

```
1  /* Header files */
2  ✓ #include<stdio.h>
3  | #include<stdlib.h>
4  | #include<assert.h>
5  | #include<string.h>
6
7  /* Symbolic constants */
8  #define SUCCESS          1
9  #define TRUE             1
10 #define FALSE            0
11 #define LIST_DATA_NOT_FOUND 2
12 #define LIST_EMPTY        3
13
14 /* Node layout definition */
15 ✓ struct node{
16     |     int data;
17     |     struct node* next;
18 };
19
20 /* Interface functions declarations */
21 /* List creation function */
22 struct node* create_list(void);
23
24 /* Data addition functions */
25 int insert_start(struct node* p_list, int new_data);
26 int insert_end(struct node* p_list, int new_data);
27 int insert_after(struct node* p_list, int existing_data, int new_data);
28 int insert_before(struct node* p_list, int existing_data, int new_data);
29
30 /* Get functions */
31 int get_start(struct node* p_list, int* p_start_data);
32 int get_end(struct node* p_list, int* p_end_data);
33
34 /* Pop Functions */
35 int pop_start(struct node* p_list, int* p_start_data);
36 int pop_end(struct node* p_list, int* p_end_data);
37
38 /* Remove functions */
39 int remove_start(struct node* p_list);
40 int remove_end(struct node* p_list);
41 int remove_data(struct node* p_list, int r_data);
42
43 /* Miscellaneous functions */
44 int find(struct node* p_list, int f_data);
45 int get_list_length(struct node* p_list);
46 int is_list_empty(struct node* p_list);
47 void show_list(struct node* p_list, const char* msg);
48
49 /* List destruction function */
50 int destroy_list(struct node* p_list);
51
```

```

52  /* Client of linked list */
53
54  int main(void){
55      struct node* p_list = NULL;
56
57      int status;
58      int data, start_data, end_data;
59      int length;
60
61      static const char* line = "-----";
62
63      p_list = create_list();
64      assert(p_list != NULL);
65      printf("List created successfully\n");
66      puts(line);
67
68      printf("Testing assertions on the empty list\n");
69      assert(is_list_empty(p_list) == TRUE);
70      assert(get_list_length(p_list) == 0);
71      assert(get_start(p_list, &start_data) == LIST_EMPTY);
72      assert(get_end(p_list, &end_data) == LIST_EMPTY);
73      assert(pop_start(p_list, &start_data) == LIST_EMPTY);
74      assert(pop_end(p_list, &end_data) == LIST_EMPTY);
75      assert(remove_start(p_list) == LIST_EMPTY);
76      assert(remove_end(p_list) == LIST_EMPTY);
77      printf("All assertions on the empty list are successful\n");
78      puts(line);
79
80      show_list(p_list, "Showing empty list immediately_after creation:");
81      puts(line);
82
83      for(data = 0; data < 5; ++data)
84      {
85          status = insert_start(p_list, data * 10);
86          assert(status == SUCCESS);
87          printf("%d inserted successfully at the start of the list\n", data*10);
88      }
89      show_list(p_list, "Showing list after inserting 5 data elements at the start:");
89      puts(line);
90
91      for(data = 1; data <= 5; ++data)
92      {
93          status = insert_end(p_list, data * 5);
94          assert(status == SUCCESS);
95          printf("%d inserted successfully at the end of the list\n", data * 5);
96      }
97      show_list(p_list, "Showing list after inserting 5 data elements at the end:\n");
98      puts(line);
99
100     status = insert_after(p_list, -5, 100);
101     assert(status == LIST_DATA_NOT_FOUND);
102     printf("Expected failure to insert data 100 after non-existent data -5\n");
103     puts(line);
104
105     status = insert_after(p_list, 0, 100);
106     assert(status == SUCCESS);
107     show_list(p_list, "Showing list after successfully inserting 100 after 0:");
108
109     status = insert_before(p_list, 43, 200);
110     assert(status == LIST_DATA_NOT_FOUND);

```

```
112 printf("Expected failure to insert data 200 after non-existent data 43\n");
113 puts(line);
114
115 status = insert_before(p_list, 0, 200);
116 assert(status == SUCCESS);
117 show_list(p_list, "Showing list after successfully inserting data 200 before 0:");
