



Provide normal and edge test cases for the code below:

```
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

typedef struct list {
    int elem;
    int capacity;
    int head;
    int tail;
    bool is_reversed;
    int64_t *arr;
} list;

void inflate(list *l);
void deflate(list *l);
void help_push_left(list *l, int64_t v);
void help_push_right(list *l, int64_t v);
bool help_pop_left(list *l);
bool help_pop_right(list *l);

list *make(int n, int64_t *seq);
void push_left(list *l, int64_t v);
void push_right(list *l, int64_t v);
bool pop_left(list *l);
bool pop_right(list *l);
int64_t peek_left(list *l);
int64_t peek_right(list *l);
int size(list *l);
bool empty(list *l);
int64_t get(list *l, int i);
void set(list *l, int i, int64_t v);
void reverse(list *l);

void inflate(list *l) {
    int new_capacity = l->capacity * 2;
    int64_t *new_arr = (int64_t *)malloc(sizeof(int64_t) * new_capacity);

    for (int i = 0; i < l->elem; i++) {
        new_arr[i] = (l->arr)[(l->head + i) % l->capacity];
    }

    free(l->arr);
    l->capacity = new_capacity;
    l->arr = new_arr;
    l->head = 0;
    l->tail = l->elem;
}

void deflate(list *l) {
    int new_capacity = l->capacity / 2;
    int64_t *new_arr = (int64_t *)malloc(sizeof(int64_t) * new_capacity);

    for (int i = 0; i < l->elem; i++) {
        new_arr[i] = (l->arr)[(l->head + i) % l->capacity];
    }

    free(l->arr);
    l->capacity = new_capacity;
    l->arr = new_arr;
    l->head = 0;
    l->tail = l->elem;
}

void help_push_left(list *l, int64_t v) {
    if (l->elem == l->capacity) {
        inflate(l);
    }

    l->head = (l->head - 1 + l->capacity) % l->capacity;
}
```

```

    l->head = (l->head + 1) % l->capacity;
    l->arr[l->head] = v;
    l->elem++;
}

void help_push_right(list *l, int64_t v) {
    if (l->elem == capacity) {
        inflate(l);
    }

    l->arr[l->tail] = v;
    l->tail = (l->tail + 1) % l->capacity;
    l->elem++;
}

bool help_pop_left(list *l) {
    if (l->elem == 0) {
        return false;
    }

    l->head = (l->head + 1) % l->capacity;
    l->elem--;
    ↓

    if (l->elem == l->capacity / 2 && l->capacity > 1) {
        deflate(l);
    }

    return true;
}

bool help_pop_right(list *l) {
    if (l->elem == 0) {
        return false;
    }

    l->tail = (l->tail - 1 + l->capacity) % l->capacity;
    l->elem--;
    ↓

    if (l->elem == l->capacity / 2 && l->capacity > 1) {
        deflate(l);
    }

    return true;
}

list *make(int n, int64_t *seq) {
    list *d_arr = (list *)malloc(sizeof(list));
    d_arr->elem = d_arr->capacity = n;
    d_arr->head = 0;
    d_arr->tail = n;
    d_arr->is_reversed = false;
    d_arr->arr = (int64_t *)malloc(sizeof(int64_t) * n);
    for(int i = 0; i < n; i++) {
        (d_arr->arr)[i] = seq[i];
    }
    return d_arr;
}

void push_left(list *l, int64_t v) {
    if (!l->is_reversed) {
        help_push_left(l, v);
    } else {
        help_push_right(l, v);
    }
}

void push_right(list *l, int64_t v) {
    if (!l->is_reversed) {
        help_push_right(l, v);
    } else {
        help_push_left(l, v);
    }
}

bool pop_left(list *l) {

```

```

        if (!l->is_reversed) {
            return help_pop_left(l);
        } else {
            return help_pop_right(l);
        }
    }

bool pop_right(list *l) {
    if (!l->is_rever ↓ {
        return help_pop_right(l);
    } else {
        return help_pop_left(l);
    }
}

int64_t peek_left(list *l) {
    if (l->elem == 0) {
        printf("Error. List is empty.\n");
        return -1;
    }

    if (!l->is_reversed) {
        return l->arr[l->head];
    } else {
        return l->arr[(l->tail - 1 + l->capacity) % l->capacity];
    }
}

int64_t peek_right(list *l) {
    if (l->elem == 0) {
        printf("Error. List is empty.");
        return -1; // Or handle appropriately
    }

    if (!l->is_reversed) {
        return l->arr[(l->tail - 1 + l->capacity) % l->capacity];
    } else {
        return l->arr[l->head];
    }
}

int size(list *l) {
    return l->elem;
}

bool empty(list *l) {
    if (l->elem == 0) {return true;}
    return false;
}

int64_t get(list *l, int i) {
    if (i >= 0 && i < l->elem) {
        if (!l->is_reversed) {
            return (l->arr)[(l->head + i) % l->capacity];
        } else {
            return (l->arr)[(l->tail - 1 - i + l->capacity) % l->capacity];
        }
    } else {
        printf("Index Error\n");
        return -1;
    }
}

void set(list *l, int i, int64_t v) {
    if (i >= 0 && i < l->elem) {
        if (!l->is_reversed) {
            (l->arr)[(l->head + i) % l->capacity] = v;
        } else {
            (l->arr)[(l->tail - 1 - i + l->capacity) % l->capacity] = v;
        }
    } else {
        printf("Inde ↓ or\n");
    }
}

