UNIVERSIDADE DE SÃO PAULO

INSTITUTO DE CIÊNCIAS MATEMÁTICAS E COMPUTAÇÃO

PROJETO 1: “Domain Name Server - DNS”

Matheus Aparecido do Carmo Alves - 9791114

São Carlos/SP

2016

PROJETO 1: “Domain Name Server - DNS”

Matheus Aparecido do Carmo Alves - 9791114

Relatório de projeto apresentado como requisito para avaliação na disciplina SCC0202 Algoritmos e Estruturas de Dados I, Instituto de Ciências Matemáticas e Computação, Universidade de São Paulo.

Prof. Gustavo Batista

Prof. PAE Igor Bueno Corrêa

Período: 23 / 09 / 2016 **a** 13 / 10 / 2016

São Carlos
2016

SUMÁRIO

1. INTRODUÇÃO

A partir do conhecimento obtido durante as aulas de Algoritmos e Estruturas de Dados I, este primeiro projeto visa à discussão sobre o **TAD Dicionário** (busca/retorno – chave/elemento) implementado em uma **Lista Ligada Simples** e em uma **"*Skip List*"** de modo a ser possível comparar o desempenho das duas implementações.

2. OBJETIVOS

A ideia base para a implementação do algoritmo é a de se fazer uma busca em uma tabela DNS ("*Domain Name Server*"), onde o usuário terá três operações básicas: Inserção, Remoção e Busca. Assim, quando o usuário criar uma tabela DNS e entrar com o nome do site para procura (chave), terá o retorno do IP do servidor, possibilitando assim a análise do **tempo de execução para casos específicos**, **eficiência para casos gerais (complexidade do algoritmo)** e **discussão geral sobre a comparação das duas listas.**

3. MÉTODOS

A metodologia utilizada para coletar dados é de viés empírico, mas se prioriza o estudo do código quanto a analise de sua complexidade para o pior caso.

Para a coleta de dados foram utilizados funções especificas da linguagem C pertencentes à biblioteca "*time.h*" que capturam o tempo real de execução de um algoritmo.

4. RESULTADOS E DISCUSSÃO

Inicialmente, vale dizer que optou-se por se realizar algumas otimizações no código, principalmente no código de inserção da *Skip List.*

A otimização na inserção é quanto à ideia de se inserir um novo elemento a partir de seu nível mais alto e ir descendo inserindo-o nos próximos níveis até a base da *Skip List,* ao invés de se inserir primeiro na base e depois ir verificando se este elemento existirá no nível superior ou não.

Para esta adaptação, a mudança principal no algoritmo é a verificação das quantidades de níveis totais para um elemento antes de sua inserção através de uma função probabilística. Desta forma, o funcionamento total do código não mudou, só que pelo fato da definição do numero total de nível ocorrer previamente, pode-se aproveitar a própria organização da lista (parecida como uma arvore binaria de busca) para acelerar o processo, buscando de maneira mais eficiente o local de inserção.

A Figura 1 ilustra o código usado para a função probabilística.

```
int letsflipacoin(){
    return rand() % 2;
}

int randlevel(int n){
    int level = 0;

    while(letsflipacoin()) level += 1;

    return level;
}
```

Figura 1

Dito isto, faremos agora uma analise de cada TAD implementado com uma analise de cada operação individualmente e ao final uma discussão geral sobre os resultados.

DADOS – LISTA LIGADA SIMPLES

Na coleta de dados para a Lista Ligada, foram obtidos os seguintes tempos (Figura 2):

NUMERO DE ELEMENTOS	TEMPO TOTAL
1.000 (1K)	**0.000 s**
10.000 (10K)	**0.000 s**
100.000 (100K)	**160.000 s**
1.000.000 (1M)	**-- ***

Figura 2

*O tempo para 1.000.000 (1M) não foi estimado por teste devido à demora de execução.

Realizando uma analise de complexidade geral do código temos que o tempo total de execução respeita a seguinte equação (i = numero de operações de inserção, s = numero de operações de busca, r = numero de operações de remoção):

$$T(n) = (a)i + (bn)s + (cn)r$$

Figura 3

Esta formula geral foi obtida através da soma das formulas de complexidade no pior caso de cada operação possível de se realizar no TAD em cima da aplicação da Lista Ligada implementada.

a. Variável 'a' da equação – Insert_LinkedList – O(1)

