RELATÓRIO DE RESOLUÇÃO DO TRABALHO 2 CORRESPONDENTE A SEGUNDA AVALIAÇÃO DA DISCIPLINA DE ESTRUTURAS DE DADOS II

Vinícius de Sousa Carvalho
viniciussccarvalho@ufpi.edu.br
Estruturas de Dados II - Juliana Oliveira de Carvalho

Resumo

Este relatório tem como principal fundamento a resolução das questões da segunda atividade avaliativa da disciplina de Estrutura de dados II, visando apresentar todos os aspectos necessários para o entendimento do problema, bem como sua solução. Todas as questões foram resolvidas com uso da linguagem de programação C, que alinhado à estrutura de dados da árvore 2-3, pode proporcionar a criação de programas capazes de solucionar as questões propostas.

Palavra-chave: Árvore 2-3, Estruturas de Dados.

Introdução

Estruturas de dados são formas organizadas de armazenar e manipular dados em um programa de computador. Essas estruturas são usadas para melhorar a eficiência do programa, tornando-o capaz de armazenar e acessar dados de maneira mais rápida e eficiente.

Uma árvore 2-3 é uma estrutura de dados em que cada nó pode ter até três filhos e pode ser considerada uma extensão da árvore binária. Cada nó de uma árvore 2-3 contém uma ou duas chaves de dados, que são usadas para armazenar e buscar informações. As árvores 2-3 são usadas para armazenar dados em memória principal, especialmente em bancos de dados, onde as operações de inserção e remoção são frequentes e precisam ser feitas de forma rápida e eficiente. Uma árvore 2-3 é capaz de manter seus dados sempre ordenados e balanceados, o que ajuda a manter o tempo de acesso aos dados constante.

Os tópicos seguintes deste trabalho estão divididos de modo a apresentar as questões propostas, bem como a lógica necessária para a resolução da mesma, exemplos de execução dos programas, testes de eficiência de busca e alteração na árvore 2-3, conclusão dos resultados alcançados e apêndice com todos os algoritmos utilizados.

Hardware utilizado

Todos os testes a seguir foram feitos utilizando o mesmo hardware, com as seguintes características:

Marca	Samsung
Sistema Operacional	Linux Ubuntu 22.04
Processador	Intel i3 7º geração
Memória RAM	4GB
Tipo de memória	DDR4 2400MHz

Seções Específicas

Os tópicos seguintes apresentam os enunciados das questões, acompanhados de sua resolução e testes de velocidade. Isso permitirá que os leitores compreendam a lógica por trás da solução e avaliem a eficiência do método utilizado.

Questão 01 e Questão 02

A questão 1 propõe um algoritmo que simula o processo de alocação e desalocação de memória do computador. Para a criação do algoritmo, deve ser levado em consideração uma memória de tamanho x, que será informada pelo usuário, bem como o seu estado. Ou seja, para uma memória de 36 bytes, o usuário deverá informar cada byte livre e ocupado, de modo que os estados da memória estejam intercalados, ou seja, se um bloco é livre, o bloco seguinte deverá ser ocupado. Após informado o estado inicial de toda a memória, o programa deve possibilitar duas funcionalidades: alocar blocos livres e desalocar blocos ocupados. para alocação, o usuário informa quantos blocos devem ser alocados e o sistema se encarrega de encontrar um nó na árvore da memória que tenha disponibilidade para alocar aquela espaço de memória requisitado; caso não haja memória suficiente, será informado ao usuário que não possível alocar. Para o processo de desalocação, o usuário deve informar o endereço do intervalo de blocos a ser removido, e o algoritmo se encarregará de encontrar, na árvore 2-3, o nó que possui esse nó e liberar os blocos informados. Cabe ressaltar a possibilidade do usuário solicitar tanto a alocação, quanto a desalocação de blocos de memória completos, bem como parciais, com resto de memória. Para ambos os casos, o sistema deve cuidar de redimensionar os blocos de memória entre os nó para garantir que mantenha sempre os estados das memória intercalados, sem que haja dois blocos livres juntos, e vice-versa. Após garantido as funcionalidades do sistema, deve ser feito um teste de desempenho do algoritmo, testando sua velocidade de busca e alteração de um bloco de memória na árvore, considerando diferentes níveis da árvore e executando múltiplos testes, a fim de reduzir variações da máquina.

Para solucionar o problema, primeiro é solicitado ao usuário o tamanho da memória que será trabalhada. Após informado, é solicitado o estado do primeiro bloco de memória, se é um bloco livre ou ocupado. Em seguida, é feita a separação dos blocos, onde o usuário informará o endereço final de cada bloco de memória até que toda a memória seja preenchida. O sistema se encarrega de inverter o estado dos blocos na inserção, já que, se informado livre no primeiro bloco, consequentemente, o próximo bloco é um bloco de memória ocupado. Para cada momento que o usuário informa a memória de um bloco, é feita a inserção de um novo nó na árvore, onde cada nó contém: a quantidade de informações daquele nó, referências para seus filhos da esquerda, centro e direita e a duas informações do nó, onde cada informação é uma struct que contém 3 informação: o endereço inicial do bloco, o endereço final do bloco e seu estado (L para livre e O para ocupado). Feito o preenchimento inicial de toda a memória, é disponibilizado para o usuário o menu de funcionalidades onde o usuário poderá alocar um espaço de memória livre, desalocar um espaço de memória ocupado e buscar um espaço de memória, verificando o tempo de busca para encontrar o bloco. A figura 01 e figura 02 apresenta, respectivamente, uma representação da memória em forma de bloco/matriz e sua construção em uma árvore 2-3.

Figura 01 - Representação da memória em forma de bloco.

Figura 02 - Representação da memória na árvore 2-3.

Ao selecionar a opção de alocação de memória, o usuário informará quantos blocos ele quer alocar e será chamado a função responsável por executar a alocação. Essa, por sua vez, buscará, na ordem da memória, o primeiro bloco livre que possui a quantidade de memória igual ou maior à informada pelo usuário. Caso seja todo o bloco, todo o bloco se tornará ocupado. Cabe ressaltar que é necessário analisar os blocos adjacentes ao bloco modificado, já que, o estado anterior deste bloco era livre, então seus vizinhos são blocos ocupados. Dessa forma, é necessário combinar esses 3 blocos formando apenas 1 com o endereço de memória total dos 3 blocos. Caso a inserção seja parcial, é feito o redimensionamento, onde haverá o aumento do bloco anterior ao analisado e diminuição do bloco modificado. Para esses casos, ao ser executado em uma árvore 2-3, deve ser levado em consideração as seguintes condições:

caso a alocação ocorra no início da memória, não existe vizinho anterior, portanto, a análise do vizinho só é feita com o próximo bloco;

caso a alocação seja feita no final, não existe o próximo bloco de memória, portanto, a análise só é feita com o vizinho anterior;

caso a alocação seja feita no meio da memória, existe tanto o vizinho anterior, quanto o próximo. Ambos devem ser levados em consideração para manter a estrutura da memória correta.

Para efetuar a identificação do próximo bloco, bem como do bloco anterior, existe um conjunto de possibilidades que devem ser analisadas de acordo com suas posições dentro da árvore 2-3. Por exemplo, para um bloco localizado na info1 de uma folha que tem 2 infos e pai e é filho do centro, seu vizinhos serão a info2 e a info1 do pai. Para um bloco localizado na Info1 de uma folha que tem 1 informação, tem pai é filho do centro, seus vizinhos serão a info1 e info2 do pai.

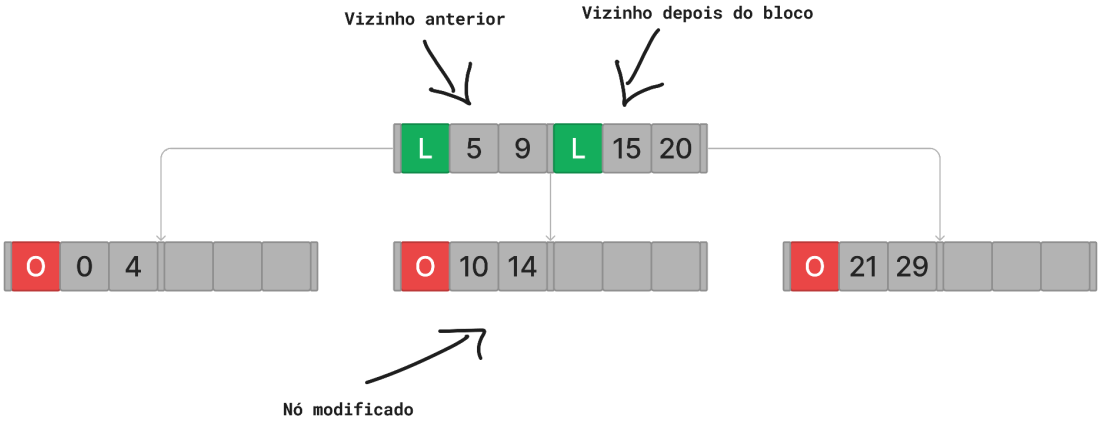

Figura 03 - Vizinho da info1 de um nó folha que possui pai e tem 1 informação.

