# Listas sequenciais e algoritmos de busca

Programação II – Engenharia de Telecomunicações

Prof. Emerson Ribeiro de Mello

mello@ifsc.edu.br

INSTITUTO FEDERAL
Santa Catarina
Campus São José

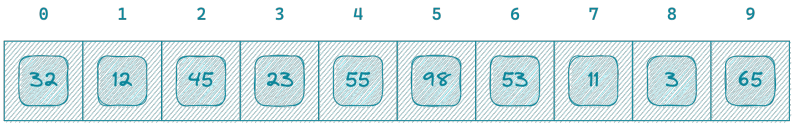
# Licenciamento

# Sumário

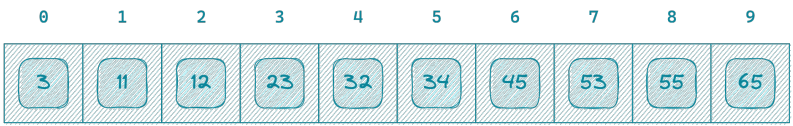
# Listas lineares em alocação sequencial

- Agrupa informações referentes a um conjunto de elementos comuns
- Um arranjo na linguagem C é um tipo de lista linear em alocação sequencial

- **Lista não ordenada**




- **Lista ordenada**



# Listas lineares em alocação sequencial

- Cada elemento possui um identificador único chamado de chave e assume-se que todas as chaves são distintas

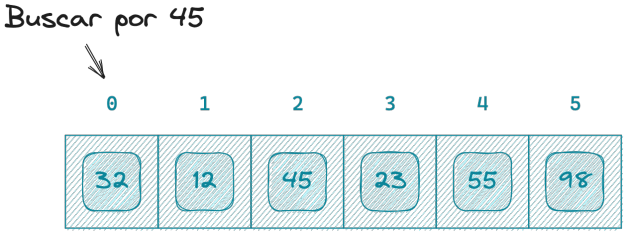
```c
typedef struct{
    int codigo;
    char nome[20];
} pessoa_t;

// lista que permite guardar até 100 inteiros. O valor é a chave
int lista[100];

// lista que permite guardar até 20 pessoas. O código de cada pessoa é a chave
pessoa_t contatos[20];
```

- **Operações básicas**
  - Busca
  - Inserção
  - Remoção
- **Outras operações**
  - Alteração
  - Ordenação
  - Combinação de duas listas

# Algoritmos de busca

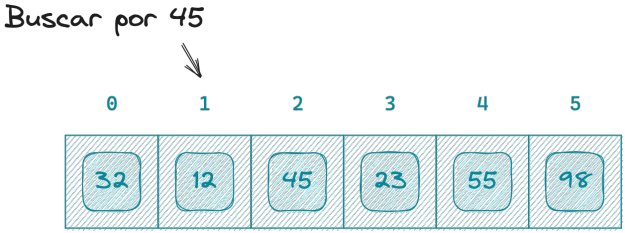
# Busca linear ou busca sequencial

- A partir do primeiro elemento, percorra toda a lista até encontrar ou chegar no final



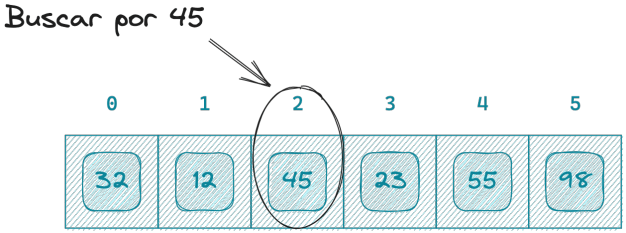
# Busca linear ou busca sequencial

- A partir do primeiro elemento, percorra toda a lista até encontrar ou chegar no final



# Busca linear ou busca sequencial

- A partir do primeiro elemento, percorra toda a lista até encontrar ou chegar no final



# Busca linear ou busca sequencial

## Busca por um elemento em uma lista não ordenada

**Algoritmo 1:** Busca linear

**Entrada:** vetor[1..n], x
**Saída:** verdade ou falso

```
1 para i de 1 até n faça
2     se x = vetor[i] então
3         retorna (verdade)
4     fim
5 fim
6 retorna (falso)
```

