```
int main(int argc, char** argv)
{
  char c = 0;
  char* commands = "ads pq"; // key commands: "left,right,rotate,confirm,pause,quit"
  int speed = 2; // sets max moves per row
  int moves_to_go = 2;
  int full = 0; // whether board is full
  init(); // initialize board an tetrominoes


  cur =
  state[
  state[

  system
  printf

  while
    {
      dr

    // process user action
    c = getchar();   // get new action
    if (c == commands[0] && !intersect(cur, state[0]-1, state[1])) state[0]--; // move left
    if (c == commands[1] && !intersect(cur, state[0]+1, state[1])) state[0]++; // move right
    if (c == commands[2] && !intersect(cur->rotated, state[0], state[1])) cur = cur->rotated;
    if (c == commands[3]) moves_to_go=0;

    // scroll down
    if (!moves_to_go--)
      {
        if (intersect(cur,state[0],state[1]+1)) // if tetromino intersected with sth
          {
            cramp_tetromino();
            remove_complete_lines();
            cur = &tetrominoes[rand() % NUM_POSES];
            state[0] = (WIDTH - cur->width)/2;
```

MAC122 - PRINCÍPIOS DE DESENVOLVIMENTO DE ALGORITMOS

Pilhas

Revisão: Alocação estática de memória

`int n; char c, int v[30], char str[100];`

Aloca bloco de memória no espaço de pilha do programa; tamanho a ser alocado é determinado por tipo; desalocação é gerenciada pelo compilador após saída de escopo.

Mais eficiente e segura que alocação dinâmica, mas não permite:

```
int n;
scanf("Quantas letras? %d", &n);
/* erro de compilação */
char s[n+1];
printf("Palavra: ");
fgets(s, n, stdin);
```

```
int* cria_vetor(int x, int n) {
   int i, v[100];
   for (i=0; i < n; i++)
      v[i] = x;
   /* devolve ponteiro para bloco
     alocado */
   return v;
}
int main() {
 int *v;
 /* valor de v é indefinido */
 v = cria_vetor(5,10);
 return 0;
}
```

Revisão: Alocação dinâmica de memória

```
void *malloc (unsigned int n);
```

Aloca um bloco de `n` *bytes* consecutivos e retorna um ponteiro para o espaço alocado ou `NULL` em caso de erro

```
void free (void *ptr);
```

Libera o espaço de memória endereçado por `ptr`

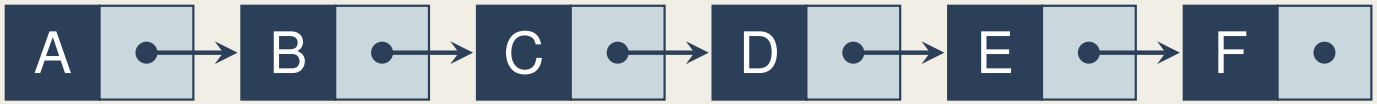
Definidos na biblioteca `stdlib.h`

```
int n;
scanf("Quantas letras? %d", &n);
char* s = malloc(n+1);
printf("Palavra: ");
fgets(s, n, stdin);
/* ... */
free(s);
```

```
int* cria_vetor(int x, int n) {
  int* v = malloc(n*sizeof(int));
  for (int i=0; i < n; i++)
     v[i] = x;
  return v;
}
int main() {
 int *v = cria_vetor(5,10);
 /* ... */
 free(v)
 return 0;
}
```

Listas encadeadas: Definição

Estrutura de dados *dinâmicas* e *autoreferenciais* usadas para representar sequências eficientemente editáveis



- Elementos são organizados em células
- Flechas são representados por ponteiros

```c
typedef struct celula_st {
    char valor;                 /* conteúdo */
    struct celula_st *prox;     /* para próxima célula */
} celula; /* tipo de dado */
```

Listas encadeadas sem cabeça

Primeira célula armazena conteúdo do primeiro elemento


```
1 celula *c1 = malloc(sizeof(celula));
2 celula *c2 = malloc(sizeof(celula));
3 celula *c3 = malloc(sizeof(celula));
4 c1->valor = 'A'; c1->prox = c2;
5 c2->valor = 'B'; c2->prox = c3;
6 c3->valor = 'C'; c3->prox = NULL;
```

Lista vazia é representada por ponteiro nulo: `c1 = NULL`

Listas encadeadas com cabeça

Segunda célula armazena conteúdo do primeiro elemento


```
1 celula *cabeca = malloc(sizeof(celula));
2 celula *c1 = malloc(sizeof(celula));
3 celula *c2 = malloc(sizeof(celula));
4 celula *c3 = malloc(sizeof(celula));
5 cabeca->prox = c1;
6 c1->valor = 'A'; c1->prox = c2;
7 c2->valor = 'B'; c2->prox = c3;
8 c3->valor = 'C'; c3->prox = NULL;
```

Lista vazia é representada por cabeça apontando para nulo:

`cabeca->prox = NULL`

Tipo de dados nativos

Conjunto de valores com operações pré-definidas (pela linguagem)

- `int`: conjunto `INT_MIN`,...,`INT_MAX` com operações aritméticas (+,−,/) e de comparação (<,==)
- `double`: conjunto de valores reais definidos pela lógica de ponto flututante equipado de operações aritméticas e comparações
- vetor: conjunto de variáveis do mesmo tipo organizadas de forma sequencial; operações de leitura/escrita em posição arbitrária

Tipos de dados definidos por usuário

São formados a partir de tipos mais básicos

Exemplos:

- `string`: conjunto de sequências de caracteres munido de operações de concatenação, busca, comparação etc.
- `typedef struct { int n, d; } racional;`: conjunto de racionais munidos de operações aritméticas e comparações

Tipo abstrato de dados (TAD)

Tipo de dados especificado pelo comportamento (em oposição a pela forma como é implementado)

Exemplos:

- Racional: valores do tipo p/q, onde p e q são inteiros com mdc igual 1, e com $q > 0$; admitem soma, multiplicação, subtração e divisão
- Conjuntos: coleção de valores do mesmo tipo, com operações de pertinência, inserção e remoção

Estrutura de dados

Modo específico de armazenar e acessar informação (em um computador)

- `struct racional` é uma estrutura de dados (organiza um racional p/q em um registro)
- Tipo `racional` e operações especificadas formam um tipo abstrato de dado

Um TAD pode ser implementado usando diferentes estruturas de dados (quando chamamos um tipo de dados concreto)

Cuidado!

