Árvore AVL em C++

NOVEMBER 22, 2008 / CESARSOUZA / 17 COMMENTS

Aqui está um código em C++ para uma Árvore AVL visando fins mais educacionais do que práticos. Apesar da teoria ser muito importante, não adianta nada ler pseudo-algorítmos sem saber como a teoria pode ser, de fato, aplicada. Ainda mais quando a maioria dos livros-texto **deixa o método de remoção como exercício**. Que mania!

A Árvore apresentada abaixo utiliza apenas uma técnica que pode parecer estranha, a primeira vista, para quem está acostumado a tratar com árvores em algoritmo puro. As subárvores esquerda e direita, presentes em cada nó, estão dispostas em um vetor de dois elementos. Isto traz uma vantagem imensa na hora de rotacionar a árvore, pois **corta a necessidade de métodos auxiliares pela metade**. Ao invés de termos de implementar rotações à esquerda e depois a direita, por exemplo, aproveitamos um mesmo método para realizar a operação de rotação em ambas as direções.

Fora este recurso adicional, o restante da implementação é bem similar aos textbooks que encontramos por aí. O nó não precisa de uma referência adicional ao seu pai (que é desnecessária, por sinal) e guarda apenas seu fator de balancemento (outras implementações guardam sua altura). O código é orientado a objetos (nada de código C disfarçado), e não está sobrecarregado com templates, embora qualquer um possa modificar o código de forma a suportá-los facilmente.

Introdução

Antes de continuar a leitura, recomendo fortemente que você dê uma olhada na teoria e tenha ao menos obtido uma noção de como a Árvore AVL funciona. Se possível, tente implementá-la uma vez para ter idéia de onde estão suas maiores dificuldades e compreender seu funcionamento. Este artigo não o ensinará o porquê da rotação, ou o porquê do balanceamento; apenas indica como ficaria uma implementação em C++ (ultra-simplificada, perto do que é usado no mundo real) destas estruturas.

Para saber mais, tanto sobre árvores como outras estruturas de dados em computação, recomendo o livro virtual do Prof. Dr. Roberto Ferrari, do curso de Estruturas de Dados do Departamento de Computação da Universidade Federal de São Carlos, que pode ser acessado gratuitamente clicando-se aqui. Você também pode ler mais sobre Árvores AVL na Wikipedia. Se possível leia na Wikipedia em inglês, que geralmente é mais completa.

Bom, agora que já temos a teoria passada, podemos continuemos em frente!

O Nó

Comecemos primeiro pela definição do Nó, que vamos apelidar aqui de **TreeNode**.

```
// Nó da Árvore AVL
//  Esta classe está implementada para trabalhar com chaves do tipo
//  string e valores associados às chaves também do tipo string. No
//  entanto, esta classe pode ser facilmente adaptada para trabalhar
//  com templates. Mantida nessa forma apenas para uma maior didática.
class TreeNode
{
    public:
    // Constrói um novo nó a partir de um item e uma chave
    TreeNode(const string& chave, const string& value)
    {
        this->Key   = chave;
        this->Value = value;
        this->Subtree[LEFT]  = NULL;
        this->Subtree[RIGHT] = NULL;
        this->Balance = EQUAL;
    }
    // Destrutor
    ~TreeNode()
    {
        // Note que destruição de um nó iniciará uma operação
        //   em cadeia em todos os seus filhos. Caso um dos
        //   filhos não deva ser apagado, deve-se setar NULL
        //   em sua referência antes de destruir este nó.
        delete this->Subtree[LEFT];
        delete this->Subtree[RIGHT];
    }
    // Balanceamento do nó
    unsigned short Balance;
    // Filhos da árvore
    // As subárvores direita e esquerda devem ser acessadas
    // utilizando apenas ->Subtree[LEFT] e ->Subtree[RIGHT]
    TreeNode* Subtree[2];
```

```
        // Arranjo <chave, valor>
        //  associado ao nó
        string Key;
        string Value;
};
```

Os acessores estão omitidos para uma maior clareza, mas também poderiam ser desnecessários caso TreeNode fosse implementada com construtor privado e declarada **friend class** de **AvlTree** (classe principal da Árvore AVL, a ser apresentada na seqüência).

