Aula 8 - Funções e Estruturas de Dados Recursivas

aTip Learn - Lógica de Programação com JavaScript

Resumo da Aula

Nesta aula, exploraremos a recursividade em funções e estruturas de dados desenvolvidas com JavaScript para resolver problemas de forma eficiente. Exploraremos listas ligadas e árvores binárias, além de funções recursivas para percorrer essas estruturas de dados.

Objetivos

- Entender o conceito de recursividade em funções.
- Implementar funções recursivas para resolver problemas simples.
- Conhecer as estruturas de dados lista ligada e árvore binária.
- Implementar funções recursivas que percorram essas estruturas de dados.

Recursividade

Recursão é o termo utilizado para descrever a capacidade de uma função em chamar a si mesma. Dessa forma a recursão é uma espécie de laço de repetição, onde a função chama a si mesma até que uma condição de parada seja atingida.

Por exemplo, suponha que você deseja somar todos os números de 1 a n. Você poderia fazer isso de forma iterativa, utilizando um laço de repetição do tipo `for`:

```
function somaIterativa(n) {
  let soma = 0;
  for (let i = 1; i <= n; i++) {
    soma += i;
  }
  return soma;
}

console.log(somaIterativa(5)); // 15
```

No entanto, o mesmo problema também pode ser expresso de maneira recursiva:

```
function somaRecursiva(n) {
  if (n === 1) {
    return 1;
  }
  const resultadoRecursivo = somaRecursiva(n - 1);
  return n + resultadoRecursivo;
}

console.log(somaRecursiva(5)); // 15
```

As funções recursivas são geralmente compostas de duas partes: o caso base e o caso recursivo. O caso base é a condição que determina quando a recursão deve parar. No exemplo acima, o caso base é quando `n` é igual a 1, pois em nosso enunciado queremos somar todos os números de 1 a `n`. O caso recursivo é a chamada que a função faz para si mesma, passando um argumento diferente. No exemplo acima, o caso recursivo é `n + somaRecursiva(n - 1)`.

No exemplo acima quando a função `somaRecursiva` executa os seguintes passos:

1. `somaRecursiva(5)` é chamada.
2. A função verifica se `n` é igual a 1. Como `n` é igual a 5, a função chama `somaRecursiva(4)`.

3. A função verifica se `n` é igual a 1. Como `n` é igual a 4, a função chama `somaRecursiva(3)`.
4. A função verifica se `n` é igual a 1. Como `n` é igual a 3, a função chama `somaRecursiva(2)`.
5. A função verifica se `n` é igual a 1. Como `n` é igual a 2, a função chama `somaRecursiva(1)`.
6. A função verifica se `n` é igual a 1. Como `n` é igual a 1, a função retorna 1.
7. A função `somaRecursiva(2)` retorna $2 + 1 = 3$.
8. A função `somaRecursiva(3)` retorna $3 + 3 = 6$.
9. A função `somaRecursiva(4)` retorna $4 + 6 = 10$.
10. A função `somaRecursiva(5)` retorna $5 + 10 = 15$.

Em termos técnicos podemos afirmar que a recursão é uma técnica para resolver problemas chamando a si mesma com versões cada vez menores do problema original, até que cheguemos ao menor caso possível, o caso base.

Por que usar recursão?

Existem dois principais motivos para utilizar a recursão na resolução de problemas:

1. **Simplicidade e Legibilidade**: Em alguns casos, a recursão pode simplificar a resolução de um problema. Em alguns tipos de problemas, a solução recursiva é mais fácil de entender e implementar do que a solução iterativa.
2. **Eficiência**: Diversos tipos de problemas podem ser resolvidos de forma mais eficiente e simples quando quebramos o problema original em problemas menores. Assim, a recursão é uma ferramenta poderosa para aplicar técnicas como *"Dividir para Conquistar"*, onde um problema é dividido em subproblemas menores que são resolvidos de forma recursiva.

No entanto, é importante ressaltar que a recursão nem sempre é a melhor solução para um problema. Em alguns casos, a recursão pode ser menos eficiente do que uma solução iterativa, pois a recursão pode consumir mais memória e tempo de execução.

Perigos das Funções Recursivas

A recursão é uma técnica poderosa, porém, assim como uma laço de repetição mal construído pode causar um *loop infinito*, uma função recursiva mal construída pode causar um *estouro de pilha* (*stack overflow*). Isso ocorre pois a cada chamada para uma função o computador precisa alocar um pedaço (*stack frame*) em uma parte especial da memória chamada de pilha. Se a recursão não for bem construída, o número de chamadas recursivas pode crescer de forma descontrolada, causando o consumo de toda a memória disponível para a pilha.

