Pilhas, Filas e Deques

Conceitos, complexidade e implementação

Prof. Marcelo de Souza

45RPE – Resolução de Problemas com Estruturas de Dados

Universidade do Estado de Santa Catarina

UDESC

Estruturas sequenciais

Ou seja, TADs sequenciais

Vamos construir tipos abstratos de dados (TADs) sequenciais que usam as estruturas de dados fundamentais (arranjos e listas encadeadas) para a implementação de estruturas mais abstratas e com comportamentos específicos.

Estruturas sequenciais

Ou seja, TADs sequenciais

Vamos construir tipos abstratos de dados (TADs) sequenciais que usam as estruturas de dados fundamentais (arranjos e listas encadeadas) para a implementação de estruturas mais abstratas e com comportamentos específicos.

Em particular, implementaremos:

- **Pilhas** (*stacks*);
- **Filas** (*queues*);
- **Deques** (*double-ended queues*).

Estruturas sequenciais

Ou seja, TADs sequenciais

Vamos construir tipos abstratos de dados (TADs) sequenciais que usam as estruturas de dados fundamentais (arranjos e listas encadeadas) para a implementação de estruturas mais abstratas e com comportamentos específicos.

Em particular, implementaremos:

- **Pilhas** (*stacks*);
- **Filas** (*queues*);
- **Deques** (*double-ended queues*).

Essas estruturas se diferenciam pela **política de inserção e remoção dos dados** adotada. Ou seja, somente serão permitidas as operações de interesse.

Pilhas

Ideia geral

Uma **pilha** implementa a política **LIFO** – *last in, first out*. Ou seja, o último elemento adicionado à estrutura é o primeiro a ser removido.

Pilhas

Ideia geral

Uma **pilha** implementa a política **LIFO** – *last in, first out*. Ou seja, o último elemento adicionado à estrutura é o primeiro a ser removido.

A pilha simula uma "pilha de objetos", logo ela tem um **topo**.

Pilhas

Ideia geral

Uma **pilha** implementa a política **LIFO** – *last in, first out*. Ou seja, o último elemento adicionado à estrutura é o primeiro a ser removido.

A pilha simula uma "pilha de objetos", logo ela tem um **topo**.

Operações:

- `push`: insere um elemento no topo da pilha.
- `pop`: remove (e retorna) o elemento do topo da pilha.
- `top`: retorna o elemento do topo da pilha (sem remover).

Pilhas

Ideia geral

Uma **pilha** implementa a política **LIFO** – *last in, first out*. Ou seja, o último elemento adicionado à estrutura é o primeiro a ser removido.

A pilha simula uma "pilha de objetos", logo ela tem um **topo**.

Operações:

- `push`: insere um elemento no topo da pilha.
- `pop`: remove (e retorna) o elemento do topo da pilha.
- `top`: retorna o elemento do topo da pilha (sem remover).

Aplicações:

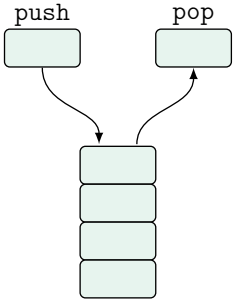
- Histórico de acesso de um navegador (voltar, avançar).
- Operações em um editor de texto (desfazer, refazer).

Pilhas

Exemplo de funcionamento

Seja uma pilha implementada usando uma lista (sequencial), onde o topo é o final da lista.

Método	Retorno	Conteúdo da pilha
`push(5)`	–	(5)
`push(3)`	–	(5, 3)
`size()`	2	(5, 3)
`pop()`	3	(5)
`is_empty()`	false	(5)
`pop()`	5	()
`is_empty()`	true	()
`pop()`	null	()
`push(7)`	–	(7)
`push(9)`	–	(7, 9)
`top()`	9	(7, 9)
`push(4)`	–	(7, 9, 4)
`size()`	3	(7, 9, 4)
`pop()`	4	(7, 9)
`push(6)`	–	(7, 9, 6)
`push(8)`	–	(7, 9, 6, 8)
`pop()`	8	(7, 9, 6)



(representação gráfica)

Pilhas

Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Pilhas

Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Pilha usando arranjo

```
push(e):                                  O(1)
  t += 1; data[t] ← e
```

Pilhas

Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Pilha usando arranjo

```
push(e):                                  O(1)
  t += 1; data[t] ← e
top():                                    O(1)
  retorna data[t]
```

Pilhas

Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Pilha usando arranjo

```
push(e):                                  O(1)
  t += 1; data[t] ← e
top():                                    O(1)
  retorna data[t]
pop():                                    O(1)
  retorna data[t]; t -= 1
```

Pilhas

Análise de complexidade

Em uma implementação usando **arranjos**, temos:

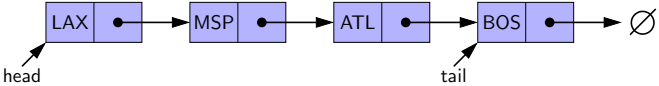
- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Pilha usando arranjo

```
push(e):                                  O(1)
  t += 1; data[t] ← e
top():                                    O(1)
  retorna data[t]
pop():                                    O(1)
  retorna data[t]; t -= 1
```

Em uma implementação usando **encadeamento**, temos:

- Uma lista simplesmente encadeada `list`.
- O topo da pilha é o início da lista.



