```c
#include <stdlib.h>
#include "llist.h"
#include "string.h"
#include <stdio.h>

void llInit(list *myList){
    myList->head = NULL;
    myList->tail = NULL;
}

int llSize(list *myList){
    int size = 0;
    node* workingNode = myList->head;
    while(workingNode != NULL){
        size++;
        workingNode = workingNode->next;
    }
    return size;
}

int llPushFront(list *myList,char *toStore){
    if(toStore != NULL){
        char *listString = malloc (strlen(toStore)+1);
        strcpy(listString, toStore);
        node* newNode = malloc (sizeof (node));
        newNode->prev = NULL;
        newNode->string = listString;
        newNode->next = myList->head;
        if(myList->head != NULL){
            myList->head->prev = newNode;
        }
        myList->head = newNode;
        if(myList->tail == NULL){
            myList->tail = newNode;
        }
        return 1;
    } else {
        return 0;
    }
}

int llPopFront(list *myList){
    if(myList->head != NULL){
        node* oldHead = myList->head;
        if(oldHead->next != NULL){
            myList->head = myList->head->next;
```

```
47              myList->head->prev = NULL;
48          } else {
49              myList->head = NULL;
50              myList->tail = NULL;
51          }
52          free(oldHead->string);
53          free(oldHead);
54          return 1;
55      } else {
56          return 0;
57      }
58  }
59
60  int llPushBack(list *myList, char *toStore){
61      if(toStore != NULL){
62          char *listString = malloc (strlen(toStore)+1);
63          strcpy(listString, toStore);
64          node* newNode = malloc (sizeof (node));
65          newNode->prev = myList->tail;
66          newNode->string = listString;
67          newNode->next = NULL;
68          if(myList->tail != NULL){
69              myList->tail->next = newNode;
70          }
71          myList->tail = newNode;
72          if(myList->head == NULL){
73              myList->head = newNode;
74          }
75          return 1;
76      } else {
77          return 0;
78      }
79  }
80
81  int llPopBack(list *myList){
82      if(myList->tail != NULL){
83          node* oldTail = myList->tail;
84          if(oldTail->prev != NULL){
85              myList->tail = myList->tail->prev;
86              myList->tail->next = NULL;
87          } else {
88              myList->tail = NULL;
89              myList->head = NULL;
90          }
91          free(oldTail->string);
92          free(oldTail);
```

```
 93          return 1;
 94      } else {
 95          return 0;
 96      }
 97  }
 98
 99  void llClear(list *myList){
100      int status = 1;
101      while(status != 0){
102          status = llPopBack(myList);
103      }
104  }
105
106  int llInsertAfter(list* myList, node *insNode, char *
     toStore){
107      if((insNode == NULL) || (toStore == NULL)){
108          return 0;
109      } else {
110          char *listString = malloc (strlen(toStore)+1);
111          strcpy(listString, toStore);
112          node* newNode = malloc (sizeof (node));
113          newNode->prev = insNode;
114          newNode->string = listString;
115          if(insNode == myList->tail){
116              myList->tail->next = newNode;
117              newNode->next = NULL;
118              myList->tail = newNode;
119          } else {
120              newNode->next = insNode->next;
121              insNode->next = newNode;
122          }
123          return 1;
124      }
125  }
126
127  int llInsertBefore(list* myList, node *insNode, char *
     toStore){
128      if((insNode == NULL) || (toStore == NULL)){
129          return 0;
130      } else {
131          char *listString = malloc (strlen(toStore)+1);
132          strcpy(listString, toStore);
133          node* newNode = malloc (sizeof (node));
134          newNode->next = insNode;
135          newNode->string = listString;
136          if(insNode == myList->head){
```

```
137             myList->head->prev = newNode;
138             newNode->prev = NULL;
139             myList->head = newNode;
140         } else {
141             newNode->prev = insNode->prev;
142             insNode->prev = newNode;
143         }
144         return 1;
145     }
146 }
147
148 int llRemove(list* myList, node *rmvNode){
149     node* workingNode = myList->head;
150     if(!(rmvNode == NULL)){
151         while(workingNode != NULL){
