Programação Procedimental

Variações de Listas e Tipo Abstrato de Dados (TAD)

Aula 10

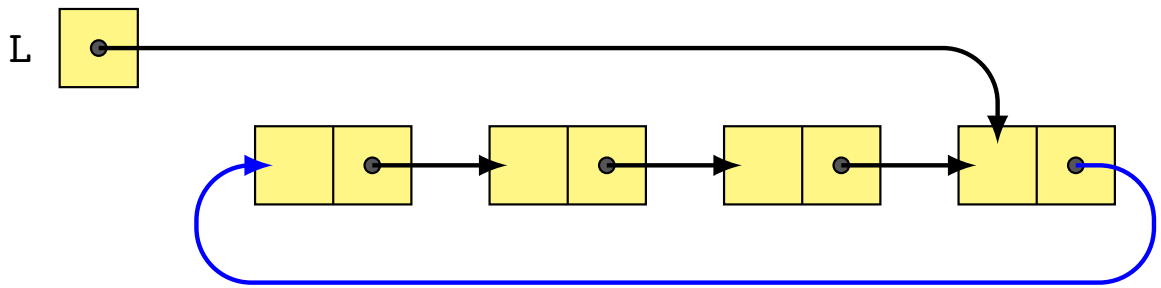
Prof. Felipe A. Louza

Roteiro

Lista circular

Lista circular:

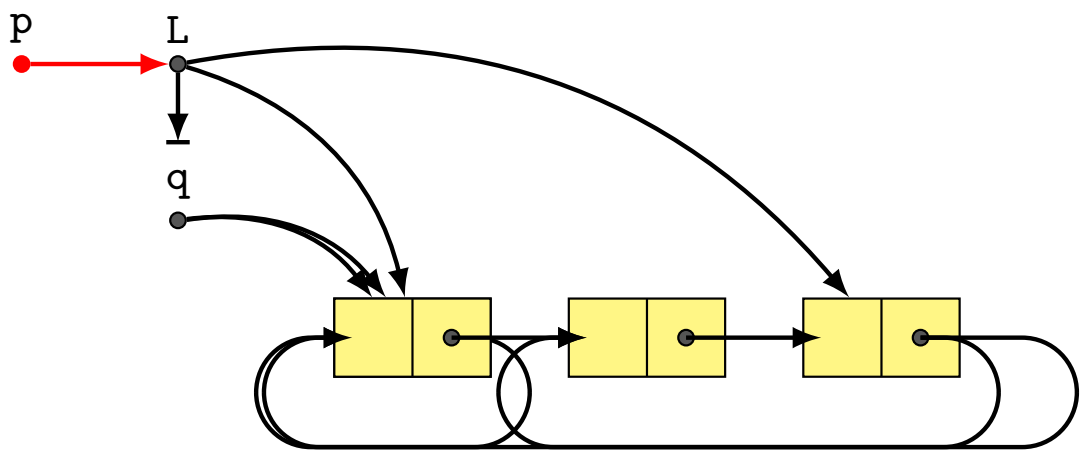
- O último elemento aponta para o primeiro.



- Vantagens?
  - Podemos varrer a lista a partir de qualquer ponto e voltar ao início.
- Observação:
  - A lista sempre aponta para o último elemento, para acessar o primeio: `(L->prox)->valor`

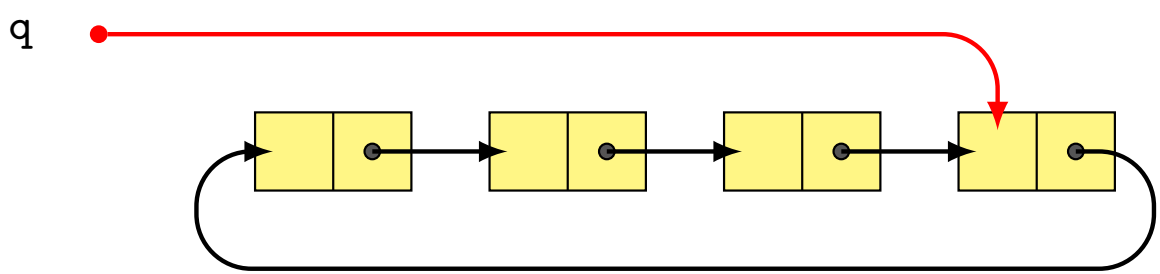
Inserindo em lista circular

```
1   void insere_comeco(No** p, int v) { //p recebe &L
2     No* q;
3     q = (No*) malloc(sizeof(No));
4     q->valor = v;
5     if(*p == NULL){
6       *p = q;
7       q->prox = q;
8     }
9     else{
10      q->prox = (*p)->prox;
11      (*p)->prox = q;
12    }
13  }
```



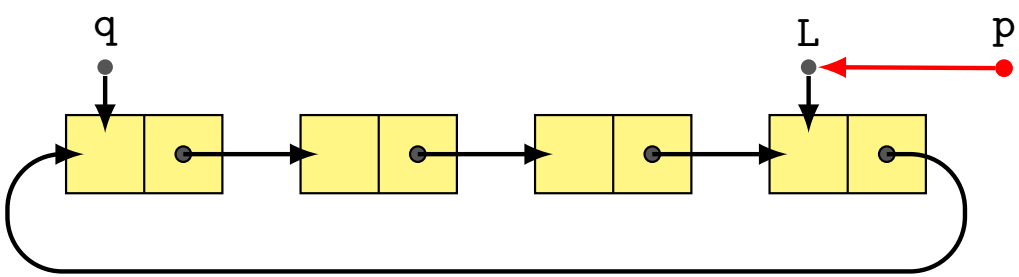
Imprimindo a lista

```c
int imprime_lista(No *q) { //q recebe L
  if(q==NULL) return -1; //lista vazia
  No *aux = q->prox;
  while(aux != q){
    printf("%d\n", aux->valor);
    aux = aux->prox;
  }
  printf("%d\n", aux->valor);//último valor
return 0;
}
```