```

```
void reverse(list *l) {
    if(l->is_reversed) {
        l->is_reversed = false;
    } else {
        l->is_reversed = true;
    }
}
```

Normal and Edge Test Cases for the Circular List Implementation

Here are test cases categorized as **normal** (expected usage) and **edge** (corner cases or unusual inputs) to test the behavior of the given implementation.

Normal Test Cases

1. Initialization and Basic Operations

- Input: `make(5, [10, 20, 30, 40, 50])`
- Operations:
 - `size()` : Expected: 5
 - `peek_left()` : Expected: 10
 - `peek_right()` : Expected: 50
 - `get(2)` : Expected: 30
 - `set(2, 35)`
 - `get(2)` : Expected: 35



2. Push and Pop Operations

- Input: `make(3, [1, 2, 3])`
- Operations:
 - `push_left(0)`
 - `push_right(4)`
 - `pop_left()` : Expected: 0
 - `pop_right()` : Expected: 4
 - `size()` : Expected: 3



3. Reversal Handling

- Input: `make(4, [11, 22, 33, 44])`
- Operations:
 - `reverse()`
 - `peek_left()` : Expected: 44
 - `peek_right()` : Expected: 11
 - `get(0)` : Expected: 44
 - `reverse()`
 - `peek_left()` : Expected: 11



4. Inflation (Dynamic Capacity Increase)

- Input: `make(2, [5, 10])`
- Operations:
 - Add items using `push_right(15), push_right(20), push_right(25)`
 - Expected `capacity` : 4 (initially 2, doubled upon reaching capacity)
 - Check all values: [5, 10, 15, 20, 25]



5. Deflation (Dynamic Capacity Decrease)

- Input: `make(6, [1, 2, 3, 4, 5, 6])`
- Operations:
 - Remove items using `pop_left()` until half capacity is reached.
 - Expected `capacity` : 3 (halved upon reducing to half elements)
 - Check remaining values: [4, 5,]



Edge Test Cases

1. Empty List Operations

- Input: `make(0, [])`
- Operations:
 - `pop_left()` : Expected: `false`
 - `pop_right()` : Expected: `false`
 - `peek_left()` : Expected: `Error message`
 - `peek_right()` : Expected: `Error` 

2. Single Element List

- Input: `make(1, [99])`
- Operations:
 - `peek_left()` : Expected: `99`
 - `peek_right()` : Expected: `99`
 - `pop_left()` : Expected: `true`
 - `pop_left()` : Expected: `false` (list now empty)

3. Reverse with Single Element

- Input: `make(1, [42])`
- Operations:
 - `reverse()` 
 - `peek_left()` : Expected: `42`
 - `peek_right()` : Expected: `42`

4. Index Out of Bounds

- Input: `make(3, [1, 2, 3])`
- Operations:
 - `get(-1)` : Expected: `Error message`
 - `get(3)` : Expected: `Error message`
 - `set(3, 99)` : Expected: `Error message`

5. Stress Test: Large Number of Operations

- Input: `make(10, [0, 1, 2, ..., 9])` 
- Operations:
 - Perform 1,000,000 random operations (`push_left`, `push_right`, `pop_left`, `pop_right`, etc.)
 - Ensure integrity and consistency (e.g., size is always correct, no memory leaks).

6. Reverse and Dynamic Capacity

- Input: `make(4, [10, 20, 30, 40])`
- Operations:
 - `reverse()`
 - Add elements to cause inflation: `push_left(50), push_left(60)`
 - Verify order after inflation: `[60, 40, 30, 20, 10]` 
 - Remove elements to cause deflation and verify order.

7. Repeated Reverse

- Input: `make(3, [100, 200, 300])`
- Operations:
 - `reverse()` multiple times (10+)
 - Ensure values remain consistent (e.g., alternating between `[300, 200, 100]` and `[100, 200, 300]`).

Notes

- Ensure proper memory management for all edge cases, especially during inflation, deflation, and repeated reversals.
- Monitor for segmentation faults, especially during operations on empty lists or invalid indices.
- Verify capacity adjustments (doubling or halving) only occur at the correct thresholds.

