# MAT A40 - Estrutura de Dados e Algoritmos I [1]

Dr. George Lima
Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade Federal da Bahia

[1] Este módulo foi baseado em conteúdos presentes em "A. V. Aho, J. D. Ullman, *Foundations of Computer Science* (C edition). Computer Science Press (W.H. Freeman), 1995." e "Paulo Feofiloff. *Algoritmos em linguagem C*. Campus/Elsevier, 2009.".

# Módulo 3

## Listas e algumas de suas implementações em C

# O tipo Lista

## Definição

Uma lista é uma sequência finita de zero ou mais elementos, não necessariamente distintos. Por exemplo, $(a_1, a_2, \ldots, a_n)$ é uma lista com $n$ elementos. Se todos os seus elementos são do mesmo tipo $T$, a lista é dita do tipo $T$.

# O tipo Lista

Possível conjunto de operações sobre listas:

- Inserção: Se $L = (a_1, a_2, \ldots, a_n)$, `insert`$(x, L)$ insere $x$ em $L$. O elemento $x$ pode ser inserido em qualquer posição na lista. Restrições sobre a posição a ser inserida podem estar associadas à aplicação.

# O tipo Lista

Possível conjunto de operações sobre listas:

- Remoção: Se $L = (a_1, a_2, \ldots, x, \ldots, a_n)$, então `delete`$(x, L)$ remove uma ocorrência de $x$, fazendo $L = (a_1, a_2, \ldots, a_n)$. Remoções de todas as ocorrências de $x$ podem ser especificadas.

# O tipo Lista

Possível conjunto de operações sobre listas:

- Verificação de pertinência: operação `lookup`$(x, L)$ retorna verdade se $x \in L$ e falso caso contrário.

# O tipo Lista

Possível conjunto de operações sobre listas:

- Concatenação: concatenar duas listas $L = (a_1, a_2, \ldots, a_n)$ e $M = (b_1, b_2, \ldots, b_n)$ significa produzir uma lista $N = \texttt{concat}(L, M) = (a_1, a_2, \ldots, a_n, b_1, b_2, \ldots, b_n)$. Concatenar uma lista $L$ com uma lista vazia ou vice-versa, produz $L$.

# O tipo Lista

Possível conjunto de operações sobre listas:

- Primeiro e último elemento: `first`($L$) e `last`($L$) retornam o primeiro e o último elemento de uma lista não vazia $L$, respectivamente. Se $L$ estiver vazia, um erro é produzido.

# O tipo Lista

Possível conjunto de operações sobre listas:

- Consulta posição: `retrieve`$(i, L)$ retorna o elemento na posição $i$ da lista ou um erro caso o tamanho de $L$ seja menor que $i$. Outra possível implementação desta operação retorna o elemento cujo conteúdo é igual a $i$. Neste último caso, $i$ é a chave de identificação do elemento na lista, cujos elementos possuem outros atributos além da chave. Em caso de não se encontrar $i$ na lista, uma sinalização é retornada pela operação.

# O tipo Lista

Possível conjunto de operações sobre listas:

- Tamanho e existência de elementos: `length`($L$) retorna o número de elementos da lista enquanto `length`($L$) retorna verdade ou falso caso a lista $L$ esteja ou não vazia, respectivamente.

# Estrutura de dados: Lista com encadeamento simples

## Definição: Lista (simplesmente) encadeada

Uma lista encadeada é formada de uma sequência de células, cada uma das quais representam um elemento do modelo lista. Cada célula associada ao elemento $a_i$ contém ainda o endereço do próximo elemento, $a_{i+1}$. Para a última célula, tal endereço do próximo elemento é nulo.

- Listas são comumente construídas através da estrutura de dados conhecida como lista encadeada.
- Usar listas encadeadas (implementação) para construir listas (modelo abstrato) é apenas uma das possibilidades disponíveis.

# TAD: lista com encadeamento simples em C

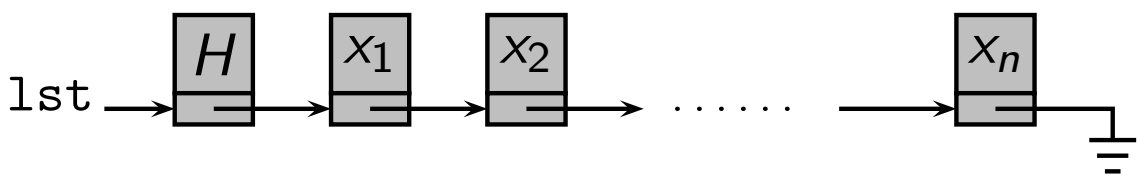
```c
typedef struct cell cell_t;

struct cell {
  int element;
  cell_t *next;
};
```

- Listas são comumente construídas através da estrutura de dados conhecida como lista encadeada.
- Usar listas encadeada (implementação) para construir listas (modelo abstrato) é apenas uma das possibilidades disponíveis.

