Programação Estruturada
Prof. Rodrigo Hausen
http://progest.compscinet.org

Agregados de Dados Heterogêneos (structs)

AGREGADO HOMOGÊNEO

Um agregado **homogêneo** de dados é um conjunto de dados que são **necessariamente** do mesmo tipo, associados a um **mesmo identificador**.

Arrays e matrizes são agregados homogêneos. P. ex.:

```
float notaP1[n];
```

Cada elemento *notaP1*[i] tem o tipo **float** e é associado ao mesmo identificador *notaP1*.

AGREGADO HETEROGÊNEO

Um agregado **heterogêneo** de dados é um conjunto de dados **não necessariamente** do mesmo tipo, associados a um **mesmo identificador**.

Exemplo: os dados associados a um aluno matriculado em uma disciplina são RA (inteiro), nome (array de char), número de faltas (inteiro), nota na P1 (float) e nota na P2 (float).

STRUCT

Para definir um agregado heterogêneo em C, precisamos definir uma **struct**.

```
struct aluno {
   int ra;
   char nome[61];
   int faltas;
   float notaP1;
   float notaP2;
};
```

A declaração acima define um agregado de dados chamado **struct aluno** com cinco *membros*, chamados ra, nome, faltas, notaP1 e notaP2.

```
struct aluno {
   int ra;
   char nome[61];
   int faltas;
   float notaP1;
   float notaP2;
};
```

declara novo tipo agregado de dados chamado `struct aluno`

```
struct aluno {
   int ra;
   char nome[61];
   int faltas;
   float notaP1;
   float notaP2;
};

struct aluno clarice;
```

declara variável de nome clarice *com tipo* `struct aluno`

```
struct aluno {
   int ra;
   char nome[61];
   int faltas;
   float notaP1;
   float notaP2;
};

struct aluno clarice;

clarice.ra = 10012235;
```

acesso ao membro
`ra` *da variável* `clarice`

```
struct aluno {
   int ra;
   char nome[61];
   int faltas;
   float notaP1;
   float notaP2;
};

struct aluno clarice;

clarice.ra = 10012235;
strncpy(clarice.nome,"Clarice Lispector",60);
```

o membro `nome` *da variável* `clarice` *é uma "string" para inicializá-la, temos que* ***copiar*** *os dados de outra string*

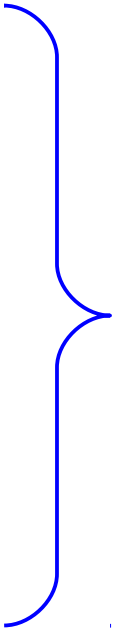
```
struct aluno {
  int ra;
  char nome[61];
  int faltas;
  float notaP1;
  float notaP2;
};

struct aluno clarice;

clarice.ra = 10012235;
strncpy(clarice.nome,"Clarice Lispector",60);
clarice.faltas = 3;
clarice.notaP1 = 6.5;
clarice.notaP2 = 7.5;
```

```
struct aluno {
   int ra;
   char nome[61];
   int faltas;
   float notaP1;
   float notaP2;
};

struct aluno clarice = {
   .ra = 10012235,
   .nome = "Clarice Lispector",
   .faltas = 3,
   .notaP1 = 6.5,
   .notaP2 = 7.5
};
```



declaração de variável com atribuição

```
struct aluno {
   int ra;
   char nome[61];
   int faltas;
   float notaP1;
   float notaP2;
};

struct aluno clarice = {
   .ra = 10012235,
   .nome = "Clarice Lispector",
   .faltas = 3,
   .notaP1 = 6.5,
   .notaP2 = 7.5
};
```

lembre que só na inicialização podemos usar = para atribuir a um array de char

```
struct aluno {
   int ra;
   char nome[61];
   int faltas;
   float notaP1;
   float notaP2;
};

struct aluno *clarice = (struct aluno *)
              malloc(sizeof(struct aluno));
```

```
struct aluno {
   int ra;
   char nome[61];
   int faltas;
   float notaP1;
   float notaP2;
};

struct aluno *clarice = (struct aluno *)
                malloc(sizeof(struct aluno));

(*clarice).ra = 10012235;
```

Acesso a membro de struct via ponteiro.
Método 1: dereferencie o ponteiro e acesse o membro usando o operador ponto.

```
struct aluno {
   int ra;
   char nome[61];
   int faltas;
   float notaP1;
   float notaP2;
};

struct aluno *clarice = (struct aluno *)
                malloc(sizeof(struct aluno));

clarice->ra = 10012235;
```

Acesso a membro de struct via ponteiro.
Método 2: acesse o membro usando o operador ->
(dereferenciação implícita)

```
struct aluno {
  int ra;
  char nome[61];
  int faltas;
  float notaP1;
  float notaP2;
};

struct aluno *clarice = (struct aluno *)
            malloc(sizeof(struct aluno));

clarice->ra = 10012235;
strncpy(clarice->nome,"Clarice Lispector",60);
clarice->faltas = 3;
clarice->notaP1 = 6.5;
clarice->notaP2 = 7.5;
…
free(clarice);
```

