```c
======================================================================
            /local/submit/submit/comp10002/ass2/hestertzehun/src/myass2.c
======================================================================

/* Solution to comp10002 Assignment 2, 2019 semester 2.

   Authorship Declaration:

   (1) I certify that the program contained in this submission is completely
   my own individual work, except where explicitly noted by comments that
   provide details otherwise.  I understand that work that has been developed
   by another student, or by me in collaboration with other students,
   or by non-students as a result of request, solicitation, or payment,
   may not be submitted for assessment in this subject.  I understand that
   submitting for assessment work developed by or in collaboration with
   other students or non-students constitutes Academic Misconduct, and
   may be penalized by mark deductions, or by other penalties determined
   via the University of Melbourne Academic Honesty Policy, as described
   at https://academicintegrity.unimelb.edu.au.

   (2) I also certify that I have not provided a copy of this work in either
   softcopy or hardcopy or any other form to any other student, and nor will
   I do so until after the marks are released. I understand that providing
   my work to other students, regardless of my intention or any undertakings
   made to me by that other student, is also Academic Misconduct.

   (3) I further understand that providing a copy of the assignment
   specification to any form of code authoring or assignment tutoring
   service, or drawing the attention of others to such services and code
   that may have been made available via such a service, may be regarded
   as Student General Misconduct (interfering with the teaching activities
   of the University and/or inciting others to commit Academic Misconduct).
   I understand that an allegation of Student General Misconduct may arise
   regardless of whether or not I personally make use of such solutions
   or sought benefit from such actions.

   Signed by: HESTER LIM TZE HUNG 1044793
   Dated:     15 OCTOBER 2019

*/
/* Import Libraries
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <assert.h>

/* Linked List functions
*/
typedef int data_t;

typedef struct node node_t;

struct node{
    data_t data;
    node_t *next;
};
typedef struct{
    node_t *head;
    node_t *foot;
} list_t;

list_t *make_empty_list(void);
int is_empty_list(list_t *list);
void free_list(list_t *list);
list_t *insert_at_head(list_t *list, data_t value);
list_t *insert_at_foot(list_t *list, data_t value);
data_t get_head(list_t *list);
list_t *get_tail(list_t *list);

/* function prototypes
*/
void print_prompt_stage_0(void);
void get_dimension();
void get_block(char **matrix);
void get_route(char **matrix, list_t *list);
void print_output(list_t *list);
void exit_if_null(void *ptr, char *msg);
void status_func(int status);
void initialize_matrix(char **matrix);
void print_output_1(char **matrix);
void print_header_1();
int check_initial_cell(int row, int column);
void check_goal_cell(int row, int column);
void print_prompt_stage_1(void);
void seperator_line(void);
void traverse_grid(char **matrix, list_t *list, list_t *repair_list);
void route_fragment(char **matrix, list_t *list, list_t *repair_list);
void repaired_matrix(char **matrix);
void print_repaired_route(list_t *repair_list);
void print_equal_seperator_line(void);
void process_stage_0(char **matrix, list_t *list);
int process_stage_1(char **matrix, list_t *list,list_t *repair_list);

struct {
     int dimension_row, dimension_column;
     int initial_row, initial_column;
     int goal_row, goal_column;
     int count;
     int num_block, num_route;
     int flag;
     int status;
     int final_row, final_column;
} coordinate_t;

struct{
    int counter_value;
    int total_count;
    int count;
} repair_t;

#define INITIAL 100 //Inital Route size - Probabaly need to be more felxible
#define BLOCK '#'
#define ROUTE '*'
#define START 'I'
#define GOAL 'G'
#define NEWLINE "\n"
#define EMPTY_CHAR ' '
#define SPECIAL_CHAR '$'

struct route {
    int row; //get each row's route
    int column; //get each column's route
};

struct route block_list[INITIAL]; //store all the blocks
struct route route_list[INITIAL];
struct route broken_route_list[INITIAL]; // store all the broken route segment
struct route repair_route_list[INITIAL]; //store all the repair route list

// Construct a queue of pairs
struct queue {
    struct route location;
    int counter;
};

struct queue queue_pair[INITIAL];

int
main(int argc, char *argv[]) {
    coordinate_t.count = 0;coordinate_t.num_block = 0;
    coordinate_t.num_route = 0;coordinate_t.flag = 1;coordinate_t.status = 0;
    char **matrix = NULL; // An array of arrays of datatype
    int i;

    // Create an empty linked list
    list_t *list;
    list = make_empty_list();

    get_dimension();