Pilhas

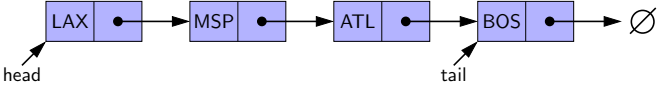
Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Em uma implementação usando **encadeamento**, temos:

- Uma lista simplesmente encadeada `list`.
- O topo da pilha é o início da lista.



Pilha usando arranjo

```
push(e):                    O(1)
  t += 1; data[t] ← e
top():                      O(1)
  retorna data[t]
pop():                      O(1)
  retorna data[t]; t -= 1
```

Pilha usando encadeamento

```
push(e):                    O(1)
  llist.add_first(e)
```

Pilhas

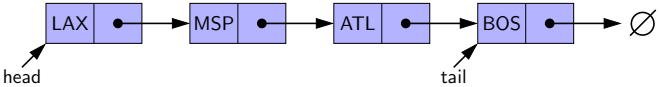
Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Em uma implementação usando **encadeamento**, temos:

- Uma lista simplesmente encadeada `list`.
- O topo da pilha é o início da lista.



Pilha usando arranjo

```
push(e):                    O(1)
  t += 1; data[t] ← e
top():                      O(1)
  retorna data[t]
pop():                      O(1)
  retorna data[t]; t -= 1
```

Pilha usando encadeamento

```
push(e):                    O(1)
  llist.add_first(e)
top():                      O(1)
  retorna llist.first()
```

Pilhas

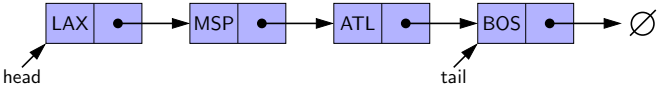
Análise de complexidade

Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Em uma implementação usando **encadeamento**, temos:

- Uma lista simplesmente encadeada `list`.
- O topo da pilha é o início da lista.



Pilha usando arranjo

```
push(e):                    O(1)
  t += 1; data[t] ← e
top():                      O(1)
  retorna data[t]
pop():                      O(1)
  retorna data[t]; t -= 1
```

Pilha usando encadeamento

```
push(e):                    O(1)
  llist.add_first(e)
top():                      O(1)
  retorna llist.first()
pop():                      O(1)
  retorna llist.remove_first()
```

Pilhas

Análise de complexidade

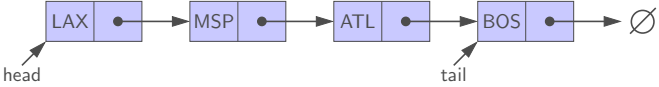
Em uma implementação usando **arranjos**, temos:

- Um vetor `data` que armazena os elementos.
- Uma variável `t` que indica o índice do topo da pilha.

Operação	`ArrayStack`	`LinkedStack`
`size`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`is_empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`top`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`push`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`pop`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Em uma implementação usando **encadeamento**, temos:

- Uma lista simplesmente encadeada `list`.
- O topo da pilha é o início da lista.



Filas

Ideia geral

Uma **fila** implementa a política **FIFO** – *first in, first out*. Ou seja, o primeiro elemento adicionado à estrutura é o primeiro a ser removido.

Filas

Ideia geral

Uma **fila** implementa a política **FIFO** – *first in, first out*. Ou seja, o primeiro elemento adicionado à estrutura é o primeiro a ser removido.

A fila simula uma "fila de pessoas", logo ela tem uma **frente** e um **final**.

Filas

Ideia geral

Uma **fila** implementa a política **FIFO** – *first in, first out*. Ou seja, o primeiro elemento adicionado à estrutura é o primeiro a ser removido.

A fila simula uma "fila de pessoas", logo ela tem uma **frente** e um **final**.

Operações:

- `enqueue`: insere um elemento no final da fila.
- `dequeue`: remove (e retorna) o elemento do início da fila.

Filas

Ideia geral

Uma **fila** implementa a política **FIFO** – *first in, first out*. Ou seja, o primeiro elemento adicionado à estrutura é o primeiro a ser removido.

A fila simula uma "fila de pessoas", logo ela tem uma **frente** e um **final**.

Operações:

- `enqueue`: insere um elemento no final da fila.
- `dequeue`: remove (e retorna) o elemento do início da fila.

