Ficha de Trabalho n.º 5

Ponteiros para *structs*

```c
struct dados_aluno
{
  char curso[5];
  int numero;
  char nome[50];
  float media;
};
typedef struct dados_aluno aluno;

int main(int argc, char *argv[])
{
  aluno a, *p;
  p = &a;

  //atribuicao de dados aluno
  strcpy((*p).nome,"ze");
  strcpy((*p).curso,"lei");
  (*p).numero = 123;
  (*p).media = 12.0f;
```

ou

```c
  strcpy(p->nome,"ze");
  strcpy(p->curso,"lei");
  p->numero = 123;
  p->media = 12.0f;
```

```c
  //apresentacao de dados aluno
  printf("nome: %s\n",(*p).nome);
  printf("curso: %s\n",(*p).curso);
  printf("numero: %d\n",(*p).numero);
  printf("media: %f\n",(*p).media);
```

ou

```c
  printf("nome: %s\n",p->nome);
  printf("curso: %s\n",p->curso);
  printf("numero: %d\n",p->numero);
  printf("media: %f\n",p->media);

  return 0;
}
```

`(*nome_ponteiro).campo` ou `nome_ponteiro->campo`

Usando subprogramas para ler os dados de um aluno:

```c
void le_aluno1(aluno *pa)
{
  printf("curso: ");
  gets(pa->curso);
  printf("nome: ");
  gets(pa->nome);
  printf("numero: ");
  scanf("%d",&pa->numero);
  printf("media: ");
  scanf("%f",&pa->media);
}
```

```c
aluno * le_aluno2()
{
  aluno *pa;
  pa = (aluno*)malloc(sizeof(aluno));

  printf("curso: ");
  gets(pa->curso);
  printf("nome: ");
  gets(pa->nome);
  printf("numero: ");
  scanf("%d",&pa->numero);
  printf("media: ");
  scanf("%f",&pa->media);
  return pa;
}
```

```c
int main(int argc, char *argv[])
{
  aluno a, *p;
  p = &a;
  le_aluno1(p);
  // ou apenas
  aluno a;
  le_aluno1(&a);

  return 0;
}
```

```c
  aluno *p;
  p = le_aluno2();
```

Ponteiros para tabelas de *structs*

```
/*…*/
int main(int argc, char *argv[])
{
	int n, i;
	aluno * pt_alunos;

	printf("numero de alunos: ");
	scanf("%d",&n);

	pt_alunos = (aluno*)malloc(n*sizeof(aluno));
// ponteiro pt_alunos guarda endereco de memoria que ocupa
// [numero de alunos x numero de bytes ocupados por um aluno] bytes
// ou seja, reserva espaco de memoria para n alunos
// necessario fazer casting porque malloc devolve ponteiro genérico
// se pt_alunos for NULL entao nao foi possivel reservar memoria

	for(i = 0; i < n; i++)
		le_aluno1(&pt_alunos[i]);

	for(i = 0; i < n; i++)
		mostra_aluno1(&pt_alunos[i]);

	free(pt_alunos);
// torna disponivel o espaço apontado pelo ponteiro pt_alunos
	return 1;
}
```

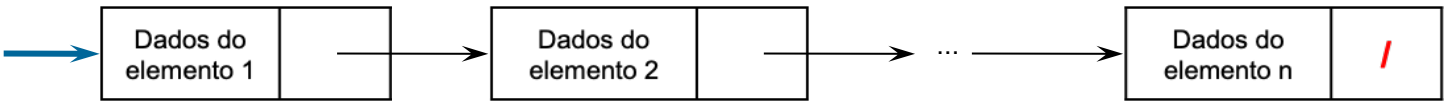
1. Relembre o exercício 4. da Ficha de Trabalho n.º 4, no qual uma pequena empresa familiar pretendia organizar os dados dos seus funcionários (número, nome, tarefa, salário). Foi definida uma estrutura de dados para guardar os dados de cada funcionário e declarada uma tabela para armazenar os dados de todos os funcionários.

