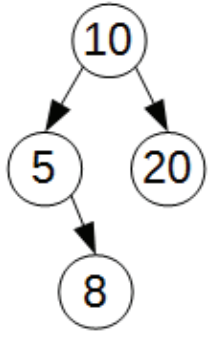
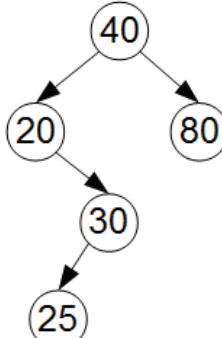
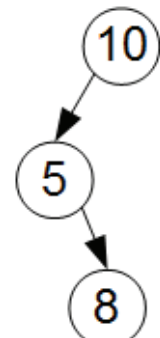
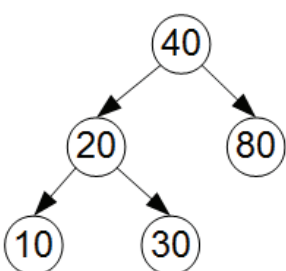
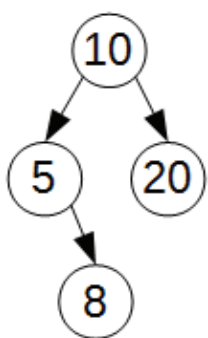
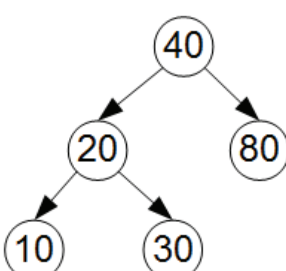


Nome: \_\_\_\_\_ Número USP: \_\_\_\_\_

**Questão 1 (1,5)**

Dadas as diversas árvores abaixo, desenhe a árvore resultante de cada operação ao lado de cada árvore original (não é necessário desenhar figuras intermediárias, apenas a árvore resultante):

<p>a) Insira o nó com chave=9 (inserção em árvore binária de busca)</p>  <pre> graph TD     10((10)) --&gt; 5((5))     10 --&gt; 20((20))     5 --&gt; 8((8))         </pre>	<p>d) Remova o nó com chave=40 (remoção em árvore de busca binária, utilizando maior/máximo a esquerda nas substituições):</p>  <pre> graph TD     40((40)) --&gt; 20((20))     40 --&gt; 80((80))     20 --&gt; 30((30))     30 --&gt; 25((25))         </pre>
<p>b) Remova o nó com chave=10 (remoção em árvore binária de busca, utilizando maior/máximo a esquerda nas substituições)</p>  <pre> graph TD     10((10)) --&gt; 5((5))     5 --&gt; 8((8))         </pre>	<p>e) Insira o nó com chave=35 (inserção em árvores AVL):</p>  <pre> graph TD     40((40)) --&gt; 20((20))     40 --&gt; 80((80))     20 --&gt; 10((10))     20 --&gt; 30((30))         </pre>
<p>c) Insira o nó com chave=9 (inserção em árvore AVL)</p>  <pre> graph TD     10((10)) --&gt; 5((5))     10 --&gt; 20((20))     5 --&gt; 8((8))         </pre>	<p>f) Insira o nó com chave=15 (inserção em árvores AVL)</p>  <pre> graph TD     40((40)) --&gt; 20((20))     40 --&gt; 80((80))     20 --&gt; 10((10))     20 --&gt; 30((30))         </pre>

