

Estruturas de Dados

Filas e Pilhas

Aula 04

Prof. Felipe A. Louza



- 1 Filas
- 2 Pilhas
- 3 Referências

1 Filas

2 Pilhas

3 Referências

Uma fila (*queue*) é uma **estrutura de dados dinâmica** que admite inserção e remoção de elementos.

- Primeiro a entrar é primeiro a sair : **FIFO** (first-in first-out)



Não podemos acessar elementos do “meio” da fila

Operações:

- **Enfileira** (*push_back*): adiciona item no “fim” da fila

Operações:

- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Operações:


- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo:



Operações:


- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Enfileira**()



Operações:


- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Enfileira**()



Operações:


- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Enfileira**()



Operações:

- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Enfileira**()



Operações:

- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Desenfileira()**



Operações:


- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Desenfileira()**



Operações:


- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Enfileira**()



Operações:


- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Enfileira**()



Operações:


- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Enfileira**()



Operações:

- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Enfileira**()



Operações:

- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Desenfileira()**



Operações:

- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Desenfileira()**



Operações:

- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

Exemplo: **Desenfileira()**



Operações:

- **Enfileira** (*push_back*): adiciona item no “fim” da fila
- **Desenfileira** (*pop_front*): remove item do “início” da fila

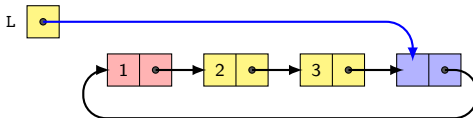
Exemplo: **Fim!**



Filas: implementação com lista ligada

Implementação: Lista ligada circular

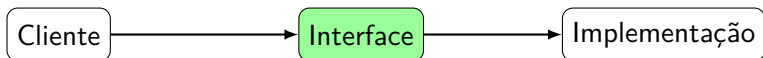
- Inserção no final: $O(1)$
- Remoção no início: $O(1)$



Operações:

- | | |
|---------------------------------|-------------------------------|
| ① <code>fila_criar()</code> | ④ <code>fila_topo()</code> |
| ② <code>fila_destruir()</code> | ⑤ <code>fila_remove()</code> |
| ③ <code>fila_adicionar()</code> | ⑥ <code>fila_tamanho()</code> |

TAD - Interface



fila.h

```
1  #ifndef FILA_H
2  #define FILA_H
3
4  #include "lista_circular.h"
5
6  //Dados
7  typedef struct {
8      int tam;
9      No *L;
10 } Fila;
11
12 //Funções
13 Fila* fila_criar();
14 void fila_destruir(Fila **F);
```

```
16 //Adicionar
17 void fila_adicionar(Fila **F, int x);
18
19 //Acessar
20 int fila_topo(Fila *F);
21
22 //Remover
23 void fila_remover(Fila **F);
24
25 //Extra
26 int fila_tamanho(Fila *F);
27
28
29 #endif
```

- Vamos reutilizar o TAD de listas circulares

Fila – Criar fila

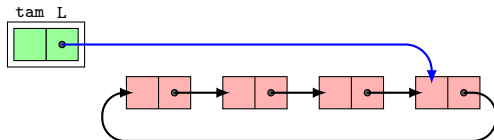


fila.c

```
5 Fila* fila_criar() {  
6     Fila *p = (Fila*) malloc(sizeof(Fila));  
7     p->tam = 0;  
8     p->L = criar_lista();  
9     return p;  
10 }
```

- Vamos alocar o espaço para uma Fila;
- Código do cliente: `Fila *F = fila_criar();`

Fila – Destruir fila

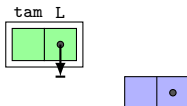


`fila.c`

```
12 void fila_destruir(Fila **p) {// p recebe &F  
13     destruir_lista(&((*p)->L));  
14     free(*p);  
15     *p = NULL;  
16 }
```

- Primeiro apagamos a lista, depois a `struct Fila`

Fila – Adicionar no final

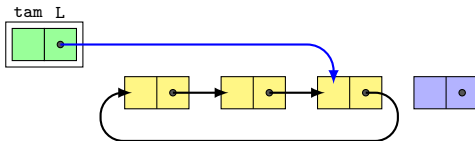


`fila.c`

```
20 void fila_adicionar(Fila **p, int x) {// p recebe &F  
21     adicionar_final(&((*p)->L), x);  
22     (*p)->tam++;  
23 }
```

