

Universidade Federal da Paraíba

Centro de Ciências e Tecnologia

Coordenação de Pós-Graduação em Informática

**BART - Uma Biblioteca Orientada a Objetos de Apoio à
Recuperação Textual**

José Eduardo Carvalho Bussmann

Campina Grande - PB

Dezembro - 1995

José Eduardo Carvalho Bussmann

**BART - Uma Biblioteca Orientada a Objetos de Apoio à
Recuperação Textual**

Dissertação apresentada ao Curso de MESTRADO EM
INFORMÁTICA da Universidade Federal da Paraíba,
em cumprimento às exigências para a obtenção do Grau
de Mestre.

Área de Concentração: Sistemas de Software

Jacques Philippe Sauvé
(*Orientador*)

Campina Grande - PB
Dezembro - 1995



B981b Bussmann, José Eduardo Carvalho.
BART : uma biblioteca orientada a objetos de apoio à recuperação textual / José Eduardo Carvalho Bussmann. - Campina Grande, 1995.
162 f.

Dissertação (Mestrado em Informática) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia.

1. Sistemas de Software. 2. BART - Biblioteca - Base de Dados. 3. Orientação à Objetos. 4. Dissertacao - Informática. I. Sauvé, Jacques Philippe. II. Universidade Federal da Paraíba - Campina Grande (PB). III. Título

CDU 004.65(043)

**BART - UMA BIBLIOTECA ORIENTADA A OBJETOS DE APOIO
À RECUPERAÇÃO TEXTUAL**

JOSÉ EDUARDO CARVALHO BUSSMANN

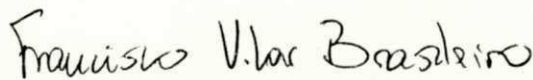
DISSERTAÇÃO APROVADA EM 19.12.95



PROF. JACQUES PHILIPPE SAUVÉ, Ph.D
Presidente



JOSÉ ANTÃO BELTRÃO MOURA, Ph.D
Examinador



FRANCISCO VILAR BRASILEIRO, Ph.D
Examinador

CAMPINA GRANDE - Pb

Para Maira e Naiara

Agradecimentos

À Maira e à Naiara pelo amor, carinho, compreensão e paciência durante a elaboração deste trabalho.

A meus pais, pelo incentivo e apoio e por sempre acreditarem em mim.

Ao meu orientador e amigo Jacques P. Sauvé, pela orientação, apoio e incentivo que muito contribuíram para a realização deste trabalho.

A todos os professores que de uma maneira ou de outra contribuíram para a conclusão deste trabalho.

Aos funcionários da COPIN e da MINIBLIO, em especial à Ana Lúcia Guimarães pela atenção e dedicação com as quais pude contar ao longo do curso.

A Luiz Maurício e Nivaldo pela amizade, oportunidade e pelo apoio no desenvolvimento deste trabalho.

A turma de desenvolvimento da Light-Infocon pela amizade e pelo compartilhamento de recursos e "macetes".

Aos amigos e colegas de curso pelos momentos de estudo e descontração.

Aos conterrâneos gaúchos pelas rodas de chimarrão, churrasco e cerveja.

Um agradecimento especial aos amigos Aninha e Jacques pela amizade e por nos proporcionarem ótimos momentos de lazer e descontração.

Resumo

O enorme volume de dados produzidos diariamente pelos computadores em todo o mundo tem feito com que a computação busque a todo momento novos mecanismos para o armazenamento e recuperação eficiente destas informações. Com este mesmo propósito este trabalho apresenta considerações de projeto e implementação de uma interface de programação (API) que oferece mecanismos para o armazenamento e recuperação de informações. Aplicações desenvolvidas com esta API são beneficiadas com características e facilidades de recuperação textual com um mínimo esforço adicional de codificação e um excelente tempo de resposta, independentemente do tamanho e da estrutura das informações e independentemente da complexidade de pesquisa.

Abstract

The large volume of computer-produced data appearing daily around the world has driven the search for new mechanisms to efficiently store and retrieve such information. It is along these lines that the present work presents design and implementation considerations for a programming interface (API) offering mechanisms for the storage and retrieval of information. Applications developed with this API benefit from embedded text retrieval features with a minimum of additional coding effort, and also receive excellent response time independently of the size and structure of the information manipulated and independently of the search complexity.

Lista de Figuras, IX**Lista de Tabelas, X****Capítulo 1 - Introdução**

- 1.1. Motivação, 2
- 1.2. Importância e Objetivos do Trabalho, 3
- 1.3. Recuperação Textual, 4
- 1.4. Trabalhos Relacionados, 6
- 1.5. Organização da Dissertação, 11

Capítulo 2 - Caracterização da BART

- 2.1. Visão Geral, 13
- 2.2. Requisitos Básicos, 14
 - 2.2.1. Flexibilidade, 14
 - 2.2.2. Abrangência, 15
 - 2.2.3. Facilidade de Programação, 16
 - 2.2.4. Portabilidade e Eficiência do Código, 16
 - 2.2.5. Robustez, 19
 - 2.2.6. Internacionalização, 20
- 2.3. Requisitos Específicos, 21
 - 2.3.1. Interação entre a BART e a Aplicação, 21
 - 2.3.1.1. Arquitetura, 23
 - 2.3.2. Interação entre a BART e o Programador, 24
 - 2.3.2.1. Objetos Internos, 25
 - 2.3.2.2. Objetos Acessíveis pelo Programador, 25

- 2.4. Principais Conceitos, 26
 - 2.4.1. Termo e Chave, 26
 - 2.4.2. Índice e Sistema de Índice, 27
 - 2.4.3. Documento, Conjunto, Grupo e Subgrupo, 28
 - 2.4.4. Referência e Localização, 29
 - 2.4.5. Proximidade, 31
 - 2.4.6. Adjacência, 32
 - 2.4.7. Tupla, 33
 - 2.4.7.1. Sêxtupla, 33
 - 2.4.7.2. Sétupla, 34
 - 2.4.8. Stopword, 34
 - 2.4.9. Goword, 35
 - 2.4.10. Máscaras, 35
 - 2.4.11. Normalizador, 36

Capítulo 3 - Utilizando os Recursos da BART

- 3.1. A Filosofia de Programação BART, 38
 - 3.1.1. Inicialização, 41
 - 3.1.2. Preparação, 42
 - 3.1.3. Acesso aos Recursos, 44
 - 3.1.4. Finalização, 46
- 3.2. Exemplos Práticos de Utilização da BART, 47
 - 3.2.1. Inicialização, 48
 - 3.2.2. Criação e Abertura de Sistemas de Índices, 49
 - 3.2.3. Configuração da BART, 51
 - 3.2.3.1. Gerenciamento de Objetos Independentes, 52
 - 3.2.4. Recursos de Indexação e Desindexação, 55
 - 3.2.5. Pesquisa, 56
 - 3.2.5.1. Pesquisa Simples, 57
 - 3.2.5.2. Pesquisa Composta, 59
 - 3.2.6. Fechamento de Sistemas de Índices, 61

Capítulo 4 - Considerações de Implementação da BART

- 4.1. Visão Geral dos Objetos, 63
- 4.2. Objetos Principais, 68
 - 4.2.1. Objeto Sessão, 69
 - 4.2.1.1. Estrutura de Memória, 71
 - 4.2.2. Objeto Sistema de Índice, 72
 - 4.2.2.1. Estruturas em Disco, 76
- 4.3. Troca de Dados entre a BART e as Aplicações, 80
 - 4.3.1. Parsers, 80
 - 4.3.2. Indexação e Desindexação de Grupos, 85
 - 4.3.4. Pesquisa de Informações, 89
 - 4.3.4.1. Linguagem de Consulta, 89
 - 4.3.4.1.1. Operadores da Linguagem, 92
 - 4.3.4.2. Estrutura de Retorno, 95
- 4.4. Avaliação de Desempenho, 97

Capítulo 5 - Conclusão

- 5.1. Avaliação dos Objetivos, 105
- 5.2. Conclusões Finais e Trabalhos Futuros, 107

Referências

- Referências Bibliográficas, 110
- Bibliografia, 112

Apêndice A - A Interface de Programação da BART

- A.1. Classe C_BART_ARQUIVO, 115
 - A.1.1. Métodos da Classe C_BART_ARQUIVO, 115
- A.2. Classe C_BART_EXPRESSAO, 117
 - A.2.1. Métodos da Classe C_BART_EXPRESSAO, 117
- A.3. Classe C_BART_FONEMA, 117

- A.3.1. Métodos da Classe C_BART_FONEMA, 118
- A.4. Classe C_BART_OCORRENCIA, 118
 - A.4.1. Métodos da Classe C_BART_OCORRENCIA, 118
- A.5. Classe C_BART_LISTA_DE_EXPRESSOES, 119
 - A.5.1. Métodos da Classe C_BART_LISTA_DE_EXPRESSOES, 120
- A.6. Classe C_BART_LISTA_DE_OCORRENCIAS, 121
 - A.6.1. Métodos da Classe C_BART_LISTA_DE_OCORRENCIAS, 122
- A.7. Classe C_BART_PARSER, 125
 - A.7.1. Métodos da Classe C_BART_PARSER, 126
- A.8. Classe C_BART_SESSAO, 127
 - A.8.1. Métodos da Classe C_BART_SESSAO, 128
- A.9. Classe C_BART_SINONIMO, 131
 - A.9.1. Métodos da Classe C_BART_SINONIMO, 131
- A.10. Classe C_BART_SISTEMA_DE_INDICES, 133
 - A.10.1. Métodos da Classe C_BART_SISTEMA_DE_INDICES, 134
- A.11. Classe C_BART_LISTA_DE_TERMOS, 141
 - A.11.1. Métodos da Classe C_BART_LISTA_DE_TERMOS, 142

Apêndice B - Exemplo de Utilização da BART

- B.1. Arquivo de Definições da Aplicação Exemplo, 145
- B.2. Código Fonte da Aplicação Exemplo, 145
- B.3. Arquivo de Definição da Classe C_PARSER, 159
- B.4. Código Fonte da Classe C_PARSER, 159

Lista de Figuras

- 1.1 - Uma base de dados textual simples, 9
- 2.1 - Arquitetura da API BART, 24
- 2.2 - A interação entre API BART e a interface de objetos internos, 26
- 3.1 - Principais etapas da interação Aplicação / BART, 40
- 4.1 - Visão geral dos objetos da BART sob o Diagrama Comentado de Booch, 64
- 4.2 - Estrutura de gerenciamento para alocação de sistemas de índices, 72
- 4.3 - Esquema lógico das estruturas de indexação da BART, 79
- 4.4 - Estrutura intermediária de indexação da BART, 86
- 4.5 - Procedimento de indexação no sistema de índices, 88
- 4.6 - Estrutura de retorno de uma pesquisa composta, 96

Lista de Tabelas

- 1.1 - Exemplo de ordenação por *sistring*, 9
- 3.1 - Exemplos de expressões de pesquisa utilizando operadores da linguagem de consulta da BART, 61
- 4.1 - Tipos de objetos retornados por métodos de outros objetos da BART, 65
- 4.2 - Tipos de objetos que podem ou que precisam ser criados pelo programador, 66
- 4.3 - Índices utilizados pelos aplicativos de testes, 98
- 4.4 - Dados de desempenho para arquivo texto de 10k, 99
- 4.5 - Dados de desempenho para arquivo texto de 100k, 99
- 4.6 - Dados de desempenho para arquivo texto de 500k, 99
- 4.7 - Dados de desempenho para arquivo texto de 1000k, 99

1. Introdução

Com o surgimento da escrita o homem desenvolveu a primeira forma de armazenamento do seu conhecimento e a primeira grande fonte transmissora de informações depois da fala. Com o passar dos anos o acúmulo de conhecimentos e informações armazenadas cresceu de tal forma que tornou-se impossível para as pessoas localizarem de forma rápida e precisa uma determinada informação em meio a tantas outras conjuntamente armazenadas.

As primeiras ferramentas de apoio a recuperação de informações começaram então a se fazer necessárias. Dentre elas, a mais importante é o *índice* - uma coleção de termos com apontadores para locais onde informações sobre eles podem ser encontradas[MANB92].

A computação, desde então, tem buscado os melhores caminhos para agilizar o armazenamento e a recuperação de informações. Para isso, ela vem fornecendo a cada ano novas ferramentas que facilitam a manipulação destas informações provendo vantagens como rapidez e eficiência para aqueles que da melhor maneira as utilizar.

Os Sistemas Gerenciadores de Banco de Dados (SGBD) por exemplo, são as ferramentas mais conhecidas concebidas para tal propósito. Estes sistemas manipulam dados estruturados e são, na sua grande maioria, desenvolvidos segundo o modelo relacional.

No entanto, os SGBD não são ferramentas desenvolvidas com o propósito de armazenar e recuperar informações não estruturadas, como textos por exemplo. Esta disfunção em relação às informações não estruturadas pode ser uma consequência da rigidez do modelo utilizado ou apenas uma carência de métodos ou funções específicas utilizadas para criação, armazenamento e recuperação de informações não estruturadas ou informações textuais.

1.1. Motivação

A carência de recursos de natureza textual nos sistemas de recuperação de informações levou alguns fabricantes de software a produzirem sistemas que se propõem a manipular exclusivamente textos. Estes produtos, por seu propósito, são conhecidos como Sistemas de Recuperação Textual (SRT), Sistemas Textuais, Sistemas de Recuperação de Informações Textuais ou ainda Bancos de Dados Textuais, caso o sistema apresente características de bancos de dados como por exemplo visões, relações, críticas ou restrições, etc.

Sistemas deste porte têm tido um grande destaque entre sistemas de recuperação de informações principalmente por que grande parte das informações manipuladas tanto em empresas como em ambientes científicos, comerciais ou domésticos ainda é não estruturada. Isto pode ser observado através dos documentos, cartas, textos (livros, artigos, publicações), leis, etc, que são manipulados diariamente e em grandes quantidades nos mais variados ramos da informática.

Em função da carência de recursos textuais e baseado no destaque que sistemas com tais recursos têm ganhado nos últimos anos, verificamos que uma ferramenta que oferece

recursos de recuperação textual e que tem a possibilidade de ser facilmente acoplada a qualquer outro aplicativo certamente tem grandes possibilidades de sucesso.

Com a agregação de recursos textuais a novos aplicativos, estaremos oferecendo a possibilidade de melhorar e facilitar a manipulação de grandes quantidades de informações não estruturadas. Com isso, os sistemas que manipulam informações nas áreas jurídica, legislativa, escriturária dentre outras, certamente ficarão otimizados e agilizados a partir do momento que fizerem uso de recursos de natureza textual.

1.2. Importância e Objetivos do Trabalho

É indiscutível a necessidade que as pessoas têm em possuir ferramentas que melhorem o desempenho no armazenamento e na recuperação das suas informações. Entretanto, a mesma tecnologia que possibilita o controle de informações através de computadores facilitando o trabalho das pessoas, ao mesmo tempo as torna mais exigentes a cada dia. O que fazia sucesso ontem pode não estar mais satisfazendo os usuários hoje, pois estes estão sempre buscando o meio mais eficiente para armazenar e recuperar informações.

Os grandes sistemas gerenciadores de informações estruturadas, por exemplo, podem suprir as necessidades de uma grande empresa no que se refere ao controle dos processos operacionais da mesma, entretanto o mesmo sistema pode não ser eficiente para a mesma empresa quando tratar dos processos administrativos e jurídicos. A informação estruturada não é portanto a melhor alternativa em todos os casos. Como já dito anteriormente as informações não estruturadas vêm ganhando destaque em diversos ambientes e estes já vêm exigindo ferramentas adequadas para manipulação deste tipo de

informação.

Diante de diferentes tipos e formatos de informações torna-se difícil para os usuários determinar que ferramenta utilizar em cada caso. Sem dúvida o ideal seria que os usuários pudessem dispor de ferramentas que possibilitassem, com o mesmo grau de eficiência e desempenho, o tratamento tanto de informações estruturadas como informações não estruturadas.

Com base nesta necessidade, o trabalho proposto objetiva definir o que é um sistema de recuperação textual, detalhar conceitos gerais e conceitos particulares relativos ao trabalho e demonstrar uma estrutura básica para a implementação de um sistema deste porte. Como resultado deste estudo será apresentada uma ferramenta¹ de apoio ao gerenciamento de informações, independentemente de qual estrutura elas possuam. Esta ferramenta chama-se *BART - Uma Biblioteca Orientada a Objetos de Apoio à Recuperação Textual* - e será apresentada no conteúdo desta dissertação.

1.3. Recuperação Textual

Um sistema de recuperação textual é todo o sistema que armazena documentos sob forma de conjuntos de palavras-chave e oferece consultas em uma linguagem natural[Yu94]. Nesta definição o autor enfatiza como os dados textuais são tratados, sem referenciar a forma ou formato dos mesmos.

¹ Durante as fases de especificação e projeto desta ferramenta houve também a participação dos pesquisadores Ananias F. de Lima Neto, Jarbas Campos, Luiz Mauricio Martins e Nivaldo Régis Júnior que contribuíram tecnicamente na elaboração e desenvolvimento do presente trabalho.

Na realidade, os dados textuais podem estar organizados em inúmeros documentos que não possuem qualquer estruturação explícita (cartas, artigos, publicações, etc) assim como podem estar estruturados em registros de dados ou tabelas de bancos de dados. O que importa realmente não é como ou de que forma os dados estão organizados pelas aplicações, mas sim como eles são tratados internamente pelos sistemas textuais. O tratamento interno dado a estes dados está relacionado diretamente com a recuperação dos mesmos.

Com relação à linguagem de consulta, [Yu94] afirma que o esquema de recuperação de dados nos sistemas textuais é feito em função de palavras. Já [Falout85] diz que as consultas em um sistema de recuperação textual consiste de palavras ou partes de palavras conectadas com operadores lógicos (E, OU, NAO). Entretanto, segundo o autor, alguns sistemas permitem operadores adicionais que representam sentenças como *palavra1 seguida de palavra2, palavra3 na mesma frase de palavra4* ou pesquisas baseadas em lógica *fuzzy* também conhecidas como *fuzzy queries*².

O trabalho apresentado tem como um de seus objetivos definir o que é um sistema de recuperação textual. Este objetivo busca apresentar, baseado em definições existentes e nas características particulares deste trabalho, o que entendemos como sendo um sistema de recuperação textual.

As características principais dos sistemas de recuperação textual se encontram nos recursos de pesquisa, através dos quais os usuários podem ter acesso a seus dados. Como já

² Na fuzzy query ou consulta difusa um documento é ou não qualificado para fazer parte da resposta levando-se em consideração o grau de relevância definido para a consulta.

visto anteriormente os esquemas de consultas em sistemas de recuperação textual são baseados na recuperação de palavras, portanto, um sistema de recuperação de informações textuais é aquele que permite que os usuários recuperem seus dados a partir de qualquer palavra existente em sua base de dados.

A pesquisa por palavra única deve ser o recurso mínimo oferecido pelo sistema. Porém, características e recursos mais elaborados podem fornecer aos usuários uma maior certeza nas informações que estão procurando. Com base nisso, apresentamos algumas características que ampliam o poder de consulta dos sistemas de recuperação textual e que acreditamos serem indispensáveis em sistemas deste tipo. Entre elas estão:

- pesquisa por argumento (qualquer palavra existente na base de dados);
- pesquisa com conectores lógicos (E, OU, NAO, XOU);
- pesquisa por prefixo, sufixo ou qualquer outro segmento de uma palavra (utilizando expressões regulares);
- pesquisa por data, hora ou por intervalos de datas ou horas;
- pesquisa por intervalos ou ocorrência de valores;
- pesquisa com sentenças compostas (expressões ou linguagem de consulta);
- pesquisa por fonemas;
- pesquisa por sinônimos;
- pesquisa por localização de palavras (documento, grupo, frase, parágrafo e proximidade);

1.4. Trabalhos Relacionados

O crescente uso de mecanismos de manipulação de informações não estruturadas e a conseqüente necessidade de consultar e recuperar tais informações têm atraído muito interesse de pesquisa nos últimos anos. Assim, o objetivo desta seção é apresentar trabalhos de pesquisa diretamente relacionados com o contexto do presente trabalho bem como

ênfatizar as contribuições e diferenciais apresentados por esta dissertação.

[Gonçal89] apresentou algoritmos para construção de árvores binárias ênfatizando a construção de árvores PATRICIA[Morri68]. Neste trabalho o autor estuda problemas de distribuição de nodos da árvore para armazenamento em disco e oferece alternativas para alocação paginada dos nodos. Além disso é apresentado um sistema de indexação utilizando a árvore PATRICIA que, basicamente, permite a indexação de arquivos texto (divididos ou não em campos definidos pelo usuário) assim como a sua recuperação através de uma linguagem de consulta. Esta linguagem possibilita a recuperação de palavras ou seqüências de palavras (utilizando conectores lógicos E, OU, NÃO) do banco de dados e oferece como resultado o número total de ocorrências que satisfazem a pesquisa e uma lista de documentos correspondentes a essa pesquisa.

[Harm92] introduz arquivos invertidos como sendo uma lista (ou índice) ordenada de palavras chaves onde cada chave aponta para o documento que a contém. Segundo a autora, diversas estruturas podem ser utilizadas para a implementação de arquivos invertidos, entre elas estão *sorted arrays*, *B-trees*, *Tries* e *PAT trees*. Ainda neste trabalho são apresentados algoritmos para construção de arquivos invertidos utilizando *sorted array* e algoritmos para uma rápida inversão de textos chamados **FAST-INV**³.

Na estrutura apresentada para construção de arquivos invertidos, inicialmente o texto é dividido nas palavras que formam uma lista de palavras que posteriormente é invertida através da ordenação alfabética destas palavras. Embora a autora ênfatize a

³ Copyright © Edward A. Fox, Whay C. Lec, Virginia Tech

possibilidade do arquivo invertido referenciar a localização completa das palavras no texto (registro, campo, parágrafo, etc), os algoritmos apresentados se restringem a referenciar apenas o registro que contém a palavra. [Falout92] apresenta uma visão geral de métodos para recuperação textual baseados em assinaturas. O autor descreve a idéia principal dos métodos de assinaturas e algumas vantagens sobre outros métodos de recuperação textual. São apresentados também uma classificação dos métodos de assinatura, a principal representação de cada classes com suas vantagens e desvantagens e uma lista de aplicativos baseados nos métodos de assinaturas. Igualmente ao trabalho apresentado por [Harm92], os algoritmos dos aplicativos se restringem a referenciar apenas o registro que contém a palavra, porém sem mencionar a possibilidade de referência completa das palavras indexadas.

Uma visão de novos índices para textos (ênfatisando PAT Array) é apresentada por [Gonn92]. Segundo o autor, PAT Array é uma implementação eficiente de PAT Tree[Gonn83]⁴ e suporta uma linguagem de consulta mais poderosa do que as estruturas tradicionais baseadas em palavras chaves e operações booleanas. Neste modelo a estrutura de referência e indexação das palavras dos textos é baseada em cadeias semi-infinitas (*semi-infinite strings* ou simplesmente *sistrings*)[Gonn83,87][Manb90]. Conforme apresentado, esta estrutura trata o texto indexado como uma longa cadeia de caracteres onde cada posição nesta cadeia é chamada de *sistring*. Cada *sistring* é definida pela posição de início (ponto de indexação ou *index point*) e estendida para a direita (seguindo a seqüência da cadeia) tanto quanto for necessário ou até atingir o final da cadeia. Nesta definição, apenas os símbolos alfabéticos precedidos por um caracter branco (início de palavras) são

⁴PAT Tree é uma evolução da árvore PATRICIA[Morri68] apresentada por [Gonn83].

indexados, além disso, o usuário deve determinar quais as posições do texto que serão indexadas. Isto é feito dependendo das seqüências de caracteres que serão consideradas como *sistrings* de pesquisa. A figura a seguir ilustra o exemplo de um texto contendo oito *index points*, onde cada um deles corresponde ao endereço de uma *sistring*.

↑	↑	↑	↑	↑	↑	↑	↑
1	6	12	14	17	25	28	36
Este texto é um exemplo de cadeias semi-infinitas							

Figura 1.1 - Uma base de dados textual simples contendo oito *index points*.

INDEX POINT	SISTRING
28	cadeias semi-infinitas
25	de cadeias semi-infinitas
12	é um exemplo de cadeias semi-infinitas
1	Este texto é um exemplo de cadeias semi-infinitas
17	exemplo de cadeias semi-infinitas
36	semi-infinitas
6	texto é um exemplo de cadeias semi-infinitas
14	um exemplo de cadeias semi-infinitas

Tabela 1.1 - Exemplo de ordenação por *sistring* da base de dados textual da figura 1.1.

Conforme colocação do autor, a principal vantagem de PAT Array sobre outras estruturas de indexação (principalmente arquivos de assinatura e arquivos invertidos) é o grande potencial de uso de outros tipos de pesquisa. Tipos estes que são ou difíceis ou muito ineficientes em arquivos invertidos, que é o caso de pesquisa por frase, pesquisas com expressões regulares e pesquisa com proximidade. O SPAT Array apresentado em [Beaz93] é uma evolução do PAT Array que busca a eficiência deste algoritmo para consultas em CD-ROM. O SPAT Array (*Short-Pat Array*) apresenta a mesma estrutura básica do PAT Array porém consiste de um índice adicional, mantido na memória principal durante a fase de consultas, que contém informações extras sobre o PAT Array e sobre os dados do arquivo texto fazendo com que o número de acessos ao CD-ROM seja reduzido.

A estrutura que utilizamos no presente trabalho é baseada em um tipo de índice lexográfico⁵ chamado **arquivo invertido** (também conhecido como lista invertida). Para compor esta estrutura utilizamos como estrutura principal de indexação árvores B+ (B+ Tree) associadas a arquivos de dados que mantêm as informações de inversão dos documentos indexados.

Ao contrário dos trabalhos citados, apresentamos uma estrutura onde as informações de inversão armazenam a referência completa das palavras indexadas. Desta forma é possível localizar precisamente não só os documentos que possuem estas palavras mas também localizar suas posições exatas dentro destes documentos.

Apesar dos mecanismos utilizados para possibilitar o armazenamento completo das referências das palavras gerarem *overhead* na indexação dos documentos destacamos duas vantagens que justificam sua utilização.

A principal delas está relacionada com a linguagem de consulta que, segundo [Gonn92], é ineficiente para pesquisas por frase, pesquisas com expressões regulares e por proximidade quando utilizadas em listas invertidas. Porém verificamos que, de posse das informações completas das palavras, é perfeitamente possível oferecer de forma fácil e eficiente uma linguagem de consulta que ofereça estes tipos de pesquisas, considerados ineficientes quando implementados sob listas invertidas.

A outra vantagem diz respeito ao destaque das palavras resultantes de uma operação de consulta. Com informações completas sobre a localização das palavras é possível para as aplicações identificar com uma maior facilidade e precisão somente aquelas palavras que

⁵ Índices lexográficos são índices ordenados[Donn92].

fazem parte do resultado da pesquisa. Com este recurso é mais fácil oferecer algum mecanismo de destaque para estas palavras (sublinhado, brilho, tarja, ...) oferecendo assim uma visualização mais rápida por parte do usuário.

1.5. Organização da Dissertação

Neste capítulo de *Introdução* apresentamos os aspectos que motivaram o desenvolvimento deste trabalho além dos objetivos, importância do mesmo e alguns aspectos diferenciais entre o presente trabalho e trabalhos relacionados. Apresentamos também definições de sistemas de recuperação textual sob o ponto de vista de outros autores bem como uma definição baseada nas características deste trabalho; finalmente, esta seção apresenta a organização do trabalho.

O capítulo 2, *Caracterização da BART*, apresenta uma visão geral da biblioteca e define os requisitos básicos e específicos que deverão ser considerados na implementação da BART.

O capítulo 3, *Utilizando os Recursos da BART*, apresenta a filosofia de programação BART caracterizando as etapas de desenvolvimento (estrutura básica) de uma aplicação e explica, com o auxílio de exemplos de programação, como o programador deve utilizar os recursos da BART.

O capítulo 4, *Considerações de Implementação da BART*, demonstra uma visão geral dos objetos que compõem a BART apresentando os relacionamentos existentes entre os mesmos e as estruturas de dados dos principais objetos. São apresentadas também considerações de implementação no que se refere à inicialização das estruturas físicas e

dinâmicas e com relação aos mecanismos de troca de dados entre a biblioteca e as aplicações. Este capítulo possui também uma seção onde são apresentados dados de desempenho de indexação e pesquisa da BART juntamente com um comparativo destes dados com dados de desempenho de outros trabalhos com recursos de recuperação textual.

No capítulo 5, *Conclusão*, é feita uma avaliação dos objetivos propostos pelo trabalho e uma análise crítica ligada às considerações feitas em relação aos requisitos definidos para a biblioteca. Além disso são apresentadas as conclusões finais do trabalho e algumas sugestões para trabalhos futuros.

O apêndice A, *A Interface de Programação da BART*, é destinado para a descrição da interface de programação através da apresentação dos métodos dos objetos colocados à disposição do programador.

O apêndice B, *Exemplo de Utilização da BART*, é destinado para a apresentação de uma aplicação que utiliza os recursos básicos de indexação, desindexação e recuperação textual da BART.

2. Caracterização da BART

Neste capítulo discutiremos as principais características da BART. Inicialmente apresentaremos uma visão geral da biblioteca e posteriormente definiremos seus requisitos básicos e específicos que envolvem a interação entre a BART e as aplicações bem como a interação do programador com os recursos oferecidos pela biblioteca. Finalmente serão apresentados os conceitos que envolvem as estruturas e a implementação da biblioteca.

2.1. Visão Geral

A BART é uma camada de software que oferece algoritmos rápidos e eficientes que auxiliam os programadores a desenvolver aplicações com recursos de recuperação textual. A API (Application Programming Interface - Interface para Programação de Aplicações) BART consiste de uma biblioteca de classes orientadas a objetos (desenvolvidas em C++) que foi projetada visando diretamente usuários programadores e desenvolvedores de aplicativos que necessitam agregar recursos otimizados para a manipulação de grandes volumes de informações (estruturadas ou não estruturadas), visando mais especificamente a manipulação de informações textuais.

Para atingir o objetivo de oferecer ao programador um conjunto consistente de objetos é necessário que a BART atenda primeiramente a um conjunto de requisitos básicos e depois a um grupo de requisitos específicos que dizem respeito a sua implementação. Os

requisitos básicos são aqueles que a BART deve seguir de forma a satisfazer as necessidades das aplicações. Já os requisitos específicos devem atender as necessidades do programador no que diz respeito a interação entre programador e API bem como entre API e as aplicações.

Tendo em vista esta necessidade as seções seguintes são dedicadas à apresentação dos referidos requisitos.

2.2. Requisitos Básicos

Para facilitar a descrição de cada requisito possibilitando uma melhor compreensão dos mesmos, esta seção foi dividida em subseções onde cada uma delas se concentra em um determinado requisito. Os requisitos básicos atendidos pela BART são: **flexibilidade, abrangência, facilidade de programação, portabilidade e eficiência do código, robustez e internacionalização**. Tais requisitos são discutidos nas seções seguintes.

2.2.1. Flexibilidade

O requisito de flexibilidade atendido pela BART diz respeito às necessidades das aplicações. Sendo assim, a BART deve oferecer mecanismos que se adaptem ao nível de complexidade individual de cada aplicação. Desta forma a biblioteca deve estar preparada para atender aplicações com níveis pequenos de complexidade bem como oferecer recursos mais complexos que atendam às aplicações que exigem um certo grau de complexidade.

Entretanto, oferecer mecanismos flexíveis de complexidade não implica em oferecer mecanismos isolados para atender cada caso. É importante que os recursos da biblioteca

possam ser escaláveis juntamente com a complexidade da aplicação. Uma aplicação que, no início, optou por um conjunto de funcionalidades mas que evoluiu durante seu ciclo de vida deve poder aumentar a complexidade de suas funcionalidades sem precisar ajustar-se novamente aos recursos da BART.

Em resumo, a flexibilidade não deve implicar em complexidade. Ela deve seguir a filosofia de que se a aplicação quer algo simples ela deve obter recursos simples de serem utilizados. Somente quando a aplicação desejar algo mais complexo é que ela vai enxergar complexidade.

2.2.2. Abrangência

Para que a BART seja uma biblioteca de uso genérico ela deve possibilitar que aplicações dos mais variados tipos acessem seus recursos. Para tanto é necessário que ela ofereça suporte para os mais diferentes tipos de dados, ou seja, dados numéricos, alfanuméricos, datas, horas, texto e dados binários.

Tipos de dados como data, hora e valor (dados numéricos) são necessários e tratados de forma diferenciada. Estes tipos de dados são armazenados em formatos particulares para que seja possível oferecer recursos especiais de pesquisa utilizando operadores relacionais como por exemplo $>$, $>=$, $<$, etc.

Contudo, como a biblioteca dá ênfase principalmente à recuperação textual, a biblioteca deve oferecer uma linguagem de consulta especial para recuperação deste tipo de informação. Além disso, como dados binários não são passíveis de indexação, a BART deve oferecer um mecanismo especial de armazenamento para estes tipos de dados.

Isto implica que, devido aos tipos de dados suportados, a indexação e o armazenamento de dados devem ser flexíveis e independentes favorecendo sempre a parte textual.

2.2.3. Facilidade de Programação

A facilidade de programação deve ser não só um requisito mas também uma característica da biblioteca. A API BART deve oferecer um bom nível de abstração de dados de forma a permitir que, durante a fase de desenvolvimento de aplicações, o programador identifique os objetos e as operações que serão utilizadas da forma mais clara possível.

Além disso, a criação de objetos bem como a manipulação dos mesmos devem ser simples e de fácil utilização, mesmo para os recursos mais avançados. Em poucas palavras o que a facilidade de programação quer é oferecer uma API que proporcione uma programação no nível mais alto possível.

2.2.4. Portabilidade e Eficiência do Código

A BART deve estar preparada para ser utilizada por aplicações de diversos tipos de ambientes e sistemas operacionais e para isso deve apresentar um código portátil que suporte as diferenças entre estes ambientes. Embora a portabilidade entre máquinas e sistemas operacionais exista na teoria, na prática ela nem sempre é tão simples, já que existem diversos fatores como microprocessadores, arquitetura do hardware e implementação e serviços do sistema operacional que influem diretamente na portabilidade de código fonte.

