DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Tipos Abstractos de Dados

Sedgewick: Capítulo 4

IAED

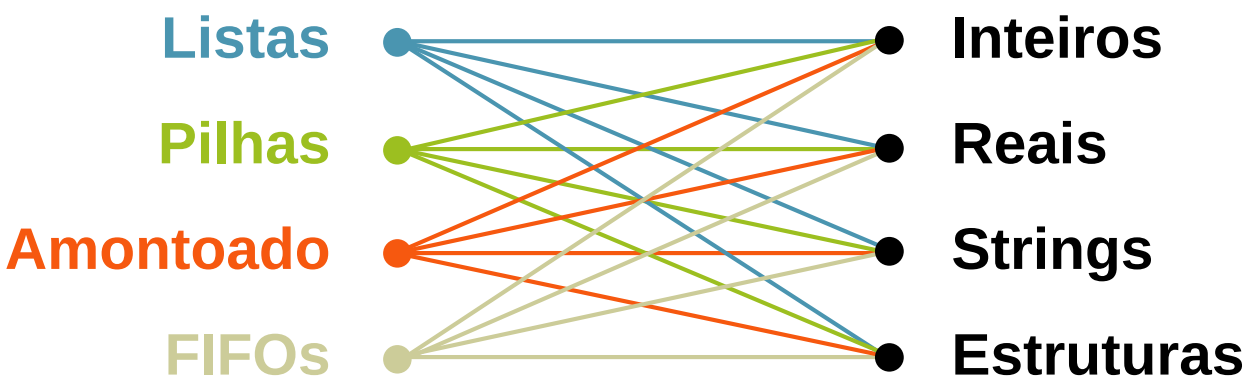
Tipos Abstractos de Dados

- Motivação
- Objectos
- Pilhas
- Exemplos de clientes
- ADTs para FIFOs e filas

- Tipos Abstractos de Dados = Abstract Data Types = ADTs

Motivação

- Mesmas estruturas são usadas com vários tipos de dados



- O procedimento para inserir um inteiro, real, uma string ou uma estrutura numa lista é similar
- O código pode (e deve) ser re-utilizado
- Abstracção dos detalhes de implementação

ADT e Colecções de Objectos

- ADTs são úteis para manipular colecções de objectos

- Operações típicas
  - Comparações entre objectos
  - Operações de entrada e saída (leitura e escrita)
  - Inserção em colecções
  - Apagamento de colecções
  - Alteração de propriedades (e.g., prioridade)

- ADTs deste tipo são denominados *filas generalizadas*

Vantagens do Uso de ADTs

- Solução elegante
- Separa os problemas:
  - Alto nível: interface de operações sobre tipo de dados
  - Baixo nível: como manter as estruturas de dados
- Permite comparar diferentes implementações
- Permite re-utilizar o código

- *Para utilizarmos ADTs, vamos aprender um pouco mais de C…* cada uma destas ADTs é em geral implementada em **ficheiros diferentes**

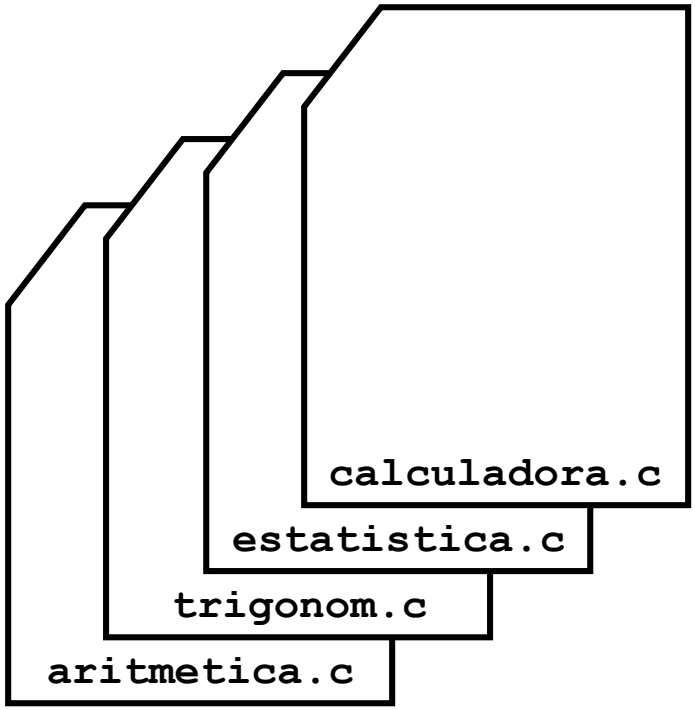
DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Regras de *Scope*

IAED

Organização de Programas

- Programas normalmente divididos em vários ficheiros
- Cada ficheiro permite implementar conjunto de funcionalidades relacionadas



Regras de *Scope*

- Em C todas as variaveis tem um **scope**, i.e., um âmbito de acessibilidade & visibilidade, que podem tomar 1 de 4 (na realidade 3) estados:
  - ***Bloco (variáveis locais)***
  - ~~***Função***~~
  - ***Ficheiro (variáveis globais apenas visíveis dentro do ficheiro onde são declaradas)***
  - ***Programa (variáveis globais visíveis em múltiplos ficheiros)***