Para blocos que não são folhas, seus vizinhos serão a maior informação do seu filho anterior a informação e a menor informação do seu filho seguinte a informação. Por exemplo, para um bloco situado na info1 de um nó que não é folha, seu vizinho anterior é a maior informação da subárvore do seu filho da esquerda e seu próximo vizinho é a menor informação da subárvore do centro. Já para um bloco situado na info2 de um nó que não é folha, seu vizinho anterior é a maior informação da subárvore do seu filho do centro e seu próximo vizinho é a menor informação da subárvore da direita. Dessa forma é possível encontrar os vizinhos e combiná-los, removendo-os da árvore e inserindo um novo nó, contendo o espaço de memória total dos 3 blocos removidos.

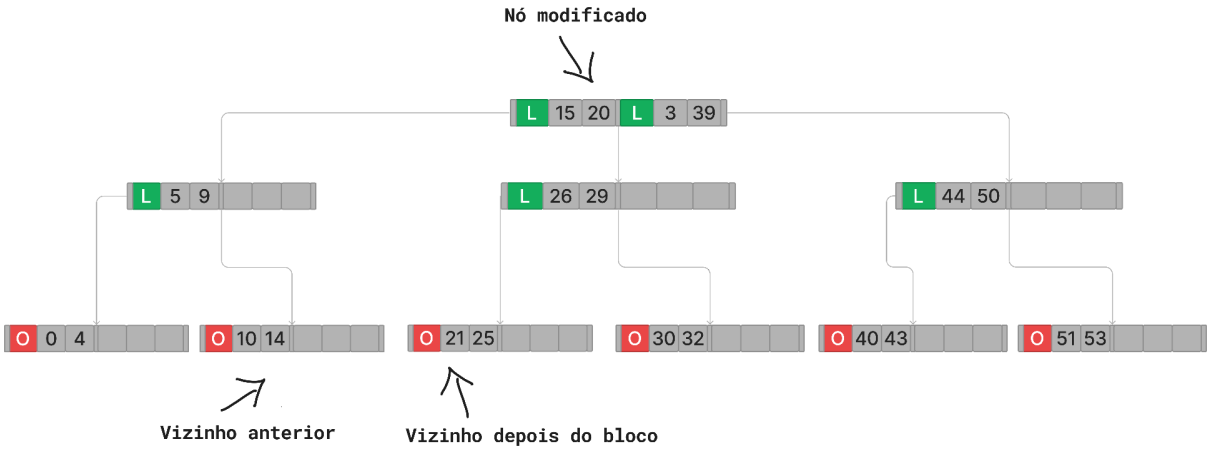

Figura 04 - Vizinho da info1 de um nó que não é folha.

Para a opção de deslocamento, o usuário informará o endereço inicial e final do intervalo que está ocupado e o sistema deverá liberar aquele espaço de memória. Cabe ressaltar, que para esse caso, também pode ser liberado um bloco de memória completo, como também parcialmente, necessitando combinar com os blocos vizinhos. Dessa forma o processo de identificação dos vizinhos segue a mesma lógica, apresentada anteriormente na etapa de alocação.

Por fim, para a opção de busca, o usuário informará o endereço inicial e final do intervalo do bloco a ser buscado e a função recursivamente buscará o bloco na árvore. Caso não encontre, uma mensagem informando será impressa na tela; caso encontre, será impresso tanto a informação de encontrada, quanto o tempo de execução em microssegundos foi necessário para encontrar a informação.

Exemplo de uso

Quantidade de memória disponível em megabyte:
30
Qual o estado do primeiro bloco(L/O)?
O
Bloco iniciando na posição 0. Informe a posição do fim do bloco de memoria:
4
Bloco iniciando na posição 5. Informe a posição do fim do bloco de memoria:
9
Bloco iniciando na posição 10. Informe a posição do fim do bloco de memoria:
14
Bloco iniciando na posição 15. Informe a posição do fim do bloco de memoria:
20
Bloco iniciando na posição 21. Informe a posição do fim do bloco de memoria:
29
O que deseja fazer? **1 - Alocar blocos de memória** **2 - Liberar Blocos de memória** **3 - Visualizar memória** **4 - Buscar bloco** **0 - Encerrar programa**
1

Quantos blocos deseja alocar?
5
Blocos de memoria alocados com sucesso! **Tempo de execucao: 9000000.00 microsegundos**
O que deseja fazer? **1 - Alocar blocos de memoria** **2 - Liberar Blocos de memoria** **3 - Visualizar memoria** **4 - Buscar bloco** **0 - Encerrar programa**
3
Estado1 : O **Inicio1 : 0** **Fim1 : 14** **Estado1 : L** **Inicio1 : 15** **Fim1 : 20** **Estado1 : O** **Inicio1 : 21** **Fim1 : 29**
O que deseja fazer? **1 - Alocar blocos de memoria** **2 - Liberar Blocos de memoria** **3 - Visualizar memoria** **4 - Buscar bloco** **0 - Encerrar programa**
0

Teste de desempenho

Para efetuar o teste de desempenho, foram realizados testes de busca e de modificação da memória (alocação e desalocação), considerando 30 testes para se conseguir o tempo médio e casos em níveis diferentes da árvore 2-3. Considere, para efeito de teste, buscas realizadas no início, meio e fim da árvore com 3 níveis de profundidade.

local da árvore 2-3	**Tempo (em microssegundos)**
Início	1533333.33
Meio	1700000.00
Fim	1733333.33

Foram realizados também testes de alocação e desalocação de bloco completo e parcial.

local da árvore 2-3	**Tempo (em microssegundos)**
Alocação (bloco completo)	18000000.00
Alocação (bloco parcial)	3000000.00
Desalocação (bloco completo)	9000000.00
Desalocação (bloco parcial)	7000000.00

Conclusão

Através dos experimentos, foi possível analisar as características e desempenho da árvore 2-3, permitindo avaliar suas vantagens e desvantagens. A análise dos resultados obtidos permitiu concluir que a árvore 2-3 é uma ferramenta valiosa para a organização e busca de dados. Os dados nos permitem concluir que a árvore 2-3 se mostrou eficiente, destacando-se pela sua capacidade de armazenar dois ou três valores em cada nó, permitindo que ela tenha uma profundidade menor em relação a outras árvores, além de se manter sempre balanceada, o que garante melhor desempenho do algoritmo.

Apêndice

Essa seção é composta pelos algoritmos criados para a resolução das questões mencionadas nos tópicos anteriores.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <time.h>

#define QTD_EXPERIMENTOS 30

typedef struct endeMem endeMem;
struct endeMem
{
```

```
    int inicio, fim;
    char estado;
};

typedef struct blocoMem blocoMem;
struct blocoMem
{
    int nInfo;
    endeMem info1, info2;
    blocoMem *esq, *cen, *dir;
};

void menu(blocoMem **raiz, int maximoBlocoMen);
endeMem *criaEnd(char estado, int inicio, int fim);
blocoMem *criaBloco(endeMem info1, blocoMem *esq, blocoMem *cen,
blocoMem *dir);
void adicionaBloco(blocoMem **raiz, endeMem info, blocoMem
*maiorNo);
int ehFolha(blocoMem *no);
int buscaBloco(blocoMem *raiz, int inicio, int fim);
blocoMem *quebraBloco(blocoMem **raiz, endeMem bloco, endeMem
*sobe, blocoMem *maiorNo);
blocoMem *insere(blocoMem **no, endeMem bloco, endeMem *promove,
blocoMem **pai);
void mostraEmOrdem(blocoMem *Raiz);
void inverteEstadoBloco(endeMem *info);
int remover23(blocoMem **Pai, blocoMem **Raiz, endeMem valor);
void menorInfoDir(blocoMem *Raiz, blocoMem **no, blocoMem
**paiNo);
void maiorInfoEsq(blocoMem *Raiz, blocoMem **no, blocoMem
**paiNo);
void alocarMemoria(blocoMem **raiz, blocoMem **pai, int
qtdBlocosAloc, int maximoBlocoMen, int* alocou);
int removeInfo2ComDirFolha(blocoMem **Raiz, blocoMem **paiNo);
void liberarMemoria(blocoMem **raiz, blocoMem **pai, endeMem
liberarEspaco, int maximoBlocoMen, int *liberou);
int blocoValido(int inicio, int fim, int qtdBlocosMaximo);
void preenchePosicoesMemoria(blocoMem **raiz, blocoMem **pai, int
*qntBlocoMem);
void definicaoMemoria(int *qntBlocoMem,char *estadoBlocoInicial);
void imprimirNo(blocoMem *raiz);
endeMem maiorInfo(blocoMem *raiz);
```

