2013-02 – Aula 19
Adaptado por Reinaldo Fortes para o curso de 2013-02
Arquivo original: 20._pesquisa_(parte_1)

Pesquisa Sequencial e Binária

Prof. Túlio Toffolo
http://www.toffolo.com.br

BCC202 – Aula 19
Algoritmos e Estruturas de Dados I

Pesquisa em Memória Primária

- Introdução - Conceitos Básicos
- Pesquisa Sequencial
- Pesquisa Binária
- Árvores de Pesquisa
  - Árvores Binárias de Pesquisa
  - Árvores AVL
- Transformação de Chave (Hashing)
  - Listas Encadeadas
  - Endereçamento Aberto
  - Hashing Perfeito

Introdução – Conceitos Básicos

- Estudo de como recuperar informação a partir de uma grande massa de informação previamente armazenada.
- A informação é dividida em registros.
- Cada registro possui uma chave para ser usada na pesquisa.
- **Objetivo da pesquisa**: Encontrar uma ou mais ocorrências de registros com chaves iguais à chave de pesquisa.
- **Pesquisa com sucesso X sem sucesso.**

Algoritmos de Pesquisa - TADs

- É importante considerar os algoritmos de pesquisa como tipos abstratos de dados (TADs), de tal forma que haja uma independência de implementação para as operações.
- Operações mais comuns:
  1. Inicializar a estrutura de dados.
  2. Pesquisar um ou mais registros com determinada chave.
  3. Inserir um novo registro.
  4. Retirar um registro específico.

Dicionário

- Dicionário é um TAD com as operações:
  1. Inicializa
  2. Pesquisa
  3. Insere
  4. Retira

- Analogia com um dicionário da língua portuguesa:
  - Chaves ⇔ palavras
  - Registros ⇔ entradas associadas com *pronúncia, definição, sinônimos, outras informações

MÉTODOS DE PESQUISA

BUSCA SEQUENCIAL

Pesquisa Sequencial

- Método de pesquisa mais simples: a partir do primeiro registro, pesquise seqüencialmente até encontrar a chave procurada; então pare.
- Armazenamento de um conjunto de registros por meio de um array.

Pesquisa Sequencial

- A Busca (find):
  - Pesquisa retorna o índice do registro que contém a chave x;
  - Caso não esteja presente, o valor retornado é -1.
  - A implementação não suporta mais de um registro com uma mesma chave, pois retorna o primeiro encontrado.

Pesquisa Sequencial

```
typedef long TChave;
typedef struct {
    TChave Chave;
    /* outros componentes */
} TItem;

typedef struct {
    TItem *v;
    int n, max;
} TDicionario;
```

Pesquisa Sequencial

```
/* inicializa um dicionário */
void TDicionario_Inicio(TDicionario *t){
    t->n = 0;
    t->max = 10;
    t->v = (TItem*) malloc(sizeof(TItem)* t->max);
}

/* encontra e retorna o índice da chave x no dicionário */
int TDicionario_Find(TDicionario *t, TChave c){
    int i;
    for (i = t->n-1; i >= 0; i--)
        if (t->v[i].chave == c)
            return i;

    return -1; // retorna -1 caso a chave não seja encontrada
}
```

Pesquisa Sequencial

```
/* insere um registro no dicionário */
void TDicionario_Insere(TDicionario *t, TItem *x){
    if (t->n == t->max) {
        t->max *= 2;
        t->v = (TItem*) realloc(t->v, sizeof(TItem) * t->max);
    }

    // n é o tamanho
    t->v[t->n++] = x;
}
```

Pesquisa Sequencial

- **Análise:**
  - Pesquisa com sucesso:
    - melhor caso : C(n) = 1
    - pior caso : C(n) = n
    - caso médio: C(n) = (n + 1) / 2
  - Pesquisa sem sucesso:
    - C (n) = n + 1.
  - O algoritmo de pesquisa sequencial é a melhor escolha para o problema de pesquisa em tabelas com até 25 registros.

MÉTODOS DE PESQUISA

BUSCA BINÁRIA

Pesquisa Binária

- Pesquisa em tabela pode ser mais eficiente se registros forem mantidos em ordem
- Para saber se uma chave está presente na tabela

  1. Compare a chave com o registro que está na posição do meio da tabela.

  2. Se a chave é menor então o registro procurado está na primeira metade da tabela

  3. Se a chave é maior então o registro procurado está na segunda metade da tabela.

  4. Repita até que a chave seja encontrada ou que se constate que a chave não existe na tabela.

Exemplo de Pesquisa Binária: chave = G

	1	2	3	4	5	6	7	8
Chaves iniciais:	*A*	*B*	*C*	*D*	*E*	*F*	*G*	*H*
	A	*B*	*C*	***D***	*E*	*F*	*G*	*H*
					E	***F***	*G*	*H*
							G	*H*

Pesquisa Binária

```
typedef long TChave;
typedef struct {
    TChave Chave;
    /* outros componentes */
} TItem;

typedef struct {
    TItem *v;
    int n, max;
} TDicionario;

/* inicializa um dicionário */
void TDicionario_Inicio(TDicionario *t){
    t->n = 0;
    t->max = 10;
    t->v = (TItem*) malloc(sizeof(TItem)* t->max);
}
```

Pesquisa Binária (Recursiva)

```
/* encontra o índice da chave x no dicionário */
int TDicionario_Find(TDicionario *t, Tchave x) {
    return TDicionario_Binaria(t, 0, t->n-1, x); // t->n é o tamanho
}

/* encontra o índice da chave x no dicionário entre esq e dir */
int TDicionario_Binaria(TDicionario *t, int esq, int dir, Tchave x) {
    int meio = (esq+dir)/2;

    if (t->v[meio].chave != x && esq == dir)
        return -1;
    else if (x > t->v[meio].chave)
        return TDicionario_Binaria(t, meio+1, dir, x);
    else if (x < t->v[meio].chave)
        return TDicionario_Binaria(t, esq, meio-1, x);
    else
        return meio;
}
```

Pesquisa Binária (Iterativa)

```
/* encontra o índice da chave x no dicionário */
int TDicionario_Find(TDicionario *t, TChave c) {
    int i, esq, dir;
    if (t->n == 0) return -1;

    esq = 0;
    dir = t->n-1;
    do {
        i = (esq + dir) / 2;
        if (c > t->v[i].chave) esq = i + 1;
        else dir = i - 1;
    } while (c != t->v[i].chave && esq <= dir);

    if (c == t->v[i].chave) return i;
    else return -1;
}
```

Pesquisa Binária

- Análise
  - A cada iteração do algoritmo, o tamanho da tabela é dividido ao meio.
  - **Logo**: o número de vezes que o tamanho da tabela é dividido ao meio é cerca de log n.
  - **Ressalva**: o custo para manter a tabela ordenada é alto: a cada inserção na posição p da tabela implica no deslocamento dos registros a partir da posição p para as posições seguintes.
  - Conseqüentemente, a pesquisa binária não deve ser usada em aplicações muito dinâmicas.

Perguntas?