# ESTRUTURAS DE DADOS

## Pilha (Vetores)

# Roteiro

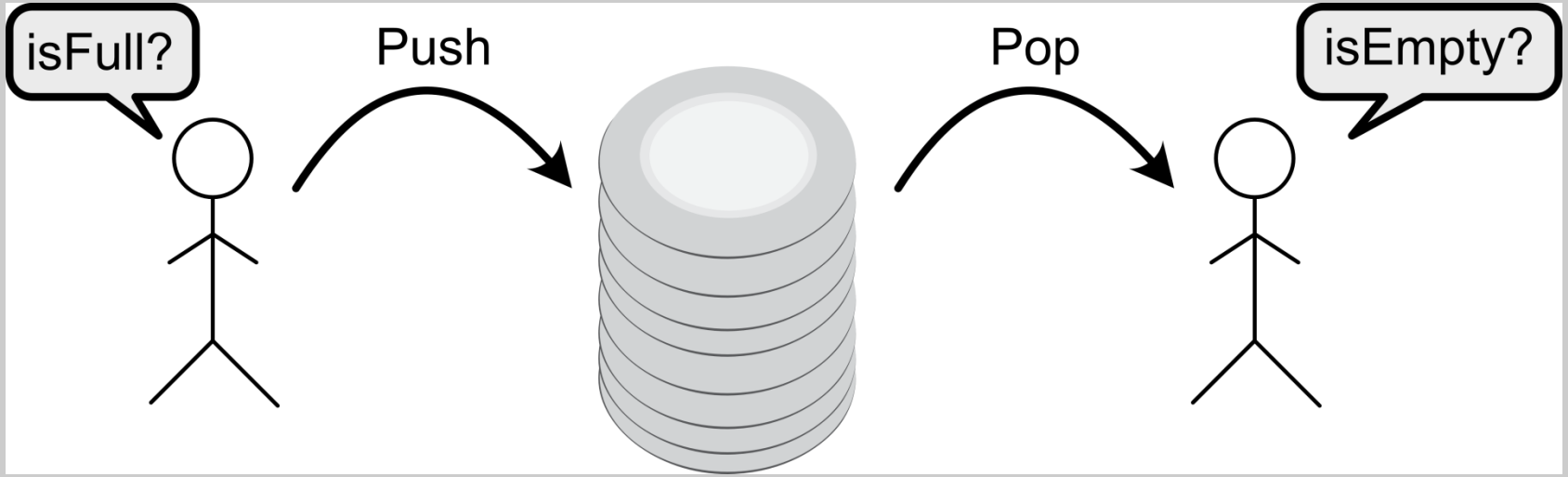
- **Motivação**
- **Tipo Abstrato de Dados**
- **Aplicações da Estrutura**
- **Detalhes de Implementação**

