```c
/*
 ESTUDANDO PARA A 1º PROVA DE ED: LISTA, PILHA E FILA.

*/

#include<locale.h>
#include<stdio.h>
#include<stdlib.h>

typedef struct lista{
    int info;
    struct lista *prox;
}No;

//LISTA ENCADEADA:

/*INICIALIZA lista: Cria lista com o 1º ponteiro, sendo uma lista vazaia,
logo o ponteiro é NULL. Como retorna o ponteiro então é do tipo No *   */

No *InicializaLista (){
    return NULL;
}

/* INSERE: Para cada elemento inserido na lista, devemos alocar
dinamicamente a memória necessária para armazenar o elemento
 e encadeá-lo na lista existente. Podemos inserir no Início,
 Meio (ordenado por algum critério) e fim.

Na main:    No* L;  L=InicializaLista();    L= InsereInicio(L, 35);
 -> Não esquecer de colocar "L=" para atualizar a cabeça. */

//INÍCIO: O ponteiro que marca o início da lista é atualizado (return nov
//Como há retorno a func é do tipo No*

 No* InsereInicio (No* L, int valor){
    No *novo;                           // 1) Cria o novo
    novo = (No*) malloc(sizeof(No));    // 2) Aloca memória p/ novo
    novo->info = valor;                 // 3) Adiciona o valor ao novo nó
    novo->prox = L;                     // 4) Encadeia o novo à lista
    return novo;                        // 5) Novo é o INÍCIO agora.
 }

    //FIM:  O início da lista não muda.
void InsereFim (No *L, int valor){
// NÃO RETORNA NADA POIS O FIM NÃO É ALTERADO
    No* aux = L;    // NECESSITA AUX apontado para L para percorrer
    No *novo;           // 1) Cria
    novo = (No*) malloc(sizeof(No));  // 2) ALoca
    novo->info = valor;  // 3) Adiciona valor ao Nó
    novo->prox = NULL;  // 4) Como será o fim, o fim aponta pra NULL
    if(L==NULL)        // 5) Caso 1: Lista não inicializada
        L=novo; // novo será o 1º nó.
    else{
        while (aux->prox != NULL)  // Aux para entre o último e o "nulo" para
```

```
55:             aux = aux->prox;     //inserir o novo
56:         aux->prox = novo;
57: // O prox aponta para o novo já está apontado para null
58: //apontar para novo  pois isso já ocorre na linha 44
59:     }
60: }
61:
62: // INSERE ORDENADA, exemplo: insere em ordem crescente:
63: No *InsereCrescente (No *L, int valor){
64:     No *aux = L;
65:     No *ant = NULL;
66:
67:     No *novo = (No*) malloc(sizeof(No));
68:
69:     novo->info = valor;
70:     novo->prox = NULL;
71:
72:     if(aux != NULL){ // SE JÁ HÁ NÓS
73:     //ENQUANTO NÃO ACHAR O UM MAIOR OU NÃO TERMINAR A LISTA
74:         while (aux != NULL && aux->info < valor){
75:          // 1) PERCORRE ATÉ ACHAR O LUGAR DE INSERIR
76:             ant = aux;
77:             aux = aux->prox;
78:         }
79:         // ACHADO O LOCAL, LIGA O "NOVO" COM SEUS SUCESSORES.
