

list.h

```
1 /*****
2 File name:      list.h (Version 2.0)
3 Author:        Doug Ryan
4 Edited:        $Author: chadd $
5 Date:          9/28/11
6 Class:         CS300
7 Assignment:    List Implementation
8 Purpose:       This file defines the constants, data structures, and function
9                prototypes for implementing a list data structure. In essence,
10               the list API is defined for other modules.
11
12 Modifications:
13 RevisionID:    $Id: list.h 56 2011-09-30 15:33:09Z chadd $
14 *****/
15
16 #ifndef LIST_H_
17 #define LIST_H_
18
19 #define MAX_LIST_ELEMENTS    1024
20
21 #define TRUE    1
22 #define FALSE   0
23
24 // List error codes for each function to use
25
26 #define NO_ERROR          0
27
28 // list create failed
29 #define ERROR_NO_LIST_CREATE    -1
30
31 // user tried to operate on an empty list
32 #define ERROR_EMPTY_LIST        -2
33
34 // user tried to add data to a full list
35 #define ERROR_FULL_LIST         -3
36
37 // user tried to peekNext when no next existed
38 #define ERROR_NO_NEXT           -6
39
40 // user tried to peekPrev when no next existed
41 #define ERROR_NO_PREV           -7
42
43 // user tried to use current when current was not defined
44 #define ERROR_NO_CURRENT        -8
45
46 // user tried to operate on an invalid list. An invalid
47 // list may be a NULL ListPtr or contain an invalid value for numElements
48 #define ERROR_INVALID_LIST      -9
49
50 // user provided a NULL pointer to the function (other than the ListPtr)
51 #define ERROR_NULL_PTR          -10
52
53
54 #define NO_CURRENT -100
55 #define EMPTY_LIST 0
56
57 // User-defined datatypes for easier reading
58
59 typedef short int BOOLEAN;
```

list.h

```
60 typedef short int ERRORCODE;
61
62
63 // The user of this data structure is only concerned with
64 // two data types: List and DATATYPE. ListElement is an internal
65 // data structure not to be directly used by the user.
66 // If the List implementation changes (to dynamic memory, a tree, etc)
67 // ListElement will change.
68
69 #define CHARACTER_VALUE 0
70 #define INTEGER_VALUE 1
71 #define FLOAT_VALUE 2
72
73
74 typedef struct
75 {
76     union
77     {
78         char charValue;
79         unsigned int intValue;
80         float floatValue;
81     };
82     unsigned short whichOne;
83 } DATATYPE;
84
85
86 typedef struct
87 {
88     DATATYPE data;
89 } ListElement;
90
91
92 // A list is an array of ListElements where the current pointer and number
93 // of elements are maintained at all times
94
95 typedef struct List* ListPtr;
96
97 typedef struct List
98 {
99     ListElement listElements[MAX_LIST_ELEMENTS];
100     int current;
101     int numElements;
102 } List;
103
104 /*****
105 *                               Allocation and Deallocation
106 *****/
107 ERRORCODE lstCreate (ListPtr);
108 // results: If list L can be created, then L exists and
109 // is empty returning NO_ERROR; otherwise,
110 // NO_LIST_CREATE is returned
111
112 ERRORCODE lstDispose (ListPtr);
113 // results: List no longer exists
114
115 /*****
116 *                               Checking number of elements in list
117 *****/
118 ERRORCODE lstSize (ListPtr, int *);
```

list.h

```
119 // results: Returns the number of elements in the list
120
121 ERRORCODE lstIsFull (ListPtr, BOOLEAN *);
122 // results: If list is full, return true;
123 // otherwise, return false
124
125 ERRORCODE lstIsEmpty (ListPtr, BOOLEAN *);
126 // results: If list is empty, return true;
127 // otherwise, return false
128
129 /*****
130 *           Peek Operations
131 *****/
132 ERRORCODE lstPeek (ListPtr, DATATYPE *);
133 // requires: List is not empty
134 // results: The value of the current element is
135 // returned through the argument list
136 // IMPORTANT: Do not change current
137
138 ERRORCODE lstPeekPrev (ListPtr, DATATYPE *);
139 // requires: List contains two or more elements and
140 // current is not the first element
141 // results: The data value of current's predecessor is returned
142 // through the argument list.
143 // IMPORTANT: Do not change current
144
145 ERRORCODE lstPeekNext (ListPtr, DATATYPE *);
146 // requires: List contains two or more elements and
147 // current is not the last element
148 // results: The data value of current's successor is returned
149 // through the argument list.
150 // IMPORTANT: Do not change current
151
152 /*****
153 *           Retrieving values and updating current
154 *****/
155
156 ERRORCODE lstFirst(ListPtr, DATATYPE *);
157 // requires: List is not empty
158 // results: The value of the first element is returned
159 // IMPORTANT: Current is changed to first
160 // if it exists
161
162 ERRORCODE lstLast(ListPtr, DATATYPE *);
163 // requires: List is not empty
164 // results: The value of the last element is returned
165 // IMPORTANT: Current is changed to
166 // last if it exists
167
168 ERRORCODE lstNext(ListPtr, DATATYPE *);
169 // requires: List is not empty, and current is not past the end
170 // of the list
171 // results: The value of the current element is returned
172 // IMPORTANT: Current is changed to the successor
173 // of the current element
174
175 ERRORCODE lstPrev(ListPtr, DATATYPE *);
176 // requires: List is not empty, and current is not past the first
177 // of the list
```

list.h

```
178 // results: The value of the current element is returned
179 // IMPORTANT: Current is changed to previous
180 // if it exists
181
182 /*****
183 *           Insertion, Deletion, and Updating
184 *****/
185
186 ERRORCODE lstDeleteCurrent (ListPtr, DATATYPE *);
187 // requires: List is not empty
188 // results: The current element is deleted and its
189 // successor and predecessor become each
190 // others successor and predecessor. If the
191 // deleted element had a predecessor, then
192 // make it the new current element; otherwise,
193 // make the first element current if it exists.
194 // The deleted element is returned through the argument
195 // list.
196
197 ERRORCODE lstInsertAfter (ListPtr, DATATYPE);
198 // requires: List is not full
199 // results: if the list is not empty, insert the new
200 // element as the successor of the current
201 // element and make the inserted element the
202 // current element; otherwise, insert element
203 // and make it current. The new element is inserted into
204 // the proper place and all other elements are shifted
205 // down the list.
206
207 ERRORCODE lstInsertBefore (ListPtr, DATATYPE);
208 // requires: List is not full
209 // results: If the list is not empty, insert the new
210 // element as the predecessor of the current
211 // element and make the inserted element the
212 // current element; otherwise, insert element
213 // and make it current. The new element is inserted into
214 // the proper place and all other elements are shifted
215 // down the list.
216
217 ERRORCODE lstUpdateCurrent (ListPtr, DATATYPE);
218 // requires: List is not empty
219 // results: The value of ListElement is copied into the
220 // current element
221
222 /*****
223 *           List Testing
224 *****/
225
226 ERRORCODE lstHasNext (ListPtr, BOOLEAN*);
227 // results: Returns true if there are more elements when traversing
228 // the list in a forward direction; otherwise, false is
229 // returned.
230
231 ERRORCODE lstHasPrev(ListPtr, BOOLEAN *);
232 // results: Returns true if the current node has a
233 // predecessor; otherwise, false is returned
234
235 #endif /* LIST_H_ */
236
```