```
endeMem menorInfo(blocoMem *raiz);
blocoMem* raiz_global = NULL;

int main()
{
    blocoMem *raiz,*pai;
    raiz = NULL;
    pai = NULL;
    int qntBlocoMem;
    preenchePosicoesMemoria(&raiz,&pai,&qntBlocoMem);
    raiz_global = raiz;
    menu(&raiz, qntBlocoMem);
    return 0;
}

void menu(blocoMem **raiz, int maximoBlocoMen)
{
    int opcao, alocou = 0, liberou = 0, qtdBlocos4Aloc,
liberInicio, liberFim;
    int inicio, fim, encontrou = 0, i;
    double tempo_total = 0;
    blocoMem *pai;
    pai = NULL;
    endeMem *blocoParaLiberar;
    do
    {
        printf("\nO que deseja fazer?\n  1 - Alocar blocos de
memoria\n  2 - Liberar Blocos de memoria\n  3 - Visualizar
memoria\n  4 - Buscar bloco\n  0 - Encerrar programa\n");
        scanf("%d", &opcao);
        switch (opcao)
        {
        case 1:
            // alocar blocos de memória
            printf("Quantos blocos deseja alocar? ");
            scanf("%d", &qtdBlocos4Aloc);


            clock_t tempo_inicio = clock();
            alocarMemoria(raiz, &pai, qtdBlocos4Aloc,
maximoBlocoMen, &alocou);
            clock_t tempo_fim = clock();
```

```
            tempo_total = (tempo_fim - tempo_inicio) * 1000000; //
transformar para microsegundos

            if (!alocou)
                printf("\nNao foi possivel alocar os blocos de
memoria!\nMemoria insuficiente\n\n");
            else
                printf("\nBlocos de memoria alocados com
sucesso!\n\nTempo de execucao: %.2f microsegundos\n",
tempo_total);
            alocou = 0;
            tempo_total = 0;
            break;
        case 2:
            // liberar blocos de memória
            printf("Onde deve iniciar a desalocar? ");
            scanf("%d",&liberInicio);
            printf("Onde deve parar de desalocar? ");
            scanf("%d",&liberFim);
            if(blocoValido(liberInicio,liberFim,maximoBlocoMen)){
                blocoParaLiberar =
criaEnd('L',liberInicio,liberFim);

                clock_t tempo_inicio = clock();

liberarMemoria(raiz,&pai,*blocoParaLiberar,maximoBlocoMen,&liberou
);
                clock_t tempo_fim = clock();
                tempo_total = (tempo_fim - tempo_inicio) *
1000000; // transformar para microsegundos

                if(!liberou)
                    printf("Nao foi possivel desalocar os blocos
de memoria informados!\n\n");
                else
                    printf("Blocos de memoria desalocados com
sucesso!\n\nTempo de execucao: %.2f microsegundos\n",
tempo_total);
            }
            else
                printf("Valores para desalocar invalidos\n");
            liberou = 0;
```

```
            tempo_total = 0;
            break;
        case 3:
            mostraEmOrdem(*raiz);
            break;
        case 4:
            // buscar um valor
            printf("\nBUSCAR BLOCO\n");
            printf("Inicio do bloco: ");
            scanf("%d", &inicio);
            printf("Fim do bloco: ");
            scanf("%d", &fim);


            for (i = 0; i < QTD_EXPERIMENTOS; i++)
            {
                clock_t tempo_inicio = clock();
                encontrou = buscaBloco(*raiz, inicio, fim);
                clock_t tempo_fim = clock();
                tempo_total += (tempo_fim - tempo_inicio);
            }

            tempo_total *= 1000000; // trasnforma para
microsegundos
            tempo_total /= QTD_EXPERIMENTOS;

            if (encontrou)
            {
                printf("\nBloco encontrado!\n\ntempo médio para
buscar: %.2f microssegundos\n\n", tempo_total);
            } else {
                printf("\nBloco nao encontrado!\n\n");
            }

            tempo_total = 0;
            break;
        default:
            break;
        }
        raiz_global = *raiz;

    } while (opcao != 0);
```

```
}

void preenchePosicoesMemoria(blocoMem **raiz, blocoMem **pai, int
*qntBlocoMem){
    endeMem *promove,*primeiroEnd;
    promove = (endeMem *)calloc(1, sizeof(endeMem));
    primeiroEnd = NULL;
    int inicioBlocoMem = 0, fimBlocoMem = 0;
    char estadoBlocoInicial;

    definicaoMemoria(qntBlocoMem, &estadoBlocoInicial);
    // Insere o primeiro bloco
    while (fimBlocoMem < *qntBlocoMem - 1)
    {
        // Lê o final de cada bloco e insere
        printf("Bloco iniciando na posicao %d.", inicioBlocoMem);
        printf("Informe a posicao do fim do bloco de memoria: ");
        scanf("%d", &fimBlocoMem); // Fim do bloco atual
        while (fimBlocoMem >= *qntBlocoMem)
        { // Permite apenas valores válidos de fim do bloco
            printf("Extouro de memoria, ultima posicao disponivel
%dmb\n", *qntBlocoMem - 1);
            printf("Insira novamente a ultima posicao do bloco:
");
            scanf("%d", &fimBlocoMem);
        }
        primeiroEnd = criaEnd(estadoBlocoInicial, inicioBlocoMem,
fimBlocoMem);
        insere(raiz, *primeiroEnd, promove, pai);
        if (estadoBlocoInicial == 'L')
            estadoBlocoInicial = 'O';
        else
            estadoBlocoInicial = 'L';

        inicioBlocoMem = fimBlocoMem + 1;
    }

}

void definicaoMemoria(int *qntBlocoMem,char *estadoBlocoInicial){
    printf("Quantidade de memoria disponivel em megabyte: ");
    scanf("%d", qntBlocoMem);
```

```
    printf("Qual o estado do primeiro bloco(L/O)? ");
    setbuf(stdin, NULL);
    scanf("%c", estadoBlocoInicial);
    *estadoBlocoInicial = toupper(*estadoBlocoInicial);
    while (*estadoBlocoInicial != 'L' && *estadoBlocoInicial !=
'O')
    {
        setbuf(stdin, NULL);
        printf("Valor invalido, por favor informe L ou O : ");
        scanf("%c", estadoBlocoInicial);
        *estadoBlocoInicial = toupper(*estadoBlocoInicial);
    }
}

void alocarMemoria(blocoMem **raiz, blocoMem **pai, int
qtdBlocosAloc, int maximoBlocoMen, int *alocou)
{
    int removeu = 0, novoInicio, novoFim;
    endeMem *aux, *aux2, *promove, maior, menor;
    blocoMem *no, *paiAux;
    aux = NULL, promove = NULL, paiAux = NULL;
    if (*raiz != NULL)
    {
        if (!*alocou)
        {
            alocarMemoria(&((*raiz)->esq), raiz, qtdBlocosAloc,
maximoBlocoMen, alocou);
        }

        if (!*alocou && (*raiz)->info1.estado == 'L')
        {
            int qtdBlocoDisp = ((*raiz)->info1.fim -
(*raiz)->info1.inicio) + 1;

            if (qtdBlocoDisp == qtdBlocosAloc) // quantidade de
blocos exata que deseja alocar
            {
                // insercao no inicio
                if ((*raiz)->info1.inicio == 0)
                {
                    if (*pai != NULL)
                    {
```

```
                        (*pai)->info1.inicio =
(*raiz)->info1.inicio;
                        // Remove o nó pois só havia 1 informação
                        removeu = remover23(pai, raiz,
(*raiz)->info1);
                        *alocou = 1;
                    }
                    else if ((*raiz)->nInfo == 2)
                    { // Se tiver info2
                        (*raiz)->info2.inicio =
(*raiz)->info1.inicio;
                        (*raiz)->info1 = (*raiz)->info2;
                        (*raiz)->info2.inicio = -1;
                        (*raiz)->info2.fim = -1;
                        (*raiz)->nInfo = 1;
                        // Altera a info2 e move info 2 pra info
1, libera Info2
                        *alocou = 1;
                    }
                    else
                    { // Não tem pai e não tem info 2
                        inverteEstadoBloco(&((*raiz)->info1));
                        *alocou = 1;
                    }
                } else if ((*raiz)->info1.fim == maximoBlocoMen -
1){
                    // insercao no fim
                    // Se tem 1 info, olha pro pai
                    if ((*pai)->nInfo == 2){
                        (*pai)->info2.fim = (*raiz)->info1.fim;
                        removeu = remover23(pai, raiz,
(*raiz)->info1);
                    }
                    else {
                        (*pai)->info1.fim = (*raiz)->info1.fim;
                        removeu = remover23(pai, raiz,
(*raiz)->info1);
                    }
                    *alocou = 1;
                } else {
                    // insercao no meio
```