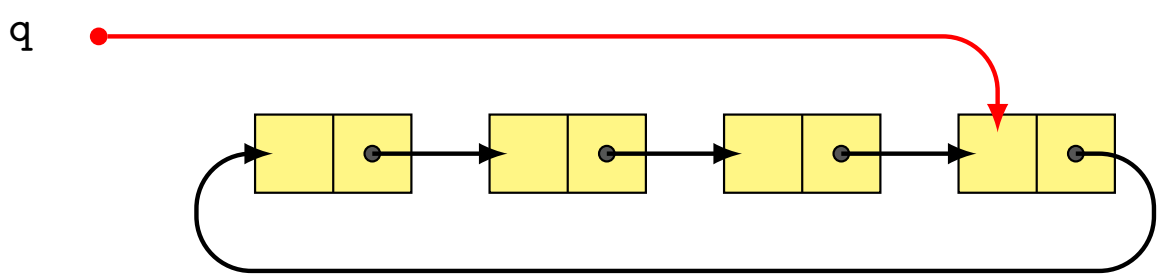
Removendo de lista circular

```
1   void remove_comeco(No** p) {  //p recebe &L
2     No* q = *p;
3     if(q==NULL) return; //lista vazia
4     if(q->prox==q){ //apenas 1 elemento
5       *p = NULL;
6       free(q);
7       return;
8     }
9     q = q->prox;
10    (*p)->prox = q->prox;
11    free(q);
12  }
```



Busca em lista circular

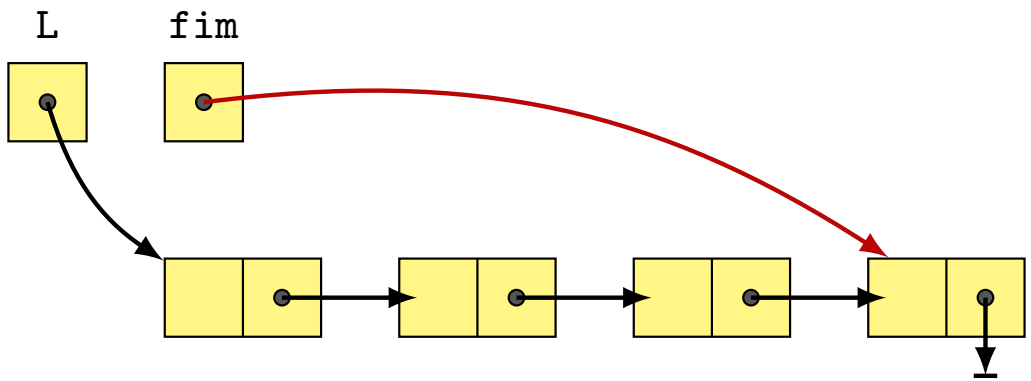
```
1   //retorna (1 == true) se x existe na lista ou (0 == false), caso contrário
2   int busca_lista(No *q, int x) { //q recebe L
3     if(q==NULL) return -1; //lista vazia
4     if(q->valor == x) return 1; //
5     No *aux = q->prox;
6     while(aux != q){
7       if(aux->valor == x) return 1; //true!
8       aux = aux->prox;
9     }
10  return 0;//false == não encontrou
11  }
```



Roteiro

Algumas alternativas

Apontador para o fim da lista:

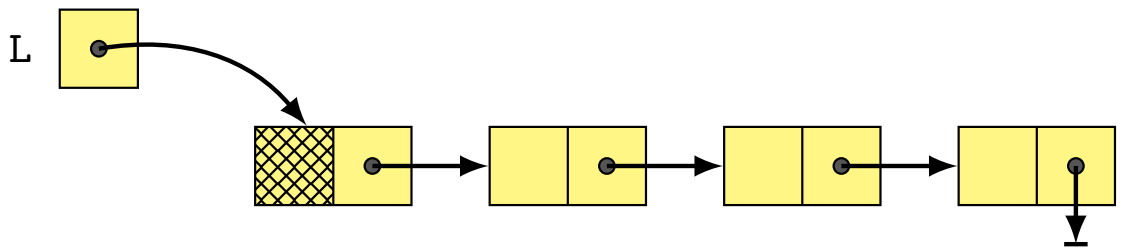


- Vantagens?
  - Inserção no final da lista.
  - Remoção?

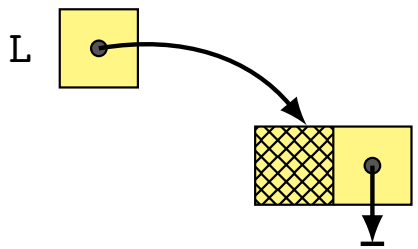
Algumas alternativas

Lista com nó cabeça:

- O primeiro nó não armazena elementos da lista, mas pode armazenar dados sobre a lista, como o número de nós da lista.



