Disciplina: ACH2024 - Algoritmos e Estruturas de Dados II
Prof: Helton Hideraldo Biscaro

Resumo P2
Autor: Bruno Kazuhiro

1 - HASH

Bit visitado:

Incluir um bit a mais, inicializado com zero, para cada posição da tabela. Quando ocorre uma colisão, esse bit é atualizado para 1.

Método preditor:

Inclui um inteiro a mais para cada posição da tabela.

Hash encadeado:

Inclui um inteiro a mais, marcando o próximo endereço disponível na tabela, eliminando assim o uso de rehash.

Funções de hashing:

Divisão:

h(k) = k % ts.

Método multiplicação

Floor(m * frac(c*k))

Onde :

Floor = inteiro arredondado para baixo.
Frac = parte fracionária.
m = table size.
c = constante entre 0 e 1.

Quadrado médio:

Elevar a chave ao quadrado, e selecionar alguns bits “do meio” do número resultante.

Dobra:

Ideal para table size = 2 ^ b, e consiste em converter k para a sua representação binária e utilizar a operação de *“ou exclusivo”* com os blocos resultantes.

Chaves alfa-numéricos.

Converter letras em números via tabela ASCII, ou transformar em números considerando letras como um número na base 26. Ambas as formas podem ser utilizadas em funções hash.

Hash:

Uma função é chamada de hash quando, dada uma chave n qualquer, ela retorna um índice para ser utilizado como um endereço de memória numa tabela hash.

A diferença entre hash e indexação é que a função hash possui uma atribuição de índices que à primeira vista soa como aleatória, e duas ou mais chaves podem coincidir em relação ao índice retornado, gerando uma colisão.

Para evitar colisões:

- Uso de algoritmos para distribuir os registros.
- Uso de mais memória
- Uso de cestos/buckets

Fórmula de densidade: r/N
r – Número de registros.
N – número de espaços.

Buckets:

Os buckets permitem que se possa associar um ou mais registros em um único endereço.

A fórmula de densidade é dada por: r/b*N

-> r : número de registros
-> N : número de espaços
-> b : número máximo de registros num cesto.

Melhora o desempenho no espalhamento, aumentando a densidade de ocupação e diminuindo o número de colisões.

Hash Ext x Hash Linear:

-> Bucket cheio:
- dobrar número de buckets
- distribuir entradas nos novos buckets
- O arquivo inteiro deve ser reorganizado.

-> Diretório de ponteiros:
- Dobra o número de entradas no diretório
- Separar buckets cheios.

Hash Linear;

Em hash linear, a função varia visando evitar overflow e não existe a necessidade de utilizar diretórios (como em hash Extensível). Quando ocorre um overflow, a função de hash se modifica.

Parâmetros:

-> Nível: indica a rodada atual (aumentado quando aumenta-se o número de dígitos binários utilizados como índice para alocação).
-> Next: Buckets que deverão sofrer divisão, inicializado com 0. Somente buckets com valores diferentes de 0 poderão ser divididos.

Hash Extensível:

Havendo a necessidade de multiplicar os buckets, o diretório inteiro sofre a duplicação (porém, somente são multiplicados os buckets que estiverem cheios), e pode ocorrer de dois diretórios apontar para o mesmo bucket.

Regra de inserção: calcular h(k), pegar os ‘n’ dígitos da chave em representação binária, localizar o bucket indicado e, se o bucket estiver cheio e o nível local for igual a ‘n’, divide o bucket e duplica o diretório. Se o bucket estiver cheio e o nível local for menor que ‘n’, divide-se o bucket, mas não duplica o diretório.

Análise HL e HE:

Em caso de distribuição uniforme, o hash linear apresenta um custo menor do que o hash extensível. Caso não seja uniforme, o hash extensível ganha uma vantagem.

Distribuição uniforme: não existem páginas de overflow.

2 - ÁRVORE B

Árvores:

-> Binária: cada nó possui no máximo dois filhos.
-> ABB:
- Balanceada: a altura da árvore tem que ser igual, ou com uma diferença de 1 elemento.
- Balanceamento dinâmico: AVL.

-> Árvore B: cada nó pode ter ‘n’ filhos, com um valor ‘n’ escolhido de forma a otimizar a blocagem física do arquivo de índice. Métodos de acesso com baixo custo.

ABB: acesso lento em caso de busca em disco, tendo como solução utilizar um arquivo os registros, e os ponteiros dariam o RRN de entrada dos filhos.

AVL: acesso inviável em memória secundária, pois havendo mais de 1000000 chaves armazenadas deve-se percorrer até 28 níveis.

Fórmula: 1.44*log 2 (N+2).

Paged Binary Tree: a cada seek, todos os registros da mesma página do arquivo são lidos, e se o próximo registro a ser recuperado estiver na mesma página, pode-se poupar o acesso ao disco.

Árvores B:

-> Cada página possui uma sequência ordenada de chaves e um conjunto de ponteiros.
-> Inexiste árvores explícitas.
-> Ordem: número máximo de ponteiros que são armazenados em um nó.
-> Splitting: Separar um nó folha em dois.
-> Promoção: incluir uma chave na raiz (geralmente o primeiro nó do segundo arranjo após o splitting).

Busca: utiliza-se a busca binária, dado que os elementos estão ordenados, com o custo de O(lg(t)*log t (n)).

