Exercícios de Fila

```
// Alan Vinícius Sega – 17754680
// Eduardo Kenzo Hoshino - 17093402

// Estruturas para Fila
typedef struct tipoElemento {
	int elemento;
	struct tipoElemento *prox;
} elementoFila;

typedef struct tipoFila {
	elementoFila *primeiro, *ultimo;
} Fila;

Fila *init() {
	Fila *f = (Fila *) malloc(sizeof(Fila));
	f->primeiro = NULL;
	f->ultimo = NULL;
	return f;
}

int empty(Fila *f) {
	if (f->primeiro == NULL) {
		return 1;
	}

	return 0;
}

int first(Fila *f, int *elemento) {
	if (empty(f)) {
		return 0;
	}

	*elemento = f->primeiro->elemento;

	return 1;
}

int enqueue(Fila *f, int elemento) {
	elementoFila *e = (elementoFila *) malloc(sizeof(elementoFila));
	if (e == NULL) {
		return 0;
	}

	if(empty(f)) {
		f->primeiro = e;
	} else {
		f->ultimo->prox = e;
	}
```

```
        f->ultimo = e;
        e->elemento = elemento;
        e->prox = NULL;

        return 1;
}

int dequeue(Fila *f, int *elemento) {
        if (empty(f)) {
                return 0;
        }

        *elemento = f->primeiro->elemento;
        f->primeiro = f->primeiro->prox;
        return 1;
}

// Exericios de Fila

// 1. Implementar a função print que imprime os elementos da fila
int print(Fila *f) {
        if (empty(f)) {
                return 0;
        }

        elementoFila *e = (elementoFila *) malloc(sizeof(elementoFila));

        if (e == NULL) {
                return 0;
        }

        e = f->primeiro;

        while(e != NULL) {
                printf("%d", e->elemento);
                e = e->prox;
                if (e != NULL) {
                        printf(", ");
                }
        }

        printf("\n");
        return 1;
}

// 2. Dada uma fila, construir um procedimento que elimina dessa fila os n primeiros
elementos. Prever a possibilidade da fila estar vazia e de não ter n elementos.
int contaElementos(Fila *f) {
        int qtd = 0;
        elementoFila *e = f->primeiro;
```

```
	while(e != NULL) {
		qtd++;
		e = e->prox;
	}

	return qtd;
}

int nDequeue(Fila *f, int n) {
	if (empty(f)) {
		return 0;
	}

	int *i = malloc(sizeof(int));

	if (i == NULL) {
		return 0;
	}

	// Verificando se a quantidade de elementos a se remover a maior do que a
quantidade de elementos na fila.
	int qtd = contaElementos(f);

	// Se a quantidade de elementos for maior que o número a se deletar, programa
para.
	if (n > qtd) {
		return 0;
	} else {
		while(n > 0) {
			dequeue(f, i);
			n--;
		}
	}

	return 1;
}

/* 3. Dadas duas filas: Q1 e Q2, contendo números inteiros ordenados. Construir um
procedimento que
* recebe as duas filas e faz a junção das duas gerando uma terceira. Essa terceira fila
deverá ter
* seus elementos ordenados. As duas filas dadas Q1 e Q2 ficarão vazias ao final do
processo.
*/
int ordenaFila(Fila *f) {
	if (empty(f)) {
		return 0;
	}

	elementoFila *e = (elementoFila*) malloc(sizeof(elementoFila));
```

```
	if (e == NULL) {
		return 0;
	}

	int i, qtd = 0;

	e = f->primeiro;
	while (qtd >= 0){
		if (e->elemento > e->prox->elemento) {
			i = e->elemento;
			e->elemento = e->prox->elemento;
			e->prox->elemento = i;
			qtd++;
		}
		e = e->prox;

		if (e->prox == NULL) {
			if (qtd == 0) {
				break;
			}
			e = f->primeiro;
			qtd = 0;
		}
	}

	return 1;
}

Fila* juntaFilas(Fila *q1, Fila *q2) {
	if (empty(q1) || empty(q2)) {
		exit(1);
		}

	Fila *q3 = (Fila*) malloc(sizeof(Fila));

	if (q3 == NULL) {
		exit(1);
	}

	int *i = malloc(sizeof(int));

	while(!empty(q1)) {
		dequeue(q1, i);
		enqueue(q3, *i);
	}

	while(!empty(q2)) {
		dequeue(q2, i);
		enqueue(q3, *i);
	}
```

```
	ordenaFila(q3);

	return q3;
}


// 4. Construir uma função que receba duas filas Q1 e Q2, contendo números inteiros ordenados e constrói uma terceira fila, contendo os elementos da fila Q1 sem os elementos de interseção de Q1 com Q2.
