Nicolas Padre de Lima DRE:118093872

Prova Estrutura de Dados

As questões de código foram feitas em C.

Q1.

Foram omitidas verificações de alocação dos ponteiros, para evitar código muito grande.

```
tno* insere(tno* ptlista, int valor){
        tno* novoNo;
        if(ptlista==NULL){
                //não existe então aloca
                ptlista=(tno*)malloc(sizeof(tno*));
                ptlista->chave = valor;
                ptlista->prox=NULL;
                return ptlista;
        }
        //Verificamos aqui se o valor é maior ou menor que o primeiro da lista
        //Caso seja menor, criamos um novo nó que vai ser o primeiro e
        //o novoNo->prox aponta para o antigo primeiro nó
        //Caso seja maior, chamamos recurssivamente passando o proximo
        //Até que preencha criando um novo nó no local correto e
        //No final retornamos o primeiro nó normalmente
        if(valor < ptlista->chave){
                novoNo = (tno*)malloc(sizeof(tno*));
                novoNo->chave = valor;
                novoNo->prox= ptlista;
                return novoNo;
        }else{//Caso entre no else, o valor é maior que a chave
                //Caso o prox seja nulo, vamos inserir no proximo
                if(ptlista->prox==NULL){
                        novoNo = (tno*)malloc(sizeof(tno*));
                        novoNo->chave = valor;
                        novoNo->prox= NULL;
                        ptlista->prox = novoNo;
                //Caso o proximo exista, chamamos recurssivamente
                //De forma que vamos inserir entre dois nós
                }else{
                        ptlista->prox=insere(ptlista->prox, valor);
                }
        }
        return ptlista;
}
```

Q2.1

Foram omitidas verificações de alocação dos ponteiros, para evitar código muito grande.
Função que retorna toda a árvore preenchida o campo soma
Usei recursão e um inteiro valorAcumulado que vai começar como 0 e ser passado por referência para que possa ser preenchido e incrementado a cada recursão.
O programa faz 3 verificações:

Primeiro if: se estamos em uma folha, preenchemos a soma e acumulamos esse valor.
Caso não tenha entrado nesse primeiro if, significa que tem nós abaixo do nó que estamos
Segundo if: Caso tenha nó à esquerda, chamamos recursivamente soma passando ptraiz->esq.
Quando a recursão voltar, atualizamos a soma para que tenha o valorAcumulado de todos os nós a esquerda e zeramos o valorAcumulado, para calcular valorAcumulado da direita.
Terceiro if: Caso tenha nó à direita, chamamos recursivamente soma passando ptraiz->dir.
Quando a recursão voltar, atualizamos a soma agora somando a chave e o valorAcumulado dos nós à direita, atualizamos o valorAcumulado e retornamos.
Dessa forma, vamos preencher todos os nós com suas somas calculadas corretamente.

```
int valorAcumulado = 0;
tno* soma(tno* ptraiz, int *valorAcumulado){
        if(ptraiz->esq==NULL && ptraiz->dir==NULL){
                ptraiz->soma = ptraiz->chave;
                *valorAcumulado= ptraiz->soma;
                return ptraiz;
        }
        if(ptraiz->esq!=NULL){
                ptraiz->esq = soma(ptraiz->esq, &valorAcumulado);
        }
        ptraiz->soma = *valorAcumulado;
        *valorAcumulado = 0;
        if(ptraiz->dir!=NULL){
                ptraiz->dir = soma(ptraiz->esq, &valorAcumulado);
        }
        ptraiz->soma = *valorAcumulado + ptraiz->soma + ptraiz->chave;
        *valorAcumulado = ptraiz->soma;
        return ptraiz;
}
```

Q2.2

Função simples que insere usando o fato de ser uma árvore de busca, então fazemos comparações do valor com chave.