Observe também que faltou dizer de onde vieram LEFT, RIGHT e EQUAL, que são na verdade originários da enumeração Directions, apresentada abaixo. É este sistema que nos permitirá cortar o número de métodos de rotação pela metade.

```
// Enumeração para acesso às subárvores e direção
//  de procura da pesquisa binária. Serve também
//  para indicar o fator de balanceamento dos nós.
enum Direction
{
    // Indica que a subárvore esquerda tem maior altura,
    // Que devemos caminhar para a esquerda na pesquisa
    // ou que queremos acessar a Subárvore da esquerda.
    LEFT  = 0,
    // Indica que a subárvore direita tem maior altura,
    // Que devemos caminhar para a direita na pesquisa
    // ou que queremos acessar a Subárvore da direita.
    RIGHT = 1,
    // Indica que as subárvore tem alturas iguais.
    // Não é válida nos outros contextos.
    EQUAL = 2,
};
```

A Árvore

Agora, apresentamos a classe da Árvore AVL propriamente dita.

```
// Classe Árvore AVL
//  Fornece métodos para criar e manipular Árvores AVL.
//  Trabalha com nós cuja chave e informação são do tipo string,
//  mas a classe poderia ser facilmente adaptada para trabalhar
//  com templates. Mantida nesta forma para uma maior didática.
class AvlTree
{
    public:
        // Construtor e destrutor padrões
        AvlTree();
        ~AvlTree();
```

```
    // Insere uma nova entrada na árvore, aceitando
    //   um arranjo chave e valor como parâmetros
    void Insert(const string& key, const string& value);
    // Remove uma entrada da árvore, aceitando
    //   um arranjo chave e valor como parâmetros
    //   Retorna falso caso a palavra não exista.
    bool Remove(const string& value);
    // Procura um valor na árvore, aceitando uma  chave como
    //  parâmetro. Se encontrado, o valor será armazenado na
    //  variável value e retornado como referência. Retorna
    //  verdadeiro se encontrar o valor, falso caso contrário.
    bool Search(const string& key, string& value);
    // Limpa a árvore
    void Clear();
    // Exibe a árvore na tela
    void Print();
    void PrintNodeDetails();
    // Carrega uma árvore do arquivo, aceitando o caminho do arquivo
    //  (path) como parâmetro. Caso esta não exista, retorna NULL
    static AvlTree* Load(const char* path);
    // Armazena uma árvore em arquivo, aceitando o caminho do arquivo
    //  (path) como parâmetro. O arquivo é armazenado em pós-ordem.
    bool Save(const char* path);
private:
    // Raiz da árvore
    TreeNode* root;
    // Rotações
    void rotateTwice(TreeNode*& node, Direction dir);
    void rotateOnce (TreeNode*& node, Direction dir);
    // Rebalanceamentos
    void updateBalance  (TreeNode* tree,  Direction dir);
    void rebalanceInsert(TreeNode*& tree, Direction dir,
                         bool& hChanged);
    void rebalanceRemove(TreeNode*& tree, Direction dir,
                         bool& hChanged);
    // Inserção e remoção recursivos
    void insert(const string& key, const string& value,
                TreeNode*& node, bool& hChanged);
    bool remove(const string& key, TreeNode*& node, bool& hChanged);
    // Métodos auxiliares
    bool save(ofstream& stream, TreeNode* node);
    void printNodeDetails(TreeNode* node);
    inline Direction opposite(Direction dir);
};
```

Construção e Pesquisa

Vamos começar pela parte mais fácil, como os métodos triviais construtores e destrutores e a straightforward *pesquisa binária*. A pesquisa poderia ser otimizada para apenas algumas poucas linhas, se fizessemos uso correto de nossa implementação de subárvores em vetores de duas posições, mas é melhor deixá-la assim para simplificar o entendimento.

```
    // Construtor
    AvlTree::AvlTree()
    {
        // Inicializamos a raíz com null.
        this->root = NULL;
    }
    // Destrutor
    AvlTree::~AvlTree()
    {
        this->Clear();
    }
    // Limpa a árvore (remove todos seus nós)
    void AvlTree::Clear()
    {
        // O destrutor de TreeNode cuidará
        //  de destruir todas as subárvores.
        delete root;
        this->root = NULL;
    }


// Procura um item na árvore, realizando pesquisa binária.
//   aceita como parâmetro uma string como chave de busca.
//   retorna o item procurado ou NULL caso não o encontre.
bool AvlTree::Search(const string& chave, string& value)
{
    // Começamos com a raiz
    TreeNode* current = this->root;
    // Como, por definição, a árvore AVL está
    //  ordenada, utilizamos a pesquisa binária
    while (current != NULL)
    {
        // Comparamos
        if (chave > current->Key)
        {
            // O valor está mais à direita
            current = current->Subtree[RIGHT];
        }
        else if (chave < current->Key)
        {
            // O valor está mais à esquerda
            current = current->Subtree[LEFT];
        }
        else
        {
            // Achamos, retornamos o item
            value = current->Value;
```

```
            return true;
        }
    }
    // O valor não estava na árvore
    return false;
}
```

