```c
//ICSI 333. System Fundamentals
//Peter Buonaiuto: 001497626
//TA Omkar Kulkarni
//Spring 2022

// This program will take user cmds to manipulate a linked list.

// The project specifies for no exception handling to be done or no extraneous logic to be
// implemented. It handles the 6 commands from spec and no more, as required.

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

//Global Constants
const int BEFORE = 0;           //Key that denotes 'before' insertion
const int AFTER = 1;            //Key that denotes 'after' insertion
const int ARG_LEN = 256;        //Maximum length of an argument

//Define structure for node which will store an index and a text
struct node
{
    char text[ARG_LEN];
    int index;
    struct node *next;
};

//Linked List Pointers:
struct node *head = NULL;
struct node *tail = NULL;

//Declare functions
void delete(int);                                   //Remove a node
void replace(int, char[]);                          //Replace a node's value
void print_list(struct node *);                     //Print all nodes
```

```
int getLength(struct node *);                                  //Return the length of the list
void updateIndeces(struct node *, int);                        //Increase or decrease indexes
void insert(struct node **, struct node **, int, char[], int);   //Insert a new node at the given
index
void insertAfter(struct node *, int);                          //Insert a node after the spec
index
void insertBefore(struct node *, int);                         //Insert a node before the spec
index
int doesExist(struct node *, char[]);                          //Check whether the text is
present
struct node * findNode(struct node*, int);                     //Return the node at the given
index

//MAIN LOGIC AND LOOP
int main() {

//Initialize variables
char response[ARG_LEN + 10];   //The text supplied by the user
char * cmd;                    //The specific three character command
char args[5][ARG_LEN];         //Argument string array
int argi;                      //Argument index
int target_index;              //The index specified for operation
char target_text[ARG_LEN];     //The text specified for operation

//Command constants
const char CMD_END[] = "end";
const char CMD_INSERT_A[] = "ina";
const char CMD_INSERT_B[] = "inb";
const char CMD_DELETE[] = "del";
const char CMD_REPLACE[] = "rep";
const char CMD_PRINT[] = "prn";

//Main prompt loop to get commands and execute delegates.
    do
    {
```

```
    printf("Enter a command: ");
    fflush(stdin);
    fflush(stdout);
    scanf("%[^\n]s", response);

    //We stored the response, now extract the arguments
    char * token = strtok(response, " ");
    argi = 0;

    //Store each argument of the given command
while(token != NULL)
{
    //Put the valid argument into our argument array
    strcpy(args[argi++], token);
  token = strtok(NULL, " ");\
    }

    cmd = args[0]; //Store the first argument as our command

    //Determine which command, if any, was called
    if ((strcmp(cmd, CMD_INSERT_A) == 0) || (strcmp(cmd, CMD_INSERT_B) == 0))
    {// INSERT

        //Gather Arguments
        target_index = atoi(args[1]);     //Index to insert after
        strcpy(target_text, args[2]);     //New text to add

        //Make sure we don't already contain this text.
        if (doesExist(head, target_text))
        {
            printf("Such text exists already: %s\n",target_text);
        }
        else //Add the new node
        {
            //Determine location of insertion; before or after based on command.
```

```
                int location = (strcmp(cmd, CMD_INSERT_A) == 0) ? AFTER :
                BEFORE;

                //Insert the node at the correct location
                insert(&head, &tail, target_index, target_text, location);
            }
        }
        else if (strcmp(cmd, CMD_DELETE) == 0)
        {// DELETE

            //Read argument for index to delete, then call delete
            target_index = atoi(args[1]);

            //Ensure this index exists. Alternatively, could check if findNode is null.
            if (target_index <= getLength(head) && target_index > 0)
            {
                delete(target_index);
            }
            else
            {
                printf("No such index\n");
                fflush(stdout);
            }
        }
        else if (strcmp(cmd, CMD_REPLACE) == 0)
        {// REPLACE

            //Setup arguments
            target_index = atoi(args[1]);
            strcpy(target_text, args[2]);

            //Make sure the index is valid.
            if (target_index <= getLength(head) && target_index > 0)
            {
                if(doesExist(head, target_text))
                {
                    printf("Such text exists already.\n");
                    fflush(stdout);
                }
                else //We can replace the text
                {
                    replace(target_index, target_text);
                }
            }
            else
            {
                printf("No such index\n");
                fflush(stdout);
            }
        }
        else if (strcmp(cmd, CMD_PRINT) == 0)
        {// PRINT
            print_list(head);
        }
        else if (strcmp(cmd, CMD_END) == 0)
        {// STOP PROGRAM
            printf("Goodbye!\n");
        }
        else
        {// UNKNOWN COMMAND
            //printf("Unknown command\n"); //Description does not allow Unknown Cmd
            Msg.
            //fflush(stdout);
        }
    } while (strcmp(cmd, CMD_END) != 0); //Prompt input until end is desired.