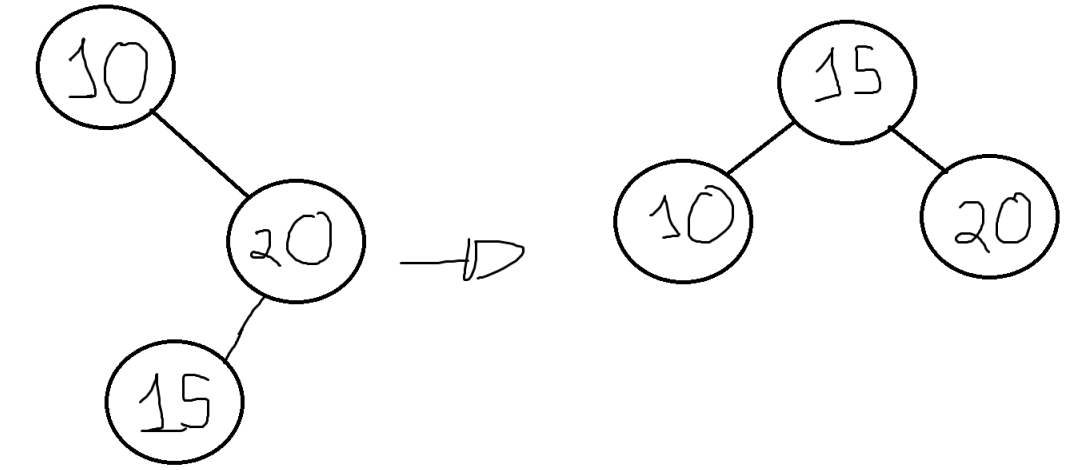
Sempre que inserimos um valor novo, atualizamos a soma dos nós acima dele na volta da recursão, de forma que o campo soma esteja sempre correto.

Foram omitidas verificações de alocação dos ponteiros, para evitar código muito grande.

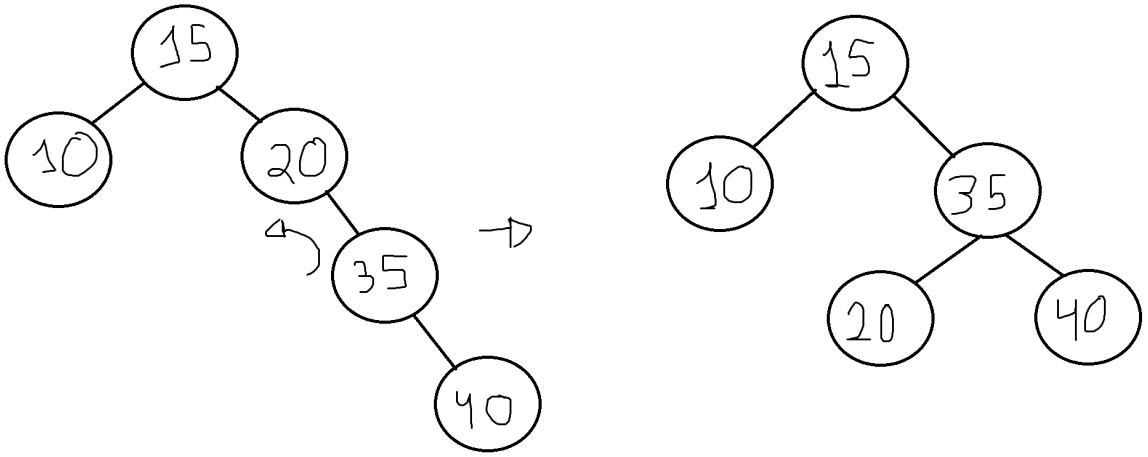
```
tno* insere(tno* ptraiz, int valor){
        if(ptraiz==NULL){
        //Criamos esse nó
                ptraiz=(tno*) malloc(sizeof(tno));
                ptraiz->soma = valor;
                ptraiz->chave = valor;
                ptraiz->esq = NULL;
                ptraiz->dir = NULL;
        }
        //Se valor menor, vamos adicionar à esquerda
        if(valor < ptraiz->chave){
                ptraiz->esq= insere(ptraiz->esq, valor);
                ptraiz->soma=valor + ptraiz->soma;
                return ptraiz;
        }
        //Caso maior, adicionamos à direita
        else{
                ptraiz->dir = insere(ptraiz->dir, valor);
                ptraiz->soma=valor + ptraiz->soma;
                return ptraiz;
        }
        return ptraiz;
}
```

Q3.1

Inserimos 10, 20 e 15, como inserimos 15 em zig zag, rotacionamos duplamente.

