```
//////////////ARBORI AVL//////////////////
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

// 1. structura de date
struct AVL{
	int gradEchilibru;
	int infoUtil;
	AVL* left;
	AVL* right;
};

// 2. creare nod avl
AVL* creareNodAVL(int info){
	AVL* nod = (AVL*)malloc(sizeof(AVL));
	nod->gradEchilibru = 0;
	nod->infoUtil = info;
	nod->left = NULL;
	nod->right = NULL;
	return nod;
}

// 3.3 functie pentru calculul maximului dintre 2 numerel
int max(int a, int b){
	return (a > b) ? a : b;
}

// 3.1 calculam inaltimea arborelui
int inaltimeArbore(AVL* arbore){
	if (arbore){
		return 1 + max(inaltimeArbore(arbore->left), inaltimeArbore(arbore-
>right));
	}
	else return 0;
}
//3. calculam gradul de echilibru al unui nod
void calculGradEchilibru(AVL* nod){
	if (nod){
		nod->gradEchilibru = inaltimeArbore(nod->right) - inaltimeArbore(nod-
>left);
	}
}

// 4. rotiri
AVL* rotireSimplaDreapta(AVL* pivot, AVL* fiuStanga){
	pivot->left = fiuStanga->right;
	calculGradEchilibru(pivot);
	fiuStanga->right = pivot;
	calculGradEchilibru(fiuStanga);
	return fiuStanga;
}

AVL* rotireSimplaStanga(AVL* pivot, AVL* fiuDreapta){
	pivot->right = fiuDreapta->left;
	calculGradEchilibru(pivot);
	fiuDreapta->left = pivot;
```

```
	calculGradEchilibru(fiuDreapta);
	return fiuDreapta;
}

AVL* rotireDublaStangaDreapta(AVL* pivot, AVL* fiuStanga){
	pivot->left = rotireSimplaStanga(fiuStanga, fiuStanga->right);
	calculGradEchilibru(pivot);
	fiuStanga = pivot->left;
	fiuStanga = rotireSimplaDreapta(pivot, fiuStanga);
	calculGradEchilibru(fiuStanga);
	return fiuStanga;
}

AVL* rotireDublaDreaptaStanga(AVL* pivot, AVL* fiuDreapta){
	pivot->right = rotireDublaDreaptaStanga(fiuDreapta, fiuDreapta->left);
	calculGradEchilibru(pivot);
	fiuDreapta = pivot->right;
	fiuDreapta = rotireSimplaStanga(pivot, fiuDreapta);
	calculGradEchilibru(fiuDreapta);
	return fiuDreapta;
}

// FUNCTIE PENTRU ECHILIBRAREA ARBORELUI
void echilibrare(AVL*& arbore){
	// 2. echilibram arborele
	// 2.1 calculam gradul de echilibrare
	calculGradEchilibru(arbore);
	if (arbore->gradEchilibru == 2){
		if (arbore->right->gradEchilibru == -1)
			//dezechilibru stanga dreapta
			arbore = rotireDublaDreaptaStanga(arbore, arbore->right);
		else
			//dezechilibru stanga
			arbore = rotireSimplaStanga(arbore, arbore->right);
	}
	else if (arbore->gradEchilibru == -2){
		if (arbore->left->gradEchilibru == 1)
			arbore = rotireDublaStangaDreapta(arbore, arbore->left);
		else
			arbore = rotireSimplaDreapta(arbore, arbore->left);
	}
}

// 5. inserare in arbore
AVL* inserareNod(AVL* arbore, int infoNou){
	// 1. inseram nodul in arbore prin recursivitate
	if (arbore){
		if (arbore->infoUtil > infoNou)
			arbore->left = inserareNod(arbore->left, infoNou);
		else if (arbore->infoUtil < infoNou)
			arbore->right = inserareNod(arbore->right, infoNou);
		else
			printf("\nNodul exista in lista! Nu il poti insera din nou! \n");
	}
	else
		arbore = creareNodAVL(infoNou);

	echilibrare(arbore);
```

```
	return arbore;
}

// 6. parcurgere
void parcurgereInOrdine(AVL* arbore){
	if (arbore){
		parcurgereInOrdine(arbore->left);
		printf("Nod %d - echilibrare %d\n", arbore->infoUtil,arbore-
>gradEchilibru);
		parcurgereInOrdine(arbore->right);
	}
}

// 7. cautare in AVL
void cautare(AVL* arbore, int valCautata){
	if (arbore){
		if (arbore->infoUtil > valCautata)
			cautare(arbore->left, valCautata);
		else if (arbore->infoUtil < valCautata)
			cautare(arbore->right, valCautata);
		else
			printf("\n\n\nNodul catat este %d - echilibrare %d\n", arbore-
>infoUtil, arbore->gradEchilibru);
	}
}
// 8. stergere cu reechilibrare
//stergere pentru 2 descendenti
void stergereLogica(AVL*& radacina, AVL*& sleft)
{
	//caut cel mai mare nod din subarborele stang
	if (sleft->right)
		stergereLogica(radacina, sleft->right);
	else{
		radacina->infoUtil = sleft->infoUtil;
		AVL* nodTmp = sleft;
		sleft = sleft->left;
		free(sleft);
		free(nodTmp);
	}
}

//stergere
void stergereNodArbore(AVL*& arbore, int cheie)
{
	if (arbore)
	{
		if (arbore->infoUtil > cheie)
			stergereNodArbore(arbore->left, cheie);
		else
		if (arbore->infoUtil < cheie)
			stergereNodArbore(arbore->right, cheie);
		else
		{
			//1.este nod frunza=>nu are niciun descendent
			if (arbore->right == NULL && arbore->left == NULL)
			{//dezaloc memoria de jos in sus si apoi anunt ca nodul a devenit
null
				free(arbore);
```

```
                        arbore = NULL;
                }
                else
                        //2.sterg nod cu un descendent
                {
                        if (arbore->left != NULL && arbore->right == NULL)
                        {
                                //salvez noodul ce treb sters intr-un tmp
                                AVL* sters = arbore;
                                //construiesc leg pe stanga sa nu pierd copilul nodului
pe care vreau sa l sterg
                                arbore = arbore->left;
                                free(sters);
                        }
                        else if (arbore->left == NULL && arbore->right != NULL)
                        {
                                AVL* stergere = arbore;
                                arbore = arbore->right;
                                free(stergere);
                        }
                        else
                                //3.sterge nod cu doi descendenti
                        {
                                stergereLogica(arbore, arbore->left);
                        }
                }
        }
        echilibrare(arbore);
    }
}

// 9. citire din fisier
void citireFisier(AVL*& arbore){
    FILE* file = fopen("ArboreAVL.txt", "r");
    int valoare;
    if (file){
        while (!feof(file)){
