Algoritmos e Estruturas de Dados: Árvore Binária de Busca

Rômulo Silva de Oliveira
Departamento de Automação e Sistemas – DAS – UFSC

romulo@das.ufsc.br
http://www.das.ufsc.br/~romulo
Maio/2011

Referências

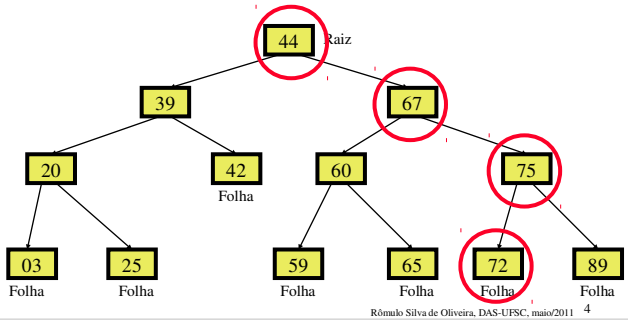
- Mastering Algorithms with C
  Kyle Loudon
  O´Reilly, 1999
- Livros de algoritmos e estruturas de dados em geral

Árvore Binária de Busca – Introdução 1/21

- Árvore binária organizada para pesquisa
- Pesquisa por um nodo inicia na raiz
- Desce nível por nível até encontrar o nodo buscado
- Quando encontra um nodo maior que o buscado, segue o pointer da esquerda
- Quando encontra um nodo menor que o buscado, segue o pointer da direita
- Caso chegue em uma folha, ele não existe na árvore

Árvore Binária de Busca – Introdução 2/21

- Busca pelo nodo 72



Árvore Binária de Busca – Introdução 3/21

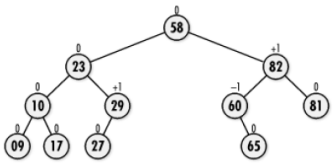
- Árvores binárias de busca são estruturas eficientes para pesquisa
- No pior caso percorre apenas um galho
- O(lg *N*)
  - *N* é o número de nodos na árvore
- Desde que a árvore esteja balanceada
- Caso a árvore esteja completamente desbalanceada: O(*N*)
  - Como uma lista encadeada

Árvore Binária de Busca – Introdução 4/21

- Árvores AVL
  Adel'son-Vel'skii e Landis, 1962
- Árvore binária
- Cada nodo possui um fator de balanceamento
- O fator de balanceamento de um nodo é
  a altura da sub-árvore cuja raiz é o seu filho esquerdo
  menos
  a altura da sub-árvore cuja raiz é o seu filho direito

Árvore Binária de Busca – Introdução 5/21

- Árvore AVL exige que todos os fatores de balanceamento sejam +1,0,-1
- Quando nodos são inseridos a árvore precisa ser ajustada
- +1 é pesada a esquerda
- -1 é pesada a direita
- 0 é balanceada



Árvore Binária de Busca – Introdução 6/21

- Inserção e pesquisa na árvore AVL é semelhante à árvore binária comum
- Após a inserção é necessário
  - Contabilizar as mudanças nos fatores de balanceamento
  - Se algum fator de balanceamento terminar +2 ou -2 é necessário rebalançear a árvore deste ponto para baixo
- Rebalançeamento é feito com **operações de rotação**

Árvore Binária de Busca – Introdução 7/21

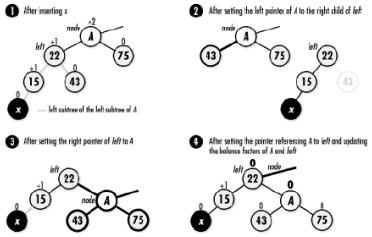
- Uma rotação rebalanceia parte de uma árvore AVL
- A rotação reposiciona nodos enquanto preserva a relação sub-árvore esquerda < pai < sub-árvore direita
- Após a rotação:
  - Os fatores de balanceamento de todos os nodos na sub-árvore rotacionada são novamente -1, 0 ou +1
- 4 tipos de rotação:
  - Left-left LL
  - Left-Right LR
  - Right-Right RR
  - Right-Left RL

Árvore Binária de Busca – Introdução 8/21

- Rotação Left-Left
  - X é o elemento recem inserido
  - A é o ancestral mais próximo de X com fator de balançeamento ±2
  - X está na sub-árvore esquerda da sub-árvore esquerda de A
  - Pivo é o filho a esquerda de A
- Ações:
  - A.esquerda ← Pivo.direita
  - Pivo.direita ← A
  - Quem apontava A agora aponta Pivo
  - Fator de balançeamento de A e Pivo fica 0

