Hashing Dinâmico

FABRICIO BENEVENUTO DE SOUZA[1], MARTÍN GÓMEZ RAVETTI[1]

[1]Universidade Federal de Minas Gerais - Departamento de Ciência da Computação
Av. Antônio Carlos, 6627, Pampulha - Belo Horizonte, MG, CEP 31270-010, Brasil
{fabricio,martin}@dcc.ufmg.br

Abstract. Neste trabalho realiza-se uma introdução às técnicas de hashing dinâmico e são apresentados esquemas básicos para sua implementação. Apresenta-se, também, um algoritmo de hashing linear e, finalmente, alguns testes realizados para diferentes seqüências de chaves uniformemente distribuídas.

1 Introdução

O objetivo básico dos métodos de acesso é a manipulação da informação, isto é, recobrar e atualizar dados em forma eficiente. Uma das possíveis classificações que podem ser realizadas respeito dos métodos de acesso são: esquemas seqüenciais, com tempo de recuperação de informação $O(n)$; esquemas com estrutura baseada em árvores, com tempo de recuperação de informação $O(log\, n)$; esquemas de transformação de chaves ou hashing, com tempo de recuperação de informação $O(1)$.

Deste bom desempenho surge o principal atrativo da utilização de hashing, quando pretendemos minimizar o tempo de acesso à informação. Contudo, o esquema estático tradicional de hashing não possui um bom desempenho num ambiente dinâmico. Quando a coleção de dados ou arquivo excede o espaço alocado estaticamente, todo o arquivo deve ser reestruturado, com um custo considerável. Também podemos alocar espaço sem necessidade.

Com o objetivo de lidar com este tipo de problema, surge o hashing dinâmico. Este tipo de esquema ou tipos de acesso, adapta-se às mudanças de tamanhos dos arquivos para evitar reorganizações muito custosas no método de acesso.

Quando e por que devemos escolher hashing dinâmico? Para responder a esta pergunta precisamos, em primeiro lugar, apresentar algumas características do hashing estático.

Consideramos, por exemplo, uma função simples, a qual através de um único caractere da chave determina seu endereço na tabela. No exemplo da figura 1 podemos ver que para guardar as quatro palavras precisamos apenas de quatro lugares. Se agora devemos inserir mais uma palavra, por exemplo, *fábrica*, neste caso o endereço calculado já está ocupado, em conseqüência, temos uma colisão. Esta situação apresenta, além da colisão, um problema de capacidade, já que a tabela fica lotada. Uma forma de corrigir estes problemas poderia ser a utilização de dois caracteres da chave para a caracterização do endereço, ao mesmo tempo em que incorporamos mais lugares possíveis para a inserção de elementos. No exemplo, foi incorporado mais um lugar na tabela modificando a função que calcula o endereço.

A consideração de mais um elemento, para sua inserção na tabela, implicaria novamente numa modificação da chave ou da tabela. Neste caso, uma segunda situação poderia ser a utilização de uma lista encadeada, nos endereços cuja capacidade foi ultrapassada. Mas em um ambiente dinâmico, a lista encadeada deverá estar diretamente relacionada com o tempo gasto na recuperação da informação.

Este simples exemplo serve para evidenciar que, conhecendo de antemão o número de chaves, quase sempre podemos encontrar uma função hash estática que resolva nossos problemas. Mas, em um ambiente dinâmico, onde as chaves e, em especial, o número de chaves, não é conhecido, a tabela pode ter que ser reestruturada em várias oportunidades.

O objetivo do hashing dinâmico é o desenho de uma função e de uma estrutura de dados que possa se adaptar a essas mudanças sem perder o desempenho característico dos esquemas de hashing respeito ao tempo de recuperação da informação. Desta forma, minimiza-se o número de vezes em que será necessário reestruturar por completo a tabela.

1.1 Notação

Neste trabalho adotaremos as seguintes terminologias e definições, que foram baseadas no trabalho de Enbody e Du [2]. Estamos assumindo que em todo momento estamos trabalhando com grandes quantidades de dados.

