

# Estrutura de Dados 2

27 de outubro de 2018

Relatório Trabalho Prático

## 1 Identificação do relatório

**Equipe:** Kelvin James de Souza Martins - RA: 1986813  
Rafael Lammel Marinheiro - RA: 1986856

**Curso:** Sistemas de Informação

**Departamento:** DAINF

**Instituição:** Universidade Tecnológica Federal do Paraná (UTFPR).

## 2 Introdução

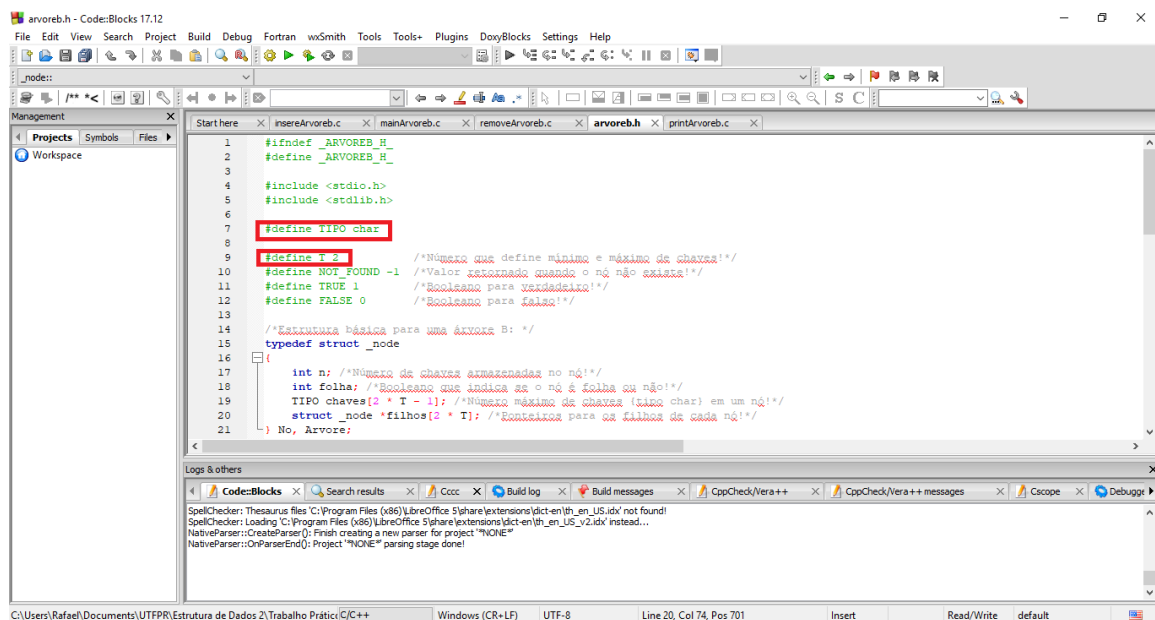
Neste projeto a dupla implementou operações de remoção e inserção em uma árvore B, tal como um arquivo Makefile. A árvore B se define como uma árvore que não precisa acessar o disco diversas vezes, pois cada nó possui um vetor de tamanho máximo  $2 * T - 1$  e mínimo  $T - 1$  sendo T um grau mínimo definido pelo programador. As regras de inserção são baseadas nas regras da árvore binária de busca (ABB), onde uma nova chave deve ser colocada no filho esquerdo sendo menor ou caso contrário no filho direito (isso é um processo recursivo). A árvore B também se destaca na sua complexidade, sendo  $\log N$ , se formos fazer uma comparação com a árvore AVL, uma análise precipitada, poderia ter uma conclusão de que elas não possuem diferença em desempenho, mas a árvore B consegue armazenar um maior número de dados com altura menor, então o N da complexidade também é menor.

Nas próximas seções descreveremos o processo de desenvolvimento com algumas ilustrações das etapas.

