

```

1      #include <stdio.h>
2      #include <stdlib.h>
3      #include <assert.h>
4
5      #include "list.h"
6
7      /* interface functions */
8      list_t* create_list(void)
9      {
10         return get_node(0);
11     }
12
13     status_t insert_start(list_t* p_list, data_t new_data)
14     {
15         generic_insert(p_list, get_node(new_data), p_list->next);
16         return (SUCCESS);
17     }
18
19     status_t insert_end(list_t* p_list, data_t new_data)
20     {
21         node_t* p_run = NULL;
22
23         p_run = p_list;
24         while(p_run->next != NULL)
25             p_run = p_run->next;
26         generic_insert(p_run, get_node(new_data), p_run->next);
27         return (SUCCESS);
28     }
29
30     status_t insert_after(list_t* p_list, data_t e_data, data_t new_data)
31     {
32         node_t* p_enode = NULL;
33
34         p_enode = search_node(p_list, e_data);
35         if(p_enode == NULL)
36             return (LIST_DATA_NOT_FOUND);
37         generic_insert(p_enode, get_node(new_data), p_enode->next);
38         return (SUCCESS);
39     }
40
41     status_t insert_before(list_t* p_list, data_t e_data, data_t new_data)
42     {
43         node_t* p_enode = NULL;
44
45         p_enode = search_node(p_list, e_data);
46         if(p_enode == NULL)
47             return (LIST_DATA_NOT_FOUND);
48         generic_insert(p_enode->prev, get_node(new_data), p_enode);
49         return (SUCCESS);
50     }
51
52     status_t get_start(list_t* p_list, data_t* p_start_data)
53     {
54         if(is_empty(p_list) == TRUE)
55             return (LIST_EMPTY);
56         *p_start_data = p_list->next->data;
57         return (SUCCESS);
58     }
59
60     status_t get_end(list_t* p_list, data_t* p_end_data)
61     {
62         node_t* p_run = NULL;
63
64         if(is_empty(p_list) == TRUE)
65             return (LIST_EMPTY);
66
67         p_run = p_list;
68         while(p_run->next != NULL)
69             p_run = p_run->next;
70         *p_end_data = p_run->data;
71
72         return (SUCCESS);
73     }
74

```

```

74
75  ✓ status_t pop_start(list_t* p_list, data_t* p_start_data)
76  {
77      if(is_empty(p_list) == TRUE)
78          return (LIST_EMPTY);
79      *p_start_data = p_list->next->data;
80      generic_delete(p_list->next);
81      return (SUCCESS);
82  }
83
84  ✓ status_t pop_end(list_t* p_list, data_t* p_end_data)
85  {
86      node_t* p_run = NULL;
87
88      if(is_empty(p_list) == TRUE)
89          return (LIST_EMPTY);
90
91      p_run = p_list;
92      while(p_run->next != NULL)
93          p_run = p_run->next;
94
95      *p_end_data = p_run->data;
96      generic_delete(p_run);
97      return (SUCCESS);
98  }
99
100  ✓ status_t remove_start(list_t* p_list)
101  {
102      if(is_empty(p_list) == TRUE)
103          return (LIST_EMPTY);
104      generic_delete(p_list->next);
105      return (SUCCESS);
106  }
107
108  ✓ status_t remove_end(list_t* p_list)
109  {
110      node_t* p_run = NULL;
111
112      if(is_empty(p_list) == TRUE)
113          return (LIST_EMPTY);
114
115      p_run = p_list;
116      while(p_run->next != NULL)
117          p_run = p_run->next;
118      generic_delete(p_run);
119
120      return (SUCCESS);
121  }
122
123  ✓ status_t remove_data(list_t* p_list, data_t r_data)
124  {
125      node_t* p_node = NULL;
126
127      p_node = search_node(p_list, r_data);
128      if(p_node == NULL)
129          return (LIST_DATA_NOT_FOUND);
130      generic_delete(p_node);
131
132      return (SUCCESS);
133  }
134
135  ✓ status_t is_empty(list_t* p_list)
136  {
137      return (p_list->next == NULL && p_list->prev == NULL);
138  }
139
140  ✓ status_t find(list_t* p_list, data_t f_data)
141  {
142      node_t* p_node = NULL;
143
144      p_node = search_node(p_list, f_data);
145      return (p_node != NULL);
146  }