OUTRO EXEMPLO: frações inteiras

Vamos definir uma struct para representar frações inteiras.

```
struct fracao {
   int num; // numerador
   unsigned int den; // denominador
};
```

Vamos definir funções para manipular e imprimir frações.

```c
struct fracao {
  int num; // numerador
  unsigned int den; // denominador
};

#include <stdio.h>

void fracao_imprime(struct fracao a) {
  printf("%d/%u", a.num, a.den);
}
```

```c
struct fracao {
  int num; // numerador
  unsigned int den; // denominador
};

#include <stdio.h>

void fracao_imprime(struct fracao a) {
  printf("%d/%u", a.num, a.den);
}
```

E se o numerador for zero? Melhor imprimir na forma mais resumida.

```c
struct fracao {
  int num; // numerador
  unsigned int den; // denominador
};

#include <stdio.h>

void fracao_imprime(struct fracao a) {
  if (a.num == 0) {
    printf("0");
  } else {
    printf("%d/%u", a.num, a.den);
  }
}
```

```c
struct fracao {
  int num; // numerador
  unsigned int den; // denominador
};

#include <stdio.h>

void fracao_imprime(struct fracao a) {
  if (a.num == 0) {
    printf("0");
  } else {
    printf("%d/%u", a.num, a.den);
  }
}
```

E se o denominador for 1?

```c
struct fracao {
  int num; // numerador
  unsigned int den; // denominador
};

#include <stdio.h>

void fracao_imprime(struct fracao a) {
  if (a.num == 0) {
    printf("0");
  } else if (a.den == 1) {
    printf("%d", a.num);
  } else {
    printf("%d/%u", a.num, a.den);
  }
}
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
```

Só para pular linha.

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
10/3
[cling]$
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
10/3
[cling]$ a.num = 0;
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
10/3
[cling]$ a.num = 0;
[cling]$
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
10/3
[cling]$ a.num = 0;
[cling]$ fracao_imprime(a); puts("");
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
10/3
[cling]$ a.num = 0;
[cling]$ fracao_imprime(a); puts("");
0
[cling]$
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
10/3
[cling]$ a.num = 0;
[cling]$ fracao_imprime(a); puts("");
0
[cling]$ a.num = 10; a.den = 1;
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
10/3
[cling]$ a.num = 0;
[cling]$ fracao_imprime(a); puts("");
0
[cling]$ a.num = 10; a.den = 1;
[cling]$
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
10/3
[cling]$ a.num = 0;
[cling]$ fracao_imprime(a); puts("");
0
[cling]$ a.num = 10; a.den = 1;
[cling]$ fracao_imprime(a); puts("");
```

```
[cling]$ .L fracao.c
[cling]$ struct fracao a;
[cling]$ a.num = 10; a.den = 3;
[cling]$ fracao_imprime(a); puts("");
10/3
[cling]$ a.num = 0;
[cling]$ fracao_imprime(a); puts("");
0
[cling]$ a.num = 10; a.den = 1;
[cling]$ fracao_imprime(a); puts("");
10
[cling]$
```

```
struct fracao {
  int num; // numerador
  unsigned int den; // denominador
};

struct fracao fracao_multiplica(
                        struct fracao a,
                        struct fracao b) {
  struct fracao result;


  return result;
}
```

```
struct fracao {
  int num; // numerador
  unsigned int den; // denominador
};

struct fracao fracao_multiplica(
                        struct fracao a,
                        struct fracao b) {
  struct fracao result;
  result.num = a.num * b.num;
  result.den = a.den * b.den;
  return result;
}
```

```
struct fracao {
  int num; // numerador
  unsigned int den; // denominador
};

struct fracao fracao_soma(
                        struct fracao a,
                        struct fracao b) {
  struct fracao result;


  return result;
}
```

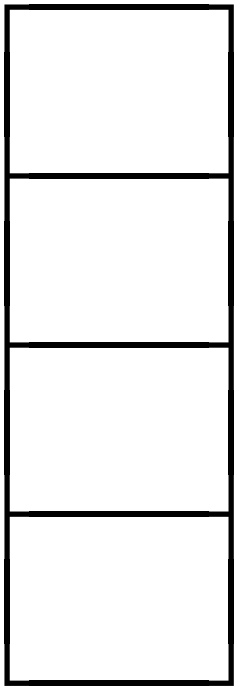
```
struct fracao {
  int num; // numerador
  unsigned int den; // denominador
};

