ESTRUTURAS DE DADOS

Pilha (Lista Encadeada)

Roteiro

- **Estrutura do Nó**
- **Tipo Abstrato de Dados**
- **Detalhes de Implementação**
- **Aplicações da Estrutura**

Estrutura do Nó

Contém os dados e o endereço do sucessor na lista.

```
typedef char ItemType;
/*
 Estrutura usada para guardar
 a informação e o endereço do
 próximo elemento.
*/
struct NodeType
{
  ItemType info;
  NodeType* next;
};
```

Com essa estrutura é possível iniciar o encadeamento.

Note que basta armazenarmos um único ponteiro, aquele que aponta para o início da estrutura.

Como todas as operações ocorrem na cabeça da pilha, conseguiremos efetuar em tempo constante.

Tipo Abstrato de Dados

```
class Stack
{
 public:
  Stack();  // Construtor
  ~Stack(); // Destrutor
  bool isEmpty() const;
  bool isFull() const;
  void print() const;

  void push(ItemType);
  ItemType pop();
 private:
  NodeType* structure;
};
```

Não mudaremos a interface pública

Mudaremos a implementação interna

Detalhes de Implementação

Implementaremos uma pilha como lista encadeada.

O ponteiro structure sempre apontará para o elemento que está no topo da pilha.

Queremos que inserções e remoções ocorram em tempo constante. Em outras palavras, independem do número de elementos na estrutura.

Implementando as ideias da aula passada.

- **Construtor e Destrutor**

```
Stack::Stack(){
  structure = NULL;
}

Stack::~Stack(){
  NodeType* tempPtr;
  while (structure != NULL) {
    tempPtr = structure;
    structure  = structure -> next;
    delete tempPtr;
  }
}
```

Implementando as ideias da aula passada.

- **Verificação de cheio ou vazio.**

```
bool Stack::isEmpty() const {
  return (structure == NULL);
}

bool Stack::isFull() const {
  NodeType* location;
  try {
    location = new NodeType;
    delete location;
    return false;
  } catch(std::bad_alloc exception){
    return true;
  }
}
```

Implementando as ideias da aula passada.

- **Inserindo elementos**

```
void Stack::push(ItemType item){
  if (!isFull()){
    NodeType* location;
    location = new NodeType;
    location->info = item;
    location->next = structure;
    structure = location;
  } else {
    throw "Stack is already full!";
  }
}
```

Implementando as ideias da aula passada.

- **Removendo elementos**

```
ItemType Stack::pop(){
  if (!isEmpty()) {
    NodeType* tempPtr;
    tempPtr = structure;
    ItemType item = structure->info;
    structure         = structure->next;
    delete tempPtr;
    return item;
  } else {
    throw "Stack is empty!";
  }
}
```

Implementando as ideias da aula passada.

- **Imprimindo a pilha na saída padrão**

```
void Stack::print() const
{
  NodeType* tempPtr = structure;
  while (tempPtr != NULL) {
    cout << tempPtr->info;
    tempPtr  = tempPtr->next;
  }
  cout << endl;
}
```

Usar a estrutura se assemelha ao anterior:

```
ItemType character;
Stack stack;
ItemType stackItem;

cout << "Adicione uma String." << endl;
cin.get(character);
while (character != '\n')
  {
    stack.push(character);
    cin.get(character);
  }

while (!stack.isEmpty())
  {
    stackItem = stack.pop();
    cout << stackItem;
  }
```

Aplicações da Estrutura

Uma pilha é uma estrutura bastante útil, principalmente quando precisamos garantir alinhamento de componentes em processos.

- **Chamada de funções na execução de programas.**
- **Análise de sintaxe de linguagens de programação.**
- **Verificação de alinhamento de parênteses em *strings*.**