Realizando um calculo de complexidade em cima desta inserção, conclui-se ao final que o custo desta operação é constante, pois independente de quantos elementos existam na lista, a inserção de um novo elemento é O(1), logo constante e igual a um valor 'a' – desta forma, qualquer otimização que ocorrer em cima do algoritmo não resultará em uma melhora significativa em cima do código.

```
void insert_linkedlist(struct linkedlist *linkedl, char *host_name, char *host_ip){
    if(linkedl->list_len == 0){
        struct node *node = create_linkedl_node(NULL,host_name,host_ip);
        linkedl->start = node;
        linkedl->list_len += 1;
        return;
    }
    else{
        struct node *node = create_linkedl_node(linkedl->start,host_name,host_ip);
        linkedl->start = node;
        linkedl->list_len += 1;
        return;
    }
}
```

Figura 4

b. Variável 'bn' da equação – Search_LinkedList – O(n)

Realizando um calculo de complexidade em cima da busca em uma lista ligada simples, é possível concluir que, para o pior caso, temos uma operação de custo linear dependente do numero de elementos existente na tabela DNS. Assim, no pior dos casos, temos uma

complexidade O(n) igual ao somatório do valor constante que representa as operações internas à busca.

$$T(n) = \sum_{i=1}^{n} b = bn$$

Figura 5

```
void search_linkedlist(struct node *node, char *name){
    if(node == NULL){
        printf("-1\n");
        return;
    }
    if(strcmp(node->host_name,name)==0){
        printf("%s\n",node->host_ip);
        return;
    }

    return search_linkedlist(node->next,name);
}
```

Figura 6

c. Variável ‘cn’ da equação – Delete_LinkedList – O(n)

Realizando um calculo de complexidade em cima da remoção em uma lista ligada simples, conclui-se que, para o pior caso, temos uma operação de custo linear dependente do numero de elementos existente na tabela DNS, pois esta depende da operação de busca, portanto possui comportamento semelhante a este.

A remoção em si possui complexidade igual a O(1), contudo, devido a procura, temos uma complexidade O(n) + O(1). Desta forma, na teoria, teríamos algo como T(n) = O(n) + O(1) = k_1*n + k_2. Na pratica, podemos reescrever esta formula (sem perder significativamente a precisão em uma analise de complexidade) como:

$$T(n) = \sum_{i=1}^{n} c = cn$$

Figura 7

```
void delete_linkedlist(struct node *node, char *name, struct node *prev, struct linkedlist * linkedl){
    if(node == NULL)
        return;

    if(strcmp(node->host_name,name)==0){
        if(prev == NULL)
            linkedl->start = node->next;
        else
            prev->next = node->next;

        free(node->host_name);
        free(node->host_ip);
        free(node);
        return;
    }

    return delete_linkedlist(node->next,name,node,linkedl);
}
```

Figura 8

d. Formula e Tempo Obtido

Após esta analise, podemos dizer que os tempos obtidos no primeiro e no segundo caso não são bons para analises e para gerar estimativas e que precisa-se de um maior numero de elementos para iniciar uma analise mais profunda do algoritmo, pois, temos um tempo de execução próximo à zero quando n é igual a 1K ou 10K. Já com 100K, temos um tempo de 160s (aproximados) de execução, o que demonstra de maneira mais concreta o custo de execução de uma série operações em uma Lista ligada, possibilitando a realização de estimativas para o tempo de execução de outros valores 'n'.

DADOS – SKIP LIST

Na coleta de dados para *Skip List*, foram obtidos os seguintes tempos (Figura 8):

NUMERO DE ELEMENTOS	TEMPO TOTAL
1.000 (1K)	**0.000 s**
10.000 (10K)	**0.000 s**
100.000 (100K)	**1.000 s**
1.000.000 (1M)	**20.000 s**

Figura 9

Realizando uma analise de complexidade geral do código temos que o tempo total de execução respeita a seguinte equação:

$$T(n) = (a\log_2 n)i + (b\log_2 n)s + (c\log_2 n)r$$

Figura 10

a.Variável '$a\log_2 n$' da equação – Insert_LinkedList – O($\log_2 n$)

Realizando um calculo de complexidade em cima desta inserção, conclui-se ao final que o custo desta operação é igual ao $\log_2$ do numero de elementos existente na lista. Isto ocorre, pois a inserção (ordenada de forma crescente no caso) depende inicialmente de se encontrar a posição desejada e depois inserir de fato o novo elemento com um custo 'a' constante. Então, uma forma de se otimizar a inserção é otimizando a busca, o que não ocorrerá significativamente.

```
void insert_skiplist(struct snode *snode, char *name, char *ip, int level, int randlv, struct snode *prev){
    if(level == 0 && strcmp(name,snode->next->host_name)<0){
        struct snode *newnode = create_skiplist_snode(name,ip,snode->next,NULL);
        if(prev != NULL)prev->downlvl = newnode;
        snode->next = newnode;
        return;
    }

    if(strcmp(name,snode->next->host_name)<0){
        if(randlv >= level){
            struct snode *newnode = create_skiplist_snode(name,ip,snode->next,NULL);
            if(prev != NULL) prev->downlvl = newnode;
            snode->next = newnode;
            prev = newnode;
        }
        return insert_skiplist(snode->downlvl,name,ip,level-1,randlv,prev);
    }

    else
        return insert_skiplist(snode->next,name,ip,level,randlv,prev);
}
```