- Lista vazia:

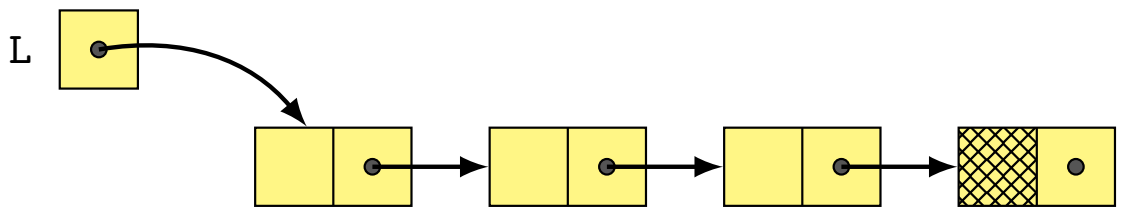


- Vantagem:
  - Simplifica as implementações.

Algumas alternativas

Nó sentinela:

- Um nó sentinela (ou dummy) é adicionado à lista para marcar fim da lista.

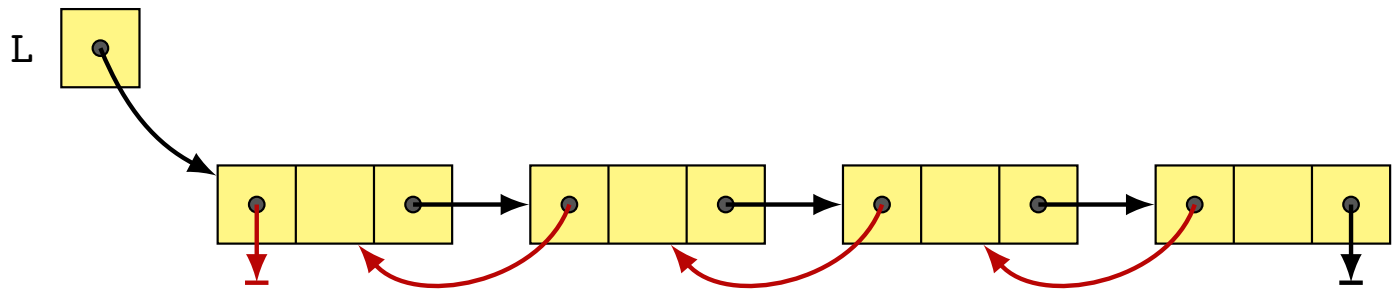


- Vantagem:
  - Permite escrever programas mais homogêneos, que não precisam tratar o caso de lista vazia.

Lista duplamente ligada

Lista duplamente ligada:

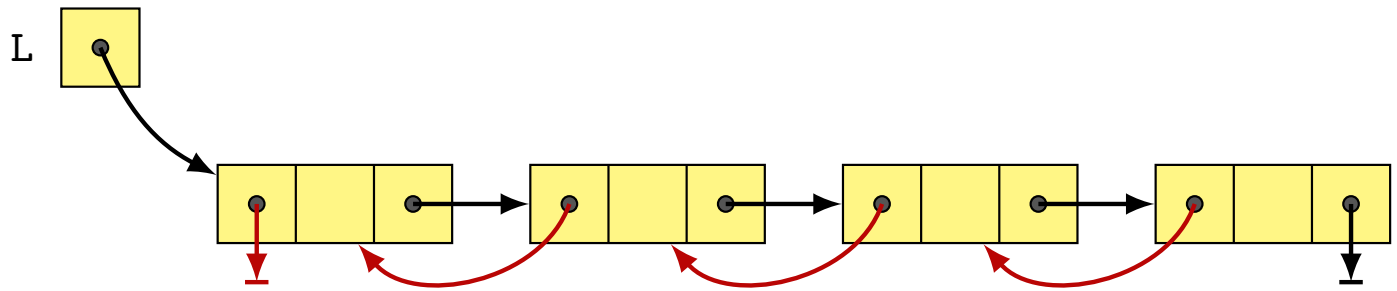
- Cada nó tem um apontador para o próximo elemento e para o elemento anterior na lista.




```c
typedef struct no {
  int valor;
  struct no *prox, *prev;
} No;
```

Algumas alternativas

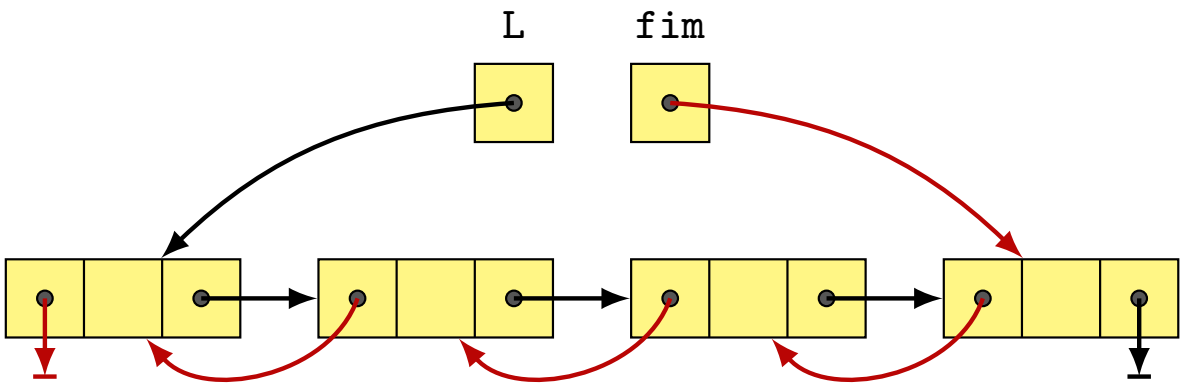
Lista duplamente ligada:



- Vantagem:
  - Facilita navegação.
- Desvantagens?
  - Manipular os ponteiros.
  - Memória (uso de mais um ponteiro).