Consideramos a busca ou recuperação de dados de uma chave k, $k \in K$ onde K é o conjunto de chaves possíveis ou domínio. Um arquivo $F \subseteq K$ é uma coleção de registros. Um *bucket* ou página corresponde a um ou a vários setores físicos de armazenamento. A capacidade da página é de b registros.

Como o tempo de acesso a disco é consideravelmente maior que o acesso à memória principal, o tempo

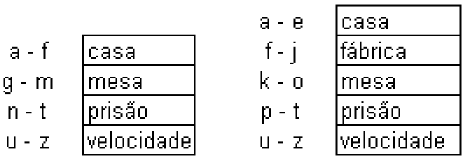

Figura 1: Exemplo de hashing estático

de recuperação de informação será avaliado em termos de números de acesso a disco

O espaço utilizado é a relação entre n e $m.b$, $(n/m.b)$ onde "n"é o número de registros no arquivo, "m"é o número de páginas utilizadas, e "b"a capacidade da página.

1.2 Hashing estático

No esquema de HE, através de uma função de Hash, é mapeada a chave de um determinado registro num determinado espaço de tamanho fixo.

É importante obter uma distribuição uniforme de chaves, sem necessidade de preservar a ordem original ou natural dos registros. Ainda que manter a ordem original seria muito desejável.

Os registros são acessados aplicando a função de Hash à chave e usando o valor da função como no espaço utilizado. Se mais de um registro for mapeado no mesmo endereço, podemos dizer que existe uma colisão, e se mais de b registros forem endereçados à mesma página, então estará estourada a capacidade da mesma (*overflow*).

Uma boa função de hash vai deixar aleatória a atribuição de endereços das chaves sobre toda a tabela. Esta função pode incluir operações como divisão, análise de bits, *foldin* o desdobramento, transformação de base e codificação algébrica. Para um número fixo de chaves, transformações específicas terão um melhor desempenho. Mas, para um conjunto arbitrário de chaves, a divisão realiza um bom trabalho.

Sempre devemos esperar a ocorrência de colisões e *overflows*, listas encadeadas e endereçamento aberto são possíveis métodos para resolver estes problemas [5]. Em geral, à medida que aumenta o número de *overflows* mais registros são armazenados em posições não determinadas pela função hash, o que implica diretamente num aumento no tempo de recuperação dos registros.

A principal restrição ou problema deste esquema é o fato de ter que fixar um espaço de endereçamento no momento da criação da tabela. O número de páginas e a capacidade são parâmetros. Esse inconveniente pode nos levar a situações onde existe um espaço de memória pequeno, ocorrendo um excessivo número de colisões e *overflows* ou a casos de desperdiço de espaço.

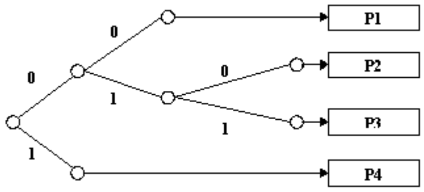

Figura 2: Exemplo de árvore trie

2 Árvores tries

Nesta secao apresentamos os conceitos de arvores tries, uma estrutura muito útil que nos possibilitará introduzir conceitos interessantes.

Uma trie é uma árvore onde a ramificação é determinada por uma porção da chave e não por ela inteira. É uma árvore *M-ária* cujos nós são vetores de M componentes. Cada nó no nível i representa o conjunto de todas as chaves que começam com a mesma seqüência de i dígitos ou caracteres, isto é apenas um exemplo já que poderia ter sido utilizada uma relação diferente.

É importante destacar que a ramificação depende unicamente da chave e não de uma comparação. O nó especifica uma ramificação com M caminhos dependendo do (i+1)-ésimo dígito ou caractere de uma chave. Por simplificação, nos exemplos consideraremos as chaves como números binários com n-bits.(M=2).