   Em vez de definir uma tabela, defina um ponteiro para *n* funcionários (*n* indicado pelo utilizador) e refaça os seguintes subprogramas:

   i. introduzir os dados de um funcionário na lista de funcionários;
   ii. listar todos os funcionários e respectivos dados;
   iii. listar os funcionários com salário superior a 500€;
   iv. procurar e apresentar todos os dados de um funcionário, usando o seu nome;
   v. actualizar os dados de um funcionário (usando o seu número).

Listas ligadas

As listas ligadas são um tipo de estrutura de dados mais versátil com dimensão ajustável.



Cada elemento da lista (nó) é, habitualmente, uma estrutura composta por dois campos:

- informação (inteiro, real, tabela, estrutura composta,...)
- ponteiro para o elemento seguinte

O ponteiro do último elemento da lista deve ser **NULL** (/) para indicar o fim da lista.

Não é necessário guardar o acesso a todos os elementos, basta o **ponteiro para o primeiro elemento da lista**. **Este ponteiro nunca deve ser perdido** (caso aconteça perde-se acesso à lista)**.**

Quando se começa a construir a lista, ela está vazia e, por isso, o ponteiro para o primeiro elemento é NULL.

Cada elemento desta lista é normalmente uma estrutura do tipo:

```
struct nome_do_tipo
{
      // Dados
      struct nome_do_tipo *seg; // ponteiro para o seguinte
};
```

Inserir um elemento numa lista ligada:

O novo elemento deve ser criado antes de ser colocado na lista:

1. Alocar memória para o novo elemento (usando `malloc`)
2. Preencher o novo elemento:
    - colocar a NULL o ponteiro para o seguinte
    - atribuir valor aos restantes campos.

A inserção na lista depende da posição da lista onde se pretende colocar (início, fim, n-ésima posição, antes ou depois de determinado elemento,...).

Remover um elemento de uma listas ligadas

Se a lista está vazia, não há valor a remover.

Se o elemento a eliminar existe:

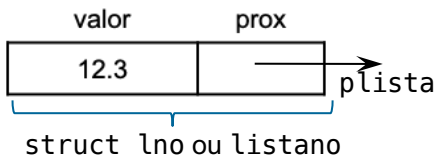
1. colocar um ponteiro a apontar para o elemento a remover
2. garantir que não é perdido o acesso à lista que está ligada ao elemento a remover através de ponteiros
3. libertar a memória ocupada pelo elemento a remover (usando `free`).

2. Considere as seguintes definições de tipos que permitem representar uma lista ligada linear simples de reais:

```
typedef struct lno * plista;
typedef struct lno
{
      float valor; // campo valor é um real
      plista prox; // campo prox é um ponteiro para um nó da lista
} listano;
```

ou

```
struct lno
{
      float valor;
      struct lno * prox;
};
typedef struct lno listano;
```




```
int main(int argc, char *argv[])
{
     // declara lista vazia
     plista plst = NULL;

     // coloca valor 12.3 no inicio da lista
     plista no = (plista)malloc(sizeof(listano));
     no->valor = 12.3;
     no->prox = plst;
     plst = no;

     // remove primeiro elemento da lista, considerando lista nao vazia
     plista paux = plst;
     plst = plst->prox;
     free(paux);

      return 1;
}
```

Escreva subprogramas que desempenhem cada uma das seguintes tarefas. Organize o main de forma que seja possível testar cada uma das alíneas.