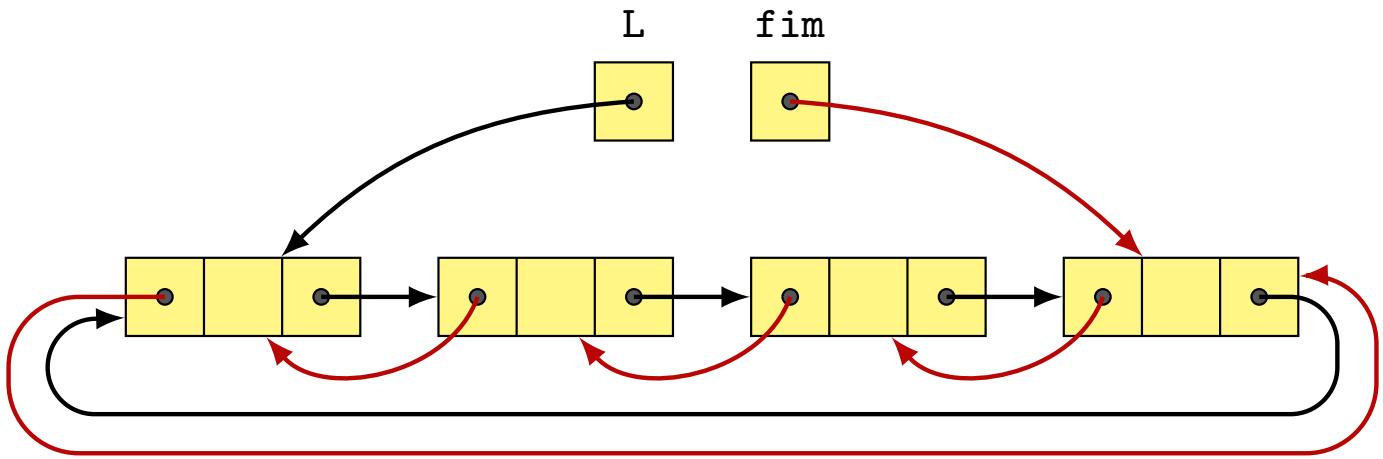
Variações - Duplamente ligada

Lista duplamente ligada (variação 1):



Variações - Lista dupla circular

Lista duplamente ligada (variação 2):



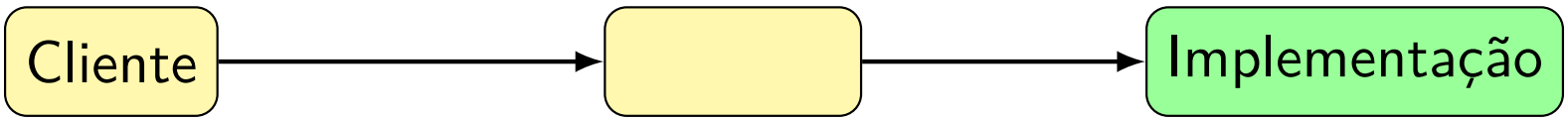
Observações:

- Variável `fim` é opcional (`fim == L->ant`)

Podemos ter uma lista dupla circular com cabeça, e outras variações...

Roteiro

TAD

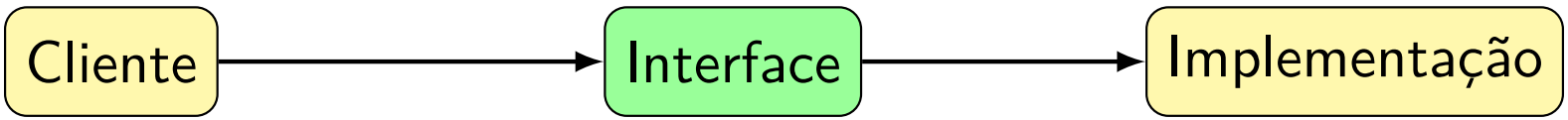


Tipo Abstrato de Dados (TAD):

- Agrupar a estrutura de dados juntamente com as operações que podem ser feitas sobre esses dados.
- Faremos algo que se parece com uma classe (POO)[1].

[1] C não é Orientada a Objetos como Python

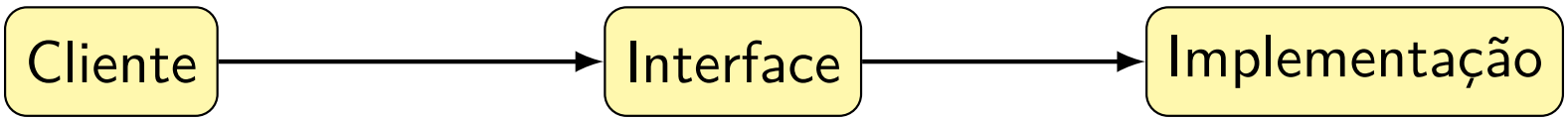
TAD



Encapsulamento:

- Os usuários do TAD (cliente) só tem acesso a algumas operações disponibilizadas sobre esses dados.
- Código mais simples, claro e elegante.
  - O cliente só se preocupa em usar funções.

