P2 - Estrutura de Dados

Pedro Mian Parra - 2207249

Exercício 1:

A árvore binária é uma estrutura de dados útil para tomar decisões bidirecionais em cada ponto de um processo, sendo definida de forma recursiva. Cada nó pode ser ligado a no máximo dois outros nós que serão chamados de filhos, sendo um à esquerda e outro à direita.

Cada nó formará uma árvore binária, podendo ter um, dois ou nenhum filho. Os nós que não possuem filhos são chamados de folha. A cada nível teremos o dobro de nós do anterior, seguindo uma progressão geométrica.

Na árvore binária de busca, sempre que olharmos para um nó, o seu filho à esquerda terá um valor menor e o seu filho à direita terá um valor maior.

Exercício 2:

a)

O objetivo da tabela hash é facilitar a busca, tornando-a mais rápida e eficiente. É muito utilizada para verificar a integridade de arquivos baixados e para armazenar e transmitir senhas de usuários.

b)

Para resolvermos os problemas de colisões, podemos utilizar o endereçamento aberto ou o encadeamento separado.

No endereçamento aberto, o algoritmo irá percorrer a tabela hash a procura de uma posição ainda não ocupada, evitando o uso de listas encadeadas.

No encadeamento separado, o algoritmo não irá procurar posições dentro da tabela hash. Ele irá armazenar dentro de cada posição do array, o início de uma lista dinâmica encadeada, inserindo nela as colisões ocorridas.

c)

-> Para o endereçamento aberto:

Vantagens: teremos um maior número de posições para a mesma quantidade de memória que seria usada no encadeamento separado, que irá diminuir o número de colisões. Ao invés de acessar os ponteiros, o algoritmo irá calcular a sequência de posições a serem armazenadas.

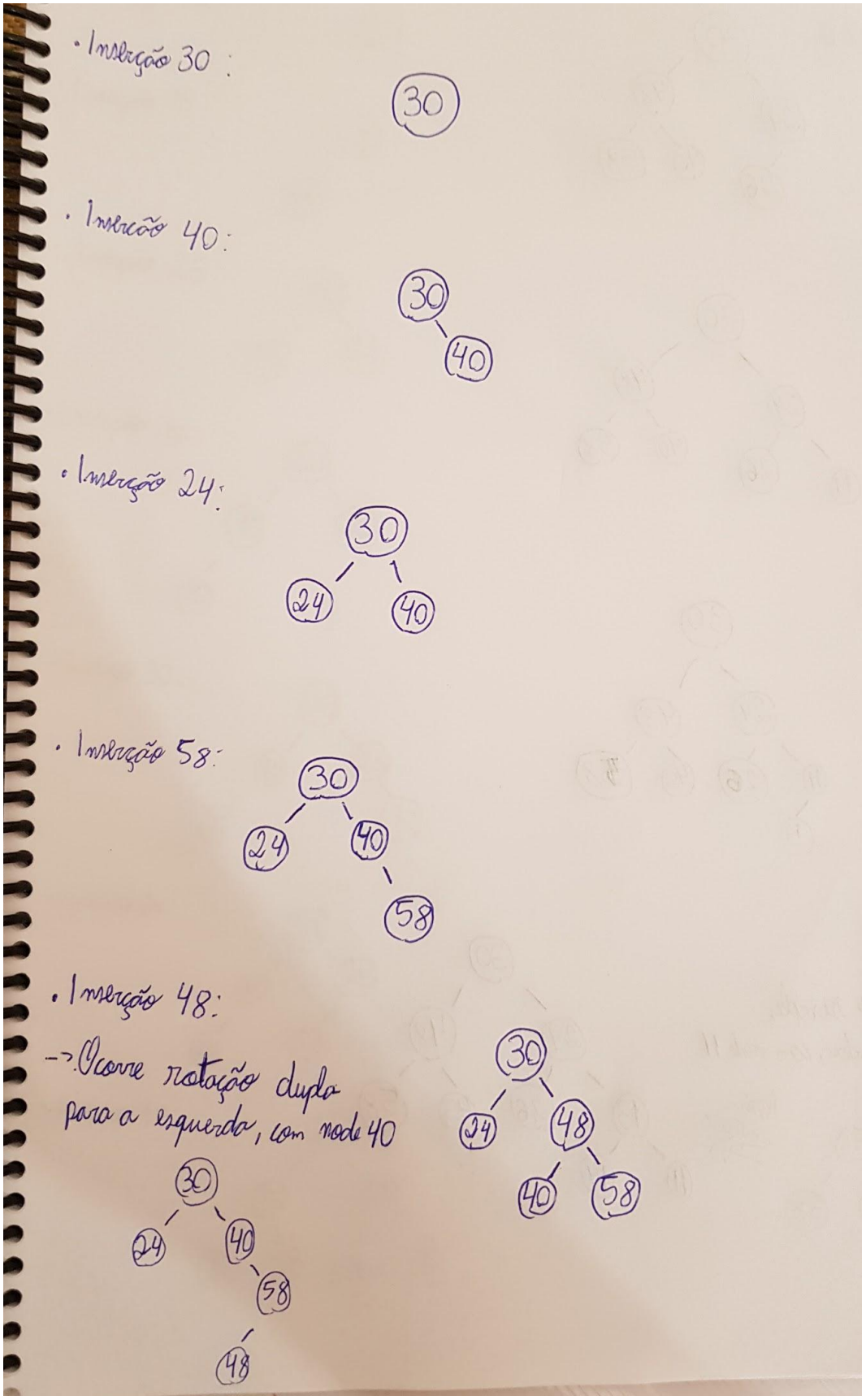
Desvantagens: o algoritmo causará um esforço de processamento maior para realizar o cálculo das posições, causando um maior custo operacional.

