```c
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include <stdio.h>
#include <limits.h>



#ifndef __LINKEDLIST_H
#define __LINKEDLIST_H

/**
 * A doubly linked-list node.
 */
typedef struct ll_node_s {
    int value;
    struct ll_node_s *prev;
    struct ll_node_s *next;
} ll_node;


/**
 * Returns a pointer to the first node of a list, given a pointer to any node
 * in the list. If the provided pointer is `NULL`, instead returns `NULL`.
 */
ll_node *ll_head(ll_node *list);

/**
 * Returns a pointer to the last node of a list, given a pointer to any node
 * in the list. If the provided pointer is `NULL`, instead returns `NULL`.
 */
ll_node *ll_tail(ll_node *list);

/**
 * Returns the number of nodes in the list, which is the same for all nodes
 * in the list and 0 for `NULL`.
 */
unsigned long ll_length(ll_node *list);

/**
 * Given a pointer to a node in a list, returns a pointer to the first node
 * at or after that node which has the given `value`. If given `NULL`, or
 * if no such node exists, returns `NULL`.
 */
ll_node *ll_find(ll_node *list, int value);



/**
 * Given a pointer to a node in a list, remove that node from the list,
 * `free`ing its memory in the process. Returns a pointer to the node that
now
 * occupies the same position in the list that the removed node used to
occupy
```

```
 * (which may be `NULL` if the removed node was the last node in the list).
 *
 * If given `NULL`, this function does nothing and returns `NULL`.
 */
ll_node *ll_remove(ll_node *list);

/**
 * Extend a list by one by adding `value` next to `list`. If `before` is 0,
 * inserts `value` immediately following the node pointed to by `list`;
 * otherwise inserts `value` immediately before that node. If `list` is NULL,
 * the newly inserted node is the entire list. In all cases, the new node is
 * allocated using `malloc` and returned by the function.
 */
ll_node *ll_insert(int value, ll_node *list, int before);

/**
 * Displays the contents of the list separated by commas and surrounded by
 * brackets, with the pointed-to node highlighted with asterisks.
 */
void ll_show(ll_node *list);


#endif /* ifdef __LINKEDLIST_H */

ll_node *ll_head(ll_node *list){
    if(list == NULL)
        return NULL;
    while(list->prev != NULL)
        list = list->prev;
    return list; //returns the head node.
}

ll_node *ll_tail(ll_node *list){
    if(list == NULL)
        return NULL;
    while(list->next != NULL)
        list = list->next;
    return list; //returns the tail node.

}
void ll_show(ll_node *list) {
    ll_node *ptr = ll_head(list);
    printf("%c", '[');
    while(ptr) {
        if (ptr->prev) printf("%c", ',');
        // if (ptr == list) putchar('*');
        printf(" %d", ptr->value);
        //  if (ptr == list) putchar('*');
        ptr = ptr->next;
    }
    puts(" ]");
}



ll_node *ll_remove(ll_node *list)
{
```

```
    ll_node *curN;

    if(list == NULL)
        return NULL;

    //removes the node

    //Singleton

    if(list->next == NULL && list->prev == NULL)
    {
        curN = NULL;

    }

    else  if(list->next == NULL){ //tail case
        curN = list->prev;
        list->prev->next = NULL;
        curN = NULL;
    }

    else if(list->prev == NULL)//head case
    {
        curN = list->next;
        list->next->prev = list->prev;
    }

    else //otherwise:
    {
        curN = list->next;
        list->prev->next = list->next;
        list->next->prev = list->prev;
    }

    free(list);
    //need anything here?
    return curN;


}
ll_node *ll_pop(ll_node *list)
{
    //puts("pop method");
    ll_node *curN;
    //puts("list before");
    // ll_show(list);

    if(list == NULL)
        return NULL;
    if(list->next == NULL && list->prev == NULL)
    {
        //puts("Singly Node");
        curN = NULL;

    }
    else{
        while(list->next != NULL) //Move to the head of the list.
            list = list->next;

        curN = list->prev;
```

```
        list->prev->next = NULL; //Set the next node's prev to NULL


    }
    // puts("list after");
    // ll_show(list);
    free(list);
    // puts("list after free");
    //  ll_show(list);
    return curN;
}

int ll_peek(ll_node *list)
{
    //puts("Peeking.");
    //ll_show(list);
    // printf("%d\n", list->value);
    return list->value;
}

ll_node *ll_insert(int value, ll_node *list, int before)
{
    //the new node
    ll_node* new  = (ll_node*)malloc(sizeof(ll_node));
    new->value = value;
    new->next = NULL;
    new->prev = NULL;

    //    errors:     insert before/after head (crash)
    //    insert after null - crash
    if(list == NULL)
        return new;



    //if before is zero, insert the value AFTER list node
    else if(before < 1)
    {
        /*    //Insert after null
         if(list->next == NULL  && list->prev == NULL)
         {
         list->next = new;
         }

         */    //Tail case
        //need if statements head or tail for both insert cases
        if (list->next == NULL) //if the element is the tail
        {
            new->prev = list;
            list->next = new;
            new->next = NULL;

        }
        //Head case
        else if(list->prev == NULL)//if the element is the head
        {
            //Error: wrong prev?

