Problema A

Marcela nem terminou de fazer o programa anterior e o professor já lhe pediu um novo programa, com aprimoramentos sobre o programa anterior. Este novo programa é a implementação de uma Árvore Binária de Pesquisa (ou Busca) mas utilizando números inteiros agora, e que deve aceitar um comando extra (R) com relação ao programa anterior:

- I *n*: Insere na árvore binária de pesquisa o elemento n.
- INFIXA: lista os elementos já inseridos segundo o percurso infixo
- PREFIXA: lista os elementos já inseridos segundo o percurso prefixo
- POSFIXA: lista os elementos já inseridos segundo o percurso posfixo
- P *n*: Pesquisa se o elemento *n* existe ou não na árvore.
- R *n*: Remove o elemento *n* da árvore, caso ele exista.

A qualquer momento pode-se inserir um elemento, visitar os elementos previamente inseridos na ordem infixa, prefixa ou posfixa, procurar por um elemento na árvore para saber se o elemento existe ou não ou ainda retirar um elemento.

Nota: Se um elemento com duas sub-árvores (direita e esquerda) for removido, o **antecessor** (o elemento maior de sub-árvore esquerda), deve ocupar o seu lugar e ao tentar retirar um elemento que não existe, nenhuma mensagem deve ser apresentada.

Entrada

A entrada contém N operações utilizando números inteiros (1-10^6) sobre uma árvore binária de Busca, que inicialmente se encontra vazia. A primeira linha de entrada contém a inserção de algum elemento. As demais linhas de entrada podem conter quaisquer um dos comandos descritos acima, conforme exemplo abaixo. O final da entrada é determinado pela string ACABOU.

Saída

Cada linha de entrada, com exceção das linhas que contém os comandos "I" ou "R", deve produzir uma linha de saída. A saída deve ser de acordo com o exemplo fornecido abaixo. Não deve haver espaço em branco após o último caractere de cada linha, caso contrário, sua submissão receberá *Presentation Error*.

Exemplo de entrada	Exemplo de saída
I 5 I 2 I 4 I 1 INFIXA PREFIXA POSFIXA P 3 P 1 INFIXA R 1 INFIXA ACABOU	1 2 4 5 5 2 1 4 1 4 2 5 3 nao existe 1 existe 1 2 4 5 2 4 5

Dica:

Existem três casos ao remover um elemento de uma árvore binária:

- Remover folhas
- Remover nós com um filho
- Remover nós com dois filhos

Observe que você deve tratar cada um desses casos.

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.

```c
#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
```

```c
};
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;
```

```c
  }
  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;
}

// ------------------------------------------------------------------ //
// Remove um nó da árvore com a chave especificada
void Remover(ArvoreBinaria *arvore, int chave) {

  // Caso 1: Nó sem filhos (folha)

  // Caso 2: Nó com apenas um filho à esquerda

  // Caso 3: Nó com apenas um filho à direita

  // Caso 4: Nó com dois filhos
}

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

```c
  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
```

```c
  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) {
```

```c
  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);
}
```