

ALGORITMOS E ESTRUTURAS DE DADOS

III

2022.2

DISPOSITIVOS DE ARMAZENAMENTO

- ↳ Memória Principal: Mais rápida e eficiente para manipulação de dados
 - ↳ Acessada diretamente pela CPU
 - ↳ Menor capacidade e maior custo
- ↳ Memória Secundária: Possui maior capacidade e é utilizada para armazenar "Grandes" conjuntos de dados
 - ↳ Acessada por meio de Interfaces (CPU não acessa diretamente)

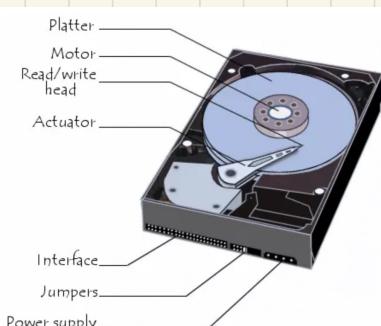
⇒ SSD → Solid State Drive

↳ Componentes Principais:

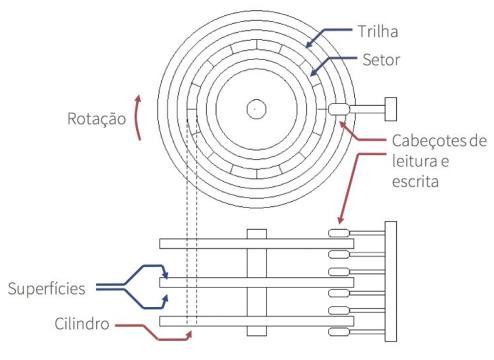
- Memória Flash: Armazena os dados e realiza operações eletricamente (Diretamente na memória)
- Controlador: Gerencia a troca de Dados entre o computador e a memória Flash

⇒ HD → Hard Drive

↳ Realiza operações de forma "Física/Mecânica" e possui diversas partes móveis



- Cada prato é dividido em anéis concêntricos → trilhas
- Cada trilha é dividida em setores → onde os dados são armazenados



- Os cabeçotes se movimentam juntos → todos na mesma trilha

↳ Cilindro: Conjunto de trilhas na mesma posição

↳ Cilindros não existem fisicamente, só são usados para o endereçamento dos dados

- ↳ Endereçamento feito por números sequenciais

↳ Setores possuem um espaço e estrutura definida



• Setor de Disco

↳ Address: Localização Física do setor

↳ Sync: Orientações para leitura do setor

↳ Setores Remapeados: Setores defeituosos substituídos por setores reserva

- ↳ Para manter a capacidade total do Disco

TEMPOS DE OPERAÇÃO

↳ Por o disco rígido ser formado por diversas partes móveis, as operações possuem "tempo ocioso"

- ↳ Posicionamento das partes móveis

⇒ tempo de acesso

↳ tempo necessário para posicionar o cabeçote no setor desejado

⇒ tempo de busca

↳ tempo para posicionamento no cilindro
↳ 2/3 do tempo máximo

⇒ Latência Rotacional

↳ tempo para posicionamento no setor correto
↳ 1/2 rotação

⇒ tempo de transferência

↳ tempo de Leitura e Escrita

$$\text{taxa de transferencia} = \frac{n \cdot t}{60}$$

n → Número de setores

t → tamanho do setor

60 → Velocidade (RPM)

MANIPULAÇÃO EM MEMÓRIA SECUNDÁRIA

⇒ Conceitos Básicos

↳ Entidade: Representação de um objeto / Entidade do "mundo real"

↳ Atributo: Propriedades e características da entidade

- ↳ Representação de um arquivo como um vetor de Bytes com posições sequenciais
 - ↳ Ponto que os dados serão lidos é identificado por um ponteiro
- ↳ Todos os valores a serem escritos num arquivo devem ser convertidos para uma sequência de Bytes
 - ↳ Número de Bytes varia de acordo com o tipo de dado convertido