Na figura 2 observamos a árvore trie formada ao considerar chaves de 4 bits e um conjunto de registros armazenados. Como podemos perceber, os registros que possuem chaves começando com 00 são colocados na página $P1$, com 010 na $P2$, com 011 na $P3$ e com 1 em $P4$.

Imaginemos a situação em que a página $P1$ sofre um *overflow*, neste caso poderemos utilizar o terceiro bit para dividir a carga desta página, então os registros com chaves 000 continuarão associados à página $P1$, já os registros com chaves começando com 001 estarão endereçados à nova página $P5$. Uma situação oposta poderia ser que a capacidade das páginas $P2$ e $P3$ não esteja sendo utilizada. Em forma similar ao caso anterior poderíamos encaminhar os registros que começam com 01 para serem encaminhados unicamente à página $P2$, eliminando desta forma a página $P3$, ver figura 3.

Como observamos no exemplo, as árvores tries são muito flexíveis já que podem diminuir ou crescer em função do número de registros, mas para encontrar um registro devemos percorrer toda a árvore. O problema surge quando, para determinadas distribuições de chaves, a árvore fica muito desbalanceada o que pode nos levar a tempos de acesso consideráveis. Uma possível solução seria a

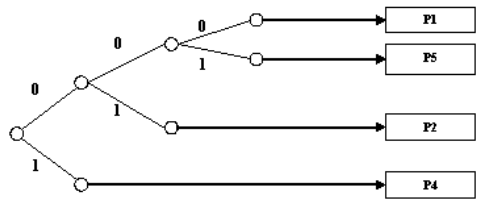

Figura 3: Exemplo de expansão e contração de uma árvore trie

reorganização dos bits, tentando balancear a árvore, mas, criar uma árvore balanceada com profundidade mínima é um problema NP-Completo, [1]. Uma distribuição uniforme das chaves levaria a uma árvore trie balanceada. Hashing dinâmico tenta resolver estes problemas.

Existem dois possíveis esquemas de trabalho referidos ao modo de endereçamento que posteriormente utilizaremos em hashing dinâmico. O primeiro é o esquema de diretórios, onde cada chave indica a posição de um diretório. Neste esquema o tempo de acesso aumenta já que o endereço calculado não leva diretamente até a página.

O segundo esquema é o chamado esquema sem diretório. Neste caso, as chaves nos encaminham diretamente até as páginas, e estas encontram-se num espaço de endereçamento contíguo. São descritos em maior profundidade na seção de hashing dinâmico.

3 Hashing dinâmico, esquemas básicos.

Nesta seção descreve-se uma função geral de hashing dinâmico (HD) e os dois esquemas de arquivos já enunciados, o esquema com diretórios e o sem diretórios.

Da mesma forma em que precisamos uma estrutura que se expanda e contraia em função do número de registros que devem ser armazenados, também precisamos de uma função que produza chaves em um intervalo onde o intervalo também possa de ser controlado.

A maioria dos esquemas dinâmicos utilizam funções que geram mais chaves de bits na medida que cresce o arquivo e menos chaves à medida que o arquivo diminui.

Uma função como esta poderia ser obtida através de uma série de funções $h_i(k)$, $i = 0, 1, ...$, que mapeiam o espaço de chaves K nos inteiros $\{0, 1, ..., 2^{i-1}\}$ tal que para qualquer k ou

$$h_i(k) = h_{i-1}(k)$$

ou

$$h_i(k) = h_{i-1}(k) + 2^{i-1}$$

.

