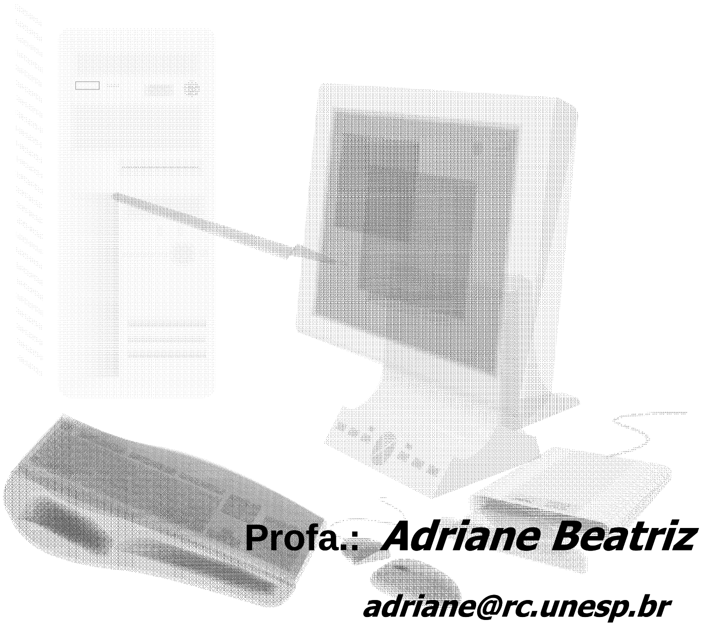
Introdução à Computação II



Busca

Profa.: *Adriane Beatriz de Souza Serapião*

adriane@rc.unesp.br

Array

- Estrutura de dados fundamental:
  - Diversas outras estruturas são implementadas usando *arrays*.
  - Em última análise, a própria memória é um *array*.
- Problemas:
  - Alocação de memória:
    - Quantas posições deve ter o *array* para uma dada aplicação?
    - O que fazer se precisarmos mais?
  - Como inserir um novo dado entre o *k*-ésimo e o (*k*+1)-ésimo elemento?
  - Como remover o *k*-ésimo elemento?

Busca Sequencial em array

- O problema de *busca* ou *pesquisa* de informação é um dos assuntos mais bem estudados em computação.
- A estrutura de dados *array* é adequada para armazenar informação na memória principal de um computador.
- Portanto, eventualmente, gostaríamos de pesquisar a existência de determinado item em um *array*.
- A *busca sequencial* ou *linear* é a forma de busca mais simples que existe.

Busca Sequencial em array

- Problema
  - Dado um *array* com n componentes, queremos determinar se um dado item x se encontra armazenado neste *array*.
- Análise
  - Para fixar as idéias, vamos supor que o array seja denominado A e o seu tipo seja:

    T **arranjo[n];**

    onde T é um tipo qualquer

Busca Sequencial em array

- **Análise (cont.)**
  - **Para pesquisarmos x em um *array* A, basta compará-lo com os componentes do *array*, a partir do primeiro componente e, assim sequencialmente, até encontrar o item x ou chegar ao final do *array*.**
  - **Se o *array* A contém o elemento x, dizemos que a busca teve *sucesso;* caso contrário, dizemos que a pesquisa teve *fracasso.***

Busca Sequencial em array

- **Algoritmo:**
  - **Devido à simplicidade do algoritmo, vamos apresentá-lo também como uma função que determina o índice i tal que $A[i] = x$, se existir; caso contrário, i é 0.**
  - **Além disso, a própria função retorna um inteiro que indica busca com sucesso ou fracasso.**

Busca Sequencial em array

- **Algoritmo:**

func ***buscasequencial*** **(*x*, *A*[1..*n*], *n*)**
{
  achei ← **falso**
  i ← **-1**
  enquanto ***i*** **<=** ***n*** e não ***achei*** fazer
  {
    i ← ***i*** **+ 1**
    se ***A*[*i*]** **=** ***x*** então ***achei*** ← ***verdadeiro***
  }
  se ***achei*** então reportar **(*i*)** senão reportar**(-1)**
}

Busca Sequencial em array

- **Programa em C:**

```
int buscaSeq ( int x, arranjo A, int n )
{
   int i = -1;
   bool achei = false;
   while ((!achei) && (i <= n))
   {
      i++;
      achei = (x == A[i]);
   };
   if (!achei) i = -1;
   return i;
}
```

Busca Sequencial em array com sentinela

- Busca sequencial simples testa três condições dentro do laço.
- É possível alterar o algoritmo para empregar apenas um teste no laço de repetição.
- Busca com *Sentinela (dummy)*:
  - Usa-se uma posição a mais no final do *array* (*A*[*n+1*]) que é carregada com uma cópia do dado sendo buscado (*x*).
  - Como é garantido que *x* será encontrado, não é preciso se precaver contra o acesso de uma posição *i* não existente.

Busca Sequencial em array com sentinela

- Melhoria da eficiência da busca sequencial:
  - O algoritmo de busca sequencial pode ser melhorado no seu tempo de execução ao colocarmos o elemento a ser pesquisado no final do *array* (sentinela).
  - Desta forma, podemos simplificar a condição composta do comando *while* para uma condição simples.

Busca Sequencial em array com sentinela

- **Algoritmo:**

```
func busca_com_sentinela (x, n, A[1..n+1])
{
  A[n+1] ← x
  i ← 0
  enquanto (A [i] ≠ x) fazer {
    i ← i + 1
  }
  se i < n então
    reportar (i)      % encontrado
  senão
    reportar(-1)      % não encontrado
}
```


Busca Sequencial em array com sentinela

- **Busca sequencial melhorada em C:**

```c
int buscaSeqSent ( int x, arranjo A, int n )
{
   int i = 0;
   A[n+1] = x;
   while ((x != A[i]) i++;
   if (i == n+1) i = -1;
   return i;
}
```

Busca Sequencial em array

- Análise da eficiência da busca sequencial:
  - A operação preponderante no algoritmo de busca sequencial é a comparação $x = A[i]$.
  - Em uma pesquisa com sucesso, o *pior caso* ocorre quando o elemento procurado se encontra na última posição do *array* $A[n]$; neste caso, são feitas n comparações.

Busca Sequencial em array

- Análise da eficiência da busca sequencial (cont.):
  - O *melhor caso* ocorre quando o elemento procurado se encontra na primeira posição do *array*; neste caso, é feita apenas uma comparação.
  - Usando probabilidade, podemos mostrar que, no *caso médio,* são feitas $n/2$ comparações em uma busca com sucesso.
  - Em uma busca com fracasso, são sempre realizadas n comparações.

Busca Sequencial – Análise

- A análise de pior caso de ambos os algoritmos para busca sequencial são obviamente $O(n)$, embora a busca com sentinela seja mais rápida.
- A análise de caso médio requer que estipulemos um modelo probabilístico para as entradas. Sejam:
  - E_1 , E_2 , … E_n as entradas v correspondentes às situações onde x=A[1], x=A[2], … x=A[n]
  - E_n entradas x tais que x <u>não</u> pertence ao *array* A
  - $p(E_i)$ a probabilidade da entrada E_i ocorrer
  - $t(E_i)$ a complexidade do algoritmo quando recebe a entrada E_i
- Assumimos:
  - $p(E_i) = q/n$ para $i < n$
  - $p(E_n) = 1-q$

Busca Sequencial – Análise de Caso Médio

- Se admitirmos $t(E_i) = i$, então temos como complexidade média:

$$\sum_{i=1}^{n} p(E_i)\, t(E_i) = (n+1)(1-q) + \frac{q}{n}\left(\sum_{i=1}^{n} i\right)$$
$$= (n+1)(1-q) + \frac{q}{n}\,\frac{n(n+1)}{2}$$
$$= \frac{(n+1)(2-q)}{2}$$

- Para q=1/2, temos complexidade média ≈ 3n/4
- Para q=0, temos complexidade média ≈ n
- Para q=1, temos complexidade média ≈ n/2

Busca Sequencial em Listas Encadeadas

- **Vamos usar a seguinte notação:**
  - ***Nulo*** **: elo (lista) nulo**
  - ***L^*** **: denota primeiro nó da lista *L*. Definido apenas se *L* não é nula**
  - ***No.Elemento*****: denota o elemento armazenado em *No***
  - ***No.Elo*****: denota o elo armazenado em No**

- **Versão iterativa:**

```
proc PesquisaLista (Lista L, Valor v)
{
   tmp ← L
   enquanto tmp ≠ Nulo fazer
   {
      se tmp^.Elemento = v então
         retornar verdadeiro
      senão
         tmp ← tmp^.Elo
   }
   retornar falso
}
```


Busca Sequencial em Listas Encadeadas

```
struct lista
{
   int info;
   struct lista *prox;
};
typedef struct lista Lista;

bool PesquisaLista (Lista* L; int x);
{
  Lista* p;
  bool achou = false;
  p = L;
  while ((p != NULL) && !achou)
  {
    if (p->info == x) achou = true;
    p = p->prox;
  };
  return achou;
}
```

Busca Sequencial em Listas Encadeadas

- **Versão recursiva*:***

```
proc PesquisaListaRec (Lista L, Valor v)
{
   se L = Nulo então
      retornar falso
   senão
      se L^.Elemento = v então
         retornar verdadeiro
      senão
         retornar PesquisaListaRec (L^.Elo, v)
}
```

- **Implemente o programa recursivo em C.**

Busca Binária em array

- **Problema:**
  - **Suponhamos que agora o *array* A esteja ordenado crescentemente.**
  - **Isto é, $A[i] \leq A[j]$, se $i < j$.**

- **Análise:**
  - **Podemos lançar mão do fato de que o *array* A está ordenado e projetar um método de busca extremamente eficiente.**

Busca Binária em array

- Análise (cont.)
  - O método de *busca binária* pode ser descrito assim:
    - Primeiro, selecionamos o elemento do meio do *array* com índice $meio = \lfloor (1 + n)/2 \rfloor$.
    - Se $x = A[meio]$, o elemento procurado foi encontrado.
    - Se $x > A[meio]$, nenhum componente da primeira metade do *array* pode ser igual a x (por quê?); logo a continuação da busca pode se restringir à segunda metade do *array*.

Busca Binária em array

- Análise (cont.)
  - Se $x < A[meio]$, analogamente, a busca pode se restringir à primeira metade do *array*.
  - Em cada uma dessas metades, aplicamos o mesmo método novamente até que x seja encontrado — neste caso, busca com sucesso — ou as metades do *sub-array* em questão se tornam vazias — neste caso, busca com fracasso.

Busca Binária em array

- **Algoritmo:**

```
func  busca_binária  (x,  n,  A [1 .. n])
{
   inf  ←  1          % limite inferior
   sup  ←  n          % limite superior
   enquanto  inf  ≤  sup  fazer  {
      meio  ←  (inf + sup)  div  2
      se  A [meio]  <  x  então
         inf  ←  meio + 1
      senão  se  A[meio]  >  x  então
            sup  ←  meio – 1
         senão
            retornar  (meio)      % Valor encontrado
   }
   retornar  (-1)                 % Valor não encontrado
}
```


Busca Binária em array

- **Programa em C (versão iterativa):**

```
int buscaBin( int x, arranjo A, int n )
{
   int meio;
   int inf = 0;
   int sup = n;
   bool achei = false;
   while ((!achei) && (inf <= sup))
   {
      meio = (inf + sup)/2;
      if (x == A[meio]) achei = true;
      else if (x > A[meio]) inf = meio + 1;
           else sup = meio – 1;
   };
   if (!achei) meio = -1;
   return meio;
}
```

Busca Binária em array

- **Programa em C (versão recursiva):**

```
Int buscaBinRec(int x, arranjo A, int inf, int sup)
 {
  int meio;
  if (inf > sup) return -1; // não achou
  else
  {
    meio = (inf+sup)/2;
    if (x < A[meio])
       return BuscaBin (x, A, inf, meio-1);
    else if (x > A[meio])
            return BuscaBin(x, A, meio+1, sup);
         else return meio;
  };
} // BuscaBin
```


Busca Binária em array

- **Análise da eficiência da busca binária:**
  - **Certamente, a busca binária é muito mais eficiente que a busca sequencial.**
  - **Na realidade, é um dos métodos de busca mais eficientes que há.**
  - **O melhor caso de uma busca com sucesso continua sendo apenas uma comparação, quando o valor procurado se encontra no meio do *array*.**

Busca Binária em array

- **Análise da eficiência da pesquisa binária:**
  - **Podemos mostrar que o pior caso da busca binária com sucesso não ultrapassa $\lfloor \log_2 n \rfloor$ + 1 comparações, onde n é o tamanho do *array*.**
  - **Também podemos mostrar que o caso médio é $\log_2 n - 1$.**
  - **A busca com fracasso faz também $\lfloor \log_2 n \rfloor$ + 1 comparações.**

Busca Binária - Análise de Complexidade

- **O algoritmo funciona examinando as posições *A*[*inf*], *A*[*inf*+1],… *A*[*sup*].**
- **Cada iteração do laço elimina aproximadamente metade das posições ainda não examinadas. No pior caso:**
  - **Inicialmente: *n***
  - **Após a 1ª iteração: ~ *n*/2**
  - **Após a 2ª iteração: ~ *n*/4**

  …
  - **Após a *k*-ésima iteração: ~ $n/2^k = 1$**
- **Logo, no pior caso, o algoritmo faz ~ $\log_2 n$ iterações, ou seja, o algoritmo tem complexidade *O*(log *n*).**

Arrays Ordenados

- **Se os dados se encontram ordenados (em ordem crescente ou decrescente), a busca pode ser feita mais eficientemente.**
- **Ordenação toma tempo $\Theta(n \log n)$.**
- **Útil se a coleção não é alterada ou se é alterada pouco frequentemente.**
- **Busca sequencial ordenada tem complexidade média = $n/2$.**
- **Busca binária tem complexidade pior caso $O(\log n)$.**

Arrays - Inserção e Remoção de Elementos

- **É preciso empregar algoritmos de busca se:**
  - **A posição do elemento a ser removido não é conhecida.**
  - **O *array* não pode conter elementos repetidos.**
- **Se o *array* é ordenado, deseja-se preservar a ordem:**
  - **Deslocar elementos para criar / fechar posições.**
- **Se o *array* não é ordenado:**
  - **<u>Inserção</u>: Adicionar elemento no final do *array*.**
  - **<u>Remoção</u>: Utilizar o elemento do final do *array* para ocupar a posição removida.**
- **Se todas as posições estão preenchidas, inserção ocasiona *overflow:***
  - **Realocar o *array*.**
  - **Reportar erro.**

Inserção em array ordenado

- **Assume-se que o *array A* pode conter elementos iguais.**
- **Expressões lógicas são avaliadas em curto-circuito.**

```
proc inserção_ordenada (x, n, max, A [1 .. max]) {
   se n <= max então {
      i ← n
      enquanto i > 1 e A [i–1] > x fazer {
         A [i] ← A [i–1]
         i ← i–1
      }
      A [i] ← x
      n ← n + 1
   }
   senão reportar ("Overflow")
}
```


Remoção em array ordenado

- **Algoritmo remove o elemento *A* [*i*].**
- **Pressupõe-se que *i* foi obtido por uma operação de busca.**

```
proc remoção_ordenada (i, n, A [1 .. n])
 {
   se i <= n então {
      n ← n–1
      enquanto i <= n fazer
     {
         A [i] ← A [i+1]
         i ← i+1
      }
   }
   senão reportar ("Erro")
}
```