Algoritmos e Estrutura de Dados

Fabrício Olivetti de França

02 de Fevereiro de 2019

UFABC

CMCC
Centro de Matemática, Computação e Cognição
Center for Mathematics, Computation and Cognition

Topics

1. Estrutura de Dados
2. Tipos de Dados Algébricos
3. Tipos Compostos
4. Tipos de Dados Abstratos
5. Listas Lineares
6. Listas com alocação sequencial

Estrutura de Dados

Estrutura de Dados

Muitos programas de computadores operam em tabelas de informação. Processamento de dados!

Essa informação geralmente é apresentada de forma estruturada e organizada.

Estrutura de Dados

Uma possível estrutura simples é a lista linear de elementos. Ela responde as perguntas:

- Qual o primeiro elemento?
- Qual o último elemento?
- Qual o elemento anterior ou próximo desse elemento?
- Quantos elementos a lista possui?

Podemos extrair diversas outras informações dessa simples estrutura!

Um exemplo de estrutura mais complexa é o nosso cérebro que possui múltiplas conexões entre seus elementos.

Estrutura de Dados

Estrutura de Dados estuda as relações estruturais presentes nos dados e técnicas de representação e manipulação destes.

Estruturas Lineares

Estruturas Lineares são estruturas de armazenamento sequencial.

Tipos de Dados Algébricos

Tipos de Dados

Tipos representam conjuntos de valores dentro de um contexto.

Tipos de Dados

Alguns tipos básicos em C:

- **void:** habitado por apenas um único valor (que não pode ser utilizado explicitamente).
- **char:** habitado por 256 valores representando caracteres da tabela ASCII.
- **int:** habitado por 2^{32} valores quando utiliza 4 bytes.

Tipos Compostos

Tipo Produto

O **Tipo Produto** combina dois ou mais tipos em uma tupla de tipos.

Em C é criado utilizando `struct`:

```c
struct ponto {
    double x;
    double y;
};
```

Tipo Soma

O **Tipo Soma** cria uma escolha entre tipos ou valores. Em C isso é feito utilizando `enum` ou `union`:

```c
enum dia_semana {DOM, SEG, TER, QUA, QUI, SEX, SAB};

int eh_folga(int dia) {
  switch(dia) {
    case DOM:
    case SAB:
      return 1;
    default: return 0;
  }
}
```

Tipo Soma

```
struct int_ou_double {
    char eh_int;
    union valor {
        int x;
        double y;
    }
};
```

Registro

Em tabelas de informação, cada elemento é denominado **registro** e é representado em C por uma `struct` (combinada com `enum` e `union`):

```
enum suits {CLUBS, DIAMONDS, HEARTS, SPADES};
struct carta {
    char tag; // 0 - virada para cima, 1 - virada para ba
    int suit;
    unsigned int rank; // valor numerico
    struct carta * next;
    char name[10];
};
```