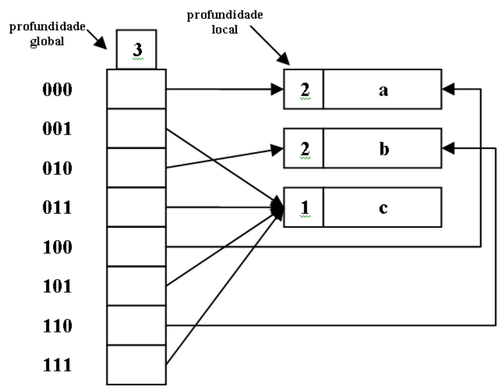

Figura 4: Exemplo de hashing dinâmico, esquema com diretórios

Existem muitas formas de obter as funções $h_i(k)$. Pode ser utilizada uma função $H(k)$ que mapeia o espaço de chaves num padrão de bits aleatório de tamanho m, para um m o suficientemente grande. Então $h_i(k)$ pode ser definida como os inteiros formados pelo último i bits de $H(k)$. Deste modo, se

$$H(k) = b_{m-1}...b_2b_1b_0,$$

onde b_i é 0 ou 1, então,

$$h_i(k) = b_{i-1}...b_1b_0.$$

Reparemos que a função h_i definida utilizando os últimos $i$bit é diferente da função utilizada no exemplo de árvores tries 2.

No esquema de hashing dinâmico podemos começar com uma expansão 0, ou seja, como 2 páginas onde seus endereços estão definidos por 1 bit (b_0). Realizando uma expansão 1, isto é, calculando o endereço da página com os bits b_1 e b_0, estaremos dobrando a tabela, já que estaremos trabalhando com 4 páginas.

Na d-ésima expansão, o arquivo terá $s = 2^d$ entradas de diretórios ou páginas, segundo seja o esquema utilizado.

No esquema com diretórios consideramos 2^d entradas, figura 4,$h_d(k)$ será um número de d-bits que indica o endereço do diretório, apontando para a página que contêm o registro com chave k. Se exatamente duas entradas apontam para a mesma página, então, todos os registros da página terão o mesmo valor nos bits$b_{d-2}...b_1b_0$. Se quatro entradas apontam para a mesma página, os registros terão os mesmos valores nos bits$b_{d-3}...b_1b_0$.

Nestes esquemas não serão permitidas páginas de *overflow*. Ou seja, caso aconteça *overflow*, outra página deve ser inserida ou criada. A inserção poderá ocorrer de duas formas:

- Se a página possui dois ponteiros apontando para ela, então a nova página será inserida de forma que um desses ponteiros aponte para ela. Desta forma, os registros na página com *overflow* serão divididos entre ela e a nova página segundo o valor $h_d(k)$.

- Caso não tenhamos dois ponteiros apontando para a página com *overflow*, então aumentaremos o número de entradas, e com isto teremos ao menos uma nova página para cada entrada de diretório. Isto é realizado utilizando a função h_{d+1} no lugar da h_d.

Neste processo de incremento das entradas é importante perceber quando devemos dobrar o número de páginas ou incrementar apenas uma página. Para isto, devemos saber qual a "profundidade global"d, que é a mesma coisa que saber quantos bits são necessários para o cálculo do endereço.

A "profundidade local"refere-se ao número de bits necessários para acessar uma determinada página. Desta forma, se uma determinada página Px apresenta *overflow*, analisamos os valores da profundidade local e global. Se os valores são iguais deveremos aumentar a profundidade global dobrando o número de entradas. Mas, se a profundidade local ainda é menor que a global, então poderemos criar uma página apenas aumentando a profundidade local. De forma similar podemos reduzir o número de páginas.

No esquema sem diretórios, assumimos que as páginas se encontram armazenadas num espaço de memória contíguo, ver figura 5. Nesta figura observamos que o arquivo está no inicio da segunda expansão(d=2). contendo $(s = 2^d = 4)$ páginas primárias de dados. Aqui não trabalhamos com diretórios, o endereço é diretamente da página que contêm os registros.

A expansão ocorre de forma similar, quando uma página apresenta *overflow*, os registros que ela contêm devem ser divididos com outra página. A diferença está em que não necessariamente a página que apresentou o *overflow* deve ser dividida, já que esta divisão ocorre em forma seqüencial. Essa forma de divisão seqüencial de páginas será melhor explicada na seção de hashing linear.

Como já foi dito, a função hash apresenta diretamente o endereço da página. Por causa disto, deveremos ter uma página para cada valor da função de hash. No esquema anterior os ponteiros apontavam a um diretório, neste diretório poderíamos ter alocado algumas páginas, o que provocava um aumento no tempo de recuperação dos dados.