152             if(workingNode == rmvNode){
153                 if((workingNode == myList->head) && (
    workingNode == myList->tail)){
154                     myList->head = NULL;
155                     myList->tail = NULL;
156                 } else if(workingNode == myList->head){
157                     workingNode->next->prev = NULL;
158                     myList->head = workingNode->next;
159                 } else if(workingNode == myList->tail){
160                     workingNode->prev->next = NULL;
161                     myList->tail = workingNode->prev;
162                 } else {
163                     workingNode->prev->next = workingNode
    ->next;
164                     workingNode->next->prev = workingNode
    ->prev;
165                 }
166                 free(workingNode->string);
167                 free(workingNode);
168                 break;
169             } else {
170                 workingNode = workingNode->next;
171             }
172         }
173         return 1;
174     } else {
175         return 0;
176     }
177 }
178
```

```
 1 #ifndef LLIST_H
 2 #define LLIST_H
 3 /* llist.h
 4  *
 5  * External (public) declarations for simple doubly-linked
   list in C.
 6  *
 7  * This list will know head and tail, and will be capable
   of forward
 8  * and reverse iteration.  The tail node should always
   assign its next
 9  * pointer to NULL to indicate the end of the list, and
   likewise, the
10  * head node should assign its previous pointer to NULL for
    the same
11  * reason.
12  *
13  * Note that the pop operations do not return a reference
   to the
14  * popped node.  This would require storage for the node to
    be
15  * released by the user, which could lead to memory
   mishandling.
16  * The user will need to use head or tail pointers to
   interact with the
17  * node prior to popping it.
18  *
19  * This list will only hold strings (char arrays) for
   simplicity, and
20  * will have ownership of the strings.  This means the
   string will
21  * need to be copied into memory under control of the list.
22  *
23  */
24
25 /* Structures */
26
27 /* a node - cannot be anonymous because we need a node *
   inside, but
28  * we can make the struct name and the typedef the same */
29 typedef struct node {
30     char *string;
31     struct node* next;  /* need struct here because inside
   typedef */
32     struct node* prev;  /* need struct here because inside
   typedef */
```

```
33 } node;
34
35 /* The list itself - struct can be anonymous */
36 typedef struct {
37     node *head;  /* could have been struct node* as well */
38     node *tail;
39 } list;
40
41 /* List methods
42  *
43  * These methods are used to create and operate on a list
   as a whole.
44  */
45
46 /* llInit()
47  *     Initialize a list structure.  An empty list will
48  *     be characterized by head and tail pointer both being
    NULL.
49  *         Parameters:  myList - a pointer to the structure
    to be init
50  *         Returns: void
51  */
52 void llInit(list *myList);
53
54 /* llSize()
55  *     Reports the current size of the list.  Will need to
   iterate
56  *     the list to get this data size there is no size
   property, nor
57  *     can there really be one given that users can access
   nodes.
58  *         Parameters:  myList - the list
59  *         Returns: int, size of list
60  */
61 int llSize(list *myList);
62
63 /* llPushFront()
64  *     Add a new node with provided data and place node at
65  *     front of list.  The new node will replace the head
   node.
66  *     This method should check to make sure the provided
   char * is
67  *     not NULL.  If it is NULL, this method should do
   nothing and
68  *     make no changes to the list.  If it is not NULL, it
   can be
```

```
 69  *     assumed that it is a valid null-terminated string.
 70  *         Parameters:  myList - the list
 71  *                      toStore - the char array to store
 72  *         Returns: int - 0 if no push (toStore was NULL)
    or non-zero
 73  *                        if push successful
 74  */
 75 int llPushFront(list *myList,char *toStore);
 76
 77 /* llPopFront()
 78  *     Removes first item in list.  Note, this does not
    return
 79  *     any data from the list.  If the data in the node is
     needed
 80  *     it should be accessed prior to the pop (list->head
    ->string).