Registro

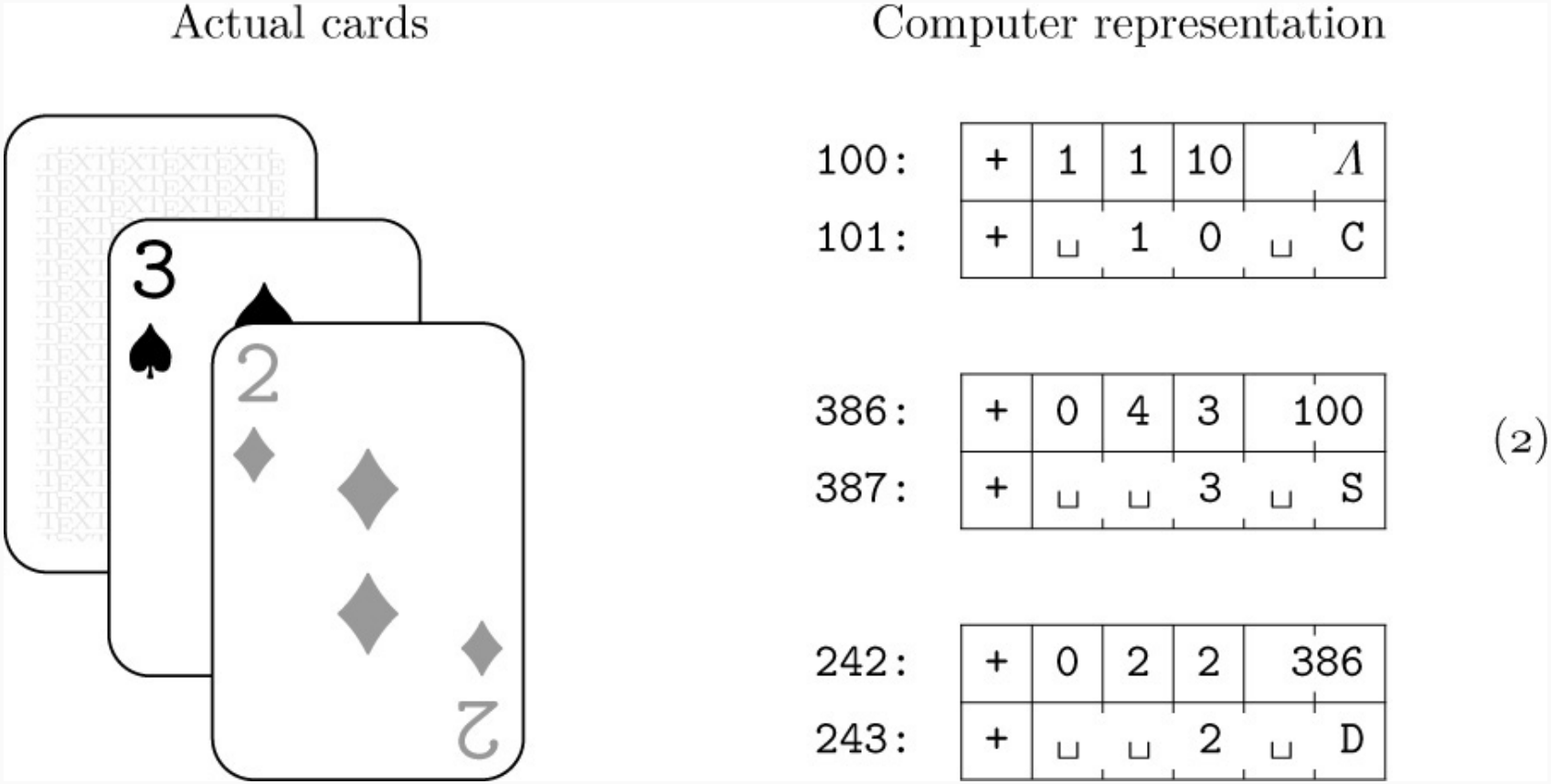


Figura 1: FONTE: The Art of Computer Programming - vol. 1

Registro

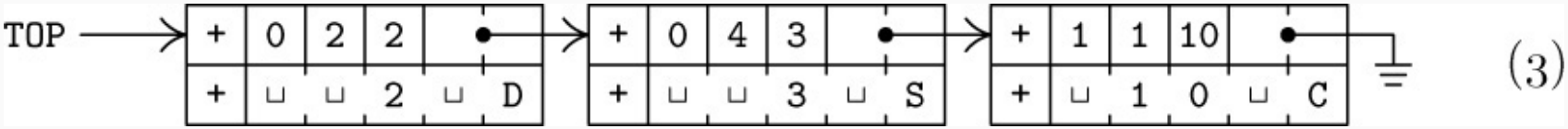


Figura 2: FONTE: The Art of Computer Programming - vol. 1

Reparem que temos um elemento recursivo na estrutura (`next`) que aponta para uma próxima carta. Essa ligação é importante para muitas estruturas que estudaremos adiante.

Tipos de Dados Abstratos

Tipos de Dados Abstratos

Um **Tipo de Dado Abstrato** é a descrição de uma estrutura de forma abstrata juntamente com as funções que devem ser implementadas para ela.

Tipos de Dados Abstratos

Nesse contexto **abstrato** significa que devemos descrever apenas os detalhes básicos da estrutura. Note que não descrevemos muitos aspectos da estrutura carta.

Listas Lineares

Listas Lineares

Listas lineares são sequências de $n \geq 0$ elementos denotados por $x_0, x_1, \ldots, x_{n-1}$ com índice iniciando em 0.

A propriedade fundamental dessa estrutura é a relação entre posições relativas dos elementos em uma linha.

O que nos interessa para essa estrutura é que se $n > 0$, x_0 é o primeiro elemento e x_{n-1} é o último e, se $0 < k < n - 1$, x_{k-1} precede x_k e ex_{k+1} sucede x_k.

Listas Lineares

As listas lineares possuem diversas operações associadas:

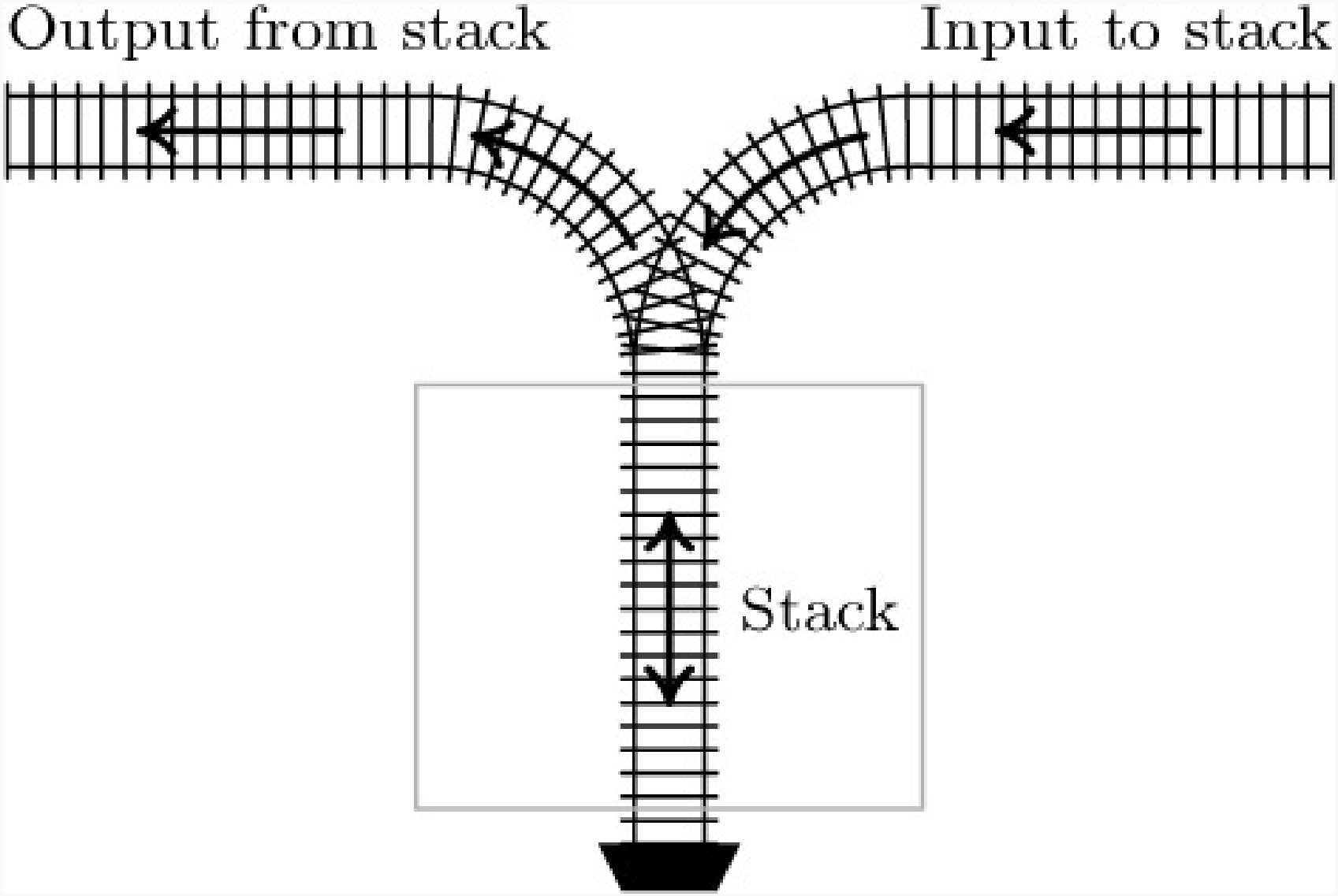
1. Acessar ou alterar o k-ésimo elemento
2. Inserir um novo registro antes ou após a posição k
3. Apagar o k-ésimo registro
4. Combinar duas listas lineares
5. Dividir uma lista em duas
6. Copiar uma lista
7. Determinar quantidade de registros
8. Ordenar os elementos em ordem crescente de acordo com um dos campos dos registros
9. Buscar um elemento da lista de acordo com um campo dos registros

As operações 1, 2 e 3 são interessantes pois podem ser mais ou menos custosas dependendo da estrutura utilizada. Além disso temos os casos especiais em que $k = 0$ e $k = n - 1$.

Nem todas as operações são sempre necessárias em um programa, e nenhuma estrutura possui desempenho ótimo para todas elas.

Portanto, as estruturas são classificadas de acordo com as operações em que elas possuem melhor desempenho.

- **Pilha (stack):** acesso, inserção e remoção de elementos sempre no final da lista. First-In Last-Out (FILO)
- **Fila (queue):** inserção no final da lista, acesso e remoção no começo da lista. First-In First-Out (FIFO)
- **Deque (double-ended queue):** acesso, inserção e remoção tanto no começo como no final da lista.


