Listas Lineares estáticas

Wanderley de Souza Alencar
adaptado por Raphael Guedes

INF
INSTITUTO DE INFORMÁTICA

UFG
UNIVERSIDADE FEDERAL DE GOIÁS

October 3, 2024

Problematização...

Há problemas que, para sua solução, exigem que se estabeleça uma relação linear de ordem entre os elementos de um aglomerado de dados:

$$(x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots \rightarrow x_n)$$

onde x_1 é o primeiro elemento, x_2 é o segundo, x_3 é o terceiro e assim, sucessivamente, até x_n que é o último elemento, com $n \in \mathbb{N}^*$.

Problematização...

Para que se possa indicar esta relação linear de ordem, é necessário, durante o desenvolvimento de *software*, que se crie uma (representação de) estrutura de dados adequada.

Qual seria esta estrutura de dados?

Problematização...

Esta estrutura de dados é chamada de lista linear.

Problematização...

Uma lista linear pode ser abstratamente representada por:

$$\mathcal{L} = (x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow \cdots \rightarrow x_{n-1} \rightarrow x_n)$$

onde cada elemento da lista está imediatamente *antes* ou imediatamente *após* outro elemento, exceto os elementos extremos – primeiro e último – que não possuem predecessor e sucessor, respectivamente.

Problematização...

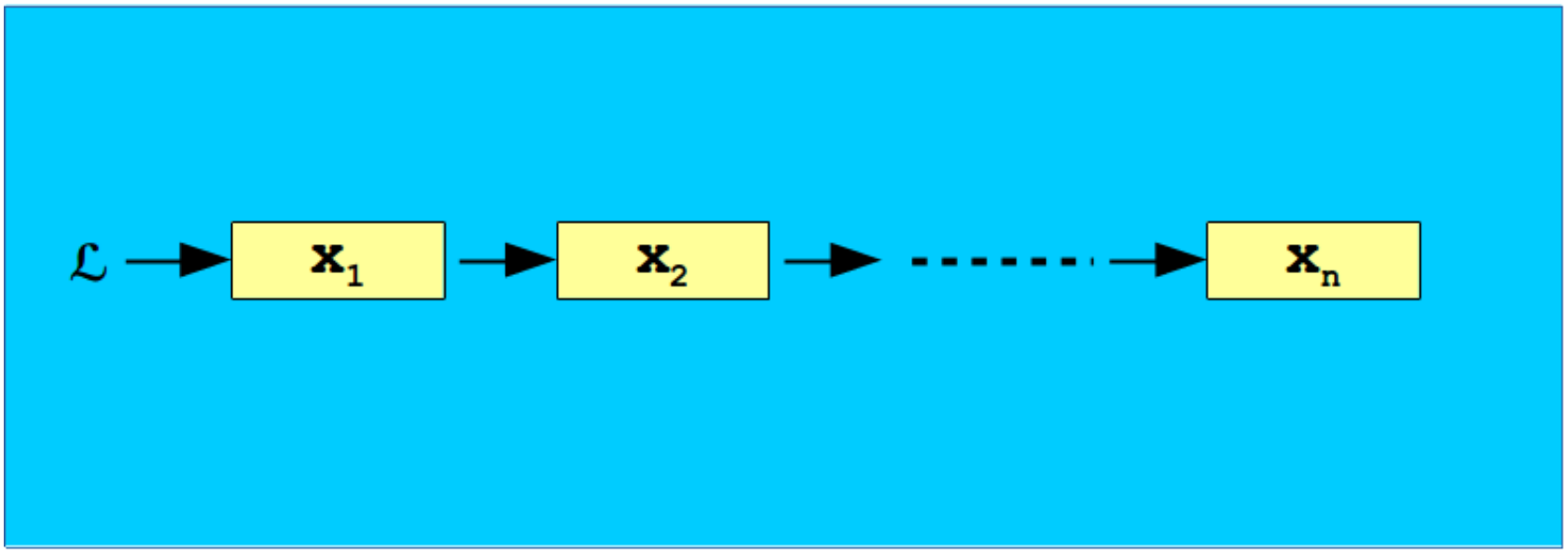
A partir deste conhecimento podemos conceber, e representar, a ideia de *listas lineares* de:

- compras (de supermercado);
- compromissos (de uma semana);
- *e-mails* (a serem lidos e respondidos);
- objetos (dispostos em determinada ordem física);

Problematização...

Vamos formalizar este conceito...

