## Descreva três modalidades de função hash

Método da multiplicação
Consiste em escolher um número real entre 0 e 1 e definir h(k) como floor(m*frac(c,k)), onde m é o tamanho da tabela, frac() é a parte fracionada de um número real e flor é a parte inteira. Resultados teóricos mostram que valores de c com boas propriedades são c = 0,6180339887 ou c = 0,381966011.

Método do quadrado médio
Consiste em elevar a chave ao quadrado e selecionar alguns dígitos do "meio" do resultado.

Método de dobra:
funciona bem para Ts = 2^k
Converte-se a chave para a sua representação binária. Digamos que o resultado foi 01011 10010 10110 e que k=5. Faremos uma operação de "ou exclusivo" com os blocos de tamanho k.
01011
10010
10110
01111

## Escreva um código em c para remover uma chave de uma tabela hash ordenada

```
bool remove (int k) {
int i, tmp;
i = h(k);
while(T[i] > k) {
i = rh (i);
}
if (T[i] < k) {
return false;
}
T[i] = -1;
i = rh(i);
while (T[i] != -1) {
tmp = T[i];
T[i] = -1;
insere(tmp);
i = rh(i);
}
return true;
}
```

## Escreva um código em c para inserir uma chave de uma tabela hash ordenada

```
bool first = true;
int i, j, newK, tmp, tk;
i = h(key)
newK = key;
while (T[i] > newK){
i = rh(i);
}
tK = T[i];
while ((tk != -1) && (tk != newK)){
tmp = T[i];
if(newK > tK) {
T[i] = newK;
newK = tmp;
if (first){
j = i;
first = false;
```

```
}
}
i = rh(i);
tK = T[i];
}
if (tK == -1) T[i] = newK;
if (first) return i;
else return j;
```

**Explique os algoritmos de hash extensível e hash linear, ressaltando as semelhanças e diferenças entre os mesmos.**

Hash extensível

A função Hash não varia.

O número de buckets varia.

À medida que os buckets se enchem, estes se duplicam, e o diretório de buckets se duplica.

Se um único bucket duplica, o diretório todo de buckets duplica.

Dois ponteiros do diretório podem apontar para o mesmo bucket,

Somente duplicam os buckets que ficam cheios.

Registros em buckets duplicados podem ser facilmente localizados através do novo ponteiro no diretório de buckets.

Possui uma melhor ocupação dos buckets, apenas divide o bucket apropriado.

Possui um custo I/O que o Hash Linear para seleções com igualdade, em caso de distribuição não uniforme.

Hash Linear

Função hash varia.

Não há diretório de buckets.

Pode haver páginas de overflow a medida que os buckets enchem.

Regularmente, a função hash se modifica e páginas de overflow são realocadas.

Possui um custo I/O menor que do Hash Extensível, para seleções com igualdade, em caso de distribuição uniforme.

Ambos utilizam dos buckets, a diferença está na duplicação do diretório. Na técnica de hashing extensível o diretório e duplicado em um único passo, já no hashing linear os buckets vão sendo duplicados gradualmente. Outra diferença é que no extensível a função de hash não varia. Já no linear, há variação da função hash.

**Explique o que são buckets e porque eles são utilizados quando se trabalha com hash em disco?**

Buckets são estruturas que permitem que um Hash armazene mais de um registro por chave, esse tipo estrutura é utilizada em hash em disco com o intuito de otimizar a estrutura, ou seja não gasta-se recursos operacionais para tratar as colisões

**Argumente porque a altura de uma árvore B com n chaves é h <= logt (n+1) / 2. Sendo que o número mínimo de elementos por nó é t - 1.**

A seguinte formula pode ser deduzida da seguinte maneira:
Como em uma Btree cada nó possui t-1 chaves, logo uma arvore de altura 1 possui 2

nós, já no nível 2 terá 2t nós e no nível 3 terá 2t^2, e na altura h terá 2t(t^h-1)
Logo a altura será logt (n+1)/2

**Descreva com suas palavras os passos necessários para a remoção de uma chave de uma árvore B.**

Primeiro verifica-se se a chave está em um nó interno ou em uma folha. No primeiro caso, o sucessor da chave (próximo elemento sequencialmente) será movido para a posição eliminada e o processo de eliminação procede com a eliminação de um valor de nó folha, que é o seguinte: checa-se se após a remoção o nó terá um número de registros iguais ou maiores à ocupação mínima, se sim, remova o registro. Caso contrário, verificam-se os irmãos à esquerda e à direita do nó do qual fora removido um elemento, se em algum deles há mais elementos do que a ocupação mínima, a chave k que separa os dois nós desce para o nó de onde foi removida uma chave e a chave do irmão é promovida no lugar desta que desceu (caso seja o irmão à esquerda, a chave mais à direita. Caso seja o irmão à direita, a chave mais à esquerda). Agora no caso dos dois irmãos terem exatamente a ocupação mínima, o nó de onde foi retirada uma chave se funde com um de seus irmão mais a chave separadora do pai. Se o pai tiver também a ocupação mínima de um nó deve-se seguir os passos da verificação dos irmãos e proceder recursivamente.

