Pesquisa em Memória Primária

Livro “Projeto de Algoritmos” – Nívio Ziviani

Capítulo 5

http://www2.dcc.ufmg.br/livros/algoritmos/

Pesquisa em Memória Primária

- Introdução - Conceitos Básicos
- Pesquisa Seqüencial
- Árvores de Pesquisa
  - Árvores Binárias de Pesquisa sem Balanceamento
  - Árvores Binárias de Pesquisa com Balanceamento
    - Árvores SBB
    - Transformações para Manutenção da Propriedade SBB
- Pesquisa Digital
  - Trie , Patricia
- 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 Pesquisa sem sucesso.**

Introdução - Conceitos Básicos

- **Tabelas**
  - Conjunto de registros ou arquivos ⇒ TABELAS
  - Tabela:associada a entidades de vida curta, criadas na memória interna durante a execução de um programa.
  - Arquivo:geralmente associado a entidades de vida mais longa, armazenadas em memória externa.
  - Distinção não é rígida:
    - tabela: arquivo de índices
    - arquivo: tabela de valores de funções.

Escolha do Método de Pesquisa mais Adequado a uma Determinada Aplicação

- Depende principalmente:
  1. Quantidade dos dados envolvidos.
  2. Arquivo estar sujeito a inserções e retiradas freqüentes.

- Se conteúdo do arquivo é estável é importante minimizar o tempo de pesquisa, sem preocupação com o tempo necessário para estruturar o arquivo

Algoritmos de Pesquisa
Tipos Abstratos de Dados

- É importante considerar os algoritmos de pesquisa como tipos abstratos de dados, com um conjunto de operações associado a uma estrutura de dados, 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.
  5. Ordenar um arquivo para obter todos os registros em ordem de acordo com a chave.
  6. Ajuntar dois arquivos para formar um arquivo maior.

Dicionário

- Nome comumente utilizado para descrever uma estrutura de dados para pesquisa.
- Dicionário é um tipo abstrato de dados 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

Pesquisa Seqüencial

- **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 do tipo estruturado arranjo:

Pesquisa Seqüencial

```
# define Maxn          10
typedef long TipoChave;

typedef struct Registro {
    TipoChave Chave;
    /* outros componentes */
} Registro;

typedef int Indice;

typedef struct Tabela {
    Registro Item[Maxn + 1];
    Indice n;
} Tabela;
```

Pesquisa Seqüencial

- Implementação para as operações Inicializa, Pesquisa :

```
void Inicializa(Tabela *T)
{
   T->n = 0;
}

Indice Pesquisa(TipoChave x, Tabela *T)
{  int i;

  T->Item[0].Chave = x;
   i = T->n + 1;
   do {
          i--;
   }  while (T->Item[i].Chave != x);
   return i;
}
```

Pesquisa Seqüencial

```
Indice getN(Tabela *T)
{  return  T->n;
}
Indice Pesquisa(TipoChave x, Tabela *T, Indice k)
{    int i;
     T->Item[0].Chave = x;
     i = k;
     do {  i--;  }  while (T->Item[i].Chave != x);
     return i;
}
void recuperaRegistros(Tabela *T, TipoChave x)
{  Indice k;
      k =Pesquisa(x,T, getN(T)+1);
       if (k!= 0) printf("Achou x no %d!\n",k);
       else printf("Nao achou x!\n");
     while (k != 0) {
           k = Pesquisa(x,T,k);   printf("Achou x no %d!\n",k);
     }
}
```

Pesquisa Seqüencial

- Implementação para a operacao Insere:

```
void Insere(Registro Reg, Tabela *T)
{
    if (T->n == Maxn)
            printf("Erro : tabela cheia\n");
    else
    {
            T->n++;
            T->Item[T->n] = Reg;
    }
}
```

Pesquisa Seqüencial

- Pesquisa retorna o índice do registro que contém a chave x;
- Caso não esteja presente, o valor retornado é zero.
- A implementação não suporta mais de um registro com uma mesma chave.
- Para aplicações com esta característica é necessário incluir um argumento a mais na função Pesquisa para conter o índice a partir do qual se quer pesquisar.

Pesquisa Seqüencial

- Utilização de um registro sentinela na posição zero do array:
  - ❑ Garante que a pesquisa sempre termina: se o índice retornado por Pesquisa for zero, a pesquisa foi sem sucesso.
  - ❑ Não é necessário testar se $i > 0$, devido a isto:
    - o anel interno da função Pesquisa é extremamente simples: o índice i é decrementado e a chave de pesquisa é comparada com a chave que está no registro.
    - isto faz com que esta técnica seja conhecida como pesquisa seqüencial rápida.

Pesquisa Seqüencial

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 seqüencial é a melhor escolha para o problema de pesquisa em tabelas com até 25 registros.

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 o processo até que a chave seja encontrada, ou fique apenas um registro cuja chave é diferente da procurada, significando uma pesquisa sem sucesso.

Exemplo de Pesquisa Binária para a 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

- 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.

Transformação de Chave (Hashing)

- Os registros armazenados em uma tabela são diretamente endereçados a partir de uma transformação aritmética sobre a chave de pesquisa.

- Hash significa:
  - Fazer picadinho de carne e vegetais para cozinhar.
  - Fazer uma bagunça. (Webster's New World Dictionary)

Transformação de Chave (Hashing)

- Um método de pesquisa com o uso da transformação de chave é constituído de duas etapas principais:

  1 - Computar o valor da **função de transformação**, a qual transforma a chave de pesquisa em um endereço da tabela.

  2 - Considerando que duas ou mais chaves podem ser transformadas em um mesmo endereço de tabela, é necessário existir um método para lidar com colisões.

- Qualquer que seja a função de transformação, algumas colisões irão ocorrer fatalmente, e tais colisões têm de ser resolvidas de alguma forma.
- Mesmo que se obtenha uma função de transformação que distribua os registros de forma uniforme entre as entradas da tabela, existe uma alta probabilidade de haver colisões.

Transformação de Chave (Hashing)

- O paradoxo do aniversário (Feller,1968, p. 33), diz que em um grupo de 23 ou mais pessoas, juntas ao acaso, existe uma chance maior do que 50% de que 2 pessoas comemorem aniversário no mesmo dia.
- Assim, se for utilizada uma função de transformação uniforme que enderece 23 chaves randômicas em uma tabela de tamanho 365, a probabilidade de que haja colisões é maior do que 50%.

Transformação de Chave (Hashing)

- A probabilidade p de se inserir N itens consecutivos sem colisão em uma tabela de tamanho M é:

$$p = \frac{M-1}{M} \times \frac{M-2}{M} \times \ldots \times \frac{M-N+1}{M} =$$

$$\prod_{i=1}^{N} \frac{M-i+1}{M} = \frac{M!}{(M-N)!M^N}$$

Transformação de Chave (Hashing)

- Alguns valores de p para diferentes valores de N, onde $M = 365$.

N	p
10	0,883
22	0,524
23	0,493
30	0,303

- Para N pequeno a probabilidade p pode ser aproximada por $p \approx N(N-1))/730$. Por exemplo, para $N = 10$ então $p \approx 87,7\%$.

Funções de Transformação

- Uma função de transformação deve mapear chaves em inteiros dentro do intervalo $[0..M - 1]$, onde M é o tamanho da tabela.

- A função de transformação ideal é aquela que:
  - Seja simples de ser computada.
  - Para cada chave de entrada, qualquer uma das saídas possíveis é igualmente provável de ocorrer.

Método mais Usado

- Usa o resto da divisão por M .

$$h(K) = K \bmod M$$

ou

$$h(K) = K \% M \quad \text{(em linguagem C)}$$

onde K é um inteiro correspondente à chave.

Método mais Usado

- Cuidado na escolha do valor de M.   M deve ser um número primo, mas não qualquer primo: devem ser evitados os números primos obtidos a partir de

$$\mathbf{b^i \pm j}$$

onde b é a base do conjunto de caracteres (geralmente b = 64 para BCD, 128 para ASCII, 256 para EBCDIC, ou 100 para alguns códigos decimais), e i e j são pequenos inteiros.

Transformação de Chaves Não Numéricas

- As chaves não numéricas devem ser transformadas em números:

$$K = \sum_{i=1}^{n} \text{Chave}[i] \times p[i],$$

- n é o número de caracteres da chave.
- Chave[i] corresponde à representação ASCII do i-ésimo caractere da chave.
- p[i] é um inteiro de um conjunto de pesos gerados aleatoriamente para $1 \leq i \leq n$.

Transformação de Chaves Não Numéricas

- Vantagem de se usar pesos:

  Dois conjuntos diferentes de pesos p1 [i] e p2 [i], $1 \leq i \leq n$, leva a duas funções de transformação h1 (K) e h2 (K) diferentes.

Transformação de Chaves Não Numéricas

- Programa que gera um peso para cada caracter de uma chave constituída de n caracteres:

```
void GeraPesos(TipoPesos p)
{  /* Gera valores aleatorios entre 1 e 10.000 */
  int i;
  struct timeval semente;
  /* Utilizar o tempo como semente para a funcao srand() */
  gettimeofday(&semente,NULL);
  srand((int)(semente.tv_sec + 1000000*semente.tv_usec));
  for (i = 0; i < n; i++)
     p[i] =  1+(int) (10000.0*rand()/(RAND_MAX+1.0));
}
```

Transformação de Chaves Não Numéricas

- Implementação da função de transformação:

```
Indice h(TipoChave Chave, TipoPesos p)
{
    int i;
   unsigned int Soma = 0;
   int comp = strlen(Chave);

   for (i = 0; i < comp; i++)
         Soma += (unsigned int)Chave[i] * p[i];

   return (Soma % M);
}
```

Listas Encadeadas

- Uma das formas de resolver as **colisões** é simplesmente construir uma lista linear encadeada para cada endereço da tabela.

- Assim, todas as chaves com mesmo endereço são encadeadas em uma lista linear.

Listas Encadeadas

- **Exemplo:** Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação

$$h(Chave) = Chave \% M$$

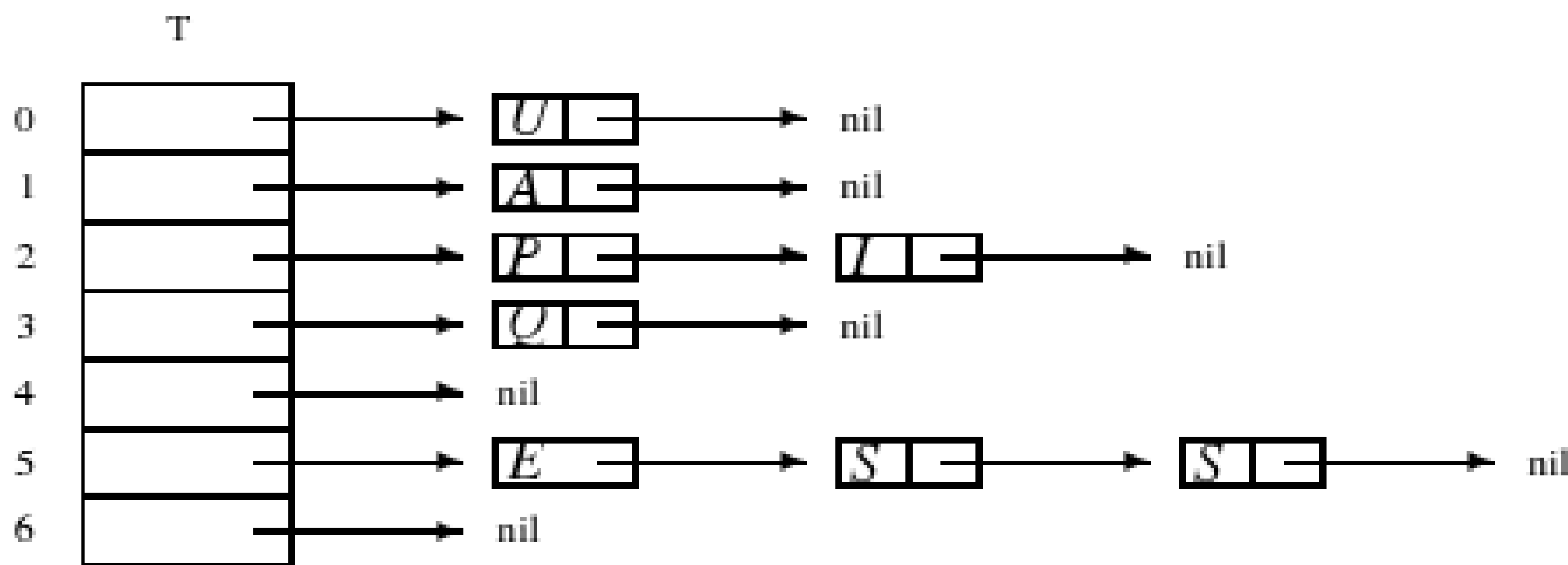
é utilizada para M = 7,

O resultado da inserção das chaves P E S Q U I S A na tabela é o seguinte:

Listas Encadeadas

- Por exemplo:

$h(A) = h(1) = 1$,
$h(E) = h(5) = 5$,
$h(S) = h(19) = 5$, etc



Estrutura do Dicionário para Listas Encadeadas

```
#define M              7
#define n             7
typedef char TipoChave[n];

typedef unsigned int  TipoPesos[n];

typedef struct TipoItem {
  /* outros componentes */
  TipoChave Chave;
} TipoItem;

typedef unsigned int Indice;
```

Estrutura do Dicionário para Listas Encadeadas

```
typedef struct Celula* Apontador;

typedef struct Celula {
  TipoItem Item;
  Apontador Prox;
} Celula;

typedef struct TipoLista {
            Celula *Primeiro, *Ultimo;
} TipoLista;

typedef TipoLista TipoDicionario[M];
```

Operações do Dicionário Usando Listas Encadeadas

```
void Inicializa(TipoDicionario T)
{
    int i;
    for (i = 0; i < M; i++)
            FLVazia(&T[i]);
}
```

Operações do Dicionário Usando Listas Encadeadas

```
Apontador Pesquisa(TipoChave Ch, TipoPesos p, TipoDicionario T)
{  /*Obs.: Apontador de retorno aponta para o item anterior da lista */
  Indice i;   Apontador Ap;
  i = h(Ch, p);
  if (Vazia(T[i])) return NULL;   /* Pesquisa sem sucesso */
  else  {
      Ap = T[i].Primeiro;
      while ((Ap->Prox->Prox != NULL) &&
              (strncmp(Ch, Ap->Prox->Item.Chave, sizeof(TipoChave)) ))
              Ap = Ap->Prox;
       if (!strncmp(Ch, Ap->Prox->Item.Chave, sizeof(TipoChave)))
              return Ap;
       else return NULL;   /* Pesquisa sem sucesso */
   }
}
```

Operações do Dicionário Usando Listas Encadeadas

```
void Insere(TipoItem x, TipoPesos p, TipoDicionario T)
{
        if (Pesquisa(x.Chave, p, T) == NULL)
                Ins(x, &T[h(x.Chave, p)]);
        else
                printf(" Registro ja  esta  presente\n");
}
```

Operações do Dicionário Usando Listas Encadeadas

```
void Retira(TipoItem x, TipoPesos p, TipoDicionario T)
{
   Apontador Ap;

   Ap = Pesquisa(x.Chave, p, T);

  if (Ap == NULL)   printf(" Registro nao esta  presente\n");

  else      Ret(Ap, &T[h(x.Chave, p)], &x);
}
```

Análise

- Assumindo que qualquer item do conjunto tem igual probabilidade de ser endereçado para qualquer entrada de T, então o comprimento esperado de cada lista encadeada é N/M, onde N representa o número de registros na tabela e M o tamanho da tabela.

- **Logo:** as operações Pesquisa, Insere e Retira custam O(1 + N/M) operações em média, onde a constante 1 representa o tempo para encontrar a entrada na tabela e N/M o tempo para percorrer a lista.

  Para valores de M próximos de N , o tempo se torna constante, isto é, independente de N .

Endereçamento Aberto

- Quando o número de registros a serem armazenados na tabela puder ser previamente estimado, então não haverá necessidade de usar apontadores para armazenar os registros.

- Existem vários métodos para armazenar N registros em uma tabela de tamanho $M > N$, os quais utilizam os lugares vazios na própria tabela para resolver as **colisões**. (Knuth, 1973, p.518)

Endereçamento Aberto

- **No Endereçamento aberto** todas as chaves são armazenadas na própria tabela, sem o uso de apontadores explícitos.

- Existem várias propostas para a escolha de localizações alternativas. A mais simples é chamada de hashing linear, onde a posição h_j na tabela é dada por:

$$h_j = (h(x) + j) \bmod M, \text{ para } 1 \leq j \leq M - 1.$$

Exemplo

- Se a i-ésima letra do alfabeto é representada pelo número i e a função de transformação

  h(Chave) = Chave % M

  é utilizada para M = 7,

- então o resultado da inserção das chaves L U N E S na tabela, usando hashing linear para resolver colisões é mostrado abaixo.

Exemplo

- Por exemplo:

$h(L) = h(12) = 5,\quad h(U) = h(21) = 0,$

$h(N) = h(14) = 0,\quad h(E) = h(5) = 5,$

$h(S) = h(19) = 5.$

	T
0	*U*
1	*N*
2	*S*
3	
4	
5	*L*
6	*E*

Estrutura do Dicionário Usando Endereçamento Aberto

```
#define Vazio        "!!!!!!!!!!"
#define Retirado     "**********"
#define M            7
#define n            11   /* Tamanho da chave */
```

Estrutura do Dicionário Usando Endereçamento Aberto

```
typedef unsigned int Apontador;

typedef char TipoChave[n];
typedef unsigned TipoPesos[n];

typedef struct TipoItem {
  /* outros componentes */
  TipoChave Chave;
} TipoItem;

typedef unsigned int Indice;
typedef TipoItem TipoDicionario[M];
```

Operações do Dicionário Usando Endereçamento Aberto

```
void Inicializa(TipoDicionario T)
{
    int i;
    for (i = 0; i < M; i++)
            memcpy(T[i].Chave, Vazio, n);
}
```

Operações do Dicionário Usando Endereçamento Aberto

```
Apontador Pesquisa(TipoChave Ch, TipoPesos p, TipoDicionario T)
{ unsigned int  i = 0;
  unsigned int  Inicial;

  Inicial = h(Ch, p);

   while ( (strcmp (T[(Inicial + i) % M].Chave,Vazio) != 0) &&
              (strcmp ( T[(Inicial + i) % M].Chave, Ch) != 0)  &&
              (i < M))
              i++;
  if (strcmp (T[(Inicial + i) % M].Chave, Ch) == 0)
               return ((Inicial + i) % M);
  else return M;   /* Pesquisa sem sucesso */
}
```

Operações do Dicionário Usando Endereçamento Aberto

```
void Insere(TipoItem x, TipoPesos p, TipoDicionario T)
{   unsigned int i = 0;        unsigned int Inicial;

    if (Pesquisa(x.Chave, p, T) < M)  {
            printf("Elemento ja esta presente\n");
            return;
    }
    Inicial = h(x.Chave, p);
    while ( (strcmp ( T[(Inicial + i) % M].Chave,Vazio) != 0) &&
             (strcmp ( T[(Inicial + i) % M].Chave, Retirado) != 0) &&
             ( i < M))  i++;
    if (i < M)   {
       strcpy (T[(Inicial + i) % M].Chave, x.Chave); /* Copiar os demais
                                              campos de x, se existirem */
    }
    else printf(" Tabela cheia\n");
}
```

Operações do Dicionário Usando Endereçamento Aberto

```
void Retira(TipoChave Ch, TipoPesos p, TipoDicionario T)
{
    Indice i;

    i = Pesquisa(Ch, p, T);

    if (i < M)   memcpy(T[i].Chave, Retirado, n);

    else   printf("Registro nao esta presente\n");
}
```

Análise

- Seja $\alpha = N/M$ o fator de carga da tabela. Conforme demonstrado por Knuth (1973), o custo de uma pesquisa com sucesso é

$$C(n) = \frac{1}{2}\left(1 + \frac{1}{1-\alpha}\right)$$

- O hashing linear sofre de um mal chamado agrupamento (clustering) (Knuth, 1973, pp.520–521).

- Este fenômeno ocorre na medida em que a tabela começa a ficar cheia, pois a inserção de uma nova chave tende a ocupar uma posição na tabela que esteja contígua a outras posições já ocupadas, o que deteriora o tempo necessário para novas pesquisas.

Análise

- Entretanto, apesar do hashing linear ser um método relativamente pobre para resolver colisões os resultados apresentados são bons.

- O melhor caso, assim como o caso médio, é $O(1)$.

Vantagens e Desvantagens de Transformação da Chave

- Vantagens:
  - Alta eficiência no custo de pesquisa, que é $O(1)$ para o caso médio.
  - Simplicidade de implementação

- Desvantagens:
  - Custo para recuperar os registros na ordem lexicográfica das chaves é alto, sendo necessário ordenar o arquivo.
  - Pior caso é $O(N)$

Árvores de Pesquisa

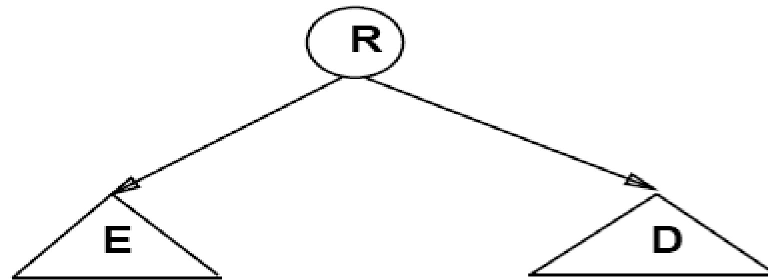
- A árvore de pesquisa é uma estrutura de dados muito eficiente para armazenar informação.

- Particularmente adequada quando existe necessidade de considerar todos ou alguma combinação de:

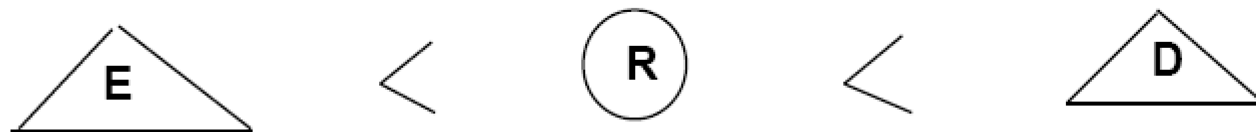
  1. Acesso direto e seqüencial eficientes.
  2. Facilidade de inserção e retirada de registros.
  3. Boa taxa de utilização de memória.
  4. Utilização de memória primária e secundária.

Árvores Binárias de Pesquisa sem Balanceamento

- Para qualquer nó que contenha um registro

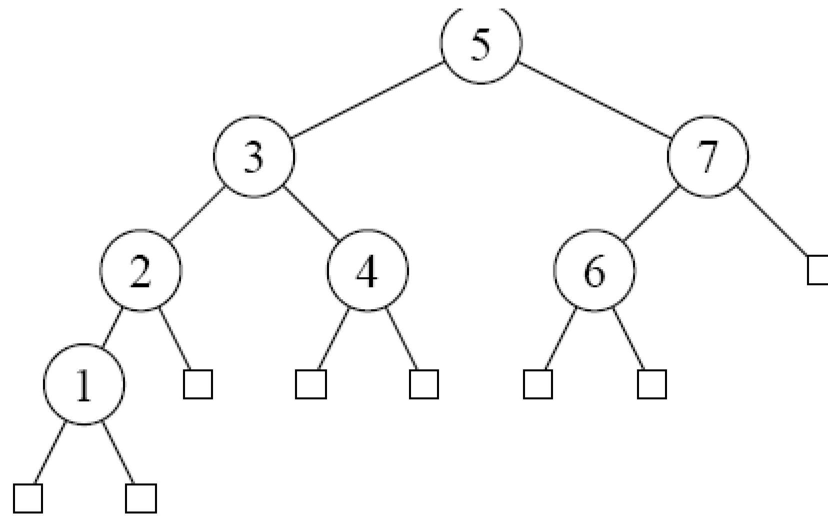


- Temos a relação invariante



1. Todos os registros com chaves menores estão na subárvore à esquerda.
2. Todos os registros com chaves maiores estão na subárvore à direita.

Árvores Binárias de Pesquisa sem Balanceamento



- O nível do nó raiz é 0.
  - Se um nó está no nível i então a raiz de suas subárvores estão no nível i + 1.
  - A altura de um nó é o comprimento do caminho mais longo deste nó até um nó folha.
  - A altura de uma árvore é a altura do nó raiz.

Implementação do Tipo Abstrato de Dados Dicionário usando a Estruturade Dados Árvore Binária de Pesquisa

- Estrutura de dados:

```
typedef long TipoChave;

typedef struct Registro {
    TipoChave Chave;
    /* outros componentes */
} Registro;

typedef struct No * Apontador;

typedef Struct No {
        Registro Reg;
        Apontador Esq, Dir;
} No;

typedef Apontador TipoDicionario;
```

Procedimento para Pesquisar na Árvore

- Para encontrar um registro com uma chave x:
  - Compare-a com a chave que está na raiz.
  - Se x é menor, vá para a subárvore esquerda.
  - Se x é maior, vá para a subárvore direita.
  - Repita o processo recursivamente, até que a chave procurada seja encontrada ou um nó folha é atingido.
  - Se a pesquisa tiver sucesso então o conteúdo do registro retorna no próprio registro x.

Procedimento para Pesquisar na Árvore

```
void Pesquisa(Registro *x, Apontador *p)
{
   if (*p == NULL) {
      printf("Erro : Registro nao esta presente na arvore\n");
   return;
   }
   if (x->Chave < (*p)->Reg.Chave) {
            Pesquisa(x, &(*p)->Esq);
            return;
   }
   if (x->Chave > (*p)->Reg.Chave)    Pesquisa(x, &(*p)->Dir);
   else *x = (*p)->Reg;
}
```

Procedimento para Inserir na Árvore

- Atingir um apontador nulo em um processo de pesquisa significa uma pesquisa sem sucesso.
- O apontador nulo atingido é o ponto de inserção.

Procedimento para Inserir na Árvore

```
void Insere(Registro x, Apontador *p)
 {
      if (*p == NULL)  {
           *p = (Apontador)malloc(sizeof(No));
           (*p)->Reg = x;
           (*p)->Esq = NULL; (*p)->Dir = NULL;
           return;
      }
      if (x.Chave < (*p)->Reg.Chave) {
            Insere(x, &(*p)->Esq);
            return;
      }
      if (x.Chave > (*p)->Reg.Chave) Insere(x, &(*p)->Dir);
      else printf("Erro : Registro ja existe na arvore\n");
}
```

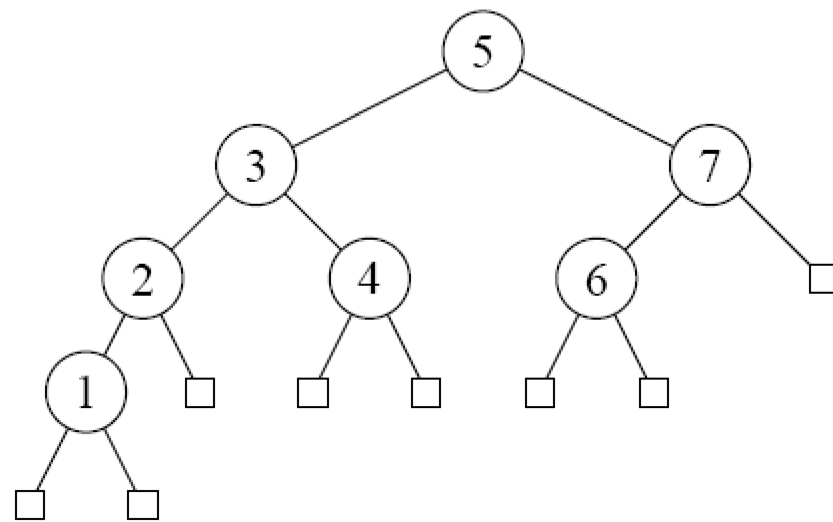
Procedimentos para Inicializar e Criar a Árvore

```
void Inicializa(Apontador *Dicionario)
{
    *Dicionario = NULL;
}
```

Procedimento para Retirar x da Árvore

- Alguns comentários:
  1. A retirada de um registro não é tão simples quanto a inserção.
  2. Se o nó que contém o registro a ser retirado possui no máximo um descendente $\Rightarrow$ a operação é simples.
  3. No caso do nó conter dois descendentes o registro a ser retirado deve ser primeiro:
     - substituído pelo registro mais à direita na subárvore esquerda;
     - ou pelo registro mais à esquerda na subárvore direita.

Exemplo da Retirada de um Registro da Árvore



- Assim: para retirar o registro com chave 5 da árvore basta trocá-lo pelo registro com chave 4 ou pelo registro com chave 6, e então retirar o nó que recebeu o registro com chave 5.

Exemplo da Retirada de um Registro da Árvore

```
void Antecessor(Apontador q, Apontador *r)
{
    if ( (*r)->Dir != NULL)
    {
        Antecessor(q, &(*r)->Dir);
        return;
    }
    q->Reg = (*r)->Reg;
    q = *r;
    *r = (*r)->Esq;
    free(q);
}
```

Exemplo da Retirada de um Registro da Árvore

```
void Retira(Registro x, Apontador *p)
{   Apontador Aux;

    if (*p == NULL) {
        printf("Erro : Registro nao esta na arvore\n");
        return;
    }
     if (x.Chave < (*p)->Reg.Chave) {
            Retira(x, &(*p)->Esq);       return;
     }
     if (x.Chave > (*p)->Reg.Chave){
            Retira(x, &(*p)->Dir);
            return;
    }
```

Exemplo da Retirada de um Registro da Árvore

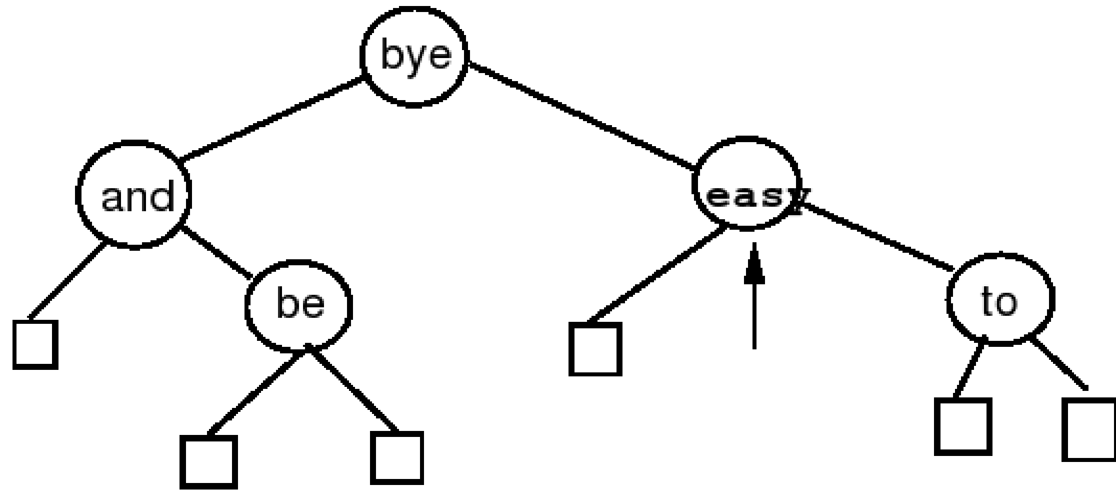
```
    if ((*p)->Dir == NULL)  {
            Aux = *p;
            *p = (*p)->Esq;
            free(Aux);
            return;
    }
    if ((*p)->Esq != NULL) {
           Antecessor(*p, &(*p)->Esq);
            return;
    }
    Aux = *p;
    *p = (*p)->Dir;
     free(Aux);
}
```

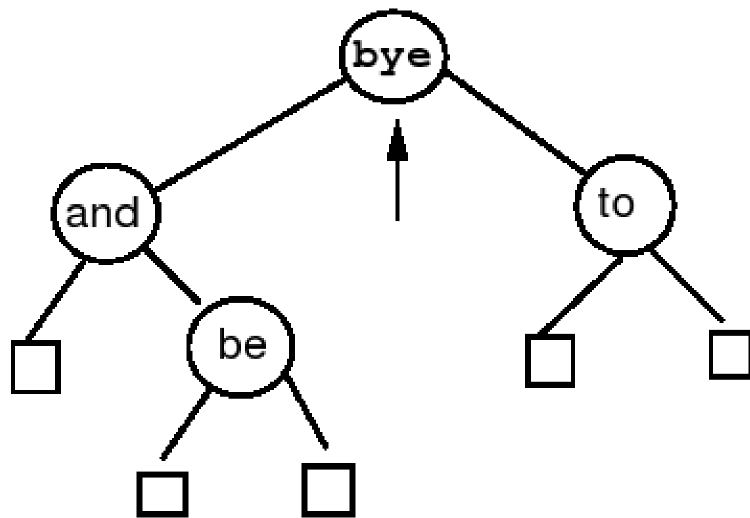
Exemplo da Retirada de um Registro da Árvore

Obs.: proc. recursivo Antecessor só é ativado quando o nó que contém registro a ser retirado possui 2 descendentes.

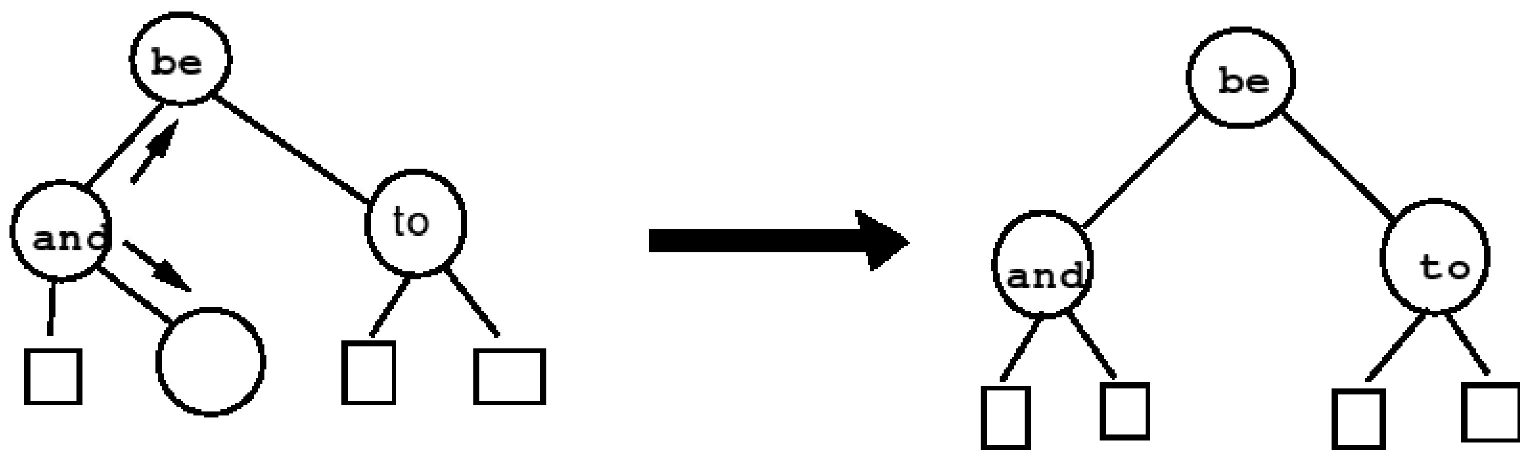
Solução usada por Wirth, 1976, p.211.

Outro Exemplo de Retirada de Nó





Outro Exemplo de Retirada de Nó

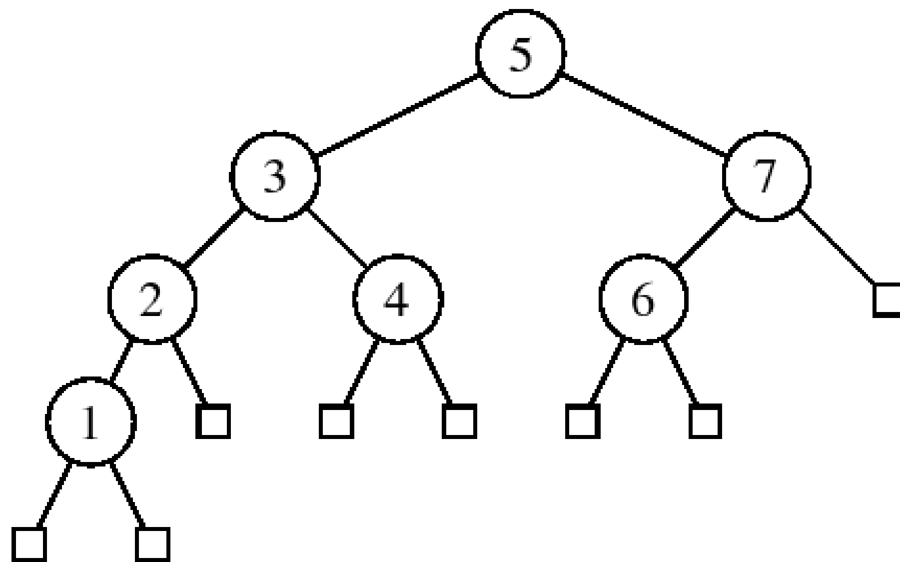


Caminhamento Central

- Após construída a árvore, pode ser necessário percorrer todo os registros que compõem a tabela ou arquivo.
- Existe mais de uma ordem de caminhamento em árvores, mas a mais útil é a chamada ordem de caminhamento central.
- O caminhamento central é mais bem expresso em termos recursivos:
  1. caminha na subárvore esquerda na ordem central;
  2. visita a raiz;
  3. caminha na subárvore direita na ordem central.
- Uma característica importante do caminhamento central é que os nós são visitados de forma ordenada.

Caminhamento Central

- Percorrer a árvore:



- usando caminhamento central recupera as chaves na ordem 1, 2, 3, 4, 5, 6 e 7.

Caminhamento Central

- O procedimento Central é mostrado abaixo:

```
void Central(Apontador p)
{
   if (p == NULL) return;
   Central(p->Esq);
   printf("%ld\n", p->Reg.Chave);
   Central(p->Dir);
}
```

Análise

- O número de comparações em uma pesquisa com sucesso:

  melhor caso : $C(n) = O(1)$

  pior caso: $C(n) = O(n)$

  caso médio : $C(n) = O(\log n)$

- O tempo de execução dos algoritmos para árvores binárias de pesquisa dependem muito do formato das árvores.

Análise

1. Para obter o pior caso basta que as chaves sejam inseridas em ordem crescente ou decrescente. Neste caso a árvore resultante é uma lista linear, cujo número médio de comparações é $(n + 1)/2$.

2. Para uma árvore de pesquisa aleatoria o número esperado de comparações para recuperar um registro qualquer é cerca de $1{,}39 \log n$, apenas 39% pior que a árvore completamente balanceada.