Aplicações:

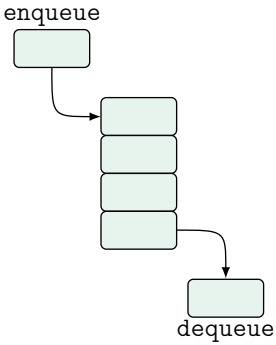
- Impressora de rede.
- Servidor Web respondendo requisições.

Filas

Exemplo de funcionamento

Seja uma fila implementada usando uma lista (sequencial), onde a frente é o início da lista.

Método	Retorno	Conteúdo da fila
`enqueue(5)`	–	(5)
`enqueue(3)`	–	(5, 3)
`size()`	2	(5, 3)
`dequeue()`	5	(3)
`is_empty()`	false	(3)
`dequeue()`	3	()
`is_empty()`	true	()
`dequeue()`	null	()
`enqueue(7)`	–	(7)
`enqueue(9)`	–	(7, 9)
`first()`	7	(7, 9)
`enqueue(4)`	–	(7, 9, 4)

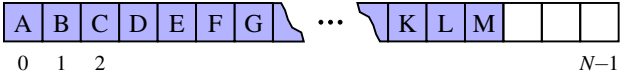


(representação gráfica)

Filas

Análise de complexidade

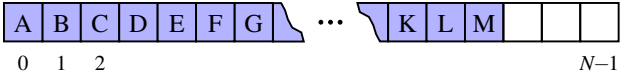
Em uma implementação usando arranjos, temos:



Filas

Análise de complexidade

Em uma implementação usando arranjos, temos:



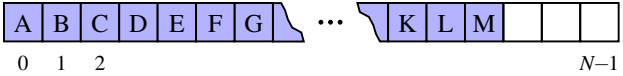
A frente da fila é no início do *array*. Como **desenfileirar** (i.e., remover do início)?

- Remover do índice 0 e fazer *shift* dos elementos. → $\mathcal{O}(n)$
- Usar uma variável (`f`) como referência para a frente da fila. → $\mathcal{O}(1)$

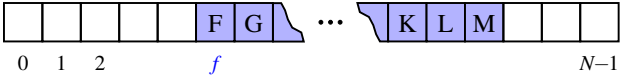
Filas

Análise de complexidade

Em uma implementação usando arranjos, temos:



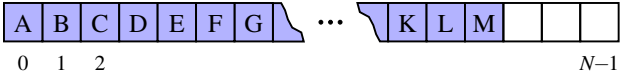
Remoção usando a variável `f`:



Filas

Análise de complexidade

Em uma implementação usando arranjos, temos:



Remoção usando a variável `f`:

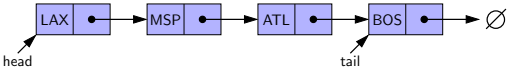
O novo desafio é controlar os espaços vagos no início do *array*:

Filas

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

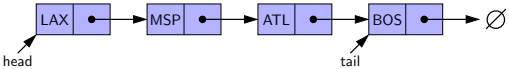


Filas

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**



Fila usando arranjo

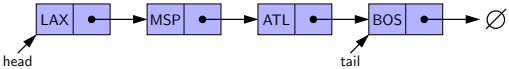
```
enqueue(e):                                  O(1)
  pos ← (f + size) % len(data)
  data[pos] ← e
```

Filas

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**



Fila usando arranjo

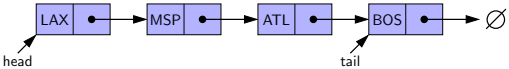
```
enqueue(e):                                  O(1)
  pos ← (f + size) % len(data)
  data[pos] ← e

dequeue():                                   O(1)
  pos ← f
  f ← (f + 1) % len(data)
  retorna data[pos]
```

Filas

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**



Fila usando arranjo

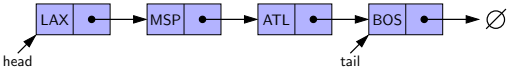
```
enqueue(e):                                  O(1)
  pos ← (f + size) % len(data)
  data[pos] ← e

dequeue():                                   O(1)
  pos ← f
  f ← (f + 1) % len(data)
  retorna data[pos]

first():                                     O(1)
  retorna data[f]
```

Filas

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**



Fila usando arranjo

```
enqueue(e):                                  O(1)
  pos ← (f + size) % len(data)
  data[pos] ← e

dequeue():                                   O(1)
  pos ← f
  f ← (f + 1) % len(data)
  retorna data[pos]

first():                                     O(1)
  retorna data[f]
```

Fila usando encadeamento

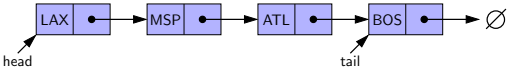
```
enqueue(e):                                  O(1)
  llist.add_last(e)

dequeue():                                   O(1)
  retorna llist.remove_first()

first():                                     O(1)
  retorna llist.first()
```

Filas

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**



Operação	`ArrayQueue`	`LinkedQueue`
`size`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`is_empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`first`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`enqueue`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`dequeue`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

Deques

Ideia geral

Um **deque** (*double-ended queue*) é uma fila com inserção e remoção em ambos os lados.