struct fracao fracao_soma(
                        struct fracao a,
                        struct fracao b) {
  struct fracao result;
  result.num = a.num*b.den + a.den*b.num;
  result.den = a.den*b.den;
  return result;
}
```

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros

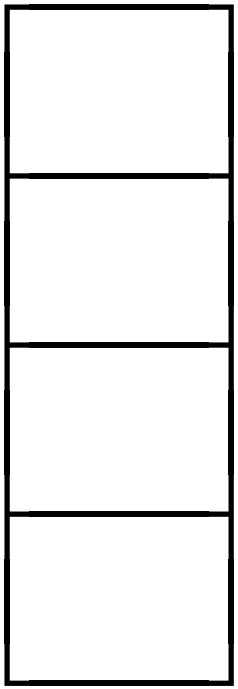


← topo

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



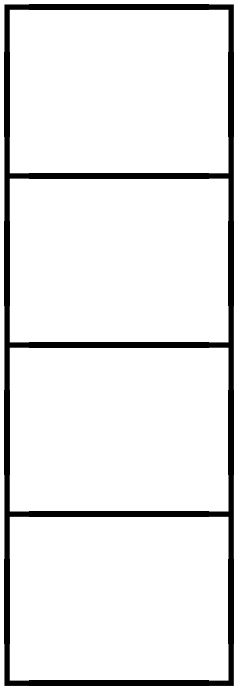
push(33)

← topo

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



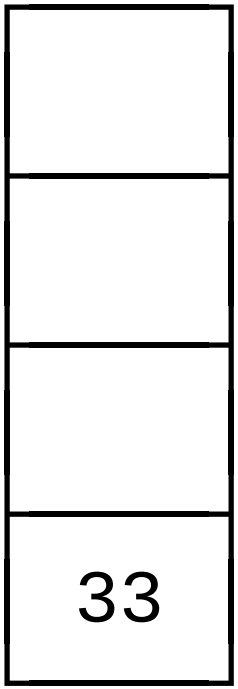
push(33)

← topo

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



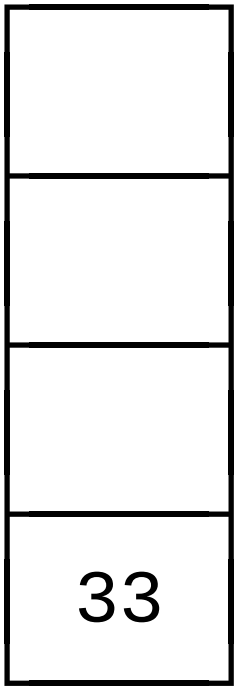
← topo

push(33)

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



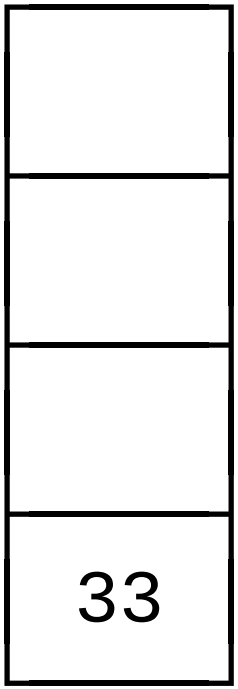
← topo

push(33)
push(42)

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



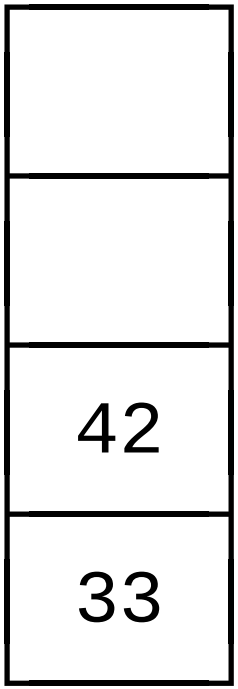
← topo

push(33)
push(42)

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



