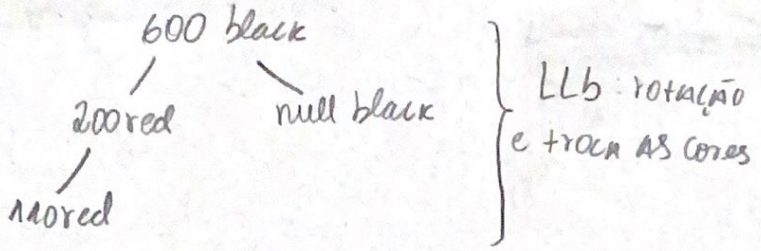
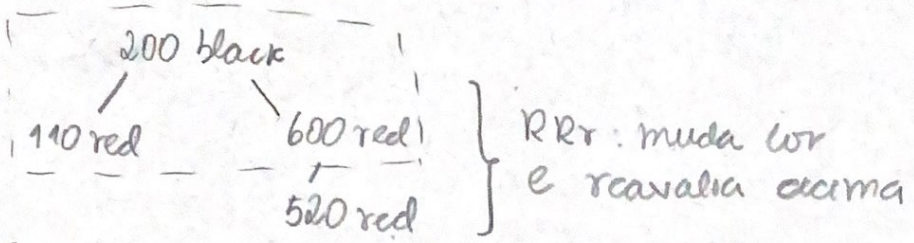
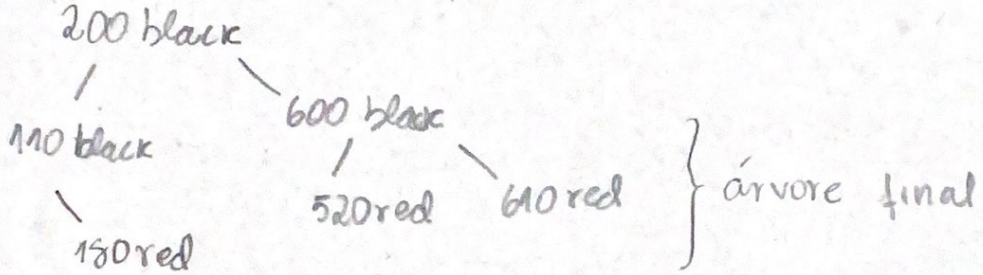
RAFAELA MARIA SOUZA CARNEIRO
MATRÍCULA 2014483

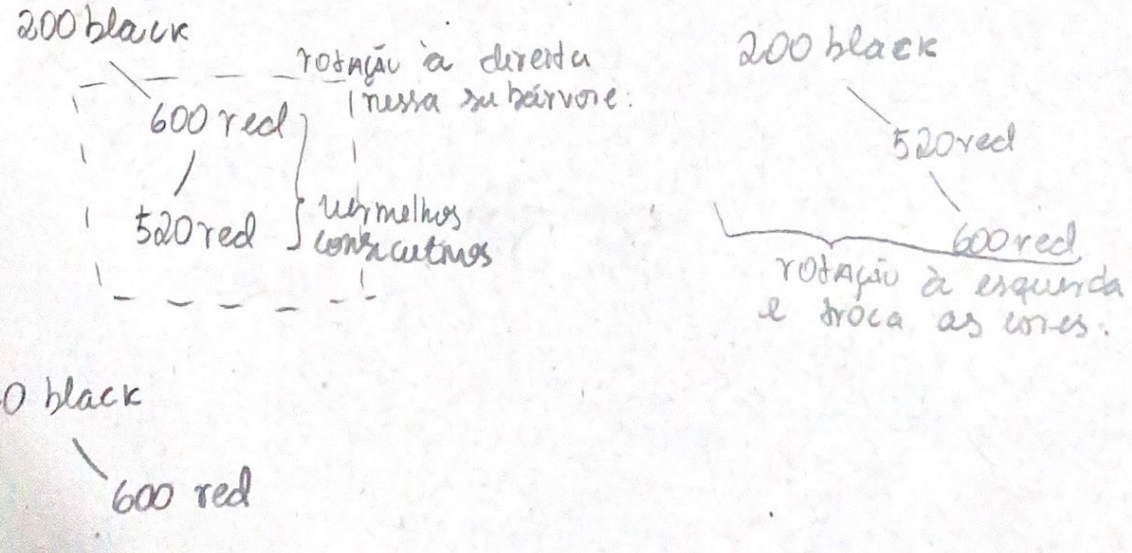
① Inserindo na sequência 600, 200, 110, 520, 180, 610