```
                    if (ehFolha(*raiz)){
                        // eh uma folha no meio da memoria, tem
pai
                        // verifica de onde estou vindo do pai
                        if ((*raiz)->nInfo == 2)
                        {
                            if ((*pai)->esq == *raiz)
                            {
                                // no eh filho da esquerda
                                (*raiz)->info1.fim =
(*raiz)->info2.fim;
                                removeu = remover23(pai, raiz,
(*raiz)->info2);
                            } else if ((*pai)->cen == *raiz){
                                // no eh filho do centro
                                (*pai)->info1.fim =
(*raiz)->info2.fim;
                                maior = (*raiz)->info2;
                                removeu = remover23(pai, raiz,
(*raiz)->info1);
                                removeu = remover23(&paiAux,
&raiz_global, maior);
                            } else {
                                // no eh filho da direita
                                (*raiz)->info1.fim =
(*raiz)->info2.fim;
                                removeu = remover23(pai, raiz,
(*raiz)->info2);
                            }
                        } else {
                            if ((*pai)->esq == *raiz)
                            {
                                // no eh filho da esquerda
                                (*pai)->info1.inicio =
(*raiz)->info1.inicio;
                                removeu = remover23(pai, raiz,
(*raiz)->info1);
                            } else if ((*pai)->cen == *raiz){
                                // no eh filho do centro
                                if ((*pai)->nInfo == 2)
                                {
                                    (*pai)->info1.fim =
```

```
(*pai)->info2.fim;
                                    maior = (*pai)->info2;
                                    removeu = remover23(pai, raiz,
(*raiz)->info1);
                                    removeu = remover23(&paiAux,
&raiz_global, maior);
                                } else {
                                    /* code */
                                    (*pai)->info1.fim =
(*raiz)->info1.fim;
                                    removeu = remover23(pai, raiz,
(*raiz)->info1);
                                }
                            } else {
                                // no eh filho da direita
                                (*pai)->info2.fim =
(*raiz)->info1.fim;
                                removeu = remover23(pai, raiz,
(*raiz)->info1);
                            }
                        }
                        *alocou = 1;

                    } else {
                        // nao eh folha

                        // busca maior info da esquerda
                        maiorInfoEsq((*raiz)->esq, &no, pai);
                        menor = menorInfo((*raiz)->cen);

                        if(no->nInfo == 2){
                            // tem duas infos
                            no->info2.fim = menor.fim;
                        }
                        else {
                            no->info1.fim = menor.fim;
                        }

                        // remove raiz e menor
                        removeu = remover23(pai, &raiz_global,
(*raiz)->info1);
                        removeu = remover23(pai, &raiz_global,
```

```
menor);

                        *alocou = 1;
                    }
                }

            } else if (qtdBlocoDisp > qtdBlocosAloc) // Resta
blocos na alocação
            {
                // insercao no inicio
                if ((*raiz)->info1.inicio == 0)
                {
                    if (*pai != NULL)
                    {
                        (*pai)->info1.inicio =
(*pai)->info1.inicio - qtdBlocosAloc;
                        (*raiz)->info1.fim = (*raiz)->info1.fim -
qtdBlocosAloc;

                    }
                    else if ((*raiz)->nInfo == 2)
                    { // Se tiver info2

                        (*raiz)->info2.inicio =
(*raiz)->info2.inicio - qtdBlocosAloc;
                        (*raiz)->info1.fim = (*raiz)->info1.fim -
qtdBlocosAloc;
                        // Altera a info2 e move info 2 pra info
1, libera Info2
                    }
                    else
                    { // Tem uma info apenas e não tem pai
                        // Subtrai os espaço de  memoria de info1
e cria um nó ocupado com o valor subtraido
                        printf("dentro do else - 244\n");
                        novoFim = (*raiz)->info1.fim;
                        (*raiz)->info1.fim = (*raiz)->info1.fim -
qtdBlocosAloc;
                        novoInicio = (*raiz)->info1.fim + 1;
                        aux = criaEnd('O', novoInicio, novoFim);
                        insere(raiz, *aux, promove, pai);
                    }
```

```
                } else if ((*raiz)->info1.fim == maximoBlocoMen -
1){
                    // insercao no fim
                    if ((*pai)->nInfo == 2) // Pai tem duas
informações, altera info2
                        (*pai)->info2.fim = (*pai)->info2.fim +
qtdBlocosAloc;
                    else // Pai tem uma info, altera info1
                        (*pai)->info1.fim = (*pai)->info1.fim +
qtdBlocosAloc;
                    (*raiz)->info1.inicio = (*raiz)->info1.inicio
+ qtdBlocosAloc;
                } else {
                    // insercao no meio
                    if(ehFolha(*raiz)){
                        if ((*raiz)->nInfo == 2)
                        {
                            (*raiz)->info2.inicio =
(*raiz)->info2.inicio - qtdBlocosAloc;
                            (*raiz)->info1.fim =
(*raiz)->info1.fim - qtdBlocosAloc;
                        } else {
                            if ((*pai)->esq == *raiz)
                            {
                                (*raiz)->info1.fim =
(*raiz)->info1.fim - qtdBlocosAloc;
                                (*pai)->info1.inicio =
(*pai)->info1.inicio - qtdBlocosAloc;
                            } else if ((*pai)->cen == *raiz)
                            {
                                (*pai)->info1.fim +=
qtdBlocosAloc;
                                (*raiz)->info1.inicio +=
qtdBlocosAloc;
                            }
                        }
                    } else {
                        maiorInfoEsq((*raiz)->esq, &no, pai);
                        if (no->nInfo == 2)
                        {
                            no->info2.fim += qtdBlocosAloc;
                        }else {
```

```
                        no->info1.fim += qtdBlocosAloc;
                    }
                    (*raiz)->info1.inicio += qtdBlocosAloc;
                }
            }
            *alocou = 1;
        }
    }

    if (!*alocou)
    {
        alocarMemoria(&((*raiz)->cen), raiz, qtdBlocosAloc,
maximoBlocoMen, alocou);
    }

    // Analisar a info2, caso tenha
    if (!*alocou && (*raiz)->nInfo == 2)
    { // Raiz tem dois filhos, checa a info 2
        if ((*raiz)->info2.estado == 'L')
        {

            int qtdBlocoDisp = (*raiz)->info2.fim -
(*raiz)->info2.inicio + 1;

            if (qtdBlocoDisp == qtdBlocosAloc)
            {
                // verificar se tem filho do centro
                // verificar se o centro tem duas infos

                if ((*raiz)->cen != NULL)
                {
                    // tem filho do centro

                    // busca a maior info do centro
                    maiorInfoEsq((*raiz)->cen, &no, pai);

                    // busca a menor info da direita
                    menor = menorInfo((*raiz)->dir);

                    // altera tamanho da maior info do centro
                    if (no->nInfo == 2)
                    {
```

```
                        // tem duas infos
                        no->info2.fim = menor.fim;
                    }
                    else
                    {
                        no->info1.fim = menor.fim;
                    }

                    // remove raiz e menor info da direita
                    removeu = remover23(pai, &raiz_global,
menor);
                    removeu = remover23(pai, &raiz_global,
(*raiz)->info2);
                    *alocou = 1;
                }
                else
                {
                    // Altera o tamanho da info1 e remove a
info2
                    (*raiz)->info1.fim = (*raiz)->info2.fim;
                    removeu = remover23(pai, raiz,
(*raiz)->info2);
                    *alocou = 1;
                }


            }
            else if (qtdBlocoDisp > qtdBlocosAloc)
            {
                if (ehFolha(*raiz))
                {
                    (*raiz)->info1.fim = (*raiz)->info1.fim +
qtdBlocosAloc;
                    (*raiz)->info2.inicio =
(*raiz)->info2.inicio + qtdBlocosAloc;
                }else {
                    // busca maior info do centro
                    maiorInfoEsq((*raiz)->cen, &no, pai);
                    if (no->nInfo == 2)
                    {
                        no->info2.fim += qtdBlocosAloc;
                    }else {
```

```
                            no->info1.fim += qtdBlocosAloc;
                        }
                        (*raiz)->info2.inicio += qtdBlocosAloc;
                    }
                    *alocou = 1;
                }
            }
        }

        if (!*alocou)
        {
            alocarMemoria(&((*raiz)->dir), raiz, qtdBlocosAloc,
maximoBlocoMen, alocou);
        }
    }
}

int blocoValido(int inicio, int fim, int qtdBlocosMaximo){
    int ehValido = 0;
    if(inicio <= fim && inicio >= 0 && inicio <= qtdBlocosMaximo -
1 && fim <= qtdBlocosMaximo - 1){
       ehValido = 1;
    }
    return ehValido;
}

void liberarMemoria(blocoMem **raiz, blocoMem **pai, endeMem
liberarEspaco, int maximoBlocoMen, int *liberou){
    int removeu = 0, novoInicio, novoFim;
    endeMem auxMaior, auxMenor, auxMaiorInfo2;
    endeMem *aux, *aux2, *promove;
    aux = NULL;
    blocoMem *paiAux,*no;
    paiAux = NULL;
    no = NULL;

    promove = (endeMem*)calloc(1,sizeof(endeMem));
    if (*raiz != NULL)
    {
        int qtdBlocosDesalocar = liberarEspaco.fim -
liberarEspaco.inicio + 1;
        if (!*liberou)
```