Rotacionando

Agora, apresentamos as rotações. Observe como o sistema de direção ajuda nesta hora:

```
// Realiza uma rotação simples numa determinada direção.
//  aceita como parâmetro um nó onde a operação será efetuada
//  e uma direção (esquerda ou direita) para realizar a rotação.
void AvlTree::rotateOnce(TreeNode*& node, Direction dir)
{
    int opposite = this->opposite(dir);
    TreeNode* child = node->Subtree[dir];
    node->Subtree[dir] = child->Subtree[opposite];
    child->Subtree[opposite] = node;
    node = child;
}
// Realiza uma rotação dupla numa determinada direção.
//  aceita como parâmetro um nó onde a operação será efetuada
//  e uma direção (esquerda ou direita) para realizar a rotação.
void AvlTree::rotateTwice(TreeNode*& node, Direction dir)
{
    int opposite = this->opposite(dir);
    TreeNode* child = node->Subtree[dir]->Subtree[opposite];
    node->Subtree[dir]->Subtree[opposite] = child->Subtree[dir];
    child->Subtree[dir] = node->Subtree[dir];
    node->Subtree[dir] = child;
    child = node->Subtree[dir];
    node->Subtree[dir] = child->Subtree[opposite];
    child->Subtree[opposite] = node;
    node = child;
}
```

Pronto! Se você reparar bem, verá que estes são exatamente os mesmos passos das rotações convencionais, porém generalizados, já que, uma vez que os métodos são simétricos, basta inverter a direção de todos os acessos (i.e. esquerda por direita) que tudo dará certo. A função opposite, utilizada logo no início dos algorítmos, realiza justamente esta inversão de direção:

```
// Retorna a direção oposta à direção dada
```

```
inline Direction AvlTree::opposite(Direction dir)
{
    return (dir == RIGHT) ? LEFT : RIGHT;
}
```

