

```

1
2 #ifndef LLIST H
3 #define LLIST H
4 /* llist.h
5  *
6  * External (public) declarations for simple doubly-linked list in C.
7  *
8  * This list will know head and tail, and will be capable of forward
9  * and reverse iteration. The tail node should always assign its next
10  * pointer to NULL to indicate the end of the list, and likewise, the
11  * head node should assign its previous pointer to NULL for the same
12  * reason.
13  *
14  * Note that the pop operations do not return a reference to the
15  * popped node. This would require storage for the node to be
16  * released by the user, which could lead to memory mishandling.
17  * The user will need to use head or tail pointers to interact with the
18  * node prior to popping it.
19  *
20  * This list will only hold strings (char arrays) for simplicity, and
21  * will have ownership of the strings. This means the string will
22  * need to be copied into memory under control of the list.
23  *
24  */
25
26 /* Structures */
27
28 /* a node - cannot be anonymous because we need a node * inside, but
29  * we can make the struct name and the typedef the same */
30 typedef struct node {
31     char *string;
32     struct node* next; /* need struct here because inside typedef */
33     struct node* prev; /* need struct here because inside typedef */
34 } node;
35
36 /* The list itself - struct can be anonymous */
37 typedef struct {
38     node *head; /* could have been struct node* as well */
39     node *tail;
40 } list;
41
42 /* List methods
43  *
44  * These methods are used to create and operate on a list as a whole.
45  */
46
47 /* llInit()
48  * Initialize a list structure. An empty list will
49  * be characterized by head and tail pointer both being NULL.
50  * Parameters: myList - a pointer to the structure to be init
51  * Returns: void
52  */
53 void llInit(list *myList);
54
55 /* llSize()
56  * Reports the current size of the list. Will need to iterate
57  * the list to get this data size there is no size property, nor
58  * can there really be one given that users can access nodes.
59  * Parameters: myList - the list
60  * Returns: int, size of list
61  */
62 int llSize(list *myList);
63
64 /* llPushFront()
65  * Add a new node with provided data and place node at
66  * front of list. The new node will replace the head node.
67  * This method should check to make sure the provided char * is
68  * not NULL. If it is NULL, this method should do nothing and
69  * make no changes to the list. If it is not NULL, it can be
70  * assumed that it is a valid null-terminated string.

```

```

71      *      Parameters: myList - the list
72      *                      toStore - the char array to store
73      *      Returns: int - 0 if no push (toStore was NULL) or non-zero
74      *                      if push successful
75      */
76      int llPushFront(list *myList, char *toStore);
77
78      /* llPopFront()
79      *      Removes first item in list. Note, this does not return
80      *      any data from the list. If the data in the node is needed
81      *      it should be accessed prior to the pop (list->head->string).
82      *      Parameters: myList - the list
83      *      Returns: int - 0 if no pop (list was empty) or non-zero
84      *                      if pop successful
85      */
86      int llPopFront(list *myList);
87
88      /* llPushBack()
89      *      Add a new node with provided data and place node at
90      *      end of list. This new node will be the new tail node.
91      *      This method should check to make sure the provided char * is
92      *      not NULL. If it is NULL, this method should do nothing and
93      *      make no changes to the list. If it is not NULL, it can be
94      *      assumed that it is a valid null-terminated string.
95      *      Parameters: myList - the list
96      *                      toStore - the char array to store
97      *      Returns: int - 0 if no push (toStore was NULL) or non-zero
98      *                      if push successful
99      */
100     int llPushBack(list *myList, char *toStore);
101
102     /* llPopBack()
103     *      Removes last item in list. Note, this does not return
104     *      any data from the list. If the data in the node is needed
105     *      it should be accessed prior to the pop (list->tail->string).
106     *      Parameters: myList - the list
107     *      Returns: int - 0 if no pop (list was empty) or non-zero
108     *                      if pop successful
109     */
110     int llPopBack(list *myList);
111
112     /* llClear()
113     *      Clears all nodes and releases all dynamic memory. List
114     *      structure should be NULLed and can be reused.
115     *      Parameters: myList - the list
116     *      Returns: nothing
117     */
118     void llClear(list *myList);
119
120
121     /* Node methods
122     *
123     * These methods allow iteration of nodes within the list. A list
124     * reference is still needed if head or tail needs to be modified.
125     */
126
127     /* llInsertAfter()
128     *      Add a new node with provided data and place node after
129     *      provided node reference.
130     *      This method should check to make sure the provided char * is
131     *      not NULL. If it is NULL, this method should do nothing and
132     *      make no changes to the list. If it is not NULL, it can be
133     *      assumed that it is a valid null-terminated string.
134     *      If this method is called on the tail node, a change to the
135     *      list structure will need to be made.
136     *      Parameters: myList - theList
137     *                      insNode - the node after which item is added
138     *                      toStore - the char array to store
139     *      Returns: int - 0 if no insert (toStore was NULL or insNode
140     *                      is NULL)