Conceito

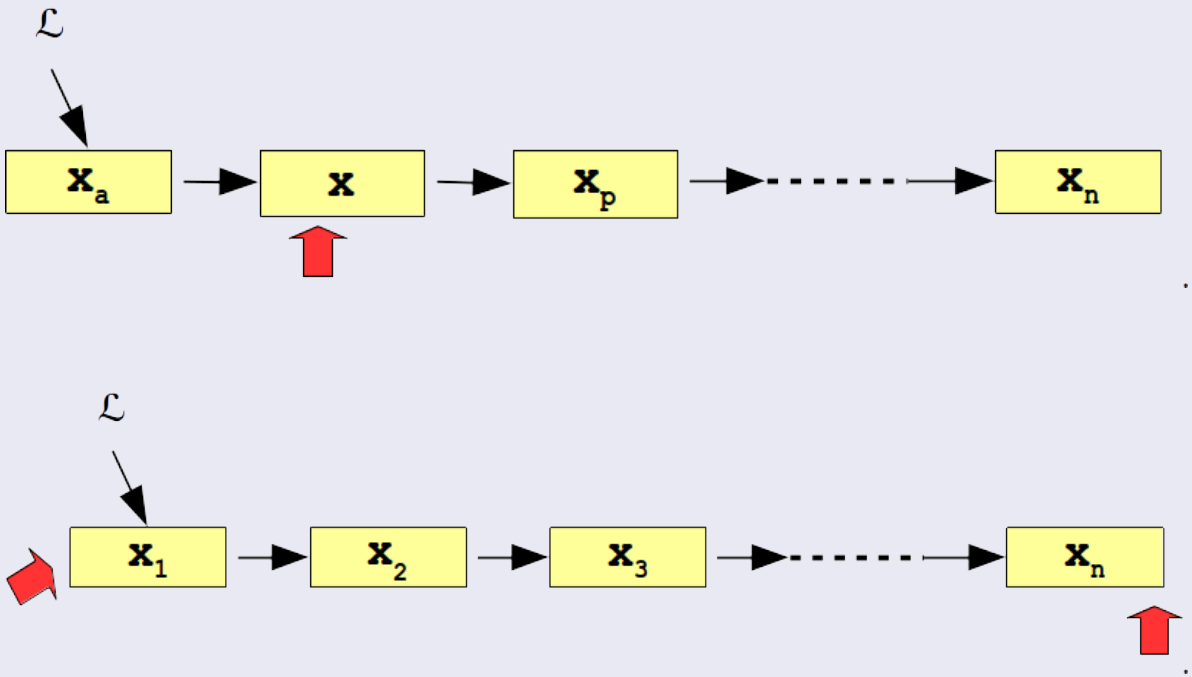


Lista Linear

Uma lista linear é uma estrutura de dados formada por uma sequência finita de células – elementos, componentes, nodos ou nós – organizada de tal maneira que reflita uma relação linear de ordem entre elas.

A relação linear de ordem define que uma célula x tenha uma, e somente uma, célula x_a que lhe seja imediatamente anterior e outra, também única, x_p que lhe seja imediatamente posterior, à exceção da primeira e da última células que não possuem célula predecessora e sucessora, respectivamente.

Lista Linear



Lista Linear

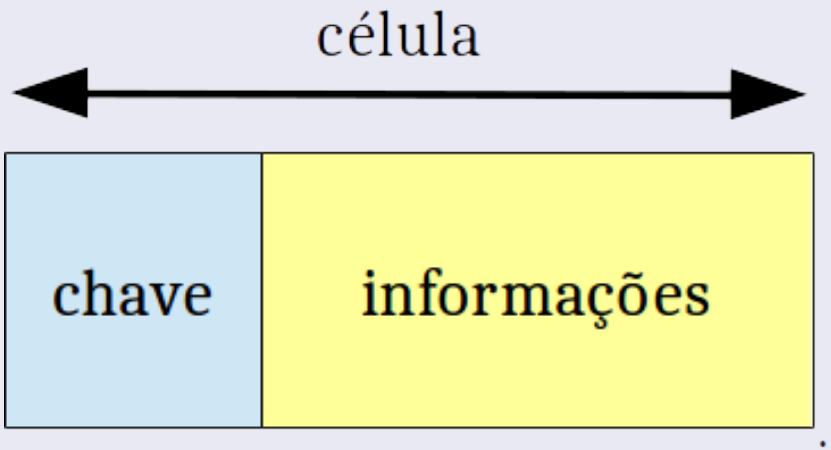
Podemos conceitualmente ter uma *lista vazia*, ou seja, uma lista sem qualquer célula, mas *pronta* para passar a tê-las:

$\mathcal{L} = \emptyset$

Lista Linear

Cada uma das células da lista linear é uma estrutura que contém:

- um membro especial, chamado de chave primária, que identifica de maneira unívoca aquela célula na lista linear;
- um conjunto de membros adicionais, cada um deles representando uma informação que se deseja armazenar.

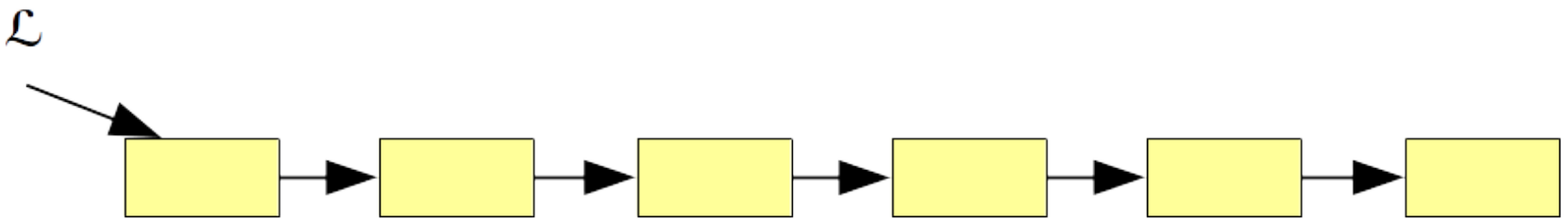


Lista Linear

Se uma lista linear armazena dados, surge, naturalmente, a seguinte indagação:

Que operações se pode realizar numa lista linear?

Operações



Fundamentais

Operações na Lista

Habitualmente as operações fundamentais sobre listas lineares incluem:

- criar uma lista vazia;
- determinar a quantidade de *células* presentes na lista;
- consultar a célula que possui determinado valor para a `chave`;
- consultar a célula que está na `k-ésima` posição;
- inserir uma célula antes (ou depois) de determinada célula;
- inserir uma célula no ínicio ou no final da lista;

Operações na Lista

Habitualmente as operações fundamentais sobre listas lineares incluem:

- remover uma célula que possui certa `chave`;
- remover uma célula que está na `k-ésima` posição;
- ordenar, sob determinado critério, as *células* da lista;
- concatenar duas listas;
- inverter a ordem das células de uma lista;
- comparar duas listas para verificar se são, ou não, iguais;
- ...