Output from stack
Input to stack
Stack

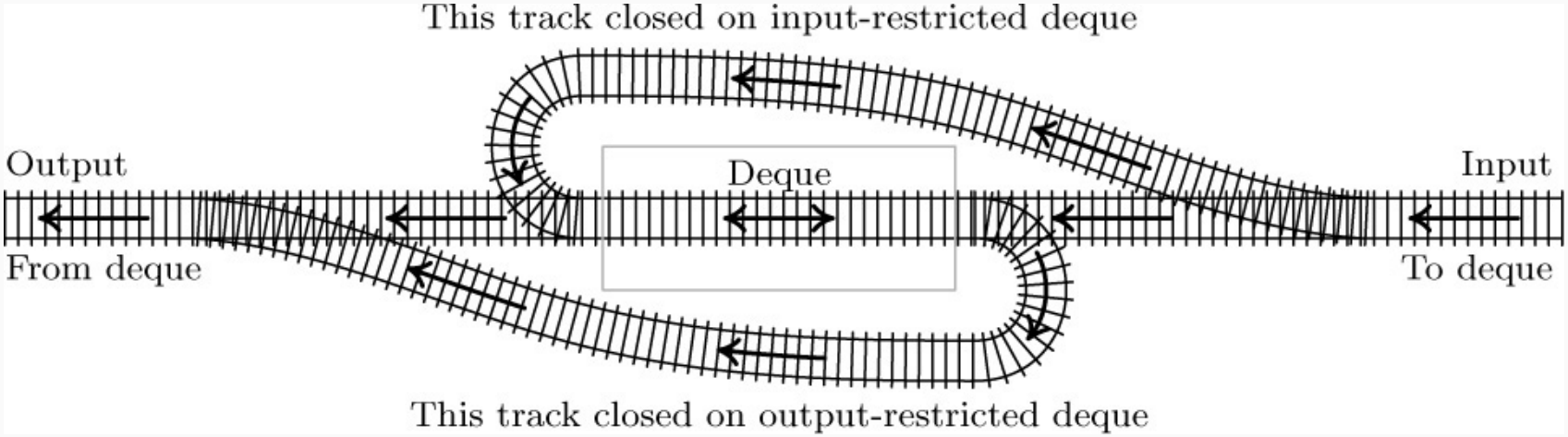


Figura 4: FONTE: The Art of Computer Programming - vol. 1

Listas com alocação sequencial

Alocação Sequencial

Forma simples e natural de representação, arrays em C: registros armazenados em segmentos sequenciais de memória.

```
struct s x;
c = sizeof(struct s);
x[j] = *(x + c*j);
```

Pilha

A **pilha** implementada em alocação sequencial possui uma capacidade máxima pré-fixada e consiste de uma array com n elementos e um apontador para o topo da pilha:

```
#define STACK_TYPE int
#define CAPACITY 100

typedef struct stack {
    STACK_TYPE data[CAPACITY];
    int top;
} stack;
```

Pilha

Inicialmente a pilha não possui elementos e o campo `top` aponta para lugar algum, representado por -1:

```
stack create_stack() {
  stack p;
  p.top = -1;
  return p;
}
```

Pilha

Essa estrutura possui apenas duas operações fundamentais:

- `push`: insere um elemento no topo da pilha.
- `pop`: remove um elemento do topo da pilha.

Pilha

```
stack * push(stack * p, STACK_TYPE x) {
  if (p->top + 1 < CAPACITY)
  {
    ++p->top;
    p->data[p->top] = x;
    return p;
  } else return NULL;
}
```

Pilha

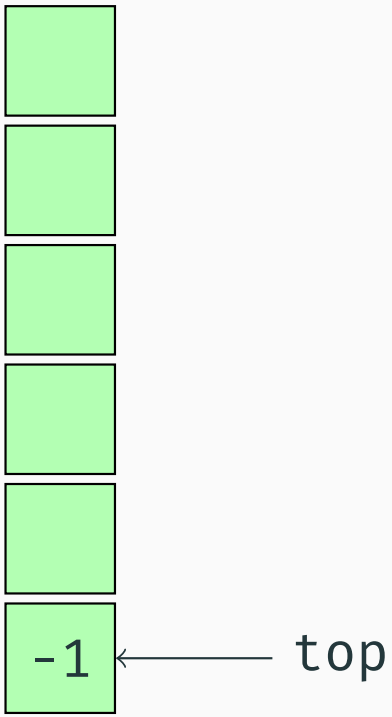
```
STACK_TYPE * pop(stack * p) {
    if (p->top == -1) return NULL;
    --p->top;
    return &(p->data[p->top + 1]);
}
```

Pilha

Exemplo de uso:

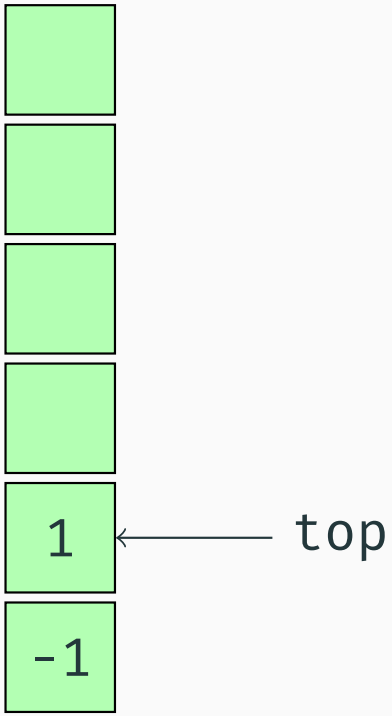
```
...
int i;
STACK_TYPE * x;
stack p = create_stack();
for (i = 0; i < 10; i++){
   if (push(&p, i) == NULL) {
       printf("ERRO! ESTOURO DE PILHA!\n");
   }
}

while( (x = pop(&p)) != NULL ) {
    printf("%d\n", x);
}
```

