Arquivos Invertidos com Árvore Patricia

Carlos Brandt

July 24, 2009

Apresentação

Neste trabalho implementamos um estrutura de dados do tipo árvore chamada PATRICIA para a criação de arquivos invertidos(índice) referente a um dado texto de acordo com um arquivo de palavras chave desejadas.

Abaixo são mostrados resultado do programa e em seguida o TAD implementado.

Resultado teste

```
chbrandt:PTree $ ./ptree teste_keys.txt teste_text.txt
3 argumentos

Criando P-Tree...
Chave sendo lida: programs
Chave sendo lida: easy
Chave sendo lida: by
Chave sendo lida: and
Chave sendo lida: be
Chave sendo lida: to
Arvore criada.

Qual a palavra-chave a procurar?
easy

 Ocorrencias de *easy*, linhas: 3, 4

Indice completo:
easy:   3 4
be:     3
and:    4 5 6
to:     3 4
programs:       3
by:     2 6 7
```

TAD

```
/******************************************************************************/
/* CREATE PATRICIA TREE */
// dep:

PTree* ptree_create(char* key_file, char* text_file) {

    int _bit;
    int var_bitPosition, var_bitValue;
    char key[vSIZE];    // string to store keys read from key_file
    PTree *p_leaf;      // pointer to leaf structures with <key> value
    PTree *paux;        // <paux> walk within structures
    PTree *p_tree;      // main ptree pointer


    FILE* fp_keyfile = fopen(key_file, "rt");
    if (fp_keyfile == NULL) {
        fprintf(stderr, "ERROR: Unable to open the keys file. Finishing program.\n");
        exit(EXIT_FAILURE);
    }


    // Initialize the Tree pointer.
    p_tree = NULL;


    /*
     * We have to compare the new keys with the ones
     * in the tree leaves.
     */
    while (fgets(key, SIZE, fp_keyfile) != NULL) {

        // Check whether the line read is empty.
        remove_newlinechar(key);
        lower_string(key);

        // If the key is empty or is just a character, do nothing.
        if (strlen(key)<=1)
            continue;

        fprintf(stderr,"Chave sendo lida: \%s\n",key);

        /*
         * First we need to treat the empty tree.
         * We simply create the first leaf to see the sun shinning...
         */
        if (p_tree == NULL) {
            p_tree = ptree_createleaf(key, NULL);
            p_tree->p_oc = search_text4key(key, text_file);
            continue;

        }

        /*
         * Right below I’ll treat the particular case of
         * the second insertion (a leaf) and Node allocation.
         */
        if (p_tree->tipo == LEAF) {

            p_leaf = ptree_createleaf(key, NULL);
            p_leaf->p_oc = search_text4key(key, text_file);

            p_tree = ptree_createnode(p_tree, p_leaf, NULL, 0);

            continue;

        }

        /*
         * By now we have a tree with two leaves and a node.
         * Lets add keys to tree from now on following the next procedures.
         */
        if (p_tree->tipo == NODE) {

            // Create the leaf node with a new key.
            p_leaf = ptree_createleaf(key, NULL);
            p_leaf->p_oc = search_text4key(key, text_file);

            /* Walk through Nodes till find a Leaf. */
            paux = walk_ptree(p_tree, p_leaf->chave, LEAF); // paux is pointing to a leaf

            /* Check bit location to verify */
            _bit = paux->p_acima->bit;
            var_bitPosition = compara_chaves(paux->chave, p_leaf->chave);
            var_bitValue = ptree_bit(var_bitPosition, p_leaf->chave);

            /*
             * If the new key differs on a later bit then the existing ones (keys),
             * just add the new node in place of the old (recently compared) key.
             */
            if (var_bitPosition == _bit)
                continue;

            if (var_bitPosition > _bit) {

                paux = ptree_createnode(paux, p_leaf, paux->p_acima, var_bitPosition); // Her