int removeIntersec(Fila *f) {
	if (empty(f)) {
		return 0;
	}

	elementoFila *e = (elementoFila*) malloc(sizeof(elementoFila));

	if (e == NULL) {
		return 0;
	}

	int i;

	e = f->primeiro;

	while(e->prox != NULL) {
		if(e->elemento == e->prox->elemento) {
			e->prox = e->prox->prox;
		}
		e = e->prox;
	}

	return 1;
}

Fila* agrupaFilas(Fila *q1, Fila *q2) {
	if (empty(q1) || empty(q2)) {
		exit(1);
	}

	Fila *f = juntaFilas(q1, q2);

	if (f == NULL) {
		exit(1);
	}

	removeIntersec(f);

	return f;
}
```

```c
/* 5. Uma fila de prioridades é organizada de acordo com a prioridade (de 0 a 9) de um elemento. Desta forma,
* o registro de dados deve conter o valor do elemento e sua prioridade. Elementos com prioridade 0 devem
* estar no início da fila, seguidos dos elementos de prioridade 1, até que, por fim, estarão os elementos
* de prioridade 9. Desta forma, ao adicionar um novo elemento na fila, este deverá ser adicionado após o
* último elemento de mesma prioridade. Como a fila estará ordenada pela prioridade, a remoção não sofre
* alteração. Implemente uma fila de prioridades, com as estruturas necessárias e as seguintes operações
* init, empty, enqueue e dequeue.
*/

typedef struct tipoElementoPrioridade {
	int elemento;
	int prioridade;
	struct tipoElementoPrioridade *prox;
} elementoFilaPrioridade;

typedef struct tipoFilaPrioridade {
	struct tipoElementoPrioridade *primeiro, *ultimo;
} FilaPrioridade;

FilaPrioridade *init() {
	FilaPrioridade *f = (FilaPrioridade *) malloc(sizeof(FilaPrioridade));
	f->primeiro = NULL;
	f->ultimo = NULL;
	return f;
}

int empty(FilaPrioridade *f) {
	if (f->primeiro == NULL) {
		return 1;
	}

	return 0;
}

int first(FilaPrioridade *f, int *elemento, int *prioridade) {
	if (empty(f)) {
		return 0;
	}

	*elemento = f->primeiro->elemento;
	*prioridade = f->primeiro->prioridade;

	return 1;
}
```

```
int enqueue(FilaPrioridade *f, int elemento, int prioridade) {
	elementoFilaPrioridade *e = (elementoFilaPrioridade *)
malloc(sizeof(elementoFilaPrioridade));
	elementoFilaPrioridade *e2 = (elementoFilaPrioridade *)
malloc(sizeof(elementoFilaPrioridade));
	if (e == NULL || e2 == NULL) {
		return 0;
	}

	if (prioridade < 0 || prioridade > 9){
		// printf("Prioridade deve ser entre 0 e 9!\n");
		return 0;
	}

	if(empty(f)) {
		f->primeiro = e2;
	} else {
		e = f->primeiro;
		while(e->prioridade <= prioridade) {
			if(e->prox == NULL) {
				f->ultimo = e2;
				break;
			} else {
				if (e->prox->prioridade <= prioridade) {
					e = e->prox;
				} else {
					break;
				}
			}
		}
		e2->prox = e->prox;
		e->prox = e2;
	}

	e2->elemento = elemento;
	e2->prioridade = prioridade;

	return 1;
}

int dequeue(FilaPrioridade *f, int *elemento) {
	if (empty(f)) {
		return 0;
	}

	*elemento = f->primeiro->elemento;
	f->primeiro = f->primeiro->prox;
	return 1;
}

int print(FilaPrioridade *f) {
```

```
	if (empty(f)) {
		return 0;
	}

	elementoFilaPrioridade *e = (elementoFilaPrioridade *)
malloc(sizeof(elementoFilaPrioridade));

	if (e == NULL) {
		return 0;
	}

	e = f->primeiro;
	printf("ELEMENTOS - ");

	while(e != NULL) {
		printf("%d", e->elemento);
		e = e->prox;
		if (e != NULL) {
			printf(", ");
		}
	}

	e = f->primeiro;
	printf("\nPRIORIDAD - ");

	while(e != NULL) {
		printf("%d", e->prioridade);
		e = e->prox;
		if (e != NULL) {
			printf(", ");
		}
	}
	printf("\n");

	return 1;
}
```