Pilha



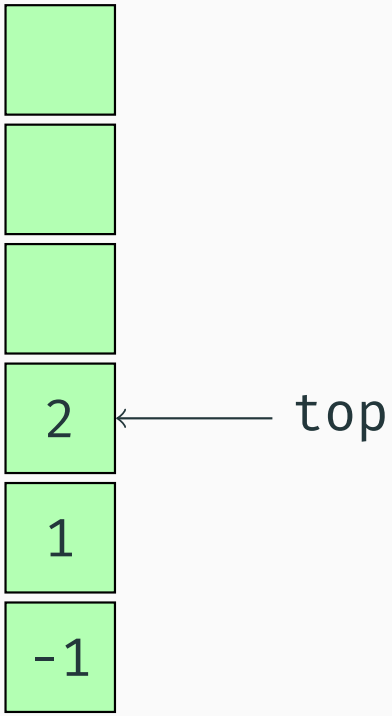
Pilha

```
push(&p, 1);
```



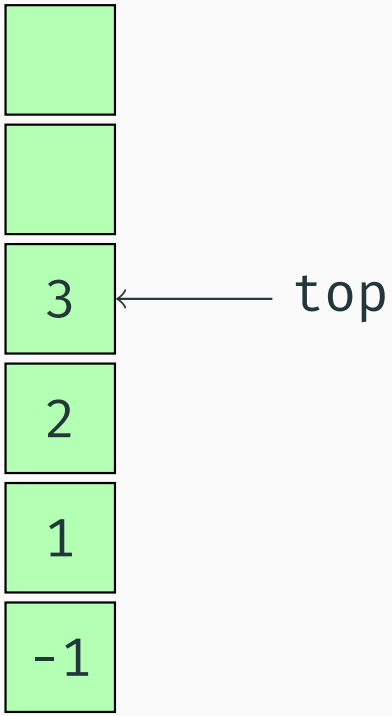
Pilha

```
push(&p, 2);
```



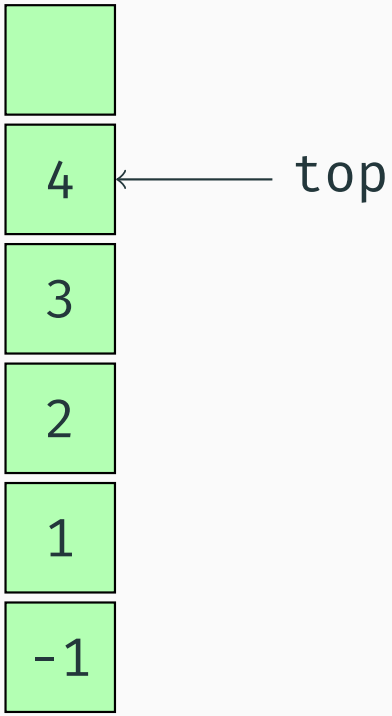
Pilha

```
push(&p, 3);
```



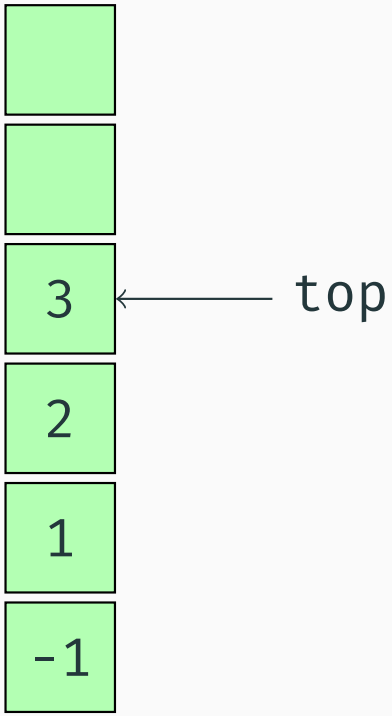
Pilha

```
push(&p, 4);
```



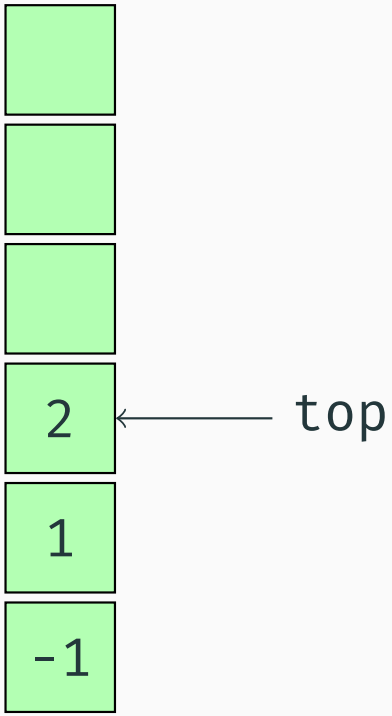
Pilha

```
pop(&p);
```



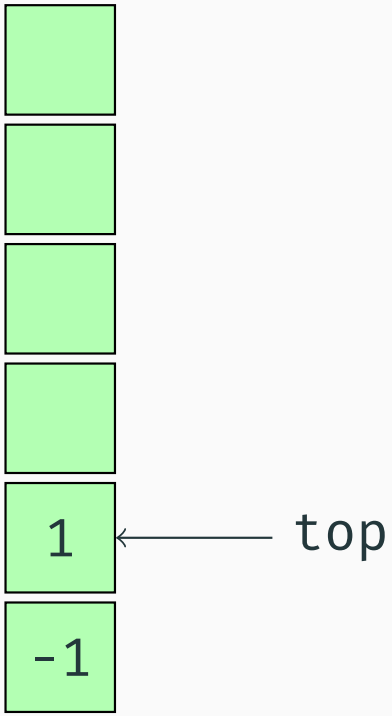
Pilha

```
pop(&p);
```



Pilha

```
pop(&p);
```



Pilha

Notem que por termos uma capacidade pré-definida as operações `push, pop` da pilha devem tomar o cuidado de não realizar operações inválidas.

Pilha

A tentativa de inserir um registro além da capacidade de uma estrutura causa o *overflow*, a tentativa de remover um registro de uma estrutura vazia causa o *underflow*.

Essas duas condições são tratadas no código retornando NULL, representando falha na operação.

Pilha

Nas pilhas, as operações 1, 2 e 3 ocorrem apenas no elemento $n - 1$.