No esquema sem diretório, poderemos efetuar a expansão dobrando o número de páginas, realizando uma operação similar à realizada no esquema anterior, isto é, considerando mais um bit para o cálculo do endereço. Mas, à diferença do esquema anterior, a incorporação do novo bit não implica necessariamente a duplicação imediata do número de páginas. As páginas serão inseridas uma a uma e em ordem seqüencial.

Na figura 5, vemos a seqüência em que as páginas são

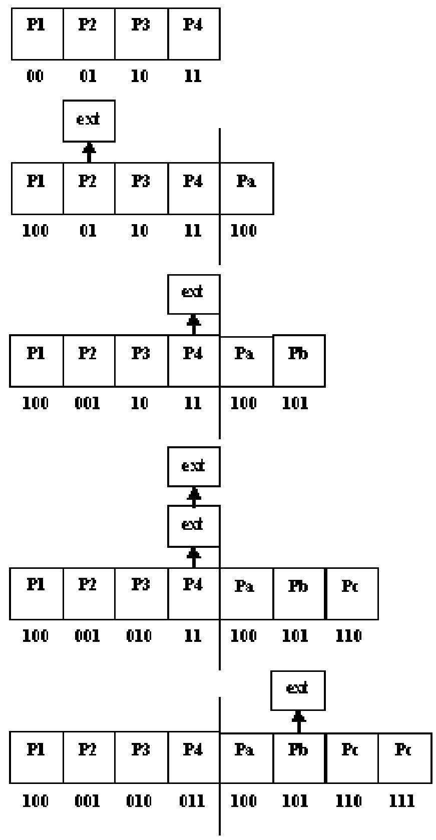

Figura 5: Exemplo de hashing dinâmico, esquema sem diretórios analisando a expansão linear

inseridas. Neste exemplo, as páginas são inseridas quando qualquer página apresenta *overflow*. Seja P2 a primeira página que apresenta *overflow*. O primeiro é inserir uma página extra de capacidade c para armazenar os registros adicionais; a partir deste ponto, o arquivo será expandido inserindo Pa e dividindo os registros entre a página $P1$ e Pa onde, para diferenciar os endereços, utiliza-se um bit extra. A função$h_2(k)$ continua sendo utilizada para as páginas $P2$, $P3$ e $P4$, já para as páginas $P1$ e Pa utiliza-se a função $h_3(k)$. Como a página $P2$ apresentou o *overflow* precisamos uma página extra para lidar com isto.

Depois de algumas inserções, se a página $P4$ apresenta *overflow*, então os registros da página $P2$ serão divididos com a nova página inserida Pb. Como vemos, neste momento a página $P2$ não precisa mais da página extra, já que esta última estará à disposição da página $P4$. Em caso que novamente a página $P4$ apresente *overflow*, a página Pc será inserida dividindo os registros da página $P3$. Supondo *overflow* na página Pb, com a inserção de Pd todas as páginas possuirão um endereço calculado com 3 bits.

No exemplo anterior observamos que as páginas são inseridas cada vez que qualquer página apresenta *overflow*, mas poderíamos utilizar uma política diferente, como ser, realizar a inserção de uma página depois de ter um número específico de inserções de registros, ou depois de ocorridos alguns *overflows*, ou ainda, quando uma página chega a um determinado nível de utilização.

3.1 linear hashing

No início da *d-ésima* expansão, a tabela terá $s = 2^d$ páginas e serão endereçadas por $h_d(k)$. As novas páginas serão inseridas no final do arquivo. Para determinar qual será a próxima página a ser dividida, será utilizado um ponteiro "p". Este ponteiro é inicializado apontando à página 0 e indica qual é a página que terá seus registros divididos.