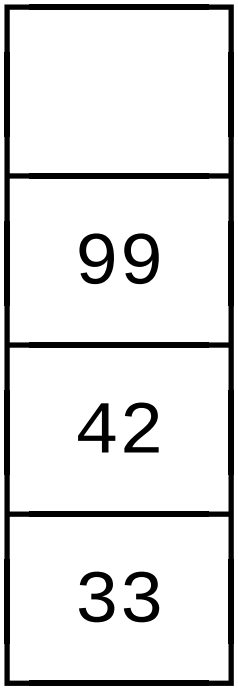
← topo

push(33)
push(42)

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



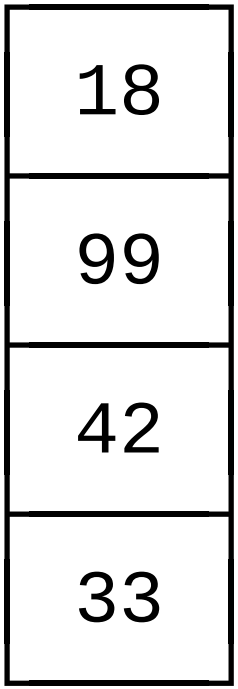
← topo

push(33)
push(42)
push(99)

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



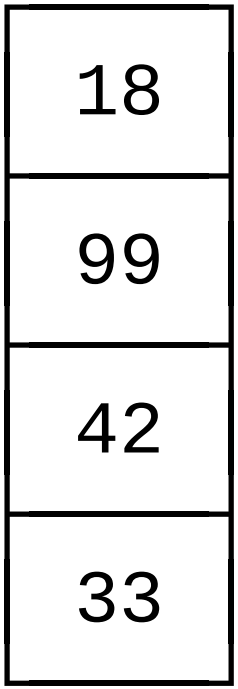
← topo

push(33)
push(42)
push(99)
push(18)

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



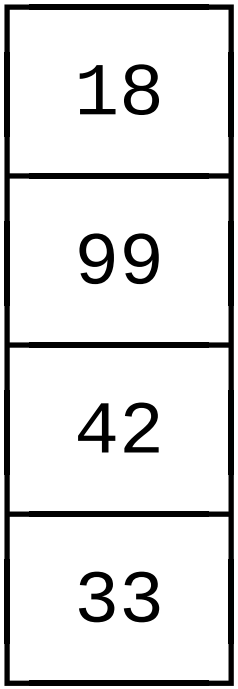
← topo

push(33)
push(42)
push(99)
push(18)
push(51)

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



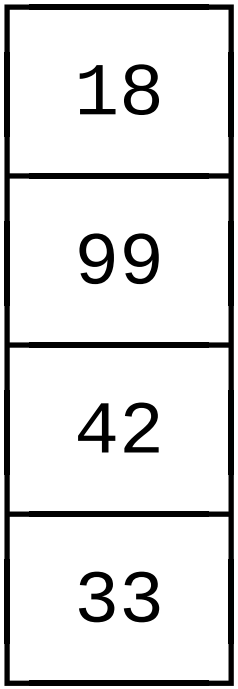
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



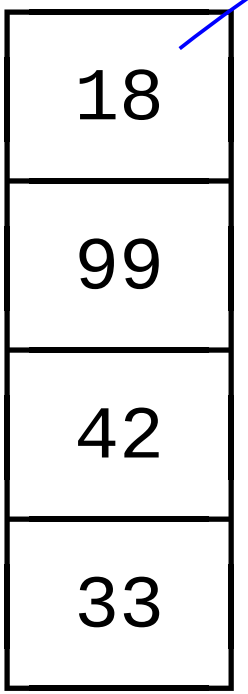
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop()

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



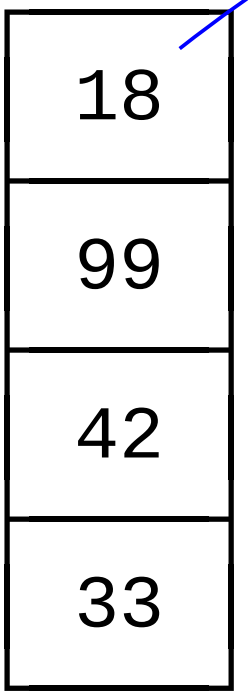
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() →

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



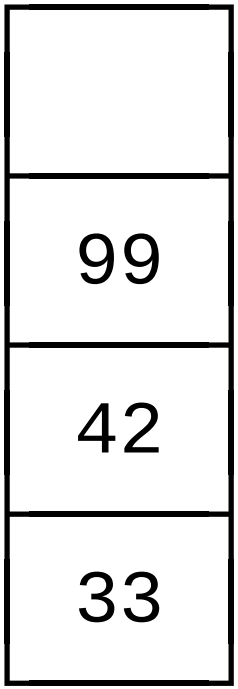
