

Estrutura de Dados: Aula 11

Iago A. Carvalho
Universidade Estadual de Campinas

Slide do professor Iago A. Carvalho - Universidade Estadual de Campinas

4 de junho de 2024

- 1 Complexidade de Algoritmos
- 2 Recursividade
- 3 Filas
- 4 Pilhas
- 5 Árvores
- 6 Ordenação
- 7 Bubble Sort
- 8 Insertion Sort
- 9 Métodos de busca
- 10 Busca Binária
- 11 Questões

Complexidade de algoritmos é uma medida do recurso (tempo ou espaço) necessário para executar um algoritmo.

As duas principais dimensões são:

Complexidade de tempo: Quanto tempo leva para executar.

Complexidade de espaço: Quanto espaço de memória é necessário.

Notação Big O é usada para descrever a complexidade de um algoritmo em termos do seu pior caso.

Representa a taxa de crescimento do tempo ou espaço conforme o tamanho da entrada aumenta.

Exemplos:

$O(1)$: Tempo constante.

$O(n)$: Tempo linear.

$O(n^2)$: Tempo quadrático.

$O(1)$ - Tempo Constante:

```
função constante(arr)
    retorne arr[0]
```

$O(n)$ - Tempo Linear:

```
função linear(arr)
    soma = 0
    para i de 0 até tamanho(arr)-1
        soma += arr[i]
    retorne soma
```

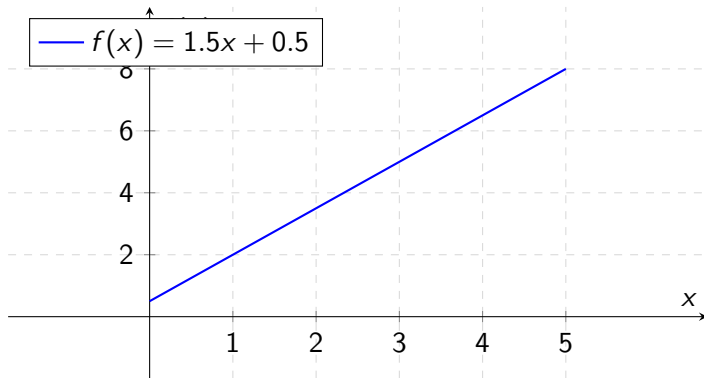
Uma **função linear** tem a forma $f(x) = ax + b$.

Uma **função linear** tem a forma $f(x) = ax + b$.
O gráfico de uma função linear é uma linha reta.

Uma **função linear** tem a forma $f(x) = ax + b$.

O gráfico de uma função linear é uma linha reta.

A complexidade linear, $O(n)$, significa que o tempo de execução cresce proporcionalmente ao tamanho da entrada.



Algoritmos com complexidade $O(n^2)$ têm um tempo de execução que cresce quadraticamente com o tamanho da entrada.

Exemplo: Ordenação por inserção.

```
função ordenacaoPorInsercao(arr)
  para i de 1 até tamanho(arr)-1
    chave = arr[i]
    j = i - 1
    enquanto j >= 0 e arr[j] > chave
      arr[j + 1] = arr[j]
      j = j - 1
    arr[j + 1] = chave
```

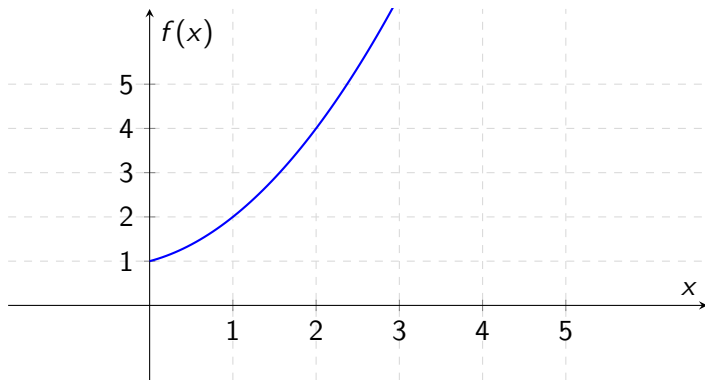
Uma **função quadrática** tem a forma $f(x) = ax^2 + bx + c$.

Uma **função quadrática** tem a forma $f(x) = ax^2 + bx + c$.
O gráfico de uma função quadrática é uma parábola.

Uma **função quadrática** tem a forma $f(x) = ax^2 + bx + c$.

O gráfico de uma função quadrática é uma parábola.

A complexidade quadrática, $O(n^2)$, significa que o tempo de execução cresce proporcionalmente ao quadrado do tamanho da entrada.



Algoritmos com complexidade **$O(\log n)$** são muito eficientes, diminuindo o problema pela metade a cada passo.

Exemplo: Pesquisa binária em um array ordenado.

```
função pesquisaBinaria(arr, chave)
    início = 0
    fim = tamanho(arr) - 1
    enquanto início <= fim
        meio = (início + fim) / 2
        se arr[meio] == chave
            retorne meio
        se arr[meio] < chave
            início = meio + 1
        senão
            fim = meio - 1
    retorne -1
```

Uma **função logarítmica** tem a forma $f(x) = \log_a(x)$, onde $a > 1$.

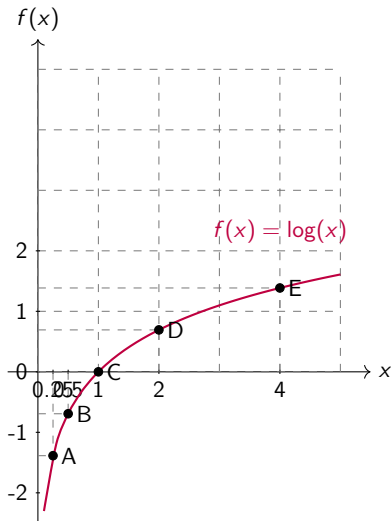
Uma **função logarítmica** tem a forma $f(x) = \log_a(x)$, onde $a > 1$.

O gráfico de uma função logarítmica é uma curva que cresce lentamente.

Uma **função logarítmica** tem a forma $f(x) = \log_a(x)$, onde $a > 1$.

O gráfico de uma função logarítmica é uma curva que cresce lentamente.

A complexidade logarítmica, $O(\log n)$, significa que o tempo de execução cresce logaritmicamente com o tamanho da entrada.



Uma **função exponencial** tem a forma $f(x) = a^x$, onde $a > 1$.

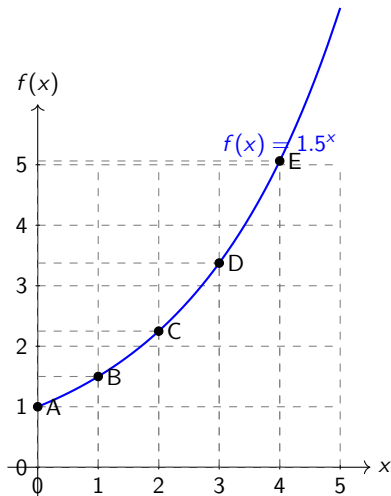
Uma **função exponencial** tem a forma $f(x) = a^x$, onde $a > 1$.

O gráfico de uma função exponencial é uma curva que cresce rapidamente.

Uma **função exponencial** tem a forma $f(x) = a^x$, onde $a > 1$.

O gráfico de uma função exponencial é uma curva que cresce rapidamente.

A complexidade exponencial, $O(2^n)$, significa que o tempo de execução cresce exponencialmente com o tamanho da entrada.

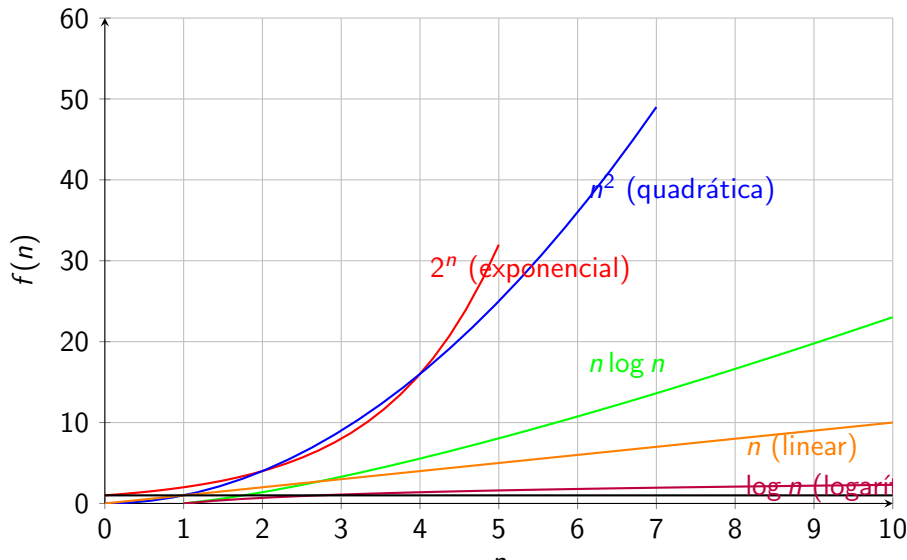


Algoritmos com complexidade $O(2^n)$ crescem exponencialmente com o tamanho da entrada.

Exemplo: Resolver o problema da Torre de Hanói.

```
função torreDeHanoi(n, origem, destino, auxiliar)
  se n == 1
    mover disco de origem para destino
  retorne
  torreDeHanoi(n-1, origem, auxiliar, destino)
  mover disco de origem para destino
  torreDeHanoi(n-1, auxiliar, destino, origem)
```

Comparação de Crescimento das Funções



A **complexidade espacial** refere-se à quantidade de memória extra que um algoritmo precisa.

Similar à análise de tempo, usa-se a notação Big O.

Exemplo: A função de Fatorial usa $O(n)$ de espaço para a pilha de chamadas recursivas.

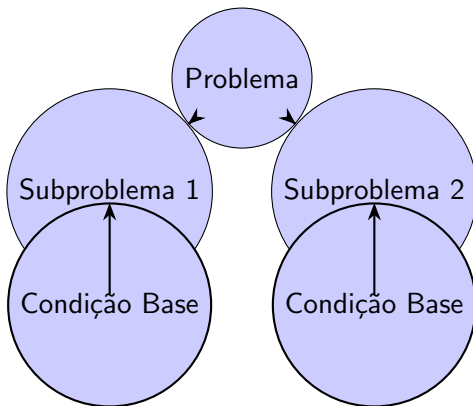
```
função fatorial(n)
  se n == 0
    retorne 1
  senão
    retorne n * fatorial(n-1)
```

- 1 Complexidade de Algoritmos
- 2 Recursividade**
- 3 Filas
- 4 Pilhas
- 5 Árvores
- 6 Ordenação
- 7 Bubble Sort
- 8 Insertion Sort
- 9 Métodos de busca
- 10 Busca Binária
- 11 Questões

Recursividade é uma técnica de programação onde uma função chama a si mesma para resolver subproblemas menores.

Recursividade é uma técnica de programação onde uma função chama a si mesma para resolver subproblemas menores.

Cada chamada recursiva deve aproximar o problema de uma **condição base** que pode ser resolvida diretamente.



O **fatorial** de um número n , denotado $n!$, é o produto de todos os números inteiros positivos até n .

A definição recursiva do fatorial é:

$$n! = \begin{cases} 1 & \text{se } n = 0 \\ n \times (n-1)! & \text{se } n > 0 \end{cases}$$

```
1 função fatorial(n)
2     se n == 0
3         retorne 1
4     senão
5         retorne n * fatorial(n-1)
```

A **sequência de Fibonacci** é uma série de números onde cada número é a soma dos dois anteriores.

Definição recursiva:

$$F(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ F(n-1) + F(n-2) & \text{se } n > 1 \end{cases}$$

```
1 função fibonacci(n)
2     se n == 0
3         retorne 0
4     se n == 1
5         retorne 1
6     senão
7         retorne fibonacci(n-1) + fibonacci(n-2)
```

Vantagens:

Código mais limpo e fácil de entender.

Vantagens:

Código mais limpo e fácil de entender.

Natural para problemas que podem ser divididos em subproblemas menores.

Vantagens:

Código mais limpo e fácil de entender.

Natural para problemas que podem ser divididos em subproblemas menores.

Desvantagens:

Pode levar a alta utilização de memória devido à pilha de chamadas.

Vantagens:

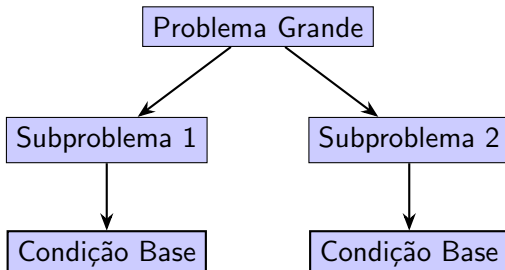
Código mais limpo e fácil de entender.

Natural para problemas que podem ser divididos em subproblemas menores.

Desvantagens:

Pode levar a alta utilização de memória devido à pilha de chamadas.

Em alguns casos, a recursão pode ser menos eficiente do que iterações.