118 puts(line);
119
120 status = get_start(p_list, &start_data);
121 assert(status == SUCCESS);
122 printf("Data at the start:%d\n", start_data);
123 show_list(p_list, "Showing list to demonstrate that get_start() returns start data without removing it:");
124 puts(line);
125
126 status = get_end(p_list, &end_data);
127 assert(status == SUCCESS);
128 printf("Data at the end:%d\n", end_data);
129 show_list(p_list, "Showing list to demonstrate that get_end() returns the end data without removing it:");
130 puts(line);
131
132 status = pop_start(p_list, &start_data);
133 assert(status == SUCCESS);
134 printf("Data at the start = %d\n", start_data);
135 show_list(p_list, "Showing list to demonstrate that pop_start() removes and returns the start data:");
136 puts(line);
137
138 status = pop_end(p_list, &end_data);
139 assert(status == SUCCESS);
140 printf("Data at the end = %d\n", end_data);
141 show_list(p_list, "Showing list to demonstrate that pop_end() removes and returns the end data:");
142 puts(line);
143
144 status = remove_start(p_list);
145 assert(status == SUCCESS);
146 show_list(p_list, "Showing list after remove_start():");
147 puts(line);
148
149 status = remove_end(p_list);
150 assert(status == SUCCESS);
151 show_list(p_list, "Showing list after remove_end():");
152 puts(line);
153
154 status = remove_data(p_list, 78);
155 assert(status == LIST_DATA_NOT_FOUND);
156 printf("Expected error in removing non-existent data 78\n");
157 puts(line);
158
159 status = remove_data(p_list, 0);
160 assert(status == SUCCESS);
161 show_list(p_list, "Showing list after removing existing data 0\n");
162 puts(line);
163
164 status = find(p_list, 91);
165 assert(status == FALSE);
166 printf("Expected return value FALSE from find() for non-existent data 91\n");
167 puts(line);
168
169 status = find(p_list, 100);
170 assert(status == TRUE);
171 printf("Expected return value TRUE from find() for existing data 30\n");
```

```
172     puts(line);
173
174     status = is_list_empty(p_list);
175     assert(status == FALSE);
176     printf("Expected return value FALSE from is_list_empty()\n");
177     puts(line);
178
179     length = get_list_length(p_list);
180     printf("Length of the list = %d\n", length);
181     puts(line);
182
183     status = destroy_list(p_list);
184     assert(status == SUCCESS);
185     p_list = NULL;
186     printf("List is destroyed successfully\n");
187     puts(line);
188
189     return(0);
190 }
191
192 /* Server of linked list */
193 /* Interface functions declarations */
194 /* List creation function */
195 struct node* create_list(void)
196 {
197     struct node* head_node = NULL;
198
199     head_node = (struct node*)malloc(sizeof(struct node));
200     if(NULL == head_node)
201     {
202         puts("out of memory");
203         exit(EXIT_FAILURE);
204     }
205
206     head_node->data = 0;
207     head_node->next = NULL;
208
209     return (head_node);
210 }
211
212 /* Data addition functions */
213 int insert_start(struct node* p_list, int new_data)
214 {
215     struct node* new_node = NULL;
216
217     new_node = (struct node*)malloc(sizeof(struct node));
218     if(NULL == new_node)
219     {
220         puts("out of memory");
221         exit(EXIT_FAILURE);
222     }
223
224     new_node->data = new_data;
225     new_node->next = p_list->next;
226     p_list->next = new_node;
227
228     return (SUCCESS);
229 }
230
```

```
231     v int insert_end(struct node* p_list, int new_data)
232     {
233         struct node* run = NULL;
234         struct node* new_node = NULL;
235
236         // Step 1: Allocate and initialize new node
237         new_node = (struct node*)malloc(sizeof(struct node));
238         if(NULL == new_node)
239         {
240             puts("out of memory");
241             exit(EXIT_FAILURE);
242         }
243
244         new_node->data = new_data;
245         new_node->next = NULL;
246
247         // Step 2: Locate the last node
248         run = p_list;
249         while(run->next != NULL)
250         {
