1. Dynamic Array

Array.h file:

```
#define ARRAY_FULL 0
#define SUCCESS 1
#define NOT_FOUND 0
#define FOUND 1

struct _array_dynamic_
{
    int *arr;
    int c_size, t_size;
};
typedef struct _array_dynamic_ Array_dyn;

struct _max_min_
{
    int max, min;
};
typedef struct _max_min_ Maxmin;

Array_dyn * initialize_dynamic_array(int size);
int insert_data_dyn(Array_dyn *, int data);
int search(Array_dyn *, int element);
int intersection(Array_dyn *, Array_dyn *);
int max_min(Array_dyn *, Maxmin);
int is_palindrome(Array_dyn *);
int sort_array(Array_dyn *);
// merge_array() : content of src array is copied to tgt array
int merge_array(Array_dyn *src, Array_dyn *tgt);
int deallocate(Array_dyn *);
```

Operation.c file:

```
#include<stdlib.h>
#include "array.h"

Array_dyn * initialize_dynamic_array(int size)
{
    Array_dyn *dyn_arr;

    dyn_arr = (Array_dyn *)malloc(sizeof(Array_dyn));

    if(dyn_arr == NULL) return NULL;

    dyn_arr->c_size = 0;
    dyn_arr->t_size = size;
    dyn_arr->arr = (int *)malloc(sizeof(int) * size);

    return dyn_arr;
}

int insert_data_dyn(Array_dyn *dyn_arr, int data)
```

```
{
    if(dyn_arr->c_size == dyn_arr->t_size) return ARRAY_FULL;

    *(dyn_arr->arr + dyn_arr->c_size) = data;
    dyn_arr->c_size++;

    return SUCCESS;
}

int search(Array_dyn *dyn_arr, int element)
{
    int i;

    for(i=0; i<dyn_arr->c_size;i++)
        if(*(dyn_arr->arr + i) == element) return FOUND;
    return NOT_FOUND;
}

int deallocate(Array_dyn *);
```

main.c file:

```
#include <stdio.h>
#include <stdlib.h>
#include<assert.h>
#include "array.h"

int main()
{
    Array_dyn *test;

    test = initialize_dynamic_array(7);
    assert(test != NULL);
    assert(test->c_size == 0 && test->t_size == 7);
    assert(test->arr != NULL);

    assert(insert_data_dyn(test, 10));
    assert(insert_data_dyn(test, 20));
    assert(insert_data_dyn(test, 30));
    assert(insert_data_dyn(test, 40));
    assert(insert_data_dyn(test, 50));
    assert(insert_data_dyn(test, 60));
    assert(insert_data_dyn(test, 70) == SUCCESS);

    assert(test->c_size == test->t_size);

    assert(insert_data_dyn(test, 80) == ARRAY_FULL);

    assert(search(test, 50));
    assert(search(test, 70));
    assert(search(test, 10));
    assert(search(test, 90) == NOT_FOUND);
```

```
    return 0;
}
```

2. Array Dynamic Intersection

Header.h:

```
#define ARRAY_FULL 0
#define SUCCESS 1
#define NOT_FOUND 0
#define FOUND 1

struct _array_dynamic_
{
    int *arr;
    int c_size, t_size;
};
typedef struct _array_dynamic_ Array_dyn;

struct _max_min_
{
    int max, min;
};
typedef struct _max_min_ Maxmin;

Array_dyn * initialize_dynamic_array(int size);
int insert_data_dyn(Array_dyn *, int data);
int search(Array_dyn *, int element);
Array_dyn * intersection(Array_dyn *, Array_dyn *);
int max_min(Array_dyn *, Maxmin);
int is_palindrome(Array_dyn *);
int bubble_sort_array(Array_dyn *);
// merge_array() : content of src array is copied to tgt array
int merge_array(Array_dyn *src, Array_dyn *tgt);
Array_dyn * deallocate(Array_dyn *);
void display(Array_dyn *);
```

operation.c file:

```
#include<stdlib.h>
#include<stdio.h>
#include<assert.h>
#include "array.h"

Array_dyn * initialize_dynamic_array(int size)
{
    Array_dyn *dyn_arr;

    dyn_arr = (Array_dyn *)malloc(sizeof(Array_dyn));

    if(dyn_arr == NULL) return NULL;

    dyn_arr->c_size = 0;
```

```
    dyn_arr->t_size = size;
    dyn_arr->arr = (int *)malloc(sizeof(int) * size);

    return dyn_arr;
}

int insert_data_dyn(Array_dyn *dyn_arr, int data)
{
    if(dyn_arr->c_size == dyn_arr->t_size) return ARRAY_FULL;

    *(dyn_arr->arr + dyn_arr->c_size) = data;
    dyn_arr->c_size++;

    return SUCCESS;
}

int search(Array_dyn *dyn_arr, int element)
{
    int i;

    for(i=0; i< dyn_arr->c_size; i++)
        if(*(dyn_arr->arr + i) == element) return FOUND;
    return NOT_FOUND;
}

Array_dyn * deallocate(Array_dyn *dyn_arr)
{
    free(dyn_arr->arr);
    free(dyn_arr);

    return NULL;

}

Array_dyn * intersection(Array_dyn *arr_a, Array_dyn *arr_b)
{
    Array_dyn *arr_c;
    int i;

    if(arr_a->c_size == 0 || arr_b->c_size == 0) return NULL;

    arr_c = initialize_dynamic_array(arr_a->c_size);

    for(i=0; i<arr_a->c_size; i++)
    {
        if(search(arr_b, arr_a->arr[i]) && !search(arr_c, arr_a->arr[i] ))
                assert(insert_data_dyn(arr_c, arr_a->arr[i]));
    }
    return arr_c;
}


int max_min(Array_dyn *, Maxmin);
```

```c
int is_palindrome(Array_dyn *dy_arr)
{
    int i=0, j = dy_arr->c_size - 1;

    while(i < j)
        if(dy_arr->arr[i++] != dy_arr->arr[j--]) return 0;
    return 1;
}

int bubble_sort_array(Array_dyn *dy_arr)
{
    int i, j, temp, flag;

    if(dy_arr->c_size <= 1) return 0;

    for(i=0; i < dy_arr->c_size; i++)
    {
        flag = 0;
        for(j=0; j < dy_arr->c_size - i - 1; j++)
        {
            if(*(dy_arr->arr + j) > *(dy_arr->arr + j + 1) )
            {
                temp = dy_arr->arr[j];
                dy_arr->arr[j] = dy_arr->arr[j+1];
                dy_arr->arr[j+1] = temp;

                flag = 1;
            }

        }
        if(flag == 0) return 1;
    }
    return 1;
}

void display(Array_dyn *dy_arr)
{
    int i;

    for(i=0; i < dy_arr->c_size; i++)
        printf(" %d ", dy_arr->arr[i]);
}
```