- 1 Complexidade de Algoritmos
- 2 Recursividade
- 3 Filas**
- 4 Pilhas
- 5 Árvores
- 6 Ordenação
- 7 Bubble Sort
- 8 Insertion Sort
- 9 Métodos de busca
- 10 Busca Binária
- 11 Questões

Filas

- Uma impressora é compartilhada em um laboratório
- Alunos enviam documentos quase ao mesmo tempo



Filas

- Uma impressora é compartilhada em um laboratório
- Alunos enviam documentos quase ao mesmo tempo



Como gerenciar a lista de tarefas de impressão?

Fila

Fila:

Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**

Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”

Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo:



Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Enfileira**()



Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Enfileira**()



Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Enfileira**()




Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Enfileira**()



Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Desenfileira()**



Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Desenfileira()**




Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Enfileira**()




Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Enfileira**()



Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Enfileira**()




Fila

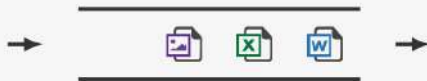
Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Enfileira**()



Fila

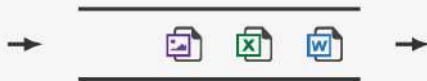
Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Desenfileira()**



Fila

Fila:

- Remove primeiro objetos **inseridos há mais tempo**
- **FIFO** (*first-in first-out*): primeiro a entrar é primeiro a sair

Operações:

- **Enfileira** (*queue*): adiciona item no “fim”
- **Desenfileira** (*dequeue*): remove item do “início”

Exemplo: **Desenfileira()**



- 1 Complexidade de Algoritmos
- 2 Recursividade
- 3 Filas
- 4 Pilhas**
- 5 Árvores
- 6 Ordenação
- 7 Bubble Sort
- 8 Insertion Sort
- 9 Métodos de busca
- 10 Busca Binária
- 11 Questões

Pilha

- Remove primeiro objetos **inseridos há menos tempo**

Pilha

- Remove primeiro objetos **inseridos há menos tempo**
- **LIFO** (*last-in first-out*): último a entrar é primeiro a sair

Pilha

- Remove primeiro objetos **inseridos há menos tempo**
- **LIFO** (*last-in first-out*): último a entrar é primeiro a sair



É como uma pilha de pratos:

Pilha

- Remove primeiro objetos **inseridos há menos tempo**
- **LIFO** (*last-in first-out*): último a entrar é primeiro a sair



É como uma pilha de pratos:

- **Empilha** os pratos limpos sobre os que já estão na pilha

Pilha

- Remove primeiro objetos **inseridos há menos tempo**
- **LIFO** (*last-in first-out*): último a entrar é primeiro a sair



É como uma pilha de pratos:

- **Empilha** os pratos limpos sobre os que já estão na pilha
- **Desempilha** o prato de cima para usar

Pilha

Operações:

Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha

Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo:



Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Empilha**(A)



Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Empilha**(A)



Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Empilha**(B)



Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Empilha**(B)

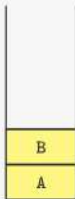


Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Desempilha()**



Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Desempilha()**



Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Empilha**(C)

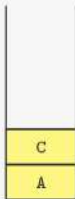


Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Empilha**(C)

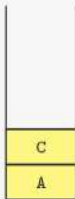


Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Empilha**(D)



Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Empilha**(D)



Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Desempilha()**

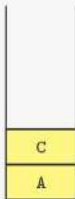


Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Desempilha()**

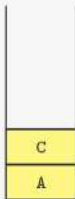


Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

Exemplo: **Desempilha()**



Pilha

Operações:

- **Empilha** (*push*): adiciona no topo da pilha
- **Desempilha** (*pop*): remove do topo da pilha

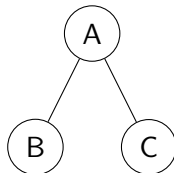
Exemplo: **Desempilha()**



- 1 Complexidade de Algoritmos
- 2 Recursividade
- 3 Filas
- 4 Pilhas
- 5 Árvores**
- 6 Ordenação
- 7 Bubble Sort
- 8 Insertion Sort
- 9 Métodos de busca
- 10 Busca Binária
- 11 Questões

Uma **árvore binária** é uma estrutura de dados em que cada nó tem no máximo dois filhos.

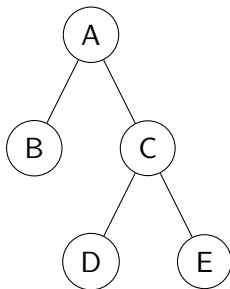
Cada nó contém um valor ou chave e dois ponteiros, um para o filho esquerdo e outro para o filho direito.



Altura: O comprimento do caminho mais longo da raiz até uma folha.

Profundidade: A distância entre a raiz e um determinado nó.

Número de Nós: Para uma árvore binária de altura h , o número máximo de nós é $2^{h+1} - 1$.

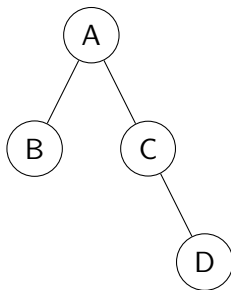


Para inserir um nó em uma árvore binária, começamos na raiz.

Comparamos o valor a ser inserido com o valor do nó atual.

Se for menor, movemos para a subárvore esquerda; se for maior, para a subárvore direita.

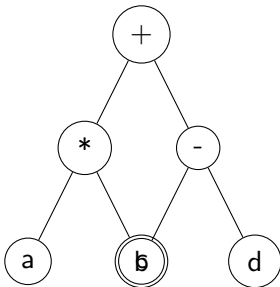
Repetimos o processo até encontrar uma posição vazia.



Pesquisa: Árvores de Busca Binária (BST) permitem busca, inserção e deleção eficientes.

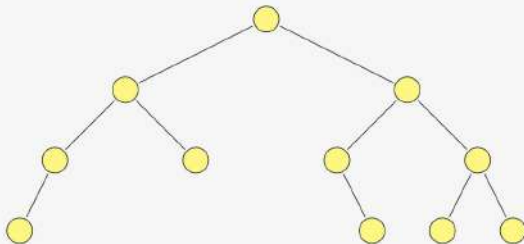
Estrutura de Dados: Usadas em heaps, árvores AVL e árvores vermelho-preto.

Expressões Matemáticas: Árvores binárias podem representar expressões aritméticas.



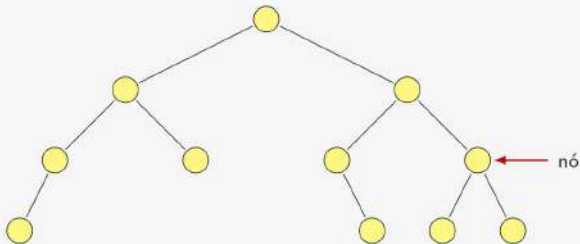
Árvores Binárias

Exemplo de uma árvore binária:



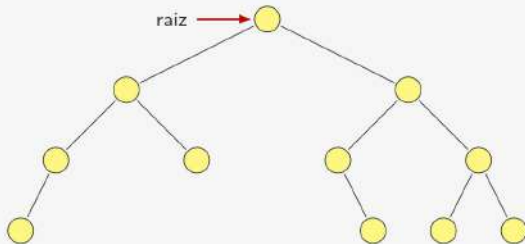
Árvores Binárias

Exemplo de uma árvore binária:



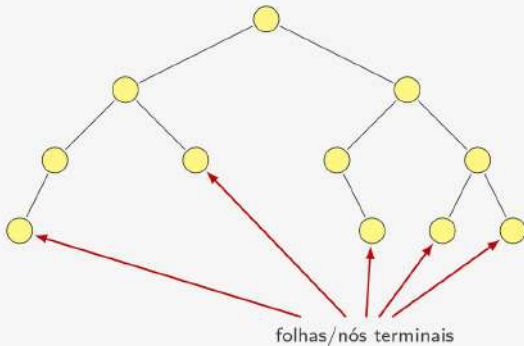
Árvores Binárias

Exemplo de uma árvore binária:



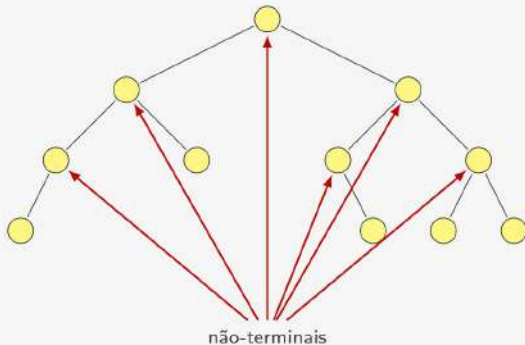
Árvores Binárias

Exemplo de uma árvore binária:



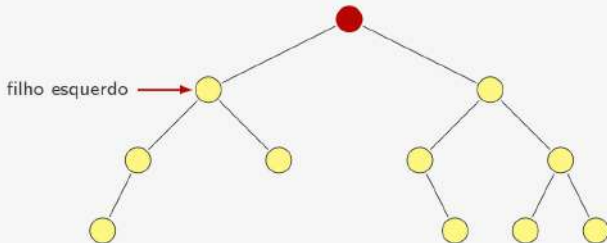
Árvores Binárias

Exemplo de uma árvore binária:



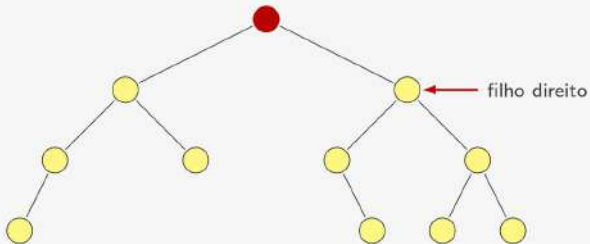
Árvores Binárias

Exemplo de uma árvore binária:



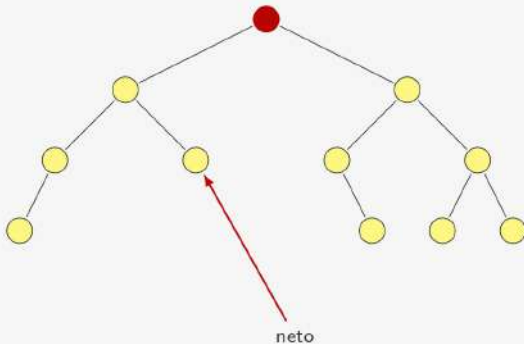
Árvores Binárias

Exemplo de uma árvore binária:



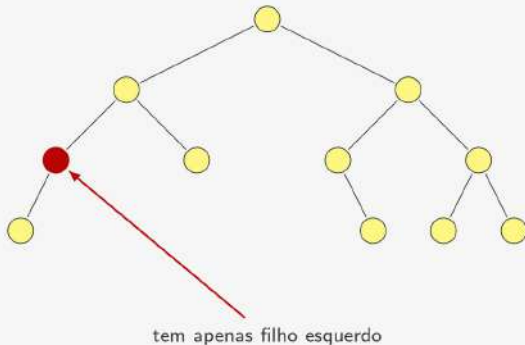
Árvores Binárias

Exemplo de uma árvore binária:



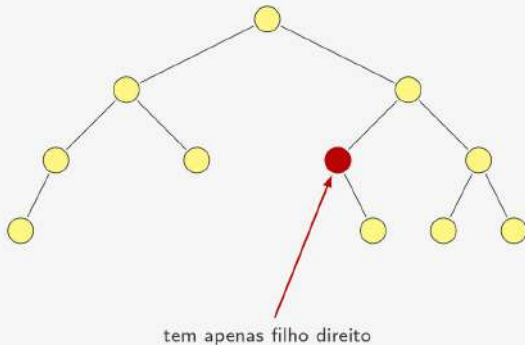
Árvores Binárias

Exemplo de uma árvore binária:



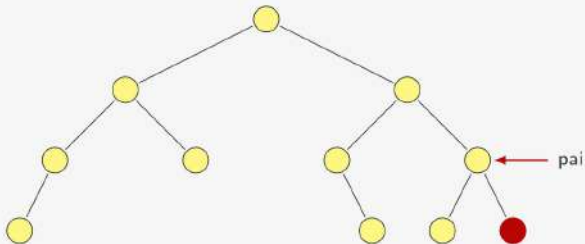
Árvores Binárias

Exemplo de uma árvore binária:



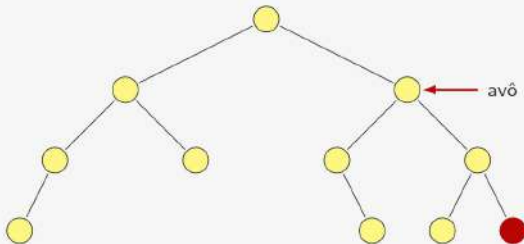
Árvores Binárias

Exemplo de uma árvore binária:



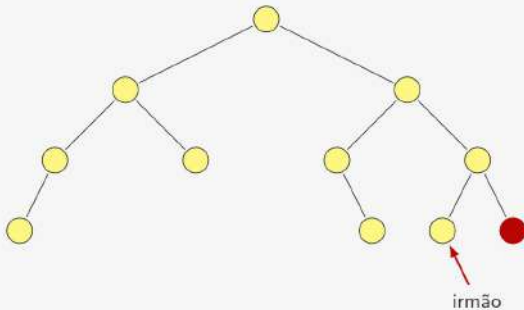
Árvores Binárias

Exemplo de uma árvore binária:



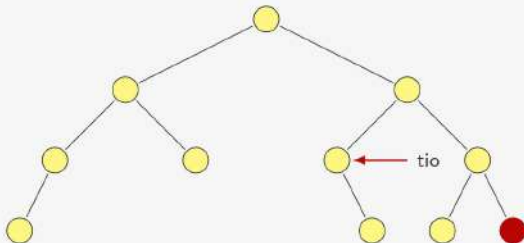
Árvores Binárias

Exemplo de uma árvore binária:



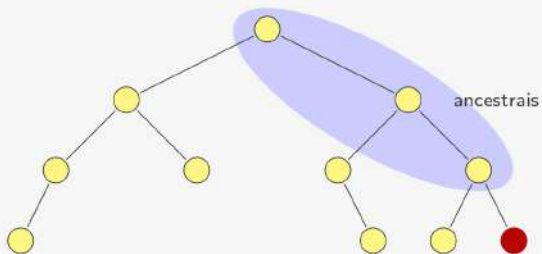
Árvores Binárias

Exemplo de uma árvore binária:



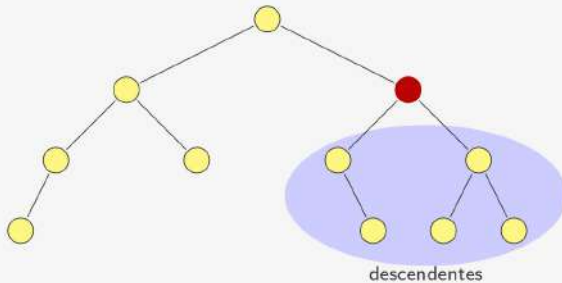
Árvores Binárias

Exemplo de uma árvore binária:



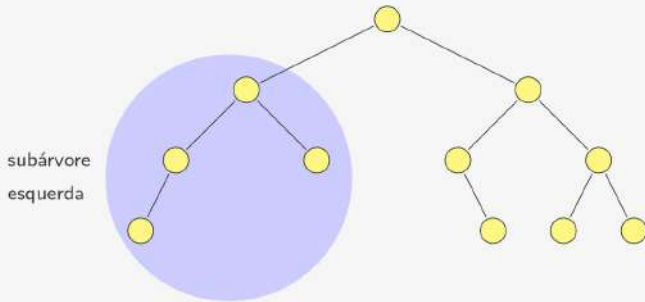
Árvores Binárias

Exemplo de uma árvore binária:



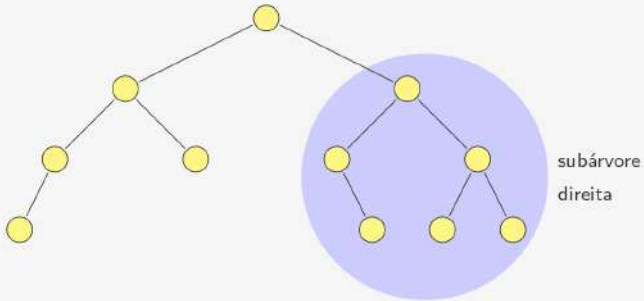
Árvores Binárias

Exemplo de uma árvore binária:



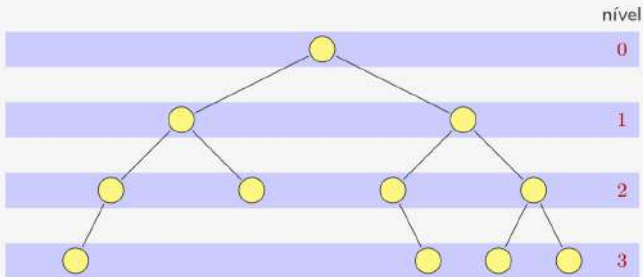
Árvores Binárias

Exemplo de uma árvore binária:



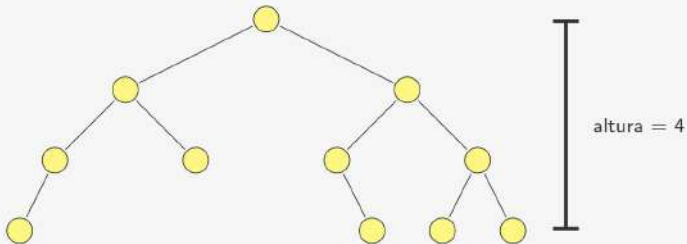
Árvores Binárias

Exemplo de uma árvore binária:



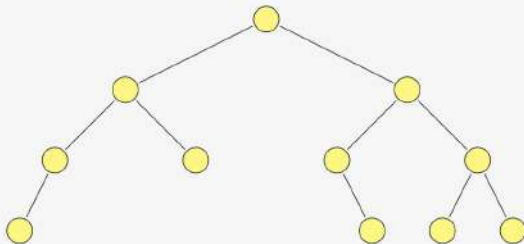
Árvores Binárias

Exemplo de uma árvore binária:



Árvores Binárias

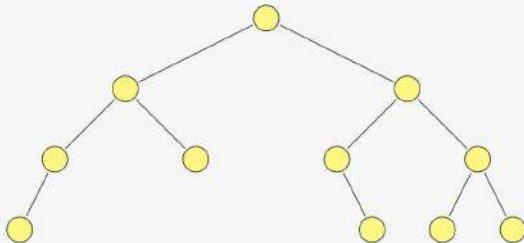
Exemplo de uma árvore binária:



Uma árvore binária é:

Árvores Binárias

Exemplo de uma árvore binária:

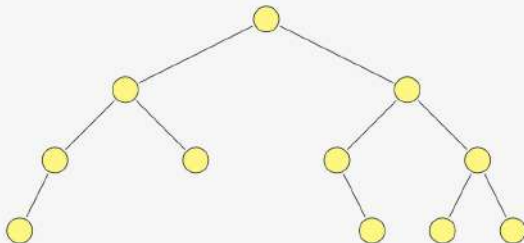


Uma árvore binária é:

- Ou o conjunto vazio

Árvores Binárias

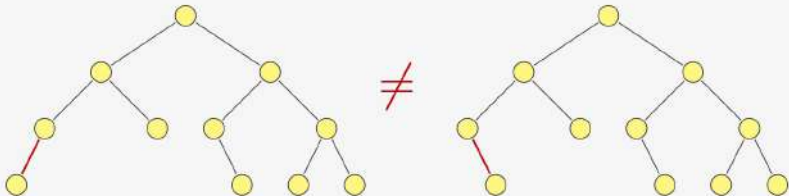
Exemplo de uma árvore binária:



Uma árvore binária é:

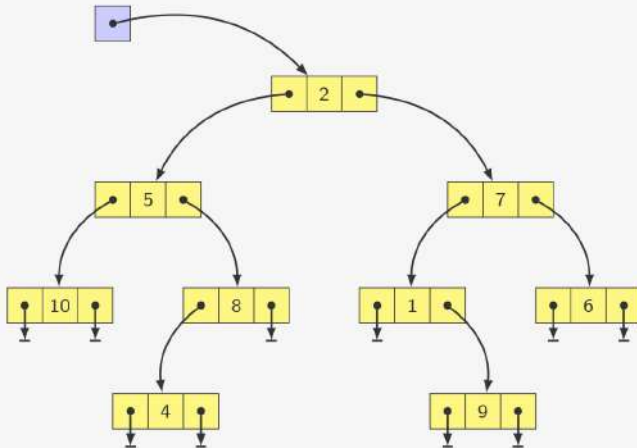
- Ou o conjunto vazio
- Ou um nó conectado a duas árvores binárias

Comparando com atenção



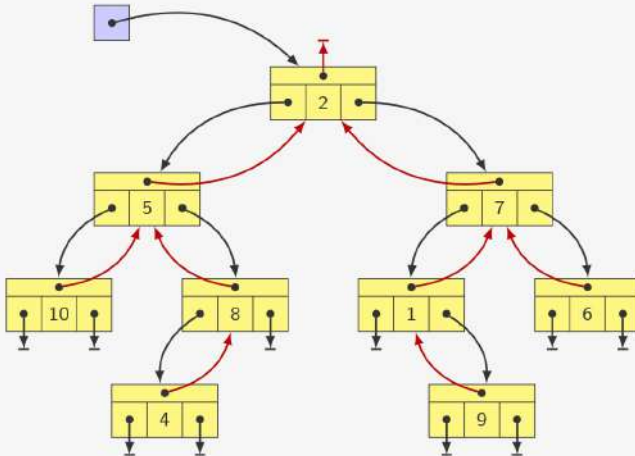
Ordem dos filhos é relevante!

Implementação



E se quisermos saber o pai de um nó?

Implementação com ponteiro para pai



- 1 Complexidade de Algoritmos
- 2 Recursividade
- 3 Filas
- 4 Pilhas
- 5 Árvores
- 6 Ordenação**
- 7 Bubble Sort
- 8 Insertion Sort
- 9 Métodos de busca
- 10 Busca Binária
- 11 Questões

Ordenação

Queremos ordenar um vetor

Ordenação

Queremos ordenar um vetor

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 1 | 6 | 5 | 2 | 4 | 0 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

Ordenação

Queremos ordenar um vetor

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 1 | 6 | 5 | 2 | 4 | 0 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

Nos códigos vamos ordenar vetores de `int`

Ordenação

Queremos ordenar um vetor

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 1 | 6 | 5 | 2 | 4 | 0 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`

Ordenação

Queremos ordenar um vetor

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 1 | 6 | 5 | 2 | 4 | 0 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|



| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos

Ordenação

Queremos ordenar um vetor

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 3 | 7 | 1 | 6 | 5 | 2 | 4 | 0 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|



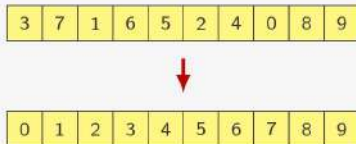
| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|

Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos
 - O valor usado para a ordenação é a `chave` de ordenação

Ordenação

Queremos ordenar um vetor



Nos códigos vamos ordenar vetores de `int`

- Mas é fácil alterar para comparar `double` ou `string`
- ou comparar `struct` por algum de seus campos
 - O valor usado para a ordenação é a `chave` de ordenação
 - Podemos até desempatar por outros campos

- 1 Complexidade de Algoritmos
- 2 Recursividade
- 3 Filas
- 4 Pilhas
- 5 Árvores
- 6 Ordenação
- 7 Bubble Sort**
- 8 Insertion Sort
- 9 Métodos de busca
- 10 Busca Binária
- 11 Questões

BubbleSort

Ideia:

BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos

BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve

BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {
```

BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {
```

BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)
```

BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)
```

BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```



i

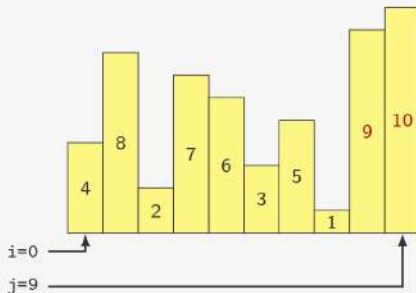
j

BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

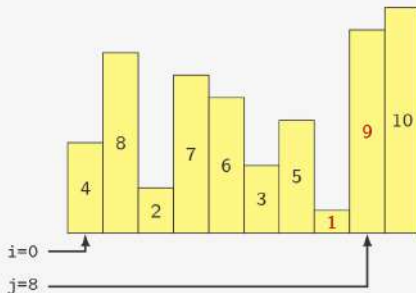


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

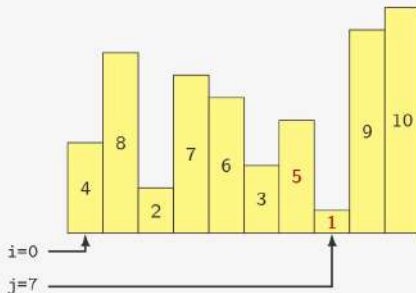


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

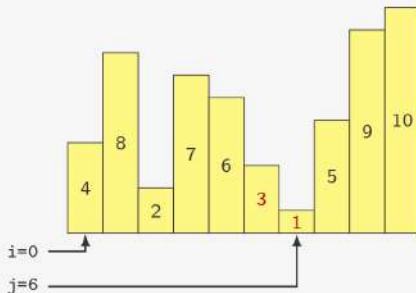


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

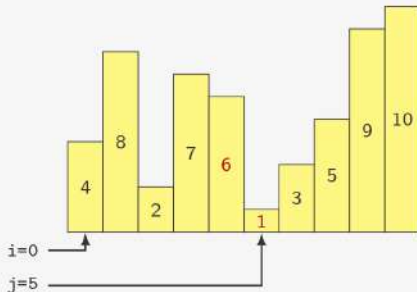


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

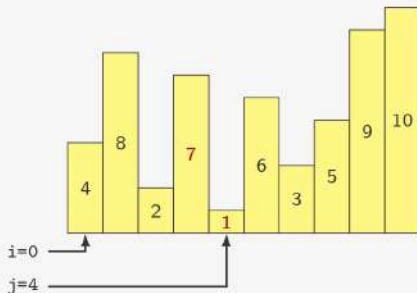


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

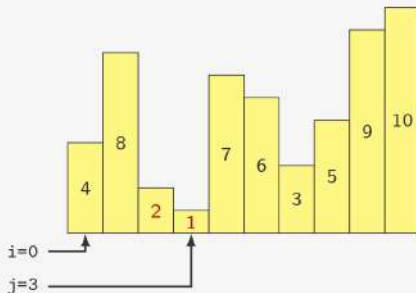


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

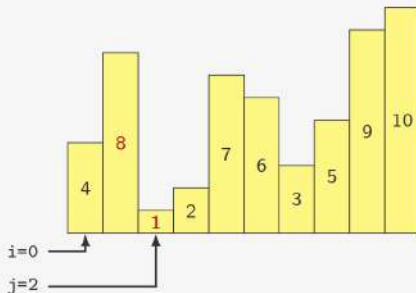


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

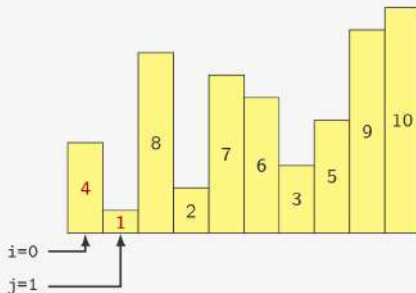


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

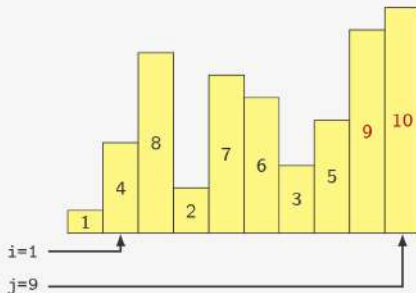


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

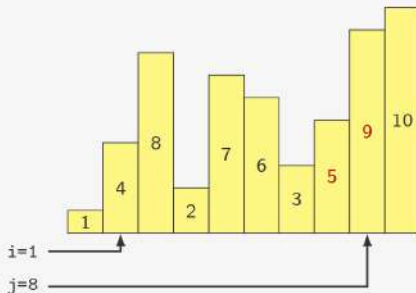


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

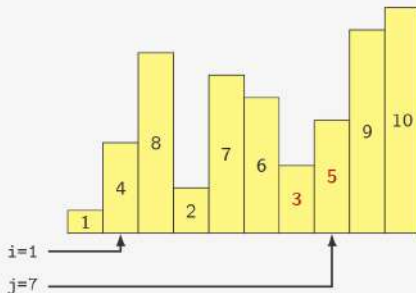


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

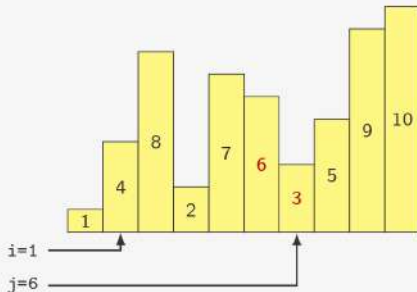


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

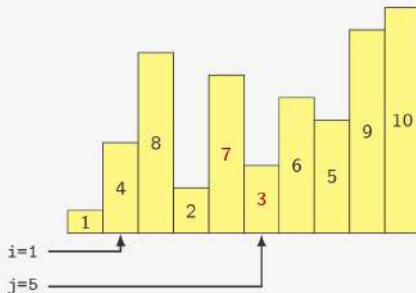


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

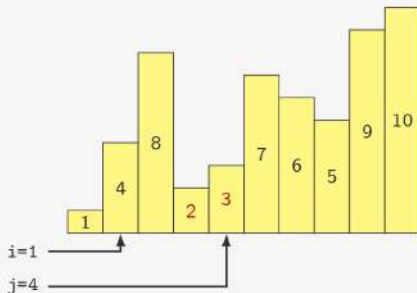


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

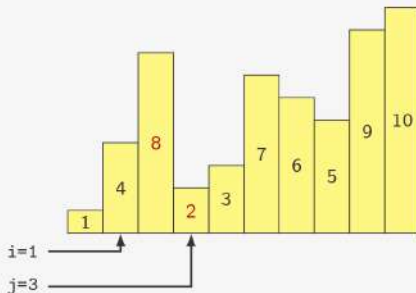


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

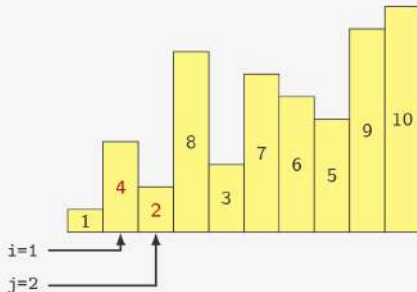


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

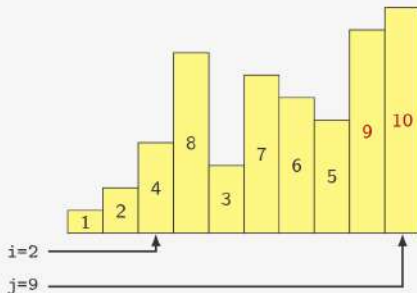


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

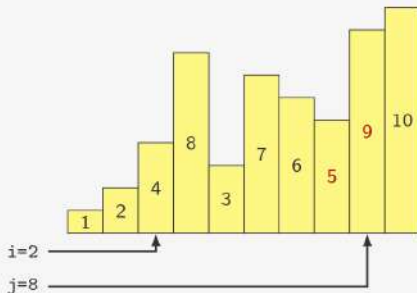


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

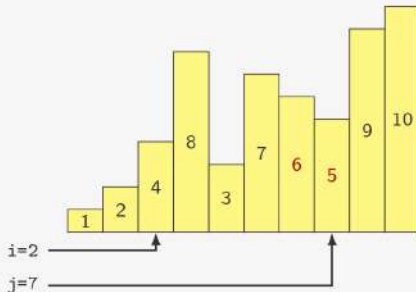


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

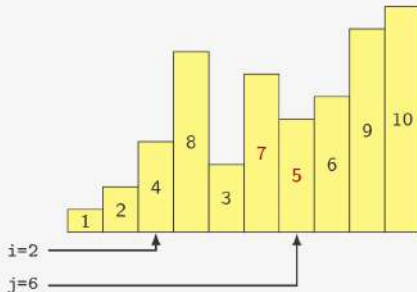


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

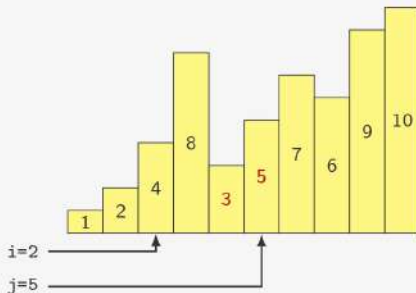


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

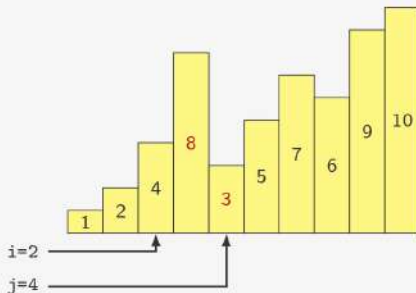


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

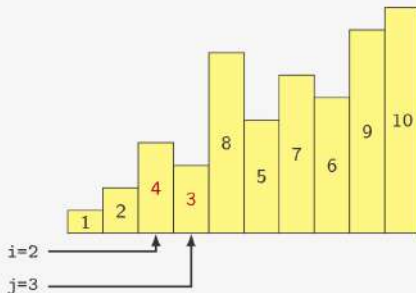


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

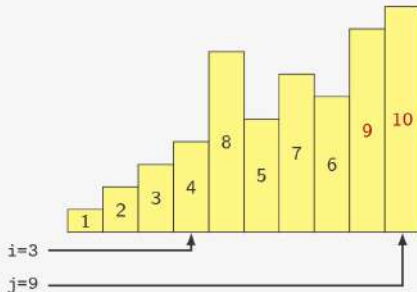


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

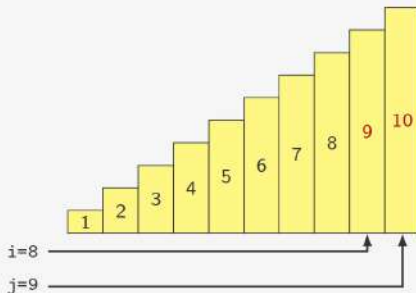


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```

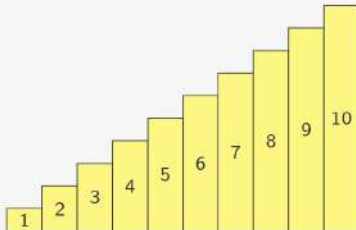


BubbleSort

Ideia:

- do fim para o começo, vamos trocando pares invertidos
- em algum momento, encontramos o elemento mais leve
- ele será trocado com os elementos que estiverem antes

```
1 void bubblesort(int *v, int n) {  
2     int i, j;  
3     for (i = 0; i < n - 1; i++)  
4         for (j = n - 1; j > i; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j-1], &v[j]);  
7 }
```



i

j

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int *v, int n) {
2     int i, j, trocou = 1;
3     for (i = 0; i < n - 1 && trocou; i++){
4         trocou = 0;
5         for (j = n - 1; j > i; j--){
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10        }
11    }
```

Parando quando não há mais trocas

Se não aconteceu nenhuma troca, podemos parar o algoritmo

```
1 void bubblesort_v2(int *v, int n) {
2     int i, j, trocou = 1;
3     for (i = 0; i < n - 1 && trocou; i++){
4         trocou = 0;
5         for (j = n - 1; j > i; j--){
6             if (v[j] < v[j-1]) {
7                 troca(&v[j-1], &v[j]);
8                 trocou = 1;
9             }
10        }
11    }
```

No pior caso toda comparação gera uma troca:

- 1 Complexidade de Algoritmos
- 2 Recursividade
- 3 Filas
- 4 Pilhas
- 5 Árvores
- 6 Ordenação
- 7 Bubble Sort
- 8 Insertion Sort**
- 9 Métodos de busca
- 10 Busca Binária
- 11 Questões

Ordenação por Inserção

Ideia:

Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado

Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta

Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort

Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {
```

Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {
```

Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)
```

Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)
```

Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

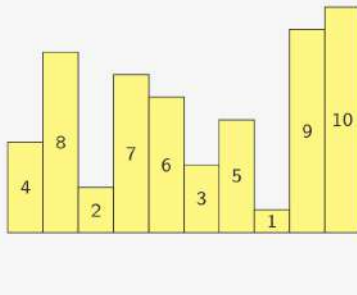
```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

