Universidade Tecnológica Federal do Paraná (UTFPR)
Departamento Acadêmico de Informática (DAINF)
Estrutura de Dados I
Professor: Rodrigo Minetto
Lista de exercícios (estrutura de dados fila circular)

Exercícios (seleção): necessário entregar TODOS (moodle)!

Exercício 1) Escreva um programa que, dado uma sequência de números aleatórios — gerados através de um *while* — determina se o número é par ou ímpar e o insere em uma fila para pares ou em uma outra fila para ímpares e no final imprime o conteúdo de ambas as filas. Para resolver este exercício utilize as operações definidas para a interface da fila circular (create, enqueue, dequeue, empty, front, print, ...). **Não** acesse os elementos armazenados na fila senão pelas operações (funções) definidas pela interface (ou seja, suponha que você não tenha acesso ao código fonte que implementa a fila circular, portanto não use o operador $\rightarrow$ para acessar os elementos da fila diretamente, se precisar de algum elemento use front, enqueue, dequeue, ...).

Exercício 2) Máquinas caça-níqueis são os principais geradores de receita para quase todos os cassinos físicos e on-line. Os componentes básicos de uma máquina caça-níqueis são:

- Moeda de caça-níquel: lugar onde você insere dinheiro ou cartões de crédito;
- Carretéis: rolo giratório com diversas imagens que formam combinações;
- Alavanca: ao puxá-la, os carretéis giram;
- Pagamento ativo: determina o valor do pagamento;

Os carretéis são rolos com grupos de símbolos que, na maioria das vezes, giram no sentido vertical. A figura abaixo ilustra uma dessas máquinas.

Sua tarefa nesse exercício é simular o movimento de 3 carretéis que contém a combinação de números de 1 até 9 em cada um deles. Os carretéis em movimento devem ser simulados através de uma fila, onde o giro aleatório para cada um dos carretéis pode ser simulado ao desinfileirar e enfileirar novamente os elementos na fila. A combinação dos números a cada sequência de giros indica se o jogador ganhou ou não. Considere como os números sorteados aqueles que ficam no início da

fila após o giro aleatório em cada fila. O jogador ganha caso os 3 números que estiverem no início de cada uma das filas forem iguais. Por exemplo:

```
5 4 1
7 3 2
1 2 4
9 1 4
4 3 5
7 7 7 -> Ganhou
```

A parte de recompensa não precisa ser implementada, apenas os sorteios e lógica para decidir se ganhou ou se o jogo continua.

Utilize uma fila circular e operações definidas pela interface para resolver este problema.

Exercícios (aprofundamento): não é necessário entregar mas é importante estudar!

Exercício 3) Qual a complexidade, utilizando a notação assintótica $\mathcal{O}(?)$, para as seguintes operações básicas de uma estrutura de dados **fila circular**

	Melhor caso	Pior caso
enqueue		
dequeue		
front		
empty		
busca		

Exercício 4) Em uma fila **não circular** — codificada em um vetor — sugere-se que a cada operação para desenfileirar um novo elemento todos os elementos remanescentes na fila sejam deslocados para frente (conforme explicado em aula). Uma estratégia alternativa é adiar o deslocamento até que a cauda da fila atinja a última posição do vetor. Quando essa situação ocorre e há uma nova tentativa para inserir um novo elemento na fila, a fila inteira é deslocada para frente, de modo que o primeiro elemento da fila fique na primeira posição do vetor. Quais são as vantagens e desvantagens dessa estratégia?

Exercício 5) Descreva que modificações você faria nas estruturas de dados para fila circular, vistas em aula, para substituir a técnica do sacrifício de uma posição para diferenciar uma fila vazia de uma fila cheia (outras opções seriam um contador para a quantidade de elementos armazenados ou uma flag para indicar fila cheia).

Exercício 6) O algoritmo **merge-sort** é um clássico em ordenação. Um de seus passos consiste em unir dois conjuntos em ordem crescente em um novo conjunto também em ordem crescente. Por exemplo, suponha as seguintes filas em ordem crescente $a = \{1, 3, 5\}$ e $b = \{2, 4, 5, 6, 8\}$ então a saída

é uma nova fila $c = \{1, 2, 3, 4, 5, 5, 6, 8\}$. Escreva uma função **merge** especial que utiliza o seguinte protótipo:

```
Queue* merge (Queue *a, Queue *b);
```

As filas a e b devem ficar vazias no decorrer da execução da função merge.

