Algoritmos e Estruturas de Dados: Árvore Binária

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

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

Referências

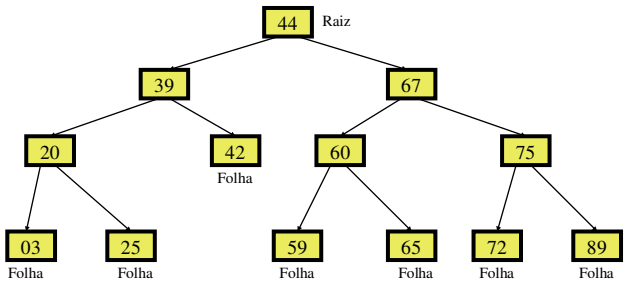
- Mastering Algorithms with C
  Kyle Loudon
  O´Reilly, 1999
- Livros de algoritmos e estruturas de dados em geral

Árvore Binária – Introdução 1/99

- Em computação, uma árvore consiste de elementos chamados nodos organizados hierarquicamente
- Nodo no topo é a raiz
- Os nodos imediatamente abaixo de um certo nodo s ão seus filhos
  - Os quais podem ter seus próprios filhos
- Com exceção da raiz, cada nodo tem exatamente um nodo pai
- O número de filhos que um nodo pode ter depende do tipo da árvore
  - Branching factor
- **Árvores binárias (binary tree)**
  - Branching factor é 2
  - Relativamente simples porém poderosas

Árvore Binária – Introdução 1/99

- Exemplo de árvore binária
  - 39 e 67 são os filhos de 44
  - Altura desta árvore é 4
  - 67 é a raiz da sub-árvore esquerda de 44



Árvore Binária – Introdução 1/99

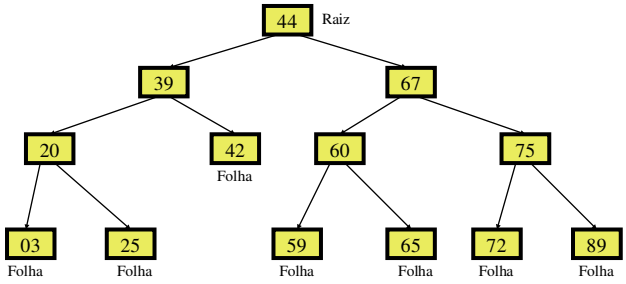
- Cada nodo da árvore binária contem:
  - Dados
  - Pointer para nodo filho da esquerda
  - Pointer para nodo filho da direita
- Caso não tenha um filho, NULL é usado para indicar isto
- Uma sub-árvore pode ser identificada pelo seu nodo raiz
- Um conjunto de árvores é uma floresta

Árvore Binária – Introdução 1/99

- **Métodos de Caminhamento**
- Caminhar sobre a árvore significa visitar todos os nodos
  - Visitar significa acessar o nodo de alguma forma
- O método de caminhamento define a ordem das visitas
  - No caso de uma lista encadeada não existem muitas opções
  - Mas existem várias opções no caso de uma árvore
- Métodos mais usuais:
  - Pré-ordem
  - Em-ordem
  - Pós-ordem
  - Por nível
- Podem ser usados para outros tipos de árvores também

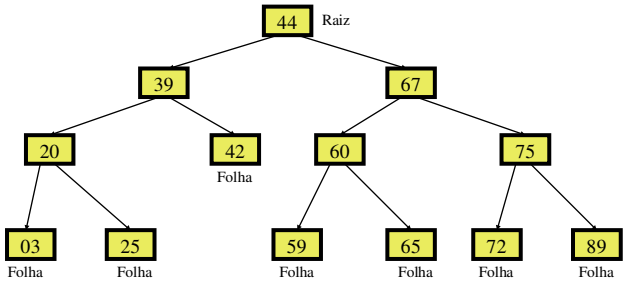
Árvore Binária – Introdução 1/99

- Método de Caminhamento: **Pré-Ordem**
  - Raiz, Esquerda, Direita
  - 44, 39, 20, 03, 25, 42, 67, 60, 59, 65, 75, 72, 89