# Busca linear ou busca sequencial

## Busca por um elemento em uma lista não ordenada

**Algoritmo 1:** Busca linear

**Entrada:** vetor[1..n], x
**Saída:** verdade ou falso

```
1 para i de 1 até n faça
2 |   se x = vetor[i] então
3 |   |   retorna (verdade)
4 |   fim
5 fim
6 retorna (falso)
```

**Algoritmo 2:** Busca linear com sentinela

**Entrada:** vetor[1..n], x
**Saída:** verdade ou falso

```
1 vetor[n] ← x ;                    // sentinela
2 i ← 1
3 enquanto vetor[i] ≠ x faça
4 |   i ← i + 1
5 fim
6 se i ≠ n então
7 |   retorna (verdade)
8 fim
9 retorna (falso)
```

- No Algoritmo 2 assume-se que a lista tem $n - 1$ elementos armazenados
- Ambos algoritmos possuem complexidade $O(n)$, porém o Algoritmo 2 evita fazer uma comparação a cada iteração

# Busca linear ou busca sequencial

## Busca por um elemento em uma lista ordenada

**Algoritmo 3:** Busca linear em lista ordenada com sentinela

**Entrada:** vetor[1..n], x
**Saída:** verdade ou falso

```
1 vetor[n] ← x ;                                   // sentinela
2 i ← 1
3 enquanto vetor[i] < x faça
4 |   i ← i + 1
5 fim
6 se i = n ∨ vetor[i] ≠ x então
7 |   retorna (falso)
8 fim
9 retorna (verdade)
```

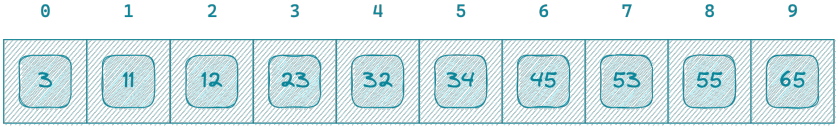
- A complexidade do pior caso é $O(n)$, igual dos algoritmos 1 e 2 para lista não ordenada, porém a complexidade média é melhor

# Busca linear ou busca sequencial

## Busca por um elemento em uma lista ordenada

Escolha um dos elementos na lista e indique o total de interações necessárias para encontrá-lo

- **Melhor caso:** 1
  - se escolheu o número 3
- **Pior caso:** *n*
  - se escolheu o número 65 ou um número que não esteja na lista



# Busca binária

- **Algoritmo para listas ordenadas** que divide a área de busca pela metade a cada iteração
- Complexidade $O(\log n)$
- A implementação recursiva segue a abordagem de **divisão e conquista**
- A biblioteca `stdlib.h` provê a função `bsearch`[1]

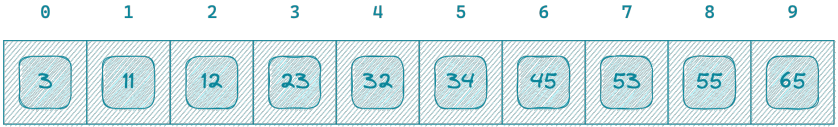
[1] https://cplusplus.com/reference/cstdlib/bsearch/

# Divisão e conquista

Divida um problema grande, de forma recursiva, em problemas menores até que possa ser resolvido de forma trivial

1. **Divisão**
   - Problema maior é dividido em subproblemas menores e similares ao original
2. **Conquista**
   - Quando problema é pequeno o suficiente, a solução do problema é calculada
3. **Combinação**
   - Resultados dos problemas menores são combinados até chegar na solução do problema original

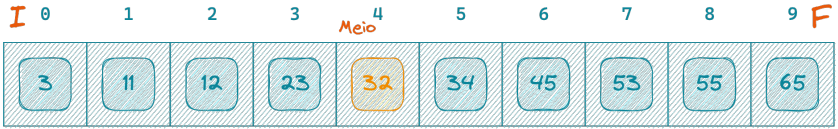
# Busca binária




## Buscar pelo número 34

1. Escolha o elemento que está no meio da lista: $M = I + (F - I)/2$
2. Se encontrou, pare. Senão, se o $M <$ número desejado, então $I = M + 1$, senão $F = M - 1$. Volte para o passo 1