Operações elementares como aritmética e comparação não estão definidas para tipos definidos por usuário

```
1 typedef struct { int x, y } ponto;
2 ponto a = {1, 2}, b = {3, 4};
3 if (a == b || a < b) { ... } /* erro - indefinido */
4 ponto c = a + b; /* erro - indefinido */
```

Tipo abstrato de dados: Interface

São especificados através de uma interface; cliente não deve acessar/modificar dados diretamente

```
1 typedef struct { int num, den; } racional;
2 void cria       (int num, int den, racional *r);
3 void multiplica (racional *p, racional *q, racional *r);
4 void soma       (racional *p, racional *q, racional *r);
5 void divide     (racional *p, racional *q, racional *r);
6 void subtrai    (racional *p, racional *q, racional *r);
7 int  menor      (racional *p, racional *q);
```

Interface descreve uso da função sem se atentar à implementação; em geral, deve ser possível alterar a implementação sem mudar o comportamento da função

Exemplo: TAD Conjuntos

- sequência *não-ordenada* de elementos do mesmo tipo (nativo) de dado

- equipado com operações de pertinência, inserção e remoção por posição arbitrária

Implementações

1. Usando um vetor desordenado
2. Usando um vetor crescente
3. Usando uma lista encadeada desordenada
4. Usando uma lista encadeada crescente

Vantagens

- Clientes não precisam se preocupar com detalhes de implementação; clientes podem escolher entre muitas implementações
- Programa continua válido se implementação for alterada
- Permite criação de código modular e mais reutilizável

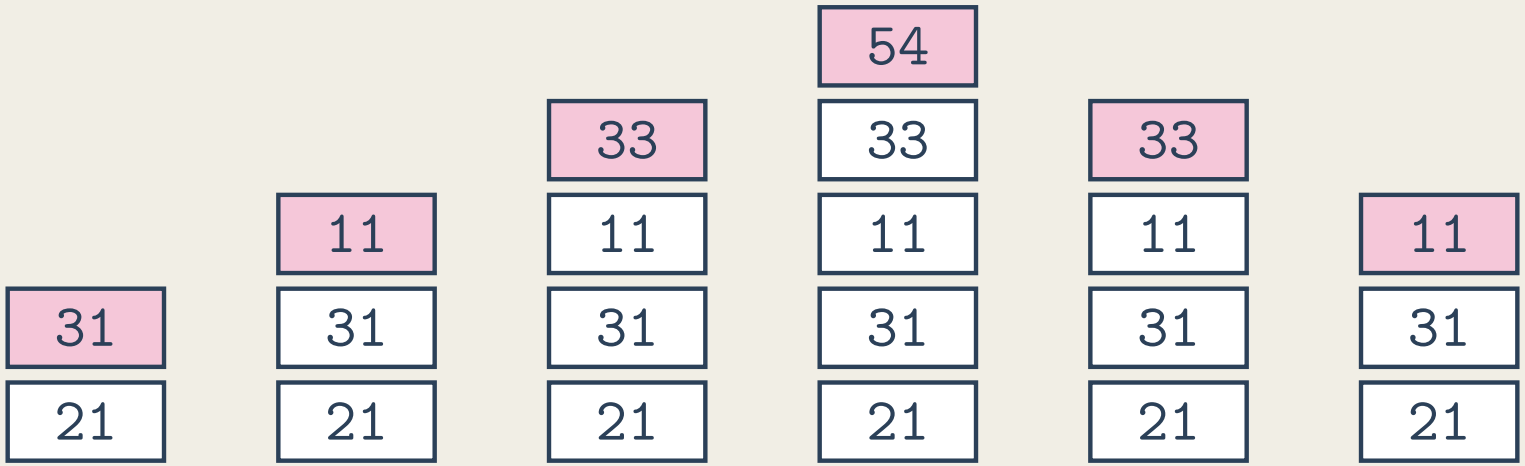
Desvantagens

- Clientes podem utilizar implementação ineficiente para uma fim; devem conhecer implementação para garantir eficiência
- Implementação não pode assumir uso de estruturas de dados
- Sobrecarga no encapsulamento (chamadas de funções desnecessárias e estruturas de dados mais complexas)

Pilha: Definição

Tipo de dados abstrato para manipular conjuntos ordenados de objetos acessados/removidos em ordem inversa de inserção (LIFO: *last-in first-out*)

- Empilhar (*push*): Insere elemento no topo
- Desempilhar (*pop*): Remove elemento do topo
- Topo (*peek*): Devolve elemento no topo
- ...



Pilha: Interface

```
1  /* pilha.h */
2  typedef int elem;          /* Tipo do elemento da pilha */
3  typedef struct pilhaTCD *pilha; /* Estrutura de dados
4                                  (depende de implementação) */
5  pilha      CriaPilha (void);  /* Cria nova pilha vazia */
6  void       DestroiPilha (pilha p);    /* Destrói pilha */
7  void       Empilha (pilha p, elem x);     /* Empilha x */
8  elem       Desempilha (pilha p); /* Remove e ret. topo */
9  int        TamanhoPilha (pilha p); /* Tamanho da pilha */
10 int        PilhaVazia (pilha p); /* Pilha esta' vazia? */
11 int        PilhaCheia (pilha p); /* Pilha esta' cheia? */
12 elem TopoPilha (pilha p);         /* Retorna topo */
```

Torres de Hanoi

- n discos, 3 agulhas
- Discos maior não pode ser colocados sobre disco menor
- Discos começam na agulha 1
- Objetivo: mover todos os discos para agulha 3