Balanceando

Bom, até aí, rotacionar é parte fácil… A coisa começa a complicar na hora de atualizar os fatores de balanceamento. Sempre após uma mudança de altura, devemos chamar o método **updateBalance()** para atualizá-los. Quem chamará este método serão os métodos de rebalancemanto após inserção ou remoção de um nó da árvore, sempre que a altura da árvore (ou de uma subárvore) mudar. Tais métodos são implementados como à seguir:

```
// Atualiza os fatores de balanceamento após uma rotação
void AvlTree::updateBalance(TreeNode* node, Direction dir)
{
    int opposite = this->opposite(dir);
    int bal = node->Subtree[dir]->Subtree[opposite]->Balance;
    // Se o fator está pesado no mesmo lado
    if (bal == dir)
    {
        node->Balance = EQUAL;
        node->Subtree[dir]->Balance = opposite;
    }
    // Se o fator está pesado do lado oposto
    else if (bal == opposite)
    {
        node->Balance = dir;
        node->Subtree[dir]->Balance = EQUAL;
    }
    else // O fator de balanceamento está igual
    {
        node->Balance = node->Subtree[dir]->Balance = EQUAL;
    }
    node->Subtree[dir]->Subtree[opposite]->Balance = EQUAL;
}
// Efetua o rebalanceamento após uma operação de inserção
void AvlTree::rebalanceInsert(TreeNode*& node, Direction dir,
                              bool& hChanged)
{
    int opposite = this->opposite(dir);
    // Se o fator de balanceamento do nó era
    //  igual a direção em que houve a inserção,
    if (node->Balance == dir) // (ou seja o nó foi inserido
    {                         //  na subárvore de maior altura
        // Temos 2 casos:
```

```
    if (node->Subtree[dir]->Balance == dir)
    {
        node->Subtree[dir]->Balance = 2;
        node->Balance = EQUAL;
        // Precisamos fazer uma rotação
        rotateOnce(node, dir);
    }
    else // Seu filho estava equilibrado ou
    {    //  pendendo para o lado oposto,
        updateBalance(node, dir);
        rotateTwice(node, dir);
    }
    hChanged = false;
}
// Já se foi exatamente o oposto,
else if (node->Balance == opposite)
{
    // O nó agora está balanceado
    node->Balance = 2;
    hChanged = false;
}
else // Se não, o nó já estava equilibrado
{
    // e agora seu equilibrio foi deslocado.
    node->Balance = dir;
}
}
// Efetua o rebalanceamento após uma operação de remoção
void AvlTree::rebalanceRemove(TreeNode*& node, Direction dir,
                              bool& hChanged)
{
    Direction opposite = this->opposite(dir);
    // Se o fator de balanceamento do nó era
    //  igual a direção em que houve a remoção,
    if (node->Balance == dir) // (ou seja, o nó foi removido
    {                         //da subárvore de maior altura)
        // o nó agora está balanceado.
        node->Balance = EQUAL;
    }
    // Já se era o oposto,
    else if (node->Balance == opposite)
    {
        // Temos 3 casos:
        if (node->Subtree[opposite]->Balance == opposite)
        {
            // Se o filho do outro lado estava "mais pesado"
            //  do lado oposto, então
            node->Subtree[opposite]->Balance = EQUAL;
            node->Balance = EQUAL;
            rotateOnce(node, opposite);
        }
```

```
        else if (node->Subtree[opposite]->Balance == EQUAL)
        {
            // Se o filho do outro lado estava equilibrado,
            node->Subtree[opposite]->Balance = dir;
            rotateOnce(node, opposite);
        }
        else
        {
            // Se o filho do outro lado estava "mais pesado"
            //  do lado mais próximo de onde foi feita a inserção,
            updateBalance(node, opposite);
            rotateTwice(node, opposite);
        }
        hChanged = false;
    }
    else
    {
        node->Balance = opposite;
        hChanged = false;
    }
}
```

Inserindo

Agora, definidos todos nossos métodos auxiliares, o método de inserção se torna relativamente pequeno…

```
// Insere um novo item na árvore, aceitando seus componentes
//   palavra (chave) e origem (dado)
void AvlTree::Insert(const string& chave, const string& value)
{
    bool hChanged = false; // Para passagem por referência
    // Chamamos o método recursivo para realizar a operação
    this->insert(chave, value, this->root, hChanged);
}
// Insere um novo item na árvore (recursivo)
void AvlTree::insert(const string& chave, const string& value,
                     TreeNode*& node, bool& hChanged)
{
    // Se o nó atual é nulo,
    if (node == NULL)
    {
        // é aqui mesmo onde vamos inserir
        node = new TreeNode(chave, value);
        // marcamos que a altura mudou, é
        //  preciso chegar o balanceamento
        hChanged = true;
    }
    else if (node->Key == chave)
    {
```

```
        // A informação já estava na árvore
        return;
    }
    else // Ainda não chegamos onde queríamos
    {
        // Prosseguimos com a pesquisa binária
        Direction dir = (chave > node->Key) ? RIGHT : LEFT;
        hChanged = false; // preparamos e chamamos recursão
        insert(chave, value, node->Subtree[dir], hChanged);
        if (hChanged) // Se a altura mudou,
        {
            // Efetuamos o rebalanceamento
            this->rebalanceInsert(node, dir, hChanged);
        }
    }
}
```

