

Listas Ligadas II

30/10/2023

Ficheiros com exemplos

- Está disponível no Moodle um **ficheiro ZIP** de suporte aos tópicos de hoje
- Implementação de tipos abstratos usando uma **lista ligada** como representação interna
- **Implementações incompletas**, que permitem trabalho autónomo de desenvolvimento e teste

Sumário

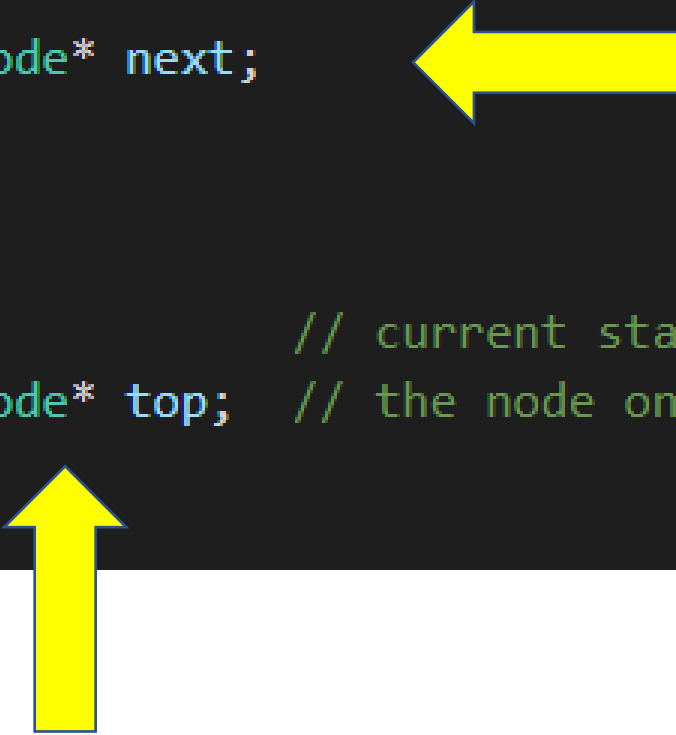
- Recap
- O TAD **LIST** – análise de algumas funções
- O TAD **SORTED LIST** – análise de algumas funções
- Apresentação do **1º TRABALHO**

