DEI
DEPARTAMENTO
DE ENGENHARIA INFORMÁTICA
TÉCNICO LISBOA

Dicas para o projecto 2

IAED, 2016/2017

Processamento de dados em tempo real

registar as entradas e saídas de produtos num armazém. Cada produto tem um código e um número de unidades.

Comando	Descrição
`a`	Incrementa ou reduz o número de unidades associadas ao produto com o código dado. Se o código não existir deverá ser criado um novo produto com esse código.
`l`	Lista alfabeticamente todos os produtos.
`m`	Escreve o produto com o maior número de unidades em stock.
`r`	Remove o produto com o código dado.
`x`	Termina o programa

Comando *a* (adicionar, s/output)

- Input:

```
a 60fdba63 4
a c614e44d 149
a ff4095a1 38
a c614e44d 36
a 5d5c3b04 28
a 60fdba63 247
a 47cd69e4 223
a 376f7f4d 43
a 376f7f4d -14
```

Número inteiro
(unidades)

Código: Sequência de 8 chars do conjunto (sem espaços)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f}

- Output:

Não tem output!!

Comando *a* (adicionar, s/output)

- Input:

```
a 60fdba63 4
a c614e44d 149
a ff4095a1 38
a c614e44d 36
a 5d5c3b04 28
a 60fdba63 247
a 47cd69e4 223
a 376f7f4d 43
a 376f7f4d -14
```

- Output:

Para cada entrada, deverá:

- Procurar se a entrada já existe.
- Se existir, incrementa o número de unidades
- Se não existir, adiciona uma nova entrada.

Número inteiro (unidades)

Código: Sequência de 8 chars do conjunto (sem espaços)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f}

Comando *l* (listar alfabéticamente)

- Input:

```
a 60fdba63 4
a c614e44d 149
a ff4095a1 38
a c614e44d 36
a 5d5c3b04 28
a 60fdba63 247
a 47cd69e4 223
a 376f7f4d 43
a 376f7f4d -14
l
```

- Output:

```
376f7f4d 29
47cd69e4 223
5d5c3b04 28
60fdba63 251
c614e44d 185
ff4095a1 38
```

Listagem por ordem alfabética

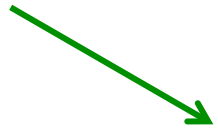
Comando *x* (sair)

- Input:

```
a 60fdba63 4
a c614e44d 149
a ff4095a1 38
a c614e44d 36
a 5d5c3b04 28
a 60fdba63 247
a 47cd69e4 223
a 376f7f4d 43
a 376f7f4d -14
l
x
```

- Output:

```
376f7f4d 29
47cd69e4 223
5d5c3b04 28
60fdba63 251
c614e44d 185
ff4095a1 38
6
```



Escreve o número total de produtos

Outro exemplo: Comando *r* (remover prod.)

- Input:

```
a 60fdba63 4
a c614e44d 149
a ff4095a1 38
a 60fdba63 -5
l
r c614e44d
a 90adba63 4
l
x
```

- Output:

```
60fdba63 0
c614e44d 149
ff4095a1 38
60fdba63 0
90adba63 4
ff4095a1 38
3
```

Outro exemplo: Comando *m* (máximo)

- Input:

```
a 60fdba63 24
a c614e44d 2
a ff4095a1 38
a c614e44d 36
a 5d5c3b04 28
a 47cd69e4 33
a 376f7f4d 43
a 376f7f4d -10
l
m
a 5d5c3b04 10
l
m
r 5d5c3b04
m
x
```

- Output:

```
376f7f4d 33
47cd69e4 33
5d5c3b04 28
60fdba63 24
c614e44d 38
ff4095a1 38
c614e44d 38
376f7f4d 33
47cd69e4 33
5d5c3b04 38
60fdba63 24
c614e44d 38
ff4095a1 38
5d5c3b04 38
c614e44d 38
5
```

Como organizar os dados?

- Não existe limite para o número de produtos
- Usar estruturas de dados dinâmicas!
  - Listas (simplesmente ligadas?)
  - Tabelas de dispersão?
  - Árvores binárias (de pesquisa? equilibradas?)
  - Priority queues / binary heap?
  - Outra?
- Não se conhece a frequência de utilização de cada comando.

