```
2
3
      #ifndef LLIST H
      #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
          pointer to NULL to indicate the end of the list, and likewise, the
10
11
12
13
          head node should assign its previous pointer to NULL for the same
          reason.
14
         Note that the pop operations do not return a reference to the
15
16
17
18
19
20
21
22
22
24
25
26
27
28
33
33
33
33
33
33
33
33
33
                            This would require storage for the node to be
          popped node.
          released by the user, which could lead to memory mishandling.
          The user will need to use head or tail pointers to interact with the
          node prior to popping it.
       * This list will only hold strings (char arrays) for simplicity, and * will have ownership of the strings. This means the string will
        * need to be copied into memory under control of the list.
       */
      /* Structures */
        * a node - cannot be anonymous because we need a node * inside, but
       * we can make the struct name and the typedef the same */
      typedef struct node {
            char *string;
           struct node* next; /* need struct here because inside typedef */
struct node* prev; /* need struct here because inside typedef */
      } node;
      /* The list itself - struct can be anonymous */
      typedef struct {
   node *head;
                            /* could have been struct node* as well */
            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()
               Initialize a list structure. An empty list will be characterized by head and tail pointer both being NULL.
48
49
50
51
53
54
55
56
57
59
60
                     Parameters: myList - a pointer to the structure to be init
                     Returns: void
      void llInit(list *myList);
               Reports the current size of the list. Will need to iterate
               the list to get this data size there is no size property, nor
               can there really be one given that users can access nodes.
                     Parameters: myList - the list Returns: int, size of list
61
62
      int llSize(list *myList);
63
64
      /* llPushFront()
               Add a new node with provided data and place node at front of list. The new node will replace the head node. This method should check to make sure the provided char * is not NULL. If it is NULL, this method should do nothing and make no changes to the list. If it is not NULL, it can be
65
66
67
       *
68
       *
69
70
               assumed that it is a valid null-terminated string.
```

```
71
                    Parameters:
                                    mvList - 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 any data from the list. If the data in the node is needed
 80
 81
83
84
85
86
88
99
91
93
               it should be accessed prior to the pop (list->head->string).
                    Parameters: myList - the list
Returns: int - 0 if no pop (list was empty) or non-zero
                                      if pop successful
       int llPopFront(list *myList);
       /* llPushBack()
               Add a new node with provided data and place node at
               end of list. This new node will be the new tail node.
               This method should check to make sure the provided char * is
               not NULL. If it is NULL, this method should do nothing and make no changes to the list. If it is not NULL, it can be
               assumed that it is a valid null-terminated string.
 94
 95
96
97
98
                                   myList - the list
                    Parameters:
                    toStore - the char array to store
Returns: int - 0 if no push (toStore was NULL) or non-zero
        *
                                      if push successful
 99
100
       int llPushBack(list *myList, char *toStore);
101
102
       /* llPopBack()
               Removes last item in list. Note, this does not return any data from the list. If the data in the node is needed
103
104
105
               it should be accessed prior to the pop (list->tail->string).
        *
                    Parameters: myList - the list
106
107
                    Returns: int - 0 if no pop (list was empty) or non-zero
                                      if pop successful
108
109
110
       int llPopBack(list *myList);
111
       /* llClear()
112
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
       /* Node methods
121
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
129
130
               Add a new node with provided data and place node after
               provided node reference.
               This method should check to make sure the provided char * is not NULL. If it is NULL, this method should do nothing and
131
132
        *
               not NULL.
        *
               make no changes to the list. If it is not NULL, it can be
               assumed that it is a valid null-terminated string.
133
134
        *
               If this method is called on the tail node, a change to the
135
        *
               list structure will need to be made.
136
                                   myList - theList
                    Parameters:
                                    insNode - the node after which item is added toStore - the char array to store
137
        *
138
        *
                    Returns: int - 0 if no insert (toStore was NULL or insNode
139
140
                                                         is NULL)
```

