Algoritmos de Busca Sequencial e Binária

Profª. Rose Yuri Shimizu

Algoritmos de Busca

- Objetivo:
  - Recuperação de informação a partir de dados previamente armazenados
  - A informação é dividida em registros que possuem uma chave
    - Chave de identificação: usadas nas buscas
- Conjunto de registros formam as chamadas tabela de símbolos, map, dicionário, vetor associativo
  - Coleção de pares de chave-valor
  - Mecanismo abstrato para armazenar informações que podem ser acessadas através de uma chave
  - Exemplos:
    - **Busca sequencial**
    - **Busca binária**
    - **Árvores de busca**
    - Árvore B (e variantes): comumente usadas quando o vetor associativo é muito grande para ser armazenado completamente em memória principal (ex. banco de dados)
    - Pesquisa digital
    - Tabelas hash

chave	valor
www.ebay.com	66.135.192.87
www.princeton.edu	128.112.128.15
www.cs.princeton.edu	128.112.136.35
www.harvard.edu	128.103.60.24
www.yale.edu	130.132.51.8
www.cnn.com	64.236.16.20
www.google.com	216.239.41.99
www.nytimes.com	199.239.136.200
www.apple.com	17.112.152.32
www.slashdot.org	66.35.250.151
www.espn.com	199.181.135.201
www.weather.com	63.111.66.11
www.yahoo.com	216.109.118.65
...	...
www.ime.usp.br	143.107.45.37

Busca sequencial

- Método de pesquisa mais simples
- A partir do primeiro registro, pesquisa sequencialmente até encontrar a chave procurada
- Os registros estão organizados em uma estrutura de dados do tipo array ou lista encadeada

Busca sequencial

- Complexidade para busca (dados não-ordenados)
  - Melhor caso:

Busca sequencial

- Complexidade para busca (dados não-ordenados)
  - Melhor caso: $O(1)$
    - ⋆ Encontra na primeira posição
  - Caso médio:

Busca sequencial

- Complexidade para busca (dados não-ordenados)
  - Melhor caso: $O(1)$
    - ⋆ Encontra na primeira posição
  - Caso médio: $O(n) = (n+1)/2$
    - ⋆ Necessita checar cerca de metade dos registros
  - Pior caso:

Busca sequencial

- Complexidade para busca (dados não-ordenados)
  - Melhor caso: $O(1)$
    - Encontra na primeira posição
  - Caso médio: $O(n) = (n+1)/2$
    - Necessita checar cerca de metade dos registros
  - Pior caso: $O(n)$
    - Necessita checar todas as chaves, o valor buscado se encontra na última posição
  - Pesquisa sem sucesso: $O(n) = n+1$

Busca sequencial

- Listas encadeadas:
  - Inserção e remoção:

Busca sequencial

- Listas encadeadas:
  - Inserção e remoção: $O(1)$
  - Busca:

Busca sequencial

- Listas encadeadas:
  - Inserção e remoção: $O(1)$
  - Busca: $O(n)$
- Vetores não-ordenados:
  - Inserção e remoção:

Busca sequencial

- Listas encadeadas:
  - Inserção e remoção: $O(1)$
  - Busca: $O(n)$
- Vetores não-ordenados:
  - Inserção e remoção: $O(1)$
  - Busca:

Busca sequencial

- Listas encadeadas:
  - Inserção e remoção: $O(1)$
  - Busca: $O(n)$
- Vetores não-ordenados:
  - Inserção e remoção: $O(1)$
  - Busca: $O(n)$
- Vetores ordenados:
  - Busca:

Busca sequencial

- Listas encadeadas:
  - Inserção e remoção: $O(1)$
  - Busca: $O(n)$
- Vetores não-ordenados:
  - Inserção e remoção: $O(1)$
  - Busca: $O(n)$
- Vetores ordenados:
  - Busca: $O(lgn)$
  - Inserção e remoção:

Busca sequencial

- Listas encadeadas:
  - Inserção e remoção: $O(1)$
  - Busca: $O(n)$
- Vetores não-ordenados:
  - Inserção e remoção: $O(1)$
  - Busca: $O(n)$
- Vetores ordenados:
  - Busca: $O(lgn)$
  - Inserção e remoção: $O(n)$

Busca binária

- Método eficiente
- Para conjunto de dados ordenados
- Primeira comparação: elemento do meio
- Paradigma da divisão e conquista

```
procurar 18
índices ... 2   3   4   5   6    7    8    9    10   11 ...
vetor  [... 6 | 7 | 8 | 9 | 10 | 11 | 12 | 15 | 18 | 20 ...]
```