  ➡ Procurar soluções que permitam lidar com todos os comandos eficientemente.

Como organizar os dados?

- Estrutura de dados para as entradas?

Esta estrutura de dados deverá permitir

- Introduzir um novo produto na colecção
- Procurar um elemento já existente
- Obter o produto com mais unidades em stock
- Listar todos os elementos de forma ordenada

Tudo isto de forma eficiente!!!

Pensar na eficiência da solução escolhida

- A eficiência vai ser avaliada (componente avaliação automática & discussão).
- A escolha das estruturas de dados vai ter impacto na eficiência que se pode atingir.

- Complexidade da inserção de um novo produto?
- Complexidade da pesquisa de um produto?
- Complexidade na procura do elemento com maior número de unidades?
- Complexidade na impressão ordenada?

Abstracção de dados - para o 20! ☺

- O uso de ADTs vai ser avaliado (componente qualidade de código).

Separação entre a implementação do ADT e o cliente

- Definição do interface no `ADT.h`, implementação no `ADT.c`
- Cliente apenas usa funções definidas no `ADT.h`

➡ Verificação: posso substituir a implementação `ADT1.c` pela `ADT2.c` (ambas compatíveis com `ADT.h`), e o programa continua a funcionar?

Separação entre o tipo de objectos guardados e a definição e implementação do ADT

- Definição do interface no `Item.h`, implementação no `Item.c`

➡ Verificação: posso substituir o par `Item1.h`/`Item1.c` pelo par `Item2.h`/ `Item2.c` (ambos compatíveis com `ADT.h`), e o programa continua a funcionar?

Estruturas úteis

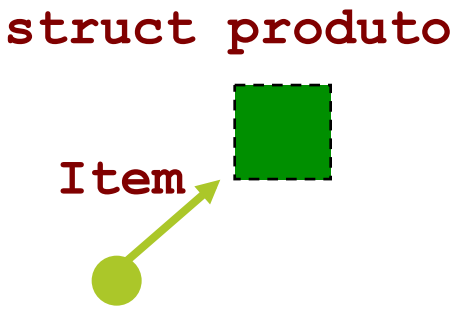
```
typedef struct produto{
    Chave, numero de unidades,
       etc.
}Produto;
```

struct produto

Estruturas úteis

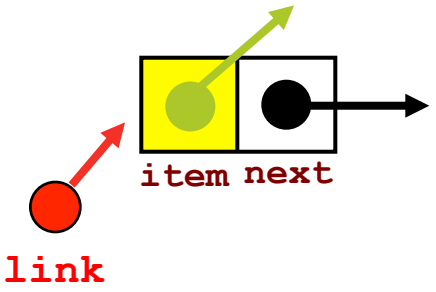
O Item passa a ser um pointer para esta estrutura permitindo uma abstracção conveniente

```
typedef struct produto{
    Chave, numero de unidades,
       etc.
}*Item;
```

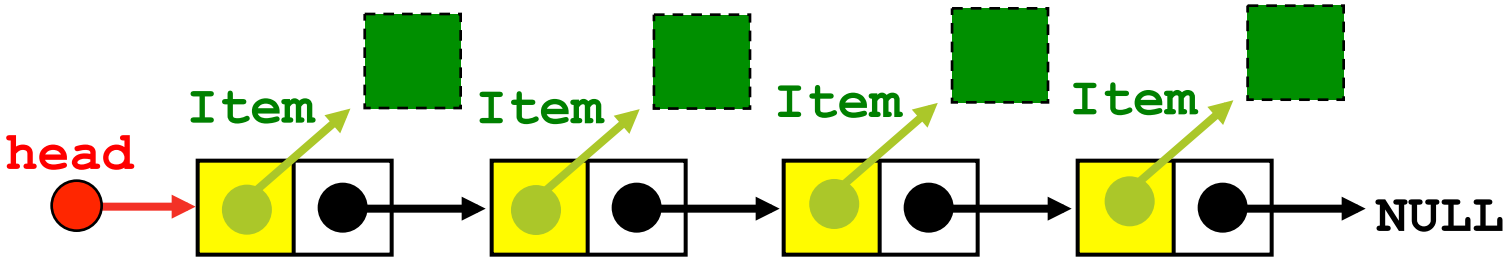


ADT: Estruturas possíveis (ver aulas!)

