CONTENTS

LISTĂ SIMPLU ÎNLĂNȚUITĂ

DEFINIRE STRUCTURĂ LISTĂ SIMPLU ÎNLĂNȚUITĂ

```
struct Masina {
	char* nrInregistrare;
	int an;
	float pret;
	int nrLocuri; };
```

FUNCȚIE CREARE STRUCTURĂ LISTĂ SIMPLU ÎNLĂNȚUITĂ

```
Masina initMasina(const char* _nrInregistrare, int _an, float _pret, int _nrLocuri) {
	Masina m;
	m.nrInregistrare = (char*)malloc(sizeof(char)*(strlen(_nrInregistrare) + 1));
	strcpy(m.nrInregistrare, _nrInregistrare);
	m.an = _an;
	m.nrLocuri = _nrLocuri;
	m.pret = _pret;
	return m; }
```

FUNCȚIE AFIȘARE STRUCTURĂ LISTĂ SIMPLU ÎNLĂNȚUITĂ

```
void afisareMasina(Masina m) {
	printf("Numar: %s, an: %d, numar locuri: %d, pret: %5.2f\n", m.nrInregistrare, m.an, m.nrLocuri, m.pret);}
```

DEFINIRE NOD LISTĂ SIMPLU ÎNLĂNȚUITĂ

```
struct Nod {
	Masina info;
	Nod* next; };
```

FUNCȚIE CREARE NOD LISTĂ SIMPLU ÎNLĂNȚUITĂ

```
Nod* initNod(Masina m, Nod* _next) {
	Nod* nou = (Nod*)malloc(sizeof(Nod));
	nou->info = m;//shallow
	//nou->info = initMasina(m.nrInregistrare, m.an, m.pret, m.nrLocuri);//deep
	nou->next = _next;
	return nou; }
```

FUNCȚIE FIFO INSERARE LA FINAL DE LISTĂ SIMPLU ÎNLĂNȚUITĂ LINIARĂ

```
Nod* pushQueue(Nod* cap, Masina m) {
	Nod* nou = initNod(m, NULL);
	if (cap) {
		Nod*p = cap;
		while (p->next) {
			p = p->next; }
		p->next = nou; }
	else {
		cap = nou; }
	return cap; }
```

FUNCȚIE INSERARE LA ÎNCEPUT DE LISTĂ SIMPLU ÎNLĂNȚUITĂ LINIARĂ

```
node* insertAtTheBegin(ZOO zoo, node* head) {
	node* newNode = createNode(zoo, head);
	return newNode; }
```

FUNCTIE FIFO EXTRAGERE (DE LA INCEPUT)

```
Masina popQueue(Nod* &cap) {
	if (cap) {
		Masina m = cap->info;
		Nod* aux = cap;
		cap = cap->next;
		free(aux);
		return m; }
	else {
		return initMasina(NULL, 0, 0, 0); } }
```

FUNCTIE FIFO VERIFICARE LISTA GOALA

```
int isEmpty(Nod* cap) {
	return cap == NULL; }
```

FUNCȚIE FIFO AFIȘARE LISTĂ SIMPLU ÎNLĂNȚUITĂ LINIARĂ

```
void afisareLista(Nod* &cap) {
	if (cap) {
		while (cap) {
			afisareMasina(popQueue(cap)); } }
	else {
		printf("Lista este goala!"); } }
```

FUNCȚIE PARCURGERE CU CALCULARE AVG LISTĂ SIMPLU ÎNLĂNȚUITĂ LINIARĂ

```
float AvgAnimals(node* head) {
	if (head) {
		float avg = 0;
		int noNodes = 0;
		while (head) {
			avg = avg + head->value.noAnimals;
			head = head->next;
			noNodes++; }
		return avg / noNodes; }
	else
		return 0; }
```

FUNCȚIE ȘTERGERE ÎNTREAGA LISTĂ SIMPLU ÎNLĂNȚUITĂ LINIARĂ

```
void stergereLista(Nod* &cap) {
	while (cap) {
		Nod* temp = cap;
		cap = cap->next;
		free(temp->info.nrInregistrare);
		free(temp); } }
```

FUNCȚIE INSERARE CRESCĂTOARE IN LISTĂ SIMPLU ÎNLĂNȚUITĂ LINIARĂ

```
node* AscendingInsertion(ZOO zoo, node* head) {
	if (head) {
		node* temp = head;
		if (temp->value.noAnimals > zoo.noAnimals) {
			head = insertAtTheBegin(zoo, head);
			return head; }
		while (temp->next && temp->next->value.noAnimals < zoo.noAnimals) {
			temp = temp->next; }
		node* newNode = createNode(zoo, temp->next);
		temp->next = newNode;
		return head; }
	else {
		return createNode(zoo, NULL); } }
```

FUNCȚIE FILTRARE LISTĂ SIMPLU ÎNLĂNȚUITĂ

```
void filterList(Nod* &cap, int min, int max) {
	while (cap) {
		Masina m = popQueue(cap);
		if (m.nrLocuri >= min && m.nrLocuri <= max) {
			afisareMasina(m); } } }
```

FUNCȚIE TRANSFORMARE LISTĂ SIMPLU ÎNLĂNȚUITĂ IN VECTOR

```
Masina* toVector(Nod* &cap) {
	int lungimeLista = countLista(cap);
	Masina* vect = (Masina*)malloc(sizeof(Masina)*lungimeLista);
	int i = 0;
	if (cap) {
		while (cap) {
			Masina m = popQueue(cap);
			vect[i++] = m; }
		return vect; }
	else {
		return NULL; } }
```