Main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include<assert.h>
#include "array.h"

int main()
{
    Array_dyn *test, *two, *arr_three;
```

```
    two = initialize_dynamic_array(9);
    test = initialize_dynamic_array(7);
    assert(test != NULL);
    assert(test->c_size == 0 && test->t_size == 7);
    assert(test->arr != NULL);

    assert(insert_data_dyn(test, 10));
    assert(insert_data_dyn(test, 20));
    assert(insert_data_dyn(test, 30));
    assert(insert_data_dyn(test, 40));
    assert(insert_data_dyn(test, 30));
    assert(insert_data_dyn(test, 10));
    assert(insert_data_dyn(test, 10) == SUCCESS);

    assert(test->c_size == test->t_size);

    assert(insert_data_dyn(test, 80) == ARRAY_FULL);

    assert(insert_data_dyn(two, 40));
    assert(insert_data_dyn(two, 30));
    assert(insert_data_dyn(two, 10));
    assert(insert_data_dyn(two, 10));
    assert(insert_data_dyn(two, 90));
    assert(insert_data_dyn(two, 60));

    assert(search(test, 40));
    assert(search(test, 30));
    assert(search(test, 10));
    assert(search(test, 90) == NOT_FOUND);

   // test = deallocate(test);

   // assert(test == NULL);
   arr_three = intersection(test, two);
   printf("\n After Intersection data: ");
   display(arr_three);

   assert(arr_three->c_size == 3);

   assert(is_palindrome(test) == 0);

   assert(bubble_sort_array(test));

    printf("\n After sorting data : ");
   display(test);

   test=deallocate(test);
    return 0;
}
```

3. Static Arrray

Header.h file:

```
#ifndef ARRAY_STATIC
#define ARRAY_STATIC


#define MAX_SIZE 20
#define FOUND 1
#define NOT_FOUND 0

struct _array_
{
    int array[MAX_SIZE];
    int c_size, t_size;
};

typedef struct _array_ Array;

Array intialize_array(int);
Array insert_data(Array, int);
int search(Array, int);


#endif
```

Operation.h

```

#include "array.h"


Array intialize_array(int size)
{
    Array my_arr;

    my_arr.c_size = 0;
    my_arr.t_size = size > 0 && size <= MAX_SIZE? size: MAX_SIZE;

    return my_arr;
}


Array insert_data(Array my_arr, int data)
{
    if(my_arr.c_size == my_arr.t_size) return my_arr;

    my_arr.array[my_arr.c_size++] = data;

    return my_arr;
}
int search(Array my_arr, int data)
```

```
{
    int i;

    for(i=0; i<my_arr.c_size;i++)
    {
        if(my_arr.array[i] == data) return FOUND;
    }
    return NOT_FOUND;
}
```

Main.c

```
#include<assert.h>
#include "array.h"

int main()
{
    Array test_arr;

    //initialize array
    test_arr = intialize_array(10);
    assert(test_arr.c_size == 0 && test_arr.t_size == 10);

    //insert data in a array

    test_arr = insert_data(test_arr, 10);
    test_arr = insert_data(test_arr, 20);
    test_arr = insert_data(test_arr, 30);
    test_arr = insert_data(test_arr, 40);
    test_arr = insert_data(test_arr, 50);
    test_arr = insert_data(test_arr, 60);
    test_arr = insert_data(test_arr, 70);
    test_arr = insert_data(test_arr, 80);
    test_arr = insert_data(test_arr, 90);
    test_arr = insert_data(test_arr, 100);

    assert(test_arr.c_size == 10);
    assert(test_arr.array[9] == 100);

    //array full
    test_arr = insert_data(test_arr, 55);
    assert(test_arr.c_size == 10);

    assert(search(test_arr, 100));
    assert(search(test_arr, 70));
    assert(search(test_arr, 50));
    assert(search(test_arr, 55) == 0);

    return 0;
}
```

4. Employee_VLSI

Header.h

```
#define FULL 0
#define SUCCESS 1
#define NAME_SIZE 20
struct _result_set_
{
    int status;
    char *comment;
};
typedef struct _result_set_ Resultset;

struct _employee_
{
    int emp_id;
    char emp_name[NAME_SIZE];
    float emp_salary;
};
typedef struct _employee_ Employee;

struct _array_emp_
{
    Employee *arr;
    int c_size, t_size;
    float average_sal, sum;
};
typedef struct _array_emp_ Emparray;