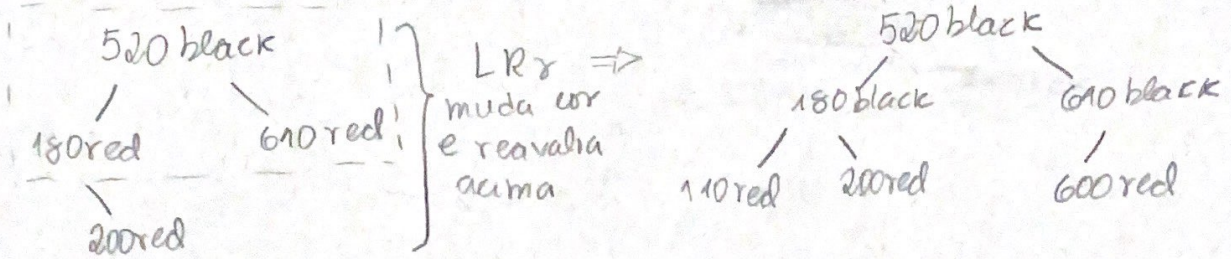




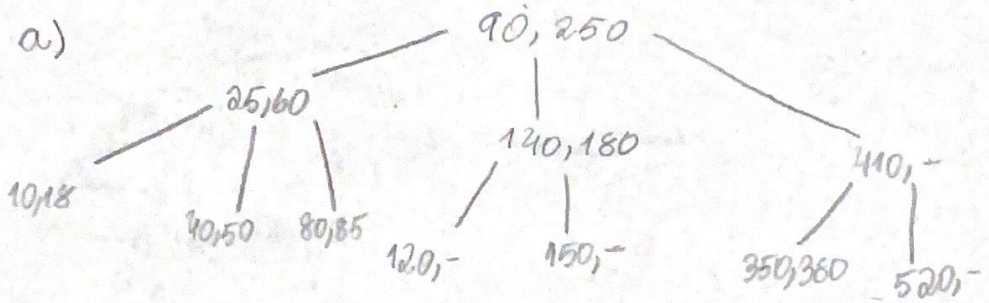
Se inserir a sequência 200, 600, 520 ocorre uma rotação dupla à esquerda:



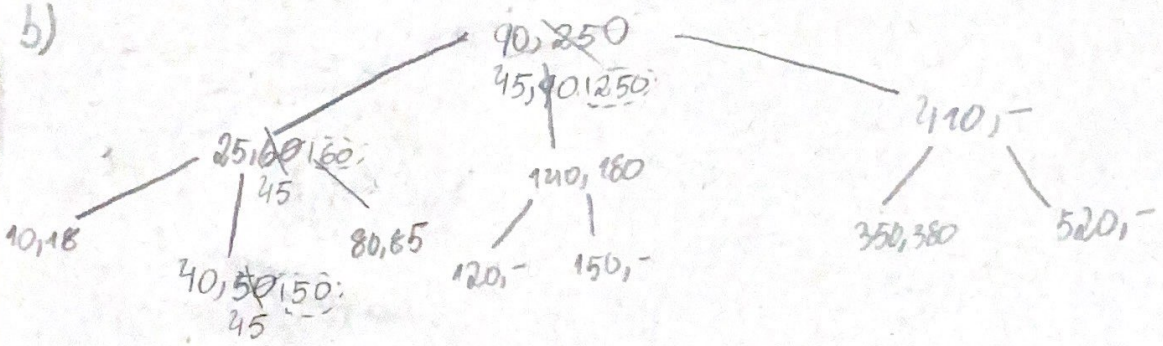
Se inserir as chaves na sequência 520, 180, 610, 200, 110, 600, ocorrerá apenas mudança de cores.



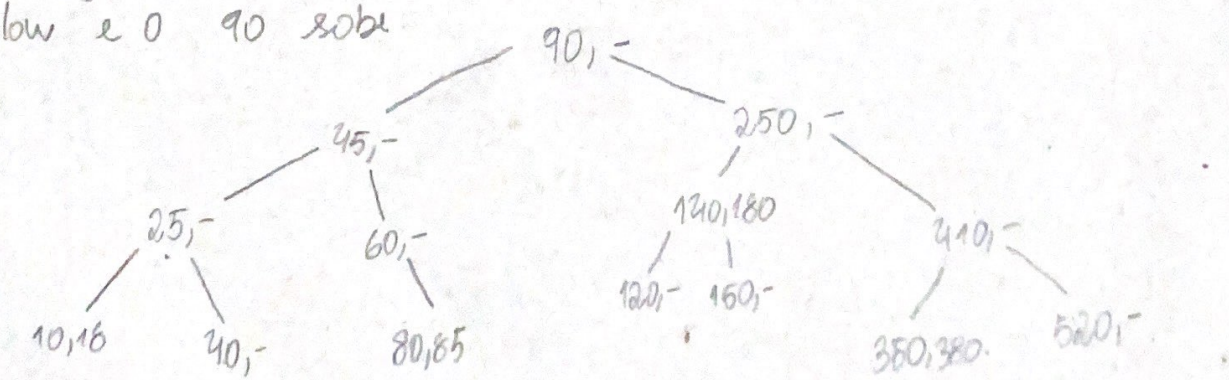
② a)



