UNIVERSIDADE FEDERAL DE OURO PRETO

RUAN TIENGO ROCHA

TRABALHO PRATICO 1

MUNDO DOS BLOCOS

BELO HORIZONTE
2021

1. Introdução

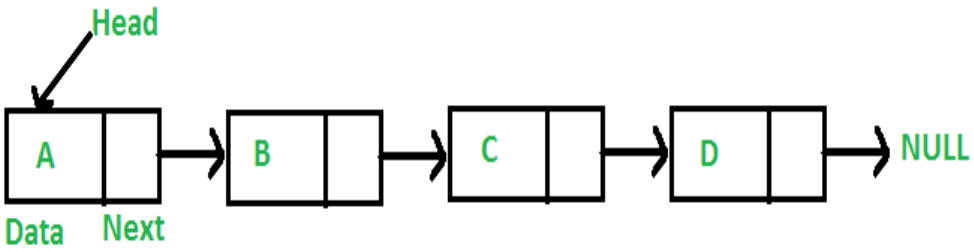
O objetivo da criação desse programa, é abstrair do mundo real a ideia de um conjunto de blocos, e criar tipos de movimentações que são possíveis de serem feitas com as manipulações desses blocos. O programa, funciona recebendo um número inteiro do usuário representando o número de listas (e por conseguinte, número de blocos) que farão parte do ”Mundo dos Blocos”. Em seguida são dadas as opções ao usuário de escolher qual operação será executada e quais blocos serão movidos. Esses procedimentos podem ser feitos até o comando de saída do usuário.

2. Implementação

O programa é dividido em 3 arquivos, 2 cabeçalhos e um arquivo source, sendo eles, respectivamente: LinkedList.h, MundoBlocos.h e main.c. Para facilitar a descrição desse programa, trataremos por cada arquivo de cabeçalho separadamente.

LinkedList

Esse arquivo trata da implementação do tipo abstrato de dados. No programo feito foi usada a implementação de uma lista encadeada simples. Nessa implementação, as células funcionam como os blocos, e os itens como os valores dos blocos.



1.1

Nesta lista foi optado o método de inicialização com cabeça, a fim de facilitar a implementação os métodos de inserção e remoção de células.

```
typedef int NumBloco;
typedef struct ESTItem {
    NumBloco numBloco;
} TItem;

typedef struct ESTCelula {
    TItem item;
    struct ESTCelula *nextCelula;
} TCelula;

typedef struct ESTList {
    TCelula *primeiro;
    TCelula *ultimo;
} Lista;

void IniciaLista(Lista *lista) {
    lista->primeiro = (TCelula *) malloc(sizeof(TCelula));
    lista->ultimo = lista->primeiro;
```

```
    lista->primeiro->nextCelula = NULL;
}
```

A explicação dessa implementação é muito simples, é criada uma estrutura “TItem”, e nela é contido o numero do bloco, já na estrutura TCelula, que é a abstração do bloco em si, é armazenado nela, a estrutura TItem representando o numero do bloco, e uma referência ao próximo bloco. Já na estrutura lista teremos o valor de duas referências a 2 TCelulas, a primeira da lista, e a última. Para iniciar a lista temos a inicialização com cabeça em que o primeiro elemento da lista é igualado ao último, e o próximo elemento, até ser adicionado, sempre é igual a nulo.

```
 void Insere(Lista *lista, TItem *item) {
    lista->ultimo->nextCelula = (TCelula *) malloc(sizeof(TCelula));
    lista->ultimo = lista->ultimo->nextCelula;
    lista->ultimo->item = *item;
    lista->ultimo->nextCelula = NULL;
}

void Retira(Lista *list,TItem* item) {
    TCelula *atual = list->primeiro->nextCelula;
    TCelula *previa = list->primeiro;
    while (atual != NULL) {
        if (atual->item.numBloco == item->numBloco) {
            previa->nextCelula = atual->nextCelula;
            if (atual == list->ultimo)
                list->ultimo = previa;
            free(atual);
            return;
        }
        previa = atual;
        atual = atual->nextCelula;
    }
}
```

O método de inserção funciona de maneira simples em que a lista a ter o valor inserido é passada como parâmetro junto com o item a ser adicionado, já no método é instanciado um espaço de memória para a próxima célula, que passa a ser a última célula da lista, e é atribuída a ela o valor passado como referência.

Já no método de remoção são passados como parâmetro a lista a ter o elemento retirado, e o elemento a ser retirado, basicamente nesse método são criados duas células para representar a célula atual e a passada, após isso, é criado um método while para percorrer toda a lista até a célula a ser retirada ser encontrada, e quando esse item é encontrado, a célula previa recebe o valor da atual e quando chegar no ultimo elemento, ela recebe a célula a ser retirada e é deletada pelo programa.

```
void *RetiraUltimo(Lista *lista) {
    TCelula *aux = lista->primeiro->nextCelula;
    TCelula *aux2;
    int cont = 0;
    while (cont < TamanhoLista(lista) - 2) {
        aux = aux->nextCelula;
        cont++;
```

```
    }
    lista->ultimo = aux;
    aux2 = aux->nextCelula;
    aux->nextCelula = NULL;
}
```

Um importante método para a implementação do código foi uma variação do método “Retira”, porém nesse caso, não é necessário escolher qual elemento será retirado, ele sempre retira o último bloco da lista especificada no parâmetro.

```
void InserirElementoCelula(Lista *lista, TCelula *celula) {
    TItem *item = malloc(sizeof(struct ESTItem));
    *item = celula->item;
    return Insere(lista, item);
}

void RetiraElementoCelula(Lista *lista, TCelula *celula) {
    TItem *item = malloc(sizeof(struct ESTItem));
    *item = celula->item;
    return Retira(lista, item);
}
```

Esses métodos, cumprem a função de tirar a necessidade de o tempo todo trabalharmos com o ponteiro de itens, fazendo com que nos retornemos diretamente a celula e ele crie o item, e o retorne para o método “Retira” e “Insere”.

```
int TamanhoLista(Lista *list) {
    int tam = 0;
    TCelula *aux = list->primeiro->nextCelula;
    while (aux != NULL) {
        tam++;
        aux = aux->nextCelula;
    }
    return tam;
}

void ImprimeLista(Lista *lista) {
    TCelula *aux = lista->primeiro->nextCelula;
    while (aux != NULL) {
        printf("|  %d  | - ", aux->item.numBloco);
        aux = aux->nextCelula;
    }
    printf("\n");
}

TItem *generateItem(int x) {
    TItem *a = malloc(sizeof(TItem));
    a->numBloco = x;
    return a;
}
```