Emparray * initialise_employee(int);
int insert_employee_data(Emparray *, Employee);
Employee search(Emparray *, char *);
Employee get_max_sal_emp_details(Emparray *);
int get_employee_count(Emparray *);
float get_average_sal(Emparray *);
void display(Emparray *);
```

Operation.c

```
#include <stdlib.h>
#include<stdio.h>
#include<string.h>
#include "employee.h"

Emparray * initialise_employee(int size)
{
    Emparray *emp;

    emp = (Emparray *)malloc(sizeof(Emparray));
    if(emp == NULL) return NULL;

    emp->c_size = 0;
    emp->t_size = size;
    emp->arr = (Employee *)malloc(sizeof(Employee) * size);
```

```
    return emp;
}
int insert_employee_data(Emparray *my_emp, Employee data)
{
    if(my_emp->c_size == my_emp->t_size) return FULL;

    *(my_emp->arr + my_emp->c_size) = data;
    my_emp->c_size++;

    return SUCCESS;
}

void display(Emparray *my_emp)
{
    int i;

    for(i=0;i < my_emp->c_size; i++)
        printf("\n %d, %s, %f", \
               (my_emp->arr + i)->emp_id, \
               (my_emp->arr + i)->emp_name, \
               (my_emp->arr + i)->emp_salary);
}


Employee search(Emparray * my_emp, char *name)
{
    int i;
    Employee dummy;

    strcpy(dummy.emp_name, "Invalid");
    dummy.emp_id = -1;

    for(i=0; i<my_emp->c_size;i++)
        if(!strcmp((my_emp->arr + i)->emp_name, name)) return *(my_emp->arr + i);

    return dummy;
}

Employee get_max_sal_emp_details(Emparray *);
int get_employee_count(Emparray *my_emp)
{
    return my_emp->c_size;
}
float get_average_sal(Emparray *);
```

Main.c

```
#include <string.h>
#include <stdlib.h>
#include <assert.h>
#include "employee.h"
```

```c
int main()
{
    Emparray *test;
    Employee data;

    test = initialise_employee(10);

    assert(test != NULL);
    assert(test->c_size == 0 && test->t_size == 10);
    assert(test->arr != NULL);

    data.emp_id = 1001;
    strcpy(data.emp_name, "Akshay");
    data.emp_salary = 100000;
    assert(insert_employee_data(test, data));


    data.emp_id = 1002;
    strcpy(data.emp_name, "Alia");
    data.emp_salary = 300000;
    assert(insert_employee_data(test, data));


    data.emp_id = 1003;
    strcpy(data.emp_name, "Prabhas");
    data.emp_salary = 400000;
    assert(insert_employee_data(test, data));


    data.emp_id = 1004;
    strcpy(data.emp_name, "Kareena");
    data.emp_salary = 600000;
    assert(insert_employee_data(test, data));


    data.emp_id = 1005;
    strcpy(data.emp_name, "Ranbir");
    data.emp_salary = 900000;
    assert(insert_employee_data(test, data));


    data.emp_id = 1006;
    strcpy(data.emp_name, "Deepika");
    data.emp_salary = 600000;
    assert(insert_employee_data(test, data));


    data.emp_id = 1007;
    strcpy(data.emp_name, "Salman");
    data.emp_salary = 900000;
    assert(insert_employee_data(test, data));

    assert(test->c_size == 7);
    assert(test->arr->emp_id == 1001);
```

```
    assert((test->arr + 6)->emp_id == 1007);

    display(test);

    data = search(test, "Aamir");
    assert(data.emp_id == -1);

    return 0;
}
```

5. Employee ESD

Header.h

```
struct _employee_
{
    int emp_id;
    char emp_name[10];
    float emp_salary;
};
typedef struct _employee_ Employee;


struct _emp_arr_
{
    Employee *arr;
    int c_size, t_size;
};

typedef struct _emp_arr_ Emp;

Emp * initialise_employee(int);
int insert_data_employee(Emp *, Employee);
int search(Emp *, char *);
```

Operation.c

```
#include<stdlib.h>
#include "employee.h"

Emp * initialise_employee(int size)
{
    Emp *my_emp;

    my_emp = (Emp *) malloc(sizeof(Emp));
    my_emp->c_size = 0;
    my_emp->t_size = size;
    my_emp->arr = (Employee *)malloc(sizeof(Employee) * size);

    return my_emp;
}
int insert_data_employee(Emp *my_emp, Employee data)
```

```
{
    if(my_emp->c_size == my_emp->t_size) return 0;

    *(my_emp->arr + my_emp->c_size) = data;
    my_emp->c_size++;

    return 1;
}
int search(Emp *, char *);
```

Main.c

```
#include <string.h>
#include <stdlib.h>
#include<assert.h>
#include "employee.h"

int main()
{
    Emp *test;
    Employee data;

    test = initialise_employee(10);
    assert(test->c_size == 0 && test->t_size == 10);

    data.emp_id = 1001;
    strcpy(data.emp_name, "Ajay");
    data.emp_salary = 10000;
    assert(insert_data_employee(test, data));


    data.emp_id = 1002;
    strcpy(data.emp_name, "Alia");
    data.emp_salary = 600000;
    assert(insert_data_employee(test, data));

    data.emp_id = 1003;
    strcpy(data.emp_name, "Salman");
    data.emp_salary = 300000;
    assert(insert_data_employee(test, data));

    data.emp_id = 1004;
    strcpy(data.emp_name, "Sara");
    data.emp_salary = 750000;
    assert(insert_data_employee(test, data));

    data.emp_id = 1005;
    strcpy(data.emp_name, "Rahul");
    data.emp_salary = 500000;
    assert(insert_data_employee(test, data));

    assert(test->c_size == 5);
    assert((test->arr + 4)->emp_id == 1005);
```

```
    return 0;
}
```

6. Linked List

Header.h

```
#define FAILED 0
#define INSERT_SUCCESS 1
#define DELETED 1

struct _node_
{
    int data;
    struct _node_ *ptr;
};
typedef struct _node_ Node;

struct _linklist_
{
    Node *head, *tail;
    int count;
};
typedef struct _linklist_ List;

