

Algoritmos e Estrutura de Dados II

Trabalho Prático 03 - Árvores de Pesquisa/Hashing

Lucas Pereira Monteiro¹

¹Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)

lucasmonteiro@dcc.ufmg.br

Resumo. *O objetivo deste trabalho é a implementação e organização de um índice remissivo. O índice remissivo consiste em uma lista alfabética de palavras-chave ou palavras relevantes no texto com a indicação dos locais no texto onde cada-palavra chave ocorre. A estrutura para organização e implementação do índice remissivo será a árvore SBB, também conhecida como symmetric binary B-tree¹.*

1. Introdução

A manipulação de árvores é um dos aspectos fundamentais em Ciência da Computação. Para se fazer um melhor uso das linguagens estruturadas², é importante utilizar os Tipos Abstratos de Dados, os quais encapsulam os detalhes de implementação. Além disso, temos que o bom uso das estruturas básicas de programação (Pilha, Lista e Fila) somados ao uso de algoritmos de ordenação resultam em um grande conjunto de ferramentas que possibilita ao programador resolver problemas, desde muito simples até muito complexos.

O objetivo desse trabalho é implementar Tipos Abstratos de Dados (TAD's) para a manipulação de árvores SBB, com a finalidade de ser gerir um índice remissivo.

Ao final do trabalho prático, espera-se praticar os conceitos básicos e necessários de programação utilizados no mesmo. São eles: manipulação de arquivos, manipulação de vetores, alocação dinâmica, operações com pilha, lista e fila e, por fim, manipulação de árvores SBB. Além da implementação e manipulação de TAD's e ponteiros, os quais são extremamente necessários para um bom desenvolvimento em praticamente qualquer trabalho utilizando-se a linguagem C.

1.1. Descrição superficial do problema e da sua resolução

Uma editora possui diversos títulos em seu acervo, porém estão desatualizados. Um dos itens que precisa ser atualizado é o índice remissivo de cada livro. O índice remissivo consiste em uma lista alfabética de palavras-chave ou palavras relevantes do texto com a indicação dos locais no texto onde cada palavra-chave ocorre. A empresa precisará que seja feita a leitura dos índices que ela possui e permita operações de inserção de novos registros e busca por elemento e intervalos. Ela também gostaria de calcular o número de ocorrências de cada registro do índice, dentre outras operações.

O problema proposto pelo TP é:

¹<https://cs.uwaterloo.ca/research/tr/1980/CS-80-51.pdf>

²http://en.wikipedia.org/wiki/Programming_language

- Representação hierárquica dos registros por meio de árvore com balanceamento SBB aninhadas, além da implementação das operações de inserção e busca por um único elemento.
- Implementação da busca por intervalo e um método para obtenção de todas as chaves que aparecem em páginas ímpares.
- Implementação da busca por chaves em um intervalo de páginas e um contador do número de ocorrências de cada um dos registros utilizando técnicas de hashing.

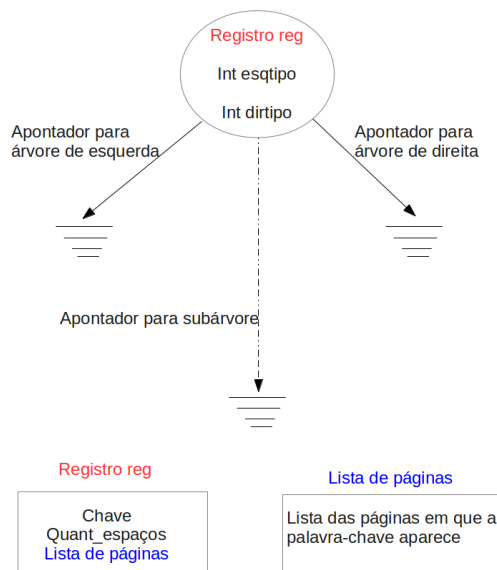
2. Implementação

2.1. Estruturas utilizadas

A estrutura principal do programa será explicada abaixo, visto que a mesma faz uso de todas as outras estruturas.