            fscanf(file, "%d", &valoare);
            arbore = inserareNod(arbore, valoare);
        }
    }
    fclose(file);
}

void main(){
    AVL* arbore = NULL;
    citireFisier(arbore);
    parcurgereInOrdine(arbore);
    printf("\n-----------------------\n");
    cautare(arbore, 11);
    stergereNodArbore(arbore, 5);
    printf("\n-----------------------\n");
    parcurgereInOrdine(arbore);

}
```

```
/////////ARBORI BINARI DE CAUTARE/////////
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

//informatia utila
struct Medic{
	char* numeMedic;
	int idMedic;
};

//structura de date
struct ABC{
	Medic* infoUtil;
	ABC* right;
	ABC* left;
};

//initializare structura
ABC* creareNodArboreABC(Medic* medic){
	ABC* nod = (ABC*)malloc(sizeof(ABC));
	nod->infoUtil = medic;
	nod->left = NULL;
	nod->right = NULL;
	return nod;
}

//inserare in arbore
void inserareArbore(ABC*& arbore, ABC* nod){
	if (arbore == NULL)
		arbore = nod;
	else
	{
		if (strcmp(arbore->infoUtil->numeMedic, nod->infoUtil->numeMedic) > 0)
			inserareArbore(arbore->left, nod);
		else if (strcmp(arbore->infoUtil->numeMedic, nod->infoUtil->numeMedic) < 0)
			inserareArbore(arbore->right, nod);
		else
			 //printf("Cheia exista deja!"); //asta scriem in cazul in care nu
dorim sa avem cheie de mai multe ori in arbore
			// daca dorim sa avem cheie multipla trebuie sa alegem pe ce parte
sa o inseram (stanga sau dreapta) iar daca va urma o valoare mai mica/mare vom insera la
sfarsitul ultimei chei multiple
			// inserare pe partea drapta a cheii multiple
			inserareArbore(arbore->right, nod);
	}
}

//parcurgere in ordine SRD
void parcurgereInOrdine(ABC* arbore){
	if (arbore){
		parcurgereInOrdine(arbore->left);
		printf("%s-%d\n", arbore->infoUtil->numeMedic, arbore->infoUtil->idMedic);
		parcurgereInOrdine(arbore->right);
	}
}
```

```
//parcurgere in preordine RSD
void parcurgerePreOrdine(ABC* arbore){
	if (arbore){
		printf("%s-%d\n", arbore->infoUtil->numeMedic, arbore->infoUtil->idMedic);
		parcurgerePreOrdine(arbore->left);
		parcurgerePreOrdine(arbore->right);
	}
}

//parcurgere in postordine SDR
void parcurgerePostOrdine(ABC* arbore){
	if (arbore){
		parcurgerePostOrdine(arbore->left);
		parcurgerePostOrdine(arbore->right);
		printf("%s-%d\n", arbore->infoUtil->numeMedic, arbore->infoUtil->idMedic);
	}
}

//stergere pentru 2 descendenti
void stergereLogica(ABC*& radacina, ABC*& sleft)
{
	//caut cel mai mare nod din subarborele stang
	if (sleft->right)
		stergereLogica(radacina, sleft->right);
	else{
		Medic* tmp = radacina->infoUtil;
		radacina->infoUtil = sleft->infoUtil;
		ABC* nodTmp = sleft;
		sleft = sleft->left;
		free(tmp->numeMedic);
		free(tmp);
		free(sleft);
		free(nodTmp);
	}
}

//stergere
void stergereNodArbore(ABC*& arbore, char* cheie)
{
	if (arbore)
	{
		if (strcmp(arbore->infoUtil->numeMedic, cheie) > 0)
			stergereNodArbore(arbore->left, cheie);
		else
		if (strcmp(arbore->infoUtil->numeMedic, cheie) < 0)
			stergereNodArbore(arbore->right, cheie);
		else
		{
			//1.este nod frunza=>nu are niciun descendent
			if (arbore->right == NULL && arbore->left == NULL)
			{//dezaloc memoria de jos in sus si apoi anunt ca nodul a devenit
null
				free(arbore->infoUtil->numeMedic);
				free(arbore->infoUtil);
				free(arbore);
				arbore = NULL;
			}
			else
```

```
                        //2.sterg nod cu un descendent
                {
                        if (arbore->left != NULL && arbore->right == NULL)
                        {
                                //salvez noodul ce treb sters intr-un tmp
                                ABC* sters = arbore;
                                //construiesc leg pe stanga sa nu pierd copilul nodului
pe care vreau sa l sterg
                                arbore = arbore->left;
                                free(sters->infoUtil->numeMedic);
                                free(sters->infoUtil);
                                free(sters);
                        }
                        else if (arbore->left == NULL && arbore->right != NULL)
                        {
                                ABC* stergere = arbore;
                                arbore = arbore->right;
                                free(stergere->infoUtil->numeMedic);
                                free(stergere->infoUtil);
                                free(stergere);
                        }
                        else
                                //3.sterge nod cu doi descendenti
                        {
                                stergereLogica(arbore, arbore->left);
                        }
                }
        }
    }
}

//citire in fisier
void citireFisier(ABC*& arbore){
    FILE* file = fopen("arboreBinarDeCautare.txt", "r");
    if (file){
        while (!feof(file)){
            Medic* medic = (Medic*)malloc(sizeof(Medic));
            char buffer[100];
            fscanf(file, "%s", &buffer);
            medic->numeMedic = (char*)malloc(strlen(buffer) + 1);
            strcpy(medic->numeMedic, buffer);
            fscanf(file, "%d", &medic->idMedic);
            ABC* nod = creareNodArboreABC(medic);
            inserareArbore(arbore, nod);
        }
    }
    fclose(file);
}

void main(){
    ABC* arbore = nullptr;
    citireFisier(arbore);
    parcurgereInOrdine(arbore);
    printf("---------------\n");
    stergereNodArbore(arbore, "Dana");
    parcurgereInOrdine(arbore);
}
```

```
//////////////Structura HEAP - cozi de prioritate//////////
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

//informatia utila
struct Student{
	char* nume;
	int id;
};

//structura de date
struct Heap{
	Student* vectorStudenti;
	int nrElementeHeap; //nr de elemente existente in heap