- Incrementamos o valor de `tam`
- Custo computacional: $O(1)$

Fila – Adicionar no final

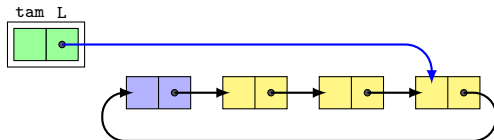


`fila.c`

```
20 void fila_adicionar(Fila **p, int x) {// p recebe &F  
21     adicionar_final(&((*p)->L), x);  
22     (*p)->tam++;  
23 }
```

- Incrementamos o valor de `tam`
- Custo computacional: $O(1)$

Fila – Acessar (início)



lista_circular.c

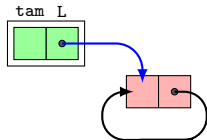
```
127 int acessar_primeiro(No *p){  
128     if(p==NULL) return -1; //lista vazia  
129     No *q = p->prox;  
130     return q->dado;  
131 }
```

fila.c

```
27 int fila_topo(Fila *p) {// p recebe F  
28     return acessar_primeiro(p->L);  
29 }
```

- Custo computacional: $O(1)$

Lista circular – Remover do início

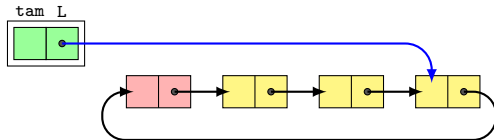


fila.c

```
33 void fila_remover(Fila **p) {// p recebe &L  
34     remover_inicio(&((*p)->L));  
35     (*p)->tam--;  
36 }
```

- Custo computacional: $O(1)$

Lista circular – Remover do início

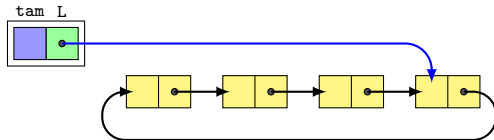


fila.c

```
33 void fila_remover(Fila **p) {// p recebe &L  
34     remover_inicio(&((*p)->L));  
35     (*p)->tam--;  
36 }
```

- Custo computacional: $O(1)$

Fila – Tamanho



fila.c

```
40 int fila_tamanho(Fila *p) {// p recebe L  
41     return p->tam;  
42 }
```

- Custo computacional: $O(1)$



exemplo1.c

```
1  #include <stdio.h>
2  #include "fila.h"
3
4  int main(){
5      int i, num;
6      Fila *F = fila_criar();
7      scanf("%d", &num);
8      for(i=1; i<=num; i++) fila_adicionar(&F, i);
9      while(fila_tamanho(F) > 0){
10         printf("%d\n", fila_topo(F));
11         fila_remove(&F);
12     }
13     fila_destruir(&F); //testar valgrind!!
14     return 0;
15 }
```


Como compilar?

Teremos dois TADs diferentes:

- `fila.c`
- `lista_circular.c`

Vamos compilar por partes:

- `gcc -Wall -Werror -c lista_circular.c`
 - vai gerar o arquivo compilado `lista_circular.o`
- `gcc -Wall -Werror -c fila.c`
 - vai gerar o arquivo compilado `fila.o`
- `gcc exemplo1.c fila.o lista_circular.o -o exemplo1`
 - `compila`, e faz a linkagem, `gerando o executável exemplo1`

Makefile

Vamos usar o **Makefile** para compilar:

```
1 CFLAGS= -Wall -Werror
2
3 all: exemplo1
4
5 exemplo1: exemplo1.c fila.o lista_circular.o
6     gcc $^ -o $@
7
8 #regra genérica
9 %.o: %.c %.h
10     gcc $(CFLAGS) -c $<
```

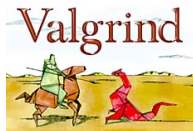
Relembrando:

- **\$^** representa todas as dependências da regra, e **\$@** o alvo
- Regras genéricas:
 - Para cada ***.o** criamos uma regra com dependências ***.c** e ***.h**
 - **\$<** representa a primeira dependência

Podemos verificar se toda memória foi desalocada com o **Valgrind**:

```
1 $ valgrind --leak-check=yes ./exemplo1
```

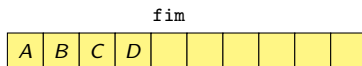
- Podemos encontrar **vazamento de memória** e **acesso a posições inválidas**:
 - Valgrind avisa que houveram bytes perdidos
 - Indica qual o **malloc** responsável pelo vazamento



Fila - implementação com vetor

Primeira ideia:

- Variável **fim** indica o fim da fila no vetor
- Inserimos no final do vetor: $O(1)$



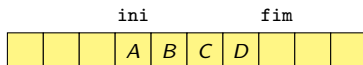
Após remover um elemento, precisamos mover todos para a esquerda

- Custo computacional: $O(n) \leftarrow$ ruim!