```
            liberarMemoria(&((*raiz)->esq), raiz, liberarEspaco,
maximoBlocoMen, liberou);
        if (!*liberou && (*raiz)->info1.estado == 'O'){
            int qtdBlocoDispInfo1 = ((*raiz)->info1.fim -
(*raiz)->info1.inicio) + 1;

            if (qtdBlocoDispInfo1 >= qtdBlocosDesalocar){ //Tem a
exata quantidade de memória desejada
                if((*raiz)->info1.inicio == liberarEspaco.inicio
&& (*raiz)->info1.fim == liberarEspaco.fim){//Caso perfeito, o
espaço a ser liberado é a info1
                    if(ehFolha(*raiz))
                    {
                        // quantidade de infos da raiz
                        if((*raiz)->nInfo == 1){
                            // verifica se tem pai
                            if (*pai != NULL)
                            {
                                // tem pai
                                // de onde a raiz veio
                                if ((*pai)->esq == *raiz)
                                {
                                    // filho da esquerda
                                    aux = criaEnd('L',
(*raiz)->info1.inicio, (*pai)->info1.fim);
                                    auxMenor = (*raiz)->info1;
                                    auxMaior = (*pai)->info1;
                                    removeu = remover23(pai, raiz,
auxMenor);
                                    removeu = remover23(pai, raiz,
auxMaior);

                                    insere(raiz, *aux, promove,
pai);

                                    *liberou = 1;

                                } else if ((*pai)->cen == *raiz)
                                {
                                    // filho do centro

                                    if ((*pai)->nInfo == 2)
```

```
                                    {
                                        auxMaior = (*raiz)->info1;
                                        auxMaiorInfo2 =
(*pai)->info2;
                                        aux = criaEnd('L',
(*pai)->info1.inicio, (*pai)->info2.fim);

                                        paiAux = NULL;
                                        removeu = remover23(pai,
raiz, auxMaior);
                                        paiAux = NULL;
                                        removeu =
remover23(&paiAux, &raiz_global, auxMaiorInfo2);

                                        (*pai)->info1.fim =
auxMaiorInfo2.fim;


                                        *liberou = 1;
                                    } else {
                                        aux = criaEnd('L',
(*pai)->info1.inicio, (*raiz)->info1.fim);
                                        auxMenor = (*pai)->info1;
                                        auxMaior = (*raiz)->info1;
                                        removeu = remover23(pai,
raiz, auxMaior);
                                        removeu =
remover23(&paiAux, &raiz_global, auxMenor);

                                        insere(&raiz_global, *aux,
promove, &paiAux);
                                        *liberou = 1;
                                    }

                                } else {
                                    // filho da direita
                                    aux = criaEnd('L',
(*pai)->info2.inicio, (*raiz)->info1.fim);
                                    auxMenor = (*pai)->info2;
                                    auxMaior = (*raiz)->info1;
                                    paiAux = NULL;
                                    removeu = remover23(pai, raiz,
auxMaior);
```

```
                                    removeu = remover23(&paiAux,
&raiz_global, auxMenor);

                                    insere(raiz, *aux, promove,
pai);
                                    *liberou = 1;
                                }


                            } else {
                                // não tem pai
                                // inverter estado

inverteEstadoBloco(&((*raiz)->info1));
                                *liberou = 1;
                            }

                        } else {
                            // tem duas info
                            aux = criaEnd('L',
(*raiz)->info1.inicio, (*raiz)->info2.fim);
                            if (*pai != NULL)
                            {
                                // tem pai
                                // de onde a raiz veio
                                if ((*pai)->esq == *raiz)
                                {
                                    // filho da esquerda
                                    auxMenor = (*raiz)->info1;
                                    auxMaior = (*raiz)->info2;
                                    removeu = remover23(pai, raiz,
auxMenor);
                                    removeu = remover23(pai, raiz,
auxMaior);

                                    insere(raiz, *aux, promove,
pai);

                                } else if ((*pai)->cen == *raiz)
                                {
                                    // filho do centro
                                    aux->inicio =
```

```
(*pai)->info2.inicio;
                                    auxMenor = (*pai)->info1;
                                    auxMaior = (*raiz)->info1;
                                    auxMaiorInfo2 =
(*raiz)->info2;
                                    paiAux = NULL;
                                    removeu =
remover23(&raiz_global, pai, auxMenor); // Perigo: removendo info
do pai
                                    removeu = remover23(pai, raiz,
auxMaior);
                                    removeu = remover23(pai, raiz,
auxMaiorInfo2);

                                    insere(raiz, *aux, promove,
pai);

                                } else {
                                    // filho da direita
                                    printf("Filho da direita\n");
                                    aux->inicio =
(*pai)->info2.inicio;
                                    auxMenor = (*pai)->info2;
                                    auxMaior = (*raiz)->info1;
                                    auxMaiorInfo2 =
(*raiz)->info2;
                                    removeu =
remover23(&raiz_global, raiz, auxMenor);
                                    removeu = remover23(pai, raiz,
auxMaior);
                                    removeu = remover23(pai, raiz,
auxMaiorInfo2);

                                    insere(raiz, *aux, promove,
pai);
                                    *liberou = 1;
                                }
                            } else {
                                // não tem pai
                                removeu = remover23(&paiAux,
&raiz_global, (*raiz)->info2);
                                (*raiz)->info1.fim = aux->fim;
```

```
                            (*raiz)->info1.estado = 'L';
                            *liberou = 1;
                        }
                    }
                }else {
                    // buscar maior valor da esquerda

                    printf("Esse caso\n");
                    auxMaior = maiorInfo((*raiz)->esq);
                    auxMenor = menorInfo((*raiz)->cen);
                    aux = criaEnd('L', auxMaior.inicio,
auxMenor.fim);

                    removeu = remover23(pai, raiz, auxMenor);
                    removeu = remover23(pai, raiz, auxMaior);

                    (*raiz)->info1.inicio = auxMaior.inicio;
                    (*raiz)->info1.fim = auxMenor.fim;
                    (*raiz)->info1.estado = 'L';

                    *liberou = 1;
                }
            }
            // Info1 é ocupado, tem a quantidade necessário
pra desalocar porém deve retirar uma parte da info1
            else if((*raiz)->info1.inicio ==
liberarEspaco.inicio){
                // verifica se esta no inicio da memoria
                if((*raiz)->info1.inicio == 0){
                    // inicio da memoria
                    (*raiz)->info1.inicio =
(*raiz)->info1.inicio + qtdBlocosDesalocar;
                    aux =
criaEnd('L',liberarEspaco.inicio,liberarEspaco.fim);
                    insere(raiz, *aux, promove, pai);
                } else{
                    // não esta no inicio da memoria
                    // verifica sou filho do centro ou da
direita

                    // verifica se tem pai
                    if(*pai != NULL){
```

```
                            if ((*pai)->cen == *raiz) // raiz eh
filho do centro
                                (*pai)->info1.fim =
(*pai)->info1.fim + qtdBlocosDesalocar;
                            else // raiz eh filho da direita
                                (*pai)->info2.fim =
(*pai)->info2.fim + qtdBlocosDesalocar;
                        } else {
                            // se ele nao tem pai, ele eh a raiz e
tem filho da esquerda
                            // verifica se filho da esquerda tem
duas infos
                            if((*raiz)->esq->nInfo == 2){
                                // filho da esquerda tem duas
infos
                                // altera info2
                                (*raiz)->esq->info2.fim =
(*raiz)->esq->info2.fim + qtdBlocosDesalocar;
                            } else {
                                // filho da esquerda tem uma info
                                // altera info1
                                (*raiz)->esq->info1.fim =
(*raiz)->esq->info1.fim + qtdBlocosDesalocar;
                            }
                        }
                        (*raiz)->info1.inicio =
(*raiz)->info1.inicio + qtdBlocosDesalocar;
                    }
                    *liberou = 1;
                }
                else if((*raiz)->info1.fim == liberarEspaco.fim){
                    // No fim do bloco

                    // verifica se esta no fim da memoria
                    if((*raiz)->info1.fim == maximoBlocoMen - 1){
                        // fim da memoria
                        (*raiz)->info1.fim = (*raiz)->info1.fim -
qtdBlocosDesalocar;
                        aux =
criaEnd('L',liberarEspaco.inicio,liberarEspaco.fim);
                        insere(raiz, *aux, promove, pai);
                    } else{
```