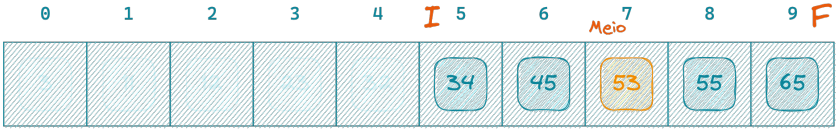
# Busca binária




## Buscar pelo número 34

1. Escolha o elemento que está no meio da lista: $M = I + (F - I)/2$
2. Se encontrou, pare. Senão, se o $M <$ número desejado, então $I = M + 1$, senão $F = M - 1$. Volte para o passo 1

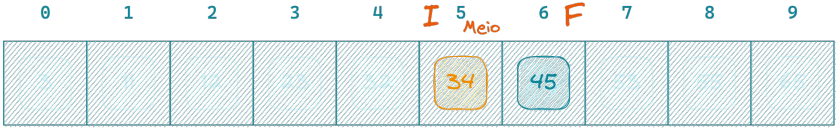
# Busca binária




## Buscar pelo número 34

1. Escolha o elemento que está no meio da lista: $M = I + (F - I)/2$
2. Se encontrou, pare. Senão, se o $M <$ número desejado, então $I = M + 1$, senão $F = M - 1$. Volte para o passo 1

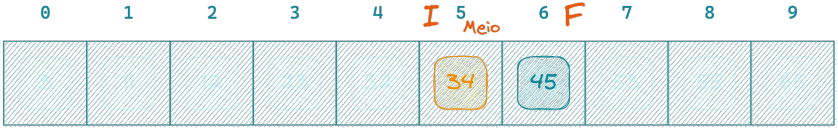
# Busca binária




## Buscar pelo número 34

1. Escolha o elemento que está no meio da lista: $M = I + (F - I)/2$
2. Se encontrou, pare. Senão, se o $M <$ número desejado, então $I = M + 1$, senão $F = M - 1$. Volte para o passo 1

# Busca binária




## Buscar pelo número 34

1. Escolha o elemento que está no meio da lista: $M = I + (F - I)/2$
2. Se encontrou, pare. Senão, se o $M <$ número desejado, então $I = M + 1$, senão $F = M - 1$. Volte para o passo 1

- Foram necessárias 3 iterações para encontrar
  - Se fosse com a busca linear, quantas iterações?
- Escolha qualquer outro número e veja quantas iterações são necessárias

# Busca binária

## Algoritmo iterativo

**Algoritmo 4:** Algoritmo iterativo da busca binária

**Entrada:** vetor[1..n], n, alvo
**Saída:** verdade ou falso

```
1  inicio ← 1;
2  fim ← n;
3  enquanto inicio ≤ fim faça
4  |  meio ← inicio + (fim − inicio)/2;
5  |  se vetor[meio] = alvo então
6  |  |  retorna (verdade);
7  |  senão se vetor[meio] < alvo então
8  |  |  inicio ← meio + 1;
9  |  senão
10 |  |  fim ← meio − 1;
11 |  fim
12 fim
13 retorna (falso)
```

# Busca binária

## Algoritmo recursivo

**Algoritmo 5:** Algoritmo recursivo da busca binária

**Entrada:** vetor[1..n], inicio, fim, alvo
**Saída:** verdade ou falso

```
1  se inicio ≤ fim então
2      meio ← inicio + (fim − inicio)/2;
3      se vetor[meio] = alvo então
4          retorna (verdade);
5      fim
6      se vetor[meio] > alvo então
7          retorna busca_binaria(vetor, inicio, meio-1, alvo);
8      fim
9      retorna busca_binaria(vetor, meio+1, fim, alvo);
10 fim
11 retorna (falso)
```

# Algoritmos de busca

## Resumo

- Só é possível usar busca binária em lista ordenada
- Busca binária é mais eficiente que a busca linear, principalmente para grandes valores de $n$
- Para o caso médio, com $n = 10^5 = 100.000$
  - Busca linear realiza $\frac{10^5+1}{2} \approx 50.000$ acessos
  - Busca binária realiza $\log_2 10^5 \approx 16$ acessos
- Busca binária é aplicada em cenários de aprendizado de máquina, computação gráfica e em banco de dados

