Notas de Aula - AED1 – Algoritmos de Pesquisa (parte 2)
Prof. Jefferson T. Oliva

Na aula passada foi formalizada a busca e apresentados alguns métodos, como:
- Busca sequencial
→ Para dados desordenados
→ Para dados ordenados
→ Recuperação recorrente de registros: mover-para-frente e transposição
- Busca sequencial indexada

Hoje iremos abordar outras abordagens de busca: binária, interpolação e árvores de busca.

Busca Binária

Os dados devem estar ordenados em um arranjo.

A busca começa no meio da tabela
- Se igual, busca bem-sucedida
- Se menor, busca-se na metade inferior do arranjo
- Se maior, busca-se na metade superior do arranjo

Em cada passo, o tamanho do arranjo em que se busca é dividido por 2.

É uma técnica de divisão e conquista.

Ver slides entre 5 e 8.

Implementação:

```
static int busca_bin(int x, int v[], int ini, int fim){
        int meio;

        if ((ini == fim) && (x == v[ini]))
                return ini;
        else if (ini < fim){
                meio = (ini + fim) / 2;
                if (x == v[meio])
                        return meio;
                else if (x < v[meio])
                        return busca_bin(x, v, ini, meio - 1);
                else
                        return busca_bin(x, v, meio + 1, fim);
        else
                return -1;
}
int busca_binaria(int x, int v, int n){
        return busca_bin(v, 0, n - 1);
}
```

Complexidade: O(log n)
- Em cada comparação, o número de candidatos é dividido pela metade

Vantagens: eficiência da busca e fácil implementação.

Desvantagens:
- Nem todo arranjo está ordenado
- Exige o uso de um arranjo para armazenar os dados, ou seja, não é possível implementar esse método de busca em listas encadeadas. A solução é utilizar árvore binária de busca (outra estrutura de dados) para o armazenamento de dados.
- Não é possível implementar busca binária em lista encadeada, pois nessa estrutura não possui acesso por meio de índices. Alternativa: usar árvore binária de busca.

A busca binária pode ser usada com a organização de tabela sequencial indexada.
- Em vez de pesquisar o índice sequencialmente, pode-se usar uma busca binária. Também, a busca binária pode ser aplicada na região da tabela que abrange o índice.

Busca por Interpolação

Derivação da busca binária.

Se as chaves estiverem uniformemente distribuídas, a busca por interpolação pode ainda mais eficiente que a busca binária.

Com chaves uniformemente distribuídas, pode-se esperar que x esteja aproximadamente na posição:

*meio = ini + (fim − ini) * (x – v[ini]) / (v[fim] – v[ini])* ==> ***Ver interpolação linear***
ini e fim são redefinidos da mesma forma em relação à busca binária.

Ver slides 14 e 15. Experimente buscar 86, 37 e 57.

Implementação:

```
static int busca_in(int x, int v[], int ini, int fim){
        int meio;

        if ((ini == fim) && (x == v[ini]))
                return ini;
        else if (ini < fim){
                meio = ini + (fim - ini) * (x - v[ini]) / (v[fim] - v[ini]);
                if (x == v[meio])
                        return meio;
                else if (x < v[meio])
                        return busca_in(x, v, ini, meio - 1);
                else
                        return busca_in(x, v, meio + 1, fim);
        }else
                return -1;
}
```

```
int busca_interpolacao(int x, int v[], int n){
        return busca_in(x, v, 0, n - 1);
}
```

Complexidade:
- Se as chaves estiverem uniformemente distribuídas, a complexidade será de O(log(log n)).
- Para chaves uniformemente distribuídas, raramente precisará mais que log(log n) comparações.
- Se as chaves não estiverem uniformemente distribuídas, a busca por interpolação pode ser tão ruim quanto uma busca sequencial. Um exemplo seria a busca pela chave 92 no seguinte arranjo: {12, 25, 33, 37, 48, 57, 86, 1000000}, onde apenas o parâmetro "*ini*" seria atualizado, mas com apenas incremento de um, ou seja, assumiria os seguintes valores: 0, 1, 2, 3, 4, 5, 6, 7.

No caso de uma lista telefônica, se quisermos procurar o número de "João". Nesse situação a busca interpolada apresenta melhor desempenho uma vez que podemos pular diretamente para o espaço de memória onde os nomes começam com 'J'.

Desvantagem: em situações práticas, as chaves tendem a se aglomerar em torno de determinados valores e não são uniformemente distribuídas. Exemplo: há uma quantidade maior de nomes começando com "S" do que com "Q".

Árvores de Busca

Nota: árvore é tema da disciplina "Algoritmos e Estrutura de Dados 2 (AE23CP)"

Árvore é uma estrutura de dados muito eficiente para armazenamento de informação (incluindo as que devem ser organizadas de forma hierárquica).

Como as árvores armazenam informações de forma hierárquica, a sua estrutura de dados não é linear.

Aplicações: sistemas de arquivos, processamento de língua natural, inteligência artificial, etc.

Árvore é um conjunto finito de nós
- Existe um nó raiz r com zero ou mais sub-árvores
- Os nós internos são filhos de r
- Cada sub-árvore também possui um nó raiz, que é descendente (nó filho) de r
- Nós folhas são os nós que não possuem filhos

Tipos de árvores:
- Árvore binária
- Árvore binária de busca
- AVL
- Árvore trie
- Árvore PATRICIA
- Árvore vermelha-preta (rubro-negra)
- Árvore B
- Etc.

Árvores Binárias de Busca

Árvore binária
- Árvore em que cada nó contém um ou dois filhos
- As sub-árvores também contém entre um e dois nós, exceto se são nós-folhas
- Podemos representar uma árvore por meio de um registro (struct) de três campos:
→ Informação
→ Sub-árvore esquerda
→ Sub-árvore direita

Estrutura para representação de um nó de uma árvore binária de busca:

```
typedef struct Node Node;

struct Node{
        int item;
        Node *right;
        Node *left;
};
```

Árvore binária de busca é uma árvore binária com propriedades de ordenação:
- Todos os nós com chaves menores que a da raiz ficam do lado esquerdo
- Todos os nós com chaves maiores que a da raiz ficam do lado direito

Principais operações em uma árvore binária de busca:
- Pesquisa
- Inserção
- Remoção

A busca é iniciada pela raiz da árvore
- Caso o valor seja encontrado, o nó é retornado
- Caso contrário:
→ É realizada uma chamada recursiva para a sub-árvore esquerda se o valor procurado é menor que o item verificado
→ É realizada uma chamada recursiva para a sub-árvore direita se o valor procurado é maior

Implementação da busca:

```
Node* search(Node* tree, int value){
        if (tree != NULL)
                if (tree->item == value)
                        return tree;
                else if (tree->item < value)
                        return search(tree->left, value);
                else
                        return search(tree->right, value);
        else
                return NULL;
}
```

Complexidade da operação de busca
- Melhor caso: O(1)
- Caso médio: O(log n)
- Pior caso: O(n)

A complexidade das operações em árvores é diretamente relacionada com o balanceamento da árvore (será visto em "Algoritmos e Estrutura de Dados 2")

Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Horowitz, E., Sahni, S. Rajasekaran, S. Computer Algorithms. Computer Science Press, 1998.

Rosa, J. L. G. Métodos de Busca. SCE-181 - Introdução à Ciência da Computação II. Slides. Ciência de Computação. ICMC/USP, 2018.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007.