## 3 Instruções

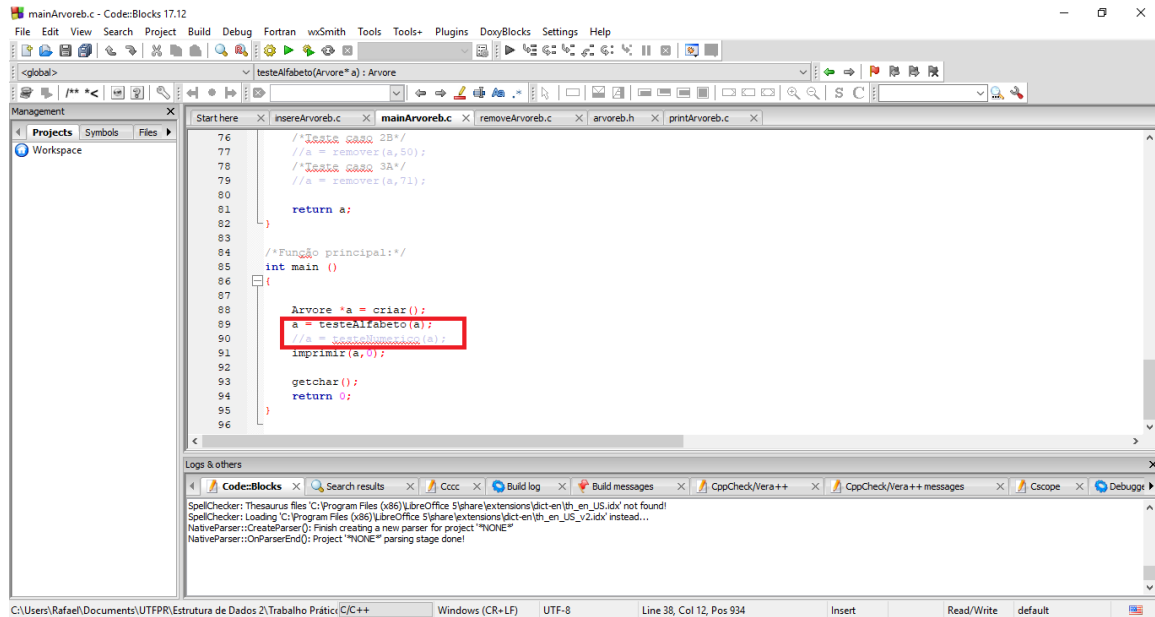
O código é acompanhado de um arquivo Makefile para sua execução. Ele imprime duas árvore B, uma com chaves "int" e grau mínimo 3 e outra com chaves "char" e grau mínimo 2. Ela está configurada para rodar com "char".

Para trocar o tipo e o grau mínimo da árvore, é necessário modificar os arquivos arvoreb.h, mainArvoreb.c, printArvoreb.c e removeArvoreb.c. No arquivo arvoreb.h, altere onde estão circulados em vermelho na imagem para o tipo e grau mínimo desejado.