```
                        // não esta no fim da memoria
                        // verifica sou filho do centro ou da
esquerda

                        if ((*pai)->cen == *raiz){ // raiz eh
filho do centro
                            if((*pai)->nInfo == 2){
                                (*raiz)->info1.fim =
(*raiz)->info1.fim - qtdBlocosDesalocar;
                                (*pai)->info2.inicio =
(*pai)->info2.inicio - qtdBlocosDesalocar;
                            }
                            else{
                                (*raiz)->info1.fim =
(*raiz)->info1.fim - qtdBlocosDesalocar;
                                aux =
criaEnd('L',liberarEspaco.inicio,liberarEspaco.fim);
                                insere(raiz, *aux, promove, pai);
                            }
                        }
                        else if((*pai)->esq == *raiz){
                            (*raiz)->info1.fim =
(*raiz)->info1.fim - qtdBlocosDesalocar;
                            (*pai)->info1.inicio =
(*pai)->info1.inicio - qtdBlocosDesalocar;
                        } // raiz eh filho da esquerda
                    }
                    *liberou = 1;
                } else if ((*raiz)->info1.inicio <
liberarEspaco.inicio && (*raiz)->info1.fim > liberarEspaco.fim)
                {
                    // desalocando no meio de um bloco
                    if ((*raiz)->info1.inicio == 0)
                    {
                        // inicio da memoria
                        // verifica se tem pai
                        aux = criaEnd('O', (*raiz)->info1.inicio,
liberarEspaco.inicio - 1);
                        aux2 = criaEnd('L', liberarEspaco.inicio,
liberarEspaco.fim);
                        (*raiz)->info1.inicio = liberarEspaco.fim
+ 1;
```

```
                        insere(raiz, *aux2, promove, pai);

                        insere(pai, *aux, promove, &paiAux);

                    }else if ((*raiz)->info1.fim == maximoBlocoMen
- 1){
                        // fim da memoria
                        printf("Entrei aq\n");
                        aux2 = criaEnd('L', liberarEspaco.inicio,
liberarEspaco.fim);
                        aux = criaEnd('O', liberarEspaco.fim + 1,
(*raiz)->info1.fim);
                        (*raiz)->info1.fim = liberarEspaco.inicio
- 1;
                        insere(raiz, *aux2, promove, pai);
                        insere(pai, *aux, promove, &paiAux);
                    } else {
                        // meio da memoria
                        aux2 = criaEnd('L', liberarEspaco.inicio,
liberarEspaco.fim);
                        aux = criaEnd('O', liberarEspaco.fim + 1,
(*raiz)->info1.fim);
                        (*raiz)->info1.fim = liberarEspaco.inicio
- 1;
                        insere(raiz, *aux2, promove, pai);
                        insere(pai, *aux, promove, &paiAux);
                    }
                    *liberou = 1;
                }
            }
        }
        if (!*liberou)
            liberarMemoria(&((*raiz)->cen), raiz, liberarEspaco,
maximoBlocoMen, liberou);
        if (!*liberou && (*raiz)->info2.estado == 'O'){
            int qtdBlocoDispInfo2 = ((*raiz)->info2.fim -
(*raiz)->info2.inicio) + 1;
            if (qtdBlocoDispInfo2 >= qtdBlocosDesalocar){//Tem a
quantidade necessária de blocos
                if((*raiz)->info2.inicio == liberarEspaco.inicio
&& (*raiz)->info2.fim == liberarEspaco.fim){//quer toda a info2
                    if(ehFolha(*raiz)){//Se for folha
```

```
                        printf("Ehfolha\n");
                        if((*raiz)->info1.estado == 'L'){// e
info1 for livre, junta as infos
                            printf("Ehfolha e info1 eh livre\n");
                            (*raiz)->info1.fim =
(*raiz)->info2.fim;
                            (*raiz)->info2.inicio = -1;
                            (*raiz)->info2.fim = -1;
                            (*raiz)->nInfo = 1;

                            if(*pai != NULL){//Se tiver pai, junta
a info 1 da raiz com a do pai
                            //Nó veio da esq do pai
                                if((*pai)->nInfo == 2){
                                    (*pai)->info1.inicio =
(*pai)->info2.inicio;
                                    removeu = remover23(pai, raiz,
(*raiz)->info1);
                                }

                            }
                        }
                    }
                    else{
                        endeMem maior,menor, auxRaiz;
                        maior = maiorInfo((*raiz)->cen);
                        menor = menorInfo((*raiz)->dir);

                        auxRaiz.inicio = (*raiz)->info2.inicio;
                        auxRaiz.fim = (*raiz)->info2.fim;

                        removeu = remover23(pai, raiz, menor);
                        removeu = remover23(pai, raiz, maior);
                        removeu = remover23(pai, raiz, auxRaiz);

                        auxRaiz.estado = 'L';
                        auxRaiz.inicio = maior.inicio;
                        auxRaiz.fim = menor.fim;

                        insere(raiz, auxRaiz, promove, pai);
                    }
                    *liberou = 1;
```

```
                }
                else if((*raiz)->info2.inicio <=
liberarEspaco.inicio && (*raiz)->info2.fim >=
liberarEspaco.fim){//Quer parte da info2
                    if(ehFolha(*raiz)){//Apenas redimensiona as
infos
                        if((*raiz)->info2.inicio ==
liberarEspaco.inicio){//Inicio da info2
                            (*raiz)->info1.fim =
(*raiz)->info1.fim + qtdBlocosDesalocar;
                            (*raiz)->info2.inicio =
(*raiz)->info2.inicio + qtdBlocosDesalocar;
                        }
                        else if((*raiz)->info2.fim ==
liberarEspaco.fim){//fim da info2
                            (*raiz)->info2.fim =
(*raiz)->info2.fim - qtdBlocosDesalocar;
                            aux =
criaEnd('L',liberarEspaco.inicio,liberarEspaco.fim);
                            paiAux = NULL;
                            insere(raiz,*aux,promove,&paiAux);
                        }
                        else{//Meio da info2
                            aux = criaEnd('O',liberarEspaco.fim +
1,(*raiz)->info2.fim);//3

                            (*raiz)->info2.fim =
liberarEspaco.inicio - 1;//1

                            aux2 =
criaEnd('L',liberarEspaco.inicio,liberarEspaco.fim);//2

                            paiAux = NULL;
                            insere(raiz,*aux2,promove,&paiAux);
                            paiAux = NULL;
                            insere(raiz,*aux,promove,&paiAux);
                        }
                    }
                    else{//Se não for folha
                        if((*raiz)->info2.inicio ==
liberarEspaco.inicio){//Inicio da info2
                            if((*raiz)->cen->nInfo == 2)//Se o
```

```
centro diver duas infos
                                (*raiz)->cen->info2.fim =
(*raiz)->cen->info2.fim + qtdBlocosDesalocar;
                            else//Se o centro tiver 1 info só
                                (*raiz)->cen->info1.fim =
(*raiz)->cen->info1.fim + qtdBlocosDesalocar;
                            (*raiz)->info2.inicio =
(*raiz)->info2.inicio + qtdBlocosDesalocar;
                        }
                        else if((*raiz)->info2.fim ==
liberarEspaco.fim){//Fim da info2
                            (*raiz)->dir->info1.inicio =
(*raiz)->dir->info1.inicio - qtdBlocosDesalocar;
                            (*raiz)->info2.fim =
(*raiz)->info2.fim - qtdBlocosDesalocar;
                        }
                        else{//Meio da info2
                            aux2 =
criaEnd('O',liberarEspaco.fim+1,(*raiz)->info2.fim);//2

                            (*raiz)->info2.fim =
liberarEspaco.inicio - 1;//1

                            aux =
criaEnd('L',liberarEspaco.inicio,liberarEspaco.fim);//3

                            paiAux = NULL;
                            insere(raiz,*aux2,promove,&paiAux);
                            paiAux = NULL;
                            insere(raiz,*aux,promove,&paiAux);
                        }
                    }
                    *liberou = 1;
                }
            }
        }
        if (!*liberou)
            liberarMemoria(&((*raiz)->dir), raiz, liberarEspaco,
maximoBlocoMen, liberou);
    }
}
```