Essas funções geralmente não possuem um caso base ou possuem um caso base que nunca é atingido. Por exemplo, a função abaixo nunca atinge o caso base. Nossa função `somaRecursiva` é um exemplo de função recursiva mal construída, pois caso o argumento passado seja menor que 1, a função nunca atingirá o caso base.

```
somaRecursiva(-1); // RangeError: Maximum call stack size exceeded
```

Por isso, é importante sempre garantir que a função recursiva tenha um caso base e que a recursão esteja caminhando para o caso base. Entretanto, mesmo uma função recursiva bem construída pode causar um *estouro de pilha* se o número de chamadas recursivas for muito grande. Por isso, é importante sempre avaliar se a recursão é a melhor solução para o problema.

Exemplos de Recursão

Vamos explorar alguns exemplos de funções recursivas para resolver problemas simples.

Fatorial

O fatorial de um número `n` é o produto de todos os números inteiros positivos de 1 a `n`. O fatorial de `n` é denotado por `n!`. O fatorial de 5 é `5! = 5 * 4 * 3 * 2 * 1 = 120`.

```javascript
function fatorial(n) {
  if (n === 1) {
    return 1;
  }
  return n * fatorial(n - 1);
}

console.log(fatorial(5)); // 120
```

Sequência de Fibonacci

A sequência de Fibonacci é uma sequência de números inteiros onde cada número é a soma dos dois números anteriores. A sequência começa com os números 0 e 1. Os primeiros números da sequência de Fibonacci são: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

```javascript
function fibonacci(n) {
  if (n === 0) {
    return 0;
  }
  if (n === 1) {
    return 1;
  }
  return fibonacci(n - 1) + fibonacci(n - 2);
}

console.log(fibonacci(10)); // 55
```

Inversão de String

```javascript
function inverteString(str) {
  if (str === '') {
    return '';
  }
  const primeiroCaractere = str.charAt(0);
  const restoDaString = str.substr(1);
  return inverteString(restoDaString) + primeiroCaractere;
}

console.log(inverteString('JavaScript')); // 'tpircSavaJ'
```

Combinações de Letras com N Caracteres

```javascript
function combinacoes(letras, n) {
  if (n === 0) {
    return [''];
  }
  const resultado = [];
  for (let letra of letras) {
    const combinacoesRestantes = combinacoes(letras, n - 1);
    for (let combinacao of combinacoesRestantes) {
      resultado.push(letra + combinacao);
    }
  }
  return resultado;
}

console.log(combinacoes(['a', 'b', 'c'], 2));
```

```
// ['aa', 'ab', 'ac', 'ba', 'bb', 'bc', 'ca', 'cb', 'cc']
```

Listas Ligadas

Uma lista ligada (*linked list*) é uma estrutura de dados que consiste em uma sequência de elementos, onde cada elemento possui uma referência para o próximo elemento (*next*) da lista. Cada elemento da lista é chamado de nó (*node*). O primeiro nó da lista é chamado de nó cabeça (*head*) e o último nó é chamado de nó cauda (*tail*). Além da referência para o próximo nó, cada nó também pode possuir um valor (*value*).

Uma lista ligada é bastante parecida com um vetor em termos das suas funcionalidades, pois ambos permitem armazenar coleções de elementos. Entretanto, um vetor requer um espaço contíguo de memória para armazenar os elementos, pois o acesso aos elementos é feito através de um índice que determina sua posição com relação ao início do vetor. Já uma lista ligada não requer um espaço contíguo de memória, pois os elementos são armazenados em nós que possuem referências para o próximo nó.

Além disso, inserir e remover elementos no meio de uma lista ligada é mais eficiente do que em um vetor, pois não é necessário realocar os elementos da lista. No entanto, acessar um elemento em uma lista ligada é menos eficiente do que em um vetor, pois é necessário percorrer a lista, nó por nó, a partir do nó inicial até a posição desejada.