Árvore Binária de Busca – Introdução 9/21

- Rotação Left-Left



Árvore Binária de Busca – Introdução 10/21

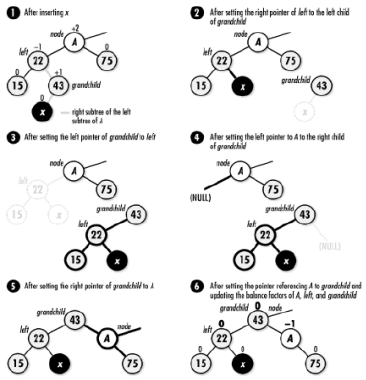
- Rotação Right-Right
  - X é o elemento recem inserido
  - A é o ancestral mais próximo de X com fator de balançeamento ±2
  - X está na sub-árvore direita da sub-árvore direita de A
- Simétrico à rotação Left-Left

Árvore Binária de Busca – Introdução 11/21

- Rotação Left-Right
  - X é o elemento recem inserido
  - A é o ancestral mais próximo de X com fator de balançeamento ±2
  - X está na sub-árvore direita da sub-árvore esquerda de A
  - Pivo é o filho esquerdo de A
  - Neto é o filho direito de Pivo
- Ações:
  - Pivo.direita ← Neto.esquerda
  - Neto.esquerda ← Pivo
  - A.esquerda ← Neto.direita
  - Neto.direita ← A
  - Quem apontava A agora aponta Neto

Árvore Binária de Busca – Introdução 12/21

- Rotação Left-Right

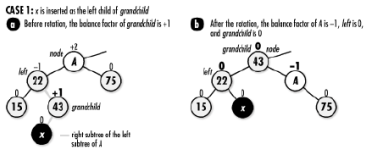


Árvore Binária de Busca – Introdução 13/21

- Rotação Left-Right
- Ajuste do fator de balanceamento
- A e Pivo depende do caso
- Neto fica 0
- Demais não mudam

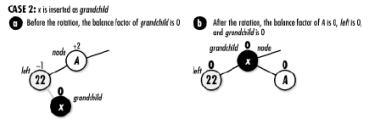
Árvore Binária de Busca – Introdução 14/21

- Rotação Left-Right
  - Ajuste do fator de balanceamento depende do caso
- Fator de balanceamento do Neto era +1
  - A fica -1
  - Pivo fica 0



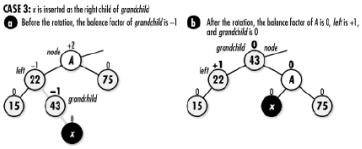
Árvore Binária de Busca – Introdução 15/21

- Rotação Left-Right
  - Ajuste do fator de balanceamento depende do caso
- Fator de balanceamento do Neto era 0
  - A fica 0
  - Pivo fica 0



Árvore Binária de Busca – Introdução 16/21

- Rotação Left-Right
  - Ajuste do fator de balanceamento depende do caso
- Fator de balanceamento do Neto era –1
  - A fica 0
  - Pivo fica +1

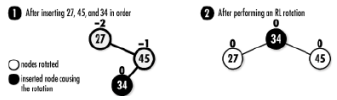


Árvore Binária de Busca – Introdução 17/21

- Rotação Right-Left
  - X é o elemento recem inserido
  - A é o ancestral mais próximo de X com fator de balançeamento ±2
  - X está na sub-árvore esquerda da sub-árvore direita de A
  - Pivo é o filho direito de A
  - Neto é o filho esquerdo de Pivo
- Simétrico à rotação Left-Right

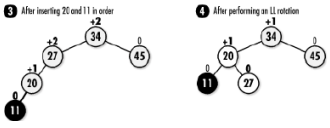
Árvore Binária de Busca – Introdução 18/21

- Exemplo inserção
  - Rotação RL é necessária



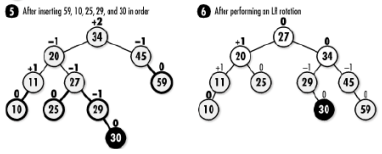
Árvore Binária de Busca – Introdução 19/21