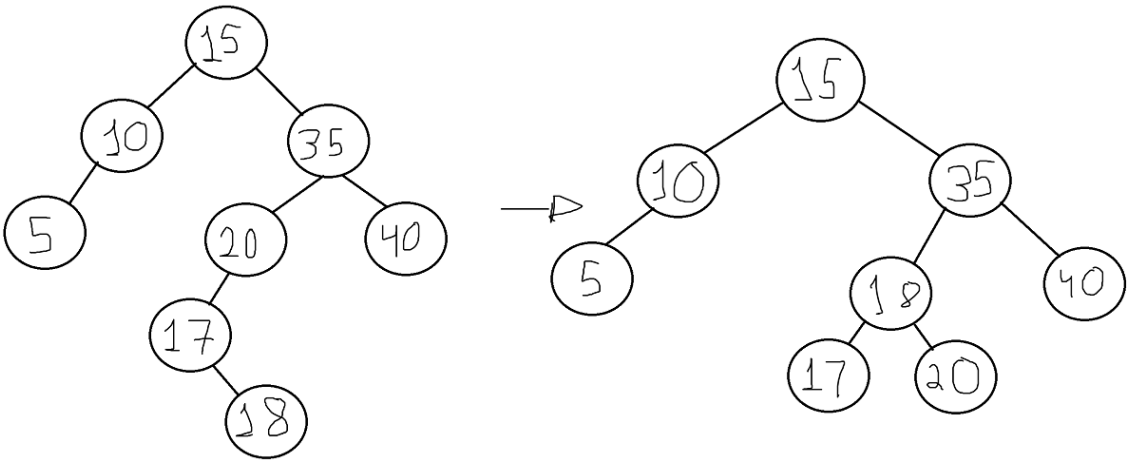


Inserimos o 35 e o 40, o 20 fica desbalanceado então fazemos uma rotação simples para esquerda.

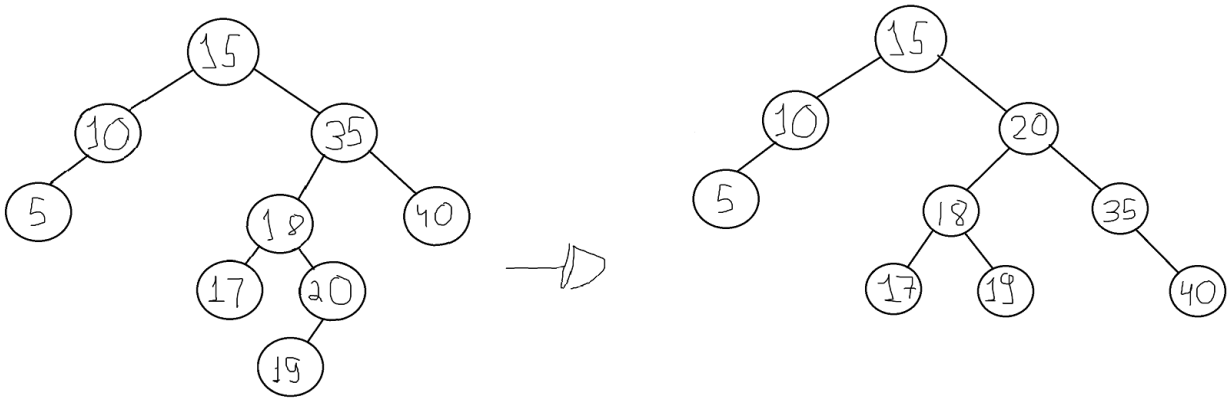


Q3.2

Inserimos o 17 e o 18, como inserimos o 18 em zag zig, fazemos uma rotação dupla.