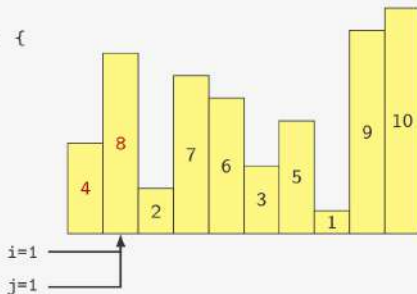


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

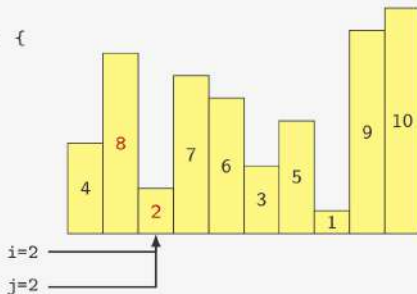


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

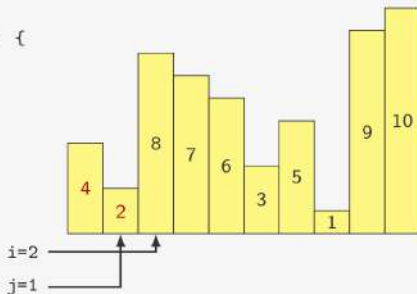


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

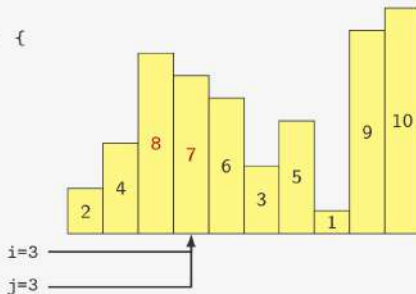


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

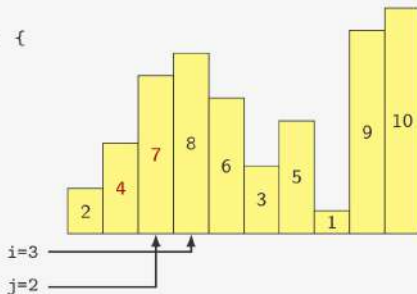


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

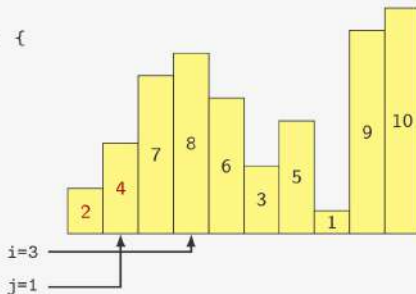


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

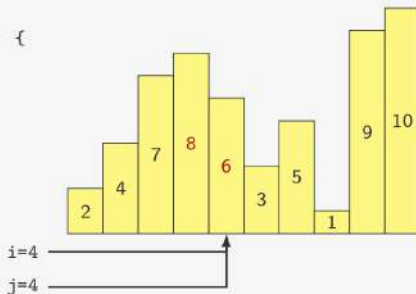


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

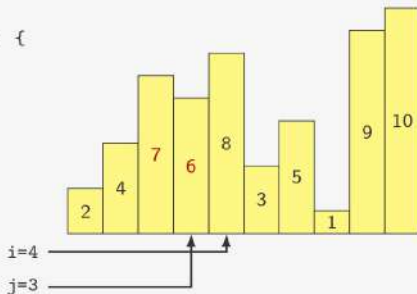


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

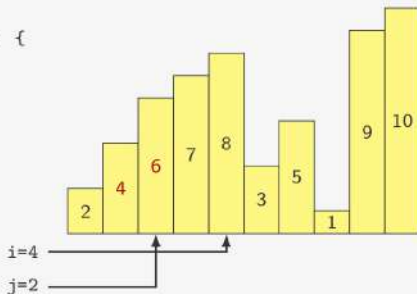


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

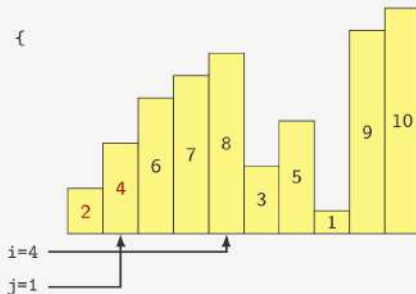


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {
2     int i, j;
3     for (i = 1; i < n; i++)
4         for (j = i; j > 0; j--)
5             if (v[j] < v[j-1])
6                 troca(&v[j], &v[j-1]);
7 }
```

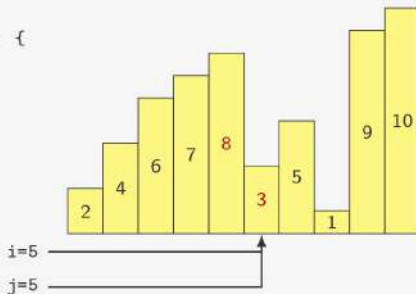


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

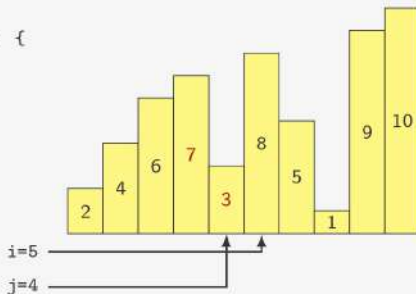


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

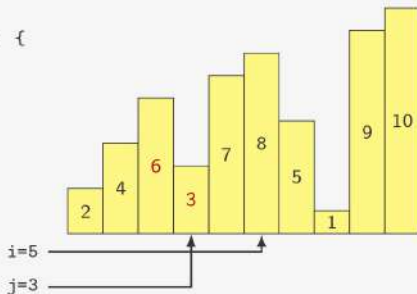


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

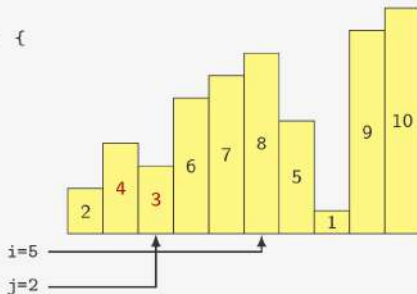


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

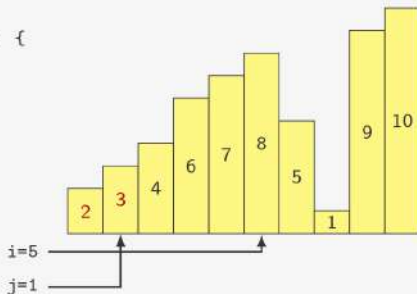


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

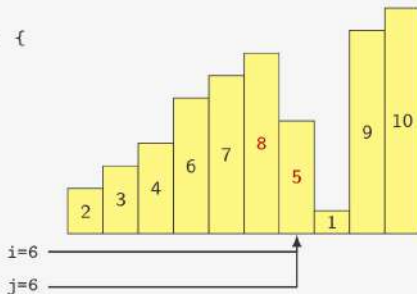


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

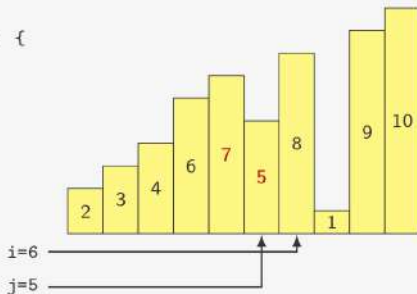


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

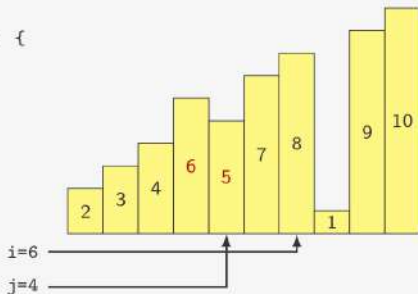


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

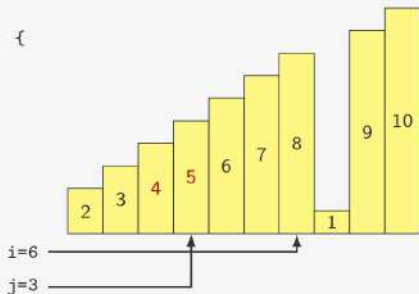


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

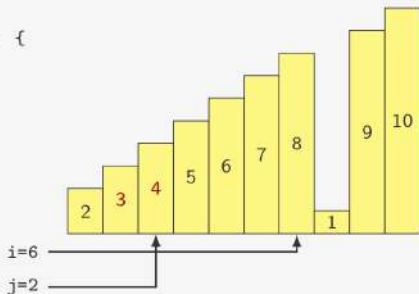


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

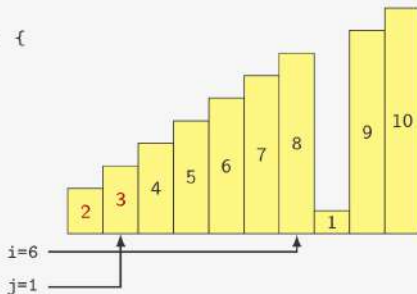


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

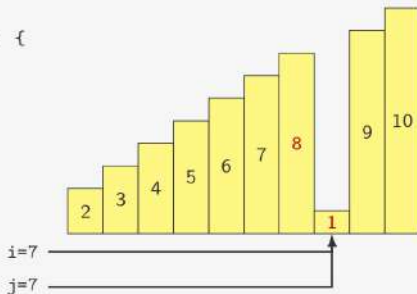


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

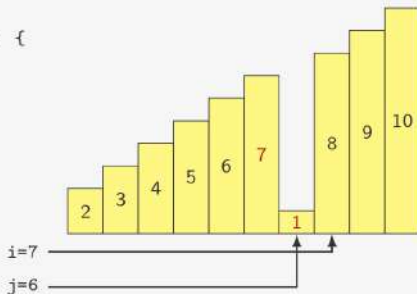


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

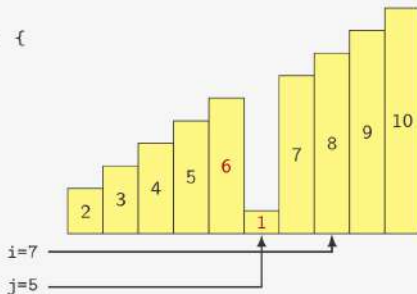


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

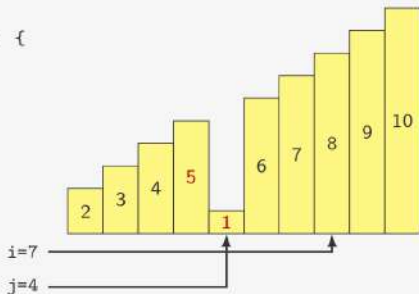


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

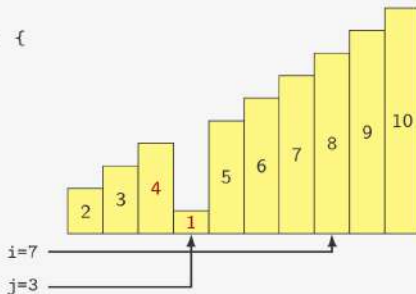


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

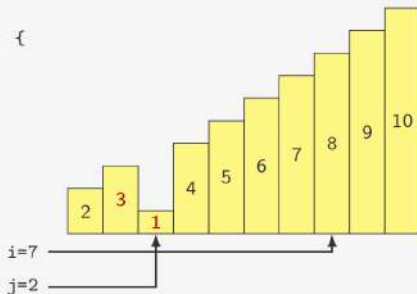


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

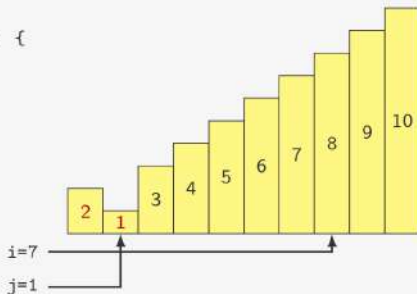


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

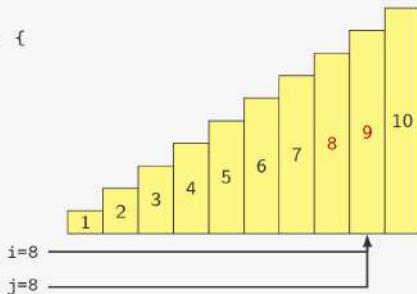


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

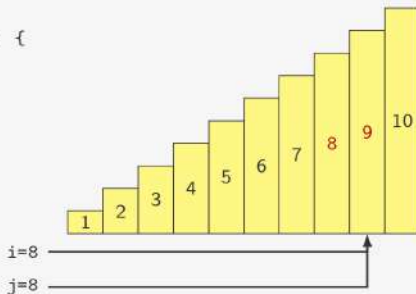


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

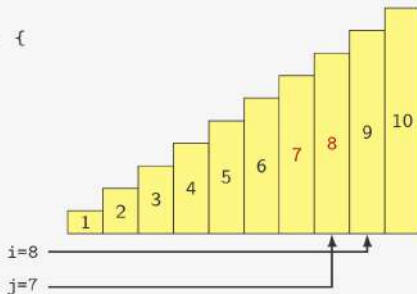


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

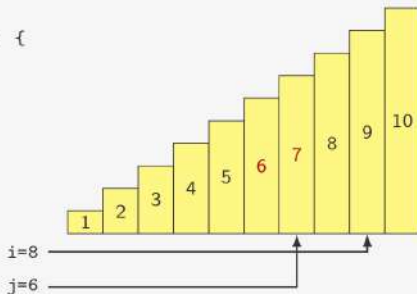


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

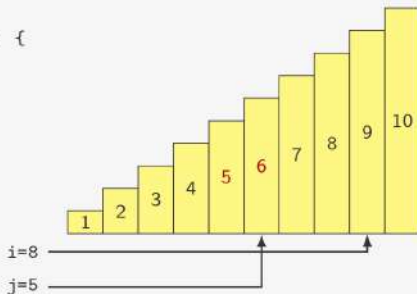


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

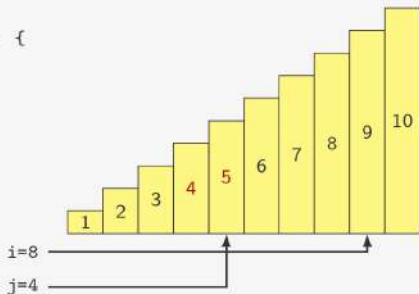


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

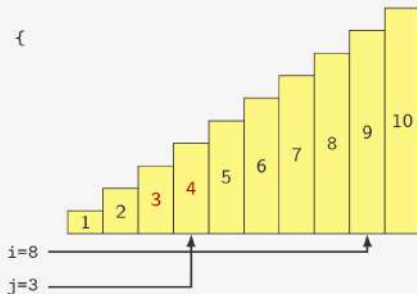


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

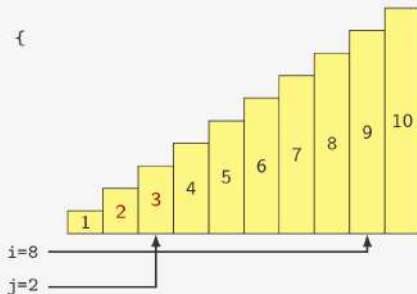


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

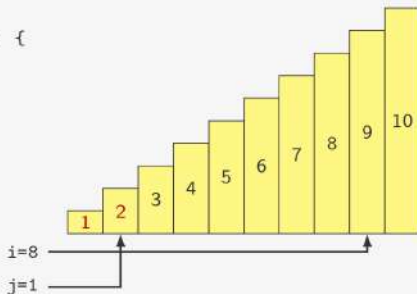


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```

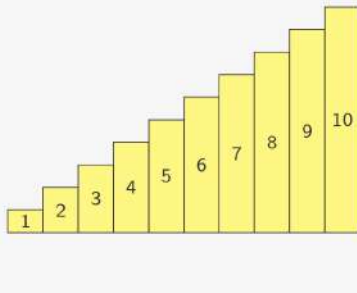


Ordenação por Inserção

Ideia:

- Se já temos $v[0], v[1], \dots, v[i-1]$ ordenado
- Inserimos $v[i]$ na posição correta
 - fazemos algo similar ao BubbleSort
- Ficamos com $v[0], v[1], \dots, v[i]$ ordenado