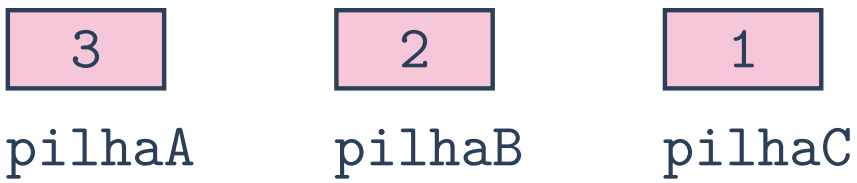
Torres de Hanoi com 3 discos: Resolução

Passo 1: `x = desempilha(pilhaA); empilha(pilhaC, x);`

Torres de Hanoi com 3 discos: Resolução

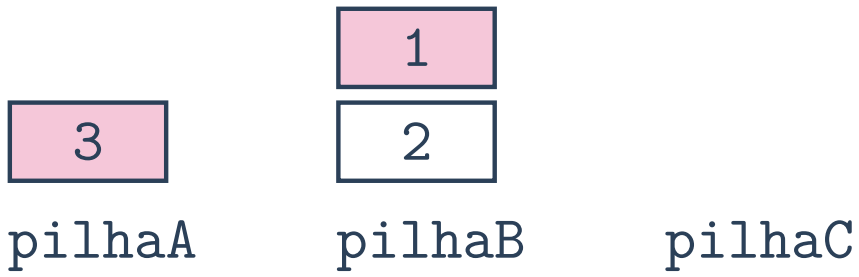
Passo 2: `x = desempilha(pilhaA); empilha(pilhaB, x);`

Torres de Hanoi com 3 discos: Resolução



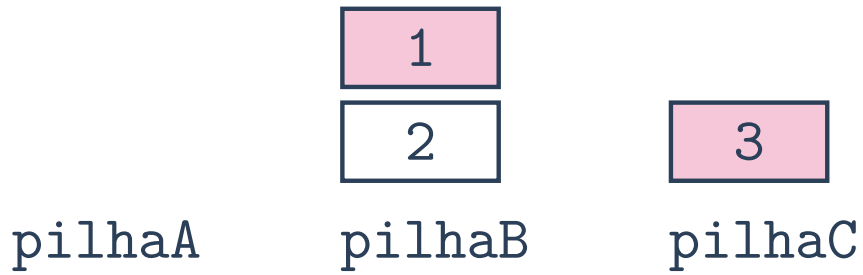
Passo 3: `x = desempilha(pilhaC); empilha(pilhaB, x);`

Torres de Hanoi com 3 discos: Resolução



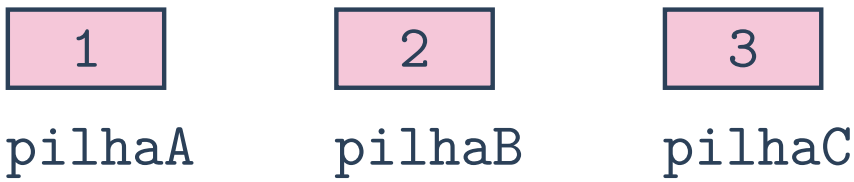
Passo 4: `x = desempilha(pilhaA); empilha(pilhaC, x);`

Torres de Hanoi com 3 discos: Resolução



Passo 5: `x = desempilha(pilhaB); empilha(pilhaA, x);`

Torres de Hanoi com 3 discos: Resolução



Passo 6: `x = desempilha(pilhaB); empilha(pilhaC, x);`

Torres de Hanoi com 3 discos: Resolução

Passo 7: `x = desempilha(pilhaA); empilha(pilhaC, x);`

Torres de Hanoi com 3 discos: Resolução

Pronto!

Casamento de parênteses

Objetivo: Encontrar parêntese de abertura de expressão atual para cada posição de string (ou *buffer* de texto)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	3	4	4	4	4	4	3	10	10	10	10	10	3	0

Casamento de parênteses

Objetivo: Encontrar parêntese de abertura de expressão atual para cada posição de string (ou *buffer* de texto)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	0	0	3	4	4	4	4	4	3	10	10	10	10	10	3	0

Algoritmo: Percorrer posições de string de `i=0` a `i=n-1`:

1. Se encontrar `(`, empilhar e escrever `i`
2. Se encontrar `)`, desempilhar e escrever `topo`
3. Para qualquer outro caractere, escrever `topo`

Casamento de parênteses: Implementação

```
1  /* pmath.c */
2  char expr[CAP] = "(1+((2+3)*(4*5)))";
3  char paren[CAP]; /* parêntese ( de expr de pos i */
4  int t;
5
6  pilha p = CriaPilha();
7
8  for (int i=0; expr[i] != '\0'; i++) {
9    if (expr[i] == '(') { Empilha(p, i); t = i; }
10   else if (expr[i] == ')') t = Desempilha(p);
11   else t = TopoPilha(p);
12   paren[i] = t;
13 }
14 DestroiPilha(p);
```

Casamento de parênteses: Expressões desbalanceadas

Como modificar algoritmo para identificar parênteses mal-casados?

Bem-casados:

0	1	2	3	4	5	6	7	8	9	10
(	1	+	(	(	2	+	3	)	)	)

0	1	2	3	4	5	6	7	8	9	10
(	+	1	(	(	2	+	3	)	)	)

Mal-casados:

0	1	2	3	4	5	6	7	8	9	10
(	+	1	(	(	2	+	3	)	(	)

0	1	2	3	4	5	6	7	8	9	10
(	+	1	4	)	+	2	)	(	5	4

0	1	2	3	4	5	6	7	8	9	10
(	+	1	4	)	+	2	)	5	4	1

Casamento de parênteses: Expressões desbalanceadas

Como modificar algoritmo para identificar parênteses mal-casados?

Algoritmo: Percorra posições de string de `i=0` a `i=n-1`:

1. Se encontrar `(`, empilhe e escreva `i`
2. Se encontrar `)`, desempilhe `topo`; se `s[topo]` não for `(` ou pilha estiver vazia retorne `mal-casado`
3. Para qualquer outro caractere, escreva `topo`

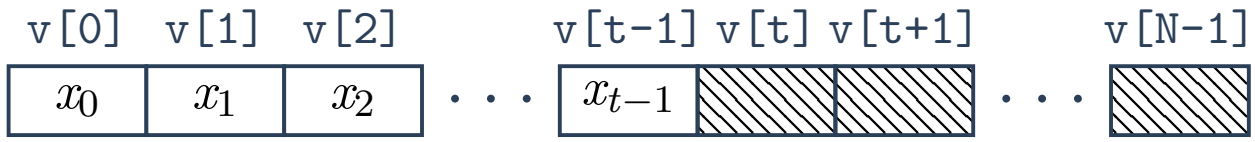
Se ao terminar, pilha não estiver vazia, retorne `mal-casado`

Pilhas: Implementação em vetor de tamanho fixo

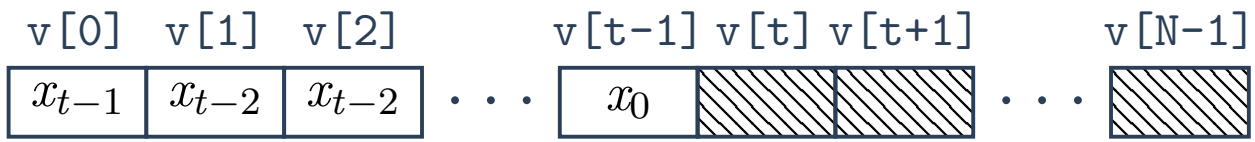
Como implementar pilha para que inserção/remoção/consulta sejam feitas em tempo constante (independentemente do tamanho do vetor)?