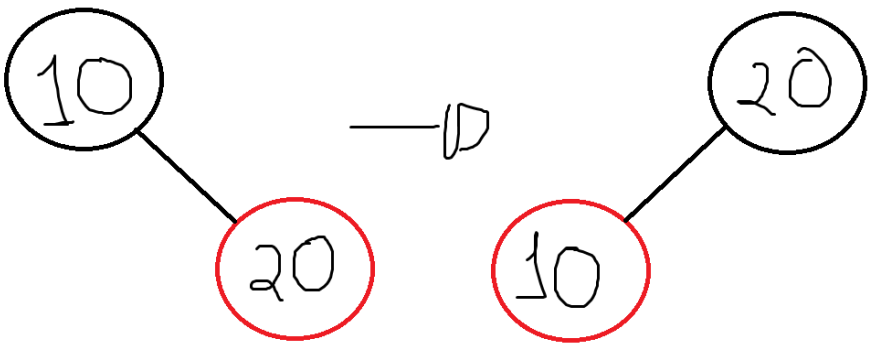


Ao inserir o 19, precisamos arrumar o 35 que está desbalanceado, fazendo uma rotação dupla.

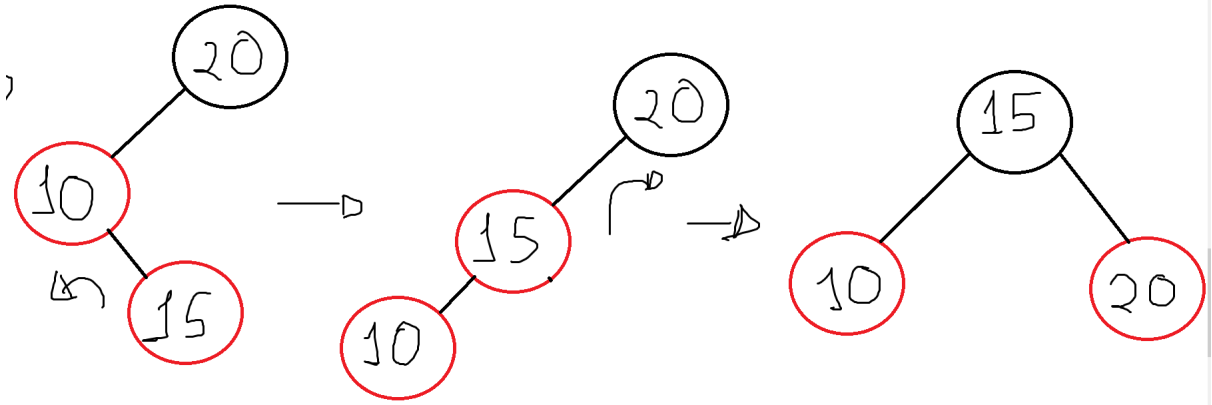


Q4.1

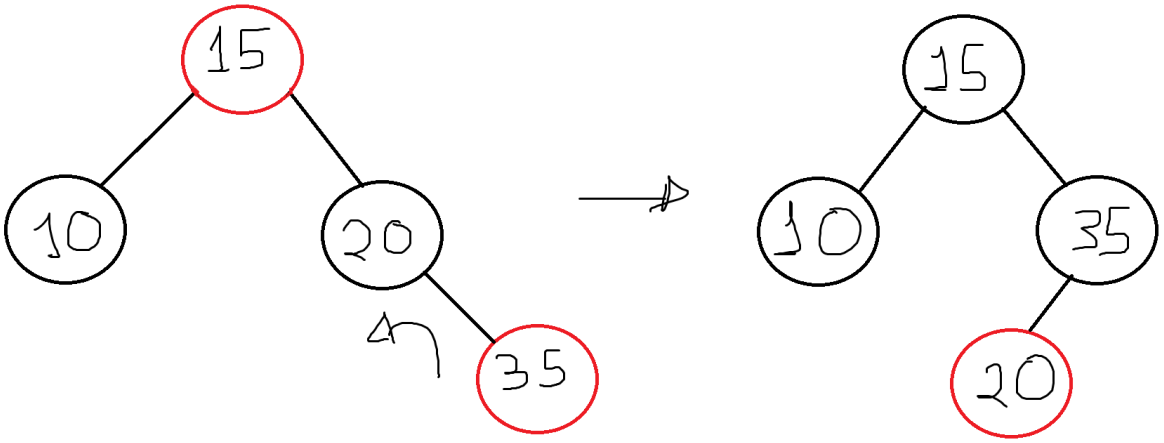
Adicionamos o 10 e o 20, ao adicionar o 20 precisamos rotacionar para esquerda.