```
endeMem maiorInfo(blocoMem *raiz){
    endeMem maiorInf;
    maiorInf.inicio = -1;
    maiorInf.fim = -1;
    if(raiz != NULL){
        if(ehFolha(raiz)){
            if(raiz->nInfo == 2){
                maiorInf = raiz->info2;
            }
            else
                maiorInf = raiz->info1;
        }
        else{
            if(raiz->nInfo == 1){
                maiorInf = maiorInfo(raiz->cen);
            }
            else{
                maiorInf = maiorInfo(raiz->dir);
            }
        }
    }
    return maiorInf;
}

endeMem menorInfo(blocoMem *raiz){
    endeMem menorInf;
    menorInf.inicio = -1;
    menorInf.fim = -1;
    if(raiz != NULL){
        if(ehFolha(raiz)){
            menorInf = raiz->info1;
        }
        else{
            menorInf = maiorInfo(raiz->esq);
        }
    }
    return menorInf;
}

void inverteEstadoBloco(endeMem *info)
{
    if (info->estado == 'L')
```

```
        info->estado = 'O';
    else
        info->estado = 'L';
}

endeMem *criaEnd(char estado, int inicio, int fim)
{
    endeMem *novoEnd;
    novoEnd = (endeMem *)malloc(sizeof(endeMem));
    novoEnd->estado = estado;
    novoEnd->inicio = inicio;
    novoEnd->fim = fim;
    return novoEnd;
}

blocoMem *criaBloco(endeMem info1, blocoMem *esq, blocoMem *cen,
blocoMem *dir)
{
    blocoMem *novoNo;
    novoNo = (blocoMem *)malloc(sizeof(blocoMem));
    novoNo->info1 = info1;
    novoNo->info2.inicio = -1;
    novoNo->info2.fim = -1;
    novoNo->esq = esq;
    novoNo->cen = cen;
    novoNo->dir = dir;
    novoNo->nInfo = 1;
    return novoNo;
}

void adicionaBloco(blocoMem **raiz, endeMem info, blocoMem
*maiorNo)
{
    if (info.inicio > ((*raiz)->info1.fim))
    {
        (*raiz)->info2 = info;
        (*raiz)->dir = maiorNo;
    } // Checar se precisa da outra condição
    else
    {
        (*raiz)->info2 = (*raiz)->info1;
        (*raiz)->info1 = info;
```

```
        (*raiz)->dir = (*raiz)->cen;
        (*raiz)->cen = maiorNo;
    }
    (*raiz)->nInfo = 2;
}

int ehFolha(blocoMem *no)
{
    int ehFolha = 0;
    if (no->esq == NULL && no->cen == NULL && no->dir == NULL)
        ehFolha = 1;
    return ehFolha;
}

blocoMem *insere(blocoMem **no, endeMem bloco, endeMem *promove,
blocoMem **pai)
{
    blocoMem *maiorNo;
    maiorNo = NULL;
    endeMem promove1;
    if (*no == NULL)                                // Se a raiz for
nula
        *no = criaBloco(bloco, NULL, NULL, NULL); // Info1, Esq,
Cen, Dir
    else
    { // Se for folha
        if (ehFolha(*no))
        {
            if ((*no)->nInfo == 1) // Se tiver uma informação
adiciona
                adicionaBloco(no, bloco, NULL);
            else
            { // Se estiver cheio, quebra o nó e sobe a info do
meio pro pai, se não tiver pai cria um nó
                maiorNo = quebraBloco(no, bloco, promove,
NULL);//3
                if (*pai == NULL)
                {
                    *no = criaBloco(*promove, *no, maiorNo, NULL);
                    maiorNo = NULL;
                }
            }
```

```
        }
        else
        { // Raiz não é nula e não é folha, chama recursivo para
um dos filhos
            if (bloco.fim < (*no)->info1.inicio)
                maiorNo = insere(&((*no)->esq), bloco, promove,
no);
            else if (((*no)->nInfo == 1) || (bloco.fim <
(*no)->info2.inicio))
                maiorNo = insere(&((*no)->cen), bloco, promove,
no);//1
            else
                maiorNo = insere(&((*no)->dir), bloco, promove,
no);//2

            if (maiorNo != NULL)
            {
                if ((*no)->nInfo == 1)
                { // Se tiver inserido e o nó só tiver uma
informação adiciona
                    adicionaBloco(no, *promove, maiorNo);
                    maiorNo = NULL;
                }
                else
                {
                    maiorNo = quebraBloco(no, *promove, &promove1,
maiorNo);
                    if (*pai == NULL)
                    {
                        *no = criaBloco(promove1, *no, maiorNo,
NULL);
                        maiorNo = NULL;
                    }
                    else{
                        *promove = promove1;
                    }
                }
            }
        }
    }
    return maiorNo;
}
```

```
blocoMem *quebraBloco(blocoMem **raiz, endeMem bloco, endeMem
*sobe, blocoMem *maiorNo)
{
    blocoMem *novoNo;
    if (bloco.fim < (*raiz)->info1.inicio)
    {
        *sobe = (*raiz)->info1; // Testar esse caso
        //*sobeGlobal = (*raiz)->info1;
        novoNo = criaBloco((*raiz)->info2, (*raiz)->cen,
(*raiz)->dir, NULL);
        (*raiz)->info1 = bloco;
        (*raiz)->cen = maiorNo;
    }
    else if (bloco.fim < (*raiz)->info2.inicio)
    {
        *sobe = bloco; // Testar esse caso
        //*sobeGlobal = bloco;
        novoNo = criaBloco((*raiz)->info2, maiorNo, (*raiz)->dir,
NULL);
    }
    else
    {
        *sobe = (*raiz)->info2; // Linha perigosa
        //*sobeGlobal = (*raiz)->info2;
        novoNo = criaBloco(bloco, (*raiz)->dir, maiorNo, NULL);
    }
    (*raiz)->info2.inicio = -1;
    (*raiz)->info2.fim = -1;
    (*raiz)->nInfo = 1;
    (*raiz)->dir = NULL;
    return novoNo;
}

void mostraEmOrdem(blocoMem *Raiz)
{
    if (Raiz != NULL)
    {
        mostraEmOrdem(Raiz->esq);
        printf("Estado1 : %c\n", Raiz->info1.estado);
        printf("Inicio1 : %d\n", Raiz->info1.inicio);
        printf("Fim1 : %d\n", Raiz->info1.fim);
```

```
        mostraEmOrdem(Raiz->cen);
        if (Raiz->info2.inicio >= 0)
        {
            printf("Estado2 : %c\n", Raiz->info2.estado);
            printf("Inicio2 : %d\n", Raiz->info2.inicio);
            printf("Fim2 : %d\n", Raiz->info2.fim);
        }
        mostraEmOrdem(Raiz->dir);
    }
}

int buscaBloco(blocoMem *raiz, int inicio, int fim)
{
    int encontrou = 0;
    if (raiz != NULL)
    {
        // verifica se o bloco esta a esquerda da info1
        if (fim < (*raiz).info1.inicio)
            encontrou = buscaBloco((*raiz).esq, inicio, fim);
        else if ((*raiz).info1.inicio == inicio &&
(*raiz).info1.fim == fim)
            encontrou = 1;
        else if ((*raiz).nInfo == 1 || (fim <
(*raiz).info2.inicio))
            encontrou = buscaBloco((*raiz).cen, inicio, fim);
        else if ((*raiz).info2.inicio == inicio &&
(*raiz).info2.fim == fim)
            encontrou = 1;
        else
            encontrou = buscaBloco((*raiz).dir, inicio, fim);
    }
    return encontrou;
}

int remover23(blocoMem **Pai, blocoMem **Raiz, endeMem valor)
{
    int removeu = 0;
    blocoMem *no = NULL, *no1, *paiNo = NULL, *paiNo1 = NULL,
**aux;
    aux = (blocoMem **)malloc(sizeof(blocoMem *));
    no1 = (blocoMem *)malloc(sizeof(blocoMem));
```

```
    if (*Raiz != NULL)
    {
        if (ehFolha(*Raiz) == 1)
        {
            if ((*Raiz)->nInfo == 2) // Quando é folhac e tem duas
informações
            {
                if (((*Raiz)->info2.inicio == valor.inicio) &&
((*Raiz)->info2.fim == valor.fim))
                { // quando é folha, tem duas informações e o
numero ta na segunda posição
                    (*Raiz)->info2.inicio = -1;
                    (*Raiz)->info2.fim = -1;
                    (*Raiz)->nInfo = 1;
                    removeu = 1;
                }
                else if (((*Raiz)->info1.inicio == valor.inicio)
&& ((*Raiz)->info1.fim == valor.fim))
                { // quando é folha, tem duas informações e o
numero ta na primeira posição do nó
                    (*Raiz)->info1 = (*Raiz)->info2;
                    (*Raiz)->info2.inicio = -1;
                    (*Raiz)->info2.fim = -1;
                    (*Raiz)->nInfo = 1;
                    removeu = 1;
                }
            }
            else if (((*Raiz)->info1.inicio == valor.inicio) &&
((*Raiz)->info1.fim == valor.fim)) // Quando é folha e tem uma
informação
            {
                if (*Pai == NULL) // Se não tem pai é pq é o
último elemento do nó
                {
                    free(*Raiz);
                    *Raiz = NULL;
                    removeu = 1;
                }
                else if (*Raiz == (*Pai)->esq) // Verifica se sou
filho da esquerda
                {
                    (*Raiz)->info1 = (*Pai)->info1;
```