Sugestão: Comece por organizar o main para facilitar os testes dos vários subprogramas.

a) Criar e devolver uma lista vazia;

```
plista crialista()
{
      plista p = NULL;
      return p;
}
```

b) Indicar se uma lista é vazia;

```
int listavazia(plista lst)
{
      if (lst == NULL)
            return 1;
      else
            return 0;
}
```

c) Mostrar uma lista;

```
void escrevelista(plista lst)
```

```
{
	if(listavazia(lst))
		printf("a lista e vazia!!!\n");
	else
	{
		printf("[");
		do
		{
			printf(" %.1f,",lst->valor);
			lst = lst->prox;
		}while(lst != NULL);
		printf("\b ]\n");
	}
}
```

d) Acrescentar um elemento *x* no início de uma lista;

```
void junta_no_ini_lista(float x, plista * lst)
{
	plista no = (plista)malloc(sizeof(listano));

	if (no == NULL)
	{
		perror("ERRO!!! Nao ha memoria disponivel...");
		exit(-1);
	}
	else
	{
		no->valor = x;
		no->prox = * lst;
		* lst = no;
	}
}
```

e) Acrescentar um elemento *x* no fim de uma lista;

f) Remover o primeiro elemento de uma lista;

g) Remover o último elemento de uma lista;

h) Remover o *n*–ésimo elemento de uma lista;

i) Remover a primeira ocorrência do elemento *x* de uma lista;

j) Remover todas as ocorrências do elemento *x* de uma lista;

k) Determinar o comprimento (número de elementos) de uma lista;

l) Procurar um valor *x* numa lista devolvendo um ponteiro para este elemento;

m) Ler os valores de uma lista, construindo-a;

n) Limpar uma lista;

o) Mostrar o primeiro elemento de uma lista;

p) Mostrar o último elemento de uma lista;

q) Mostrar a cauda (todos os elementos excepto o primeiro) de uma lista;

r) Juntar duas listas numa só;

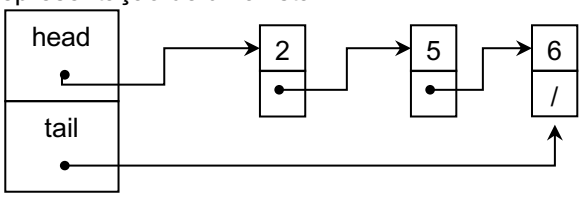
s) Copiar uma lista para outra;

t) Trocar dois elementos de uma lista;

u) Ordenar uma lista por ordem crescente dos seus valores;

v) Ordenar uma lista por ordem decrescente dos seus valores.

3. Considere a seguinte representação de uma lista:



a) Crie uma nova estrutura lista ajustada a esta nova forma;

b) Altere as alíneas a), b), e), g), j), n), p) e t) do exercício anterior de modo que possam ser usadas nesta nova representação.

4. Considere a estrutura de dados (abstracta) *Pilha* (*stack*, LastInFirstOut).

Elabore subprogramas que executem as operações usadas habitualmente com esta estrutura:

a) Criar uma pilha vazia;

b) Indicar se uma pilha está vazia;

c) Empilhar um elemento na pilha (*push*);

d) Retirar um elemento da pilha (*pop*);

e) Mostrar o elemento que está no topo da pilha (*top*).

5. Considere a estrutura de dados (abstracta) *Fila* (*queue*, FirstInFirstOut).

Elabore subprogramas que executem as operações usadas habitualmente com esta estrutura:

a) Criar uma fila vazia;

b) Indicar se uma fila está vazia;

c) Inserir um elemento no fim da fila (juntar);

d) Remover um elemento do início da fila (remover);

e) Mostrar o elemento que está no início da fila (frente).

6. Refaça o exercício 4. da Ficha de Trabalho n.º 4 (referido no exercício 1.) usando uma lista ligada de funcionários em vez de uma tabela.

a) Defina uma estrutura de dados que, além dos dados de um funcionário, tenha um ponteiro.

b) Elabore subprogramas para

i. introduzir os dados de um funcionário na lista;

ii. apresentar todos os dados dos funcionários na lista;

iii. apresentar todos os dados dos funcionários com salário superior a 500€;

iv. procurar e devolver uma estrutura funcionário, usando o seu nome (caso o funcionário não exista, deverá ser devolvido NULL);

v. actualizar os dados de um funcionário (usando o seu número);

vi. ordenar a lista por ordem alfabética dos nomes dos funcionários.