	int memorieAlocata; //dimensiunea memoriei alocate (ca nr de elemente) pentru heap
};

//prelucrari
// 1. functie pentru interschimbarea a doua pozitii in cadrul vectorului heap
void interschimbarePozitii(Heap*& heap, int poz1, int poz2){
	Student tmp = heap->vectorStudenti[poz1];
	heap->vectorStudenti[poz1] = heap->vectorStudenti[poz2];
	heap->vectorStudenti[poz2] = tmp;
}

// 2. functie pentru filtrarea (reordonarea) heapului
void filtrare(Heap*& heap, int index){
	//consideram indexul maxim curent ca fiind cel transmis ca si parametru, adica cel
de la care plecam din arbore in vederea filtrarii
	int indexMax = index;
	//indexul fiului din stanga al nodului curent s-ar afla ca si pozitie in arbore pe
urmatorul nivel, inmultim cu 2 pentru ca fiecare nod parinte care cate 2 copii (+1 pt
stanga)
	int indexStanga = 2 * index + 1;
	//indexul fiului din dreapta al nodului curent s-ar afla ca si pozitie in arbore
pe urmatorul nivel, inmultim cu 2 pentru ca fiecare nod parinte are cate 2 copii (+2 pt
dreapta)
	int indexDreapta = 2 * index + 2;

	//verificam daca se respecta proprietatea de ordonare a arborelui binar ca si
concept
	if (indexStanga < heap->nrElementeHeap && heap->vectorStudenti[indexStanga].id >
heap->vectorStudenti[indexMax].id)
		indexMax = indexStanga;
	if (indexDreapta < heap->nrElementeHeap && heap->vectorStudenti[indexDreapta].id >
heap->vectorStudenti[indexMax].id)
		indexMax = indexDreapta;

	//daca varful actual nu respecta proprietatea de ordonare atunci coboram nodul in
arbore si reapelam recursiv procedura
	if (indexMax != index){
		interschimbarePozitii(heap, indexMax, index);
		filtrare(heap, indexMax);
	}
}
```

```
// 3. functie pentru construirea structurii heap
void construireHeap(Heap*& heap, Student vector[], int nrStudenti){
	heap = (Heap*)malloc(sizeof(Heap));
	heap->memorieAlocata = nrStudenti;
	heap->nrElementeHeap = nrStudenti;
	heap->vectorStudenti = (Student*)malloc(sizeof(Student)*nrStudenti);
	//copiez studentii in vectorul din heap
	for (int i = 0; i < nrStudenti; i++)
		heap->vectorStudenti[i] = vector[i];

	// Rearanjam elem a.i. sa satisfaca prop de ordonare folosind
	// metoda Filtrare pt coborarea elem in arbore (elementele din
	// a doua jumatate a masivului respecta implicit proprietatea
	// de heap deoarece reprezinta subarbori cu maxim un element)
	for (int i = (nrStudenti - 1) / 2; i >= 0; i--)
		filtrare(heap, i);
}

// 4.1 functie pentru cresterea memoriei heap in cazul in care aceasta este plica
(folosita la inserare)
void cresteMemoriaAlocata(Heap*& heap){
	// alocam un vector nou de dimeniune +1 fata de cea curenta
	Student* vectorNou = (Student*)malloc(sizeof(Student)*(heap->memorieAlocata + 1));
	for (int i = 0; i < heap->nrElementeHeap; i++)
		vectorNou[i] = heap->vectorStudenti[i];
	free(heap->vectorStudenti);
	heap->vectorStudenti = vectorNou;
	heap->memorieAlocata++;
}

// 4. functie pentru inserarea unui nou element in heap
void insereazaElement(Heap*& heap, Student element){
	//verificam daca mai avem memorie libera in heap, daca nu mai alocam spatiu
	if (heap->memorieAlocata == heap->nrElementeHeap)
		cresteMemoriaAlocata(heap);

	//expandam heap-ul
	heap->nrElementeHeap++;
	//adaugam noul element la sfarsitul heapului
	int index = heap->nrElementeHeap - 1;
	heap->vectorStudenti[index] = element;

	//ii calculam indexul parintelui si il urcam in arbore atat cat este nevoie pentru
a pastra proprietatea de structura
	int indexParinte = (index - 1) / 2;
	while (indexParinte >= 0 && heap->vectorStudenti[index].id > heap-
>vectorStudenti[indexParinte].id){
		interschimbarePozitii(heap, index, indexParinte);
		index = indexParinte;
		indexParinte = (index - 1) / 2;
	}
}

// 5. functie pentru extragerea elementului maxim din heap
Student extragereMaxim(Heap*& heap){
	// pentru ca heap-ul are elemente de la 0 la nrElemente-1 facem un -- a.i. sa
raman fix cu pozitia dorita
	heap->nrElementeHeap--;
```

```
	//Student elementDeReturnat = heap->vectorStudenti[heap->nrElementeHeap];
	interschimbarePozitii(heap, 0, heap->nrElementeHeap);
	filtrare(heap, 0);
	return heap->vectorStudenti[heap->nrElementeHeap];
}

// functie pentru citirea din fisier a vetcorului de studenti
void citireFisier(Heap*& heap){
	FILE* file = fopen("StructuraHEAP.txt", "r");
	int nrStudenti;
	Student vectorStudenti[100];
	int i = 0;
	if (file){
		fscanf(file, "%d", &nrStudenti);
		while (!feof(file)){
			Student stud;
			char buffer[50];
			fscanf(file, "%s", &buffer);
			stud.nume = (char*)malloc(strlen(buffer) + 1);
			strcpy(stud.nume, buffer);
			fscanf(file, "%d", &stud.id);
			vectorStudenti[i] = stud;
			i++;
		}
		construireHeap(heap, vectorStudenti, nrStudenti);
	}
	fclose(file);
}

void afisareImproprieINCORECTA(Heap* heap){
	for (int i = 0; i < heap->nrElementeHeap; i++){
		printf("\n %d - %s", heap->vectorStudenti[i].id, heap-
>vectorStudenti[i].nume);
	}
}

void main(){
	Heap* heap = NULL;
	citireFisier(heap);
	afisareImproprieINCORECTA(heap);
	printf("\n----------------------------------\n");
	Student stud = extragereMaxim(heap);
	printf("Maximul este: %d - %s ", stud.id, stud.nume);
	afisareImproprieINCORECTA(heap);
}
```

```
////////GRAFURI - LISTA DE ADIACENTA (-lista de liste-)/////
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

//structuri de date
struct NodSecundar{
	int infoUtil;
	NodSecundar* nextSecundar;
};

struct NodPrincipal{
	int infoUtil;
	NodPrincipal* nextPrincipal;
	NodSecundar* listaNoduri;
};

//prelucrari pe principala
NodPrincipal* creareNodPrincipal(int info){
	NodPrincipal* nod = (NodPrincipal*)malloc(sizeof(NodPrincipal));
	nod->infoUtil = info;
	nod->nextPrincipal = NULL;
	nod->listaNoduri = NULL;
	return nod;
}