A) Não pode ser feito

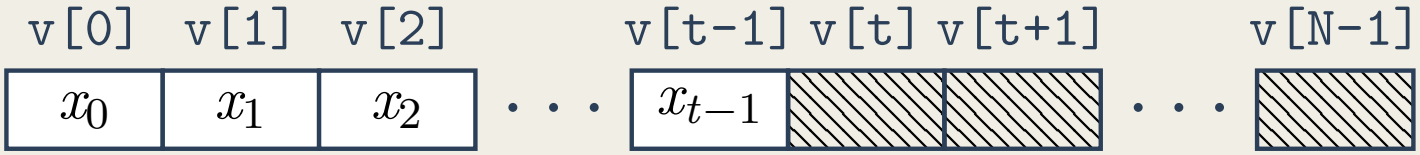
B)



C)

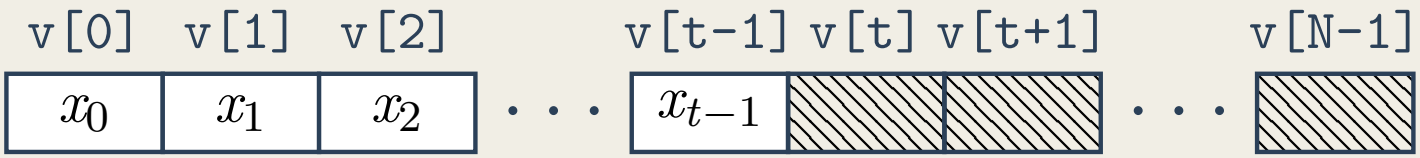


Pilhas: Implementação em vetor dinâmico



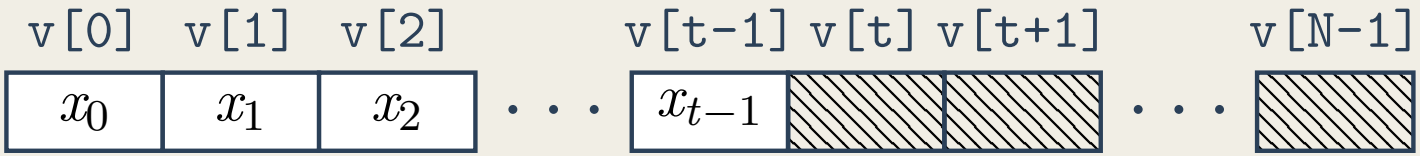

```
1  /* pilha.c */
2  struct pilhaTCD {
3    elem *vetor;
4    int topo; /* índice do topo */
5    int cap; /* N */
6  }
7
8  int PilhaVazia(pilha p) {
9    return (p->topo == 0);
10 }
11
12 elem TopoPilha(pilha p) {
13   return p->vetor[p->topo-1];
14 }
```

Pilhas: Implementação em vetor dinâmico



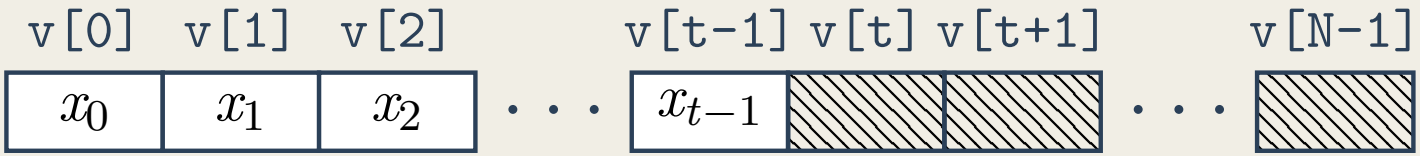

```
1  pilha CriaPilha() {
2    pilha p = malloc(sizeof(pilhaTCD));
3    p->topo = 0; p->cap = CAP0;
4    p->vetor = malloc(p->cap*sizeof(elem));
5    return p;
6  }
7
8  void DestroiPilha(pilha p) {
9    free(p->vetor); free(p)
10 }
```

Pilhas: Implementação em vetor dinâmico




```
1 void Empilha(pilha p, elem x) {
2   if (p->topo == p->cap) {
3     // sobrecarga
4   }
5   p->vetor[p->topo++] = x;
6 }
```

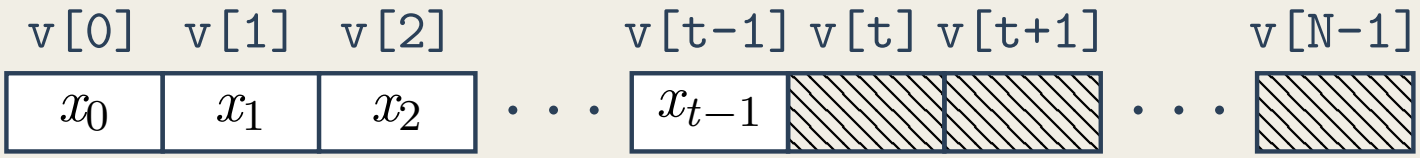
Pilhas: Implementação em vetor dinâmico




```
1 void Empilha(pilha p, elem x) {
2   if (p->topo == p->cap) {
3     // sobrecarga
4   }
5   p->vetor[p->topo++] = x;
6 }
```

Sobrecarga: Inserir elemento em pilha cheia

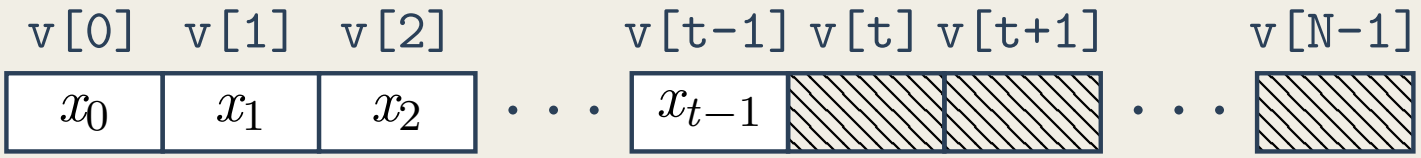
Pilhas: Implementação em vetor dinâmico




```
1 void Empilha(pilha p, elem x) {
2   if (p->topo == p->cap) {
3     p->cap *= 2; /* dobramos capacidade */
4     p->vetor = realloc(p->vetor, p->cap*sizeof(elem));
5   }
6   p->vetor[p->topo++] = x;
7 }
```

Custo computacional (quando sobrecarregado)?