```
1 void insertionsort(int *v, int n) {  
2     int i, j;  
3     for (i = 1; i < n; i++)  
4         for (j = i; j > 0; j--)  
5             if (v[j] < v[j-1])  
6                 troca(&v[j], &v[j-1]);  
7 }
```



- 1 Complexidade de Algoritmos
- 2 Recursividade
- 3 Filas
- 4 Pilhas
- 5 Árvores
- 6 Ordenação
- 7 Bubble Sort
- 8 Insertion Sort
- 9 Métodos de busca**
- 10 Busca Binária
- 11 Questões

Métodos de Busca

Busca Linear

Busca Binária

Exemplos e Aplicações

Perguntas e Respostas

chave = 45

| | | | | | | | | | |
|----|---|----|----|----|----|---|----|----|----|
| 20 | 5 | 15 | 24 | 67 | 45 | 1 | 76 | 21 | 11 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

chave = 100

| | | | | | | | | | |
|----|---|----|----|----|----|---|----|----|----|
| 20 | 5 | 15 | 24 | 67 | 45 | 1 | 76 | 21 | 11 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

No primeiro exemplo, a função deve retornar 5, enquanto no segundo exemplo, a função deve retornar -1.

A busca sequencial é o algoritmo mais simples de busca:

Percorra a lista comparando a chave com os valores dos elementos em cada uma das posições.

Se a chave for igual a algum dos elementos, retorne a posição correspondente na lista.

Se a lista toda foi percorrida e a chave não for encontrada, retorne o valor -1 .

Exemplo de Código em C para Busca Sequencial:

```
1 #include <stdio.h>
2
3 int buscaSequencial(int lista[], int tamanho, int chave) {
4     for (int indice = 0; indice < tamanho; indice++) {
5         if (lista[indice] == chave) {
6             return indice;
7         }
8     }
9     return -1;
10 }
```

- 1 Complexidade de Algoritmos
- 2 Recursividade
- 3 Filas
- 4 Pilhas
- 5 Árvores
- 6 Ordenação
- 7 Bubble Sort
- 8 Insertion Sort
- 9 Métodos de busca
- 10 Busca Binária**
- 11 Questões

Visão Geral:

Algoritmo eficiente para listas ordenadas.

Compara a chave de busca com o elemento no meio da lista.

Processo:


Se a chave é igual ao meio, a posição é retornada.

Se a chave é menor, busca na metade inferior.

Se a chave é maior, busca na metade superior.

Repete o processo até encontrar a chave ou esgotar as opções (retorna -1 se não encontrar).

Chave = 15



| | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|-----|
| 1 | 5 | 15 | 20 | 24 | 45 | 67 | 76 | 78 | 100 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

$\text{pos_ini} = 0, \quad \text{pos_fim} = 9, \quad \text{pos_meio} = 4$

Como $\text{lista}[\text{pos_meio}] > \text{chave}$, devemos continuar a busca na primeira metade da região e, para isso, atualizamos a variável pos_fim .

Chave = 15




| | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|-----|
| 1 | 5 | 15 | 20 | 24 | 45 | 67 | 76 | 78 | 100 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

$\text{pos_ini} = 0, \quad \text{pos_fim} = 3, \quad \text{pos_meio} = 1$

Como $\text{lista}[\text{pos_meio}] < \text{chave}$, devemos continuar a busca na segunda metade da região e, para isso, atualizamos a variável pos_ini .

Chave = 15




| | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|-----|
| 1 | 5 | 15 | 20 | 24 | 45 | 67 | 76 | 78 | 100 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

$\text{pos_ini} = 2, \quad \text{pos_fim} = 3, \quad \text{pos_meio} = 2$

Finalmente, encontramos a chave ($\text{lista}[\text{pos_meio}] = \text{chave}$) e, sendo assim, devolvemos a sua posição na lista (pos_meio).

Chave = 50




| | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|-----|
| 1 | 5 | 15 | 20 | 24 | 45 | 67 | 76 | 78 | 100 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

$\text{pos_ini} = 0, \quad \text{pos_fim} = 9, \quad \text{pos_meio} = 4$

Como $\text{lista}[\text{pos_meio}] > \text{chave}$, devemos continuar a busca na primeira metade da região e, para isso, atualizamos a variável pos_fim .

Chave = 50




| | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|-----|
| 1 | 5 | 15 | 20 | 24 | 45 | 67 | 76 | 78 | 100 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

$\text{pos_ini} = 5, \quad \text{pos_fim} = 9, \quad \text{pos_meio} = 7$

Como $\text{lista}[\text{pos_meio}] < \text{chave}$, devemos continuar a busca na segunda metade da região e, para isso, atualizamos a variável pos_ini .

Chave = 50




| | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|-----|
| 1 | 5 | 15 | 20 | 24 | 45 | 67 | 76 | 78 | 100 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

$\text{pos_ini} = 5, \quad \text{pos_fim} = 6, \quad \text{pos_meio} = 5$

Como $\text{lista}[\text{pos_meio}] < \text{chave}$, devemos continuar a busca na segunda metade da região e, para isso, atualizamos a variável pos_ini .

Chave = 50



| | | | | | | | | | |
|---|---|----|----|----|----|----|----|----|-----|
| 1 | 5 | 15 | 20 | 24 | 45 | 67 | 76 | 78 | 100 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

$\text{pos_ini} = 6, \quad \text{pos_fim} = 6, \quad \text{pos_meio} = 6$

Como $\text{pos_ini} > \text{pos_fim}$, determinamos que a chave não está na lista e retornamos o valor -1.

- 1 Complexidade de Algoritmos
- 2 Recursividade
- 3 Filas
- 4 Pilhas
- 5 Árvores
- 6 Ordenação
- 7 Bubble Sort
- 8 Insertion Sort
- 9 Métodos de busca
- 10 Busca Binária
- 11 Questões**

Qual das seguintes afirmações melhor descreve uma vantagem das listas ligadas em comparação com os arrays?

- a) Listas ligadas permitem a realização de operações matemáticas complexas diretamente nos elementos.
- b) Listas ligadas facilitam a inserção e remoção de elementos em qualquer posição, sem necessidade de reorganizar todos os outros elementos.
- c) Listas ligadas oferecem melhor compressão de dados do que os arrays.
- d) Listas ligadas utilizam menos memória geral do que os arrays para armazenar a mesma quantidade de dados.
- e) Listas ligadas permitem a execução de código mais rápido durante a busca de elementos, pois não é necessário percorrer elemento a elemento.

Qual das seguintes afirmações melhor descreve uma vantagem das listas ligadas em comparação com os arrays?

- a) Listas ligadas permitem a realização de operações matemáticas complexas diretamente nos elementos.
- b) **Listas ligadas facilitam a inserção e remoção de elementos em qualquer posição, sem necessidade de reorganizar todos os outros elementos.**
- c) Listas ligadas oferecem melhor compressão de dados do que os arrays.
- d) Listas ligadas utilizam menos memória geral do que os arrays para armazenar a mesma quantidade de dados.
- e) Listas ligadas permitem a execução de código mais rápido durante a busca de elementos, pois não é necessário percorrer elemento a elemento.

Em qual das seguintes situações o uso de uma lista ligada é mais adequado do que um array?

- a) Quando o tamanho dos dados é conhecido e não muda.
- b) Quando há necessidade de acesso aleatório rápido aos elementos.
- c) Quando há muitas inserções e deleções de elementos em posições arbitrárias.
- d) Quando os dados precisam ser armazenados de forma contígua na memória.
- e) Quando a ordem dos elementos não é importante.

Em qual das seguintes situações o uso de uma lista ligada é mais adequado do que um array?

- a) Quando o tamanho dos dados é conhecido e não muda.
- b) Quando há necessidade de acesso aleatório rápido aos elementos.
- c) **Quando há muitas inserções e deleções de elementos em posições arbitrárias.**
- d) Quando os dados precisam ser armazenados de forma contígua na memória.
- e) Quando a ordem dos elementos não é importante.

Qual das seguintes operações não é típica de uma pilha?

- a) Push (Empilhar)
- b) Pop (Desempilhar)
- c) Peek (Espiar)
- d) Enqueue (Enfileirar)
- e) IsEmpty (EstáVazia)

Qual das seguintes operações não é típica de uma pilha?

- a) Push (Empilhar)
- b) Pop (Desempilhar)
- c) Peek (Espiar)
- d) **Enqueue (Enfileirar)**
- e) IsEmpty (EstáVazia)

Qual das seguintes operações não é típica de uma fila?

- a) Enqueue (Enfileirar)
- b) Dequeue (Desenfileirar)
- c) Front (Frente)
- d) Push (Empilhar)
- e) IsEmpty (EstáVazia)

Qual das seguintes operações não é típica de uma fila?

- a) Enqueue (Enfileirar)
- b) Dequeue (Desenfileirar)
- c) Front (Frente)
- d) **Push (Empilhar)**
- e) IsEmpty (EstáVazia)

Em uma fila, qual operação permite remover o elemento na frente da fila?

- a) Enqueue (Enfileirar)
- b) Dequeue (Desenfileirar)
- c) Peek (Espiar)
- d) Push (Empilhar)
- e) Pop (Desempilhar)

Em uma fila, qual operação permite remover o elemento na frente da fila?

- a) Enqueue (Enfileirar)
- b) **Dequeue (Desenfileirar)**
- c) Peek (Espiar)
- d) Push (Empilhar)
- e) Pop (Desempilhar)

Considere uma estrutura de dados que possui as seguintes características:

1. Os elementos são armazenados de forma hierárquica. 2. Cada elemento (nó) pode ter zero ou mais filhos. 3. A estrutura permite realizar buscas e operações de inserção de forma eficiente.

Com base nas características descritas, qual é a estrutura de dados mais adequada?

- a) Array.
- b) Fila.
- c) Árvore.
- d) Pilha.
- e) Lista Ligada.

Considere uma estrutura de dados que possui as seguintes características:

1. Os elementos são armazenados de forma hierárquica.
2. Cada elemento (nó) pode ter zero ou mais filhos.
3. A estrutura permite realizar buscas e operações de inserção de forma eficiente.

Com base nas características descritas, qual é a estrutura de dados mais adequada?

- a) Array.
- b) Fila.
- c) **Árvore.**
- d) Pilha.
- e) Lista Ligada.

Considere uma estrutura de dados que possui as seguintes características:

1. Os elementos seguem a ordem de chegada. 2. Permite operações de inserção na extremidade traseira e remoção na extremidade dianteira. 3. É útil para gerenciar processos em sistemas operacionais e filas de impressão, por exemplo.

Com base nas características descritas, qual é a estrutura de dados mais adequada?

- a) Árvore.
- b) Fila.
- c) Pilha.
- d) Array.
- e) Lista Ligada.

Considere uma estrutura de dados que possui as seguintes características:

1. Os elementos seguem a ordem de chegada. 2. Permite operações de inserção na extremidade traseira e remoção na extremidade dianteira. 3. É útil para gerenciar processos em sistemas operacionais e filas de impressão, por exemplo.

Com base nas características descritas, qual é a estrutura de dados mais adequada?

- a) Árvore.
- b) **Fila.**
- c) Pilha.
- d) Array.
- e) Lista Ligada.

```
1 // a) suponha que as funções
  enfileirar e desenfileirar
  estejam implementadas:
2
3 Fila* fila = criarFila();
4 enfileirar(fila, 5); // inserindo 5
5 enfileirar(fila, 10); // inserindo 10
6 enfileirar(fila, 15); // inserindo 15
7 enfileirar(fila, 20); // inserindo 20
8 desenfileirar(fila);
9 enfileirar(fila, 25);
10 desenfileirar(fila);
```

```
1 // suponha que as funções empilhar e
  desempilhar estejam
  implementadas:
2
3 Pilha* pilha = criarPilha();
4 empilhar(pilha, 5); // inserindo 5
5 empilhar(pilha, 10); // inserindo 10
6 empilhar(pilha, 15); // inserindo 15
7 empilhar(pilha, 20); // inserindo 20
8 desempilhar(pilha);
9 empilhar(pilha, 25);
10 desempilhar(pilha);
```

Considere as seguintes operações. Se os elementos 5, 10, 15 e 20 forem inseridos nessa ordem em uma fila e em uma pilha. Qual será a sequência final de remoção dos elementos restantes em cada estrutura?

- a) Fila: 15, 20, 25 e Pilha: 15, 10, 5.
- b) Fila: 15, 20, 25 e Pilha: 25, 15, 10.
- c) Fila: 10, 15, 20 e Pilha: 5, 10, 15.
- d) Fila: 20, 15, 10 e Pilha: 10, 15, 20.
- e) Fila: 25, 15, 10 e Pilha: 15, 20, 25.