TAD

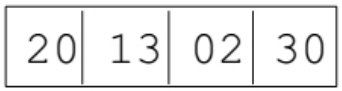


Encapsulamento:

- Usuário do TAD *vs.* Programador do TAD.
  - Usuário só "enxerga" a interface, não a implementação.
- Podemos mudar a implementação sem quebrar clientes.
  - Os resultados das funções precisam ser os mesmos

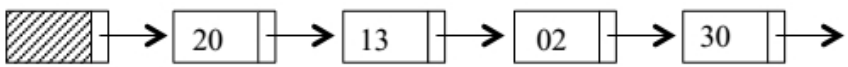
Exemplo de TAD: Lista de números inteiros.

Implementação por Vetor:




```
1 int insereInicio(Lista *L, int x){
2   int i;
3   for(i=L.size; i>=0; i--)
4     L[i+1] = L[i];
5   L[0] = x;
6 }
```

Implementação por Lista Ligada:




```
1 int insereInicio(Lista *L, int x){
2   No *aux = malloc(sizeof(No));
3   aux->prox = L->prox;
4   L->prox = aux;
5 }
```

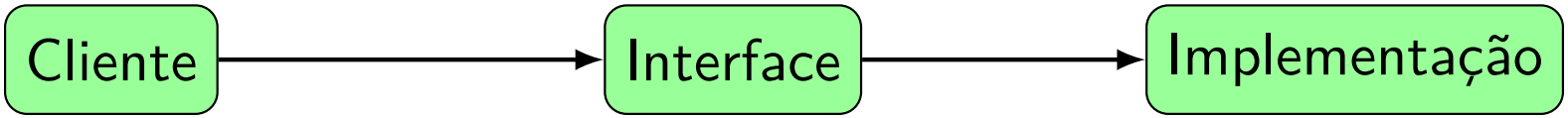
Interface:

```
1 typedef int* Lista;
2 //typedef No* Lista;
3 ...
4 int insereInicio(Lista *L, int x);
5 ...
```

Programa principal:

```
1 int main(){
2   Lista *L;
3   int x = 20;
4   criaLista(L);
5   insereInicio(L, x);
6   ...
7 return 0;
8 }
```

Em resumo...



- Interface: conjunto de operações de um TAD
  - Consiste dos nomes e definições usadas para executar as operações.
- Implementação: conjunto de algoritmos que realizam as operações
  - A implementação é o único "lugar" que uma variável é acessada diretamente
- Cliente: código que utiliza/chama uma operação
  - O cliente nunca acessa a variável diretamente

TAD

Vantagens:

- Maior independência e facilidade de manutenção do código.
- Maior potencial de reutilização de código.
  - Mais fácil colaborar com outros programadores
- Abstração
  - Não precisa disponibilizar o código fonte da biblioteca

TAD

Em linguagem C:

- A implementação da interface é feita pela definição de tipos juntamente com a protótipo de funções.
- Uma boa técnica de programação é utilizar arquivos separados:
  - `NomeTAD.h`: com as declarações de funções e tipos.
  - `NomeTAD.c`: com as implementações de funções.

Utilizando um TAD:

- Os programas, ou outros TADs, que utilizam o seu TAD devem usar o `#include` para o arquivo `NomeTAD.h`

principal.c

```
1  #include <stdio.h>
2  #include "NomeTAD.h"
3
4  int main(){
5  ...
6  }
```

TAD

Compilando um TAD (arquivo `.c`):

- `gcc -c NomeTAD.c`: compila um arquivo individual sem a função `main()`.
- Resultado: arquivo objeto `NomeTAD.o`
- Em seguida, compilamos o arquivo com o programa principal e incluímos (ligamos) `NomeTAD.o`.

Terminal

```
1 gcc -c NomeTAD.c
2 gcc principal.c NomeTAD.o
```

Roteiro

Números Complexos

Vamos criar um programa que lida com números complexos

- Um número complexo é da forma $a + b\,i$
  - a e b são números reais
  - $i = \sqrt{-1}$ é a unidade imaginária

Operações:

- somar dois números complexos; e
- calcular o valor absoluto: $\sqrt{a^2 + b^2}$

Reflexão

Quando somamos 2 variáveis `float`:

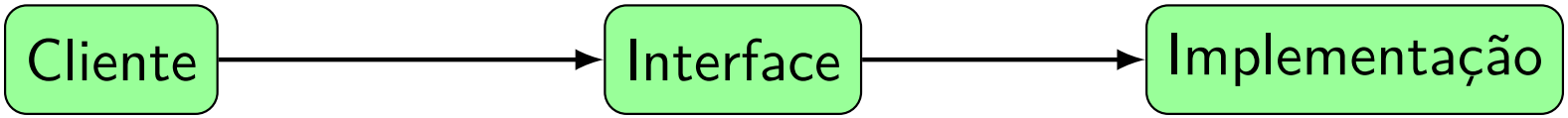
- não nos preocupamos como a operação é feita
  - internamente o float é representado por um número binário
  - Ex: `0.3` é representado como `00111110100110011001100110011010`
- o compilador esconde os detalhes!

E se quisermos lidar com números complexos?

Será que também podemos abstrair o conceito de um número complexo?

- Sim - faremos um TAD

Relembrando...