 81  *         Parameters:  myList - the list
 82  *         Returns: int - 0 if no pop (list was empty) or
    non-zero
 83  *                        if pop successful
 84  */
 85 int llPopFront(list *myList);
 86
 87 /* llPushBack()
 88  *     Add a new node with provided data and place node at
 89  *     end of list.  This new node will be the new tail
    node.
 90  *     This method should check to make sure the provided
    char * is
 91  *     not NULL.  If it is NULL, this method should do
    nothing and
 92  *     make no changes to the list.  If it is not NULL, it
     can be
 93  *     assumed that it is a valid null-terminated string.
 94  *         Parameters:  myList - the list
 95  *                      toStore - the char array to store
 96  *         Returns: int - 0 if no push (toStore was NULL)
    or non-zero
 97  *                        if push successful
 98  */
 99 int llPushBack(list *myList, char *toStore);
100
101 /* llPopBack()
102  *     Removes last item in list.  Note, this does not
    return
103  *     any data from the list.  If the data in the node is
```

```
103  needed
104  *     it should be accessed prior to the pop (list->tail
    ->string).
105  *         Parameters:  myList - the list
106  *         Returns: int - 0 if no pop (list was empty) or
    non-zero
107  *                        if pop successful
108  */
109 int llPopBack(list *myList);
110
111 /* llClear()
112  *     Clears all nodes and releases all dynamic memory.
    List
113  *     structure should be NULLed and can be reused.
114  *         Parameters:  myList - the list
115  *         Returns: nothing
116  */
117 void llClear(list *myList);
118
119
120 /* Node methods
121  *
122  * These methods allow iteration of nodes within the list
    .  A list
123  * reference is still needed if head or tail needs to be
    modified.
124  */
125
126 /* llInsertAfter()
127  *     Add a new node with provided data and place node
    after
128  *     provided node reference.
129  *     This method should check to make sure the provided
    char * is
130  *     not NULL.  If it is NULL, this method should do
    nothing and
131  *     make no changes to the list.  If it is not NULL, it
     can be
132  *     assumed that it is a valid null-terminated string.
133  *     If this method is called on the tail node, a change
     to the
134  *     list structure will need to be made.
135  *         Parameters:  myList - theList
136  *                      insNode - the node after which
    item is added
137  *                      toStore - the char array to store
```

```
138  *         Returns: int - 0 if no insert (toStore was NULL
     or insNode
139  *                                          is NULL)
140  *                        non-zero if insert successful
141  */
142 int llInsertAfter(list* myList, node *insNode, char *
    toStore);
143
144 /* llInsertBefore()
145  *     Add a new node with provided data and place node
    before
146  *     provided node reference.
147  *     This method should check to make sure the provided
    char * is
148  *     not NULL.  If it is NULL, this method should do
    nothing and
149  *     make no changes to the list.  If it is not NULL, it
     can be
150  *     assumed that it is a valid null-terminated string.
151  *     If this method is called on the head node, a change
     to the
152  *     list structure will need to be made.
153  *         Parameters:  myList - theList
154  *                      insNode - the node before which
    item is added
155  *                      toStore - the char array to store
156  *         Returns: int - 0 if no insert (toStore was NULL
     or insNode
157  *                                          is NULL)
158  *                        non-zero if insert successful
159  */
160 int llInsertBefore(list* myList, node *insNode, char *
    toStore);
161
162 /* llRemove()
163  *     Removes the node referenced.  Releases
164  *     all associated dynamic memory.
165  *     If this method is called the current head or tail
    node, changes
166  *     to the list structure may need to be made.
167  *         Parameters:  myList - the list
168  *                      rmvNode - the node prior to the
    node to be
169  *                                removed.