Para resolver este exercício utilize as operações definidas para a interface da fila circular (create, enqueue, dequeue, empty, front, ...). **Não** acesse os elementos armazenados na fila senão pelas operações (funções) definidas pela interface (ou seja, suponha que você não tenha acesso ao código fonte que implementa a fila circular, portanto não use o operador $\rightarrow$ para acessar os elementos da fila diretamente, se precisar de algum elemento use front, enqueue, dequeue, ...).

$\star$ **Exercício 7)** Escreva uma função que recebe uma fila q como entrada e inverte os elementos em uma nova fila r (reversed). Utilize somente as operações enqueue, dequeue e empty, além é claro das funções para criar e destruir uma fila. Por exemplo, se $q = \{1, 2, 3, 4\}$ então $r = \{4, 3, 2, 1\}$. Você pode utilizar uma fila auxiliar t com o mesmo tamanho de q como espaço temporário para auxiliar nessa tarefa. Utilize o seguinte protótipo para a sua função:

```
Queue* reverse (Queue *q);
```

Exercício 8) Problema de '**Josephus**': a história conta que na revolta judaica contra Roma, Josephus e mais trinta e nove de seus companheiros resistiram aos romanos em uma caverna. Com a derrota iminente, eles decidiram que prefeririam morrer a serem escravos dos romanos. Eles se organizaram em um círculo. Um homem foi designado como número um, outro como dois, e assim por diante, e percorrendo o círculo no sentido horário e contando eles matavam todo o sétimo homem (após a morte a contagem reiniciava no próximo homem após aquele que foi morto). Josephus, de acordo com a lenda, era entre outras coisas um matemático realizado. Ele instantaneamente descobriu onde deveria sentar-se para ser o último a ser executado. Quando chegou a hora, em vez de matar-se, ele se juntou ao lado romano. Você pode encontrar muitas versões diferentes desta história. Algumas versões matam cada terceiro homem e outras permitem que o último homem fuja em um cavalo. Em qualquer caso, a ideia é a mesma. Existe uma brincadeira infantil conhecida como '**batata quente**' que segue a mesma lógica do conto acima.

Supondo que $n = 5$ (cinco pessoas) e $m = 3$ (passo da morte), então a sequência de eliminações é dado por: 3, 1, 5, 2, 4 — tal que o último elemento da sequência (4) é o sobrevivente — pois

```
 1   2   3   4   5
 1   2  *3   4   5  (1 iteração - elimina o 3)
*1   2  *3   4   5  (2 iteração - elimina o 1)
*1   2  *3   4  *5  (3 iteração - elimina o 5)
*1  *2  *3   4  *5  (4 iteração - elimina o 2)
*1  *2  *3  *4  *5  (5 iteração - elimina o 4)
```

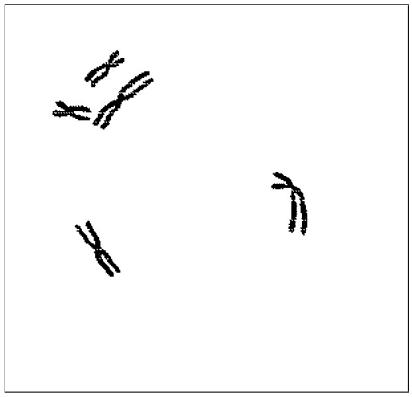
Para resolver este exercício utilize as operações definidas para a interface da fila circular (create, enqueue, dequeue, empty, front, ...). **Não** acesse os elementos armazenados na fila senão pelas operações (funções) definidas pela interface (ou seja, suponha que você não tenha acesso ao código fonte que implementa a fila circular, portanto não use o operador $\rightarrow$ para acessar os elementos da fila diretamente, se precisar de algum elemento use front, enqueue, dequeue, ...).