Block scope

- Em C todas as variaveis tem um **scope**, i.e., um âmbito de acessibilidade & visibilidade
  - ***Bloco***

```
int soma(int v[], int n) {
   int i, soma = 0;
   for (i = 0; i < n; i++)
       soma += v[i];
   return soma;
}
```

Block scope

- Em C todas as variaveis tem um **scope**, i.e., um âmbito de acessibilidade & visibilidade
  - ***Bloco***
- A utilização do qualificador **`static`** permite manter o valor da variável entre chamadas à função

```
int soma(int v[], int n) {
   int i,
   static int soma=0;
   for (i = 0; i < n; i++)
       soma += v[i];
   return soma;
}
```

Block scope

- Em C todas as variaveis tem um **scope**, i.e., um âmbito de acessibilidade & visibilidade
  - ***Bloco***

```
void bubble(int a[], int l, int r) {
  int i, j;
  for (i = l; i < r; i++)
    for (j = r; j > i; j--)
      if (a[j-1] > a[j]) {
            int t = a[j-1];
            a[j-1] = a[j];
            a[j] = t;
      }
}
```

File & Program scope: “variáveis globais”

- Um programa C pode ser composto por conjunto de objectos externos, que podem ser variáveis ou funções
- Podem ser utilizadas por qualquer função, ao contrário de variáveis *internas/locais*, que apenas podem ser utilizadas dentro da uma função ou bloco

```
int acumulador;

void soma(int valor) {
  acumulador += valor;
}
```

File scope: “variáveis globais estáticas”

- Variáveis globais definidas como estáticas permitem limitar o seu *scope* ao ficheiro em que são definidas
- Funções também podem ser definidas como estáticas:
  - Limita scope da função entre ponto da definição e fim do ficheiro onde definição ocorre

```
static int acumulador;

void soma(int valor) {
  acumulador += valor;
}
```

Program scope: “variáveis externas”

- Uma variável externa é *definida* quando são indicadas as propriedades da variável, e quando são especificados os seus requisitos em termos de memória

```
int a;
```

- Uma variável externa é *declarada* quando apenas são indicadas as suas propriedades

```
extern int a;
```

- Uma variável apenas pode ter uma definição, embora possa ser declarada várias vezes
  - Dimensão de um array obrigatória na definição do array, mas opcional na declaração
  - Inicialização de uma variável externa apenas pode ter lugar na definição da variável

De uma forma geral, temos que…

- Os programas são divididos em vários módulos

- **Cada módulo providencia um conjunto de funcionalidades bem definido** e coerente, incluindo a especificação, a documentação e os testes de validação necessários.

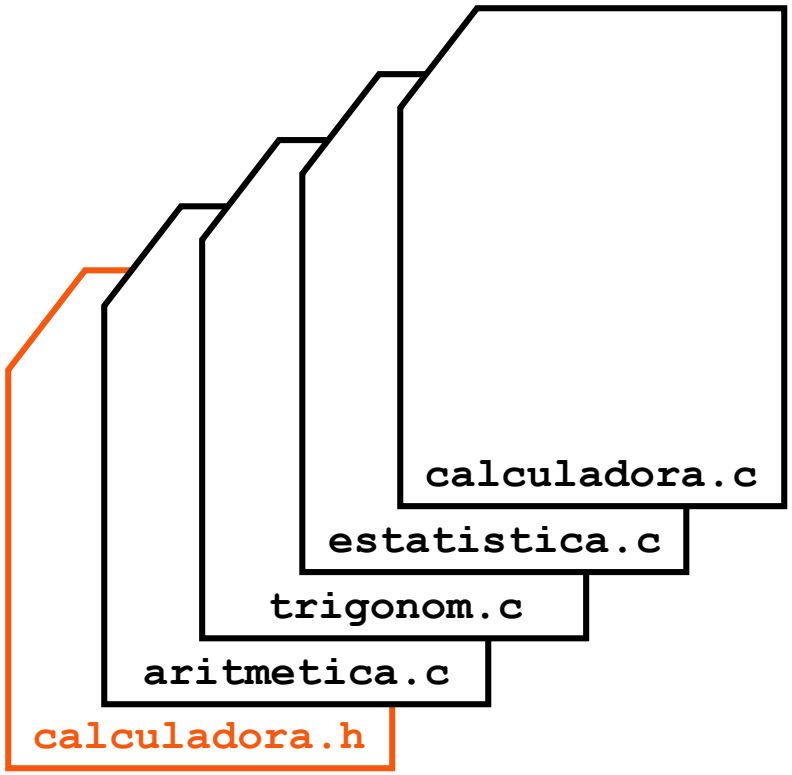
- No caso da programação em C, cada um desses módulos é em geral implementado em **ficheiros diferentes** e, de facto, cada módulo pode ser subdividido também por vários ficheiros.