NodPrincipal* inserareListaPrincipala(NodPrincipal*& lista, int nodInfo){
	NodPrincipal* nod = creareNodPrincipal(nodInfo);
	NodPrincipal* tmp = lista;
	if (lista == NULL)
		lista = nod;
	else{
		while (tmp->nextPrincipal)
			tmp = tmp->nextPrincipal;
		tmp->nextPrincipal = nod;
	}
	return nod;
}

//prelucrari pe secundara
NodSecundar* creareNodSecundar(int info){
	NodSecundar* nod = (NodSecundar*)malloc(sizeof(NodSecundar));
	nod->infoUtil = info;
	nod->nextSecundar = NULL;
	return nod;
}

void inserareListaSecundara(NodSecundar** lista, int nodInfo){
	NodSecundar* nod = creareNodSecundar(nodInfo);
	nod->nextSecundar = *lista;
	*lista = nod;
}

//citirea din fisier a grafului
void citireFisier(NodPrincipal*& graf){
	FILE* file = fopen("Grafuri-ListaDeAdiacenta.txt", "r");
	int varf, nrVarfuri;
```

```
	if (file){
		while (!feof(file)){
			//citim informatia pentru nodul principal si il cream
			fscanf(file, "%d", &varf);
			NodPrincipal* tmp = inserareListaPrincipala(graf, varf);

			//citim numarul de noduri adiacente
			fscanf(file, "%d", &nrVarfuri);

			//citim toate nodurile secundare
			for (int i = 0; i < nrVarfuri; i++){
				fscanf(file, "%d", &varf);
				NodSecundar* lista = tmp->listaNoduri;
				inserareListaSecundara(&lista, varf);
				tmp->listaNoduri = lista;
			}
		}
	}
	fclose(file);
}

//o parcurgere oarecare de lista ->graf
void parcurgereSimpla (NodPrincipal* graf){
	while (graf){
		printf("Nodul: %d \n", graf->infoUtil);
		NodSecundar* noduri = graf->listaNoduri;
		while (noduri){
			printf("%d \t", noduri->infoUtil);
			noduri = noduri->nextSecundar;
		}
		printf("\n");
		graf = graf->nextPrincipal;
	}
}

void main(){
	NodPrincipal* graf = NULL;
	citireFisier(graf);
	parcurgereSimpla(graf);

}
```

```
//////////////GRAFURI - MATRICE DE ADIACENTA//////////////
#include "stdio.h"
#include "stdlib.h"
#include "string.h"
#include "memory.h"

//algoritm de parcurgere in latime
void parcurgereBF(int** matrice, int nrVarfuri, int nodStart){
	int* vectorCoada = NULL;
	int* vizitat = NULL;
	int pIn = -1;
	int pOut = 0;

	vectorCoada = (int*)malloc(sizeof(int)*nrVarfuri);
	vizitat = (int*)malloc(sizeof(int)*nrVarfuri);
	memset(vizitat, 0, sizeof(int)*nrVarfuri);

	//marcam nodul de start ca fiind vizitat
	vizitat[nodStart] = 1;
	//si il inseram in coada
	vectorCoada[++pIn] = nodStart;

	//cat timp coada mai are elemente
	while (pIn >= pOut){
		for (int i = 0; i < nrVarfuri; i++){
			nodStart = vectorCoada[pOut];
			if (matrice[nodStart][i] == 1 && vizitat[i] == 0){
				//marcare nod ca fiind vizitat
				vizitat[i] = 1;
				//inserare vecin nevizitat in coada
				vectorCoada[++pIn] = i;
			}
		}
		printf("%3d", ++vectorCoada[pOut++]);
	}
}

void citireMatriceAdiacenta(int**& matriceAdiacenta){
	FILE* file = fopen("Grafuri-MatriceDeAdiacenta.txt", "r");
	int nrArce = 0, nrVarfuri = 0;
	if (file){
		fscanf(file, "%d", &nrArce);
		fscanf(file, "%d", &nrVarfuri);

		//aloc spatiu pentru matrice
		matriceAdiacenta = (int**)malloc(sizeof(int*)*nrVarfuri);
		memset(matriceAdiacenta, 0, (sizeof(int*)*nrVarfuri));
		for (int i = 0; i < nrVarfuri; i++){
			matriceAdiacenta[i] = (int*)malloc(sizeof(int));
			memset(matriceAdiacenta[i], 0, sizeof(int)*nrVarfuri);
		}

		//citim arcele si construim matricea de adiacenta
		int sursa, destinatie;
		for (int i = 0; i < nrArce; i++){
			fscanf(file, "%d %d", &sursa, &destinatie);
```

```
                    matriceAdiacenta[sursa - 1][destinatie - 1] =
matriceAdiacenta[destinatie - 1][sursa - 1] = 1;
             }

             //afisare matrice de adiacenta
             for (int i = 0; i<nrVarfuri; i++){
                    for (int j = 0; j<nrVarfuri; j++){
                           printf("%3d", matriceAdiacenta[i][j]);
                    }
                    printf("\n");
             }

             for (int i = 0; i<nrVarfuri; i++){
                    printf("Parcurgere grad fin varful %d: \n", i + 1);
                    parcurgereBF(matriceAdiacenta, nrVarfuri, i);
                    printf("\n--------------------------------\n");
             }
       }
       fclose(file);
}

void main(){
       int** matrice = NULL;
       citireMatriceAdiacenta(matrice);
}
```

```
//////////////////COADA/////////////////
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

//informatia utila
enum tipAngajat{PROF, MED};

union tipOreLucrate{
	double nrOreGarda;
	int nrOreNorma;
};

struct  Angajat{
	char* numeAngajat;
	tipAngajat profesie;
	tipOreLucrate oreLucrate;
};

//structura de date
struct Nod{
	Angajat* infoUtil;
	Nod* next;
};

struct Coada{ //folosesc aceasta structura pentru ca retin inceputul si sfarsitul
intregii liste ci nu al fiecarui nod (deci nu pot pune prim si ultim in structura Nod)
	Nod* prim;
	Nod* ultim;
};

//initializare
void initCoada(Coada*& coada){
	coada->prim = coada->ultim = NULL;
}

//creare nod
Nod* creareNod(Angajat* info){
	Nod* nou = (Nod*)malloc(sizeof(Nod));
	nou->infoUtil = info;
	nou->next = NULL;
	return nou;
}

//functie empty
bool emptyCoada(Coada* coada){
	if (coada->prim == NULL)
		return true;
	else
		return false;
}

//adaugare in coada
void inserareCoada(Coada*& coada, Nod* nou){
	//!!!coada nu se parcurge pentru ca avem deja o referinta la ultimul nod
	if (emptyCoada(coada))
		coada->prim = coada->ultim = nou;
```

```
	else{
		coada->ultim->next = nou;
		coada->ultim = nou;
	}
}

//parcurgere = extragere din coada pentru ca este coada
Nod* extragereNodCoada(Coada** coada){
	Nod* extrage = (*coada)->prim;
	(*coada)->prim = extrage->next;
	return extrage;
}

//stergerea primului nod care contine cheia nume
void stergereDupaNume(Coada*& coada, char* nume){
	//!!!!!se face parcurgere si in coada nu se parcurge niciodata!!!!!!!!!