Let's
RECAP

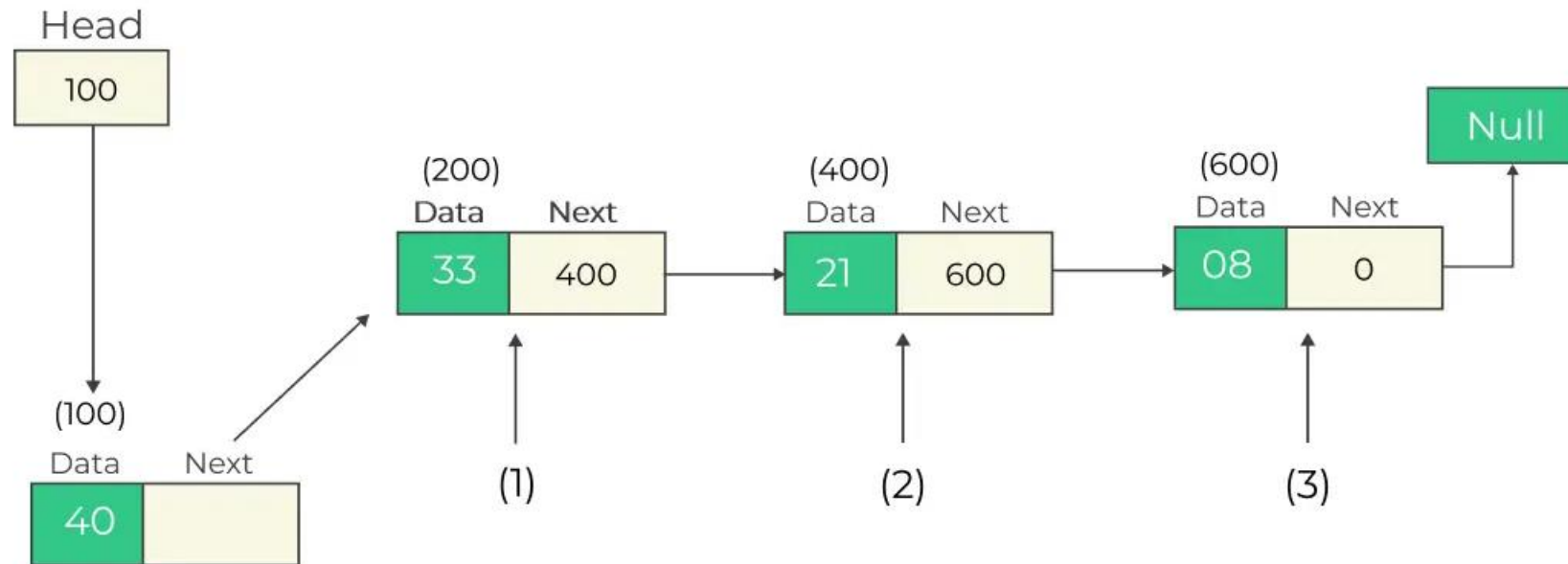
Recapitulação

O TAD **Stack** – Usando uma **lista ligada**

```
struct _PointersStackNode {  
    void* data;  
    struct _PointersStackNode* next;  
};  
  
struct _PointersStack {  
    int cur_size;           // current stack size  
    struct _PointersStackNode* top; // the node on the top of the stack  
};
```



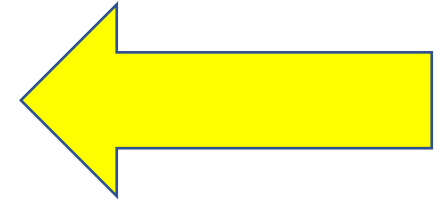
Stack usando uma **lista ligada** – **push(40)**



[prepinsta.com]

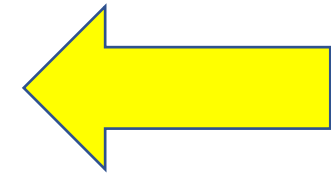
Push

```
void StackPush(Stack* s, void* p) {  
    assert(s != NULL);  
  
    struct _PointersStackNode* aux;  
    aux = (struct _PointersStackNode*)malloc(sizeof(*aux));  
    assert(aux != NULL);  
  
    aux->data = p;  
    aux->next = s->top;  
  
    s->top = aux;  
  
    s->cur_size++;  
}
```



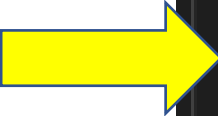
Pop

```
void* StackPop(Stack* s) {  
    assert(s != NULL && s->cur_size > 0);  
  
    struct _PointersStackNode* aux = s->top;  
    s->top = aux->next;  
    s->cur_size--;  
  
    void* p = aux->data;  
  
    free(aux);  
  
    return p;  
}
```



O TAD **Queue** – Usando uma **lista ligada**

```
struct _PointersQueueNode {  
    void* data;  
    struct _PointersQueueNode* next;  
};  
  
struct _PointersQueue {  
    int size; // current Queue size  
    struct _PointersQueueNode* head; // the head of the Queue  
    struct _PointersQueueNode* tail; // the tail of the Queue  
};
```



Queue usando uma **lista ligada**

Adding the elements into Queue



Removing the elements from Queue



Printing the Queue



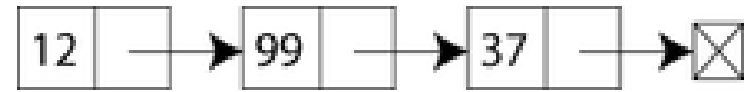
prepinsta.com]

Enqueue

```
void QueueEnqueue(Queue* q, void* p) {  
    assert(q != NULL);  
  
    struct _PointersQueueNode* aux;  
    aux = (struct _PointersQueueNode*)malloc(sizeof(*aux));  
    assert(aux != NULL);  
  
    aux->data = p;  
    aux->next = NULL;  
  
    q->size++;  
  
    if (q->size == 1) {  
        q->head = aux;  
        q->tail = aux;  
    } else {  
        q->tail->next = aux;  
        q->tail = aux;  
    }  
}
```