Através deste requisito (portabilidade) definiremos alguns pontos importantes para a implementação da BART a fim de definir previamente para que tipos de ambientes a biblioteca poderá ser portada.

Com relação ao tratamento de memória existe um ponto de fundamental importância e que é um requisito indispensável para a utilização da BART - memória virtual. O grande volume de dados que podem vir a ser manipulados durante etapas de indexação e pesquisa pode não caber no espaço de endereçamento da memória principal da aplicação. Assim, como a BART não prevê o tratamento de swaps na sua implementação, é imprescindível que o ambiente operacional no qual se executarão aplicações que utilizam os recursos desta biblioteca implemente recursos de memória virtual. Caso contrário, as limitações de memória física podem prejudicar ou impedir o bom funcionamento dos recursos.

A proteção de memória é outro fator importante a ser considerado. Contudo, este fator não é diretamente importante para a BART, mas sim para as aplicações usuárias de seus recursos. Aplicações mais críticas (aplicações distribuídas ou cliente/servidor) devem evitar ambientes operacionais que não tenham este tipo de preocupação.

Ainda com relação ao tratamento de memória, é necessário que haja um esquema flexível para alocação de memória onde a aritmética de apontadores não fique presa em limites de segmentos.

Além dos requisitos de memória outros dois fatores, ainda com relação a sistemas operacionais, devem ser considerados. Estes requisitos são esquemas de travamento de arquivos e procedimentos de E/S. Para facilitar a portabilidade do código deve-se usar as funcionalidades padrão para este tipo de processo, evitando mecanismos nativos de cada

sistema operacional.

Com relação às ferramentas de desenvolvimento que originalmente já oferecem inúmeras funcionalidades e bibliotecas particulares de recursos⁶, deve-se tomar o cuidado de evitar a utilização destes mecanismo ditos facilitadores, de forma a minimizar as dependências com relação ao ambiente de desenvolvimento. O uso de bibliotecas particulares pode facilitar o desenvolvimento em ambientes específicos, contudo certamente dificultará o porte do código fonte para outros ambientes.

No que diz respeito a eficiência do código, sabemos que os mecanismos que auxiliam as pessoas no seu dia-a-dia também fazem com que estas mesmas pessoas exijam software cada vez mais rápidos. Isto se deve ao fato das pessoas estarem sempre em busca das formas mais eficientes para armazenar e recuperar suas informações.

Com base nesta certeza a BART deve se preocupar em garantir que seus métodos de acesso, tanto para dados estruturados quanto para dados não estruturados, sejam os mais eficientes possíveis. Para isso é necessário elaborar estruturas de dados que sejam eficientes e ao mesmo tempo compatíveis com o volume de informação que será armazenada. Como apoio na verificação da eficiência das estruturas de dados a serem empregadas devemos utilizar ferramentas de perfil (*profile*) de forma a obter uma nível de informação mais detalhada da eficiência das estruturas de dados em tempo de execução de código.

⁶ OWL (Object Windows Library) e BWCC (Borland Windows Custom Controls) ambos da Borland e MFC (Microsoft Foundation Class) são alguns exemplos de bibliotecas que oferecem recursos para "facilitar" o desenvolvimento de aplicações em seus respectivos ambientes.

2.2.5. Robustez

A capacidade de um sistema em se manter funcionando mesmo após a ocorrência de erros ou falhas durante a execução de uma aplicação é uma característica de substancial importância para qualquer sistema de armazenamento e recuperação de dados. Como a proposta da BART é oferecer mecanismos para este tipo de sistemas é fundamental que ela se preocupe com características de robustez.

Assim sendo é requisito para a implementação desta biblioteca que ela se preocupe com mecanismos de reconstrução de seus índices ou sistema de índices (ver definições em 2.4.2.) bem como de qualquer tipo de informação que esteja sob sua responsabilidade. Para isso a BART deve oferecer recursos para, primeiro verificar sua própria estrutura a fim de determinar se houveram falhas em processamentos anteriores e segundo para tentar corrigir os problemas encontrados.

Uma maneira de minimizar a ocorrência de falhas durante a execução de aplicações é a utilização de recursos de flush (esvaziamentos ou descargas). Estes recursos consistem na sincronização das estruturas de dados mantidas na memória principal para a memória secundária, atualizando fisicamente imagens destas estruturas em disco. Entretanto, existem dois pontos importantes com relação a mecanismos de flush.

Em primeiro lugar é importante ressaltar que apesar destes mecanismos atenderem as situações em boa parte dos casos, eles funcionam como mecanismo de prevenção e não como mecanismo de recuperação.

Outra questão diz respeito ao desempenho do sistema. Sistemas que utilizam mecanismos de flush são mais lentos pois não utilizam recursos de buferização de suas

saídas. Por outro lado sistemas que utilizam buferização de saída estão mais sujeitos a falhas. Como recursos de flush afetam o desempenho do sistema, seria importante a existência de mecanismos que possibilitassem ao sistema optar entre velocidade de execução ou recursos de prevenção de falhas.

Independentemente destes fatores é necessário que existam mecanismos para recuperação de falhas, quando elas acontecerem. Contudo, em algumas situações como erros fatais, por exemplo, existe a impossibilidade de correção dos erros. Neste caso deve-se oferecer algum mecanismo de log para informar o que realmente aconteceu a fim de auxiliar no esclarecimento do problema.

2.2.6. Internacionalização

Por se propor a ser uma biblioteca de uso genérico é razoável que a BART previna-se para as diferenças entre conjuntos de caracteres particulares a cada idioma existente hoje a nível mundial. Por esta razão devem existir meios de ajustar alguns parâmetros de funcionamento da biblioteca de forma a satisfazer este requisito.

No que se refere a tabelas de caracteres é importante suportar conjuntos de caracteres internacionais além do ISO8859/1. Suporte a Unicode[Unic91] não é essencial neste momento, embora deva existir a facilidade de ser incluído suporte total a este sistema futuramente.

O problema maior relacionado aos mecanismos de internacionalização se deve à dificuldade de oferecer tais recursos de forma que sejam facilmente portáveis. Em geral, cada sistema operacional adota um mecanismo diferente para o tratamento deste problema.

Como não existe nenhum padrão a melhor alternativa é isolar os mecanismos particulares de internacionalização para da mesma forma isolar o problema facilitando a portabilidade (requisito 2.2.4.).

Recursos que envolvem características típicas para cada idioma devem oferecer suporte a nível de configuração, como é o caso de sinônimos, fonemas, linguagem de pesquisa dentre outros. Detalhes dos recursos de configuração podem ser vistos na seção 3.2.3.

2.3. Requisitos Específicos

Os requisitos específicos estão relacionados basicamente com dois aspectos. O primeiro diz respeito à interação entre a BART e as aplicações no que se refere à troca de informações entre as duas partes. O outro aspecto esta relacionado com a forma de interação entre a API da biblioteca e o programador, os objetos que devem ser vistos pelo programador e os objetos de uso interno da biblioteca aos quais, em princípio, o programador não tem acesso.

2.3.1. Interação entre a BART e a Aplicação

A interação entre a BART e uma aplicação deve ocorrer basicamente em cinco grupos de operações:

1. alocação e liberação de recursos;
2. configuração de recursos;
3. gerenciamento de índices e dados;

4. recuperação de dados;

5. tratamento de erros.

O grupo 1 (alocação e liberação de recursos) está relacionado com a etapa de preparação da biblioteca para que a mesma aloque e inicialize os recursos necessários para poder oferecê-los às aplicações. Este grupo também é responsável pela liberação de todos os recursos e estruturas alocados nesta etapa de preparação.

O grupo 2 (configuração de recursos) é um passo intermediário entre a etapa de preparação e o uso propriamente dito dos recursos. Neste passo a aplicação realizará a configuração dos objetos que julgar pertinentes para poder adequar os recursos da BART da maneira que melhor lhe convier. Isso inclui definição de stopwords, gowords ou máscaras (ver definição nas seções 2.4.8, 2.4.9 e 2.4.10 respectivamente) além das definição de sinônimos, fonemas e operadores para idiomas compatíveis com a aplicação, o que previamente atende parte do requisito 2.2.6. que trata de recursos de internacionalização das aplicações. Porém, ao contrário das operações relacionadas com o grupo 1 e apesar de ser uma etapa importante na interação da biblioteca com as aplicações, a configuração de recursos não é uma etapa obrigatória para a utilização dos recursos da BART.

As operações do grupo 3 (gerenciamento de índices e dados) caracterizam as operações de troca efetiva de dados no sentido aplicação → BART. Através deste grupo de operações as aplicações fornecem os dados e o tipo de tratamento que a biblioteca deve fazer sobre os mesmos. Estão relacionadas com este grupo operações de indexação, desindexação e armazenamento de dados.

No grupo 4 (recuperação de dados), as operações também são caracterizadas pela troca de dados entre as aplicações e a BART porém neste caso a troca de dados ocorre no

sentido BART → aplicação. Através de consultas ou pesquisas de informações, as aplicações recebem tipos de dados que fazem referência às informações fornecidas à BART através das operações de indexação ou armazenamento de dados do grupo 3.

As operações do grupo 5 (tratamento de erros) estão presente em todos os demais grupos e são as responsáveis pelo fornecimento de informações referentes a falhas durante a execução de qualquer outra operação. O objetivo principal deste grupo é informar às aplicações erros de execução durante a utilização dos recursos da BART porém, um esquema de log para tratamento de erros fatais pode ser adequado a este grupo de operações atendendo à solicitação do requisito de robustez (seção 2.2.5.).

2.3.1.1. Arquitetura

A figura 2.1 demonstra a arquitetura geral da BART e a forma com a qual os aplicativos acessam seus recursos. Como pode ser observado a aplicação pode tratar o armazenamento de seus dados diretamente ou utilizando recursos oferecidos pela API BART. Contudo, embora atendendo a requisitos de abrangência (seção 2.2.2.) a BART deva oferecer operações para o armazenamento de dados, a aplicação continua com a responsabilidade de gerenciar o armazenamento e o acesso aos seus dados, ou seja, independentemente de quem realizar a tarefa de gravação ou leitura dos dados não será tirada da aplicação a responsabilidade de gerenciá-los.

Portanto, a principal relação entre a BART e as aplicações está ligada ao fato de que a biblioteca armazena índices que referenciam os dados das aplicações, mas a aplicação é quem deve gerenciar o armazenamento de seus próprios dados, mesmo que para isso utilize recursos oferecido pela BART.

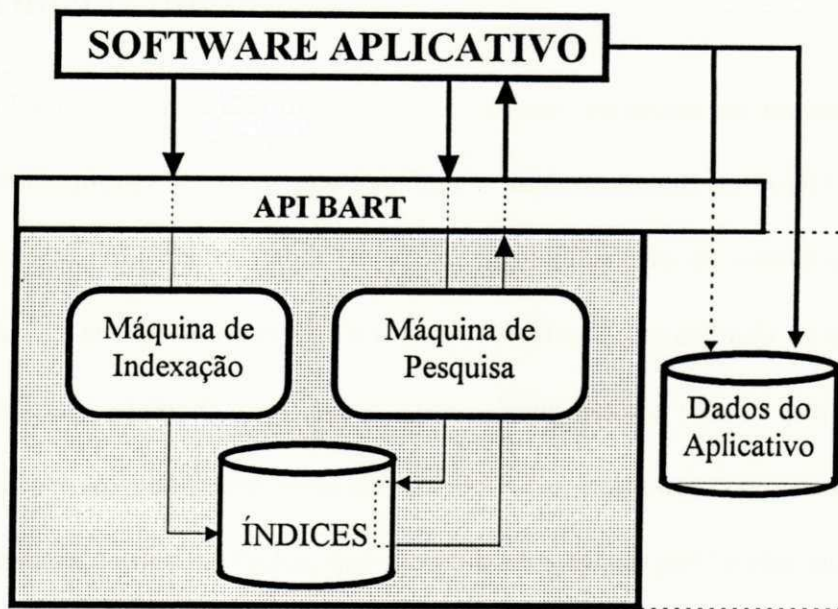


Figura 2.1 - Arquitetura da API BART.

2.3.2. Interação entre a BART e o Programador

Como visto na seção anterior, as aplicações têm diversas formas de interagir com a BART. Entretanto, para que esta interação exista em uma aplicação final, é ou foi necessária uma prévia interação entre o programador da aplicação e a API BART. Esta interação entre programador e BART é definida nesta seção.

O objetivo desta seção não é detalhar a nível de objetos a interface de programação, mas sim apresentar uma especificação em alto nível, baseada nos grupos de operações da seção 2.3.1, referente à visão do programador em relação à biblioteca. Para tanto dividimos esta seção definindo primeiramente o que são os objetos internos e posteriormente definindo o que são os objetos acessíveis pelo programador.

2.3.2.1. Objetos Internos

Os objetos internos são um grupo de objetos invisíveis ao programador e que controlam as estruturas de dados que habilitam o funcionamento da BART. São também estes objetos que realizam as tarefas de baixo nível (alocação de memória ou recursos, operações de E/S em disco, chamadas a rotinas do sistema operacional) tornando este tipo de operação transparente e, em muitos casos, desnecessária para o programador. Em resumo, estes objetos não fazem parte da API BART e, portanto, o programador não tem poder de manipulação destes objetos, que são utilizados apenas por tarefas internas, ou seja, somente os objetos disponíveis na API enxergam internamente estes objetos.

2.3.2.2. Objetos Acessíveis pelo Programador

Os objetos acessíveis pelo programador formam a camada de software entre a BART e as aplicações ou a API BART propriamente dita. Este grupo de objetos tem uma dupla função. Primeiro a função de oferecer uma interface de programação de alto nível, permitindo que o programador manipule objetos com funcionalidades e características bem definidas, o que facilita a interação entre o programador e os métodos destes objetos. A outra função destes objetos é interagir com a camada de objetos internos quando necessitarem realizar tarefas de baixo nível.

A figura 2.2 demonstra a arquitetura da BART a nível de objetos. Podemos observar que ao acessarmos os cinco grupos de operação (definidos na seção 2.3.1.) estamos interagindo diretamente com a API BART; contudo, quando estas operações necessitarem de alguma tarefa de baixo nível elas interagem com uma interface interna para objetos que realizam estas tarefas.

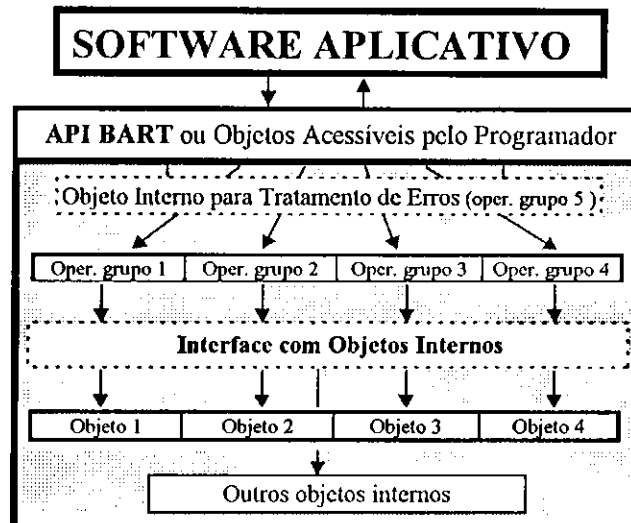


Figura 2.2 - A interação entre API BART e a interface de objetos internos.

2.4. Principais Conceitos

Nos capítulos seguintes onde apresentamos as estruturas dos objetos da API BART, a própria API e exemplos de programação, comumente aparecem alguns termos que possuem significados especiais para a BART. Portanto, para facilitar a compreensão dos referidos capítulos bem como o entendimento dos recursos da biblioteca, esta seção foi dividida em subseções onde cada uma apresenta um conceito ou conceitos relacionados que são tratados pela BART.

2.4.1. Termo e Chave

O *termo* é a menor unidade de informação passível de ser armazenada em um índice. Um termo é descrito por um conjunto de caracteres e é denominado desta forma por poder representar além de palavras, datas, horas ou valores numéricos. Um termo pode representar também uma seqüência de caracteres que formam as máscaras (ver definição na seção 2.4.10.).

Já uma *chave* é o nome dado a um termo após o mesmo ter sido adicionado em um índice, ou seja, após o termo ter sido indexado. É portanto através das chaves que a BART proporciona a rápida recuperação das informações a elas associadas.

2.4.2. Índice e Sistema de Índice

Índice é a estrutura de armazenamento das chaves. No processo de indexação os termos são inseridos em um índice e um valor é associado a ele. Este valor indica a localização física das informações associadas a cada termo.

Originalmente, a BART oferece oito tipos diferentes de índices: índice de palavras, índice de palavras inversas ou backwards, índice de valores, índice de datas, índice de horas, índice de fonemas, índice de referências e índice de chave única.

1. Índice de palavras. Armazena as chaves que representam todos os termos indexados;
2. Índice de palavras inversas. Igualmente ao índice de palavras as chaves deste índice representam todos os termos indexados. Porém, neste índice, as chaves são armazenadas de forma inversa, ou seja, do último caractere para o primeiro;
3. Índice de valores. Armazena as chaves que representam valores numéricos;
4. Índice de datas. Armazena as chaves que representam datas;
5. Índice de horas. Armazena as chaves que representam horas;
6. Índice de fonemas. Neste índice as chaves representam os valores fonéticos dos termos indexados;
7. Índice de referência. Neste índice as chaves representam as referências (definida na seção 2.4.3.) de todos os termos;
8. Índice de Chave Única. Neste índice as chaves representam termos

indexados como chave única.

O *sistema de índices* é um conjunto de índices e estruturas de informações que permitem armazenar e recuperar termos e suas completas localizações. Um sistema de índice gerência índices e arquivos de controle relacionados a uma aplicação. Uma aplicação pode ter n sistemas de índices simultaneamente.

A estrutura do sistema de índices será descrita com mais detalhes no capítulo 4 seção 4.2.2.

2.4.3. Documento, Conjunto, Grupo e Subgrupo

A palavra *conjunto* é usada para representar e/ou referenciar um documento. Para a BART, um *documento* é o conjunto de grupos onde a aplicação mantém seus dados. Um registro de dados ou um arquivo texto são exemplos de documentos.

Um conjunto é formado por um ou mais grupos. Um *grupo* é a entidade que contém os termos que a aplicação deseja indexar. Um grupo pode ser um campo em um registro de dados ou qualquer estrutura de informação armazenada em memória ou disco.

O *subgrupo* possui as mesmas características de um grupo, ou seja, contém termos que a aplicação deseja indexar. Contudo, um subgrupo é usado para tratar campos com uma semântica semelhante à dos campos multivalorados dos bancos de dados. Deste modo, um grupo pode conter vários subgrupos.

Para exemplificar estas definições suponha uma aplicação que possui seus dados estruturados em campos e que estes dados são armazenados em registros de um arquivo. A

BART considera cada registro deste arquivo como sendo um *documento* (ou um *conjunto*) e cada campo do registro é tratado como um *grupo*. Se algum destes campos possuir multivalorações elas serão representadas pelos *subgrupos*.

Analisando um arquivo texto como exemplo teríamos que este arquivo seria um *documento* composto de um único *grupo* e nenhum *subgrupo*.

2.4.4. Referência e Localização

A *referência* é o endereço de identificação de um documento. Ela é composta por três partes onde cada uma possui um valor numérico: conjunto, grupo e subgrupo.

1. o valor do subgrupo identifica o valor da multivaloração de um grupo;
2. o valor do grupo identifica o número do grupo. Se o grupo representar um campo em um registro de dados então o valor do grupo identifica o número do campo. Por outro lado se o grupo representar um arquivo texto qualquer, então este arquivo é considerado como sendo um campo e o valor do grupo identificará este campo.
3. o valor do conjunto identifica o número de um registro em um arquivo de dados. O conjunto pode também identificar um arquivo texto qualquer e, se isso acontecer, todo o arquivo será tratado como sendo um único registro.

Como pôde ser visto na seção 2.3.1.1. a aplicação é quem é responsável pelo gerenciamento de seus respectivos dados. A BART simplesmente mantém uma estrutura de índices que referenciam estes dados. Portanto, para que a biblioteca saiba a referência das informações que está indexando, os respectivos valores para conjunto, grupo e subgrupo

devem ser informados pela aplicação quando a mesma requisitar a indexação de algum grupo.

Além da referência existe outro conjunto de informações que auxiliam a BART a referenciar as informações das aplicações de forma precisa. A *localização* é este outro conjunto de informações e armazena a posição exata de um termo com relação ao grupo ao qual o termo pertence.

Igualmente à referência, uma localização também é composta por três subdivisões representando valores numéricos: parágrafo, frase e seqüência. A necessidade deste conjunto de informações adicional está relacionado ao fato da BART ser uma ferramenta de recuperação textual e, portanto, necessita armazenar informações com tais características. Para isso ser possível, a biblioteca trata cada grupo como sendo um texto único e, ao receber as informações das aplicações (através dos grupos), quebra estas informações em parágrafos, frases dentro dos parágrafos e palavras dentro das frases. Assim temos:

1. um valor identificador do parágrafo onde o termo está localizado;
2. um identificador da frase (em relação ao parágrafo) onde o texto está localizado;
3. um identificador da posição exata do termo dentro da frase.

Ao contrário das informações de referência, as informações da localização não são informadas pelas aplicações. Neste caso os valores de parágrafo, frase e seqüência são gerados internamente para cada grupo de termos que uma aplicação indexar junto a BART. Estes valores são obtidos através de um objeto *parser* que faz a análise gramatical das

informações dos grupos⁷. Detalhes sobre o objeto parser podem ser vistos na seção 4.3.1.

2.4.5. Proximidade

Pesquisa por proximidade é definida como sendo uma busca em todos os lugares onde uma cadeia $c1$ está distante, a pelo menos um número fixo de caracteres (definidos pelo usuário), de uma outra cadeia $c2$. Nesta definição de [Gonn92], a pesquisa por proximidade considera válida a proximidade das cadeias qualquer que seja a cadeia que apareça primeiro no texto, ou seja, considera a proximidade independentemente da ordem das cadeias no texto. Além disso, a proximidade é válida para qualquer cadeia, seja ela uma palavra ou apenas parte dela.

A BART baseia-se neste conceito para implementar este recurso na sua linguagem de pesquisa. Entretanto, a BART faz duas considerações com relação ao conceito descrito acima. Primeiro ela restringe o conceito permitindo apenas cadeias que representem palavras inteiras, e não partes delas. Segundo, a BART qualifica pesquisas por proximidade em duas classes: *pesquisa por intervalo de proximidade* e *pesquisa por proximidade de localização*.

A pesquisa por intervalos de proximidade é uma busca de todos os documentos que possuem uma palavra $p1$ distante em até n palavras (n definido pelo usuário) de uma outra palavra $p2$, independentemente da ordem na qual as palavras aparecem. Neste caso, a BART restringe a proximidade das palavras para palavras dentro da mesma frase, ou seja, somente palavras que estiverem na mesma frase são recuperadas com este tipo de pesquisa.

⁷ Como é de responsabilidade do *parser* fazer a análise gramatical dos grupos, é também de sua responsabilidade definir o que, em cada grupo, é um parágrafo, uma frase e uma seqüência.

A pesquisa por proximidade de localização não requer que seja informado um número máximo de palavras entre as palavras pesquisadas. Ela simplesmente requer um identificador do nível de localização desejado para as duas palavras. Portanto, a pesquisa por proximidade de localização é uma busca de todos os documentos que possuem uma palavra $p1$ num mesmo nível de localização l (definido pelo usuário) de uma outra palavra $p2$, independentemente da ordem na qual as palavras aparecem. O nível de localização l é um identificador para considerar apenas palavras num mesmo grupo, subgrupo, parágrafo ou frase.

2.4.6. Adjacência

O conceito de adjacência é uma evolução do conceito de proximidade definido pela BART na seção anterior. Assim sendo, a pesquisa por adjacência é uma busca de todos os documentos que possuem uma palavra $p1$ distante em até n palavras (n definido pelo usuário) de uma outra palavra $p2$, nesta ordem, ou seja, a palavra $p1$ deve obrigatoriamente aparecer antes da palavra $p2$ na seqüência do texto.

Com base nisso podemos afirmar que a pesquisa por adjacência também é uma pesquisa por intervalo de proximidade de palavras, mas que considera importante a ordem das mesmas. Além disso, igualmente à pesquisa por intervalo de proximidade, a pesquisa por adjacência também restringe a proximidade das palavras para palavras dentro da mesma frase.

2.4.7. Tupla

No contexto de bancos de dados o conceito de tupla é conhecido como um item de uma coleção ou grupo de itens de dados que descrevem uma entidade, sendo similar a um registro de um arquivo ou uma linha em uma tabela. Para a BART o conceito de tupla é bastante semelhante. Neste caso a *tupla* também é um item de um grupo de itens dados que descrevem uma entidade. Porém na BART o grupo de itens de dados é representado por uma lista e a entidade relacionada ao item é um termo.

Existem dois tipos de tupla na BART: a *sêxtupla* e a *sétupla*. Estes dois tipos de tupla são descritos nas próximas seções individualmente.

2.4.7.1. Sêxtupla

As *sêxtuplas* são informações de identificação dos termos, e são utilizadas pela BART em tempo de indexação e desindexação, representando o endereço completo de cada termo. Como o próprio nome sugere, a *sêxtupla* é um objeto composto por seis elementos. Elementos estes oriundos da referência do termo mais suas informações de localização. Portanto, uma *sêxtupla* é formada por:

(conjunto, grupo, subgrupo, parágrafo, frase, seqüência)

Quando um grupo é indexado ou desindexado, cada termo recebe uma *sêxtupla* formada pelas informações referentes à referência (informadas pela aplicação) e as informações referentes à sua localização (geradas internamente). Como é impossível dois termos estarem exatamente na mesma posição de uma mesma frase, é garantido que cada *sêxtupla* seja um identificador único para representar cada termo.

Por outro lado, é possível a ocorrência do mesmo termo em diferentes frases de um grupo e até mesmo a ocorrência do mesmo termo em diferentes grupos. Neste caso, contudo, existirão n sêxtuplas diferentes referenciando um mesmo termo.

2.4.7.2. Sétupla

A *sétupla* é uma evolução da sêxtupla. É utilizada pela BART para fornecer informações precisas a respeito de um termo como resultado de alguma pesquisa.

A sétupla fornece uma informação adicional em relação às sêxtuplas. Igualmente ela fornece as seis informações referentes à identificação dos termos mais uma sétima informação que é o próprio termo cujos valores da sêxtupla estão referenciando. Deste modo temos que uma sétupla é formada pelos seguintes elementos:

(conjunto, grupo, subgrupo, parágrafo, frase, seqüência, termo)

No caso da sétupla todas as informações são fornecidas pela BART, pois trata-se de um resultado de pesquisa que previamente fora armazenado pela própria biblioteca no momento da indexação do grupo.

2.4.8. Stopword

As *stopwords* são as unidades definidas nas *stoplists*. *Stoplist* é uma lista de palavras consideradas sem valor de indexação e usada para eliminar a possibilidade de indexação destas palavras[FRAK92].

Por se tratar de palavras sem valores de indexação e, portanto, sem valor de

pesquisa, as stoplists são usadas para otimizar a indexação eliminando a indexação de palavras que possivelmente nunca serão pesquisadas. O processo de verificação das stopwords se dá durante a indexação de um grupo, onde cada termo com potencial de indexação é verificado. Se o termo verificado estiver em uma stoplist ele não será indexado.

Geralmente as stopwords correspondem a artigos, preposições, pronomes, etc, mas uma aplicação pode utilizar as stopwords para eliminar a indexação de qualquer palavra que julgue não ser importante para seu propósito.

2.4.9. Goword

Ao contrário das stopwords, as *gowords* são palavras que são consideradas com um grande valor de indexação e pesquisa. As listas de *gowords* são usadas para forçar a indexação destas palavras. Por se tratar de um caso particular, o processo de indexação de *gowords* é independente do processo normal de indexação.

Palavras existentes nas listas de *gowords* devem obrigatoriamente ser indexadas caso apareçam nos grupos definidos pelas aplicações. Durante a indexação destes grupos, os termos são procurados nas listas de *gowords* e só serão indexados se estiverem presentes entre as *gowords*.

2.4.10. Máscaras

As *máscaras* são um mecanismo de reconhecimento de padrões que permite a identificação de termos que representem datas, horas ou valores em meio a informações do tipo texto. Com este mecanismo é possível definirmos vários padrões diferentes para cada

uma destas representações a fim de permitir sua identificação. Uma vez que estes padrões forem reconhecidos é possível realizar a indexação de cada um deles no seu respectivo formato. Conforme já dito, estes tipos de dados possuem formatos especiais de armazenamento para permitir recursos especiais de pesquisa utilizando operadores relacionais (<, <=, >, etc).

Um exemplo bastante claro da importância deste recurso está na identificação de datas. Supondo um processo de indexação de um arquivo texto sem reconhecimento de padrões para datas. Teremos que o resultado final deste processo será a indexação de todos os termos que compõem o arquivo no formato de cadeias. Desta forma a pesquisa por intervalos de datas neste arquivo não seria permitida, uma vez que não é possível realizar tal tarefa sobre informações do tipo cadeia.

Por outro lado, se o reconhecimento de padrões de datas fosse feito sobre este arquivo, todas as datas coincidentes com os padrões definidos seriam convertidas para o formato de data e indexado neste respectivo formato em um índice específico. Este procedimento habilitaria a pesquisa por intervalos de datas para este arquivo. De forma análoga, este exemplo também cabe no reconhecimento de padrões de tipo hora e valor.

2.4.11. Normalizador

O normalizador é uma tabela de caracteres auxiliares que são utilizados pela BART. Seu objetivo é converter cada caractere das informações fornecidas pela aplicação em um caractere correspondente definido na tabela de normalizadores da BART. Assim é possível garantir que todos os termos serão indexados da mesma forma assegurando que o resultado de uma consulta não será alterado em função da diferença de caixa dos termos fornecidos

em uma operação de pesquisa.

A tabela de normalizadores é definida pelo programador (ou pela aplicação) e portanto, é ele que define a forma como as conversões serão realizadas.

3. Utilizando os Recursos da BART

Este capítulo, como o próprio nome sugere, apresenta as principais considerações para o desenvolvimento de aplicações que desejarem utilizar os recursos textuais da BART. Essa apresentação está dividida em duas partes. A primeira parte mostra a filosofia de programação BART, através da demonstração dos passos que habilitam a utilização dos recursos.

A segunda parte é destinada à apresentação de exemplos de programação. Os exemplos (apresentados na seção 3.2.) mostram considerações sobre inicialização da BART; criação, abertura e fechamento de sistemas de índices; configuração e gerenciamento de objetos de configuração; utilização de recursos de indexação, desindexação e pesquisa.

3.1. A Filosofia de Programação BART

Antes de desenvolver uma aplicação que utilize os recursos da BART, devemos ter conhecimento da filosofia de programação que envolve a API de programação desta biblioteca. Para isso é importante que, primeiramente, conheçamos as diferentes etapas de desenvolvimento das aplicações que utilizarão os objetos da BART. O fato desta biblioteca ter sido implementada sob o paradigma de orientação a objetos torna importante que conheçamos também a filosofia de programação orientada a objetos e os conceitos

relacionados a esta filosofia.

Dentre estes conceitos destacamos o encapsulamento de dados, herança e parametrização de classes. O encapsulamento de dados possibilita a criação de unidades funcionais bem definidas com acesso rigorosamente controlado a seus dados. Os mecanismos de herança possibilitam o reaproveitamento de classes já existentes na elaboração de novas classes, implicando apenas na definição das diferenças ou evoluções desejadas. As classes paramétricas são importantes porque permitem que sejam definidas classes sem a necessidade de explicitar os tipos de dados que serão utilizados. Com isso a reutilização de códigos através da implementação de rotinas genéricas fica facilitada.

Com relação à filosofia de desenvolvimento da BART temos que uma aplicação que interage com a biblioteca possui quatro etapas distintas de desenvolvimento no que se refere às etapas de execução desta aplicação. São elas: inicialização, preparação, interação de dados e finalização (figura 3.1). Cada uma destas etapas são discutidas nas seções seguintes.

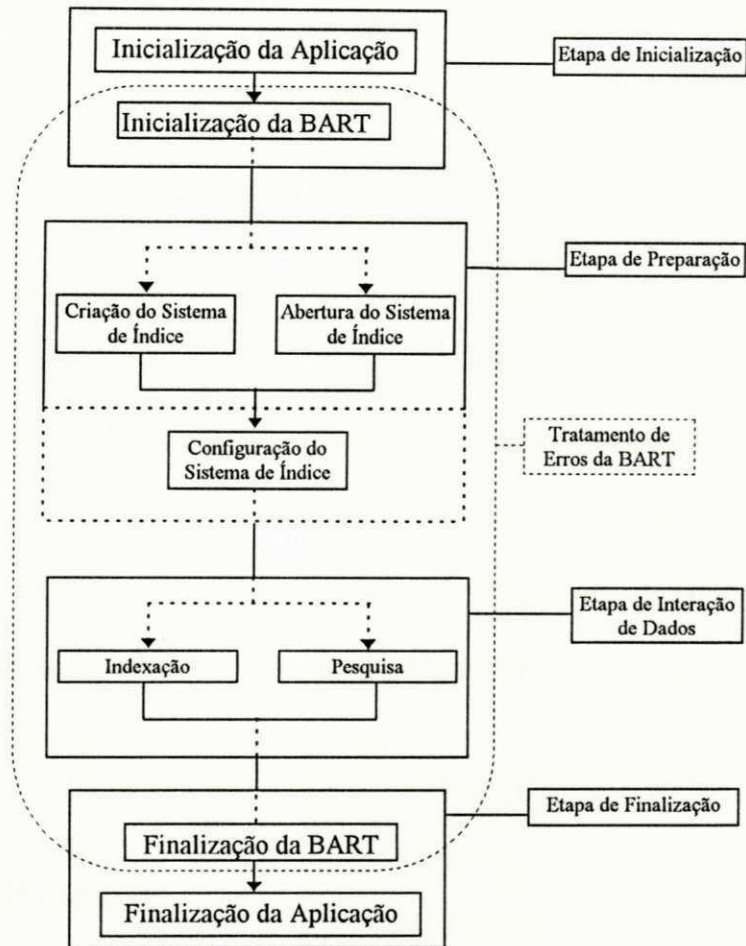


Figura 3.1 - Principais etapas da interação Aplicação / BART.