De uma forma geral, temos que…

- Existem em geral dois tipos de ficheiros:
  - ***header files***, ou ficheiros de cabeçalho, onde devem ser incluídas todas as declarações partilhadas para um dado módulo, incluindo o cabeçalho das funções e os tipos de dados;
  - ***source files***, ou ficheiros fonte, onde devem ser implementadas todas as funções.

Header Files (Ficheiros de Cabeçalho)

- São utilizados para incluirem **todas** as declarações partilhadas por mais de um ficheiro.



Exemplo 1: “variáveis externas”

- Variáveis externas
  - utilizadas antes de serem definidas, ou definidas noutro ficheiro, deverão ser declaradas com a palavra-chave `extern`

```
#include “stuff.h”

int x;

int main()
{
    x=2;
    print();
    return 0;
}
```

main.c

```
#ifndef _STUFF_
#define _STUFF_

#include <stdio.h>

void print();

extern int x;

#endif
```

stuff.h

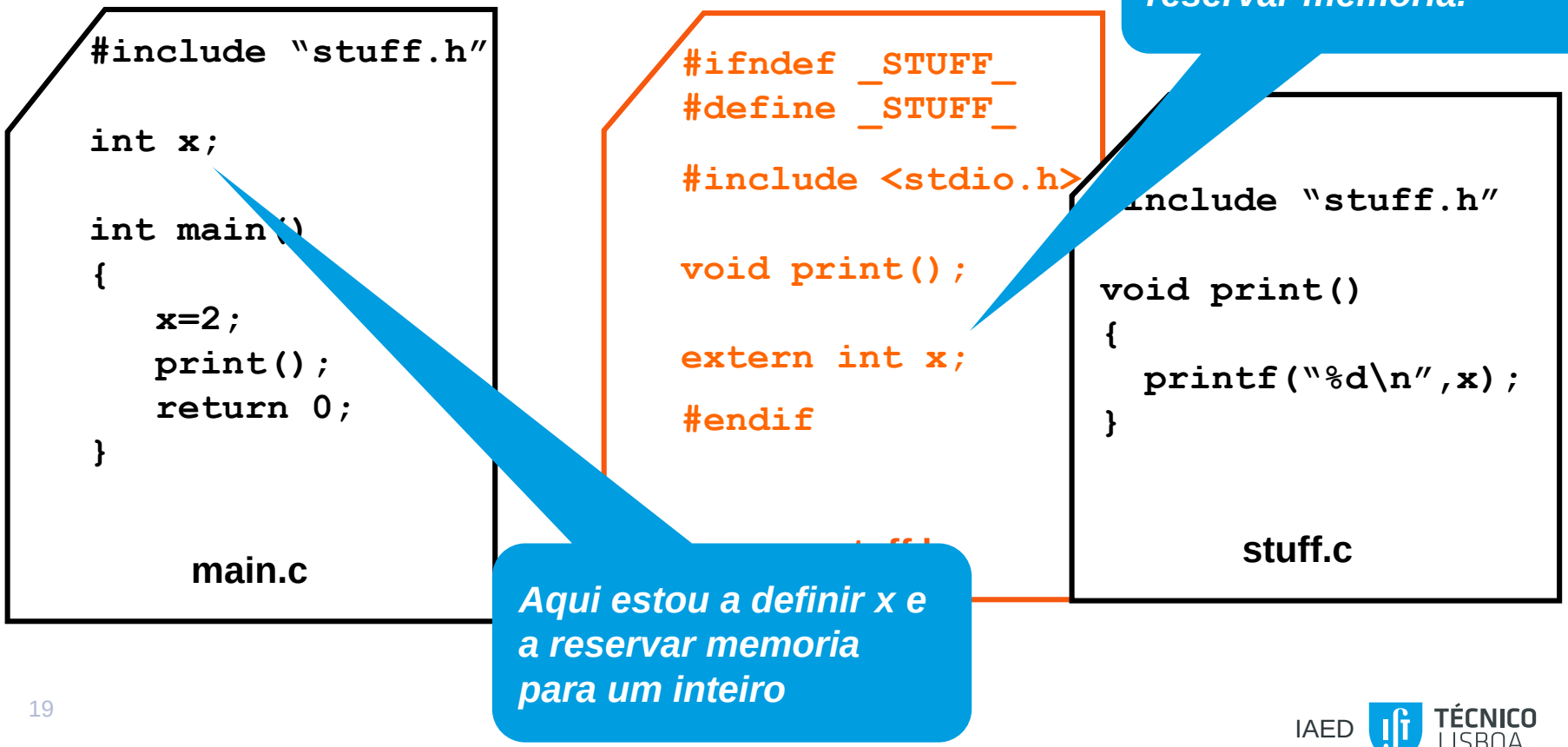
```
#include “stuff.h”

void print()
{
  printf(“%d\n”,x);
}
```

stuff.c

Exemplo 1: “variáveis externas”

- Variáveis externas
  - utilizadas antes de serem definidas, ou definidas noutro ficheiro deverão ser declaradas com a palavra-chave e



Exemplo 2: Módulo para números complexos

- *complexos.h*

Garante que este ficheiro é incluído apenas uma vez.

```
#ifndef _COMPLEXOS_H_
#define _COMPLEXOS_H_

typedef struct {
  float real, img;
} complexo;

complexo soma(complexo a, complexo b);
complexo le_complexo();
void escreve_complexo(complexo a);

#endif
```