Na primeira expansão é inserida uma nova página no final do arquivo e os registros da página 0 dividem-se entre ela e a nova página, sendo que esta última será a número $s+p$. No fim da divisão de registros o ponteiro é incrementado. Quando o número de páginas é dobrado, o ponteiro aponta à página "s"$(2^{d+1}pag)$. Neste ponto, "d"deve ser incrementado e o ponteiro retorna ao início do arquivo; desta forma, estamos em condições de reiniciar a expansão.

No esquema com diretório, o endereçamento resulta imediato, já que a função $h_d(k)$ calcula o endereço do diretório, onde encontramos um ponteiro que aponta até a página que contêm o registro procurado.

No esquema sem diretórios devemos levar em consideracao o problema de que à esquerda do ponteiro já ocorreu a expansão enquanto que à direita dele esta ainda não ocorreu. Isto obriga à utilização de diferentes funções de hash para o cálculo do endereço. Para acessar os registros contidos em páginas ainda não divididas, ou seja, os que se encontram à direita do ponteiro "p"até a página "s-1", deve ser utilizada a função hash $h_d(k)$; para acessar os outros devemos utilizar a função $h_{d+1}(k)$.

Uma das diferenças fundamentais entre estes esquemas é que trabalhando com diretórios sem *overflow* pode ser garantido que precisaremos dois acessos à memória para cada pedido, um para o diretório e outro até a página.

Com o esquema sem diretórios, podemos manter muitas consultas com um único acesso, mas ao permitir *overflow*, estes provocam que alguns requerimentos precisem de dois ou mais acessos. A comparação é mais difícil se são permitidos *overflows* no esquema com diretórios.

A distinção encontra-se no fato de que ao utilizar diretórios podemos ou não permitir *overflows*, já no esquema sem diretórios devemos permití-los.

3.2 Utilização de espaço

Definimos como taxa de utilização a relação $(n/w.b)$, onde n é o número de registros, b é a capacidade da página e w é o número de páginas. Quanto maior for a taxa de utilização menor será a quantidade de espaço desperdiçado.

Deve-se observar que seja qual for o esquema utilizado, o tratamento do *overflow* ou a seleção das estruturas, podem modificar a taxa de utilização.

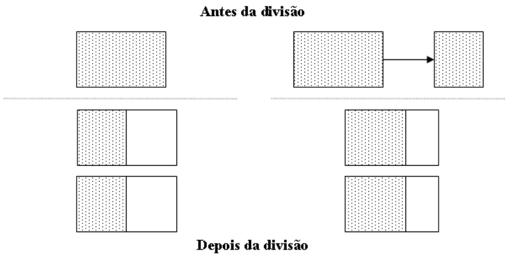

Figura 6: Divisão de páginas permitindo e não permitindo *overflow*

Se não permitimos o *overflow*, ao dividir uma página cheia (b registros), ela será dividida em duas páginas com a mesma capacidade b, ou seja pode esperar-se uma taxa de utilização média de 50%. Depois de realizadas varias expansões, pode esperar-se uma taxa superior a 50% mas inferior a 100%.

Quando permitimos *overflow* ao dividir uma página, onde o excesso de registros pode estar limitado a um parâmetro "c"(página extra), é necessário dividir uma página cheia (b+c) em duas páginas com capacidade b, para ter em média uma taxa superior a 50%, aproximadamente 66% caso $c = b$, ver figura 6. Ao mudar o valor de "c", diferentes taxas de utilização podem ser alcançadas. Uma observação importante é que todas estas afirmações são válidas considerando uma distribuição uniforme das chaves. O não atendimento desta hipótese implicará novos valores médios da taxa de utilização.

Existe uma interessante relação custo-benefício entre a taxa de utilização e o tempo de acesso. Segundo o trabalho de Larson [3] pode concluir-se que para obter elevadas taxas de utilização deve permitir-se um maior número de páginas extras. O aumento destas cadeias de páginas provocam um aumento no tempo de acesso. No trabalho de Enbody e Du [2] podem ser encontradas diversas técnicas para o tratamento do *overflow*.