-> Para o encadeamento separado:

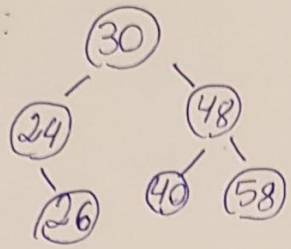
Vantagens: teremos como pior caso a inserção do primeiro elemento, caso a lista não for ordenada.

Desvantagens: a busca levará um tempo proporcional ao número de elementos dentro da lista. Será preciso percorrer a lista à procura do elemento. O algoritmo consome mais memória na sua execução.

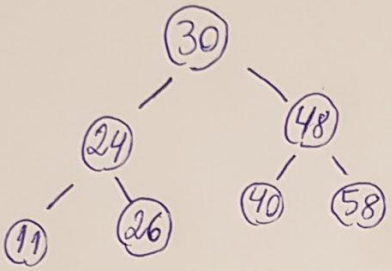
Exercício 3: a)



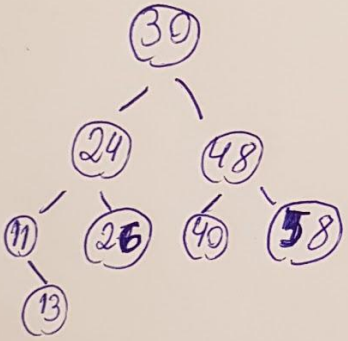
• Inserção 26:



• Inserção 11

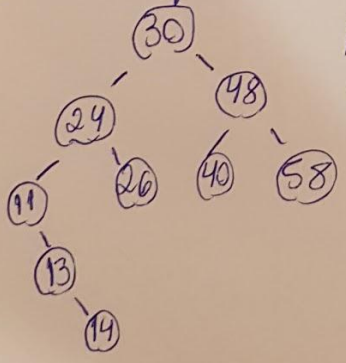


• Inserção 13:

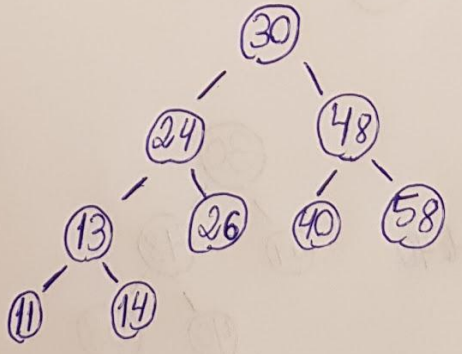


• Inserção 14:

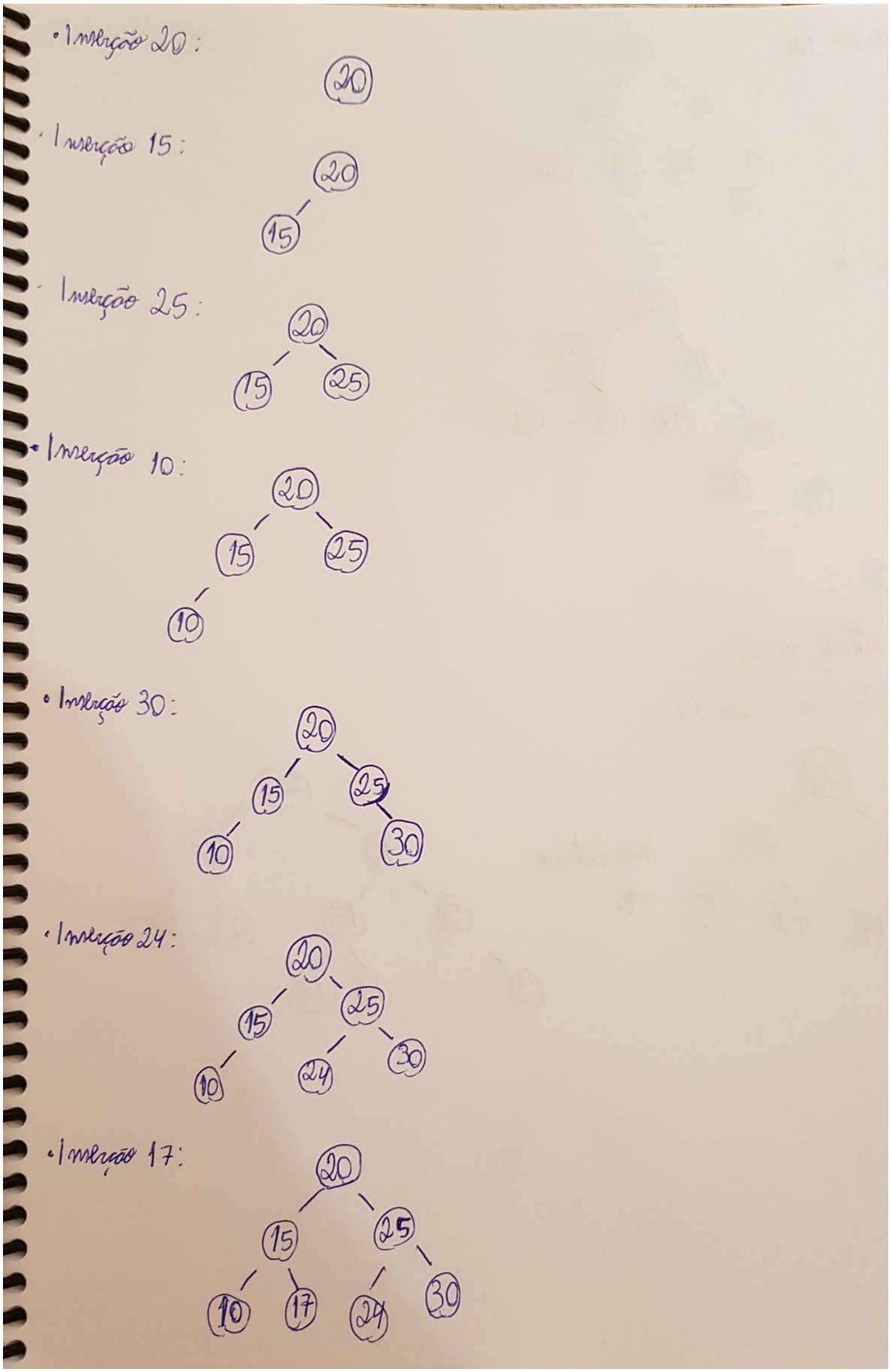
→ Ocorre rotação simples para a esquerda, com nodo 11