	if (!emptyCoada(coada)){
		//verific mai intai prima pozitie
		if (strcmp(coada->prim->infoUtil->numeAngajat, nume) == 0){
			Nod* sterge = coada->prim;
			coada->prim = sterge->next;
			free(sterge);
		}

		Nod* lista = coada->prim;
		while (lista->next != NULL && (strcmp(lista->next->infoUtil->numeAngajat,
nume) != 0))
			lista = lista->next;

		if (lista->next){
			Nod* sterge = lista->next;
			lista->next = sterge->next;
			free(sterge);
		}
	}
}
void citireFisier(Coada*& coada){
	FILE* file = fopen("Angajati.txt", "r");
	if (file){
		while (!feof(file)){
			Angajat* ang = (Angajat*)malloc(sizeof(Angajat));

			char buffer[100];
			fscanf(file, "%s", &buffer);
			ang->numeAngajat = (char*)malloc(strlen(buffer) + 1);
			strcpy(ang->numeAngajat, buffer);

			int tipAng;
			fscanf(file, "%d", &tipAng);
			switch (tipAng)
			{
			case 0:
				ang->profesie = PROF;
				fscanf(file, "%d", &ang->oreLucrate.nrOreNorma);
				break;
			case 1:
				ang->profesie = MED;
				fscanf(file, "%lf", &ang->oreLucrate.nrOreGarda);
```

```
                        break;
                    }
                    Nod* nod = creareNod(ang);
                    inserareCoada(coada, nod);
                }
        }
        fclose(file);
}

void main(){
        Coada* coada = (Coada*)malloc(sizeof(Coada));
        initCoada(coada);
        citireFisier(coada);

        stergereDupaNume(coada, "Costel");
        while (!emptyCoada(coada)){
                Nod* tmp = extragereNodCoada(&coada);
                printf("%s\t", tmp->infoUtil->numeAngajat);
        }
}
```

```
//////////LISTA DUBLA//////////
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

//informatia utila
enum tipAero{PAS, CAR};

union tipIncarcatura{
	short int nrLocuri;
	double greutateMaxima;
};

//structura de date
struct FAV{
	char* idAeronava;
	tipAero aeronava;
	tipIncarcatura incarcatura;
};

struct Nod{  //next si prev se pune pentru fiecare nod!!!
	FAV* infoUtil;
	Nod* next;
	Nod* prev;
};

struct ListaDubla{
	Nod* prim;
	Nod* ultim;
};

//creare nod
Nod* creareNod(FAV* fav){
	Nod* nou = (Nod*)malloc(sizeof(Nod));
	nou->infoUtil = fav;
	nou->next = NULL;
	nou->prev = NULL;
	return nou;
}

ListaDubla* initLista(){
	ListaDubla* listaDubla = (ListaDubla*)malloc(sizeof(ListaDubla));
	listaDubla->prim = NULL;
	listaDubla->ultim = NULL;
	return listaDubla;
}

//inserare la inceput
void inserareInceput(ListaDubla*& listaDubla, Nod* nou){
	if (listaDubla->prim == NULL)
		listaDubla->prim = listaDubla->ultim = nou;
	else{
		nou->next = listaDubla->prim;
		listaDubla->prim->prev = nou;
		listaDubla->prim = nou;
	}
}
```

```
//inserare la sfarsit
void inserareSfarsit(ListaDubla*& listaDubla, Nod* nou){
	if (listaDubla->prim == NULL)
		listaDubla->prim = listaDubla->ultim = nou;
	else{
		nou->prev = listaDubla->ultim;
		listaDubla->ultim->next = nou;
		listaDubla->ultim = nou;
	}
}

//parcurgere
void parcurgere(ListaDubla* listaDubla){
	/*!!!!!!!!!Daca fac ca in acest comentariul voi avea o problema pentru ca nu voi
mai avea listaDubla->prim; pe viitor acesta va fi NULL
	ListaDubla* tmp = listaDubla;
	while (tmp->prim != NULL){
		printf("%s-%d \t", tmp->prim->infoUtil->idAeronava, tmp->prim->infoUtil-
>aeronava);
		tmp->prim = tmp->prim->next;
	}*/

	Nod* primulNod = listaDubla->prim;
	while (primulNod != NULL){
	printf("%s-%d \t", primulNod->infoUtil->idAeronava, primulNod->infoUtil-
>aeronava);
	primulNod = primulNod->next;
	}
}

//stergere dupa un anumit id
void stergereDupaID(ListaDubla*& listaDubla, char* id){
	Nod* primulNod = listaDubla->prim;
	if (primulNod != NULL){
		if (strcmp(primulNod->infoUtil->idAeronava, id) == 0){
			Nod* stergere = primulNod;
			primulNod = stergere->next;
			primulNod->prev = NULL;
			//daca e primul si singurul nod:
			if (listaDubla->prim == NULL)
			  listaDubla->ultim == NULL;
			free(stergere);
		}

		while (primulNod->next != NULL && strcmp(primulNod->next->infoUtil-
>idAeronava, id) != 0)
			primulNod = primulNod->next;
		if (primulNod->next != NULL){
			Nod* stergere = primulNod->next;
			primulNod->next = stergere->next;
			stergere->next->prev = primulNod;
			free(stergere);
		}
	}
}

//citire din fisier
```

```
void citireFisier(ListaDubla*& listaDubla){
	FILE* file = fopen("flaero.txt", "r");
	if (file){
		while(!feof(file)){
			FAV* fav = (FAV*)malloc(sizeof(FAV));

			char buffer[100];
			fscanf(file, "%s", &buffer);
			fav->idAeronava = (char*)malloc(strlen(buffer) + 1);
			strcpy(fav->idAeronava, buffer);

			int tipAero;
			fscanf(file, "%d", &tipAero);
			switch (tipAero)
			{
			case 0:
				fav->aeronava = PAS;
				fscanf(file, "%hd", &fav->incarcatura.nrLocuri);
				break;
			case 1:
				fav->aeronava = CAR;
				fscanf(file, "%lf", &fav->incarcatura.greutateMaxima);
				break;
			}
			Nod* nou = creareNod(fav);
			//inserareInceput(listaDubla, nou);
			inserareSfarsit(listaDubla, nou);
		}
	}
	fclose(file);
}

void main(){
	ListaDubla* lista = initLista();
	citireFisier(lista);
	parcurgere(lista);
	printf("\n---------------------\n");
	stergereDupaID(lista, "A2");
	parcurgere(lista);
}
```

```
//////////LISTA DUBLA CIRCULARA//////////
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

//informatia utila
enum tipAero{ PAS, CAR };

union tipIncarcatura{
	short int nrLocuri;
	double greutateMaxima;
};

//structura de date
struct FAV{
	char* idAeronava;
	tipAero aeronava;
	tipIncarcatura incarcatura;
};

struct Nod{  //next si prev se pune pentru fiecare nod!!!