Embora seja uma tarefa de suma importância, o tratamento de erros não é considerado como uma etapa de desenvolvimento na implementação de aplicações. Esta desconsideração ocorre por não ser o tratamento de erros uma tarefa que ocorra isoladamente. Pelo contrário, esta funcionalidade é uma tarefa comum a todas as demais etapas, sendo portanto, um processo presente em todos os momentos da implementação de cada etapa.

Com exceção da etapa de interação de dados e da fase de configuração de sistemas de índices (etapa de preparação), todas as demais etapas são realizadas através do uso de um objeto global que gerencia a alocação e liberação dos recursos disponíveis na BART.

Este objeto, nomeado *BART_Sessao*, é uma instância da classe *C_BART_SESSAO* (ver definição na seção 4.2.1.). Através deste objeto é possível acessar as rotinas de inicialização, preparação e encerramento da BART.

As funcionalidades da etapa de interação dos dados e configuração de sistemas de índices são acessadas através de objetos da classe denominada *C_BART_SISTEMA_DE_INDICES* (definida em 4.2.2.). Os objetos desta classe são obtidos através das rotinas de preparação (criação e abertura de sistemas de índices). Portanto, para a utilização dos recursos textuais da BART é imprescindível que antes, sejam seguidas as etapas de inicialização e preparação da BART que, a partir deste momento, será também referenciada pelo objeto *BART_Sessao*.

3.1.1. Inicialização

A etapa de inicialização é caracterizada por duas fases distintas: inicialização da aplicação e inicialização da *BART_Sessao*. No que se refere à inicialização das aplicações é difícil apresentarmos alguma consideração já que tal tarefa apresenta particularidades referentes a cada aplicação.

A inicialização da *BART_Sessao* é caracterizada pela chamada do método *Inicializar_BART()* que recebe como parâmetro o número máximo de objetos *C_BART_SISTEMA_DE_INDICES* que poderão ser manipulados simultaneamente. Não é obrigatório que este método seja a primeira tarefa da aplicação a ser executada mas, é obrigatório que ele seja o primeiro recurso da BART a ser utilizado. Qualquer tentativa de execução das rotinas de preparação ou de acesso aos recursos da BART antes da execução deste método resultará em falhas.

Em princípio, o método `Inicializar_BART()` só deve ser chamado uma única vez durante a execução da aplicação. Caso ocorram chamadas repetidas a este método, a aplicação receberá como aviso a indicação que esta tarefa já fora executada anteriormente. As eventuais chamadas repetidas não alteram a configuração inicial definida na primeira chamada.

3.1.2. Preparação

A etapa preparação se refere a dois aspectos. O primeiro deles está relacionado com a criação e abertura de sistemas de índices e o segundo aspecto diz respeito a configuração destes objetos.

Para a realização das tarefas de criação e abertura de sistemas de índices também devemos utilizar o objeto global `BART_Sessao`. Através dele podemos acessar os métodos `BART_CriarSistemaDeIndices()` e `BART_AbrirSistemaDeIndices()`. O método `BART_CriarSistemaDeIndices()` deve ser usado para criar um sistema de índices novo. São parâmetros deste método algumas informações necessárias para a inicialização das estruturas físicas que formarão o sistema de índices. São eles: nome do sistema de índices (sem extensão⁸), tipos de índices a serem criados, tamanho máximo de chave permitido e tamanho das partes inteira e decimal para representação de valores numéricos.

Todos os parâmetros de criação de um sistema de índices são invariáveis, isto é, uma vez definidos não podem mais ser alterados. Quando um sistema de índices é criado, um arquivo de configuração deste sistema de índices é gerado armazenando as informações

⁸A BART anexa automaticamente ao nome do sistema de índices as extensões ".AD" para arquivo de dados, ".AI" para arquivo de índices, ".AC" para arquivo de configuração e ".AR" para arquivos de referências.

dos parâmetros de criação. Futuramente, quando este sistema de índices for aberto, através do método `BART_AbrirSistemaDeIndices()`, os parâmetros de criação são lidos deste arquivo de configuração, não sendo necessária a passagem de todos os parâmetros novamente.

O parâmetro de retorno dos métodos `BART_CriarSistemaDeIndices()` e `BART_AbrirSistemaDeIndices()` é um ponteiro para um objeto do tipo `C_BART_SISTEMA_DE_INDICES`. Quando obtemos um objeto deste tipo as demais rotinas (configuração, indexação, desindexação, pesquisa e finalização) estão automaticamente habilitadas.

Com exceção da configuração dos operadores de pesquisa (definidos em 4.3.4.1.1.) que são configurados de forma global em `BART_Sessao`, todas as demais rotinas de configuração são referentes aos sistemas de índices individualmente. A BART oferece seis possibilidades diferentes de configuração de recursos, que são stopwords, gowords, sinônimos, fonemas, máscaras e normalizadores.

Estes recursos são definidos individualmente por sistema de índices para poder flexibilizar as aplicações (requisito 2.2.1.) no sentido que a aplicação só utilizará estes recursos nos sistemas de índices que necessitarem de recursos mais complexos. Com a definição individual por sistema de índice a aplicação também se reserva ao direito de poder utilizar configurações diferentes para cada sistema de índices.

A configuração dos operadores de pesquisa é definida globalmente por estar relacionada diretamente com a linguagem de pesquisa da aplicação, e não com algum sistema de índice em particular.

Os métodos e os procedimentos para configuração dos recursos citados acima podem ser vistos através dos exemplos, nas seções 3.2.3. e 3.2.3.1. Os detalhes referentes aos parâmetros dos métodos dos objetos de configuração podem ser vistos no apêndice A.

3.1.3. Acesso aos Recursos

Após a etapa de preparação a aplicação estará totalmente pronta para acessar os recursos de indexação e pesquisa da BART. Para isso devem ser utilizados os objetos sistemas de índices obtidos durante a etapa de preparação.

Os objetos sistemas de índices oferecem dois métodos de indexação que podem ser utilizados pela aplicação. O primeiro, chamado *BART_IndexarGrupo()*, deve ser utilizado pela aplicação para realizar a indexação normal dos termos existentes em um grupo. Em princípio, todos os termos existentes nos grupo serão indexados. Porém, caso a aplicação configure alguma lista de stopwords, serão indexados todos os termos existentes no grupo, menos aqueles que estiverem definidos na lista.

O outro método, denominado *BART_IndexarGoWord()*, deve ser utilizado quando a aplicação desejar indexar algum termo especial. Neste caso, os termos do grupo a ser indexado só serão realmente indexados se fizerem parte da lista de gowords configurada pela aplicação. Caso nenhuma lista de gowords seja configurada, o método *BART_IndexarGoWord()* não terá efeito de indexação.

Embora tenham significados semânticos bastante diferentes, ambos os métodos de indexação possuem a mesma sintaxe quanto a passagem de parâmetros nas suas chamadas. Nos dois casos, a aplicação deve informar a referência dos termos - conjunto, grupo e

subgrupo (ver definições na seção de conceitos do capítulo 3), deve informar em que índices os termos do grupo devem ser indexados e deve fornecer o *parser*⁹ que será responsável pela divisão das informações do grupo, separando-as em parágrafos, frases e termos.

No que se refere à desindexação de dados a BART, através dos sistemas de índices, oferece também duas formas para a realização desta tarefa. A primeira delas é através do método *BART_DesindexarGrupo()*, cujo procedimento para desindexação é exatamente igual ao processo de indexação usando o método *BART_IndexarGrupo()*. A aplicação deve fornecer a referência do grupo que quer desindexar, os índices dos quais os termos devem ser removidos e o *parser* que realiza a função de divisão das informações dos grupos em termos.

Outra forma de desindexação de informações é através da utilização do método *BART_DesindexarPorReferencia()*. Neste caso, o procedimento de desindexação é mais simples. Para desindexar informações utilizando este método basta a aplicação fornecer a referência dos dados que deseja desindexar e os índices nos quais estes dados foram indexados.

Embora seja de utilização mais fácil, para ser utilizado, o método de desindexação por referência exige que a aplicação defina a criação do índice de referência no momento da criação do sistema de índices. É este índice que armazena os termos por suas respectivas referência, o que permite que os mesmos sejam desindexados através desta informação. A tentativa de utilização desta opção de desindexação sem a criação do referido índice

⁹O *parser* é um objeto com características bastante importantes e específicas e, por este motivo, será apresentado em uma seção particular no próximo capítulo.

resultará em falhas.

Como recurso de pesquisa a BART possui dois mecanismos distintos para a obtenção dos resultados desejados. O recurso de mais fácil utilização, chamado de pesquisa simples, permite o acesso direto a um índice específico pela pesquisa de uma chave qualquer ou através de métodos que permitem realizar uma varredura seqüencial dos índices em ordem alfabética crescente ou decrescente.

A pesquisa composta, embora um pouco mais complexa, permite que a aplicação utilize a linguagem de consulta da BART (ver seção 4.3.4.1.) com o objetivo de realizar pesquisas mais específicas. Geralmente este tipo de pesquisa é utilizado quando envolve mais de um termo, combinando-os com operadores de pesquisa e operações de restrição sobre o resultado obtido. Ambos os métodos de pesquisa são apresentados através dos exemplos na seção 3.2.5. A seção 4.3.4. do capítulo 4 apresenta com maiores detalhes a linguagem de consulta e a funcionalidade dos operadores de pesquisa.

3.1.4. Finalização

A etapa de finalização consolida as atividades desenvolvidas pela aplicação durante seu período de execução. Ela garante que todos os recursos alocados aos sistemas de índices (memória, abertura de arquivos) sejam efetivamente liberados.

Embora tenha sido definida como a etapa de finalização da BART, na verdade esta etapa é caracterizada pela finalização de operações sobre sistemas de índices. Quando uma aplicação for realmente ser encerrada, ela deve obrigatoriamente utilizar o método *BART_FecharSistemaDeIndices()* para cada sistema de índices que estiver acessando de

forma a garantir a liberação dos recursos alocados a eles.

Contudo, mesmo que a aplicação não estiver sendo finalizada, é perfeitamente possível que ela utilize o referido método apenas como mecanismo de liberação de recursos de forma a estar apta a utilizar outros sistemas de índices ou até mesmo outros recursos particulares que tenha disponível.

O método de finalização de sistemas de índices é um método global, pertencente ao objeto `BART_Sessao`, e deve ser chamado passando-se como parâmetro o sistema de índices que se deseja fechar. Este sistema de índices é representado pelo apontador do sistema de índices obtido através de um dos métodos de preparação da BART, que são `BART_CriarSistemaDeIndices()` e `BART_AbrirSistemaDeIndices()`.

3.2. Exemplos Práticos de Utilização da BART

Esta seção mostra como a filosofia de programação BART, apresentada anteriormente, pode ser utilizada na prática. Os exemplos apresentados não contêm a implementação completa de nenhuma aplicação, mas sim trechos de código (escritos em C++) que demonstram os aspectos mais importantes a serem considerados no desenvolvimento de aplicações. No apêndice B pode ser visto o código de uma aplicação que utiliza os recursos básicos de indexação e pesquisa textual da BART.

Os exemplos apresentados nesta seção seguem a seqüência de inicialização, preparação, interação de dados e finalização (conforme apresentado na seção 3.1.) e apresentam, quando necessário, explicações dos detalhes do código fonte e de eventuais saídas geradas pelos métodos. Detalhes como parâmetros de entrada e de retorno bem

como uma descrição mais detalhada dos métodos devem ser verificados no apêndice A que está reservado para uma apresentação mais detalhada da API BART.

A API BART é composta de 36 classes e 601 métodos, dos quais 11 classes e 129 métodos são oferecidas ao programador. Nesta seção serão apresentadas apenas as classes e métodos mais importantes.

A representação (...) no código do programa exemplo indica algum trecho de código referente a aplicação e que, portanto, não é importante para o propósito desta seção que visa unicamente demonstrar o uso dos recursos da BART. A seqüência "C_BART_" representa as classes de objetos oferecidos pela biblioteca e tipos de dados precedidos pela seqüência "TB_" representam tipos de dados definidos pela BART e usados principalmente como parâmetros dos métodos.

3.2.1. Inicialização

No primeiro trecho de código apresentado demonstramos como uma aplicação deve proceder para inicializar o objeto BART_Sessao de forma a poder ter acesso à etapa de preparação e, conseqüentemente, ter acesso aos recursos textuais da biblioteca. O código simplesmente faz uma chamada ao método Inicializar_BART(), testa seu retorno e, em caso de falha, apresenta uma mensagem de erro na saída padrão.

```

/* Definições de #include necessárias da aplicação */
#include          "bart.h"
const TB_INT     NumSistemaDeIndices = 3;
main()
{
    TB_INT iErro;
    (...)
    if( BART_Sessao.Inicializar_BART( NumSistemaDeIndices ) != 0 ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }
    (...)
}

```

Conforme dito anteriormente, o valor do parâmetro de `Inicializar_BART()` representa o número máximo de objetos da classe `C_BART_SISTEMA_DE_INDICES` que podem ser manipulados simultaneamente pela mesma aplicação.

Podemos observar que o tratamento de erros também é feito através do objeto global `BART_Sessao`. Este objeto oferece métodos para obter o número do erro ocorrido e a descrição deste mesmo erro.

3.2.2. Criação e Abertura de Sistemas de Índices

Nesta seção apresentamos dois exemplos referentes à etapa de preparação. No primeiro demonstramos como criar um novo sistema de índices e no outro mostramos como abrir um sistema de índices já existente.

Exemplo 1:

```

/* Definições de #include necessários da aplicação */
#include      "bart.h"

const TB_CHAR      *NOME_SIST_INDICES = "exemplo";
const TB_INT       INDICES            = PALAVRA | DATA | VALOR;
const TB_INT       TAM_MAX_DA_CHAVE  = 30;
const TB_INT       PARTE_INTEIRA     = 10;
const TB_INT       PARTE_DECIMAL     = 3;

main()
{
    TB_INT          iErro;
    C_BART_SISTEMA_DE_INDICES *SistemaDeIndices;

    (...)

    /* Considerar a execução da inicialização da BART do exemplo 3.2.1. */

    if( (SistemaDeIndices = BART_Sessao.CriarSistemaDeIndices( NOME_SIST_INDICES,
                                                                INDICES, TAM_MAX_DA_CHAVE, PARTE_INTEIRA,
                                                                PARTE_DECIMAL )) == NULL ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }
    (...)
}

```

Neste exemplo, o programador cria um sistema de índices que recebe o nome de “exemplo”. O objeto retornado pelo método de criação (apontado por *SistemaDeIndices*) está automaticamente aberto e pronto para ser utilizado pelos métodos de configuração ou de acesso aos demais recursos da BART.

O sistema de índices criado é composto dos índices de palavras, datas e valores (definição INDICES). O tamanho máximo das chaves é diferente para cada um dos índices criados. Para o índice de palavras este valor é definido por TAM_MAX_DA_CHAVE, que neste caso é igual a 30. Para o índice de valores o tamanho da chave é definido pelas definições de tamanho das partes inteira e decimal e para o índice de datas o tamanho é definido como o tamanho padrão para representação de datas, ou seja, dois bytes para dia, dois bytes para mês e quatro bytes para ano (DD/MM/AAAA).

O exemplo seguinte mostra o procedimento a ser utilizado para a abertura de um sistema de índices já existente.

Exemplo 2:

```

/* Definições de #include necessários da aplicação */
#include "bart.h"

const TB_CHAR *NOME_SIST_INDICES = "exemplo";

main()
{
    TB_INT iErro;
    C_BART_SISTEMA_DE_INDICES *SistemaDeIndices;

    (...)

    /* Considerar a execução da inicialização da BART do exemplo 3.2.1. */
    if( (SistemaDeIndices =
        BART_Sessao.AbrirSistemaDeIndices( NOME_SIST_INDICES )) == NULL ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }
    (...)
}

```

Neste exemplo mostramos a abertura do sistemas de índices criado no exemplo anterior. Podemos observar que no método de abertura somente o nome do sistema de índice é necessário como parâmetro. Todas as demais informações, conforme citado anteriormente, estão armazenadas em um arquivo de configuração gerado no momento da criação do sistema de índices.

3.2.3. Configuração da BART

Os métodos de configuração da BART são acessados através de métodos específicos do sistema de índices. Contudo, as definições do que será configurado são tarefas para alguns objetos, chamados objetos independentes, que são construídos de forma isolada em relação aos sistemas de índices e posteriormente passados como parâmetros nos métodos de configuração.

Para facilitar a compreensão do que são estes objetos independentes, primeiramente apresentaremos exemplos de criação e gerenciamento de alguns destes objetos para só depois mostrarmos como efetivamente estes objetos realizarão a configuração de um sistema de índices.

3.2.3.1. Gerenciamento de Objetos Independentes

Como o número de objetos configuráveis é relativamente grande e a tarefa de configuração, em alguns casos, é semelhante para recursos diferentes, optamos por demonstrar apenas a configuração de dois objetos. O primeiro deles, chamado *C_BART_LISTA_DE_TERMOS*, é um objeto básico na configuração dos recursos. Com exceção da configuração de normalizadores e dos operadores de pesquisa, todos os demais recursos configuráveis dependem deste objeto. O outro objeto, nomeado *C_BART_SINONIMO*, é um objeto que contém definições de termos que são tratados como sinônimos por uma aplicação.

Nos dois exemplos seguintes demonstramos duas formas diferentes para criar um objeto que contenha as definições de algumas stopwords que a aplicação julgue importantes para si. É importante observarmos que nestes exemplos é feita somente a definição do objeto que contém as stopwords, a configuração do recurso propriamente dito ainda não é realizada neste momento.

Exemplo 1:

```

/* Definições de #include necessários da aplicação */
#include "bart.h"

const TB_CHAR *STOPWORDS = "A E O DA DE PARA PORQUE";

main()
{
    TB_INT iErro;
    C_BART_LISTA_DE_TERMOS ListaDeStopWords;
    (...)

    /*Considerar a execução do código de um dos exemplos de 3.2.2.*/

    if( ListaDeStopWords.BART_ConstruirLista( STOPWORDS ) != 0 ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }
    (...)
}

```

Exemplo 2:

```

/* Definições de #include necessários da aplicação */
#include "bart.h"

const TB_INT NUM_STOPWORDS = 7;
const TB_CHAR *Stopwords[] = { "A", "E", "O", "DA", "DE", "PARA", "PORQUE" };

main()
{
    TB_INT iErro;
    int i;
    C_BART_LISTA_DE_TERMOS ListaDeStopWords;
    (...)

    /*Considerar a execução do código de um dos exemplos de 3.3.2.*/

    for( i = 0; i < NUM_STOPWORDS; i++ ){
        if( ListaDeStopWords.BART_AdicionarTermo( Stopwords[i] ) != 0 ){
            iErro = BART_Sessao.ObterErro();
            cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
            (...)
        }
    }
    (...)
}

```

O próximo exemplo demonstra duas etapas para a definição de um objeto que contenha informações de sinônimos definidos pela aplicação.

Neste exemplo também está sendo feita somente a definição do objeto. Igualmente aos exemplos anteriores (stopwords), a configuração do recurso ainda não é realizada neste momento.

```

/* Definições de #include necessários da aplicação */
#include    "bart.h"

main()
{
    TB_INT          iErro;
    C_BART_LISTA_DE_TERMOS  ListaDeTermos;
    C_BART_SINONIMO   Sinonimo;

    (...)

    /*Considerar a execução do código de um dos exemplos de 3.2.2.*/

    if( ListaDeTermos.BART_ConstruirLista( "CASA LAR MORADIA" ) != 0 ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }
    if( Sinonimo.BART_AdicionarSinonimo( "APARTAMENTO", ListaDeTermos,
        BIDIRECIONAL ) != 0 ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }
    (...)
}

```

Podemos observar que a definição de um objeto do tipo sinônimo é feita em duas etapas. A primeira consiste da criação de uma lista de termos (método `ListaDeTermos.BART_ConstruirLista()`) e a segunda realiza a definição destes termos como sinônimo de um outro termo também passado como parâmetro (método `Sinonimo.BART_AdicionarSinonimo()`). O parâmetro *BIDIRECIONAL* indica como, internamente, é feita a definição dos termos passados como parâmetros (ver seção A.9. do apêndice A para maiores detalhes dos parâmetros).

Uma vez que temos definidos os objetos independentes de configuração, os recursos de configuração do sistema de índices já podem ser utilizados para concretizar a etapa de preparação de um sistemas de índices. Assim, encerrando a seção 3.2.3. apresentamos um exemplo que demonstra como utilizar os objetos independentes de configuração para configurar os recursos definidos nos três exemplos da seção 3.2.3.1.

```

/* Definições de #include necessários da aplicação */
#include "bart.h"

main()
{
    TB_INT iErro;
    C_BART_LISTA_DE_TERMOS ListaDeStopWords;
    C_BART_LISTA_DE_TERMOS ListaDeTermos;
    C_BART_SINONIMO Sinonimo;
    C_BART_SISTEMA_DE_INDICES *SistemaDeIndices;

    (...)

    /*Considerar a execução do código de um dos exemplos de 3.2.2.
    e dos exemplos de 3.2.3.1. */

    if( SistemaDeIndices->BART_ConfiguraStopWords( ListaDeStopWords ) == NULL ||
        SistemaDeIndices->BART_ConfiguraSinonimos( Sinonimo ) == NULL ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }
    (...)
}

```

Utilizando-se os métodos `BART_ConfigurarStopWords()` e `BART_ConfigurarSinonimos()`, o sistema de índices *SistemaDeIndices* está preparado para filtrar as stopwords durante a indexação e também está preparado para a realização de pesquisas por sinônimos. Os demais recursos de configuração são também definidos de forma semelhante. Para maiores detalhes dos outros recursos de configuração ver apêndice A.

3.2.4. Recursos de Indexação e Desindexação

Conforme dito na seção 3.1.3. existem dois tipos diferentes de métodos tanto para indexação quanto para a desindexação dos dados. O exemplo que apresentamos nesta seção demonstra a utilização dos métodos `BART_IndexarGrupo()` e `BART_DesindexarGrupo()` para realizar a atualização do conteúdo do campo 3, do registro 5 de uma aplicação. O valor zero no quarto parâmetro indica que o campo 3 não possui multivaloração.

```

/* Definições de #include necessários da aplicação */
#include "bart.h"

const TB_INT INDICES = PALAVRA | DATA | VALOR;

main()
{
    TB_INT iErro;
    C_BART_SISTEMA_DE_INDICES *SistemaDeIndices;
    C_BART_PARSER Parser;

    (...)

    /* Considerar a execução de um dos exemplos de 3.2.2. */

    /*Neste momento a aplicação define no parser os termos a serem desindexados*/
    if( SistemaDeIndices->BART_DesindexarGrupo( INDICES, 5, 3, 0, &Parser ) != 0 ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }

    /*Aqui a aplicação define no parser os termos a serem indexados*/
    if( SistemaDeIndices->BART_IndexarGrupo( INDICES, 5, 3, 0, &Parser ) != 0 ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }
    (...)
}

```

Como as informações de indexação armazenam tanto os termos de cada grupo quanto dados posicionais de cada termo em relação ao grupo é necessário que a aplicação desindexe o conteúdo anterior para só então indexar o conteúdo atualizado. Se a aplicação não desindexar os dados anteriores aos dados atualizados, a nova indexação do grupo poderá estar armazenando informações erradas e até mesmo inexistentes.

3.2.5. Pesquisa

Igualmente aos métodos de indexação e desindexação, existem duas maneiras distintas para a aplicação realizar pesquisas de informações na BART. Nesta seção apresentamos os dois tipos de pesquisa oferecidos pela BART: *pesquisa simples* e *pesquisa composta*. Cada um destes tipos é apresentado em uma seção individual que demonstra as características particulares de cada caso.

3.2.5.1. Pesquisa Simples

A realização de uma pesquisa simples é dividida em três fases. As duas primeiras fases são destinadas à obtenção do resultado desejado (utilizando métodos da classe `C_BART_SISTEMA_DE_INDICES`) e a terceira fase se destina a interpretação do resultado utilizando métodos da classe `C_BART_LISTA_DE_OCORRENCIAS`.

A primeira fase consiste em obter uma determinada chave de um índice. Para isso devemos utilizar um dos seguintes métodos: `BART_ObterChaveCorrente()`, `BART_ObterPrimeiraChave()`, `BART_ObterUltimaChave()`, `BART_ObterProximaChave()`, `BART_ObterChaveAnterior()` ou `BART_ObterChave()`. Maiores detalhes de cada um dos referidos métodos podem ser vistos no apêndice A. Uma vez obtida a chave desejada devemos então obter a lista de ocorrências referente a esta chave. Para isso utiliza-se o método `BART_ObterLOdaChaveCorrente()`. A partir da lista de ocorrências a aplicação pode saber as referências e localizações dos dados e proceder conforme sua necessidade.

Nesta seção apresentamos dois exemplos de utilização da pesquisa simples. No primeiro exemplo mostramos como uma aplicação utiliza os métodos de acesso direto às informações dos sistemas de índices para implementar uma varredura seqüencial no índice de PALAVRA e apresentar os valores das chave obtidas. Neste exemplo não é considerado o tratamento de listas de ocorrências das chaves.

No segundo exemplo mostramos a pesquisa de uma chave específica e a apresentação das informações referentes a primeira ocorrência desta chave através da manipulação da lista de ocorrências da mesma. Para facilitar a compreensão deste exemplo suponha que a função `MostrarRegistro()` chamada neste exemplo receba um identificador

de *conjunto* (que pode ser um registro ou um arquivo qualquer), um identificador de *grupo* e um identificador de *subgrupo*. A partir destes identificadores esta função realiza alguma operação particular para recuperar as informações referenciadas por estes valores e apresenta os dados em algum dispositivo de saída.

Exemplo 1:

```

/* Definições de #include necessários da aplicação */
#include "bart.h"

const TB_INT TamanhoMaximoDaChave = 30;

main()
{
    TB_INT          iErro;
    TB_INT          iRet;
    C_BART SISTEMA_DE_INDICES *SistemaDeIndices;
    TB_CHAR         Chave[ TamanhoMaximoDaChave + 1 ];

    (...)

    /* Considerar a execução de um dos exemplos de 3.2.2. */

    if( (iRet = SistemaDeIndices->BART_ObterPrimeiraChave( PALAVRA, Chave )) < 0 ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    } else if( iRet > 0 ){
        cout << "Índice de palavras está vazio.";
        (...)
    }
    while( 1 ){
        cout << Chave;
        if( (iRet = SistemaDeIndices->BART_ObtemProximaChave( PALAVRA, Chave )) <= 0 ){
            iErro = BART_Sessao.ObterErro();
            cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
            (...)
        } else if( iRet > 0 ){
            break;
        }
    }
    (...)
}

```


Exemplo 2:

```

/* Definições de #include necessários da aplicação */
#include "bart.h"

const TB_INT TamanhoMaximoDaChave = 30;

main()
{
    TB_INT          iErro;
    TB_INT          iRet;
    C_BART SISTEMA DE INDICES      *SistemaDeIndices;
    C_BART_LISTA DE OCORRENCIAS    *OcorrenciasDaChave;
    C_BART_OCORRENCIA             *ReferenciaDaChave;
    TB_CHAR            Chave[ TamanhoMaximoDaChave + 1 ];

    (...)

    /* Considerar a execução de um dos exemplos de 3.2.2. */

    if( (iRet = SistemaDeIndices->BART_ObterChave( PALAVRA, "COMPUTADOR", Chave, CHAVE_IGUAL )) < 0 ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    } else if( iRet > 0 ){
        cout << "Chave não encontrada.";
        (...)
    } else {
        cout << "A chave encontrada é = " << Chave;
    }

    if( (OcorrenciasDaChave = SistemaDeIndices->BART_ObterLOdaChaveCorrente( PALAVRA )) == NULL ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
    } else if( (ReferenciaDaChave = OcorrenciasDaChave->BART_ObterPrimeiraOcorrencia()) == NULL ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
    } else {
        MostrarRegistro( ReferenciaDaChave->BART_ObterConjunto(),
                        ReferenciaDaChave->BART_ObterGrupo(),
                        ReferenciaDaChave->BART_ObterSubGrupo() );
    }

    (...)
}

```

3.2.5.2. Pesquisa Composta

Ao contrário da pesquisa simples, a pesquisa composta não é realizada a partir de simples palavras ou varredura de índices. Esta pesquisa apresenta uma linguagem de consulta (ver seção 4.3.4.1.) que permite a preparação de pesquisas mais elaboradas combinando os operadores oferecidos pela linguagem.

Nesta seção apresentamos um exemplo de utilização do método de pesquisa da classe `C_BART_SISTEMA_DE_INDICES`. A pesquisa é feita utilizando-se a linguagem de consulta da BART. Contudo, neste exemplo consideramos apenas o código que utiliza os recursos de pesquisa. Posterior ao exemplo apresentamos alguns exemplos de expressões de pesquisa, seus significados e resultados que satisfazem as expressões.

```

/* Definições de #include necessários da aplicação */
#include "bart.h"

main()
{
    TB_INT          iErro;
    TB_INT          iRet;
    BART_SISTEMA_DE_INDICES *SistemaDeIndices;
    BART_LISTA_DE_EXPRESSOES *ListaDeExpressoes;
    TB_CHAR         *Expressao;

    (...)

    /* Considerar a execução de um dos exemplos de 3.2.2. */
    Expressao = DigitarExpressao();

    if( (ListaDeExpressoes = SistemaDeIndices->BART_Pesquisar( Expressao )) == NULL ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }
    (...)
}

```

O resultado da pesquisa (no exemplo representado por *ListaDeExpressoes*) é um ponteiro para um objeto do tipo *C_BART_LISTA_DE_EXPRESSOES*. Como o objetivo dos exemplos é demonstrar de forma simples a utilização dos principais recursos da BART, não mostramos aqui exemplo de manipulação deste tipo de objeto. A manipulação e a estrutura dos objetos do tipo *C_BART_LISTA_DE_EXPRESSOES* são apresentadas no capítulo 4, seção 4.3.4.2.

O tabela 3.1 descreve alguns exemplos de expressões de pesquisa e os respectivos resultados que atendem à expressão.

Expressão	Significado / Resultado
("Hardware" OU "Software") E "Produtos"	Recuperar qualquer <i>conjunto</i> que contenha ambos os termos hardware e produtos ou ambos os termos software e produtos
"Hardware" E "Software" E "Produtos"	Recuperar qualquer <i>conjunto</i> que contenha simultaneamente todos os termos hardware, software e produtos
"Hardware" OU "Software" OU "Produtos"	Recuperar qualquer <i>conjunto</i> que contenha qualquer um dos termos hardware, software ou produtos
(^"Hardware" NAFRASE ^"Software")[GRUPO = 5]	Normaliza os termos Hardware e Software e recupera qualquer <i>conjunto</i> que contenha estes dois termos em uma mesma frase (NAFRASE) do <i>grupo</i> identificado pelo valor 5
"Produtos" E ~"Desenvolvimento"	Recupera os <i>conjuntos</i> que possuírem o termo Produto combinado ou com o termo Desenvolvimento ou com qualquer termo sinônimo de Desenvolvimento
'>01/11/1994'	Recupera qualquer <i>conjunto</i> que referencie datas a partir de 01/11/1994
**/12/*"	Recupera qualquer <i>conjunto</i> que referencie qualquer dia de dezembro de qualquer ano
**"	Recupera todos os termos indexadas pela aplicação

Tabela 3.1 - Exemplos de expressões de pesquisa utilizando operadores da linguagem de consulta da BART.

3.2.6. Fechamento de Sistemas de Índices

O fechamento de um sistema de índices é uma tarefa bastante simples de ser realizada. Para realizar esta tarefa devemos utilizar o método `BART_FecharSistemaDeIndices()` do objeto global `BART_Sessao`, como mostrado no exemplo que segue.

```

/* Definições de #include necessários da aplicação */
#include "bart.h"

main()
{
    TB_INT iErro;
    C_BART_SISTEMA_DE_INDICES *SistemaDeIndices;

    (...)

    /* Considerar a execução de um dos exemplos de 3.2.2. */

    if( BART_Sessao.BART_FecharSistemaDeIndices( SistemaDeIndices ) != 0 ){
        iErro = BART_Sessao.ObterErro();
        cout << BART_Sessao.DescreverErro( iErro ) << "Erro nº =" << iErro;
        (...)
    }
    (...)
}

```

Os objetos do tipo C_BART_SISTEMA_DE_INDICES não devem ser removidos pela aplicação. Como é o objeto global BART_Sessao quem aloca os recursos para os objetos sistema de índices criados ou abertos pela aplicação, é de responsabilidade do mesmo objeto realizar a liberação dos recursos por ele alocados.

Em relação à API BART, além dos recursos que foram apresentados destacamos ainda a existência de métodos que proporcionam recursos como definição, manipulação e configuração de gowords, fonemas e máscaras; configuração de mnemônicos para os operadores da linguagem de consulta; recursos para combinação de listas de ocorrências que se baseiam na mesma semântica dos operadores de pesquisa; mecanismos de armazenamento de registros ou dados de tamanhos variáveis; recursos de armazenamento de resultados de pesquisa além de outros recursos que podem ser vistos com detalhes de sintaxe e passagem de parâmetros nas seções do apêndice A.

4. Considerações de Implementação da BART

Neste capítulo apresentamos os detalhes de implementação e as estruturas dos principais objetos da BART. Inicialmente mostramos uma visão geral destes objetos e a relação existente entre eles. Posterior a isso demonstramos as definições das classes dos principais objetos e como as estruturas de dados destes objetos ficam organizadas a partir da etapa de inicialização. Finalmente apresentamos a organização das estruturas de dados durante a etapa de interação de dados na qual realmente ocorre a troca de dados entre as aplicações e a biblioteca.