- Interface: conjunto de operações e definições de um TAD
- Implementação: conjunto de algoritmos que realizam as operações
- Cliente: código que utiliza/chama uma operação

Como criar um TAD

Construindo um TAD:

- Um nome para o tipo a ser usado
  - Ex: `complexo`
  - Uma `struct` com um `typedef`
- Quais funções ele deve responder
  - `soma`, `absoluto`, etc...
  - Consideramos quais serão as entradas, e o resultado esperado
  - Idealmente, cada função tem apenas uma responsabilidade

Ou seja, primeiro definimos a interface

- Depois, escrevemos uma possível implementação

Números Complexos - Interface

complexo.h

```
1  typedef struct {
2    double real;
3    double imag;
4  } complexo;
5
6  complexo complexo_novo(double real, double imag);
7  complexo complexo_soma(complexo a, complexo b);
8  double complexo_absoluto(complexo a);
9
10 complexo complexo_le();
11 void complexo_imprime(complexo a);
12
13 int complexos_iguais(complexo a, complexo b);
14 complexo complexo_multiplicacao(complexo a, complexo b);
15 complexo complexo_conjugado(complexo a);
```

Números Complexos - Implementação

complexo.c

```c
#include <stdio.h>
#include <math.h>
#include "complexos.h"

complexo complexo_novo(double real, double imag) {
  complexo c;
  c.real = real;
  c.imag = imag;
  return c;
}

complexo complexo_soma(complexo a, complexo b) {
  return complexo_novo(a.real + b.real, a.imag + b.imag);
}

complexo complexo_le() {
  complexo a;
  scanf("%lf %lf", &a.real, &a.imag);
  return a;
}
...
```

Números Complexos - Implementação

complexo.c

```
1  ...
2  double complexo_absoluto(complexo a) {
3    return sqrt(a.real*a.real + a.imag*a.imag);
4  }
5
6  void complexo_imprime(complexo a) {
7    printf("%lf + %lfi\n", a.real, a.imag);
8  }
9
10 int complexos_iguais(complexo a, complexo b) {
11   return (a.real == b.real) && (a.imag == b.imag);
12 }
13
14 complexo complexo_multiplicao(complexo a, complexo b) {
15   return complexo_novo(a.real*b.real - a.imag*b.imag,
16                        a.real*b.imag + b.real*a.imag);
17 }
18
19 complexo complexo_conjugado(complexo a) {
20   return complexo_novo(a.real, - a.imag);
21 }
```

Números Complexos - Exemplo de Cliente

E quando formos usar TAD de números complexos em nossos programas?

```c
#include <stdio.h>
#include "complexos.h"

int main() {
  complexo a, b, c;
  a = complexo_le();
  b = complexo_le();
  c = complexo_soma(a, b);
  complexo_imprime(c);
  printf("%lf\n", complexo_absoluto(c));

return 0;
}
```

Como compilar?

Temos três arquivos diferentes:

- `cliente.c` contém a função `main`
- `complexos.c` contém a implementação
- `complexos.h` contém a interface

Vamos compilar por partes:

```
1 gcc -Wall -Werror -c cliente.c
```

- vai gerar o arquivo compilado `cliente.o`

```
1 gcc -Wall -Werror -c complexos.c
```

- vai gerar o arquivo compilado `complexos.o`

```
1 gcc cliente.o complexos.o -lm -o cliente
```

- faz a linkagem, gerando o executável `cliente`

Roteiro

Makefile

É mais fácil usar um Makefile para compilar:

```
1  all: cliente
2
3  cliente: cliente.o complexos.o
4          gcc cliente.o complexos.o -lm -o cliente
5
6  cliente.o: cliente.c complexos.h
7          gcc -Wall -Werror -c cliente.c
8
9  complexos.o: complexos.c complexos.h
10         gcc -Wall -Werror -c complexos.c
```

Basta executar `make` na pasta com os arquivos:

- `cliente.c`, `complexos.c`, `complexos.h`
- `Makefile`

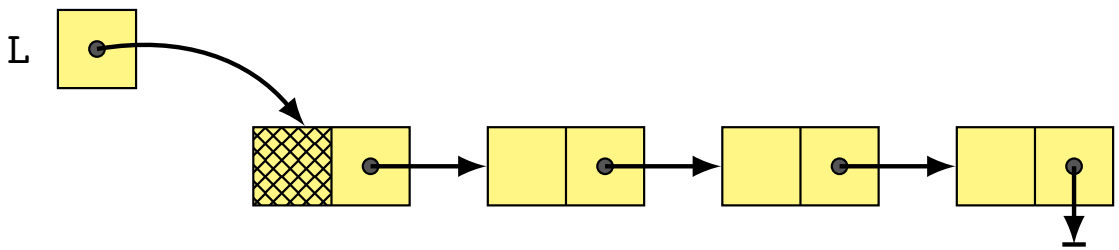
Apenas recompila o que for necessário!

Roteiro

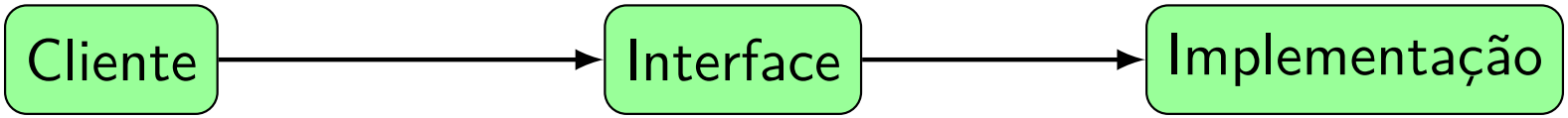
TAD

Exemplo de TAD:

- Lista de inteiros: 3, 5, 9.
- Implementação: Lista Ligada com nó cabeça.