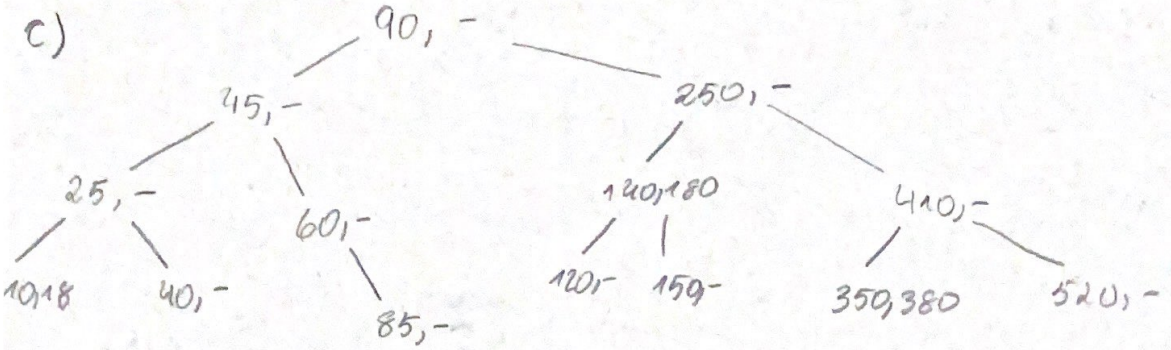
b)



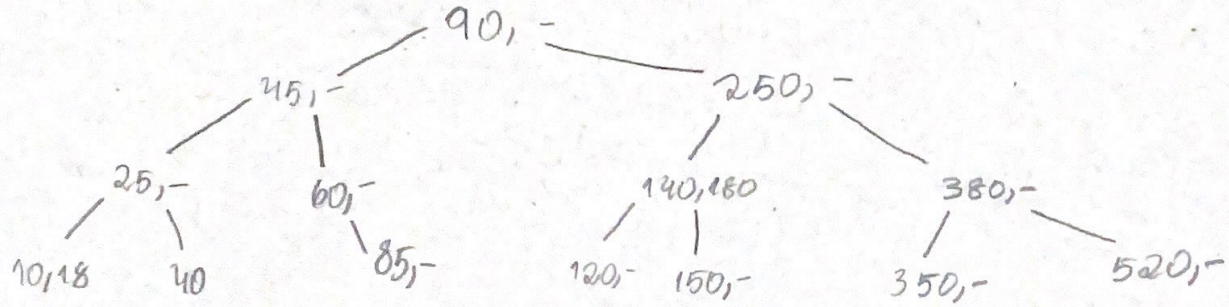
45 entra no lugar do 50, o 50 vai pl área de overflow e o 45 sobe, vai pl lugar do 60, o 60 vai pl área de overflow e o 45 sobe, vai pl lugar do 90, o 90 vai pl lugar do 250, o 250 vai pl área de overflow e o 90 sobe



c)



d) primeiro troca a chave 410 de lugar com a chave 380 e depois remove a 410, ficando com:



③ a)

-1	-1	2	3	13	15	17	25	14	9	10
0	1	2	3	4	5	6	7	8	9	10

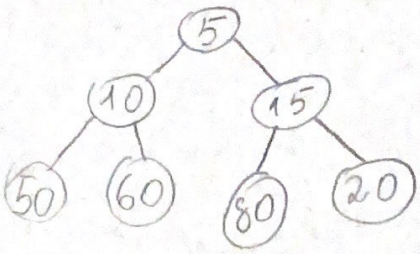
14 15 23 25 14 14
13 13 14 15 25 25 14

• as colisões foram tratadas incrementando o valor de k; a chave 13 teve k = 2; a 14 teve k = 4; a 15 teve k = 2 e a 25 teve k = 4.

b) i. coloca valor 0 (zero) na posição da tabela que se encontra a chave 3. (nesse caso é a posição 3).