# Lista com encadeamento simples

Lista com cabeça: `cell_t *lst`:

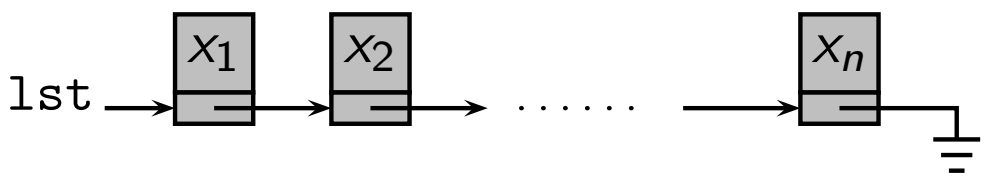
- `lst` aponta para o elemento cabeça da lista
- Elemento cabeça contém o endereço do primeiro elemento da lista
- Lista vazia contém ao menos um elemento



# Lista com encadeamento simples

Lista sem cabeça: `cell_t *lst`:

- `lst` aponta para o primeiro elemento da lista
- Lista vazia não contém elementos(`lst` aponta para `NULL`)



# Criando uma lista (com cabeça)

Lista vazia contém uma célula (cabeça da lista):

```
1  cell_t *newCell() {
2
3   cell_t *p = malloc( sizeof (cell_t));
4   if (p) {
5    p -> next = NULL;
6    p -> element = -1;
7    return p;
8   } else return(NULL);
9  }
10
11 int main() {
12    cell_t *list = newCell(); // cria lista vazia
13 }
```

- A lista criada contém um elemento, mesmo estando vazia. Por que?
- Quais as consequências se o código da linha 10 fosse escrito como `cell_t *list = NULL`?

# Operação de inserção

```
1 void insert(int x, cell_t *lhead) {
2
3   cell_t *p = newCell(); // Lembrar de testar se houve
4                          // problemas com alocacao de espaco
5
6   p     -> next = lhead -> next;
7   lhead -> next = p;
8   p     -> element = x;
9 }
```

- Modifique o código para inserir mantendo a lista ordenada.
- Modifique o código para inserir na última posição.
- Qual a complexidade de tempo de execução associada a cada uma das três implementações?

# Operação de busca

```
1 cell_t* search(int x, cell_t *lhead) {
2
3   cell_t *p = lhead; // Falta testar alocacao
4   while(p->next) {
5     if (p->next->element == x) return p;
6     else p = p->next;
7   }
8   return p->next;
9 }
```

- Qual a complexidade da busca?
- Qual o elemento retornado na linha 5? Por que?
- Por que o uso de outro ponteiro, declarado localmente, para percorrer a lista?
- Como seria uma versão recursiva desta função?

# Operação de remoção

Busca e remove elemento $x$:

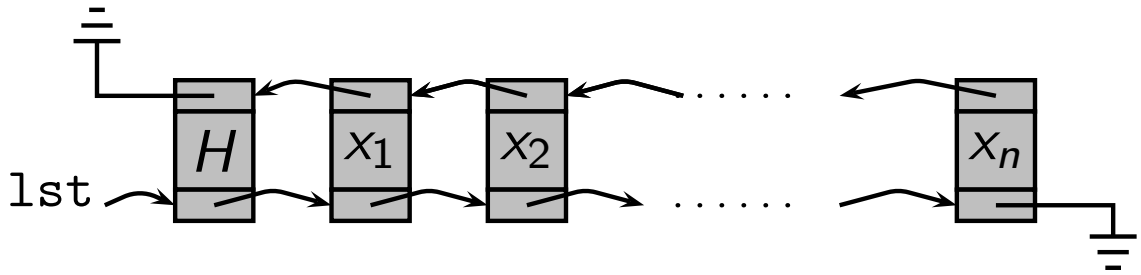
```
1  cell_t *delete(int x, cell_t *lhead) {
2
3    cell_t *p = search(x,lhead);
4
5    if(p) {
6      cell_t *q = p -> next; // p -> next != 0?
7      p -> next = q -> next; // q -> next != 0?
8      free (q);
9    }
10   return (lhead);
11 }
```

# Exercício de fixação – Lista com encadeamento simples

- Considere que a lista encadeada, além do elemento cabeça, possui um ponteiro para o último elemento, de forma que seja possível ter acesso ao último elemento de forma rápida. Como poderia o tipo `cell_t` ser redefinido para atender esta característica?
- Redefina as operações sobre lista encadeada vistas até aqui de forma a considerar a modificação acima.
- Defina as funções para as operações `first(cell_t *l)` e `last(cell_t *l)`, que devolvem o primeiro e último elemento da lista `l`, respectivamente.
- Defina a função `cell_t *concat(cell_t * l1,cell_t * l2)` que devolve a lista resultante da concatenação de `l1` com `l2`.

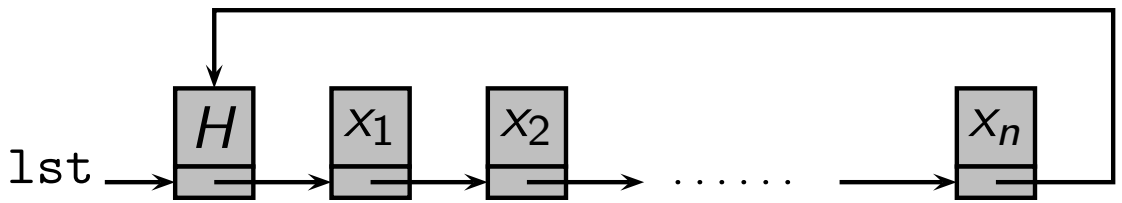
# Outras implementações de listas

- Lista com encadeamento duplo (útil onde se deseja mover em ambas as direções).




  - Pode-se caminhar em ambas as direções na lista (maior flexibilidade), mas gasta-se mais espaço

- Lista circular (útil em selecionar elementos, um após o outro, repetidamente)




- Outras possibilidades? Combinar: circular, linear, duplamente encadeada, múltiplas listas, com/sem nó cabeça.