	FAV* infoUtil;
	Nod* next;
	Nod* prev;
};

struct ListaDubla{
	Nod* prim;
	Nod* ultim;
};

//creare nod
Nod* creareNod(FAV* fav){
	Nod* nou = (Nod*)malloc(sizeof(Nod));
	nou->infoUtil = fav;
	nou->next = NULL;
	nou->prev = NULL;
	return nou;
}

ListaDubla* initLista(){
	ListaDubla* listaDubla = (ListaDubla*)malloc(sizeof(ListaDubla));
	listaDubla->prim = NULL;
	listaDubla->ultim = NULL;
	return listaDubla;
}

//inserare la inceput
void inserareInceput(ListaDubla*& listaDubla, Nod* nou){
	if (listaDubla->prim == NULL)
		listaDubla->prim = listaDubla->ultim = nou;
	else{
		nou->next = listaDubla->prim;
		listaDubla->prim->prev = nou;
		listaDubla->prim = nou;

		//urmatoarele 2 linii de cod o transforma in lista circulara
```

```
			listaDubla->ultim->next = listaDubla->prim;
			listaDubla->prim->prev = listaDubla->ultim;
		}
}

//inserare la sfarsit
void inserareSfarsit(ListaDubla*& listaDubla, Nod* nou){
		if (listaDubla->prim == NULL)
			listaDubla->prim = listaDubla->ultim = nou;
		else{
			nou->prev = listaDubla->ultim;
			listaDubla->ultim->next = nou;
			listaDubla->ultim = nou;

			//urmatoarele 2 linii de cod o transforma in lista circulara
			listaDubla->ultim->next = listaDubla->prim;
			listaDubla->prim->prev = listaDubla->ultim;
		}
}

//parcurgere
void parcurgere(ListaDubla* listaDubla){
		/*!!!!!!!!!Daca fac ca in acest comentariul voi avea o problema pentru ca nu voi
mai avea listaDubla->prim; pe viitor acesta va fi NULL
		ListaDubla* tmp = listaDubla;
		while (tmp->prim != NULL){
		printf("%s-%d \t", tmp->prim->infoUtil->idAeronava, tmp->prim->infoUtil-
>aeronava);
		tmp->prim = tmp->prim->next;
		}*/

		//voi verifica daca nodurile parcurse sunt egale cu primul nod
		Nod* primulNod = listaDubla->prim;
		printf("%s-%d \t", primulNod->infoUtil->idAeronava, primulNod->infoUtil-
>aeronava);
		primulNod = primulNod->next;
		while (primulNod != NULL && primulNod!=listaDubla->prim){
			printf("%s-%d \t", primulNod->infoUtil->idAeronava, primulNod->infoUtil-
>aeronava);
			primulNod = primulNod->next;
		}
}

//stergere dupa un anumit id
void stergereDupaID(ListaDubla*& listaDubla, char* id){
		Nod* primulNod = listaDubla->prim;
		if (primulNod != NULL){
			if (strcmp(primulNod->infoUtil->idAeronava, id) == 0){
				Nod* stergere = primulNod;
				primulNod = stergere->next;
				primulNod->prev = NULL;
				//daca e primul si singurul nod:
				if (listaDubla->prim == NULL)
					listaDubla->ultim == NULL;
				free(stergere);
			}
```

```
		while (primulNod->next != NULL && strcmp(primulNod->next->infoUtil-
>idAeronava, id) != 0)
			primulNod = primulNod->next;
		if (primulNod->next != NULL){
			Nod* stergere = primulNod->next;
			primulNod->next = stergere->next;
			stergere->next->prev = primulNod;
			free(stergere);
		}
	}
}

//citire din fisier
void citireFisier(ListaDubla*& listaDubla){
	FILE* file = fopen("flaero.txt", "r");
	if (file){
		while (!feof(file)){
			FAV* fav = (FAV*)malloc(sizeof(FAV));

			char buffer[100];
			fscanf(file, "%s", &buffer);
			fav->idAeronava = (char*)malloc(strlen(buffer) + 1);
			strcpy(fav->idAeronava, buffer);

			int tipAero;
			fscanf(file, "%d", &tipAero);
			switch (tipAero)
			{
			case 0:
				fav->aeronava = PAS;
				fscanf(file, "%hd", &fav->incarcatura.nrLocuri);
				break;
			case 1:
				fav->aeronava = CAR;
				fscanf(file, "%lf", &fav->incarcatura.greutateMaxima);
				break;
			}
			Nod* nou = creareNod(fav);
			//inserareInceput(listaDubla, nou);
			inserareSfarsit(listaDubla, nou);
		}
	}
	fclose(file);
}

void main(){
	ListaDubla* lista = initLista();
	citireFisier(lista);
	parcurgere(lista);
	printf("\n---------------------\n");
	stergereDupaID(lista, "A2");
	parcurgere(lista);
}
```

```
//////////////////LISTA SIMPLA//////////////////////

#include "stdio.h"
#include "stdlib.h"
#include "string.h"

struct NodLista{
	int infoUtil;
	NodLista* next;
};

//creare nod
NodLista* creareNod(int info){
	NodLista* nou = nullptr;
	nou = (NodLista*)malloc(sizeof(NodLista));
	nou->infoUtil = info;
	nou->next = NULL;
	return nou;
}

//adaugare nod in lista la inceput
void inserareInceput(NodLista*& lista, NodLista* nod){
	if (lista == NULL)
		lista = nod;
	else{
		nod->next = lista;
		lista = nod;
	}
}

//adaugare nod in lista la sfarsit
void inserareSfarsit(NodLista*& lista, NodLista* nod){
	if (lista == NULL)
		lista = nod;
	else{
		while (lista->next != NULL)
			lista = lista->next;
		lista->next = nod;
	}
}

//inserare in interiorul listei dupa un anumit nod dat
void inserareInterior(NodLista*& lista, NodLista* nod, int info){
	NodLista* tmp = lista;
	while (tmp != NULL && tmp->infoUtil != info)
		tmp = tmp->next;
	if (tmp->infoUtil == info){
		nod->next = tmp->next;
		tmp->next = nod;
	}
}

//inserarea dupa ultima aparitie a unei anumite valori date
void inserareInteriorUltimaPozitie(NodLista** lista, NodLista* nou, int info){
	NodLista* tmp = *lista;
	NodLista* ref = NULL;
```

```
	while (tmp->next != NULL){
		if (tmp->infoUtil == info)
			ref = tmp;
		tmp = tmp->next;
	}

	//deoarece ultima pozitie va avea tmp->next false o tratez separat
	if (tmp->infoUtil == info)
		ref = tmp;

	//verific daca am ceva in referinta in care am salvat ultima aparitie si daca am
inserez
	if (ref != NULL){
		nou->next = ref->next;
		ref->next = nou;
	}
}

//parcurgere lista cu afisare
void parcurgere(NodLista* lista){
	while (lista != NULL){
		printf("Info util: %d \n", lista->infoUtil);