80:         novo->prox = aux;
81:         //SE O ANT FOR NULL, É O 1º NÓ
82:         if(ant == NULL) return novo;
83:         else{ //SENÃO LIGA SEUS ANTECESSORES AO "NOVO"
84:             ant->prox = novo;
85:             return L;
86:         }
87:
88:     }
89:     if (aux == NULL){ // CASO SEJA O PRIMEIRO NÓ
90:         return novo;
91:     }
92: }
93: //MERGE (INTERCALA DESORDENADO)
94: N} *Intercala(No* L, No* M){
95:     No* aux1 = L;
96:     No* aux2 = M;
97:     No *aux;
98:     while(aux1->prox != NULL && aux2){
99: // Para aux1 ter mesma quantia de el. de aux2 ou aux1 ser >
100:         aux = aux2->prox;
101:         aux2->prox = aux1->prox;
102:         aux1->prox = aux2;
103:
104:         aux1 = aux2->prox;
105:         aux2 = aux;
106:     }
107:     if(aux1->prox == NULL){ // Aux1 já terminou
108:         aux1->prox = aux2;
```

```
109:     }
110:     return L;
111: }
112:  //REMOVE X ESPECIFICO
113:  /* função retira: retira elemento da lista */
114: No* Remove (No* l, int v) {
115:     No* ant = NULL; /* ponteiro para elemento anterior */
116:     No* p = l; /* ponteiro para percorrer a lista*/
117: /* procura elemento na lista, guardando anterior */
118:     while (p != NULL && p->info != v) {
119:         ant = p;
120:         p = p->prox;
121:     }
122: /* verifica se achou elemento */
123:     if (p == NULL)  return l; /* não achou: retorna lista original */
124: /* retira elemento */
125: //retira elemento do inicio
126:     if (ant == NULL) l = p->prox;
127: //* retira elemento do meio da lista
128:     else ant->prox = p->prox;
129:     free(p);
130:     return l;
131: }
132: //REMOVE DO FIM
133: NODE *RemoveFim (NODE *L){
134:     NODE *aux = L;
135:     NODE *ant = NULL;
136:     while(aux->prox != NULL){
137:         ant = aux;
138:         aux = aux->prox;
139:     }
140:     if(ant == NULL){ //caso de só ter 1 el.
141:         L = NULL;
142:         free(aux);
143:         aux = NULL;
144:     }
145:     else{ //Caso de ter MAIS de 1 el.
146:         ant -> prox = NULL;
147:         free(aux);
148:         aux = NULL;
149:     }
150:     return L;
151: }
152: //Compara lista ou seja, verifica se uma lista é IGUAL a outra.
153: //Supondo que lista1 <= lista2
154:
155: int Compara(NODE *L, NODE *M)
156:     NODE *aux1, *aux2;
157:
158:     while(aux1 && aux1->info == aux2->info){
159:     // Se fosse para quaisquer tamanho de lista:
160:     //aux1 && aux2 && aux1->info == aux2->info
161:         aux1 = aux1->prox;
162:         aux2 = aux2->prox;
```

```
163:     }
164:     if(aux1) return 0 // ou if(aux1 != NULL)
165:     //ou seja ainda existe lista
166:     else return1;
167: }
168: // Verifica se há uma sublista em uma lista.
169: //Recursiva.Ex procura 123 em 12345
170: int isSub(NODE* L1, NODE*L2){
171: 
172:     if(L2){
173:     //Condição de parada BackTrack
174:         if(compara(L1,L2))  return 1
175:         else isSub(L1, L2->prox)     return 0;
176:     }
177: }
178: //IMPRIME:
179: void Imprime (NODE *L){
180:     NODE *aux = L;
181:     while (aux != NULL){
182:         printf("|%d| -> ", aux->info);
183:         aux = aux->prox;
184:     }
185:     printf("NULL\n");
186: }
187: /* Função imprime recursiva em ordem */
188: void imprime_rec (No* L){
189:     if (L == NULL) printf(" NULL\n");
190: /* imprime primeiro elemento */
191:     printf(" |%d|->", L->info);
192: /* imprime sub-lista */
193:     imprime_rec(L->prox);
194: }
195: /* Função imprime recursiva do fim para o ini */
196: void imprime_rec (No* L){
197:     if (L == NULL) printf("NULL \n");
198: /* imprime sub-lista */
199:     imprime_rec(L->prox);
200: /* imprime primeiro elemento */
201:     printf(" -> |%d|", L->info);
202: }
203: 
204: //DESALOCA:
205: void DesfazLista(No*L){
206:     No*aux = L;
207:     while(L != NULL){
208:         aux = L;
209:         L = L->prox;
210:         free(aux);
211:     }
212:     aux=NULL;
213: }
214: int VerificaVazia (NODE *L){ //Verifica se a lista está vazia
215:     if (L != NULL)  // Se lista NÃO vazia retorna 0
216:         return 0;
```

```
217:     else    return 1;
218: }
219: //---------------------------------------------
220: //Pilha * Pilha *Pilha *Pilha *Pilha *Pilha *Pilha *Pilha *Pilha *Pilha *
221: /*A ideia fundamental da pilha é que todo o acesso
222: a seus elementos é feito através do seu topo. Quando
223: um elemento novo é introduzido na pilha, passa a ser o
224: elemento do topo,e o únicoelemento que pode ser
225: removido da pilha é o do topo. O primeiro
226: que sai é o último que entrou (LIFO)
227: FUNÇÕES:
228: typedef struct pilha Pilha;
229:
230: Pilha* cria (void);
231: void push (Pilha* p, float v);
232: float pop (Pilha* p);
233: int vazia (Pilha* p);
234: void libera (Pilha* p);
235: A função cria aloca dinamicamente a estrutura da pilha,
236: inicializa seus campos e retorna seu ponteiro;
237: as funções push e pop inserem e retiram, respectivamente,
238: um valor real na pilha; a função vazia informa se a
239: pilha está ou não vazia; e a função libera destrói
240: a pilha, liberando toda a memória usada pela estrutura.*/
241:
242: struct pilha{
243:     No* topo; // NOTE QUE É DO TIPO LISTA
244: };
245:
246: typedef struct pilha PILHA;
247:
248: PILHA *CriaPilha(){
249:     PILHA *pilha = (PILHA*)malloc(sizeof(PILHA));
250:     pilha->topo = NULL; // TOPO É NULL = 0 EL.