    // Create a fresh segment of memory
    matrix = (char**)malloc(coordinate_t.dimension_row * sizeof(*matrix));
    for(i = 0; i < coordinate_t.dimension_row; i++){
        matrix[i] = (char*)malloc(sizeof(*matrix)*coordinate_t.dimension_column);

    }
    exit_if_null(matrix, "initial allocation");
    assert(matrix != NULL);

    process_stage_0(matrix,list);

    if(coordinate_t.status == 4){
        get_block(matrix);
        list_t *repair_list;
        repair_list = make_empty_list();
        process_stage_1(matrix, list, repair_list);
    }
    print_equal_seperator_line();

    return 0;
}

/* Do Stage 1 - Drawing and Replanning
*/
int
process_stage_1(char **matrix, list_t *list,list_t *repair_list){
    seperator_line();
    traverse_grid(matrix,list, repair_list);

    // Check if there is a route that could be repaired
    if(repair_t.count > coordinate_t.dimension_row *
        coordinate_t.dimension_column){
        initialize_matrix(matrix);
        repaired_matrix(matrix);
        print_header_1();
        print_output_1(matrix);
        printf("The route cannot be repaired\n");
        return 0;
    }

    route_fragment(matrix,list, repair_list);
    //Reinitialize matrix
    initialize_matrix(matrix);
    repaired_matrix(matrix);
    print_header_1();
    print_output_1(matrix);
    seperator_line();
    print_repaired_route(repair_list);
    // The route has been repaired so the route must be valid
    status_func(5);
    return 0;
}

/* Do Stage 0 - Reading and Analyzing Input Data
*/
void
process_stage_0(char **matrix, list_t *list){
    print_prompt_stage_0();
    initialize_matrix(matrix);
    get_block(matrix);
    get_route(matrix,list);
    print_output(list);
    print_prompt_stage_1();
    print_header_1();
    print_output_1(matrix);
}

/* Print out the matrix after completing Stage 1
*/
void
print_repaired_route(list_t *repair_list){
    int row,column,i;
    int count = 0;
    while (!is_empty_list(repair_list)){
        i = get_head(repair_list);
        count ++;
        row = repair_route_list[i].row;
        column = repair_route_list[i].column;
        printf("[%d,%d]",row, column);
        if(i == repair_t.counter_value - 1){
            printf(".");
        }else{
            printf("->");
        }
        if(count ==5){
            printf(NEWLINE);
            count = 0;
        }
        repair_list = get_tail(repair_list);
    }
    printf(NEWLINE);
}

/* Redraw the Matrix
*/
void
repaired_matrix(char **matrix){
    int row, column;
    int i;
    // Store all the blocks into the matrix
    for(i = 0; i < coordinate_t.num_block; i++){
        row = block_list[i].row;
        column = block_list[i].column;
        matrix[row][column] = BLOCK;
    }

    //Store 'I' ,'G' and '*' into the matrix
    for(i = 0; i < repair_t.counter_value ; i++){
        row = repair_route_list[i].row;
        column = repair_route_list[i].column;
        if(row == coordinate_t.initial_row &&
            column == coordinate_t.initial_column){
            matrix[row][column] = START;
        }else if(row == coordinate_t.final_row &&
            column == coordinate_t.final_column){
            matrix[row][column] = GOAL;
        }else{
            matrix[row][column] = ROUTE;
        }
    }
}

void
traverse_grid(char **matrix, list_t *list, list_t *repair_list){
    /* Now go throught the array of struct and check it with the matrix
    if the matrix has '#' in it then we need to step forward once and backwards
    once to get where is the location of the repair segment
    */
    int i,j;
    int row, column, flag;
    flag = 0;
    int counter_value = 0;
    int broken_route, broken_row, broken_column;
    broken_route = 0; broken_row = 0; broken_column = 0;
    i = 0;

    // Get the first instance where the route is blocked
    while(i < coordinate_t.num_route){
        row = route_list[i].row;
        column = route_list[i].column;
        //printf("[%d,%d], %d\n",row,column, i);
        if(matrix[row][column] == BLOCK){
            broken_route = i - 1;
            broken_row = route_list[broken_route].row;
            broken_column =  route_list[broken_route].column;
            i++;
            break;
        }else{
            repair_route_list[counter_value].row = row;
            repair_route_list[counter_value].column = column;
            insert_at_foot(repair_list, counter_value);
            counter_value++;
        }
        i++;
    }
    // Store the counter value into repair_t
    repair_t.counter_value = counter_value;

    list = make_empty_list();

    int list_num = 0;
    queue_pair[list_num].location.row = broken_row;
    queue_pair[list_num].location.column = broken_column;
    queue_pair[list_num].counter = 0;
    /*printf("([%d,%d], %d)\n", queue_pair[list_num].location.row,
    queue_pair[list_num].location.column, queue_pair[list_num].counter);*/
    list = insert_at_foot(list,list_num);

    /* Starting from the first pair, we then traverse the queue.