```

```

148     len_t get_length(list_t* p_list)
149     {
150         node_t* p_node = NULL;
151         len_t len = 0;
152
153         for(p_node = p_list->next; p_node != NULL; p_node = p_node->next, ++len)
154             ;
155         return (len);
156     }
157
158     void show(list_t* p_list, const char* msg)
159     {
160         node_t* p_run = NULL;
161
162         if(msg)
163             puts(msg);
164
165         printf("[START]<-->");
166         p_run = p_list->next;
167         while(p_run != NULL)
168         {
169             printf("[%d]<-->", p_run->data);
170             p_run = p_run->next;
171         }
172         puts("[END]");
173     }
174
175     status_t destroy_list(list_t** pp_list)
176     {
177         node_t* p_run = NULL;
178         node_t* p_run_next = NULL;
179         list_t* p_list = NULL;
180
181         p_list = *pp_list;
182         p_run = p_list->next;
183         while(p_run != NULL)
184         {
185             p_run_next = p_run->next;
186             free(p_run);
187             p_run = p_run_next;
188         }
189
190         free(p_list);
191         p_list = NULL;
192         *pp_list = NULL;
193
194         return (SUCCESS);
195     }
196
197     /* concat immutable */
198     list_t* concat_lists_imm(list_t* p_list_1, list_t* p_list_2)
199     {
200         list_t* p_concat_list = NULL;
201         node_t* p_run = NULL;
202
203         p_concat_list = create_list();
204         for(p_run = p_list_1->next; p_run != NULL; p_run = p_run->next)
205             assert(insert_end(p_concat_list, p_run->data) == SUCCESS);
206
207         for(p_run = p_list_2->next; p_run != NULL; p_run = p_run->next)
208             assert(insert_end(p_concat_list, p_run->data) == SUCCESS);
209
210         return (p_concat_list);
211     }

```



```

212 /* concat mutable */
213
214 status_t concat_list_m(list_t* p_list_1, list_t** pp_list_2)
215 {
216     list_t* p_list_2 = NULL;
217     node_t* p_run = NULL;
218
219     p_list_2 = *pp_list_2;
220     if(is_empty(p_list_2) == TRUE)
221     {
222         free(p_list_2);
223         p_list_2 = NULL;
224         *pp_list_2 = NULL;
225         return (SUCCESS);
226     }
227
228     p_run = p_list_1;
229     while(p_run->next != NULL)
230         p_run = p_run->next;
231
232     p_run->next = p_list_2->next;
233     p_list_2->next->prev = p_run;
234
235     free(p_list_2);
236     p_list_2 = NULL;
237     *pp_list_2 = NULL;
238
239     return (SUCCESS);
240 }
241
242 /* merge sorted lists */
243 list_t* merge_lists(list_t* p_list_1, list_t* p_list_2)
244 {
245     node_t* p_run_1 = NULL;
246     node_t* p_run_2 = NULL;
247     list_t* p_merged_list = NULL;
248
249     p_merged_list = create_list();
250     p_run_1 = p_list_1->next;
251     p_run_2 = p_list_2->next;
252
253     while(TRUE)
254     {
255         if(p_run_1 == NULL)
256         {
257             while(p_run_2 != NULL)
258             {
259                 assert(insert_end(p_merged_list, p_run_2->data) == SUCCESS);
260                 p_run_2 = p_run_2->next;
261             }
262             break;
263         }
264
265         if(p_run_2 == NULL)
266         {
267             while(p_run_1 != NULL)
268             {
269                 assert(insert_end(p_merged_list, p_run_1->data) == SUCCESS);
270                 p_run_1 = p_run_1->next;
271             }
272             break;
273         }
274
275         if(p_run_1->data <= p_run_2->data)
276         {
277             assert(insert_end(p_merged_list, p_run_1->data) == SUCCESS);
278             p_run_1 = p_run_1->next;
279         }
280         else
281         {
282             assert(insert_end(p_merged_list, p_run_2->data) == SUCCESS);
283             p_run_2 = p_run_2->next;
284         }
285     }
286
287     return (p_merged_list);
288 }
289