Pilha

As operações 4, 5, 8 e 9, embora possíveis de serem implementadas, não fazem parte do conjunto de operações em pilhas. Para essas operações outras estruturas são mais interessantes.

Pilha

O tamanho da pilha (operação 7) é facilmente recuperado pelo campo `top`:

```
int size_stack(stack p) {
  return p.top + 1;
}
```

Pilha

Finalmente, a cópia de uma pilha (operação 6) é uma simples cópia da estrutura:

```
stack copy_stack(stack p) {
  stack new_p;
  new_p.top = p.top;
  memcpy(new_p.data, p.data,
         sizeof(STACK_TYPE)*CAPACITY);
  return new_p;
}
```

Fila

A **Fila** em uma estrutura sequencial é similar a pilha, porém temos dois apontadores, um para o começo e outro para o final da fila:

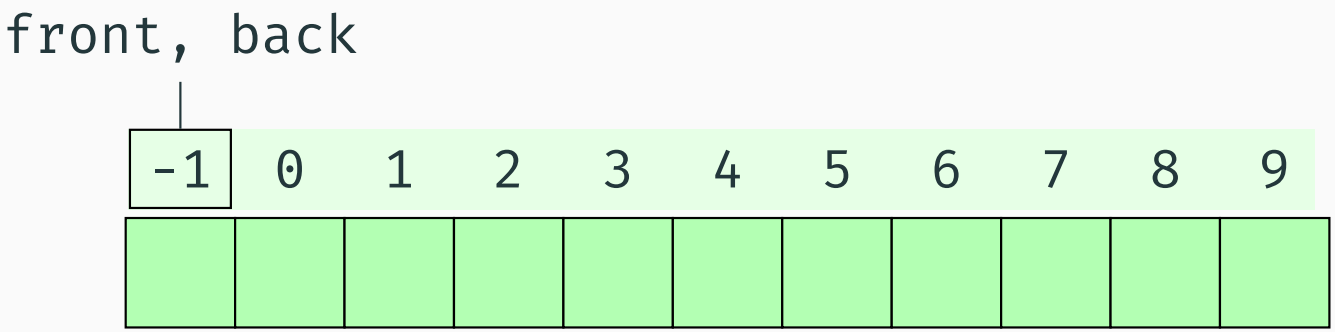
```
typedef struct queue {
    QUEUE_TYPE data[CAPACITY];
    int front;
    int back;
} queue;
```

Fila

A inicialização de uma fila define os apontadores como −1, sinalizando fila vazia:

```
queue create_queue() {
  queue p;
  p.front = p.back = -1;
  return p;
}
```

Fila



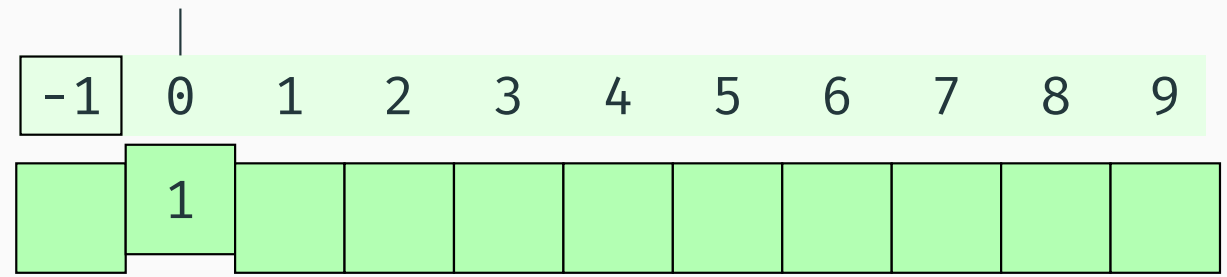
Fila

Para inserir um elemento na fila, apontamos o campo **back** para uma posição adiante e inserimos o novo registro nessa posição.

```
queue * push(queue * p, QUEUE_TYPE x) {
  if ( p->back + 1 >= CAPACITY ) return NULL;

  if (p->front==-1) p->front=0;
  p->back = p->back + 1;
  p->data[p->back] = x;
  return p;
}
```