Operações na Lista

O conjunto de operações a ser *efetivamente* implementado dependerá dos requisitos estabelecidos pela aplicação sendo construída e, portanto, poderá ser bastante diferente de uma aplicação para outra.

Operações Fundamentais

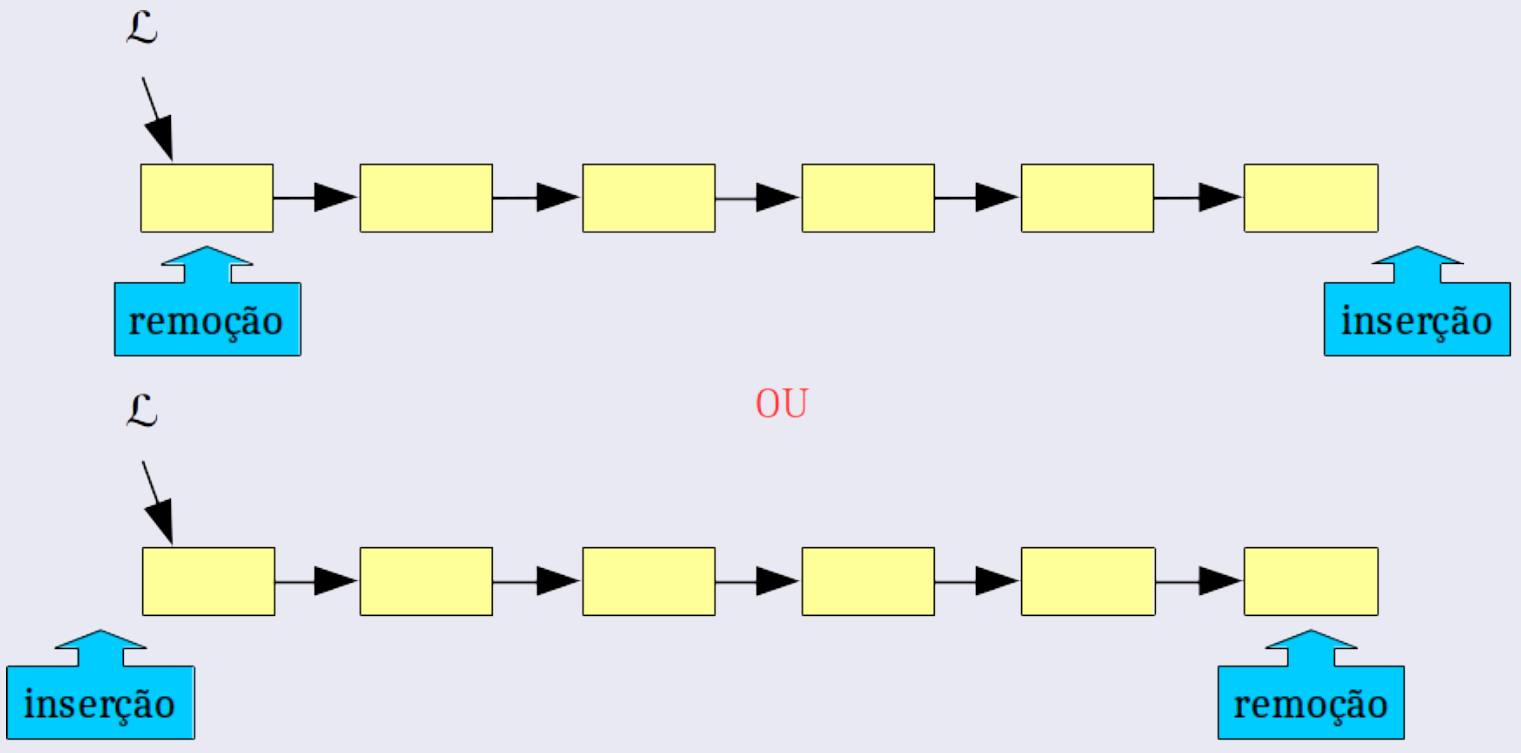
...

Aplicando-se algumas *restrições* às operações permitidas surgem variações de listas lineares:

- Fila;
- Pilha.