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

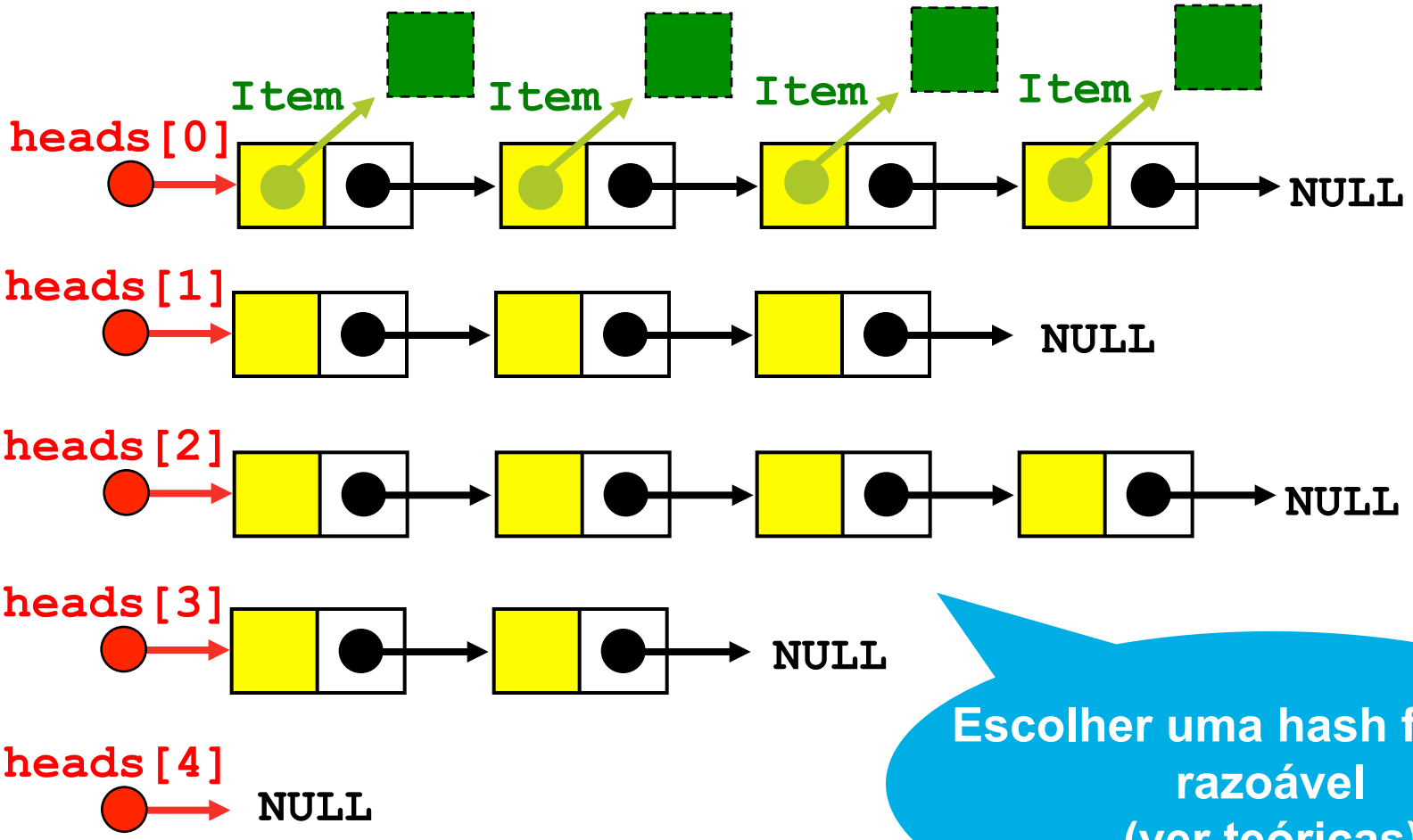


Criamos uma lista com toda a informação



ADT: Estruturas possíveis (ver aulas!)