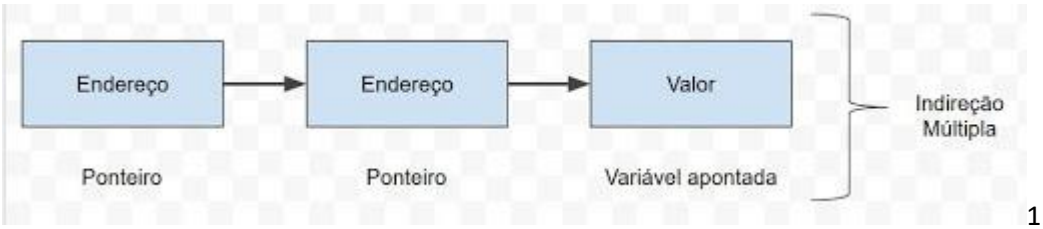
Esses métodos cumprem funções simples em nosso programa, o “TamanhoLista” retorna o tamanho da lista enviada como parâmetro. O imprime lista tem a importante função de imprimir a lista para o usuário. Já o método “generateItem” eu o uso para facilitar a manipulação dos blocos.

MundoBlocos

Nesse cabeçalho foi incluído o header LinkedList (responsável pela implementação da lista encadeada simples), para realizar a manipulação das listas.

```
typedef struct ESTMundoBlocos {
    Lista **listas;
    int nListas;
} TBlocos;
```

O primeiro desafio proposto é pensar numa maneira de representar esse vetor de listas para criar a estrutura de blocos, que como se sabe, são representadas por um vetor de listas encadeadas simples. Para abstrair esse pensamento foi feito um ponteiro de ponteiros, em que um ponteiro aponta para outros endereços de memorias que alocam espaços em memória para uma lista. Além disso,



1.2

```
void IniciaBlocos(TBlocos *mundoBlocos, int numListas) {
    mundoBlocos->nListas = numListas;
    mundoBlocos->listas = malloc(mundoBlocos->nListas * sizeof(Lista *));
    int cont = 0;
    while (cont != mundoBlocos->nListas) {
        mundoBlocos->listas[cont] = malloc(sizeof(Lista));
        IniciaLista(mundoBlocos->listas[cont]);
        Insere(mundoBlocos->listas[cont], generateItem(cont));
        cont++;
    }
}
```

Esse método inicia o “mundo dos blocos”, em que inicialmente é recebida o numero de listas que o usuário deseja, e o mundoBlocos que é desejado iniciar, após isso, o ponteiro das listas recebe dinamicamente o seu tamanho, e após isso, é criado um laço de repetição para criar dinamicamente cada lista do mundo dos blocos. A segunda etapa, também dentro do laço, é inserida o primeiro bloco (proporcionalmente ao número de listas) de cada uma das listas, tendo como valor a posição deles, Exemplo o bloco na lista de posição 3, terá valor 3, e assim para todos os blocos criados.

```
TCelula *ProcuraBloco(TBlocos *bloco, int value) {

    for (int i = 0; i < bloco->nListas; i++) {
        TCelula *celula = bloco->listas[i]->primeiro->nextCelula;
```

```
        for (int j = 0; j < TamanhoLista(bloco->listas[i]); j++) {
            if (celula->item.numBloco == value) {
                return celula;
            }
            celula = celula->nextCelula;
        }
    }
    return NULL;
}
```

Esse método é fundamental para o programa pois é a partir dele que o usuário pode digitar o número dos blocos que ele deseja mover e o programa “passa esse valor” para o número dos blocos (TCelulas). Ele funciona de maneira simples percorrendo todas as listas até encontrar os valores solicitados, e quando encontra, retorna a celula(bloco) que tem esse valor.

Agora que o Mundo Dos Blocos já está devidamente iniciado e já esta explicitado como ele recebe os dados do usuario, é o momento de tratar das operações problemas.

MoveAontoB

Para facilitar a visualização os métodos serão tratados por etapas para facilitar a explicação e visualização de como ele realmente funciona e o método auxiliar que foi usado.

```
void moveAontoB(TBlocos *mBlocos, TCelula *a, TCelula *b) {

    // Find the list where the block is
    int posA = BuscaPosListas(mBlocos, a);
    int posB = BuscaPosListas(mBlocos, b);
```

Para iniciar esse método, nós recebemos duas células e, primeiro de tudo precisamos de saber em que lista esses blocos estão contidos por isso, foi implementado o método “BuscaPosListas”, vejamos a sua implementação e explicação de como funciona:

```
int BuscaPosListas(TBlocos *mBlocos, TCelula *a) {
    TCelula *aux;
    int cont = 0;
    for (int i = 0; i < mBlocos->nListas; i++) {
        cont = 0;
        aux = mBlocos->listas[i]->primeiro->nextCelula;
        while (aux != NULL) {
            if (aux->item.numBloco == a->item.numBloco) {
                return i;
            }
            cont++;
            aux = aux->nextCelula;
        }
    }
}
```

Esse método em teoria é bastante simples, ele é um laço encaixo em que percorre todas as listas (em que o primeiro laço roda todas as listas e o segundo laço roda todas as listas até encontrar o bloco, e quando esse bloco é encontrado, é retornada a posição da lista.

```
void moveAontoB(TBlocos *mBlocos, TCelula *a, TCelula *b) {
// Continuação "move AontoB"
Lista *listaA = mBlocos->listas[posA];
Lista *listaB = mBlocos->listas[posB];
```

Agora o que é feito é apontar ponteiros para as listas que contém as células para iniciarmos os processos de manipulação dos blocos.

```
void moveAontoB(TBlocos *mBlocos, TCelula *a, TCelula *b) {
// Continuação "move AontoB"
if (listaA->ultimo != a) TornaUltimo(mBlocos, listaA);
if (listaB->ultimo   != b) TornaUltimo(mBlocos, listaB);
```