Pilhas: Implementação em vetor dinâmico




```
1 void Empilha(pilha p, elem x) {
2   if (p->topo == p->cap) {
3     p->cap *= 2; /* dobramos capacidade */
4     p->vetor = realloc(p->vetor, p->cap*sizeof(elem));
5   }
6   p->vetor[p->topo++] = x;
7 }
```

Custo computacional (quando sobrecarregado)? Proporcional a `p->topo`

Pilhas: Implementação em vetor dinâmico

Por que duplicar capacidade de vetor cheio? Por que não aumentar por tamanho constante (p.ex. 1)?

```
1 void Empilha(pilha p, elem x) {
2   if (p->topo == p->cap) {
3     p->cap *= 2;
4     p->vetor = realloc(p->vetor, p->cap*sizeof(elem));
5   }
6   p->vetor[p->topo++] = x;
7 }
```

Pilhas: Implementação em vetor dinâmico

Por que duplicar capacidade de vetor cheio? Por que não aumentar por tamanho constante (p.ex. 1)?

```
1 void Empilha(pilha p, elem x) {
2   if (p->topo == p->cap) {
3     p->cap *= 2;
4     p->vetor = realloc(p->vetor, p->cap*sizeof(elem));
5   }
6   p->vetor[p->topo++] = x;
7 }
```

Dobrando capacidade: Suponha que `CAP0=1` e $N = 2^i$ inserções são feitas; custo de copiar elementos (para realocar memória) é

$$\underbrace{1 + 2 + 4 + 8 + \ldots + N/2}_{i-1 \text{ termos}} \sim N$$

Pilhas: Implementação em vetor dinâmico

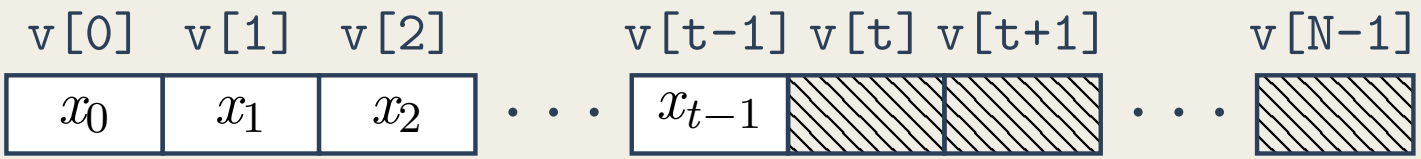
Por que duplicar capacidade de vetor cheio? Por que não aumentar por tamanho constante (p.ex. 1)?

```
1 void Empilha(pilha p, elem x) {
2   if (p->topo == p->cap) {
3     p->cap *= 2;
4     p->vetor = realloc(p->vetor, p->cap*sizeof(elem));
5   }
6   p->vetor[p->topo++] = x;
7 }
```

Aumento por constante: Agora suponha que aumentássemos a capacidade por 1 a cada nova inserção; então custo seria

$$1 + 2 + 3 + 4 \ldots + N - 1 \sim N^2$$

Pilhas: Implementação em vetor dinâmico




```
1 elem Desempilha(pilha p) {
2     if (p->topo <= p->cap/4) {
3       p->cap /= 2;
4       p->vetor = realloc(p->vetor, p->cap*sizeof(elem));
5     }
6     return p->vetor[--p->topo];
7 }
```

Pilhas: Implementação em vetor dinâmico

Por que meiar capacidade apenas com 1/4 de vetor cheio? Por que não reduzir quando meio cheio/vazio?

```
1 elem Desempilha(pilha p) {
2     if (p->topo <= p->cap/4) {
3       p->cap /= 2;
4       p->vetor = realloc(p->vetor, p->cap*sizeof(elem));
5     }
6     return p->vetor[--p->topo];
7 }
```

Pilhas: Implementação em vetor dinâmico

Por que meiar capacidade apenas com 1/4 de vetor cheio? Por que não reduzir quando meio cheio/vazio?

```
1 elem Desempilha(pilha p) {
2     if (p->topo <= p->cap/4) {
3       p->cap /= 2;
4       p->vetor = realloc(p->vetor, p->cap*sizeof(elem));
5     }
6     return p->vetor[--p->topo];
7 }
```

Considere sequência N de alternâncias de inserção/remoção para vetor cheio (tamanho n); inserção dobra capacidade (custo n), remoção meia capacidade (custo n) $\Rightarrow$ custo total $= Nn$ (proporcional a tamanho)

Pilhas: Implementação em vetor dinâmico

Por que meiar capacidade apenas com 1/4 de vetor cheio? Por que não reduzir quando meio cheio/vazio?

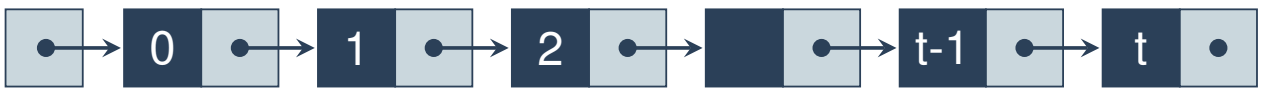
```
1 elem Desempilha(pilha p) {
2     if (p->topo <= p->cap/4) {
3       p->cap /= 2;
4       p->vetor = realloc(p->vetor, p->cap*sizeof(elem));
5     }
6     return p->vetor[--p->topo];
7 }
```

Teorema: Para estratégia adotada (meiar quando 1/4 cheio), qualquer sequência de N inserções e remoções toma tempo *amortizado* proporcional a N (e constante no tamanho do vetor) – pior caso ainda proporcional a tamanho

Pilhas: Implementação em lista encadeada

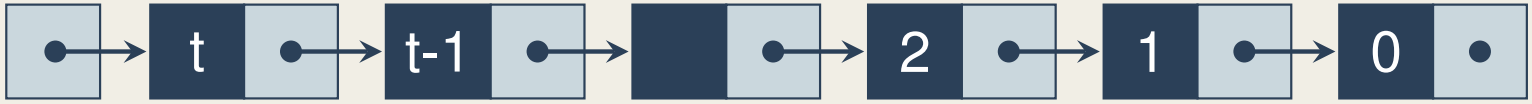
Como implementar pilha para que inserção/remoção/consulta sejam feitas em tempo constante (independentemente do tamanho da lista)?