    When traversing a pair in the queue, for each cell in the grid
    that is adjacent to the cell in the tranversed pair and is not blocked,
    we add a fresh pair to the end of the queue composed of the adjacent cell
    and a counter value that is greater than the counter value in the currently
    traversed pair by one.
    */

    int prev_counter = 0;
    int route_num;
    int prev_list_num;
    int total_count =0;
    int found = 0;
    int count = 0;
    while(found == 0 && count <= coordinate_t.dimension_row
        * coordinate_t.dimension_column){
        for(i = 0; i < coordinate_t.dimension_row; i++){
            for(j = 0;j < coordinate_t.dimension_column; j++){
                //found the latest counter value
                if(i == queue_pair[list_num].location.row &&
                    j == queue_pair[list_num].location.column){
                    prev_list_num = list_num;
                    prev_counter = queue_pair[list_num].counter;
                    /*Check above, below, left and right and
                    insert into coordinate where appropriate.
                    Inser a Special Character ’$’ if we have traversed before
                    in the matrix
                    */
                    if((i+1) >= 0 && (i+1) < coordinate_t.dimension_row &&
                        matrix[i+1][j] == EMPTY_CHAR ){
                        total_count++; //increment the linked list array
                        queue_pair[total_count].location.row = i + 1;
                        queue_pair[total_count].location.column = j;
                        queue_pair[total_count].counter = prev_counter + 1;
                        list = insert_at_foot(list,total_count);
                        matrix[i+1][j] = SPECIAL_CHAR;
                    }
                    if((i-1) >= 0 && (i-1) < coordinate_t.dimension_row &&
                        matrix[i-1][j] == EMPTY_CHAR){
                        total_count++;
                        queue_pair[total_count].location.row = i - 1;
                        queue_pair[total_count].location.column = j;
                        queue_pair[total_count].counter = prev_counter + 1;
                        list = insert_at_foot(list,total_count);
                        matrix[i-1][j] = SPECIAL_CHAR;
                    }
                    if((j-1) >= 0 && (j-1) < coordinate_t.dimension_column
                        && matrix[i][j-1] == EMPTY_CHAR){
                        total_count++;
                        queue_pair[total_count].location.row = i;
                        queue_pair[total_count].location.column = j - 1;
                        queue_pair[total_count].counter = prev_counter + 1;
                        list = insert_at_foot(list,total_count);
                        matrix[i][j-1] = SPECIAL_CHAR;
                    }
                    if((j+1) >= 0 && (j+1) < coordinate_t.dimension_column
                        && matrix[i][j+1] == EMPTY_CHAR){
                        total_count++;
                        queue_pair[total_count].location.row = i;
                        queue_pair[total_count].location.column = j + 1;
                        queue_pair[total_count].counter = prev_counter + 1;
                        list = insert_at_foot(list,total_count);
                        matrix[i][j+1] = SPECIAL_CHAR;
                    }
                    //Finished checking so now increment the list value;
                    list_num = prev_list_num + 1;
                    // Increment the counter


                    //Now check if one of the four conditions is in the broken segment list then found = 1
                    for(route_num = broken_route + 2;
                        route_num < coordinate_t.num_route; route_num++){
                        row = route_list[route_num].row;
                        column = route_list[route_num].column;
                        if(row == (i+1) && column == j){
                            total_count++;
                            found = 1;
                            queue_pair[total_count].location.row = i + 1;
                            queue_pair[total_count].location.column = j;
                            queue_pair[total_count].counter = prev_counter + 1;
                            list = insert_at_foot(list,total_count);
                            //printf("Found at [%d,%d]\n",row,column);
                        }
                        if(row == (i-1) && column == j){
                            found = 1;
                            total_count++;
                            queue_pair[total_count].location.row = i - 1;
                            queue_pair[total_count].location.column = j;
                            queue_pair[total_count].counter = prev_counter + 1;
                            list = insert_at_foot(list,total_count);