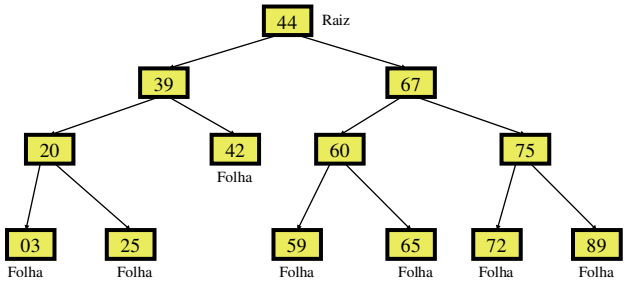
Árvore Binária – Introdução 1/99

- Método de Caminhamento: **Em-Ordem**
  - Esquerda, Raiz, Direita
  - 03, 02, 25, 39, 42, 44, 59, 60, 65, 67, 72, 75, 89



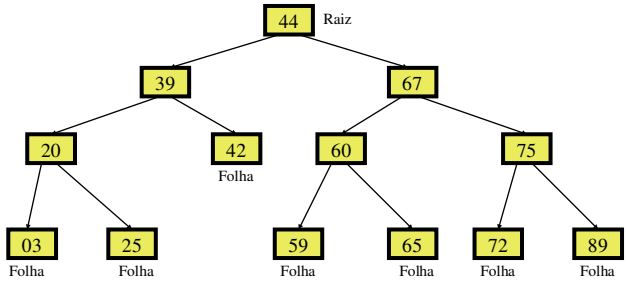
Árvore Binária – Introdução 1/99

- Método de Caminhamento: **Pós-Ordem**
  - Esquerda, Direita, Raiz
  - 03, 25, 20, 42, 39, 59, 65, 60, 72, 89, 75, 67, 44



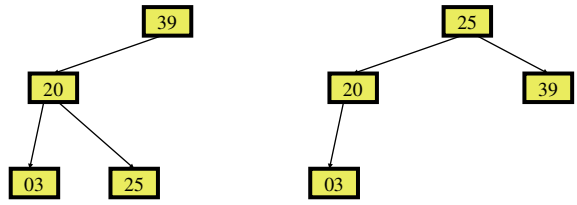
Árvore Binária – Introdução 1/99

- Método de Caminhamento: **Por Nível**
  - Raiz, e depois todos os do nível abaixo, e assim por diante
  - 44, 39, 67, 20,42,60,75, 03,25,59,65,72,89



Árvore Binária – Introdução 1/99

- **Balançeamento**
  - Balançear a árvore significa deixa-la com a menor altura possível, para um dado número de nodos
  - Um nível precisa estar cheio para ser criado o nível de baixo
- Uma árvore binária está balanceada se a altura das duas sub-árvores de cada nodo nunca diferem por mais que 1



Árvore Binária – Interface 1/14

- ***bitree_init***
- void bitree_init(BiTree **tree*, void (**destroy*)(void **data*));
- **Retorno**
  - Nenhum.
- **Descrição**
  - Inicializa uma árvore binária especificada por *tree*. Esta operação deve ser chamada para uma árvore binária antes de qualquer operação. O argumento *destroy* provê uma maneira para liberar dados dinamicamente alocados quando *bitree_destroy* é chamada. Por exemplo, se a árvore contem dados dinamicamente alocados usando *malloc*, *destroy* deveria ser associada com *free* para liberar os dados quando a árvore binária for destruida. Para dados estruturados contendo diversos membros alocados dinamicamente, *destroy* deveria ser associada com uma função definida pelo usuário a qual chama *free* para cada membro alocado dinamicamente assim como para a estrutura ela própria. Para uma árvore binária contendo dados que não devem ser liberados, *destroy* deve ser NULL.

Árvore Binária – Interface 2/14

- ***bitree_destroy***
- void bitree_destroy(BiTree *tree);
- **Retorno**
  - Nenhum.
- **Descrição**
  - Destrói a árvore binária especificada por *tree*. Nenhuma outra operação é permitida após a chamada de *bitree_destroy* a não ser que *bitree_init* seja chamada novamente. A operação *bitree_destroy* remove todos os nodos de uma árvore binária e chama a função passada como *destroy* para *bitree_init* uma vez para cada nodo quando o mesmo é removido, desde que *destroy* não seja NULL.