**Questão 2 (1,6) Dado o seguinte programa:**

```
#include <stdio.h>
#include <malloc.h>
#define true 1
#define false 0
typedef int bool;
typedef enum{esquerdo, direito} LADO;
typedef char TIPOCHAVE;

typedef struct aux{
    TIPOCHAVE chave;
    struct aux *esq, *dir;
} NO, * PONT;

PONT buscarChave(TIPOCHAVE ch, PONT raiz){
    if (raiz == NULL) return NULL;
    if (raiz->chave == ch) return raiz;
    PONT aux = buscarChave(ch,raiz->esq);
    if (aux) return aux;
    return buscarChave(ch,raiz->dir);
}

PONT criarNovoNo(TIPOCHAVE ch){
    PONT novoNo = (PONT)malloc(sizeof(NO));
    novoNo->esq = NULL;
    novoNo->dir = NULL;
    novoNo->chave = ch;
    return novoNo;
}

bool inserirFilho(PONT raiz, TIPOCHAVE
novaChave, TIPOCHAVE chavePai, LADO lado){
    PONT pai = buscarChave(chavePai,raiz);
    if (!pai) return false;
    PONT novo = criarNovoNo(novaChave);
    if (lado == esquerdo){
        novo->esq = pai->esq;
        pai->esq = novo;
    }else{
        novo->esq = pai->dir;
        pai->dir = novo;
    }
    return true;
}

void exibirArvore(PONT raiz){
    if (raiz == NULL) return;
    printf("%c ",raiz->chave);
    exibirArvore(raiz->esq);
    exibirArvore(raiz->dir);
}

int max(int a, int b){
    if (a>b) return a;
    return b;
}

int funcao2(PONT raiz){
    if (!raiz ) return -1;
    int e = funcao2(raiz->esq);
    int d = funcao2(raiz->dir);
    return 1 + max(e,d);
}

int funcao1(PONT raiz){
    if (!raiz ) return 0;
    return 1 + funcao1(raiz->esq) +
funcao1(raiz->dir);
}

void inicializar(PONT * raiz){
    *raiz = NULL;
}

void criarRaiz(PONT * raiz, TIPOCHAVE ch){
    *raiz = criarNovoNo(ch);
}

int main(){
    PONT raiz;
    inicializar(&raiz);
    criarRaiz(&raiz,'a');
    inserirFilho(raiz,'b','a',esquerdo);
    inserirFilho(raiz,'c','a',direito);
    printf("Funcao1 [parte 1]: %i.\n",
funcao1(raiz));
    printf("Funcao2 [parte 1]: %i.\n",
funcao2(raiz));
    exibirArvore(raiz);
    printf("\n");

    inserirFilho(raiz,'d','c',esquerdo);
    inserirFilho(raiz,'e','b',esquerdo);
    inserirFilho(raiz,'f','a',esquerdo);
    inserirFilho(raiz,'g','a',esquerdo);
    printf("Funcao1 [parte 2]: %i.\n",
funcao1(raiz));
    printf("Funcao2 [parte 2]: %i.\n",
funcao2(raiz));
    exibirArvore(raiz);
    printf("\n");

    inserirFilho(raiz,'h','g',esquerdo);
    inserirFilho(raiz,'i','h',direito);
    printf("Funcao1 [parte 3]: %i.\n",
funcao1(raiz));
    printf("Funcao2 [parte 3]: %i.\n",
funcao2(raiz));
    return 0;
}
```

Preencha as lacunas abaixo com o que seria impresso pela execução do programa:

**Funcao1 [parte 1]: \_\_\_\_\_ . Funcao2 [parte 2]: \_\_\_\_\_ .**

**Funcao2 [parte 1]: \_\_\_\_\_ . \_\_\_\_\_**

**\_\_\_\_\_ Funcao1 [parte 3]: \_\_\_\_\_ .**

**Funcao1 [parte 2]: \_\_\_\_\_ . Funcao2 [parte 3]: \_\_\_\_\_ .**

**Questão 3 (2,0)** Assinale cada sentença como verdadeira (V) ou falsa (F). Lembre-se das diferenças entre árvore binária, heap máximo, árvore de busca binária [ou árvore binária de busca], e árvore AVL. Caso não saiba se uma dada sentença é verdadeira ou falsa, sugere-se deixá-la em branco, pois cada alternativa assinalada incorretamente anulará uma alternativa assinalada corretamente.

- [ ] Em uma **árvore binária de busca** sempre é possível fazer busca binária e a complexidade assintótica da busca será, no pior caso,  $O(\log n)$ .
- [ ] Em uma **árvore AVL** sempre é possível fazer busca binária e a complexidade assintótica da busca será, no pior caso,  $O(\log n)$ .
- [ ] Todo **heap máximo** é uma **árvore de busca binária**.
- [ ] Em um **heap máximo** que não tenha elementos repetidos, o elemento de menor valor sempre será encontrado em uma folha (ou na própria raiz, caso o heap contenha apenas um elemento).
- [ ] Em qualquer nó de uma **árvore AVL** a diferença de balanceamento entre a altura da subárvore a esquerda e a altura da subárvore a direita é exatamente 1.
- [ ] Dada uma **árvore de busca binária**, o nó de menor valor desta árvore sempre será uma folha.
- [ ] Toda **árvore de busca binária** completa é também uma árvore AVL.
- [ ] Em uma **árvore AVL** qualquer nó da árvore sempre terá seu valor maior ou igual ao valor de seus filhos.
- [ ] Em uma **árvore binária de busca** que não contenha nós com valores repetidos e que contenha ao menos dois elementos, o nó de maior valor poderá ser a raiz.
- [ ] Um **heap máximo** é uma árvore binária completa ou quase completa.