List * initialise_list();
int insert_at_beg(List *, int data);
int insert_at_end(List *, int data);
int insert_after_data(List *, int data, int element);
int insert_at_pos(List *, int data, int pos);

int search_list(List *, int element);
int merge_list(List *, List *);
int split_list(List *, int split_index);

int delete_at_beg(List *);
int delete_at_end(List *);
int delete_element(List *, int element);

List * free_list(List *); // deallocate memory of the list
```

Operation.c

```
#include<stdlib.h>
#include "list.h"

List * initialise_list()
{
    List *my_list;

    my_list = (List *)malloc(sizeof(List));
```

```
    my_list->head = my_list->tail = NULL;
    my_list->count = 0;

    return my_list;
}

Node *get_node(int data)
{
    Node *new_node;

    new_node = (Node *)malloc(sizeof(Node));
    new_node->data = data;
    new_node->ptr = NULL;

    return new_node;
}
int insert_at_beg(List *my_list, int data)
{
    Node *new_node = get_node(data); //create new node and store data

    if(new_node == NULL) return FAILED;

    if(my_list->count == 0) my_list->head = my_list->tail = new_node;
    else
    {
        new_node->ptr = my_list->head;
        my_list->head = new_node;
    }
    my_list->count++;

    return INSERT_SUCCESS;
}
int insert_at_end(List *my_list, int data)
{
    Node *new_node = get_node(data); //create new node and store data

    if(new_node == NULL) return FAILED;

    if(my_list->count == 0) my_list->head = my_list->tail = new_node;
    else
    {
        my_list->tail->ptr = new_node;
        my_list->tail = new_node;
    }
    my_list->count++;

    return INSERT_SUCCESS;
}
int insert_after_data(List *my_list, int data, int element)
{
    Node * new_node, *temp;

    if(my_list->count == 0) return FAILED; // check if list is empty
    // check if the element is the last node then insert at the end
```

```
    if(my_list->tail->data == element) return insert_at_end(my_list, data);
    // traverse through the list and find the position of element
    for(temp = my_list->head; temp!=NULL && temp->data != element; temp = temp->ptr);
    // if element not found
    if(temp == NULL)
        return FAILED;
    // if element found
    new_node = get_node(data); // create new node
    //make necessary connection
    new_node->ptr = temp->ptr;
    temp->ptr = new_node;

    my_list->count++;

    return INSERT_SUCCESS;
}

//Lab exercise
int insert_at_pos(List *my_list, int data, int pos)
{
    Node *temp, *new_node;
    int i;
    // check if list is empty, if true return fail
    if(my_list->count==0) return FAILED;

    if(my_list->count==pos) return insert_at_end(my_list,data);
        temp=my_list->head;
        for(i=0;i<pos &&temp->ptr!=NULL; i++)
            temp=temp->ptr;
    //element not found
    if (temp == NULL) return FAILED;

    new_node=get_node(data);

    new_node->ptr=temp->ptr;
    temp->ptr=new_node;

    my_list->count++;

    return INSERT_SUCCESS;


}


int search_list(List *my_list, int element)
{
    Node *temp;

    for(temp = my_list->head; temp != NULL; temp = temp->ptr)
        if(temp->data == element) return 1;
    return 0;
}
```

```c
int delete_at_beg(List *my_list)
{
    Node *temp;
    // check if list is empty, if true return fail
    if(my_list->count == 0) return 0;

    // check if list has one element
    if(my_list->count == 1)
    {
        free(my_list->head);
        my_list->head = NULL;
        my_list->tail = NULL;
    }
    //move the head node
    else
    {
        temp = my_list->head;
        my_list->head = my_list->head->ptr;
        free(temp); //free the previous head
    }

    // decrement the count
    my_list->count--;
    // return succeed
    return DELETED;
}

//Lab Exercise
int delete_at_end(List *my_list)
{
    Node *temp=my_list->head;
    // check if list empty
    if(my_list->count==0) return 0;
    // check if list has one element
    if(my_list->count==1)
    {
        free(my_list->head);
        my_list->head = NULL;
        my_list->tail = NULL;
    }
    // go to N-1 node
    else
    {
        while(temp->ptr!=my_list->tail)
            temp=temp->ptr;
        // free Nth node
        free(my_list->tail);
        my_list->tail=temp;
        my_list->tail->ptr=NULL;
        // decrement the counter
        my_list->count--;
    }
    return DELETED;
}
```