Fila - implementação com vetor

Segunda ideia:

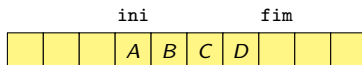
- Variável **ini** indica o começo da fila
- Variável **fim** indica o fim da fila



Fila - implementação com vetor

Segunda ideia:

- Variável **ini** indica o começo da fila
- Variável **fim** indica o fim da fila

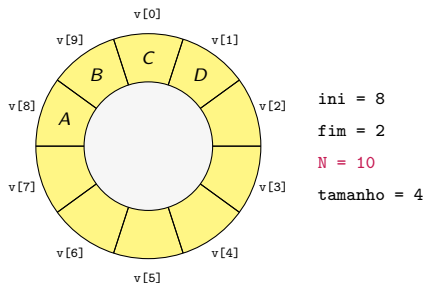


E se, ao inserir, tivermos espaço apenas à esquerda de **ini**?

- temos mover **toda a fila** para o começo do vetor $\leftarrow O(n)!$

Fila - implementação com vetor (fila circular)

Terceira ideia: considerar o vetor de tamanho N de maneira **circular**



As manipulações de índices são realizadas módulo N

Fila - Listas circular vs. vetor circular

	Lista circular	Vetor circular
Inserção	$O(1)$	$O(1)$
Remoção	$O(1)$	$O(1)$
Acesso (topo)	$O(1)$	$O(1)$
Tamanho	$O(1)$	$O(1)$

Comparação:

- **Custo computacional** de cada operação: $O(1)$
- Lista ligada sem restrição de tamanho máximo

Exemplos de aplicações

Algumas aplicações de filas:

- Gerenciamento de fila de impressão
- Buffer do teclado
- Escalonamento de processos
- Comunicação entre aplicativos/computadores
- Percurso de estruturas de dados complexas (grafos etc.)



1 Filas

2 Pilhas

3 Referências

Uma pilha (*stack*) é uma **estrutura de dados dinâmica** que também admite inserção e remoção de elementos.

- Último a entrar é primeiro a sair : **LIFO** (last-in first-out)



Não podemos acessar elementos do “meio” da pilha

Pilhas

Operações:

- Empilha (*push*): adiciona no “topo” da pilha
- Desempilha (*pop*): remove do “topo” da pilha

Exemplo:



Pilhas

Operações:

- **Empilha** (*push*): adiciona no “topo” da pilha
- **Desempilha** (*pop*): remove do “topo” da pilha

Exemplo: **Empilha**(A)



Pilhas

Operações:

- Empilha (*push*): adiciona no “topo” da pilha
- Desempilha (*pop*): remove do “topo” da pilha

Exemplo: Empilha(A)

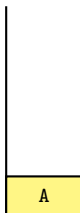


Pilhas

Operações:

- Empilha (*push*): adiciona no “topo” da pilha
- Desempilha (*pop*): remove do “topo” da pilha

Exemplo: Empilha(B)

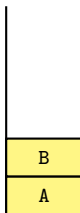


Pilhas

Operações:

- Empilha (*push*): adiciona no “topo” da pilha
- Desempilha (*pop*): remove do “topo” da pilha

Exemplo: Empilha(B)

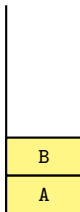


Pilhas

Operações:

- **Empilha** (*push*): adiciona no “topo” da pilha
- **Desempilha** (*pop*): remove do “topo” da pilha

Exemplo: **Desempilha()**

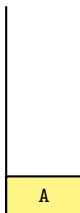


Pilhas

Operações:

- **Empilha** (*push*): adiciona no “topo” da pilha
- **Desempilha** (*pop*): remove do “topo” da pilha

Exemplo: **Desempilha()**

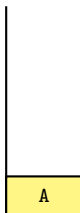


Pilhas

Operações:

- Empilha (*push*): adiciona no “topo” da pilha
- Desempilha (*pop*): remove do “topo” da pilha