Removendo

E por conseguinte, o método de remoção, que é tradionalmente ignorado pelos livros-texto, por dizerem ser muito complicado, fora do escopo, entre outras baboseiras, também fica (um pouco) mais fácil.

```
// Remove um item da árvore a partir de sua chave (palavra)
//  retorna true caso o item tenha sido removido com sucesso,
//  falso caso o item não tenha sido encontrado na árvore.
bool AvlTree::Remove(const string& palavra)
{
    bool hChanged = false; // para passagem por referência
    // Chamamos o método recursivo para realizar a operação.
    return this->remove(palavra, this->root, hChanged);
}
// Remove um item da árvore a partir de sua chave (palavra)
//  retorna true caso o item tenha sido removido com sucesso,
//  falso caso o item não tenha sido encontrado na árvore. (recursivo)
bool AvlTree::remove(const string& chave, TreeNode*& node,
                     bool& hChanged)
{
   // debug("Tree: remocao recursiva: " + chave + ", " + node->Key);
    bool success = false;
    // O nó não foi encontrado
    if (node == NULL)
    {
        hChanged = false;
        return false;
    }
```

```
// Achamos o nó
else if (chave == node->Key)
{
    // Se o nó tiver ambos os filhos
    if (node->Subtree[LEFT] != NULL && node->Subtree[RIGHT] != NULL )
    {
        // Encontraremos um substituto para o nó
        TreeNode* substitute = node->Subtree[LEFT];
        // Navegamos até o nó mais a direita da subárvore da esquerda
        while (substitute->Subtree[RIGHT] != NULL)
        {
            substitute = substitute->Subtree[RIGHT];
        }
        // Trocamos suas informações <chave,valor>
        node->Key   = substitute->Key;
        // Chamamos recursão para remover o nó
        success = remove(node->Key, node->Subtree[LEFT], hChanged);
        if (hChanged) // Se a altura mudou,
        {
            // Rebalanceamos
            rebalanceRemove(node, LEFT, hChanged);
        }
    }
    else // O nó tem apenas um ou nenhum filho
    {
        // Preparamos para apagar o nó
        TreeNode* temp = node;
        // Vemos se um dos seus filhos é diferente de NULL
        Direction dir = (node->Subtree[LEFT] == NULL) ? RIGHT : LEFT;
        // Substituímos o nó por um dos seus filhos
        node = node->Subtree[dir];
        // Deletamos o nó (lembramos que devemos
        //  setar os filhos para null para que o
        //  destrutor não inicie a destrução em cadeia
        temp->Subtree[LEFT]  = NULL;
        temp->Subtree[RIGHT] = NULL;
        delete temp;
        // A altura mudou, propagamos a mudança
        hChanged = true;
    }
    // Retornamos verdadeiro, achamos o nó
    //  e a operação foi concluída
    return true;
}
else // Ainda não encontramos o nó
{
    // Continuamos procurando (pesquisa binária)
    Direction dir = (chave > node->Key) ? RIGHT : LEFT;
    if (node->Subtree[dir] != NULL)
    {
        // Recursão para a direção onde deverá estar a chave
```

```
            success = this->remove(chave, node->Subtree[dir], hChanged);
        }
        else
        {
            // O nó não está na árvore
            hChanged = false;
            return false;
        }
        if (hChanged) // Se a altura mudou,
        {
            // Rebalanceamos
            this->rebalanceRemove(node, dir, hChanged);
        }
        // Retornamos informando se achamos o nó ou não
        return success;
    }
}
```