- sbb é o nó da árvore sbb, que é a estrutura principal do arquivo. Cada nó possui um apontador para um nó da esquerda e um para o nó da direita (árvore comum). No entanto, como o trabalho exigia um hierarquia entre registros, foi criado um outro apontador para uma sub-árvore. Além disso, cada nó da árvore contém um registro, que identifica-o com uma chave e a quantidade de espaços que está do início da linha. Caso seja usada a função de inserção, a quantidade de espaços será a quantidade do pai + 1. Por fim, cada nó também terá uma lista de páginas. Essa lista é um indicativo de em quais páginas a palavra-chave desse nó ocorre.

```
typedef struct registro {
    char *chave;           // string que contém a palavra-chave
    int qut_espacos;       // quantidade de espaços que a p-chave
                           // possui, do começo da linha
    TipoLista listaPaginas; // contém as informações referentes às
                           // páginas em que a palavra chave
                           // se encontra
    struct registro *Prox; // Usado para fila, lista e pilha
} registro;
```



2.2. Decisões e Percurso de Implementação

Para o desenvolvimento do programa, segui uma linha de raciocínio, visando resolver o mesmo de forma algorítmica.

Primeiramente, fiz uma função que armazena a quantidade de registros que será necessário guardar, para montar a árvore inicial. Depois armazenei todos os elementos lidos em um vetor. Esse vetor foi usado para posterior povoamento da árvore.

Em cada posição do vetor, foi armazenado a quantidade de espaços que cada elemento está distante do início da linha e a chave. Após o vetor estar preenchido, coloca todos os elementos do vetor em uma árvore.

Para preencher a árvore, foi criado um mapa, contendo o caminho que deveria ser feito na mesma para a inserção de um novo elemento. O mapa tem o tamanho do vetor, portanto sua indexação é direta e ocorre em $O(1)$. A cada interação com o vetor o mapa é atualizado.

Exemplo: Inserir uma chave que dista 2 espaços do início da linha. Vai percorrendo a árvore até achar o último elemento que está na posição [0] do mapa e vai para sua sub-árvore, depois para a posição [1] e sua sub-árvore e assim se chega na árvore que quer inserir.

Para a função **busca** a implementação foi a seguinte. Percorre toda a árvore com uma flag = 0, quando se acha o nó procurado a flag muda para 1, empilha o valor lido e retorna. A cada retorno faz a verificação da flag, se for 1 empilha o elemento e retorna. Dessa forma teremos uma pilha com todos os elementos em que a flag estava 1. Depois, para impressão é só desempilhar um a um e imprimir.

Para a função de **busca intervalo** foi utilizado a mesma técnica para a função de busca. A ideia diferente é que após ter as duas pilhas montadas começa a desempilhar, comparando os dois elementos desempilhados, se forem iguais repete o procedimento. Faz esse procedimento até achar nós diferentes. Quando acha os nós diferentes, isso quer dizer que o último nó igual é o nó de ligação entre as duas pilhas. Portanto, ele será o nó do caminho. Por fim, pegamos a pilha da esquerda e empilhamos ela novamente, desempilha e imprime (essa vai conter o caminho de baixo para cima), depois imprime o último nó igual (nó de ligação) e, finalmente, desempilha a segunda pilha imprimindo seus elementos.

Para a função de **busca chaves pagina** foi criada uma lista. É necessário percorrer toda a árvore verificando se cada nó possui um página que esteja no intervalo, se sim, coloca na lista. Após percorrer toda a árvore a lista vai estar preenchida, então é só imprimí-la em ordem alfabética.

Para a função de **inserir** é necessário achar o nó pai, percorrendo toda a árvore. Quando achar, insere o nó filho na sua sub-árvore.

Finalmente, para a função **contar impares**, o procedimento é o seguinte: percorre toda a árvore verificando se o nó possui alguma página ímpar, se sim, insere em uma lista. Após percorrer toda a árvore a lista vai estar preenchida. Então, conta-se os elementos da lista e imprime.

2.3. Funções Principais e Análise de Complexidade

- void InserirNovo(registro filho, sbb **raiz, registro pai);
Procura pelo pai na árvore, se achar, insere na sua sub-árvore. Para inserção é necessário percorrer no máximo o tamanho da árvore, portanto a complexidade é $O(n)$.
- void ContaChavesEmPaginasImpares(sbb *raiz, TipoListaRegistros *l);
Preenche uma lista com todos os elementos que contém página ímpar. É necessário percorrer toda a árvore, logo a complexidade é $O(n)$.
- int SalvaCaminho(sbb *raiz, registro chave, int *flag, TipoPilhaRegistros *p);
É necessário percorrer a árvore até o máximo número de elementos para achar o caminho da árvore até o nó, resultando em complexidade $O(n)$.
- void SalvaRegistrosEmIntervaloDePaginas(TipoListaRegistros *v, int pag1, int pag2, sbb *raiz);
Como nas funções anteriores, é necessário percorrer a árvore até o final, para verificar se cada elemento está no intervalo de página, logo a ordem de complexidade é $O(n)$.
- CaminhoEntreDoisNos(sbb *raiz, registro reg1, registro reg2, FILE *arqSaida);
A função vai percorrer a árvore para achar se cada registro passado como parâmetro está nela. Portanto, ela pode percorrer até $2*n$, onde n é o número de elementos da árvore. Logo, sua complexidade é $O(n)$.
- void PreencheVetorRegistros(registro *v, FILE *arqEntrada);
Percorre o arquivo até achar o primeiro “n”, portanto a complexidade é $O(n)$, onde n é o número de linhas até chegar a uma linha vazia.
- void PreencheVetorUltimoElemento(registro *ultimos_por_nivel, int pos, registro *vetor_registros);