Inserção: localiza em qual nó o elemento deve ser inserido.

Se o nó estiver cheio, faça o splitting.

Se o nó pai estiver cheio, repita o processo recursivamente até poder inserir.

Exclusão:

- Elemento em um nó não folha: sucessor movido para a posição que deseja-se eliminar.
- Elemento em uma folha: depende da ocupação mínima.
- Se o número de elementos for maior ou igual a OM, apenas remove-se o elemento.
- Se o número de elementos for menor, reorganize a AB.
  - Pode-se pegar elementos de nós irmãos, caso estejam com número de elementos maior que a OM.
  - Sendo igual, o nó atual é concatenado ao seu irmão, formando um único nó.
    - Se o nó pai tiver com o número igual a OM, deve-se incluí-lo nessa concatenação também, podendo alterar o tamanho máximo da árvore.

Lista Generalizada:

São generalizadas todas as listas que possuírem nós de diferentes tipos.

Método para as listas:

Head() – retorna a informação do primeiro elemento da lista.
Tail() – retorna todos os elementos da lista, exceto o head() dela.
First() – retorna o primeiro nó da lista.

Métodos para os nós:

Info() – retorna a informação armazenada no nó.
Next() – retorna o nó seguinte.
Nodetype() – retorna o tipo do nó.

Modificação da lista:

Push(l,x) – cria um nó com a informação ‘x’ e faz com que a lista ‘l’ aponte para esse nó.
Addon(l,x) – retorna uma outra lista ‘l’ tal que tail(‘lista’) = ‘l’ e head(‘lista’) = ‘x’.
setInfo(x, y) – atualiza o info de ‘x’ para ‘y’.
setNext(x,y) – atualiza o next de ‘x’ para ‘y’.
setHead(l,x) – atualiza a informação do head da lista para ‘x’.

Um nó ‘n’ é acessível a partir de uma lista ou de um ponteiro externo se existe uma sequência de gerações head() e tail() que, ao aplicar na lista ‘l’, o resultado é uma lista em que

‘n’ é o primeiro elemento.

Garbage Collector:

Consiste em apagar todos os nós que não possuem ponteiros ligados a eles.

Structs:

- Árvore B:

```
typedef struct arv{
        int Num_Chaves;
        int Chaves[];
        arv* Filhos[];
    }
```

- Lista Generalizada:

```
typedef struct No{
        int type;
        // 1 para int
        // 2 para char
        // 3 para No
        union{
            int infoInt;
            char infoChar;
            No* infoNo;
        }
        No* prox;
    }
```

Códigos:

- Inserção em hash encadeado:

```
bool insert (int k){
    int i = h(k);
    if(T[i] == -1){
        t[i] = k;
        return true;
    }
    while(next[i] != -1 && t[i] != k) i = next[i];
    if(T[i] == k) return false;
    T[dispo] = k;
    next[i] = dispo;
    while(T[dispo] != -1&& dispo > -1) dispo--;
    return true;
}
```

- Remoção em hash encadeado:

```
bool remove(int k){
    int i = h(k);
    if(T[i] == k){
        if(next[i] == -1){
            T[i] = -1;
```

```
                    return true;
                } else {
                        T[i] = T[next[i]].
                    T[next[i]] = -1;
                    next [i] = next[next[i]];
                    next[next[i]] = -1;
                    return true;
                }
            }
            while(T[next[i]] != k && next[i] != -1) I = next[i];
            if(T[i] == -1) return false;
            T[next[i]] = -1;
            next[next[i]] = -1;
            return false;
        }
```

- Busca em Árvore B:

```
            int buscaBin(int chave, No* in){
                int ini = 0;
                int fim = in-> num_chaves - 1;
                int meio;
                while(ini < fim){
                    meio = (int) (ini+fim)/2
                    if(n->chaves[meio] == chave) return meio;
                    else if(n->chaves[meio] < chave) ini = meio +1;
                    else fim = meio -1;
                }
                return ini;
            }
            bool buscaArvoreB(int chave, no* raiz){
                int p;
                no* aux = raiz;
                while(aux != null){
                    p = buscaBin(chave, aux);
                if((p<aux->num_Chaves) && (aux->chaves[p] ==
                chave)) return true;
                        else aux = aux->filhos[p];
                }
                return false;
            }
```

- Imprime Árvore:

```
            void imprimeOrdem(no* raiz){
                    no* aux = raiz;
                    if(aux == null) return;
                    int c = n->num_chaves;
                    int i;
                    for(i = 0; i < c; i++){
```

```
                imprimeOrdem(n->filhos[i]);
                printf("%d\n", n->chaves[i]);
            }
            imprimeOrdem(no->filhos[i]);
        }
```

- Somar uma unidade a todos os ints de uma lista generalizada:

```
        void addUm(lista l){
            no* p = first(l);
                while(p != null){
                    if(nodetype(p) == 1) setInfo(p, info(p) +
1);

                    p = next(p);
                }
            }
```

- Deleta todos os ‘w’s de uma lista generalizada:

```
            void delW(list l){
                no* ant = null;
                no* p = first(l);
                while(p != null){
                    if(nodetype(p) == 2 && info(p) == 'w'){
                        if(ant == null) l = tail(l)
                        else setNext(ant, next(p));
                     } else ant = p;
                    p = next(p);
                }
            }
```