



UNIVERSIDADE DE FORTALEZA
CURSO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA: PROGRAMAÇÃO FUNCIONAL

INTEGRANTES DA EQUIPE:

Bárbara Bezerra Alves – 2317806
Leandro de Souza Cunha - 2313608
Pedro Lucas Freitas Campos - 2318049
Rênio Pereira Braga Queiroz - 2313593

FORTALEZA-CE
MARÇO/2025

LINK DO REPOSITÓRIO

<https://github.com/Pedrolucasfcampos/fila-de-espera---progfuncional.git>

RELATÓRIO

DESCRIÇÃO GERAL DO SISTEMA

O sistema é um gerenciador de fila de pacientes que permite a adição de novos pacientes, a remoção de pacientes quando consultados e encerrar o programa. Utiliza uma interface simples baseada em prompt e alert. Além disso, a aplicação também contém um contador de pacientes atendidos durante a sessão.

PAPÉIS

Pedro Lucas Freitas Campos - Responsável pela implementação do código.

Rênio Pereira Braga Queiroz - Responsável pela implementação do código.

Leandro de Souza Cunha - Responsável pelo relatório.

Bárbara Bezerra Alves - Responsável pelos testes e revisão do sistema.

REQUISITOS DO SISTEMA

Funcionais:

RF01: O sistema deve permitir a adição de novos pacientes à fila de espera - Código:

```
const adicionarPaciente = (pacientes, nome)
```

RF02: Caso uma opção inválida seja selecionada o sistema deve avisar ao usuário e permitir novas tentativas - Código: default: {

```
    alert("Opção inválida. Tente novamente.")
```

```
    return main(pacientes, contarPaciente)
```

RF03: O sistema deve permitir a consulta dos pacientes da fila - Código: const
consultarPaciente = (pacientes, contarPaciente)

RF04: O sistema deve ter um contador de pacientes atendidos durante aquela sessão
- Código: const criarContadorPacientes

RF05: O sistema deve exibir um menu com opções - Código: const opcao =
parseInt(prompt(criarMenu(pacientes)))

RF06: O sistema deve permitir que o usuário encerre aquela sessão a qualquer momento - Código: case 3: {

```
alert("Encerrando o programa.")
```

```
Return
```

Não Funcionais:

RNF01: O sistema deverá utilizar em seu código-fonte ao menos uma função lambda, ao menos uma list comprehension, ao menos uma closure em alguma função e uma função de alta ordem - A parte de código desse requisito não funcional estará mais abaixo na sessão dos “conceitos de programação funcional”.

RNF02: O código deve ser compilável e implementado em javascript. Além de ser executável em um navegador - Código: todo o código é compilável e o usuário pode interagir com ele através das funções “prompt” e “alert”.

RNF03: O código visa alcançar a imutabilidade na lista, não alterando o valor da lista, mas criando quando necessário - Código: adicionarPaciente e consultarPaciente.

CONCEITOS DE PROGRAMAÇÃO FUNCIONAL

Função Lambda:

```
const criarContadorPacientes = () => { let contador = 0; return () => ++contador; };
```

List Comprehension:

```
const listaPacientes = pacientes.map((paciente, index) => index + 1 + " " + paciente)
```

Closure:

“criarContadorPacientes” encapsula a variável “contador” dentro da função retornada.

Função de Alta Ordem:

“consultarPaciente” recebe “contarPaciente” como argumento e a chama dentro da função.

PROMPTS E RESPOSTAS DO CHATBOT(CHATGPT)

(o chatbot foi utilizado como auxílio para a verificação e validação extra dos conceitos)