`Int → 4 Bytes Char → 1 Byte
Float → 4 Bytes Bool → 1 Byte`

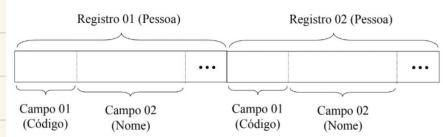
- ↳ Registro: Sequência de Bytes do arquivo que representa uma entidade

- ↳ Registro pode possuir informações "técnicas" sem relação com a entidade

↳ Dados de Criptografia, Metadados, Informações de encadeamento

↳ Campo: Sequência de Bytes para representar um atributo de uma entidade

Jogador	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
<code>int ID = 25;</code>	00	00	00	19	00	08	43	6F	6E	63	65	69	C3	A7	C3	A3	6F	42	47	99	9A
<code>String nome = "Conceição";</code>																					
<code>float pontos = 49.90;</code>																					



- ↳ Bytes de um arquivo não necessariamente representam em registro

↳ É necessário uma descrição de que os Bytes significam

↳ Sem descrição podem significar qualquer coisa

⇨ Conversão Para Bytes

- String

↳ Pode possuir tamanho fixo: Número de Bytes fixo independente do tamanho da string

↳ Uso de Separadores: Caracteres utilizados para marcar o fim de uma string

↳ \n \0 ; \

↳ tamanho: Usa 2 Bytes para armazenar o tamanho da string

- Valores Primitivos

↳ Não devem ser armazenados como string (Dificulta operações numéricas)

↳ Idealmente não devem consumir mais Bytes que o necessário

↳ float, int, bool, long, double

- Codificação de Caracteres

↳ Forma como os caracteres serão representados em um arquivo

↳ Binária ou telta de cada caractere/símbolo

↳ ASCII

↳ Codificação de 8 Bits (ISO)

↳ Unicode: Representa o maior número de símbolos possível de forma universal

↳ Diferentes alfabetos

↳ UTF-8: Unicode com tamanho variável (1/4 Bytes)

↳ Usado na Web

↳ Unicode Transformation Format

- 1 Byte → Tabela ASCII
- 2 Bytes → Latinos, Hebraicos, etc
- 3 Bytes → Asiáticos
- 4 Bytes → Outros Caracteres e Símbolos

TIPOS DE REGISTROS

⇒ Registros de tamanho Fixo

- ⇒ Cada Entidade tem sempre o mesmo tamanho
 - ↳ Posição de inicio de cada entidade é conhecida (x Bytes por entidade)
- ⇒ Pode haver desperdício ou falta de espaço dependendo do atributo armazenado

	ID	NOME		PREÇO
Byte	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27			
	1	'G'	'E'	2.750,00
Byte	28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55	2	'M'	395,00
Byte	56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83	3	'T'	70,00
	int 4 bytes string 20 bytes float 4 bytes			

- Campos podem ter tamanho Fixo ou Variável

⇒ Registro de tamanho Variável

ID; TÍTULO; AUTOR; PREÇO	
Byte	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
	1 7 'P' 'S' 'I' 'C' 'O' 'S' 'E' 9 'R' 'O' 'B' 'B' 'I' 'L' 'O' 'C' 'H' 33,6 2
Byte	32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63
	5 'A' 'F' 'T' 'E' 'R' 9 'A' 'N' 'N' 'A' 'I' 'T' 'O' 'D' 'D' 27,90 3 4 'G' 'R' 'E' 'Y' 22,40
Byte	64 65 66 67 68 69 70 71 72 73 74 75 76 77 78
	9 'E' 'I' 'L' 'J' 'A' 'M' 'E' 'S' 22,40

- Arquivo "se ajusta" ao tamanho de cada campo

↳ Acesso de forma Sequencial (ou aleatório com índices)