# Inserção e remoção em listas sequenciais

# Operações de inserção e remoção

**Para evitar ambiguidade, assume-se que todas as chaves são distintas (únicas)**

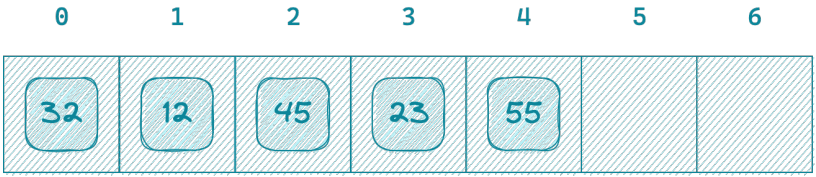
- **Lista não ordenada**
  - Insere no final
  - Para remover, primeiro **é necessário buscar pelo elemento** e depois mover último elemento para posição do elemento que será removido
- **Lista ordenada**
  - Para inserir, **buscar pela posição** onde o elemento deverá ser inserido e deslocar elementos posteriores àquele que será inserido
  - Para remover, primeiro **é necessário buscar pelo elemento** e depois deslocar elementos posteriores àquele que será removido

## Complexidade em tempo

Determinada pela **busca** e pela **movimentação** dos elementos

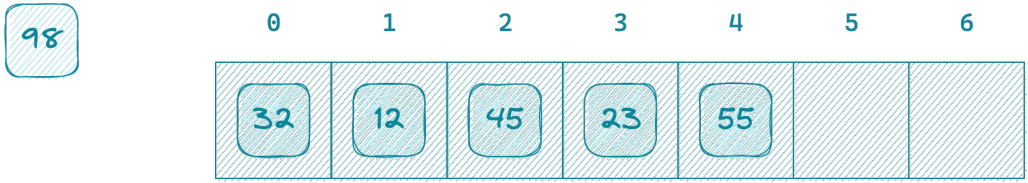
# Lista não ordenada

## Inserção de novo elemento no final – complexidade $O(1)$



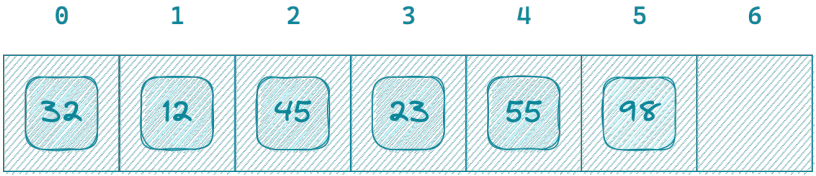
# Lista não ordenada

## Inserção de novo elemento no final – complexidade $O(1)$



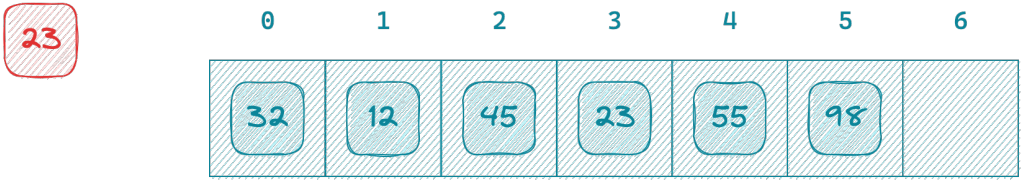
# Lista não ordenada

## Inserção de novo elemento no final – complexidade $O(1)$



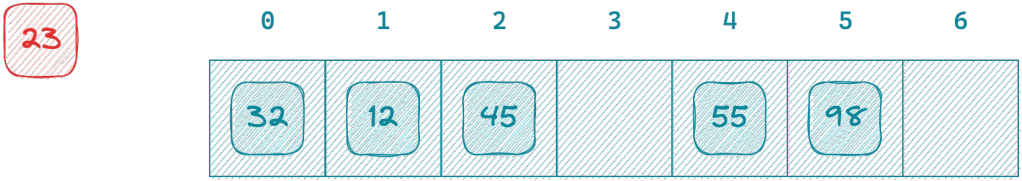
# Lista não ordenada

**Remoção – É necessário fazer uma busca linear pelo elemento, assim complexidade $O(n)$**



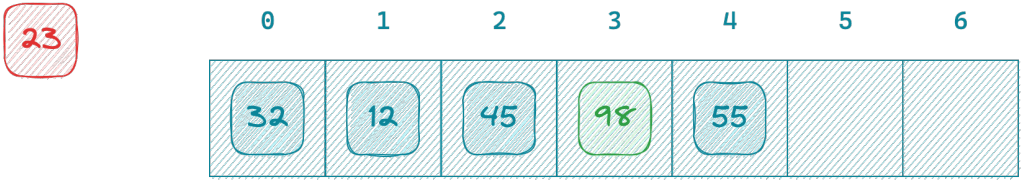