4.1. Visão Geral dos Objetos

Conforme a filosofia de programação apresentada no capítulo anterior, a BART pode ser definida como um conjunto de objetos direcionados à realização de quatro etapas específicas. Para a realização destas etapas o programador deve seguir a seqüência de inicialização, preparação, interação de dados e finalização. Para isso ele necessita inicialmente utilizar um objeto global já oferecido pela BART, receber os objetos retornados pelos métodos da BART e criar diretamente alguns objetos que devem ser fornecidos para a biblioteca durante a realização das etapas principais.

O objetivo desta seção é apresentar a relação existente entre os objetos da BART. Com isso esperamos esclarecer quando o programador deverá utilizar os objetos fornecidos

ou retornados pela BART e quando é de responsabilidade do programador instanciar diretamente os objetos que necessita. A figura 4.1 mostra uma visão geral do objetos da BART apresentando as relações existentes entre os objetos que a compõem.

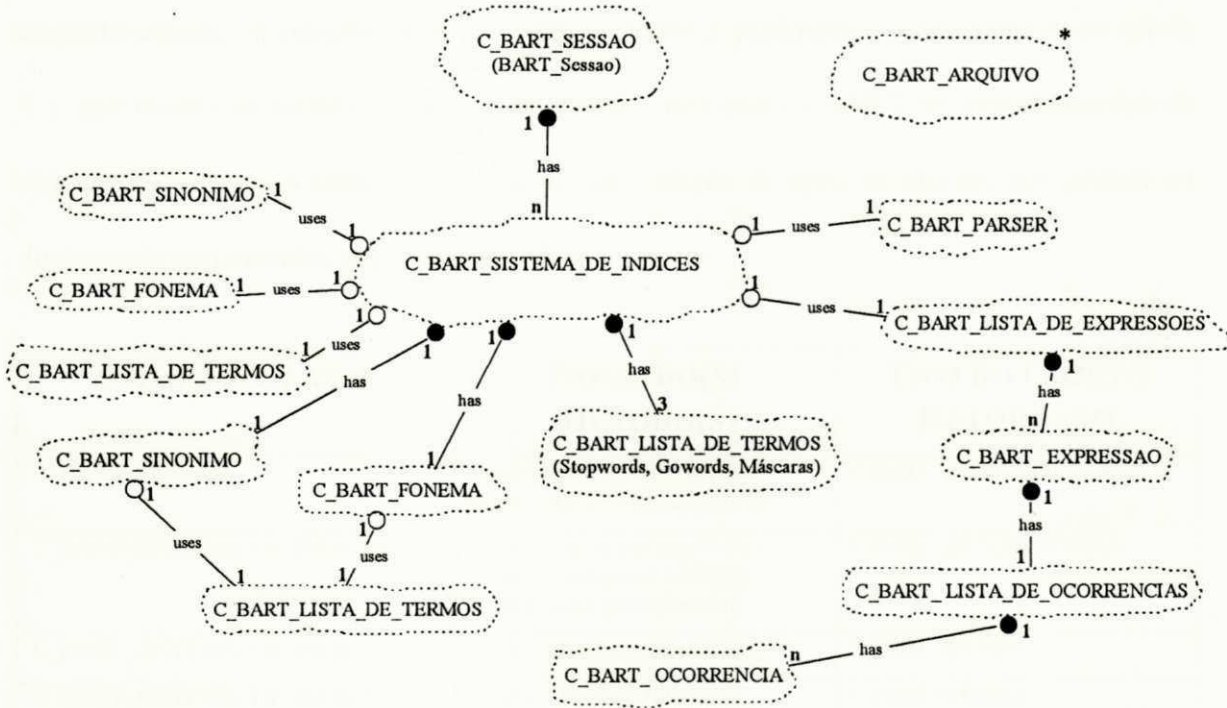


Figura 4.1 - Visão geral dos objetos da BART sob o Diagrama Comentado de Booch¹⁰.

Os objetos C_BART_SESSAO e C_BART_SISTEMA_DE_INDICES como podemos observar são os dois principais objetos da biblioteca. O primeiro por gerenciar os recursos de alocação de sistemas de índices e por permitir a utilização e acesso aos mesmos; o segundo por estar relacionado diretamente com os principais recursos textuais da BART além dos principais recursos de configuração.

¹⁰ O diagrama para representação de classes proposto por [Booch94] é bastante simplificado e bem conhecido pelos grupos de desenvolvimento em C++[Horstm95].

* O objeto C_BART_ARQUIVO possui funções bastante particulares e não apresenta qualquer tipo de relação com nenhum outro objeto da BART.

Podemos observar também que existem objetos que são atributos de outros objetos e objetos que são utilizados como parâmetros de entrada ou saída de métodos de outros objetos. Para melhor entendimento da relação objetos → atributos ver a definição das classe C_BART_SESSAO e C_BART_SISTEMA_DE_INDICES nas seções 4.2.1. e 4.2.2. respectivamente. A relação existente entre métodos e parâmetros é apresentada na tabela 4.1, que mostra as classes e respectivos métodos nos quais a BART retorna algum tipo de objeto. Em seguida, a tabela 4.2 apresenta uma relação de tipos de objetos que podem ser diretamente instanciados pelo programador.

NOME DA CLASSE	NOME DO(S) MÉTODO(S)	TIPO DO OBJETO RETORNADO
C_BART_SESSAO	BART_CriarSistemaDeIndices, BART_AbrirSistemaDeIndices	C_BART_SISTEMA_DE_INDICES
C_BART_SISTEMA_DE_INDICES	BART_ConfigurarStopWords, BART_ConfigurarGoWords, BART_ConfigurarMascaras	C_BART_LISTA_DE_TERMOS
C_BART_SISTEMA_DE_INDICES	BART_ConfigurarSinonimos	C_BART_SINONIMO
C_BART_SISTEMA_DE_INDICES	BART_ConfigurarFonemas	C_BART_FONEMA
C_BART_SISTEMA_DE_INDICES	BART_Pesquisar	C_BART_LISTA_DE_EXPRESSOES
C_BART_SISTEMA_DE_INDICES	BART_ObterLOChaveCorrente	C_BART_LISTA_DE_OCORRENCIAS
C_BART_LISTA_DE_EXPRESSOES	BART_ObterPrimeiraExpressao, BART_ObterProximaExpressao, BART_ObterUltimaExpressao, BART_ObterExpressaoAnterior, BART_ObterExpressaoResultante	C_BART_EXPRESSAO
C_BART_EXPRESSAO	BART_ObterListaDeOcorrencias	C_BART_LISTA_DE_OCORRENCIAS
C_BART_LISTA_DE_OCORRENCIAS	BART_ObterPrimeiraOcorrencia, BART_ObterUltimaOcorrencia, BART_ObterProximaOcorrencia, BART_ObterOcorrenciaCorrente, BART_ObterOcorrenciaAnterior, BART_ObterOcorrencia	C_BART_OCORRENCIA
C_BART_SINONIMO	BART_ObterSinonimos	C_BART_LISTA_DE_TERMOS

Tabela 4.1 - Tipos de objetos retornados por métodos de outros objetos da BART.

A tabela seguinte descreve os objetos que podem ser instanciados diretamente pelo programador, situações em que a BART necessita que tal procedimento seja realizado e

exemplos de situações onde o programador pode criar alguns objetos e utilizá-los como recursos particulares da aplicação.

NOME DA CLASSE	RECURSO DA BART QUE NECESSITA DO OBJETO
C_BART_ARQUIVO	Nenhum. Criado pelo usuário como recurso particular de armazenamento de dados da aplicação.
C_BART_EXPRESSAO	Nenhum. Pode ser criado para recuperar resultados de pesquisa previamente armazenados em disco.
C_BART_FONEMA	O recurso de configuração de fonemas utilizados na indexação e pesquisa fonética exige que o usuário crie um objeto desta classe e defina os valores válidos para as referidas operações.
C_BART_OCORRENCIA	Nenhum. Pode ser criado como objeto auxiliar na comparação de ocorrências em uma lista de ocorrências.
C_BART_LISTA_DE_EXPRESSOES	Nenhum. Pode ser criado para recuperar resultados de pesquisa previamente armazenados em disco.
C_BART_LISTA_DE_OCORRENCIAS	Nenhum. Pode ser criado para recuperar resultados de pesquisa previamente armazenados em disco ou como objeto auxiliar para receber resultados de operações entre listas.
C_BART_PARSER	Imprescindível nas operações de indexação e desindexação. Deve ser criado e implementado pelo programador. (Ver seção 4.3.1. para maiores detalhes)
C_BART_SINONIMO	O recurso de configuração de sinônimos utilizados na pesquisa por sinônimos exige que o usuário crie um objeto desta classe e defina os termos relacionados da maneira que desejar.
C_BART_LISTA_DE_TERMOS	Este objeto é necessário nas configurações de máscaras, stopwords e gowords e necessário na definição dos valores dos objetos C_BART_FONEMA e C_BART_SINONIMO.

Tabela 4.2 - Tipos de objetos que podem ou que precisam ser criados pelo programador.

É de responsabilidade do programador remover todos os objetos obtidos através de chamadas aos métodos da BART (tabela 4.1.), exceto os objetos da classe C_BART_SISTEMA_DE_INDICES. Os objetos desta classe devem ser removidos utilizando-se o método BART_Sessao.FecharSistemaDeIndices(), conforme visto no capítulo anterior. É também de responsabilidade do programador remover todos os objetos diretamente criados (tabela 4.2.), mesmo que estes objetos tenham sido utilizados em algum recurso de configuração.

Embora os recursos de programação da linguagem C++ permitam que o usuário instancie diretamente qualquer objeto da BART, ele não deve fazê-lo para os objetos das classes `C_BART_SESSAO` e `C_BART_SISTEMA_DE_INDICES`.

Em relação ao primeiro, a BART já oferece o objeto `BART_Sessao` criado globalmente. Este objeto é acessado por todos os demais objetos da biblioteca quando estes precisam utilizar os recursos de tratamento de erros e, além disso, este objeto gerencia a liberação ou não de recursos e fornece informações relevantes para os objetos da classe `C_BART_SISTEMA_DE_INDICES`. Portanto, a criação de um ou mais objetos da classe `C_BART_SESSAO` além do objeto `BART_Sessao` já oferecido, certamente ocasionará problemas referentes ao tratamento de erros da BART e à liberação de recursos para criação de novos sistemas de índices. Isto sem dúvida prejudicaria o bom funcionamento dos demais objetos da biblioteca.

Além disso, como não fazia parte dos objetivos deste trabalho a implementação dos algoritmos de uma árvore B+, consideramos a utilização de um código já existente (biblioteca) para a implementação das classes `C_BART_SESSAO` e `C_BART_INDICE` (interna). Entretanto o código da biblioteca utilizada se baseia em diversas variáveis globais para controle interno de suas estruturas. Desta forma, a cada instanciação de um novo objeto da classe `C_BART_SESSAO` estas globais seria reinicializadas violando as informações das sessões já existentes. Esta é, portanto, a principal razão da impossibilidade de instanciação de uma ou mais sessões além do objeto `BART_Sessao` já oferecido.

Os objetos da classe `C_BART_SISTEMA_DE_INDICES` não devem ser instanciados diretamente por uma razão bastante simples. Ao criar um objeto da classe `C_BART_SISTEMA_DE_INDICES`, o objeto `BART_Sessao` informa ao objeto criado um

número de identificação. O sistema de índices utiliza este número em combinação com valores de alguns atributos próprios, para formar o identificador (*handle*) que lhe permite acessar o arquivo de índices correto.

Como o programador não dispõe deste valor, para criar um objeto da classe `C_BART_SISTEMA_DE_INDICES` ele teria que fornecer um número aleatório, uma vez que ele desconhece a origem do valor fornecido por `BART_Sessao`. Através deste procedimento o programador dificilmente conseguirá ter acesso ao arquivo de índices corretamente e, portanto ele poderá estar comprometendo a integridade dos dados do sistema de índices. Por esta razão os objetos da classe `C_BART_SISTEMA_DE_INDICES` devem ser acessados ou criados utilizando-se os métodos adequados do objeto `BART_Sessao`.

4.2. Objetos Principais

Como visto no capítulo 4, as etapas de inicialização e preparação são os dois primeiros passos obrigatórios para que tenhamos acesso aos recursos da BART.

O serviço da etapa de inicialização (executado pelo método `Inicializar_BART()`) é responsável pela preparação da estrutura, em memória, que possibilita a criação e abertura de sistemas de índices e, conseqüentemente, a utilização dos demais recursos da biblioteca.

Quanto à etapa de preparação, o serviço de criação de sistemas de índices é responsável pela inicialização das estruturas físicas de armazenamento da BART. Este método (`BART_CriarSistemaDeIndices()`) cria e inicializa os arquivos onde a BART armazena índices, dados e informações referentes a parâmetros de criação.

Conforme dito anteriormente os objetos das classes C_BART_SESSAO e C_BART_SISTEMA_DE_INDICES são os principais objetos oferecidos pela BART. Cada um destes objetos apresenta características particulares que são apresentadas individualmente nas seções seguintes (4.2.1. e 4.2.2.), juntamente com as respectivas estruturas de dados utilizadas para a inicialização da memória, criação das estruturas físicas da BART e oferecimento dos recursos de indexação e recuperação textual da BART.

4.2.1. Objeto Sessão

As principais funções da classe C_BART_SESSAO são armazenar informações referentes à estrutura de memória que gerencia a liberação ou não da criação ou abertura de sistemas de índices e gerenciar um objeto que realiza o tratamento de erros. A estrutura de memória que gerencia a liberação de recursos é alocada dinamicamente em função do valor do parâmetro de inicialização do objeto BART_Sessao. Esta estrutura é apresentada com maiores detalhes na seção seguinte (4.2.1.1.).

A estrutura de dados e definição da classe C_BART_SESSAO são apresentadas logo abaixo e, em seguida, é apresentada uma rápida descrição dos atributos e métodos principais da referida classe. A descrição é feita de acordo com a numeração indicada no início de cada linha.

```

1. class C_BART_SESSAO {
2.     private:
3.         C_BART_SISTEMA_DE_INDICES    **pcSistemaDeIndices;
4.         C_BART_TABELA_DE_SIMBOLOS    cTabelaDeSimbolos;
5.         TB_INT                       iStatusDeInicializacao;
6.         TB_INT                       iNumMaximoDeSistemasDeIndices;
7.         C_BART_ERRO                  BART_Erro;
8.
9.     /* As definições dos métodos privados da classe não são apresentadas por
10.        não influenciarem no entendimento da mesma */
11.
12.     public:
13.         C_BART_SESSAO();
14.         ~C_BART_SESSAO();
15.         C_BART_SISTEMA_DE_INDICES    *BART_CriarSistemaDeIndices(TB_CHAR *, TB_INT, TB_INT,

```

```

16.                                     TB_INT, TB_INT );
17. C_BART_SISTEMA_DE_INDICES          *BART_AbrirSistemaDeIndices( TB_CHAR * );
18. TB_INT                               BART_FecharSistemaDeIndices( C_BART_SISTEMA_DE_INDICES * );
19. TB_INT                               BART_RemoverSistemaDeIndices( TB_CHAR * );
20. TB_INT                               Inicializar_BART( TB_INT );
21. TB_INT                               BART_ObterErro( void );
22. TB_CHAR                              *BART_DescreverErro( TB_INT );
23. TB_INT                               BART_ObterPrimeiroErro( void );
24. TB_INT                               BART_ObterProximoErro( void );
25. TB_CHAR                              *BART_ConfigurarOperadores( TB_CHAR * );
26. TB_CHAR                              *BART_ObterDefinicaoDeOperador( TB_CHAR * );
27.);

```

3: vetor de pontadores para objetos da classe C_BART_SISTEMA_DE_INDICES através do qual o objeto BART_Sessao gerencia a liberação dos recursos;

4: objeto interno da BART no qual o objeto BART_Sessao armazena e consulta as definições dos operadores e símbolos válidos na linguagem de consulta;

5: indicador de inicialização. Através deste estado o objeto sabe se já foi ou não inicializado, protegendo-se das repetidas chamadas ao método de inicialização;

6: atributo de controle interno. Armazena o valor passado no parâmetro de inicialização;

7: objeto interno da BART que armazena os valores de erros ocorridos e erros decorrentes em uma operação qualquer da BART;

15, 17, 18: métodos de criação, abertura e fechamento (respectivamente) de sistemas de índices. O método de criação recebe o nome e alguns parâmetros de configuração do sistema de índices a ser criado. O método de abertura recebe somente o nome do sistema de índices a ser aberto e o método de fechamento recebe o apontador do objeto sistema de índices que se deseja fechar, obtido a partir de um dos dois métodos anteriores;

19: método para remoção física de um sistema de índices. Recebe o nome do sistema de índices como parâmetro;

20: método de inicialização da biblioteca. Recebe como parâmetro o número de objetos da classe `C_BART_SISTEMA_DE_INDICES` que poderão ser manipulados simultaneamente;

21, 22, 23, 24: métodos para obtenção dos valores de erros ocorridos durante alguma operação da BART;

25: método de configuração dos operadores e símbolos válidos na linguagem de consulta da BART. Sempre que o valor de algum símbolo ou operador for alterado, o método retornará às definições antigas para todos os valores;

26: método para a obtenção da definição atual de um operador ou símbolo da tabela de símbolos da BART. Recebe o operador ou símbolo *default* e retorna o valor atualizado para o referido operador.

4.2.1.1. Estrutura de Memória

O método `BART_Sessao.Inicializar_BART()`, como visto anteriormente, recebe como parâmetro o número máximo (n) de objetos da classe `C_BART_SISTEMA_DE_INDICES` que podem ser manipulados simultaneamente. Com base neste valor n o objeto `BART_Sessao` prepara um vetor de apontadores para objetos deste tipo (figura 4.2a) no qual serão inseridos os endereços dos objetos criados ou abertos pelos métodos de criação e abertura respectivamente (figura 4.2b).

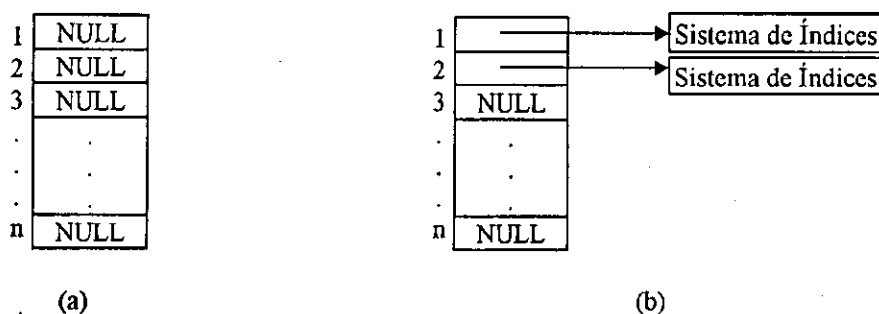


Figura 4.2 - Estrutura de gerenciamento para alocação de sistemas de índices.

Através deste vetor o objeto BART_Sessao gerencia a quantidade de recursos disponíveis para a criação ou abertura de outros sistemas de índices.

Em razão desta estrutura devemos utilizar o método BART_Sessao.FecharSistemaDeIndices() quando desejarmos encerrar as operações sobre um determinado sistema de índices e nunca utilizar qualquer operador da linguagem C++.

Se a operação de remoção for feita utilizando-se diretamente os operadores da linguagem C++ (*delete SistemaDeIndices*, por exemplo) a estrutura interna do objeto BART_Sessao estará sendo violada e acabará armazenando endereços de memória que não estão mais alocados em seu espaço de endereçamento. Tal operação certamente ocasionará erros de execução da aplicação.

4.2.2. Objeto Sistema de Índice

```

1. class C_BART_SISTEMA_DE_INDICES (
2.     private:
3.         TB_INT             iNumeroDoSistemaDeIndices;
4.         TB_INT             iTamanhoMaximoDaChave;
5.         TB_INT             iTamanhoDaParteInteira;
6.         TB_INT             iTamanhoDaParteDecimal;
7.         C_BART_ARQUIVO_DE_DADOS *pcArquivoDeListasDeOcorrencias;
8.         C_BART_ARQUIVO_DE_DADOS *pcArquivoDeTermos;
9.         TB_CHAR            *pszNomeDoSistemaDeIndices;
10.        TB_CHAR            *pTabelaDeNormalizadores;
11.        C_BART_LISTA_DE_TERMOS *pcStopWord;
12.        C_BART_LISTA_DE_TERMOS *pcGoWord;
13.        C_BART_LISTA_DE_TERMOS *pcMascarasDeValores;
14.        C_BART_LISTA_DE_TERMOS *pcMascarasDeDatas;
15.        C_BART_LISTA_DE_TERMOS *pcMascarasDeHoras;
16.        C_BART_INDICE        *pcIndices[ MAX_INDEX ];

```

```

17.     C_BART_FONEMA                *pcFonema;
18.     C_BART_SINONIMO              *pcSinonimo;
19.     C_BART_ARQUIVO_DE_TRAVAMENTO *pcTravamento;
20.     C_BART_TABELA_HASH<C_BART_TERMO> cthStopWord;
21.     C_BART_TABELA_HASH<C_BART_TERMO> cthGoWord;
22.
23.     /* As definições dos métodos privados da classe não são apresentadas por
24.        não influenciarem no entendimento da mesma */
25.
26.     public:
27.     C_BART_SISTEMA_DE_INDICES( void );
28.     ~C_BART_SISTEMA_DE_INDICES( void );
29.     C_BART_LISTA_DE_TERMOS        *BART_ConfigurarMascaras( C_BART_LISTA_DE_TERMOS *, TB_INT );
30.     C_BART_LISTA_DE_TERMOS        *BART_ConfigurarGoWords( C_BART_LISTA_DE_TERMOS * );
31.     C_BART_LISTA_DE_TERMOS        *BART_ConfigurarStopWords( C_BART_LISTA_DE_TERMOS * );
32.     C_BART_FONEMA                 *BART_ConfigurarFonemas( C_BART_FONEMA * );
33.     C_BART_SINONIMO               *BART_ConfigurarSinonimos( C_BART_SINONIMO * );
34.     TB_CHAR                       *BART_ConfigurarTabelaDeNormalizadores( TB_CHAR * );
35.     TB_INT                        BART_IndexarGoWords( TB_INT, TB_ULONG, TB_UINT, TB_UINT,
36.                                                         C_BART_PARSER * );
37.     TB_INT                        BART_IndexarGrupo( TB_INT, TB_ULONG, TB_UINT, TB_UINT,
38.                                                       C_BART_PARSER * );
39.     TB_INT                        BART_DesindexarGrupo( TB_INT, TB_ULONG, TB_UINT,
40.                                                         TB_UINT, C_BART_PARSER * );
41.     TB_INT                        BART_DesindexarPorReferencia( TB_INT, TB_ULONG,
42.                                                                  TB_UINT, TB_UINT );
43.     C_BART_LISTA_DE_EXPRESSOES     *BART_Pesquisar( TB_CHAR * );
44.     TB_INT                        BART_RemoverChaveCorrente( TB_INT );
45.     TB_INT                        BART_ObterChaveCorrente( TB_INT, TB_CHAR * );
46.     TB_INT                        BART_ObterPrimeiraChave( TB_INT, TB_CHAR * );
47.     TB_INT                        BART_ObterUltimaChave( TB_INT, TB_CHAR * );
48.     C_BART_LISTA_DE_OCORRENCIAS    *BART_ObterLOdaChaveCorrente( TB_INT );
49.     TB_INT                        BART_ObterProximaChave( TB_INT, TB_CHAR * );
50.     TB_LONG                       BART_ObterNumeroDeOcorrenciasDaChaveCorrente( TB_INT );
51.     TB_LONG                       BART_ObterNumeroDeChaves( TB_INT );
52.     TB_INT                        BART_ObterChaveAnterior( TB_INT, TB_CHAR * );
53.     TB_INT                        BART_ObterChave( TB_INT, TB_CHAR *, TB_CHAR *, TB_INT );
54.     TB_INT                        BART_Normalizar( TB_CHAR *, TB_CHAR *, TB_INT );
55.     TB_INT                        BART_TravarSistemaDeIndices( TB_INT );
56.     TB_INT                        BART_DestravarSistemaDeIndices( void );
57.     TB_INT                        BART_ZerarSistemaDeIndices( void );
58.);

```

3: número do sistema de índices. Este valor é informado ao sistema de índices pelo objeto BART_Sessao quando da sua criação ou abertura;

4: tamanho máximo da chave. Este valor é definido na criação do sistema de índices e vale para os índices de palavras, backwards e chave única. Para o índice de fonemas o tamanho máximo da chave é o dobro do valor definido para o índice de palavras. Este índice é definido desta forma porque cada caractere fonético é representado por um caracter especial que ocupa dois bytes. O tamanho das chaves de referências, datas e horas são fixos em função dos formatos das mesmas, ou seja, oito bytes para referência (quatro para CONJUNTO, dois para GRUPO e dois para SUBGRUPO), oito bytes para data

(AAAAMMDD - no formato interno) e quatro bytes para horas (HHMM - no formato interno);

5, 6: a parte inteira e a parte decimal definem o tamanho máximo da chave do índice de valores usado no armazenamento de representações numéricas;

7: objeto interno da BART que gerencia o armazenamento e recuperação das listas de ocorrências das chaves em um sistema de índices;

8: objeto interno da BART que gerencia o armazenamento e recuperação das listas de termos indexados sob uma mesma referência;

9: armazena o nome do sistema de índices, sem a extensão. A este nome são concatenadas as devidas extensões para que a BART possa preparar os arquivos referentes ao sistema de índices;

10: representa a tabela de normalizadores definida para o referido sistema de índices;

11, 12: apontadores para objetos do tipo C_BART_LISTA_DE_TERMOS que contém cópias definições de stopwords e gowords respectivamente;

13, 14, 15: apontadores para objetos do tipo C_BART_LISTA_DE_TERMOS que contém as definições de máscaras para os índices de valores, datas e horas respectivamente;

16: vetor de apontadores para objetos do tipo C_BART_INDICE (objeto interno) da BART, que utilizam uma estrutura de indexação baseada nas estruturas de árvores B+;

17: apontadores para objetos do tipo C_BART_FONEMA que contém as definições

de fonemas realizadas pelas aplicações e configuradas através do método de configuração de fonemas;

18: apontadores para objetos do tipo `C_BART_SINONIMO` que contém as definições de sinônimos realizadas pelas aplicações e configuradas através do método de configuração de sinônimos;

19: objeto interno da BART que gerencia o travamento dos arquivos de um sistema de índices quando uma aplicação estiver utilizando os recursos de indexação/desindexação;

20, 21: objetos internos da BART que armazenam as definições de stopwords e gowords e oferecem uma pesquisa com chave *hash* para a rápida localização dos termos definidos;

29, 30, 31, 32, 33, 34: métodos utilizados nas configurações dos objetos utilizados pelo sistema de índices para a utilização dos referidos recursos. Qualquer um dos métodos de configuração retorna os valores anteriormente definidos;

35, 37, 39: métodos para indexação ou desindexação de grupos. Recebe os índices que serão utilizados pelo procedimento, os valores de conjunto, grupo e subgrupo e um objeto do tipo `C_BART_PARSER` que realiza a análise léxica dos grupo;

41: método para a desindexação de termos através de sua referência. Recebe os índices de onde os termos devem ser desindexados e os valores que compõem a referência dos termos;

43: método que permite a utilização da linguagem de consulta da BART para a

realização de pesquisas compostas. A definição de pesquisa composta bem como o detalhamento da estrutura do objeto retornado por este método podem ser vistas na seção 4.3.4.2. deste capítulo;

44: método utilizado para a remoção de uma chave no índice definido como parâmetro;

45, 46, 47, 48, 49, 50, 51, 52, 53: métodos utilizados para a realização de pesquisas simples em um sistema de índices. A definição de pesquisa simples pode ser vista na seção 3.2.5.1. do capítulo anterior;

54: método utilizado para a realização da normalização de um conjunto de caracteres;

55, 56: métodos utilizados para o travamento e liberação de um sistema de índices nas operações de indexação e desindexação;

57: método que realiza a reinicialização de um sistema de índices, removendo todas as suas informações, mas mantendo os valores e estruturas configuradas na sua criação. Todas as configurações e definições feitas no sistema de índices também são preservadas.

4.2.2.1. Estruturas em Disco

Quando um sistema de índices é criado, algumas estruturas em memória secundária são inicializadas para permitir que a BART armazene seus dados e informações de maneira otimizada. Por se tratar de estruturas diferentes cada uma delas está relacionada com um arquivo diferente.

Quando estas estruturas são inicializadas pelo menos três arquivos são criados: um arquivo de configuração, um arquivo de índices e um arquivo de referências das chaves. Se na criação de um sistema de índices for especificada a criação do índice de referências, então um quarto arquivo é também criado. Este último arquivo é responsável pelo armazenamento dos grupos de termos indexados sob uma mesma referência.

O arquivo de configuração representa um arquivo simples de registro único utilizado para armazenar os valores definidos nos parâmetros de criação de um sistema de índices, que são: índices a serem utilizados, tamanho máximo da chave, tamanhos das partes inteira e decimal usadas na representação de valores numéricos. Além dessas informações é também armazenado um campo com informações referentes à versão de implementação.

Conforme dito anteriormente, a BART oferece oito tipos diferentes de índices que podem ser acessados simultaneamente por um mesmo sistema de índices. Cada um destes índices é representado por uma estrutura de dados baseada na implementação de uma B+Tree, também conhecidas como árvore B+. As estruturas das árvores B+ de todos os oito índices são armazenadas no referido arquivo de índices. Cada nodo destas árvores B+ armazena essencialmente uma chave e um valor associado. Este valor associado possui uma semântica diferente dependendo do tipo de índice que está sendo utilizado.

A semântica [CHAVE-VALOR → Localização da Lista de Ocorrências] é válida para a maioria dos índices, exceto para o índice de chave única e para o índice de referências.

Para o índice de chave única esta restrição ocorre pela seguinte razão. Por se tratar de um índice de chave única não é permitido que tenhamos mais de uma ocorrência da

mesma chave, logo não existe a necessidade de armazenar uma lista de ocorrências com uma única ocorrência da chave relacionada. Igualmente aos demais índices os nodos do índice de chave única também armazenam a chave e um valor associado, contudo este valor não representa o endereço físico da lista de ocorrências das chaves, mas sim o número do registro onde a referida chave ocorre.

O índice de referência possui uma outra particularidade no que se refere à semântica das informações que armazena. Enquanto os demais índices seguem a semântica [CHAVE-VALOR → Localização da Lista de Ocorrências] o índice de referência segue uma idéia com características invertidas. A inversão de conceitos ocorre no sentido que, para os outros índices a chave representa um termo e a lista referenciada pelo valor associado contém todas as ocorrências desta chave em um sistema de índices. No índice de referência temos que [CHAVE-VALOR → Localização de Termos] e neste caso a chave é representada não por um termo, mas pela referência dos termos. O valor associado à referência indica o endereço físico onde estão armazenados todos os termos indexados sob a referência representada pela chave.

A figura 4.3 demonstra o esquema lógico das estruturas de indexação destacando a existência de diferentes índices no arquivo de índices e a referência feita por cada índice aos arquivos relacionados. Como o arquivo de configuração não influencia diretamente nas estruturas de indexação das informações ele não é considerado nesta ilustração.

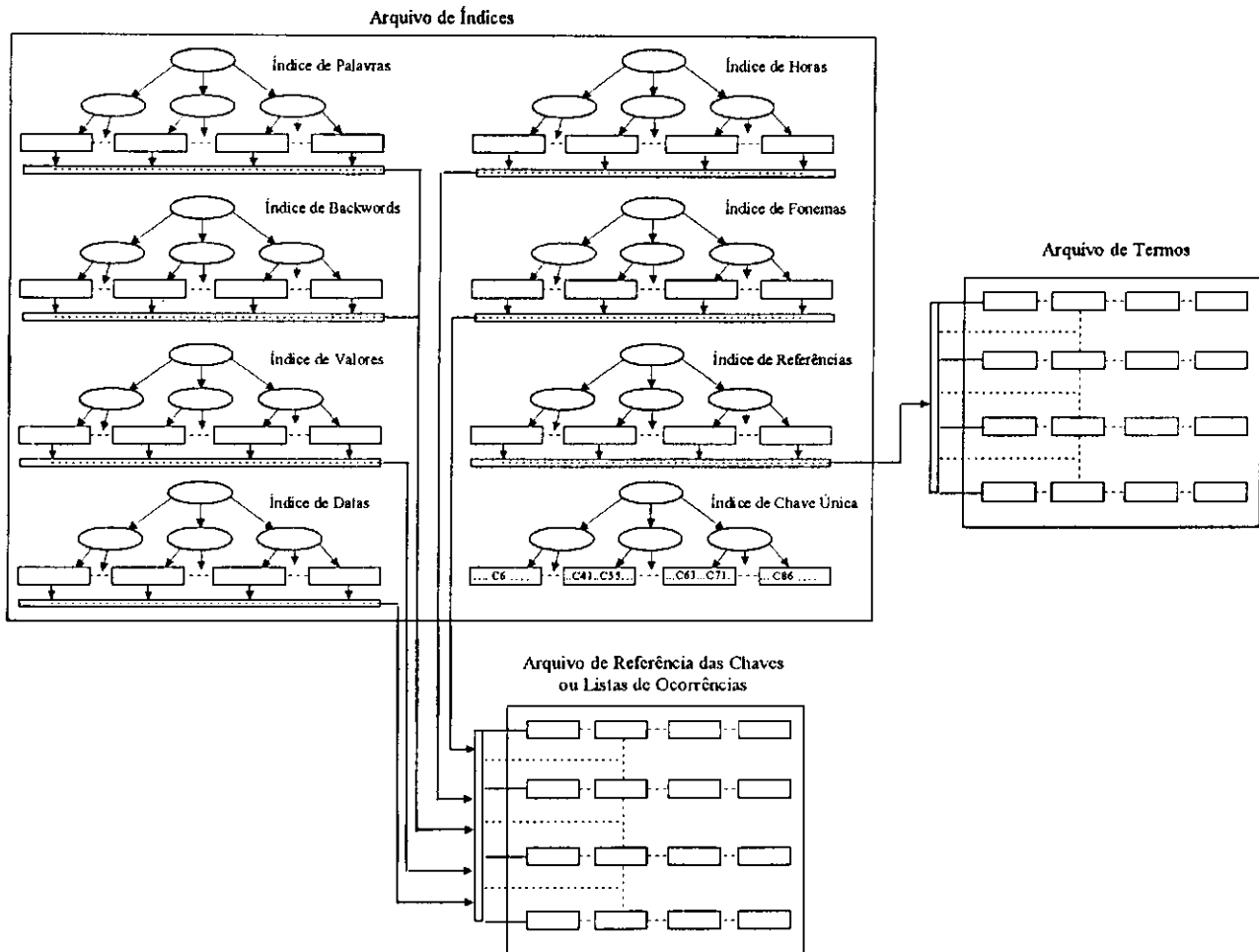


Figura 4.3 - Esquema lógico das estruturas de indexação da BART.