```

```
141      *                      non-zero if insert successful
142      */
143      int llInsertAfter(list* myList, node *insNode, char *toStore);
144
145      /* llInsertBefore()
146      *   Add a new node with provided data and place node before
147      *   provided node reference.
148      *   This method should check to make sure the provided char * is
149      *   not NULL. If it is NULL, this method should do nothing and
150      *   make no changes to the list. If it is not NULL, it can be
151      *   assumed that it is a valid null-terminated string.
152      *   If this method is called on the head node, a change to the
153      *   list structure will need to be made.
154      *   Parameters: myList - theList
155      *               insNode - the node before which item is added
156      *               toStore - the char array to store
157      *   Returns: int - 0 if no insert (toStore was NULL or insNode
158      *               is NULL)
159      *               non-zero if insert successful
160      */
161      int llInsertBefore(list* myList, node *insNode, char *toStore);
162
163      /* llRemove()
164      *   Removes the node referenced. Releases
165      *   all associated dynamic memory.
166      *   If this method is called the current head or tail node, changes
167      *   to the list structure may need to be made.
168      *   Parameters: myList - the list
169      *               rmvNode - the node prior to the node to be
170      *                       removed.
171      *   Returns: nothing
172      */
173      int llRemove(list* myList, node *rmvNode);
174
175      #endif
176
```

```

1  #include "llist.h"
2  #include <stdio.h>
3  #include <string.h>
4  #include <stdlib.h>
5  //Maximum Length
6  #define str len 100
7
8  void llInit(list *mylist) {
9
10     mylist-> head = NULL;
11     mylist-> tail = NULL;
12 }
13
14 int llSize(list *mylist) {
15     int count = 0;
16     for (node *n=mylist->head; n != NULL; n = n->next)
17     {
18         count++;
19     }
20     return count;
21 }
22
23 int llPushFront(list *myList, char *toStore){
24     //allocating memory for a node
25     node *new node= malloc(sizeof(node));
26     //allocating memory for string
27     new node->string=malloc(str len);
28     strcpy(new node->string,toStore);
29
30     //setting the node as head node
31     if(myList->head==NULL && myList->tail==NULL){
32         myList->head=new node;
33         myList->tail=new node;
34     }
35
36     else{
37         new node->next=myList->head;
38         myList->head->prev=new node;
39         new node->prev=NULL;
40         //update my head with the new node
41         myList->head= new node;
42     }
43
44     free(new node);
45     free(new node->string);
46 }
47
48 int llPopFront(list *myList){
49     if(myList==NULL) {
50         printf("List is Empty \n");
51         return 0;
52     }
53     //ptr to the head node
54     node *ptr = myList->head;
55     myList->head = myList->head->next;
56     //delete
57     free(ptr);
58 }
59
60 int llPushBack(list *myList, char *toStore){
61     //allocating memory for a node
62     node *new node= malloc(sizeof(node));
63     //allocating memory for string
64     new node->string=malloc(str len);
65     if(new node == NULL){
66         printf("Failed to allocate");
67         return 0;
68     }
69     strcpy(new node->string,toStore);
70     new_node->next=NULL;