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() → 18

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



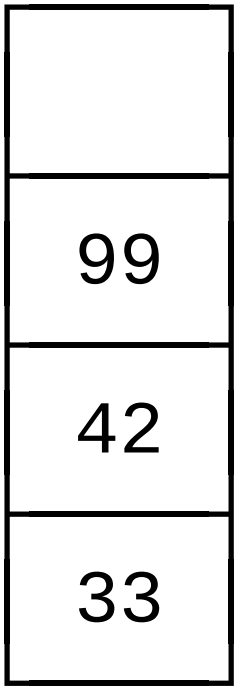
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() → 18

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



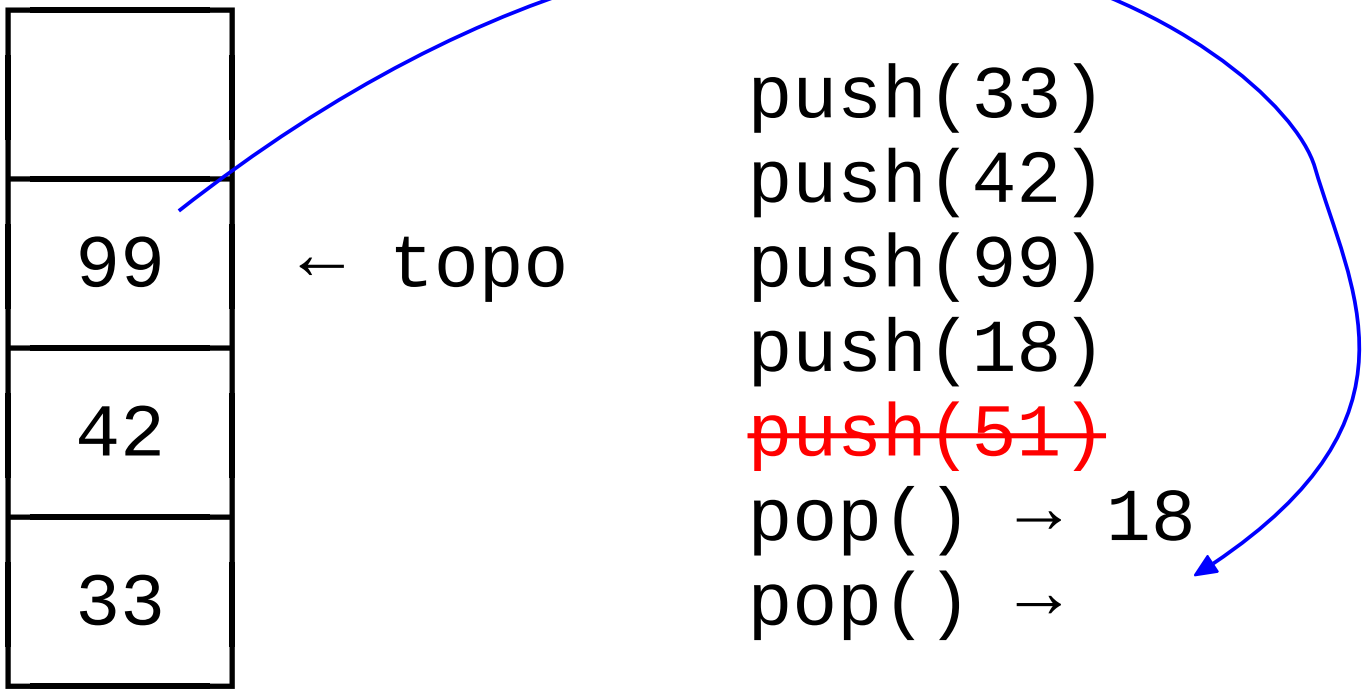
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() → 18
pop()

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

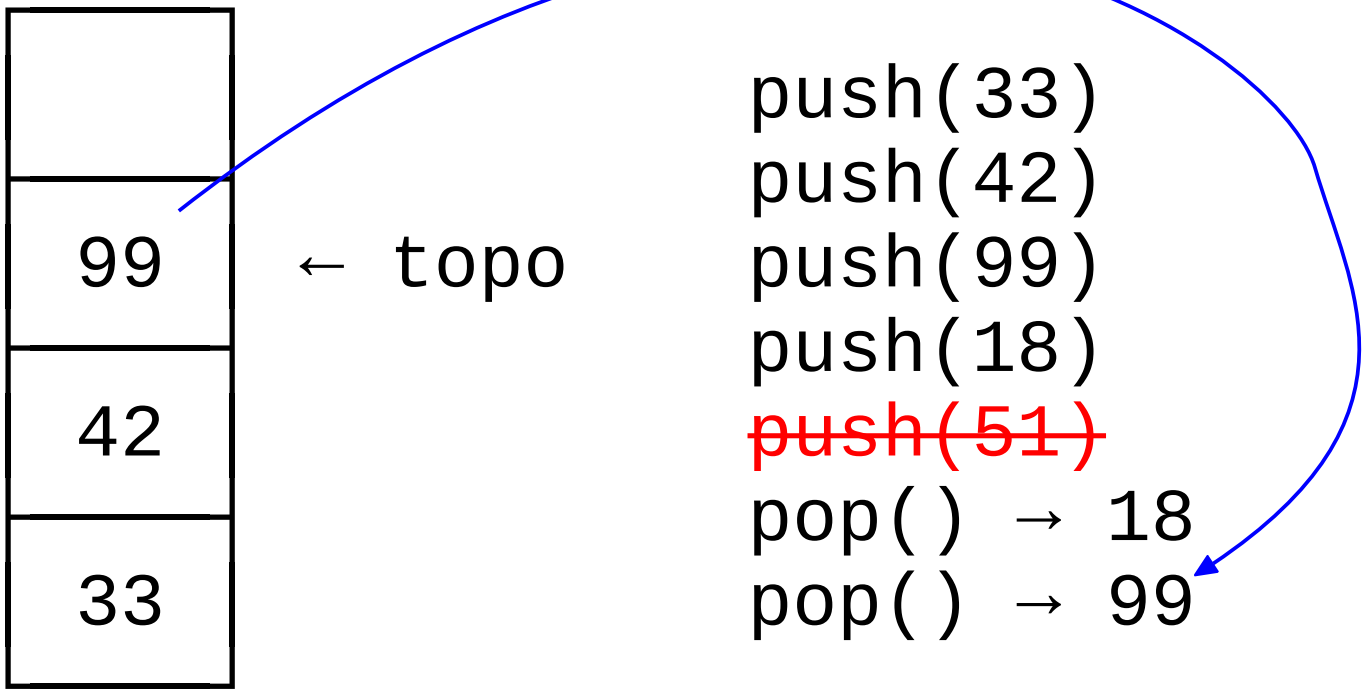
Ex.: uma pilha de inteiros



PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

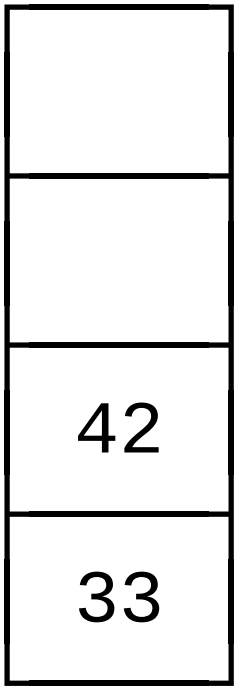
Ex.: uma pilha de inteiros



PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



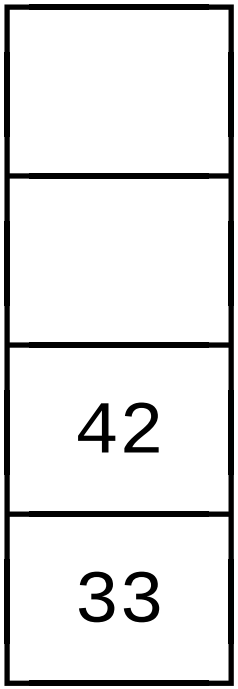
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() → 18
pop() → 99

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



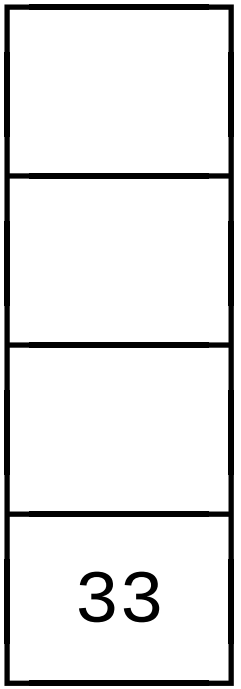
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() → 18
pop() → 99
pop()

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



