

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "list.h"
4
5  /* List Interface Routines */
6
7  list_t *create_list(void)
8  {
9      list_t *lst = get_node(0);
10     lst->next = lst->prev = lst;
11     return (lst);
12 }
13
14 result_t insert_beg(list_t *lst, data_t new_data)
15 {
16     g_insert(lst, get_node(new_data), lst->next);
17     return (SUCCESS);
18 }
19
20 result_t insert_end(list_t *lst, data_t new_data)
21 {
22     g_insert(lst->prev, get_node(new_data), lst);
23     return (SUCCESS);
24 }
25
26 result_t insert_after_data(list_t *lst, data_t e_data, data_t new_data)
27 {
28     node_t *e_node = search_node(lst, e_data);
29     if(!e_node)
30         return (DATA_NOT_FOUND);
31     g_insert(e_node, get_node(new_data), e_node->next);
32     return (SUCCESS);
33 }
34
35 result_t insert_before_data(list_t *lst, data_t e_data, data_t new_data)
36 {
37     node_t *e_node = search_node(lst, e_data);
38     if(!e_node)
39         return (DATA_NOT_FOUND);
40     g_insert(e_node->prev, get_node(new_data), e_node);
41     return (SUCCESS);
42 }
43
44 result_t delete_beg(list_t *lst)
45 {
46     if(is_empty(lst))
47         return (LIST_EMPTY);
48     g_delete(lst->next);
49     return (SUCCESS);
50 }
51
52 result_t delete_end(list_t *lst)
```

```
53 {
54     if(is_empty(lst))
55         return (LIST_EMPTY);
56     g_delete(lst->prev);
57     return (SUCCESS);
58 }
59
60 result_t delete_data(list_t *lst, data_t e_data)
61 {
62     node_t *e_node = search_node(lst, e_data);
63     if(!e_node)
64         return (DATA_NOT_FOUND);
65     g_delete(e_node);
66     return (SUCCESS);
67 }
68
69 result_t examine_beg(list_t *lst, data_t *p_data)
70 {
71     if(is_empty(lst))
72         return (LIST_EMPTY);
73     *p_data = lst->next->data;
74     return (SUCCESS);
75 }
76
77 result_t examine_end(list_t *lst, data_t *p_data)
78 {
79     if(is_empty(lst))
80         return (LIST_EMPTY);
81     *p_data = lst->prev->data;
82     return (SUCCESS);
83 }
84
85 result_t examine_and_delete_beg(list_t *lst, data_t *p_data)
86 {
87     if(is_empty(lst))
88         return (LIST_EMPTY);
89     *p_data = lst->next->data;
90     g_delete(lst->next);
91     return (SUCCESS);
92 }
93
94 result_t examine_and_delete_end(list_t *lst, data_t *p_data)
95 {
96     if(is_empty(lst))
97         return (LIST_EMPTY);
98     *p_data = lst->prev->data;
99     g_delete(lst->prev);
100     return (SUCCESS);
101 }
102
103 result_t find(list_t *lst, data_t f_data)
104 {
```

```
105     node_t *f_node = search_node(lst, f_data);
106     if(f_node)
107         return (TRUE);
108     return (FALSE);
109 }
110
111 void display(list_t *lst)
112 {
113     node_t *run;
114
115     printf("[beg]<->");
116
117     for(run = lst->next; run != lst; run = run->next)
118         printf("[%d]<->", run->data);
119
120     printf("[end]\n");
121 }
122
123 result_t is_empty(list_t *lst)
124 {
125     return (lst->next == lst && lst->prev == lst);
126 }
127
128 len_t len(list_t *lst)
129 {
130     node_t *run = lst->next;
131     len_t len = 0;
132
133     for(; run != lst; run = run->next, ++len)
134         ;
135
136     return (len);
137 }
138
139 data_t *to_array(list_t *lst, len_t *p_len)
140 {
141     len_t lst_len = len(lst);
142     data_t *arr;
143     node_t *run;
144     int i;
145
146     if(lst_len <= 0)
147         return (NULL);
148
149     arr = (data_t*)xmalloc(lst_len, sizeof(data_t));
150
151     for(run = lst->next, i = 0; run != lst; run = run->next, ++i)
152         arr[i] = run->data;
153
154     *p_len = lst_len;
155     return (arr);
156 }
```