1 - PERGUNTA:

```
const criarContadorPacientes = () => { let contador = 0; return () => ++contador; }; const  
criarMenu = (pacientes) => { const listaPacientes = pacientes .map((paciente, index) =>  
index + 1 + " " + paciente) .join("\n"); return Fila de espera:\n${ listaPacientes ?  
listaPacientes + "\n" : "" }\nEscolha uma das opções pelo número:\n1. Novo paciente\n2.  
Consultar paciente\n3. Sair\n; }; const adicionarPaciente = (pacientes, nome) =>
```

```
[...pacientes, nome]; const consultarPaciente = (pacientes, contarPaciente) => { if
(pacientes.length === 0) { return { pacientes, mensagem: "Não há pacientes na fila." }; }
const [pacienteConsultado, ...restante] = pacientes; const totalAtendidos =
contarPaciente(); return { pacientes: restante, mensagem: "Paciente consultado: " +
pacienteConsultado + "\n\nTotal atendidos: " + totalAtendidos }; }; const main =
(pacientes = [], contarPaciente) => { const opcao =
parseInt(prompt(criarMenu(pacientes))); switch (opcao) { case 1: { const nome =
prompt("Digite o nome do novo paciente:"); return main(adicionarPaciente(pacientes,
nome), contarPaciente); } case 2: { const { pacientes: novosPacientes, mensagem } =
consultarPaciente(pacientes, contarPaciente); alert(mensagem); return
main(novosPacientes, contarPaciente); } case 3: { alert("Encerrando o programa.");
return; } default: { alert("Opção inválida. Tente novamente."); return main(pacientes,
contarPaciente); } } }; const contarPaciente = criarContadorPacientes(); main([],
contarPaciente); Os requisitos funcionais do código acima são : RF01: O sistema deve
permitir a adição de novos pacientes à fila de espera RF02: Caso uma opção inválida seja
selecionada o sistema deve avisar ao usuário e permitir novas tentativas RF03: O sistema
deve permitir a consulta dos pacientes da fila RF04: O sistema deve ter um contador de
pacientes atendidos durante aquela sessão RF05: O sistema deve exibir um menu com
opções RF06: O sistema deve permitir que o usuário encerre aquela sessão a qualquer
momento E os não funcionais são: RNF01: O sistema deverá utilizar em seu código-fonte
ao menos uma função lambda, ao menos uma list comprehension, ao menos uma closure
em alguma função e uma função de alta ordem RNF02: O código deve ser compilável e
implementado em javascript. Além de ser executável em um navegador RNF03: O código
visa alcançar a imutabilidade na lista, não alterando o valor da lista, mas criando quando
necessário realize para mim a identificação de onde esses requisitos estão codificados
```

RESPOSTA:

Aqui está a identificação dos requisitos no código fornecido:

Requisitos Funcionais (RF)

- **RF01 - O sistema deve permitir a adição de novos pacientes à fila de espera**
 - Implementado na função adicionarPaciente, chamada na opção 1 do switch na main().
- **RF02 - Caso uma opção inválida seja selecionada o sistema deve avisar ao usuário e permitir novas tentativas**
 - Implementado no default do switch dentro da main(), onde um alert informa sobre a opção inválida e a função é chamada novamente.
- **RF03 - O sistema deve permitir a consulta dos pacientes da fila**
 - Implementado na função consultarPaciente, chamada na opção 2 do switch dentro da main().
- **RF04 - O sistema deve ter um contador de pacientes atendidos durante aquela sessão**

- Implementado com a função `criarContadorPacientes`, que usa uma closure para manter o estado do contador entre chamadas. O contador é atualizado na `consultarPaciente()`.
- **RF05 - O sistema deve exibir um menu com opções**
 - Implementado na função `criarMenu()`, chamada na `main()`.
- **RF06 - O sistema deve permitir que o usuário encerre aquela sessão a qualquer momento**
 - Implementado na opção 3 do switch, onde um alert é exibido antes de encerrar a execução.

Requisitos Não Funcionais (RNF)

- **RNF01 - O sistema deverá utilizar em seu código-fonte ao menos uma função lambda, ao menos uma list comprehension, ao menos uma closure em alguma função e uma função de alta ordem**
 - **Função lambda:** O código não usa explicitamente uma função lambda ((param) => expressão), mas pode ser ajustado para incluir uma.
 - **List comprehension:** Implementado na `criarMenu()`, onde `map()` é usado para formatar os pacientes.
 - **Closure:** Implementado na `criarContadorPacientes()`, que mantém o estado do contador entre chamadas.
 - **Função de alta ordem:** `adicionarPaciente()` e `consultarPaciente()` retornam novos estados da lista em vez de modificá-la diretamente.
- **RNF02 - O código deve ser compilável e implementado em JavaScript, além de ser executável em um navegador**
 - O código usa `prompt` e `alert`, que são funções padrão do navegador, garantindo sua execução sem compilação.
- **RNF03 - O código visa alcançar a imutabilidade na lista, não alterando o valor da lista, mas criando quando necessário**
 - `adicionarPaciente()` e `consultarPaciente()` usam o operador spread `[...]` para criar novas listas sem modificar a original.