- i.e., é possível inserir e remover do início e do fim, mas não do meio.

Deques

Ideia geral

Um **deque** (*double-ended queue*) é uma fila com inserção e remoção em ambos os lados.

- i.e., é possível inserir e remover do início e do fim, mas não do meio.

Operações:

- `add_first`: insere um elemento no início da estrutura.
- `add_last`: insere um elemento no final da estrutura.
- `remove_first`: remove (e retorna) o elemento do início da estrutura.
- `remove_last`: remove (e retorna) o elemento do final da estrutura.

Deques

Ideia geral

Um **deque** (*double-ended queue*) é uma fila com inserção e remoção em ambos os lados.

- i.e., é possível inserir e remover do início e do fim, mas não do meio.

Operações:

- `add_first`: insere um elemento no início da estrutura.
- `add_last`: insere um elemento no final da estrutura.
- `remove_first`: remove (e retorna) o elemento do início da estrutura.
- `remove_last`: remove (e retorna) o elemento do final da estrutura.

Aplicações:

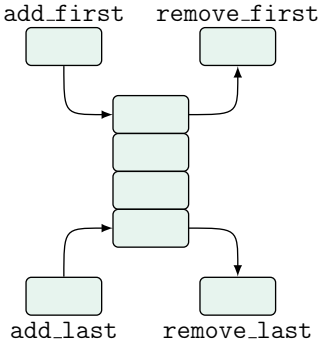
- Algoritmos de escalonamento de processos.
- Base para outras estruturas (e.g., fila dupla).

Deques

Exemplo de funcionamento

Seja um deque implementado usando uma lista (sequencial), onde a frente é o início da lista.

Método	Retorno	Conteúdo do deque
`add_last(5)`	–	(5)
`add_first(3)`	–	(3, 5)
`add_first(7)`	–	(7, 3, 5)
`first()`	7	(7, 3, 5)
`remove_last()`	5	(7, 3)
`size()`	2	(7, 3)
`remove_last()`	3	(7)
`remove_first()`	7	()
`add_first(6)`	–	(6)
`last()`	6	(6)
`add_first(8)`	–	(8, 6)
`is_empty()`	false	(8, 6)
`last()`	6	(8, 6)



(representação gráfica)

Deques

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

Deques

Análise de complexidade

Implementação usando **arranjos**

Deque usando arranjo

```
add_first(e):                    O(1)
  f ← (f - 1) % len(data)
  data[f] ← e
```

Implementação usando **encadeamento**

Deques

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

Deque usando arranjo

```
add_first(e):                                  O(1)
  f ← (f - 1) % len(data)
  data[f] ← e

add_last(e):                                   O(1)
  f ← (f + size) % len(data)
  data[f] ← e
```

Deques

Análise de complexidade

Implementação usando **arranjos**

Deque usando arranjo

```
remove_first():                    O(1)
  pos ← f
  f ← (f + 1) % len(data)
  retorna data[pos]
```

Implementação usando **encadeamento**

Deques

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

Deque usando arranjo

```
remove_first():                                  O(1)
  pos ← f
  f ← (f + 1) % len(data)
  retorna data[pos]

remove_last():                                   O(1)
  pos ← (f + size - 1) % len(data)
  size ← size - 1
  retorna data[pos]
```

Deques

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

Deque usando arranjo

```
remove_first():                    O(1)
  pos ← f
  f ← (f + 1) % len(data)
  retorna data[pos]

remove_last():                     O(1)
  pos ← (f + size - 1) % len(data)
  size ← size - 1
  retorna data[pos]
```

Deque usando encadeamento

```
add_first(e):                      O(1)
  llist.add_first(e)

add_last(e):                       O(1)
  llist.add_last(e)

remove_first():                    O(1)
  llist.remove_first()

remove_last():                     O(1)
  llist.remove_last()
```

Deques

Análise de complexidade

Implementação usando **arranjos**

Implementação usando **encadeamento**

Operação	`ArrayDeque`	`LinkedDeque`
`size`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`is_empty`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`first`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`last`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`add_first`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`add_last`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`remove_first`	$\mathcal{O}(1)$	$\mathcal{O}(1)$
`remove_last`	$\mathcal{O}(1)$	$\mathcal{O}(1)$

45RPE – Resolução de Problemas com Estruturas de Dados
Prof. Marcelo de Souza