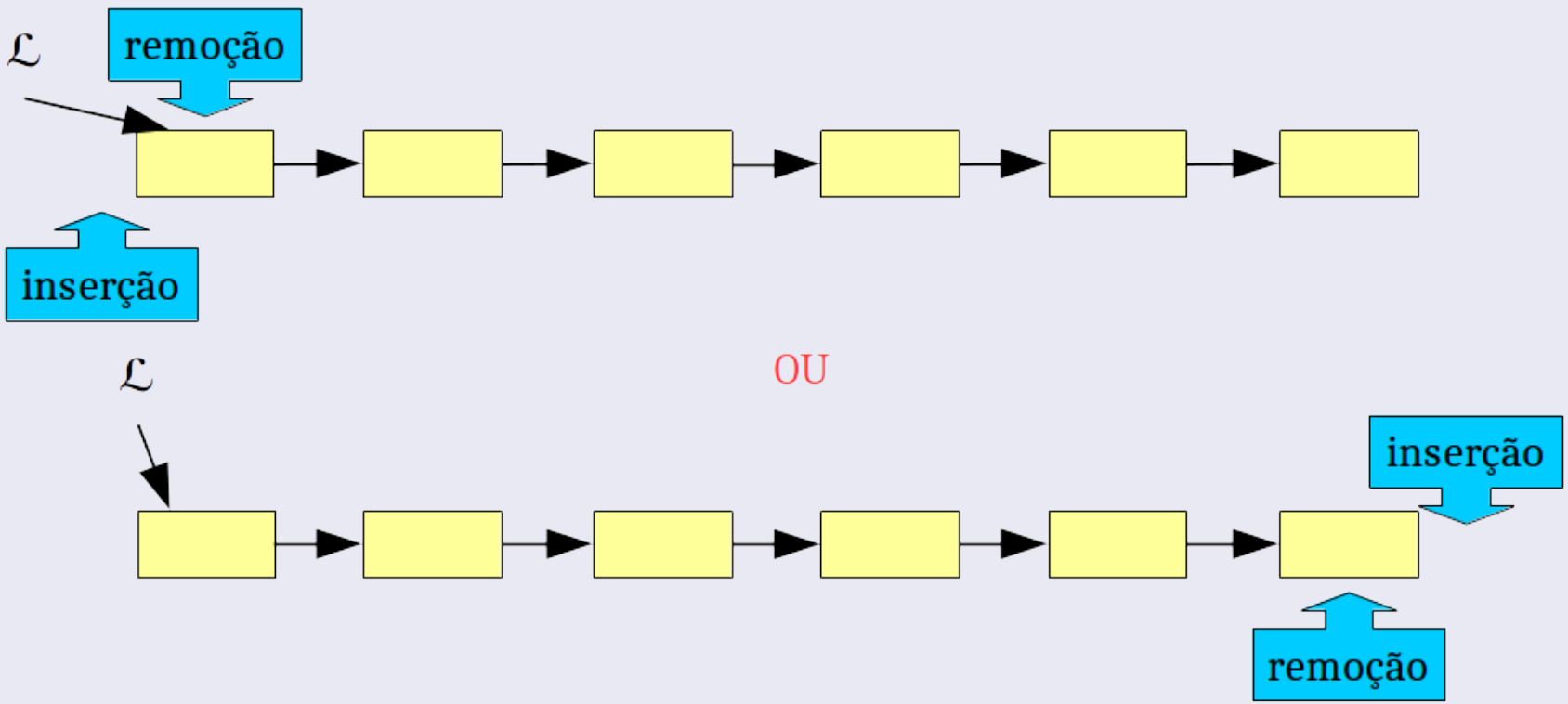
Fila

Numa *fila* as inserções são permitidas numa extremidade e as remoções na outra, resultando num critério de inserção/remoção denominado "*First In, First Out*" (FIFO).



Pilha

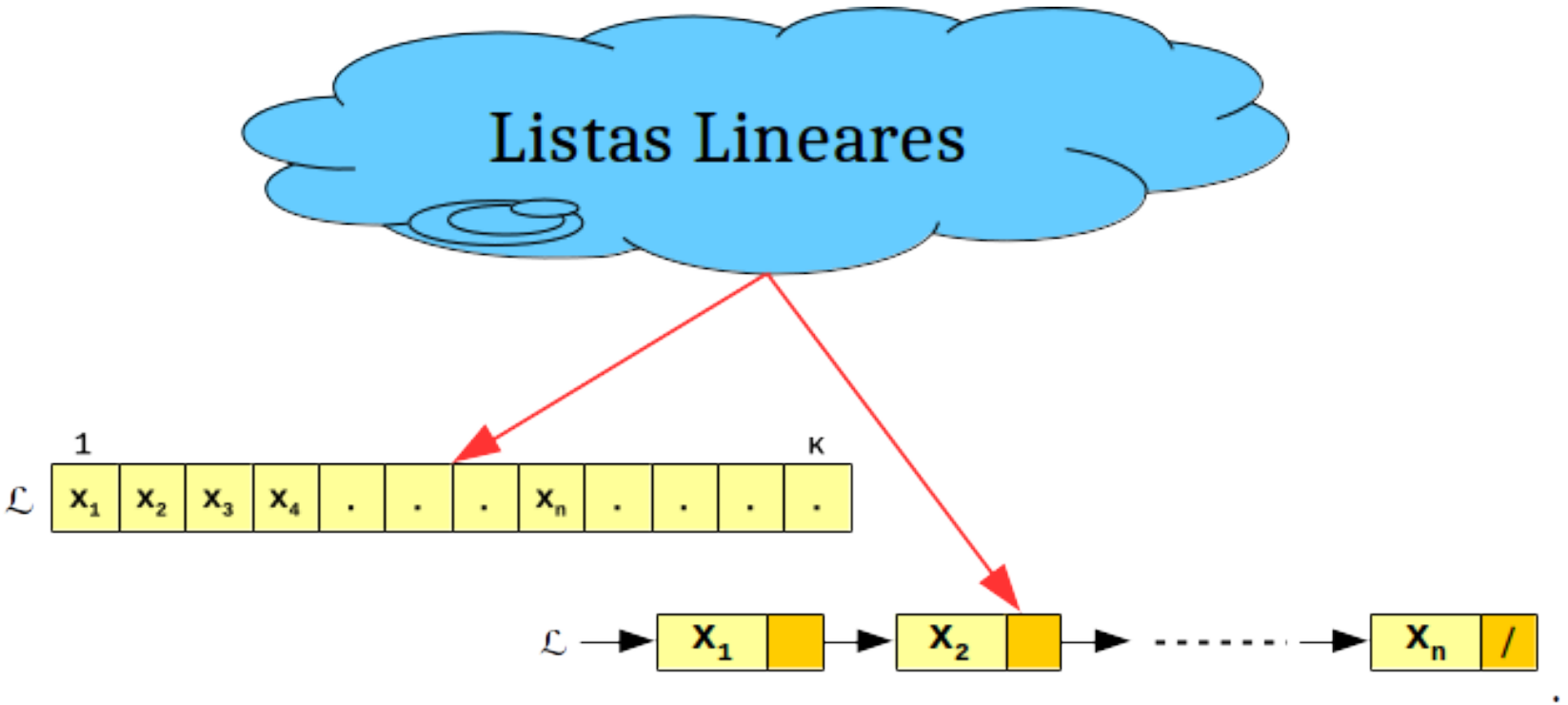
Numa *pilha* as inserções e remoções são permitidas numa única extremidade, resultando num critério de inserção/remoção denominado "*Last In, First Out*" (LIFO).