		lista = lista->next;
	}
}

//stergerea unui element de pe o anumita pozitie
void stergerePozitie(NodLista** lista, int poz){
	NodLista* tmp = *lista;
	int contor = 1;
	//trebuie sa ma pozitionez inaintea nodului de sters
	while ((contor<poz - 1) && (tmp->next->next!=NULL)){
		tmp = tmp->next;
		contor++;
	}

	if (poz == 1){
		//stergere de la inceput
		NodLista* stergere = *lista;
		*lista = stergere->next;
		free(stergere);
	}
	else if (poz == 2){
		NodLista* stergere = (*lista)->next;
		(*lista)->next = (*lista)->next->next;
		free(stergere);
	}
	else{
		NodLista* stergere = tmp->next;
		tmp->next = stergere->next;
		free(stergere);
	}
}

void main(){
	NodLista* lista = nullptr;

	NodLista* nod = creareNod(7);
```

```
        NodLista* nod2 = creareNod(9);
        NodLista* nod3 = creareNod(3);

        inserareInceput(lista, nod);
        inserareSfarsit(lista, nod2);
        inserareInceput(lista, nod3);

        //parcurgere(lista);

        NodLista* nod4 = creareNod(2);
        inserareInterior(lista, nod4, 7);
        parcurgere(lista);

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

        stergerePozitie(&lista, 4);
        parcurgere(lista);

        printf("\n-----------------------------\n");
        NodLista* nod5 = creareNod(5);
        inserareInceput(lista, nod5);
        NodLista* nod6 = creareNod(5);
        inserareInterior(lista, nod6, 7);
        NodLista* nod7 = creareNod(100);
        inserareInteriorUltimaPozitie(&lista, nod7, 5);
        parcurgere(lista);
}
```

```
//////////////STIVA//////////////////

#include "stdio.h"
#include "stdlib.h"
#include "string.h"

struct NodLista{
	int infoUtil;
	NodLista* next;
};

//creare nod
NodLista* creareNod(int info){
	NodLista* nou = nullptr;
	nou = (NodLista*)malloc(sizeof(NodLista));
	nou->infoUtil = info;
	nou->next = NULL;
	return nou;
}

//put stiva (adaugare elemente in stiva)
void put(NodLista*& lista, NodLista* nod){
	if (lista == NULL)
		lista = nod;
	else{
		nod->next = lista;
		lista = nod;
	}
}

//get stiva (extragere elemente din stiva)
NodLista* get(NodLista*& lista){
	if (lista != NULL){
		NodLista* getNod = lista;
		lista = getNod->next;
		return getNod;
	}
	else
		return nullptr;
}

void main(){
	NodLista* stiva = nullptr;

	NodLista* nod1 = creareNod(8);
	put(stiva, nod1);

	NodLista* nod2 = creareNod(2);
	put(stiva, nod2);

	NodLista* nod3 = creareNod(5);
	put(stiva, nod3);

	NodLista* rezultat = get(stiva);
	printf("Rezultatul este: %d \n", rezultat->infoUtil);

	rezultat = get(stiva);
```

```
	printf("Rezultatul este: %d \n", rezultat->infoUtil);}
////////TABELA DISPERSIE - mecanism CHAINING/////////////
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

enum Furnizor{RDSRCS, TELEKOM, ORANGE, UPC};

//informatia utila
struct Abonat{
	char* nume;
	float valoareAbonament;
	Furnizor tipFurnizor;
};

//structuri de date
struct Nod{
	Abonat* infoUtil;
	Nod* next;
};

struct HashTable{
	(Nod*)* vector; //vector de adrese la noduri
	int dimHT;
};

//functia de hash
int fhush(char* nume, int dimHT){
	return nume[0] % dimHT;
}

// //initializarea structurilor
//creare nod
Nod* creareNod(Abonat* abon){
	Nod* nod = (Nod*)malloc(sizeof(Nod));
	nod->infoUtil = abon;
	nod->next = NULL;
	return nod;
}

//insearare nod in lista (se face la sfarsit pentru ca va fi un vector de liste)
void inserareNodLista(Nod*& lista, Nod* nou){
	Nod* tmp = lista;
	if (lista == NULL)
		lista = nou;
	else{
		while (tmp->next != NULL)
			tmp = tmp->next;
		tmp->next = nou;
	}
}

//initializare tabela de dispersie
HashTable initHT(int dim){
	HashTable HT;
	HT.vector = (Nod**)malloc(sizeof(Nod*)*dim);
	HT.dimHT = dim;
	memset(HT.vector, 0, sizeof(Nod*)*dim);
```

```
	return HT;}

//inserarea nodului in tabela de dispersie
void inserareHT(HashTable HT, Nod* nou){
	//1. calculam functia de hash
	int poz = fhush(nou->infoUtil->nume, HT.dimHT);

	//2. inserare in lista aferenta pozitiei determinata de HT
		//2.1 preluam lista din tabela de dispersie de pe pozitia poz
	Nod* lista = HT.vector[poz];
		//2.2 inseram nodul in lista
	inserareNodLista(lista, nou);
		//2.3 adaugam lista modificata la loc in vector
	HT.vector[poz] = lista;
}

//cautarea dupa un nume
Nod* cautareDupaNume(HashTable ht, char* nume){
	int poz = fhush(nume, ht.dimHT);

	Nod* lista = ht.vector[poz];
	while (lista!=NULL){
		if (strcmp(lista->infoUtil->nume, nume) == 0)
			return lista;
		else
			lista = lista->next;
	}
	return nullptr;
}

//stergere dupa nume
void stergereDupaNume(HashTable ht, char* nume){
	int poz = fhush(nume, ht.dimHT);
	Nod* lista = ht.vector[poz];
	if (lista != NULL){
		if (strcmp(lista->infoUtil->nume, nume) == 0){
			Nod* sterge = lista;
			lista = sterge->next;
			free(sterge);
		}
	}
	Nod* tmp = lista;
		while (tmp->next != NULL && (strcmp(tmp->next->infoUtil->nume, nume) != 0))
			tmp = tmp->next;
		if (tmp->next != NULL){
			Nod* sterge = tmp->next;
			tmp->next = sterge->next;
			free(sterge);
		}

	ht.vector[poz] = lista;
}

//citire din fisier
void citireFisier(HashTable& HT){
	FILE* file = fopen("HT-chaining.txt", "r");
	if (file){
		while (!feof(file)){
```

```
                        Abonat* ab = (Abonat*)malloc(sizeof(Abonat));

                        char buffer[20];
                        fscanf(file, "%s", &buffer);
                        ab->nume = (char*)malloc(strlen(buffer) + 1);
                        strcpy(ab->nume, buffer);