Turn them into code so that I can copy and paste them.



Here's the test suite in C, covering normal and edge cases for your circular list implementation. This code assumes the functions and structure from your provided code are already defined.

```
c                                     Copy code
↓
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <stdbool.h>

// Helper function to print test results
void print_result(const char* test_name, bool result) {
    printf("%s: %s\n", test_name, result ? "PASSED" : "FAILED");
}

// Test suite
void run_tests() {
    // Normal Test Cases
    {
        // Test 1: Initialization and Basic Operations
        int64_t seq1[] = {10, 20, 30, 40, 50};
        list* l1 = make(5, seq1);
        print_result("Test 1 - Size Check", size(l1) == 5);
        print_result("Test 1 - Peek Left", peek_left(l1) == 10);
        print_result("Test 1 - Peek Right", peek_right(l1) == 50);
        print_result("Test 1 - Get Index 2", get(l1, 2) == 30);
        set(l1, 2, 35);
        print_result("Test 1 - Set Index 2", get(l1, 2) == 35);
        free(l1);
    }

    {
        // Test 2: Push and Pop Operations
        int64_t seq2[] = {1, 2, 3};
        list* l2 = make(3, seq2);
        push_left(l2, 0);
        push_right(l2, 4);
        print_result("Test 2 - Push Left", peek_left(l2) == 0);
        print_result("Test 2 - Push Right", peek_right(l2) == 4);
        pop_left(l2);
        pop_right(l2);
        print_result("Test 2 - Size After Pops", size(l2) == 3);
        free(l2);
    }

    {
        // Test 3: Reversal Handling
        int64_t seq3[] = {11, 22, 33, 44};
        list* l3 = make(4, seq3);
        reverse(l3);
        print_result("Test 3 - Reverse Peek Left", peek_left(l3) == 44);
        print_result("Test 3 - Reverse F↓ Right", peek_right(l3) == 11);
        reverse(l3);
        print_result("Test 3 - Re-Reverse Peek Left", peek_left(l3) == 11);
        free(l3);
    }

    {
        // Test 4: Inflation
        int64_t seq4[] = {5, 10};
        list* l4 = make(2, seq4);
        push_right(l4, 15);
        print_result("Test 4 - Inflation", size(l4) == 3);
        free(l4);
    }
}
```

```

        push_right(l4, 20);
        push_right(l4, 25);
        print_result("Test 4 - Inflation Capacity", l4->capacity == 4);
        free(l4);
    }

    {
        // Test 5: Deflation
        int64_t seq5[] = {1, 2, 3, 4, 5, 6};
        list* l5 = make(6, seq5);
        pop_left(l5);
        pop_left(l5);
        pop_left(l5);
        print_result("Test 5 - Deflation Capacity", l5->capacity == 3);
        free(l5);
    }

    // Edge Test Cases
    {
        // Test 6: Empty List Operations
        list* l6 = make(0, NULL);
        print_result("Test 6 - Pop Left on Empty", !pop_left(l6));
        print_result("Test 6 - Peek Left on Empty", peek_left(l6) == -1);
        free(l6);
    }

    {
        // Test 7: Single Element List
        int64_t seq7[] = {99};
        list* l7 = make(1, seq7);
        print_result("Test 7 - Peek Left", peek_left(l7) == 99);
        pop_left(l7);
        print_result("Test 7 - Pop Left Result", size(l7) == 0);
        free(l7);
    }

    {
        // Test 8: Reverse and Dynamic Capacity
        int64_t seq8[] = {10, 20, 30, 40};
        list* l8 = make(4, seq8);
        reverse(l8);
        push_left(l8, 50);
        push_left(l8, 60);
        print_result("Test 8 - Reverse with Inflation", peek_left(l8) == 60);
        free(l8);
    }

    {
        // Test 9: Repeated Reverse
        int64_t seq9[] = {100, 200, 300};
        list* l9 = make(3, seq9);
        for (int i = 0; i < 10; ++i) {
            reverse(l9);
        }
        print_result("Test 9 - Repeated Reverse Consistency", peek_left(l9) == 100);
        free(l9);
    }

    int main() {
        run_tests();
        return 0;
    }
}

```

How to Use

- Save this code to a file, e.g., `test_list.c`.
- Include the implementation file with your `list` structure and functions in the same directory or ensure it is linked during compilation.
- Compile using `gcc test_list.c -o test_list`.
- Run the test executable: `./test_list`.

The output will display whether each test case passed or failed.

ⓘ ⓘ ⓘ ⓘ ⓘ ⓘ

Message ChatGPT



ChatGPT can make mistakes. Check important info.

?