Com este esquema temos os índices de palavras, backword, valores, datas, horas e fonemas referenciando informações no arquivo de listas de ocorrência, o índice de referência referenciando termos no arquivo de termos e o índice de chave única armazenando uma chave e a referência do registro onde esta chave ocorre.

Os índices de palavras e backword são os únicos que possuem duas chaves referenciando a mesma lista de ocorrências. Esta característica particular ocorre porque, embora estando em árvores B+ diferentes, estes índices possuem o mesmo conteúdo, diferenciando-se apenas pela ordem das palavras. Como trata-se da mesma palavra armazenada de duas formas distintas é necessário que ambas as chaves referenciem o

mesmo conteúdo para permitir que consigamos recuperar as mesmas informações, independentemente de qual dos índices utilizemos na pesquisa. Com exceção deste caso, cada uma das demais listas de ocorrências armazenadas são referenciadas apenas por uma única chave.

4.3. Troca de Dados entre a BART e as Aplicações

A troca de dados entre a BART e as aplicações é caracterizada principalmente pela etapa de interação de dados. É nesta etapa que as aplicações efetivamente fornecem seus dados para a biblioteca para que eles sejam indexados, desindexados, armazenados ou consultados. Esta troca de dados é feita através dos objetos oferecidos pela BART, que são trocados sob forma de parâmetros de entrada e de retorno.

As seções seguintes (4.3.1. a 4.3.4.) são destinadas à apresentação das estruturas de dados que envolvem os processos de indexação, desindexação e pesquisa da dados na BART.

4.3.1. Parsers

Um dos requisitos básicos da BART (requisito 2.2.2. - abrangência) é oferecer recursos textuais genéricos para as aplicações, independentemente da natureza das mesmas. Assim qualquer aplicação está apta a utilizar os recursos de indexação e pesquisa da BART. Porém, como a biblioteca se propõe a ser genérica, é impossível que ela tenha conhecimento dos formatos e estruturas de armazenamento de dados de todas as aplicações existentes e das que por ventura vierem a existir.

Para solucionar este problema é preciso que exista um mecanismo onde a BART tenha condições de obter os dados das aplicações a fim de indexá-los (ou desindexá-los) independentemente dos seus formatos ou estruturas de armazenamento. Já que cada aplicação tem o domínio da forma de armazenamento de seus próprios dados, a solução proposta para resolver este problema consiste na adoção de um mecanismo onde, na realidade, quem implementa os algoritmos de obtenção dos dados e transferência dos mesmos para a BART é a própria aplicação.

Este esquema é utilizado através de uma classe da BART (classe C_BART_PARSER) que utiliza o conceito de métodos virtuais na implementação da solução. Para um melhor entendimento do objeto *parser*, apresentamos abaixo a estrutura e definição da classe deste objeto.

```

1. class C_BART_PARSER (
2.     private:
3.         TB_CHAR      *pszTermo;
4.         TB_INT       iEstado;
5.
6.     protected:
7.         TB_INT       BART_SetarTermoDoParser( TB_CHAR * );
8.         TB_INT       BART_SetarEstadoDoParser( TB_INT );
9.
10.    public:
11.        C_BART_PARSER( void ) {};
12.        ~C_BART_PARSER() {};
13.
14.        TB_CHAR      *BART_ObterTermoDoParser( void );
15.        TB_INT       BART_ObterEstadoDoParser( void );
16.
17.        virtual TB_INT BART_InicializarParser( void ) = 0;
18.        virtual TB_INT BART_ProcessarParser( void ) = 0;
19.        virtual TB_INT BART_FinalizarParser( void ) = 0;
20.);

```

3: atributo que contém o termo que a BART utiliza durante o processo de indexação/desindexação;

4: atributo que contém o estado de execução do *parser*, indicando o tipo de caracter encontrado no final do último processamento;

7: método utilizado para setar o atributo (3) durante o processamento do *parser*. Implementado pela BART, mas utilizado pela aplicação na implementação dos métodos virtuais;

8: método utilizado para setar o atributo (4) durante o processamento do *parser*. Implementado pela BART, mas utilizado pela aplicação na implementação dos métodos virtuais;

14: método utilizado para obter o valor setado no atributo (3) durante o processo de indexação/desindexação. Utilizado pela BART;

15: método utilizado para obter o valor setado no atributo (4) durante o processo de indexação/desindexação. Utilizado pela BART;

17, 18, 19: métodos utilizados pela BART durante o processo de indexação/desindexação. Implementado pela aplicação;

Como pôde ser visto na seção 3.4.4. nos exemplos de indexação e desindexação, o *parser* é um dos parâmetros para os respectivos métodos que realizam estas tarefas. Sendo assim fica claro que a aplicação deve instanciar um objeto deste tipo para poder fornecê-los como parâmetro.

Observando a definição da classe `C_BART_PARSER` acima, podemos verificar a existência de três métodos declarados com a sintaxe *virtual nome_do_metodo() = 0*. Esta declaração implica que não é permitido instanciar diretamente objetos desta classe. Portanto, para que seja possível instanciar objetos da classe `C_BART_PARSER` é necessário primeiro criar uma classe qualquer derivada de `C_BART_PARSER`, implementar

os métodos definidos como virtuais e então instanciar objetos a partir da definição da classe derivada.

Antes de definir uma classe derivada de C_BART_PARSER é preciso entender o algoritmo interno de processamento do parser, utilizado pela BART nos processos de indexação e desindexação, para que a aplicação efetue uma implementação correta dos métodos declarados como virtuais. O esquema abaixo demonstra este algoritmo comentando as ações tomadas quando das chamadas dos métodos virtuais do parser. Cabe salientar que este algoritmo é processado uma única vez para cada grupo que for indexado ou desindexado.

```

1. (...)
2.   if( Parser->BART_InicializarParser() != 0 ){           // Permite ao parser realizar algumas
3.                                                         // ações antes de iniciar o processo
4.                                                         // de indexação ou desindexação
5.           (...)
6.           return( NULL );
7.
8.   }
9.   inicializa o contador de parágrafo com valor 1
10.  inicializa o contador de frases com valor 1
11.  inicializa o contador de palavras com valor 0
12.  while( Parser->BART_ProcessarParser() == 0 ){         // Chama o método de processamento do
13.                                                         // parser para que o mesmo se habilite
14.                                                         // a fornecer um termo para o processo
15.                                                         // de indexação ou desindexação
16.
17.     switch( Parser->BART_ObterEstadoDoParser() ){       // Verifica o estado retornado
18.                                                         // no processamento
19.     case FINAL_DE_PALAVRA: // O processamento encontrou um termo
20.         o contador de palavras é adicionado em 1
21.         if( (Termo = Parser->BART_ObterTermoDoParser()) == NULL ){ // Obtém o
22.                                                         // termo encontrado pelo parser no último processamento
23.             (...)
24.             return( NULL );
25.         }
26.         indexa ou desindexa o termo obtido
27.         break;
28.     case FINAL_DE_FRASE: // O processamento encontrou um caracter separador de frase
29.         inicializa o contador de palavras com valor 0
30.         o contador de frases é adicionado em 1
31.         break;
32.     case FINAL_DE_PARAGRAFO: // O processamento encontrou um caracter separador de
33.                             // parágrafo
34.         inicializa o contador de frases com valor 1
35.         inicializa o contador de palavras com valor 0
36.         o contador de parágrafos é adicionado em 1
37.         break;
38.     case FINAL_DE_PROCESSAMENTO: // O processamento encontrou um caracter indicando o
39.                                 // final do processamento
40.
41.         if( Parser->BART_FinalizarParser() != 0 ){ // Permite ao parser realizar
42.                                                         // algumas ações antes de encerrar o
43.                                                         // processo de indexação/desindexação.
44.
45.             (...)
46.             return( NULL );

```

```

47.         }
48.         (... )
49.     )
50. }

```

A representação (...) indica algum trecho de código não importante para o objetivo deste esquema.

Podemos observar que ambos os métodos `BART_InicializarParser()` e `BART_FinalizarParser()` são chamados uma única vez e o método `BART_ProcessarParser()` é chamado tantas vezes quantas forem necessárias até encontrar o final do grupo que está sendo processado.

O algoritmo acima apresenta alguns valores predefinidos para o retorno do método `BART_ObterEstadoDoParser()`. Estes valores devem ser setados utilizando o método `BART_SetarEstadoDoParser()` no código do método `BART_ProcessarParser()` implementado pela aplicação. Em paralelo a esta ação deve também ser setado, o termo encontrado pelo algoritmo de processamento do parser, que também é implementado pela aplicação. Para isso deve ser usado o método `BART_SetarTermoDoParser()`.

No que se refere à implementação dos métodos virtuais por parte da aplicação, algumas considerações devem ser tomadas.

A implementação do método `BART_InicializarParser()` deve ser utilizada como mecanismo de preparação do objeto antes do início do seu processamento. Neste método a aplicação pode fazer os ajustes e configurações dos dados que julgar necessários antes de realmente o objeto ser processado. Um exemplo disso poderia ser a abertura de um arquivo a ser indexado ou o preenchimento de uma estrutura em memória que contém os dados a serem indexados.

Independentemente da origem e da estrutura dos dados que a aplicação indexará, o

método `BART_ProcessarParser()` deverá, a cada chamada, pegar um *token*¹¹ desta estrutura, analisá-lo e setar os valores correspondentes ao termo e ao estado do processamento. Sempre que for encontrado um separador¹² (frase ou parágrafo) ou uma indicação de final de processamento deverá ser setado o termo encontrado e o estado de `FINAL_DE_PALAVRA` e somente na chamada seguinte ao método de processamento é que deverá ser retornado o separador encontrado ou a indicação de final de processamento. Isso é necessário por que, se observarmos o algoritmo acima, quando o estado retorna `FINAL_DE_FRASE`, `FINAL_DE_PARAGRAFO` ou `FINAL_DE_PROCESSAMENTO` nenhum termo é indexado. Portanto ao identificar um destes valores o método `BART_ProcessarParser()` deve primeiramente retornar o termo e a indicação de `FINAL_DE_PALAVRA` para que o termo possa efetivamente ser indexado ou desindexado.

4.3.2. Indexação e Desindexação de Grupos

Na indexação de informações um ponto bastante importante que deve ser observado está relacionado com os parâmetros relativos às referências dos termos que serão indexados. Como é a aplicação quem gerencia o armazenamento de seus dados é necessário que ela informe à BART a referência dos termos que está indexando. Assim, dada uma referência, todos os termos identificados pelo parser serão indexados sob esta referência. Contudo, a identificação dos valores de localização de cada termo (parágrafo, frase e seqüência) é tarefa do parser.

¹¹ *Token* representa uma parte significativa dentro de um conjunto de informações. Cada palavra deste texto pode ser considerada com um *token* do texto.

¹² Os separadores de palavras, frases e parágrafos, assim como a indicação de final de processamento são definidos pelo programador.

Uma vez chamado o método de indexação o procedimento interno da BART será, primeiramente verificar a validade dos parâmetros que representam os índices a serem utilizados e a existência ou não do objeto parser indicado pelo último parâmetro. Feito isso o passo seguinte é iniciar o processamento do parser para obter os termos que deverão ser indexados. O algoritmo deste processo é o mesmo apresentado na seção anterior.

Uma vez que a BART obteve os termos que devem ser indexados, o passo seguinte é realizar a indexação destes termos. Para a realização desta tarefa de forma eficiente a BART monta uma estrutura de indexação intermediária na memória principal para só então efetivar a indexação no sistema de índices e arquivos de dados. A figura 4.4 demonstra esta estrutura intermediária de indexação e a figura 4.5 apresenta o procedimento de efetivação da indexação no sistema de índices e arquivos de dados.

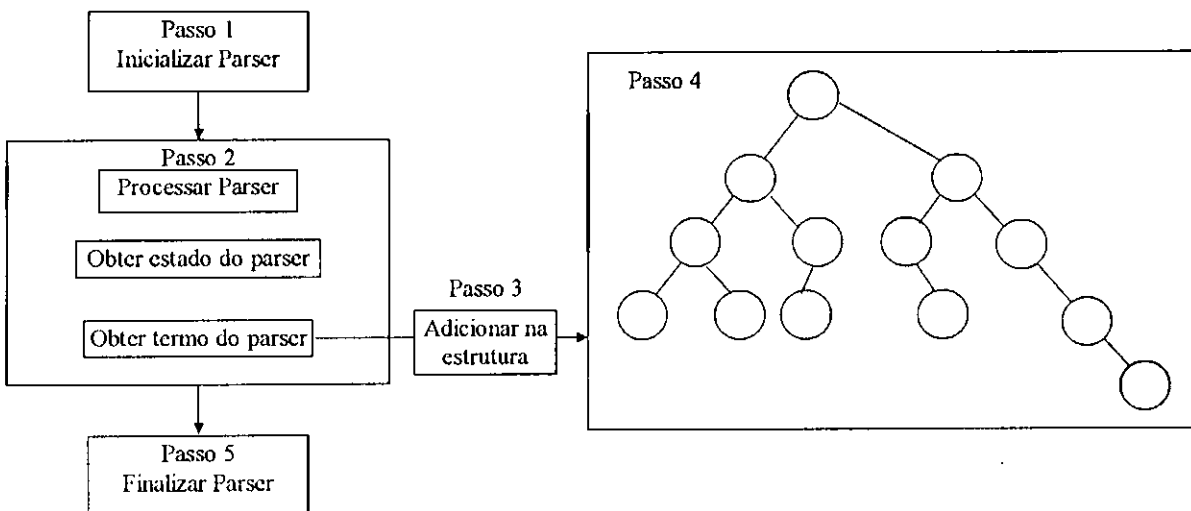


Figura 4.4 - Estrutura intermediária de indexação da BART.

Os passos 1 e 5, como já dito anteriormente, são executados uma única vez durante o processo de indexação. Já os passos 2 e 3 são executados enquanto o estado de processamento do parser for diferente de FINAL_DE_PROCESSAMENTO (como visto na seção anterior). O passo 4 é executado tantas vezes quanto o passo 2 e possui as

características abaixo.

Toda vez que um termo é obtido junto ao parser é criada uma ocorrência deste termo que é formada pelas informações de referência (passadas como parâmetro) e pelas informações de localização (identificadas pelo parser). Este par de informações são passados como parâmetro para um algoritmo interno que realiza a inserção destas informações em uma árvore binária de pesquisa. Cada nodo da árvore binária contém o termo e a lista de ocorrências do referido termo.

O procedimento de inserção na árvore binária da estrutura intermediária é bastante simples e apresenta uma pequena variação nos métodos `BART_IndexarGrupo()` e `BART_IndexarGoWords()`. No método `BART_IndexarGrupo()` o termo será inserido na árvore somente se não estiver definido na lista de stopwords; enquanto no método `BART_IndexarGoWords()` o termo será inserido na árvore apenas se estiver definido na lista de gowords.

1. para cada termo obtido do parser
2. verificar se já existe na árvore
3. se não existe
4. criar uma lista de ocorrência para o termo
5. adicionar a ocorrência do termo na lista
6. se existir
7. adicionar a ocorrência do termo na lista

Quando o processo de indexação receber o estado de que o parser encerrou seu processamento as informações existentes na árvore são descarregadas para o sistema de índices e arquivos de dados, conforme a figura 4.5.

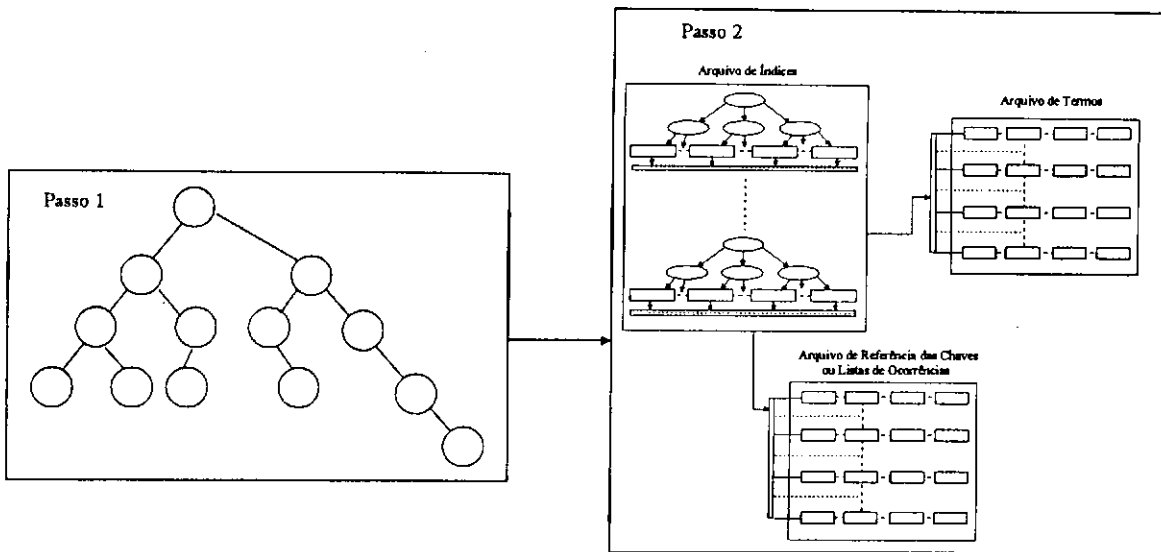


Figura 4.5 - Procedimento de indexação no sistema de índices.

Neste ponto é realizado o seguinte procedimento.

1. para cada nodo da árvore binária
2. pegar o termo do nodo e verificar se ele existe nos índices
3. se não existe
4. alocar uma lista de ocorrências no arquivo de dados
5. obter o endereço desta lista de ocorrências
6. adicionar o termo nos índices, referenciando o endereço obtido
7. gravar a lista de ocorrências do nodo no endereço obtido
8. se existir
9. obter o endereço da lista de ocorrências relacionada
10. adicionar as ocorrências da lista do nodo na lista do arquivo

Para a desindexação de grupos os passos demonstrados nas figuras 4.4 e 4.5 também são válidos. A diferença está apenas no procedimento de transferência da estrutura intermediária para o sistema de índices. Neste caso o procedimento correto é o seguinte.

1. para cada nodo da árvore binária
2. pegar o termo do nodo e verificar se ele existe nos índices
3. se não existe
4. não faz nada
5. se existir
6. obter o endereço da lista de ocorrências relacionada
7. ler o conteúdo da lista de ocorrências do arquivo
8. para cada ocorrência da lista de ocorrências do nodo
9. verificar se ela existe na lista de ocorrências do disco
10. se não existir
11. não faz nada
12. se existir
13. remover a ocorrência da lista de ocorrências do disco

14. se a lista do disco foi alterada
15. gravar a lista novamente
16. se não
17. não faz nada

4.3.4. Pesquisa de Informações

A recuperação dos dados indexados pela BART é uma das principais características da biblioteca. Atendendo ao requisito 2.2.1. (flexibilidade) a BART oferece recursos simples de pesquisa e recursos de pesquisas mais complexos que proporcionam pesquisas mais específicas e restritas a determinadas condições (ver exemplos na seção 3.2.5. do capítulo anterior).

Para utilização dos recursos simples de pesquisa a BART oferece um mecanismo de busca termo-a-termo usando recursos de acesso direto a um índice específico e recursos para obtenção de listas de ocorrências dos termos desejados. Já para a utilização de recursos avançados de pesquisa é necessário o conhecimento da sintaxe da linguagem de consulta oferecida pela BART.

O objetivo principal das seções seguintes é apresentar a sintaxe da linguagem de consulta no formato BNF e descrever a função de cada operador e símbolo utilizados nesta linguagem.

4.3.4.1. Linguagem de Consulta

Segundo [Falout85] as linguagens de consulta para textos podem ser divididas em duas categorias. A primeira é influenciada pelos sistemas gerenciadores de banco de dados e é baseada na álgebra *Booleana*. Em consultas deste tipo um documento específico está ou

não está qualificado para fazer parte da resposta. A segunda categoria é baseada em consultas difusas (*fuzzy queries*), ou seja, dada uma consulta um documento pode estar qualificado para fazer parte da resposta de acordo com o grau de relevância definido para a consulta.

A sintaxe da linguagem de consulta da BART¹³ é apresentada no esquema abaixo em formato BNF. Esta linguagem é totalmente implementada sob o modelo *Booleano*, e não oferece nenhum recurso específico para consultas difusas. Exemplos de utilização da linguagem podem ser vistos na seção 3.2.5.

```

<expressão_pesquisa>:=      <expressão>
                             | NIL
<expressão>                :=  <item>
                             | <expressão><operador><expressão>
                             | (<expressão><restrição>
<item>                     :=  <lista_qualificadores><termo><restrição>
<lista_qualificadores>    :=  <qualificador>
                             | <qualificador><lista_qualificadores>
<qualificador>           :=  ^
                             | #
                             | ~
                             | NIL
<restrição>                :=  [ <expr_restrição> ]
                             | NIL
<expr_restrição>          :=  <item_restrição>
                             | <expr_restrição><operador_restrição><expr_restrição>
<item_restrição>          :=  <localizador><operador_relacional><constante_positiva>
<operador_restrição>     :=  E
                             | OU
<localizador>             :=  CONJUNTO
                             | GRUPO
                             | SUBGRUPO
                             | PARAGRAFO
                             | FRASE
                             | SEQUENCIA
<operador_relacional>    :=  <

```

¹³ Embora a BNF defina a linguagem de consulta em português, a sintaxe dos operadores pode ser configurada conforme a aplicação julgar pertinente. A configuração dos operadores é detalhada na seção A.8.1. na definição do método BART_ConfigurarOperadores().

	>
	<=
	>=
	=
	!=
<constante_positiva>	:= constante numérico maior que 0 (zero)
<termo>	:= "<cadeia>"
	'<intervalo>'
<endereço_lista>	:= @<endereço_lista>
	constante numérica maior que 0 (zero) representando o endereço de uma lista de ocorrências
<cadeia>	:= expressão regular com tratamento de *, ?, []
<intervalo>	:= <operador_relacional><valor>
	<valor>
<valor>	:= <data>
	<hora>
	<constante_real>
<data>	:= <ano>/<mês>/<dia>
<ano>	:= constante numérica de 0 a 9999
<mês>	:= constante numérica de 1 a 12
<dia>	:= constante numérica de 1 a 31
<hora>	:= <HH:MM>
<HH>	:= constante numérica de 0 a 23
<MM>	:= constante numérica de 0 a 59
<constante_real>	:= constante numérica com dígitos decimais
<operador>	:= E
	OU
	XOU
	NÃO
	ADJ<constante_positiva>
	PROX<constante_positiva>
	NOGRUPO
	NOSUBGRUPO
	NOPARAGRAFO
	NASENTERÇA

A linguagem permite que se realizem pesquisas por qualquer palavra ou seqüências de palavras localizadas em qualquer parte dos documentos ou registros da aplicação. A linguagem permite também que sejam utilizados operadores (ver próxima seção) lógicos, de proximidade e de restrição entre as palavras da consulta.

4.3.4.1.1. Operadores da Linguagem

A linguagem de consulta da BART apresenta alguns elementos particulares que representam os operadores da linguagem. Estes operadores estão divididos em quatro categorias:

1. Qualificadores: Indicam que alguma operação deve ser feita sobre o termo seguinte, antes que o mesmo seja pesquisado. Os qualificadores válidos são:

^ - Indica que o termo seguinte deve passar pelo processo de normalização antes de ser pesquisado nos índices;

- Indica que o termo seguinte deve ser pesquisado utilizando-se os recursos de pesquisa fonética;

~ - Indica que deve ser pesquisado o termo seguinte e todos os sinônimos do referido termo.

2. Operadores Lógicos: Os operadores lógicos são utilizados para a realização de operações entre listas de ocorrências considerando-se apenas os valores definidos no atributo *conjunto*. Os operadores lógicos da BART são:

E - O operador **E** realiza uma interseção entre as ocorrências de duas listas de ocorrências selecionando apenas as ocorrências que possuírem o mesmo valor para o atributo *conjunto*. Em outras palavras apenas as ocorrências existentes em um mesmo registro serão selecionadas;

OU - O operador **OU** faz uma união de duas listas de ocorrências;

NÃO - O operador **NÃO** realiza a diferença entre duas listas de

ocorrências;

XOU - O operador **XOU** faz uma união disjunta de duas listas de ocorrências considerando apenas as ocorrências que estão em uma lista mas que não estão na outra.

3. Operadores de Proximidade: Os operadores de proximidade são utilizados para oferecer operações de pesquisa compatíveis com os conceitos de proximidade e adjacência apresentados nas seções 3.4.5. e 3.4.6. Os operadores de proximidade são:

ADJ_n - O operador **ADJ_n** realiza a junção de duas listas de ocorrências realizando uma pesquisa por intervalo de proximidade (para termos na mesma frase) considerando importante a ordem dos termos. *n* representa o intervalo de proximidade entre os termos. A definição de pesquisa por intervalo de proximidade é apresentada nas seções 2.4.5. e 2.4.6.

PROX_n - O operador **PROX_n** realiza a junção de duas listas de ocorrências realizando uma pesquisa por intervalo de proximidade (para termos na mesma frase) sem considerar importante a ordem dos termos. *n* representa o intervalo de proximidade entre os termos.

NOGRUPO - O operador **NOGRUPO** realiza a junção de duas listas de ocorrências realizando uma pesquisa por proximidade de localização para termos em um mesmo grupo. A verificação é feita avaliando-se os valores definidos no atributo *grupo*. Este operador não considera importante a ordem dos termos.

NOSUBGRUPO - O operador **NOSUBGRUPO** realiza a junção de

duas listas de ocorrências realizando uma pesquisa por proximidade de localização para termos em um mesmo subgrupo. A verificação é feita avaliando-se os valores definidos no atributo *subgrupo*. Este operador não considera importante a ordem dos termos.

NOPARAGRAFO - O operador **NOPARAGRAFO** realiza a junção de duas listas de ocorrências realizando uma pesquisa por proximidade de localização para termos ocorrentes em um mesmo parágrafo. A verificação é feita avaliando-se os valores definidos no atributo *parágrafo*. Este operador não considera importante a ordem dos termos.

NAFRASE - O operador **NAFRASE** realiza a junção de duas listas de ocorrências realizando uma pesquisa por proximidade de localização para termos ocorrentes em uma mesma frase. A verificação é feita avaliando-se os valores definidos no atributo *frase*. Este operador também não considera importante a ordem dos termos.

4. Localizadores: Os localizadores são usados para restringir o resultado de uma pesquisa ou uma lista de ocorrências para referências ou localizações de termos específicas. Assim é possível obter apenas as ocorrências relevantes para um determinado *conjunto* ou *grupo*, por exemplo. Os localizadores são:

CONJUNTO - Faz a restrição pelo número do *conjunto*;

GRUPO - Faz a restrição pelo número do *grupo*;

SUBGRUPO - Faz a restrição pelo número do *subgrupo*;

PARAGRAFO - Faz a restrição pelo número do *parágrafo*;

FRASE - Faz a restrição pelo número da *frase*;

SEQUENCIA - Faz a restrição pela posição do termo em relação à frase.

4.3.4.2. Estrutura de Retorno

Ao contrário da pesquisa simples que retorna apenas a lista de ocorrências do termo pesquisado, a pesquisa composta retorna uma estrutura um pouco mais complexa e não apenas a lista de ocorrências resultante da pesquisa. Esta estrutura é representada por objetos da classe `C_BART_LISTA_DE_EXPRESSOES`, que é descrita a seguir.

Ao utilizarmos o recursos de pesquisa composta da BART, internamente os algoritmos de pesquisa geram diversos resultados intermediários até obter o resultado final compatível com a expressão pesquisada. Cada um destes resultados intermediários é um objeto da classe `C_BART_EXPRESSAO`. Estes objetos são adicionados em `C_BART_LISTA_DE_EXPRESSOES` que, como o próprio nome sugere, implementa uma estrutura de lista encadeada contendo os resultados de pesquisa intermediários na ordem em que foram gerados.

Concluído o processamento de pesquisa a BART retorna então, um apontador para o objeto da classe `C_BART_LISTA_DE_EXPRESSOES` que contém todos os resultados intermediários e o resultado final da pesquisa no seu último elemento da lista. A figura 4.6 demonstra a estrutura interna de ambas as classes `C_BART_LISTA_DE_EXPRESSOES` e `C_BART_EXPRESSAO`.

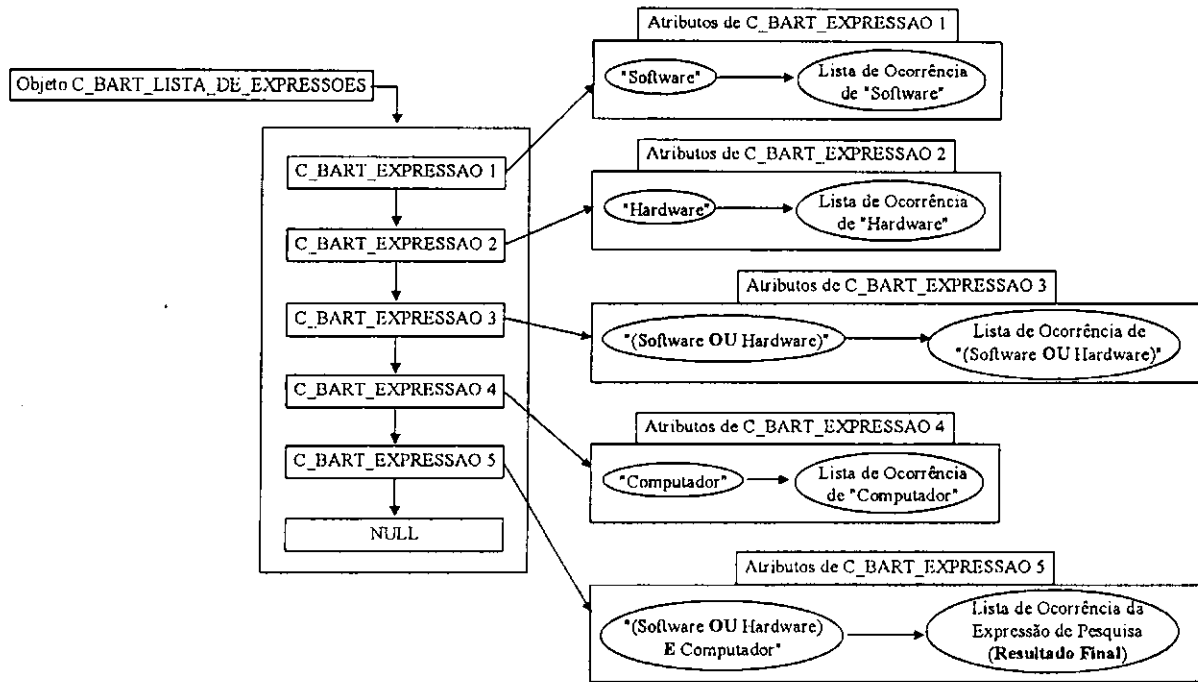


Figura 4.6 - Estrutura de retorno de uma pesquisa composta, considerando a seguinte expressão de pesquisa: "(Software OU Hardware) E Computador".

Utilizando-se os métodos de C_BART_LISTA_DE_EXPRESSOES é possível obter qualquer um dos resultados de pesquisa, seja um resultado intermediário ou o resultado final. O retorno destes métodos é um dos objetos C_BART_EXPRESSAO que originaram a lista de expressões.

Através da figura podemos observar que a classe C_BART_EXPRESSAO possui dois atributos principais: uma expressão (ou parte dela) e uma lista de ocorrências resultante desta expressão. Os métodos de C_BART_EXPRESSAO permitem que se obtenha tanto a expressão quanto a lista de ocorrências. A obtenção da lista de ocorrências dos objetos C_BART_EXPRESSAO é o ponto mais importante do tratamento do retorno de uma pesquisa composta. É a lista de ocorrências que contém as informações (ocorrências) referentes às referências e localizações de todos os termos da expressão que a originou. Portanto, é através deste objeto que a aplicação saberá em quais dos seus

registros, campos e repetições as palavras pesquisadas estão armazenadas.

4.4. Avaliação de Desempenho

Nesta seção apresentamos algumas considerações de desempenho de indexação e pesquisa da BART. Para analisar os dados de desempenho da biblioteca optamos por realizar uma análise comparativa destes dados com dados de desempenho de outros aplicativos. Para tanto, identificamos dois produtos (comercialmente conhecidos) que oferecem recursos textuais. Os aplicativos utilizados foram o LightBase e o LightBase Server (LBS), ambos de propriedade da Light-Infocon e gentilmente cedidos para a realização dos referidos testes.

Posteriormente definimos uma aplicação extremamente simples (usuária dos recursos da BART, que chamamos de DESEMP) que permite realizar operações de indexação e pesquisa. Para evitar que o aplicativo de testes da BART gerasse *overhead* de armazenamento de dados, optamos por definir uma aplicação que utilizasse unicamente o recursos de indexação e pesquisa da BART, evitando qualquer funcionalidade que prejudicasse o desempenho destes recursos. Esta estratégia permite que os dados de desempenho sejam dados precisos referentes unicamente aos recursos da BART e por esta razão o DESEMP não faz armazenamento de dados e permite apenas a indexação de arquivos texto e a recuperação das ocorrências dos termos indexados.