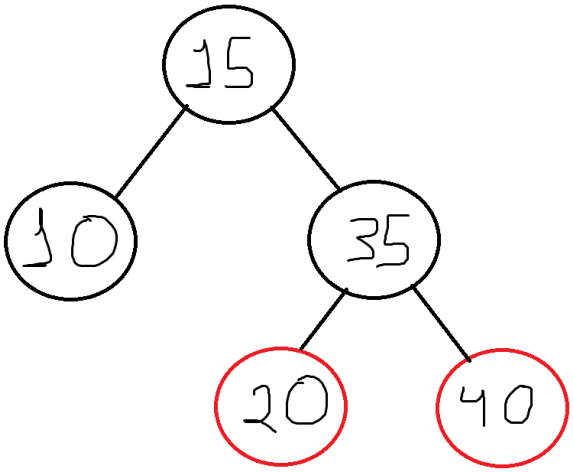
Adicionamos o 15, ao adicionar ele, o 10 fica com um filho rubro à direita, então rotacionamos à esquerda, ao fazer isso o 20 fica com um filho rubro com neto rubro, então rotacionamos à direita .



Inserimos o 35, mas antes trocamos a cor do 15, 10 e 20.
Ao inserir precisamos rotacionar a esquerda, pois o 20 tem filho rubro à direita.
Ao acabar a rotação, alteramos a cor da raiz para negra novamente.



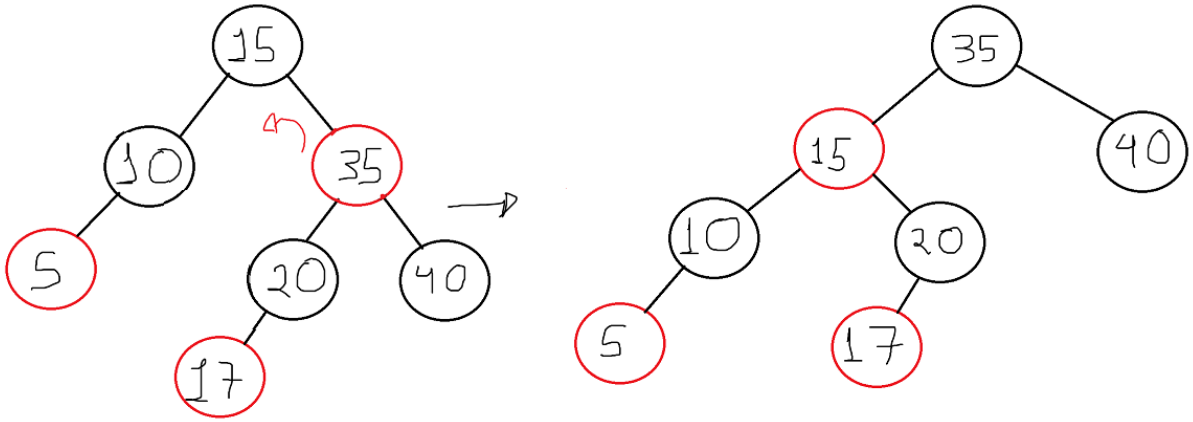
Por fim, inserimos 40 e a árvore termina assim:



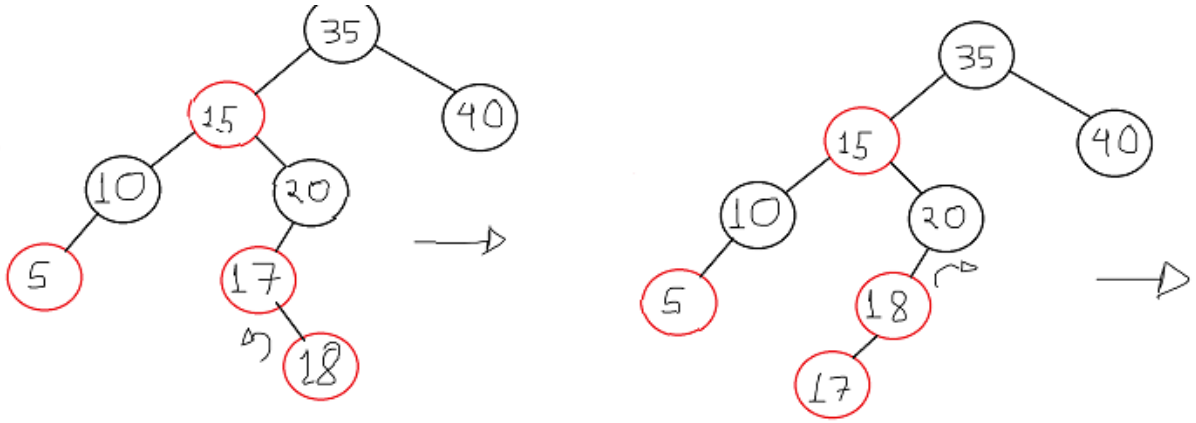
Q4.2

Inserimos o 5, ao inserir o 17, trocamos a cor do 35, 20 e 40

Ao voltarmos pro 15 o filho à direita dele é rubro, então rotacionamos para esquerda.

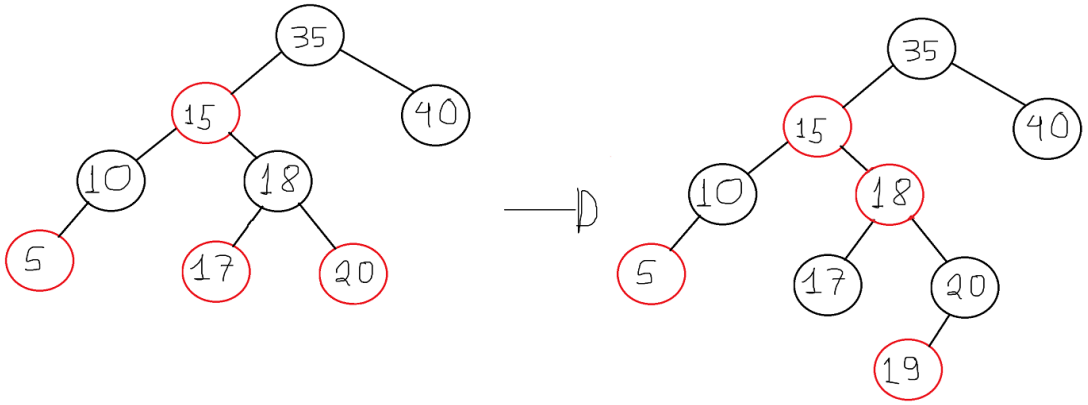


Inserimos o 18, o 17 vai ter filho rubro à direita, então rotacionamos para esquerda.

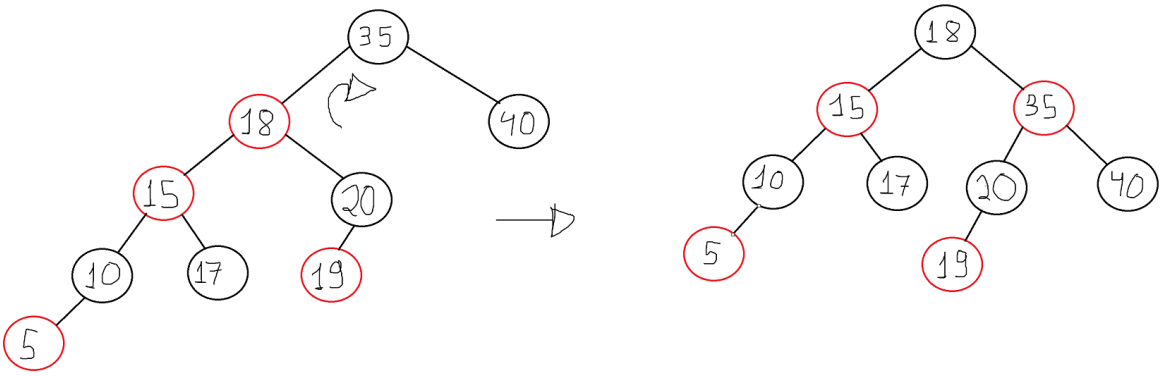


Depois da rotação, o 20 vai ter filho rubro com neto rubro, então rotacionamos para direita.