FUNCȚII VECTOR

```
void afisareVector(Masina* vect, int lungimeVect) {
	for (int i = 0; i < lungimeVect; i++) {
		afisareMasina(vect[i]); } }

void sortareVector(Masina* &vect, int lungimeVector) {
	for (int i = 0; i < lungimeVector - 1; i++) {
		for (int j = i + 1; j < lungimeVector; j++) {
			if (vect[i].an > vect[j].an) {
				Masina aux = vect[i];
				vect[i] = vect[j];
				vect[j] = aux; } } } }

void valMasini(Masina* vect, int lungimeVect) {
	int anul = -1;
	float sum = 0;
	sortareVector(vect, lungimeVect);
	for (int i = 0; i < lungimeVect; i++) {
		if (vect[i].an == anul) {
			sum += vect[i].pret; }
		else {
			if (anul == -1) {
				anul = vect[i].an;
				sum += vect[i].pret; }
			else {
				printf("Anul: %d are valoarea totala de: %5.2f\n", anul, sum);
				anul = vect[i].an;
				sum = vect[i].pret; } } }
	printf("Anul: %d are valoarea totala de: %5.2f", anul, sum); }
```

DE INSERAT LA INCEPUTUL FISIERULUI

```
#define _CRTDBG_MAP_ALLOC
#include<stdlib.h>
#include<stdio.h>
#include<string.h>
#include<crtdbg.h>
```

EXEMPLU DE APELURI ÎN MAIN LISTĂ SIMPLU ÎNLĂNȚUITĂ LINIARĂ

```
void main() {
	_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
	Nod* cap = NULL;
	Masina m1 = initMasina("B-01-ERU", 2010, 4500.87, 2);
	Masina m2 = initMasina("CJ-121-BYY", 2017, 12500.00, 7);
	Masina m3 = initMasina("TR-01-PSS", 2010, 2500.00, 4);
	Masina m4 = initMasina("B-99-PSS", 2013, 14550.00, 12);
	cap = pushQueue(cap, m1);
	cap = pushQueue(cap, m2);
	cap = pushQueue(cap, m3);
	cap = pushQueue(cap, m4);
	printf("Lista 1\n");
	afisareLista(cap);
	printf("Lista 2\n");
	afisareLista(cap);
	cap = pushQueue(cap, m1);
	cap = pushQueue(cap, m2);
	cap = pushQueue(cap, m3);
	cap = pushQueue(cap, m4);
	int count = countLista(cap);
	printf("Numar masini: %d", count);
	printf("\n\nFiltru\n");
	filterList(cap, 1, 9);
	cap = pushQueue(cap, m1);
	cap = pushQueue(cap, m2);
	cap = pushQueue(cap, m3);
	cap = pushQueue(cap, m4);
	printf("\n\nVector\n");
	count = countLista(cap);
	Masina* vect = toVector(cap);
	afisareVector(vect, count);
	printf("\n\nCalcul valori anuale\n");
	valMasini(vect, count);
	printf("\n\nVector\n");
	afisareVector(vect, count);
	free(vect);
	free(m1.nrInregistrare);
	free(m2.nrInregistrare);
	free(m3.nrInregistrare);
	free(m4.nrInregistrare);
	stergereLista(cap); }
```

LISTĂ SIMPLU ÎNLĂNȚUITĂ CIRCULARĂ

DEFINIRE STRUCTURĂ LISTĂ SIMPLU ÎNLĂNȚUITĂ CIRCULARĂ

```
struct Muzeu {
	char* denumire;
	float pret;
	char esteDeschis; };
```

FUNCȚIE CREARE STRUCTURĂ LISTĂ SIMPLU ÎNLĂNȚUITĂ CIRCULARĂ

```
Muzeu creareMuzeu(const char* denumire, float pret, char deschis) {
	Muzeu m;
	m.denumire = (char*)malloc(sizeof(char)*(strlen(denumire) + 1));
	strcpy(m.denumire, denumire);
	m.pret = pret;
	m.esteDeschis = deschis;
	return m; }
```

FUNCȚIE AFIȘARE STRUCTURĂ LISTĂ SIMPLU ÎNLĂNȚUITĂ CIRCULARĂ

```
void afisareMuzeu(Muzeu m) {
	printf("Muzeul %s ", m.denumire);
	if (m.esteDeschis) {
		printf("este deschis, "); }
	else
		printf("NU este deschis, ");
	printf("iar pretul biletului este de %5.2f.\n", m.pret); }
```

DEFINIRE NOD LISTĂ SIMPLU ÎNLĂNȚUITĂ CIRCULARĂ

```
struct Nod {
	Muzeu info;
	Nod* next; };
```

FUNCȚIE INSERARE LA ÎNCEPUT DE LISTĂ SIMPLU ÎNLĂNȚUITĂ CIRCULARĂ

```
Nod* inserareInceputListaCirculara(Nod* cap, Muzeu m) {
	Nod* nou = (Nod*)malloc(sizeof(Nod));
	nou->info = m;//shallow copy
	if (cap) {
		nou->next = cap;
		Nod* temp = cap;
		while (temp->next != cap) {
			temp = temp->next; }
		temp->next = nou; }
	else {
		nou->next = nou; }
	return nou; }
```

FUNCȚIE PARCURGERE CU AFIȘARE LISTĂ SIMPLU ÎNLĂNȚUITĂ CIRCULARĂ

```
void afisareListaCirculara(Nod*cap) {
	Nod* temp = cap;
	while (temp && temp->next != cap) {
		afisareMuzeu(temp->info);
		temp = temp->next; }
	afisareMuzeu(temp->info); }
```