}

//REPLACE
void replace(int index, char text[ARG_LEN])
```

```
{
    //At this point, the index is valid and the target string is unique.
    strcpy(findNode(head, index)->text, text);
    printf("Replaced.\n");
    fflush(stdout);
}

//DELETE
void delete(int index)
{
    //If the target is in the center of the list, then simply redirect the pointer.
    if (index > 1 && index < getLength(head))
    {
        //Set the node to the left to point to the node to the right, skipping the deleted
        node.
        findNode(head, index-1)->next = findNode(head, index+1);
        updateIndeces(findNode(head, index), -1);
    }

    //Deleting the head
    else if (index == 1)
    {
        //If head is the only element, set it to null
        if (head->next == NULL)
        {
            head = tail = NULL;
        }
        // There's more than just one element; shift head and indeces
        else
        {
            head = head->next;
            updateIndeces(head, -1);
        }
    }
    //Deleting the tail
    else
    {
        //Point the second to last to null
        findNode(head, index-1)-> next = NULL;

        //The last is now the tail
        tail = findNode(head, index-1);
    }

    printf("Deleted.\n");
    fflush(stdout);
}
//PRINT THE LIST
void print_list(struct node *h)
{
    if (h == NULL)
    {
        printf("The list is empty.\n");
    }
    else
    {
        printf("Values in the list are:\n");
        fflush(stdout);
        while (h != NULL)
        {
            printf("%d: %s\n", h->index, h->text);
            fflush(stdout);
            h = h->next;
        }
    }
}

//LENGTH OF LIST
int getLength(struct node* h)
```

```
{
    int count = 0;
    while (h != NULL)
    {//Increment the count until the next node is null.
        h = h->next;
        count++;
    }
    return count;
}

//Insert After
void insert(struct node **h, struct node **t, int index, char text[256], int location)
{
    struct node *new_node; //The new node to creat

    //Attempt memory access for New Node
    if ((new_node = (struct node *)malloc(sizeof(struct node))) == NULL) {
        printf("Node allocation failed. \n");
        exit(1); /* Stop program */
    }

    //Assign the new node's desired text
    strcpy(new_node->text, text);

    //This is the first element, so it will be index 1.
    if (*h == NULL) {
        new_node->index = 1;
        *h = *t = new_node;
        new_node->next = NULL; //This is the only element, so it has no next node.
        printf("Ok.\n");
    }

    //The given index is out of bounds, put this at the end or the beginning
    else if (index < 1 || index > getLength(head))
    {
        if (location == BEFORE)
        {//Out of bounds and before - Put at beggining

            //Set the new node's tail to the head, putting it in front. Then update head
            new_node->next = head;
            *h = new_node;

            //Set the new node's index to one. Increase all other indexes by 1.
            new_node->index = 1;
            updateIndeces(head->next, 1);

            printf("Text inserted at the beginning.\n");
        }
        else
        {//Out of bounds and after - Put at end
            new_node->next = NULL;
            *t = (*t)->next = new_node;
            new_node->index = getLength(head);

            printf("Text inserted at the end.\n");
        }
    }

    //Trying to put something before the head
    else if (index == 1 && location == BEFORE)
    {
        //Set the new node's tail to the head, putting it in front. Then update head
        new_node->next = head;
        *h = new_node;

        //Set the new node's index to one. Increase all other indexes by 1.
        new_node->index = 1;
        updateIndeces(head->next, 1);
        printf("Ok.\n");
    }
```

```
     // This element is in the middle of the list. Call delegate
    else if (index < getLength(head))
    {
        if (location == BEFORE)
        {
            insertBefore(new_node, index);
        }
        else
        {
            insertAfter(new_node, index);
        }
        printf("Ok.\n");
    }

    else //We're targetting the last index:
    {
        if (location == AFTER)
        {//Insert this node after the last node (the end)

            new_node->next = NULL;
            *t = (*t)->next = new_node;
            new_node->index = getLength(head);
        }
        else
        {//Insert this node before the last node
            insertBefore(new_node, index);
        }
        printf("Ok.\n");
    }

    fflush(stdout);
}

//Insert this node after the noted index by splitting and shifting.
```

```c
void insertAfter(struct node * new_node, int index)
{
	//Store all nodes after, to split the list in half and insert in between.
	struct node *trail = findNode(head, index+1);

	//Must increse all indexes of the trailing nodes by one
	updateIndeces(trail, 1);

	//Set the next reference of the addressed node to the new node
	findNode(head, index)->next = new_node;

	//Attatch trailing nodes to the new node
	new_node->next = trail;

	//Set index for the new node - one more than the one it is after
	new_node->index = index + 1;
}

//Insert this node before the desired index.
void insertBefore(struct node * new_node, int index)
{
	//New node's NEXT will point to index (because we add before)
	new_node->next = findNode(head, index);

	//Node at index-1 will be to the left (it's NEXT points to new node)
	findNode(head, index-1)->next = new_node;

	//Increase all trailing node indexes by 1 (new node pushed others)
	updateIndeces(findNode(head, index), 1);

	//Set the new node index to index (taking its spot)
	new_node->index = index;
}

//Returns a pointer to the node at index
```

```
struct node * findNode(struct node* h, int index)
{
    //Return null if index is out of bounds
    if (index < 1 || index > getLength(head))
    {
        return NULL;
    }
    //Body will go to the next node. Execute body index-1 times, as we start at index 1
    (head)
    while (index > 1)
    {
        if (h != NULL) //Proceed if not null
            h = h->next;
        --index;
    }
    //Return the node at index provided after looking after the head index times
    return h; //NULL if given out of bound index to find
}

//Increase all indexes of this tail for shifting
void updateIndeces(struct node *h, int change)
{
    while (h != NULL)
    {  //
        h->index += change;
        h = h->next;
    }
}
//Check if the supplied text is already in the list before atnew_nodeting insertion
int doesExist(struct node *h, char str[256])
{
    while (h != NULL)
    {
        if (strcmp(str, h->text) == 0)
        { //This node already contains the desired string. Return true
```

```
                        return 1;
                }
                h = h->next;
        }
        //No nodes were found containing the desired string. Exit with false.
        return 0;
}
```