```

```

289     /* reversal */
290     /* immutable */
291     list_t* get_reversed_list(list_t* p_list) /* immutable version */
292     {
293         list_t* p_reversed_list = NULL;
294         node_t* p_run = NULL;
295
296         p_reversed_list = create_list();
297         if(is_empty(p_list))
298             return (p_reversed_list);
299
300         p_run = p_list->next;
301         while(p_run != NULL)
302         {
303             assert(insert_start(p_reversed_list, p_run->data) == SUCCESS);
304             p_run = p_run->next;
305         }
306
307         return (p_reversed_list);
308     }
309
310
311     status_t reverse_list(list_t* p_list) /* mutable version */
312     {
313         node_t* p_run = NULL;
314         node_t* p_run_prev = NULL;
315         node_t* p_original_last = NULL;
316         node_t* p_current_last = NULL;
317
318         p_original_last = p_list;
319         while(p_original_last->next != NULL)
320             p_original_last = p_original_last->next;
321
322         p_run = p_original_last->prev;
323         p_current_last = p_original_last;
324         while(p_run && p_run != p_list)
325         {
326             p_run_prev = p_run->prev;
327             p_current_last->next = p_run;
328             p_run->prev = p_current_last;
329             p_current_last = p_run;
330             p_current_last->next = NULL;
331             p_run = p_run_prev;
332         }
333
334         if(p_list != p_original_last)
335         {
336             p_list->next = p_original_last;
337             p_original_last->prev = p_list;
338         }
339
340         return (SUCCESS);
341     }
342
343     /* list axuillary functions */
344
345     static void generic_insert(node_t* p_beg, node_t* p_mid, node_t* p_end)
346     {
347         p_mid->next = p_end;
348         p_mid->prev = p_beg;
349         if(p_beg != NULL)
350             p_beg->next = p_mid;
351         if(p_end != NULL)
352             p_end->prev = p_mid;
353     }
354
355     static void generic_delete(node_t* p_delete_node)
356     {
357         if(p_delete_node == NULL)
358             return;
359
360         if(p_delete_node->next != NULL)
361             p_delete_node->next->prev = p_delete_node->prev;
362         if(p_delete_node->prev != NULL)
363             p_delete_node->prev->next = p_delete_node->next;
364
365         free(p_delete_node);
366         p_delete_node = NULL;
367     }
368

```

```

368
369  ✓ static node_t* search_node(list_t* p_list, data_t s_data)
370  {
371      node_t* p_run = NULL;
372
373      p_run = p_list->next;
374  ✓  while(p_run != NULL)
375      {
376          if(p_run->data == s_data)
377              return (p_run);
378          p_run = p_run->next;
379      }
380
381      return (NULL);
382  }
383
384  ✓ static node_t* get_node(data_t new_data)
385  {
386      node_t* p_node = NULL;
387
388      p_node = (node_t*)xmalloc(1, sizeof(node_t));
389      p_node->data = new_data;
390      p_node->prev = NULL;
391      p_node->next = NULL;
392
393      return (p_node);
394  }
395
396  /* auxillary function */
397  ✓ static void* xmalloc(size_t nr_elements, size_t size_per_element)
398  {
399      void* p = NULL;
400
401      p = calloc(nr_elements, size_per_element);
402  ✓  if(p == NULL)
403      {
404          fprintf(stderr, "calloc:fatal:out of memory\n");
405          exit(EXIT_FAILURE);
406      }
407
408      return (p);
409  }
410
411  ✓ status_t remove_all(list_t* p_list, data_t remove_data)
412  {
413      node_t* p_run = NULL;
414      node_t* p_run_next = NULL;
415
416      p_run = p_list->next;
417  ✓  while(p_run != NULL)
418      {
419          p_run_next = p_run->next;
420          if(p_run->data == remove_data)
421              generic_delete(p_run);
422          p_run = p_run_next;
423      }
424
425      return (SUCCESS);
426  }

```