FUNCȚIE ȘTERGERE ÎNTREAGA LISTĂ SIMPLU ÎNLĂNȚUITĂ CIRCULARĂ

```
Nod* stergereListaCirculara(Nod* cap) {
	Nod* temp = cap;
	while (temp && temp->next != cap) {
		free(temp->info.denumire);
		Nod*aux = temp;
		temp = temp->next;
		free(aux); }
	free(temp);
	return NULL; }
```

EXEMPLU DE APELURI ÎN MAIN LISTĂ SIMPLU ÎNLĂNȚUITĂ CIRCULARĂ

```
void main() {
	Nod* cap = NULL;
	Muzeu m = creareMuzeu("Antipa", 20, 1);
	cap = inserareInceputListaCirculara(cap, m);

	cap = inserareInceputListaCirculara(cap, creareMuzeu("Luvru", 30, 1));
	cap = inserareInceputListaCirculara(cap, creareMuzeu("Geologie", 10, 0));
	cap = inserareInceputListaCirculara(cap, creareMuzeu("Aviatiei", 30, 1));

	afisareListaCirculara(cap);
	cap=stergereListaCirculara(cap);
	cap = inserareInceputListaCirculara(cap, creareMuzeu("ASE", 30, 1));
	afisareListaCirculara(cap); }
```

LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ (CINEMATOGRAF)

DEFINIRE STRUCTURĂ LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
struct Cinematograf {
	char* denumire;
	int nrLocuri;
	float pretBilet; };
```

FUNCȚIE CREARE STRUCTURĂ LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
Cinematograf initializareCinematograf(const char* _denumire, int _nrLocuri, float _pretBilet) {
	Cinematograf cinem;
	cinem.denumire = (char*)malloc(sizeof(char)*(strlen(_denumire) + 1));
	strcpy(cinem.denumire, _denumire);
	cinem.nrLocuri = _nrLocuri;
	cinem.pretBilet = _pretBilet;
	return cinem; }
```

FUNCȚIE AFIȘARE STRUCTURĂ LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
void afisareCinematograf(Cinematograf cinem) {
	printf("Cinematograful %s are %d locuri si un pret la bilet de %5.2f \n", cinem.denumire, cinem.nrLocuri, cinem.pretBilet); }
```

DEFINIRE NOD LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
struct Nod {
	Cinematograf informatie;
	Nod* urmator;
	Nod* anterior; };
```

FUNCȚIE CREARE NOD LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
Nod* initNod(Cinematograf cinem, Nod* _urmator, Nod* _anterior) {
	Nod* nodNou = (Nod*)malloc(sizeof(Nod));
	nodNou->informatie = cinem;
	nodNou->urmator = _urmator;
	nodNou->anterior = _anterior;
	return nodNou; }
```

DEFINIRE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
struct ListaDubluInlantuita {
	Nod* prim;
	Nod* ultim; };
```

FUNCȚIE PARCURGERE CU AFIȘARE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
void afisareLista(ListaDubluInlantuita lista) {
	if (lista.prim) {
		while (lista.prim) {
			afisareCinematograf(lista.prim->informatie);
			lista.prim = lista.prim->urmator;  } }
	else {
		printf("!!!Nu am ce afisa. Lista este goala!!!"); } }
```

FUNCȚIE INSERARE LA ÎNCEPUT DE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
ListaDubluInlantuita adaugareCinematograf(ListaDubluInlantuita lista, Cinematograf _cinematograf) {
	Nod* nodNou = initNod(_cinematograf, NULL, NULL);
	if (lista.prim) {
		nodNou->urmator = lista.prim;
		lista.prim->anterior = nodNou;
		lista.prim = nodNou; }
	else {
		lista.prim = nodNou;
		lista.ultim = nodNou; }
	return lista; }
```

FUNCȚIE PARCURGERE CU NUMĂRARE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
int numarareLista(ListaDubluInlantuita lista, int _nrLoc) {
	int nrElemente = 0;
	if (lista.prim) {
		while (lista.prim) {
			if (lista.prim->informatie.nrLocuri > _nrLoc) {
				nrElemente++; }
			lista.prim = lista.prim->urmator; } }
	else {
		printf("!!!Nu am ce numara. Lista este goala!!!"); }
	return nrElemente; }
```

FUNCȚIE EXTRAGERE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ FIFO (INSERARE LA ÎNCEPUT => EXTRAGERE LA FINAL)

```
void rulareFilm(ListaDubluInlantuita &lista) {
	if (lista.ultim) {
		printf("Detalii cinematograf ultima rulare:");
		afisareCinematograf(lista.ultim->informatie);
		Nod* aux = lista.ultim;
		lista.ultim = lista.ultim->anterior;
		lista.ultim->urmator = NULL;
		free(aux->informatie.denumire);
		free(aux); }
	else {
		printf("!!!Nu am ce rula. Lista este goala!!!"); } }
```

FUNCȚIE ȘTERGERE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
void stergereLista(ListaDubluInlantuita &lista) {
	while (lista.ultim) {
		Nod* aux = lista.ultim;
		lista.ultim = lista.ultim->anterior;
		free(aux->informatie.denumire);
		free(aux); }
	lista.ultim = NULL;
	lista.prim = NULL; }
```