251:     return pilha;
252: }
253: int VerificaPilhaVazia (PILHA *pilha){
254:     if(pilha->topo == NULL)
255:         return 1;
256:     else return 0;
257: }
258: //Insere elemento na Pilha: EMPILHA
259: void Push (PILHA *pilha, int valor){
260:     No* novo = (No*) malloc(sizeof(No));
261:     novo->info = valor;
262:     novo->prox = pilha->topo;//Empilha
263:     pilha->topo = novo;//Atualiza topo
264:     // cuidado: pilha é ponteiro logo não é apenas topo = novo.
265: }
266:
267: //Remove elemento: DESEMPILHA
268: int Pop (PILHA *pilha){
269:     No *aux=pilha->topo;
270:     int val;
```

```
271:     //ou      = p->topo->prox
272:     pilha->topo=aux->prox;
273: //topo agora é o elemento debaixo.
274:     val=aux->item; //Retornará o valor antigo do topo
275:     free(aux);
276:     return val;
277: }
278: //RETORNA TOPO:
279: int Top (PILHA *pilha){
280:     return (pilha -> topo)->item;
281: }
282: //DESTROI PILHA
283: void DestruirPilha(PILHA *pilha){
284:     while(!VerificaPilhaVazia(pilha)){
285:         Pop(pilha);
286: //Enquanto a pilha não está vazia, remove o de cima
287:     }
288: //Quando estiver vazia dá um free pilha e aponta pra null
289:     free(pilha);
290:     pilha = NULL;
291: }
292: //REMOVE elemento específico
293: void removeElem(PILHA *pilha, int elem){
294:     PILHA *aux;
295:     aux = criaPilha();
296:     // cria uma pilha auxiliar para empilhar os valores do topos
297:     //removidos da "pilha" e poder volta-los para ela posteriormente
298:     while (!VerificaPilhaVazia(pilha)){
299:         if(top(pilha) != elem)
300: //Se o topo não for o elemento, insere o topo na aux e remove o topo
301:             push(aux, pop(pilha)); //da pilha
302:         else{
303:             pop(pilha)); //ACHOU O EL., REMOVE.
304:             break;
305:  // tem que parar o while senão ele vai até esvaziar a pilha.
306:  //Sem o break, caso tivesse elemento repetido ele tbm removeria.
307:         }
308:     }
309:     while(!VerificaPilhaVazia(aux)){
310:         push(pilha, pop(aux)); // RETORNA OS ELEMENTOS DE AUX PRA PILHA
311:         //REMOVENDO-OS DE AUX AO MESMO TEMPO.
312:     }
313:
314: //IMPRIME PILHA
315: int ImprimePilha(PILHA *P){ //Imprime do último inserido para o primeiro
316:     PILHA *aux = CriaPilha();
317:
318:     while(!VerificaPilhaVazia(P)){
319:         printf("\t|%d|\n", Top(P));
320:         Push(aux, Pop(P));
321:     }
322:     while(!VerificaPilhaVazia(aux)){
323:         Push(P, Pop(aux));
324:     }
```

```
325: }
326: void removeElemRepetido(PILHA *pilha, int elem){//Deixa apenas 1 elem.