Exercício 9) Suponha uma matriz tal que porções de terra são representadas pelo símbolo '*' e regiões de água pelo símbolo de vazio ' '. Implemente um algoritmo baseado na estrutura de dados **fila** que conta o número de ilhas nesse mapa. Por exemplo, a matriz abaixo (esquerda) tem tamanho 10×10 e possui no total cinco ilhas enumeradas conforme o mapa da direita. Seu algoritmo deve receber um mapa conforme representado na matriz da esquerda e retornar o número de ilhas. Porções de terra conectadas na diagonal também fazem parte da mesma massa.

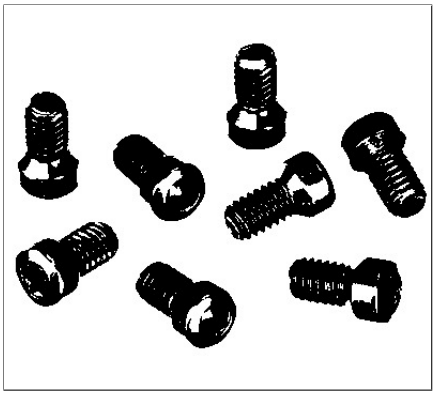
```
  0 1 2 3 4 5 6 7 8 9
0 *   *       * * * *       1   2       3 3 3 3
1     *   *   *                 2   2   3
2 * * * *     *               2 2 2 2     3
3 *     *                     2     2
4 * * * *       * * *         2 2 2 2       4 4 4
5   *   *     * * * *           2   2     4 4 4 4
6           * * *                       4 4 4
7       *     * * *                 5     4 4 4
8 *   *   *     *                   5   5   5     4
9 * * * *       * * *               5 5 5 5       4 4 4
```

```
Número de ilhas: 5
```

Algoritmos para resolver o problema acima tem diversas aplicações. A extração de componentes conexos em imagens é um problema similar. Ela é utilizada em aplicações industriais, reconhecimento de documentos, etc, tal que o objetivo é determinar o número de objetos presentes em uma imagem. Considere como exemplo as seguintes imagens



```
Número de objetos: 5
```



```
Número de objetos: 8
```

Se tiver interesse em resolver o problema com o uso de imagens utilize o programa **imagens.c** para auxiliar, senão utilize o programa **islands.c** (dentro de **arquivos.zip**).

Dica: uma estratégia é percorrer a matriz até achar um pixel ou posição da matriz que seja terra (ou objeto) então enfileire essa posição, e enquanto a fila não for vazia, desenfileire essa posição e verifique os elementos vizinhos (valores na diagonal, horizontal e vertical com distância de uma posição da coordenada desenfileirada), se eles também forem de objetos coloque-os na fila, e continue o processo. Quando a fila se tornar vazia é esperado que uma componente inteira esteja conquistada. O próximo passo é continuar a percorrer a matriz atrás de regiões que não foram exploradas.

Utilize uma fila circular e operações definidas pela interface para resolver este problema.
Determine também a complexidade da sua solução em notação $\mathcal{O}$.

Exercício 10) Em sistemas operacionais que usam paginação para gerenciamento de memória, o algoritmo de substituição de página é necessário para decidir qual página precisa ser substituída quando uma nova página chega. Sempre que uma nova página é referenciada e não está presente na memória, ocorre uma falha ou falta de página (**page fault**) e o sistema operacional substitui uma das páginas existentes pela nova página necessária. Diferentes algoritmos de substituição de página sugerem diferentes maneiras de decidir qual página substituir. O objetivo de todos os algoritmos é reduzir o número de falhas de página.

Algoritmo First-In-First-Out (FIFO): este é o algoritmo de substituição de página mais simples; nele, o sistema operacional mantém o controle de todas as páginas na memória em uma fila, a página mais antiga está na frente da fila. Quando uma página precisa ser substituída, a página na frente da fila é selecionada para remoção.

Por exemplo, considere as seguintes requisições por páginas $\{1, 3, 0, 3, 5, 6\}$ e suponha uma cache com três slots de tamanho, ou seja, não mais que três páginas podem ser mantidas na memória. Inicialmente todos os slots estão vazios, então quando as requisições pelas páginas 1, 3 e 0 chegam elas são alocadas para os slots vazios o que resulta em três falhas de páginas. No entanto, quando a página 3 é requisitada novamente ela ainda está na memória e portanto não é gerada uma falta de página (**hit**). Logo a requisição pela página 5 chega, e como ela não está disponível na memória, uma falta de página é gerada e a página 5 substitui a página no slot mais antigo, que armazena a página 1. Quando a última requisição chega, página 6, é novamente gerada uma falta de página e a página no slot mais antigo é removida (página 3). No total são geradas **cinco falta de páginas** (retorno da função).