FUNCȚIE PARCURGERE CU SUMARE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
void estimeazaImpact(ListaDubluInlantuita lista) {
	int totalLocuri = 0;
	float totalIncasari = 0.00;
	while (lista.prim) {
		totalLocuri += lista.prim->informatie.nrLocuri;
		totalIncasari += (lista.prim->informatie.pretBilet)*(lista.prim->informatie.nrLocuri);
		lista.prim = lista.prim->urmator; }
	printf("Filmul ar putea atrage maxim %d vizualizari si sa genereze maxim %5.2f lei incasati.", totalLocuri, totalIncasari); }
```

FUNCȚIE PARCURGERE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ - EXPORT ÎN VECTOR PENTRU O CONDIȚIE (FILTRARE); AFIȘARE VECTOR

```
void filtrare(ListaDubluInlantuita lista, Cinematograf vec[], int _nrLoc) {
	int i = 0;
	if (lista.prim) {
		while (lista.prim) {
			if (lista.prim->informatie.nrLocuri > _nrLoc) {
				vec[i] = lista.prim->informatie;
				i++; }
			lista.prim = lista.prim->urmator; } }
	else {
		printf("!!!Nu am ce filtra. Lista este goala!!!"); } }

void afisareVector(Cinematograf vec[], int nrElemente) {
	for (int i = 0; i < nrElemente; i++) {
		afisareCinematograf(vec[i]); } }
```

EXEMPLU DE APELURI ÎN MAIN LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
void main() {
	ListaDubluInlantuita lista;
	lista.prim = NULL;
	lista.ultim = NULL;
	lista = adaugareCinematograf(lista, initializareCinematograf("Corso", 220, 5.00));
	lista = adaugareCinematograf(lista, initializareCinematograf("Studio", 320, 60.00));
	lista = adaugareCinematograf(lista, initializareCinematograf("Patria", 120, 45.00));
	lista = adaugareCinematograf(lista, initializareCinematograf("Scala", 90, 40.00));
	lista = adaugareCinematograf(lista, initializareCinematograf("Favorit", 200, 35.50));
	afisareLista(lista);
	printf("\nESTIMARE RULARE\n");
	estimeazaImpact(lista);
	printf("\nAFISARE CINEMATOGRAF IN CARE A RULAT\n");
	rulareFilm(lista);
	printf("\nAFISARE LISTA DUPA O RULARE\n");
	afisareLista(lista);
	printf("\nESTIMARE DUPA O RULARE\n");
	estimeazaImpact(lista);
	printf("\n VECTOR \n");
	int nrLocuri;
	printf("Introduceti numarul minim de locuri in cinematograf pentru care doriti filtrarea: \n");
	scanf("%d", &nrLocuri);
	int nrEl = numarareLista(lista, nrLocuri);
	Cinematograf* vec = (Cinematograf*)malloc(sizeof(Cinematograf) * nrEl);
	filtrare(lista, vec, nrLocuri);
	printf("\n AFISARE LISTA FILTRATA CU CONDITIA DE FILTRARE: CINEMATOGRAFUL SA AIBA MAI MULT DE %d locuri\n\n",
nrLocuri);
	afisareVector(vec, nrEl);
	printf("\nAFISARE LISTA NEFILTRATA\n");
	afisareLista(lista);
	printf("\nSTERGERE LISTA\n");
	stergereLista(lista);
	printf("\nAFISARE LISTA STEARSA\n");
	afisareLista(lista);
	free(vec);
	vec = NULL; }
```

LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ (ANIMAL)

DEFINIRE STRUCTURĂ

```
struct Animal {
	float greutate;
	char* nume;
	int anNastere; };
```

FUNCȚIE CREARE STRUCTURĂ

```
Animal initAnimal(const char* nume, int an, float greutate) {
	Animal a;
	a.nume = (char*)malloc(sizeof(char)*(strlen(nume) + 1));
	strcpy(a.nume, nume);
	a.greutate = greutate;
	a.anNastere = an;
	return a;}
```

FUNCȚIE AFIȘARE STRUCTURĂ

```
void afisareAnimal(Animal a) {
	printf("%s are o greutate de %5.2f si este nascut in %d.\n", a.nume, a.greutate, a.anNastere); }
```

DEFINIRE NOD LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
struct Nod {
	Animal info;
	Nod* next;
	Nod* prev; };
```

FUNCȚIE CREARE NOD LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
Nod* initNod(Animal a, Nod* next, Nod* prev) {
	Nod* nou = (Nod*)malloc(sizeof(Nod));
	nou->info = a;
	nou->next = next;
	nou->prev = prev;
	return nou;}
```

DEFINIRE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
struct ListaDubluInlantuita {
	Nod* prim;
	Nod* ultim; };
```

FUNCȚIE INSERARE LA ÎNCEPUT DE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
ListaDubluInlantuita inserareInceput(ListaDubluInlantuita lista, Animal a) {
	Nod* nou = initNod(a, NULL, NULL);
	if (lista.prim) {
		nou->next = lista.prim;
		lista.prim->prev = nou;
		lista.prim = nou; }
	else {
		lista.prim = nou;
		lista.ultim = nou; }
	return lista; } }
```

FUNCȚIE INSERARE LA SFÂRȘIT DE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
ListaDubluInlantuita inserareSfarsit(ListaDubluInlantuita lista, Animal a) {
	Nod* nou = initNod(a, NULL, NULL);
	if (lista.ultim) {
		nou->prev = lista.ultim;
		lista.ultim->next = nou;
		lista.ultim = nou; }
	else {
		lista.prim = nou;
		lista.ultim = nou; }
	return lista; }
```

FUNCȚIE PARCURGERE CU AFIȘARE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
void printLista(ListaDubluInlantuita lista) {
	while (lista.prim) {
		afisareAnimal(lista.prim->info);
		lista.prim = lista.prim->next; } }
```

