Estruturas de Dados II: Lista, Fila, Pilha e Árvore em Python, Go e Ruby

Augusto Ortigoso Barbosa

6 de fevereiro de 2025

Resumo

As estruturas de dados desempenham um papel essencial na organização e eficiência do armazenamento e manipulação de informações. Neste artigo, são abordadas quatro estruturas fundamentais: Lista, Fila, Pilha e Árvore, detalhando suas características, operações básicas e implementações nas linguagens Python, Go e Ruby. Além disso, são apresentados exemplos práticos utilizando bibliotecas nativas e externas para demonstrar seu uso no desenvolvimento de software.

As estruturas de dados são componentes fundamentais na ciência da computação, fornecendo mecanismos eficientes para armazenar, acessar e processar informações. Este artigo explora quatro estruturas fundamentais e suas respectivas implementações em três linguagens amplamente utilizadas: Python, Go e Ruby.

1 Introdução

As estruturas de dados desempenham um papel fundamental na organização, manipulação e recuperação eficiente de informações nos mais diversos sistemas computacionais. Elas servem como base para o desenvolvimento de algoritmos e impactam diretamente a performance de aplicações, desde sistemas embarcados até bancos de dados distribuídos.

Entre as estruturas mais utilizadas estão **listas, filas, pilhas e árvores**, cada uma com características específicas que as tornam ideais para diferentes tipos de problemas. Por exemplo, listas são amplamente empregadas para armazenamento dinâmico de elementos, filas são essenciais para sistemas de processamento em ordem, pilhas aparecem frequentemente na manipulação de expressões matemáticas e na pilha de chamadas de execução, e árvores estruturam dados de forma hierárquica, sendo indispensáveis em bancos de dados e compiladores.

A escolha da linguagem de programação influencia diretamente a forma como essas estruturas são implementadas e utilizadas. Algumas linguagens oferecem bibliotecas nativas altamente otimizadas, enquanto outras permitem implementações manuais que proporcionam maior controle sobre o gerenciamento de memória e a complexidade das operações. Neste artigo, exploramos a implementação dessas quatro estruturas de dados nas linguagens **Python, Go e Ruby**, destacando as bibliotecas disponíveis, os métodos associados e suas aplicações no desenvolvimento de software.

2 Bibliotecas Disponíveis e Implementações

As linguagens de programação modernas fornecem diversas formas de trabalhar com **listas, filas, pilhas e árvores**, seja por meio de **bibliotecas nativas** ou utilizando **bibliotecas externas** especializadas.

2.1 Python

Python oferece diversas bibliotecas que facilitam o uso dessas estruturas:

- **list** – Representa listas dinâmicas, podendo ser usadas como listas encadeadas e pilhas.

- **collections.deque** – Implementação eficiente de filas e pilhas.
- **queue.Queue** – Útil para implementação de filas com sincronização entre threads.
- **heapq** – Implementação de filas de prioridade.
- **binarytree** – Biblioteca externa para manipular árvores binárias de forma mais intuitiva.

Exemplo de uso com `collections.deque` (fila eficiente):

```python
from collections import deque
fila = deque([1, 2, 3])
fila.append(4)  # Enfileirar
fila.popleft()  # Desenfileirar
print(fila)
```

2.2 Go

Diferente de Python, Go não possui bibliotecas nativas específicas para estruturas como pilhas e filas, exigindo que o programador implemente essas estruturas manualmente ou utilize pacotes externos.

- **Slices (`[]T`)** – Permitem criar listas dinâmicas e pilhas de forma eficiente.
- **Channels (`chan T`)** – São uma alternativa interessante para filas concorrentes.
- **Pacote `container/list`** – Implementação de listas duplamente encadeadas.
- **Pacote `container/heap`** – Fornece suporte a filas de prioridade.

Exemplo de uso da estrutura `container/list` para filas:

```go
package main
import (
    "container/list"
    "fmt"
)

func main() {
    fila := list.New()
    fila.PushBack(1)
    fila.PushBack(2)
    fmt.Println(fila.Remove(fila.Front())) // Remove e imprime 1
}
```

2.3 Ruby

Ruby fornece suporte nativo para listas, filas e pilhas por meio da classe `Array`, que implementa métodos convenientes para manipulação dessas estruturas.

- **Array** – Usado para listas, pilhas e filas.
- **Queue e SizedQueue** – Implementação de filas com suporte a concorrência.
- **RBTree** – Implementação eficiente de árvores binárias balanceadas.