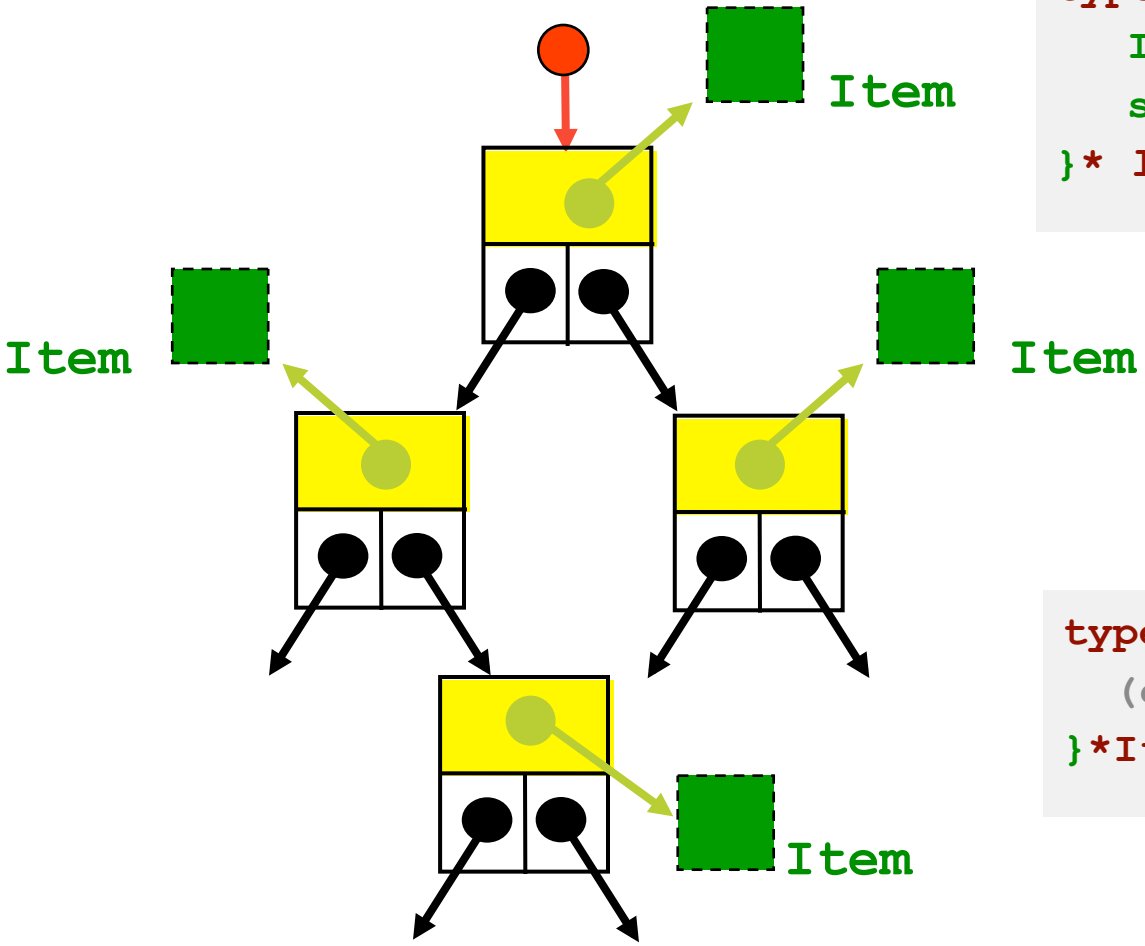
Hash table, por exemplo, com encadeamento externo...



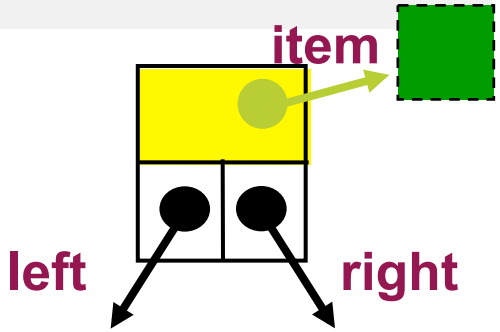
Escolher uma hash function razoável
(ver teóricas)

ADT: Estruturas possíveis (ver aulas!)

Árvores (equilibradas?) de Items... Vamos dar esta semana !