170  *         Returns: nothing
171  */
```

```
172 int llRemove(list* myList, node *rmvNode);
173 #endif
174
```

```
 1 CC=gcc
 2 CFLAGS=-c -Wall
 3 LDFLAGS=
 4 SOURCES=llist.c llTester.c
 5 OBJECTS=$(SOURCES:.c=.o)
 6 EXECUTABLE=test
 7
 8 all: $(SOURCES) $(EXECUTABLE)
 9
10 $(EXECUTABLE): $(OBJECTS)
11     $(CC) $(LDFLAGS) $(OBJECTS) -o $@
12
13 .c.o:
14     $(CC) $(CFLAGS) $< -o $@
15
16 clean:
17     rm -rf $(OBJECTS) $(EXECUTABLE)
18
```

```
 1 /*
 2 1.All of the list functions need a “reference” to the list
   structure, and according to this design, that list
   reference
 3 is passed as a pointer. Why is this necessary? Do all of
   the list functions need this to be passed as a pointer?
 4 Any exceptions? Be specific in your answer.
 5
 6 All the list functions that we used in this lab required
   passing their arguments by pointer. In our case, the list
 7 specifically needed to be passed by pointer because it was
   an instance of a struct and in order to avoid duplicating
 8 the contents of the struct on the stack (passing by value)
   a pointer was needed. The same argument is also made for
 9 passing nodes. Another side benefit of passing node structs
    by reference is that we can determine if their data is
10 equal by simple equality (node1 == node2?) which is
   convenient for our use. We also had to pass strings by
   pointer
11 because of the same aforementioned benefits but also
   because we don’t know the length of the string, and a
   pointer
12 will allow us to avoid that conundrum because C style
   strings are null terminated.
13 Overall, because the data we were passing were all either
   structs, or strings, pointers were preferred. If the data
14 were simple types such as characters or integers, pointers
   would have been unnecessary.
15
16 2.Unlike a Java or C++ implementation, this implementation
   cannot “hide” any of the internal structure of the list.
17 That is, users of the list could mess up the next and prev
   pointers if they are careless. Can you think of any way we
18 could hide the structure of the list to lessen the chances
   a user will mess up the list? Describe in brief detail.
19
20 Since there are no objects in C, you cannot embed the list
   in any wrappers. You could make its fields constant which
21 would make it so the user could not edit them, but that
   would also not allow methods to edit them either.
22 Given these considerations C is just a language that places
    a lot of trust in the user and is under the assumption
23 that they know what they are doing and will make educated
   decisions. Therefore there may be a way to “hide” the
24 structure of the list but it’s not obvious or seemingly
   straightforward.
```

```
25
26 3.What if all llClear() did was assign NULL to head and
   tail in the list structure and nothing else. Would the
   program
27 crash? Would there be any side effects? Try it and report
   results.
28
29 The program would not crash nor would there be any
   immediately noticeable side effects. Instead, the nodes and
30 associated strings would be lost in memory leaks which if
   left running over a long period of time with many resets
31 may result in the heap running out of allocable memory.
   This is why it's important to "free the malloc's".
32
33 4.This design requires the user to iterate the list
   somewhat manually as demonstrated in the sample driver.
   Propose the
34 design of an iterator for this list. What data items would
   the iterator need to store (in a structure, perhaps)? What
35 functions would the iterator supply?
36
37 The iterator could store the current position within the
   list as well as the node associated with that position.
38 Methods to set the position and increment/decrement the
   position would be useful. The intended usage would be
   easier
39 access to getting nodes within the list.
40
41 My Experience with the lab:
42
43 I think this lab went fairly well. Thankfully most of the
   code I wrote worked the first time without needed extensive
44 debugging. There were a few times where I caught edge cases
    that I forgot to handle which I had to go back and fix
45 (such as properly resetting the head and tail when removing
    the last element). Some of the helper methods I wrote to
46 verify operation did throw me for a loop since they were
   giving me output which was correct but I was
   misinterpreting
47 and that caused me to waste some time trying to debug
   working code. Valgrind was useful for checking the memory
   and gdp
48 was also useful for checking exactly which methods were
   causing errors.