Como ambiente de testes utilizamos um equipamento IBM-PC 486DX 33Mhz com 8Mbytes de RAM rodando Windows95¹⁴. Os arquivos utilizados nos testes de desempenho

¹⁴ O aplicativo LightBase não roda sob Windows95 e sim sob MS-DOS. Portanto os teste de desempenho do LightBase foram realizados rodando no modo MS-DOS do Windows95.

são arquivos texto (TXT) e possuem tamanhos de 10Kbytes, 100Kbytes, 500Kbytes e 1000Kbytes. Estes quatro arquivos foram gerados a partir do conteúdo desta dissertação.

Como medida de desempenho utilizamos apenas o tempo de resposta (TR). O TR é uma medida expressa em segundos que informa o tempo gasto pela aplicação quando a mesma realizar uma determinada tarefa. No caso da indexação, o TR representa o tempo gasto do início da indexação até que todos os dados do arquivo de teste tenham sido indexados. Nas consultas, o TR representa o tempo gasto desde a avaliação da expressão de pesquisa até o retorno dos resultados obtidos.

Em ambos os aplicativos DESEMP e LBS o TR foi medido através de rotinas internas do próprio código do aplicativo apresentando, portanto, valores bastante precisos. No caso do LightBase o TR teve que ser medido manualmente, o que nos deu apenas resultados aproximados. Esta mesma razão também impossibilitou a obtenção do TR de algumas das operações de pesquisa, visto que tais operações são extremamente rápidas. Os dados que não puderam ser obtidos estão indicados com "?".

Os índices utilizados em cada aplicativo estão descritos na tabela 4.3 e os TR's obtidos estão descritos nas tabelas seguintes¹⁵.

APLICATIVOS	ÍNDICES UTILIZADOS
LightBase	Palavra, Data, Valor, Fonema, Backword
LBS	Palavra, Data, Valor, Hora, Fonema, Backword
DESEMP	Palavra, Data, Valor, Hora, Fonema, Backword e Referência

Tabela 4.3 - Índices utilizados pelos aplicativos de testes.

¹⁵ Nas tabelas de desempenho a coluna "Indexação" indica o tempo gasto pelos aplicativos na indexação dos arquivos texto. As demais colunas representam os argumentos utilizados na obtenção dos dados de desempenho das pesquisas. O símbolo # (última coluna) representa pesquisa fonética.

ARQUIVO 10K										
Aplicação	Indexação	Computador	Comp*	*dor	*	1234.00	>1000.00	19/12/1995	>10/12/1995	#Aluísio
DESEMP	2,76	0,10	0,08	0,06	5,02	0,03	0,03	0,04	0,05	0,11
LBS	3,24	0,12	0,09	0,07	5,90	0,04	0,04	0,05	0,06	0,13
LightBase	3,88	?	?	?	6,80	?	?	?	?	?

Tabela 4.4 - Dados de desempenho para arquivo texto de 10k.

ARQUIVO 100K										
Aplicação	Indexação	Computador	Comp*	*dor	*	1234.00	>1000.00	19/12/1995	>10/12/1995	#Aluísio
DESEMP	59,41	0,24	0,36	0,35	53,25	0,05	0,05	0,16	0,24	0,30
LBS	69,89	0,28	0,41	0,42	62,65	0,06	0,06	0,19	0,29	0,35
LightBase	83,86	?	?	?	75,18	?	?	?	?	?

Tabela 4.5 - Dados de desempenho para arquivo texto de 100k.

ARQUIVO 500K										
Aplicação	Indexação	Computador	Comp*	*dor	*	1234.00	>1000.00	19/12/1995	>10/12/1995	#Aluísio
DESEMP	425,39	1,48	2,48	2,36	1039,53	0,42	0,47	1,12	1,52	2,87
LBS	500,46	1,74	2,92	2,78	1222,97	0,50	0,55	1,32	1,79	3,38
LightBase	533,71	2,09	3,51	3,30	1376,60	?	?	1,46	2,02	3,75

Tabela 4.6 - Dados de desempenho para arquivo texto de 500k.

ARQUIVO 1000K										
Aplicação	Indexação	Computador	Comp*	*dor	*	1234.00	>1000.00	19/12/1995	>10/12/1995	#Aluísio
DESEMP	1735,28	3,16	5,28	5,38	2128,51	0,57	0,60	2,01	2,68	7,16
LBS	2041,51	3,72	6,21	6,33	2504,13	0,67	0,70	2,37	3,15	8,43
LightBase	2349,82	4,46	6,71	6,92	2701,9	?	?	2,55	3,66	8,95

Tabela 4.7 - Dados de desempenho para arquivo texto de 1000k.

Para auxiliar na compreensão dos dados de desempenho apresentados é importante que entendamos as características individuais de cada aplicativo utilizado de forma a visualizar melhor a abrangência e limitação de cada um deles. Para isso apresentamos uma breve descrição de cada um dos aplicativos utilizados destacando apenas as funcionalidades

principais e o objetivo do *software*.

O LightBase é um banco de dados textual através do qual o usuário define uma base de dados estruturada em campos, onde estes campos podem ou não serem textualmente indexados. Este aplicativo gerencia não apenas índices mas também gerencia o armazenamento e recuperação dos dados de suas bases bem como a segurança e as permissões de acesso às bases. O LightBase não permite que o mesmo usuário acesse mais de uma base simultaneamente. Os algoritmos e as estruturas de indexação e recuperação de dados deste aplicativo não são conhecidas.

Já o LightBase Server (LBS) é uma interface de programação (API) que oferece recursos para o gerenciamento de bases de dados textual através de uma arquitetura Cliente/Servidor. Através desta API o usuário pode definir bases de dados estruturadas em campos que também podem ou não serem textualmente indexados. Igualmente ao LightBase, esta API oferece recursos para gerenciar não apenas índices mas também o armazenamento e recuperação dos dados de suas bases. Porém, além de gerenciar a segurança e as permissões de acesso às bases, o LBS oferece recursos para controle de acessos a dados distribuídos em rede e permite acesso a múltiplas bases simultaneamente. O LBS utiliza os recursos da BART para gerenciar as operações de indexação e pesquisa de seus dados. Este fator implica que as diferenças entre os dados de desempenho do LBS e da BART indicam o tempo gasto pelo LBS para o gerenciamento e armazenamento de seus dados.

O DESEMP, como já dito, é uma aplicação extremamente simples que foi definida apenas para testar as funcionalidades da BART, uma vez que tal ferramenta está estruturada sob forma de uma biblioteca de classes e métodos. O DESEMP permite apenas que o

usuário selecione os dados que deseja indexar (através de um arquivo texto) e, uma vez realizada a indexação deste arquivo, permite que o usuário teste a linguagem de pesquisa da BART utilizando a sintaxe apresentada na seção 4.3.4.1. O retorno de uma consulta neste aplicativo resulta apenas nas ocorrências das palavras indexadas.

Como observamos através das características de cada um destes aplicativos e analisando os dados de desempenho apresentados para o LBS e o DESEMP, podemos verificar que existe uma sensível diferença dos TR's dos testes realizados com estes aplicativos. Visto que o LBS utiliza os recursos de indexação e pesquisa da BART podemos concluir que esta diferença existe em função do *overhead* do LBS com o tratamento dado ao armazenamento (leitura e gravação) dos dados das bases. Portanto, o aumento nos TR's nos testes de desempenho do LBS não caracterizam a perda de desempenho de execução do código da BART, mas sim indicam que o *overhead* no gerenciamento de dados é pequeno em função do tempo de indexação e pesquisa. Tal afirmação indica que o custo de indexação e pesquisa textual é sensivelmente maior do que o custo de leitura/gravação de dados.

Por outro lado, para verificarmos se as rotinas de indexação e pesquisa da BART possuem um desempenho satisfatório é preciso que comparemos os dados de desempenho de outro aplicativo que não utilize a BART como método de indexação e pesquisa textual. Para este propósito utilizamos o LightBase.

Como as características do LightBase se assemelham muito com o LBS analisamos os dados de desempenho apresentados para estes dois aplicativos. Identificamos que tanto o desempenho de indexação quanto o desempenho de pesquisa do LightBase é inferior ao do LBS. Se considerarmos a afirmação anterior de que o *overhead* de leitura e gravação dos

dados das bases é pequeno e analisarmos os dados das tabelas sem considerar este fator podemos verificar que o desempenho de indexação e pesquisa da BART é consideravelmente melhor do que o desempenho do LightBase.

Sendo o LightBase um produto comercialmente consagrado e tendo o desempenho da BART como superior ao desempenho do LightBase, podemos afirmar que os recursos de indexação e pesquisa da BART apresentam um desempenho minimamente satisfatório pois, apesar desta ferramenta estar perfeitamente apta para ser utilizada no desenvolvimento de outros aplicativos, acreditamos que muito ainda pode ser feito em busca de resultados otimizados para o desempenho dos recursos oferecidos.

5. Conclusão

Atualmente a precisão e a velocidade de localização das informações têm determinado o sucesso dos sistemas de armazenamento de dados. Para maioria dos usuários o ponto mais importante em sistemas desta natureza é a capacidade de obtenção das informações desejadas de forma precisa e principalmente da forma mais rápida e fácil possível.

Sob este ponto de vista, os aspectos mais relevantes no desenvolvimento de aplicações que envolvem armazenamento e recuperação de informações são, sem sombra de dúvidas, a eficiência dos algoritmos e a facilidade de utilização dos recursos. Quanto à eficiência dos algoritmos enfatizamos que a preocupação não gira apenas em torno do desempenho relativo a tempo de execução, mas sim com relação a armazenamento seguro de dados e mecanismos robustos para tratamento de falhas. Em relação à facilidade de utilização dos recursos destacamos os aspectos que se referem à recuperação de informações, pois os mesmos estão relacionados com o tipo de sistema de armazenamento, o tipo de dados que este sistema armazena e, conseqüentemente, o tipo de consulta que ele suporta.

No capítulo 1 vimos que os SGBD são as ferramentas mais conhecidas com tais características e que são, na sua maioria, desenvolvidas sob o modelo relacional e destinadas ao gerenciamento de informações estruturadas. Vimos também que estes

sistemas não atendem na totalidade todos os segmentos e ambientes que utilizam a informação como base de desenvolvimento. Em contra-partida verificamos que sistemas de armazenamento e recuperação de dados com características textuais estão a cada dia ganhando mais destaque dentre os sistemas cujo propósito é o armazenamento de dados.

Com base nestes fatores este trabalho apresentou detalhes sobre o projeto e implementação de uma biblioteca de programação (**BART**) que unifica sistemas de armazenamento de dados estruturados e de dados com características textuais. Além disso esta ferramenta atende aos aspectos mais relevantes no desenvolvimento de aplicações de armazenamento de informações. Aspectos estes que, como visto no início desta seção, são a eficiência dos algoritmos e a facilidade de utilização dos recursos.

Para que os objetivos do trabalho (seção 1.2.) fossem alcançados, o primeiro passo foi a identificação e definição dos requisitos (básicos e específicos) desta ferramenta para possibilitar a definição das características mais importantes da mesma. Com base nisto a implementação da biblioteca pôde ser realizada oferecendo como resultado, uma interface de programação orientada a objetos de fácil agregação a qualquer outro tipo de aplicação.

Esta interface de programação (**API BART**) é a principal contribuição deste trabalho que oferece às aplicações mecanismos de armazenamento de dados estruturados e não-estruturados, atendendo aos principais aspectos citados. Características como orientação a objetos, a referência completa dos termos indexados, mecanismos para utilização de *parsers* para formatos de dados particulares ou formatos padrões bem conhecidos e a altíssima configurabilidade também contribuem para que esta API ofereça vantagens como uma linguagem de consulta mais poderosa e principalmente flexibilidade e adequação de recursos no desenvolvimento de novas aplicações.

Além disso, como contribuição deste trabalho, é importante que alguns de seus aspectos evolutivos também sejam apresentados. Para isso é necessário que antes façamos uma análise do que foi feito, a fim de verificarmos até que ponto os objetivos e requisitos propostos foram concretizados.

Assim, para concluir a realização deste trabalho este capítulo está dividido em duas seções. A primeira seção se destina à avaliação dos objetivos do trabalho e a seção seguinte se destina às conclusões finais e à apresentação de possibilidades evolutivas ou de continuidade do trabalho.

5.1. Avaliação dos Objetivos

Conforme visto na seção 1.2. os objetivos definidos para o trabalho são: (1) definir o que é um sistema de recuperação textual; (2) detalhar conceitos gerais e conceitos particulares relativos ao trabalho; (3) demonstrar uma estrutura básica para um sistema deste porte.

Verificando os objetivos e analisando o que foi apresentado neste trabalho, acreditamos que os dois primeiros objetivos foram satisfatoriamente atingidos. Com relação à realização do terceiro objetivo algumas considerações devem ser feitas.

Para possibilitar a demonstração de uma estrutura básica para um sistema de recuperação textual primeiramente foi preciso identificar alguns requisitos (básicos e específicos) que deveriam ser atendidos pela BART. Dentre estes requisitos estavam relacionados aqueles que deveriam ser seguidos para satisfazer as necessidades das aplicações e aqueles que estariam ligados diretamente com mecanismos de interação entre

programador, API e aplicações.

A identificação e definição destes requisitos (capítulo 2) determinaram os pontos mais importantes a serem considerados tanto na definição dos algoritmos e estruturas de dados que seriam implementados quanto na definição da própria API. Contudo, requisitos como **flexibilidade, robustez e eficiência** não puderam ser atendidos na implementação em função de algumas limitações das ferramentas de desenvolvimento disponíveis e em função de algumas decisões particulares. Com relação aos requisitos básicos esta seção tece comentários relativos àqueles que não puderam ser atendidos completamente.

O requisito *flexibilidade* (seção 2.2.1) cita a necessidade de mecanismos escaláveis de complexidade para atender às evoluções das aplicações. Do ponto de vista funcional e dos recursos oferecidos pela API entendemos que este requisito tenha sido atendido, visto que as aplicações determinam o nível de complexidade na utilização dos recursos da BART. Contudo, do ponto de vista estrutural da classe C_BART_SESSAO, este requisito foi comprometido em função da impossibilidade das aplicações instanciarem, em função de sua necessidade, um ou mais objetos desta classe limitando-se a utilizar apenas um objeto (BART_Sessao) já oferecido globalmente. As razões que originaram tal limitação foram apresentadas no capítulo 4.

Para garantir mecanismos de *robustez* (seção 2.2.5.) diminuindo a ocorrência de falhas foram utilizados mecanismos de flush (esvaziamento ou descargas) nos algoritmos de indexação e armazenamento. Porém, estes mecanismos prejudicaram estes mesmos recursos no que se refere à sua *eficiência* (seção 2.2.4.). Contudo, balanceando vantagens e desvantagens com relação à robustez e desempenho, entendemos que garantir a robustez mesmo perdendo em desempenho (algoritmos mais lentos) é mais importante para as

aplicações. Embora o requisito de robustez tenha previsto mecanismos opcionais para permitir que o programador decida se quer um sistema mais robusto ou um sistema mais rápido os mesmos não foram implementados, ficando desde já como sugestão para trabalhos futuros.

Com relação à definição dos requisitos específicos, acreditamos ter atingido um bom nível de *interação entre a BART e as aplicações* (seção 2.3.1.), visto que a *arquitetura* (seção 2.3.3.1.) apresentada é bastante simples, apesar dos recursos estarem classificados em etapas distintas. Esta simplicidade da arquitetura também contribuiu para que a *interação entre a BART e o programador* (seção 2.3.2.) fosse facilitada no sentido de deixar bastante claro para o programador que os recursos disponíveis são oferecidos em alto nível. Esta característica elimina a preocupação do programador em relação às tarefas de baixo nível. Tal afirmação pode ser verificada através dos exemplos de programação apresentados no capítulo 3 e no apêndice B.

5.2. Conclusões Finais e Trabalhos Futuros

Acreditamos que o desenvolvimento de uma biblioteca de programação para apoiar o desenvolvimento de outras aplicações é uma ferramenta de grande valor, principalmente do ponto de vista da agilidade no desenvolvimento de software. Códigos elaborados e testados para a resolução de problemas específicos facilitam muito o trabalho dos programadores que, na grande maioria dos casos podem acessar os recursos funcionais das bibliotecas preocupando-se apenas com questões de integração.

Entendemos que o trabalho realizado contribuiu para a confirmação destes aspectos sobretudo no sentido que oferece uma biblioteca bastante genérica para apoiar o

desenvolvimento de aplicações que necessitem de recursos de armazenamento e recuperação de informações estruturadas ou não. Contudo, mesmo sendo uma biblioteca genérica, verificamos que alguns recursos que refletem a necessidade atual das aplicações comerciais ou domésticas não foram abordados por este trabalho.

Neste sentido reconhecemos que este trabalho tem espaço para receber inovações e complementos a fim de oferecer recursos que se adequem com necessidades atuais das aplicações. Portanto, para tornar este trabalho ainda mais completo, entendemos que outros trabalhos poderão ser desenvolvidos para abordar novas funcionalidades.

Para novos trabalhos que por ventura venham a ser realizados, sugerimos algumas funcionalidades. São elas:

1. Mecanismos para o programador optar entre algoritmos mais rápidos ou algoritmos mais robustos;
2. Implementação de recursos que utilizem UNICODE como recurso de internacionalização;
3. Oferecimento de recursos de pesquisa baseados em lógica difusa (*fuzzy query*);
4. Especificar e implementar objetos da classe C_BART_PARSER compatíveis com os editores de textos e formatos de dados bastante conhecidos, como por exemplo MS-Word, WordPerfect, arquivos dbf entre outros, para disponibilizá-los para aplicações que necessitem atender a estes formatos;

5. Implementação de algoritmos de compactação (compressão) de chaves e dados;

6. Implementação de *threads* para ganho de desempenho;

7. Conforme já dito, na implementação da BART utilizamos a árvore B+ (B+Tree) como estrutura de armazenamento de índices. Como evolução deste trabalho sugerimos uma análise comparativa entre a árvore B+ e outras estruturas de indexação de palavras. Estruturas como TRIE[Horow82], PAT Array[Gonn92] ou manipulação de cadeias semi-infinitas[Gonn83] poderiam ser investigadas comparando-se pontos como desempenho e requisitos de memória.

Referências Bibliográficas

- [Baez93] BAEZA-YATES, R., BARBOSA, E. F., ZIVIANI, N.
Efficient Text Searching for Read-Only Optical Disks
Revista Brasileira de Computação, Vol.7, No.1, Dezembro 1993, pág. 3-12.
- [Booch94] BOOCH, GRADY.
Object Oriented Analysis and Design
por Horstmann, Cay S., *Mastering Object Oriented Design in C++*
San Jose State University, John Wiley & Sons, Inc. 95.
- [Falout85] FALOUTSOS, C.
Access Methods for Text
ACM Computing Surveys
Vol. 17, No. 1, March 1985, pág. 49-74.
- [Falout92] FALOUTSOS, C.
Signature Files
William B. Frakes and Ricardo Baeza-Yates Eds
Prentice Hall, New Jersey, 1992.
- [Frak92] FRAKES, W. B., BAEZA-YATES, R.
Information Retrieval - Data Structures & Algorithms
Prentice Hall, Englewood Cliffs, New Jersey, USA, 1992.
- [Gonçal89] GONÇALVES, C. C.
A Árvore PATRICIA como Método de Acesso para Bancos de Dados Não Estruturados
Dissertação de Mestrado - Universidade Federal de Minas Gerais
Belo Horizonte - 1989
- [Gonn83] GONNET, G. H.
Unstructured Data Bases or Very Efficient Text Searching
ACM PODS - Vol 2. pág. 117-124
Atlanta, GA, 1983.

- [Gonn87] GONNET, G. H.
PAT 3.1: An Efficient Text Searching System
User's Manual, Center for the New OED
University of Waterloo, CA, 1987
- [Gonn92] GONNET, G. H., BAEZA-YATES, R., SNIDER, T.
New Indices for Text: PAT Tree and PAT Array
William B. Frakes and Ricardo Baeza-Yates Eds
Prentice Hall, New Jersey, 1992.
- [Harm92] HARMAN, D., FOX, E., BAEZA-YATES, R., LEE, W.
Inverted Files
William B. Frakes and Ricardo Baeza-Yates Eds
Prentice Hall, New Jersey, 1992.
- [Horow82] HOROWITZ, E., SAHNI S.
Fundamentals of Data Structures
Pitman International Text - Pitman Publishing Inc, London, 1982
- [Horstm95] HORSTMANN, CAY S.
Mastering Object Oriented Design in C++
San Jose State University, John Wiley & Sons, Inc. 95.
- [Manb90] MANBER, U., MYERS, G.
Suffix Array: A New Method for On-Line String Searches
ACM-SIAM - Symposium on Discrete Algorithms, pág. 319-327, Jan. 1990
- [Manb92] MANBER, UDI.
Foreword in Information Retrieval - Data Structures & Algorithms
William B. Frakes and Ricardo Baeza-Yates Eds
Prentice Hall, New Jersey, 1992.
- [Morri68] MORRISON, D. R.
**PATRICIA - Practical Algorithm To Retrieval Information Coded In
Alphanumeric**
Journal of the ACM - Vol 15 - N° 4, pág. 514-534, 1968
- [Unic91] **The Unicode Standard - Worldwide Character Encoding**
The Unicode Consortium - Version 1.0, Volume 1
Addison-Wesley Publishing Company, Inc., 1991.
- [Yu94] YU, C., MENG, W.
Progress in Database Search Strategies
IEEE Software, May 1994, pág. 11-19

Bibliografia

- ANIK, P. G., FLYNN, R. A.
Versioning a Full Text Information Retrieval System
Proceeding of the 15th Annual International ACM/SIGIR
Conference on Research and Development in Information Retrieval
Denmark, June 1992, pág. 98-111.
- AOE, JUN-ICHI
Key Search Strategies
IEEE Computer Society Technology Series
IEEE Computer Society Press, 1991
- BAEZA-YATES, R., GONNET, G. H.
A New Approach to Text Searching
Communications of the ACM
Vol. 35, No. 10, October 1992, pág. 74-82.
- BAEZA-YATES, R.
Text Retrieval: Theory and Practice
Information Processing 92
Vol. 1, Elsevier Science Publishers, Holland, 1992, pág. 465-476.
- BAEZA-YATES, R., BARBOSA, E. F., ZIVIANI, N.
Hierarchies of Indices for Text Searching
Proceedings of the First South American Workshop on String Processing
Belo Horizonte, Brazil, September 1993, pág. 25-41.
- BURKE, J. J., RYAN, B.
Gigabytes On-Line
Byte, October 1989, pág. 259-267.
- BURTON, F. W., LEWIS, G. N.
A Robust Variation on Interpolation Search
Information Processing Letters
Vol. 10, No. 4,5, July 1980, pág. 198-201.
- CHEN, P. M., LEE, K., GIBSON, G., KATZ, R. H., PATTERSON, D. A.
RAID: High-Performance, Reliable Secondary Storage
ACM Computing Surveys
Vol. 26, No. 2, June 1994, pág. 145-185.

- CHRISTODOULAKIS, S.
Analysis of Retrieval Performance for Records and Objects Using Optical Disk Technology
ACM Transaction on Database Systems
Vol. 12, No. 2, June 1987, pág. 137-169.
- CONSENS, M. P., MILO, T.
Optimizing Queries on Files
Proceedings of the ACM/SIGMOD '94
Conference on Research and Development in Management of Data
Minneapolis, Minnesota, May 1994, pág. 301-312.
- GLAVITSCH, U., SCHÄUBLE, P., WECHSLER, M.
Metadata for Integrating Speech Documents in a Text Retrieval System
SIGMOD RECORD 1994
Vol. 23, No. 4, December 1994, pág. 57-63
- GONNET, G. H., ROGERS, L. D.
The Interpolation-Sequential Search Algorithm
Information Processing Letters
Vol. 6, No. 4, August 1977, pág. 136-139.
- GONNET, G. H., ROGERS, L. D., GEORGE, J. A.
An Algorithm and Complexity Analysis of Interpolation Search
Acta Informatica, Vol 13, 1980, pág. 39-52.
- GONNET, G. H., BAEZA-YATES, R.
Handbook of Algorithms and Data Structures
Addison-Wesley Publishing Co., Second Edition, 1991.
- HARVEY, D. A.
State of the Media
Byte, November 1990, pág. 275-281.
- KLEIN, S. T., BOOKSTEIN, A., DEERWESTER, S.
Storing Text Retrieval System on CD-ROM: Compression and Encryption Considerations
ACM Transactions on Information Systems
Vol. 7, No. 3, July 1989, pág. 230-245.
- LARSON, P.
A Method for Speeding up Text Retrieval
Proceedings of the ACM/SIGMOD
Conference on Research and Development in Management of Data
San Jose, California, May 1983, pág. 117-123.

- PERL, Y., REINGOLD, E. M.
Understanding the Complexity of Interpolation Search
Information Processing Letters
Vol. 6, No. 6, December 1977, pág. 219-222.
- RUBIN, F.
Experiments in Text File Compression
Communications of the ACM
Vol 19, No. 11, November 1976, pág. 617-623.
- SCHILDT, H.
C Completo e Total - Incluindo C++ & ANSI C
McGrow-Hill - 1990
- TOMASIC, A., GARCIA-MOLINA, H., SHOENS, K.
Incremental Updates of Inverted Lists for Text Document Retrieval
ACM-SIGMOD
Conference on Research and Development in Management of Data
Minneapolis, Minnesota, USA, May 1994, pág. 289-300.
- WU, SUN., MANBER, UDI
Fast Text Searching Allowing Errors
Communications of the ACM
Vol. 35, No. 10, October 1992, pág. 83-91.
- YAN, T. W., GARCIA-MOLINA, H.
Index Structures for Selective Dissemination of Information Under the Boolean Model
ACM Transactions on Database Systems
Vol. 19, No. 2, June 1994, pág. 332-364.
- ZIVIANI, N.
Projeto de Algoritmos em C e Pascal
Editora Pioneira, São Paulo, Brasil, 1993.

A. A Interface de Programação da BART

A interface de programação da BART possui um total de 129 métodos divididos entre 11 classes de objetos visíveis pelo programador. Neste apêndice apresentamos uma pequena descrição do que cada classe representa e os principais métodos de cada uma destas classes. Para cada um dos métodos apresentados é descrita sua funcionalidade, parâmetros de entrada, valores de retorno e sintaxe.

A.1. Classe C_BART_ARQUIVO

A classe C_BART_ARQUIVO descreve objetos para tratamento de arquivo onde o usuário poderá armazenar os objetos de dados que desejar.

A.1.1. Métodos da Classe C_BART_ARQUIVO

TB_INT BART_CriarArquivo(TB_CHAR *Nome, TB_UINT Bloco);

Cria o arquivo *Nome* para o armazenamento de dados. Se *Nome* já existir seu conteúdo será zerado. *Bloco* representa o tamanho de cada bloco do arquivo. Sempre que o arquivo precisar crescer, ele o fará em *Bloco* bytes. Este método retorna 0 se operação bem sucedida ou negativo em caso de erro.

TB_INT BART_LerDados(void *Dados, TB_ULONG Localizacao);

Faz uma leitura de dados no arquivo na posição indicada por *Localizacao*. Os dados lidos são copiados para *Dados* que deve ter espaço suficiente para receber as informações obtidas. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

TB_INT BART_AbrirArquivo(TB_CHAR *Nome);

Abre o arquivo *Nome* para leitura ou armazenamento de dados. Caso *Nome* não exista ou não for encontrado um erro será retornado. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

TB_INT BART_RemoverDados(TB_ULONG Localizacao);

Remove os dados do arquivo referenciados *Localizacao*. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

TB_ULONG BART_ArmazenarDados(void *Dados, TB_UINT Tamanho);

Armazena *Tamanho* bytes de *Dados* no arquivo. Retorna a localização dos dados no arquivo se a operação foi bem sucedida ou 0 em caso de erro.

**TB_INT BART_AtualizarDados(TB_ULONG Localizacao, void *Dados,
TB_UINT Tamanho);**

Atualiza a posição do arquivo indicada por *Localizacao* com *Tamanho* bytes de *Dados*. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

A.2. Classe C_BART_EXPRESSAO

A classe C_BART_EXPRESSAO descreve objetos originados por uma pesquisa composta e contém a expressão (ou parte dela) que originou o objeto e a lista de ocorrências correspondente.

A.2.1. Métodos da Classe C_BART_EXPRESSAO

TB_CHAR *BART_ObterExpressao(void);

Retorna a expressão (cadeia de caracteres) que originou o objeto. Em caso de erro retorna um apontador NULL.

BART_LISTA_DE_OCORRENCIAS *BART_ObterListaDeOcorrencias(void);

Obtém a lista de ocorrências relacionada com a expressão que originou o objeto. Em caso de erro retorna um apontador NULL.

A.3. Classe C_BART_FONEMA

A classe C_BART_FONEMA descreve objetos que contém uma lista, onde cada nodo possui a definição de uma seqüência de caracteres (fonemas) associada a um valor. Este valor representa o valor fonético da seqüência de caracteres que será usado nos recursos de indexação e pesquisa fonética.

A.3.1. Métodos da Classe C_BART_FONEMA

```
TB_INT BART_AdicionarFonema( C_BART_LISTA_DE_TERMOS *Fonemas,  
                             TB_UINT ValorFonetico );
```

Adiciona um grupo de *Fonemas* associados ao mesmo *ValorFonetico*. Retorna 0 se operação bem sucedida ou negativo caso contrário.

```
TB_INT BART_RemoveFonemas( C_BART_LISTA_DE_TERMOS *Fonemas );
```

Remove as definições de fonemas existentes na lista de *Fonemas*. Retorna 0 se operação bem sucedida e negativo caso contrário.

A.4. Classe C_BART_OCORRENCIA

A classe C_BART_OCORRENCIA descreve objetos que representam as ocorrências (referência+localização) dos termos indexados em um sistema de índices.

A.4.1. Métodos da Classe C_BART_OCORRENCIA

```
TB_UINT BART_ObterGrupo( void );
```

Retorna o número do grupo no qual o termo da ocorrência é referenciado.

```
TB_UINT BART_ObterParagrafo( void );
```

Retorna o número do parágrafo no qual o termo da ocorrência está localizado.

TB_UINT BART_ObterFrase(void);

Retorna o número da frase no qual o termo da ocorrência está localizado.

TB_UINT BART_ObterSequencia(void);

Retorna a posição da frase na qual o termo da ocorrência se encontra.

TB_ULONG BART_ObterConjunto(void);

Retorna o número do conjunto no qual o termo da ocorrência é referenciado.

TB_UINT BART_ObterSubGrupo(void);

Retorna o número da repetição (multivaloração) do campo no qual o termo da ocorrência é referenciado.

TB_CHAR *BART_ObterTermo(void);

Retorna o termo da ocorrência.

A.5. Classe C_BART_LISTA_DE_EXPRESSOES

A classe C_BART_LISTA_DE_EXPRESSOES descreve objetos resultantes de pesquisas compostas contendo todos os resultados (intermediários e final) gerados durante a avaliação da expressão de pesquisa. Cada um destes resultados é representado por um objeto da classe C_BART_EXPRESSAO.

A.5.1. Métodos da Classe C_BART_LISTA_DE_EXPRESSOES

C_BART_EXPRESSAO *BART_ObterPrimeiraExpressao(void);

Retorna o primeiro objeto C_BART_EXPRESSAO dentre os objetos de compõe lista de objetos resultantes da pesquisa. Em caso de erro retorna um apontador NULL.

C_BART_EXPRESSAO *BART_ObterUltimaExpressao(void);

Retorna o último objeto C_BART_EXPRESSAO dentre os objetos de compõe lista de objetos resultantes da pesquisa. Em caso de erro retorna um apontador NULL.

C_BART_EXPRESSAO *BART_ObterProximaExpressao(void);

Retorna o próximo objeto C_BART_EXPRESSAO da lista de objetos resultantes da pesquisa. Em caso de erro retorna um apontador NULL.

C_BART_EXPRESSAO *BART_ObterExpressaoAnterior(void);

Retorna o objeto C_BART_EXPRESSAO anterior da lista de objetos resultantes da pesquisa. Em caso de erro retorna um apontador NULL.

C_BART_EXPRESSAO *BART_ObterExpressaoResultante(void);

Retorna o objeto C_BART_EXPRESSAO que contém o resultado final da pesquisa. Em caso de erro retorna um apontador NULL.

```
TB_INT BART_CarregarListaDeExpressoes( TB_CHAR *Nome );  
TB_INT BART_CarregarListaDeExpressoes( FILE *Arquivo );
```

Carrega de um arquivo em disco o resultado de uma pesquisa armazenada através de um dos métodos *BART_GravarListaDeExpressoes()*. Se for passado como parâmetro o *Nome* do arquivo, este arquivo será então aberto e a leitura será feita no início do mesmo. Caso o parâmetro seja um apontador para o *Arquivo*, a leitura será realizada na posição corrente deste arquivo. Retorna 0 se operação bem sucedida ou negativo em caso contrário.

```
TB_INT BART_GravarListaDeExpressoes( TB_CHAR *Nome );  
TB_INT BART_GravarListaDeExpressoes( FILE *Arquivo );
```

Realiza a gravação do resultado da pesquisa contido no objeto. Se for passado como parâmetro o *Nome* do arquivo, este arquivo será criado e a gravação será feita no início do mesmo. Caso o parâmetro seja um apontador para o *Arquivo*, a gravação será realizada na posição corrente do mesmo. Retorna 0 se operação bem sucedida ou negativo em caso contrário.

A.6. Classe C_BART_LISTA_DE_OCORRENCIAS

A classe C_BART_LISTA_DE_OCORRENCIAS descreve objetos que contém as listas de ocorrências de um ou mais termos indexados em um sistema de índices.

A.6.1. Métodos da Classe C_BART_LISTA_DE_OCORRENCIAS

C_BART_OCORRENCIA *BART_ObterPrimeiraOcorrencia(void);

Obtém a primeira ocorrência da lista. Em caso de erro retorna um apontador NULL.

C_BART_OCORRENCIA *BART_ObterUltimaOcorrencia(void);

Obtém a última ocorrência da lista. Em caso de erro retorna um apontador NULL.

