr == 0)
253    {
254       cerr << "FindAverage() attempted on empty list" << endl;
255       cerr << "An arbitrary zero value is returned" << endl;
256       return 0.0;
257    }
258    else
259    {
260       int sum = 0,
261           count = 0;
262
263       while (headPtr != 0)
264       {
265          ++count;
266          sum += headPtr->data;
267          headPtr = headPtr->link;
268       }
269
270       return double(sum) / count;
271    }
272 }
273
274 void ListClear(Node*& headPtr, int noMsg)
275 {
276    int count = 0;
277
278    Node *cursor = headPtr;
279    while (headPtr != 0)
280    {
281       headPtr = headPtr->link;
282       delete cursor;
283       cursor = headPtr;
284       ++count;
285    }
286    if (noMsg) return;
287    clog << "Dynamic memory for " << count << " nodes freed"
288         << endl;
289 }
290
291
292
```