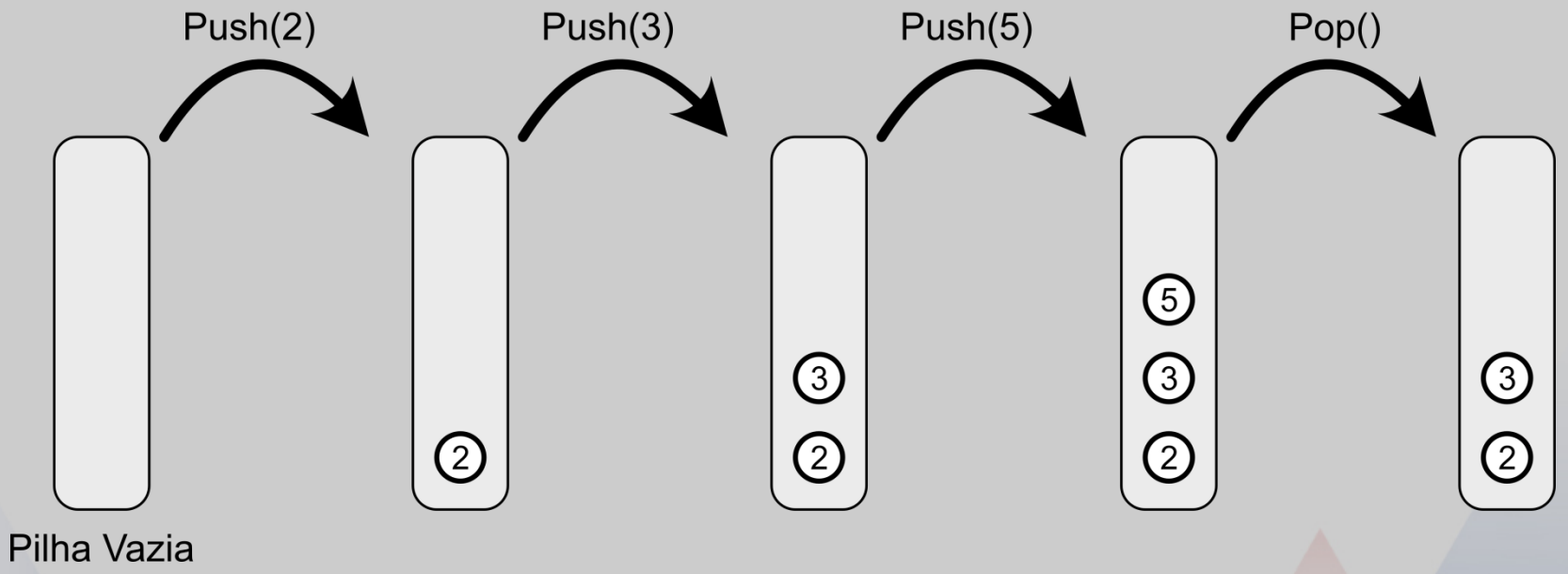
# Motivação

**Uma pilha é uma estrutura linear na qual inserções e remoções ocorrem no topo da pilha.**

**Suponha que duas pessoas estão lavando e enxugando pratos:**

- **Ao terminar de lavar um prato, alguém colocará no topo da pilha de pratos a serem enxugados.**
- **Se já tem prato demais, quem lava fará uma pausa.**
- **Alguém tirará o prato que está no topo da pilha para enxugar e guardar em um lugar adequado.**
- **Se a pilha estiver vazia, quem enxuga faz uma pausa.**


isFull?
Push
Pop
isEmpty?


Push(2)
Push(3)
Push(5)
Pop()
2
3
2
5
3
2
3
2
Pilha Vazia

## Resumindo

- **O primeiro elemento a entrar na estrutura tem que ser o último a sair.**
- **O último elemento a entrar tem que ser o primeiro a sair.**
- **Comportamento parecido com o botão "desfazer" de qualquer editor de texto.**
- **Inserções e remoções ocorrem no topo.**