FUNCȚIE EXTRAGERE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
Animal extragereDinListaDupaNume(ListaDubluInlantuita &lista, const char* nume) {
	Nod* temp = lista.prim;
	while (temp && strcmp(temp->info.nume, nume) != 0) {
		temp = temp->next; }
	if (temp) {
		Animal rezultat = temp->info;
		if (temp->prev) {
			temp->prev->next = temp->next;
			if (temp->next) {
				temp->next->prev = temp->prev; }
			else{
				lista.ultim = temp->prev; }
			free(temp); }
		else {
			lista.prim = temp->next;
			lista.prim->prev = NULL;
			free(temp); }
		return rezultat; }
	else {
		return initAnimal("", 0, 0); } }
```

FUNCȚIE ȘTERGERE LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
void stergereLista(ListaDubluInlantuita &lista) {
	while (lista.ultim) {
		Nod* aux = lista.ultim;
		lista.ultim = lista.ultim->prev;
		free(aux->info.nume);
		free(aux); }
	lista.ultim = NULL;
	lista.prim = NULL; }
```

EXEMPLU DE APELURI ÎN MAIN LISTĂ DUBLU ÎNLĂNȚUITĂ LINIARĂ

```
void main() {
	ListaDubluInlantuita lista;
	lista.prim = NULL;
	lista.ultim = NULL;
	lista = inserareInceput(lista, initAnimal("Pisica", 2010, 6));
	lista = inserareInceput(lista, initAnimal("Caine", 2016, 8));
	lista = inserareInceput(lista, initAnimal("Hamster", 2017, 1));
	printLista(lista);
	lista = inserareSfarsit(lista, initAnimal("Papagal", 2016, 8));
	printLista(lista);
	printf("\n\n");
	afisareAnimal(extragereDinListaDupaNume(lista, "Caine"));
	printf("\n\n");
	printLista(lista);
	stergereLista(lista); }
```

LISTĂ DUBLU ÎNLĂNȚUITĂ CIRCULARĂ

ARBORI BINARI

FUNCTIE INSERARE IN ARBORE BINAR

```
nod* inserareInArbore(nod* rad, Cladire c) {
	if (rad != NULL) {
		if (c.numar < rad->inf.numar) {
			rad->st = inserareInArbore(rad->st, c);  }
		else {
			rad->dr = inserareInArbore(rad->dr, c); } }
	else {
		rad = creareNod(c, NULL, NULL); }
	return rad; }
```

FUNCTIE PARCURGERE ARBORE SRD (STANGA – RADACINA – DREAPTA)

```
void afisareSRD(nod* rad) {
	if (rad != NULL) {
		afisareSRD(rad->st);
		afisareCladire(rad->inf);
		afisareSRD(rad->dr); } }
```

FUNCTIE AFLARE INALTIME ARBORE

```
int inaltimeArbore(NOD* rad) {
	if (rad) {
		int hSt = inaltimeArbore(rad->st);
		int hDr = inaltimeArbore(rad->dr);
		return 1 + (hSt > hDr ? hSt : hDr); }
	else {
		return 0; } }
```

FUNCTIE STERGERE ARBORE

```
nod* stergereArbore(nod* rad) {
	if (rad) {
		stergereArbore(rad->st);
		stergereArbore(rad->dr);
		free(rad->inf.adresa);
		free(rad);
		return NULL; } }
```

FUNCTIE AFISARE NIVEL ARBORE

```
void afisareNivel(nod* rad, int nivelDorit, int nivelCurent) {
	if (rad) {
		if (nivelDorit == nivelCurent) {
			afisareCladire(rad->inf); }
		else {
			afisareNivel(rad->st, nivelDorit, nivelCurent + 1);
			afisareNivel(rad->dr, nivelDorit, nivelCurent + 1); } } }
```

FUNCTIE CAUTARE DUPA CRITERIU (ID)

```
Serial cautareSerialDupaID(NOD* rad, int id) {
	if (rad) {
		if (rad->info.id == id) {
			return rad->info;  }
		else {
			if (rad->info.id > id) {
				return cautareSerialDupaID(rad->st, id); }
			else {
				return cautareSerialDupaID(rad->dr, id); } } }
	else {
		Serial s;
		s.id = -1;
		s.numarActori = -1;
		s.actori = NULL;
		return s; } }
```

FUNCȚIE TRANSFORMARE DIN ARBORE ÎN VECTOR CU CONDIȚIE

```
int transformareDinArboreInVectorCuConditie(int a, Nod* rad, int arr[], const char* dataLiv) {
	int count = 0;
	if (rad) {
		count += transformareDinArboreInVectorCuConditie(a, rad->st, arr, dataLiv);
		if (strcmp(rad->info.dataLivrare, dataLiv) == 0) {
			arr[a + count++] = rad->info.Id; }
		count += transformareDinArboreInVectorCuConditie(a + count, rad->dr, arr, dataLiv); }
	return count; }
```

EXEMPLU APEL MAIN ARBORE

```
void main() {
	Cladire c = instantiazaCladire(5, "Bucuresti", 45.6);
	afisareCladire(c);
	//free(c.adresa);
	nod* rad = creareNod(c, NULL, NULL);
	rad = inserareInArbore(rad, instantiazaCladire(3, "Iasi", 34));
	rad = inserareInArbore(rad, instantiazaCladire(8, "Timisoara", 12));
	rad = inserareInArbore(rad, instantiazaCladire(6, "Craiova", 28));
	rad = inserareInArbore(rad, instantiazaCladire(7, "Brasov", 24));
	afisareSRD(rad);
	printf("\n%d\n", getInaltime(rad));
	afisareNivel(rad, 3, 1);
	rad = stergereArbore(rad);
	//free(c.adresa); }
```

