```c

#ifndef LLIST H
#define LLIST H
/* llist.h
 *
 * External (public) declarations for simple doubly-linked list in C.
 *
 * This list will know head and tail, and will be capable of forward
 * and reverse iteration.  The tail node should always assign its next
 * pointer to NULL to indicate the end of the list, and likewise, the
 * head node should assign its previous pointer to NULL for the same
 * reason.
 *
 * Note that the pop operations do not return a reference to the
 * popped node.  This would require storage for the node to be
 * 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;
} list;

/* List methods
 *
 * These methods are used to create and operate on a list as a whole.
 */

/* llInit()
 *     Initialize a list structure.  An empty list will
 *     be characterized by head and tail pointer both being NULL.
 *         Parameters:  myList - a pointer to the structure to be init
 *         Returns: void
 */
void llInit(list *myList);

/* llSize()
 *     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
 */
int llSize(list *myList);

/* 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
 *     assumed that it is a valid null-terminated string.
 *         Parameters:  myList - the list
 *                      toStore - the char array to store
 *         Returns: int - 0 if no push (toStore was NULL) or non-zero
 *                        if push successful
 */
int llPushFront(list *myList,char *toStore);

/* llPopFront()
 *     Removes first item in list.  Note, this does not return
 *     any data from the list.  If the data in the node is needed
 *     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.
 *         Parameters:  myList - the list
 *                      toStore - the char array to store
 *         Returns: int - 0 if no push (toStore was NULL) or non-zero
 *                        if push successful
 */
int llPushBack(list *myList, char *toStore);

/* llPopBack()
 *     Removes last item in list.  Note, this does not return
 *     any data from the list.  If the data in the node is needed
 *     it should be accessed prior to the pop (list->tail->string).
 *         Parameters:  myList - the list
 *         Returns: int - 0 if no pop (list was empty) or non-zero
 *                        if pop successful
 */
int llPopBack(list *myList);

/* llClear()
 *     Clears all nodes and releases all dynamic memory.  List
 *     structure should be NULLed and can be reused.
 *         Parameters:  myList - the list
 *         Returns: nothing
 */
void llClear(list *myList);


/* Node methods
 *
 * These methods allow iteration of nodes within the list.  A list
 * reference is still needed if head or tail needs to be modified.
 */

/* llInsertAfter()
 *     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
 *     make no changes to the list.  If it is not NULL, it can be
 *     assumed that it is a valid null-terminated string.
 *     If this method is called on the tail node, a change to the
 *     list structure will need to be made.
 *         Parameters:  myList - theList
 *                      insNode - the node after which item is added
 *                      toStore - the char array to store
 *         Returns: int - 0 if no insert (toStore was NULL or insNode
 *                                        is NULL)
```

```c
 *                        non-zero if insert successful
 */
int llInsertAfter(list* myList, node *insNode, char *toStore);

/* llInsertBefore()
 *     Add a new node with provided data and place node before
 *     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
 *     make no changes to the list.  If it is not NULL, it can be
 *     assumed that it is a valid null-terminated string.
 *     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
 *                      toStore - the char array to store
 *         Returns: int - 0 if no insert (toStore was NULL or insNode
 *                                        is NULL)
 *                        non-zero if insert successful
 */
int llInsertBefore(list* myList, node *insNode, char *toStore);

/* llRemove()
 *     Removes the node referenced.  Releases
 *     all associated dynamic memory.
 *     If this method is called the current head or tail node, changes
 *     to the list structure may need to be made.
 *         Parameters:  myList - the list
 *                      rmvNode - the node prior to the node to be
 *                                removed.
 *         Returns: nothing
 */
int llRemove(list* myList, node *rmvNode);

#endif
```

```
#include "llist.h"
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
//Maximum Length
#define str_len 100

void llInit(list *mylist) {

    mylist-> head = NULL;
    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){
         myList->head=new_node;
         myList->tail=new_node;
     }

     else{
        new_node->next=myList->head;
        myList->head->prev=new_node;
        new_node->prev=NULL;
        //update my head with the new node
        myList->head= new_node;
    }

    free(new_node);
    free(new_node->string);
}

int llPopFront(list *myList){
    if(myList==NULL) {
        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
     node *new_node= malloc(sizeof(node));
     //allocating memory for string
     new_node->string=malloc(str_len);
     if(new_node == NULL){
         printf("Failed to allocate");
         return 0;
     }
    strcpy(new_node->string,toStore);
    new_node->next=NULL;
    new node->prev = myList->tail;
    myList->tail->next=new node;
    //set tail to the new node
    myList->tail= new node;
    free(new node);
    free(new node->string);
}

int llPopBack(list *myList){

   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){

    //get current head position
    node *ptr= myList->head;
    node *next;

    while(ptr!=NULL){
    next=ptr->next;
    free(ptr);
    ptr=next;
    }
    myList->head=NULL;
}

int llInsertAfter(list* myList, node *insNode, char *toStore){
    //allocating memory for a node
    node *new node= malloc(sizeof(node));
    //allocating memory for string
    new node->string=malloc(str len);
    if(new node== NULL){
         printf("Failed to allocate");
         return 0;
     }
     strncpy(new node->string,toStore, str len);