Exemplo 2: Módulo para números complexos

- *complexos.c*

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

complexo soma(complexo a, complexo b)
{
  a.real += b.real;
  a.img += b.img;
  return a;
}
complexo le_complexo()
{
  complexo a;
  char sign;
  scanf("%f%c%fi", &a.real, &sign, &a.img);
  if (sign == '-') a.img *= -1;
  return a;
}
void escreve_complexo(complexo a)
{
  if (a.img >= 0) printf("%f+%fi", a.real, a.img);
  else printf("%f%fi", a.real, a.img);
}
```

Exemplo 2: Módulo para números complexos

- *main.c*

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

int main()
{
  complexo x, y, z;
  x = le_complexo();
  y = le_complexo();
  z = soma(x, y);

  escreve_complexo(x);
  printf(" + ");
  escreve_complexo(y);
  printf(" = ");
  escreve_complexo(z);
  printf("\n");

  return 0;
}
```

Quando começa a ficar mais complicado devemos usar o Makefile (ver lab01)

Compilação!

```
$ gcc –Wall -o complexos main.c complexos.c
```

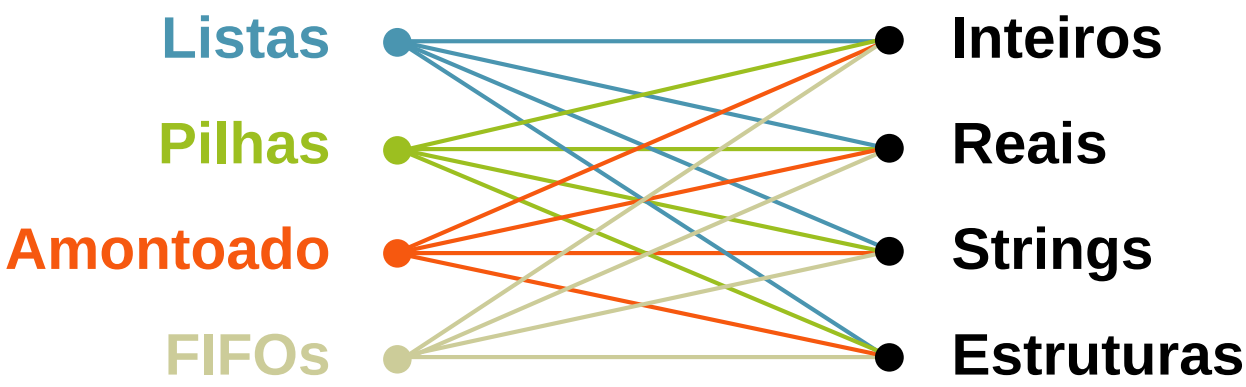
DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Vamos voltar aos tipos de dados abstractos...

IAED

Agora que já sabemos trabalhar código com vários ficheiros... Podemos montar ADTs

- Mesmas estruturas são usadas com vários tipos de dados



- O procedimento para inserir um inteiro, real, uma string ou uma estrutura numa lista é similar
- O código pode (e deve) ser re-utilizado
- Abstracção dos detalhes de implementação

Projecto de ADTs

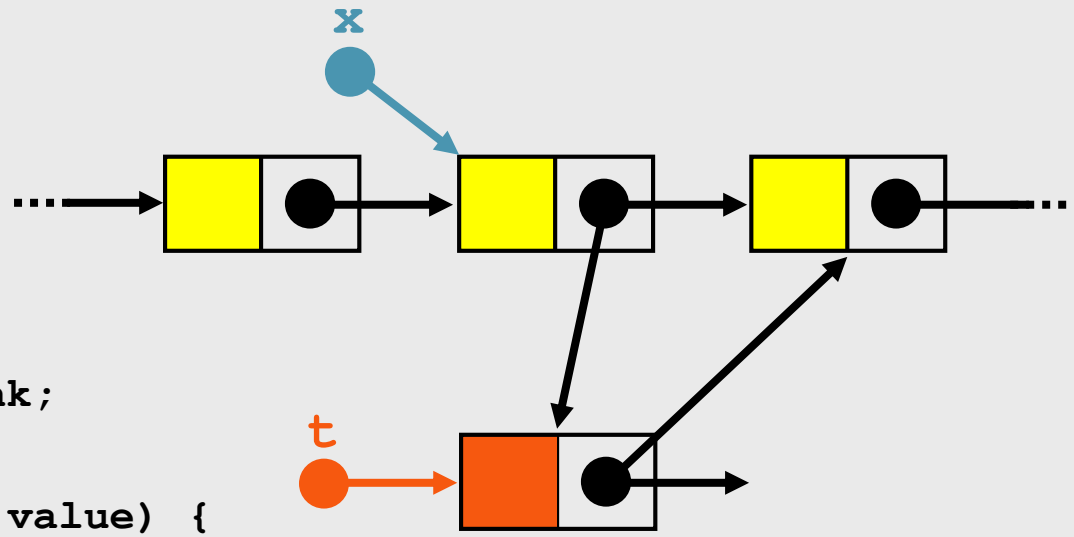
- Três problemas separados:
  - Definição da interface
  - Implementação do ADT
  - Projecto e implementação do cliente