49 As for working with Dynamic memory, I was familiar with it
   before so implementation was fairly quick, though I did
```

```
49 have
50 to look up the syntax for how to allocate enough memory for
    a struct.
51
52  */
53
54 #include <stdlib.h>
55 #include "llist.h"
56 #include <stdio.h>
57
58 /*
59  * An iterator allows for easy traversal over a list.
60  * It requires an associated list reference to opperate.
61  * the node value and index value retain where in the list
   the iterator is currently at.
62  */
63 typedef struct iterator {
64     node *node;
65     int index;
66     list *list;
67 } iterator;
68
69 /*
70  * sets up the iterator for the supplied list starting at
   the head
71  */
72 void initIterator(iterator* itr, list* myList){
73     itr->node = myList->head;
74     itr->index = 0;
75     itr->list = myList;
76 }
77
78 /*
79  * Gets the value of the node at the iterator's current
   location
80  */
81 char * getIteratorValue(iterator* itr){
82     return itr->node->string;
83 }
84
85 /*
86  * Gets the current index of the iterator
87  */
88 int getIteratorIndex(iterator* itr){
89     return itr->index;
90 }
```

```
 91
 92 /*
 93  * increments the iterator to the next node.
 94  * returns non-zero if sucessful and 0 if next node is
    NULL
 95  */
 96 int next(iterator* itr){
 97     if(itr->node->next != NULL){
 98         itr->node = itr->node->next;
 99         itr->index++;
100         return 1;
101     } else {
102         return 0;
103     }
104 }
105
106 /*
107  * decrements the iterator to the previous node.
108  * returns non-zero if sucessful and 0 if next node is
    NULL
109  */
110 int previous(iterator* itr){
111     if(itr->node->prev != NULL){
112         itr->node = itr->node->prev;
113         itr->index--;
114         return 1;
115     } else {
116         return 0;
117     }
118 }
119
120 /*
121  * Set's the iterator to the desired position in the list.
122  * returns non-zero if sussful and 0 if out of bounds.
123  * if return is zero, iterator will not change
124  */
125 int setPosition(iterator* itr, int position){
126     if(position > llSize(itr->list))
127         return 0;
128     if(itr->index != position){
129         if(itr->index < position){
130             while(itr->index < position){
131                 if(itr->node->next != NULL){
132                     itr->node = itr->node->next;
133                     itr->index++;
134                 } else {
```

```
135                     return 0;
136                 }
137             }
138         } else {
139             while(itr->index > position){
140                 if(itr->node->prev != NULL){
141                     itr->node = itr->node->prev;
142                     itr->index--;
143                 } else {
144                     return 0;
145                 }
146             }
147         }
148     }
149     return 1;
150 }
151 
152 static void printList(list* myList){
153     node* workingNode = myList->head;
154     int size = 0;
155     printf("[list data]\n");
156     while(workingNode != NULL){
157         printf("-node: %d val: %s\n",size,workingNode->
    string);
158         size++;
159         workingNode = workingNode->next;
160     }
161 }
162 
163 static void printHeadAndTail(list* myList){
164     if(!((myList->head == NULL) || (myList->tail == NULL
    )))
165         printf("head: %s, tail: %s\n",myList->head->string
    ,myList->tail->string);
166     else
167         printf("head: NULL, tail: NULL\n");
168 }
169 
170 static node * getNode(list* myList, int position){
171     node* nodeWalker = myList->head;
172     for(int i =0;i<position;i++){
173         nodeWalker = nodeWalker->next;
174     }