     if(myList->head==NULL && myList->tail==NULL){
         myList->head=new node;
         myList->tail=new node;
     }
     else if(insNode->next==NULL){
        insNode->next=new node;
        new node->prev=insNode;
        new node->next=NULL;
        myList->tail=new node;
     }
     else{
         new node->next=insNode->next;
         new node->prev=insNode;
         insNode->next=new node;
         insNode->next->prev=new node;
     }
     printf("\nllInsertAfter() OK!\n");
     free(new node);
     free(new node->string);
}

int llInsertBefore(list* myList, node *insNode, char *toStore){
    //allocating memory for a node
    node *new node= malloc(sizeof(node));
    //allocating memory for string
    new node->string=malloc(str len);
    if(new node== NULL){
         printf("Failed to allocate");
         return 0;
     }
      strncpy(new node->string,toStore, str len);
      if(myList->head==NULL && myList->tail==NULL){
         myList->head=new node;
         myList->tail=new node;
     }
     else if(insNode->prev == NULL){
         insNode->prev=new node;
         new node->next=insNode;
         new node->prev=NULL;
         myList->head=new node;
     }
     else{ new node->prev=insNode->prev;
          insNode->prev->next=new node;
          insNode->prev=new node;
          new node->next=insNode;
     }
     printf("\nllInsertBefore() OK");
     free(new node);
     free(new node->string);
}
int llRemove(list* myList, node *rmvNode){

    if(myList == NULL){
        return 0;
    }
    //if node is at head
    else if(rmvNode->prev==NULL) {
        myList->head->next->prev = NULL;
        myList->head = myList->head->next;
    }
    //if node is at tail
    else if(rmvNode->next==NULL){
         //ptr to tail node
        rmvNode = myList->tail;
        myList->tail->next=NULL;
        myList->tail=myList->tail->prev;
    }
    else {
        rmvNode->prev->next=rmvNode->next->prev;
    }
    free(rmvNode->string);
    free(rmvNode);
}
```

```c
/* 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.
 * 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.
 * 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.
 * 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.
 * Ans) Yes, the program will crash and the result was Segmentation error
 * 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 to the node that you want it to point.
 *
 */
/* Experiences with this lab:
 * working with segmentation faults and memory leaks was a challenging part
 * gdb was very useful for debugging, adding a breakpoint and checking if there
 * is a valid memory address helped me a fix most of the errors
 * valgrind was used to fix memory leaks, as it showed the line number and the file name
 * it was easy to locate, but took lot of research and efforts to fix memory leaks
 *
 */

#include <stdio.h>
#include "llist.h"
#include <stdlib.h>

void display_list(list *myList){
    node *ptr = myList->head;
     while(ptr){
        printf( "\nnode :%s",ptr->string);
        ptr=ptr->next;
     }
    printf("\ndisplay_list() OK\n");
}

int main() {
      list *myList;
      //allocating memory for a list
      myList= malloc(sizeof(list));

      llInit(myList);

      //Pushing DATA at the FRONT
      printf("\n**Pushing DATA **");
      llPushFront(myList,"A");
      llPushFront(myList,"B");
       //pushing at back
      llPushBack(myList,"C");
      llPushBack(myList,"D");
      display_list(myList);
```

```
        printf("\nSize: %d\n",llSize(myList));
        printf("\n*************************\n");
        printf("End of Pushiing Data");
        printf("\n*************************\n");


        //Insert Before a specified node
        printf("\n**Inserting Node Before HEAD Position**");
        llInsertBefore(myList,myList->head,"F");
        display list(myList);
        printf("\nSize: %d\n",llSize(myList));

         printf("\n**Inserting Node Before tail**");
        llInsertBefore(myList,myList->tail,"M");
        display list(myList);
        printf("\nSize: %d\n",llSize(myList));


        printf("\n*************************\n");
        printf("**End of Inserting Node Before a Specified**");
        printf("\n*************************\n");

        //Insert After a Specified Node
        printf("\n**Inserting Node at HEAD->NEXT->NEXT->PREV*");
        llInsertAfter(myList,myList->head->next->next->prev,"E");
        display list(myList);
         printf("\nSize: %d\n",llSize(myList));

        printf("\n**Inserting Node after Tail**\n");
        llInsertAfter(myList,myList->tail,"O");
        display list(myList);
        printf("\nSize: %d\n",llSize(myList));
        printf("\n*************************\n");
        printf("**End of Inserting Node after a Specified Node**");
        printf("\n*************************\n");

      //remove node at specified position
        printf("**\nRemoving node at the HEAD Position**\n");
        llRemove(myList,myList->head);
        display list(myList);
        printf("\nSize: %d\n",llSize(myList));

        printf("\n*************************\n");
        printf("End of Removing node at Specified");
        printf("\n*************************\n");


       //Delete Node at Front
        printf("\nDeleteing  Node at HEAD POSITION**\n");
        llPopFront(myList);
        display list(myList);
       // printf("\nSize: %d\n",llSize(myList));
        printf("\n*************************\n");
        printf("End of Delete head");
        printf("\n*************************\n");

        //POP Tail
        printf("\n**Delete the Tail**\n");
        llPopBack(myList);
        display list(myList);
       // printf("\nSize: %d\n",llSize(myList));
        printf("\n*************************\n");
        printf("*End of Deleting Tail*");
        printf("\n*************************\n");

```

```c
       //Clear the list
       printf("\n**clear the list**\n");
       llClear(myList);
       printf("\nSize: %d\n",llSize(myList));
       display list(myList);
       printf("\n** End of test Driver **\n");

  }

```

```
CC=gcc
CFLAGS=-I. -g
DEPS = llist.h
OBJ = llist.o driver.o
EXEC NAME = lldriver

%.o: %.c $(DEPS)
    $(CC) -c -o $@ $< $(CFLAGS)

$(EXEC NAME): $(OBJ)
    $(CC) -o $@ $^ $(CFLAGS)

clean:
    rm $(EXEC NAME)
    rm *.o


```