A) Não pode ser feito

B) 

C) 

Pilhas: Implementação em lista encadeada




```
1  struct celula {
2    elem valor;
3    struct celula *prox;
4  };
5
6  typedef struct celula *lista;
7
8  struct pilhaTCD {
9    lista topo; /* cabeça da lista */
10   int tamanho;
11 }
```

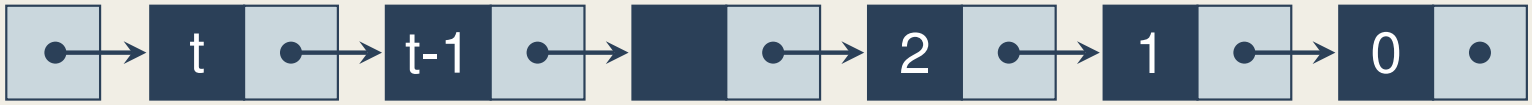
Pilhas: Implementação em lista encadeada


```
1 pilha CriaPilha () {
2  pilha p = (pilha) malloc (sizeof(pilhaTCD));
3  p->topo = malloc(sizeof(struct celula));
4  p->topo->prox = NULL;
5  p->tamanho  = 0;
6  return p;
7 }
```

Pilhas: Implementação em lista encadeada


```
1 int PilhaVazia(pilha p) {
2   return (p->tamanho == 0);
3 }
4
5 elem TopoPilha(pilha p) {
6   return p->topo->prox->valor;
7 }
```

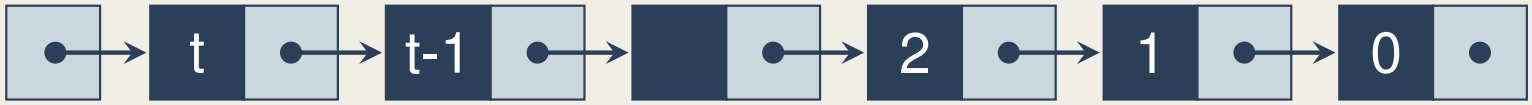
Pilhas: Implementação em lista encadeada




```
1 void Empilha (pilha p, elem x) {
2  lista n = (lista) malloc (sizeof(struct celula));
3  n->valor = x;
4  n->prox = p->topo->prox;
5  p->topo->prox = n;
6  p->tamanho++;
7 }
```

Complexidade de pior caso?

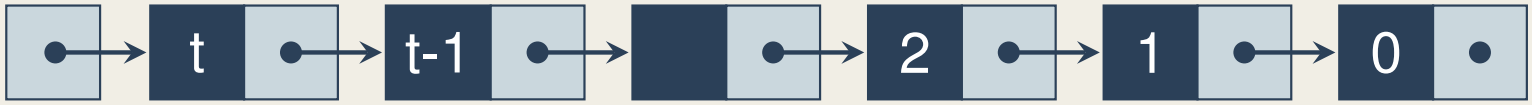
Pilhas: Implementação em lista encadeada




```
1 void Empilha (pilha p, elem x) {
2  lista n = (lista) malloc (sizeof(struct celula));
3  n->valor = x;
4  n->prox = p->topo->prox;
5  p->topo->prox = n;
6  p->tamanho++;
7 }
```

Complexidade de pior caso? Constante (em tamanho da lista)

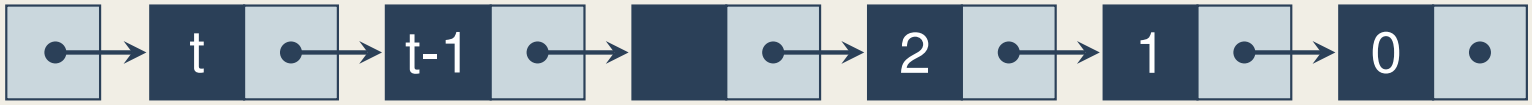
Pilhas: Implementação em lista encadeada




```
1 elem Desempilha (pilha p) {
2   lista c = p->topo->prox;
3   elem x = c->valor;
4   p->topo->prox = c->prox;
5   free(c)
6   p->tamanho--;
7   return x;
8 }
```

Complexidade de pior caso?

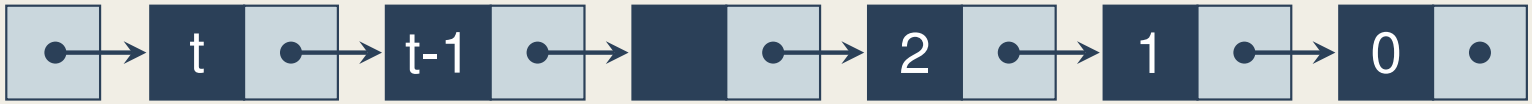
Pilhas: Implementação em lista encadeada




```
1 elem Desempilha (pilha p) {
2   lista c = p->topo->prox;
3   elem x = c->valor;
4   p->topo->prox = c->prox;
5   free(c)
6   p->tamanho--;
7   return x;
8 }
```

Complexidade de pior caso? Constante (em tamanho da lista)

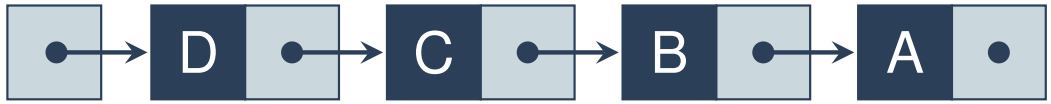
Pilhas: Implementação em lista encadeada




```
1 void DestroiPilha (pilha p) {
2   while(!PilhaVazia(p)) Desempilha(p);
3   free(p->topo);
4   free(p);
5 }
```