ARBORI AVL

INITIALIZĂRI ARBORE AVL

```
#define _CRTDBG_MAP_ALLOC
#include<stdio.h>
#include<malloc.h>
#include<string>
#include<crtdbg.h>

struct Avion {
	char* model;
	int nrLocuri;					// vom folosi acest atribut drept cheie pt arbore
	int nrLocuriOcupate;
	float* preturiBilete; };
Avion initAvion(const char* model, int nrLocuri, int nrLocuriOcupate, float* preturiBilete) {
	Avion avion;
	avion.model = (char*)malloc(sizeof(char) * (strlen(model) + 1));
	strcpy_s(avion.model, strlen(model) + 1, model);
	avion.nrLocuri = nrLocuri;
	avion.nrLocuriOcupate = nrLocuriOcupate;
	avion.preturiBilete = (float*)malloc(sizeof(float) * avion.nrLocuriOcupate);
	for (int i = 0; i < avion.nrLocuriOcupate; i++) {
		avion.preturiBilete[i] = preturiBilete[i]; }
	return avion; }

void afisareAvion(Avion avion) {
	printf("Avionul %s are %d locuri dar au fost ocupate doar %d: ", avion.model, avion.nrLocuri, avion.nrLocuriOcupate);
	for (int i = 0; i < avion.nrLocuriOcupate; i++) {
		printf("%5.1f ", avion.preturiBilete[i]); }
	printf("\n"); }

Avion citesteAvionDinFisier(FILE* f) {
	Avion avion;
	char buffer[20];
	fscanf(f, "%s", buffer);
	avion.model = (char*)malloc(sizeof(char) * (strlen(buffer) + 1));
	strcpy_s(avion.model, strlen(buffer) + 1, buffer);
	fscanf(f, "%d", &avion.nrLocuri);
	fscanf(f, "%d", &avion.nrLocuriOcupate);
	avion.preturiBilete = (float*)malloc(sizeof(float) * avion.nrLocuriOcupate);
	for (int i = 0; i < avion.nrLocuriOcupate; i++) {
		fscanf(f, "%f", &(avion.preturiBilete[i])); }
	return avion; }

void stergereAvion(Avion avion) {
	free(avion.model);
	free(avion.preturiBilete);
	avion.model = NULL;
	avion.preturiBilete = NULL; }

struct NOD {
	Avion info;
	NOD* st;
	NOD* dr; };
```

FUNCȚIE ÎNĂLȚIME ARBORE AVL

```
int inaltimeArbore(NOD* rad) {
	if (rad) {
		int inaltimeST = inaltimeArbore(rad->st);
		int inaltimeDR = inaltimeArbore(rad->dr);
		return ( 1 + (inaltimeST > inaltimeDR ? inaltimeST : inaltimeDR) ); }
	else return 0; }
```

FUNCȚIE CALCUL GRAD ECHILIBRU ARBORE AVL

```
int calculareGradEchilibru(NOD* rad) {
	if (rad) {
		int inaltimeST = inaltimeArbore(rad->st);
		int inaltimeDR = inaltimeArbore(rad->dr);
		return inaltimeST - inaltimeDR; }
	else return 0; }
```

FUNCȚIE ROTIRE A DREAPTA ARBORE AVL

```
NOD* rotireLaDreapta(NOD* rad) {
	if (rad) {
		NOD* aux = rad->st;
		rad->st = aux->dr;
		aux->dr = rad;
		return aux; }
	return rad; }		//in cazul in care nu avem nimic, returneaza de fapt rad (NULL)
```

FUNCȚIE ROTIRE A STÂNGA ARBORE AVL

```
NOD* rotireLaStanga(NOD* rad) {
	if (rad) {
		NOD* aux = rad->dr;
		rad->dr = aux->st;
		aux->st = rad;
		return aux; }
	return rad; }
```

FUNCȚIE INSERARE ÎN ARBORE AVL

```
NOD* inserareAvionInArboreEchilibrat(Avion info, NOD* rad) {
	if (rad) {
		if (rad->info.nrLocuri > info.nrLocuri) {
			rad->st = inserareAvionInArboreEchilibrat(info, rad->st); }
		else {
			rad->dr = inserareAvionInArboreEchilibrat(info, rad->dr); }
		if (calculareGradEchilibru(rad) == 2) {
			if (calculareGradEchilibru(rad->st) != 1) {
				rad->st = rotireLaStanga(rad->st); }
			rad = rotireLaDreapta(rad); }
		if (calculareGradEchilibru(rad) == -2) {
			if (calculareGradEchilibru(rad->dr) != -1) {
				rad->dr = rotireLaDreapta(rad->dr); }
			rad = rotireLaStanga(rad); }
		return rad; }
	else {
		NOD* nodNou = (NOD*)malloc(sizeof(NOD));
		nodNou->info = info;
		nodNou->dr = nodNou->st = NULL;
		return nodNou; } }
```

FUNCȚIE AFIȘARE ARBORE AVL

```
void afisareArbore(NOD* rad)	{ //inordine
	if (rad) {
		afisareArbore(rad->st);
		afisareAvion(rad->info);
		afisareArbore(rad->dr); } }
```