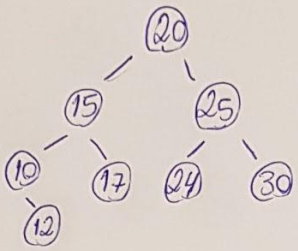
Após rotação



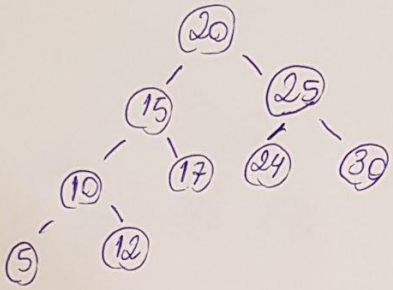
Exercício 3: b)



• Inserção 12:

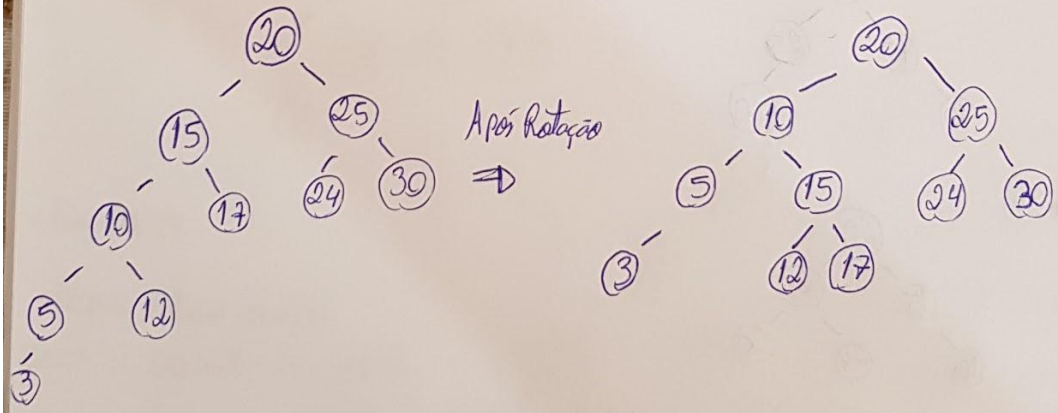


• Inserção: 5

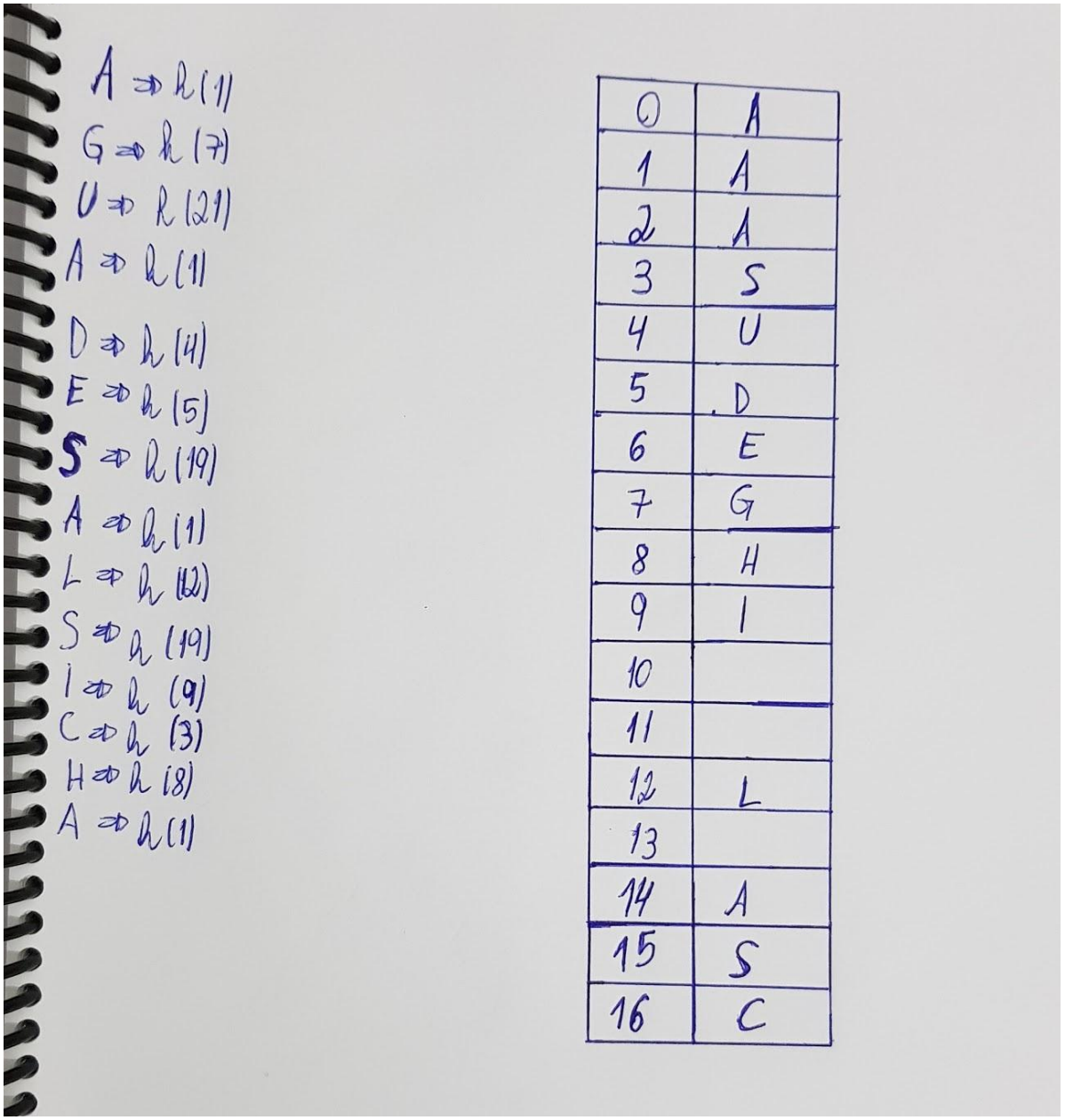


• Inserção 3:

→ Ocorre a rotação simples para a direita, com node 15

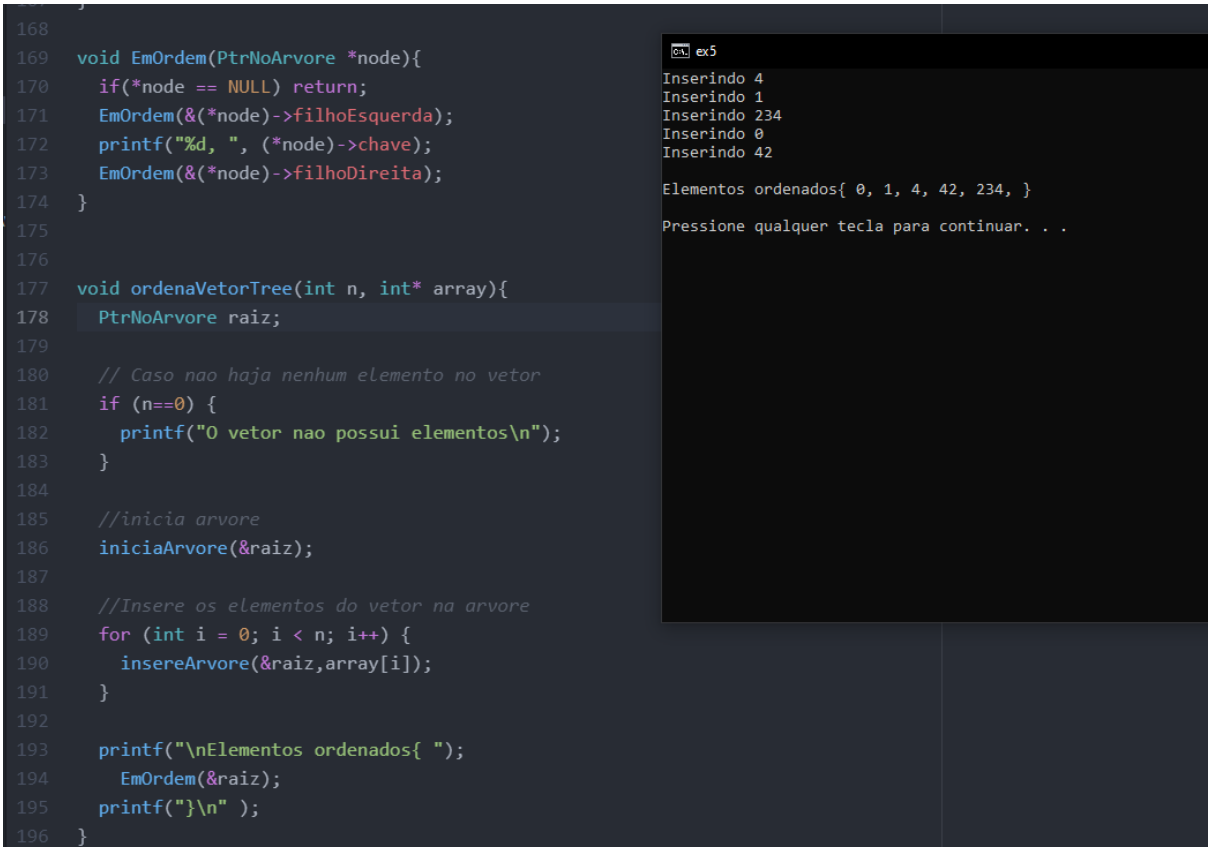


Exercício 4:


A ⇒ R(1)
G ⇒ R(7)
U ⇒ R(21)
A ⇒ R(1)
D ⇒ R(4)
E ⇒ R(5)
S ⇒ R(19)
A ⇒ R(1)
L ⇒ R(12)
S ⇒ R(19)
I ⇒ R(9)
C ⇒ R(3)
H ⇒ R(8)
A ⇒ R(1)
0 A
1 A
2 A
3 S
4 U
5 D
6 E
7 G
8 H
9 I
10
11
12 L
13
14 A
15 S
16 C

Exercício 5:

```
void EmOrdem(PtrNoArvore *node){
  if(*node == NULL) return;
  EmOrdem(&(*node)->filhoEsquerda);
  printf("%d, ", (*node)->chave);
  EmOrdem(&(*node)->filhoDireita);
}


void ordenaVetorTree(int n, int* array){
  PtrNoArvore raiz;

  // Caso nao haja nenhum elemento no vetor
  if (n==0) {
    printf("O vetor nao possui elementos\n");
  }

  //inicia arvore
  iniciaArvore(&raiz);

  //Insere os elementos do vetor na arvore
  for (int i = 0; i < n; i++) {
    insereArvore(&raiz,array[i]);
  }

  printf("\nElementos ordenados{ ");
    EmOrdem(&raiz);
  printf("}\n" );
}
```