Exemplo: Empilha(C)

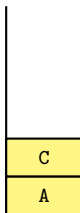


Pilhas

Operações:

- **Empilha** (*push*): adiciona no “topo” da pilha
- **Desempilha** (*pop*): remove do “topo” da pilha

Exemplo: **Empilha**(C)

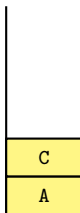


Pilhas

Operações:

- **Empilha** (*push*): adiciona no “topo” da pilha
- **Desempilha** (*pop*): remove do “topo” da pilha

Exemplo: **Empilha**(D)

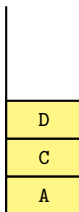


Pilhas

Operações:

- **Empilha** (*push*): adiciona no “topo” da pilha
- **Desempilha** (*pop*): remove do “topo” da pilha

Exemplo: **Empilha**(D)

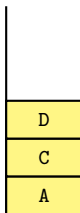


Pilhas

Operações:

- **Empilha** (*push*): adiciona no “topo” da pilha
- **Desempilha** (*pop*): remove do “topo” da pilha

Exemplo: **Desempilha()**

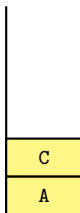


Pilhas

Operações:

- **Empilha** (*push*): adiciona no “topo” da pilha
- **Desempilha** (*pop*): remove do “topo” da pilha

Exemplo: **Desempilha()**

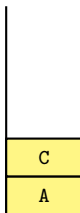


Pilhas

Operações:

- **Empilha** (*push*): adiciona no “topo” da pilha
- **Desempilha** (*pop*): remove do “topo” da pilha

Exemplo: **Desempilha()**

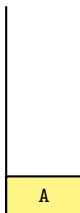


Pilhas

Operações:

- **Empilha** (*push*): adiciona no “topo” da pilha
- **Desempilha** (*pop*): remove do “topo” da pilha

Exemplo: **Desempilha()**



Pilhas

Operações:

- Empilha (*push*): adiciona no “topo” da pilha
- Desempilha (*pop*): remove do “topo” da pilha

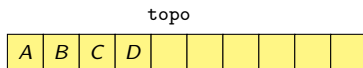
Exemplo: Fim!



Pilha - implementação com vetor

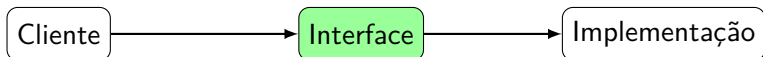
Implementação: vetores

- Inserção no final: $O(1)$
- Remoção no final: $O(1)$



Operações:

- | | |
|----------------------------------|--------------------------------|
| ① <code>pilha_criar()</code> | ④ <code>pilha_topo()</code> |
| ② <code>pilha_destruir()</code> | ⑤ <code>pilha_remove()</code> |
| ③ <code>pilha_adicionar()</code> | ⑥ <code>pilha_tamanho()</code> |



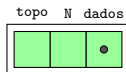
pilha.h

```
1 #ifndef PILHA_VETOR_H
2 #define PILHA_VETOR_H
3
4 //Dados
5 typedef struct {
6     int topo, N;
7     int *dados;
8 } Pilha;
9
10 #define MAX 100
11
12 //Funções
13 Pilha* pilha_criar();
14 void pilha_destruir(Pilha **P);
```

```
16 //Adicionar
17 void pilha_adicionar(Pilha **P, int x);
18
19 //Acessar
20 int pilha_topo(Pilha *P);
21
22 //Remover
23 void pilha_remover(Pilha **P);
24
25 //Extra
26 int pilha_tamanho(Pilha *P);
27
28
29 #endif
```

- Vamos definir uma tamanho máximo da Pilha como **constante**

Pilha circular - Criar pilha

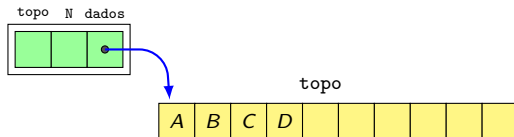


`pilha.c`

```
5 Pilha* pilha_criar() {  
6     Pilha *P = malloc(sizeof(Pilha));  
7     P->dados = malloc(MAX * sizeof(int));  
8     P->topo = 0;  
9     P->N = MAX;  
10    return P;  
11 }
```