```
// Lab Exercise

int delete_element(List *my_list, int element)
{
    Node *temp=my_list->head;
    Node *previous;
    //int i;
    // check id list is empty
    if(my_list->count==0) return 0;
    // check if the element is at head position
    if(my_list->count==1)
    {
        free(my_list->head);
        my_list->head = NULL;
        my_list->tail = NULL;
    }
    //check if it is last element
    else{
        while (temp != NULL && temp->data != element)
        {
            previous = temp;
            temp = temp->ptr;
        }
    // if not return FAIL
    if (temp == NULL) return FAILED;
    previous->ptr = temp->ptr;
    // if found, free the node
    free (temp);
    }
    //decrement count
    my_list->count--;
    //return DELETED
    return DELETED;
}
void display_list(List *my_list)
{
    Node *current = my_list->head;

    printf("List contents: ");
    while(current!=NULL)
    {
        printf("%d -> ",current->data);
        current=current->ptr;
    }
    printf("NULL\n");
}
List * free_list(List *); // deallocate memory of the list
```

Mian.c

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include<assert.h>
#include "list.h"

int main()
{
    List *test_list;

    test_list = initialise_list();
    assert(test_list->head == NULL && test_list->tail == NULL);
    assert(test_list->count == 0);

    assert(insert_at_beg(test_list, 10));
    assert(test_list->head == test_list->tail);
    assert(test_list->count == 1);

    assert(insert_at_beg(test_list, 20));
    assert(insert_at_beg(test_list, 30));
    assert(insert_at_beg(test_list, 40));
    assert(insert_at_beg(test_list, 50));
    assert(insert_at_beg(test_list, 60));

    assert(test_list->count == 6);
    assert(test_list->head->data == 60);
    assert(test_list->tail->data == 10);
    assert(test_list->head->ptr->ptr->ptr->ptr->data == 20);

    assert(search_list(test_list, 10));
    assert(search_list(test_list, 20));
    assert(search_list(test_list, 30));
    assert(search_list(test_list, 40));
    assert(search_list(test_list, 50));
    assert(search_list(test_list, 60));

    assert(search_list(test_list, 70) == 0);

    assert(insert_at_end(test_list, 70));
    assert(insert_at_end(test_list, 80));
    assert(insert_at_end(test_list, 90));

    assert(test_list->count == 9);
    assert(test_list->tail->data == 90);

    assert(insert_after_data(test_list, 100, 90));
    assert(test_list->tail->data == 100);

    assert(insert_after_data(test_list, 100, 90));

    assert(insert_after_data(test_list, 110, 50));
    assert(test_list->head->ptr->ptr->data == 110);
    assert(search_list(test_list,110) == 1);
    assert(test_list->count ==12);

    assert(delete_at_beg(test_list));
```

```
    assert(delete_at_beg(test_list));
    assert(delete_at_beg(test_list));

    assert(test_list->head->data == 40);
    assert(test_list->count == 9);

    assert(delete_at_end(test_list));
    assert(test_list->tail->data==100);
    assert(test_list->count ==8);

    assert(delete_element(test_list,70));
    assert(test_list->count ==7);

    assert(insert_at_pos(test_list,230,3));
    assert(test_list->count==8);

    assert(insert_at_beg(test_list, 10));
    assert(insert_at_beg(test_list, 20));
    assert(insert_at_beg(test_list, 30));
    assert(delete_at_beg(test_list));
    assert(delete_at_end(test_list));
    display_list(test_list);
    return 0;
}
```

7. Queue

Header.h

```
#ifndef _INCLUDED_QUEUE_
#define _INCLUDED_QUEUE_

#include <stdint.h>

#define MAX_QUEUE_LEN 32

struct _queue_ {
    uint32_t size;              /* actual size */
    uint32_t count;             /* occupied */
    uint32_t head;              /* removing end */
    uint32_t tail;              /* add new items here */
    int32_t  q[MAX_QUEUE_LEN];  /* queue data */
};
typedef struct _queue_ Queue;

struct _queue_result_ {
    int32_t data;
    uint32_t status;
};
typedef struct _queue_result_ QueueResult;


#define QUEUE_OK     1
```

```
#define QUEUE_FULL   2
#define QUEUE_EMPTY  4

Queue    queue_new(uint32_t size);
Queue*   queue_add(Queue *q, int32_t data, QueueResult *result);
Queue*   queue_remove(Queue *q, QueueResult *result);
uint32_t queue_full(Queue *q);
uint32_t queue_empty(Queue *q);

#endif
```

Operation.c

```
#include <assert.h>
#include <stddef.h>
#include "queue.h"

Queue  queue_new(uint32_t size)
{
    size = (size > 0 && size < MAX_QUEUE_LEN) ? size : MAX_QUEUE_LEN;

    Queue q = {size, /*count*/ 0, /*head*/0, /*tail*/0, /*data*/{0} };

    return q;
}

Queue* queue_add(Queue *q, int32_t data, QueueResult *result)
{
    assert(q != NULL && q->count <= q->size);

    if (q->count < q->size) {
        q->q[q->tail] = data;
        q->tail = (q->tail + 1) % q->size;
        ++q->count;
        result->status = QUEUE_OK;
    } else {
        result->status = QUEUE_FULL;
    }

    assert(result->status == QUEUE_OK || q->count == q->size);
    return q;
}

Queue* queue_remove(Queue *q, QueueResult *result)
{
    assert(q != NULL && q->count <= q->size);

    if (q->count > 0) {
        result->data = q->q[q->head];
        q->head = (q->head + 1) % q->size;
        --q->count;
        result->status = QUEUE_OK;
```

```
    } else {
        result->status = QUEUE_EMPTY;
    }

    assert(q->count < q->size);
    assert(result->status == QUEUE_OK || q->count == 0);

    return q;
}

uint32_t queue_full(Queue *q)
{
    assert(q != NULL && q->count <= q->size);
    return (q->count == q->size);
}

uint32_t queue_empty(Queue *q)
{
    return (q->count == 0);
}
```