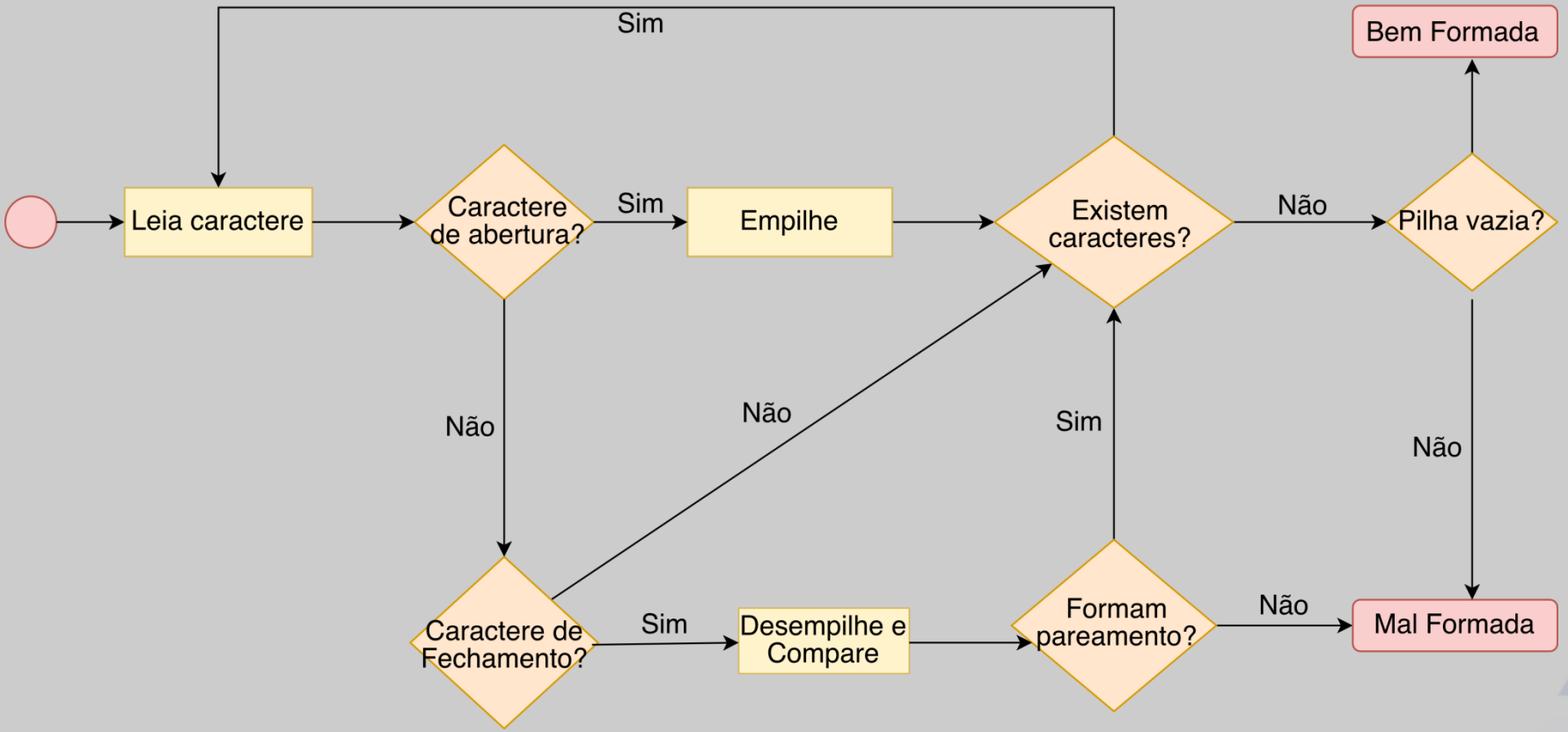
Uma *string* é dita bem formada se:

- **Contém zero ou mais caracteres, o que inclui os caracteres: {, }, (,), [e].**
- **Os caracteres {, (e [devem ser pareados com os caracteres },) e], respectivamente.**

Exemplos:

- **a{b}a Bem formada**
- **a{b[c]b}a Bem formada**
- **a{b}[c]{b}a Bem formada**
- **a{b]a Mal formada**
- **a{b[c}b]a Mal formada**
- **a{b[}c]{b}a Mal formada**

Fluxograma da Aplicação:



```
int main() {
  ItemType character;
  Stack stack;
  ItemType stackItem;

  cout << "Insira uma string." << endl;
  cin.get(character);

  bool isMatched = true;
  while (isMatched && character != '\n')
    {
      if (character == '{' || character== '(' || character==  '['){
        stack.push(character);
      }
      if(character == '}' || character== ')' || character==  ']'){
        if (stack.isEmpty()) {
          isMatched = false;
        } else {
          stackItem = stack.pop();
          isMatched = (
                       (character == '}' && stackItem== '{')
                       || (character== ')' && stackItem == '(')
                       || (character== ']' && stackItem == '[')
                       );
```

```
      if(character == '}' || character== ')' || character==  ']'){
        if (stack.isEmpty()) {
          isMatched = false;
        } else {
          stackItem = stack.pop();
          isMatched = (
                       (character == '}' && stackItem== '{')
                       || (character== ')' && stackItem == '(')
                       || (character== ']' && stackItem == '[')
                       );
        }
      }
      cin.get(character);
    }


  if (isMatched && stack.isEmpty() ) {
    cout << "Bem formada \n";
  } else {
    cout << "Mal formada \n";
  }
}
```

ESTRUTURAS DE DADOS

Pilha (Lista Encadeada)