- Primeiro alocamos a `struct Pilha`, depois o `vetor`

Pilha circular - Criar pilha

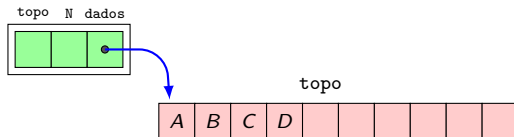


`pilha.c`

```
5 Pilha* pilha_criar() {  
6     Pilha *P = malloc(sizeof(Pilha));  
7     P->dados = malloc(MAX * sizeof(int));  
8     P->topo = 0;  
9     P->N = MAX;  
10    return P;  
11 }
```

- Primeiro alocamos a `struct Pilha`, depois o `vetor`

Pilha circular - Destruir pilha

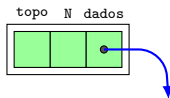


`pilha.c`

```
13 void pilha_destruir(Pilha **p) {// p recebe &P  
14     free((*p)->dados);  
15     free(*p); //testar valgrind!  
16     *p = NULL;  
17 }
```

- Primeiro apagamos o **vetor**, depois a **struct Pilha**

Pilha circular - Destruir pilha

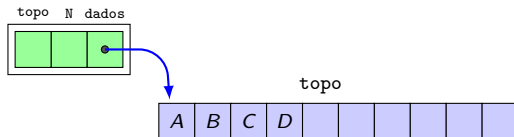


`pilha.c`

```
13 void pilha_destruir(Pilha **p) {// p recebe &p  
14     free((*p)->dados);  
15     free(*p); //testar valgrind!  
16     *p = NULL;  
17 }
```

- Primeiro apagamos o `vetor`, depois a `struct Pilha`

Pilha circular - Adicionar no final

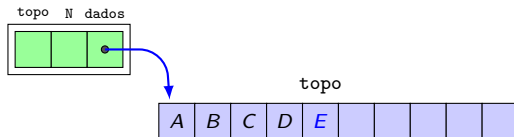


`pilha.c`

```
19 void pilha_adicionar(Pilha **p, int x) {// p recebe &P  
20     (*p)->dados[(*p)->topo] = x;  
21     (*p)->topo++;  
22 }
```

- Adicionar **E**

Pilha circular - Adicionar no final

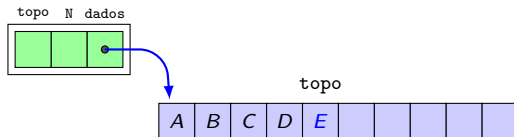


`pilha.c`

```
19 void pilha_adicionar(Pilha **p, int x) {// p recebe &P  
20     (*p)->dados[(*p)->topo] = x;  
21     (*p)->topo++;  
22 }
```

- Adicionar **E**

Pilha circular - Adicionar no final

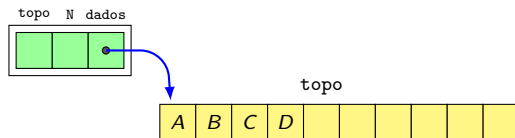


`pilha.c`

```
19 void pilha_adicionar(Pilha **p, int x) {// p recebe &P  
20     (*p)->dados[(*p)->topo] = x;  
21     (*p)->topo++;  
22 }
```

- Adicionar **E**
- Custo computacional: $O(1)$ ← se a pilha estourar, temos acessos inválidos!!

Pilha circular - Acessar (final)

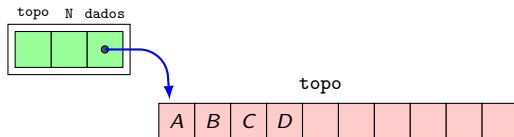


`pilha.c`

```
24 int pilha_topo(Pilha *p){  
25     return (p->topo)?p->dados[p->topo-1]:-1;  
26 }
```

- Custo computacional: $O(1)$

Pilha circular - Remover do final

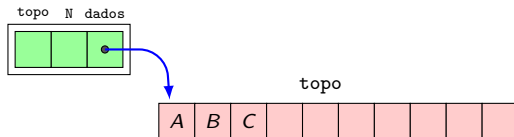


`pilha.c`

```
28 void pilha_remove(Pilha **p) {// p recebe P  
29     if((*p)->topo==0) return;  
30     (*p)->topo--;  
31 }
```