251             run = run->next;
252         }
253
254         // Step 3: Append the new node at the last position
255         run->next = new_node;
256
257         return (SUCCESS);
258     }
259
260     v int insert_after(struct node* p_list, int existing_data, int new_data)
261     {
262         struct node* existing_node = NULL;
263         struct node* new_node = NULL;
264         struct node* run = NULL;
265
266         // Step 1: Search for the node containing the first occurrence of the existing data
267         run = p_list->next;
268         while(run != NULL)
269         {
270             v if(run->data == existing_data)
271             {
272                 break;
273             }
274             run = run->next;
275         }
276
277         // If existing data is not found then return error as such
278         if(NULL == run)
279             return (LIST_DATA_NOT_FOUND);
280
281
282         // Step 2: Allocate and initialize new node
283         existing_node = run;
284         new_node = (struct node*)malloc(sizeof(struct node));
285         if(NULL == new_node)
286         {
287             puts("out of memory");
288             exit(EXIT_FAILURE);
289         }
290
291         new_node->data = new_data;
292
293         // Step 3: Insert the new node at its appropriate position
294         new_node->next = existing_node->next;
295         existing_node->next = new_node;
296
297         return (SUCCESS);
298     }
```

```
300  v int insert_before(struct node* p_list, int existing_data, int new_data)
301  {
302      struct node* run = NULL;
303      struct node* run_previous = NULL;
304      struct node* new_node = NULL;
305
306      // Step 1: Search for the first occurrence of the existing data
307      // maintaining the back pointer
308      run_previous = p_list;
309      run = p_list->next;
310      while(run != NULL)
311      {
312          if(run->data == existing_data)
313          {
314              break;
315          }
316
317          run_previous = run;
318          run = run->next;
319      }
320
321      // if the existing data is not found throughout the list then return an error
322      // saying as such
323      if(run == NULL)
324          return (LIST_DATA_NOT_FOUND);
325
326      // Step 2: Allocate and initialize the new node
327      new_node = (struct node*)malloc(sizeof(struct node));
328      if(NULL == new_node)
329      {
330          puts("out of memory");
331          exit(EXIT_FAILURE);
332      }
333
334      new_node->data = new_data;
335      // Step 3: Insert the new node at the appropriate position
336      new_node->next = run;
337      run_previous->next = new_node;
338
339      return (SUCCESS);
340  }
341
342  /* Get functions */
343  v int get_start(struct node* p_list, int* p_start_data)
344  {
345      if(p_list->next == NULL)
346          return (LIST_EMPTY);
347
348      *p_start_data = p_list->next->data;
349
350      return (SUCCESS);
351  }
352
353  v int get_end(struct node* p_list, int* p_end_data)
354  {
355      struct node* run = NULL;
356
357      if(p_list->next == NULL)
358          return (LIST_EMPTY);
359
360      run = p_list->next;
361      while(run->next != NULL)
362      {
363          run = run->next;
364      }
365
366      *p_end_data = run->data;
367
368      return (SUCCESS);
369  }
370
```

```
370      /* Pop Functions */
371  ✓ int pop_start(struct node* p_list, int* p_start_data)
372  {
373      struct node* delete_previous = NULL;
374      struct node* delete_node = NULL;
375      struct node* delete_next = NULL;
376
377      if(p_list->next == NULL)
378          return (LIST_EMPTY);
379
380      *p_start_data = p_list->next->data;
381
382      delete_previous = p_list;
383      delete_node = p_list->next;
384      delete_next = p_list->next->next;
385
386      delete_previous->next = delete_next;
387
388      free(delete_node);
389      delete_node = NULL;
390
391      return (SUCCESS);
392  }
393
394
395  ✓ int pop_end(struct node* p_list, int* p_end_data)
396  {
397      struct node* run = NULL;