FUNCȚIE CĂUTARE IN ARBORE AVL

```
Avion cautareAvionInArboreDupaNrLocuri(NOD* rad, int nrLocuri) {
	if (rad) {
		if (rad->info.nrLocuri == nrLocuri) {
			Avion a = initAvion(rad->info.model, rad->info.nrLocuri, rad->info.nrLocuriOcupate, rad->info.preturiBilete);
			return a;		}
		else if (rad->info.nrLocuri > nrLocuri) {
			return cautareAvionInArboreDupaNrLocuri(rad->st, nrLocuri); }
		else {
			return cautareAvionInArboreDupaNrLocuri(rad->dr, nrLocuri); } }
	else {
		return initAvion("", -1, -1, NULL); } }
```

FUNCȚIE ȘTERGERE ARBORE AVL

```
void stergereArbore(NOD* rad) { // in postordine
	if (rad) {
		stergereArbore(rad->st);
		stergereAvion(rad->info);
		stergereArbore(rad->dr); } }
```

EXEMPLU APEL MAIN PENTRU ARBORE AVL

```
int main() {
	_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
	//--arborele binar echilibrat
	//--deschidere fisier
	printf("--ARBORELE BINAR ECHILIBRAT (avioane incarcate din fisier)\n");
	NOD* rad = NULL;
	FILE* fisier = fopen("avioane.txt", "r");
	if (fisier) {
		int nrAV = 0;
		fscanf(fisier, "%d", &nrAV);
		for (int i = 0; i < nrAV; i++) {
			rad = inserareAvionInArboreEchilibrat(citesteAvionDinFisier(fisier), rad); } }
	fclose(fisier);
	afisareArbore(rad);		//afiseaza inordine (crescator)
	printf("\n-------Inaltimea arborelui este: %d", inaltimeArbore(rad));
	stergereArbore(rad); }
```

TABELE DE DISPERSIE HASH TABLES

STRUCTURĂ HASH TABLE

```
struct HashTable {
	CotaIntretinere* *vector;
	int dim; };
```

FUNCȚIE INIȚIALIZARE HASH TABLE

```
HashTable initHT(int dim) {
	HashTable h;
	h.vector = (CotaIntretinere**)malloc(sizeof(CotaIntretinere*)*dim);
	for (int i = 0; i < dim; i++) {
		h.vector[i] = NULL; }
	h.dim = dim;
	return h; }
```

FUNCȚIE CREARE HASH INDEX (HASH FUNCTION)

```
int hashFunction(CotaIntretinere c, HashTable h) {
	return c.nrApartament % h.dim; }
```

FUNCȚIE AFIȘARE HASH TABLE

```
void afisareHT(HashTable h) {
	for (int i = 0; i < h.dim; i++) {
		if (h.vector[i] != NULL) {
			afisareCota(*(h.vector[i])); } } }
```

FUNCȚIE INSERARE HASH TABLE

```
int inserareHT(HashTable h, CotaIntretinere c) {
	if (h.dim > 0) {
		int pozitie = hashFunction(c, h);
		if (h.vector[pozitie]) {
			int index = (pozitie + 1) % h.dim;
			while (h.vector[index] != NULL && index != pozitie) {
				index = (index + 1) % h.dim; }
			if (index != pozitie) {
				h.vector[index] = (CotaIntretinere*)malloc(sizeof(CotaIntretinere));
				*h.vector[index] = initCota(c.adresa, c.nrApartament, c.nrPersoaneApartament, c.anul, c.luna,
c.valoareIntretinere);
				return index; }
			else {
				return -1; //tabela plina } }
		else {
			h.vector[pozitie] = (CotaIntretinere*)malloc(sizeof(CotaIntretinere));
			*h.vector[pozitie] = initCota(c.adresa, c.nrApartament, c.nrPersoaneApartament, c.anul, c.luna,
c.valoareIntretinere);
			return pozitie; } }
	else {
		return -2; //tabela inexistenta } }
```

FUNCȚIE CALCUL SUMĂ DUPĂ CRITERIU

```
void afisareIntretinereAnuala(HashTable h, int _nrApartament, int _anul, Adresa _adresa) {
	float val = 0.00;
	if (h.dim > 0) {
		for (int i = 0; i < h.dim; i++)		{
			if (h.vector[i]->nrApartament == _nrApartament && h.vector[i]->anul == _anul && h.vector[i]->adresa.nr ==
_adresa.nr && strcmp(h.vector[i]->adresa.strada, _adresa.strada) == 0) {
				val += h.vector[i]->valoareIntretinere; } } }
	printf("\n\nValoare intretinere anuala %5.2f", val); }
```

FUNCȚIE NUMĂRARE ELEMENTE DUPĂ UN PRAG

```
void afisareNrCote(HashTable h, float prag) {
	int nr = 0;
	if (h.dim > 0) {
		for (int i = 0; i < h.dim; i++) {
			if (h.vector[i]->valoareIntretinere > prag) {
				nr++; } } }
	printf("\n\n%d cote de intretinere depasesc valoarea de %5.2f.", nr, prag); }
```

FUNCȚIE ȘTERGERE ELEMENTE DUPĂ CRITERII

```
void stergeCote(HashTable h, int _nrApartament, Adresa _adresa) {
	if (h.dim > 0) {
		for (int i = 0; i < h.dim; i++) {
			if (h.vector[i] && h.vector[i]->nrApartament == _nrApartament && h.vector[i]->adresa.nr == _adresa.nr
				&& strcmp(h.vector[i]->adresa.strada, _adresa.strada) == 0) {
				free(h.vector[i]->adresa.strada);
				h.vector[i]->adresa.strada=NULL;
				free(h.vector[i]);
				h.vector[i] = NULL; } } } }
```