```
157
158 list_t *to_list(data_t *p_data, len_t len)
159 {
160     list_t *new_list = create_list();
161     int i;
162
163     for(i = 0; i < len; i++)
164         insert_end(new_list, p_data[i]);
165
166     return (new_list);
167 }
168
169 list_t *merge(list_t *lst1, list_t *lst2)
170 {
171     list_t *lst3 = create_list();
172     node_t *run1 = lst1->next, *run2 = lst2->next;
173     flag_t from_lst1 = FALSE, from_lst2 = FALSE;
174
175     while(TRUE){
176
177         if(run1 == lst1){
178             from_lst1 = TRUE;
179             break;
180         }
181
182         if(run2 == lst2){
183             from_lst2 = TRUE;
184             break;
185         }
186
187         if(run1->data <= run2->data){
188             insert_end(lst3, run1->data);
189             run1 = run1->next;
190         }
191         else{
192             insert_end(lst3, run2->data);
193             run2 = run2->next;
194         }
195     }
196
197     if(from_lst1){
198         while(run2 != lst2){
199             insert_end(lst3, run2->data);
200             run2 = run2->next;
201         }
202     }
203     else if(from_lst2){
204         while(run1 != lst1){
205             insert_end(lst3, run1->data);
206             run1 = run1->next;
207         }
208     }
```

```
209
210     return (lst3);
211 }
212
213 /*
214 void ncat(list_t **pp_lst, int nr_lists, ...);
215
216 main()
217 {
218     list_t *lst1, lst2, lst3, lst4, lst5;
219     list_t *master_lst;
220     // populate lst1 to lst5
221
222     ncat(&master_lst, 5, lst1, lst2, lst3, lst4, lst5);
223 }
224 */
225
226 list_t *concat(list_t *lst1, list_t *lst2)
227 {
228     list_t *new_list = create_list();
229     node_t *run;
230
231     for(run = lst1->next; run != lst1; run = run->next)
232         insert_end(new_list, run->data);
233
234     for(run = lst2->next; run != lst2; run = run->next)
235         insert_end(new_list, run->data);
236
237     return (new_list);
238 }
239
240 result_t destroy_list(list_t **pp_list)
241 {
242     list_t *p_list = *pp_list;
243     node_t *run, *run_next;
244
245     for(run = p_list->next; run != p_list; run = run_next){
246         run_next = run->next;
247         free(run);
248     }
249
250     free(p_list);
251     *pp_list = NULL;
252     return (SUCCESS);
253 }
254
255 /* List auxillary routines */
256
257 void g_insert(node_t *beg, node_t *mid, node_t *end)
258 {
259     mid->next = end;
260     mid->prev = beg;
```

```
261     beg->next = mid;
262     end->prev = mid;
263 }
264
265 void g_delete(node_t *e_node)
266 {
267     e_node->next->prev = e_node->prev;
268     e_node->prev->next = e_node->next;
269     free(e_node);
270 }
271
272 node_t *search_node(list_t *lst, data_t s_data)
273 {
274     node_t *run;
275
276     for(run = lst->next; run != lst; run = run->next)
277         if(run->data == s_data)
278             return (run);
279
280     return (NULL);
281 }
282
283 static node_t *get_node(data_t new_data)
284 {
285     node_t *new_node = (node_t*)xmalloc(1, sizeof(node_t));
286     new_node->data = new_data;
287     return (new_node);
288 }
289
290 /* Auxillary routines */
291 static void *xmalloc(int nr_elements, int size_per_element)
292 {
293     void *p = calloc(nr_elements, size_per_element);
294     if(!p){
295         fprintf(stderr, "xmalloc:fatal:out of memory\n");
296         exit(EXIT_FAILURE);
297     }
298     return (p);
299 }
300
301
302
```