            new->prev = list;
```

```
            new->next = list->next;//list pointer exists (next)
            list->next->prev = new;
            list->next = new;
        }


        //Otherwise...
        else
        {
            new->prev = list;
            new->next = list->next;//list pointer exists (next)
            list->next->prev = new; //list->next pointer doesn't exist
            list->next = new;
        }
    }

    //need if statements head or tail for both insert cases
    else if(list->next == NULL)
    {
        new->next = list;
        new->prev = list->prev;

        list->prev = new;
        list->prev->prev->next = new;
    }
    else if(list->prev == NULL)
    {
        new->next = list;

        list->prev = new;
    }


    else
    {



        //insert value BEFORE node
        new->next = list;
        new->prev = list->prev;

        list->prev->next = new;
        list->prev = new;
    }

    return new ;
}




ll_node *ll_push(int value, ll_node *list){

    return ll_insert(value, list, 0);
}
```

```c
int main(int argc, char const *argv[]){

    char buff[4096];
    size_t size = 4096;
    int i;
    ll_node *stack;
    ll_node *temp;
    int count = 0;

    while(1) //EOF check
    {
        if(read(0, buff, size) == 0)
        {
            ll_show(stack);
            return 0;
        }
        //  puts("Wow");
        //  printf("Buffer is ");
        //  puts(buff);

        char *string = strdup(buff);
        char *s;
        int num;
        char *t;



        int result;
        int op1;
        int op2;


        //Sort through the buffer.
        while(( s = strsep(&string," ")) != NULL){
            // ll_show(stack);


            //puts("Stack is ");
            //  ll_show(stack);

            //  printf("s is (%s) \n", s);
            if(strcmp(s," ") != 0 && strcmp(s,"") != 0 )
            {

                if(((num = strtol(s,&t,10)) != 0) || strcmp(s,"0") == 0 ||
strcmp(s,"0\n")  == 0 || strcmp(s,"0 ") == 0 || strcmp(s,"0 \n") == 0 ||
strstr(s,"0") != NULL)
                {
                    //  printf("Adding %d to stack. \n", num);
                    stack = ll_push(num,stack);
                    count += 1;


                }

                //Adding
                else if(strcmp(s,"+") == 0 || strcmp(s,"+\n")  == 0 ||
strcmp(s,"+ ") == 0 || strcmp(s,"+ \n") == 0)
                {
                    if(count < 2)
```

```
                    {
                        //                ll_show(stack);
                        return 0;
                    }

                    //                puts("Adding");
                    op2 = ll_peek(stack);
                    stack = ll_pop(stack);
                    //                ll_show(stack);
                    //                printf("op2 = %d \n", op2);

                    op1 = ll_peek(stack);

                    stack = ll_pop(stack);
                    //                ll_show(stack);
                    //                printf("op1 = %d \n", op1);

                    result = op1 + op2;

                    //   printf("result = %d \n", result);

                    stack =  ll_push(result,stack);
                    //                puts("push result onto stack");
                     //              ll_show(stack);
                    count-= 1;
                }
                //Subtracting
                else if(strcmp(s,"-") == 0 || strcmp(s,"-\n")  == 0 ||
strcmp(s,"- ") == 0 || strcmp(s,"- \n") == 0)
                {
                    if(count < 2)
                    {
                        //   ll_show(stack);
                        return 0;
                    }
                    //                puts("Subtracting");
                    op2 = ll_peek(stack);
                    stack = ll_pop(stack);
                    //                ll_show(stack);
                    //              printf("op2 = %d \n", op2);

                    op1 = ll_peek(stack);

                    stack = ll_pop(stack);
                    //                ll_show(stack);
                    //                printf("op1 = %d \n", op1);

                    result = op1 - op2;

                    //                printf("result = %d \n", result);

                    stack = ll_push(result,stack);
                    //              puts("push result onto stack");
                    //              ll_show(stack);
                    count-= 1;
                }
                //Multiplying
                else if(strcmp(s,"*") == 0 || strcmp(s,"*\n")  == 0 ||
strcmp(s,"* ") == 0 || strcmp(s,"* \n") == 0)
                {
```

```
                    if(count < 2)
                    {
                        //        ll_show(stack);
                        return 0;
                    }
                    //               puts("Mult.");
                    op2 = ll_peek(stack);
                    stack = ll_pop(stack);
                    //             ll_show(stack);
                    //           printf("op2 = %d \n", op2);

                    op1 = ll_peek(stack);

                    stack = ll_pop(stack);
                    //       ll_show(stack);
                    //         printf("op1 = %d \n", op1);

                    result = op1 * op2;

                    //     printf("result = %d \n", result);

                    stack = ll_push(result,stack);
                    // puts("push result onto stack");
                    //  ll_show(stack);
                    count-= 1;
                }
                //Dividing
                else if(strcmp(s,"/") == 0 || strcmp(s,"/\n")  == 0 ||
strcmp(s,"/ ") == 0 || strcmp(s,"/ \n") == 0)
                {
                    if(count < 2)
                    {
                        //        ll_show(stack);
                        return 0;
                    }
                    //                puts("Dividing");
                    op2 = ll_peek(stack);
                    stack = ll_pop(stack);
                    //              ll_show(stack);
                    //            printf("op2 = %d \n", op2);

                    op1 = ll_peek(stack);

                    stack = ll_pop(stack);
                    //          ll_show(stack);
                    //        printf("op1 = %d \n", op1);

                    result = op1 / op2;

                    //       printf("result = %d \n", result);

                    stack = ll_push(result,stack);
                    //     puts("push result onto stack");
                     //  ll_show(stack);
                    count-= 1;
                }
                else{
                     ll_show(stack);

                    return 0;
```

```
                }

            }//ends big if statement
            memset(buff,0,strlen(buff));
        }
             //   ll_show(stack);
    }


    return 0;
}
```