Dequeue

```
void* QueueDequeue(Queue* q) {  
    assert(q != NULL && q->size > 0);  
  
    struct _PointersQueueNode* aux = q->head;  
    void* p = aux->data;  
  
    q->size--;  
  
    if (q->size == 0) {  
        q->head = NULL;  
        q->tail = NULL;  
    } else {  
        q->head = aux->next;  
    }  
  
    free(aux);  
  
    return p;  
}
```



[Wikipedia]

O TAD **LISTA**

LISTA – Funcionalidades


- Conjunto de **elementos** do **mesmo tipo**
- Inserção / remoção / substituição / consulta em **qualquer posição**
- **insert() / remove() / replace() / get()**
- **size() / isEmpty() / isFull()**
- **init() / destroy() / clear()**

LISTA – Funcionalidades

- Associar um **índice implícito** a cada nó
 - O **primeiro nó** tem índice **ZERO**
- Associar um **iterador**
 - Ponteiro : **current**
 - Índice : **currentPos**
- **Movimentar** o iterador para o início / fim / índice

Nó da lista & Cabeçalho da lista

```
struct _PointersListNode {  
    void* data;  
    struct _PointersListNode* next;  
};  
  
struct _PointersList {  
    int size; // current List size  
    struct _PointersListNode* head; // the head of the List  
    struct _PointersListNode* tail; // the tail of the List  
    struct _PointersListNode* current; // the current node  
    int currentPos;  
};
```



ListCreate

```
List* ListCreate(void) {  
    List* l = (List*)malloc(sizeof(List));  
    assert(l != NULL);  
  
    l->size = 0;  
    l->head = NULL;  
    l->tail = NULL;  
    l->current = NULL;  
    l->currentPos = -1; // Default: before the head of the list  
    return l;  
}
```

Move – Casos particulares

```
int ListMove(List* l, int newPos) {  
    if (newPos < -1 || newPos > l->size) {  
        return -1;  
    } // failure  
  
    if (newPos == -1 || newPos == l->size) {  
        l->current = NULL;  
    } else if (newPos == 0) {  
        l->current = l->head;  
    } else if (newPos == l->size - 1) {  
        l->current = l->tail;  
    } else {
```

Move – Casos gerais

```
    } else {  
        if (l->currentPos == -1 || l->currentPos == l->size ||  
            newPos < l->currentPos) {  
            l->current = l->head;  
            l->currentPos = 0;  
        }  
  
        for (int i = l->currentPos; i < newPos; i++) {  
            l->current = l->current->next;  
        }  
    }  
    l->currentPos = newPos;  
  
    return 0; // success  
}
```


Move – As outras funções são mais simples

```
int ListMoveToNext(List* l) { return ListMove(l, l->currentPos + 1); }  
  
int ListMoveToPrevious(List* l) { return ListMove(l, l->currentPos - 1); }  
  
int ListMoveToHead(List* l) { return ListMove(l, 0); }  
  
int ListMoveToTail(List* l) { return ListMove(l, l->size - 1); }
```

Desenvolvimento – Assegurar os invariantes

```
void ListTestInvariants(const List* l) {  
    assert(l->size >= 0);  
    if (l->size == 0)  
        assert(l->head == NULL && l->tail == NULL);  
    else  
        assert(l->head != NULL && l->tail != NULL);  
    if (l->size == 1) assert(l->head == l->tail);  
    assert(-1 <= l->currentPos && l->currentPos <= l->size);  
    if (l->currentPos == -1 || l->currentPos == l->size)  
        assert(l->current == NULL);  
}
```