Inserção de Inteiro

```
struct node {
  int value;
  struct node *next;
};
typedef struct node *link;

void insert(link x, int value) {
  link t;

  t = (link) malloc(sizeof(struct node));
  t->value = value;
  t->next = x->next;
  x->next = t;
}
```

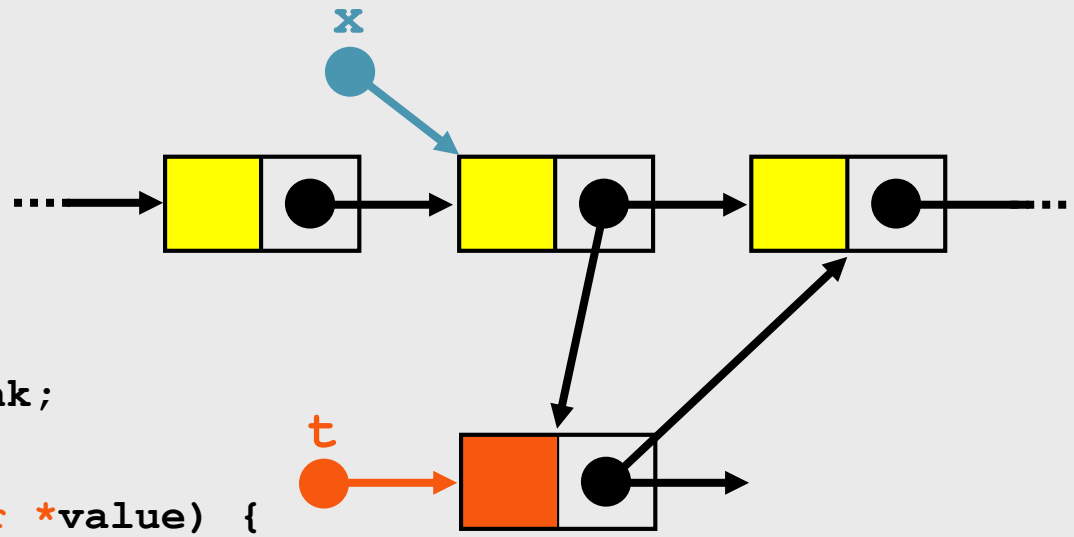


Inserção de String

```
struct node {
  char *value;
  struct node *next;
};
typedef struct node *link;

void insert(link x, char *value) {
  link t;

  t = (link) malloc(sizeof(struct node));
  t->value = strdup(value);
  t->next = x->next;
  x->next = t;
}
```

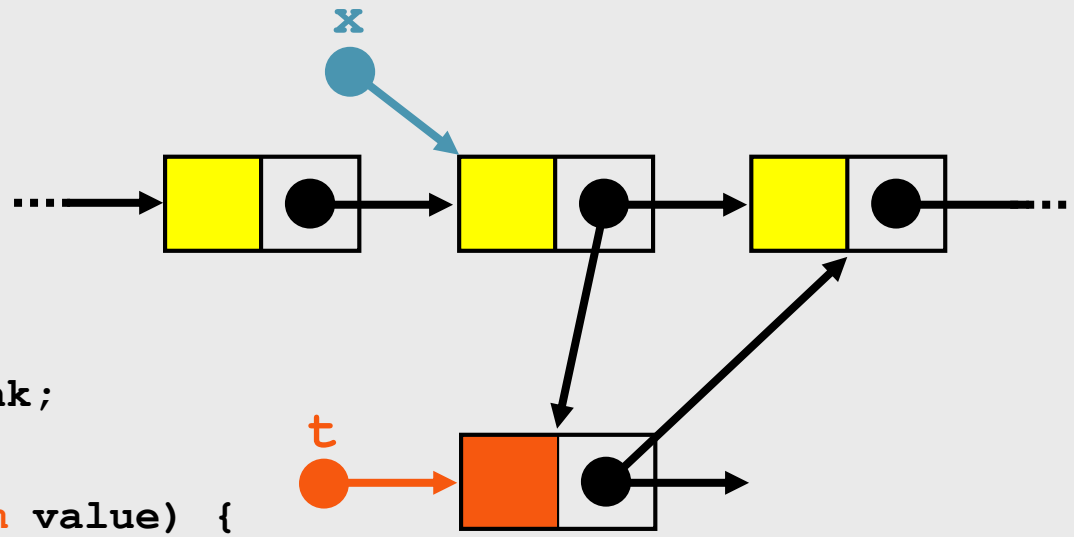


Inserção de Elemento Genérico do Tipo `Item`

```
struct node {
  Item value;
  struct node *next;
};
typedef struct node *link;

void insert(link x, Item value) {
  link t;

  t = (link) malloc(sizeof(struct node));
  t->value = NEWitem(value);
  t->next = x->next;
  x->next = t;
}
```