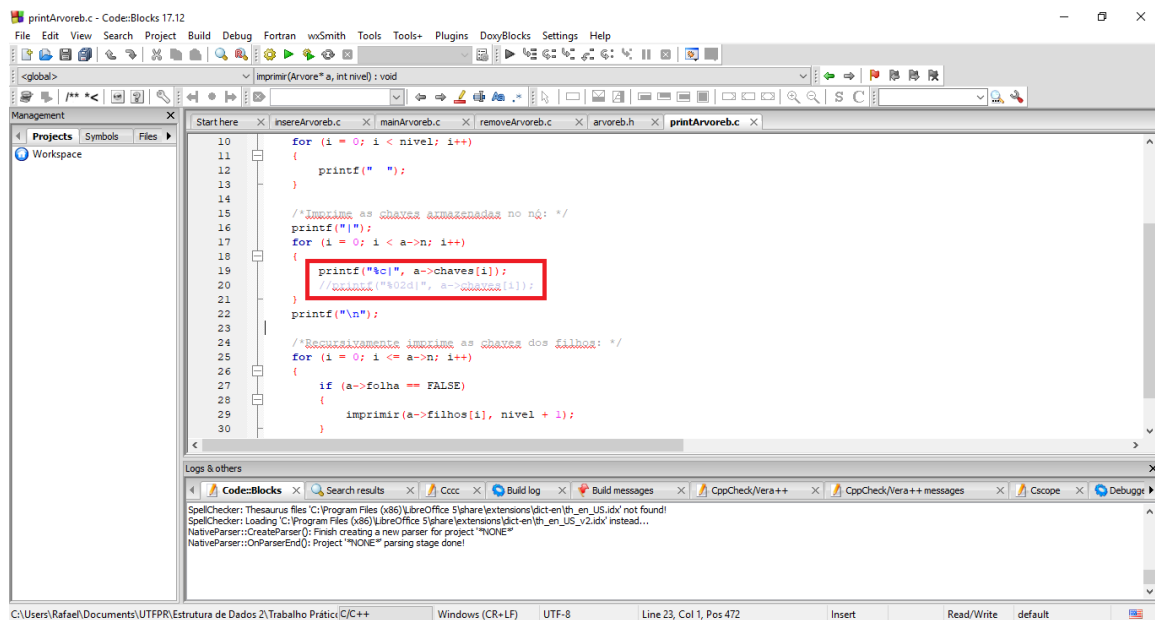


```
1 #ifndef _ARVOREB_H
2 #define _ARVOREB_H
3
4 #include <stdio.h>
5 #include <stdlib.h>
6
7 #define TIPO char
8
9 #define T 2
10 #define NOT_FOUND -1
11 #define TRUE 1
12 #define FALSE 0
13
14 /*Estrutura básica para uma árvore B: */
15 typedef struct _node
16 {
17     int n; /*Número de chaves armazenadas no nó!*/
18     int folhas; /*Booleano que indica se o nó é folha ou não!*/
19     TIPO chaves[2 * T - 1]; /*Número máximo de chaves (tipo char) em um nó!*/
20     struct _node *filhos[2 * T]; /*Ponteiros para os filhos de cada nó!*/
21 } No, Arvore;
```

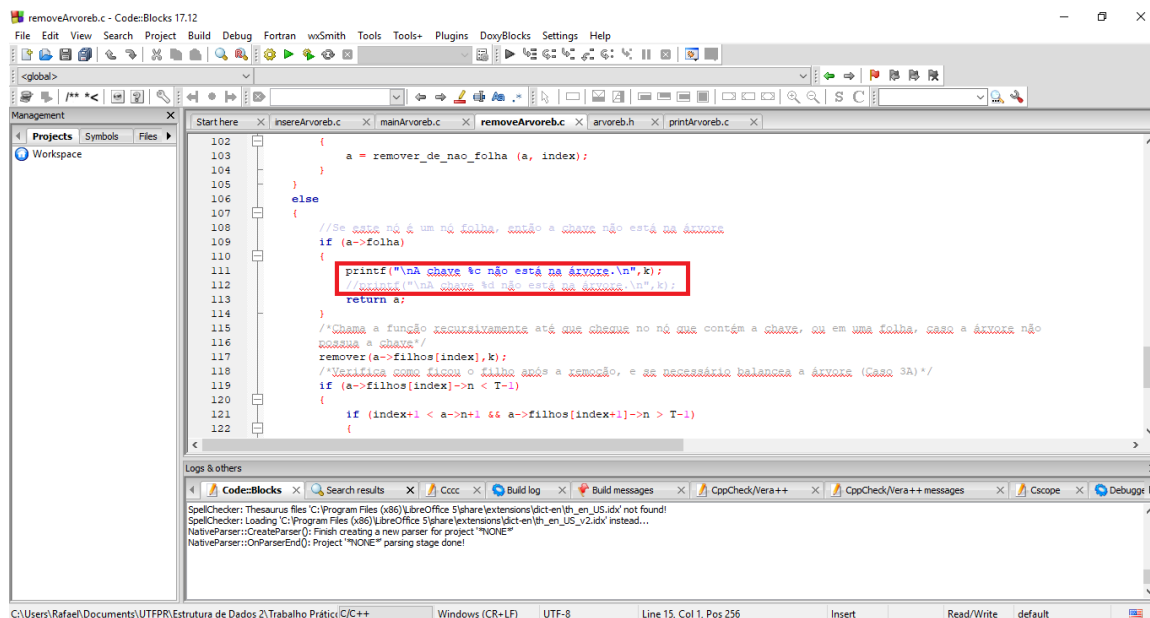
No arquivo mainArvoreb.c, dentro da main, remova os comentários da função equivalente ao tipo que você quer (Numérico = "int", Alfabeto = "char"), como indica a figura abaixo no retângulo vermelho:



Em printArvoreb.c, basta alterar o tipo de saída do printf circulado em vermelho na imagem abaixo para corresponder ao tipo da sua árvore:

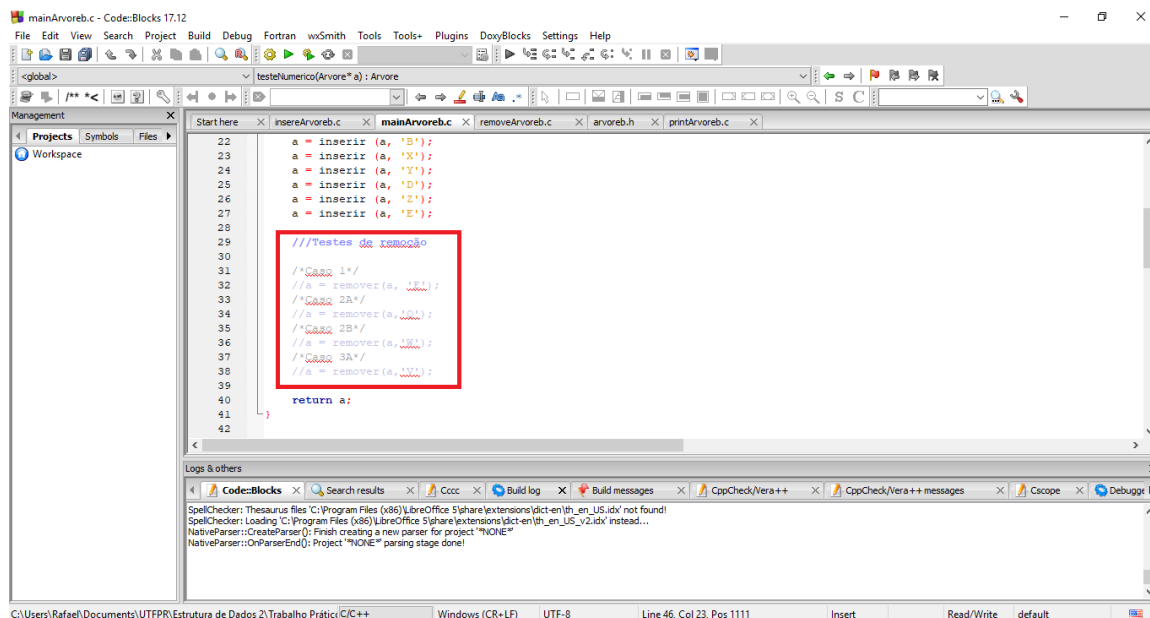


Enfim, para `removeArvoreb.c`, basta fazer a mesma coisa que em `printArvoreb.c`:

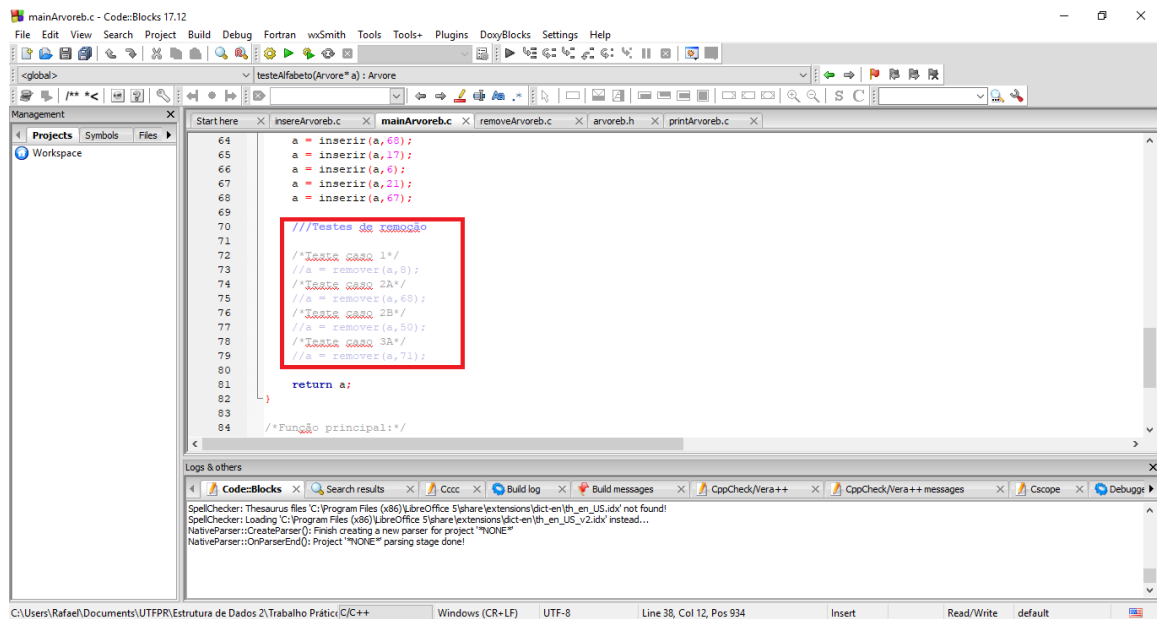


```
102 {
103     a = remover_de_nao_folha (a, index);
104 }
105 }
106 else
107 {
108     //Se essa nã é um nó folha, então a chave não está na árvore
109     if (a->folha)
110     {
111         printf("\\nA chave %c não está na árvore.\\n",k);
112         return a;
113     }
114     /*Chama a função recursivamente até que chegue no nó que contém a chave, ou em uma folha, caso a árvore não
115     possua a chave*/
116     remover(a->filhos[index],k);
117     /*Verifica como ficou o filho após a remoção, e se necessário balanceia a árvore (Caso 3A)*/
118     if (a->filhos[index]->n < T-1)
119     {
120         if (index+1 < a->n+1 && a->filhos[index+1]->n > T-1)
121         {
122             //...
```

Nosso código também vem com alguns testes de remoção. Eles se encontram dentro das funções `testeAlfabeto` e `testeNumerico`, ambas com seus respectivos testes. Para realizar uma remoção, basta tirar do comentário qualquer uma das chamadas de remoção, mas como nosso programa só implementou 4 formas de remoção, pode causar problemas rodar mais do que uma na mesma execução. Recomendamos que se teste uma por vez.



```
22 a = inserir (a, 'B');
23 a = inserir (a, 'X');
24 a = inserir (a, 'Y');
25 a = inserir (a, 'D');
26 a = inserir (a, 'Z');
27 a = inserir (a, 'E');
28
29
30
31 //Testes de remoção
32 //Caso 1*/
33 //a = remover(a, 10);
34 //Caso 2A*/
35 //a = remover(a, 10);
36 //Caso 2B*/
37 //a = remover(a, 10);
38 //Caso 3A*/
39 //a = remover(a, 10);
40
41 return a;
42 }
```



## 4 Makefile

A construção do nosso Makefile foi rápida e teve poucos problemas. conseguimos entender rapidamente a estrutura simples.

[Arquivo a ser gerado] : [Dependências - Arquivos que servem como fonte]  
 [O código que deve ser executado pelo terminal]

Tivemos alguns problemas com o TAB, que precisa necessariamente ter 4 espaços. Nós usamos o editor Atom que tem o TAB configurado como 2 espaços. Alteramos isso, e nosso Makefile funcionou perfeitamente.