↳ Possui um indicador com o tamanho do Registro

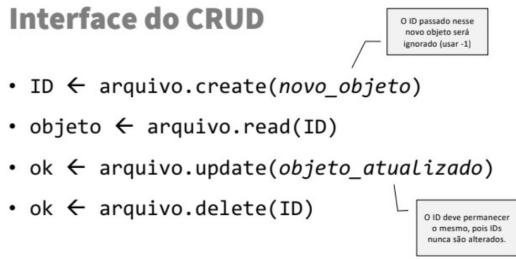
ARQUIVOS

⇒ Arquivos Sequenciais

- ↳ Registros são acessados na ordem que estão armazenados
 - ↳ Não são bons para acesso aleatório
 - ↳ Chave de Ordenação: Atributo utilizado para ordenar o arquivo
 - ↳ Geralmente usados quando sofrem poucas alterações
 - ↳ Precisa ser reordenado eventualmente
 - ↳ Normalmente utilizados como arquivos temporários
 - ↳ Características Identificadoras: Chaves numéricas sequenciais para facilitar reações
 - ↳ Chaves Internas (não-visíveis para o usuário)
- ⇒ Operações em Arquivos → CRUD

Interface do CRUD

- ID ← arquivo.create(novo_objeto)
- objeto ← arquivo.read(ID)
- ok ← arquivo.update(objeto_atualizado)
- ok ← arquivo.delete(ID)



- ↳ Lápis de 1 Byte que indica se o registro foi "excluído"
 - ↳ Exclusão Lógica (não Física)

- ↳ Para Update, caso o registro aumente de tamanho é necessário criar um registro maior no final do arquivo e atribuir uma lápide ao registro atual
- ↳ Algoritmos Básicos para Create, Read, Update e Delete

ORDENAÇÃO EXTERNA

- Ocorre quando os dados a serem ordenados são maiores que a capacidade da memória principal
 - ↳ Ordenação em arquivos
 - ↳ Acesso Sequencial

INTERCALAÇÃO BALANCEADA

- Ordena intercalando registros de várias fontes balanceadas
 - ↳ Arquivos temporários de tamanho aproximado

- I Distribuição → Distribui os Blocos de registros em M caminhos
 - ↳ Caminhos: Arquivos temporários

- Registros são atribuídos a blocos que são ordenados na memória principal e depois enviados aos arquivos temporários
 - ↳ Ordenação com outros Algoritmos (Heap, Quick, Merge)

- II Intercalação → Intercala os registros no arquivo final
 - ↳ Ordenação em si

- Intercalação é feita de forma alternada entre os arquivos temporários

- Arquivo original



- Ordenação pelo ID
 - ↳ Existem outros atributos p/ cada registro

→ Intercalação: Segmentos (Blocos) semi ordenados
São intercalados em outro segmento ordenado
↳ Gera um segmento ordenado com o dobro
do tamanho do segmento-base

→ São feitas diversas intercalações até que
se tenha um arquivo com todos os registros
ordenados

⇒ Análise de Complexidade → "Passadas"

→ Tem como base o número de leituras de todos
os registros

$$1 + \lceil \log_m \left(\frac{N}{b} \right) \rceil$$

N → N° de registros
b → tamanho do bloco
m → N° de arquivos temporários

→ Etapa de Distribuição

→ Algoritmo pode ser otimizado

→ Atributo

⇒ Segmento de tamanho variável

→ Visa diminuir o custo de ordenação entre
os blocos

→ tamanho do segmento só para aumentar
quando o primeiro número do bloco seguinte for
menor que o último número do segmento
ordenado

Arq.Temp 1:	5	10	28	29	31	32	37	40	3	13	15	30	1
Arq.Temp 2:	2	7	12	35	8	9	38	49	17	18	36	39	46
<hr/>													
Distribuição	2	5	7	10	12	28	29	31	32	35	37	40	
Arq.Temp 3:	2	5	7	10	12	28	29	31	32	35	37	40	
Arq.Temp 4:													