Inserção de Inteiro

```c
#include "Item.h"

struct node {
  Item value;
  struct node *next;
};
typedef struct node *link;

void insert(link x, Item value) {
  link t;

  t = (link) malloc(sizeof(struct node));
  t->value = NEWitem(value);
  t->next = x->next;
  x->next = t;
}
```

```c
typedef int *Item;

int* NEWitem(int* value)
{
  int *v=(int*)malloc(sizeof(int));
  *v=*value;
  return v;
}
```

Item.h / Item.c

Inserção de String

```
#include "Item.h"

struct node {
  Item value;
  struct node *next;
};
typedef struct node *link;

void insert(link x, Item value) {
  link t;

  t = (link) malloc(sizeof(struct node));
  t->value = NEWitem(value);
  t->next = x->next;
  x->next = t;
}
```

```
typedef char *Item;

char* NEWitem(char *value)
{
  return strdup(value);
}
```

Item.h / Item.c

Inserção de Elemento Genérico do Tipo `Item`

```
#include "Item.h"

struct node {
  Item item;
  struct node *next;
};
typedef struct node *link;

void insert(link x, Item item) {
  link t;

  t = (link) malloc(sizeof(struct node));
  t->item = NEWitem(item);
  t->next = x->next;
  x->next = t;
}
```

Esta implementação assume que fazemos uma cópia do "item" passado como argumento

Inserção de Elemento Genérico do Tipo `Item`

```
#include "Item.h"

struct node {
  Item item;
  struct node *next;
};
typedef struct node *link;

void insert(link x, Item item) {
  link t;

  t = (link) malloc(sizeof(struct node));
  t->item = item;
  t->next = x->next;
  x->next = t;
}
```

Contudo, se assumirmos que o argumento já tem o pointer para o item a ser guardado, basta-nos fazer a atribuição

Neste caso, o NEWitem é chamado antes de chamar o insert.

Inserção de Elemento Genérico do Tipo `Item`

```
#include "Item.h"

struct node {
  Item item;
  struct node *next;
};
typedef struct node *link;

void insert(link x, Item item) {
  link t;

  t = (link) malloc(sizeof(struct node));
  t->item = item;
  t->next = x->next;
  x->next = t;
}
```

Também funciona se o Item for um int, um float, etc.

Outros Exemplos de Abstracção

- Comparação
  - Para inteiro, operação `x1 == x2`
  - Para string, operação `!strcmp(x1,x2)`

- Para Inteiro

```
typedef int Item;
#define eq(A,B) (A==B)
```

- Para String

```
typedef char* Item;
#define eq(A,B) (!strcmp(A,B))
```

exemplo: ADT Stack/pilha (LIFO)

- Definição da Interface

STACK.h

```
void STACKinit(int);
int STACKempty();
void STACKpush(Item);
Item STACKpop();
```

Temos (pelo menos) 2 formas de implementar uma Stack/ Pilha ! ...com o mesmo interface!

Tabelas / Vectores

Listas

ADT Stack - Implementação com Tabela

STACK.c

```
#include <stdlib.h>
#include "Item.h"
#include "STACK.h"
static Item *s;
static int N;

void STACKinit(int maxN) {
  s = (Item *)malloc(maxN*sizeof(Item));
  N = 0;
}
int STACKempty() {
  return N == 0;
}
void STACKpush(Item item) {
  s[N++] = item;
}
Item STACKpop() {
  return s[--N];
}
```

ADT Stack - Implementação com Lista

STACK.c

```
#include <stdlib.h>
#include "Item.h"
#include "STACK.h"

struct STACKnode {
  Item item;
  struct STACKnode* next;
};
typedef struct STACKnode* link;

static link head;

link NEWnode(Item item, link next) {
  link x = (link)malloc(sizeof(STACKnode));
  x->item = item;
  x->next = next;
  return x;
}
```

ADT Stack - Implementação com Lista

STACK.c

```
void STACKinit(int maxN) {
  head = NULL;
}
int STACKempty() {
  return head == NULL;
}
void STACKpush(Item item) {
  head = NEWnode(item, head);
}
Item STACKpop() {
  Item item = head->item;
  link t = head->next;
  free(head);
  head = t;
  return item;
}
```

Outro exemplo: filas de espera (FIFO)

Interface do ADT *Queue*

QUEUE.h

```
void QUEUEinit(int);
int  QUEUEempty();
void QUEUEput(Item);
Item QUEUEget();
```

insiro no fim.

Retiro do início

Mais uma vez, temos (pelo menos) 2 formas de implementar uma fila de espera ... com o mesmo interface!