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

```
#include "l<mark>list.h"</mark>
 2
      #include <stdio.h>
 3
      #include <string.h>
 4
      #include <stdlib.h>
      //Maximum Length
 5
 6
      #define str len 100
 8
      void llInit(list *mylist) {
 9
10
           mylist-> head = NULL;
112345678991212134567899122222222223333333333349
          mylist-> tail = NULL;
      int llSize(list *mylist) {
           int count = 0;
           for (node *n=mylist->head; n != NULL; n = n->next)
                 count++;
           return count;
      }
       int llPushFront(list *myList, char *toStore){
            //allocating memory for a node
node *new node= malloc(sizeof(node));
            //allocating memory for string
new node->string=malloc(str_len);
             strcpy(new node->string,toStore);
            //setting the node as head node
            if(myList->head==NULL && myList->tail==NULL){
                 mvList->head=new node:
                 myList->tail=new node;
            }
                new node->next=myList->head;
               myList->head->prev=new node;
               new node->prev=NULL;
                //update my head with the new node
41
42
43
               myList->head= new node;
44
45
46
           free(new node);
           free(new node->string);
      }
47
48
      int llPopFront(list *myList){
           if(myList==NULL) {
49
50
51
53
54
55
55
55
55
60
               printf("List is Empty \n");
                return 0;
           //ptr to the head node
           node *ptr = myList->head;
          myList->head = myList->head->next;
           //delete
           free(ptr);
      }
      int llPushBack(list *myList, char *toStore){
    //allocating memory for a node
61
62
63
            node *new node= malloc(sizeof(node));
            //allocating memory for string
64
            new node->string=malloc(str len);
            if(new node == NULL) {
    printf("Failed to allocate");
65
66
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
76
           free(new node);
           free(new node->string);
 77
      }
 78
 79
      int llPopBack(list *myList){
 80
81
82
83
84
85
86
87
88
99
91
92
93
          if(myList==NULL) {
                printf("List is Empty \n");
                return 0;
           node *ptr= myList->tail;
           myList->tail=myList->tail->prev;
          myList->tail->next=NULL;
           //delete
           free(ptr);
      }
      void llClear(list *myList){
 94
95
96
97
98
           //get current head position
           node *ptr= myList->head;
           node *next;
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));
           //allocating memory for string
new node->string=malloc(str len);
110
111
           if(new node== NULL){
112
                printf("Failed to allocate");
113
114
                return 0;
115
116
            strncpy(new node->string,toStore, str len);
117
118
            if(myList->head==NULL && myList->tail==NULL){
119
                mvList->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
            else{
128
                new node->next=insNode->next;
129
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);
           if(new node== NULL) {
    printf("Failed to allocate");
144
145
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;
                new node->next=insNode;
155
156
                new node->prev=NULL;
157
                myList->head=new node;
158
159
           else{ new node->prev=insNode->prev;
                 insNode->prev->next=new node;
160
161
                 insNode->prev=new node;
162
                 new node->next=insNode;
163
164
            printf("\nllInsertBefore() OK");
           free(new node);
free(new node->string);
165
166
167
      int llRemove(list* myList, node *rmvNode){
168
169
170
           if(myList == NULL){
171
               return 0;
172
173
           //if node is at head
174
          else if(rmvNode->prev==NULL) {
175
               myList->head->next->prev = NULL;
176
               mvList->head = mvList->head->next;
177
178
           //if node is at tail
179
          else if(rmvNode->next==NULL){
180
                //ptr to tail node
181
               rmvNode = myList->tail;
182
               myList->tail->next=NULL;
183
               myList->tail=myList->tail->prev;
184
           élse {
185
               rmvNode->prev->next=rmvNode->next->prev;
186
187
188
           free(rmvNode->string);
189
           free(rmvNode):
190
      }
191
```

₹ ₽

₽ ₽

₽

₽

₽

₽

₽

₽

₽

Z

Z

₽

₽

```
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
       st Ans.) The structure uses pointers to access the nodes next, previois or the string.
      * 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
       st 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, intiazlize 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
       st 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?
       * Ans) the iterator would need a single pointer to a node and will point directly
 8
      to the node that you want it to point.
 9
      */
10
     /* Experiences with this lab:
11
      * working with segmentation faults and memory leaks was a challenging part
12
      * qdb was very useful for debugging, adding a breakpoint and checking if there
13
       * is a valid memory address helped me a fix most of the errors
14
15
       * valgrind was used to fix memory leaks, as it showed the line number and the file
       * it was easy to locate, but took lot of research and efforts to fix memory leaks
16
17
       */
18
19
20
     #include <stdio.h>
     #include "llist.h"
21
22
     #include <stdlib.h>
23
24
     void display list(list *myList){
25
          node *ptr = myList->head;
26
           while(ptr){
               printf( "\nnode :%s",ptr->string);
27
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
            printf("\n**Pushing DATA **");
llPushFront(myList, "A");
llPushFront(myList, "B");
41
42
43
44
              //pushing at back
            llPushBack(myList,"C");
llPushBack(myList,"D");
45
46
47
            display_list(myList);
```

```
printf("\nSize: %d\n",llSize(myList));
printf("\n*****************\n");
48
49
             printf("End of Pushiing Data");
printf("\n***************************);
50
51
52
53
54
             //Insert Before a specified node
             printf("\n**Inserting Node Before HEAD Position**");
55
56
             llInsertBefore(myList,myList->head,"F");
 57
             display list(myList);
printf("\nSize: %d\n",llSize(myList));
 58
59
60
              printf("\n**Inserting Node Before tail**");
61
             llInsertBefore(myList,myList->tail,"M");
             display list(myList);
printf("\nSize: %d\n",llSize(myList));
62
63
64
65
             printf("\n********************\n"):
66
             printf("**End of Inserting Node Before a Specified**");
67
             printf("\n****************\n"):
68
69
70
             //Insert After a Specified Node
71
             printf("\n**Inserting Node at HEAD->NEXT->PREV*");
 72
             llInsertAfter(myList,myList->head->next->next->prev,"E");
 73
             display list(myList);
              printf("\nSize: %d\n",llSize(myList));
 74
 75
 76
             printf("\n**Inserting Node after Tail**\n");
 77
             llInsertAfter(myList,myList->tail,"0");
 78
             display list(myList);
             printf("\nSize: %d\n",llSize(myList));
printf("\n*********************\n");
 79
80
             printf("**End of Inserting Node after a Specified Node**");
81
             printf("\n********************************
82
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);
             printf("\nSize: %d\n",llSize(myList));
88
89
             printf("\n**********************\n");
90
             printf("End of Removing node at Specified");
91
             printf("\n*****************************\n"):
92
93
94
95
            //Delete Node at Front
             printf("\nDeleteing Node at HEAD POSITION**\n");
96
97
             llPopFront(myList);
98
             display list(myList);
99
            // printf("\nSize: %d\n",llSize(myList));
             printf("\n*****************\n");
100
             printf("End of Delete head");
101
             printf("\n*********************\n");
102
103
             //POP Tail
104
             printf("\n**Delete the Tail**\n");
105
             llPopBack(myList);
106
107
             display list(myList);
108
            // printf("\nSize: %d\n",llSize(myList));
             printf("\n******************************
printf("*End of Deleting Tail*");
109
110
             printf("\n*********************\n"):
111
112
```

/home/shanthosh/LAB/driver.c
Page 3 of 3

Thu 24 Oct 2019 02:13:22 PM CDT