                            //printf("Found at [%d,%d]\n",row,column);
                        }
                        if(row == i && column == (j-1)){
                            found = 1;
                            total_count++;
                            queue_pair[total_count].location.row = i;
                            queue_pair[total_count].location.column = j - 1;
                            queue_pair[total_count].counter = prev_counter + 1;
                            list = insert_at_foot(list,total_count);
                            //printf("Found at [%d,%d]\n",row,column);
                        }
                        if(row == i && column == (j+1)){
                            found = 1;
                            total_count++;
                            queue_pair[total_count].location.row = i;
                            queue_pair[total_count].location.column = j + 1;
                            queue_pair[total_count].counter = prev_counter + 1;
                            list = insert_at_foot(list,total_count);
                            //printf("Found at [%d,%d]\n",row,column);
                        }

                    }
                }
            }
        }
        count++;
    }
    // Store the total count into struct repair_t
    repair_t.total_count = total_count;

    repair_t.count = count;
    // Print out the array of structure of linked list
    while (!is_empty_list(list)){
        list_num = get_head(list);
        /*printf("([%d,%d], %d)\n", queue_pair[list_num].location.row,
        queue_pair[list_num].location.column, queue_pair[list_num].counter);*/
        list = get_tail(list);
    }
    free_list(list);
    list = NULL;
}

/* Construct a route fragment between cell s at which the broken
    segment starts and cell t from the last pair in the queue by walking from
    cell t towards cell s(GOING BACKWARD!!!). by progressing, at each cell,
    towards an adjacent cell with the smallest value; if multiple adjacent
    cells have the same counter value, the prederence is given to the one
    that comes earlier in this list: above, below, left , right
*/
void
route_fragment(char **matrix,list_t *list, list_t *repair_list){
    int i,j;
    int prev_row, prev_column;
    int row, column, counter;
    int prev_counter;

    list = make_empty_list();
    i = repair_t.total_count;
    list = insert_at_foot(list,i);
    prev_row = queue_pair[i].location.row ;
    prev_column =  queue_pair[i].location.column;
    prev_counter = queue_pair[i].counter;
    while(i >= 0){
        for(j = 0;j <= i - 1; j++){
            /*printf("([%d,%d], %d)\n",queue_pair[i].location.row,
            queue_pair[i].location.column, queue_pair[i].counter);
            */
            row = queue_pair[j].location.row ;
            column =  queue_pair[j].location.column;
            counter = queue_pair[j].counter;
            // Insert adjacent coordinates based on above, below, left, right
            if(row == prev_row + 1 && column == prev_column
                && counter == prev_counter - 1){
                list = insert_at_head(list,j);
                prev_row = queue_pair[j].location.row ;
                prev_column =  queue_pair[j].location.column;
                prev_counter = queue_pair[j].counter;
            }else if(row == prev_row - 1 && column == prev_column
                && counter == prev_counter - 1){
                list = insert_at_head(list,j);
                prev_row = queue_pair[j].location.row ;
                prev_column =  queue_pair[j].location.column;
                prev_counter = queue_pair[j].counter;
            }else if(row == prev_row  && column == prev_column - 1
                && counter == prev_counter - 1){
                list = insert_at_head(list,j);
                prev_row = queue_pair[j].location.row ;
                prev_column =  queue_pair[j].location.column;
                prev_counter = queue_pair[j].counter;
            }else if(row == prev_row && column == prev_column + 1
                && counter == prev_counter - 1){
                list = insert_at_head(list,j);
                prev_row = queue_pair[j].location.row ;
                prev_column =  queue_pair[j].location.column;
                prev_counter = queue_pair[j].counter;
            }
        }
        i--;
    }
    int counter_value = repair_t.counter_value;
    i = get_head(list);
    list = get_tail(list);
    // Print out the array of structure of linked list
    while (!is_empty_list(list)){
        i = get_head(list);
        repair_route_list[counter_value].row = queue_pair[i].location.row;
        repair_route_list[counter_value].column = queue_pair[i].location.column;
        insert_at_foot(repair_list, counter_value);
        counter_value++;
        list = get_tail(list);
    }
    free_list(list);
    list = NULL;