Tabelas / Vectores

Listas

Implementação do ADT FIFO (Listas)

QUEUE.c

```
#include <stdlib.h>

#include "Item.h"
#include "QUEUE.h"

struct QUEUEnode {
  Item item;
  struct QUEUEnode* next;
};
typedef struct QUEUEnode* link;

static link head, tail;
```

Implementação do ADT FIFO (Listas)

QUEUE.c

```
void QUEUEinit(int maxN) {
  head = NULL;
  tail = NULL;
}

int QUEUEempty() {
  return head == NULL;
}

link NEWnode(Item item, link next) {
  link x = (link) malloc(sizeof(struct QUEUEnode));

  x->item = item;
  x->next = next;
  return x;
}
```

Inicio a head e a tail a NULL

Implementação do ADT FIFO (Listas)

QUEUE.c (cont)

```
void QUEUEput(Item item) {
  if (head == NULL) {
    head = (tail = NEWnode(item, head));
    return;
  }
  tail->next = NEWnode(item, tail->next);
  tail = tail->next;
}

Item QUEUEget() {
  Item item = head->item;
  link t = head->next;
  free(head);
  head = t;
  return item;
}
```

Quando a fila está vazia...

Nos outros casos, insiro no fim.

Actualizo a tail.

Guardo o conteudo do primeiro elemento numa variavel auxiliar.

Liberto a memória e retorno o elemento removido

Implementação do ADT FIFO (Tabelas)

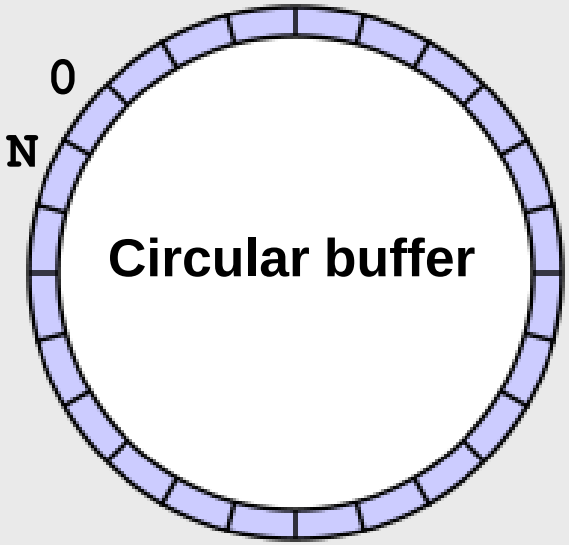
QUEUE.c

```
#include <stdlib.h>

#include "Item.h "
#include "QUEUE.h"

static Item *q;
static int N, head, tail;

void QUEUEinit(int maxN) {
  q = (Item *) malloc((maxN+1)*sizeof(Item));
  N = maxN+1;
  head = N;
  tail = 0;
}
```



Para podermos distinguir entre full & empty queue...

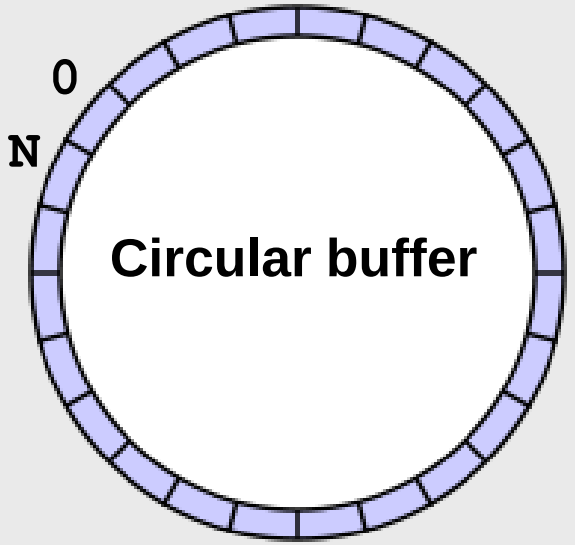
Implementação do ADT FIFO (Tabelas)

QUEUE.c (cont)

```
int QUEUEempty() {
  return head % N == tail;
}

void QUEUEput(Item item) {
  q[tail++] = item;
  tail = tail % N;
}

Item QUEUEget() {
  head = head % N;
  return q[head++];
}
```



ADTs de 1ª Ordem

- Nos ADTs estudados só é possível ter uma instância da estrutura de dados
  - A informação da estrutura de dados é guardada em variáveis globais
  - As funções operam sobre uma única estrutura de dados

- Como fazer quando pretendemos ter várias instâncias do mesmo ADT ?
  - Exemplo: múltiplas FIFOS

ADTs de 1ª Ordem

- **Solução:** em vez da informação da estrutura de dados ser guardada em variáveis globais, é guardada numa estrutura que é passada como argumento a cada função

QUEUE.h

```
struct QUEUEnode { Item item; link next; };
struct queue { link head; link tail; };
typedef struct queue *Q;
typedef struct QUEUEnode *link;

Q     QUEUEinit(int maxN);
int   QUEUEempty(Q);
void  QUEUEput(Q, Item);
Item  QUEUEget(Q);
```

Podemos adicionar outras informações (e.g., "length")

QUEUE.h (original)

```
typedef struct QUEUEnode* link;

struct QUEUEnod
  Item item; li

static link hea
```

QUEUE.h (original)

```
void QUEUEinit(int);
int  QUEUEempty();
void QUEUEput(Item);
Item QUEUEget();
```

Implementação do ADT FIFO de 1ª Ordem

QUEUE.c (cont)

```
#include <stdlib.h>

#include "Item.h"
#include "QUEUE.h"



link NEWnode(Item item, link next) {
  link x = (link) malloc(sizeof(struct QUEUEnode));

  x->item = item;
  x->next = next;
  return x;
}
```