```
1 // a) suponha que as funções
  enfileirar e desenfileirar
  estejam implementadas:
2
3 Fila* fila = criarFila();
4 enfileirar(fila, 5); // inserindo 5
5 enfileirar(fila, 10); // inserindo 10
6 enfileirar(fila, 15); // inserindo 15
7 enfileirar(fila, 20); // inserindo 20
8 desenfileirar(fila);
9 enfileirar(fila, 25);
10 desenfileirar(fila);
```

```
1 // suponha que as funções empilhar e
  desempilhar estejam
  implementadas:
2
3 Pilha* pilha = criarPilha();
4 empilhar(pilha, 5); // inserindo 5
5 empilhar(pilha, 10); // inserindo 10
6 empilhar(pilha, 15); // inserindo 15
7 empilhar(pilha, 20); // inserindo 20
8 desempilhar(pilha);
9 empilhar(pilha, 25);
10 desempilhar(pilha);
```

Considere as seguintes operações. Se os elementos 5, 10, 15 e 20 forem inseridos nessa ordem em uma fila e em uma pilha. Qual será a sequência final de remoção dos elementos restantes em cada estrutura?

- a) **Fila: 15, 20, 25 e Pilha: 15, 10, 5.**
- b) Fila: 15, 20, 25 e Pilha: 25, 15, 10.
- c) Fila: 10, 15, 20 e Pilha: 5, 10, 15.
- d) Fila: 20, 15, 10 e Pilha: 10, 15, 20.
- e) Fila: 25, 15, 10 e Pilha: 15, 20, 25.

```
1 // a) suponha que as funções
  enfileirar e desenfileirar
  estejam implementadas:
2
3 Fila* fila = criarFila();
4 enfileirar(fila, 'A');
5 enfileirar(fila, 'B');
6 enfileirar(fila, 'C');
7 enfileirar(fila, 'D');
8 desenfileirar(fila);
9 enfileirar(fila, 'E');
10 desenfileirar(fila);
```

```
1 // suponha que as funções empilhar e
  desempilhar estejam
  implementadas:
2 Pilha* pilha = criarPilha();
3 empilhar(pilha, 'A');
4 empilhar(pilha, 'B');
5 empilhar(pilha, 'C');
6 empilhar(pilha, 'D');
7 desempilhar(pilha);
8 empilhar(pilha, 'E');
9 desempilhar(pilha);
```

Considere as seguintes operações. Se os elementos 'A', 'B', 'C' e 'D' forem inseridos nessa ordem em uma fila e em uma pilha. Qual será a sequência final de remoção dos elementos restantes em cada estrutura?

- a) Fila: 'D', 'C', 'B' e Pilha: 'A', 'B', 'C'.
- b) Fila: 'C', 'D', 'E' e Pilha: 'E', 'C', 'B'.
- c) Fila: 'B', 'C', 'D' e Pilha: 'A', 'C', 'E'.
- d) Fila: 'C', 'D', 'E' e Pilha: 'C', 'B', 'A'.
- e) Fila: 'E', 'D', 'C' e Pilha: 'B', 'C', 'D'.

```
1 // a) suponha que as funções
  enfileirar e desenfileirar
  estejam implementadas:
2
3 Fila* fila = criarFila();
4 enfileirar(fila, 'A');
5 enfileirar(fila, 'B');
6 enfileirar(fila, 'C');
7 enfileirar(fila, 'D');
8 desenfileirar(fila);
9 enfileirar(fila, 'E');
10 desenfileirar(fila);
```

```
1 // suponha que as funções empilhar e
  desempilhar estejam
  implementadas:
2 Pilha* pilha = criarPilha();
3 empilhar(pilha, 'A');
4 empilhar(pilha, 'B');
5 empilhar(pilha, 'C');
6 empilhar(pilha, 'D');
7 desempilhar(pilha);
8 empilhar(pilha, 'E');
9 desempilhar(pilha);
```

Considere as seguintes operações. Se os elementos 'A', 'B', 'C' e 'D' forem inseridos nessa ordem em uma fila e em uma pilha. Qual será a sequência final de remoção dos elementos restantes em cada estrutura?

- a) Fila: 'D', 'C', 'B' e Pilha: 'A', 'B', 'C'.
- b) Fila: 'C', 'D', 'E' e Pilha: 'E', 'C', 'B'.
- c) Fila: 'B', 'C', 'D' e Pilha: 'A', 'C', 'E'.
- d) **Fila: 'C', 'D', 'E' e Pilha: 'C', 'B', 'A'.**
- e) Fila: 'E', 'D', 'C' e Pilha: 'B', 'C', 'D'.


```
1 // a) suponha que as funções
  enfileirar e desenfileirar
  estejam implementadas:
2
3 Fila* fila = criarFila();
4 enfileirar(fila, 100);
5 enfileirar(fila, 200);
6 enfileirar(fila, 300);
7 enfileirar(fila, 400);
8 desenfileirar(fila);
9 enfileirar(fila, 500);
10 desenfileirar(fila);
```

```
1 // suponha que as funções empilhar e
  desempilhar estejam
  implementadas:
2 Pilha* pilha = criarPilha();
3 empilhar(pilha, 100);
4 empilhar(pilha, 200);
5 empilhar(pilha, 300);
6 empilhar(pilha, 400);
7 desempilhar(pilha);
8 empilhar(pilha, 500);
9 desempilhar(pilha);
```

Considere as seguintes operações. Se os elementos 100, 200, 300 e 400 forem inseridos nessa ordem em uma fila e em uma pilha. Qual será a sequência final de remoção dos elementos restantes em cada estrutura?

- a) Fila: 300, 400, 500 e Pilha: 500, 300, 200.
- b) Fila: 200, 300, 400 e Pilha: 100, 300, 500.
- c) Fila: 400, 300, 200 e Pilha: 300, 200, 100.
- d) Fila: 500, 400, 300 e Pilha: 200, 300, 400.
- e) Fila: 300, 400, 500 e Pilha: 100, 200, 300.

```
1 // a) suponha que as funções
  enfileirar e desenfileirar
  estejam implementadas:
2
3 Fila* fila = criarFila();
4 enfileirar(fila, 100);
5 enfileirar(fila, 200);
6 enfileirar(fila, 300);
7 enfileirar(fila, 400);
8 desenfileirar(fila);
9 enfileirar(fila, 500);
10 desenfileirar(fila);
```

```
1 // suponha que as funções empilhar e
  desempilhar estejam
  implementadas:
2 Pilha* pilha = criarPilha();
3 empilhar(pilha, 100);
4 empilhar(pilha, 200);
5 empilhar(pilha, 300);
6 empilhar(pilha, 400);
7 desempilhar(pilha);
8 empilhar(pilha, 500);
9 desempilhar(pilha);
```

Considere as seguintes operações. Se os elementos 100, 200, 300 e 400 forem inseridos nessa ordem em uma fila e em uma pilha. Qual será a sequência final de remoção dos elementos restantes em cada estrutura?

- a) Fila: 300, 400, 500 e Pilha: 500, 300, 200.
- b) Fila: 200, 300, 400 e Pilha: 100, 300, 500.
- c) Fila: 400, 300, 200 e Pilha: 300, 200, 100.
- d) Fila: 500, 400, 300 e Pilha: 200, 300, 400.
- e) **Fila: 300, 400, 500 e Pilha: 100, 200, 300.**

```
1 // a) suponha que as funções
  enfileirar e desenfileirar
  estejam implementadas:
2
3 Fila* fila = criarFila();
4 enfileirar(fila, 1); // inserindo 1
5 enfileirar(fila, 2); // inserindo 2
6 enfileirar(fila, 3); // inserindo 3
7 enfileirar(fila, 4); // inserindo 4
8 desenfileirar(fila);
9 enfileirar(fila, 5);
10 desenfileirar(fila);
```

```
1 // suponha que as funções empilhar e
  desempilhar estejam
  implementadas:
2 Pilha* pilha = criarPilha();
3 empilhar(pilha, 1); // inserindo 1
4 empilhar(pilha, 2); // inserindo 2
5 empilhar(pilha, 3); // inserindo 3
6 empilhar(pilha, 4); // inserindo 4
7 desempilhar(pilha);
8 empilhar(pilha, 5);
9 desempilhar(pilha);
```

Considere as seguintes operações. Se os elementos 1, 2, 3 e 4 forem inseridos nessa ordem em uma fila e em uma pilha. Qual será a sequência final de remoção dos elementos restantes em cada estrutura?

- a) Fila: 3, 4, 5 e Pilha: 3, 2, 1.
- b) Fila: 4, 3, 2 e Pilha: 1, 2, 3.
- c) Fila: 3, 4, 5 e Pilha: 5, 3, 2.
- d) Fila: 2, 3, 4 e Pilha: 1, 3, 5.
- e) Fila: 5, 4, 3 e Pilha: 2, 3, 4.

```
1 // a) suponha que as funções
  enfileirar e desenfileirar
  estejam implementadas:
2
3 Fila* fila = criarFila();
4 enfileirar(fila, 1); // inserindo 1
5 enfileirar(fila, 2); // inserindo 2
6 enfileirar(fila, 3); // inserindo 3
7 enfileirar(fila, 4); // inserindo 4
8 desenfileirar(fila);
9 enfileirar(fila, 5);
10 desenfileirar(fila);
```

```
1 // suponha que as funções empilhar e
  desempilhar estejam
  implementadas:
2 Pilha* pilha = criarPilha();
3 empilhar(pilha, 1); // inserindo 1
4 empilhar(pilha, 2); // inserindo 2
5 empilhar(pilha, 3); // inserindo 3
6 empilhar(pilha, 4); // inserindo 4
7 desempilhar(pilha);
8 empilhar(pilha, 5);
9 desempilhar(pilha);
```

Considere as seguintes operações. Se os elementos 1, 2, 3 e 4 forem inseridos nessa ordem em uma fila e em uma pilha. Qual será a sequência final de remoção dos elementos restantes em cada estrutura?

- a) **Fila: 3, 4, 5 e Pilha: 3, 2, 1.**
- b) Fila: 4, 3, 2 e Pilha: 1, 2, 3.
- c) Fila: 3, 4, 5 e Pilha: 5, 3, 2.
- d) Fila: 2, 3, 4 e Pilha: 1, 3, 5.
- e) Fila: 5, 4, 3 e Pilha: 2, 3, 4.

```
1 // Operações com fila e pilha
2 Fila* fila = criarFila();
3 Pilha* pilha = criarPilha();
4
5 enfileirar(fila, 1);
6 enfileirar(fila, 2);
7 enfileirar(fila, 3);
8 enfileirar(fila, 4);
9 desenfileirar(fila);
10 empilhar(pilha, desenfileirar(fila));
11 empilhar(pilha, desenfileirar(fila));
12 empilhar(pilha, desenfileirar(fila));
13 empilhar(pilha, 5);
14 desempilhar(pilha);
```

Qual será a sequência final de remoção dos elementos restantes na pilha?

- a) 4, 3, 2.
- b) 3, 2, 4.
- c) 2, 3, 4.
- d) 5, 2, 3.
- e) 6, 3, 2.

```
1 // Operações com fila e pilha
2 Fila* fila = criarFila();
3 Pilha* pilha = criarPilha();
4
5 enfileirar(fila, 1);
6 enfileirar(fila, 2);
7 enfileirar(fila, 3);
8 enfileirar(fila, 4);
9 desenfileirar(fila);
10 empilhar(pilha, desenfileirar(fila));
11 empilhar(pilha, desenfileirar(fila));
12 empilhar(pilha, desenfileirar(fila));
13 empilhar(pilha, 5);
14 desempilhar(pilha);
```

Qual será a sequência final de remoção dos elementos restantes na pilha?

- a) **4, 3, 2.**
- b) 3, 2, 4.
- c) 2, 3, 4.
- d) 5, 2, 3.
- e) 6, 3, 2.

```
1 // Operações com fila e pilha
2 Fila* fila = criarFila();
3 Pilha* pilha = criarPilha();
4
5 enfileirar(fila, 10);
6 enfileirar(fila, 20);
7 enfileirar(fila, 30);
8 enfileirar(fila, 40);
9 desenfileirar(fila);
10 empilhar(pilha, desenfileirar(fila));
11 empilhar(pilha, desenfileirar(fila));
12 empilhar(pilha, desenfileirar(fila));
13 empilhar(pilha, 50);
14 desempilhar(pilha);
```

Qual será a sequência final de remoção dos elementos restantes na pilha?

- a) 40, 30, 20.
- b) 30, 20, 40.
- c) 20, 30, 40.
- d) 50, 20, 30.
- e) 60, 30, 20.

