llcpImp.cpp

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

// definition of PropTarget
// (put at near top to facilitate printing and grading)
void PropTarget(Node*& headptr, int value){
   Node* cur = headptr;
   Node* pre = nullptr;
   bool found = false;

  if (headptr == 0){
   Node *newNodePtr = new Node;
   newNodePtr->data = value;
   newNodePtr->link = headptr;
   headptr = newNodePtr;
 }
   else{
       while (cur != nullptr) {
         if(cur->data != value){
            pre = cur;
            cur = cur->link;
         }
         else{
            found = true;

            if(pre != nullptr){
               pre->link = cur->link;
               cur->link = headptr;
               headptr = cur;
               cur = pre->link;
            }
            else{
               pre = cur;
               cur = cur->link;
            }
         }
       }

    if(found == false){
      Node* newNode = new Node;
      newNode->data = value;
      newNode->link = nullptr;

      pre->link = newNode;
      }
   }
}


int FindListLength(Node* headPtr)
{
```

```
54    int length = 0;
55
56    while (headPtr != 0)
57    {
58       ++length;
59       headPtr = headPtr->link;
60    }
61
62    return length;
63 }
64
65 bool IsSortedUp(Node* headPtr)
66 {
67    if (headPtr == 0 || headPtr->link == 0) // empty or 1-node
68       return true;
69    while (headPtr->link != 0) // not at last node
70    {
71       if (headPtr->link->data < headPtr->data)
72          return false;
73       headPtr = headPtr->link;
74    }
75    return true;
76 }
77
78 void InsertAsHead(Node*& headPtr, int value)
79 {
80    Node *newNodePtr = new Node;
81    newNodePtr->data = value;
82    newNodePtr->link = headPtr;
83    headPtr = newNodePtr;
84 }
85
86 void InsertAsTail(Node*& headPtr, int value)
87 {
88    Node *newNodePtr = new Node;
89    newNodePtr->data = value;
90    newNodePtr->link = 0;
91    if (headPtr == 0)
92       headPtr = newNodePtr;
93    else
94    {
95       Node *cursor = headPtr;
96
97       while (cursor->link != 0) // not at last node
98          cursor = cursor->link;
99       cursor->link = newNodePtr;
100   }
101 }
102
103 void InsertSortedUp(Node*& headPtr, int value)
104 {
105    Node *precursor = 0,
106         *cursor = headPtr;
107
108    while (cursor != 0 && cursor->data < value)
109    {
```

```
110         precursor = cursor;
111         cursor = cursor->link;
112      }
113
114      Node *newNodePtr = new Node;
115      newNodePtr->data = value;
116      newNodePtr->link = cursor;
117      if (cursor == headPtr)
118         headPtr = newNodePtr;
119      else
120         precursor->link = newNodePtr;
121
122      ////////////////////////////////////////////////////////
123      /* using-only-cursor (no precursor) version
124      Node *newNodePtr = new Node;
125      newNodePtr->data = value;
126      //newNodePtr->link = 0;
127      //if (headPtr == 0)
128      //   headPtr = newNodePtr;
129      //else if (headPtr->data >= value)
130      //{
131      //   newNodePtr->link = headPtr;
132      //   headPtr = newNodePtr;
133      //}
134      if (headPtr == 0 || headPtr->data >= value)
135      {
136         newNodePtr->link = headPtr;
137         headPtr = newNodePtr;
138      }
139      //else if (headPtr->link == 0)
140      //   head->link = newNodePtr;
141      else
142      {
143         Node *cursor = headPtr;
144         while (cursor->link != 0 && cursor->link->data < value)
145            cursor = cursor->link;
146         //if (cursor->link != 0)
147         //   newNodePtr->link = cursor->link;
148         newNodePtr->link = cursor->link;
149         cursor->link = newNodePtr;
150      }
151
152      ////////////////// commented lines removed /////////////////
153
154      Node *newNodePtr = new Node;
155      newNodePtr->data = value;
156      if (headPtr == 0 || headPtr->data >= value)
157      {
158         newNodePtr->link = headPtr;
159         headPtr = newNodePtr;
160      }
161      else
162      {
163         Node *cursor = headPtr;
164         while (cursor->link != 0 && cursor->link->data < value)
165            cursor = cursor->link;
```

```
166         newNodePtr->link = cursor->link;
167         cursor->link = newNodePtr;
168      }
169      */
170      ////////////////////////////////////////////////////////
171   }
172
173   bool DelFirstTargetNode(Node*& headPtr, int target)
174   {
175      Node *precursor = 0,
176           *cursor = headPtr;
177
178      while (cursor != 0 && cursor->data != target)
179      {
180         precursor = cursor;
181         cursor = cursor->link;
182      }
183      if (cursor == 0)
184      {
185         cout << target << " not found." << endl;
186         return false;
187      }
188      if (cursor == headPtr) //OR precursor == 0
189         headPtr = headPtr->link;
190      else
191         precursor->link = cursor->link;
192      delete cursor;
193      return true;
194   }
195
196   bool DelNodeBefore1stMatch(Node*& headPtr, int target)
197   {
198      if (headPtr == 0 || headPtr->link == 0 || headPtr->data == target) return false;
199      Node *cur = headPtr->link, *pre = headPtr, *prepre = 0;
200      while (cur != 0 && cur->data != target)
201      {
202         prepre = pre;
203         pre = cur;
204         cur = cur->link;
205      }
206      if (cur == 0) return false;
207      if (cur == headPtr->link)
208      {
209         headPtr = cur;
210         delete pre;
211      }
212      else
213      {
214         prepre->link = cur;
215         delete pre;
216      }
217      return true;
218   }
219
220   void ShowAll(ostream& outs, Node* headPtr)
221   {
```

```
222     while (headPtr != 0)
223     {
224        outs << headPtr->data << "  ";
225        headPtr = headPtr->link;
226     }
227     outs << endl;
228  }
229
230  void FindMinMax(Node* headPtr, int& minValue, int& maxValue)
231  {
232     if (headPtr == 0)
233     {
234        cerr << "FindMinMax() attempted on empty list" << endl;
235        cerr << "Minimum and maximum values not set" << endl;
236     }
237     else
238     {
239        minValue = maxValue = headPtr->data;
240        while (headPtr->link != 0)
241        {
242           headPtr = headPtr->link;
243           if (headPtr->data < minValue)
244              minValue = headPtr->data;
245           else if (headPtr->data > maxValue)
246              maxValue = headPtr->data;
247        }
248     }
249  }
250
251  double FindAverage(Node* headPtr)
252  {
253     if (headPtr == 0)
254     {
255        cerr << "FindAverage() attempted on empty list" << endl;
256        cerr << "An arbitrary zero value is returned" << endl;
257        return 0.0;
258     }
259     else
260     {
261        int sum = 0,
262            count = 0;
263
264        while (headPtr != 0)
265        {
266           ++count;
267           sum += headPtr->data;
268           headPtr = headPtr->link;
269        }
270
271        return double(sum) / count;
272     }
273  }
274
275  void ListClear(Node*& headPtr, int noMsg)
276  {
277     int count = 0;
```

```
278
279     Node *cursor = headPtr;
280     while (headPtr != 0)
281     {
282        headPtr = headPtr->link;
283        delete cursor;
284        cursor = headPtr;
285        ++count;
286     }
287     if (noMsg) return;
288     clog << "Dynamic memory for " << count << " nodes freed"
289          << endl;
290  }
291
292
```