Árvore Binária – Interface 3/14

- ***bitree_ins_left***
- int bitree_ins_left(BiTree *tree, BiTreeNode *node, const void *data);
- **Retorno**
  - 0 se a inserção do nodo tiver sucesso, ou -1 caso contrário.
- **Descrição**
  - Insere um nodo como o filho da esquerda do nodo *node* na árvore binária especificada por *tree*. Se *node* já tem um filho esquerdo, *bitree_ins_left* retorna -1. Se *node* for NULL, o novo nodo é inserido como o nodo raiz. A árvore deve estar vazia para ser possível inserir um nodo como o nodo raiz; caso contrário, *bitree_ins_left* retorna -1. Quando sucesso, o novo nodo contem um pointer para od dados, logo a memória referenciada por *data* deve permanecer válida pelo tempo que o nodo permanecer na árvore binária. É responsabilidade do chamador gerenciar a memória associada com os dados.

Árvore Binária – Interface 4/14

- ***bitree_ins_right***
- int bitree_ins_right(BiTree *tree, BiTreeNode *node, const void *data);
- **Retorno**
  - 0 se a inserção do nodo tiver sucesso, ou -1 caso contrário.
- **Descrição**
  - Esta operação é similar a *bitree_ins_left*, exceto que ela insere um nodo como o filho direito de *node* na árvore binária especificada por *tree*.

Árvore Binária – Interface 5/14

- ***bitree_rem_left***
- void bitree_rem_left(BiTree *tree, BiTreeNode *node);
- **Retorno**
  - Nenhum.
- **Descrição**
  - Remove a subárvore cuja raiz é o filho esquerdo de *node* da árvore binária especificada por *tree*. Se *node* for NULL, todos os nodos da árvore são removidos. A função passada como *destroy* para *bitree_init* é chamada uma vez para cada nodo que é removido, desde que *destroy* não seja NULL.

Árvore Binária – Interface 6/14

- ***bitree_rem_right***
- void bitree_rem_right(BiTree *tree, BiTreeNode *node);
- **Retorno**
  - Nenhum.
- **Descrição**
  - Esta operação é similar a *bitree_rem_left*, exceto que ela remove a sub-árvore cuja raiz é o filho direito de *node* da árvore binária especificada por *tree*.

Árvore Binária – Interface 7/14

- ***bitree_merge***
- int bitree_merge(BiTree *merge, BiTree *left, BiTree *right, const void *data);
- **Retorno**
  - 0 se a junção das árvores teve sucesso, ou -1 caso contrário.
- **Descrição**
  - Junta as duas árvores binárias especificadas por *left* e *right* em uma única árvore binária *merge*. Após a junção estar completa, *merge* contem *data* em seu nodo raiz, e *left* e *right* são as sub-árvores esquerda e direita de sua raiz. Uma vez que as árvores foram juntadas, *left* e *right* estarão como se *bitree_destroy* tivesse sido chamada para elas.

Árvore Binária – Interface 8/14

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

Árvore Binária – Interface 9/14

- ***bitree_root***
- BiTreeNode *bitree_root(const BiTree **tree*);
- **Retorno**
  - Nodo na raiz da árvore.
- **Descrição**
  - Macro que obtem o nodo na raiz da árvore binária especificada por *tree*.

Árvore Binária – Interface 10/14

- ***bitree_is_eob***
- int bitree_is_eob(const BiTreeNode **node*);
- **Retorno**
  - 1 se o nodo marca o final de um galho, or 0 caso contrário.
- **Descrição**
  - Macro que determina se o nodo especificado por *node* marca o final de um galho na árvore binária.

Árvore Binária – Interface 11/14

- ***bitree_is_leaf***
- int bitree_isleaf(const BiTreeNode *node);
- **Retorno**
  - 1 se o nodo é uma folha, ou 0 caso contrário.
- **Descrição**
  - Macro que determina se o nodo especificado por *node* é uma folha na árvore binária.

