

## Problema B

Você foi contratado para desenvolver um sistema que gerencia múltiplas árvores binárias de busca. Uma das operações mais importantes deste sistema é a capacidade de fundir duas árvores em uma única árvore que contenha todos os elementos de ambas.

Dadas duas árvores binárias de busca, sua tarefa é criar uma nova árvore que contenha todos os elementos das duas árvores originais e, em seguida, exibir os elementos dessa nova árvore em ordem crescente (percurso em ordem/infixa).

Para este problema, você **DEVE resolver com árvores binárias de busca**. Observe também que você deve tratar o caso de elementos repetidos ao juntar duas árvores binárias.

### Entrada

A entrada consiste em duas seções, uma para cada árvore. A primeira linha possui um inteiro  $N_1$  representando o número de elementos da primeira árvore. A segunda linha possui  $N_1$  inteiros separados por espaço, representando os valores inteiros (positivos e negativos) a serem inseridos na primeira árvore (na ordem dada). A terceira linha possui um inteiro  $N_2$  representando o número de elementos da segunda árvore. A quarta linha possui  $N_2$  inteiros separados por espaço, representando os valores inteiros (positivos e negativos) a serem inseridos na segunda árvore (na ordem dada).

### Saída

Imprima uma única linha contendo todos os elementos da árvore resultante em **ordem crescente** (percurso infixa), separados por um espaço.

Exemplo de entrada	Exemplo de saída
3 5 3 7 3 4 6 8	3 4 5 6 7 8

Dica: Existem muitas formas de implementar uma árvore binária. Na última aula vimos uma árvore binária implementada com recursão. Abaixo segue um código iterativo (não existe

recursão) que você pode utilizar como base para resolver o problema.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Estrutura de um nó da árvore binária
struct no {
    int chave;                // Valor armazenado no nó
    struct no* esquerda;     // Ponteiro para o filho à esquerda
    struct no* direita;      // Ponteiro para o filho à direita
    struct no* pai;          // Ponteiro para o nó pai
};
typedef struct no No;

// Estrutura da árvore binária de busca
struct arvore_binaria {
    No *raiz;                // Ponteiro para o nó raiz da árvore
};
typedef struct arvore_binaria ArvoreBinaria;

// ----- //
// Insere um novo nó na árvore binária de busca iterativo
void Inserir(ArvoreBinaria *arvore, No *novo_no) {
    No *pai_atual = NULL;
    No *no_atual = arvore->raiz;

    // Percorre a árvore até encontrar a posição correta
    while (no_atual != NULL) {
        pai_atual = no_atual;
        if (novo_no->chave < no_atual->chave)
            no_atual = no_atual->esquerda;
        else
            no_atual = no_atual->direita;
    }

    // Define o pai do novo nó
    novo_no->pai = pai_atual;

    // Se a árvore está vazia, o novo nó se torna a raiz
    if (pai_atual == NULL) {
        arvore->raiz = novo_no;
    } else {
        // Insere o novo nó como filho esquerdo ou direito
        if (novo_no->chave < pai_atual->chave)
            pai_atual->esquerda = novo_no;
    }
}
```

```

        else
            pai_atual->direita = novo_no;
    }
}

// ----- //
// Busca um nó com a chave especificada na árvore
No* BuscarNo(ArvoreBinaria *arvore, int chave) {
    No *no_atual = arvore->raiz;

    // Percorre a árvore procurando pela chave
    while (no_atual != NULL) {
        if (chave == no_atual->chave)
            return no_atual;
        if (chave < no_atual->chave)
            no_atual = no_atual->esquerda;
        else
            no_atual = no_atual->direita;
    }
    return NULL; // Retorna NULL se não encontrar
}

// ----- //
// Encontra o nó com o menor valor a partir de um nó dado
No* EncontrarMinimo(No *no) {
    while (no->esquerda != NULL)
        no = no->esquerda;
    return no;
}