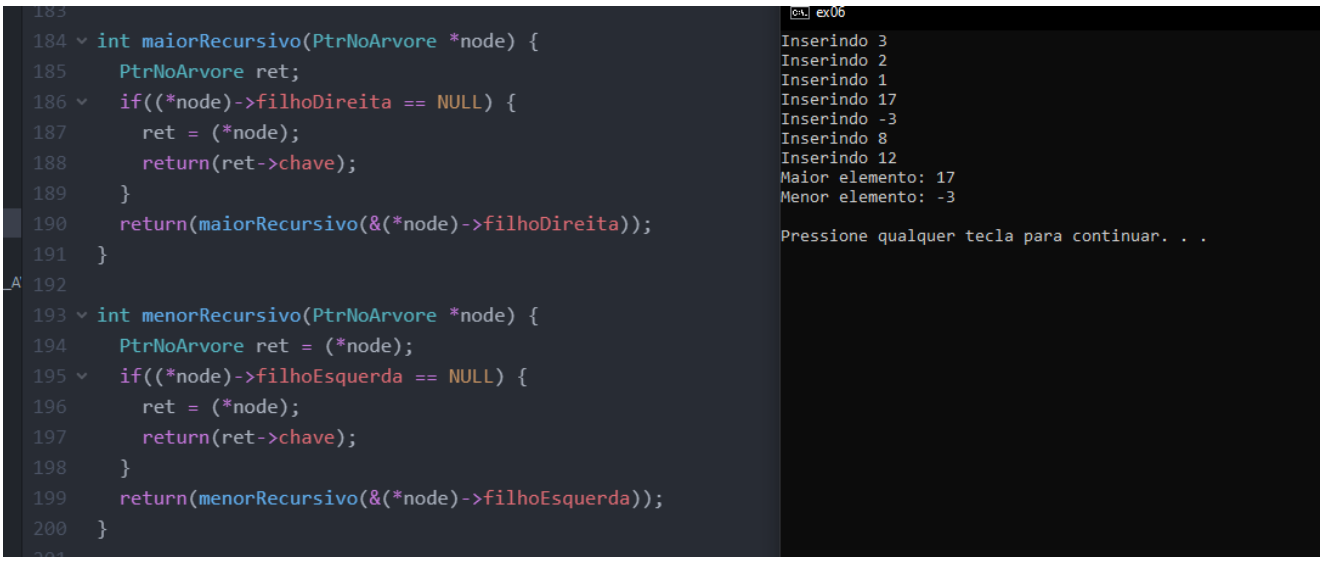


```
168
169  void EmOrdem(PtrNoArvore *node){
170    if(*node == NULL) return;
171    EmOrdem(&(*node)->filhoEsquerda);
172    printf("%d, ", (*node)->chave);
173    EmOrdem(&(*node)->filhoDireita);
174  }
175
176
177  void ordenaVetorTree(int n, int* array){
178    PtrNoArvore raiz;
179
180    // Caso nao haja nenhum elemento no vetor
181    if (n==0) {
182      printf("O vetor nao possui elementos\n");
183    }
184
185    //inicia arvore
186    iniciaArvore(&raiz);
187
188    //Insere os elementos do vetor na arvore
189    for (int i = 0; i < n; i++) {
190      insereArvore(&raiz,array[i]);
191    }
192
193    printf("\nElementos ordenados{ ");
194      EmOrdem(&raiz);
195    printf("}\n" );
196  }
```

```
ex5
Inserindo 4
Inserindo 1
Inserindo 234
Inserindo 0
Inserindo 42

Elementos ordenados{ 0, 1, 4, 42, 234, }

Pressione qualquer tecla para continuar. . .
```

Exercício 6:

```
int maiorRecursivo(PtrNoArvore *node) {
  PtrNoArvore ret;
  if((*node)->filhoDireita == NULL) {
    ret = (*node);
    return(ret->chave);
  }
  return(maiorRecursivo(&(*node)->filhoDireita));
}

int menorRecursivo(PtrNoArvore *node) {
  PtrNoArvore ret = (*node);
  if((*node)->filhoEsquerda == NULL) {
    ret = (*node);
    return(ret->chave);
  }
  return(menorRecursivo(&(*node)->filhoEsquerda));
}
```



```
int maiorRecursivo(PtrNoArvore *node) {
  PtrNoArvore ret;
  if((*node)->filhoDireita == NULL) {
    ret = (*node);
    return(ret->chave);
  }
  return(maiorRecursivo(&(*node)->filhoDireita));
}

int menorRecursivo(PtrNoArvore *node) {
  PtrNoArvore ret = (*node);
  if((*node)->filhoEsquerda == NULL) {
    ret = (*node);
    return(ret->chave);
  }
  return(menorRecursivo(&(*node)->filhoEsquerda));
}
```

```
ex06
Inserindo 3
Inserindo 2
Inserindo 1
Inserindo 17
Inserindo -3
Inserindo 8
Inserindo 12
Maior elemento: 17
Menor elemento: -3

Pressione qualquer tecla para continuar. . .
```

Exercício 7:
EM BRANCO