- Exemplo inserção
  - Rotação LL é necessária



Árvore Binária de Busca – Introdução 20/21

- Exemplo inserção
  - Rotação LR é necessária



Árvore Binária de Busca – Introdução 21/21

- Remoção pode ser completa
  - Requer rebalanceamento da árvore
- Remoção pode ser preguiçosa
  - Nodo é marcado como "escondido" mas não é realmente removido
  - O campo *hidden* da estrutura *AvlNode* recebe 1
- Se o dado for inserido novamente, apenas torna o nodo vis ível
- Remoção preguiçosa é aceitável se o número de nodos removidos for pequeno com relação ao número de nodos inseridos
- Se muitos nodos serão removidos,
  melhor fazer uma remoção completa e ajustar a árvore

Árvore Binária de Busca – Interface 1/6

- ***bistree_init***
  - void bistree_init(BisTree **tree*, void (*compare)(const void *key1, const void *key2),
    void (**destroy*)(void **data*));
- **Retorno**
  - Nenhum.
- **Descrição**
  - Inicializa a árvore binária de busca especificada por *tree*. Esta operação deve ser chamada para uma árvore binária de busca antes que a árvore possa ser usada com qualquer outra operação. O pointer para função *compare* especifica uma função definida pelo usuário para comparar elementos. Esta função deve retornar 1 se *key1>key2*, 0 se *key1=key2*, e -1 se *key1<key2*. O argumento *destroy* prove uma maneira para liberar dados dinamicamente alocados quando *bistree_destroy* for chamada. Ele funciona de maneira similar àquela descrita para *bitree_destroy*. Para uma árvore binária de busca contendo dados que não devem ser liberados, *destroy* deve ser NULL.

Árvore Binária de Busca – Interface 2/6

- ***bistree_destroy***
  - void bistree_destroy(BisTree **tree*);
- **Retorno**
  - Nenhum.
- **Descrição**
  - Destrói a árvore binária de busca especificada por *tree*. Nenhuma outra operação é permitida após a chamada de *bistree_destroy* a não ser que *bistree_init* seja chamada novamente. A operação *bistree_destroy* remove todos os nodos da árvore binária de busca e chama a função passada como *destroy* para *bistree_init* uma vez para cada nodo a medida que eles são removidos, desde que *destroy* não seja NULL.

Árvore Binária de Busca – Interface 3/6

- ***bistree_insert***
  - int bistree_insert(BisTree *tree, const void *data);
- **Retorno**
  - 0 se a inserção do nodo tiver sucesso, 1 se o nodo já está na árvore, ou -1 caso contrário.
- **Descrição**
  - Insere um nodo na árvore binária de busca especificada por *tree*. O novo nodo contem um pointer para *data*, assim a memória referenciada por *data* deve permanecer válida pelo tempo que o nodo permanecer na árvore binária de busca. É responsabilidade do chamador gerenciar a memória associada com *data*.

Árvore Binária de Busca – Interface 4/6

- ***bistree_remove***
  - int bistree_remove(BisTree *tree, const void *data);
- **Retorno**
  - 0 se a remoção do nodo teve sucesso, ou -1 caso contrário.
- **Descrição**
  - Remove o nodo que confere com *data* da árvore binária de busca especificada por *tree*. Na realidade, esta operação apenas realiza uma "remoção preguiçosa", na qual o nodo é simplesmente marcado como "escondido". Logo, nenhum pointer para os dados que conferem com *data* é retornado. Além disto, os dados na árvore devem permanecer válidos mesmo após terem sido removidos. Consequentemente, o tamanho da árvore binária de busca, como retornado por *bistree_size*, não decresce após a remoção de um nodo.

Árvore Binária de Busca – Interface 5/6

- ***bistree_lookup***
  - int bistree_lookup(const BisTree *tree, void **data);
- **Retorno**
  - 0 se o dado é encontrado na árvore binária de busca, ou -1 caso contrário.
- **Descrição**
  - Determina se um nodo confere com *data* na árvore binária de busca especificada como *tree*. Se um nodo é encontrado, *data* aponta para o dado encontrado na árvore binária de busca após o retorno.

Árvore Binária de Busca – Interface 6/6

- ***bistree_size***
  - int bistree_size(const BisTree *tree);
- **Retorno**
  - Número de nodos na árvore.
- **Descrição**
  - Macro que avalia o número de nodos na árvore binária de busca especificada por *tree*.