```
                    paiNo = *Pai;
                    menorInfoDir((*Pai)->cen, &no, &paiNo);
                    (*Pai)->info1 = no->info1;
                    removeu = 1;

                    if (no->nInfo == 2)
                    {
                        no->info1 = no->info2;
                        no->info2.inicio = -1;
                        no->info2.fim = -1;
                        no->nInfo = 1;
                    }
                    else
                    {
                        if (paiNo->nInfo == 1)
                        {
                            (*Raiz)->info2 = no->info1;
                            (*Raiz)->nInfo = 2;
                            free(no);
                            *Pai = *Raiz;
                        }
                        else
                        {
                            no->info1 = paiNo->info2;
                            paiNo1 = paiNo;
                            menorInfoDir(paiNo->dir, &no1,
&paiNo1);
                            paiNo->info2 = no1->info1;

                            if (no1->nInfo == 2)
                            {
                                no1->info1 = no1->info2;
                                no1->info2.inicio = -1;
                                no1->info2.fim = -1;
                                no1->nInfo = 1;
                            }
                            else
                            {
                                no->info2 = paiNo->info2;
                                no->nInfo = 2;
                                paiNo->info2.inicio = -1;
                                paiNo->info2.fim = -1;
```

```
                                paiNo->nInfo = 1;
                                free(no1);
                                paiNo1->dir = NULL;
                            }
                        }
                    }
                }
                else if (*Raiz == (*Pai)->cen) // Verifica se razi
e filho do centro
                {
                    removeu = 1;
                    if ((*Pai)->nInfo == 1) // O pai tem só uma
informação
                    {
                        if (((*Pai)->esq)->nInfo == 2) // O filho
a esquerda do pai tem duas informações
                        {
                            (*Raiz)->info1 = (*Pai)->info1;
                            (*Pai)->info1 = ((*Pai)->esq)->info2;
                            ((*Pai)->esq)->info2.inicio = -1;
                            ((*Pai)->esq)->info2.fim = -1;
                            ((*Pai)->esq)->nInfo = 1;
                        }
                        else // O filho a esquerda do pai tem uma
informação
                        {
                            ((*Pai)->esq)->info2 = (*Pai)->info1;
                            free(*Raiz);
                            ((*Pai)->esq)->nInfo = 2;
                            *aux = (*Pai)->esq;
                            free(*Pai);
                            *Pai = *aux;
                        }
                    }
                    else // O pai tem duas informações
                    {
                        (*Raiz)->info1 = (*Pai)->info2;
                        paiNo = *Pai;
                        menorInfoDir((*Pai)->dir, &no, &paiNo);
                        (*Pai)->info2 = no->info1;

                        if (no->nInfo == 2)
```

```
            {
                no->info1 = no->info2;
                no->info2.inicio = -1;
                no->info2.fim = -1;
                no->nInfo = 1;
            }
            else
            {
                (*Raiz)->nInfo = 2;
                (*Raiz)->info2 = (*Pai)->info2;
                (*Pai)->info2.inicio = -1;
                (*Pai)->info2.fim = -1;
                (*Pai)->nInfo = 1;
                free(no);
                (*Pai)->dir = NULL;
            }
        }
    }
    else // Raiz e filho da direita
    {
        removeu = 1;
        paiNo = *Pai;
        maiorInfoEsq((*Pai)->cen, &no, &paiNo);

        if (no->nInfo == 1)
        {
            no->info2 = (*Pai)->info2;
            (*Pai)->info2.inicio = -1;
            (*Pai)->info2.fim = -1;
            (*Pai)->nInfo = 1;
            no->nInfo = 2;
            free(*Raiz);
            *Raiz = NULL;
        }
        else
        {
            (*Raiz)->info1 = (*Pai)->info2;
            (*Pai)->info2 = no->info2;
            no->info2.inicio = -1;
            no->info2.fim = -1;
            no->nInfo = 1;
        }
```

```
                }
            }
        }
        else
        { // se nao é folha
            if (valor.fim < (*Raiz)->info1.inicio)
                removeu = remover23(Raiz, &(*Raiz)->esq, valor);
            else if (((*Raiz)->info1.inicio == valor.inicio) &&
((*Raiz)->info1.fim == valor.fim))
            {
                paiNo = *Raiz;
                menorInfoDir((*Raiz)->cen, &no, &paiNo);
                (*Raiz)->info1 = no->info1;
                remover23(Raiz, &(*Raiz)->cen, (*Raiz)->info1);
                removeu = 1;
            }
            else if (((*Raiz)->nInfo == 1) || (valor.fim <
(*Raiz)->info2.inicio))
            {
                removeu = remover23(Raiz, &(*Raiz)->cen, valor);
            }
            else if (((*Raiz)->info2.inicio == valor.inicio) &&
((*Raiz)->info2.fim == valor.fim)) // info2 a ser removida
            {
                // verifica se é folha
                if(ehFolha(*Raiz)){
                    (*Raiz)->info1.fim = (*Raiz)->info2.fim;
                    (*Raiz)->info2.inicio = -1;
                    (*Raiz)->info2.fim = -1;
                    (*Raiz)->nInfo = 1;
                    removeu = 1;
                } else if(ehFolha((*Raiz)->dir)) // verificar se o
filho da direita eh folha
                {
                    // remover info2 com direita folha
                    removeu = removeInfo2ComDirFolha(Raiz, Pai);
                }else {
                    // direita nao eh folha
                    // obter no com menor info da direita
                    // trocar info2 com menor info da direita
                    menorInfoDir((*Raiz)->dir, &no, &paiNo);
                    (*Raiz)->info2 = no->info1;
```

```
                    // remover menor info da direita
                    removeu = remover23(Raiz, &(*Raiz)->dir,
no->info1);
                }
            }
            else
            {
                removeu = remover23(Raiz, &(*Raiz)->dir, valor);
            }
        }
    }
    return removeu;
}

int removeInfo2ComDirFolha(blocoMem **Raiz, blocoMem **paiNo){
    // decobrir se filho da direita tem 2 info
    if ((*Raiz)->dir->nInfo == 2)
    {
        // trocar info2 com filho da direita
        (*Raiz)->info2 = (*Raiz)->dir->info1;
        // remover info2 do filho da direita
        (*Raiz)->dir->info1 = (*Raiz)->dir->info2;
        (*Raiz)->dir->info2.inicio = -1;
        (*Raiz)->dir->info2.fim = -1;
        (*Raiz)->dir->nInfo = 1;
    }
    else
    {
        // filho da direita tem 1 info
        // verificar se centro tem duas info
        if ((*Raiz)->cen->nInfo == 2)
        {
            // centro tem duas info
            // trocar info2 com centro
            (*Raiz)->info2 = (*Raiz)->cen->info2;
            // remover info2 do centro
            (*Raiz)->cen->info2.inicio = -1;
            (*Raiz)->cen->info2.fim = -1;
            (*Raiz)->cen->nInfo = 1;
        }
        else
```

```
        {

            // Info da direita vai para info 2 do centro
            (*Raiz)->cen->info2 = (*Raiz)->dir->info1;
            (*Raiz)->cen->nInfo = 2;

            // remove info2 da raiz
            (*Raiz)->info2.inicio = -1;
            (*Raiz)->info2.fim = -1;
            (*Raiz)->nInfo = 1;

            // remove info da direita
            int removeu = remover23(Raiz, &(*Raiz)->dir,
(*Raiz)->dir->info1);
        }

    }

}

void menorInfoDir(blocoMem *Raiz, blocoMem **no, blocoMem **paiNo)
{
    if (Raiz->esq != NULL)
    {
        *paiNo = Raiz;
        menorInfoDir(Raiz->esq, no, paiNo);
    }
    else
        *no = Raiz;
}

void maiorInfoEsq(blocoMem *Raiz, blocoMem **no, blocoMem **paiNo)
{
    if(Raiz != NULL){
        if(ehFolha(Raiz))
            *no = Raiz;
        else{
            if(Raiz->nInfo == 1){
                maiorInfoEsq(Raiz->cen, no, paiNo);
            }
            else{
                maiorInfoEsq(Raiz->dir, no, paiNo);
```

```
            }
        }
    }
}


void imprimirNo(blocoMem *raiz)
{
    if(raiz != NULL){
        printf("info1 -> Inicio %d Fim %d Estado
%c\n",raiz->info1.inicio,raiz->info1.fim,raiz->info1.estado);
        if(raiz->nInfo == 2){
            printf("info2 -> Inicio %d Fim %d Estado
%c\n",raiz->info2.inicio,raiz->info2.fim,raiz->info2.estado);
        }
    }
}
```