Exemplo com `Queue` para filas concorrentes:

```ruby
require 'thread'
fila = Queue.new
fila.push(1)
fila.push(2)
puts fila.pop  # Remove e imprime 1
```

3 Implementação em Python, Go e Ruby

3.1 Lista

Uso da biblioteca nativa:

```python
# Python
lista = [1, 2, 3, 4]
```

```go
// Go (com container/list)
package main
import (
    "container/list"
    "fmt"
)
func main() {
    lista := list.New()
    lista.PushBack(1)
    lista.PushBack(2)
    for e := lista.Front(); e != nil; e = e.Next() {
        fmt.Println(e.Value)
    }
}
```

```ruby
# Ruby
lista = [1, 2, 3, 4]
```

3.2 Pilha

Uso da estrutura 'list' em Python:

```python
# Python
pilha = []
pilha.append(1)
pilha.append(2)
print(pilha.pop())  # Remove e imprime 2
```

Uso de slices em Go (implementação manual):

```go
// Go (usando slices)
package main
import "fmt"
func main() {
    var pilha []int
    pilha = append(pilha, 1)
    pilha = append(pilha, 2)
    topo := pilha[len(pilha)-1]
    pilha = pilha[:len(pilha)-1]  // Remove o topo
    fmt.Println(topo)  // Imprime 2
}
```

Uso da estrutura 'Array' em Ruby:

```ruby
# Ruby
pilha = []
pilha.push(1)
pilha.push(2)
puts pilha.pop  # Remove e imprime 2
```

3.3 Fila

Uso da biblioteca 'queue' em Python:

```python
# Python
import queue
fila = queue.Queue()
```

```
fila.put(1)
fila.put(2)
print(fila.get())  # Remove e imprime 1
```

Uso da biblioteca 'container/list' em Go:

```
// Go (com container/list)
package main
import (
    "container/list"
    "fmt"
)
func main() {
    fila := list.New()
    fila.PushBack(1)
    fila.PushBack(2)
    fmt.Println(fila.Remove(fila.Front())) // Remove e imprime 1
}
```

Uso da biblioteca 'thread/Queue' em Ruby:

```
# Ruby
require 'thread'
fila = Queue.new
fila.push(1)
fila.push(2)
puts fila.pop  # Remove e imprime 1
```

3.4 Árvore

Uso da biblioteca 'binarytree' em Python:

```
# Python
from binarytree import Node
raiz = Node(10)
raiz.left = Node(5)
raiz.right = Node(15)
print(raiz)
```

Implementação de árvore binária em Go:

```
package main

import "fmt"

type Node struct {
    value int
    left  *Node
    right *Node
}

func (n *Node) insert(value int) {
    if value < n.value {
        if n.left == nil {
            n.left = &Node{value: value}
        } else {
            n.left.insert(value)
        }
    } else {
        if n.right == nil {
            n.right = &Node{value: value}
        } else {
            n.right.insert(value)
        }
    }
}

func (n *Node) print() {
    if n != nil {
```

```
        n.left.print()
        fmt.Println(n.value)
        n.right.print()
    }
}

func main() {
    root := &Node{value: 10}
    root.insert(5)
    root.insert(15)
    root.insert(2)
    root.insert(7)
    root.insert(12)
    root.insert(17)

    root.print() // Imprime a arvore em ordem
}
```

Uso da biblioteca 'rbtree' em Ruby:

```
# Ruby
require 'rbtree'
arvore = RBTree.new
arvore[10] = "Raiz"
arvore[5] = "Esquerda"
arvore[15] = "Direita"
puts arvore.keys
```

4 Conclusão

Foram apresentadas quatro estruturas de dados fundamentais e suas implementações em Python, Go e Ruby, destacando as bibliotecas disponíveis em cada linguagem. Observamos que:

- Python possui um suporte mais abrangente, com bibliotecas nativas robustas como 'queue', 'collections.deque' e 'binarytree'.
- Go oferece 'container/list' para listas encadeadas, mas exige mais implementação manual para árvores e pilhas.
- Ruby possui suporte nativo para listas e pilhas, mas a implementação de árvores depende de bibliotecas externas como 'rbtree'.

Cada linguagem possui abordagens diferentes para o gerenciamento de estruturas de dados, e a escolha da mais adequada depende do contexto de desenvolvimento. Enquanto Python oferece maior simplicidade e suporte nativo, Go enfatiza eficiência e flexibilidade, e Ruby equilibra simplicidade com o uso de bibliotecas externas.