Busca binária

- Dividir o vetor no meio:
  - Intervalo de elementos = diferença entre os índices
  - Metade do intervalo + o deslocamento a esquerda
  - $meio = (11 - 2)/2 = 4 + 2 = 6$

Busca binária

- Dividir o vetor no meio:
  - Intervalo de elementos = diferença entre os índices
  - Metade do intervalo + o deslocamento a esquerda
  - $meio = (11 - 2)/2 = 4 + 2 = 6$
- Comparar como elemento central

```
  2   3   4   5   6    7    8    9    10   11     meio = 6
[ 6 | 7 | 8 | 9 | 10 | 11 | 12 | 15 | 18 | 20 ]  v[6] == 18??
```

Busca binária

- Procurar à esquerda?
- $18 < v[6]$??

```
  2   3   4   5   6   7   8   9   10  11    meio = 6
[ 6 | 7 | 8 | 9 | 10 | 11 | 12 | 15 | 18 | 20 ]  18 < v[6]??
```

Busca binária

- Procurar à direita?
- $18 > v[6]$??

```
  2   3   4   5   6   7   8   9   10  11     meio = 6
[ 6 | 7 | 8 | 9 | 10 | 11 | 12 | 15 | 18 | 20 ]  18 > v[6]??
```

Busca binária

- Recursivo: dividir e comparar
- $meio = (11 - 7)/2 = 2 + 7 = 9$

```
  7     8     9     10    11      meio = 9
[ 11  |  12  |  15  |  18  |  20 ]  v[9] == 18??
```

Busca binária

- Recursivo: procurar à esquerda?
- $18 < v[9]$??

```
  7     8     9     10    11      meio = 9
[ 11  |  12  |  15  |  18  |  20 ]  18 < v[9]??
```

Busca binária

- Recursivo: procurar à direita?
- $18 > v[9]$??

```
  7     8     9     10    11      meio = 9
[ 11  |  12  |  15  |  18  |  20 ]  18 > v[9]??
```

Busca binária

- Recursivo: dividir e comparar
- $meio = (11 - 10)/2 = 0 + 10 = 10$
- v[10] == 18?? `return 10`

```
  7     8     9     10    11      meio = 10
[ 11  |  12  |  15  |  18  |  20 ]  v[10] == 18??
```

```c
//macro: campo chave do dado A
#define key(A) (A.chave)

typedef int Key;
typedef struct data Item;
struct data { Key chave; char info[100]; };

int binary_search(Item *v, int l, int r, Key k) {
    //condição de parada
```

```c
//macro: campo chave do dado A
#define key(A) (A.chave)

typedef int Key;
typedef struct data Item;
struct data { Key chave; char info[100]; };

int binary_search(Item *v, int l, int r, Key k) {
    //condição de parada
    if(l > r) return -1;

    //calcular o índice central
```

```c
//macro: campo chave do dado A
#define key(A) (A.chave)

typedef int Key;
typedef struct data Item;
struct data { Key chave; char info[100]; };

int binary_search(Item *v, int l, int r, Key k) {
    //condição de parada
    if(l > r) return -1;

    //calcular o índice central
    int m = (l+r)/2; //l+(r-l)/2

    //comparar k com o elemento central
```

```c
//macro: campo chave do dado A
#define key(A) (A.chave)

typedef int Key;
typedef struct data Item;
struct data { Key chave; char info[100]; };

int binary_search(Item *v, int l, int r, Key k) {
    //condição de parada
    if(l > r) return -1;

    //calcular o índice central
    int m = (l+r)/2; //l+(r-l)/2

    //comparar k com o elemento central
    if(k == key(v[m])) return m;

    //procurar à esquerda?
```

```c
//macro: campo chave do dado A
#define key(A) (A.chave)

typedef int Key;
typedef struct data Item;
struct data { Key chave; char info[100]; };

int binary_search(Item *v, int l, int r, Key k) {
    //condição de parada
    if(l > r) return -1;

    //calcular o índice central
    int m = (l+r)/2; //l+(r-l)/2

    //comparar k com o elemento central
    if(k == key(v[m])) return m;

    //procurar à esquerda?
    if(k < key(v[m]))
        //intervalo à esquerda?
```

```c
//macro: campo chave do dado A
#define key(A) (A.chave)

typedef int Key;
typedef struct data Item;
struct data { Key chave; char info[100]; };

int binary_search(Item *v, int l, int r, Key k) {
    //condição de parada
    if(l > r) return -1;

    //calcular o índice central
    int m = (l+r)/2; //l+(r-l)/2

    //comparar k com o elemento central
    if(k == key(v[m])) return m;

    //procurar à esquerda?
    if(k < key(v[m]))
        //intervalo à esquerda?
        return binary_search(v, l, m-1, k);

    //senão procurar à direita; intervalo?
```

```c
//macro: campo chave do dado A
#define key(A) (A.chave)

typedef int Key;
typedef struct data Item;
struct data { Key chave; char info[100]; };

int binary_search(Item *v, int l, int r, Key k) {
    //condição de parada
    if(l > r) return -1;

    //calcular o índice central
    int m = (l+r)/2; //l+(r-l)/2

    //comparar k com o elemento central
    if(k == key(v[m])) return m;

    //procurar à esquerda?
    if(k < key(v[m]))
        //intervalo à esquerda?
        return binary_search(v, l, m-1, k);

    //senão procurar à direita; intervalo?
    return binary_search(v, m+1, r, k);
}
```

```c
int binary_search(Item *v, int l, int r, Key k) {
    if(l > r) return -1;
    int m = (l+r)/2;

    if(k == key(v[m]))
        return m;

    if(k < key(v[m]))
        return binary_search(v, l, m-1, k);

    return binary_search(v, m+1, r, k);
}
```

v[] = {4, 5, 6, 7, 8, 9, 10, 11, 12, 15, 18, 20}

```
binary_search(v, 2, 11, 18)
|     m = (2+11)/2 = 6
|     binary_search(v, 6+1, 11, 18)
|     |     m = (7+11)/2 = 9
|     |     binary_search(v, 9+1, 11, 18)
|     |     |     m = (10+11)/2 = 10
|     |     |     return 10
|     |     return 10
|     return 10
return 10
```

Busca binária

- Função custo da recorrência
  - A cada iteração do algoritmo, o tamanho do vetor é dividido ao meio
  - Complexidade: até $\lfloor lgN \rfloor + 1$ comparações (acerto ou falha)

$$\begin{aligned} f(n) &= f(n/2) + 1 \\ &= f(n/4) + 2 \\ &= f(n/8) + 3 \\ &= f(n/2^k) + k, 2^k = n \rightarrow k.log_2 2 = log_2 n \rightarrow k = log_2 n \\ &= f(1) + log_2 n \end{aligned}$$

- Existe o custo para manter o vetor ordenado
  - Algoritmos de ordenação

```c
#define key(A) (A)
typedef int Item;
typedef int (*Compare)(const void *, const void *);

int check(const void* v1, const void* v2){
    const Item *a = v1;
    const Item *b = v2;
    return key(*a) - key(*b); //0   v1 = v2
                              //-   v1 < v2
                              //+   v1 > v2
}

int binary_search(Item *v, int l, int r, Item k, Compare comp)
{
    if(l > r) return -1;
    int m = (l+r)/2; //l+(r-l)/2

    if(comp(&k, &v[m])==0)
        return m;

    if(comp(&k, &v[m])<0)
        return binary_search(v, l, m-1, k, comp);

    return binary_search(v, m+1, r, k, comp);
}
```

```
26
27 int main(int argc, char *argv[]) {
28     int n, x;
29     scanf("%d", &n);
30
31     int *v = malloc(n*sizeof(int));
32     for(int i=0; i<n; i++) {
33         scanf("%d", &v[i]);
34     }
35
36     scanf("%d", &x);
37     binary_search(v, 0, n-1, x, check);
38 }
39
```

```c
//Standard C library - bsearch : stdlib.h (man bsearch)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct mi {
    int           nr;
    const char   *name;
};

static struct mi  months[] = {
    { 1, "jan" }, { 2, "feb" }, { 3, "mar" }, { 4, "apr" },
    { 5, "may" }, { 6, "jun" }, { 7, "jul" }, { 8, "aug" },
    { 9, "sep" }, {10, "oct" }, {11, "nov" }, {12, "dec" }
};

static int comp(const void *m1, const void *m2) {
    const struct mi *mi1 = m1; //constante: somente leitura
    const struct mi *mi2 = m2; //constante: somente leitura
    return strcmp(mi1->name, mi2->name); //0     m1 == m2
                                         //-     m1 < m2
                                         //+     m1 > m2
}

```

```c
int main(int argc, char *argv[])
{
    qsort(months, 12, sizeof(struct mi), comp);

    for (int i = 1; i < argc; i++) {
        struct mi key;
        struct mi *res;

        key.name = argv[i];
        res = bsearch(&key, months, 12, sizeof(struct mi), comp);
        if (res == NULL)
            printf("'%s': unknown month\n", argv[i]);
        else
            printf("%s: month #%d\n", res->name, res->nr);
    }

    exit(EXIT_SUCCESS);
}
```