175     return nodeWalker;
176 }
177 
```

```
178 int main(void){
179     printf("------BEGIN NORMAL TESTS-----\n");
180     list myList;
181     llInit(&myList);
182     printf("initalized...\n");
183     llPushFront(&myList, "one");
184     printf("TEST 1: testing one element...\n");
185     printf("-size: %d\n",llSize(&myList));
186     printList(&myList);
187     printHeadAndTail(&myList);
188     printf("TEST 2: removing element...\n");
189     llRemove(&myList, getNode(&myList,0));
190     printf("-size: %d\n",llSize(&myList));
191     printList(&myList);
192     printHeadAndTail(&myList);
193     printf("TEST 3: creating list of 5 elements...\n");
194     llPushFront(&myList, "two");
195     llPushFront(&myList, "one");
196     llPushBack(&myList, "three");
197     llPushBack(&myList, "four");
198     llPushBack(&myList, "five");
199     printf("-size: %d\n",llSize(&myList));
200     printList(&myList);
201     printHeadAndTail(&myList);
202     printf("TEST 4: popping head and tail nodes\n");
203     llPopBack(&myList);
204     llPopFront(&myList);
205     printf("-size: %d\n",llSize(&myList));
206     printList(&myList);
207     printHeadAndTail(&myList);
208     printf("TEST 5: recreating list of 5 and removing
    middle node\n");
209     llPushFront(&myList, "one");
210     llPushBack(&myList, "five");
211     llRemove(&myList, getNode(&myList,2));
212     printf("-size: %d\n",llSize(&myList));
213     printList(&myList);
214     printHeadAndTail(&myList);
215     printf("TEST: 6 removing head and tail node\n");
216     llRemove(&myList, getNode(&myList,3));
217     llRemove(&myList, getNode(&myList,0));
218     printf("-size: %d\n",llSize(&myList));
219     printList(&myList);
220     printHeadAndTail(&myList);
221     printf("TEST 7: clearing list\n");
222     llClear(&myList);
```

```
223     printf("-size: %d\n",llSize(&myList));
224     printList(&myList);
225     printHeadAndTail(&myList);
226     printf("------NORMAL TESTS COMPLETE-----\n");
227     printf("\n");
228     printf("-----BEGIN ITERATOR TESTS------\n");
229     printf("Initalize list of 5 elements\n");
230     llPushFront(&myList, "two");
231     llPushFront(&myList, "one");
232     llPushBack(&myList, "three");
233     llPushBack(&myList, "four");
234     llPushBack(&myList, "five");
235     printf("-size: %d\n",llSize(&myList));
236     printList(&myList);
237     printHeadAndTail(&myList);
238     printf("TEST 0: Initalize Iterator...\n");
239     iterator itr;
240     initIterator(&itr,&myList);
241     printf("value: %s index: %d \n",getIteratorValue(&itr
    ),getIteratorIndex(&itr));
242     printf("TEST 1: increment iterator\n");
243     next(&itr);
244     printf("value: %s index: %d \n",getIteratorValue(&itr
    ),getIteratorIndex(&itr));
245     printf("TEST 2: decrement iterator\n");
246     previous(&itr);
247     printf("value: %s index: %d \n",getIteratorValue(&itr
    ),getIteratorIndex(&itr));
248     printf("TEST 2.5: decrement below range\n");
249     previous(&itr);
250     printf("value: %s index: %d \n",getIteratorValue(&itr
    ),getIteratorIndex(&itr));
251     printf("TEST 3: set position to 3\n");
252     setPosition(&itr,3);
253     printf("value: %s index: %d \n",getIteratorValue(&itr
    ),getIteratorIndex(&itr));
254     printf("TEST 4: set position to out of bounds\n");
255     setPosition(&itr,10);
256     printf("value: %s index: %d \n",getIteratorValue(&itr
    ),getIteratorIndex(&itr));
257     printf("TEST 4: set position back to zero\n");
258     setPosition(&itr,0);
259     printf("value: %s index: %d \n",getIteratorValue(&itr
    ),getIteratorIndex(&itr));
260     printf("------ITERATOR TESTS COMPLETE-----\n");
261 }
```