→ 3 é menor que o 40

↳ Muda de Arquivo

⇒ Seleção por Substituição

↳ Gera segmentos ordenados maiores na fase de distribuição

↳ Fila/Heap de prioridade

↳ Para Inserção

- Heap Normal / Min-Heap: Menor elemento na Raiz
- Insere na Raiz
- Remove da raiz e insere no segmento

Min-Heap x Árvore Binária

↳ No min-heap o único requisito é que o nó-pai seja menor que os filhos

↳ Não há relação entre os filhos (menor não necessariamente está à esquerda)

↳ Estrutura de Heap pode ser representada em vetor

↳ Mais simples na implementação

- Relação: Filho Esg. = $i \cdot 2 + 1$

↳ Retorna o índice

$$\text{Filho Dir.} = i \cdot 2 + 2$$

$i \rightarrow$ Índice no vetor

$$\text{Pai} = \text{int}\left(\frac{i-1}{2}\right)$$

↳ tamanho do Heap é dado pela capacidade de ordenação da memória principal

→ Nós além de possuir o valor possui o número do segmento ordenado

↳ Serve como prioridade

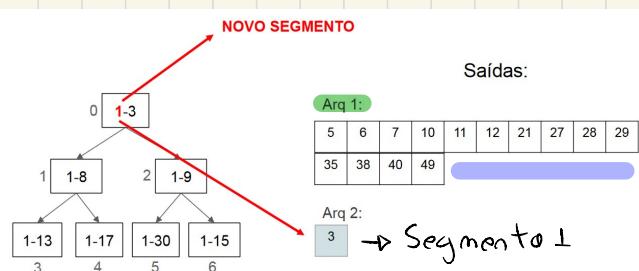
↳ Tem mais "peso" que o número

↳ Quanto maior → mais "pesado"

O - X

→ Insere-se no segmento ordenado até que não se tenha mais nós do mesmo segmento

↳ Cria outro segmento para continuar a distribuição



→ Números do Segmento 0

↳ "Menores" que os do Segmento 1

→ Segmentos criados são distribuídos de forma alternada entre os arquivos

→ Local onde o segmento 2 será inserido

→ Depois da Distribuição realizou as Intercalações

↳ Intercalação Polifísica

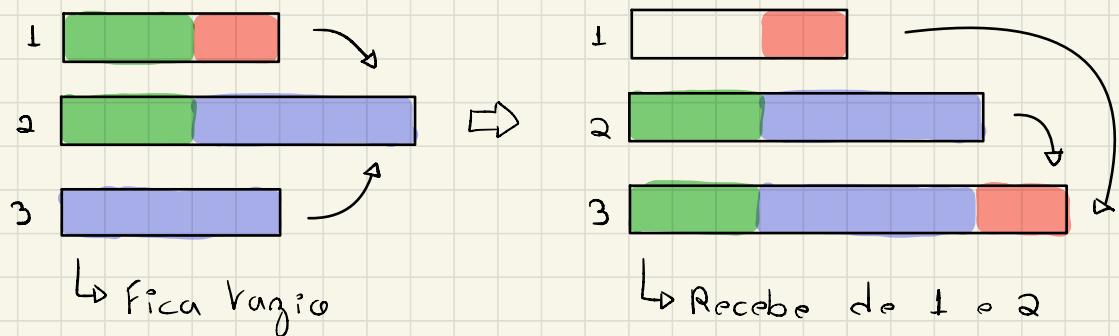
Arquivos → Fitas

→ Visa a "econômica" do arquivos temporários

↳ Usa somente M+1 arquivos (2M anteriormente)

→ Estratégia: Intercala até que uma fita de entrada fique vazia ela então será usado para próxima intercalação (Próxima fita de Saída)

- Blocos ordenados distribuídos de forma desigual entre as fitas
- É necessário deixar uma fita vazia