Figura 11

b.Variável '$\mathbf{blog_2\ n}$' da equação – Search_LinkedList – O($\mathbf{log_2\ n}$)

Realizando um calculo de complexidade em cima desta inserção, conclui-se ao final que o custo desta operação é igual ao $\log_2$ do numero de elementos existente na lista. Isto é verdade devido ao formato que a estrutura da *Skip List* toma pelo processo probabilístico de se criar novos níveis. Desta forma, o que ocorre de fato é uma busca binaria, então este algoritmo possui uma difícil otimização significativa possível.

```
void search_skiplist(struct snode *node, char *name){
    if(node == NULL){
        printf("-1\n");
        return;
    }
    if(strcmp(node->next->host_name,name)==0){
        printf("%s\n",node->next->host_ip);
        return;
    }

    if(strcmp(name,node->next->host_name)<0)
        return search_skiplist(node->downlvl,name);
    else return search_skiplist(node->next,name);
}
```

Figura 12

c.Variável '$\mathbf{clog_2\ n}$' da equação – Delete_LinkedList – O($\mathbf{log_2\ n}$)

Realizando um calculo de complexidade em cima da remoção em uma *Skip List*, conclui-se que, para o pior caso, temos uma operação de custo $\log_2$ do numero de elementos existente na lista DNS. Assim como na inserção, a remoção depende de uma busca para poder remover de fato um elemento especifico da lista. Logo, uma forma de se otimizar a remoção é otimizando a busca, o que não ocorrerá significativamente de fato, como discutido anteriormente.

```
void delete_skiplist(struct snode *node, char *name){
    if(node == NULL)
        return;

    if(strcmp(node->next->host_name,name)==0){
        struct snode *next = node->next->next;
        free(node->next->host_ip);
        free(node->next->host_name);
        free(node->next);
        node->next = next;
        return delete_skiplist(node->downlvl,name);
    }

    if(strcmp(name,node->next->host_name)<0)
        return delete_skiplist(node->downlvl,name);
    else return delete_skiplist(node->next,name);
}
```

Figura 13

5. CONCLUSÕES

Após a implementação do TAD sugerido e da realização dos testes pedidos, fica claro que a eficiência apresentada pelo algoritmo de *Skip List* é maior do que a apresentada por uma Lista Ligada Simples quando tende-se a ter um 'banco de dados' ou uma tabela com características de crescer com o passar do tempo – característica provada facilmente pela comparação das complexidades no pior caso.

Contudo, para uma aplicação com um baixo numero de elementos, ambas as aplicações acabam se mostrando eficientes e a Lista Ligada Simples possui uma implementação muito mais fácil do que a *Skip List,* sendo assim uma ótima opção para este cenário.

No código apresentado, existem alguns ajustes que ainda podem ser feitos quanto a otimização, mas não impactaram significativamente na eficiência total do algoritmo.

De modo geral, tem-se o melhor desempenho da *Skip List* sobre a Lista Ligada para a grande maioria dos casos de teste, assim como seu maior nível de otimização e flexibilidade para adaptação a outros cenários de aplicação – mudando desde a ordem/mode de inserção até a funcionalidade da lista.

6. REFERENCIAS

"*IME – USP"*
http://www.ime.usp.br/~pf/algoritmos/aulas/lista.html

"Wikipedia – Lista Ligada"
https://pt.wikipedia.org/wiki/Lista_ligada

"Computer Science University of Maryland – Class"
https://www.cs.umd.edu/class/spring2008/cmsc420/L12.SkipLists.pdf

"Emory College – Departament of Mathematics And Computer Science - Course"
http://www.mathcs.emory.edu/~cheung/Courses/323/Syllabus/Map/skip-list-impl.html

"University Of Southern California"
http://ee.usc.edu/~redekopp/cs104/slides/L23_SkipLists.pdf

"UMBC - Department of Computer Science and Electrical Engineering"
https://www.csee.umbc.edu/courses/undergraduate/341/fall01/Lectures/SkipLists/skip_lists/skip_lists.html

"Github"
https://gist.github.com