Árvore Binária – Interface 12/14

- ***bitree_data***
- void *bitree_data(const BiTreeNode **node*);
- **Retorno**
  - Dados armazenados em um nodo.
- **Descrição**
  - Macro que obtem os dados armazenados no nodo de uma árvore binária especificado por *node*.

Árvore Binária – Interface 13/14

- ***bitree_left***
- BiTreeNode *bitree_left(const BiTreeNode **node*);
- **Retorno**
  - Filho esquerdo da nodo especificado.
- **Descrição**
  - Macro que obtem o nodo de uma árvore binária que é o filho esquerdo de um nodo especificado por *node*.

Árvore Binária – Interface 14/14

- ***bitree_right***
- BiTreeNode *bitree_right(const BiTreeNode **node*);
- **Retorno**
  - Filho direito do nodo especificado.
- **Descrição**
  - Macro que obtem o nodo de uma árvore binária o qual é o filho direito do nodo especificado por *node*.

Árvore Binária – Header 1/3

```
#ifndef BITREE_H
#define BITREE_H

#include <stdlib.h>

typedef struct BiTreeNode_ {
    void                *data;
    struct BiTreeNode_  *left;
    struct BiTreeNode_  *right;
} BiTreeNode;

typedef struct BiTree_ {
    int             size;
    int             (*compare)(const void *key1, const void *key2);
    void            (*destroy)(void *data);
    BiTreeNode      *root;
} BiTree;
```


Árvore Binária – Header 2/3

```
void bitree_init(BiTree *tree, void (*destroy)(void *data));

void bitree_destroy(BiTree *tree);

int bitree_ins_left(BiTree *tree, BiTreeNode *node, const void *data);

int bitree_ins_right(BiTree *tree, BiTreeNode *node, const void *data);

void bitree_rem_left(BiTree *tree, BiTreeNode *node);

void bitree_rem_right(BiTree *tree, BiTreeNode *node);

int bitree_merge(BiTree *merge, BiTree *left, BiTree *right, const void *data);
```


Árvore Binária – Header 3/3

```
#define bitree_size(tree) ((tree)->size)

#define bitree_root(tree) ((tree)->root)

#define bitree_is_eob(node) ((node) == NULL)

#define bitree_is_leaf(node) ((node)->left == NULL && (node)->right == NULL)

#define bitree_data(node) ((node)->data)

#define bitree_left(node) ((node)->left)

#define bitree_right(node) ((node)->right)

#endif
```


Árvore Binária – Implementação 1/9

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

#include "bitree.h"

void bitree_init(BiTree *tree, void (*destroy)(void *data)) {
    tree->size = 0;
    tree->destroy = destroy;
    tree->root = NULL;
    return;
}
```


Árvore Binária – Implementação 2/9

```
void bitree_destroy(BiTree *tree) {

    bitree_rem_left(tree, NULL);
    memset(tree, 0, sizeof(BiTree));
    return;
}
```

Árvore Binária – Implementação 3/9

```
int bitree_ins_left(BiTree *tree, BiTreeNode *node, const void *data) {

    BiTreeNode      *new_node,      **position;

    if (node == NULL) {         // Allow insertion at the root only in an empty
    tree.
    if (bitree_size(tree) > 0)      return -1;
    position = &tree->root;
    }
    else {          // Normally allow insertion only at the end of a branch.

    if (bitree_left(node) != NULL)      return -1;
    position = &node->left;
    }
```


Árvore Binária – Implementação 4/9

```
    if ((new_node = (BiTreeNode *)malloc(sizeof(BiTreeNode))) == NULL)
        return -1;

    new_node->data = (void *)data;
    new_node->left = NULL;
    new_node->right = NULL;
    *position = new_node;

    tree->size++;
    return 0;
}
```


Árvore Binária – Implementação 5/9

```
int bitree_ins_right(BiTree *tree, BiTreeNode *node, const void *data) {

    BiTreeNode      *new_node,      **position;

    if (node == NULL) {         // Allow insertion at the root only in an empty
    tree.
    if (bitree_size(tree) > 0)      return -1;
    position = &tree->root;
    }
    else {          // Normally allow insertion only at the end of a branch.