- ↳ Intercalação em muitas fases
 - ↳ Fases não envolvem todos os blocos
- ↳ Eficiência depende da Distribuição Inicial
 - ↳ É mais eficiente para M pequeno ($M < 8$)

ARQUIVO INDEXADO

- ↳ Registros Acessados de forma aleatória
 - ↳ Busca é realizada primeiro no índice e depois no arquivo de dados
 - ↳ Índice varia com o atributo (Indexado)
- ⇒ Conceitos Gerais
- ↳ Índices Primários: Ordem do arquivo original é mantida no índice
- ↳ Índices Secundários: Ordem no arquivo original é alterada

- ↳ Índice Direto: Índice aponta diretamente para o arquivo de dados
- ↳ Índice Indireto: Índice aponta para outro índice / Arquivo intermediário
- ↳ Índice Densos: Todos os registros possuem índice (1:1)
- ↳ Índice Esparsos: Alguns registros tem índice
- ↳ Operações com arquivos indexados não são diferentes de arquivos comuns
 - ↳ Pesquisas só são feitas no índice
 - ↳ Índices devem estar sempre atualizados
- ↳ Usuários não devem poder acessar Índices
 - ↳ CRUD não muda, Algoritmos sim
 - ↳ Interface
 - ↳ Execução
- ↳ Busca sequencial não é muito eficiente em arquivo do índice
 - ↳ Para otimizar podemos assumir índices como estruturas hierárquicas (Arvôres de múltiplos caminhos)

ÁRVORES DE MÚLTIPLOS CAMINHOS

- ↳ Árvores mais "Densas" / Baixas que árvores binárias
 - ↳ Caminhos entre nós mais curtos (menos acessos)

ÁRVORE B

- ↳ Ordem: Número máximos de filhos de um nó / Página

→ Regras:

- Nós com no mínimo 50% de ocupação
- Número de filhos = número de chaves + 1
- Todas as folhas estão no mesmo nível
 - ↳ "Cresce para cima"
- N° de chaves = N° de Filhos - 1

→ Estrutura e regras semelhante a Árvore 2-3-4

- ↳ Fragmentação por Ascensão
- ↳ Mesma Lógica de Operações

→ Ordem - 1

⇒ Inserção

- Insere os elementos até atingir o limite
- ↳ No limite: Fragmenta o nó e o maior elemento do nó sobe
- ↳ Fragmentação cria um nível acima

⇒ Remoção

I Elemento numa Folha com + 50% de ocupação

- Basta remover o elemento

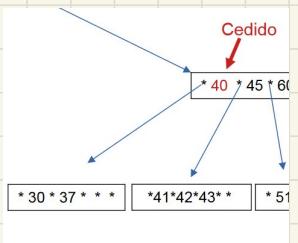
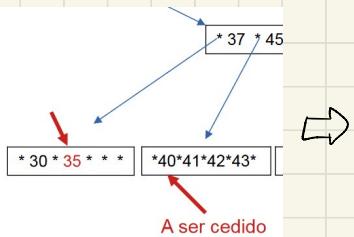
II Elemento não está numa folha

- trocamos o elemento pelo seu antecessor (se tiver na estrutura)

→ menor

III Folha fica com - de 50% de ocupação

- Se páginas irmãs puderem ceder um elemento (e) ainda ficarem com 50% de ocupação)



• É necessário atualizar a página - Pai p/ manter a estrutura da árvore

IV Irmas não podem ceder

↳ Caso as irmas não possam ceder Será feita uma Junção das Células vizinhas

↳ Junção das duas folhas que "ficarão" com menos de 50% de ocupação

ÁRVORE B+

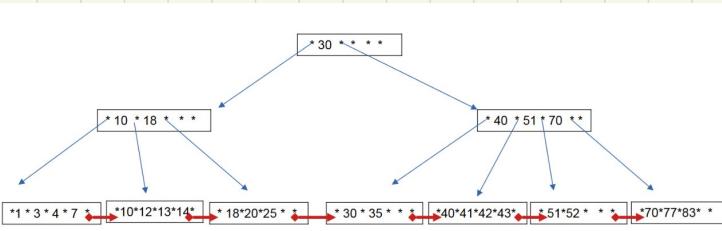
↳ Modificação da árvore B onde todas as chaves são armazenadas nas folhas