EXEMPLU DE APEL ÎN MAIN HASH TABLE

```
void main() {
	_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);

	Adresa a1 = initAdresa("Magheru", 5);
	Adresa a2 = initAdresa("Unirii", 8);
	Adresa a3 = initAdresa("Fane Babanu", 22);
	HashTable ht = initHT(7);
	int r1 = inserareHT(ht, initCota(a1, 34, 2, 2014, 9, 350.55));
	int r2 = inserareHT(ht, initCota(a1, 34, 2, 2014, 10, 240.45));
	int r3 = inserareHT(ht, initCota(a1, 34, 2, 2015, 11, 500.00));
	int r4 = inserareHT(ht, initCota(a1, 38, 5, 2014, 11, 770.00));
	int r5 = inserareHT(ht, initCota(a2, 3, 1, 2014, 11, 100.01));
	int r6 = inserareHT(ht, initCota(a3, 25, 2, 2014, 11, 430.00));
	int r7 = inserareHT(ht, initCota(a3, 26, 3, 2014, 11, 380.00));
	afisareHT(ht);
	printf("\n\nRezultate inserari\n");
	printf("R1: %d\n", r1);
	printf("R2: %d\n", r2);
	printf("R3: %d\n", r3);
	printf("R4: %d\n", r4);
	printf("R5: %d\n", r5);
	printf("R6: %d\n", r6);
	printf("R7: %d\n", r7);
	afisareIntretinereAnuala(ht, 38, 2014, a1);
	afisareNrCote(ht, 240);
	stergeCote(ht, 34, a1);
	printf("\n\nSterg 34 a1\n");
	afisareHT(ht);
	stergeCote(ht, 38, a1);
	stergeCote(ht, 3, a2);
	stergeCote(ht, 25, a3);
	stergeCote(ht, 26, a3);
	printf("\n\nSterg tot\n");
	afisareHT(ht);
	free(a1.strada);
	free(a2.strada);
	free(a3.strada);
	free(ht.vector); }
```

HEAP

STRUCTURĂ HEAP

```
struct HEAP {
	Mesaj * vector;
	int dim; };
```

FUNCȚIE AFIȘARE HEAP

```
void printHEAP(HEAP heap) {
	for (int i = 0; i < heap.dim; i++) {
		afisareMesaj(heap.vector[i]); } }
```

FUNCȚIE FILTRARE HEAP (ADUCE IN NODUL ZERO PRIORITATEA MAXIMA)

```
void filter(HEAP heap, int position) {
	int pozLeft = 2 * position + 1;
	int pozRight = 2 * position + 2;
	int max = position;
	if (pozLeft < heap.dim && heap.vector[max].prioritate < heap.vector[pozLeft].prioritate) {
		max = pozLeft; }
	if (pozRight < heap.dim && heap.vector[max].prioritate < heap.vector[pozRight].prioritate) {
		max = pozRight; }
	if (max != position) {
		Mesaj aux = heap.vector[max];
		heap.vector[max] = heap.vector[position];
		heap.vector[position] = aux;
		if (2 * max + 1 < heap.dim) {
			filter(heap, max);  } } }
```

FUNCȚIE EXTRAGERE HEAP

```
Mesaj extrage(HEAP &heap) {
	if (heap.dim > 0) {
		Mesaj rezultat = heap.vector[0];
		Mesaj* temp = (Mesaj*)malloc(sizeof(Mesaj)*(heap.dim - 1));
		for (int i = 1; i < heap.dim; i++) {
			temp[i - 1] = heap.vector[i]; }
		heap.dim--;
		free(heap.vector);
		heap.vector = temp;
		for (int i = (heap.dim - 2) / 2; i >= 0; i--) {
			filter(heap, i); }
		return rezultat; }
	else {
		return initMesaj("", -1); } }
```

FUNCȚIE INSERARE HEAP

```
void inserare(HEAP &heap, Mesaj m) {
	Mesaj* temp = (Mesaj*)malloc(sizeof(Mesaj)*(heap.dim + 1));
	for (int i = 0; i < heap.dim; i++) {
		temp[i] = heap.vector[i]; }
	temp[heap.dim] = m;
	free(heap.vector);
	heap.vector = temp;
	heap.dim++;
	for (int i = (heap.dim - 2) / 2; i >= 0; i--) {
		filter(heap, i); } }
```

EXEMPLU DE APEL IN MAIN HEAP

```
void main()
{
	_CrtSetDbgFlag(_CRTDBG_ALLOC_MEM_DF | _CRTDBG_LEAK_CHECK_DF);
	HEAP heap;
	heap.dim = 6;
	heap.vector = (Mesaj*)malloc(sizeof(Mesaj)*heap.dim);
	heap.vector[0] = initMesaj("Mesaj", 2);
	heap.vector[1] = initMesaj("Hello", 6);
	heap.vector[2] = initMesaj("How are you?", 8);
	heap.vector[3] = initMesaj("I am fine", 4);
	heap.vector[4] = initMesaj("Thank you", 3);
	heap.vector[5] = initMesaj("But you", 9);
	printHEAP(heap);
	for (int i = (heap.dim - 2) / 2; i >= 0; i--) {
		filter(heap, i); }
	printf("\n\n");
	printHEAP(heap);
	printf("\n\n");
	Mesaj m = extrage(heap);
	afisareMesaj(m);
	free(m.text);
	inserare(heap, initMesaj("Element nou", 10));
	printf("\n\n");
	printHEAP(heap);
	printf("\n\n");
	m = extrage(heap);
	afisareMesaj(m);
	free(m.text); }
```