## Chuẩn bị thi cuối kỳ DSA

### Thống kê

```
void solve(int a[], int n, int loop) {
    time_t t;
    srand((unsigned)time(&t));
    int fre1[10] = {};
    int fre2[10] = {};
    int total_fre1 = 0, total_fre2 = 0;
    for(int i = 0; i < loop; i++) {
        innitialArray(a, n, -100, 100);
        int cond1 = 0, cond2 = 0;
        f(a, n, &cond1, &cond2);
        fre1[cond1] += 1; fre2[cond2] += 1;
        total_fre1 += cond1; total_fre2 += cond2;
    }
    int k1 = sizeof fre1 / sizeof(int);
    int k2 = sizeof fre2 / sizeof(int);
    printf("\nK1:   | ");
    for(int i = 0; i < k1; i++) {
        printf("%2d ", i);
    }
    printf("\nFre:  | ");
    for(int i = 0; i < k1; i++) {
        printf("%2d ", fre1[i]);
    }

    printf("\nK2:   | ");
    for(int i = 0; i < k2; i++) {
        printf("%2d ", i);
    }
    printf("\nFre:  | ");
    for(int i = 0; i < k2; i++) {
        printf("%2d ", fre2[i]);
    }
    printf("\nAverage_k1 = %.2f | Average_k2 = %f",
1.0*total_fre1/loop, 1.0*total_fre2/loop);
```

```
    printf("\n-------------------------\n");
}
```

## ARRAY

1. SEARCH

```
int searchArray(int a[], int n, int value) {
    for(int i = 0; i < n; i++) {
        if(a[i] == value) // nếu tìm thấy a[i] bằng value hàm trả về giá trị i là chỉ số
            return i;
    }
    return -1; // không tìm thấy trả về -1
}
```

2. INSERT

```
void insertArray(int a[], int *n, int value, int index) {
    (*n)++; // chèn nên tăng độ dài mảng thêm 1

    for(int i = *n - 1; i > index; i--) {
        a[i] = a[i - 1]; // dời mảng qua phải 1 đơn vị tính từ idx + 1
    }
    a[index] = value;
}
```

3. DELETE

```
void deleteArray(int a[], int *n, int index) {
    for(int i = index; i < *n - 1; i++) {
        a[i] = a[i + 1];
    }
    a[*n - 1] = 0;
    (*n)--;
}
```

## ORDERED ARRAY

1. BINARY SEARCH

Trả về chỉ số của phần tử. Nếu không có trả về `n`

```
int binarySeach(int a[100], int value, int n) {
    int left = 0, right = n - 1;
    while(left <= right) {
        int mid = (left + right) / 2;
        if(a[mid] == value) {
            return mid;
        }else if(a[mid] > value) {
            left = mid + 1;
        }else right = mid - 1;
    }
    return n;
}
```

2. LOWER BOUND

Trả về chỉ số của phần từ đầu tiền không nhỏ hơn `value`. Nếu không có trả về `n`

```
int lower_bound(int arr[], int n, int x) {
    int l = 0, r = n;
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] < x) {
            l = mid + 1;
        } else {
            r = mid;
        }
    }
    return l;
}
```

3. UPER BOUND

Trả về chỉ số phần tử đầu tiên lớn hơn `value`. Nếu không có trả về `n`

```
int upper_bound(int arr[], int n, int x) {
    int l = 0, r = n;
    while (l < r) {
        int mid = l + (r - l) / 2;
        if (arr[mid] <= x) {
            l = mid + 1;
        } else {
            r = mid;
        }
    }
    return l;
}
```

**SORT**

1. BUBBLE SORT

```
void bubbleSort(float a[], int n) {
        for(int i = 0; i < n - 1; i++) {
                for(int j = 0; j < n - i - 1; j++) {
                        if(a[j] > a[j + 1]) {
                                int tmp = a[j];
                                a[j] = a[j + 1];
                                a[j + 1] = tmp;
                        }
                }
        }
}
```

2. SELECTION SORT

```
void selectionSort(int a[], int n) {
    for(int i = 0; i < n - 1; i++) {
        int min_idx = i;
        for(int j = i + 1; j < n; j++) {
            if(a[j] < a[min_idx])
                min_idx = j;
```

```
        }
        swap(&a[i], &a[min_idx]);
    }
}
```

3. INSERT SORT

```
void insertSort(int a[], int n) {
    int i, key, j;
    for(int i = 1; i < n; i++) {
        key = a[i];
        j = i - 1;
        while(j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j--;
        }
        a[j + 1] = key;
    }
}
```