# Lista não ordenada

## Remoção – É necessário fazer uma busca linear pelo elemento, assim complexidade $O(n)$



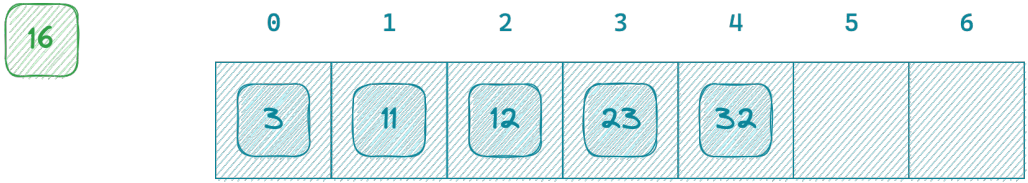
# Lista não ordenada

**Remoção – É necessário fazer uma busca linear pelo elemento, assim complexidade $O(n)$**



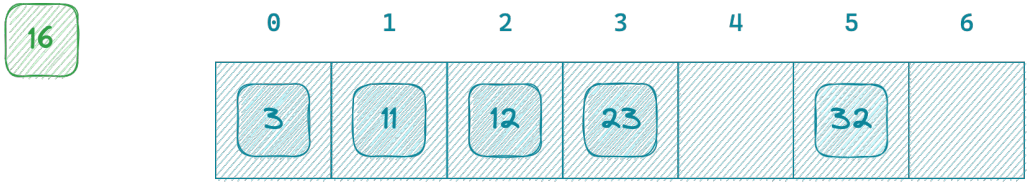
# Lista ordenada

## Inserção – complexidade $O(n)$



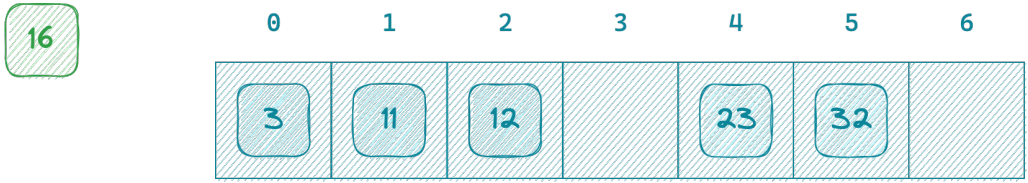
# Lista ordenada

## Inserção – complexidade $O(n)$



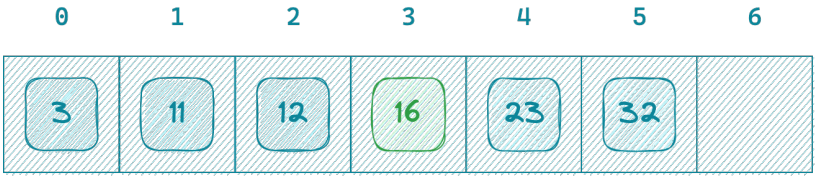
# Lista ordenada

## Inserção – complexidade $O(n)$



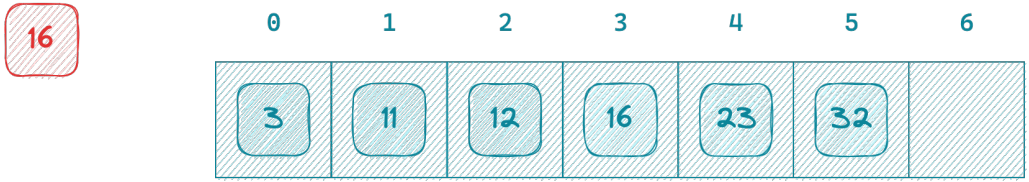
# Lista ordenada

## Inserção – complexidade $O(n)$



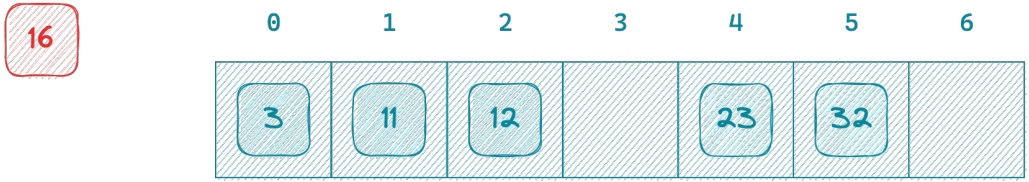
# Lista ordenada

**Remoção — Fazer busca binária pelo elemento ($O(\log n)$) e mover elementos ($O(n)$), logo complexidade $O(n)$**



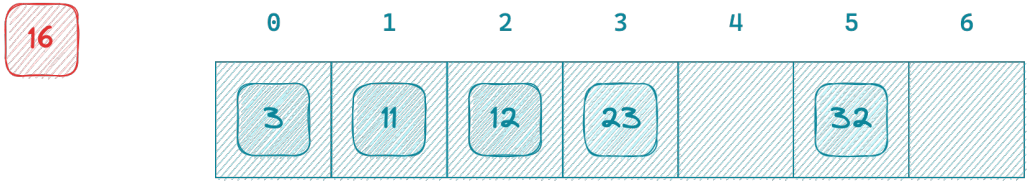
# Lista ordenada

**Remoção — Fazer busca binária pelo elemento ($O(\log n)$) e mover elementos ($O(n)$), logo complexidade $O(n)$**



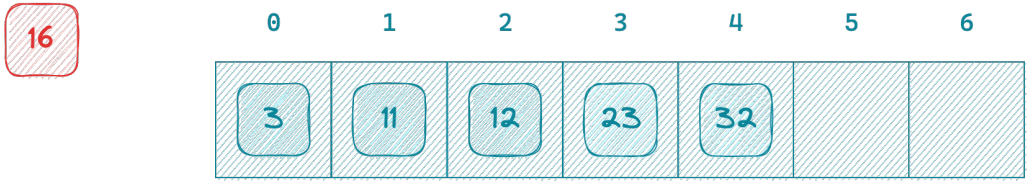