...

Estas variações serão estudadas futuramente.

Representações



Representações

Como *representar computacionalmente* uma lista linear?

Alocação de memória: dinâmica x estática

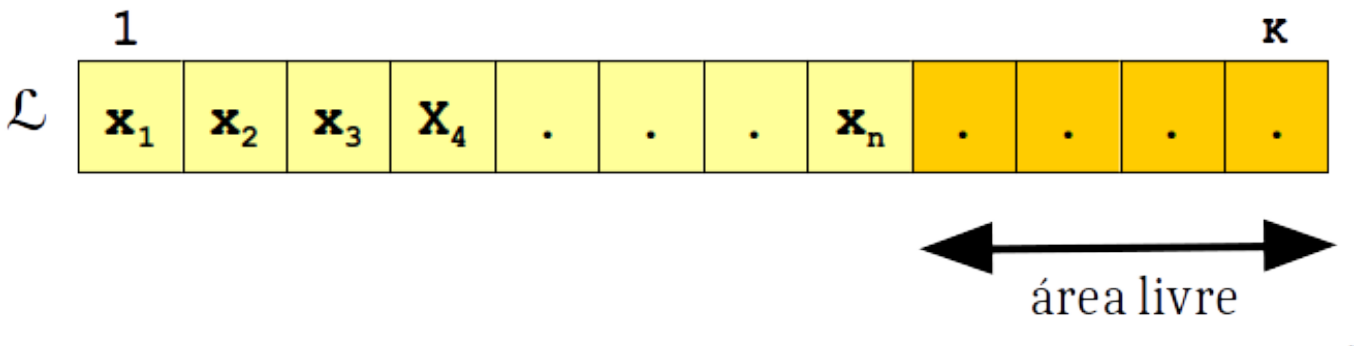
Para implementação computacional de uma lista linear há, essencialmente, duas representações:

1. por contiguidade (alocação estática de memória);
2. por encadeamento (alocação dinâmica de memória).

Alocação de memória: estática

Na representação por contiguidade (alocação estática):

- a área de memória é reservada em *tempo de compilação* do programa-fonte, ...
- ...assim é necessário estabelecer um *tamanho máximo* para a lista;
- as células são armazenadas *contiguamente* na memória principal, ...
- ...o que permite uso de *índices* para a localização delas.



Alocação de memória: estática

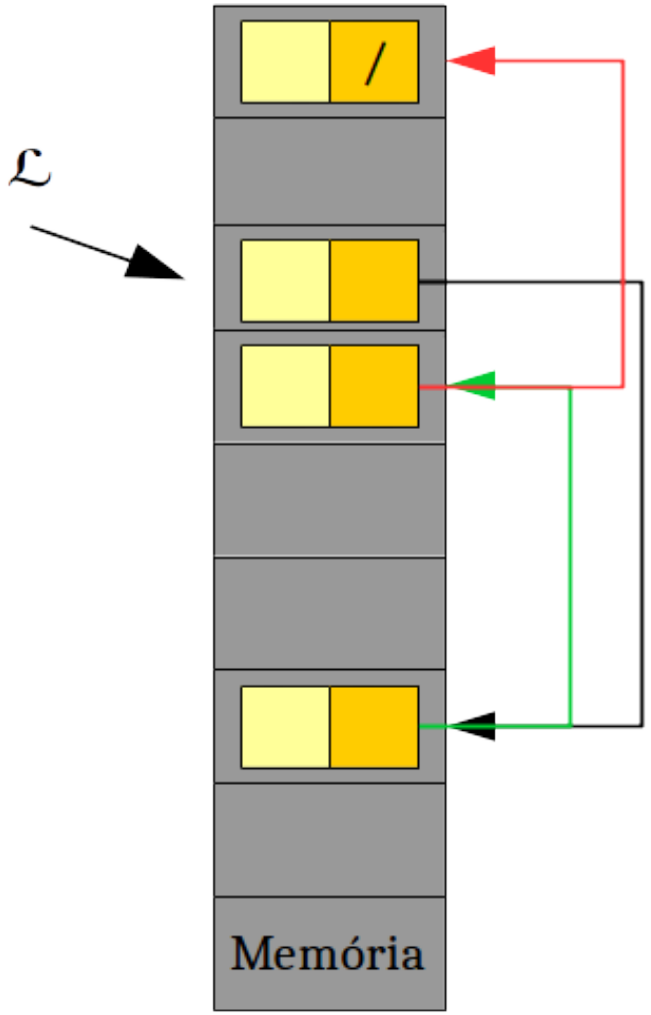
A memória total utilizada pela lista linear é constante, independentemente do número de células que a lista linear contenha, e corresponde à memória necessária para armazenar *tamanho máximo* células daquela lista.

Alocação de memória: dinâmica

Na representação por encadeamento (alocação dinâmica):

- a área de memória é reservada em *tempo de execução* do programa, ...
- ...o que torna o tamanho da estrutura (teoricamente) ilimitado;
- não é possível prever como as células são armazenadas na memória principal, ...
- ...o que impede uso de índices para a localização delas.

Representações

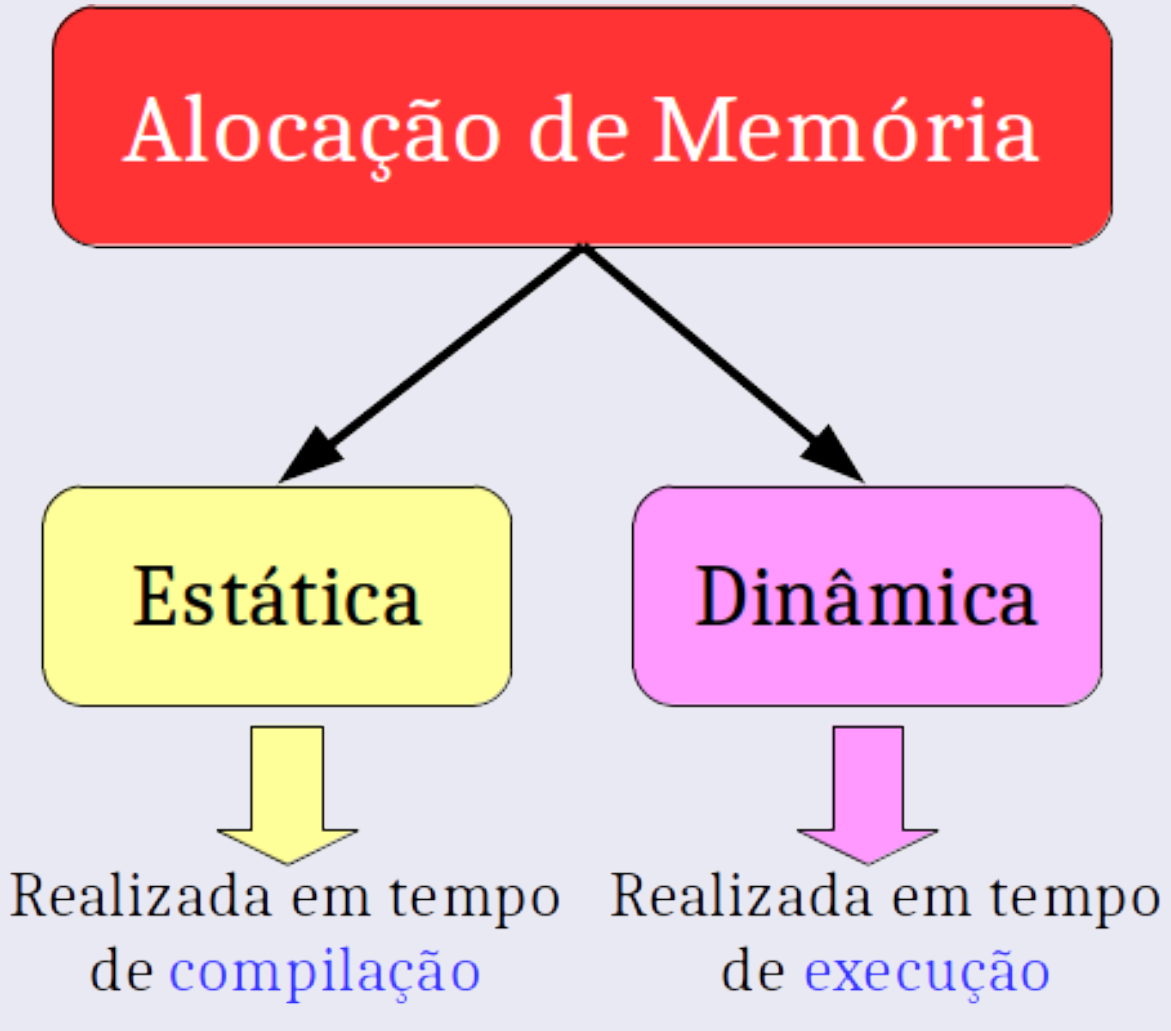


Alocação de memória: dinâmica

A memória total utilizada pela lista varia no transcorrer da execução do programa de acordo com o número de células nela.

Quando não mais necessitamos de uma célula, ela pode ser desalocada, ou seja, ter sua memória devolvida para uso pelo sistema computacional.

Alocação de memória: dinâmica x estática



Alocação de memória: dinâmica x estática

A escolha da maneira mais conveniente a ser utilizada deve ser balizada por fatores como:

1. quais os tamanhos, mínimo e máximo, esperados para a lista?
2. qual é a relação entre o tamanho máximo esperado para a lista e a quantidade de memória disponível no sistema computacional?
3. quais as operações serão realizadas sobre ela?
4. qual a frequência de realização de cada operação?

Alocação de memória: dinâmica x estática

Vamos, agora, detalhar a implementação pelas representações possíveis:

1. por contiguidade (alocação estática);
2. por encadeamento (alocação dinâmica).

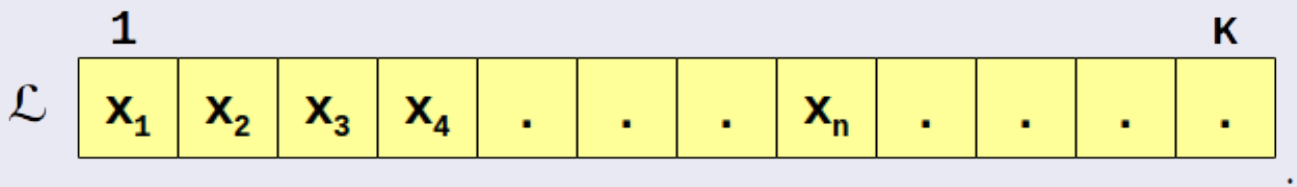
Lista linear por contiguidade

Dada uma lista linear $\mathcal{L}_1 = (x_1, x_2, \ldots, x_n)$, com $n \in \mathbb{N}^*$, utiliza-se de um vetor associado a duas variáveis de controle, para representá-la:

- `TamanhoLista`: contém o número de células presentes na lista naquele instante;
- `TamanhoMaximoLista`: contém o número máximo de células que a lista poderá conter, correspondendo ao tamanho do vetor que foi definido para representá-la, digamos, $\kappa \in \mathbb{N}^*$.