Main.c

```
#include <assert.h>
#include "queue.h"

void test_one_element_queue()
{
    Queue q = queue_new(1);
    QueueResult res;

    assert(queue_empty(&q));
    assert(!queue_full(&q));

    queue_add(&q, 100, &res);
    assert(res.status == QUEUE_OK);

    assert(!queue_empty(&q));
    assert(queue_full(&q));

    queue_add(&q, 100, &res);
    assert(res.status == QUEUE_FULL);

    queue_remove(&q, &res);
    assert(res.status == QUEUE_OK);

    assert(queue_empty(&q));
    assert(!queue_full(&q));
}

void test_two_element_queue()
{
```

```
    Queue q = queue_new(2);
    QueueResult res;

    assert(queue_empty(&q));
    assert(!queue_full(&q));

    queue_add(&q, 100, &res);
    assert(res.status == QUEUE_OK);
    assert(!queue_empty(&q));
    assert(!queue_full(&q));

    queue_add(&q, 200, &res);
    assert(res.status == QUEUE_OK);
    assert(!queue_empty(&q));
    assert(queue_full(&q));

    queue_add(&q, 300, &res);
    assert(res.status == QUEUE_FULL);
    assert(!queue_empty(&q));
    assert(queue_full(&q));

    queue_remove(&q, &res);
    assert(res.data == 100 && res.status == QUEUE_OK);
    assert(!queue_empty(&q));
    assert(!queue_full(&q));

    queue_remove(&q, &res);
    assert(res.data == 200 && res.status == QUEUE_OK);
    assert(queue_empty(&q));
    assert(!queue_full(&q));

    queue_remove(&q, &res);
    assert(res.status == QUEUE_EMPTY);
    assert(queue_empty(&q));
    assert(!queue_full(&q));
}

void test_generic_element_queue()
{
    Queue q = queue_new(0);
    QueueResult res;

    assert(queue_empty(&q));
    assert(!queue_full(&q));

    int32_t i;
    for (i = 0; i < MAX_QUEUE_LEN; ++i) {
        queue_add(&q, i, &res);
        assert(res.status == QUEUE_OK);
    }
    assert(queue_full(&q));

    for (i = 0; i < MAX_QUEUE_LEN; ++i) {
        queue_remove(&q, &res);
```

```
        assert(res.data == i && res.status == QUEUE_OK);
    }
    assert(queue_empty(&q));
}

int main()
{
    test_one_element_queue();

    test_two_element_queue();

    test_generic_element_queue();

    return 0;
}
```

8. Stack

Header.h

```
#ifndef _INCLUDED_STACK_H_
#define _INCLUDED_STACK_H_

#include <stdint.h>

#define MAX_DEPTH 32

struct _stack_ {
    uint32_t size;              /* requested stack depth */
    int32_t  top;               /* index of the topmost element */
    float    data[MAX_DEPTH];   /* actual contents */
};
typedef struct _stack_ *Stack;

/* status bits used in the Result structure */
#define RESULT_INVALID    0
#define STACK_OK          1
#define STACK_FULL        2
#define STACK_EMPTY       4

struct _stack_result_ {
    float data;
    uint32_t status;
};
typedef struct _stack_result_ StackResult;

/* The ADT interface */
/*
 * NOTE: we copy Stack instances across procedure calls.
 * This is inefficient and not-completely-right. But till we learn explicit
 * memory management using 'malloc' and 'free', we continue to pass
 * objects by value than by address.
 */
```

```
Stack    stack_new  (uint32_t size);
void     stack_delete(Stack stk);
uint32_t stack_full (const Stack stk);
uint32_t stack_empty(const Stack stk);
Stack   stack_push (Stack stk, float data, StackResult *result);
Stack   stack_pop  (Stack stk, StackResult *result);
Stack   stack_peek (const Stack stk, StackResult *result);

#endif
```

Operation.c

```
#include <assert.h>
#include <string.h>
#include <stdlib.h>
#include <stddef.h>
#include "stack.h"

/*
 * We deal with small-size stack data structure.
 * Stack objects are passed to functions by reference. This is more efficient
 *   than passing them by value.
 */
Stack stack_new(uint32_t size)
{
    size = (size > 0 && size < MAX_DEPTH) ? size : MAX_DEPTH;

    Stack stk = (Stack) malloc(sizeof(struct _stack_));
    assert(stk != NULL);

    stk->size = size;
    stk->top = -1;
    memset(stk->data, 0, sizeof(stk->data));

    return stk;
}

void stack_delete(Stack stk)
{
    assert(stk != NULL);
    free(stk);
}

uint32_t stack_full(const Stack stk)
{
    assert(stk != NULL);
    return ((stk->top + 1) == stk->size);
}

uint32_t stack_empty(const Stack stk)
{
    assert(stk != NULL);
```

```
    return stk->top == -1;
}

Stack stack_push(Stack  stk, float data, StackResult *result)
{
    assert(stk != NULL);
    if (stk->top + 1 < stk->size) {
        stk->data[++stk->top] = data;
        result->data = data;
        result->status = STACK_OK;
    } else {
        result->data = data;
        result->status = STACK_FULL;
    }

    return stk;
}

Stack stack_pop(Stack  stk, StackResult *result)
{
    assert(stk != NULL);
    if (stk->top > -1) {
        result->data = stk->data[stk->top];
        result->status = STACK_OK;
        --stk->top;
    } else {
        result->status = STACK_EMPTY;
    }

    return stk;
}

Stack stack_peek(const Stack  stk, StackResult *result)
{
    assert(stk != NULL);
    if (stk->top > -1) {
        result->data = stk->data[stk->top];
        result->status = STACK_OK;
    } else {
        result->status = STACK_EMPTY;
    }

    return stk;
}
```