Árvore Binária de Busca – Header 1/2

```
#ifndef BISTREE_H
#define BISTREE_H

#include "bitree.h"

#define     AVL_LFT_HEAVY    1
#define     AVL_BALANCED     0
#define     AVL_RGT_HEAVY    -1


typedef struct AvlNode_ {

    void        *data;
    int         hidden;
    int         factor;

} AvlNode;


typedef BiTree BisTree;
```

Árvore Binária de Busca – Header 2/2

```
void bistree_init(BisTree *tree,
    int (*compare)(const void *key1, const void *key2),
    void (*destroy)(void *data));

void bistree_destroy(BisTree *tree);

int bistree_insert(BisTree *tree, const void *data);

int bistree_remove(BisTree *tree, const void *data);

int bistree_lookup(BisTree *tree, void **data);

#define bistree_size(tree) ((tree)->size)

#endif
```


Árvore Binária de Busca – Implementação 1/24

```
#include <stdlib.h>
#include <string.h>

#include "bistree.h"

static void destroy_right(BisTree *tree, BiTreeNode *node);
```


Árvore Binária de Busca – Implementação 2/24

```
static void rotate_left(BiTreeNode **node) {

    BiTreeNode      *left, *grandchild;

    left = bitree_left(*node);

    if (((AvlNode *)bitree_data(left))->factor == AVL_LFT_HEAVY) {

        // Perform an LL rotation
        bitree_left(*node) = bitree_right(left);
        bitree_right(left) = *node;
        ((AvlNode *)bitree_data(*node))->factor = AVL_BALANCED;
        ((AvlNode *)bitree_data(left))->factor = AVL_BALANCED;
        *node = left;
    }
```


Árvore Binária de Busca – Implementação 3/24

```
else {
        // Perform an LR rotation

        grandchild = bitree_right(left);
        bitree_right(left) = bitree_left(grandchild);
        bitree_left(grandchild) = left;
        bitree_left(*node) = bitree_right(grandchild);
        bitree_right(grandchild) = *node;
```


Árvore Binária de Busca – Implementação 4/24

```
    switch (((AvlNode *)bitree_data(grandchild))->factor) {
        case AVL_LFT_HEAVY:
            ((AvlNode *)bitree_data(*node))->factor = AVL_RGT_HEAVY;
            ((AvlNode *)bitree_data(left))->factor = AVL_BALANCED;
            break;
        case AVL_BALANCED:
            ((AvlNode *)bitree_data(*node))->factor = AVL_BALANCED;
            ((AvlNode *)bitree_data(left))->factor = AVL_BALANCED;
            break;
        case AVL_RGT_HEAVY:
            ((AvlNode *)bitree_data(*node))->factor = AVL_BALANCED;
            ((AvlNode *)bitree_data(left))->factor = AVL_LFT_HEAVY;
            break;
        }
    ((AvlNode *)bitree_data(grandchild))->factor = AVL_BALANCED;
    *node = grandchild;
    }
return;
}
```


Árvore Binária de Busca – Implementação 5/24

```
static void rotate_right(BiTreeNode **node) {

    BiTreeNode      *right,   *grandchild;

    right = bitree_right(*node);

    if (((AvlNode *)bitree_data(right))->factor == AVL_RGT_HEAVY) {

        // Perform an RR rotation

        bitree_right(*node) = bitree_left(right);
        bitree_left(right) = *node;
        ((AvlNode *)bitree_data(*node))->factor = AVL_BALANCED;
        ((AvlNode *)bitree_data(right))->factor = AVL_BALANCED;
        *node = right;
    }
```

Árvore Binária de Busca – Implementação 6/24

```
else {

        // Perform an RL rotation

        grandchild = bitree_left(right);
        bitree_left(right) = bitree_right(grandchild);
        bitree_right(grandchild) = right;
        bitree_right(*node) = bitree_left(grandchild);
        bitree_left(grandchild) = *node;
```


Árvore Binária de Busca – Implementação 7/24

```
    switch (((AvlNode *)bitree_data(grandchild))->factor) {
        case AVL_LFT_HEAVY:
                ((AvlNode *)bitree_data(*node))->factor = AVL_BALANCED;
                ((AvlNode *)bitree_data(right))->factor = AVL_RGT_HEAVY;
                break;
        case AVL_BALANCED:
                ((AvlNode *)bitree_data(*node))->factor = AVL_BALANCED;
                ((AvlNode *)bitree_data(right))->factor = AVL_BALANCED;
                break;
        case AVL_RGT_HEAVY:
                ((AvlNode *)bitree_data(*node))->factor = AVL_LFT_HEAVY;
                ((AvlNode *)bitree_data(right))->factor = AVL_BALANCED;
                break;
        }
    ((AvlNode *)bitree_data(grandchild))->factor = AVL_BALANCED;
    *node = grandchild;
}
return;
}
```