Lista linear por contiguidade

Vetor, de tamanho κ, representando uma lista linear $\mathcal{L}$:



Lista linear por contiguidade

Note que:

- a lista linear $\mathcal{L}$ ocupa as posições de 1 a n do vetor;
- as posições de $(n+1)$ a κ correspondem à área livre do vetor, ou seja, área destinada ao crescimento de $\mathcal{L}$;
- a lista linear está limitada ao tamanho do vetor, que é κ.

Declaração das estruturas para lista, por contiguidade:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  #define SUCESSO 1 // constantes-base para
5  #define FALHA -1 // para controle de codigo-fonte
6  #define CHAVE_INVALIDA 0 //
7  #define TAMANHO_MAXIMO_LISTA 100 // e tamanho maximo da lista
8
9  typedef struct {
10       unsigned int chave;
11       unsigned int dado;
12 } Celula;
13
14 typedef struct {
15     Celula celulas [TAMANHO_MAXIMO_LISTA];
16     unsigned int tamanho;
17 } ListaLinear;
```

Criar uma lista inicialmente vazia, por contiguidade:

```
1  int criarListaVazia(ListaLinear * lista) {
2      lista->tamanho = 0;
3      return (SUCESSO);
4  }
5  //
6  // Ou com inicializacao das celulas da lista com chaves invalidas
7  //
8  int criarListaVazia(ListaLinear * lista) {
9      int i;
10
11     for (i = 0; (i < TAMANHO_MAXIMO_LISTA); i++) {
12         lista->celulas[i].chave = CHAVE_INVALIDA;
13     }
14     lista->tamanho = 0;
15     return (SUCESSO);
16 }
```

Criar uma lista com uma única célula, por contiguidade:

```
1  int criarListaChave(ListaLinear * lista, Celula celula) {
2      lista->celulas[0] = celula;
3      lista->tamanho = 1;
4      return (SUCESSO);
5  }
6  //
7  // ou inicializando as demais celulas
8  //
9  int criarListaChave(ListaLinear * lista, Celula celula) {
10     int i;
11     for (i = 0; (i < TAMANHO_MAXIMO_LISTA); i++) {
12         lista->celulas[i].chave = CHAVE_INVALIDA;
13     }
14     lista->celulas[0] = celula;
15     lista->tamanho = 1;
16     return (SUCESSO);
17 }
```

Determinar o tamanho da lista, por contiguidade:

```
1 int tamanhoLista(ListaLinear lista) {
2     if (lista.tamanho >= 0) {
3         return(lista.tamanho);
4     }
5     else {
6         return(FALHA);
7     }
8 }
```

Mostrar uma determinada célula da lista, por contiguidade:

```
1 void mostrarCelula(Celula celula) {
2     printf("Chave.....: %u\n", celula.chave);
3     printf("Dado.....: %u\n", celula.dado);
4 }
```

Mostrar toda a lista, por contiguidade:

```
1  void mostrarLista(ListaLinear lista) {
2      int i;
3
4      if (lista.tamanho == 0) {
5          printf("Atencao: a lista esta vazia.\n");
6      }
7      else {
8          printf("A lista linear possui %u elementos.\n\n", lista.tamanho);
9          for (i = 0; (i < lista.tamanho); i++) {
10             printf("Elemento n.: %u\n", (i+1));
11             mostrarCelula(lista.celulas[i]);
12         }
13     }
14 }
```

Consultar célula baseado na posição dela, por contiguidade:

```
1  //
2  // Retorna a celula que esta na posicao desejada, ou
3  // CHAVE_INVALIDA se a posicao nao existe.
4  //
5  Celula consultaListaPosicao(ListaLinear lista, unsigned int intPosicao
       ) {
6
7      Celula celulaResultado;
8
9      if ((intPosicao > 0) && (intPosicao <= lista.tamanho)) {
10         celulaResultado = lista.celulas[intPosicao - 1];
11     }
12     else {
13         celulaResultado.chave = CHAVE_INVALIDA;
14     }
15     return(celulaResultado);
16 }
```

Consultar célula baseado na chave dela, por contiguidade:

```
1  //
2  // Retorna a celula com a chave desejada, ou
3  // CHAVE_INVALIDA se a chave nao existe na lista
4  //
5  Celula consultaListaChave(ListaLinear lista, Celula celula) {
6      unsigned int i;
7
8      for (i = 0; (i < lista.tamanho); i++) {
9          if (lista.celulas[i].chave == celula.chave) {
10             return(lista.celulas[i]);
11         }
12     }
13     celula.chave = CHAVE_INVALIDA;
14     return(celula);
15 }
```

Inserir célula no início da lista, por contiguidade:

```
1  int insInicio (ListaLinear * lista, Celula celula) {
2      unsigned int i;
3
4      if (lista->tamanho == TAMANHO_MAXIMO_LISTA) {
5          return(FALHA); // a lista ja esta cheia: overflow
6      }
7      else {
8          for (i = lista->tamanho; (i > 0); i--) {
9              lista->celulas[i] = lista->celulas[i-1];
10         }
11         lista->celulas[0] = celula; // insercao no inicio da lista
12         lista->tamanho++;
13         return(SUCESSO);
14     }
15 }
```

Inserir célula no final da lista, por contiguidade:

```
1  int insFinal (ListaLinear * lista, Celula celula) {
2      unsigned int i;
3      Celula auxiliar;
4
5      if (lista->tamanho == TAMANHO_MAXIMO_LISTA) {
6          return(FALHA); // a lista esta cheia: overflow
7      }
8      else {
9          lista->celulas[lista->tamanho] = celula; // insercao no final da lista
10         lista->tamanho++;
11         return(SUCESSO);
12     }
13 }
```

Inserir célula por chave (ascendente), por contiguidade:

```
1  int insOrdem (ListaLinear * lista, Celula celula) {
2      unsigned int i, j;
3
4      if (lista->tamanho == TAMANHO_MAXIMO_LISTA) {
5          return(FALHA); // a lista esta cheia: overflow
6      }
7      else {
8          if (lista->tamanho == 0) {
9              return (insInicio(lista, celula));
10         }
11         else {
12             if (celula.chave < lista->celulas[0].chave) {
13                 return (insInicio(lista, celula));
14             }
15             else {
16                 if (celula.chave >= lista->celulas[lista->tamanho-1].
                       chave) {
17                     return (insFinal(lista, celula));
18                 }
```

Inserir célula por chave (ascendente), por contiguidade:

```
1                    // continuacao do insOrdem...
2                    else {
3                        i = 0;
4                        while ((celula.chave >= lista->celulas[i].chave) && (i <
                             lista->tamanho)) {
5                            i++;
6                        }
7                        if (i == lista->tamanho) {
8                            return(insFinal(lista, celula));
9                        }
10                       else {
11                           for (j = lista->tamanho;(j > i); j--) {
12                               lista->celulas[j] = lista->celulas[j-1];
13                           }
14                           lista->celulas[i] = celula;
15                           lista->tamanho++;
16                           return(SUCESSO);
17                       }
18                   }
19               }
20           }
21       }
22   }
```

Remover célula no início da lista, por contiguidade:

```
1  Celula remInicio (ListaLinear * lista) {
2      unsigned int i;
3      Celula celulaResultado;
4
5      if (lista->tamanho == 0) {
6          celulaResultado.chave = CHAVE_INVALIDA;
7          return(celulaResultado);
8      }
9      else {
10         celulaResultado = lista->celulas[0];
11         for (i = 0; (i < lista->tamanho - 1); i++) {
12             lista->celulas[i] = lista->celulas[i+1];
13         }
14         lista->tamanho--;
15         return(celulaResultado);
16     }
17 }
```

Remover célula no final da lista, por contiguidade:

```
1  Celula remFinal (ListaLinear * lista) {
2
3      Celula celulaResultado;
4
5      if (lista->tamanho == 0) {
6          celulaResultado.chave = CHAVE_INVALIDA;
7          return(celulaResultado);
8      }
9      else {
10         celulaResultado = lista->celulas[lista->tamanho - 1];
11         lista->tamanho--;
12         return(celulaResultado);
13     }
14 }
```

Remover célula baseado na chave, por contiguidade:

```
1  Celula remChave (ListaLinear * lista, Celula celula) {
2      unsigned int i, j, k;
3      unsigned int intQuantidadeRemocoes;
4
5      Celula celulaResultado;
6
7      if (lista->tamanho == 0) {
8          celulaResultado.chave = CHAVE_INVALIDA;
9          return(celulaResultado);
10     }
11     else {
12         if (celula.chave == lista->celulas[0].chave) {
13             while (celula.chave == lista->celulas[0].chave) {
14                 celulaResultado = remInicio(lista);
15                 if (celulaResultado.chave == CHAVE_INVALIDA) {
16                     return(celulaResultado);
17                 }
18             }
19         }
20         else {
```

Remover célula baseado na chave, por contiguidade:

```
1
2           if (celula.chave == lista->celulas[lista->tamanho - 1].chave) {
3               while (celula.chave == lista->celulas[lista->tamanho - 1].chave) {
4                   celulaResultado = remFinal(lista);
5                   if (celulaResultado.chave == CHAVE_INVALIDA) {
6                       return(celulaResultado);
7                   }
8               }
9           }
10          else {
11              i = 0;
12              while ((celula.chave > lista->celulas[i].chave) && (i < lista->tamanho)) { i++;}
13              if (i == lista->tamanho) {
14                  celulaResultado.chave = CHAVE_INVALIDA;
15                  return(celulaResultado);
16              }
17              else {
```

Remover célula baseado na chave, por contiguidade:

```
1                       intQuantidadeRemocoes = 0;
2                       j = i;
3                       while ((celula.chave == lista->celulas[j].chave) && (j < lista->tamanho)) {
4                           intQuantidadeRemocoes++;
5                           j++;
6                       }
7                       if (intQuantidadeRemocoes == 0) {
8                          celulaResultado.chave = CHAVE_INVALIDA;
9                          return(celulaResultado);
10                      }
11                      else {
12                         celulaResultado = lista->celulas[i];
13                         for (j = i; (j < (lista->tamanho - intQuantidadeRemocoes)); j++) {
14                             if (j + intQuantidadeRemocoes < lista->tamanho) {
15                                lista->celulas[j] = lista->celulas[j + intQuantidadeRemocoes];
16                             }
17                         }
18                         lista->tamanho -= intQuantidadeRemocoes;
19                         return(celulaResultado);
20                      }
21                  }
22              }
23          }
24      }
25  }
```

Bibliografia

- ASCÊNCIO, A. F. G. e ARAÚJO, G. S. de., *Estruturas de dados*, São Paulo: Pearson Education, 2010.
- CORMEN, T. H. *et al.*, *Introduction to algorithms*, 3^{th} ed., MIP Press, 2009.
- SKIENA, S. S., *The algorithm design manual*, 2^{nd} ed., Springer-Verlag, 2008.
- MARTINS, P. R. (org.), *Algoritmos e estruturas de dados*, São Paulo: Pearson Education, 2009.
- TANEMBAUN, A. A., LANGSAM, Y. e AUGUSTEIN, M., *Estruturas de dados usando C*, São Paulo: Makron Books, 1995.

Saiba Mais...

- vídeos indicados na área da disciplina na plataforma de educação;
- vídeos disponíveis na *web*. Por exemplo, no YouTube;
- textos explicativos, e códigos-fonte, publicados em diversos *websites* que tratam sobre estruturas de dados.