1. Elabore um programa que faça uma busca binária recursiva num vetor com 20 elementos.

#include <iostream>

using namespace std;

void buscaBinaria(int *vetor*[], int *num*, int *inicio*, int *fim*);

void listarVetor(int *vetor*[], int *n*);

main(){

    int num, n = 20;

    int vetor[n] = {1, 5, 16, 20, 24, 25, 29, 30, 33, 36, 37, 40, 53, 57, 61, 63, 65, 67, 69, 74};

    listarVetor(vetor, n);

    cout << "\n\nNumero a ser buscado: ";

    cin >> num;

    buscaBinaria(vetor, num, 0, n);

}

void buscaBinaria(int *vetor*[], int *num*, int *inicio*, int *fim*){

    int meio = (*inicio* + *fim*) / 2;

    if (*inicio* < *fim*){

        if (*num* == *vetor*[meio]){

            cout << "O numero " << *num* << " esta na posicao: " << meio + 1;

        }

        else{

            if (*num* < *vetor*[meio]){

                buscaBinaria(*vetor*, *num*, *inicio*, meio);

            }

            else{

                buscaBinaria(*vetor*, *num*, meio + 1, *fim*);

            }

        }

    }

    else{

        cout << "Numero nao encontrado na lista";

    }

}

void listarVetor(int *vetor*[], int *n*){

    for (int i = 0; i < *n*; i++){

        cout << *vetor*[i] << " ";

    }

}

2. Faça um programa crie uma lista encadeada (pode ser simplesmente ou duplamente) que implemente as funções:

a. inserir() – o usuário deve fornecer o valor do elemento que será inserido na lista encadeada. Informar o endereço de memória onde o elemento foi armazenado.

b. retirar() – o usuário deve fornecer o elemento a ser retirado. Mostrar mensagem caso elemento não seja encontrado.

c. crescente() – listagem dos elementos em ordem crescente.

d. decrescente() – listagem dos elementos em ordem decrescente.

Quais são as complexidade de pior caso das funções implementadas por você? Cite e explique.

#include <iostream>

using namespace std;

struct *no*{

    int info;

    struct *no*\* prox;

};

typedef struct *no*\* *noPtr*;

*noPtr* topo = NULL;

int menu();

bool isListaVazia();

void inserir();

void retirar();

int length();

void crescente();

void decrescente();

main(){

    int op;

    do{

        op = menu();

        switch(op){

            case 1: inserir();

            break;

            case 2: retirar();

            break;

            case 3: crescente();

            break;

            case 4: decrescente();

            break;

        }

    } while(op != 0);

}

int menu() {

 int op;

 cout << "\nEscolha o numero da operacao: 1 - Inserir; 2 - Retirar; 3 - Listar de forma crescente; 4- Listar de forma decrescente; 0 - Sair" << endl;

 cin >> op;

 if(op > 4){

     cout << "Opcao invalida" << endl;

 }

 return op;

}

bool isListaVazia(){

    return (topo == NULL);

}

void inserir(){

*noPtr* aux, ant, p = **new** *no*;

    cout << "Digite o valor do elemento: ";

    cin >> p -> info;

    if(isListaVazia()){

        p->prox = topo;

        topo = p;

    } else {

        aux = topo;

        while(aux != NULL && p->info > aux->info){

            ant = aux;

            aux = aux->prox;

        }

        if (aux == topo){

            p->prox = aux;

            topo = p;

        }else {

            p->prox = aux;

            ant->prox = p;

        }

    }

    cout << "Elemento inserido no endereco de memoria: " << p;

}

void retirar(){

*noPtr* ant, aux = topo;

    int valor;

    bool achei = false;

    if(!isListaVazia()){

        cout << "Escolha um elemento para ser retirado: ";

        cin >> valor;

        if(valor == aux->info){

            topo = aux->prox;

**delete**(aux);

        } else{

            while(aux != NULL){

                ant = aux;

                aux = aux -> prox;

                if(valor == aux->info){

                    achei = true;

                    break;

                }

        }

        if(achei){

            ant->prox = aux->prox;

**delete**(aux);

        } else {

            cout << "Elemento nao se encontra na lista ordenada";

            }

         }

    } else {

        cout << "\nLista ordenada se encontra vazia" << endl;