Posteriormente tivemos problemas, que a priori achamos que fosse no código, mas logo percebemos que era no Makefile: toda vez que alteravamos o código para testes de chars para ints ou vice versa, o programa não executava da forma esperada. Depois de quase dois dias revisando, percebemos que nosso .h não é compilado, mas ele é usado em todos os nossos .c. Quando mudamos o tipo da variável alteramos o .h, mas apenas alguns .c, fazendo com que alguns .c fiquem com o .h antigo e outros com o novo. Resolvemos isso simplesmente adicionando o .h como dependência de todos os .o, como visto no exemplo abaixo:

```
buscaArvoreb.o: buscaArvoreb.c arvoreb.h
gcc -o buscaArvoreb.o -c buscaArvoreb.c
```

Isso foi tudo que fizemos com o Makefile. Na próxima seção discutiremos sobre a inserção.

## 5 Inserção

A dupla começou o desenvolvimento da árvore pela inserção de elementos, pois já tínhamos a base no livro indicado pela professora. A implementação foi basicamente transformar aquele pseudo-código no nosso código C. Tivemos alguns problemas em relação ao tamanho dos vetores, já que no pseudo-código eles começavam pelo número 1 e na linguagem C começa-se pelo 0.

Nas primeiras versões do nosso código, alguns problemas foram apresentados ao dividir um nó, não verificávamos se o lugar apropriado para o novo elemento era o nó esquerdo ou direito, por consequência disso, todo novo elemento automaticamente ia para o nó esquerdo, demoramos para localizar o erro, porém após identificado, a solução foi elementar.

```
else
{
    //encontrando o filho aonde o K deve entrar.
    while(i>0 && k < x->chaves[i-1])
    {
        i--;
    }
    if(x->filhos[i]->n == (2*T)-1) //Verificar se o filho está cheio.
    {
        dividir_no(x,i,x->filhos[i]);
        if (k > x->chaves[i])
        {
            i=i+1;
        }
    }
    inserir_arvore_nao_cheia (x->filhos[i], k);
}
return x;
}
```

## 6 Remoção

A remoção foi a parte mais complicada desse projeto. Como era necessário escolher 4 das 6 formas de remoção apresentadas, a dupla analisou as formas mais simples para realizar primeiro.

### 6.1 Caso 1

Começamos pelo caso 1, que é quando queremos remover a chave de uma folha que tem pelo menos T chaves. Essa parte foi super simples e a implementação foi feita em poucos minutos. Basicamente chegamos no nó que contém a chave, removemos e rearranjamos o nó se necessário.

## 6.2 Caso 2A e 2B

Quando removemos algo de um nó não folha, verificamos se algum dos dois filhos tem pelo menos  $T$  chaves, se o da esquerda tiver, é feito o caso 2A, onde pegamos o filho máximo(maior elemento da sub-árvore) da esquerda e trocamos com o elemento que será removido, após essa troca, passamos a tratar a remoção no elemento máximo(por conta das propriedades da árvore B, ele sempre estará em um nó folha), no caso 2B, o filho a direita é quem possui pelo menos  $T$  chaves, então ao invés de pegarmos o máximo, fazemos a troca com o mínimo e passamos a tratar a remoção nele, assim como no caso 2A.

Percebe-se que os casos são extremamente parecidos, por esse motivo decidimos que seria melhor deixar ambas na mesma seção para facilitar na descrição.

## 6.3 Caso 3A

Esse caso acontece quando a remoção é feita em um nó folha que possui o mínimo de chaves( $T - 1$ ) e se pelo menos um dos irmãos possuem pelo menos  $T$  chaves, ao remover o elemento, o pai desce para balancear o nó, no irmão pegamos o elemento máximo ou mínimo(depends de qual irmão, esquerdo ou direito) e colocamos no lugar aonde anteriormente o pai estava.

Na hora da implementação, nos atentamos a um caso específico, o elemento que estará sendo removido, pode estar em alguma das extremidades da subárvore, nesse caso, dependendo de qual lado está, verificamos a quantidade de elementos de somente um irmão.