```
1 // Operações com fila e pilha
2 Fila* fila = criarFila();
3 Pilha* pilha = criarPilha();
4
5 enfileirar(fila, 10);
6 enfileirar(fila, 20);
7 enfileirar(fila, 30);
8 enfileirar(fila, 40);
9 desenfileirar(fila);
10 empilhar(pilha, desenfileirar(fila));
11 empilhar(pilha, desenfileirar(fila));
12 empilhar(pilha, desenfileirar(fila));
13 empilhar(pilha, 50);
14 desempilhar(pilha);
```

Qual será a sequência final de remoção dos elementos restantes na pilha?

- a) **40, 30, 20.**
- b) 30, 20, 40.
- c) 20, 30, 40.
- d) 50, 20, 30.
- e) 60, 30, 20.


```
1 // Operações com fila e pilha
2 Fila* fila = criarFila();
3 Pilha* pilha = criarPilha();
4
5 enfileirar(fila, 100);
6 enfileirar(fila, 200);
7 enfileirar(fila, 300);
8 enfileirar(fila, 400);
9 desenfileirar(fila);
10 empilhar(pilha, desenfileirar(fila));
11 empilhar(pilha, desenfileirar(fila));
12 empilhar(pilha, desenfileirar(fila));
13 empilhar(pilha, 500);
14 desempilhar(pilha);
```

Qual será a sequência final de remoção dos elementos restantes na pilha?

- a) 300, 200, 400.
- b) 400, 300, 200.
- c) 200, 300, 400.
- d) 500, 200, 300.
- e) 600, 300, 200.

```
1 // Operações com fila e pilha
2 Fila* fila = criarFila();
3 Pilha* pilha = criarPilha();
4
5 enfileirar(fila, 100);
6 enfileirar(fila, 200);
7 enfileirar(fila, 300);
8 enfileirar(fila, 400);
9 desenfileirar(fila);
10 empilhar(pilha, desenfileirar(fila));
11 empilhar(pilha, desenfileirar(fila));
12 empilhar(pilha, desenfileirar(fila));
13 empilhar(pilha, 500);
14 desempilhar(pilha);
```

Qual será a sequência final de remoção dos elementos restantes na pilha?

- a) 300, 200, 400.
- b) **400, 300, 200.**
- c) 200, 300, 400.
- d) 500, 200, 300.
- e) 600, 300, 200.

```
1 // Operações com fila e pilha
2 Fila* fila = criarFila();
3 Pilha* pilha = criarPilha();
4
5 enfileirar(fila, 'X');
6 enfileirar(fila, 'Y');
7 enfileirar(fila, 'Z');
8 enfileirar(fila, 'W');
9 desenfileirar(fila);
10 empilhar(pilha, desenfileirar(fila));
11 empilhar(pilha, desenfileirar(fila));
12 empilhar(pilha, desenfileirar(fila));
13 empilhar(pilha, 'V');
14 desempilhar(pilha);
```

Qual será a sequência final de remoção dos elementos restantes na pilha?

- a) 'W', 'Z', 'Y'.
- b) 'Z', 'Y', 'W'.
- c) 'Y', 'Z', 'W'.
- d) 'V', 'Y', 'Z'.
- e) 'U', 'Z', 'Y'.

```
1 // Operações com fila e pilha
2 Fila* fila = criarFila();
3 Pilha* pilha = criarPilha();
4
5 enfileirar(fila, 'X');
6 enfileirar(fila, 'Y');
7 enfileirar(fila, 'Z');
8 enfileirar(fila, 'W');
9 desenfileirar(fila);
10 empilhar(pilha, desenfileirar(fila));
11 empilhar(pilha, desenfileirar(fila));
12 empilhar(pilha, desenfileirar(fila));
13 empilhar(pilha, 'V');
14 desempilhar(pilha);
```

Qual será a sequência final de remoção dos elementos restantes na pilha?

- a) 'W', 'Z', 'Y'.
- b) 'Z', 'Y', 'W'.
- c) 'Y', 'Z', 'W'.
- d) 'V', 'Y', 'Z'.
- e) 'U', 'Z', 'Y'.

Função recursiva:

```
1 #include <stdio.h>
2
3 int funcao_recursiva(int n) {
4     if (n == 0) {
5         return 1;
6     } else {
7         return n * funcao_recursiva(n -
8             1);
9     }
}
```

Função iterativa:

```
1 int funcao_iterativa(int n)
2 {
3     int resultado = 1;
4     for (int i = 1; i <= n;
5         i++) {
6         resultado *= i;
7     }
8     return resultado;
9 }
```

As duas implementações produzem o mesmo resultado para o valor da variável numero igual a 3?

- a) Sim, ambas produzem o resultado 6.
- b) Não, a função recursiva produz 3 e a função iterativa produz 3.
- c) Não, a função recursiva produz 9 e a função iterativa produz 9.
- d) Não, a função recursiva produz 1 e a função iterativa produz 1.
- e) Não, a função recursiva produz 10 e a função iterativa produz 1.

Função recursiva:

```
1 #include <stdio.h>
2
3 int funcao_recursiva(int n) {
4     if (n == 0) {
5         return 1;
6     } else {
7         return n * funcao_recursiva(n -
8             1);
9     }
}
```

Função iterativa:

```
1 int funcao_iterativa(int n)
2 {
3     int resultado = 1;
4     for (int i = 1; i <= n;
5         i++) {
6         resultado *= i;
7     }
8     return resultado;
9 }
```

As duas implementações produzem o mesmo resultado para o valor da variável numero igual a 3?

- a) **Sim, ambas produzem o resultado 6.**
- b) Não, a função recursiva produz 3 e a função iterativa produz 3.
- c) Não, a função recursiva produz 9 e a função iterativa produz 9.
- d) Não, a função recursiva produz 1 e a função iterativa produz 1.
- e) Não, a função recursiva produz 10 e a função iterativa produz 1.

Considere os seguintes métodos de ordenação: *insertionSort* e *selectionSort*.

Dado o array {29, 10, 14, 37, 13}, qual será o array ordenado após a aplicação de cada um desses métodos?

- a) Para o *insertionSort*, o array ordenado será: 10, 13, 14, 29, 37. Para o *selectionSort*, o array ordenado será: 10, 13, 14, 29, 37.
- b) Para o *insertionSort*, o array ordenado será: 29, 10, 14, 37, 13. Para o *selectionSort*, o array ordenado será: 13, 14, 10, 29, 37.
- c) Para o *insertionSort*, o array ordenado será: 13, 10, 29, 14, 37. Para o *selectionSort*, o array ordenado será: 29, 13, 14, 10, 37.
- d) Para o *insertionSort*, o array ordenado será: 10, 29, 13, 14, 37. Para o *selectionSort*, o array ordenado será: 10, 14, 13, 29, 37.

Considere os seguintes métodos de ordenação: *insertionSort* e *selectionSort*.

Dado o array {29, 10, 14, 37, 13}, qual será o array ordenado após a aplicação de cada um desses métodos?

- a) **Para o *insertionSort*, o array ordenado será: 10, 13, 14, 29, 37. Para o *selectionSort*, o array ordenado será: 10, 13, 14, 29, 37.**
- b) Para o *insertionSort*, o array ordenado será: 29, 10, 14, 37, 13. Para o *selectionSort*, o array ordenado será: 13, 14, 10, 29, 37.
- c) Para o *insertionSort*, o array ordenado será: 13, 10, 29, 14, 37. Para o *selectionSort*, o array ordenado será: 29, 13, 14, 10, 37.
- d) Para o *insertionSort*, o array ordenado será: 10, 29, 13, 14, 37. Para o *selectionSort*, o array ordenado será: 10, 14, 13, 29, 37.

Considere o array $\{15, 8, 42, 4, 23, 16\}$. Qual é a quantidade de comparações feitas para encontrar o elemento 23 utilizando o algoritmo de busca linear ?

- a) 1
- b) 2
- c) 3
- d) 4
- e) 5
- f) 6

Considere o array $\{15, 8, 42, 4, 23, 16\}$. Qual é a quantidade de comparações feitas para encontrar o elemento 23 utilizando o algoritmo de busca linear ?

- a) 1
- b) 2
- c) 3
- d) 4
- e) **5**
- f) 6

Considere o array $\{3, 9, 27, 81, 243\}$. Qual é a quantidade de comparações feitas para encontrar o elemento 81 utilizando o algoritmo de busca linear ?

- a) 1
- b) 2
- c) 3
- d) 4
- e) 5

Considere o array $\{3, 9, 27, 81, 243\}$. Qual é a quantidade de comparações feitas para encontrar o elemento 81 utilizando o algoritmo de busca linear ?

- a) 1
- b) 2
- c) 3
- d) 4
- e) 5

Considere o array $\{18, 5, 12, 7, 25, 9\}$. Qual é a quantidade de comparações feitas para encontrar o elemento 7 utilizando o algoritmo de busca linear ?

- a) 1
- b) 2
- c) 3
- d) 4
- e) 5

Considere o array $\{18, 5, 12, 7, 25, 9\}$. Qual é a quantidade de comparações feitas para encontrar o elemento 7 utilizando o algoritmo de busca linear ?

- a) 1
- b) 2
- c) 3
- d) 4
- e) 5

Considere o código em C que implementa a busca linear em um *array*:

```
1 #include <stdio.h>
2
3 int main() {
4     int array[] = {8, 19, 24, 33, 45, 50};
5     int x = 33;
6     int n = sizeof(array) / sizeof(array[0]);
7     for (int i = 0; i < n; i++) {
8         if (array[i] == x) {
9             printf("Elemento encontrado na posição:
10                 %d\n", i);
11             break;
12         }
13     }
14     return 0;
15 }
```

Assinale a alternativa CORRETA:

- a) V - V - F - V.
- b) F - V - F - F.
- c) V - F - V - V.
- d) F - F - V - F.
- e) V - V - V - F.

Classifique V para as sentenças verdadeiras e F para as falsas:

- ☐ () O código realiza 3 comparações para encontrar o elemento 33.
- ☐ () O código imprime a posição do elemento 33 no *array*.
- ☐ () O algoritmo utilizado é o de busca binária.
- ☐ () A condição de parada do laço é quando o elemento é encontrado ou o *array* é totalmente percorrido.

Considere o código em C que implementa a busca linear em um *array*:

```
1 #include <stdio.h>
2
3 int main() {
4     int array[] = {8, 19, 24, 33, 45, 50};
5     int x = 33;
6     int n = sizeof(array) / sizeof(array[0]);
7     for (int i = 0; i < n; i++) {
8         if (array[i] == x) {
9             printf("Elemento encontrado na posição:
10                 %d\n", i);
11             break;
12         }
13     }
14     return 0;
15 }
```

Assinale a alternativa CORRETA:

- a) **V - V - F - V.**
- b) F - V - F - F.
- c) V - F - V - V.
- d) F - F - V - F.
- e) V - V - V - F.

Classifique V para as sentenças verdadeiras e F para as falsas:

- () O código realiza 3 comparações para encontrar o elemento 33.
- () O código imprime a posição do elemento 33 no *array*.
- () O algoritmo utilizado é o de busca binária.
- () A condição de parada do laço é quando o elemento é encontrado ou o *array* é totalmente percorrido.

Considere o código em C que implementa a busca linear em um *array*:

```
1 #include <stdio.h>
2
3 int main() {
4     int array[] = {5, 10, 15, 20, 25, 30};
5     int x = 20;
6     int n = sizeof(array) / sizeof(array[0]);
7     for (int i = 0; i < n; i++) {
8         if (array[i] == x) {
9             printf("Elemento encontrado na posição:
10                 %d\n", i);
11             break;
12         }
13     }
14     return 0;
15 }
```

Assinale a alternativa CORRETA:

- a) V - V - F - V.
- b) F - V - F - F.
- c) V - F - V - V.
- d) F - F - V - F.
- e) V - V - V - F.

Classifique V para as sentenças verdadeiras e F para as falsas:

- () O código realiza 4 comparações para encontrar o elemento 20.
- () O código imprime a posição do elemento 20 no *array*.
- () O algoritmo utilizado é o de busca binária.
- () A condição de parada do laço é quando o elemento é encontrado ou o *array* é totalmente percorrido.

Considere o código em C que implementa a busca linear em um *array*:

```
1 #include <stdio.h>
2
3 int main() {
4     int array[] = {5, 10, 15, 20, 25, 30};
5     int x = 20;
6     int n = sizeof(array) / sizeof(array[0]);
7     for (int i = 0; i < n; i++) {
8         if (array[i] == x) {
9             printf("Elemento encontrado na posição:
10                 %d\n", i);
11             break;
12         }
13     }
14     return 0;
15 }
```

Assinale a alternativa CORRETA:

- a) **V - V - F - V.**
- b) F - V - F - F.
- c) V - F - V - V.
- d) F - F - V - F.
- e) V - V - V - F.

Classifique V para as sentenças verdadeiras e F para as falsas:

- ☐ () O código realiza 4 comparações para encontrar o elemento 20.
- ☐ () O código imprime a posição do elemento 20 no *array*.
- ☐ () O algoritmo utilizado é o de busca binária.
- ☐ () A condição de parada do laço é quando o elemento é encontrado ou o *array* é totalmente percorrido.