4. QUICK SORT

```
int partition(int a[], int left, int right) {
    int pivot = right;
    right--;
    while(1) {
        while(a[left] <= a[pivot] && left <= right) left++;
        while(a[right] >= a[pivot] && right >= left) right--;
        if(left < right) {
            swap(&a[left], &a[right]);
        }else {
            break;
        }
    }
    swap(&a[left], &a[pivot]);
    return left;
}

void quick_sort(int a[],int left, int right) {
```

```
    if(right - left <= 0) return;
    int pivot = partition(a, left, right);
    quick_sort(a, left, pivot - 1);
    quick_sort(a, pivot + 1, right);
}
```

## STACK

1. CÀI BẰNG MẢNG

```
#define MAX 5

typedef struct
{
    int a[MAX];
    int top;
} Stack;

void init(Stack *s){
    s->top = -1;
}

int isEmpty(Stack* s) {
    if(s->top == -1) {
        return 1;
    }else return 0;
}
int isFull(Stack* s) {
    if(s->top == MAX - 1) {
        return 1;
    }
    return 0;
}
void push(Stack* s, int value) {
    s->a[++s->top] = value;
}
int pop(Stack* s) {
    int value = s->a[s->top];
    --s->top;
    return value;
```

```
}
void displayStack(Stack* s) {
    printf("\nStack: ");
    for(int i = 0; i <= s->top; i++) {
        printf("%3d ", s->a[i]);
    }
    printf("\n");
}
```

2. CÀI ĐẶT BẰNG DSLK

```
struct Node {
    int data;
    struct Node* next;
};
typedef struct Node Node;

Node* makeNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

struct Stack {
    Node* top;
};
typedef struct Stack Stack;

void init(Stack *st) {
    st->top = NULL;
}

void push(Stack *st, int data) {
    Node* newNode = makeNode(data);
    if(st->top == NULL) {
        st->top = newNode;
    }else {
        newNode->next = st->top;
        st->top = newNode;
```

```
    }
}

void pop(Stack *st) {
    if(st->top == NULL) return;
    Node* tmp = st->top;
    st->top = st->top->next;
    free(tmp);
}

int top(Stack *st) {
    if(st->top != NULL) {
        return st->top->data;
    }
}

int size(Stack st) {
    int cnt = 0;
    while(st.top != NULL) {
        ++cnt;
        st.top = st.top->next;
    }
    return cnt;
}

void displayStack(Stack* st){
    for (Node* node = st->top; node != NULL; node = node->next)
{
        printf("Node address: %p | ", &(node->data));
        printf("data = %d| ", node->data);
        printf("next node address = %p\n", node->next);
    }
    printf("\n");
}
```

## QUEUE

1. CÀI ĐẶT BẰNG MẢNG

```
```

## 2. CÀI ĐẶT BẰNG DSLK

```
typedef struct Nodetype{
    int data;
    struct Nodetype* next;
}Node;

Node* makeNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}

typedef struct Queuetype {
    Node* head;
    Node* tail;
}Queue;

void init(Queue *q) {
    q->head = NULL;
    q->tail = NULL;
}

int isEmpty(Queue *q) {
    if(q->head == NULL) return 1;
    return 0;
}

void push(Queue *q, int data) {
    Node* newNode = makeNode(data);
    if(q->head == NULL) {
        q->head = newNode;
        q->tail = q->head;
    }else {
        q->tail->next = newNode;
        q->tail = q->tail->next;
    }
}
```

```
void pop(Queue *q) {
    if(q->head == NULL) return;
    if(q->head == q->tail) {
            free(q->head);
            init(q);
    }else {
            Node* tmp = q->head;
            q->head = q->head->next;
            free(tmp);
    }
}

int size(Queue q) {
    int cnt = 0;
    while(q.head != NULL) {
        ++ cnt;
        q.head = q.head->next;
    }
    return cnt;
}

int front(Queue q) {
    return q.head->data;
}

void displayQueue(Queue* q){
    for (Node* node = q->head; node != NULL; node = node->next)
{
        printf("Node address: %p | ", &(node->data));
        printf("data = %d| ", node->data);
        printf("next node address = %p\n", node->next);
    }
    printf("\n");
}
```

**DSLK LINKED LIST**

Struct Linked List

```
struct LinkedList {
    Node* head;
};
typedef struct LinkedList LinkedList;

void init(LinkedList *list) {
        list->head = NULL;
}
```

Make Node

```
Node* makeNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

Insert First

```
void insertFirst(int data, LinkedList *list) {
    Node* newNode = makeNode(data);
    newNode->next = list->head;
    list->head = newNode;
}
```

Size

```
int size(LinkedList *list) {
    int size = 0;
    Node* tmp = list->head;
    while(tmp != NULL) {
        size += 1;
        tmp = tmp->next;
    }
    return size;
}
```

Insert Tail (BACK)

```
void insertTail(int data, LinkedList *list) {
    Node* newNode = makeNode(data);
    if(list->head == NULL) {
        list->head = newNode;
    }else {
        Node* tmp = list->head;
        while(tmp->next != NULL) {
            tmp = tmp->next;
        }
        tmp->next = newNode;
    }
}
```

Insert pos K

```
void insertK(int data, LinkedList *list, int pos) {
    int n = size(list);
    if(pos < 1 || pos > n + 1) return;
    if(pos == 1) {
        insertFirst(data, list);
    }else {
        Node* tmp = list->head;
        for(int i = 1;i <= pos - 2; i++) {
            tmp = tmp->next;
        }
        Node *newNode = makeNode(data);
        newNode->next = tmp->next;
        tmp->next = newNode;
    }
}
```

Delete First

```
void deleteFirst(LinkedList* list) {
    if(list->head == NULL)
        return;
```

```
    Node* tmp = list->head;
    list->head = list->head->next;
    free(tmp);
}
```

Delete Tail (BACK)

```
void deleteTail(LinkedList* list) {
    if(list->head == NULL)
        return;
    Node *tmp = list->head;
    if(tmp->next == NULL) {
        list->head = NULL;
        free(tmp);
        return;
    }
    while(tmp->next->next != NULL) {
        tmp = tmp->next;
    }
    Node *last = tmp->next; //node cuoi
    tmp->next = NULL;
    free(last);
}
```

Delete pos K

```
void deleteK(LinkedList *list, int pos) {
    int n = size(list->head);
    if(pos < 1 || pos > n) return; // ko hop le
    if(pos == 1) deleteFirst(list);
    else {
        Node* tmp = list->head;
        for(int i = 1; i <= pos - 2; i++) {
            tmp = tmp->next;
        }
        // tmp = pos - 1
        Node* kth = tmp->next; //node thu pos
        // cho pos - 1 ket noi voi node thu pos + 1
        tmp->next = kth->next;
```

```
        free(kth);
    }
}
```

PrintList

```
void printList(LinkedList *list) {
        for(Node* p = list->head; p != NULL; p = p->next) {
                printf("Node address: %p | Node data: %d | Next
Node address: %p\n", &p, p->data, p->next);
        }
        printf("\n");
}
```

## SORT DSLK LINKEDLIST

1. Selection sort DSLK LINKEDLIST

```
void selectionSortLinkedList(LinkedList list) {
    for(Node* p = list.head; p != NULL; p = p->next) {
        Node* minNode = p;
        for(Node* q = p->next; q != NULL; q = q->next) {
            if(q->data < minNode->data) {
                minNode = q;
            }
        }
        int tmp = p->data;
        p->data = minNode->data;
        minNode->data = tmp;
    }
}
```

2. Bubble sort DSLK LINKEDLIST

```
void bubbleSortLinkedList(LinkedList list) {
    for(Node* p = list.head; p->next != NULL; p = p->next) {
        for(Node* q = list.head; q->next != NULL && q != NULL;
q = q->next) {
```

```
            if(q->data > q->next->data) {
                int tmp = q->data;
                q->data = q->next->data;
                q->next->data = tmp;
            }
        }
    }
}
```

## TREE

Khởi tạo NODE

```
typedef struct TreeNode{
    int data;
    struct TreeNode* left; // luu dia chi node con ben trai
    struct TreeNode* right; // luu dia chi node con ben phai
}Node;
```

Mỗi Node có một Node con bên trái và Node con bên phải lưu địa chỉ

MakeNode function

```
TreeNode* makeNode(int data) {
    TreeNode* newNode = (TreeNode*)malloc(sizeof(TreeNode));
    newNode->data = data;
    newNode->left = NULL;
    newNode->right = NULL;
    return newNode;
}
```

Hàm insert kiểu VOID

```
void insertNode(int data, TreeNode* root) {
    if(!root) return;
    if(data < root->data) {
        // Nếu node bên trái là null thì tạo node
        if(root->left == NULL) {
```

```
            root->left = makeNode(data);
        }else {
        // Không thì tiếp tục tìm bên trái
            insertNode(data, root->left);
        }
    }else if(data > root->data) {
        // Nếu node con bên phải là null thì tạo node mới
        if(root->right == NULL) {
        // Không thì tiếp tục tìm bên phải
            root->right = makeNode(data);
        }else {
            insertNode(data, root->right);
        }
    }
}
```

Hàm insert kiểu đệ quy

```
TreeNode* insert(TreeNode* root, int data) {
    if(!root) {
        return makeNode(data);
    }
    if(data < root->data) {
        root->left = insert(root->left, data);

    }else if(data > root->data) {
        root->right = insert(root->right, data);
    }
    return root;
}
```

Hàm xóa kiểu con trỏ

```
/*
Thao tác xóa:
TH1: Node cần xóa không có con -> giải phóng node này
TH2: Node cần xóa có 1 con -> gán nó là con nó và giải phòng
    Node con
```

```
TH3: Node cần xó có đủ 2 con
    Bước 1: Tìm node con X nhỏ nhất lớn hơn Node cần xóa
    Bước 2: Gán giá trị Node cần xóa thành giá trị của node X
    Bước 3: Xóa node X ra khỏi cây
*/

// Ham minNode
// Ta chỉ cần đi sang phải và sau đó đi sang trái liên tục
// khi đến Node lá thì đó là Node nhỏ nhất lớn hơn Node cần xóa
TreeNode* minNode(TreeNode* root) {
    TreeNode* tmp = root;
    while(tmp != NULL && tmp->left != NULL) {
        tmp = tmp->left;
    }
    return tmp;
}

TreeNode* deleleNode(TreeNode* root, int key) {
    if(!root) return root;
    if(key < root->data) {
        root->left = deleleNode(root->left, key);
    }
    else if(key > root->data) {
        root->right = deleleNode(root->right, key);
    }
    else {
        // key == root->data
        // không có con trái
        if(root->left == NULL) {
            Node* tmp = root->right;
            free(root);
            return tmp;
        }
        // không có con phải
        else if(root->right == NULL) {
            Node* tmp = root->left;
            free(root);
            return tmp;
        }
        // Còn lại
        else {
```

```
            Node* tmp = minNode(root->right);
            root->data = tmp->data;
            root->right = deleleNode(root->right, tmp->data);
        }
    }
    return root;
}
```

Hàm xóa kiểu void

```
void deleleNode(TreeNode** node, int val) {
    if(!(*node)) return; // không tồn tại val trong cây
    if(val > (*node)->data) {
        deleleNode(&(*node)->right, val);
    }else if(val < (*node)->data) {
        deleleNode(&(*node)->left, val);
    }else {
        // Nếu node này k có lá
        if(!(*node)->left && !(*node)->right) {
            free(*node);
            *node = NULL;
        }
        // Nếu node này chỉ có lá phải
        else if(!(*node)->left) {
            TreeNode* tmp = (*node)->right;
            *node = tmp;
        }
        // Nếu node này chỉ có lá trái
        else if(!(*node)->right) {
            TreeNode* tmp = (*node)->left;
            *node = tmp;
        }
        // Nếu node này có cả 2 lá
        else {
            // ta đi tìm Node nhỏ nhất lớn hơn val
            TreeNode* tmp = minNode((*node)->right);
            // gán giá trị của tmp cho node và xóa tmp
            (*node)->data = tmp->data;
            deleleNode(&(*node)->right, tmp->data);
        }
```

```
    }
}
```

QUEUE TREE NODE

```
typedef struct NodeType {
    int data;
    struct NodeType* left, * right;
} TreeNode;

typedef struct queueNode {
    TreeNode *node;
    struct queueNode* next;
} qNode;

qNode* makeqNode(TreeNode* data) {
    qNode* newNode = (qNode*)malloc(sizeof(qNode));
    newNode->node = data;
    newNode->next = NULL;
    return newNode;
}

typedef struct queue {
    qNode* head;
    qNode* tail;
} queue;

void init(queue* q) {
    q->head = NULL;
    q->tail = NULL;
}

void push(queue* q, TreeNode* data) {
    qNode* newNode = makeqNode(data);
    if(q->head == NULL) {
        q->head = newNode;
        q->tail = q->head;
    }else {
        q->tail->next = newNode;
        q->tail = q->tail->next;
```

```
    }
}

void pop(queue* q) {
    if(q->head == q->tail) {
        free(q->head);
        init(q);
    }else {
        qNode* tmp = q->head;
        q->head = q->head->next;
        free(tmp);
    }
}

int isEmpty(queue* q) {
    if(q->head == NULL) return 1;
    else return 0;
}

TreeNode* front(queue* q) {
    return q->head->node;
}
```