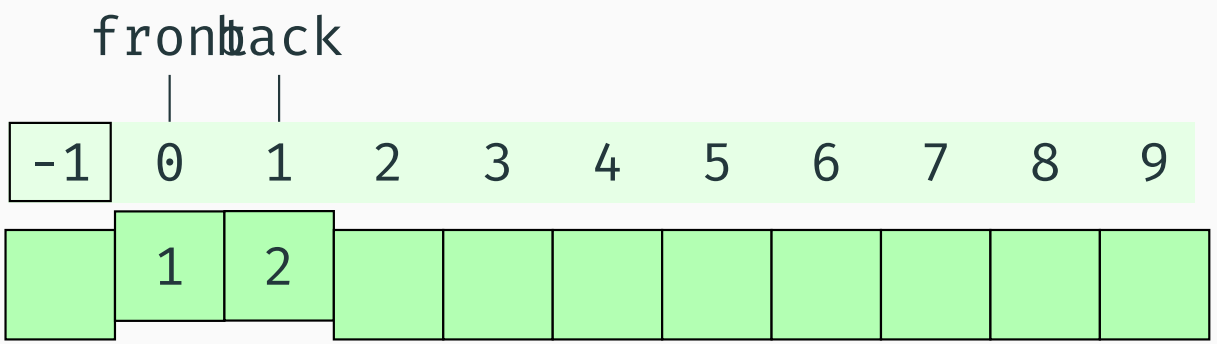
Fila

```
push(&p, 1);
```



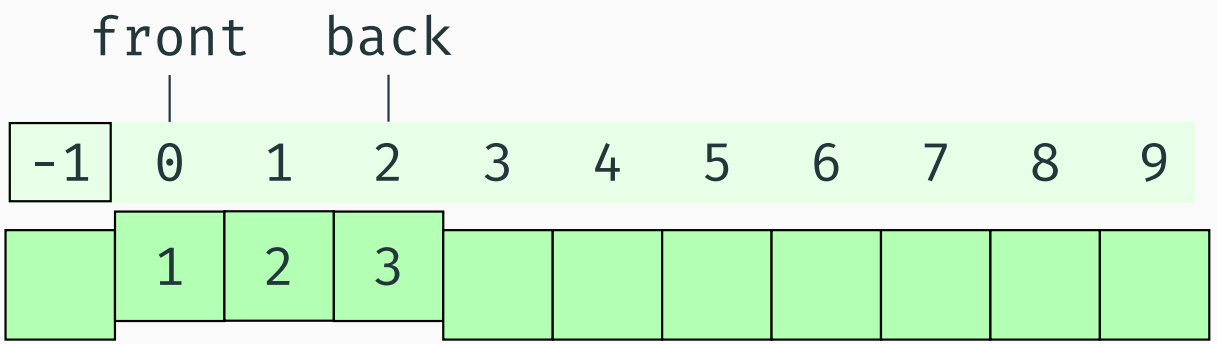
Fila

```
push(&p, 2);
```



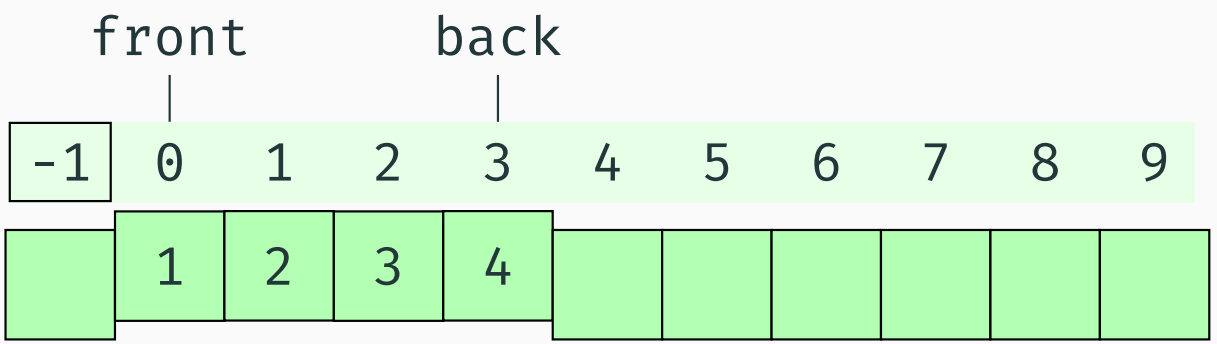
Fila

```
push(&p, 3);
```



Fila

```
push(&p, 4);
```



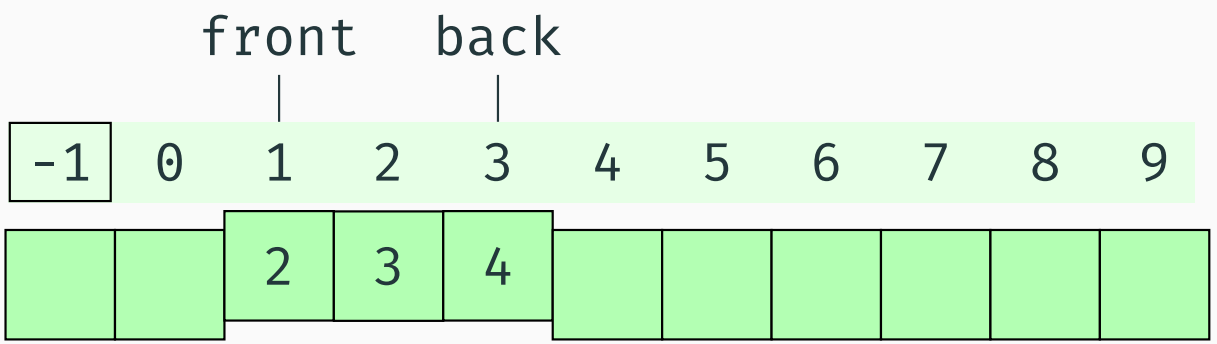
Fila

A remoção de um elemento simplesmente move o campo `front` um passo adiante.

```
QUEUE_TYPE * pop(queue * p) {
    int f = p->front;

    if (f == -1) return NULL;

    if (p->front==p->back) p->front=p->back=-1;
    else p->front = p->front + 1;
    return &(p->data[f]);
}
```