- Só faz atribuição por indexação direta ao vetor, resultando em complexidade $O(1)$.
- registro *CaminhoParaInsercao(registro *ultimos_por_nivel, int pos, registro *vetor_registros);
Faz uso da função anterior, somente uma vez a cada interação, logo a complexidade será $O(1)$.
- int VerificaExistenciaRegistro(registro r, TipoListaRegistros *l);
Dado uma lista com n registros, a função faz a verificação se o registro está presente na lista, resultando em no máximo n comparações. Logo a complexidade é $O(n)$.
- int ContaElementosNaLista(TipoListaRegistros l);
Dado uma lista com n registros, a função faz a contagem de elementos presentes na lista, resultando em n contagens. Logo a complexidade é $O(n)$.
- void OrdenaListaRegistros(TipoListaRegistros *l1, TipoListaRegistros *Lista);
Para n elementos, a função precisa de percorrer $l1$ n^2 vezes, procurando o menor elemento e mais n^2 vez para desempilhar esse elemento. Portanto, serão realizadas no máximo $2 \cdot (n^2)$ comparações, resultando em uma complexidade $O(n^2)$.
- FILE *AbreArquivo(char *nome_arquivo, char *modo_leitura);
Para se abrir qualquer arquivo a ordem de complexidade é constante, portanto a ordem de complexidade é $O(1)$.
- void FechaArquivo(FILE *arquivo);
Para se fechar qualquer arquivo a ordem de complexidade é constante, portanto a ordem de complexidade é $O(1)$.

3. Organização do código e Detalhes Técnicos

3.1. Organização

O código enviado foi dividido em vários arquivos, para uma melhor organização do código. São eles:

- arvoreB.c/h*: Contém todas as estruturas necessárias para a manipulação de árvores.
- io.c/h*: Cuida da parte de manipulação dos arquivos.
- main.c*: Arquivo principal, que faz o programa executar.
- operacao.c/h*: Contém as funções necessárias para leitura e execução das operações lidas do arquivo.
- pagina.c/h*: Contém as funções para a organização das páginas
- registro.c/h*: Contém as funções para criação e manipulação dos registros.

3.2. Detalhes técnicos

A IDE de desenvolvimento utilizada foi o Codeblocks 10.05.

O notebook onde foram rodados os testes é um Core i5 com 4GB de RAM.

Os testes foram executados no sistema operacional Linux Mint 14.

Para executar o programa basta inicializá-lo por linha de comando, como no exemplo seguinte:

`./exec entrada.txt saida.txt`

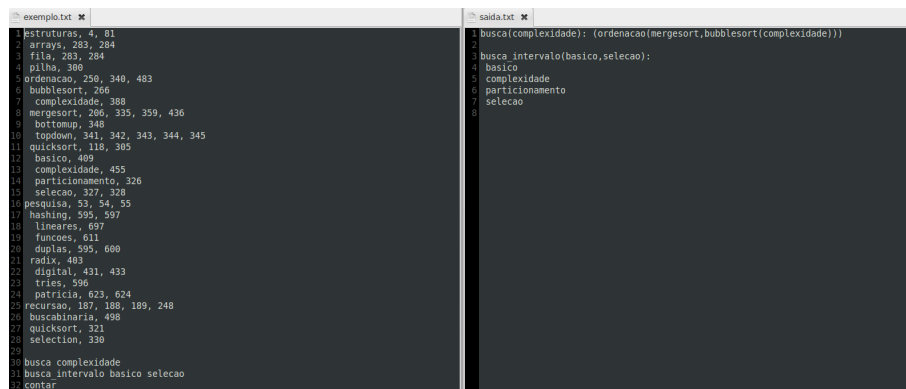
onde *exec* é o nome do executável, *entrada* contém o nome do arquivo de entrada, *saida* contém o nome do arquivo de saída.

4. Testes

Foram realizados três tipos de teste diferentes, os quais foram passados no arquivo que os monitores disponibilizaram no fórum da disciplina. Para esses testes o programa resultou nos valores esperados, mostrando exatamente o resultado de cada uma das funções pedidas. Abaixo, será colocado o teste e a saída.