Esse movimento exige que garantamos que não exista nenhum bloco encima de nenhum dos blocos que desejamos manipular, por isso entramos numa parte fundamental do algoritmo, que verifica se existe algum bloco encima do blocos que desejamos mover, e que se caso exista, devemos volta-los para a posição original (posição de quando inicializamos a lista). Portanto veremos agora o método "TornaUltimo":

```
void TornaUltimo(TBlocos *bloco, Lista *lista) {
    TCelula *aux = lista->primeiro->nextCelula;
    int cont = TamanhoLista(lista);
    while (cont != 0) {
        TCelula *aux2 = lista->ultimo;
        int valor = getNumBloco(lista->ultimo);
        InserirElementoCelula(bloco->listas[valor], aux2);
        RetiraUltimo(lista);
        cont--;
    }
}
```

Esse método tem uma auxiliar contadora que recebe o tamanho da lista que foi passada como parâmetro, e um função while que é repetida o número de vezes da variável cont (tamanho da lista), esse laço busca procurar qual é a lista original do elemento, e inseri-lo na posição original.

```
// add a element
InserirElementoCelula(listaB, a);

// remove the last element
RetiraElementoCelula(listaA, a);
```

Após verificadas as condições para serem realizadas as operações, são feitas a inserção do bloco A em cima do bloco B.

Código completo:

```
void moveAontoB(TBlocos *mBlocos, TCelula *a, TCelula *b) {

    // Find the list where the block is
    int posA = BuscaPosListas(mBlocos, a);
    int posB = BuscaPosListas(mBlocos, b);


    // check that there are no blocks above, and if so, return those blocks
to their original position
    Lista *listaA = mBlocos->listas[posA];
    Lista *listaB = mBlocos->listas[posB];
    if (listaA->ultimo != a) TornaUltimo(mBlocos, listaA);
    if (listaB->ultimo   != b) TornaUltimo(mBlocos, listaB);


    // add a element
    InserirElementoCelula(listaB, a);

    // remove the last element
    RetiraElementoCelula(listaA, a);

}
```

MoveAoverB

```
void moveAoverB(TBlocos *mBlocos, TCelula *a, TCelula *b) {
    // Find the list where the block is
    int posA = BuscaPosListas(mBlocos, a);
    int posB = BuscaPosListas(mBlocos, b);
    Lista *listaA = mBlocos->listas[posA];
    Lista *listaB = mBlocos->listas[posB];

    // check that there are no blocks above, and if so, return those blocks
to their original position
    if (listaA->ultimo != a) TornaUltimo(mBlocos, listaA);

    // add a element
    InserirElementoCelula(listaB, a);
    // remove the last element
    RetiraElementoCelula(listaA, a);

}
```

Esse algoritmo não tem necessidade de uma explicação detalhada como a anterior pois ele é uma copia exata do algoritmo do “MoveAontoB” o único detalhe que o diferencia é que não há necessidade de chamar a função “TornaUltimo” para ambas as listas, já que a lista b pode continuar com todos os elementos da mesma maneira, apenas o bloco A que não pode ter nenhum elemento encima dele para ser retirado.

PileAontoB

```
void pileAontoB(TBlocos *mBlocos, TCelula *a, TCelula *b) {
    int posA = BuscaPosListas(mBlocos, a);
    int posB = BuscaPosListas(mBlocos, b);
```

```
    Lista *listaA = mBlocos->listas[posA];
    Lista *listaB = mBlocos->listas[posB];
    if (listaB->ultimo != b) TornaUltimo(mBlocos, listaB);
    Pile(listaA, listaB, a);
}
```

As 4 primeiras linhas desse código têm a mesma funcionalidade dos 2 anteriores, eles identificam em que lista estão as células e criam ponteiros apontando para essas listas. Para obedecer que o bloco B seja o último, é chamado o método “TornaUltima” que retorna os blocos em cima do b para a posição original. A grande novidade está no método “Pile”, foi dado esse nome pela semelhança com uma Pilha, em que são retirados os elementos da lista a, e inseridos em ordem que foram desempilhados na lista b.

```
void Pile(Lista *listaA, Lista *listaB, TCelula *a) {
    TCelula* aux = listaA->primeiro->nextCelula;
    TCelula* passado = listaA->ultimo;
    while (aux->item.numBloco != a->item.numBloco){
        aux = aux->nextCelula;
    }
    while (aux != NULL){
        InserirElementoCelula(listaB,aux);
        aux = aux->nextCelula;
    }
    while (passado->item.numBloco != a->item.numBloco){
        RetiraUltimo(listaA);
        passado = listaA->ultimo;
    }
    RetiraElementoCelula(listaA,a);
}
```

```
void pileAontoB(TBlocos *mBlocos, TCelula *a, TCelula *b) {
    int posA = BuscaPosListas(mBlocos, a);
    int posB = BuscaPosListas(mBlocos, b);
    Lista *listaA = mBlocos->listas[posA];
    Lista *listaB = mBlocos->listas[posB];
    if (listaB->ultimo != b) TornaUltimo(mBlocos, listaB);
    Pile(listaA, listaB, a);
}
```

O método “Pile” cria um laço passando todos os blocos que estão em cima do bloco b e ele mesmo para a lista b e após isso cria um laço para retirar todos esses elementos da lista A.

```
void pileAoverB(TBlocos *mBlocos, TCelula *a, TCelula *b) {
    int posA = BuscaPosListas(mBlocos, a);
    int posB = BuscaPosListas(mBlocos, b);
    Lista *listaA = mBlocos->listas[posA];
    Lista *listaB = mBlocos->listas[posB];
    Pile(listaA, listaB, a);
}
```

Esse método, assim como o “MoveAontoB” é quase uma copia do seu antecessor, a única diferença dele para o “pileAontoB” é que ele não verificação alguma se existe algum elemento a cima dos

blocos passados como parâmetro , ele simplesmente desempilha os blocos da lista a e os empilha na lista B.