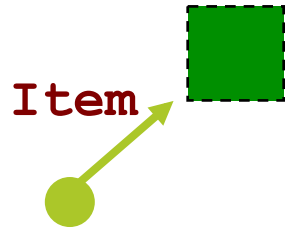

```
typedef struct node{
   Item item;
   struct node *left, *right;
}* link;
```



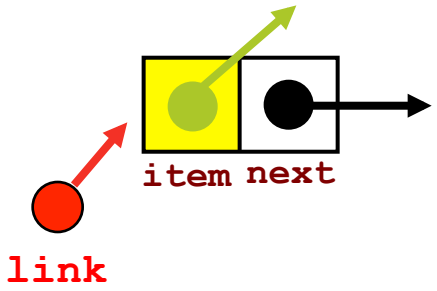

```
typedef struct produto{
  (chave, unidades, ...)
}*Item;
```

Mais abstracção

```
typedef struct produto{
(…)
}*Item;
```



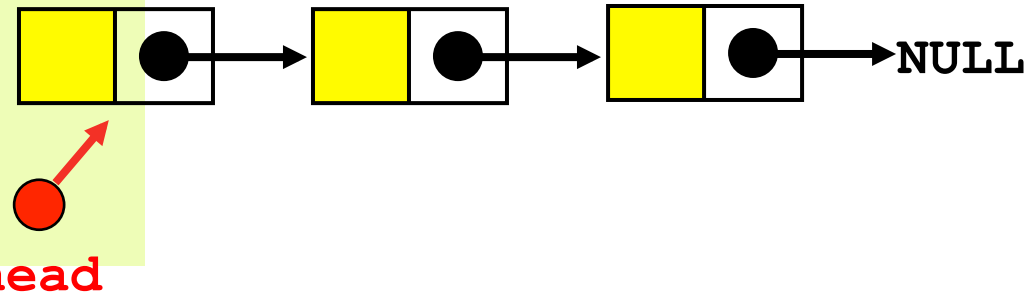

```
typedef struct node{
   void* item;
   struct node*next;
}*link;
```



Mais abstracção (exemplo para uma lista)

Esta também pode dar jeito…

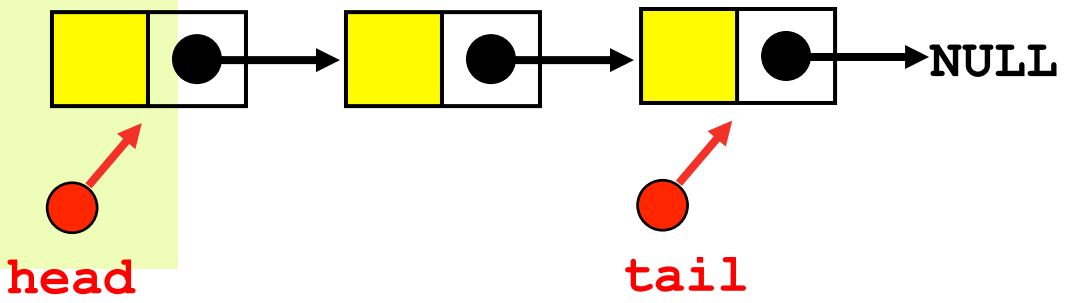
```
typedef struct list{
   link head;
   int size;
}*List;
```



Mais abstracção (exemplo para uma queue)

Esta também pode dar jeito…

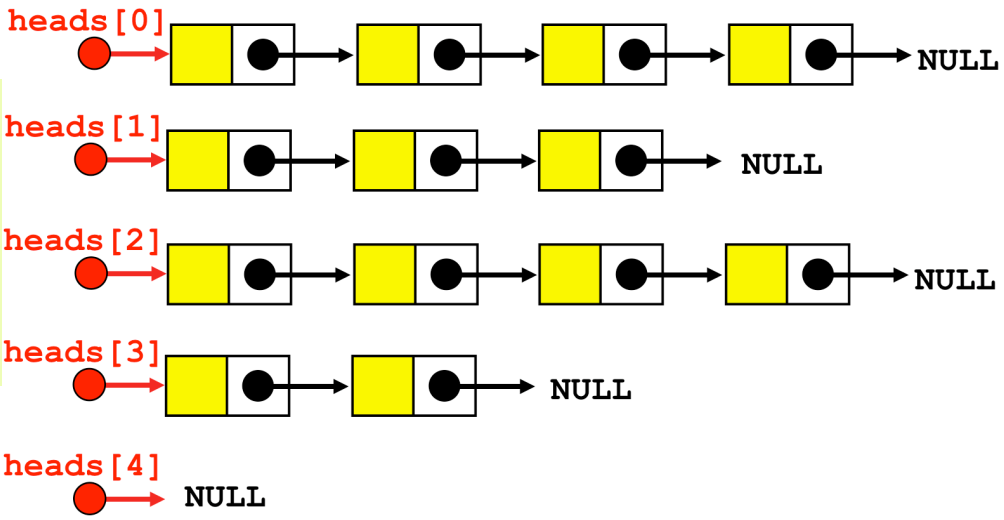
```
typedef struct queue{
   link head, tail;
   int size;
}*Queue;
```