```

```

71     new node->prev = myList->tail;
72     myList->tail->next=new node;
73     //set tail to the new node
74     myList->tail= new node;
75     free(new node);
76     free(new node->string);
77 }
78
79 int llPopBack(list *myList){
80
81     if(myList==NULL) {
82         printf("List is Empty \n");
83         return 0;
84     }
85
86     node *ptr= myList->tail;
87     myList->tail=myList->tail->prev;
88     myList->tail->next=NULL;
89     //delete
90     free(ptr);
91 }
92
93 void llClear(list *myList){
94
95     //get current head position
96     node *ptr= myList->head;
97     node *next;
98
99     while(ptr!=NULL){
100         next=ptr->next;
101         free(ptr);
102         ptr=next;
103     }
104     myList->head=NULL;
105 }
106
107 int llInsertAfter(list* myList, node *insNode, char *toStore){
108     //allocating memory for a node
109     node *new node= malloc(sizeof(node));
110     //allocating memory for string
111     new node->string=malloc(str len);
112     if(new node== NULL){
113         printf("Failed to allocate");
114         return 0;
115     }
116     strncpy(new node->string,toStore, str len);
117
118     if(myList->head==NULL && myList->tail==NULL){
119         myList->head=new node;
120         myList->tail=new node;
121     }
122     else if(insNode->next==NULL){
123         insNode->next=new node;
124         new node->prev=insNode;
125         new node->next=NULL;
126         myList->tail=new node;
127     }
128     else{
129         new node->next=insNode->next;
130         new node->prev=insNode;
131         insNode->next=new node;
132         insNode->next->prev=new node;
133     }
134     printf("\nllInsertAfter() OK!\n");
135     free(new node);
136     free(new node->string);
137 }
138
139 int llInsertBefore(list* myList, node *insNode, char *toStore){
140     //allocating memory for a node

```

```

141     node *new node= malloc(sizeof(node));
142     //allocating memory for string
143     new node->string=malloc(str len);
144     if(new node== NULL){
145         printf("Failed to allocate");
146         return 0;
147     }
148     strncpy(new node->string,toStore, str len);
149     if(myList->head==NULL && myList->tail==NULL){
150         myList->head=new node;
151         myList->tail=new node;
152     }
153     else if(insNode->prev == NULL){
154         insNode->prev=new node;
155         new node->next=insNode;
156         new node->prev=NULL;
157         myList->head=new node;
158     }
159     else{ new node->prev=insNode->prev;
160         insNode->prev->next=new node;
161         insNode->prev=new node;
162         new node->next=insNode;
163     }
164     printf("\nllInsertBefore() OK");
165     free(new node);
166     free(new node->string);
167 }
168 int llRemove(list* myList, node *rmvNode){
169     if(myList == NULL){
170         return 0;
171     }
172     //if node is at head
173     else if(rmvNode->prev==NULL) {
174         myList->head->next->prev = NULL;
175         myList->head = myList->head->next;
176     }
177     //if node is at tail
178     else if(rmvNode->next==NULL){
179         //ptr to tail node
180         rmvNode = myList->tail;
181         myList->tail->next=NULL;
182         myList->tail=myList->tail->prev;
183     }
184     else {
185         rmvNode->prev->next=rmvNode->next->prev;
186     }
187     free(rmvNode->string);
188     free(rmvNode);
189 }
190
191

```

```

1  /* 1.All of the list functions need a "reference" to the list structure, and
   according to this design, that list reference is passed as a pointer. Why is this
   necessary? Do all of the list functions need this to be passed as a pointer? Any
   exceptions? Be specific in your answer.
2  * Ans.) The structure uses pointers to access the nodes next, previous or the string.
3  * 2.Unlike a Java or C++ implementation, this implementation cannot "hide" any of
   the internal structure of the list. 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 could hide
   the structure of the list to lessen the chances a user will mess up the list?
   Describe in brief detail.
4  * Ans.) There is no encapsulation feature in C, but it can be performed in C also,
   I could think the only to solve this problem. For example, initialize struct node
   in one header file, and define the struct in another header file, and when you use
   the struct in a C file, this way the member declarations is unknown, the size is
   unknown.
5  * 3.What if all llClear() did was assign NULL to head and tail in the list
   structure and nothing else. Would the program crash? Would there be any side
   effects? Try it and report results.
6  * Ans) Yes, the program will crash and the result was Segmentation error
7  * 4.This design requires the user to iterate the list somewhat manually as
   demonstrated in the sample driver. Propose the design of an iterator for this list.
   What data items would the iterator need to store (in a structure, perhaps)? What
   functions would the iterator supply?
8  * Ans) the iterator would need a single pointer to a node and will point directly
   to the node that you want it to point.
9  */
10 /*
11 /* Experiences with this lab:
12 * working with segmentation faults and memory leaks was a challenging part
13 * gdb was very useful for debugging, adding a breakpoint and checking if there
14 * is a valid memory address helped me a fix most of the errors
15 * valgrind was used to fix memory leaks, as it showed the line number and the file
   name
16 * it was easy to locate, but took lot of research and efforts to fix memory leaks
17 *
18 */
19
20 #include <stdio.h>
21 #include "llist.h"
22 #include <stdlib.h>
23
24 void display_list(list *myList){
25     node *ptr = myList->head;
26     while(ptr){
27         printf( "\nnode :%s",ptr->string);
28         ptr=ptr->next;
29     }
30     printf("\ndisplay list() OK\n");
31 }
32
33 int main() {
34     list *myList;
35     //allocating memory for a list
36     myList= malloc(sizeof(list));
37
38     llInit(myList);
39
40     //Pushing DATA at the FRONT
41     printf("\n**Pushing DATA **");
42     llPushFront(myList,"A");
43     llPushFront(myList,"B");
44     //pushing at back
45     llPushBack(myList,"C");
46     llPushBack(myList,"D");
47     display_list(myList);