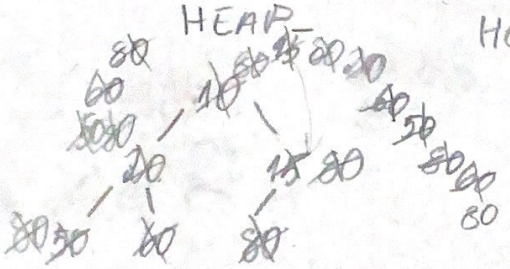
ii. usa a função de dispersão para inserir a chave 4

(4) b) {5, 10, 15, 50, 60, 80, 20} representação como vetor



c) 1) cria vetor com o tamanho máximo do heap;
2) coloca o valor da raiz do heap no vetor;
3) troca o valor da raiz no heap pela última prioridade e tira essa última prioridade do heap;
4) conserta o heap (a partir da raiz, confere se todos os pais são menores que seus filhos - se não forem, é preciso trocá-los de lugar)

Executando esses 4 passos em um laço de repetição até terminar de preencher o vetor ficamos com:



HEAPSORT

5	10	15	20	50	60	80

```c
/* QUESTAO 4 */

#include <stdio.h>
#include <stdlib.h>

struct heap {
    int max;
    int pos;
    int* prioridade;
}; typedef struct heap Heap;

/* FUNCOES AUXILIARES */
static void troca(int a, int b, int* v) {
    int f = v[a];
    v[a] = v[b];
    v[b] = f;
}

static void corrige_acima(Heap* heap, int pos)  // verifica se o pai eh maior que o no atual; se sim, troca eles e sobe verificando ate a raiz; senao, interrompe
{
    while (pos > 0)
    {
        int pai = (pos-1)/2;
        if (heap->prioridade[pai] > heap->prioridade[pos])
            troca(pos,pai,heap->prioridade);
        else
            break;
    pos=pai;
    }
}

static void corrige_abaixo(Heap* heap)      // verifica se o pai eh maior que o menor filho; se for, troca eles e desce conferindo ate o final do heap
{
    int pai = 0;
    while (2*pai+1 < heap->pos)             // enquanto a pos do filho esquerdo for menor do que a proxima posicao vazia
    {
        int filho_esq = 2*pai+1;
        int filho_dir = 2*pai+2;
        int filho;
        if (filho_dir >= heap->pos) filho_dir = filho_esq;              // se a pos do filho direito for maior/igual a 1a pos vazia do heap, entao ele nem existe
        if (heap->prioridade[filho_esq] < heap->prioridade[filho_dir])  // se filho esq < filho direito, trocariamos que trocar o pai pelo filho esq
            filho = filho_esq;
        else                                                            // senao, temos que trocariamos o pai pelo filho dir
            filho = filho_dir;
        if (heap->prioridade[pai] > heap->prioridade[filho])            // se pai > filho, troca pai pelo filho
            troca(pai, filho, heap->prioridade);
        else                                                            // senao, o heap ja ta direito
            break;
        pai = filho;                                                    // desce
    }
}

void heap_insere(Heap* heap, int prioridade)
{
    if (heap->pos < heap->max)
    {
        heap->prioridade[heap->pos] = prioridade;
        corrige_acima(heap, heap->pos);
```

```
        heap->pos++;
    }
    else
        printf("Heap CHEIO!\n");
}

int heap_remove(Heap* heap)     // se o heap nao estiver vazio nova raiz eh a ultima prioridade,
decrementa a nova pos de insercao e retorna a raiz antiga.
{                               // se estiver vazio retorna -1
    if (heap->pos > 0) {
        int topo = heap->prioridade[0];
        heap->prioridade[0] = heap->prioridade[heap->pos-1];        // a raiz passa a ser a
ultima prioridade que foi inserida no heap
        heap->pos--;
        corrige_abaixo(heap);
        return topo;                                                // retorna a raiz
    }
    else {
        printf("Heap VAZIO!");
        return -1;
    }
}

int* heapsort(Heap* heap) {
    int* vetor = (int*) malloc(heap->max*sizeof(int));
    int i;
    for (i = 0; i < heap->max; i++)
        vetor[i] = heap_remove(heap);
    return vetor;
}
/* FIM DAS FUNCOES AUXILIARES*/

int elems_menores(Heap* min_heap, int x) {
    int* vet = heapsort(min_heap);      // o heapsort faz um vetor em ordem crescente com os
valores do heap
    int qtd=0, i;
    for (i=0; i < min_heap->max; i++) {     // como esta ordenado, eh so buscar ate chegar em um
elemento maior ou igual a x
        if(vet[i] >= x)
            break;
        qtd++;
    }
    return qtd;
}

/* INICIO DO PRORGRAMA DE TESTE */


int main() {
    Heap* heap = (Heap*) malloc(sizeof(Heap));
    heap->max = 7; heap->pos = 0;
    heap->prioridade = (int*) malloc(9*sizeof(int));
    int lista[7] = {10, 50, 20, 5, 60, 80, 15};
    int i = 0;
    while (i < 7) {
        heap_insere(heap, lista[i]);
        i++;
    }
    printf("heap:\n");
    for (i = 0; i < 7; i++)
        printf("%d ", heap->prioridade[i]);
    printf("\n");
    int qtd = elems_menores(heap, 17);
    printf("qtd = %d\n", qtd);

    return 0;
```

```
}
```