

QOSSAY RIDA

Implementation for

Linked List

Cursor Linked List

Stack

Queues

BST

AVL Tree

صفحات من الذلة

Linked List:

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

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

```
typedef struct node* List;
```

```
typedef List Position;
```

```
List MakeEmpty(List L);
```

```
int IsEmpty(List L);
```

```
int IsLast(List P , List L);
```

```
List Find(int X , List L);
```

```
List FindPrevious(int X , List L);
```

```
void Delete(int X , List L);
```

```
void Insert(int X , List L , Position P);
```

```
void PrintList(List L);
```

```
void DeleteList(List L);
```

```
int size (List L);
```

```
int compare (List L1,List L2);
```

```
struct node {
```

```
    int Data;
```

```
    List Next ;
```

```
};
```

```

List MakeEmpty(List L){
    if(L != NULL)
        DeleteList( L );

    L = (List)malloc(sizeof(struct node));

    if(L == NULL)
        printf("Out of memory!\n");

    L->Next = NULL;
    return L;
}

int IsEmpty(List L){
    return L->Next == NULL;
}

int IsLast(List P , List L){
    return P->Next == NULL;
}

Position Find(int X , List L){

    Position P;
    P = L->Next;

    while(P != NULL && P->Data != X)
        P = P->Next;

    return P;
}

Position FindPrevious(int X , List L){

    Position P;
    P = L;
    while(P->Next != NULL && P->Next->Data != X)
        P = P->Next;

    return P;
}

```

```

void Delete(int X , List L){

    Position P, temp;
    P = FindPrevious(X, L);

    if( !IsLast(P, L) ){
        temp = P->Next;
        P->Next = temp->Next;
        free(temp);
    }
}

```

```

void Insert(int X , List L , Position P){

    Position temp;
    temp = (Position)malloc(sizeof(struct node));
    temp->Data = X;
    temp->Next = P->Next;
    P->Next = temp;

}

```

```

void PrintList(List L){

    Position P = L;

    if( IsEmpty(L))
        printf("Empty list\n");

    else
        do{
            P=P->Next;
            printf("%d\t", P->Data);
        }while( ! IsLast(P, L) );

    printf("\n");

}

```

```
void DeleteList(List L){
```

```
    Position P, temp;
```

```
    P = L->Next;
```

```
    L->Next = NULL;
```

```
    while(P != NULL){
```

```
        temp = P->Next;
```

```
        free(P);
```

```
        P=temp;
```

```
    }
```

```
}
```

```
int size (List L){
```

```
    Position ptr = L->Next;
```

```
    int count=0 ;
```

```
    while (ptr != NULL){
```

```
        count++;
```

```
        ptr=ptr->Next;
```

```
    }
```

```
    return count;
```

```
}
```

```
int compare (List L1 , List L2){
```

```
    List ptr = L2;
```

```
    while (L1->Next != NULL){
```

```
        L1=L1->Next;
```

```
        while(L2->Next != NULL){
```

```
            L2=L2->Next;
```

```
            if (L1->Data == L2->Data)
```

```
                return 0;
```

```
        }
```

```
        L2=ptr->Next;
```

```
    }
```

```
    return 1;
```

```
}
```

Cursor Linked List:

```
#include <stdio.h>

#include <stdlib.h>

#define spaceSize 11

typedef int List;

typedef int Position;

void InitializeCursorSpace();

List CursorList();

int CursorAlloc();

List MakeEmpty( List L );

int IsEmpty( List L );

int IsLast();

void Insert( int X, List L );

Position Find( int X, List L );

Position FindPrevious( int X, List L );

void Delete( int X, List L );

void CursorFree( Position P );

void DeleteList( List L );

void PrintList( List L );

struct node {
    int element;
    int next;
};

struct node cursorSpace[spaceSize];
```

```

void InitializeCursorSpace(){
    for(int i = 0; i<spaceSize-1; i++)
        cursorSpace[i].next=i+1;
    cursorSpace[spaceSize - 1].next = 0;
}

List CursorList(){
    List L = CursorAlloc();
    cursorSpace[ L ].next = 0;
    return L;
}

int CursorAlloc(){
    Position P;
    P = cursorSpace[ 0 ].next;
    cursorSpace[ 0 ].next = cursorSpace[ P ].next;

    if( P == 0 ){
        printf("Out of space!\n");
        return -1;
    }

    return P;
}

List MakeEmpty( List L ){
    if( L )
        DeleteList( L );

    L = CursorAlloc();

    if( L == 0 )
        printf("Out of memory!");

    cursorSpace[ L ].next = 0;
    return L;
}

```

```
int IsEmpty( List L ){
```

```
    return cursorSpace[ L ].next == 0;
```

```
}
```

```
int IsLast( Position P, List L ){
```

```
    return cursorSpace[ P ].next == 0;
```

```
}
```

```
void Insert( int X, List L ){
```

```
    Position TmpCell, P = L;
```

```
    TmpCell = CursorAlloc( );
```

```
    while( P && cursorSpace[P].next != 0)
```

```
        P = cursorSpace[P].next;
```

```
    cursorSpace[ TmpCell ].element = X;
```

```
    cursorSpace[ TmpCell ].next = cursorSpace[ P ].next;
```

```
    cursorSpace[ P ].next = TmpCell;
```

```
}
```

```
Position Find( int X, List L ){
```

```
    Position P;
```

```
    P = cursorSpace[ L ].next;
```

```
    while( P && cursorSpace[ P ].element != X )
```

```
        P = cursorSpace[ P ].next;
```

```
    return P;
```

```
}
```

```
Position FindPrevious( int X, List L ){
```

```
    Position P;
```

```
    P = L;
```

```
    while(cursorSpace[P].next &&
```

```
           cursorSpace[ cursorSpace[P].next ].element != X)
```

```
        P = cursorSpace[ P ].next;
```

```
    return P;
```

```
}
```

```
void Delete( int X, List L ){
```

```
    Position P, TmpCell;
```

```
    P = FindPrevious( X, L );
```

```
    if( !IsLast( P, L ) ){
```

```
        TmpCell = cursorSpace[ P ].next;
```

```
        cursorSpace[ P ].next = cursorSpace[ TmpCell ].next;
```

```
        CursorFree( TmpCell );
```

```
    }
```

```
}
```

```
void CursorFree( Position P ){
```

```
    cursorSpace[ P ].next = cursorSpace[ 0 ].next;
```

```
    cursorSpace[ 0 ].next = P;
```

```
}
```

```

void DeleteList( List L ){
    Position P, Tmp;
    P = cursorSpace[ L ].next;
    cursorSpace[ L ].next = 0;

    while( P != 0 ){
        Tmp = cursorSpace[ P ].next;
        CursorFree( P );
        P = Tmp;
    }
}

```

```

void PrintList( List L ){
    Position P = L;
    while( P != 0 ){
        printf("%d \t %d \t %d\n",
            P, cursorSpace[P].element, cursorSpace[P].next);
        P = cursorSpace[P].next;
    }
}

```

Stack (Linked List implementation):

```
#include <stdio.h>

#include <stdlib.h>

typedef struct node* PtrToNode;
typedef PtrToNode Stack;

int IsEmpty( Stack S );

Stack CreateStack();

void MakeEmpty( Stack S );

void Pop( Stack S );

int Top( Stack S );

void Push( int X, Stack S );

void DisposeStack( Stack S );

struct node{
    int Element;
    PtrToNode Next;
};

int IsEmpty( Stack S ){
    return S->Next == NULL;
}
```

```

Stack CreateStack(){
    Stack S= (Stack)malloc(sizeof(struct node));

    if( S == NULL )
        printf("Out of space!");

    S->Next = NULL;
    MakeEmpty( S );
    return S;
}

```

```

void MakeEmpty( Stack S ){
    if( S == NULL )
        printf("Out of space!");

    else
        while( !IsEmpty( S ))
            Pop(S);
}

```

```

void Pop( Stack S ){
    PtrToNode firstCell;

    if( IsEmpty( S ) )
        printf("Empty stack" );

    else{
        firstCell = S->Next;
        S->Next = S->Next->Next;
        free(firstCell);
    }
}

```

```
int Top( Stack S ){  
    if( !IsEmpty( S ) )  
        return S->Next->Element;  
  
    printf("Empty stack");  
    return 0;  
}
```

```
void Push( int X, Stack S ){  
  
    PtrToNode temp;  
    temp = ( Stack )malloc( sizeof( struct node ) );  
  
    if( temp == NULL )  
        printf("Out of space!");  
  
    else{  
        temp->Element = X;  
        temp->Next = S->Next;  
        S->Next = temp;  
    }  
}
```

```
void DisposeStack( Stack S ){  
  
    MakeEmpty( S );  
    free( S );  
}
```

Stack (Simple Array implementation):

```
#include <stdio.h>

#include <stdlib.h>

#define MAX_SIZE 11

void push(int value);
void pop();
int Top();
int IsEmpty();
void MakeEmpty();

int stack[MAX_SIZE];
int top = -1;

void push(int value) {
    if (top == MAX_SIZE - 1) {
        printf("Stack overflow\n");
        return;
    }

    top++;
    stack[top] = value;
}
```

```
void pop() {  
    if (top == -1)  
        printf("Stack underflow\n");  
  
    top--;  
}
```

```
int Top() {  
    if (top == -1) {  
        printf("Stack is empty\n");  
        return -1;  
    }  
  
    return stack[top];  
}
```

```
int IsEmpty() {  
    if (top == -1)  
        return 1;  
  
    return 0;  
}
```

```
void MakeEmpty() {  
    top = -1;  
}
```

Stack (Complex Array implementation):

```
#include <stdio.h>

#include <stdlib.h>

#define MINSTACKSIZE 5

typedef struct StackRecord* Stack;

Stack CreateStack(int size);

void Push(int x, Stack S);

int Top (Stack S);

void Pop(Stack S);

int IsEmpty(Stack S);

int IsFull(Stack S);

void MakeNull(Stack S);

void DisposeStack(Stack S);

struct StackRecord {

    int Top;

    int Capacity;

    int *Array;

};
```

```

Stack CreateStack(int size){
    Stack S;

    if( size < MINSTACKSIZE )
        printf("Stack size is too small");

    S = (Stack) malloc( sizeof( struct StackRecord ) );

    if( S == NULL )
        printf("Out of space");

    S->Array = (int*) malloc( sizeof(int) * size );

    if( S->Array == NULL )
        printf("Out of space!!!");

    S->Top = -1;
    S->Capacity = size ;
    return S ;
}

```

```

void Push(int x,Stack S){
    if(IsFull( S ))
        printf("Full stack");

    else
        S->Array[++S->Top] = x;
}

```

```

int Top (Stack S){
    if(IsEmpty(S))
        printf("Empty stack");

    else
        return S->Array[S->Top];

    return -1;
}

```

```
void Pop(Stack S){  
    if(IsEmpty(S))  
        printf("Empty stack");  
  
    else  
        S->Top--;  
}
```

```
int IsEmpty(Stack S){  
    return S->Top == -1 ;  
}
```

```
int IsFull(Stack S){  
    return S->Top == S->Capacity-1;  
}
```

```
void MakeNull(Stack S){  
    S->Top = -1;  
}
```

```
void DisposeStack(Stack S) {  
    if (S != NULL) {  
        free(S->Array);  
        free(S);  
    }  
}
```

Queues (Linked List implementation):

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

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

```
typedef struct Node* Node;
```

```
typedef struct Queue* Queue;
```

```
Node newNode(int data);
```

```
Queue createQueue();
```

```
int isEmpty(Queue q);
```

```
void enqueue(int x, Queue q);
```

```
int dequeue(Queue q);
```

```
void displayQueue(Queue q);
```

```
void freeQueue(Queue q);
```

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

```
struct Queue {  
    Node front;  
    Node rear;  
};
```

```
Node newNode(int data) {
```

```
    Node temp = (Node)malloc(sizeof(struct Node));  
    temp->data = data;  
    temp->next = NULL;  
    return temp;
```

```
}
```

```
Queue createQueue() {
```

```
    Queue q = (Queue)malloc(sizeof(struct Queue));  
    q->front = q->rear = NULL;  
    return q;
```

```
}
```

```
int isEmpty(Queue q) {
```

```
    return q->front == NULL;
```

```
}
```

```
void enqueue(int x, Queue q) {
```

```
    Node temp = newNode(x);  
    if (isEmpty(q)) {  
        q->front = q->rear = temp;  
        return;
```

```
    }
```

```
    q->rear->next = temp;  
    q->rear = temp;
```

```
}
```

```
int dequeue(Queue q) {  
    if (isEmpty(q)) {  
        printf("Queue underflow\n");  
        return -1;  
    }  
  
    Node temp = q->front;  
  
    q->front = q->front->next;  
  
    if (q->front == NULL)  
        q->rear = NULL;  
  
    int data = temp->data;  
    free(temp);  
    return data;  
}
```

```
void displayQueue(Queue q) {  
    if (isEmpty(q)) {  
        printf("Queue is empty\n");  
        return;  
    }  
  
    Node temp = q->front;  
  
    while (temp != NULL) {  
        printf("%d ", temp->data);  
        temp = temp->next;  
    }  
  
    printf("\n");  
}
```

```
void freeQueue(Queue q) {  
    Node temp = q->front;  
    while (temp != NULL) {  
        Node next = temp->next;  
        free(temp);  
        temp = next;  
    }  
    free(q);  
}
```

- It can be noticed that operations in Queues and Stacks give $O(1)$ running time (we are not traversing elements)
- Linked List implementation of Queue is quite straightforward:
 - Insert at the end of the list
 - Remove from beginning

Queues (Array implementation):

```
#include<stdio.h>

#include <stdlib.h>

#define MinQueueSize 5

typedef struct QueueRecord* Queue;

int IsEmpty( Queue Q );

int IsFull( Queue Q );

Queue CreateQueue(int MaxElements);

void MakeEmpty(Queue Q);

void DisposeQueue(Queue Q);

int Succ(int Value, Queue Q);

void Enqueue(int X, Queue Q);

int Front(Queue Q);

void Dequeue(Queue Q);

int FrontAndDequeue(Queue Q);

struct QueueRecord{

    int Capacity;
    int Front;
    int Rear;
    int Size;
    int *Array;

};
```

```
int IsEmpty(Queue Q){  
    return Q->Size == 0;  
}
```

```
int IsFull(Queue Q){  
    return Q->Size == Q->Capacity;  
}
```

```
Queue CreateQueue(int MaxElements){  
    Queue Q;  
    if( MaxElements < MinQueueSize )  
        printf("Queue size is too small\n");  
    Q = (Queue)malloc(sizeof( struct QueueRecord ));  
    if( Q == NULL )  
        printf("Out of space");  
    Q->Array = (int*)malloc(sizeof(int) * MaxElements);  
    if( Q->Array == NULL )  
        printf("Out of space");  
    Q->Capacity = MaxElements;  
    MakeEmpty(Q);  
    return Q;  
}
```

```
void MakeEmpty(Queue Q){  
    Q->Size = 0;  
    Q->Front = 1;  
    Q->Rear = 0;  
}
```

```
void DisposeQueue(Queue Q){
```

```
    if( Q != NULL ){
        free( Q->Array );
        free( Q );
    }
}
```

```
int Succ(int Value, Queue Q){
```

```
    if( ++Value == Q->Capacity )
        Value = 0;

    return Value;
}
```

```
void Enqueue(int X, Queue Q){
```

```
    if( IsFull( Q ) )
        printf("Full Queue");

    else{
        Q->Size++;
        Q->Rear = Succ(Q->Rear, Q);
        Q->Array[Q->Rear] = X;
    }
}
```

```
int Front(Queue Q){
```

```
    if(!IsEmpty(Q))
        return Q->Array[ Q->Front ];

    printf("Empty Queue!");
    return 0;
}
```

```
void Dequeue(Queue Q){  
    if(IsEmpty(Q))  
        printf("Empty Queue!");  
  
    else{  
        Q->Size--;  
        Q->Front = Succ( Q->Front, Q );  
    }  
}
```

```
int FrontAndDequeue(Queue Q){  
    int X = 0;  
  
    if(IsEmpty(Q))  
        printf("Empty Queue!");  
  
    else{  
        Q->Size--;  
        X = Q->Array[ Q->Front ];  
        Q->Front = Succ( Q->Front, Q );  
    }  
  
    return X;  
}
```