Desenvolvimento – Assegurar os invariantes



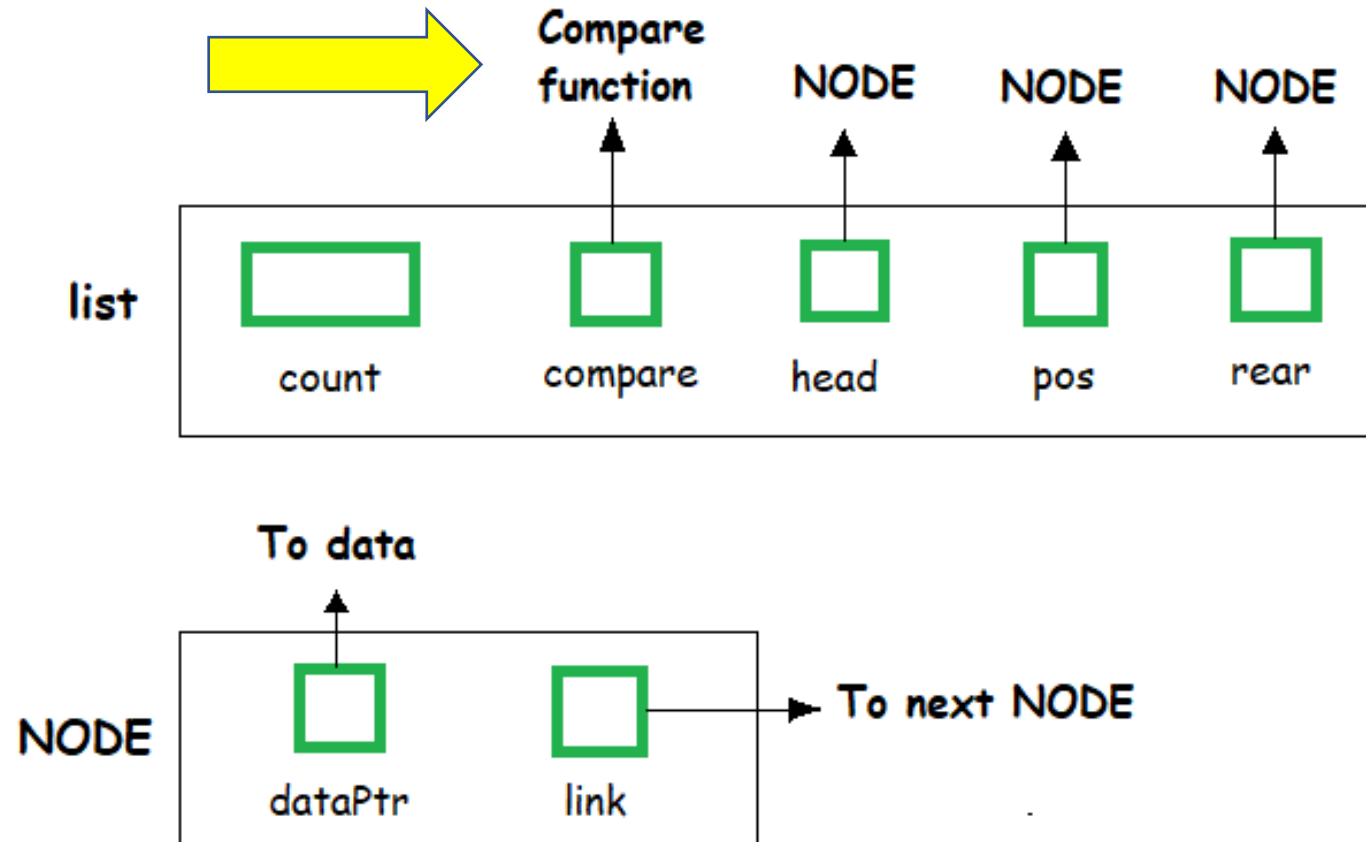
```
struct _ListNode* sn = l->head;
for (int i = 0; i < l->size; i++) {
    if (i == l->size - 1)
        assert(sn == l->tail && sn->next == NULL);
    else
        assert(sn->next != NULL);
    if (i == l->currentPos) assert(sn == l->current);
    sn = sn->next;
}
```

O TAD LISTA ORDENADA

O TAD LISTA ORDENADA

- Conjunto de elementos do mesmo tipo
- Armazenados em ordem de acordo com um critério
 - Registrar uma função comparadora
- A junção de um novo elemento à lista mantém a ordem !!
- A procura de um elemento fica facilitada !!
 - Porquê ?