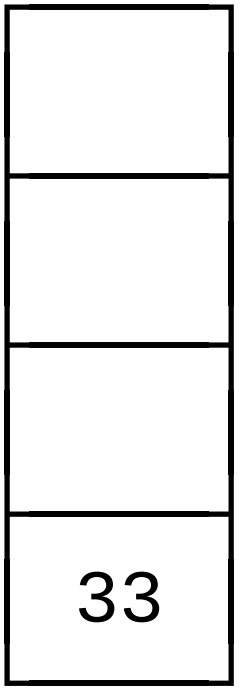
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() → 18
pop() → 99
pop() → 42

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



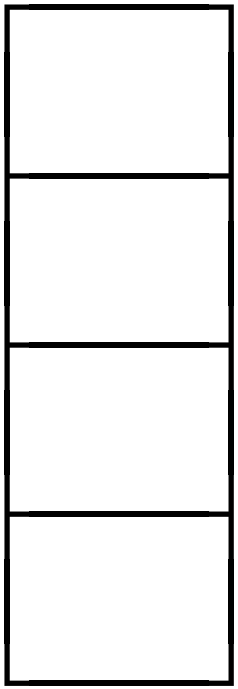
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() → 18
pop() → 99
pop() → 42
pop()

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



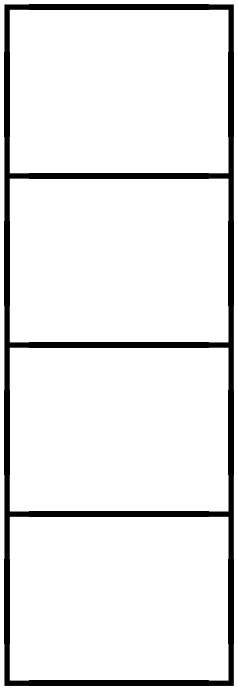
← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() → 18
pop() → 99
pop() → 42
pop() → 33

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



← topo

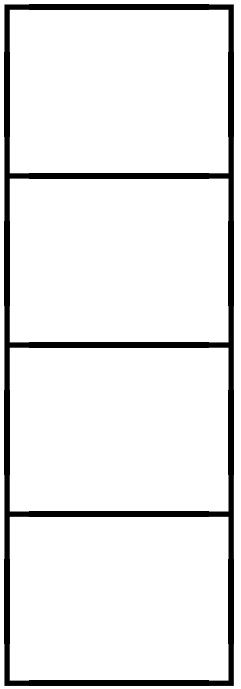
push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() → 18
pop() → 99
pop() → 42
pop() → 33

pop()

PILHA

Estrutura para armazenamento de dados onde as inserções (push) e remoções (pop) acontecem apenas em uma das pontas.

Ex.: uma pilha de inteiros



← topo

push(33)
push(42)
push(99)
push(18)
~~push(51)~~
pop() → 18
pop() → 99
pop() → 42
pop() → 33

~~pop()~~