QUEUE.c (original)

```
link NEWnode(Item item, link next) {
  link x = (link) malloc(sizeof(struct QUEUEnode));

  x->item = item;
  x->next = next;
  return x;
}
```

Implementação do ADT FIFO de 1ª Ordem

QUEUE.c (cont)

```
Q QUEUEinit(int maxN) {
  Q q = (Q)malloc(sizeof(struct queue));
  q->head = NULL;
  q->tail = NULL;
  return q;
}

int QUEUEempty(Q q) {
  return q->head == NULL;
}
```

QUEUE.c (original)

```
void QUEUEinit(int maxN) {
  head = NULL;
  tail = NULL;
}

int QUEUEempty() {
  return head == NULL;
}
```

Implementação do ADT FIFO de 1ª Ordem

QUEUE.c (cont)

```
void QUEUEput(Q q, Item item) {
  if (q->head == NULL) {
    q->head = (q->tail = NEWnode(item, q->head));
    return;
  }
  q->tail->next = NEWnode(item, q->tail->next);
  q->tail = q->tail->next;
}
```

QUEUE.c (original)

```
void QUEUEput(Item item) {
  if (head == NULL) {
    head = (tail = NEWnode(item, head));
    return;
  }
  tail->next = NEWnode(item, tail->next);
  tail = tail->next;
}
```

Implementação do ADT FIFO de 1ª Ordem

QUEUE.c (cont)

```
Item QUEUEget(Q q) {
  Item item = q->head->item;
  link t = q->head->next;
  free(q->head);
  q->head = t;
  return item;
}
```

QUEUE.c (original)

```
Item QUEUEget() {
  Item item = head->item;
  link t = head->next;
  free(head);
  head = t;
  return item;
}
```

Implementação do ADT FIFO de 1ª Ordem

QUEUE.c (cont)

```
Item QUEUEget(Q q) {
  Item item = q->head->item;
  link t = q->head->next;
  free(q->head);
  q->head = t;
  return item; /* esta função retorna o item de forma a que a sua
   informação seja processada (se for o caso) e, se necessário, a
   sua memoria seja libertada posteriormente  */
}
```

Cliente do ADT FIFO de 1ª Ordem

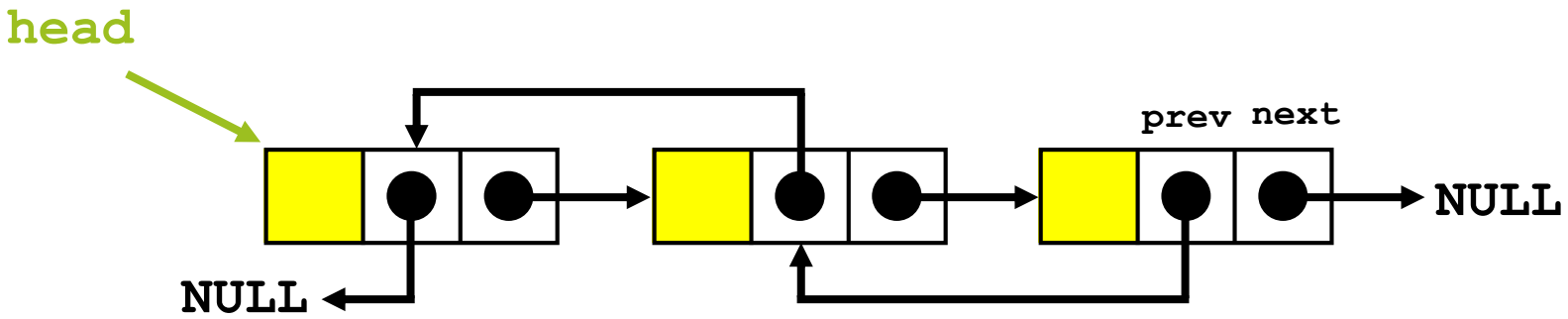
```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "QUEUE.h"
#define M 10

int main(int argc, char *argv[]) {
  int i, j, N = atoi(argv[1]);
  Q queues[M];

  for (i = 0; i < M; i++)
    queues[i] = QUEUEinit(N);
  for (i = 0; i < N; i++)
    QUEUEput(queues[rand() % M], i);
  for (i = 0; i < M; i++) {
    for (j = 0; !QUEUEempty(queues[i]); j++)
      printf("%3d ", QUEUEget(queues[i]));
    printf("\n");
  }
  return 0;
}
```

Conseguem imaginar uma forma alternativa de implementar uma queue FIFO?

- Usando listas duplamente ligadas
- Guardando apenas a **`head`** (i.e., não temos a **`tail`**)
- Inserção e remoção: $O(1)$



Conseguem imaginar uma forma alternativa de implementar uma queue FIFO?

- Usando listas duplamente ligadas
- Guardando apenas a **`head`** (i.e., não temos a **`tail`**)
- Inserção e remoção: $O(1)$
- A **`tail`** é sempre a **`head->prev`**