                paux = connect_nodes(paux, p_leaf->chave, paux->p_acima); // paux exits point
                continue;

            } else {

                /*
                 * Else, if the new key differs in earlier bit position, look for
                 * the place inside the tree where to place the new node.
                 */
                paux = walk_upnodes(paux, var_bitPosition, BELOW);
                paux = ptree_createnode(paux, p_leaf, paux->p_acima, var_bitPosition);
                paux = connect_nodes(paux, p_leaf->chave, paux->p_acima);

            }
            /* fi */
            p_tree = walk_upnodes(paux, 0, ABOVE);
        }
        /* fi */

    }
    /* while done */

    return p_tree;
}
/*///*/


/******************************************************************************/
/* READ KEY OCCURRENCES ON TEXT */
//

int ptree_getoccurrences(PTree *p_tree, char *key, int **occurrences) {

    PTree* p;
    List* paux;
    int contador = 0, i;

    p = p_tree;

    // Lower key character.
    lower_string(key);

    // Walk through tree till a Leaf and check whether the keys match each other.
    p = walk_ptree(p, key, KEY);
    if (p == NULL) {
        *occurrences = NULL;
        return 0;
    }

    // If they match, count it.
    paux = p->p_oc;
    while (paux != NULL) {
        contador++;
        paux = paux->p;
    }

    // Create occurrences vector and add to it the occurrences.
    *occurrences = (int*) malloc(contador * sizeof (int));
    paux = p->p_oc;
    for (i = 0; i < contador; i++) {
        (*occurrences)[i] = paux->noc;
        paux = paux->p;
    }

    return contador;
}
/*///*/


/******************************************************************************/
/* PRINT TREE KEYS */
// dep:

void ptree_print(PTree* p) {

    if (p != NULL) {

        ptree_print(p->zero);
        ptree_print(p->um);

        // Find a Leaf, print key and occurrences.
        if (p->tipo == LEAF) {
            printf("\%s: \t", p->chave);
            List* pL = p->p_oc;
            while (pL != NULL) {
                printf("\%d ", pL->noc);
                pL = pL->p;
            }
            printf("\n");
        }

    }

    return;
}
/*///*/


List* list_destroy(List* p);
```

```
/******************************************************************************/
/* DELETE P-TREE NODES*/
// dep:

PTree* ptree_destroy(PTree* p) {

    if (p != NULL) {
        ptree_destroy(p->zero);
        ptree_destroy(p->um);
        list_destroy(p->p_oc);
        free(p);
    }

    return NULL;
}
/*///*/


/******************************************************************************/
/* DELETE LIST OF KEY OCCURRENCES */
// dep:

List* list_destroy(List* p) {
    if (p != NULL) {
        list_destroy(p->p);
        free(p);
    }
    return NULL;
}
/*///*/
```

Função ”bit” auxiliar

```
void ascii2bit(char letra, int* bin);
int* int2bit(int inteiro, int* bin);
int compara_chaves(char* pkey, char* key);


/******************************
 *
 * ptree_bit():
 *   Funcao que recebe um dado valor posicional de um procurado bit
 *   e uma string(ponteiro) para verificar o valor do tal do bit(0,1).
 *   A funcao retorna o valor do i-esimo bit (0,1) da string.

 */
int ptree_bit(int i, char* key) {
    // "i" assume valores de 1 a 64 (as palavras tem 8 letras(char) no maximo).
    // 8 letras x 8 bits = 64

    int posicao, bit;
    int bin[] = {0, 0, 0, 0, 0, 0, 0, 0};

    /*
     * Primeiro temos que encontrar que letra corresponde ao i-esimo bit pedido
     * e que bit eh este dentre os 8 de cada elemento char de uma string:
     */
    i--;
    posicao = i / 8;
    bit = i \% 8;

    /*
     * Agora convertemos o caractere correspondente ‘a "posicao" encontrada para binario:
     */
    ascii2bit(key[posicao], bin);

    // Retorna o valor binario do bit "bit" da letra em "posicao":
    return bin[bit];
}
/* --- */
```