```
struct pilha_int {
   int *numeros;
   int capacidade;
   int topo;
};
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};



void pilha_int_constroi(struct pilha_int *p,
                        int capacidade) {









}
```

```
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};



void pilha_int_constroi(struct pilha_int *p,
                        int capacidade) {
  p->capacidade = capacidade;
  p->topo = -1;
  p->numeros = (int *)
               malloc(capacidade*sizeof(int));




}
```

```
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

#include <stdlib.h>


void pilha_int_constroi(struct pilha_int *p,
                        int capacidade) {
  p->capacidade = capacidade;
  p->topo = -1;
  p->numeros = (int *)
               malloc(capacidade*sizeof(int));




}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

#include <stdlib.h>


void pilha_int_constroi(struct pilha_int *p,
                        int capacidade) {
  p->capacidade = capacidade;
  p->topo = -1;
  p->numeros = (int *)
               malloc(capacidade*sizeof(int));

}
```

E se malloc não conseguir alocar memória?

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

#include <stdlib.h>


void pilha_int_constroi(struct pilha_int *p,
                        int capacidade) {
  p->capacidade = capacidade;
  p->topo = -1;
  p->numeros = (int *)
               malloc(capacidade*sizeof(int));
  if (p->numeros == NULL) {


  }
}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

#include <stdlib.h>


void pilha_int_constroi(struct pilha_int *p,
                        int capacidade) {
  p->capacidade = capacidade;
  p->topo = -1;
  p->numeros = (int *)
               malloc(capacidade*sizeof(int));
  if (p->numeros == NULL) {
    puts("erro de alocação!");
    p->capacidade = 0;
  }
}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

#include <stdlib.h>
#include <stdio.h>

void pilha_int_constroi(struct pilha_int *p,
                        int capacidade) {
  p->capacidade = capacidade;
  p->topo = -1;
  p->numeros = (int *)
               malloc(capacidade*sizeof(int));
  if (p->numeros == NULL) {
    puts("erro de alocação!");
    p->capacidade = 0;
  }
}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

// Retorna verdadeiro se a pilha está vazia
int pilha_int_vazia(struct pilha_int *p) {

}

// Retorna verdadeiro se a pilha está cheia
int pilha_int_cheia(struct pilha_int *p) {

}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

// Retorna verdadeiro se a pilha está vazia
int pilha_int_vazia(struct pilha_int *p) {
  return p->topo < 0;
}

// Retorna verdadeiro se a pilha está cheia
int pilha_int_cheia(struct pilha_int *p) {

}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

// Retorna verdadeiro se a pilha está vazia
int pilha_int_vazia(struct pilha_int *p) {
  return p->topo < 0;
}

// Retorna verdadeiro se a pilha está cheia
int pilha_int_cheia(struct pilha_int *p) {
  return p->topo >= p->capacidade - 1;
}
```

```
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

void
pilha_int_push(struct pilha_int *p, int n) {




}
```

```
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

void
pilha_int_push(struct pilha_int *p, int n) {

    ++p->topo;

    p->numeros[p->topo] = n;

}
```

```
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

void
pilha_int_push(struct pilha_int *p, int n) {



    p->numeros[++p->topo] = n;

}
```

```
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

void
pilha_int_push(struct pilha_int *p, int n) {

    E se a pilha estiver cheia?

   p->numeros[++p->topo] = n;

}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

void
pilha_int_push(struct pilha_int *p, int n) {
  if (pilha_int_cheia(p)) {
    puts("pilha cheia!");
  } else {
    p->numeros[++p->topo] = n;
  }
}
```

```
struct pilha_int {
   int *numeros;
   int capacidade;
   int topo;
};

int pilha_int_pop(struct pilha_int *p) {




}
```

```c
struct pilha_int {
   int *numeros;
   int capacidade;
   int topo;
};

int pilha_int_pop(struct pilha_int *p) {

    int valor = p->numeros[p->topo];

    p->topo--;

    return valor;

}
```

```
struct pilha_int {
   int *numeros;
   int capacidade;
   int topo;
};

int pilha_int_pop(struct pilha_int *p) {

    int valor = p->numeros[p->topo--];


    return valor;

}
```

```
struct pilha_int {
   int *numeros;
   int capacidade;
   int topo;
};

int pilha_int_pop(struct pilha_int *p) {



    return p->numeros[p->topo--];

}
```

```
struct pilha_int {
   int *numeros;
   int capacidade;
   int topo;
};

int pilha_int_pop(struct pilha_int *p) {


      return p->numeros[p->topo--];

}
```

E se a pilha estiver vazia?

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

int pilha_int_pop(struct pilha_int *p) {
  if (pilha_int_vazia(p)) {
    puts("pilha vazia!");

  } else {
    return p->numeros[p->topo--];
  }
}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

int pilha_int_pop(struct pilha_int *p) {
  if (pilha_int_vazia(p)) {
    puts("pilha vazia!");
    Somos obrigados a retornar um int!
  } else {
    return p->numeros[p->topo--];
  }
}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

int pilha_int_pop(struct pilha_int *p) {
  if (pilha_int_vazia(p)) {
    puts("pilha vazia!");
    return 0;
  } else {
    return p->numeros[p->topo--];
  }
}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

void pilha_int_destroi(struct pilha_int *p) {




}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

void pilha_int_destroi(struct pilha_int *p) {

    free(p->numeros);
    p->numeros = NULL;
    p->capacidade = 0;
    p->topo = -1;

}
```

```c
struct pilha_int {
  int *numeros;
  int capacidade;
  int topo;
};

void pilha_int_destroi(struct pilha_int *p) {
  if (p->numeros != NULL) {
    free(p->numeros);
    p->numeros = NULL;
    p->capacidade = 0;
    p->topo = -1;
  }
}
```

STRUCTS: BOAS PRÁTICAS

Sempre que definir structs, adquira o hábito de definir as funções que as manipulem.

Sempre que possível, manipule as structs usando suas funções. Evite mexer manualmente nos campos fora das funções (facilita futuros *refactorings*).

Nomenclatura sugerida: ao definir uma struct *XYZ*, nomeie as funções relacionadas como *XYZ*_nome_da_funcao para evitar conflito de nomes.