Árvore Binária de Busca – Implementação 8/24

```
static void destroy_left(BisTree *tree, BiTreeNode *node) {
    BiTreeNode      **position;

    if (bitree_size(tree) == 0)     return;
    if (node == NULL)
        position = &tree->root;
    else    position = &node->left;

    if (*position != NULL) {
        destroy_left(tree, *position);
        destroy_right(tree, *position);
        if (tree->destroy != NULL)
            tree->destroy(((AvlNode *)(*position)->data)->data);

        free((*position)->data);
        free(*position);
        *position = NULL;
        tree->size--;
    }
    return;
}
```


Árvore Binária de Busca – Implementação 9/24

```
static void destroy_right(BisTree *tree, BiTreeNode *node) {

    BiTreeNode      **position;

    if (bitree_size(tree) == 0)     return;
    if (node == NULL)
        position = &tree->root;
    else    position = &node->right;

    if (*position != NULL) {
        destroy_left(tree, *position);
        destroy_right(tree, *position);
        if (tree->destroy != NULL)
            tree->destroy(((AvlNode *)(*position)->data)->data);

        free((*position)->data);
        free(*position);
        *position = NULL;
        tree->size--;
    }
return;
}
```


Árvore Binária de Busca – Implementação 10/24

```
static int insert(BisTree *tree, BiTreeNode **node, const void *data, int   *balanced) {

    AvlNode         *avl_data;
    int             cmpval,  retval;

    if (bitree_is_eob(*node)) {

        // Handle insertion into an empty tree
        if ((avl_data = (AvlNode *)malloc(sizeof(AvlNode))) == NULL)
            return -1;

        avl_data->factor = AVL_BALANCED;
        avl_data->hidden = 0;
        avl_data->data = (void *)data;

        return bitree_ins_left(tree, *node, avl_data);

    }
```


Árvore Binária de Busca – Implementação 11/24

```
    else { // Handle insertion into a tree that is not empty
        cmpval = tree->compare(data, ((AvlNode *)bitree_data(*node))->data);
        if (cmpval < 0) {               // Move to the left

            if (bitree_is_eob(bitree_left(*node))) {
                if ((avl_data = (AvlNode *)malloc(sizeof(AvlNode))) == NULL)
                    return -1;
                avl_data->factor = AVL_BALANCED;
                avl_data->hidden = 0;
                avl_data->data = (void *)data;

                if (bitree_ins_left(tree, *node, avl_data) != 0)    return -1;
                *balanced = 0;
            }
            else {
                if ((retval = insert(tree, &bitree_left(*node), data, balanced)) != 0) {
                    return retval;
                }
            }
```

Árvore Binária de Busca – Implementação 12/24

```
    // Ensure that the tree remains balanced
    if (!(*balanced)) {
    switch (((AvlNode *)bitree_data(*node))->factor) {
    case AVL_LFT_HEAVY:
    rotate_left(node);
    *balanced = 1;
    break;
    case AVL_BALANCED:
    ((AvlNode *)bitree_data(*node))->factor = AVL_LFT_HEAVY;
    break;
    case AVL_RGT_HEAVY:
    ((AvlNode *)bitree_data(*node))->factor = AVL_BALANCED;
    *balanced = 1;
    }
    }

} /* if (cmpval < 0) */
```


Árvore Binária de Busca – Implementação 13/24

```
    else if (cmpval > 0) {    // Move to the right
        if (bitree_is_eob(bitree_right(*node))) {
            if ((avl_data = (AvlNode *)malloc(sizeof(AvlNode))) == NULL)
                return -1;
            avl_data->factor = AVL_BALANCED;
            avl_data->hidden = 0;
            avl_data->data = (void *)data;

            if (bitree_ins_right(tree, *node, avl_data) != 0)
                return -1;
            *balanced = 0;
        }
        else {
            if ((retval = insert(tree, &bitree_right(*node), data, balanced)) != 0) {
                return retval;
            }
        }
```


