Árvores – Codificação 1

4. Códigos para Árvore Binária

Na estrutura de dados árvore binária cada elemento possui, além da carga útil de dados, dois ponteiros, um para a subárvore da esquerda e outro para a subárvore da direita.
Então a definição da estrutura pode ser codificada como no exemplo a seguir:

```
struct Elemento {
    int dado;
    struct Elemento *esq;
    struct Elemento *dir;
};

typedef struct Elemento *Arvore;
```

As operações mínimas definidas para uma árvore binária são:
- A criação de uma árvore vazia (criaArvore);
- A inserção de um elemento em uma árvore (insereArvore);
- A busca de um elemento em uma árvore (buscaArvore);
- A remoção de um elemento de uma árvore (removeArvore);
- O percurso em pré-ordem (preOrdem);
- O percurso em in-ordem (inOrdem);
- O percurso em pós-ordem (posOrdem);

Opcionalmente, podem ser definidos também as seguintes operações:
- Teste se uma árvore está vazia (arvoreVazia);
- Remover todos os elementos de uma árvore (liberaArvore);

Por sua maior complexidade, o código de retirada de um elemento de uma árvore será deixado para a próxima semana, e os códigos opcionais são deixados como exercícios.

4.1. A criação de uma árvore vazia (criaArvore)

Observe pela definição de tipo, que o tipo `Arvore` é um ponteiro para uma estrutura do tipo `Elemento`.
Desta forma, uma árvore vazia é um ponteiro para Elemento cujo valor é NULL.
Assim, a função criaArvore() é trivial, da mesma forma que a `criaLista()` e `criaPilha()`, já estudados.

```
Arvore criaArvore(){
    return NULL;
}
```

4.2. A inserção de um elemento em uma árvore (insereArvore)

Com exceção da criação de uma árvore vazia, as demais operações sobre as árvores são recursivas. Usando a recursividade, o laço de repetição está implícito na chamada recursiva da função. Toda

estrutura de repetição precisa de uma condição de parada. No caso de uma função recursiva, a condição de parada é um caso trivial, que não chama novamente a função recursiva, e no caso de inserção de um elemento em uma árvore, o caso trivial é quando a árvore está vazia.
Então é preciso testar em primeiro lugar se a árvore está vazia, e fazer a inserção.
Caso a árvore não esteja vazia, é preciso testar se o elemento a ser inserido é menor que a raiz. Se for menor, a função deve fazer uma chamada recursiva para inserir o elemento na subárvore da esquerda. Caso contrário, deve fazer uma chamada recursiva para inserir o elemento na subárvore da direita.

O código pode ser implementado em C da seguinte forma:

```
void insereArvore(Arvore *a, int valor)
{
     // Busca da posição de inserção
     if ((*a) == NULL){ // se a árvore é vazia
          // aloca o espaço para novo registro
          (*a) = (Arvore)malloc(sizeof(struct Elemento));
          // preenche os valores
          (*a) -> dado = valor;
          (*a) -> esq = NULL;
          (*a) -> dir = NULL;
     }
     else if (valor < (*a) -> dado) // se o elemento é menor que a raiz
          // insere na subárvore da esquerda
          insereArvore(&((*a) -> esq), valor);
     else if (valor > (*a) -> dado) // se o elemento é maior que a raiz
          // insere na subárvore da direita
          insereArvore(&((*a) -> dir), valor);
     else // se o elemento é igual à raiz
          // não deixa inserir elemento duplicado
          printf("\nValor ja existe... tente outro.\n");
}
```

Observe que o primeiro parâmetro da função `insereArvore`, `a`, é um ponteiro para `Arvore`. E `Arvore` é um ponteiro para um elemento. Ou seja, a árvore está sendo passada por referência (endereço da variável), e não por valor (cópia do valor contido na variável informada). Isso acontece para que a função chamada consiga alterar o valor original, ou seja, que o endereço alocado possa ser armazenado na variável original, e não em uma cópia.