↳ Folhas possuem ponteiros entre si (Lista encadeada)

↳ Todas as chaves devem existir em nó folha e podem existir em nós internos

↳ Cópias de elementos das folhas existem em nós internos

↳ Usado para manter a estrutura da Árvore e princípio de Busca



⇨ Operações

↳ Iguais a árvore B mas com o cuidado de elementos repetidos

Busca → Primeiro na Árvore, depois entre os nós encadeados

Inserção → Ao Haver fragmentação, elemento do meio é copiado para o nó-pai e mantido no nó folha

Remoção → Em nó folha: Remove normalmente
Em nó Interno: Remove, acha Substituto e remove recursivamente nos Sub-Árvores

ÁRVORE B*

↳ Variação da Árvore B com variações no Split

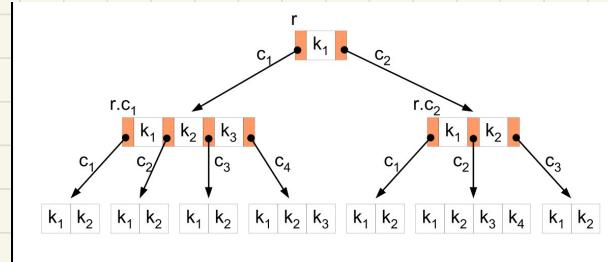
- Split é adiado até que 2 nós irmãos estejam completamente cheios

↳ Relações: $m \rightarrow$ Ordem $K \rightarrow$ Descendentes

- Ordem: Número máximo de descendentes
- Páginas tem no mínimo $(2m-1)/3$ descendentes

- Páginas tem $K-1$ chaves

↳ Validas p/ nós Internos



→ Arvore B

↳ One-to-two Split:

↳ Quando um nó está cheio ele se divide em 2 e os elementos são redistribuídos nos novos nós

↳ Elemento do meio (contando os 2 nós) sobe

↳ Redistribuição entre Nós-Irmãos: $\rightarrow B^*$

↳ Caso o nó esteja cheio, insere no nó irmão (até que ambos estejam cheios)

↳ Two-to-three Split: $\rightarrow B^*$

↳ Caso os 2 nós-Irmãos estejam cheios, crie-se um novo nó e os elementos são redistribuídos entre os 3 nós

↳ Elementos que ascendem idealmente são os maiores de cada nó

HASHING

↳ Tabelas de dispersão também podem ser usadas como arquivo de Índice

↳ Principal vantagem: custo de acesso $O(1)$

↳ Posição do Registro no Índice depende da função de dispersão

↳ Eficiência depende de uma boa função, norteando o número de registros

↳ Registro no índice deve possuir tamanho fixo

↳ Função retorna uma posição relativa (Deve ser multiplicada pelo tamanho do registro)

⇒ Escolha de função de dispersão

↳ Quanto mais dispersos os índices melhor (menos chance de colisão)

- Escolha p/ Campos numéricos

↳ Operação matemática (elevar, dividir, multiplicar)

↳ Conversão de Base (troca da base do número)

- Escolha p/ Campos não numéricos

↳ Geralmente usa o valor ascii do campo ou parte do campo

- Método da Dobrá

↳ Considera que os números estão num vetor que é "Dobrado" na medida sucessivamente e os números sobrepostos são somados (ou outra operação)

↳ Processo se repete até que o número resultante (dígitos) formem um número menor que o tamanho da tabela hash