4.1. EXEMPLO.TXT

Para o arquivo exemplo.txt, segue a figura de resultado e o arquivo de entrada.

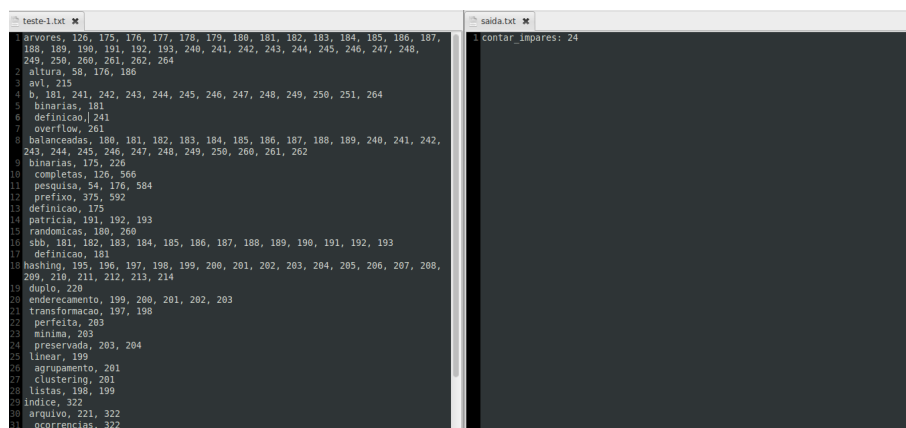


```
exemplo.txt
1 estruturas, 4, 81
2 arrays, 283, 284
3 fila, 283, 284
4 pilha, 306
5 ordenacao, 250, 340, 483
6 bubblesort, 266
7 complexidade, 388
8 mergesort, 266, 335, 359, 436
9 bottomup, 348
10 topdown, 341, 342, 343, 344, 345
11 quicksort, 116, 305
12 basico, 409
13 complexidade, 455
14 particionamento, 326
15 selecao, 327, 328
16 pesquisa, 53, 54, 55
17 hashing, 595, 597
18 lineares, 607
19 funcoes, 611
20 duplas, 595, 600
21 radix, 403
22 digital, 431, 433
23 tries, 596
24 patricia, 623, 624
25 recursao, 187, 188, 189, 248
26 buscabinnaria, 498
27 quicksort, 321
28 selection, 338
29
30 busca complexidade
31 busca_intervalo basico selecao
32 contar

saida.txt
1 busca(complexidade): (ordenacao(mergesort,bubblesort(complexidade)))
2
3 busca_intervalo(basico,selecao):
4 basico
5 complexidade
6 particionamento
7 selecao
8
```

4.2. TESTE-1.TXT

Para o arquivo teste-1.txt, segue a figura de resultado e o arquivo de entrada.

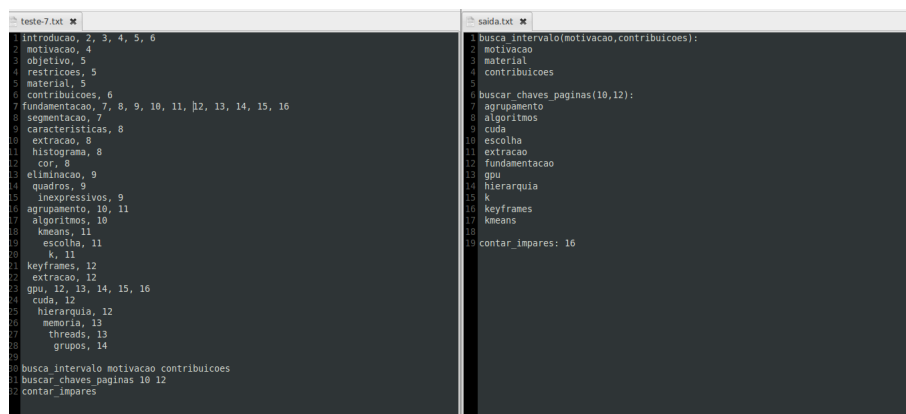


```
teste-1.txt
1 arvores, 126, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187,
2 188, 189, 190, 191, 192, 193, 240, 241, 242, 243, 244, 245, 246, 247, 248,
3 249, 250, 260, 261, 262, 264
4 altura, 98, 176, 186
5 avl, 215
6 b, 181, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 264
7 binarias, 181
8 definicao, 241
9 overflow, 261
10 balanceadas, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 240, 241, 242,
11 243, 244, 245, 246, 247, 248, 249, 250, 260, 261, 262
12 binarias, 175, 226
13 completas, 126, 568
14 pesquisa, 54, 176, 584
15 prefixo, 375, 592
16 definicao, 175
17 patricia, 191, 192, 193
18 randomicas, 180, 260
19 sbb, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193
20 definicao, 181
21 hashing, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208,
22 209, 210, 211, 212, 213, 214
23 duplo, 228
24 enderecamento, 199, 200, 201, 202, 203
25 transformacao, 197, 198
26 perfeita, 203
27 minima, 203
28 preservada, 203, 204
29 linear, 199
30 agrupamento, 201
31 clustering, 201
32 listas, 198, 199
33 indice, 322
34 arquivo, 221, 322
35 ocorrencias, 322
36

saida.txt
1 contar impares: 24
```