Relembrando...



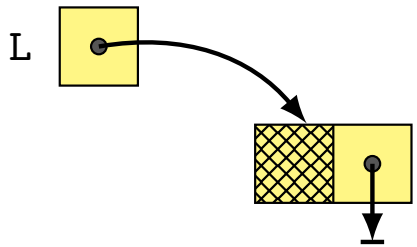
- Interface: conjunto de operações e definições de um TAD
- Implementação: conjunto de algoritmos que realizam as operações
- Cliente: código que utiliza/chama uma operação

TAD

minhaLista.h

```c
/* TAD: Minha Lista */
/* Tipo Exportado */
typedef struct no { //lista com nó cabeça
  int v;
  struct no* prox;
} Lista;

/* Funçoes Exportadas */

void cria_lista(Lista** p);
void insere_final(Lista* p, int valor);
void insere_comeco(Lista* p, int valor);

void remove_final(Lista* p);
void remove_comeco(Lista* p);

void imprime_lista(Lista* p);
void libera_lista(Lista** p);
```

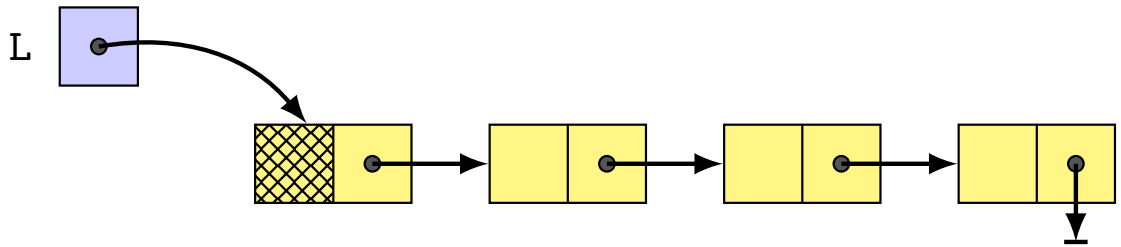
TAD



minhaLista.c

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "minhaLista.h"
4
5  void cria_lista(Lista** p){
6    Lista *q = (Lista*) malloc(sizeof(Lista));
7    q->v = 0;
8    q->prox = NULL;
9    *p = q;
10 }
11 ...
```

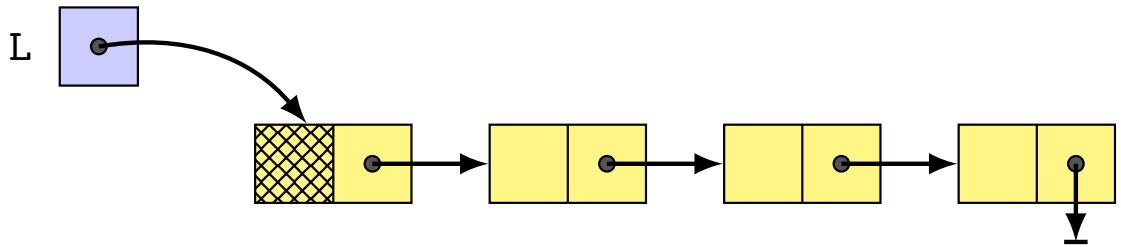
TAD



minhaLista.c

```
1  ...
2  void insere_final(Lista* p, int valor){
3    Lista *q, *aux;
4    q = (Lista*) malloc(sizeof(Lista));
5    q->v = valor;
6    q->prox = NULL;
7
8    aux = p;
9    while (aux->prox != NULL) aux = aux->prox;
10   aux->prox = q;
11 }
12 ...
```

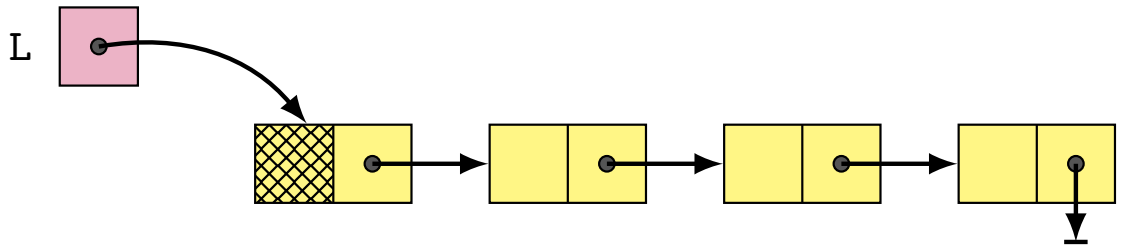
TAD



minhaLista.c

```
1 ...
2 void insere_comeco(Lista* p, int valor){
3   Lista* q = (Lista*) malloc(sizeof(Lista));
4   q->v = valor;
5   q->prox = p->prox;
6   p->prox = q;
7 }
8 ...
```

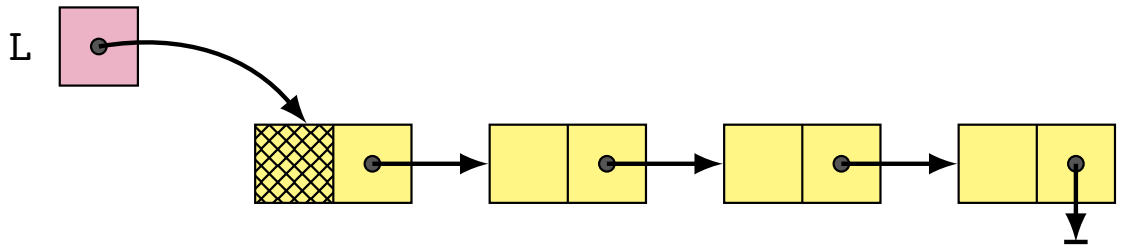
TAD



minhaLista.c