```
struct pilha_int { int *numeros; ... };

void pilha_int_constroi(struct pilha_int *, int);
int pilha_int_vazia(struct pilha_int *);
int pilha_int_cheia(struct pilha_int *);
void pilha_int_push(struct pilha_int *, int);
int pilha_int_pop(struct pilha_int *);
void pilha_int_destroi(struct pilha_int *);
```

```
struct pilha_int { int *numeros; ... };

void pilha_int_constroi(struct pilha_int *, int);
int pilha_int_vazia(struct pilha_int *);
int pilha_int_cheia(struct pilha_int *);
void pilha_int_push(struct pilha_int *, int);
int pilha_int_pop(struct pilha_int *);
void pilha_int_destroi(struct pilha_int *);


struct pilha_dbl { double *numeros; ... };

void pilha_dbl_constroi(struct pilha_dbl *, int);
int pilha_dbl_vazia(struct pilha_dbl *);
int pilha_dbl_cheia(struct pilha_dbl *);
void pilha_dbl_push(struct pilha_dbl *, double);
double pilha_dbl_pop(struct pilha_dbl *);
void pilha_dbl_destroi(struct pilha_dbl *);
```

STRUCTS: BOAS PRÁTICAS

Mantenha a consistência nas definições das funções relacionadas a uma struct.

Sugere-se que a struct (ou o ponteiro para ela) seja **sempre** passada no primeiro parâmetro.

```
void pilha_int_constroi(struct pilha_int *, int);
int pilha_int_vazia(struct pilha_int *);
int pilha_int_cheia(struct pilha_int *);
void pilha_int_push(struct pilha_int *, int);
int pilha_int_pop(struct pilha_int *);
void pilha_int_destroi(struct pilha_int *);
```

PARA CASA

0. Altere fracao.c e:

→ implemente a função transforme_em_irredutivel que transforma a fração passada em irredutível (use o mdc entre numerador e denominador)

→ altere as funções soma e multiplica para que elas sempre produzam frações irredutíveis

→ implemente a função fracao_compara conforme instruções em fracao.c

PARA CASA

1. Altere a função pilha_int_insere para transformar a pilha de inteiros em uma **pilha dinâmica**: se não houver espaço para inserir um número na pilha, deve ser alocado um novo espaço para a pilha **com o dobro do tamanho** do anterior, os elementos antigos devem ser copiados e o novo elemento é inserido.

Em caso de nova alocação, lembre de liberar (free) a alocação antiga.

PARA CASA

2. Crie as funções

```
int pilha_int_tamanho(struct pilha_int *p)
```

que retorna a quantidade de elementos inseridos na pilha de inteiros e

```
int pilha_int_elemento(struct pilha_int *p, int i)
```

que retorna o i-ésimo elemento da pilha, para i entre 0 e o índice do último elemento na pilha.

Sua função deve verificar se o índice é válido. Caso não seja, deve imprimir uma mensagem de erro e retornar 0.

PARA CASA

3. Crie uma pilha de alunos, onde cada elemento da pilha é uma **struct aluno**.

As operações devem ser pilha_aluno_constroi, pilha_aluno_destroi, pilha_aluno_vazia, pilha_aluno_cheia, pilha_aluno_push, pilha_aluno_pop, pilha_aluno_tamanho e pilha_aluno_elemento.

Para testar, pegue o arquivo alunos.c da aula passada e insira os dados ali contidos na pilha.

PARA CASA

4. Crie funções pilha_aluno_ordena_media e pilha_aluno_ordena_nome, que ordenam a pilha de alunos pela média (0,4 P1 + 0,6 P2) e pelo nome.
Use *bubble sort*.

Dica: a função strcmp de string.h permite comparar duas strings *s1* e *s2*.

strcmp(*s1*, *s2*) é
- < 0 se *s1* vem antes de *s2* no dicionário
- ==0 se *s1* é igual a *s2*
- > 0 se *s1* vem após *s2* no dicionário

PARA CASA

5. Crie uma struct vetor_dbl para representar um vetor cujas coordenadas são doubles, junto com seu tamanho. Crie as funções para: criar um vetor, soma, produto interno, módulo (tamanho de acordo com a distância euclidiana) e ângulo entre dois vetores.

6. Crie uma struct matriz_dbl para representar matrizes bidimensionais, junto com suas coordenadas. Crie as funções: criar matriz, soma, produto matriz x matriz e matriz x vetor.