    //Now add the last part into the repair list;
    for(i = repair_t.counter_value + 2; i < coordinate_t.num_route; i++){
        row = route_list[i].row;
        column = route_list[i].column;
        repair_route_list[counter_value].row = row;
        repair_route_list[counter_value].column = column;
        insert_at_foot(repair_list,counter_value);
        counter_value++;
    }
    repair_t.counter_value = counter_value;
}

// Get the Dimension Row and Column
void
get_dimension(){
    int x,y;
    scanf("%dx%d ",&x,&y);
    coordinate_t.count++;
    coordinate_t.dimension_row = x;
    coordinate_t.dimension_column = y;

}

// Get the Row and Column of each block including inital cell and goal cell
void
get_block(char **matrix){
    int row,column;
    while(scanf("[%d,%d] ", &row, &column) == 2){
        coordinate_t.count++;
        if(coordinate_t.count == 2){
            coordinate_t.initial_row = row;
            coordinate_t.initial_column = column;
        } else if(coordinate_t.count == 3){
            coordinate_t.goal_row = row;
            coordinate_t.goal_column = column;
        } else {
            matrix[row][column] = BLOCK;
            block_list[coordinate_t.num_block].row = row;
            block_list[coordinate_t.num_block].column = column;
            coordinate_t.num_block++;
        }
    }
}

// Get the row and column of each routes
void
get_route(char **matrix, list_t *list){
    int row,column;
    int prev_row, prev_column;
    while(coordinate_t.flag){
        if(scanf("$ [%d,%d]",&row,&column) == 2){ //Get the initial cell
            matrix[row][column] = START;
            coordinate_t.flag = 0;
            check_initial_cell(row, column);
            // If input line 2 is not the same as the starting route

            // Store the route’s row and column into an array of structures
            route_list[coordinate_t.num_route].row = row;
            route_list[coordinate_t.num_route].column = column;
            // Put it into a linked list
            list = insert_at_foot(list,coordinate_t.num_route);

            coordinate_t.num_route++;
            prev_row = row;
            prev_column = column;
        }

        while(scanf(" -> [%d,%d]",&row, &column) == 2){ // Get rest of routes
```

```
            /* Check for Stage 3 if the route contains a move that traverses
            more than one cell.
            */
            if((row == prev_row + 1 && column == prev_column) ||
                (row == prev_row - 1 && column == prev_column) ||
                (row == prev_row && column == prev_column + 1) ||
                (row == prev_row && column == prev_column - 1)){
                    prev_row = row;
                    prev_column = column;
                }else {
                    //Status have not changed before
                    if(coordinate_t.status == 0){
                        coordinate_t.status = 3;
                    }
                }
            /* Check for Stage 4 if there is a presence of a block at one of
            the cells visited in the route.
            */
            if(matrix[row][column] == BLOCK){
                if(coordinate_t.status == 0){
                    coordinate_t.status = 4;
                }
            }else{
                matrix[row][column] = ROUTE;
            }

            // Store the route’s row and column into an array of structures
            route_list[coordinate_t.num_route].row = row;
            route_list[coordinate_t.num_route].column = column;
            // Put it into a linked list
            list = insert_at_foot(list, coordinate_t.num_route);
            coordinate_t.num_route++;
        }
        // Get the final cell here to check
        matrix[row][column] = GOAL;
        check_goal_cell(row, column);
        coordinate_t.final_row = row;
        coordinate_t.final_column = column;
    }
}


/* Check if the first cell is different from the inital cell
    supplied at line 2 of the input */
int
check_initial_cell(int row, int column){
    if(row != coordinate_t.initial_row ||
        column != coordinate_t.initial_column){
        if(coordinate_t.status == 0){ //Status have not changed before
            coordinate_t.status = 1;
        }
            return 1;
    } else{
            return 0;
    }
}

/* Check if the last cell in the route is different from the goal cell
    given at line 3 of the input
*/
void
check_goal_cell(int row, int column){
    if(row != coordinate_t.goal_row || column != coordinate_t.goal_column){
        if(coordinate_t.status == 0){
            coordinate_t.status = 2;
        }
    }
}

/* Update the Status based on specific conditions
*/
void
status_func(int status) {
    if (status==1) {
        printf("Initial cell in the route is wrong!\n");
    }
    else if (status==2) {
        printf("Goal cell in the route is wrong!\n");