    }

}

int length(){

*noPtr* aux = topo;

    int contador = 0;

        while(aux != NULL) {

            aux = aux->prox;

            contador++;

        }

    return contador;

}

void crescente(){

*noPtr* aux = topo;

    while(aux != NULL) {

        cout << "info: " << aux->info << endl;

        aux = aux->prox;

    }

    cout << "tamanho da lista ordenada: " << length() << endl;

}

void decrescente(){

*noPtr* aux = topo;

    int vetor[length()];

    for(int i = 0; i < length(); i++){

        vetor[i] = aux->info;

        aux = aux->prox;

    }

    for(int i = length() - 1; i >= 0; i--){

        cout << "info: " << vetor[i] << endl;

    }

    cout << "tamanho da lista ordenada: " << length() << endl;

}

Todas as funções implementadas são O(n), por não possuírem laços aninhados, seu tempo de iteração irá variar de acordo com N.

Inserir(): A função itera a lista encadeada ordenada uma única vez e encontra a posição correta onde o elemento será inserido, tempo de iteração proporcional ao N.

Retirar(): O princípio é o mesmo da função inserir(), porém removendo o elemento e refazendo as ligações dos elementos.

Crescente(): Itera a lista uma única vez e imprime os elementos, tempo de iteração varia de acordo com o N.

Decrescente(): Por mais que existam dois “for” na função, eles não estão aninhados, logo, as complexidades são somadas e não multiplicadas, resultando em uma função O(2n). Como analisamos o pior caso, jogando o N para infinito, o 2 se torna indiferente, resultando em uma complexidade O(n).

3. Time sharing é um termo usado em Sistemas Operacionais (SO) e consiste, basicamente, em dedicar tempo de forma alternada entre diferentes processos. Como o tempo para cada tarefa é bem pequeno, o usuário tem a impressão que todos os processos estão sendo executados simultaneamente (multitasking). Para simular a ideia de time sharing, crie as seguintes funções:

a. Inclua novas requisições de processos numa lista simplesmente encadeada. Esses processos aguardam para serem executados sequencialmente.

b. Retire o processo com maior tempo de espera na lista, ou seja, que entrou primeiro.

Utilize como base a struct:

struct processo {

int cod;

struct processo\* prox;

};

#include <iostream>

using namespace std;

struct *processo* {

    int cod;

    struct *processo*\* prox;

};

typedef struct *processo*\* *processoPtr*;

*processoPtr* topo = NULL, ultimo = NULL;

int menu();

void incluirProcesso();

void retirarProcesso();

bool isSemProcesso();

void listar();

main(){

    int op;

do{

    op = menu();

    switch(op){

        case 1: incluirProcesso();

        break;

        case 2: retirarProcesso();

        break;

        case 3: listar();

        break;

    }

} while(op != 0);

}

void incluirProcesso(){

*processoPtr* p = **new** *processo*;

    cout << "\nDigite o codigo do processo a ser inserido: ";

    cin >> p->cod;

    p->prox = NULL;

    if(isSemProcesso()){

        topo = p;

        ultimo = p;

    } else {

        ultimo->prox = p;

        ultimo = p;

    }

    listar();

}

void retirarProcesso(){

*processoPtr* p;

    if(isSemProcesso()){

        cout << "\nNenhum processo em execucao!";

    } else {

        p = topo;

        topo = p->prox;

        cout << "\nO processo de codigo " << p->cod << " foi retirado." << endl;

**delete** p;

    }

}

void listar(){

*processoPtr* p;

    p = topo;

    int contador = 0;

    while(p != NULL) {

        cout << "Codigo do processo " << contador + 1 <<": " << p->cod << endl;

        p = p->prox;

        contador++;

    }

    if(!isSemProcesso()){

        cout << "Processos aguardando para serem executados..." << endl;

        cout << "Quantidade de processos: " << contador << endl;

    }

}

int menu() {

 int op;

 cout << "\nEscolha o numero da operacao: 1 - Incluir processo; 2 - Retirar processo; 0 - Sair" << endl;

 cin >> op;

 if(op > 2){

     cout << "Opcao invalida" << endl;

 }

 return op;

}

bool isSemProcesso(){

    return (topo == NULL);

}