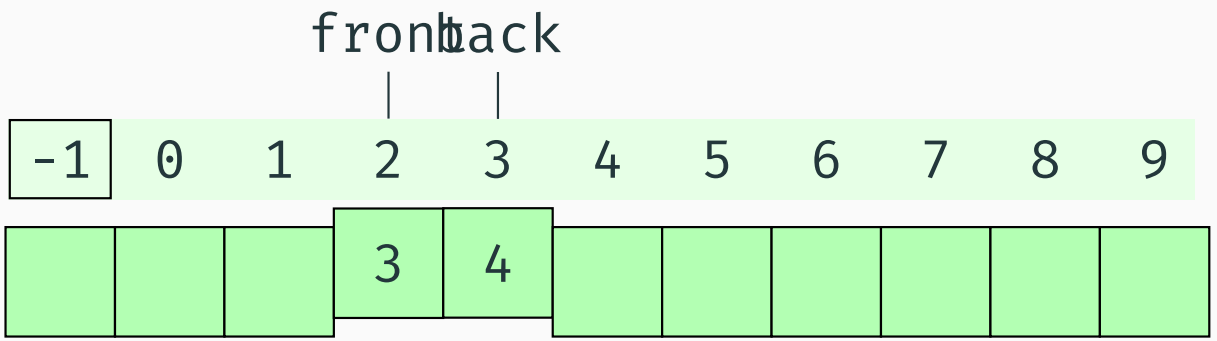
Fila

```
pop(&p);
```



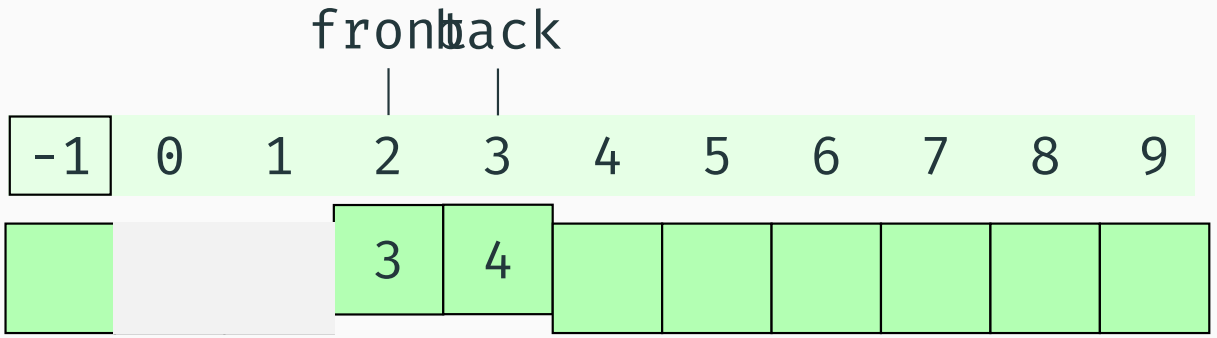
Fila

```
pop(&p);
```



Fila

Um problema com essa implementação é que a cada elemento removido, a fila perde a capacidade em uma unidade. A área em cinza não poderá mais ser utilizada!



Fila

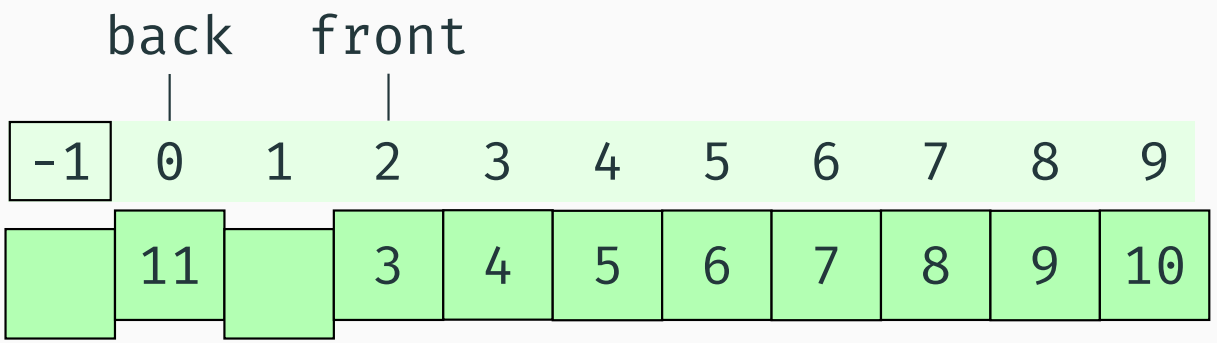
Para resolver esse problema, podemos definir o movimento dos ponteiros como circular:

```
queue * push(queue * p, QUEUE_TYPE x) {
  if ( (p->back + 1)%CAPACITY == p->front) return NULL;

  if (p->front==-1) p->front=0;
  p->back = (p->back + 1)%CAPACITY;
  p->data[p->back] = x;
  return p;
}
```

Fila

```
QUEUE_TYPE * pop(queue * p) {
    int f = p->front;

    if (f == -1) return NULL;

    if (p->front==p->back) p->front=p->back=-1;
    else p->front = (p->front + 1)%CAPACITY;
    return &(p->data[f]);
}
```

Fila



Próxima Aula

Aprenderemos sobre a alocação por **lista ligada**.