**Questão 4 (3,5)** Dada uma **árvore binária de busca**, que use as estruturas definidas a seguir, Complete as duas funções solicitadas.

```
#include <stdio.h>
#include <malloc.h>
#define true 1
#define false 0
typedef int bool;
typedef int TIPOCHAVE;

typedef struct aux{
    TIPOCHAVE chave;
    struct aux *esq, *dir;
} NO;

typedef NO* PONT;

PONT criarNovoNo(TIPOCHAVE ch){
    PONT novoNo = (PONT)malloc(sizeof(NO));
    novoNo->esq = NULL;
    novoNo->dir = NULL;
    novoNo->chave = ch;
    return novoNo;
}

PONT maiorAEsquerda(PONT p, PONT *ant){
    *ant = p;
    p = p->esq;
    while (p->dir) {
        *ant = p;
        p = p->dir;
    }
    return(p);
}

PONT buscaNo(PONT raiz, TIPOCHAVE ch, PONT *pai){
    PONT atual = raiz;
    *pai = NULL;
    while (atual) {
        if(atual->chave == ch) return atual;
        *pai = atual;
        if(ch < atual->chave) atual=atual->esq;
        else atual = atual->dir;
    }
    return NULL;
}
```

**PONT buscaBinariaIterativa(PONT raiz, TIPOCHAVE ch)** – função iterativa que recebe o endereço da raiz de uma árvore binária (*raiz*) e uma chave a ser buscada na árvore (*ch*). A função deverá retornar o endereço do nó que possui a chave buscada, caso encontre esse nó, e NULL caso contrário.

**bool excluirNo(PONT\* raiz, TIPOCHAVE ch)** – função que recebe o endereço da variável que contém o endereço da raiz da árvore e uma chave (*ch*) e exclui o nó que contém essa chave (caso ele exista) e retorna *true*; caso contrário retorna *false* (considerando as regras de exclusão em uma árvore binária de busca, utilizando o maior a esquerda em caso de substituição).

**O código a ser completado se encontra na próxima página** (se desejar, ao invés de completar, você pode fazer uma nova implementação, respeitando a assinatura da função). Caso ache que há linhas desnecessárias no código, basta riscá-las (ou deixá-las em branco). **Para cada uma das funções não esqueça de acertar todos os campos necessários.**

```

PONT buscaBinariaIterativa(PONT raiz, TIPOCHAVE ch){
    PONT atual = raiz;
    while ( _____ ) {
        if(atual->chave == ch) return _____ ;
        if(ch < atual->chave) atual = _____ ;
        else atual = _____ ;
    }
    return NULL;
}

bool excluirNo(PONT *raiz, TIPOCHAVE ch){
    PONT atual, no_pai, substituto, maiorEsq, paiMaiorEsq;
    atual = buscaNo(_____, _____, _____);
    if(atual == NULL) return _____ ;
    if(!atual->esq || !atual->dir) { // nó tem zero ou um filho
        if(atual->esq) substituto = _____ ;
        else substituto = _____ ;
        if(!no_pai) { _____ // nó a ser excluído é raiz
            _____ = substituto;
        }else {
            if(no_pai->esq == atual) _____ = substituto;
            else _____ = substituto;
        }
        free(atual);
    } else { _____ // nó a ser excluído tem dois filhos
        maiorEsq = maiorAEsquerda(_____, _____);
        atual->chave = _____;
        if (paiMaiorEsq->esq == maiorEsq)
            _____ = maiorEsq->esq;
        else _____ = maiorEsq->esq;
        free(_____);
    }
    return true;
}

```

**Questão 5 (1,4)** Dada a estrutura de dados apresentada na Questão 4, implemente a função abaixo, considerando Árvores AVL:

**PONT rotacaoLL(PONT p)** função que recebe o endereço do nó p (nó cujo balanceamento estaria com problemas dentro da árvore AVL, valendo -2), faz a rotação LL (quando os nós u e v estão a esquerda de seus pais), e retorna o endereço do nó que será a nova raiz da subárvore (originalmente iniciada por p) após a rotação/rebalanceamento. Notem que: 1o) não há atributo de balanceamento nessa estrutura [então você não deverá mexer nesse atributo, apenas acertar os ponteiros]); 2o) esta função tratará apenas a rotação LL (não tratará nenhuma rotação R [RR ou RL] e nem mesmo a rotação LR; 3o) esta função só deverá fazer a rotação (isto é, acertar os ponteiros) e retornar o ponteiro do nó que será a nova raiz da subárvore inicialmente iniciada por p; considere que uma função principal de inserção já foi executada e chamou sua função apenas para reorganizar o trecho iniciado pelo nó apontado por p.