BST:

```
#include <stdio.h>

#include <stdlib.h>

typedef struct Node* TNode;

TNode MakeEmpty(TNode T);

TNode Find(int X, TNode T);

TNode FindMin(TNode T);

TNode FindMax(TNode T);

TNode Insert(int X, TNode T);

TNode Delete(int X, TNode T);

struct Node{

    int Element;
    TNode Left;
    TNode Right;

};

TNode MakeEmpty(TNode T){

    if(T != NULL){

        MakeEmpty(T->Left);
        MakeEmpty(T->Right);
        free(T);
    }

    return NULL;

}
```

```
TNode Find(int X, TNode T){  
    if( T == NULL)  
        return NULL;  
  
    else if( X < T->Element )  
        return Find( X, T->Left );  
  
    else if( X > T->Element )  
        return Find(X, T->Right);  
  
    else  
        return T;  
}
```

```
TNode FindMin(TNode T){  
    if(T == NULL)  
        return NULL;  
  
    else if(T -> Left == NULL)  
        return T;  
  
    else  
        return FindMin(T->Left );  
}
```

```
TNode FindMax(TNode T){  
    if(T == NULL)  
        return NULL;  
  
    else if(T -> Right == NULL)  
        return T;  
  
    else  
        return FindMax(T->Right);  
}
```

```

TNode Insert(int X, TNode T){
    if( T == NULL){
        T = (TNode)malloc( sizeof( struct Node ) );

        if( T == NULL)
            printf("Out of space");

        else{
            T->Element = X;
            T->Left = T->Right = NULL;
        }
    }

    else if( X < T->Element )
        T->Left = Insert( X, T->Left);

    else if( X > T->Element)
        T->Right = Insert( X, T->Right );

    return T;
}

```

```

TNode Delete(int X, TNode T){
    TNode TmpCell;

    if( T == NULL )
        printf( "Element not found" );

    else if( X < T->Element )
        T->Left = Delete( X, T->Left );

    else if( X > T->Element )
        T->Right = Delete( X, T->Right );

    else
        if( T->Left && T->Right ){
            TmpCell = FindMin( T->Right );
            T->Element = TmpCell->Element;
            T->Right = Delete( T->Element, T->Right );
        }

        else{
            TmpCell = T;

            if( T->Left == NULL )
                T = T->Right;

            else if( T->Right == NULL )
                T = T->Left;

            free(TmpCell);
        }

    return T;
}