Mais abstracção (exemplo para uma hashtable)

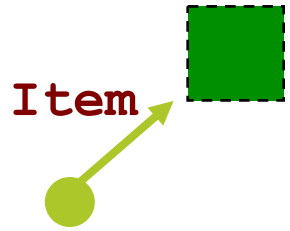
Esta também pode dar jeito…

```
typedef struct hashtable{
   link* heads;
   int M, size;
}*hashtable;
```



Funções úteis – Item's

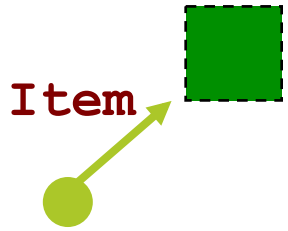
```
typedef struct produto{
   (…)
}*Item;
```




```
Item NewItem ( valores a guardar no novo item )
{
   /* cria um novo Item */
}
```

Funções úteis – Item's

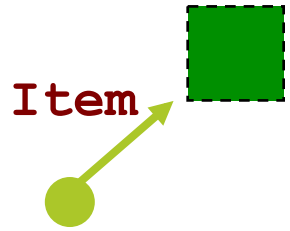
```
typedef struct produto{
   (…)
}*Item;
```




```
void showItem(Item x)
{
  /* escreve o conteúdo de um item – ver enunciado */
}
```

Funções úteis – Item's

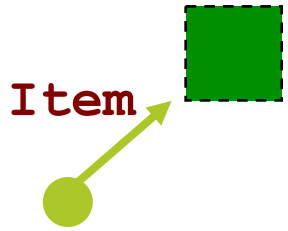
```
typedef struct produto{
   (…)
}*Item;
```




```
int less(Item a, Item b)
{
   /* retorna um 1 se a for menor que b dado um critério dado.
   Retorna 0 em todos os outros casos */
}
```

Funções úteis – Item's

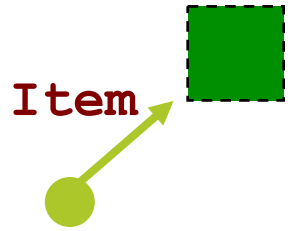
```
typedef struct produto{
   (…)
}*Item;
```




```
int cmpItem(Item a, Item b)
{
   /* retorna um valor < 0 se a<b, 0 se forem iguais e um valor
   >0 se b>a */
}
```

Funções úteis – Item's

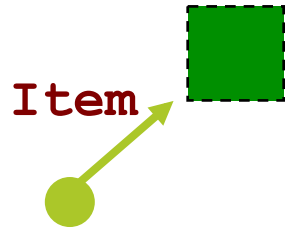
```
typedef struct hashtag{
   (…)
}*Item;
```




```
Item newItem(  conteudo a guardar no item  );
void showItem(Item a);
int cmpItem(Item a, Item b);
void sort(Item a[], int l, int r); /* se necessário */
```

Funções úteis – Item's

```
typedef struct hashtag{
   (…)
}*Item;
```




```
Item newItem(  conteudo a guardar no item  );
void showItem(Item a);
int cmpItem(Item a, Item b);
void sort(Item a[], int l, int r, int(*comp)(Item, Item) ); /*
   se necessário */
```

Ponteiros para funções:
Ver slides finais

Funções úteis – Estrutura de dados

```
Init?
Insert?
Search?
PrintSorted?
Free?
Remove?

GetMax?
Count?
```

Fugas de memória

Para utilizar o valgrind, comece por compilar o seu código com o gcc mas com a *flag* adicional -g :

```
$ gcc -g -Wall -o proj2 *.c
```

Esta flag adiciona informação para *debugging* que será posteriormente utilizada pelo *valgrind*. Depois disso, basta escrever

```
$ valgrind ./proj2 < teste01.in
```

Neste exemplo, estaria a usar o teste01.in como *input*.