4.3. TESTE-7.TXT

Para o arquivo teste-7.txt, segue a figura de resultado e o arquivo de entrada.



```
teste-7.txt
1 introducao, 2, 3, 4, 5, 6
2 motivacao, 4
3 objetivo, 5
4 restricoes, 5
5 material, 5
6 contribuicoes, 6
7 fundamentacao, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16
8 segmentacao, 7
9 caracteristicas, 8
10 extracao, 8
11 histograma, 8
12 cor, 8
13 eliminacao, 9
14 quadros, 9
15 inexpressivos, 9
16 agrupamento, 10, 11
17 algoritmos, 10
18 kmeans, 11
19 escolha, 11
20 k, 11
21 keyframes, 12
22 extracao, 12
23 gpu, 12, 13, 14, 15, 16
24 cuda, 12
25 hierarquia, 12
26 memoria, 13
27 threads, 13
28 grupos, 14
29 busca_intervalo motivacao contribuicoes
30 buscar_chaves paginas 10 12
31 contar_impares

saida.txt
1 busca_intervalo(motivacao,contribuicoes):
2 motivacao
3 material
4 contribuicoes
5
6 buscar_chaves_paginas(10,12):
7 agrupamento
8 algoritmos
9 cuda
10 escolha
11 extracao
12 fundamentacao
13 gpu
14 hierarquia
15 k
16 keyframes
17 kmeans
18
19 contar_impares: 16
```

5. Conclusão

O entendimento do problema proposto foi um pouco demorado, pois houve alguns detalhes que não tinha entendido bem. No princípio do desenvolvimento do projeto, tive algumas dificuldades com a inserção em árvores SBB, visto que foi a primeira vez que precisei utilizar árvores. A parte de desenvolver uma lógica para a leitura dos espaços e posterior inserção na árvore levou muito tempo. Demorei bastante tempo para pensar em criar um mapa com o caminho da inserção. Após pensar nesse mapa também demorei algum tempo para implementar e atualizá-lo a cada inserção.

Após mexer por um longo tempo no TP e ir em algumas monitorias (as quais me ajudaram bastante, pois pude desenvolver algumas ideias e conversar com outros alunos sobre o problema) consegui resolvê-lo inteiramente. Para os testes disponibilizados pelos monitores, todas as minhas saídas foram certas, pois conferi com outros alunos e também fiz a inserção “na mão”, obtendo o mesmo resultado. Porém, não sei o que aconteceu, mas meu programa não passa em nenhum teste do prático. As figuras dos meus testes estão acima, como estão um pouco pequenas, para não desformatar muito o arquivo, se necessário, utilizando-se o zoom conseguimos ver as figuras perfeitamente. Se for necessário, também posso procurar os monitores e mostrar meu programa executando o código, mostrando que as saídas estão corretas.

Após a finalização do trabalho, fiquei bem satisfeito com meu desempenho, visto que pude praticar todos os conceitos vistos em AEDSII, que foram: alocação dinâmica, manipulação de arquivos, ordenação, uso de listas, pilhas e filas, além manipulação de árvores. Tenho a certeza que após finalizar esse TP, estou mais preparado para AEDSIII, pois pude me aprofundar bastante nos conceitos acima. Portanto, imagino que os resultados ficaram dentro do esperado, uma vez que pude praticar bastante os conceitos da linguagem C.

Por fim, acredito que o trabalho proposto cumpriu seu objetivo, de proporcionar ao aluno maior prática e aprendizagem da linguagem C.

6. Referências

- 1 Ziviani, N. (2004). Projeto de Algoritmos com Implementações em C e Pascal. Editora Thomson.
- 2 Schwartz, W.R (2013). http://homepages.dcc.ufmg.br/~william/teaching/2013_01/files/AEDSII_aula.021_arvores_binarios_de_pesquisa_SBB.pdf