O TAD LISTA ORDENADA





Função comparadora – Ponteiro para função



```
typedef struct _SortedList List;  
typedef int (*compFunc)(const void* p1, const void* p2);  
  
List* ListCreate(compFunc compF);  
  
void ListDestroy(List** p);  
  
void ListClear(List* l);
```

Exemplos – Inteiros e Datas

```
int comparator(const void* p1, const void* p2) {  
    int d = *(int*)p1 - *(int*)p2;   
    return (d > 0) - (d < 0);  
}
```

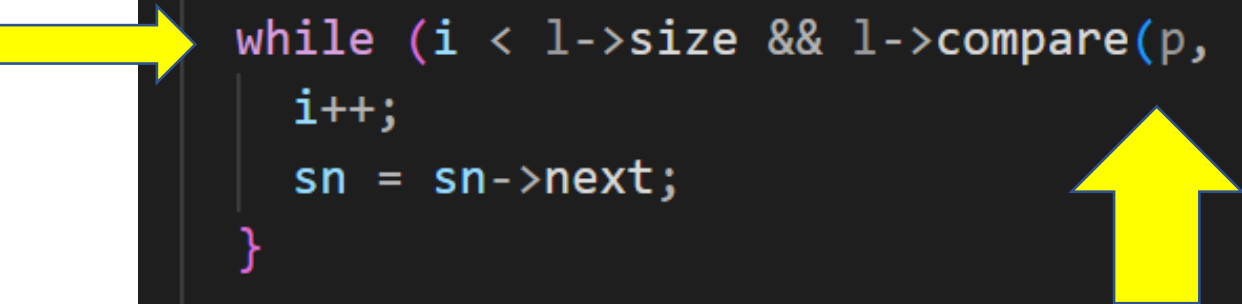
```
int comparatorForDates(const void* p1, const void* p2) {  
    return DateCompare((Date*)p1, (Date*)p2);   
}
```

ListSearch

```
// starting at the current node, search for the first node with a value of *p
// on failure the current node is not changed
//
int ListSearch(List* l, const void* p) {
    int i = (l->currentPos < 0) ? 0 : l->currentPos;

    struct _ListNode* sn = (l->currentPos < 0) ? l->head : l->current;


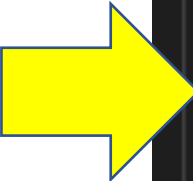
    while (i < l->size && l->compare(p, sn->item) > 0) {
        i++;
        sn = sn->next;
    }
}
```



ListSearch

```
→ if (i == l->size) {  
    |   return -1;  
    | } // failure  
  
→ if (l->compare(p, sn->item) < 0) {  
    |   return -1;  
    | } // failure  
  
→ l->current = sn;  
  l->currentPos = i;  
  
  return 0; // success  
}
```

ListInsert – Caso particular



```
int ListInsert(List* l, void* p) {  
    struct _ListNode* sn = (struct _ListNode*)malloc(sizeof(struct _ListNode));  
    assert(sn != NULL);  
    sn->item = p;  
    sn->next = NULL;  
  
    // Empty list  
    if (l->size == 0) {  
        l->head = l->tail = sn;  
        l->size = 1;  
        return 0;  
    }  
  
    // Search
```


ListInsert – Procurar na lista ordenada

```
// Search
```

```
int i = 0;
```

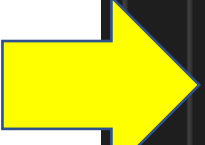
```
struct _ListNode* prev = NULL;
```

```
struct _ListNode* aux = l->head;
```



```
while (i < l->size && l->compare(p, aux->item) > 0) {
```

```
    i++;
```




```
    prev = aux;
```


```
    aux = aux->next;
```

```
}
```

ListInsert – Casos possíveis




```
if (i == l->size) { // Append at the tail
    l->tail->next = sn;
    l->tail = sn;
    l->size++;
    return 0;
}
```



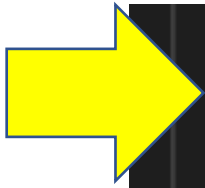
```
if (l->compare(p, aux->item) == 0) { // Already exists
    free(sn);
    return -1;
} // failure
```


ListInsert – Inserir no início



```
if (i == 0) { // Append at the head
    sn->next = l->head;
    l->head = sn;
    l->size++;
    if (l->currentPos >= 0) {
        l->currentPos++;
    }
    return 0;
}
```

ListInsert – Inserir na posição correta



```
sn->next = aux;  
prev->next = sn;  
l->size++;  
if (l->currentPos >= i) {  
    l->currentPos++;  
}  
  
return 0;  
}
```

Tarefa

- Analisar os ficheiros disponibilizados
- Identificar as **funções incompletas**
- **Implementar** essas funções
- **Testar** com novos exemplos de aplicação