4 Implementação

A implementação realizada neste trabalho baseia-se no código apresentado por Larson em 1988 [4]. O esquema escolhido foi o de hashing linear sem a utilização de diretórios. Algumas decisões adotadas para esta implementação são:

- Para o tratamento de *overflow* utilizamos uma página extra de capacidade 1 ($c = 1$).
- Foram implementadas as funções Hash, figura 8, para o cálculo do endereço; a função Insere, figura 10, para a inserção de registros e a função Expande, figura 12, para a expansão da tabela.
- As chaves são números inteiros únicos.

Além destas funções foram implementadas as funções auxiliares para a inicialização da tabela e sua impressão. O código completo encontra-se disponível em (http://www.dcc.ufmg.br/ martin/PAA/Seminario).

4.1 Testes

Foram realizados alguns testes básicos com esta implementação, basicamente tentando analisar o desempenho do algoritmo respeito à taxa de utilização e o tempo de recuperação de registros.

A taxa de utilização foi calculada como apresentado na seção 3.2. Para o tempo de recuperação foi calculado o número de acessos à memória assim como o número de comparações realizadas uma vez recuperada a página que contém o registro.

Foram geradas instâncias de 30 chaves únicas e uniformemente distribuídas em inteiros de 0 a 100. A capacidade da página foi fixada em três registros e a da página extra em um só.($b = 3$, $c = 1$). A tabela 4.1 apresenta os resultados obtidos.

A tendência dos resultados obtidos permitem suspeitar a verificação dos resultados obtidos por Larson [3], isto é, uma taxa de utilização média de 69%. No caso do número de acessos e comparações realizadas, pode-se observar que para este número de chaves seria interessante aumentar a capacidade da página extra. Isto provocaria tanto uma diminuição do menor número de páginas divididas quanto do número de acessos necessários para a recuperação de registros.

Como pode ser observado nos resultados dos testes, o hashing linear pode apresentar variações cíclicas no tempo de recuperação de dados. Existem novas implementações, como por exemplo a *spiral storage* ou hashing espiral, que cuida deste tipo de problemas. Uma interessante implementação de hashing espiral pode ser encontrada em [4].

Figura 7: Tipos de dados

```
typedef long TipoChave;
typedef unsigned long Endereco;

typedef struct Registro {
    TipoChave Chave;
    struct Registro *Prox;
    /* Outros componentes */
} Registro;

typedef struct Tabela {
    long p;
    /*Proximo segmento a ser dividida*/
    long maxp;
    /*valor maximo de p nesta expansao*/
    long numreg;
    /*Numero de registros na tabela*/
    long numseg;    /*Numero de segmentos*/
    short int overflow;
    Registro *diretorio[DIRECTORYSIZE];
} Tabela;
```

		Pior Caso	
Amostra	T.U.	Acessos	Comp.
1	76.90%	3	5
2	68.18%	3	5
3	75.00%	3	5
4	71.42%	3	5
5	58.82%	3	5
6	83.33%	2	4
7	71.43%	3	5
Média	72.15%	2.85	4.85

Tabela 1: Comportamento do algoritmo para diferentes seqüências de chaves

Figura 8: Função Hashing

```
Endereco hash(TipoChave k, Tabela T) {
    Endereco endereco;

    endereco = k % T.maxp;
    if (endereco < T.p) {
        endereco = k % (T.maxp * 2);
    }
    return endereco;
}
```

5 Conclusões e trabalhos futuros

Neste trabalho apresentamos uma introdução ao hashing dinâmico. Ambos esquemas apresentados, com e sem diretórios, podem tratar bem o problema de estruturas dinâmicas, estendendo e contraindo a tabela segundo a necessidade.

Foi apresentado um algoritmo que trabalha com hashing linear. Foram geradas instâncias e resolvidas. Mesmo sendo poucos testes, percebemos a tendência a confirmar os resultados obtidos por Larson [3], apresentando uma taxa de utilização perto dos 69% para uma distribuição uniforme das chaves.