↳ Geralmente só o bit/Algarismo menos significativo após a operação é mantido

Ex →

$$71 = 00010\ 00111 \Rightarrow 01111 \Rightarrow 15 \quad \rightarrow \text{"or" entre os bits espelhados (Dobra)}$$

$$46 = 00001\ 01110 \Rightarrow 11110 \Rightarrow 30$$

- Operação mod

↳ Geralmente o resto da divisão é usado na função

↳ Resto da divisão pelo tamanho da tabela

$$h(x) = x \bmod m \quad m \rightarrow \text{tamanho}$$

↳ Para diminuir a chance de colisão é ideal que "m" seja um número primo

- ↳ Não seja potência de 2
- ↳ Ajuda a "espalhar os valores"

COLISÕES

↳ Quando dois números/registros são mapeados para mesma posição do array índice

↳ Endereçamento Aberto

↳ Procura uma nova posição vazia dentro da própria estrutura

↳ Uso de uma fórmula/função matemática para encontrar a nova posição

• Sondagem Linear $h(x) = [h(k) + i] \% m$

↳ Procura a nova posição de forma sequencial e circular na tabela

↳ Se "retornar" ao mesmo índice a tabela está cheia

↳ Caso haja muito uso de sondagem, o tempo

de acesso aumenta (para aquele registro)

- Sondagem Quadrática $h(x) = [h(k) + i^2] \% m$

- ↳ Avanço até a próxima posição de forma quadrática e circular

- ↳ Avanço "mais rápido" que a Linear

- Duplo Hashing

- ↳ Implementa outra função a ser utilizada em caso de colisão

- ↳ Posição "Atual" serve como base para segunda função

- ↳ Vantagem: Chaves mais espalhadas

- ↳ Desvantagem: Distância entre endereços pode gerar buscas adicionais

- ⇒ Encadeamento

- ↳ Usa uma área "extra" para armazenar registros após as colisões

- ↳ Interno: Área dentro da própria estrutura

- ↳ Externo: Área fora da estrutura (outro arquivo)

- ↳ Geralmente estrutura reserva é uma lista encadeada

- ↳ tempo muda p/ parte encadeada

- ↳ Passa a ser $O\left(\frac{1}{m}\right)$ $n \rightarrow$ nº de registros
 $m \rightarrow$ tamanho da estrutura
- ↳ $O(1+n/m)$ para operações

↳ Para facilitar a manutenção, utilizam-se flags para indicar posições livres ou ocupadas

BUCKETS

↳ Otimização do acesso ao disco

↳ Define que cada posição no índice pode conter mais de um registro/Entrada

↳ Vários pares Chave-Endereço por posição do índice

↳ Tratamento de Colisões

↳ Colisões ocorrem quando o Bucket estiver cheio

↳ Soluções:

- Alocar o registro no próximo Bucket com espaço disponível
 - ↳ Endereçamento Aberto: Pontoiro "próximo" para o próximo Bucket usado/Disponível

- Usar outras técnicas de Hashing (Anteriores)
 - ↳ Considera que cada posição possui um Bucket

↳ Função de dispersão ideal é uniforme e aleatória

HASHING DINÂMICO

↳ Capacidade da tabela depende da necessidade (pode aumentar e diminuir)

↳ Hashing Estático: Capacidade é definida no começo e não pode ser alterada com facilidade

↳ Caso seja aumentada o Hash muda e é necessário reposicionar todos os elementos

↳ Hashing Extensível: Somente os registros do bucket são reposicionados

↳ tamanho do diretório é sempre equivalente a uma potencia de 2

Função: $h(k) = k \% \cdot 2^p$ $p \rightarrow$ "Profundidade"

↳ Profundidade: Corresponde ao numero de Bits para representação das posições do diretório/Índice

↳ Global: No Índice

↳ Local: No Bucket

Diretório		Buckets	
$p = 2$		Chave	End.
00		4	E0
01		6	E3
10		22	E4
11		31	E7

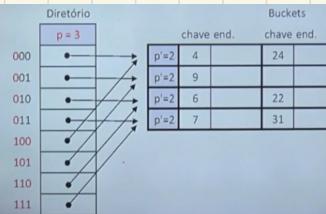
↳ Quando se aumenta a profundidade, dobra o tamanho do índice (1 bit a mais)