# 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:
  int length;
  ItemType* structure;
};
```

# 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*.**

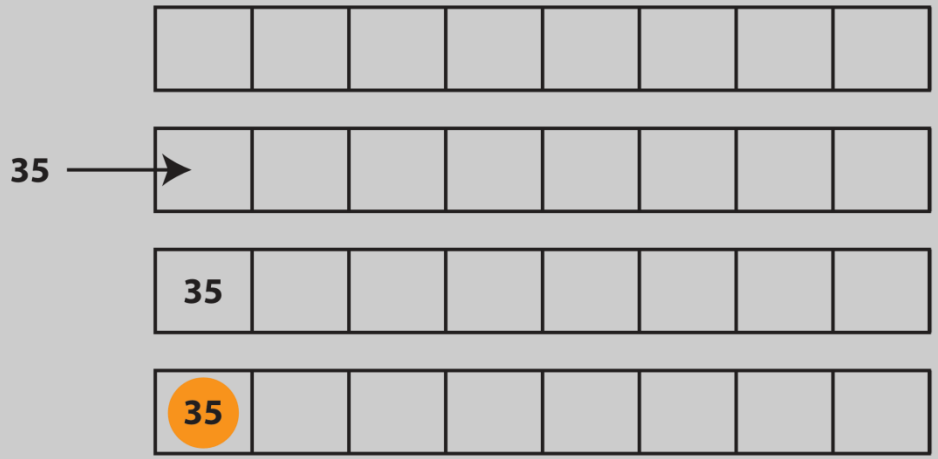
# Detalhes de Implementação

**Implementaremos uma pilha como um vetor.**

**A posição do topo da pilha depende do número de elementos que estão na pilha.**

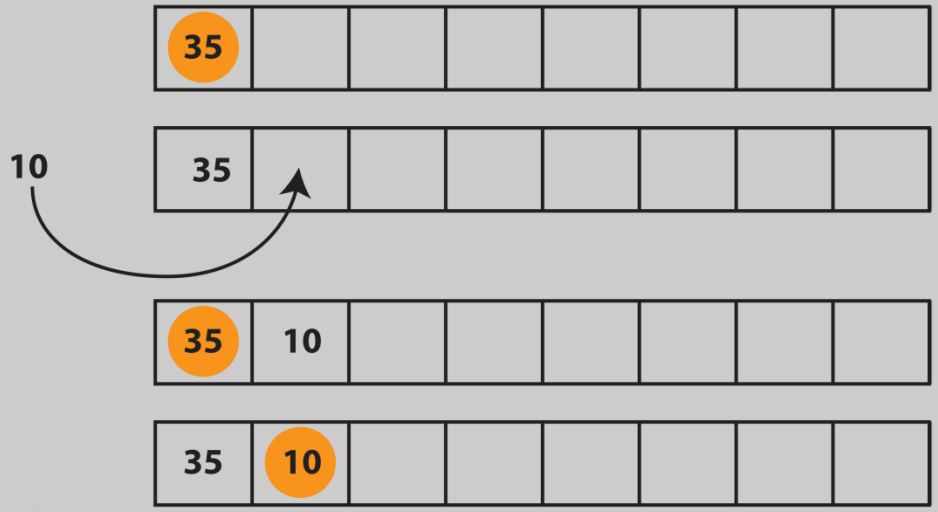
**Queremos que inserções e remoções ocorram em tempo constante. Em outras palavras, independem do número de elementos na estrutura.**

Push: 35




O tamanho do vetor aumentou de 0 para 1. O topo passou a ser o elemento de índice 0.

Push: 10

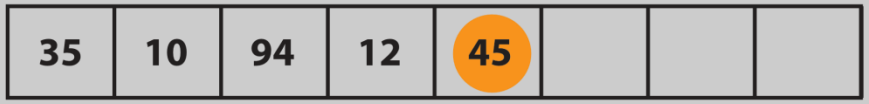



O tamanho do vetor aumentou de 1 para 2. O topo passou a ser o elemento de índice 1.

Push: 35


Push: 10

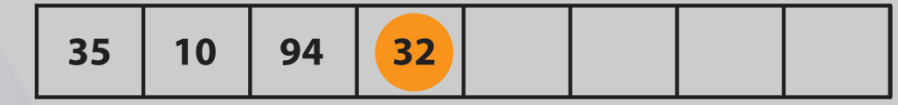

Push: 94, 12, 45




Pop


Pop


Push: 32




**O topo é sempre o último elemento do vetor, podendo ser encontrado na posição length-1.**

## Implementando essa ideia, temos:

- **Construtor e Destrutor**

```
Stack::Stack()
{
  length = 0;
  structure = new ItemType[MAX_ITEMS];
}

Stack::~Stack()
{
  delete [] structure;
}
```

**Implementando essa ideia, temos:**

- **Verificação de cheio ou vazio.**

```
bool  Stack::isEmpty() const
{
  return (length == 0);
}

bool Stack::isFull() const
{
  return (length == MAX_ITEMS);
}
```

**Implementando essa ideia, temos:**

- **Inserindo elementos**

```
void Stack::push(ItemType item)
{
  if (!isFull()){
    structure[length] = item;
    length++;
  } else {
    throw "Stack is already full!";
  }
}
```

## Implementando essa ideia, temos:

- **Removendo elementos**

```
ItemType Stack::pop()
{
  if (!isEmpty()){
    ItemType aux = structure[length - 1];
    length--;
    return aux;
  } else {
    throw "Stack is empty!";
  }
}
```

**Implementando essa ideia, temos:**

- **Imprimindo a pilha na saída padrão**

```
void Stack::print() const
{
  cout << "Pilha = ";
  for (int i = 0; i < length; i++) {
    cout << structure[i];
  }
  cout << endl;
}
```

# Usando a estrutura:

```
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;
  }
```

# ESTRUTURAS DE DADOS

## Pilha (Vetores)