Program

```
int Program() {
    desenhoLogo();
    TBlocos *mBlocos = malloc(sizeof(TBlocos));
    int tam;
    printf("\nDigite o numero de listas: \n");
    scanf("%d", &tam);
    while (tam < 2){
        printf("Valor invalido, digite outro: \n");
        fflush(stdin);
        scanf("%d", &tam);
    }
    IniciaBlocos(mBlocos, tam);
    while (1) {
        imprimeBloco(mBlocos);
        int digito;
        printf("\nDigite o comando a ser feito \n 0- quit \n 1- Move A onto
B\n 2- Move A over B\n");
        printf(" 3- Pile A onto B\n 4- Pile A over B\n");
        fflush(stdin);
        scanf("%d", &digito);
        if (digito == 0) {
            free(mBlocos);
            break;
        }
        if (digito == 1) {
            int a;
            int b;
            printf("---------------------------------\n");
            printf("A posição das listas variam de 0 a %d\n", mBlocos-
>nListas - 1);
            printf("---------------------------------\n");
            printf("Digite o valor do mundoBlocos a: \n");
            fflush(stdin);
            scanf("%d", &a);

            printf("Digite o valor de mundoBlocos b: \n");
            fflush(stdin);
            scanf("%d", &b);
            if (a > mBlocos->nListas - 1 || b > mBlocos->nListas - 1) {
                printf("Numero superior ao numero de listas, tente
novamente");
            } else if (mBlocos->nListas == 1) {
                printf("Não ha listas sufiencentes para realizar essa
operação, inicie o programa novamente");
            } else if (a == b) {
                printf("A = B, tente novamente");
            } else if (BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, a)) ==
                       BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, b))) {
                printf("Lista igual");
            } else {
                moveAontoB(mBlocos, ProcuraBloco(mBlocos, a),
ProcuraBloco(mBlocos, b));
            }
            printf("\n");
```

```
        }
        if (digito == 2) {
            int a;
            int b;
            printf("---------------------------------\n");
            printf("A posição das listas variam de 0 a %d\n", mBlocos-
>nListas - 1);
            printf("---------------------------------\n");
            printf("Digite o valor do a: \n");
            fflush(stdin);
            scanf("%d", &a);

            printf("Digite o valor de b: \n");
            fflush(stdin);
            scanf("%d", &b);
            if (a > mBlocos->nListas - 1 || b > mBlocos->nListas - 1) {
                printf("Numero superior ao numero de listas, tente
novamente");
            } else if (mBlocos->nListas == 1) {
                printf("Não ha listas sufiencentes para realizar essa
operação, inicie o programa novamente");
            } else if (a == b) {
                printf("A = B, tente novamente");
            } else if (BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, a)) ==
                       BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, b))) {
                printf("Lista igual");
            } else
                moveAoverB(mBlocos, ProcuraBloco(mBlocos, a),
ProcuraBloco(mBlocos, b));
            printf("\n");
        }
        if (digito == 3) {
            int a;
            int b;
            printf("---------------------------------\n");
            printf("A posição das listas variam de 0 a %d\n", mBlocos-
>nListas - 1);
            printf("---------------------------------\n");
            printf("Digite o valor do a: \n");
            fflush(stdin);
            scanf("%d", &a);

            printf("Digite o valor de b: \n");
            fflush(stdin);
            scanf("%d", &b);
            if (a > mBlocos->nListas - 1 || b > mBlocos->nListas - 1) {
                printf("Numero superior ao numero de listas, tente
novamente");
            } else if (mBlocos->nListas == 1) {
                printf("Não ha listas sufiencentes para realizar essa
operação, inicie o programa novamente");
            } else if (a == b) {
                printf("A = B, tente novamente");
            } else if (BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, a)) ==
                       BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, b))) {
                printf("Lista igual");
            } else
                pileAontoB(mBlocos, ProcuraBloco(mBlocos, a),
ProcuraBloco(mBlocos, b));
            printf("\n");
        }
```

```
        if (digito == 4) {
            int a;
            int b;
            printf("---------------------------------\n");
            printf("A posição das listas variam de 0 a %d\n", mBlocos-
>nListas - 1);
            printf("---------------------------------\n");
            printf("Digite o valor do a: \n");
            fflush(stdin);
            scanf("%d", &a);

            printf("Digite o valor de b: \n");
            fflush(stdin);
            scanf("%d", &b);
            if (a > mBlocos->nListas - 1 || b > mBlocos->nListas - 1) {
                printf("Numero superior ao numero de listas, tente
novamente");
            } else if (mBlocos->nListas == 1) {
                printf("Não ha listas sufiencentes para realizar essa
operação, inicie o programa novamente");
            } else if (a == b) {
                printf("A = B, tente novamente");
            } else if (BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, a)) ==
                       BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, b))) {
                printf("Lista igual");
            }else
                pileAoverB(mBlocos, ProcuraBloco(mBlocos, a),
ProcuraBloco(mBlocos, b));
            printf("\n");
        }
    }
}
```

Esse é método é o método final do programa, que permitirá ao usuário digitar as entradas, e “filtrar” as entradas e verificar se elas são aceitáveis (como se são números maiores que 0 ou menores que o número de listas).

Main

```
#include "MundoBlocos.h"

int main() {
    Program();
    return 0;
}
```

O arquivo main ficou responsável apenas por rodar o método “Program”.

3. Listagem dos testes

Testando o método IniciaBlocos

```
Digite o numero de listas:
10
0   |  0  | -
1   |  1  | -
2   |  2  | -
3   |  3  | -
4   |  4  | -
5   |  5  | -
6   |  6  | -
7   |  7  | -
8   |  8  | -
9   |  9  | -
```

Esse é o caso em que é passado um inteiro maior ou igual a dois é inserido pelo usuário, e o programa avança com sucesso.

```
Digite o numero de listas:
1
Valor invalido, digite outro:
```

Esse caso é o caso em que o numero digitado é menor que 2, ele pede outro numero ao usuario ate ser inserido um numero maior que 2.

```
Digite o numero de listas:
2.5
0   |  0  | -
1   |  1  | -
```

Casos em que é digitado um numero reais que não sejam inteiros, ele desconsidera o ponto flutuante e fazendo o numero de listas na parte inteira do numero.