Fugas de memória

Para utilizar o valgrind, comece por compilar o seu código com o gcc mas com a *flag* adicional -g :

```
$ gcc -g -Wall -o proj2 *.c
```

Esta flag adiciona informação para *debugging* que será posteriormente utilizada pelo *valgrind*. Depois disso, basta escrever

```
$ valgrind --tool=memcheck --leak-check=yes ./proj2 < teste01.in
```

Neste exemplo, estaria a usar o teste01.in como *input*.

Fugas de memória

Deverá estar particularmente atento/a a mensagens como:

- `Invalid read/write`: indicam está a tentar ler ou escrever fora da área de memória reservada por si.
- `Conditional jump or move depends on uninitialised value(s)`: utilização de variáveis não inicializadas dentro expressões condicionais.
- Quantidade de memória alocada e libertada na *heap*. Aqui fica um exemplo de output quando tudo corre bem:

```
HEAP SUMMARY:
in use at exit: 0 bytes in 0 blocks
total heap usage: 1,024,038 allocs, 1,024,038 frees, 28,682,096 bytes allocated
All heap blocks were freed -- no leaks are possible
```

Sugestões finais...

- Procure usar abstracções
- Idente e documente convenientemente o seu código.
- Organize o código da melhor forma, se possível em vários ficheiros.
- Teste o seu código com o *Valgrind* e/ou *gdb* e elimine eventuais fugas de memória.
- Teste o seu código à medida que o escreve.
- Submeta com antecedência.

Bons códigos!

Pequeno parêntesis: Ponteiros para Funções

- É possível ter ponteiros para funções
- O nome de uma função é um ponteiro para essa função

```
int soma(int a, int b) { return a+b; }

int main() {
  int (*ptr)(int, int);

  ptr = soma;

  printf("%d\n", (*ptr)(3,4));
  return 0;
}
```

Exemplo: algoritmos de ordenação

- É possível ter ponteiros para funções
- O nome de uma função é um ponteiro para essa função

```
void insertion(Item a[], int l, int r, int(*comp)(Item,
   Item) )
{
    int i,j;
    for (i = l+1; i <= r; i++) {
        Item v = a[i];
        j = i-1;
        while ( j>=l && comp(v, a[j])<0) {
            a[j+1] = a[j];
            j--;
        }
        a[j+1] = v;
    }
}
```

Exemplo: algoritmos de ordenação

- É possível ter ponteiros para funções
- O nome de uma função é um ponteiro para essa função

```
int cmpItem(Item a, Item b)
{
   /* retorna um valor < 0 se a<b, 0 se forem iguais e um valor
   >0 se b>a */
}
```

```
/* vamos ordenar um vector vec do indice 0 ao indice 234 */

insertion(vec, 0, 234, cmpItem);
```

Stdlib…

http://www.cplusplus.com/reference/cstdlib/qsort/

Se definir as comparações de uma forma geral (com void*) pode tentar usar funções de ordenação da **stdlib.h**....

```
int cmpItem(const void *a, const void* b)
{
   /* retorna um valor < 0 se a<b, 0 se forem iguais e um valor
   >0 se b>a */
   /* recebe ponteiros para aquilo que queremos organizar.
   Se tiver a ordenar inteiros função deverã receber endereços
   de inteiros */
   /* antes de utilizar o a e o b, tem de converter para aquilo
   que precisa. Exemplo para inteiros:
   int i1 = *((int*)a);
   int i2 = *((int*)b);
}
```

Stdlib…

http://www.cplusplus.com/reference/cstdlib/qsort/

Se definir as comparações de uma forma geral (com void*) pode tentar usar funções de ordenação da **stdlib.h**....

```
int cmpItem(const void *a, const void* b)
{
   /* retorna um valor < 0 se a<b, 0 se forem iguais e um valor
   >0 se b>a */
   /* recebe ponteiros para aquilo que queremos organizar.
   Se tiver a ordenar Item's, a função deverã receber Item* */
   /* antes de utilizar o a e o b, tem de converter para aquilo
   que precisa. Exemplo para item's:
   Item i1 = *((Item*)a);
   Item i2 = *((Item*)b);

}
```