Árvore Binária de Busca – Implementação 14/24

```
    // Ensure that the tree remains balanced
    if (!(*balanced)) {
        switch (((AvlNode *)bitree_data(*node))->factor) {
            case AVL_LFT_HEAVY:
                ((AvlNode *)bitree_data(*node))->factor =
    AVL_BALANCED;
                *balanced = 1;
                break;
            case AVL_BALANCED:
                ((AvlNode *)bitree_data(*node))->factor =
    AVL_RGT_HEAVY;
                break;
            case AVL_RGT_HEAVY:
                rotate_right(node);
                *balanced = 1;
            }
    }
} /* if (cmpval > 0) */
```


Árvore Binária de Busca – Implementação 15/24

```
else {
        // Handle finding a copy of the data
        if (!((AvlNode *)bitree_data(*node))->hidden) {
            // Do nothing since the data is in the tree and not hidden
            return 1;
        }
```


Árvore Binária de Busca – Implementação 16/24

```
    else {                  // Insert the new data and mark it as not hidden

            if (tree->destroy != NULL) {
                tree->destroy(((AvlNode *)bitree_data(*node))->data);
            }
            ((AvlNode *)bitree_data(*node))->data = (void *)data;
            ((AvlNode *)bitree_data(*node))->hidden = 0;
            *balanced = 1;

        }

    }

}

return 0;

}
```


Árvore Binária de Busca – Implementação 17/24

```
static int hide(BisTree *tree, BiTreeNode *node, const void *data) {

    int     cmpval,   retval;

    if (bitree_is_eob(node))
        return -1;

    cmpval = tree->compare(data, ((AvlNode *)bitree_data(node))->data);

    if (cmpval < 0)  // Move to the left
        retval = hide(tree, bitree_left(node), data);
    else if (cmpval > 0)  // Move to the right
        retval = hide(tree, bitree_right(node), data);
    else { // Mark the node as hidden
        ((AvlNode *)bitree_data(node))->hidden = 1;
        retval = 0;
    }
return retval;
}
```

Árvore Binária de Busca – Implementação 18/24

```
static int lookup(BisTree *tree, BiTreeNode *node, void **data) {

    int     cmpval,   retval;

    if (bitree_is_eob(node)) {
        // Return that the data was not found
        return -1;
    }

    cmpval = tree->compare(*data, ((AvlNode *)bitree_data(node))->data);

    if (cmpval < 0) {
        // Move to the left
        retval = lookup(tree, bitree_left(node), data);
    }
    else if (cmpval > 0) {
        //  Move to the right
        retval = lookup(tree, bitree_right(node), data);
    }
```


Árvore Binária de Busca – Implementação 19/24

```
    else {
        if (!((AvlNode *)bitree_data(node))->hidden) {
            // Pass back the data from the tree
            *data = ((AvlNode *)bitree_data(node))->data;
            retval = 0;
        }
        else {
            //  Return that the data was not found
            return -1;
        }

    }
    return retval;
}
```


Árvore Binária de Busca – Implementação 20/24

```
void bistree_init(BisTree *tree,
    int (*compare)(const void *key1, const void *key2),
    void (*destroy)(void *data)) {

    bitree_init(tree, destroy);
    tree->compare = compare;
    return;
}
```


Árvore Binária de Busca – Implementação 21/24

```
void bistree_destroy(BisTree *tree) {
    destroy_left(tree, NULL);
    memset(tree, 0, sizeof(BisTree));
    return;
}
```


Árvore Binária de Busca – Implementação 22/24

```
int bistree_insert(BisTree *tree, const void *data) {

    int balanced = 0;
    return insert(tree, &bitree_root(tree), data, &balanced);
}
```


Árvore Binária de Busca – Implementação 23/24

```
int bistree_remove(BisTree *tree, const void *data) {

    return hide(tree, bitree_root(tree), data);
}
```

Árvore Binária de Busca – Implementação 24/24

```
int bistree_lookup(BisTree *tree, void **data) {

    return lookup(tree, bitree_root(tree), data);

}
```


Árvore Binária de Busca – Sumário

- Método eficiente para implementar tabelas
- Esforço mesmo no pior caso é bom
- Mais complexo que tabela por dispersão
- Mas o módulo pode ser implementado uma vez e reusado muitas vezes

- Existem inúmeras variações de árvores de busca
- Com algumas vantagens e desvantagens pontuais
- Mas o princípio é sempre o mesmo:
  Manter a árvore balanceada