    if (bitree_right(node) != NULL)     return -1;
    position = &node->right;
    }
```


Árvore Binária – Implementação 6/9

```
    if ((new_node = (BiTreeNode *)malloc(sizeof(BiTreeNode))) == NULL)
        return -1;

    new_node->data = (void *)data;
    new_node->left = NULL;
    new_node->right = NULL;
    *position = new_node;

    tree->size++;

    return 0;
}
```


Árvore Binária – Implementação 7/9

```
void bitree_rem_left(BiTree *tree, BiTreeNode *node) {
    BiTreeNode      **position;
    if (bitree_size(tree) == 0)     return;

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

    if (*position != NULL) {
        bitree_rem_left(tree, *position);
        bitree_rem_right(tree, *position);

        if (tree->destroy != NULL)
            tree->destroy((*position)->data);

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


Árvore Binária – Implementação 8/9

```
void bitree_rem_right(BiTree *tree, BiTreeNode *node) {
    BiTreeNode      **position;
    if (bitree_size(tree) == 0)     return;

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

    if (*position != NULL) {
        bitree_rem_left(tree, *position);
        bitree_rem_right(tree, *position);

        if (tree->destroy != NULL)
            tree->destroy((*position)->data);

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

Árvore Binária – Implementação 9/9

```
int bitree_merge(BiTree *merge, BiTree *left, BiTree *right, const void *data) {

    bitree_init(merge, left->destroy);

    if (bitree_ins_left(merge, NULL, data) != 0) {
        bitree_destroy(merge);
        return -1;
    }

    bitree_root(merge)->left = bitree_root(left);
    bitree_root(merge)->right = bitree_root(right);

    merge->size = merge->size + bitree_size(left) + bitree_size(right);

    left->root = NULL;
    left->size = 0;
    right->root = NULL;
    right->size = 0;

    return 0;
}
```


Árvore Binária – Caminhamento 1/3

```
int preorder(const BiTreeNode *node, List *list) {

    if (!bitree_is_eob(node)) {

        if (list_ins_next(list, list_tail(list), bitree_data(node)) != 0)
            return -1;

        if (!bitree_is_eob(bitree_left(node)))
            if (preorder(bitree_left(node), list) != 0)
                return -1;

        if (!bitree_is_eob(bitree_right(node)))
            if (preorder(bitree_right(node), list) != 0)
                return -1;
    }
    return 0;
}
```


Árvore Binária – Caminhamento 2/3

```
int inorder(const BiTreeNode *node, List *list) {

    if (!bitree_is_eob(node)) {
        if (!bitree_is_eob(bitree_left(node)))
            if (inorder(bitree_left(node), list) != 0)
                return -1;

        if (list_ins_next(list, list_tail(list), bitree_data(node)) != 0)
            return -1;

        if (!bitree_is_eob(bitree_right(node)))
            if (inorder(bitree_right(node), list) != 0)
                return -1;
    }

return 0;
}
```


Árvore Binária – Caminhamento 3/3

```
int postorder(const BiTreeNode *node, List *list) {

    if (!bitree_is_eob(node)) {

        if (!bitree_is_eob(bitree_left(node)))
            if (postorder(bitree_left(node), list) != 0)
                return -1;

        if (!bitree_is_eob(bitree_right(node)))
            if (postorder(bitree_right(node), list) != 0)
                return -1;

        if (list_ins_next(list, list_tail(list), bitree_data(node)) != 0)
            return -1;
    }
    return 0;
}
```


Árvore Binária – Sumário

- Permite várias operações interessantes
  - Mas exige que o balaçeamento seja bom
- Muitas operações ficam simples quando recursão é usada
  - Mas exige espaço em pilha
- Capaz de garantir tempo de busca determinista
  - Tabela hash fornece tempo de busca probabilista
- Existe o gasto de 2 pointers para cada nodo com dados