Main.c

```
#include <assert.h>
#include <stddef.h>
#include "stack.h"

void test_capacity_one_stack()
```

```
{
    Stack stk = stack_new(1);
    StackResult result;

    assert(stack_empty(stk));
    assert(!stack_full(stk));

    stack_peek(stk, &result);
    assert(result.status == STACK_EMPTY);
    stack_pop(stk, &result);
    assert(result.status == STACK_EMPTY);

    stack_push(stk, 99, &result);
    assert(result.status == STACK_OK);
    assert(stack_full(stk));

    stack_push(stk, 222, &result);
    assert(result.status == STACK_FULL);
    stack_peek(stk, &result);
    assert(result.data == 99 && result.status == STACK_OK);

    stack_pop(stk, &result);
    assert(result.data == 99 && result.status == STACK_OK);
    assert(stack_empty(stk));

    stack_delete(stk), stk = NULL;
}

void test_arbitrary_stack()
{
    Stack stk = stack_new(0);
    StackResult result = { 0, RESULT_INVALID };
    int i;

    for (i = 0; i < MAX_DEPTH; i++) {
        stack_push(stk, i, &result);
        assert(result.status == STACK_OK);
        result.status = RESULT_INVALID;
    }

    stack_push(stk, i, &result);
    assert(result.status == STACK_FULL);

    for (i = 0; i < MAX_DEPTH; i++) {
        result.status = 0;
        stack_peek(stk, &result);
        assert(result.status == STACK_OK);
        assert(result.data == MAX_DEPTH - i - 1);

        result.status = RESULT_INVALID;
        stack_pop(stk, &result);
        assert(result.status == STACK_OK);
    }
    assert(stack_empty(stk));
```

```
    stack_delete(stk), stk = NULL;
}

int main()
{
    test_capacity_one_stack();

    test_arbitrary_stack();

    return 0;
}
```

9. Split

Header.h

```
#define SUCCESS 1
#define ARRAY_FULL 0
#define SPIT_FAILED NULL;

struct _array_
{
    int *arr;
    int c_size, t_size;
};
typedef struct _array_ Array;


struct split_array_reference
{
    int *index_ref;
    int count;
};
typedef struct split_array_reference Split;

Split * initialise_split_array(int size);

Array * initialise_array(int);
int insert_data(Array *, int);
Split * split_array(Array *, int split);
```

Operation.h

```
#include<stdlib.h>
#include "array.h"

Array * initialise_array(int size)
{
    Array *my_arr;

    my_arr = (Array *)malloc(sizeof(Array));
```

```
    if(my_arr == NULL) return NULL;

    my_arr->c_size = 0;
    my_arr->t_size = size;
    my_arr->arr = (int *)malloc(sizeof(int) * size);

    return my_arr;
}
int insert_data(Array *my_arr, int data)
{
    // Check if array is FULL return 0 if true
    if(my_arr->c_size == my_arr->t_size) return ARRAY_FULL;

    //Insert data at c_size position with pointer arithmetic
    *(my_arr->arr + my_arr->c_size) = data;
    my_arr->c_size++; // Increment the c_size

    return SUCCESS; // returns 1
}

Split * initialise_split_array(int size)
{
    Split *index;

    index = (Split *)malloc(sizeof(Split) * size);

    return index;
}


Split * split_array(Array *my_arr, int split)
{
    int sub_part, i, j, rem;;
    Split *index;

    if(split > my_arr->c_size/2) return SPIT_FAILED;

    rem = my_arr->c_size;
    sub_part = my_arr->c_size/ split;
    if(sub_part * split  < my_arr->c_size) sub_part++;

    index = initialise_split_array(split);

    for(i=0, j =0;i < my_arr->c_size && j< split; i += sub_part, j++)
    {
        (index+j)->index_ref  = (my_arr->arr + i);

        if(rem > sub_part)
            (index+j)->count = sub_part;
        else break;
        rem -= sub_part;
    }
    (index+j )->count = rem;
```

```
    return index;
}

/*void display(Split *new_arr)
{
    int i;
    for(i=0; i<)
}
*/
```

Main.c

```
#include<assert.h>
#include<stdlib.h>
#include<stdio.h>
#include "array.h"

int main()
{
    Array *test;
    Split *test_index;

    test = initialise_array(10);
    assert(test != NULL);

    assert(insert_data(test, 10));
    assert(insert_data(test, 20));
    assert(insert_data(test, 30));
    assert(insert_data(test, 40));
    assert(insert_data(test, 50));
    assert(insert_data(test, 60));
    assert(insert_data(test, 70));
    assert(insert_data(test, 80));


    assert(test->c_size  + 2 == test->t_size);

    test_index = split_array(test,3);

    printf("\n data = %d and count = %d", *(test_index->index_ref), test_index->count);
    printf("\n data = %d and count = %d", *(test_index+1)->index_ref, (test_index+1)-
>count);
      printf("\n data = %d and count = %d", *(test_index+2)->index_ref, (test_index+2)-
>count);


    return 0;
}
```

10. Application assignment

Header.h

```
struct _student_
{
    int stu_app_num;
    char stu_name[50];
    int stu_rank; //note that for NRI students and management students rank should be 1000
    char quota[15]; //Rank,Management,NRI
};
typedef struct _student_ Student;


struct _stu_app_
{
    Student *app;
    int c_Seat, t_Seat;
};

typedef struct _stu_app_ Seat;

Seat * initialise_student(int totalSeats);
int insert_data_student(Seat *my_app, Student data);
void sort_students_by_rank(Seat *my_app);
void generateAdmittedList(Seat *my_app);
```