398      struct node* run_previous = NULL;
399
400      if(p_list->next == NULL)
401          return (LIST_EMPTY);
402
403      run_previous = p_list;
404      run = p_list->next;
405      while(run->next != NULL)
406      {
407          run_previous = run;
408          run = run->next;
409      }
410
411      *p_end_data = run->data;
412
413      free(run);
414      run = NULL;
415      run_previous->next = NULL;
416
417      return (SUCCESS);
418  }
419
420      /* Remove Functions */
421  ✓ int remove_start(struct node* p_list)
422  {
423      struct node* delete_previous = NULL;
424      struct node* delete_node = NULL;
425      struct node* delete_next = NULL;
426
427      if(p_list->next == NULL)
428          return (LIST_EMPTY);
429
430      delete_previous = p_list;
431      delete_node = p_list->next;
432      delete_next = p_list->next->next;
433
434      delete_previous->next = delete_next;
435
436      free(delete_node);
437      delete_node = NULL;
438
439      return (SUCCESS);
440  }
441
```

```
440 }  
441  
442     ✓ int remove_end(struct node* p_list)  
443     {  
444         struct node* run = NULL;  
445         struct node* run_previous = NULL;  
446  
447         if(p_list->next == NULL)  
448             return (LIST_EMPTY);  
449  
450         run_previous = p_list;  
451         run = p_list->next;  
452         while(run->next != NULL)  
453         {  
454             run_previous = run;  
455             run = run->next;  
456         }  
457  
458         free(run);  
459         run = NULL;  
460         run_previous->next = NULL;  
461  
462         return (SUCCESS);  
463     }  
464  
465     ✓ int remove_data(struct node* p_list, int r_data)  
466     {  
467         struct node* run = NULL;  
468         struct node* run_previous = NULL;  
469  
470         run_previous = p_list;  
471         run = p_list->next;  
472         while(run != NULL)  
473         {  
474             if(run->data == r_data)  
475                 break;  
476  
477             run_previous = run;  
478             run = run->next;  
479         }  
480  
481         if(run == NULL)  
482             return (LIST_DATA_NOT_FOUND);  
483  
484         run_previous->next = run->next;  
485         free(run);  
486         run = NULL;  
487  
488         return (SUCCESS);  
489     }  
490  
491     /* Miscellaneous functions */  
492     ✓ int find(struct node* p_list, int f_data)  
493     {  
494         struct node* run = NULL;  
495  
496         run = p_list->next;  
497         while(run != NULL)  
498         {  
499             if(run->data == f_data)  
500                 return (TRUE);  
501             run = run->next;  
502         }  
503  
504         return (FALSE);  
505     }
```

```
505     }
506
507     int get_list_length(struct node* p_list)
508     {
509         int len = 0;
510         struct node* run = NULL;
511
512         run = p_list->next;
513         while(run != NULL)
514         {
515             len = len + 1;
516             run = run->next;
517         }
518
519         return (len);
520     }
521
522     int is_list_empty(struct node* p_list)
523     {
524         return (p_list->next == NULL);
525     }
526
527     void show_list(struct node* p_list, const char* msg)
528     {
529         struct node* run = NULL;
530
531         if(msg != NULL)
532             puts(msg);
533
534         printf("[START]->");
535         run = p_list->next;
536         while(run != NULL)
537         {
538             printf("[%d]->", run->data);
539             run = run->next;
540         }
541         printf("[END]\n");
542     }
543
544     /* List destruction function */
545     int destroy_list(struct node* p_list)
546     {
547         struct node* run = NULL;
548         struct node* run_next = NULL;
549
550         run = p_list;
551         while(run != NULL)
552         {
553             run_next = run->next;
554             free(run);
555             run = run_next;
556         }
557
558         return (SUCCESS);
559     }
560 }
```