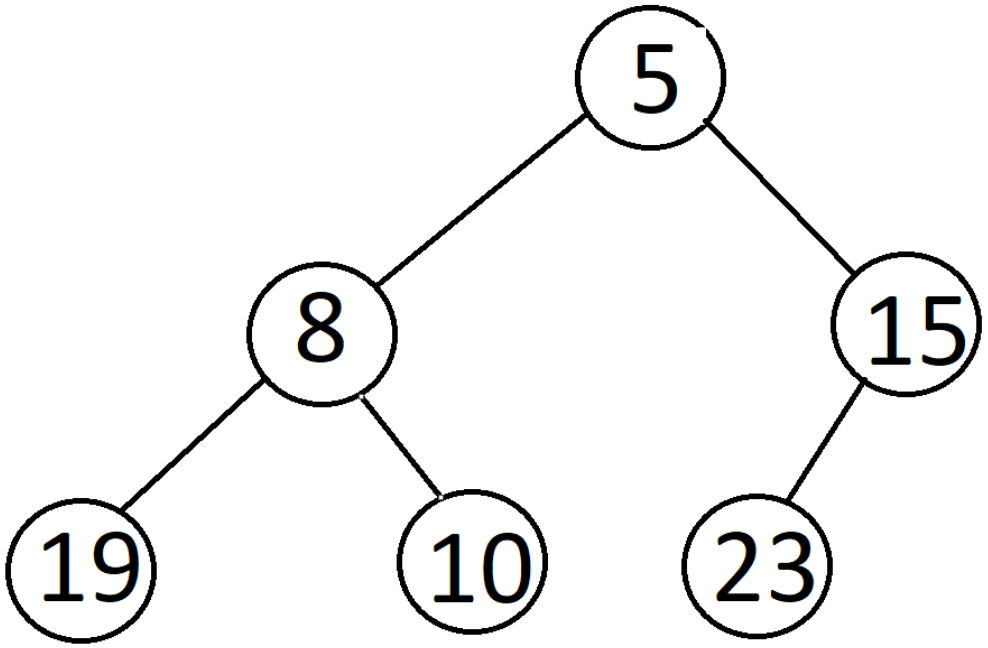
Re-colorimos o 18, 17 e 20 para inserirmos o 19.



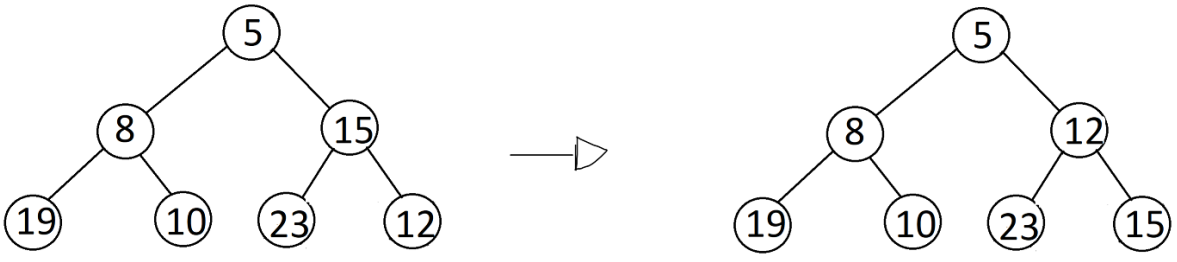
O 15 está com filho rubro a direita, então rotacionamos para esquerda.
Depois de rotacionado, o 35 está com filho rubro com neto rubro, rotacionamos então para a direita, de forma que nossa árvore final fica:



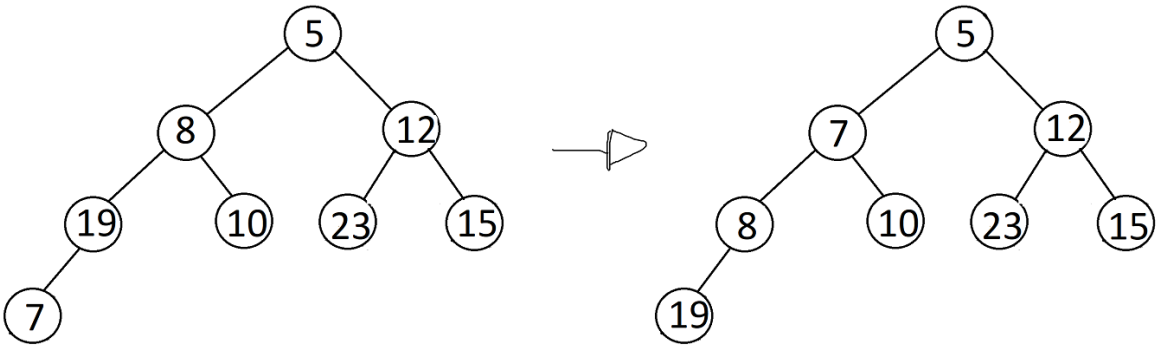
Q5.1
A representação como árvore é:



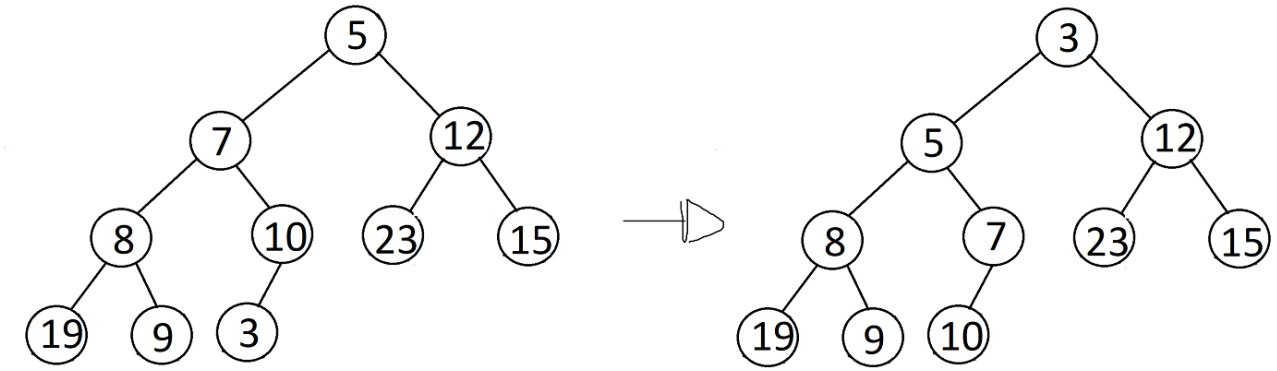
Q5.2
Primeiro inserimos o 12, aí precisaremos subir, pois o 15 é maior que 12, então os trocamos



Inserimos agora o 7, precisaremos subir duas vezes, pois o 19 e o 8 são maiores que 7

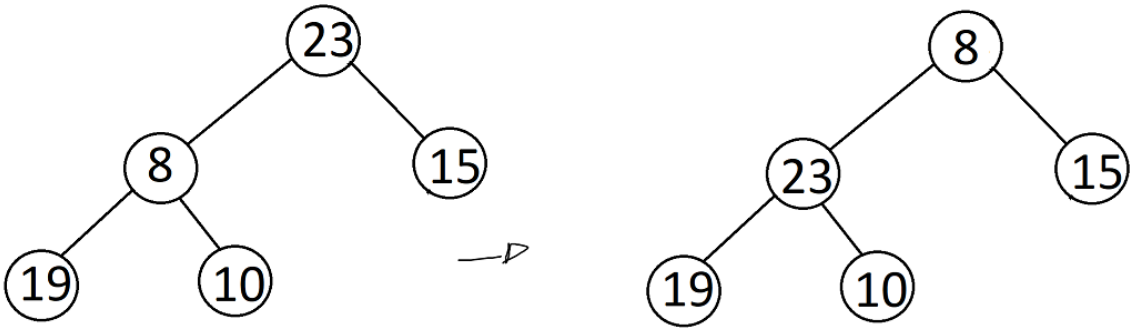


Vamos agora inserir o 9 e o 3, quando inserirmos o 9, não vai dar problema pois o 8 é menor que 9, porém quando inserirmos o 3, precisaremos subir 3 vezes, pois, o 10, o 7 e o 5 são menores que ele. Vamos terminar então com a árvore assim:



Q5.3

Tiramos o 5 e tiramos o 23 e colocamos no lugar dele.
Precisamos agora ir descendo o 2.
Entre 8 e 15, o 8 é menor, então trocamos o 8 e o 23



Entre 19 e 10, 10 é menor, então trocamos o 10 e o 23, resultando na árvore final: