Algoritmos e Estrutura de Dados II – 2012/1 – Lista III-B (0.5 pontos)

Nome:

Assinatura:

Questão 1. Considere as seguintes estruturas de dados.

```
struct numero_conta {
    int agencia;
    int conta;
};

struct cliente {
    char nome[128];
    int cpf;
    struct numero_conta conta;
    int telefone;
    int cep;
    char endereco[1024];
};
```

Implemente três funções de pesquisa binária como a seguir:

```
int binsearch_cpf(struct cliente *v, int cpf);
int binsearch_nome(struct cliente *v, char *nome);
int binsearch_conta(struct cliente *v, struct numero_conta conta);
```

Estas funções devem fazer uma busca binária no vetor `v` pelas chaves recebidas como parâmetro. Assuma que o vetor `v` está ordenado pela chave usada na busca. Suas funções devem retornar o índice do cliente no vetor `v` ou `-1` se não existir cliente com a chave procurada.

```
int binaria_cpf(struct cliente *v, int cpf) {
    int i, esq, dir;
    if(TAMANHO == 0) { return 0; } /* número de elementos em v */
    esq = 1;
    dir = TAMANHO;
    do {
        i = (esq + dir) / 2;
        if(cpf > v[i].cpf) {
            esq = i + 1; /* procura na partição direita */
        } else {
            dir = i - 1; /* procura na part esquerda */
        }
    } while((cpf != v[i].cpf) && (esq <= dir));
    if(cpf == v[i].cpf) { return i; }
    else { return 0; }
}

int binaria_nome(struct cliente *v, char *nome) {
    ...
        if(strcmp(nome, v[i].nome) > 0) {
            esq = i + 1; /* procura na partição direita */
        } else {
            dir = i - 1; /* procura na part esquerda */
```

```
        }
    } while((strcmp(nome, v[i].nome) != 0) && (esq <= dir));
    if(strcmp(nome, v[i].nome) == 0) { return i; }
    else { return 0; }
}

int binsearch_conta(struct cliente *v, struct numero_conta conta);
    ...
        if(conta.agencia > v[i].conta.agencia ||
                (conta.agencia == v[i].conta.agencia &&
                conta.numero > v[i].conta.numero)) {
            esq = i + 1; /* procura na partição direita */
        } else {
            dir = i - 1; /* procura na part esquerda */
        }
    } while((conta.agencia != v[i].conta.agencia
            && conta.numero != v[i].conta.numero) && (esq <= dir));
    if(conta.agencia == v[i].conta.agencia
            && conta.numero == v[i].conta.numero)) {
        return i;
    } else { return 0; }
}
```

Questão 2. Modifique as funções implementadas na Questão 1 de forma que elas retornem o índice do cliente com a maior chave menor ou igual à chave procurada. Assuma que não existem clientes com chave repetida. Nota: O índice retornado é o índice onde o cliente seria inserido para manter o vetor `v` ordenado.

```
int binaria_cpf(struct cliente *v, int cpf) {
    ...
    } while((cpf != v[i].cpf) && (esq <= dir));
    if(cpf > v[i].cpf) i++;
    return i;
}
```

Questão 3. Defina um tipo abstrato de dados `struct conjunto` e implemente as seguintes funções:

```
/* criar um novo tipo abstrato de dados conjunto */
struct conjunto * cria_conjunto(void);

/* inserir o registro [reg] no conjunto [c] */
void insere_conjunto(struct conjunto *c, struct registro *reg);

/* pesquisa pela chave contida no registro [reg]. retorne um
 * ponteiro para o registro contido em [c] que possui a mesma chave
 * que [reg]. se nao houver registro em [c] com a mesma chave que
 * [reg], retorne NULL. */
struct registro * pesquisa_conjunto(struct conjunto *c, struct registro *reg);

/* remove o registro em [c] que possui a mesma chave que [reg] e
 * retorna um ponteiro para esse registro. se nao houver registro
 * em [c] com a mesma chave que [reg], retorne NULL. */
struct registro * remove_conjunto(struct conjunto *c, struct registro *reg);
```

Nesta questão basta definir uma implementação para suportar as operações listadas. Uma lista encadeada atende os requisitos; neste caso, as funções de pesquisa e remoção terão complexidade $O(n)$, onde n é o tamanho do conjunto.

Questão 4. Dada a implementação do tipo abstrato de dados na Questão 3, indique quantas comparações serão feitas para inserir as seguintes chaves: `q u e s t a o f c i l`.

