

Pesquisa em memória principal e secundária

Estruturas de dados II
Prof. Allan Rodrigo Leite

Pesquisa

- Ação de recuperar uma informação em um conjunto de dados
 - Como a ordenação, a operação de pesquisa é de grande importância
 - Utilização frequente e presente em diversos tipos de software
- Visa encontrar uma ou mais ocorrências de registros com chaves iguais à chave de acesso (pesquisa)
 - Esta operação pode resultar em sucesso ou não
- Exemplos
 - Procurar o contato de uma pessoa em uma agenda de contatos
 - Procurar por uma palavra em um texto ou um conjunto de textos
 - Dado um número entre 0 e 1000, adivinhar o número que se escolheu

Pesquisa

- Cada unidade de informação é mantida em uma estrutura complexa
 - Contém chave primária, além dos dados que compõem a informação

```
typedef struct registro {  
    void *dados;  
    int chave;  
} Registro;
```

- O conjunto dos registros normalmente é armazenado em tipos abstratos de dados como:
 - Estruturas lineares (listas e variações)
 - Estruturas hierárquicas (árvores binárias e variações)

Pesquisa

- Algoritmos de pesquisa
 - As rotinas que executam pesquisas devem ser eficientes
 - Isto é, executar com o menor número de iterações possíveis
 - O número de iterações depende do algoritmo de pesquisa utilizado
 - A escolha do algoritmo de pesquisa depende da:
 - Quantidade de registros envolvidos
 - Frequência das operações de inserção e de exclusão de registros
- Exemplo
 - Quando a operação de pesquisa é muito mais frequente do que a operação de inserção, deve-se minimizar o tempo de pesquisa pela ordenação dos registros
 - Este cenário é o mais comum!

Algoritmos de pesquisa

- Pesquisa em memória principal
 - Pesquisa sequencial (linear) ou com sentinela
 - Pesquisa binária
 - Pesquisa por interpolação
 - Pesquisa direta (*hashing*)
- Pesquisa em memória secundária
 - Árvore binária
 - Árvore B, B+ e B*
 - Árvores Trie e Patrícia

Pesquisa sequencial ou linear

- Método de pesquisa simples
 - Utilizado quando os dados não estão ordenados pela chave de acesso
- Princípio
 - Inicia a pesquisa pelo primeiro registro
 - Avança sequencialmente (registro por registro)
 - Termina ao alcançar o último registro
 - Com sucesso: chave pesquisada é encontrada
 - Sem sucesso: todos os registros são visitados e a chave não é encontrada

Pesquisa sequencial ou linear

```
int pesquisaSequencial(int chave, int v[], int n = 5) {  
    int i;  
    for (i = 0; i < n; i++) {  
        if (v[i] == chave) {  
            return i;  
        }  
    }  
    return -1; //índice inválido  
}
```

Pesquisa sequencial ou linear

- Análise de complexidade
 - Para uma pesquisa com sucesso, temos:
 - 1 iteração no melhor caso
 - n iterações no pior caso
 - $(n + 1) / 2$ iterações no caso médio
 - Para uma pesquisa sem sucesso, temos:
 - $n + 1$ iterações
 - O número total de comparações são:
 - Melhor caso: 2
 - Pior caso: $(n + 1) + n = 2n + 1$
 - Médio caso: $(2n + 1 + 2) / 2 = (2n + 3) / 2$
 - Assintoticamente, o algoritmo é $O(n)$ em complexidade de tempo

Pesquisa sequencial com sentinela

- O algoritmo de pesquisa sequencial pode ser acelerado
 - Atribui-se a chave de pesquisa ao registro contido na posição $n + 1$
- Com isso, este registro fictício passa funcionar como sentinela
 - Mesmo no pior caso, a chave será encontrada na posição $n + 1$
 - Se o elemento alvo for encontrado em uma posição anterior a $n + 1$, significa que o elemento está na lista
 - No entanto, se o elemento alvo só for encontrado na posição $n + 1$, isto significa que ele não está presente na lista

Pesquisa sequencial com sentinela

- Objetivo do registro sentinela
 - Usar o elemento alvo como indicação que a lista não tem mais registros a serem lidos
 - Eliminar a necessidade de cada passo no laço testar se já chegou ao final da lista

Pesquisa sequencial com sentinela

```
int pesquisaSequencialSentinela(int chave, int v[], int n) {  
    int i = 0;  
    v[n] = chave; //A última posição do vetor possui o sentinela  
    while (v[i] != chave) {  
        i++;  
    }  
    if (i < n) return i;  
    return -1; //Índice inválido  
}
```

Pesquisa binária

- A pesquisa em uma tabela pode ser mais eficiente se os registros forem mantidos em ordem
- Princípio
 - Similar ao utilizado ao procurar uma palavra em um dicionário
 - Compara-se a chave procurada com a chave do registro no conjunto
 - Esta comparação indica
 - A chave foi encontrada, ou em qual das metades a pesquisa deve prosseguir, segundo este mesmo princípio

Pesquisa binária

- Algoritmo básico
 - Comparar a chave de acesso (pesquisa) com o registro central da lista
 - Se a chave for menor, o registro alvo está na primeira metade da lista
 - Se a chave for maior, o registro alvo está na segunda metade da lista
- Repita o processo até que a chave seja encontrada
 - Quando existe apenas um registro e a chave é diferente da procurada, isto significa uma pesquisa sem sucesso