// ----- //
// Encontra o sucessor de um nó (próximo nó em ordem crescente)
No* EncontrarSucessor(No *no) {
    // Se tem filho à direita, o sucessor é o mínimo da subárvore direita
    if (no->direita != NULL)
        return EncontrarMinimo(no->direita);

    // Caso contrário, sobe na árvore até encontrar o sucessor
    No *pai = no->pai;
    while (pai != NULL && no == pai->direita) {
        no = pai;
        pai = pai->pai;
    }
    return pai;
}

```

```

// ----- //
// Cria uma nova árvore binária vazia
ArvoreBinaria* CriarArvore() {
    ArvoreBinaria *arvore = (ArvoreBinaria*)malloc(sizeof(ArvoreBinaria));
    arvore->raiz = NULL;
    return arvore;
}

// ----- //
// Cria um novo nó com a chave especificada
No* CriarNo(int chave) {
    No *novo_no = (No*)malloc(sizeof(No));
    novo_no->chave = chave;
    novo_no->pai = NULL;
    novo_no->esquerda = NULL;
    novo_no->direita = NULL;
    return novo_no;
}

// ----- //
// Percorre a árvore em pré-ordem (raiz, esquerda, direita)
void PercorrerPreOrdem(No *no_atual, int *primeira_impressao) {
    if (no_atual == NULL)
        return;

    // Imprime a raiz primeiro
    if (*primeira_impressao == 1) {
        printf("%d", no_atual->chave);
        *primeira_impressao = 0;
    } else {
        printf(" %d", no_atual->chave);
    }

    // Depois percorre a subárvore esquerda
    if (no_atual->esquerda != NULL)
        PercorrerPreOrdem(no_atual->esquerda, primeira_impressao);

    // Por fim percorre a subárvore direita
    if (no_atual->direita != NULL)
        PercorrerPreOrdem(no_atual->direita, primeira_impressao);

    return;
}

// ----- //
// Percorre a árvore em ordem (esquerda, raiz, direita)

```

```

void PercorrerEmOrdem(No *no_atual, int *primeira_impressao) {
    if (no_atual == NULL)
        return;

    // Primeiro percorre a subárvore esquerda
    if (no_atual->esquerda != NULL)
        PercorrerEmOrdem(no_atual->esquerda, primeira_impressao);

    // Depois imprime a raiz
    if (*primeira_impressao == 1) {
        printf("%d", no_atual->chave);
        *primeira_impressao = 0;
    } else {
        printf(" %d", no_atual->chave);
    }

    // Por fim percorre a subárvore direita
    if (no_atual->direita != NULL)
        PercorrerEmOrdem(no_atual->direita, primeira_impressao);

    return;
}

// ----- //
// Percorre a árvore em pós-ordem (esquerda, direita, raiz)
void PercorrerPosOrdem(No *no_atual, int *primeira_impressao) {
    if (no_atual == NULL)
        return;

    // Primeiro percorre a subárvore esquerda
    if (no_atual->esquerda != NULL)
        PercorrerPosOrdem(no_atual->esquerda, primeira_impressao);

    // Depois percorre a subárvore direita
    if (no_atual->direita != NULL)
        PercorrerPosOrdem(no_atual->direita, primeira_impressao);

    // Por fim imprime a raiz
    if (*primeira_impressao == 1) {
        printf("%d", no_atual->chave);
        *primeira_impressao = 0;
    } else {
        printf(" %d", no_atual->chave);
    }

    return;
}

```

```
}

// ----- //
// Destrói recursivamente todos os nós da árvore
void DestruirNos(No *raiz) {
    if (raiz == NULL)
        return;

    if (raiz->esquerda != NULL)
        DestruirNos(raiz->esquerda);

    if (raiz->direita != NULL)
        DestruirNos(raiz->direita);

    free(raiz);
    return;
}

// ----- //
// Destrói a árvore inteira e libera toda a memória
void DestruirArvore(ArvoreBinaria *arvore) {
    DestruirNos(arvore->raiz);
    free(arvore);
}
```