C_BART_OCORRENCIA *BART_ObterProximaOcorrencia(void);

Obtém a próxima ocorrência da lista. Em caso de erro retorna um apontador NULL.

TB_LONG BART_ObterNumeroDeOcorrencias(void);

Retorna o número de objetos C_BART_OCORRENCIA existentes na lista ou um valor negativo em caso de erro.

C_BART_OCORRENCIA *BART_ObterOcorrencia(TB_LONG *Posicao*);

Obtém a n-ésima (*Posicao*) ocorrência da lista. Em caso de erro retorna um apontador NULL.

C_BART_OCORRENCIA *BART_ObterOcorrenciaAnterior(void);

Obtém a ocorrência anterior da lista. Em caso de erro retorna um apontador NULL.

**TB_INT BART_ADJ(C_BART_LISTA_DE_OCORRENCIAS *Lista1,
C_BART_LISTA_DE_OCORRENCIAS *Lista2,
TB_INT Proximidade);**

Realiza a junção das *Lista1* e *Lista2* considerando a semântica do operador de proximidade **ADJn** (ver seção 4.3.4.1.1.). O resultado é composto das ocorrências adjacentes em até *Proximidade* palavras. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

**TB_INT BART_E(C_BART_LISTA_DE_OCORRENCIAS *Lista1,
C_BART_LISTA_DE_OCORRENCIAS *Lista2);**

Realiza a junção das *Lista1* e *Lista2* considerando a semântica do operador lógico **E** (ver seção 4.3.4.1.1.). Retorna 0 se operação bem sucedida ou negativo em caso de erro.

**TB_INT BART_NOGRUPO(C_BART_LISTA_DE_OCORRENCIAS *Lista1,
C_BART_LISTA_DE_OCORRENCIAS *Lista2);**

Realiza a junção das *Lista1* e *Lista2* considerando a semântica do operador de proximidade **NOGRUPO** (ver seção 4.3.4.1.1.). Retorna 0 se operação bem sucedida ou negativo em caso de erro.

**TB_INT BART_NOPARAGRAFO(C_BART_LISTA_DE_OCORRENCIAS *Lista1,
C_BART_LISTA_DE_OCORRENCIAS *Lista2
);**

Realiza a junção das *Lista1* e *Lista2* considerando a semântica do operador de proximidade **NOPARAGRAFO** (ver seção 4.3.4.1.1.). Retorna 0 se operação bem sucedida ou negativo em caso de erro.

**TB_INT BART_NAFRASE(C_BART_LISTA_DE_OCORRENCIAS *Lista1,
C_BART_LISTA_DE_OCORRENCIAS *Lista2);**

Realiza a junção das *Lista1* e *Lista2* considerando a semântica do operador de proximidade **NAFRASE** (ver seção 4.3.4.1.1.). Retorna 0 se operação bem sucedida ou negativo em caso de erro.

**INT BART_NOSUBGRUPO(C_BART_LISTA_DE_OCORRENCIAS *Lista1,
C_BART_LISTA_DE_OCORRENCIAS *Lista2);**

Realiza a junção das *Lista1* e *Lista2* considerando a semântica do operador de proximidade **NOSUBGRUPO** (ver seção 4.3.4.1.1.). Retorna 0 se operação bem sucedida ou negativo em caso de erro.

**TB_INT BART_PROX(C_BART_LISTA_DE_OCORRENCIAS *Lista1,
C_BART_LISTA_DE_OCORRENCIAS *Lista2,
TB_INT Proximidade);**

Realiza a junção das *Lista1* e *Lista2* considerando a semântica do operador de proximidade **PROXn** (ver seção 4.3.4.1.1.). O resultado é composto das ocorrências adjacentes em até *Proximidade* palavras. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

**TB_INT BART_NAO(C_BART_LISTA_DE_OCORRENCIAS *Lista1,
C_BART_LISTA_DE_OCORRENCIAS *Lista2);**

Realiza a junção das *Lista1* e *Lista2* considerando a semântica do operador lógico **NAO** (ver seção 4.3.4.1.1.). Retorna 0 se operação bem sucedida ou negativo em caso de erro.

```
TB_INT BART_OU( C_BART_LISTA_DE_OCORRENCIAS *Lista1,  
                C_BART_LISTA_DE_OCORRENCIAS *Lista2 );
```

Realiza a junção das *Lista1* e *Lista2* considerando a semântica do operador lógico OU (ver seção 4.3.4.1.1.). Retorna 0 se operação bem sucedida ou negativo em caso de erro.

```
TB_INT BART_XOU( C_BART_LISTA_DE_OCORRENCIAS *Lista1,  
                C_BART_LISTA_DE_OCORRENCIAS *Lista2 );
```

Realiza a junção das *Lista1* e *Lista2* considerando a semântica do operador lógico XOU (ver seção 4.3.4.1.1.). Retorna 0 se operação bem sucedida ou negativo em caso de erro.

```
TB_INT BART_RestringirLista( C_BART_LISTA_DE_OCORRENCIAS *Lista,  
                             TB_CHAR *Restricao );
```

Restringe o conteúdo de *Lista* obedecendo à expressão de *Restricao*. Os *localizadores* de restrição e a sintaxe da expressão podem ser vistos na seção 4.3.4.1.1. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

```
TB_INT BART_RemoverOcorrencias( void );
```

Remove todas as ocorrências que compõe a lista. Retorna 0 se operação bem sucedida ou negativo em caso contrário.

A.7. Classe C_BART_PARSER

A classe C_BART_PARSER descreve objetos que realizam a análise léxica dos grupos que as aplicações indexam em um sistema de índices. Detalhes sobre a estrutura e

funcionamento dos métodos desta classe podem ser vistos na seção 4.3.1.

A.7.1. Métodos da Classe C_BART_PARSER

```
virtual TB_INT C_BART_PARSER::BART_InicializarParser( void );  
virtual TB_INT C_BART_PARSER::BART_ProcessarParser( void );  
virtual TB_INT C_BART_PARSER::BART_FinalizarParser( void );
```

Os três métodos definidos acima são métodos virtuais que devem ser implementados pelo programador. O objetivo principal destes métodos é separar os termos dos grupos que serão indexados.

BART_InicializarParser(): Este método de inicialização é chamado uma única vez. É nele que o programador pode realiza alguma eventual tarefa necessária antes do processamento do parser, como por exemplo abrir um arquivo;

BART_ProcessarParser(): É chamado várias vezes até que todos os termos de um grupo sejam separados e identificados (análise léxica) para permitir a contagem e definição de sua localização.

BART_FinalizarParser(): Este método de finalização do parser é chamado uma única vez. É nele que o programador pode realiza alguma eventual tarefa necessária após o processamento do parser, como por exemplo fechar um arquivo;

```
TB_INT BART_SetarEstadoDoParser( TB_INT Estado );
```

Define o *Estado* ou tipo de informação obtida pelo parser após o processamento. Esta informação pode indicar que o processamento encontrou um separadore de termos, bem como o final de uma frase, parágrafo ou o final do processamento. Este método deve

ser chamado na implementação do método virtual `BART_ProcessarParser()` por parte da aplicação, e é utilizado quando for necessário enviar uma mensagem para os métodos de indexação ou desindexação para que os mesmos realizem a ação correta em função das informações obtidas pelo parser. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

TB_INT BART_SetarTermoDoParser(TB_CHAR *Termo);

Atribui ao objeto parser o *Termo* obtido após uma chamada ao método de processamento do mesmo (`BART_ProcessarParser()`). Este método deve ser chamado sempre que o processamento do Parser encontrar um separador de palavra para que a mesma possa ser indexada ou desindexada. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

A.8. Classe C_BART_SESSAO

A classe `C_BART_SESSAO` descreve o objeto que inicializa as estruturas da biblioteca e gerencia a alocação de recursos para os objetos da classe `C_BART_SISTEMA_DE_INDICES`. Um objeto desta classe já está definido globalmente da seguinte forma:

```
C_BART_SESSAO BART_Sessao;
```

Detalhes sobre a estrutura desta classe podem ser vistos na seção 4.2.1.

A.8.1. Métodos da Classe C_BART_SESSAO

```
TB_INT BART_FecharSistemaDeIndices( C_BART_SISTEMA_DE_INDICES *SI );
```

Remove o sistema de índices apontado por *SI*, fechando os arquivos de controle e liberando os recursos alocados a este objeto. Esta operação desabilita qualquer operação sobre o objeto *SI*. Retorna 0 se operação bem sucedida ou negativo em caso contrário.

```
C_BART_SISTEMA_DE_INDICES *BART_CriarSistemaDeIndices(
    TB_CHAR*Nome,
    TB_INT TamanhoMaximoDaChave,
    TB_INT Indices,
    TB_INT ParteInteira,
    TB_INT ParteDecimal );
```

Instancia um sistema de índices, alocando recursos, criando e inicializando fisicamente os arquivos de controle que irão conter os índices e as localizações das suas chaves. O objeto criado já estará aberto e disponível para utilização. Os arquivos criados terão os seus nomes formados pela junção de *Nome* as extensões .AD (dados), .AC (configuração), .AI (índices) e .AT (termos). Os termos indexados no sistema de índices deverão ter no máximo *TamanhoMaximoDaChave* bytes. *Indices* define os índices que serão criados neste sistema de índices (ver tipos possíveis na seção 2.4.2.). Os dois últimos parâmetros definem, respectivamente, a *ParteInteira* e a *ParteDecimal* usados para representação e indexação de valores numéricos. Caso o sistema de índices *Nome* já existir um erro é retornado e as estruturas do sistema de índices existente são mantidas. Retorna um apontador para o objeto da classe C_BART_SISTEMA_DE_INDICES criado ou NULL em caso de erro.

TB_CHAR *BART_ConfigurarOperadores(TB_CHAR *Operadores);

Este método define ou altera mnemônicos para os *Operadores* lógicos, de proximidade ou localizadores usados nas expressões de pesquisa e de restrição. Caso seja passada uma cadeia vazia como parâmetro, nenhuma definição será alterada e serão retornadas as definições correntes. Retorna um apontador para uma cadeia de caracteres, contendo os mnemônicos anteriormente setados ou NULL em caso de erro. Abaixo é apresentado o formato da expressão de definição dos *Operadores* em formato BNF.

```

<mapeamento>           := <expressão_mapeamento>
                        | NIL

<expressão_mapeamento> := <item_mapeamento>
                        | <item_mapeamento> ; <expressão_mapeamento>

<item_mapeamento>     := <operador_mapeamento> = <mnemônico>

<operador_mapeamento> := ADJ
                        | E
                        | GRUPO
                        | NOGRUPO
                        | NOPARAGRAFO
                        | NAFRASE
                        | NOSUBGRUPO
                        | PROX
                        | NAO
                        | OU
                        | PARAGRAFO
                        | FRASE
                        | CONJUNTO
                        | SUBGRUPO
                        | SEQUENCIA
                        | XOU

<mnemônico>           := Cadeia de caracteres

```

TB_INT BART_ObterErro(void);

Obtém o estado da última operação realizada por qualquer método de qualquer objeto da BART. Retorna o número do estado que é negativo se houveram erros ou 0 caso contrário.

TB_INT BART_ObterPrimeiroErro(void);

Obtém o número do erro que está na primeira posição de um *log* de erros. Todos os demais erros existentes neste *log* são consequências deste primeiro. Retorna o número do erro ou 0 caso não eles não tenham ocorrido.

TB_INT BART_ObterProximoErro(void);

Este método permite fazer uma varredura no *log* de erros obtendo um-a-um os respectivos valores de erro. Este método só poderá ser utilizado após a chamada do método `BART_ObterPrimeiroErro(void)`, caso contrário não apresentará nenhuma indicação de erros, mesmo que eles existam. Retorna o número do próximo erro do *log* e 0 caso se tenha obtido o último erro do *log*.

TB_INT Inicializar_BART(TB_INT NumeroDeSistemasDeIndices);

Inicializa a BART, criando e inicializando suas estruturas internas em função do *NumeroDeSistemasDeIndices* que a aplicação pretende utilizar simultaneamente. Este método deve ser o primeiro método da biblioteca a ser chamado. Retorna 0 se operação bem sucedida ou negativo em caso contrário.

C_BART_SISTEMA_DE_INDICES *BART_AbrirSistemaDeIndices(
TB_CHAR *Nome);

Instancia um sistema de índices já existente, abrindo seus arquivos de controle e alocando os recursos necessário. O *Nome* do sistema de índices não deve possuir extensão e se indicar um sistema de índices inexistente um erro será retornado. Retorna um apontador para o objeto C_BART_SISTEMA_DE_INDICES aberto ou NULL em caso de erro.

TB_CHAR *BART_DescreverErro(TB_INT Erro);

Informa a descrição do *Erro* obtido por um dos métodos BART_ObterErro(), BART_ObterPrimeiroErro() ou BART_ObterProximoErro(). Retorna uma cadeia de caracteres contendo a descrição do *Erro* ou NULL caso o valor de *Erro* indique um valor desconhecido.

A.9. Classe C_BART_SINONIMO

A classe C_BART_SINONIMO descreve objetos que definem termos e um conjunto de sinônimos relativos a cada termo. Estes objetos são utilizados no recursos de pesquisa por sinônimos.

A.9.1. Métodos da Classe C_BART_SINONIMO

TB_INT BART_AdicionarSinonimos(TB_CHAR *Palavra,
C_BART_LISTA_DE_TERMOS *Sinonimos,
TB_INT Tipo);

Define uma lista de *Sinonimos* para *Palavra*. O *Tipo* de definição indica se deverá

ou não ser utilizado bidirecionamento ou associatividade entre as definições dos sinônimos.

Veja abaixo um exemplo para cada caso:

Tipo = NORMAL

```
BART_AdicionarSinonimos( "CASA", "LAR MORADIA", NORMAL );
```

Neste caso LAR e MORADIA serão definidos como sinônimo de CASA, mas CASA não será definida como sinônimo de LAR nem de MORADIA. LAR e MORADIA também não serão sinônimos entre si.

Tipo = BIDIRECIONAL

```
BART_AdicionarSinonimos( "CASA", "LAR MORADIA", BIDIRECIONAL );
```

Neste caso LAR e MORADIA serão definidos como sinônimos de CASA, assim como CASA será definida como sinônimo de LAR e como sinônimo de MORADIA. LAR e MORADIA não são definidos como sinônimos entre si.

Tipo = ASSOCIATIVO

```
BART_AdicionarSinonimos( "CASA", "LAR MORADIA", ASSOCIATIVO );
```

Neste caso LAR e MORADIA serão definidos como sinônimos de CASA, assim como CASA e MORADIA serão definidos como sinônimo de LAR e ainda CASA e LAR serão definidos como sinônimo de MORADIA.

O *Tipo* de definição poderá ser formado pela junção de bits de ASSOCIATIVO e BIDIRECDIONAL, utilizando o operador | (ou) da linguagem C++. Retorna 0 se operação bem sucedida e negativo caso contrário.

TB_INT BART_EliminarSinonimos(void);

Realiza a remoção de todas as definições de sinônimos feita para o objeto. Retorna 0 se operação bem sucedida e negativo caso contrário.

**TB_INT BART_RemoverSinonimos(TB_CHAR *Palavra,
C_BART_LISTA_DE_TERMOS *Sinonimos,
TB_INT Tipo);****TB_INT BART_RemoverSinonimos(TB_CHAR *Palavra);**

Na segunda declaração, o método BART_RemoverSinonimos() removerá todos os termos que foram definidos como sinônimos de *Palavra*.

Já na primeira declaração, o método BART_RemoverSinonimos() removerá apenas os *Sinonimos* que estiverem definido para *Palavra*. O *Tipo* de remoção obedece a mesma semântica do método BART_AdicionarSinonimos(). Retorna 0 se operação bem sucedida e negativo caso contrário.

C_BART_LISTA_DE_TERMOS *BART_ObterSinonimos(TB_CHAR *Palavra);

Retorna um apontador para um objeto da classe C_BART_LISTA_DE_TERMOS contendo todos os sinônimos definidos para *Palavra* ou NULL em caso de erro.

A.10. Classe C_BART_SISTEMA_DE_INDICES

A Classe C_BART_SISTEMA_DE_INDICES descreve objetos que gerenciam os dados das aplicações através de suas estruturas de indexação e recursos de pesquisa.

A.10.1. Métodos da Classe C_BART_SISTEMA_DE_INDICES

TB_INT BART_ChecarSistemaDeIndices(void);

Checa a integridade de um sistema de índices verificando seus índices e demais estruturas de armazenamento. Retorna 0 se o objeto estiver íntegro ou negativo em caso contrário.

TB_INT BART_ZerarSistemaDeIndices(void);

Zera o conteúdo de um sistema de índices reinicializando suas estruturas da mesma forma como foram originalmente criadas. Retorna 0 se operação bem sucedida ou negativo em caso contrário.

TB_INT BART_CorrigirSistemaDeIndices(void);

Corrige inconsistências existentes no sistema de índices. Retorna 0 se operação bem sucedida ou negativo em caso contrário.

TB_INT BART_RemoverChaveCorrente(TB_INT *Indice*);

Remove a chave corrente do índice indicado pelo valor de *Indice*. Retorna 0 se operação bem sucedida ou negativo em caso contrário.

TB_INT BART_ObterChaveCorrente(TB_INT *Indice*, TB_CHAR **Chave*);

Obtém a chave corrente do índice representado pelo valor de *Indice*. Não ocorrendo erros o valor da chave corrente é copiado para *Chave*, que deve possuir espaço suficiente para receber toda a cadeia de caracteres que representa a chave. Retorna 0 se operação bem sucedida, 1 se não existir chave corrente e um valor negativo em caso de erro.

TB_INT BART_ObterPrimeiraChave(TB_INT *Indice*, TB_CHAR **Chave*);

Obtém a primeira chave (ordem alfabética ou numérica crescente) do índice representado pelo valor de *Indice*. Caso não ocorram erros o valor da chave corrente é copiado para *Chave*, que deve possuir espaço suficiente para receber toda a cadeia de caracteres que representa a chave. A chave obtida passa a ser a chave corrente do *Indice*. Retorna 0 se operação bem sucedida, 1 se a chave não foi recuperada e um valor negativo em caso de erro.

**TB_INT BART_ObterChave(TB_INT *Indice*, TB_CHAR **ChavePesquisada*,
TB_CHAR **ChaveRetornada*, TB_INT *TipoPesquisa*);**

Pesquisa a existência da *ChavePesquisada* no índice representado pelo valor de *Indice*. Caso não houverem erros, o valor da chave recuperada será copiado em *ChaveRetornada*, que deve possuir espaço suficiente para receber toda a cadeia de caracteres que representa a chave. A chave obtida passa a ser a chave corrente do *Indice*. O *TipoPesquisa* obedece às seguintes definições.

CHAVE_IGUAL: Pesquisa a existência de uma chave com valor igual à *ChavePesquisada*;

CHAVE_MAIOR: Pesquisa a existência da chave imediatamente posterior (alfabeticamente) ou maior (numericamente) à *ChavePesquisada*;

CHAVE_MAIOR_IGUAL: Pesquisa a existência de uma chave com valor igual à *ChavePesquisada* ou, caso ela não exista, pesquisa a existência da chave imediatamente posterior (alfabeticamente) ou maior (numericamente) à *ChavePesquisada*;

CHAVE_MENOR: Pesquisa a existência da chave imediatamente anterior (alfabeticamente) ou menor (numericamente) à *ChavePesquisada*;

CHAVE_MENOR_IGUAL: Pesquisa a existência de uma chave com valor igual à *ChavePesquisada* ou, caso ela não exista, pesquisa a existência da chave imediatamente anterior (alfabeticamente) ou menor (numericamente) à *ChavePesquisada*;

Retorna 0 se operação bem sucedida, 1 se a chave não foi recuperada e um valor negativo em caso de erro.

TB_INT BART_ObterUltimaChave(TB_INT *Indice*, TB_CHAR **Chave*);

Obtém a última chave (ordem alfabética ou numérica crescente) do índice representado pelo valor de *Indice*. Caso não ocorram erros o valor da chave corrente é copiado para *Chave*, que deve possuir espaço suficiente para receber toda a cadeia de caracteres que representa a chave. A chave obtida passa a ser a chave corrente do *Indice*. Retorna 0 se operação bem sucedida, 1 se a chave não foi recuperada e um valor negativo em caso de erro.

C_BART_LISTA_DE_OCORRENCIAS *BART_ObterLOdaChaveCorrente(TB_INT *Indice*);

Obtém a lista de ocorrências do termo equivalente a chave corrente do *Indice*. Retorna um apontador para um objeto da classe C_BART_LISTA_DE_OCORRENCIAS se operação bem sucedida ou NULL em caso contrário.

TB_INT BART_ObterProximaChave(TB_INT *Indice*, TB_CHAR **Chave*);

Obtém a próxima chave (ordem alfabética ou numérica crescente) do índice representado pelo valor de *Indice*. Caso não ocorram erros o valor da chave corrente é copiado para *Chave*, que deve possuir espaço suficiente para receber toda a cadeia de caracteres que representa a chave. A chave obtida passa a ser a chave corrente do *Indice*. Chamadas sucessivas a este método possibilitam uma varredura sequencial das chaves, no *Indice*, na sua ordem crescente. Retorna 0 se operação bem sucedida, 1 se a chave não foi recuperada e um valor negativo em caso de erro.

TB_LONG BART_ObterNumeroDeChave(TB_INT *Indice*);

Retorna um inteiro longo com o número de chaves que compõem o *Indice* ou um valor negativo em caso de erros.

TB_INT BART_ObterChaveAnterior(TB_INT *Indice*, TB_CHAR **Chave*);

Obtém a chave anterior (ordem alfabética ou numérica crescente) do índice representado pelo valor de *Indice*. Caso não ocorram erros o valor da chave corrente é copiado para *Chave*, que deve possuir espaço suficiente para receber toda a cadeia de caracteres que representa a chave. A chave obtida passa a ser a chave corrente do *Indice*. Chamadas sucessivas a este método possibilitam uma varredura sequencial das chaves, no *Indice*, na sua ordem decrescente. Retorna 0 se operação bem sucedida, 1 se a chave não foi recuperada e um valor negativo em caso de erro.

```
TB_INT BART_IndexarGoWords( TB_INT Indices, TB_ULONG Conjunto,  
                            TB_UINT Grupo, TB_UINT SubGrupo,  
                            C_BART_PARSER *Parser );
```

Este método indexa as gowords existentes em um determinado grupo. Os índices que serão utilizados na indexação são indicados pela composição dos bits de *Indices*. *Conjunto*, *Grupo* e *SubGrupo* indicam a referência dos termos do grupo no qual será verificada a existência de gowords. O *Parser* fará a identificação dos termos do grupo. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

```
TB_INT BART_IndexarGrupo( TB_INT Indices, TB_ULONG Conjunto,  
                          TB_UINT Grupo, TB_UINT SubGrupo,  
                          C_BART_PARSER *Parser );
```

Este método realiza a indexação dos termos existentes em um determinado grupo. Os índices que serão utilizados na indexação são indicados pela composição dos bits de *Indices*. *Conjunto*, *Grupo* e *SubGrupo* indicam a referência dos termos do grupo. O *Parser* fará a identificação dos termos. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

```
C_BART_LISTA_DE_EXPRESSOES *BART_Pesquisar(  
                                          TB_CHAR*ExprDePesquisa );
```

Realiza uma pesquisa composta (ver seção 3.2.5.) de termos, obedecendo os argumentos da *ExprDePesquisa*. Retorna um apontador para um objeto da classe C_BART_LISTA_DE_EXPRESSAO se a operação bem sucedida ou NULL em caso contrário.

```
C_BART_LISTA_DE_TERMOS *BART_ConfigurarGoWords(  
    C_BART_LISTA_DE_TERMOS *GoWords);
```

Este método deve ser utilizado para atribuir uma lista de palavras (*GoWords*) que identificarão as gowords que um sistema de índices utilizará no processo de indexação das mesmas. Retorna um apontador para um objeto da classe C_BART_LISTA_DE_TERMOS que contém as gowords anteriormente atribuídas ou NULL se não existia nenhuma definição anterior ou caso ocorra algum erro.

```
C_BART_LISTA_DE_TERMOS *BART_ConfigurarMascaras(  
    C_BART_LISTA_DE_TERMOS *Mascaras,  
    TB_INT Tipo );
```

Este método deve ser utilizado para atribuir uma lista de termos (*Mascaras*) que identificarão as máscaras que um sistema de índices utilizará para a identificação de datas, horas e valores durante o processo de indexação de termos. O *Tipo* indica se as *Mascaras* estão sendo definidas para o índice da datas, horas ou valores. Retorna um apontador para um objeto da classe C_BART_LISTA_DE_TERMOS que contém as máscaras anteriormente atribuídas ou NULL se não existia nenhuma definição anterior ou caso ocorra algum erro.

```
C_BART_FONEMA *BART_ConfigurarFonemas( C_BART_FONEMAS *Fonema );
```

Este método é utilizado para atribuir a um sistema de índices o objeto *Fonema* que deverá ser utilizado na indexação e pesquisa fonética. Retorna um apontador para o objeto da classe C_BART_FONEMA anteriormente definido ou NULL se não existia nenhuma definição anterior ou caso ocorra algum erro.

**TB_CHAR *BART_ConfigurarTabelaDeNormalizadores(
TB_CHAR *Normalizadores);**

Define um conjunto de caracteres que formarão a tabela de normalizadores do sistema de índices. Retorna os normalizadores anteriormente definidos ou NULL caso não exista nenhuma tabela de normalização setada ou ainda se ocorrer algum erro.

**C_BART_LISTA_DE_TERMOS *BART_ConfigurarStopWords(
C_BART_LISTA_DE_TERMOS *StopWords);**

Este método deve ser utilizado para atribuir uma lista de palavras (*StopWords*) que identificarão as stopwords (termos que não são indexados) que um sistema de índices utilizará no processo de indexação. Retorna um apontador para um objeto da classe C_BART_LISTA_DE_TERMOS que contém as stopwords anteriormente atribuídas ou NULL se não existia nenhuma definição anterior ou caso ocorra algum erro.

**C_BART_SINONIMO *BART_ConfigurarSinonimos(
C_BART_SINONIMO *Sinonimo);**

Este método deverá ser utilizado para setar o objeto *Sinonimo* que será utilizado por um sistema de índices para realizar pesquisas por sinônimos. Retorna um apontador para o objeto da classe C_BART_SINONIMO anteriormente setado ou NULL se não existia nenhuma definição anterior ou caso ocorra algum erro.

**TB_INT BART_Normalizar(TB_CHAR *Origem, TB_CHAR *Destino,
TB_INT Tamanho);**

Normaliza o conteúdo da *Origem* em *Tamanho* bytes, copiando os caracteres normalizados para *Destino*. A normalização é feita conforme a tabela de normalizadores definida no método BART_ConfigurarTabelaDeNormalizadores(). Caso não exista

nenhuma tabela de normalização definida o conteúdo de *Origem* é simplesmente duplicado em *Destino*. Retorna 0 se operação bem sucedida ou negativo em caso contrário.

```
TB_INT BART_DesindexarGrupo( TB_INT Indices, TB_ULONG Conjunto,  
                             TB_UINT Grupo, TB_UINT SubGrupo,  
                             C_BART_PARSER *Parser );
```

Este método realiza a desindexação dos termos existentes em um determinado grupo. Os índices que serão utilizados na desindexação são indicados pela composição dos bits de *Indices*. *Conjunto*, *Grupo* e *SubGrupo* indicam a referência dos termos do grupo. O *Parser* fará a identificação dos termos. Retorna 0 se operação bem sucedida ou negativo em caso de erro.

```
TB_INT BART_DesindexarPorReferencia( TB_INT Indices, TB_ULONG Conjunto,  
                                     TB_UINT Grupo, TB_UINT SubGrupo );
```

Localiza na árvore de referências a chave formada pela referência de *Conjunto*, *Grupo* e *SubGrupo*. Todos os termos relacionados com esta chave serão então desindexados dos *Indices* em que estiverem indexados. Retorna 0 se operação bem sucedida, ou negativo em caso de erro.

A.11. Classe C_BART_LISTA_DE_TERMOS

A classe C_BART_LISTA_DE_TERMOS descreve objetos que são uma lista de termos. Esta lista é, em geral, utilizada como parâmetro de métodos de outras classes.

A.11.1. Métodos da Classe C_BART_LISTA_DE_TERMOS

TB_INT BART_AdicionarTermo(TB_CHAR *Termo);

Adiciona uma cópia de *Termo* à lista de termos do objeto. Esta cópia é adicionada em ordem alfabética. Retorna 0 se operação bem sucedida e negativo caso contrário.

TB_INT BART_ConstruirLista(TB_CHAR *Termos);

Inserir na lista de termos do objeto todos os termos apontados por *Termos*. Os *Termos* deverão estar separados por espaços em branco ou caracteres de tabulação. Cadeias entre aspas são consideradas como um único termo. Retorna 0 se operação bem sucedida ou negativo em caso contrário.

TB_INT BART_RemoverTermos(void);

Remove todos os termos da lista de termos do objeto. Retorna 0 se operação bem sucedida ou negativo caso contrário.

TB_INT BART_RemoverTermo(TB_CHAR *Termo);

Remove o *Termo* da lista de termos do objeto. Retorna 0 se operação bem sucedida e negativo caso contrário.

TB_CHAR *BART_ObterPrimeiroTermo(void);

Retorna o primeiro termo (cadeia de caracteres) da lista de termos do objeto. Retorna NULL se a lista de termos do objeto estiver vazia ou se ocorrer um erro. Neste caso é preciso verificar através dos métodos de tratamento de erro da classe

C_BART_SESSAO para obter a confirmação ou não de erros.

TB_LONG BART_ObterNumeroDeTermos(void);

Retorna um inteiro longo contendo o número de termos existentes na lista de termos do objeto. Em caso de erro um valor negativo é retornado.

TB_CHAR *BART_ObterProximoTermo(void);

Retorna o próximo termo (cadeia de caracteres) da lista de termos do objeto. Retorna NULL se o final da lista for atingida, se a lista de termos estiver vazia ou se ocorrer um erro. Neste caso é preciso verificar através dos métodos de tratamento de erro da classe C_BART_SESSAO para obter a confirmação ou não de erros.

TB_CHAR *BART_PesquisarTermos(TB_CHAR *Termo);

Verifica na lista de termos do objeto a existência ou não de *Termo*. Se encontrado, retorna uma cadeia de caracteres contendo o termo. O método retornará NULL se o termo não estiver na lista ou se houver algum erro. Neste caso é preciso verificar através dos métodos de tratamento de erro da classe C_BART_SESSAO para obter a confirmação ou não de erros.

B. Exemplo de Utilização da BART

Este apêndice é destinado à apresentação de uma aplicação que utiliza recursos de indexação e recuperação textual da BART. O objetivo do código apresentado é **ilustrar a integração da aplicação com a API BART.**

Para tanto apresentamos um aplicativo simples para tratamento de cadastro e consulta de livros, entendendo que tal aplicativo é um exemplo real da necessidade da utilização de recursos textuais. Neste exemplo, todos os seis campos são textualmente indexados permitindo que os registros sejam recuperados a partir das informações de qualquer um destes campos.

Como o objetivo deste exemplo não é apresentar detalhes de como o aplicativo gerencia seus dados, não será dada ênfase aos objetos encarregados desta tarefa. Por outro lado, além do demonstrativo da integração APLICAÇÃO/BART apresentamos também a implementação do *parser* utilizado pelo aplicativo exemplificado. O código do *parser* é bastante acessível e deve ser aproveitado para auxiliar na compreensão de como os métodos virtuais *BART_InicializarParser()*, *BART_ProcessarParser* e *BART_FinalizarParser()* devem ser implementados de acordo com a necessidade de cada aplicação.