- Remover do final: **D**

Pilha circular - Remover do final

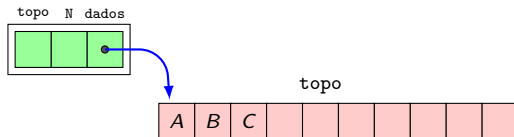


`pilha.c`

```
28 void pilha_remove(Pilha **p) {// p recebe P  
29     if((*p)->topo==0) return;  
30     (*p)->topo--;  
31 }
```

- Remover do final: **D**

Pilha circular - Remover do final

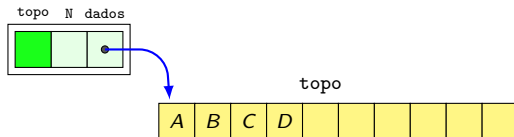


`pilha.c`

```
28 void pilha_remove(Pilha **p) {// p recebe P  
29     if((*p)->topo==0) return;  
30     (*p)->topo--;  
31 }
```

- Remover do final: **D**
- Custo computacional: **$O(1)$**

Pilha circular - Tamanho



`pilha.c`

```
33 int pilha_tamanho(Pilha *p) { // p recebe P
34     return p->topo;
35 }
```

- Custo computacional: $O(1)$



exemplo2.c

```
1  #include <stdio.h>
2  #include "pilha_vetor.h"
3
4  int main(){
5      int i, num;
6      Pilha *F = pilha_criar();
7      scanf("%d", &num);
8      for(i=1; i<=num; i++) pilha_adicionar(&F, i);
9      while(pilha_tamanho(F) > 0){
10         printf("%d\n", pilha_topo(F));
11         pilha_remover(&F);
12     }
13     pilha_destruir(&F); //testar valgrind!!
14     return 0;
15 }
```

Makefile e Valgrind

Vamos usar o **Makefile** para compilar:

```
1 exemplo2: exemplo2.c pilha_vetor.o
2 gcc $^ -o $@
```

Vamos verificar a memória com o **Valgrind**:

```
1 $ valgrind --leak-check=yes ./exemplo2
```



	Vetor
Inserção	$O(1)$
Remoção	$O(1)$
Acesso (topo)	$O(1)$
Tamanho	$O(1)$

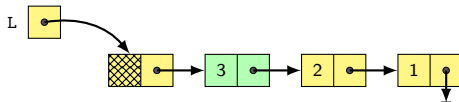
Comparação:

- **Custo computacional** de cada operação: $O(1)$
- Temos uma restrição no tamanho máximo do vetor

Pilhas: implementação com lista ligada

Implementação: Lista com nó cabeça

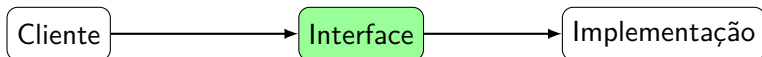
- Inserção no início: $O(1)$
- Remoção no início: $O(1)$



Operações:

- | | |
|----------------------------------|--------------------------------|
| ① <code>pilha_criar()</code> | ④ <code>pilha_topo()</code> |
| ② <code>pilha_destruir()</code> | ⑤ <code>pilha_remove()</code> |
| ③ <code>pilha_adicionar()</code> | ⑥ <code>pilha_tamanho()</code> |

TAD - Interface



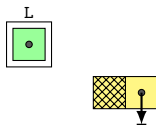
pilha.h

```
1  #ifndef PILHA_H
2  #define PILHA_H
3
4  #include "lista_com_cabeca.h"
5
6  //Dados
7  typedef struct {
8      No *L;
9  } Pilha;
10
11 //Funções
12 Pilha* pilha_criar();
13 void pilha_destruir(Pilha **P);
```

```
15 //Adicionar
16 void pilha_adicionar(Pilha **P, int x);
17
18 //Acessar
19 int pilha_topo(Pilha *P);
20
21 //Remover
22 void pilha_remover(Pilha **P);
23
24 //Extra
25 int pilha_tamanho(Pilha *P);
26
27 #endif
```

- Vamos reutilizar o TAD de listas com nó cabeça

Pilha – Criar pilha

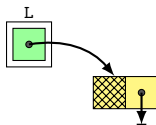


`pilha.c`

```
5 Pilha* pilha_criar() {  
6     Pilha *p = (Pilha*) malloc(sizeof(Pilha));  
7     p->L = criar_lista();  
8     return p;  
9 }
```

- Vamos alocar o espaço para uma `Pilha`;
- Código do cliente: `Pilha *P = pilha_criar();`