```

```

48     printf("\nSize: %d\n",llSize(myList));
49     printf("\n*****\n");
50     printf("End of Pushing Data");
51     printf("\n*****\n");
52
53
54     //Insert Before a specified node
55     printf("\n**Inserting Node Before HEAD Position**");
56     llInsertBefore(myList,myList->head,"F");
57     display_list(myList);
58     printf("\nSize: %d\n",llSize(myList));
59
60     printf("\n**Inserting Node Before tail**");
61     llInsertBefore(myList,myList->tail,"M");
62     display_list(myList);
63     printf("\nSize: %d\n",llSize(myList));
64
65
66     printf("\n*****\n");
67     printf("**End of Inserting Node Before a Specified**");
68     printf("\n*****\n");
69
70     //Insert After a Specified Node
71     printf("\n**Inserting Node at HEAD->NEXT->NEXT->PREV**");
72     llInsertAfter(myList,myList->head->next->next->prev,"E");
73     display_list(myList);
74     printf("\nSize: %d\n",llSize(myList));
75
76     printf("\n**Inserting Node after Tail**\n");
77     llInsertAfter(myList,myList->tail,"O");
78     display_list(myList);
79     printf("\nSize: %d\n",llSize(myList));
80     printf("\n*****\n");
81     printf("**End of Inserting Node after a Specified Node**");
82     printf("\n*****\n");
83
84     //remove node at specified position
85     printf("**\nRemoving node at the HEAD Position**\n");
86     llRemove(myList,myList->head);
87     display_list(myList);
88     printf("\nSize: %d\n",llSize(myList));
89
90     printf("\n*****\n");
91     printf("End of Removing node at Specified");
92     printf("\n*****\n");
93
94
95     //Delete Node at Front
96     printf("\nDeleteing Node at HEAD POSITION**\n");
97     llPopFront(myList);
98     display_list(myList);
99     // printf("\nSize: %d\n",llSize(myList));
100    printf("\n*****\n");
101    printf("End of Delete head");
102    printf("\n*****\n");
103
104    //POP Tail
105    printf("\n**Delete the Tail**\n");
106    llPopBack(myList);
107    display_list(myList);
108    // printf("\nSize: %d\n",llSize(myList));
109    printf("\n*****\n");
110    printf("**End of Deleting Tail**");
111    printf("\n*****\n");
112

```



```
113 //Clear the list
114 printf("\n**clear the list**\n");
115 llClear(myList);
116 printf("\nSize: %d\n",llSize(myList));
117 display list(myList);
118 printf("\n** End of test Driver **\n");
119
120 }
121
```

```
1 CC=gcc
2 CFLAGS=-I. -g
3 DEPS = llist.h
4 OBJ = llist.o driver.o
5 EXEC NAME = lldriver
6
7 %.o: %.c $(DEPS)
8     $(CC) -c -o $@ $< $(CFLAGS)
9
10 $(EXEC NAME): $(OBJ)
11     $(CC) -o $@ $^ $(CFLAGS)
12
13 clean:
14     rm $(EXEC NAME)
15     rm *.o
16
17
```