    }
    else if (status==3) {
        printf("There is an illegal move in this route!\n");
    }
    else if (status==4) {
        printf("There is a block on this route!\n");
    }
    else{
        printf("The route is valid!\n");
    }
}

/* prints the output for STAGE 0
 */
void
print_output(list_t *list) {
    printf("The grid has %d rows and %d columns.\n",
        coordinate_t.dimension_row, coordinate_t.dimension_column);
    printf("The grid has %d block(s).\n",coordinate_t.num_block);
    printf("The initial cell in the grid is [%d,%d].\n",
        coordinate_t.initial_row, coordinate_t.initial_column);
    printf("The goal cell in the grid is [%d,%d].\n",
        coordinate_t.goal_row, coordinate_t.goal_column);
    printf("The proposed route in the grid is:\n");

    // Print out the linked list
    int i;
    int count = 0;
    while (!is_empty_list(list)){
        i = get_head(list);
        printf("[%d,%d]",route_list[i].row,route_list[i].column);
        list = get_tail(list);
        count++;
        if(i == coordinate_t.num_route - 1){
            printf(".");
        }else{
            printf("->");
        }

        if(count == 5){
            printf(NEWLINE);
            count = 0;
        }
    }
    printf(NEWLINE);
    free_list(list);
    list = NULL;
    status_func(coordinate_t.status);
}

/* Initialize the entire Matrix with ’ ’
*/
void
initialize_matrix(char **matrix){
    int i,j;
    for(i = 0; i< coordinate_t.dimension_row ; i++){
        for(j = 0;j < coordinate_t.dimension_column; j++){
            matrix[i][j] = EMPTY_CHAR;
        }
    }
}

/* Print the header for Stage 1
*/
void
print_header_1(){
    int i;
    printf(" ");
    for(i = 0;i < coordinate_t.dimension_column; i++){
        printf("%d",i % 10);
    }
    printf(NEWLINE);

}

/* Print the visualization For Stage 1
*/
void
print_output_1(char **matrix){
    int i,j;
    for(i = 0; i< coordinate_t.dimension_row ; i++){
        printf("%d",i % 10);
        for(j = 0;j < coordinate_t.dimension_column ; j++){
            printf("%c", matrix[i][j]);
        }
        printf(NEWLINE);
    }
}

/* Test each pointer after any of the memory allocation routines has been used.
If the allocation fails, the pointer is NULL, and the program execution should
be aborted.
*/
void
exit_if_null(void *ptr, char *msg){
    if(!ptr){
        printf("unexpected null pointer: %s\n", msg);
        exit(EXIT_FAILURE);
    }
}

/* Seperate the visualization
*/
void
seperator_line(void){
    printf("---------------------------------------------\n");
}
/* prints the prompt indicating ready for input for Stage 0
 */
void
print_prompt_stage_1(void){
    printf("==STAGE 1========================================\n");
}

/* prints the prompt indicating ready for input for Stage 0
 */
void
print_prompt_stage_0(void) {
    printf("==STAGE 0========================================\n");
}

/* Print seperator line indicating end of program
*/
void
print_equal_seperator_line(void){
    printf("=============================================\n");
}

/* Linked List Structures
    Reference from pg 172 in Programiing, Problem Solving and Abstraction
*/

list_t
*make_empty_list(void){
    list_t *list;
    list = (list_t*)malloc(sizeof(*list));
    assert(list != NULL);
    list -> head = list -> foot = NULL;
    return list;

}

int
is_empty_list(list_t *list){
    assert(list != NULL);
    return list->head == NULL;
}

void
free_list(list_t *list){
    node_t *curr, *prev;
    assert(list != NULL);
    curr = list -> head;
    while(curr){
        prev = curr;
        curr = curr->next;
        free(prev);
    }
    free(list);
}

list_t
*insert_at_head(list_t *list, data_t value){
    node_t *new;
    new = (node_t*)malloc(sizeof(*new));
    assert(list!=NULL && new != NULL);
    new->data = value;
    new->next = list->head;
    list-> head = new;
    if(list -> foot == NULL){
        /*this is the first insertion into the list*/
        list -> foot = new;
    }
    return list;
}

list_t
*insert_at_foot(list_t *list, data_t value){
    node_t *new;
    new = (node_t*)malloc(sizeof(*new));
    assert(list!=NULL && new != NULL);
    new->data = value;
    new->next = NULL;
    if(list->foot == NULL){
        /* this is the first insertion into the line*/
        list->head = list->foot = new;
    } else {
        list->foot->next = new;
        list->foot = new;
    }
    return list;
}

data_t
get_head(list_t *list){
    assert(list != NULL && list->head != NULL);
    return list->head->data;
}

list_t
*get_tail(list_t *list){
    node_t *oldhead;
    assert(list != NULL && list -> head != NULL);
    oldhead = list->head;
    list->head = list->head->next;
    if(list->head == NULL){
        /* the only list node just got deleteed*/
        list->foot = NULL;
    }
    free(oldhead);
    return list;
}
//Algorithm is fun
```