Questão 5. Implemente uma função de busca sequencial em lista encadeada (ponteiros). Transforme essa função numa função auto-organizante: toda vez que uma busca com sucesso for realizada, coloque o elemento buscado no começo da lista encadeada. Essa heurística tenta colocar os elementos mais populares no começo da lista para reduzir o tempo médio em uma busca com sucesso.

```
struct registro * pesquisa_sequencial(struct lista *lista, int chave) {
    struct no *ant = lista->cabeca;
    while(ant->prox && ant->prox->chave != chave) ant = ant->prox;
    if(ant->prox == NULL) return NULL; /* busca sem sucesso */
    struct no *aux = ant->prox;
    ant->prox = ant->prox->prox;
    if(lista->ultimo == aux) lista->ultimo = ant;
    aux->prox = lista->cabeca->prox;
    lista->cabeca->prox = aux;
}
```

Questão 6. Considere um procedimento de criação de lista auto-organizante como definida na Questão 5 com dois passos. Primeiro, fazemos uma busca por um elemento que queremos inserir (possivelmente movendo-o para o começo da lista). Se o elemento não estiver presente, inserimos o elemento no início da lista. Mostre a ordem final da lista auto-organizante depois da inserção dos seguintes elementos: `q u e s t a o f a c i l`.

`l i c a f o t s e u q`

Questão 7. Implemente uma função de busca binária não-recursiva.

Ver Questão 1.

Questão 8. Modifique uma função de busca binária de forma que ela retorne o número de elementos com a chave procurada. (Não é preciso retornar o índice dos elementos.)

```
int binaria_conta_iguais(struct cliente *v, int chave) {
    ...
    } while((chave != v[i].chave) && (esq <= dir));
    if(chave != v[i].chave) return 0;
    int contador = 1;
    int j = i-1;
    while(chave == v[j].chave) {
        contador++;
        j--;
    }
    j = i+1;
    while(chave == v[j].chave) {
```

```
        contador++;
        j++;
    }
    return contador;
}
```

Questão 9. Encontre os tamanhos (aproximados) de vetores de entrada para os quais busca binária é 10, 100 e 1000 vezes mais rápida que busca sequencial.

Assumindo busca com sucesso, o custo médio de uma busca sequêncial é $N/2$ e o custo médio de uma busca binária é $\lg(N)$. Temos $\lg(2^7) \approx \frac{2^7}{2\times 10}$, $\lg(2^{11}) \approx \frac{2^{11}}{2\times 100}$ e $\lg(2^{15}) \approx \frac{2^{15}}{2\times 1000}$.

Questão 10. Telefones em Belo Hozizonte começam com código 31 e têm 8 números. Implemente um tipo abstrato de dados que permita saber se um número de telefone já existe em $O(1)$. Quando novos telefones são vendidos ou contratos cancelados, seu tipo abstrato de dados deve ser capaz de inserir e remover telefones na base de dados com custo $O(1)$. Seu tipo abstrato de dados deve utilizar no máximo 12500000 bytes de memória (aproximadamente 12 MiB) para qualquer número de telefones na base de dados.

O número máximo de telefones distintos é 99.999.999. Para cada telefone precisamos apenas um bit para representar se ele existe ou não. Use um vetor de bits para armazenar os telefones existentes e indexe o vetor pelo número do telefone.

Questão 11. Considerando as estruturas abaixo, implemente uma função *não-recursiva* de busca em árvore binária de pesquisa. Sua função deve seguir o cabeçalho abaixo.

```
struct arvore {
    struct arvore *esq;
    struct arvore *dir;
    struct registro *reg;
};

struct registro {
    int chave;
    /* outros dados. */
};
```

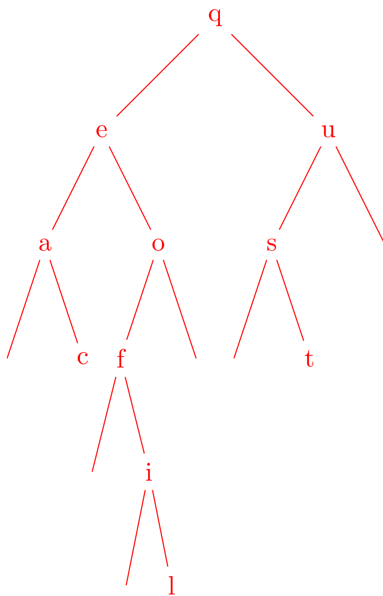
```
struct registro * busca_iterativa(struct arvore *a, int chave) {
    while(a != NULL && a->reg->chave != chave) {
        if(chave < a->reg->chave) { a = a->esq; }
        else { a = a->dir; }
    }
    if(a == NULL) { return NULL; }
    else { return a->reg; }
}
```

Questão 12. Modifique a resposta da Questão 11 para implementar uma função não-recursiva de inserção em árvore binária de busca.

```
/* arrumimos que a função recebe um ponteiro para o ponteiro pra raiz.
 * para inserir o primeiro nó da árvore, basta chamar como a seguir:
```

```
 * struct arvore *raiz = NULL;
 * insere_iterativo(&raiz, registro); */
void insere_iterativo(struct arvore **a, struct registro * reg) {
    while(*a != NULL) {
        if(reg->chave < *a->reg->chave) { a = &(a->esq); }
        else if(reg->chave > *a->reg->chave) { a = &(a->dir); }
        else { printf("chave ja existe.\n"); return; }
    }
    *a = cria_arvore(reg);
}
```

Questão 13. Mostre a árvore binária de pesquisa resultante da inserção dos elementos `q u e s t a o f a c i l` pela função implementada na Questão 12. Mostre também o total de comparações feitas durante o processo.