# Lista ordenada

**Remoção — Fazer busca binária pelo elemento ($O(\log n)$) e mover elementos ($O(n)$), logo complexidade $O(n)$**



# Lista ordenada

**Remoção — Fazer busca binária pelo elemento ($O(\log n)$) e mover elementos ($O(n)$), logo complexidade $O(n)$**



# Listas lineares em alocação sequencial

## Resumo

- Em geral são usadas em situações quando sofrem poucas remoções e inserções ao longo do tempo
- Situação favorável quando inserções ou remoções não acarretarem em deslocamento dos nós
  - Quando os nós estão em posições especiais, como a primeira ou a última posição
    - Filas, pilhas e *deques*[2] satisfazem tais condições

[2] Abreviação para *double ended queue (fila com dupla terminação)*

# Curiosidade

- O algoritmo original da **busca binária** tinha um *bug* de estouro de representação de inteiros (*overflow*) que só foi descoberto 20 anos depois
  - Em 2006 foi corrigido na implementação da biblioteca padrão do Java[3]
  - Vetor de tamanho $2^{31} - 1$ e o alvo na posição $2^{31} - 2$

**Algoritmo 6:** Busca binária correta

**Entrada:** vetor[1..n], n, alvo
**Saída:** verdade ou falso

```
1  inicio ← 1;
2  fim ← n;
3  enquanto inicio ≤ fim faça
4  |  meio ← inicio + (fim − inicio)/2;
5  |  se vetor[meio] = alvo então
6  |  |  retorna (verdade);
7  |  senão se vetor[meio] < alvo
   |   então
8  |  |  inicio ← meio + 1;
9  |  senão
10 |  |  fim ← meio − 1;
11 |  fim
12 fim
13 retorna (falso)
```