327:     PILHA *aux;
328:     aux = CriaPilha();
329:     int cont = 0;
330:     // cria uma pilha auxiliar para empilhar os valores do topos
331:     //removidos da "pilha" e poder volta-los para ela posteriormente
332:     while (!VerificaPilhaVazia(pilha)){
333:         if(Top(pilha) != elem)
334: //Se o topo não for o elemento, insere o topo na aux e remove o topo
335:             Push(aux, Pop(pilha)); //da pilha
336:         else{
337:             if(cont==0){//Se achou o 1º, deixa ele
338:                 Push(aux, Pop(pilha));
339:                 cont++;
340:             }
341:             if(cont>0 && Top(pilha)==elem)
342:                 Pop(pilha); //Se achou mais de um remove
343:         }
344:     }
345:     while(!VerificaPilhaVazia(aux)){
346:         Push(pilha, Pop(aux)); // RETORNA OS ELEMENTOS DE AUX PRA PILHA
347:         //REMOVENDO-OS DE AUX AO MESMO TEMPO.
348:     }
349: }
350: //----------------------------------------
351: /*FILA*FILA*FILA*FILA*FILA*FILA*FILA*FILA*FILA*FILA*FILA*FILA*FILA*FILA*F
352: O que diferencia a fila da pilha é a
353: ordem de saída dos elementos: enquanto na pilha
354: “o último que entra é o primeiro que sai”, na fila
355: “o primeiro que entra é o primeiro que sai(FIFO)
356: inserir um novo elemento no final da fila e só
357: podemos retirar o elemento do início.
358:
359: Fila* cria (void);
360: void insere (Fila* f, float v);
361: float retira (Fila* f);
362: int vazia (Fila* f);
363: void libera (Fila* f);*/
364:
365: typedef struct fila{
366:     No *Ini;
367:     No *fim;
368: }Fila;
369:
370: Fila* CriaFila(){
371:     Fila* fila = (Fila*)malloc(sizeof(Fila));
372:     fila->Ini = NULL;
373:     fila->fim = NULL;
374:     return fila;
375: }
376:
377: int VerificaFilaVazia(Fila *fila){
378:     if(fila->Ini == NULL)   return 1;
```

```
379:     else return 0;
380: }
381: //Insere no fim e remove do inicio.
382: void InsereFila(Fila*fila, int valor){
383:     No* novo = (No*)malloc(sizeof(No));
384:     novo->info = valor;
385:     novo->prox = NULL;
386:     if(VerificaFilaVazia(fila)){
387:         fila->Ini = novo;
388:         fila->fim = novo;
389:     }
390:     else{//Se já possui elemento
391:         (fila->fim)->prox = novo;
392:         fila->fim = novo;
393:     }
394: }
395: //Remove (no inicio)
396: int removeFila(Fila* fila){
397:     No*aux = fila->Ini;
398:     int valor;
399:     fila->Ini = aux->prox;
400:     valor = aux->info;
401:     free(aux);
402:     aux = NULL;
403:     if(fila->Ini = NULL)
404:         fila->fim = NULL;
405:     return valor; //valor que estava no inicio.
406: }
407:
408: //Verifica inicio:
409: int VerificaIni(Fila* fila){
410:     return (fila->Ini)->info;
411: }
412:
413: //Fila com prioridade:Qto > a priori < o num.
414: //Insira o abaixo ao invés do Nó da lista.
415: typedef struct NoPriori{
416:     int info;
417:     int priori;
418:     struct NoPriori *prox;
419: }NoPri;
420:
421: void insereFilaPri(Fila* fila, int val, int prior){
422:     NoPri* novo = (NoPri*)malloc(sizeof(NoPri));
423:     novo->info = val;
424:     novo->priori = prior;
425:     if(VerificaFilaVazia(fila)){
426:         fila->Ini = novo;
427:         fila->fim = novo
428:         novo->prox = NULL;
429:     }
430:     else{
431:         while(aux != NULL && aux->priori <= prior){
432:             ant = aux;
```

```
433:             aux = aux->prox;
434:         }
435:         if(ant == NULL){ //Inserir no Ini
436:             novo->priori = aux;
437:             fila->Ini = novo;
438:         }
439:         else{//Insere meio ou fim
440:             ant->prox = novo;
441:             novo->prox = aux;
442:             if(aux==NULL)
443:                 fila->fim = novo;
444:         }
445:     }
446: }
447: //Destruir fila
448:
```