Se precisar de mais detalhes ou ajustes, me avise! 🚀

2 - PERGUNTA:

```
const criarContadorPacientes = () => { let contador = 0; return () => ++contador; };
const criarMenu = (pacientes) => { const listaPacientes = pacientes .map((paciente, index) =>
index + 1 + " " + paciente) .join("\n"); return `Fila de espera:\n${ listaPacientes ?
listaPacientes + "\n" : "" }\nEscolha uma das opções pelo número:\n1. Novo paciente\n2.
Consultar paciente\n3. Sair\n`; };
const adicionarPaciente = (pacientes, nome) => [...pacientes, nome];
const consultarPaciente = (pacientes, contarPaciente) => { if
```

```
(pacientes.length === 0) { return { pacientes, mensagem: "Não há pacientes na fila." }; }
const [pacienteConsultado, ...restante] = pacientes; const totalAtendidos =
contarPaciente(); return { pacientes: restante, mensagem: "Paciente consultado: " +
pacienteConsultado + "\n\nTotal atendidos: " + totalAtendidos }; }; const main =
(pacientes = [], contarPaciente) => { const opcao =
parseInt(prompt(criarMenu(pacientes))); switch (opcao) { case 1: { const nome =
prompt("Digite o nome do novo paciente:"); return main(adicionarPaciente(pacientes,
nome), contarPaciente); } case 2: { const { pacientes: novosPacientes, mensagem } =
consultarPaciente(pacientes, contarPaciente); alert(mensagem); return
main(novosPacientes, contarPaciente); } case 3: { alert("Encerrando o programa.");
return; } default: { alert("Opção inválida. Tente novamente."); return main(pacientes,
contarPaciente); } } }; const contarPaciente = criarContadorPacientes(); main([],
contarPaciente);
```

O código acima cumpre os requisitos do trabalho abaixo? • Criar código implementado e compilável • Utilização no código-fonte dos conceitos de Programação Funcional (indicando, no documento de requisitos, onde estas construções funcionais estão sendo usadas): o Criar ao menos uma função lambda; o Criar ao menos uma list comprehension; o Criar ao menos uma closure em alguma função; o Criar ao menos uma função de alta ordem.

RESPOSTA:

Vamos analisar seu código em relação aos requisitos do trabalho:

✓ Código implementado e compilável

Seu código é funcional e utilizável em um ambiente que suporte prompt e alert, como navegadores.

✓ Uso de conceitos de Programação Funcional

1. Função lambda (Arrow Function)

a. O código usa arrow functions, que são equivalentes a funções lambda em JavaScript:

js

CopiarEditar

```
const criarContadorPacientes = () => { let contador = 0; return () => ++contador; };
```

Aqui, a função criarContadorPacientes retorna outra função (closure), que incrementa contador.

2. List comprehension (Map e Join)

a. No JavaScript, o conceito de list comprehension pode ser simulado com map() e join(). Seu código usa isso na criação do menu:

js

CopiarEditar

```
const criarMenu = (pacientes) => {  
  const listaPacientes = pacientes  
    .map((paciente, index) => index + 1 + "º " + paciente)  
    .join("\n");
```

Esse trecho transforma a lista de pacientes em uma string formatada.

3. Closure

a. A função `criarContadorPacientes` é um exemplo de closure, pois mantém o estado **de contador mesmo após a execução inicial**:

js

CopiarEditar

```
const criarContadorPacientes = () => {  
  let contador = 0;  
  return () => ++contador;  
};
```

A variável `contador` não é acessível fora da função, mas sua instância é preservada dentro da função retornada.

4. Função de alta ordem

a. O código usa funções de alta ordem (funções que retornam outras funções ou as recebem como argumentos). Um exemplo é `consultarPaciente`, que recebe `contarPaciente` como argumento:

js

CopiarEditar

```
const consultarPaciente = (pacientes, contarPaciente) => {  
  if (pacientes.length === 0) {  
    return { pacientes, mensagem: "Não há pacientes na fila." };  
  }  
  const [pacienteConsultado, ...restante] = pacientes;  
  const totalAtendidos = contarPaciente();
```

```
return {  
    pacientes: restante,  
    mensagem: "Paciente consultado: " + pacienteConsultado + "\n\nTotal atendidos: " +  
totalAtendidos  
};  
};
```

Aqui, consultarPaciente usa contarPaciente(), que é uma função passada como argumento.

Conclusão

Seu código atende aos requisitos do trabalho! 🎉

Ele é compilável e faz uso dos conceitos exigidos de programação funcional.