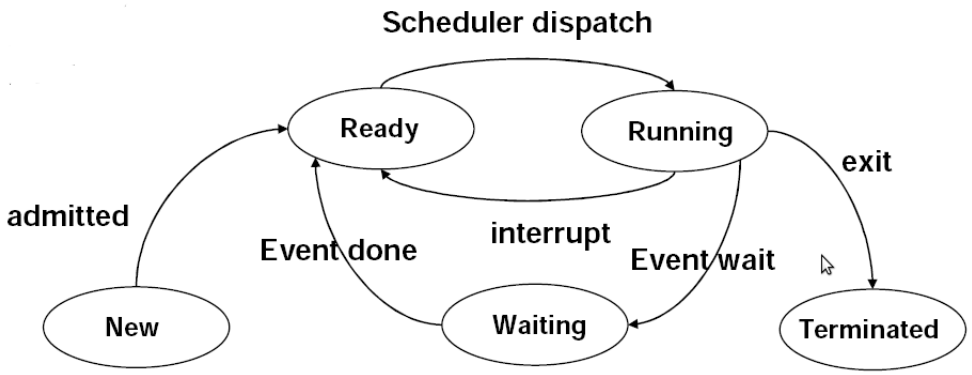
Escreva uma função tal que dado **nresq** requisições de acesso a páginas, armazenadas em **requests**[], e uma cache com tamanho **nslots**, retorna o número total de falta de páginas para o conjunto de entrada. Utilize o seguinte protótipo para a sua função:

```
int pfault (int requests[], int nreq, int nslots);
```

Utilize uma fila circular e operações definidas pela interface para resolver este problema.

Exercício 11) Quando um computador é multiprogramado, ele muitas vezes tem variados *processos* (programas em execução) que competem pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais processos estão simultaneamente no estado pronto para execução. Se somente uma CPU se encontrar disponível, deverá ser feita uma escolha de qual processo executará em seguida. A parte do sistema operacional que faz a escolha é chamada de *escalonador*, e o algoritmo que ele usa é o *algoritmo de escalonamento*. As vantagens do escalonamento são: aumento da taxa de utilização da CPU, melhor utilização dos recursos e redução do tempo de execução de um conjunto de programas. O ciclo de vida de um processo é ilustrado na figura abaixo.

O algoritmo de escalonameto **round-robin** é um algoritmo simples, antigo e justo para escalonamento de processos em um sistema operacional. Este algoritmo foi projetado especialmente para



sistemas de tempo compartilhado, pois ele depende de um temporizador. Este algoritmo é imune a problemas de **starvation** (quando um processo nunca é executado).

O funcionamento deste algoritmo acontece da seguinte forma: a cada processo é atribuído um intervalo de tempo, o seu *quantum* (assuma que seja igual para todos os processos), definido pelo sistema operacional (**SO**) e que determina o período de tempo entre cada sinal de interrupção. Se, ao final do quantum o processo estiver ainda executando, a CPU sofrerá uma interrupção (preempção) e será dada a outro processo. Se o processo terminou antes que o quantum tenha decorrido, a CPU é alternada para outro processo.

Projete um algoritmo baseado no tipo estruturado fila para que dado um certo valor de quantum e uma lista de n processos, cada processo definido por um id (identifier) e pela quantidade de processamento de CPU necessária para execução, determina a ordem que eles são terminados. Considere como exemplo a seguinte execução:

```
./round-robin [quantum] [n] [id1] [cpu1] [id2] [cpu2] ...

./round-robin 20.0 4 1 53.0 2 17.0 3 68.0 4 24.0
Processo 2, finalizado em 37.0 ms.
Processo 4, finalizado em 121.0 ms.
Processo 1, finalizado em 134.0 ms.
Processo 3, finalizado em 162.0 ms.
```

Utilize o seguinte protótipo para a sua função:

```
void round_robin (Queue *q, double quantum, int n);
```

Utilize uma fila circular e operações definidas pela interface para resolver este problema.