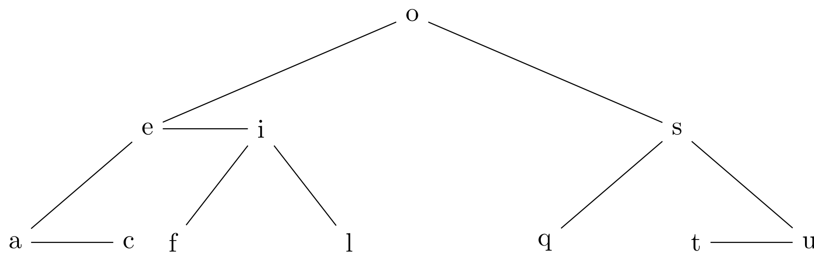


Número de comparações por elemento inserido (em ordem): `0 1 1 2 3 2 2 3 3 3 4 5`, para um total de 29 comparações entre elementos.

Questão 14. Se soubermos a frequência de busca por cada elemento em uma base de dados, devemos começar a incluí-los numa árvore de busca pelos mais ou menos frequentemente buscados? Justifique sua resposta.

Devemos começar inserindo pelos elementos buscados mais frequentemente, pois assim eles ficarão próximos à raiz. Isso diminuirá a média de comparações necessárias nas buscas.

Questão 15. Desenhe a árvore SBB resultante da inserção das chaves `q u e s t a o f a c i l`. Assuma que a árvore não contém chaves duplicadas.



Questão 16. Qual é a altura máxima e mínima de uma árvore SBB com N folhas?

A altura máxima acontece quando a árvore SBB não têm nós horizontais. Neste caso, o i-ésimo nível armazena 2^{i-1} nós e os i primeiros níveis armazenam $2^i - 1$ nós (lembre que SBBs são balanceadas). Se a árvore tem N nós, então $i = \lceil \lg(N+1) \rceil$.

A altura mínima acontece quando todos os ponteiros verticais são seguidos de um ponteiro horizontal. Neste caso, o i-ésimo nível da árvore armazena $3 \times 4^{i-1}$ e os i primeiros níveis armazenam $4^i - 1$ nós. Se a árvore tem N nós, então $i = \lceil \log_4(N+1) \rceil$.

Questão 17. Implemente uma função de *hashing* para cadeias de caracteres com o seguinte cabeçalho: `int hash(char *chave)`.

```
int hash(char *chave) {
    int i;
    unsigned long h = 0;
    for(i = 0; chave[i] != '\0'; i++) {
        h = h * 31;
        h += (unsigned long)chave[i];
    }
    return (int)(h % M);
}
```

Questão 18. Mostre o estado final de um dicionário com resolução de conflitos por encadeamento depois da inserção das chaves `q u e s t a o f a c i l`. Considere que o dicionário não armazena chaves duplicadas, que o valor de cada caractere é sua ordem no alfabeto e que o dicionário possui uma tabela com 5 listas (isto é, $M = 5$).

```
0 u a f
1 q l
2 c
3 s i
4 e t o
```

Questão 19. Repita a Questão 18 para um dicionário onde as listas encadeadas em cada posição da tabela são mantidas ordenadas. O custo de inserção dos elementos depende da ordem de inserção? Justifique sua resposta.

```
0 a f u
1 l q
2 c
3 i s
4 e o t
```

O custo de inserção depende da ordem dos elementos; pois a busca pela posição de um novo elemento na lista ordenada pode demorar mais ou menos tempo dependendo da posição do novo elemento.

Questão 20. Quanto tempo pode levar a inserção de N elementos num dicionário com resolução de conflitos com endereçamento aberto? Justifique sua resposta.

No pior caso pode levar tempo proporcional a $O(N)$, por exemplo se todos os N elementos caírem na mesma posição do dicionário.

Questão 21. Mostre o conteúdo de um dicionário com resolução de conflitos com endereçamento aberto depois da inserção das chaves `q u e s t a o f c i l`. Considere que a tabela do dicionário tem 17 posições e que o valor de cada caractere é sua ordem no alfabeto.

```
 0 a
 1 s
 2 t
 3 u
 4 e
 5 f
 6 c
 7
 8 i
 9
10
11 l
12
13
14 o
15
16 q
```

Ocorreu a penas uma colisão (`c`).