Como trabalhos futuros, sugerimos uma análise com grande quantidades de dados, utilizando diferentes distribuições e tratamentos de *overflow*. Comparar, ainda, resultados com outro tipo de estruturas como, por exemplo, as árvores B e B*, ver Ziviani [5]. Outras possibilidades poderiam da pesquisa dos métodos de hashing dinâmico perfeito como em [**?**].

Referências

[1] D. Comer and R. Sethi. The complexity of trie index construction. *Journal of the ACM*, 24(3):428–440, 1977.

[2] R.J. Enbody and H. C. Du. Dynamic hashing schemes. *ACM Computing Surveys.*, 20(2):85–113, 1988.

[3] Per-Ake Larson. Linear hashing with overflow handling by linear probing. *ACM Transactions Database*, 10(1):75–89, 1985.

[4] Per-Ake Larson. Dynamic hashing. *Communications of the ACM*, 31(4):446–457, 1988.

[5] Nivio Ziviani. *Projeto de Algoritmos Com implementações em Pascal e C*. Pioneira Thomson, Brasil, 2004.

Figura 9: Inicializa Tabela

```
void Inicializa(Tabela *T) {
    int i;

    T->p = 0;
    T->maxp = INITIALSIZE;
    T->numreg = 0;
    T->numseg = 1;
    for (i=0; i < T->maxp; i++) {
        T->diretorio[i] =
            (Registro *) malloc(sizeof(Registro));
        T->diretorio[i]->Chave = 1;
        T->diretorio[i]->Prox = NULL;
    }
}
```

Figura 10: Insere a chave k na tabela

```
void Insere(Tabela *T, TipoChave k) {
    int i;
    Registro *reg, *aux;
    Endereco endereco;

    endereco = hash(k, *T);
    reg = (Registro *) malloc(sizeof(Registro));
    reg->Prox = NULL;
    reg->Chave = k;
    aux = T->diretorio[endereco];
    for (i=0; aux->Prox != NULL ;
                        aux = aux->Prox, i++);
    aux->Prox = reg;
    if (i >= SEGMENTSIZE)
    { // Cria nova posicao
        T->diretorio[endereco]->Chave++;
        Expande(T);
    }
    T->numreg++;
}
```

Figura 11: Imprime a tabela

```
void Imprime(Tabela *T) {
    int i;
    Registro *aux;

    printf("\n\n");
    for (i=0; i<(T->maxp+T->p); i++) {
        printf("%d: ", i);
        aux = T->diretorio[i];
        for (; aux != NULL ; aux = aux->Prox)
            printf("%ld\t", aux->Chave);
        printf("\n");
    }
}
```

Figura 12: Expande a tabela

```
int Expande(Tabela *T)
{
    Endereco newaddress, oldaddress;
    Registro *corrente;
    Registro *primeiro;
    Registro *anterior;
    Registro *aux;

    if (T->maxp+T->p >= 20)
    {
        printf("Acabou o espaco\n");
        return 1;
    }
    oldaddress = T->p;
    newaddress = T->maxp + T->p;

    /*Ajusta as variaveis de estado*/
    T->p++;
    if (T->p == T->maxp) {
        T->maxp *= 2;
        T->p = 0;
    }
    T->numseg++;

/*Realoca os registros para uma nova pagina*/
    corrente = T->diretorio[oldaddress];
    T->diretorio[newaddress] =
      (Registro *) malloc(sizeof(Registro));
    T->diretorio[newaddress]->Chave = 1;
    T->diretorio[newaddress]->Prox = NULL;

    while (corrente->Prox != NULL)
    {
        if (hash(corrente->Prox->Chave,*T)
                               ==newaddress)
        {
            aux= corrente->Prox->Prox;
            primeiro = T->diretorio[newaddress];
            anterior = primeiro->Prox;
            primeiro->Prox = corrente->Prox;
            primeiro->Prox->Prox = anterior;
            corrente->Prox = aux;
        }
        else {
            corrente = corrente->Prox;
        }
    }
    return 0;
}
```