Qual metodo ele entrará

```
Digite o comando a ser feito
 0- quit
 1- Move A onto B
 2- Move A over B
 3- Pile A onto B
 4- Pile A over B
```

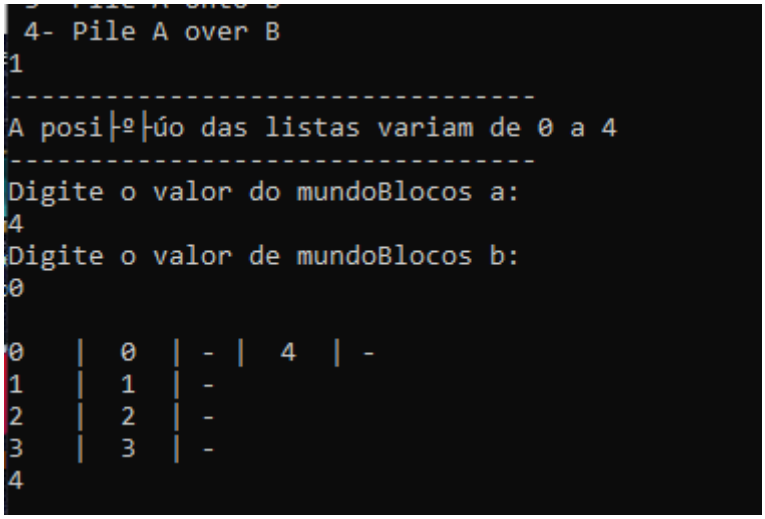
Aqui qualquer numero diferente de 0 a 4, ele imprimira o Mundo Dos Blocos e perguntara novamente ao usuario

Move A onto B

CASO 1: Blocos unicos em listas diferentes

MOVE 4 ONTO 0:

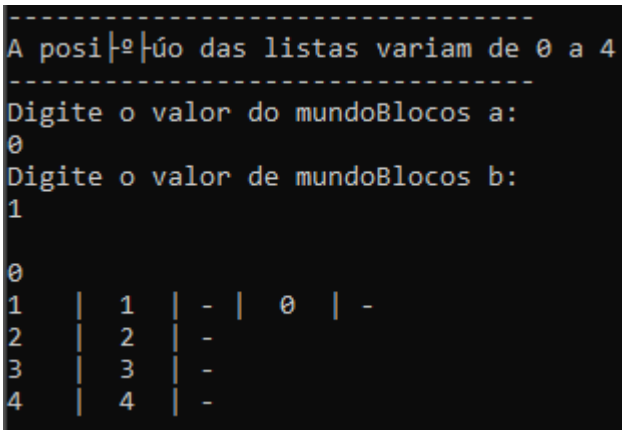
Desloca o bloco 4 para encima do bloco 0.



CASO 2: Bloco A com algum elemento encima, e B unico

MOVE 0 ONTO 1:

Descoloca o bloco 4 para a lista original e coloca o bloco 0 em cima do bloco 1.



Repare que aqui o valor A entrou no metodo “Torna Ultimo”, como havia um elemento acima do bloco 0, o programa retornou o valor 4 a lista original.

CASO 3: Os blocos A e B não são os primeiros das suas listas correspondentes (antes desse teste foi feito move 2 onto 3)

Move 0 onto 2: Desloca o bloco 0 para cima do bloco 2.

```
A posi├º├úo das listas variam de 0 a 4
--------------------------------
Digite o valor do mundoBlocos a:
0
Digite o valor de mundoBlocos b:
2

0
1   |  1  | -
2
3   |  3  | - |  2  | - |  0  | -
4   |  4  | -
```

Move A over B

CASO 1: Blocos unicos em listas diferentes

```
Digite o valor do a:
0
Digite o valor de b:
1

0
1   |  1  | - |  0  | -
2   |  2  | -
3   |  3  | -
4   |  4  | -
```

CASO 2: Blocos com outros blocos em cima (antes desse teste foi feito move 3 onto 4)

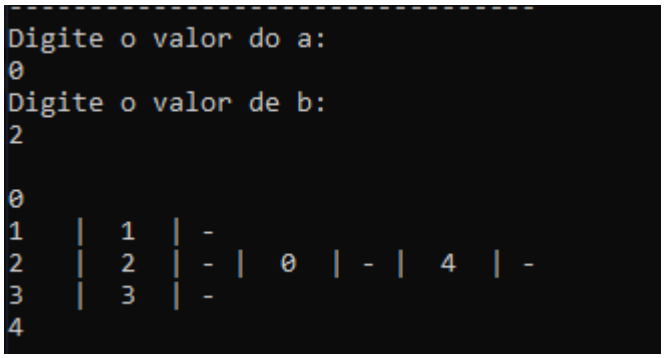
```
---------------------------------
Digite o valor do a:
1
Digite o valor de b:
4

0   |  0  | -
1
2   |  2  | -
3
4   |  4  | - |  3  | - |  1  | -
```

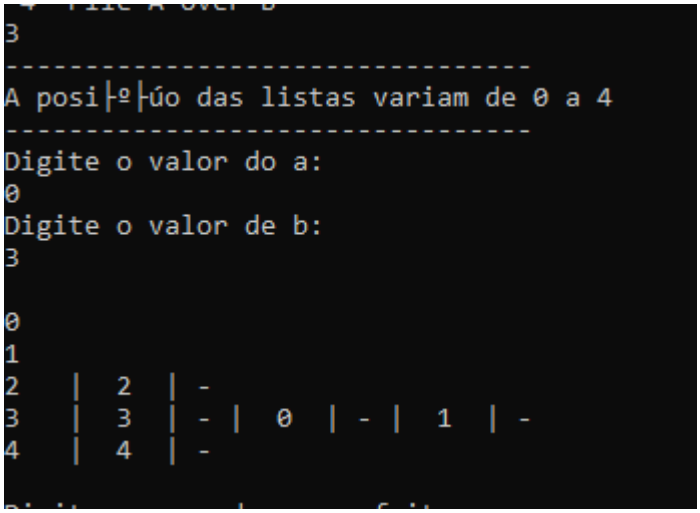
Repare que o 0 voltou a posição original, tambem porque a função “TornaUltimo” foi chamada.