B.1. Arquivo de Definições da Aplicação Exemplo

```
#include "bart.h"

const TB_CHAR *SISTEMADEINDICES "bdlivros"
const TB_CHAR *BANCODEDADOS "bdlivros.dat"

const TB_INT CHAVEMAXIMA 50
const TB_INT PARTEINTEIRA 5
const TB_INT PARTEDECIMAL 2

const TB_INT INDICES WORDTREE | DATETREE | VALUETREE
const TB_INT N_SISTEMASDEINDICES 1

const TB_INT TAMANHOTITULO 35
const TB_INT TAMANHOAUTOR 35
const TB_INT TAMANHOEDITORIA 35
const TB_INT TAMANHODATA 10
const TB_INT TAMANHOPRECO PARTEINTEIRA + PARTEDECIMAL
const TB_INT TAMANHOASSUNTO 1024
const TB_INT TAMANHODOREGISTRO TAMANHOTITULO + TAMANHOAUTOR +
TAMANHOEDITORIA + TAMANHODATA +
TAMANHOPRECO + TAMANHOASSUNTO

const TB_INT NUMERODECAMPOS 6

const TB_INT REGISTROVAZIO -4
const TB_INT PESQUISANAOBATE -3
const TB_INT LIVROS_EOF -2
const TB_INT LIVROS_ERRO -1
const TB_INT LIVROS_OK 0

const TB_INT MAX_STRING 1024 + 1
```

B.2. Código Fonte da Aplicação Exemplo

```
#include <windows.h>
#include <stdio.h>
#include <io.h>
#include <string.h>
#include <stdlib.h>
#include <cl3d.h>
#include <bart.h>
#include "bcodados.h"
#include "lparser.h"
#include "dialogo.h"
#include "livros.h"

ATOM NEAR RegistrarClasseLivros( HINSTANCE );
LRESULT CALLBACK LivroProc( HWND, UINT, WPARAM, LPARAM );
LRESULT CALLBACK LivroDlgProc( HWND, UINT, WPARAM, LPARAM );
LRESULT CALLBACK PesquisaDlgProc( HWND, UINT, WPARAM, LPARAM );
DLGPROC lpLivro;
```

```

DLGPROC          lpPesquisa;
static HINSTANCE _hInstance;
static HWND      _hMainWin;
char             _szLivroClass[] = "LivroCLASS";
int              Msg( char *, UINT, BOOL = TRUE );
char            *MsgErro( char *, int );
int              AtribuirConteudoParaRegistro( char *, int );
char            *ObterConteudoDoRegistro( int );
long            GravarRegistro( void );
int             LerRegistro( long );
int             DeletarRegistro( long );
int             IndexarRegistro( long, int );
int             DesindexarRegistro( long, int );
int             LimparRegistro( HWND );
void            MontarExpressao( char *, int );
int             MostrarRegistro( HWND, long );
int             RefazerPesquisa( long );
int             CriarBancoDeDados( void );

C_BART_SISTEMA_DE_INDICES      *pcSistemaDeIndices = NULL;
C_BART_LISTA_DE_OCORRENCIAS    *pcListaDeOcorrencias = NULL;
C_BART_LISTA_DE_EXPRESSOES     *pcListaDeExpressoes = NULL;
C_BART_EXPRESSAO               *pcExpressao = NULL;
C_BART_OCORRENCIA              *pcOcorrencia = NULL;
C_BANCO_DE_DADOS               BancoDeDados;
C_PARSER                       Parser;
char                            szExpr[ MAX_STRING ];

#pragma warn -par
int PASCAL
WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpszCmdLine, int nCmdShow )
{
    MSG msg;
    HWND hWnd;

    _hInstance = hInstance;
    Ctl3dRegister( hInstance );
    Ctl3dAutoSubclass( hInstance );

    if( hPrevInstance == NULL &&
        RegistrarClasseLivros( hInstance ) == NULL ){
        Msg( "Erro no registro da \"windows class\"", MB_OK );
        return( -1 );
    }
    if( BART_Sessao.Inicializar_BART( N_SISTEMASDEINDICES ) != OK ){
        Msg( MsgErro( "BART não pode ser inicializada",
                    BART_Sessao.BART_ObterErro() ), MB_OK );
        return( -1 );
    }
    if( CriarBancoDeDados() != LIVROS_OK ){
        return( -1 );
    }
    if( hWnd = CreateWindow( _szLivroClass, "Cadastro de Livros", WS_DISABLED,
                            CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                            CW_USEDEFAULT, NULL, NULL, hInstance, 0 ) == NULL ){
        Msg( "Não foi possível criar a janela principal da aplicação", MB_OK );
    }
}

```



```

        return( -1 );
    }
    _hMainWin = hWnd;
    ShowWindow( hWnd, SW_HIDE );
    while( GetMessage( &msg, NULL, 0, 0 ) ){
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    if( BART_Sessao.BART_FecharSistemaDeIndices( pcSistemaDeIndices ) != OK ||
        BancoDeDados.Fechar_BD() != LIVROS_OK ){
        Msg( "Erro no fechamento dos arquivos de dados ou no sistema de índices", MB_OK );
        return( -1 );
    }
    Ctl3dUnregister( hInstance );
    return( 0 );
}

int
CriarBancoDeDados( void )
{
    int    iRet = LIVROS_OK;

    if( access( BANCODEDADOS, 00 ) < 0 ){
        if( BancoDeDados.Criar_BD( BANCODEDADOS ) != LIVROS_OK ){
            BancoDeDados.Remover_BD();
            Msg( "Não foi possível criar base de dados para o cadastro de livros", MB_OK );
            iRet = LIVROS_ERRO;
        } else {
            BART_Sessao.BART_RemoverSistemDeIndices( SISTEMADEINDICES );
            if( pcSistemaDeIndices =
                BART_Sessao.BART_CriarSistemaDeIndices( SISTEMADEINDICES,
                CHAVEMAXIMA, INDICES, PARTEINTEIRA,
                PARTEDECIMAL ) == NULL ){
                Msg( "Não foi possível criar sistema de índices", MB_OK );
                BancoDeDados.Remover_BD();
                iRet = LIVROS_ERRO;
            } else if( BART_Sessao.BART_FecharSistemaDeIndices( pcSistemaDeIndices ) /
                != OK ){
                Msg( "Erro no fechamento do sistema de índices", MB_OK );
                BancoDeDados.Remover_BD();
                iRet = LIVROS_ERRO;
            } else if( BancoDeDados.Fechar_BD() != LIVROS_OK ){
                Msg( "Erro no fechamento da base de dados", MB_OK );
                BancoDeDados.Remover_BD();
                iRet = LIVROS_ERRO;
            }
        }
    }
    return( iRet );
}

int
Msg( char *pszMsg, UINT uiStyle, BOOL bError )
{
    return( MessageBox( _hMainWin, pszMsg, bError ? "ERRO" : "MENSAGEM", uiStyle ) );
}

```

```

char *
MsgErro( char *pszMsg, int iErro )
{
    static char    szErro[ MAX_STRING ];

    memset( szErro, 0, MAX_STRING );
    sprintf( szErro, "%s BART erro nº = %d", pszMsg, iErro );
    return( szErro );
}

ATOM NEAR RegistrarClasseLivros( HINSTANCE hInstance )
{
    WNDCLASS    LivroClass;

    LivroClass.style        = CS_HREDRAW | CS_VREDRAW;
    LivroClass.lpfnWndProc  = LivroProc;
    LivroClass.cbClsExtra   = 0;
    LivroClass.cbWndExtra   = 0;
    LivroClass.hInstance   = hInstance;
    LivroClass.hIcon        = LoadIcon( hInstance, MAKEINTRESOURCE( ICONE_LIVROS ) );
    LivroClass.hCursor      = LoadCursor( hInstance, IDC_CROSS );
    LivroClass.hbrBackground = COLOR_WINDOW + 1;
    LivroClass.lpszMenuName = NULL;
    LivroClass.lpszClassName = _szLivroClass;
    return( RegisterClass( &LivroClass ) );
}

LRESULT CALLBACK
LivroProc( HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam )
{
    switch( uMsg ){
    case WM_CREATE:
        if( (pcSistemaDeIndices =
            BART_Sessao.BART_AbrirSistemaDeIndices( SISTEMADEINDICES )) == NULL ){
            Msg( MsgErro( "Erro na abertura do sistema de índices",
                BART_Sessao.BART_ObterErro() ), MB_OK );
            PostQuitMessage( -1 );
            break;
        }
        if( BancoDeDados.Abrir_BD( BANCODEDADOS ) != LIVROS_OK ){
            Msg( "Erro na abertura da base de dados", MB_OK );
            PostQuitMessage( -1 );
            break;
        }
        lpLivro = (DLGPROC)MakeProcInstance( (FARPROC)LivroDlgProc, _hInstance );
        DialogBox( _hInstance, MAKEINTRESOURCE(DIALOGO_LIVRO), hWnd,
            (DLGPROC)lpLivro );
        FreeProcInstance( (FARPROC)lpLivro );
        PostQuitMessage( 0 );
        break;
    default:
        return( DefWindowProc( hWnd, uMsg, wParam, lParam ) );
    }
    return( 0 );
}

```

```

int
AtribuirConteudoParaRegistro( char *pszConteudo, int iCampo )
{
    return( BancoDeDados.AtribuirConteudoParaCampo( iCampo, pszConteudo ) );
}

char *
ObterConteudoDoRegistro( int iCampo )
{
    return( BancoDeDados.ObterConteudoDoCampo( iCampo ) );
}

long
GravarRegistro( void )
{
    return( BancoDeDados.GravarRegistro() );
}

int
LerRegistro( long lRegistro )
{
    return( BancoDeDados.LerRegistro( lRegistro ) );
}

int
DeletarRegistro( long lRegistro )
{
    return( BancoDeDados.DeletarRegistro( lRegistro ) );
}

int
IndexarRegistro( long lRegistro, int iCampo )
{
    char *pszInfo;

    if( (pszInfo = BancoDeDados.ObterConteudoDoCampo( iCampo )) == NULL ||
        Parser.AtribuirCampoParaPARSER( pszInfo ) != LIVROS_OK ||
        pcSistemaDeIndices->BART_IndexarGrupo( INDICES, lRegistro, iCampo, 0,
        &Parser ) != OK ){
        return( LIVROS_ERRO );
    }
    return( LIVROS_OK );
}

int
DesindexarRegistro( long lRegistro, int iCampo )
{
    char *pszInfo;

    if( (pszInfo = BancoDeDados.ObterConteudoDoCampo( iCampo )) == NULL ||
        Parser.AtribuirCampoParaPARSER( pszInfo ) != LIVROS_OK ||
        pcSistemaDeIndices->BART_DesindexarGrupo( INDICES, lRegistro, iCampo, 0,
        &Parser ) != OK ){
        return( LIVROS_ERRO );
    }
    return( LIVROS_OK );
}

```

```

int
LimparRegistro( HWND hWnd )
{
    char    szReg[ MAX_STRING ];

    sprintf( szReg, "Registro: %ld", BancoDeDados.ProximoRegistroLivro() );
    SendDlgItemMessage( hWnd, IDB_NUMREG, WM_SETTEXT, 0, (LPARAM)(LPCSTR)szReg );
    return( BancoDeDados.LimparConteudoDoRegistro() );
}

void
MontarExpressao( char *pszCadeia, int iCampo )
{
    char    szTmp[ MAX_STRING ];
    char    *pszToken;

    memset( szTmp, 0, MAX_STRING );
    if( strlen( pszCadeia ) > 0 ){
        if( strlen( szExpr ) > 0 ){
            strcat( szExpr, " E " );
        }
        while( (pszToken = GetToken( &pszCadeia )) != NULL ){
            sprintf( szTmp, "\"%s\"[GRUPO=%d]", pszToken, iCampo );
            strcat( szExpr, szTmp );
            if( *pszCadeia != '\0' ){
                strcat( szExpr, " E " );
            }
        }
    }
}

int
MostrarRegistro( HWND hWnd, long lRegistro )
{
    char    szReg[ MAX_STRING ];
    char    *pszTxt;
    int     i;

    if( (i = LerRegistro( lRegistro )) != LIVROS_OK ){
        return( i );
    } else for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
        if( (pszTxt = ObterConteudoDoRegistro( i - IDC_TITULO )) == NULL ){
            return( LIVROS_ERRO );
        }
        SendDlgItemMessage( hWnd, i, WM_SETTEXT, 0, (LPARAM)(LPCSTR)pszTxt );
    }
    sprintf( szReg, "Registro: %ld", lRegistro );
    SendDlgItemMessage( hWnd, IDB_NUMREG, WM_SETTEXT, 0, (LPARAM)(LPCSTR)szReg );
    return( LIVROS_OK );
}

```

```

int
RefazerPesquisa( long lPos )
{
    if( pcListaDeExpressoes != NULL ){
        delete pcListaDeExpressoes;
        pcListaDeExpressoes = NULL;
        pcExpressao = NULL;
        pcListaDeOcorrencias = NULL;
        pcOcorrencia = NULL;
    }
    if( (pcListaDeExpressoes = pcSistemaDeIndiccs->BART_Pesquisar( szExpr )) == NULL ){
        return( LIVROS_ERRO );
    }
    if( (pcExpressao = pcListaDeExpressoes->BART_ObterExpressaoResultante()) == NULL ||
        (pcListaDeOcorrencias = pcExpressao->BART_ObterListaDeOcorrencias()) == NULL ){
        return( LIVROS_ERRO );
    }
    if( (pcOcorrencia = pcListaDeOcorrencias->BART_ObterOcorrencia( lPos )) == NULL &&
        (pcOcorrencia = pcListaDeOcorrencias->BART_ObterUltimaOcorrencia()) == NULL ){
        return( PESQUISANAOBATE );
    }
    return( LIVROS_OK );
}

```

LRESULT CALLBACK

LivroDlgProc(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam)

```

{
    PAINTSTRUCT          ps;
    char                 DlgTxt[ TAMANHOASSUNTO ];
    long                 lRegistro;
    int                  i;
    static BOOL          bPesquisando = FALSE;
    C_BART_OCORRENCIA   *pcOcc;

    switch( uMsg ){
    case WM_INITDIALOG:
        for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
            SendDlgItemMessage( hDlg, i, EM_LIMITTEXT,
                iTamanhoDosCampos[ i - IDC_TITULO ], 0 );
        }
        memset( DlgTxt, 0, TAMANHOASSUNTO );
        LimparRegistro( hDlg );
        return( TRUE );
    case WM_PAINT:
        BeginPaint( hDlg, &ps );
        EndPaint( hDlg, &ps );
        return( TRUE );
    case WM_COMMAND:
        if( !bPesquisando ){
            for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
                if( SendDlgItemMessage( hDlg, i, WM_GETTEXTLENGTH,
                    0, 0 ) > 0 ){
                    EnableWindow( GetDlgItem( hDlg, IDB_ADICIONAREG ),
                        TRUE );
                    break;
                }
            }
        }
    }
}

```

```

        if( i > IDC_ASSUNTO ){
            EnableWindow( GetDlgItem( hDlg, IDB_ADICIONAREG ), FALSE );
        }
    } else {
        EnableWindow( GetDlgItem( hDlg, IDB_ADICIONAREG ), FALSE );
    }
    switch( wParam ){
    case IDB_ADICIONAREG:
        if( LimparRegistro( hDlg ) != LIVROS_OK ){
            Msg( "Erro na inicialização do registro", MB_OK );
            EndDialog( hDlg, FALSE );
            return( FALSE );
        }
        for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
            GetDlgItemText( hDlg, i, DlgTxt,
                iTamanhoDosCampos[ i - IDC_TITULO ] );
            if( AtribuirConteudoParaRegistro( DlgTxt, ( i - IDC_TITULO ) ) !=
                LIVROS_OK ){
                Msg( "Erro atribuindo valor a campo", MB_OK );
                EndDialog( hDlg, FALSE );
                return( FALSE );
            }
        }
        if( (lRegistro = GravarRegistro()) == LIVROS_ERRO ){
            DeletarRegistro( lRegistro );
            Msg( "Erro gravando registro", MB_OK );
            EndDialog( hDlg, FALSE );
            return( FALSE );
        }
        for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
            if( IndexarRegistro( lRegistro, ( i - IDC_TITULO ) ) != LIVROS_OK ){
                while( i >= IDC_TITULO ){
                    DesindexarRegistro( lRegistro, ( i - IDC_TITULO ) );
                    --i;
                }
                DeletarRegistro( lRegistro );
                Msg( MsgErro( "Erro na indexação do registro",
                    BART_Sessao.BART_ObterErro() ), MB_OK );
                EndDialog( hDlg, FALSE );
                return( FALSE );
            }
        }
        memset( DlgTxt, 0, TAMANHOTITULO );
        for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
            SendDlgItemMessage( hDlg, i, WM_SETTEXT, 0,
                (LPARAM)(LPCSTR)DlgTxt );
        }
        SetFocus( GetDlgItem( hDlg, IDC_TITULO ) );
        LimparRegistro( hDlg );
        return( TRUE );
    case IDB_ATUALIZA:
        for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
            if( DesindexarRegistro( pcOcorrencia->BART_ObterConjunto(),
                ( i - IDC_TITULO ) ) != LIVROS_OK ){
                Msg( MsgErro( "Erro na desindexação do registro",
                    BART_Sessao.BART_ObterErro() ), MB_OK );
                EndDialog( hDlg, FALSE );
            }
        }
    }
}

```

```

        return( FALSE );
    }
}
if( LimparRegistro( hDlg ) != LIVROS_OK ){
    Msg( "Erro de inicialização do registro", MB_OK );
    EndDialog( hDlg, FALSE );
    return( FALSE );
}
if( DeletarRegistro( pcOcorrencia->BART_ObterConjunto() ) ==
    LIVROS_ERRO ){
    Msg( "Erro na deleção do registro", MB_OK );
    EndDialog( hDlg, FALSE );
    return( FALSE );
}
for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++){
    GetDlgItemText( hDlg, i, DlgTxt,
        iTamanhoDosCampos[ i - IDC_TITULO ] );
    if( AtribuirConteudoParaRegistro( DlgTxt, (i - IDC_TITULO) ) !=
        LIVROS_OK ){
        Msg( "Erro de atribuição de valores no campo", MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    }
}
if( (IRegistro = GravarRegistro()) == LIVROS_ERRO ){
    Msg( "Erro na gravação do registro", MB_OK );
    EndDialog( hDlg, FALSE );
    return( FALSE );
}
for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++){
    if( IndexarRegistro( IRegistro, (i - IDC_TITULO) ) != LIVROS_OK ){
        while( i >= IDC_TITULO ){
            DesindexarRegistro( IRegistro, (i - IDC_TITULO) );
            --i;
        }
        DeletarRegistro( IRegistro );
        Msg( MsgErro( "Erro na indexação do registro",
            BART_Sessao.BART_ObterErro() ), MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    }
}
memset( DlgTxt, 0, TAMANHOTITULO );
for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++){
    SendDlgItemMessage( hDlg, i, WM_SETTEXT, 0,
        (LPARAM)(LPCSTR)DlgTxt );
}
SetFocus( GetDlgItem( hDlg, IDC_TITULO ) );
if( (i = RefazerPesquisa(
    pcListaDeOcorrencias->BART_ObterPosicao(pcOcorrencia))) !=
    LIVROS_OK ){
    if( i == LIVROS_ERRO ){
        Msg( MsgErro( "Erro na atualização da pesquisa",
            BART_Sessao.BART_ObterErro() ), MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    } else if( i == PESQUISANAOBATE ){

```

```

        Msg("Registro resultante da pesquisa foi alterado. Resultado /
            da pesquisa pode não estar correto",
            MB_OK, FALSE );
        return( TRUE );
    }
}
if( MostrarRegistro( hDlg, pcOcorrencia->BART_ObterConjunto() ) !=
    LIVROS_OK ){
    Msg( "Erro na exibição do registro", MB_OK );
    EndDialog( hDlg, FALSE );
    return( FALSE );
}
return( TRUE );
case IDB_DELETAREG:
for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
    if( DesindexarRegistro( pcOcorrencia->BART_ObterConjunto(),
        (i - IDC_TITULO) ) != LIVROS_OK ){
        Msg( MsgErro( "Erro na desindexação do registro",
            BART_Sessao.BART_ObterErro() ), MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    }
}
if( LimparRegistro( hDlg ) != LIVROS_OK ){
    Msg( "Erro na inicialização do registro", MB_OK );
    EndDialog( hDlg, FALSE );
    return( FALSE );
}
if( DeletarRegistro( pcOcorrencia->BART_ObterConjunto() ) ==
    LIVROS_ERRO ){
    Msg( "Erro na deleção do registro", MB_OK );
    EndDialog( hDlg, FALSE );
    return( FALSE );
}
memset( DlgTxt, 0, TAMANHOTITULO );
for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
    SendDlgItemMessage( hDlg, i, WM_SETTEXT, 0,
        (LPARAM)(LPCSTR)DlgTxt );
}
SetFocus( GetDlgItem( hDlg, IDC_TITULO ) );
if( ( i = RefazerPesquisa(
    pcListaDeOcorrencias->BART_ObterPosicao( pcOcorrencia ) ) ) !=
    LIVROS_OK ){
    if( i == LIVROS_ERRO ){
        Msg( MsgErro( "Erro na atualização da pesquisa",
            BART_Sessao.BART_ObterErro() ), MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    } else {
        Msg("Registro resultante da pesquisa foi alterado. Resultado /
            da pesquisa pode não estar correto ",
            MB_OK, FALSE );
        return( TRUE );
    }
}
if( MostrarRegistro( hDlg, pcOcorrencia->BART_ObterConjunto() ) !=
    LIVROS_OK ){

```



```

        Msg( "Erro na exibição do resultado de pesquisa", MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    }
    return( TRUE );
case IDB_REINDEXABASE:
    if( pcListaDeExpressoes != NULL ){
        delete pcListaDeExpressoes;
        pcListaDeExpressoes = NULL;
        pcExpressao = NULL;
        pcListaDeOcorrencias = NULL;
        pcOcorrencia = NULL;
    }
    if( LimparRegistro( hDlg ) != LIVROS_OK ){
        Msg( "Erro na inicialização do registro", MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    }
    lRegistro = 0L;
    while( 1 ){
        i = MostrarRegistro( hDlg, lRegistro );
        if( i == LIVROS_ERRO ){
            Msg( "Erro na leitura do resgitro", MB_OK );
            EndDialog( hDlg, FALSE );
            return( FALSE );
        } else if( i == LIVROS_EOF ){
            break;
        } else if( i == REGISTROVAZIO ){
            lRegistro++;
        }
    }
    for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
        if( DesindexarRegistro( lRegistro, ( i - IDC_TITULO ) ) !=
            LIVROS_OK ){
            Msg( MsgErro( "Erro na reindexação do registro",
                BART_Sessao.BART_ObterErro() ), MB_OK );
            EndDialog( hDlg, FALSE );
            return( FALSE );
        }
        if( IndexarRegistro( lRegistro, ( i - IDC_TITULO ) ) !=
            LIVROS_OK ){
            while( i >= IDC_TITULO ){
                DesindexarRegistro( lRegistro,
                    ( i - IDC_TITULO ) );
                --i;
            }
            DeletarRegistro( lRegistro );
            Msg( MsgErro( "Erro na reindexação do registro",
                BART_Sessao.BART_ObterErro() ), MB_OK );
            EndDialog( hDlg, FALSE );
            return( FALSE );
        }
    }
    lRegistro++;
}
memset( DlgTxt, 0, TAMANHOTITULO );
for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++ ){
    SendDlgItemMessage( hDlg, i, WM_SETTEXT, 0,

```

```

        (LPARAM)(LPCSTR)DlgTxt );
    }
    SetFocus( GetDlgItem( hDlg, IDC_TITULO ) );
    EnableWindow( GetDlgItem( hDlg, IDB_PROXIMORESULTADO ), FALSE );
    EnableWindow( GetDlgItem( hDlg, IDB_RESULTPREVIO ), FALSE );
    EnableWindow( GetDlgItem( hDlg, IDB_ADICIONAREG ), FALSE );
    EnableWindow( GetDlgItem( hDlg, IDB_DELETAREG ), FALSE );
    EnableWindow( GetDlgItem( hDlg, IDB_ATUALIZA ), FALSE );
    if( LimparRegistro( hDlg ) != LIVROS_OK ){
        Msg( "Erro na inicializaçãod o registro", MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    }
    return( TRUE );
case IDB_PESQUISAREG:
    if( pcListaDeExpressoes != NULL ){
        delete pcListaDeExpressoes;
        pcListaDeExpressoes = NULL;
        pcExpressao = NULL;
        pcListaDeOcorrencias = NULL;
        pcOcorrencia = NULL;
    }
    for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++){
        if( SendDlgItemMessage( hDlg, i, WM_GETTEXTLENGTH, 0, 0 ) > 0 ){
            break;
        }
    }
    if( i <= IDC_ASSUNTO ){
        memset( szExpr, 0, MAX_STRING );
        for( ; i <= IDC_ASSUNTO; i++){
            GetDlgItemText( hDlg, i, DlgTxt,
                iTamanhoDosCampos[ i - IDC_TITULO ] );
            MontarExpressao( DlgTxt, ( i - IDC_TITULO ) );
        }
    }
    else {
        lpPesquisa = (DLGPROC)MakeProcInstance(
            (FARPROC)PesquisaDlgProc, _hInstance );
        DialogBox( _hInstance,
            MAKEINTRESOURCE(DIALOGO_PESQUISA),
            hDlg, (DLGPROC)lpPesquisa );
        FreeProcInstance( (FARPROC)lpPesquisa );
    }
    if( strlen( szExpr ) > 0 ){
        if( (pcListaDeExpressoes =
            pcSistemaDeIndices->BART_Pesquisar( szExpr )) == NULL ){
            Msg( MsgErro( "Erro na realização da pesquisa",
                BART_Sessao.BART_ObterErro() ), MB_OK );
            return( TRUE );
        }
        if( (pcExpressao =
            pcListaDeExpressoes->BART_ObterExpressaoResultante()) ==
            NULL || (pcListaDeOcorrencias =
            pcExpressao->BART_ObterListaDeOcorrencias()) == NULL ){
            Msg( MsgErro( "Erro montando resultado da pesquisa",
                BART_Sessao.BART_ObterErro() ), MB_OK );
            EndDialog( hDlg, FALSE );
            return( FALSE );
        }
    }

```

```

    }
    if( pcOcorrencia =
        pcListaDeOcorrencias->BART_ObterPrimeiraOcorrencia() ==
            NULL ){
        Msg( "Resultado vazio", MB_OK, FALSE );
        return( TRUE );
    }
    if( MostrarRegistro( hDlg, pcOcorrencia->BART_ObterConjunto() ) !=
        LIVROS_OK ){
        Msg( "Erro na exibição do resultado da pesquisa", MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    }
    EnableWindow( GetDlgItem( hDlg, IDB_PROXIMORESULTADO ),
        TRUE );
    EnableWindow( GetDlgItem( hDlg, IDB_RESULTPREVIO ), TRUE );
    EnableWindow( GetDlgItem( hDlg, IDB_ATUALIZA ), TRUE );
    EnableWindow( GetDlgItem( hDlg, IDB_DELETAREG ), TRUE );
    bPesquisando = TRUE;
}
SetFocus( GetDlgItem( hDlg, IDC_TITULO ) );
return( TRUE );
case IDB_PROXIMORESULTADO:
    if( (pcOcc = pcListaDeOcorrencias->BART_ObterProximaOcorrencia() ==
        NULL ){
        MessageBeep( -1 );
        return( TRUE );
    } else {
        pcOcorrencia = pcOcc;
        pcOcc = NULL;
    }
    if( MostrarRegistro( hDlg, pcOcorrencia->BART_ObterConjunto() ) !=
        LIVROS_OK ){
        Msg( "Erro na exibição do resultado da pesquisa", MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    }
    return( TRUE );
case IDB_RESULTPREVIO:
    if( (pcOcc = pcListaDeOcorrencias->BART_ObterOcorrenciaAnterior() ==
        NULL ){
        MessageBeep( -1 );
        return( TRUE );
    } else {
        pcOcorrencia = pcOcc;
        pcOcc = NULL;
    }
    if( MostrarRegistro( hDlg, pcOcorrencia->BART_ObterConjunto() ) !=
        LIVROS_OK ){
        Msg( "Erro na exibição do resultado da pesquisa", MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    }
    return( TRUE );
case IDB_LIMPAREG:
    if( pcListaDeExpressoes != NULL ){
        delete pcListaDeExpressoes;
    }

```

```

        pcListaDeExpressoes = NULL;
        pcExpressao = NULL;
        pcListaDeOcorrencias = NULL;
    }
    memset( DlgTxt, 0, TAMANHOASSUNTO );
    memset( szExpr, 0, MAX_STRING );
    for( i = IDC_TITULO; i <= IDC_ASSUNTO; i++){
        SendDlgItemMessage( hDlg, i, WM_SETTEXT, 0,
            (LPARAM)(LPCSTR)DlgTxt);
    }
    SetFocus( GetDlgItem( hDlg, IDC_TITULO ) );
    bPesquisando = FALSE;
    EnableWindow( GetDlgItem( hDlg, IDB_PROXIMORESULTADO ), FALSE );
    EnableWindow( GetDlgItem( hDlg, IDB_RESULTPREVIO ), FALSE );
    EnableWindow( GetDlgItem( hDlg, IDB_ADICIONAREG ), FALSE );
    EnableWindow( GetDlgItem( hDlg, IDB_DELETAREG ), FALSE );
    EnableWindow( GetDlgItem( hDlg, IDB_ATUALIZA ), FALSE );
    if( LimparRegistro( hDlg ) != LIVROS_OK ){
        Msg( "Erro na inicialização do registro", MB_OK );
        EndDialog( hDlg, FALSE );
        return( FALSE );
    }
    return( TRUE );
case IDB_SAIDA:
    SendMessage( hDlg, WM_COMMAND, (WPARAM)IDB_LIMPAREG, 0L );
    EndDialog( hDlg, TRUE );
    return( TRUE );
} //switch
} //switch
return( FALSE );
}

```

LRESULT CALLBACK

```

PesquisaDlgProc( HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam )

```

```

{
    PAINTSTRUCT ps;

    switch( uMsg ){
    case WM_INITDIALOG:
        SendDlgItemMessage( hDlg, ID_EXPRESSAO, EM_LIMITTEXT, MAX_STRING, 0 );
        return( TRUE );
    case WM_PAINT:
        BeginPaint( hDlg, &ps );
        EndPaint( hDlg, &ps );
        return( TRUE );
    case WM_COMMAND:
        if( SendDlgItemMessage( hDlg, ID_EXPRESSAO, WM_GETTEXTLENGTH, 0, 0 ) > 0 ){
            EnableWindow( GetDlgItem( hDlg, ID_PESQUISA ), TRUE );
        } else {
            EnableWindow( GetDlgItem( hDlg, ID_PESQUISA ), FALSE );
        }
        switch( wParam ){
        case ID_PESQUISA:
            memset( szExpr, 0, MAX_STRING );
            GetDlgItemText( hDlg, ID_EXPRESSAO, szExpr, MAX_STRING );
            EndDialog( hDlg, TRUE );
            return( TRUE );
        }
    }
}

```

```

        case ID_FIMPESQUISA:
            EndDialog( hDlg, TRUE );
            return( TRUE );
        } //switch
    } //switch
    return( FALSE );
}

```

B.3. Arquivo de Definição da Classe C_PARSER

```

#ifndef    _PARSER

#define    _PARSER

#include<stdio.h>
#include<string.h>
#include<parser.h>    // Arquivo de definição da classe C_BART_PARSER

const int TAMBUFFER    4096

class C_PARSER : public LTC_PARSER {
private:
    TB_CHAR    tctBuffer[ TAMBUFFER + 1 ];
    TB_CHAR    *ptctBuffer;
    TB_CHAR    tctBufferAux[ 10 ];
    int        iIndBufAux;
    int        iAux;
    int        iFlag;

    TB_CHAR    ObterCaractere();
    TB_CHAR    *Parte( int * );
    void        ReporCaractere( TB_CHAR );
public:
    C_PARSER( void );
    ~C_PARSER();

    TB_INT    BART_InicializarParser();
    TB_INT    BART_ProcessarParser();
    TB_INT    BART_FinalizarParser();
    int        AtribuirCampoParaPARSER( TB_CHAR * );
};

#endif

```

B.4. Código Fonte da Classe C_PARSER

```

#include<stdio.h>
#include<ctype.h>
#include<io.h>
#ifdef BORLAND
#include<alloc.h>
#else
#include<malloc.h>

```

```

#endif
#include <const.h>
#include "parser.h"
#include "livros.h"

const int      MAX_PALAVRA      50

C_PARSER::C_PARSER( void )
{
    memset( tctBuffer, 0, TAMBUFFER );
    ptctBuffer = tctBuffer;
    iIndBufAux = -1;
    iAux = ERR;
    iFlag = ERR;
}

C_PARSER::~C_PARSER()
{
}

TB_INT
C_PARSER::BART_InicializarParser()
{
    if( iAux != OK ){
        return( ERR );
    }
    return( OK );
}

TB_INT
C_PARSER::BART_ProcessarParser()
{
    TB_CHAR    *Buffer;

    if( iAux != OK ){
        return( ERR );
    }
    if( (Buffer = Parte( &iFlag )) == NULL ){
        return( ERR );
    }
    BART_SetarTermoDoParser( Buffer );
    BART_SetarEstadoDoParser( iFlag );
    if( iFlag == FINAL_DE_PROCESSAMENTO ){
        iAux = ERR;
    }
    return( OK );
}

TB_INT
C_PARSER::BART_FinalizarParser()
{
    return( OK );
}

```

```

int
C_PARSER::AtribuirCampoParaPARSER( TB_CHAR *ptctCadeia )
{
    if( strlen( ptctCadeia ) >= TAMBUFFER ){
        return( LIVROS_ERRO );
    }
    strcpy( tctBuffer, ptctCadeia );
    ptctBuffer = tctBuffer;
    iAux = OK;
    return( LIVROS_OK );
}

TB_CHAR *
C_PARSER::Parte( int *iFlag )
{
    TB_CHAR    Caractere;
    TB_CHAR    CaracAux;
    static TB_CHAR tctPalavra[ MAX_PALAVRA ];
    int        iTmp = 0;
    int        iFim = FALSE;

    while( iFim == FALSE ){
        switch( (Caractere = ObterCaractere()) ){
            case '\0':
                if( iTmp == 0 ){
                    *iFlag = FINAL_DE_PROCESSAMENTO;
                } else {
                    ReporCaractere( Caractere );
                }
                iFim = TRUE;
                tctPalavra[ iTmp ] = '\0';
                break;
            case '\r':
            case '\n':
                if( iTmp == 0 ){
                    if( *iFlag == FINAL_DE_PARAGRAFO ){
                        break;
                    }
                    *iFlag = FINAL_DE_PARAGRAFO;
                } else {
                    ReporCaractere( Caractere );
                }
                tctPalavra[ iTmp ] = '\0';
                iFim = TRUE;
                break;
            case ' ':
            case '\t':
            case '!':
                if( iTmp == 0 ){
                    break;
                }
                *iFlag = FINAL_DE_PALAVRA;
                tctPalavra[ iTmp ] = '\0';
                iFim = TRUE;
                break;
            case '!':
            case '?':

```

```

case '!':
case ';':
case ':':
    CaracAux = ObterCaractere();
    ReporCaractere( CaracAux );
    if( Caractere != ':' || (CaracAux != '\0' &&
        !isdigit( CaracAux )) ){
        if( iTmp == 0 ){
            if( *iFlag == FINAL_DE_FRASE ){
                break;
            }
            *iFlag = FINAL_DE_FRASE;
        } else {
            ReporCaractere( Caractere );
        }
        tctPalavra[ iTmp ] = '\0';
        iFim = TRUE;
        break;
    }
default:
    tctPalavra[ iTmp++ ] = Caractere;
    *iFlag = FINAL_DE_PALAVRA;
    break;
} // switch
}
return( tctPalavra );
}

void
C_PARSER::ReporCaractere( TB_CHAR Caractere )
{
    tctBufferAux[ ++iIndBufAux ] = Caractere;
}

TB_CHAR
C_PARSER::ObterCaractere( void )
{
    if( iIndBufAux >= 0 ){
        return( tctBufferAux[ iIndBufAux-- ] );
    }
    if( *ptctBuffer == '\0' ){
        return( '\0' );
    }
    return( *ptctBuffer++ );
}

```