                        fscanf(file, "%f", &ab->valoareAbonament);
                        fscanf(file, "%d", &ab->tipFurnizor);

                        Nod* nod = creareNod(ab);
                        inserareHT(HT, nod);
                }
        }
        fclose(file);
}

//afisare la consola
void showHT(HashTable HT){
        for (int i = 0; i < HT.dimHT; i++){
                printf("------- %d \n", i);
                Nod* lista = HT.vector[i];
                while (lista){
                        printf("%s-%f\n", lista->infoUtil->nume, lista->infoUtil-
>valoareAbonament);
                        lista = lista->next;
                }
        }
}

void main(){
        HashTable HT = initHT(15);
        citireFisier(HT);
        showHT(HT);

        Nod* rezultat = cautareDupaNume(HT, "Alexandra");
        if (rezultat != NULL)
                printf("\n\n Rezultatul cautarii este: %s - %f - %d \n", rezultat-
>infoUtil->nume, rezultat->infoUtil->valoareAbonament, rezultat->infoUtil->tipFurnizor);
        else
                printf("\n\n Nu exista acest abonat!");

        printf("\n\n\n");

        stergereDupaNume(HT, "Andreea");
        showHT(HT);
}
```

```
//////////HASH TABLE - mecanism LINEAR PROBING//////////
#include "stdio.h"
#include "stdlib.h"
#include "string.h"

//informatia utila
struct Student{
	char* nume;
	int nota;
};

//structura de date
struct Nod{
	Student* infoUtil;
	Nod* next;
};

struct HashTable{
	Nod** vector;
	int dimHT;
};

//functia de hash
int fhash(int nota, int dim){
	return nota % dim;
}

//initializarea structurilor
Nod* creareNod(Student* stud){
	Nod* nod = (Nod*)malloc(sizeof(Student));
	nod->infoUtil = stud;
	nod->next = NULL;
	return nod;
}

HashTable* initHT(int dim){
	HashTable* ht = (HashTable*)malloc(sizeof(HashTable));
	ht->vector = (Nod**)malloc(sizeof(Nod*)*dim);
	ht->dimHT = dim;
	//memset(ht->vector, 0, sizeof(Nod)*ht->dimHT);  //cu memset crapa
	for (int i = 0; i < ht->dimHT; i++)
		ht->vector[i] = NULL;
	return ht;
}

void inserareHT(HashTable*& ht, Nod* nou){
	int poz = fhash(nou->infoUtil->nota, ht->dimHT);

	if (ht->vector[poz] == NULL)
		ht->vector[poz] = nou;

	else{
		while (poz < ht->dimHT && ht->vector[poz] != NULL) //parcurgem vectorul
pornind de la pozitia in care am gasit cheia in jos
			poz++;
```

```
		if (poz == ht->dimHT) {//daca am ajuns la finalul vectorului
			poz = fhash(nou->infoUtil->nota, ht->dimHT);

			while (poz>0 && ht->vector[poz] != NULL) //parcurgem vectorul
pornind de la pozitia in care am gasit cheia in sus
				poz--;

			if (poz == 0)  //daca ajungem la inceputul vectorului apare
coliziunea de tip probing si nu putem insera elementul pentru ca vectorul este ocupat in
totalitate
				printf("\nCOLIZIUNE!");
			else{
				ht->vector[poz] = nou;
			}
		}
		else{
			ht->vector[poz] = nou;
		}
	}
}

//cautare in ht
Nod* cautareHT(HashTable* ht, int nota){
	Nod* nodGasit = nullptr;
	int poz = fhash(nota, ht->dimHT);

	if (ht->vector[poz] != NULL){
		if (ht->vector[poz]->infoUtil->nota == nota)
			nodGasit = ht->vector[poz];
	}

	else{
		while (poz < ht->dimHT&&ht->vector[poz] != NULL){
			if (ht->vector[poz]->infoUtil->nota == nota)
				nodGasit = ht->vector[poz];
			else
				poz++;
		}
		if (nodGasit==nullptr){
			int poz = fhash(nota, ht->dimHT);
			while (poz>0 && ht->vector[poz] != NULL){
				if (ht->vector[poz]->infoUtil->nota == nota)
					nodGasit = ht->vector[poz];
				else
					poz--;
			}
		}
	}
	if (nodGasit != nullptr){
		printf("Rezultat = %d: %s - %d", poz, nodGasit->infoUtil->nume, nodGasit-
>infoUtil->nota);
		return nodGasit;
	}
	else
		return nullptr;
}

//stergere
```

```
void stergereHT(HashTable*& ht, int nota){
	bool gasit = false;
	int poz = fhash(nota, ht->dimHT);

	if (ht->vector[poz] != NULL){
		if (ht->vector[poz]->infoUtil->nota == nota){
			free(ht->vector[poz]);
			ht->vector[poz] = NULL;
			gasit = true;
		}
	}

	else{
		while (poz < ht->dimHT&&ht->vector[poz] != NULL){
			if (ht->vector[poz]->infoUtil->nota == nota){
				free(ht->vector[poz]);
				ht->vector[poz] = NULL;
				gasit = true;
			}

			else
				poz++;
		}
		if (!gasit){
			int poz = fhash(nota, ht->dimHT);
			while (poz>0 && ht->vector[poz] != NULL){
				if (ht->vector[poz]->infoUtil->nota == nota){
					free(ht->vector[poz]);
					ht->vector[poz] = NULL;
					gasit = true;
				}
				else
					poz--;
			}
		}
	}
	if (!gasit){
		printf("\nnu exista acest student!");
	}
	else
		printf("\nStudentul a fost sters de pe pozitia %d \n", poz);

}

void showHT(HashTable* ht){
	for (int i = 0; i < ht->dimHT; i++){
		if (ht->vector[i]!=NULL)
			printf("%d: %s - %d \n", i, ht->vector[i]->infoUtil->nume, ht-
>vector[i]->infoUtil->nota);
	}
}

void citireFisier(HashTable*& ht){
	FILE* file = fopen("HT-LinearProbing.txt", "r");
	if (file){
		while (!feof(file)){
			Student* stud = (Student*)malloc(sizeof(Student));
			char buffer[100];
```

```
                    fscanf(file, "%s", &buffer);
                    stud->nume = (char*)malloc(strlen(buffer) + 1);
                    strcpy(stud->nume, buffer);
                    fscanf(file, "%d", &stud->nota);

                    Nod* nod = creareNod(stud);
                    inserareHT(ht, nod);
                }
        }
        fclose(file);
}

void main(){
        HashTable* ht = initHT(20);
        citireFisier(ht);
        showHT(ht);
        Nod* cautare = cautareHT(ht, 45);
        stergereHT(ht, 27);
        showHT(ht);
}
```