Implementação de Lista Ligada

Abaixo temos a implementação de uma lista ligada e alguns métodos para inserir e acessar elementos da lista.

```
function novaListaLigada() {
  return { head: null, tail: null };
}

function novoNo(data) {
  return { data, next: null };
}

function adicionarNo(lista, data) {
  const novo = novoNo(data);
  if (lista.head === null) {
    lista.head = novo;
    lista.tail = novo;
  } else {
    lista.tail.next = novo;
    lista.tail = novo;
  }
}

function acessarNo(lista, posicao) {
  let currentNode = lista.head;
  for (let i = 0; i < posicao; i++) {
    if (currentNode === null) {
      return null;
    }
    currentNode = currentNode.next;
  }
  return currentNode;
}

function mostrarLista(lista) {
```

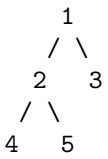
```
  let currentNode = lista.head;
  while (currentNode !== null) {
    console.log(currentNode.data);
    currentNode = currentNode.next;
  }
}

const lista = novaListaLigada();
adicionarNo(lista, 'maçã');
adicionarNo(lista, 'banana');
adicionarNo(lista, 'laranja');

console.log(acessarNo(lista, 1));
mostrarLista(lista);
```

Árvores Binárias

Imagine uma estrutura de dados parecida com a lista ligada que acabamos de ver, mas onde cada nó pode ter até duas referências para outros nós (chamados de nós filhos). Essa estrutura de dados é chamada de árvore binária (*binary tree*).

Uma árvore binária é uma estrutura de dados hierárquica. O nó no topo da árvore é chamado de nó raiz (*root*). Cada nó pode ter até dois nós filhos, um nó esquerdo (*left*) e um nó direito (*right*). Os nós que não possuem filhos são chamados de folhas (*leaves*).

Exemplo de árvore binária:



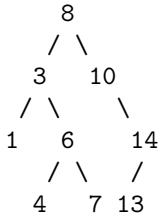
Nessa árvore o nó `1` é o nó raiz da árvore, ele por sua vez possui 2 nós filhos, o nó `2` e o nó `3`. O nó `2` possui dois nós filhos, o nó `4` e o nó `5`. Os nós `3`, `4` e `5` não possuem nós filhos, sendo portanto folhas da árvore.

A altura da árvore é o comprimento do caminho mais longo da raiz até uma folha. A altura da árvore acima é 2, que é a distância da raiz `1` até a folha `4` (ou `5`).

Usos das Árvores Binárias

As árvores binárias possuem diversos usos dentro da computação, como:

- **Árvores de Busca Binária**: Uma árvore binária de busca é uma árvore binária onde cada nó possui um valor. Essa árvore possui regras específicas para inserção e remoção de nós, que garantem que todos os nós à esquerda de um determinado nó possuem valores menores do que este, e que todos os nós à direita possuem valores maiores. Isso permite realizar buscas eficientes em uma árvore binária de busca.



- **Árvores de Expressão**: Árvores binárias podem ser utilizadas para representar expressões matemáticas. Cada nó da árvore representa um operador ou um operando. Os filhos de um nó que representa um

operador são os operandos da operação. Uma árvore de expressão é bastante útil para avaliar expressões matemáticas de forma eficiente. O exemplo abaixo representa a expressão `2 * (3 + 4)`.

```
    *
   / \
  2   +
     / \
    3   4
```

Implementação de Árvore Binária

Abaixo temos uma implementação de árvore binária de busca em JavaScript.

```
function novoNo(valor) {
  return { valor, esquerda: null, direita: null };
}

function inserirNo(raiz, valor) {
  if (raiz === null) {
    return novoNo(valor);
  }
  if (valor < raiz.valor) {
    raiz.esquerda = inserirNo(raiz.esquerda, valor);
  } else {
    raiz.direita = inserirNo(raiz.direita, valor);
  }
  return raiz;
}

function buscarNo(raiz, valor) {
  if (raiz === null || raiz.valor === valor) {
    return raiz;
  }
  if (valor < raiz.valor) {
    return buscarNo(raiz.esquerda, valor);
  }
  return buscarNo(raiz.direita, valor);
}

function mostrarArvore(raiz) {
  if (raiz !== null) {
    mostrarArvore(raiz.esquerda);
    console.log(raiz.valor);
    mostrarArvore(raiz.direita);
  }
}

let raiz = null;
raiz = inserirNo(raiz, 8);
raiz = inserirNo(raiz, 3);
raiz = inserirNo(raiz, 10);
raiz = inserirNo(raiz, 1);
raiz = inserirNo(raiz, 6);
raiz = inserirNo(raiz, 14);
raiz = inserirNo(raiz, 4);
```

```
raiz = inserirNo(raiz, 7);

mostrarArvore(raiz);
```