Métodos Adicionais

Tcharam! O essencial é isto. Se quisermos ir além, podemos definir alguns métodos adicionais, para, por exemplo, imprimir detalhes sobre os nós, salvar a árvore em disco e recuperá-la de um arquivo.

```
// Exibe a árvore na tela utilizando arte ASCII. Árvores
//  muito grandes ou esparsas terão dificuldade para
//  caberem no console.
void AvlTree::Print()
{
    // Verificamos se há árvore
    if (this->root != NULL)
    {
        // Instanciamos um novo objeto da classe AsciiTreeView
        AsciiTreeView treeView(this);
        // Solicitamos a impressão.
        treeView.Print();
    }
    else // A árvore está vazia
    {
        cout << "Arvore vazia" << endl;
    }
}
// Imprime detalhes sobre cada nó da árvore,
//  percorrendo-a em ordem.
void AvlTree::PrintNodeDetails()
{
```

```
    this->printNodeDetails(this->root);
}
// Imprime detalhes sobre cada nó da árvore,
//  percorrendo-a em ordem (recursivo).
void AvlTree::printNodeDetails(TreeNode* node)
{
    if (node != NULL)
    {
        this->printNodeDetails(node->Subtree[LEFT]);
        cout << "Node Key: "  << node->Key    << endl;
        cout << "- Origem: "  << node->Value   << endl;
        cout << "- Balance: " << node->Balance << endl;
        if (node->Subtree[LEFT] != NULL)
        cout << "- Left: "  << node->Subtree[LEFT]->Key << endl;
        if (node->Subtree[RIGHT] != NULL)
        cout << "- Right: " << node->Subtree[RIGHT]->Key << endl;
        cout << endl;
        this->printNodeDetails(node->Subtree[RIGHT]);
    }
}
// ------------------
// Métodos para Salvar & Carregar
// ------------------
// Carrega uma árvore a partir de um arquivo, aceitando
//  como entrada o seu caminho.
AvlTree* AvlTree::Load(const char* path)
{
    // Constrói a árvore a partir do arquivo, construindo
    //  a árvore diretamente já que os items no arquivo
    //  foram armazenados em pré-ordem.
    //  (e portanto o carregamento será da ordem de O(n)
    ifstream stream(path);
    if (stream.is_open())
    {
        AvlTree* tree = new AvlTree();
        string chave, value;
        while (!stream.eof())
        {
            // Lemos os valores:
            //  primeiro a palavra, depois a origem
            stream >> chave;
            stream >> value;
            if (stream.fail()) break;
            // Inserimos na árvore. Como estamos lendo
            //  em pré-ordem, esta operação dura O(1)
            tree->Insert(chave, value);
        }
        stream.close();
        return tree;
    }
    return NULL;
```

```
}
// Salva a árvore para um arquivo em disco, aceitando
//  uma cadeia de caracteres indicando seu caminho.
//  Retorna true em caso de sucesso, falso caso contrário.
bool AvlTree::Save(const char* path)
{
    // Se a árvore está vazia,
    if (this->root == NULL)
    {
        // Apagamos o arquivo.
        std::remove(path);
        return true;
    }
    // Armazenamos as entradas no arquivo, salvando
    //  as entradas em pré-ordem, primeiro a palavra,
    //  depois a origem.
    ofstream stream(path);
    if (stream.is_open())
    {
        // Chamamos função recursiva
        this->save(stream, this->root);
        stream.close();
        // Arquivo salvo com sucesso
        return true;
    }
    else
    {
        // Erro ao salvar arquivo
        return false;
    }
}
// Salva a àrvore para um arquivo, recursivamente, de nó
//  em nó, em pré-ordem, retornando false em caso de erro.
bool AvlTree::save(ofstream& stream, TreeNode* node)
{
    if (stream.is_open())
    {
        if (node != NULL)
        {
            // Salvamos os valores em pré-ordem;
            //  primeiro a palavra, depois a origem
            stream << node->Key   << endl;
            stream << node->Value << endl;
            // Caminhamos com a recursão
            this->save(stream, node->Subtree[LEFT]);
            this->save(stream, node->Subtree[RIGHT]);
        }
        // Nó salvo com sucesso
        return true;
    }
    // Algum erro ocorreu
```

```
    else return false;
}
```

Desenhando

E por último, mas não menos importante, aqui está uma classe (adaptada as pressas de um código C originally created by ponada) capaz de **desenhar a árvore** no console em arte ASCII. Muito interessante, e bastante engenhoso. Mas funciona que é uma beleza!

O código de desenho pode ser visualizado no fonte disponível na conclusão. Infelizmente, como não está suficientemente adaptada, então não detalharei o C++ código aqui. Já o código original, em C, pode ser obtido aqui.

Conclusão

Agora que já apresentamos todo o código, clique aqui para fazer download do código fonte completo deste artigo, juntamente com uma aplicação demonstrando o uso da árvore e seus métodos. O código é, na verdade, um exercício desenvolvido para a disciplina de Projeto e Análise de Algoritmos da Universidade Federal de São Carlos, ministrada durante o segundo semestre de 2008. O exercício visa demonstrar as propriedades de uma Árvore AVL e sua utilização como mecanismo de indexação. No caso, arquivos texto contendo palavras são lidas e armazenadas na árvore juntamente com o nome do arquivo de sua origem.

Finalizando, espero que este artigo tenha elucidado alguns conceitos do funcionamento de uma Árvore AVL! Caso algo tenha permanecido incompleto, se ainda restaram dúvidas quanto à implementação, ou se você simplesmente tem alguma sugestão a fazer, por favor deixe um comentário!