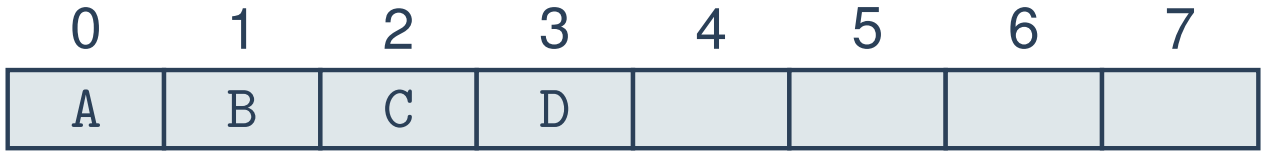
Comparação de implementações

Lista encadeada



- Pior caso constante
- Espaço de memória extra (para ligações entre células)

Vetor dinâmico



- Tempo amortizado constante (pior caso linear)
- Menos espaço de memória (e acesso otimizado para elementos de tipos simples como inteiros)

Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

Algoritmo de 2 pilhas de Dijkstra

Criar pilha `operandos` e pilha `operações`

Percorrer símbolos `e[i]` da esquerda para direita:

- Se `e[i]` é número: empilhe em `operandos`
- Se `e[i]` é operação: empilhe em `operações`
- Se `e[i]` é `(`: não faça nada
- Se `e[i]` é `)`: desempilhe operador e dois valores, empilhe resultado em `operandos`

Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

operandos operações

Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

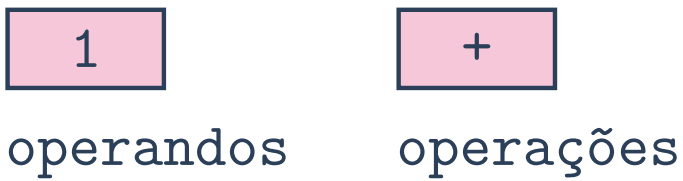
```
(1+((2+3)*(4*5)))
```

operandos operações

Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

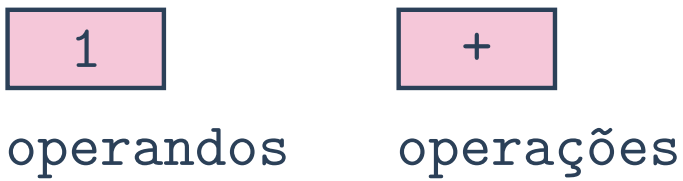
```
(1+((2+3)*(4*5)))
```



Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

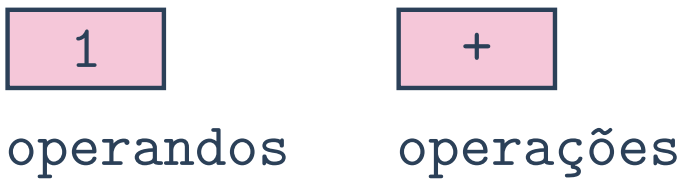
```
(1+((2+3)*(4*5)))
```



Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

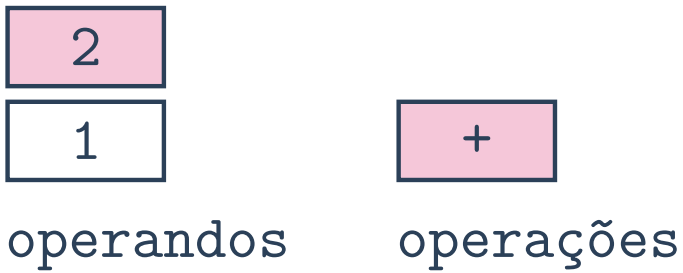


Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

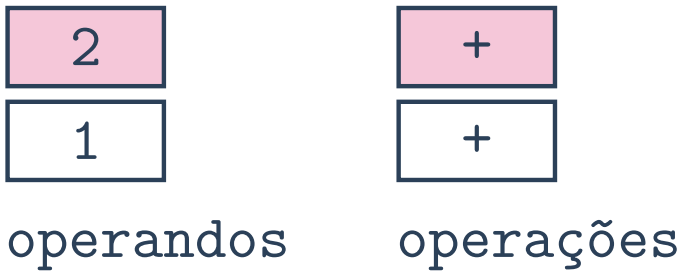


Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

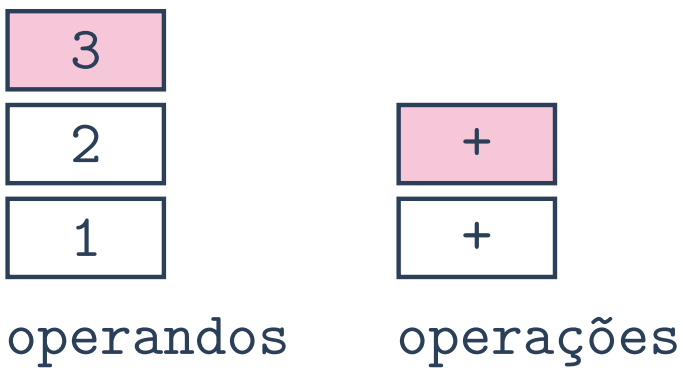


Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

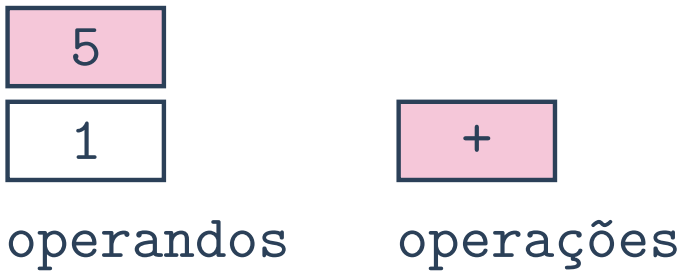
(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)



Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

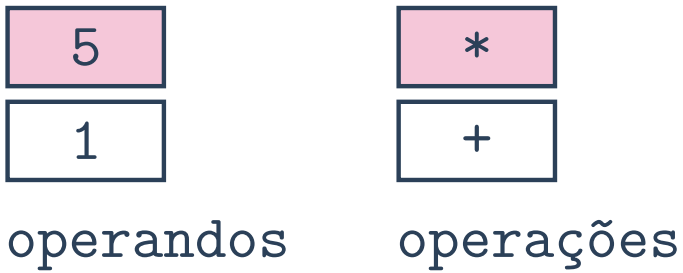
```
(1+((2+3)*(4*5)))
```



Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

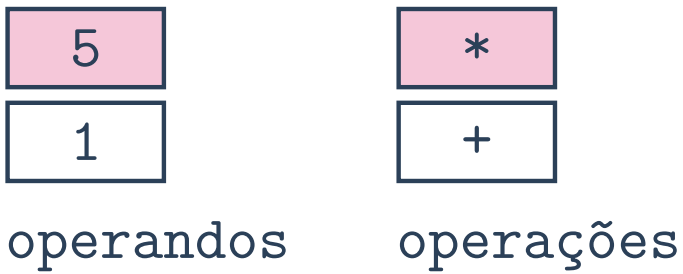
```
(1+((2+3)*(4*5)))
```



Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

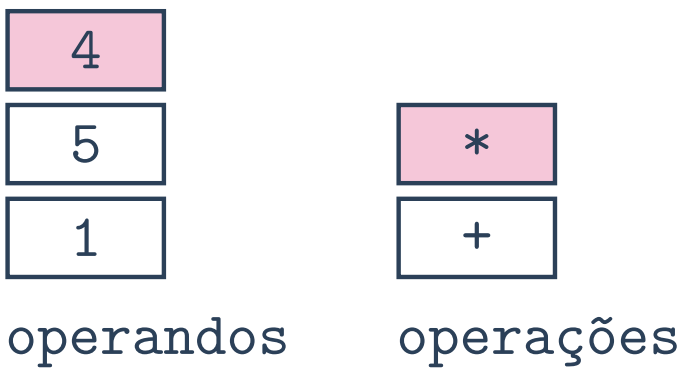


Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

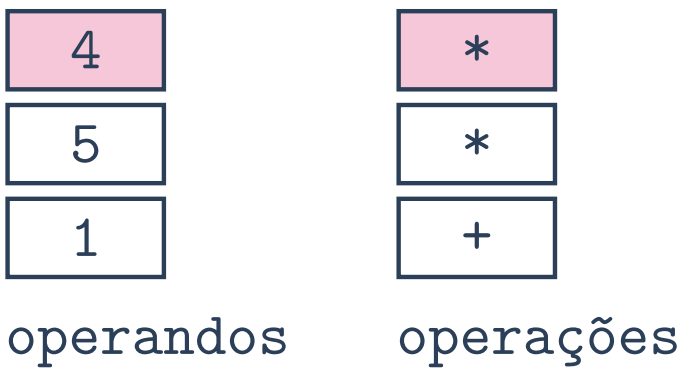


Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

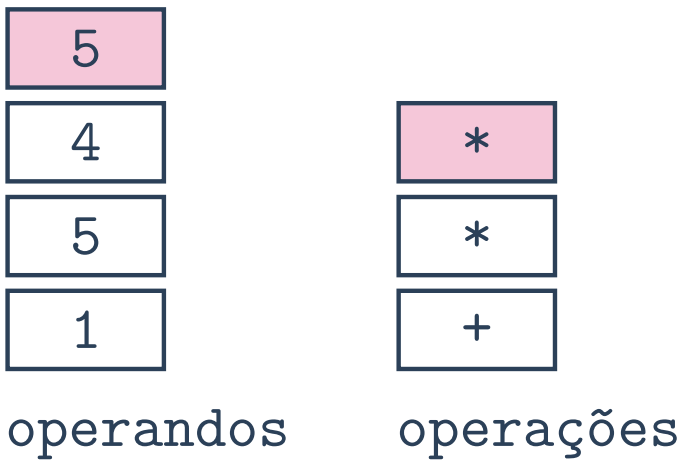


Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)

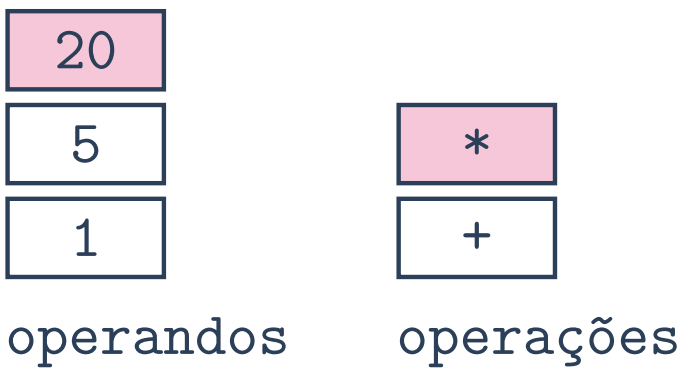


Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

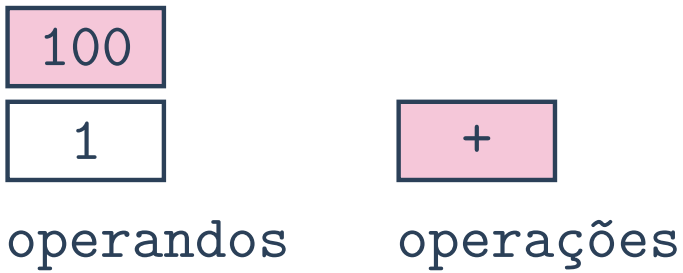
(	1	+	(	(	2	+	3	)	*	(	4	*	5	)	)	)



Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```



Aplicação: Avaliação de expressões aritméticas

Calcular expressão aritmética bem-formada, ex:

```
(1+((2+3)*(4*5)))
```

Aplicação: Avaliação de expressões aritméticas

Corretude: Quando algoritmo encontra `)`, ele avalia expressão dentro do parênteses associado, e a substitui por seu resultado na pilha de operandos:

$$\texttt{(1+((2+3)*(4*5)))} \Rightarrow \texttt{(1+(5*(4*5)))}$$

Fato: O algoritmo tem o mesmo comportamento se expressão estiver em notação pósfixa (polonesa reversa):

$$\texttt{(1((2 3 +)(4 5 *)*)+)}$$

$\Rightarrow$ parênteses são redundantes para essa notação!

$$\texttt{1 2 3 + 4 5 * * +}$$

Obs: interface adotada por calculadoras comuns

Para casa

Exercícios 8A, 8B e 8C (desafio!)