Pilha – Criar pilha

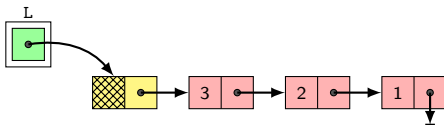


`pilha.c`

```
5 Pilha* pilha_criar() {  
6     Pilha *p = (Pilha*) malloc(sizeof(Pilha));  
7     p->L = criar_lista();  
8     return p;  
9 }
```

- Vamos alocar o espaço para uma `Pilha`;
- Código do cliente: `Pilha *P = pilha_criar();`

Pilha – Destruir pilha

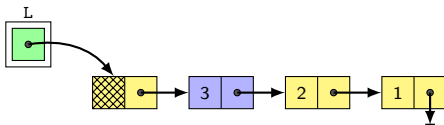


`pilha.c`

```
11 void pilha_destruir(Pilha **p) {// p recebe &F  
12     destruir_lista(&((*p)->L));  
13     free(*p); //testar valgrind!  
14     *p = NULL;  
15 }
```

- Primeiro apagamos a lista, depois a `struct Pilha`

Pilha – Adicionar no início

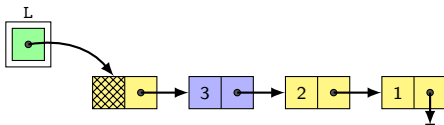


`pilha.c`

```
19 void pilha_adicionar(Pilha **p, int x) {// p recebe &F  
20     adicionar_inicio((*p)->L, x);  
21 }
```

- O TAD de lista com cabeça incrementa o tamanho da lista
- Custo computacional: $O(1)$

Pilha – Acessar (início)



`lista_com_cabeca.c`

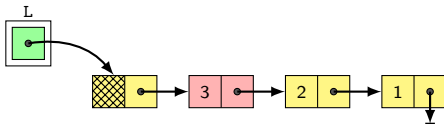
```
111 int acessar_primeiro(No *p){  
112     if(p->prox==NULL) return -1; //lista vazia  
113     return (p->prox)->dado;  
114 }
```

`pilha.c`

```
25 int pilha_topo(Pilha *p) {// p recebe F  
26     return acessar_primeiro(p->L);  
27 }
```

- Custo computacional: $O(1)$

Lista circular – Remover do início

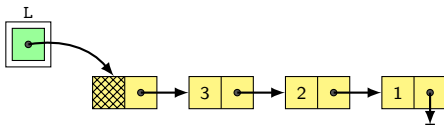


`pilha.c`

```
31 void pilha_remover(Pilha **p) {// p recebe &L  
32     remover_inicio((*p)->L);  
33 }
```

- O TAD de lista com cabeça incrementa o tamanho da lista
- Custo computacional: $O(1)$

Pilha – Tamanho



`pilha.c`

```
37 int pilha_tamanho(Pilha *p) {// p recebe L  
38     return tamanho_lista(p->L);  
39 }
```

- Custo computacional: $O(1)$

	Lista ligada	Vetor
Inserção	$O(1)$	$O(1)$
Remoção	$O(1)$	$O(1)$
Acesso (topo)	$O(1)$	$O(1)$
Tamanho	$O(1)$	$O(1)$

Comparação:

- **Custo computacional** de cada operação: $O(1)$
- Lista ligada sem restrição de tamanho máximo

Exemplos de aplicações

Algumas aplicações de pilhas:

- Balanceamento de parênteses
 - expressões matemáticas
 - linguagens de programação
 - HTML...

Veremos algumas dessas aplicações na próxima aula

Algumas aplicações de pilhas:

- Balanceamento de parênteses
 - expressões matemáticas
 - linguagens de programação
 - HTML...
- Cálculo e conversão de notações
 - pré-fixa
 - pós-fixa
 - infixa (com parênteses)

Algumas aplicações de pilhas:

- Balanceamento de parênteses
 - expressões matemáticas
 - linguagens de programação
 - HTML...
- Cálculo e conversão de notações
 - pré-fixa
 - pós-fixa
 - infixa (com parênteses)
- Percurso de estruturas de dados complexas (grafos etc.)
- Recursão

Dúvidas?

1 Filas

2 Pilhas

3 Referências

- ① Materiais adaptados dos slides do Prof. Rafael C. S. Schouery, da Universidade Estadual de Campinas.
- ② Feofiloff, Paulo. Algoritmos em linguagem C. Elsevier Brasil, 2009.