**Dada uma lista generalizada escreva um código que delete todos os elementos cujo campo "info" seja igual a 'h'.**

```
void deleteh(list l){
no ant = null;
no p = first(l);
while (p != null){
if(nodetype(p) = 2 && stecomp("h", info(p) == 0)
if (ant == null)
l = tail(l);
else setnext(ant,next(p));
else{
ant = p;
}
p = next(p);
}
}
```

**Escreva um código para adicionar uma unidade a toda informação inteira de uma lista generalizada**

```
void somarUm(list l){
no p = first(l);
while (p != null){
if(nodetype(p) = 1)
setinfo(p,info(p)+1);
p = next(p);
}
}
```

**Escreva um algoritmo em C para verificar se, dados um Grafo G, representado por uma matriz de adjacências e dois vertices i e j, existe um caminho ligando o vértice i ao j. (Cuidado com loop infinito)**

R:

```
bool existeCaminho(int i,int j, TMatrix M)
{
int n= M[0].lenght;
int k,l;
visitado[i]=true;
for (k=0;k<n;k++)
if((M[i][k]!=0)&&(visitado[k]==false))
{
if (k==j)
return true;
else
return existeCaminho(k,j,M);
}
return false; }
```

## O brasileirão é formado por 20 times em turno e returno. Use seus conhecimentos em teoria dos grafos para determinar o número de partidas de cada turno. Justifique sua resposta.

DESENHAR 20 GRAFOS E CONTAR AS ARESTAS...
Ou também fazer 1 rodada de cada turno e multiplicar pelo numero de rodadas no turno.
Da 190

## Calcular o tempo para ordenar 100 Mega usando merge-sort

Tempo seek 10 ms
Latencia Rotacional 8.3 ms
Taxa transferencia 1229 bytes/ms
buffer de 10000 bytes para escrita
2 mega de ram
Resposta:
1. Fase de Ordenação:
(a) Leitura dos registros para a memória com objetivo de criar as corridas
(b) Escrita das corridas ordenadas para o disco.
2. Fase de Intercalação
(a) Leitura das corridas para a intercala¸c˜ao (Merge-Sort)
(b) Escrita do arquivo final no Disco.
Vamos criar 50 corridas (100/2 = 50)
Run = 20 x (8.3 + 10) = 366 ms = 0,366
Transferencia = 100000/1229 * 1000 = 100/1229 = 81,366 segundos
A-)Tempo total de leitura e criação das corridas = 81,732 ( 0,366 + 81,366)
B-)O tempo total de escrita das corridas é identico, já que as operações são iguais 81,732 s.
C-)Tempo de merge: tem-se 50 corridas de 2 mega cada. Logo cada buffer irá armazenar 1/50 de uma corrida e cada corrida necessita de 50 acessos para ser lida por completo. Isso implica num total de 2500 seeks.
2500 * ( 10 + 8.3 ) = 2500 * (18,3) = 45750 ms = 45,75 s somados com os 81,73 segudos da transferencia de 100 mega temos 127,48 segundos.
D-) 100 mega * 1000000 = 100000000 / 10000 = 10000 seeks ( um seek pra cada vez que o buffer enche)
[10000 * (8.3 + 10 ] / 1000 = 183 segundos
O tempo total é dê A + B + C + D = 81,73 + 81,73 + 127,48 + 183 = 7 minutos e 54 segundos

## O que é aglomeração primaria ? E secundária ? Pq devemos evitar e como evitar.

Aglomeração primária:
Probabilidade do índice: inicialmente, a probabilidade de um índice ser selecionado é igual para todas as posiçõesP(i)=1T.
Aglomeração primária: depois que uma chave é colocada, a probalidade de alguns índices passam a ser maiores quando eles são a próxima chave vazia em uma lista de colisões. O rehash linear gera uma aglomeração primária bem grande.
Solução para aglomeração primária:
Tabela de permutação: fazer uma tabela de permutação que faz o papel do rehash.
Rehash com dois parâmetros: por exemplo: rh(c,j)=(h(c)+√j)%T. Porém gera uma aglomeração secundária.
Novo rehash: várias rotinas diferentes para rehash().
Split: duas rotinas lineares de rehash com duas constantes distintas. Só na primeira colisão, h(k)<kusa a primeira constante, senão usa a segunda constante. O problema é remover uma chave que é a primeira na lista de colisões, não sabemos de devemos usar a constante 1 ou 2.