Pile A onto B

Pile 4 onto 0: Leva o bloco 4 para cima do bloco 0 (que estava em cima do bloco b)



Pile A over B



4. Conclusão

A primeira dificuldade para implementar o trabalho foi pensar numa maneira de abstrair a ideia do mundo dos blocos para a programação, e como mover os blocos. Após essa primeira dificuldade todo o andamento da implementação da resolução dos problemas foi muito simples até eu esbarrar num problema ao qual tive que recorrer a ajuda do professor, que foi no método de buscar a posição da lista, que eu estava certo que ele estava funcionando porque o programa rodava porem apresentava muitos problemas com a posição das células, então estava pensando que eu estava errando na implementação dos ponteiros e passei muito tempo procurando algum erro do tipo, e não encontrava até que meu professor me mostrou o erro e todo o programa funcionou sem maiores problema. Esse trabalho, para mim especificamente foi maravilhoso de ser feito, compreendi como funciona os tipos abstratos de dado como nunca, e após todo esse trabalho posso afirmar que sai dele, um programador muito melhor, desde aprender como comentar uma função de maneira apropriada, como realmente funcionam os ponteiros, e como realmente programar uma lista encadeada e manipula-la corretamente.3.

5. Bibliografia

Para todo o desenvolvimento eu não procurei fontes externas as aulas e aos slides disponibilizados ao professor por isso, não há bibliografia com exceção das fontes onde eu busquei as imagens que não são do meu código ou do compilador

1.1: https://www.geeksforgeeks.org/data-structures/linked-list/

1.2: http://www.bosontreinamentos.com.br/programacao-em-linguagem-c/ponteiros-em-c-indirecao-multipla/