4.3. A busca de um elemento em uma árvore (buscaArvore)

De modo similar à inserção de elemento, a busca de um elemento em uma árvore também é recursiva, parando com as chamadas recursivas se encontrou o elemento ou se a árvore está vazia.
Caso a árvore não seja vazia, o elemento buscado é comparado com a raiz. Se o elemento buscado é menor que a raiz, a função vai chamar recursivamente a busca do elemento na subárvore da esquerda. Se o elemento buscado é maior que a raiz, a função vai buscar recursivamente o elemento na subárvore da direita. E se o elemento buscado não for menor nem maior que a raiz, então ele é igual, e portanto, o elemento foi encontrado.

A função buscaArvore retorna o endereço do elemento buscado, caso ele seja encontrado, ou NULL, em caso contrário. Observe que quando a função faz uma chamada recursiva (está buscando

o elemento em uma das subárvores), ela vai retornar o valor que a busca na subárvore retornar. Se a busca na subárvore encontrar o elemento, retorna o endereço do elemento e se não encontrar, retorna NULL. E é exatamente esse o valor que a função da busca na árvore vai retornar.

O código pode ser implementado em C da seguinte forma:

```
Arvore buscaArvore(Arvore a, int valor)
{
    if (a == NULL){ // Árvore vazia, não encontrou elemento, retorna NULL
        return NULL;
    }
    else if (valor < a -> dado) // Se menor que a raiz, busca à esquerda
        return buscaArvore(a -> esq, valor);
    else if (valor > a -> dado) // Se maior que a raiz, busca à direita
        return buscaArvore(a -> dir, valor);
    else // Nem menor, nem maior, encontrou o elemento. Retorna o endereço do elemento
        return a;
}
```

4.4. O percurso em pré-ordem (preOrdem)

No percurso em pré-ordem a raiz é visitada (processada) antes de visitar os filhos.
Nesse percurso, primeiro a raiz é visitada, e depois, chama-se recursivamente o percurso em pré-ordem para a subárvore da esquerda e para a subárvore da direita.
A visita é qualquer processamento que se deseja fazer com cada elemento da árvore. Para este exemplo, a visita será a impressão do elemento.
O código pode ser implementado da seguinte forma:

```
void preOrdem(Arvore a)
{
    if (a != NULL) {
        printf("\n%d", a->dado);
        preOrdem(a->esq);
        preOrdem(a->dir);
    }
}
```

4.5. O percurso em in-ordem (inOrdem)

No percurso em in-ordem, a raiz é visitada depois de percorrer em in-ordem uma das subárvores e antes de percorrer em in-ordem a outra subárvore. Se a ordem for esquerda, raiz, direita, os elementos da árvore são visitados em ordem crescente, conforme exemplificado no primeiro código. Se a ordem for direita, raiz, esquerda, os elementos da árvore são visitados em ordem decrescente, conforme o exemplo do segundo código.

```
void inOrdemAsc(Arvore a)
{  // Imprime os elementos da árvore em ordem ascendente
    if (a != NULL) {
        inOrdemAsc(a->esq);
        printf("\n%d", a->dado);
        inOrdemAsc(a->dir);
    }
}

void inOrdemDesc(Arvore a)
{  // Imprime os elementos da árvore em ordem descendente

    if (a != NULL) {
        inOrdemDesc(a->dir);
        printf("\n%d", a->dado);
        inOrdemDesc(a->esq);
    }
}
```

4.6. O percurso em pós-ordem (posOrdem)

No percurso em pós-ordem a raiz é visitada (processada) depois de visitar os filhos.
Nesse percurso, primeiro chama-se recursivamente o percurso em pré-ordem para a subárvore da esquerda, depois para a subárvore da direita e por último a raiz é visitada.

O código pode ser implementado da seguinte forma:

```
void posOrdem(Arvore a)
{
    if (a != NULL) {
        posOrdem(a->esq);
        posOrdem(a->dir);
        printf("\n%d", a->dado);
    }
}
```