C#, Computer Science, Português, Programming, Tips

17 Comments

Felps

FEBRUARY 2, 2009 AT 6:19 PM

Aew cesinha da hora hauhauhsuahsa
Muito legal seu site:>

Bolívar

JUNE 30, 2009 AT 12:52 AM

Muito bom seu artigo, me ajudou bastante na construção dos meus algoritmos sobre árvores da faculdade, parabéns pela iniciativa!

Anonymous

SEPTEMBER 20, 2009 AT 11:03 PM

esse codigo parece que esta errado, num dos testes que fiz a sequencia 10,20,30,40,25,27,5 ele monta o 5 como filho direito do 40 na subarvores direita!!!

César Souza

SEPTEMBER 21, 2009 AT 11:24 AM

Você está inserindo estes números como strings? Pois, se estiver, a comparação de strings se dá por ordem alfabética, então realmente o número 5 deveria estar à direita do 40…

Se você quiser usar números, recomendo que adapte o código para utilizar templates. Apenas não utilizei templates no código original para evitar cobrir o código com detalhes de implementação.

[]’s,

Anonymous

SEPTEMBER 25, 2009 AT 12:26 AM

Valeu Cesar ,desculpe falta de atenção minha, realmente eu usei a sequencia como se fosse uma string. No trabalho que eu estou fazendo eu preciso converter essa strings em inteiros fazer calculos e converter de volta para string. Qual a melhor maneira de fazer isso?

Anonymous

NOVEMBER 9, 2010 AT 1:10 PM

Isto é em C e não em c++ como diz sua pagina..

César Souza

NOVEMBER 9, 2010 AT 1:27 PM

Isto o quê? O código disponibilizado nesta página é C++.

Até mais,
César

Emerson

DECEMBER 2, 2010 AT 8:39 PM

Tem como você colocar o código em outro link? Não consigo baixar.. tá dando TimeOut.

De qualquer forma, eu consegui fazer funcionar aqui copiando o código do post, mas queria ver aquele esquema de imprimir a arvore em ASCII.. uma vez que o site do autor também nao existe mais.

Parabéns e obrigado!

César Souza

DECEMBER 2, 2010 AT 8:48 PM

Sim, sim, o servidor da universidade parece estar com problemas, inacessível para algumas pessoas (inclusive pra mim).

Hosteei temporariamente no site de outro projeto, veja se este link agora funciona: http://accord-net.somee.com/Downloads/AvlTreeDemo.rar.

Até mais,
César

Emerson

DECEMBER 2, 2010 AT 9:10 PM

César,
muito obrigado!! Obrigado também pela resposta rápida.

Estou tentando fazer uma implementação teste de uma alternativa de persistência com Java e C++ (usando JNI).

Em Java, as árvores ficaram boas, porém com muitos dados (~20gb) em memória o garbage collector começa a atrapalhar tudo. Vou tentar fazer essas árvores em C++ usando seu código pra ver como fica.

valeu!!

Raui

DECEMBER 9, 2010 AT 10:56 AM

Voce tem o código em c??

se tiver seria possivel me arrumar?

tnk

César Souza

DECEMBER 14, 2010 AT 7:35 PM

Olá Raui,

O código em C é muito similar ao código C++. Tenho certeza de que não será tão difícil portá-lo.

[]s,

César

Danilo

NOVEMBER 26, 2011 AT 6:32 PM

O link de download tá fora.
Pode arrumar pra gente?

Valeu.

Anonymous

MAY 19, 2012 AT 8:18 PM

Tentei implementar com templates, mas aparentemente ele não ta inserindo todos elementos (bem, é o q diz minha funcão de impressão, pelo menos!)
pode me ajudar??

Vancley Simão

FEBRUARY 3, 2013 AT 5:32 PM

o link para download do arquivo esta quebrado, alguma outra forma de disponibilizar o código-fonte ? .-.

Obrigado.

Tassio

MAY 7, 2014 AT 1:42 PM

Olá tudo bem? O link para download não está funcionando, tem outro servidor que pode ser feito o download? Obrigado!

Pingback: Árvore B em Arquivo (em C++) | ~/cesarsouza/blog