6. Codigo fonte

```
7.
   #include "MundoBlocos.h"

   int main() {
       Program();
       return 0;
   }
```

```
//
// Created by ruant on 03/03/2021.
//

#ifndef TP1_LINKEDLIST_H
#define TP1_LINKEDLIST_H

#include "stdlib.h"
#include "stdio.h"

typedef int TBloco;
typedef struct ESTItem {
    TBloco numBloco;
} TItem;

typedef struct ESTCelula {
    TItem item;
    struct ESTCelula *nextCelula;
} TCelula;

typedef struct ESTList {
    TCelula *primeiro;
    TCelula *ultimo;
} Lista;

void IniciaLista(Lista *lista) {
    lista->primeiro = (TCelula *) malloc(sizeof(TCelula));
    lista->ultimo = lista->primeiro;
    lista->primeiro->nextCelula = NULL;
}

void Insere(Lista *lista, TItem *item) {
    lista->ultimo->nextCelula = (TCelula *) malloc(sizeof(TCelula));
    lista->ultimo = lista->ultimo->nextCelula;
```

```
    lista->ultimo->item = *item;
    lista->ultimo->nextCelula = NULL;
}

int TamanhoLista(Lista *list) {
    int tam = 0;
    TCelula *aux = list->primeiro->nextCelula;
    while (aux != NULL) {
        tam++;
        aux = aux->nextCelula;
    }
    return tam;
}

void ImprimeLista(Lista *lista) {
    TCelula *aux = lista->primeiro->nextCelula;
    while (aux != NULL) {
        printf("|  %d  | - ", aux->item.numBloco);
        aux = aux->nextCelula;
    }
    printf("\n");
}

TItem *generateItem(int x) {
    TItem *a = malloc(sizeof(TItem));
    a->numBloco = x;
    return a;
}

TCelula *RetiraUltimo(Lista *lista) {
    TCelula *aux = lista->primeiro->nextCelula;
    TCelula *aux2;
    int cont = 0;
    while (cont < TamanhoLista(lista) - 2) {
        aux = aux->nextCelula;
        cont++;
    }
    lista->ultimo = aux;
    aux2 = aux->nextCelula;
    aux->nextCelula = NULL;
    return aux2;
}

void Retira(Lista *list,TItem* item) {
    TCelula *current = list->primeiro->nextCelula;
    TCelula *previous = list->primeiro;
    while (current != NULL) {
        if (current->item.numBloco == item->numBloco) {
            previous->nextCelula = current->nextCelula;
            if (current == list->ultimo)
                list->ultimo = previous;
            free(current);
            return;
        }
        previous = current;
        current = current->nextCelula;
    }
}

void InserirElementoCelula(Lista *lista, TCelula *celula) {
    TItem *item = malloc(sizeof(struct ESTItem));
```

```
    *item = celula->item;
    return Insere(lista, item);
}

void RetiraElementoCelula(Lista *lista, TCelula *celula) {
    TItem *item = malloc(sizeof(struct ESTItem));
    *item = celula->item;
    return Retira(lista, item);
}

#endif
//
// Created by ruant on 04/03/2021.
//

#ifndef TP1_TBLOCOS_H

#include "LinkedList.h"

#define TP1_TBLOCOS_H
/*Structure of the Blocks containing a Pointer pointing to several pointers
and an integer n> 0, which represents the number
of lists*/
typedef struct ESTMundoBlocos {
    Lista **listas;
    int nListas;
} TBlocos;

/*the function below receives a variable of type TBlocos and an integer n>
0, it allocates space for the lists
 * inside of the variable mundoBlocos and creates the number of lists and
the number of blocks needed*/
void IniciaBlocos(TBlocos *mundoBlocos, int numListas) {
    mundoBlocos->nListas = numListas;
    mundoBlocos->listas = malloc(mundoBlocos->nListas * sizeof(Lista *));
    int cont = 0;
    while (cont != mundoBlocos->nListas) {
        mundoBlocos->listas[cont] = malloc(sizeof(Lista));
        IniciaLista(mundoBlocos->listas[cont]);
        Insere(mundoBlocos->listas[cont], generateItem(cont));
        cont++;
    }
}

/*/ * Receive a variable of type TBlocos and an int value and search for
the position of the block in the
 * list array * /*/
TCelula *ProcuraBloco(TBlocos *bloco, int value) {

    for (int i = 0; i < bloco->nListas; i++) {
        TCelula *celula = bloco->listas[i]->primeiro->nextCelula;
        for (int j = 0; j < TamanhoLista(bloco->listas[i]); j++) {
            if (celula->item.numBloco == value) {
                return celula;
            }
            celula = celula->nextCelula;
        }
    }
    return NULL;
}
```

```c
/*Print the mBlocos (in the left the numbers of lists and in the right the
array of lists)*/
void imprimeBloco(TBlocos *mBlocos) {
    for (int i = 0; i < mBlocos->nListas; i++) {
        printf("%d   ", i);
        ImprimeLista(mBlocos->listas[i]);
    }
}

// Receive a variable celula and return the Block number who have this
number
int getNumBloco(TCelula *celula) {
    return celula->item.numBloco;
}


/* Receives a variable of type TBloco and an int value and looks for the
position of the block in the array of lists */
int BuscaPosListas(TBlocos *mBlocos, TCelula *a) {
    TCelula *aux;
    int cont = 0;
    for (int i = 0; i < mBlocos->nListas; i++) {
        cont = 0;
        aux = mBlocos->listas[i]->primeiro->nextCelula;
        while (aux != NULL) {
            if (aux->item.numBloco == a->item.numBloco) {
                return i;
            }
            cont++;
            aux = aux->nextCelula;
        }
    }
}


/*Receive a variable type TBlocos and a list and make the element passed as
a parameter in the function who was called
 * as the last element in the list and return anothers blocks for the
original position*/
void TornaUltimo(TBlocos *bloco, Lista *lista) {
    TCelula *aux = lista->primeiro->nextCelula;
    int cont = TamanhoLista(lista);
    while (cont != 0) {
        TCelula *aux2 = lista->ultimo;
        int valor = getNumBloco(lista->ultimo);
        InserirElementoCelula(bloco->listas[valor], aux2);
        RetiraUltimo(lista);
        cont--;
    }
}

/*Move the block on top of block b, returning blocks that are already
loaded on
a or b to their original positions.*/
void moveAontoB(TBlocos *mBlocos, TCelula *a, TCelula *b) {

    // Find the list where the block is
    int posA = BuscaPosListas(mBlocos, a);
    int posB = BuscaPosListas(mBlocos, b);
```

```
    // check that there are no blocks above, and if so, return those blocks
to their original position
    Lista *listaA = mBlocos->listas[posA];
    Lista *listaB = mBlocos->listas[posB];
    if (listaA->ultimo != a) TornaUltimo(mBlocos, listaA);
    if (listaB->ultimo != b) TornaUltimo(mBlocos, listaB);


    // add a element
    InserirElementoCelula(listaB, a);

    // remove the last element
    RetiraElementoCelula(listaA, a);

}
/*Moves block a at the top of the hill where block b is, returning any
blocks that
are already over to their original positions*/
void moveAoverB(TBlocos *mBlocos, TCelula *a, TCelula *b) {
    // Find the list where the block is
    int posA = BuscaPosListas(mBlocos, a);
    int posB = BuscaPosListas(mBlocos, b);
    Lista *listaA = mBlocos->listas[posA];
    Lista *listaB = mBlocos->listas[posB];

    // check that there are no blocks above, and if so, return those blocks
to their original position
    if (listaA->ultimo != a) TornaUltimo(mBlocos, listaA);

    // add a element
    InserirElementoCelula(listaB, a);
    // remove the last element
    RetiraElementoCelula(listaA, a);

}

/*Stack the list elements from the blockA in listB and remove them from
listA*/
void Pile(Lista *listaA, Lista *listaB, TCelula *a) {
    TCelula *aux = listaA->primeiro->nextCelula;
    TCelula *passado = listaA->ultimo;
    while (aux->item.numBloco != a->item.numBloco) {
        aux = aux->nextCelula;
    }
    while (aux != NULL) {
        InserirElementoCelula(listaB, aux);
        aux = aux->nextCelula;
    }
    while (passado->item.numBloco != a->item.numBloco) {
        RetiraUltimo(listaA);
        passado = listaA->ultimo;
    }
    RetiraElementoCelula(listaA, a);
}
/*Moves the block a together with all the blocks that are on it on top of
the
block b, returning any blocks that are already on b to their original
positions.*/
void pileAontoB(TBlocos *mBlocos, TCelula *a, TCelula *b) {
    int posA = BuscaPosListas(mBlocos, a);
    int posB = BuscaPosListas(mBlocos, b);
```

```
    Lista *listaA = mBlocos->listas[posA];
    Lista *listaB = mBlocos->listas[posB];
    if (listaB->ultimo != b) TornaUltimo(mBlocos, listaB);
    Pile(listaA, listaB, a);
}
/*Place block a together with all the blocks that are on it on the mound
