***EXERCÍCIO DE APROFUNDAMENTO - TEORIA***

lógico buscar(L, ctd)

Entrada: lista L, conteúdo ctd a ser buscado

Saída: sucesso ou falha na operação

temp ← L.prim θ(1)

enquanto temp ≠ NULO faça O(n) ou Ω(1)

se temp.conteudo = ctd então O(n) ou Ω(1)

retorna verdadeiro 1

temp ← temp.prox O(n) ou Ω(1)

retorna falso θ(1)

Resposta:

No geral a sub-rotina é O(n) quando a lista não for vazia (pois a estrutura de repetição poderá percorrer toda a lista) ou Ω(1) quando a lista for vazia, pois executará uma vez cada comando do código.

lógico inserir\_inicio(L, e)

Entrada: lista L, elemento e a ser inserido

Saída: sucesso ou falha na operação

cel: célula a ser inserida

se invalido(ctd) então θ(1)

retorna falso

cel.conteudo ← ctd

cel.prox ← L.prim

L.prim ← cel θ(1)

L.qtde ← L.qtde + 1

retorna verdadeiro

Resposta:

No geral essa sub-rotina executará de uma forma constante, não havendo um pior ou melhor caso, sendo θ(1)

lógico inserir\_meio(L, ctd, k)

Entrada: lista L, conteúdo ctd, k-ésima posição

Saída: sucesso ou falha na operação

cel: célula a ser inserida

se invalido(ctd) ou invalido(k) então θ(1)

retorna falso

cel.conteudo ← ctd θ(1)

se k = 1 então

cel.prox ← L.prim θ(1)

L.prim ← cel

senão θ(1)

temp ← L.prim

para i ← 1 até k-2 passo 1 faça O(n - 2) = O(n) ou Ω(1)

temp ← temp.prox

cel.prox ← temp.prox θ(1)

temp.prox ← cel θ(1)

L.qtde ← L.qtde + 1 θ(1)

retorna verdadeiro θ(1)

Resposta: No geral a sub-rotina será O(n) no pior caso e θ(1) no melhor caso.

lógico inserir\_fim(L, ctd)

Entrada: lista L, conteúdo ctd a inserir

Saída: sucesso ou falha na operação

cel: célula a ser inserida

se invalido(ctd) então θ(1)

retorna falso

cel.conteudo ← ctd θ(1)

cel.prox ← NULO

se esta\_vazia(L) então θ(1)

L.prim ← cel

senão

temp ← L.prim θ(1)

enquanto temp.prox ≠ NULO faça O(n) ou Ω(1)

temp ← temp.prox

temp.prox ← cel θ(1)

L.qtde ← L.qtde + 1 θ(1)

retorna verdadeiro

Resposta: Se a lista tiver foi vazia ela não executará a estrutura de repetição, tornando-a Ω(1), caso contrário o sistema de repetição será executado tornando o pior caso da sub-rotina em O(n).

lógico remover\_inicio(L)

Entrada: lista L

Saída: sucesso ou falha na operação

se vazia(L) então θ(1)

retorna falso

L.prim ← L.prim.prox

L.qtde ← L.qtde - 1 θ(1)

retorna verdadeiro

Resposta: Independente de qualquer coisa, o código terá a mesma complexidade θ(1).

lógico remover\_meio(L, k)

Entrada: lista L, k-ésima posição

Saída: sucesso ou falha na operação

se vazia(L) ou invalido(k) então

retorna falso

temp ← L.prim

se L.qtde = 1 então θ(1)

L.prim ← NULO

senão

para i ← 1 até k-2 passo 1 faça O(n) ou Ω(1)

temp ← temp.prox

temp.prox ← temp.prox.prox θ(1)

L.qtde ← L.qtde - 1

retorna verdadeiro θ(1)

Resposta: Se a lista tiver até 3 posições, o sistema repetição não será executado, tornando-a (em seu melhor caso) Ω(1), caso contrário será O(n) e o sistema de repetição irá percorrer a lista inteira.

lógico remover\_fim(L)

Entrada: lista L

Saída: sucesso ou falha na operação

se vazia(L) então θ(1)

retorna falso

se L.qtde = 1 então θ(1)

L.prim ← NULO

senão

temp ← L.prim O(1)

enquanto temp.prox.prox ≠ NULO faça O(n) ou Ω(1)

temp ← temp.prox

temp.prox ← NULO Ω(1)

L.qtde ← L.qtde – 1 θ(1)

retorna verdadeiro

Resposta: Se a lista tiver pelo menos 3 quantidades o sistema de repetição será acionado tornando a lista O(n), caso a lista seja vazia ou tenha menos de 3 quantidade será (em seu melhor caso) Ω(1).

***EXERCÍCIO DE REVISÃO - PRÁTICA***

#include <bits/stdc++.h>

#include <chrono>

#include “funcoes.h”

using namespace std;

using namespace std:: chrono;

void exibir(short \*v, int n); // vetor de inteiros (inteiros de 2 bytes)

void preencher(short \*v, int n, short min, short max);

void ordenar(short \*v, int n);

const int N = 1000000;

int main() {

srand(time(NULL));

vetor[N];

preencher(vetor, N, 10, 20);

auto inicio = steady\_clock::now();

ordenar(vetor, N); // implemente a ordenação por flutuação

auto fim = steady\_clock::now();

auto intervalo = fim - inicio;

cout << duration\_cast<seconds>(intervalo).count() << "s" << endl;

cout << duration\_cast< milliseconds >(intervalo).count() << "ms" << endl;

cout << duration\_cast<microseconds>(intervalo).count() << "us" << endl;

cout << duration\_cast< nanoseconds >(intervalo).count() << "ns" << endl;

cout << duration<double>{intervalo}.count() << "s" << endl;

return 0;

}

***EXERCÍCIO DE APROFUNDAMENTO - PRÁTICA***

void ordenacao(short \*v, int tam) {

int i, j, chave; θ(1)

for(i = 1; i < tam; i++) { O(n)

chave = v[i];

j = i - 1;

while(j >= 0 && v[j] > chave){ O(n) ou Ω(1)

v[j + 1] = v[j];

j--;

}

v[j + 1] = chave; O(n)

}

}

R: Neste caso, se o vetor estiver ordenado, somente um sistema de repetição irá executar o tamanho do vetor, tornando o nível de complexidade Ω(n), caso contrário executara os dois sistemas de repetição, tornando o nível de complexidade O(n²).