```
1  ...
2  void remove_final(Lista* p){
3    Lista* q = p;
4    if(q->prox!=NULL){
5      while (q->prox->prox != NULL)
6        q = q->prox;
7      Lista* aux = q->prox;
8      q->prox = NULL;
9      free(aux);
10   }
11 }
12 ...
```

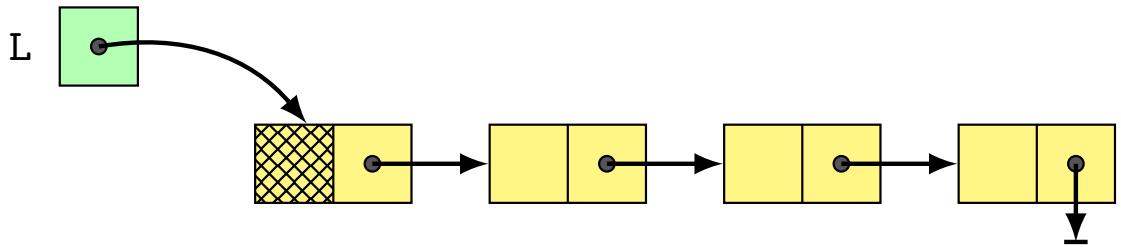
TAD



minhaLista.c

```
1  ...
2  void remove_comeco(Lista* p){
3    Lista* q;
4    if (p->prox != NULL){
5      q = p->prox;
6      p->prox = q->prox;
7      free(q);
8    }
9  }
10 ...
```

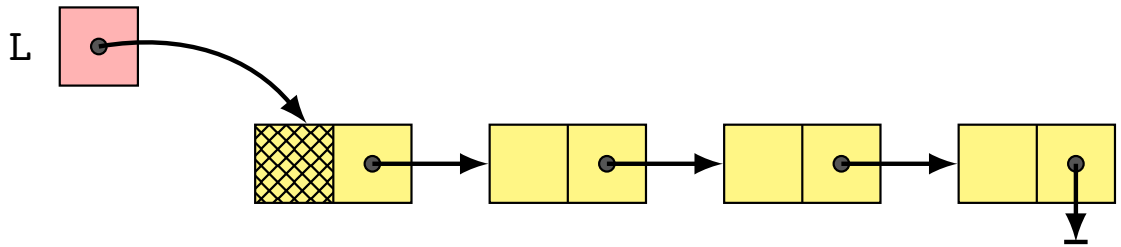
TAD



minhaLista.c

```
1 ...
2 void imprime_lista(Lista* p){
3   Lista *q;
4   for (q = p->prox; q != NULL; q = q->prox)
5     printf ("%d\n", q->v);
6 }
7 ...
```

TAD



minhaLista.c

```
1 ...
2 void libera_lista(Lista** p){
3   Lista* q;
4   while (*p != NULL) {
5     q = *p;
6     *p = (*p)->prox;
7     free(q);
8   }
9 }
```

TAD

meuPrograma.c

```c
#include <stdio.h>
#include <stdlib.h>
#include "minhaLista.h"

int main(){
  int i;
  Lista *L;

  cria_lista(&L);
  for(i=1; i<10; i++)
    insere_comeco(L, i);

  remove_final(L);
  remove_comeco(L);
  imprime_lista(L);
  libera_lista(&L);

return 0;
}
```

Como compilar?

Temos três arquivos diferentes:

- `meuPrograma.c` contém a função `main`
- `minhaLista.c` contém a implementação
- `minhaLista.h` contém a interface

Vamos compilar por partes:

```
1 gcc -Wall -Werror -c meuPrograma.c
```

- vai gerar o arquivo compilado `meuPrograma.o`

```
1 gcc -Wall -Werror -c minhaLista.c
```

- vai gerar o arquivo compilado `minhaLista.o`

```
1 gcc meuPrograma.o minhaLista.o -lm -o meuPrograma
```

- faz a linkagem, gerando o executável `meuPrograma`

Makefile

É mais fácil usar um `Makefile` para compilar:

```
1  all: meuPrograma
2
3  meuPrograma: meuPrograma.o minhaLista.o
4          gcc meuPrograma.o minhaLista.o -lm -o meuPrograma
5
6  meuPrograma.o: meuPrograma.c minhaLista.h
7          gcc -Wall -Werror -c meuPrograma.c
8
9  minhaLista.o: minhaLista.c minhaLista.h
10         gcc -Wall -Werror -c minhaLista.c
```

Basta executar `make` na pasta com os arquivos:

- `meuPrograma.c`, `minhaLista.c`, `minhaLista.h`
- `Makefile`

Apenas recompila o que for necessário!

Fim

Dúvidas?

Roteiro

Referências

1. Feofiloff, Paulo. Algoritmos em linguagem C. Elsevier Brasil, 2009.
2. Materiais adaptados dos slides dos Profs. Rafael Schouery e Lehilton L. C. Pedrosa, da UNICAMP.