**Algoritmo 7:** Overflow

**Entrada:** vetor[1..n], n, alvo
**Saída:** verdade ou falso

```
1  inicio ← 1;
2  fim ← n;
3  enquanto inicio ≤ fim faça
4  |  meio ← (inicio + fim)/2;
5  |  se vetor[meio] = alvo então
6  |  |  retorna (verdade);
7  |  senão se vetor[meio] < alvo
   |   então
8  |  |  inicio ← meio + 1;
9  |  senão
10 |  |  fim ← meio − 1;
11 |  fim
12 fim
13 retorna (falso)
```

[3] https://ai.googleblog.com/2006/06/extra-extra-read-all-about-it-nearly.html

# Exercício 1

- Crie um repositório na sua conta GitHub com o nome `libprg` para hospedar uma biblioteca em C baseada no exemplo[4] (ou faça um *fork*)
- Crie o arquivo `lista_linear.c` e implemente ali funções para representar **listas lineares em alocação sequencial** (ordenada e não ordenada) para armazenamento de números inteiros
  - É esperado que seja feito uso de alocação dinâmica de memória
  - As definições das funções devem ser feitas obrigatoriamente no arquivo de cabeçalho `libprg.h`
- Além das operações de inserção e remoção, deve-se implementar os seguintes algoritmos de busca
  - Busca linear
  - Busca binária iterativa (somente para listas ordenadas)
  - Busca binária recursiva (somente para listas ordenadas)

[4] https://github.com/emersonmello/libprg

# Exercício 2

- Crie um projeto com CMake de uma aplicação em C que dependa da biblioteca `libprg`, criada no exercício anterior
  - Faça uso do `FetchContent` no CMake para baixar a dependência
- Faça um menu interativo que permita ao usuário executar as seguintes operações
  1. Criar uma lista povoada de tamanho *n* (usuário indica se será ordenada ou não)
  2. Inserir um novo número
  3. Remover um número
  4. Buscar por um número usando busca linear
  5. Buscar por um número usando busca binária iterativa
  6. Buscar por um número usando busca binária recursiva
- Deve-se imprimir na tela o tempo gasto (relógio de parede) para executar cada uma das operações descritas nos itens de 2 a 6 da lista acima

# Ferramentas para ajudar no desenvolvimento em C

# Valgrind

- Ferramenta[5] que pode ajudá-los na escrita de algoritmos mais eficientes e corretos
- Pode ser usada para encontrar **vazamento de memória** e acesso a **posições inválidas de memória**
  - Casos típicos quando se está trabalhando com alocação dinâmica de memória e vetores
  - Muito útil para achar a causa de um *Segmentation fault*
- Pode ser usado de forma integrada com o CLion[6], mas é necessário que tenha o valgrind instalado em sua máquina

```
# gerando binário com informações de depuração. Parâmentro -g
gcc -g main.c -o saida

valgrind --leak-check=yes ./saida
```

[5] https://valgrind.org/docs/manual/quick-start.html
[6] https://www.jetbrains.com/help/clion/memory-profiling-with-valgrind.html

# GCC

## Parâmetros adicionais para encontrar vazamento de memória

```
# -Wall para apresentar todos avisos (warning) ao compilar
# -g inclui informações para depuração (-g0 a -g3)
# -fsanitize=address para detectar diferentes erros referentes a acesso a memória

gcc -Wall -g -fsanitize=address main.c -o saida
```

```c
// Adaptado de https://valgrind.org/docs/manual/quick-start.html
#include <stdlib.h>

void f(void){
    int* x = malloc(100 * sizeof(int));
    x[100] = 0; // problema 1: acesso a posição inválida
}               // problema 2: vazamento de memória - x não é liberado

int main(void){
    for(int i=0; i < 1000; i++){
        f();
    }
    return 0;
}
```

# Referências

## Aula baseada em


- CORMEN, Thomas H. et al. **Algoritmos: teoria e prática**. LTC, 2012. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/9788595158092>.
- SZWARCFITER, Jayme Luiz; MARKENZON, Lilian. **Estruturas de dados e seus algoritmos**. LTC, 2010. Disponível em: <https://app.minhabiblioteca.com.br/reader/books/978-85-216-2995-5>.