that contains block b.*/
void pileAoverB(TBlocos *mBlocos, TCelula *a, TCelula *b) {
    int posA = BuscaPosListas(mBlocos, a);
    int posB = BuscaPosListas(mBlocos, b);
    Lista *listaA = mBlocos->listas[posA];
    Lista *listaB = mBlocos->listas[posB];
    Pile(listaA, listaB, a);
}

/*Drawing*/
void desenhoLogo() {
    printf("\n"
           "  _____ _______    ___  ___  _________    _________  _________
\n"
           "|\\    _ \\  _    \\|\\  \\|\\  \\|\\    ___  \\|\\    ___ \\|\\
__  \\          \n"
           "\\ \\  \\\\\\__\\ \\  \\ \\  \\\\\\  \\ \\  \\\\ \\  \\ \\
\\_|\\ \\ \\   \\|\\   \\          \n"
           " \\ \\  \\\\|__| \\  \\ \\  \\\\\\  \\ \\  \\\\ \\  \\ \\  \\
\\\\ \\ \\   \\\\\\   \\         \n"
           "  \\ \\  \\     \\ \\  \\ \\  \\\\\\  \\ \\  \\\\ \\  \\ \\
\\_\\\\ \\ \\   \\\\\\   \\        \n"
           "   \\ \\__\\     \\ \\__\\ \\_______\\ \\__\\\\ \\__\\
\\_______\\ \\_______\\       \n"
           "    \\|__|      \\|__|\\|_______|\\|__|
\\|__|\\|_______|\\|_______|      \n"
           "
\n"
           "
\n"
           "
\n"
           "                      _________  _________  _________
\n"
           "                |\\    ___ \\|\\   __  \\|\\   ____\\
\n"
           "                \\ \\  \\_|\\ \\ \\  \\|\\  \\ \\  \\___|_
\n"
           "                 \\ \\  \\ \\\\ \\ \\  \\\\\\  \\ \\_____  \\
\n"
           "                  \\ \\  \\_\\\\ \\ \\  \\\\\\  \\|____|\\  \\
\n"
           "                   \\ \\_______\\ \\_______\\____\\_\\  \\
\n"
           "                    \\|_______|\\|_______|\\_________\\
\n"
           "                                        \\|_________|
\n"
           "
\n"
           "
\n"
           "
\n"
           "  _________  ___        _________  _________  _________  _________
\n"
           "|\\     __  \\|\\  \\      |\\      __  \\|\\     ____\\|\\     __  \\|\\
```

```
____\\      \n"
            "\\ \\  \\|\\ /\\ \\  \\    \\ \\  \\|\\  \\ \\  \\___|\\ \\
\\|\\  \\ \\  \\___|_    \n"
            " \\ \\    __  \\ \\  \\    \\ \\  \\\\\\  \\ \\  \\    \\ \\
\\\\\\  \\ \\_____  \\   \n"
            "  \\ \\  \\|\\  \\ \\  \\____\\ \\  \\\\\\  \\ \\  \\____\\ \\
\\\\\\  \\|____|\\  \\  \n"
            "   \\ \\_______\\ \\_______\\ \\_______\\ \\_______\\
\\_______\\____\\_\\  \\ \n"
            "
\\|_______|\\|_______|\\|_______|\\|_______|\\|_______|\\_________\\\n"
            "
\\|_________|\n"
            "
\n"
            "
");
}

/* Program */
int Program() {
    desenhoLogo();
    TBlocos *mBlocos = malloc(sizeof(TBlocos));
    int tam;
    printf("\nDigite o numero de listas: \n");
    scanf("%d", &tam);
    while (tam < 2) {
        printf("Valor invalido, digite outro: \n");
        fflush(stdin);
        scanf("%d", &tam);
    }
    IniciaBlocos(mBlocos, tam);
    while (1) {
        imprimeBloco(mBlocos);
        int digito;
        printf("\nDigite o comando a ser feito \n 0- quit \n 1- Move A onto
B\n 2- Move A over B\n");
        printf(" 3- Pile A onto B\n 4- Pile A over B\n");
        fflush(stdin);
        scanf("%d", &digito);
        if (digito == 0) {
            free(mBlocos);
            break;
        }
        if (digito == 1) {
            int a;
            int b;
            printf("---------------------------------\n");
            printf("A posição das listas variam de 0 a %d\n", mBlocos-
>nListas - 1);
            printf("---------------------------------\n");
            printf("Digite o valor do mundoBlocos a: \n");
            fflush(stdin);
            scanf("%d", &a);

            printf("Digite o valor de mundoBlocos b: \n");
            fflush(stdin);
            scanf("%d", &b);
            if (a > mBlocos->nListas - 1 || b > mBlocos->nListas - 1) {
                printf("Numero superior ao numero de listas, tente
novamente");
```

```
            } else if (mBlocos->nListas == 1) {
                printf("Não ha listas sufiencentes para realizar essa
operação, inicie o programa novamente");
            } else if (a == b) {
                printf("A = B, tente novamente");
            } else if (BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, a)) ==
                       BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, b))) {
                printf("Lista igual");
            } else {
                moveAontoB(mBlocos, ProcuraBloco(mBlocos, a),
ProcuraBloco(mBlocos, b));
            }
            printf("\n");
        }
        if (digito == 2) {
            int a;
            int b;
            printf("---------------------------------\n");
            printf("A posição das listas variam de 0 a %d\n", mBlocos-
>nListas - 1);
            printf("---------------------------------\n");
            printf("Digite o valor do a: \n");
            fflush(stdin);
            scanf("%d", &a);

            printf("Digite o valor de b: \n");
            fflush(stdin);
            scanf("%d", &b);
            if (a > mBlocos->nListas - 1 || b > mBlocos->nListas - 1) {
                printf("Numero superior ao numero de listas, tente
novamente");
            } else if (mBlocos->nListas == 1) {
                printf("Não ha listas sufiencentes para realizar essa
operação, inicie o programa novamente");
            } else if (a == b) {
                printf("A = B, tente novamente");
            } else if (BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, a)) ==
                       BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, b))) {
                printf("Lista igual");
            } else
                moveAoverB(mBlocos, ProcuraBloco(mBlocos, a),
ProcuraBloco(mBlocos, b));
            printf("\n");
        }
        if (digito == 3) {
            int a;
            int b;
            printf("---------------------------------\n");
            printf("A posição das listas variam de 0 a %d\n", mBlocos-
>nListas - 1);
            printf("---------------------------------\n");
            printf("Digite o valor do a: \n");
            fflush(stdin);
            scanf("%d", &a);

            printf("Digite o valor de b: \n");
            fflush(stdin);
            scanf("%d", &b);
            if (a > mBlocos->nListas - 1 || b > mBlocos->nListas - 1) {
                printf("Numero superior ao numero de listas, tente
novamente");
```

```
            } else if (mBlocos->nListas == 1) {
                printf("Não ha listas sufiencentes para realizar essa
operação, inicie o programa novamente");
            } else if (a == b) {
                printf("A = B, tente novamente");
            } else if (BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, a)) ==
                       BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, b))) {
                printf("Lista igual");
            } else
                pileAontoB(mBlocos, ProcuraBloco(mBlocos, a),
ProcuraBloco(mBlocos, b));
            printf("\n");
        }
        if (digito == 4) {
            int a;
            int b;
            printf("---------------------------------\n");
            printf("A posição das listas variam de 0 a %d\n", mBlocos-
>nListas - 1);
            printf("---------------------------------\n");
            printf("Digite o valor do a: \n");
            fflush(stdin);
            scanf("%d", &a);

            printf("Digite o valor de b: \n");
            fflush(stdin);
            scanf("%d", &b);
            if (a > mBlocos->nListas - 1 || b > mBlocos->nListas - 1) {
                printf("Numero superior ao numero de listas, tente
novamente");
            } else if (mBlocos->nListas == 1) {
                printf("Não ha listas sufiencentes para realizar essa
operação, inicie o programa novamente");
            } else if (a == b) {
                printf("A = B, tente novamente");
            } else if (BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, a)) ==
                       BuscaPosListas(mBlocos, ProcuraBloco(mBlocos, b))) {
                printf("Lista igual");
            } else
                pileAoverB(mBlocos, ProcuraBloco(mBlocos, a),
ProcuraBloco(mBlocos, b));
            printf("\n");
        }
    }
}

#endif //TP1_TBLOCOS_H
```