↳ Aumento da profundidade global mantem os novos índices associados ao bucket original

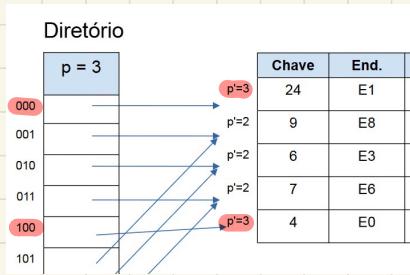
• Ao criar outro bucket um dos ponteiros "novos" irá para nova posição

↳ Profundidade Local Aumenta

↳ Remanejamento dos registros



- ↳ Apenas os registros/chaves dos buckets modificados são movidos
- ↳ Para os outros, os novos ponteiros apontam para o mesmo bucket



→ 3 bits necessários

- também é possível diminuir o N° de Buckets: Juntar elementos em um outro Bucket mais lógico
- ↳ Ponteiro passa para o "novo" Bucket

→ O(1)

O(log N)

- ↳ Custo de busca em flash é menor que em Arvore mas só é possível busca pela chave primária e não é possível selecionar intervalos

- ↳ Geralmente Índice Hash Sobrecarregu com mais facilidade

- ↳ Árvore é mais fácil de manipular, escalar

BUSCA POR CHAVE SECUNDÁRIA

- ↳ Chaves que identificam mais de um registro

- ↳ Pesquisa retorna um conjunto de registros

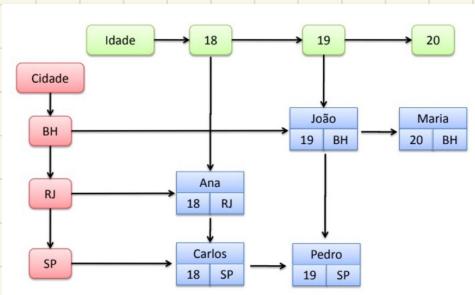
- ↳ Arquivo para busca de chaves deve ser organizado

- ↳ Maior eficiencia da Busca

- ↳ Multihista e Índice Invertido

MULTILISTA

- ↳ Encadeia registros que possuem o mesmo valor para chave secundária
- ↳ Diretório: Área onde os índices são armazenados



Posição	Nome	Idade	Cidade
01	João	19	BH
02	Ana	18	RJ
03	Pedro	19	SP
04	Maria	20	BH
05	Carlos	18	SP

→ Original

→ Multilista

Posição	Nome	Idade	Link I.	Cidade	Link C.
01	João	19	03	BH	04
02	Ana	18	05	RJ	-1
03	Pedro	19	-1	SP	05
04	Maria	20	-1	BH	-1
05	Carlos	18	-1	SP	-1

ÍNDICE INVERTIDO

- ↳ Usa parte de um registro para fazer a busca pelo registro
- ↳ Ideal para campos não numéricos
- ↳ termos São Identificados e cria-se uma lista com os registros que contêm cada termo

Cód	Título	.
1	Implementação de sistemas de bancos de dados	...
2	Sistemas de bancos de dados	...
3	Estruturas de dados e seus algoritmos	...
4	Dominando algoritmos	...
5	Estruturas de dados em java	...
6	Core Java	...
7	Biblioteca do programador Java	...

- Combina os termos p/ chegar no "conjunto" da Busca

Termos	Registros
algoritmos	3 4
bancos	1 2
biblioteca	7
core	6
dados	1 2 3 5
dominando	4
estruturas	3 5
implementação	1
java	5 6 7
programador	7
seus	3
sistemas	1 2

→ Vantagens:

- Bons p/ consultas com vários campos
- Não depende do atributo

→ Desvantagens

- Gasto de memória
- Operações difíceis