Pesquisa binária

```
int pesquisaBinaria(int chave, int v[], int n) {  
    int inicio = 0, meio, fim = n - 1;  
    while (inicio <= fim) {  
        meio = (inicio + fim) / 2;  
        if (chave < v[meio]) {  
            fim = meio - 1;  
        } else if (chave > v[meio]) {  
            inicio = meio + 1;  
        } else {  
            return meio;  
        }  
    }  
    return -1; //Índice impossível  
}
```

Pesquisa binária

- Análise de complexidade
 - O número de registros pesquisados é reduzido à metade a cada iteração:
 - $n, n / 2, n / 4, n / 8, \dots, n / 2^k$
 - Queremos que $n / 2^k \leq 1$, logo $k \geq \log_2 n$
- A chave pesquisada deve ser comparada com o último elemento restante, assim, o número máximo de comparações é $1 + \log_2 N$
 - Assintoticamente, o algoritmo é $O(\log_2 n)$ em complexidade de tempo

Pesquisa binária

- Análise de complexidade
 - Exemplos
 - Lista com 16 registros equivale a 4 iterações
 - Lista com 1024 registros equivale a 10 iterações
 - Lista com 1000000 registros equivale a 20 iterações
 - Desempenho muito superior em relação a pesquisa sequencial
 - $\log_2 n$ cresce muito devagar com o aumento de n

Pesquisa binária recursiva

```
int pesquisaBinariaRecursiva(int chave, int v[], int ini, int fim) {  
    int meio = (ini + fim) / 2;  
    if (ini > fim)  
        return -1;  
    if (chave == v[meio])  
        return meio;  
    else if (chave < v[meio])  
        return pesquisaBinariaRecursiva(chave, v, ini, meio - 1);  
    else  
        return pesquisaBinariaRecursiva(chave, v, meio + 1, fim);  
}
```

Pesquisa por interpolação

- A pesquisa por interpolação pode ser mais eficiente do que a binária
 - Quando as chaves estiverem uniformemente distribuídas dentro da lista
- O algoritmo é o mesmo da pesquisa binária, adotando-se uma outra estratégia para calcular o valor da variável meio
 - Que neste caso não será obrigatoriamente o meio da lista

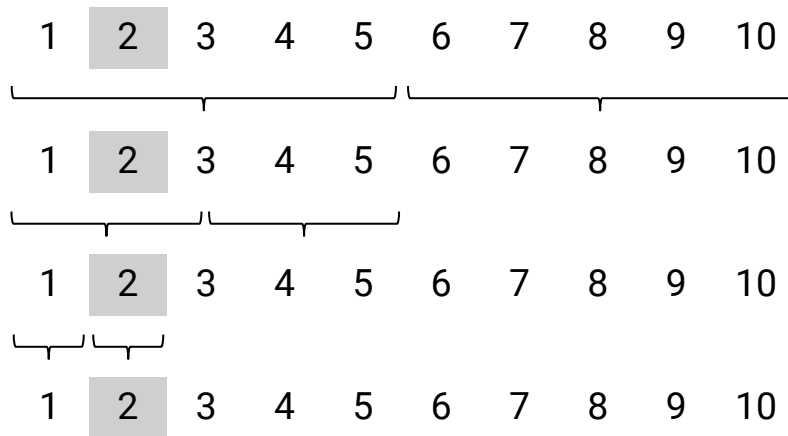
$$\text{meio} = \text{ini} + \frac{(\text{fim} - \text{ini}) \times (\text{chave} - v[\text{ini}])}{v[\text{fim}] - v[\text{ini}]}$$

Pesquisa por interpolação

```
int pesquisaInterpolacao(int chave, int v[], int n) {  
    int ini = 0, meio, fim = n - 1;  
    while (ini <= fim) {  
        meio = ini + ((fim - ini) * (chave - v[ini])) / (v[fim] - v[ini]);  
        if (chave < v[meio]) {  
            fim = meio - 1;  
        } else if (chave > v[meio]) {  
            ini = meio + 1;  
        } else {  
            return meio;  
        }  
    }  
    return -1; //Índice impossível  
}
```

Pesquisa por interpolação

- Exemplo pesquisa binária
 - Número de registros: 10
 - Chave de acesso (pesquisa): 2



Pesquisa por interpolação

- Exemplo pesquisa por interpolação
 - Número de registros: 10
 - Chave de acesso (pesquisa): 2

1 2 3 4 5 6 7 8 9 10

$$\text{meio} = \frac{\text{ini} + (\text{fim} - \text{ini}) \times (\text{chave} - v[\text{ini}])}{v[\text{fim}] - v[\text{ini}]}$$



$$\text{meio} = \frac{0 + (9 - 0) \times (2 - 1)}{10 - 1}$$



meio = 1

1 2 3 4 5 6 7 8 9 10



Pesquisa por interpolação

- Análise de complexidade
 - Requer $\log_2(\log_2 n)$ comparações se as chaves estiverem uniformemente distribuídas
 - Entretanto, caso as chaves não estiverem uniformemente distribuídas, o método degrada sua eficiência e torna-se ruim
 - No pior caso se compara com a busca sequencial
 - Em situações práticas as chaves tendem a se aglomerar em torno de determinados valores e não são uniformemente distribuídas
 - Por exemplo, agenda de contatos

Pesquisa em memória principal e secundária

Estruturas de dados II
Prof. Allan Rodrigo Leite