

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 80 2011-10-07 22:12:44Z 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;
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
```

list.h

```
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
75 // NEW DATATYPE FOR THE QUEUE
76 typedef struct Q_DATATYPE
77 {
78     int intValue; // end user data
79 }Q_DATATYPE;
80
81
82 /* DATATYPE really represents the PQ datatype since it contains the user's
83 * data (Q_DATATYPE) and priority
84 *
85 * ListElement is really the List datatype.
86 */
87 typedef struct
88 {
89     /* Queue data
90     *
91     */
92     Q_DATATYPE data;
93     int priority;
94
95
96
97
98     /* FOR ACADEMIC PURPOSES:
99     * These two items remain so that the listDriver will still compile and run.
100    * Your queue and queue driver MUST NOT use the union or whichOne.
101    * These are merely to not break existing code!
102    */
103    union
104    {
105        char charValue;
106        unsigned int intValue;
107        float floatValue;
108    };
109    unsigned short whichOne;
110
111 } DATATYPE;
112
113
114 typedef struct
115 {
116     DATATYPE data;
117 } ListElement;
118
119
120 // A list is an array of ListElements where the current pointer and number
121 // of elements are maintained at all times
122
123 typedef struct List* ListPtr;
124
125 typedef struct List
126 {
127     ListElement listElements[MAX_LIST_ELEMENTS];
128     int current;
```

list.h

```

129  int numElements;
130 } List;
131
132 /*****
133 *           Allocation and Deallocation
134 *****/
135 ERRORCODE lstCreate (ListPtr);
136 // results: If list L can be created, then L exists and
137 // is empty returning NO_ERROR; otherwise,
138 // NO_LIST_CREATE is returned
139
140 ERRORCODE lstDispose (ListPtr);
141 // results: List no longer exists
142
143 /*****
144 *           Checking number of elements in list
145 *****/
146 ERRORCODE lstSize (ListPtr, int *);
147 // results: Returns the number of elements in the list
148
149 ERRORCODE lstIsFull (ListPtr, BOOLEAN *);
150 // results: If list is full, return true;
151 // otherwise, return false
152
153 ERRORCODE lstIsEmpty (ListPtr, BOOLEAN *);
154 // results: If list is empty, return true;
155 // otherwise, return false
156
157 /*****
158 *           Peek Operations
159 *****/
160 ERRORCODE lstPeek (ListPtr, DATATYPE *);
161 // requires: List is not empty
162 // results: The value of the current element is
163 // returned through the argument list
164 // IMPORTANT: Do not change current
165
166 ERRORCODE lstPeekPrev (ListPtr, DATATYPE *);
167 // requires: List contains two or more elements and
168 // current is not the first element
169 // results: The data value of current's predecessor is returned
170 // through the argument list.
171 // IMPORTANT: Do not change current
172
173 ERRORCODE lstPeekNext (ListPtr, DATATYPE *);
174 // requires: List contains two or more elements and
175 // current is not the last element
176 // results: The data value of current's successor is returned
177 // through the argument list.
178 // IMPORTANT: Do not change current
179
180 /*****
181 *           Retrieving values and updating current
182 *****/
183
184 ERRORCODE lstFirst(ListPtr, DATATYPE *);
185 // requires: List is not empty
186 // results: The value of the first element is returned
187 // IMPORTANT: Current is changed to first
188 // if it exists
189
190 ERRORCODE lstLast(ListPtr, DATATYPE *);
191 // requires: List is not empty
192 // results: The value of the last element is returned

```

list.h

```
193 // IMPORTANT: Current is changed to
194 // last if it exists
195
196 ERRORCODE lstNext(ListPtr, DATATYPE *);
197 // requires: List is not empty, and current is not past the end
198 // of the list
199 // results: The value of the current element is returned
200 // IMPORTANT: Current is changed to the successor
201 // of the current element
202
203 ERRORCODE lstPrev(ListPtr, DATATYPE *);
204 // requires: List is not empty, and current is not past the first
205 // of the list
206 // results: The value of the current element is returned
207 // IMPORTANT: Current is changed to previous
208 // if it exists
209
210 /*****
211 *          Insertion, Deletion, and Updating
212 *****/
213
214 ERRORCODE lstDeleteCurrent (ListPtr, DATATYPE *);
215 // requires: List is not empty
216 // results: The current element is deleted and its
217 // successor and predecessor become each
218 // others successor and predecessor. If the
219 // deleted element had a predecessor, then
220 // make it the new current element; otherwise,
221 // make the first element current if it exists.
222 // The deleted element is returned through the argument
223 // list.
224
225 ERRORCODE lstInsertAfter (ListPtr, DATATYPE);
226 // requires: List is not full
227 // results: if the list is not empty, insert the new
228 // element as the successor of the current
229 // element and make the inserted element the
230 // current element; otherwise, insert element
231 // and make it current. The new element is inserted into
232 // the proper place and all other elements are shifted
233 // down the list.
234
235 ERRORCODE lstInsertBefore (ListPtr, DATATYPE);
236 // requires: List is not full
237 // results: If the list is not empty, insert the new
238 // element as the predecessor of the current
239 // element and make the inserted element the
240 // current element; otherwise, insert element
241 // and make it current. The new element is inserted into
242 // the proper place and all other elements are shifted
243 // down the list.
244
245 ERRORCODE lstUpdateCurrent (ListPtr, DATATYPE);
246 // requires: List is not empty
247 // results: The value of ListElement is copied into the
248 // current element
249
250 /*****
251 *          List Testing
252 *****/
253
254 ERRORCODE lstHasNext (ListPtr, BOOLEAN*);
255 // results: Returns true if there are more elements when traversing
256 // the list in a forward direction; otherwise, false is
```

list.h

```
257 // returned.
258
259 ERRORCODE lstHasPrev(ListPtr, BOOLEAN *);
260 // results: Returns true if the current node has a
261 // predecessor; otherwise, false is returned
262
263 #endif /* LIST_H_ */
264
265
```