```

AVL Tree:

```
#include <stdio.h>

#include <stdlib.h>

typedef struct avl_node* AVLNode;
typedef AVLNode AVLTree;

int max(int a, int b);
int height(AVLNode T);
AVLTree createAVLTree();
AVLNode Find(int X, AVLTree T);
int FindCount(int X, AVLTree T);
AVLNode singleRotateWithLeft(AVLNode K2);
AVLNode singleRotateWithRight(AVLNode K1);
AVLNode doubleRotateWithLeft(AVLNode K3);
AVLNode doubleRotateWithRight(AVLNode K1);
int getBalance(AVLNode T);
AVLTree Insert(int X, AVLTree T);
AVLNode findMin(AVLTree T);
AVLTree Delete(int X, AVLTree T);

struct avl_node{
    int element;
    int count;
    AVLNode left;
    AVLNode right;
    int height;
}
```

```
int max(int a, int b) {  
    return (a > b) ? a : b;  
}
```

```
int height(AVLNode T) {  
    if (T == NULL)  
        return -1;  
    else  
        return T->height;  
}
```

```
AVLTree createAVLTree() {  
    return NULL;  
}
```

```
AVLNode Find(int X, AVLTree T) {  
    if (T == NULL)  
        return NULL;  
    else if (X < T->element)  
        return Find(X, T->left);  
    else if (X > T->element)  
        return Find(X, T->right);  
    else  
        return T;  
}
```

```
int FindCount(int X, AVLTree T) {  
    AVLNode node = Find(X, T);  
    if (node == NULL)  
        return 0;  
    else  
        return node->count;  
}
```

```
AVLNode singleRotateWithLeft(AVLNode K2) {  
    AVLNode K1 = K2->left;  
    K2->left = K1->right;  
    K1->right = K2;  
    K2->height = max(height(K2->left), height(K2->right)) + 1;  
    K1->height = max(height(K1->left), K2->height) + 1;  
    return K1;  
}
```

```
AVLNode singleRotateWithRight(AVLNode K1) {  
    AVLNode K2 = K1->right;  
    K1->right = K2->left;  
    K2->left = K1;  
    K1->height = max(height(K1->left), height(K1->right)) + 1;  
    K2->height = max(height(K2->right), K1->height) + 1;  
    return K2;  
}
```

```
AVLNode doubleRotateWithLeft(AVLNode K3) {  
    K3->left = singleRotateWithRight(K3->left);  
    return singleRotateWithLeft(K3);  
}
```

```
AVLNode doubleRotateWithRight(AVLNode K1) {  
    K1->right = singleRotateWithLeft(K1->right);  
    return singleRotateWithRight(K1);  
}
```

```
int getBalance(AVLNode T) {  
    if (T == NULL)  
        return 0;  
    else  
        return height(T->left) - height(T->right);  
}
```

```
AVLNode findMin(AVLTree T) {  
    if (T == NULL)  
        return NULL;  
    else if (T->left == NULL)  
        return T;  
    else  
        return findMin(T->left);  
}
```

```
AVLTree Insert(int X, AVLTree T) {
```

```
    → if (T == NULL) {  
        T = (AVLTree)malloc(sizeof(struct avl_node));  
        if (T == NULL) {  
            printf("Out of memory!\n");  
            exit(1);  
        }  
        T->element = X;  
        T->count = 1;  
        T->height = 0;  
        T->left = T->right = NULL;  
    }
```

```
    → else if (X < T->element) {  
        T->left = Insert(X, T->left);  
        if (height(T->left) - height(T->right) == 2) {  
            if (X < T->left->element)  
                T = singleRotateWithLeft(T);  
            else  
                T = doubleRotateWithLeft(T);  
        }  
    }
```

```

    → else if (X > T->element) {
        T->right = Insert(X, T->right);
        if (height(T->right) - height(T->left) == 2) {
            if (X > T->right->element)
                T = singleRotateWithRight(T);
            else
                T = doubleRotateWithRight(T);
        }
    }

    → else
        T->count++;

        T->height = max(height(T->left), height(T->right)) + 1;
    return T;
}

```

```
AVLTree Delete(int X, AVLTree T) {
```

```
    AVLNode temp;
```

```
    ➡ if (T == NULL) {
```

```
        printf("Element not found\n");
```

```
        return T;
```

```
    }
```

```
    ➡ else if (X < T->element) {
```

```
        T->left = Delete(X, T->left);
```

```
        if (height(T->right) - height(T->left) == 2) {
```

```
            if (height(T->right->left) > height(T->right->right))
```

```
                T = doubleRotateWithRight(T);
```

```
            else
```

```
                T = singleRotateWithRight(T);
```

```
        }
```

```
    }
```

```
    ➡ else if (X > T->element) {
```

```
        T->right = Delete(X, T->right);
```

```
        if (height(T->left) - height(T->right) == 2) {
```

```
            if (height(T->left->right) > height(T->left->left))
```

```
                T = doubleRotateWithLeft(T);
```

```
            else
```

```
                T = singleRotateWithLeft(T);
```

```
        }
```

```
    }
```



else {

```
    if (T->count > 1)
        T->count--;
```

else {

```
        ★ if (T->left == NULL && T->right == NULL) {
            free(T);
            T = NULL;
        }
```

```
        ★ else if (T->left == NULL) {
            temp = T;
            T = T->right;
            free(temp);
        }
```

```
        ★ else if (T->right == NULL) {
            temp = T;
            T = T->left;
            free(temp);
        }
```

```
        ★ else {
            temp = findMin(T->right);
            T->element = temp->element;
            T->count = temp->count;
            T->right = Delete(T->element, T->right);
            if (height(T->left) - height(T->right) == 2) {
                if (height(T->left->right) > height(T->left->left))
                    T = doubleRotateWithLeft(T);
                else
                    T = singleRotateWithLeft(T);
            }
        }
```

}

}

}

```
    if (T != NULL)
        T->height = max(height(T->left), height(T->right)) + 1;
    return T;
}
```

الباقي بعدين زهقت