Operation.c

```
#include <stdio.h>
#include <stdlib.h>
#include "application.h"

Seat * initialise_student(int totalSeats)
{
    Seat *my_app = malloc(sizeof(Seat));
    my_app->c_Seat = 0;
    my_app->t_Seat = totalSeats;
    my_app->app = malloc(sizeof(Student) * totalSeats);
    return my_app;
}

int insert_data_student(Seat *my_app, Student data)
{
    if(my_app->c_Seat == my_app->t_Seat)
        return 0;

    my_app->app[my_app->c_Seat] = data;
    my_app->c_Seat++;

    return 1;
}

void sort_students_by_rank(Seat *my_app)
{
```

```
    int i, j;
    for (i = 0; i<my_app->c_Seat-1; i++) {
        for (j = 0; j<my_app->c_Seat-i-1; j++) {
            if (my_app->app[j].stu_rank > my_app->app[j + 1].stu_rank) {
                // Swap the two students
                Student temp = my_app->app[j];
                my_app->app[j] = my_app->app[j + 1];
                my_app->app[j + 1] = temp;
            }
        }
    }
}

void generateAdmittedList(Seat *my_app)
{
    int merit_count = 0;
    int management_count = 0;
    int NRI_count = 0;
    printf("Admitted Student List:\n");

    for (int i = 0; i < my_app->t_Seat; i++) {
        if ((my_app->app[i].quota[0] == 'M' && management_count < 2) || (my_app-
>app[i].quota[0] == 'N' && NRI_count < 2)) { // M=Management and N=NRI quota
            printf("ID: %d\t Name: %s\t\t\t Quota: %s \n", my_app->app[i].stu_app_num,
my_app->app[i].stu_name, my_app->app[i].quota);
            if (my_app->app[i].quota[0] == 'M') {
                management_count++;
            } else {
                NRI_count++;
            }
        }
        else if (merit_count < 6){ // Merit
            printf("ID: %d\t Name: %s\t Rank: %d\t Quota: %s \n", my_app-
>app[i].stu_app_num, my_app->app[i].stu_name, my_app->app[i].stu_rank, my_app-
>app[i].quota);
            merit_count++;
        }
    }
}
```

Main.c

```
#include <string.h>
#include <stdlib.h>
#include<assert.h>
#include "application.h"

int main()
{
    Seat *admission;
    Student data;
```

```
    admission = initialise_student(16);

    //Enter student details
    //note that for NRI students and management students rank should be 1000
    data.stu_app_num = 23;
    strcpy(data.stu_name, "Niranjan");
    data.stu_rank = 90;
    strcpy(data.quota, "Rank");
    insert_data_student(admission,data);

    data.stu_app_num = 25;
    strcpy(data.stu_name, "Prasad");
    data.stu_rank = 18;
    strcpy(data.quota, "Rank");
    insert_data_student(admission,data);

    data.stu_app_num = 36;
    strcpy(data.stu_name, "Pramood");
    data.stu_rank = 427;
    strcpy(data.quota, "Rank");
    insert_data_student(admission,data);

    data.stu_app_num = 42;
    strcpy(data.stu_name, "Naveen");
    data.stu_rank = 612;
    strcpy(data.quota, "Rank");
    insert_data_student(admission,data);

    data.stu_app_num = 15;
    strcpy(data.stu_name, "Deepak");
    data.stu_rank = 35;
    strcpy(data.quota, "Rank");
    insert_data_student(admission,data);

    data.stu_app_num = 67;
    strcpy(data.stu_name, "Thushar");
    data.stu_rank = 40;
    strcpy(data.quota, "Rank");
    insert_data_student(admission,data);

    data.stu_app_num = 74;
    strcpy(data.stu_name, "Adharsh");
    data.stu_rank = 30;
    strcpy(data.quota, "Rank");
    insert_data_student(admission,data);

    data.stu_app_num = 82;
    strcpy(data.stu_name, "Shashank");
    data.stu_rank = 24;
    strcpy(data.quota, "Rank");
    insert_data_student(admission,data);

    data.stu_app_num = 90;
    strcpy(data.stu_name, "Rohith");
```

```
    data.stu_rank = 15;
    strcpy(data.quota, "Rank");
    insert_data_student(admission,data);

    data.stu_app_num = 105;
    strcpy(data.stu_name, "Akash");
    data.stu_rank = 1000;
    strcpy(data.quota, "Management");
    insert_data_student(admission,data);

    data.stu_app_num = 115;
    strcpy(data.stu_name, "Prathap");
    data.stu_rank = 1000;
    strcpy(data.quota, "NRI");
    insert_data_student(admission,data);

    data.stu_app_num = 162;
    strcpy(data.stu_name, "Darshan");
    data.stu_rank = 1000;
    strcpy(data.quota, "NRI");
    insert_data_student(admission,data);

    data.stu_app_num = 133;
    strcpy(data.stu_name, "Chinmai");
    data.stu_rank = 1000;
    strcpy(data.quota, "Management");
    insert_data_student(admission,data);

    data.stu_app_num = 144;
    strcpy(data.stu_name, "Reddy");
    data.stu_rank = 1000;
    strcpy(data.quota, "Management");
    insert_data_student(admission,data);

    data.stu_app_num = 165;
    strcpy(data.stu_name, "Sathvik");
    data.stu_rank = 1000;
    strcpy(data.quota, "Management");
    insert_data_student(admission,data);

    data.stu_app_num = 126;
    strcpy(data.stu_name, "Narendra");
    data.stu_rank = 1000;
    strcpy(data.quota, "NRI");
    insert_data_student(admission,data);

    sort_students_by_rank(admission);
    generateAdmittedList(admission);

    return 0;
}
```